Linux Programing Hints

Oct 01, 1994 By Michael K. Johnson (/user/1000867)

in

(/issue/6)

From Issue #6 October 1994 (/issue/6)

Most Linux users have at least heard of Makefiles, but many do not know how powerful a program make is. It is thought of as a tool for maintaining other programs, but it is far more. It can make sense out of chaos in any project where some files are created from other files, whether the end product is a program or a book or an automated post to Usenet. Even if you have never written a makefile, th

An Introduction to make

Most Linux users have at least heard of Makefiles, but many do not know how powerful a program make is. It is thought of as a tool for maintaining other programs, but it is far more. It can make sense out of chaos in any project where some files are created from other files, whether the end product is a program or a book or an automated post to Usenet. Even if you have never written a makefile, this tutorial will set you on your way to using **make** effectively.

by Michael K. Johnson

Many people are confused by **make**: maybe you are too. You know that it is hard to use, because it has a weird syntax unlike any other program you use. If you are lucky, you have been warned that it is important to have tab characters (not spaces) in certain places, and you know that if you mess up a makefile you won't be able to fix it.

Writing a makefile of your own is out of the question. It is difficult, and besides, you aren't really a programmer, anyway. What could this programmer's maintenance program do for you, and why should you learn its weird syntax? Or maybe you are a programmer, and you don't want to learn this tool called make, which has a syntax different from any language you have ever learned.

The reason for the weird syntax is that make does a job very different from normal programming tools, and it is well-suited to that very different job. Understanding the job is the first step to understanding make. Once you understand the job, and have learned a little bit about make, you will be able to write short, powerful makefiles.

We will pretend for the moment that you are writing a book, although the exact same ideas apply to writing a program. (I just want to emphasize that make isn't just for Real Programmers.) You are using LaTeX. Each chapter has several figures. These figures are done in xfig, and need to be converted to PostScript format with fig2dev before being included in your book.

Here comes the problem. You are occasionally editing the figures with xfig, and forgetting to make a PostScript copy of each figure when you are done, so you write a large shell script that converts xfig to encapsulated PostScript (EPS) for each figure. It is large, bulky, and inflexible, but you get the job done right each time you print. Unfortunately, it takes a while to convert all the files from xfig to encapsulated PostScript. Even if you have only made a minor change in one figure, it re-converts all of your figures. This is annoying, and takes a while for it to convert your 80 or so figures.

Welcome to the wonderful world of make

The intelligence of make is summarized by two concepts: dependencies and rules. You need to tell make that the dvi file (we'll call it book.dvi) needs to be created from the main LaTeX file (book.tex), and from the encapsulated PostScript files (*.eps). In make terminology, book.dvi depends on book.tex and *.eps. Also, each of the .eps files depends on the corresponding .fig file. You also need to give make rules for turning book.tex into book.dvi, and for turning the .fig files into .eps files.

Newbie Note:All lines containing shell commands in your makefile must start with the TAB character.

Here is an example of a makefile that will do everything that is necessary:

```
0:  # This is a makefile to create book.dvi
1:  EPSFIGS = fig1.eps fig2.eps fig3.eps fig4.eps \
2:  fig5.eps fig6.eps fig7.eps fig8.eps fig9.eps \
3:   <... more .eps files, too many to print ...> \
4:   fig78.eps fig79.eps fig80.eps fig81.eps
5:
6:  book.dvi: book.tex $(EPSFIGS)
7:          latex book.tex
8:
9:  fig1.eps: fig1.fig
10:         fig2dev -Lps fig1.fig fig1.eps
11:
12: fig2.eps: fig2.fig
13:         fig2dev -Lps fig2.fig fig1.eps
14:
<many more similar rules that you can imagine>
220:
```

This file can be saved as **makefile** or **Makefile**; either will work. If you have both files in the same directory, **makefile** will be used instead of **Makefile**. In addition, there are other names that can be used and rules to control that. See the *GNU Make* manual if you care (you usually won't). People almost always use **Makefile** because capital letters show up at the top of directory listings.

The first line, line 0, is a comment. Line 1 starts to define the variable EPSFIGS. The backslashes continue the line, so the EPSFIGS variable contains the names of all the EPS files from fig1.eps through fig81.eps, and logically all those lines are really one long line. Line 6 tells make that if book.tex or any of the .eps files are newer than book.dvi, then book.dvi has to be recreated. Line 7 explains how to do this. It is very important that this line start with a TAB character. This is how make knows that this is a shell command to be executed to update the dependency that preceeds it. Eight spaces will not work. Spaces can follow a tab, but the first character on that line must be a TAB. The rest of the lines work the same way: line 9 says that fig1.eps depends on fig1.fig, and line 10 tells make how to update fig1.eps from fig1.fig if fig1.fig has been updated since the last time fig1.eps was created.

Simply typing **make** will automatically make book.dvi, because the first target in the makefile (here consisting of lines 6 and 7) is the **default target**. You could conceivably type **"make fig1.eps"** to just update fig1.eps from fig1.fig, and that only if necessary. If it is not necessary, make will tell you **"fig1.eps is up to date."**

The basic syntax of a makefile can be simplified to this:

- Any line can be continued onto the next line by making the last character of the line be a backslash character.
- Variables are defined with lines containing an equal sign: **FOO=bar**.
- Variables are referenced by enclosing them in parentheses (or curly braces, but parentheses are preferred and are more portable) and prepending a dollar sign: **$(FOO)** (or **${FOO}**).
- Files are made to depend on others by putting the file that is created before a colon, and a list of files needed to create or update that file after the colon, on

the same line.

- A list of shell commands for creating or updating the file follows that line directly on lines with a TAB character as the very first character. Each line is run by a separate invocation of the shell, so a cd command on one line will only have effect on that line. To make successive lines be part of the same shell invocation, append " ;\" to the line to make the next line really be another part of the same line.
- Comments begin with the "#" character.

Knowing those six very simple rules will allow you to maintain most makefiles that you will find on the Internet, and will allow you to create almost any makefile you need. However--

_____