ECE20019 Open Source Software, Spring 2017

2017-05-12

GNU make build system

The following slides are from

- https://www.cs.tau.ac.il/~roded/courses/softp-b06/Makefile.ppt
- https://www.cse.unr.edu/~bebis/CS308/PowerPoint/Makefiles.ppt

The Makefile Utility

ABC – Chapter 11, 483-489

Motivation

- Small programs ⟶ single file
- "Not so small" programs :
  - Many lines of code
  - Multiple components
  - More than one programmer

Motivation – continued

- Problems:
  - Long files are harder to manage (for both programmers and machines)
  - Every change requires long compilation
  - Many programmers cannot modify the same file simultaneously

Motivation – continued

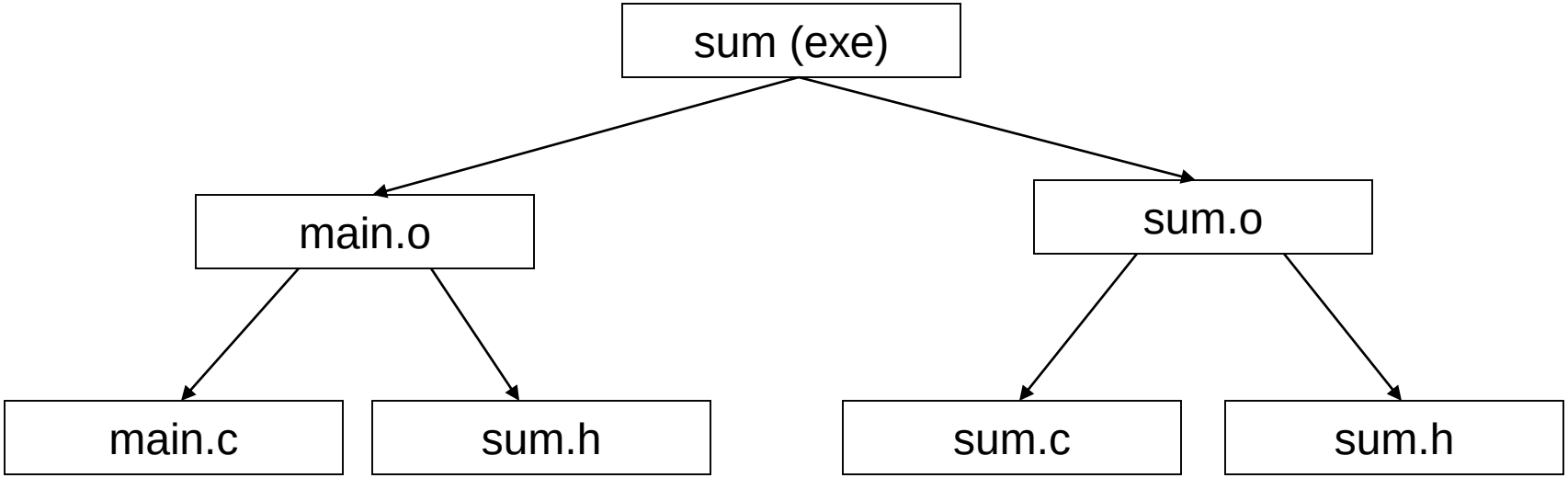
- Solution : divide project to multiple files
- Targets:
  - Good division to components
  - Minimum compilation when something is changed
  - Easy maintenance of project structure, dependencies and creation

Project maintenance

- Done in Unix by the Makefile mechanism
- A makefile is a file (script) containing :
  - Project structure (files, dependencies)
  - Instructions for files creation
- The make command reads a makefile, understands the project structure and makes up the executable
- Note that the Makefile mechanism is not limited to C programs

Project structure

- Project structure and dependencies can be represented as a DAG (= Directed Acyclic Graph)
- Example :
  - Program contains 3 files
  - main.c., sum.c, sum.h
  - sum.h included in both .c files
  - Executable should be the file sum


sum (exe)
main.o
sum.o
main.c
sum.h
sum.c
sum.h

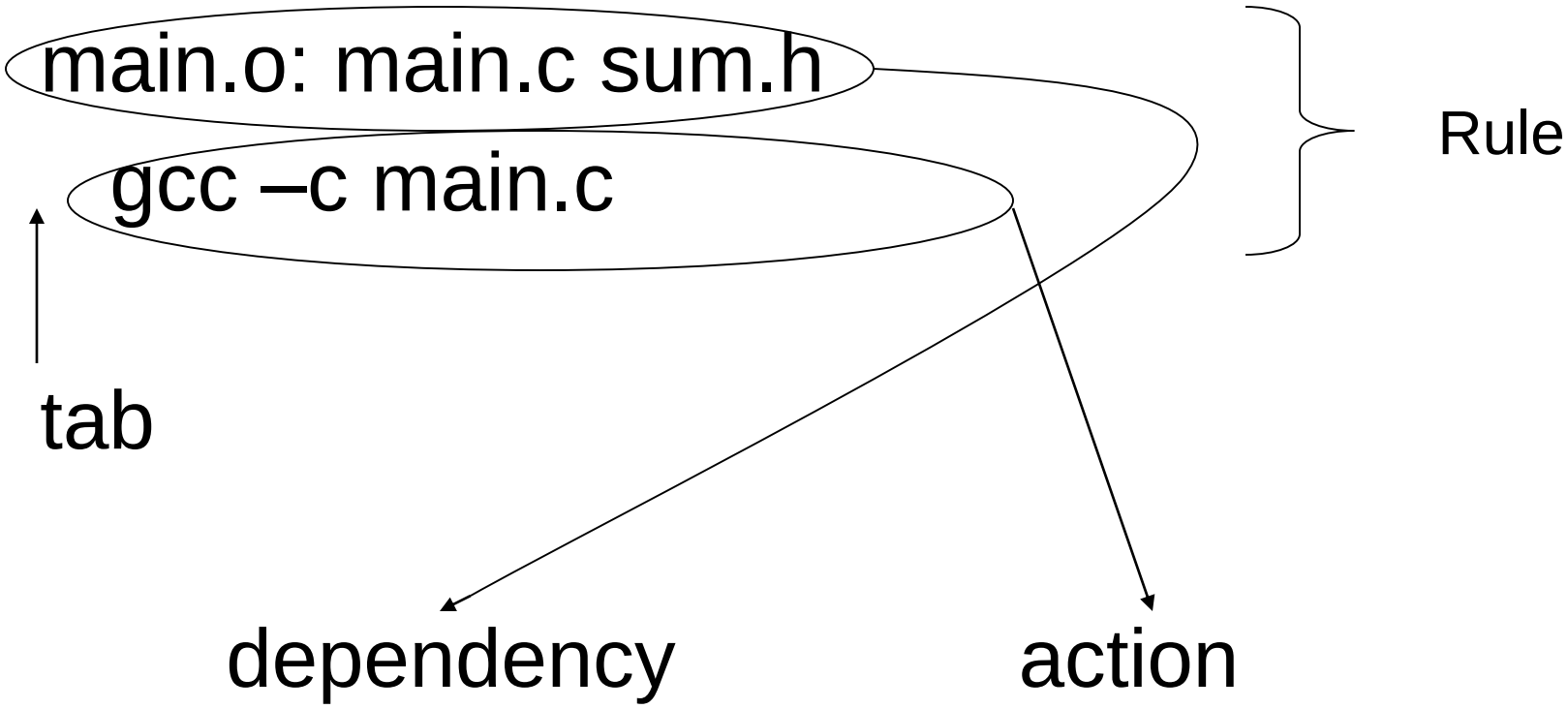
makefile

```
sum: main.o sum.o
    gcc –o sum main.o sum.o

main.o: main.c sum.h
    gcc –c main.c

sum.o: sum.c sum.h
    gcc –c sum.c
```

Rule syntax



Equivalent makefiles

- .o depends (by default) on corresponding .c file. Therefore, equivalent makefile is:

```
sum: main.o sum.o
    gcc –o sum main.o sum.o

main.o: sum.h
    gcc –c main.c

sum.o: sum.h
    gcc –c sum.c
```

make operation

- Project dependencies tree is constructed
- Target of first rule should be created
- We go down the tree to see if there is a target that should be recreated. This is required when the target file is older than one of its dependencies
- In this case we recreate the target file according to the action specified, on our way up the tree. Consequently, more files may need to be recreated
- If something was changed, linking is performed

make operation - continued

- make operation ensures minimum compilation, when the project structure is written properly
- Do not write something like:

  prog: main.c sum1.c sum2.c

  gcc –o prog main.c sum1.c sum2.c

  which requires compilation of all project when something is changed

Make operation - example

File	Last Modified
sum.h	08:39
sum.c	09:14
sum.o	09:35
main.o	09:56
sum	10:03
main.c	10:45

Make operation - example

- Operations performed:

  gcc –c main.c

  gcc –o sum main.o sum.o

- main.o should be recompiled (main.c is newer).
- Consequently, main.o is newer than sum and therefore sum should be recreated (by re-linking).

Useful gcc Options

- Include: -I<path>
- Define: -D<identifier>
- Optimization: -O<level>

Example:

```
gcc –DDEBUG –O2 –I/usr/include
example.c –o example -lm
```

Another makefile example

```
# Makefile to compare sorting routines

BASE = /home/blufox/base
CC          =   gcc
CFLAGS  =   -O –Wall
EFILE      =   $(BASE)/bin/compare_sorts
INCLS     =   -I$(LOC)/include
LIBS        =   $(LOC)/lib/g_lib.a \
                    $(LOC)/lib/h_lib.a
LOC        =   /usr/local

OBJS = main.o    another_qsort.o    chk_order.o \
           compare.o    quicksort.o

$(EFILE): $(OBJS)
     @echo “linking …”
     @$(CC) $(CFLAGS) –o $@ $(OBJS) $(LIBS)

$(OBJS): compare_sorts.h
     $(CC) $(CFLAGS) $(INCLS) –c $*.c

# Clean intermediate files
clean:
     rm *~ $(OBJS)
```

Example - continued

- We can define multiple targets in a makefile
- Target clean – has an empty set of dependencies. Used to clean intermediate files.
- make
  - Will execute the first target
- make clean
  - Will remove intermediate files

Phony targets

Phony targets:

Targets that have no dependencies. Used only as names for commands that you want to execute.

```
clean :
    rm $(OBJS)
```

or

```
.PHONY : clean
clean:
    rm $(OBJS)
```

To invoke it: `make clean`

Typical phony targets:

`all` – make all the top level targets

```
.PHONY : all
all: my_prog1 my_prog2
```

`clean` – delete all files that are normally created by `make`

`print` – print listing of the source files that have changed

Automatic variables

Automatic variables are used to refer to specific part of rule components.

```
target : dependencies
TAB   commands              #shell commands
```

```
eval.o : eval.c eval.h
    g++ -c eval.c
```

`$@` - The name of the target of the rule (`eval.o`).

`$<` - The name of the first dependency (`eval.c`).

`$^` - The names of all the dependencies (`eval.c eval.h`).

`$?` - The names of all dependencies that are newer than the target

Reference

- Good tutorial for makefiles

  http://www.gnu.org/software/make/manual/make.html