CSC 256/456: Operating Systems

System Calls, Kernel Mode, and Process Implementation

John Criswell
University of Rochester

Today

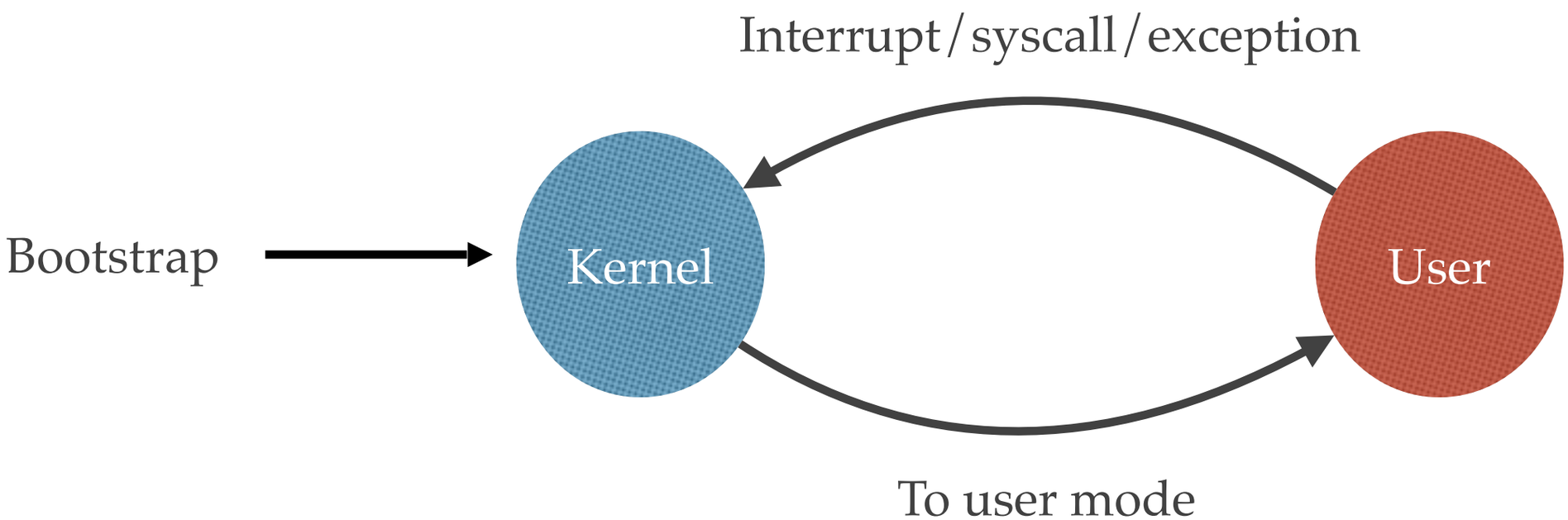
- User Mode and Kernel Mode
- System calls and the interrupt interface
- Basic I/O
- Context switches and the scheduling process

System Protection

- User programs typically not trusted
  - May use unfair amount of resources
  - May maliciously cause other programs or OS to fail
- System provides two hardware modes:
  - User mode: Some access to hardware resources restricted
  - Kernel mode: Full access to hardware resources

Transition between User/Kernel Mode

- When does the machine run in kernel mode?
  - after machine boot
  - interrupt handler
  - system call
  - exception



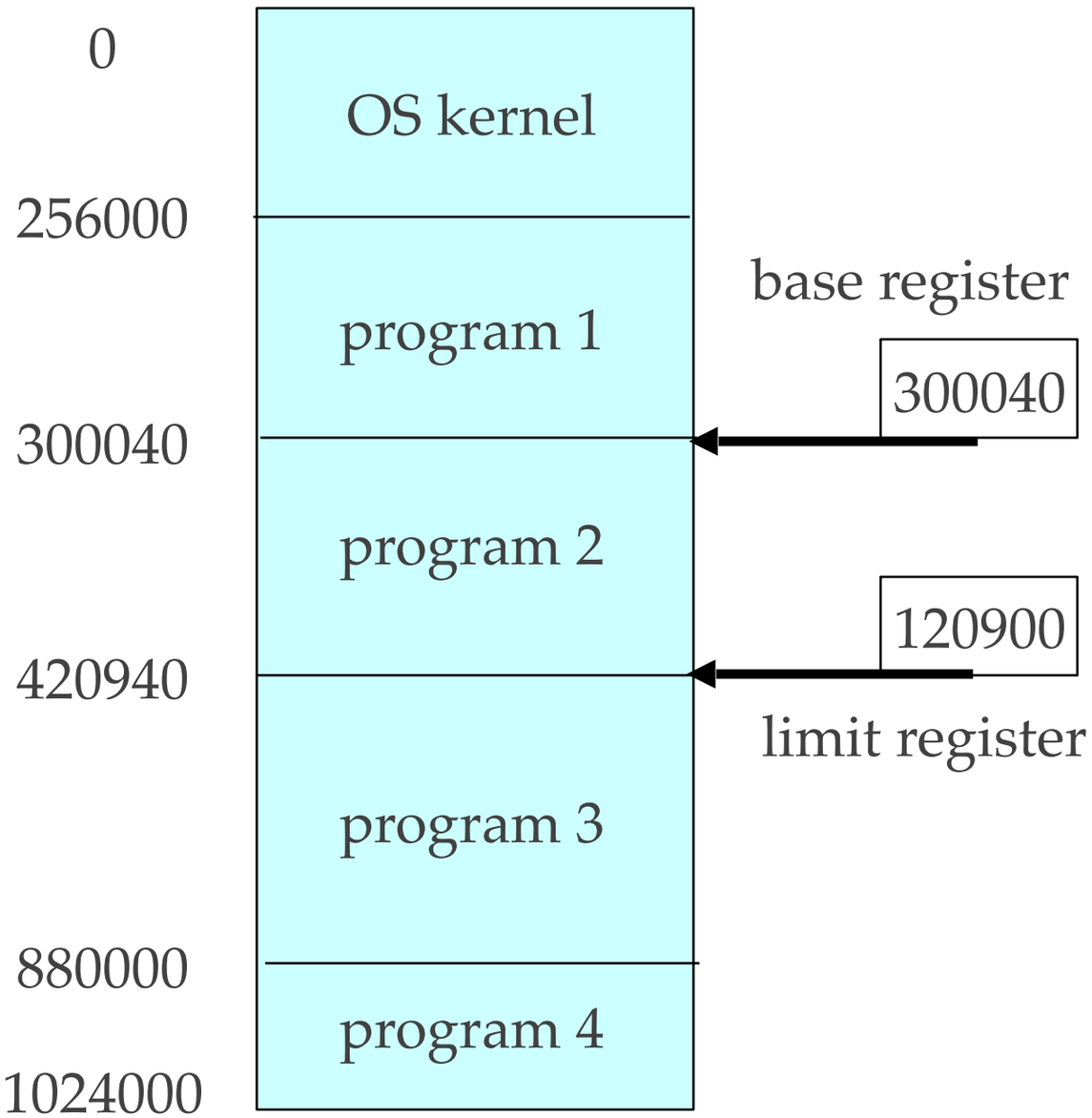
CPU Protection

- Goal of CPU protection
  - A user program can't hold the CPU for ever
- Timer – interrupts computer after specified period to ensure the OS kernel maintains control
- Timer is decremented every clock tick
- When timer reaches the value 0, an interrupt occurs
- CPU time sharing is implemented in the timer interrupt

Memory Protection

- Goal of memory protection?
  - A user program can't use arbitrary amount of memory
  - A user program can't access data belonging to the operating system or other user programs
- How to achieve memory protection?
  - Indirect memory access
    - Memory access with a virtual address which needs to be translated into physical address
  - Add two registers that determine the range of legal addresses a program may access:
    - Base register – holds the smallest legal physical memory address
    - Limit register – contains the size of the range
    - Memory outside the defined range is protected

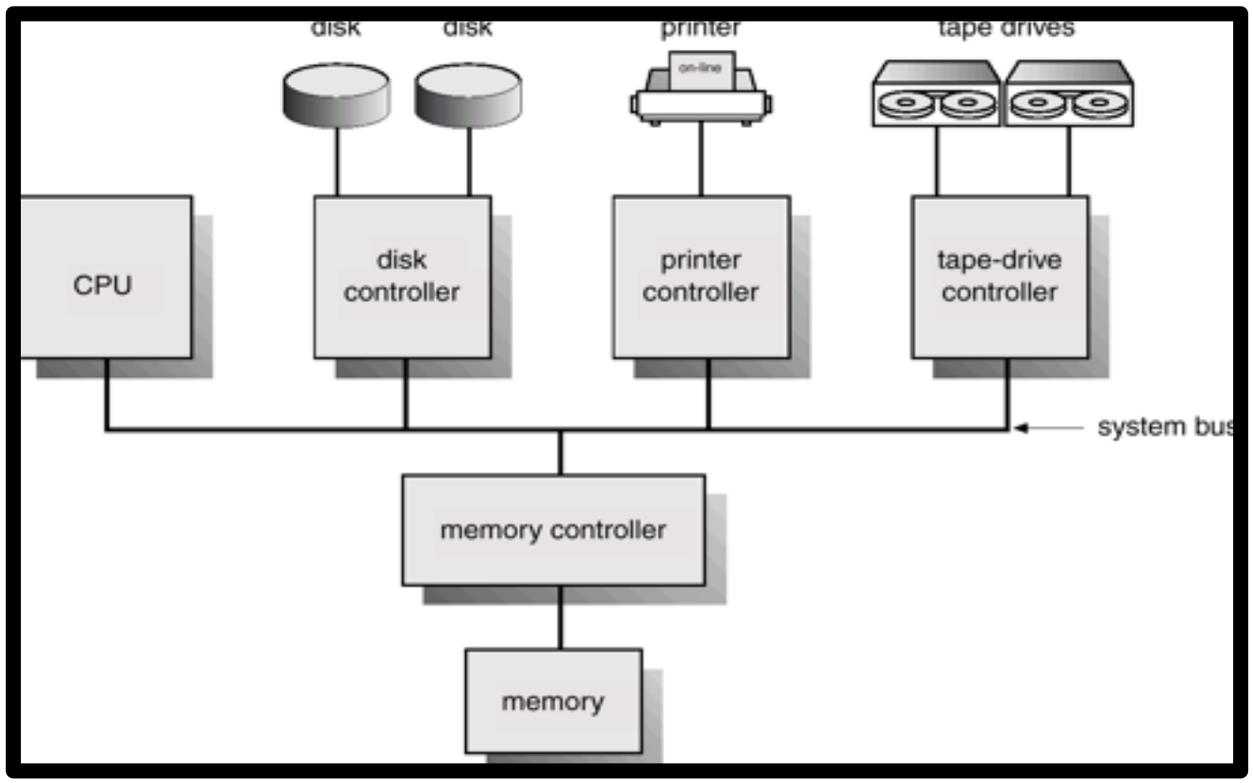
Hardware Address Protection



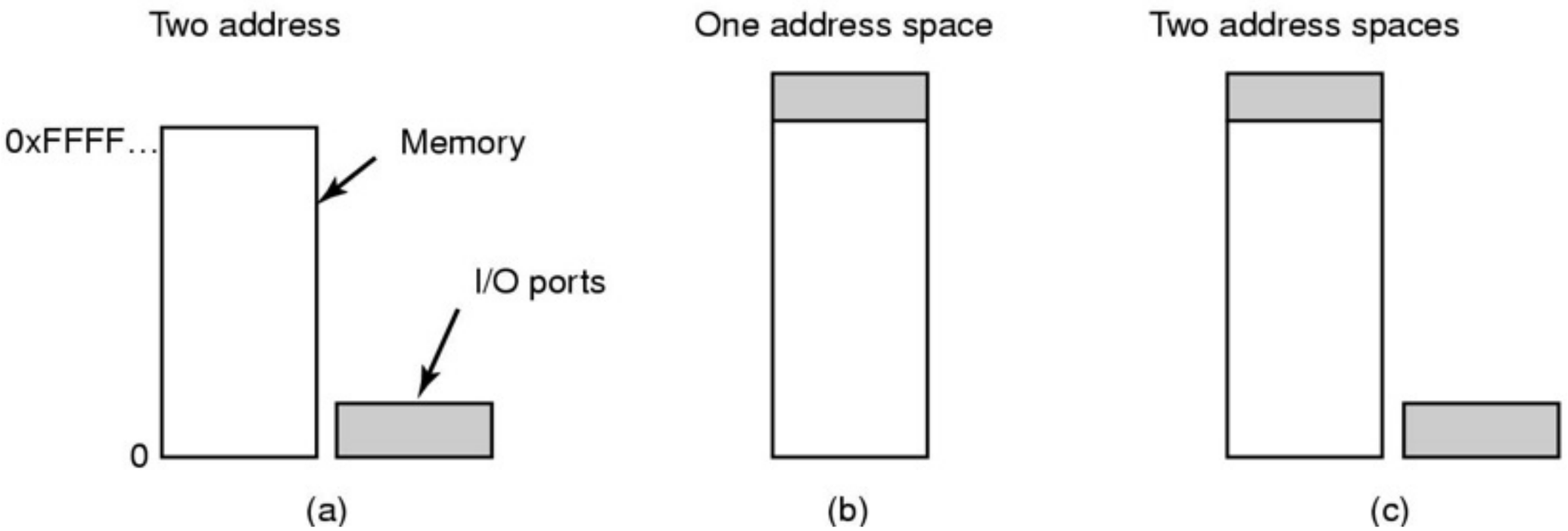
- Address of each memory address is checked against "base" and "base+limit"
- Trap to the OS kernel if it falls outside of the range (an exception)

I/O Device Controllers

- I/O devices have both mechanical component & electronic component
- The electronic component is the device controller
  - It contains control logic, command registers, status registers, and on-board buffer space



I/O Ports & Memory-Mapped I/O

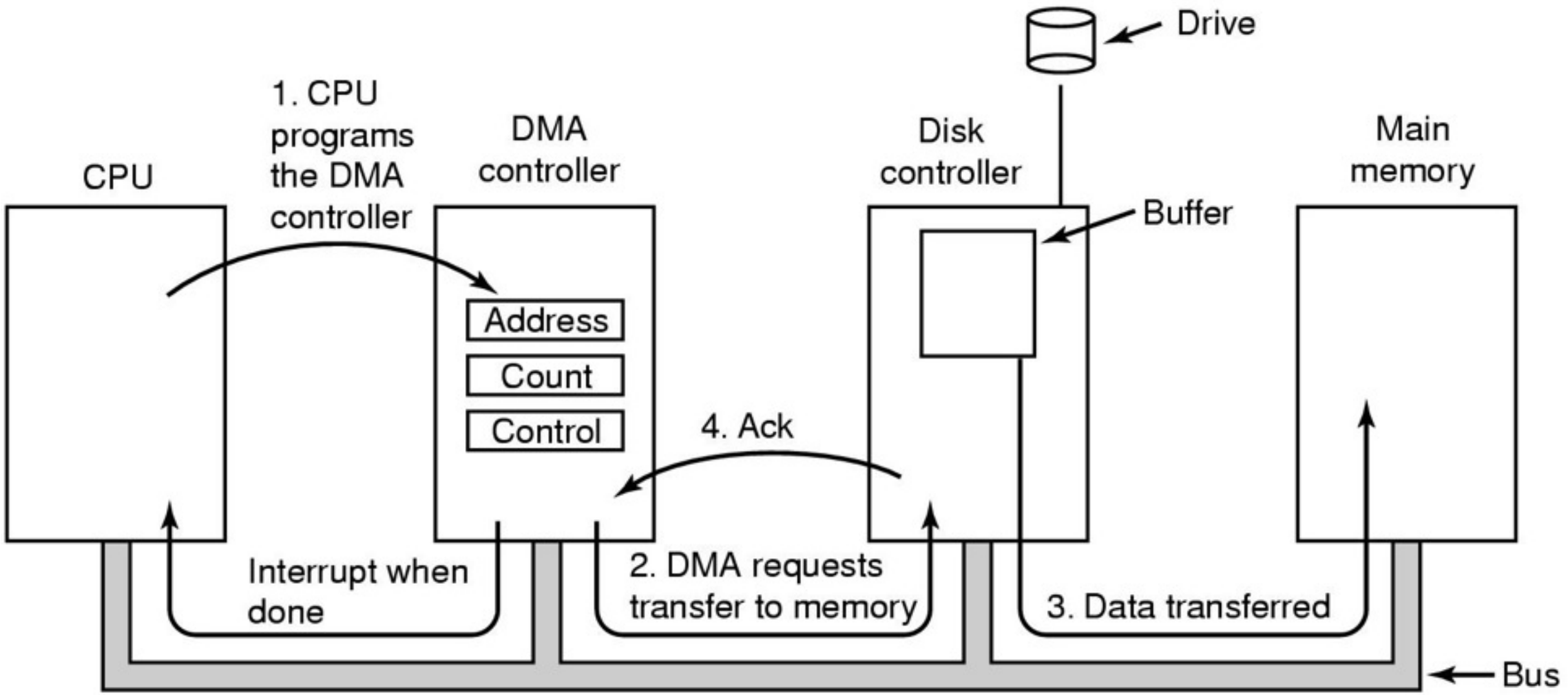


- Separate I/O and memory space
  - Special I/O commands (e.g., IN/OUT)
- Memory-mapped I/O

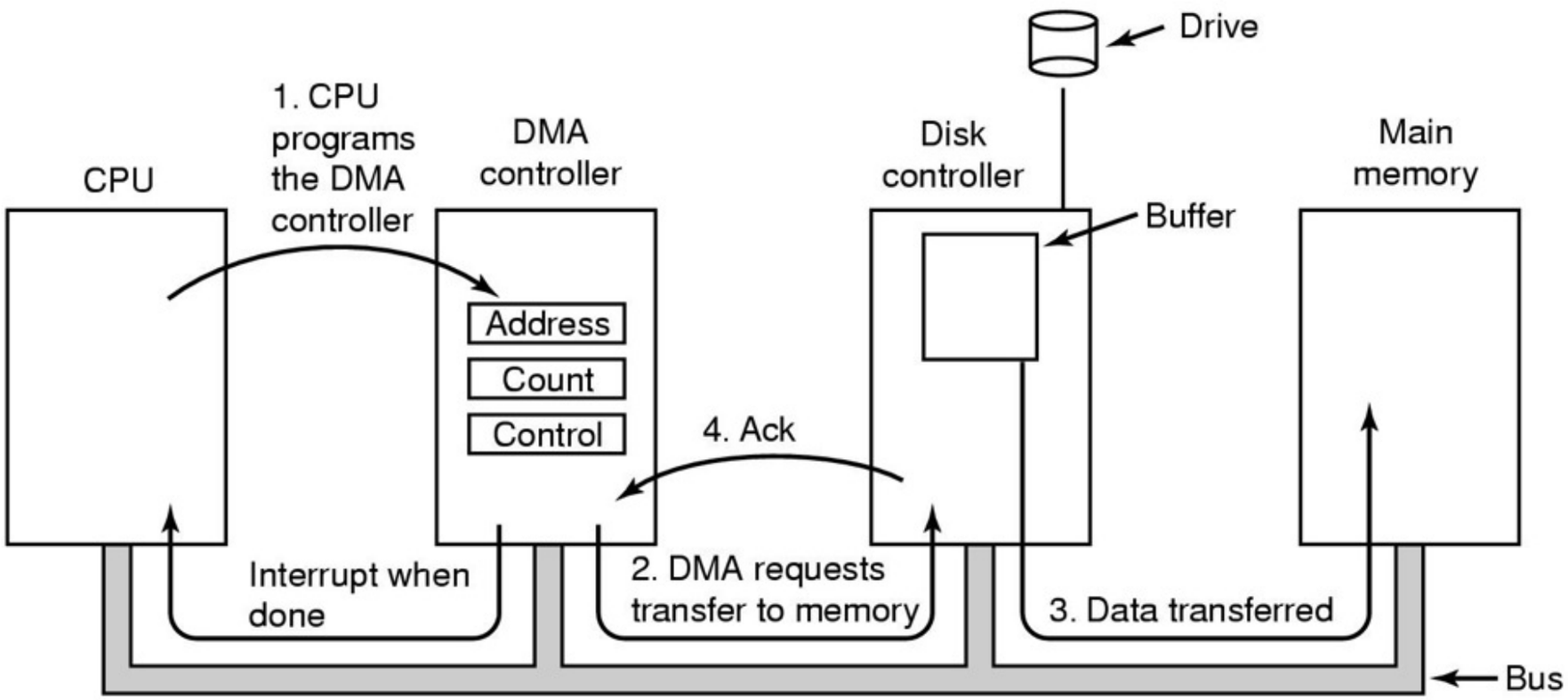
I/O Operations

- How is I/O done?
  - I/O devices are much slower than CPU
- Synchronous (polling)
  - Start I/O
  - In a loop, CPU checks the device status register until it shows the operation completed
- Asynchronous (interrupt-driven)
  - After I/O starts, control returns to the user program without waiting for I/O completion
  - Device controller later informs CPU that it has finished its operation by causing an interrupt
  - When an interrupt occurs, current execution is put on hold; the CPU jumps to a service routine called an “interrupt handler”

Direct Memory Access (DMA)



DMA Questions



- Does the CPU send virtual or physical addresses to the DMA controller?
- Can the disk controller directly read data into main memory, bypassing its controller buffer?

Protection of I/O Devices

- User programs are not allowed to directly access I/O devices
  - Special I/O instructions can only be used in kernel mode
    - Controller registers can only be accessed in kernel mode
- So device drivers, I/O interrupt handlers must run in kernel mode
- User programs perform I/O through requesting the OS (using system calls)

Device-Controller-Software Relationship

Application

User Mode Software

High-Level OS Kernel Component

Device Driver

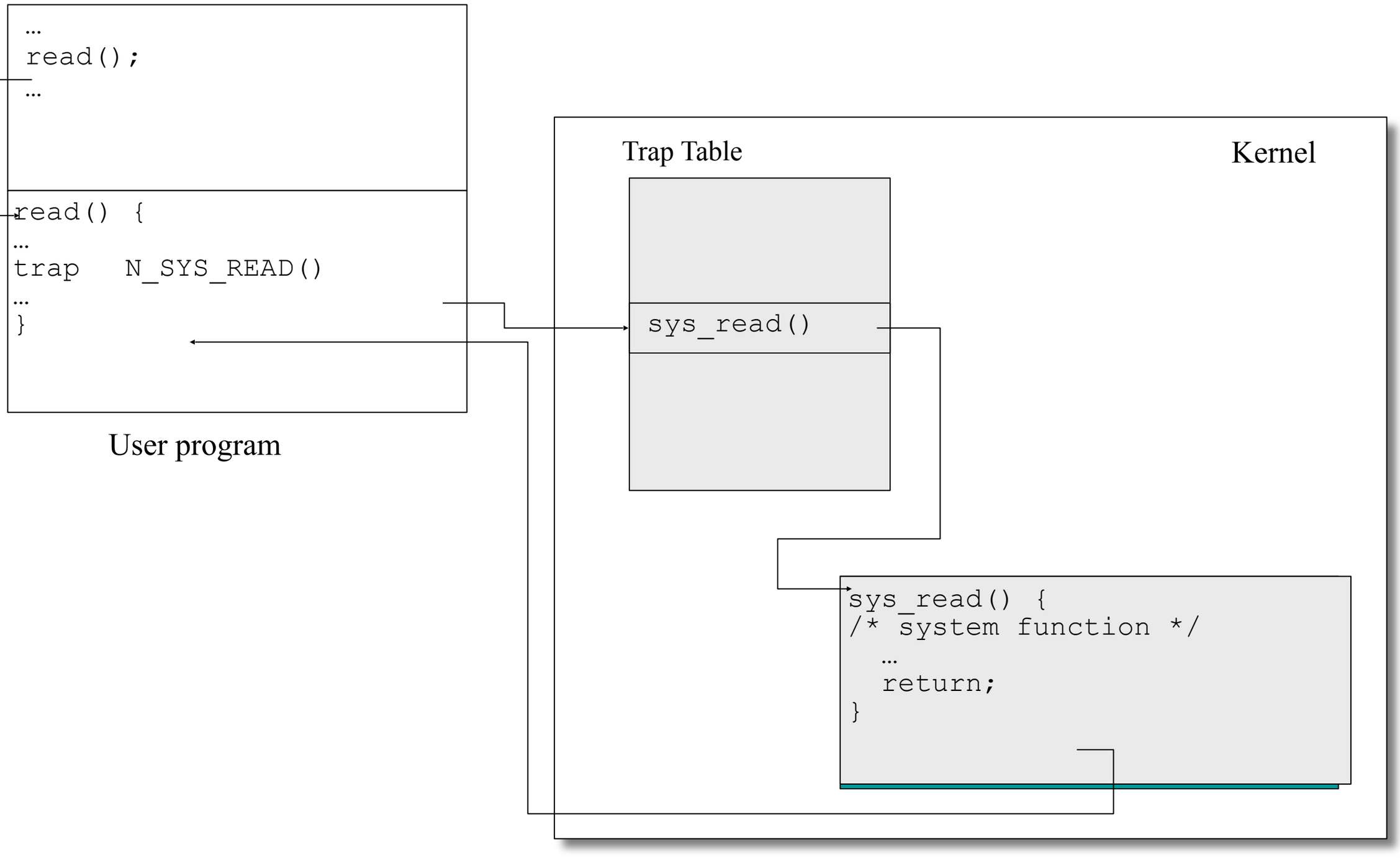
Kernel Mode Software

Device Controller

Device

Hardware

System Call Using the Trap Instruction



Processes

- Def: A process is an instance of a running program.
  - Not the same as "program" or "processor"
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - Private address space
    - Each program seems to have exclusive use of main memory.
- How are these Illusions maintained?
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system

Process Management

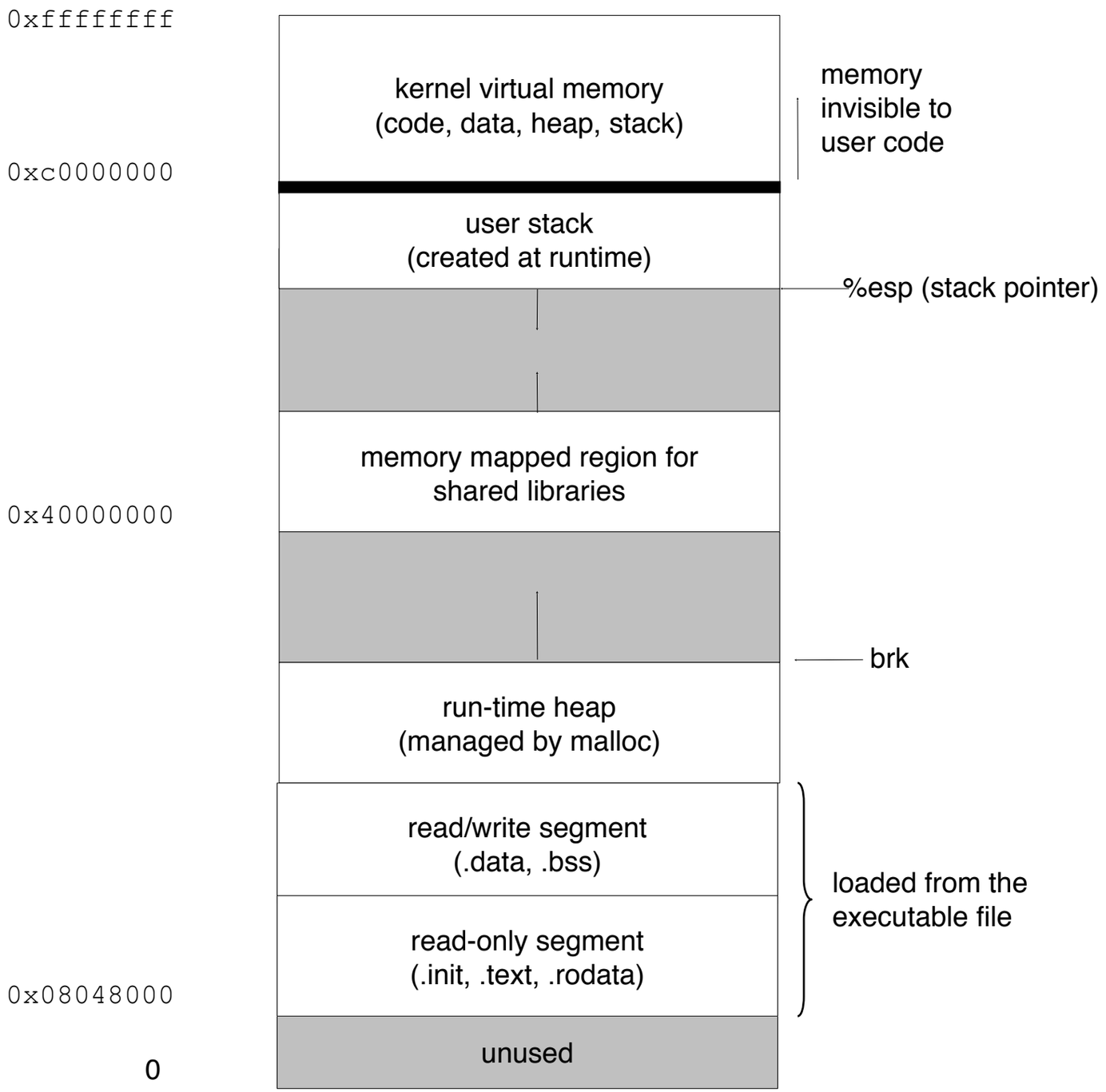
- A process is a program in execution
  - Unit of work – A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task
  - Protection domain

- OS responsibilities for process management:
  - Process creation and deletion
  - Process scheduling, suspension, and resumption
  - Process synchronization, inter-process communication

Process and Its Image

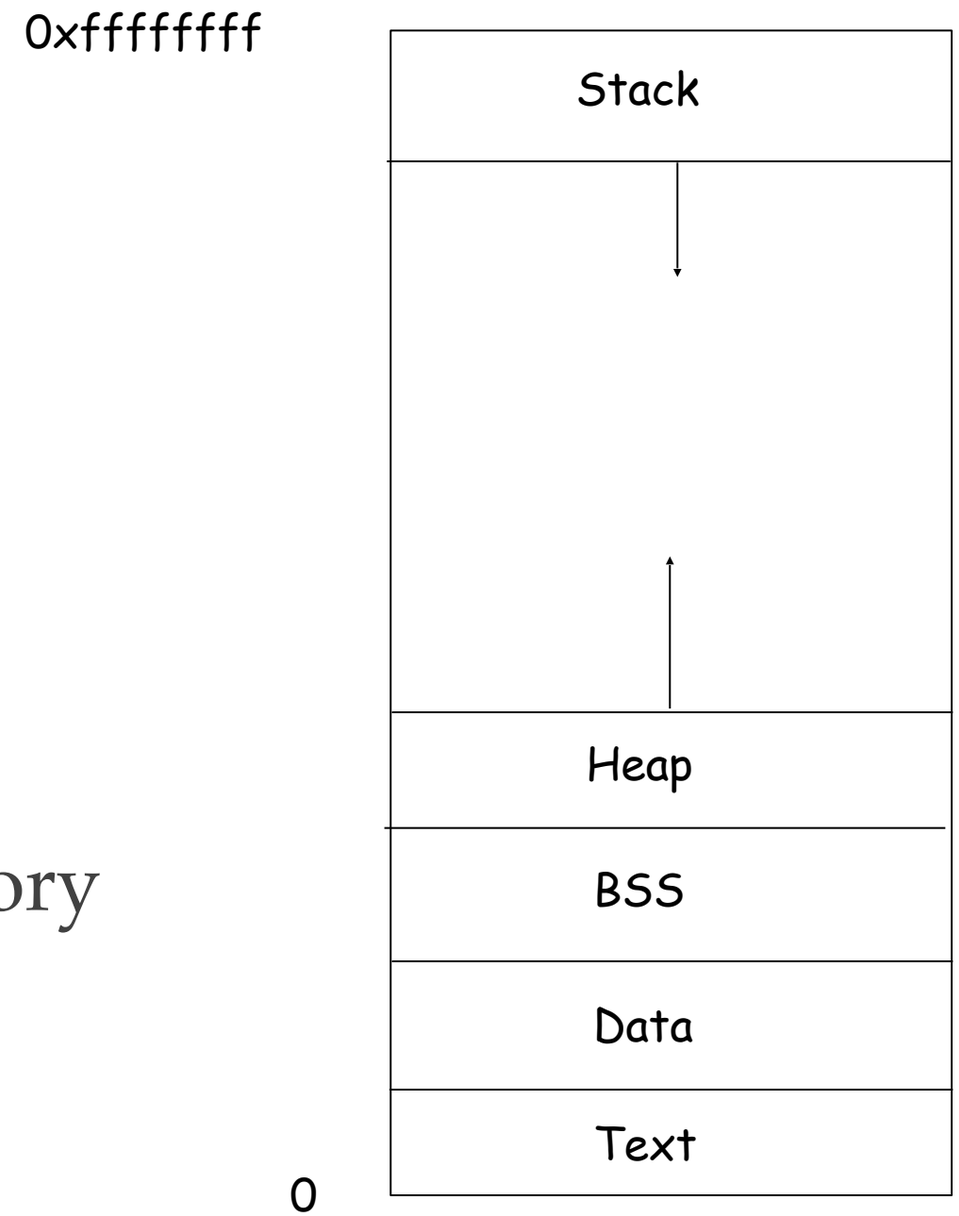
- A process's state/image in a computer includes:
  - User-mode address space
  - Kernel data structures maintained on process's behalf
  - Registers (including program counter and stack pointer)
- Address space and memory protection
  - Physical memory is divided into user memory and kernel memory
  - Kernel memory can only be accessed when in the kernel mode
  - Each process has its own exclusive address space in the user-mode memory space (sort-of)

Process Private Address Space



User-Mode Address Space

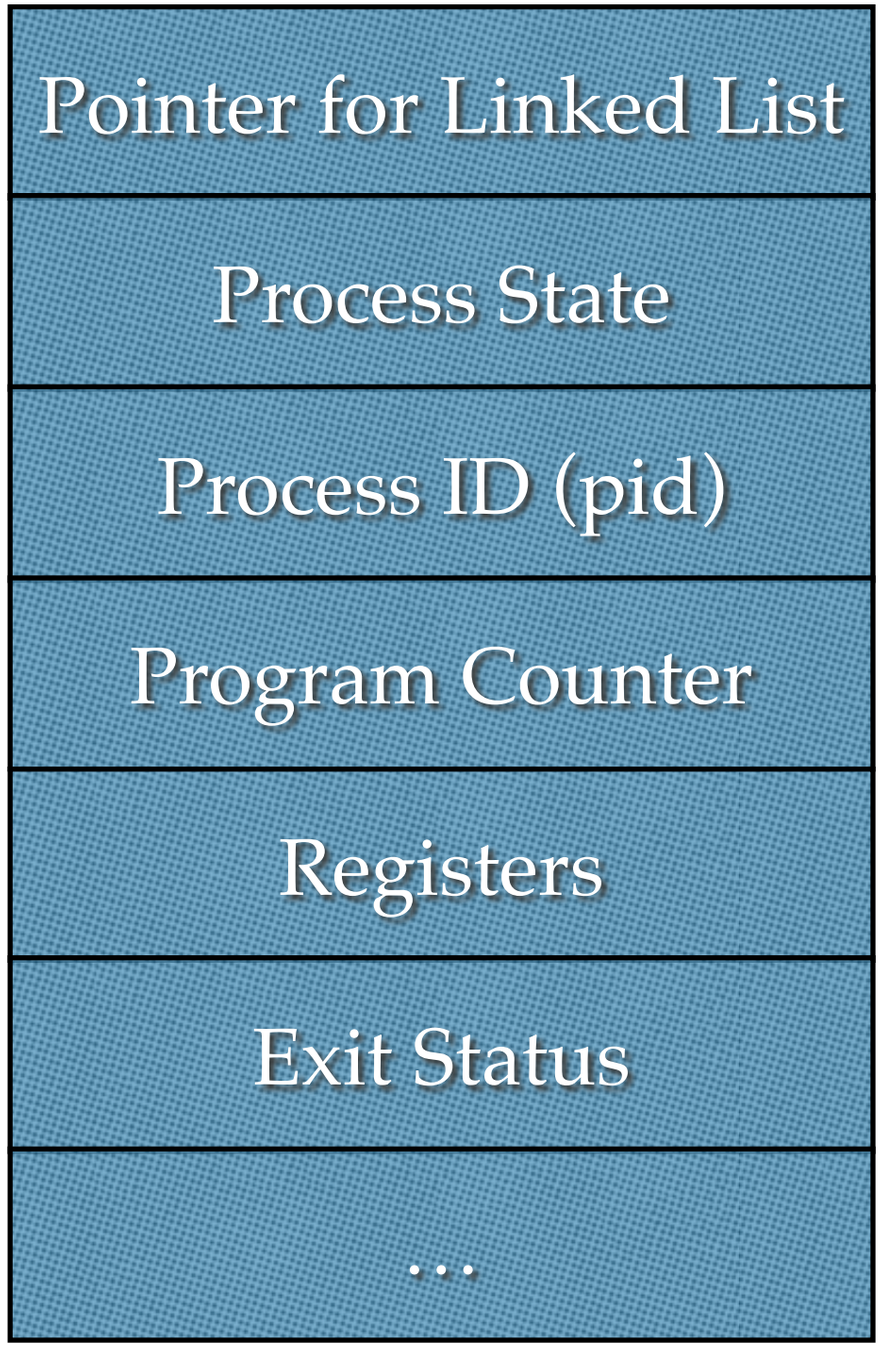
- Text: Program code
- Data: Initialized global and static variables
- BSS (Block Stated by Symbol): Uninitialized global and static variables
- Heap: Dynamically allocated memory
- Stack: Local variables and function activation records



Process Control Block (PCB)

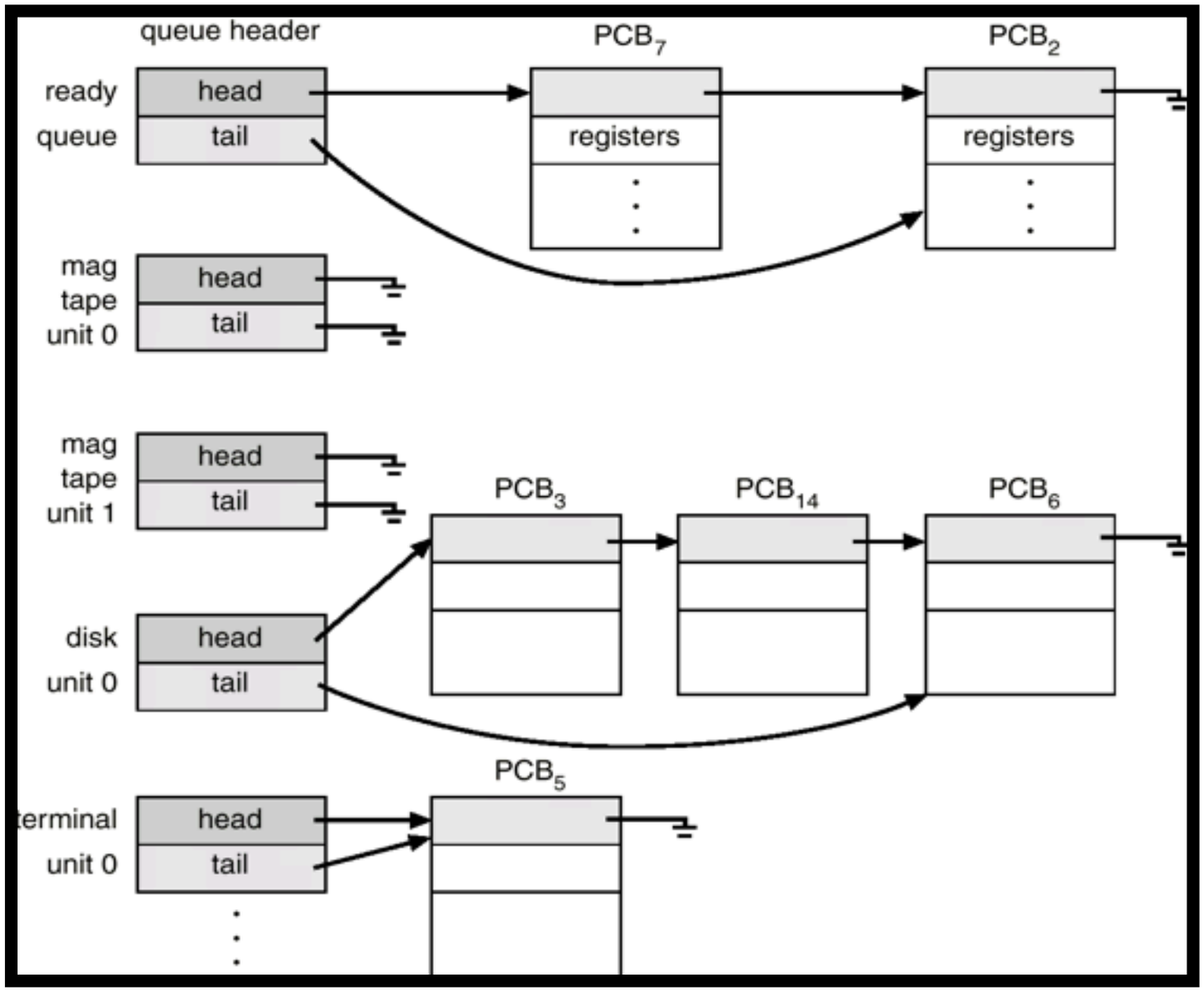
OS data structure (in kernel memory) for maintaining information associated with each process.

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- Information about open files



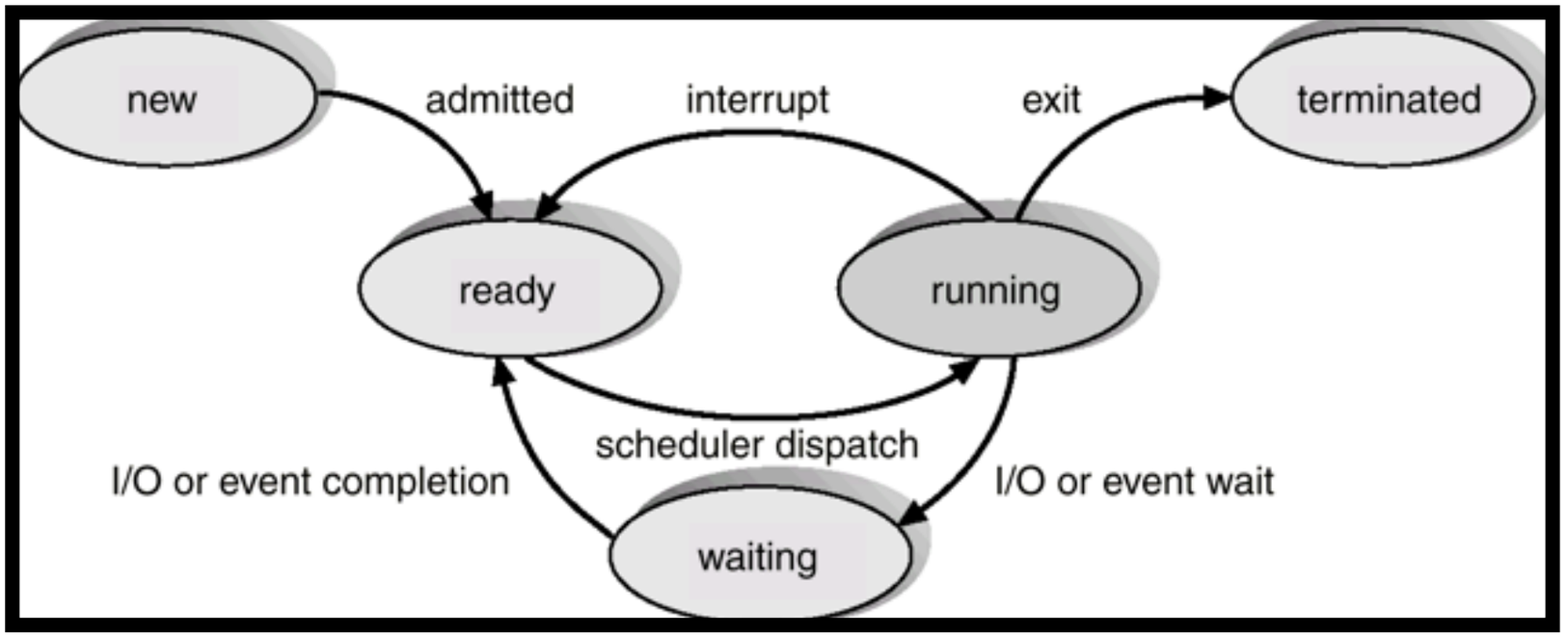
Queues for PCBs

- Ready queue – set of all processes ready for execution.
- Device queues – set of processes waiting for an I/O device.
- Process migration between the various queues.



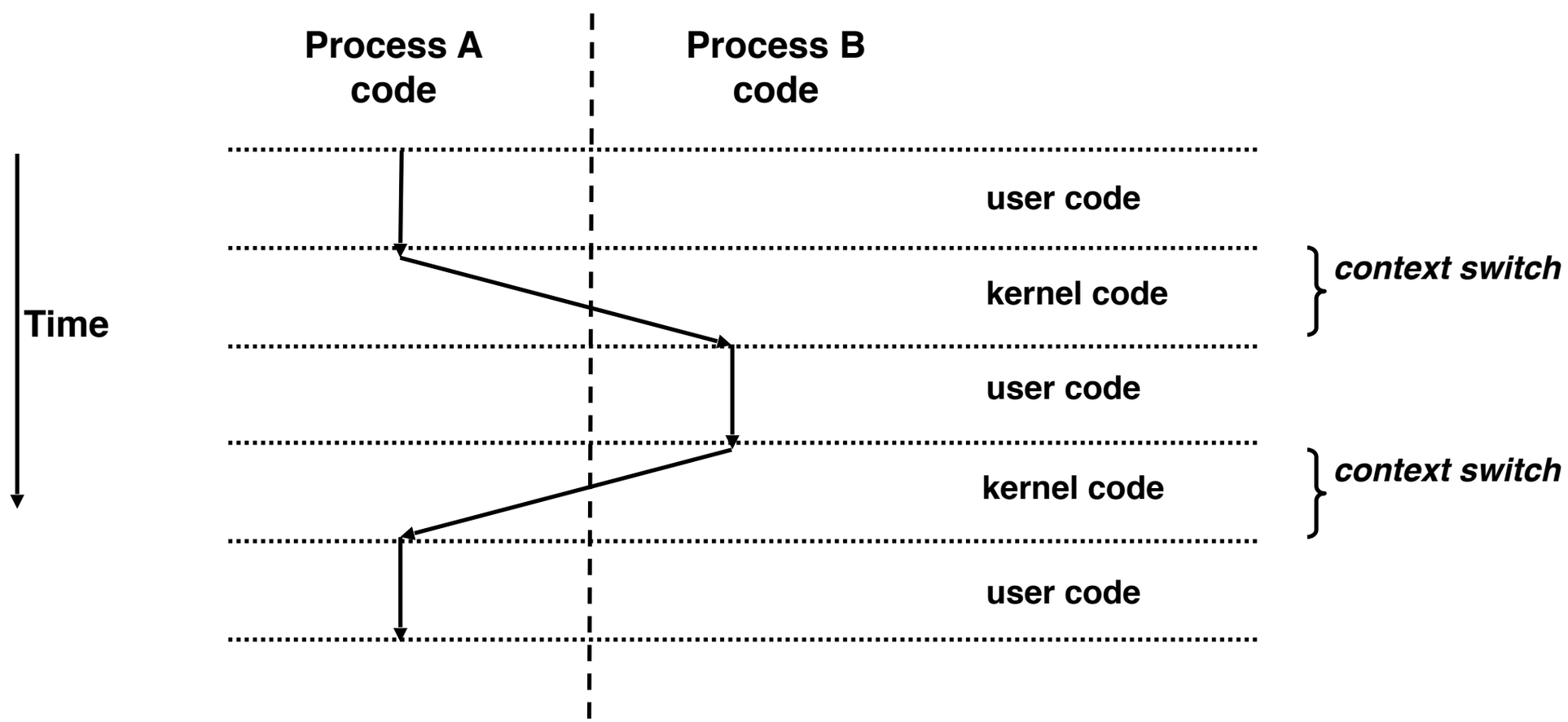
Process State

- As a process executes, it changes state
  - new: The process is being created
  - ready: The process is waiting to be assigned to a processor
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - terminated: The process has finished execution



Context Switching

- Processes are managed by the OS kernel
  - Kernel is "part" of user process; is not its own process
- Control flow passes from one process to another via context switch



Scheduling: Transferring Context Blocks

Coroutines

transfer(other)

save callee-saved registers on stack

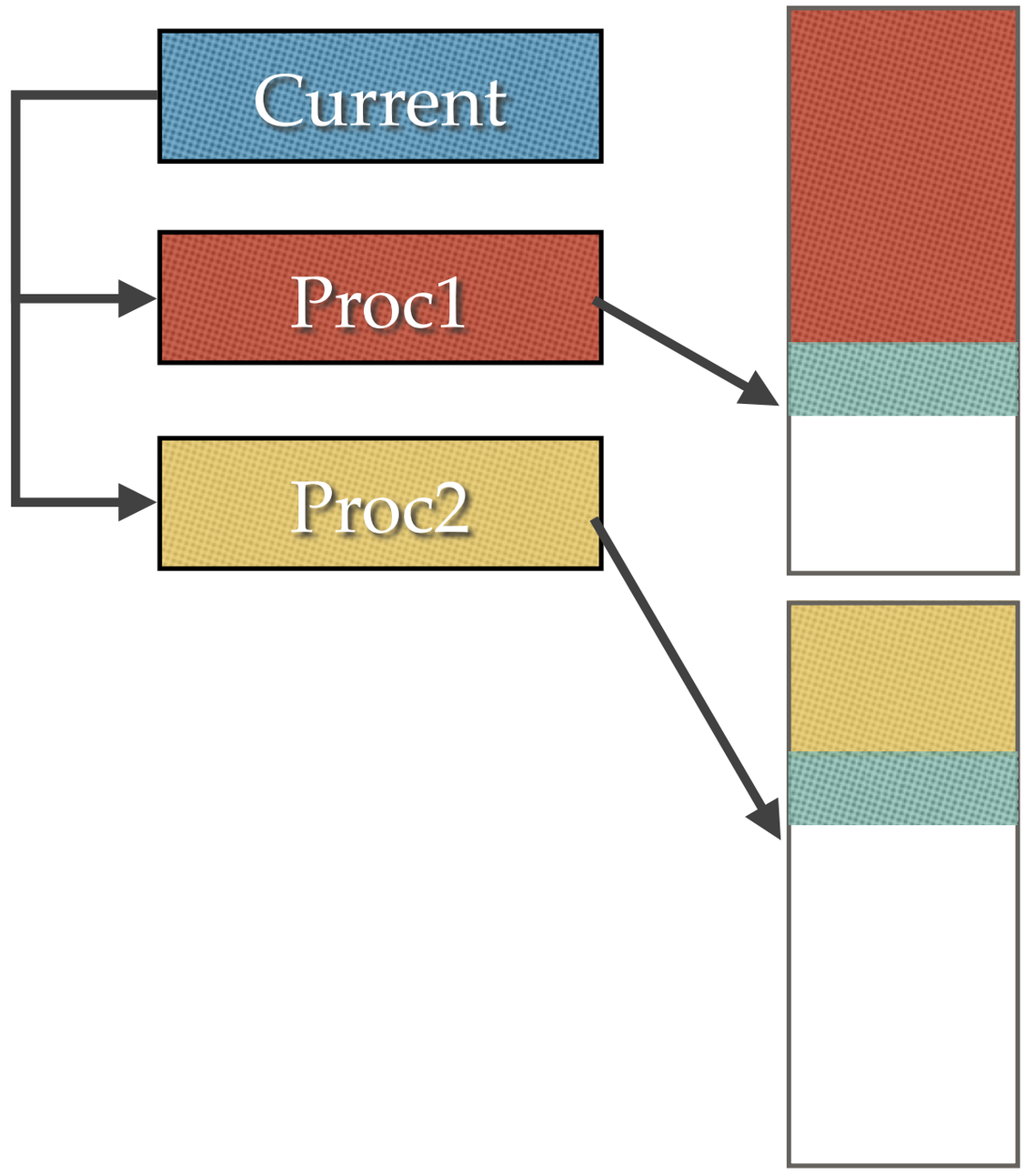
*current := sp

current := other

sp := *current

pop callee-saved registers (except sp!)

return (into different coroutine!)



Cooperative Multitasking

- Use Ready List to reschedule voluntarily
  - AKA cooperative threading

reschedule:

- t := dequeue(ready_list)
- transfer(t)

yield:

- enqueue(ready_list, current)
- reschedule

sleep_on(q):

enqueue(q, current)
reschedule

Preemptive Multitasking

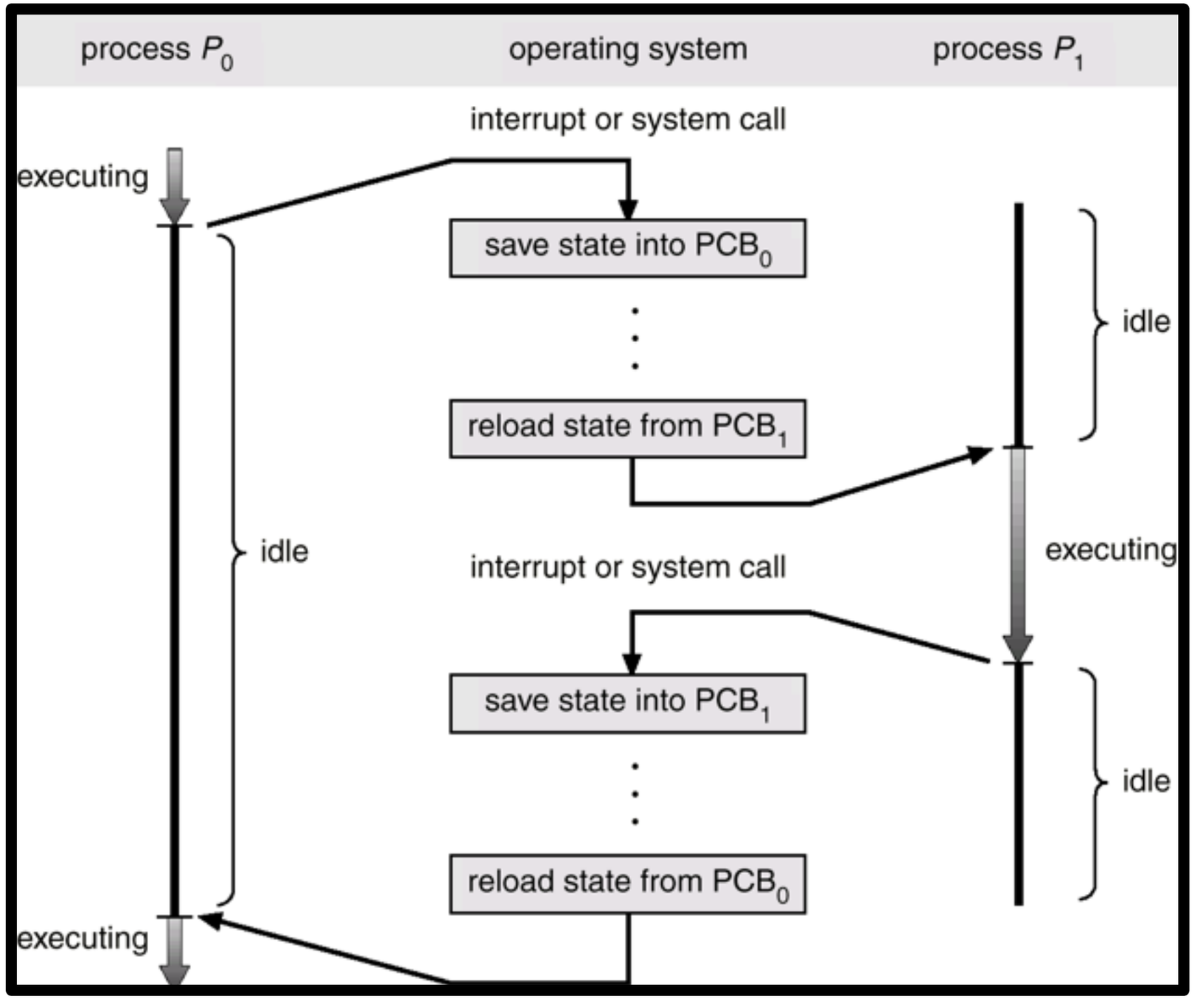
- Use timer interrupts or signals to trigger involuntary yields
- Protect scheduler data structures by disabling/reenabling interrupts prior to/after rescheduling

```
yield:
  disable_signals
  enqueue(ready_list,
  current)
  reschedule
  re-enable_signals
```

CPU Switch From Process to Process

When can the OS switch the CPU from one process to another?

Which one to switch to?
Scheduling!



Process Termination

- Process executes last statement and gives the control to the OS (exit)
  - Notify parent if it is wait-ing
  - Deallocate process's resources
- The OS may forcefully terminate a process.
  - Software exceptions
  - Receiving certain signals

Credits

- Parts of the lecture slides contain original work from Gary Nutt, Andrew S. Tanenbaum, Dave O'Hallaron, Randal Bryant, Kai Shen, and Sandhya Dwarkadas. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).