Machine problem 1: Preliminaries

Objectives

1. Log in to fourier (the course server) with SSH
2. Learn to use a terminal multiplexer (tmux)
3. Clone the public, read-only course repository from Bitbucket
4. Obtain write access to our shared, private repository
5. Commit changes to your local repository
6. Push changes to your private repository
7. Pull/Merge updates from the public repository
8. Edit files on fourier
9. Compile C code with `gcc`
10. Use make to run tasks
11. 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 work on machine problems for this class. Using fourier will ensure a consistent working environment, and we will be using a comparable system to evaluate your submissions.

Before trying to log in to fourier, you will first need to connect to the IIT VPN (if you are connected to the IIT network directly, you can skip this step). You can find instructions for connecting to the IIT VPN at https://ots.iit.edu/network-infrastructure/vpn-remote-access. Note that you will need to do this every time you wish to work on fourier remotely.

After connecting to the VPN, to log in to fourier 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.

Starter code repository

Each machine problem will require you to access and clone a separate, private code repository containing the starter codebase, implement and test your solution on your own machine, then push any changes you make for us to evaluate.

You will claim your private repository using a GitHub invitation link -- these will always be found next to the assignment writeup links on the course website. After following the link, you'll be prompted to create an account on GitHub (or sign in, if you already have one) and pick your IIT username from a list to accept the assignment. GitHub will then clone the starter code into a private repository and take you to a URL that looks something like this https://github.com/cs351/mp-prelim-USER (where *USER* is your own GitHub username).

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

Next, you will clone your repository on fourier so you can work on and make changes to it.

To do this, you must first register your SSH key with GitHub. You can find your SSH key in the file "~/.ssh/id_rsa.pub" when logged into Fourier. You should copy its contents and paste them into a new SSH key entry in the "SSH and GPG keys" section of the GitHub settings area. If you need help doing this, please ask a TA!

On your repository's Github page, you should see a green button labeled "Code". Click it, then click on the "SSH" label, then copy the URL in the textfield, which should look something like "git@github.com:cs351/mp-prelim-USER.git"

Now run the following command, replacing the URL with the one you just copied:

```
git clone git@github.com:cs351/mp-prelim-USER.git
```

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.
```

The command will also create a folder named "mp-prelim-USER", in which you will find all the files for this lab.

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. 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. Editors/IDEs such as Atom and Visual Studio Code have good support for such workflows. Feel free to reach out for help setting them up.

Signing your project header

Open and read the "main.c" file located in your newly cloned repository. You'll find a header comment at the top, which you should edit and replace with your own name, IIT email address, AID, and today's date. After doing this, 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 Github, however. To do that, you need to run this next command:

```
git push
```

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/mp-prelim-USER.git
    b0bf469..9b59620  master -> master
```

This tells you that all your commits have now been synchronized with your repository on Github, and we can pull them for grading. In the future, you'll need to push your commits whenever you're done working 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 "00-prelim" 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.

Finishing up

To earn the points for this machine problem, you must have **signed and committed your "main.c" file and pushed your changes to the Github 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.

Last updated: Fri Sep 9 19:23:15 2022