Makefiles

Ned Nedialkov

McMaster University
Canada

SE 3F03
January 2014

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
```

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
```

The object files and the executable 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: prerquisites
(TAB) command line(s)
```

- On left of : target to be built
- On right of : files on which it depends
- Next line(s) contains a command; must start with a TAB
- Each line must end with a return
- Comments start with #

- `cat -v -t e makefile`
  - `-v -t` shows TABs as `^I`
  - `-e` shows `$` at the end of each line
- e.g

```
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

```
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
```

- `make` checks if any of the files on the right of `:` are newer than program
- If so, it rebuilds `program`
- But before that, it checks if `main.c` is newer than `main.o`. If so, it recompiles `main.c`
- It applies the same to `test.o` and `lo.o`
- If any of the source files is newer than the corresponding object file, `program` is rebuilt
- `make` goes recursively down a “tree" and rebuilds targets

Macros

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

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

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

- `$@` 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 then 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

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

Suffix rules

- ```
  .SUFFIXES: .o.c.asm
  .c.o:
          $(CC) $(CFLAGS) -c $<
  .asm.o:
          $(AS) $(ASFLAGS) $<
  ```
- `$<` 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"`
- Overwrites `CFLAGS`