Linux Programming

Over the course of COP4600 you're going to write actual, if simple, device drivers for the Linux operating system. To do so, you will need to complete several preliminary tasks.

In this document, we're going to cover the essentials of programming in Linux.

If at any point during this you get lost or have too much trouble, don't be afraid to restart it! None of these steps take more than a few minutes.

Step 1: Compile Something

This will be familiar if you've ever programmed in a Linux environment before. We're going to make sure **gcc** is working.

1. Load the text editor by doing the following:
    a. Click on the Ubuntu icon in the top-left.
    b. Click on the Applications icon at the bottom (next to the Home icon).
    c. Type "editor".
    d. Click on the **Text Editor** application that's already installed.
2. Write a tiny C program. Hello World works just fine.
3. Save it in your home directory with a **.c** extension.
4. Open a terminal window.
5. From the terminal, compile and run the program. If you named it **hello.c**, do this:

```
gcc -o hello hello.c
./hello
```

Congratulations; you've compiled and run a Linux program on your Linux VM.

Step 2: Create a Makefile

Makefiles are the UNIX-like approach to programming project files. A makefile is simply a script that contains the structure of a project, and the files that need to be created.

Make a Makefile

Create a text file called *Makefile* in your home directory, with the following lines:

```
hello: hello.c
	gcc -o hello hello.c
```

*It's important that the second line be indented with a tab character, not with spaces - otherwise, make won't work. The **g** in **gcc** may be under the **e** in **hello.c** instead of under the space as it appears here - that's fine, as long as you're using a tab character.*

Try the Makefile

From the command line, run `make`. If you haven't changed anything else since Step 3, you'll get:

```
make: `hello' is up to date.
```

Now, edit **hello.c**, change something, and run `make` again. This time, **hello** will be recreated, and make will tell you how it's doing the job:

```
gcc -o hello hello.c
```

This is normal, because of what the makefile is really doing.

Step 3: Understand your Makefile

Makefiles are entirely about *dependencies* and *targets*, and this simplest possible makefile translates to:

> *"The target **hello** is dependent on **hello.c**. If **hello** doesn't exist, or **hello.c** has been updated since **hello** was updated, you can recreate **hello** by running* `gcc -o hello hello.c`*."*

Think about this for a few minutes and you'll see that makefiles are as powerful as any other project mechanism - if not, sadly, as intuitive as some.

Step 4: Split Compilation and Linking

If we want to split the compilation and linking steps, that's simple too:

```
hello: hello.o
	gcc -o hello hello.o

hello.o: hello.c
	gcc -c hello.c
```

This sets up two targets: **hello**, which is dependent on **hello.o**, and **hello.o**, which is in turn dependent on **hello.c**. It gives **make** instructions for creating both of them. Modify **hello.c**, and **make** will solve the dependency graph and recreate both targets.