Makefile Basics for Linux

For anyone wanting to really learn make and Makefiles, I highly recommend this book - http://oreilly.com/catalog/make3/book/index.csp, but I will attempt here to give you the very basics to get you going. To follow along, download the following code from my UA-A website here - http://math.uaa.alaska.edu/~ssiewert/a335_code/EXAMPLES/POSIX/, which is also included as a zip file for convenience (note instructions for download, unzip, build, and debug follow after this section on the Makefile).

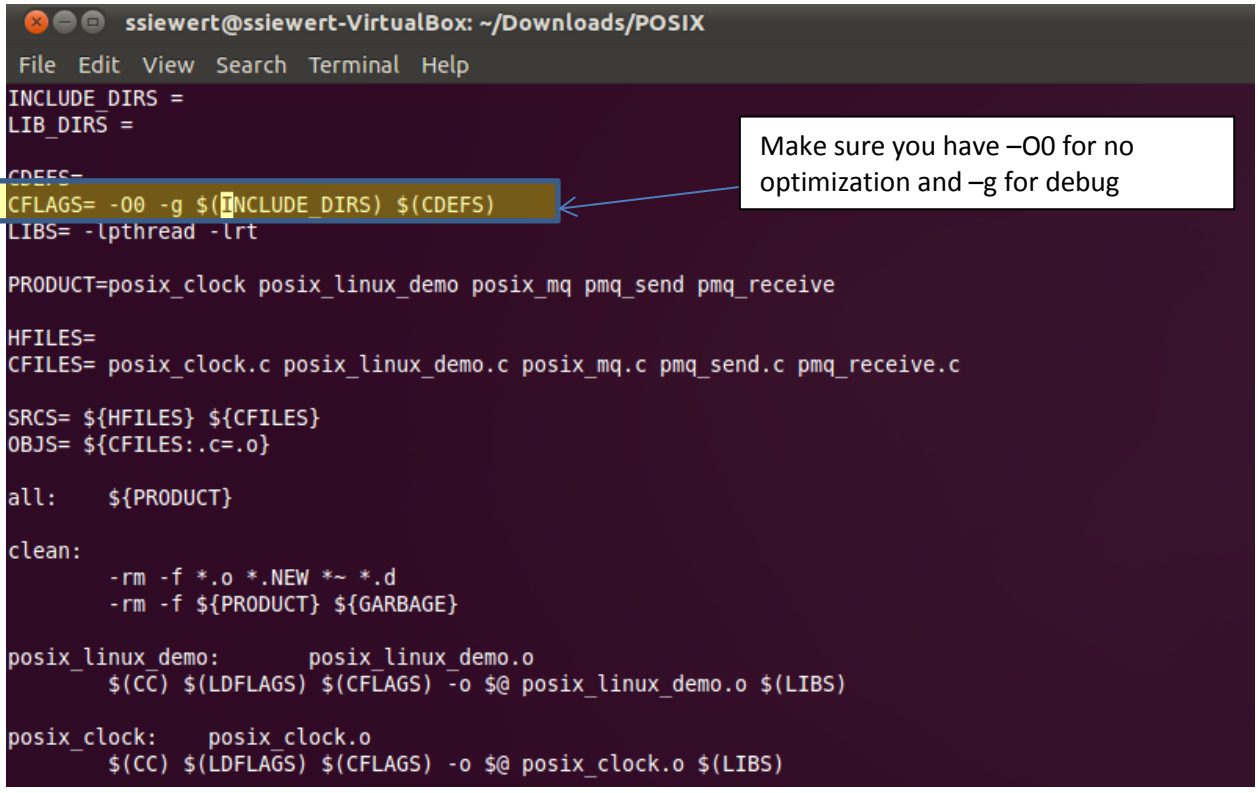
Also, I highly recommend installing Virtual-Box and then Ubuntu Linux on it running both on Windows with Linux as a Guest-OS on the virtual machine – note that I have instructions for doing this – you need a Linux installation you can play with anywhere if you want to learn Linux.

If you're new to Linux and/or just don't like command line type development, here's some recommendations for the course on using Makefiles:

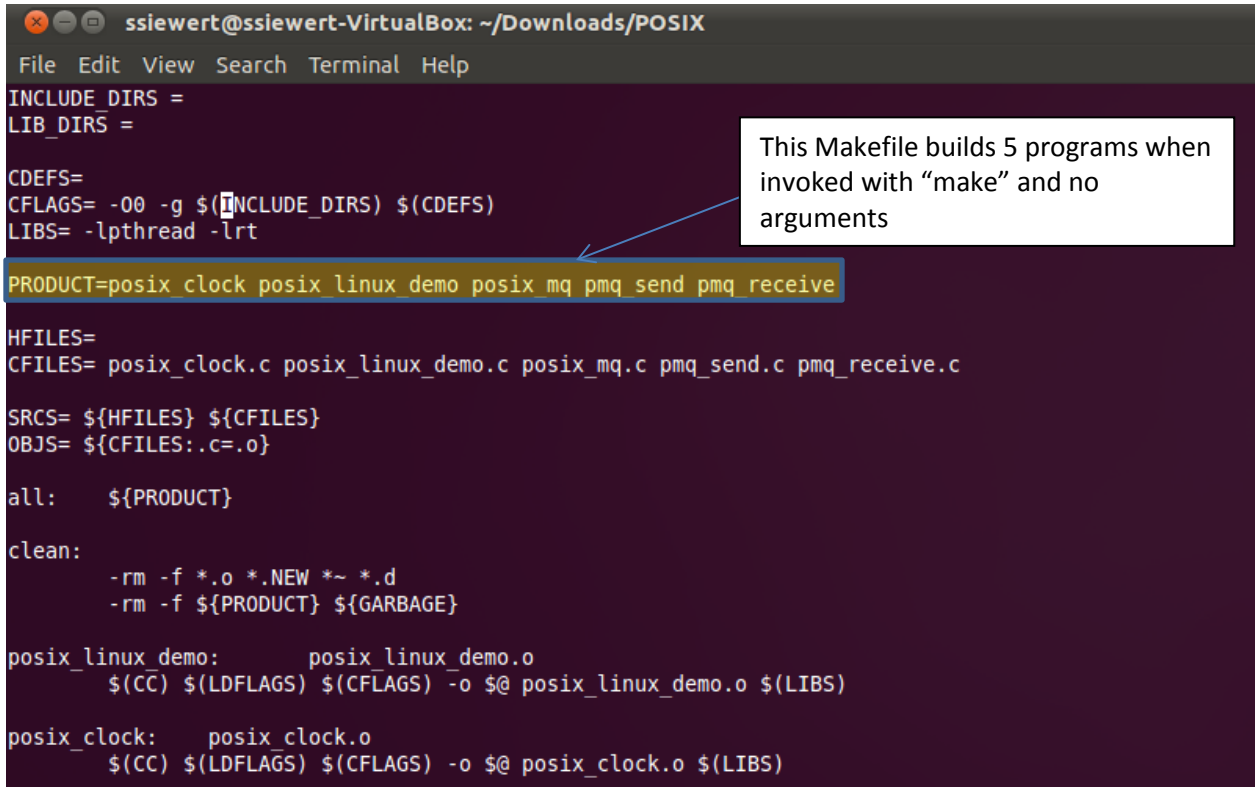
1. First, always start with an example – I have many posted here:

2. Make sure flags are set for no optimization (-O0) and debug symbols (-g) until you have your code working, then you can turn off debug symbols and turn on optimization if you want/need to do so.

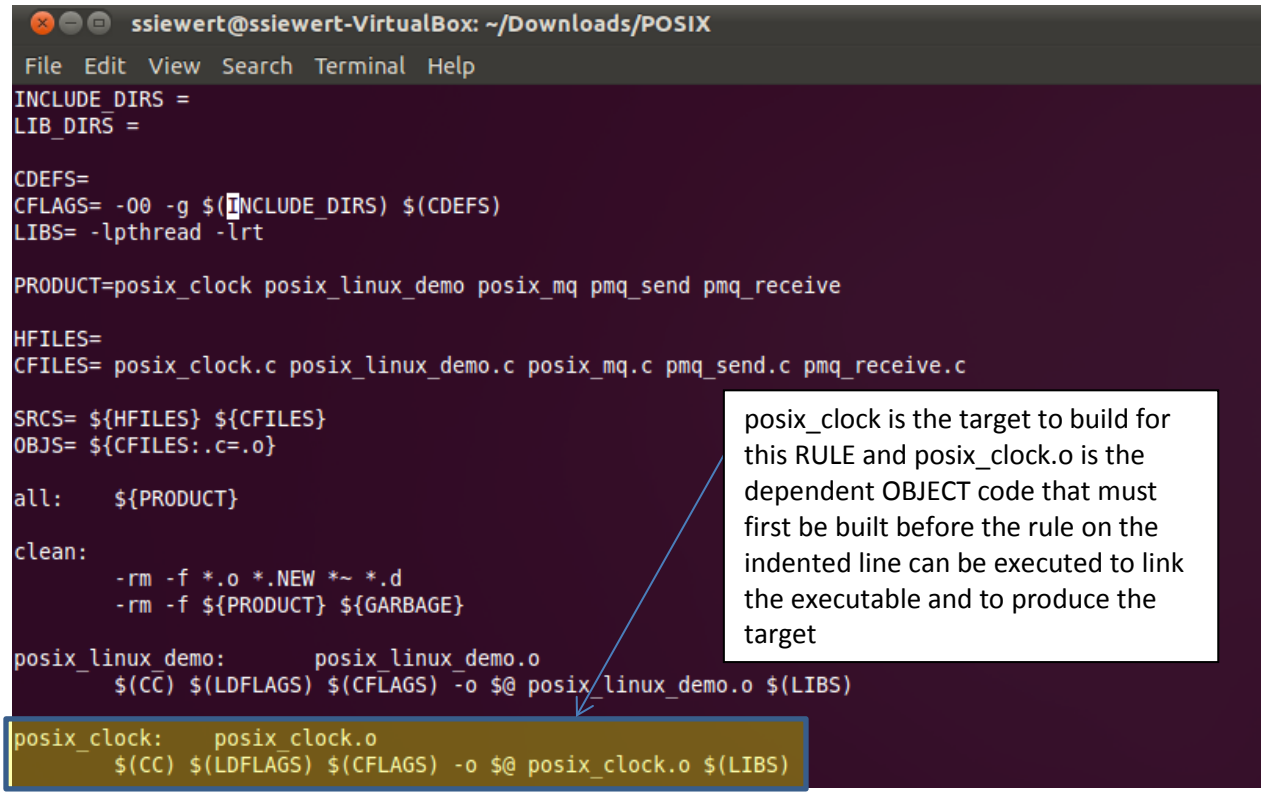
Here's a Makefile that can be executed by simply running "make" in the same directory in which it appears (as long as the name of the file is Makefile) that I will annotate with notes describing in a series of shots of the same file viewed on my system:



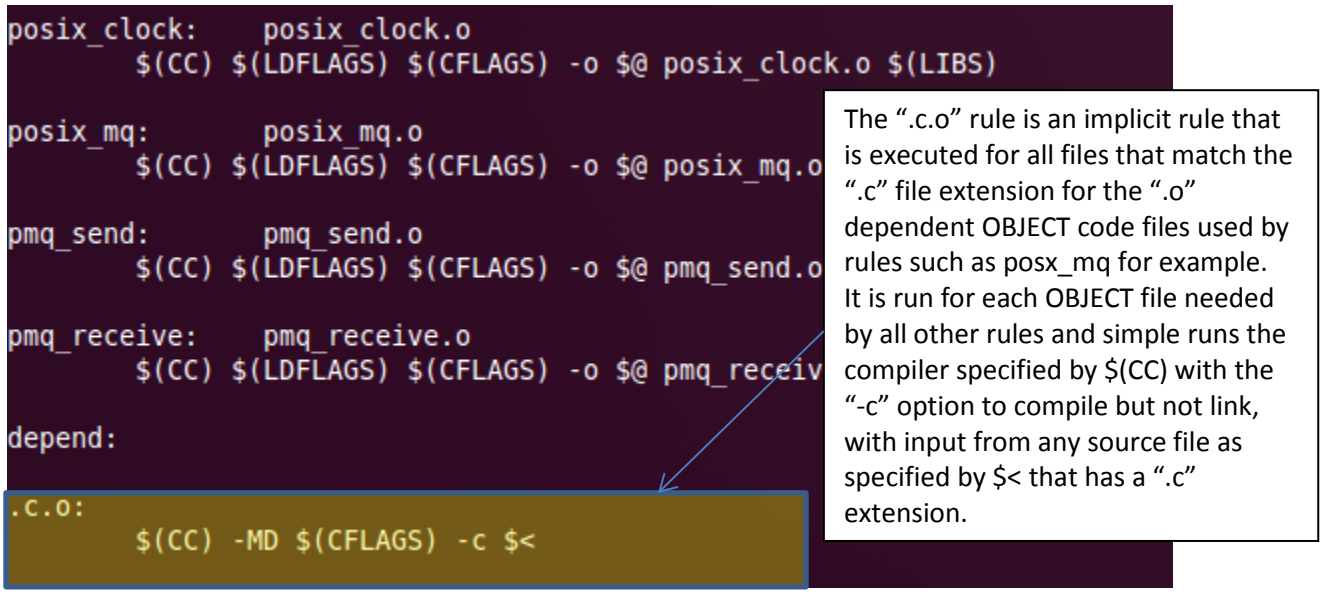
Now, also notice that we need to have an executable name as a build PRODUCT like the pmq_send and pmq_receive programs used in Exercise #2.



Now that the basic definitions have been specified in the Makefile, we must create a target and method for each PRODUCT entry – e.g. let’s look at posix_clock, and executable that this Makefile will build when invoked by “make” and no arguments:



So, to understand how the posix_clock.o dependent OBJECT code is built, we must look at the implicit Dot-C, Dot-Oh rule (".c.o") which tells the make how to derive object code from C source code to implement the rules to build an executable program:

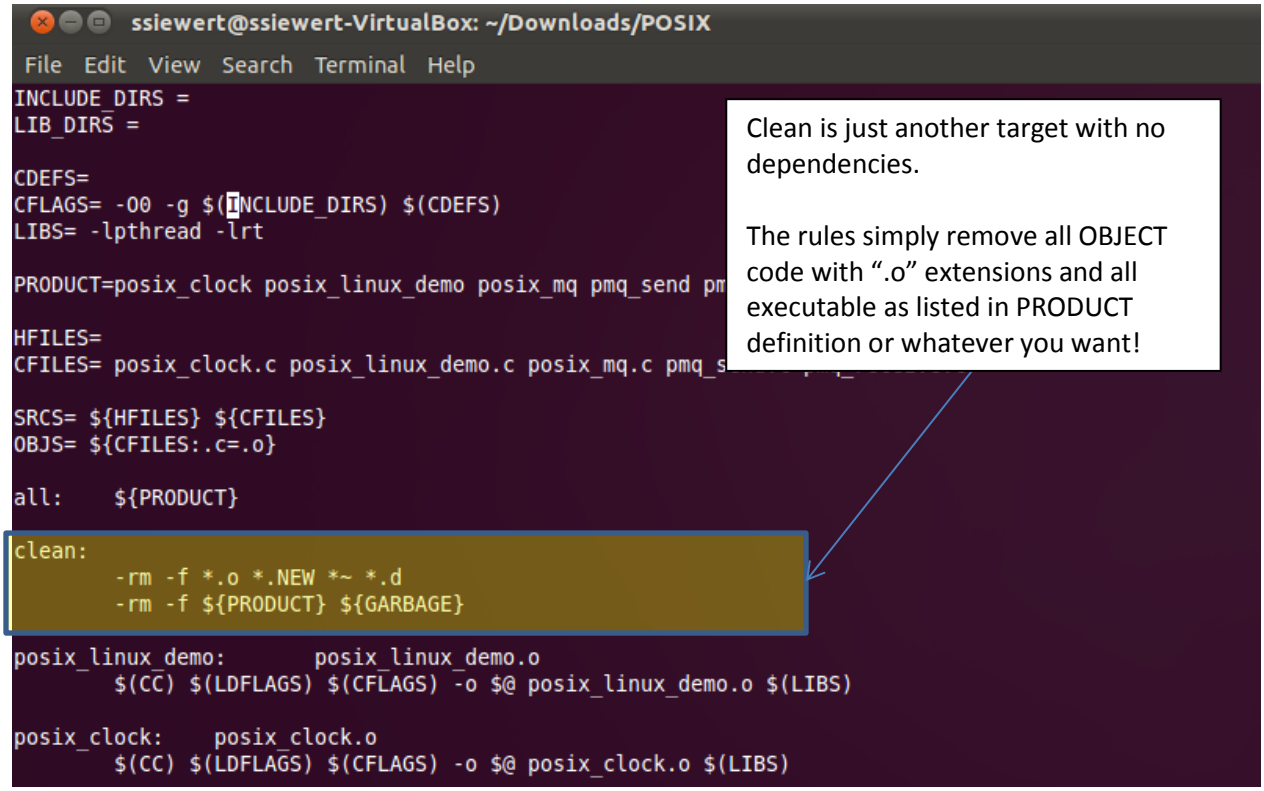


That is pretty much it – all Makefile "make" rules have the format of:

<target>: <dependencies>
<method>

As has been described here by example … If any file date is updated by an edit or by any other means, the next time make is invoked, it will rebuild running all rules for any targets that are out of date (where their dependencies have been modified since the last build).

One final note – it is nice to have a "make clean", so a user can rebuild ALL:

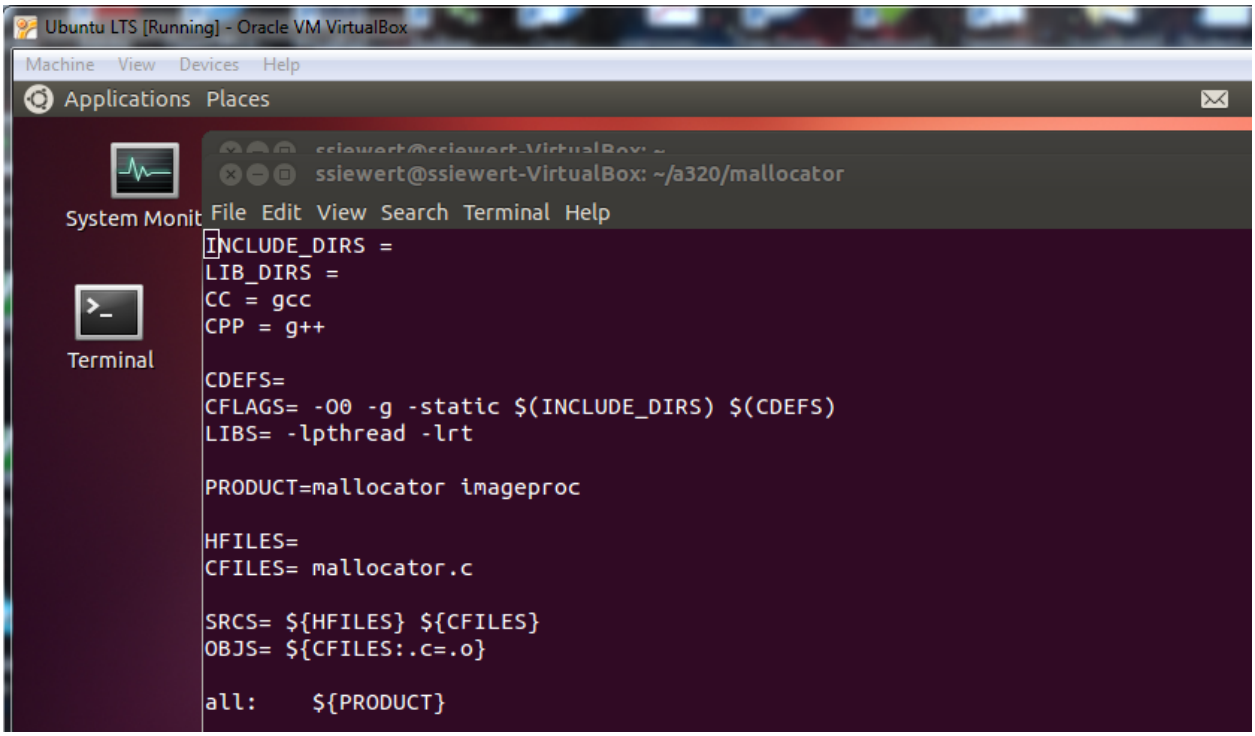


So, to understand how the posix_clock.o dependent OBJECT code is built, we must look at the implicit .c.o rule.

Furthermore, one more tip on Makefiles. Use "make clean" between builds to ensure that you remove stale object code and executables. The make application checks file dates, but for example, if you wanted to rebuild a target with –static (-S) CFLAG to statically link rather than default of dynamic, you'd want to do this (my command history is shown using the "history" command):

1379 cd mallocator/
…
1385 make
1386 ls
1387 objdump -t imageproc > imageproc.dynamic
1388 make clean
1389 vi Makefile
1390 make
1391 objdump -t imageproc > imageproc.static
1392 ls
1393 diff imageproc.dynamic imageproc.static

Note that in the step where I use “vi”, I edit the Makefile and add the –static to the CFLAGS as follows:

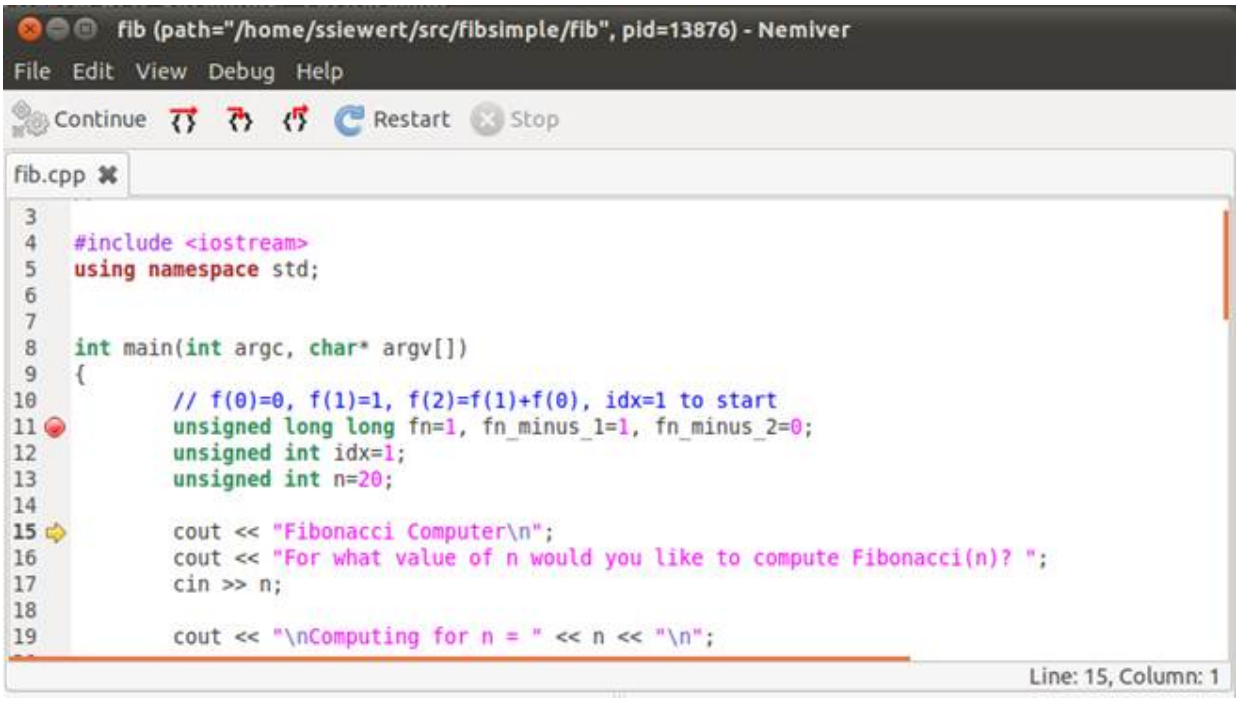


As you become more comfortable with Makefile, you’ll want to modify the flags, rules and targets as needed and do “make clean” and “make” to re-build a project.

Once you have your target executable built with make, I recommend debugging your built executable on the Beagle xM and VB-Linux with:

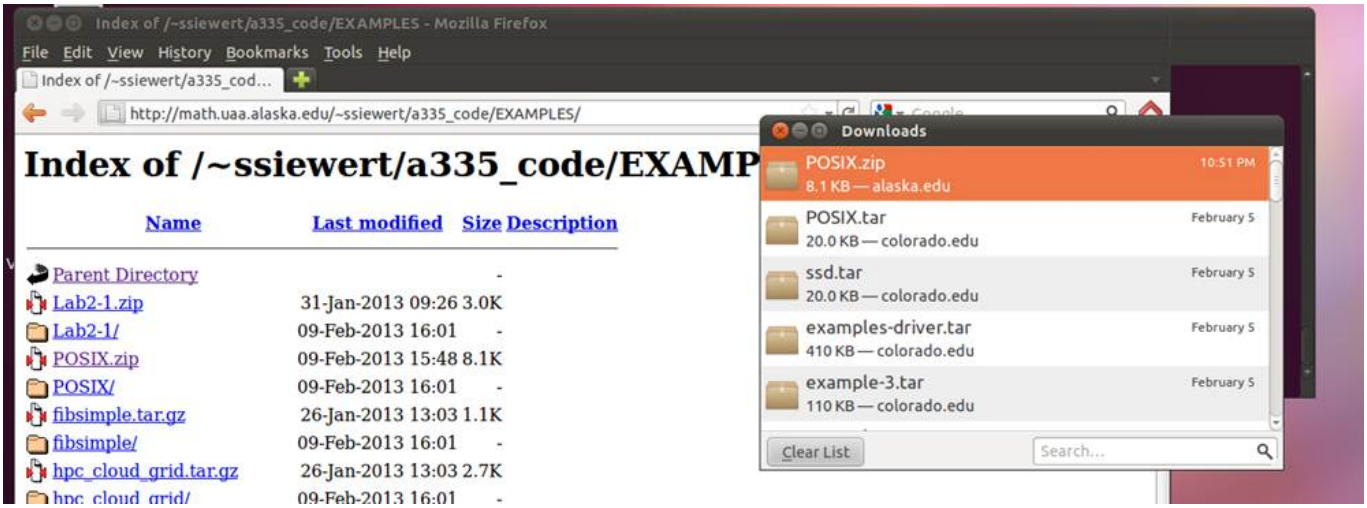
sudo apt-get install nemiver

This is a debugger only and you can load and run code with ***nemiver fib*** for example after your code is built using ***make***. It runs nicely on Beagle xM (remember the Beagle is “like” a cell phone, so it just can’t handle well running all of Eclipse – I tried, and it’s just too slow).

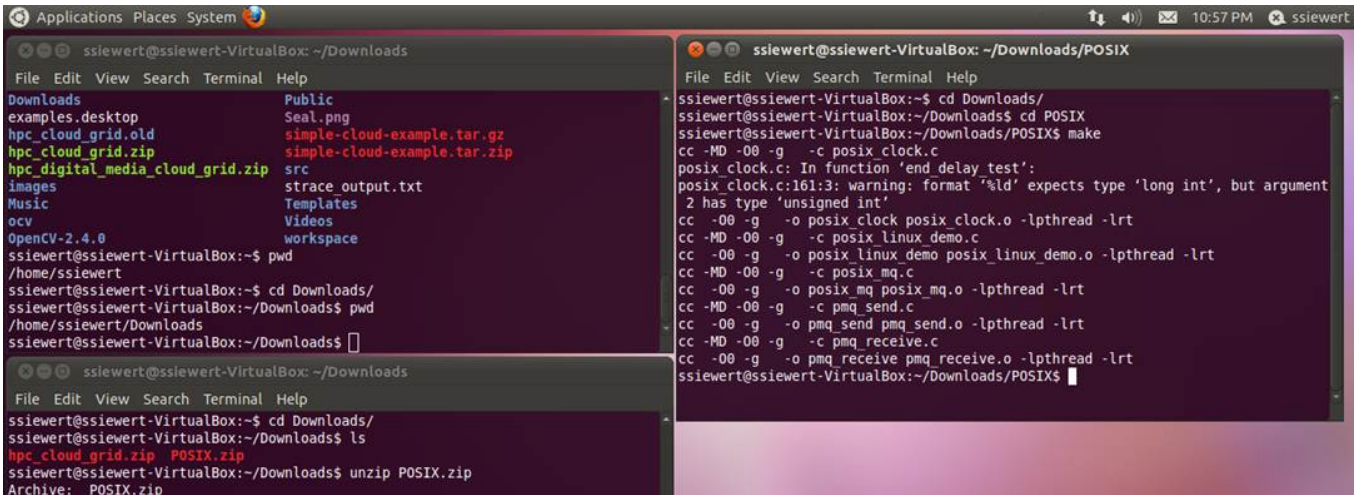


Now, here's some detailed instructions to download, unzip, build, and debug this example:

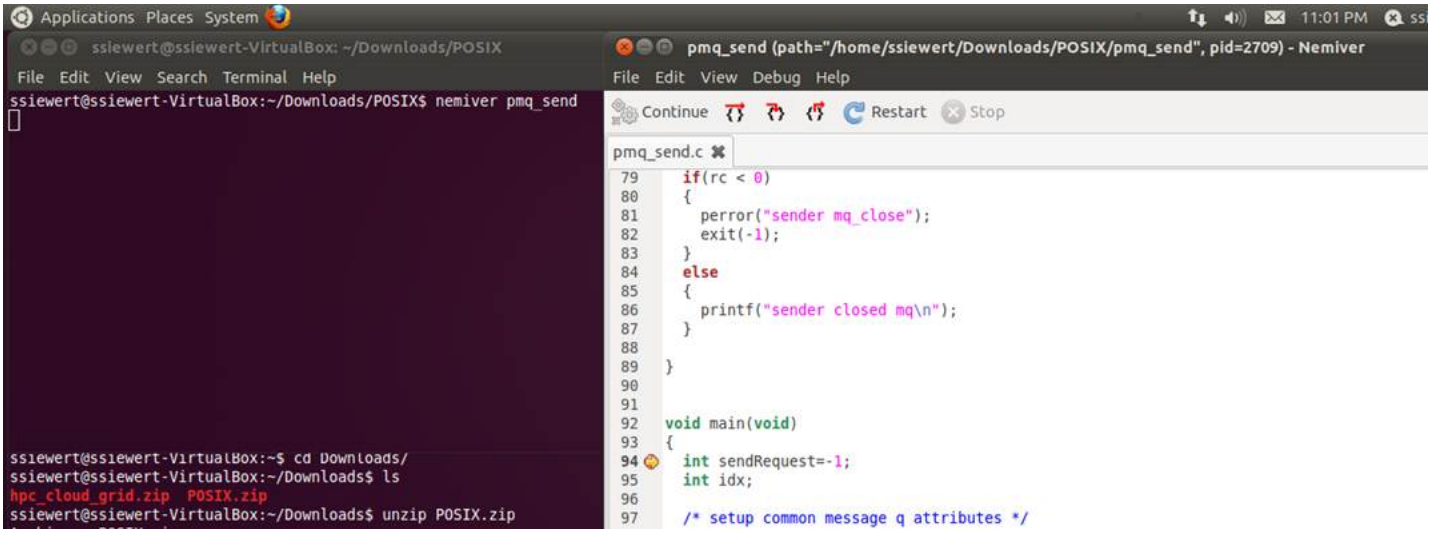
Here are a series of screen shots from download to build and debug – hopefully this helps – if anyone is still stuck, glad to go over in class (downloading, building, and debugging code is an absolutely necessary skill that you should have picked up in Lab #1, but if not, we need to make sure all can do this after this Lab #2 for sure). So, glad to spend the time to make sure we are all good on this:



Now that I have it downloaded, I go to my home directory and I'll just unzip it right in downloads for simplicity and do a "make" to build the examples (note that I went to downloads in the first window, unzipped in the second, and in the third did a make, but all in my /home/faculty/ssiewert/Downloads directory – you should do exactly the same!):



Now, after doing the "make" exactly as I've done above, you should be able to debug with nemiver. Make SURE that you first install nemiver with "sudo apt-get install nemiver". Then just do this:



Some common misconceptions to watch out for:

You can't run a debugger on source code directly, it can only be run in fact on an executable – you must first build the executable from sources with "make". The "-g" option and "-O0" must be set, but in the example I believe I already did this for you.

It's different than Visual Studio, where for example as you know, if the source is not built, it builds it and then starts the debugger – here these are 2 different steps.

Once your C source has been built, you can debug with Nemiver.

To build, all you have to do is type "make" in the same directory where you unzip the example code.

To debug it, here is a screen shot of me doing this now – just simple invocation of the debugger with the built executable: