Makefiles

CS 2XA3

Term I, 2018/19

Outline

Example

Assume we have files **main.c**, **test.c**, and **lo.asm**
Consider the makefile

```
program: main.o test.o lo.o
    gcc -o program main.o test.o lo.o
main.o: main.c
    gcc -c main.c
test.o: test.c
    gcc -c test.c
lo.o: lo.asm
    nasm -f elf lo.asm
clean:
    rm *.o program
```

Example

If you type **make**, the following happens (assuming ***.o** files do not exist)

```
gcc -c main.c
gcc -c test.c
nasm -f elf lo.asm
gcc -o program main.o test.o lo.o
```

If you type **make clean**

```
rm *.o program
```

All the object files and the executable **program** are removed.

Calling **make**

- **make** searches for a file with name **makefile** in the current directory
- If there is no such a file, **make** searches for a file with name **Makefile**
- To use a file with any name, type
  make -f <filename>
- To see what would be executed, but without executing it, type **make -n**

Syntax

target: dependences
(TAB) command line(s)

- target: target to be built
- dependences : files on which the target depends
- Next line(s) contains command(s);
  must start with a TAB
- Each line must end with a return (`\n`)
- Comments start with `#`

Syntax

`cat -v -t -e makefile`

- **`-v -t`** shows TABs as **`^I`**
- **`-e`** shows **`$`** at the end of each line

Syntax

e.g. for the makefile from the 1st slide
`cat -v -t -e makefile` will output:

```
program:  main.o test.o lo.o$
^Igcc -o program main.o test.o lo.o$
main.o:  main.c$
^Igcc -c main.c$
test.o:  test.c$
^Igcc -c test.c$
lo.o:  lo.asm$
^Inasm -f elf lo.asm$
clean:$
^Irm *.o program$
```

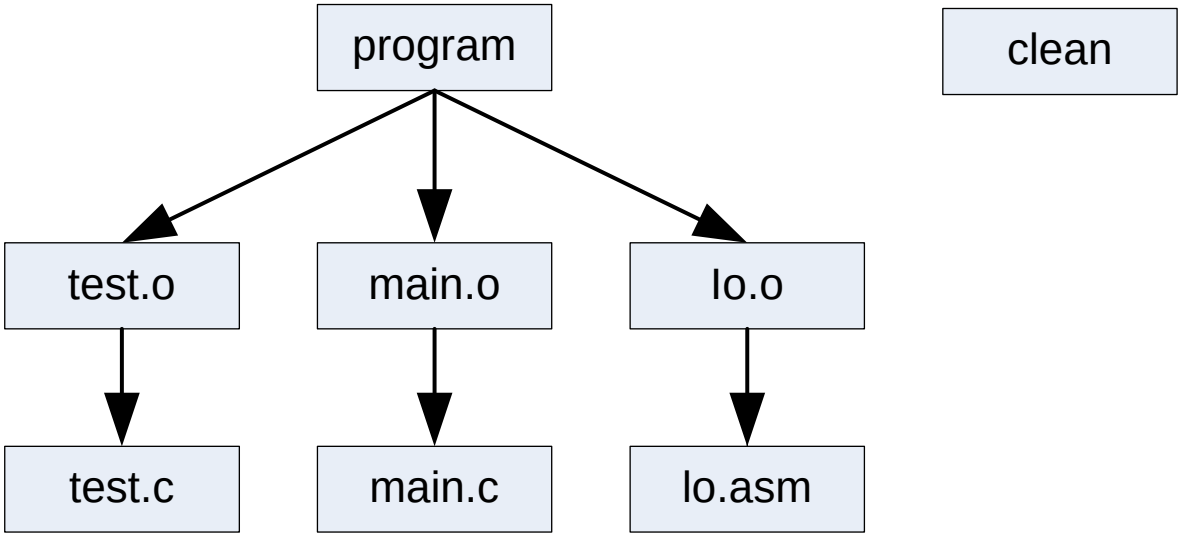
How it works

`make` makes the dependency graph, must be directed acyclic (= no cycles) graph.

Then it recursively goes down the tree and checks for each node if it is older than some of the files it depends on, and if so, rebuild the node. But first it checks if the files it depends on are “fresh”.

How it works

Consider the makefile from the 1st slide:



How it works

If **`make program`** is executed (or simply **`make`**)

- If **`test.c`** is newer than **`test.o`**, it will rebuild **`test.o`** by compilation **`gcc ...`**
- If **`man.c`** is newer than **`main.o`**, it will rebuild **`main.o`** by compilation **`gcc ...`**
- If **`lo.asm`** is newer than **`lo.o`**, it will rebuild **`lo.o`** by calling assembler **`nasm ...`**
- If any of **`test.o`**, **`main.o`**, and **`lo.o`** is newer than **`program`**, it will rebuild **`program`** by compilation and linking **`gcc ...`**

How it works

If **`make clean`** is executed

- Since there is no file called **`clean`** in the directory, it is "rebuild" by executing **`rm ...`**
- As long as there is no file called **`clean`** in the directory, typing **`make clean`** will thus always force the cleanup to be executed.

Macros

- Defined as `name = string`
- To access the value of **name:** `$(name)` or `${name}`
- Some internally defined macros:
  - `$(CC)` the default C compiler
  - `$(CXX)` the default C++ compiler
  - `$(LD)` loader/linker
- Example

```
main.o:  main.c
$(CC) -c main.c
```

- To see all internally defined macros, type `make -p`

Macros

- **$@** evaluates to current target.
  Here it evaluates to **program**:

```
program:  main.o test.o lo.o
    $(CC) -o $@ main.o test.o lo.o
```

- **$?** evaluates to a list of prerequisites that are newer than the current target:

```
program:  main.o test.o lo.o
    $(CC) -o $@ $?
```

A better makefile

```
OBJS = main.o test.o lo.o
AS = nasm
ASFLAGS = -f elf
program:  $(OBJS)
   $(CC) -o $@ ${OBJS}
main.o:  main.c
   $(CC) -c $?
test.o:  test.c
   $(CC) -c $?
lo.o:  lo.asm
   $(AS) $(ASFLAGS) $?
clean:
   rm $(OBJS) program
```

Setting flags

- **`CFLAGS = -g -Wall -ansi -pedantic -O2`**
  - **`-Wall`** warning on everything
  - **`-ansi`** **ANSI C**
  - **`-O2`** optimization level 2
  - **`-g`** for producing debugging information
- Use always **`-Wall`**, and for portability **`-ansi`**
- Example of using **`CFLAGS`**:

```
main.o:  main.c
   $(CC) $(CFLAGS) -c $?
```

Suffix rules

- ```
  .SUFFIXES: .o.c.asm
  .c.o:
      $(CC) $(CFLAGS) -c $<
  .asm.o:
      $(AS) $(CFLAGS) $<
  ```
- `$<` is like `$?` but is used only in suffix rules
- Suppose **make** wants to create **main.o**
  - from the suffix rules it knows to search for a file **main.c**
  - if it does not exist, it searches for **main.asm**
  - then it applies the suffix rule
  ```

Final makefile

```
.SUFFIXES: .o .asm
.asm.o:
  $(AS) $(ASFLAGS) $<
AS = nasm
ASFLAGS = -f elf
OBJS = main.o test.o lo.o
program:  $(OBJS)
   $(CC) -o $@ $?
clean:
  rm $(OBJS) program
```

Command line options

- `make CFLAGS=-g`
- `make CFLAGS="-g␣-Wall"`
- Overrides `CFLAGS` defined in the makefile