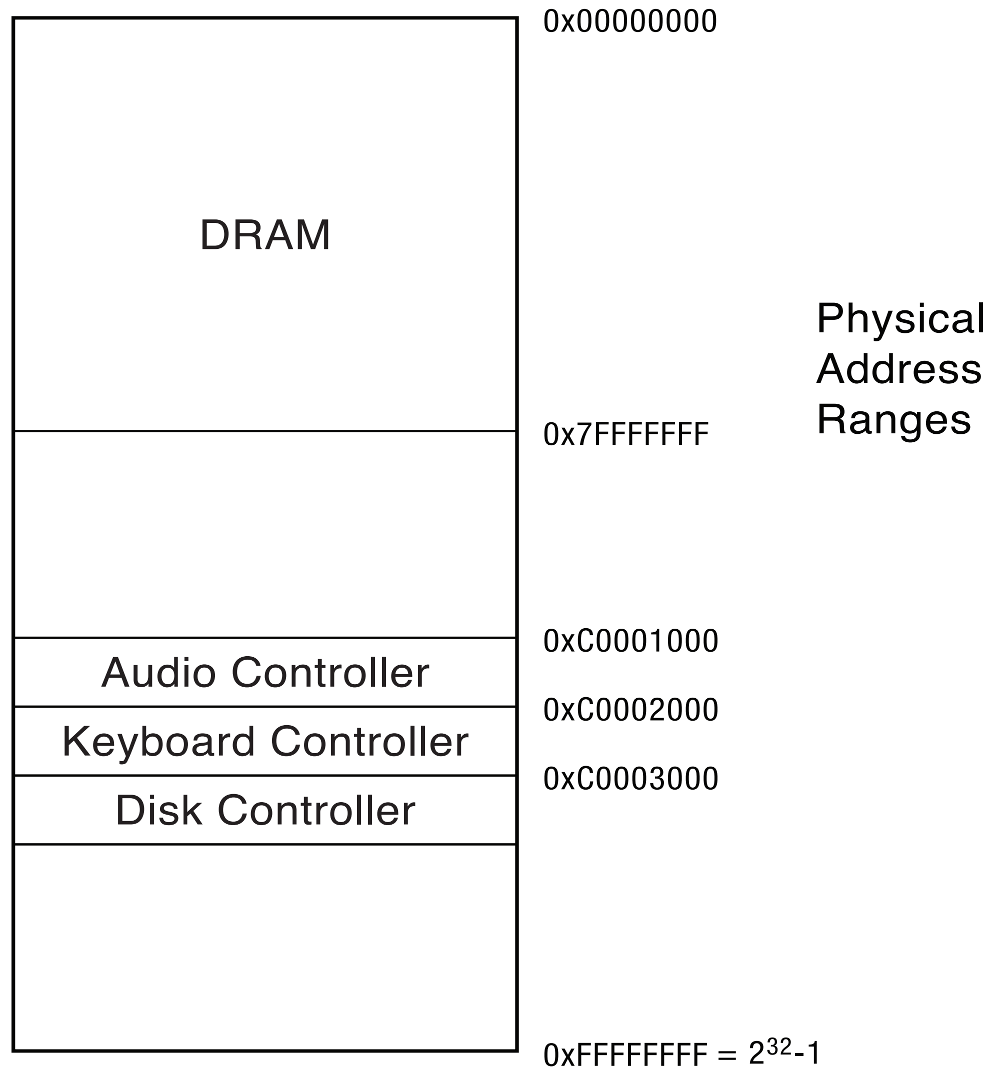
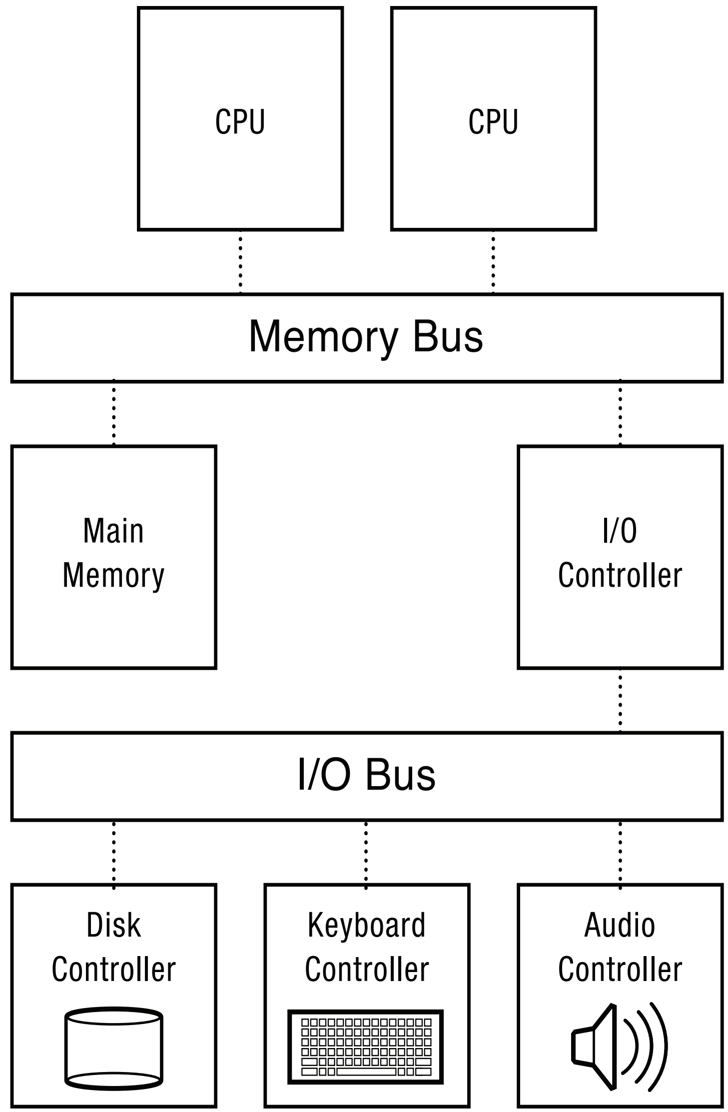
The Kernel Abstraction

Physical Address Layout



Device I/O

- Polling vs. interrupts
- Programmed I/O vs. DMA
- One operation at a time vs. queue of operations



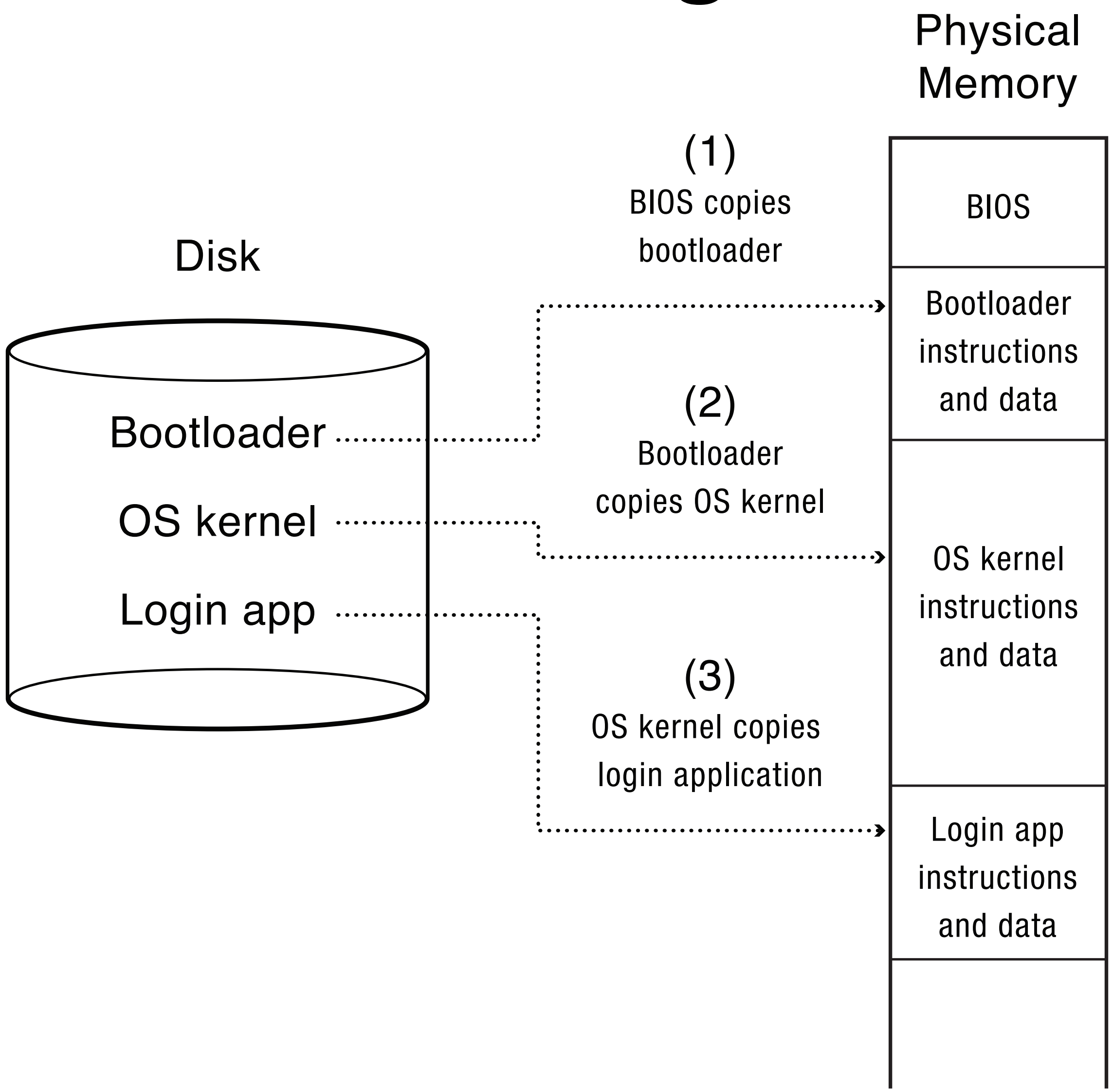
Question

- What (hardware, software) do you need to be able to run an untrustworthy application?

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?

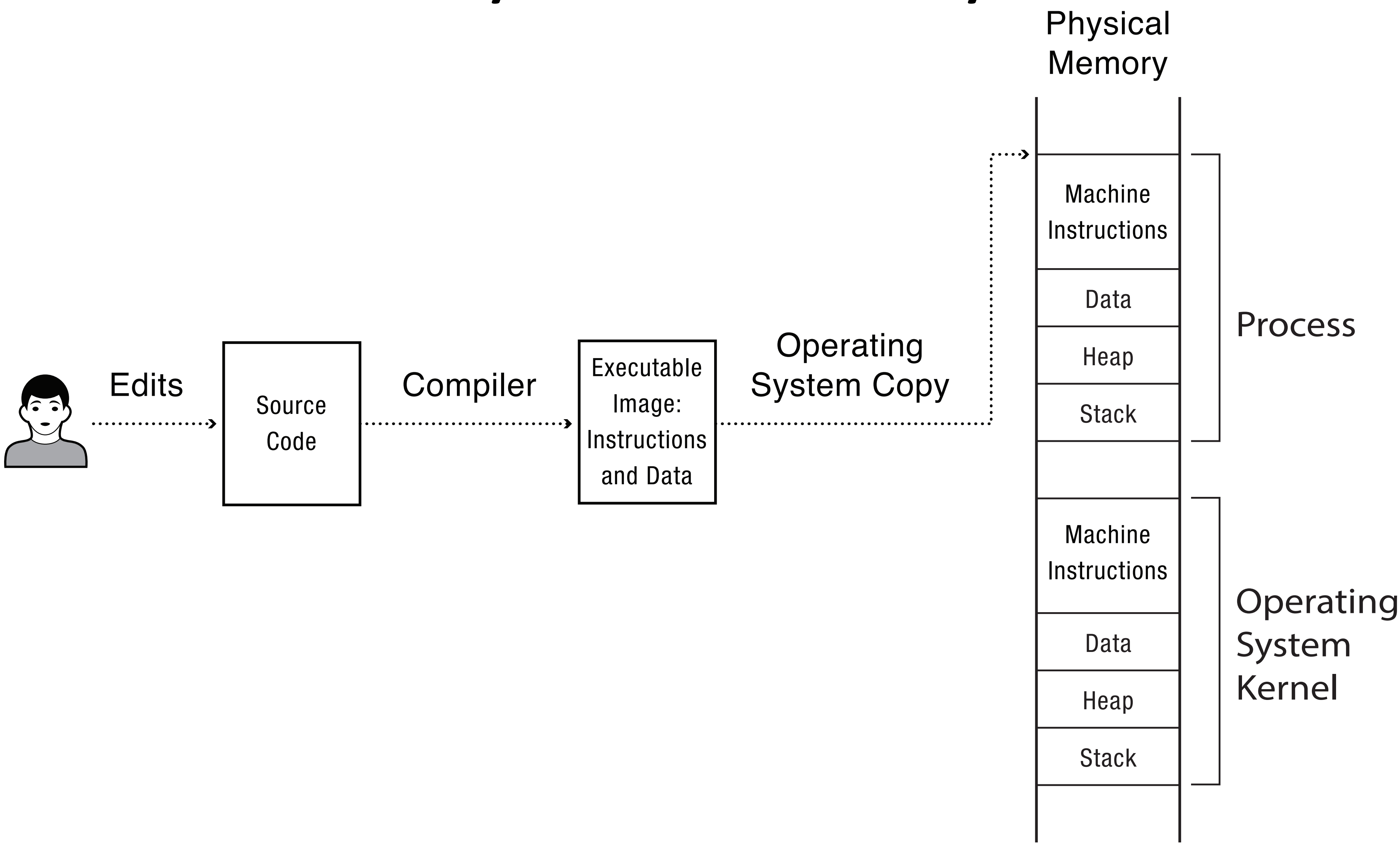
Booting


Physical Memory
Disk
(1)
BIOS copies bootloader
BIOS
Bootloader instructions and data
Bootloader
OS kernel
Login app
(2)
Bootloader copies OS kernel
OS kernel instructions and data
(3)
OS kernel copies login application
Login app instructions and data

Challenge: Protection

- How do we execute code with restricted privileges?
  - Either because the code is buggy or if it might be malicious
- Some examples:
  - A script running in a web browser
  - A program you just downloaded off the Internet
  - A program you just wrote that you haven't tested yet

Physical Memory



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 the process can access
    - Other permissions the process has (e.g., which system calls it can make, what files it can access)

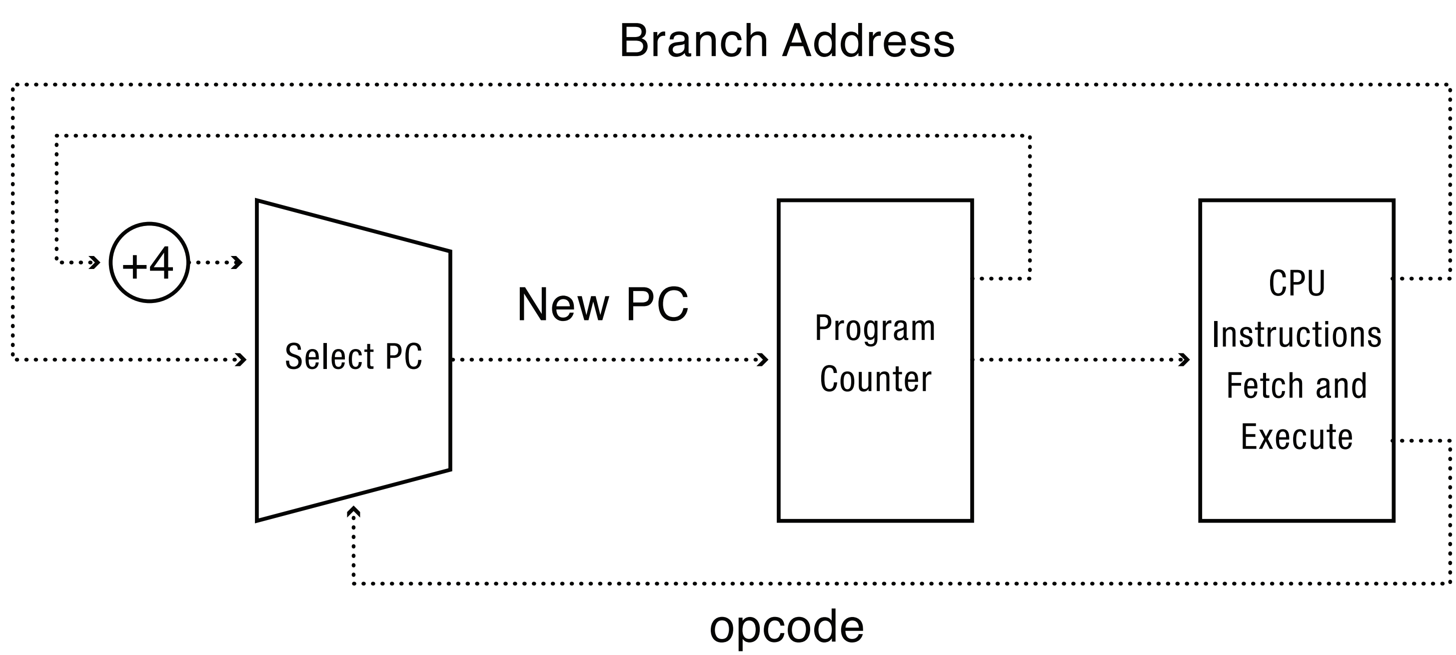
Thought Experiment

- How can we implement execution with limited privilege?
  - Execute each program instruction in a simulator
  - If the instruction is permitted, do the instruction
  - Otherwise, stop the process
  - Basic model in Javascript and other interpreted languages
- How do we go faster?
  - Run the unprivileged code directly on the CPU!

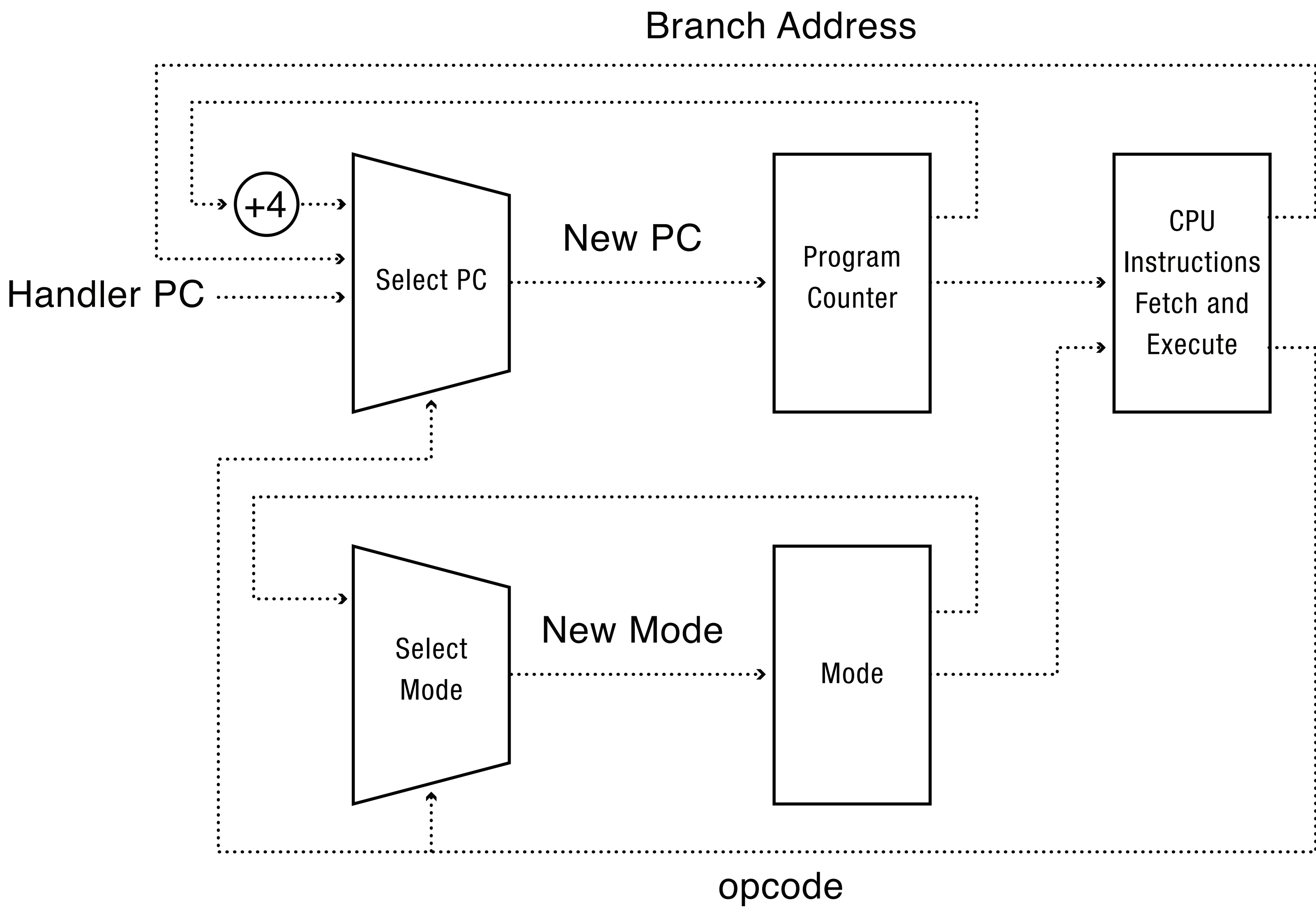
Dual-Mode Operation

- 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

A Model of a CPU



A CPU with Dual-Mode Operation



Hardware Support for 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

Privileged instructions

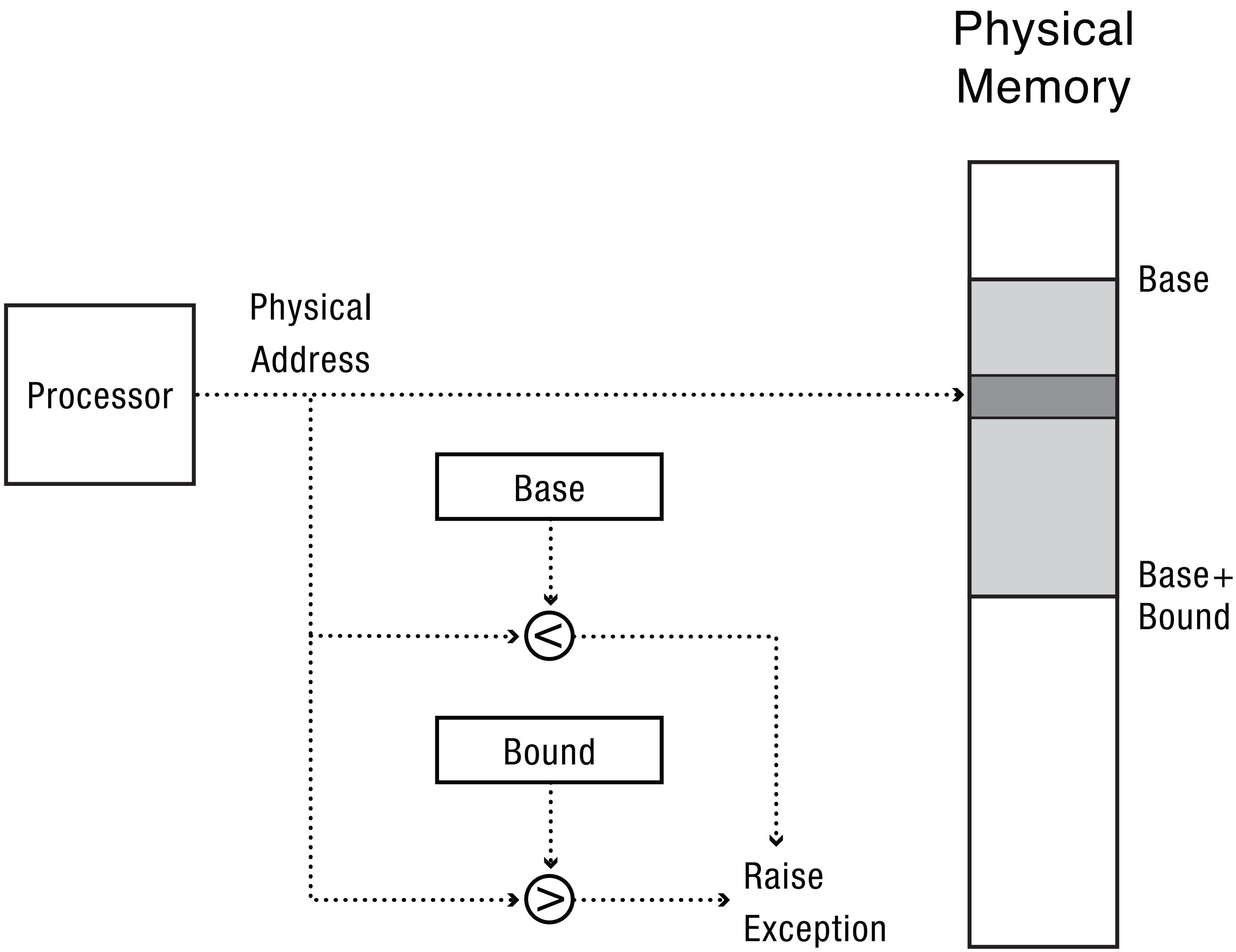
- Examples?

- What should happen if a user program attempts to execute a privileged instruction?

Question

- For a “Hello world” program, the kernel must copy the string from the user program memory into the screen memory.
- Why not allow the application to write directly to the screen’s buffer memory?

Simple Memory Protection

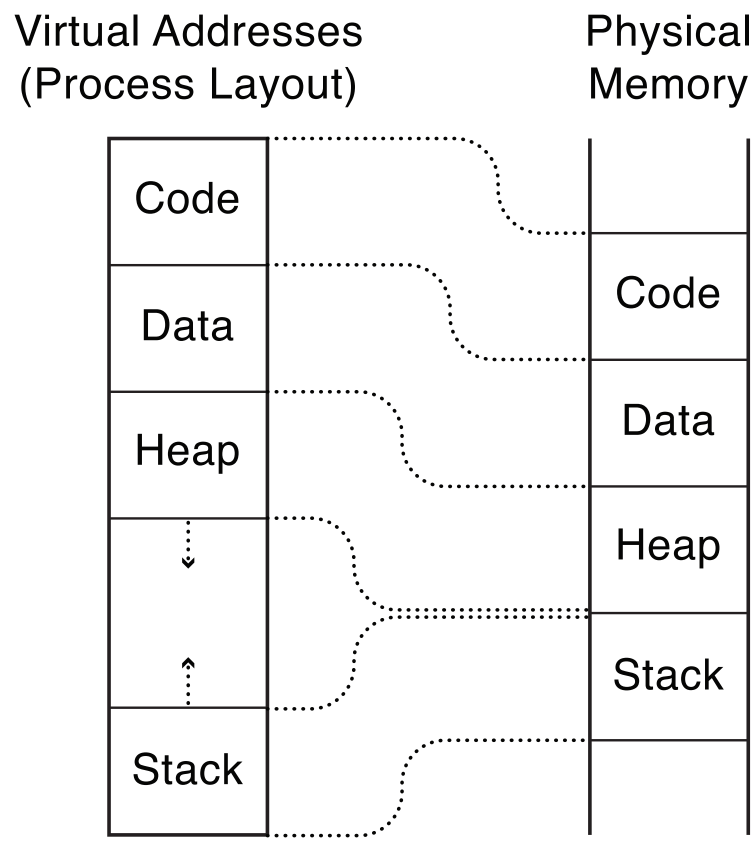


Towards Virtual Addresses

- Problems with base and bounds?

Virtual Addresses

- Translation done in hardware, using a table
- Table set up by operating system kernel



Virtual Address Example

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

What happens if we run two instances of this program at the same time?

What if we took the address of a procedure local variable in two copies of the same program running at the same time?

Virtual Address != Physical Address

- The same virtual address in two different processes can refer to different physical addresses. Why?
- The same virtual address in two different processes can refer to the same physical address. Why?
- Different virtual addresses can refer to the same physical address. Why?

Question

- Suppose you have a type-safe object-oriented language. If the OS only ran programs written in that language, would it still need hardware memory address protection?
  - Hint: who do you trust?
- What if the contents of every object were encrypted (with a checksum to detect corruption) when it wasn't running?
  - For when you get serious about ADTs

Hardware Timer

- Hardware device that periodically interrupts the processor
  - Returns control to the kernel handler
  - Interrupt frequency set by the kernel
    - Not by user code!
  - Interrupts can be temporarily deferred
    - Not by user code!
    - Interrupt deferral crucial for implementing mutual exclusion

User->Kernel Mode Switch

- From user mode to kernel mode
  - Interrupts
    - Triggered by timer and I/O devices
    - Can occur in user or kernel mode
  - Exceptions
    - Triggered by unexpected or malicious program behavior
  - System calls (protected procedure call)
    - Request for kernel to do some operation on behalf of app
    - Only limited # of very carefully coded entry points
- Trap: user->kernel mode switch

Question

- Examples of exceptions

- Examples of system calls

Kernel->User Mode Switch

- From kernel mode to user mode
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall (UNIX signal)
    - Asynchronous notification to user program

Restoring User State

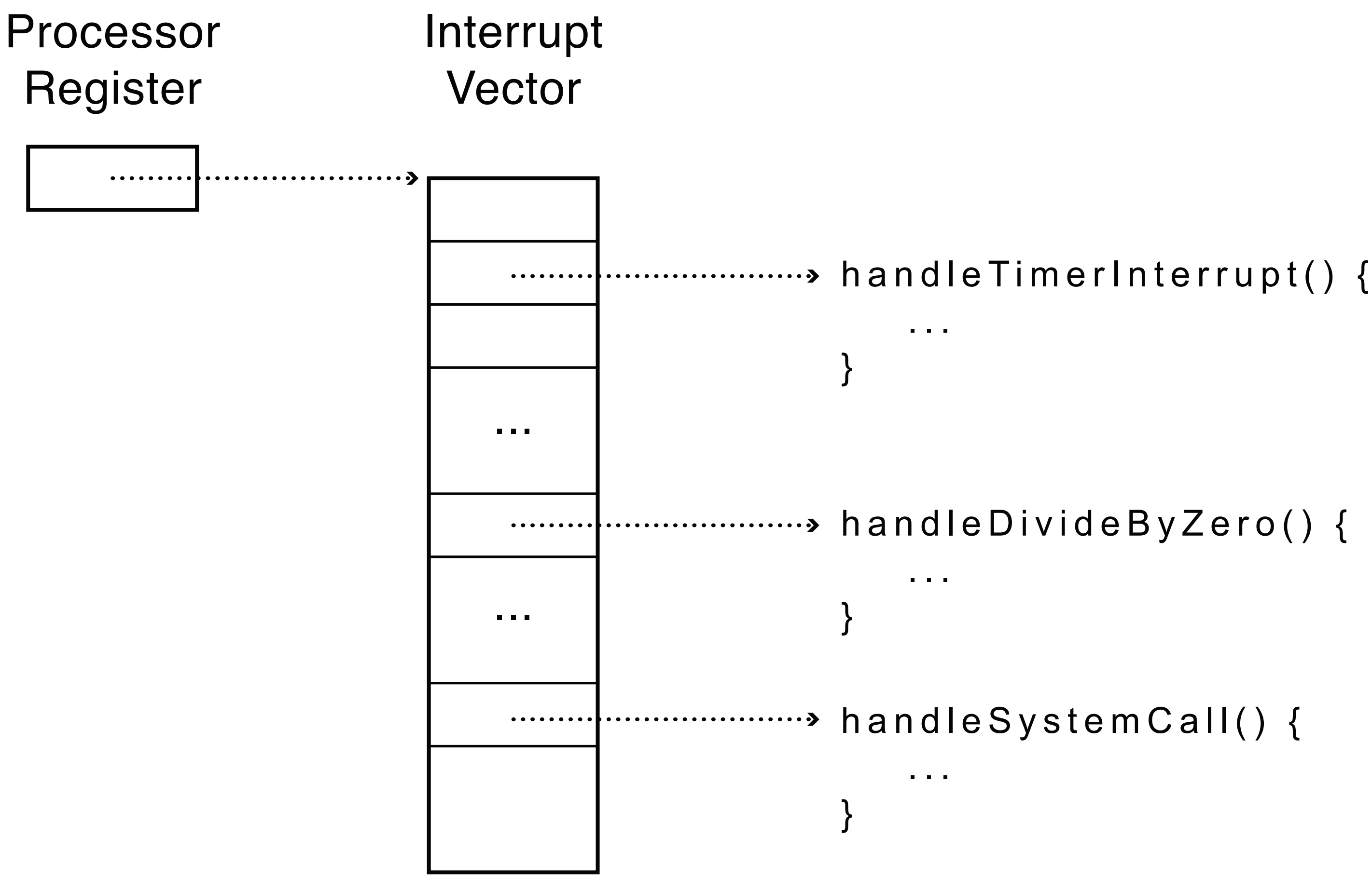
- We need to be able to interrupt and transparently resume the execution of a user program for several reasons:
  – I/O device signals I/O completion
  – Periodic hardware timer to check if app is hung
  – Multiplexing multiple apps on a single CPU
  – App unaware it has been interrupted!

How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Atomic transfer of control
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

Interrupt Vector

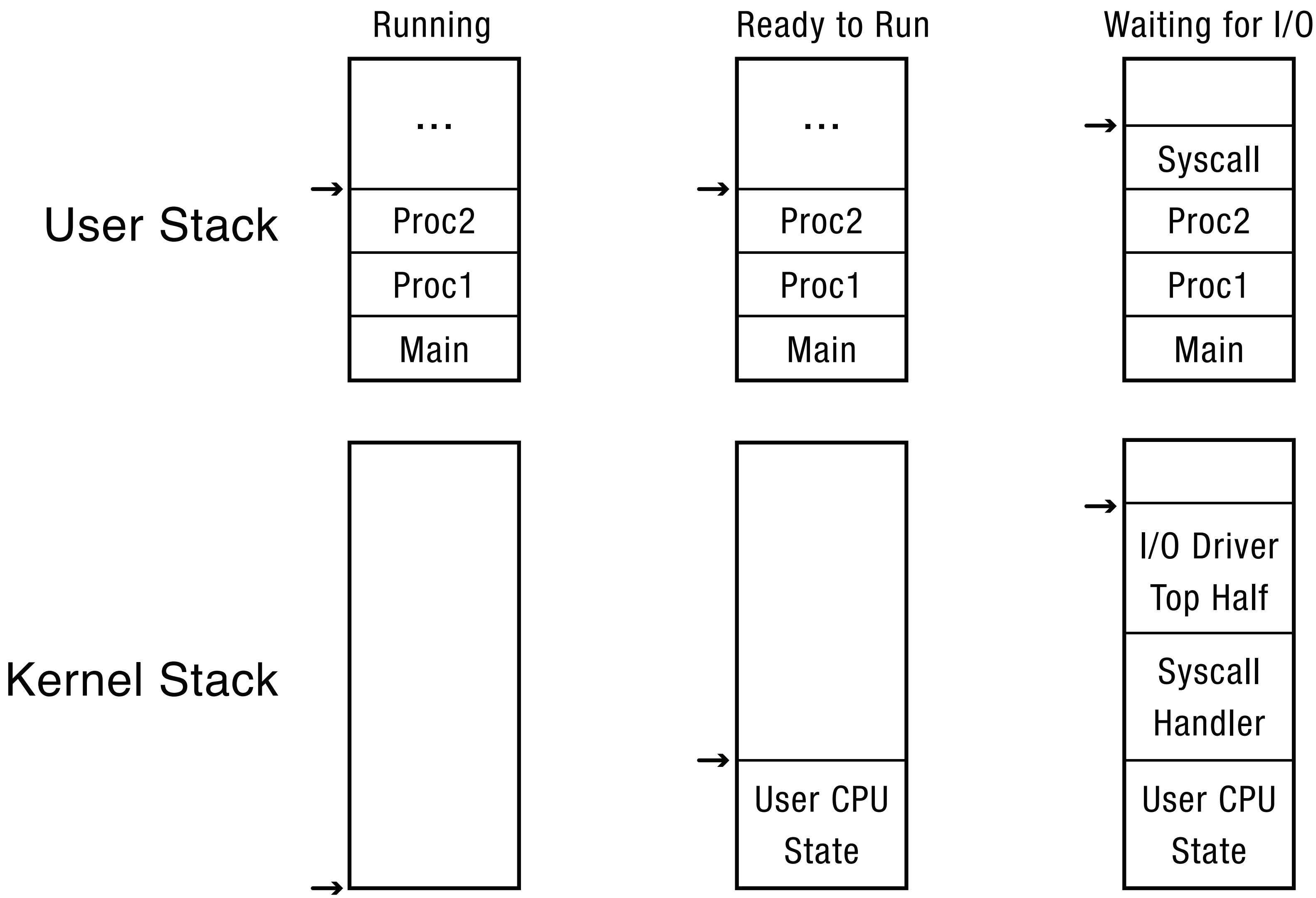
- Table set up by OS kernel; pointers to code to run on different events



Interrupt Stack

- Per-processor, located in kernel (not user) memory
  - Usually a process/thread has both: kernel and user stack
- Why can’t the interrupt handler run on the stack of the interrupted user process?

Interrupt Stack



Interrupt Masking

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - On x86
    - CLI: disable interrrupts
    - STI: enable interrupts
    - Only applies to the current CPU (on a multicore)
- We'll need this to implement synchronization in chapter 5

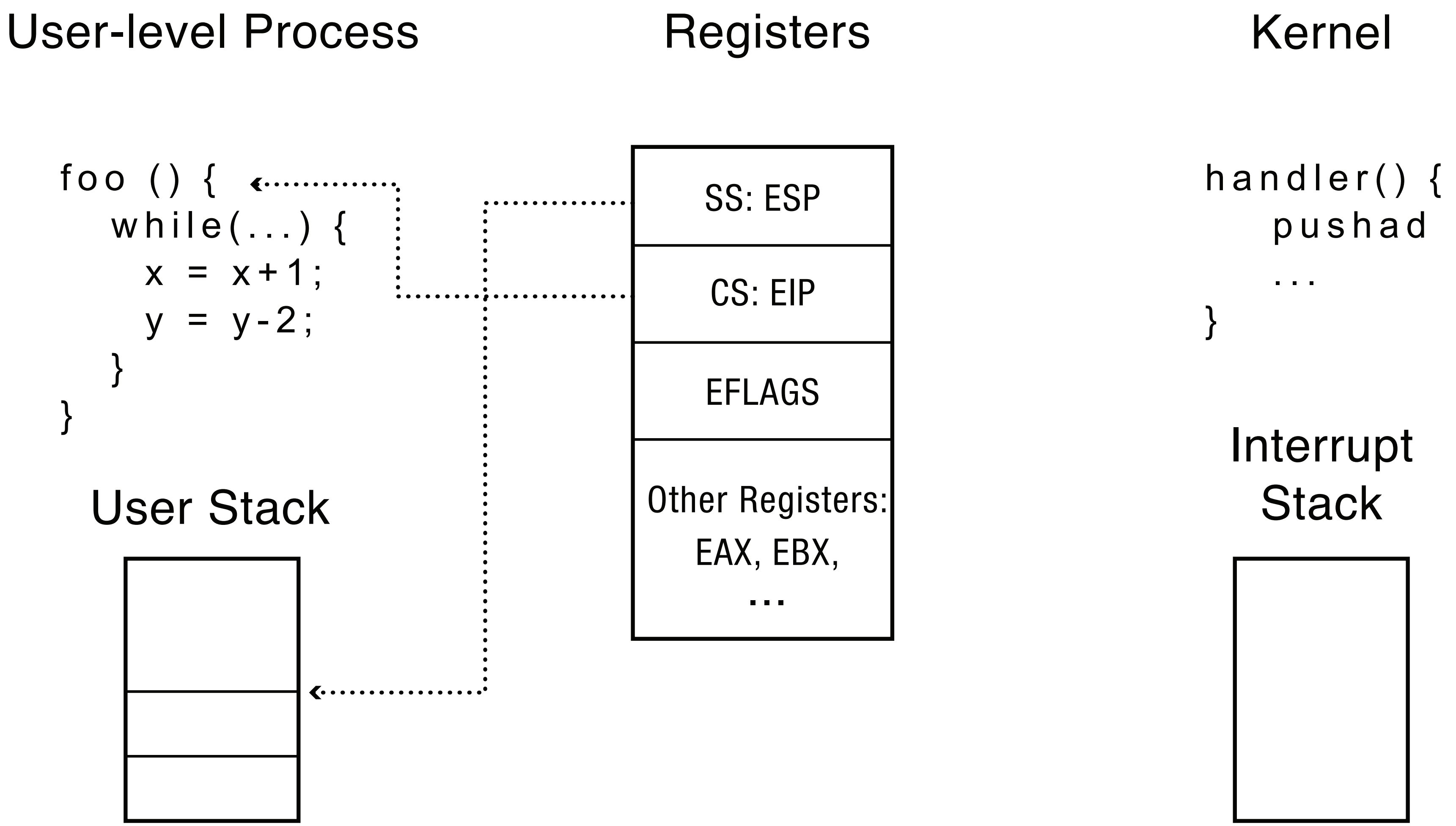
Interrupt Handlers

- Non-blocking, run to completion
  - Minimum necessary to allow device to take next interrupt
  - Any waiting must be limited duration
  - Wake up other threads to do any real work
    - Linux: semaphore
- Rest of device driver runs as a kernel thread

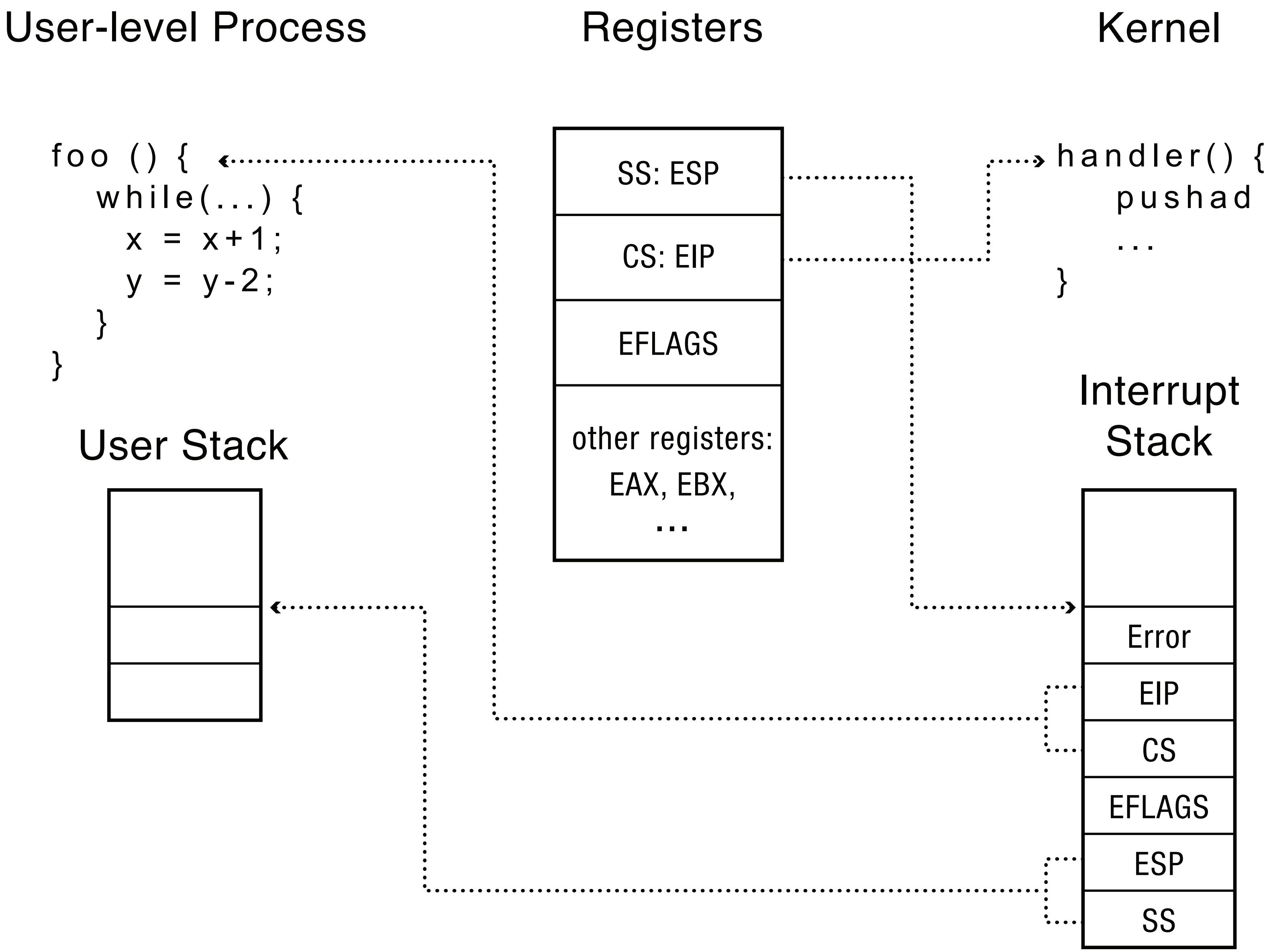
Case Study: x86 Interrupt (Hardware Support)

- Hardware saves current stack pointer
- Saves current program counter
- Saves current processor status word (condition codes)
- Switches to kernel stack
- Puts SP, PC, PSW on stack
- Switches to kernel mode
- Vectors through interrupt table
- Interrupt handler saves registers it might clobber

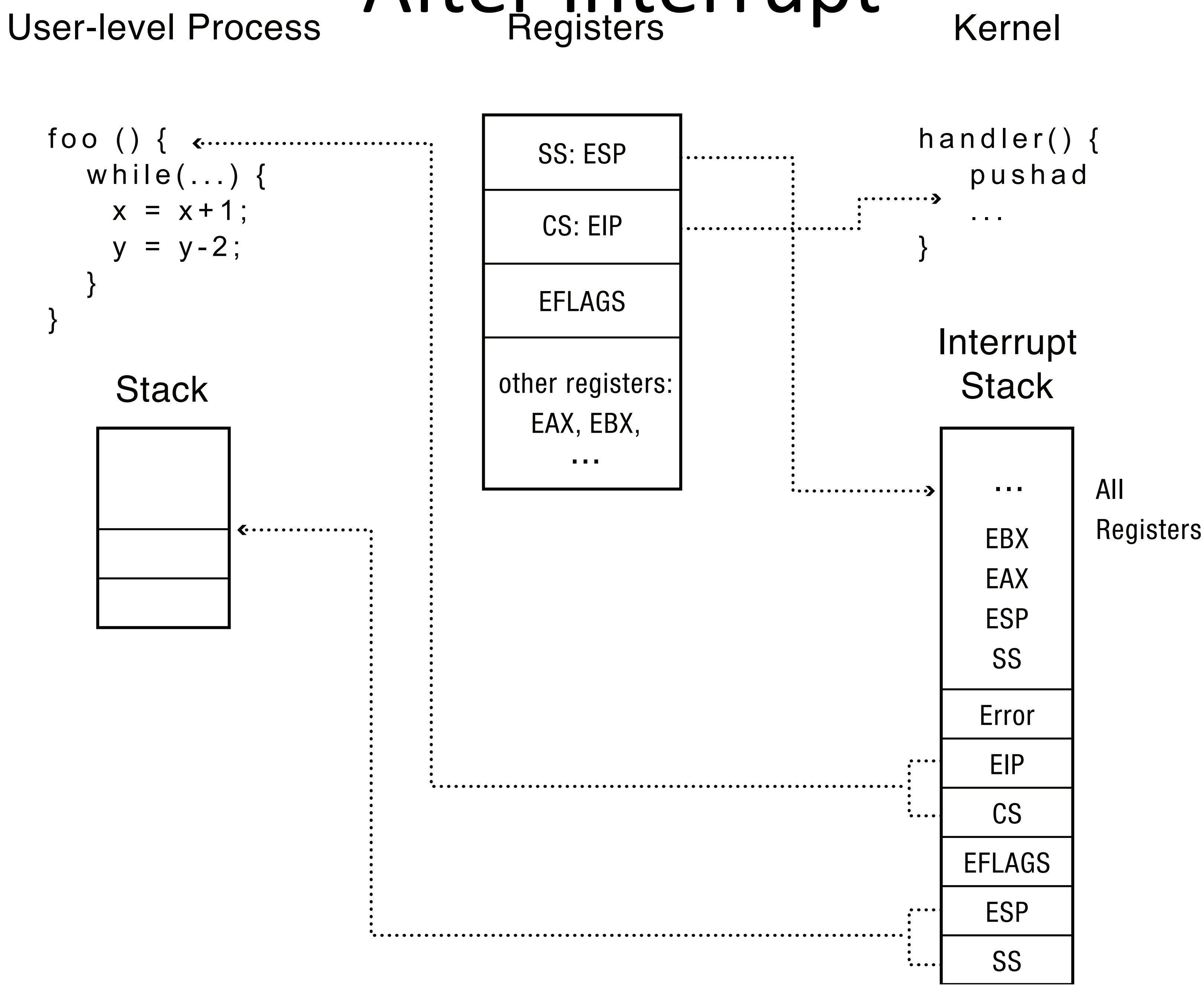
Before Interrupt



During Interrupt



After Interrupt



Question

- Why is the stack pointer saved twice on the interrupt stack?
  - Hint: is it the same stack pointer?

At end of handler

- Handler restores saved registers
- Atomically return to interrupted process/ thread
  - Restore program counter
  - Restore program stack
  - Restore processor status word/condition codes
  - Switch to user mode

Question

- Suppose the OS over-writes a value in the trapframe. What happens when the handler returns?

- Why might the OS want to do this?

Question

- The trapframe is stored on the interrupt stack; where is it stored after a context switch to a different process?

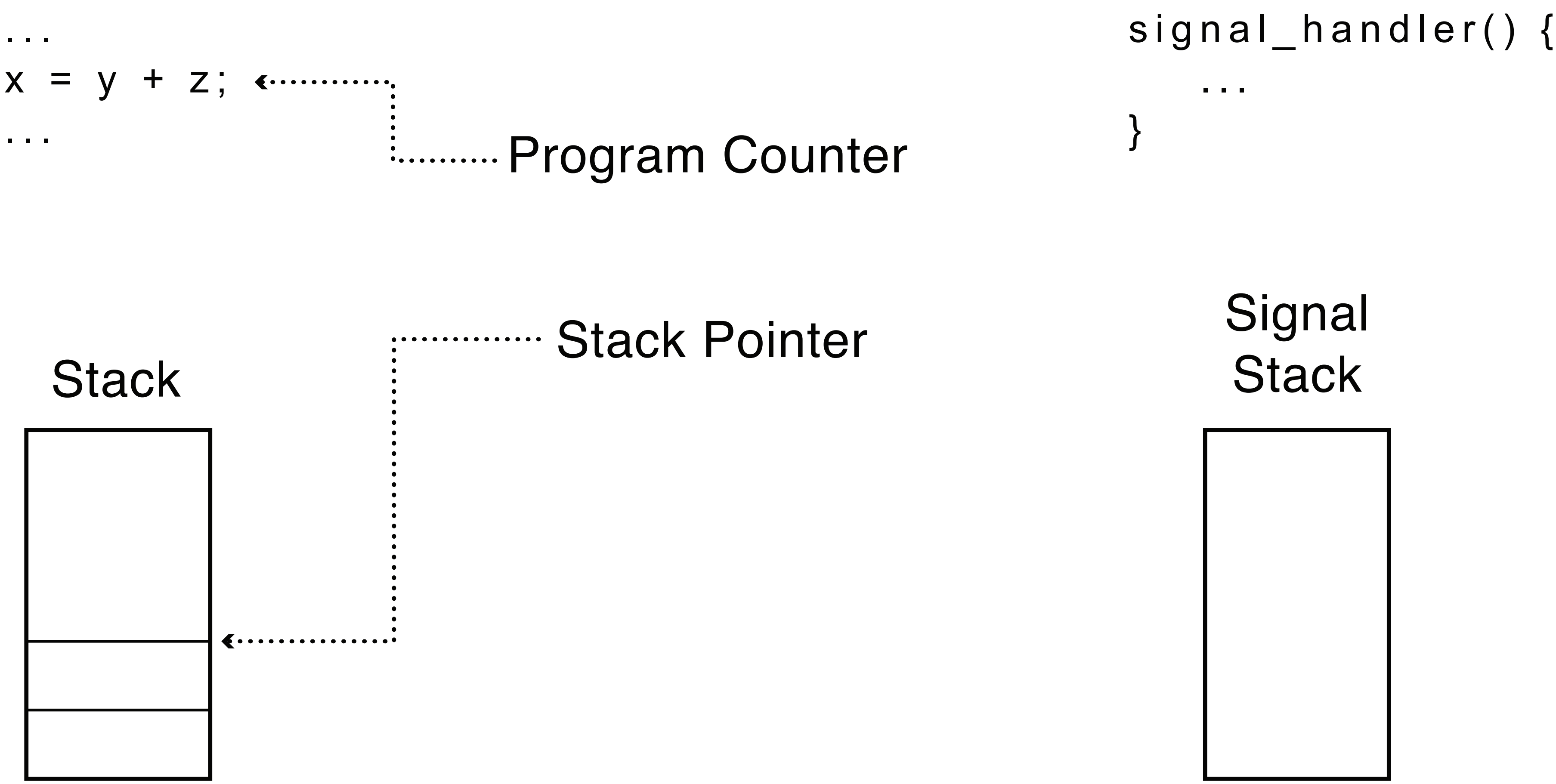
Upcall: User-level event delivery

- Notify user process of some event that needs to be handled right away
  - Time expiration
    - Real-time user interface
    - Time-slice for user-level thread manager
  - Interrupt delivery for VM player
  - Asynchronous I/O completion (async/await)
- AKA UNIX signal

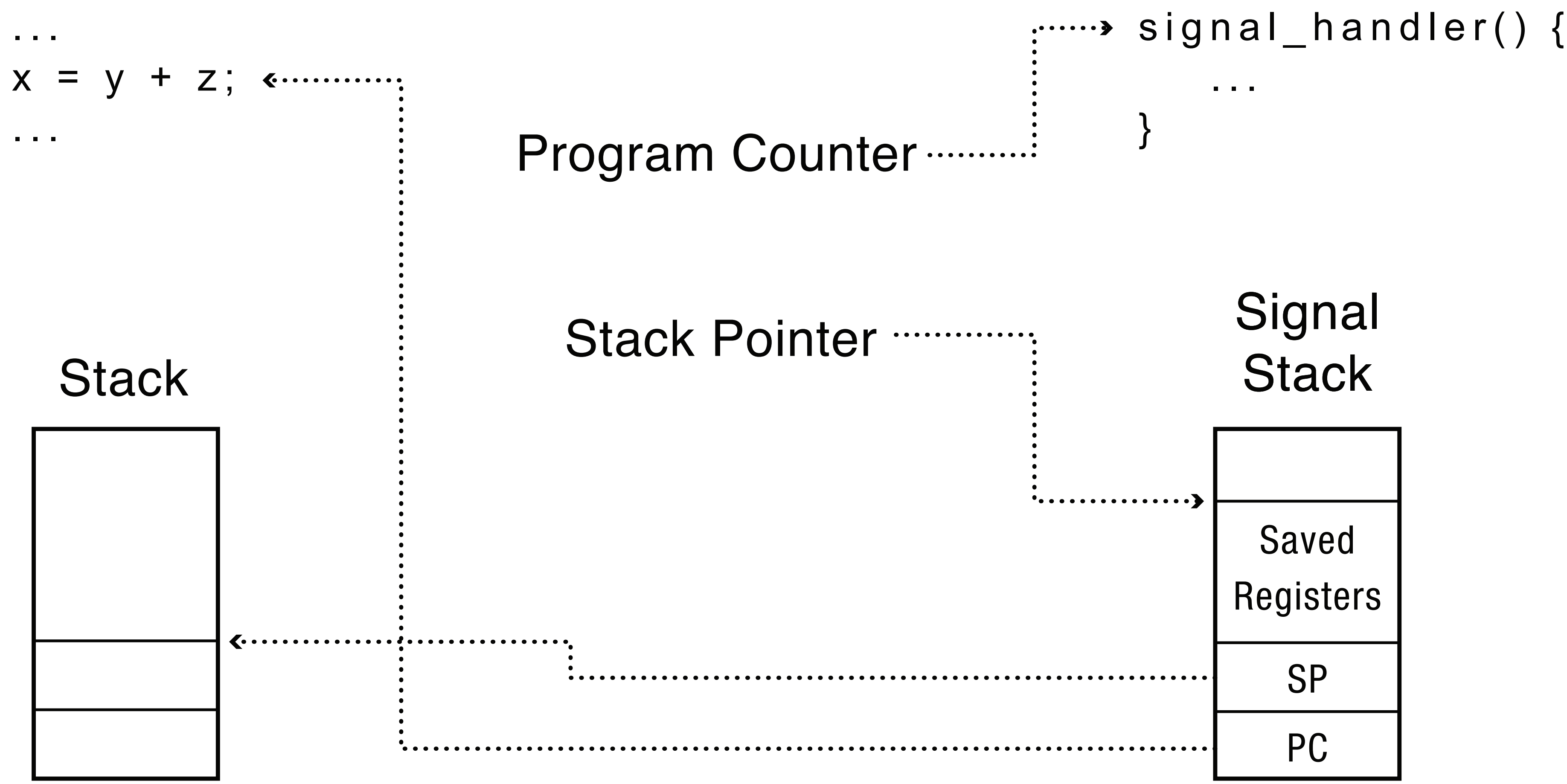
Upcalls vs Interrupts

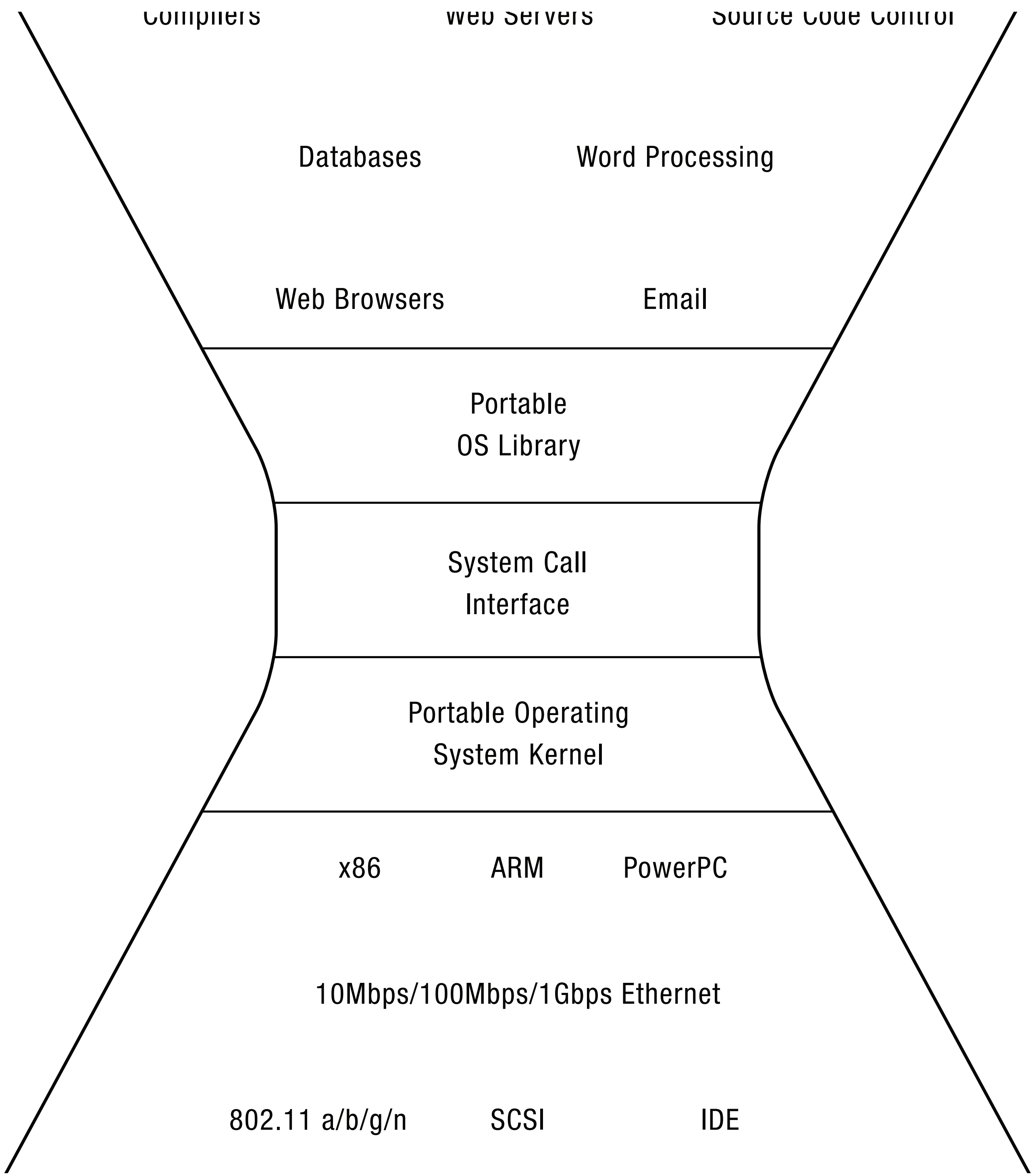
- Signal handlers = interrupt vector
- Signal stack = interrupt stack
- Automatic save/restore registers = transparent resume
- Signal masking: signals disabled while in signal handler

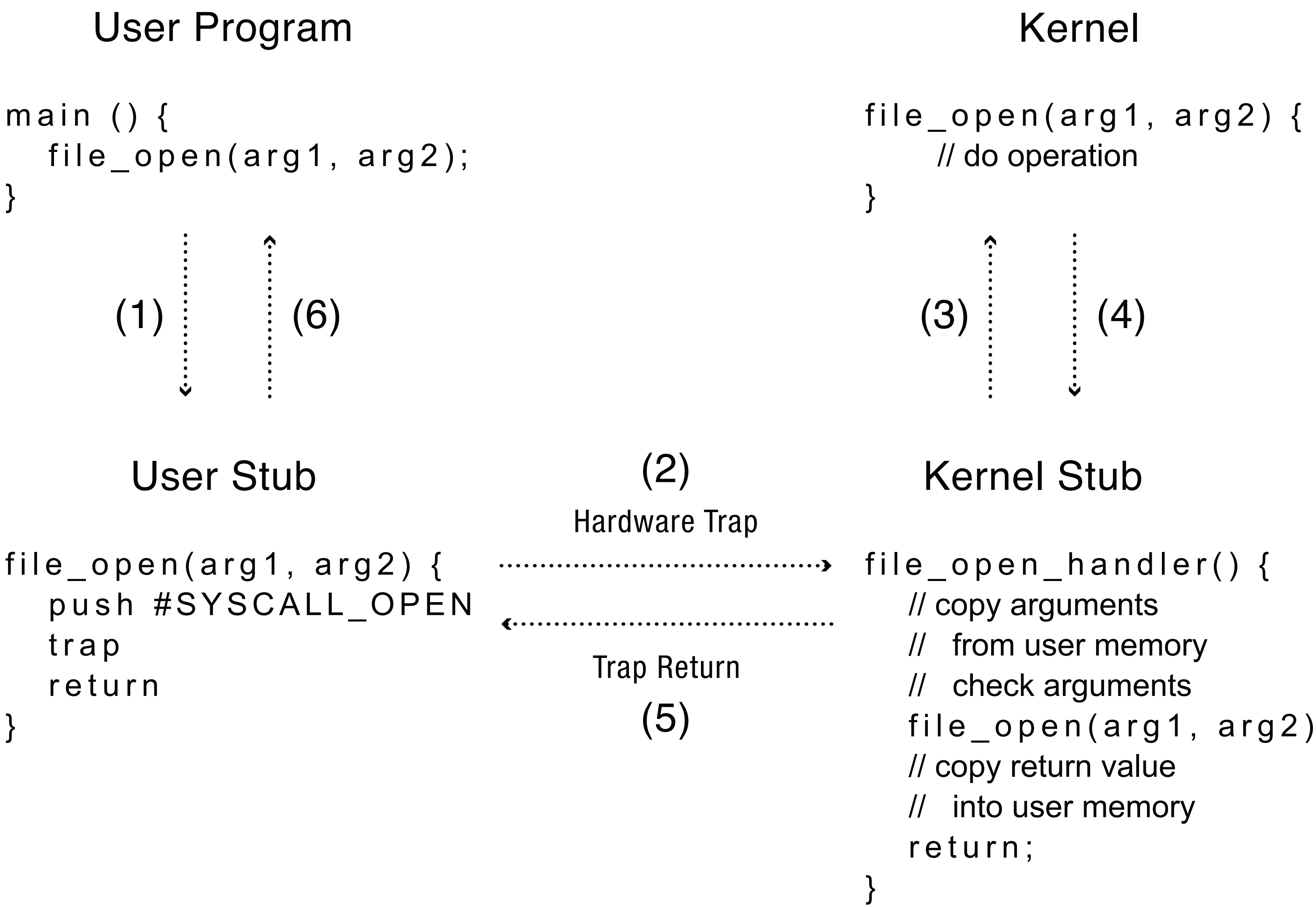
Upcall: Before



Upcall: During




Databases
Word Processing
Web Browsers
Email
Portable
OS Library
System Call
Interface
Portable Operating
System Kernel
x86
ARM
PowerPC
10Mbps/100Mbps/1Gbps Ethernet
802.11 a/b/g/n
SCSI
IDE

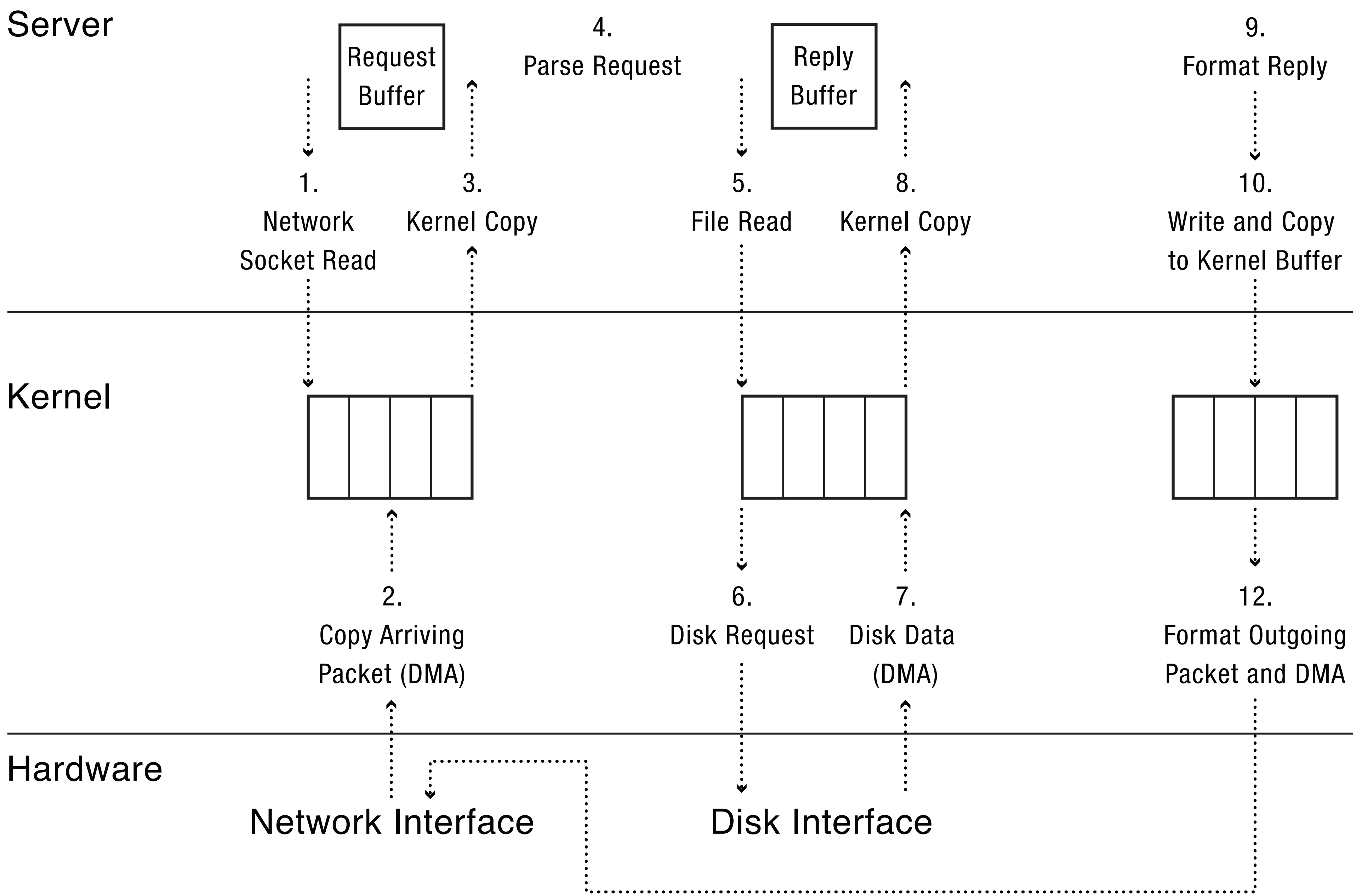

User Program
Kernel
main () {
file_open(arg1, arg2);
}
file_open(arg1, arg2) {
// do operation
}
(1)
(6)
(3)
(4)
User Stub
(2)
Hardware Trap
Kernel Stub
file_open(arg1, arg2) {
push #SYSCALL_OPEN
trap
return
}
file_open_handler() {
// copy arguments
// from user memory
// check arguments
file_open(arg1, arg2);
// copy return value
// into user memory
return;
}
Trap Return
(5)

Kernel System Call Handler

- Locate arguments
  - In registers or on user stack
  - *Translate* user addresses into kernel addresses
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back into user memory
  - *Translate* kernel addresses into user addresses

Question

- How many user-kernel transitions are needed for a static web server to read an incoming HTTP request and reply with the file data?



Server
Request Buffer
4. Parse Request
Reply Buffer
9. Format Reply
1. Network Socket Read
3. Kernel Copy
5. File Read
8. Kernel Copy
10. Write and Copy to Kernel Buffer
Kernel
2. Copy Arriving Packet (DMA)
6. Disk Request
7. Disk Data (DMA)
12. Format Outgoing Packet and DMA
Hardware
Network Interface
Disk Interface

Debugging as Engineering

- Much of your time in this course will be spent debugging
  - In industry, 50% of software dev is debugging
  - Even more for kernel development
- How do you reduce time spent debugging?
  - Produce working code with smallest effort
- Optimize a process involving you, code, computer

Debugging as Science

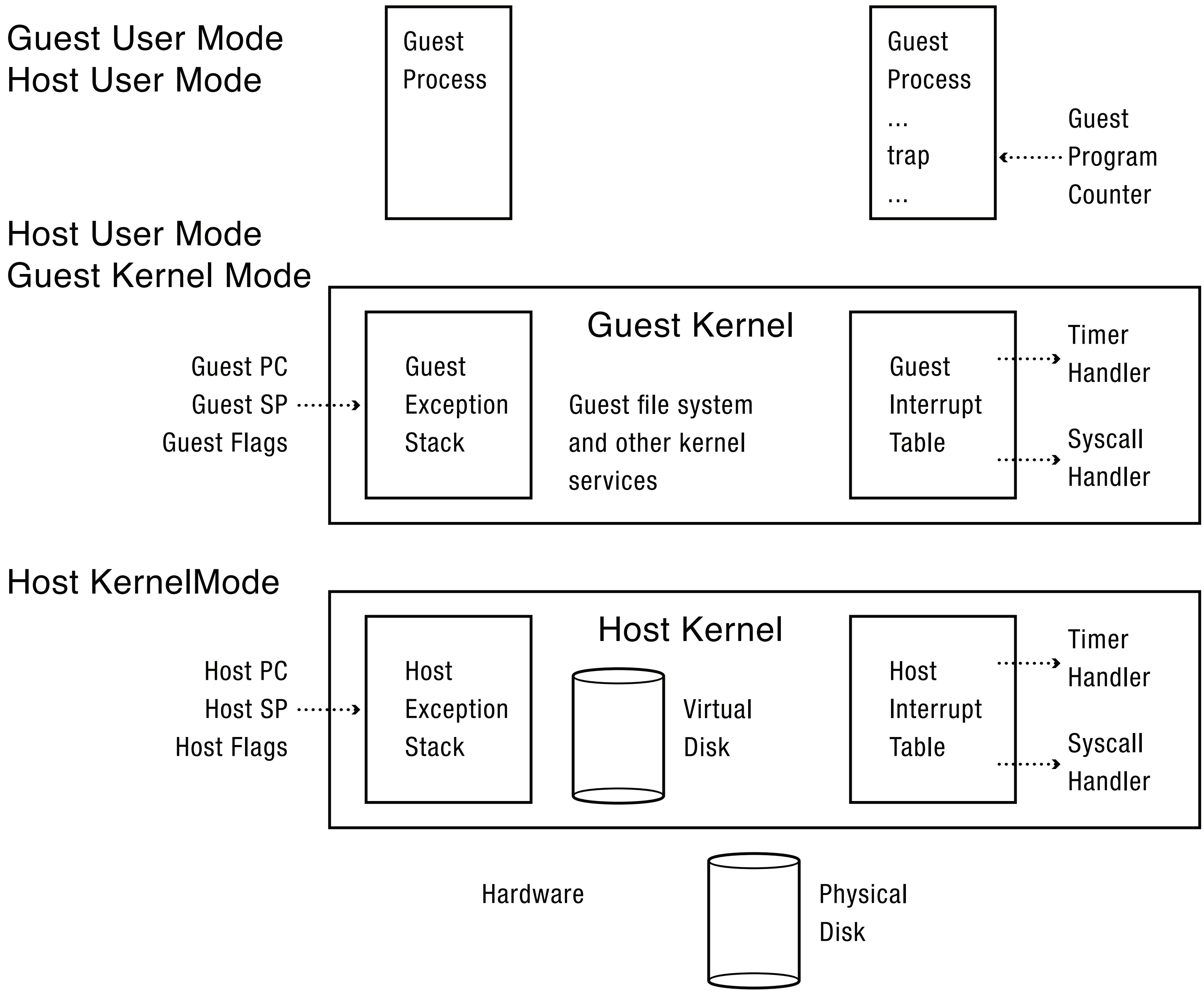
- Understanding -> design -> code
  - not the opposite
- Form a hypothesis that explains the bug
  - Which tests work, which don't. Why?
  - Add tests to narrow possible outcomes
- Use best practices
  - Always walk through your code line by line
  - Module tests – narrow scope of where problem is
  - Develop code in stages, with dummy replacements for later functionality

Virtual Machines

- How do we debug an operating system kernel?
  - Is the debugger an application? Part of the kernel?
- Can we run legacy applications on a new operating system kernel?
- Solution: virtual machine
  - Run a “guest” operating system as a process
  - Run “guest” applications on the guest OS kernel
- Examples: KVM, Vmware, Xen, Citrix, QEMU, System 161

Question

- Can we run a guest operating system directly on the CPU in user mode?
  - Emulate the behavior of privileged instructions executed by guest OS, as if executed on the hardware
- If hardware is "virtualizable"
  - Privileged instructions must cause trap when at user level, rather than fail silently
  - Or kernel must somehow re-write those instructions to cause a trap (vmware)
- Underlying kernel called the "virtual machine monitor" or "host kernel"


Guest User Mode
Host User Mode
Guest
Process
Guest
Process
...
trap
...
Guest
Program
Counter
Host User Mode
Guest Kernel Mode
Guest Kernel
Guest PC
Guest SP
Guest Flags
Guest
Exception
Stack
Guest file system
and other kernel
services
Guest
Interrupt
Table
Timer
Handler
Syscall
Handler
Host KernelMode
Host Kernel
Host PC
Host SP
Host Flags
Host
Exception
Stack
Virtual
Disk
Host
Interrupt
Table
Timer
Handler
Syscall
Handler
Hardware
Physical
Disk

User-Level Virtual Machine

- How does VM Player work?
  - Runs as a user-level application
  - How does it catch privileged instructions, interrupts, device I/O?
- Installs kernel driver, transparent to host kernel
  - Requires administrator privileges!
  - Modifies interrupt table to redirect to kernel VM code
  - If interrupt is for VM, upcall
  - If interrupt is for another process, reinstalls interrupt table and resumes kernel

Case Study: MIPS Interrupt/Trap (Hardware Support)

- Two entry points: TLB miss handler, everything else
- Hardware saves trap type: syscall, exception, interrupt
  - And which type of interrupt/exception/syscall
- Saves program counter: where to resume
- Saves old mode (kernel/user), interruptable bits
- Sets kernel-mode, interrupts disabled
- For TLB (memory) faults
  - Saves virtual address and virtual page
- Jumps to general exception handler
- Handler saves stack pointer, registers (using k0, k1)

System 161

- Machine simulator that runs the OS kernel in a user-level process
  - Simulates the execution of each instruction in turn
- User-level applications run inside the simulator, as if running on real hardware running the OS
- No special support needed from the underlying OS kernel
- Flexible but slow