Silviu Sosiade

Linux Programming: Building and Debugging

Make and Makefile

We have just learnt from the previous chapter how to build a program from multiple source files. It seems easy and straight forward. But what if instead of three modules we have to deal with hundreds of source files (quite normal for large projects)?. Building a command line for gcc with so many files is obviously not an option. What do we do then?

The first thing that may spring to mind is to create a program or script that will attempt to automate the process of building the code and make this process easy and repeatable. The good news is that you do not have to write anything because such a program already exists. This is another GNU tool called "make". The make program understands and executes different rules specified in an input file called "Makefile". So while the make tool remains generic, the makefile contains the rules governing how a particular program has to be build.

I will start by presenting an example for a makefile that is functionally equivalent to the shell commands that I used to build the hello executable in the previous chapter. The goal of this example is to give a brief overview of the makefile syntax and usage.

To begin, lets assume I have the following files in my current working directory.

```
$ ls
main.c Makefile module1.c module1.h module2.c module2.h module3.c module3.h
```

The header and source files are the same files I used in the previous example. The content of the makefile file is the following:

```
$ cat Makefile
module1.o: module1.c module1.h
   gcc -c module1.c

module2.o: module2.c module2.h
   gcc -c module2.c

module3.o: module3.c module3.h
   gcc -c module3.c

main.o: main.c
   gcc -c main.c

hello: main.o module1.o module2.o module3.o
   gcc module1.o module2.o module3.o main.o -o hello
```

Before explaining the makefile syntax let me show you how I build the hello executable first:

```
$ make hello
gcc -c main.c
gcc -c module1.c
gcc -c module2.c
gcc -c module3.c
gcc module1.o module2.o module3.o main.o -o hello
```

Just by exploring the output of this command we can start getting some understanding about what make tool does. It compiles the source code first and then link the object files into the final executable.

If I list the content of the same directory once again I can see now the object files and executable being created.

```
$ ls
hello main.c main.o Makefile module1.c module1.h module1.o module2.c module2.h
module2.o module3.c module3.h module3.o
```

If I run the executable I will get the same output as in my previous example when I built the executable from the command line:

```
$ ./hello
Hello from module 1
Hello from module 2
Hello from module 3
```

But what happens if I try to execute the make command once again?

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

It appears that make knows somehow that nothing has changed the since last build. So it does not recompile and link anything. Make detects this by checking the timestamps of the various files it uses or generates. If the binary artifacts are older that the source code then it does the compilation. Otherwise not.

To prove this, I am going to change the timestamp of module2.c

```
$ ls -l module2*
-rw-rw-r--. 1 ssosiade ssosiade  103 Jul 30 07:29 module2.c
-rw-rw-r--. 1 ssosiade ssosiade   70 Jul 29 07:56 module2.h
-rw-rw-r--. 1 ssosiade ssosiade 1512 Jul 30 07:30 module2.o
```

```
$ touch module2.c
```

```
$ ls -l module2*
-rw-rw-r--. 1 ssosiade ssosiade  103 Jul 30 07:32 module2.c
-rw-rw-r--. 1 ssosiade ssosiade   70 Jul 29 07:56 module2.h
-rw-rw-r--. 1 ssosiade ssosiade 1512 Jul 30 07:30 module2.o
```

Now module2.c appears to be newer that module2.o. If I run the “make hello” again then module2.c only will be recompiled again and the executable will be recreated.

```
$ make hello
gcc -c module2.c
gcc module1.o module2.o module3.o main.o -o hello
```

This feature reduces the total time of the build by building only parts of the system which are affected by the latest code changes since the last build. This feature really makes a significant difference especially for large systems where a build from scratch (clean builds) could take hours .

Lets go back now to the makefile and explain how make utility understands from the content of the makefile what to do and what to build.

The key element in a makefile is represented by a **rule**. A make rule starts with a **target** and has the following format:

```
target: dependencies
<TAB> command1
<TAB> command2
<TAB> commandN
```

Please note that <TAB> must be the TAB character and it cannot be replaced by a sequence of blanks in order to achieve the same level of indentation. This is one of the most common errors in makefiles. A message like:

```
“Makefile:11: *** missing separator (did you mean TAB instead of 8 spaces?). Stop.”
```

would indicate for example that a TAB is missing on line 11.

When a target is invoked make will process the dependencies first followed by the sequential execution of all the commands specified in the target body (“command1”, “command2”...”commandN”).

A target can be invoked in one of the following ways:

1) From the command line by using “make <target>”

2) As part of the processing of another target

To understand this better lets analyze some of the rules presented in the makefile from my example:

```
module3.o: module3.c module3.h
   gcc -c module3.c
```

For the above rule the “target” (module3.o) depends on module3.c and module3.h. When this target is invoked the make tool will try to determine if module3.o exists and whether its timestamp is newer that module3.c and module3.h respectively. If any of these is not true then make will invoke the command “gcc -c module3.c”

Target "module3.o" can be independently invoked from the command line as follows:

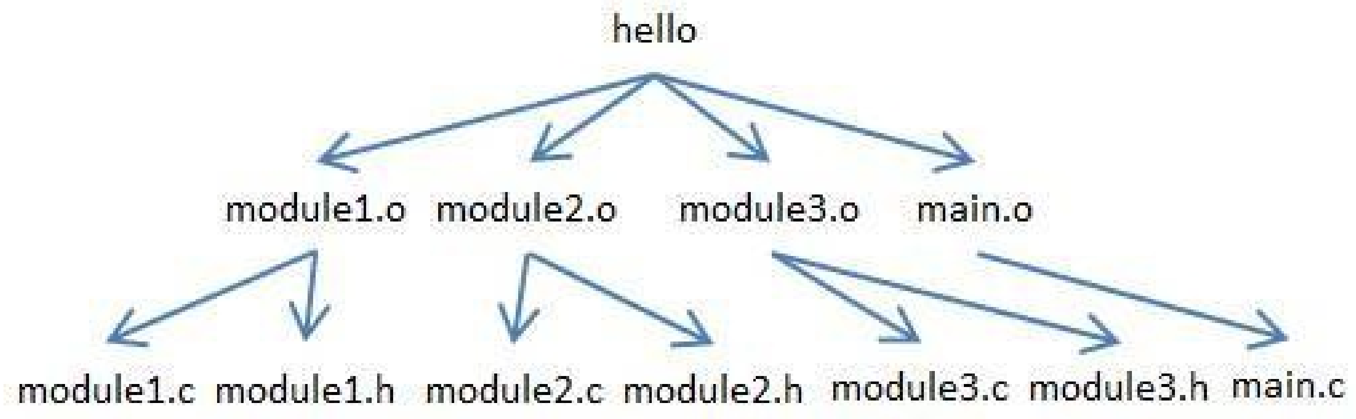
```
$ make module3.o
gcc -c module3.c
```

Lets explore this rule:

```
hello: main.o module1.o module2.o module3.o
   gcc module1.o module2.o module3.o main.o -o hello
```

In this rule, "hello" target is dependent on the following list of targets: "main.o module1.o module2.o module3.o". Before executing the gcc command make will first need to invoke module1.o target, followed by module2.o, and so on. A complex target will result in a "dependency tree" that needs to be processed.

This is for example the dependency tree for "hello" target from my example:



The command(s) associated with a target can be any Linux shell commands. There is no restriction on what shell commands can be used in makefiles. Therefore makefile targets can be written that so that files can be moved around or deleted or to do any other operations that you would normally expect to see in a regular shell script. In reality, many makefiles contains targets such as “make install” or “make deploy” which has nothing to do with compiling and linking C/C++ code.

For example, below is a “clean” target whose purpose is to delete the executable and all the objects files previously created by “make hello” command.

```
clean:
   rm -rf hello *.o
```

Thus the following command “**make clean hello**” represents a way to force make to rebuild everything from scratch:

```
$ make clean hello
rm -rf hello *.o
gcc -c main.c
gcc -c module1.c
gcc -c module2.c
gcc -c module3.c
gcc module1.o module2.o module3.o main.o -o hello
```

The make tool contains what is called **“builtin rules”** (or **“default rules”**). What do you think will happen if I remove the target “module1.o” from my makefile and try to invoke “make module1.o” from the command line? You would probably be tempted to say that this command will fail. Well, this is not quite the case:

```
$ make module1.o
```

```
cc -c -o module1.o module1.c
```

Why is this? First let’s not forget that make was written particularly for controlling and automating the build process. To make the work easy some default rules for a number of popular programming languages (C, C++, Fortran, Modula-2, and so on…) have been integrated into make tool. These rules act like “fallback rules” if no other matching rules are found in the makefile. Therefore based on the *.o suffix the make tool is able to determine that the target is referring to a C (or C++) object file so it must look for a module1.c (or module1.cpp) file in the current folder.

I am personally not convinced how useful these builtin rules are. Built in rules may cause subtle issues sometimes and very hard to detect. For instance the builtin rules for C use the standard C compiler “cc” whereas the makefile uses GNU C compiler “gcc”. Some people probably do find these rules useful but I am personally in favor of having full control on how programs are built so I prefer to either disable built in rules (make --no-builtin-rules) or override them in my makefiles.

The overriding of the built in rules is possible by using what make tool understands by “**pattern based rules**”. Here is how the previous makefile will look like by using a single rule that specifies the all the object files (*.o) should be created from a corresponding *.c file and by using “gcc” compiler.

```
$ cat Makefile
%.o: %.c
   gcc -c $^
```

```
hello: main.o module1.o module2.o module3.o
   gcc module1.o module2.o module3.o main.o -o hello
```

```
clean:
   rm -rf hello *.o
```

One could easily notice that the use of pattern based rules makes the makefiles look tidier and more compact.

Before going any further I need to provide you a quick explanation regarding the pattern based rule written above. “%.o: %.c” says that for any given <file name>.o the make tool must search for a <file name.c> file in the current folder. To some extent this is similar to using “*” wildcard in Linux but with one constrain: it must be an exact match for the file name for both object and source file.

What does “$^” from the last makefile mean? This refers to an internal variable which expands to the name of the prerequisite (<file name.c>. Similarly “$@” contains the name of the target (<file name.o>).

I just mentioned that “$^” refers to a “**variable**”. Yes, we can use variables when writing makefiles.

I will give you a quick example to understand how variables can be useful in makefiles. Let's assume that I want to change the compiler, so instead of using gcc complier I want to use a different compiler. In all the makefiles that I presented so far if I want to change GNU compiler then I need to change the makefiles in several places where “gcc” string is present. What if

I use a variable .lets name it CC, to store the name of the compiler and have this variable used in the rest of the makefile instead of hardcoding the name of the compiler? Now I will only to change the “value” of this variable and this change will be propagated everywhere in the makefile. See the following makefile for example:

```
$ cat Makefile
CC=gcc

%.o: %.c
   ${CC} -c $^

hello: main.o module1.o module2.o module3.o
   ${CC} module1.o module2.o module3.o main.o -o hello

clean:
   rm -rf hello *.o
```

Similarly, the compiler parameters represents something that may need to be changed quite often during the development of a program or even after. This represents another good example where makefile variables could be useful:

```
$ cat Makefile
CC=gcc
CCFLAGS=-Wall -g

%.o: %.c
   ${CC} ${CCFLAGS} -c $^

hello: main.o module1.o module2.o module3.o
   ${CC} module1.o module2.o module3.o main.o -o hello
```

```
clean:
   rm -rf hello *.o
```

Makefile also supports conditional statements (if-then-else). I am going to refer to another common use case in order to introduce the conditional statements for makefile. I would like to expand the makefile from my previous example so that I can invoke it as “make DEBUG=true” from the command line when I want to build the program with debugging information (-g).

Here is how this can be implemented by using makefile conditional statements:

```
$ cat Makefile
CC=gcc
```

```
ifeq ($(DEBUG),true)
CCFLAGS=-Wall -g
else
CCFLAGS=-Wall
endif
```

```
hello: main.o module1.o module2.o module3.o
   ${CC} module1.o module2.o module3.o main.o -o hello
```

```
%.o: %.c
   ${CC} ${CCFLAGS} -c $^
```

```
clean:
   rm -rf hello *.o
```

The last topic regarding makefile syntax that I would to discuss are **functions**. A function call in terms of makefile syntax resembles to some extent a variable reference.

```
$(function arguments)
```

You may or may not have a variable on the left hand side storing the return value of the function:

```
var := $(function arguments)
```

Below is an example of a function call that stores the names of all the files with extension txt in the variable ALL_TEXT_FILES. This is done by invoking the "shell" function and passing "ls *.tst" as input argument.

```
ALL_TEXT_FILES:=$(shell ls *.txt)
```

Make has a sizable number of functions that can be usefull in different situations. The list is far too long to be covered in this book. But I strongly recommend bookmark and to check the official make documentation (https://www.gnu.org/software/make/manual) to learn about various functions being available. Using functions allows achieving greater flexibility therefore functions are extensively used in almost any makefile used in building slightly more complex software products.

It's probably the time to see an example of using functions in the makefile that I have been gradually refining in this chapter. One limitation of the last makefile I presented is that every time when I will be adding a new C module I will also need to manually update the dependency list for the hello target. I would like automate this somehow and be able just to drop a C source file in that folder and have the makefile capable of figuring out which object files are required to build the executable.

This is where functions can help. See in the makefile below how to implement this:

```
SOURCE_FILES = $(wildcard *.c)
OBJECT_FILES = $(SOURCE_FILES:.c=.o)

CC=gcc

ALL_TEXT_FILES:=$(shell ls *.txt)

ifeq ($(DEBUG),true)
CCFLAGS=-Wall -g
else
CCFLAGS=-Wall
endif

hello: ${OBJECT_FILES}
   ${CC} ${OBJECT_FILES} -o $@

%.o: %.c
   ${CC} ${CCFLAGS} -c $^

clean:
   echo ${ALL_TEXT_FILES}
   rm -rf hello *.o
```

This concludes the brief presentation of the make tool and makefile syntax. The purpose of this chapter is to give you a good starting point in working with makefiles in order to build your Linux programs.

When you will need to explore the build system for a large system you will likely see an entire collection of makefiles and that build system may look a bit overwhelming and hard to understand. But you should not worry, reading complex makefiles represents a challenge even for seasoned Linux programmers. However all the complexity is built on top of the simple concepts that I presented in this book and with a little time and patience you will be able to understand what the build system does and gradually be able to maintain and update it.