Module 15: Make Tool In UNIX

In Unix environment, Make is both a productivity enhancement utility as well as a program management tool. It is particularly useful in managing the development of large-sized programs. A program may be considered to be large sized when it uses a very large number of functions or it may involve a large number of developers. It is more often the case that different teams of programmers are engaged in developing different functions or sub-parts of a large programming project. In most large-sized projects, one often requires that some common definitions hold across all the functions under development. This is more often the case and is true regardless of the number of persons involved in the development (of large programming project). Clearly, all the teams must use and interpret all common definitions consistently.

In this chapter we shall discuss the facilities which make tool makes available. Make tool not only facilitates consistent usage of definitions in the large program development context, but also helps to avoid wasteful re-compilations. Thus it enhances the productivity of individuals as well as that of teams. As we shall, see it is also useful in the context of software installations.

15.1 When to Use Make

Let us examine a commonly occurring scenario in a c program development effort involving a large team of programmers. They all may be sharing a common (.h) file which may have some commonly used data definitions. In case each member of such a team has his own copy of (.h) file, then it would be very difficult to ensure consistency. Not every member may compile his program with a consistent and current data definitions. The problem becomes even more acute when, within a set of files, there are frequent updates.

Let us illustrate how problems may arise in yet another scenario. For instance, suppose some project implementation involves generating a large user-defined library for later use and subsequent integration. Such a library may evolve over time. Some groups may offer updated or new library modules and definitions for use by the rest of the project group pretty regularly. Consistent use of a new and evolving library is imperative in such a project. If different team members use inconsistent definitions or interpretations, then this can potentially have disastrous consequences.

Note that both these scenarios correspond to large-sized applications development environment. In such cases it is imperative that we maintain consistency of all the definitions which may be spread over several files. Also, such development scenarios often require very frequent re-compilations.

*Mak*e or *make* helps in the management of programs that spread over several files. The need arises in the context of programming projects using any high level programming language or even a book writing project in TEX. Mainly changes in the definitions like (.h) files or modification of (.c) files and libraries of user-defined functions require to be linked to generate a new executable. Large programs and projects require very frequent re-compilations. Make can be also be used in a Unix context where tasks can be expressed as Unix shell commands to account for certain forms of dependencies amongst files. In these situation the use of make is advised.

Make is also useful in installation of new software. In almost all new installations, one needs to relate with the present software configuration and from it, derive the interpretations. This helps to determine what definitions the new software should assume. Towards the end of this chapter we will briefly describe how installations software use a mastermakefile. They essentially use several minimakefiles which helps in generating the appropriate installation configuration. Both Windows and Unix offer their makefile versions. Most of the discussion would apply to both these environments 1.

15.2 How Make Works

Make avoids unnecessary compiling. It checks recency of an object code relative to its source. Suppose a certain source file has been modified but is not re-compiled. Make would find that the compiled file is now older than the source. So, the dependency of the object on the source will ensure re-compilation of source to generate new object file.

To ensure a consistency, make uses a rule-based inferencing system to determine the actions needed. It checks on object code dependencies on sources by comparing its recency with regards to time of their generation or modification. Actions in command lines ensure consistency using rules of dependence. Any time a (.c) or a (.h) file is modified, all the dependent objects are recreated.

Now we can state how make works: make ensures that a file, which is dependent on its input files, is consistent with the latest definitions prevailing in the input files. This also ensures that in the ensuing compilation, linking can be fully automated by Make. Thus, it

helps in avoiding typing out long error prone command sequences. Even the clean-up process following re-compilation can be automated as we will see later.

Makefile Structure: The basic structure of make file is a sequence of targets, dependencies, and commands as shown below:

```
---------------------------------------------------------------
| <TARGET> : SET of DEPENDENCIES /* Inline comments */|
| <TAB> command /* Note that command follows a tab */ |
| <TAB> command /* There may be many command lines */ |
| . |
| . |
| <TAB> command /* There may be many command lines */ |
---------------------------------------------------------------
```

The box above has one instance of a target with its dependencies paired with a possible set of commands in sequence. *A makefile is a file of such paired sequences.*

The box above defines one rule in a makefile. A makefile may have several such rules. Each rule identifies a target and its dependencies. Note that every single rule defines a direct dependency. However, it is possible that there are nested dependencies, i.e. “a" depends on “b" and “b" depends on “c". Dependencies are transitive and the dependency of “a" on “c" is indirect (or we may say implied). Should any of the dependencies undergo a modification, the reconstruction of the target is imperative. This is achieved automatically upon execution of makefiles.

Nesting of dependencies happens when a certain dependency is a sub-target. The makefile is then a description of a tree of sub-targets where the leaves are the elementary files like (.c) or (.h) files or libraries which may be getting updated ever so often. We shall use a one-level tree description for our first example. In subsequent examples, we shall deal with multi-level target trees.

As a simple case, consider the following “helloworld" program. We will demonstrate the use of a make file through this simple example.

Step1: Create a program file helloWorld.c as shown below:

```
#include<stdio.h>
#include<ctype.h>
main
```

```
{
printf("HelloWorld \n");
   }
```

Step2: Prepare a file called “Makefile" as follows:

```
# This may be a comment
hello: helloWorld.c
cc -o hello helloWorld.c
# this is all in this make file
```

Step3: Now give a command as follows:

```
make
```

Step4: Execute helloWorld to see the result.

To see the effect of make, first repeat the command make and note how make responds to indicate that files are up to date needing no re-compilation. Now modify the program. Let us change the statement in printf to:

```
printf("helloWorld here I come ! \n")
```

Now execute make again.

One can choose a name different from Makefile. However, in that case use:

```
make -f given_file_name.
```

To force make to re-compile a certain file one can simply update its time by a Unix touch command as given below:

```
touch <filename> /* this updates its recent modification time */
```

Make command has the following other useful options:

- -n option: With this option, make goes through all the commands in makefile without executing any of them. Such an option is useful to just check out the makefile itself.
- -p option: With this option, make prints all the macros and targets as it progresses.
- -s option: With this option, make is silent. Essentially, it is the opposite of -p option.
- -i option : With this option make ignores any errors or exceptions which are thrown up. This is indeed very helpful in a development environment. During development, sometimes one needs to focus on a certain part of a program. One may have to do this with several modules in place. In these situations, one often

wishes to ignore some obvious exceptions. This ensures that one is not bogged down in reaching the point of test which is in focus at the moment.

Another Example: This time around we shall consider a two-level target definition. Suppose our executable \fin" depends upon two object files (a.o) and (b.o) as shown in Figure 15.1. We may further have (a.o) depend upon (a.cc) and (x.h) and (y.h). Similarly, (b.o) may depend upon (b.cc, z.h) and (y.h). In that case our dependencies would be as shown in Figure 15.1. These dependencies dictate the following:

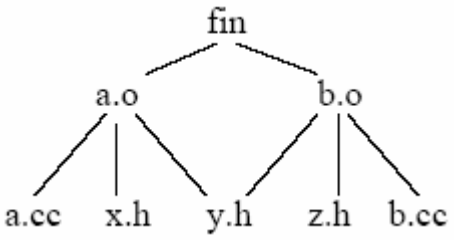

Figure 15.1: Dependencies amongst files.

- A change in x.h will result in re-compiling to get a new a.o and fin.
- A change in y.h will result in re-compiling to get a new a.o, b.o, and fin.
- A change in z.h will result in re-compiling to get a new b.o and fin.
- A change in a.cc will result in re-compiling to get a new a.o and fin.
- A change in b.cc will result in re-compiling to get a new b.o and fin.

Assuming we are using a gnu g++ compiler, we shall then have a make file with the following rules:

```
fin : a.o b.o                                   /* the top level of target */
g++ -o fin a.o b.o                          /* the action required */
a.o : a.c x.h y.h                               /* the next level of targets */
g++ -g -Wall -c a.cc -o a.o     /* hard coded command line */
b.o : b.c z.h y.h                               /* the next level of targets */
g++ -g -Wall -c b.cc -o b.o
x.h :                                           /* empty dependencies */
y.h :
z.h :                                       /* these rules are not needed */
```

The bottom three lines in the example have empty dependencies. These are also referred to as pseudo targets. This make file clearly brings out the two levels of dependencies.

Also, all the command lines in this example use hard-coded commands. Hard-coded commands are specific commands which are relevant only in that programming environment. For instance, all commands here are relevant in the g++ context only. To make *make* files more portable, we shall have to use generic symbols. Each such symbol then can be assigned specific values by using macros as we shall see a little later.

Linking with Libraries: In Unix and c program development environments it is a common practice to let users develop there own libraries. This helps in creating a customized computing environment. In fact, come to think of it, the X series of graphics package is nothing but a set of libraries developed by a team of programmers. These are now available to support our windows and graphics packages.

In this example we shall think of our final executable to be fin which depends upon a.o which in turn depends upon x.h, y.h and as in the last example. The library we shall linkup with is defined as thelib and is generated by a program thelib.cc using definitions from thelib.h. We shall make use of comments to elaborate upon and explain the purpose of various make lines.

```
#
# fin depends on a.o and a library libthelib.a
#
fin : a.o libthelib.a
g++ -o fin a.o -L -lthelib
#
# a.o depends on three files a.c, x.h and y.h
# The Wall option is a very thorough checking option
# available under g++
a.o : a.cc x.h y.h
g++ -g -Wall -c a.cc
#
# Now the empty action lines.
#
x.h :
y.h :
thelib.h :
```

```
#
# Now the rules to rebuild the library libthelib.a
#
libthelib.a thelib.o
ar rv libthelib.a thelib.o
thelib.o: thelib.cc thelib.h
g++ -g -Wall thelib.cc
# end of make file
```

If we run this make file we should expect to see the following sequence of actions

1. g++ -c a.cc
2. cc -c thelib.cc
3. ar rv libthelib.a thelib.o
4. a - thelib.o
5. g++ -o fin a.o -L -lthelib

15.3 Macros, Abstractions, and Shortcuts

We have now seen three make files. Let us examine some aspects of these files.

1. If we wish to use a c compiler we need to use the cc cmd. However, if we shift to MS environment we shall have to use cl in place of cc. If we use the Borland compiler then we need to use bcc. We, therefore, need to modify make files.

   In addition, there are many repeated forms. For instance, the compiler used is always gnu (g++) compiler.

   Both of the factors above can be handled by using variables. We could define a variable, say CC, which could be assigned the appropriate value like cl, cc, g++, or bcc depending on which environment is being used.

   Variables in make are defined and interpreted exactly as in shell scripts. For instance we could define a variable as follows:

   ```
   CC = g++ /* this is a variable definition */
   # the definition extends up to new line character or
   # up the beginning of the inline comment
   ```

   In fact almost all environments support CC macro which expands to appropriate compiler command, i.e., it expands to cc in Unix, cl in MS and bcc for Borland. A typical usage in a command is shown below:

$(CC) -o p p.c /* here p refers to an executable */

Note that an in-built or a user-defined macro is used as $(defined Macro). Note that the characters $ and () are required. Such a definition of a macro helps in porting make files across the platforms. We may define CC also as a user defined macro.

2. We should notice some typical patterns used to generate targets. These may require some intermediate target which itself is further dependent on some sources. Consequently, many file-name stems are often repeated in the make file. In addition, we have also seen repeated use of the compilation options as in -g -Wall. One way to avoid having to make mistakes during the preparation of make files is to use user defined macros to capture both file name stems and flag patterns.

Let us look at flags first. Suppose we have a set of flags for c compilation. We may define a new macro as shown below :

CFLAGS = -o -L -lthelib

These are now captured as follows:

$(CC) $(CFLAGS) p.c

Another typical usage is when we have targets that require many objects as in the example below:

t : p1.o p2.o p3.o /* here t is the target and pi the program stems */

We can now define macros as follows:

TARGET = t

OBJS = p1.o p2.o p3.o

$(TARGET): $(OBJS)

Now let us address the issue with file name or program name stems. Often we have a situation in which a target like p.o is obtained by compiling p.c, i.e., we have the stem, (i.e. the part of string without extension) repeated for source. All systems allow the use of a macro ($*) to get the stem. The target itself is denoted by macro ($@).

So if we have a situation as follows:

target : several objects

cc cflags target target_stem.c

This can be encoded as follows :

$(TARGET): $(OBJS)

```
    $(CC) $(CFLAGS) $@ $*.c
```

15.4 Inference Rules in Make

The use of macros results in literal substitutions. However, it also gives a sense of generalization for use of a rule pattern. This is what precisely an inference rule does. In an inference rule we define the basic dependency and invoke it repeatedly to get the desired effect. For instance, we need a.c file to get an object or a.tex file to get a.dvi file. These are established patterns and can be encoded as inference rules for make files.

An inference rule begins with a (.) (period) symbol and establishes the relationship. So a .c.o means we need an a.c file for generating the a.o file and it may appear as follows:

```
.c.o:
    $(CC) $(CFLAGS) $*.c
```

In fact because the name stems of .c and o. file is to be the same we can use a built-in macro ($<) to code the above set of lines as follows:

```
.c.o:
    $(CC) $(CFLAGS) $<
```

15.5 Some Additional Options

make allows use of some convenient options. For instance, if we wish to extend the range of options for the file extensions then we define a suffix rule as follows:

```
.SUFFIXES .tex .in
```

This allows us to use the file extensions .tex and .in in our makefile. Suppose we wish to use only a certain selected set of extensions. This can be done as follows

```
.SUFFIXES # this line erases all extension options
.SUFFIXES .in .out # this adds the selections .in and .out
```

Suppose we do not wish to see the enlisted sequences of makefile outputs then we may use a rule as follows :

```
.SILENT (also make -s as an option)
```

Suppose we wish that makefile on run should not abort when an error occurs, then we can choose an option -i or IGNORE rule.

```
.IGNORE (or use make -i as an option)
```

Use make -p to elicit information on the macros used in makefile.

We may on occasions like to use a makefile but may wish to get rid of all the intermediate outputs. This can be done by using pseudo-targets as shown below.

cleanup: # absence of dependency means always true

rm *.o # remove the .o files

rm *.k # remove some other files that are not needed

As the case above shows we may have more than one action when multiple actions are required to be taken.

In case a command line in a make file is very long and spills beyond the terminal window width, then we can extend it to continue on the next line. This is done by using \ (backslash) at the end. This also is useful to make some statement more explicitly readable as shown below.

p.o : p.c \

p.h

Target name	Use effect
all:	Build every thing which may create multiple execute files
clean:	Remove all .o, .a and core files to clean up the disk space
install:	Install the compiled program in its final resting place which may be like /usr/local/bin
libs:	Build only the libraries that the program depends on

Table 15.1: Use of conventions.

Gnumake: Amongst the facilities, gnu make allows us to include files. This is useful in the context of defining a set of make variables in one file for an entire project and include it when needed. The include statement is used as follows:

include project.mk common_vars.mk other_files.mk

Some Standard Conventions: Over the years some conventions have emerged in using make files. These identify some often used target names. In Table 15.1 we list these with their interpretations.

Typically, one provides a makefile for each directory in which one codes. One can create a top level directory and have the make commands executed to change the directory and run the make there as shown below:

all:

(cd src; make)

(cd math; make)

(cd ui; make)

Each command launched by make gets launched within its own shell. Typically this is Bourne shell.

Imake: Imake program generates platform specific make files. One writes a set of general rules and if configured properly, it yields a suitable makefile for the individual platforms. This is achieved through a template file called “Imake.tmpl" which first determines the machine configuration and the OS (as sun.cf for sun and sgi.cf for sgi). Next it looks for the local customization in site.def and project specific configuration in “Project.tmpl". If needed, it also uses the X11 release and motif-related information. The best way to create Imake files is to use an older file as a template and modify it to suit the present needs.

15.6 Mastermakefiles

We notice that we had to specify the dependencies explicitly. Of course, we now had the luxury of the file name stems which could be encoded. However, it often helps to use some compiler options which generate the dependencies. The basic idea is to generate such dependencies and use these repeatedly. For instance, the gnu compiler g++ with –MM option gets all the dependencies.

Once we use the (-MM) type of option to generate the options, clearly we are generating minimakefiles within a mastermakefile. This form of usage is also done by programs that install make using makefiles.