CS29206: Systems Programming Lab

Autumn 2025

Creating and using makefiles

Acknowledgement

Several slides, and contents of some others, have been taken from the slides made by Prof. Abhijit Das for an earlier offering of the course

Resources

- "The Linux Development Platform" by R. Rehman and C. Paul(Chapter 4)
- GNU make manual, https://www.gnu.org/software/make/manual/

Introduction

- As discussed, a software project may consist of a very large number of files (separated in different modules of the project)
  - Creating the final application/library the project provides requires compiling and linking all of these (building the software)
- Challenges:
  - Compiling the entire thing from scratch every time may take a very long time
  - If one file changes, should not have to recompile everything, only that file and any other files that depend on itshould be recompiled
    - Example: In the staque library seen
      - If stack.c changes, only stack.c need to be recompiled, and the library built again
      - If defs.h changes, both stack.c and queue.c need to be recompiled as both stack.h and queue.h depend on defs.h

Make utility

- The GNU make utility automates this building process
- Provides a way to create a file with
  - Compilation instructions for different modules
  - Specifying dependencies between files/modules
- Running the make utility on this file will
  - Compile only files that have changed since the last time they are compiled
    - Uses the Last-Modified-Time on the file to check if a file needs to be compiled
    - If any file is re-compiled, re-compiles everything that depend on it
  - Allows for building specific modules only instead of the whole project also
  - Allows for building the entire project (irrespective of time change) also (clean build)

- The file normally has a default name
  - GNUmakefile, or Makefile, or makefile (the last one is more common)
  - make command, when run, will search for the files with names in this order in the current directory
  - You can force the make utility to use other file names by using the -f option when run
- Running the make utility
  - make
    - Will look for a make file in the order shown above, will build the first target in that file
  - make -f <filename>
    - Will build the first target in the file <filename>
  - make <target name>
    - Will build only the specified target in the make file (may not be first)
  - We will see what is a target shortly

Contents of a make file

- Basically, has a set of rules
- Each rule is of the form

```
Targetname: List of dependencies
        command 1
        command 2
        command 3
        …
```

- Each line of a 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)
- A target is rebuilt whenever a dependency file has a timestamp (last-modified-time) that is newer than the target
- A target may have no dependency, in which case it is always rebuilt.
- The commands in the rule are executed to build the target from Phony targets are always built
- Absence of commands in rules is allowed. Such rules mean:
  - Set the dependencies
  - Use a predefined make rule to build the target

Example: Building the staque library

The makefile: Version 1

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

stack.o: stack.h defs.h

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

- library is a phony target that depends on stack.o, queue.o. Given the latest versions of these, the ar command creates the library from there
- stack.o and queue.o depends on
  - the respective header files they use, which we specify
  - The respective .c files. But make already knows .o comes from .c, so no need to specify either the dependency or the gcc –c command to get the .o from .c

Initially. Note the last-modified-time of the files marked in blue

```
$ ls -l
-rw-r--r-- 1 agupta faculty 138 Aug  4 15:37 defs.h
-rw-r--r-- 1 agupta faculty 117 Aug  4 15:49 makefile
-rw-r--r-- 1 agupta faculty 777 Aug  4 15:18 queue.c
-rw-r--r-- 1 agupta faculty 748 Aug  4 15:18 queue.h
-rw-r--r-- 1 agupta faculty 894 Aug  4 15:37 stack.c
-rw-r--r-- 1 agupta faculty 617 Aug  4 15:18 stack.h
```

Run make. Builds the first target library. Finds it depends on the targets stack.o and queue.o and so builds those targets first. So all targets are built.

```
$ make
cc    -c -o stack.o stack.c
cc    -c -o queue.o queue.c
ar rcs -o libstaque.a stack.o queue.o
```

The .o files and the .a library are created.

```
$ ls -l *.o *.a
-rw-r--r-- 1 agupta faculty 6102 Aug  4 15:50 libstaque.a
-rw-r--r-- 1 agupta faculty 2832 Aug  4 15:50 queue.o
-rw-r--r-- 1 agupta faculty 2896 Aug  4 15:50 stack.o
```

Modify only stack.c. “touch” is a command that just changes the last-modified-time of the file, not the content. Run make again. make again tries to build the target library. It compares the times of stack.o (Aug 4 15:50) with stack.c (Aug 4 15:55), finds stack.c is more recent, so builds the target stack.o again. queue.c is older than queue.o, so target queue.o is not built again. So only the changed file is recompiled as we want.

```
$ touch stack.c
$ ls -l stack.c
-rw-r--r-- 1 agupta faculty 894 Aug  4 15:55 stack.c
$ make
cc    -c -o stack.o stack.c
ar rcs -o libstaque.a stack.o queue.o
```

```
$ ls -l
-rw-r--r-- 1 agupta faculty  138 Aug  4 15:37 defs.h
-rw-r--r-- 1 agupta faculty 6102 Aug  4 15:56 libstaque.a
-rw-r--r-- 1 agupta faculty  117 Aug  4 15:49 makefile
-rw-r--r-- 1 agupta faculty  777 Aug  4 15:18 queue.c
-rw-r--r-- 1 agupta faculty  748 Aug  4 15:18 queue.h
-rw-r--r-- 1 agupta faculty 2832 Aug  4 15:50 queue.o
-rw-r--r-- 1 agupta faculty  894 Aug  4 15:55 stack.c
-rw-r--r-- 1 agupta faculty  617 Aug  4 15:18 stack.h
-rw-r--r-- 1 agupta faculty 2896 Aug  4 15:56 stack.o
```

```
$ touch queue.h
$ make
cc    -c -o queue.o queue.c
ar rcs -o libstaque.a stack.o queue.o
```

queue.h is changed. make finds queue.h is more recent than queue.o, and queue.o depends on queue.h. So make rebuilds the target queue.o, and then the library which depends on it. stack.o is not rebuilt as nothing it depends on has changed.

```
$ touch defs.h
$ make
cc    -c -o stack.o stack.c
cc    -c -o queue.o queue.c
ar rcs -o libstaque.a stack.o queue.o
```

defs.h is changed. Since both the targets stack.o and queue.o depend on defs.h, so make rebuilds both the targets stack.o and queue.o, and then the library which depends on them.

- What if creating the .o from .c requires additional compilation flags? (For example, you want to compile with -Wall
  - make will only use the -c flag by default. If you want anything extra, you have to specify the compilation command yourself. So your makefile will now look like this.

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

stack.o: stack.h defs.h
        gcc –Wall –c –o stack.o stack.c

queue.o: queue.h defs.h
         gcc –Wall –c –o stack.o stack.c
```

We will do more of makefile in the next class

Practice in Lab

- Get the codes of the .h and .c files for stack and queue example. Type in the makefile (name it makefile) and run the following commands. At each step see what happens from what is (and is not) printed, and the timestamps of the files using ls -l
  - make
  - make makefile
  - make library
  - make stack.o
  - make queue.o
- Change each of the file's timestamps (using the touch command) and run make and explain to yourself what you see getting displayed.

Practice in Lab

1. Can you add a target named clean at the end of the makefile such that building the clean target will unconditionally delete all .o files (only) from the directory?
   - Will this target have any dependency?
   - What will be the command?
   - What command will you use from the $ prompt to build the clean target?
2. Suppose that the library libstaque.a also includes functions from a file mymath.c. mymath.c has only 1 function called mysqrt() with the prototype void mysqrt(double) which just calls the C math library function sqrt() to compute and print the square root of the parameter passed.
   - Write the mymath.c file. No need to create any separate mymath.h file.
   - Modify the makefile so that
     - The library libstaque.a will also include the function
     - In addition to the static library, the dynamic library libstaque.so is also created when make is run