Architectural Support for OS
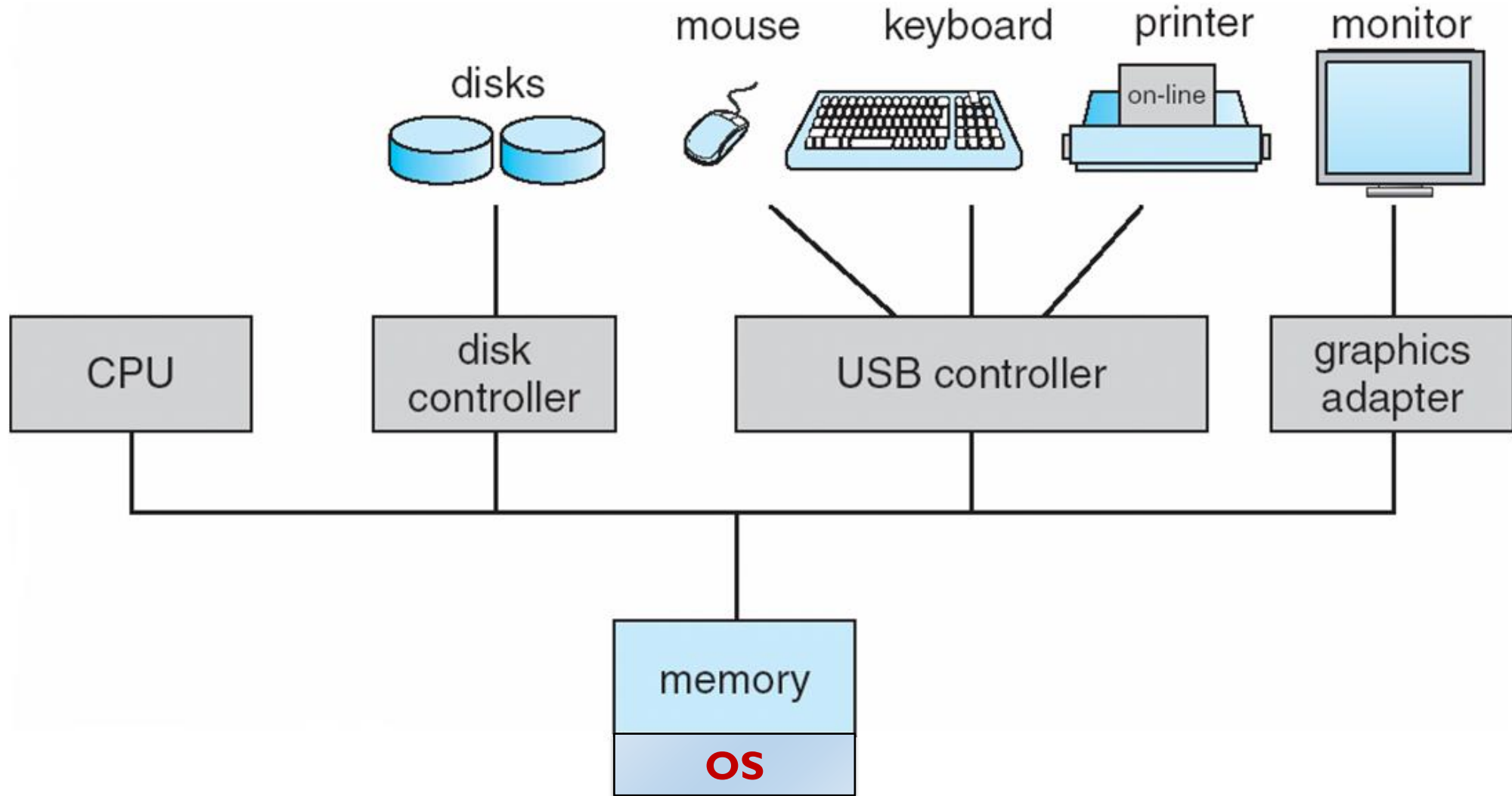
Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University
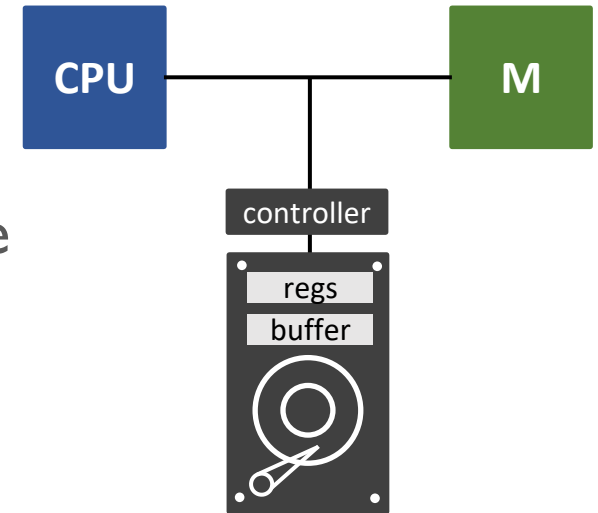
Fall 2023

Computer System Organization

Issue #1: I/O

- **How to perform I/Os efficiently?**
  - I/O devices and CPU can execute concurrently
  - Each device controller is in charge of a particular device type
  - Each device has a local buffer
  - CPU issues specific commands to I/O devices
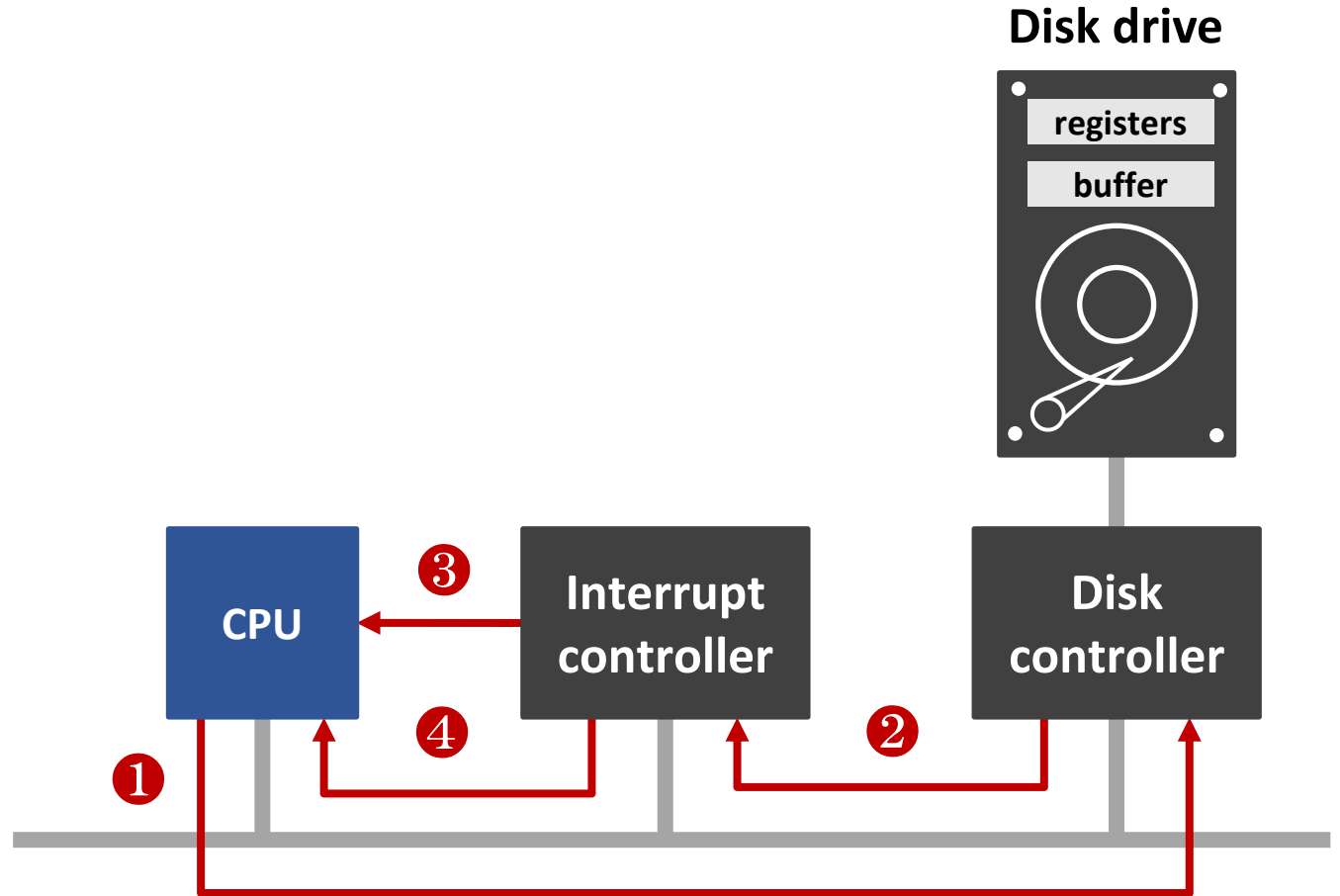 - CPU moves data between main memory and local buffers

- **CPU is a precious resource; it should be freed from time-consuming tasks**
  - Checking whether the issued command has been completed or not
  - Moving data between main memory and device buffers

Interrupts

- How does the kernel notice an I/O has finished?

  - _____

  - Hardware interrupt

"Do homework!"

"OK"

"Done?"

"No"

"Done?"

"No"

"Done?"

"Yes"

"Do homework!"

"OK"

"I am done!"

Disk drive

registers

buffer

CPU

❸

Interrupt controller

❹

Disk controller

❷

❶
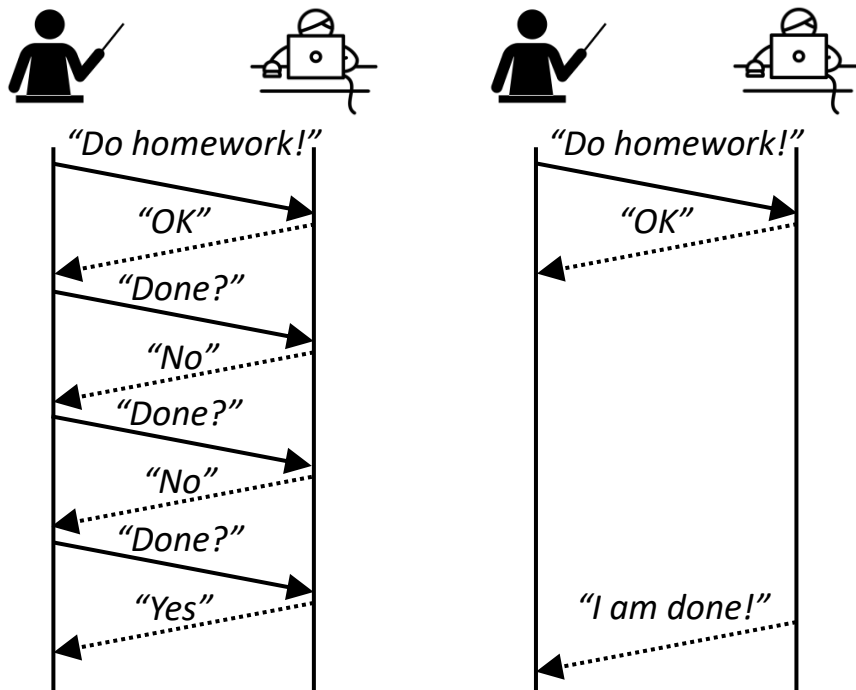
Interrupt Handling

- **Preserves the state of the CPU**
  - In a fixed location
  - In a location indexed by the device ID
  - On the system stack

- **Determines the type**
  - Polling
  - Vectored interrupt system

- **Transfers control to the interrupt service routine (ISR) or interrupt handler**

❶ *interrupt*

current instruction

next instruction

❷ *Dispatch to handler*

❸ *Execute the handler*

interrupt handler

❹ *Return from interrupt*
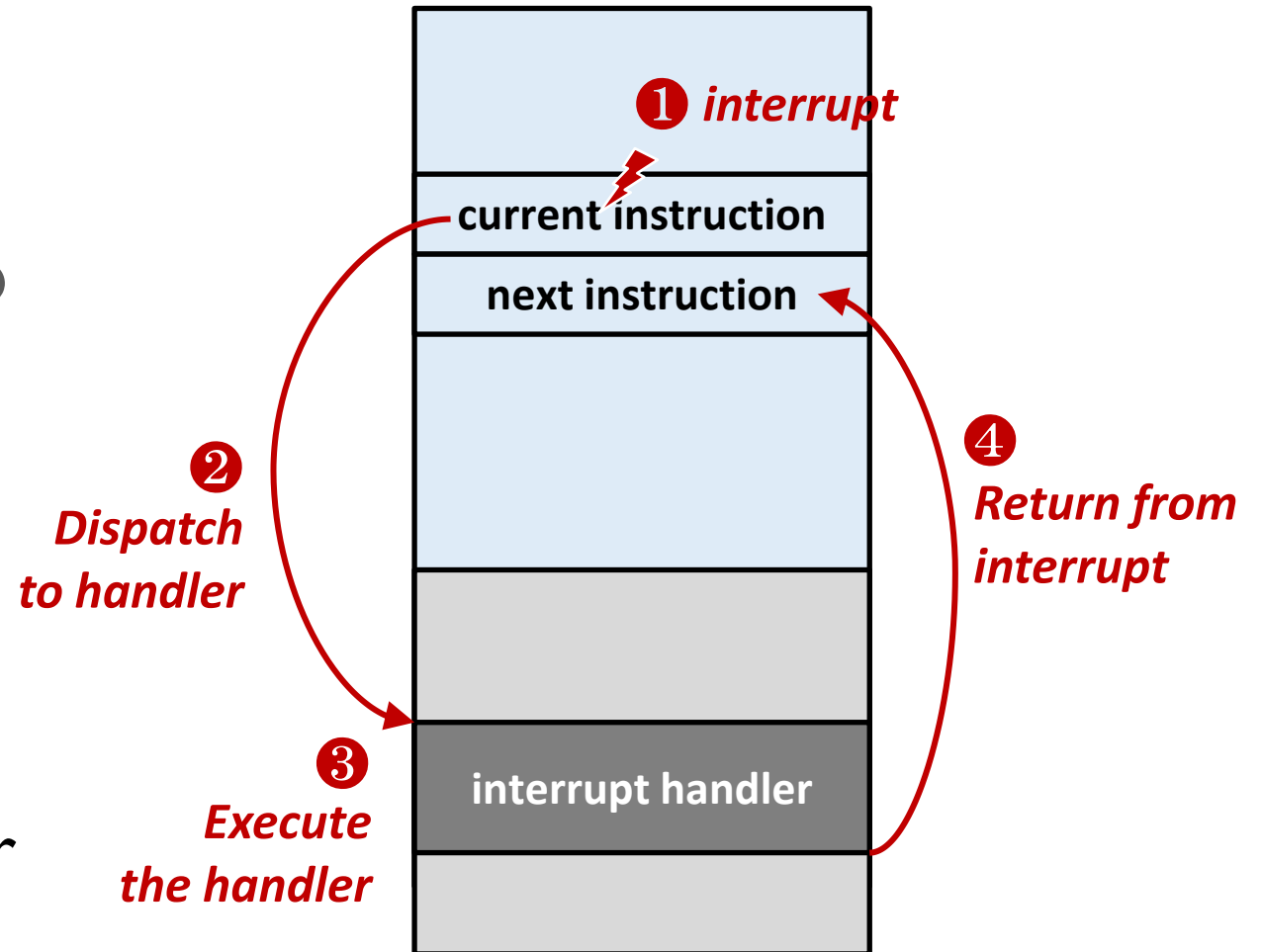
Data Transfer Modes

- _____ (PIO)
  - CPU is involved in moving data between I/O devices and memory
  - By special I/O instructions vs. by memory-mapped I/O
  - e.g., keyboard, mouse, …

- DMA (Direct Memory Access)
  - Used for high-speed I/O devices to transmit information at close to memory speeds
  - Device controller transfers blocks of data from the local buffer directly to main memory (or vice versa) without CPU intervention
  - Only an interrupt is generated per block
  - DMA controller oversees the overall data transfer

Disk I/O Example

8th Gen Intel® Core™ Processors

Intel® UHD Graphics

DDR4 2xDIMMs per Channel Up to 2666 MHz[1]

DDR4 2xDIMMs per Channel Up to 2666 MHz[1]

DMI 3.0

Intel® Z370 Chipset

SPI

SATA controller

❶ *send read command*
❷ *queue command & ack*
❸ *ack*
❹ *perform disk read*
❺ *buffer data*
❻ *notify*
❼ *DMA setup*
❽ *DMA*
❾ *interrupt*
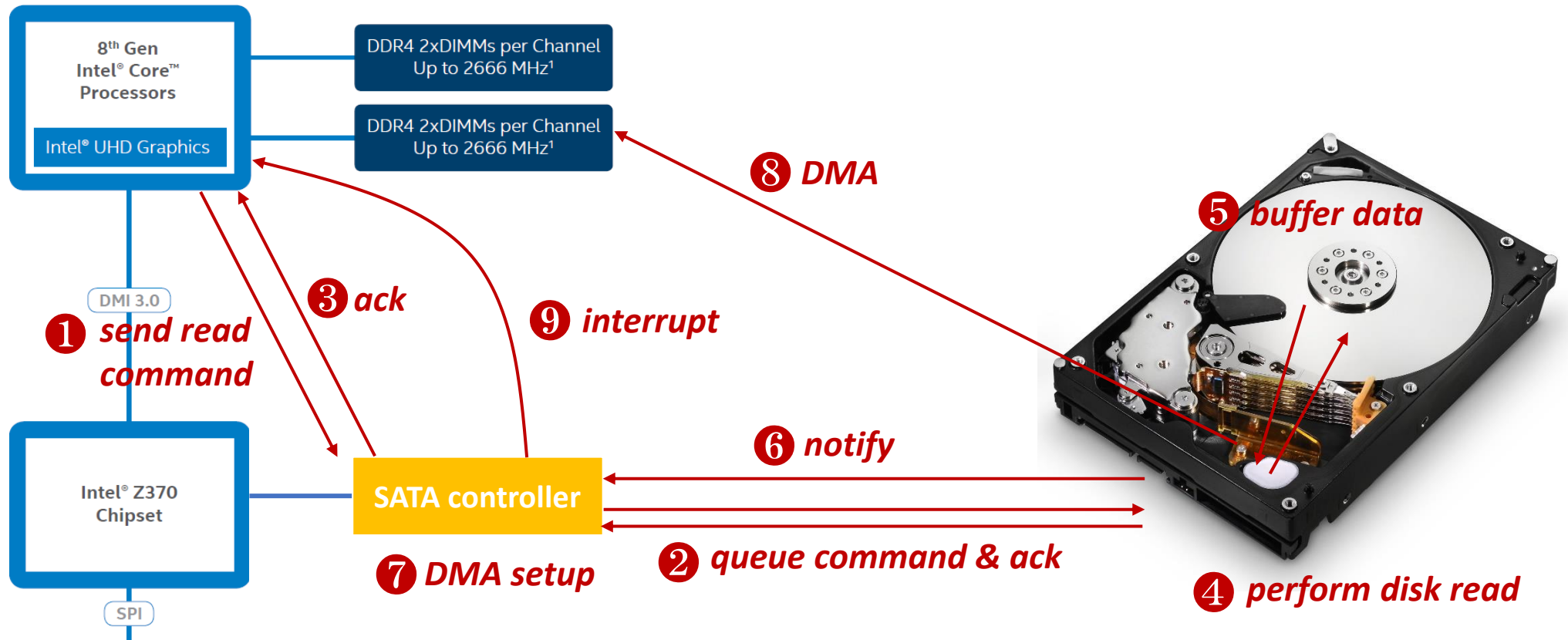
Issue #2: Protection

- **How to prevent user applications from harming the system?**
  - What if an application accesses disk drives directly?
  - What if an application executes the HLT instruction?

HLT—Halt

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F4	HLT	NP	Valid	Valid	Halt

Description

Stops instruction execution and places the processor in a HALT state.

Protected Instructions

- Protected or _____ instructions
  - The ability to perform certain tasks that cannot be done from user mode

  - Direct I/O access
    - e.g., `in` / `out` instructions in x86
  - Accessing system registers
    - Control registers
    - System table locations (e.g., interrupt handler table)
    - Setting special "mode bits", etc.
  - Memory state management
    - Page table updates, page table base address, TLB loads, etc.
 - HLT instruction in x86

CPU Modes of Operation

- **Kernel mode vs. user mode**

  - How does the CPU know if a protected instruction can be executed?

  - The architecture must support at least two modes of operation:
    kernel and user mode

    - 4 privilege levels in x86_64:   Ring 0 > 1 > 2 > 3
    - 4 privilege levels in ARM: EL3 > EL2 > EL1 > EL0
    - 3 privilege levels in RISC-V: Machine > Supervisor > User

  - Mode is set by a status bit in a protected register

    - IA-32: Current Privilege Level (CPL) in CS register
    - ARM: Mode field in CPSR register

- **Protected instructions can only be executed in the corresponding privileged level**
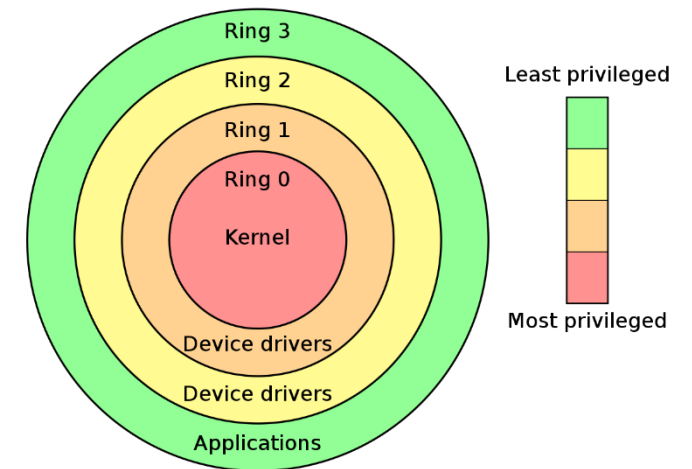
Issue #3: Servicing Requests

- How to ask services to the OS?

  - How can an application read a file if it cannot access disk drives?

  - Even a "`printf()`" call requires hardware access

  - User programs must ask the OS to do something privileged

System Calls

- OS defines a set of system calls
  - Programming interface to the services provided by OS
  - OS protects the system by rejecting illegal requests
  - OS may impose a quota on a certain resource
  - OS may consider fairness while sharing a resource

- A system call is a _____ procedure call
  - System call routines are in the OS code
  - Executed in the kernel mode
  - On entry, user mode → kernel mode switch
  - On exit, CPU mode is changed back to the user mode

System Calls Example

- POSIX vs. Win32

Category	POSIX	Win32	Description
Process Management	fork	CreateProcess	Create a new process (CreateProcess = fork + exec)
	waitpid	WaitForSingleObject	Wait for a process to exit
	execve	(none)	Execute a new program
	exit	ExitProcess	Terminate execution
	kill	(none)	Send a signal
File Management	open	CreateFile	Create a file or open an existing file
	close	CloseHandle	Close a file
	read	ReadFile	Read data from a file
	write	WriteFile	Write data to a file
	lseek	SetFilePointer	Move the file pointer
	stat	GetFileAttibutesEx	Get various file attributes
	chmod	(none)	Change the file access permission
File System Management	mkdir	CreateDirectory	Create a new directory
	rmdir	RemoveDirectory	Remove an empty directory
	link	(none)	Make a link to a file
	unlink	DeleteFile	Destroy an existing file
	chdir	SetCurrentDirectory	Change the current working directory
	mount	(none)	Mount a file system

Exceptional Events

- ## Interrupts
  - Generated by hardware devices
    - Triggered by a signal in INTR or NMI pins (x86_64)
  - Asynchronous

- ## Exceptions
  - Generated by software executing instructions
    - Unintentional: Divide-by-zero, …
    - Intentional: **syscall** instruction in x86_64 or **ecall** instruction in RISC-V
  - Synchronous
  - Exception handling is same as interrupt handling

Exceptions in x86_64

- ■ _____
  - Intentional
  - System call traps, breakpoint traps, special instructions, …
  - Return control to "next" instruction
- ■ Faults
  - Unintentional but possibly recoverable
  - Page faults (recoverable), protection faults (unrecoverable), …
  - Either re-executing faulting ("current") instruction or abort
- ■ _____

  - Unintentional and unrecoverable (parity error, machine check, …)
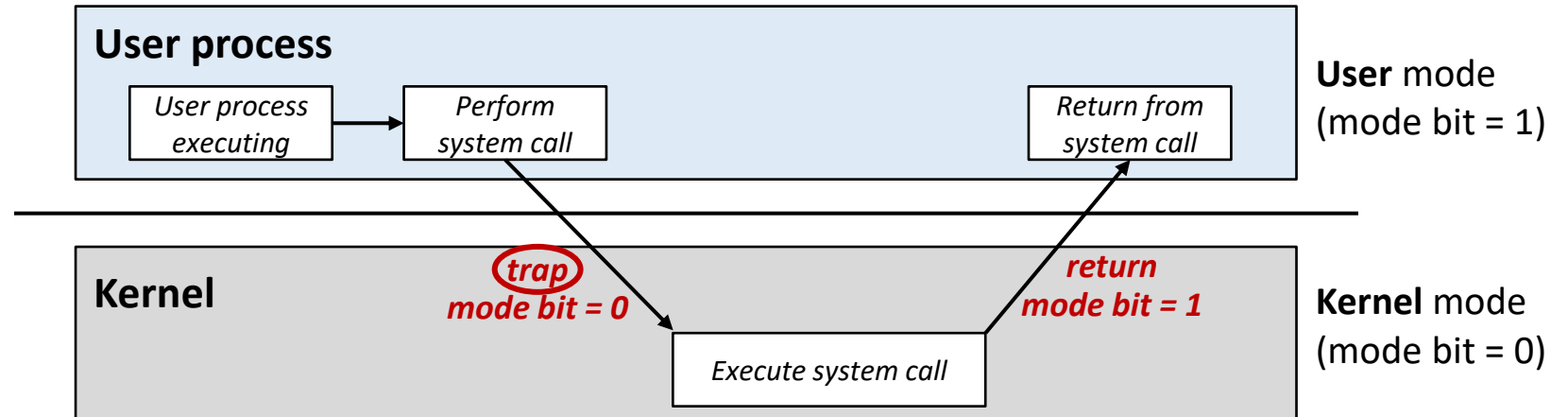 - Abort the current program

OS Trap

- There must be a special "trap" instruction that:
  - Causes an exception, which invokes a kernel handler
  - Passes a parameter indicating which system call to invoke
  - Saves caller's state (registers, mode bits)
  - Returns to user mode when done with restoring its state
  - OS must verify caller's parameters (e.g., pointers)

Examples:

SYSCALL instruction (x86_64)

ECALL instruction (RISC-V)

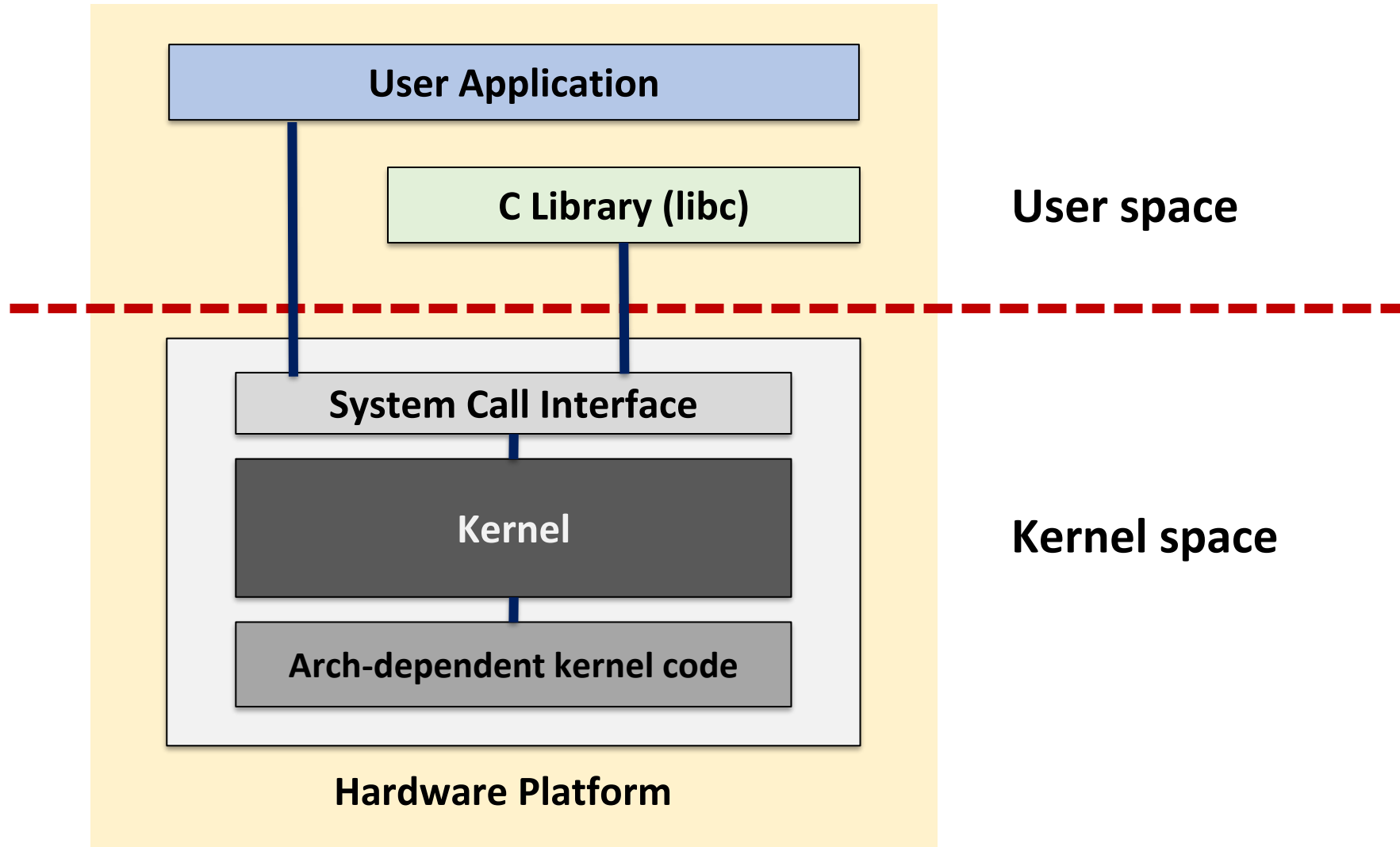| User process			User mode (mode bit = 1)
User process executing → Perform system call		Return from system call	

Kernel		return mode bit = 1	Kernel mode (mode bit = 0)
trap mode bit = 0	Execute system call		

Implementing System Calls

- `count = read(fd, buf, 512);`

User space

```
main: mv    a0, <fd>        ❶ store fd to a0
      mv    a1, <buf>       ❷ store buf to a1
      li    a2, 512         ❸ store 512 to a2
      call  read
      ...                              ❿ return to caller
```

❹ call read()

User program

```
read: li    a7, 5          ❺ store read syscall # to a7
      ecall
      ret
```

❻ trap to the kernel

❾ return from trap

C library

Kernel space

| Dispatch | ❼ find read handler | | ❽ jump to handler | read syscall handler |
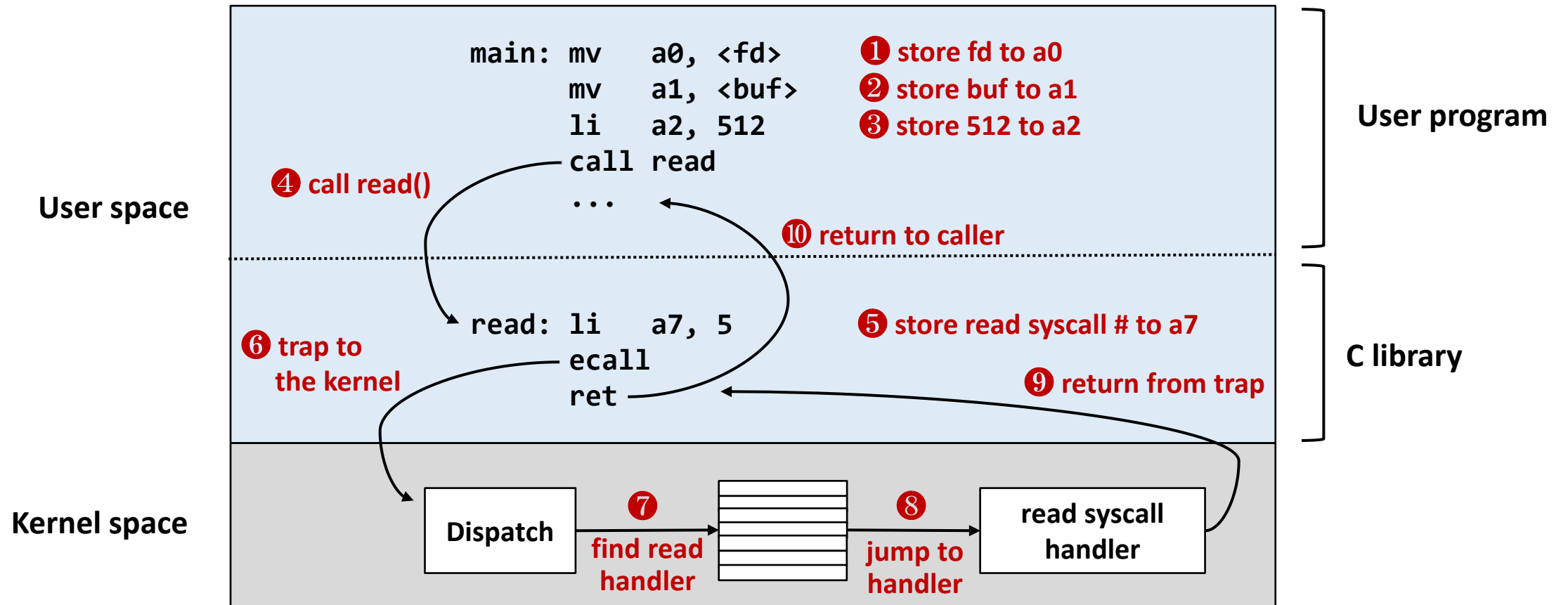
Typical (Monolithic) OS Structure

Issue #4: Control

- How to take the control of the CPU back from the running program?

- Cooperative approach
  - Each application periodically transfers the control of the CPU to OS by calling various system calls
  - A special system call can be used just to release the CPU (e.g., `yield()`)
  - Can be used when _____
  - What if a process ends up in an infinite loop?
   (due to a bug or with a malicious intent)

Timers

- **A non-cooperative approach**
  - Use a hardware timer that generates a periodic interrupt
  - The timer interrupt transfers control back to OS

- **The OS preloads the timer with a time to interrupt**
  - 10ms for Linux 2.4, 1ms for Linux 2.6, 4ms for Linux 5.5
  - 10ms for xv6

- **The timer is privileged**
  - Only the OS can load it

Issue #5: Memory Protection

- **How can we protect memory?**
  - Unlike the other hardware resources, we allow applications to access memory directly without OS intervention. Why?

- **From malicious users:**
  OS must protect user applications from each other

- **For integrity and security:**
  OS must also protect itself from user applications

Simplest Memory Protection

- Use base and limit registers
- Base and limit registers are loaded by OS before starting an application
- CPU generates an exception if the memory address is out of bound
- Can be used in a simple embedded environment
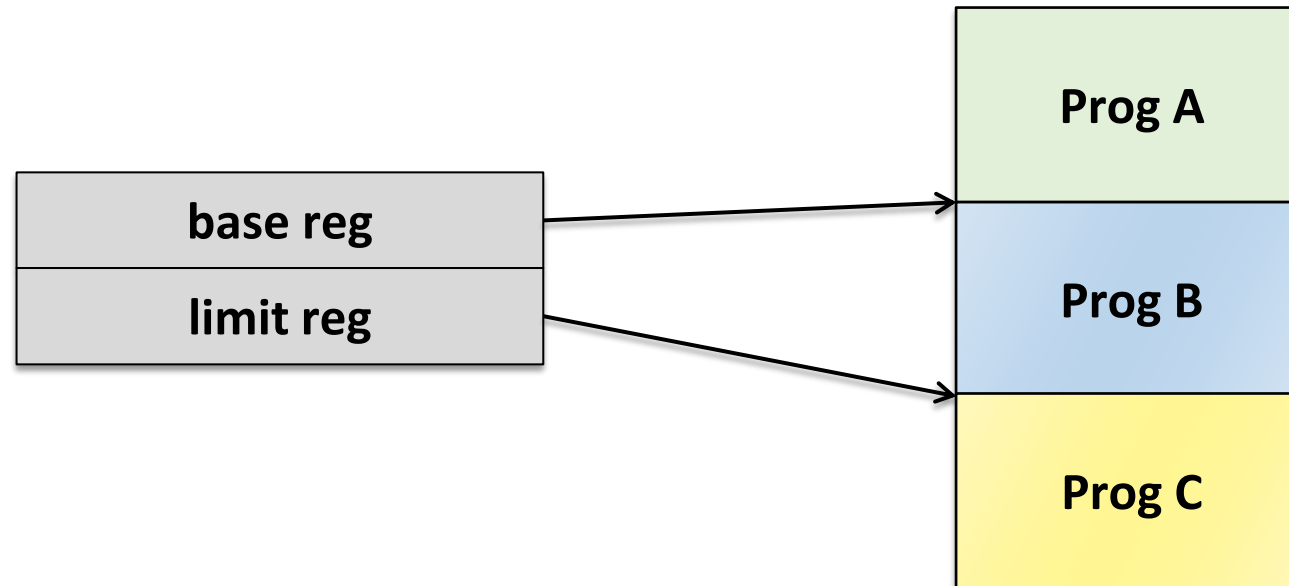
Virtual Memory

- **Modern CPUs are equipped with memory management hardware**
  - MMU (Memory Management Unit)

- **MMU provides more sophisticated memory protection mechanisms**
  - Virtual memory
  - Paging: page tables, page protection, TLBs
  - Segmentation: segment tables, segment protection

- **Manipulation of MMU is a privileged operation**

Issue #6: Synchronization

- How to coordinate concurrent activities?
  - What if multiple concurrent streams access the shared data?
  - Interrupt can occur at any time and may interfere with the interrupted code

```
LOAD R1 ← Mem[X]

ADD R1 ← R1, #1

                          LOAD R1 ← Mem[X]

                          ADD R1 ← R1, #1

                          STORE R1 → Mem[X]


STORE R1 → Mem[X]
```

- Turn off/on interrupts?

Atomic Instructions

■ Requires special atomic instructions

- Read-Modify-Write (e.g., INC, DEC)

- Test-and-Set

- Compare-and-Swap

- LOCK prefix in x86_64

- LL (Load Locked) & SC (Store Conditional) in MIPS

■ RISC-V "A" extension

- LR (Load Reserved) & SC (Store Conditional) instructions

- AMO (Atomic Memory Operation) instructions
  - Swap, integer add, bitwise AND/OR/XOR, integer max/min (signed/unsigned)

Summary

- The functionality of an OS is limited by architectural features
  - Multiprocessing on MS-DOS/8086?

- The structure of an OS can be simplified by architectural support
  - Interrupt, DMA, atomic instructions, etc.

- Most proprietary OSes were developed with the certain architecture in mind
  - SunOS/Solaris for SPARC
  - IBM AIX for Power/PowerPC
  - HP-UX for PA-RISC