CE 705 Real time systems
Readings: Makefiles

Sometimes during your life with Linux you will probably have to deal with ***make***, even if you don't plan to do any programming. It's likely you'll want to patch and rebuild the kernel and that involves running ***make***.

For some of our examples, we'll draw on the current makefile for the Linux kernel. It exploits a lot of extensions in the powerful GNU version of ***make***, so we'll describe some of those as well as the standard ***make*** features. A good introduction to make is provided in ***Managing Projects with make*** by Andrew Oram and Steve Talbott. GNU extensions are well documented by the GNU ***make*** manual.

Most users see ***make*** as a way to build object files and libraries from sources and to build executables from object files. More conceptually, ***make*** is a general-purpose program that builds ***targets*** from ***prerequisites***. The target can be a program executable, a PostScript document, or whatever. The prerequisites can be C code, a TEX test tile, and so on.

While you can write simple shell scripts to execute gcc commands that build an executable program, ***make*** is special in that it knows which targets need to be rebuilt and which don't. An object file needs to be recompiled only if its corresponding source has changed.

For example, say you have a program that consists of three C source files. If you were to build the executable using the command:

papaya$ gcc –o foo.c bar.c baz.c

each time you changed any of the source files, all three would be recompiled and relinked into the executable. If you only changed one source file, this is a real waste of time. What you really want to do is recompile only the one source file that changed into an object file and relink all of the object files in the program to from the executable. ***make*** can automate this process for you.

What make Does

The basic goal of ***make*** is to let you build a file in small steps. If a lot of source files make up the final executable, you can change one and rebuild the executable without having to recompile everything. In order to give you this flexibility, ***make*** records what files you need to do your build.

Here's a trivial makefile. Call it ***makefile*** or ***Makefile*** and keep it in the same directory as the source file:

```
edimh: main.o edit.o
        gcc –o edimh main.o edit.o

main.o: main.c
        gcc –c main.c

edit.o: edit.c
        gcc –c edit.c
```

This file builds a program named ***edimh*** from two source files named ***main.c*** and ***edit.c***. You aren't restricted to C programming in a makefile, the commands could be anything.

Three entries appear in the file. Each contains a dependency line that shows how a file is built. Thus the first line says that ***edimh*** (the name before the colon) is built from the two object files ***main***.o and ***edit***.o (the name after the colon). What this line tells ***make*** is that it should execute the following gcc line whenever one of those object files change. The lines containing commands have to being with tabs (not spaces).

The command:

papaya$ make edimh

Executes the gcc line if there isn't currently any file named ***edimh***. But the gcc line also executes if ***edimh*** exists, but one of the object files is newer. Here, ***edimh*** is called a target. The files after the colon are called either ***dependent*** or ***prerequisites***.

The next two entries perform the same services for the object files. ***main.o*** is built if it doesn't exist or if the associated source file main.c is newer. ***edit.o*** is built from ***edit.c***.

How does ***make*** know if a file is new? It looks at time stamp, which the filesystem associates with every file. You can see time stamp by issuing the ***ls –l*** command. Since the time stamp is accurate to one second, it reliably tells ***make*** whether you've edited a source file since the latest compilation or have compiled an object file since the executable was last built.

Let's try the makefile and see what it does:

papaya$ make edimh

gcc –c main.c
gcc –c edit.c
gcc –o edimh main.o edit.o

If we edit main.c and reissue the command, it rebuilds only the necessary files, saving us some time:

papaya$ make edimh
gcc –c main.c
gcc –o edimh main.o edit.o

It doesn't matter what order the three entries are within the makefile. ***make*** figures out which files depend on which and executes all the commands in the right order. Putting the entry for ***edimh*** first is convenient, because that becomes the file built by default. In other words, typing ***make*** is the same as typing make ***edimh***.

Here's more extensive makefile. See if you can figure out what it does:

```
install: all
        mv edimh /usr/local
        mv readimh /usr/local

all: edimh readimh

readimh: main.o edit.o
        gcc –o edimh main.o read.o

main.o: main.c
        gcc –c main.c

edit.o: edit.c
        gcc –c edit.c

read.o: read.c
        gcc –c read.c
```

First we see the target **install**. This is never going to generate a file; it's called a ***phony target*** because it exists just so you execute the commands listed under it. But before install runs, all has to run, because install depends on all. (remember, the order of the entries in the file doesn't matter.)

So ***make*** turns to the all target. There are no commands under it (this is perfectly legal), but it depends on ***edimh*** and ***readimh***. These are real files; each is an executable program. So ***make*** keeps tracing back through the list of dependencies until it arrives at

the .c files, which don't depend on anything else. Then it painstakingly rebuilds each of the targets.

Here is a sample run (you may need root privilege to install the files in the /usr/local directory):

```
papaya$ make install
gcc –c main.c
gcc –c edit.c
gcc –o edimh main.o edit.o
gcc –c read.c
gcc –o readimh main.o read.o
mv edimh /usr/local
mv readimh /usr/local
```

So the effect of this makefile is to do a complete build and install. First it builds the files needed to create ***edimh***. Then it builds the additional object file it needs to create ***readimh***. With those two executables created, the all target is satisfied. Now ***make*** can go on to build the install target, which means moving the two executables to their final home.

Many makefiles, including the ones that build Linux, contain a variety of phony target to do routine activities. For instance, the makefile for the Linux kernel includes commands to remove temporary files:

```
clean:  archclean
        rm –f kernel/ksyms.lst
        rm –f core ‘find . – name ‘*.[oas]’ –print’
.
.
.
```

and to create a list of object files and the header files they depend on (this is a complicated but important task; if a header file changes, you want to make sure the files that refer to it are recompiled):

```
depend dep:
        tuoch tools/version.h
        for I in init/*.c;do echo –n “init/”;$(cpp) –m $$I;done > .tmpdep
.
.
.
```

Some of these shell commands get pretty complicated.

Some Syntax Rules

The hardest thing about maintaining makefiles, at least if you're new to them, is getting the syntax right. If you use space where you're supposed to use tabs or vice versa, your makefile blows up. And the error messages are really confusing.

Always put a tab at the beginning of a command-not space. And don't use a tab before any other line.

You can place a hash mark***(#)*** anywhere on a line to start a comment. Everything after any hash marked is ignored.

If you put a backslash at the end of a line , it continues on the next line. That works for long commands and other types of makefile lines too.

Now let's look at some of the powerful features of ***make***, which form a kind of programming language of their own.

Macros

When people use a filename or other string more than once in a makefile, they tend to assign it to a macro. That's simply a string that ***make*** expands to another string. For instance, you could change the beginning of our trivial makefile to read:

```
OBJECTS = main.o edit.o

edimh: $( OBJECTS)
        gcc –o edimh $( OBJECTS)
```

When ***make*** runs, it simply plugs in **main.o edit.o** wherever you specify ***$(OBJECTS).*** If you have to add another object file to project, just specify it on the first line of the file. The dependency line and command will then be updated correspondingly.

Don't forget the parentheses when you refer to ***$(OBJECTS***). Macros may resemble shell variables like ***$HOME*** and ***$PATH***, but they're not the same.

One macro can be defined in terms of another macro, so you could say something like:

```
ROOT = /usr/local
HEADERS = $(ROOT) / include
SOURCES = $(ROOT) / src
```

In this case, HEADERS evaluates to the directory /usr/local/include and SOURCES to /usr/local/src. If you are installing this package on your system and don't want it to be in /usr/local, just choose another name and change the line that defines **ROOT**.

By the way, you don't have to use uppercase names for macros, but that's universal convention.

An extension in GNU make allows you to add to the definition of a macro. This uses a := string in place of an equal sing:

DRIVERS =drivers/block/block.a

ifdef CONFIG_SCSI
DRIVERS := $ (DRIVERS) drivers/scsi/scsi.a
endif

The first line is a normal macro definition, setting the DRIVERS macro to the filename driver/block/block.a. The next definition adds the filename drivers/scsi/scsi.a. But it takes effect only if the macro CONFIG_SCSI is defined. The full definition in that case becomes:

drivers/block/block.a drivers/scsi/scsi.a

So how do you define CONFIG_SCSI? You could put it in the makefile, assigning any string you want:

CONFIG_SCSI = yes

But you'll probably find it easier to on the make command line. Here's how to do it:

papaya$ make CONFIG_SCSI=yes target_name

One subtlety of using macros is that you can leave them undefined. If no one define them, a null string is substituted (that is, you end up with nothing where the macro is supposed to be). But this also gives you the option of defining the macro as an environment variable. For instance, if you don't define ***CONFIG_SCSI*** in the makefile, you could put this in your. ***bashrc*** file, for use with the bash shell:

export CONFIG_SCSI=yes

Or put this in .cshrc if you useee csh or tcsh:

setrnv CONFIG_SCSI=yes

All your builds will then have **CONFIG_SCSI** defined.

Suffix Rules and Pattern Rules

For something as routine as building an object file from a source file, you don't want to specify every single dependency in your makefile.
Unix compilers enforce a simple standard (compile a file ending in the suffix .c to create a file ending in the suffix .o) and ***make*** provides a feature called suffix rules to cover all such files.

Here's a simple suffix rule to compile a C source file, which you could put in your makefile**:**

```
.c.o:
        gcc –c $(CFLAG) $<
```

The .c.o: line means, "use a .c prerequisite to build a .o file." ***CFLAGS*** is a macro into which you can plug any compiler options you want: ***-g*** for debugging, for **instance, or *–O*** for optimization. The string $< is a cryptic way of saying "the prerequisite." So the name of your .c file is plugged in when ***make*** executes this command.

Here's a sample run using this suffix rule. The command line passes both the ***–g*** option and the ***–O*** option:

```
papaya$ make CFLAGS="-o –g" edit.o
gcc –c –O –g edit.c
```

You actually don't have to specify this rule in your makefile, because something very similar is already built into make. It even uses ***CFLAGS***, so you can determine the option used for compiling just by setting that variable. The makefile used to build the Linux kernel currently contains the following definition, a whole slew of ***gcc*** options:

```
CFLAGS = -Wall –Wstrict-prototypes –O2 –fomit-frame-pointer –pipe
```

The –D option is used to define symbols appearing in #ifdefs, you may need to pass lots of such options to your makefile, such as ***–DDEBUG or –DBSD***. If you do this on the make command line, be sure to put the shell to pass the set to your makefile as one argument:

```
papaya$ make CFLAGS="-DDEBUG –DBSD"  . . .
```

GNU make offers something called pattern rules, which are even better than suffix rules. A pattern rule is just like a regular dependency line, but it contains percent signs instead of exact filenames.

We see the ***$<*** string to refer to the prerequisite, but we also see ***$@***, which refers to the output file. So the name of .o file is plugged in three. Both of these are built-in macros; make defines them every time it executes an entry.

Another common built-in macro is ***$****, which refers to the name of prerequisite stripped of the suffix. So if the prerequisite is ***edit.c***, the string ***$*.s*** would evaluate to ***edit.s*** (an assembly language source file).

Here's something useful you can do with a pattern rule that you can't do with a suffix rule: you add the string ***_dgb*** to the name of the output file, so that later you can tell that you compiled it with debugging information:

```
%_dgb.o: %.c
        gcc –c –g –o $@ &(CFLAGS) $<

DEBUG_OBJECTS = mai_dbg.o edit_dbg.o

edimh_dbg: $(DEBUG_OBJECTS)
        Gcc –o $@ $(DEBUG_OBJECTS)
```

Now you can build all your objects in two different ways: one with debugging information and one without. They'll have different filenames, so you can keep them in one directory:

```
papaya$ make edimh_dbg
gcc –c –g –o main_dbg.o main.c
gcc –c –g –o edit_dbg.o edit.c
gcc –o edimh_dbg main_dbg.o edit_dbg.o
```

Multiple Command

Any shell commands can be executed in a makefile. But things can get kind of complicated because make executes each command in a separate shell. So this would not work:

```
target:
        cd obj
        HOST_DIR=/home/e
```

```
        mv *.o $HOST_DIR
```

Neither the cd command nor the definitions of the variable ***HOST_DIR*** have any effect on subsequent commands. You have to string everything together into one command. The shell uses a semicolon as a separator between commands, so you can combine them all on one line like this:

```
target:
        cd obj ; HOST_DIR=/home/e ; mv *.o $$HOST_DIR
```

One more change: to define and use a shell variable within the command, you have to double the dollar sign. This lets make know that you mean it to be a shell variable, not a macro.

You may find the file easier to read you break the semicolon-separated commands onto multiple lines, using backslashes so that make considers them one line:

```
target:
        cd obj ; \
        HOST_DIR=/home/e ; \
        mv *.o  $$HOST_DIR
```

Sometimes makefiles contain their own make commands; this is called recursive make. It looks like this:

```
linuxsubdirrs: dummy
        set  -e; for I in $(SUBDIRS); do $(MAKE) –c $$I; done
```

The macro $(MAKE) invokes make. There are a few reasons for nesting makes. One reason, which applies to this examples, is to perform builds in multiple directories (each of these other directories has to contain its own makefile). Another reason is to define macros on the command line, so you can do builds with a variety of macro definitions.

GNU makes offer another powerful interface to the shell as an extension. You can issue a shell command and assign its output to a macro. A couple of examples can be found in the Linux kernel makefile, but we'll just show a simple example here:

```
HOST_NAME = $(shell uname –n)
```

This assign the name of your network node-the output of the uname –n co,,and-to the macro HOST_NAME.

make offers a couple of conventions you may occasionally want to use. One is to put an at-sign before a command, which keeps make from echoing the command when it's executed:

```
@if [ -x /bin/dnsdomainname ]; then \
        echo #define LINUX_COMPILE_DOMAIN \ " 'dnsdomainname'\"; \
else \
        echo #define LINUX_COMPILE_DOMAIN \ " 'domainname'\"; \
fi >> tools/version.h
```

Another convention is to put a hyphen before a command, which tells make to keep going even if the command fails. This may be useful if you want to continue after an mv or cp command fails:

- **mv sdimh /usr/local;**
- **mv readimh /usr/local**

Include Other Makefiles

Larger projects tend to break parts of their makefiles into separate files. This makes it easy for different makefiles in different directories to share things, particularly macro definitions. The line:

include filename

reads in the contents of filename. You can see this in Linux kernel makefile, for instance:

include .depend

If you look in the file .depend, you'll find a bunch of makefile entries: to be exact, lines declaring that object files depend on header files. (By the way, .depend might not exist yet; it has to be created by another entry in the makefile.)

Sometimes include lines refer to macros instead of the filename, as in:

include $(INC_FILE)

In this case, INC_FILE must be defined either as an environment variable or as a macro. Doing things this way gives you more control over which file is used.