reSilient coMputer archItectures and LIfE Sciences

Politecnico di Torino

Department of Control and Computer Engineering

MAKEFILES

STEFANO DI CARLO

SEPARATE COMPILATION

- Large programs are generally separated into multiple files, e.g., `main.c addmoney.c removemoney.c money.h`
- With several files, we can compile and link our program as usual using

```
gcc addmoney.c removemoney.c main.c
```

- When compiling in this manner produces a problem, we fix the problem and recompile.
- But, we ended up recompiling everything with
  - `gcc addmoney.c removemoney.c main.c`
  - even if we had to make a very simple change to just one file.
- This is wasteful.

SEPARATE COMPILATION

- What we should do instead is separately compile source files to intermediate object files and then link them together
- So, for the files
  - `addmoney.c`
  - `removemoney.c`
  - `main.c`

  We want to compile each piece separately and then link them together.
- When we just compile source code (without linking it together), it means that we take the .c files and generate intermediate object (.o) files.

SEPARATE COMPILATION

- To just compile source code, use the -c flag with the compiler...

```
gcc -c addmoney.c
gcc -c removemoney.c
gcc -c main.c
```

- This will generate the object files `addmoney.o, removemoney.o`, and `main.o`
- Finally, to link the object files (.o) into an executable that we can run, we use the compiler again (although this time it will just pass the .o files on to the linking stage):

```
gcc -o money addmoney.o removemoney.o main.o
```

MAKE UTILITY

- The Unix make program is a handy utility that can be used to build things ranging from programs to documents.
- Helps you to build and manage projects
- Types of statements that can go in a makefile
  - **macro definition** – name that you define to represent a variable that may occur several times within the makefile
  - **target definition** – lists the target file, its required files, and commands to execute the required files in order to produce the target.
  - **Suffix rules** – indicate the relationship between target and source file suffixes (filename extensions).
  - **Suffix declarations** – lists of suffixes (file extensions) used in suffix rules

MAKE UTILITY

- target definition
  - `target: dependencies`
  - `[tab] commands`
- **targets** – labels that appear in column 1 and are followed by the ":" character.
- **dependencies** - a list of files following the name of the target. These are the files that are needed to make the target. The target "depends on these files." If any dependency is newer than the target, the target will be rebuilt.
- **commands** – specify the procedure for building the target. Each line must begin with the tab character, not spaces.

MAKE UTILITY

- For a project consisting of the files `main.c` , `removemoney.c` , `addmoney.c` and `money.h`, the trivial way to compile the files and obtain an executable is

```
gcc -o money main.c removemoney.c addmoney.c
```

- A makefile for doing this would look like:

```
money: main.o removemoney.o addmoney.o
        cc -o money main.c removemoney.c addmoney.c
```

- In this example,
  - target is money.
  - The dependencies are main.o, removemoney.o, and addmoney.o
- For make to execute correctly, it must meet all the dependencies of money. If `main.o`, `removemoney.o`, or `addmoney.o` is newer than money, then make rebuilds money

MAKE UTILITY - RUNNING MAKE ON THE COMMAND LINE

- There are different ways to run make.

```
make
```

  - Looks in the current directory for a file named makefile or Makefile and runs the commands for the first target

```
make -f <filename>
```

  - Looks in the current directory for a makefile with the given name and runs the commands of the first target.

```
make <target>
```

  - Looks for a file named makefile or Makefile and locates the target. This does not have to be the first target. It will run the commands for that target provided the dependencies are more recent than the target.

MAKE UTILITY

- Example

```
money: main.o removemoney.o addmoney.o
        cc -o money main.c addmoney.c removemoney.c
```

- To build money, type either of the following commands:

```
make
```

  - Or

```
make money
```

MAKE UTILITY

- Example

```
money: main.o addmoney.o removemoney.o
        cc -o money main.c addmoney.c removemoney.c
clean:
         rm *.o *.err
```

- In this example, there are two targets: money and clean
- The second target has no dependencies.
- The command make clean will remove all object files and all .err files.

MACROS

- You want to use macros to make it easy to make changes.
  - For example, if you use macros, it's easy to change the compiler and compiler options different compilers. It's easy to turn on and off debug options.
  - Without macros, you would use a lot of search and replace.
- You use macros in makefiles for the same reason you define constants in programs. It's easier to update the files and make it more flexible.

MACROS

- Predefined macro-based names:
  - `$@` — the current target’s full name
  - `$?` — a list of the target’s changed dependencies
  - `$<` — similar to $? But identifies a single file dependency and is used only in suffix rules
  - `$*` — the target file’s name without a suffix
- Another useful macro-based facility permits one to change prefixes on the fly. The macro
  - `$(@:.o=.err)` says use the target name but change the .o to .err.

MACRO DEFINITION

- A makefile line with the following syntax

```
MACRO-NAME = macro value.
```

- Invoked using the syntax

```
$(MACRO-NAME)
```

- Result is that $(MACRO-NAME) is replaced by the current value of the macro.

- Examples

```
OBJS = main.o removemoney.o addmoney.o
CC = gcc
CFLAGS = -DDBG_PIX -DDBG_HIT
```

VARIABLES

- The old way (no variables)

```
my_prog : eval.o main.o
    gcc -o my_prog eval.o main.o
 eval.o : eval.c eval.h
    gcc -c -g eval.c
main.o : main.c eval.h
    gcc -c -g main.c
```

Defining variables on the command line:

Take precedence over variables defined in the makefile.

```
make C=cc
```

- A new way (using variables)

```
C = gcc
OBJS = eval.o main.o
HDRS = eval.h

my_prog : eval.o main.o
    $(C) -o my_prog $(OBJS)
eval.o : eval.c
    $(C) -c -g eval.c
main.o : main.c
    $(C) -c -g main.c
$(OBJS) : $(HDRS)
```

MAKE OPTIONS

- **make** options:
  - **-f** filename - when the makefile name is not standard
  - **-t** - (touch) mark the targets as up to date
  - **-q** - (question) are the targets up to date, exits with 0 if true
  - **-n** - print the commands to execute but do not execute them
  - **/ -t, -q**, and **-n**, cannot be used together **/**
  - **-s** - silent mode
  - **-k** - keep going – compile all the prerequisites even if not able to link them !!

VPATH

- VPATH variable – defines directories to be searched if a file is not found in the current directory.
  - VPATH = dir : dir …
  - / VPATH = src:../headers /

  - vpath directive (lower case!) – more selective directory search:
  - vpath pattern directory
  - / vpath %.h headers /

  - GPATH:
  - GPATH – if you want targets to be stored in the same directory as their dependencies.

IMPLICIT RULES

- Implicit rules are standard ways for making one type of file from another type.
- There are numerous rules for making an .o file – from a .c file, a .p file, etc. make applies the first rule it meets.
- If you have not defined a rule for a given object file, make will apply an implicit rule for it.
- Example:

Our makefile

```
my_prog : eval.o main.o
    $(C) -o my_prog $(OBJS)
$(OBJS) : $(HEADERS)
```

The way make understands it

```
my_prog : eval.o main.o
    $(C) -o my_prog $(OBJS)
$(OBJS) : $(HEADERS)
eval.o : eval.c
    $(C) -c eval.c
main.o : main.c
    $(C) -c main.c
```

EXAMPLE OF A MAKEFILE

```
CC = gcc
DIR = /home/faculty/crahn/public_html/cop4833/lib
CFLAGS = -g -I$(DIR) -I. -c
LFLAGS = -g
opt: analysis.o flow.o io.o misc.o opt.o opts.o peephole.o regs.o vect.o
	$(CC) $(LFLAGS) -o opt analysis.o flow.o io.o misc.o opt.o opts.o peephole.o 
regs.o vect.o
analysis.o: analysis.c analysis.h $(DIR)/misc.h $(DIR)/opt.h $(DIR)/vect.h
	$(CC) $(CFLAGS) analysis.c
flow.o: $(DIR)/flow.c $(DIR)/flow.h $(DIR)/opt.h
	$(CC) $(CFLAGS) $(DIR)/flow.c
io.o: $(DIR)/io.c $(DIR)/io.h analysis.h $(DIR)/misc.h $(DIR)/opt.h peephole.h 
$(DIR)/regs.h
	$(CC) $(CFLAGS) $(DIR)/io.c
```

READINGS

- These slides were created by copying (sometimes verbatim!) material from the manual http://www.gnu.org/software/make/manual/make.html .

- Read this manual for more information (just reading Chapter 2 will suffice).

reSilient coMputer archItectures and LIfE Sciences

Politecnico di Torino

Department of Control and Computer Engineering

THANK YOU!