Systems Programming Laboratory, Spring 2023

Introduction to make

Abhijit Das
Bivas Mitra

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

February 2, 2023

Why make at all?

- All large software projects are designed as modules.
- Compiling and linking all the modules gives the final product (an application or a library).
- There may be hundreds of modules each consisting of multiple files.
- A complete compilation of several millions of lines of code is time-consuming.
- Not all modules are dependent on one another.
- If one module changes, only that module and other affected modules need to be recompiled.
- This process is called software building.
- The GNU make utility automates this building process.

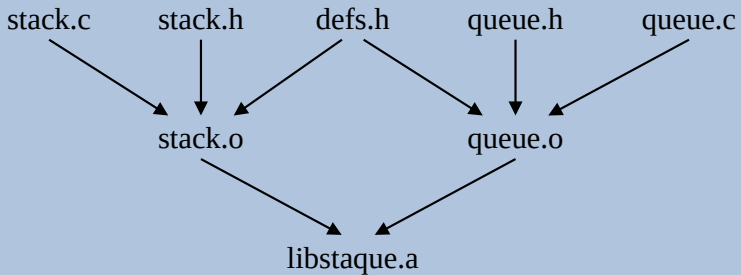
A simple example: Building the static library libstaque.a

- The following set of commands is used.

```
gcc -c -Wall stack.c
gcc -c -Wall queue.c
ar rcs libstaque.a stack.o queue.o
```

- These commands can be written in a shell script and executed to get the final product.
- For this small example, this is fine.
- If the source consists of thousands of files, compiling all of these is a slow process.
- Not all modules need recompilation after every change.
- If one makes (small/large) changes only in queue.c and/or queue.h, there is no need to recompile stack.c.
- Make helps you in the selective (re)compilation process.
- But you must instruct how to do it.

Example: The dependencies



- libstaque.a depends only on the object files stack.o and queue.o.
- stack.o can be generated by compiling stack.c with the –c option.
- This compilation additionally requires the header files defs.h and stack.h.
- queue.o can be generated by compiling queue.c with the –c option.
- This compilation additionally requires the header files defs.h and queue.h.

Makefile

- The dependency and compilation instructions are written in a file. The following names are searched in the given order.
  - GNUmakefile
  - makefile
  - Makefile
- For using other makefiles, run make with the –f option.

```
make -f mymakefile
```

- You run the utility as:

```
make
```

or

```
make TargetName
```

Rules in makefiles

- A rule is of the form:

```
TargetName: List of dependencies
        Command 1
        Command 2
        Command 3
        ...
```

- Each line of command must start with a tab.
- A line (may be empty) **not** starting with a tab ends the rule.
- The target may be the name of a file or a symbolic name (phony).
- The dependency list may be empty (but make knows some default dependencies).
- Absence of commands in rules is allowed. Such rules mean:
  - Set the dependencies.
  - Use a predefined make rule to build the target.

How make works

- make checks timestamps to decide which parts of the project need to be recompiled.
- The commands are executed if one or more dependency file(s) is/are modified **after** the target was last built.
- Phony targets are always built.

Rule examples

```
library: stack.o queue.o
        ar rcs libstaque.a stack.o queue.o

stack.o: defs.h stack.h
queue.o: defs.h queue.h
```

- library is a phony target that depends on stack.o and queue.o. The build involves invoking the `ar` command.
- stack.o (a filename target) depends on the header files defs.h and stack.h.
- queue.o (another filename target) depends on the header files defs.h and queue.h.
- What make already knows is this:
  - stack.o also depends on stack.c, and queue.o also depends on queue.c. There is no need to specify these dependencies.
  - stack.o can be obtained from stack.c and queue.o from queue.c by invoking gcc –c. It is not needed to write the commands explicitly.
- What make does not know is what additional compilation flags you need with gcc –c.

Rule examples (continued)

- Suppose that you call:

```
make library
```

- Since library is a phony target, it is always rebuilt.
- Before invoking `ar`, make checks whether any/both of the dependencies stack.o and queue.o need(s) to be rebuilt.
- If the timestamp of stack.o is more recent than all of the files defs.h, stack.h and stack.c, then stack.o is not rebuilt. If one or more of these dependencies is/are modified after the timestamp of stack.o, it is rebuilt using gcc –c.
- If the timestamp of queue.o is more recent than all of the files defs.h, queue.h and queue.c, then queue.o is not rebuilt. If one or more of these dependencies is/are modified after the timestamp of queue.o, it is rebuilt using gcc –c.

Which target to build?

- If you run make without any target name, the target of the **first rule** is built. For example, if library is the first rule in our example, it is built if make is called without an explicit target name.
- You can specify the target additionally like:

```
make stack.o
```

or

```
make queue.o
```

Make variables

- Variables can be set using the assignment operator = (recursive) or := (evaluate once).
- a variable VAR can be accessed as $(VAR) or ${VAR}.
- Default variables
  - SHELL specifies which shell to use for running the commands.
  - CC specifies the C compiler you want to use.
  - CFLAGS stands for the additional compilation flags that you use during gcc –c.

```
SHELL = /bin/sh
CC = gcc
CFLAGS = -O2 -g -I.
AR = ar
LIBNAME = libstaque.a
OBJFILES = stack.o queue.o

library: $(OBJFILES)
        $(AR) rcs $(LIBNAME) $(OBJFILES)

$(OBJFILES): defs.h
stack.o: stack.h
queue.o: queue.h
```

Run make

```
$ make
gcc -O2 -g -I.   -c -o stack.o stack.c
gcc -O2 -g -I.   -c -o queue.o queue.c
ar rcs libstaque.a stack.o queue.o
$ make
ar rcs libstaque.a stack.o queue.o
$ touch defs.h
$ make
gcc -O2 -g -I.   -c -o stack.o stack.c
gcc -O2 -g -I.   -c -o queue.o queue.c
ar rcs libstaque.a stack.o queue.o
$ touch queue.c
$ make
gcc -O2 -g -I.   -c -o queue.o queue.c
ar rcs libstaque.a stack.o queue.o
$
```

Rules to install

- Often the final products (like libstaque.a and the header files) need to be installed in the system area.
- Run make in the superuser mode as:

```
sudo make install
```

```
LIBDIR = /usr/local/lib
INCLUDEDIR = /usr/include
INCLUDESUBDIR = $(INCLUDEDIR)/staque

install: library
        cp $(LIBNAME) $(LIBDIR)
        -mkdir $(INCLUDESUBDIR)
        cp defs.h stack.h queue.h $(INCLUDESUBDIR)
        cp staque.h $(INCLUDEDIR)
```

- A dash before a command directs make to ignore errors. Here, if the directory /usr/include/staque already exists, mkdir fails. But make moves forward ignoring the error.

Run make install

```
$ sudo make install
[sudo] password for abhij:
ar rcs libstaque.a stack.o queue.o
cp libstaque.a /usr/local/lib
mkdir /usr/include/staque
mkdir: cannot create directory `/usr/include/staque': File exists
make: [Makefile:30: install] Error 1 (ignored)
cp defs.h stack.h queue.h /usr/include/staque
cp staque.h /usr/include
$
```

Cleaning previous builds

```
RM = rm -f

clean:
        -$(RM) $(OBJFILES)

distclean:
        -$(RM) $(OBJFILES) $(LIBNAME)
```

Difference between = and :=

- = is the *recursive* assignment operator.
- := is the *evaluate once* assignment operator.
- If the recursive evaluation of a variable VAR eventually (in one or more steps) depends upon $(VAR), then further expansion of $(VAR) will again involve $(VAR), and the process continues ad infinitum.

```
VAR1 = $(VAR2)
VAR2 = Hi $(VAR1)
```

- Here, `$(VAR1)` expands to `$(VAR2)` which in turn expands to `Hi $(VAR1)`.
- Replacing one (or both) = to := stops the infinite recursive substitution.

An example for the difference between = and :=

makefile

```
SHELL = /bin/bash

AS := Artim
AS = $(AS) Savib

ST = Sad
ST := $(ST) Tijihba

as:
        @echo Hi $(AS)

st:
        @echo Hi $(ST)
```

Running make

```
$ make as
makefile:4: *** Recursive variable 'AS' references itself (eventually).
Stop.
$ make st
Hi Sad Tijihba
```

Writing makefile in pieces

Syntax

```
include file1 file2 file3 ...
```

Example

```
STARTMKF = defs.mk primitives.mk
include preamble.mk $(STARTMKF) util*.mk
```

- Suppose that there are four matches `util1.mk`, `util2.mk`, `util3.mk`, `utilfinal.mk`.
- The following seven files are included:
  - `preamble.mk`
  - `defs.mk`
  - `primitives.mk`
  - `util1.mk`
  - `util2.mk`
  - `util3.mk`
  - `utilfinal.mk`

Recursive make

- Useful when several subdirectories possess independent makefiles.
- cd to each subdirectory, and call make.
- Each line of command opens a new shell, so both cd and make must be in the same line.

```
SHELL = /bin/sh

all:
        cd static; make
        cd shared; make

install:
        cd static; make install
        cd shared; make install

clean:
        cd static; make clean
        cd shared; make clean
```

Run recursive make

```
$ make
cd static; make
make[1]: Entering directory '/home/abhij/IITKGP/course/lab/SPL/Spring23/prog/libstaque/static'
gcc -O2 -g -I.   -c -o stack.o stack.c
gcc -O2 -g -I.   -c -o queue.o queue.c
ar rcs libstaque.a stack.o queue.o
make[1]: Leaving directory '/home/abhij/IITKGP/course/lab/SPL/Spring23/prog/libstaque/static'
cd shared; make
make[1]: Entering directory '/home/abhij/IITKGP/course/lab/SPL/Spring23/prog/libstaque/shared'
gcc -O2 -g -fPIC -I.   -c -o stack.o stack.c
gcc -O2 -g -fPIC -I.   -c -o queue.o queue.c
gcc -shared -o libstaque.so stack.o queue.o
make[1]: Leaving directory '/home/abhij/IITKGP/course/lab/SPL/Spring23/prog/libstaque/shared'
$ make clean
cd static; make clean
make[1]: Entering directory '/home/abhij/IITKGP/course/lab/SPL/Spring23/prog/libstaque/static'
rm -f stack.o queue.o
make[1]: Leaving directory '/home/abhij/IITKGP/course/lab/SPL/Spring23/prog/libstaque/static'
cd shared; make clean
make[1]: Entering directory '/home/abhij/IITKGP/course/lab/SPL/Spring23/prog/libstaque/shared'
rm -f stack.o queue.o
make[1]: Leaving directory '/home/abhij/IITKGP/course/lab/SPL/Spring23/prog/libstaque/shared'
$
```

Practice exercises

1. Some (but not all) targets in a software project needs rebuilding. You do not want to rebuild these targets actually. Instead you only want to know what commands will be executed for updating the targets. Figure out how you can run make to do this.
2. Investigate how you can force make to rebuild a target and all its dependencies even if one or more of these targets are up-to-date.
3. Investigate the behavior of make in the presence of circular dependencies as in the following makefile.

```
a: b c
	@echo "Building a"
b: c a
	@echo "Building b"
c: a b
	@echo "Building c"
```

4. Consider the following makefile containing an error (the command **ecko**). You run **make**, **make -k**, and **make -i**. What differences do you see? Explain.

```
a: b c
	@echo "Building a"
b:
	@ecko "Building b"
c:
	@echo "Building c"
```

Practice exercises

5. Consider the following two makefiles.

makefile1

```
A = $(B)
B = "Hello"
all:
	@echo $(A)
```

makefile2

```
A := $(B)
B = "Hello"
all:
	@echo $(A)
```

What difference(s) do **`make -f makefile1`** and **`make -f makefile2`** exhibit? Explain why.

6. **[Target-specific variables]** Consider the following makefile.

```
A = $(B)
one:
	@echo $(A) $(B)
two:
	@echo $(A) $(B)
```

You want **`make one`** to print **`abra abra`**, and **`make two`** to print **`cadabra cadabra`**. You are not allowed to change the above five lines. Explain how you can add some extra lines to this makefile in order to achieve your goal.

Practice exercises

7. A software project builds a library **`libfoobar.a`** by compiling **`foo.c`** and **`bar.c`**. The file **`foo.c`** includes **`foo.h`**, whereas the file **`bar.c`** includes **`bar.h`**. The header file **`foo.h`** in turn includes **`foobar.h`**, **`foo1.h`**, and **`foo2.h`**, whereas the header file **`bar.h`** includes **`foobar.h`**, **`bar1.h`**, **`bar2.h`**, and **`bar2.h`**. All the files reside in the same directory. Write a makefile to build the library. You do not have to actually write any .c or .h file

8. Suppose that you want to create a dynamic library of functions for working with rational numbers of the form a/b with a any integer and with b a positive integer. To do this, you declare the **`rational`** data type (and nothing else) in a header file **`rat.h`**. Then, you write three C files along with three corresponding header files.

`rbasic.c` (and **`rbasic.h`**): This defines a **`gcd()`** function and another function **`rconv()`** that converts a rational number a/b to the lowest terms satisfying $\gcd(a,b) = 1$.

`rarith.c` (and **`rarith.h`**): This defines basic arithmetic functions on rational numbers: **`radd()`**, **`rsub()`**, **`rmul()`**, and **`rdiv()`**.

`rmath.c` (and **`rmath.h`**): This defines the functions **`r2dbl()`** [convert a/b to a **`double`**], **`rsqrt()`** [**`double`**-valued square root of a rational number], and **`rlog()`** [**`double`**-valued log of a rational number]. This file uses the math library available in a standard system directory as **`libm.a`**.

All the files related to rational numbers reside in one directory. Write a makefile in the directory to generate the dynamic library **`librational.so`**. You do not have to actually write any **`.c`** or **`.h`** file.

Practice exercises

9. You have a project whose files are stored in the following directory hierarchy. The root of the project is the directory **`/home/foobar/project`**, which has two subdirectories **`basics`** and **`utilities`**. The **`utilities`** directory further has two subdirectories **`online`** and **`offline`**. Each directory (including the project root) has a makefile which can be used independently of one another. But to build the project, all the makefiles in all the directories must be used. Show how you can build the project by executing a single make command from the command prompt (write the makefile this make command will execute).

10. You need to build two separate libraries **`libfoo.a`** and **`libbar.a`**. The foo files reside in a subdirectory **`foo`**, and the bar files reside in a subdirectory **`bar`**. Each subdirectory has its independent makefile to build the corresponding library in the respective directory. The foo library is independent, but the compilation of the bar library uses the flags **`-L../foo`** and **`-lfoo`** to use the functions in **`libfoo.a`**. The source files of the bar directory include appropriate header files from the foo directory. But the makefile of the **`bar`** directory does not specify dependencies on any of the foo files. So you need to build **`libfoo.a`** first and then **`libbar.a`**. Write a makefile to prepare the two libraries using a single **`make`** command.

(a) How do you compile an application program in the current directory, that uses only the bar library?

(b) Suppose that the sources of the foo library are changed. If you run **`make`** from the current directory, will it rebuild **`libbar.a`**? What if you recompile the application program after this **`make`**?

11. Repeat the last exercise with the exception that the dynamic libraries **`libfoo.so`** and **`libbar.so`** are to be built (instead of the static ones).