Problem Set 1: Unix Commands[1]

WARNING: IF YOU DO NOT FIND THIS PROBLEM SET TRIVIAL, I WOULD NOT RECOMMEND YOU TAKE THIS OFFERING OF 300 AS YOU DO NOT POSSESS THE REQUISITE BACKGROUND TO PASS THE COURSE.

- See course webpage for due date and time.
- Submit deliverables to CourSys: https://courses.cs.sfu.ca/
- Late penalty is 10% per calendar day (each 0 to 24 hour period past due).
  - Maximum 2 days late (20%)
- This assignment is to be done in **groups (see FAQ for group formation)**. Do not show another student your code, do not copy work found online, and do not post questions about the assignment online. Please email as per instructions in FAQ.
  You may use general ideas you find online and from others, but your solution must be your own.
- See the marking guide for details on how each part will be marked.

When asked to explain something, don't copy and paste text found online or in the man pages. Explain the idea in your own words.

FUTURE:
- Don't ask students to do *(x+1) = 10; different for 32 vs 64 bit? But forces students to think about pointers and realize they are tricky.... maybe good?
- Make clear students should look at output, but can consult man pages and internet. Remind them that they must answer in their own words.

1. Unix/Linux Commands

1.1 Readings

If not very familiar with Unix/Linux commands, read Chapter 4 and Sec 9.2 of **Running Linux** (Available as an Electronic Book through SFU Library).

1.2 Running Commands

After reading and practicing, create a text file named `Commands.txt` and in it write a one-line description for each of the following commands. I suggest you create a folder for CMPT 300, and sub folders for each assignment.

1. `xemacs &`
2. `cd`
3. `cat ~/.bashrc > tmpfile.txt`
4. `ln -s tmpfile.txt ~/tmp-alias`
5. `ls -al`
6. `chmod a+rwx tmpfile.txt`
7. `grep  bash /etc/passwd`
8. `ps -ef | more`

1 Some parts provided by Prof. Wei Tsang Ooi of University of Singapore.

9. `man 2 chown`
10. `gcc test.c 2> error-msg`

Suggestion: Use `xemacs` or `emacs` for typing your answers (for practice).

2. Compiling a C Program

In a file named `gcc_and_gdb.txt`, answer each of the questions in bold. List these answers as 2a), 2b), and so on, for question (a), and (b), and so on.

1. Copy the following code into a file named `hello.c`

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

void say_hello (int times) {
    for (int i=0; i < times; i++) {
      printf ("Hello World\n");
    }
}

int main (int argc, char *argv[]) {
    say_hello(atoi(argv[1]));
    return 0;
}
```

2. Compile the code as follows (the `$` indicates command entered in shell, don't actually type `$`):
   `$ gcc -std=c99 hello.c`
   - The `-std=c99` option enables the C99 extensions to C, which among other things includes declaring a variable inside the for-loop statement.
   - This should generate the executable `a.out`.
3. Run `a.out`, giving it the parameter 5:
   `$ ./a.out 5`
   - In Linux/UNIX, you cannot simply type "`a.out`" to run the executable because the current directory is not in the path where the OS looks for programs that match the entered name.
4. Instead of building to `a.out` (default executable's name), use the `-o` option:
   `$ gcc -std=c99 -o hello hello.c`
   - Run this executable with:
     `$ ./hello 5`

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

5. Run the command
   `$ gcc -E hello.c`
   - The outputs are too long and scrolls too fast. You can either pipe the output to `less`:
     `$ gcc -E hello.c | less`
   - Or redirect the output to a file and view `hello.out` in your favourite editor:
     `$ gcc -E hello.c > hello.out`

   (a) What does the -E option mean? What does the pre-processor do?
6. Run the command:
   `$ gcc -std=c99 -c hello.c`

(b) What file is created by this command? State its name and explain what it is.

7. Now run:
   ```
   $ gcc -std=c99 hello.o
   ```
   (c) What do you get?

8. Comment out the `main()` function in `hello.c` using `/*` and `*/` (but leave the function say `hello()` in there). Run again:
   ```
   $ gcc -std=c99 hello.c
   ```
   (d) What error message do you see? Briefly explain this error.

9. Run
   ```
   $ gcc -std=c99 -c hello.c
   ```
   (e) Any error message now? Explain the differences in the outputs you see above, before and after commenting out `main()` and with and without `-c`.

3. Debugging with gdb

1. Uncommented `main()` which you commented out in the previous question.
2. Run the program `hello` without any arguments:
   ```
   $ ./hello
   ```
   (a) What do you get? What does this error message mean?

3. To find out what causes the error, we are going to use the debugger gdb.
4. Recompile `hello.c` with the `-g` option to create an executable file with additional information for the debugger (so-called "debug information").
   ```
   $ gcc -g -std=c99 -o hello hello.c
   ```
5. To run the debugger on the executable `hello`, run
   ```
   $ gdb hello
   ```
   - You should see a number of lines of text. Te final line should be the gdb prompt:
     ```
     (gdb)
     ```
   - You can now issue commands into gdb by typing on the prompt.

6. The first command we are going to issue is `run`, or its abbreviation, `r`.
   ```
   (gdb) r
   ```
7. The debugger will now run `hello`. When a segmentation fault is received, the debugger will display where the error occurs.

   (b) What is the name of the function within which segmentation fault occurs?

   - You might not recognize 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.

8. To print the stack frame, run either of the following. `bt` is the abbreviation for backtrace.
   ```
   (gdb) where
   ```
   or
   ```
   (gdb) bt
   ```

(c) Which library function we call in `hello.c` causes the error?

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

9. 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
```
   - If asked, reply that yes, you do want to start the program from its beginning.

10. 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
```
(d) Record the output. What does the value `argv` means?

11. Run each of the following commands:
```
(gdb) info local
(gdb) info args
```
(e) What does each of these commands do?

12. The variable `argv` is an array of strings. Recall that each string in C is an array of char.

    Predict what each of the following will do, then run them.
```
(gdb) p argv[0]
(gdb) p argv[0][1]
```

13. Now run:
```
(gdb) p argv[1]
```
(f) What do you get? Explain why running hello without command line argument leads to a segmentation fault error?

14. You can quit gdb with the q command:
```
(gdb) q
```

4. Pointers in gdb

1. Create a C program with the following code (no #include's are needed):

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

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

2. Now, tell the debugger to break at line number 6 (the closing curly-brace of the function `main()`) and run the program. Then list the source code:
```
(gdb) b 6
(gdb) r
(gdb) list
```

(a) Use gdb to print out and record the values of the following: `x, y, *x, *y, &x, &y`. What do you see? What do they mean?

3. 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
```
(b) Use gdb to print out and record the value of any of the following which change: `x, y, *x, *y, &x, &y`. For each changed value, explain why?

4. Now, run the following
```
(gdb) set *(x+1) = 10
```
(c) Use gdb to print out and record the values of the following: `x, y, *x, *y, &x, &y`. What has changed? Why?

5. Now run the following
```
(gdb) set x = &y
(gdb) set y = x
```
- Predict what this will do. Hint, try forcing the print to display in hex by adding /x:
(gdb) p /x y

(d) Use gdb to print out and record the values of the following: `x, y, *x, *y, &x, &y`. What values do you see? Explain the changes.

6. Now, exit from the debugger, change your C program to the following, and re-compile
```
int main(void)
{
      int *x = 0;
      int y =  x;
      return 0;
}
```
7. You should get a warning message. Consider what causes the warning.
8. Now, recompile with the `-Wall` option of gcc.
(e) What is the option -Wall for?
9. Edit the program to remove all the warning messages.

5. Deliverables

Submit the following deliverabels to CourSys:

1. `Commands.txt`
2. `gcc_and_gdb.txt`

You do not need to submit your code.

Please remember that all submissions will automatically be compared for unexplainable similarities.