Sistemi Operativi

Corso di Laurea in Informatica

a.a. 2020-2021

SAPIENZA
UNIVERSITÀ DI ROMA

Gabriele Tolomei

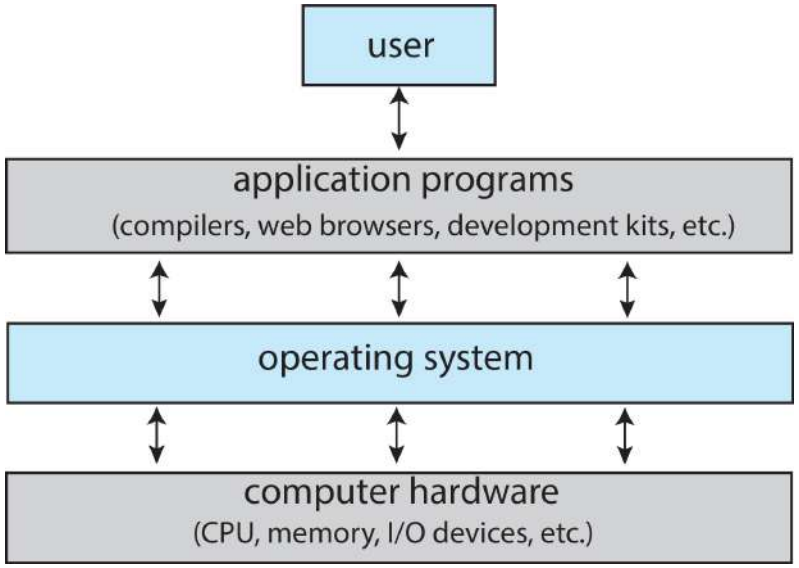
Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

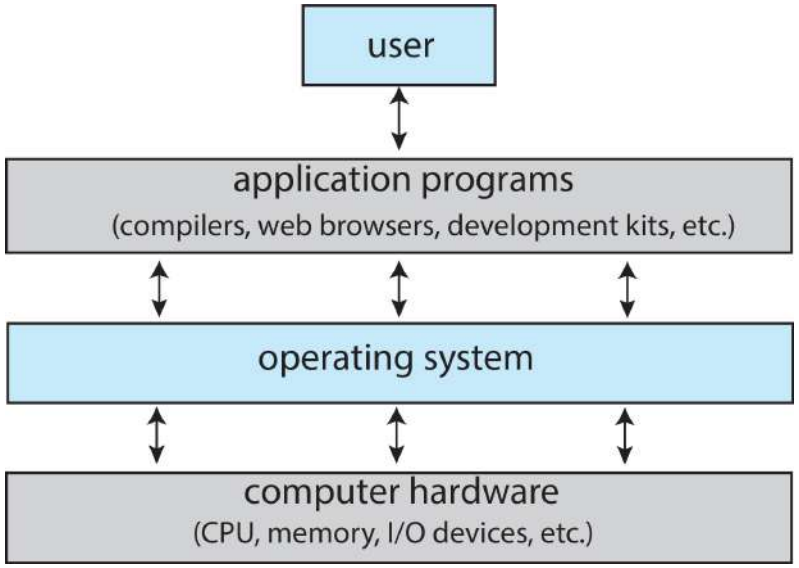
Recap from Last Lecture

- Operating System is a complex system which plays several roles:
  - resource manager
  - virtual machine
  - HW/SW interface



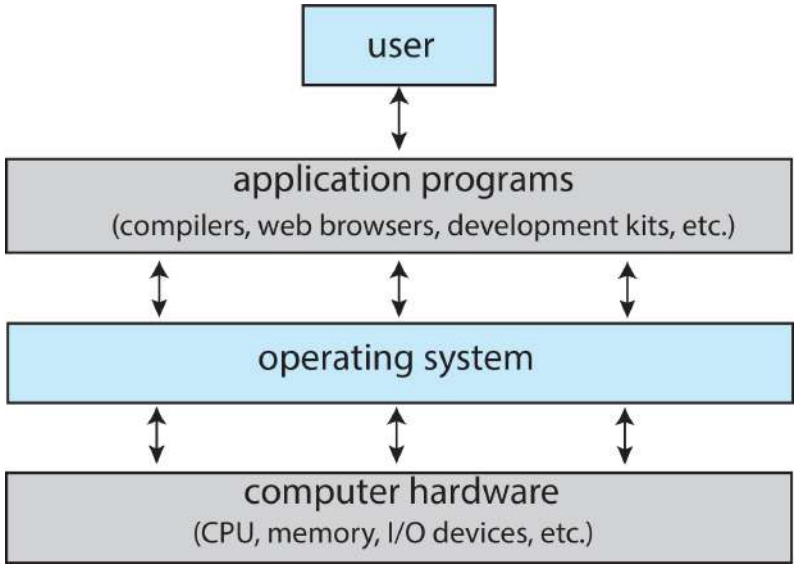
Recap from Last Lecture

- Operating System is a complex system which plays several roles:
  - resource manager
  - virtual machine
  - HW/SW interface
- Exposes services to users/applications (SW) leveraging the physical machine (HW)



Recap from Last Lecture

- Operating System is a complex system which plays several roles:
  - resource manager
  - virtual machine
  - HW/SW interface
- Exposes services to users/applications (SW) leveraging the physical machine (HW)
- Changes in HW may affect OS design



OS and Computer Architecture

- Computer architecture review

OS and Computer Architecture

- Computer architecture review
- Basic OS functionalities (enabled by architectural features)

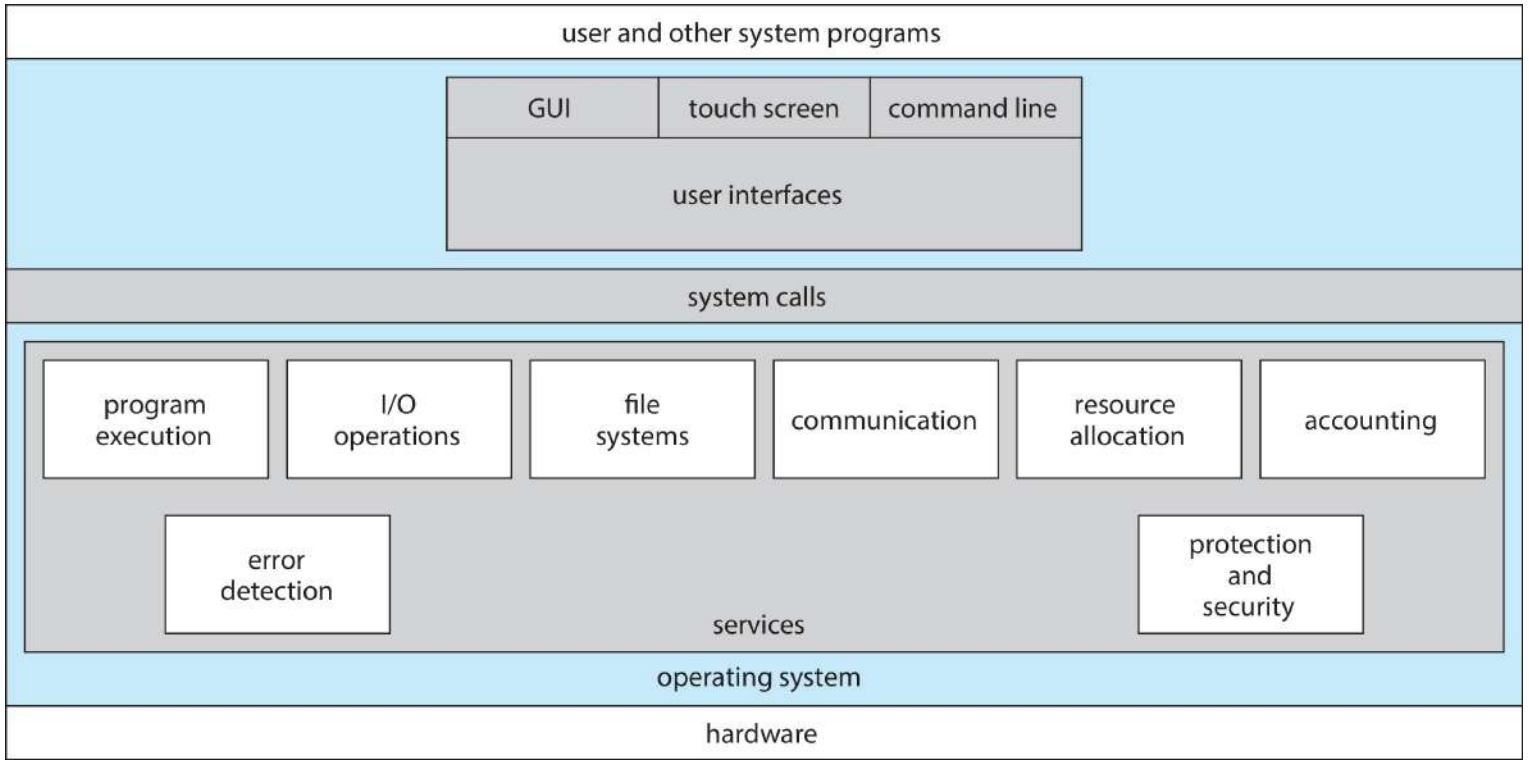
OS and Computer Architecture

- Computer architecture review
- Basic OS functionalities (enabled by architectural features)
- What the OS can do is partially dictated by the underlying architecture

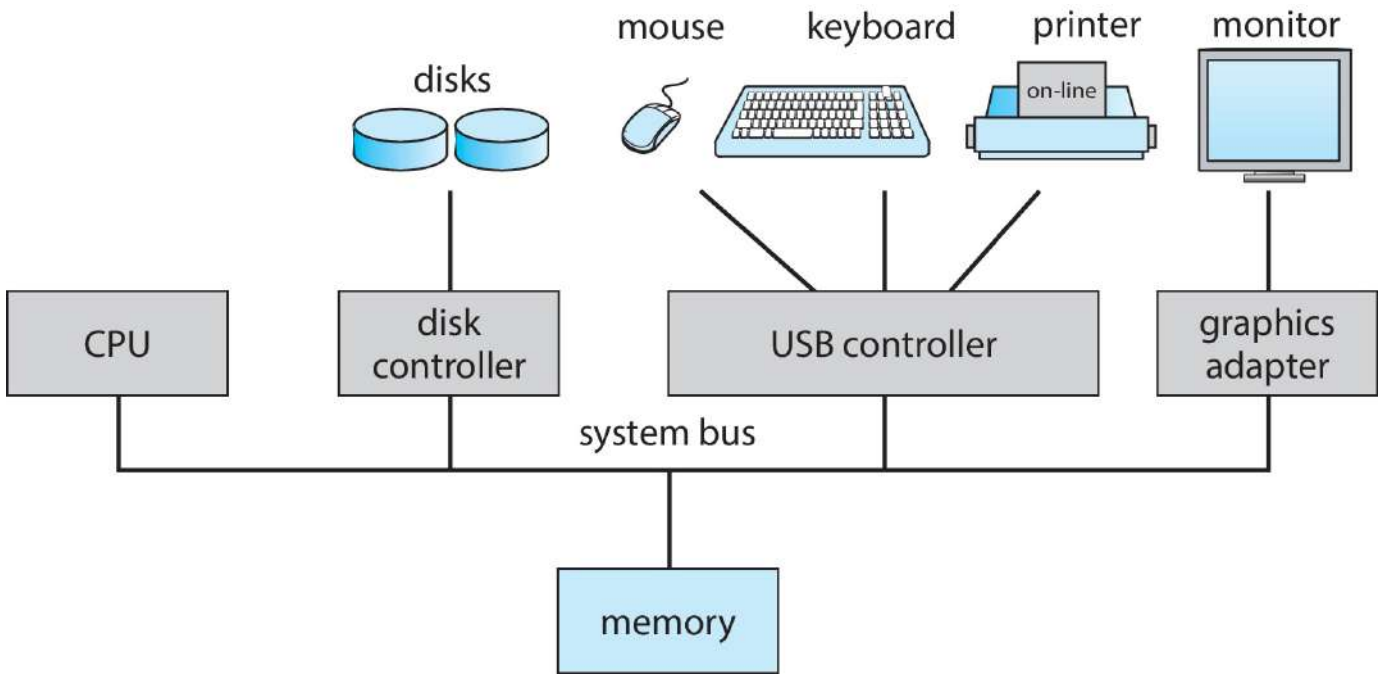
OS and Computer Architecture

- Computer architecture review
- Basic OS functionalities (enabled by architectural features)
- What the OS can do is partially dictated by the underlying architecture
- Architectural support may significantly simplify or complicate the OS design

Modern OS Functionalities



Generic High-Level Computer Architecture



Generic High-Level Computer Architecture

- CPU → the processor that performs the actual computation
  - Multiple cores are now common in modern architectures

Generic High-Level Computer Architecture

- CPU → the processor that performs the actual computation
  - Multiple cores are now common in modern architectures
- Main Memory → stores data and instructions used by the CPU

Generic High-Level Computer Architecture

- CPU → the processor that performs the actual computation
  - Multiple cores are now common in modern architectures
- Main Memory → stores data and instructions used by the CPU
- I/O devices → terminal, keyboard, disks, etc.
  - associated with specific device controllers

Generic High-Level Computer Architecture

- CPU → the processor that performs the actual computation
  - Multiple cores are now common in modern architectures
- Main Memory → stores data and instructions used by the CPU
- I/O devices → terminal, keyboard, disks, etc.
  - associated with specific device controllers
- System Bus → communication medium between CPU, memory, and peripherals

Computer Architecture Model

- Conceptually, the same architectural model for many computing devices:
  - PCs/laptops
  - High-end servers
  - Smartphones/Tablets
  - etc.

Computer Architecture Model

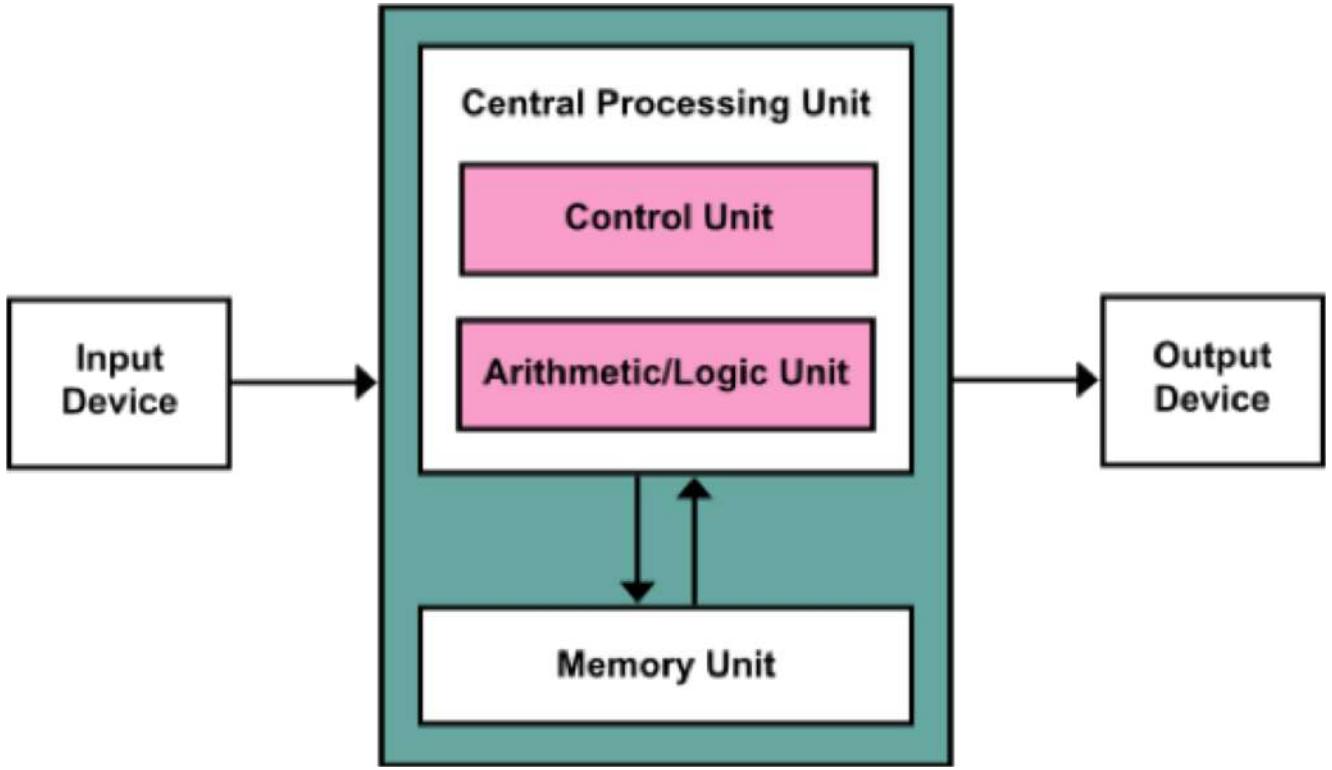
- Conceptually, the same architectural model for many computing devices:
  - PCs/laptops
  - High-end servers
  - Smartphones/Tablets
  - etc.
- Based on stored-program concept (as opposed to fixed-program)

Computer Architecture Model

- Conceptually, the same architectural model for many computing devices:
  - PCs/laptops
  - High-end servers
  - Smartphones/Tablets
  - etc.
- Based on stored-program concept (as opposed to fixed-program)

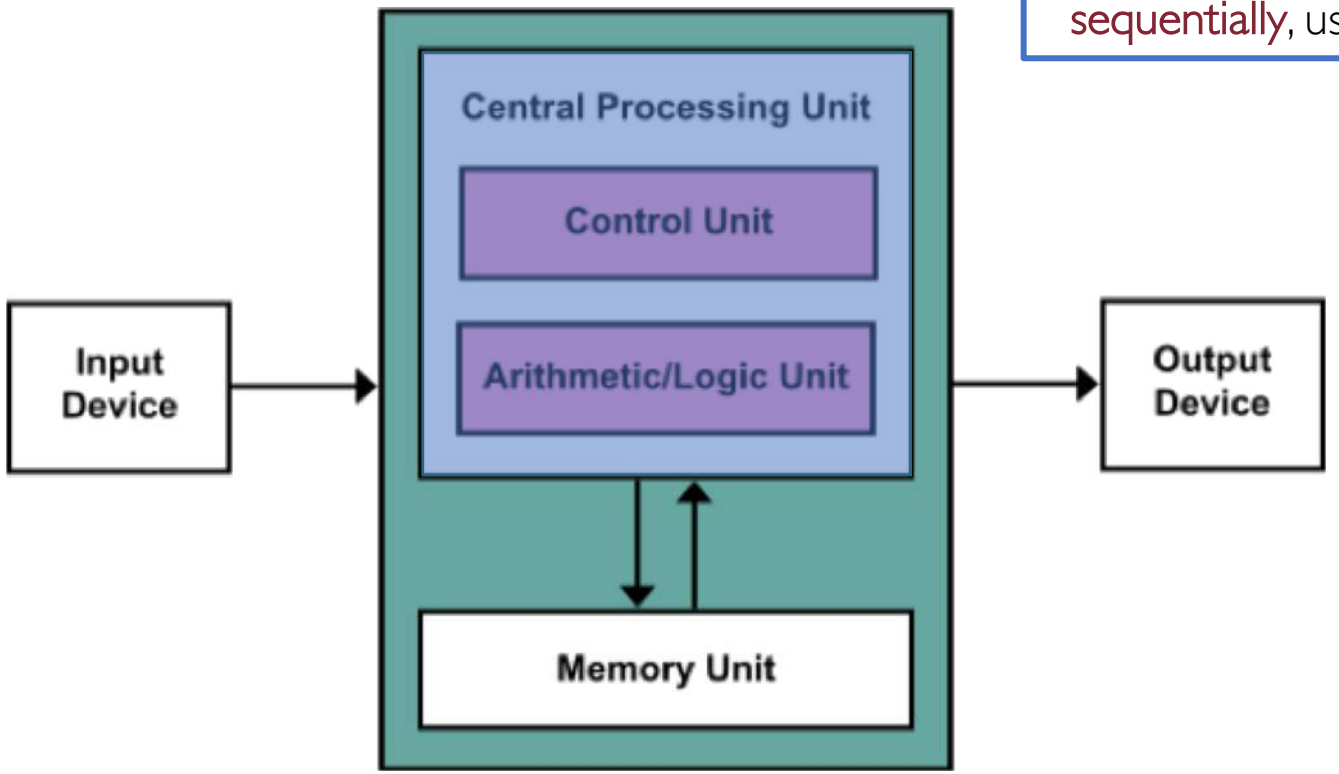
John von Neumann

von Neumann Architecture

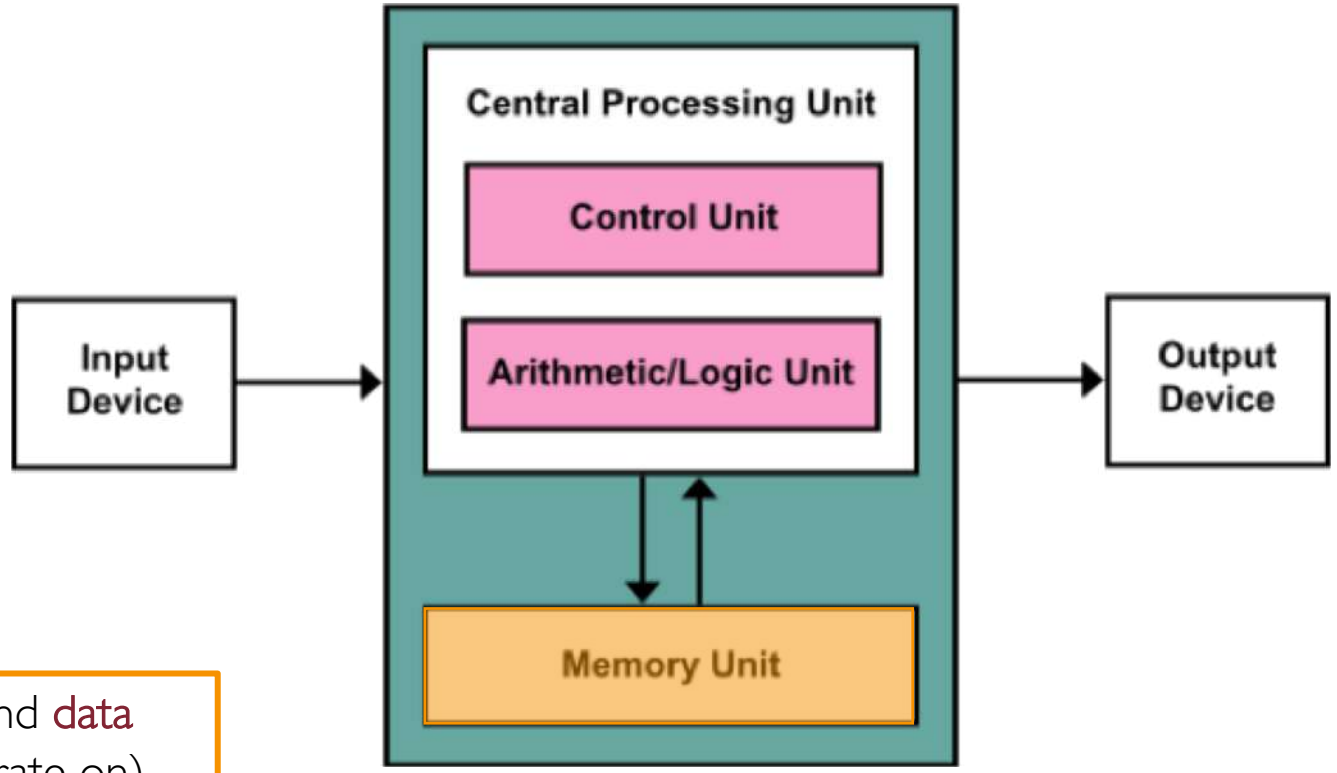


von Neumann Architecture

Executes instructions (i.e., basic arithmetical/logical computations) sequentially, using internal registers



von Neumann Architecture



Contains instructions and data
(which instructions operate on)

Central Processing Unit

(CPU)

Instruction Cycle: Fetch-Decode-Execute

- The CPU performs the following 3 steps cyclically:

Instruction Cycle: Fetch-Decode-Execute

- The CPU performs the following 3 steps cyclically:
  - Fetch: retrieves an instruction from a specific memory address whose value is contained in a special CPU register, called **Program Counter** (PC)

Instruction Cycle: Fetch-Decode-Execute

- The CPU performs the following 3 steps cyclically:
  - Fetch: retrieves an instruction from a specific memory address whose value is contained in a special CPU register, called **Program Counter** (PC)
  - Decode: interprets the fetched instruction

Instruction Cycle: Fetch-Decode-Execute

- The CPU performs the following 3 steps cyclically:
  - Fetch: retrieves an instruction from a specific memory address whose value is contained in a special CPU register, called **Program Counter** (PC)
  - Decode: interprets the fetched instruction
  - Execute: runs the actual decoded instruction

Machine Language

- Defines the set of elementary instructions the CPU is able to execute directly (i.e., on the HW)

Machine Language

- Defines the set of elementary instructions the CPU is able to execute directly (i.e., on the HW)
- The machine language is represented by binary numeral system

Machine Language

- Defines the set of elementary instructions the CPU is able to execute directly (i.e., on the HW)
- The machine language is represented by binary numeral system
- Each instruction is encoded as a sequence of bits
  - A single bit is the smallest unit of (digital) information
  - It takes on two possible values: 0 or 1

Machine Language

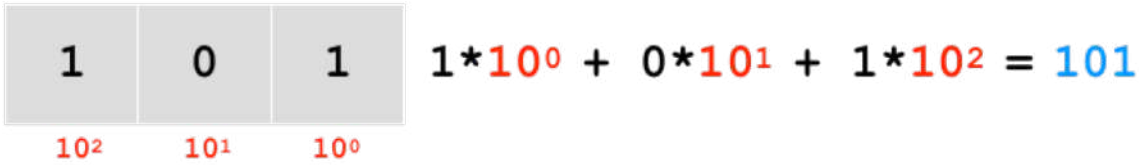
- Defines the set of elementary instructions the CPU is able to execute directly (i.e., on the HW)
- The machine language is represented by binary numeral system
- Each instruction is encoded as a sequence of bits
  - A single bit is the smallest unit of (digital) information
  - It takes on two possible values: 0 or 1
- A word is the unit of data the CPU can directly operate on
  - today ranging from 32 to 64 bits

Binary vs. Decimal Numeral System

- In natural language we usually implicitly refer to the decimal numeral system (base-10)

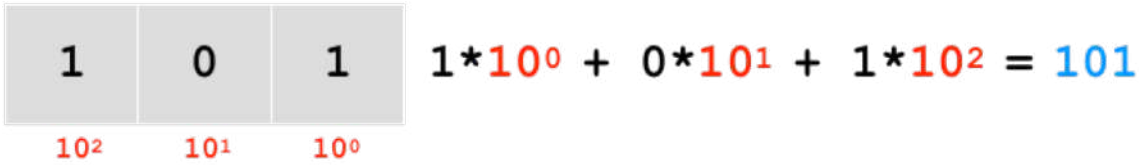
Binary vs. Decimal Numeral System

- In natural language we usually implicitly refer to the decimal numeral system (base-10)
- In base-10 system, each digit can only take one out of 10 possible values: 0, 1, …, 9

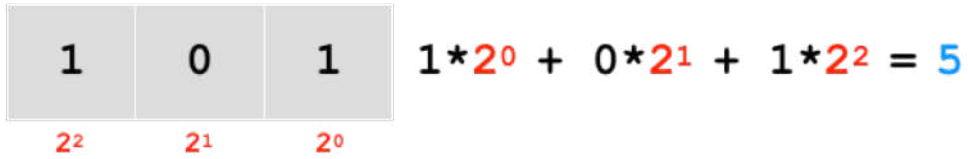


Binary vs. Decimal Numeral System

- In natural language we usually implicitly refer to the decimal numeral system (base-10)
- In base-10 system, each digit can only take one out of 10 possible values: 0, 1, …, 9



- In binary system (base-2), each digit is a bit



A Side Note on Units

Prefixes for multiples of bits (bit) or bytes (B)

Decimal				Binary				
Value		SI		Value		IEC		JEDEC
1000	10^3	k	kilo	1024	2^{10}	Ki	kibi	K kilo
1000^2	10^6	M	mega	1024^2	2^{20}	Mi	mebi	M mega
1000^3	10^9	G	giga	1024^3	2^{30}	Gi	gibi	G giga
1000^4	10^{12}	T	tera	1024^4	2^{40}	Ti	tebi	–
1000^5	10^{15}	P	peta	1024^5	2^{50}	Pi	pebi	–
1000^6	10^{18}	E	exa	1024^6	2^{60}	Ei	exbi	–
1000^7	10^{21}	Z	zetta	1024^7	2^{70}	Zi	zebi	–
1000^8	10^{24}	Y	yotta	1024^8	2^{80}	Yi	yobi	–

Instruction Set (Architecture)

- The collection of instructions defined by the machine language

Instruction Set (Architecture)

- The collection of instructions defined by the machine language
- Machine language instructions are made of 2 parts:
  - An operator (op code)
  - Zero or more operands representing either CPU internal registers or memory addresses

Instruction Set (Architecture)

- The collection of instructions defined by the machine language
- Machine language instructions are made of 2 parts:
  - An operator (op code)
  - Zero or more operands representing either CPU internal registers or memory addresses
- An abstraction of the underlying physical (hardware) architecture (e.g., x86, ARM, SPARC, MIPS, etc.)

Instruction Set (Architecture)

- The collection of instructions defined by the machine language
- Machine language instructions are made of 2 parts:
  - An operator (op code)
  - Zero or more operands representing either CPU internal registers or memory addresses
- An abstraction of the underlying physical (hardware) architecture (e.g., x86, ARM, SPARC, MIPS, etc.)
- Each realization of the same instruction set is an implementation of a physical architecture (e.g., x86 → Intel, AMD, Cyrix, etc.)

CPU Registers

- On-chip storage whose size typically coincides with the CPU word size

CPU Registers

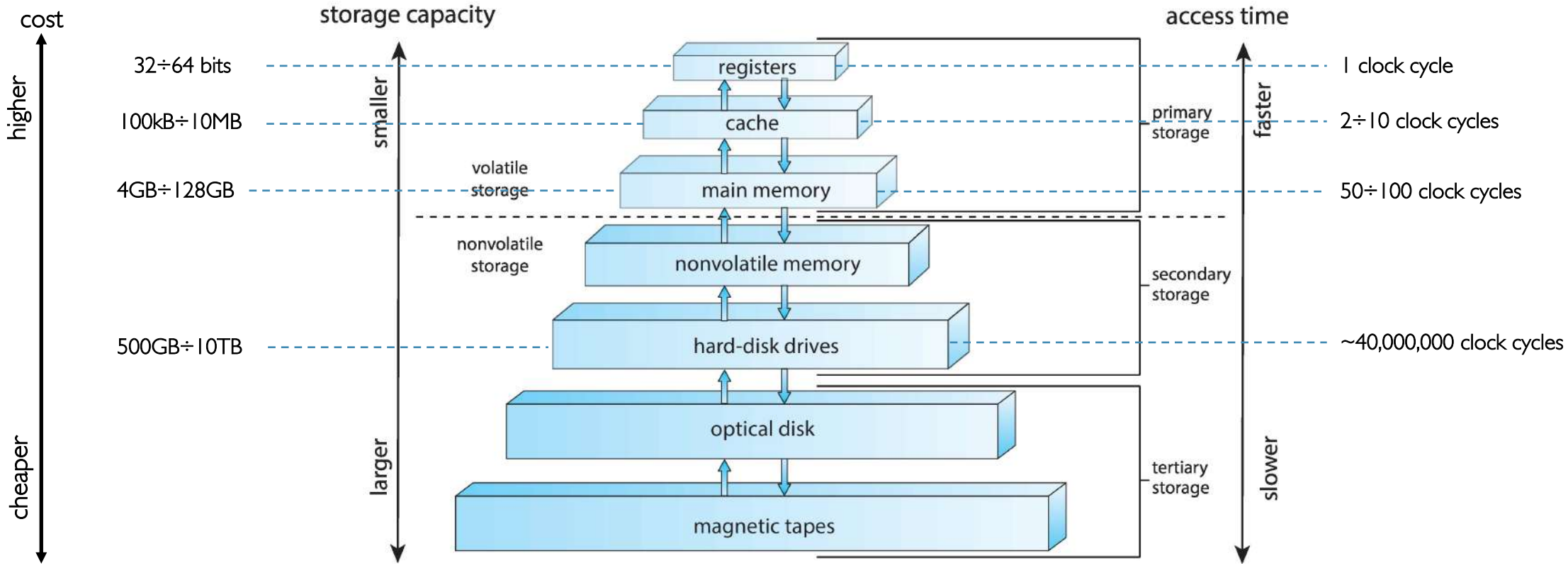
- On-chip storage whose size typically coincides with the CPU word size
- General-purpose (x86):
  - `eax`, `ebx`, `ecx`, etc.

CPU Registers

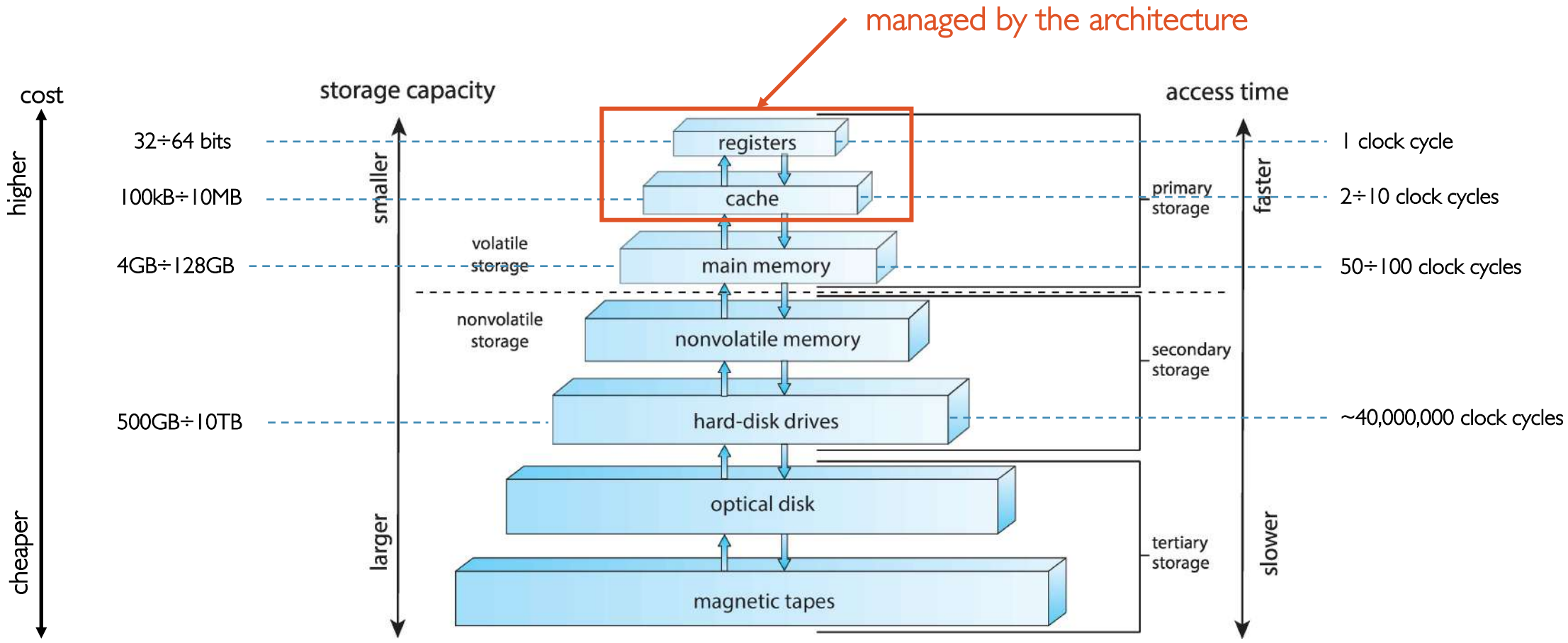
- On-chip storage whose size typically coincides with the CPU word size
- General-purpose (x86):
  - `eax`, `ebx`, `ecx`, etc.
- Special-purpose (x86):
  - `esp` → Stack pointer for top address of the stack
  - `ebp` → Stack base pointer for the address of the current stack frame
  - `eip` → Instruction pointer, holds the program counter (i.e., the address of next instruction)

Memory

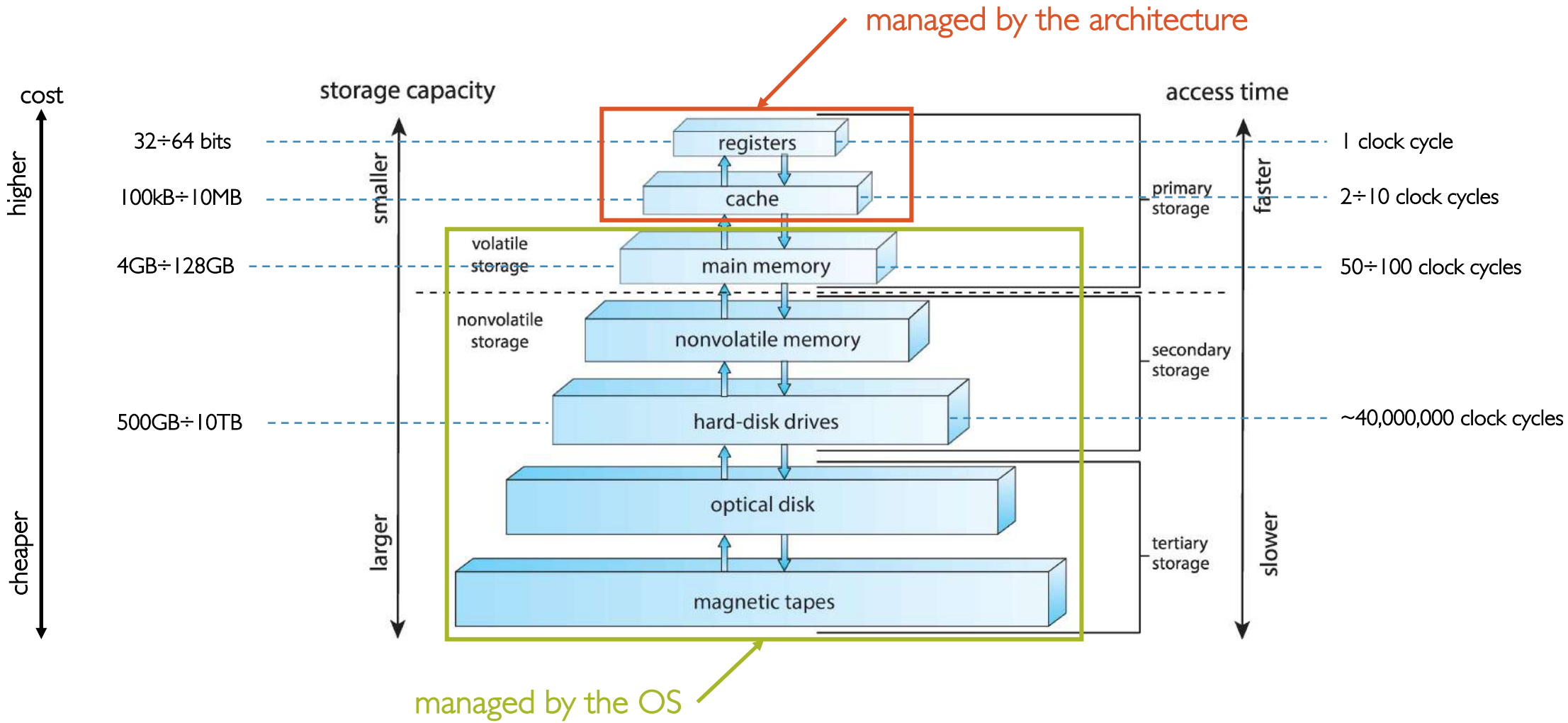
Memory Hierarchy



Memory Hierarchy



Memory Hierarchy



Main Memory Representation

- Essentially, a sequence of cells

Main Memory Representation

- Essentially, a sequence of cells
- Each cell is logically organized in groups of 8 bits (1 Byte) or multiple of it (e.g., 32 bits = 4 Bytes)

Main Memory Representation

- Essentially, a sequence of cells
- Each cell is logically organized in groups of 8 bits (1 Byte) or multiple of it (e.g., 32 bits = 4 Bytes)
- Each cell has its own address

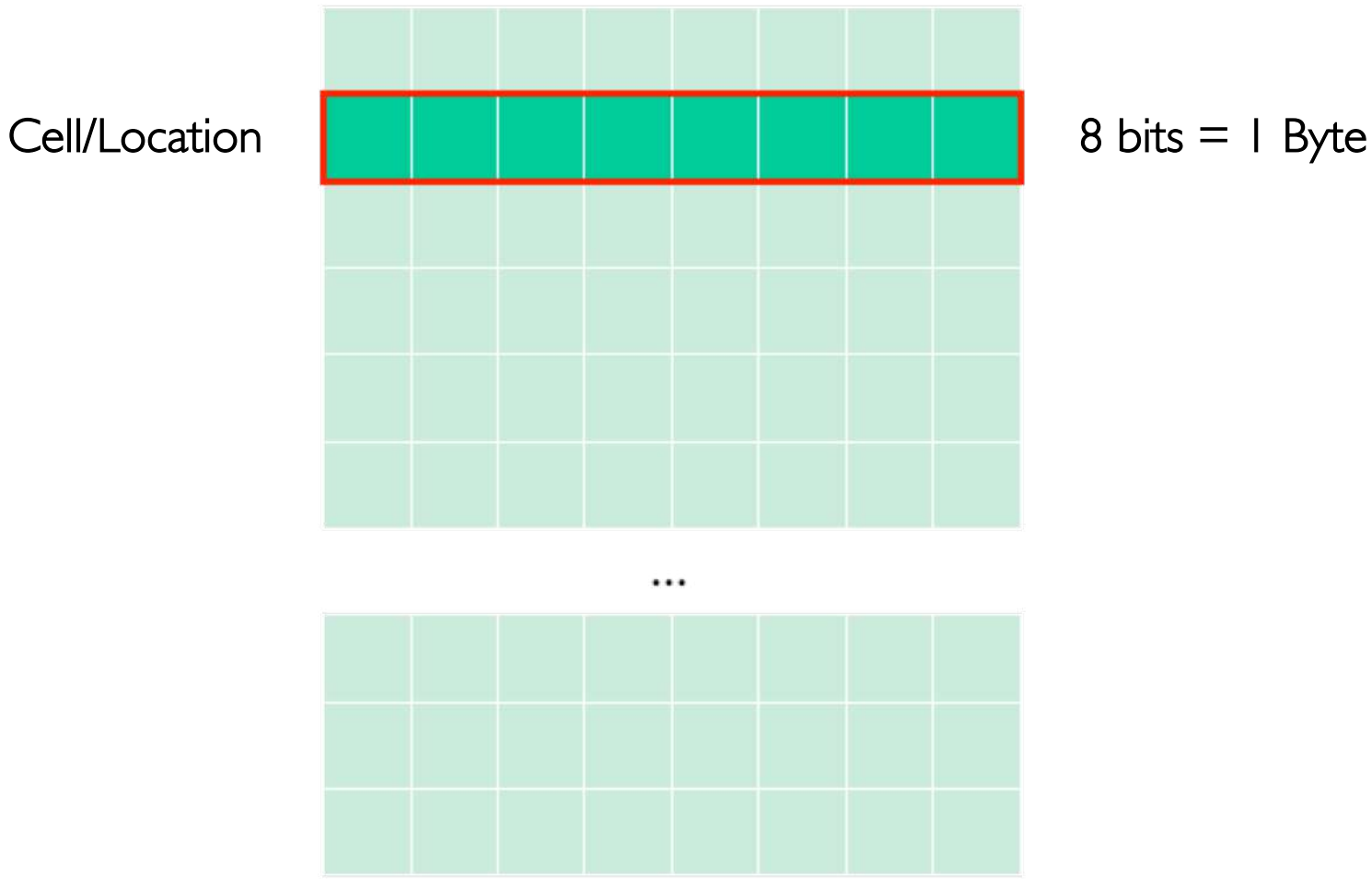
Main Memory Representation

- Essentially, a sequence of cells
- Each cell is logically organized in groups of 8 bits (1 Byte) or multiple of it (e.g., 32 bits = 4 Bytes)
- Each cell has its own address
- CPU (and I/O devices) read from/write to main memory referencing memory location addresses

Main Memory Representation

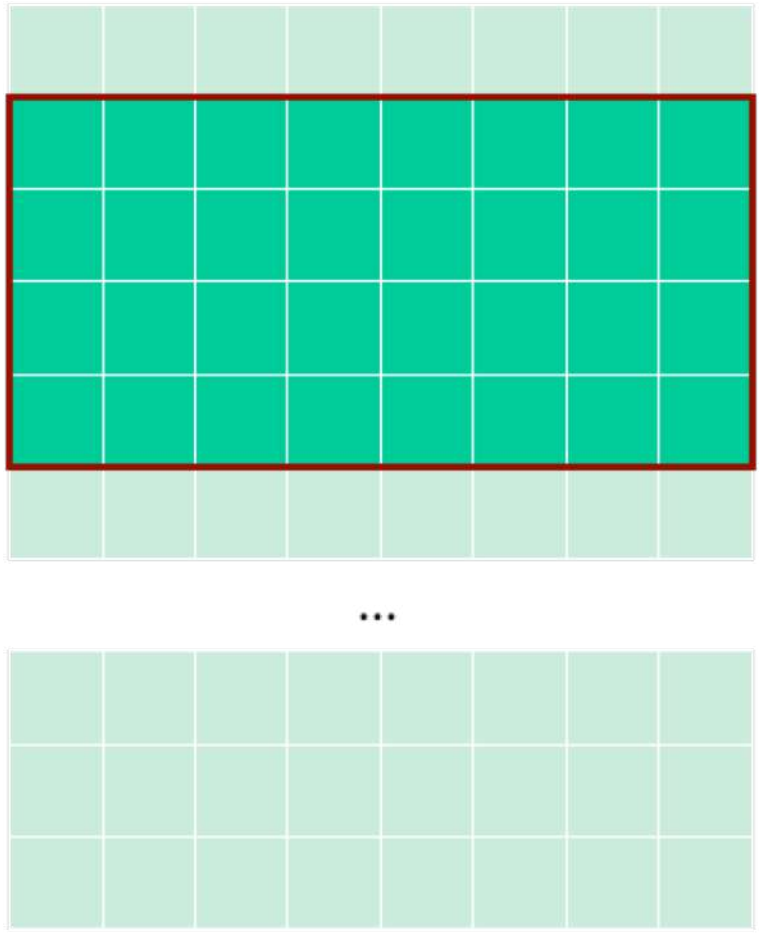
- Essentially, a sequence of cells
- Each cell is logically organized in groups of 8 bits (1 Byte) or multiple of it (e.g., 32 bits = 4 Bytes)
- Each cell has its own address
- CPU (and I/O devices) read from/write to main memory referencing memory location addresses
- The smallest addressable unit is usually 1 Byte

Memory Cell (1)



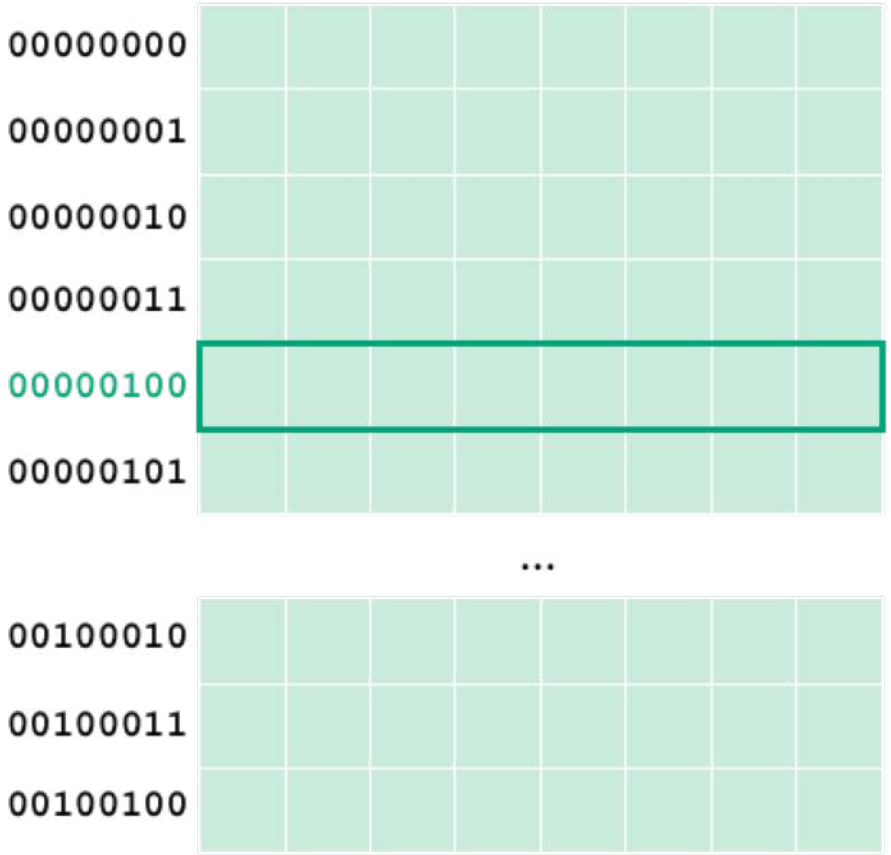
Memory Cell (2)

Cell/Location



32 bits = 4 Bytes

Memory Address (Single Byte)



Computer Buses

System Bus

- Initially, a single bus to handle all the traffic

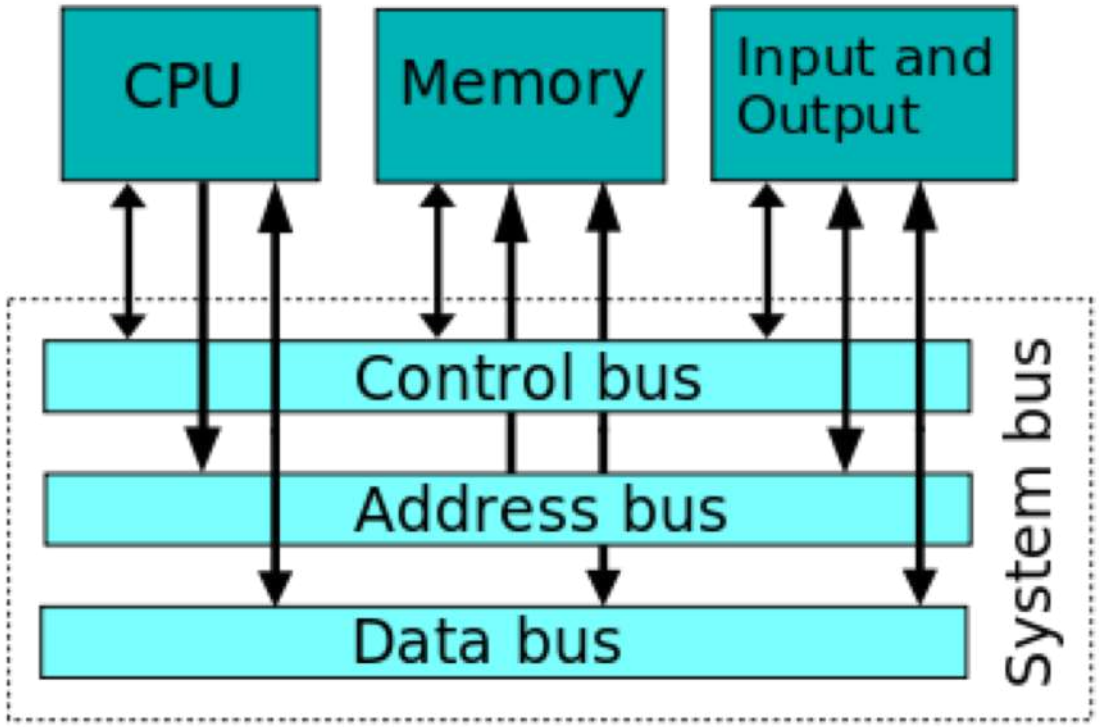
System Bus

- Initially, a single bus to handle all the traffic
- Combines the functions of:
  - Data bus → to actually carry information
  - Address bus → to determine where such information should be sent
  - Control bus → to indicate which operation should be performed

System Bus

- Initially, a single bus to handle all the traffic
- Combines the functions of:
  - Data bus → to actually carry information
  - Address bus → to determine where such information should be sent
  - Control bus → to indicate which operation should be performed
- More dedicated buses have been added to manage CPU-to-memory and I/O traffic
  - PCI, SATA, USB, etc.

System Bus


CPU
Memory
Input and Output
Control bus
Address bus
Data bus
System bus

I/O Devices

Components

- Each I/O device is made of 2 parts:

Components

- Each I/O device is made of 2 parts:
  - the physical device itself

Components

- Each I/O device is made of 2 parts:
  - the physical device itself
  - the device controller (chip or set of chips controlling a family of physical devices)

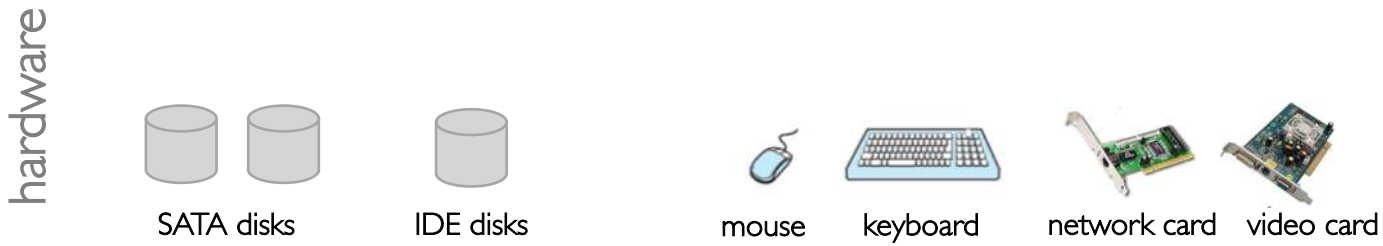
Components

- Each I/O device is made of 2 parts:
  - the physical device itself
  - the device controller (chip or set of chips controlling a family of physical devices)
- Can be categorized as:
  - storage, communications, user-interface, etc.

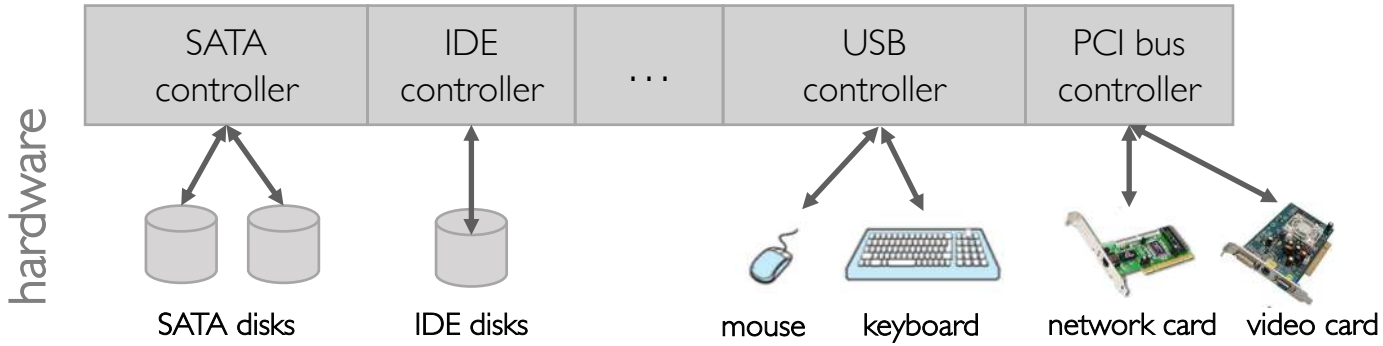
Components

- Each I/O device is made of 2 parts:
  - the physical device itself
  - the device controller (chip or set of chips controlling a family of physical devices)
- Can be categorized as:
  - storage, communications, user-interface, etc.
- OS talks to a device controller using a specific device driver

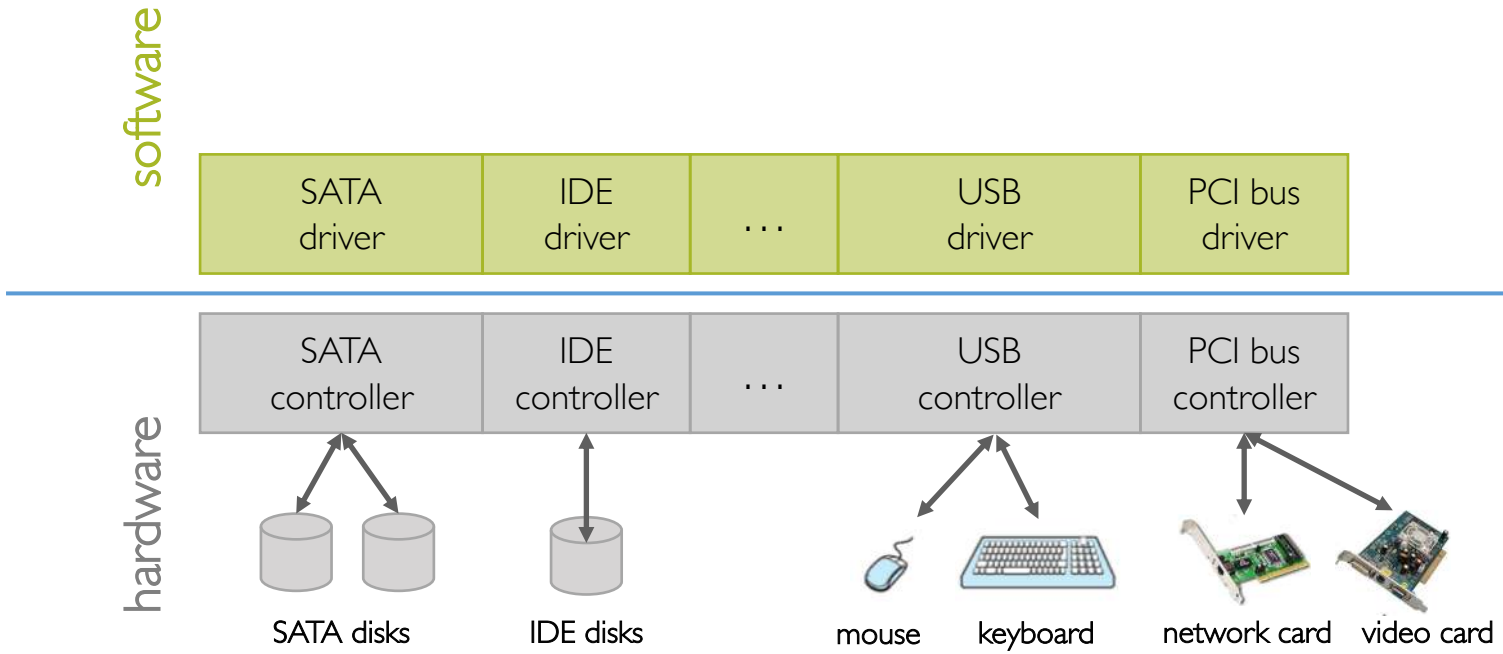
Device Drivers: OS Software Abstraction



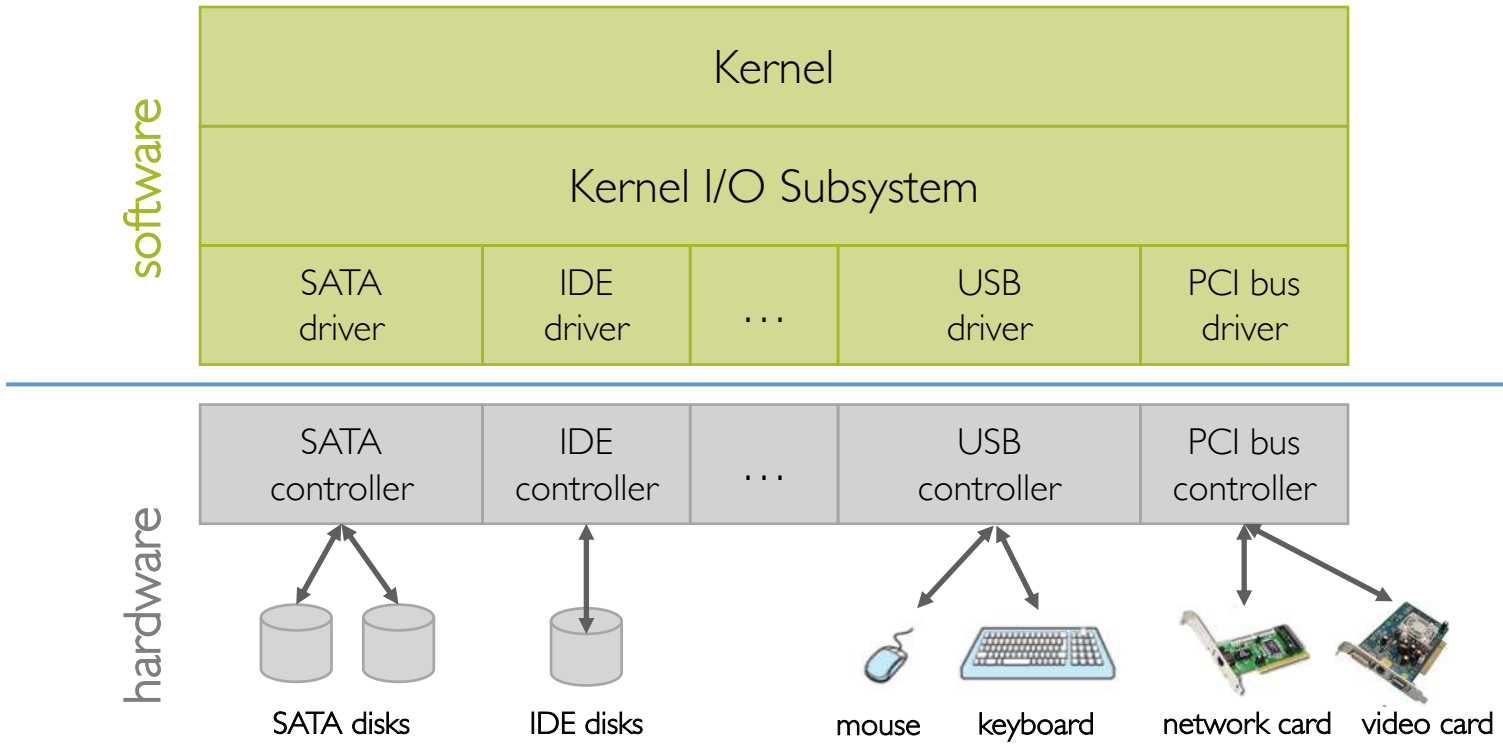
Device Drivers: OS Software Abstraction



Device Drivers: OS Software Abstraction



Device Drivers: OS Software Abstraction



Communicating with Device Controllers

- Every device controller has a number of dedicated registers to communicate with it:

Communicating with Device Controllers

- Every device controller has a number of dedicated registers to communicate with it:
  - Status registers → provide status information to the CPU about the I/O device (e.g., idle, ready for input, busy, error, transaction complete)

Communicating with Device Controllers

- Every device controller has a number of dedicated registers to communicate with it:
  - Status registers → provide status information to the CPU about the I/O device (e.g., idle, ready for input, busy, error, transaction complete)
  - Configuration/Control registers → used by the CPU to configure and control the device

Communicating with Device Controllers

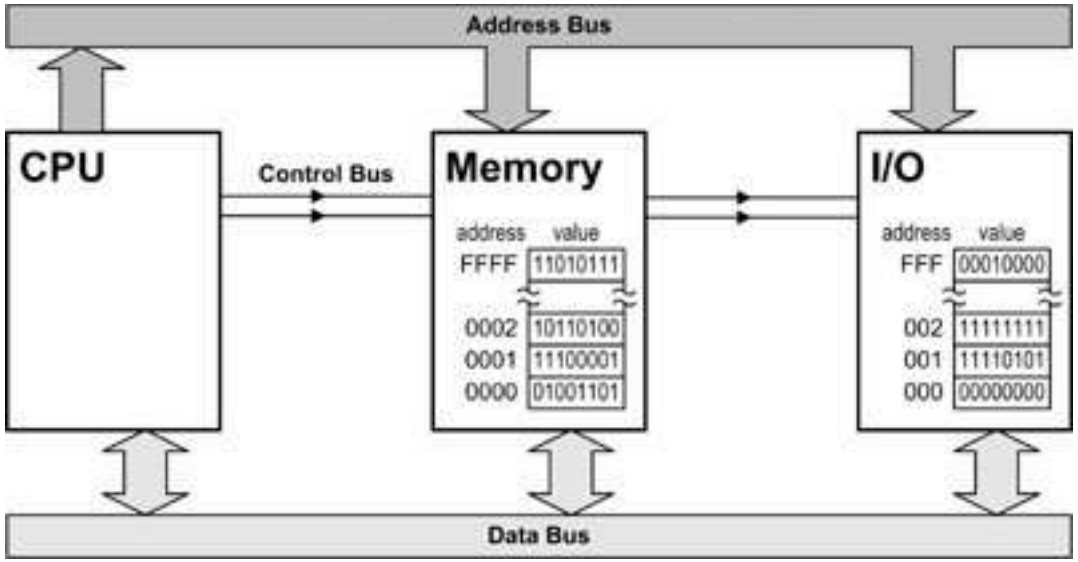
- Every device controller has a number of dedicated registers to communicate with it:
  - Status registers → provide status information to the CPU about the I/O device (e.g., idle, ready for input, busy, error, transaction complete)
  - Configuration/Control registers → used by the CPU to configure and control the device
  - Data registers → used to read data from or send data to the I/O device

Communicating with Device Controllers

- Every device controller has a number of dedicated registers to communicate with it:
  - Status registers → provide status information to the CPU about the I/O device (e.g., idle, ready for input, busy, error, transaction complete)
  - Configuration/Control registers → used by the CPU to configure and control the device
  - Data registers → used to read data from or send data to the I/O device

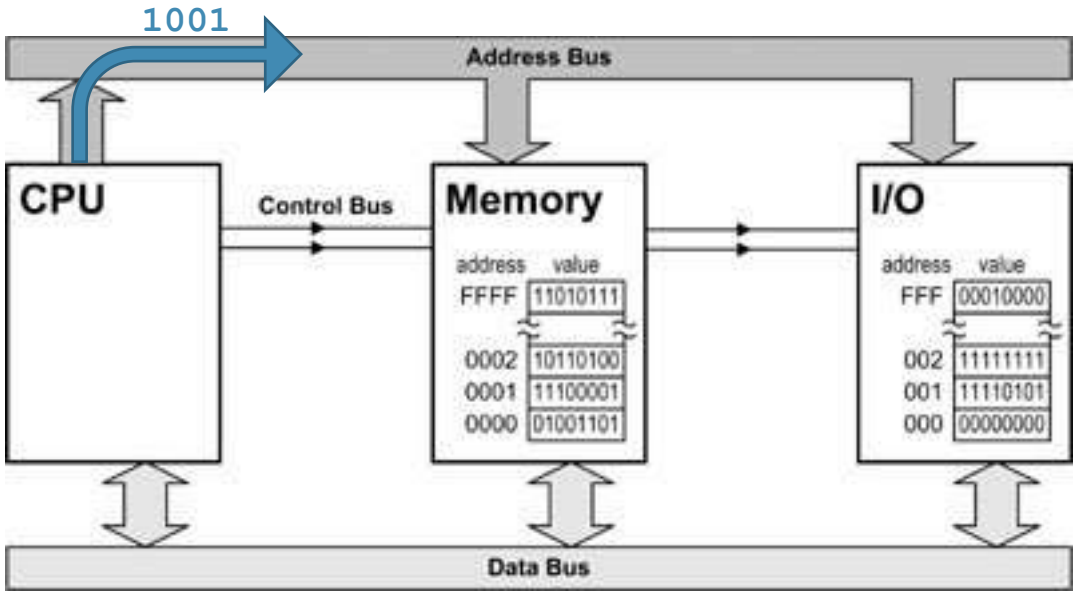
How does the CPU know how to address (registers of) I/O devices?

Addressing Using the System Bus



How does CPU reference Memory addresses?

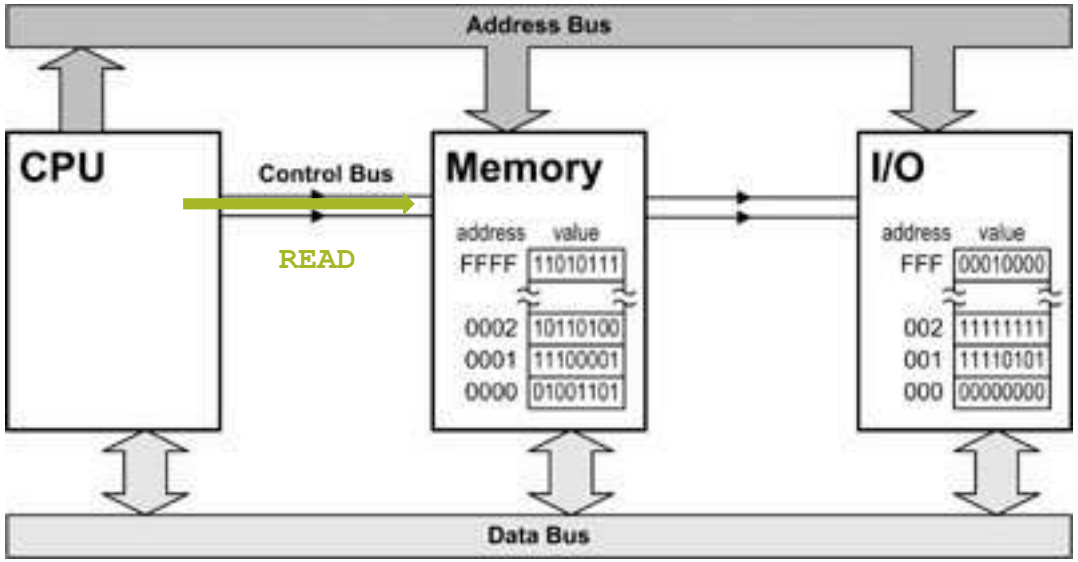
Addressing Using the System Bus



How does CPU reference Memory addresses?

It puts the address of a byte of memory on the address bus

Addressing Using the System Bus

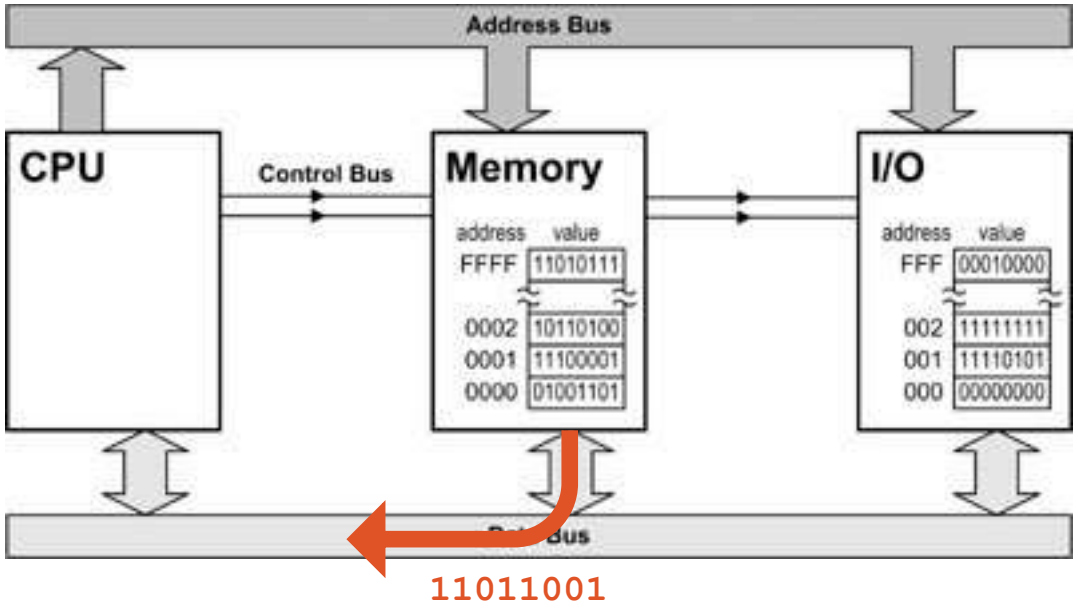


How does CPU reference Memory addresses?

It puts the address of a byte of memory on the address bus

It raises the **READ** signal on the control bus

Addressing Using the System Bus



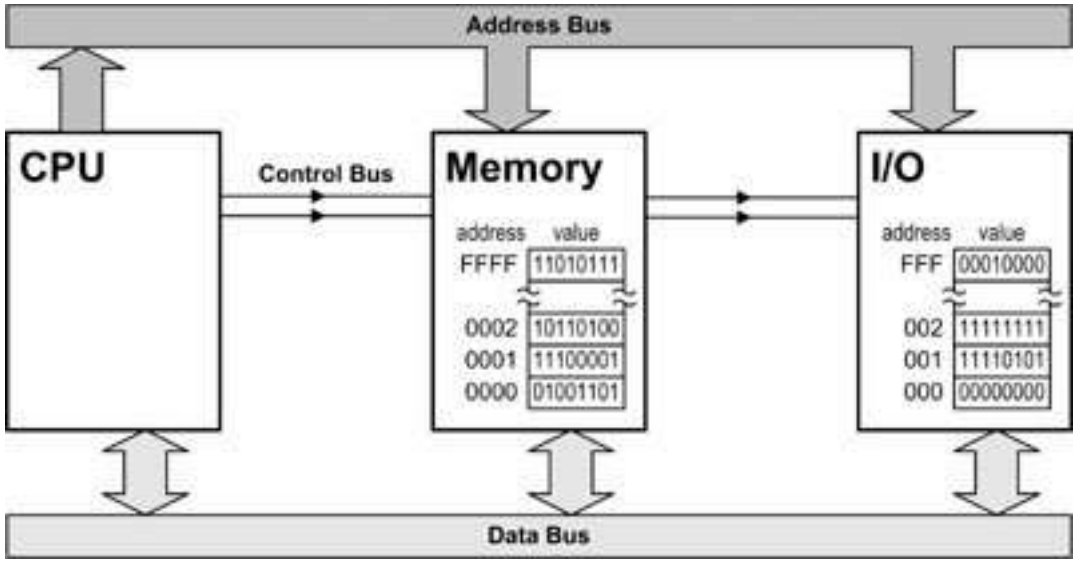
How does CPU reference Memory addresses?

It puts the address of a byte of memory on the address bus

It raises the **READ** signal on the control bus

Eventually, the RAM replies with the memory content on the data bus

Addressing Using the System Bus



How does CPU reference Memory addresses?

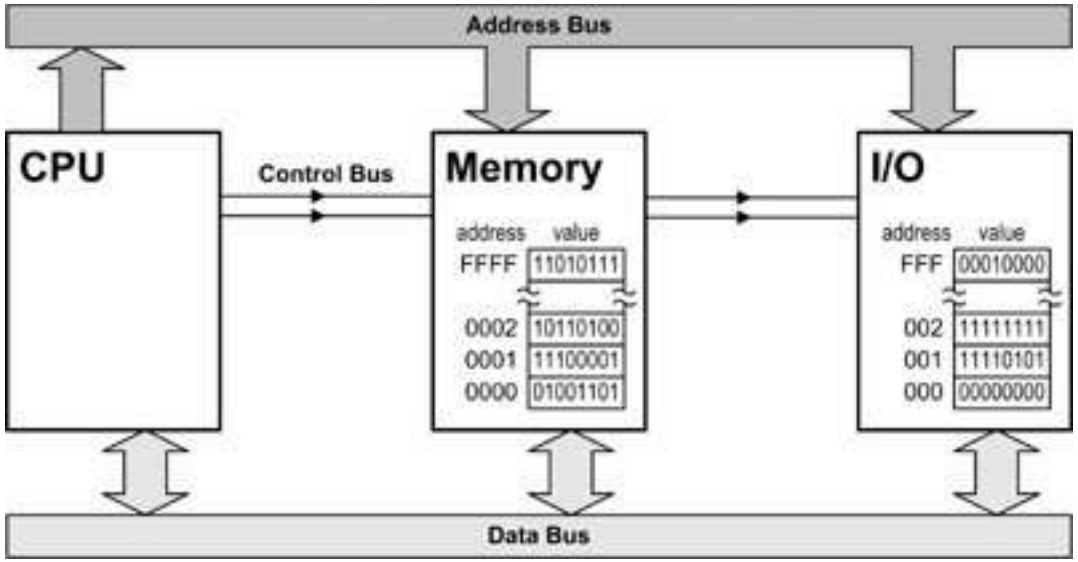
It puts the address of a byte of memory on the address bus

It raises the **READ** signal on the control bus

Eventually, the RAM replies with the memory content on the data bus

How about I/O devices? How to distinguish between Memory and I/O devices?

Addressing Using the System Bus



How does CPU reference Memory addresses?

It puts the address of a byte of memory on the address bus

It raises the **READ** signal on the control bus

Eventually, the RAM replies with the memory content on the data bus

How about I/O devices? How to distinguish between Memory and I/O devices?

The control bus has a special line called "**M/#IO**" which asserts whether the CPU wants to talk to memory or an I/O device

Port- vs. Memory-Mapped I/O

- CPU can talk to a device controller in 2 ways:

Port- vs. Memory-Mapped I/O

- CPU can talk to a device controller in 2 ways:
  - Port-mapped I/O → referencing controller's registers using a separate I/O address space

Port- vs. Memory-Mapped I/O

- CPU can talk to a device controller in 2 ways:
  - Port-mapped I/O → referencing controller's registers using a separate I/O address space
  - Memory-mapped I/O → mapping controller's registers to the same address space used for main memory

Port-Mapped I/O

- Each I/O device controller's register is mapped to a specific port (address)
- Requires special class of CPU instructions (e.g., **IN**/**OUT**)
  - The **IN** instruction reads from an I/O device, **OUT** writes
- When you use the **IN** or **OUT** instructions, the **M/#IO** is not asserted, so memory does not respond and the I/O chip does

Memory-Mapped I/O

- Memory-mapped I/O "wastes" some address space but doesn't need any special instruction
- To the CPU I/O device ports are just like normal memory addresses
- The CPU use `MOV`-like instructions to access I/O device registers
- In this way, the **`M/#IO`** is asserted indicating the address requested by the CPU refers to main memory

Port- vs. Memory-Mapped I/O

```
MOV DX,1234h
MOV AL,[DX]     ;reads memory address 1234h (memory address space)
IN AL,DX        ;reads I/O port 1234h (I/O address space)
```

Both put the value **`1234h`** on the CPU address bus, and both assert a **`READ`** operation on control bus

Port- vs. Memory-Mapped I/O

```
MOV DX,1234h
MOV AL,[DX]     ;reads memory address 1234h (memory address space)
IN AL,DX        ;reads I/O port 1234h (I/O address space)
```

The first one will assert **M/#IO** to indicate that the address belongs to memory address space

Port- vs. Memory-Mapped I/O

```
MOV DX,1234h
MOV AL,[DX]     ;reads memory address 1234h (memory address space)
IN AL,DX        ;reads I/O port 1234h (I/O address space)
```

The second one will **not** assert **M/#IO** to indicate that the address belongs to I/O address space

Performing I/O Tasks

- Polling
  - CPU periodically checks for the I/O task status

Performing I/O Tasks

- Polling
  - CPU periodically checks for the I/O task status
- Interrupt-driven
  - CPU receives an interrupt from the controller (device or DMA) once the I/O task is done (either successfully or abnormally)

Performing I/O Tasks

- Polling
  - CPU periodically checks for the I/O task status
- Interrupt-driven
  - CPU receives an interrupt from the controller (device or DMA) once the I/O task is done (either successfully or abnormally)
- Programmed I/O
  - CPU does the actual work of moving data

Performing I/O Tasks

- Polling
  - CPU periodically checks for the I/O task status
- Interrupt-driven
  - CPU receives an interrupt from the controller (device or DMA) once the I/O task is done (either successfully or abnormally)
- Programmed I/O
  - CPU does the actual work of moving data
- Direct Memory Access (DMA)
  - CPU delegates off the work to a dedicated DMA controller

Performing I/O Tasks

- Polling
  - CPU periodically checks for the I/O task status
- Interrupt-driven
  - CPU receives an interrupt from the controller (device or DMA) once the I/O task is done (either successfully or abnormally)

HOW?

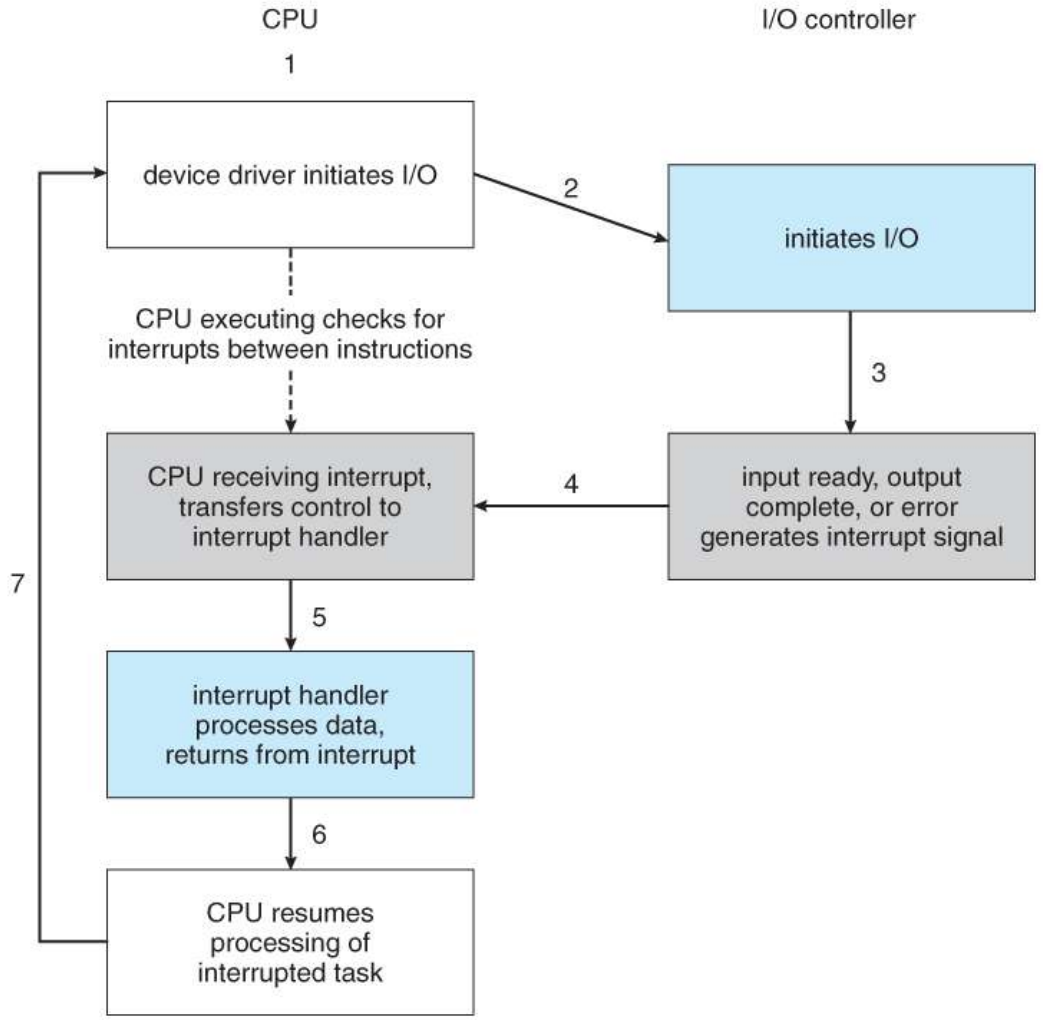
- Programmed I/O
  - CPU does the actual work of moving data
- Direct Memory Access (DMA)
  - CPU delegates off the work to a dedicated DMA controller

Performing I/O Tasks

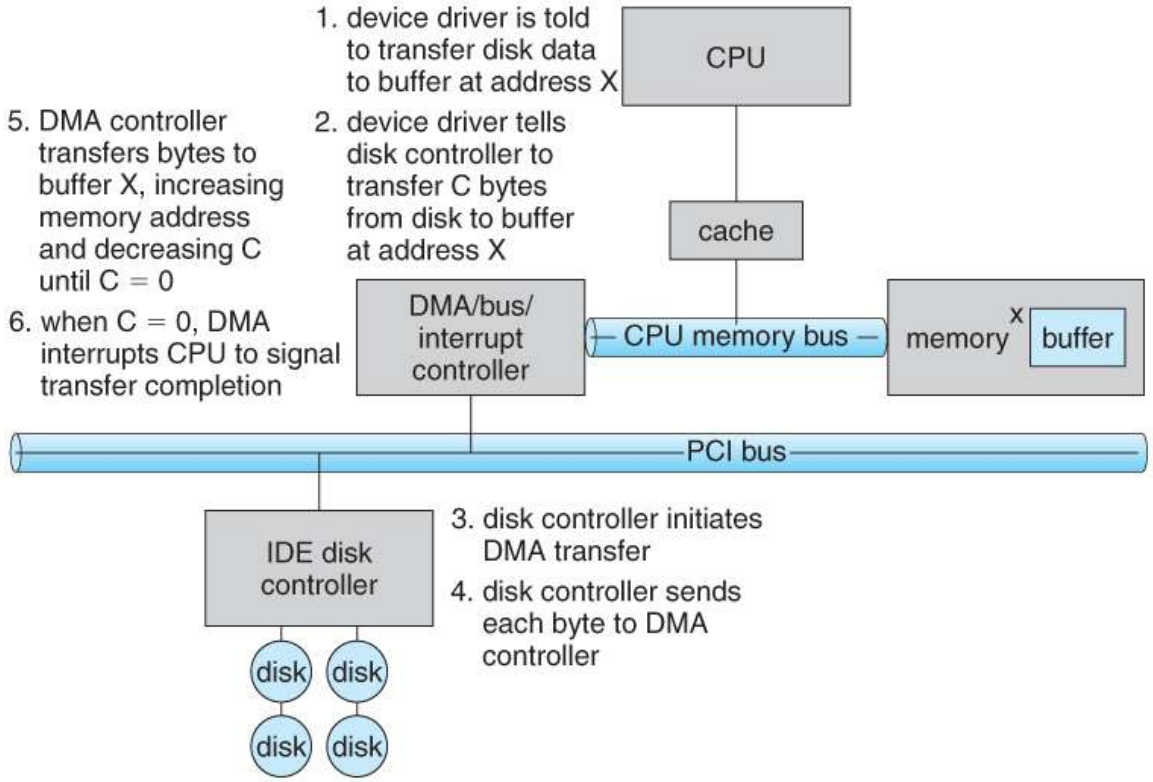
- Polling
  - CPU periodically checks for the I/O task status
- Interrupt-driven
  - CPU receives an interrupt from the controller (device or DMA) once the I/O task is done (either successfully or abnormally)
- Programmed I/O
  - CPU does the actual work of moving data
- Direct Memory Access (DMA)
  - CPU delegates off the work to a dedicated DMA controller

WHO?

Interrupt-driven I/O



Direct Memory Access (DMA)



Overcome the limitation of Programmed I/O

Maybe wasteful to tie up the CPU transferring data in and out of registers one byte at a time

Useful for devices that transfer large quantities of data (such as disk controllers)

Typically, used in combination with interrupt-driven I/O

Architectural Features Enabling OS Services

OS Service	HW Support
Protection and Security	Kernel/user mode, protected instructions, base/limit registers
System calls	Trap instructions and interrupt vectors
Exception handling	Trap instructions and interrupt vectors
I/O operations	Trap instructions, interrupt vectors, and memory mapping
Scheduling	Timer
Synchronization	Atomic instructions
Virtual memory	Translation Look-aside Buffer (TLB)

Protection and Security

Privileged Instructions

- Some CPU instructions are more sensitive than others
  - `MOV %eax, %ebx` → move the content of the register `ebx` into `eax`
  - `MOV %eax, [%ebx]` → move the content of memory indexed by register `ebx` to `eax`
  - `HLT` → halt the system
  - `INT X` → generate interrupt `X`

Privileged Instructions

- Some CPU instructions are more sensitive than others
  - `MOV %eax, %ebx` → move the content of the register `ebx` into `eax`
  - `MOV %eax, [%ebx]` → move the content of memory indexed by register `ebx` to `eax`
  - `HLT` → halt the system
  - `INT X` → generate interrupt `X`

Privileged Instructions

- Some CPU instructions are more sensitive than others
  - `MOV %eax, %ebx` → move the content of the register `ebx` into `eax`
  - `MOV %eax, [%ebx]` → move the content of memory indexed by register `ebx` to `eax`
  - `HLT` → halt the system
  - `INT X` → generate interrupt `X`

Idea: sensitive (privileged) instructions can be executed only by the OS

Kernel vs. User Mode

- 2 different "states" while executing CPU instructions

Kernel vs. User Mode

- 2 different "states" while executing CPU instructions
- Kernel mode is unrestricted:
  - The OS can perform any instruction (including privileged ones)

Kernel vs. User Mode

- 2 different "states" while executing CPU instructions
- Kernel mode is unrestricted:
  - The OS can perform any instruction (including privileged ones)
- User mode is restricted so that the user is **not** able to:
  - Address I/O (directly)
  - Manipulate the content of main memory
  - Halt the machine
  - Switch to kernel mode
  - etc.

Kernel vs. User Mode

- 2 different "states" while executing CPU instructions
- Kernel mode is unrestricted:
  - The OS can perform any instruction (including privileged ones)
- User mode is restricted so that the user is **not** able to:
  - Address I/O (directly)
  - Manipulate the content of main memory
  - Halt the machine
  - Switch to kernel mode
  - etc.

Implemented in HW!
A status bit stored in a protected CPU register
(0=kernel, 1=user)

Beyond Kernel vs. User Mode

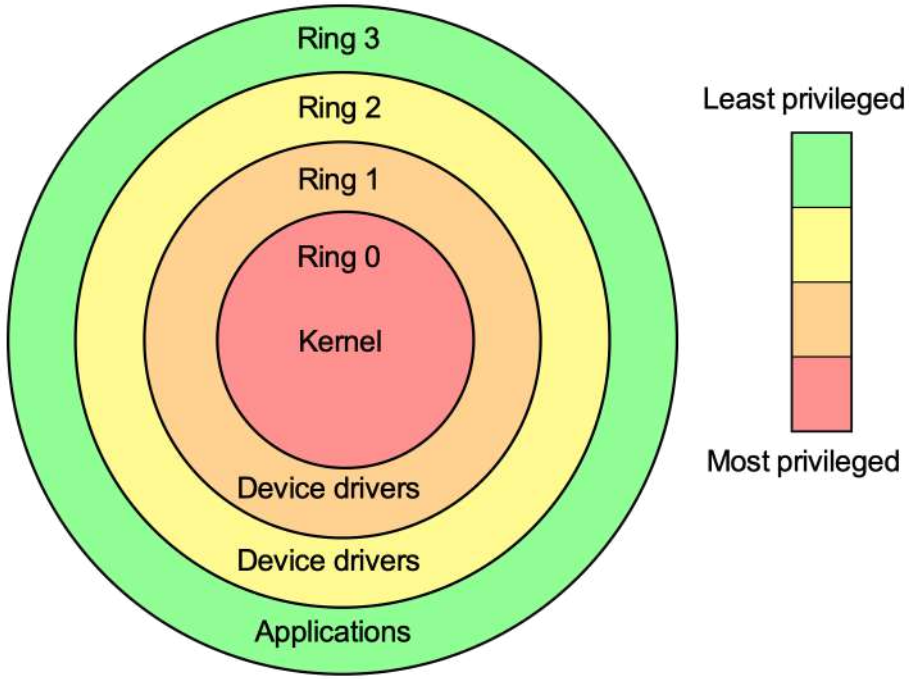
- The underlying HW must support at least kernel and user mode

Beyond Kernel vs. User Mode

- The underlying HW must support at least kernel and user mode
- More fine-grained solutions are also possible

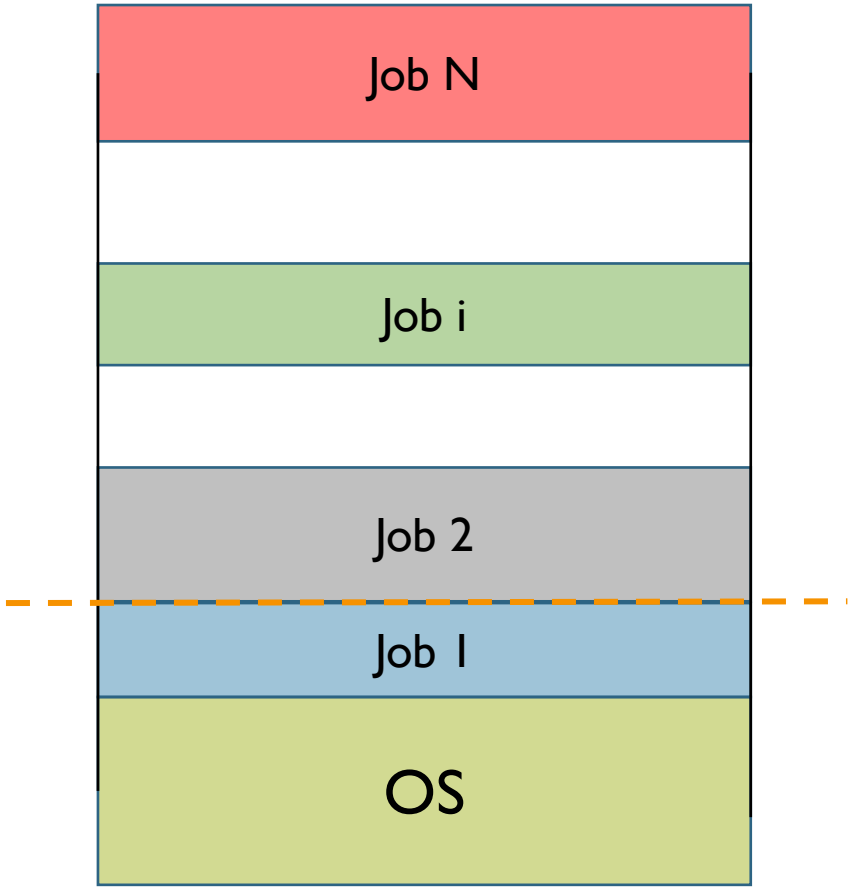
Beyond Kernel vs. User Mode

- The underlying HW must support at least kernel and user mode
- More fine-grained solutions are also possible
- Protection Rings
  - 4 different privilege levels {0, …, 3}
  - Still implementable in HW (2 bits)



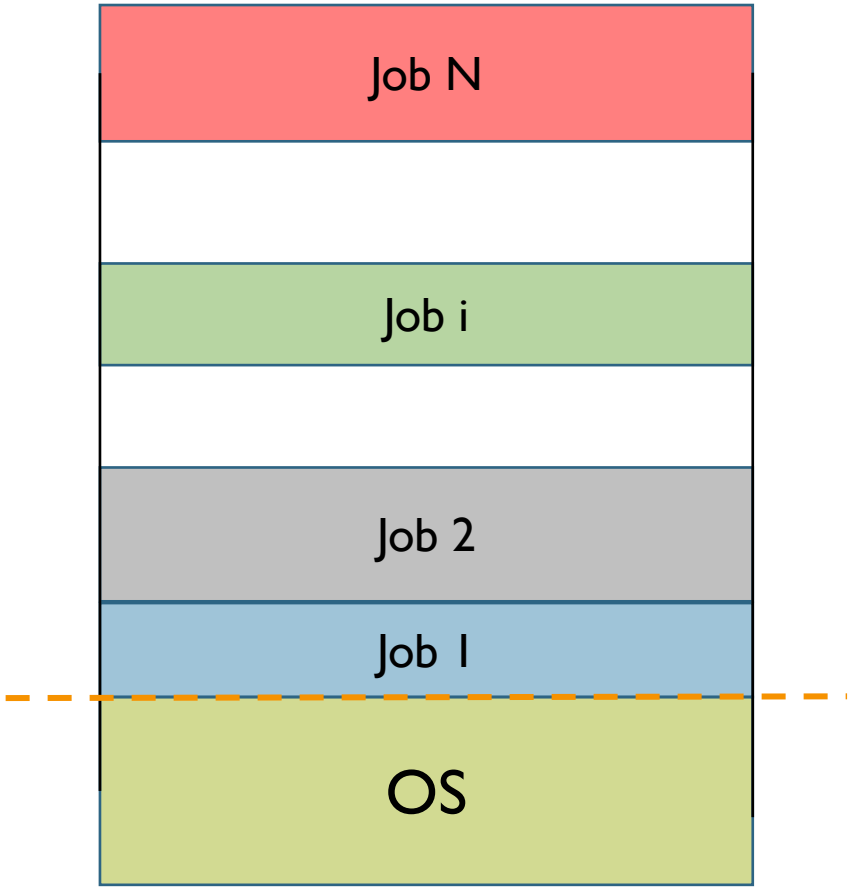
Memory Protection

- Architecture must provide support for the OS to:
  - Protect user programs from each other



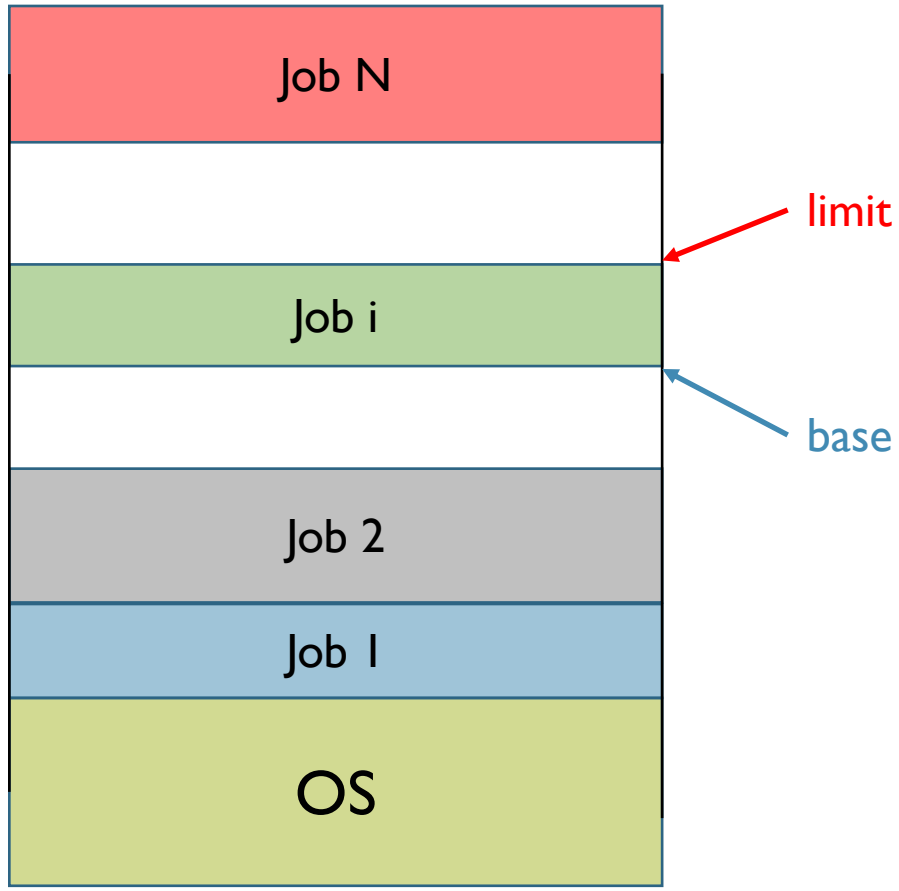
Memory Protection

- Architecture must provide support for the OS to:
  - Protect the OS from user programs



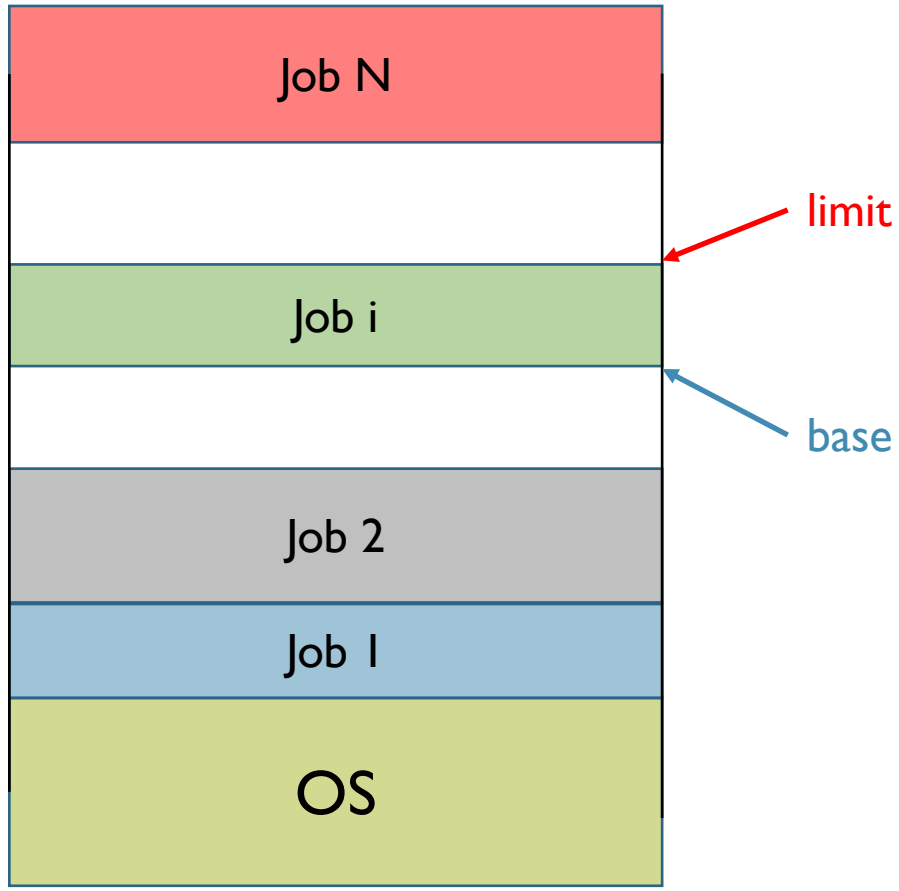
Memory Protection

- The simplest technique is to have 2 dedicated registers
  - base → contains the starting valid memory address
  - limit → contains the last valid memory address



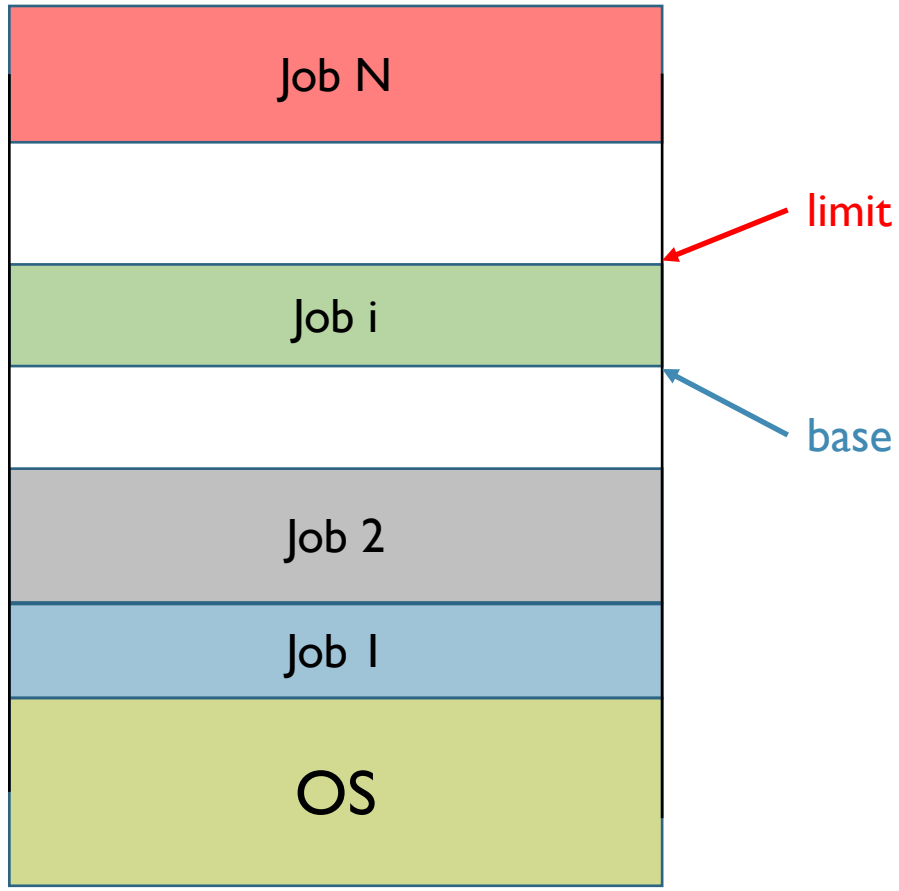
Memory Protection

- The simplest technique is to have 2 dedicated registers
  - base → contains the starting valid memory address
  - limit → contains the last valid memory address
- The OS loads the base and limit registers upon program startup



Memory Protection

- The simplest technique is to have 2 dedicated registers
  - base → contains the starting valid memory address
  - limit → contains the last valid memory address
- The OS loads the base and limit registers upon program startup
- The CPU checks each memory address referenced by user program falls between base and limit values



Program vs. Process

- A program is an executable file which resides on the persistent memory (e.g., disk),
  - contains only the set of instructions needed to accomplish a specific job
  - e.g., the **`ls`** program is an executable file stored at **`/bin/ls`** on the disk of a UNIX-like OS

Program vs. Process

- A program is an executable file which resides on the persistent memory (e.g., disk),
  - contains only the set of instructions needed to accomplish a specific job
  - e.g., the **`ls`** program is an executable file stored at **`/bin/ls`** on the disk of a UNIX-like OS
- A process is a particular instance of a program when loaded to main memory
  - e.g., multiple instances of the **`ls`** program above, thus multiple processes for the same program

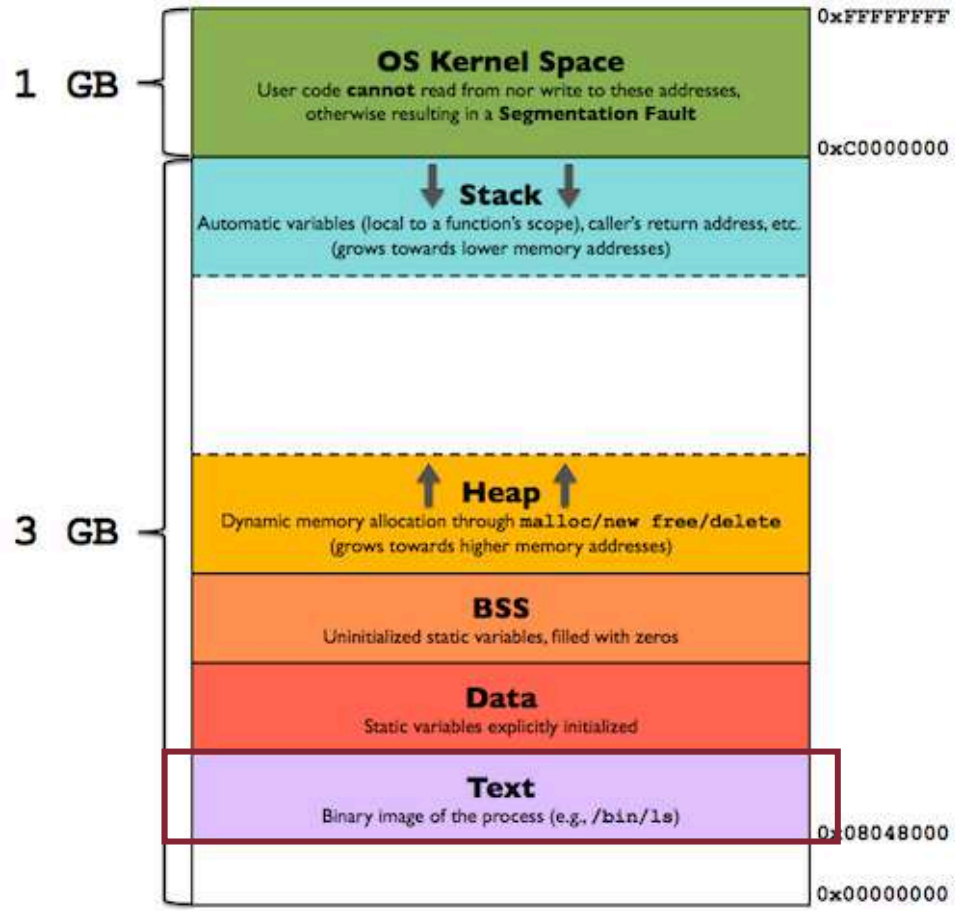
Program vs. Process

- A program is an executable file which resides on the persistent memory (e.g., disk),
  - contains only the set of instructions needed to accomplish a specific job
  - e.g., the **`ls`** program is an executable file stored at **`/bin/ls`** on the disk of a UNIX-like OS
- A process is a particular instance of a program when loaded to main memory
  - e.g., multiple instances of the **`ls`** program above, thus multiple processes for the same program

program → "static/passive" vs. process → "dynamic/active"

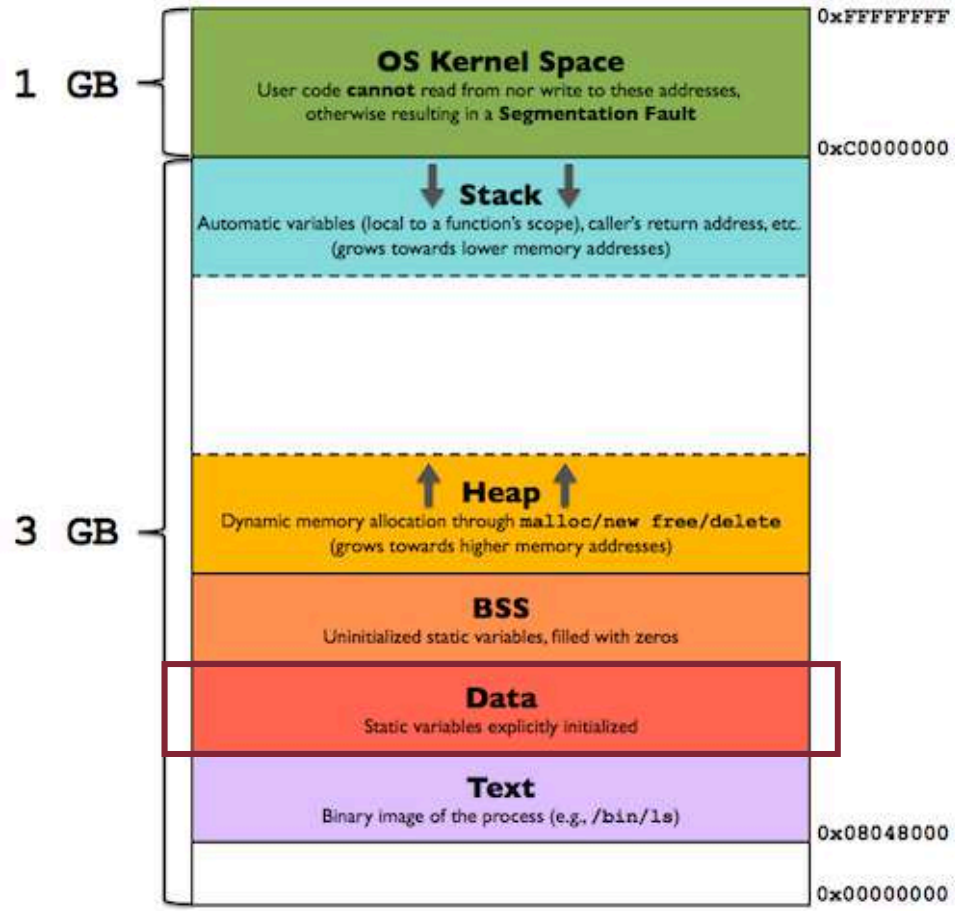
Process: In-Memory Layout

- Text → contains executable instructions



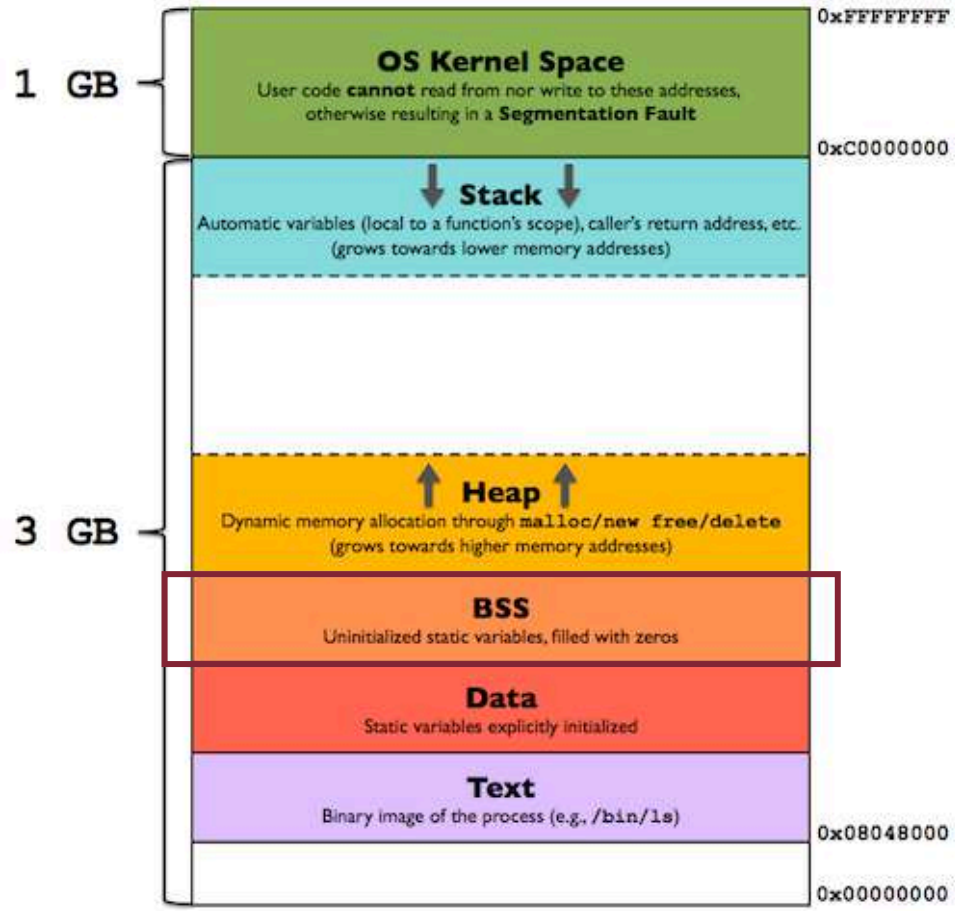
Process: In-Memory Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)



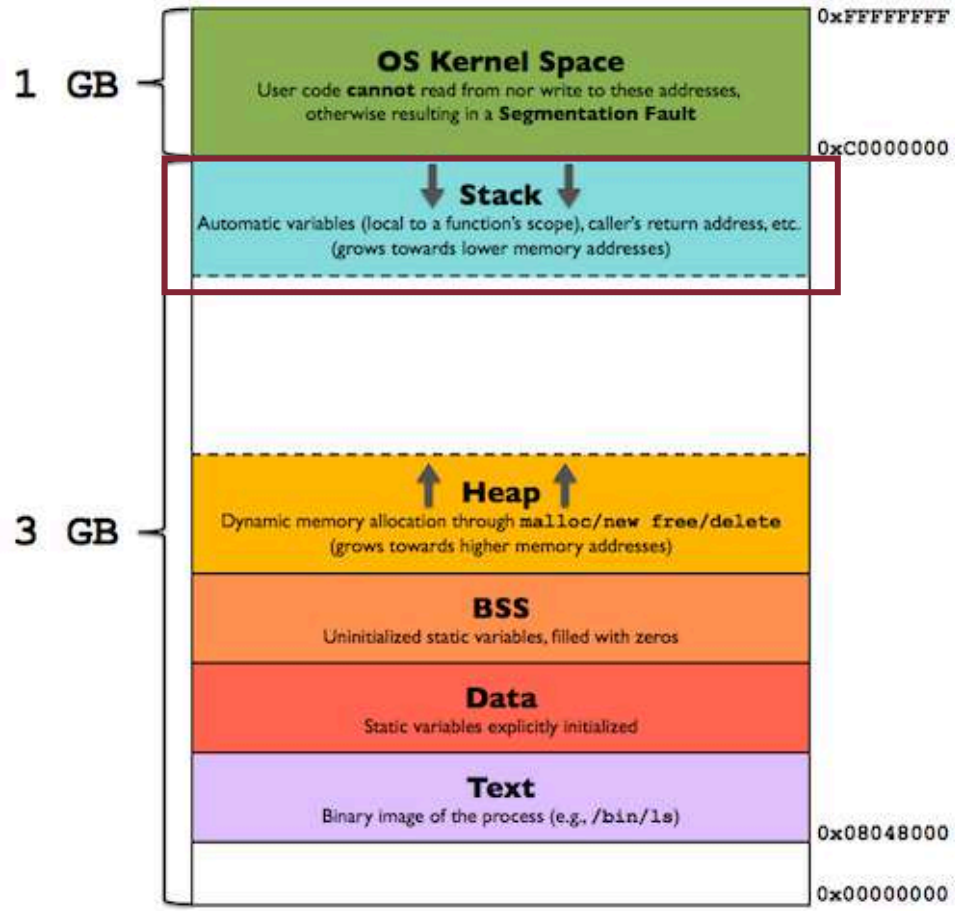
Process: In-Memory Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)
- BSS (Block Started by Symbol) → global and static variable (uninitialized or initialized to 0)



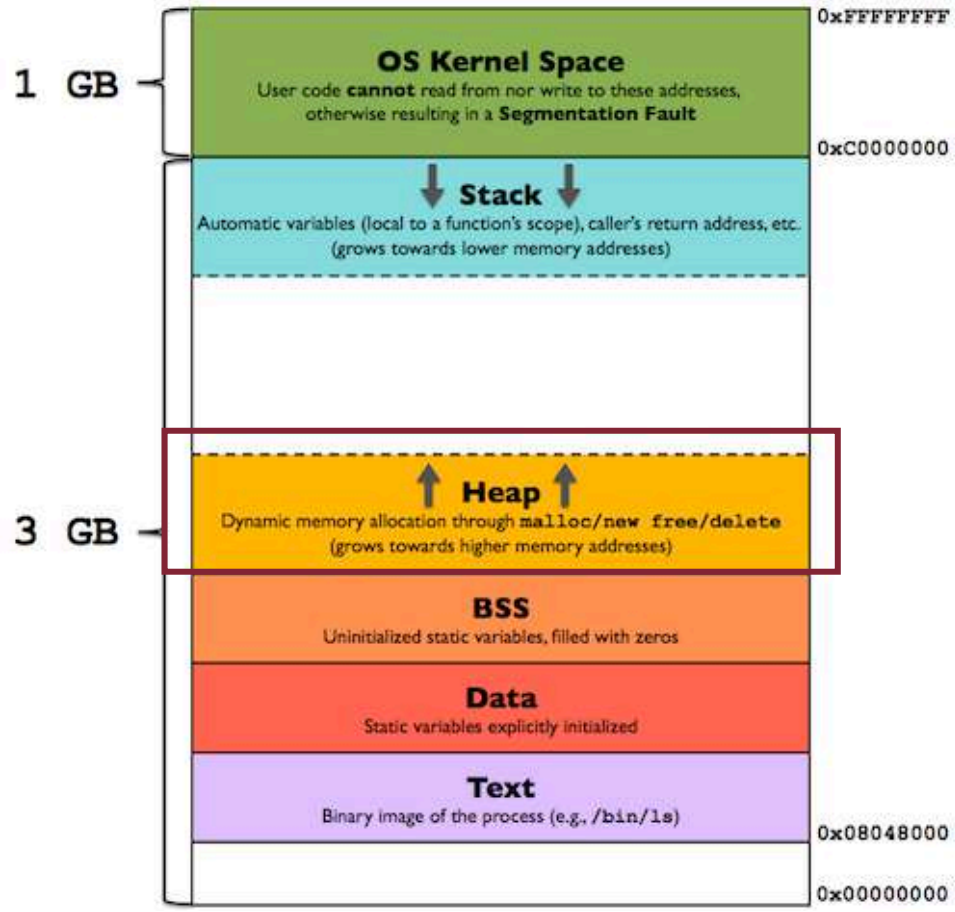
Process: In-Memory Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)
- BSS (Block Started by Symbol) → global and static variable (uninitialized or initialized to 0)
- Stack → LIFO structure used to store all the data needed by a function call (**stack frame**)



Process: In-Memory Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)
- BSS (Block Started by Symbol) → global and static variable (uninitialized or initialized to 0)
- Stack → LIFO structure used to store all the data needed by a function call (**stack frame**)
- Heap → used for dynamic allocation



Function Call: Stack Frame

- Each function uses a portion of the stack, and we call it a stack frame
- The stack frame for each function is divided into 3 parts:
  - function parameters
  - back-pointer to the previous stack frame
  - local variables

Stack Frame: Function Parameters

- This part of a function's stack frame is set up by the caller
- 2 operations are defined on a stack:
  - **`push`** → used to place items onto the stack
  - **`pop`** → user to remove items from the stack
- Different languages may push the parameters on in different orders

Stack Frame: Function Parameters

```
foo (a, b, c);
```

Stack Frame: Function Parameters

```
foo (a, b, c);
```

The caller converts to

```
push c
push b
push a
call foo
```

Stack Frame: Function Parameters

```
foo (a, b, c);
```

The caller converts to

```
push c
push b
push a
call foo
```

- Each item is pushed onto the stack, the stack grows down
- The stack-pointer register (**`esp`**) is decremented by 4 bytes (in 32-bit mode), and the item is copied to the memory location pointed to by the stack-pointer register
- The **`call`** instruction will implicitly push the return address on the stack

```
[esp + 0]   - return address
[esp + 4]   - parameter 'a'
[esp + 8]   - parameter 'b'
[esp + 12]  - parameter 'c'
```

Stack Frame: Base Pointer

```
[esp + 0]   - return address
[esp + 4]   - parameter 'a'
[esp + 8]   - parameter 'b'
[esp + 12]  - parameter 'c'
```

This way of accessing parameters from the callee might get clumsy, especially when several local variables need to be stored

Stack Frame: Base Pointer

```
[esp + 0]   - return address
[esp + 4]   - parameter 'a'
[esp + 8]   - parameter 'b'
[esp + 12]  - parameter 'c'
```

This way of accessing parameters from the callee might get clumsy, especially when several local variables need to be stored

Save the stack frame base pointer (**ebp**)!

```
push ebp        ; save previous stackbase-pointer register
mov  ebp, esp   ; ebp = esp
```

```
[ebp + 16]  - parameter 'c'
[ebp + 12]  - parameter 'b'
[ebp + 8]   - parameter 'a'
[ebp + 4]   - return address
[ebp + 0]   - saved stackbase-pointer register
```

Stack Frame: Putting All Together

```
[ebp + 16]  - parameter 'c'
[ebp + 12]  - parameter 'b'
[ebp + 8]   - parameter 'a'
[ebp + 4]   - return address
[ebp + 0]   - saved stackbase-pointer register
```

Parameters (of the caller) are accessed through the stack frame base pointer (**ebp**)

Stack Frame: Putting All Together

```
[ebp + 16]  - parameter 'c'
[ebp + 12]  - parameter 'b'
[ebp + 8]   - parameter 'a'
[ebp + 4]   - return address
[ebp + 0]   - saved stackbase-pointer register
```

Parameters (of the caller) are accessed through the stack frame base pointer (**ebp**)

```
[esp + (# - 4)] - top of local variables section
[esp + 0]       - bottom of local variables section
```

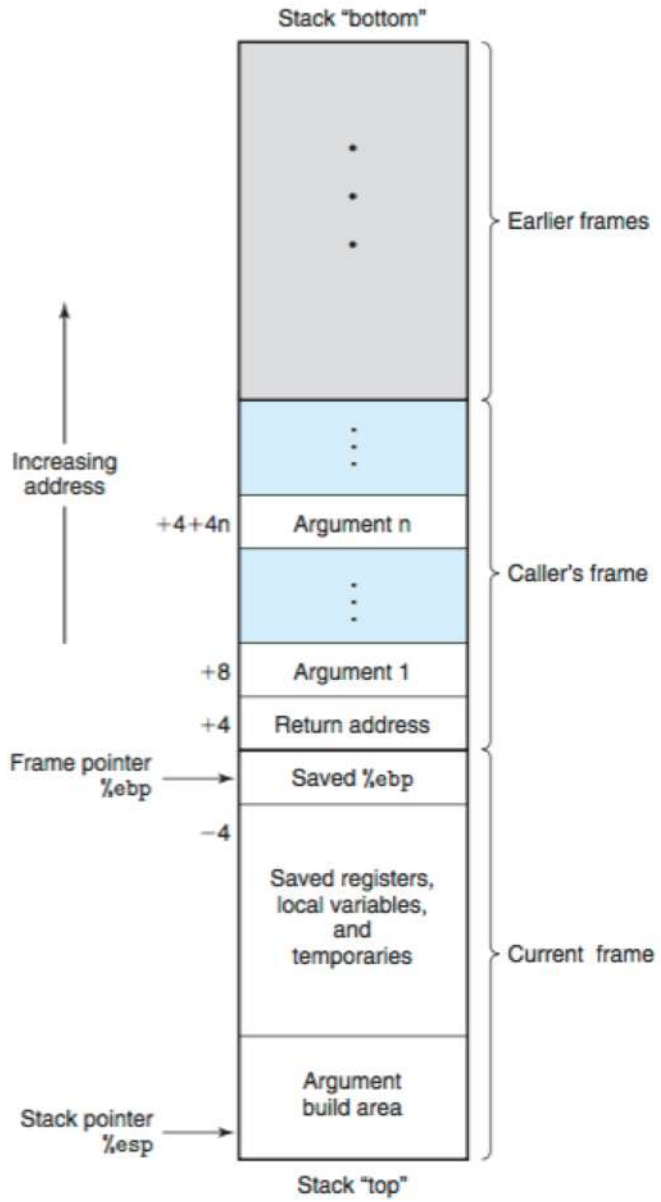
Local variables (of the callee) are accessed through the stack frame top pointer (**esp**)

Stack Frame: Cleanup

```
mov esp, ebp   ; undo the carving of space for the local variables
pop ebp        ; restore the previous stackbase-pointer register
```

The old stack frame base pointer is restored

Stack: Outline



System Calls, Exceptions, I/O Interrupts

Preserving Usability: System Calls

- Privileged instructions cannot be executed in user mode directly

Preserving Usability: System Calls

- Privileged instructions cannot be executed in user mode directly
- But programs running in user mode can ask the OS to perform some restricted operations on their behalf (in kernel mode)

Preserving Usability: System Calls

- Privileged instructions cannot be executed in user mode directly
- But programs running in user mode can ask the OS to perform some restricted operations on their behalf (in kernel mode)
- For example, a user program may require to:
  - write data to a file stored on disk
  - send data over the network interface
  - etc.

Preserving Usability: System Calls

- Privileged instructions cannot be executed in user mode directly
- But programs running in user mode can ask the OS to perform some restricted operations on their behalf (in kernel mode)
- For example, a user program may require to:
  - write data to a file stored on disk
  - send data over the network interface
  - etc.

Crossing protection boundaries using system calls

Exceptions and Interrupts

- Exceptions
  - software-generated
  - e.g., program error like division by 0

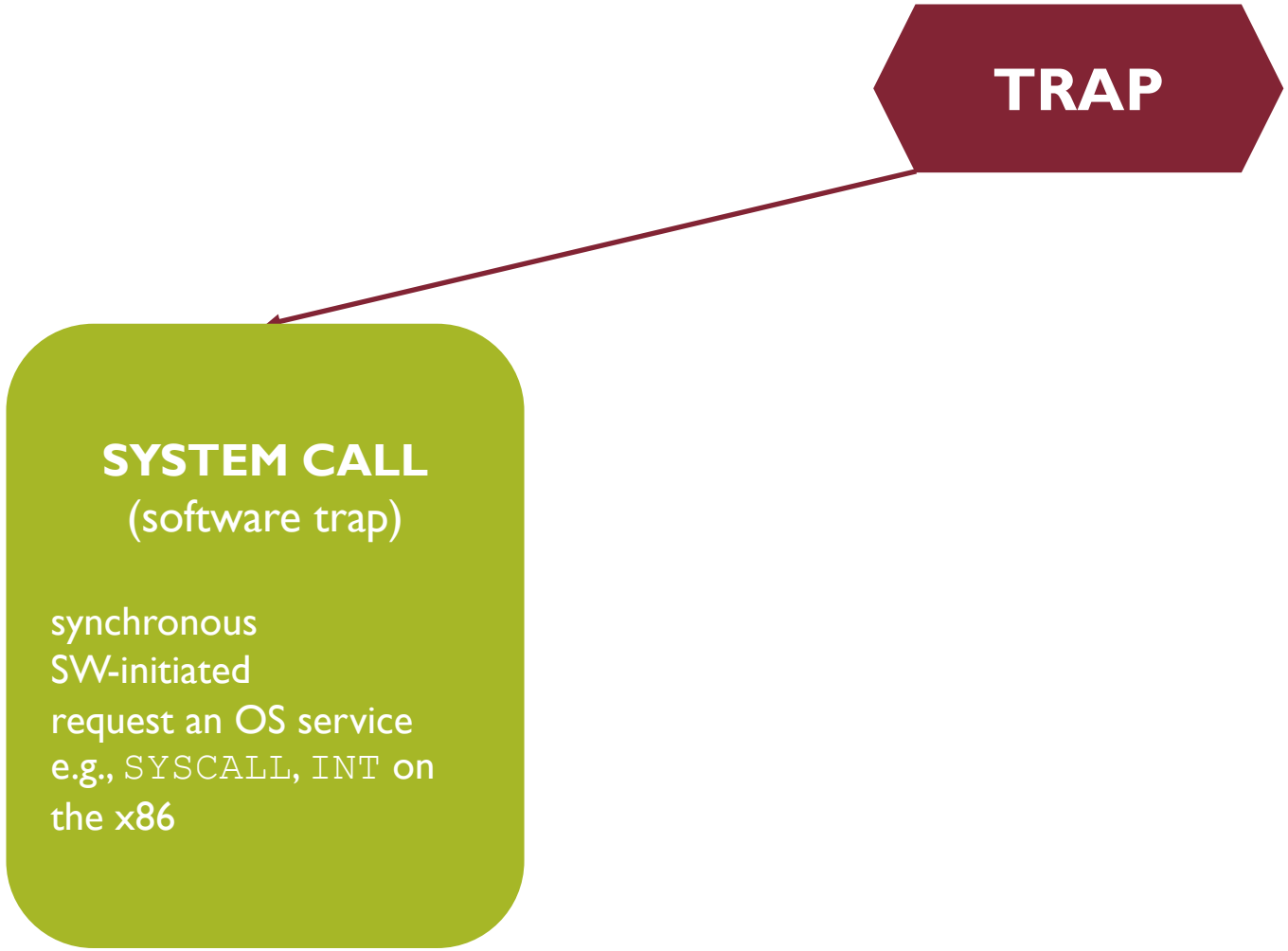
Exceptions and Interrupts

- Exceptions
  - software-generated
  - e.g., program error like division by 0
- Interrupts
  - hardware-generated (by external devices)
  - e.g., I/O completion or timer interrupt on a multi-tasking system

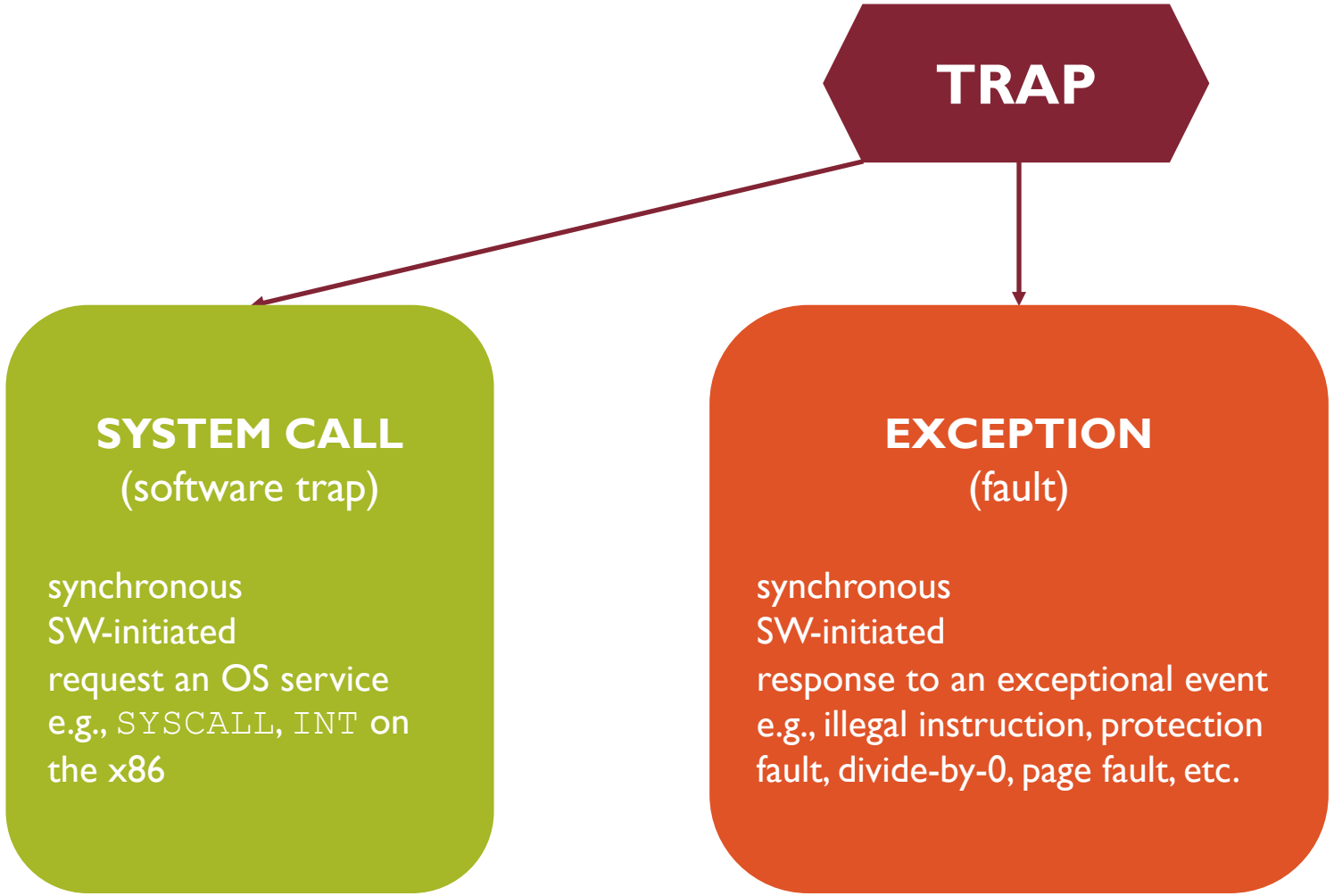
A Quick Note on Terminology

We will refer to trap as any event that causes switch to OS kernel mode

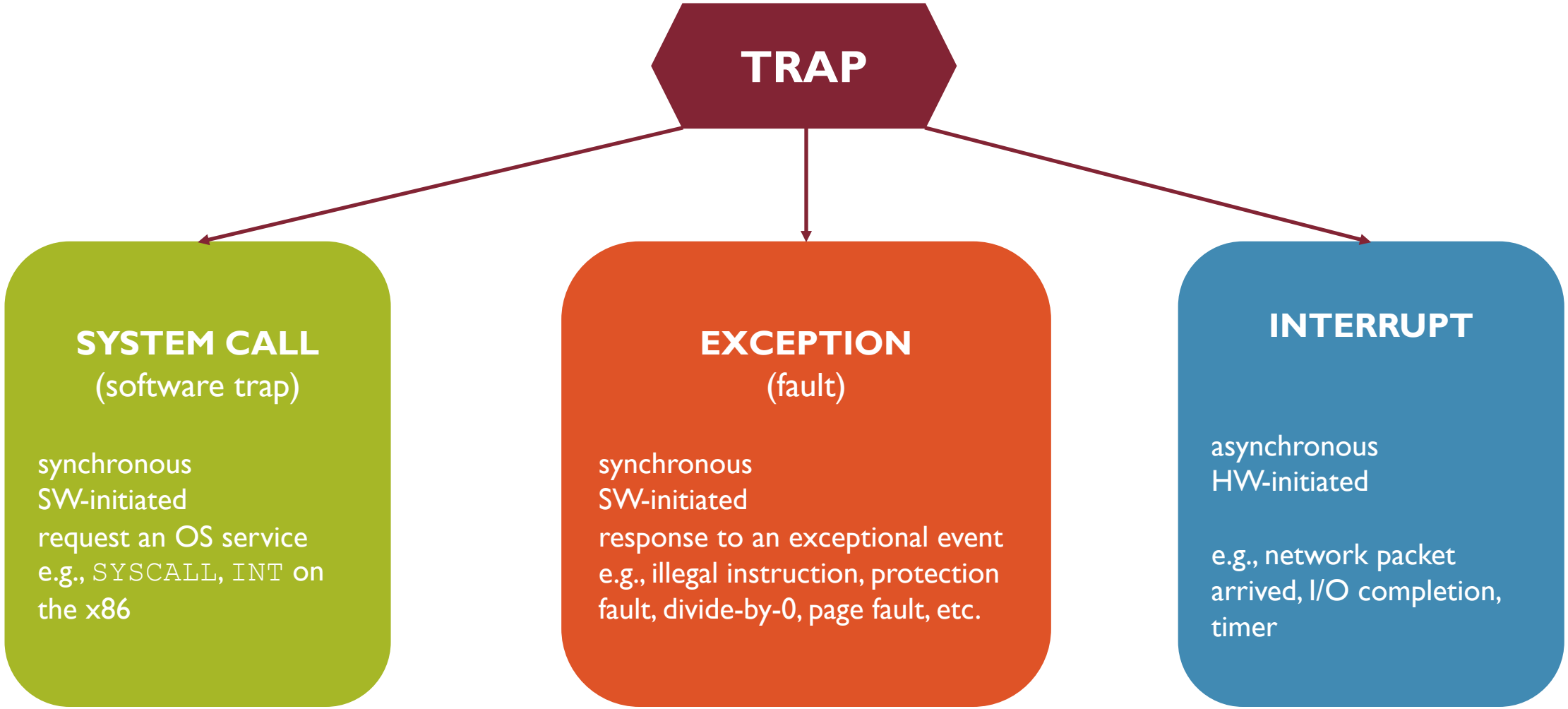
A Quick Note on Terminology



A Quick Note on Terminology



A Quick Note on Terminology



Scheduling and Synchronization

Timer

- Hardware facility to enable CPU scheduling

Timer

- Hardware facility to enable CPU scheduling
- It is just a clock which marks the time of the day

Timer

- Hardware facility to enable CPU scheduling
- It is just a clock which marks the time of the day
- In multi-tasking systems, allows the CPU not to be monopolized by "selfish" processes

Timer

- Hardware facility to enable CPU scheduling
- It is just a clock which marks the time of the day
- In multi-tasking systems, allows the CPU not to be monopolized by "selfish" processes
- The timer generates an interrupt every, say, 100 microseconds

Timer

- Hardware facility to enable CPU scheduling
- It is just a clock which marks the time of the day
- In multi-tasking systems, allows the CPU not to be monopolized by "selfish" processes
- The timer generates an interrupt every, say, 100 microseconds
- At each timer interrupt, the CPU scheduler takes over and decides which process to execute next

Atomic Instructions

- Interrupts may occur at any time and interfere with running processes

Atomic Instructions

- Interrupts may occur at any time and interfere with running processes
- OS must be able to synchronize the activities of cooperating, concurrent processes

Atomic Instructions

- Interrupts may occur at any time and interfere with running processes
- OS must be able to synchronize the activities of cooperating, concurrent processes
- Hardware must ensure that short sequences of instructions (e.g., read-modify-write) are executed atomically by either:
  - Disabling interrupts before the sequence and re-enable them afterwards

  or

  - Special instructions that are natively executed atomically

Virtual Memory

What is Virtual Memory?

- It is an abstraction (of the actual, physical main memory)

What is Virtual Memory?

- It is an abstraction (of the actual, physical main memory)
- It gives each process the illusion that physical memory is just a contiguous address space (virtual address space)

What is Virtual Memory?

- It is an abstraction (of the actual, physical main memory)
- It gives each process the illusion that physical memory is just a contiguous address space (virtual address space)
- It allows to run programs without them being entirely loaded in main memory
  - They are entirely loaded in virtual memory, though!

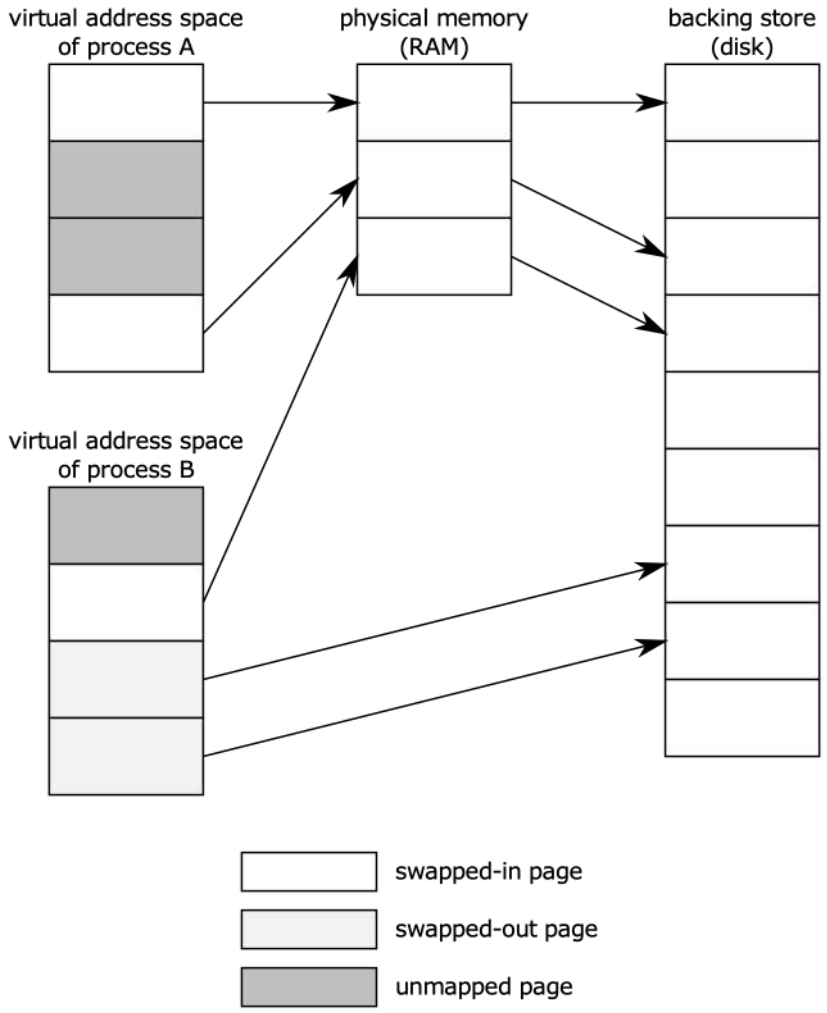
What is Virtual Memory?

- It is an abstraction (of the actual, physical main memory)
- It gives each process the illusion that physical memory is just a contiguous address space (virtual address space)
- It allows to run programs without them being entirely loaded in main memory
  - They are entirely loaded in virtual memory, though!
- Implemented both in HW (MMU) and SW (OS)
  - MMU is responsible for translating virtual addresses into physical ones
  - OS is responsible for managing virtual address spaces

Virtual vs. Physical Address Space

- On a 64 bit system the CPU is able to address 2^{64} bytes = 16 exbibytes (EiB)
- Virtual address space ranges from 0 to $2^{64} - 1$
- This is about a billion times more than main memory capacity currently available!
- Virtual address space is typically divided into contiguous blocks of the same size (e.g., 4 KiB), called pages
- Pages which are not loaded in main memory are stored on disk

Virtual vs. Physical Address Space



Memory Management Unit (MMU)

- Maps virtual addresses to physical ones through a page table managed by the OS

Memory Management Unit (MMU)

- Maps virtual addresses to physical ones through a page table managed by the OS
- Uses a cache called Translation Look-aside Buffer (TLB) with "recent mappings" for quicker lookups

Memory Management Unit (MMU)

- Maps virtual addresses to physical ones through a page table managed by the OS
- Uses a cache called Translation Look-aside Buffer (TLB) with "recent mappings" for quicker lookups
- The OS must be aware of which pages are loaded in main memory and which ones are on disk

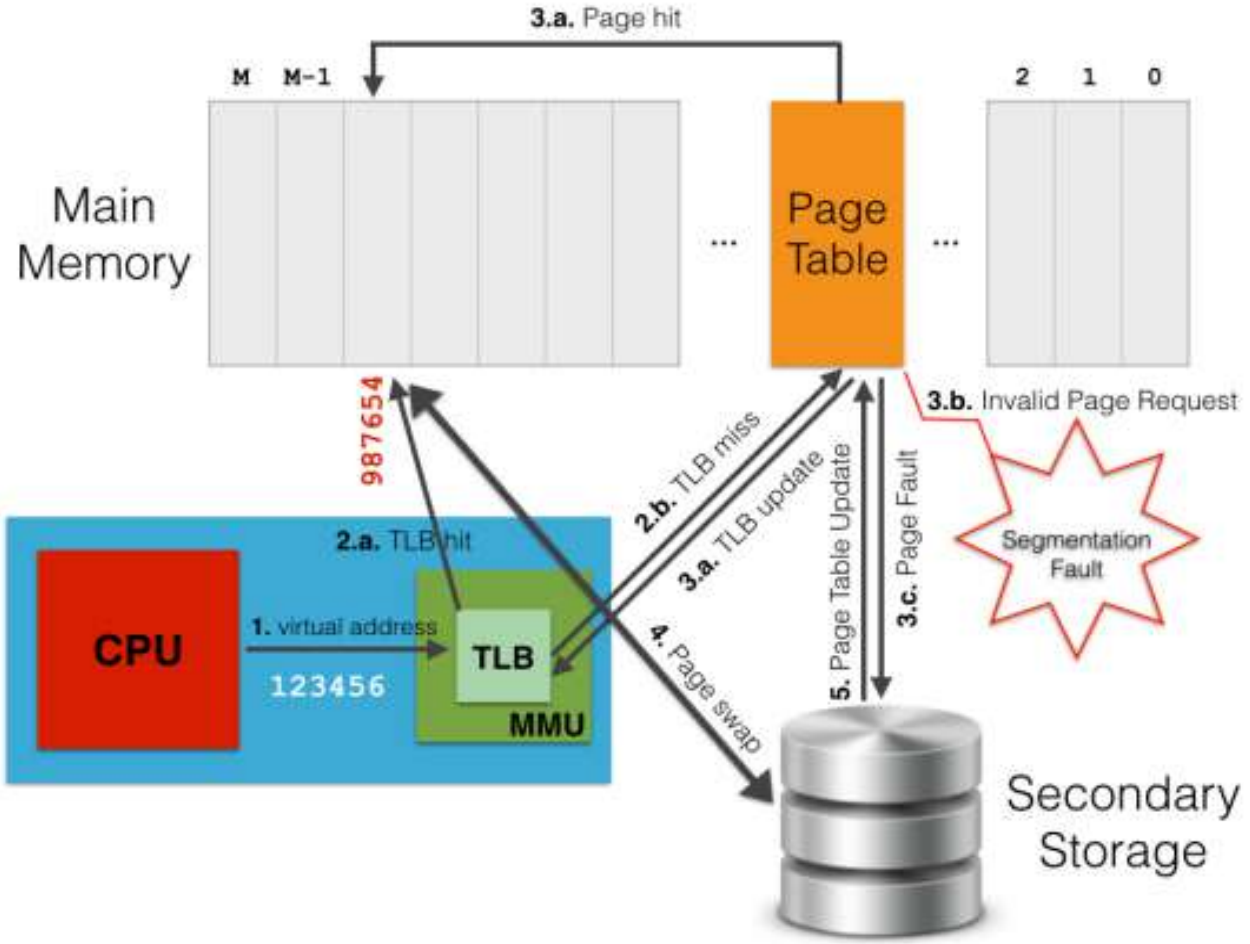
Lookup Fault

- Page lookup may fail due to 2 reasons:
  - no valid mapping exists for a virtual address → segmentation fault
  - requested page is not in main memory → page fault
- Both cases the Page Supervisor (OS kernel) takes over

Lookup Fault Handling

- In case of a page fault the Page Supervisor:
  - checks if there is enough room in main memory
  - if not, it has to free some space by removing a page from main memory and storing to disk (swapping)
    - several page replacement algorithms (e.g., LRU)
  - retrieves the missing page from disk and stores it in main memory
  - updates the Page Table and the TLB cache

Lookup Fault Handling



Summary

- Architecture support is key to OS design

Summary

- Architecture support is key to OS design
- Most of the services provided by the OS to the applications rely on specific HW features

Summary

- Architecture support is key to OS design
- Most of the services provided by the OS to the applications rely on specific HW features
- The OS is tightly coupled to the HW of the host machine

Summary

- Architecture support is key to OS design
- Most of the services provided by the OS to the applications rely on specific HW features
- The OS is tightly coupled to the HW of the host machine
- Advice: Keep your Computer Architecture book at hand!