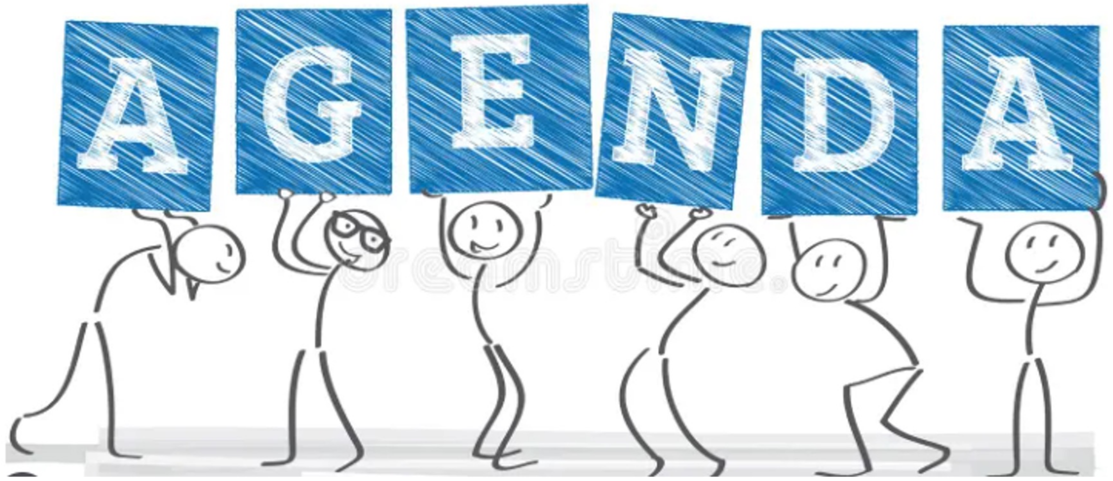
Operating Systems

Lecture 1.4

Linux **`make`** utility

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda



- Intro to UNIX `make` utility
- Structure & Components of a `makefile`
- How `make` utility work
- Multiple Targets in `makefile`
- Multiple `makefiles` in a Project
- Use of `macros` in a `makefile`
- Binary vs open source software
- Installing open source software

Understanding the `make` Utility

What is `make` utility in Linux?

The UNIX "**`make`**" utility is designed to **automate** the **task of compiling** large complex programs with **multiple components** efficiently

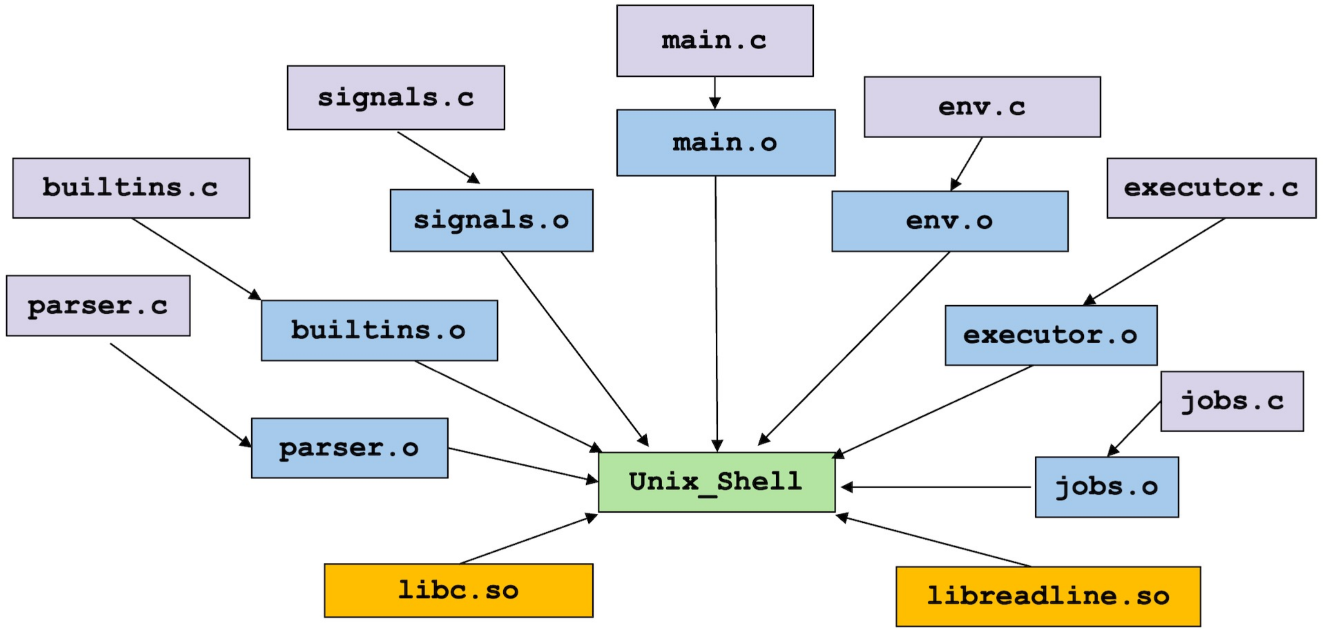
Why make utility?

- In C or C++ software development, projects are typically composed of multiple source files (`.c`) and header files (`.h`).
- The Linux kernel 6.12 source code contains 87,235 total files with 39,816,411 lines of code and text, surpassing 40 million lines as of January 2025. The estimated distribution is shown below:
  - .c files : ~35,000-40,000 files
  - .h files: ~15,000-20,000 files
  - Other files: ~30,000-35,000 files (Makefiles, Kconfig, documentation, scripts, etc.)
- If even a small change is made in one source or header file, we need to manually recompile every file and then relink them to create the final executable (`vmlinuz`).
- This becomes extremely inefficient because:
  - Recompiling all files every time is a time-consuming process.
  - Developers must manually recompile dependent files.
  - Forgetting to recompile a changed dependency might cause runtime bugs or incorrect results.

Why make utility? (cont...)

Consider the development of a custom Unix shell. A shell is a complex program made up of many interdependent components. For instance, you divide the shell project into different parts for handling different components. Each source file has its own purpose, and multiple of them may include the same headers. To build this project, we need to compile each .c file into an object file and then all object files are linked into a final executable.

- **`main.c`**: Entry point of the shell
- **`parser.c`:** For handling user input parsing and command tokenization
- **`executor.c`:** For executing the parsed commands
- **`builtins.c`:** For implementing built-in commands (like cd, exit, export)
- **`env.c`:** For managing environment variables (get/set)
- **`jobs.c`:** For managing background and foreground job control
- **`signals.c`:** For Handling signal-related functionality (e.g., Ctrl+C, Ctrl+Z)



Suppose you make a small change in the `input.c` file:
- **Option A:** Recompile all **`.c`** files and then re-link everything (very slow)
- **Option B:** Recompile only **`input.c`** and then re-link everything (fast and efficient)

What if you want to make changes in **`input.h`** file:
- You must recompile every `.c` file that includes **`input.h`**. and then re-link everything

Why make utility? (cont…)

Solution is `make` utility: The UNIX make utility is a powerful tool that automates compilation and linking by analyzing dependencies and modification times, ensuring only the changed or affected files are compiled.

- The **`make`** utility reads a specification file named **`makefile`** or **`Makefile`**, that is a configuration file for make utility describing how the modules of a software system depend on each other.
- The **`make`** utility uses this dependency specification in the `makefile`, and the time when various components were modified, to minimize the amount of recompilation.

Advantage of `make` utility:

- Makes management of large s/w projects with multiple source files easy.
- No need to recompile a source file that has not been modified, only those files that have been changed are recompiled, others are simply relinked.

Structure of `Makefile`

Structure of `makefile`

A `makefile` consists of a set of dependency rules having following format:

```
target: dependency1 dependency2 … dependencyN
<tab> command
```

- The **target** is the name of the executable to be build.
- The **dependency list** are the name of the files on which the target depends. These are the files that are needed to make the target. These files need to exist before the command(s) for the target are run. If any dependency is newer than the target, the target will be rebuilt.
- The **command** is the shell command to create the target from dependencies. Each line must begin with the tab character, not spaces.

```
# File: makefile
hello: hello.o
	gcc hello.o -o hello
	echo "Build hello"
hello.o: hello.c
	gcc -c hello.c
```

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Example

Let’s consider a basic C program that prints a greeting message. The file **main.c** is the entry point of the program, while **greet.c** and **greet.h** contain the greet function definition and function declaration respectively.

```c
// file: main.c

#include "greet.h"
int main(){
  greet("Aris");
  return 0;
}
```

```c
// file: greet.c

#include <stdio.h>
#include "greet.h"
void greet(const char* name){
   printf("Greetings %s!\n",
name);
}
```

```c
// file: greet.h

void greet(const char* name);
```

The **makefile** to build the final executable for this project is shown. When you run **make**, by default it starts with the first target and checks if **greetings** needs to be rebuilt. It does this by checking the timestamp of its dependencies (`main.o`, `greet.o`). If any one of the object files needs to be rebuilt (because its `.c` or `.h` file changed), **make** runs the associated rule. Finally, it links the updated .o files into the final program.

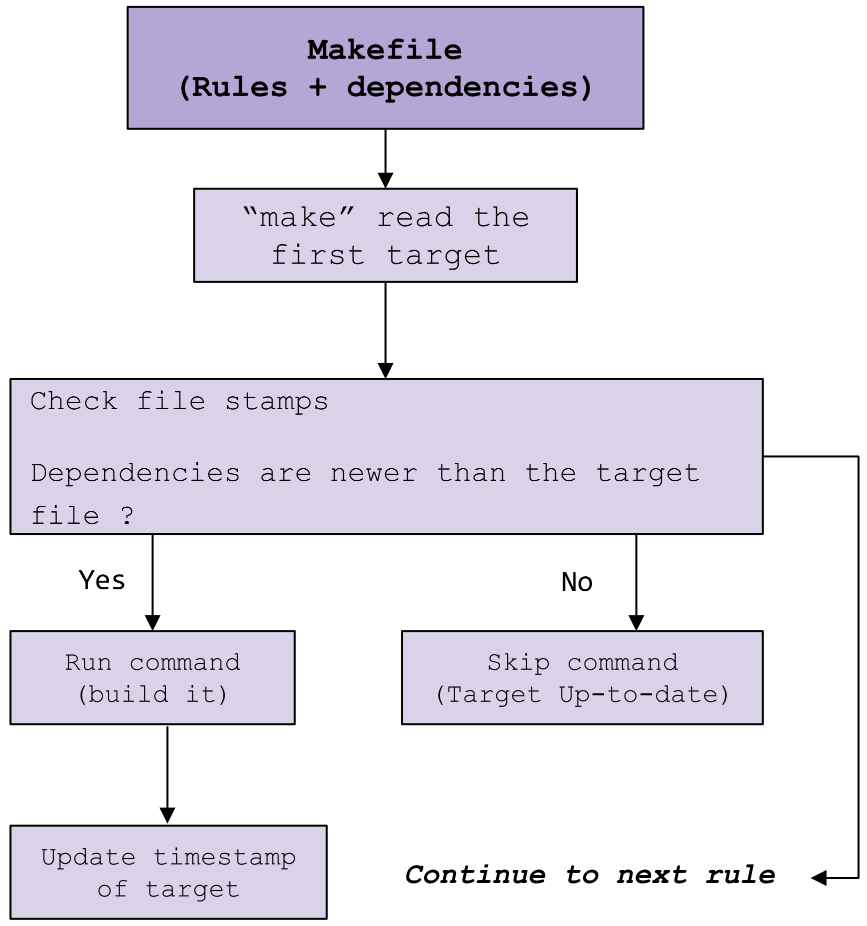
```make
# Link object files together into an target executable
greetings: main.o greet.o
    gcc -o greetings main.o greet.o

# Compile main.c
main.o: main.c greet.h
    gcc -c main.c

# Compile greet.c
greet.o: greet.c greet.h
    gcc -c greet.c
```

How make works

- The **make** utility reads a file named **makefile** in the current directory.
- It looks for the first target to build (unless a specific target is passed).
- The **make** utility checks file timestamps:
  - If the target doesn't exist, or if any dependency is newer than the target, then
  - The command is executed to rebuild the target.
- The **make** utility does this recursively, resolving dependencies in order.
- Only changed files (and those depending on them) are rebuilt.
- Once all updated object files are compiled, it links them into the final executable.


Makefile
(Rules + dependencies)
"make" read the first target
Check file stamps
Dependencies are newer than the target file ?
Yes
No
Run command (build it)
Skip command (Target Up-to-date)
Update timestamp of target
Continue to next rule

Running make on the Command Line

There are multiple ways to run make:

- When run w/o any arguments, make looks for a file named `makefile` or `Makefile` in the current directory and builds the **first target** defined.

```
$ make
```

- If the filename is other than the default, you can specify that filename using –f option

```
$ make -f <filename>
```

- You can tell make to builds only the specified target. If the target's prerequisites have changed since the last build, make will recompile them.

```
$ make <target>
```

```
# File: hello_makefile
hello: hello.o
   gcc hello.o -o hello
hello.o: hello.c
   gcc -c hello.c
```

```
$ make -f hello_makefile
gcc -c hello.c
gcc hello.o -o hello
```

Demonstration

UNIX make utility

Lec1.4/ex1/*

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Multiple Targets of `Makefile`

A Multi-File C Program

```c
//Lec1.4/ex2/mymath.c
double myadd(double, double);
double mysub(double, double);
double mymul(double, double);
double mydiv(double, double);
```

```c
//Lec1.4/ex2/myadd.c
double mysub(double a, double b){
   return a - b;
}
```

```c
//Lec1.4/ex2/mysub.c
double mysub(double a, double b){
   return a - b;
}
```

```c
//Lec1.4/ex2/mymul.c
double mymul(double a, double b){
   return a * b;
}
```

```c
//Lec1.4/ex2/mydiv.c
double mydiv(double a, double b){
        if (b != 0)
                return a / b;
        return 0;
}
```

```c
//Lec1.4/ex2/prog1.c
#include <stdio.h>
#include <mymath.h>
int main(){
   double x, y;
   printf("Enter first number: ");
   scanf("%lf",&x);
   printf("Enter second number: ");
   scanf("%lf",&y);
   double ans1 = myadd(x,y);
   double ans2 = mysub(x,y);
   double ans3 = mymul(x,y);
   double ans4 = mydiv(x,y);
   printf("a + b = %7.2lf\n",ans1);
   printf("a - b = %7.2lf\n",ans2);
   printf("a * b = %7.2lf\n",ans3);
   printf("a/b   = %7.2lf\n",ans4);
   return 0;
}
```

Multiple Targets in `makefile`

- A `makefile` can have multiple targets, with each target representing a specific goal, like building an executable, running the program, cleaning up temporary files, or installing software. By default, make builds **only the first target** it finds.

- **`all`:** Many programmers specify **`all`** as the first target in their `makefile` and then list the other targets as being dependencies for the all target. When you run `make all`, it will build all the targets in the `makefile` and is useful for building the complete software in one go.

```
all: myexe
	@echo "Build complete."
```

- **`install`:** Copies built programs to system directories (usually `/usr/local/bin`). When you run `make install`, it will first build all the mentioned targets (here it is `myexe`), and then will copy the files at required locations.

```
install: myexe
	@echo "Installing myexe..."
	@cp myprogram /usr/local/bin/
	@chmod 755 /usr/local/bin/myexe
	@echo "Installation complete.”
```

Note: The @ symbol in Makefiles prevents make from printing (echoing) the command before executing it.

Multiple Targets in makefile

- **uninstall:** The **uninstall target** is used to remove installed programs from system directories.

```
uninstall:
    @echo "Uninstalling myexe..."
    @rm -f /usr/local/bin/myexe
    @echo "Uninstall complete.”
```

- **clean:** The **clean target** is used to remove temporary or intermediate files like object files and executables to build the target from scratch. You can run it with command: **make clean**, and it executes the **rm** command. This helps maintain a tidy workspace or prepare the project for a fresh build. If there is no .o file in the current working directory, `make` will return an error, so to avoid it we use –`f` option.

```
clean:
    @echo "Cleaning..."
    @rm -f *.o myexe
```

- **distclean:** The **distclean target** is more thorough than clean, as it removes configuration files, backups, documentation, and distribution archives.

```
distclean: clean
    echo "Deep cleaning..."
    rm -f *~
    rm -f *.bak
    rm -f tags
    echo "Deep clean complete."
```

Multiple Targets in `makefile` (cont...)

Phony Targets:

- **The Problem:** The **`make`** utility assumes every target represents a file. If you have a target named `clean` and there's also a file named "`clean`" in your directory, The **`make`** utility checks the file's timestamp. Since the file exists and has no dependencies to compare against, so **`make`** considers the target up-to-date and skips executing the cleaning commands.
- **The Solution:** Phony targets solve this by explicitly telling **`make`** that certain targets don't represent files but rather represent actions or commands you want to execute, declared using the **`.PHONY`** directive as shown:

```
.PHONY: all clean install uninstall distclean
```

- **How It Works:** When a target is declared as phony, `make` treats it as a command rather than a file. This ensures the associated commands always execute, regardless of whether files with matching names exist in the directory.
- **Common Use Cases:** Phony targets are typically used for:
  - Build operations (`all, install`)
  - Clean-up tasks (`clean, distclean`)
  - Testing and validation (`test, check`)
  - Maintenance actions (`uninstall, backup`)

A makefile with Multiple Targets

```
//Lec1.4/ex2/prog1.c
all: myexe install

myexe: mysub.o prog1.o myadd.o mydiv.o mymul.o
	gcc mysub.o prog1.o myadd.o mydiv.o mymul.o -o myexe

myadd.o: myadd.c mymath.h
	gcc -c -I. myadd.c
mysub.o: mysub.c mymath.h
	gcc -c -I. mysub.c
mydiv.o: mydiv.c mymath.h
	gcc -c -I. mydiv.c
mymul.o: mymul.c mymath.h
	gcc -c -I. mymul.c
prog1.o: prog1.c mymath.h
	gcc -c -I. prog1.c
# Utility targets
clean:
	rm -f *.o myexe
install: myexe
	@cp myexe /usr/bin
	@chmod a+x /usr/bin/myexe
	@chmod og-w /usr/bin/myexe
	@echo "myexe successfully installed in /usr/bin"
```

```
uninstall:
	@rm -f /usr/bin/myexe
	@echo "myexe successfully un-installed"
.PHONY: all clean install uninstall
```

Demonstration

Multiple Targets

```
Lec1.4/ex2/*
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Multiple Makefiles in a Project

- Large software projects organize source code into multiple directories (`src/`, `include/`, `lib/`, `tests/`, `modules/`) to maintain logical separation of concerns and improve code maintainability.
- For large projects, a single monolithic `Makefile` becomes error-prone, and difficult to maintain.
- Multiple `makefiles` allow each directory to define its own build rules, compilation flags, and dependencies.
- There are two approaches to implement multiple `makefiles`:

```
ex3/
├── myadd.c
├── mysub.c
├── mydiv.c
├── mymul.c
├── prog1.c
├── mymath.
└── d1/
    ├── mymod.c
    └── mymod.h
```

Recursive Make Approach

- 2 files: `makefile` + `d1/makefile`
- Uses `make -C d1 mymod.o` to build subdir
- `d1/makefile` uses `../mymath.h` to reference parent header
- Separate `clean` targets for each directory
- Each directory manages its own object files

Include Directive Approach

- 2 files: `makefile` + `d1/Rules.mk`
- Uses `include d1/Rules.mk` to include rules
- Main `makefile` uses `d1/mymod.c` and `-Id1` for subdirectory includes
- One `clean` target for everything
- Single `makefile` manages all object files from all directories

Demonstration

Multiple makefiles

Lec1.4/ex3/*

GitHub Code Repository Link: [*https://github.com/arifpucit/OS-Codes*](https://github.com/arifpucit/OS-Codes)

Macros in Makefile

- Macros in **`makefile`** let you define reusable values (like compiler names, flags, or file names) once and reference them wherever needed. We can define macros/variables in a `makefile` as:

  `MACRONAME=value`

- We can access the macros as :

  `$(MACRONAME)`

- In the opposite `makefile`, you can see two macros, one specify the compiler and the other specify the compilation flags. Closely observe the definition of macros and their usage
- This approach makes your `makefile` easier to modify, reuse, and understand.

```
# Define CC and CFLAG variable
CC = gcc
CFLAGS = -std=c11 -O0 -ggdb -Wall


hello: hello.o
  $(CC) $(CFLAGS) hello.o -o hello
hello.o: hello.c
  $(CC) $(CFLAGS) -c hello.c
```

Demonstration

MACROS

Lec1.4/ex4/*

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Automatic Variables in makefile

- Make provides several special internal automatic variables that are automatically set and apply only within the rule where they are used.
- These variables represent different parts of the target and its dependencies, making a `makefile` more flexible and maintainable.

Variable	Description
`$@`	Name of the target of the rule
`$*`	Target name without extension
`$<`	Name of the first dependency/prerequisite of the rule
`$^`	A space separated list of all dependencies, with duplicates removed
`$?`	List of all dependencies that are newer than the target
`$$`	A literal dollar sign, used for escaping in shell commands

```
myexe: main.o utils.o
   echo "Target: $@"                # myexe
   echo "All prerequisites: $^" # main.o utils.o
   echo "Updated prerequisites: $?" #shows only changed .o

main.o: main.c
   echo "Compiling source: $<"  # main.c
   echo "Target object: $@"     # main.o
   echo "Stem name: $*"         # main
```

Demonstration

Auto Variables

```
Lec1.4/ex5/*
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Binary software
vs
Open-source software

Binary Software Packages

- A binary package is a collection of files bundled into a single file containing
  - executable files (compiled for a specific platform),
  - man/info pages,
  - copyright information,
  - configuration and installation scripts.
- It is easy to install a software from its binary package built for your machine and OS, as the dependencies are already resolved.
- For the Debian based distributions (Ubuntu, Kali, Mint, ArchLinux) they come in `.deb` format and the package managers available are `apt`, `dpkg`, `aptitude`, and `synaptic`.
- For RedHat based distributions (Fedora, CentOS, OpenSuse) the packages come in `.rpm` format and the available package managers are `rpm` and `yum`.
- For Mac OS the packages come in `.dmg` format and the available package managers are `brew` and `fink`.

Open-Source Software Packages

- An Open-source software is a software with its source code made available with a license in which the copyright holder provides the rights to study, change, and distribute the software to anyone and for any purpose (GNU GPL). Normally distributed as a `tarball` containing:
  - Source code files
  - README and INSTALL
  - AUTHORS
  - Configure script
  - Makefile.am and Makefile.in

- A source package is eventually converted into a binary package for a platform on which it is configured, build and installed. We normally use source packages to install software for following reasons:
  - We cannot find a corresponding binary package.
  - We want to enhance functionalities of a software.
  - We want to fix a bug in a software.

Example: Installing Open-Source Software

- Download a basic hello-world C project from following GitHub repository:

```
$ git clone https://github.com/irvanherz/hello-world-autotools-template
$ cd hello-world-autotools-template
```

- To convert this source package into a binary package and install it on our Linux machine, we need to recite the following magic spell:

```
$ autoreconf --install
$ ./configure && make && sudo make install
```

- Once installed you can run it using following command:

```
$ hello
Hello world!
```

- The `autoreconf` runs the `configure.ac` and generate the `configure` script, `Makefile.in` and macros in `m4` directory.
- The `configure` script checks your system for required tools/libraries and generates a system-specific `Makefile` from `Makefile.in`
- Finally the `make install` copies the compiled program and its files to system directories (e.g., `/usr/local/bin`) so it can be run globally.

```
hello-world/
├── configure.ac
├── Makefile.am
├── README.md
└── src/
    ├── main.c
    └── Makefile.am
```

Demonstration

Installing Open Source Softwares

```
Lec1.4/opensource/
hello-world-autotools-template
```

`cmake` is a cross platform **Makefile generator**. It is an effort to develop a better way to configure build and deploy complex softwares written in various languages, across many different platforms

To Do

- Watch SP video lecture on **`make`** utility:
  https://youtu.be/8hG0MTyyxMI?si=_9vtEEKQyJHk_bob
- Watch SP video lecture on **`autotools`** and **`cmake`**
  https://youtu.be/Ncb_xzjGAwM?si=fXBuGkPWJfHDsl7y

Coming to office hours does NOT mean that you are academically weak!