HW 0: **Executable**

Vaishaal Shankar and David Culler

August 31, 2014

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 11:59 pm 9/8/2014

1 Setup

In order to handle dependencies and help those of you who may have trouble getting a development environment set up we have generated a virtual machine box that is preconfigured with all the tools necessary for this class. The following are steps in setting up the virtual machine. (This requires a GB or so on a reasonable machine. We will provide a preconfigured setup on an inst machine in case you do not have a machine capable of a personal installation.)

1.1 GitHub

Code submission for all projects and homework in the class will be handled via GitHub so it is imperative that you get a GitHub account (if you don't already have one). It is suggested but not mandatory to use your berkeley.edu email address as you can request 5 free private repositories from GitHub. In this class we will provide you with private repositories for all your projects.

EXTREMELY IMPORTANT: Once you have set up a GitHub account please fill out this form in order for us to associate your cs162 class login and name with your account (and to collect some other useful information). You will not be able to complete or submit any assignments (including this one) without doing this!!! You should get an email confirming your submission, and your instructional account form should be attached.

1.2 Vagrant

Vagrant is an open source tool for creating, configuring & managing virtual machines. We are going to use this tool to manage our virtual machine for this class. Setting up Vagrant to work for our class is pretty easy.

Note: If you do not want to setup Vagrant on your own machine, you can use any of the instructional machines for this homework. For the later projects, you can use icluster24.eecs to run Pintos, but this machine will be significantly slower than developing locally.

1. Behind the scenes Vagrant uses VirtualBox (an open source virtualization product) so first you will need to download & install the appropriate version from the VirtualBox website (126 MB) [1]. We will talk in class about virtual machines. It is a software version of raw hardware. To use it you will need to install an operating system onto it. We have built a version of Linux for the class with a snapshot of all the tools installed.

2. Now install the appropriate version of Vagrant from their website (78 MB) [2]. You can think of Vagrant as a commandline version of VirtualBox.

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

```
vagrant init cs162/fall2014
vagrant up
vagrant ssh
```

[1] https://www.virtualbox.org/wiki/Downloads
[2] http://www.vagrantup.com/downloads.html

That should download the correct virtual machine from our servers and drop you into an ssh session. Note the 'up' will take a while. You'll want to have a good network connection and start early.

1.2.1 Windows

Since windows does not support ssh, the third command will cause an error message prompting you to download Cygwin or something similar that supports an ssh client. Here is a good guide on setting up vagrant in windows.

1.2.2 ssh-keys

You will need to setup your ssh keys in order to access GitHub from within your VM.

New GitHub Users

From within your VM, follow the GitHub's guide on how to generate ssh keys and add them to GitHub account. Note: to copy your public key to GitHub, you may need to copy and paste out of your terminal instead of using xclip through ssh. Be sure not to include extra white space.

Existing GitHub Users

If you already have GitHub all set up on your local machine, you can use ssh-agent forwarding to utilize your local credentials within the virtual machine.

First, be sure to have your private key added to your local ssh agent:

```
$ eval "$(ssh-agent -s)"
$ ssh-add <path_to_your_private_key>
```

Then simply use `vagrant ssh` to ssh into your machine.

The rest of this document will assume you are inside your virtual machine.

1.2.3 repos

Each of you will have access to two repositories in this course. One will be a personal repository that will be used to submit individual assignments and homework. The other will be a group repository that you will share with the rest of your group (once you've formed your group that is). A skeleton group and personal repository should already exist inside the `code` directory in your home folder with the names group and personal. They are just not linked to your own personal & group remotes. Since you don't have groups yet, we will set up just your personal repository for now. Both these repositories should have the `staff` remotes which will contain the skeleton code for all homeworks and projects. (Currently only hw0 and project1).

1. First cd into your personal repository

   ```
   cd code/personal
   ```

2. Now add the appropriate remotes (remotes allow you to point to different online repos from within your local repo)

   ```
   git remote add personal git@github.com:Berkeley-CS162/<two-letter-login>
   ```

   Please note you will only be able to push/pull from these repos once we have added you to the Berkeley-CS162 organization. This should be automatic if you have been enrolled in the course for more than a few days. If not please contact the instructors on Piazza.

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

```
git pull staff master
touch test
git add .
git commit -m "test commit"
git push personal master
```

4. Test out the autograder The CS162 autograder monitors the branches starting with ag for commits and runs tests on the commits pushed to that branch. This branch exists purely for your own experimental purposes and you can push there as many times as you want.

```
git checkout -b ag/hw0
git push personal ag/hw0
```

   Within a few minutes you should receive an email from the autograder. (If not, please notify the instructors via Piazza).

1.3 Shared Folders

Currently you are forced to use a terminal based editor to edit your code through the ssh connection. But vagrant provides folder sharing between your guest and host machines. The folder `/vagrant` on your virtual machine is shared with the home folder of your host machine. You can move the code directory into there if you want to edit the code with a native gui editor.

At this point, you should have seen the `hw0` directory in your repo, which contains `main.c`, `wc.c` and `map.c`.

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. These all come preinstalled in the provided virtual machine.

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 bother any of the TAs at office hours!

2.1 git

git is a (relatively new) version control software that helps keep track of your code. Think of it on Dropbox (but optimized for code) on steroids. GitHub is one of the many only services that provide a place to HOST your code, git is just a program on your computer but GitHub provides a place to "push" your code.

If you have gotten to this point, you have already successfully used git, but understanding some of the inner-workings will really help you on your hacking. If you have never used git or want a fresh start, we recommend you start here. If you sort of understand git this presentation we made and this website are useful in understanding the inner workings a bit more.

At this point, `cd` into `hw0` and `ls` should show a couple of `.c` files.

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 and here. And of course the official GNU documentation (though it may be a bit dense) here

For now we will use the simplest form of it, without even a `Makefile`. You will want to learn how to build decent Makefiles before long! You can compile and link `wc.c` with simply:

```
make wc
```

This created an executable, which you can run. Try

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

How is this different from the following?

```
wc wc.c
```

Hint: `which wc`. Your first assignment is goiong to be to fill out the code in `wc.c` so that it implements `man wc`, except that it takes a single input file (or stdin in none is specified), no flags, and, if a second file name is specified, the output destination, otherwise `stdout`.

2.3 gdb

Debugging C programs is hard. Simply put, it doesn't give you nice exceptions or stack traces. Fortunately that's where gdb comes. If you compile your programs with a special flag `-g` then the binary will have debug symbols, which will allow gdb to do its magic, if your run your C program inside gdb it will allow you to not only look at a stack trace, but inspect variables, change variables, pause code and much more!

Normal gdb has a very plain interface, Thus, we have installed `cgdb` for you to use on the virtual machines, which has syntax highlighting and few other nice features.

gdb can also attach to running programs (which will be useful when debugging your OS)

This is an excellent read on understanding not only how to use gdb but how your programs work.

Again official documentation is also good if not a bit verbose.

Take a moment to begin working on your `wc`. Provide the -g flag when you `make`. Start the program under gdb. Set a break point at main. Run to there. Try out various commands. Figure out how to pass command line args. Add local variables and see how you can probe their values. Learn about single step, next and break.

2.4 tmux

tmux is a terminal multiplexer, it basically simulates having multiple terminal tabs but 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>` After which you are in a regular terminal but pressing `ctrl-a + c` will create a new window. `ctrl-a + n` will jump to the nth window.

`ctrl-a + d` will "detach" your tmux session which you can re-attach 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. It helps keeping your environment persistent.

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

This is the tmux config we set up in the VM, so the key-mappings in the VM will correspond to this.

Note: By default tmux commands begin with ctrl-b but we remapped it to ctrl-a in the virtual machine, if you want to try out tmux else where you will need to do that yourself

2.5 vim

vim is a nice text editor to use in the terminal. It's well worth learning. Here is a good series to get better at vim. Others may prefer emacs. You will need to get proficient and a editor that is well suited for writing code.

Note: By default you enter and exit normal mode in vim by pressing the colon key, I (Vaishaal) got tired of pressing shift, so I mapped semi colon to colon. I shared this setting in the virtual machine we provided you

3 Your first assignment

3.1 make

You have probably been using `gcc` to compile your programs, but this grows tedious and complicated as the number of files you need to compile increases. You will need to write a `Makefile` that compiles `main.c`, `wc.c`, and `map.c`. You will also need to write a target `check` that runs some sort of test that will verify the output of `wc` and `main`. (You may do this any way you wish, but you MUST verify that the output is existent and non-erroneous). It may also help to write a clean target to remove your binaries. If all of this is new to you please read 2.2

3.2 wc

We are going to use `wc.c` to get you thinking once again in C with an eye to how application utilize the operating system - passing command line arguments from the shell, reading and writing files, and standard file descriptors. All these things you encountered in cs61c, but they will take on new meaning in cs162.

Your first task to write a clone of the unix tool `wc` which counts the number of words inside a particular text file. You can run unix's `wc` to see what your output should look like, and try to mimic its basic functionality (don't worry about the flags or anything) in wc.c.

While you are working on this take the time to get some experience with gdb. Use it to step through your code and examine variables.

3.3 executables and addresses

Now that you have dusted off your C skills and gained some familiarity with the cs162 tools, we want to understand better what is really in an executing program and what the operating system actually deals with.

Load up your `wc` executable in gdb with a single input file command line argument, set a breakpoint at `wc` and run to there. Take a look at the stack using `where` or `backtrace` (bt).

While you are looking through gdb try and think about the following questions, and put the answers in the file `gdb.txt`.

- What is the value of `infile`? (hint: print infile)
- What is the object referenced by `infile`? (hint: *infile)
- What is the value of `ofile`? How is it different from that of `infile`? Why?
- What is the address of the function `wc`?
- Try `info stack`. Explain what you see.
- Try `info frame`. Explain what you see.
- Try `info registers`. Which registers are holding aspects of the program that you recognize?

We have just peeled away the abstraction layers that is the onion of an executing program: the source language program, compiled into an object, linked into a binary executable, that is loaded and executed on a computer. The operating system meets the application as an executable file when you run it. There is more to the executable than meets the eye. Let's look down inside.

```
objdump -x wc
```

You will see that it has several segments, names of functions and variables in your program correspond to labels with addresses or values. And the guts of everything is chunks of stuff within segments.

While you are looking through the objdump try and think about the following questions and put the answers in the file `objdump.txt`.

- What file format is used for this binary? And what architecture is it compiled for?
- What are the names of segments you find?
- What segment contains `wc` (the function) and what is it's address? (hint: objdump -w wc — grep wc)
- What about `main`?
- How do these correspond to what you observed in gdb when you were looking at the loaded, executing program?
- Do you see the stack segment anywhere? Explain.
- What about the heap?

OK, now you are ready to write a program that reveals its own executing structure. The second file in hw0, `map.c` provides a rather complete skeleton. You will need to modify it to get the addresses that you are looking for and get the type casts right so that it compiles without warning. The output of the solution looks like the following (the addresses will be different).

```
precise64   hw0  ./map
Main  @ 40058c
recur @ 400544
Main stack: 7fffda11f73c
static data: 601028
Heap: malloc 1: 671010
Heap: malloc 2: 671080
recur call 3: stack@ 7fffda11f6fc
recur call 2: stack@ 7fffda11f6cc
recur call 1: stack@ 7fffda11f69c
recur call 0: stack@ 7fffda11f66c
```

Now think about the following questions and put the answers in `map.txt`.

- Using objdump on the map executable, find which of these are defined in the executable and the segment that they are in. You can now make a list of the segments and what they are used for.
- What direction is the stack growing in?
- How large is the stack frame for each call to recur?
- Where is the heap? What direction is it growing in?
- Are the allocated object contiguous?
- Make a high level map of the address space for the program containing each of the segments, where they start and end, where the holes are, and what direction things grow in.

3.4 user limits

The size of the dynamically allocated segments, stack and heap, is something the operating system has to deal with. How large should these be? Poke around a bit to find out how to get and set user limits on linux. Modify main.c so that it prints out the maximum stack size, the maximum number of processes, and maximum amount of file descriptors. Currently you compile and run it 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 statistics. What other limits on the user process can you determine? (Hint: `man rlimit`)

You should expect output similar to this:

```
precise64   hw0  make
precise64   hw0  ./main
stack size: 8192
process limit: 2782
max file descriptors: 1024
precise64   hw0  make check
pass!
precise64   hw0  echo $?
0
```

3.5 autograder & submission

Note: This autograder will probably not be live until 9/3/2014, we need to wait till all of you have filled out the google form before we turn on the autograder, so please don't expect a result till then
To push to autograder do:

```
make clean
git add .
git commit -m "hw0 test"
git checkout -b ag/hw0
git push personal ag/hw0
```

This saves your work and it gives the instructors a chance to see the progress you are making. Congratulations for not waiting till the last minute. Within a few minutes you should receive an email from the autograder. (If not, please notify the instructors via Piazza).

Now in order to finally submit your code, you need to push to the branch `release`

```
make clean
git add .
git commit -m "hw0 submission"
git checkout -b release/hw0
git push personal release/hw0
```

The reason we gave you two types of branches with an autograder, is that the `ag/*` are testing branches, nothing on it will be graded whereas you **must** submit to release in order to get graded. So please only push to `release/*` when you intend to submit.

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