John Hardy

3/18/18

Operating Systems

# Homework 3 Tutorial

The process of finishing this project was by no means easy. By far the hardest part was getting started early, knowing that I had a long journey before me. The benefit of starting early was that I could work at a slow pace, and could take my time as I completed each objective.

## 1. Getting Ubuntu running

The first step was to simply get my Ubuntu virtual machine running so that I could have a platform to start from. To do this, I download the VMware 14 virtual machine online, which gave me a free 30 day trail, more than enough time to complete this assignment. I then downloaded the Ubuntu 16.04 iso from the Ubuntu website.

The machine I used to complete this assignment was my Windows 10 Desktop computer with 16 GB of RAM and two separate drives. One SSD that held my native Windows 10 Operating System (only 250 GB roughly) and a 500 GB HDD that would hold my Ubuntu 64-bit Virtual Machine (only 35 GB would actually be used for the VM). I also set up my VM to use 10.6 GB of my available RAM while it was running. I took the time to set all of this up on my nicer desktop computer so that the compilation time of the build would be much shorter than on my older Laptop which was already very slow in running Ubuntu.

Without much effort I was able to get Ubuntu loaded up and running. The only weird problem I ran into was my back-lit RGB keyboard would magically turn off the back light every single time my cursor entered the virtual machine and would magically turn back on when it left the VM. I assume this minor inconvenience was caused by a driver problem. Dark keyboards aside, I was able to start the assignment.

## 2. Grabbing the Source

Since I already had Ubuntu running in my VM ware virtual machine, I decided to go with the apt-get route. Upon pasting the command: `apt-get source linux-image-$(uname -r)` into my terminal, I immediately get an error saying: "`E: You must put some 'source' URIs in your sources.list`". Apparently the current version of Ubuntu that I am using did not list its source in the `sources.list` file. This is likely because I downloaded the iso file for the virtual machine online. I looked online for help and found an Ubuntu Sources List Generator: https://repogen.simplylinux.ch/ which gave me my source list. I used the file manager to get to the appropriate directory, and clicked on `sources.list` to edit it. Doing this pulled up the system settings GUI and I found an option to download the source files. Apparently all I had to do was tick an option to have the computer update the source list for the latest releases of the source code. Doing this I went back to the console and re-entered my command. Viola! Just like that I was downloading the source code for my Ubuntu release. So much for going through the trouble of the source list generator.

From here I went to the next step which was downloading the packages needed for my build environment. I went back to the tutorial for building Linux and pasted the command for grabbing my build packages into the terminal: `sudo apt-get build-dep linux-image-$(uname -r)`. Immediately strings of text went whizzing by as the machine downloaded the packages, asking for permissions once or twice to allocate space needed for these packages on the disk. Five minutes later, my build packages were done downloading and unpacking, I was now ready to move on to the next step in the compilation chain!

## 3. The System Call

The next step was to find the system call table. This was not easy because the directories were wrong in the tutorial. However, after 30 minutes of googling and searching through directories in the file manager, I was able to find the call table in: `/home/john/linux-hwe-4.13.0/arch/x86/entry/syscalls`. From here I continued with what the tutorial told me to do. Since my call table had non-specific system calls all the way up to 332, I added mine as call number 333, and I used the name `supworld`.

For this next step, I had to make a choice between different folders since there were multiple choices. I had to decide which Linux version folder I should go to in order to change my `syscall.h` header file. Ultimately I decided on: `/usr/src/linux-headers-4.13.0-37/include/linux/syscall.h` since it was the most recent one and it looked the closest to the tutorial. I opened the file and added the line of code at the bottom of the `#endif`. Unfortunately, I did not have the right permissions to save the file *sigh*. After 5 minutes of thinking on my own, I made a startling realization. I had downloaded the source files into my home directory for my user, but I was in the root directory of the file system. I was not trying to change my custom Linux kernel, I was actually trying to change the Kernel of the host OS! It was a good thing that I was barred from actually saving the file, or I could have screwed up my Ubuntu kernel horribly! Realizing my mistake, I changed the `syscall.h` file in this directory: `/home/john/linux-hwe-4.13.0/include/linux/syscall.h`.

Next I made my system call. I went into the kernel directory and created a new text document containing the copy and pasted code from the tutorial, replacing the printed text with “`Sup World`”. Interestingly enough, this step was surprisingly easy. I was now ready to edit the make files.

Saving and exiting the text editor, I scroll up and open the make file and add the necessary `supworld.o` to the `obj-y`. Saving and exiting I go up one directory and change the next make file to note the `.syscall` on the extraversion line. Now I have successfully completed all the steps necessary for the creation of my System Call. I am now ready to move on to the next step, building the kernel.

## 4. Building the Kernel

With all the files in place and my concurrency level set to 2, I start the build process. I copy and paste my first command: `fakeroot debian/rules` clean and see a flurry of compiling code whizz through my screen. Next up I copy and paste the command: `fakeroot debian/rules` binary-headers binary-generic binary-perarch into my terminal and press enter. Expecting the compilation to take a long time I leave my VM and look at something else. When I return 5 minutes later to check on the build, I notice it ran into an error early in the process:

```
/home/john/linux-hwe-4.13.0/include/linux/once.h:7:1: error: expected
'=', ',', ';', 'asm' or '__attribute__' before 'bool'

 bool __do_once_start(bool *done, unsigned long *flags);
```

That is incredible! The people who wrote this code must have made a mistake when they wrote the code which caused this error. With courage in my heart, I look at the once.h file and notice something. The bool __do_once_start function declaration is found within the scope of an #ifndef body. Even with my limited understanding of C, I could see that something looked wrong here. I cut and paste the two functions declarations present in the body and move them to the bottom of the script after the #endif hoping that this would fix the problem. I save and exit and start the compilation over again.

Once again, the compilation failed, with the same problem oddly enough. I thought that changing the file would have cause a different error but it did not. Furthermore, the build also found another file that looks like it had an error inside of it: /home/john/linux-hwe-4.13.0/arch/x86/include/uapi/asm/ldt.h. I go back to once.h and undo what I did to the code. I decide to try to fix the code myself one more time before consulting google. This time I add the keywords "__attribute__" before the bool in the function declaration. I run the compilation after saving and closing, hoping for the best outcome.

Unfortunately, the compilation still ran into the same errors and I was unable to figure out what the computer was unhappy with in the once.h file. I go back and undo my changes, and consult google. Looking online, I see that the parameters binary-headers, binary-generic, and binary-perarch are separated in some builds. I decide to try each of them separately to see which one is causing problems. I try the first parameter: fakeroot debian/rules binary-headers which compiles with no problems. Then I try the next parameter: fakeroot debian/rules binary-generic. A new stream of text flutters through the screen, but upon reaching the once.h file, the compiler fails as expected. Now I know which step is causing the error. With this in mind, I try the final parameter: fakeroot debian/rules binary-perarch. This process took longer than the previous steps, taking about 6 minutes to complete. The coolest part about this step was that I was listening to music on YouTube while it was building, and every time the build got to a very large file, the music would audibly stutter for a second or two. Luckily, the build finished with no errors and looking into the directory above my Linux kernel, I could see 3 more .deb files waiting for installation.

But still unable to build the generic binaries, I go to the Linux Kernel repository and find the corresponding once.h file. Upon inspection, I realize that the once.h in git is different than mine, but the difference is in a part of the code that is not causing the problem. I decide to make the changes anyway to see if it would work. As expected, the changes did nothing.

I looked up the problem online some more and realized that **there is no type in C called 'bool'**! So, that likely means that one of the header files included (likely `types.h`) is not correctly supplying the type needed for the `bool`. I followed the header files `types.h` and found a declaration for `bool` as `_Bool` but nowhere could I find `_Bool` being defined. Obviously, programmers are not immune to mistakes, and sometimes, when they change the name of a header or write code for a file, they do not check to make sure that they did not mess up their code elsewhere. I decide to take matters into my own hand and insert the following type definition I found on stack overflow: `typedef enum { false, true } bool;` from here I ran the command to see if my type definition patch worked. After running the compilation, there was a conflict between the enumerated `bool` and the `bool` in `types.h` so I went ahead and changed the `types.h` header file as well. Soon I realized that this was not the problem at all. The type definitions were being created as they should have been, I simply had overlooked the header files that they were coming from. This meant that I had spent a long time trying to create the type definition of bool in `once.h` when that was not necessary at all.

Having run out of options, I decide to try running the command: `fakeroot debian/rules binary` instead, just to see what would happen. Once again, when the compiler reached the generic binaries folder, it ran into the same problem as before. After consulting Professor Forney it became apparent that there could have been a problem with the system not being updated correctly. So I feed the script `apt-get update` but it refuses to work due to permission being denied. At this point I decide to go back to step 2 and try the git method rather than the apt-get method. Perhaps there are missing files in the source files I have.

## 2. Grabbing the Source (again)

I go back one directory to my home directory and move all of the files in my failed build to a new folder so that I have them for safe-keeping. I then go back to my console and feed it the command `git clone git://kernel.ubuntu.com/ubuntu/ubuntu-<release codename>.git` which throws an error since `release codename` was not specified. I decide to just go with `git clone https://github.com/torvalds/linux.git` instead since that is what I found online. This does not work initially since bash did not have git installed. After installing git, I feed the console the appropriate command and watch as git queues up a HUGE download with 5,910,830 objects. It quickly became apparent to me that using the git method would take much longer, so I let the program run in the background while I work on other work (just for reference, it took 10 minutes to download 1% of the content). After an hour and a half, I scrapped the download seeing that it would take far too long to finish.

I decided to try apt-get one more time. I went back to my home directory and ran the apt-get command. Within 3 minutes I had all the files ready to go. I notice that some of the files there had not been there before, which leads me to believe that I may have accidentally deleted them previously. With this in mind, I resolved not to touch anything and just to follow the tutorial carefully.

## 3. The System Call (again)

Once again I had to change the some files in order to implement my system call. I followed the same instructions as before going to the syscall directory and changing the 64 bit system call table to hold my supworld system call. I then changed around my header files and added the code for my system call. I notice that I had forgotten to change the name of the `myservice` function to `supworld` before, so I made sure to do that this time around. Then I changed the two make files in the kernel and main directory of my kernel image as the last step. Now I was ready to see if all this work was pointless or not.

4. **Building the Kernel (again)**

First I make sure to set my concurrency again, then I go to the root of my kernel and run the first command: `fakeroot debian/rules clean`. This command is successful as expected, so I move on to the next one: `fakeroot debian/rules binary-headers binary-generic binary-perarch`. Once again, as expected, even after hours of trouble-shooting, the same problems come up with the `once.h` file. I decide that there must be something wrong with my machine or build otherwise I would not be getting so many problems so early, so I decide to go back to step one and download virtual box and try that instead of VMware. Note that by doing this, I effectively have to start this entire project over again from the start because I need to create a new Linux image for Virtual Box to work.

1. **Getting Ubuntu Running (again)**

I start by going to the Oracle website and downloading Virtual Box. Then I download another Ubuntu iso from the Ubuntu website to create my virtual machine with. After creating a virtual box image and navigating a couple of menus to set up my Ubuntu image, I am on my way to getting it working. After 30 minutes my Ubuntu VM is finally up and running. The next steps are pretty straight-forward and I have done them before so I will jump to step 4.

4. **Building the Kernel (again-again)**

After repeating steps 2 and 3 with no issues, I am ready to start my build for the third time. Hopefully using Virtual Box rather than VMware will make the difference this time. I set my concurrency level to two to utilize my resources as much as possible and feed it the first command: `fakeroot debian/rules clean`. A flurry of text dances past the screen and the headers are built with no problems. Now it was time for the real test. Hopefully all this hard work I put into building the kernel in Virtual Box pays off now. Nervously, I paste then next command into the terminal and hit enter: `fakeroot debian/rules binary-headers binary-generic binary-perarch`. I watch the console carefully and hopefully as meaningless streams of text whizz by. Unfortunately, when the build reaches the binary-generic files like it did before, the stupid once.h file does not compile. At this point I felt very discouraged. I felt that I had spent hours and hours trying to build the kernel already, but I am still unable to get past this problem with the build. Was it because I am running the software on a Virtual Machine? Was it because I am using a bad architecture for my computer? I was not sure what it could be.

After thinking for a while, I decide to try updating various software used in the build to see if that was causing a problem. I upgraded my gcc and g++ to the latest version and ran the build. This did not fix the problem, so I tried to update my apt-get. I feed the terminal `sudo apt-get upgrade` and watch as it updates my apt-get files. Again this did not seem to fix the problem when I tried to compile my binaries. At this point, I am lead to believe that future attempts to fix the problem on this computer are likely futile as I will need to show Professor Forney what I have done up to this point in order to get the most help possible. I decide to move everything to my slower but much more portable laptop computer. Again I will summarize what I did as the steps on my Windows 10 laptop are pretty much exactly the same.

## 4. Building the Kernel (I can't remember how many times I have tried this)

After installing virtual box and the Ubuntu iso onto my laptop, I go ahead and start the process of getting Ubuntu running. Once everything is running, I use the apt-get route to grab my source files. In doing so, I see a warning in the console that tells me there might be unreleased packages for the kernel that can be grabbed off of git using this command:

```
git clone git://git.launchpad.net/~ubuntu-
kernel/Ubuntu/+source/linux/+git/xenial –b hwe
```

Knowing that this would likely take hours to download, I decide to continue until I run into any problems. At this point I decided NOT to modify the kernel at all and to just try building the kernel by itself to see if I run into the same problem as before with the non-compiling generic binaries. After about 10 minutes of sitting around waiting for an error I realize that there will not likely be a compilation error, so I set my pc to not power down and I go off to do other things while I wait for the generic binaries to compile. 3 and ½ hours later, my generic binaries have finished compiling. I have compiled the binaries for the first time with no problems. I decide to compile my perarch binaries next. This one took less time and after only 20 minutes it was finished compiling. As this point I had all the ingredients to install all of the compiled binaries as my new kernel. This is proof that I can indeed compile my own kernel and perhaps there was something wrong in my modification of the kernel that made it not build in the first place.

I decide to try and create my system call as I did before and build everything a second time. That way, it would be less likely to run into some sort of compilation problem unless it was something that I had introduced. With this in mind I go through the same steps as before to implement my system call. Along the way I make doubly sure that all the directories go where they are supposed to and that even the header files exist where they need to. I also double checked carefully to make sure that my syntax was correct as well. Even after checking for mistakes, when I run the compilation for the generic binaries build I notice a warning for "`no build rules for myservice.o`". Seeing this I instantly knew I made a mistake, so I stop the build and fix the mistaken copy paste in the obj-y part of the make file. I restart the build and watch as it detects my changes to the make file and compiles like it should. A few hours later THE BUILD FINISHES. For once I have not run into a serious problem when trying to build the generic binaries. With no hesitation, I build the perarch binaries.

## 5. Installing the Kernel

After all the building has finished, I am ready to install my kernel. To install all of the .deb files I run the command: `sudo dpkg -i linux*4.13.0-37.42*.deb` which completes the installation of the generic binaries and image files. Note that some of the linux tools and cloud-based tools did not install correctly due to configuration problems, but these do not appear to be necessary for running the new kernel, so I ignore the problem. After this has finished I run `sudo reboot` and hold shift on the splash screen. When I get to make my choice between kernels, I don't see any new kernels to select that match the version I created. Perhaps the current kernel was overwritten? I select the current kernel and start the machine. When I get back to the home screen, I get to work writing my `syscall_test.c` file to see if my hard work has paid off yet.

After building my test, I am ready to compile and run to see if everything is working. I go to my terminal, compile the code and prepare myself for the moment of truth. I enter the command: `./a.out` into the terminal and press enter. My script runs and reports that my system call was indeed called and returned the value 0! This is a good sign! That means that it likely executed as it should have and is waiting for me to find it in the kernel log. To find the appropriate text I run the command: `dmesg | grep “Sup World”` to find any text in the kernel log with the words “Sup World” in it. Running this command the first time did not work. I looked through the `dmesg` text myself and did not see `“Sup World”` immediately. Sensing that this could have been a fluke do to a recent overwrite to the kernel log from background processes, I decide to try again to see if it would work this time. Doing so yielded the expected result: `[ 1746.249861] Sup World`. I had done it! My custom Linux kernel was running and I had successfully created my own system call and called it! The elation I felt at seeing those beautiful words on the screen was indescribable. I felt as if I had just completed a very long journey and that all the hard work and long nights I spent working on this assignment had finally payed off. I had changed and compiled my own kernel for the first time in my life. Very few people can say that they have ever done such as thing.

The last step now is to package all of my code, this tutorial, and my screen shot from this momentous occasion into a zip file to turn in. But before I do that, I will first take a nice long break for a while and appreciate what I have completed.