Chapter 1
Introduction

CS 3423 Operating Systems
National Tsing Hua University

Chapter 1: Introduction

- What OS's Do
- System Organization
- System Architecture
- OS Structure
- OS Operations
- Process Management
- Memory Management
- Storage Management
- Protection and Security
- Kernel Data Structures
- Computing Environments
- Open-Source OS's

A Computer System consists of...

- Users
  - people, other computers
- Applications
  - code that solves computing problems by using system resources
    - e.g, word processors, compilers, browser, databases, games, ..
- OS
  - controls and coordinates use of resources (both hardware and software)
- Hardware
  - tangible mechanisms for computing (CPU), storage (memory, disk), communication, and other I/O (sensors, actuators)

Four Components of a Computer System

users

application programs (editor, shell, compiler)

browser?
database?
"middleware"

system programs (shell, loader, backup...)

OS

hardware

What an OS does - depends on point of view

- General Users
  - convenience, ease of use and good performance
- Shared computer (mainframe) users
  - want responsiveness and throughput
- Mobile users
  - want mobile-friendly UI, battery
- Embedded computers
  - possibly timing guarantee

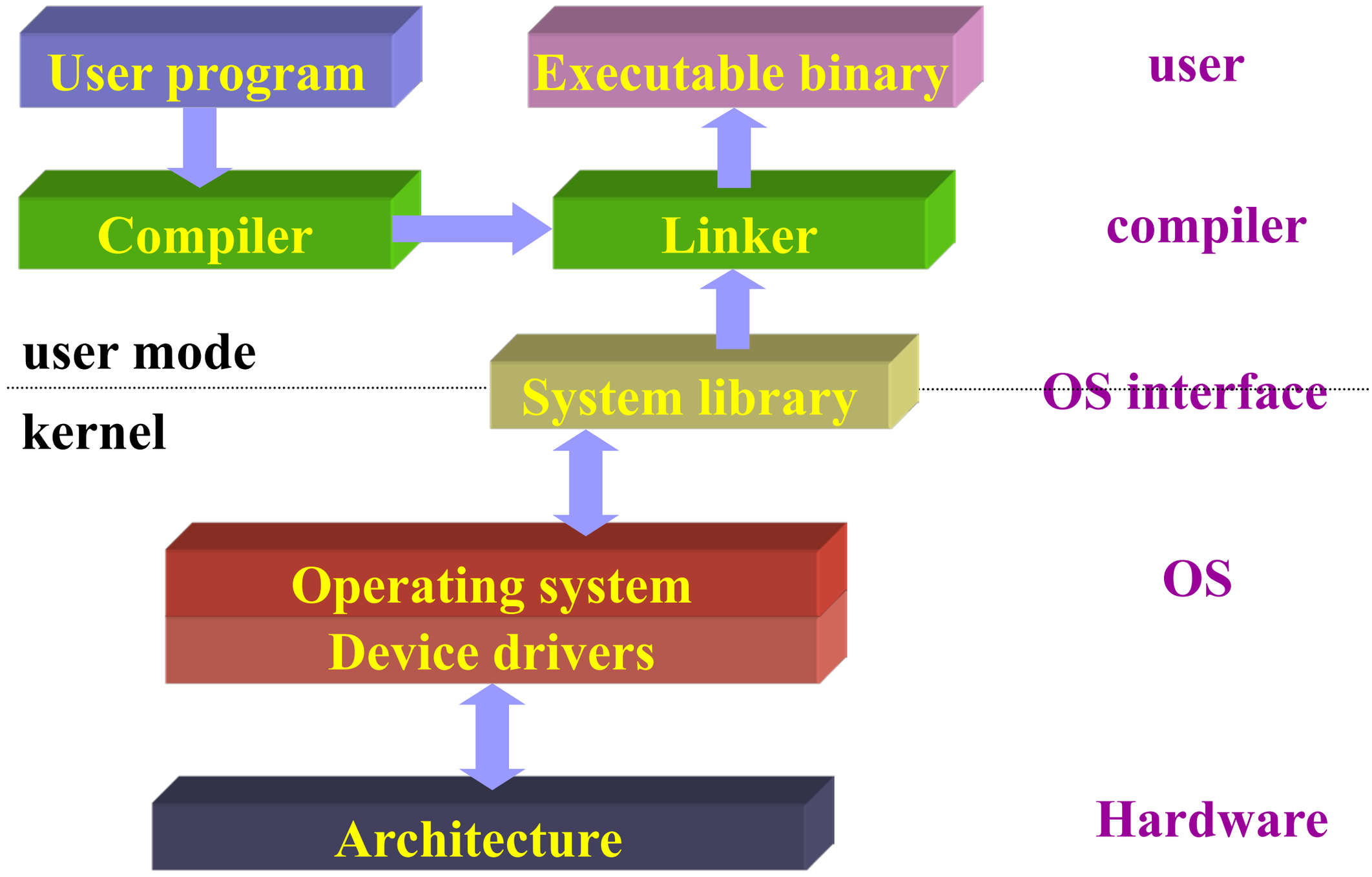
Definition of an OS

- Resource Allocator
  - manages and allocates resources: CPU time, storage space, use of I/O devices,
  - goal: ensures efficiency, fairness, and security
- Control program
  - controls execution of user programs
  - controls operation of I/O devices
  - goals: prevent errors and improper use

Multitasking OS

- Supports multiple user applications by managing resources and processes
  - Loading/unloading code, scheduling program execution
- Provides API for user applications
  - API = Application programming interfaces
  - Services: use of memory, I/O devices, storage, communication

General-Purpose OS



Boundary of OS?

- Kernel => definitely part of OS
  - the "core" part that is resident and governs OS functionality
- Boot loader? (bundled)
  - ROM code executed on power up, loads the OS into memory
- System programs (bundled)
  - Program loaders — for OS to start running a program
  - Interpreters: CLI shell (DOS? BASIC?) GUI (X11? iOS)? Java?
  - Compiler and linker? Device driver? library?
- Middleware — layer crossing the network (bundled)
  - Web browser? is IE an inseparable part of Windows? ("no" -- ruled by DOJ)
  - Database, multimedia, cloud drive, location service...

Goals of an OS

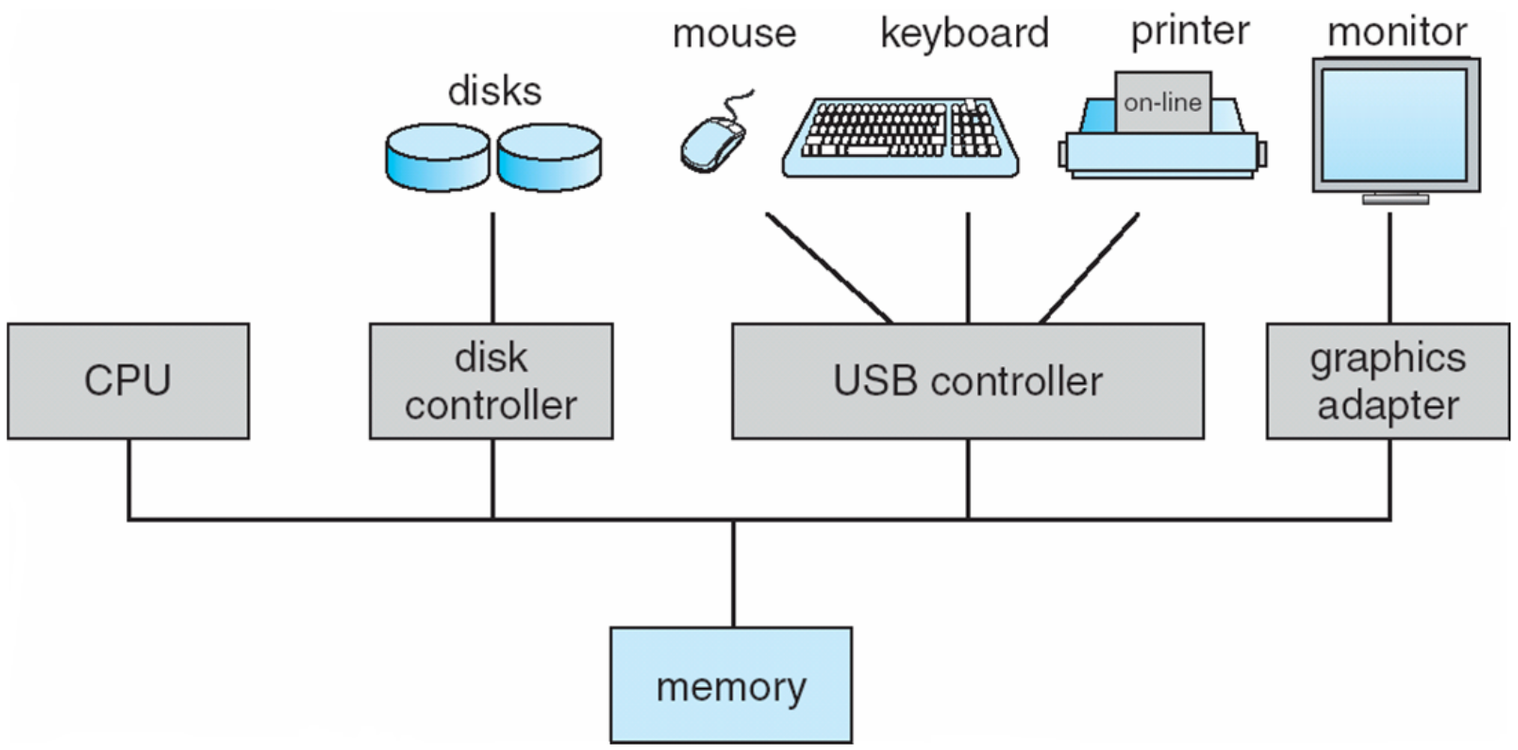
- Convenience
  - make computer system easier to use and to program
  - esp. true for smaller computer systems
  - desirable; technically not essential
- Efficiency
  - do same work in shorter time or consuming less power
  - esp. true for large, shared, multiuser systems and mobile
- The two goals are sometimes contradictory

Importance of an OS

- System API (application programming interface)
  - are the only interface betw. user applications and hardware
  - APIs are usually designed to be general purpose, but not performance driven
    - example: standard I/O, file I/O, TCP/IP connection, ...
- OS code can't afford to contain bugs
  - any crash or bug could cause downtime, data corruption, financial loss, or loss of life
- OS and Architecture influence each other
  - and programming language to some extent

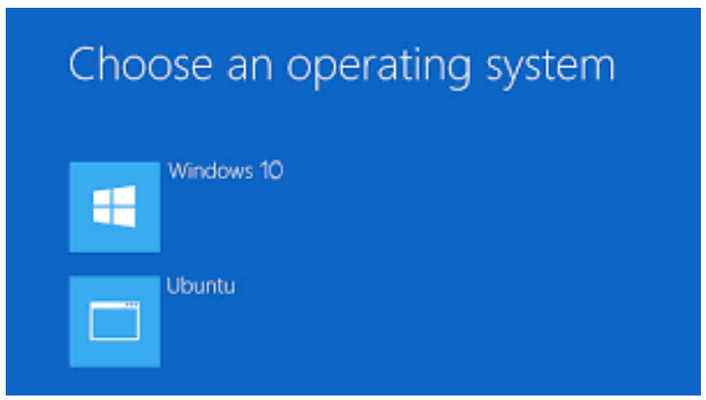
Computer System Organization

- One or more CPUs
- Device controllers
- Interconnect of CPUs, devices, and memory
- Goal: concurrent execution of CPU & devices competing for memory cycles



Powering up: Bootstrapping

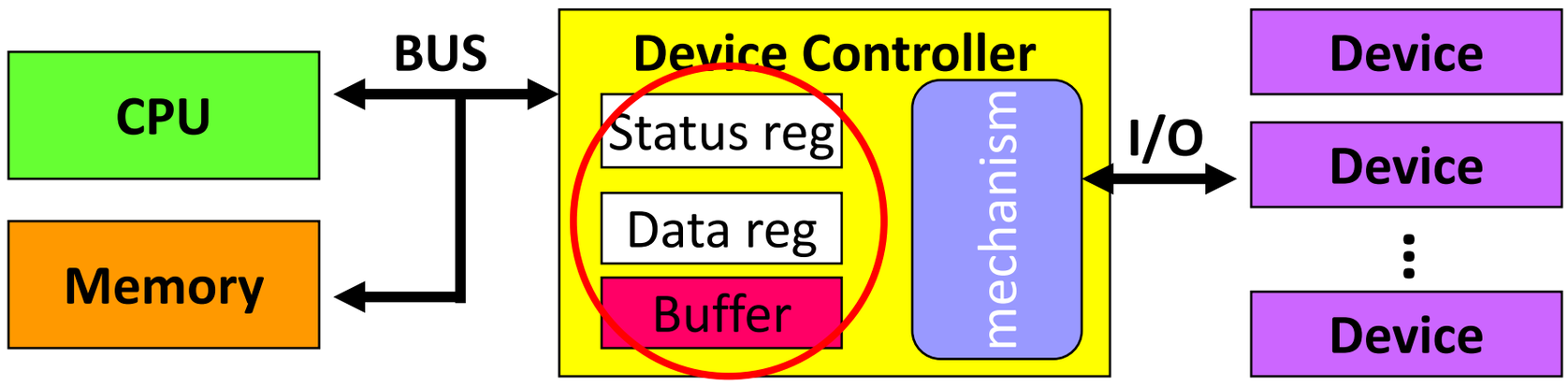
- CPU's initial program in firmware
  - ROM, EEPROM, flash, etc => nonvolatile memory
  - CPU starts at a given location (hardware defined)
  - Goal of bootloader: do enough to load in the OS kernel!



- Bootloader may need
  - device driver for storage of OS (disk, flash, network)
  - copying OS image from storage into memory (RAM)
    - some embedded systems may run directly from flash memory, without copying OS image to RAM
  - jump to starting point of OS

Computer System Operation

- Device controller for a particular device type
- Status register, data register, buffer on device controller
  - read/written by both CPU and device controller
- Notification by device controller to CPU
  - "data ready" signal, aka "interrupt request" (IRQ)



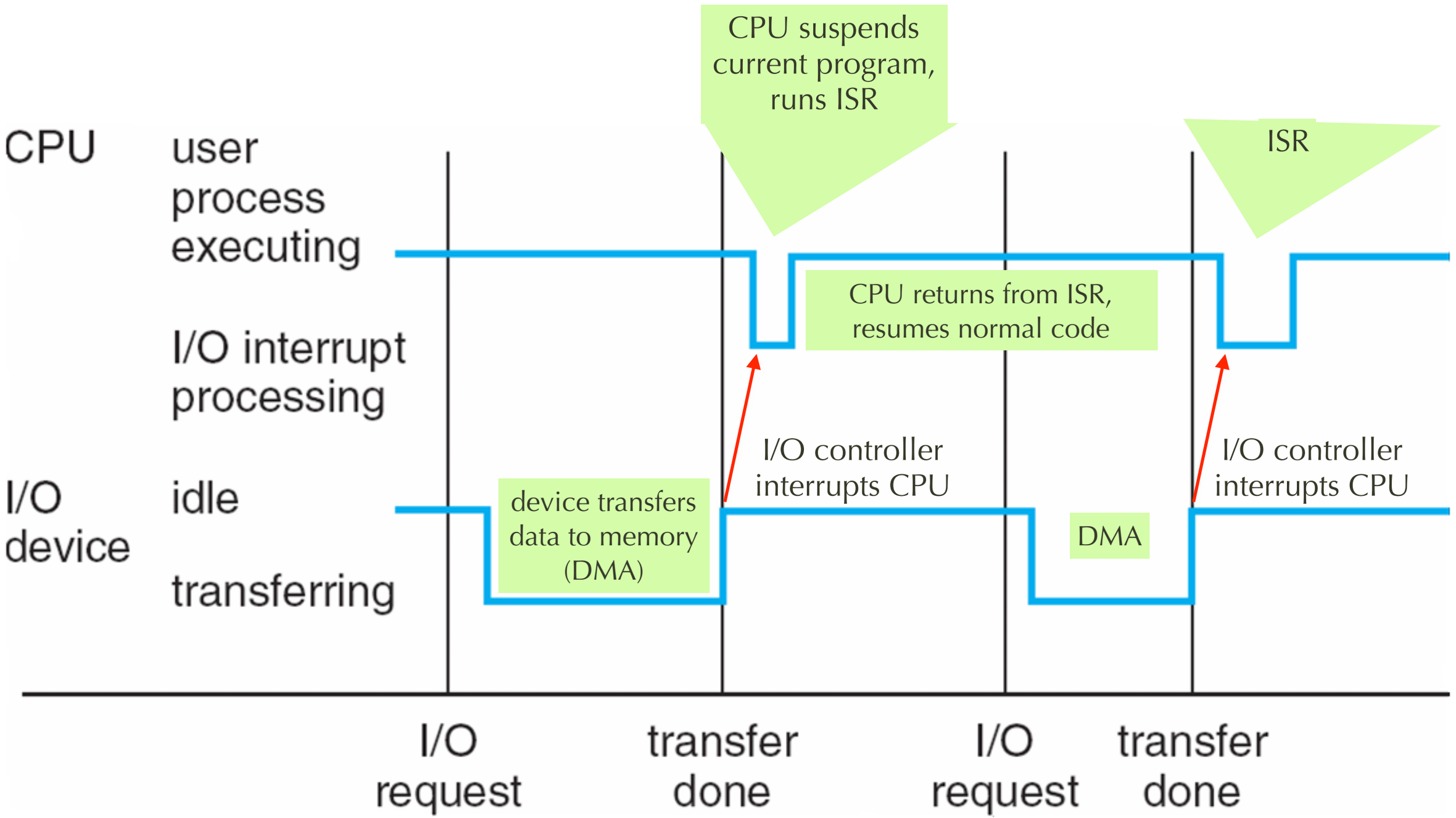
Polling: Busy / Wait Output

- Simplest (but inefficient) way to program device
  - machine instruction to test when device is ready
- Example: 8051 serial port
  - special function registers (SFR)
    - RI is the receive data-ready flag bit, ==1 if a char received
    - SBUF contains the character buffer
- **while** (!RI) { ; } *// busy wait until RI flag is set to 1*
  myChar = SBUF; *// read from the buffer*

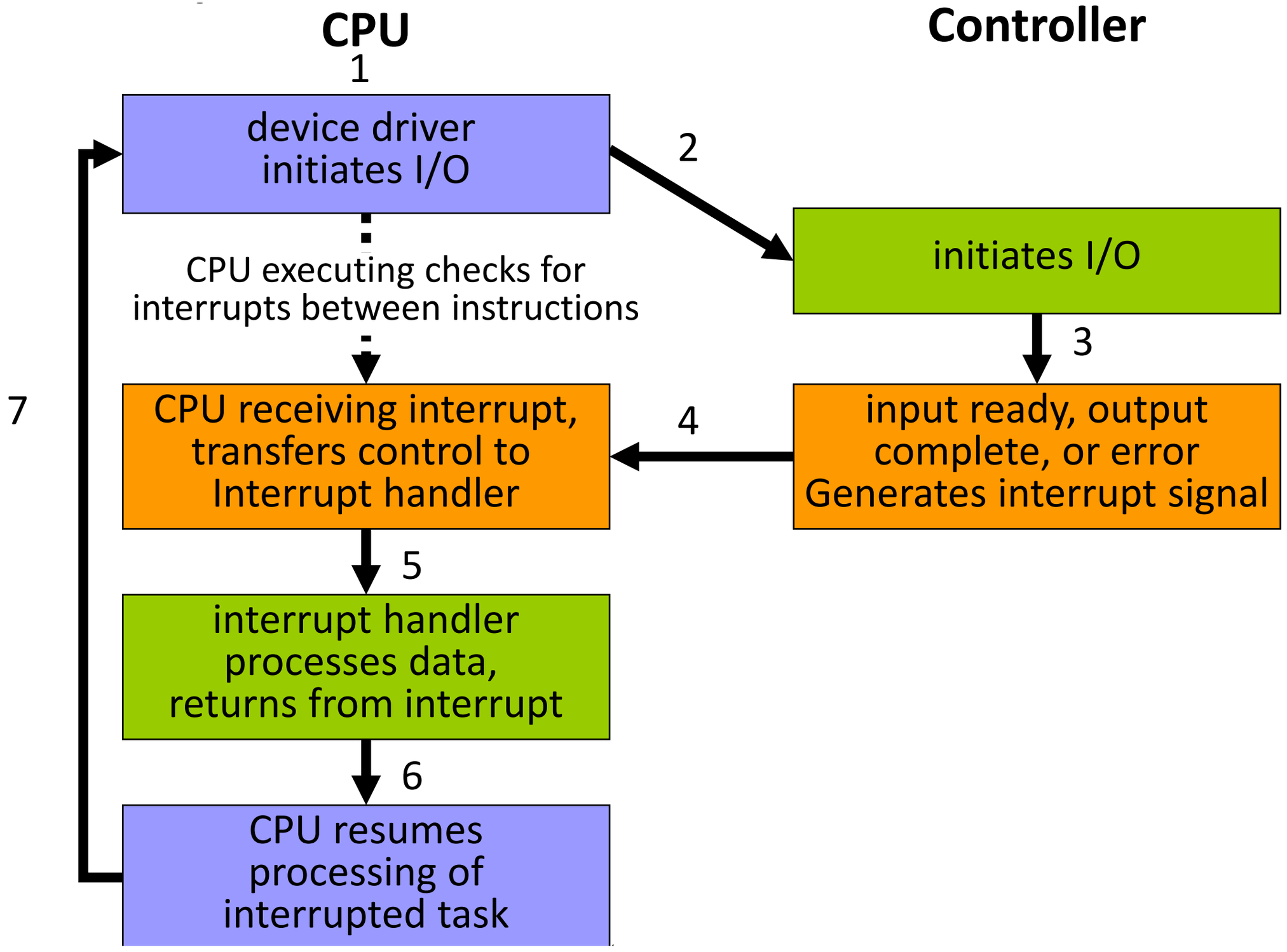
Interrupt I/O

- Busy waiting is inefficient
  - Poor CPU utilization: can't do anything else!
  - Could try to interleave several I/Os, but code becomes very messy
- Solution: interrupt I/O
  - CPU can execute regular code
  - upon data ready, CPU jumps to subroutine (interrupt service routine = ISR) to handle I/O

Interrupt I/O Timeline (data from device to CPU)



Interrupt-Driven I/O

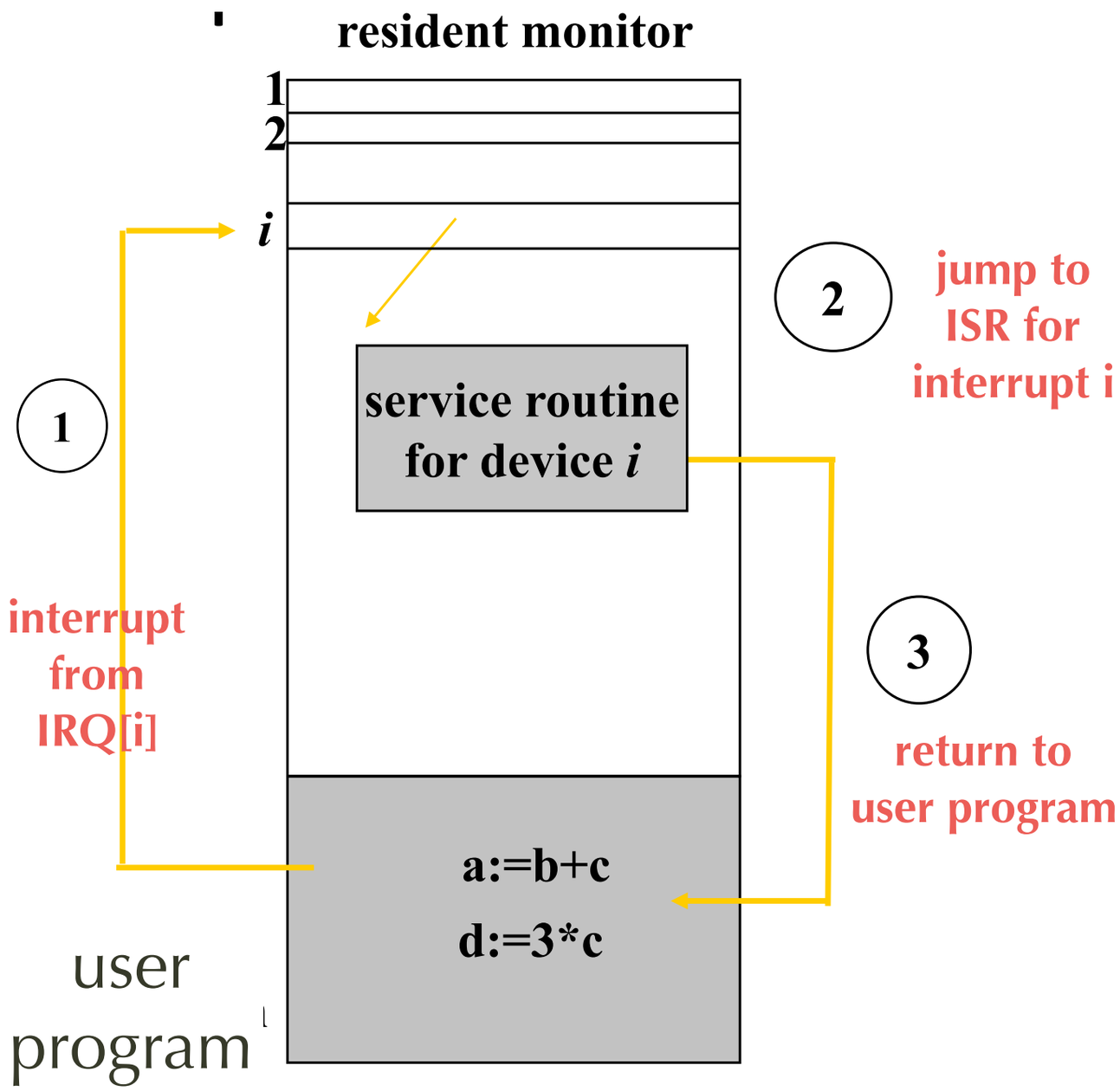


Interrupt: hardware & software

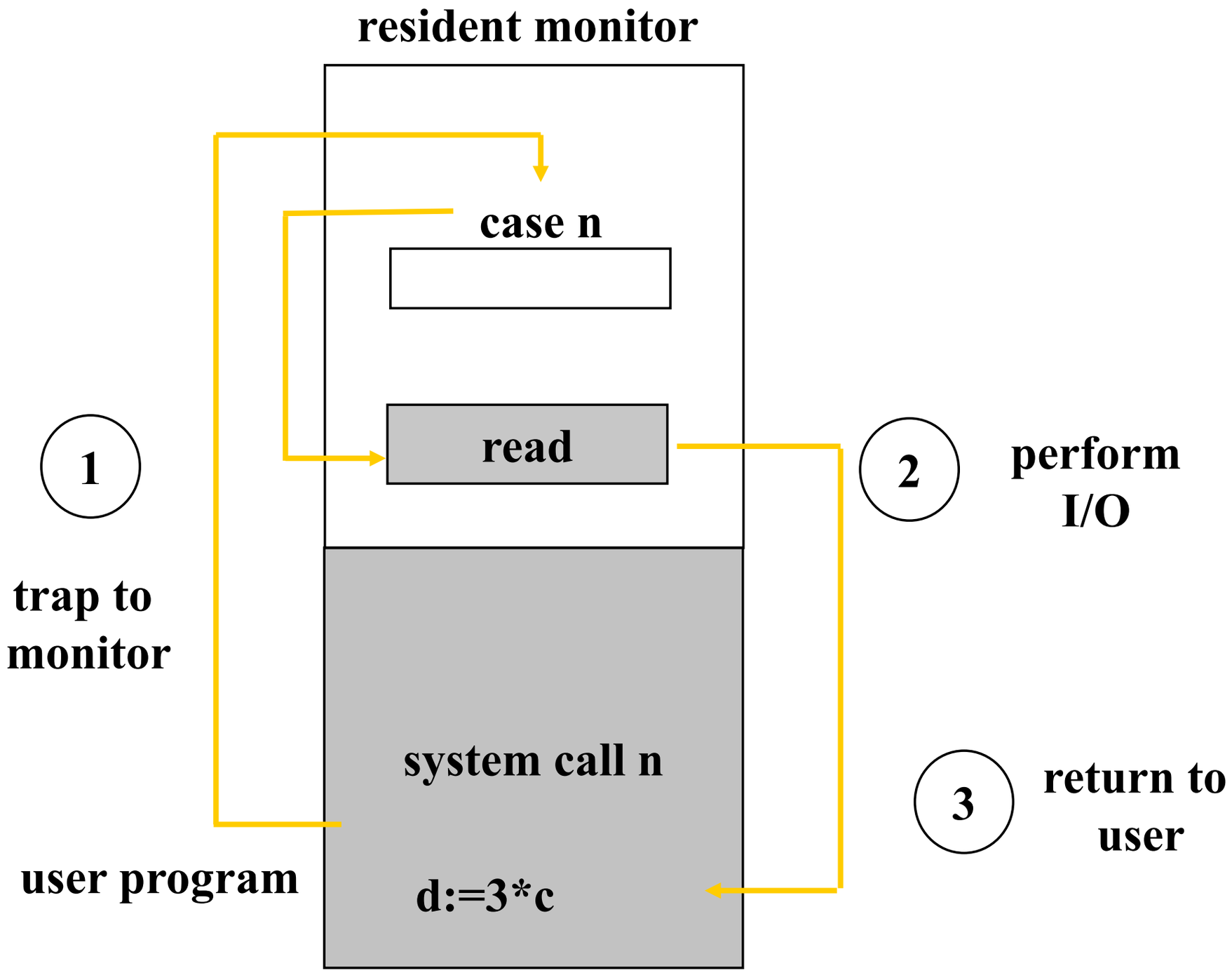
- Modern OSs are interrupt-driven
- Interrupt may be trigged from either hardware or software
- Hardware interrupt
  - someone asserting in IRQ line (setting to TRUE)
- Software interrupt, also called **trap**
  - caused by error (e.g., divide by 0, invalid memory access)
  - by user request for an OS service (system call) — pass the trap# as a parameter

Hardware Interrupt

- Interrupt vector
  - array of addresses of ISRs, indexed by the interrupt number
  - each interrupt number is associated with a hardware source of IRQ



Software Interrupt



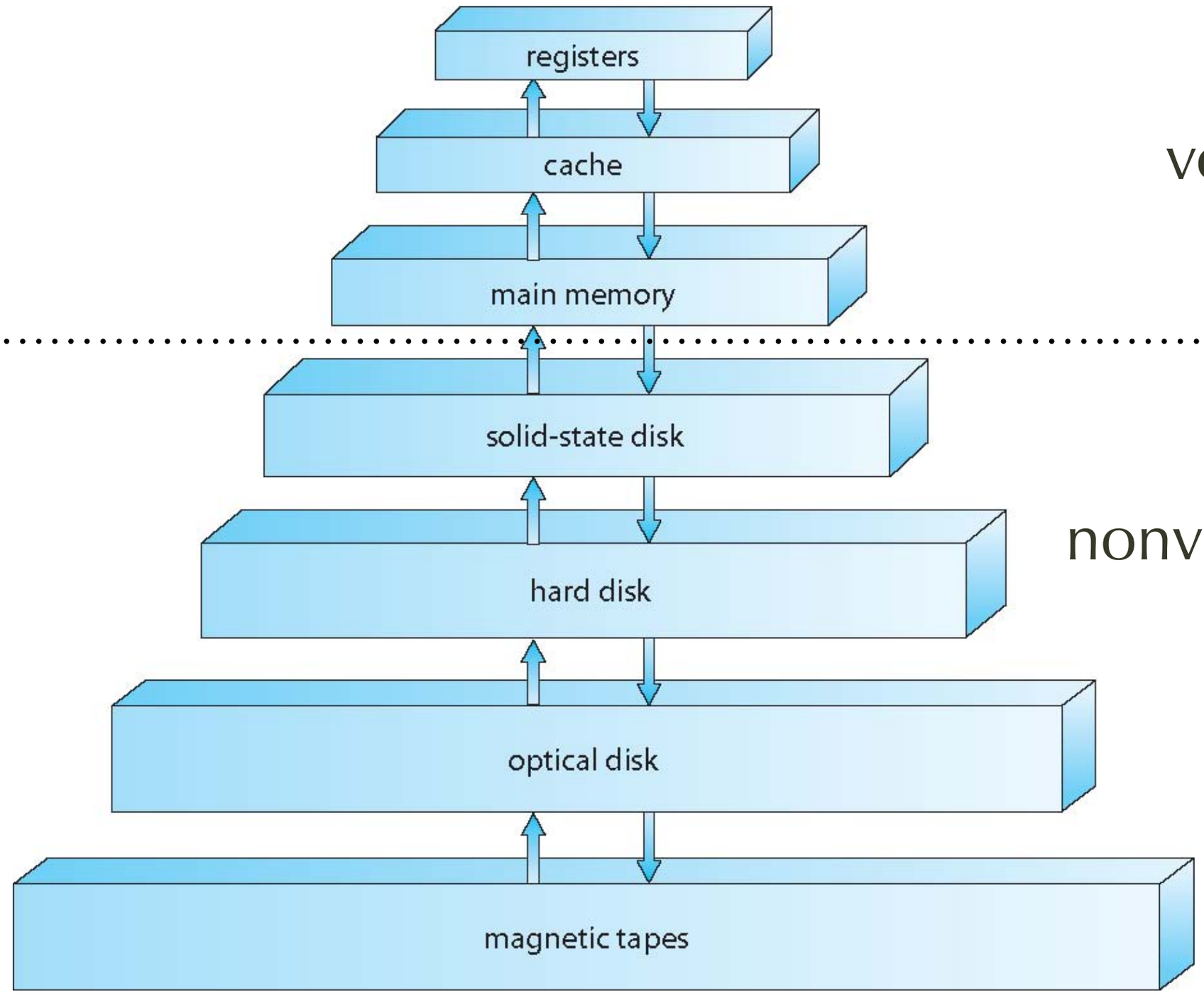
Common Functions of Interrupts

- CPU suspends execution of current program, transfer control to ISR through interrupt vector
  - interrupt # indexed into the table
- CPU saves address of the interrupted routine before jumping to ISR
  - so that the ISR can return to interrupted code
- Interrupts may be nested
  - i.e., while executing one ISR, some architectures allow another higher-priority interrupt to come in!
  - lower-priority ones would not get processed and may get lost

Review of Topics

- System organization
- Bootstrapping
- Interrupt vs. Polling
- Steps in handling an interrupt
- Trap vs. interrupt

Storage Device Hierarchy



volatile memory

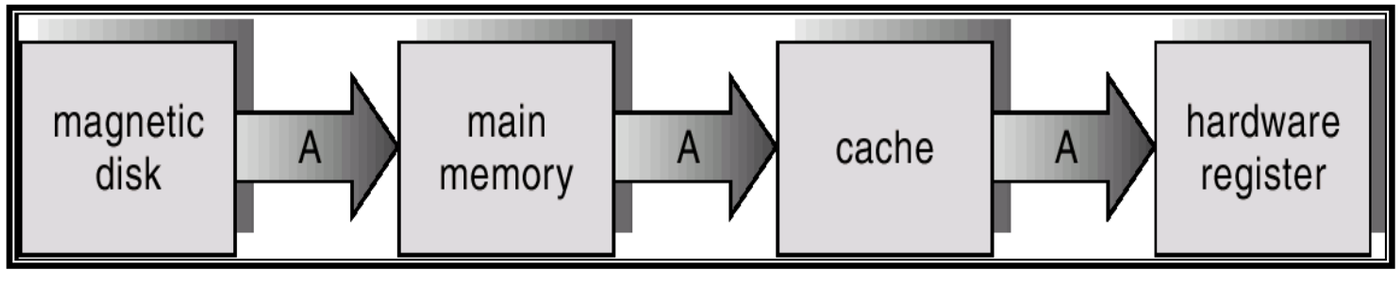
nonvolatile memory

Storage-Device Hierarchy

- Storage systems are organized hierarchically
  - by speed, cost, and volatility
- Main memory
  - the only “large” temporary storage medium that the CPU can access directly
  - RAM: random access memory; SRAM, DRAM etc
- Secondary storage
  - provides nonvolatile storage
  - Magnetic disk, solid state disks (SSD), etc

Caching

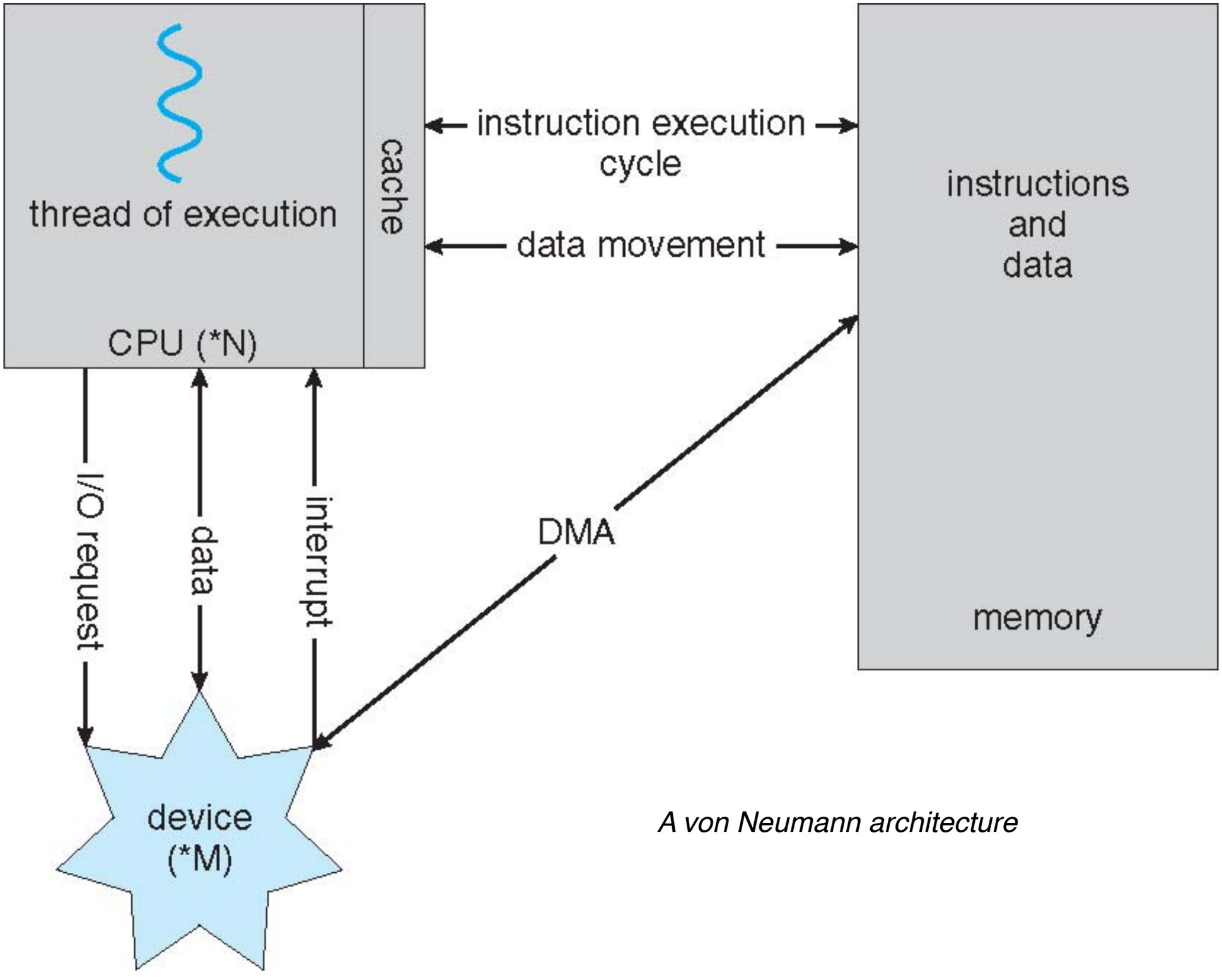
- Small but fast copy of subset of bigger (slower) memory
  - Important principle, performed in hardware, OS, software
  - Data copied to faster storage temporarily
- (Read) Cache is accessed first
  - If so (“cache hit”), use the copy from cache first (fast)
  - if not (“cache miss”), copy data from memory to cache (slow)
- Principle of locality
  - temporal locality and spatial locality
  - Cache size and replacement policy



Direct Memory Access (DMA)

- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
  - frees up CPU from copying data between device & memory
- Completion notification
  - One interrupt is generated per block (e.g., 512 bytes)
  - Rather than the one interrupt per byte or per word

How DMA Works



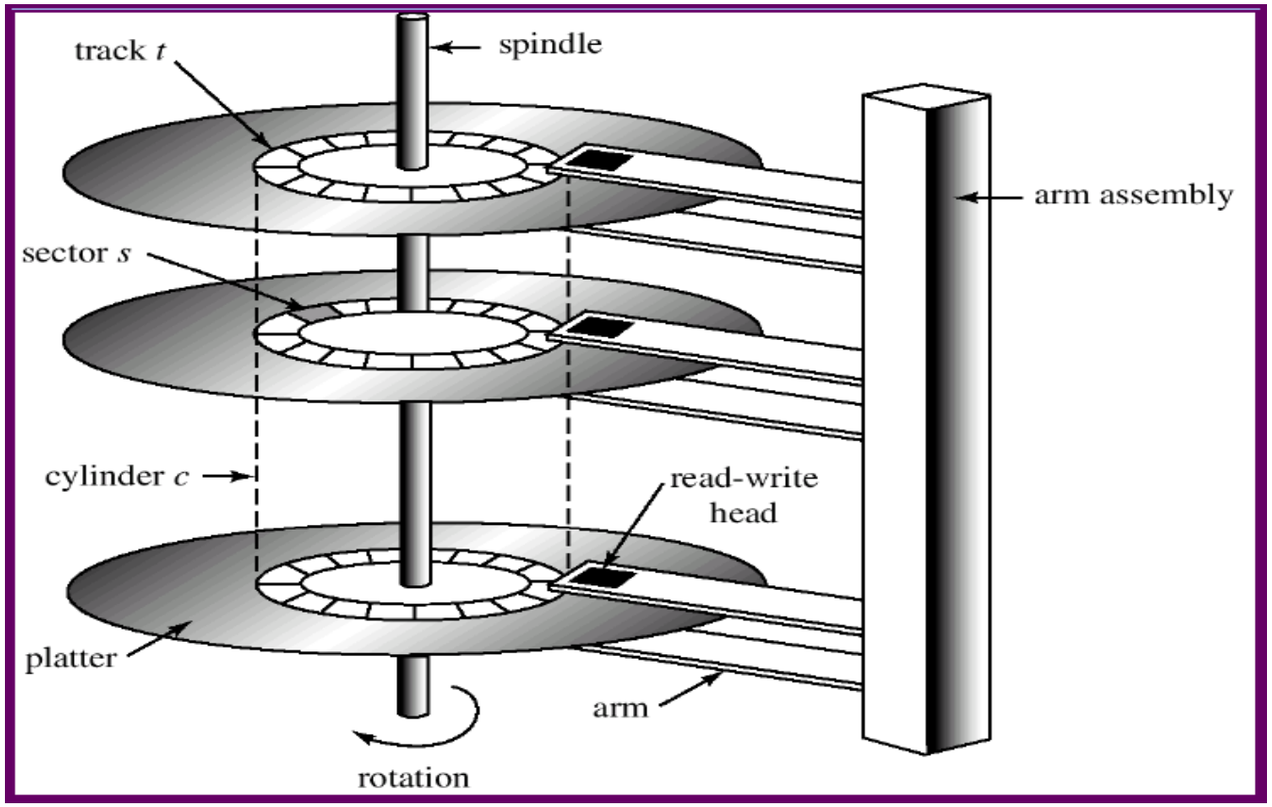
A von Neumann architecture

RAM: Random-Access Memory

- SRAM: static RAM
  - typically six transistors per bit
  - easy interface, fast access, retains content as long as powered
  - used for cache in general CPU; or main memory for embedded MCU
- DRAM: dynamic RAM
  - High density: one transistor per bit, low cost
  - Organized as row/column, more complex access
  - Loses value due to leakage or read => need refreshing!
  - used mainly for main memory in general-purpose PC

Disk Mechanism

- Spinning platters, could be multiple
- Read/write head an arm for each platter
- Transfer time = data size / transfer rate
- Positioning time
  - Seek time (moving head on cylinder to the right track)
  - Rotational latency (waiting for disk to rotate to right sector)



Performance of various levels of storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Coherency and Consistency Issue

- The same data may appear in different levels
  - issue: changing the copy in cache makes it inconsistent with the copy in memory memory!
- Uniprocessor
  - use highest-level copy (assuming all changes come from processor)
- Shared-memory multiprocessor systems
  - Difficult! the processor making the change needs to “invalidate” the other cached copies => forces a cache miss on others, fetch from memory again

Review of Topics

- Storage device hierarchy
- Caching and Issues
- DMA structure
- RAM: SRAM vs DRAM
- Disks
- Cache coherence

Hardware Protection

Dual-Mode Operation
I/O Protection
Memory Protection
CPU Protection

Dual-Mode Operation

- Context: multiple programs sharing resources
  - OS needs to ensure incorrect program can't disrupt or corrupt other programs
- Hardware support using modes of operation
  - **User mode**: execution done for user code
  - **Monitor** mode, aka **kernel** mode, **system** mode, or **privileged** mode

Mode Switch and Privileged Instructions

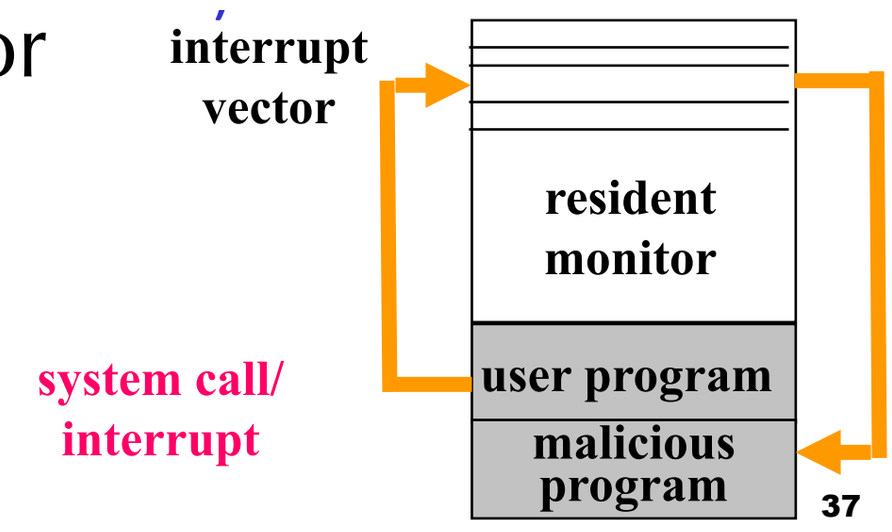
- From User mode to Kernel mode
  - interrupt - triggered by hardware (timer, I/O, signal)
  - trap - executing a trap instruction (`syscall` instruction on MIPS)
  - fault - page fault, divide by zero, etc
- Privileged instructions
  - can be executed ONLY in monitor mode
  - e.g., direct I/O instruction, access special registers, instruction to return from interrupt

Example: `syscall` instruction in MIPS

- Instructions needed in `syscall`
  1. Load the service number in register `$v0`.
  2. Load args into `$a0`, `$a1`, `$a2`, or `$f12` if any
  3. Issue the `syscall` instruction.
  4. Retrieve return values, if any, from result registers as specified.
- `syscall` is not a privileged instruction
  - but it switches mode to privileged mode,
- `ERET` is a privileged instruction
  - returns from a `syscall`, switches from privileged mode to user mode upon return from system call

Protection of I/O

- All I/O instructions must be *privileged instructions* in order to offer protection
  - I/O devices can be shared between users
  - OS must prevent user program from gaining control in monitor mode
    - user program could gain control by overwriting an entry in the interrupt vector

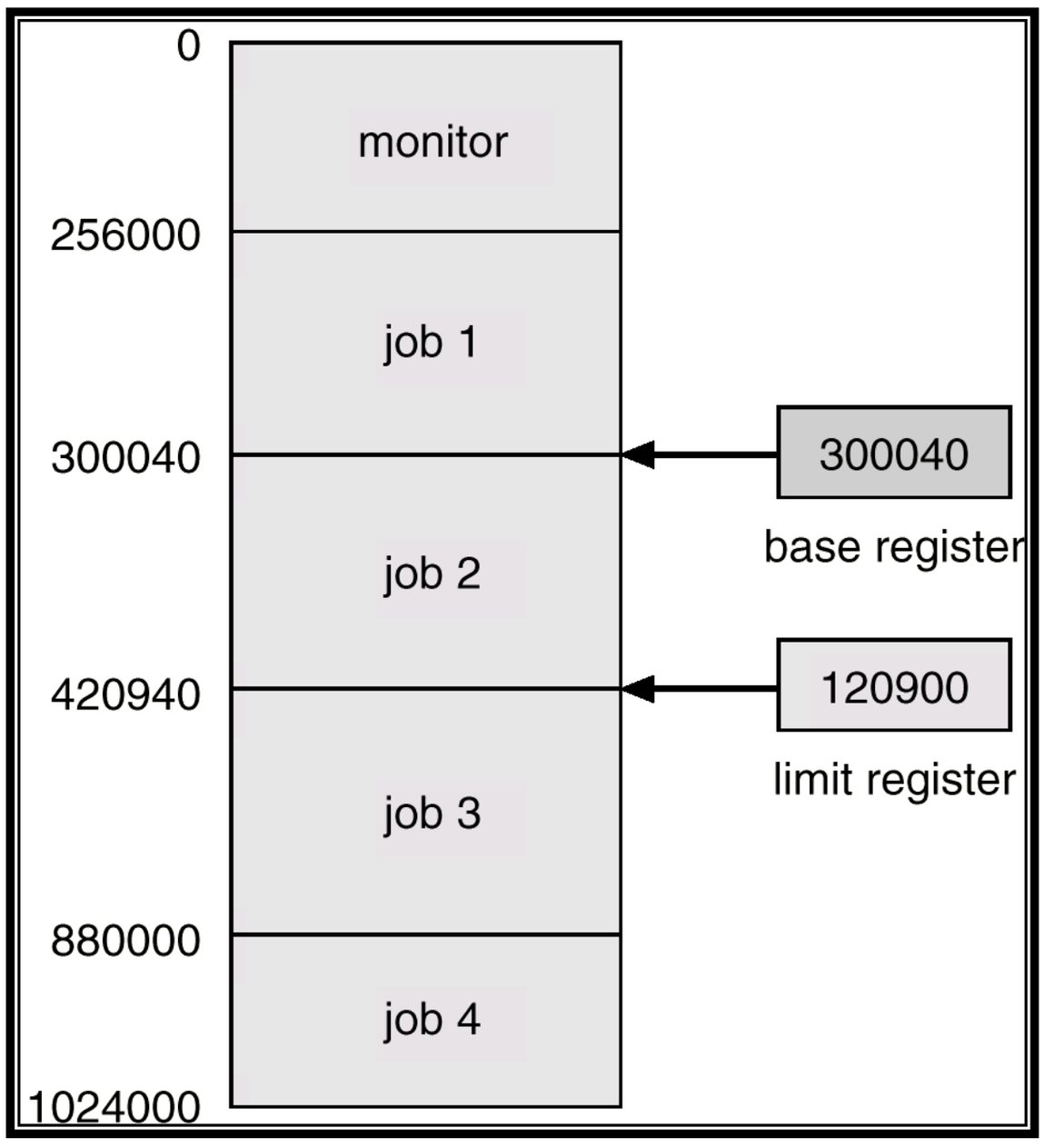


Protection of Memory

- Memory to protect
  - Interrupt vector and ISR (shouldn't be user writable)
  - Data owned by other user processes (shouldn't be readable or writable, unless the user allows part of it)
- Possible hardware support using registers
  - Base register: starting address of legal physical men address
  - Limit registers : size of the range
  - Attempt for user program to access memory outside the defined range => caught by hardware, handled by OS

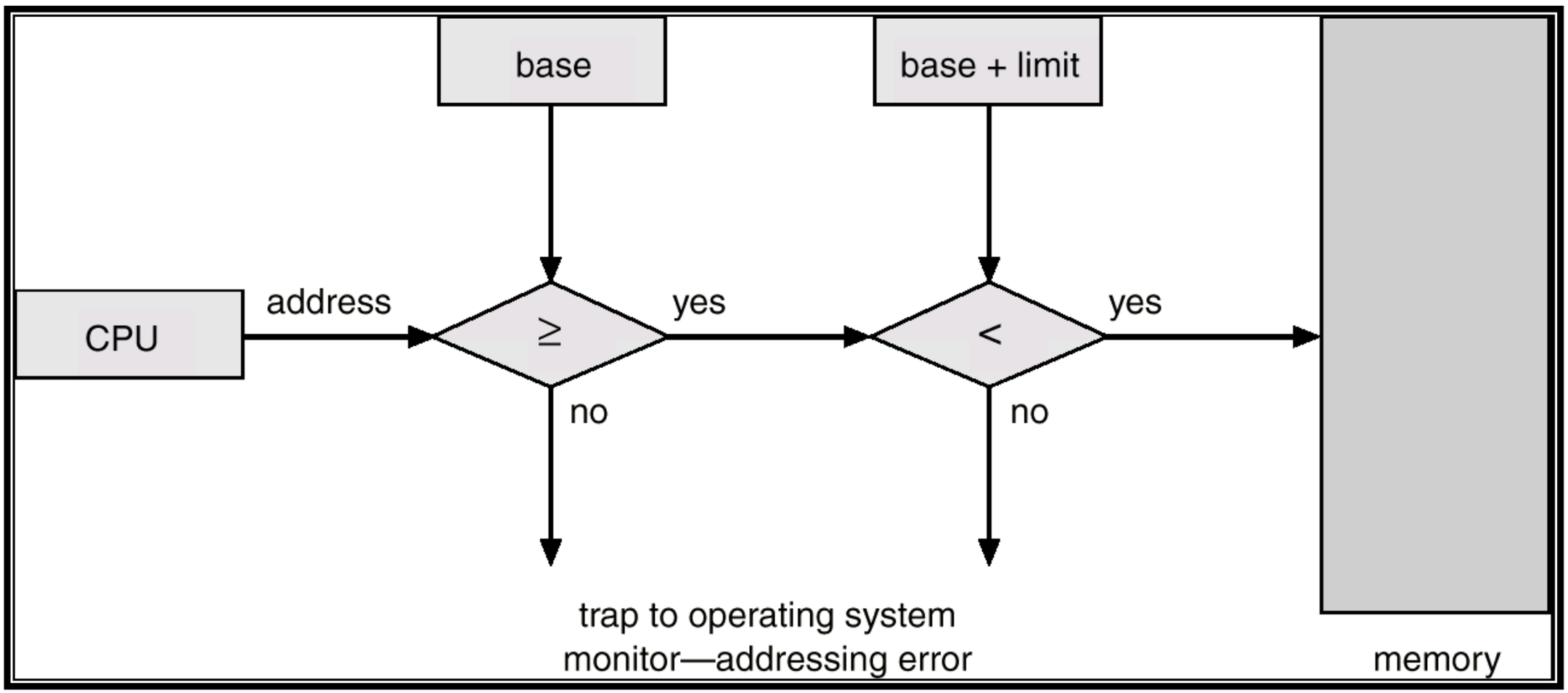
Use of Base and Limit Registers

- Base: 300040
- Limit: 120900
- Ending address: 42939



Hardware Address Protection

- Flowchart

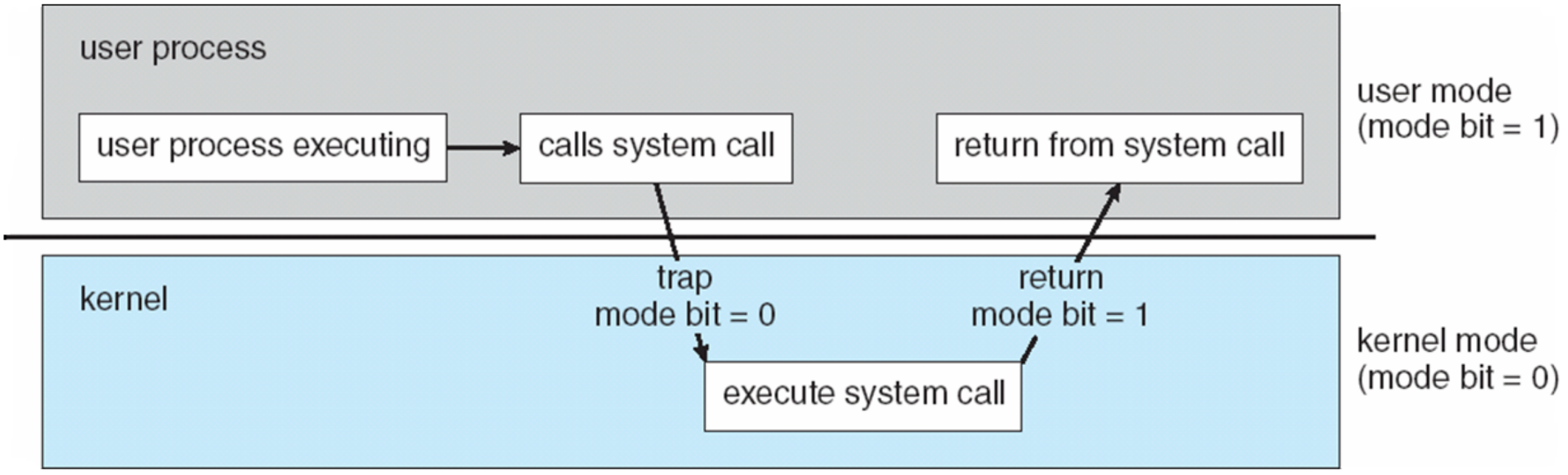


Protection of CPU

- Prevent user program from hogging CPU
  - infinite loop (intentional or unintentional)
  - not making system calls
- Hardware support: Timer
  - interrupts after a specified number of cycles
  - timer can count number of clocks
  - commonly used for time sharing
- Load Timer => privileged instruction

OS regaining control by timer interrupt: timer

- Timer is set to interrupt the computer after some time
  - Operating system set a timer (privileged instruction)
  - Counter == 0 => generate an interrupt
- OS defines the ISR for timer
  - Allows scheduler/dispatcher to context switch or terminate program



Review

- Dual mode operation
  - Different ways to switch to kernel mode
  - Privileged instructions and uses
- Protections
  - I/O access: privileged I/O instructions
  - Memory access: use base and bound registers
  - CPU use: hardware mechanism for OS to regain control