The Kernel (and Process) Abstraction

Chapter 2-3 OSPP

Part I

Announcements

- HW #1 will be out on Thursday
- Today: kernel
  - Asynchrony
  - Processes
  - Protection

Kernel

- The software component that controls the hardware directly, and implements the core privileged OS functions.

- Modern hardware has features that allow the OS kernel to protect itself from untrusted user code.

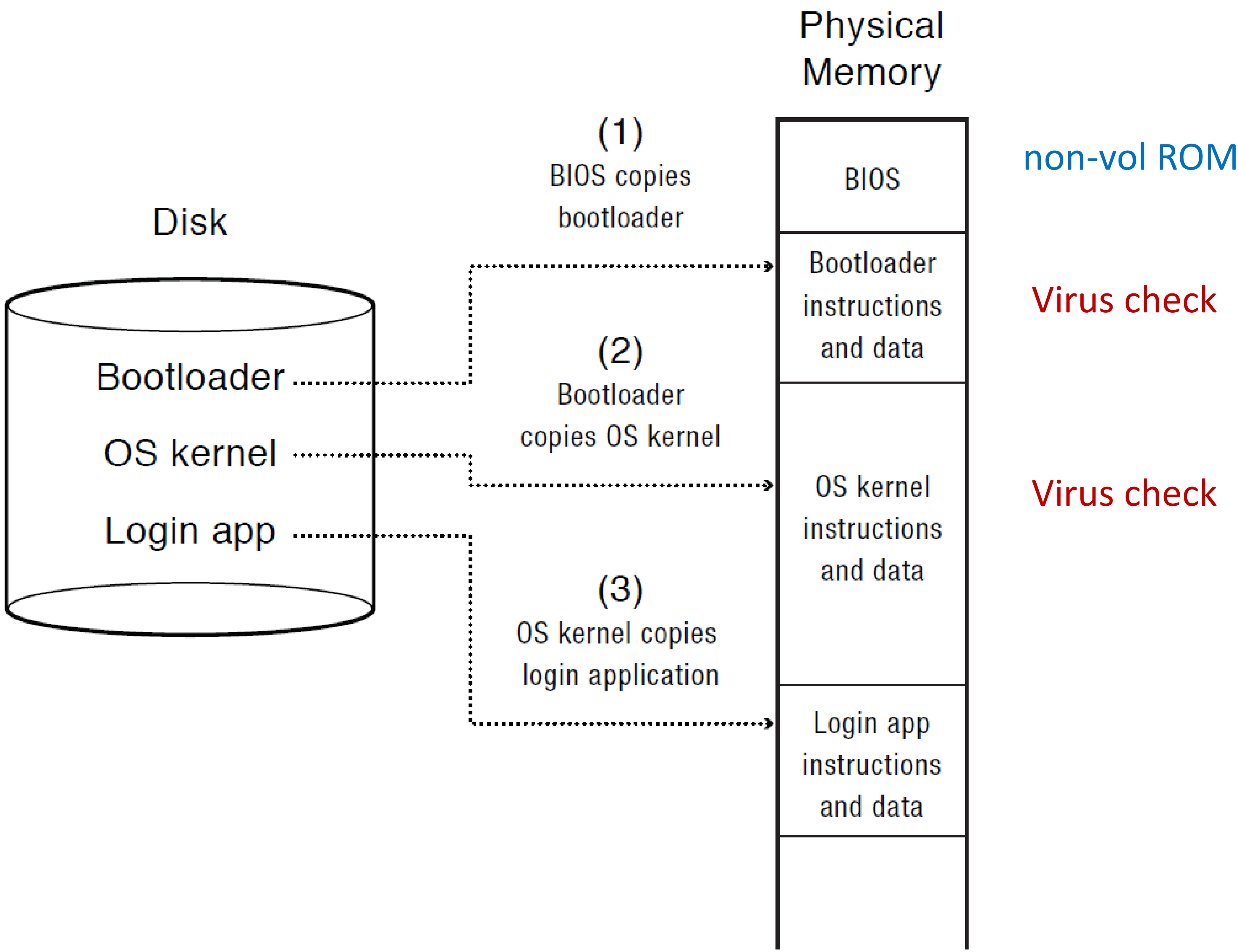
Kernel Protection

- Reliability
  - crashes
- Security
  - Write to arbitrary disk (or memory) locations
- Privacy
  - User files

Does kernel/OS teach any lessons I can use?

- Yes!
- Protection
  - Trend is for apps to be mini-OS?
  - Browser
- Resource management
  - Trend is to give apps X resources and let them figure out how to share
  - User threads, virtual machines
- Asynchrony and many others
  - How?

A Short Digression: Starting the Kernel: Booting



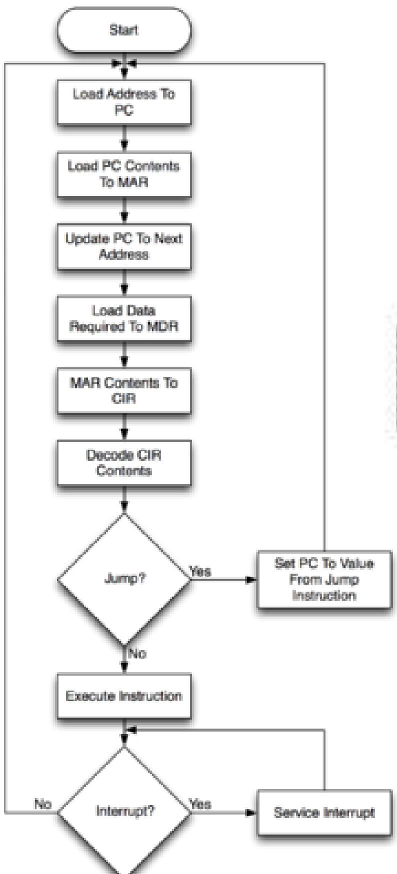
Challenge: Asynchrony: Device Interrupts

- OS kernel needs to communicate with physical devices
- Devices operate asynchronously from the CPU
  - Polling: Kernel waits until I/O is done
  - Interrupts: Kernel can do other work in the meantime
- Device access to memory
  - Programmed I/O: CPU reads and writes to device
  - Direct memory access (DMA) by device

Device Interrupts

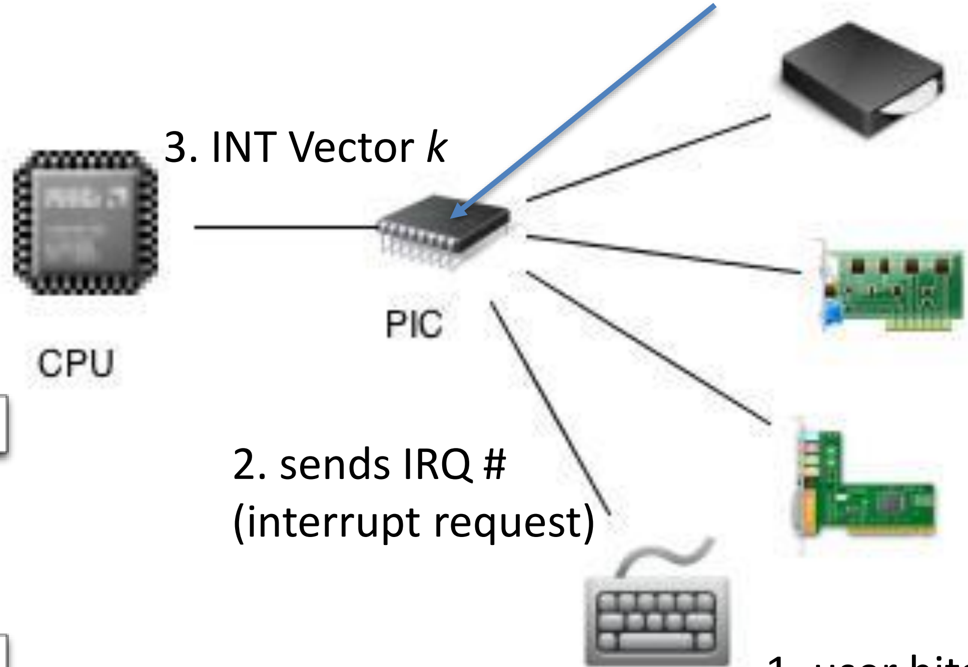
- How do device interrupts work?
  - Where does the CPU run after an interrupt?
  - What is the interrupt handler written in? C? Java?
  - What stack does it use?
  - Is the work the CPU had been doing before the interrupt lost forever?
  - If not, how does the CPU know how to resume that work?
  - Will come back to this soon

How it all happens



programmable interrupt controller (x86)

3. INT Vector *k*



2. sends IRQ #
(interrupt request)

1. user hits a key

CPU checks for interrupts after each instruction cycle

oops, looks like we are "polling" after all but in h/w ☺

Device Driver and I/O Interrupts

- Top half of driver called from syscall handler
  - issues privileged instructions: read from disk, done

- Bottom half
  - called when interrupt arrives
  - interrupt handling: I/O completion or error recovery

Interrupt Handler

- Bottom half
  - runs first called directly by hardware, saves state of hardware, then enables top half to run
- Top half ~ interrupt handler specifics
  - for an I/O event: calls driver bottom half: e.g. data copying

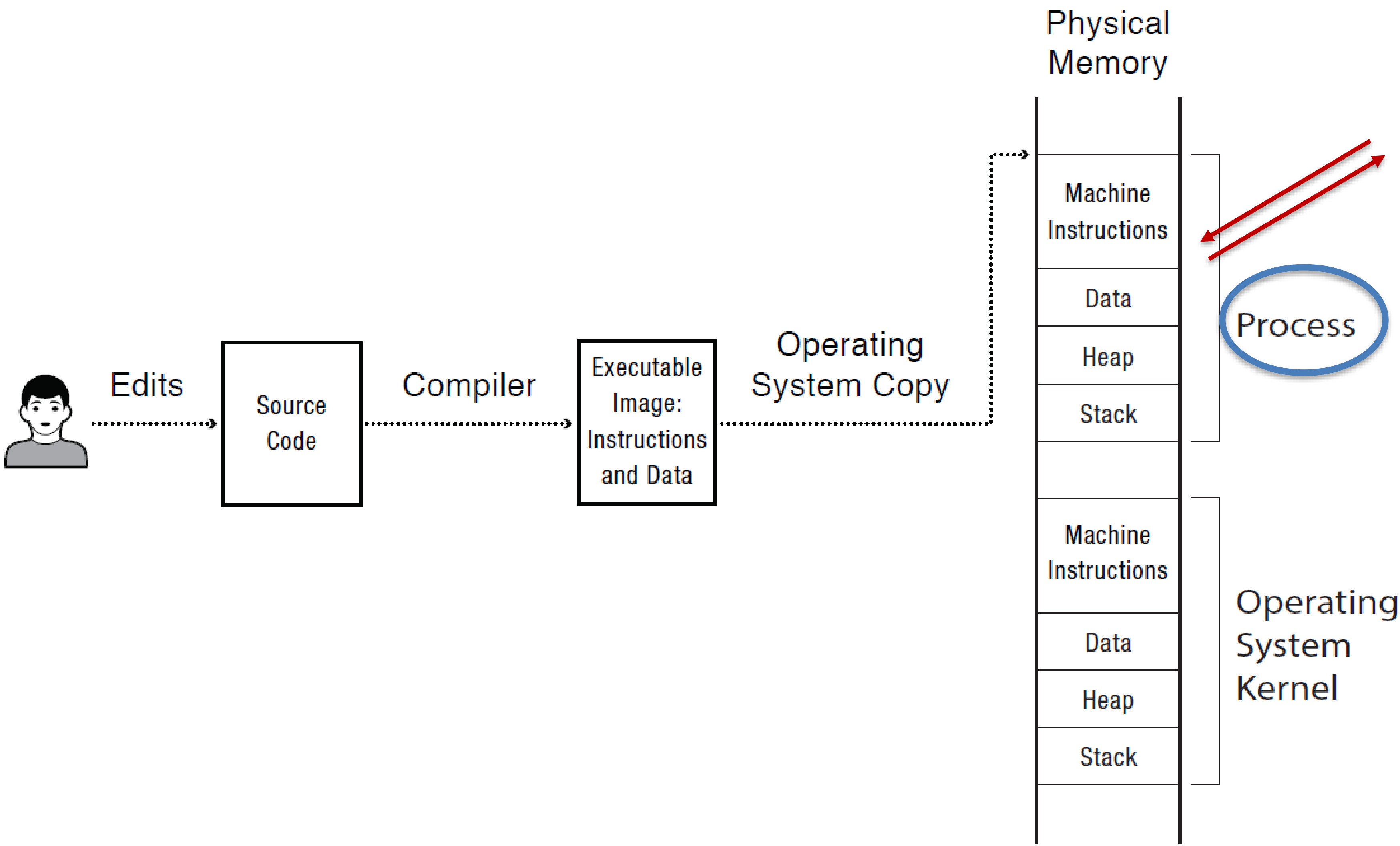
Buggy Device Drivers

- Validate/inspect
- User-level drivers
- Running drivers in VM
- Sandboxing
  - mini-execution environment in the kernel

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
  - First see how OS does it

A Problem: both constrain and protect process



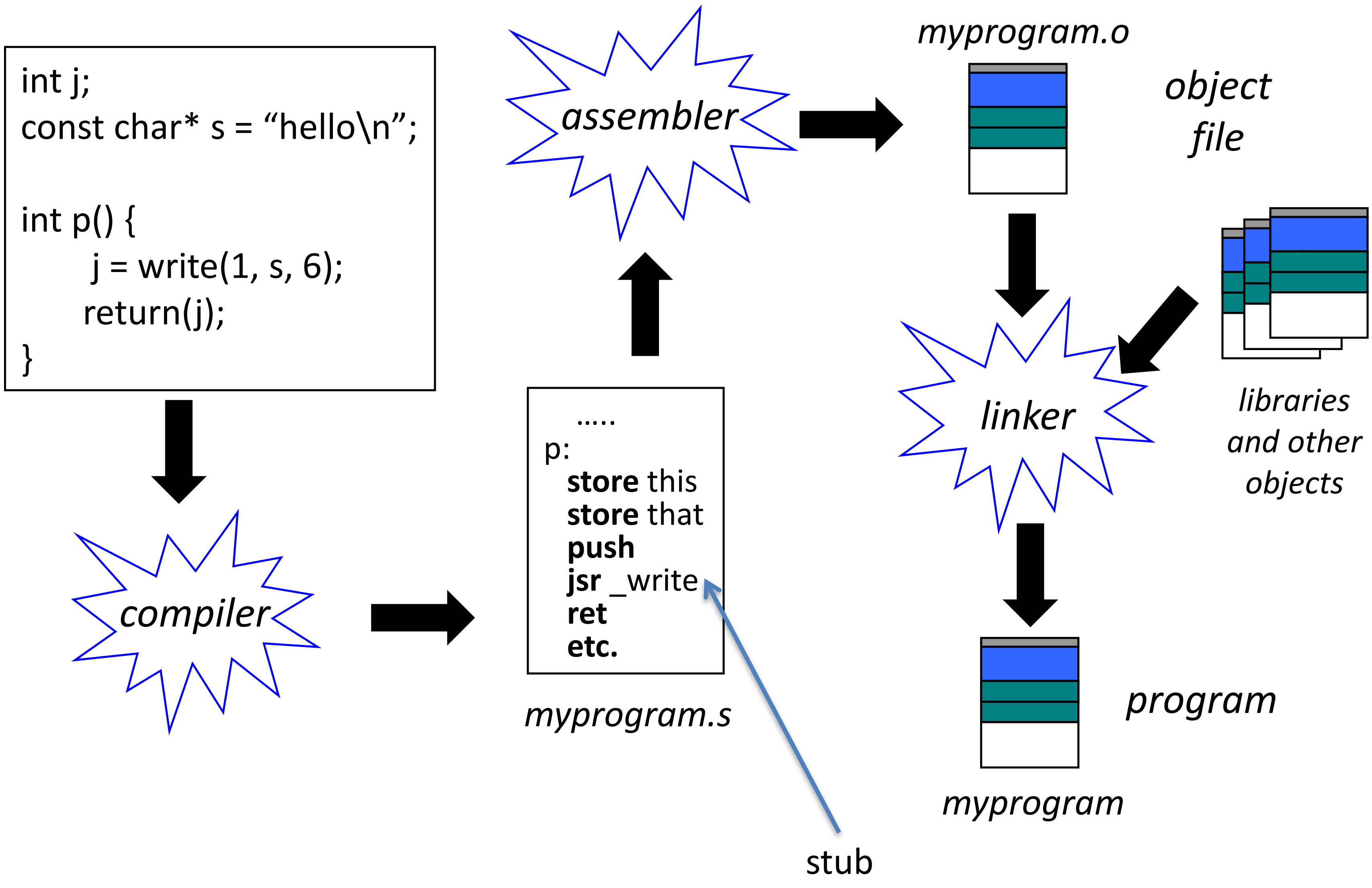
Main Points

- Process concept
  - A process is the OS abstraction for executing a program with limited privileges but that is isolated
- Dual-mode operation: user vs. kernel
  - Kernel-mode: execute with complete privileges
  - User-mode: execute with fewer privileges
  - Processor is a warden (OS) and an inmate (process)!
- Safe control transfer
  - How do we switch from one mode to the other?

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)

The Birth of a Program



Birth of a Process: Process State

Stored in PCB (Process Control Block)

Information associated with each process

- Program counter, stack pointer
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information
- Open files, signals (if UNIX)

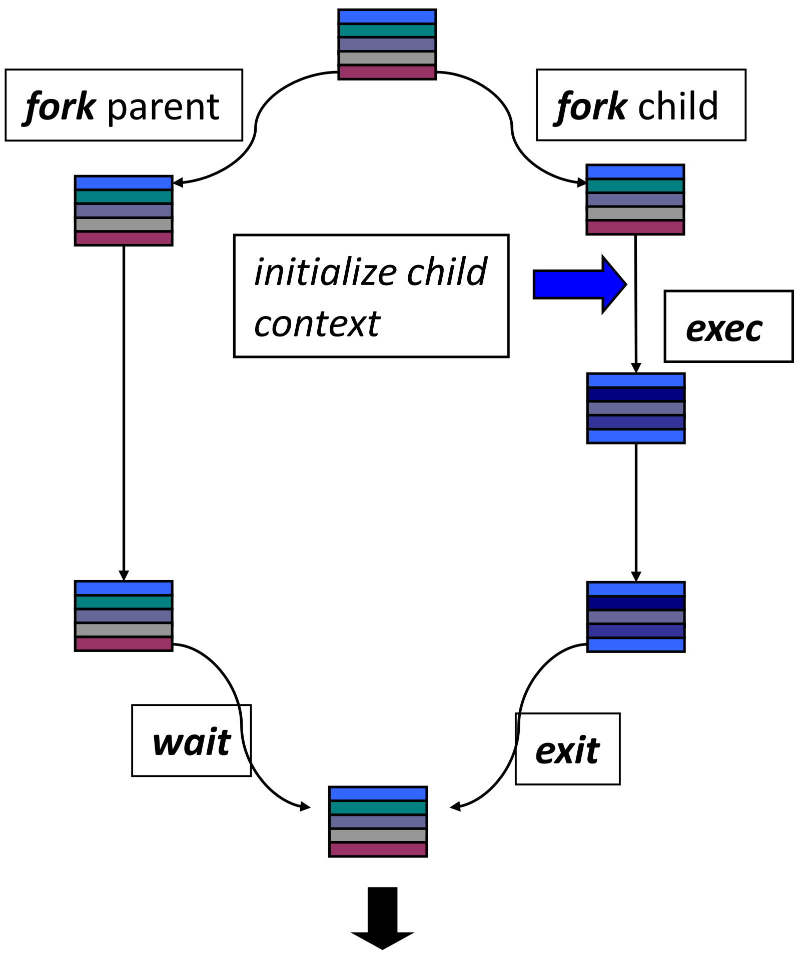
Process API

- Very briefly

UNIX Process Management

- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec – system call to change the program being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send a notification to another process
- UNIX/LINUX clone – similar to fork but used with threads

Unix `Fork/Exec/Exit/Wait` Example



int pid = fork();

Create a new process that is a clone of its parent.

exec*("program" *[, argvp, envp]*);

Overlay the calling process virtual memory with a new program, and transfer control to it.

exit(status);

Exit with status, destroying the process.

int pid = wait*(&status);

Wait for exit (or other status change) of a child.

Corner cases: orphans and zombies

Example: Process Creation in Unix

```
int pid;
int status = 0;

if (pid = fork()) {
      /* parent */
      .....
      pid = wait(&status);
} else {
      /* child */
      .....
      exit(status);
}
```

The **fork** syscall returns <u>twice</u>: it returns a zero to the child and the child process ID (pid) to the parent.

Parent uses **wait** to sleep until the child exits; **wait** returns child pid and status.

Implementing UNIX fork

Steps to implement UNIX fork

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
  - mostly sets up the page table
  - some implementations share portions of address space initially (copy-on-write)
- Inherit parent execution context (e.g., any open files, PC, SP)
- Inform the scheduler that the new process is ready to run

Implementing UNIX exec

- Steps to implement UNIX exec
  - Load the program into the current address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start'‘ (reset PC)

Questions

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately? Why?

Starting a New Process

- Allocate PCB; in Unix this is already done by `fork`
- Allocate memory (as needed, on demand)
  - "Copy" program from disk into memory
  - Allocate user stack
  - Allocate heap
- Allocate kernel stack (sys calls, interrupts, exceptions)

Starting a New Process (cont'd)

- Transfer to user mode
- If Exec path (vs. fork)
  - Copy arguments into user memory (e.g. `argc, argv`)
  - Jump to `start` address

```
start (arg1, arg2) {
  main (arg1, arg2);
  exit ();
}
```

Why not just call main?

Back to Protection

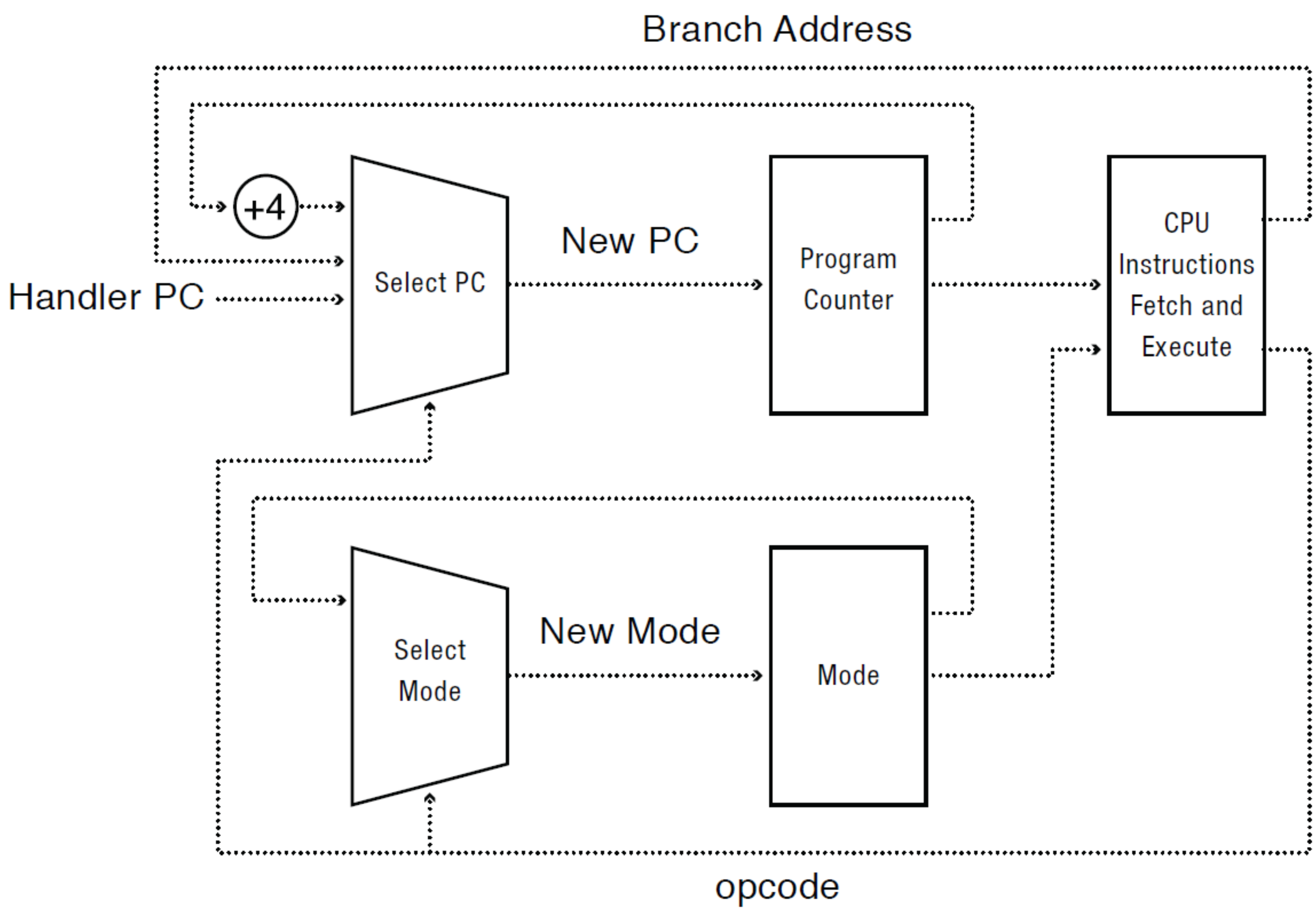
Thought Experiment

- How can we implement execution with limited privilege (no hardware support)?
  - Execute each program instruction in a simulator
  - If the instruction is permitted, do the instruction
  - Basic model in Javascript and other interpreted languages
- How do we go faster?
  - Run the unprivileged code directly on the CPU!
  - Checking in hardware

Hardware Support: 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 CPU with Dual-Mode Operation



Where do interrupts fit in?

The Kernel Abstraction

Chapter 2 OSPP

Part II

Hardware Support: 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?
  - Change mode bit in EFLAGs register!
  - Change which memory locations a user program can access
  - Send commands to I/O devices
  - Read data from/write data to I/O devices
  - Jump into kernel code

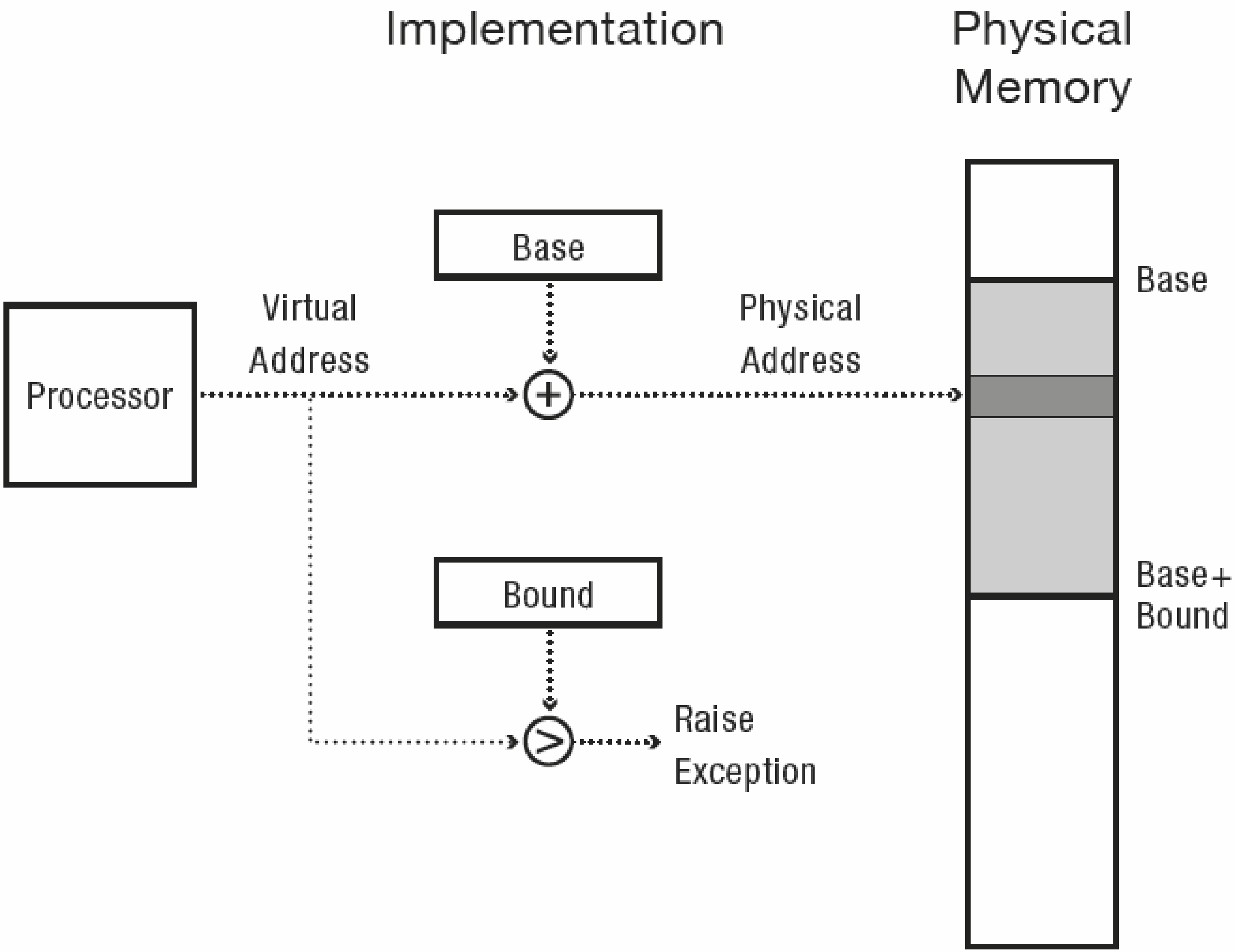
- What should happen if a user program attempts to execute a privileged instruction?

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?
  - one app can over-write the display of another or corrupt the display (interspersed updates)

Simple Memory Protection

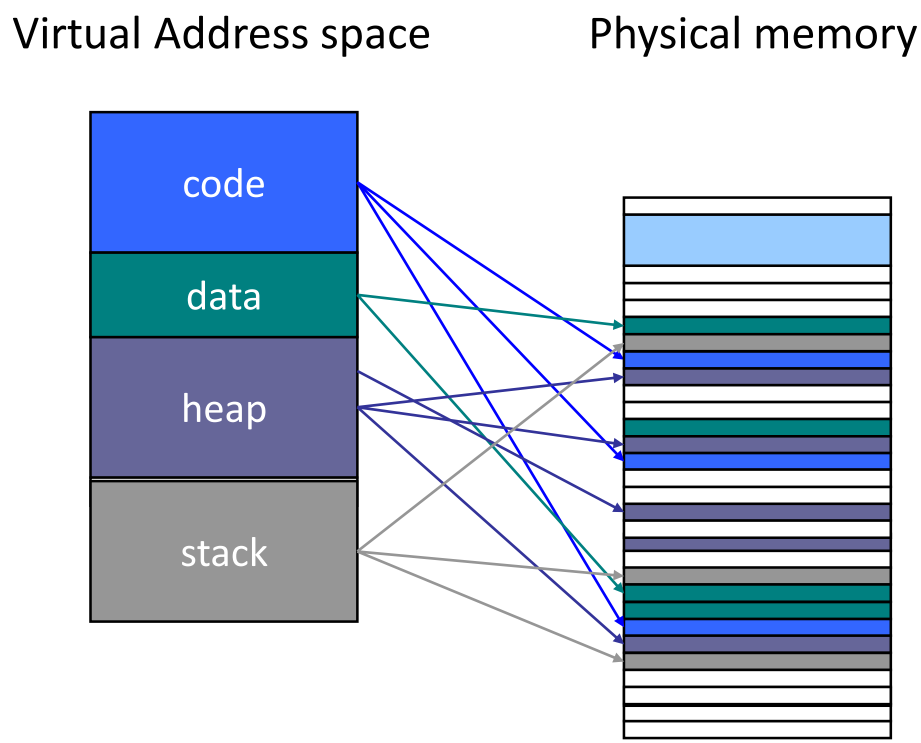


Towards Virtual Addresses

- Problems with base and bound?
  - Expandable heap?
  - Expandable stack?
  - Memory sharing between processes?
  - Memory fragmentation

Virtual Addresses

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



Example

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

What happens if we run two instances of this program at the same time: `staticVar`, `localVar`?

Back to Interrupts: 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
  - Important for protection as well as scheduling

Mode Switch

- From user mode to kernel mode
  - Interrupts
    - Triggered by timer and I/O devices
  - Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (aka protected procedure call)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

Simple Examples

- Examples of exceptions
  - Memory error
  - Divide by 0

- Examples of system calls
  - `read/write`

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

How do we handle interrupts safely?

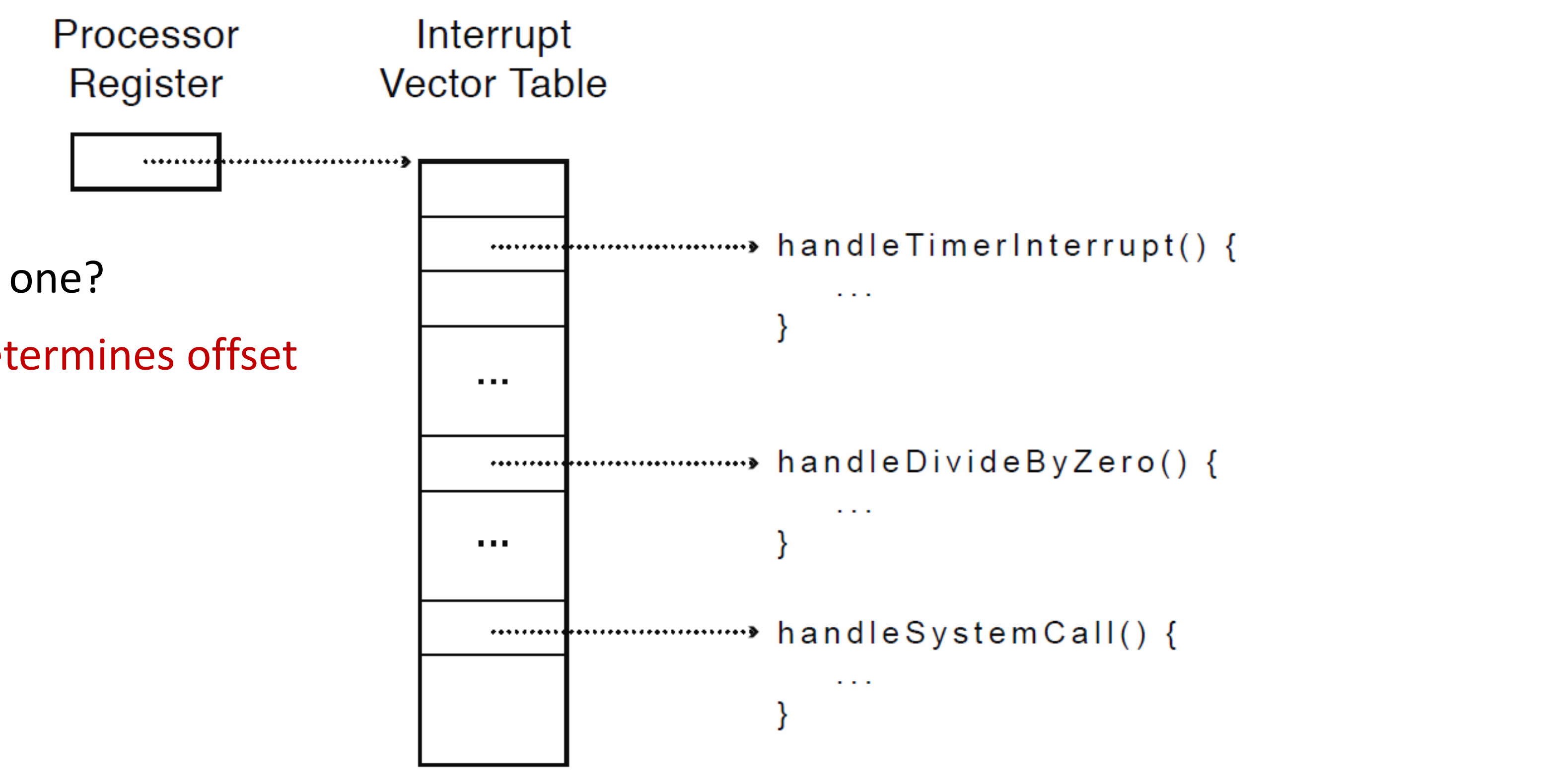
- Interrupt vector
  - Limited number of entry points into kernel
- Atomic transfer of control
  - Single instruction to change (all changed together)
    - Program counter
    - Stack pointer
    - Kernel/user mode
    - Memory protection
- Transparent re-startable execution
  - User program does not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

Interrupt Vector

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

Which one?

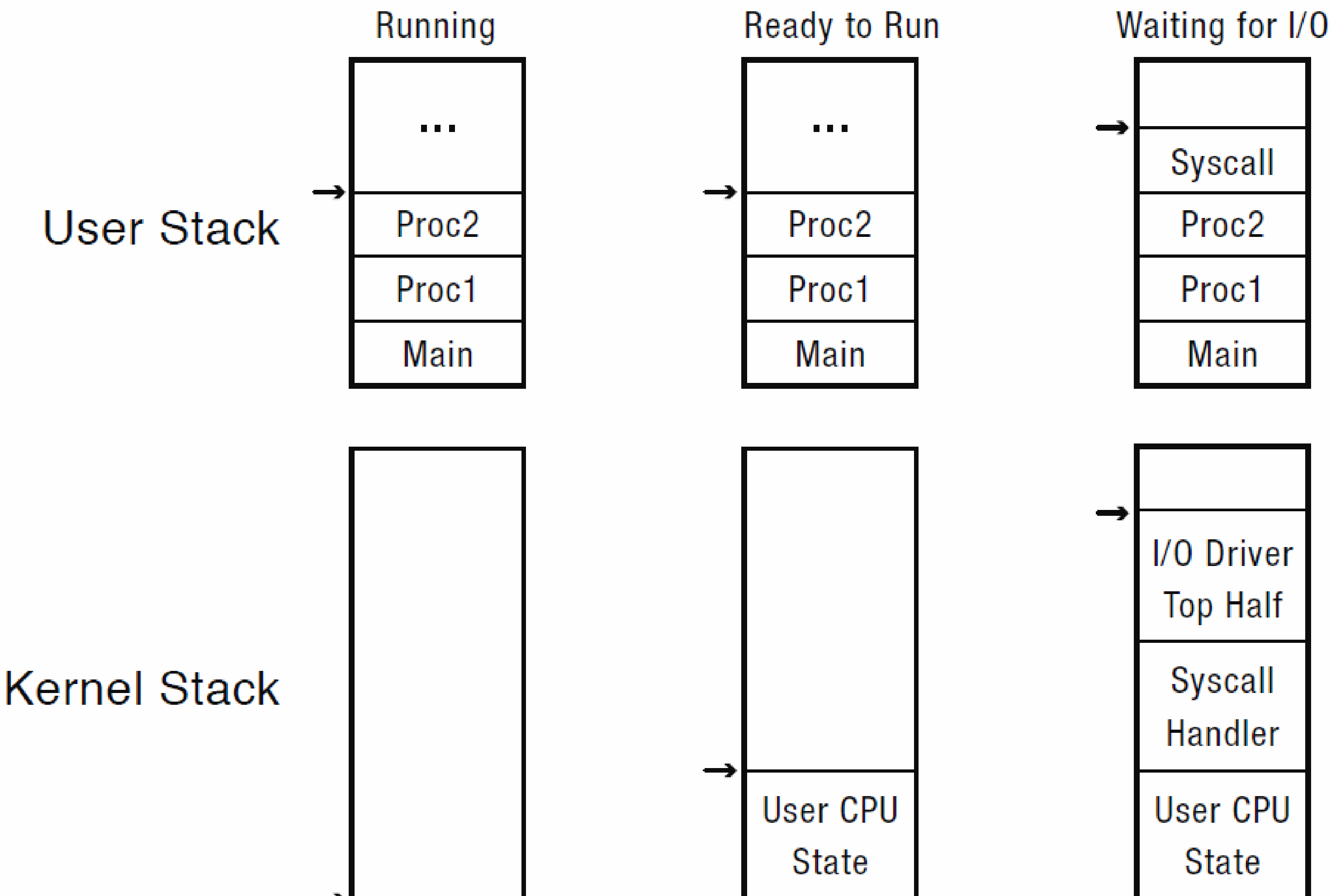
IRQ determines offset



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?
  - user stack may be corrupted or modified

Interrupt Stack



User mode

Preempted
(timer int)

Blocked
(syscall "int")

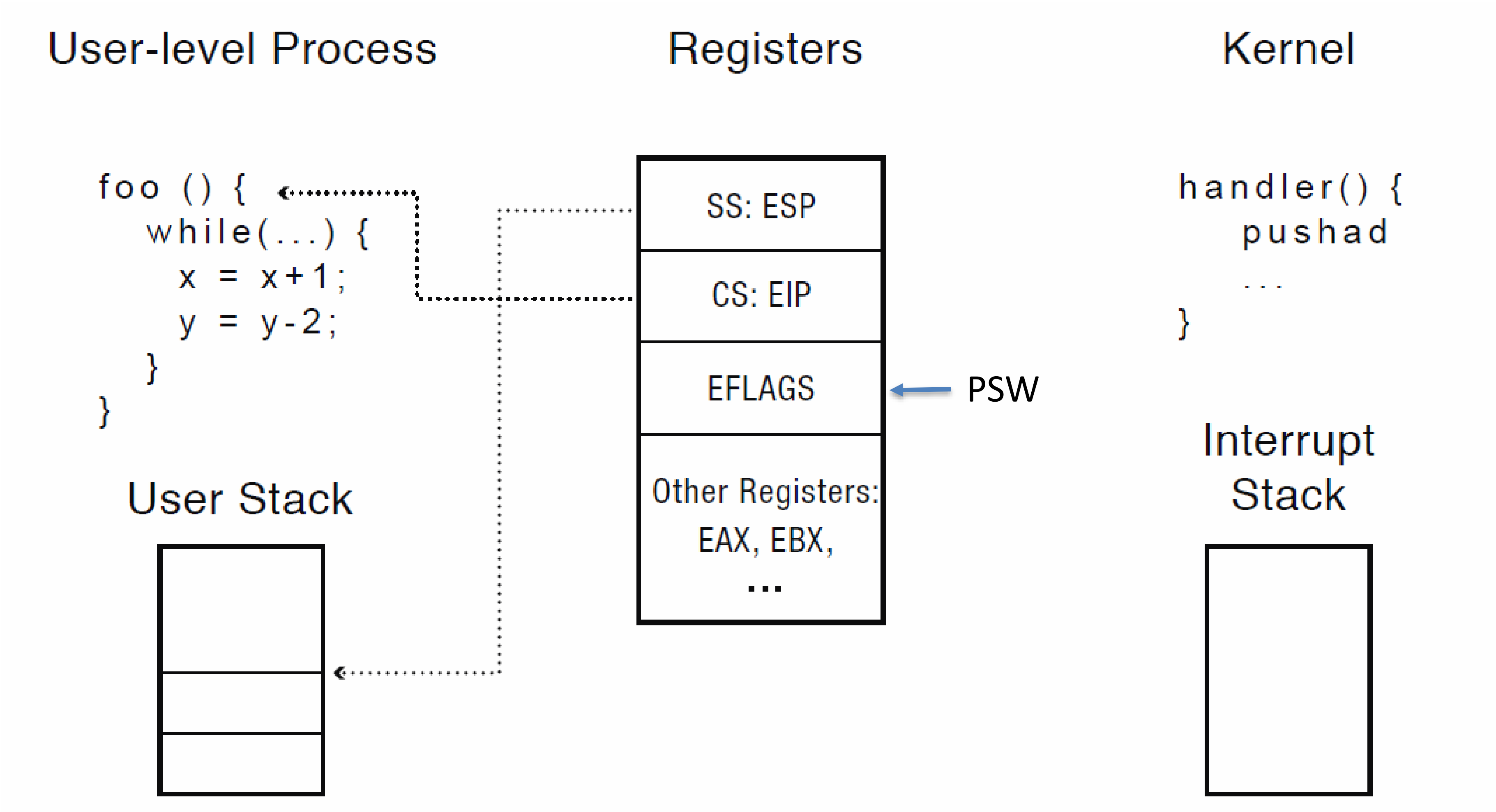
Interrupt Masking

- Interrupt handler runs with interrupts off
  - Why do we need to mask/buffer interrupts in the handler?
  - 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 interrupts
    - STI: enable interrupts
    - Only applies to the current CPU core
- We'll need this to implement synchronization in chapter 5

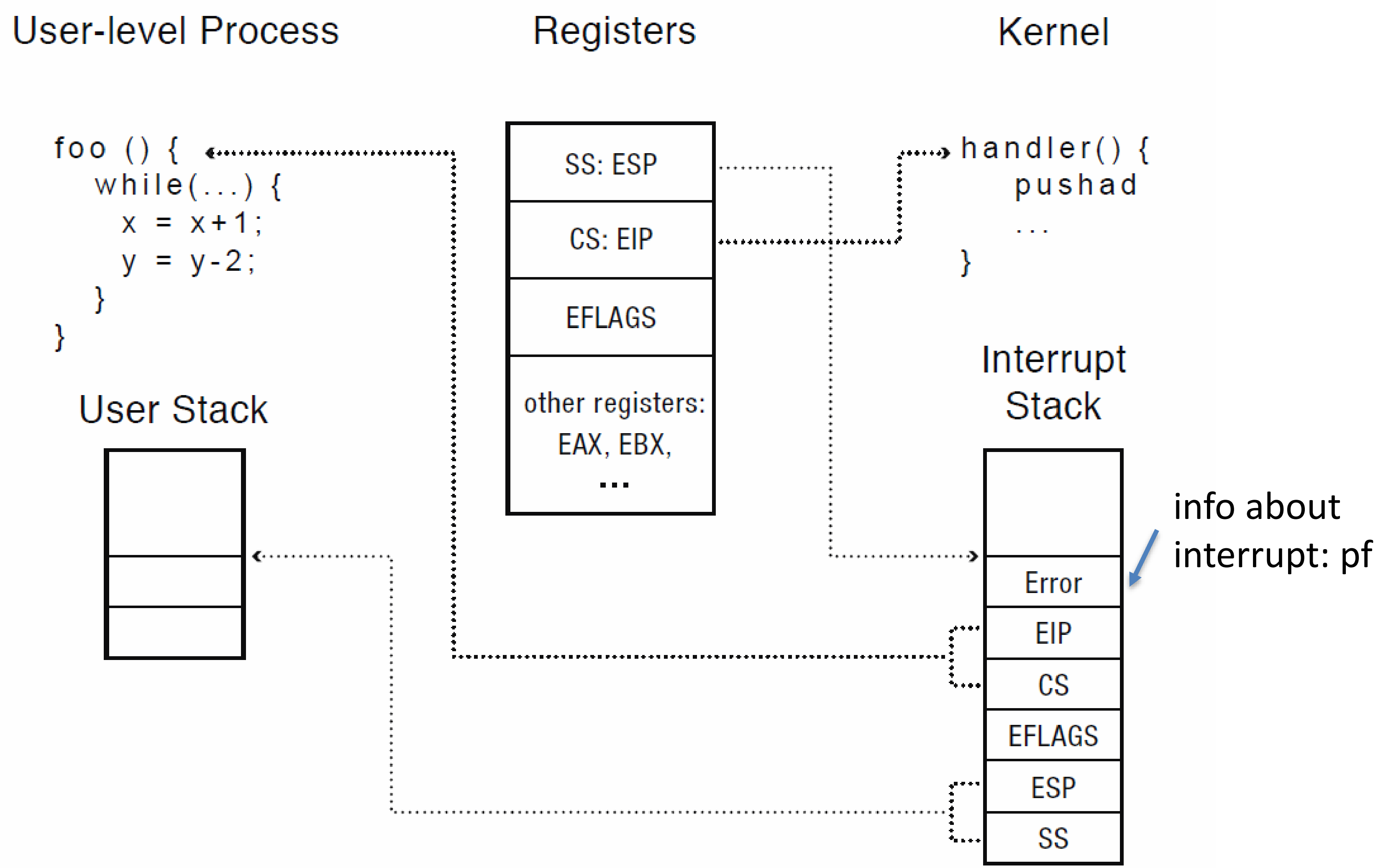
Case Study: x86 Interrupt

- Save current stack pointer
- Save current program counter
- Save current processor status word (condition codes: conditional results, arithmetic carry, …)
- Switch to kernel stack; put SP, PC, PSW on stack
- **Switch to kernel mode**
- Vector through interrupt table
- Interrupt handler saves registers it might use
  - `pushad`: save 'em all

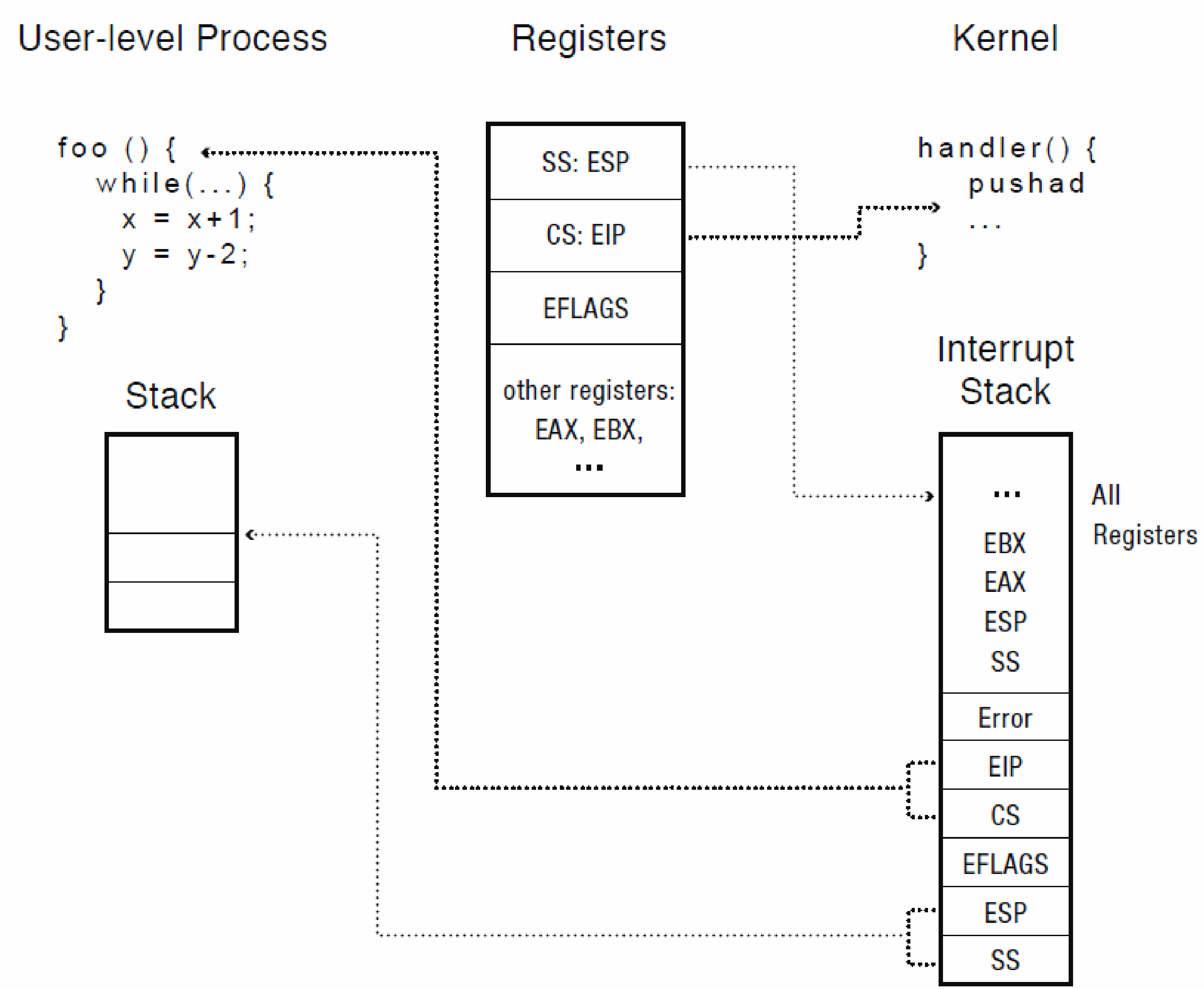
Before Interrupt



Jumped to Interrupt Handler



Executing the handler



At end of handler

- Handler restores saved registers
- Atomically returns to interrupted process/thread (hopefully)
  - Restore program counter
  - Restore program stack
  - Restore processor status word
  - **Switch to user mode**

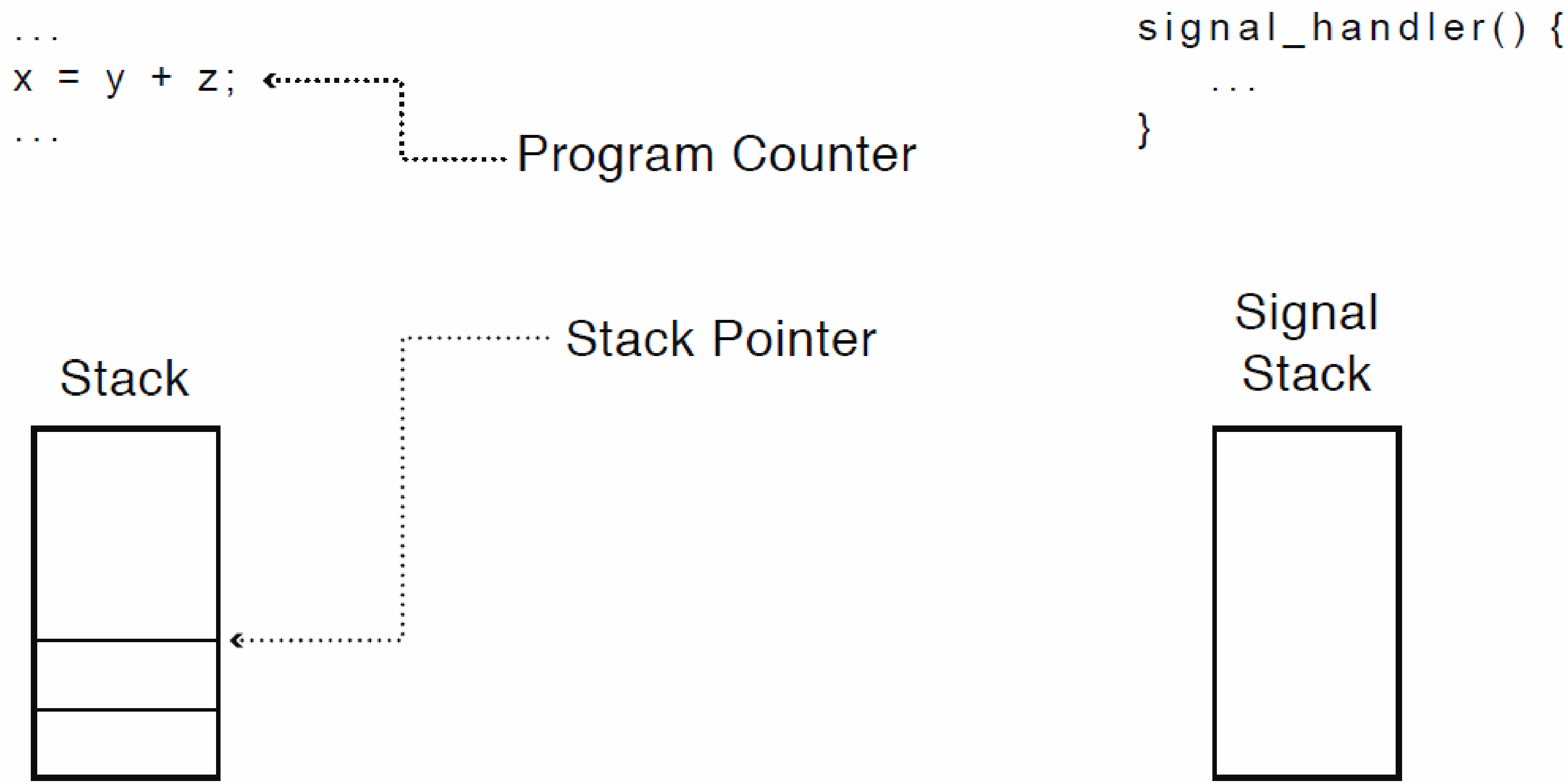
Upcall: User-level event delivery

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

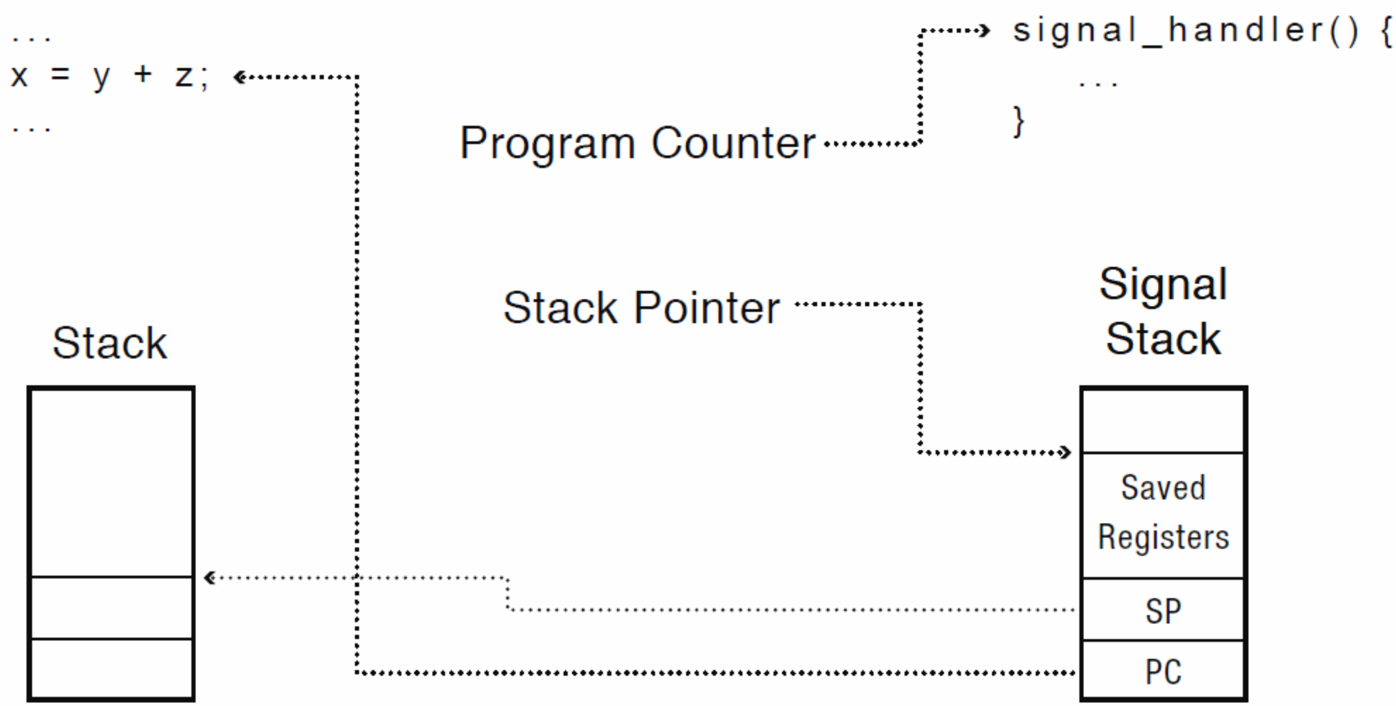
Upcalls vs Interrupts

- Signal handlers ~ interrupt vector
- Signal stack ~ interrupt stack
- Automatic save/restore registers = transparent resume
- Signal masking: signals disabled while in signal handler
- But it runs in user-land

Upcall: Before

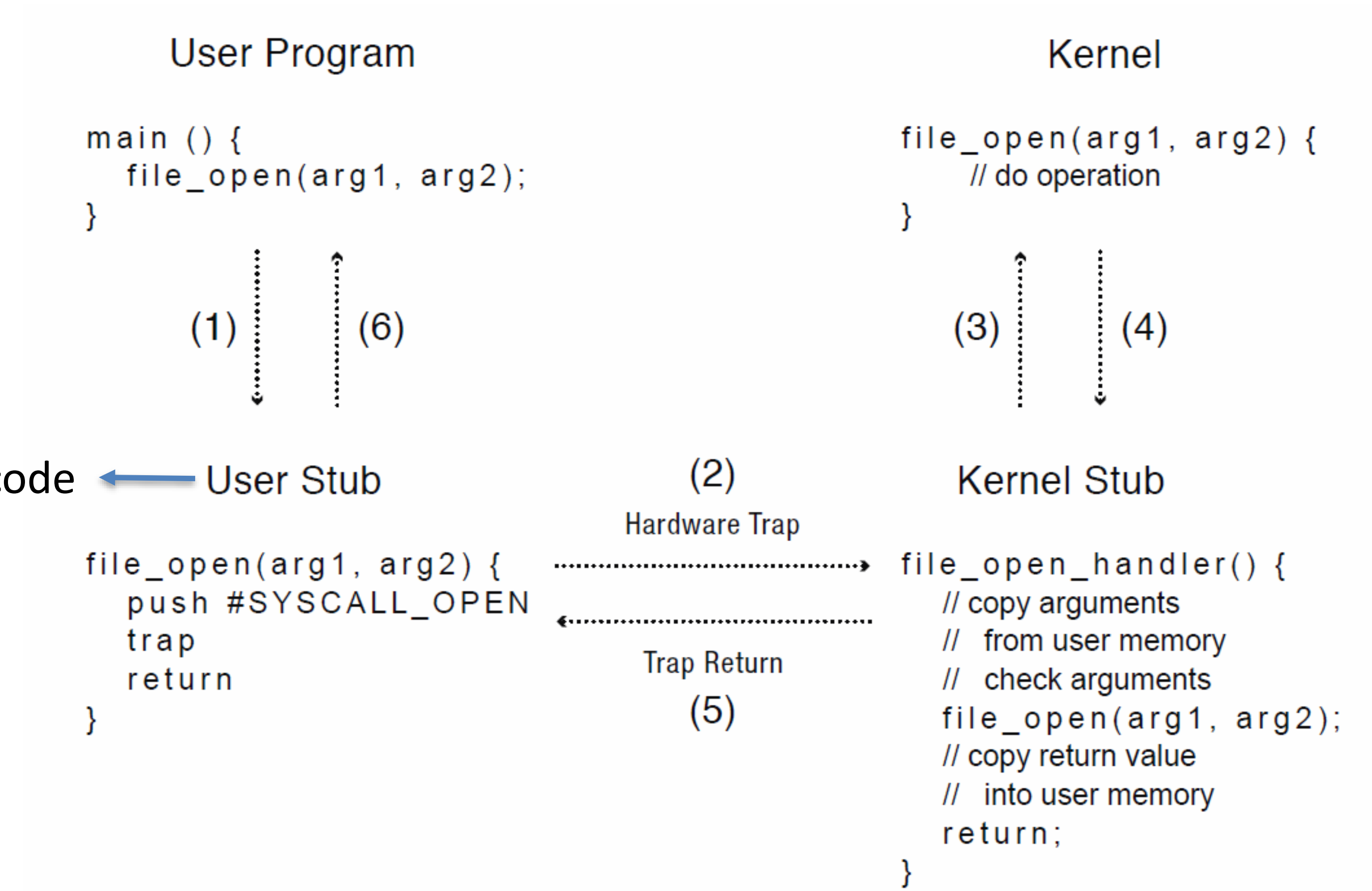


Upcall: During



Making system calls secure

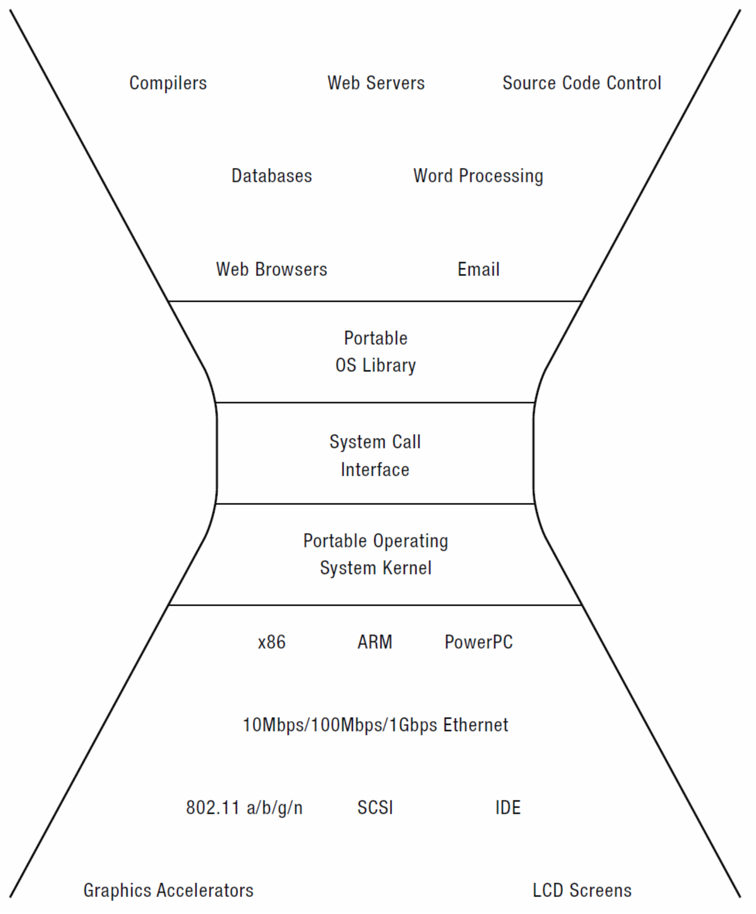
e.g. ~ libc code



Kernel System Call Handler

- Locate arguments
  - In registers or on user stack
  - *Translate* user addresses (VA) into kernel addresses (PA)
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
  - Time-of-check vs. Time-of-use (TOCTOU) attack avoided
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back into user memory
  - *Translate* kernel addresses into user addresses

Genius of OS software stack



One Implication of this

- Get to choose where to put functionality
- User-level process
  - Unix: user-level shell, login
- User-level library
  - Unix: lib.c (I/O, fork/exec, …)
- OS kernel
  - File system, network stack, etc

Next Week

- Threads
- Read Chap. 4 OSPP