Makefiles explained

Parallel Programming, SoSe 2009

SoSe 2009

30. April 2009

Agenda

- recap: compiling and linking multiple source files
- using make
- writing a small Makefile
- variables
- dependencies
- parallel builds

Recapitulation: Compiling and linking

- a program is usually split up in several '.c-files'
- these are compiled to object-files
- object-files are linked together to form an executable
- changes in one source-file may affect several object files

The simple approach

- the simplest way is calling gcc like:

  ```
  gcc *.c -o myprogram
  ```
- it may work for very simple projects
- this approach does not scale well
- you cannot exclude specific files from being linked

The magic of make

- let's say you have a file 'foo.c'
- type: `make foo`
- Make will compile it into a binary 'foo'
- great! But `gcc foo.c -o foo` is easy.

More source files

foo.c

```
#include <stdio.h>
#include "bar.h"

int main(int argc,char**argv){
        hello();
        return 0;
}
```

bar.h

```
void hello();
```

bar.c

```
#include <stdio.h>

void hello(){
    printf("Hello world.\n");
}
```

Defining dependencies

- the benefit of Make comes in defining dependencies
- several .c-files generating several .o-files
- a first simple 'Makefile' to compile foo.c and bar.c into 'program':

```
program: foo.o bar.o
        $(CC) foo.o bar.o -o program
```

The output of 'make' reads:

```
$ make
cc    -c -o foo.o foo.c
cc    -c -o bar.o bar.c
cc foo.o bar.o -o program
```

But how does make know what to do?

Defining dependencies

- `program: foo.o bar.o` defines a dependency
- one of the c-files changes $\Rightarrow$ make recompiles and relinks
- but we still have to write a lot manually :(
- foo.o, bar.o and even program are written out twice :(
- let's do a better Makefile

Defining dependencies

```

# my source and object files:
SOURCES := foo.c bar.c
OBJS := $(patsubst %.c, %.o, $(SOURCES))

# default compiler-flags:
CFLAGS=-g -Wall

# mark targets as "virtual"
.PHONY: all

# all 'calls' program
all:    program

# define the target 'program'
program: $(OBJS)
        $(CC) $(CFLAGS) $(OBJS) -o $@

# make a clean working dir
.PHONY: clean
clean:
        rm -f $(OBJS) program
```

Some notes

- a line like 'program:' defines a target
- a target usually generates a file with the target's name
- other targets need to be defined as 'PHONY',
  to prevent conflicts with files
- if there is no rule for the a target to create .o files, Make has an implicit rule
- all shell-commands need to be indented using a TAB (not spaces)

Linking against system libs

```makefile
SOURCES := foo.c bar.c
OBJS := $(patsubst %.c, %.o, $(SOURCES))

#generate a list of libs
LDFLAGS  = -lm -lpthread  -lgthread
LDFLAGS += `gtk-config --cflags` `gtk-config --libs`

CFLAGS=-g -Wall

.PHONY: all

all:    program

# define the target 'program'
program: $(OBJS)
        $(CC) $(CFLAGS) -o $@ $(OBJS) $(LDFLAGS)

clean:
        rm -f $(OBJS) program
```

Some notes on variables

- GNU-make knows two types of Variables
- recursively expanded variables (=):
- evaluated on each occurence
- can include references to other variables

- simply expanded variables (:=):
- evaluated once when defined

see: `http://www.gnu.org/software/automake/manual/make/Flavors.html`

More complex dependencies

- what if there are more #include-statements?
- what about #ifdef-protected #includes?
- do I have to specify them all?
- NO! - use makedepend:

The makedepend program reads each sourcefile in sequence and parses it like a C-preprocessor, processing all #include, #define, #undef, #ifdef, #ifndef, #endif, #if, #elif and #else directives so that it can correctly tell which #include, directives would be used in a compilation. Any #include, directives can reference files having other #include directives, and parsing will occur in these files as well.

Linking against system libs

```
SOURCES := foo.c bar.c
OBJS := $(patsubst %.c, %.o, $(SOURCES))

#generate a list of libs
LDFLAGS  = -lm -lpthread  -lgthread
LDFLAGS += `gtk-config --cflags` `gtk-config --libs`

CFLAGS=-g -Wall

.PHONY: all

all:     program

program: $(OBJS)
         $(CC) $(CFLAGS) -o $@ $(OBJS) $(LDFLAGS)

#calculate dependencies:
depend:
         makedepend -- $(CFLAGS) -- $(SOURCES)

clean:
         rm -f $(OBJS) program
```

predefined variables

make knows some Variables:

```
# common variables like

CFLAGS= ...
LDFLAGS= ...

program: $(OBJS)
        $(CC) $^ -o $@

# $@ - the name of the target
# $^ - the names of all prerequisites
# $< - The name of the first prerequisite

# RM - the rm-command
# CC - C-Compiler
# ...
```

see `info make` ;)

parallel make

- GNU-make can be used to do parallel builds
- all dependencies must be properly defined
- use `make -j <num>` to execute make
- num specifies the number of parallel processes

more info

a german tutorial:
http://www.ijon.de/comp/tutorials/makefile.html
documentation:
http://www.gnu.org/software/automake/manual/make/index.html#Top