R

بسم الله الرحمن الرحیم

سیستم عامل

جلسه دوم – معماری کامپیوتر

آنچه گذشت

جلسه‌ی قبل، معرفی درس

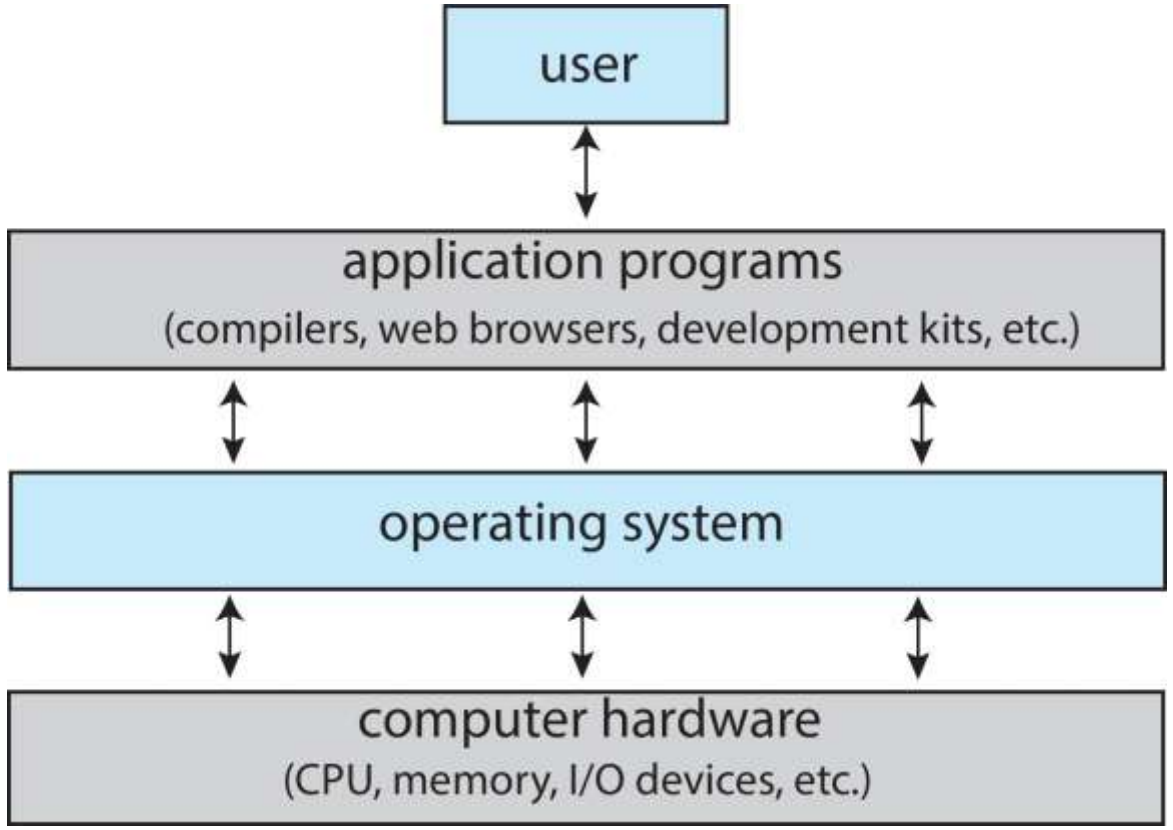
قوانین درس

- تمرین اول منتشر شده و تا ۲۰ مهر وقت دارید.
  - شروع کنید که گیر و گور نخورید...
  - کلاس رفع اشکال تمرین ۱۲ مهر، ساعت ۱۹
- سایت alireza.dev/teaching/os1403
  - تقویم درس، زمان آزمونک‌ها و تمرین‌ها و میان‌ترم
    - آزمونک نخست: ۱۶ مهر
    - میان‌ترم: ۲۱ آبان
  - اسلایدهای هر جلسه، لینک تمرین‌ها
  - لینک عضویت در کوئرا

کارهایی که سیستم عامل می‌کند

- وظیفه‌ی Abstract Machine
- وظیفه‌ی Resource Mangement

Abstract Machine



OS as a resource manager

- Allocating resources to applications
- Making efficient use of limited resources
- Protecting applications from each other

OS is just a program

- So it needs hardware support…

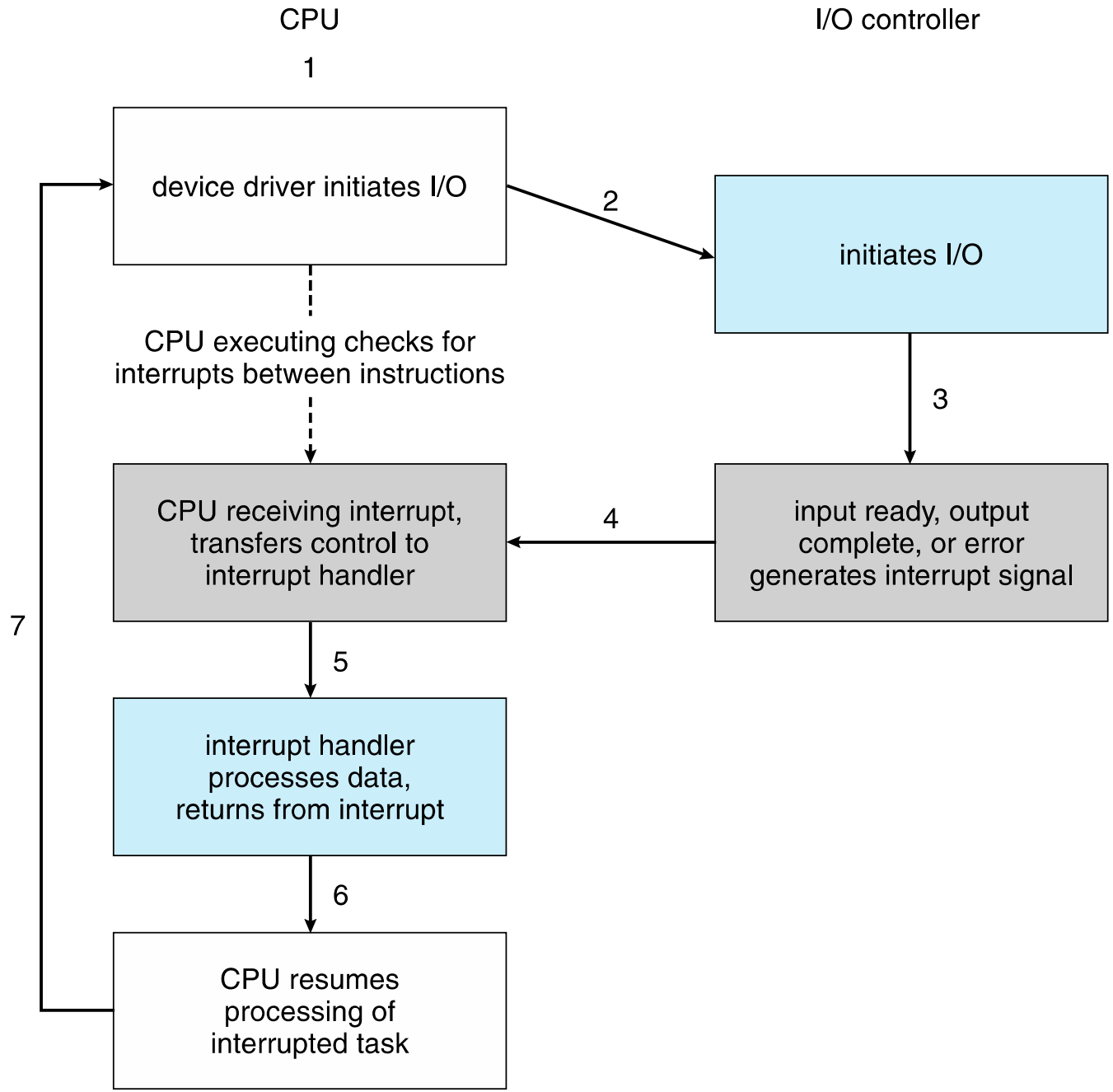
Boot

- **Bootstrap program** is loaded at power-up or reboot
  - *Typically stored in ROM or EPROM, generally known as **firmware***
  - *Initializes all aspects of system*
  - *Loads operating system kernel and starts execution*

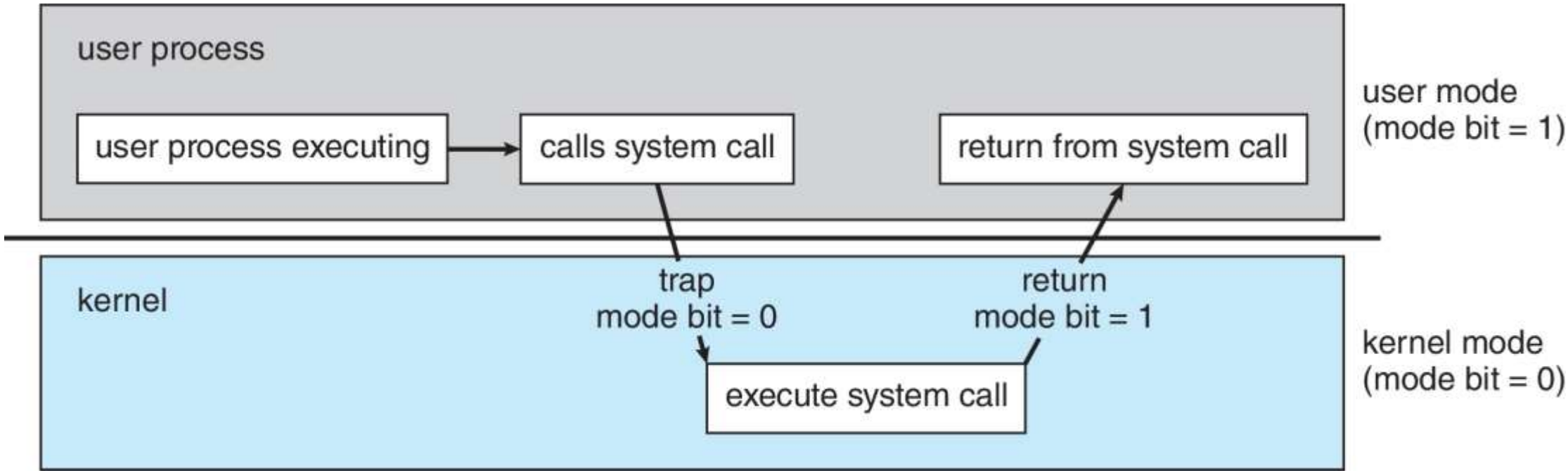
Boot

- When power initialized on system, execution starts at a fixed memory location
- Operating system must be made available to hardware so hardware can start it
  - *Small piece of code – **bootstrap loader**, **BIOS**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it*
  - *Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk*
  - *Modern systems replace BIOS with **Unified Extensible Firmware Interface (UEFI)***
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**

Interrupt


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for
interrupts between instructions
3
CPU receiving interrupt,
transfers control to
interrupt handler
4
input ready, output
complete, or error
generates interrupt signal
7
5
interrupt handler
processes data,
returns from interrupt
6
CPU resumes
processing of
interrupted task

Trap


user process
user process executing
calls system call
return from system call
user mode
(mode bit = 1)
kernel
trap
mode bit = 0
return
mode bit = 1
execute system call
kernel mode
(mode bit = 0)

این جلسه

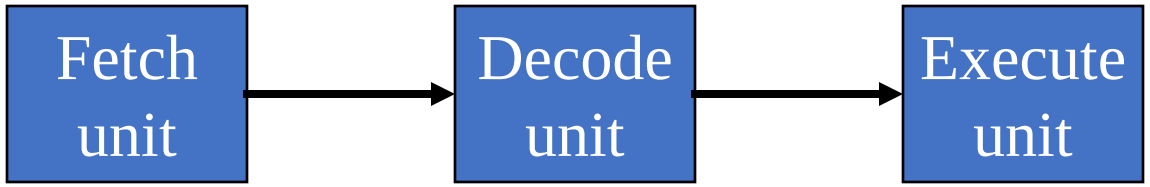
معرفی اجزاء سیستم عامل

پردازنده

Real World Complexity

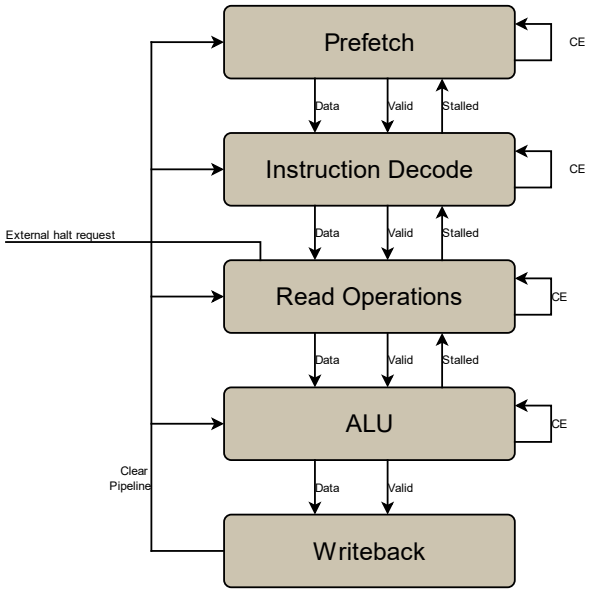
- Our simple model applies, but real systems have more complexity
  - *Pipelined CPUs*
  - *Superscalar CPUs*
  - *Multi-level memory hierarchies*
  - *Virtual memory*
  - *Complexity of devices and buses*

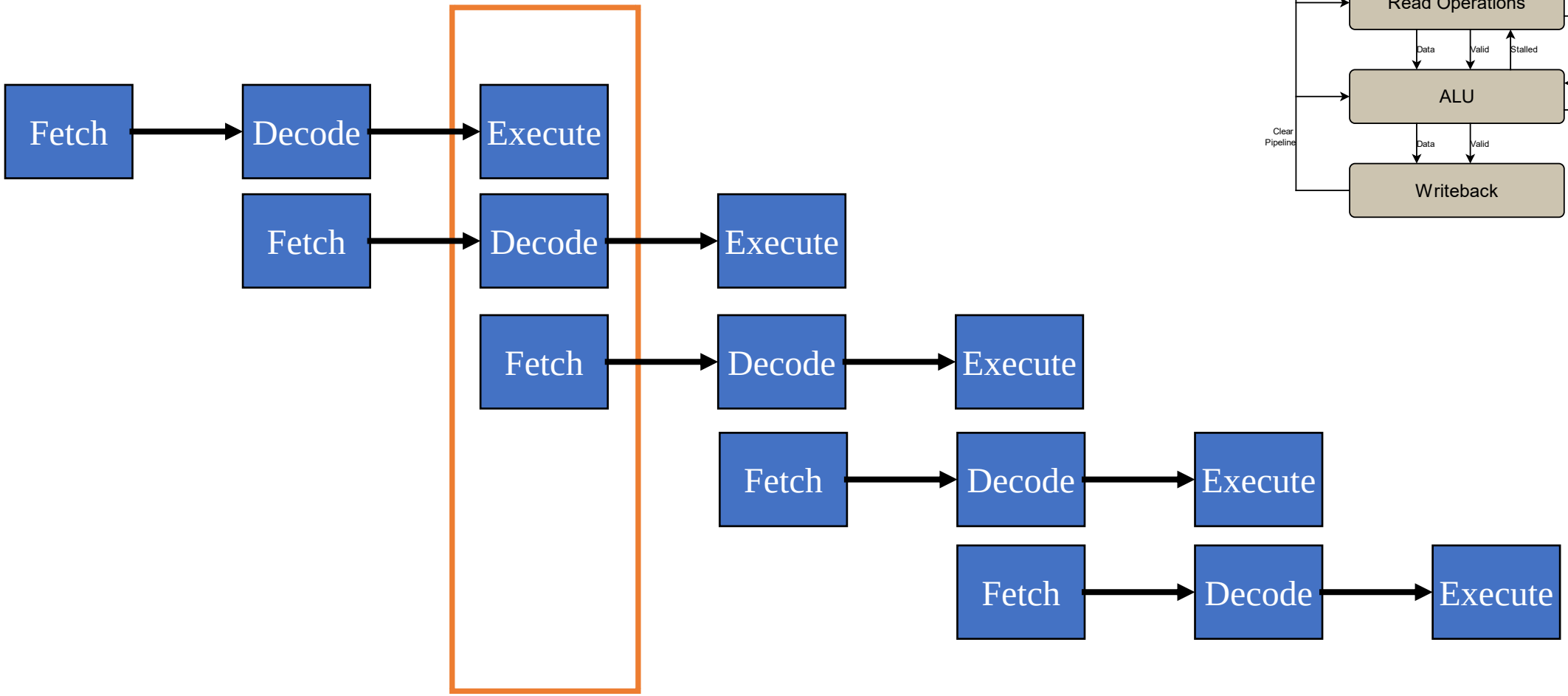
Pipelined CPUs



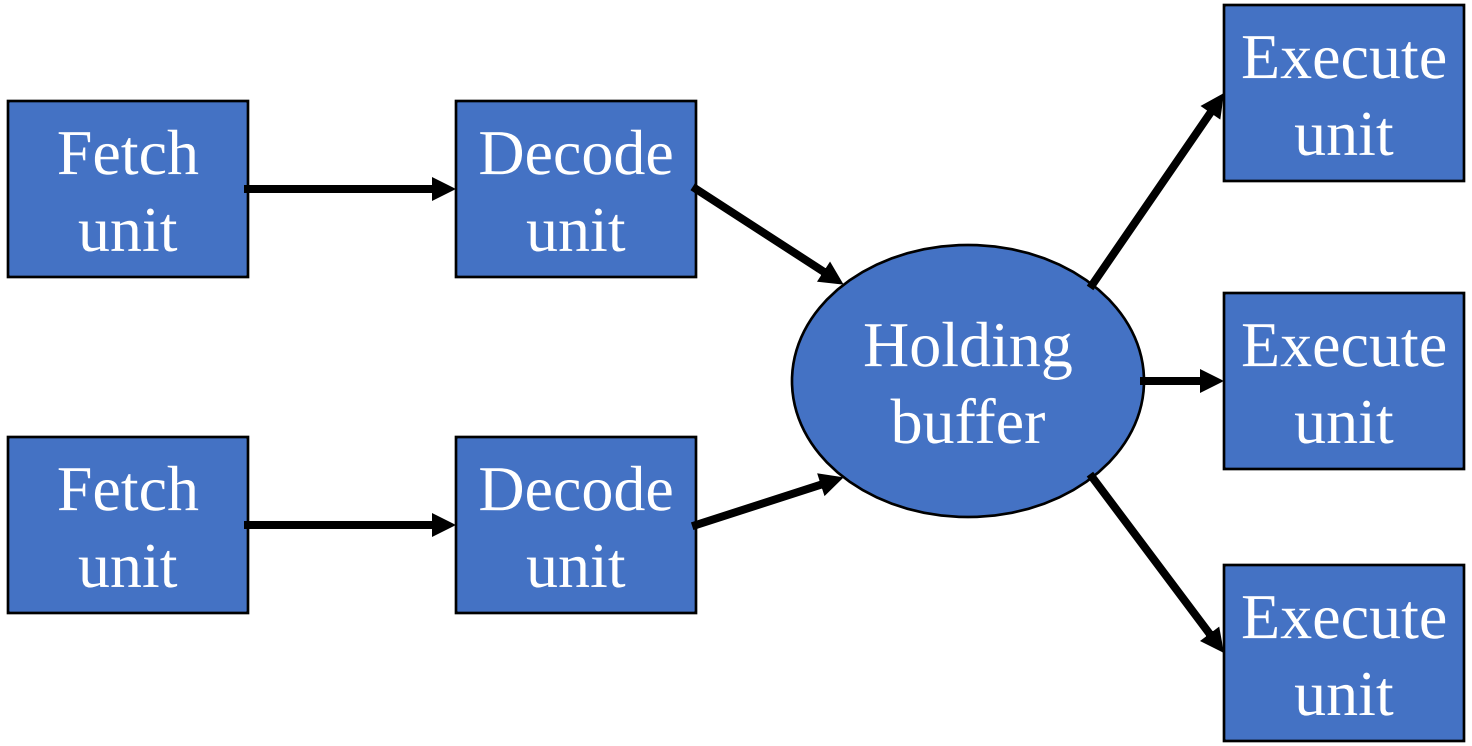
Execution of current instruction performed in parallel with decode of next instruction and fetch of the one after that

Pipelined CPUs





Superscalar CPUs



What does this mean for the OS?

- Pipelined CPUs
  - *more complexity in capturing the state of a running application*
  - *more expensive to suspend and resume applications*
- Superscalar CPUs
  - *even more complexity in capturing state of a running application*
  - *even more expensive to suspend and resume applications*
  - *support from hardware is useful ie. precise interrupts*
- More details, but fundamentally the same task
- The BLITZ CPU is not pipelined or superscalar and has precise interrupts

Multiprocessing

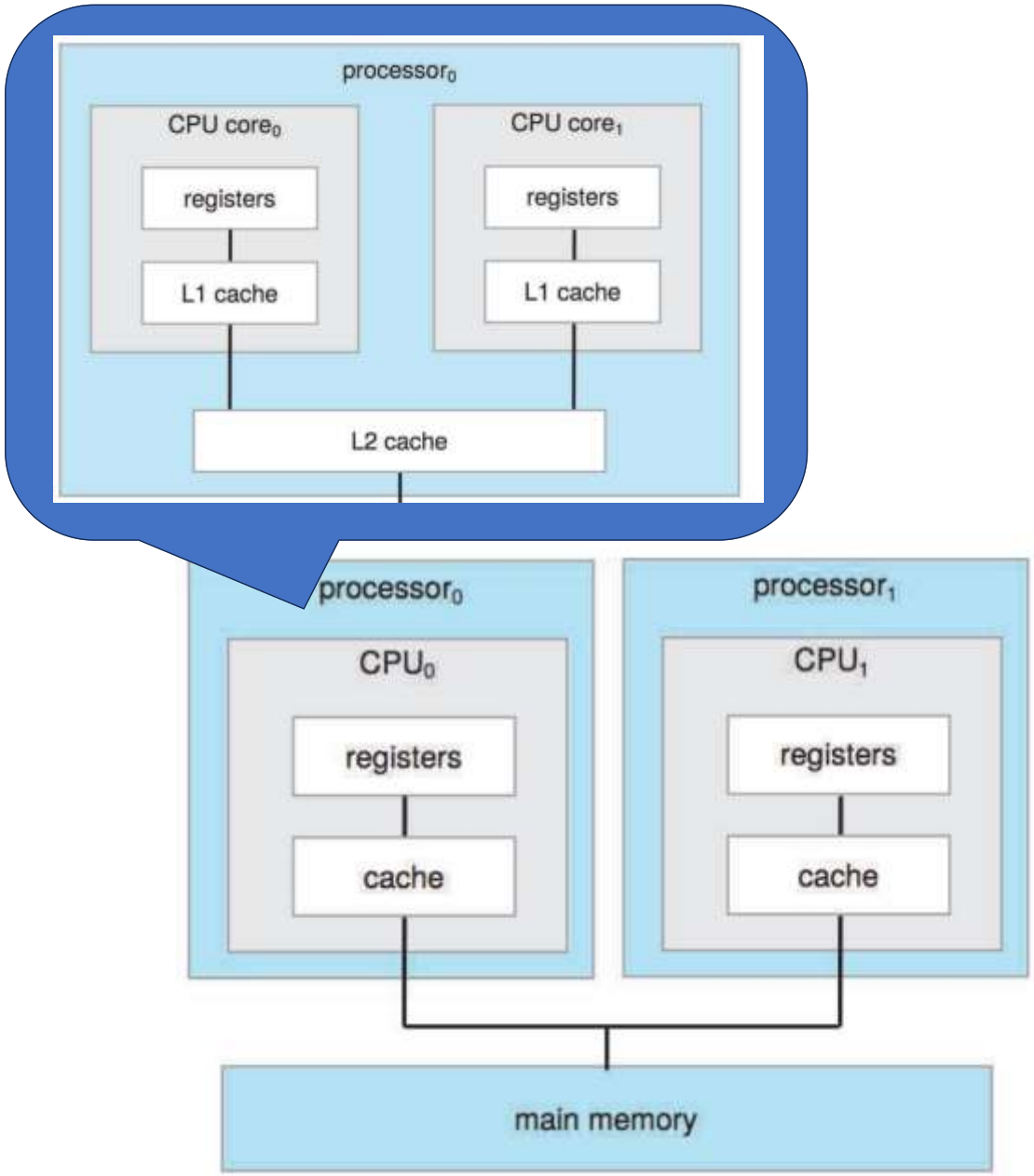


Figure 1.8 Symmetric multiprocessing architecture.

Non-uniform memory access

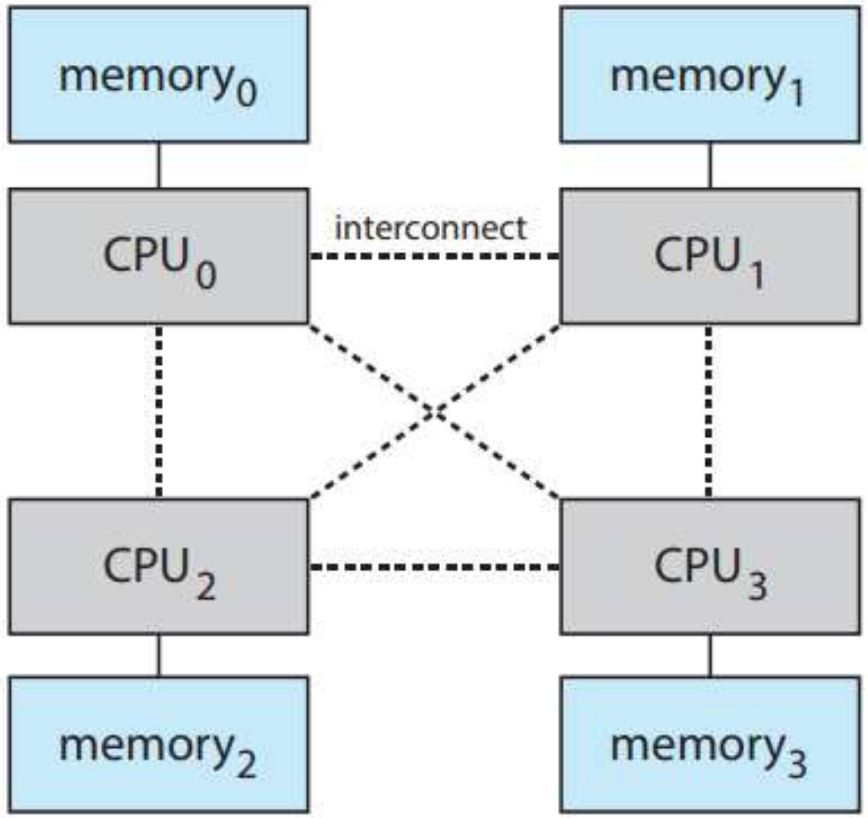


Figure 1.10 NUMA multiprocessing architecture.

حافظه

The memory hierarchy

- 5GHz processor → 0.2 ns clock cycle
- Data/instruction cache access time → 0.5ns – 10 ns
  - *This is where the CPU looks first!*
  - *Memory this fast is very expensive !*
  - *Size: too small for whole program*
- Main memory access time → 60 ns
  - *Slow, but cheap*
  - *Size 4GB+*
- Magnetic disk →
  10 ms, 200+ Gbytes

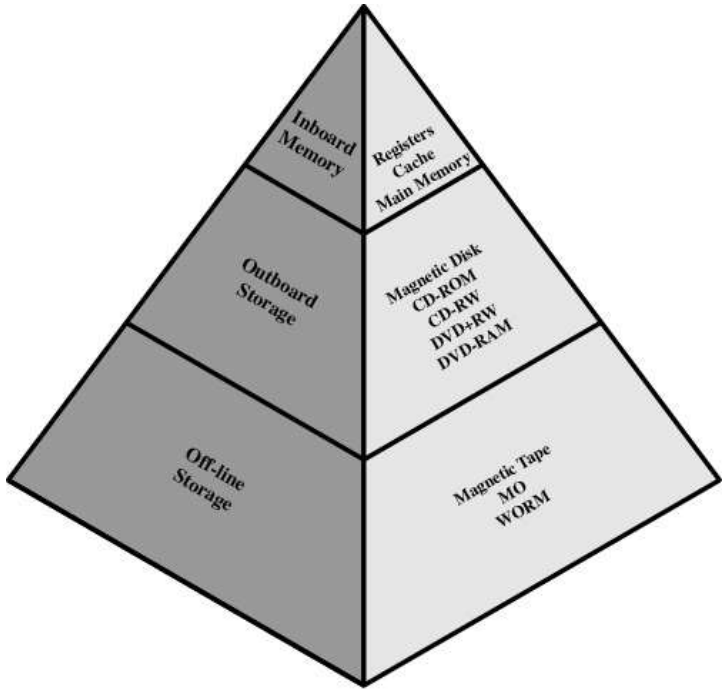
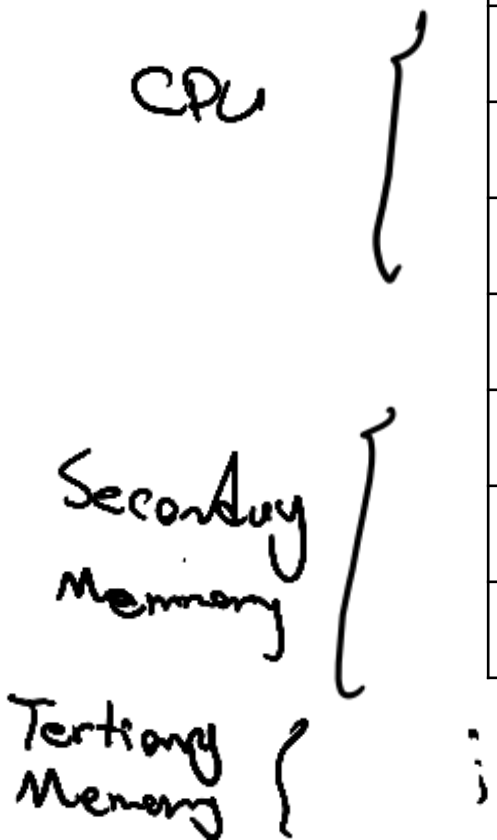


Figure 1.14 The Memory Hierarchy

The memory hierarchy



Memory Type	Access Time
L1 Cache	0.5 ns to 1 ns
L2 Cache	3 ns to 10 ns
L3 Cache	10 ns to 20 ns
Main Memory (DDR4/5)	50 ns to 80 ns
SSD (NVMe)	50 µs to 100 µs
SSD (SATA)	100 µs to 500 µs
HDD	5 ms to 15 ms

The memory hierarchy

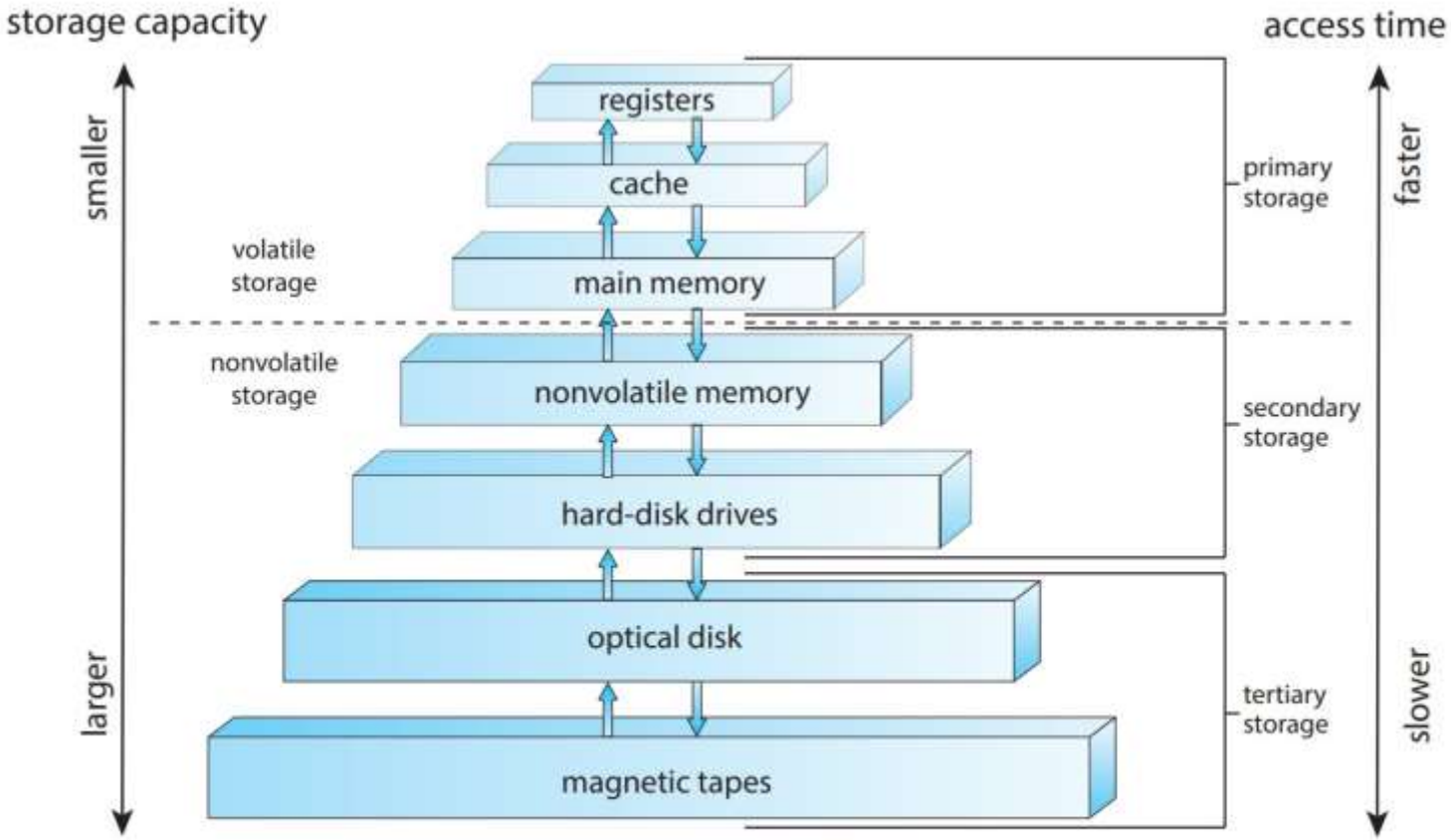


Figure 1.6 Storage-device hierarchy.

Managing the Memory Hierarchy

- Movement of data from main memory to cache
  - *Cache lines loaded on demand automatically*
  - *Placement and replacement policy fixed by hardware*
- Movement of data from cache to main memory affected by OS
  - *Instructions for "flushing" the cache*
  - *Used to maintain consistency of main memory*
- Movement of data among lower levels of the memory hierarchy is under direct control of the OS
  - *Virtual memory page faults and file system calls*

Memory Hierarchy Challenges

- How do you keep the contents of memory consistent across layers of the hierarchy?
- How do you allocate space at layers of the memory hierarchy “fairly” across different applications?
- How do you hide the latency of the slower subsystems, such as main memory and disk?

Other Memory-Related Issues

- How do you protect one application's area of memory from that of another application?
- How do you *relocate* an application in memory?
  - *How does the programmer know where the program will ultimately reside in memory?*

Memory Protection & Relocation

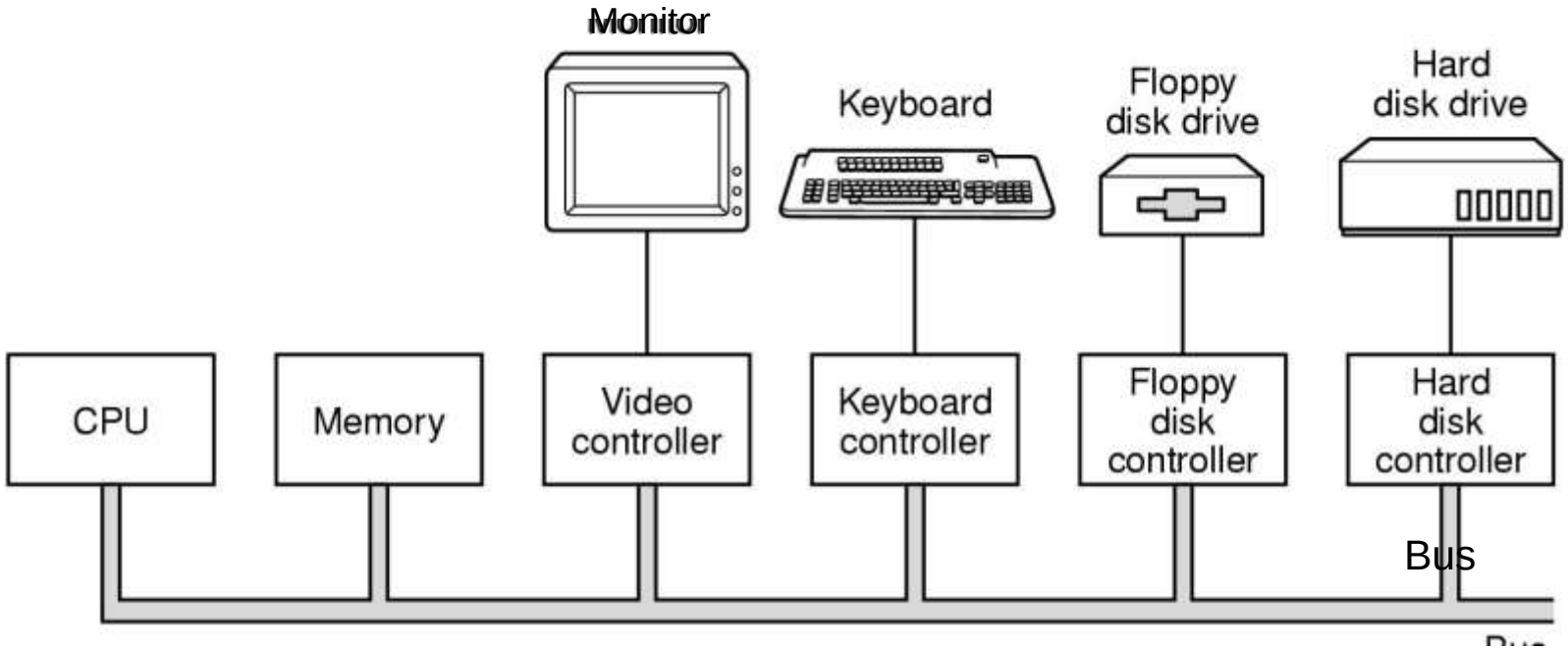
- Virtual vs physical addresses
  - *Address range in each application starts at 0*
  - *Applications use virtual addresses, but hardware and OS translate them automatically into physical addresses*
- Base register concept
  - *Get the CPU to interpret address indirectly via a base register*
  - *Base register holds starting, or base address*
  - *Add base value to address to get a real address before main memory is accessed*
- Relocation is simple
  - *Just change the base register value!*

Paged virtual memory

- The same basic concept, but ...
  - *Supports non-contiguous allocation of memory*
  - *Allows processes to grow and shrink dynamically*
  - *Requires hardware support for page-based address translation*
    - Sometimes referred to as a memory management unit (MMU) or a translation lookaside buffer (TLB)
- Much more on this later ...

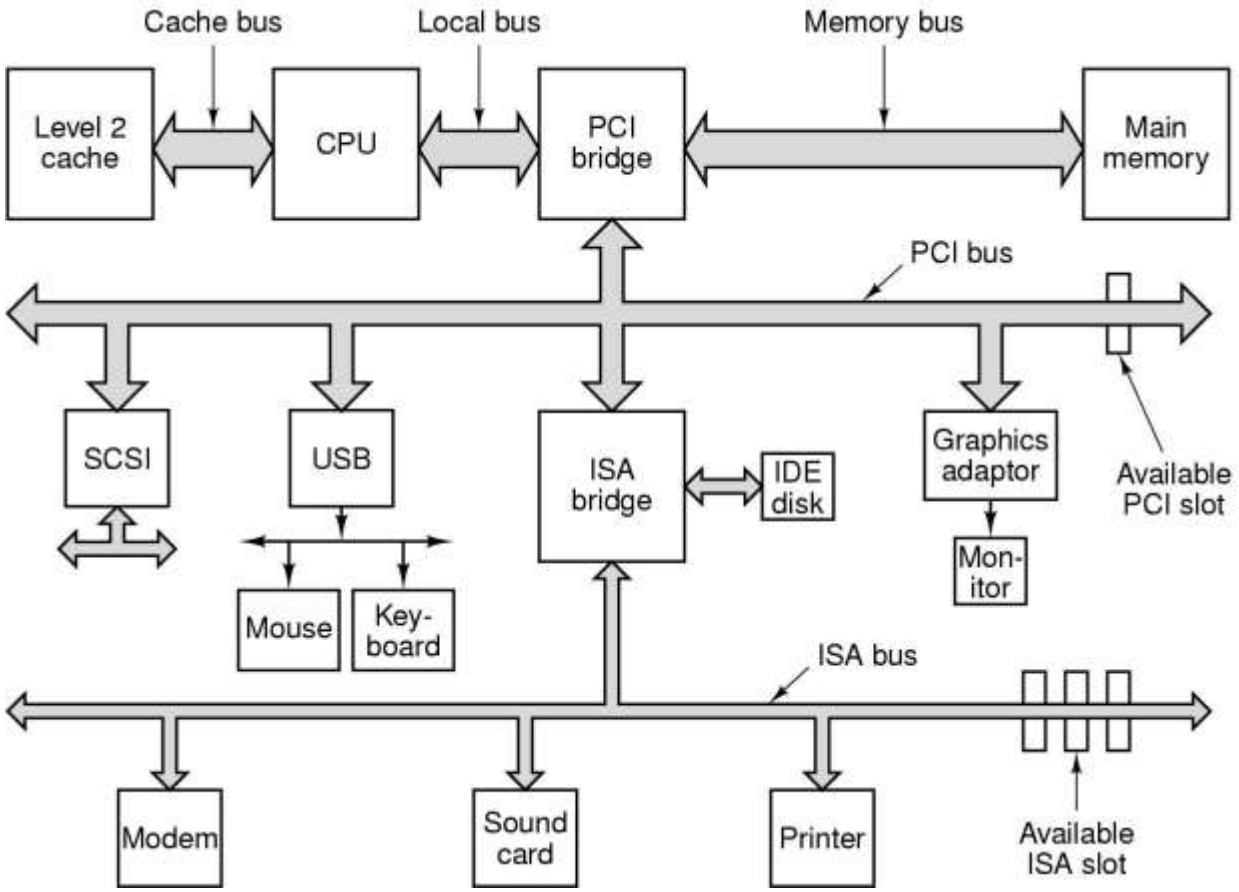
دستگاه‌های ورودی و خروجی

I/O Devices



A simplified view of a computer system

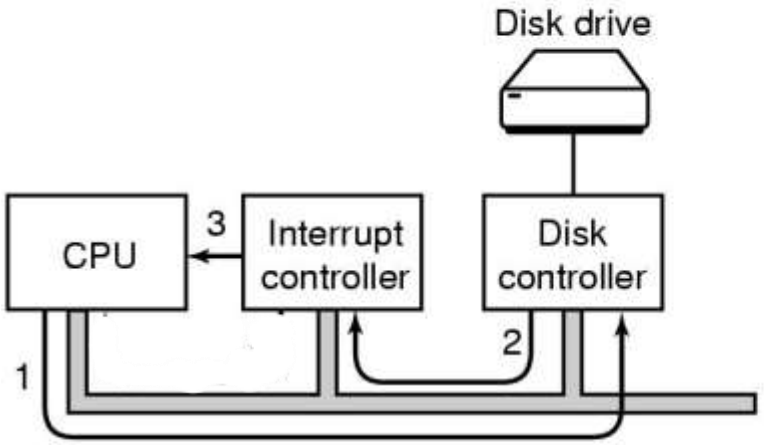
Structure of a Pentium System

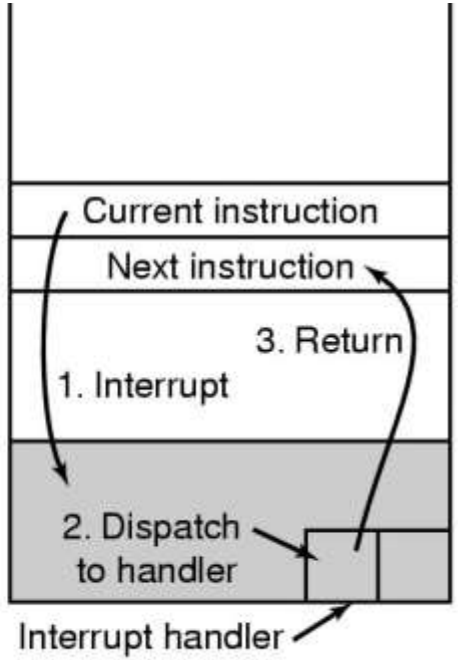


Program-Device Interaction

- Devices vs device controllers vs device drivers
  - *Device drivers are part of the OS (ie. software)*
  - *Programs call the OS which calls the device driver*
  - *Device drivers interact with device controllers using special IO instructions or by reading/writing controller registers that appear as memory locations*
  - *Device controllers are hardware that communicate with device drivers via interrupts*

Device to Program Interaction





Types of Interrupt/Trap

- Timer interrupts
  - *Allows OS to regain control of the CPU*
  - *One way to keep track of time*
- I/O interrupts
  - *Keyboard, mouse, disks, network, etc…*
- Program generated (traps & faults)
  - *Address translation faults (page fault, TLB miss)*
  - *Programming errors: seg. faults, divide by zero, etc.*
  - *System calls like read(), write(), gettimeofday()*

SYSTEM CALLS

System calls

System calls are the mechanism by which programs invoke the OS

Implemented via a TRAP instruction

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Types of System Calls

- Process control
  - *create process, terminate process*
  - *end, abort*
  - *load, execute*
  - *get process attributes, set process attributes*
  - *wait for time*
  - *wait event, signal event*
  - *allocate and free memory*
  - *Dump memory if error*
  - ***Debugger** for determining **bugs, single step** execution*
  - ***Locks** for managing access to shared data between processes*

Types of System Calls (Cont.)

- File management
  - *create file, delete file*
  - *open, close file*
  - *read, write, reposition*
  - *get and set file attributes*
- Device management
  - *request device, release device*
  - *read, write, reposition*
  - *get device attributes, set device attributes*
  - *logically attach or detach devices*

Types of System Calls (Cont.)

- Information maintenance
  - *get time or date, set time or date*
  - *get system data, set system data*
  - *get and set process, file, or device attributes*
- Communications
  - *create, delete communication connection*
  - *send, receive messages if **message passing model** to **host name** or **process name***
    - From **client** to **server**
  - ***Shared-memory model** create and gain access to memory regions*
  - *transfer status information*
  - *attach and detach remote devices*

Types of System Calls (Cont.)

- Protection
  - *Control access to resources*
  - *Get and set permissions*
  - *Allow and deny user access*

System Call Implementation

User-level code

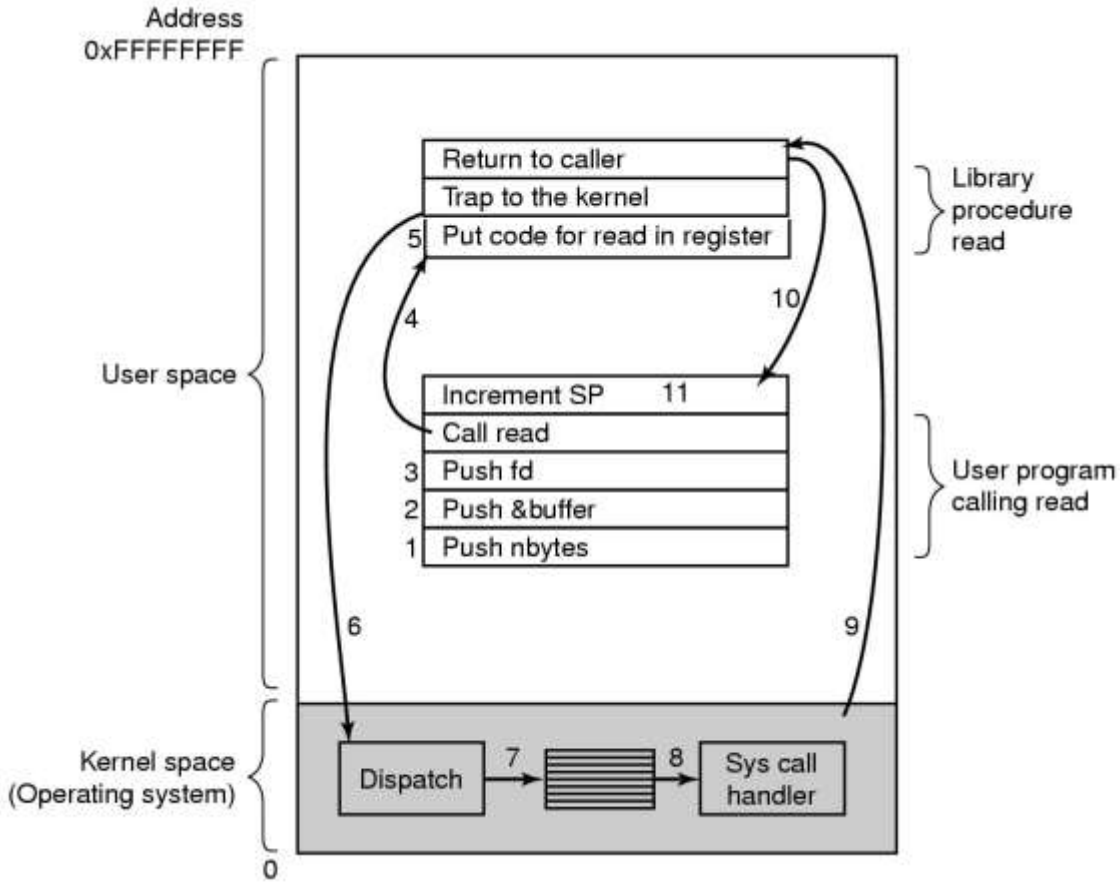
```
Process usercode
  { ...
   read (file, buffer, n);
  } ...

Procedure read(file, buff,
  n)
  { ...
   read(file, buff, n)
  } ...
```

Library code

```
_read:
  LOAD r1, @SP+2
  LOAD r2, @SP+4
  LOAD r3, @SP+6
  TRAP Read_Call
```

Read(fd,nbytes,buffer) System Call



Linkers and Loaders

- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
  - *Also brings in libraries*
- Program resides on secondary storage as binary executable
- Must be brought into memory by **loader** to be executed
  - ***Relocation*** *assigns final addresses to program parts and adjusts code and data in program to match those addresses*
- Modern general purpose systems don't link libraries into executables
  - *Rather,* ***dynamically linked libraries*** *(in Windows,* ***DLLs****) are loaded as needed, shared by all that use the same version of that same library (loaded once)*
- Object, executable files have standard formats, so operating system knows how to load and start them

The Role of the Linker and Loader