CS Students Linux User's Guide

Writing a Makefile

Author: Jaco Kroon (jaco@kroon.co.za)
Version: 1.0
Last modified: Mon Aug 11 13:27:34 SAST 2003

Table of Contents

4.2 Writing a Makefile
- 4.2.1 Why Use a Makefile?
- 4.2.2 A Simple Makefile Example
- 4.2.3 How Does make Determine Which Commands to Execute?
- 4.2.4 Make Variables
- 4.2.5 Built-in make Variables
- 4.2.6 Pattern Matching in Makefiles
- 4.2.7 Auto Dependencies
- 4.2.8 Concluding on Makefiles

Back to CSSLUG

4.2 Writing a Makefile

4.2.1 Why Use a Makefile?

In any project you can have any number of source files. Now imagine having to recompile even the simplest of programs consisting of two source files, binarytree.c and mainprog.c. You will need to type the following command:

```
gcc -o outprogname binarytree.c mainprog.c
```

Now, since that mainprog.c files takes ages to compile, you don't want to recompile it everytime, the same goes for the binarytree.c file. Now imagine you had a hundred source files (think big, and then some more).

A Makefile makes this simple by allowing us to specify rules (consisting of dependencies and commands) to build our project.

4.2.2 A Simple Makefile Example

The following is a very simple, but complete Makefile:

```
outprogname : binarytree.o mainprog.o
	gcc -o outprogname binarytree.o mainprog.o

binarytree.o : binarytree.c
	gcc -c binarytree.c

mainprog.o : mainprog.c
	gcc -c mainprog.c
```

Please note that the indented lines begins with a tab character.

In the case where both files needs to be compiled, make will issue three commands - yes this can be wasteful - namely:

```
gcc -c binaryproc.c
gcc -c binarytree.c
gcc -o outprogname binarytree.o mainprog.o
```

It is clear that these are the indented lines. You could have executed all of these commands by hand, if you were so inclined (programmers are known to be lazy), but typing `make` is easier. Now going on to dissect the above, a Makefile consists of a set of rules. Each rule consists of dependencies and commands. The lines with the colons in are dependencies and the lines that are tabbed in is commands. For each dependency, the commands required to bring the target - the file(s) before the colon - up to date follows it one the next line, indented with a tab character. The file(s) to the the right of the colon is the files on which the target relies, these are almost always used as input to the commands on the lines following the dependency.

It should be clear that the two rules at the bottom of the given Makefile produces .o files, created from the .c files (the -c flag tells gcc to compile the source file, but not link it). The topmost rule takes the two object files and links them together to create the final executable.

4.2.3 How Does make Determine Which Commands to Execute?

There are two reasons why make would decide a file needs to be made:

1. The file does not exist

2. The file is outdated

The first case is simple, the file does not exist - nuff said. The second case should be reasonably clear as well. Considering the simple Makefile above, edit binarytree.c. You would now like to compile outprogname. It relies on binarytree.o and mainprog.o. binarytree.o in turn relies on binarytree.c and mainprog.o on mainprog.c. You've just edited binarytree.c, so as you probably expect, binarytree.o is outdated and as a result so is outprogname. Make then goes ahead and first builds binarytree.o and then outprogname using the commands given.

A file is considered outdated if and only if:

- It is listed on the left of a colon
- A file on the right of that same colon is newer than that file

File "a" is considered newer than file "b" if its last modified time is more recent than file "b"'s.

4.2.4 Make Variables

Suppose you would like to use another C compiler, say cc. Now you can either replace gcc right through with cc, or you could have been smart and used variables. A make variable, as in any programming language, is simply a place holder for another value. You can create such a variable in a Makefile with a line such as:

```
cc=gcc
```

Actually, variables in make work very much like in bash scripting. So to actually replace the value of `cc` (which evaluates to `gcc`) just use `${cc}`. Same people prefer using () instead of {}, but it doesn't matter which one you choose. Now let's asume that we would also like to be able to specify different levels of optimization as well as those debugging flags Derick has mentioned. We would need to specify the -O flags to the compile commands and the -g flag to all of them. Ok, so let's create two more variables: The first, called `cflags`, contains the flags we would like to pass to the compiler, and the second, called `ldflags`, contains the options we would like to pass to the linking command. The resulting Makefile will look something like:

```
cc=gcc
cflags=-O3 -g
ldflags=-g

outprogname : binarytree.o mainprog.o
    ${cc} ${ldflags} -o outprogname binarytree.o mainprog.o

binarytree.o : binarytree.c
    ${cc} ${cflags} -c binarytree.c

mainprog.o : mainprog.c
```

```
	${cc} ${cflags} -c mainprog.c
```

4.2.5 Built-in Make Variables

Now I'm really lazy, and I mean that. So I'm aiming at reducing the size of my Makefiles even more (the previous step actually enlarged it, but it made mass adjustments easier - similar to constants in large programs). Also, I would like to keep the amount of debugging I've got to do in my Makefiles to a minimum. For these, and probably many others, make has a few built-in variables. I list a few of the useful ones, along with an explanation below:

Variable	Description
$@	This will always expand to the current target.
$<	The name of the first prerequisite. This is the first item listed after the colon. (binarytree.o in the first rule above)
$?	The names of all the prerequisites that are newer than the target. In the above example this will evaluate to binarytree.o for the first rule.
$^	The names of all the prerequisites, with spaces between them. No duplicates
$+	Like $^, but with duplicates. Items are listed in the order they were specified in the rule.

4.2.6 Pattern Matching in Makefiles

As far as I know, this is a GNU extension, so if you plan to create very portable Makefiles, don't use it. In this case I suggest you use tools such as autoproject, automake, autoconf to build your Makefiles for you (Sorry M $ fans, but the configure scripts created by these tools doesn't work on Windows AFAIK).

As I mentioned above, I would like to get my Makefiles to be as short and compact as possible. Now, considering the variables mentioned above, take a look at:

```
binarytree.o : binarytree.c
	${cc} ${cflags} -c $<

mainprog.o : mainprog.c
	${cc} ${cflags} -c $<
```

Now I see a lot of repetition here. The only parts that differ here are the bases of the filesnames. Luckily, there is a way of compacting these two rules into one -- it's called pattern matching and has saved me quite a bit of time in my short lifetime. To use a pattern, you use % where you would like to actually match anything into it. For example, the above could be written as:

```
%.o : %.c
	${cc} ${cflags} -c $<
```

Now that looks a lot simpler. In general, I have the following rule in all my Makefiles:

```
%.o : %.c Makefile
    ${cc} ${cflags} -c $<
```

Which is identical to the above, except that it forces a complete recompile each time I modify the Makefile. I do this so that if I change my Makefile, changes will take effect. For example, so I drop the -O3 flag from the cflags variable. Normally this would indicate that you would like the project to be recompiled without optimation

Also, it must be mentioned that it becomes possible to have two rules for the same target. I'm still not sure what make does in this case, but I suspect that both sets of commands gets executed.

4.2.7 Auto Dependencies

This is bordering on really advanced and I'm not always 100 % sure whether this works (according to the info pages it should, but it sometimes seems that the makefile.dep file doesn't get reloaded). Use with caution (even though you would definitely have the same problem without it).

In order to keep things simple, I often have a config.h file lying around somewhere (this is also a convention in the autoproject set of tools). Now if you change it you want all the files that includes it to be recompiled. Oh bugger, now which ones are those, since that other header file also includes it. Now, what if we we're to list every source file with all it's header files in a rule, and then all the header files with all the header files they include etc...

This will lead us back to where we began and it will probably be easiest to compile by hand again.

But as Derick mentioned, gcc has a -M switch, which will output exactly what we need. There are two variants of this switch: -M, which will list all header files included, including system header files, and -MM, which will only list header files included with the `#include "..."` directive and not those with the `#include <...>`

`directive.`

`At this point in time, it is probably a good idea to mention that a rule does not need to have commands associated with it. The following is therefor a totally valid rule:`

```
file.c : somefile.h anotherfile.h
```

`As a matter of fact, this is excactly the type of rules that gcc -MM outputs. So we add The following to our Makefile`

```
makefile.dep : *.[Cch]
    for i in *.[Cc]; do gcc -MM "$${i}"; done > $@
include makefile.dep
```

Note the double $, this is because make uses $ as variable expansion, in this case we would like a single $ to be part of the command, and as in C/C++ with \, we simply put a double $ sign. For those not familiar with bash scripting, this command will execute a for loop, not unlike a for loop in programming, but using variable substitutions (as is available in PHP and Perl). The [Cch] is not unlike a ? in wildcard expansion, except that it will only match the characters C, c and h. This is called a glob expression. Strictly speaking * is also a glob expression (man 7 glob). Thus the command

```
gcc -MM "$i"
```

will be executed for every .c and .C file, substituting $i for each file. Then we capture the combined output from these commands into the file makefile.dep.

The next line tells make to include makefile.dep as part of the current Makefile. This is where the problem kicks in. Make first loads and parses the entire Makefile, thus including makefile.dep, before actually processing any rules. Now, I suppose you can see that when makefile.dep gets updated, it is already included. According to the info pages, this should cause make to "abort" and restart, causing the updated makefile.dep to be loaded. I sometimes suspect this is not the case, but it is still better than not having this in at all.

4.2.8 Concluding on Makefiles

That should give you a great head start on Makefiles. This is by no means a complete guide, even though it covers practically everything I have ever used. There are other nice tricks like

```
glfont.o : override cflags+=$(shell freetype-config --cflags)
```

Which will cause the command freetype-config --cflags to be executed and it's output made part of the cflags variable whilst compiling glfont.o. Also the += specifies that it should be appended to the existing cflags variable instead of merely discarding the existing value as would happen with =. The override keyword also has special meaning in this context.

So where can you find more information on this, and other features of GNU make?

```
info make
```

should tell you all you need to know. Enjoy Makefiles -- they truly make your life easier.

Back to CSSLUG