HW 0: **Introduction to CS 162**

CS 162

Due: January 31, 2020

Contents

This semester, you will be using various tools in order to submit, build, and debug your code. This assignment is designed to help you get up to speed on some of these tools.
This assignment is due at 11:59 pm on 1/31/2020.

1 Setup

1.1 GitHub and the Autograder

Code submission for all projects and homework in the class will be handled via GitHub so you will need a GitHub account. We will provide you with private repositories for all your projects. **You must not use your own public repositories for storing your code. Throughout the course, if you discover repositories with CS 162 solutions, please notify the course staff. Using solutions you may discover on-line is not permitted. Seek course staff for help. What you turn in should reflect your work.** Visit cs162.eecs.berkeley.edu/autograder[1] to register your GitHub account with the autograder.

1.2 Vagrant

We have prepared a Vagrant virtual machine image that is pre-configured with all the tools necessary to run and test your code for this class. Vagrant is an tool for managing virtual machines. You can use Vagrant to download and run the virtual machine image we have prepared for this course. (The virtual machine for the course is new this term. Do not use one from a previous semester.)

Note: If you do not want to set up Vagrant on your own machine, take a look at the CS162 VM provisioner[2] on GitHub for more options. You can run the VM on a variety of hypervisors, cloud computing platforms, or even on bare metal hardware.

(If you are using Windows, these steps may or may not work. If they do work, you should be fine. If they don't work—which is likely to happen if you have an older version of Windows that doesn't support SSH on the command line—then skip to the section below labeled "Windows").

1. Vagrant depends on VirtualBox (an open source virtualization product) so first you will need to download and install the latest version from the VirtualBox website[3]. **We have observed that certain earlier versions of VirtualBox, specifically versions 6.0.0 to 6.0.4, do not properly boot our class VM. We recommend using version 6.0.10, the latest version of VirtualBox at the time of writing.** We will talk in class about virtual machines, but you can think of it as a software version of actual hardware.

2. Now install the latest version of Vagrant from the Vagrant website[4].

3. Once Vagrant is installed, type the following into your terminal:

```
$ mkdir cs162-vm
$ cd cs162-vm
$ vagrant init cs162/spring2020
$ vagrant up
$ vagrant ssh
```

[1] https://cs162.eecs.berkeley.edu/autograder
[2] https://github.com/Berkeley-CS162/vagrant/
[3] https://www.virtualbox.org/wiki/Downloads
[4] http://www.vagrantup.com/downloads.html

These commands will download our virtual machine image from our server and start a ssh session. The "up" command will take a while, and may require an Internet connection.

4. You need to run all vagrant commands from the cs162-vm directory you created earlier. Do NOT delete that directory, or vagrant will not know how to manage the VM you created.

5. You can run `vagrant halt` to stop the virtual machine. If this command does not work, make sure you are running it from your host machine, not inside SSH. To start the virtual machine the next time, you only need to run `vagrant up` and `vagrant ssh`. All of the other steps do not need to be repeated.

1.2.1 Windows (OS X and Linux users can skip this section)

Your Windows installation may not support SSH from the command line, especially if you do not have the latest version. In this case, the "vagrant ssh" command from the above steps will cause an error message prompting you to download Cygwin or something similar that supports an ssh client. Here[5] is a good guide on setting up Vagrant with Cygwin in windows.

Alternatively, it is possible to use PuTTY instead of Cygwin, but this might be slightly more work to set up.

If you get an error about your VM bootup timing out, you may need to enable VT-x (virtualization) on your CPU in BIOS.

1.2.2 Troubleshooting Vagrant

If "`vagrant up`" fails, try running "`vagrant provision`" and see if it fixes things. As a last resort, you can run "`vagrant destroy`" to destroy the VM. Then, start over with "`vagrant up`".

1.2.3 Git Name and Email

Run these commands to set up your Name and Email that will be used for your Git commits. Make sure to replace "Your Name" and "your_email@berkeley.edu" with your REAL name and REAL email.

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your_email@berkeley.edu"
```

1.2.4 ssh-keys

You will need to setup your ssh keys in order to authenticate with GitHub from your VM.

New GitHub Users

SSH into your VM and run the following:

```
$ ssh-keygen -N "" -f ~/.ssh/id_rsa
$ cat ~/.ssh/id_rsa.pub
```

The first command created a new SSH keypair. The second command displayed the public key on your screen. You should log in to GitHub and go to github.com/settings/ssh[6] to add this SSH public key to your GitHub account. The title of your SSH keypair can be "CS 162 VM". The key should start with "ssh-rsa" and end with "vagrant@development".

[5] https://gist.github.com/rogerhub/456ae31427aafe5b70f7
[6] https://github.com/settings/ssh

Experienced GitHub Users

If you already have a GitHub SSH keypair set up on your local machine, you can use your local ssh-agent to utilize your local credentials within the virtual machine via ssh agent forwarding. Simply use `vagrant ssh` to ssh into your machine. The Vagrant should enable SSH agent forwarding automatically. If this doesn't work, you can also use the instructions in the previous "New GitHub Users" section.

1.2.5 Repos

You will have access to two private repositories in this course: a personal repository for homework, and a group repository for projects. We will publish skeleton code for homeworks in Berkeley-CS162/student0[7] and we will publish skeleton code for group projects in Berkeley-CS162/group0[8]. These two skeleton code repositories are already checked out in the home folder of your VM, inside `~/code/personal` and `~/code/group`.

You will use the "Remotes" feature of Git to pull code from our skeleton repos (when we release new skeleton code) and push code to your personal and group repos (when you submit code). Your working files will be stored within the VM. Back them up by pushing to your github repo. Save your work early and often. Several small clear commits and pushes is good practice. Communication with course staff will often involve looking at the code and commits in your repo.

The Git Remotes feature allows you to link GitHub repositories to your local Git repository. We have already set up a remote called "staff" that points to our skeleton code repos on GitHub, for both your personal and group repo. You will now add your own remote that points to your private repo so you can submit code.

You should have received the link to your personal private GitHub repo when you registered with the autograder earlier. Add a new remote by doing the following steps in your VM:

1. First cd into your personal repository.

   ```
   cd ~/code/personal
   ```

2. If the directory does not exist, run:

   ```
   git clone https://github.com/Berkeley-CS162/studentXXX.git ~/code/personal
   cd ~/code/personal/
   git remote add staff https://github.com/Berkeley-CS162/student0.git
   ```

3. Then visit your personal repo on GitHub and find the SSH clone URL. It should have the form "git@github.com:Berkeley-CS162/..."

4. Now add the remote

   ```
   git remote add personal YOUR_GITHUB_CLONE_URL
   ```

5. You can get information about the remote you just added

   ```
   git remote -v
   git remote show personal
   ```

6. Pull the skeleton, make a test commit and push to `personal master`

[7] https://github.com/Berkeley-CS162/student0/
[8] https://github.com/Berkeley-CS162/group0/

```
git pull staff master
touch test_file
git add test_file
git commit -m "Added a test file."
git push personal master
```

In this course, "master" is the default Git branch that you will use to push code to the autograder. You can create and use other branches, but only the code on your master branch will be graded. You should do this test commit before Monday. We want to know that everyone has got this basic infrastructure in place.

7. Within 30 minutes you should receive an email from the autograder. (If not, please notify the instructors via Piazza). Check cs162.eecs.berkeley.edu/autograder[9] for more information.

1.3 Autograder

Here are some important details about how the autograder works:

- The autograder will automatically grade code that you push to your master branch, UNLESS the assignment you are working on is LATE.
- If your assignment is late, you can still get it graded, but you will be using slip days. You can request late grading using the autograder's web interface at cs162.eecs.berkeley.edu/autograder[10].
- Your final score in the autograder is not the maximum of your attempts, but rather your score for only your latest build. Any non-autograded components, like style and written portions, will be graded based on your last build for the assignment.

The autograder is for grading, not for testing. You should develop and carry out your tests in your local environment. Lots of spurious autograder submissions can interfere with people getting their work done. And the turnaround time is too slow for testing. It provides final confirmation that your tests are consistent with ours.

1.4 Editing code in your VM

The VM contains a SMB server that lets you edit files in the vagrant user's home directory. With the SMB server, you can edit code using text editors on your host and run git commands from inside the VM. **This is the recommended way of working on code for this course**, but you are free to do whatever suits you best. One possibility is just using a non-graphical text editor in an SSH session.

1.4.1 Windows

1. Open the file browser, and press `Ctrl L` to focus on the location bar.
2. Type in `\\192.168.162.162\vagrant` and press Enter.
3. The username is **vagrant** and the password is **vagrant**.

You should now be able to see the contents of the vagrant user's home directory.

[9]https://cs162.eecs.berkeley.edu/autograder
[10]https://cs162.eecs.berkeley.edu/autograder

1.4.2 Mac OS X

1. Open Finder.
2. In the menu bar, select **Go** → **Connect to Server...**.
3. The server address is `smb://192.168.162.162/vagrant`.
4. The username is **vagrant** and the password is **vagrant**.

You should now be able to see the contents of the vagrant user's home directory.

1.4.3 Linux

Use any SMB client to connect to the `/vagrant` share on 192.168.162.162 with the username **vagrant** and password **vagrant**. Your distribution's file browser probably has support for SMB out of the box, so look online for instructions about how to use it.

1.5 Shared Folders

The `/vagrant` directory inside the virtual machine is connected to the home folder of your host machine. You can use this connection if you wish, but the SMB method in the previous section is recommended. (You can also learn more about the file system of your local machine by finding where the file system of your VM is mounted. Can you find it?)

2 Useful Tools

Before continuing, we will take a brief break to introduce you to some useful tools that make a good fit in any system hacker's toolbox. Some of these (git, make) are MANDATORY to understand in that you won't be able to compile/submit your code without understanding how to use them. Others such as gdb or tmux are productivity boosters; one helps you find bugs and the other helps you multitask more effectively. All of these come pre-installed on the provided virtual machine. They are ESSENTIAL.

Note: We do not go into much depth on how to use any of these tools in this document. Instead, we provide you links to resources where you can read about them. We highly encourage this reading even though not all of it is necessary for this assignment. We guarantee you that each of these will come in handy throughout the semester. If you need any additional help, feel free to ask any of the TA's at office hours!

2.1 Git

Git is a version control program that helps keep track of your code. GitHub is only one of the many services that provide a place to host your code. You can use git on your own computer, without GitHub, but pushing your code to GitHub lets you easily share it and collaborate with others.

At this point, you have already used the basic features of git, when you set up your repos. But an understanding the inner workings of git will help you in this course, especially when collaborating with your teammates on group projects.

If you have never used git or want a fresh start, we recommend you start here[11]. If you sort of understand git, this presentation[12] we made and this website[13] will be useful in understanding the inner workings a bit more.

2.2 make

make is a utility that automatically builds executable programs and libraries from source code by reading files called Makefiles, which specify how to derive the target program. How it does this is pretty cool: you list dependencies in your Makefile and make simply traverses the dependency graph to build everything. Unfortunately, make has very awkward syntax that is, at times, very confusing if you are not properly equipped to understand what is actually going on.

A few good tutorials are here[14] and here[15]. And of course the official GNU documentation (though it may be a bit dense) here[16].

For now we will use the simplest form of make: without a `Makefile`. (But you will want to learn how to build decent Makefiles before long!) You can compile and link `wc.c` by simply running:

```
$ make wc
```

This created an executable, which you can run. Try

```
$ ./wc wc.c
```

How is this different from the following? (Hint: run "`which wc`".)

```
$ wc wc.c
```

[11]http://git-scm.com/book/en/Getting-Started
[12]http://goo.gl/cLBs3D
[13]http://think-like-a-git.net/
[14]http://wiki.wlug.org.nz/MakefileHowto
[15]http://mrbook.org/blog/?s=make
[16]http://www.gnu.org/software/make/manual/make.html

2.3 man

`man` – the user manual pages – is really important. There are lots of stuff on the web, but the documentation in `man` is definitive. The `man` pages can be accessed through a terminal. For instance, if you wanted to learn more about the `ls` command, simply type "`man ls`" into your terminal. If you were curious about a function called `fork`, you could learn more about it by typing "`man fork`" into your terminal.

2.4 gdb

Debugging C programs is hard. Crashes don't give you nice exception messages or stack traces by default. Fortunately, there's the GNU Debugger, or gdb for short. If you compile your programs with a special flag `-g` then the output executable will have debug symbols, which allow gdb to do its magic. If your run your C program inside gdb, you will be able to not only look get a stack trace, but also inspect variables, change variables, pause code and much more! Moreover, gdb can even start new processes and attach to existing processes (which will be useful when debugging PintOS.)

Normal gdb has a very plain interface. So, we have installed `cgdb` for you to use on the virtual machine, which has syntax highlighting and few other nice features. In cgdb, you can use `i` and `ESC` to switch between the upper and lower panes.

This[17] is an excellent read on understanding how to use gdb. The official documentation[18] is also good, but a bit verbose.

2.5 tmux

tmux is a terminal multiplexer. It basically simulates having multiple terminal tabs, but displays them in one terminal session. It saves having to have multiple tabs of sshing into your virtual machine.

You can start a new session with `tmux new -s <session_name>`

Once you create a new session, you will just see a regular terminal. Pressing `ctrl-b + c` will create a new window. `ctrl-b + n` will jump to the nth window.

`ctrl-b + d` will "detach" you from your tmux session. Your session is still running, and so are any programs that you were running inside it. You can resume your session using `tmux attach -t <session_name>`. The best part is this works even if you quit your original ssh session, and connect using a new one.

Here[19] is a good tmux tutorial to help you get started.

2.6 vim

vim is a nice text editor to use in the terminal. It's well worth learning. Here[20] is a good series to get better at vim. Others may prefer emacs. Whichever editor you choose, you will need to get proficient with an editor that is well suited for writing code.

If you want to use Sublime Text, Atom, CLion, or another GUI text editor, look at 1.4 Editing code in your VM, which shows you how to access your VM's filesystem from your host.

2.7 ctags

ctags is a tool that makes it easy for you to navigate large code bases. Since you will be reading a lot of code, using this tool will save you a lot of time. Among other things, this tool will allow you to jump to any symbol declaration. This feature together with your text editor's go-back-to-last-location feature is very powerful.

[17] http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html
[18] https://sourceware.org/gdb/current/onlinedocs/gdb/
[19] http://danielmiessler.com/study/tmux/
[20] http://derekwyatt.org/vim/tutorials/

Instructions for installing ctags can be found for vim here[21] and for sublime here[22]. If you don't use vim or sublime, ctags still is probably supported on your text editor although you might need to search installation instructions yourself.

3 Your First Assignment

3.1 words

Programming in C is a very important baseline skill for CS 162. This exercise should make sure you're comfortable with the basics of the language. In particular, you need to be fluent in working with `structs`, linked data structures (e.g., lists), pointers, arrays, `typedef` and such, which 61C may have touched only lightly.

You will be writing a program called `words`. `words` is a program that counts (1) the total amount of words and (2) the frequency of each word in a file(s). It then prints the results to `stdout`. Like most Linux utilities in the real world, your program should read its input from each of the files specified as command line arguments, printing the cumulative word counts. If no file is provided, your program should read from `stdin`.

In C, header files (suffixed by `.h`) are how we engineer abstractions. They define the objects, types, methods, and – most importantly – documentation. The corresponding `.c` file provides the implementation of the abstraction. You should be able to write code with the header file without peeking under the covers at its implementation.

In this case, `words/word_count.h` provides the definition of the `word_count` struct, which we will use as a linked list to keep track of a word and its frequency. This has been `typedef`'d into `WordCount`. This means that instead of typing out `struct word_count`, we can use `WordCount` as shorthand. The header file also gives us a list of functions that are defined in `words/word_count.c`. Part of this assignment is to write code for these functions in `words/word_count.c`.

We have provided you with a compiled version of a `sort_words` so that you do not need to write the `wordcount_sort` function. However, you may still need to write your own comparator function (i.e. `wordcount_less`). The `Makefile` links this in with your two object files, `words.o` and `word_count.o`.

Note that `words.o` is an ELF formatted binary. As such you will need to use a system which can run ELF executables to test your program (such as the 162 VM). Windows and OS X do **NOT** use ELF and as such should not be used for testing.

For this section, you will be making changes to `words/main.c` and `words/word_count.c`. After editing these files, `cd` into the `words` directory and run `make` in the terminal. This will create the `words` executable. (Remember to run `make` after making code changes to generate a fresh executable). Use this executable (and your own test cases) to test your program for correctness. The autograder will use a series of test cases to determine your final score for this section.

For the below examples, suppose we have a file called `words.txt` that contains the following content:

```
abc def AaA
bbb zzz aaa
```

[21] http://ricostacruz.com/til/navigate-code-with-ctags.html
[22] https://github.com/SublimeText/CTags

3.1.1 Total Word Count

Your first task will be to implement total word count. When executed, `words` will print the total number of words counted to `stdout`. At this point, you will not need to make edits to `word_count.c`. A complete implementation of the `num_words()` function can suffice.

A word is defined as a sequence of contiguous alphabetical characters of length greater than one. All words should be converted to their lower-case representation and be treated as not case-sensitive. The maximum length of a word has been defined at the top of `main.c`.

After completing this part, running `./words words.txt` should print:

```
The total number of words is: 6
```

3.1.2 Word Frequency Count

Your second task will be to implement frequency counting. Your program should print each unique word as well as the number of times it occurred. This should be sorted in order of frequency (low first). The alphabetical ordering of words should be used as a tie breaker. The `wordcount_sort` function has been defined for you in `wc_sort.o`. However, you will need to implement the `wordcount_less` function in `main.c`.

You will need to implement the functions in `word_count.c` to support the linked list data structure (i.e. `WordCount` a.k.a. `struct word_count`). The complete implementation of `word_count.c` will prove to be useful when implementing `count_words()` in `main.c`.

After completing this part, running `./words -f words.txt` should print:

```
1 abc
1 bbb
1 def
1 zzz
2 aaa
```

Hint: You can run:

```
  cat <filename>
      | tr " " "\n"
      | tr -s "\n"
      | tr "[:upper:]" "[:lower:]"
      | tr -d -C "[:lower:]\n"
      | sort
      | uniq -c
      | sort -n
```

to verify the basic functionality of your program (don't treat this as a testing spec though).

3.2 user limits

Now that you have dusted off your C skills and gained some familiarity with the CS 162 tools, we want you to understand what is really inside of a running program and what the operating system needs to deal with.

The operating system needs to deal with the size of the dynamically allocated segments: the stack and heap. How large should these be? Poke around a bit to find out how to get and set these limits on Linux. Modify `limits.c` so that it prints out the maximum stack size, the maximum number of processes, and maximum number of file descriptors. Currently, when you compile and run `limits.c` you will see it print out a bunch of system resource limits (stack size, heap size, ..etc). Unfortunately all the values will be 0. Your job is to get this to print the ACTUAL limits (use the soft limits, not the hard limits). (Hint: run "`man getrlimit`")

You should expect output similar to this:

```
stack size: 8388608
process limit: 2782
max file descriptors: 1024
```

You can run `make limits` to compile your code.

3.3 The A-Z's of GDB

Now we're going to use a sample program, `map`, for some GDB practice. Before you start, be sure to take a look at `map.c` and `recurse.c` which form the program. Once you feel familiar with the program, you can compile it by running "`make map`".

Write down the commands you use to complete each step of the following walk-through. Be sure to also **record your answers to all questions in bold** `gdb.txt`.

a. Run GDB

b. Set a breakpoint at the beginning of the program's execution

c. Run the program until the breakpoint

d. **Where does argv point to?**

e. **What's at the address of argv?**

f. Step until you reach the first call to recur.

g. **What is the memory address or the recur function?**

h. Step into the first call to recur.

i. Step until you reach the if statement

j. Switch into assembly view

k. Step over instructions until you reach the 'callq' instruction

l. **What values are in all the registers?**

m. Step into the call instruction

n. Switch back to C code mode.

o. Now print out the current call stack (hint: what does the backtrace command do?)

p. Now set a breakpoint on the recur function which is only triggered when the argument is 0

q. Continue until the breakpoint is hit

r. Print the call stack now

s. Now go up the call stack until you reach main, what was argc?

t. Now step until the return statement

u. Switch back into the assembly view

v. **Which instructions correspond to the 'return 0' in C?**

w. Now switch back to the source layout.

x. Finish the next 3 function calls

y. Run the program to completion.

z. Quit gdb

3.4 Compiling, Assembling, and Linking

Now that you've seen how `map` works, let's take a dive into what how we went from high level C code to an executable.

There are 10 written questions for this section – put your answers in `call.txt`.

Before we start, we'll be using a few compiler flags which are likely new to you. Here's a summary of the flags we'll be using.

- `-Wall` – Enables all compiler warnings
- `-m32` – Compiles the code for the i386 architecture.
- `-S` – Invokes the COMPILER only.
- `-c` – Invokes the COMPILER and ASSEMBLER only.

Let's start by invoking the compiler. The compiler takes high level C code and produce a variant of x86 known as 8086 or i386 assembly.

To compile map.c, run:

```
$ gcc -m32 -S -o map.S map.c
```

This will only invoke the compiler for `map.c` and output the assembly code in `map.S`.

1. Generate `recurse.S` and find which instruction(s) corresponds to the recursive call of `recur(i-1)`.

Now we will assemble our compiled code into an executable. To assemble our code we can run:

```
$ gcc -m32 -c map.S -o map.obj
```

This turns our raw x86 code (`map.S`) into machine code or an object file (`map.out`).
We can also combine these steps by just running `gcc -m32 -c` on our C file directly. We can run:

```
$ gcc -m32 -c recurse.c -o recurse.obj
```

The assembler converts the raw assembly code into an object file which contains code as well as other data and metadata necessary for execution. Different operating systems use different types of object files. In this class, we will be using ELF (Executable and Linkable Format), the object format used by Linux. Let's start by taking a look `map.out` and `recurse.obj`. These are binary files, so we will use the `objdump` program to read them.

```
$ objdump -D map.obj
$ objdump -D recurse.obj
```

2. What do the `.text` and `.data` section contain?

The assembler generates a `symbol table` which is part of the object file. The `symbol table` contains all the symbols that can be globally referenced (referenced outside the object file) from another object file (i.e. global/static variables and functions).

3. What command do we use to view the symbols in an ELF file? (Hint: We can use `objdump` again, look at "`man objdump`" to find the right flag).

Here's an excerpt from the `map.obj` symbol table:

```
00000000 g     O .data 00000004 stuff
00000000 g     F .text 00000060 main
...
00000000         *UND* 00000000 malloc
00000000         *UND* 00000000 recur
```

4. What do the `g`, `O`, `F`, and `*UND*` flags mean?
5. Vaguely describe another location where we can find the symbol for `malloc`.
6. Where else can we find a symbol for `recur`? Which file is this in? **Copy and paste the relevant portion of the symbol table.**

Finally, let's link our 2 objects files to create an executable.

```
$ gcc -m32 map.obj recurse.obj -o map
```

Note that to we could've just called `gcc -m32 map.c recurse.c -o map` on the C files to do this entire process in a single command. Often times build systems will separate these commands in order to speed up compile times (since only the changed files need to be recompiled).

7. Examine the symbol table of the entire `map` program now. What has changed?

3.5 Autograder & Submission

3.5.1 Autograder

To push to your code to the autograder do:

```
cd ~/code/personal/hw0
git status
git add gdb.txt call.txt
git add limits.c map.c recurse.c words/main.c words/word_count.c Makefile
git commit -m "Finished my first CS 162 assignment xD"
git push personal master
```

This saves your work and it gives the instructors a chance to see the progress you are making. Congratulations for not waiting until the last minute.

Within a few minutes you should receive an email from the autograder. (If not, please notify the instructors via Piazza). Check cs162.eecs.berkeley.edu/autograder[23] for more information.

Your work on `gdb.txt` and `call.txt` will not be graded by the autograder and instead will be graded manually.

Hopefully after this you are slightly more comfortable with your tools. You will need them for the long road ahead!

[23]https://cs162.eecs.berkeley.edu/autograder

3.5.2 Gradescope

Written portions of assignments will be submitted to Gradescope. If you're enrolled in the class you should've already been added to Gradescope.

If you're not on Gradescope or the autograder, please make a private post on piazza, or email Alex Wu (email can be found on the course website[24]) and **include your name, email, and sid**.

[24]https://cs162.eecs.berkeley.edu/staff/