Lecture #10: C Compiler and Make

Reading: *none*

Handouts: The Program Organizer *make*

---

1. Greetings and felicitations
2. How the compiler works
   - symbol tables
   - stages of compilation
   - linking and loading
3. Makefiles
   - what they are
   - dependencies
   - entries
   - the Makefile

The Program Organizer *make*

Introduction

When writing large programs, compiling can be quite time consuming. Suppose you have to make a change to one line in one function in your code; despite the fact that you made a small change you must wait to recompile the entire program. Would it not be nice to be able to break the program up into several pieces and recompile just those that you have changed? It would be even more convenient if there were an automated way to have the pieces that you have changed recompiled and the pieces that had not changed left alone.

Make is a facility for automated maintenance of programs. Make uses a file called a makefile which specifies the dependencies between component files, and the commands that will bring all component files up to date. Suppose you have a program that is in 3 different files and you modify one of these files. If one of the other unmodified files depends on something in the modified file, then the unmodified file should be recompiled as well as the one that was modified. When there are many source files, and the dependencies between them are complex, it would be very helpful if there were a way to specify these dependencies automatically. This is the role of the make facility. Make also serves to document how the components fit together.

The *makefile* provides the following information:

- the names of the files that comprise the program system;
- the interdependencies of these files; and
- how to regenerate the program system.

Note that if file B depends on file A and file A is changed, then file B must be recompiled even though it was not changed. This is the type of dependency that is specified in the makefile. The makefile must be in the current working directory; make assumes that the makefile is named makefile or Makefile unless told otherwise. To run *make*, type:

```
make argument
```

where *argument* is one of the targets given to the left of a ':' in a dependency rule (see below). If no argument is specified make will start with the first dependency rule in the makefile. To tell make that the makefile has a name other than the default , use the –f flag to pass the name of the makefile. If the makefile were called *dactmake*, the command:

```
make -f dactmake dact
```

will run make with the desired makefile, starting with the dependency rule that begins with *dact*.

Description File

A typical entry for a "description file" is of the form

```
target : components list
TAB    command1
TAB    command2
```

where TAB means the tab character, not the word TAB; it must come first on a command line. The target is the name of the file to be created; the components list is a list of files that are used to create the target file. This line is called a dependence rule. If make determines that the target needs to be updated, the commands following the dependence rule' will be executed. Make terminates if any of the commands is unsuccessful. An example is:

```
dact: main.o init.o process.o process2.o
      gcc -ansi -o dact main.o init.o process.o process2.o
```

The first line tells make that dact needs to be remade if any of the files to the right of the ':' are changed. Before checking the time the files were last changed, make will look for any dependence rule lines that start with each of components (main.o, init.o, process.o, and process2.o). If it finds any such lines, it will check if those components are up to date. Make determines which files have been changed by examining the time of last modification for each of the files. After checking that all the component files are up to date, and remaking any that were not, make brings dact up to date. If any of the files to the right of the ':' (main.o, init.o, process.o, or process2.o), were modified after the file to the left of the ':' (dact), then that file needs to be remade.

The second line tells make how it should remake the target dact, instructing it to link together the four object files.

If the component list has 0 elements then the commands given on the following lines will be executed only if you type

```
make target
```

Make will execute only those commands and then exit; for example, see the target *clean* in the example section below.

Example

A complete example description file follows. The line numbers are not part of the file; we'll use them to describe the contents of the makefile.

```
# makefile for dact
# the sharp sign means the rest or the line is a comment
dact:  main.o init.o process.o process2.o
       gcc -ansi main.o init.o process.o process2.o -o dact

# the next line says main.o depends on main.c
# the line after it says to create main.o with the command
# gcc -ansi -c main.c
main.o:              main.c
       gcc -ansi -c main.c

# process.o depends on both process.c and process.h
# and is created with the command gcc -ansi -c process.c
process.o:           process.c process.h
       gcc -ansi -c process.c

process2.o:          process2.c
       gcc -ansi -c process2.c
# clean is a target with no components; when you make it, the
# files main.o, init.o, process.o, and process2.o are removed
clean:
       rm main.o init.o process.o process2.o
```

Remember that *make* will only continue if no errors are reported by the commands it executes. If nothing needs to be updated or remade, *make* will report that the target is up to date:

```
make dact
dact is up to date
```

The above example does not fully exploit the capabilities of the *make* facility. Three of the four types of statements in a description file have been discussed above. The fourth and final type of statement is a macro definition. This statement has the form:

```
NAME = value
```

The value of a macro is accessed by ${NAME}. Note that "macro definitions" are strikingly similar to shell script variables; macross are simply parameters used in the makefile. Lines 3-4 in the example above could be replaced with:

```
OBJS = mail.o init.o process.o process2.o

dact : $(OBJS)
gcc -ansi $(OBJS) -o main.o
```

and the last line of the makefile could be replaced with

```
rm $(OBJS)
```

Touch

The command *touch*'(1) updates the modification date of a file to the current time and can be used to force the remaking of a target. Just type

```
touch filename
```

and the modification date of the file *filename* will be changed. In the above example, even if no files had actually been modified, the command

```
touch process.h
```

would cause process.o to be remade, which would in turn require dact to be remade.

Acknowledgement

This document was originally written by Kendrick Mock, and modified by Matt Bishop.