Unix make Utility

What is it?

The **make** utility is a software engineering tool for managing computer programs. It is most useful when a program has many parts. To simplify compiling and linking, **make** reduces human errors (e.g., typos, incorrect file names) and automatically knows when it is necessary to re-build one or more of the files.

When multiple people work on a program, the program usually consists of many source files, many machine object files, and an executable. When someone makes a change, it isn't always necessary to re-compile everything. Instead, you can define rules to tell **make** dependencies. For example, re-compile the `p5Insert.c` file if it is more recent than its corresponding `p5Insert.o` file.

The rules for **make** are placed in a file named **Makefile** which will contain the dependency rules and how to build files that are dependent on other files.

Example #1: Compiling and Linking without using make

Suppose you have to build a program executable named **p5.** Its source code is in the files:

```
p5Driver.c p5Insert.c p5Print.c
```

To create the executable, you had to create the machine object (.o) files and then link it. Perhaps, you used the following commands (the $ represents a Linux command prompt):

```
$ gcc –g –c p5Driver.c –o p5Driver.o
$ gcc –g –c p5Insert.c –o p5Insert.o
$ gcc –g –c p5Print.c –o p5Print.o
$ gcc –g –o p5 p5Driver.o p5Insert.o p5Print.o
```

One of the common mistakes is to specify the wrong file for the output of a gcc command, causing the file to be lost.

Example #2: Compiling and Linking using make

A Makefile (literally named Makefile) contains dependency rules and statements for how to build things. The Makefile rules below will be simplified in subsequent examples. Here is an example Makefile for example #1:

```
# Rules for building each .o
p5Driver.o: p5Driver.c p5Include.h
       gcc –g –c p5Driver.c –o p5Driver.o
```

```
p5Insert.o: p5Insert.c p5Include.h
	gcc –g –c p5Insert.c –o p5Insert.o

p5Print.o: p5Print.c p5Include.h
	gcc –g –c p5Print.c –o p5Print.o

# Rule for building the executable
p5: p5Driver.o p5Insert.o p5Print.o
	gcc –g –o p5 p5Driver.o p5Insert.o p5Print.o

# Special rule to remove the .o files
clean:
	rm –f p5Driver.o p5Insert.o p5Print.o
```

To actually build your program p5, we type the following Linux command from the directory that contains Makefile and those other files:

```
$ make p5
```

The **make** utility will look through that Makefile to see if there are dependencies for "p5". It finds the rule:

```
p5: p5Driver.o p5Insert.o p5Print.o
	gcc –g –o p5 p5Driver.o p5Insert.o p5Print.o
```

That tells the make utility that p5 is dependent on each of those listed **.o** files. That means if any of those .o files is more recent than p5, it must re-build p5. Even better than that, it checks for dependency rules for each of those .o files. Notice that each of those have a dependency (in the Makefile) on a corresponding .c file. If any of the .o files is out of date (i.e., its source .c file is more recent), those are re-built.

To re-build p5, it executes the listed gcc command. Note that there **must be a tab** in front of build command (e.g., gcc) within the Makefile. If you have told vi to automatically expand tabs to spaces, you must force it to allow you to enter a tab. One way to do that is while in insert mode, press CTRL-V and then press a TAB.

In example #2, also notice we have a rule without a dependency that states

```
# Special rule to remove the .o files
clean:
	rm –f p5Driver.o p5Insert.o p5Print.o
```

We can have make remove those .o files by simply typing

```
$ make clean
```

Example #3: simplifying make using macros

The **make** utility has some additional features to simplify your rules. You can define **make macros** which can be expanded when they are encountered. Instead of having to type the list of machine object files multiple times (once for the p5 dependency, again for the gcc build rule, and again for the clean) , we can use a macro for the list of .o files.

```
# Define the list of .o files for p5
p5OBJECTS = p5Driver.o p5Insert.o p5Print.o

# Rules for building each .o
p5Driver.o: p5Driver.c p5Include.h
	gcc –g –c p5Driver.c –o p5Driver.o

p5Insert.o: p5Insert.c p5Include.h
	gcc –g –c p5Insert.c –o p5Insert.o

p5Print.o: p5Print.c p5Include.h
	gcc –g –c p5Print.c –o p5Print.o

# Rule for building the executable
p5: ${p5OBJECTS}
	gcc –g –o p5 ${p5OBJECTS}

# Special rule to remove the .o files
clean:
	rm –f ${p5OBJECTS}
```

Example #4: simplifying make using suffix rules

The **make** utility also has special suffix rules to simplify your dependencies for repetitive things based on file suffixes.

```
# Define the list of .o files for p5
p5OBJECTS = p5Driver.o p5Insert.o p5Print.o
p5INCLUDES = p5Include.h

# Default Rules for building each .o
%.o: %.c ${p5INCLUDES}
	gcc –g –c $<

# Rule for building the executable
p5: ${p5OBJECTS}
	gcc –g –o p5 ${p5OBJECTS}

# Special rule to remove the .o files
clean:
	rm –f ${p5OBJECTS}
```

With this Makefile, we have made it very simple to build our executable. We can simply type

```
$ make p5
```

If you want to build p5Driver.o (if necessary), you can type

```
$ make p5Driver.o
```

If you want to build p5Driver.o regardless of whether it is more current than its .c file, you force **make** to re-build it by including the –B switch:

```
$ make –B p5Driver.o
```