Operating System Concepts

Lecture 2: Interaction with Hardware

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Today's class

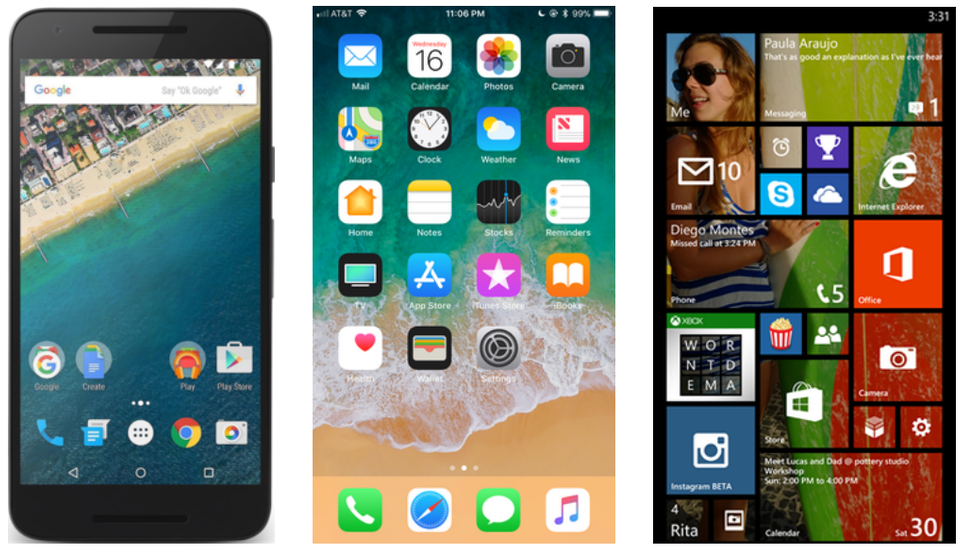
- Why study OS?
- Interaction between OS and hardware
  - CPU, registers, memory, device controllers, interrupt architecture,

Why study OS?

- knowledge of how OS works is crucial for efficient and secure programming
  - OS code is **efficient** (runs fast using minimum resources), **reliable** (fails infrequently), **complex** (addresses various timing and concurrency issues), and **secure**
- studying OS is studying the design of large software systems
  - Windows Vista is more than 50 million lines of code
- studying OS makes you a better programmer!
  - teaches you how to make tradeoffs between performance and usability, performance and design simplicity, ….
  - teaches you how to manage complexity through appropriate abstractions (e.g., file system, process)
- it's an active area of research!
  - new application spaces: cloud, mobile and edge systems, embedded and real-time systems
  - SOSP: ACM Symposium on Operating Systems Principles
  - OSDI: USENIX Symposium on Operating Systems Design and Implementation

OS is everywhere!

Desktop & Laptop Computers

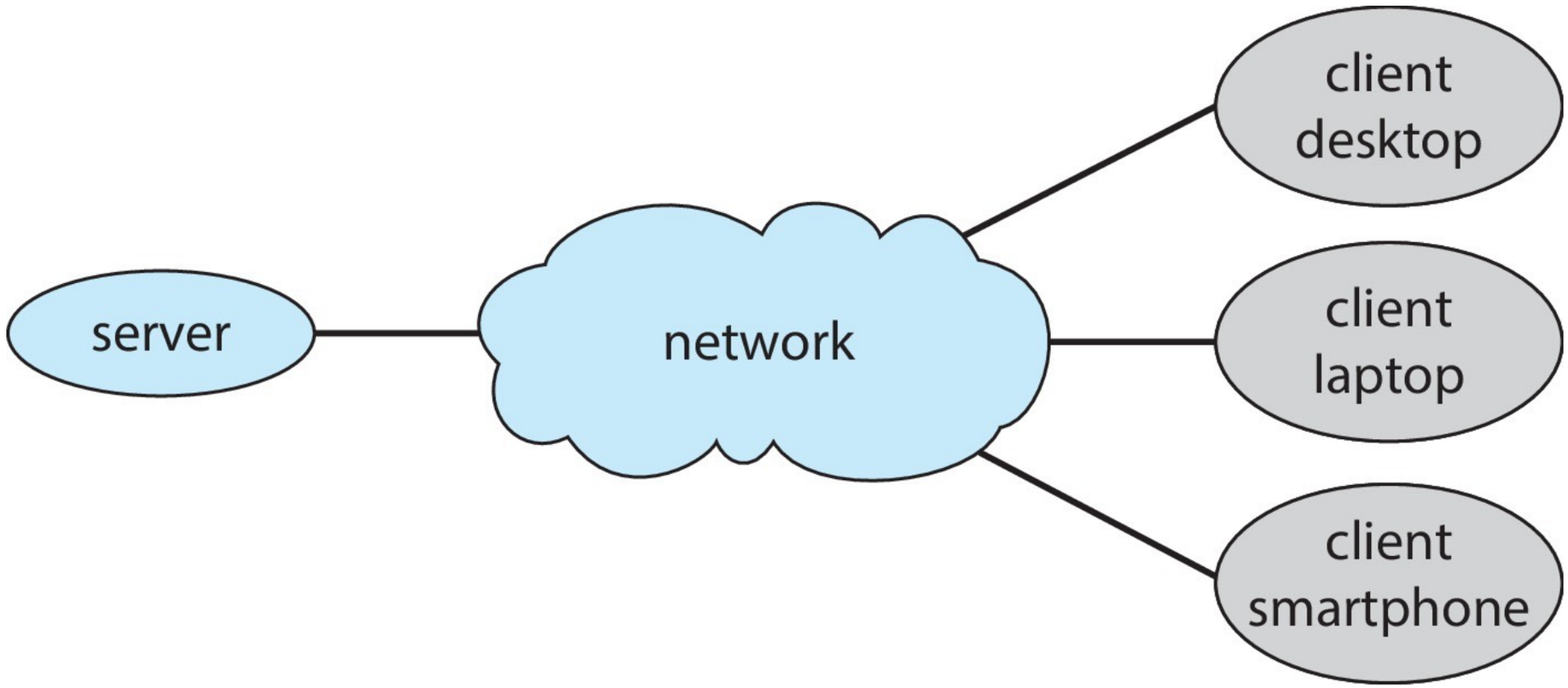


Mobile Devices

Embedded Computers

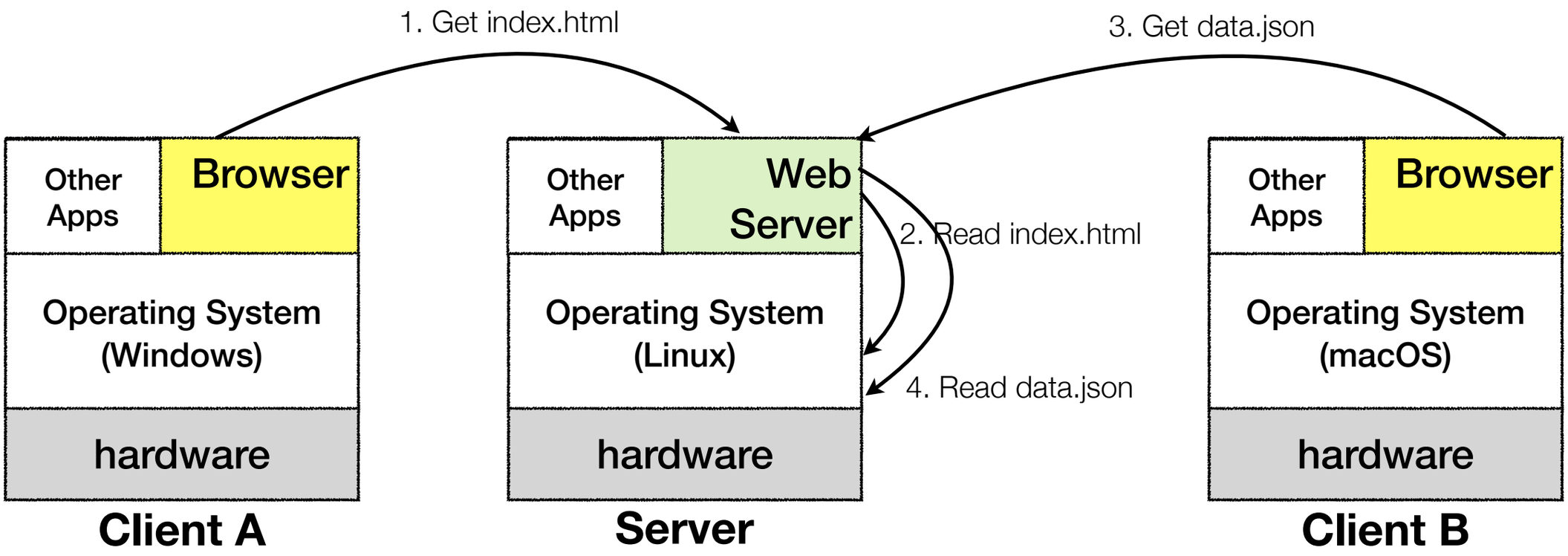
Wearables

Example: the client-server model



- each client can send a request to the server to perform some operation (e.g., running a query)
- server performs the operation upon receiving the request, and sends the results back to the client

Example: the client-server model



- You will learn how OS
  - allows multiple (local or remote) user programs to communicate with each other, sharing data
  - handles concurrent requests
  - supports access to shared data using atomic operations
  - protects the system against malicious scripts

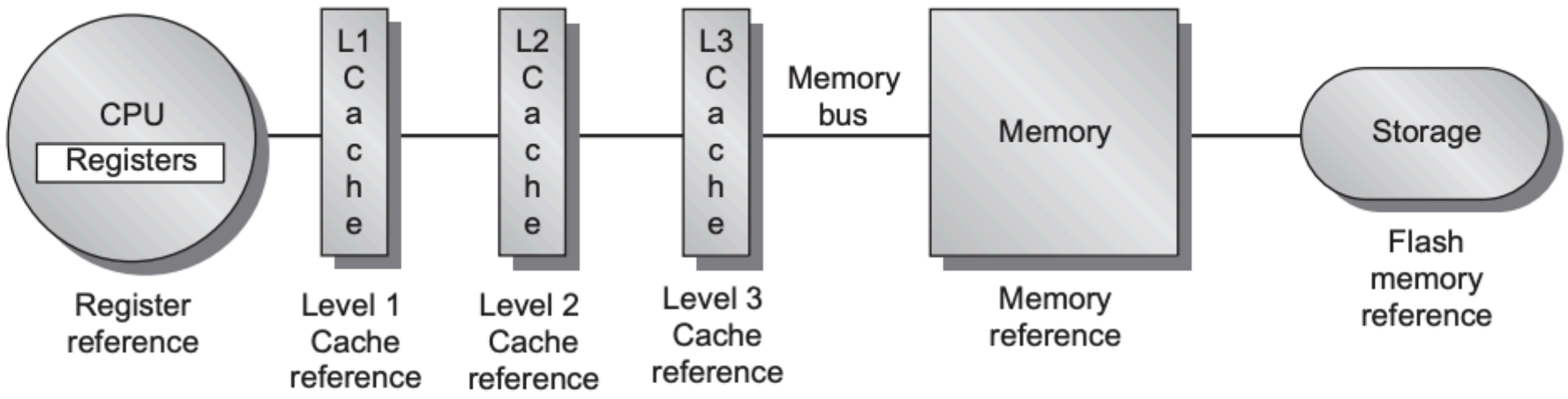
Interfacing Hardware

Components

- CPUs, registers, 3 levels of cache, memory, disk
- System bus connecting CPUs, controllers (disk, I/O), and memory
- I/O devices and controllers
  - more than one device can be attached to a single controller
  - CPU and device controllers run in parallel, yet they compete for memory cycles!
  - controller maintains local buffer and special-purpose registers
  - for each device controller, there's a device driver in OS that knows how to interact with the device controller (e.g. loading its registers before I/O begins)
  - controller notifies device driver (OS) via an interrupt (signal sent to CPU)

Memory hierarchy — recap

- CPU fetches and runs instructions one at a time
  - registers are CPU's workspace
- Programmers desire a large amount of fast memory (expensive!)
  - an economical solution: a memory hierarchy organized into several levels; each level is smaller, faster, and more expensive per byte than the next
  - it takes advantage of **locality** and **cost-performance tradeoffs**



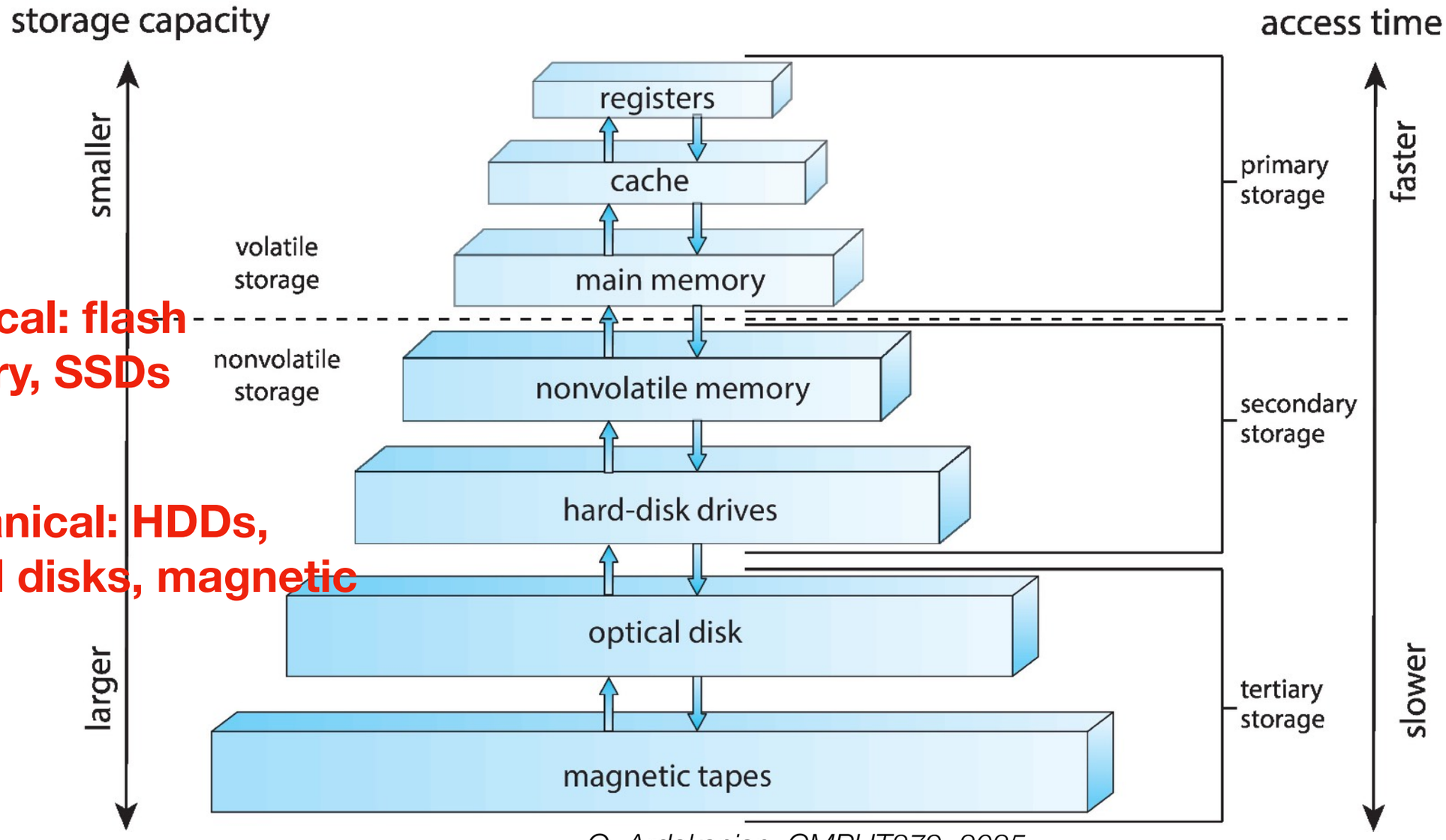
Laptop	Size:	1000 bytes	64 KB	256 KB	4-8 MB	4–16 GB	256 GB-1 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS
Desktop	Size:	2000 bytes	64 KB	256 KB	8-32 MB	8–64 GB	256 GB-2 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS

10^{-12} 10^{-6}

Memory hierarchy for a desktop and a laptop

Memory hierarchy — recap

- Storage systems primarily differ in speed (access time), size and volatility
- Volatile storage loses its content when power is lost

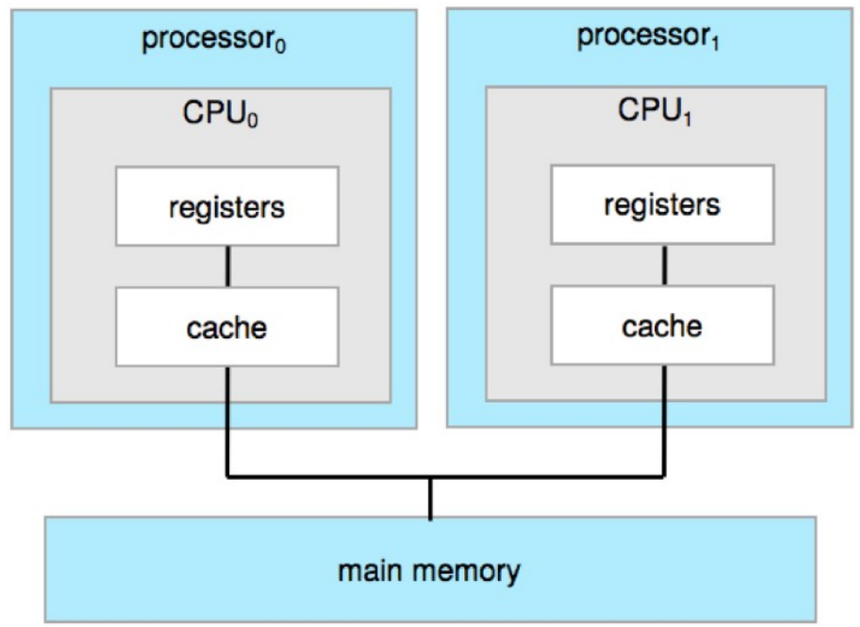


Electrical: flash memory, SSDs

Mechanical: HDDs, optical disks, magnetic tapes

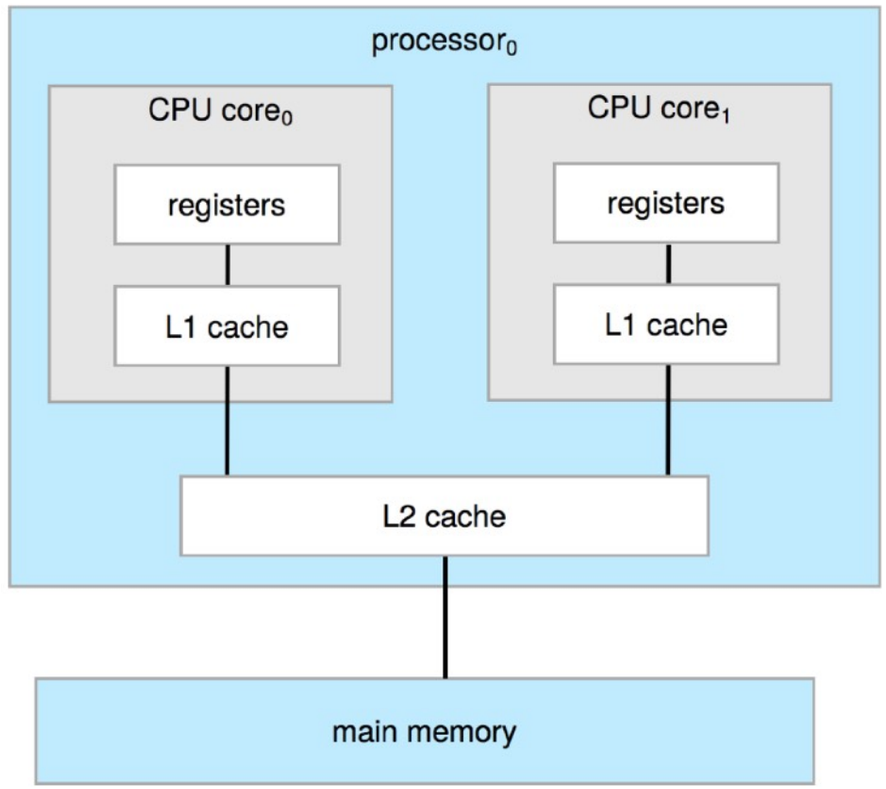
Single-processor vs. multiprocessor systems

- Single-processor system contains a single general-purpose processor with one processing core
  - there might be several device-specific processors (for disk, keyboard, and graphics controllers) but they do not run user processes
- Multiprocessor system contains two (or more) processors each with a single-core CPU
  - each processor has its own set of registers and cache but they all share the main memory
  - multiple processes can run simultaneously
    - increased throughput
  - load balancing might be necessary



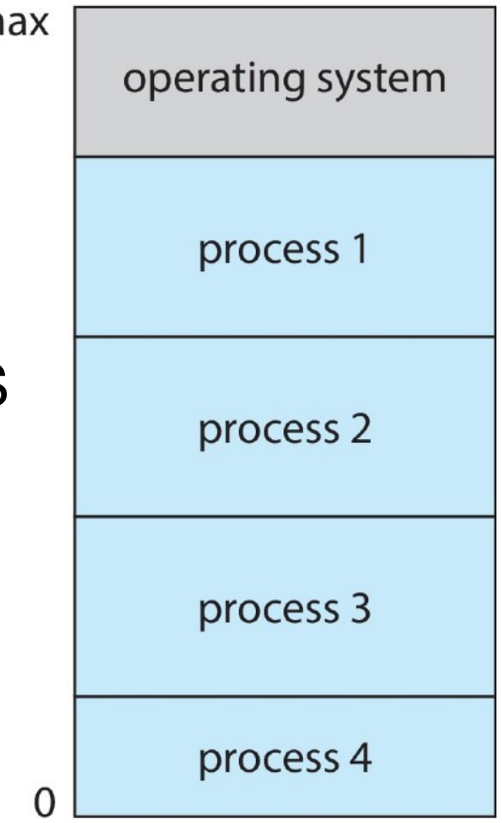
Multicore systems

- Multicore system has multiple cores residing on a single processor chip
  - each core has its own registers and L1 cache; L2 cache is shared between processing cores
  - it can be more efficient than a multiprocessor system, because on-chip communication is **faster** than inter-chip communication, and uses much **less power**
- N cores appear to OS as N CPUs



Multiprogramming and multitasking

- Multiprogramming
  - OS keeps several processes in memory simultaneously
  - OS picks one of them for execution (**scheduling**); when that process terminates or has to wait (e.g., for I/O), OS runs another process on the CPU
  - Unlike a non-multiprogrammed system, the CPU wouldn't be idle as long as at least one process has to execute
  - This increases **CPU utilization**



- Multitasking is an extension of multiprogramming
  - CPU executes multiple processes by switching between them frequently to ensure reasonable **response time** (typically less than 1 second)
  - Choosing the process that runs next requires scheduling

How does multiprogramming work?

- When a process waits for I/O, multiprogramming allows CPU to execute another process, overlapping I/O and computation
  - the process waiting for I/O is added to the waiting queue of the I/O device (e.g., terminal, disks, video board, printer, network card)
- Buffering
  - data fills the local buffer of I/O device
- Interrupt handling
  - I/O events trigger a signal (known as **interrupt**)
  - control is then transferred to the OS
  - OS adds the process back into the ready queue

Interrupt

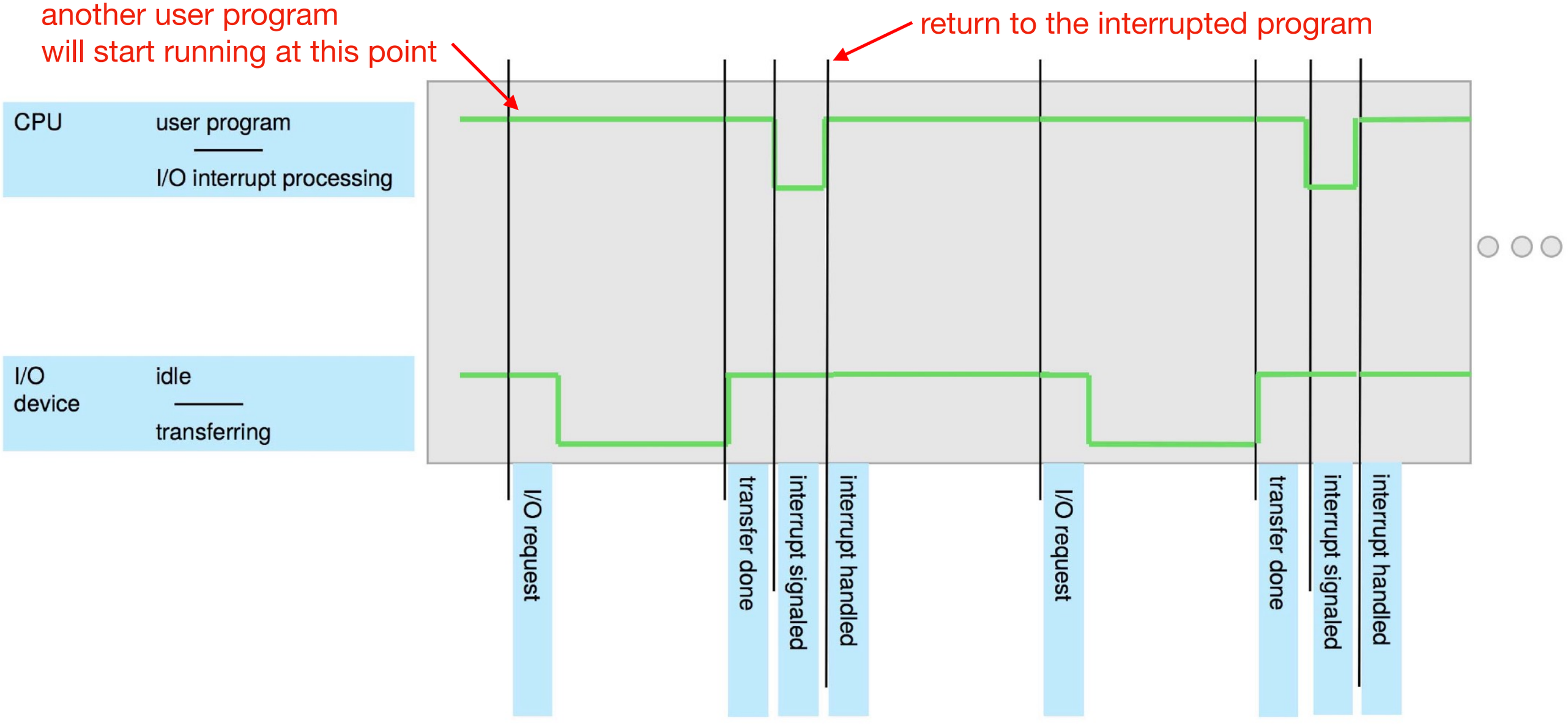
- Definition: a *signal* sent by a **device controller** to CPU's interrupt-request-line to inform that an event has occurred (e.g., I/O request completed)
  - each device controller has its own small processor that executes **asynchronously** with the main CPU
  - CPU senses the interrupt-request-line after executing every instruction
  - if a controller raises an interrupt, CPU stops whatever it was doing, catches the interrupt (reads the interrupt number), and dispatches it to the interrupt-handler (aka interrupt service routine) through the in-memory **interrupt vector** containing the starting addresses of all handlers;
    - interrupt number is an index into the interrupt vector
  - the interrupt-specific handler clears the interrupt by servicing the device
  - the handler also saves the state information of the interrupted program by storing registers and program counter; it restores the state after clearing the interrupt
    - necessary because the interrupt handler may need to modify the CPU state

0: 0x2ff080000	keyboard
1: 0x2ff100000	mouse
2: 0x2ff100480	timer
3: 0x2ff123010	Disk 1

starting address of the interrupt service routine

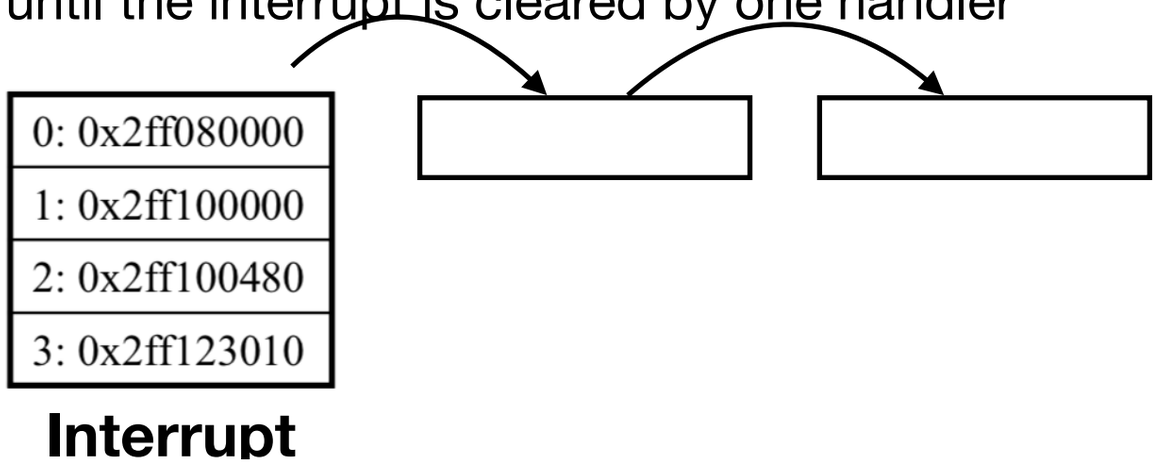
Interrupt-based asynchronous I/O

After I/O starts, control returns to user program without waiting for I/O completion



Interrupt handling — other considerations

- How to defer interrupt handling during the execution of a critical instruction sequence? two interrupt-request-lines
  - CPU has two interrupt-request-lines: non-maskable and maskable
  - device controllers raise an interrupt through the maskable line, CPU turns off this line before running critical instructions
  - the non-maskable line is reserved for hardware faults, such as unrecoverable memory errors
- What if there are more interrupt handlers than addresses available in the interrupt vector? interrupt chaining
  - each element in the interrupt vector points to the head of a list of interrupt handlers; handlers in this list are called one by one until the interrupt is cleared by one handler



Interrupt handling — other considerations

- How to defer interrupt handling during the execution of a critical instruction sequence? two interrupt-request-lines
  - CPU has two interrupt-request-lines: non-maskable and maskable
  - device controllers raise an interrupt through the maskable line
  - the non-maskable line is reserved for hardware faults such as unrecoverable memory errors
- What if there are more interrupt handlers than addresses available in the interrupt vector? interrupt chaining
  - each element in the interrupt vector points to the head of a list of interrupt handlers; handlers in this list are called one by one until the interrupt is cleared by one handler
- How to distinguish urgent events from non-urgent ones? interrupt priority levels
  - CPU defers the handling of low-level interrupts to handle a high-level interrupt

Trap

- Definition: a trap or exception is a software-generated interrupt caused by an error (e.g., division by zero) or a user request for OS service (i.e. **system call**)

0: 0x00080000	Illegal address
1: 0x00100000	Memory violation
2: 0x00100480	Illegal instruction
3: 0x00123010	System call

Trap Vector

Protection

- Hardware has a status bit that indicates the current mode (user or kernel)
  - there could be more than two operation modes
    e.g., ARMv8 systems have seven modes
  - user programs run in the user mode
  - kernel code runs in the kernel mode with **full privileges** of hardware
- Operation modes provide the means for protecting OS from errant users
  - code could be buggy or malicious
- Examples of privileged instructions are I/O control, timer management, interrupt management; they can only run in the kernel mode
- Invoking a system call allows user program to run privileged instructions

crossing protection boundary