Princeton University
COS 217: Introduction to Programming Systems
"Make" Tutorial

This tutorial describes the most fundamental aspects of the "make" tool. For much more information, see Chapter 7 of *Programming with GNU Software* (Loukides and Oram). To follow along with this tutorial, create a new directory and copy these files into it:

```
gccmultifile.pdf
make1.pdf
make2.pdf
intmath.h
intmath.c
testintmath.c
makefile1
makefile2
makefile3
makefile
```

Review

Recall the testintmath program from precept #4. It consists of the files intmath.h, intmath.c, and testintmath.c. Take a look at those files to refresh your memory.

Also recall the process that you use to build that program. Take a look at the file gccmultifile.pdf to refresh your memory. In particular, note that testintmath.c and intmath.c are preprocessed, compiled, and assembled independently to produce object files testintmath.o and intmath.o. Then testintmath.o and intmath.o are linked together (with object code from libraries) to form executable file testintmath. Finally, recall that the "-c" option tells gcc to omit the link step, that is, to only preprocess, compile, and assemble to produce .o files.

Motivation

Consider the program build process. There are two approaches that you might use.

Approach 1 is to use source files to build executable files directly. For the testintmath program, that approach is illustrated by make1.pdf. But that's not the way builds are done in the "real world."

Approach 2 is to use source files to build object files, and then use object files to build executable files. That approach is illustrated by make2.pdf. That's the approach that is used (for large programs) in the "real world."

The second approach has a substantial advantage: after changing a source file, only those files that depend on it must be rebuilt. Consider some examples…

Suppose you change intmath.c. Then you must preprocess, compile, and assemble intmath.c to reproduce intmath.o, and must link to reproduce testintmath. But you need not preprocess, compile, or assemble testintmath.c to reproduce testintmath.o. In this little program, performing such a "partial build" instead of a "complete build" is no big deal. But if the program consisted of many.c files, the amount of time saved could be substantial.

Suppose you change testintmath.c. Then you must preprocess, compile, and assemble testintmath.c to reproduce testintmath.o, and must link to reproduce testintmath. But you need not preprocess, compile, or assemble intmath.c to reproduce intmath.o. Again, in this little program performing such a "partial build" instead of a "complete build" is no big deal. But if the program consisted of many .c files, the amount of time saved could be substantial.

Finally, suppose you change intmath.h. Since intmath.h is #included into intmath.c, changing intmath.h effectively changes intmath.c. Similarly, since intmath.h is #included into testintmath.c, changing intmath.h effectively changes testintmath.c. So you must preprocess, compile, and assemble both intmath.c and testintmath.c to reproduce intmath.o and testintmath.o. And you also must link to reproduce testinmath. In other words, it would be necessary to do a complete build.

So, doing partial builds instead of complete builds can save a substantial amount of time. But doing partial builds *manually* is tedious and error-prone. It would be handy if there were a tool that could do partial builds *automatically*. How would such a tool work?

The input to the tool would be a dependency graph of the kind shown in make2.pdf. That graph shows the dependencies among files (e.g. file intmath.o depends upon files intmath.c and intmath.h), and the command necessary to build each file from the files upon which it depends. File date/time stamps also would be input to the tool. The tool's algorithm would be something like this: if file B depends on A, and the date/time stamp of A is newer than the date/time stamp of B, then rebuild B using the specified command.

Executing "Make"

Such a tool exists. Its name is "make." The "make" tool automates the program build process, automatically performing partial builds when possible.

You provide file dependencies and build commands. "Make" analyzes the specified file dependencies and file date/time stamps to determine which files must be built. It then builds them using the specified build commands. "Make" can perform other maintenance tasks as well.

This is the syntax for executing the "make" command:

```
make [-f makefile] [target]
```

where

- *makefile* is a text file containing specialized commands. Those specialized commands specify file dependencies and build commands. If you don't explicitly specify a makefile, then "make" uses the file named "makefile" if one exists. If not, then is uses the file named "Makefile". If that doesn't exist, then "make" generates an error.

- *target* identifies what "make" should build. Most frequently the target is an executable binary file. Sometimes the target is an object file. The default target is the target of the first dependency rule defined in the *makefile*. The next section describes dependency rules.

Dependency Rules

A makefile contains dependency rules. Each dependency rule has this format:

```
target: dependencies
    command
```

The *command* must be on separate line. **The *command* line must begin with a tab character.** The dependency rule tells "make" that it should (1) build *target* if and only it is older than any of its *dependencies*, and (2) use *command* to do the build.

Given *target*, "make" traverses the dependency rules, rebuilding *target* and all other targets upon which it depends, if any of those targets is out of date.

Consider makefile1. It contains three dependency rules. This is the first one:

```
testintmath: testintmath.o intmath.o
      gcc -Wall -ansi -pedantic testintmath.o intmath.o -o testintmath
```

That dependency rule tells "make" that (1) "testintmath depends upon testintmath.o and intmath.o", and (2) "if you ever need to build testintmath from testintmath.o and intmath.o, then gcc…" is the command that you should use." Thus that dependency rule captures the bottom part of the graph shown in make2.pdf.

This is the second dependency rule:

```
testintmath.o: testintmath.c intmath.h
      gcc -Wall -ansi -pedantic -c testintmath.c
```

It tells "make" that (1) "testintmath.o depends upon testintmath.c and intmath.h", and (2) "if you ever need to build testintmath.o from testintmath.c and intmath.h, then gcc…" is the command that you should use." Thus that dependency rule captures the upper left part of the graph shown in make2.pdf.

The third dependency rule is similar.

Now, to get a sense of how "make" works, execute this sequence of commands:

```
make -f makefile1 testintmath
```

"Make" builds three files: testintmath.o, intmath.o, and testintmath.

```
ls
```

The output confirms that "make" built those three files. Execute testintmath if you want.

```
touch intmath.c
```

That command changes the date/time stamp of intmath.c to now – just as if you had edited it.

```
make -f makefile1 testintmath
```

"make" rebuilds only intmath.o and testintmath. It does not rebuild testintmath.o. Thus that command demonstrates that "make" does partial builds.

```
make -f makefile1 testintmath
```

"Make" reports that all files are up to date.

```
make -f makefile1
```

"Make" reports that testintmath is up to date. The point is that the default target is the target of the first dependency rule – testintmath in this case.

Non-File Targets

makefile2 illustrates "non-file" targets. Consider the third dependency rule in that file:

```
clean:
      rm -f testintmath *.o
```

There is no file named "clean." There never will be a file named "clean." Nevertheless, you can issue the command:

```
make –f makefile2 clean
```

In that case, "make" notices that a file named "clean" does not exist, and so must be built. "Make" executes the command "rm –f testintmath *.o" in an attempt to build that file.

The effect is that "make" deletes the testintmath executable file and all object (.o) files. Thus it "cleans" your directory.

Consider the second dependency rule in that makefile2:

```
clobber: clean
      rm -f *~ \#*\# core
```

There is no file named "clobber." There never will be a file named "clobber." Nevertheless, you can issue the command:

```
make –f makefile2 clobber
```

In that case, "make" notices that a file named "clobber" does not exist, and so must be built. "Clobber" depends upon "clean," so "make" attempts to build the "clean" target first. As noted above, the effect is that "make" deletes the executable file and all object files. Then it returns to the clobber rule, and executes the command "rm –f *~ \#*\# core". That command deletes any files that end with '~' (xemacs backup files), any files that start with '#' and end with '#' (another kind of xemacs backup file), and any file named "core." Thus it really clobbers your working directory.

Finally, consider the first dependency rule in makefile2:

```
all: testintmath
```

There is no file named "all." There never will be a file named "all." Nevertheless, you can issue the command:

```
make –f makefile2 all
```

In that case, "make" notices that a file named "all" does not exist, and so must be built. "all" depends upon "testintmath," so "make" attempts to build the "testintmath" target first. The effect is that "make" rebuilds the executable file and object files, as required.

Try issuing this sequence of commands to confirm those observations:

```
make -f makefile2 clean
make -f makefile2 clobber
make -f makefile2 all
make -f makefile2
```

Non-file targets are used heavily in makefiles. You can use non-file targets for all kinds of program maintenance tasks: submit, test, print, release, install, etc.

Macros

"Make" provides a macro definition and expansion facility. It is similar to the C preprocessor's #define facility. To define a macro in a makefile, type this:

```
macroname = macrodefinition
```

Then to call a macro, type this:

```
$(macroname)
```

makefile3 illustrates macros. In that file this macro definition:

```
CC = gcc
```

commands "make" to expand each subsequent occurrence of "CC" to "gcc". This line:

```
# CC = gccmeminfo
```

is a comment, and so "make" ignores it.

Notice the call of the CC macro in the commands within the dependency rules that follow. By commenting-in or commenting-out the "CC = gcc" line, and commenting-out or commenting-in the "CC = gccmeminfo" line, you easily can tell "make" whether to use gcc or gccmeminfo when building.

Similarly, this sequence of lines:

```
CCFLAGS = -Wall -ansi -pedantic
# CCFLAGS = -Wall -ansi -pedantic -g
# CCFLAGS = -Wall -ansi -pedantic -DNDEBUG
# CCFLAGS = -Wall -ansi -pedantic -DNDEBUG -O3
```

allows you easily to specify/change the set of flags that "make" provides to the gcc (or gccmeminfo) command.

When given makefile3, "make" performs exactly the same as when given makefile2. Give it a try!

Conclusion

Finally, look at the file named makefile. It is identical to makefile3. The point is that "makefile" is the default makefile name; if you don't use the –f option when executing the "make" command, by default "make" will use "makefile". To illustrate that, issue these commands:

```
make clobber
make
```