CS351 Lab #1

Linux Server and C

Instructions:

- *Assigned date: Thursday January 27th, 2022*
- *Due date: 11:59PM on Monday January 31st, 2022*
- *Maximum Points: 5*
- *This lab must be done individually*
- *Please post your questions to BB*
- *Only a softcopy submission is required; it will automatically be collected through GIT at the deadline; email confirmation will be sent to your HAWK email address; late submission will be penalized at 10% per day; your submission will not be graded until you have submitted a single page PDF file with your name, email, A#, assignment number, and URL to the gitlab repo*

1 Your Assignment

Objectives

1. Log in to course server (fourier.cs.iit.edu) with SSH
2. Learn to use a terminal multiplexer (tmux)
3. Clone your private course repository from GitLab
4. Commit changes to your local repository
5. Push changes to your private repository
6. Edit files on fourier.cs.iit.edu
7. Compile C code with `gcc`
8. Use make to run tasks
9. Understand the basics of how the make build tool works

Overview

You'll write plenty of code this semester, but before you get to it you first need to learn how to edit, test, and submit your work. We'll cover those steps here.

To keep this writeup short we'll do lots of interactive demonstrations, but the critical steps and commands will be documented for future reference.

`fourier.cs.iit.edu`

At this point you should've received an e-mail containing login credentials for `fourier.cs.iit.edu`, a CS department Linux server on which you will be able to implement and test your assignments on for this class. Using fourier.cs.iit.edu will ensure a consistent working environment, and we will be using a comparable system to evaluate your submissions. If you're interested in working on your own Linux

machine (or virtual machine), this is possible for many of the labs – but please make sure to you're your assignments prior to the deadline on the course linux server fourier.cs.iit.edu.

To log in to fourier.cs.iit.edu, you'll need an SSH client. At a terminal on most computers running Linux, macOS, or Windows 10, you can do this with the command:

```
ssh username@fourier.cs.iit.edu
```

On Windows you may need to download a separate SSH client such as PuTTY if you don't have a command line client.

After logging in, resist the overwhelming urge to immediately start being productive! Repeat after me: TERMINAL MULTIPLEXERS ARE AWESOME. You'll learn about them next.

tmux

MPs require multiple coding, testing, and debugging sessions to complete, which require multiple corresponding login sessions on fourier.cs.iit.edu. Each time you login, you'll have to navigate into the appropriate work directory, re-open source file(s) in editors, start up debugging sessions, run trace files to obtain output, etc. All before you can even *think* about getting work done.

Ugh.

All the above constitute annoying *cognitive overhead*, and we can all use less of that (especially given how difficult it is for most students to allocate time to sit down and work on coursework in the first place).

Wouldn't it be great if you could set things up --- your editing windows, debuggers, trace output position --- *just the way you like it*, ***one time***, then have it remain that way each time you log back in? Heck yeah. That's where terminal multiplexers fit in.

Technically, a terminal multiplexer is a program that:

1. Allows you to create and manage multiple terminal sessions from a single screen. You can run a different program in each session (e.g., editor, debugger, shell).
2. Continues to run after you log out, so when you reconnect you can simply pick up where you left off.

Less technically, a terminal multiplexer = bliss.

To start tmux, just use the command `tmux`. I'll walk you through a few demo sessions, but the important keys are as follows:

- `C-b`: default prefix key (prefixes all following command keys)
- **c: create a new window**
- **n/p: change to the next/previous window**
- `"`: split the current window
- `<space>`: arrange the panes in the current window according to some preset
- `o`: change to the next pane in the current window
- `!`: break the current pane out of the current window
- `?`: list all key bindings
- **d: detach from tmux**

After detaching from a session, the following command will re-attach you to it (given that you only have one tmux session running):

```
tmux at
```

You should **NOT** create a new tmux session (with the command `tmux`) each time you log in. Instead, you need only do this once --- on subsequent logins you will simply reattach to your existing session with `tmux at`.

So... Logged in: check. tmux session attached: check. Moving on.

Git / GitLab

Git is a distributed version control system, and GitLab is a Git repository hosting service. You'll use the first to manage all your code and the second to share it with me.

Start by claiming your private repository on GitLab via the invitation link. You'll be prompted to create an account on GitLab (or sign in, if you already have one), and then you'll be asked to pick your username from a list --- please locate and select your IIT Hawk ID and accept the assignment. When the process is complete, you should be taken to a URL that looks something like this (where *ID* is your assigned ID):

```
https://gitlab.com/cs351-spring2022/student-<ID>
```

This is your repository's homepage on GitLab. You can always come back here to see the status of your work as reflected on GitLab. When we ask you to submit work you will push it here, and we will pull work from this GitLab repository for grading. Remember, if your work isn't here we can't see it!

Next, you should clone your repository on fourier.cs.iit.edu so you can work on and make changes to it. On your repository's GitLab page, you should see a green button labeled "Clone or download". Click it and copy the URL in the textfield.

When logged in to fourier.cs.iit.edu, run the following command, replacing the URL with the one you just copied:

```
git clone https://gitlab.com/cs351-spring2022/student-<ID>.git
```

This will prompt you for your GitLab username and password. If successful, you will see output like the following:

```
remote: Enumerating objects: 151, done.
remote: Counting objects: 100% (151/151), done.
remote: Compressing objects: 100% (103/103), done.
remote: Total 151 (delta 6), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (151/151), 3.59 MiB | 3.10 MiB/s, done.
Resolving deltas: 100% (6/6), done.
Checking connectivity... done.
```

Editing files

A good programmer has strong opinions about programming languages, coding conventions, and text editors. There are only two UNIX text editors worth learning: Emacs and Vim. Both are installed on fourier.cs.iit.edu. Pick one and learn a new feature every day.

You can also use an editor or IDE on your own computer to edit files remotely on fourier.cs.iit.edu. Editors/IDEs such as Atom and Visual Studio Code have good support for such workflows, though they are also supported in Emacs and Vim.

Signing your project README

Open and read the "README.md" file located in the root of your repository. You'll find an honor pledge at the bottom of the file we ask you to sign with your name and e-mail address. Please do so and save your changes.

When done, you can commit your changes to the repository with the following command:

```
git commit -am "Signing honor pledge"
```

What this command does is save a record of your changes to the repository, annotated with the message in double quotes. Everytime you make a substantive set of changes to one or more files in your repository you should commit your work with a short, descriptive commit message.

One great thing about commits is that you can easily compare the differences between them (or between a commit and the current state of your files, known as the "working tree"). This can be a real lifesaver if you accidentally introduce a bug into your code and want to see what you changed since the last commit, or even just roll back all your changes entirely.

Commits are not automatically sent to GitLab, however. To do that, you need to run this next command:

```
git push
```

After entering your GitLab username and password, you should see output that looks like this:

```
Counting objects: 5, done.
Delta compression using up to 12 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 318 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/cs351/cs351-mp-iitjohndoe.git
    b0bf469..9b59620  master -> master
```

This tells you that all your commits have now been synchronized with your repository on GitLab, and we can pull them for grading. In the future, you'll need to push your commits whenever you're done with your work on a given machine problem and want to make your work available to us.

Building

You'll write code in the next machine problem. Promise. This time we'll skip that part and jump to the fun parts: building and running.

For the rest of this machine problem you'll work in the "lab1" subdirectory of your repository. In there, you'll find these files:

- "Makefile": more on this later
- "hello.h" and "hello.c": files that declare and define the `say_hello_to` API
- "main.c": contains the definition of the `main` function

Let's try to compile "main.c" (the `$` indicates the shell prompt, and is followed by the command you should enter):

```
$ gcc main.c
/tmp/ccbmZeXz.o: In function `main':
main.c:(.text+0x24): undefined reference to `say_hello_to'
main.c:(.text+0x30): undefined reference to `say_hello_to'
collect2: ld returned 1 exit status
```

Didn't work. (Why not?)

Try compiling "hello.c":

```
$ gcc hello.c
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/crt1.o: In
function `_start':
(.text+0x20): undefined reference to `main'
collect2: ld returned 1 exit status
```

What gives?

Now try:

```
$ gcc hello.c main.c
```

Good. No errors. The compiler created an executable for you named "a.out", by default. Run it:

```
$ ./a.out
Hello world!
$ ./a.out Master
Hello Master!
```

We can of course rename the executable, but let's have the compiler put it in the right place for us:

```
$ rm a.out
$ gcc -o hello hello.c main.c
$ ./hello Overlord
Hello Overlord!
```

But what about the multi-stage compilation and linking process discussed in class? This is actually going on behind the scenes already (try invoking gcc with the `-v` flag to see what it's doing). To build the intermediate object files and link them together in separate steps, do:

```
$ gcc -c hello.c
$ gcc -c main.c
$ gcc -o hello hello.o main.o
```

See how we're referring to the ".o" files in the third step? Those are the object files that were generated when we invoked gcc with the `-c` flag, which tells it to stop before the linking step.

If the project being built is complex enough, it may be necessary to separate the final linking step from the creation of a multitude of intermediate object files. Manually invoking the compiler in such situations is a real pain, and definitely not something we'd want to keep doing!

Enter the standard C build tool: **make**

make

`make` is used to automate builds. Try it out (first, we delete the files we previously built):

```
$ rm -f *.o hello
$ make
gcc -g -Wall -O2   -c -o hello.o hello.c
gcc -g -Wall -O2   -c -o main.o main.c
gcc -g -Wall -O2 -o hello hello.o main.o
```

Woohoo! There's automation for you! Try it again:

```
$ make
make: Nothing to be done for `all'.
```

`make` knows that no files have changed since we last build the executable, see. We can update the timestamp of one of the files (using `touch`) to force a rebuild:

```
$ touch hello.c
$ make
gcc -g -Wall -O2   -c -o hello.o hello.c
gcc -g -Wall -O2 -o hello hello.o main.o
```

See how it only rebuilds one of the intermediate object files? Pretty nifty.

We can also ask it to run a test for us, then clean up generated files:

```
$ make test
```

```
Running test...
./hello tester
Hello tester!
$ make clean
rm -f hello.o main.o  hello
```

Makefiles

`make` is not magical, of course --- it makes use of the provide "Makefile" to determine what action(s) to take, depending on the specified target. For reference, here are the contents of said Makefile. Note the four targets (`all`, `hello`, `test`, and `clean`), two of which we invoked explicitly before --- the first target (`all`), it turns out, is used by default when we ran the `make` command with no argument.

```
CC      = gcc
CFLAGS  = -g -Wall -O2
SRCS    = hello.c main.c
OBJS    = $(SRCS:.c=.o)

all: hello

hello: $(OBJS)
    $(CC) $(CFLAGS) -o hello $(OBJS)

test: hello
    @echo "Running test..."
    ./hello tester

clean:
    rm -f $(OBJS) hello
```

We'll go over as much of this during our lab demo as we can, but you should check out the GNU `make` manual for details --- at the very least, skim through the Introduction.

2 What you will submit

To earn the points for this machine problem, you must have **signed and committed your README file and pushed your changes to the GitLab repository**. The procedure for doing this is similar in most future labs, so be sure you know how to do this!

Procedural takeways:

1. Commit often (`git commit -am "Commit message"`), and push to submit (`git push`)
2. `make` to build, and often times some variation on `make test` to run a hardcoded test.

Your solution will be collected automatically at the deadline. If you want to submit your homework later, you will have to push your final version to your GIT repository and you will have let the TA know of it through email. There is no need to submit anything on BB for this assignment. If you cannot access your repository contact the TAs. You can find a git cheat sheet here: https://www.git-tower.com/blog/git-cheat-sheet/

When you have finished implementing the complete assignment as described above, you should submit your solution to your private git repository. You should hand in:

1. **Readme:** A plain text file that includes the history of all your commands from fourier.cs.iit.edu. This should be included as history-fourier.cs.iit.edu.txt in the "lab0" folder. Don't forget to add the file to the repo, and then commit the changes.

```
$ export HISTTIMEFORMAT='%F %T '
$ history > history-fourier.txt
```

Submit code/report through GIT.

Grades for late programs will be lowered 10% per day late.