viralpatel.net
Viral Patel's home page

Search

Tech Blog NEW | Taj OS | Operating System Tutorials | Guest Book | Sitemap

Menu
Home
FAQ
Screen Shots
Download
Tutorials
Any Comments?

Comments
Ankit
09 Dec 2011
I don't know how to install from .img fi..
>> show
Masum
22 Nov 2011
How to install Taj os in my pc. and ..
>> show
Me
21 Oct 2011
TAJ seems to lock up in boot using Bochs..
>> show

Recent Tutorials
Graphical User Interface
Protected mode
Booting Process
CHS to LBA Translation
Makefile Tutorial
Real mode
Writing Hello World Bootloader
FAT File System
Partition Table
Segmentation
View All »

Makefile Tutorial

Introduction

Make is one of the original Unix tools for Software Engineering. By S.I. Feldman of AT&T Bell Labs circa 1975. But there are public domain versions (eg. GNU) and versions for other systems (eg. Vax/VMS).

Related tools are the language compilers (cc, f77, lex, yacc, etc.) and shell programming tools (eg. awk, sed, cp, rm, etc.). You need to know how to use these.

Important adjuncts are lint (source code checking for obvious errors) ctags (locate functions, etc. in source code) and mkdepend. These are nice, and good programmers use them.

Important, and related tools, are the software revision systems SCCS (Source Code Control System) and RCS (Revision Control System -- the recommended choice)

The idea is to automate and optimize the construction of programs/files -- ie. to leave enough foot prints so that others can follow.

Makefile Naming

make is going to look for a file called *Makefile*, if not found then a file called *makefile*. Use the first (so the name stands out in listings).

You can get away without any Makefile (but shouldn't)! Make has default rules it knows about.

Makefile Components

- **Comments**

  Comments are any text beginning with the pound (#) sign. A comment can start anywhere on a line and continue until the end of the line. For example:

  ```
  # $Id: slides,v 1.2 1992/02/14 21:00:58 reggers Exp $
  ```

- **Macros**

  Make has a simple macro definition and substitution mechanism. Macros are defined in a Makefile as = pairs. For example:

  ```
  MACROS=  -me
  PSROFF=  groff -Tps
  DITROFF= groff -Tdvi
  CFLAGS= -O -systype bsd43
  ```

  There are lots of default macros -- you should honor the existing naming conventions. To find out what rules/macros make is using type:

  ```
  % make -p
  ```

  NOTE: That your environment variables are exported into the make as macros. They will override the defaults.

  You can set macros on the make command line:

  ```
  % make "CFLAGS= -O" "LDFLAGS=-s" printenv
     cc -O printenv.c -s -o printenv
  ```

- **Targets**

  You make a particular target (eg. make all), in none specified then the first target found:

  ```
  paper.dvi: $(SRCS)
         $(DITROFF) $(MACROS) $(SRCS) >paper.dvi
  ```

  NOTE: The the line beginning with `$(DITROFF)` begins with TAB not spaces.
  The target is made if any of the dependent files have changed. The dependent files in this case are represented by the `$(SRCS)` statement.

- **Continuation of Lines**

  Use a back slash (\). This is important for long macros and/or rules.

- **Conventional Macros**

  There are lots of default macros (type "make -p" to print out the defaults). Most are pretty obvious from the rules in which they are used:

  ```
  AR = ar
  GFLAGS =
  GET = get
  ASFLAGS =
  MAS = mas
  AS = as
  FC = f77
  CFLAGS =
  CC = cc
  LDFLAGS =
  LD = ld
  LFLAGS =
  LEX = lex
  YFLAGS =
  YACC = yacc
  LOADLIBS =
  MAKE = make
  MAKEARGS = 'SHELL=/bin/sh'
  SHELL = /bin/sh
  MAKEFLAGS = b
  ```

- **Special Macros**

  Before issuing any command in a target rule set there are certain special macros predefined.

  1. $@ is the name of the file to be made.
  2. $? is the names of the changed dependents.

  So, for example, we could use a rule

  ```

  printenv: printenv.c
          $(CC) $(CFLAGS) $? $(LDFLAGS) -o $@
  ```

  alternatively:

  ```
  printenv: printenv.c
          $(CC) $(CFLAGS) $@.c $(LDFLAGS) -o $@
  ```

  There are two more special macros used in implicit rules. They are:

  1. $< the name of the related file that caused the action.
  2. $* the prefix shared by target and dependent files.

- **Makefile Target Rules**

  The general syntax of a Makefile Target Rule is

  ```
      target [target...] : [dependent ....]
      [ command ...]
  ```

  Items in brackets are optional, ellipsis means one or more. Note the tab to preface each command is required.

  The semantics is pretty simple. When you say "make target" make finds the target rule that applies and, if any of the dependents are newer than the target, make executes the com- mands one at a time (after macro substitution). If any dependents have to be made, that happens first (so you have a recursion).

  A make will terminate if any command returns a failure sta- tus. That's why you see rules like:

  ```
  clean:
          -rm *.o *~ core paper
  ```

  Make ignores the returned status on command lines that begin with a dash. eg. who cares if there is no core file?

  Make will echo the commands, after macro substition to show you what's happening as it happens. Sometimes you might want to turn that off. For example:

  ```
  install:
          @echo You must be root to install
  ```

- **Example Target Rules**

  For example, to manage sources stored within RCS (sometimes you'll need to "check out" a source file):

  ```
  SRCS=x.c y.c z.c

  $(SRCS):
          co $@
  ```

  To manage sources stored within SCCS (sometimes you'll need to "get" a source file):

  ```
  $(SRCS):
          sccs get $@
  ```

  Alternativley, to manage sources stored within SCCS or RCS let's generalize with a macro that we can set as required.

  ```
  SRCS=x.c y.c z.c
  # GET= sccs get
  GET= co

  $(SRCS):
          $(GET) $@
  ```

  For example, to construct a library of object files

  ```
  lib.a: x.o y.o z.o
          ar rvu lib.a x.o y.o z.o
          ranlib lib.a
  ```

  Alternatively, to be a bit more fancy you could use:

  ```
  OBJ=x.o y.o z.o
  AR=ar

  lib.a: $(OBJ)
          $(AR) rvu $@ $(OBJ)
          ranlib $@
  ```

  Since AR is a default macro already assigned to "ar" you can get away without defining it (but shouldn't).

If you get used to using macros you'll be able to make a few rules that you can use over and over again. For example, to construct a library in some other directory

```
INC=../misc
OTHERS=../misc/lib.a

$(OTHERS):
        cd $(INC); make lib.a
```

Beware:, the following will not work (but you'd think it should)

```
INC=../misc
OTHERS=../misc/lib.a

$(OTHERS):
        cd $(INC)
        make lib.a
```

Each command in the target rule is executed in a separate shell. This makes for some interesting constructs and long continuation lines.

To generate a tags file

```
SRCS=x.c y.c z.c
CTAGS=ctags -x >tags

tags:   $(SRCS)
        ${CTAGS} $(SRCS)
```

On large projects a tags file, that lists all functions and their invocations is a handy tool.
To generate a listing of likely bugs in your problems

```
lint:
        lint $(CFLAGS) $(SRCS)
```

Lint is a really good tool for finding those obvious bugs that slip into programs -- eg. type classes, bad argu- ment list, etc.

- **Some Basic Make Rule**

  People have come to expect certain targets in Makefiles. You should always browse first, but it's reasonable to expect that the targets all (or just make), install, and clean will be found.

  1. **make all** -- should compile everything so that you can do local testing before installing things.
  2. **make install** -- should install things in the right places. But watch out that things are installed in the right place for your system.
  3. **make clean** -- should clean things up. Get rid of the executables, any temporary files, object files, etc.

  You may encounter other common targets, some have been already mentioned (tags and lint).

- **An Example Makefile for printenv**

```
# make the printenv command
#
OWNER=bin
GROUP=bin
CTAGS= ctags -x >tags
CFLAGS= -O
LDFLAGS= -s
CC=cc
GET=co
SRCS=printenv.c
OBJS=printenv.o
SHAR=shar
MANDIR=/usr/man/manl/printenv.l
BINDIR=/usr/local/bin
```

```
DEPEND= makedepend $(CFLAGS)
all:    printenv

# To get things out of the revision control system
$(SRCS):
        $(GET) $@
# To make an object from source
       $(CC) $(CFLAGS) -c $*.c

# To make an executable

printenv: $(OBJS)
        $(CC) $(LDFLAGS) -o $@ $(OBJS)

# To install things in the right place
install: printenv printenv.man
        $(INSTALL) -c -o $(OWNER) -g $(GROUP) -m 755 printenv $(BINDIR)
        $(INSTALL) -c -o $(OWNER) -g $(GROUP) -m 644 printenv.man $(MANDIR)

# where are functions/procedures?
tags: $(SRCS)
        $(CTAGS) $(SRCS)

# what have I done wrong?
lint: $(SRCS)
        lint $(CFLAGS) $(SRCS)

# what are the source dependencies
depend: $(SRCS)
        $(DEPEND) $(SRCS)

# to make a shar distribution
shar:   clean
        $(SHAR) README Makefile printenv.man $(SRCS) >shar

# clean out the dross
clean:
        -rm printenv *~ *.o *.bak core tags shar

# DO NOT DELETE THIS LINE -- make depend depends on it.
printenv.o: /usr/include/stdio.h
```

- **Makefile Implicit Rules**

  Consider the rule we used for printenv

  ```

  printenv: printenv.c
          $(CC) $(CFLAGS) printenv.c $(LDFLAGS) -o printenv
  ```

  We generalized a bit to get

  ```
  printenv: printenv.c
          $(CC) $(CFLAGS) $@.c $(LDFLAGS) -o $@
  ```

  The command is one that ought to work in all cases where we build an executable x out of the source code x.c This can be stated as an implicit rule:

  ```
  .c:
          $(CC) $(CFLAGS) $@.c $(LDFLAGS) -o $@
  ```

  This Implicit rule says how to make x out of x.c -- run cc on x.c and call the output x. The rule is implicit because no particular target is mentioned. It can be used in all cases.

  Another common implicit rule is for the construction of .o (object) files out of .c (source files).

  ```
  .o.c:
  ```

```
        $(CC) $(CFLAGS) -c $<
```

alternatively

```
.o.c:
        $(CC) $(CFLAGS) -c $*.c
```

- **Make Dependencies**

  It's pretty common to have source code that uses include files. For example:

```
% cat program.c

#include
#include        "defs.h"
#include        "glob.h"
        etc....
main(argc,argv)
        etc...
```

The implicit rule only covers part of the source code depen- dency (it only knows that program.o depends on program.c). The usual method for handling this is to list the dependen- cies separately;

```
        etc...
        $(CC) $(CFLAGS) -c $*.c
        etc...
program.o: program.c defs.h glob.h
```

Usually an implicit rule and a separate list of dependencies is all you need. And it ought to be easy enough to figure out what the dependencies are.

However, there are a number of nice tools around that will automatically generate dependency lists for you. For example (trivial):

```
DEPEND= makedepend $(CFLAGS)
        etc...
# what are the source dependencies

depend: $(SRCS)
        $(DEPEND) $(SRCS)

        etc....
# DO NOT DELETE THIS LINE -- ....

printenv.o: /usr/include/stdio.h
```

These tools (mkdepend, mkmkf, etc.) are very common these days and aren't too difficult to use or understand. They're just shell scripts that run cpp (or cc -M, or etc.) to find out what all the include dependencies are. They then just tack the dependency list onto the end of the Makefile.