

Makefile Basics for Linux

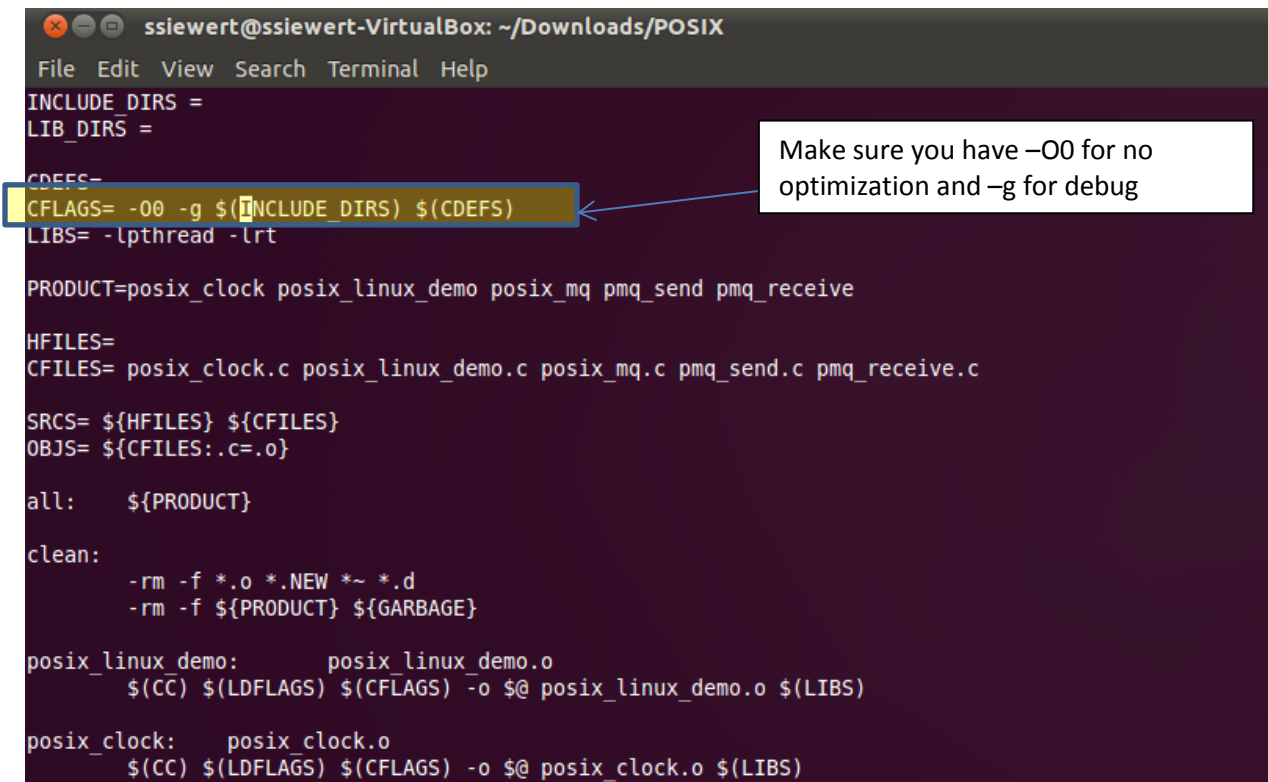
For anyone wanting to really learn make and Makefiles, I highly recommend this book - <http://oreilly.com/catalog/make3/book/index.csp>, but I will attempt here to give you the very basics to get you going. To follow along, download the following code from my UA-A website here - http://math.uaa.alaska.edu/~ssiewert/a335_code/EXAMPLES/POSIX/, 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:



```
ssiewert@ssiewert-VirtualBox: ~/Downloads/POSIX
File Edit View Search Terminal Help
INCLUDE_DIRS =
LIB_DIRS =

CDEFS=
CFLAGS= -O0 -g $(INCLUDE_DIRS) $(CDEFS)
LIBS= -lpthread -lrt

PRODUCT=posix_clock posix_linux_demo posix_mq pmq_send pmq_receive

HFILES=
CFILES= posix_clock.c posix_linux_demo.c posix_mq.c pmq_send.c pmq_receive.c

SRCS= ${HFILES} ${CFILES}
OBS= ${CFILES:.c=.o}

all:  ${PRODUCT}

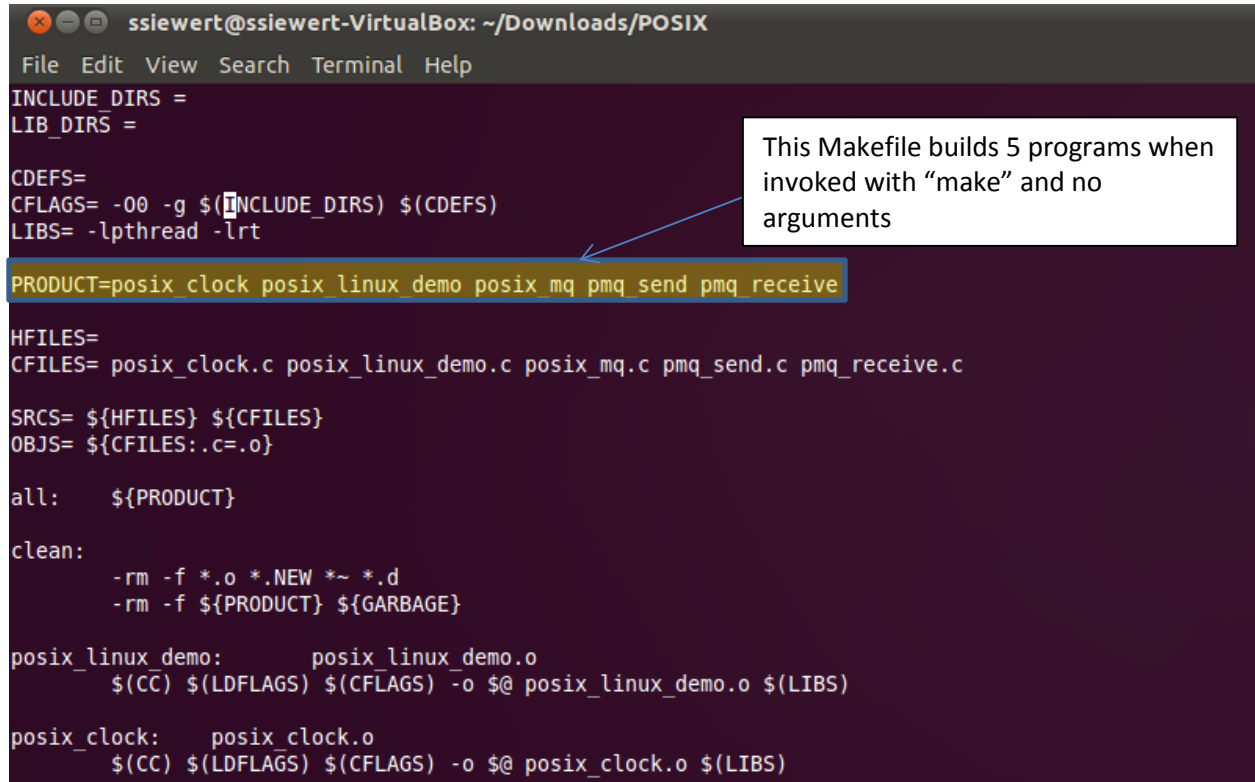
clean:
    -rm -f *.o *.NEW *~ *.d
    -rm -f ${PRODUCT} ${GARBAGE}

posix_linux_demo:    posix_linux_demo.o
    $(CC) $(LD_FLAGS) $(CFLAGS) -o $@ posix_linux_demo.o $(LIBS)

posix_clock:    posix_clock.o
    $(CC) $(LD_FLAGS) $(CFLAGS) -o $@ posix_clock.o $(LIBS)
```

Make sure you have -O0 for no optimization and -g for debug

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.



```
ssiewert@ssiewert-VirtualBox: ~/Downloads/POSIX
File Edit View Search Terminal Help
INCLUDE_DIRS =
LIB_DIRS =

CDEFS=
CFLAGS= -O0 -g $(INCLUDE_DIRS) $(CDEFS)
LIBS= -lpthread -lrt

PRODUCT=posix clock posix linux demo posix mq pmq send pmq receive

HFILES=
CFILES= posix_clock.c posix_linux_demo.c posix_mq.c pmq_send.c pmq_receive.c

SRCS= ${HFILES} ${CFILES}
OBS= ${CFILES:.c=.o}

all:    ${PRODUCT}

clean:
    -rm -f *.o *.NEW *~ *.d
    -rm -f ${PRODUCT} ${GARBAGE}

posix_linux_demo:    posix_linux_demo.o
    $(CC) $(LDFLAGS) $(CFLAGS) -o $@ posix_linux_demo.o $(LIBS)

posix_clock:    posix_clock.o
    $(CC) $(LDFLAGS) $(CFLAGS) -o $@ posix_clock.o $(LIBS)
```

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:

```

ssiewert@ssiewert-VirtualBox: ~/Downloads/POSIX
File Edit View Search Terminal Help
INCLUDE_DIRS =
LIB_DIRS =

CDEFS=
CFLAGS= -O0 -g $(INCLUDE_DIRS) $(CDEFS)
LIBS= -lpthread -lrt

PRODUCT=posix_clock posix_linux_demo posix_mq pmq_send pmq_receive

HFILES=
CFILES= posix_clock.c posix_linux_demo.c posix_mq.c pmq_send.c pmq_receive.c

SRCS= ${HFILES} ${CFILES}
OBJS= ${CFILES:.c=.o}

all:    ${PRODUCT}

clean:
    -rm -f *.o *.NEW *~ *.d
    -rm -f ${PRODUCT} ${GARBAGE}

posix_linux_demo:    posix_linux_demo.o
    $(CC) $(LDFLAGS) $(CFLAGS) -o $@ posix_linux_demo.o $(LIBS)

posix_clock:    posix_clock.o
    $(CC) $(LDFLAGS) $(CFLAGS) -o $@ posix_clock.o $(LIBS)

```

posix_clock is the target to build for this RULE and posix_clock.o is the dependent OBJECT code that must first be built before the rule on the indented line can be executed to link the executable and to produce the target

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:

```

posix_clock:    posix_clock.o
    $(CC) $(LDFLAGS) $(CFLAGS) -o $@ posix_clock.o $(LIBS)

posix_mq:    posix_mq.o
    $(CC) $(LDFLAGS) $(CFLAGS) -o $@ posix_mq.o

pmq_send:    pmq_send.o
    $(CC) $(LDFLAGS) $(CFLAGS) -o $@ pmq_send.o

pmq_receive:    pmq_receive.o
    $(CC) $(LDFLAGS) $(CFLAGS) -o $@ pmq_receive.o

depend:

.c.o:
    $(CC) -MD $(CFLAGS) -c $<

```

The “.c.o” rule is an implicit rule that is executed for all files that match the “.c” file extension for the “.o” dependent OBJECT code files used by rules such as posix_mq for example. It is run for each OBJECT file needed by all other rules and simply runs the compiler specified by \$(CC) with the “-c” option to compile but not link, with input from any source file as specified by \$< that has a “.c” extension.

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

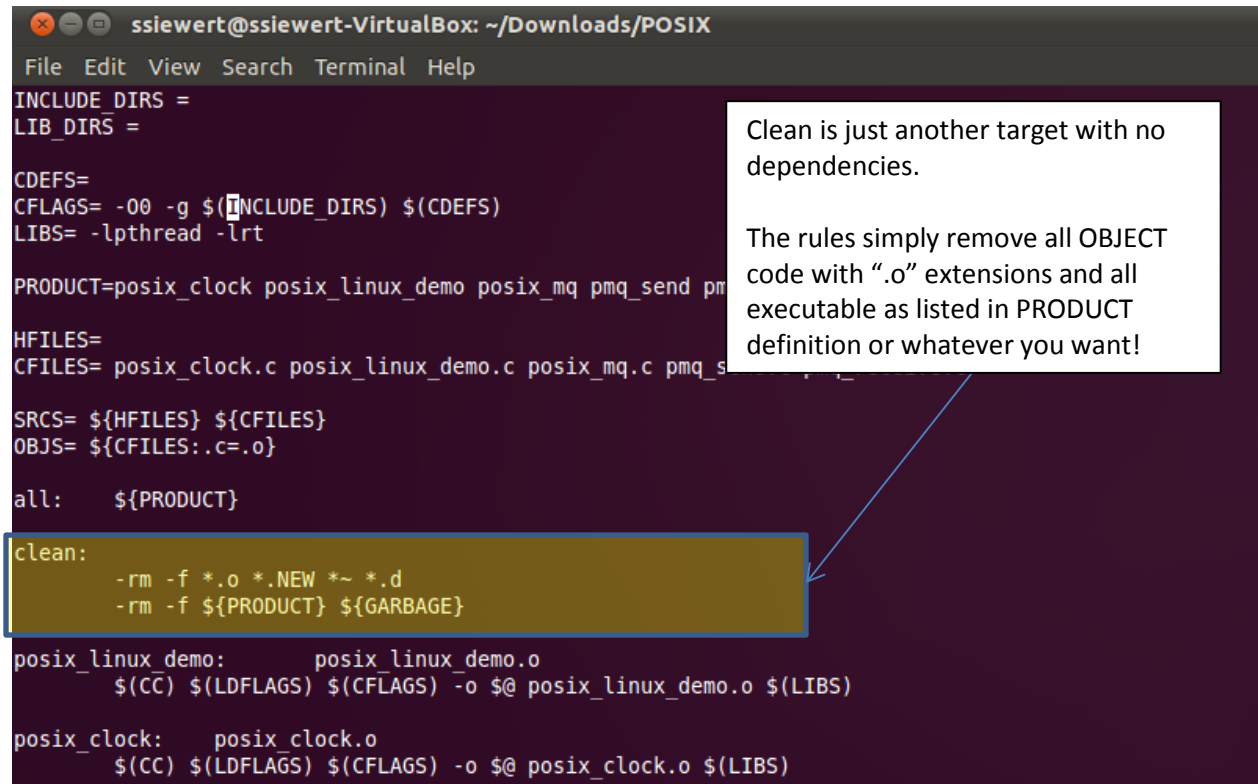
```

<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:



```
ssiewert@ssiewert-VirtualBox: ~/Downloads/POSIX
File Edit View Search Terminal Help
INCLUDE_DIRS =
LIB_DIRS =

CDEFS=
CFLAGS= -O0 -g $(INCLUDE_DIRS) $(CDEFS)
LIBS= -lpthread -lrt

PRODUCT=posix_clock posix_linux_demo posix_mq pmq_send pmq_receive

HFILES=
CFILES= posix_clock.c posix_linux_demo.c posix_mq.c pmq_send.c pmq_receive.c

SRCS= ${HFILES} ${CFILES}
OBS= ${CFILES:.c=.o}

all:    ${PRODUCT}

clean:
    -rm -f *.o *.NEW *~ *.d
    -rm -f ${PRODUCT} ${GARBAGE}

posix_linux_demo:    posix_linux_demo.o
    $(CC) $(LDFLAGS) $(CFLAGS) -o $@ posix_linux_demo.o $(LIBS)

posix_clock:    posix_clock.o
    $(CC) $(LDFLAGS) $(CFLAGS) -o $@ posix_clock.o $(LIBS)
```

Clean is just another target with no dependencies.

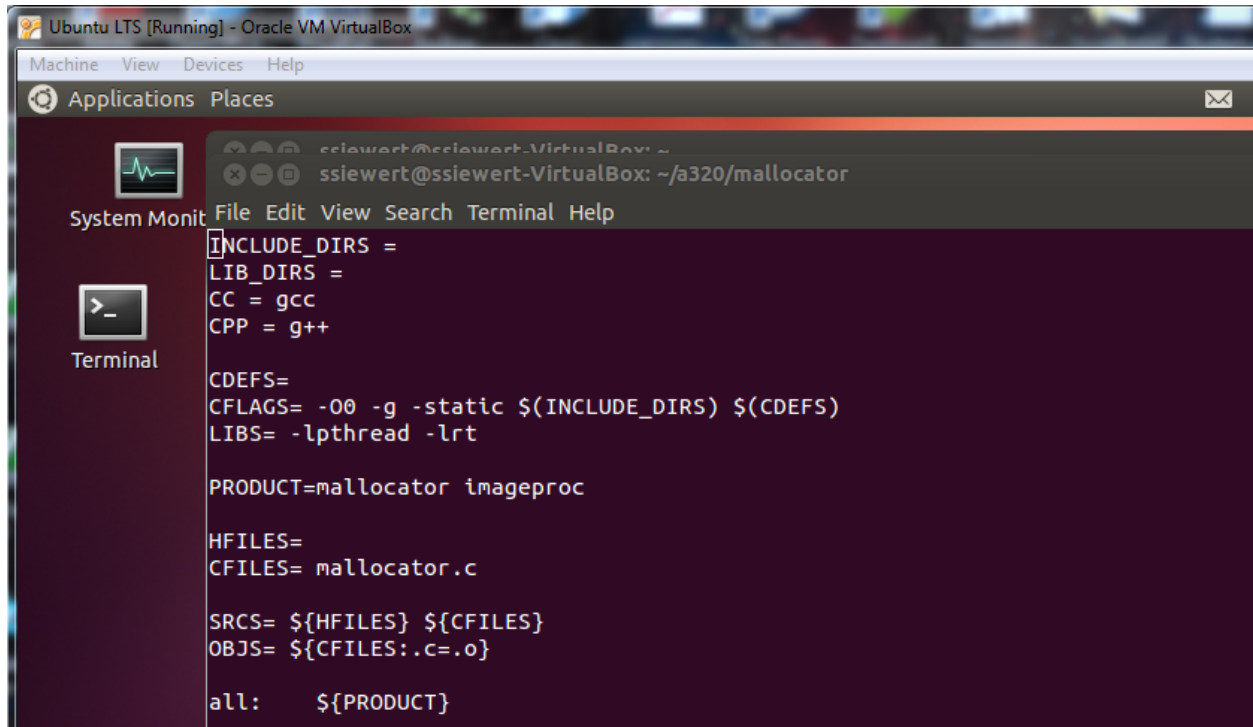
The rules simply remove all OBJECT code with “.o” extensions and all executable as listed in PRODUCT definition or whatever you want!

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 mallocatort/
...
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:

A screenshot of a terminal window titled "Ubuntu LTS [Running] - Oracle VM VirtualBox". The terminal shows a user editing a Makefile. The Makefile content is as follows:

```
ssiewert@ssiewert-VirtualBox: ~/a320/mallocator
File Edit View Search Terminal Help
INCLUDE_DIRS =
LIB_DIRS =
CC = gcc
CPP = g++

CDEFS=
CFLAGS= -O0 -g -static $(INCLUDE_DIRS) $(CDEFS)
LIBS= -lpthread -lrt

PRODUCT=mallocator imageproc

HFILES=
CFILES= mallocator.c

SRCS= ${HFILES} ${CFILES}
OBS= ${CFILES:.c=.o}

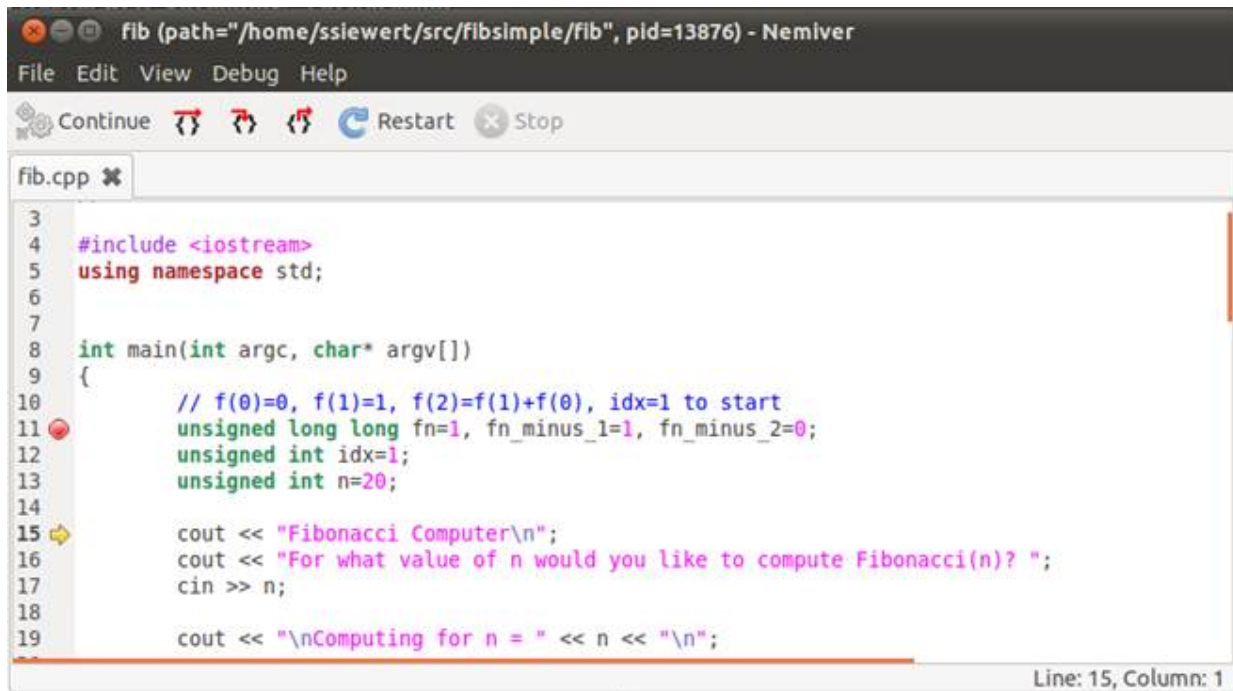
all:  ${PRODUCT}
```

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).



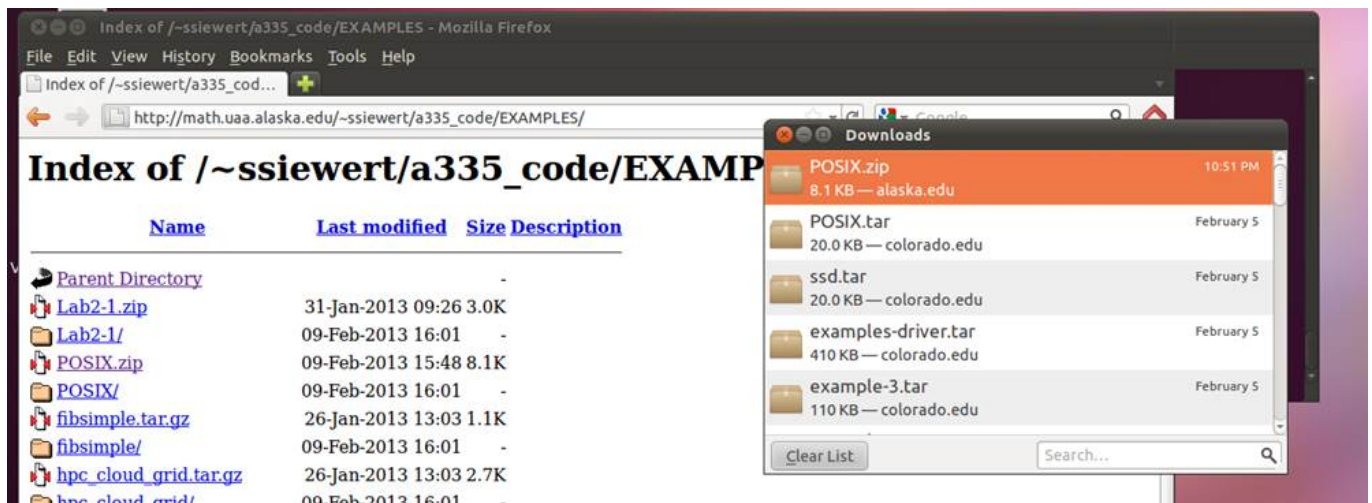
The screenshot shows a debugger window titled "fib (path=\"/home/ssiewert/src/fibsimple/fib\", pid=13876) - Nemiver". The window has a menu bar (File, Edit, View, Debug, Help) and a toolbar with buttons for Continue, Restart, and Stop. The main area displays the source code of "fib.cpp". The code is as follows:

```
3
4 #include <iostream>
5 using namespace std;
6
7
8 int main(int argc, char* argv[])
9 {
10     // f(0)=0, f(1)=1, f(2)=f(1)+f(0), idx=1 to start
11     unsigned long long fn=1, fn_minus_1=1, fn_minus_2=0;
12     unsigned int idx=1;
13     unsigned int n=20;
14
15     cout << "Fibonacci Computer\n";
16     cout << "For what value of n would you like to compute Fibonacci(n)? ";
17     cin >> n;
18
19     cout << "\nComputing for n = " << n << "\n";
```

The status bar at the bottom right indicates "Line: 15, Column: 1".

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!):


```
ssiewert@ssiewert-VirtualBox: ~/Downloads
File Edit View Search Terminal Help
Downloads
examples.desktop      Public
hpc_cloud_grid.old    Seal.png
hpc_cloud_grid.zip     simple-cloud-example.tar.gz
hpc_digital_media_cloud_grid.zip  simple-cloud-example.tar.zip
images                src
Music                  strace_output.txt
ocv                     Templates
OpenCV-2.4.0           Videos
workspace              workspace
ssiewert@ssiewert-VirtualBox:~$ pwd
ssiewert@ssiewert-VirtualBox:~$ cd Downloads/
ssiewert@ssiewert-VirtualBox:~/Downloads$ pwd
/home/ssiewert/Downloads
ssiewert@ssiewert-VirtualBox:~/Downloads$

ssiewert@ssiewert-VirtualBox: ~/Downloads
File Edit View Search Terminal Help
ssiewert@ssiewert-VirtualBox:~$ cd Downloads/
ssiewert@ssiewert-VirtualBox:~/Downloads$ ls
hpc_cloud_grid.zip  POSIX.zip
ssiewert@ssiewert-VirtualBox:~/Downloads$ unzip POSIX.zip
Archive:  POSIX.zip
```

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:

```
ssiewert@ssiewert-VirtualBox: ~/Downloads/POSIX
File Edit View Search Terminal Help
ssiewert@ssiewert-VirtualBox:~/Downloads/POSIX$ nemiver pmq_send

```

Some common misconceptions to watch out for:

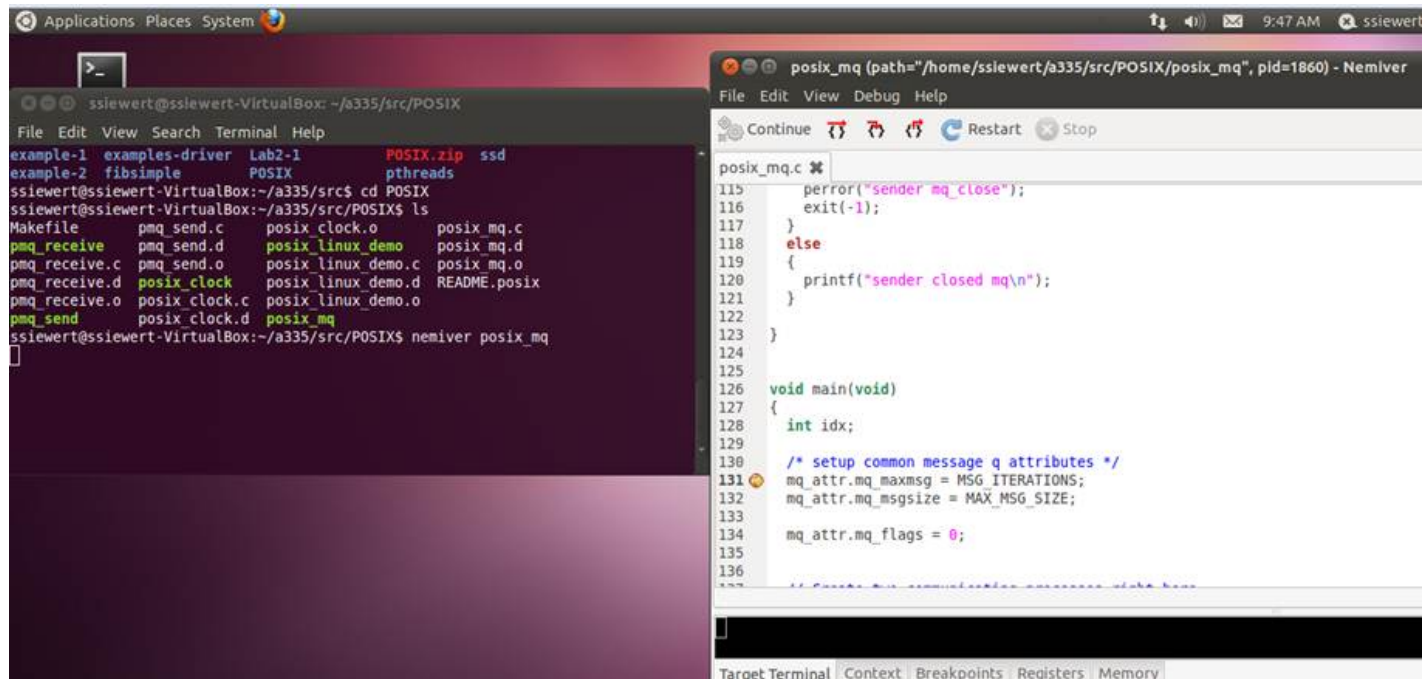
You can’t run a debugger on source code directly, it can only be run in fact on an executable – you must first build the executable from sources with “make”. The “-g” option and “-O0” must be set, but in the example I believe I already did this for you.

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:



The screenshot shows a terminal window on the left and a debugger window on the right. The terminal window displays the following commands and output:

```
ssiewert@ssiewert-VirtualBox: ~/a335/src/POSIX
example-1 examples-driver Lab2-1 POSIX.zip ssd
example-2 fibsimple POSIX pthreads
ssiewert@ssiewert-VirtualBox:~/a335/src$ cd POSIX
ssiewert@ssiewert-VirtualBox:~/a335/src/POSIX$ ls
Makefile      pmq_send.c    posix_clock.o  posix_mq.c
pmq_receive   pmq_send.d    posix_linux_demo.o  posix_mq.d
pmq_receive.c pmq_send.o    posix_linux_demo.c  posix_mq.o
pmq_receive.d posix_clock   posix_linux_demo.d  README.posix
pmq_receive.o posix_clock.c posix_linux_demo.o
pmq_send      posix_clock.d posix_mq

ssiewert@ssiewert-VirtualBox:~/a335/src/POSIX$ nemiver posix_mq
```

The debugger window, titled "posix_mq (path="/home/ssiewert/a335/src/POSIX/posix_mq", pid=1860) - Nemiver", shows the source code of "posix_mq.c". The code includes a function "main" that sets up a message queue and a loop that sends and receives messages. The debugger is currently paused at line 131, which is the first line of the loop.

```
115 perror("sender mq close");
116 exit(-1);
117 }
118 else
119 {
120     printf("sender closed mq\n");
121 }
122 }
123 }
124
125
126 void main(void)
127 {
128     int idx;
129
130     /* setup common message q attributes */
131     mq_attr.mq_maxmsg = MSG_ITERATIONS;
132     mq_attr.mq_msgsize = MAX_MSG_SIZE;
133     mq_attr.mq_flags = 0;
134
135
136
```