Managing Projects With make(1)

Working with make, making it work.

Oleg Goldshmidt

olegg@il.ibm.com

IBM Haifa Research Laboratories

Agenda

- Introduction
- A disorganized tour of makefiles
  - stopping here and there and consulting the guidebook
- A few non-trivial real life makefile snippets
- A few software engineering tips
- Useful make options
- The makefile that builds this presentation

What `make(1)` does

- not just compilation
- general project management
- a very good example of the UNIX/Linux philosophy:
  - different tools work well together
  - single protocol for inter-tool communication — ASCII
  - `make(1)` is a “glue” that makes different tools “stick together”
- in big projects, you want to only do what is really needed for each goal
- `make(1)` is very good at tracing dependencies and doing only what is really needed
- `make(1)` is a programming language

GNU make

- another creation of Richard Stallman and Roland McGrath
- conforms to POSIX.2 (IEEE Standard 1003.2-1992)
- very portable — runs on everything that generates heat
- some nifty features compared to other implementations
  - will be mentioned later
- some features missing compared to other implementations
  - none of the features are mandated by POSIX.2

Makefiles: Contents

- different targets, a.k.a. goals
- target dependencies, a.k.a. prerequisites
- rules, i.e., how to do parts of project management (specific targets)
  - often how to build prerequisites for other targets
- variables, a.k.a. macros
- directives — commands to do something special while reading a makefile
  - include another makefile
  - conditionals
  - verbatim definitions (for canned command sequences etc.)

Makefiles: Naming Conventions

- by default GNU make looks for the following files in order: `GNUmakefile` , `makefile` , `Makefile`
- `GNUmakefile` is deprecated, should only be used for makefiles that will not be understood by other versions of `make(1)`
- other versions of `make(1)` will also look for `makefile` , `Makefile`
- recommend `Makefile` — appears early in the directory listing
  - we are in UNIX: file names are case-sensitive

Makefiles: Non-Standard Names

- use `-f NAME` or `--file=NAME` option
  - can be repeated — makefiles are concatenated in order
  - default makefiles are not checked
- conventional extension: `.mk` :
  - `make -f foo.mk -f ../other_makefiles/bar.mk my_program`
- no makefile at all — give a target and `make(1)` will try to find the right implicit rules

Basic Syntax

- Typical beginner's makefile contains targets and rules like

```
foo.o: foo.c foo.h bar.h
    cc -o foo.o -g foo.c # starts with tab!
```

- perfectly correct, but
  - not scalable (are we going to list dependencies by hand for all 958 files in the project?)
  - not flexible (what if we decide to compile with different flags, different compilers, etc?)

Variables: the Standard (Weird) Flavor

Expanded recursively. Example: file `var1.mk`

```
FOO = $(BAR)
BAR = $(XYZZY)
XYZZY = Yeah?

all:
        echo $(FOO)

$ make -f var1.mk
echo Yeah?
Yeah?
```

Variables: the Standard (Weird) Flavor

- advantage:

  ```
  CPPFLAGS = $(IFLAGS) -DDEBUG
  IFLAGS = -Ifoo -Ibar
  ```

  does what is intended.

- disadvantages
  - infinite loops

    ```
    CPPFLAGS = $(CPPFLAGS) -DDEBUG
    ```

    does not do what is intended (`make(1)` fails detecting infinite loop)
  - any functions in definition will be executed every time the variable is referenced — slow
  - wildcards and the `shell` function are uncontrollable

Variables: the GNU (Sane) Flavor

Expanded once, contains no references to other variables in definition, uses their values at the time of definition.

```
x := foo
y := bar
y := $(x) $(y)
x := later
```

- behave like variables in most “normal” programming languages
- makes programming more predictable
- allows using functions, wildcards in definitions
- specific to GNU `make(1)`

Variables: Finer Points

- leading whitespace is stripped, trailing whitespace is not

```
nullstring :=
space := $(nullstring) # a single space

dir := /foo/bar          # do not do this!
file := $(dir)/xyzzy     # contains spaces!
```

- conditional assignment

```
FOO ?= bar
```

  - only works if `FOO` is not defined
  - if `FOO` is set to empty value it is still defined

- `CPPFLAGS += -DDEBUG` does what you expect for GNU-style variables

Recursive Expansion and Appending

```
CFLAGS = $(WARNINGS) -O2
CFLAGS += -pg # enable profiling
WARNINGS = -W -Wall
echo:
        echo $(CFLAGS)

$ make -f append.mk # with "="
echo -W -Wall -O2 -pg
-W -Wall -O2 -pg

$ make -f append.mk # with ":="
echo  -O2 -pg
-O2 -pg
```

Built-In Variables

- $@: the current target, or the archive when target is an archive member
- $%: the target member name when target is an archive member
  - if the target is `foo.a(bar.o)` then $@ is `foo.a`, $% is `bar.o`
- $<: the first prerequisite
- $?: all the prerequisites newer than the target
- $^: all the prerequisites, no duplicates
- $+: all the prerequisites, with duplicates
  - useful for linking, where repetitions count
- $*: the stem that an implicit rule matches

Built-In Rules

```
%.o:    %.c
    $(CC) -o $@ -c $(CFLAGS) $(CPPFLAGS) $<
%.o:    %.cc
    $(CXX) -o $@ -c $(CXXFLAGS) $(CPPFLAGS) $<
%.o:    %.s
    $(AS) -o $@ $(ASFLAGS) $<
%.s:    %.S
    $(CPP) $(CPPFLAGS) $< $@
%.c:    %.y
    $(YACC) $(YFLAGS) $<
%.c:     %.l
    $(LEX) $(LFLAGS) $<
libfubar.a:    foo.o bar.o xyzzy.o
    $(AR) $(ARFLAGS) $@ $?
```

Changing Automatic Variables

- command line: `make CC=insight++`
- environment: `export CC=insight++; make -e`
  - this ignores the value of `SHELL`
- target-specific variables

```
CFLAGS := -W -Wall -pedantic
LDFLAGS :=
OBJ := foo.o
prog:   $(OBJ)
        $(CC) -o $@ $^ $(LDFLAGS)
prof:   CFLAGS += -pg
prof:   LDFLAGS += -pg
prof:   prog
```

Conditionals

```
ifdef BUILD_SHARED
CFLAGS += -fPIC
endif

ifeq ($(CC),gcc)
CPPFLAGS += -DGCC_SOURCE=1
else
CPPFLAGS += -UGCC_SOURCE
endif
```

Text Functions

```
SRC := foo.c bar.c xyzzy.c
OBJ := $(subst .c,.o,$(SRC))
OBJ := $(patsubst %.c,%.o,$(SRC))
OBJ := $(SRC:.c=.o)

ifeq ($(strip $(findstring -pg,$(CFLAGS))),-pg)
LDFLAGS += -pg
endif

SRC := foo.c bar.c main1.c main2.c xyzzy.c
MAINS := $(filter main%,$(SRC))
REST := $(filter-out main%,$(SRC))

SORTED := $(sort $(SRC))
```

Filename and Shell Functions

```
HDRS := foo/hdr1.h bar/hdr2.h hdr3.h
HDRDIRS := $(dir $(HDRS)) # foo/ bar/ ./

ifeq ($(notdir $(CC)),gcc)
CFLAGS += -pedantic
endif

clean:
    $(RM) *.o

OBJS := *.o # the literal *.o
OBJS := $(wildcard *.o) # as expected

SRC := $(shell /bin/ls *.c)
```

The `foreach` Function

```
SUBDIRS := foo bar xyzzy
COMMON_TARGETS := all opt prof debug clean

MAKEDIR = $(MAKE) -C $(d) $@;

$(COMMON_TARGETS):
    $(foreach d,$(SUBDIRS),$(MAKEDIR)) true

VPATH := $(foreach d,$(SUBDIRS),$(d):)../src

IN_VPATH = $(subst :, ,$(VPATH))

CPPFLAGS += $(foreach d,$(IN_VPATH),-I$(d))
```

Creating Your Own Functions

```
swap = $(2) $(1)
xyzzy = $(call swap,foo,bar)

IN_PATH = $(subst :, ,$(PATH))
allprog = $(addsuffix /$(1),$(IN_PATH))
which = $(firstword $(wildcard $(allprog)))
LS := $(call which,ls)

inc = -I$(1)
map = $(foreach a,$(2),$(call $(1),$(a)))
CPPFLAGS += $(call map,inc,$(IN_VPATH))
```

Recursive Make

```
SUBDIRS = foo bar baz
subdirs: $(SUBDIRS)
$(SUBDIRS):
        $(MAKE) -C $@
foo: baz
.PHONY: subdirs $(SUBDIRS)

DBGFLAGS := -g
OPTFLAGS := -O2
debug:
        $(MAKE) CFLAGS="$(DBGFLAGS)"
prod:
        $(MAKE) CFLAGS="$(OPTFLAGS)"
```

Dependency Generation

```
DEPCC := /usr/local/bin/gcc
DEPFLAGS := -MM -MG $(CPPFLAGS)
DEPSED := \
    sed 's/\\(.*\\)\\.o[ :]*/\\1.d &  /g'
DEPSCR = \
    "$(DEPCC) $(DEPFLAGS) $< | $(DEPSED) > $@"

%.d:    %.c
  @echo "Generating dependencies for $< ... "
  @$(SHELL) -ec $(DEPSCR)
  @test -s $@ || $(RM) $@

-include $(DEP)
```

Prototype Generation

```
PROTO_SUFFIX := _proto
CPROTO := cproto
TEMPLATE := -P"\nint\nf(\n\ta,\n\tb\n\t)"
CPROFLAGS := $(CPPFLAGS) -DCPROTO $(TEMPLATE)
OLDPROTO = $*$(PROTO_SUFFIX).h
NEWPROTO = $*$(PROTO_SUFFIX).new
%$(PROTO_SUFFIX).stmp: %.c
  @$(CPROTO) $(CPROFLAGS) $< > $(NEWPROTO); \
    diff -ubw $(OLDPROTO) $(NEWPROTO) \
      > /dev/null 2>&1 || \
    mv -f $(NEWPROTO) $(OLDPROTO); \
    $(RM) $(NEWPROTO)
  @touch $@
%$(PROTO_SUFFIX).h: %$(PROTO_SUFFIX).stmp
        @test -f $@ || $(RM) $<
```

Tags Generation

```
ETAGS := etags
ETAGS_FLAGS := --append --declarations \
               --globals --members

TAGS:
  @for f in $(SRC); do \
    for d in $(subst :, ,.:$(VPATH)); do \
      test -r $$d/$$f && \
      (echo "Adding tags for $$d/$$f to $@"; \
      $(ETAGS) -o $@ $(ETAGS_FLAGS) $$d/$$f) \
      || true; \
    done; \
  done
```

Make Software Engineering (I)

- makefiles are a part of your project, and all SE principles apply
- think of the user, make his/her life easier
- use abstractions as much as possible for flexibility
  - macroize as much as possible: you will be able to control things from the command line or environment, maintenance will be easier
  - it is not guaranteed that I will keep a directory structure identical to yours
    - if at all, use relative, not absolute paths
    - provide variables for important directories, they can be modified individually, even on a one-time basis

Make Software Engineering (II)

- let the user customize things without editing obscure makefiles
  - `CPPFLAGS += $(USR_CPPFLAGS)` will allow the user to recompile with
    `make USR_CPPFLAGS="-DDEBUG_LEVEL=8"`
- rely on facilities provided by default, such as

  `$(CC), $(CXX), $(CFLAGS), $(CPPFLAGS), $(LD) $(LDFLAGS), $(RM), $(AR), $(ARFLAGS)`

  etc.
- distinguish between `CFLAGS and CPPFLAGS`
- conform to conventional targets, such as `all, clean` etc.

Make Software Engineering (III)

- provide variables for common tools: I use cproto and gawk, you use something else
- you thought a lot about your directory structure, didn't you?
  - directories are likely to be logically distinct modules, hopefully reusable
  - provide a makefile for each
  - use recursive invocations to build subtrees
  - do not introduce makefile dependencies between things that are not compile-time dependent
  - provide a prominent top-level makefile that builds everything
- document your makefiles in README

Useful Options

- `-f` : specify a makefile.
- `-n` : dry run.
- `-t` : touch — mark the targets up to date.
- `-q` : question — is the target up to date?
- `-W` : what if this file is modified?
- `-o` : consider this file old.
- `-C` : change directory before running.
- `-e` : environment overrides.

More Useful Options

- `-r` : do not use built-in rules.
- `-R` : do not use built-in variables.
- `-p` : print rules database.
- `-d` : print debugging information.
- `-I` : where to look for included makefiles.
- `-k` : keep going.
- `-j` : run jobs in parallel.
- `-l` : specify maximal load.

GNU Make Features

- the “sane” variable flavour
- pass command-line variable assignments to recursive invocations of `make(1)`
- the `-C` (“change directory”) option
- `.PHONY`
- functions
- the `-o` (“old file”) option
- conditional execution
- search path for makefiles
- remaking makefiles
- expanded catalogue of rules

Further Details

- `info make`
- O'Reilly