Running your first Linux Program

This document describes how edit, compile, link, and run your first linux program using:

- Gnome – a nice graphical user interface desktop that runs on top of X-Windows
- X-term – a command line interface to a linux shell
- Emacs – a popular ide for linux developers
- g++ - a C++ compiler
- Make – the code dependency manager for linux developers

Start by logging onto your Linux workstation. Enter your userid and password and you should see a screen that looks like this:

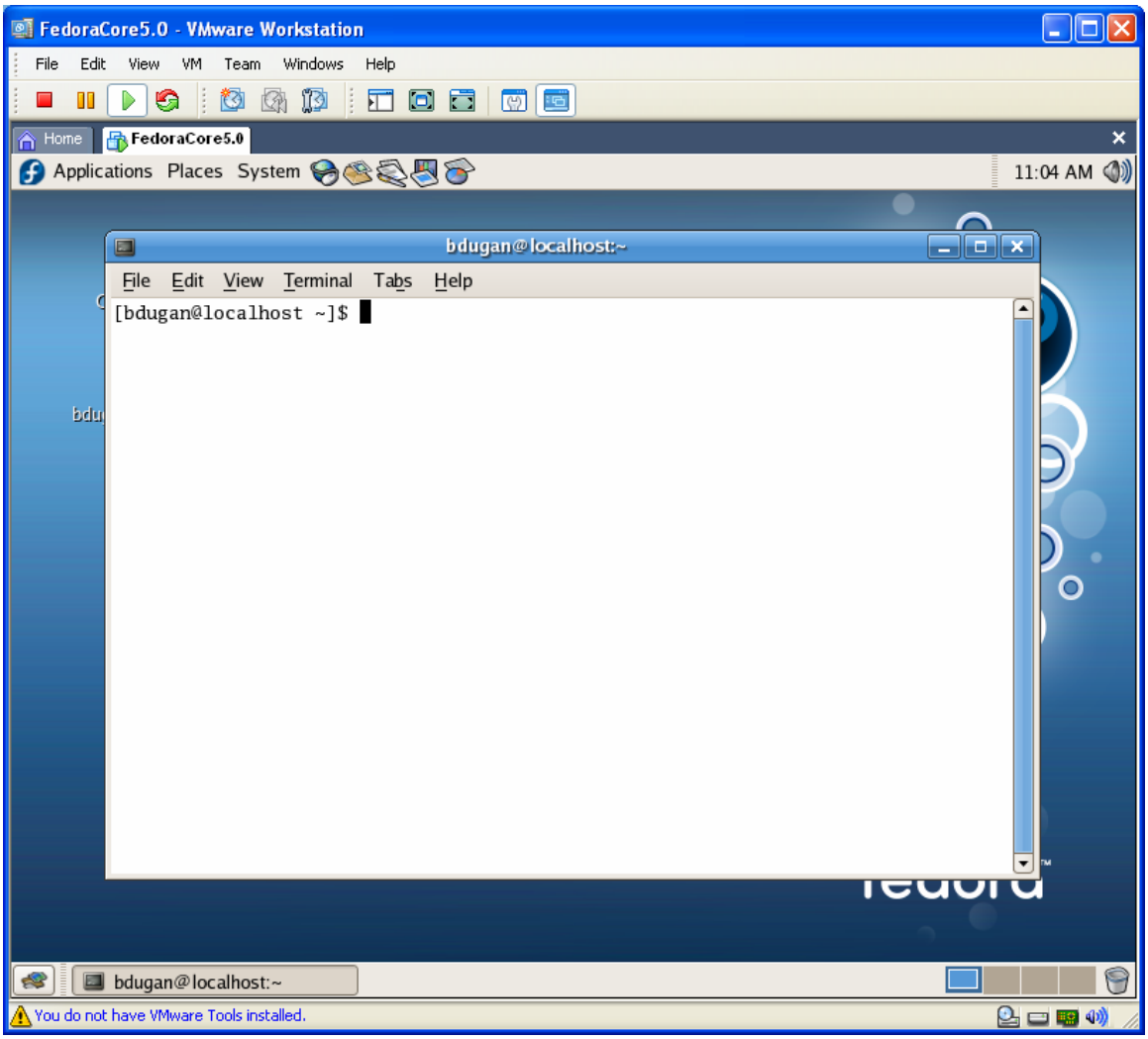
Welcome to LINUX! It looks similar to Windows, and much of your experience with Windows will help you navigate around the graphical user interface. If you click on Applications drop down at the top of the screen, you'll see a set of options similar to the

“Start” button in Windows.

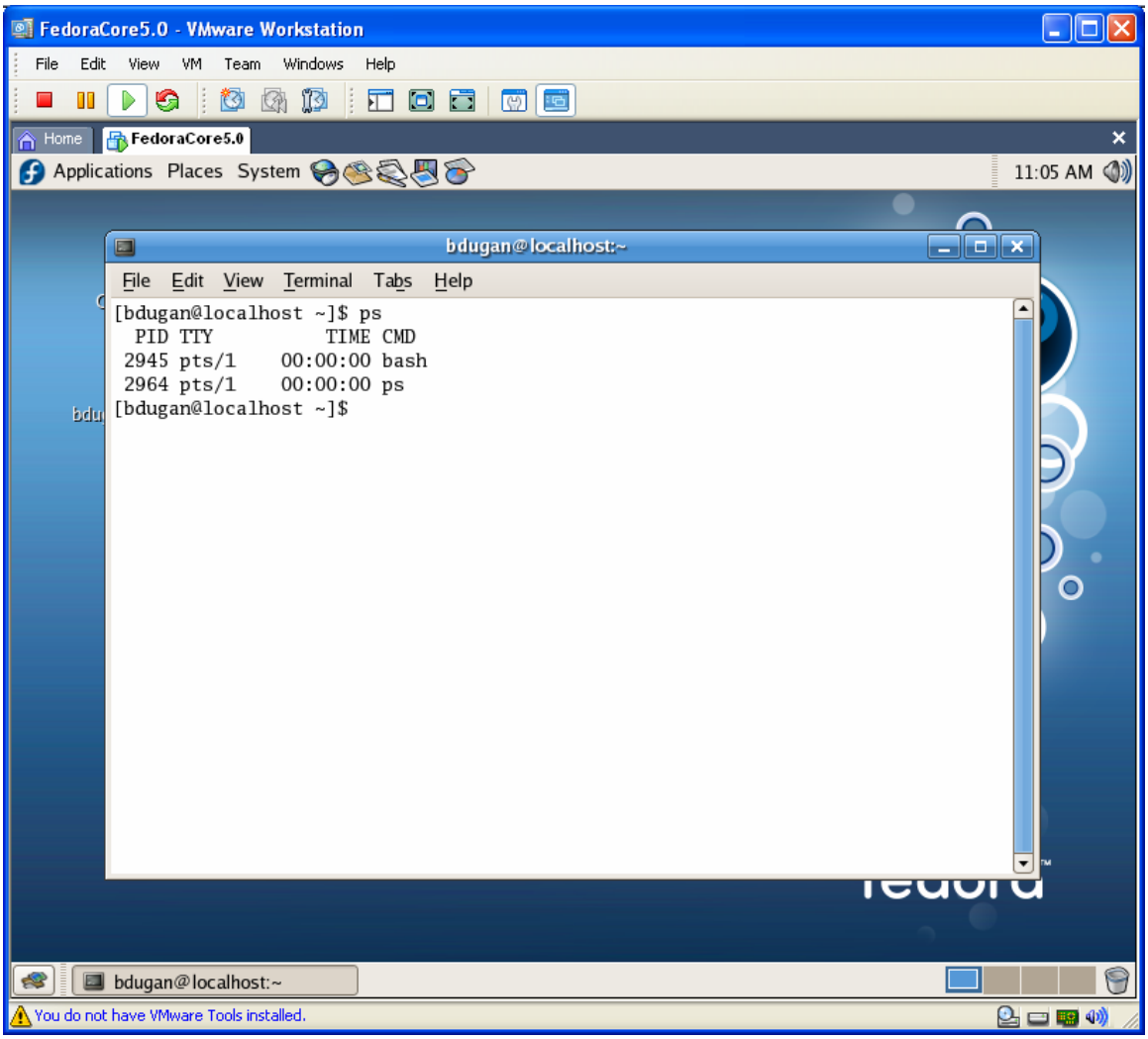
Take some time to explore the programs and the task bar. One thing you should try to figure out right away is how to bring up a web browser. One of the popular Linux web browsers is called Firefox.

OK. Let’s get our hands dirty. Linux is based on Unix which is OLD. Unix die hards don’t always see a need for fancy graphics, it gets in the way of getting important things done. We’re going to bring up a character based interface to the operating system called a “shell”.

Click Select Accessories->Terminal to bring up a terminal window. You should see a terminal window (it’s called an “x-term”) appear. It looks a lot like a DOS command prompt window:



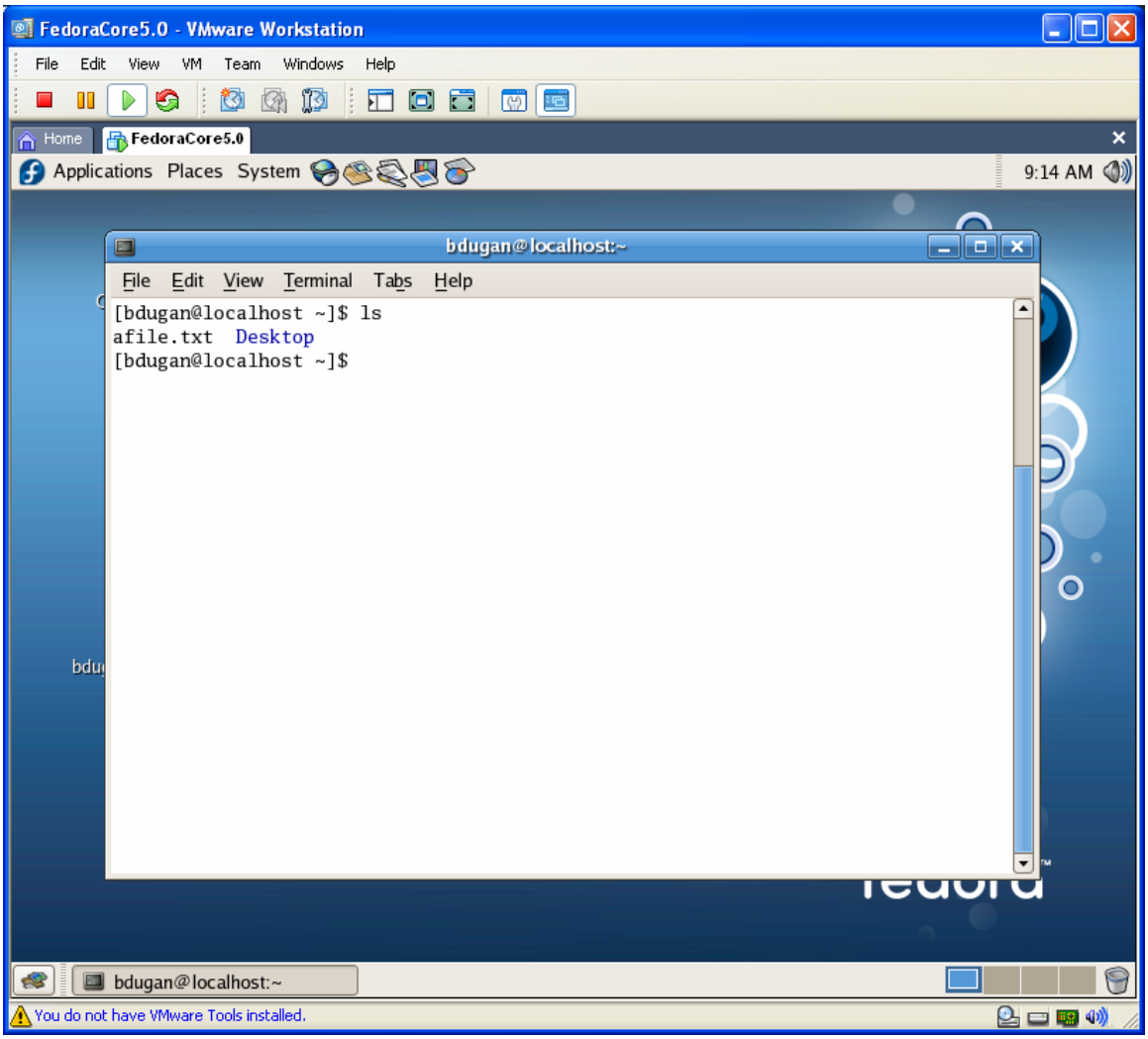
Type the shell command “ps” which stands for “process status”. This will give a list of processes that are under the control of the shell you just created. You should see two of these:



The first process is “bash”, that’s the name of the shell you are running inside the x-term. The second process is the “ps” command itself..

Now type “ls”. This stands for “list directory contents”. It is similar to the DOS “dir” command. When you type “ls” without any arguments, you are listing files that are in shell’s current directory. Notice filenames and directories appear in different colors. I have a directory called “os” which appears in blue. I have a file called “first.db” which appears in white. I have some other files which appear in red.

At the moment, the files I’m looking at are in my home director which happens to be the default directory you are placed in when you log into the system. To find out your current directory in a shell, type “pwd”. You should see “/home/<username>” where username is the name of the user you logged in as. My userid is “bdugan”:



Now it's time to create your own directory so we can get started programming. Type in the command:

mkdir operatingsystems

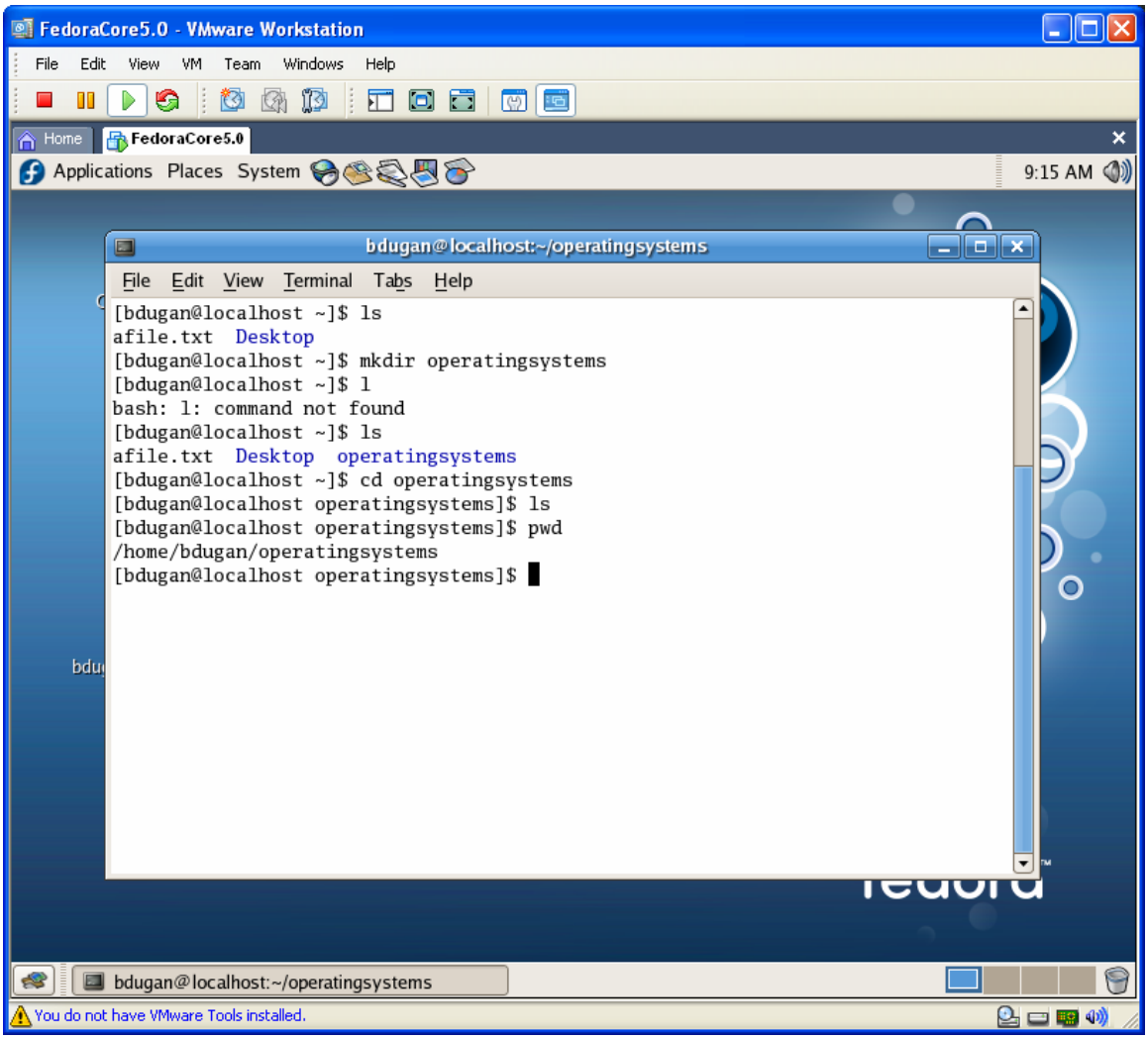
Now type "ls" to verify that the directory exists.

Now change to this new directory:

cd operatingsystems

Type "ls" and you will see that there are no files in this directory.

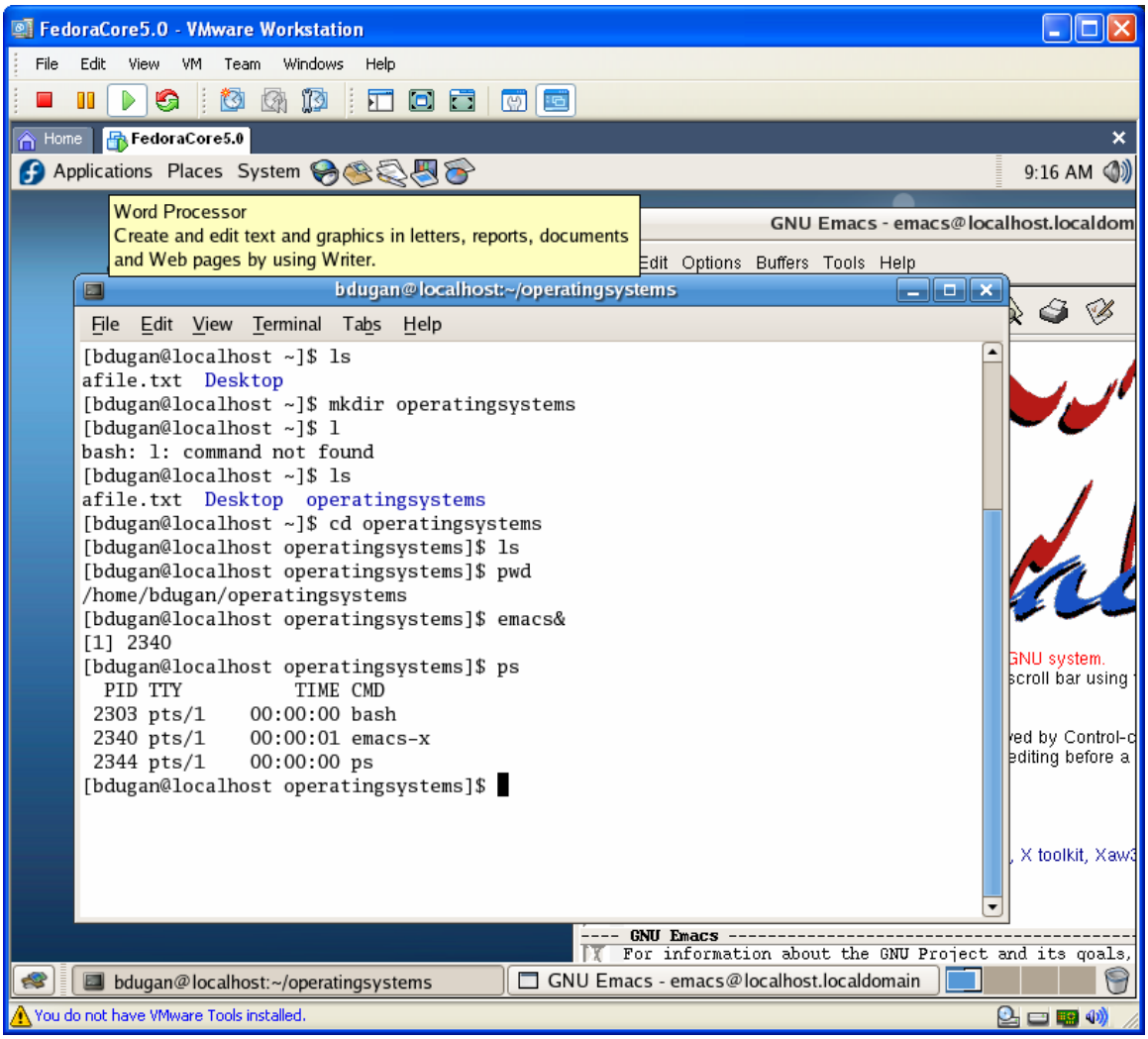
Type "pwd" to verify that you are in the operating systems directory



Now it's time to bring up the editor you'll be using for your programs in this class. Type the following command:

emacs&

Note the "&" that appears after the command. The "&" says to the shell, "I want you to execute this command as a new process, and I also want you to continue to provide the user with a shell prompt". Move the editor out of the way with the mouse for a second and type "ps". You should emacs as one of the processes that is running under the control of the bash shell:

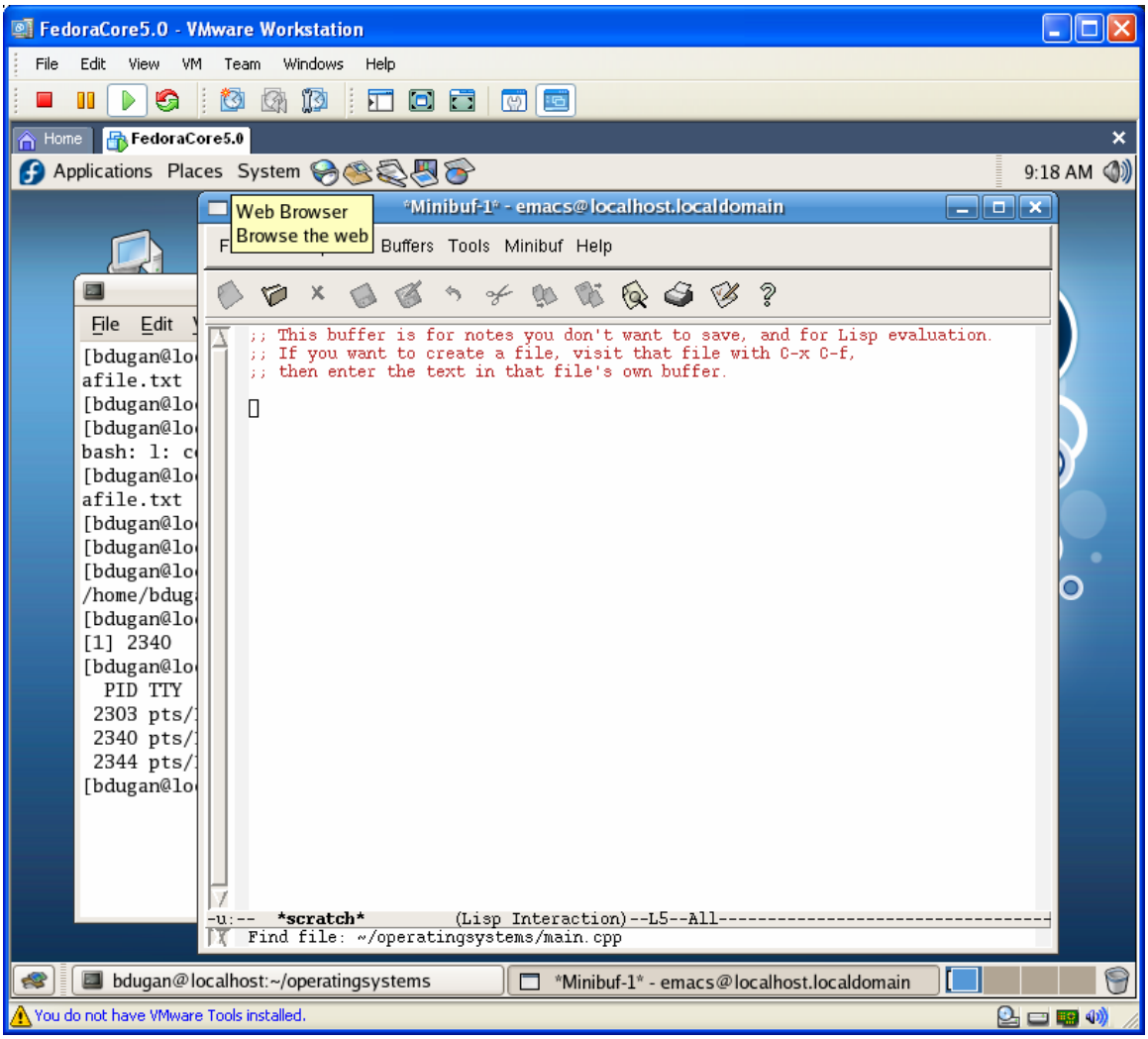


Now drag the emacs back so you can start working with it. Emacs works on "buffers" which are sort of like files. Buffers become files when they are saved to disk. You can use the graphical user interface provided by emacs to perform many editing activities, but emacs has an ancient lineage that pre-dates GUIs. This ancestory means that the editor will also work with just keyboard commands.

Let's start by creating the files you'll need for your "Hello World" program. To create a file, make sure that the editor is active by clicking on it and type the following sequence:

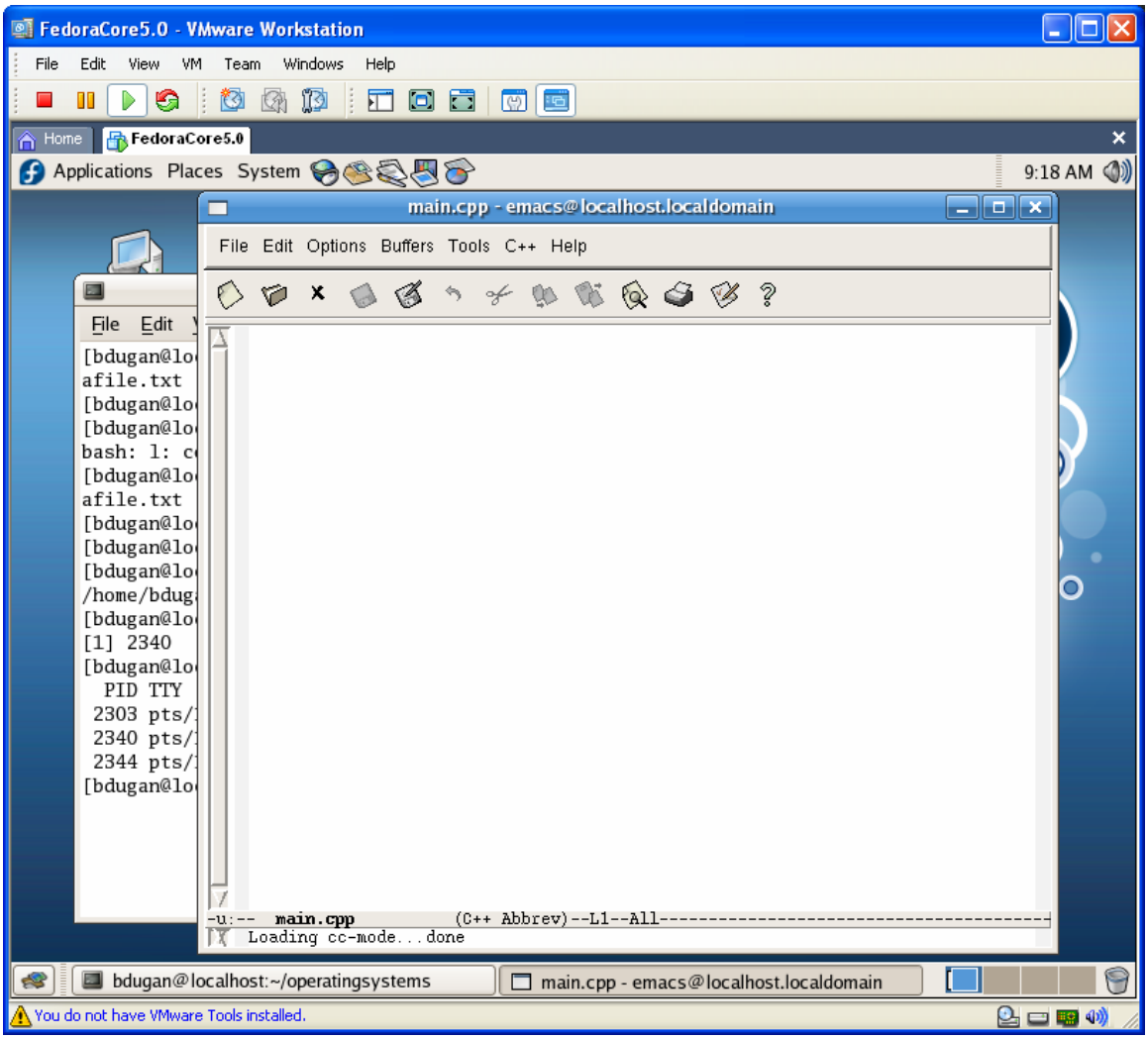
<CTRL>-x <CTRL>-f

Hold the <CTRL> key down and at the same time press the "x" key. Then hold the <CTRL> key down and press the "f" key. You should see something like this:



emacs is prompting you to find a file in the subdirectory "~/operatingsystems". That "~" is a special character which in unix parlance represents your home directory… it is the equivalent of "/home/<username>" or in my case "/home/bdugan". We don't want to find a file, we want to create a new one, so type:

main.cpp <ENTER>



Notice that emacs has entered “cc-mode” and placed a C++ drop down item in the menu bar at the top of the editor. Emacs is ready for you to start writing some C++ code. The cursor is now inside the main window of the editor and you are ready to start writing your first program!

Most of the editing keys on your keyboard will work in emacs as you’d expect them too. For example, when you press the <BACKSPACE> key, you will delete the character to the left of the cursor and move the cursor back one space.

Emacs also has some built in keyboard sequences that do the same thing:

- <CTRL>-f – move cursor forward one character
- <CTRL>-b – move cursor backward one character
- <CTRL>-d – delete character at cursor position, slide text to right of cursor postion one character to left
- <CTRL>-n - move cursor one line down
- <CTRL>-p – move cursor one line up
- <CTRL>-g – whenever you get in trouble, type this a couple of times to get back to the main editing buffer

Why would you EVER want to learn these sequences? As it turns out there's a pretty good reason. Emacs is ubiquitous. It runs on almost every computer hardware platform on the planet. Every computer has a slightly different keyboard. You may not be guaranteed, for example, that a computer has an <UPARROW> key! Most computers have the <CTRL> key and a "p" key however.

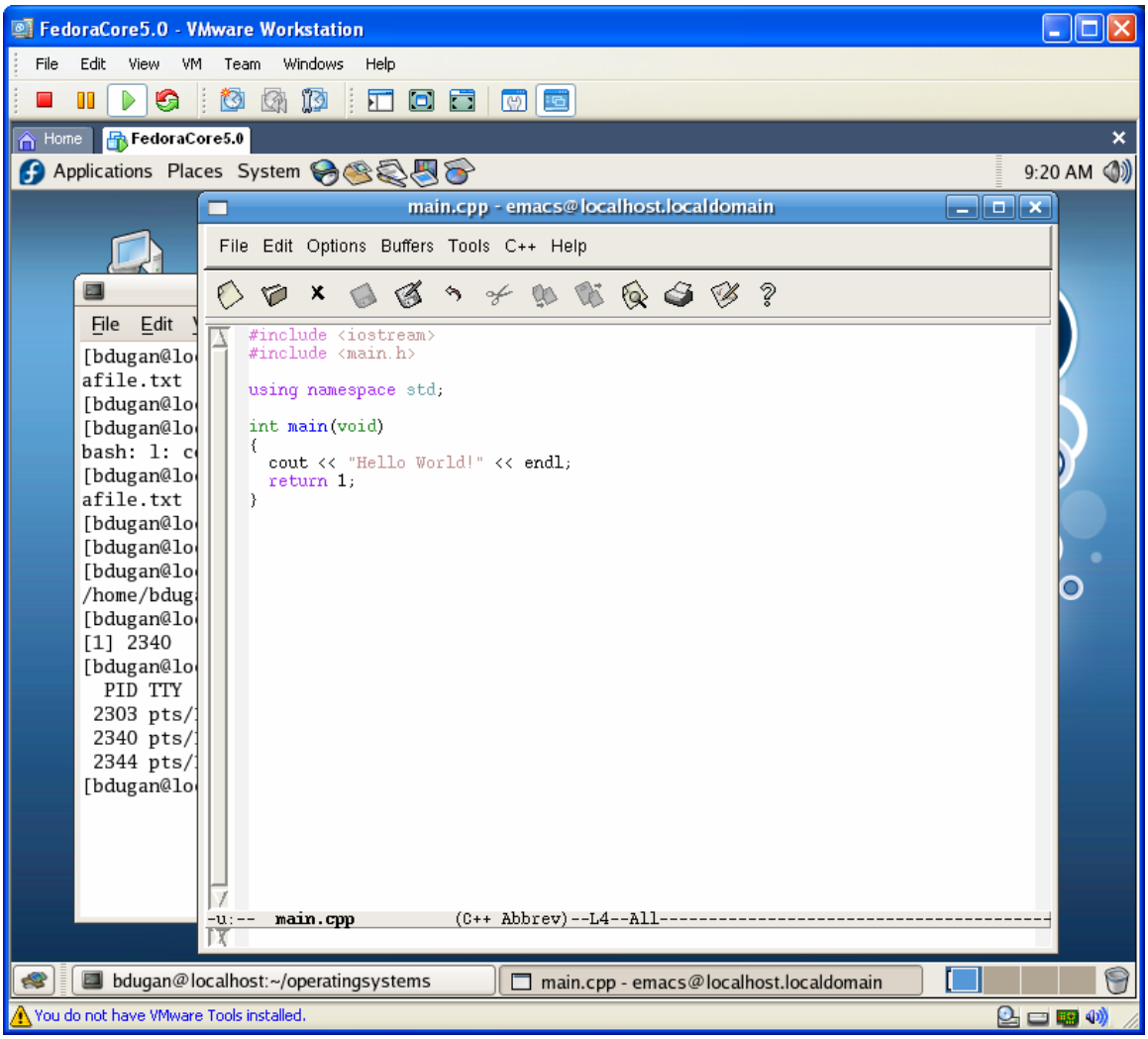
Let's write our first program:

```
#include <iostreams>
#include <main.h>

using namespace std;

int main(void)
{
        cout << "Hello World!" << endl;
        return 1;
}
```

It should look something like this in your editor:



Save the file by issuing the command:

<CTRL>-x s

OK. Now create the “main.h” file and add the following comment:

```
// Nothing goes here yet…
```

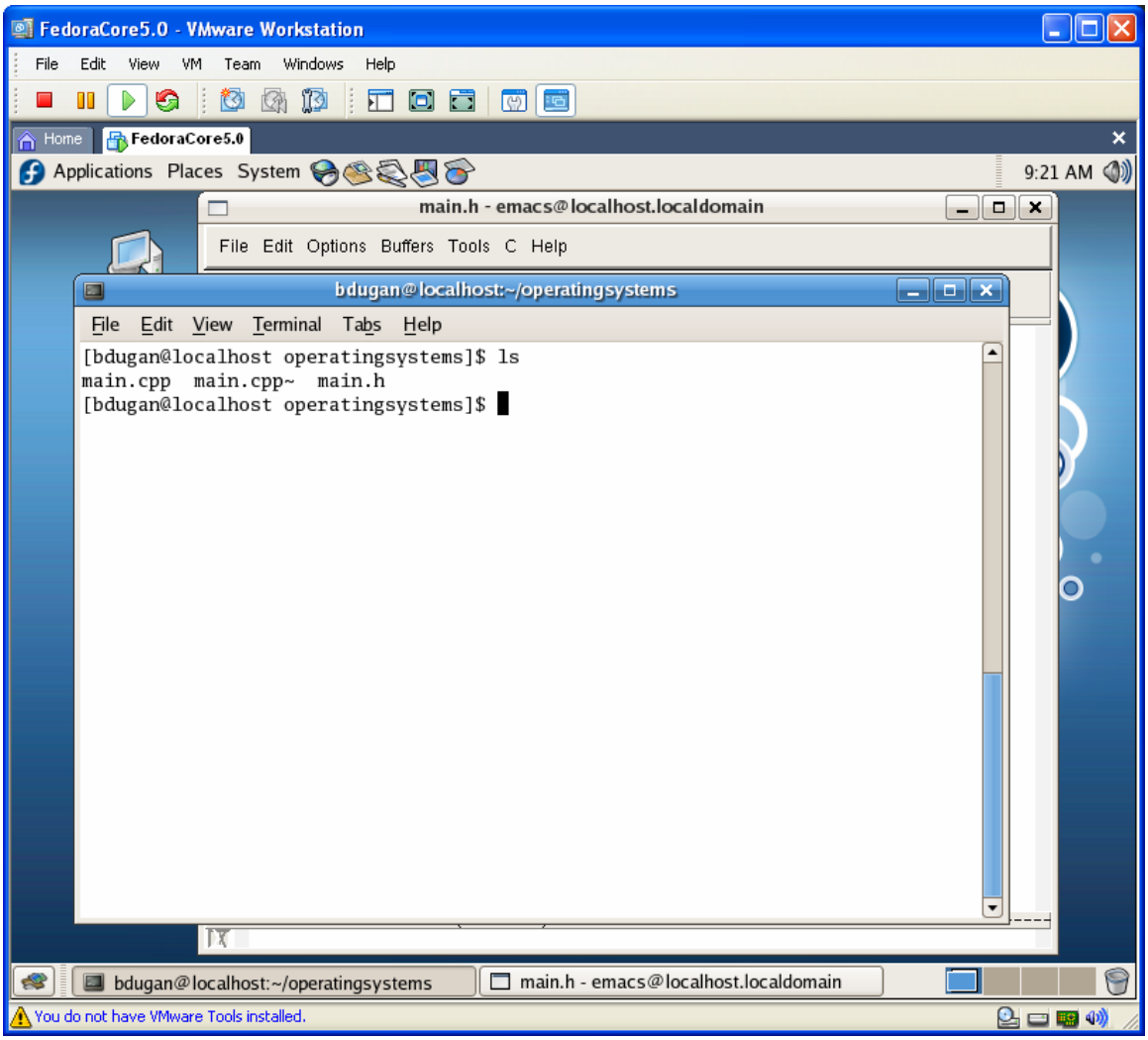
Then save the file.

Compile, Link, and Run

Now you are ready to compile, link, and run your program.

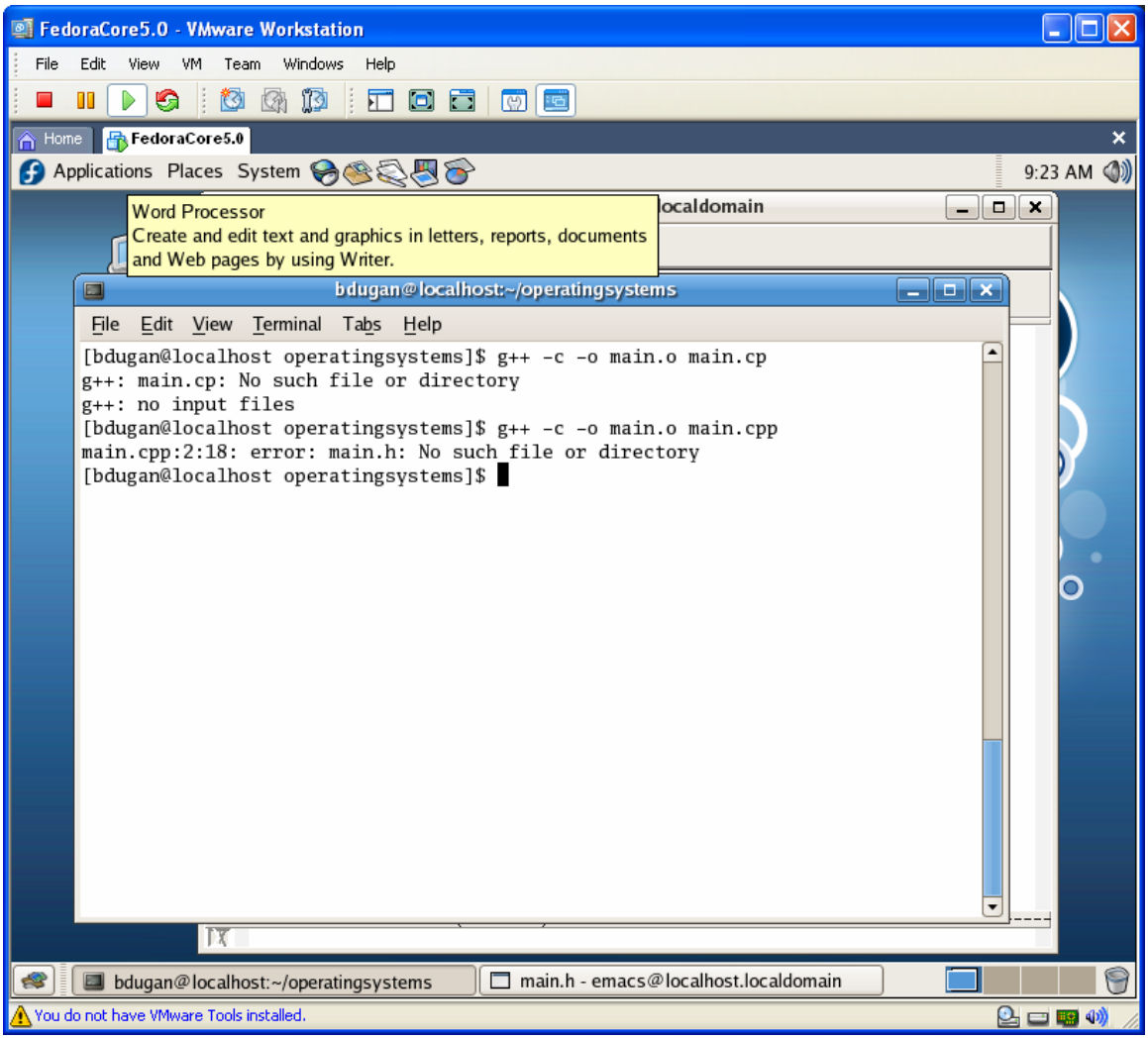
First, we’ll do this by hand. Then we’ll automate everything using a makefile.

Leave the editor and click on the x-term window you created earlier. Type “ls” to verify that you have added two new files to the “operatingsystems” subdirectory:



Now invoke the g++ compiler to compile your program:

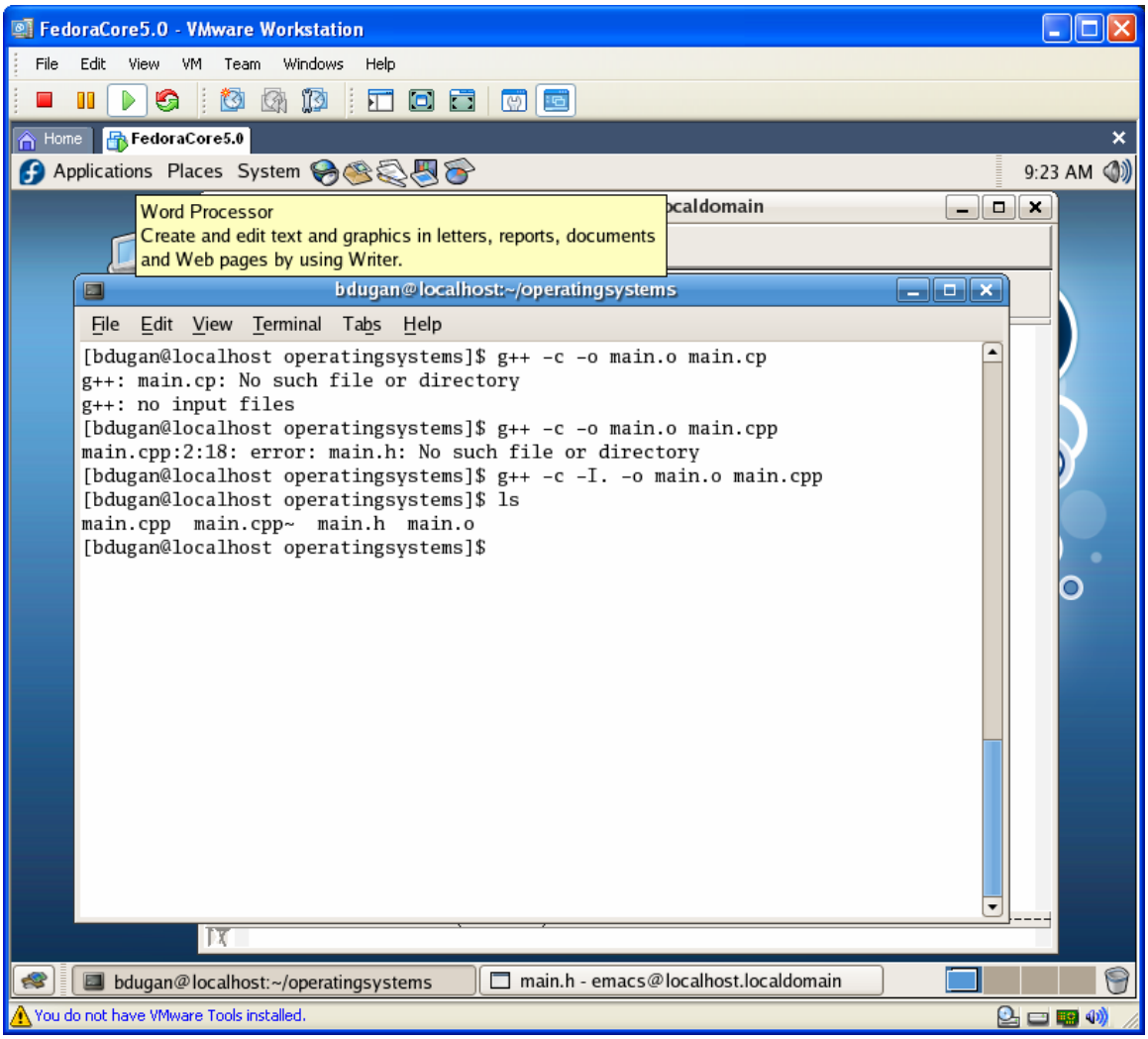
g++ -c –o main.o main.cpp



What happened? It looks like the compilation failed because the compiler couldn't find the include file "main.h". You can fix this by adding "-I." to the compiler command:

g++ -c –I. –o main.o main.cpp

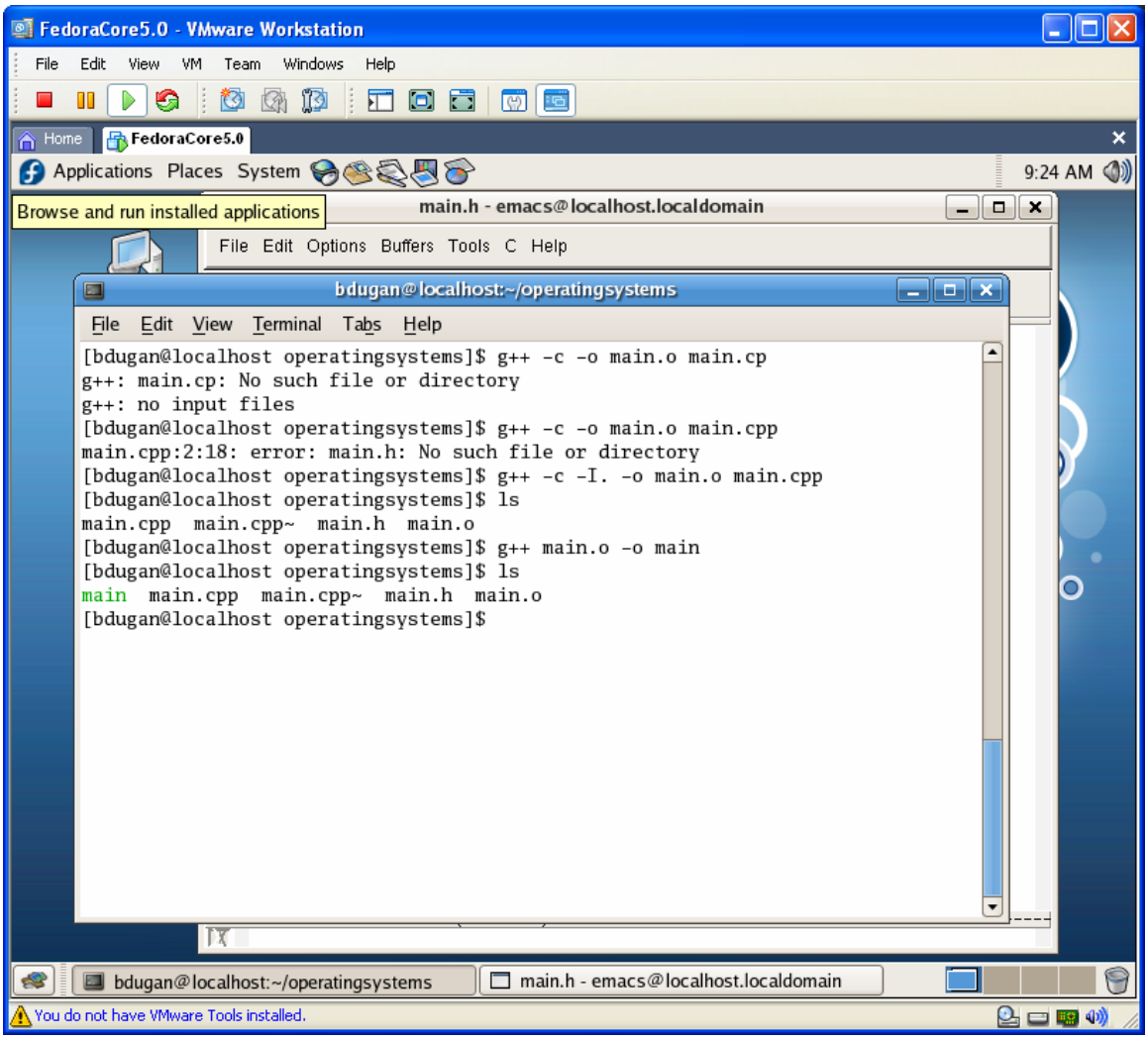
This command tells the compiler to compile main.cpp (via the "-c" argument) and produce an object file "main.o" as a result (via the "-o" argument). It also tells the compiler that in addition to any default directories, it should look in the current directory for include files (via the "-I." argument). Type "ls" to verify that you now have a "main.o" object file:



We can't run the program yet because we have an OBJECT file (main.o) not an EXECUTABLE file. To create an executable file, we need to run the linker:

g++ main.o –o main

This command tells the linker to link the object file main.o with all default libraries and produce an executable named "main" (via the –o parameter). Type "ls" again to show that main has been created:

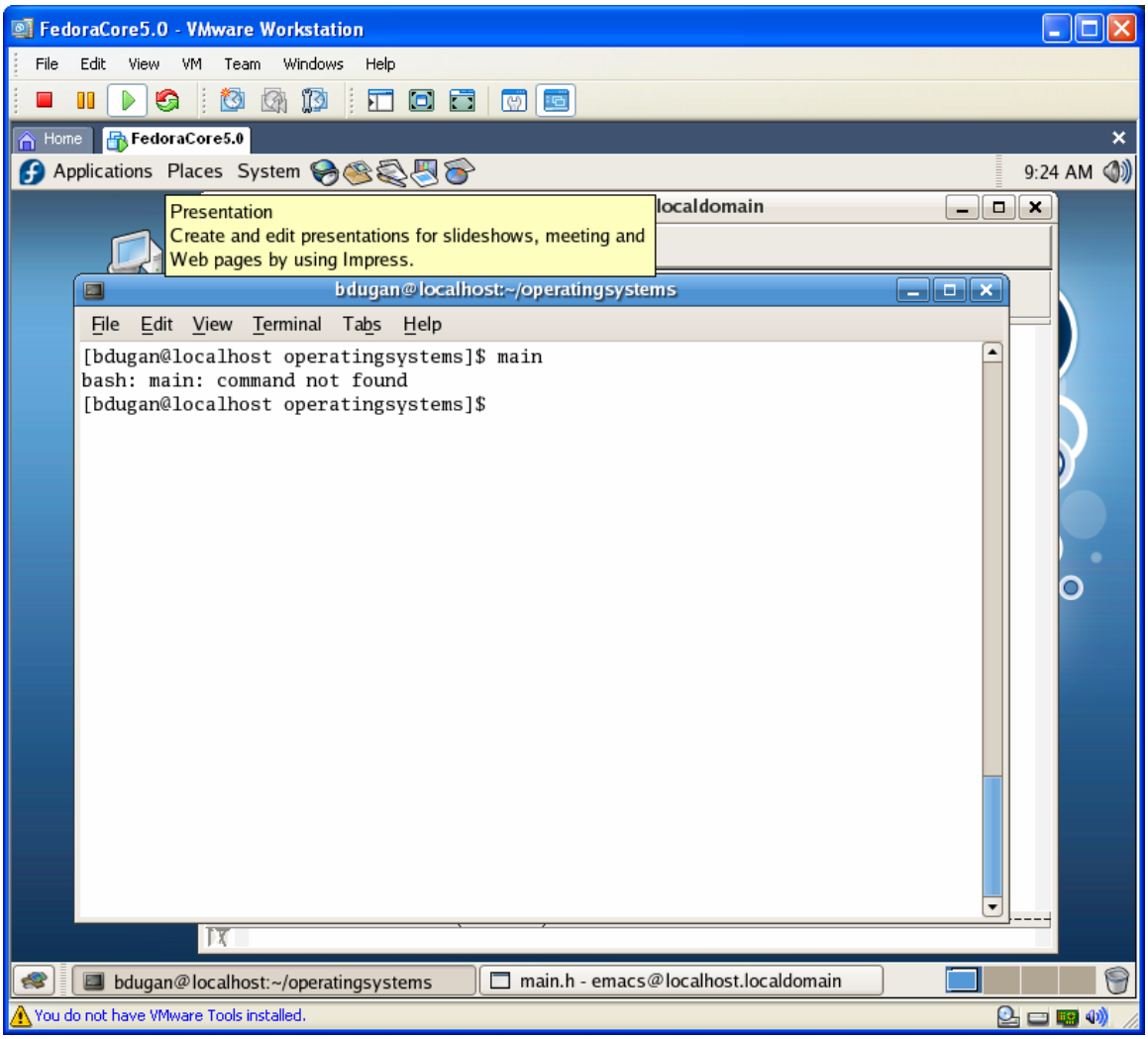


PATH and SHELL VARIABLES

Now let's run the program! At the shell command prompt type:

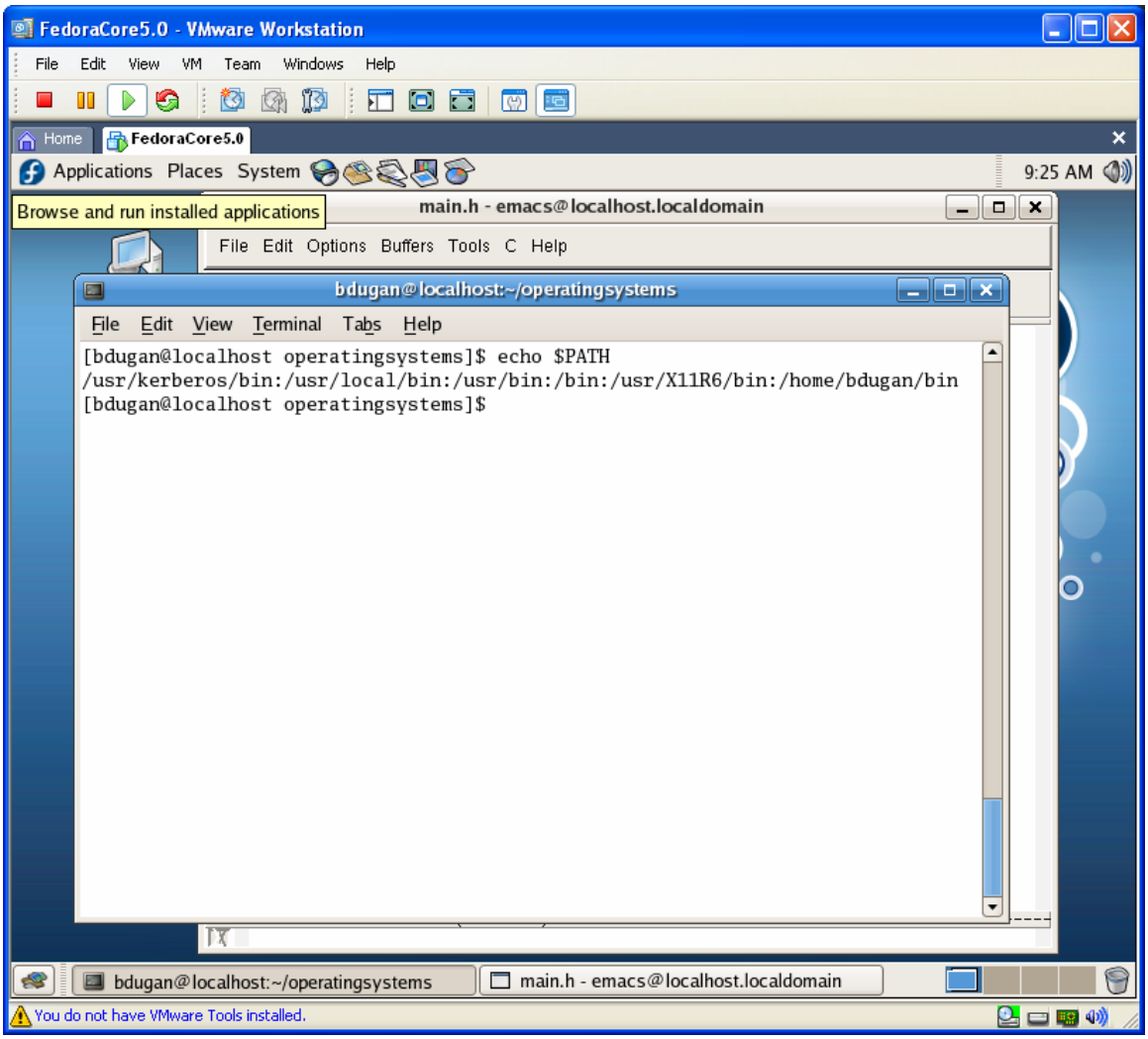
main <ENTER>

You should see something that looks like this:



What happened? The shell returned an error stating that the command was not found. This happened because the shell did not find the command in list of directories it was supposed to search for commands. Your "operatingsystems" directory is not in this list. This list is stored in something called a shell environment variable. The name of the environment variable is called "PATH". To see the list type:

```
echo $PATH
```



So the directories "/kerberos/bin" and "/usr/local/bin" and "/bin", etc. are searched by the shell for a match to the commands that you type at the shell prompt. A couple of interesting things to note here. First, the shell will match the command with the FIRST executable it finds in the list of directories provided in the PATH. Second, "operatingsystems" is not listed as one of the directories. Third, those slashes "/" seem to be going the wrong way. Microsoft operating systems also use slashes to specify the location of directories and files, but the slashes go the other way: "\". As it turns out, it's Microsoft slashes that are going the wrong way… Microsoft copied many of the basic shell command and file management mechanisms from Unix when they created DOS and changed them slightly. This is an example of one of those changes.

There are lots of environment variables that have been set by default for your shell. If you want to see the list type:

env <ENTER>

These variables have different meanings and uses. For example there's one variable "LOGNAME" which will tell you the name of the user who created the shell. You can

alter these variables or add your own. You can also write programs that check these variables.

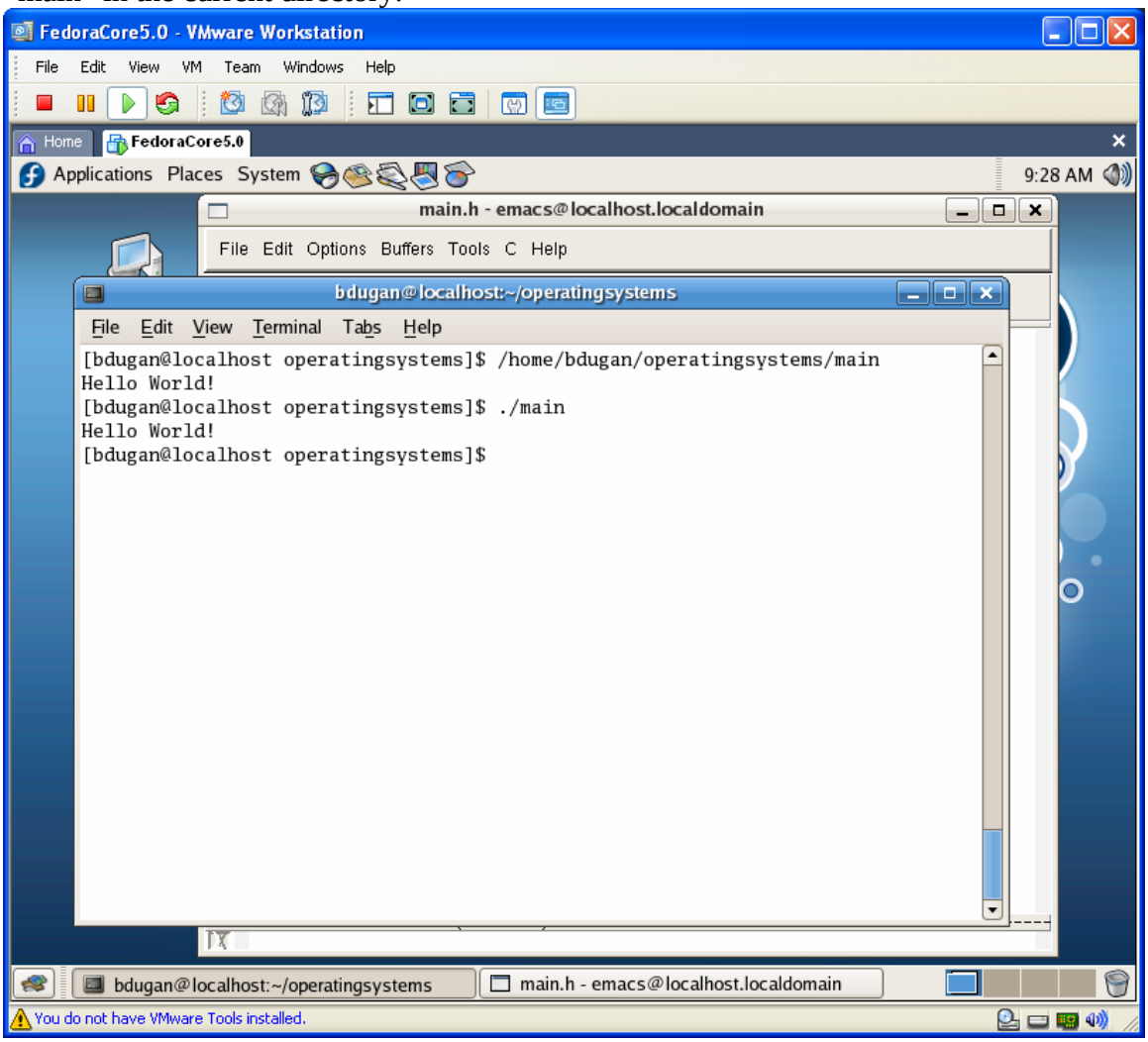
OK. Back to your program. We need to tell the shell EXACTLY where to find your main program. Type in the following command:

/home/<username>/operatingsystems/main

Where <username> is the name of the username you used to log into the workstation. It worked! Unix users hate typing a lot so you can shorten the command to just:

./main

The "./" means start at the current directory. So "./main" means execute the command "main" in the current directory.



Makefiles

Typing in the compiler and linker commands over and over each time you want to build an executable is tedious. It can also become time consuming as you build large programs that require lots of source and header files. It would be nice to have a mechanism which only compiles the files you've changed since the last time your executable was created.

Most of you have been using sophisticated development environments like Code Warrior, Turbo-C++, and Visual C++ which do this kind of thing automatically for you. You may have not even been aware of it!

For the most part, these environments owe their origins to editors like emacs and program building systems like makefiles.

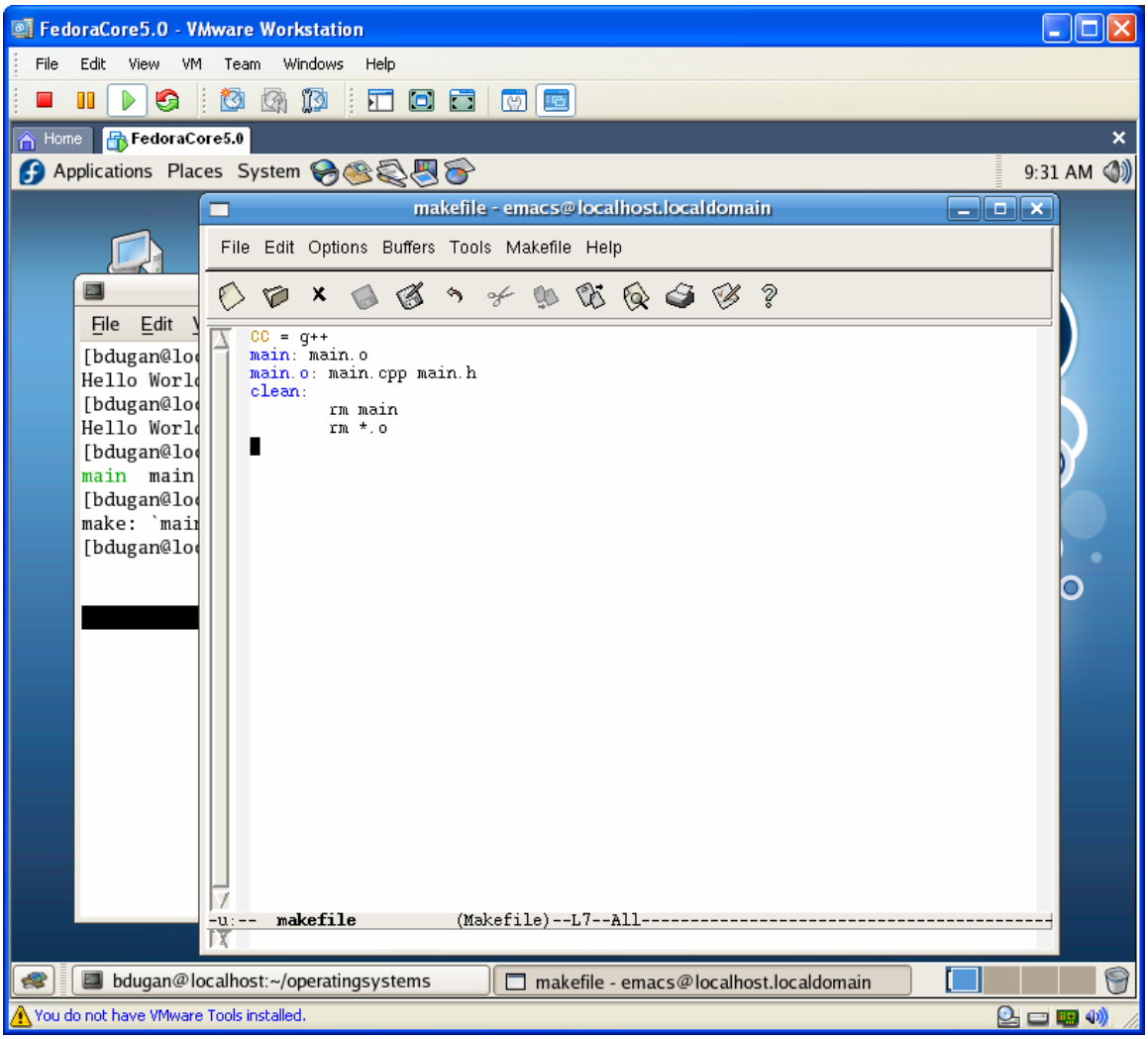
You use a makefile to identify and act upon:

- source files you want to compile
- compiler you want to use
- parameters to compiler
- object files you want to link
- linker you want to use
- parameters to linker

Makefiles have a weird and picky syntax. I still get a little mystified by them from time to time so I'm going to give you the makefile for our simple program verbatim. You should go through the makefile tutorial on the class website to get more details and a better understanding. You'll need this understanding for the rest of the projects you complete in this course.

Go back to the emacs editor and create a new file called "makefile". Put the following in the file and save the file:

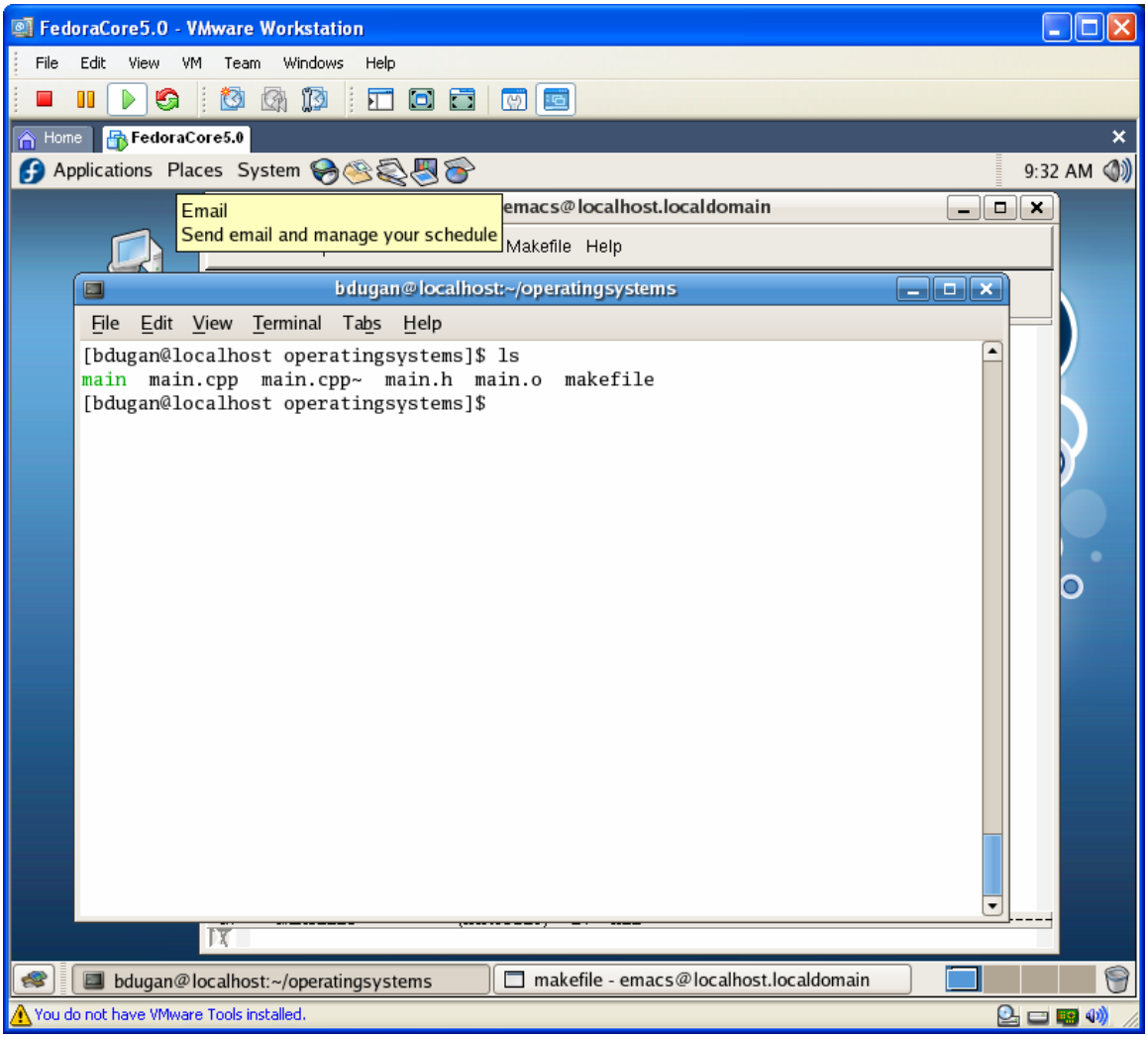
```
CC = g++
main: main.o
main.o: main.cpp main.h
clean:
	rm main
	rm *.o
```



The first line “CC = g++” tells the make program that the “g++” compiler and linker will be used to make programs with this makefile. The second line “main: main.o” states that an executable “main” will be created from the object file “main.o”. Because nothing appears beneath this line, the default link command will be used when creating this executable. The third line “main.o: main.cpp main.h” states that the object file “main.o” should be built from two files: “main.cpp” and “main.h”. Again because nothing appears beneath this line, it also states that the default compile command should be used when creating this object file.

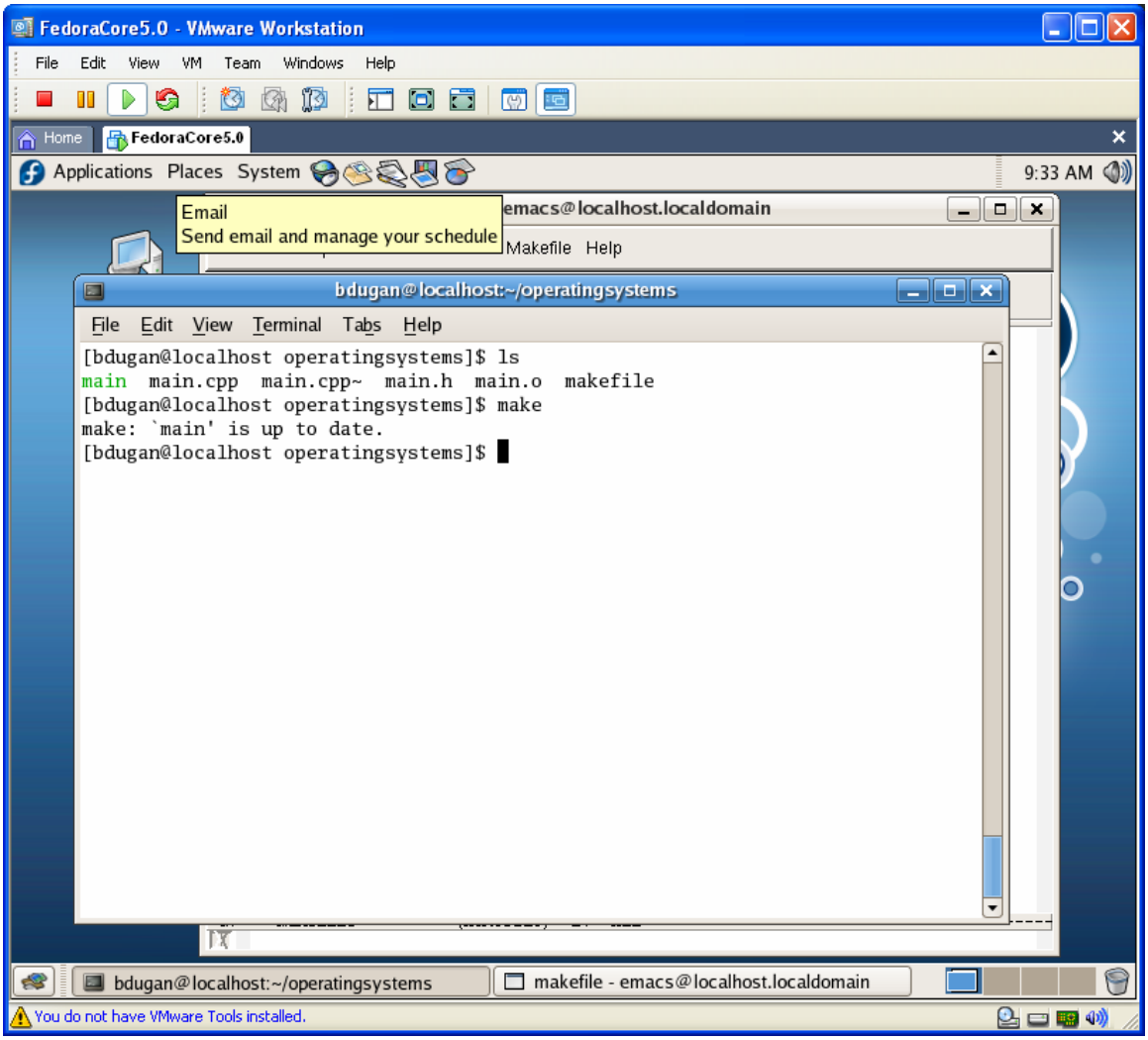
The final line “clean:” states that if the user types “make clean” two commands are supposed to be executed “rm main” and “rm *.o”. “rm” is a shell command to remove a file. So these two commands will remove the main program executable and any object files in the current directory providing you with a clean slate for building your program.

OK. Let’s try using this makefile. First, make sure you save it. Go back to the shell and use “ls” again to make sure that it exists:



Now execute the makefile by typing the command:

make <ENTER>



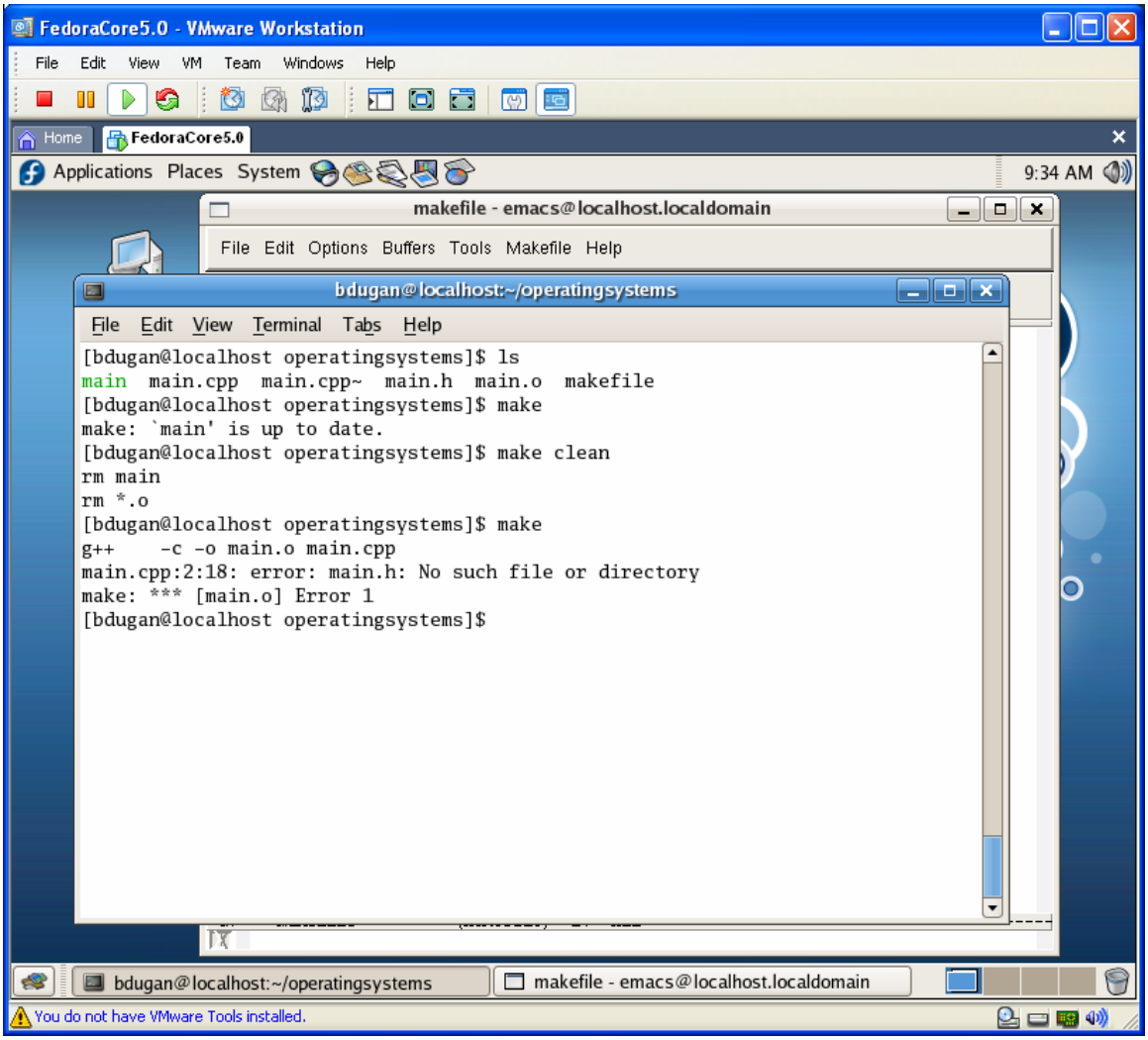
What happened? The make program determined that nothing needed to be done based on the date/times that the files used to build main were created. The timestamps for the main.cpp and main.h files were older than then main.o file. The main.o timestamp was compared against the main executable timestamp and it was found to be older. Nothing needed to be done!

Ok… we want to force the make program to do some work. Type:

make clean <ENTER>

This should erase the main executable and any object files. Now type:

make <ENTER>



What happened? This time we can see that the make program attempted to compile "main.cpp" and failed because it couldn't find main.h. We know how to solve this problem… we need to add the current directory to the include path that is searched by the compiler. This can be accomplished by setting another special variable "CPPFLAGS" in the makefile. This variable should be set to any special flags you want to add to the defaults used by the compiler. Add the following line just below the "CC = g++" line in the makefile:

```
CPPFLAGS = -I.
```

Then save the makefile and run it again from the shell:

Run your “main” program one more time to verify that it was build correctly.