Introduction to GNU Make

A quick introduction to GNU Make

Mitch Davis
mjd@afork.com

for SZLUG, 14-Aug-2011

What is make?

Make is a system which builds software. It knows:

- What to build
- How to build it

Simple rules + expert system = powerful behaviour

How to use GNU make?

- Instructions are in a file called `Makefile`
- Just run `make`

Why learn/use make?

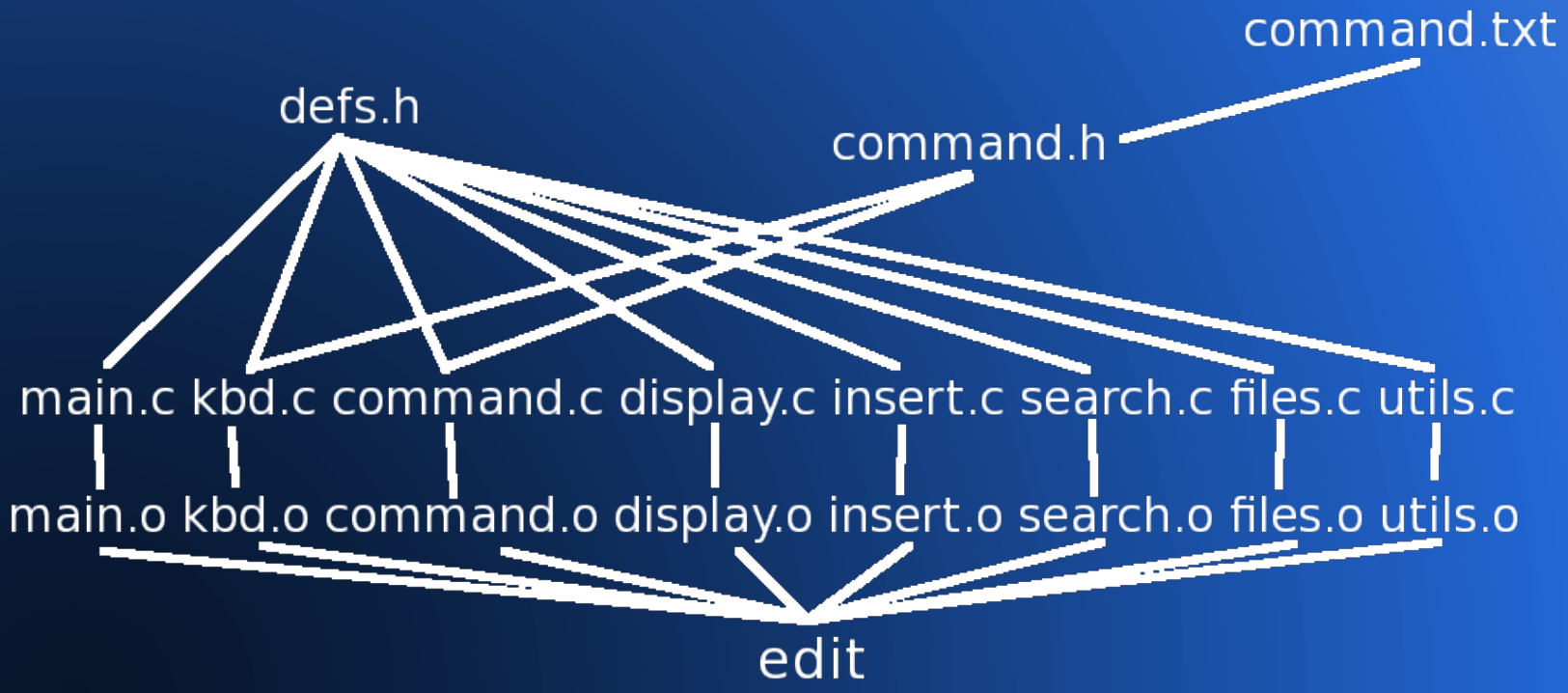
- Minimal rebuild ⇒ time savings
- Parallel builds ⇒ uses all your CPUs
- Understand / maintain other makefiles

Which version?

- Classic make
  Too old!
- cmake / scons
  Newer, better, but not widely used
- GNU make (gmake)
  Just right!

Linux: `make`
Other systems: `gmake` (may need to compile)

An example problem



Basic makefile blocks

```
target: prerequisites
   commands
```

`target: prerequisites` is the *rule*

`commands` is the *recipe*

Makefile rules

- Rules say *when* to rebuild something.
- If prerequisite changes, target must be rebuilt
- Make uses file timestamps
- Rebuilds if:
  - Prerequisite is more recent than target, or
  - target is missing

Makefile recipes

- Recipe says *how* to build something.
- Recipe lines *must* start with Tab character
- First rule is the default
- Common to have a default rule called `all`

A simple makefile

```
edit : main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
   cc -o edit main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o

main.o : main.c defs.h
   cc -c main.c
kbd.o : kbd.c defs.h command.h
   cc -c kbd.c
command.o : command.c defs.h command.h
   cc -c command.c
display.o : display.c defs.h buffer.h
   cc -c display.c
insert.o : insert.c defs.h buffer.h
   cc -c insert.c
search.o : search.c defs.h buffer.h
   cc -c search.c
files.o : files.c defs.h buffer.h command.h
       cc -c files.c
utils.o : utils.c defs.h
   cc -c utils.c
```

This makefile is very basic and can be improved a great deal!

Variables

- Simply expanded variables: `VAR:=`

  Evaluated when defined

  ```
  DATE=$(shell date); cc -DDATE="$(DATE)"
  ```

- Recursively expanded variables: `VAR=`

  Evaluated when used

  ```
  CFLAGS=-O2 -g $(OTHER)
  OTHER += -fpic
  ```

- Appending: `VAR+=`
- To use: `$(VAR)`
- Substitution: `SRCS=foo.c bar.c; $(SRCS:.c=.o)`

Variables example

Say `OBJECTS` variable holds list of object files:

```
objects = program.o foo.o utils.o
program: $(objects)
	cc -o program $(objects)

$(objects): defs.h
```

Pattern rules

We've seen *explicit* rules

```
insert.o : insert.c defs.h buffer.h
```

Pattern rules say how to build a class of files.

Have a `%` (the “stem”) as the common part.

Example:

```
%.o: %.c
```

⇒ No need to list recipe for every rule

Automatic variables

Automatic variables are useful in pattern rules

```
%.o: %.c
	cc -o $@ -c $<
```

Some automatic variables

- `$@` The name of the target
- `$<` The name of the first changed prerequisite
- `$^` The name of all the prerequisites

More about recipes

Recipes are one or more commands to run, in order to rebuild something.

Passed to shell.

Can use make variables: `mkdir -p $(BINDIR)`

“Gotchas”:

- Shell state (for example, environment vars) not shared between lines
- Use `$$` instead of `$`

A better makefile

```
OBJECTS = main.o kbd.o command.o display.o insert.o search.o files.o utils.o

all: edit

edit: $(OBJECTS)
   cc -o $@ $^

%.o: %.c
   cc -o $@ -c $<

$(OBJECTS): defs.h
kbd.o: command.h
command.o: command.h
display.o: buffer.h
insert.o: buffer.h
search.o: buffer.h
files.o: buffer.h command.h
```

Functions (1)

Make has many functions. Here are some useful ones:

- info/error
```
$(info CFLAGS=$(CFLAGS))
```
- patsubst
```
$(patsubst $(SRCDIR)/%.c,$(BINDIR)/%.o,$(SRCS))
```
- addprefix/addsuffix
```
CFLAGS += $(addprefix -L,$(INCDIRS))
```
- filter/sort
```
SRCS_c := $(sort $(filter %.c,$(SRCS)))
```

Functions (2)

More useful functions:

- dir / notdir / basename

```
RESOLV=/etc/resolv.conf
$(info $(dir $(RESOLV)) ⇒ /etc/
$(info $(notdir $(RESOLV)) ⇒ resolv.conf
$(info $(basename $(RESOLV)) ⇒ resolv
```

- shell
- Wildcard

```
SRCS_c := $(wildcard *.c)
```

```
foreach
  dirs := a b c d
  files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

What make can't do

Make's unit of computation is the file. So it doesn't work for non-file things:

- Non-local objects (such as firmware blobs on a remote website)

Make expects to run one recipe to rebuild one target:

- Has problems with Java because builds are done per-directory, not per-file.

Advanced make: Automatic C/C++ dependencies

Even with implicit rules, we still need to say the relation between `.o` and `.h` files.

A rule tells compiler to auto-generate rules for objects:

```
cc -M main.c
```

⇒ `main.o : main.c defs.h` ⇒ `.d` temporary file.

Change makefile to include `.d` files

Now no need to say what `.h` files a `.c` file includes!

Advanced make: parallel builds

Parallel builds: `-jn`

Suggestion: Use n = CPUs + 1

Computed variables

```
CONFIG_foo=y
 ⋮
OBJECTS_$(CONFIG_FOO) += foo.o
 ⋮
target: $(OBJECTS_y)
```

Very common in Linux kernel

Advanced make: Libraries and abstractions

- GNU make can include other files

```
PROGRAMS    = server client

server_OBJS = server.o server_priv.o server_access.o
server_LIBS = priv protocol
client_OBJS = client.o client_api.o client_mem.o
client_LIBS = protocol

include my_make_library
```

Advanced: Recursive make

For multi-directory projects, make can recursively traverse a directory tree:

```
for dir in lib webclient cmdline; do $(MAKE) -C $$dir; done
```

Advanced: Non-recursive make!

- Recursive make is very inefficient, must traverse whole tree, even if nothing to do.
- Possible to do it using includes, rather than submakes.
- As used by Linux kernel
- Recursive make considered harmful
  `http://miller.emu.id.au/pmiller/books/rmch/`

Advanced: GNU autotools

- `configure.ac` produces
- `configure.in` which produces
- `configure` which uses
- `Makefile.in` to produce
- `Makefile`

Advanced: Non-program targets

- Useful when you have one job to do on many files, such as compression.
- Make a target which is the list of processed files. `($wildcard)` function is useful.
- Run `make -j`*n*
- Have a coffee, come back when done.

More information

- Make info page: `info make`
- GNU Make Standard Library
  http://gmsl.sourceforge.net/
- Google!

Thank you

- Thanks for the committee for the opportunity
- Thanks for Atommann's translation
- Thanks for listening!