

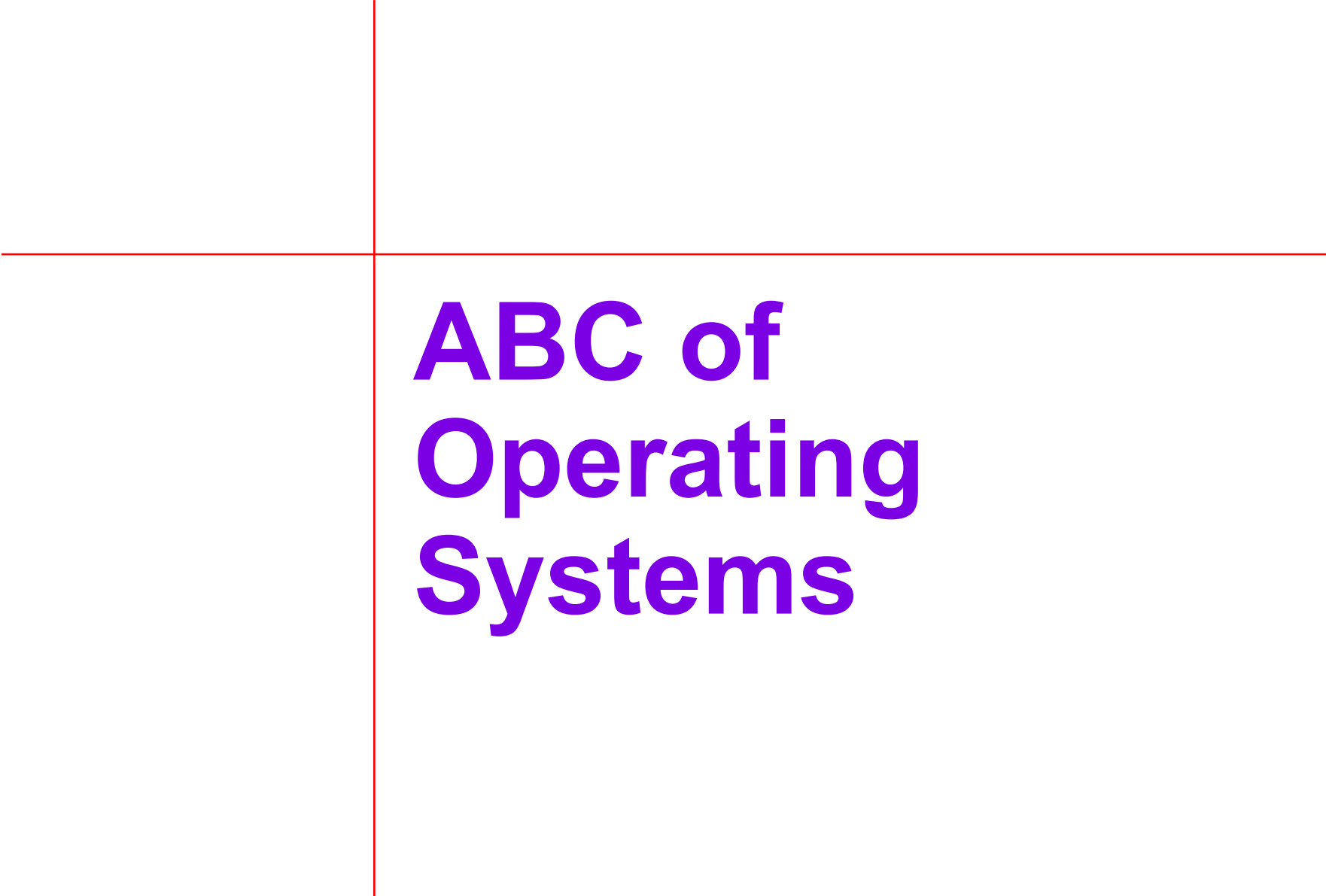
Introduction to Computers and Programming

Lecture 9 –
OS, library and gcc gdb
commands

Tien-Fu Chen

Dept. of Computer Science and
Information Engineering

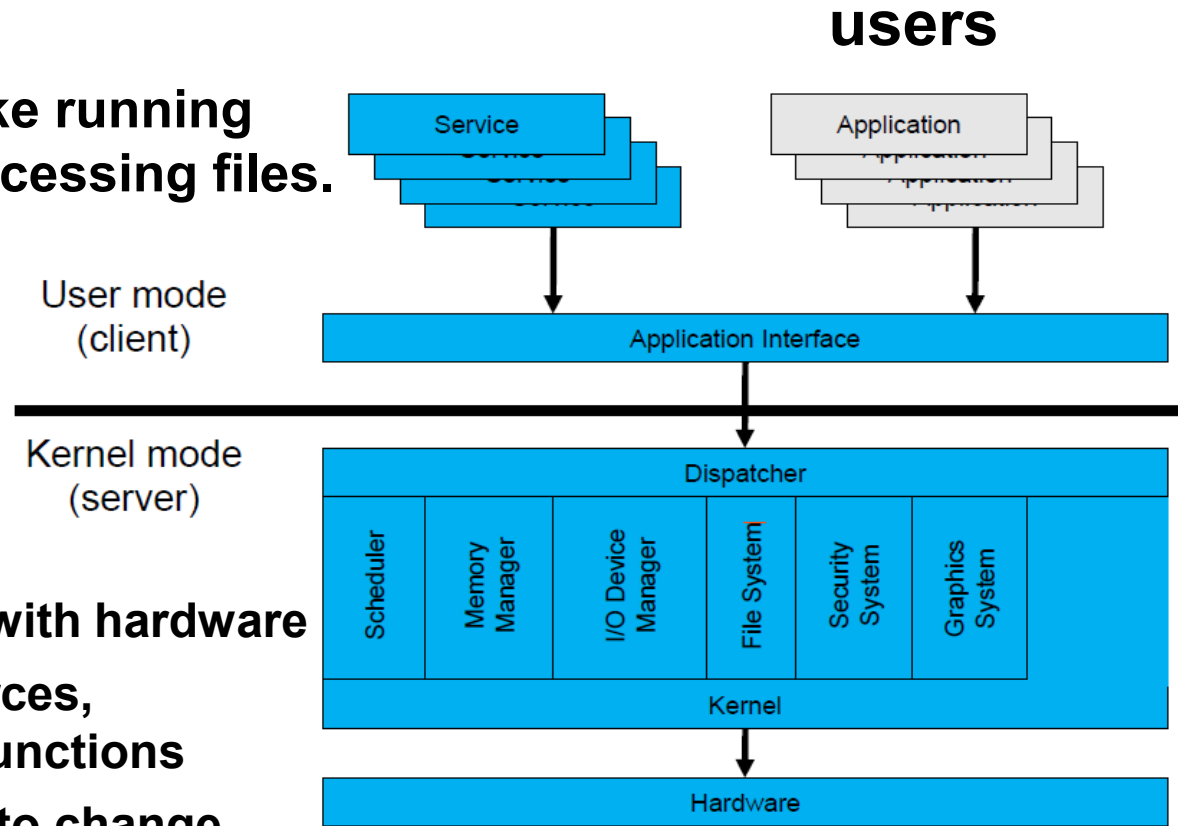
National Yang Ming Chiao Tung Univ.



ABC of Operating Systems

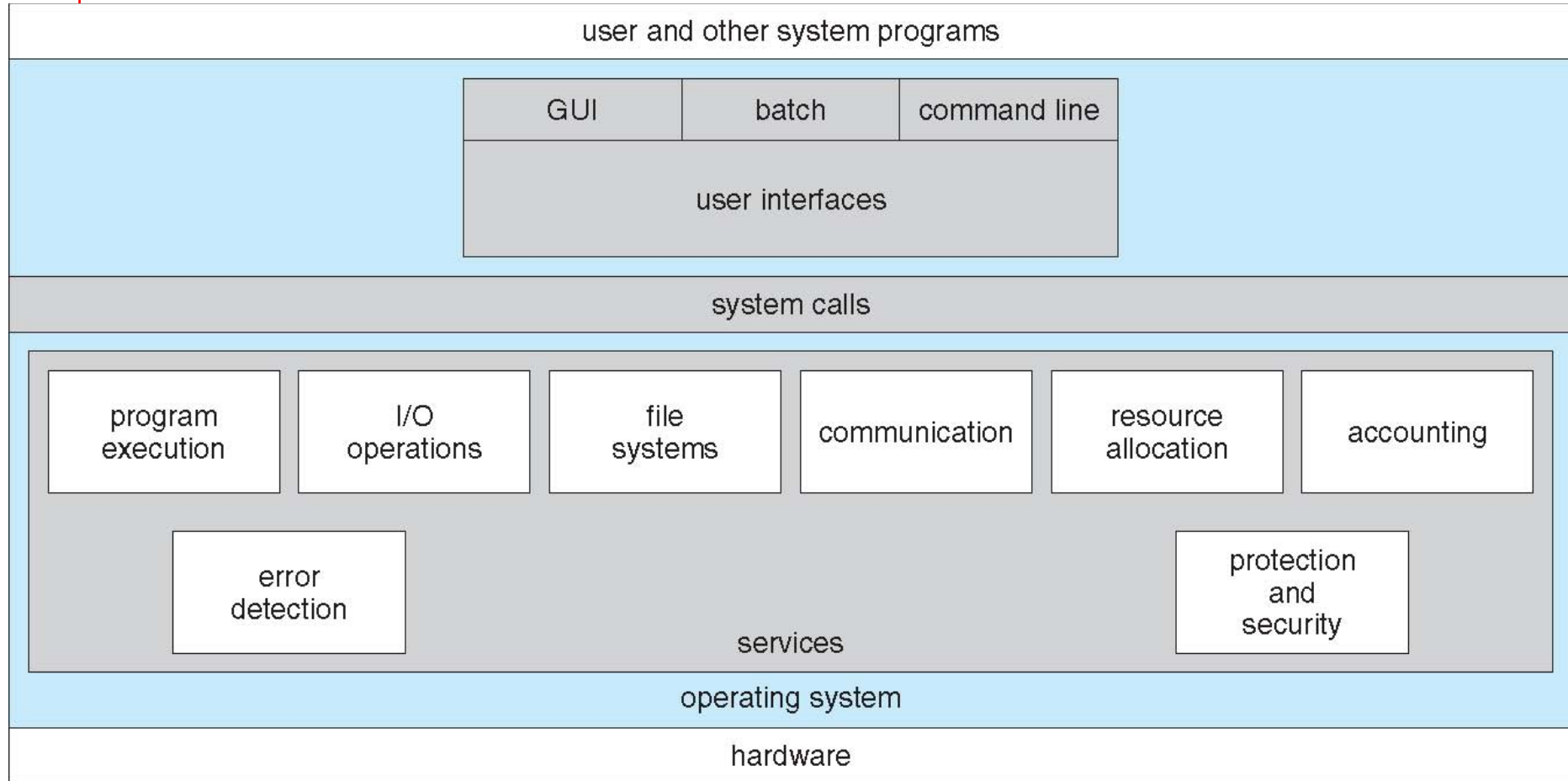
Structure of Operating System

- ❑ **User Mode:** the actual interface between the user and the system.
- ❑ It controls things like running applications and accessing files.



- ❑ **Kernel Mode:** interact with hardware
- ❑ Manage system resources, controlling hardware functions
- ❑ System calls are used to change mode from User to Kernel.

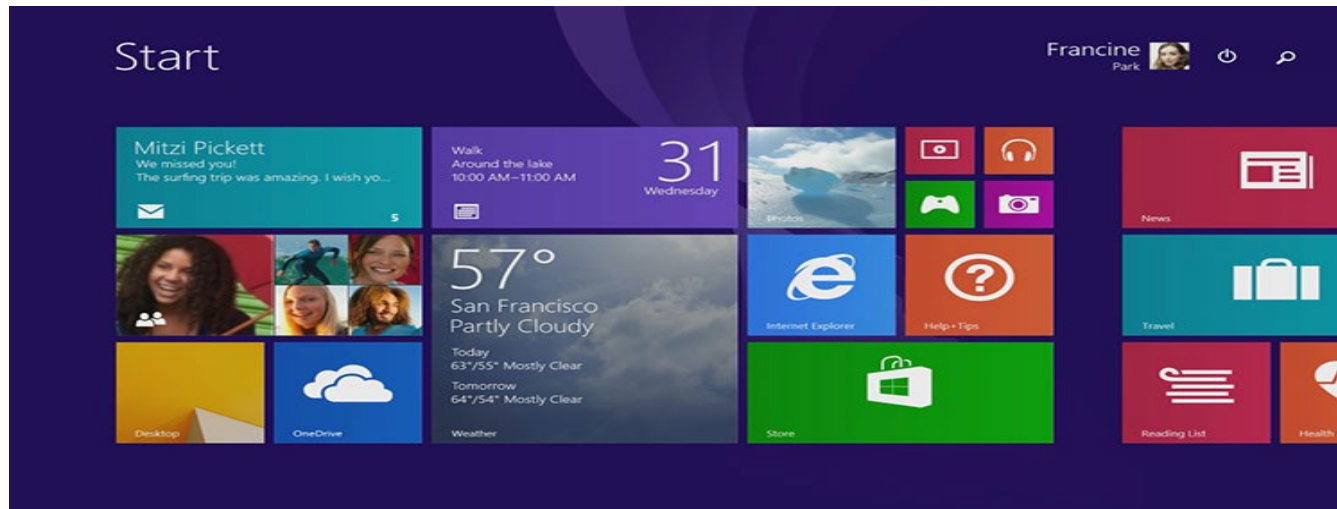
A View of Operating System Services



Microsoft Windows

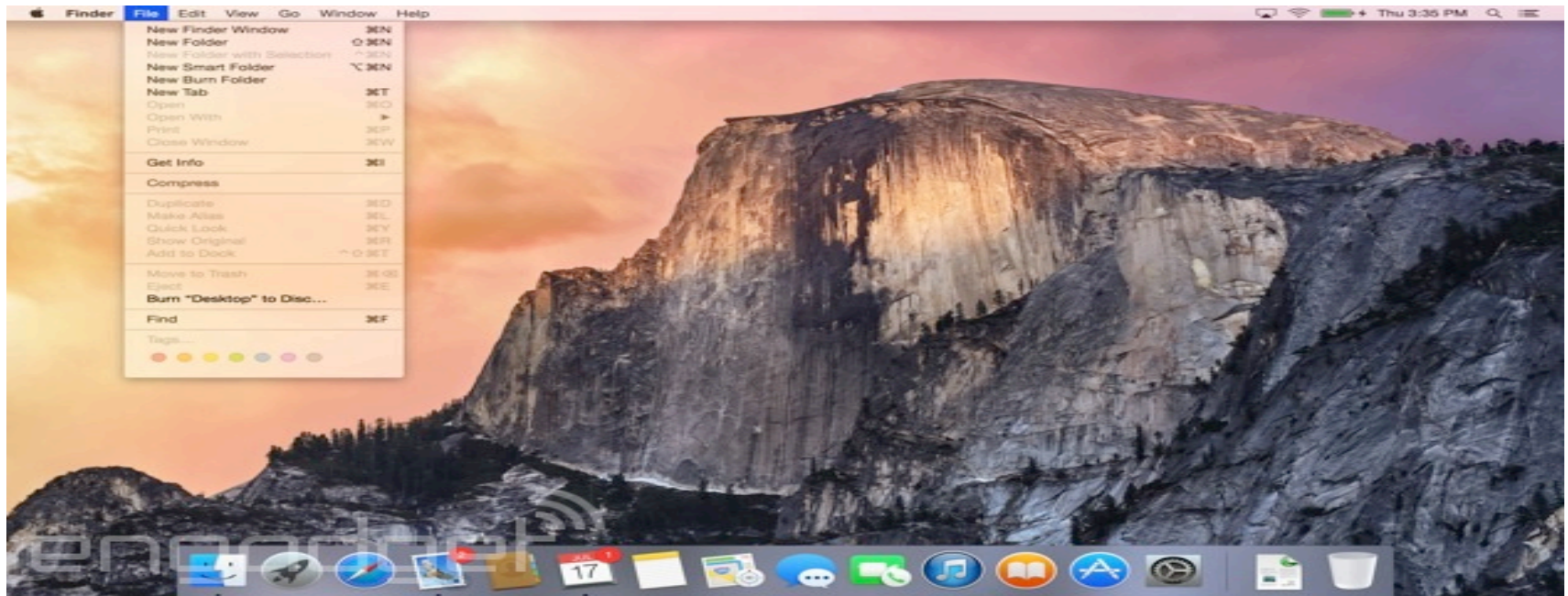


- ❑ The graphical Microsoft operating system designed for Intel-platform desktop and notebook computers.
- ❑ Best known, greatest selection of applications available.
- ❑ Current editions include Windows 7, 8, 8.1 and 10.

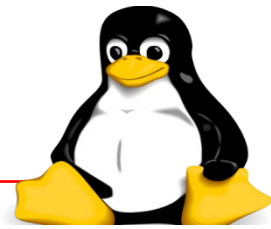


Mac OS

- ❑ User-friendly, runs on Mac hardware. Many applications available.
- ❑ Current editions include: Sierra, High Sierra, Mojave, Catalina & Big Sur—Version XI (Released in Nov 2020)



Linux



- ❑ **Linux:** An open-source, cross-platform OS that runs on desktops, notebooks, tablets, and smartphones.
 - The name *Linux* is a combination *Linus* (the first name of the first developer) and *UNIX*.
- ❑ Linux is more stable than other operating systems.
- ❑ Users are free to modify, improve, and redistribute code
- ❑ Developers are not allowed to charge money for Linux kernel, but they can do for **distributions (distros)**.
- ❑ Some popular distros are **Ubuntu, Fedora, Debian, Linux Mint, Arch Linux,**
- ❑ Debian and Fedora would be good choices for proficient programmers.

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Example of Standard API (library)

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t  read(int fd, void *buf, size_t count)
```

ssize_t	read	(int fd, void *buf, size_t count)
return value	function name	parameters

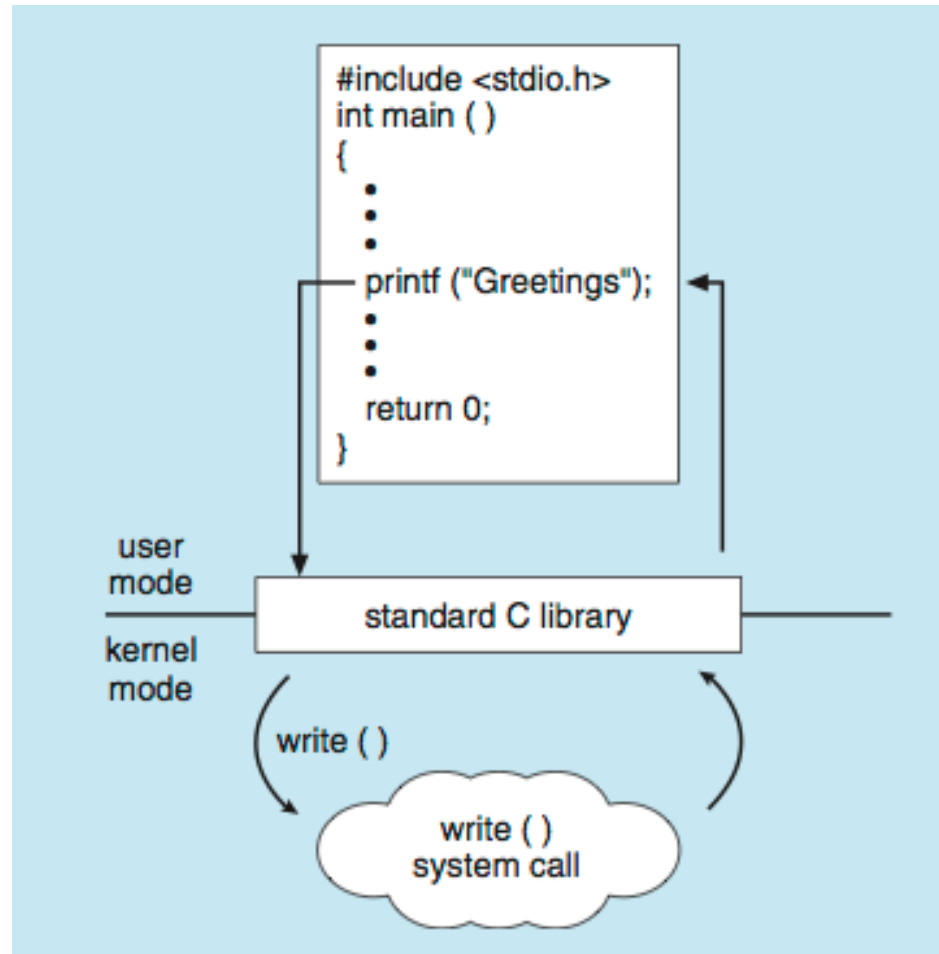
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

Invoke C Library Example

- ❑ C program invoking printf() library call, which calls write() system call





Linux commands and Shell command line

Shell keys and arguments

- ❑ **shell**: An interactive program that uses user input to manage the execution of other programs.
 - bash : the default shell program on most Linux/Unix systems
 - ❑ **Command-line arguments**
 - most options are a - followed by a letter such as -c
 - some are longer words preceded by two - signs, such as --count
 - ❑ **parameters can be combined**: ls -l -a -r can be ls -lar
-
- ~ “tilde” indicates your home directory: /home/you
 - * “star”: wildcard, matches anything
 - ? wildcard, matches any one character
 - ! History substitution, do not use
 - & run a job in the background, or redirect errors

Shell commands

- ❑ **shell**: An interactive program that uses user input to manage the execution of other programs.
 - `bash` : the default shell program on most Linux/Unix systems

```
$ pwd
/homes/iws/dravir
$ cd data
$ ls
file1.txt file2.txt
$ ls -l
-rw-r--r-- 1 dravir vgrad_cs 0 2010-03-29 17:45 file1.txt
-rw-r--r-- 1 dravir vgrad_cs 0 2010-03-29 17:45 file2.txt
$ cd ..
$ man ls
$ exit
```

command	description
<code>exit</code>	logs out of the shell
<code>ls</code>	lists files in a directory
<code>pwd</code>	outputs the current working directory
<code>cd</code>	changes the working directory
<code>man</code>	brings up the manual for a command

Basic unix commands

command	description
ls	list files in a directory
pwd	output current working directory
cd	change the working directory
mkdir	create a new directory
rmdir	delete a directory (must be empty)

command	description
man	get help on a command
clear	clears out output from console
exit	exits and logs out of the shell

directory	description
.	the directory you are in ("working directory")
..	the parent of the working directory (../.. is grandparent, etc.)
~	your home directory (on many systems, this is /home/ <i>username</i>)
~ <i>username</i>	<i>username</i> 's home directory
~/Desktop	your desktop

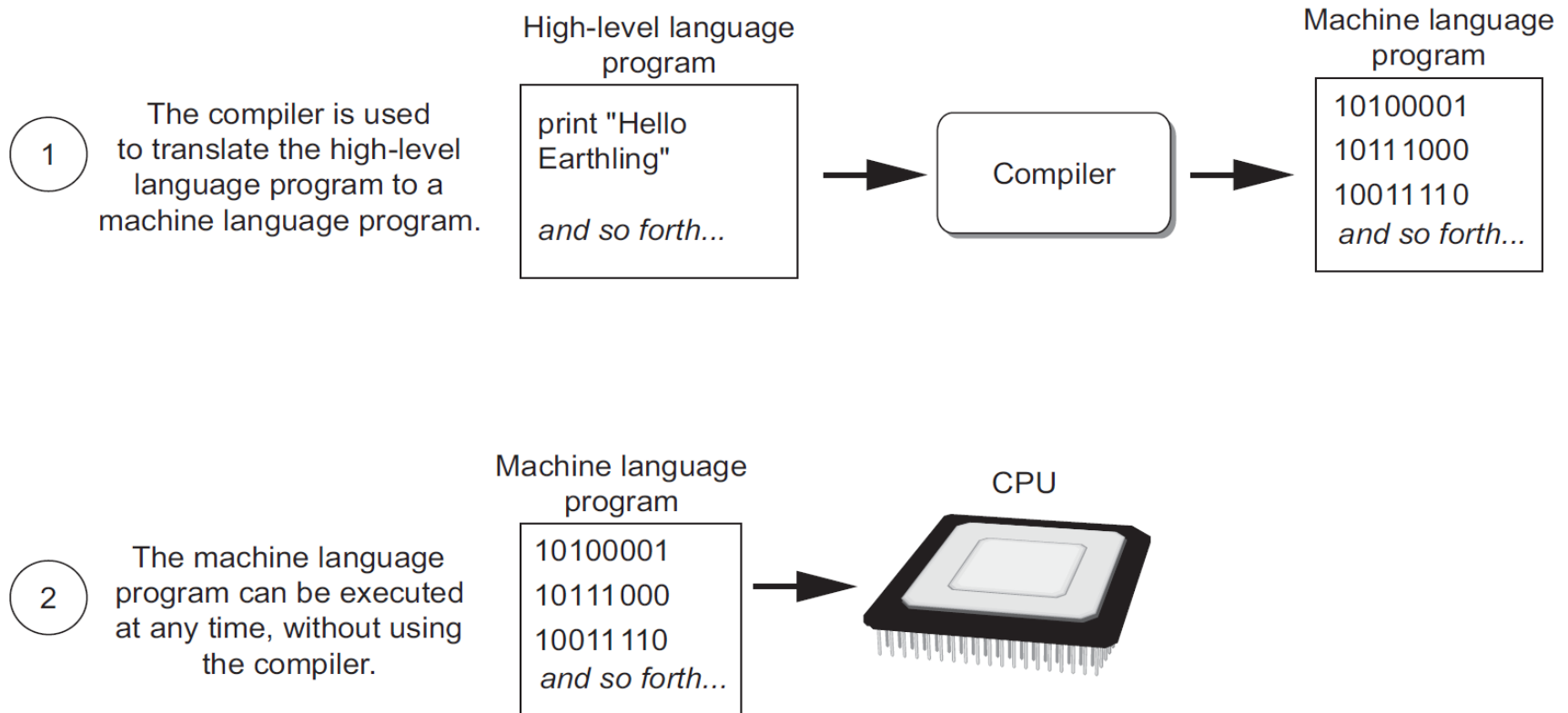
command	description
cp	copy a file
mv	move or rename a file
rm	delete a file
touch	create a new empty file, or update last-modified time stamp



Compile, Link and make

How program is generated – by compiler

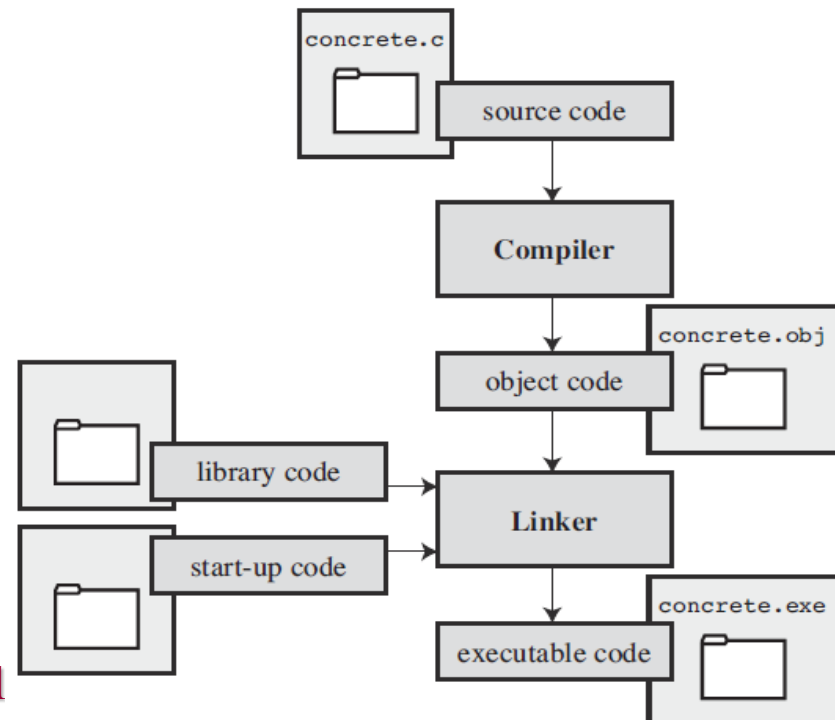
- ❑ A compiler is a program that translates a high-level language program into a separate machine language program.



Compiling and Linking

- ❑ Three steps to generate programs:
 - **Preprocessing.** The **preprocessor** obeys commands that begin with # (known as **directives**)
 - **Compiling.** A **compiler** translates then translates the program into machine instructions (**object code**).
 - **Linking.** A **linker** combines the object code produced by the compiler with any additional code needed to yield a complete executable program.

The preprocessor is usually integrated with the compiler.

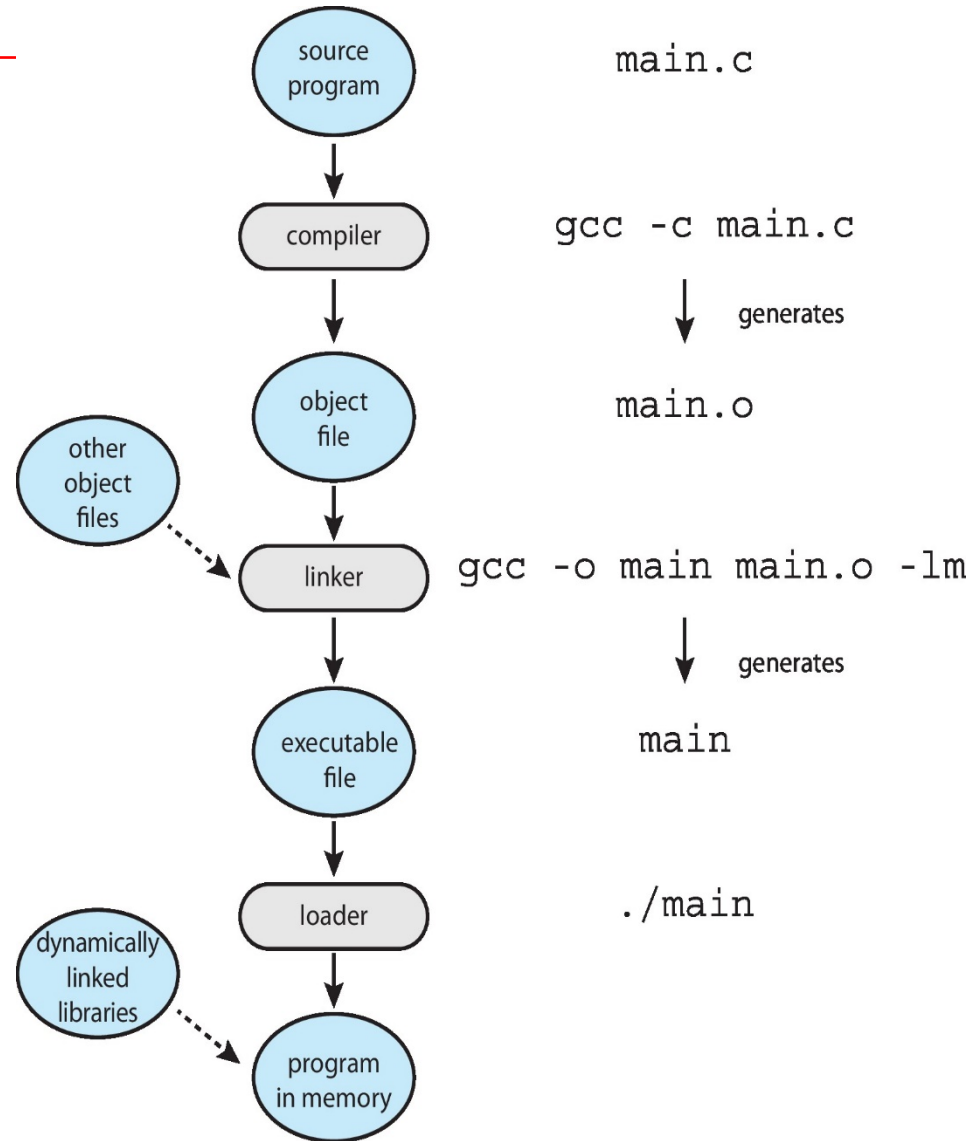


What is gcc?

❑ gcc

- GNU C/C++ Compiler
- Console-based compiler for Unix/Linux based platforms and others; can cross-compile code for various architectures
- **gcc** for C; **g++** for C++
- **gcc** performs all of these:
 - ❑ preprocessing
 - ❑ compilation
 - ❑ assembly and
 - ❑ linking

❑ As always: \$ **man gcc**



Compiling: C

command	description
gcc	GNU C compiler

- ❑ To **compile** a C program called *source.c*, type:

`gcc -o target source.c` $\xrightarrow{\text{produces}}$ `target`

(where **target** is the name of the executable program to build)

- the compiler builds an actual *executable file*

- Example: `gcc -o hi hello.c`

Compiles the file `hello.c` into an executable called “hi”

- ❑ To **run** your program, just execute that file:

- Example: `./hi`

Object files (.o)

- A .c file can also be **compiled** into an *object (.o) file* with **-c**

```
$ gcc -c part1.c           →      part1.o
                             produces
$ ls
part1.c  part1.o  part2.c
```

- a .o file is a binary “blob” of compiled C code that cannot be directly executed, but can be directly **linked** into a larger *executable* later

- You can **compile** and **link** a mixture of .c and .o files:

```
$ gcc -o myProgram part1.o part2.c → myProgram
                                   produces
```

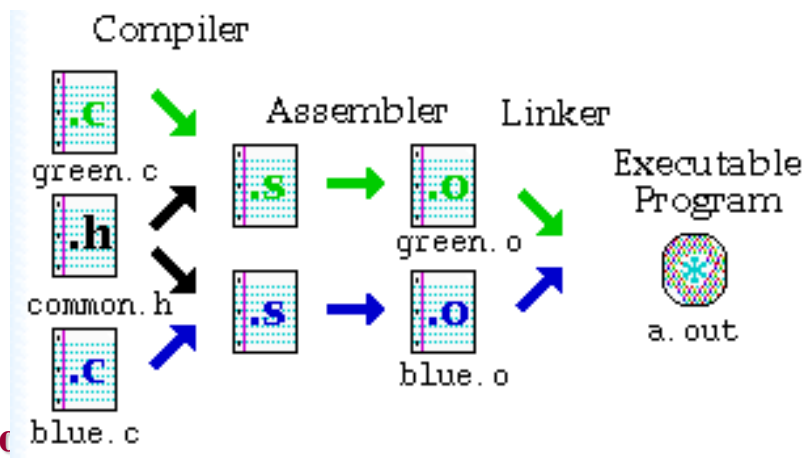
Avoids recompilation of unchanged partial program files (e.g. **part1.o**)

Building a Multiple-File Program

- ❑ A GCC command that builds `justify`:

```
gcc -o justify justify.c line.c word.c
```

- ❑ The three source files are first compiled into object code.
- ❑ The object files are then automatically passed to the linker, which combines them into a single file.
- ❑ The `-o` option specifies that we want the executable file to be named `justify`.



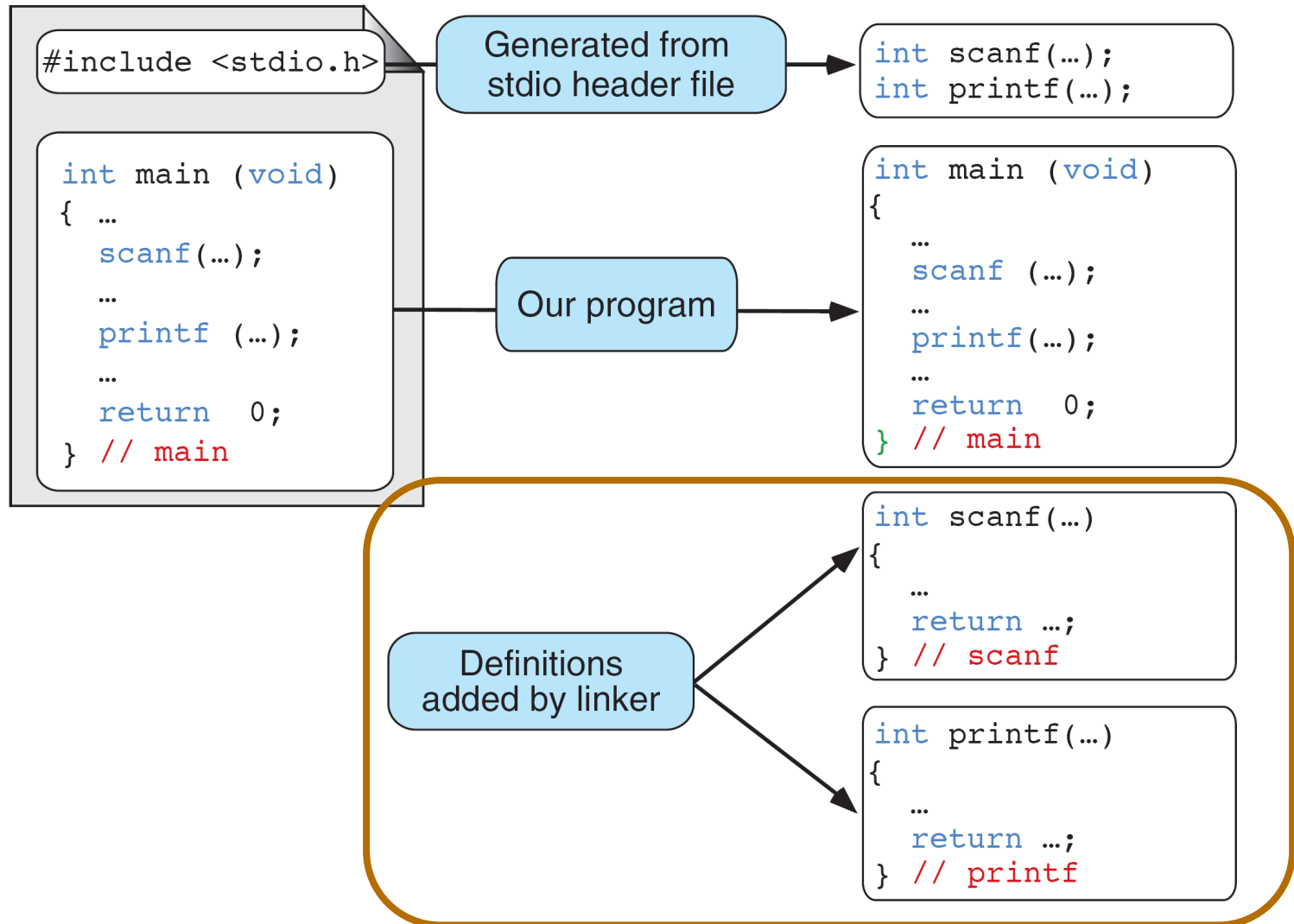
gcc Options

□ most often used options:

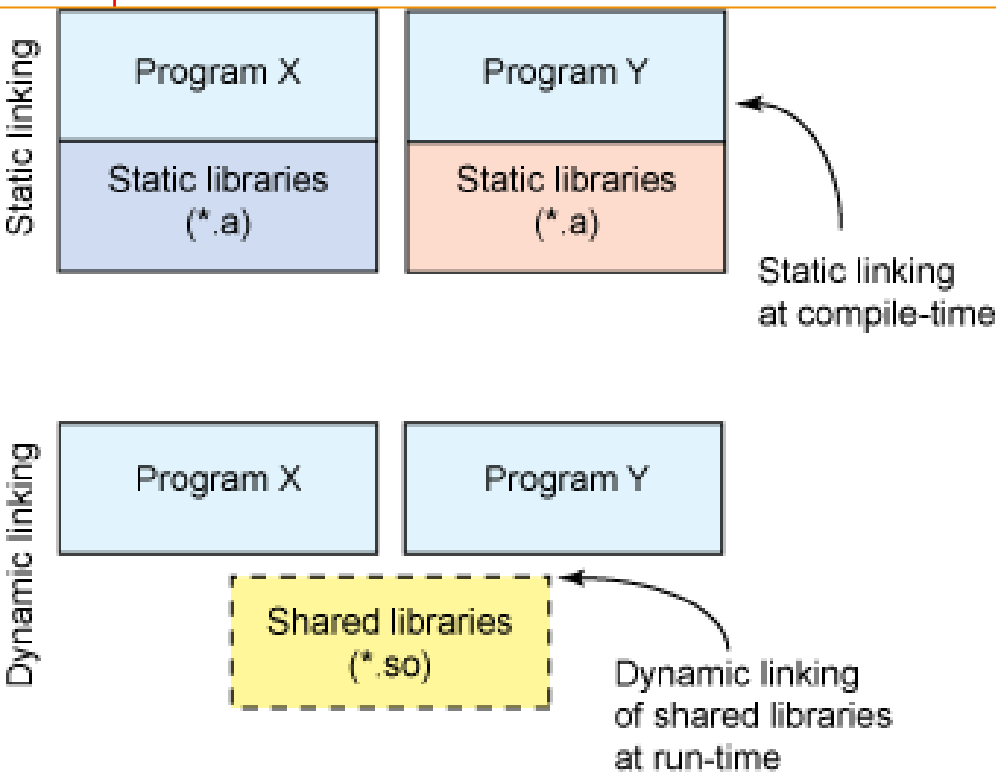
- To compile: **-c**
- Specify output filename: **-o <filename>**
- Include debugging symbols: **-g**
- GDB friendly output: **-ggdb**
- Show all (most) warnings: **-Wall**
- Be stubborn about standards: **-ansi** and **-pedantic**
- Optimizations: **-O***, -O1, -O2, -O3,

```
gcc -g -Wall -ansi -c process.c
```

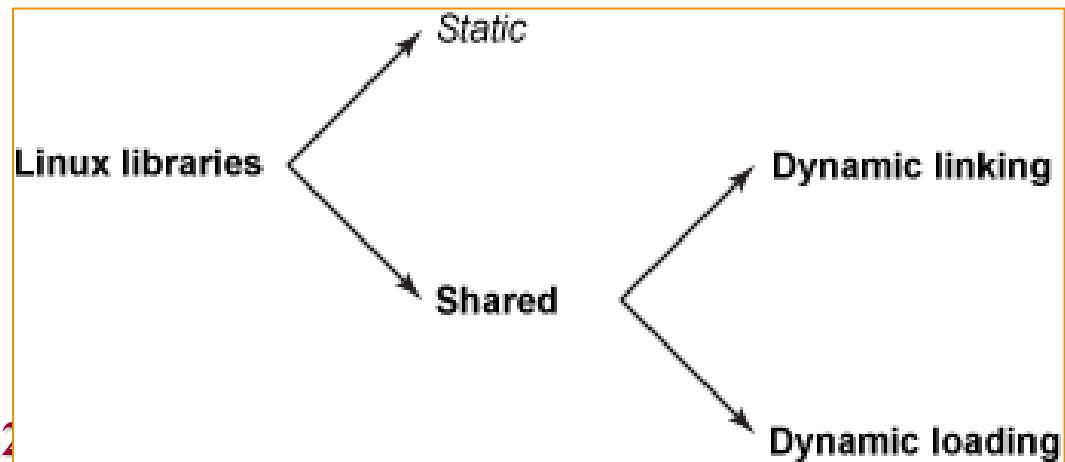

Library Functions and the Linker



Dynamic linking within Linux



[ref: Anatomy of Linux dynamic libraries](#)



```

/* main.c */
#include <string.h>

void func_1(...);
...
void func_n(...);

int main(...) {
    ...
}

void func_1(...) {
    ...
}

...
void func_n(...) {
    ...
}

```

```

/* func.c */
#include <string.h>
void func_1(...);
...
void func_n(...);

void func_1(...) {
    ...
}

...
void func_n(...) {
    ...
}

```

```

/* main.c */
#include <string.h>
void func_1(...);
...
void func_n(...);

int main(...) {
    ...
}

```

```

/* func.h */
void func_1(...);
...
void func_n(...);

```

```

/* func.c */
#include <string.h>
#include "func.h"

void func_1(...) {
    ...
}

...
void func_n(...) {
    ...
}

```

```

/* main.c */
#include <string.h>
#include "func.h"

int main(...) {
    ...
}

```

gdb

- ❑ GDB is invoked with the shell command **gdb**.
- ❑ Once started, it reads commands from the terminal until you tell it to exit with the GDB command **quit**.
 - The most usual way to start GDB is with one argument or two, specifying an executable program as the argument:
obelix[4] > gdb program
 - You can also start with both an executable program and a core file specified:
obelix[5] > gdb program core
 - You can, instead, specify a process ID as a second argument, if you want to debug a running process:
obelix[6] > gdb program 1234
would attach GDB to process **1234**

Common Commands for gdb

<code>b(reak) [file:]function</code>	Set a breakpoint at function (in file).
<code>r(un) [arglist]</code>	Start program (with arglist, if specified).
<code>bt</code> or <code>where</code>	Backtrace: display the program stack; especially useful to find where your program crashed or dumped core.
<code>print expr</code>	Display the value of an expression.
<code>C (ontinue)</code>	Continue running your program (after stopping, e.g. at a breakpoint).
<code>n(ext)</code>	Execute next program line (after stopping); step over any function calls in the line.
<code>s(tep)</code>	Execute next program line (after stopping); step into any function calls in the line.
<code>help [name]</code>	Show information about GDB command name, or general information about using GDB.
<code>up/down</code>	traverse function call.
<code>l(ist)</code>	print the source code

How useful are debuggers?

- ❑ Debuggers can be great for seeing how small programs execute
- ❑ Great for certain types of problems
 - Identifying the line on which the program crashes
 - Seeing state of procedure stack at crash
- ❑ Less useful for non-crashing programs
- ❑ Disadvantages of debuggers
 - Not available on some systems
 - System-dependent user interface
 - Too many low-level details
 - ❑ Try debugging linked lists
 - Clicking over statements is tedious
 - Deal poorly with large amounts of data
 - Difficult to find intermittent bugs
- ❑ Debuggers are an important tool, but not the only, or even most important one for debugging programs

Difficult Bugs

❑ Make the bug reproducible

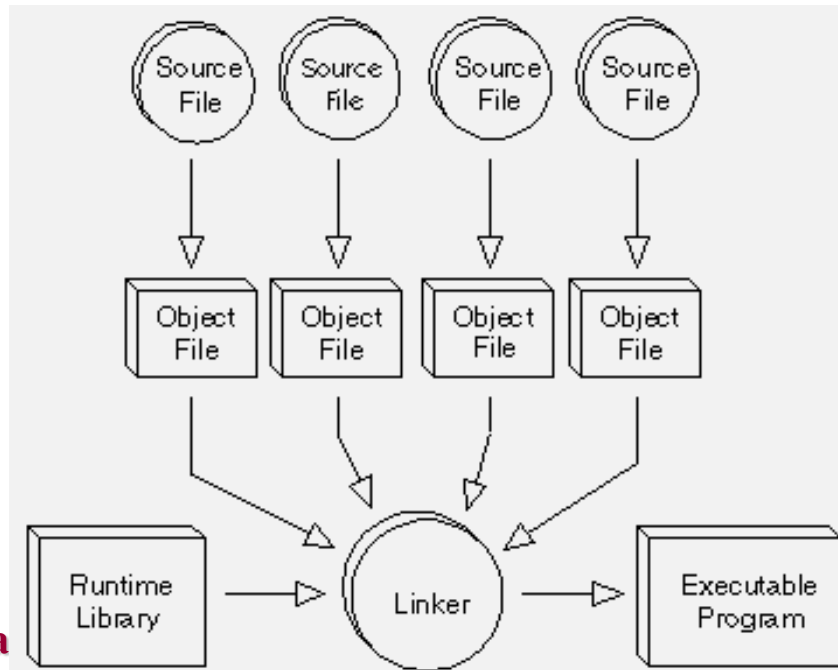
- Most difficult bugs to fix are those that are difficult to repeat
 - ❑ Big reason why concurrent programming is so difficult
- Find input and parameter settings that cause the bug to appear every time
- Will need to reproduce bug again and again
- If bug cannot be made reproducible, that is also information

❑ Divide and conquer

- Reduce the size of the input required to trigger bug
- Make test case as small as possible
- Use binary search
 - ❑ Throw away half the input
 - ❑ See which half of the input causes the bug
 - ❑ Repeat

Building a Multiple-File Program

- ❑ Each source file must be compiled separately.
- ❑ Header files don't need to be compiled.
- ❑ The contents of a header file are automatically compiled whenever it is included.
- ❑ For each source, the compiler generates its object code, having the extension `.o` in UNIX and `.obj` in Windows.



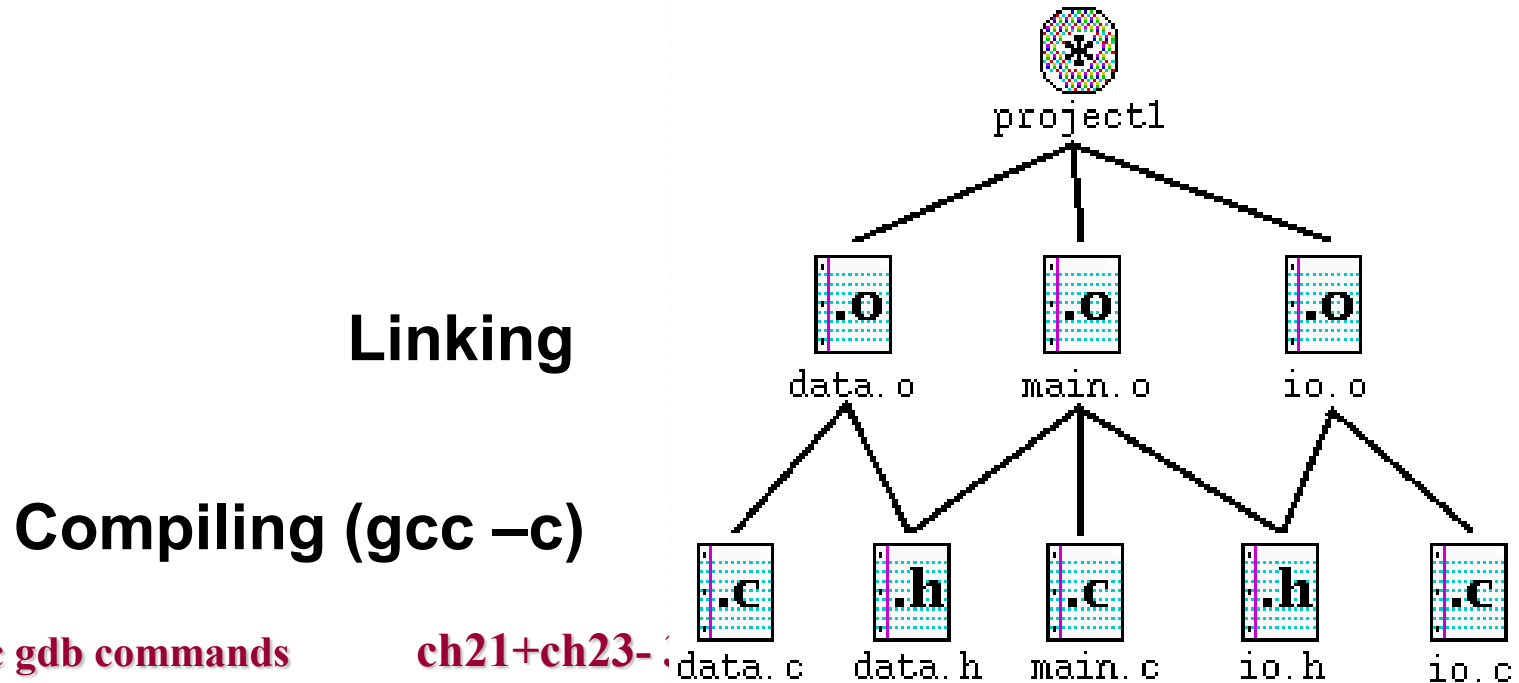
make

- ❑ **make** : A utility for automatically compiling ("building") executables and libraries from source code.
 - a very basic compilation manager
 - often used for C programs, but not language-specific
 - primitive, but still widely used due to familiarity, simplicity
 - similar programs: ant, maven, IDEs (Eclipse), ...

- ❑ **Makefile** : A script file that defines rules for what must be compiled and how to compile it.
 - Makefiles describe which files depend on which others, and how to create / compile / build / update each file in the system as needed.

Dependencies

- ❑ **dependency** : When a file relies on the contents of another.
 - can be displayed as a *dependency graph*
 - to build `main.o`, we need `data.h`, `main.c`, and `io.h`
 - if any of those files is updated, we must rebuild `main.o`
 - if `main.o` is updated, we must update `project1`



Makefile

- ❑ A makefile not only lists the files that are part of the program, but also describes ***dependencies*** among the files.
- ❑ the file `foo.c` includes the file `bar.h`.
 - `foo.c` “depends” on `bar.h`, because a change to `bar.h` will require us to recompile `foo.c`.

target

```
justify: justify.o word.o line.o
        gcc -o justify justify.o word.o line.o
```

```
justify.o: justify.c word.h line.h
        gcc -c justify.c
```

rule

```
word.o: word.c word.h
        gcc -c word.c
```

command

```
line.o: line.c line.h
        gcc -c line.c
```

Running make

\$ make *target*

- uses the file named Makefile in current directory
- Finds a rule in Makefile for building ***target*** and follows it
 - ❑ if the ***target*** file does not exist, or if it is older than any of its *sources*, its *commands* will be executed

❑ variations:

\$ make

- builds the ***first*** target in the Makefile by default

\$ make -f *makefilename*

\$ make -f *makefilename target*

- uses a makefile other than Makefile

How does make process rules?

```
myprogram : file1.c file2.c file3.c
```

```
gcc -o myprogram file1.c file2.c file3.c
```

- if myprogram **does not exist**, then it will execute command:

```
gcc -o myprogram file1.c file2.c file3.c
```

(Which will create a file called “myprogram”)

- if myprogram **does exist**, its timestamp will be compared to the timestamps on file1.c, file2.c, and file3.c
 - If any sources have been modified since myprogram was created, the gcc command will be run to update myprogram.
 - If any of the sources were the target of another rule, the timestamps on *its* dependencies would be checked recursively and rebuilt before rebuilding myprogram

More Makefile - variables

```
# Simple Makefile with use of gcc could

CC=gcc
CFLAGS=-g -Wall -ansi -pedantic
OBJ:=ccountln.o parser.o process.o fileops.o
EXE=ccountln

all: $(EXE)

$(EXE): $(OBJ)
        $(CC) $(OBJ) -o $(EXE)

ccountln.o: ccountln.h ccountln.c
        $(CC) $(CFLAGS) -c ccountln.c

...
```


Special variables

<code>\$@</code>	the current target file
<code>\$\$</code>	all sources listed for the current target
<code>\$\$</code>	the first (left-most) source for the current target

Example Makefile:

```
myprog: file1.o file2.o file3.o
      gcc $(CFLAGS) -o $$ $^
```

```
file1.o: file1.c file1.h file2.h
      gcc $(CFLAGS) -c $$
```

* http://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables

Pattern Rules

- ❑ Rather than specifying individually how to convert every .c file into its corresponding .o file,

```
# conversion from .c to .o
```

```
%.o: %.c
```

```
$(CC) -c $(CFLAGS) $< -o $@
```

- “To create filename.o from filename.c, run `gcc -c -g -Wall filename.c -o filename.o`”
- The rule listed above is actually pre-defined in make, so you do not need to include it in your makefile.

```
main.o: io.h data.h
```

```
data.o: data.h
```

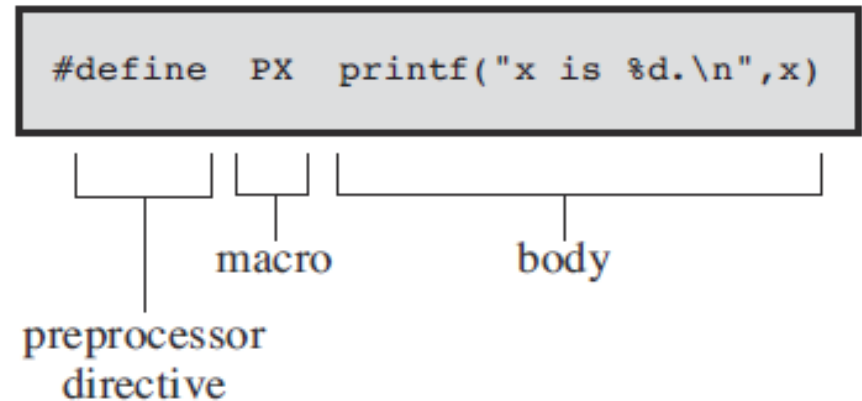
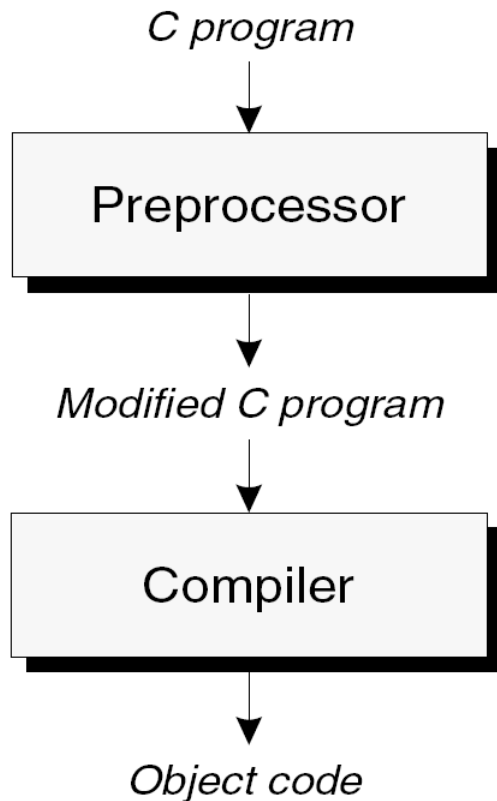
https://www.gnu.org/software/make/manual/html_node/make-Deduces.html#make-Deduces



Library for C

How the Preprocessor Works

- ❑ The preprocessor's role in the compilation process:



The #include Directive

- ❑ The #include directive has two primary forms.
- ❑ The first is used for header files that belong to C's own library:

```
#include <filename>
```

- ❑ The second is used for all other header files:

```
#include "filename"
```

- ❑ The difference between the two has to do with how the compiler locates the header file.

Standard Library

`<stddef.h>` *Common Definitions*

Provides definitions of frequently used types and macros.

`<stdio.h>` *Input/Output*

Provides a large input/output functions, including operations on both sequential and random-access files.

`<stdlib.h>` *General Utilities*

Provides functions that perform the following operations:

- Converting strings to numbers
- Generating pseudo-random numbers
- Performing memory management tasks
- Communicating with the operating system
- Searching and sorting
- Performing conversions between multibyte characters and wide characters

Standard Library

`<string.h>` *String Handling*

Provides functions that perform string operations, as well as functions that operate on arbitrary blocks of memory.

`<time.h>` *Date and Time*

Provides functions for determining the time (and date), manipulating times, and formatting times for display.

`<math.h>` *Mathematics*

Provides common mathematical functions.

`<assert.h>` *Diagnostics*

Contains only the `assert` macro

Inline functions

- ❑ An inline function replaces a function call with inline code
 - ❑ Calls to the function be as fast as possible and perform some other sorts of optimizations
 - ❑ An inline function should be short.
- ```
#include <stdio.h>
inline static void eatline()
{
 while (getchar() != '\n')
 continue;
}
int main()
{
 ...
 eatline(); // function call
 ...
}
```