Housekeeping (Lecture 2 - 5/27/2025)

- PA1 is due at 11:45pm on Tuesday, 6/3/2025
  - if you have code from current or a previous semester, *do not look at/copy/share* any code from it
    - it's best if you just get rid of it
  - get started soon
    - if you are stuck, make sure you come to see me during office hours, send me email, or post in the class Piazza Forum
- If you haven't received Lecture 1 material, you should watch the recorded video as early as possible so you won't get surprised about *grading*

Copyright © William C. Cheng

Housekeeping (Lecture 2 - 5/27/2025)

- *Grading guidelines* is the *ONLY* way we will grade and we can only grade on a *standard 32-bit Ubuntu Linux 16.04 inside VirtualBox/UTM* or on AWS Free Tier
  - due to our *fairness* policy
  - the *grading guidelines* is *part of the spec*
  - although not recommended, you can do your development on a different platform
    - you must test your code on the "standard" platform because those are the *only* platforms the grader is allowed to grade on
  - if you make submission, make sure you run through the *Verify Your Submission* procedure as if you are the grader

- I have change the PA2 submission deadline to be the same as the PA3 submission deadline
  - this makes it similar to Prof. Ryutov’s class in fall/spring

Copyright © William C. Cheng

Housekeeping (Lecture 2 - 5/27/2025)

- I will be out of town this Thursday (to be at my daughter's graduation on the east coast)
  - I will record about 120 minutes of Lecture 3 on Zoom and will get them posted on the class website (around 40-45 minutes each)
    - I will make a Piazza post when the videos are available
  - no live class in KAP 146 12:30pm - 2:50pm this Thursday
  - sorry about the inconvenience

Copyright © William C. Cheng

Ch 1: Introduction

Bill Cheng

http://merlot.usc.edu/william/usc/

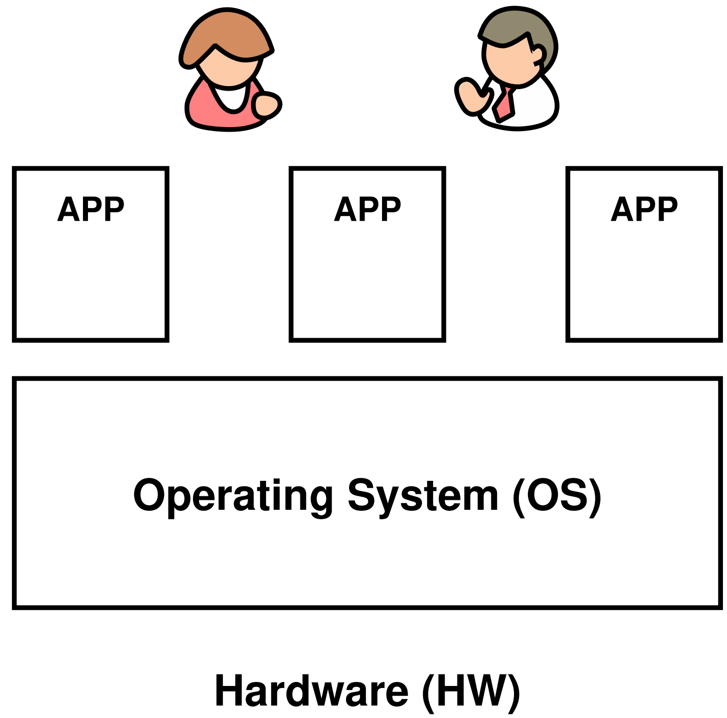
Copyright © William C. Cheng

(1.1) What Is An Operating System?

Copyright © William C. Cheng

What Is An Operating System?

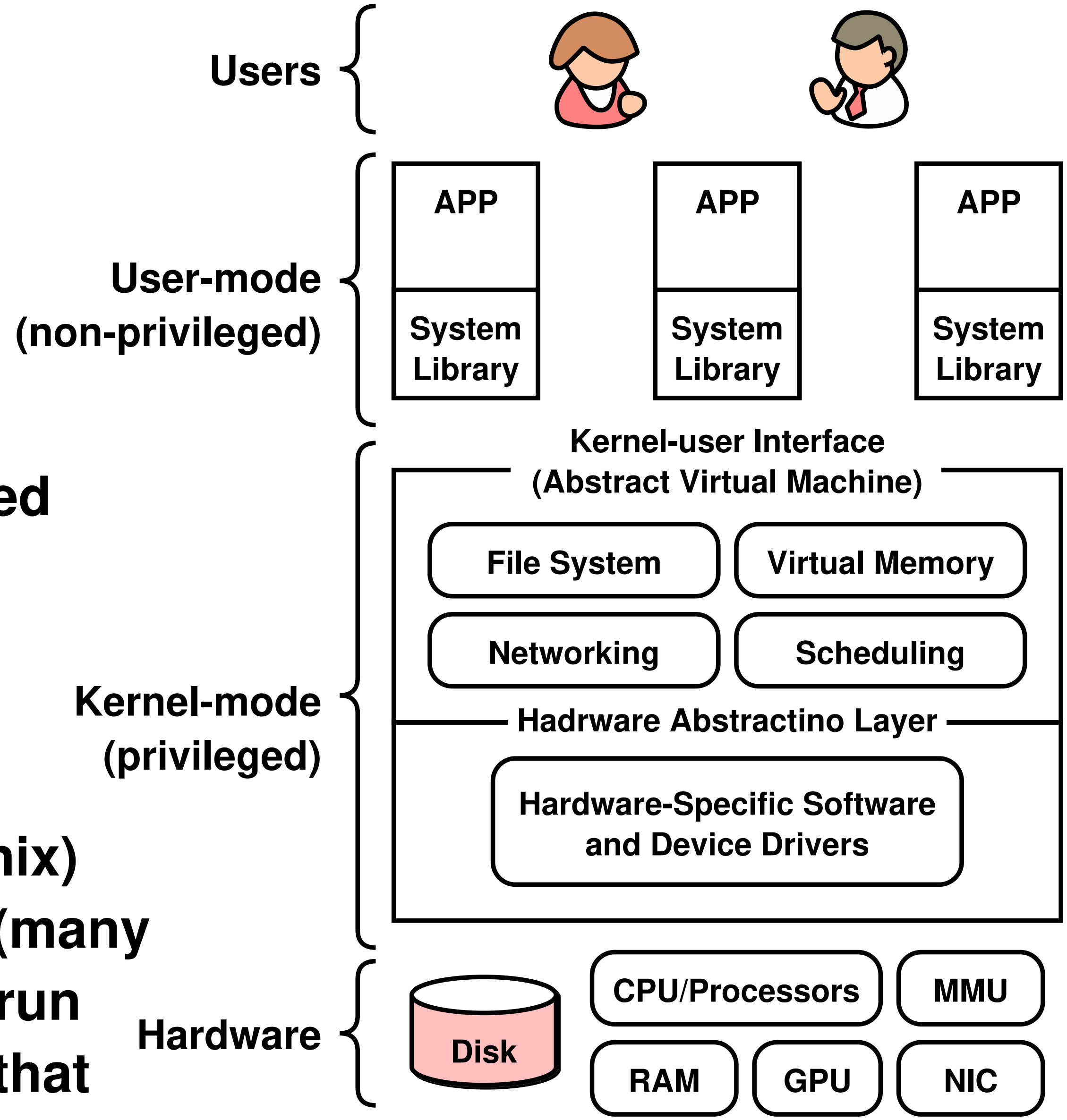
- **Software to manage a computer's resources for its users and applications**
  - generally, there are 3 types of HW
    1) CPU/processor (including GPU)
    2) RAM
    3) devices
  - we use the terms CPU, core, and processor interchangeably
    - although technically speaking, a multicore processor has multiple CPUs in it
    - if we assume that we only have a single-core processor, then processor = core = CPU
    - therefore, unless otherwise specified, this class only talks about *single-core processors* (multi-core processor system = multi-processor system)



Copyright © William C. Cheng

What Is An Operating System?

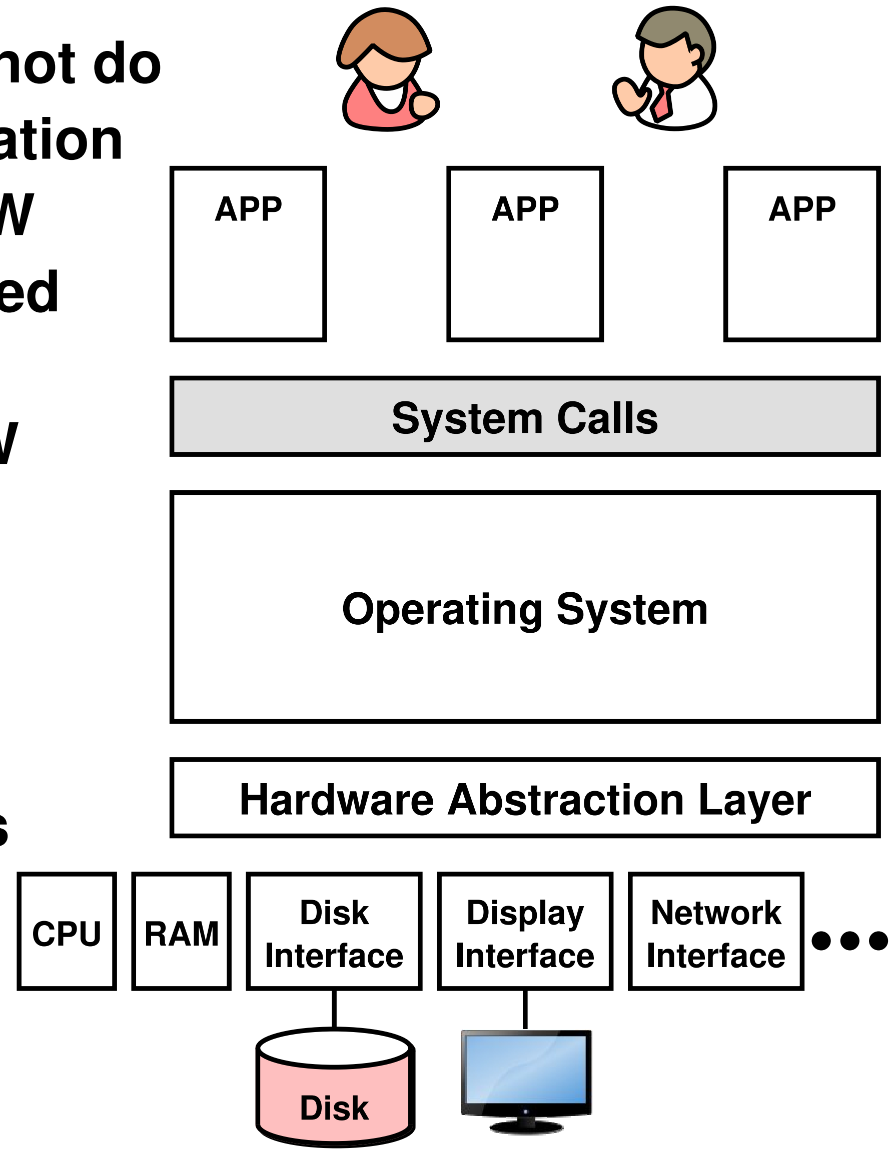
- Software to manage a computer's resources for its users and applications
  - *kernel* is part of the OS that can perform privileged operations
    - e.g., execute privileged machine instructions
    - e.g., talk to HW
  - does OS = kernel?
    - yes for some old OS (e.g., Sixth Edition Unix)
    - no for modern OSes (many OS components can run as user applications that do not require privilege at all time — not our focus)



Copyright © William C. Cheng

What Is An Operating System?

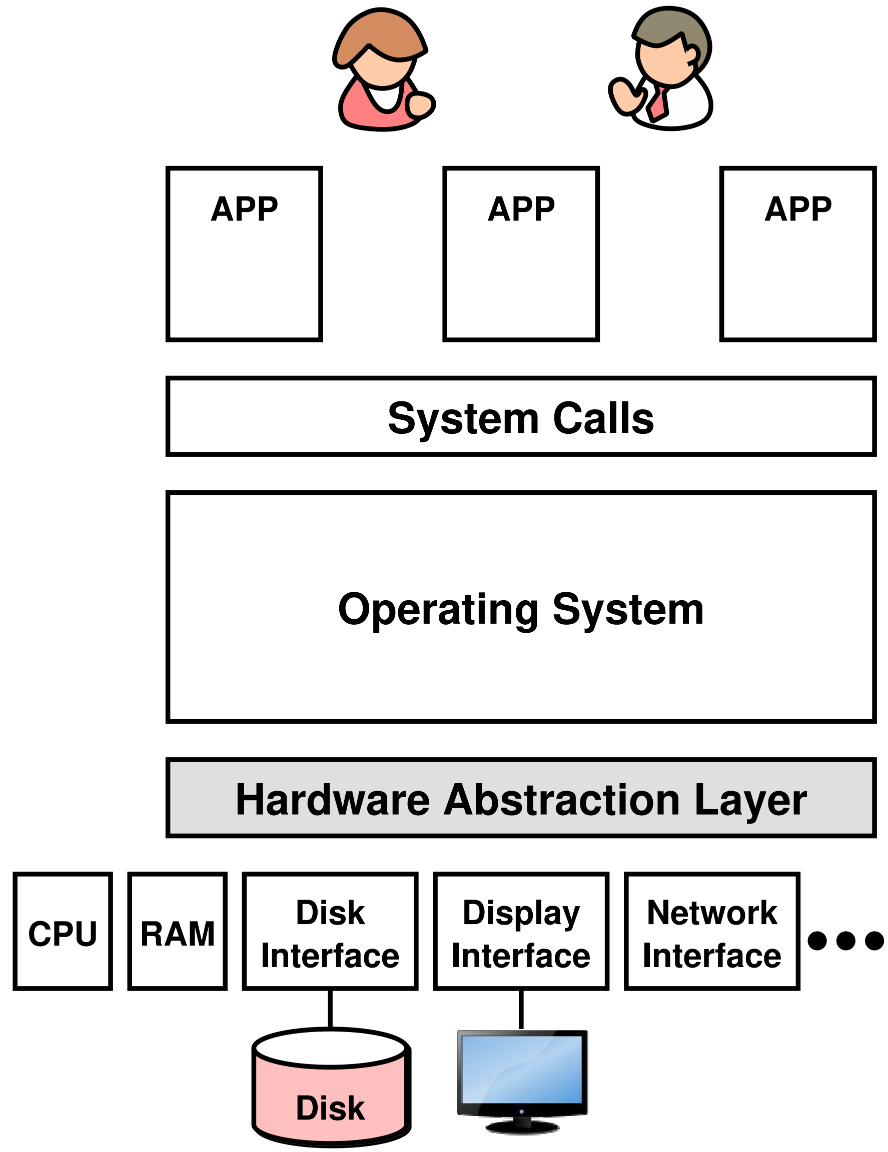
- There are things an application cannot do
  - e.g., it cannot run another application
  - they don't know how to talk to HW
    - therefore, they are not permitted to talk to HW
  - they don't know how to *share* HW and system resources (such as CPU and RAM)
  - they make *system calls* to ask the OS for assistance
    - OS does not trust applications
    - applications can only enter the kernel through there controlled entry points



Copyright © William C. Cheng

What Is An Operating System?

- ***Hardware Abstractino Layer (HAL)*** **makes the OS independent of the hardware**
  - **e.g., same OS code can run on different CPU types (just need to recompile the OS)**
  - **e.g., same OS code can talk to keyboards from any keyboard manufacturer**



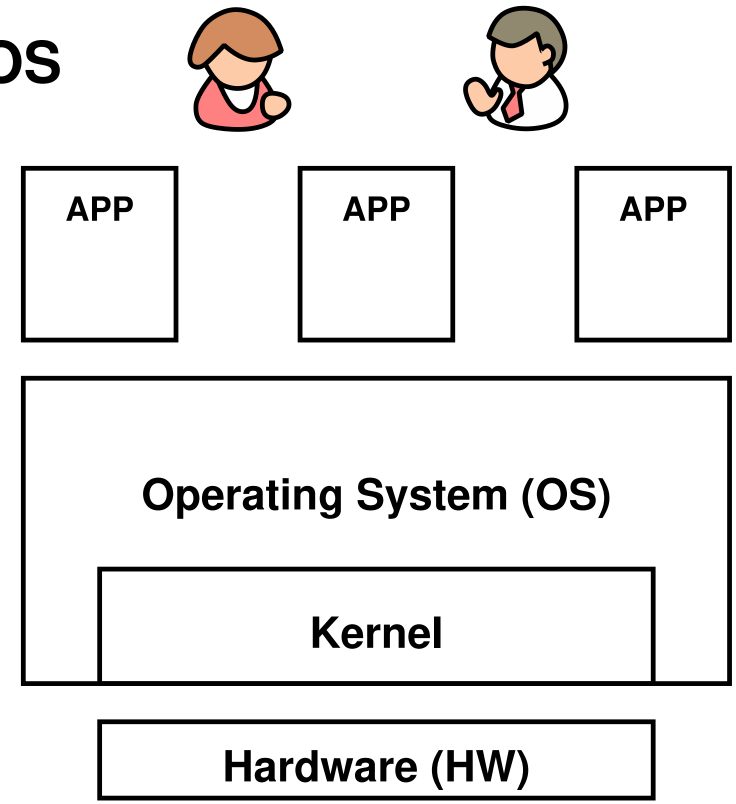
Copyright © William C. Cheng

(1.2) Operating System Evaluation

Copyright © William C. Cheng

What Is Kernel?

- The kernel is the core component of the OS
  - it has full access to all of the HW
  - it's responsible for
    - process management
    - CPU scheduling
    - memory management
    - storage management
    - protection and security
    - device and I/O management (not discussed in this class)



- Our focus is on the *kernel*
  - most of the time, we use the term "kernel" and "OS" interchangeably (since the kernel is the focus of this class)

Copyright © William C. Cheng

OS Challenges

- Reliability: does the system do what it was designed to do?
- Availability: what portion of the time is the system performing useful work?
- Security: can the system be compromised by an attacker?
- Performance
  - latency (or response time): how long does an operation take to complete?
  - throughput: how many operations can be done per unit of time?
  - overhead: how much extra work is done by the OS?
  - fairness: how equal is the performance received by different users?
  - predictability: how consistent is the performance over time?
- Portability
  - for programs: API and the Abstract Virtual Machine (AVM)
  - for OS: Hardware Abstraction Layer

Copyright © William C. Cheng

OS Is Complex

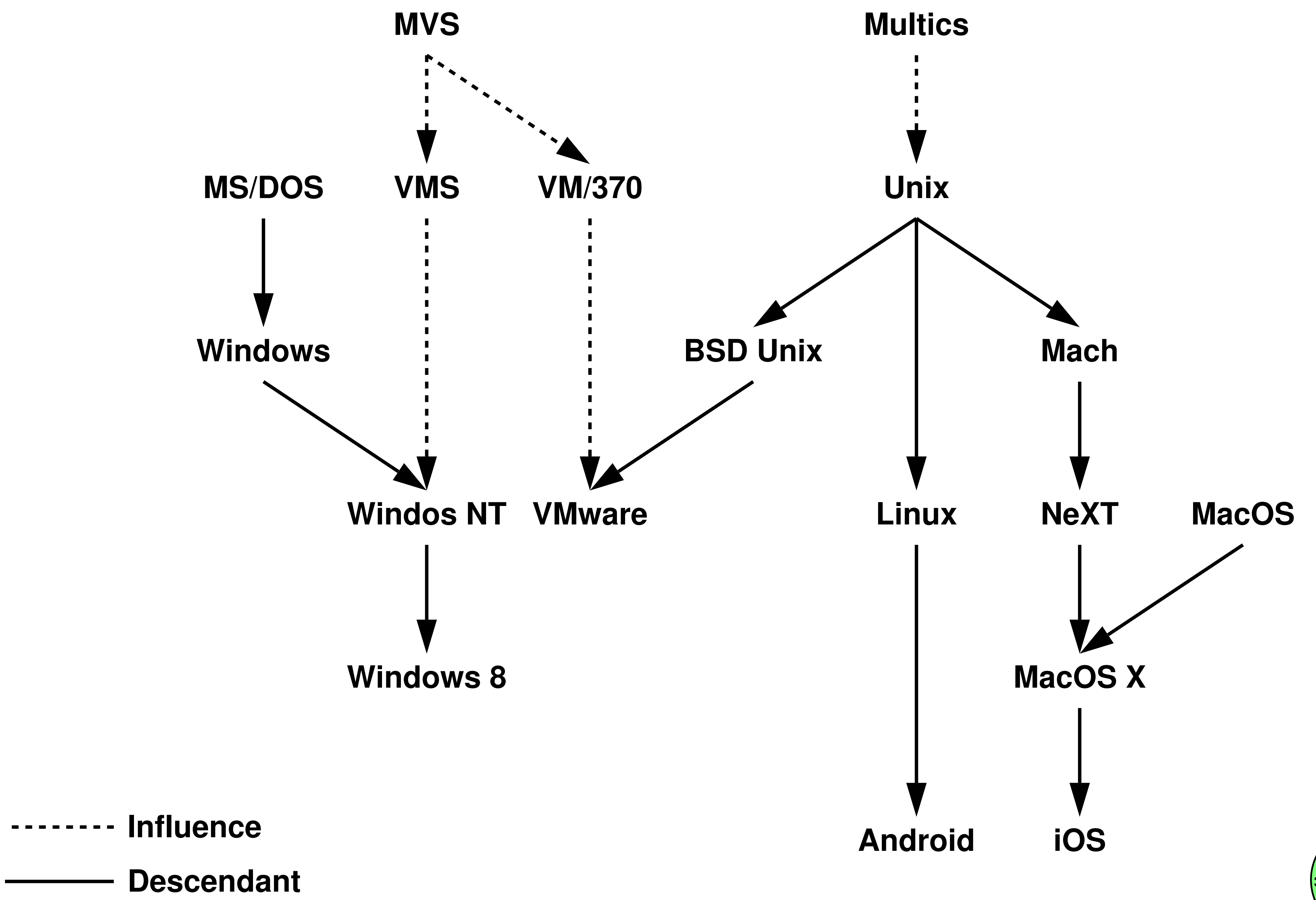
- **How can we build and understand a complex, messy system?**
  - **come up with some structure**
    1) **manage complexity through abstraction *(e.g., address space, process, file)***
    2) **apply good design principles (e.g., separation of policy from mechanism)**
    3) **consider tradeoffs (e.g., optimize for common case)**

Copyright © William C. Cheng

(1.3) Operating System: Past, Present, and Future

Copyright © William C. Cheng

OS History



Copyright © William C. Cheng

Computer Performance Over Time

	1981	1997	2014	Factor (2014/1981)
CPU speed (MIPs)	1	200	2500	2.5K
CPUs per computer	1	1	10+	10+
CPU $/MIPs	$100K	$25	$0.20	500K
DRAM capacity (MB/$)	0.002	2	1K	500K
Disk capacity (GB/$)	0.003	7	25K	10M
Home Internet	300 bps	256 Kbps	20 Mbps	100K
Machine room network	10 Mbps (shared)	100 Mbps (switched)	10 Gbps (switched)	1000
Users/computer	100:1	1:1	1:several	100+

Copyright © William C. Cheng

Extra Slides

Copyright © William C. Cheng

Ch 2: The Kernel Abstraction

Bill Cheng

http://merlot.usc.edu/william/usc/

Copyright © William C. Cheng

Challenge: Protection

- A central role of OS is *protection*
  - isolating bad applications and users so that they do not corrupt other applications or the OS (which is the protector of other applications)

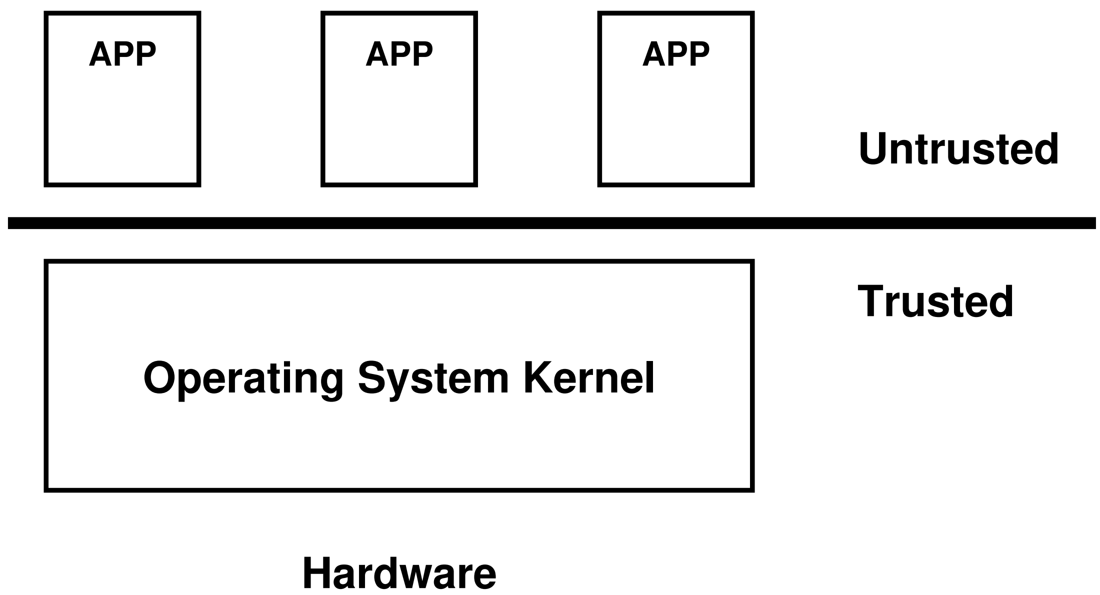
- Protection is essential to achiving some of the OS goals
  - *reliability* - when an application crashes, it must not affect the OS
  - *security* - protect other applications and the OS from malicious applications
    - trusted code vs. untrusted code
  - *privacy* - on a multi-user system, one user must not be able to access information of another user
  - *fair resource allocation* - an application must not be allowed to use an unfair amount of shared resources (e.g., CPU time, memory, disk space, etc.)

- Implementing protection is the job of the OS kernel

Copyright © William C. Cheng

Challenge: Protection

- The OS kernel is the lowest level of software running on the system and has full access to all machine hardware
  - must trust the OS kernel to do anything with the hardware
  - everything else is untrusted and must run in a restricted environment



Copyright © William C. Cheng

Main Points

- Process concept
  - a process is the OS abstraction for executing a program with limited privileges
- *Dual-mode* operation: user vs. kernel
  - *kernel-mode:* execute with complete privileges
  - *user-mode:* execute with fewer privileges
- Safe control transfer
  - how do we switch from one mode to the other?

Copyright © William C. Cheng

Process Concept

- A process is the execution of an application program with restricted rights
  - the process is the abstraction for protected execution provided by the OS kernel
  - a process needs permission from the OS kernel before accessing memory of any other process, before reading/writing to disk, before changing hardware settings, etc.

- The OS kernel runs directly on the processor with unlimitted rights
  - what about applications?
  - in order to have *good performance*, applications also need to run directly on the processor
    - but with all potentially dangerous operations *disabled*
  - hardware can help to improve performance
    - in general, the more hardware help the better/faster
      - how much can and should hardware help?

Copyright © William C. Cheng

Dual-mode Operation

- Split personality of a process
  - when running the OS kernel, it's in charge of everything and can do anything it wants
  - when running applicatino code, it needs to ask permissions to do anything potentially harmful to other applications or the OS kernel
  - remember that they are running on the same processor, sometimes completely trustworthy and other times completely untrusted

Safe Control Transfer

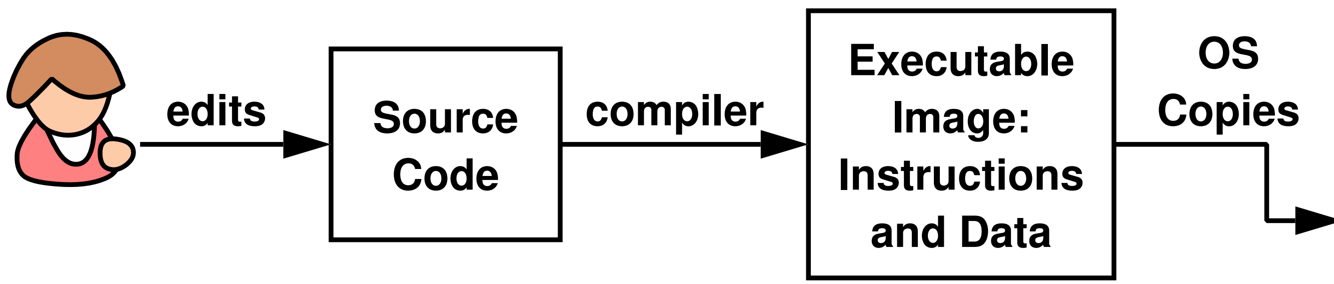
- Application to OS kernel: make *system call*
  - return from system call

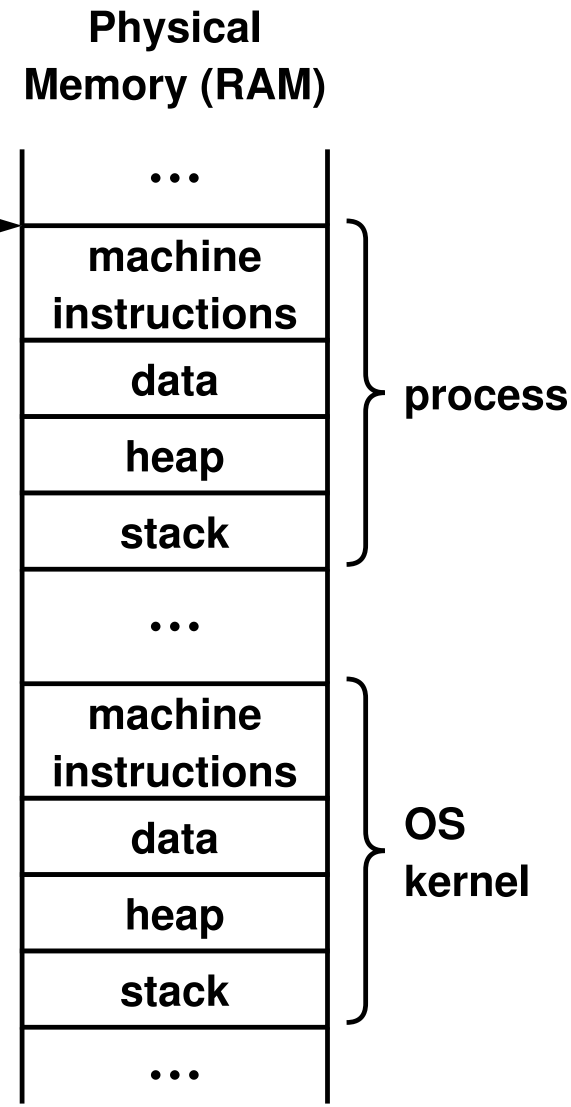
Copyright © William C. Cheng

(2.1) The Process Abstraction

Copyright © William C. Cheng

The Process Abstraction

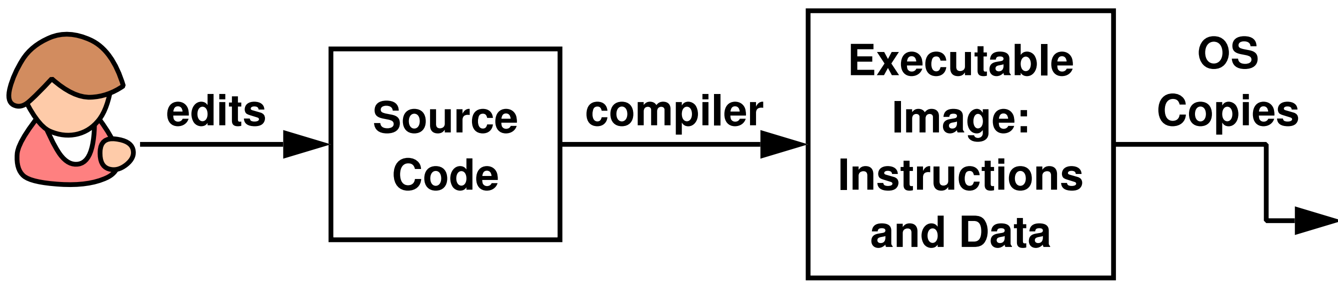


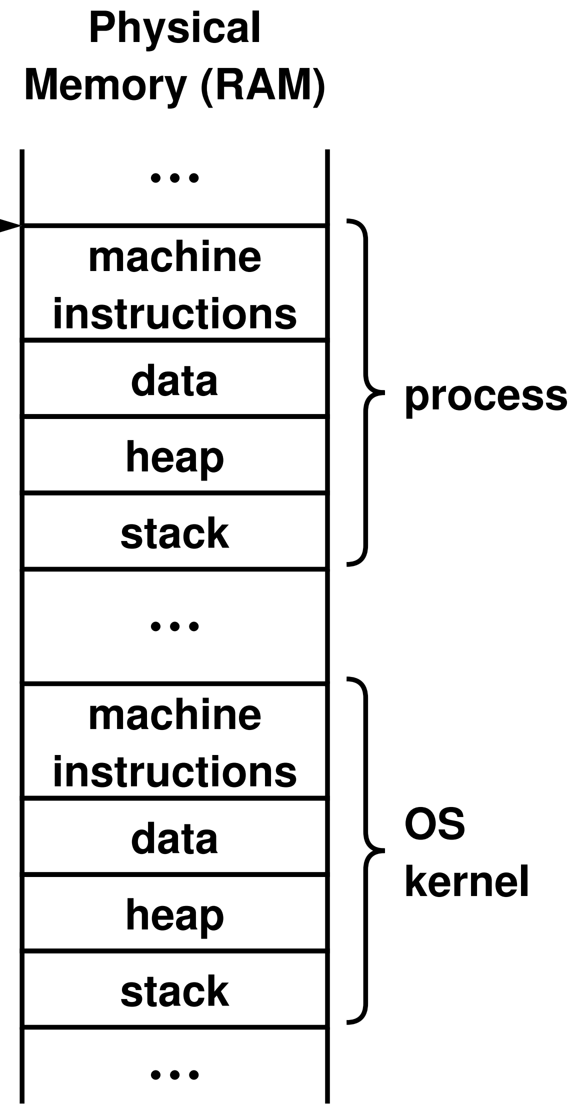


- A process is a running program
  - it has an *address space* that's made up of *memory segments*
    - machine instructions are kept inside the *text* segment
    - global variables are kept inside the *data* segment
    - the *heap* holds dynamically allocated data structures that the process might need
    - the *stack* holds the state of local variables and function arguments during procedure calls
  - conceptually, the OS kernel has its own address space

Copyright © William C. Cheng

The Process Abstraction

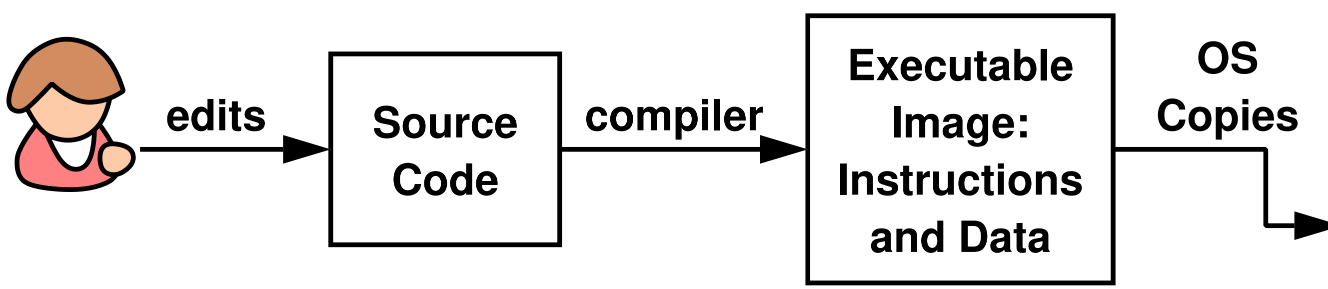


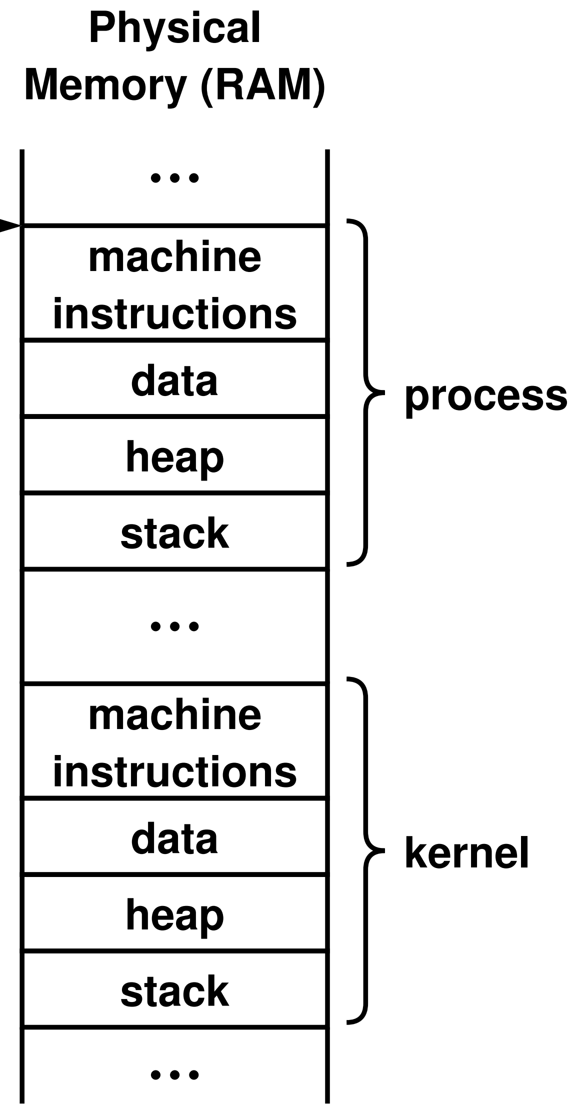


- What's to keep the process from modifying data in the OS kernel or other processes?
- What's to keep the process from modifying data on disk?

Copyright © William C. Cheng

The Process Abstraction





Solution:

- *process* concept
  - a process is the OS abstraction for executing a program with limited privileges
- dual-mode operation: *user* vs. *kernel*
  - kernel-mode: execute with complete privileges
  - user-mode: execute with fewer privileges
- safe control transfer
  - how do we switch from one mode to the other?

Copyright © William C. Cheng

The Process Abstraction

- *Process:* an *instance* of a program, running with limited rights
  - *thread:* a sequence of instructions within a process
    - potentially many threads per process (for now 1:1)
  - *address space:* set of rights of a process
    - memory that a process can access
    - other permissions the process has (e.g., what memory is shared with another process)

Copyright © William C. Cheng

Process Control Block (PCB)

- OS maintains information about every process in a data structure called a *Process Control Block (PCB)*
- PCB contains information such as:
  - where the process data (e.g., code, global variabls, stack, heap) is stored in memory
  - where its executable image resides on disk
  - which user asked to execute the program
  - what privileges the process has
  - ...

Copyright © William C. Cheng

(2.2) Dual-Mode Operation

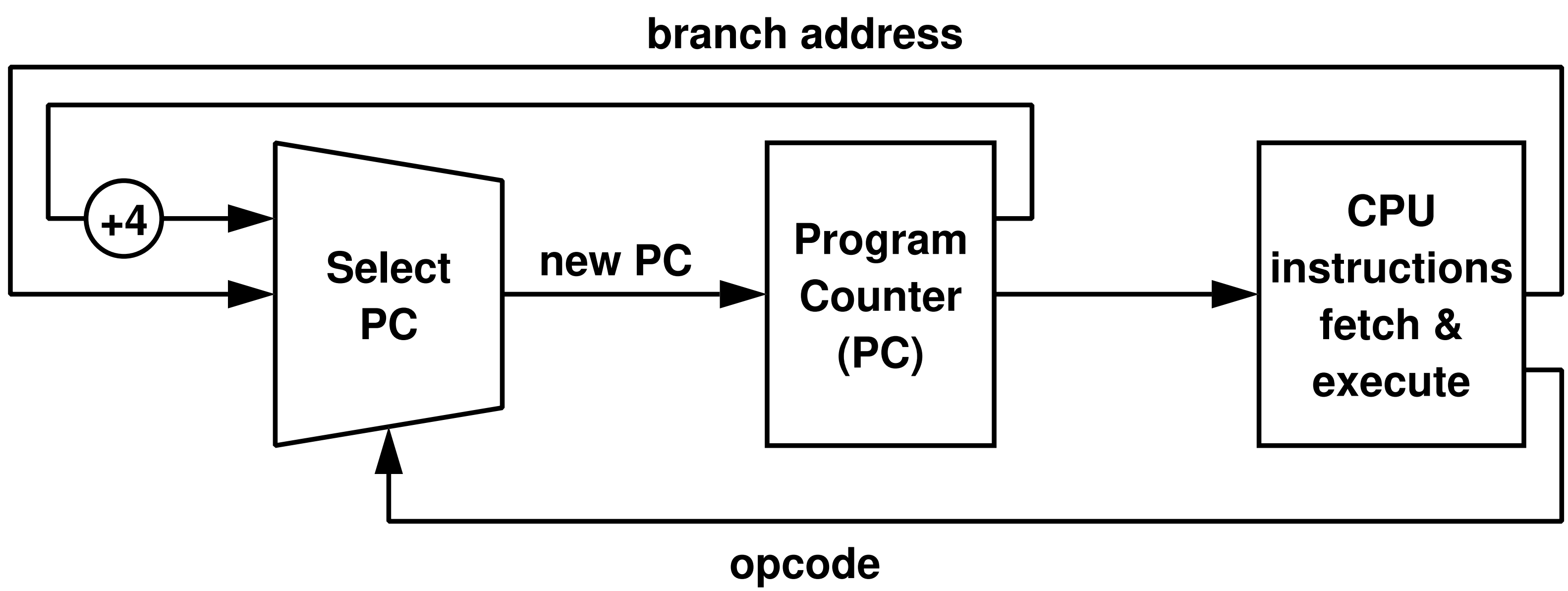
Copyright © William C. Cheng

(2.2) Dual-Mode Operation

- Kernel mode
  - execution with the full privileges of the hardware
  - read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- User mode
  - limited privileges
  - only those granted by the operating system kernel
- On the x86, mode stored in EFLAGS register
- On the MIPS, mode in the status register

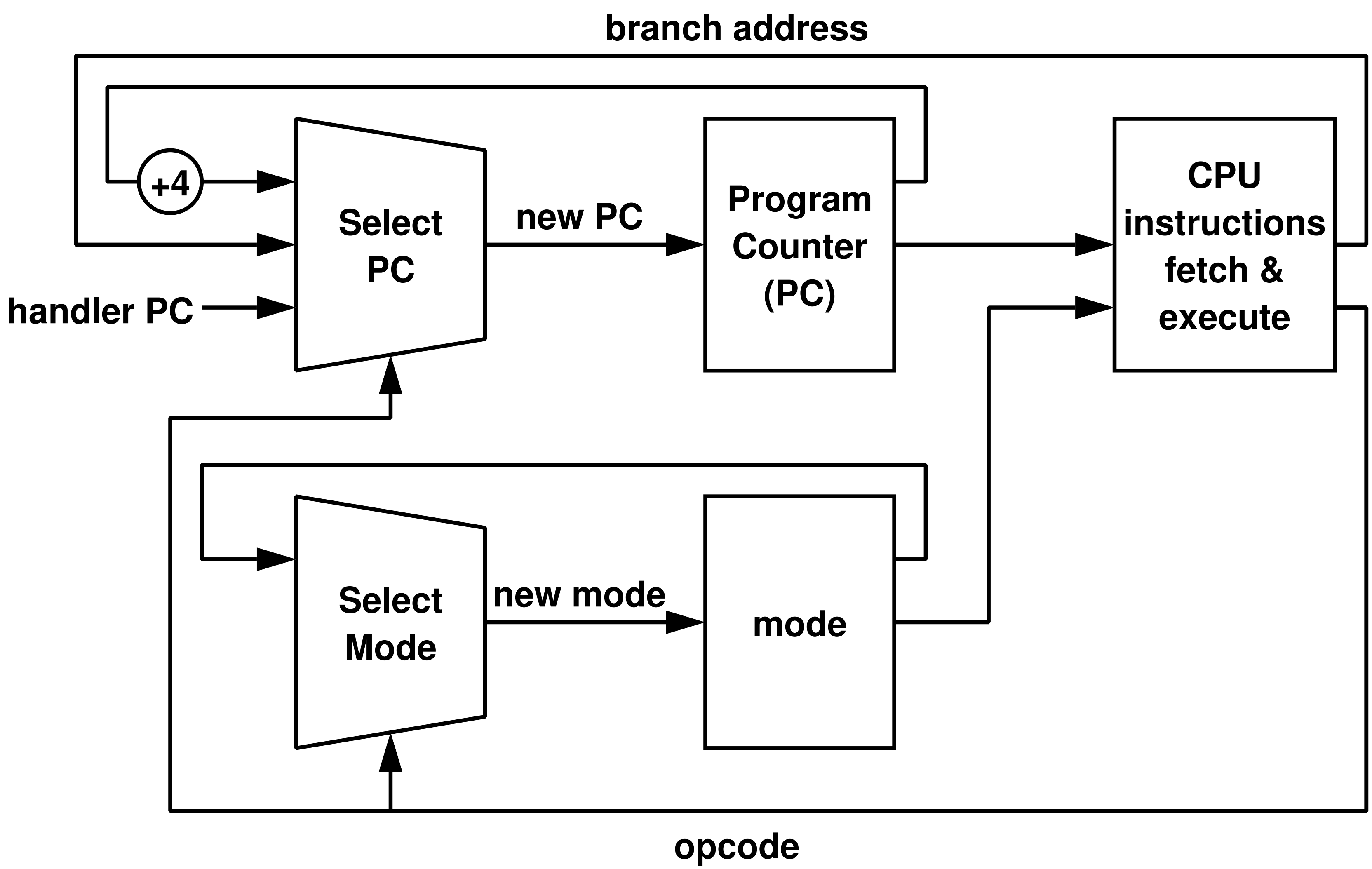
Copyright © William C. Cheng

A Model of a CPU



Copyright © William C. Cheng

A CPU with Dual-Mode Operation



Copyright © William C. Cheng

A CPU with Dual-Mode Operation

- Privileged instructions
  - available to kernel
  - not available to user code
- Limits on memory accesses
  - to prevent user code from overwriting the kernel
- Timer
  - to regain control from a user program in a loop
- Safe way to switch from user mode to kernel mode, and vice versa

Copyright © William C. Cheng

Privileged Instructions

- What would be an example of a privileged instruction?
  - change mode bit in EFLAGS register
  - change which memory location a user program can access
  - send commands to I/O devices
  - read data from or write data to I/O devices
  - jump into kernel code

Copyright © William C. Cheng

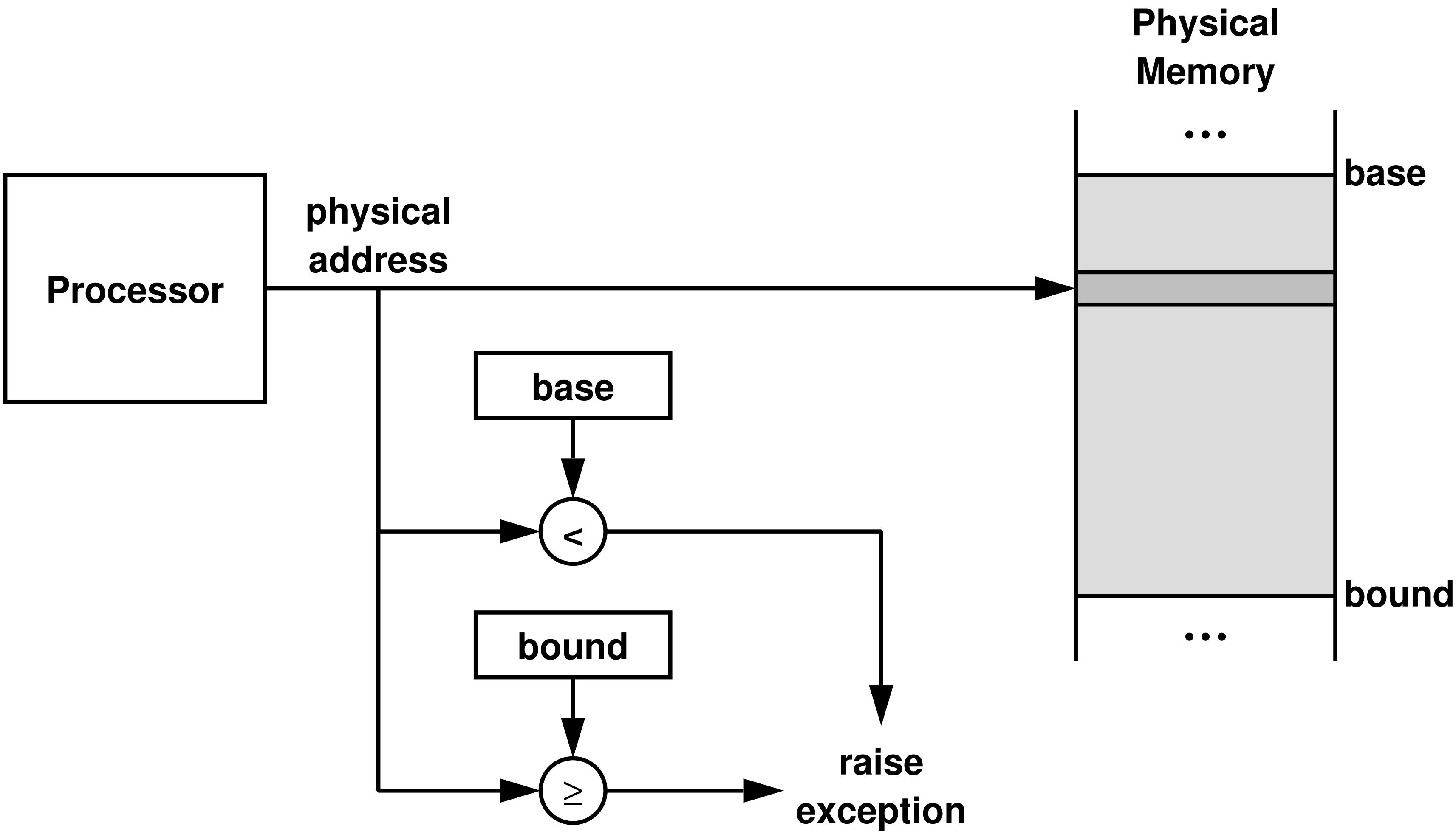
Privileged Instructions

- What should happen if a user program attempts to execute a privileged instruction?
  - would cause a *processor exception* (in hardware)
    - which would cause the processor to transfer control to an *exception handler* in the OS kernel
      - usually, the kernel simply halts the process after a privilege violation

- What bad thing can happen if an application can jump into kernel mode at any location in the kernel?
  - it may crash the kernel
  - it may allow the application to access privileged data
  - it may allow the application to bypass security checks

Copyright © William C. Cheng

Simple Memory Protection: Base & Bound Registers



- Can only modify these registers using privileged instructions
  - otherwise, application can access data that belongs to the kernel or other processes

Copyright © William C. Cheng

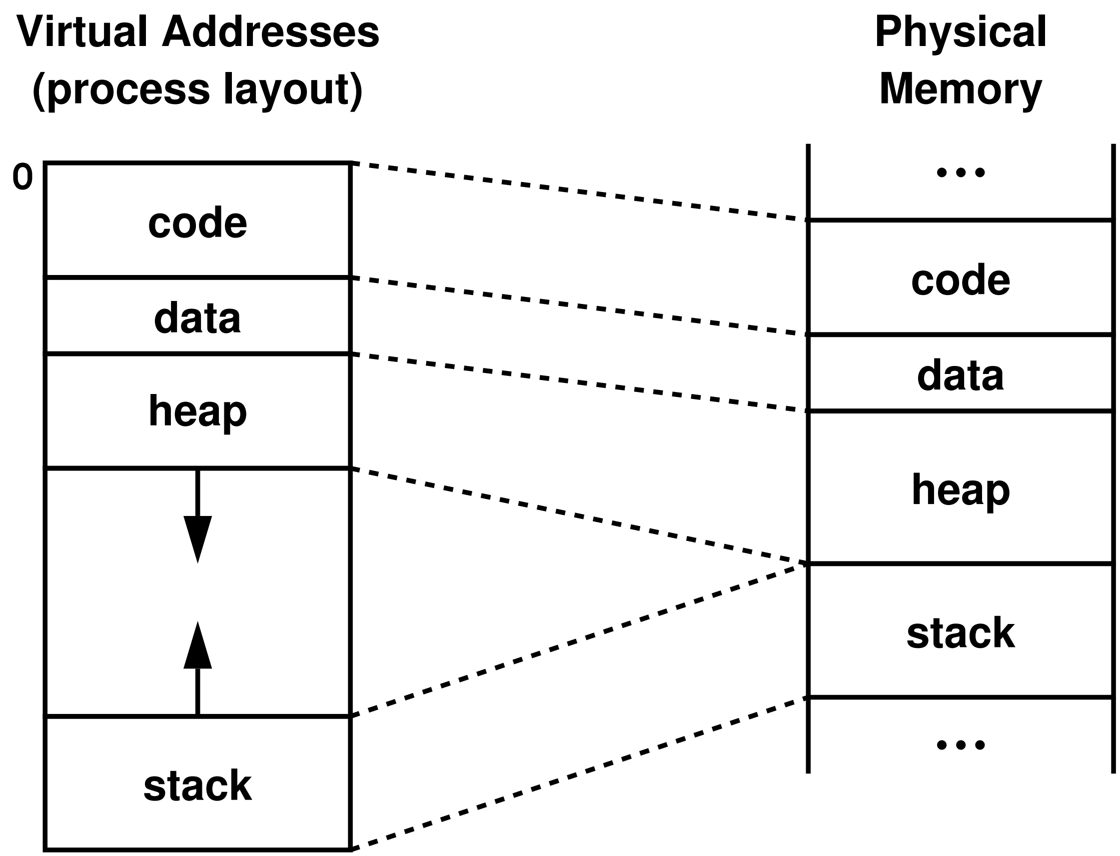
Towards Virtual Addresses

- What's are the problems with base and bound?
  - addresses used by an application must be contiguous (cannot have gaps)
    - it would be nice if the stack and heap can grow and shrink
  - cannot share code between processes
  - absolute addresses are difficult to use
    - how to load the same program at two different memory locations?
      - e.g., "`jmp 0x12345678`"
  - memory fragmentation

Copyright © William C. Cheng

Virtual Addresses

- Translation done in hardware (on every address), using a table set up by the OS kernel
  - the (virtual) memory of every process starts at the same place, i.e., 0
    - these memory "segments" can be located anywhere in physical memory



Copyright © William C. Cheng

Virtual Address Example

```
int staticVar = 0; // a static variable
main() {
   staticVar += 1;
   sleep(10); // sleep for 10 seconds
   printf("static address: %x, value: %d\n",
        &staticVar, staticVar);
}
```

- If you run two instances of this program simultaneously, you would get the same printout from them if *virtual addresses* are used
  - if you don't have support for virtual addresses and have to use physical addresses, the printout will be different

Copyright © William C. Cheng

Virtual Address Example

```
int staticVar = 0; // a static vari
main() {
  staticVar += 1;
  sleep(10); // sleep for 10 second
  printf("static address: %x, value
      &staticVar, staticVar);
}
```

- my color codes for code
  - reserved/key words are in blue
  - numeric and string constants are in red
  - comments are in green
  - black otherwise

- If you run two instances of this program simultaneously, you would get the same printout from them if *virtual addresses* are used
  - if you don’t have support for virtual addresses and have to use physical addresses, the printout will be different

Copyright © William C. Cheng

Hardware Timer

- What if a programming bug causes a user process to go into an infinite loop and never give up the processor?
  - we need a way for the OS kernel to gain control periodically
- Hardware timer is a device that periodically interrupts the processor
  - returns control to the kernel handler
  - interrupt frequency set by the kernel
    - not by user code
    - expires every few milliseconds (human reaction time is a few hundred of milliseconds)
  - interrupts can be temporarily deferred
    - not by user code
    - interrupt deferral crucial for implementing mutual exclusion

Copyright © William C. Cheng

(2.3) Types of Mode Transfer

Copyright © William C. Cheng

Types of Mode Transfer

- User Mode To Kernel Mode
- Kernel Mode To User Mode

Copyright © William C. Cheng

Mode Switch: User Mode To Kernel Mode

- **Interrupts (if we don't say "software interrupt", we mean "hardware interrupt")**
  - triggered by timer and I/O devices
- **Exceptions**
  - triggered by unexpected program behavior
  - or malicious behavior!
  - e.g., divide by zero, execute a privileged instruction
  - when such an exception occurs, the thread in the user process *"traps"* into the kernel
- **System calls (aka protected procedure call)**
  - request by program for kernel to do some operation on its behalf
  - only limited number of very carefully coded *kernel entry points*
  - e.g., read data from disk, create another process
  - also *"trap"* into the kernel
  - when a thread in a user process makes a system call, it *"traps"* into the kernel

Copyright © William C. Cheng

Mode Switch: User Mode To Kernel Mode

- Interrupts (if we don't say "software interrupt", interrupt")
  - triggered by timer and I/O devices
- Exceptions
  - triggered by unexpected program behavior
  - or malicious behavior!
  - e.g., divide by zero, execute a privileged instruction
  - when such an exception occurs, the thread in the user process *"traps"* into the kernel
- System calls (aka protected procedure call)
  - request by program for kernel to do some operation on its behalf
  - only limited number of very carefully coded *kernel entry points*
  - e.g., read data from disk, create another process
  - also *"trap"* into the kernel
  - when a thread in a user process makes a system call, it *"traps"* into the kernel

> - *trap* is a *synchronous* event (to transfer from user mode to kernel mode)
> - *interrupt* is an *asynchronous* event (can happen at any time)

Copyright © William C. Cheng

Mode Switch: Kernel Mode To User Mode

- New process (or new thread) starts
  - jump to first instruction in program (or thread)
- Return from interrupt, exception, system call
  - resume suspended execution
- Process (or thread) context switch
  - resume some other process (or thread)
- User-level upcall (UNIX signal)
  - asynchronous notification to user program

Copyright © William C. Cheng

(2.4) Implementing Safe Mode Transfer

Copyright © William C. Cheng

Implementing Safe Mode Transfer

- Context switch code must be carefully crafted
  - relies on hardware support
- Most OS has a common sequence of instructions for enter the kernel and for returning to user level, regardless of cause
  - at a minimum, this common sequence must provide
    - limited entry into the kernel
      - an entry point must be set up by the kernel and not allow entry into the kernel at arbitrary points
    - *atomic* changes to processor state
      - processor mode, program counter, stack pointer, memory protection registers all change at the same time
    - transparent, restartable execution
      - an interrupt must be invisible to the user process (i.e., serviced transparently)
      - if an interrupt is serviced in the middle of an instruction execution, the CPU needs to be able to restart or finish the execution of that instruction seamlessly

Copyright © William C. Cheng

Common Interrupt / Exception Handling

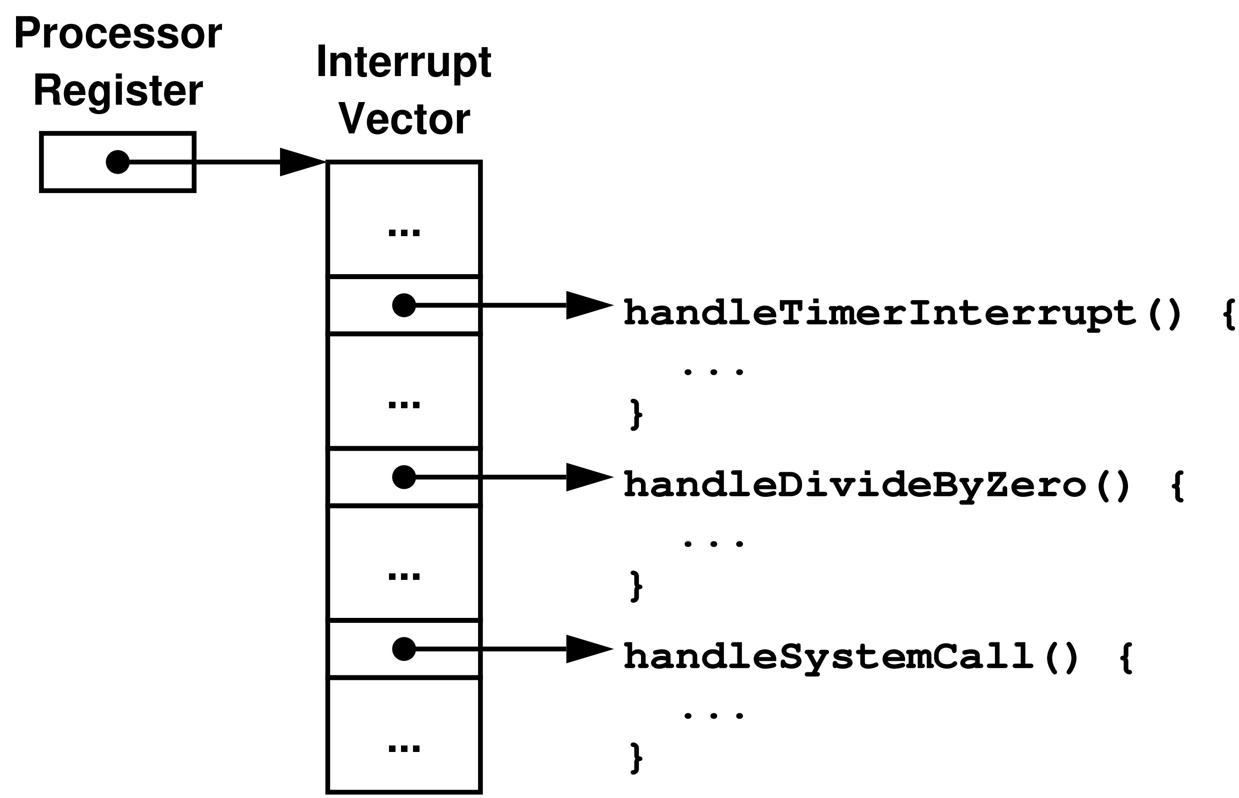
- On an interrupt (or exception), the following happens
  1) the processor saves its current state to memory
  2) further events are deferred
  3) changes to kernel mode
  4) jump to the interrupt or exception handler

- When the handler finishes, the steps are reversed and the processor state is restored from its saved location
  - the interrupted entity has no idea that an interrupt has occurred

Copyright © William C. Cheng

Interrupt Vector (Table)

- Table set up by OS kernel
  - the interrupt vector is an array of function pointers, pointing to code to run on different events
  - a special purpose processor register stores the address of this array



Copyright © William C. Cheng

Interrupt Stack

- On most processors, a special hardware register points to an *interrupt stack*
  - when an interrupt or a trap causes a context switch into the kernel, the hardware changes the stack pointer to point to the interrupt stack
    - the hardware automatically saves some of the interrupted thread's registers by pushing them onto the interrupt stack before calling the handler
- Why can't you use the process's user-level stack to store the saved state?
  - user-level stack pointer may be invalid (malicious user)
  - on a multi-processor system, another thread in the same process that runs in a different processor may modify the saved state

Copyright © William C. Cheng