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/
- Slides from Prof. Abhijit Das’s page (https://cse.iitkgp.ac.in/~abhij/course/lab/SPL/Spring24/slides/make.pdf)

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 it should 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
  - 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
```

Comments, cleaning files, and clean build

```
#makefile to create the libstaque.a static
library

#library is a phony target.
library: stack.o queue.o
        ar rcs -o libstaque.a stack.o queue.o

#target to build stack.o
stack.o: stack.h defs.h

#target to build queue.o
queue.o: queue.h defs.h

#phony target to clean all .o files
clean:
        rm -f *.o
```

- Lines starting # are taken as comments
- make will build the target library
- make clean will just remove all .o files, no compilation
- To force all files to be compiled (irrespective of last modified time),
  - First run "make clean"
  - Then run "make" (will rebuild both stack.o and queue.o targets as trying to build library will not find either stack.o or queue.o files)

```
$ make
ar rcs -o libstaque.a stack.o queue.o
$ make clean
rm -f *.o
$ make
cc    -c -o stack.o stack.c
cc    -c -o queue.o queue.c
ar rcs -o libstaque.a stack.o queue.o
```

What you have seen so far

- Motivation for using make and makefile
- Notion of building a software using make and makefile
- Rules (targets, dependencies and commands) in a makefile
  - All rules you have seen so far are explicit rules, there is a rule specified explicitly for every target used
  - Though we have used some implicit dependencies and commands
    - .o depends on .c
    - Default C compiler is invoked automatically to create .o from .c
    - Called Implicit as you do not need to specify it explicitly
- Adding your own compilation flags
- Putting comments in makefile

More on rules and commands

Implicit rules

- We have seen implicit dependencies and commands
- Even the whole rule can be implicit

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

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

- stack.c and queue.c got compiled automatically as make didn't find any .o files and rules to build the targets stack.o and queue.o that the target library depends on.
- Invoked the implicit rule to get .o from .c
- Running make again does not recompile the .c files as now it finds the .o files

Using Implicit things can be tricky

- The line starting with tabs are shell commands
- Can be anything, not just compilation commands
  - Example: the rm -f *.o command used for target clean
- We have just added an echo command to stack.o target
- Now remove stack.o, then run make
- We would expect stack.o to be rebuilt

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

stack.o: stack.h defs.h
            echo "building stack.o"

queue.o: queue.h defs.h

clean:
        rm -f *.o
```

```
$ rm *.o
$ make -f makefile-echo
$ make -f makefile-echo
echo "building stack.o"
building stack.o
cc    -c -o queue.o queue.c
ar rcs -o libstaque.a stack.o queue.o
ar: stack.o: No such file or directory
make: *** [makefile-echo:3: library] Error 1
```

- Prints the echo output
- But doesn't rebuild stack.o, gives an error
- The problem
  - Since you added a command to the target stack.o, make will not do anything extra for that target, assumes you know what to do
    - No implicit command invoked
  - You will have to add your own compilation command to the target stack.o now
- queue.o was built fine by implicit command as it has no other command

The corrected makefile

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

stack.o: stack.h defs.h
	echo "building stack.o"
	gcc -Wall -c stack.c -o stack.o

queue.o: queue.h defs.h

clean:
	rm -f *.o
```

```
$ rm *.o
$ make -f makefile-echo
echo "building stack.o"
building stack.o
gcc -Wall -c stack.c -o stack.o
cc    -c -o queue.o queue.c
ar rcs -o libstaque.a stack.o queue.o
```

- In general, using implicit rules can give errors or unwanted results if you do not understand exactly how it works
- Suggest that you use everything explicit now
  - Specify a rule explicitly for each target used
  - Specify all dependencies explicitly
  - Specify all commands explicitly

Using Variables

Using variables

- A makefile will typically have a bunch of variables
- Defined usually at the beginning before the start of rules
- Can be accessed anywhere in the makefile by placing the variable inside $( ), will be replaced with its value
- Helps in writing structured and easily understandable makefiles
  - Good way to define repeated things like the same dependencies, same compilation flags etc.

```
OBJS = stack.o queue.o
HDR_S = defs.h stack.h
HDR_Q = defs.h queue.h
CFLAGS = -Wall -c
AR = ar rcs
CC = gcc
RM = rm -f

library: $(OBJS)
        $(AR) -o libstaque.a $(OBJS)

stack.o: $(HDR_S) stack.c
        $(CC) $(CFLAGS) stack.c -o stack.o

queue.o: $(HDR_Q) queue.c
        $(CC) $(CFLAGS) queue.c -o queue.o

clean:
        $(RM) *.o
```

- Runs exactly the same as earlier

```
$ make clean
rm  -f *.o
$ make
gcc -Wall -c stack.c -o stack.o
gcc -Wall -c queue.c -o queue.o
ar rcs -o libstaque.a stack.o queue.o
```

- Some variables have default meanings
  - 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
    - If you use implicit command for compilation from .c to .o, -c is automatically added CFLAGS provides extra options you may want to add
    - We have used explicit commands in our examples so we have also specified –c in it

Types of variables

- Two types of variables can be defined
  - Variables defined using the recursive assignment operator =
    - Allows for a value of a variable to be expanded/changed based on value of another variable defined later in the makefile
  - Variables defined using the evaluate once assignment operator :=
    - makesmakes the definition of a variable fixed at the time of definition

```
OBJ1 = ftp.o
OBJ2 = common.o
OBJS = $(OBJ1) $(OBJ2)
mytarget:
        echo $(OBJS)
OBJ1 = ftp.o tftp.o
```

```
$ make
ftp.o tftp.o common.o
```

```
OBJ1 = ftp.o
OBJ2 = common.o
OBJS := $(OBJ1) $(OBJ2)
mytarget:
        echo $(OBJS)
OBJ1 = ftp.o tftp.o
```

```
$ make
ftp.o common.o
```

- Should be careful in using recursive variables
  - If = is used, 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.
  - Example:

    VAR1 = $(VAR2)

    VAR2 = Hi $(VAR1)

    - Here, $(VAR1) expands to $(VAR2) which in turn expands to Hi $(VAR1), and so on ad infinitum
    - Replacing one (or both) = to := stops the infinite recursive substitution

Working with more than one makefile

Using more than one makefile

- Large projects will have multiple modules with code spread over multiple directories/subdirectories
- Having a single makefile to build the entire project will be impractical for such large projects
- Typically there will be multiple makefiles spread over multiple directories/subdirectories
- Two ways to handle multiple makefiles
  - Include a makefile inside another makefile
    - Will work the same way as #include for C files
  - Have higher level makefiles that call make on lower level makefiles

Including other makefiles

- Use the include keyword in the makefile
  - include <file1> <file2> …
  - Files will be included in the order they are specified (<file1> first, <file2> next, …)
- Example:
  - Suppose we want to build separate libraries for stack and queue functions first
    - Build libstack.a from stack.c, defs.h, and stack.h
    - Build libqueue.a from queue.c, defs.h, and queue.h
    - Finally build libstaque.a from libstack.a and libqueue.a

```
HDR_S = defs.h stack.h
CFLAGS = -Wall -c
CC = gcc

libstack: stack.o
	ar rcs -o libstack.a stack.o

stack.o: $(HDR_S) stack.c
	$(CC)  -c stack.c -o stack.o
```

makefile-stack

makefile

```
OBJS = libstack libqueue

library: $(OBJS)
	ar rcs -o libstaque.a libstack.a libqueue.a

include makefile-stack
include makefile-queue
```

makefile-queue

```
HDR_Q = defs.h
CFLAGS = -Wall -c
CC = gcc

libqueue: queue.o
	ar rcs -o libqueue.a queue.o

queue.o: $(HDR_Q) queue.c
	$(CC)  -c queue.c -o queue.o
```

```
$ ls
defs.h    makefile-queue queue.h  stack.c makefile  makefile-stack  queue.c stack.h

$ make
gcc  -c stack.c -o stack.o
ar rcs -o libstack.a stack.o
gcc  -c queue.c -o queue.o
ar rcs -o libqueue.a queue.o
ar rcs -o libstaque.a libstack.a libqueue.a

$ ls
defs.h     libstaque.a     makefile-stack  queue.h  stack.c libqueue.a  makefile queue.o
stack.h  libstack.a  makefile-queue  queue.c     stack.o
```

- But typically all files will not be in the same directory
  - We will want all stack related files in a stack directory, same for queue
- Lets move all stack-related files to a stack subdirectory, same for queue. Main directory is stack-queue-lib

Modified makefile in main directory

```
OBJS = libstack libqueue

library: $(OBJS)
	ar rcs -o libstaque.a libstack.a libqueue.a

include ./stack/makefile

include ./queue/makefile
```

```
$ pwd
..../stack-queue-lib
$ ls
makefile  queue  stack
$ ls stack
defs.h  makefile  stack.c  stack.h
$ ls queue
defs.h  makefile  queue.c  queue.h
$ make
make: *** No rule to make target 'defs.h',
needed by 'stack.o'.  Stop.
```

Error as the makefile for libstack.a is running in stack-queue-lib directory only, defs.h is not there

Recursive/Hierarchical make

- We saw that including makefiles in the top level makefile may have problems with large projects
- Use hierarchical make
  - Top level makefile just invokes the make command on lower level makefiles directly
  - Lower level makefiles also ensures that files needed by top level makefiles are copied to appropriate place as part of make
- For our example
  - makefile in stack-queue-library directory will invoke the makefiles in stack and queue subdirectories
  - Stack subdirectory will build libstack.a and copy it to stack-queue-library directory (so that the ar command for building libstaque.a finds it)

makefile in stack-queue-lib directory

```
OBJS = libstack libqueue

library: $(OBJS)
	ar rcs -o libstaque.a libstack.a libqueue.a

libstack:
	cd stack; make

libqueue:
	cd queue; make
```

makefile in stack subdirector (line in blue added)

```
HDR_S = defs.h stack.h
CFLAGS = -Wall -c
CC = gcc

libstack: stack.o
	ar rcs -o libstack.a stack.o
	cp libstack.a ..

stack.o: $(HDR_S) stack.c
	$(CC)  -c stack.c -o stack.o
```

Make the same change to makefile in queue subdirectory, then run make from stack-queue-lib directory. Study the output printed by the make command on screen, tells you exactly what is going on.

Interaction of make with shell

- Each line starting with a tab is a shell command
- Each command is executed in a new subshell created
- If you want more than one command to execute in the same subshell, write them in one line separated by semicolon
- Consider a toy makefile to be run from /home/mydir. It has only one phony target whose intended purpose is to change to parent directory and print the directory (so print /home)

```
mytarget:
	cd ..
	pwd
```

WRONG: will print /home/mydir only. WHY?

```
mytarget:
	cd ..; pwd
```

CORRECT: will correctly print /home

Another example

- Consider our earlier makefile in stack-queue-lib directory
- Change it to have the cd and make commands in different lines
- If you run make now, the make command for libstack and libqueue target will run the make command on the makefile in stack-queue-lib directory only, not those in stack and queue subdirectory!!
  - And cause an infinite loop!!!
- Effect of the cd command (directory change) is gone when the line changes, as a new shell is invoked to run make, and its current directory is same as the directory it is invoked from (stack-queue-lib)

```
OBJS = libstack libqueue

library: $(OBJS)
	ar rcs -o libstaque.a libstack.a libqueue.a

libstack:
	cd stack
	make

libqueue:
	cd queue
	make
```

- However, sometime you want them on different lines
- We could have also written the top-level makefile as shown. When the target make-libs is built:
  - A new shell is created for the cd command. Will go to stack subdirectory, call make in the same shell (so it is still in stack subdirectory), so invokes makefile in stack directory
  - Come back to the parent shell on line change, so back to the main (stack-queue-lib) directory
  - A new shell is created for the cd command again. Will go to queue subdirectory, call make in the same shell (so it is still in queue subdirectory), so invokes makefile in queue directory
- Need to understand what is happening and choose as per your need

```
library: make-libs
	ar rcs -o libstaque.a libstack.a libqueue.a

make-libs:
	cd stack; make
	cd queue; make
```

Some final issues: 1

- So far, in all makefiles shown for the stack/queue example, the ar command is always executed anyway, even if nothing that the library being built depends on changes
  - Kept deliberately simple as we just wanted to show basic one-level dependency (library on stack.o and queue.o) for introducing make
  - But this is wasteful. Should not have to run ar if nothing changes
- Can be easily fixed. Just create a two-level dependency

```
OBJS = stack.o queue.o
HDR_S = defs.h stack.h
HDR_Q = defs.h queue.h
CFLAGS = -Wall -c
CC = gcc

library: libstaque.a

libstaque.a: $(OBJS)
	ar rcs -o libstaque.a $(OBJS)

queue.o: $(HDR_Q) queue.c
	$(CC) -c queue.c -o queue.o

stack.o: $(HDR_S) stack.c
	$(CC)  -c stack.c -o stack.o
```

Some final issues: 2

- Building libraries from libraries
  - For building libstaque.a from the individual libraries libstack.a and libqueue.a, we used the command "ar rcs -o libstaque.a libstack.a libqueue.a" in the makefile
  - This doesn't work! (mentioned in class while explaining that there is something wrong with it ☺, a few of you tried it also)
    - Do an "nm libstaque.a" to see if you can see all function names.
    - Trying to compile a file with -lstaque will give error (run and see)
  - But the ar command runs fine with no errors and builds libstaqueue.a (just does not build the library properly), so allows the make examples to go through fine
    - Kept simple deliberately to explain make, as you know the ar rcs command only so far
- But then what is the correct way to include functions from one library inti another library?

- Intuitive and easy
  - You used .o files to create the libraries. First do the reverse and extract the .o files from the libraries! (Use x option of ar command)
  - Make the new library with all the .o files extracted (and any others if any) the same way as before
  - Put the sequence of commands on the right-side window (except ls, which is just for demonstration) in the makefile replacing the "ar rcs …" command.
  - The libstaque.a generated will now work fine
    - Do an "nm libstaque.a" again and compare with the earlier one

No .o files here initially

```
$ ls -l
total 20
-rw-r--r-- 1 agupta faculty 3060 Aug 10 12:10 libqueue.a
-rw-r--r-- 1 agupta faculty 3114 Aug 10 12:10 libstack.a
-rw-r--r-- 1 agupta faculty  143 Aug 10 12:05 makefile
drwxr-xr-x 2 agupta faculty 4096 Aug 10 12:10 queue
drwxr-xr-x 2 agupta faculty 4096 Aug 10 13:35 stack
```

```
$ rm -f *.o
$ ar x libstack.a
$ ar x libqueue.a
$ ls
libqueue.a  libstack.a  makefile  queue
queue.o  stack  stack.o
$ar rcs libstaque.a *.o
$ ls
libqueue.a  libstaque.a  queue   stack
libstack.a  makefile     queue.o  stack.o
```

Summary

- Understand rules, target, dependency, commands
- Understand both implicit and explicit rules and dependencies
  - But for assignments, use all explicit (rules, dependencies, commands)
  - But you may get questions in written lab tests or moodle quizzes on both
- Understand use of variables, both = and :=
  - For simple uses in your assignments, any one can be used
  - But you may get questions in written lab tests or moodle quizzes on both
- Understand how to include makefiles in other makefiles and recursive make when working with multiple makefiles
  - For assignments, use recursive make only
  - But you may get questions in written lab tests or moodle quizzes on both