Review 1

Operating Systems
Wenbo Shen

Some slides are from Henri Casanova@Hawaii and Yajin Zhou@ZJU

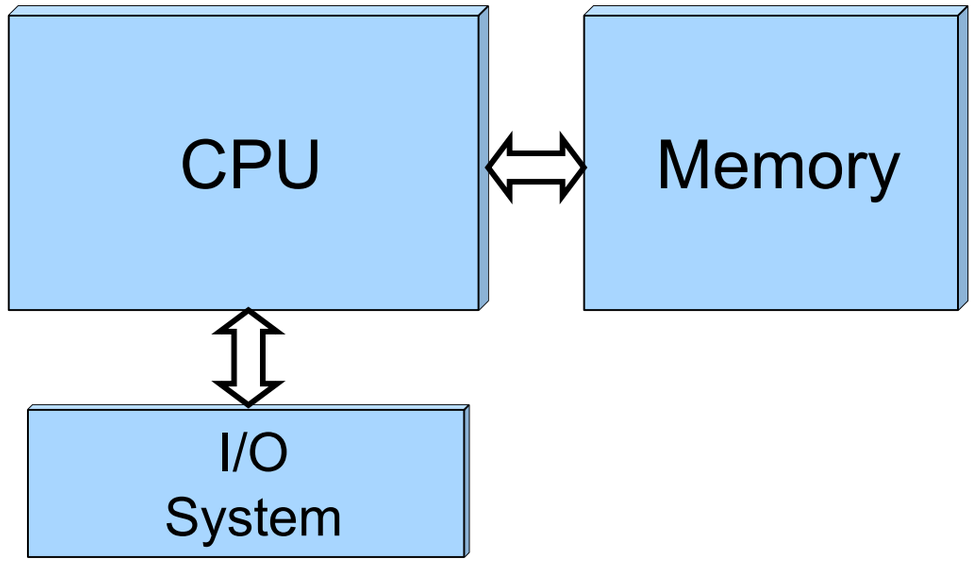
What have we studied so far

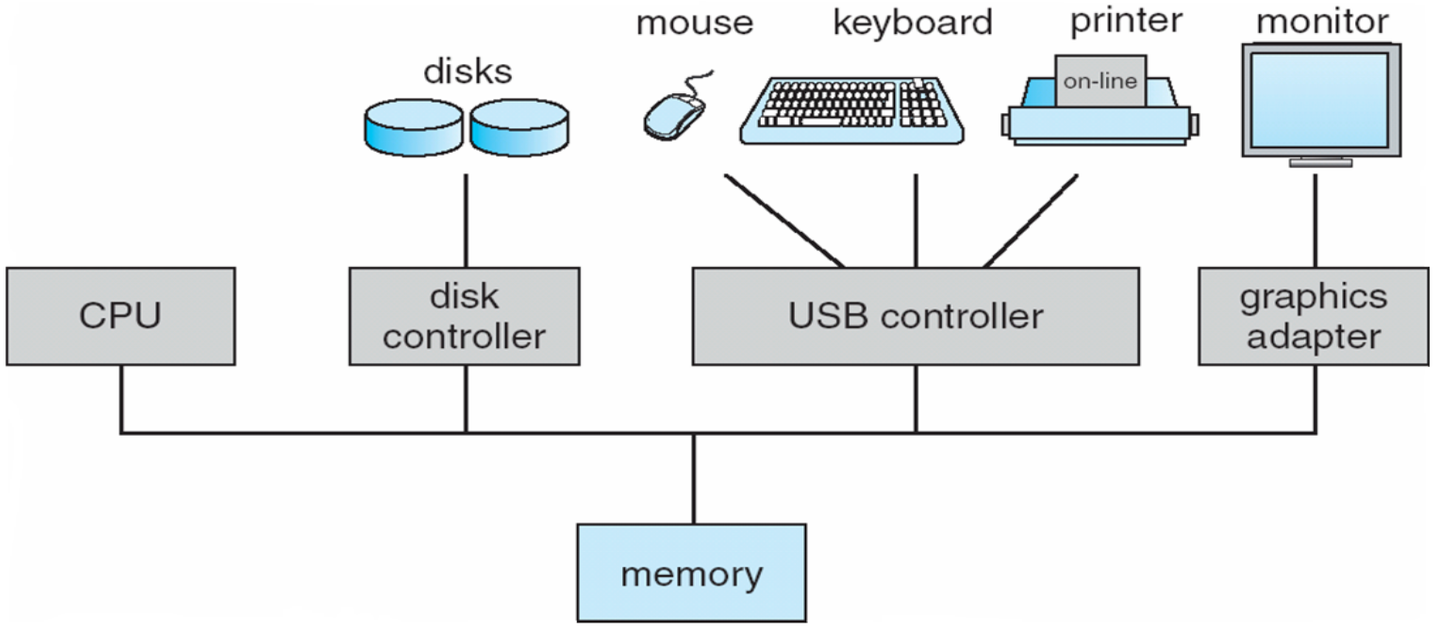
- Computer architecture
- OS overview
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

What have we studied so far

- Computer architecture
- OS overview
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

Von-Neumann Model





- Amazingly, it's still possible to think of the computer this way at a conceptual level (model from ~70 years ago!!!)
- A computer today has some differences

Data Stored in Memory

- All “information” in the computer is in binary form
  - Since Claude Shannon’s M.S. thesis in the 30’s
  - 0: zero voltage, 1: positive voltage (e.g., 5V)
  - bit: the smallest unit of information (0 or 1)
- The basic unit of memory is a byte
  - 1 Byte = 8 bits, e.g., “0101 1101”
- Each byte in memory is labeled by a unique address
- All addresses in the machine have the same number of bits
  - e.g., 64-bit addresses on ARM64
- The processor has instructions that say “Read the byte at address X and give me its value” and “Write some value into the byte at address X”

Conceptual View of Memory

address	content
0000 0000 0000 0000	0110 1110
0000 0000 0000 0001	1111 0100
0000 0000 0000 0010	0000 0000
0000 0000 0000 0011	0000 0000
0000 0000 0000 0100	0101 1110
0000 0000 0000 0101	1010 1101
0000 0000 0000 0110	0000 0001
0000 0000 0000 0111	0100 0000
0000 0000 0000 1000	1111 0101
...	...

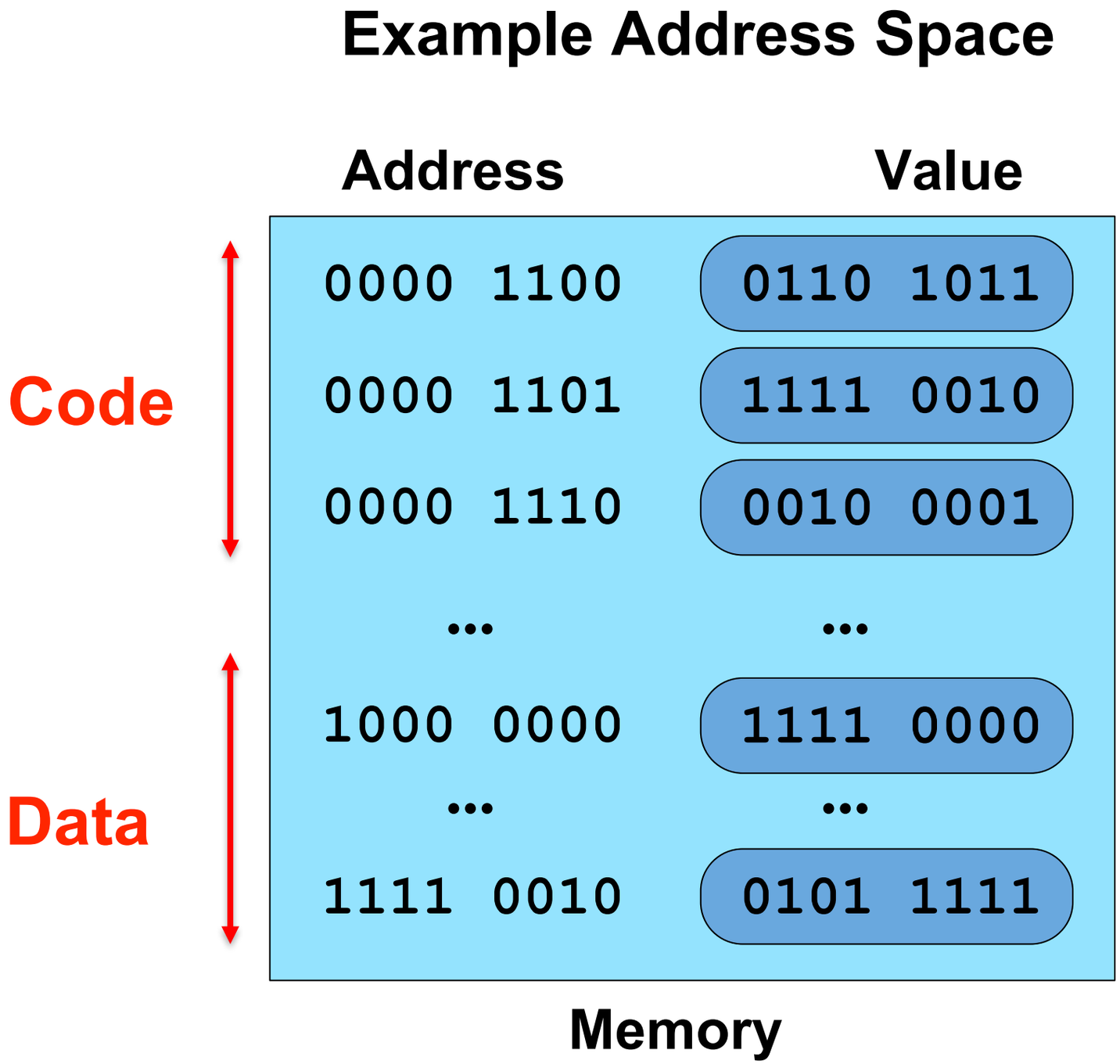
Conceptual View of Memory

address	content
0000 0000 0000 0000	0110 1110
0000 0000 0000 0001	1111 0100
0000 0000 0000 0010	0000 0000
0000 0000 0000 0011	0000 0000
0000 0000 0000 0100	0101 1110
0000	
0000	
0000	
0000 0000 0000 1000	1111 0101
...	...

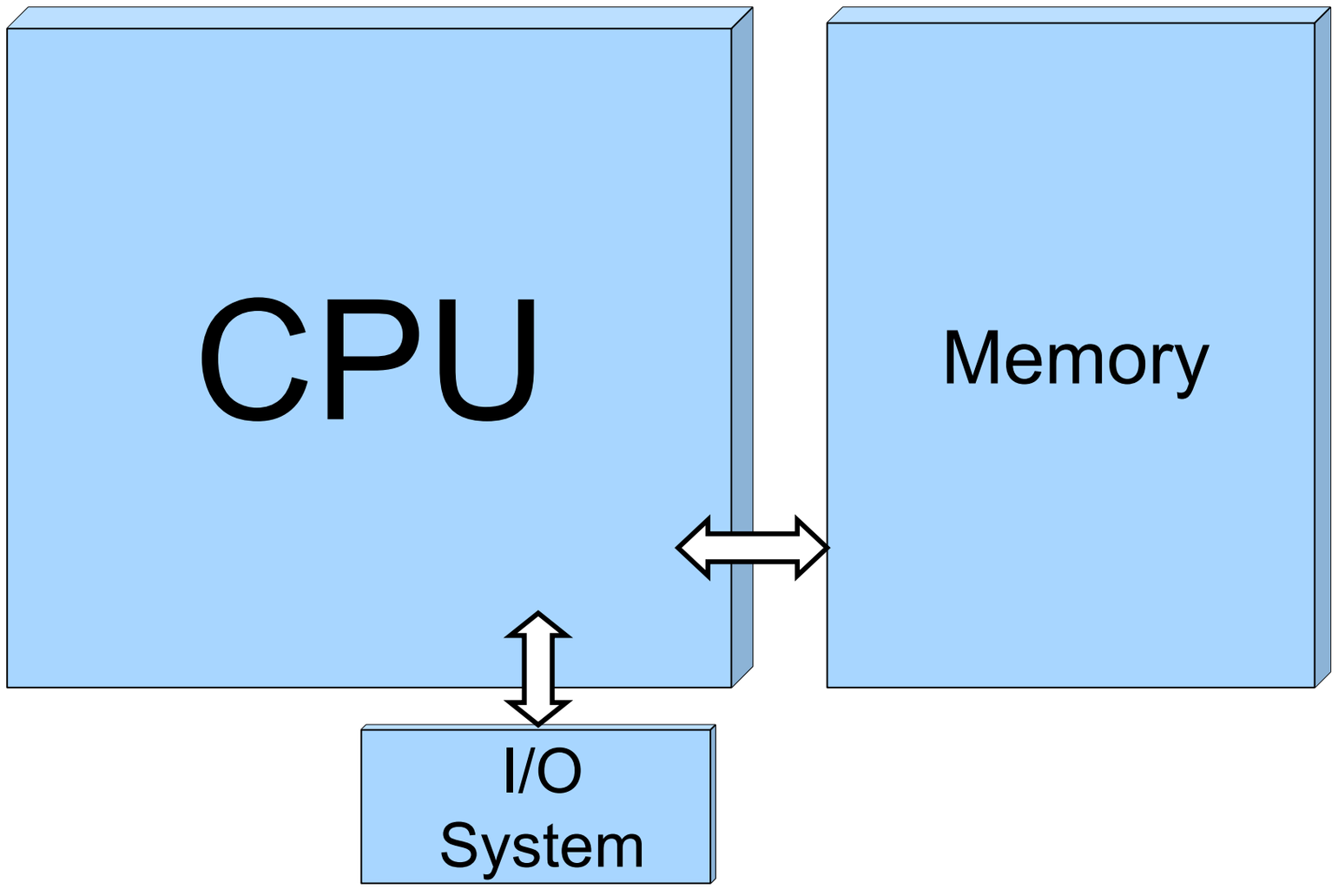
At address 0000 0000 0000 0100 the content is 0101 1110

Both Code and Data in Memory

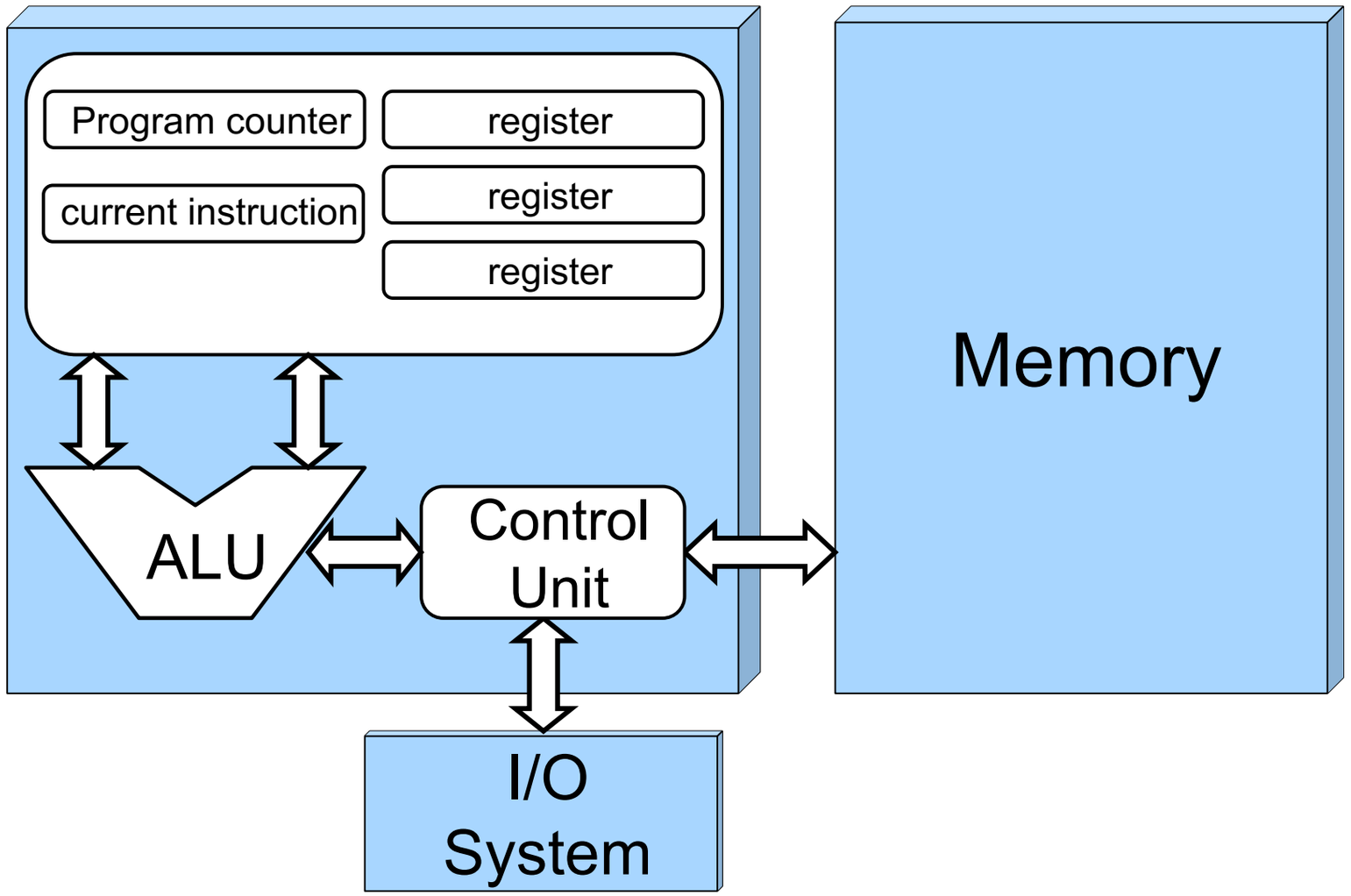
- Once a program is loaded in memory, its address space contains both code and data
- To the CPU those are not really different, but the programmer knows which bytes are data an which are code
  - Always conveniently hidden from you if you've never written assembly
  - But we'll have to keep code/data straight in these lecture notes



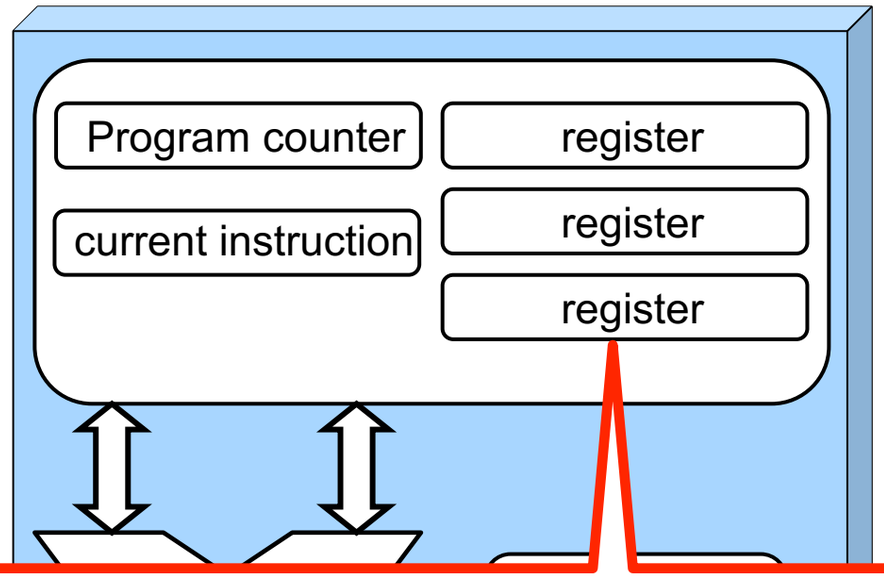
What's in the CPU?



What's in the CPU?



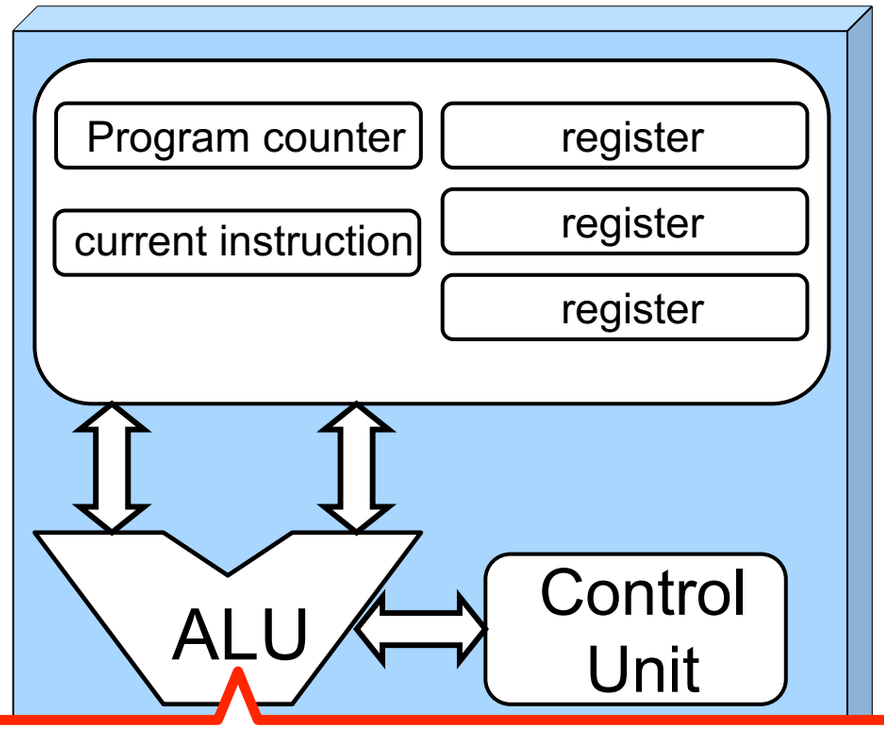
What's in the CPU?



Registers: the "variables" that hardware instructions work with

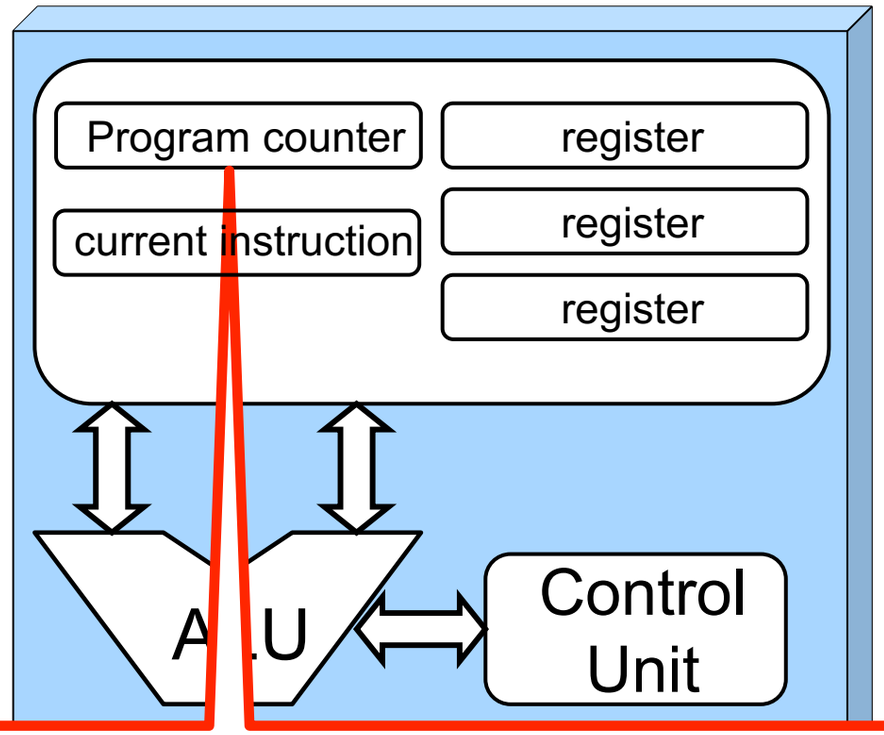
Data can be loaded from memory into a register
Data can be stored from a register back into memory
Operands and results of computations are in registers
Accessing a register is really fast
There is a limited number of registers

What's in the CPU?



Arithmetic and Logic Unit: what you do computation with
Used to compute a value based on current register values and store the result back into a register
+, *, /, -, OR, AND, XOR, etc.

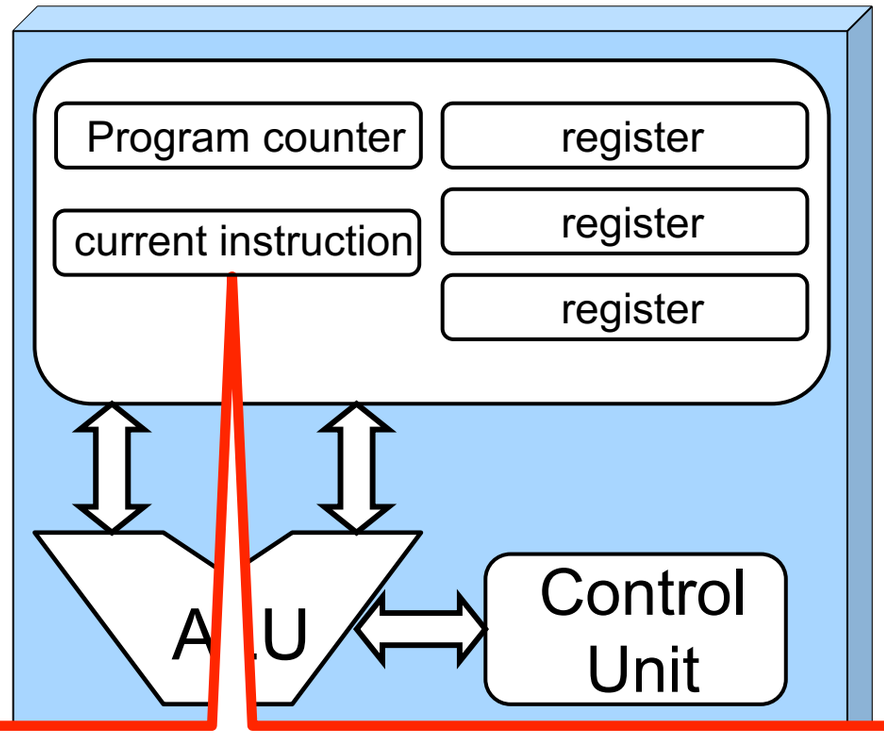
What's in the CPU?



Program Counter: Points to the next instruction

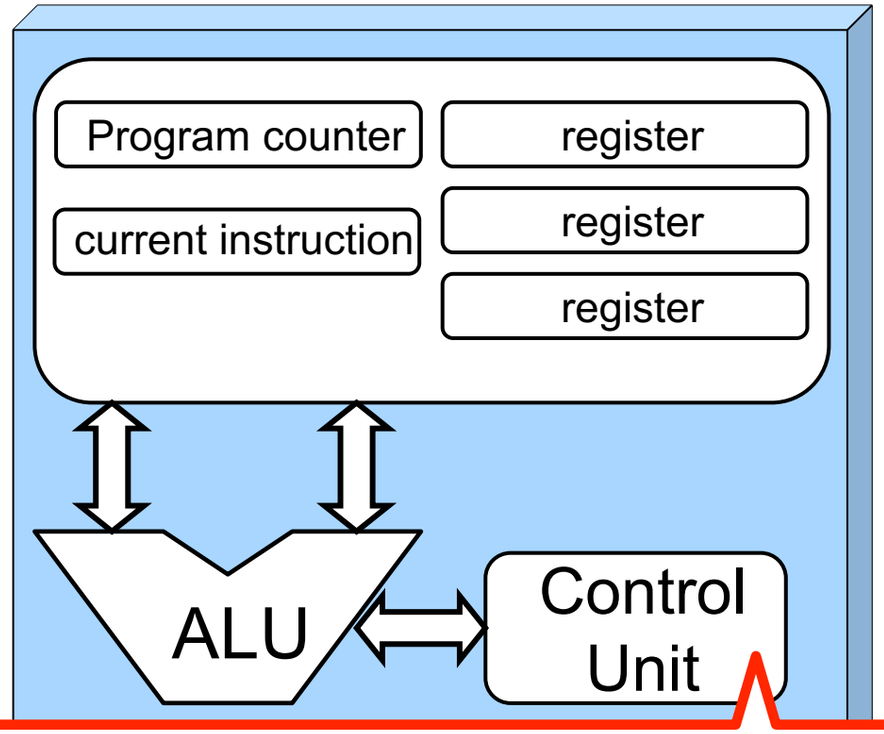
Special register that contains the address in memory of the next instruction that should be executed
(gets incremented after each instruction, or can be set to whatever value whenever there is a change of control flow)

What's in the CPU?



Current Instruction: Holds the instruction that's currently being executed

What's in the CPU?



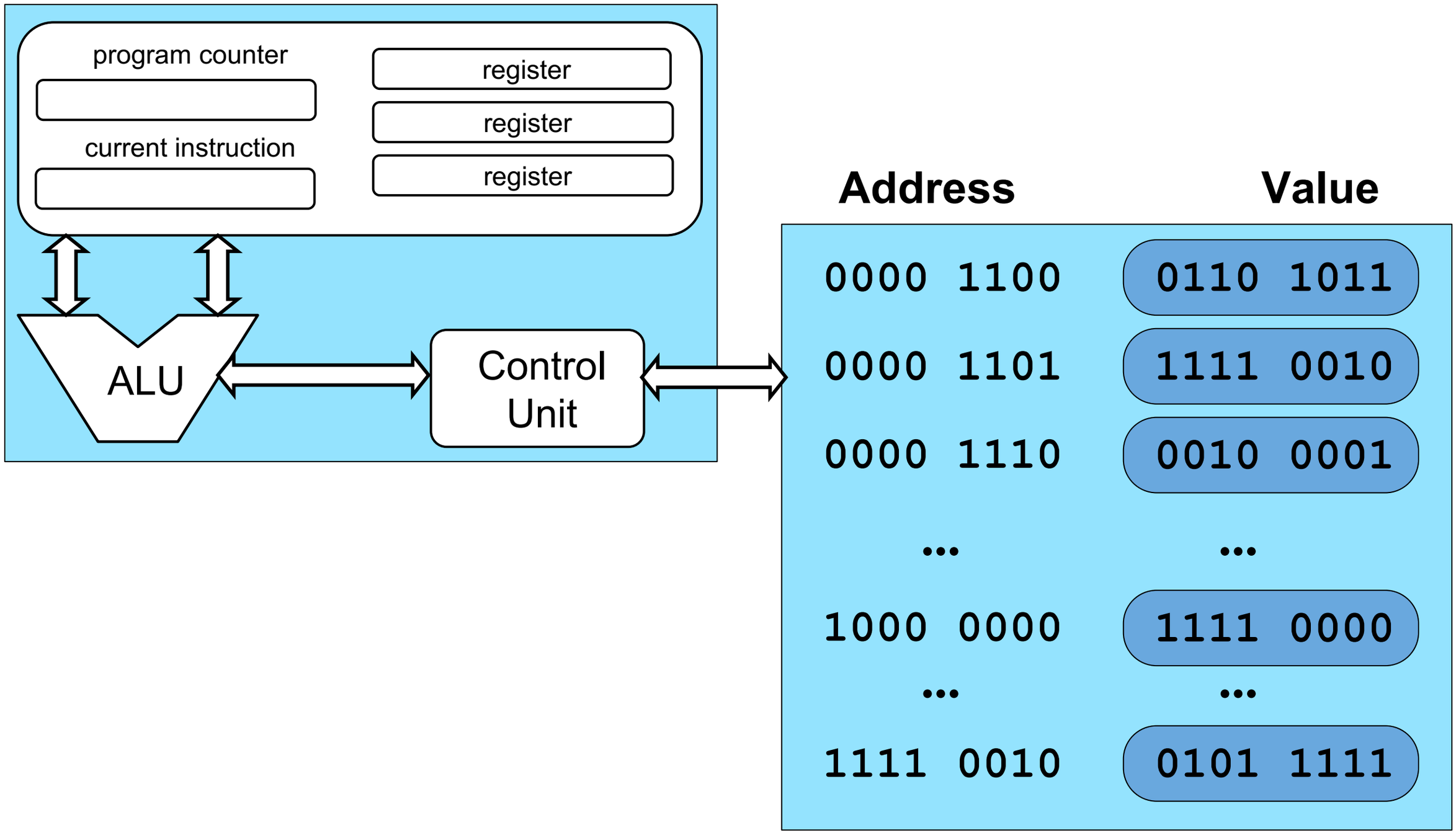
Control Unit: Decodes instructions and make them happen

Logic hardware that decodes instructions (i.e., based on their bits) and sends the appropriate (electrical) signals to hardware components in the CPU

Fetch-Decode-Execute Cycle

- The Fetch-Decode-Execute cycle
  - The control unit fetches the next program instruction from memory
    - Using the program counter to figure out where that instruction is located in the memory
  - The instruction is decoded and signals are send to hardware components
    - e.g., is the instruction loading something from memory? is it adding two register values together?
    - Operands are fetched from memory and put in registers, if needed
    - The ALU executes computation, if any, and store results in the registers
    - Register values are stored back to memory, if needed
    - Repeat
- Computers today implement MANY variations on this model
- But one can still program with the above model in mind
  - but certainly without (fully) understanding performance issues

Fetch-Decode-Execute



What have we studied so far

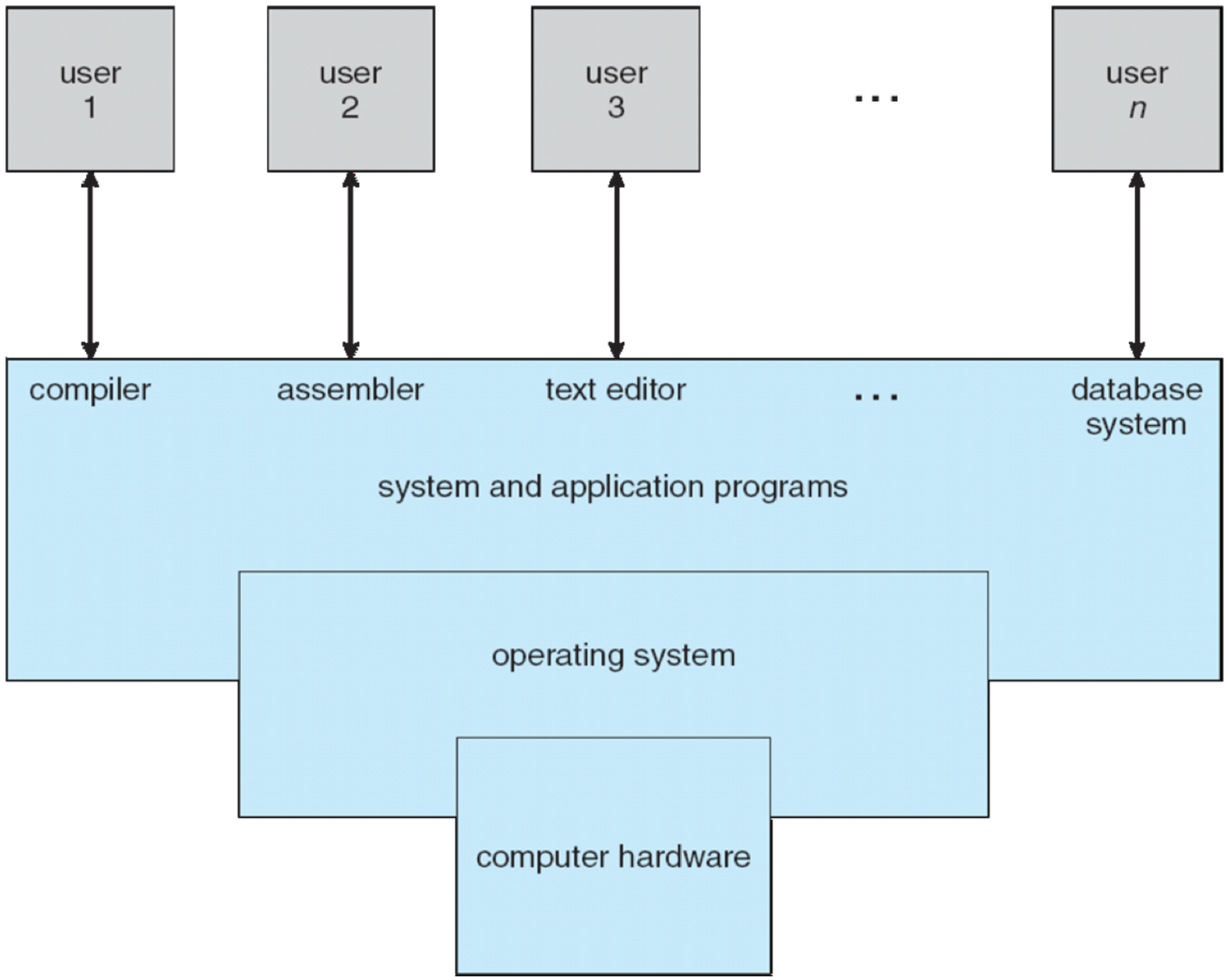
- Computer architecture
- OS overview
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

What Operating Systems Do

- OS is a **resource allocator**
  - The OS manages all resources
  - The OS decides who (which running program) gets what resource (share) and when

- OS is a **control program**
  - it controls program execution to prevent **errors** and **improper use** of system

Four Components of a Computer System



OS Events

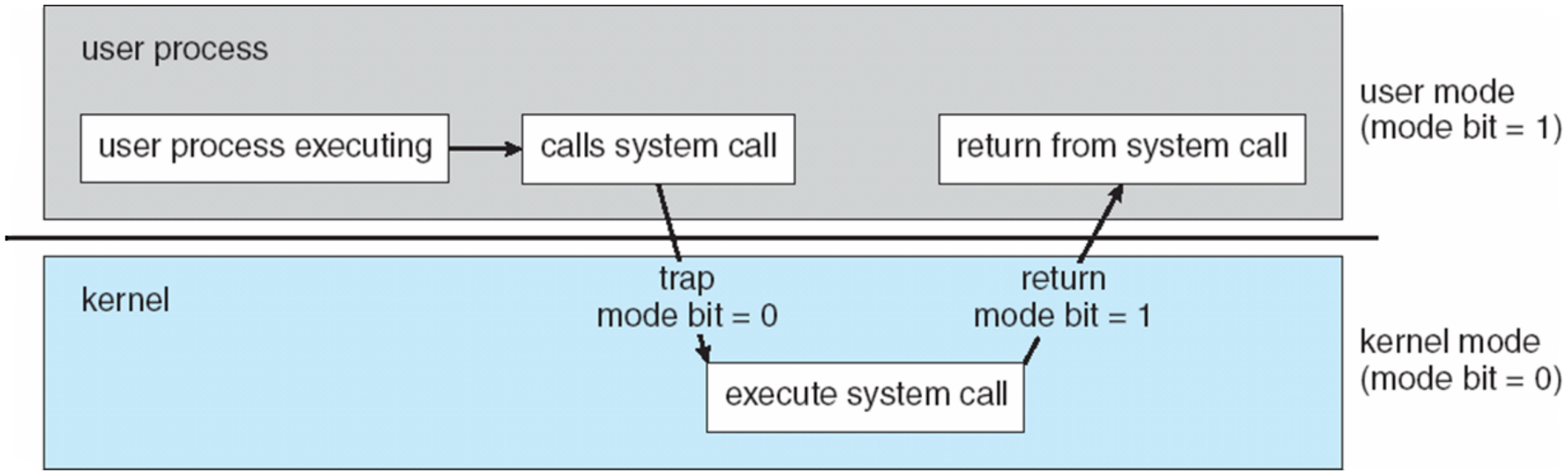
- There are two kinds of events: interrupts and traps (or exceptions)
  - The two terms are often confused (even in the textbook)
  - The term fault often refers to unexpected events
- Interrupts are caused by external events
  - Hardware-generated
  - e.g., some device controller says “something happened”
- Traps are caused by executing instructions
  - Software-generated interrupts
  - e.g., the CPU tried to execute a privileged instruction but it’s not in kernel mode
  - e.g., a division by zero

System Calls

- When a user program needs to do something privileged, it calls a system call
  - e.g., to create a process, write to disk, read from the network card
- A system call is a special kind of trap
- Every Instruction Set Architecture (ISA) provides a system call instruction that
  - Causes a trap, which maps to a kernel handler
  - Passes a parameter determining which system call to place (a number)
  - Saves caller state (PC, regs, mode) so it can be restored later
- On the x86 architecture the instruction is called int

```
mov   eax, 12
int               // places system call #12
```

System Calls



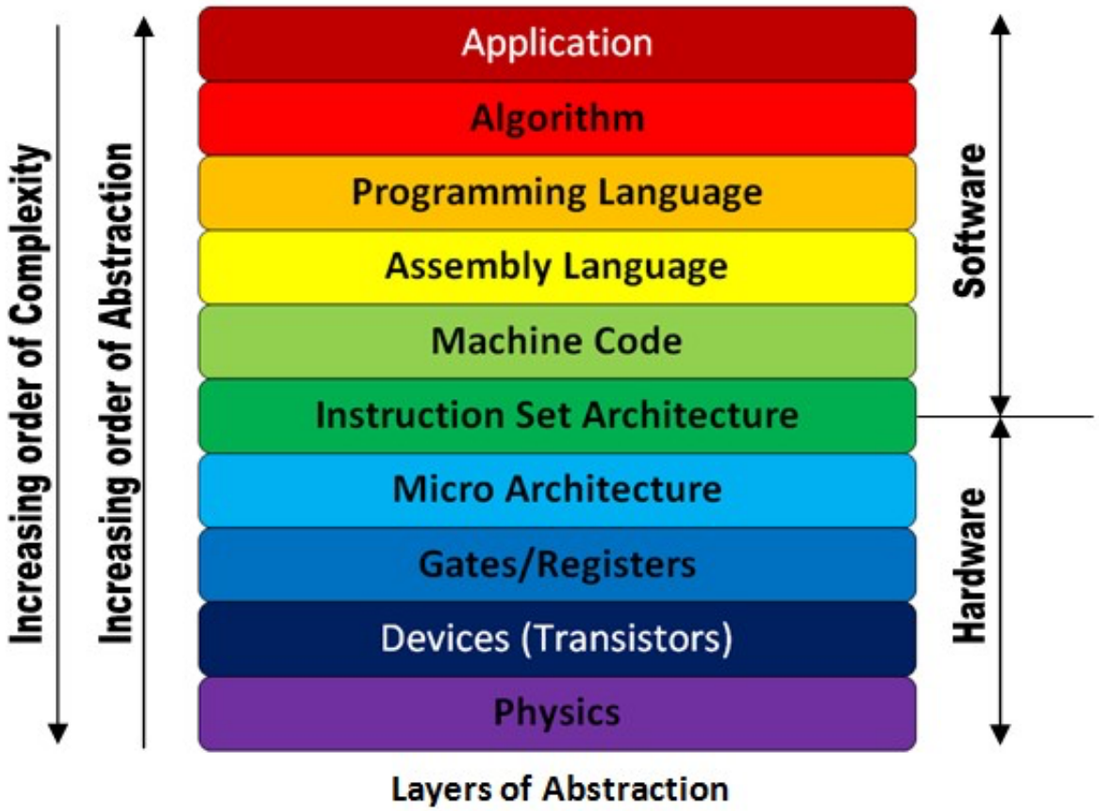
Main OS Services

- Process Management
- Memory Management
- Storage Management
- I/O Management
- Protection and Security

What have we studied so far

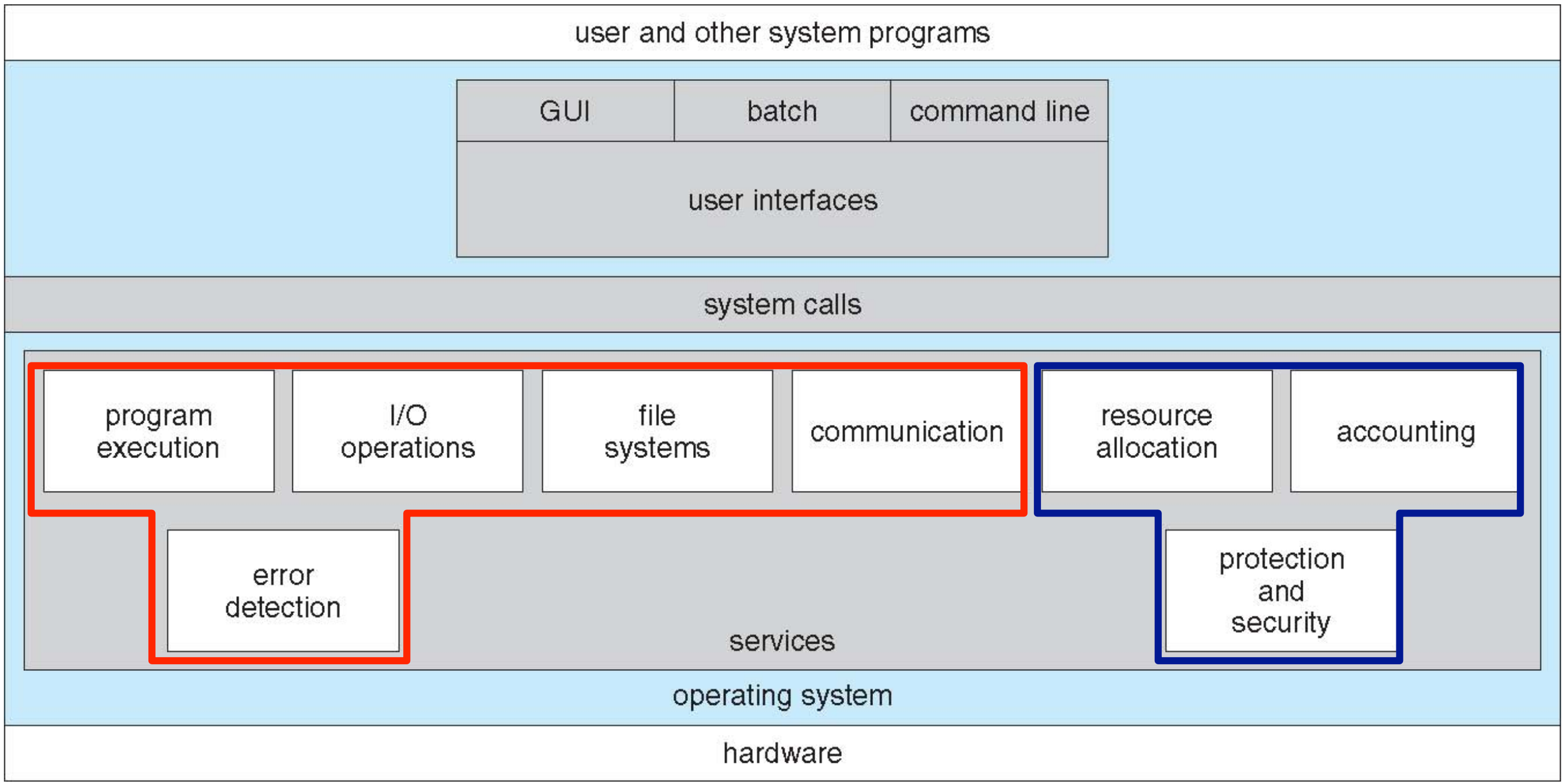
- Computer architecture
- OS overview
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

Abstraction



Abstractions are fundamental to everything we do in computer science. Abstraction makes it possible to write a large program by dividing it into small and understandable pieces, to write such a program in a high-level language like C without thinking about assembly, to write code in assembly without thinking about logic gates, and to build a processor out of gates without thinking too much about transistors.

OS Services and Features



Helpful to users

Better efficiency/operation

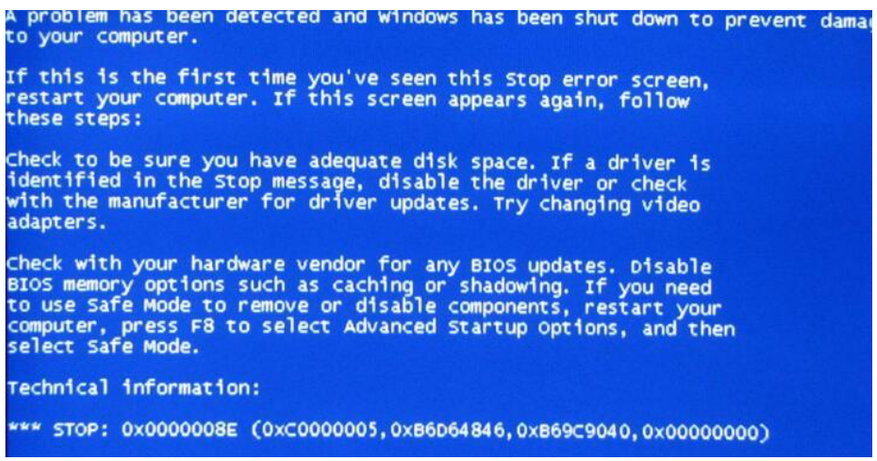
Operating System Services (User-Visible)

- **User interface**
  - most operating systems have a user interface (UI).
  - e.g., command-Line (CLI), graphics user interface (GUI), or batch
- **Program execution: from program to process**
  - load and execute an program in the memory
  - end execution, either normally or abnormally
- **I/O operations**
  - a running program may require I/O such as file or I/O device
- **File-system manipulation**
  - read, write, create and delete files and directories
  - search or list files and directories
  - permission management

```
  13 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/1
  14 root      rt   0       0      0      0 S   0.0  0.0   0:00.13 watchdog/1
  15 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/1
  16 root      20   0       0      0      0 S   0.0  0.0   0:00.11 ksoftirqd/1
  18 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/1:+
  19 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
  20 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 netns
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$ ls
Desktop    Downloads  Pictures  Templates  examples.desktop
Documents  Music      Public    Videos     os2018fall
os@os:~$ pwd
/home/os
os@os:~$
```

Operating System Services (User-Visible)

- **Communications**
  - processes exchange information, on the same system or over a network
  - via shared memory or through message passing
- **Error detection**
  - OS needs to be constantly aware of possible errors
  - errors in CPU, memory, I/O devices, programs
  - it should take appropriate actions to ensure correctness and consistency

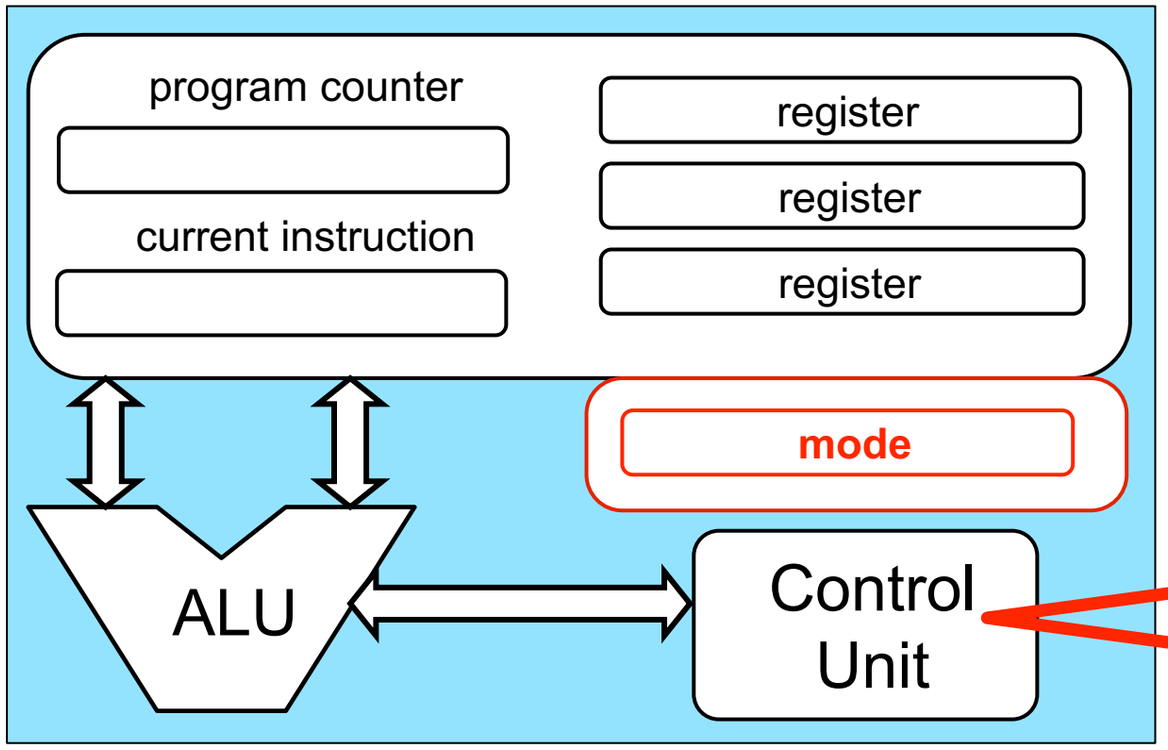


Operating System Services (System View)

- **Resource allocation**
  - allocate resources for multiple users or multiple jobs running concurrently
  - many types of resources: CPU, memory, file, I/O devices
- **Accounting/Logging**
  - to keep track of which users use how much and what kinds of resources
- **Protection and security**
  - protection provides a mechanism to control access to system resources
    - access control: control access to resources
    - isolation: processes should not interfere with each other
  - security authenticates users and prevent invalid access to I/O devices
    - a chain is only as strong as its weakest link
  - protection is the **mechanism**, security towards the **policy**

```
top - 01:25:31 up 14:16,  3 users,  load average: 0.00, 0.00, 0.00
Tasks:  98 total,   1 running,  97 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.0 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1003772 total,   217452 free,    69248 used,   717072 buff/cache
KiB Swap:  1046524 total,  1046012 free,      512 used.   749832 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26713 root      20   0       0      0      0 S   0.3  0.0   0:00.01 kworker/u6+
    1 root      20   0  119600   5076   3336 S   0.0  0.5   0:03.39 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.01 kthreadd
    4 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:+
    6 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 mm_percpu_+
    7 root      20   0       0      0      0 S   0.0  0.0   0:00.04 ksoftirqd/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:01.02 rcu_sched
    9 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
   10 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.13 watchdog/0
   12 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/0
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/1
   14 root      rt   0       0      0      0 S   0.0  0.0   0:00.13 watchdog/1
   15 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/1
   16 root      20   0       0      0      0 S   0.0  0.0   0:00.11 ksoftirqd/1
   18 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/1:+
   19 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
   20 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 netns
```

User vs. Kernel Mode



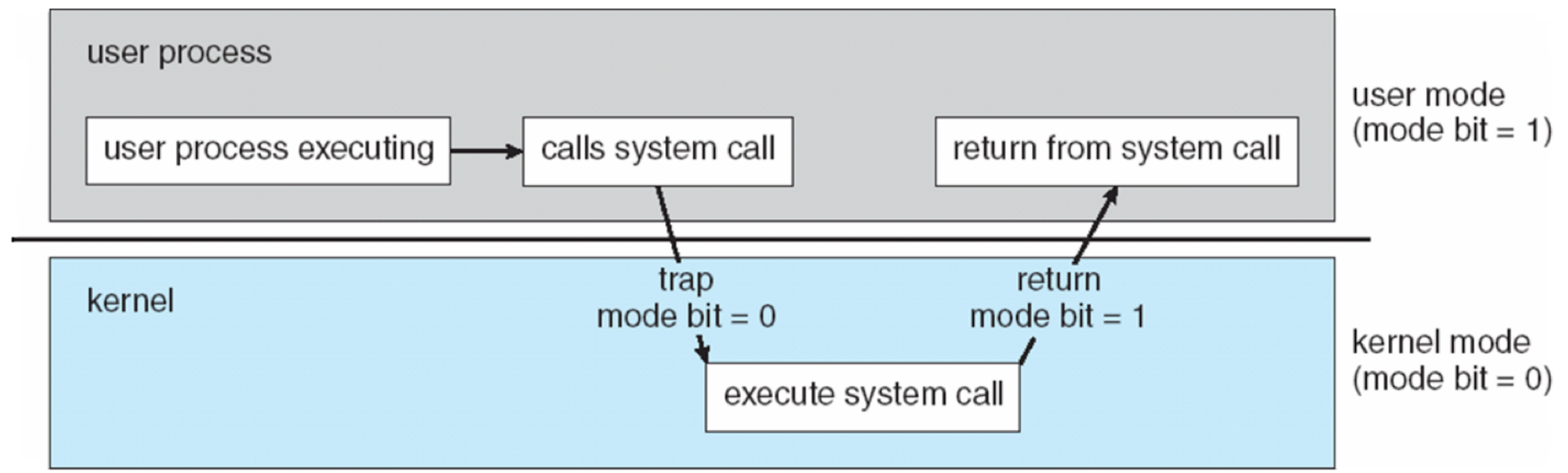
- Decode instruction
- Determine if instruction is privileged or not
  - Based on the instruction code (e.g., the binary code for all privileged instructions could start with ’00’)
- If instruction is privileged and mode == user, then abort!
  - Raise a “trap”

Dual-mode operation

- Operating system is usually interrupt-driven (why?)
  - Efficiency, regain control (timer interrupt)
- **Dual-mode operation** allows OS to protect itself and other system components
  - **user mode** and **kernel mode (or privileged mode)**
  - a **mode** bit distinguishes when CPU is running user code or kernel code
  - some instructions designated as **privileged**, only executable in kernel
  - **system call** changes mode to kernel, return from call resets it to user

Transition between Modes

- **System calls, exceptions, interrupts** cause transitions between kernel/user modes

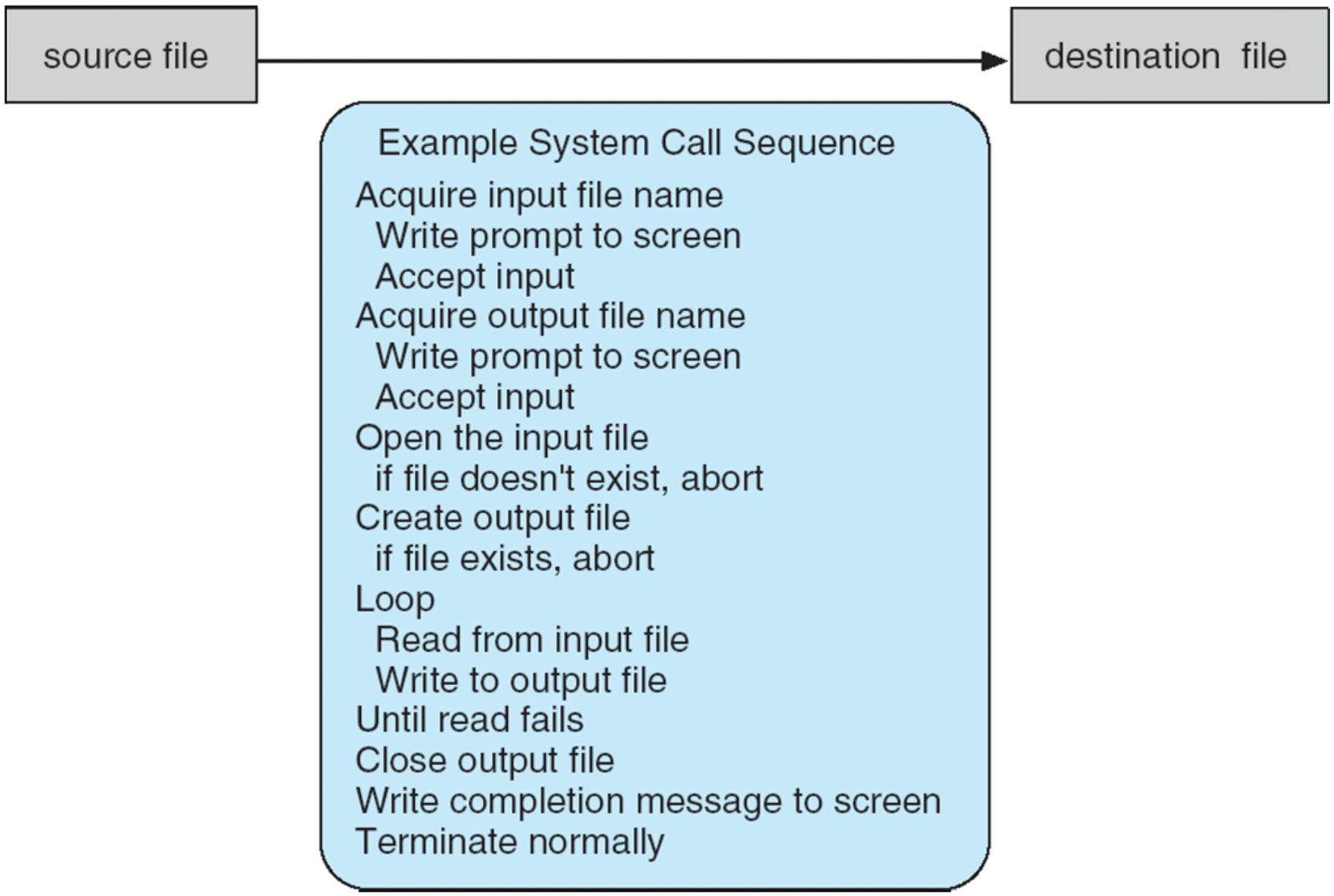


System Calls

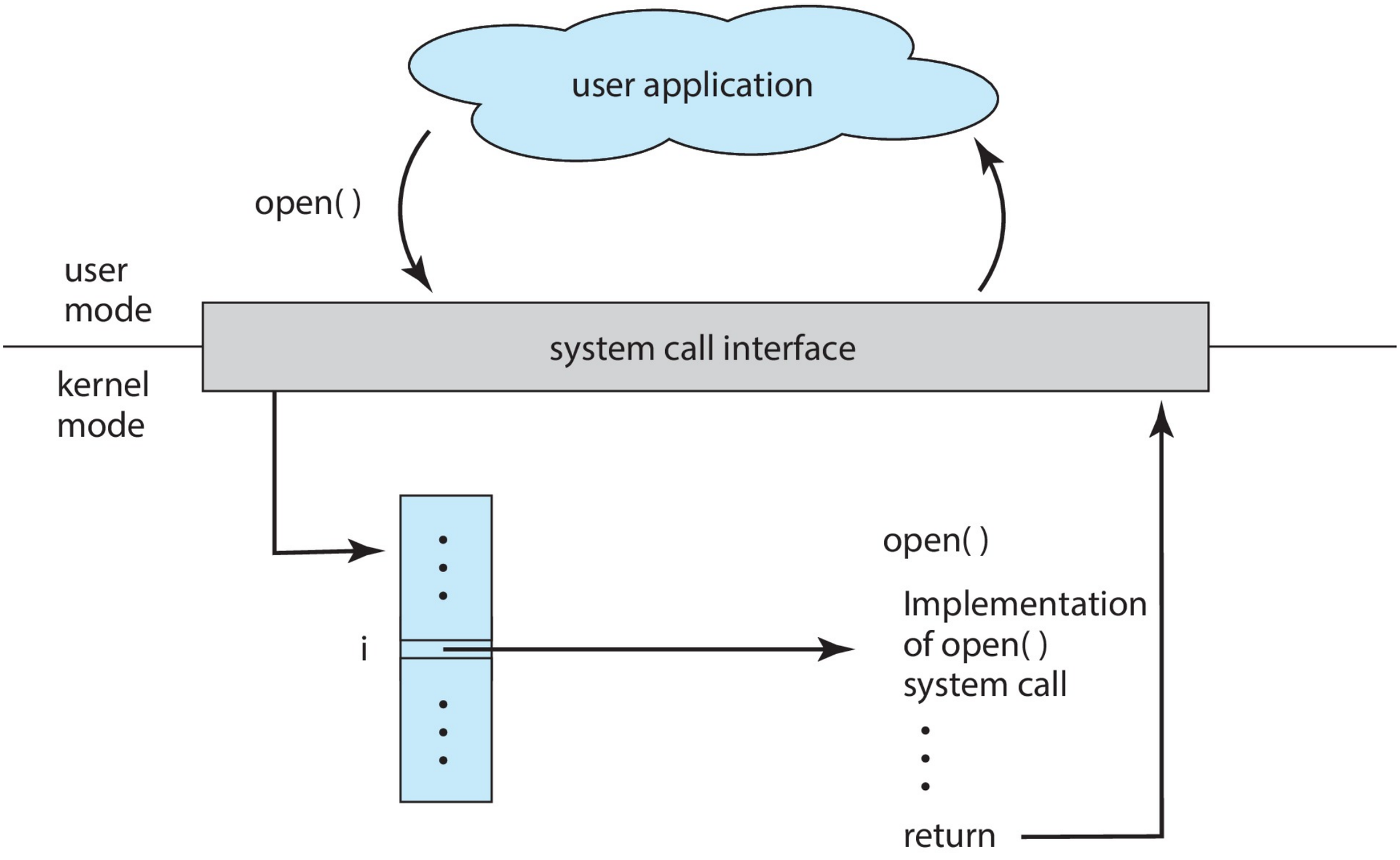
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

Example of System Calls

- System call sequence to copy the contents of one file to another file



API – System Call – OS Relationship



The Hidden Syscall Table

- Let's look a bit inside the Linux Kernel
- Linux kernel v5.3, ARM64
  - arch/arm64/kernel/sys.c

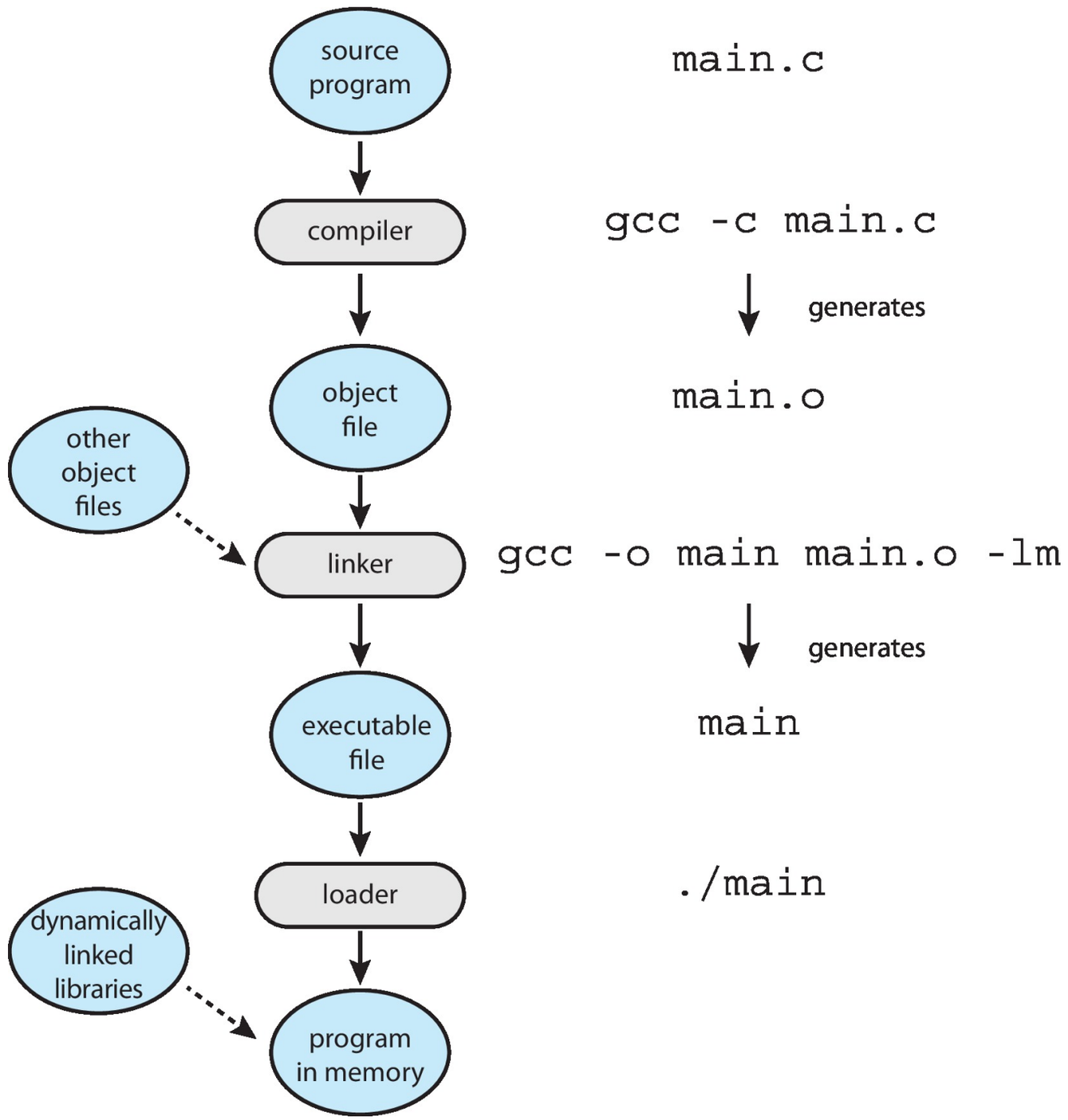
```
#undef __SYSCALL
#define __SYSCALL(nr, sym)      [nr] = (syscall_fn_t)__arm64_##sym,

const syscall_fn_t sys_call_table[__NR_syscalls] = {
        [0 ... __NR_syscalls - 1] = (syscall_fn_t)sys_ni_syscall,
#include <asm/unistd.h>
};
```

```
#define __NR_setxattr 5
__SYSCALL(__NR_setxattr, sys_setxattr)
#define __NR_lsetxattr 6
__SYSCALL(__NR_lsetxattr, sys_lsetxattr)
#define __NR_fsetxattr 7
__SYSCALL(__NR_fsetxattr, sys_fsetxattr)
#define __NR_getxattr 8
__SYSCALL(__NR_getxattr, sys_getxattr)
#define __NR_lgetxattr 9
__SYSCALL(__NR_lgetxattr, sys_lgetxattr)
#define __NR_fgetxattr 10
__SYSCALL(__NR_fgetxattr, sys_fgetxattr)
#define __NR_listxattr 11
__SYSCALL(__NR_listxattr, sys_listxattr)
```

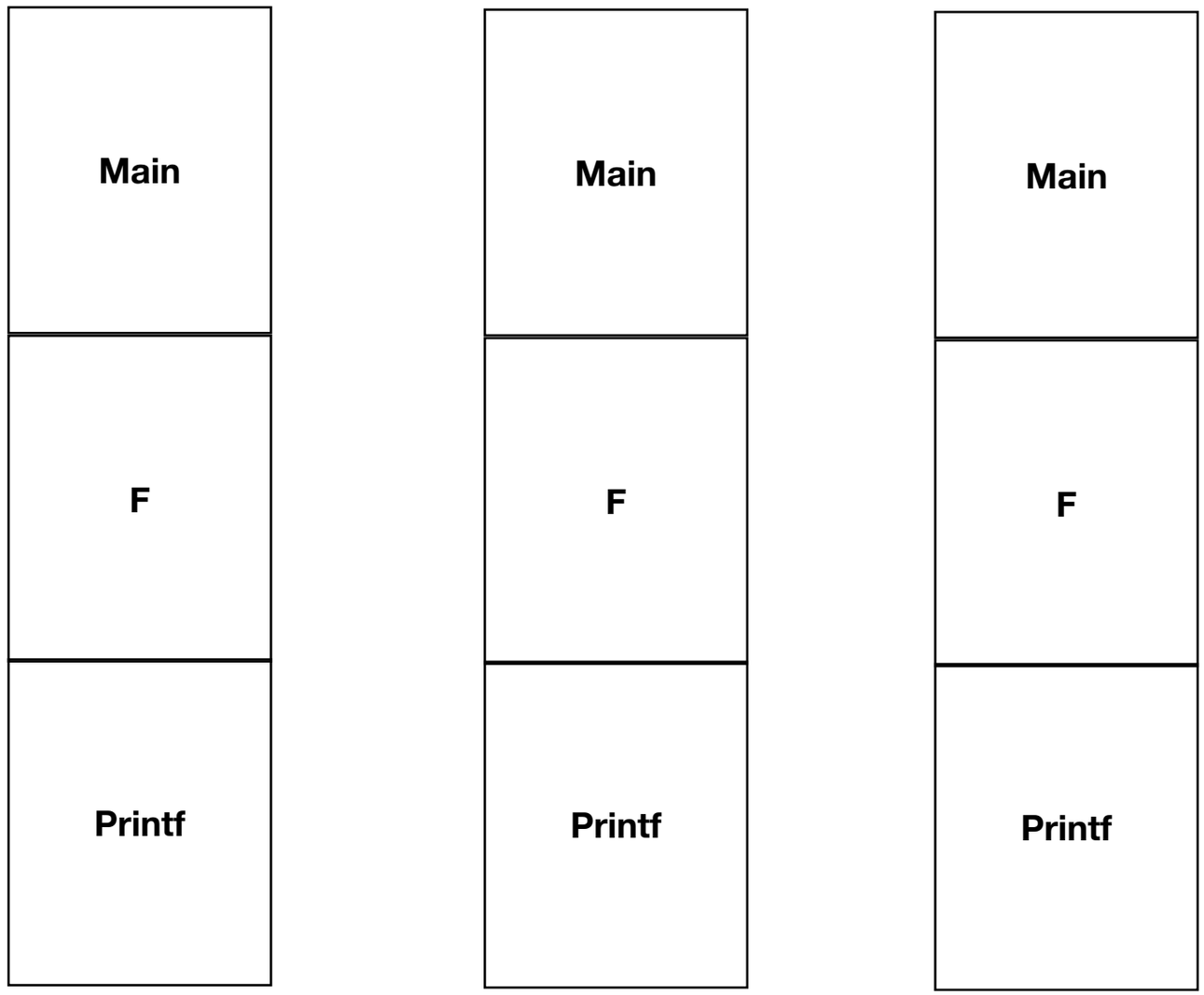
Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications

The Role of the Linker and Loader

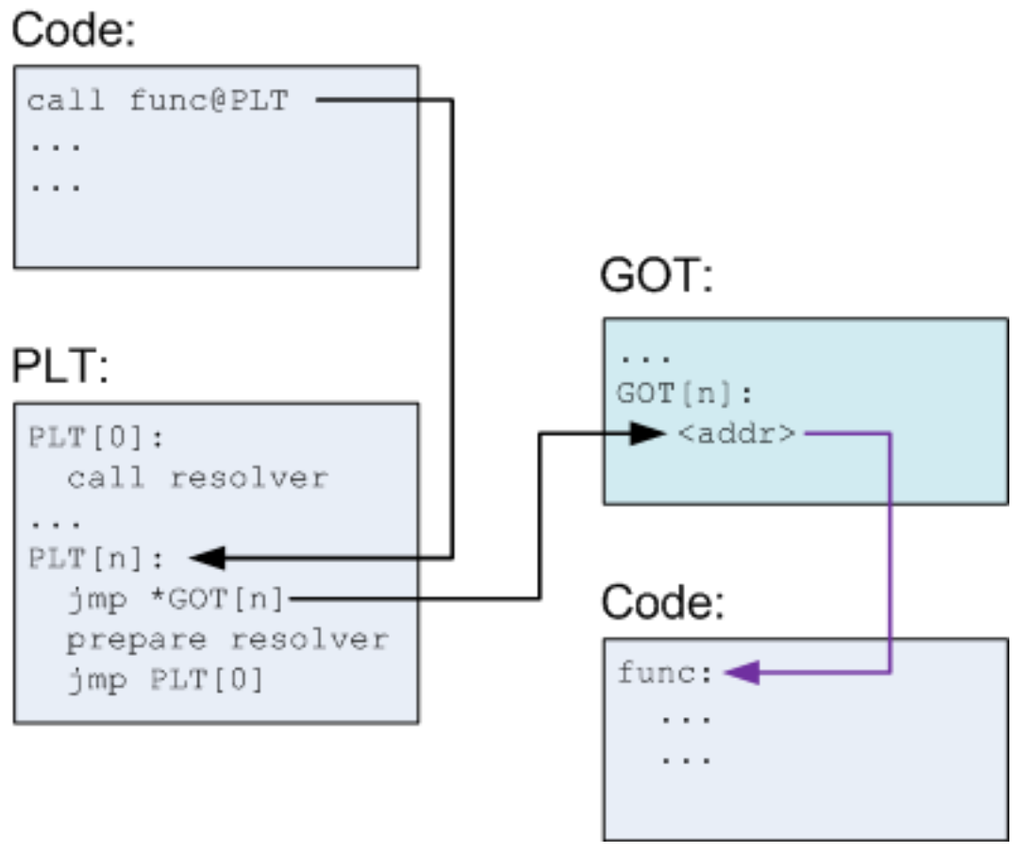


Static Linking: in memory



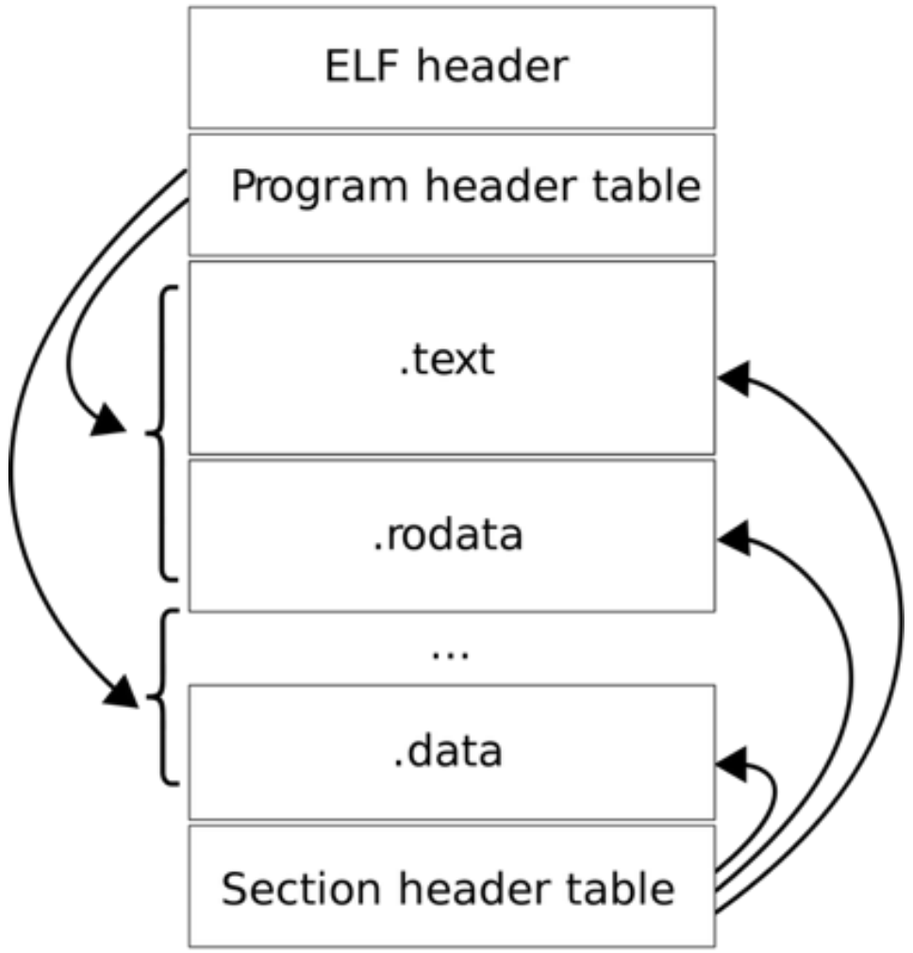
Dynamic linking

- Code -> PLT -> GOT
- Lazy binding



ELF binary basics

- What's main（a.out）
  - Executable and Linkable Format - ELF
  - Program header table and section header table
    - For Linker and Loader
  - .text: code
  - .rodata: initialized read-only data
  - .data: initialized data
  - .bss: uninitialized data
- Quiz
  - Where does static variable go?
  - Static const?



ELF binary basics

- Dump all sections
  - *readelf -S a.out*

```
Wenbos-MacBook-Pro:c-hello wenbo$ readelf -S a.out
There are 31 section headers, starting at offset 0x1708:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000 00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS        0000000000000238 00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE            0000000000000254 00000254
       0000000000000020  0000000000000000   A       0     0     4

  [13] .text             PROGBITS        0000000000000550 00000550
       00000000000001a2  0000000000000000  AX       0     0     16
  [14] .fini             PROGBITS        00000000000006f4 000006f4
       0000000000000009  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS        0000000000000700 00000700
       0000000000000010  0000000000000000   A       0     0     4

  [23] .data             PROGBITS        0000000000002000 00001000
       0000000000000010  0000000000000000  WA       0     0     8
  [24] .tm_clone_table   PROGBITS        0000000000002010 00001010
       0000000000000000  0000000000000000  WA       0     0     8
  [25] .bss              NOBITS          0000000000002010 00001010
       0000000000000001  0000000000000000  WA       0     0     1
```

Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure – MS-DOS
  - Monolithic -- UNIX
  - Layered – an abstrcation
  - Microkernel -Mach

Separate Policy and Mechanism

- Policy: **which** question
  - Which process should the process to be switched
- Mechanism: **how** question
  - How does an operating system performs a context switch

- Real world example – Door locks
  - Policy – which has permission
  - Mechanism – how to enforce
- Advantages & Disadvantages

What have we studied so far

- Computer architecture
- OS overview
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

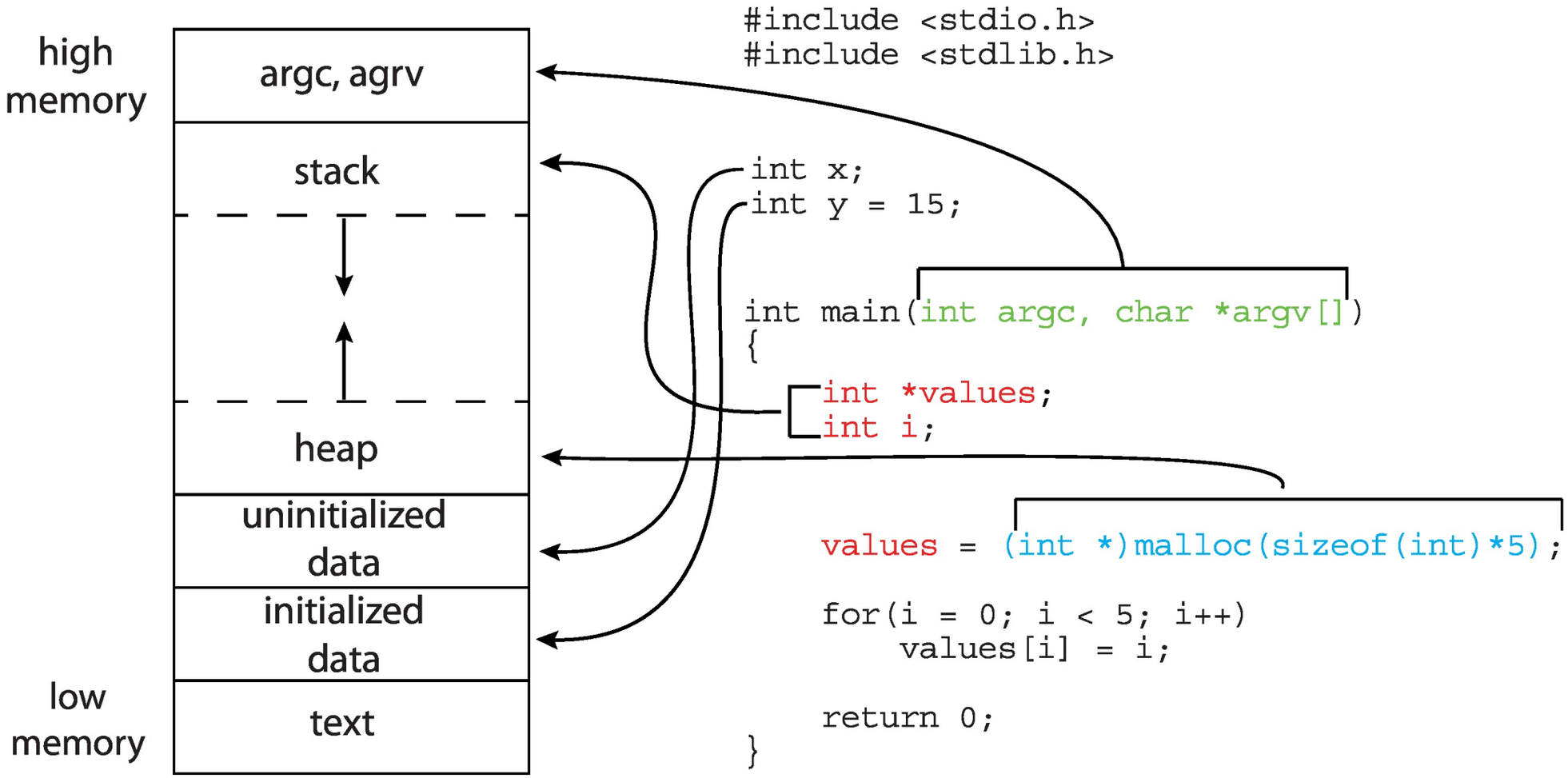
Process Concept

- A process is a program in execution (execution unit)
  - program: passive entity (bytes stored on disk as part of an executable file)
  - becomes a process when it's loaded in memory
- Multiple processes can be associated to the same program
  - on a shared server each user may start an instance of the same application (e.g., a text editor, the Shell)
- A running system consists of multiple processes
  - OS processes, user processes
- "job" and "process" are used interchangeably in OS texts

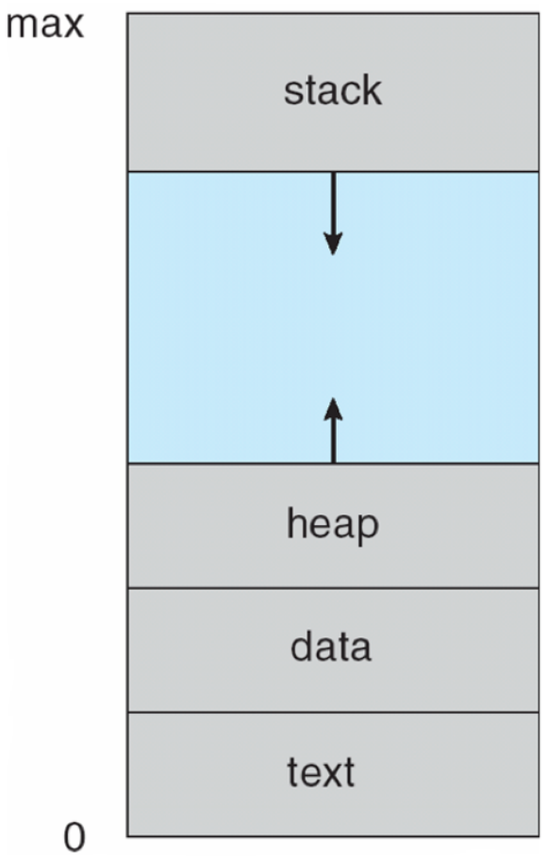
Process Concept

- Process =
  - code (also called the text)
    - initially stored on disk in an executable file
  - program counter
    - points to the next instruction to execute (i.e., an address in the code)
  - content of the processor's registers
  - a runtime stack
  - a data section
    - global variables (.bss and .data in x86 assembly)
  - a heap
    - for dynamically allocated memory (malloc, new, etc.)

Memory Layout of a C Program

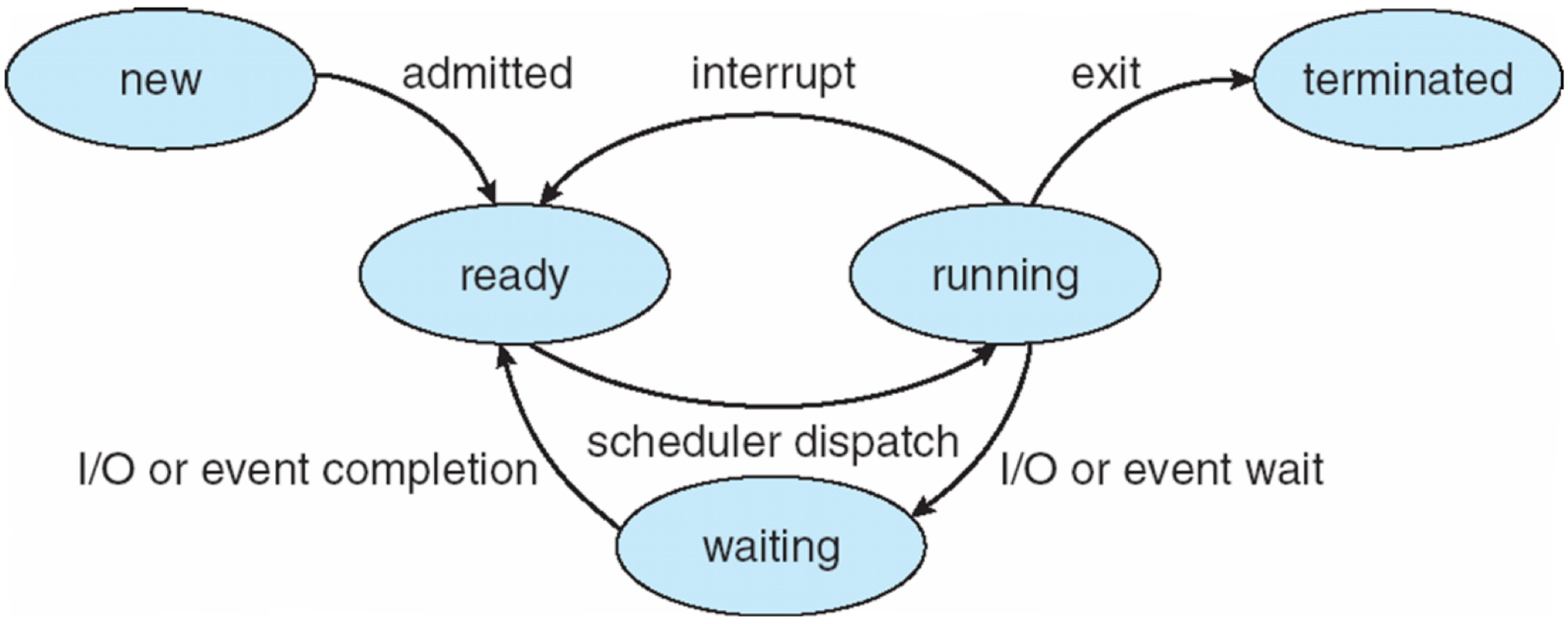


Process in Memory



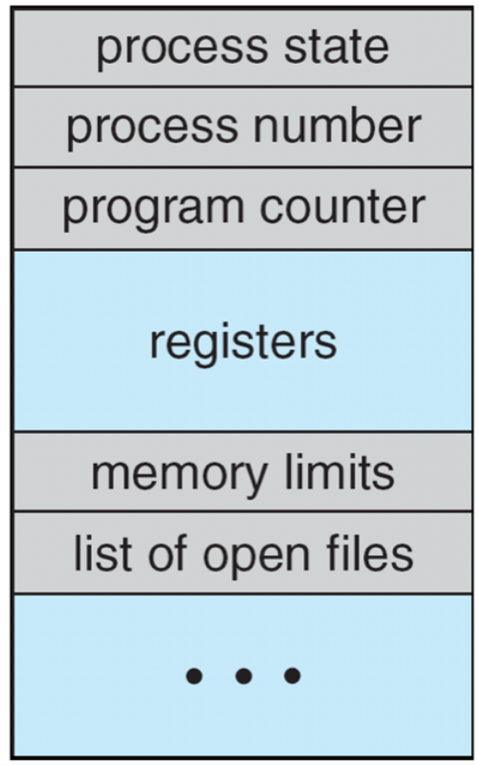

```
os@os:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 2752536                            /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 2752536                            /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 2752536                            /bin/cat
0108d000-010ae000 rw-p 00000000 00:00 0                                  [heap]
7f3b4c98d000-7f3b4d34c000 r--p 00000000 08:01 3284766                    /usr/lib/locale/locale-archive
7f3b4d34c000-7f3b4d50c000 r-xp 00000000 08:01 2102132                    /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d50c000-7f3b4d70c000 ---p 001c0000 08:01 2102132                    /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d70c000-7f3b4d710000 r--p 001c0000 08:01 2102132                    /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d710000-7f3b4d712000 rw-p 001c4000 08:01 2102132                    /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d712000-7f3b4d716000 rw-p 00000000 00:00 0
7f3b4d716000-7f3b4d73c000 r-xp 00000000 08:01 2102104                    /lib/x86_64-linux-gnu/ld-2.23.so
7f3b4d900000-7f3b4d925000 rw-p 00000000 00:00 0
7f3b4d93b000-7f3b4d93c000 r--p 00025000 08:01 2102104                    /lib/x86_64-linux-gnu/ld-2.23.so
7f3b4d93c000-7f3b4d93d000 rw-p 00026000 08:01 2102104                    /lib/x86_64-linux-gnu/ld-2.23.so
7f3b4d93d000-7f3b4d93e000 rw-p 00000000 00:00 0
7ffff3ba3000-7ffff3bc4000 rw-p 00000000 00:00 0                          [stack]
7ffff3bcd000-7ffff3bd0000 r--p 00000000 00:00 0                          [vvar]
7ffff3bd0000-7ffff3bd2000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Process State



Process Control Block (PCB)

- In the kernel, each process is associated with a **process control block**
  - process number (pid)
  - process state
  - **program counter (PC)**
  - CPU registers
  - CPU scheduling information
  - memory-management data
  - accounting data
  - I/O status



- Linux's PCB is defined in struct task_struct:
  http://lxr.linux.no/linux+v3.2.35/include/linux/sched.h#L1221

Process Creation

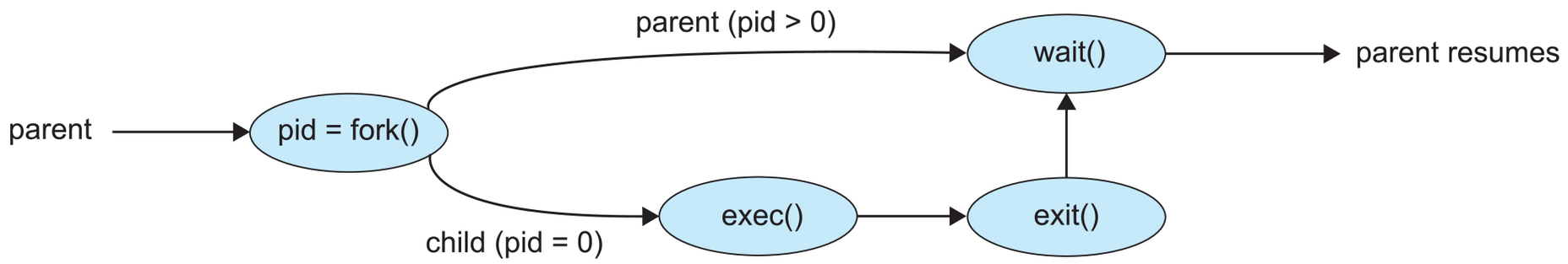
- **fork()** creates a new process
- The child is is a copy of the parent, but...
  - It has a different pid (and thus ppid)
  - Its resource utilization (so far) is set to 0
- fork() returns the child's pid to the parent, and 0 to the child
  - Each process can find its own pid with the getpid() call, and its ppid with the getppid() call
- Both processes continue execution after the call to fork()
- **exec()** overwrites the process' address space with a new program
- **wait()** waits for the child(ren) to terminate

C Program Forking Separate Process

```c
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
    pid_t  pid;
    pid = fork();                           /* fork another process */
    if (pid < 0) {                          /* error occurred while forking */
        fprintf(stderr, "Fork Failed");
        return -1;
    } else if (pid == 0) {                  /* child process */
        execlp("/bin/ls", "ls", NULL);
    } else {                                /* parent process */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```

Process Terminations

- A process terminates itself with the exit() system call
  - This call takes as argument an integer that is called the process' exit/return/error code
- All resources of a process are deallocated by the OS
  - physical and virtual memory, open files, I/O buffers, ...
- A process can cause the termination of another process
  - Using something called "signals" and the kill() system call



Zombie - They're dead.. but alive!

- When a child process terminates
  - Remains as a zombie in an "undead" state
  - Until it is "reaped" (garbage collected) by the OS
- Rationale:
  - The parent may still need to place a call to wait(), or a variant, to retrieve the child's exit code
- The OS keeps zombies around for this purpose
  - They're not really processes, they do not consume ~~resources~~ CPU
  - They only consume a slot in memory
    - Which may eventually fill up and cause fork() to fail

Getting rid of zombies

- A zombie lingers on until:
  - its parent has called wait() for the child, or
  - its parent dies
- It is bad practice to leave zombies around unnecessarily
- When a child exits, a SIGCHLD signal is sent to the parent
- A typical way to avoid zombies altogether:
  - The parent associates a handler to SIGCHLD
  - The handler calls wait()
  - This way all children deaths are “acknowledged”
  - See nozombie_example.c

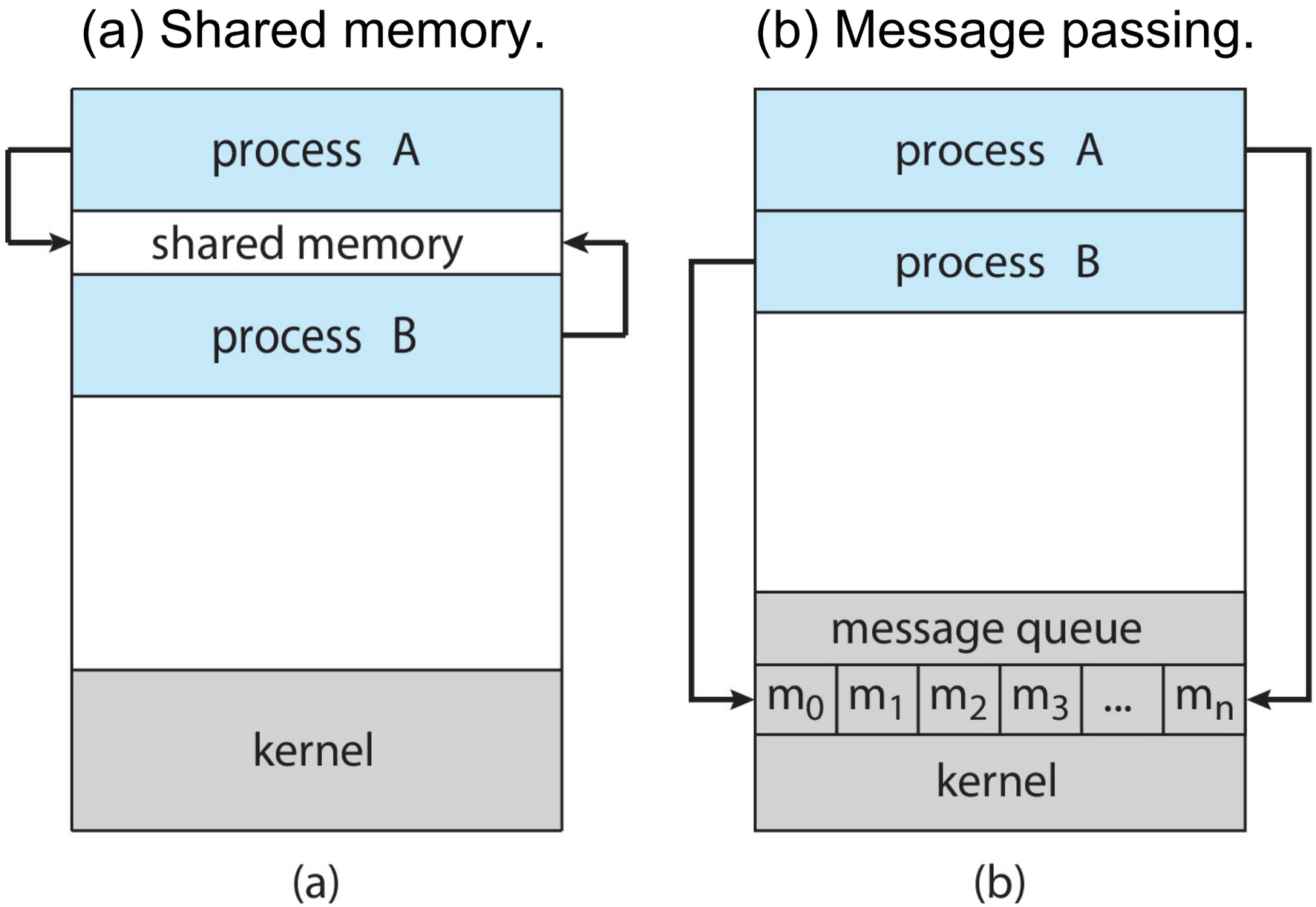
Orphans

- An orphan process is one whose parent has died
- In this case, the orphan is “adopted” by the process with pid 1
  - init on a Linux system
  - launchd on a Mac OS X system
  - Demo : orphan_example1.c
- The process with pid 1 does handle child termination with a handler for SIGCHLD that calls wait (just like in the previous slide!)
- Therefore, an orphan never becomes a zombie

What have we studied so far

- Computer architecture
- OS overview
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

IPC Communication Models



Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is an issue

Message Passing

- Processes communicate with each other by exchanging messages
  - without resorting to shared variables
- Message passing provides two operations:
  - **send** (message)
  - **receive** (message)
- Message passing may be either **blocking** or **non-blocking**
- Blocking is considered **synchronous**
  - **blocking send** has the sender block until the message is received
  - **blocking receive** has the receiver block until a message is available
- Non-blocking is considered **asynchronous**
  - **non-blocking send** has the sender send the message and continue
  - **non-blocking receive** has the receiver receive a valid message or null