# CS29206 Systems Programming Laboratory Spring 2024

## Introduction to make

**Abhijit Das**
**Pralay Mitra**

- 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.

stack.c   stack.h   defs.h   queue.h   queue.c

stack.o   queue.o
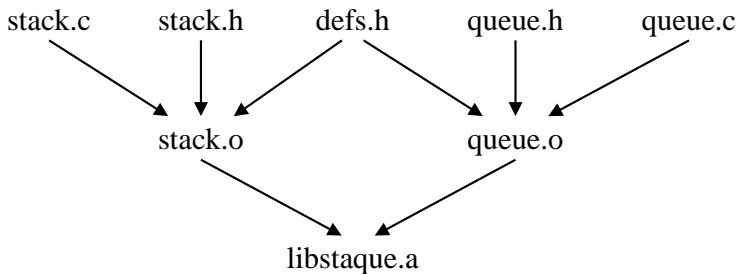
libstaque.a

- 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.

- 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)
```