National University of Singapore
School of Computing

CS2106 **Laboratory 1** Semester 1 11/12

MATRICULATION NUMBER: | | | | | | | | | |

In this lab exercise, you will get familiarize with some basic UNIX commands, editing and compiling C programs, as well as debugging C programs.

The exercise has been tested on SunFire. You should use SunFire for this exercise. If you try the exercise on Linux machines, the results may be slightly different (but should not impact your learning).

For SunFire server: To remotely access SunFire, use your favourite ssh client and ssh into `sunfire.comp.nus.edu.sg`. Use your SoC UNIX username and password to login. If you do not have an account on SunFire, apply for one here: `http://mysoc.comp.nus.edu.sg/~newacct`.

This is an *ungraded* lab exercise. Completing the exercise, however, will help you understand the concepts covered in the exercise better and will be much helpful for the subsequent labs. In particular, being familiar pointers and `gdb` will save you time in debugging your programs later.

You should complete the lab before 28 August 2011. We will discuss selected questions during the lab session of Week 4 (29 August 2011 - 2 Septemebr 2011).

Every lab exercise is an *individual* exercise.

Ideally, you should work on the lab exercise `before` the lab session. During the lab session, raise any difficulties you encounter or doubts that you have with the lab TAs. You may also discuss your solution with the lab TAs.

1. **Navigating and Manipulating Files and Directories**.

   You will learn to use the basic commands `ls`, `mkdir`, `rmdir`, `rm`, `cd`, `pwd`, `ln`, `cp`, `mv`.

   `man` is your best friend. Use the `man` command to find out what these commands do and what the available options are.

   (a) Use a combination of the above commands, and create the following file systems hierarchy under your home directory:

   ```
   cs2106/lab01/01/
   cs2106/lab01/02/
   cs2106/lab01/03/
   ```

   Note down the commands you used.

   .....

   .....

   .....

   .....

   (b) Under `cs2106/lab01/01/` directory, create two text files named `foo` and `bar` with an editor (you can use `vim` or other editors). You can put any text you want into the two files.

   Under `cs2106/lab01/02/` directory, using the command `ln`, create a *hard link* to the file `foo`, which you just created, and a *soft link* to the file `bar` that you just created.

   Note down the commands you used.

   .....

   .....

   .....

   .....

   (c) Make a copy of the files `foo` and `bar` to the folder `cs2106/lab01/03/`

   Note down the commands you used.

   .....

   .....

   .....

   .....

   (d) Now change the content of the files `foo` and `bar` under `cs2106/lab01/01` with a text editor, and save the files.

   Check the content of the files under `cs2106/lab01/02` and `cs2106/lab01/03`. What do you observe?

   .....

   .....

   .....

   .....

(e) Now, remove the directory `cs2106/lab01/01` (and its file content) completely. Check the content of the files under `cs2106/lab01/02` and `cs2106/lab01/03`. What do you observe?

.....

.....

.....

.....

(f) Copy the whole directory `cs2106/lab01/03` to `cs2106/lab01/01`. Note down the command you use.

.....

.....

.....

.....

(g) Check the content of the files under `cs2106/lab01/02`. What do you observe?

.....

.....

.....

.....

2. **Useful utilities, redirections, and pipes**

   Go through the examples given in the textbook Section 1.5.6 and 10.2.3, try them out on SunFire (modify them if necessary) to make sure you understand the notion of input redirection `<`, output redirection `>`, pipe `|`, and understand the purpose of utilities `sort`, `head`, `tail`, `wc`, and `grep`.

   `man` is your best friend. Use the `man` command to find out what the various commands do and what are the available options.

   (a) The command `who` lists the current users logging into SunFire. Using who, in combination of other commands using pipe, write a command that counts how many users are currently logged in in SunFire.

   .....

   .....

   .....

   .....

   (b) ($\star$) The command `ps` lists the current processes running on SunFire; the command `cut` extracts columns of text from a file; the command `uniq` remove consecutive lines that are duplicates. Using these commands, in combination with `wc`, `sort`, `grep` that you have seen in Section 1.5.6 and Section 10.2.3, construct a sequence of pipes that counts how many *unique* users are currently running `sshd` on SunFire. Assume that username are at most 8 characters in length.

   .....

   .....

   .....

   .....

3. **Compiling and Running C programs**

We now explore how to use `gcc`, the GNU C Compiler. You should first read Section 1.8.3 of the textbook if you have not done so.

You can do the rest of this lab exercise in any directory, but to keep things organized, I suggest that you do it under `cs2106/lab01/`.

You may find the man page for `gcc` useful.

Use the following command to download the program `hello.c` from the Web:

```
wget http://www.comp.nus.edu.sg/~ooiwt/cs2106/lab01/hello.c
```

The program is similar to what I shown in class.

Use the following command to generate an executable file from `hello.c`

```
gcc hello.c
```

This command should create a new executable file called `a.out`.

Run `a.out`, giving it a parameter, which is the number of times to print "Hello World." on the screen. Example:

```
./a.out 10
```

`a.out` is the default name of the executable created with `gcc`. You can specify the output executable file name with the `-o` option.

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

Now, you should see an executable file called `hello` in the same directory. You can run this file just like running `a.out`.

```
./hello 10
```

Now, let's explore the different stages of converting a C program into an executable: pre-processing, compiling, and linking.

(a) Run the command

```
gcc -E hello.c
```

The output is too long and scrolls too fast. You can either pipe the output of the command to a program called `less`:

```
gcc -E hello.c | less
```

or redirect the output to a file:

```
gcc -E hello.c > hello.out
```

and view the file `hello.out` using your favorite editor.

What do you see? What does the option `-E` mean? What does the pre-processor do?

.....

.....

.....

.....

(b) Run the command:

```
gcc -c hello.c
```

What file is created by this command? If you now run

```
gcc hello.o
```

what do you get?

.....

.....

.....

.....

(c) Comment out the `main( )` function in `hello.c` using `/*` and `*/` (but leave the function `say_hello( )` in there). Run again:

```
gcc hello.c
```

What error message do you see?

.....

.....

.....

.....

(d) Run

```
gcc -c hello.c
```

Any error message now? Explain the differences in the outputs you see above, before and after commenting out `main( )` and with and without `-c`.

.....

.....

.....

.....

4. **Debugging with gdb**. In this section, you will learn (i) the basic commands of a debugger `gdb`; (ii) what is a segmentation fault error. You will go through some steps that is typical in finding out pointer errors in your program with `gdb`.

   (a) Run the program `hello` without any argument.

   ```
   ./hello
   ```

   What do you get? What does this error message mean?

   .....

   .....

   .....

   .....

   (b) To find out what causes the error, we are going to use the debugger `gdb`. First, make sure that you have uncommented the function `main( )` which you have commented out in the previous question.

   Now, recompile `hello.c` with `-g` option to create an executable file with additional information for the debugger.

   ```
   gcc -g -o hello hello.c
   ```

   To run the debugger on the executable `hello`, run

   ```
   gdb hello
   ```

   You should see a prompt that says

   ```
   (gdb)
   ```

   You can now issue commands into `gdb` by typing on the prompt.

   The first command we are going to issue is `run`, or its abbreviation, `r`.

   ```
   (gdb) r
   ```

   (c) The debugger will now run `hello`. When a segmentation fault is received, the debugger will display where the error occurs. What is the name of the function within which segmentation fault occurs?

   .....

   .....

   .....

   .....

   (d) You might not recorgnize the function where the error occurs as it does not appear inside the code `hello.c` at all. Some of the function calls we made in `hello.c` must have lead to this function.

   Run

   ```
   (gdb) where
   ```

   to print out the stack frame. Which library function we call in `hello.c` causes the error?

   .....

   .....

   .....

   .....

(e) Now, lets trace through the code line-by-line, examining the variables to find out what went wrong.

To examine the variables while the program is running, we need to first "break" the program. To do this, we set the breakpoint at the function `main( )` with `b` command and rerun the program.

```
(gdb) b main
(gdb) r
```

The debugger will now stop at `main( )`. Let's examine the content of the variable `argc` and `argv` with the `print` command (abbreviated `p`).

```
(gdb) p argc
(gdb) p argv
```

Note down the output. What does the value `argv` means?

.....

.....

.....

.....

(f) The variable `argv` is an array of strings. Recall that each string in C is an array of `char`. Note down the ouput of the following expression:

```
(gdb) p argv[0]
(gdb) p argv[0][1]
```

What does the expression argv[0][1] refer to?

.....

.....

.....

.....

(g)
```
(gdb) p argv[1]
```

What do you get? Can you explain why running `hello` without command line argument leads to a segmentation fault error?

.....

.....

.....

.....

5. **Pointers**. Now, let's have some fun with pointers in C. Besides getting familiarize with the meaning of operator * and &, You will see from this exercise that: (i) C treats both value and address (`int` and `int *`) interchangably, (ii) with pointers you can modify the value of other variables unintentionally.

   Create a C program with the following code:

```
int main()
{
  int *x;
  int y;
  x = 0;
  y = 0;
}
```

   Compile the code with debugging option and load the resulting executable in a debugger.

   Now, tell the debugger to break at line number 5 (the closing } of the function `main()`) and run the program.

```
(gdb) b 5
(gdb) r
```

   (a) Use the debugger to print out the values of the following. What do you see? What do they mean?

x	
y	
*x	
*y	
&x	
&y	

   .....

   .....

   .....

   .....

   (b) Now, we are going to use the debugger "set" command to change the values of these variables.

```
(gdb) set x = &y
(gdb) set *x = 1
```

   Use the debugger to print out the values of the following. What has changed? Why?

x	
y	
*x	
*y	
&x	
&y	

.....

.....

.....

.....

(c) Now, run the following

```
(gdb) set *(x+1) = 10
```

Use the debugger to print out the values of the following. What has changed? Why? (Note: your answer might not be the intended answer if you use a machine other than SunFire).

x	
y	
*x	
*y	
&x	
&y	

.....

.....

.....

.....

(d) (⋆) Now run the following

```
(gdb) set x = &y
(gdb) set y = x
```

Think about how it will change the values of the expression below. Use the debugger to print out their values to see if you are right. What values do you see? Explain the changes.

x	
y	
*x	
*y	
&x	
&y	

.....

.....

.....

.....

(e) Now, exit from the debugger, change your C program to the following, and re-compile

```
int main()
{
  int *x = 0;
  int y = x;
}
```

You should get a warning message. What causes the warning?

.....

.....

.....

.....

(f) Now, recompile with the `-Wall` option of gcc. What is the option `-Wall` for?

.....

.....

.....

.....

(g) Edit the program to remove all the warning messages. Note down your new program below.

.....

.....

.....

.....

.....

.....

THE END