CSE120
Principles of Operating Systems

Architecture Support for OS

Why are you still here?

- You should run away from my CSE120!

Announcement

- Have you visited the web page?
  - http://cseweb.ucsd.edu/classes/fa18/cse120-a/
- Project 0 (installation & Submission) out
  - Due on 10/9
  - Some students have submitted (Great job! I am proud of you☺)
- Project groups for project 1- 3
  - Size 1-3
  - Remember to fill the google doc form
- TAs & tutors will work together to provide maximal lab hour coverage
  - Check the Lab Hour Google Calendar (available from the class webpage)
  - Entirely optional

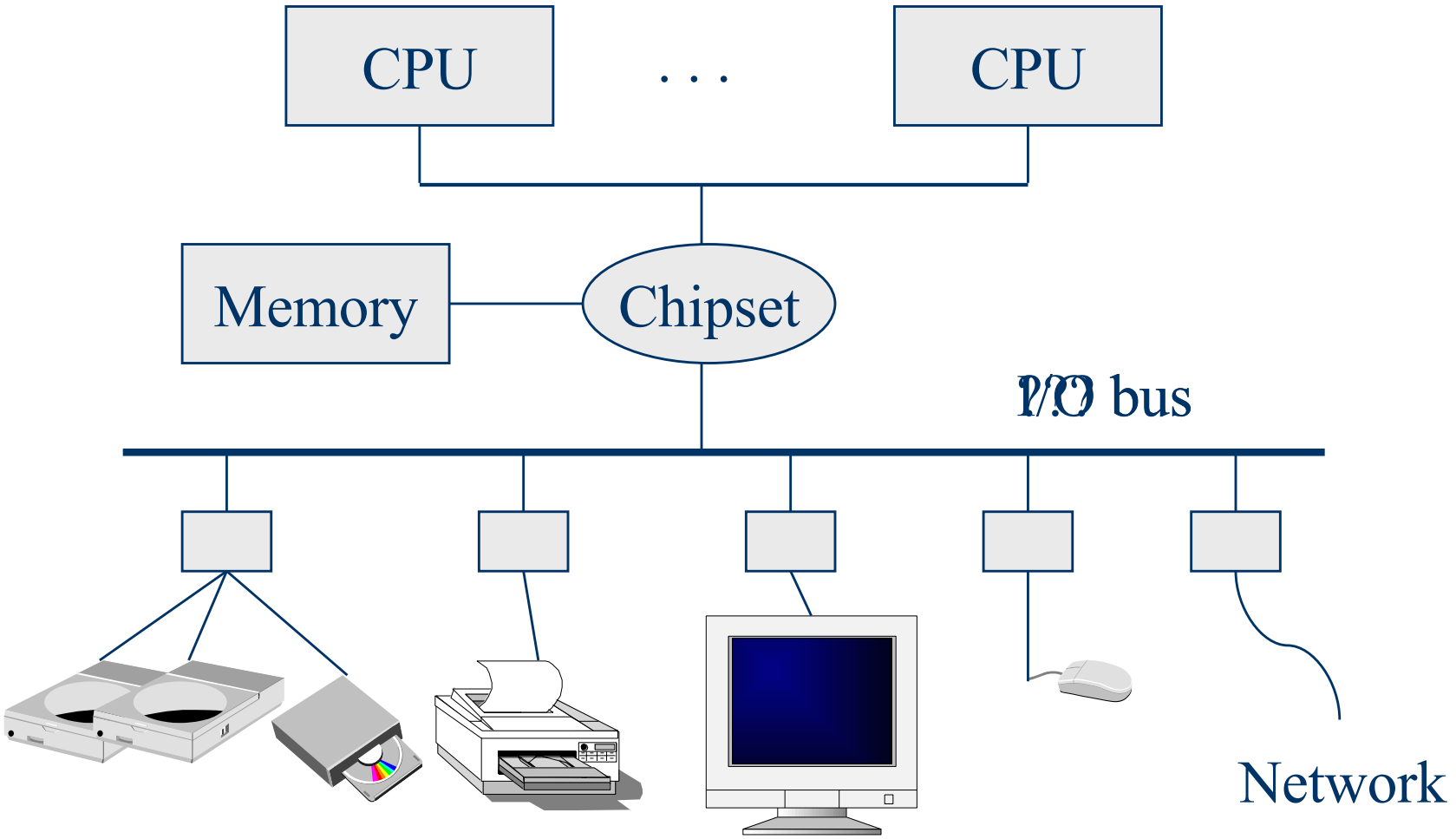
Why Start With Hardware?

- Operating system functionality depends upon hardware
  - Key goals of an OS are to enforce protection and resource sharing
  - If done well, applications can be oblivious to HW details

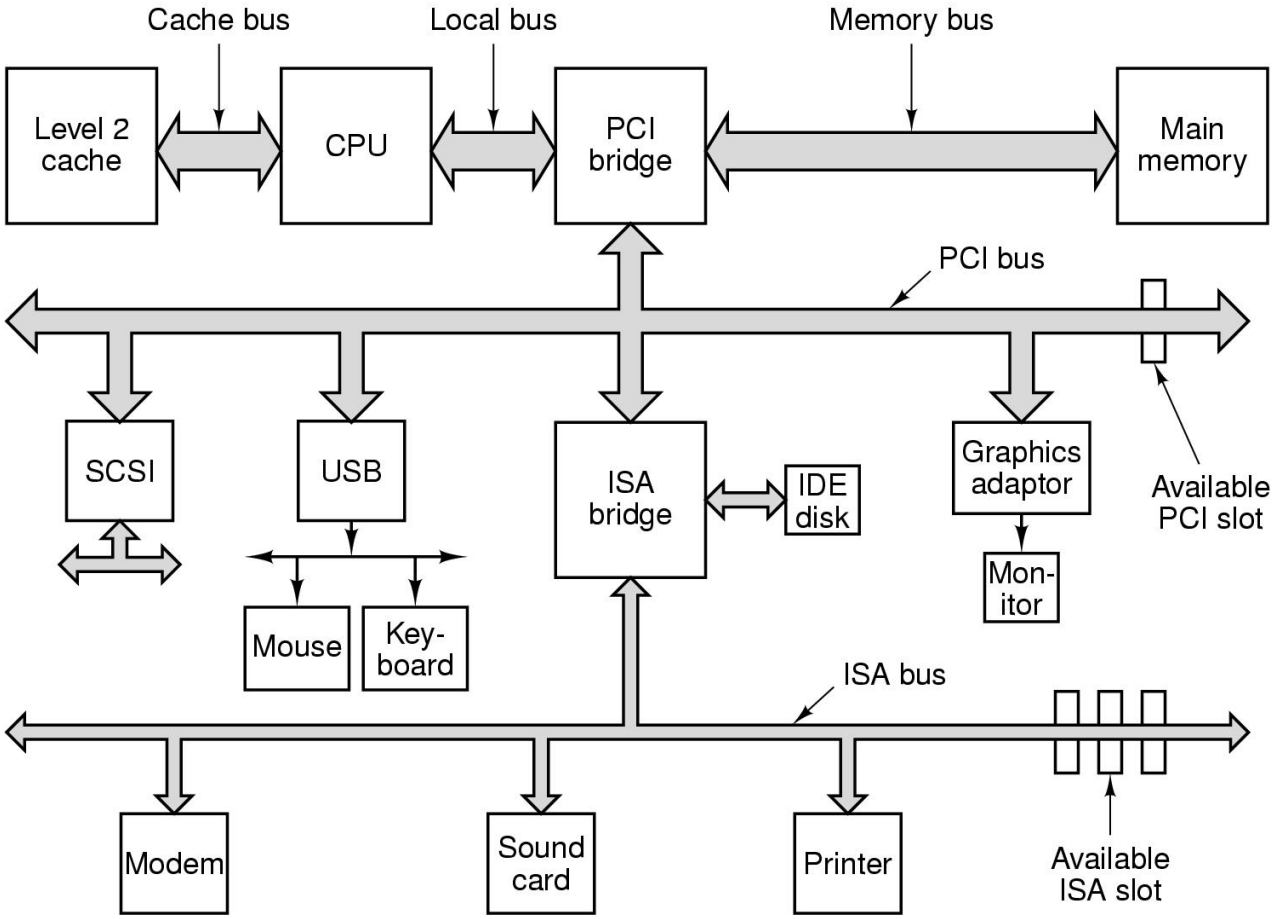
So what is inside a computer?

- An introduction with a real computer
  - http://www.youtube.com/watch?v=VWzX4MEYOBk
- An abstract overview
  - http://www.youtube.com/watch?v=Q2hmuqS8bwM&feature=related
- http://www.youtube.com/watch?v=FwtF7tz2p-s&feature=related

A Typical Computer from a Hardware Point of View

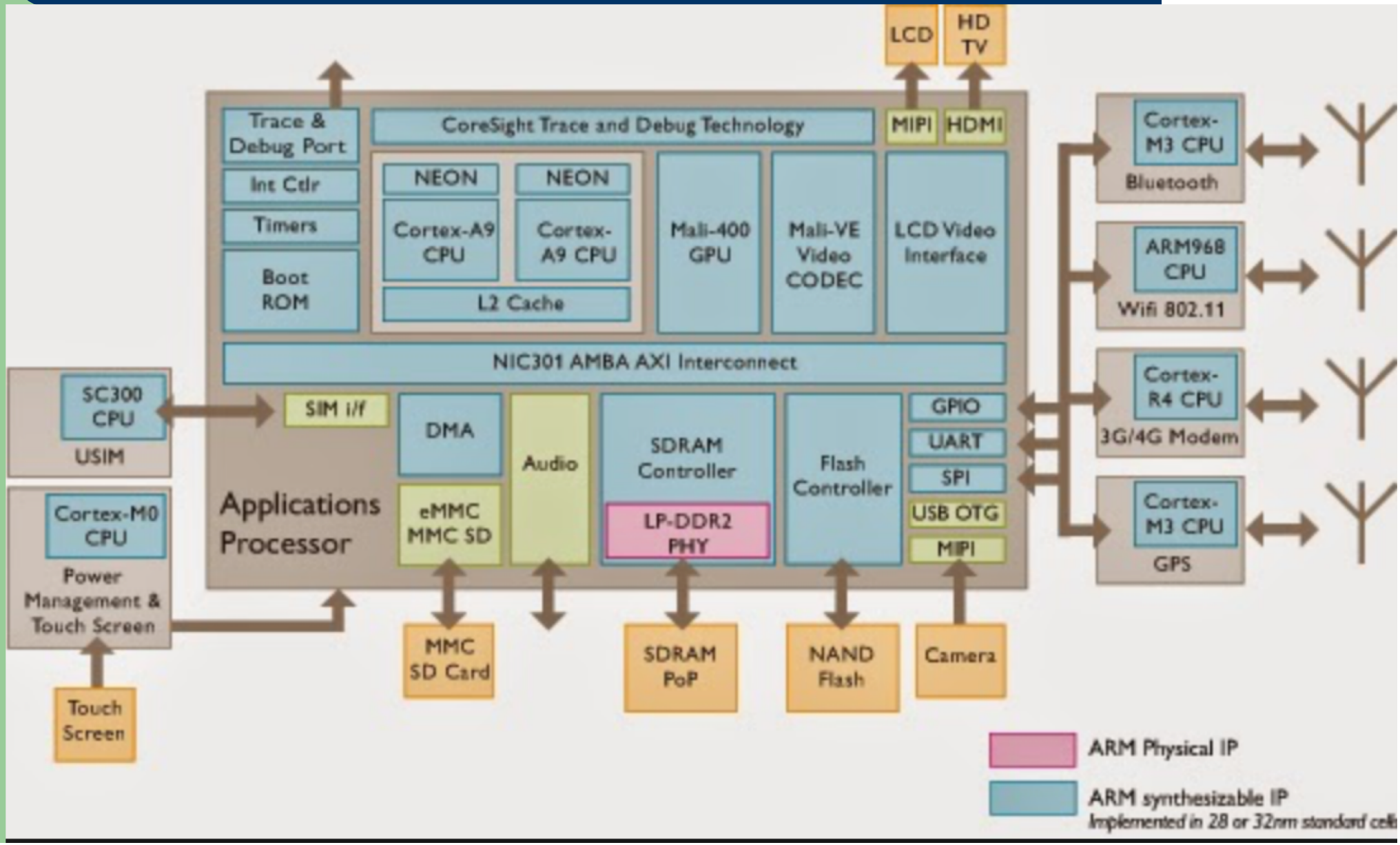


Pentium System: can you read it?

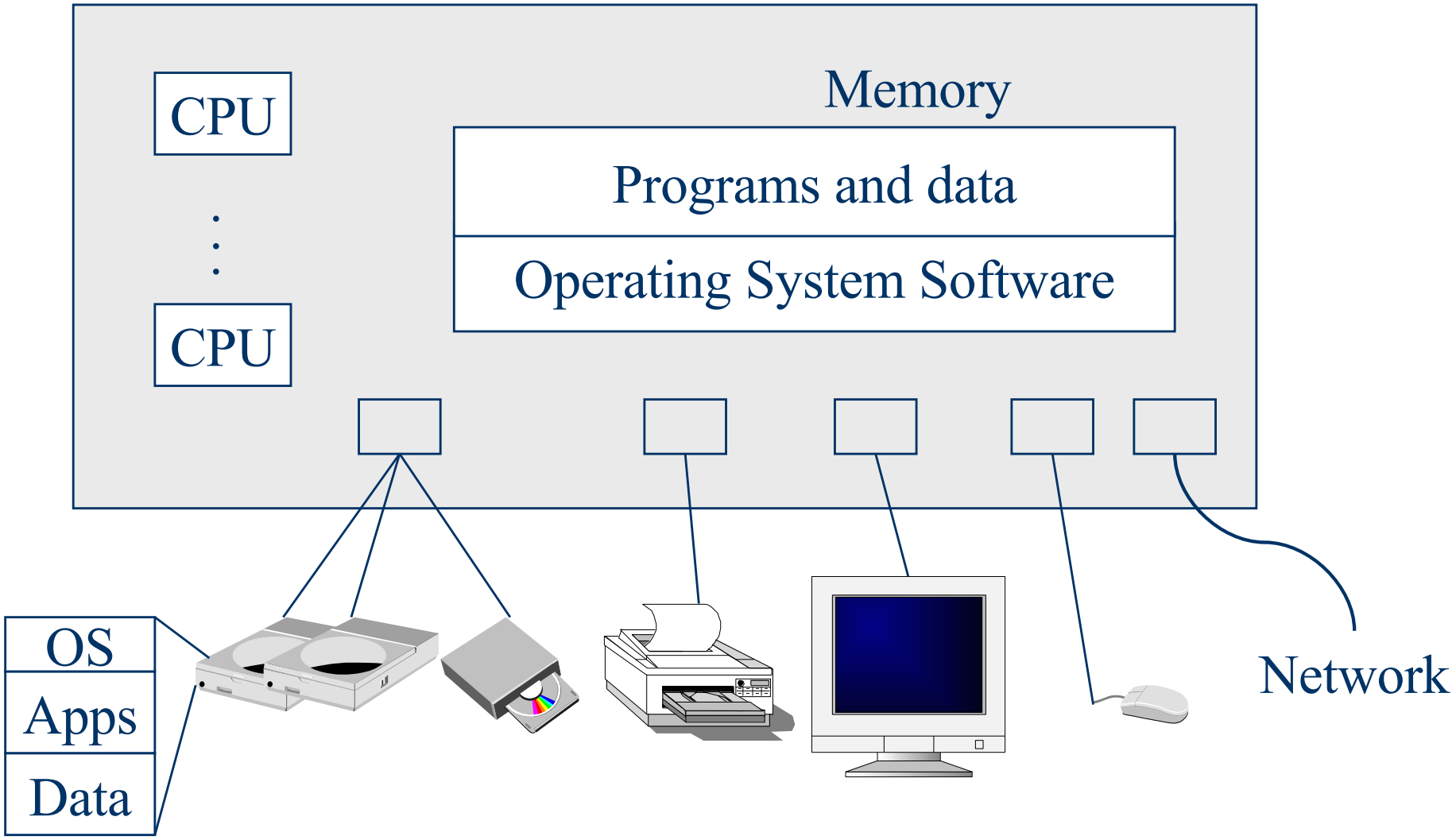


Structure of a large PC system

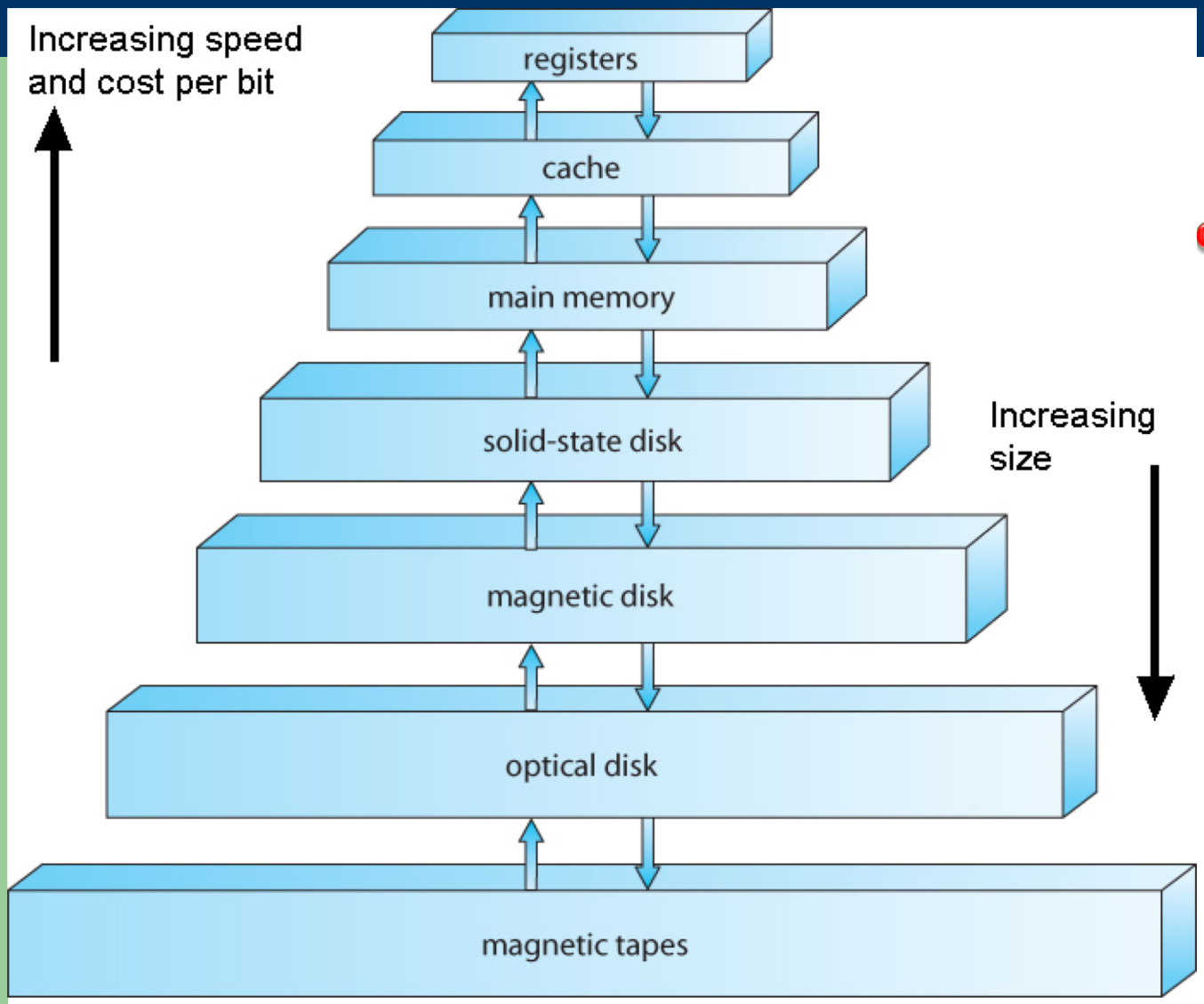
Today's Smartphone



A Typical Computer System (black box)

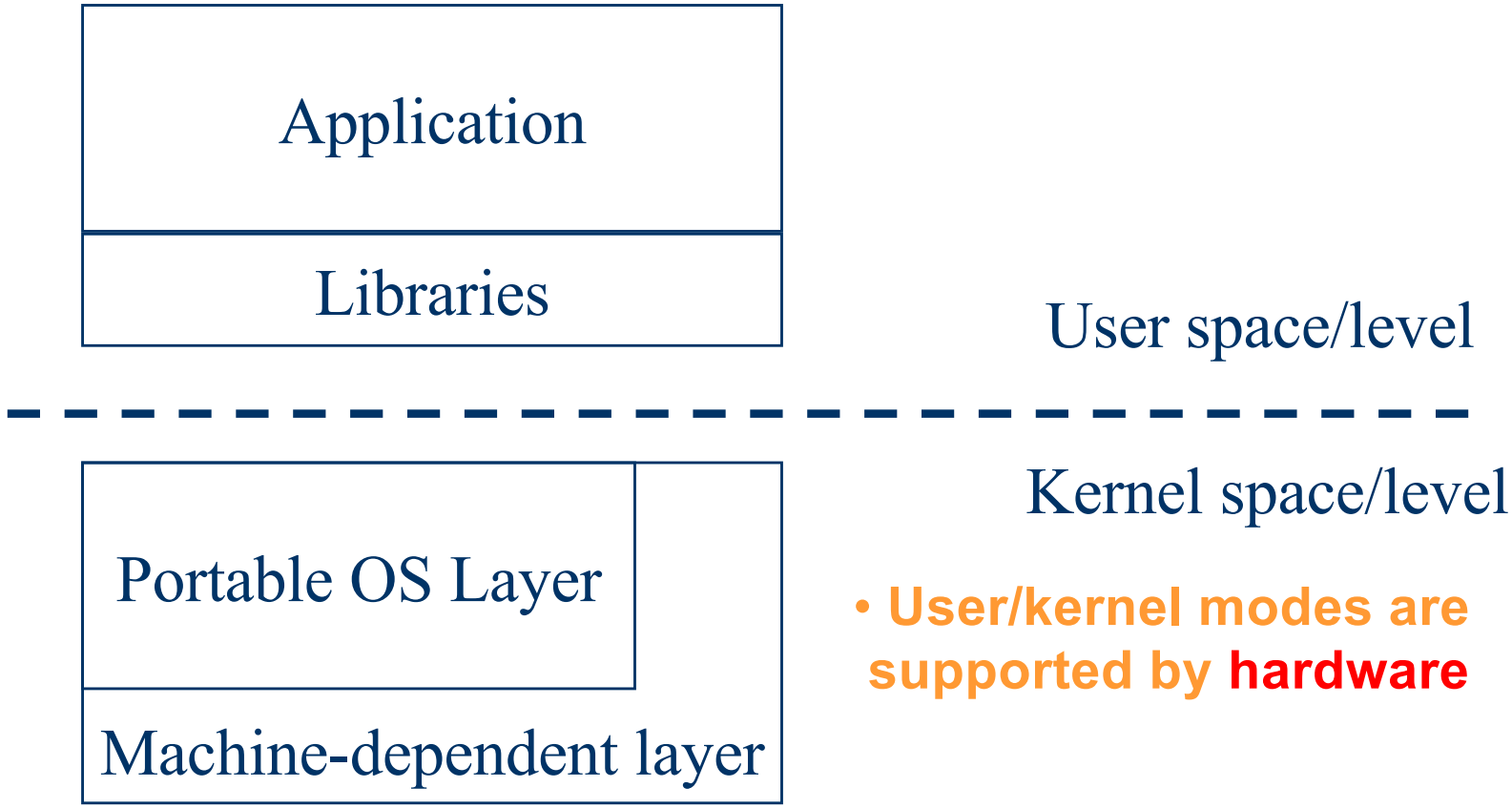


Memory-Storage Hierarchy



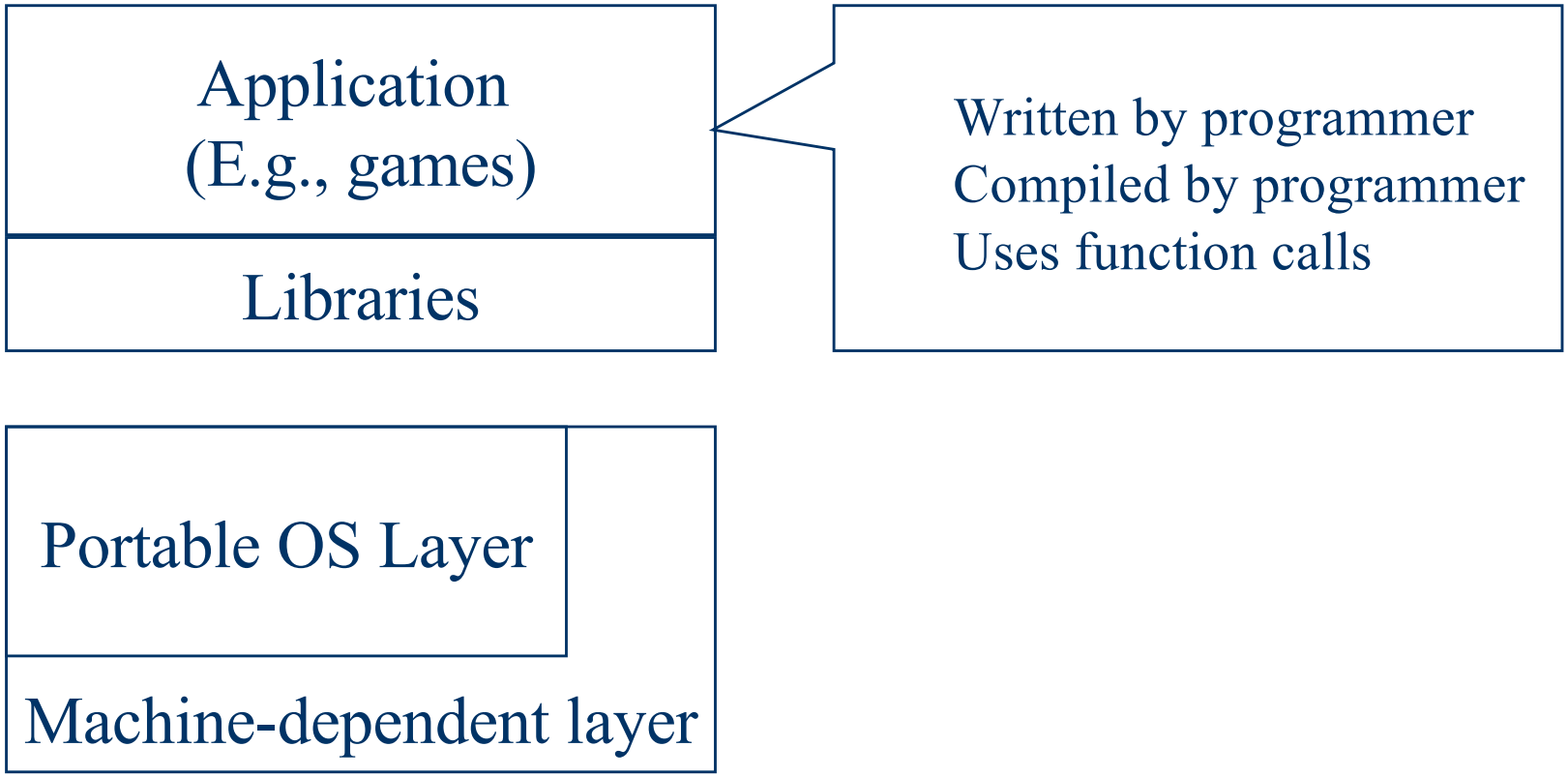
Why such complicated Hierarchy?

A peek into Unix/Linux/MacOS/Windows

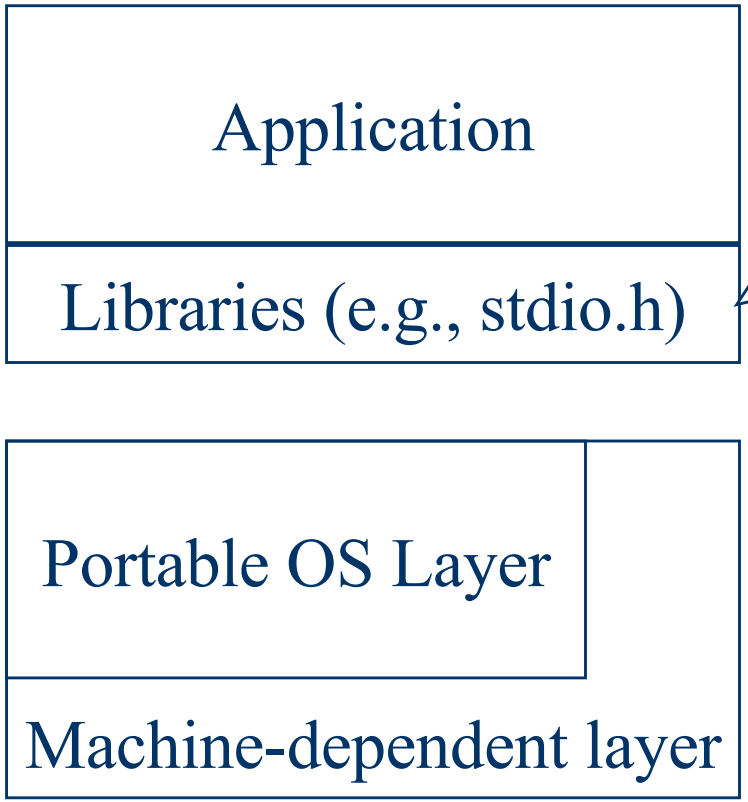


- **User/kernel modes are supported by hardware**

Application

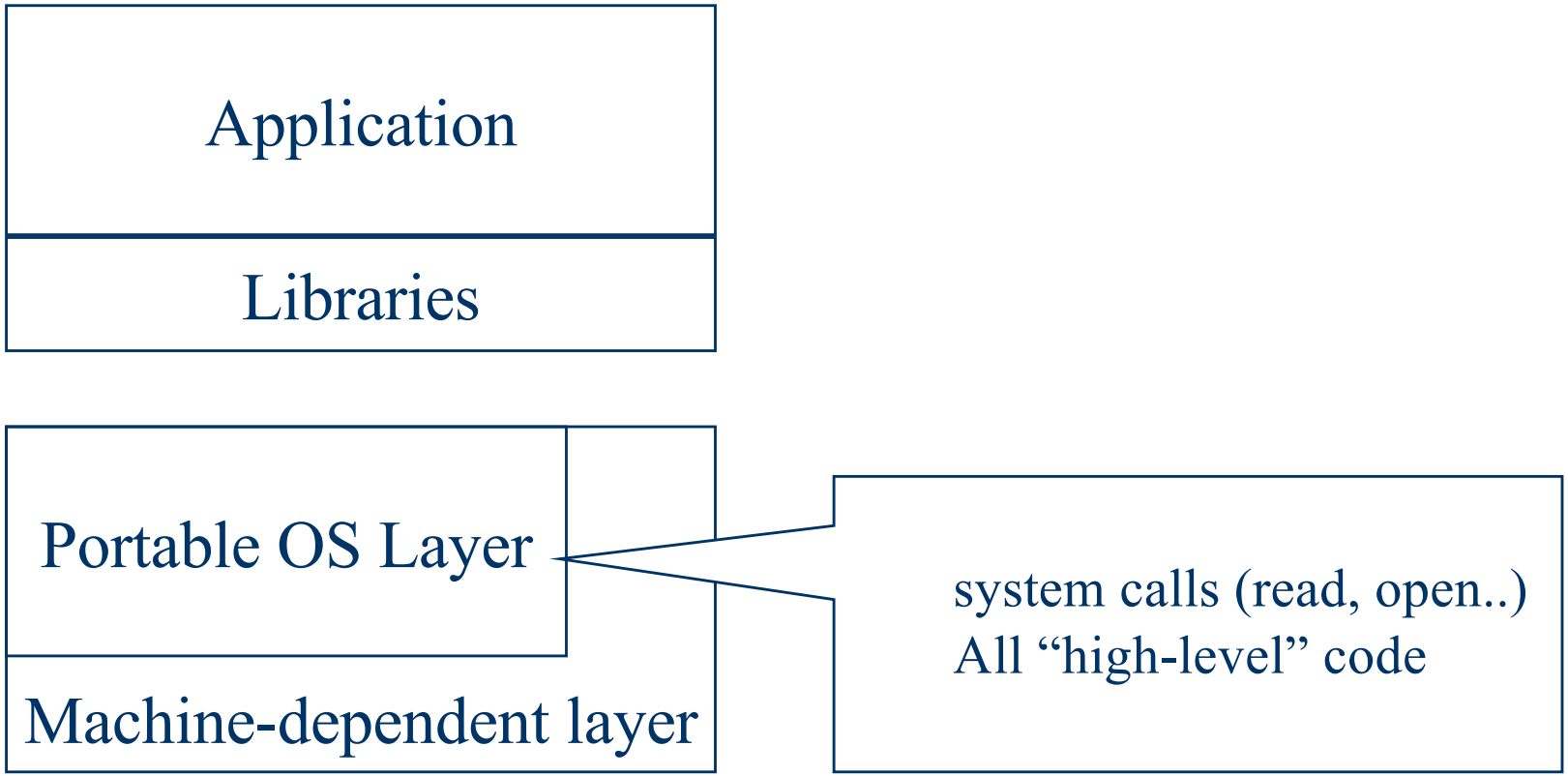


Libraries

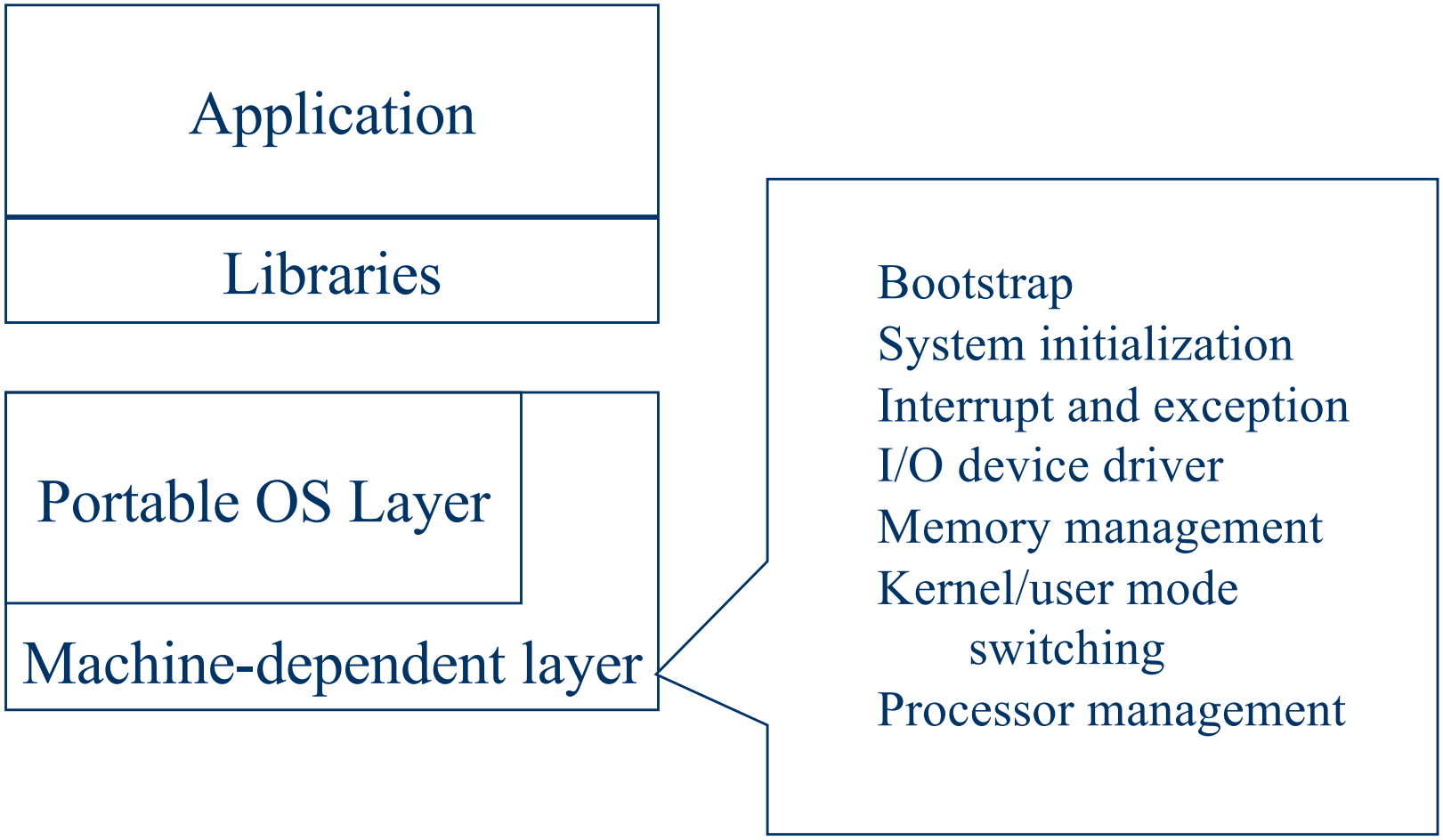


Written by elves
Provided pre-compiled
Defined in headers
Input to linker (compiler)
Invoked like functions
May be "resolved" when program is loaded

Typical OS Structure



Typical OS Structure



Hardware Features for OS

- Features that directly support the OS include
  - Protection (kernel/user mode)
  - Protected/privilege instructions
  - Memory protection
  - System calls
  - Interrupts and exceptions
  - Timer (clock)
  - I/O control and operation
  - Synchronization (will talk about this later)

Protected/Privileged Instructions

- A subset of instructions of every CPU is restricted to use only by the OS
  - Known as protected (privileged) instructions
- Only the operating system can
  - Directly access I/O devices (disks, printers, etc.)
    - Security, fairness
  - Manipulate memory management state
    - Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - Kernel mode, interrupt level
  - Halt instruction (why?)

Now close your eyes

- Imagine a world that any program can
  - Directly access I/O devices
  - Write anywhere in memory
  - Execute machine halt instruction
  - …
- What would happen?
  - Do you trust such computer systems to manage
    - Your banking account?
    - Your email account?
    - Your facebook account? ☺

OS Protection

- How do we know when it can execute a protected/privilege instruction now?
  - An easy answer: when it runs the OS code.
- But how does it know that it runs the OS code?
  - **Hardware** must support (at least) two modes of operation: kernel mode and user mode
  - Mode is indicated by a status bit in a protected control register
  - User programs execute in user mode
  - OS executes in kernel mode (OS == "kernel")
- Protected instructions only execute in *kernel* mode
  - CPU checks mode bit when protected instruction executes
  - Attempts to execute in user mode are detected and prevented
- Setting the mode bit must be a protected instruction

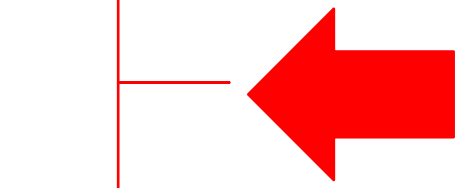
Memory Protection

- Why?
  - OS must be able to protect programs from each other
  - OS must protect itself from user programs
- Memory management hardware (called MMU) provides memory protection mechanisms
- Manipulating MMU uses protected (privileged) operations

Hardware Features for OS

- Features that directly support the OS include
  - Protection (kernel/user mode)
  - Protected instructions
  - Memory protection
  - System calls
  - Interrupts and exceptions
  - Timer (clock)
  - I/O control and operation
  - Synchronization

Questions?



OS

- Is OS like a tiger mom?
  - Always there monitoring “kids” (user level programs)
    - Do your homework, don’t text, you are grounded, you are playing too much video games, don’t snapchat, …
- Actually OS is NOT a tiger mom
  - OS is more like a

    ……

    ……

    ……..
  - Secretary

Commonality 1: Access to Resources

- OS is powerful, is a secretary powerful?
  - Yes, who has the master keys that can open every office?
  - Who knows the code to reserve conference rooms, printing hundreds of pages, …?
  - Who schedule classes?
    - Time and location

Commonality 2: Event-Driven

- Most importantly, both OSes and secretaries are event-driven
  - They are not constantly there monitoring the whole system (building)
  - Secretary's job is event driven:
    - Internal requests
      - E.g a professor asks her to book a trip
    - External events
      - E.g. a phone call from outside
    - Exceptions
      - E.g. Fire alarm

OS Control Flow

- After the OS has booted, all entry to the kernel happens as the result of an event
  - event immediately stops current execution
  - changes mode to kernel mode, event handler is called
- Kernel defines a handler for each event type
  - specific types are defined by the architecture
    - e.g.: timer event, I/O interrupt, system call trap
- When the processor receives an event of a given type, it
  - transfers control to handler within the OS
  - handler saves program state (PC, regs, etc.)
  - handler functionality is invoked
  - handler restores program state, returns to program

Events

- An event is an “unnatural” change in control flow
  - Events immediately stop current execution
  - Changes mode, context (machine state), or both
- The kernel defines a handler for each event type
  - Event handlers always execute in kernel mode
- Once the system is booted, all entry to the kernel (OS) occurs as the result of an event

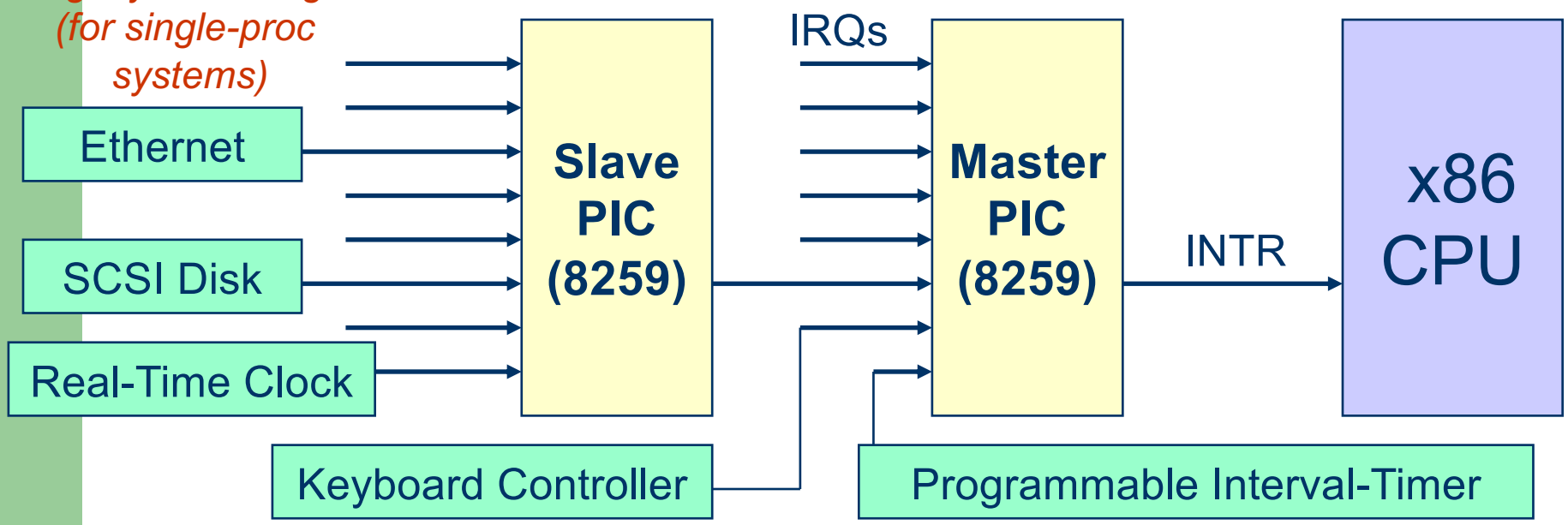
Categorizing Events

- Two kinds of events, interrupts and exceptions
  - Difference?
- Exceptions are caused by executing instructions
  - CPU requires software intervention to handle a fault or trap
- Interrupts are caused by an external event
  - Device finishes I/O, timer expires, etc.

Interrupts

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers

Interrupt Hardware

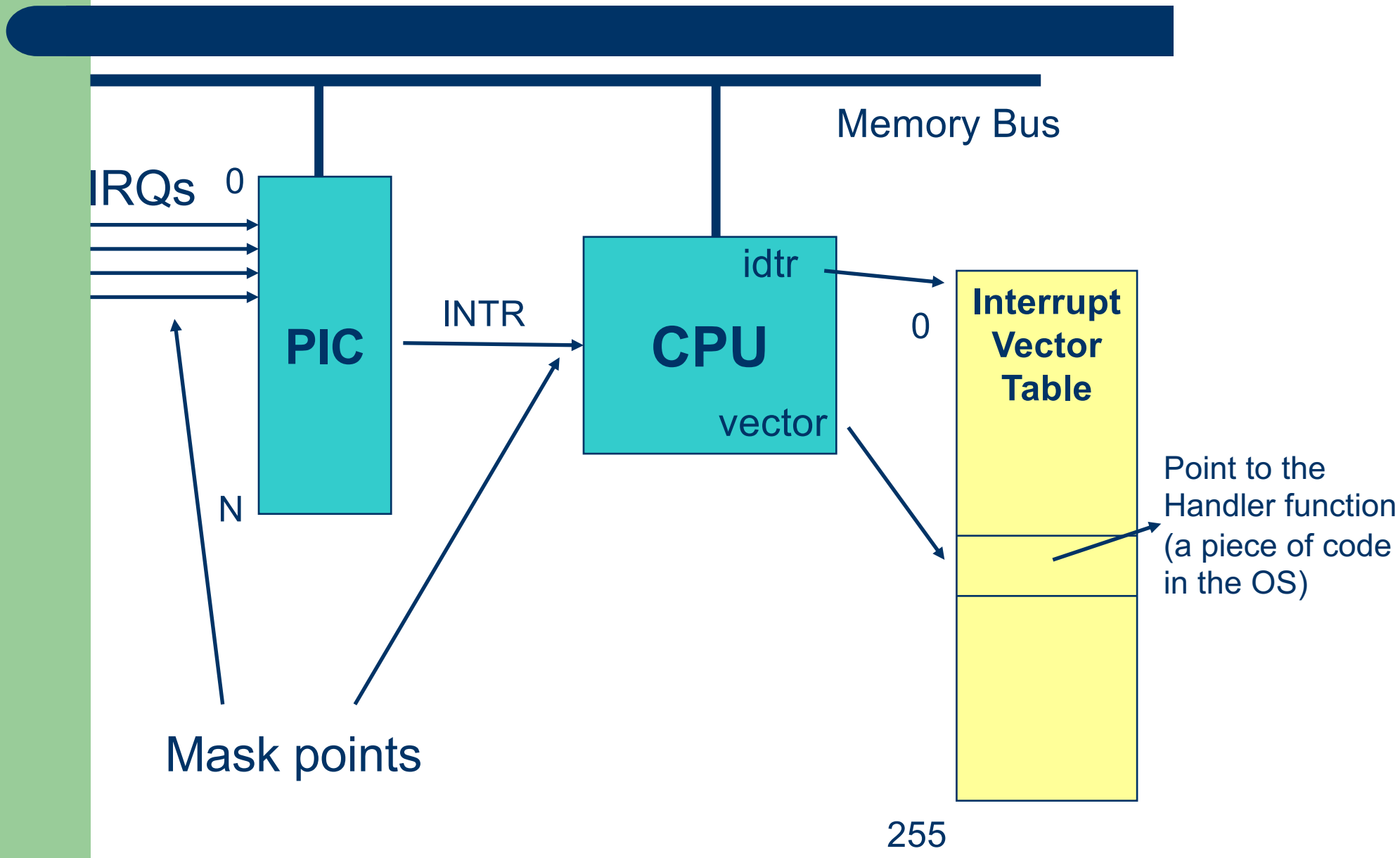


- I/O devices have (unique or shared) *Interrupt Request Lines* (IRQs)
- IRQs are mapped by special hardware to *interrupt vectors*, and passed to the CPU
- This hardware is called a *Programmable Interrupt Controller* (PIC)

The `Interrupt Controller'

- Responsible for telling the CPU when a specific external device wishes to 'interrupt'
  - Needs to tell the CPU *which* one among several devices is the one needing service
- PIC translates IRQ to *vector*
  - Raises interrupt to CPU
  - Vector available in register
  - Waits for ack from CPU
- Interrupts can have varying priorities
  - PIC also needs to prioritize multiple requests
- Possible to "mask" (disable) interrupts at PIC or CPU

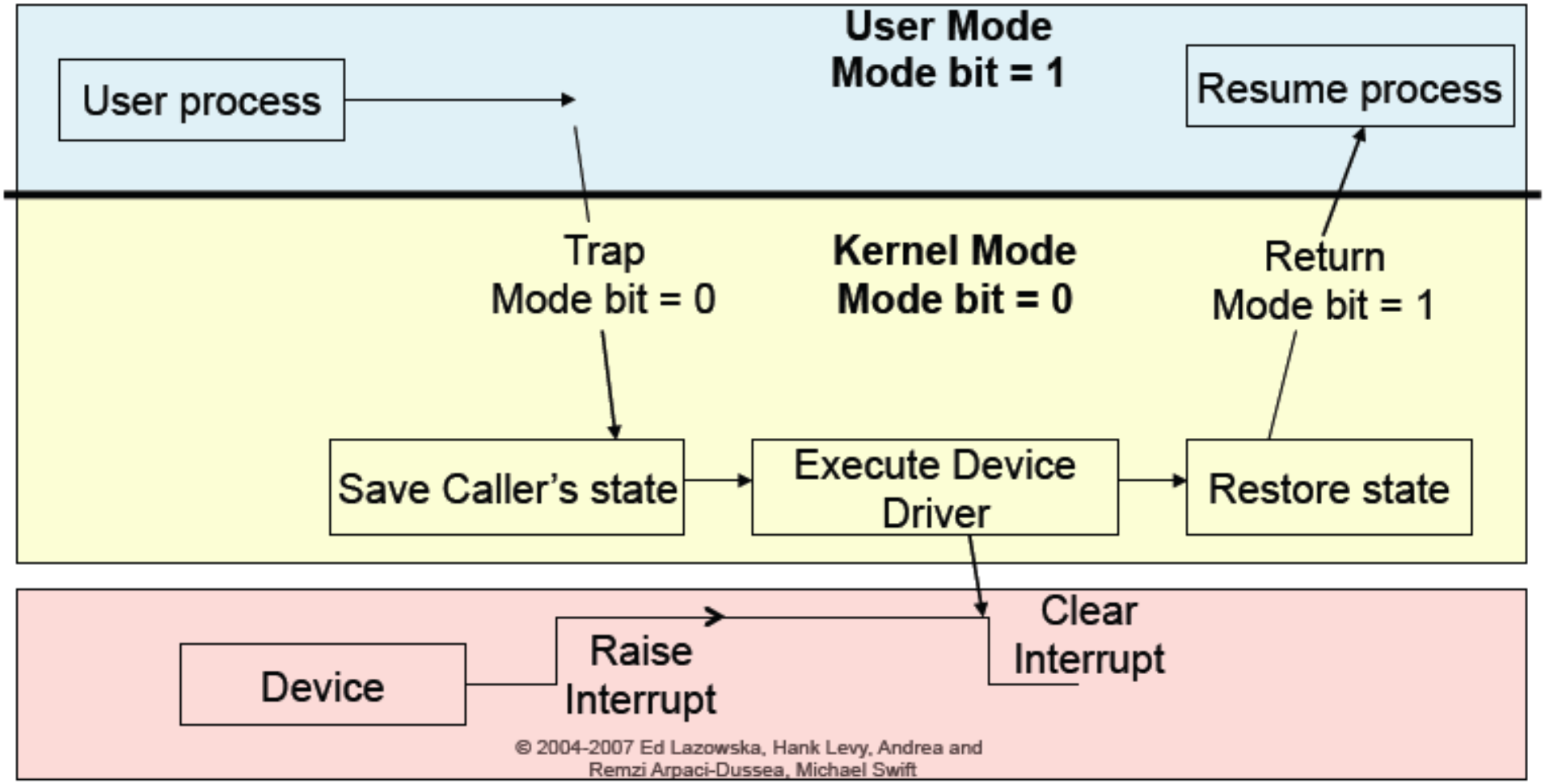
Hardware to Software



Interrupt Vector Table (x86)

Identifier	Description
0	Divide error
1	Debug exception
2	Non-maskable interrupt
3	Breakpoint
4	Overflow
5	Bounds check
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	(reserved)
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection fault
14	Page fault
15	(reserved)
14	Page fault
15	(reserved)
16	Coprocessor error
17	alignment error (80486)
18-31	(reserved)
32-255	External (HW) interrupts

Interrupt Illustrated



I/O Control

- I/O issues
  - Initiating an I/O
  - Completing an I/O
- Initiating an I/O
  - Special instructions

I/O Completion

- I/O process
  - OS initiates I/O
  - Device operates independently of rest of machine
  - Device sends an interrupt signal to CPU when done
  - OS maintains a vector table containing a list of addresses of kernel routines to handle various events
  - CPU looks up kernel address indexed by interrupt number, context switches to routine

I/O Example

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU(hardware) stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU(hardware) reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver (OS) processes packet (reads device registers to find packet in memory)
6. Upon completion, restores saved state from stack

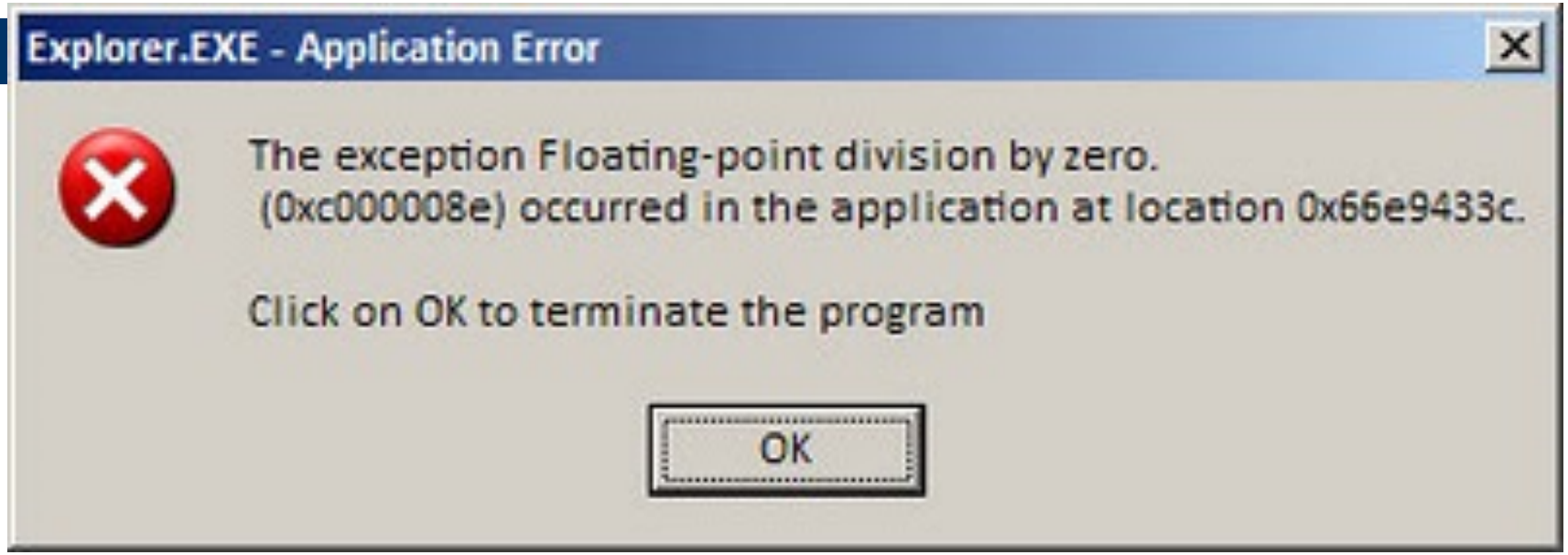
Timer

- http://www.online-stopwatch.com/countdown-timer/
- The timer is critical for an operating system
- It is the fallback mechanism by which the OS reclaims control over the machine
  - Timer is set to generate an interrupt after a period of time
    - Setting timer is a privileged instruction
  - When timer expires, generates an interrupt
  - Handled by kernel, which controls resumption context
    - Basis for OS scheduler *(more later…)*
- Prevents infinite loops
  - OS can always regain control from erroneous or malicious programs that try to hog CPU
- Also used for time-based functions (e.g., *sleep()*)

Interrupt Questions

- Interrupts halt the execution of a process and transfer control (execution) to the operating system
  - Can the OS be interrupted? (Consider why there might be different interrupt levels)
- Interrupts are used by devices to have the OS do stuff

Exception: Faults



Exception: Faults

- Hardware detects and reports "exceptional" conditions
  - Examples:
    - Page fault, unaligned access, divide by zero
- Upon exception, hardware "faults" (verb)
  - Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted. Why?

Handling Faults (1)

- The kernel may handle unrecoverable faults by killing the user process
  - Program fault with no registered handler
  - Halt process, write process state to file, destroy process
  - In Unix, the default action for many signals (e.g., SIGSEGV)
- What about faults in the kernel?
  - Dereference NULL, divide by zero, undefined instruction
  - These faults considered fatal, operating system crashes
  - Unix panic, Windows “Blue screen of death”
    - Kernel is halted, state dumped to a core file, machine locked up


Windows
A fatal exception 0E has occurred at 0028:C0011E36 in VXD VMM(01) +
00010E36. The current application will be terminated.
* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will
lose any unsaved information in all your applications.
Press any key to continue _

Handling Faults (advanced)

- Some faults are handled by "fixing" the exceptional condition and returning to the faulting context
  - Page faults cause the OS to place the missing page into memory
- Some faults are handled by notifying the process
  - Fault handler changes the saved context to transfer control to a user-mode handler on return from fault
  - Handler must be registered with OS
  - Unix signals or NT user-mode Async Procedure Calls (APCs)
    - SIGALRM, SIGHUP, SIGTERM, SIGSEGV, etc.

Switching Gear now…

- Only OS has direct access to hardware device, but what if a user level program want to do I/O?
  - Send a message to network
  - Print a sentence into the display
  - …
- OS needs to support user level programs to do LIMITED types of operations on hardware devices
- Note: not direct access
  - Analogy: Bob needs to get my laptop from my office, should my secretary give her the key to my office or should she get the laptop for him?

System Calls

- For a user program to do something "privileged" (e.g., I/O) it must call an OS procedure
  - Known as crossing the protection boundary, or a protected procedure call
- Hardware provides a system call instruction that:
  - Causes an exception
  - Passes a parameter determining the system routine to call
  - Saves caller state (PC, regs, mode) so it can be restored
    - Why save state?
  - Returning from system call restores this state
- Requires architectural support to:
  - Verify input parameters (e.g., valid addresses for buffers)
  - Restore saved state, reset mode, resume execution

System Call Functions

- Process control
  - Create process, allocate memory
- File management
  - Create, read, delete file
- Device management
  - Open device, read/write device, mount device
- Information maintenance
  - Get time, get system data/parameters
- Communications
  - Create/delete channel, send/receive message

- Programmers generally do **not** use system calls directly
  - They use runtime libraries (e.g. Java, C)
  - Why?

Refresh Your Assembly Code Experience

Convert to assembly:

```
while   (save[i] == k)
    i += 1;
```

i and k are in $s3 and $s5 and base of array save[] is in $s6

Shift left

```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:
```

Can someone explain the assembly code?

Function Call

```
#Listing 3
.globl main

main:     movl $10, %eax
          call foo
          …

foo:      addl $5, %eax
          ret
```

System Call

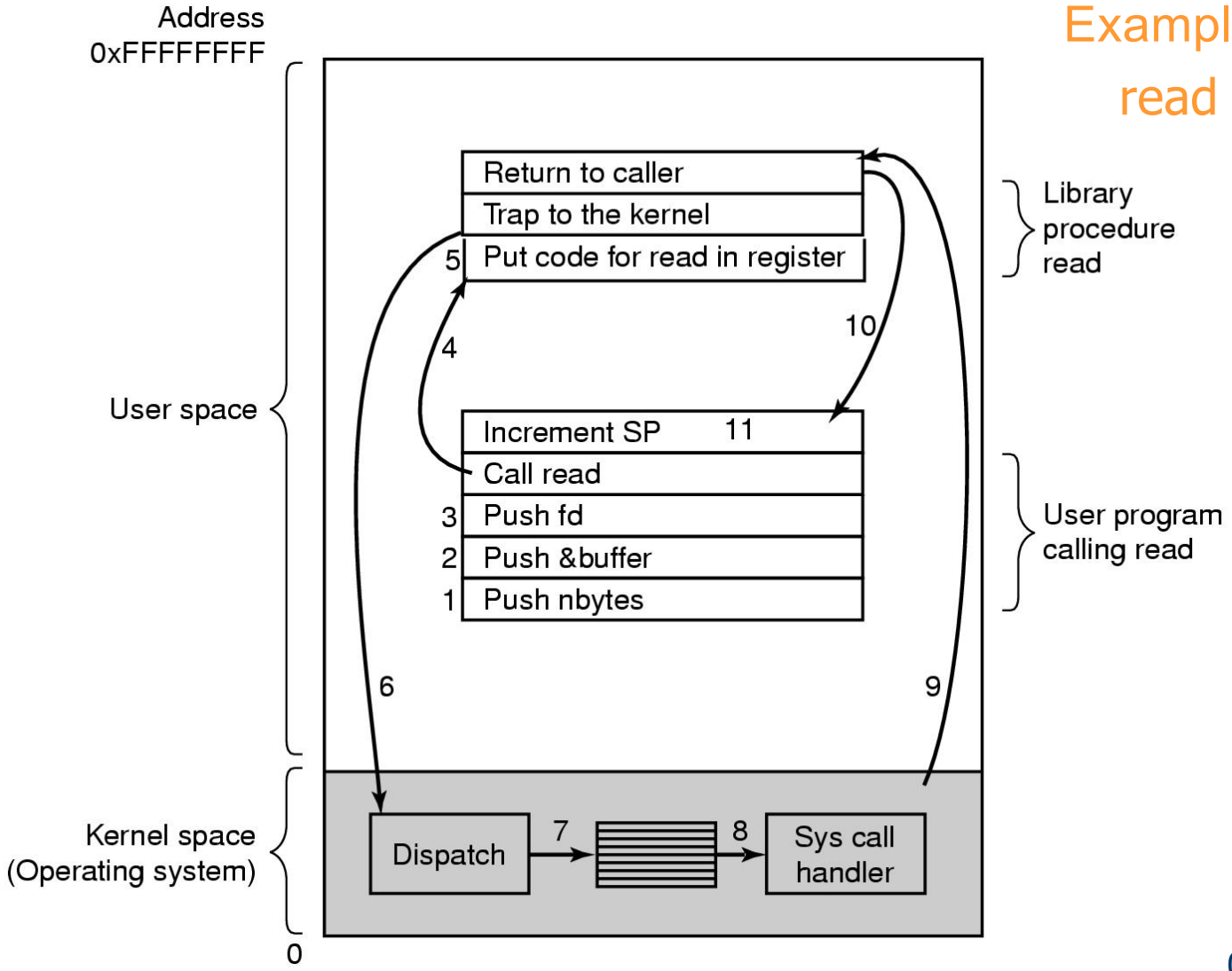
```
open:
        push    dword mode
        push    dword flags
        push    dword path
        mov     eax, 5
        push    eax                     ; Or any other dword
        int     80h
        add     esp, byte 16
```

- More information can be found in http://www.int80h.org/

Steps in Making a System Call

Example:

read (fd, buffer,nbytes)



System Call Issues

- What would happen if kernel didn't save state before a system call?
- Why must the kernel verify arguments?
- Why is a table of system calls in the kernel necessary?

Once Again--Summary

- After the OS has booted, all entry to the kernel happens as the result of an event
  - event immediately stops current execution
  - changes mode to kernel mode, event handler is called
- Kernel defines handlers for each event type
  - specific types are defined by the architecture
    - e.g.: timer event, I/O interrupt, system call trap
- When the processor receives an event of a given type, it
  - transfers control to handler within the OS
  - handler saves program state (PC, regs, etc.)
  - handler functionality is invoked
  - handler restores program state, returns to program

Summary

- Protection
  - User/kernel modes
  - Protected instructions
- System calls
  - Used by user-level processes to access OS functions
  - Access what is “in” the OS
- Exceptions
  - Unexpected event during execution (e.g., divide by zero)
- Interrupts
  - Timer, I/O