by Peter Collinson, Hillside Systems

DENISE ORTAKALES

make: Mastermind of the Update

It was interesting to read Rich Morin's I/Opener article "Page Processing in Perl" in *SunExpert* (March 1998, Page 43). It turns out that I have reached a similar conclusion about generating Web pages. Like Rich, I've also written a processor routine in Perl that takes page descriptions and spits out pages.

Perhaps four years back, I realized that each of my Web pages has a large chunk of HTML that remains essentially the same across any particular set of pages. The constant part usually provides a consistent look and feel across a whole Web site, or part of one. I now create pages by writing a small Perl program that fills in a template taken from a file, where the variable sections in the template are supplied from the Perl code. Creating these little Perl program description files is easy, because all you supply is the information that differs on each page. Also, if I want to make global changes to the site, I can alter the template file and recompile the pages. I can change the site without the massive time-consuming, error-prone task of editing each page individually.

My processor is designed to be a simple classical macro processor, although it's not that elegant and only does a bare-bones job. The processor scans the source file for a name (enclosed in double braces, `{{CONTENT}}`, for example) and replaces any embraced name it finds with contents of a variable in the Perl program (taken from an associative array, for Perl-literate readers). There are also simple macro definitions and conditional facilities.

I seem to be using this beast on every Web site I design. It can compile static pages from simple short Perl programs that are applied to a template file, provide dynamic pages from CGI scripts in response to input from forms and add addressee information and other material into stock email messages.

However, the point of this article is not to discuss the ins and outs of converting Perl programs into HTML documents, but to look at the way I've automated the compilation system that uses it. I want to ensure that if I change a page description, I'll run the appropriate programs on the appropriate files to generate the updated pages. The update task is masterminded by a standard UNIX tool: `make`.

I've discussed `make` in this magazine before. I wrote two consecutive articles, "Make: Parts I and II," February 1992, Page 34, and March 1992, Page 26. This was back in the dim, distant past for many readers, so I make no apology for revisiting some of the details of `make` again–although this article does contain information that was not discussed previously.

make: The Basics

The notion that underlies `make` is easy to understand. Many files on a UNIX system are created by applying a program to a set of source files. It's a simple observation that the modification time stored with each generated file must be later than the time stored for

each of its component source files. In make-speak, each generated file *depends* on the set of source files.

If we discover that the modification time on one of the source files is later than that on a generated file, then the generated file is out-of-date and needs to be updated. The make command is fed a user-generated control file that provides a specification of how a particular generated file, the *target*, depends on a set of source files. The control file contains a *rule*, or a set of rules, used to convert each source file into a target file. Unless it is given a specific file name, the make command first looks for a control file called makefile, then for a file called Makefile. You can choose which flavor you prefer. Most people these days use the capitalized name, and many versions of make now complain if both files are present in the same directory. To avoid confusion, I'll call one of these files a "makefile."

The makefile will contain several definitions of the form

```
target: dependency list
<tab>  rule
```

The syntax tells make that the particular *target* depends on a list of files, and to create the target the *rule* should be executed. In most versions of make there must be a tab character before the rule, and I've emphasized that here, I'll be showing white space in the examples that follow. The rule is simply a shell command that is run by make. It is assumed to create the target. In general, if the command fails returning an error status, then make will spot this and terminate any further execution.

Actually, the rule section can contain several commands, appearing on several lines. Each command is run in a separate shell, and this can catch you out sometimes.

Let's look at a specific example. The easiest way to show you what happens is to pick up a programming task, but the strategy can be applied to many nonprogramming jobs, so don't think this article is aimed just at programmers.

If we are compiling a C program from a source file called hello.c, then we might create a makefile that contains

```
hello: hello.c
    cc -o hello -O hello.c
```

The target is a compiled program called hello that depends on the source file hello.c. If the modification time on hello.c is later than that on hello, then we will invoke the rule. The C compiler is called and told to place its output into a file called hello (the -o option). We've also asked for the compiler to run the optimizer as part of its work (the -O option).

I should emphasize that if the target file hello was created more recently than hello.c, then the rule is not run by make. However, if we omit the dependency, starting the line with

```
hello:
```

then the rule will always be run. The trick with constructing makefiles is to get the dependencies right.

The example above is a no-brainer. There is no huge win when compiling a single file, except that you can type make as a reflex action and not have to worry about how the program is compiled. Actually, the ability to just type make can be a big win, saving a lot of time in understanding just how the command is compiled.

However, the real power of the make command is apparent when there are several source files to be compiled into the final target. For example,

```
hello: a.o b.o
    cc -o hello a.o b.o

a.o: a.c header.h
    cc -c -O a.c

b.o: b.c header.h
    cc -c -O b.c
```

The file starts with the final target, the hello program that depends on two compiled modules, a.o and b.o. We place the final target first because by default make will create the first target that it finds in the makefile. So when we type make, a tree of dependencies will be built, and that tree is then traversed looking for work to be done.

If the targets a.o and b.o are not current, they are each created from their own rules. This will result in separate runs of the C compiler that is given the -c switch telling it to create a compiled module. Notice that each compilation depends on the header.h file. If I change the header file, then both a.c and b.c will be recompiled. When a.o and b.o exist and are more recent than the current version of the hello file, then the rule to create the final target will be executed.

Incidentally, I mentioned that make could be given targets to work on, so we can type

```
$ make a.o
```

to run the system to create that particular intermediate step. This feature is perhaps more useful if you have several programs to be compiled in the same makefile:

```
all: prog1 prog2

prog1: prog1.o proghdr.h
    cc -c -O prog1.c

prog2: prog2.o proghdr.h
    cc -c -O prog2.c
and so on
```

Now we can type make to create both programs and select one or the other by supplying a specific request on the command line. This makefile also demonstrates that the rule section can be empty.

In many makefiles, you'll find a bunch of ad-hoc targets

that perform useful functions:

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

The `clean` target is used to ensure that created files in the directory are removed. The hyphen preceeding the `rm` command is a small bit of magic. Normally `rm` will complain and return a failure status when it is given a file to delete that doesn't exist. The nonzero status will cause `make` to terminate because one of its children failed. We can tell `make` that we can tolerate failure by placing a hyphen before the command. Quite often, I'll add the `-f` flag to an `rm` command in a makefile because the `-f` flag changes the default behavior; the command doesn't complain and die if a file doesn't exist.

Other common targets in makefiles are: `install` to place the final binary into public use in the file system; `dist` to create a distribution, perhaps using the `tar` command to create a file that can be transported elsewhere; `print` to print the sources; and `depend` to create a dependency set that is appended to the makefile.

More Syntax

Well, all the above concentrates somewhat on programming, but we can use `make` for any purpose where the modification time of files is important. For example, let's say we are working on several important files in a directory and want to create a checkpoint copy from time to time. We create a subdirectory called `checkpoint` and write a makefile like this:

```
FILES= file1 file2 file3 file4 file5

check:  $(FILES)
    cp $? checkpoint
    touch check
```

We type `make` and only the files that we have altered since the last run will be copied into the `checkpoint` directory. I've introduced some new things into this makefile. First, you can define macro replacements by using the `NAME=`*list* statement. When `make` encounters a `$(NAME)` elsewhere in the makefile, the text is replaced by the contents of the list. So in the example above, our main target, `check`, depends on all the files in the list `FILES`.

The dollar syntax is used to introduce other results of the dependency check. Here we are using `$?`, which is replaced by the list of files that are newer than the target. There are other magic dollar values that I'll get to later. So if `file2` and `file4` are newer than the target, and we type `make`, then the `cp` command becomes

```
cp file2 file4 checkpoint
```

The target is a file called `check` that is there simply to record the time that the last checkpoint copy was made. It's created by the `touch` command, which exists on all UNIX systems and simply changes a named file, updating its modification time.

We should probably add a way of making a complete new checkpoint copy to the makefile instantly:

```
checkpoint:
    cp $(FILES) checkpoint
    touch check
```

so we can now say

```
$ make checkpoint
```

and make a complete copy of all the files. Also, in this type of makefile, I like to add something that I can use to obtain all the names of the "interesting" files:

```
names:
    echo $(FILES)
```

which means I can use the backquote operator in the shell using the names. The command

```
$ grep fred 'make names'
```

will look for `fred` in the files that are relevant in the current directory.

Implicit Rules

To make things somewhat easier for programmers, `make` comes with a set of default rules that are generally loaded when the command starts. The default rule set used to be part of the binary of the program, but in recent years it has migrated to a file stored in some public place; it's `/usr/share/lib/make/make.rules` on my Sun running Solaris 2.6.

The default rules work with file suffixes, so the rules understand that a file ending in `.c` is a C source file, a file ending in `.o` is an object file containing a compiled module and so on. The `make` program itself knows only about suffixes and is given a strong hint of the order in which files will appear by the contents of the special target `.SUFFIXES`. The target is given all the suffixes that the rules know about. A simplified default version for C programmers might be

```
.SUFFIXES= .o .c .h
```

The order of the list is important. Target suffixes come first, so programs are made from `.o` files that are created from `.c` files. C program files can also include header files that will end in `.h`. The default file will contain a set of special rules that specify how to turn a file with one suffix into a file with another, so to make a `.c` file into a `.o` file you will see a rule like this:

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

The various macro definitions here will be established with default settings, so `CC` will be `cc`, and `CFLAGS` and `CPPFLAGS`

are set to null strings. The purpose of these definitions is to permit the user to set values in their own makefiles that establish local definitions for the default rules. The `$<` at the end of the line is another magic variable that is replaced with the name of the dependency file. This rule can now be used whenever it's necessary to create a `.o` file from a `.c` file, so our makefile to create `hello` from `a.c` and `b.c` can be made much simpler:

```
OBJS=a.o b.o
CFLAGS=-O

hello: $(OBJS)
    cc -o hello $(OBJS)
```

Notice that there is no explicit mention of the `.c` files. We've told `make` that it needs two object files and it will use the suffix rules to look for candidate source files. When using the full default set, these candidate files can be written in C (with a `.c` suffix), in C++ (with a `.cc` suffix), in FORTRAN (with a `.f` suffix) and so on. In this case, we are programming in C, so `make` will successfully find `a.c` and `b.c`, apply the compilation rule (using our version of `CFLAGS`) and create the program.

The good thing about creating a rule that applies to known suffixes is that the makefile becomes very simple. We can add new target files by adding their name to the TARGETS list.

Making Your Own Rules

Well, I started this article by discussing how I created HTML files from a Perl source, so let's return to that example. We can, of course, create makefiles that contain specific rules telling `make` how to create a `.html` file from a `.pl` file. This looks something like

```
page.html: page.pl
    $(PERL) -w page.pl > page.html
```

where we set the `PERL` macro value to be the location of the Perl binary on the system. The temptation is to cut and paste this first definition whenever we add a new page into the makefile and change the bits that need editing to specify how to create the new page from the brand-new Perl source file. However, in the long run, it's much simpler to create a rule and use the ability of `make` to deduce things about the files it needs from the rule. The makefile looks like this:

```
TARGETS=m.html p.html

PERL=/usr/local/bin/perl

.SUFFIXES: .html .pl

.pl.html:
    $(PERL) -w $< > $@

all: $(TARGETS)
```

The first line sets up the names of the HTML files that we are creating. When we want to add a new file, all we do is add a new target name to the list. In fact, the makefile only has one target, `all`, and we tell `make` that it's dependent on the files we need to create.

We use the implicit rule that is defined in the middle of the file to do the work of creating the target files. We must first tell `make` that we will be making a `.html` file from a `.pl` file, and we do this by setting two new suffixes into the `.SUFFIXES` target. Actually, this adds the two new suffixes to the current extant list, so to clear the list we'd need to say:

```
.SUFFIXES:
.SUFFIXES: .html .pl
```

Once we have defined the two new suffixes we're using, we can define a rule that is used to create a `.html` file from a `.pl` file. This runs the Perl interpreter with the `-w` option to help with error checking–it takes its program from a file in the dependency list (`$<`) to its equivalent target (`$@`).

We are now in a situation where any change in a `.pl` file will result in the appropriate `.html` file being rebuilt when `make` is typed. However, I've also said that the Perl script uses a template file, and it would be a good idea to rebuild the pages whenever that template file is altered. We need to add an explicit dependency to force a rebuild. We can do this by adding

```
$(TARGETS): template
```

to the end of the makefile. Now when the template file changes, all the files in the `TARGETS` list automatically become out-of-date and are rebuilt.

The good thing about creating a rule that applies to known suffixes is that the makefile becomes very simple. We can add new target files by adding their name to the `TARGETS` list. The makefile can be easily reused in new parts of the system by copying it and replacing the `TARGETS` list with the new targets that are appropriate for that directory.

Finally

If you are running Solaris, and type `make`, and get a "Command not found" message, then you need to be aware that the `make` command lives in `/usr/ccs/bin` by default. You need to change your search path to include that directory. Check that the directory exists on your system first, and if not, politely ask your systems administrator to load it for you. It's an option to the standard installed system.

***Peter Collinson** runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.