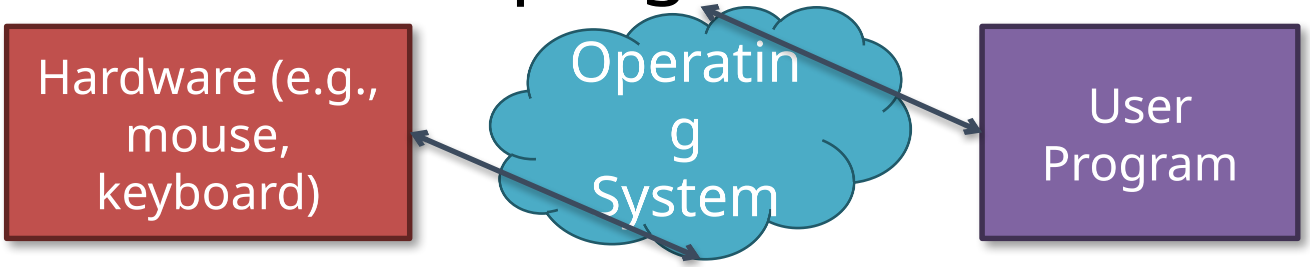
CS 5600
Computer Systems

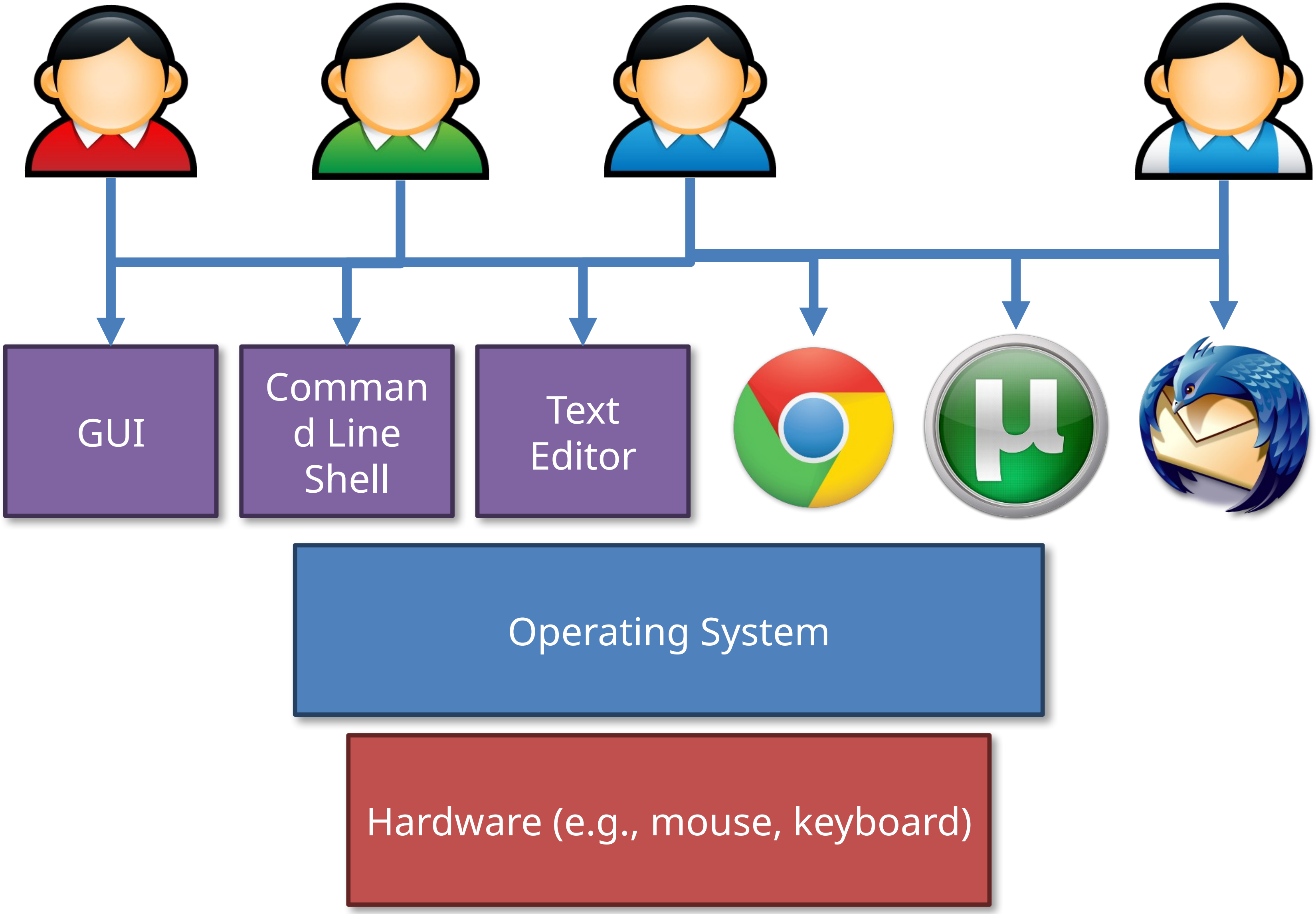
Lecture 3: Hardware, CPUs, and a Basic Operating System

What is an Operating System?

- An OS is any and all software that sits between a user program and the hardware



- OS is a resource manager and allocator
  - Decides between conflicting requests for hardware access
  - Attempts to be efficient and fair
- OS is a control program
  - Controls execution of user programs
  - Prevents errors and improper use



GUI

Command Line Shell

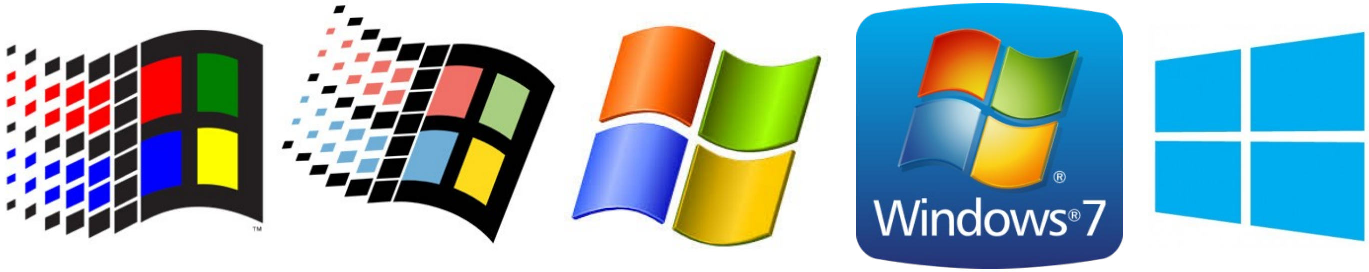
Text Editor

Operating System

Hardware (e.g., mouse, keyboard)

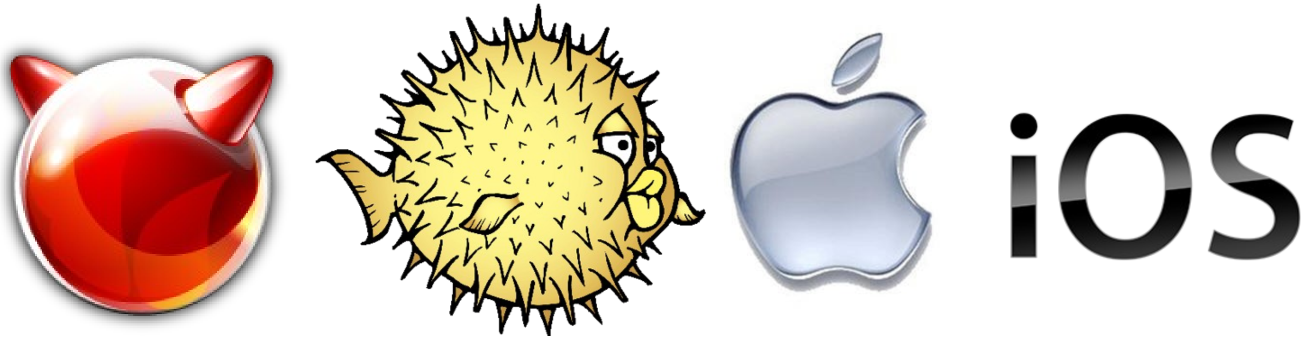
Many Different OSes

Windows



Linux

BSD



Goals for Today

- By the end of class, we will build a very, very simple command line OS
- But to get there, we need to understand the hardware platform we are building on
  - Basics of PC architecture
  - How devices and the CPU communicate
  - What is the basic functionality of the CPU
  - How do we boot into an OS

- Hardware Basics
- PC Bootup Sequence
- A Simple OS Example
- Kernels

Basic Computer Architecture

- Architecture determines many properties of the OS
  - How does the OS load and take control?
  - How does the OS interacts with devices?
  - How does the OS manage CPU and memory?
- This discussion will focus on the IBM PC architecture
  - Until recently, most computers could still run MS-DOS based programs from the 80's and 90's
  - Remains the most popular architecture
    - But not for long…
  - Alternatives: Amiga, Macintosh, PowerPC, etc.

Some History

- 1981: IBM releases a Personal Computer (PC) to compete with Apple
  - Basic Input/Output System (BIOS) for low-level control
  - Three high-level OSes, including MS-DOS
  - Developers were asked to write software for DOS or BIOS, not bare-metal hardware
- 1982: Compaq and others release IBM-compatible PCs
  - Different hardware implementations (except 808x CPU)
  - Reverse engineered and reimplemented BIOS
  - Relied on customized version of MS-DOS

In 1983, personal computing was anyone's game.

THE BATTLE FOR CTRL BEGINS

HALT AND CATCH FIRE

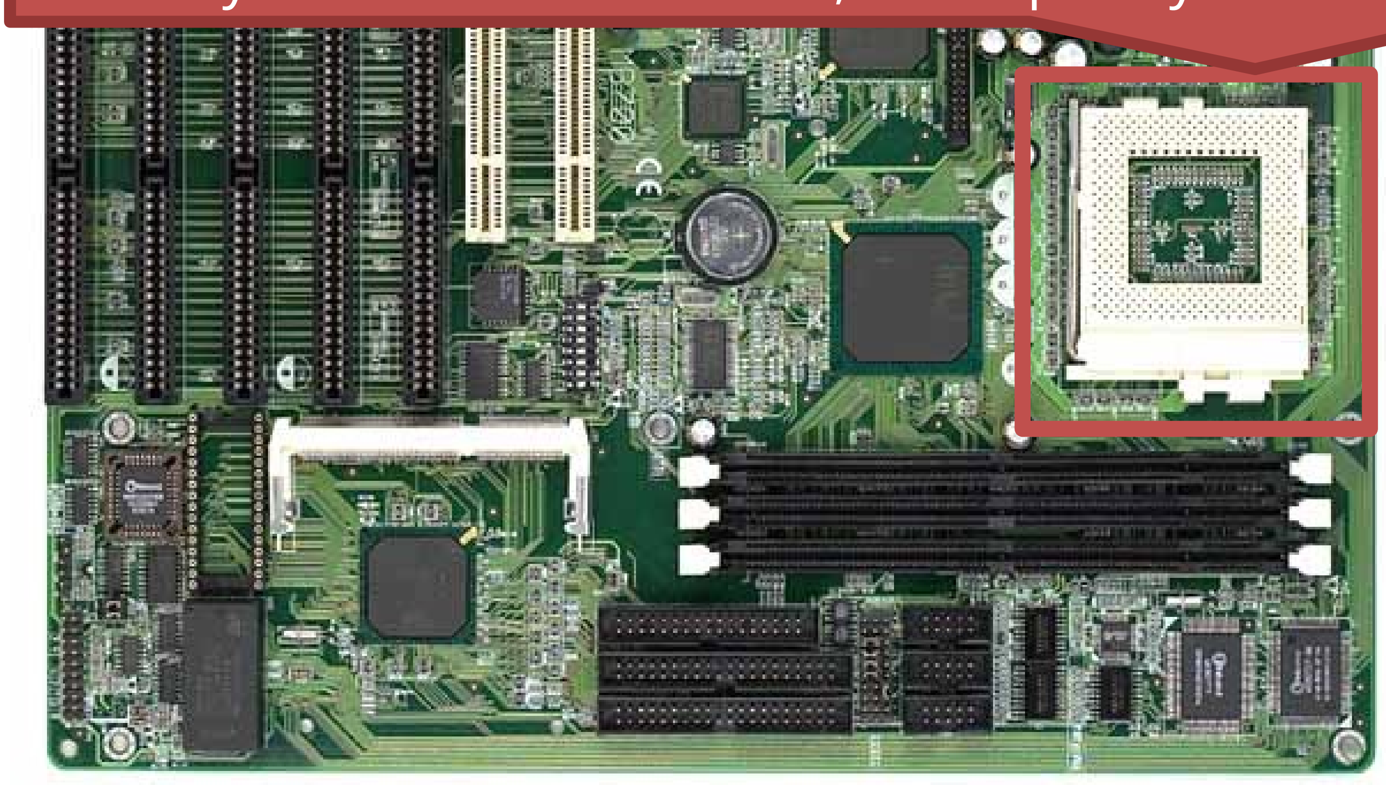
PREMIERES JUNE 1 SUNDAYS 10/9c

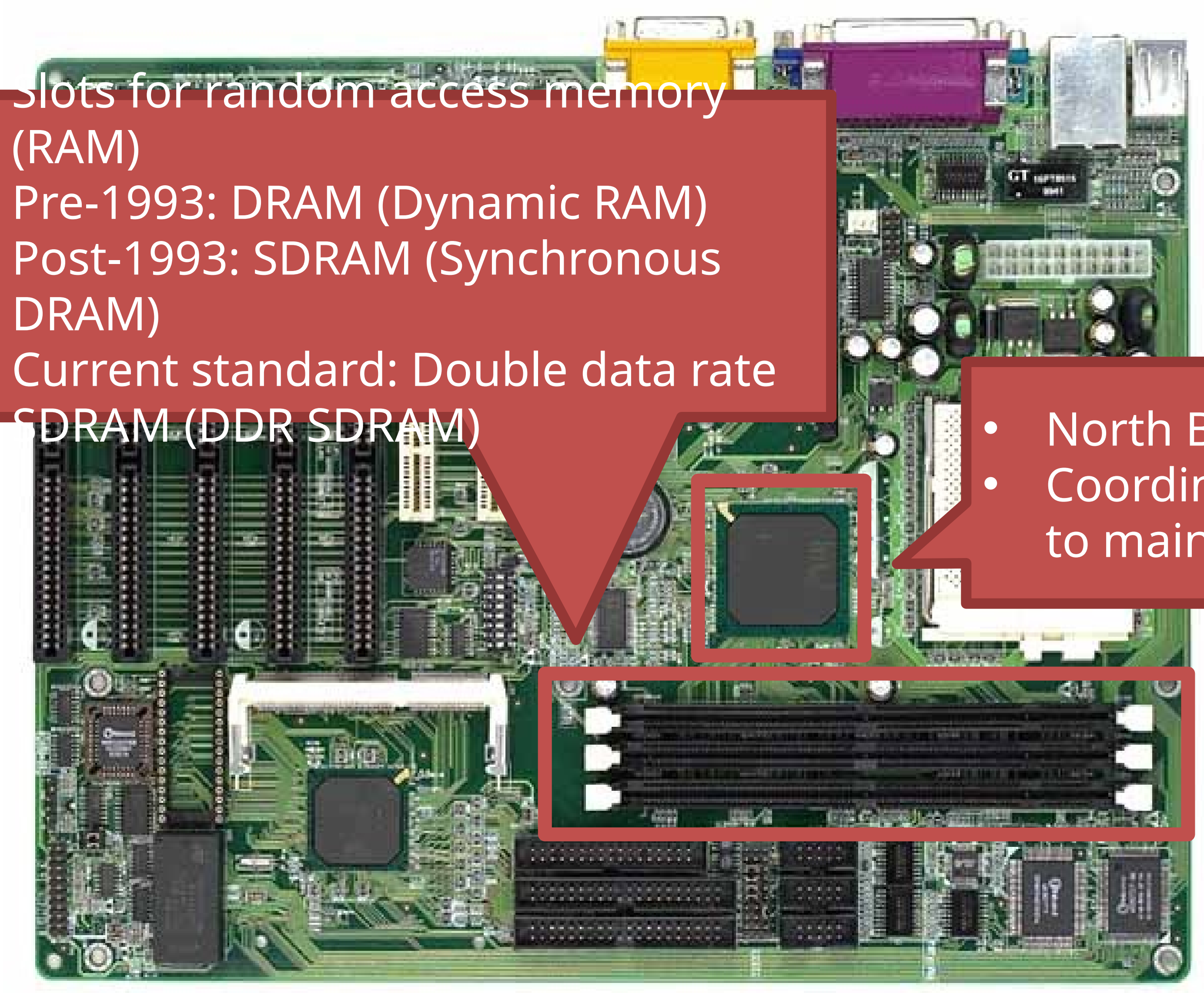
amc

IBM Loses Control

- 1985: IBM clones dominated computer sales
  - Used the same underlying CPUs and hardware chips
  - Close to 100% BIOS compatibility
  - MS-DOS was ubiquitous
  - Thus, IBM PC hardware became the de-facto standard
- 1986: Compaq introduces 80386-based PC
- 1990's: Industry is dominated by "WinTel"

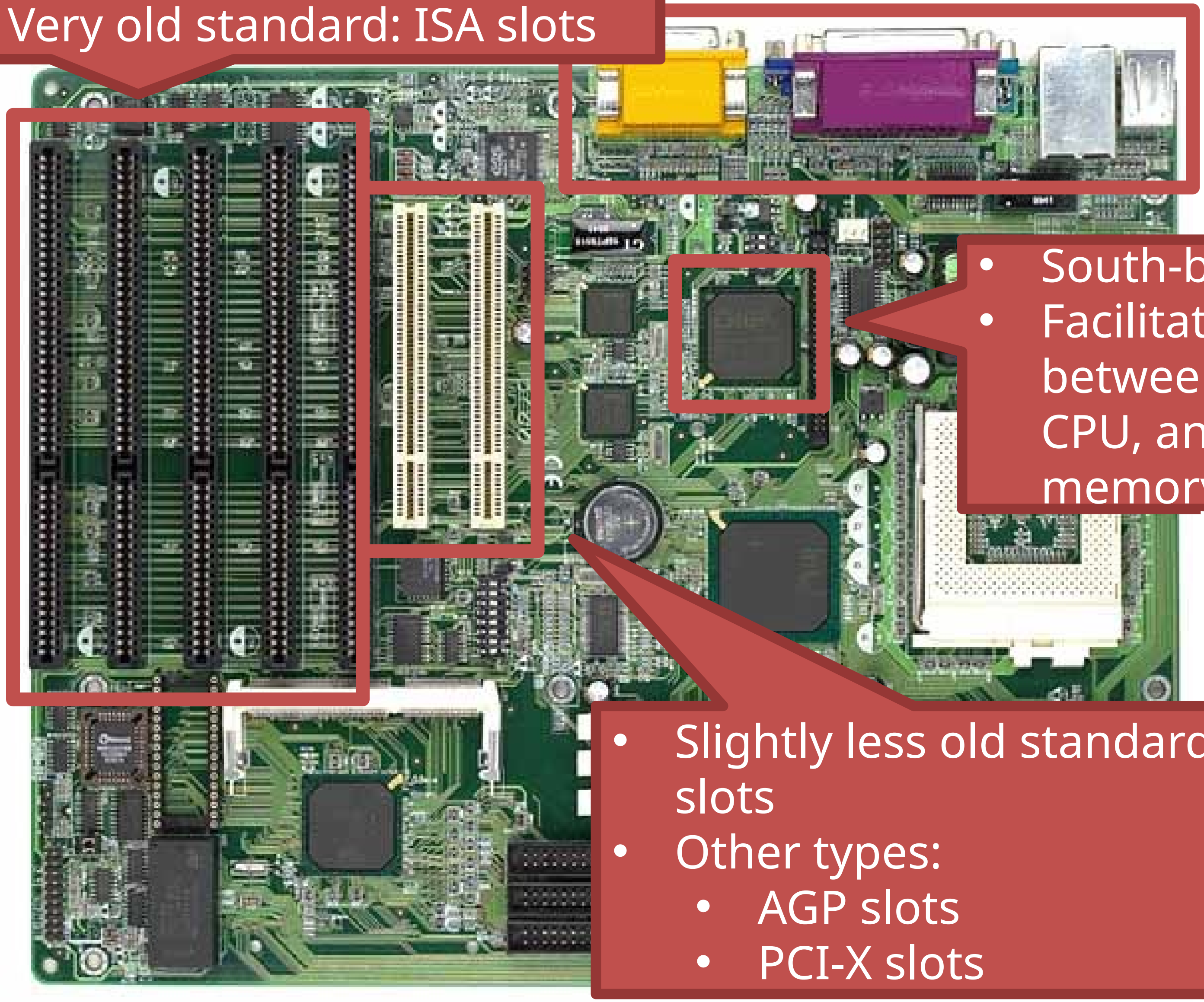
- CPU Socket
- Many different physical socket standards
  - This a Pentium 1 socket
- Physical standard is less important than Instruction Set Architecture (ISA)
  - IBM PCs are Intel 80836 compatible
  - Original x86 design
  - Intel, AMD, VIA
- Today’s dominant ISA: x86-64, developed by AMD





- Slots for random access memory (RAM)
- Pre-1993: DRAM (Dynamic RAM)
- Post-1993: SDRAM (Synchronous DRAM)
- Current standard: Double data rate SDRAM (DDR SDRAM)

- North Bridge
- Coordinates access to main memory

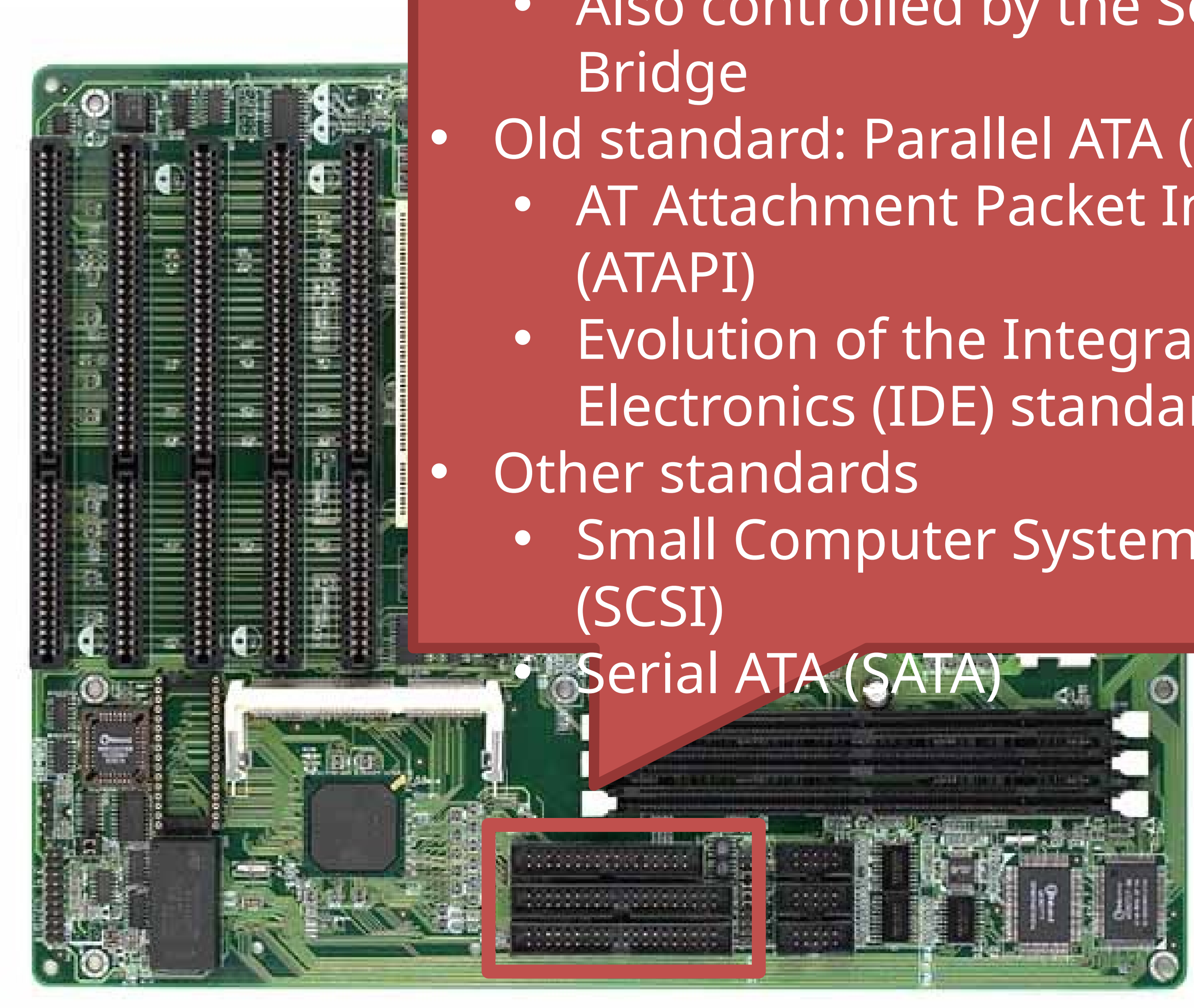


- I/O device slots
- Attached to the south-bridge bus
- Very old standard: ISA slots

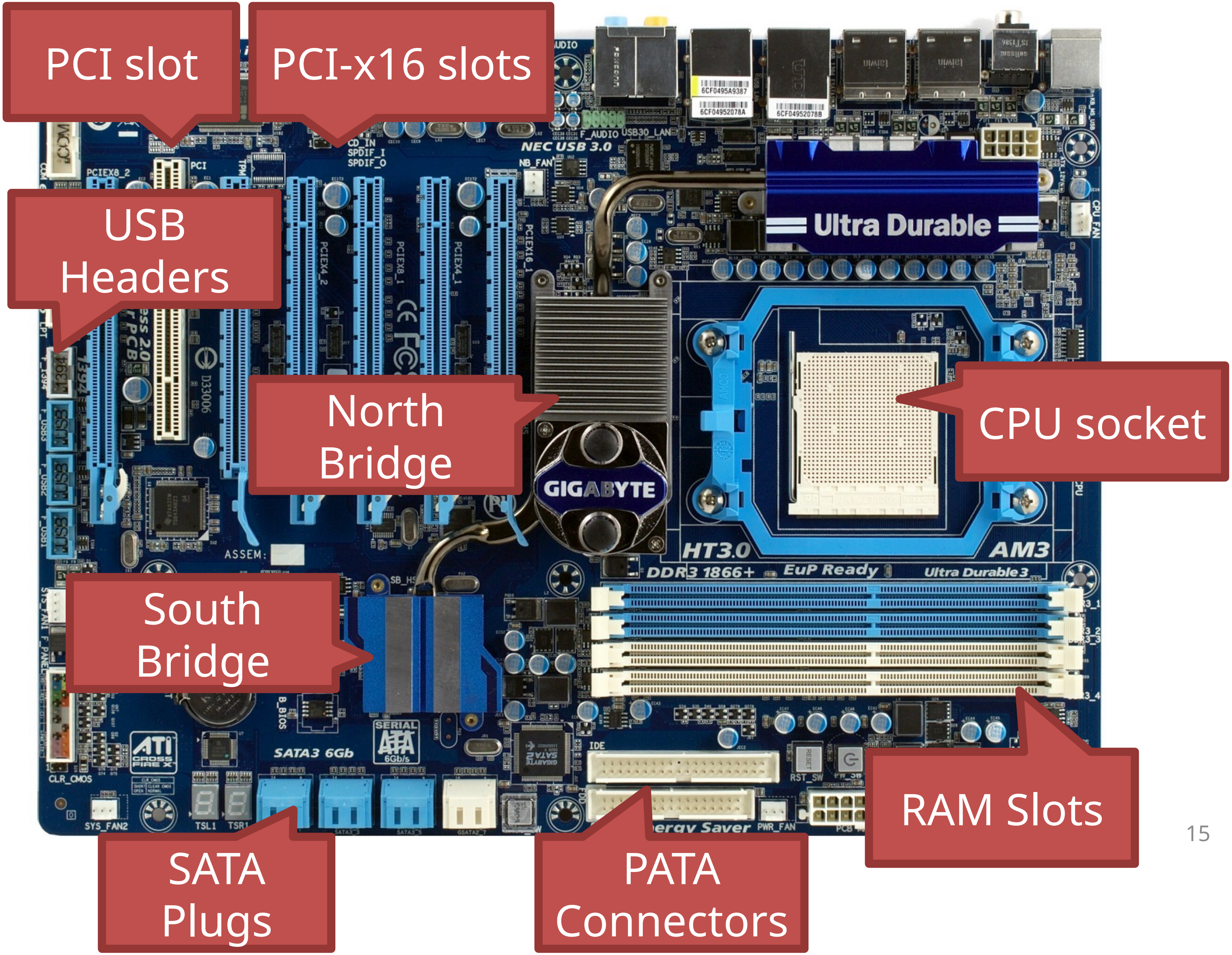
- Built in I/O also on the PCI/ISA bus

- South-bridge
- Facilitates I/O between devices, the CPU, and main memory

- Slightly less old standard: PCI slots
- Other types:
  - AGP slots
  - PCI-X slots



- Storage connectors
  - Also controlled by the South Bridge
- Old standard: Parallel ATA (P-ATA)
  - AT Attachment Packet Interface (ATAPI)
  - Evolution of the Integrated Drive Electronics (IDE) standard
- Other standards
  - Small Computer System Interface (SCSI)
  - Serial ATA (SATA)



PCI slot

PCI-x16 slots

USB Headers

North Bridge

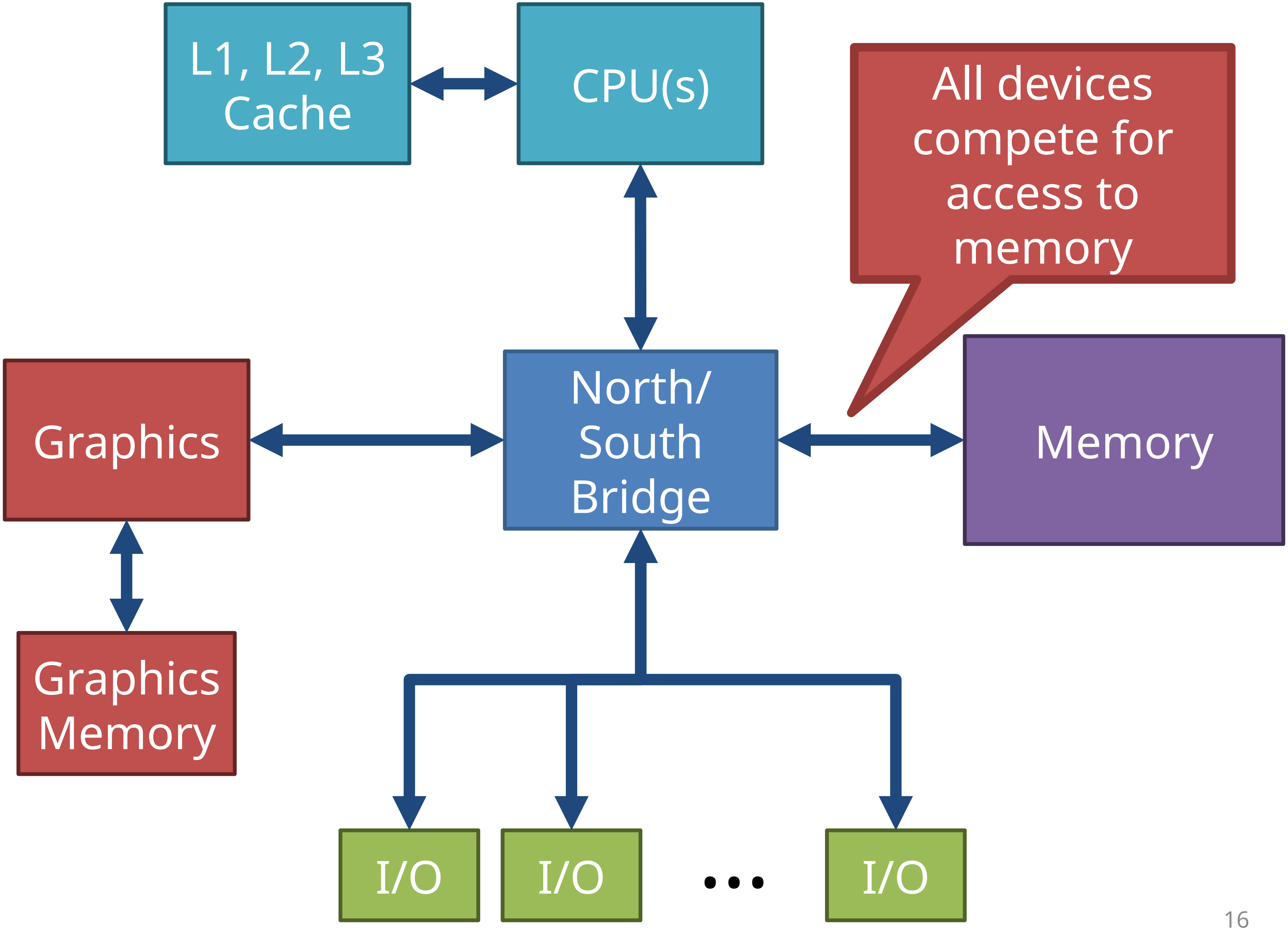
CPU socket

South Bridge

SATA Plugs

PATA Connectors

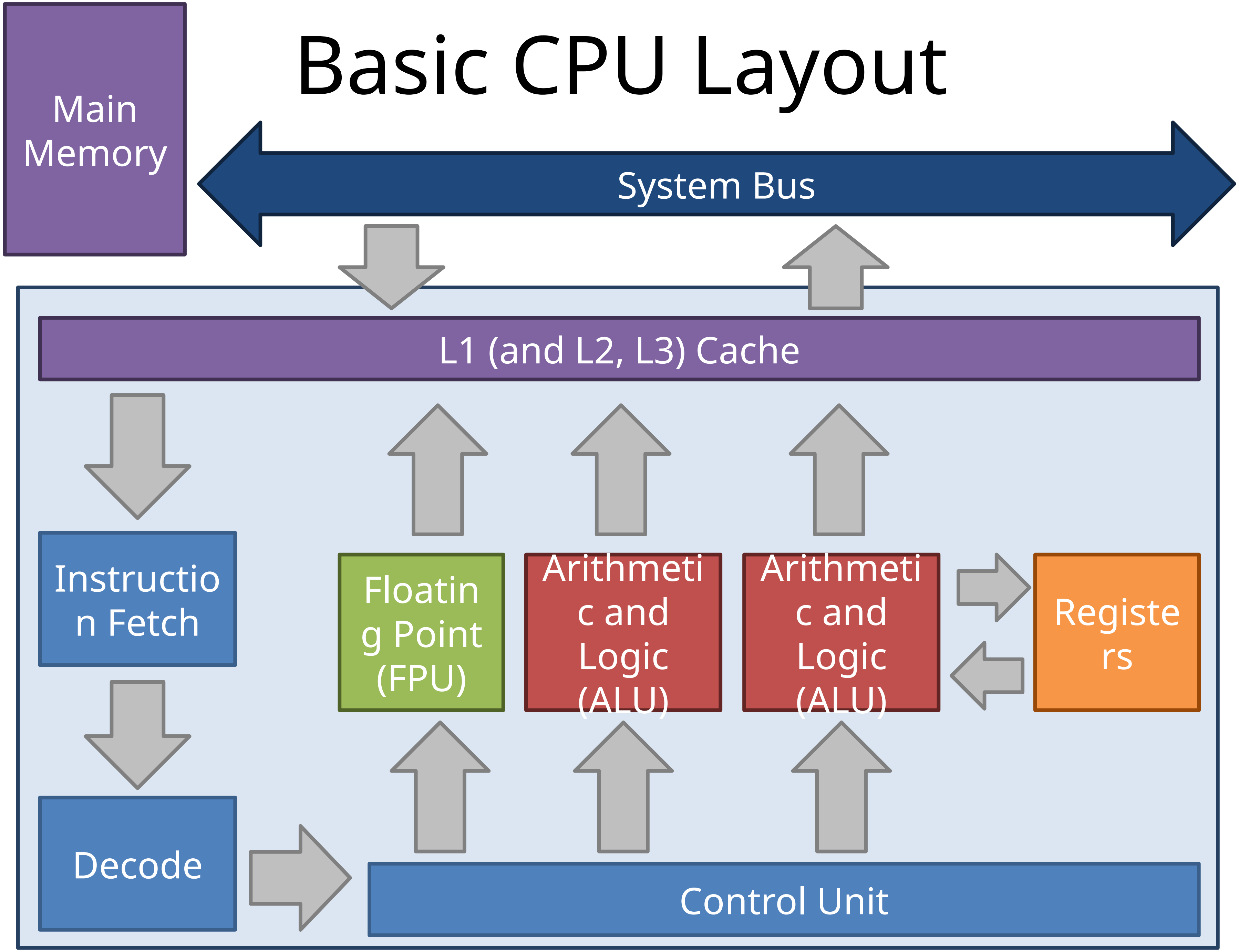
RAM Slots

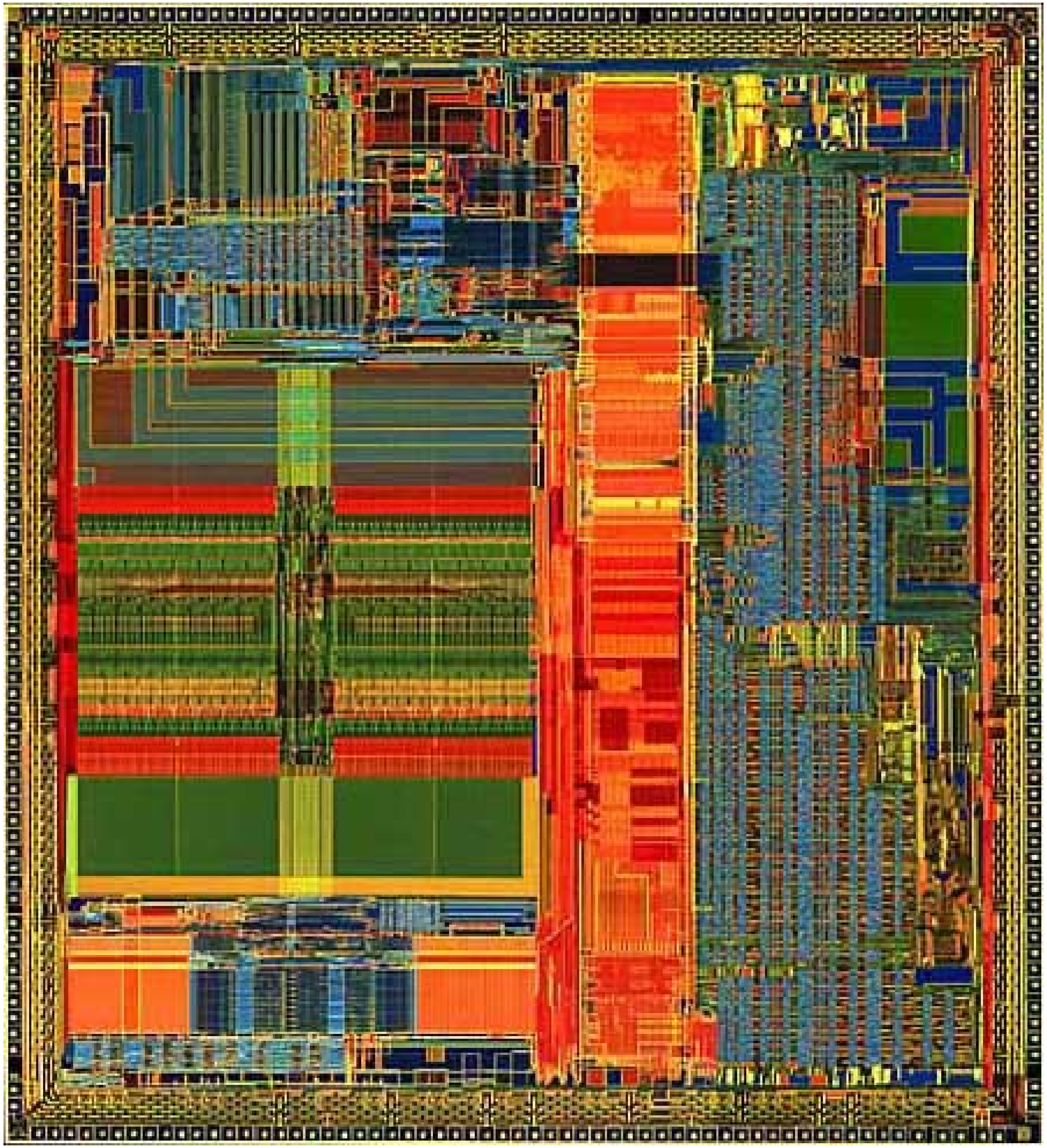

L1, L2, L3 Cache
CPU(s)
All devices compete for access to memory
Graphics
North/ South Bridge
Memory
Graphics Memory
I/O
I/O
…
I/O

x86 History

- 1978: Intel 8086 – 16 bit
- 1982: Intel 80286 – introduces protected mode and memory paging
- 1985: Intel 80386 – 32 bit
- 1989: Intel 80486 – integrates x87 FPU and cache
- 1993: Pentium and Pentium MMX
- 1997, 1999: Pentium II and III
- 2003, 2004: AMD Athlon 64 and Pentium 4
  - AMD pioneers the move to 64 bit x86-64

Basic CPU Layout

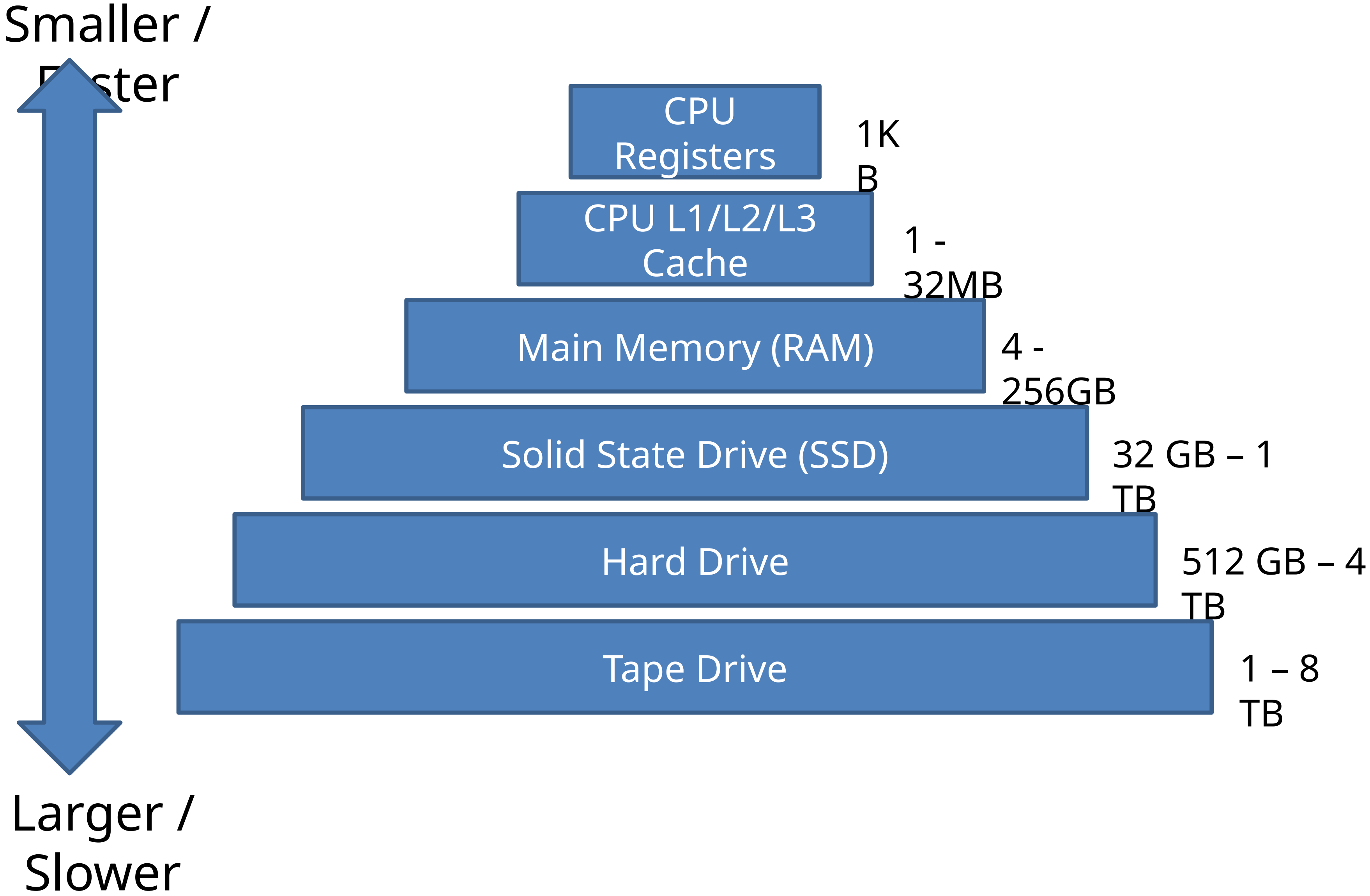




Registers

- Storage built in to the CPU
  - Can hold values or pointers
  - Instructions operate directly on registers
  - Things from memory can be loaded into registers…
  - … or things in registers can be moved to memory
- Some registers have special functions
  - Pointing to the current instruction in memory
  - Pointing to the top of the stack
  - Configuring low-level CPU features

Storage Hierarchy



Smaller / Faster

CPU Registers — 1K B

CPU L1/L2/L3 Cache — 1 - 32MB

Main Memory (RAM) — 4 - 256GB

Solid State Drive (SSD) — 32 GB – 1 TB

Hard Drive — 512 GB – 4 TB

Tape Drive — 1 – 8 TB

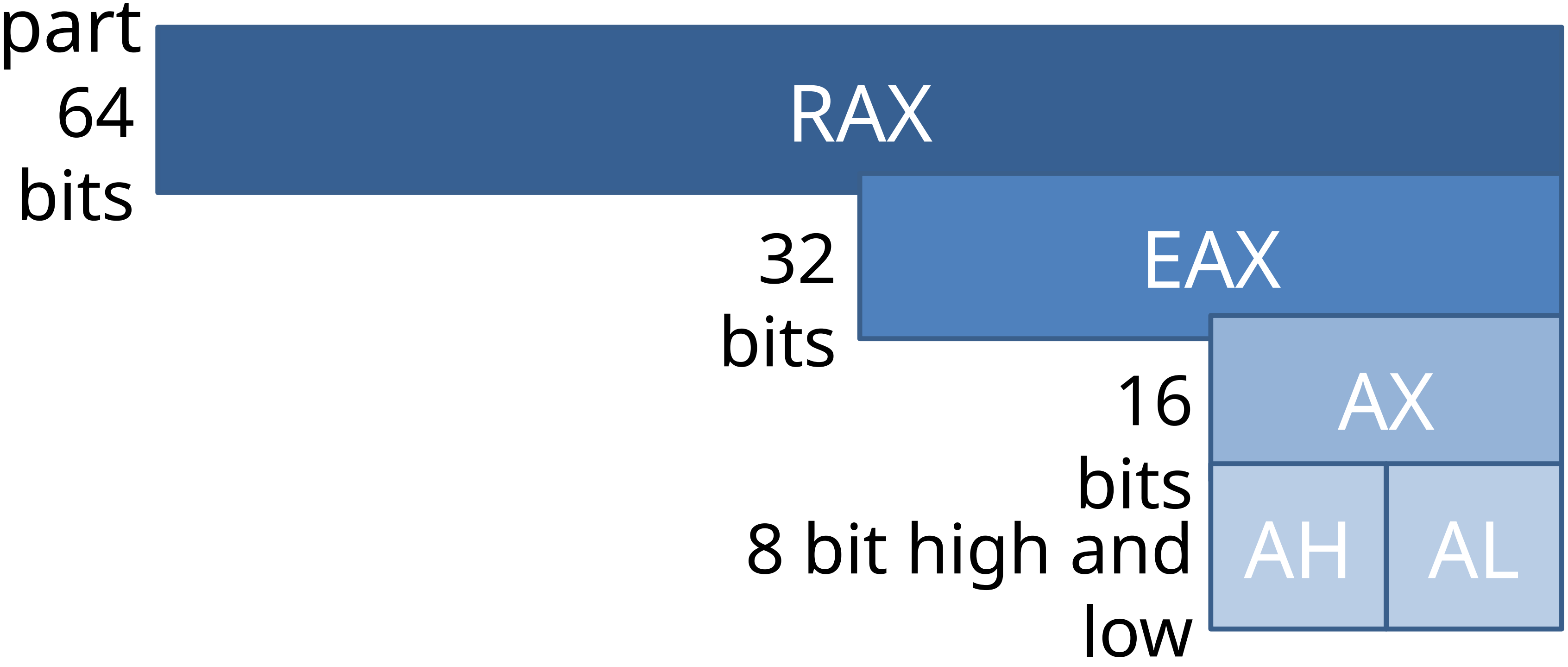
Larger / Slower

x86 Registers (32 bit)

- General purpose registers
  - EAX, EBX, ECX, EDX
  - Used for pretty much anything
- Stack registers
  - ESP: Points to the top of the stack
    - The stack grows down, so this is the lowest address
  - EBP: Points to the bottom of the current stack frame
    - Not strictly necessary, may be used as a general purpose register
- Other registers
  - EIP: Points to the currently executing instruction
  - EFLAGS: Bit vector of flags
    - Stores things like carry (after addition), equals zero (after a comparison), etc.

Register Layout

- x86 has gone through 16, 32, and 64 bit versions
- Registers can be addressed in whole or in part



64 bits

RAX

32 bits

EAX

16 bits

AX

8 bit high and low

AH

AL

x86 Assembly Examples

1) mov eax, 0x00ABCDE ; move 32-bit value to EAX
2) mov ax, 0xABCD ; move 16-bit value to AX
3) mov ah, 0xAB
4) mov al, 0xCD ; 3) + 4) are equivalent to 2)
5) add eax, ebx
6) mov edx, [esp] r in EDX

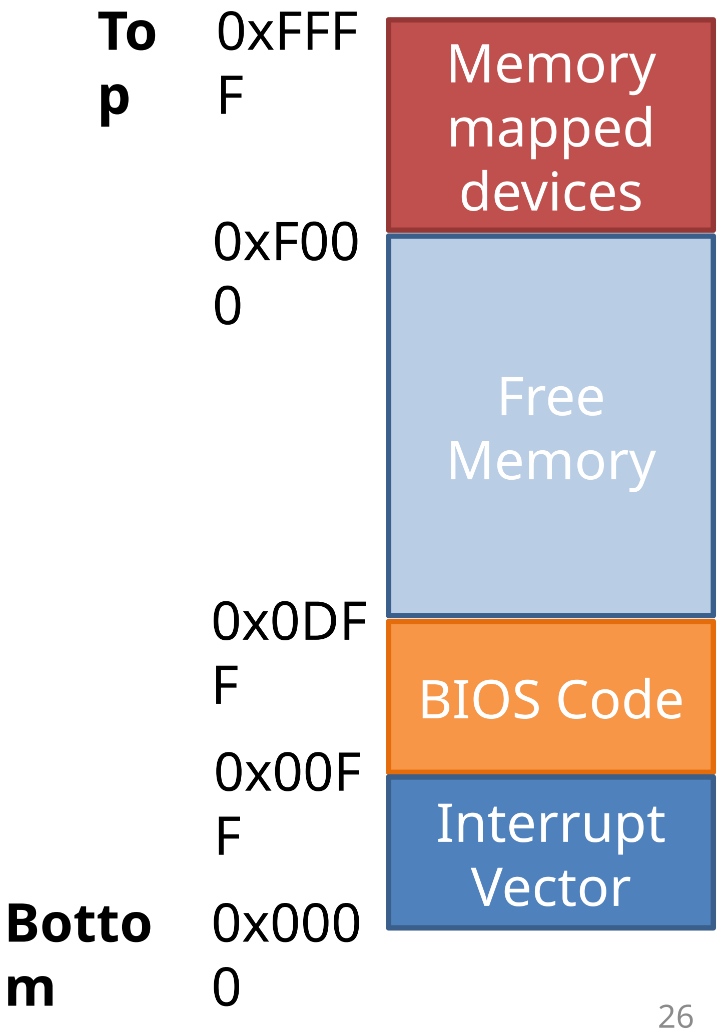
Important x86

In Intel assembly syntax, [] means dereference (e.g. *ptr in C)

	Instructions	Description	Exam
Basics	mov	Move data from one place to another	mov eax, 7 mov edx, [0xF0FF] mov [ebx], eax
	add / sub	Add / subtract values in registers	add eax, ebx
Func.	inc / dec	Increment / decrement the value in a register	inc eax
Stack	call	Push EIP onto the stack and jump to a function	call 0x80FE4C
Control Flow	ret	Pop the top of the stack into EIP	ret
	push / pop	Add or remove items to the stack	push eax
	int	Execute the given interrupt handler	int 0x80
	jmp	Load the given value into EIP	jmp 0x80FE4C

Typical Memory Layout

- Example: 64KB of memory
  - 2^{16} bytes
  - Addresses from 0 to 65535
- Not all memory is free
  - Specific ranges get used by devices, system services, the BIOS, etc.



Memory Addressing

- Memory can be addressed as:
  - Individual bytes
  - Multi-byte words, double words, quad words, etc.

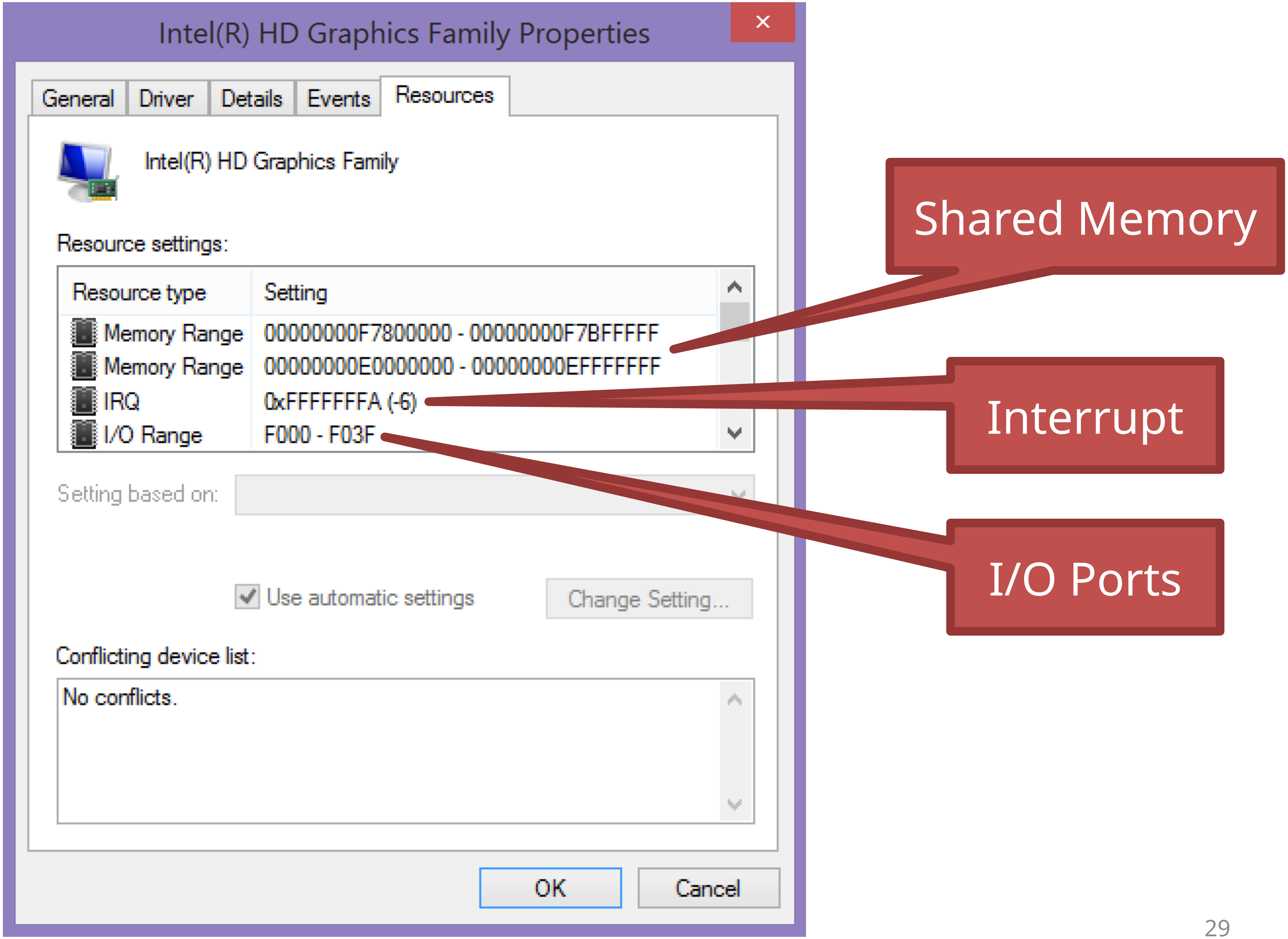
Address	Value
0xC7	0x00
0xC6	0x00
0xC5	0x10
0xC4	0x05
0xC3	0x00
0xC2	‘c’
0xC1	‘b’

```
mov ax, [0xC4]    ; AX = 0x0510
```

```
mov ah, [0xC0]    ; AH = ‘a’
```

Communicating with Devices

- CPU and devices execute concurrently
- How do CPU and devices communicate?
  1. I/O Ports
     - I/O-only memory space shared by the CPU and a device
  2. Memory mapping
     - Regions of RAM that are shared by the CPU and a device
  3. Interrupts
     - Signal from a device to the CPU
     - Interrupt causes the OS to switch to handler code



Intel(R) HD Graphics Family Properties

General | Driver | Details | Events | Resources

Intel(R) HD Graphics Family

Resource settings:

Resource type	Setting
Memory Range	00000000F7800000 - 00000000F7BFFFFF
Memory Range	00000000E0000000 - 00000000EFFFFFFF
IRQ	0xFFFFFFFA (-6)
I/O Range	F000 - F03F

Setting based on:

☑ Use automatic settings

Conflicting device list:

No conflicts.

Shared Memory

Interrupt

I/O Ports

I/O Ports

- Oldest method of interacting with devices
- CPU and devices share a virtual 16-bit memory space
  - Each device is assigned some portion of the address space
  - E.g. 0xF000 – 0xF03F
- CPU and device communicate by reading/writing to the virtual memory space

I/O Port Example

- Assume there is a serial port attached to port 0x3F8

Output

```
        mov eax, 1
        mov dx, 0x3F8
start:  out dx, al
        inc eax
        cmp eax, 32
        jne start
        …
```

Input

```
        mov ebx, 0
        mov dx, 0x3F8
start:  in dx, ax
        mov [esp + ebx], ax
        inc ebx
        cmp ebx, 31
        jle start
        …
```

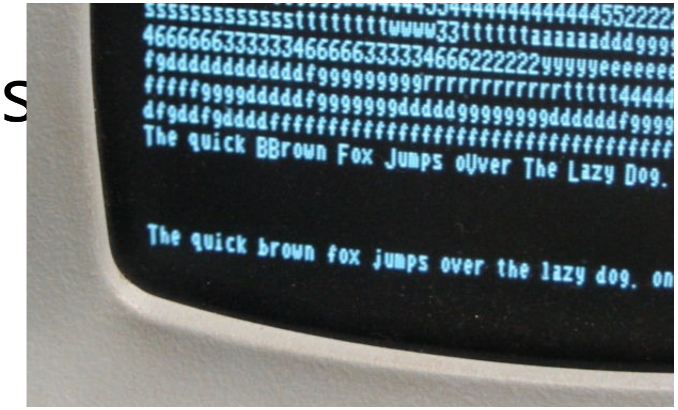
Problem With I/O Ports

- CPU must mediate all transfers
  - Suppose you want to move data from disk to memory…
  - CPU must copy each 16-bit value from the I/O port to memory
- All I/O must be synchronous
  - Suppose the disk wants to send data…
  - … but the CPU isn’t reading the I/O port with in
- Bottom line: I/O ports are **slow**

Memory Mapped Devices

- RAM shared by the CPU and a device
- Example: Console frame buffer
  - Address range of 1920 bytes
  - Corresponds to 24 lines of text, 80 characters wide
  - CPU writes characters into the memory range…
  - dware dis

Address	Value
0xC2	‘e’
0xC1	‘h’
0xC0	‘T’


The quick BBrown Fox Jumps oUver The Lazy Dog.
The quick brown fox jumps over the lazy dog.

Example Keyboard Controller

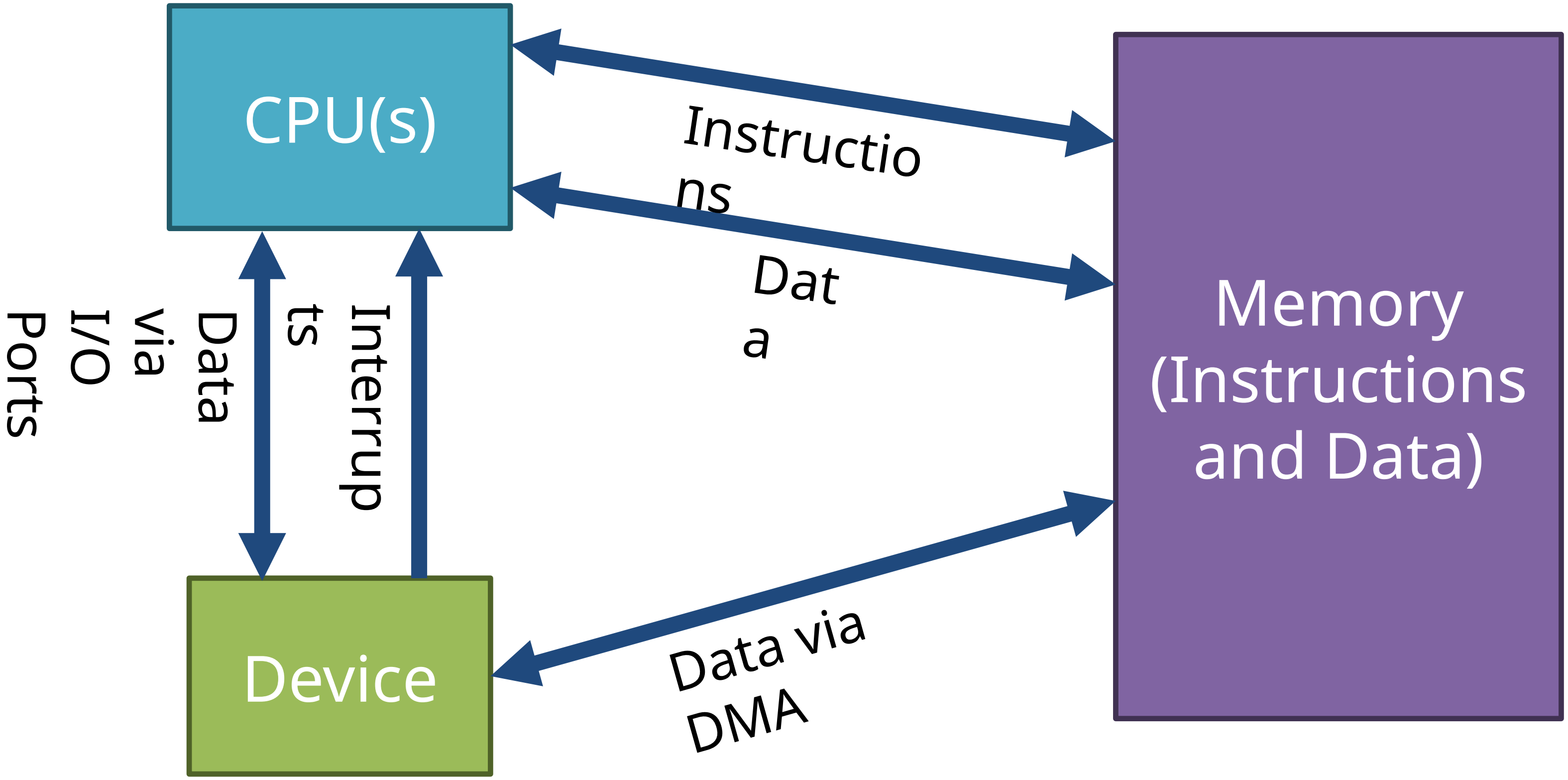
- Simple, memory mapped keyboard interface
  - – 2 bytes: keycode byte and status byte
- When a key is pressed:
  - – The keycode byte is set to the character
  - – The status byte is set to 1
- When the key is read:
  - – CPU sets the status byte to 0

Address	Purpos e
0xF801	keycode
0xF800	status

Direct Memory Access (DMA)

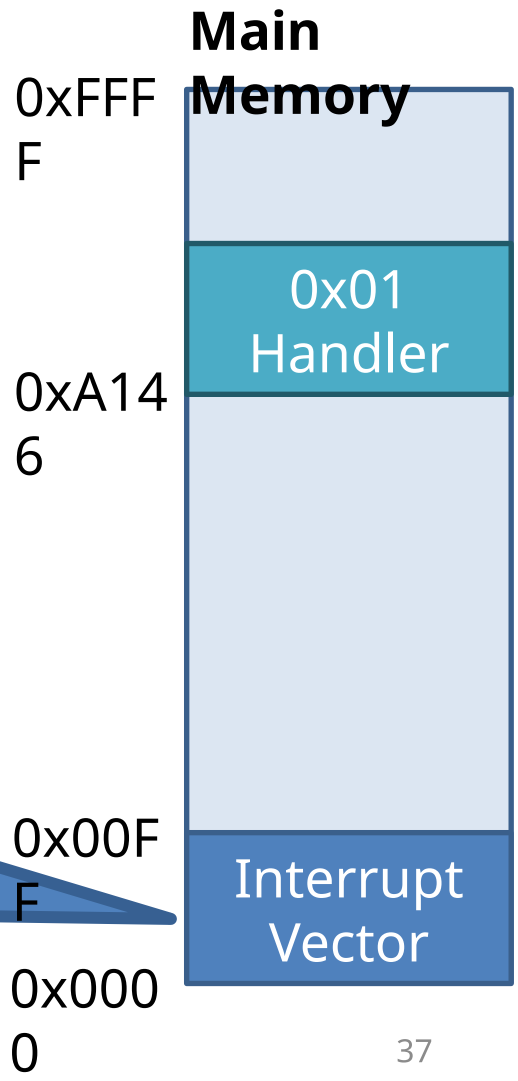
- Enables devices to transfer blocks of data directly to memory
  - Interrupt generated when transfer is completed
- Much faster than the alternative method
  - Interrupt generated after each byte is transferred

DMA in Action



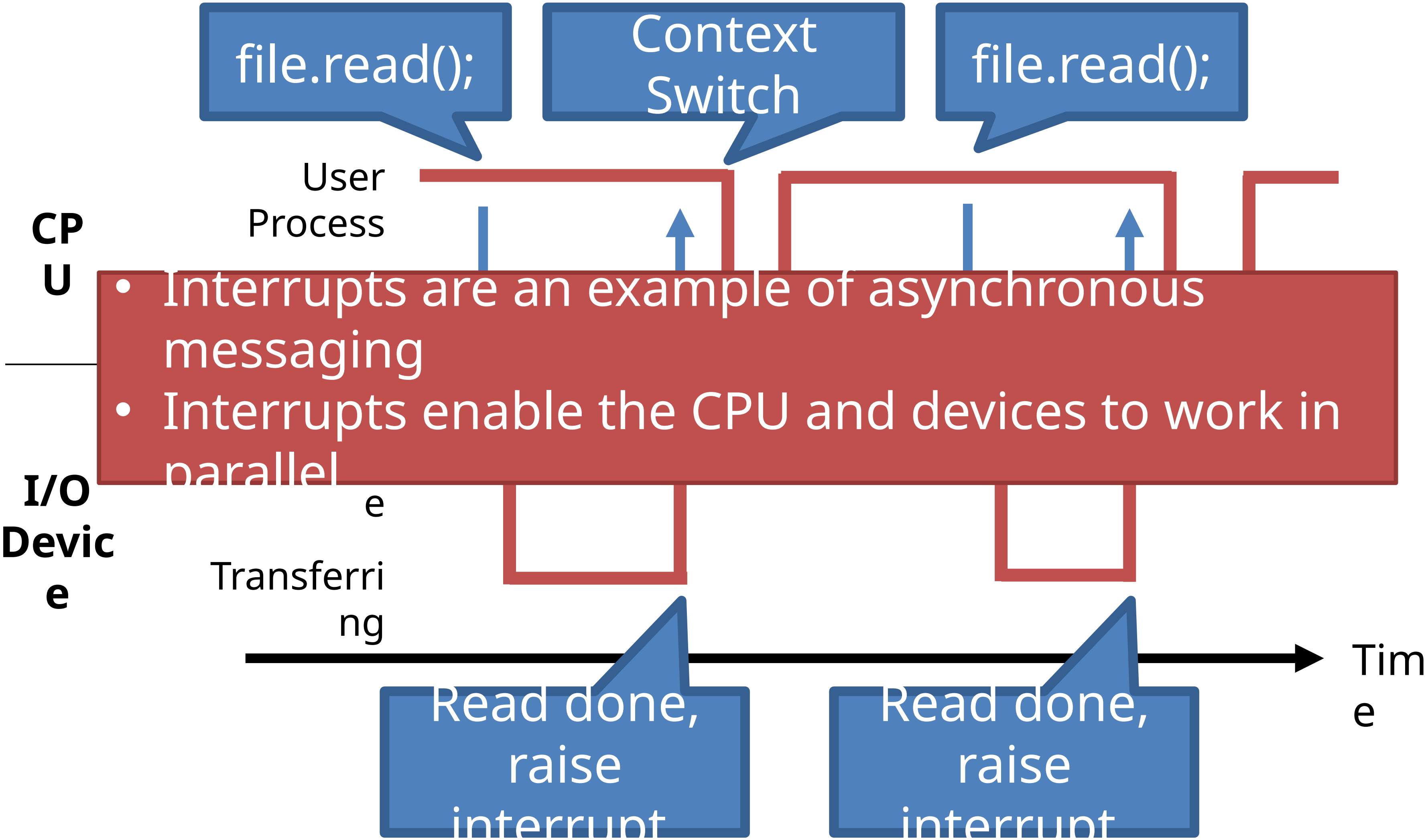
Interrupts

- Interrupts transfer control from the running code to an interrupt handler
  - Each interrupt is assigned a number
  - Number acts as an index into the Interrupt Vector Table
  - Table maps interrupt numbers to handlers
- Interrupts cause a context switch
  - State of the CPU must be saved before the switch…
  - … and restored after the handler completes



Interrupt Number	Pointer to Handler
0x02	0x0000
0x01	0xA146

Interrupt Timeline



- Interrupts are an example of asynchronous messaging
- Interrupts enable the CPU and devices to work in parallel

Today’s Servers/Desktops

- Much greater homogeneity/compatibility
  - Apple and PC use the same internals
  - Powerful industry groups ratify strict standards for hardware compatibility
    - Joint Electron Device Engineering Council (JEDEC)
    - PCI Special Interest Group (PCI SIG)
    - USB Implementors Forum (USB-IF)
- No longer IBM-PC compatible
  - x86-64
  - Unified Extensible Firmware Interface (UEFI) instead of Basic Input/Output System (BIOS)

- Hardware Basics
- PC Bootup Sequence
- A Simple OS Example
- Kernels

What Happens After You Push Power?

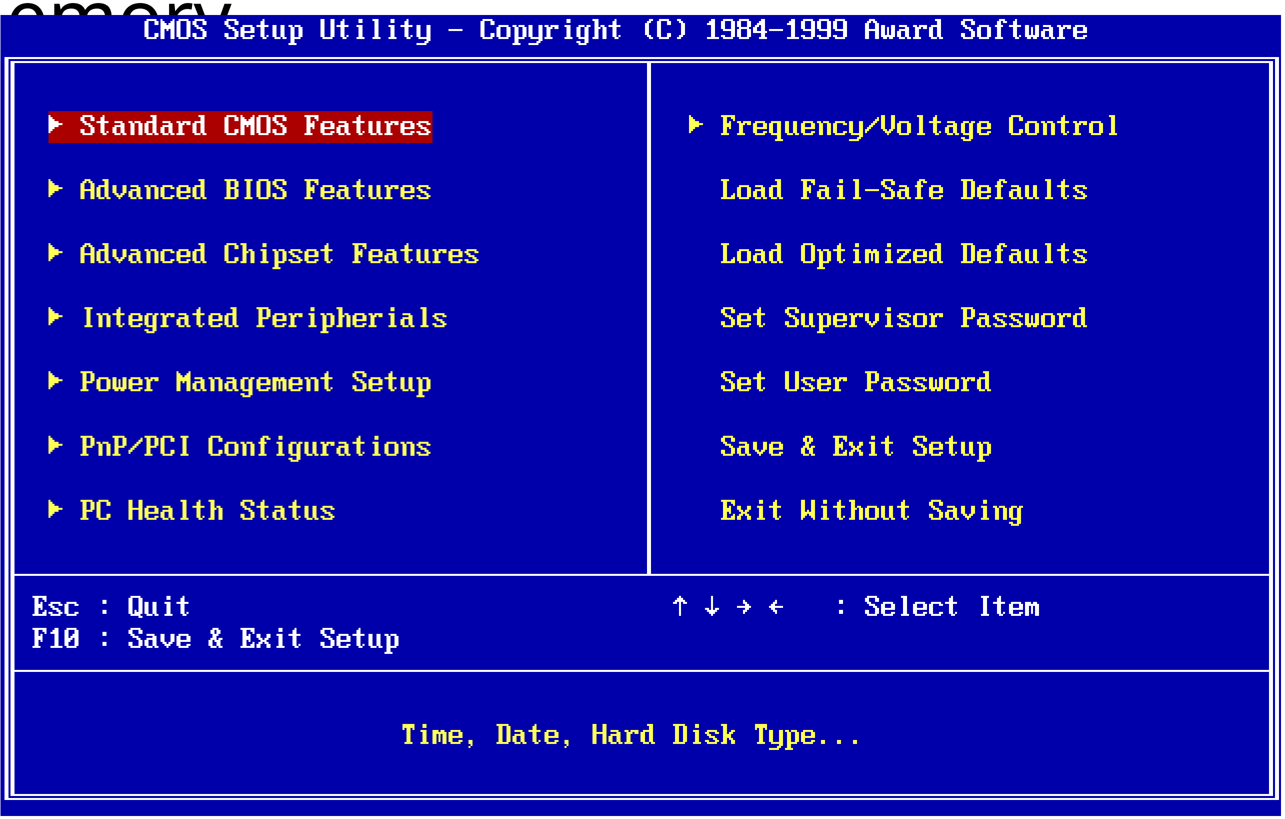
- A lot happens in between
  - Pushing power…
  - … And arriving at the desktop
- Basic steps
  1. Start the BIOS
  2. Load settings from CMOS
  3. Initializing devices
  4. Run POST
  5. Initiate the bootstrap sequence

Starting the BIOS

- Basic Input/Output System (BIOS)
  - A mini-OS burned onto a chip
- Begins executing a soon as a PC powers on
  - Code from the BIOS chip gets copied to RAM at a low address (e.g. 0xFF)
  - jmp 0xFF (16 bits) written to RAM at 0xFFFF0 (2^{20}-16)
  - x86 CPUs always start with 0xFFFF0 in the EIP register
- Essential goals of the BIOS
  - Check hardware to make sure its functional
  - Install simple, low-level device drivers
  - Scan storage media for a Master Boot Record (MBR)
    - Load the boot record into RAM
    - Tells the CPU to execute the loaded code

Loading Settings from CMOS

- BIOS often has configurable options
  - Values stored in battery-backed CMOS memory



```
        CMOS Setup Utility - Copyright (C) 1984-1999 Award Software

  ▶ Standard CMOS Features              ▶ Frequency/Voltage Control
  ▶ Advanced BIOS Features                Load Fail-Safe Defaults
  ▶ Advanced Chipset Features             Load Optimized Defaults
  ▶ Integrated Peripherials               Set Supervisor Password
  ▶ Power Management Setup                Set User Password
  ▶ PnP/PCI Configurations                Save & Exit Setup
  ▶ PC Health Status                      Exit Without Saving

 Esc : Quit                           ↑ ↓ → ←   : Select Item
 F10 : Save & Exit Setup

                      Time, Date, Hard Disk Type...
```

Initialize Devices

- Scans and initializes hardware
  - CPU and memory
  - Keyboard and mouse
  - Video
  - Bootable storage devices
- Installs interrupt handlers in memory
  - Builds the Interrupt Vector Table
- Runs additional BIOSes on expansion cards
  - Video cards and SCSI cards often have their own BIOS
- Runs POST test
  - Check RAM by read/write to each address

Bootstrapping

- Problem: we need to find and load a real OS
- BIOS identifies all potentially bootable devices
  - Tries to locate Master Boot Record (MBR) on each device
  - Order in which devices are tried is configurable
- MBR has code that can load the actual OS
  - Code is known as a bootloader
- Example bootable devices:
  - Hard drive, SSD, floppy disk, CD/DVD/Bluray, USB flash drive, network interface card

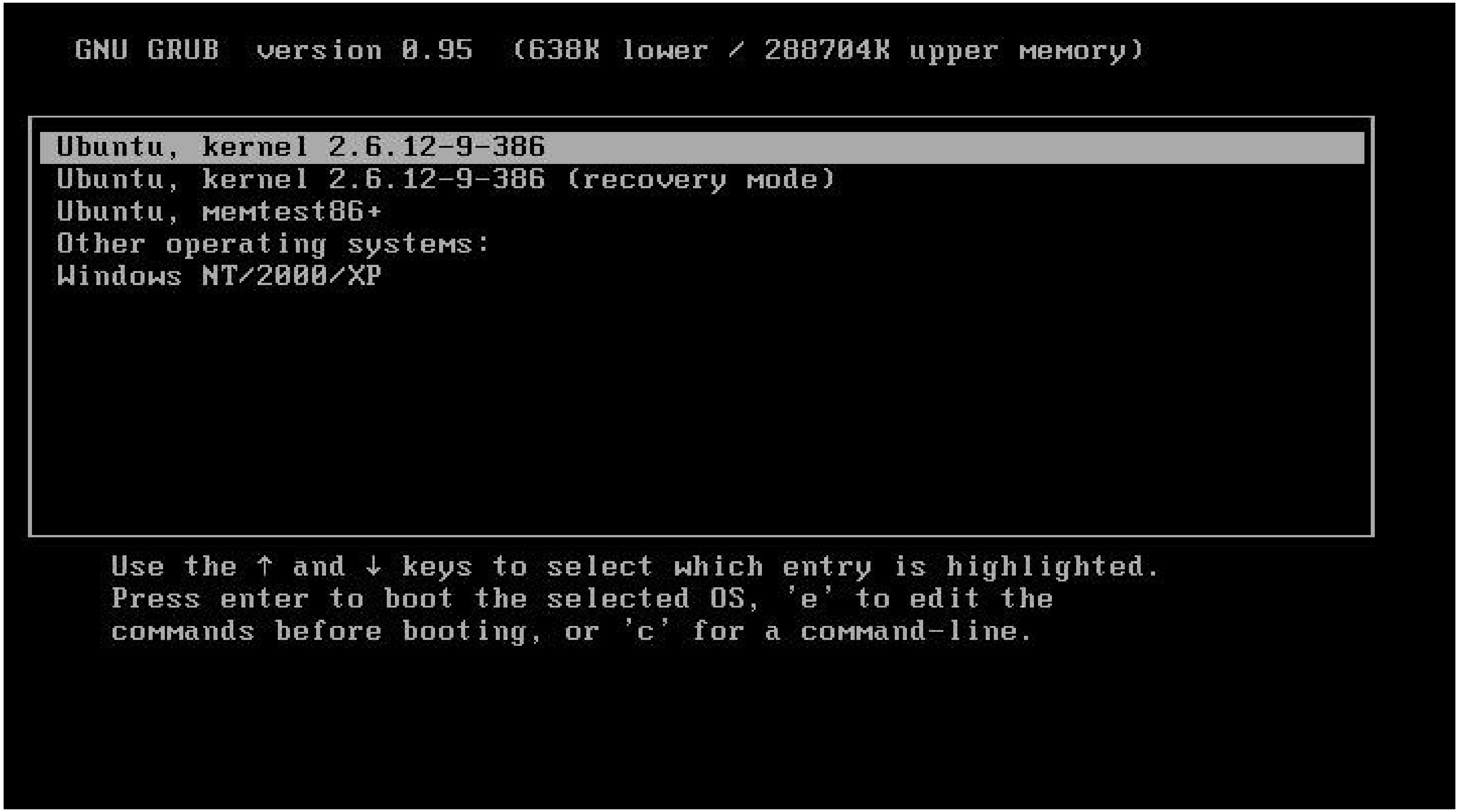
The Master Boot Record (MBR)

- Special 512 byte file written to sector 1 (address 0) of a storage device
- Contains
  - 446 bytes of executable code
  - Entries for 4 partitions
- Too small to hold an entire OS
  - Starts a sequence of chain-loading

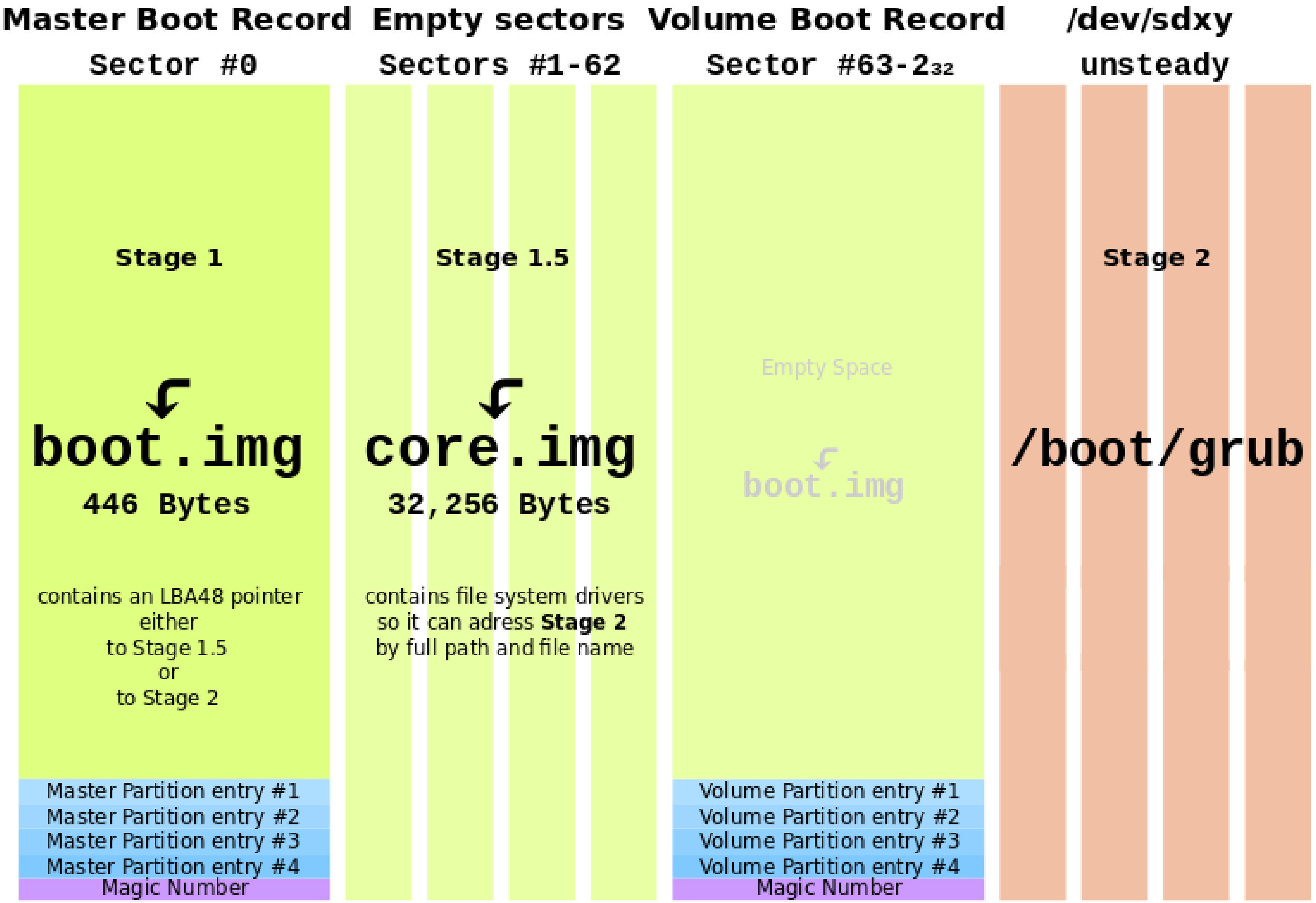
Address		Description	Size (Bytes)
Hex	Dec.		
0x000	0	Bootstrap code area	446
0x1BE	446	Partition Entry #1	16
0x1CE	462	Partition Entry #2	16
0x1DE	478	Partition Entry #3	16
0x1EE	494	Partition Entry #4	16
0x1FE	510	Magic Number	2

Example Bootloader: GRUB

- Grand Unified Bootloader
  - Used with Unix, Linux, Solaris, etc.



```
  GNU GRUB  version 0.95  (638K lower / 288704K upper memory)

 Ubuntu, kernel 2.6.12-9-386
 Ubuntu, kernel 2.6.12-9-386 (recovery mode)
 Ubuntu, memtest86+
 Other operating systems:
 Windows NT/2000/XP

    Use the ↑ and ↓ keys to select which entry is highlighted.
    Press enter to boot the selected OS, 'e' to edit the
    commands before booting, or 'c' for a command-line.
```



Master Boot Record
Sector #0

Stage 1

boot.img
446 Bytes

contains an LBA48 pointer
either
to Stage 1.5
or
to Stage 2

Master Partition entry #1
Master Partition entry #2
Master Partition entry #3
Master Partition entry #4
Magic Number

Empty sectors
Sectors #1-62

Stage 1.5

core.img
32,256 Bytes

contains file system drivers
so it can adress **Stage 2**
by full path and file name

Volume Boot Record
Sector #63-2_{32}

Empty Space

boot.img

Volume Partition entry #1
Volume Partition entry #2
Volume Partition entry #3
Volume Partition entry #4
Magic Number

/dev/sdxy
unsteady

Stage 2

/boot/grub

Each **partition table entry** comprises of **16 octets**:

Flag	Start CHS	Type	End CHS	Start LBA	**Size**
1	3	1	3	4	**4 octets**

- Hardware Basics
- PC Bootup Sequence
- A Simple OS Example
- Kernels

Status Check

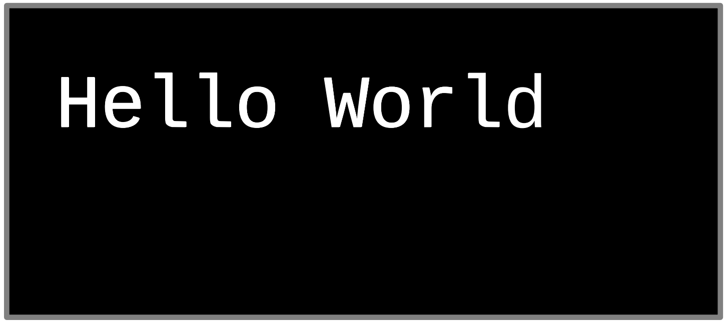
- At this point we understand:
  - The basic building blocks of device I/O
  - How memory is laid out
  - Basic x86 instructions
  - How the BIOS locates and executes a bootloader
- What’s next?
  - Let’s build a tiny OS for the bootloader to load

Goals of Our Simple OS

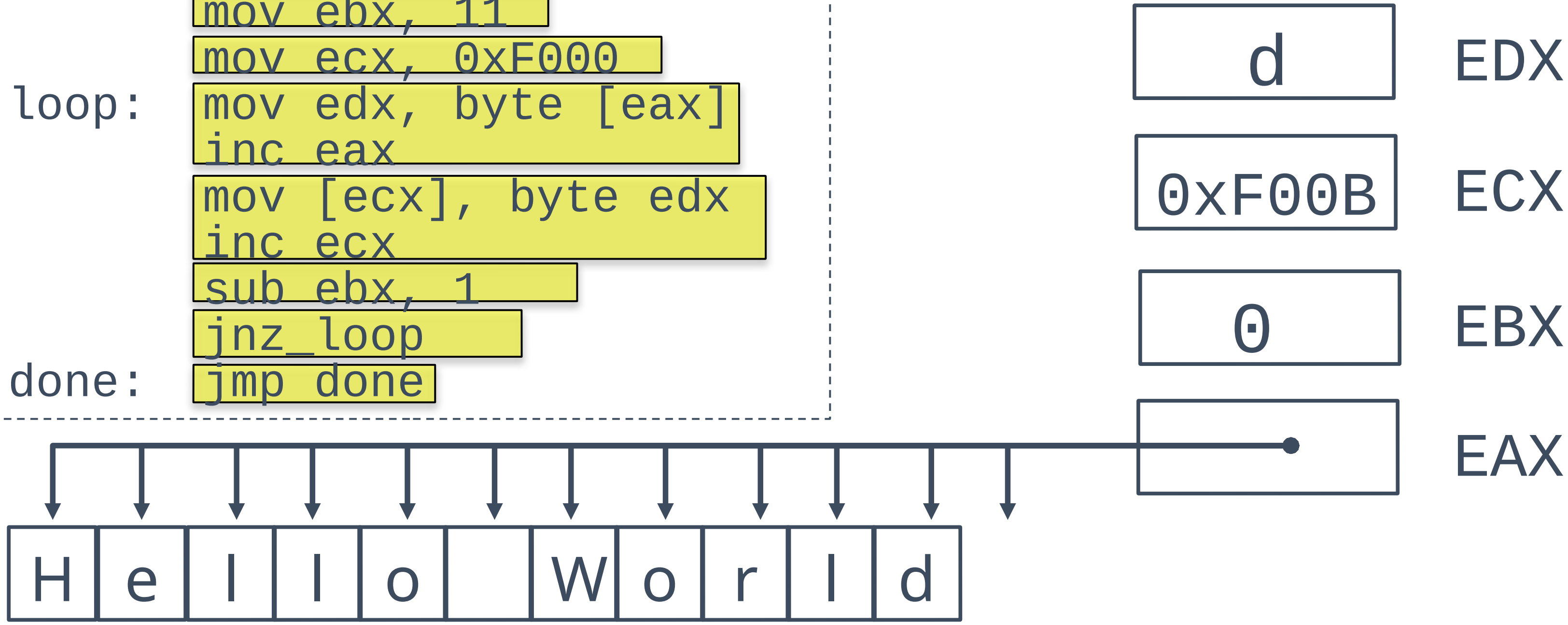
- APIs for device access
  - Read from the keyboard
  - Read and write to a simple disk
  - Display text to the screen
- Ability to run a simple user program
- A basic command line for running programs

Sample Program 1 – Hello World

- Writes “Hello World” to the frame buffer
- Then loops forever



```
fbuf = 0xF000
str:   'Hello World'

begin: mov eax, str
       mov ebx, 11
       mov ecx, 0xF000
loop:  mov edx, byte [eax]
       inc eax
       mov [ecx], byte edx
       inc ecx
       sub ebx, 1
       jnz loop
done:  jmp done
```



Sample Program 1 – Hello World

- In this program, there is no OS
  - Program interacts directly with hardware
- This approach might be used for highly-constrained, low-cost environments
  - Example: simple embedded devices
- In a system like this, the program is usually written into read-only-memory (ROM) at the factory

Sample Program 2 – Keyboard to Screen

- Reads input from the keyboard and writes it to the frame buffer

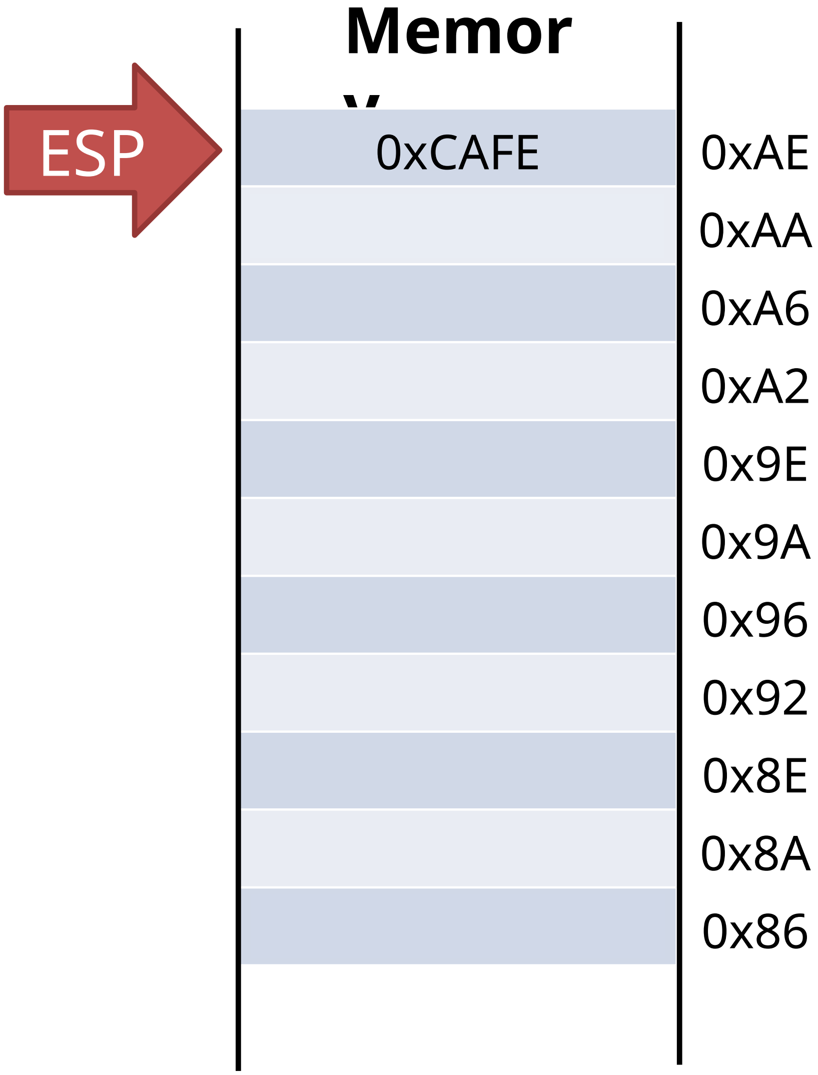
```
framebuf = 0xF000
status = 0xF800
keycode = 0xF801

begin: mov eax, 0xF000
loop:  mov ebx, byte [0xF800]
       cmp ebx, 0
       jz  loop

       mov ebx, byte [0xF801]
       mov [eax], byte ebx
       inc eax
        mov [0xF800], 0
       jmp loop
```

- How can we turn this functionality into an API?

The x86 Stack

- x86 CPU uses ESP register to implement a push down stack
- Examples:

```
push 0x01
pop eax       ; EAX = 1
```



Memor
y

ESP

0xCAFE	0xAE
	0xAA
	0xA6
	0xA2
	0x9E
	0x9A
	0x96
	0x92
	0x8E
	0x8A
	0x86

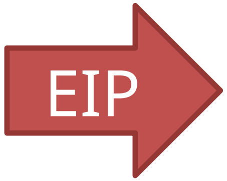
Function Calls and the Stack

- The stack is used to implement function calls
  - `call addr`: call a function
    - Calculates a return address (the address of the instruction following call)
    - Pushes the return address on to the stack
    - Jumps to `addr (EIP = addr)`
  - `ret`: return from a function
    - Pops the return address from the stack
    - Jumps to the return address

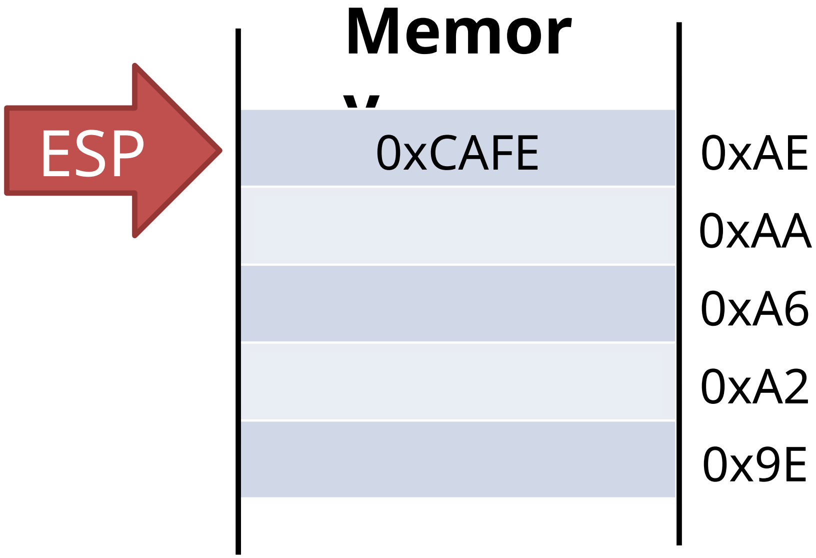
Function Call Example

- Suppose we have code that calls

i = add(1, 2);



```
              ...
10.           push 2
11.           push 1
12.           call add
13.           mov ecx, eax
              ...
20.     add:        mov eax, [esp + 4]
21.           mov ebx, [esp + 8]
22.           add eax, ebx
23.           ret
```



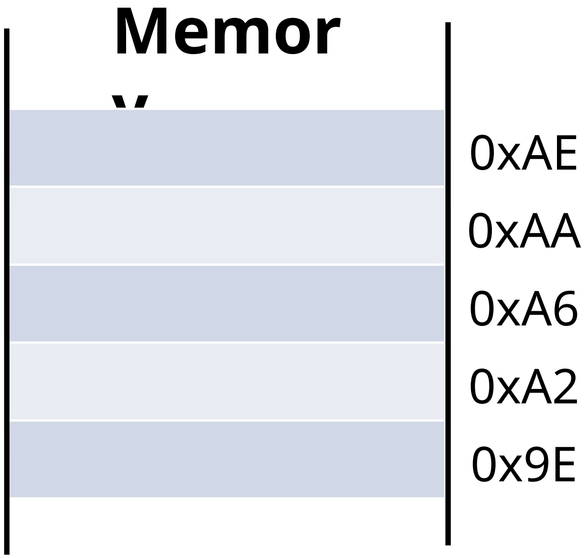
Argument Ordering and Return Values

- Function arguments are always pushed in reverse order
  - Why?
  - To support functions with a variable number of arguments
  - Example: printf("%i %f %s", a, pi, str);
    - Argument 1 tells you how many more arguments there are on the stack
- By convention, return values are always placed in EAX
  - This is why (typical) functions may only return one value

Pop Quiz

- What does the stack look like after calling **f(7, 10)**?

```
            …
27.         call f
            …
59.   f:    mov eax, [esp + 4]
            …
```


Memory

0xAE

0xAA

0xA6

0xA2

0x9E

A Minimal Operating System

- We’ll now write a simple OS that can:
  - Read keyboard input
  - Write it to the frame buffer
- *getkey()*
  - Loops until a key has been pressed
  - Loads the key into EAX

```
getkey:  mov edx, byte [status]
         cmp edx, 0
         jz  getkey
         mov [status], 0
         mov eax, byte [keycode]
         ret
```

putchar()

- *putchar()*
  - Writes the 2 byte function argument to the frame buffer
  - Maintains the frame buffer cursor

```
bufptr: variable holding a pointer to the frame buffer

putchar: mov eax, word [esp+ 4]
         mov ebx, dword [bufptr]
         mov [ebx], word eax
         add ebx, 1
         mov [bufptr], ebx
         ret
```

Using *getkey()* and *putchar()*

- We can now rewrite Sample Program 2 using our simple OS

Old Code

```
framebuf = 0xF000
status = 0xF800
keycode = 0xF801

begin: mov eax, 0xF000
loop:  mov ebx, byte [0xF800]
       cmp ebx, 0
       jz  loop

       mov ebx, byte [0xF801]
       mov [eax], byte ebx
       inc eax
       jmp loop
```

New Code

```
loop:     call getkey
          push eax
          call putchar
          pop  eax
          jmp  loop
```

Why Do We Care?

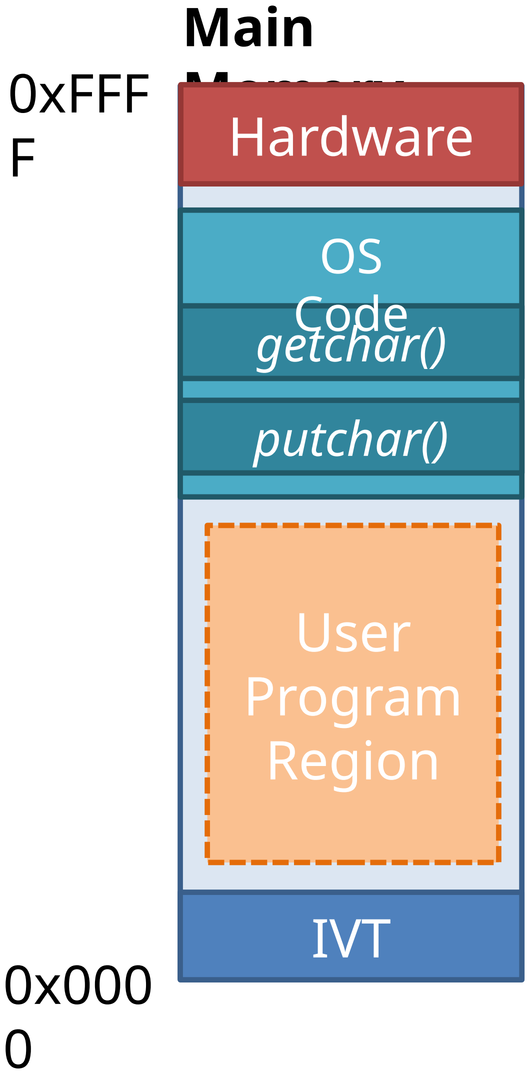
- Reusability
  - Many programs can use *getchar()* and *putchar()*
- Abstraction
  - Programs don’t need to know details of the keyboard and frame buffer interfaces
- Portability
  - Program could run on another OS that supports *getchar()* and *putchar()*…
  - … even if the hardware interface has changes

Let’s Build a Basic Shell

- Almost all OSes include a *shell*
  - A program that takes commands from the user
  - Earliest (and best) shells were command lines
  - Modern shells are GUIs
- Let’s build a shell into our OS
  1. Read a command line from the keyboard
  2. Read the associated program from the disk
  3. Load the program into memory and execute it

Basic Program Loading

- Memory regions are reserved
  - Memory mapped hardware
  - OS code
  - IVT
  - Etc…
- To load and run a program:
  - Read the program from disk into the program region of memory
  - Use call to jump to the first instruction of the program
    - Called the entry point



Example Disk Controller

- Disk drive interface
  - Reads and writes occur in 512-byte blocks
  - Block numbers start at 0
- To write to block *B*
  - Copy the data into range 0xF900 – 0xFAFF
  - Write *B* to 0xF822
  - Write 'W' to 0xF820
    - Tells the drive to write the buffer
  - Drive will write 0 to 0xF820 when the transfer is complete

Address	Purpose
0xFAFF	512-byte buffer
…	
0xF900	
0xF822	block address
0xF820	cmd/status

Example Disk Controller (cont.)

- To read from block *B*
  - Write *B* to 0xF822
  - Write ‘R’ to 0xF820
    - Tells the drive to read data from the disk into the buffer
  - Drive will write 0 to 0xF820 when the transfer is complete
  - Data from *B* is now available in 0xF900 – 0xFAFF

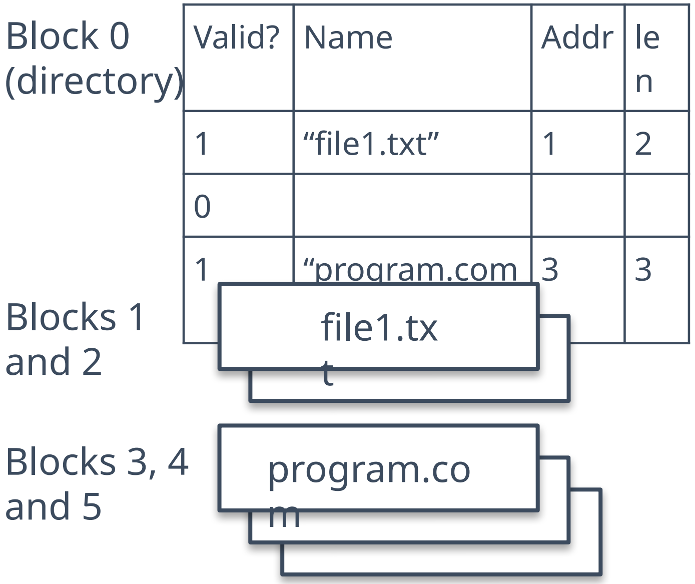
Address	Purpose
0xFAFF	512-byte buffer
…	
0xF900	
0xF822	block address
0xF820	cmd/status

Additional OS APIs

- Assume we have already implemented some functions
  - *read_disk_block()* reads a block from the disk to some address in memory
  - *getline()* reads a line from the keyboard and stores it into a buffer

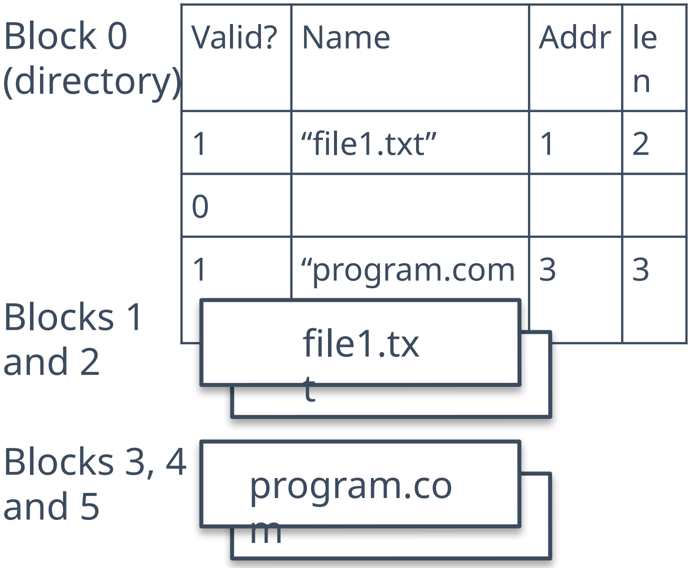
Basic File System

- Assume the disk is divided into blocks
- We introduce a trivial file system
  - Block 0 is the directory mapping of the file system
  - Other blocks are program data blocks



Valid?	Name	Addr	len
1	“file1.txt”	1	2
0			
1	“program.com	3	3

File System Structures



Block 0 (directory)

Valid?	Name	Addr	len
1	“file1.txt”	1	2
0			
1	“program.com	3	3

Blocks 1 and 2

Blocks 3, 4 and 5

```
struct dirent {
        bool valid;
        char name[16];
        int start;
        int len;
};

struct dirent directory[BLK_SIZ/sizeof(struct dirent)]

read_disk_block(/*blk #*/ 0, /*destination*/ directory);
```

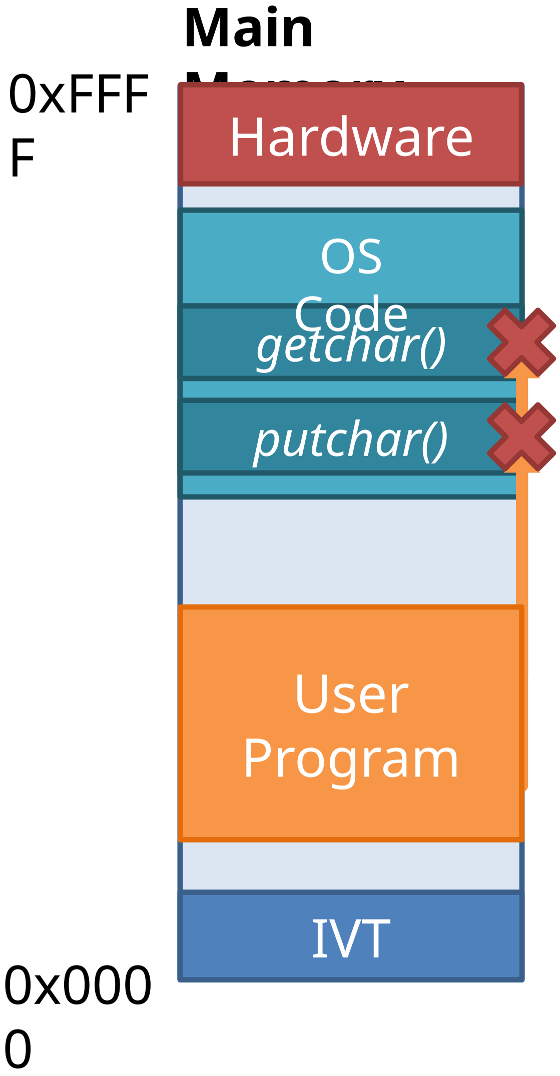
Basic Command Line

- We can now write a very simple command line:

```
char buffer[80];
struct dirent directory[NDIR];
int i, start, count;
void *program_base = …; /* probably 0x100 or so */
while (1) {
  getline(buffer);
  read_disk_block(DIR_SECTOR, directory);
  for (i = 0; i < NDIR; i++) {
    if (strcmp(buffer,directory[i].name)== 0 && directory[i].valid) {
      for (j = 0; j < directory[i].len; j++)
        read_disk_block(/* block # */ directory[i].start + j,
                        /* destination*/ program_base + j * 512);
      break;
    }
  }
  asm(“call program_base”);
  /* returns here when program is done */
}
```

Limitations of Our OS

- Crashes if it can’t find the specified program
- If the program crashes, the whole OS needs to be restarted
- Programs may not run if the OS is upgraded
  - Why?
  - Programs access OS APIs directly
  - What if the addresses change?



System Call Interface

- System call table
  - Layer of indirection to abstract the OS APIs
  - Table is always located at a fixed position
  - Each OS API is given a specific index in the table
- Programs access APIs via the table, instead of hard coding the function address

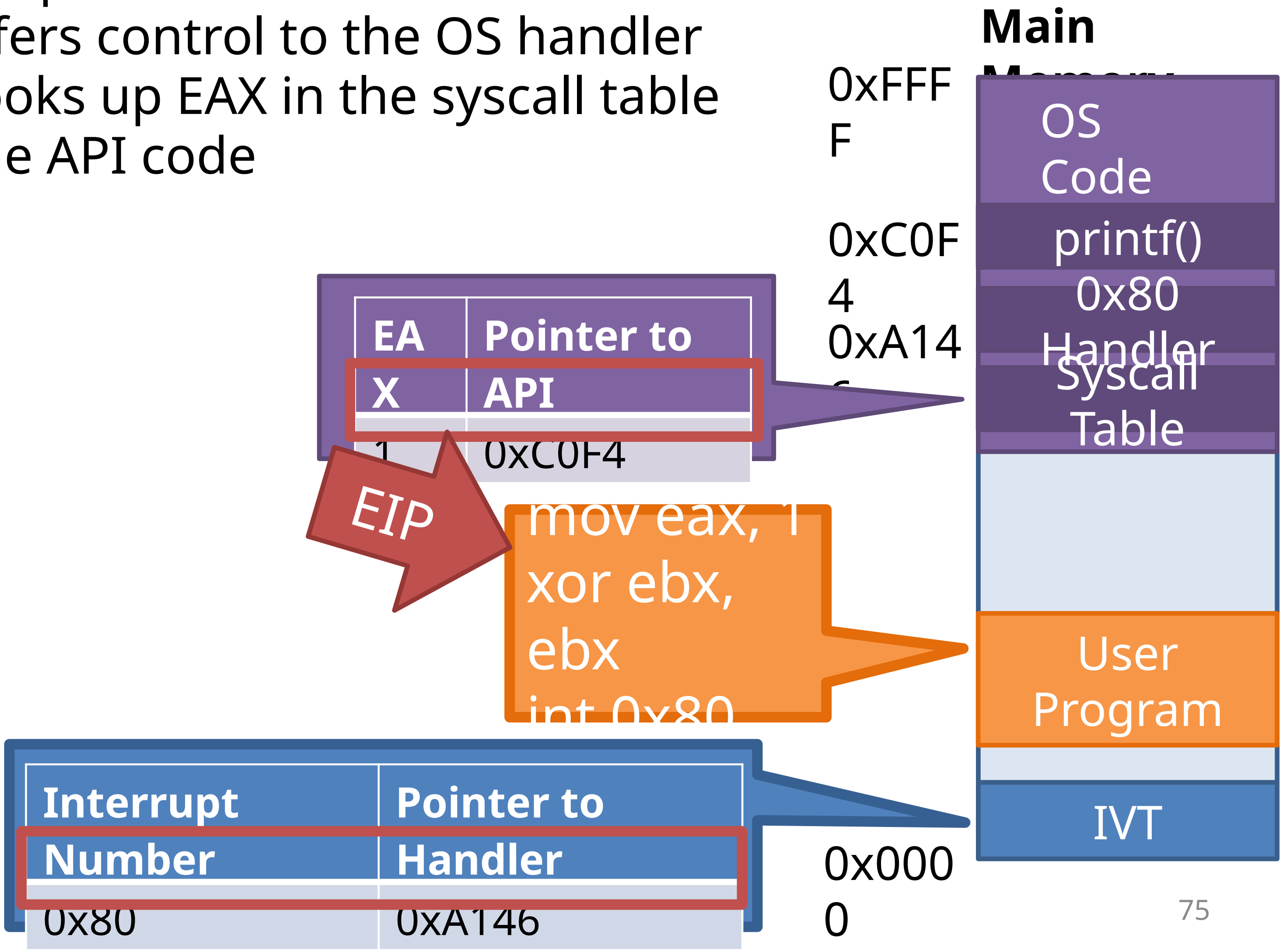
Traps: Software Interrupts

- Software can also generate interrupts
  - Traps and exceptions are software interrupts
- Example:

```
mov eax, 1                                          t()
= 1
xor ebx, ebx ; exıt() takes one parameter
int 0x80     ; transfer control to the Linux kernel
```

(Simplified) System Call Example

1. Software executes int 0x80, pushes EIP
2. CPU looks up handler in the IVT
3. CPU transfers control to the OS handler
4. Handler looks up EAX in the syscall table
5. Jump to the API code



EAX	Pointer to API
1	0xC0F4

```
mov eax, 1
xor ebx, ebx
int 0x80
```

Interrupt Number	Pointer to Handler
0x80	0xA146

CPU Support for System Calls

- Many CPUs provide instruction support for invoking system calls
  - On x86, system calls are initiated via an interrupt
- Example: Linux system calls
  - On x86, int 0x80 is the system call interrupt
  - EAX holds the table index of the desired API

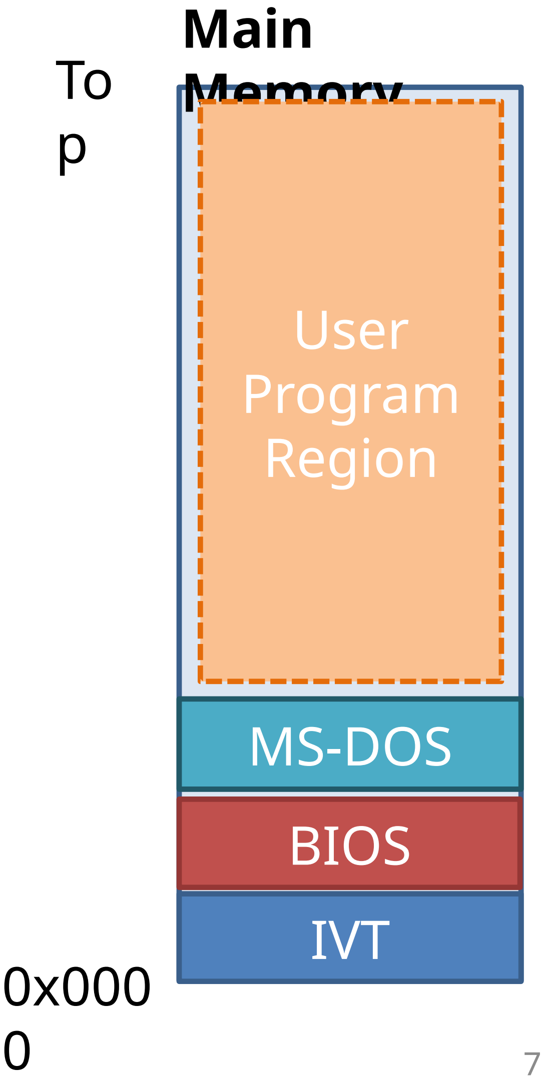
EAX	Function
1	sys_exit
2	sys_fork
3	sys_read
4	sys_write

Our OS So Far

- Almost at the level of MS-DOS 1.0
- Current features:
  - Interface to frame buffer
  - Interface to keyboard controller
  - Interface to disk controller
  - Command line
  - Ability to load and execute simple programs
  - Uses interrupts to make system calls

Similarities with MS-DOS 1.0

- Separate OS and program spaces
- System call table accessed via interrupt
- Command line is part of OS
- Similar keyboard controller, frame buffer, and disk controller



- Hardware Basics
- PC Bootup Sequence
- A Simple OS Example
- Kernels

Towards a Kernel

- “The one program running at all times on the computer” is the **kernel**
- Typically loaded by the bootloader
- Questions:
  - What are the features that kernels should implement?
  - How should we architect the kernel to support these features?

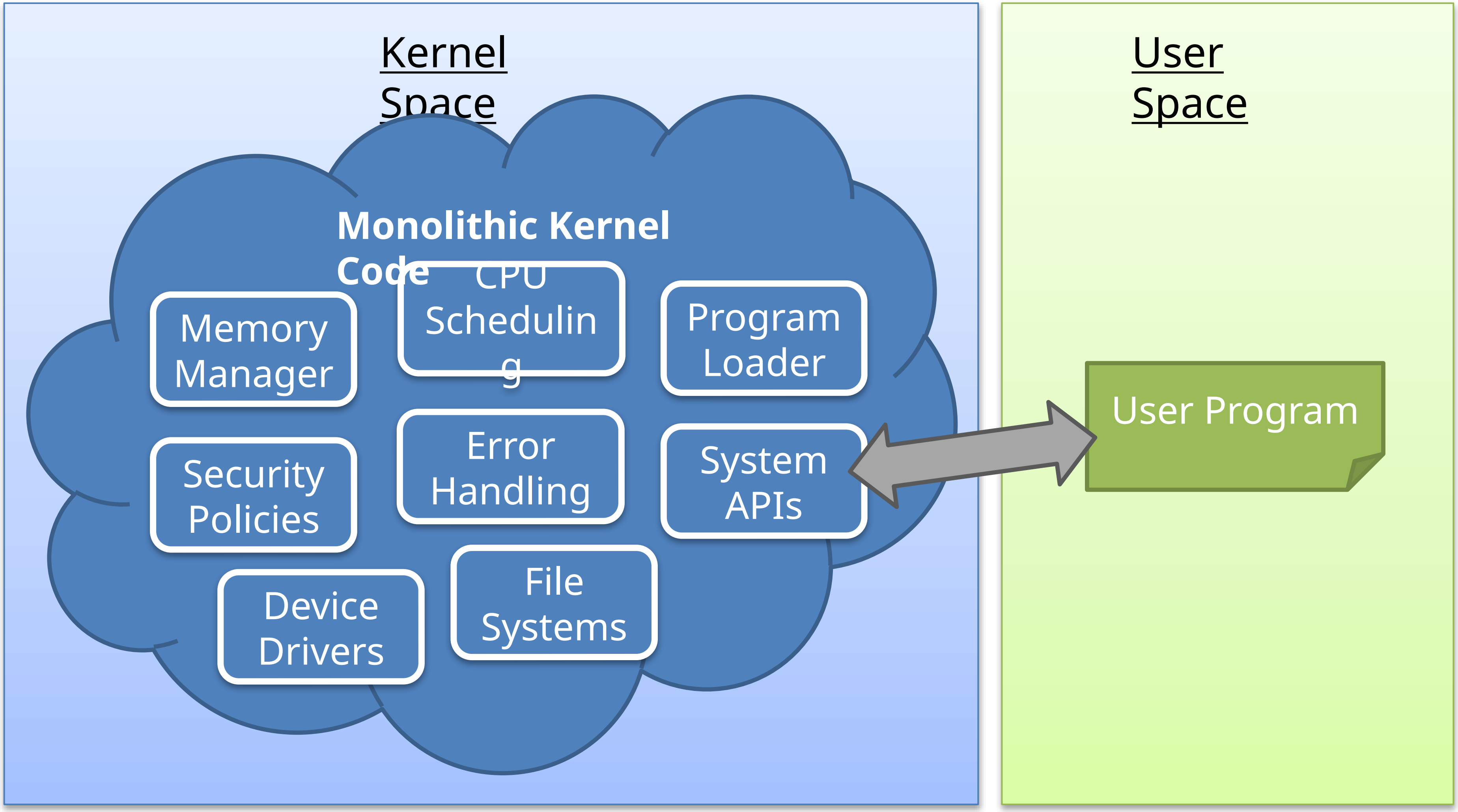
Kernel Features

- Device management
  - Required: CPU and memory
  - Optional: disks, keyboards, mice, video, etc.
- Loading and executing programs
- System calls and APIs
- Protection and fault tolerance
  - E.g. a program crash shouldn't crash the computer
- Security
  - E.g. only authorized users should be able to login

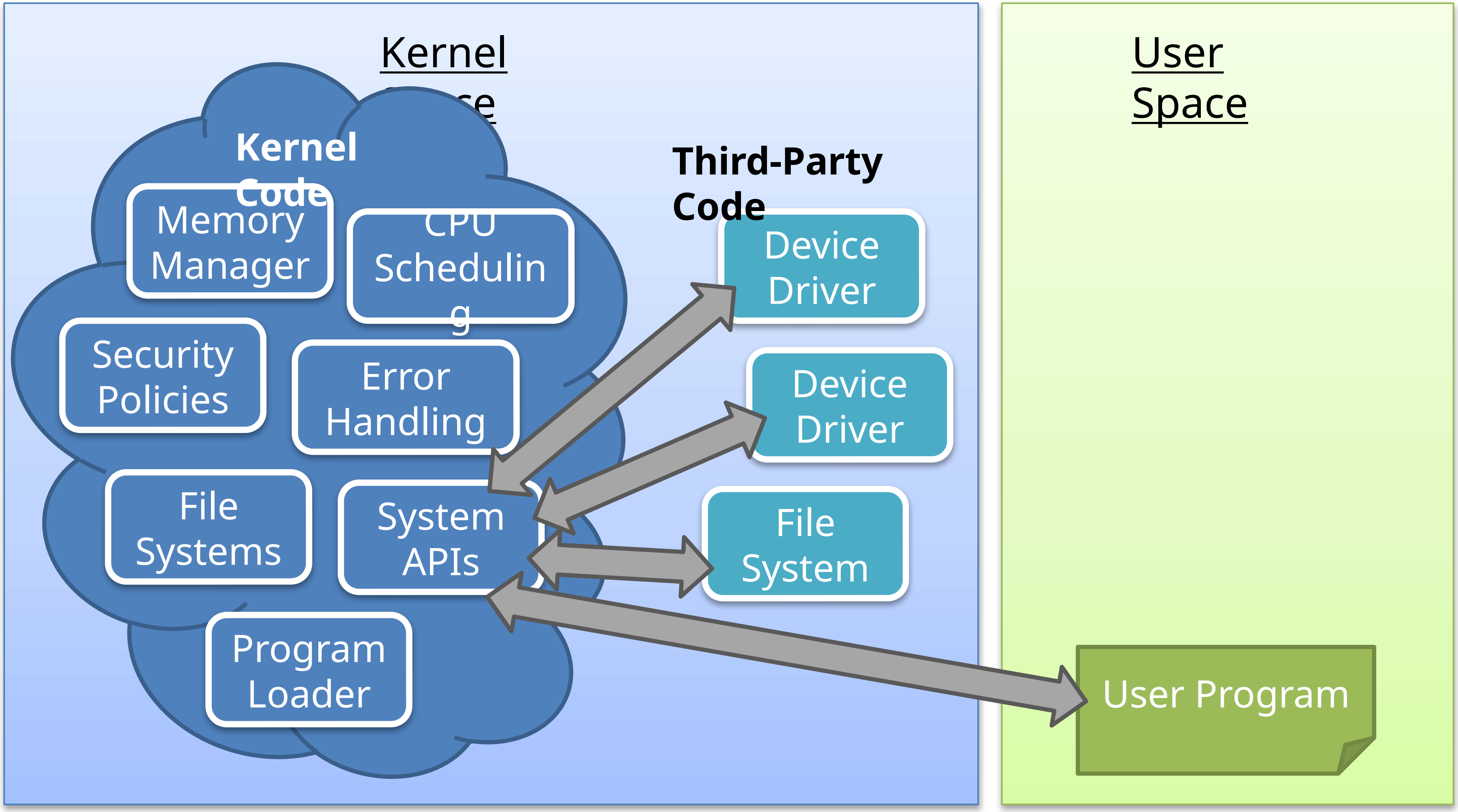
Architecting Kernels

- Three basic approaches
  1. Monolithic kernels
     - All functionality is compiled together
     - All code runs in privileged kernel-space
  2. Microkernels
     - Only **essential** functionality is compiled into the kernel
     - All other functionality runs in unprivileged user space
  3. Hybrid kernels
     - Most functionality is compiled into the kernel
     - Some functions are loaded dynamically
     - Typically, all functionality runs in kernel-space

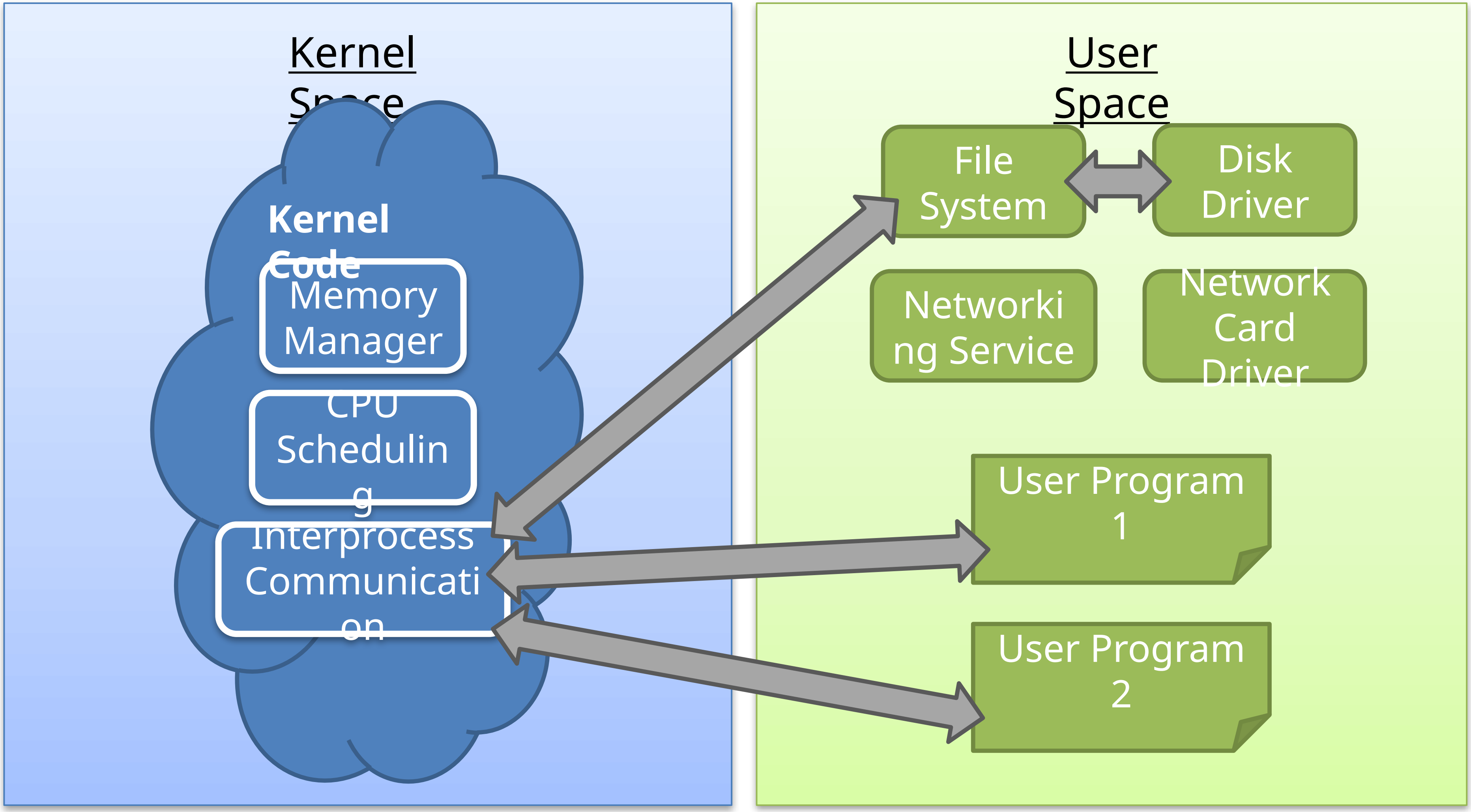
Monolithic Kernel

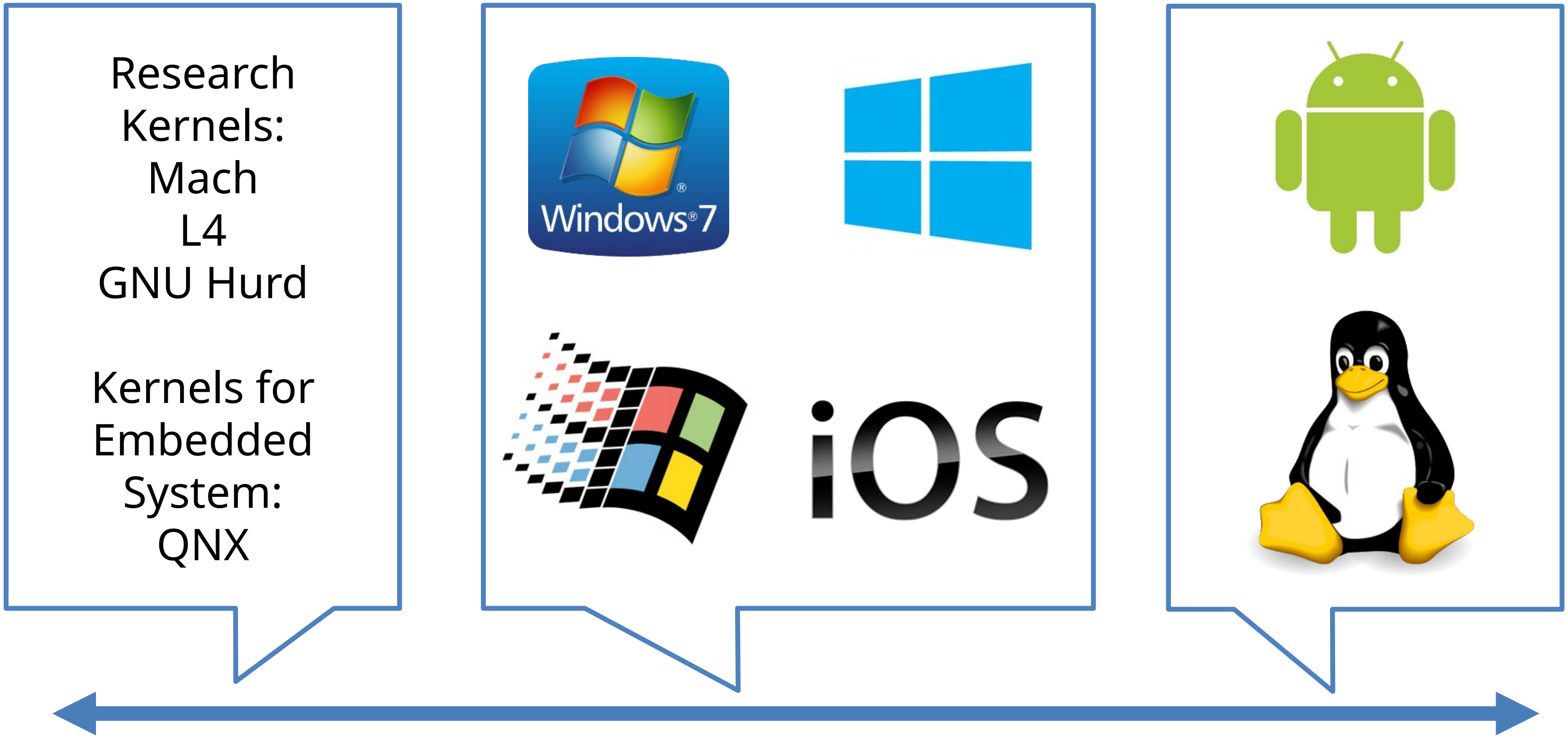


Hybrid Kernel



Microkernel





Research
Kernels:
Mach
L4
GNU Hurd

Kernels for
Embedded
System:
QNX

Microkernels:
Small code
base,
Few features

Hybrid Kernels:
Pretty large code
base,
Some features
delegated

**Monolithic
Kernels:**
Huge code base,
Many features

Pros/Cons of Monolithic Kernels

- Advantages
  - Single code base eases kernel development
  - Robust APIs for application developers
  - No need to find separate device drivers
  - Fast performance due to tight coupling
- Disadvantages
  - Large code base, hard to check for correctness
  - Bugs crash the entire kernel (and thus, the machine)

Pros/Cons of Microkernels

- Advantages
  - Small code base, easy to check for correctness
    - Excellent for high-security systems
  - Extremely modular and configurable
    - Choose only the pieces you need for embedded systems
    - Easy to add new functionality (e.g. a new file system)
  - Services may crash, but the system will remain stable
- Disadvantages
  - Performance is slower: many context switches
  - No stable APIs, more difficult to write applications