Advanced C Programming

gmake, gdb

Sebastian Hack
hack@cs.uni-sb.de

Christoph Weidenbach
weidenbach@mpi-inf.mpg.de

18.11.2008

SAARLAND UNIVERSITY
COMPUTER SCIENCE

make

- Automate and optimize construction of software
- Specify dependencies among files
- and give rules how to transform them
- Can be used for any kind of “compilation task”
  - Preparing LaTeX documents
  - Transforming images using . . .
  - and so on
- Several variants exist:
  - GNU Make (covered in this lecture)
  - Microsoft `nmake`
  - BSD make

GNU Make

- most powerful make variant
- available on almost every platform
- POSIX.2 compatible
- SysV make variant
- Attention: not entirely compatible to BSD make and `nmake`

What is Make?

An Example

- Suppose we have a small project containing:
  - Two translation units `kbd.c` `console.c`
  - Two header files `defs.h` `command.h` both included by both `.c` files
  - The resulting binary shall be called `edit`

`kbd.c`

```
#include "defs.h"
#include "command.h"
...
```

`console.c`

```
#include "defs.h"
#include "command.h"
...
```

- To build `edit`
  - we compile both `.c` files to `.o` files
  - link the `.o` files together
- When we develop (edit `.c` and `.h` files)
  - we need to rebuild the `.o` files affected by the changes
  - and finally the binary
- Writing the appropriate compiler invocations by hand all the time is cumbersome

What is Make?

- `defs.h command.h` and `console.c` are prerequisites for `console.o`
- `console.o` needs to be rebuilt when one of those are changed
- Rules describe dependencies and give commands how files are produced from others:

```
# target   prerequisites (dependencies)
console.o: console.c defs.h command.h
    # commands
    cc -c console.c
```

... means

If the modification time of one or more of

`console.c defs.h command.h`

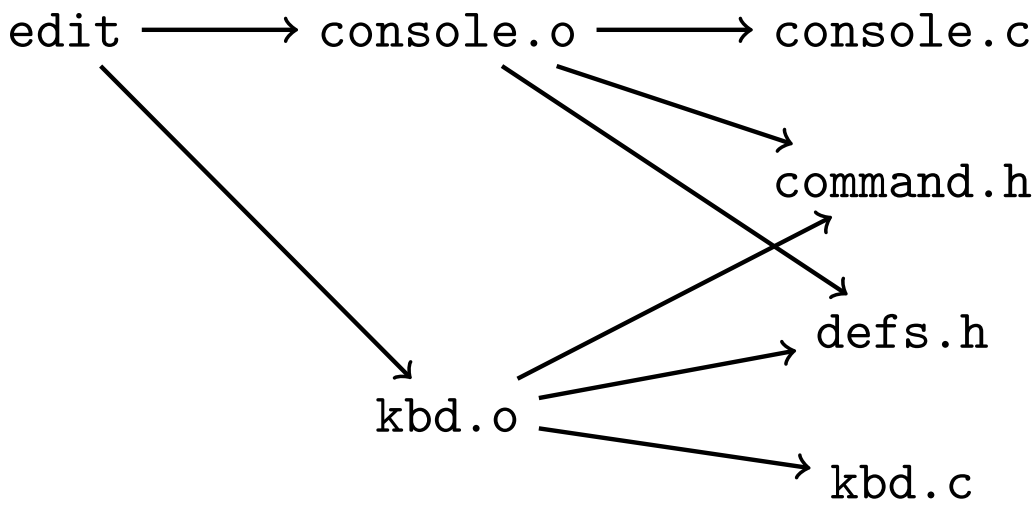
is newer than the one of `console.o`, execute

`cc -c console.c`

to update `console.o`

Dependencies

- According to the rules, Make constructs a dependency graph
- This graph needs to be acyclic (DAG)
- In our example:



- When processing the `Makefile` Make traverses the graph from leaves to root
- If the modification date of a child is newer than the node's, the node needs to be redone

Make

Basics

- Basic syntax

```
tgt1 tgt2 ... : preq1 preq2 ...
    cmd1
    cmd2
    ...
```

- Ingredients:
  - Targets: `tgt1`, `tgt2`, ...
  - Prerequisites: `preq1`, `preq2`, ...
  - Commands: `cmd1`, `cmd2`, ...
- `tgt1`, `tgt2`, ..., `preq1`, `preq2`, ... are files
- `tgt1`, `tgt2`, ... are dependent on `preq1`, `preq2`, ...
- Executing `cmd1`, `cmd2` produces `tgt2`, ..., from `preq1`, ...
- Attention:
  - commands must be preceded by a tab
  - Otherwise: `*** missing separator.  Stop.`

Variables

- For example, some C project:

```
edit : kbd.o console.o
    cc -o edit kbd.o console.o
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c
console.o : console.c defs.h command.h
    cc -c console.c
clean :
    rm -f kbd.o console.o edit
```

- Variables simplify your life:

```
objects = kbd.o console.o
edit : $(objects)
    cc -o $@ $(objects)
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c
console.o : console.c defs.h command.h
    cc -c console.c
clean :
    rm -f $(objects) edit
```

- `$@` name of target(s) in rule

Variables

- Variables are evaluated lazily
- If variable is never used, right side is not evaluated
  ☞ take care of side effects (use `:=`)
- What does this print?

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:
    echo $(foo)
```

- If you want expansion at definition point, use `:=`

```
ugh := Huh?
bar := $(ugh)
foo := $(bar)

all:
    echo $(foo)
```

- Add to a list with `+=`

```
files += a.c b.c
```

- Set variable only when not yet set: `?=`

Implicit Rules

- Life is even simpler:

```
objects = kbd.o console.o
edit : $(objects)
    cc -o edit $(objects)

kbd.o : defs.h command.h
console.o : defs.h command.h

clean :
   rm edit $(objects)
```

- Make has a database of implicit rules
- It knows how to make a `.o` file from a `.c` file:

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

- `$<` name of first prerequisite in rule
- `$(CC)` name of C compiler on the system
- `$(CPPFLAGS)` flags to give the C preprocessor
- `$(CFLAGS)` flags to give the C compiler

Implicit Rules

- You can (re-)define them yourself:

```
# Compile a LaTeX file
%.pdf : %.tex
    pdflatex $<

# Convert png to jpeg
%.jpg : %.png
    pngtopnm $< | pnmtojpeg > $@
```

- For C projects, you do not need to redefine implicit rules
- But you might want to set the variables `$(CFLAGS)`, ...
- Example:

```
CC       = icc         # use intel C compiler
CFLAGS   = -O3         # activate all optimizations
CPPFLAGS += -I/usr/local/include # add to include path
```

Automatically Computed Prerequisites

- Since GCC parses all the C files ...
- ... it can also compute the dependencies automatically
- Use switch `-M` instead of `-c` to emit Make rules from `.c` files
- For example:

```
/* kbd.c */
#include "defs.h"
#include "command.h"
/* ... */
```

and

```
shell$ gcc -M kbd.c
kbd.o: kbd.c defs.h command.h
```

Automatically Computed Prerequisites

Practice

- Define implicit rule to create a `.d` file from a `.c` file

```
%.d : %.c
    $(CC) -M $< > $@
```

- After first target, include all `.d` files
  (variables come in handy!)

```
ifeq ($(findstring $(MAKECMDGOALS), clean),)
-include $(objects:.o=.d)
endif
```

  - `$(a:x=y)` substitutes suffix `x` by `y` in every word in list `a`
  - `ifdef` avoids creating dependencies when only cleaning
  - `-` in front of command suppresses warnings
  - `include` creates dependency! ☞ causes `.d` files to be created

- Dependencies are updated automatically!
  Homework: Why?

Our example now

```
objects = kbd.o console.o
depends = $(objects:.o=.d)

.PHONY: clean

edit : $(objects)
    cc -o $@ $(objects)

ifeq ($(findstring $(MAKECMDGOALS), clean),)
-include $(depends)
endif

%.d : %.c
    $(CC) -M $< > $@

clean :
    rm -f $(objects) edit
```

- `clean` is no file!
- To avoid confusion with potentially existing files declare as phony

Make

Tips & Tricks

- It is not bad to put configuration settings to be provided by the user into a separate file

`Makefile`

```
...
include config.mak
...
# Adapt C flags for
# debug/optimized build
ifdef NDEBUG
CFLAGS += -O3 -DNDEBUG
else
CFLAGS += -O0 -g
endif

CFLAGS   += $(MY_CFLAGS)
CPPFLAGS += $(MY_CPPFLAGS)
edit : $(objects)
```

`config.mak`

```
NDEBUG      = 1
MY_CFLAGS   =
MY_CPPFLAGS = -I$(HOME)/include
```

- For all the details, see GNU Make manual

Make

Tips & Tricks 2

- Put generated files (`.o`, `.d`, final binary) into separate directory
- Requires more Make and compiler flag magic

```
builddir = build
sources  = kbd.c console.c
objects  = $(addprefix $(builddir)/,$(sources:.c=.o))
deps     = $(objects:.o=.d)

...

$(builddir)/%.o : %.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -o $@ -c $<

$(builddir)/%.d : %.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -MT $(@:.d=.o) -M $< > $@
```

☞ generated files do not pollute your source directory

Make

Tips & Tricks 3

- For nicer output, use Linux kernel style pretty printing

```
Q ?= @
...
$(builddir)/%.d : %.c
    @echo "===> DEPEND $@"
    $(Q)$(CC) $(CFLAGS) $(CPPFLAGS) ...
$(builddir)/%.o : %.c
    @echo "===> COMPILE $@"
    $(Q)$(CC) $(CFLAGS) $(CPPFLAGS) -c $<
...
```

- `@` at the beginning of the line does not print the command
- See full output with

```
shell$ make Q=
```

Make

Tips & Tricks 4 — General Remarks

1. Provide target `all` that build everything
   Make it the first (default) target
2. Use `make -j N` to build simultaneously on `N` CPUs
3. Never call Make recursively in subdirectories
   - Instead, use includes
   - Calling make recursive disrupts automated dependency tracking
   - Parallelization not possible!
4. The Quick Reference in the GNU Make Manual is very good!

GDB

- Compile program with debug support:
  - Debug symbols: `-g`
  - No optimizations: `-O0`
- Why?
- Debug symbols tell the debugger
  - Which objects are where (functions, global variables)
  - layout of stack frames
  - layout of structs
  - types, names, and so on
- Optimizations alter the program to strongly by
  - function inlining
  - loop unrolling
  - if-conversion
  - code re-ordered

  ☞ hard to establish relation between source and binary
- Using `-O0` everything remains as in the source

Breakpoints

- Tell the debugger when to stop the execution

```
(gdb) b myfunc
```

  stops execution each time `myfunc` is entered

- Can also give filename:lineno
- Can be dependent on condition

```
(gdb) b myfunc if x > 5
(gdb) b file.c:55 if node->id==4711
```

- Beware of side effects in expressions!

Watchpoints

- A breakpoint on data

```
(gdb) watch a
(gdb) watch *p
```

- gdb stops whenever watched expression changes
- Program execution might be slow ☞ conditions checked on each instruction
- Some architectures have hardware support for signalling changing memory contents
  ☞ debug registers

Commands

Controlling Execution

- `continue` run till next breakpoint
- `step` goes to next line of source code
  will enter functions
- `next` goes to next line of source code
  will step over functions
- use abbreviations: `cont`, `s`, and `n`

Inspecting the stack

- `backtrace` (`bt`) shows active stack frames
- `frame N` switches to given stack frame
- `info locals` gives values for local variables in current frame

Viewing Data

- Use `print` (`p`) to view value of expression
- Use `x` to inspect contents of memory
- Use `display` to show contents at each prompt

```
print somevar
x &somevar
x/t &somevar          # binary
display /x somevar    # hex format
```

- `/x` is a format
- Some Formats:
  - `x` hex
  - `t` binary
  - `f` float
  - `a` address
  - `s` string
  - ...

Macros

- GDB has a powerful macro language
- Define macros to be laoded at start in `.gdbinit`

Some examples:

1. Execute to a certain program location and show instruction at program counter

```
define g
tbreak $arg0
continue
x/1i $pc
echo ----------\n
end
```

2. Custom print routines

```
define vec
call printf("[%f, %f, %f, %f]\n",
    $arg0[0], $arg0[1], $arg[2], $arg[3])
end
```

How does it work?

- At breakpoints, gdb changes the machine code
- Inserts code that causes a trap
- On x86, there is a special instruction called `int3`
- You can use that yourself
- Suppose you have some events where it is too cumbersome to specify breakpoints, call

```
int do_breakpoints = 0; /* e.g. set by command line */

#if defined(__GNUC__)
      && (defined(__i386__) || defined(__x86_64))
extern void enter_debugger(void) {
    if (do_breakpoints)
        __asm__ __volatile__("int3");
}
#else
extern void enter_debugger(void) { }
#endif
```