# Operating system structures

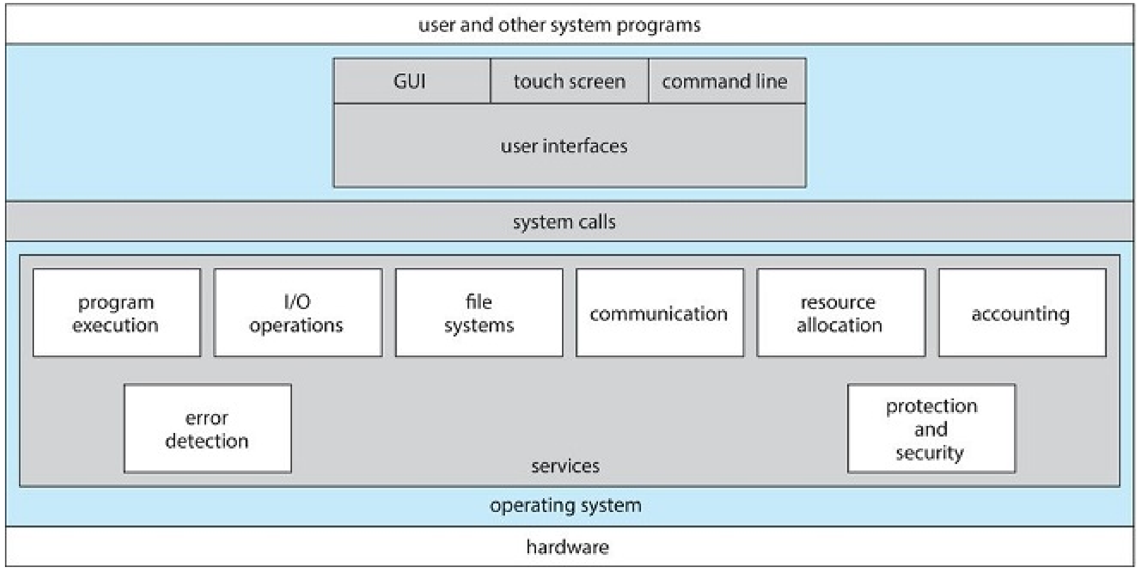
Chapter 2

# 2.1 Operating system services: Multi-view perspective

## Internal structure vs external interface

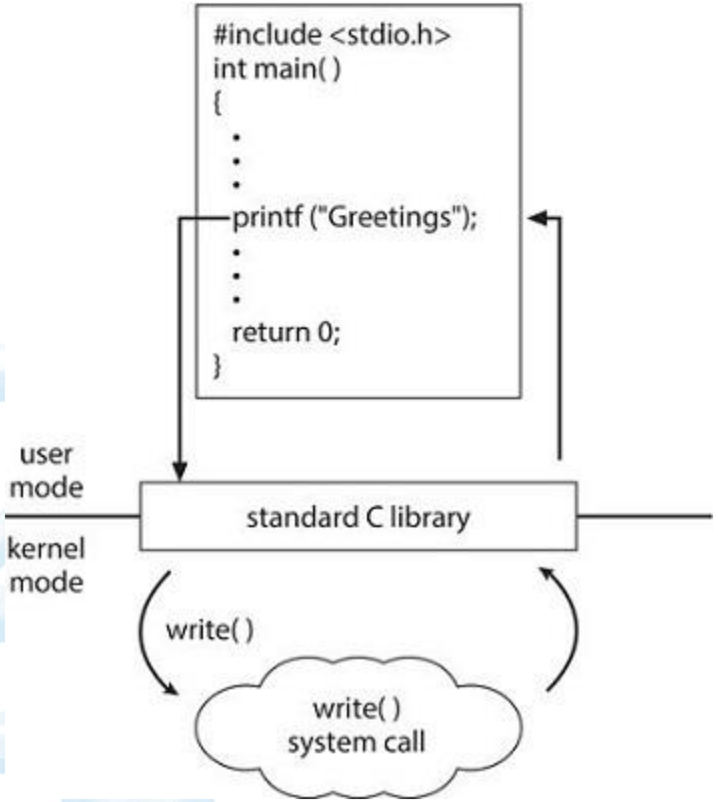
- **View 1: services provided by system**
  - For applications and users
- **View 2: interface towards**
  - Users and programmers
  - Users: Gui or CLI (e.g.: cp f1.txt f2.txt)
  - Apps & programmers: API / system calls
- **View 3: components and interconnections of the OS**
  - Internal architecture/structure
  - Monolytics, layered, microkernel…

## Operating system services



# Example: printf(“Greetings”)
# From API call to system call = 3 layers

## API call: printf() in stdio.h




## System call: write() → syscall()

- Write() is still a lower-level api call
- Syscall() is actual system call
- 1 = STDOUT_FILENO

```c
#include <unistd.h>
int main(void) {
  write(1, "hello, world!\n", 14);
  return 0;
}
```

```c
#include <unistd.h>
#include <sys/syscall.h>
int main(void) {
  syscall(SYS_write, 1, "hello, world!\n", 14);
  return 0;
}
```

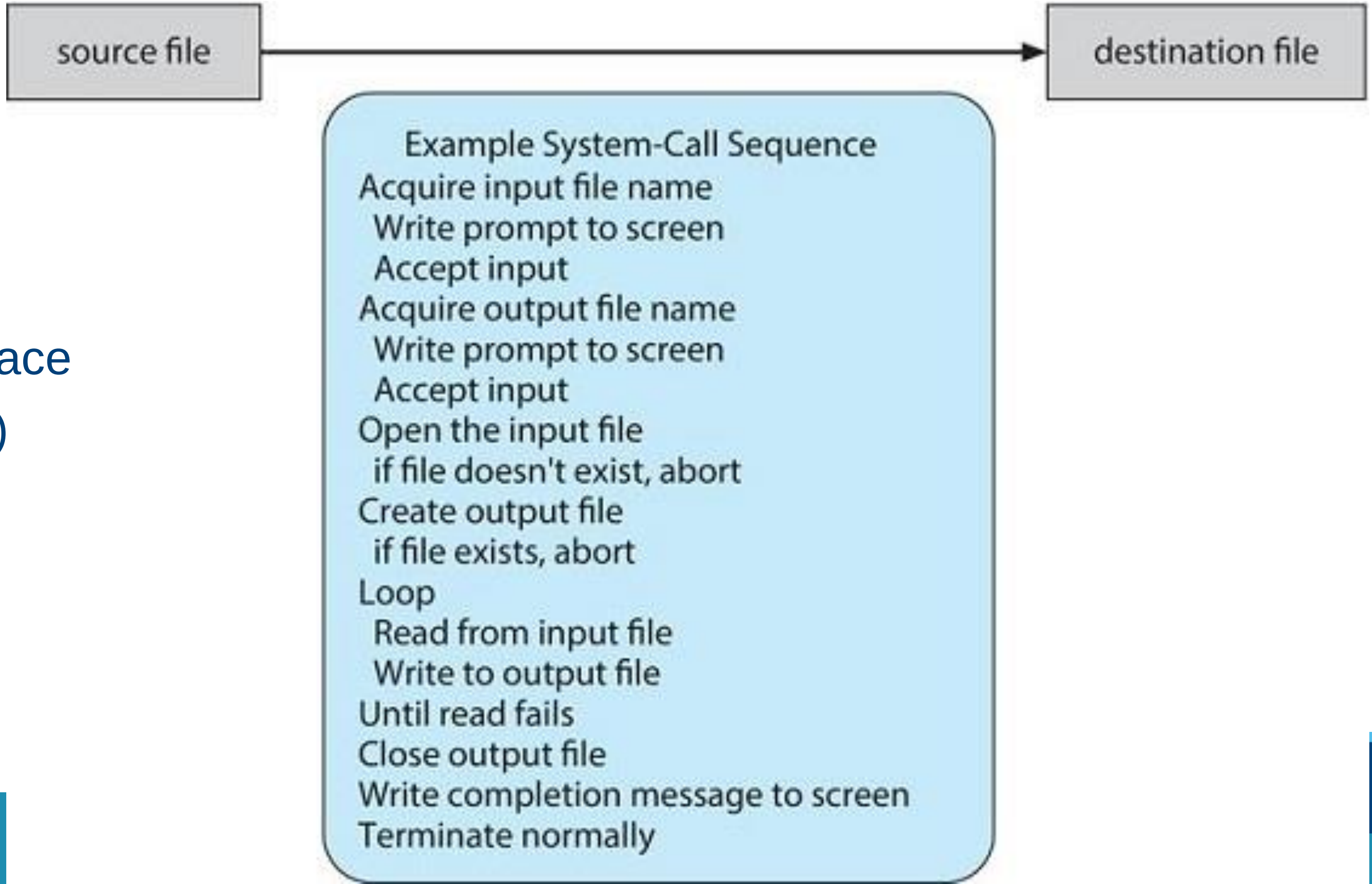
# 2.3 System calls

### Interface

- To the OS services
- Functions
- Written in C / C++
- Some low-level tasks: ASM

- **Used by programs** in user space
  - OS programs/tools (e.g. cp)
  - Your applications
- Example
  - cp in.txt out.txt

### 2.3.1 E.g. “cp”: copy in.txt to out.txt



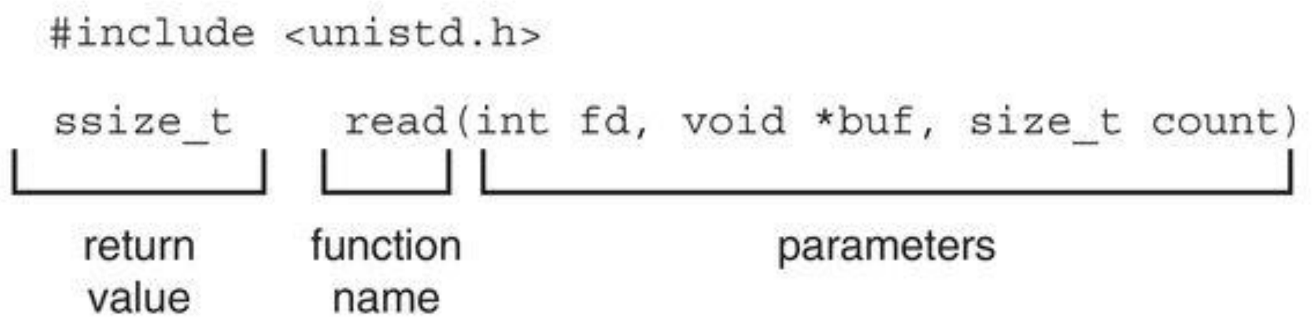
# 2.3.2 Application programming interface

## API

- **Even simple programs**
  - 1000s system calls per second
- **Too low level for application devs**
  - Solution: API
  - Higher-level functions for app devs
  - More stable, portable API
- **E.g.**
  - Win32 api for Windows
    - Abstracts over Win95/Win98/WinNT/Win10 kernel interface, which variates
  - POSIX api for linux/unix/macos
    - Kernel calls/interface abstracted

## Libc implements POSIX for C

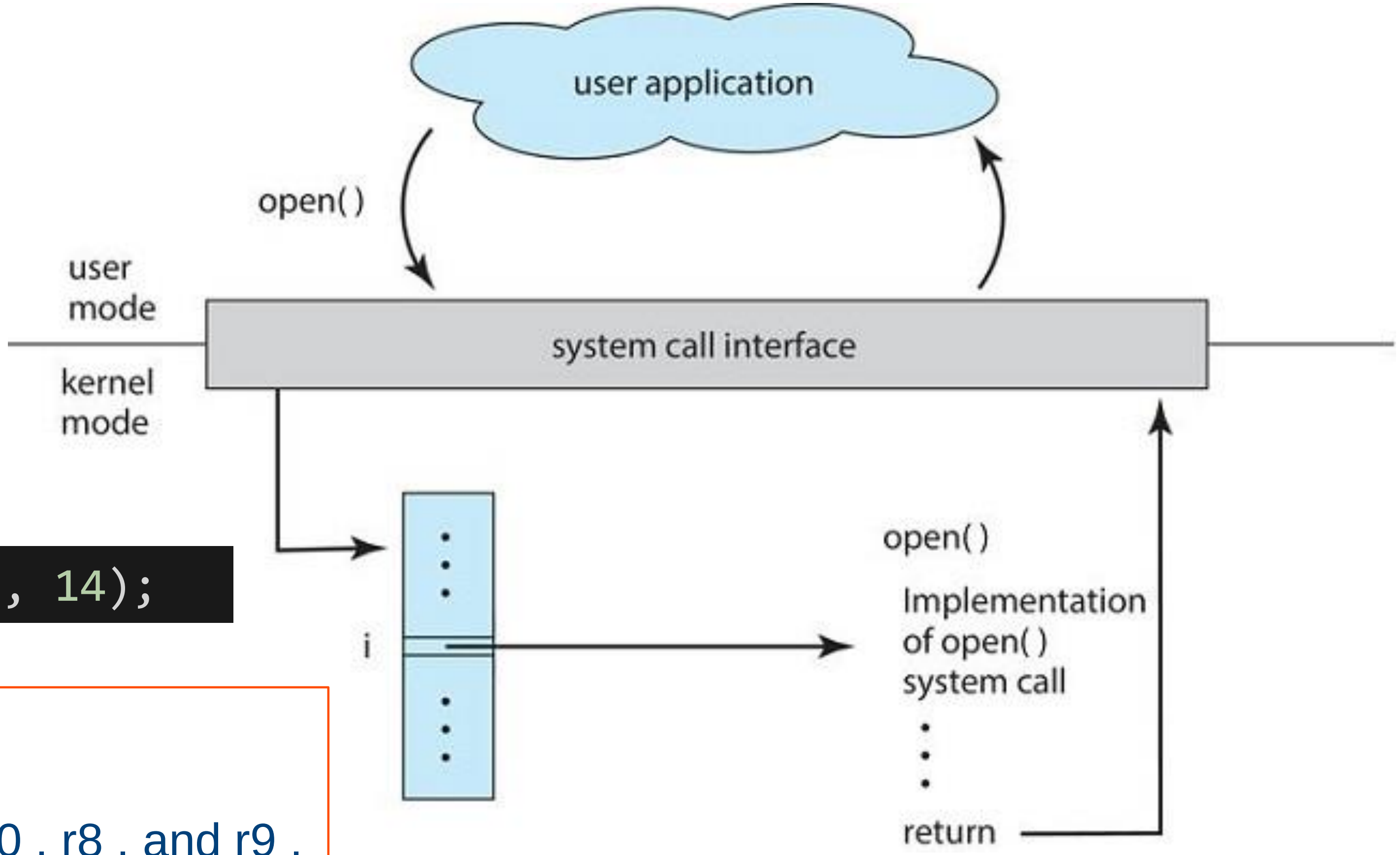
- E.g. : read() and write()
- int fd: file descriptor id
- Void *buf: buffer for the data
- Size_t count: nb of bytes to read



## From API tot system call
## From user mode to kernel mode

User application
→ API
→ System call
→ Kernel mode
→ Look up system call implementation
→ Execute implementation
→ Return

```
syscall(SYS_write, 1, "hello, world!\n", 14);
```



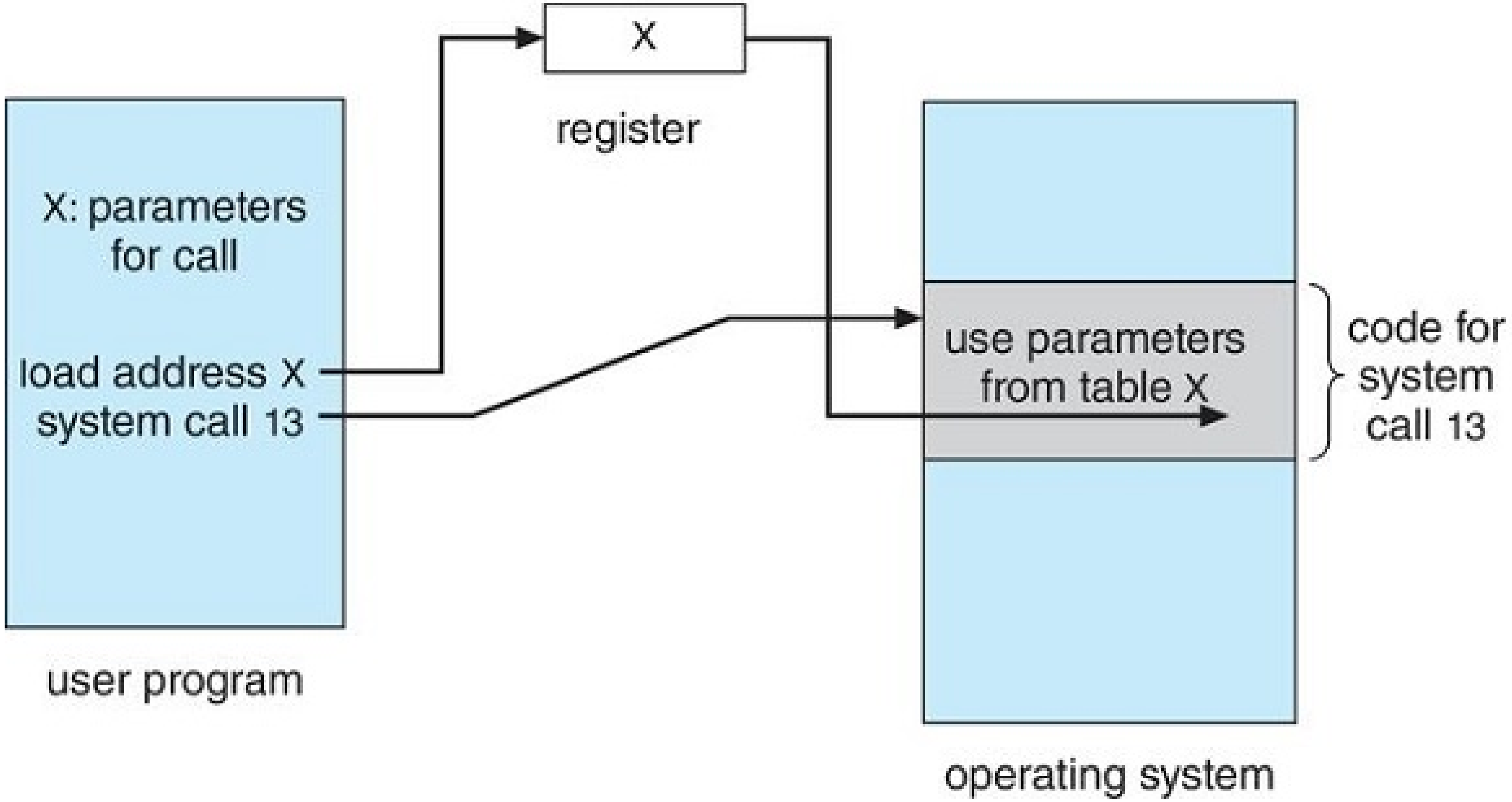

To make a system call in 64-bit Linux,

- place the system call number in rax ,
- then its arguments, in order, in rdi , rsi , rdx , r10 , r8 , and r9 ,
- then invoke syscall instruction .
- Some system calls return information, usually in rax .

# Passing parameters to OS

3 ways to pass params to system call

- → via registers (for small set of params)
- → Via block or table in memory
  - → address via register
- → User program pushes on stack,
  - → popped by OS.




```
syscall(SYS_write, 1, "hello, world!\n", 14);
```

## 2.3.3 Types of system calls

**Process control**

- Create and terminate process
- Allocate and free memory

**File management**

- Create delete read write open close

**Device management**

- Attach device. Read/write.

**Information management**

- Time and date. timers

**Communications**

- Shared memory. Pipes.

**Protection**

- File permissions

*EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS*

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

| | Windows | Unix |
|---|---|---|
| Process control | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject() | fork()<br>exit()<br>wait() |
| File management | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| Device management | SetConsoleMode()<br>ReadConsole()<br>WriteConsole() | ioctl()<br>read()<br>write() |
| Information maintenance | GetCurrentProcessID()<br>SetTimer()<br>Sleep() | getpid()<br>alarm()<br>sleep() |
| Communications | CreatePipe()<br>CreateFileMapping()<br>MapViewOfFile() | pipe()<br>shm_open()<br>mmap() |
| Protection | SetFileSecurity()<br>InitlializeSecurityDescriptor()<br>SetSecurityDescriptorGroup() | chmod()<br>umask()<br>chown() |

# 2.8 Operating system structure

Monoliths, modules and micro-kernels

## 2.8.1 Monolithic structure

“no structure at all”.

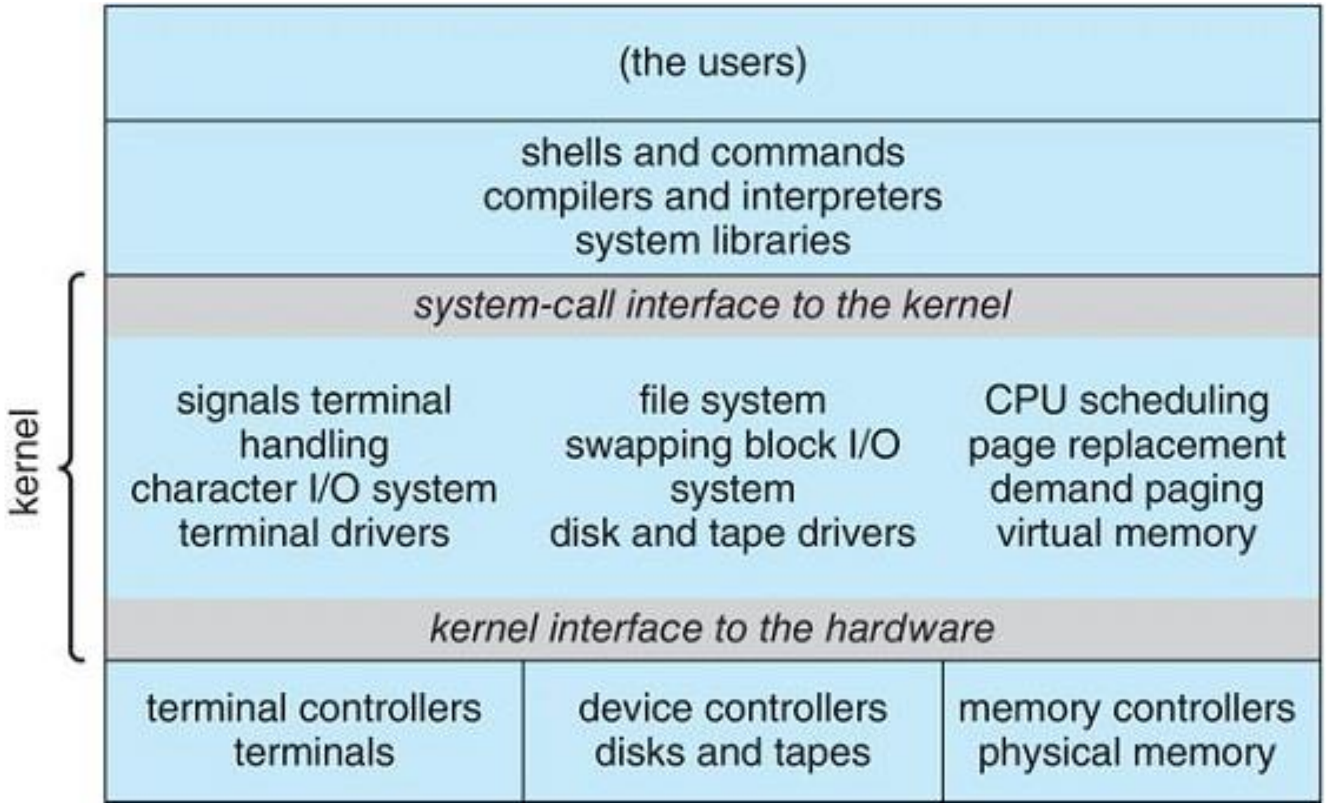
Kernel = 1 static binary file

e.g. UNIX

1 monolythic kernel + system programs

Kernel =

- Everything below system call interface
- Above physical hardware



# e.g. linux

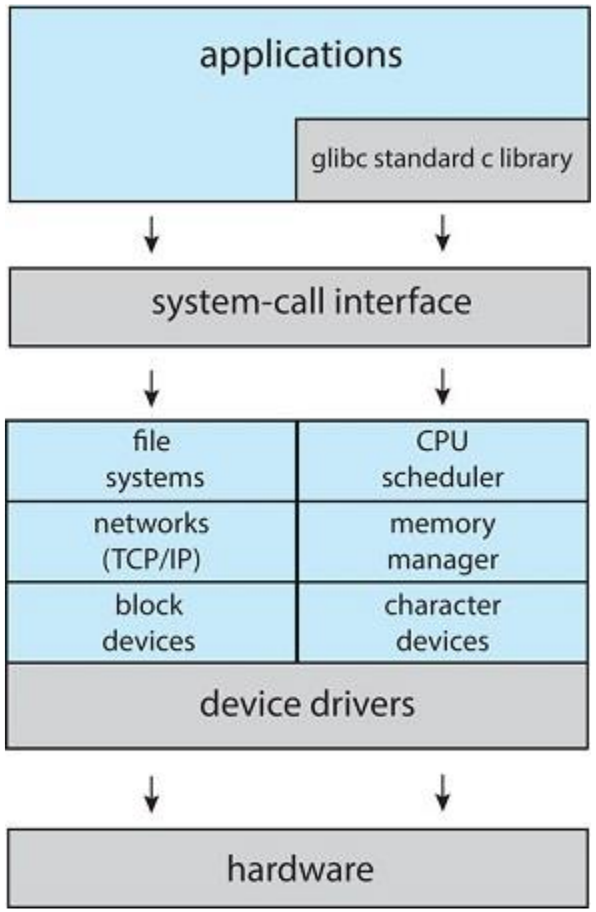
Based on UNIX, structured similarly

Glibc = api for applications

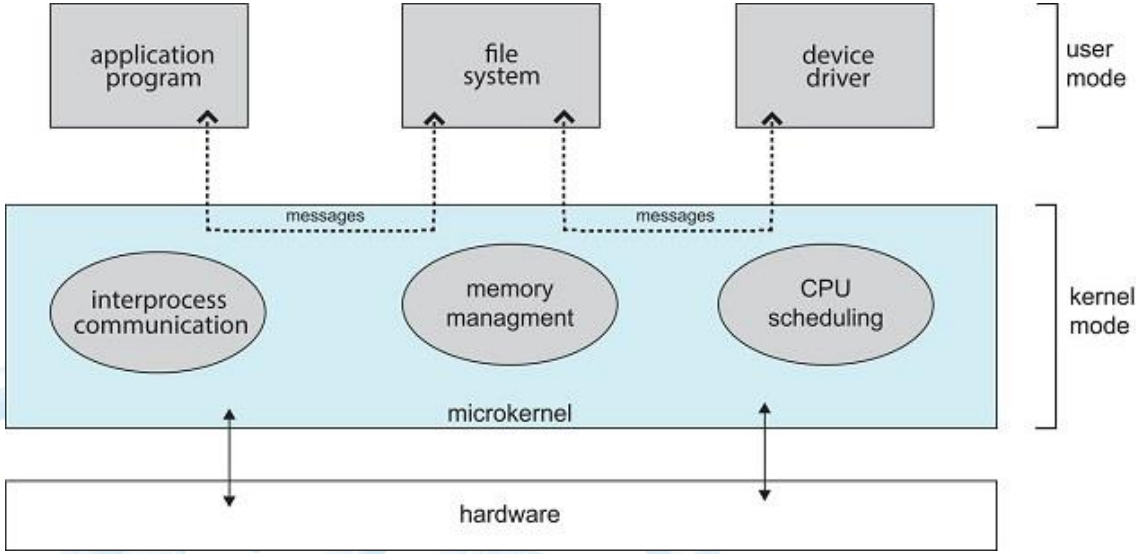
Linux kernel = monolithic
- Runs entirely in a single address space
- Modular design for runtime modification

+ advantages
- performance, speed, efficiency

-disadvantages
Difficult to implement
Difficult to extend



# 2.8.3 microkernels




- Modularization and minimization of the kernel

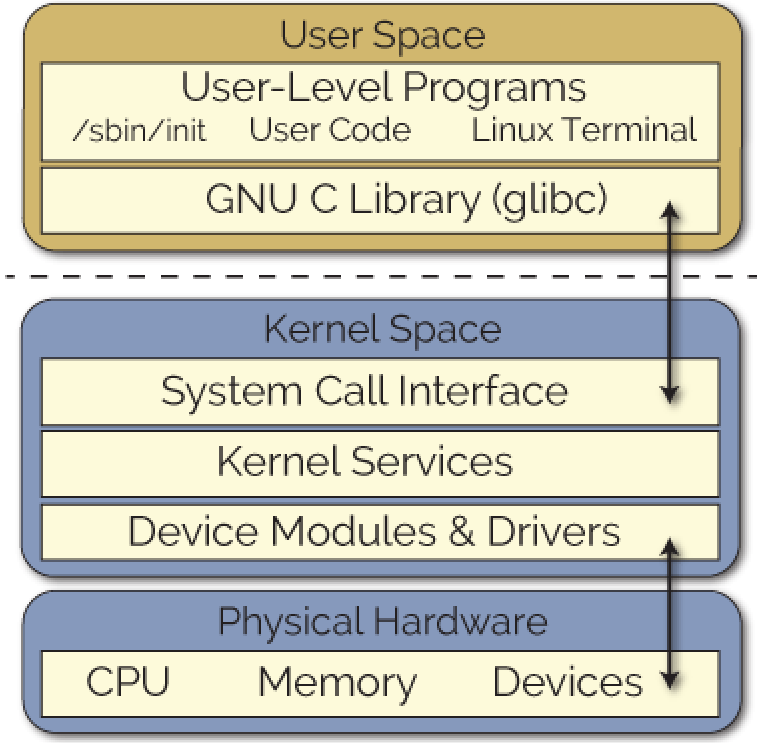
- e.g. Mach (1980s)
- reaction against UNIX's unmanageability

- Removes all non-essential components from kernel
- Implemented as user-level processes
- Different address spaces
- = much smaller kernel

- Today: Darwin kernel
- - Used by macOS and iOS.

# 2.8.4 Modules

- **Micro-kernel problems**
  - Have performance overhead
  - Lots of interprocess communication
  - Lots of context switches
- **Loadable kernel modules**
  - Run in kernel space
  - More efficient communication
  - Dynamically loaded
    - e.g. when usb drived plugged in
  - New services do not require kernel recompilation
    - E.g. for new file systems



# Quiz me quick

**Microkernel = kernel**

- A) containing many components that are optimized to reduce resident memory size
- B) that is compressed before loading in order to reduce its resident memory size
- C) that is compiled to produce the smallest size possible when stored to disk
- D) that is stripped of all nonessential components

**To load OS services dynamically you need**

- A) Virtual machines
- B) Modules
- C) File systems
- D) Graphical user interfaces

# Software engineering tools

gcc: compiling and linking.

Make, gdb, git.

# 2.5 Linkers and loaders

**Compilation** (gcc)

- From source to **relocatable object file**
- Designed to be loaded into physical mem

**Linker** (gcc)

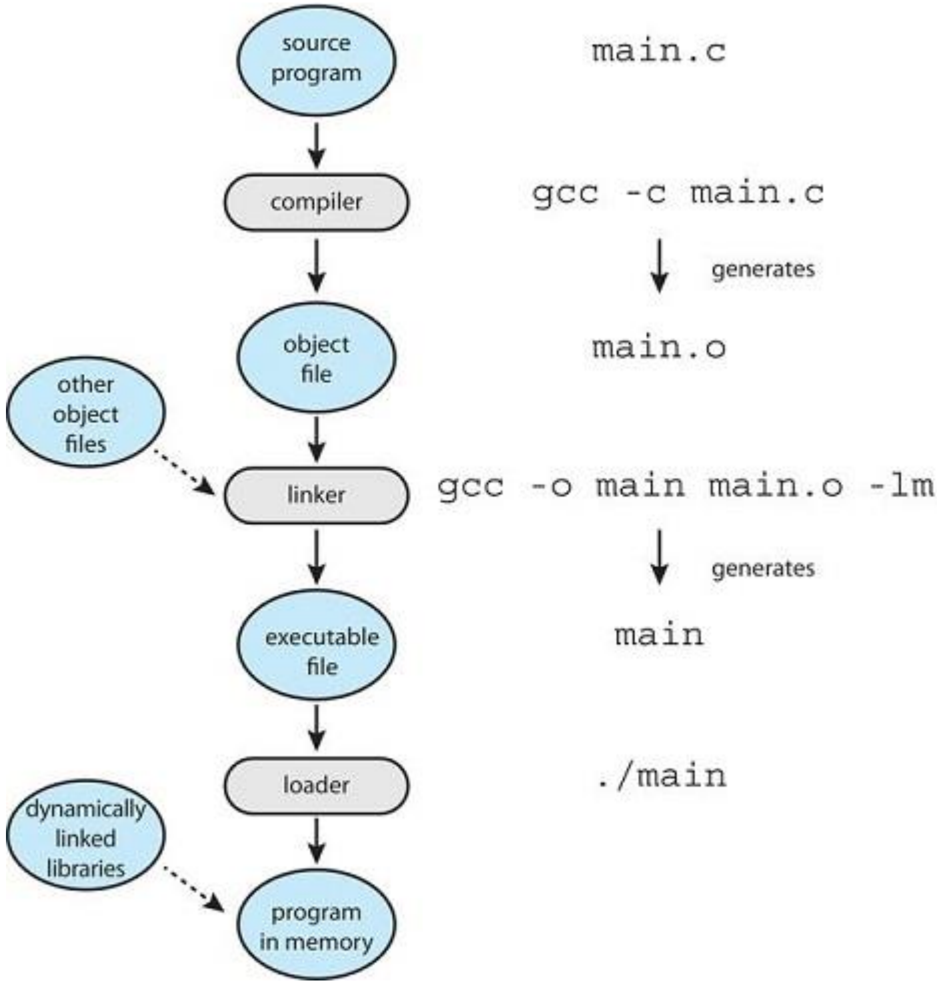
- Combines relocatable files into **executable**
- Includes other object files and libraries
- E.g. standard C lib (e.g. printf)
- E.g. math lib (-lm)
- UNIX: Executable and Linkable Format (ELF)

**Loader** (./)

- Relocation to final address
- Adjusts code and data in program to match

Dynamically linked library

- Loaded by need at **runtime**
- Shared by all programs that need it.



## Make:
## Complex compilation automation

Makefile:

```
hello:
	echo "hello world"
```

```
$ make
echo "hello world"
hello world
```

$ make blah

- Make is given blah as the target, so it first searches for this target
  - blah requires blah.o, so make searches for the blah.o target
  - blah.o requires blah.c, so make searches for the blah.c target
  - blah.c has no dependencies, so the echo command is run
- The gcc -c command is then run, because all of the blah.o dependencies are finished
- The top gcc command is run, because all the blah dependencies are finished
- That's it: blah is a compiled c program

```
blah: blah.o
 gcc blah.o -o blah # Runs third

blah.o: blah.c
 gcc -c blah.c -o blah.o # Runs second

blah.c:
 echo "int main() { return 0; }" > blah.c # Runs first
```

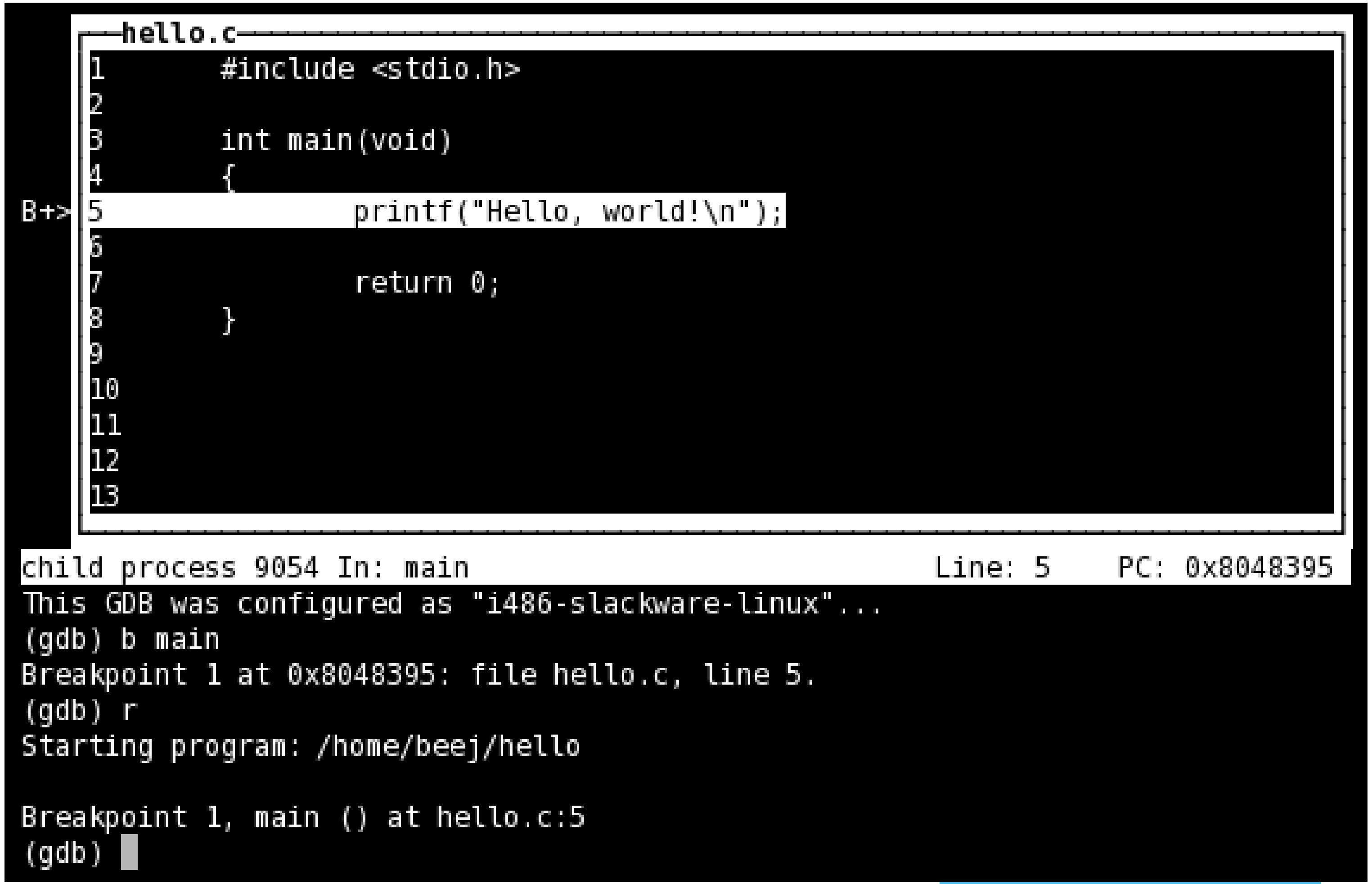
## Gdb in a nutshell

## $gdb -help

- **b** - Puts a breakpoint at the current line
- **b N** - Puts a breakpoint at line N
- **b fn** - Puts a breakpoint at the beginning of function "fn"
- **d N** - Deletes breakpoint number N
- **info break** - list breakpoints
- **r** - Runs the program until a breakpoint or error
- **c** - Continues running the program until the next breakpoint or error
- **f** - Runs until the current function is finished
- **s** - Runs the next line of the program
- **n** - Like s, but it does not step into functions
- **p var** - Prints the current value of the variable "var"
- **bt** - Prints a stack trace
- **u** - Goes up a level in the stack
- **d** - Goes down a level in the stack
- **q** - Quits gdb

## Adding debug info: gcc –g hello.c –o hello

## Run debugger: $gdb hello



# Git version control

# get clone of remote repo on your local machine:

git clone <repo>

e.g:

$ git clone git@host:username/repository.git.

$ git clone git@github.com:johndoe/my-app.git.

# do work

→ You editing files in the repo.

# see changes

git status

git add main.c

git commit

#Push your changes to the remote repo on git server:

git push