CS242: System Software Lab

Makefile

Tut02 : 5th Aug 2024

Dr. A. Sahu

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

- Makefile
  - Example
  - Rules
  - Dependency

Makefile

Introduction

- "make" utility in Unix
  - Is one of the original tools designed by S. I. Fieldman of AT&T Bell labs 1977.
- What is "make"?
  - The tool is designed to allow programmers to
  - efficiently compile large complex programs with many components easily.
- You can place the commands to compile a program in a Unix script
  - but this will cause ALL modules to be compiled every time.
- The "make" utility allows us to only compile those
  - that have changed and the modules that depend upon them.

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 can not modify the same file simultaneously
  - Division to components is desired

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
- Makefile mechanism is
  - not limited to C or Fortran programs

How Does it Work?

- When you type the command “make” the OS looks for a file called either “makefile” or “Makefile”.
- This file contains a series of directives that tell the “make” utility
  - How to compile your program and in what order.
- Each file will be associated with a list of other files by which it is dependent.
  - This is called a dependency line.
- If any of the associated files have been recently modified,
  - The make utility will execute a directive command just below the dependency line.

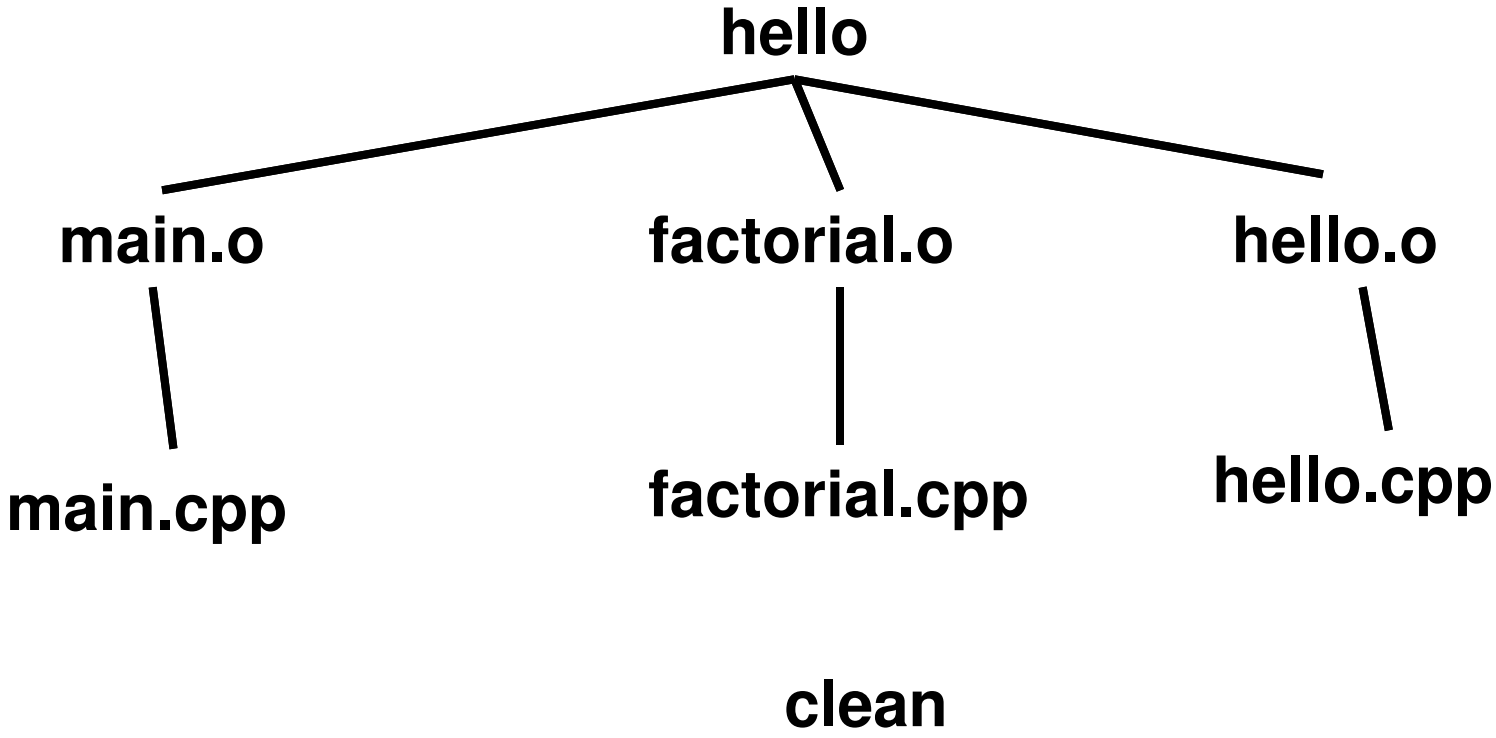
How Does it Work?

- The “make” utility is recursive.
  - For instance, if a very low level utility is the only thing changed,
  - it could cause all of the modules within a program to be re-compiled.
- After the utility finishes the file,
  - it goes through and checks all of the dependencies again to make sure all are up to date.

Simple Example

```
hello: main.o factorial.o hello.o
	g++ main.o factorial.o hello.o -o hello
main.o: main.cpp
	g++ -c main.cpp
factorial.o: factorial.cpp
	g++ -c factorial.cpp
hello.o: hello.cpp
	g++ -c hello.cpp
clean:
	rm -rf *o hello
```

Dependency Tree for the makefile



Components of a Makefile

- Comments
- Rules
- Dependency Lines
- Shell Lines
- Macros
- Inference Rules

Comments

- A comment is indicated by the character “#”.
  - All text that appears after it will be ignored by the make utility until the end of line is detected.
- Comments can start anywhere.
- There are more complex comment styles that involve continuation characters but please start each new comment line with an #
- Example
  - #
  - # This is a comment
  - projecte.exe : main.obj io.obj # this is also a comment.

Rules

- Tell make when and how to make a file.
- The format is as follows:
  - A rule must have a dependency line and may have an action or shell line after it. The action line is executed if the dependency line is out of date.
  - Example: `hello.o: hello.cpp`

    `g++ -c hello.cpp`
  - This shows hello.o as a module that requires hello.cpp as source code. If the last modified date of hello.cpp is newer than hello.o, than the next line (shell line) is executed.
  - Together, these two lines form a rule.

Dependency Lines

- The lines with a ":" are called dependency lines.
  - To the left are the dependencies
  - To the right are the sources needed to make the dependency.
- At the running of the make utility, the time and date when Project.exe was last built are compared to the dates when main.obj and io.obj were built.
- If either main.obj or io.obj have new dates, then the shell line after the dependency line is executed.

Dependency Lines

- The make process is recursive in that
  - It check all dependencies to make sure they are not out of date before completing the build process.
- **It is important that all dependencies be placed in a descending order in the file**.
- Some files may have the same dependencies. For instance, suppose that two files needed a file called bitvect.h.
- What would the dependency look like:

main.obj this.obj: bitvect.h

Shell Lines

- The indented lines (**must have tab**) that follow each dependency line are called shell lines. Shell lines tell make how to build the target.
- A target can have more than one shell line. Each line must be preceded by a tab.
- After each shell is executed, make checks to see if it was completed without error.
- You can ignore this but I would not at this point.

Shell Lines

- After each shell line is executed, Make checks the shell line **exit status**.
- Shell lines that returning an exit status of zero (0) means without error and non-zero if there is an error.
- The first shell line that returns an exit status of non-zero will cause the make utility to stop and display an error.
- You can override this by placing a "-" in front of the shell command, but I would not do this.
  - Example:
    - **gcc –o my my.o mylib.o**

Macros

- Comes from the Greek word makros meaning large.
- Basically it is a shorthand or alias used in the makefile
- A string is associated with another usually larger string
- Inside the file, to expand a macro, you have to place the string inside of $( ) .
- The whole thing is expanded during execution of the make utility.

Macros

Examples of macros:

- HOME = /project/projectdirs/astro250/nugent
- CPP = $(HOME)/cpp
- TCPP = $(HOME)/tcpp
- PROJ = .
- INCL = -I $(PROJ) –I$(CPP) –I$(TCPP)
- You can also define macros at the command line such as
  - make DIR = /myhomedir/
  - And this would take precedence over the one in the file.

Inference Rules

- Inference rules are a method of generalizing the build process. In essence, it is a sort of wild card notation.
- The “%” is used to indicate a wild card.
- Examples:

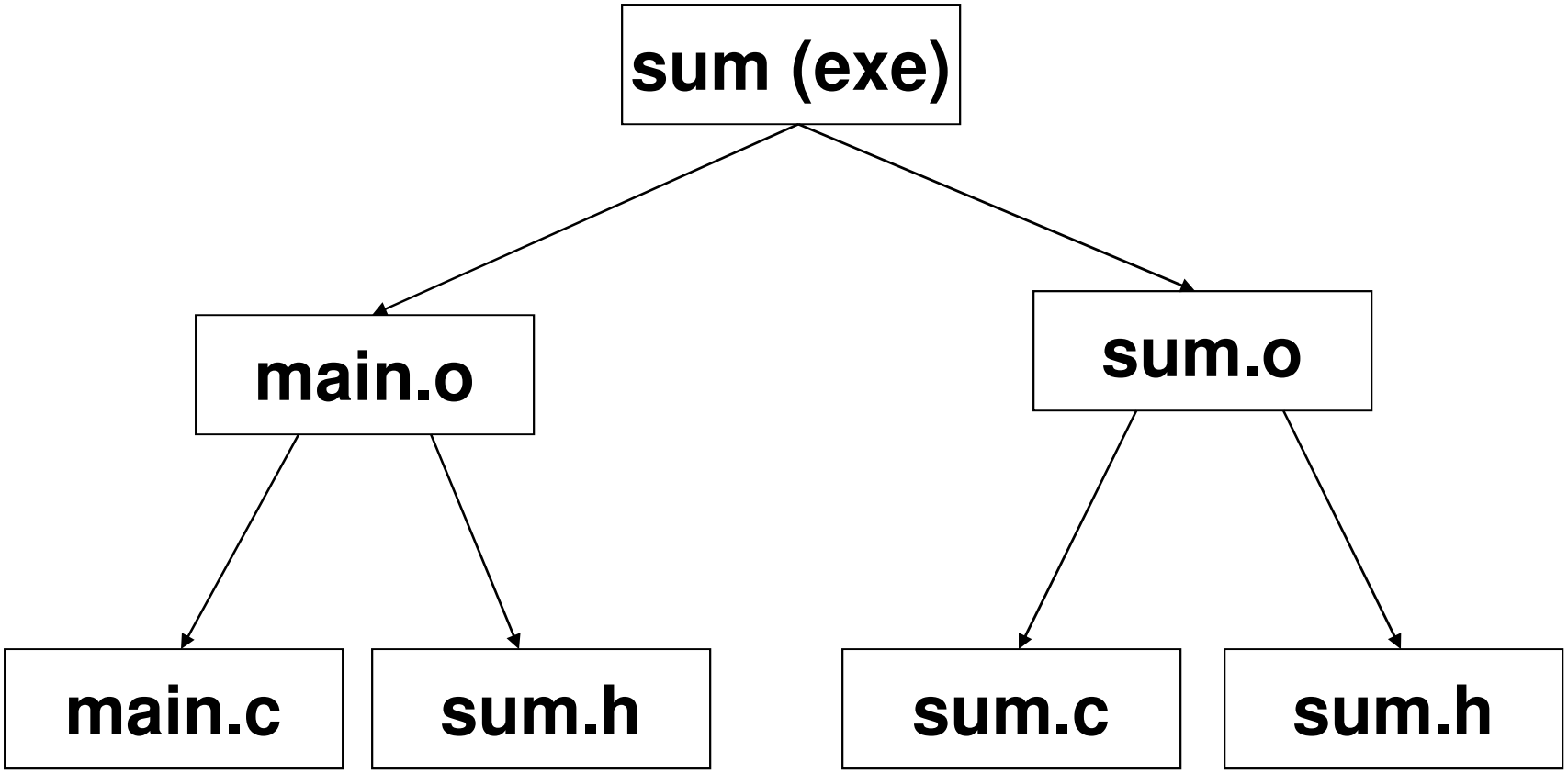
```
%.obj : %.c
        $(CC) $(FLAGS) –c $(.SOURCE)
```

- All .obj files have dependencies of all %.c files of the same name.

Built in / Special Macros

- **$@** The file name of the target.
- **$<** The name of the first dependency.
- **$*** The part of a filename which matched a suffix rule.
- **$?** The names of all the dependencies newer than the target separated by spaces.
- **$^** The names of all the dependencies separated by spaces, but with duplicate names removed.
- **$+** The names of all the dependencies separated by spaces with duplicate names included and in the same order as in the rule.

Another Example



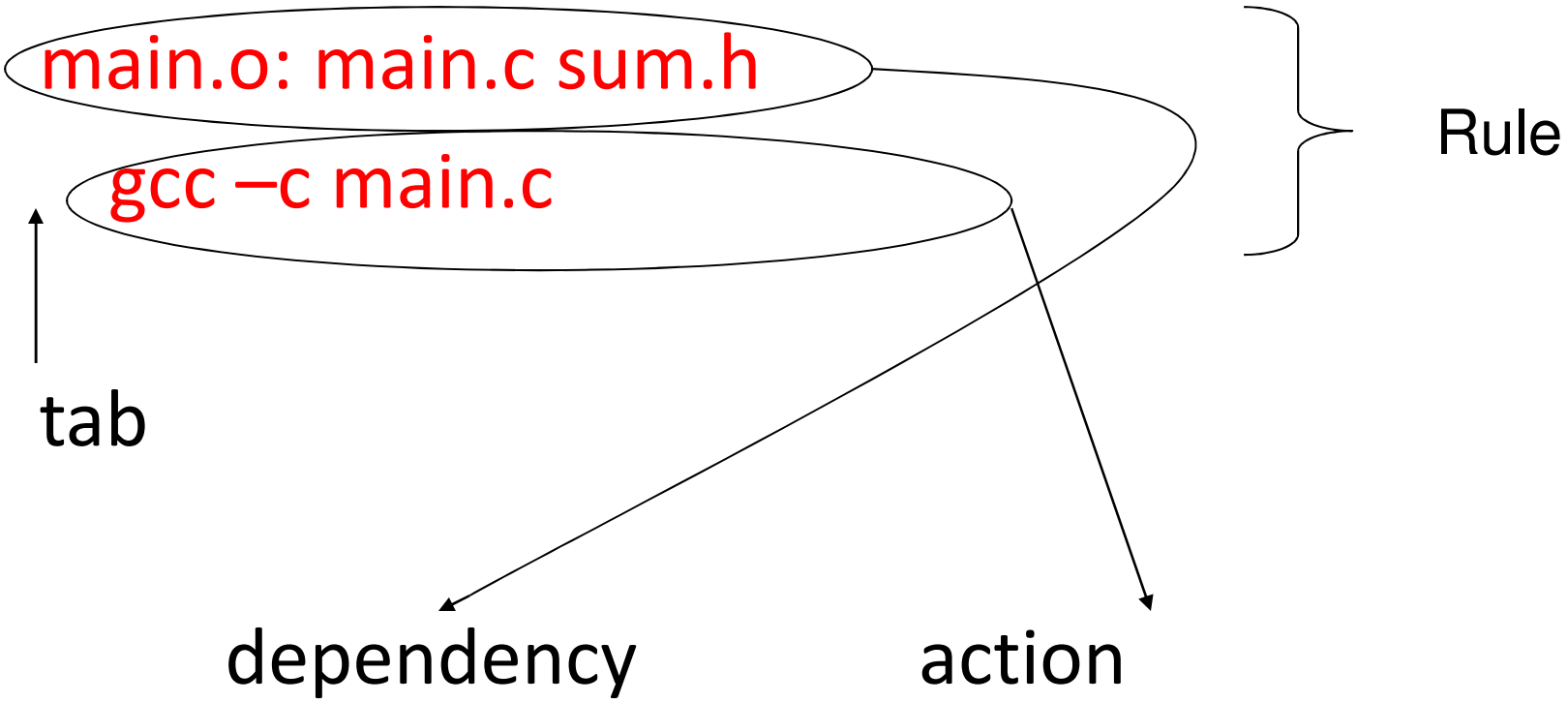
Example: makefile

```
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
```

Equivalent makefiles - continued

- We can compress identical dependencies and use built-in macros to get another (shorter) equivalent makefile :

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

main.o sum.o: sum.h
    gcc –c $*.c
```

$@ The file name of the target
gcc –o sum main.o sum.o

$* The part of a filename which matched a suffix rule.
gcc –c main.c
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 the case 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 is changed, linking is usually necessary

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

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).

Another Makefile Example

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

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)
```

Multiple Target

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

Passing parameters

- Note that assigning a value to a variable within the makefile overrides any value passed from the command line.
- For example:

  command line : make PAR=1

  in the makefile:

  PAR = 2
- PAR value within the makefile will be 2, overriding the value sent from the command line

Conditional statements

- Simple conditional statements can be included in a makefile.
- Usual syntax is:

```
ifeq (value1, value2)
        body of if
else
        body of else
endif
```

Conditional statements - example

```
sum: main.o sum.o
    gcc –o sum main.o sum.o
main.o: main.c sum.h
    gcc –c main.c
#deciding which file to compile to create sum.o
ifeq ($(USE_SUM), 1)
sum.o: sum1.c sum.h
    gcc –c sum1.c –o $@
else
sum.o: sum2.c sum.h
    gcc –c sum2.c –o $@
endif
```

Reference

- Good tutorial for makefiles

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

Thanks