CS2106: Lab 1

1 Introduction to the Labs

If you have already used Unix before then most of the introduction below should be familiar and you can skip all the bits which you already know. For example, if you have used `sunfire` (by connecting to `sunfire.comp.nus.edu.sg`), then the main difference is that perhaps you may not have used a graphical user interface (GUI) with Unix.

1.1 Logging In

Your tutor will give you a user name and password for the machines in the lab which are running Ubuntu 16.04.1 (one of the Linux distributions).

You may also want to install Ubuntu on your own machine – the simplest is to run it in a Virtual Machine such as Virtual Box. The closest distribution to the labs is 16.04.5. See `ubuntu-setup.pdf` for a rough guide to install using Virtual Box on Windows.

1.2 User Interfaces

When you use Linux, there is the command line interface and also the graphical desktop interface. We will mostly talk about the command line since that is closer to the OS.

1.2.1 The Command Line Interface

Try starting a terminal window. The terminal you started is actually composed of two programs. One is a graphical application which gives a window which emulates a terminal. The other is a shell which interprets ASCII command lines from the terminal. The shell uses a *command line interface*, because it is text based. You type in commands to a *shell* which performs the command(s) in that line of text. In Windows, there is also a command line shell, `cmd` which has a similar purpose.

There are a number of possible shells available, `bash` (Bourne Again SHell) is the default and is a GNU enhancement descended from the original Unix shell, the Bourne shell (`sh`). A list of shells of typical shells in Unix are: `sh`, `bash`, `csh`, `ksh`, `tcsh`, `zsh`. (Not all these shells may be available). You can stick to the default choice of `bash` unless you feel like using another shell ;-)

1.3 Using `man`

You should try out the Unix manual command: `man`. For example, it is self documenting: (the '`$`' is meant to be the command-line shell prompt, you may see something else)

```
$ man man
```

Other commands which you may have tried are also in the manual page, eg. `man date`.

Take note that Unix man pages are written in a terse fashion. It is meant as a reference rather than a tutorial. An example of a long man page is the one for `gcc`.

The manual is in sections, and an entry might be in more than one section, e.g. `man 1 login` documents the command line `login` program, and `man 3 login` documents the files which record the users of the system. You can search for keywords with the `-k` option, eg. `man -k login`.

As the manual is divided into sections, the various sections have their own introduction. Try `man intro` or `man 1 intro`. System calls are described in `man 2 intro`. The introduction for library functions is in `man 3 intro`.

1.4 Listing Files and Directories

The `ls` command will list all the files in the specified directories. You can use the `-l` option for a long listing format which displays information like size, permissions, owner, group, creation date, etc. If you do not specify a directory, by default it uses the current directory. The `-R` option causes ls to list recursively. Some examples to try:

```
$ ls
$ ls -l
$ ls /
$ ls -l /
```

1.5 Editing Files

You can use various text editors either directly from the desktop or the terminal. Some editors are strictly text based and some have a graphical interface. Many editors are available such as: `emacs`, `vi`, `vim` (`gvim`, graphical), `gedit`, etc. Not all such editors may be installed. If you are using your own Ubuntu, `apt` can be used to install various software (see also the `ubuntu-setup` document). It is hard to recommend a specific editor but in general it is better to learn/use a more powerful editor.

1.6 Logging Out

After you have finished, you `MUST` log out from Linux. Make sure you have saved any files first. Choose logout from the top right-hand icon.

1.7 Backup and Transferring Files

Please see some of the options in `ubuntu-setup.pdf`.

1.8 Some Things to Remember!

The following points are to be noted for Unix in general and also Linux:

- Unix is case sensitive. Most commands are lowercase.
- Unlike Windows, Unix has no drive letters. Everything is in some directory.
- Unix uses forward slash (`/`) to separate directory names, while Windows uses backslash (\).
- The `*` wildcard is treated uniformly by Unix shells. In Windows, `*` works differently for different programs.
- It is worthwhile to look first at the man page of a command
- Try using various GUI and command line programs. Some programs need administrator privileges (in Unix, this is the `root` user who has *superuser* privileges) to run.

Happy Linux-ing!

2 Sample Usage Session

Skip if you already know how to use Unix.

1. `mkdir junk` — makes a new directory `junk`
2. `cd junk` — change directory to it, you are now in `junk`
3. `pwd` — prints the path to your current directory (Print Working Directory)
4. `echo abcd > test1.txt` — makes a new file `test1.txt` with contents "`abcd`"
5. `less test1.txt` — display `test1.txt` using the pager. '`q`' will quit from less and '`h`' will give the help screen.

Select an editor to use (see previous editing section).

1. edit the file test1.txt, eg. `gedit test1.txt`
2. change "`abcd`" to something else
3. save the file in the editor (this is editor specific)
4. `cat test1.txt` — concatenate 1 or more files to the output, so this displays `test1.txt` because output is the terminal
5. `cat test1.txt test1.txt >test2.txt` — `test1.txt` is concatenated twice and the output saved to `test2.txt`
6. `cp test1.txt test3.txt` — copies `test1.txt` to `test3.txt`
7. `ls` — listing shows the 3 files
8. `rm test1.txt` — deletes `test1.txt`
9. `ls` — only `test2.txt` and `test3.txt`
10. `rmdir ../junk` — try to remove directory, complains about non-empty directory
11. `rm test*.txt; cd .. ; rmdir junk` — no more files in directory, go up to parent directory, remove junk directory
12. `logout` — you can use the `System` menu in the GUI
13. Login. Read the man pages of all of the above

3 Lab 1: C Exercises

3.1 Purpose

These exercises are to gain familiarity with writing some simple C programs, compiling, execution and debugging. It is meant to be straightforward. **Only Exercise 4 and 5 are graded** and to be submitted. If you are familiar with C programming, you can skip to Exercise 4.

3.2 Exercise 1: Hello (not graded)

The purpose of this simple exercise is simply to be able to edit, modify and compile a simple program.

Modify the program `lab1-hello.c` so that It then prints "Hello" followed by n exclamation marks and a newline. Eg. if the number was 3, then your program would print out

```
Hello!!!
```

It already reads in n and prints "Hello". You should first compile and run `lab1-hello.c` before modifying it so that you know how to use the C compiler.

Notes: Notice how the input is read into `n`, i.e. how `n` is passed to `scanf()`.

To compile and then run: (assumes no errors from gcc)

```
gcc -o lab1-hello lab1-hello.c
./lab1-hello
```

The "`-o`" option tells `gcc` to create the executable file `lab1-ex1`. You may not need the "`./`", it will depend on your shell settings. (If you use `bash` the setting is in the `PATH` environment variable, but do not worry if you do not understand `PATH` for now).

3.3 Exercise 2: String Length (not graded)

The purpose of this exercise is to show some code using strings and pointers in C. Look at, `lab1-len.c` written in a common C style (C coding style and source code formatting is quite typical). You should understand what the code is doing. It uses arguments to the process, as covered in Unix Processes in lectures. Compile and test it.

3.4 Exercise 3: Mystery Function (not graded)

C has bit manipulation operators. Bit manipulation is useful for low level code which might have to look at the actual bit representation and change it. The purpose of this exercise is to look at bit manipulation and also play with recursion:

- $expr_1$ `<<` $expr_2$: shifts $expr_1$ left by the number of bits given in $expr_2$. Shift one bit left means that the least significant (the rightmost) bit becomes the second least significant bit and with a 0 replacing that rightmost bit. Eg. `(2 << 1)` $\equiv 10_2$ `<<` 1 becomes 4 (100_2). Excess bits which are shifted off to the left are discarded.

- $expr_1$ `>>` $expr_2$: shifts $expr_1$ right by the number of bits given in $expr_2$. For an unsigned integer, 0 bits are shifted in from the left to replace the high bits. For signed integers, the behavior of the sign bit (leftmost bit) is implementation dependent in C — 0 bits are shifted in but the leftmost sign bit may or may not be preserved.

  (Notes: Java is similar but has 2 versions, sign bit preservation (`>>`) or all 0-bits (`>>>`)).

The C code below is for a `mystery` function:

```
unsigned int mystery(unsigned int n)
{
    unsigned int x;
    if (n == 0)
       return 0;
    if (n & 1) x = 2;
    else x = 0;
    return x + (mystery(n >> 1) << 1);
}
```

3.4.1 Part 3A: Separate Compilation

We first investigate how to do separate compilation. Create a file, `lab1-mystery.c`, containing only the mystery function. (This tests your use of an editor). The main program is in `lab1-ex3.c` which reads an integer n and prints out `mystery(`n`)`.

Note: Note the use of a function prototype in `lab1-ex3.c`:

```
extern unsigned int mystery(unsigned int);
```

This is because in the main program, `mystery()` is used but the code is not declared. The function prototype is used to declare its return value and arguments.

To compile and then run:

```
$ gcc -c lab1-mystery.c
$ gcc -c lab1-ex3.c
$ gcc -o lab1-ex3 lab1-mystery.o lab1-ex3.o
```

The first two commands are to use `gcc` to compile and create the object code (.o files) from your C files. The command uses `gcc` as the linker to combine the object files into an executable program, `lab1-ex3`. You can run the executable created.

3.4.2 Part 3B: Investigating Local Variables

You know that mystery is recursive therefore there are many instances of the local variable `x` for each invocation of the function. How can you modify the code to show that there are various *instances* of `x` when mystery is run?

Remark:
This exercise links up with our coverage of stack frames.

3.4.3 Part 3C: Exploring/Debugging Mystery

Here we want to first understand how to use the debugger to examine the execution of a program. To use the debugger you should compile your program with the `-g` option, eg. `gcc -g -c lab1-mystery.c`. The `-g` option tells the compiler to produce extra debugging information into the object/executable file which is then used by the debugger. (Use steps similar to 3A with "`-g`" to create the `lab1ex3` executable).

There are several debuggers available. We will use `gdb` the GNU debugger. To use the debugger, instead of running your program directly, you run it under the control of the debugger,

```
$ gdb lab1ex3
```

A partial list of debugger commands are: (abbreviations for the commands is also give)

- `help` (h): help on various commands
- `run` (r): run the program, you can pass arguments the usual way on the command line, eg. `r arg1 arg2`
- `print` (p): print an expression, eg. `p x` prints the variable `x` in mystery() provided you are inside the function. you can print either be a variable or any expression.
- `break` (b): set a breakpoint - a running program will execute until it comes to specified breakpoint when it stops and control goes back to the debugger. Breakpoints could be a line number or a function, eg. `b mystery` will stop execution whenever mystery() is entered into.
- `step` (s): steps execution to the next source line. If the statement contains a function, then steps into the function.
- `next` (n): like step but treats a function call like one instruction so doesn't stop inside the function.
- `continue` (c): continue running after breakpoint. contains a function, then steps into the function.
- `list` (l): list source code, can specify line or function, eg. list main. Without arguments, lists source code around current line.
- `backtrace` (bt): prints the stack frames (see the lecture in *comporg*), i.e. it shows you what functions have been called to reach this point including the arguments.

More documentation on `gdb` can be found with the `info` command, e.g. `info gdb`. (Note that relies on the particular info page being present).

Try the following: (commands typed are indicated by the boxed text)

1. `gdb lab1ex3`: start gdb
2. `b mystery`: set a breakpoint in mystery(), note that the current source line in C is printed
3. `r`: runs `lab1ex3`
4. execution of `lab1ex3` stops at start of mystery()
5. `l`: list program source
6. `p n`: prints argument to mystery()
7. `bt`: you can see that main() calls mystery()
8. step a few times
9. `c`: continue running, will stop at next call to mystery (assumes initial n is big enough)
10. continue for a few recursive calls and then do a `bt`. You can see clearly the recursion which is going on. This is another way of answering part 3B.
11. `q`: (q)uit the debugger and executing program

You should now have a feel of using the debugger. In general, a combination of output printing as well as using the debugger is useful for finding and fixing bugs.

The debugger is also useful for examining memory including looking at the stack. In this exercise, using the debugger will be one way of understanding how the `mystery()` function works.

3.5 Exercise 4: Basic Palindrome (Graded: 70% of mark)

A *palindrome* is a string which when reversed is the same. For example, "`ABBA`" is a palindrome as reversing it gives the same result. However, "`ABBB`" is not a palindrome as it's is "`BBBA`". A classic palindrome is: "`able was i ere i saw elba`".[1] Your task is to write the function:

```
int check_palindrome(char *s);
```

It returns 1 if the string `s` is a palindrome and returns 0 otherwise. Write your function in the file `lab1-checkpal.c`. Note that the file should **only** contain the `check_palindrome` function. It is called from the file `lab1-ex4.c` which illustrates how to read from *standard input* (called `stdin`) in Unix. (A Unix program has a notion of 3 kinds of input-output streams: standard input (for reading input), standard output, standard error).

The `lab1-ex4.c` illustrates the use of `malloc(int` n`)` which requests n bytes of memory from the OS. This comes from the process heap which we will cover in the Memory Unit. For now, just consider it as a form of object creation.

Note that `lab1-ex4.c` is not very robust and assumes that the input is always less than 128 characters. Furthermore, it does not do any form of error checking, but the code is only meant to be test `check_palindrome()`. Intentionally, Exercise 4 is meant to be straightforward as a basic graded exercise to write code with C pointers and C strings.

3.6 Exercise 5: Extended Palindrome (Graded: 30% of mark)

Your task is to write the function:

```
int read_palindrome(); // input comes from stdin
```

which will read one line from standard input and returns 1 if the line is a palindrome and 0 otherwise. A line is terminated by the newline character ('`\n`') and the does not include the newline. The file `lab1-readpal.c` contains your `read_palindrome` function which will be called by the main program in `lab1-ex5.c`.

An important difference from Exercise 4 is that there is **no** assumption about the length of the input. You are also **not** allowed to read the input **twice**, e.g. read the input, forget you read the input but remember the length, read the input again. which results in the input being read twice. You are also not allowed to create a very large buffer to store the input reasoning that the input line might be expected to be smaller than a very large buffer. The reason for this restriction is that we will consider the memory usage of the program. So a constant sized large buffer means that memory is wasted if the line is short. Your program, `lab1-readpal.c`, will be graded on correctness, CPU time and memory usage. You can obtain runtime statistics on a process as follows:

```
$ /usr/bin/time --verbose ./a.out
```

This will print various (accounting) statistics about the process which is executing the `a.out` binary when it terminates. The memory statistic to use is "*Maximum resident set size*" in KB. Note that Exercise 5 is harder than Exercise 4, there are many implementation strategies. Given the criteria, you should do your own testing for CPU time and memory usage on long strings.

You should note that as your code is contained in `read_palindrome` and any other associated functions, it can be called from any `main` program. Our test code will use a different `main` from `lab1-ex5.c`.

[1] The story is that when Napoleon Bonaparte was exiled to the island of Elba, he uttered "`Able was I ere I saw Elba`". We have made the string to be lower-case to be a strict palindrome. Some definitions of palindrome to cater to English ignore case, punctuation, etc. but our definition is for strict palindromes.

4 Submission Instructions

Please follow the submission intructions to make grading easier. Suppose your name is "Dennis Ritchie"[2] and your student ID is U123Y. Given a filename such as `ex8.c`, then you should rename it to `ritchie-U123Y-ex8.c` to make the filename associated with your own student information. The start of every C file should have a comment block giving the information below, e.g.:

```
/*
 * Name: Dennis Ritchie
 * ID: U123Y
 * file: ex8.c
 */
```

Submit a single **zip** file (**do not submit other file types**, e.g. arj) using the above format, e.g. `ritchie-U123Y-lab1.zip` containing the two files for Exercise 4 and 5 with filenames in the same style. In addition, there should be a comment block in Exercise 5 (`lab1-readpal.c`) to explain how your code deals with the unknown line length. Be careful that Exercise 4 and 5 files should not have `main`.

[2] `https://en.wikipedia.org/wiki/Dennis_Ritchie`