Computer Operating Systems

Tom Anderson
Antoine Kaufmann
Winter 2017
http://courses.cs.washington.edu/courses/csep551/17wi

Course Structure

- How operating systems work
  - OSPP, xv6, and classic OS papers
  - At the level of working code, in a specific OS (xv6)
  - Build some key pieces of OS functionality
- Recent trends in operating systems
  - Read/discuss some recent research papers

Problem Sets

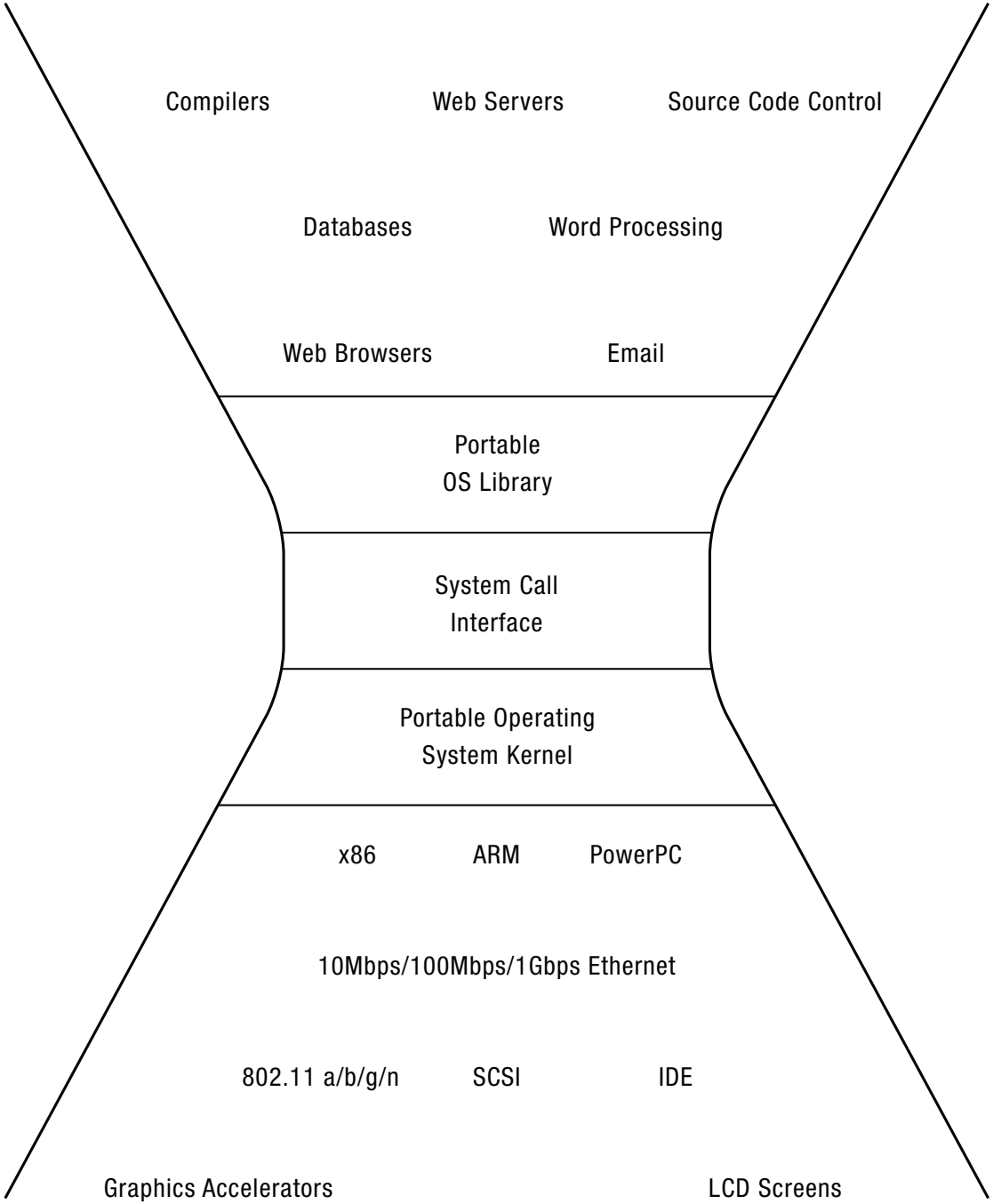
- Highest priority for class time
- Seven in all, one due every 1.5 weeks
  - A few written questions about xv6
  - Then build some core aspect of OS functionality
- Aim: basic functionality
  - Don't worry about writing comprehensive tests
- Mechanics
  - OK to drop one assignment for free
  - OK to submit two assignments late
  - OK to drop more with penalty

Blogs

- One blog post per week, on **one** of the research papers for that week
- Equal in aggregate to one problem set
- Short, unique comment, observation or question
- By Thursday at 4pm

What is an OS?

- Lowest level of software running on a machine
  - Provides convenient programming abstraction (but what does it use itself?)
  - Processor and device concurrency
  - Isolation among multiple users
  - Physical resource limits (e.g., malloc can fail)
  - Physical device config and management


Compilers
Web Servers
Source Code Control
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
Graphics Accelerators
LCD Screens

Device I/O

- OS kernel needs to communicate with physical devices
  - Network, disk, video, USB, keyboard, mouse, …
- Devices operate asynchronously from the CPU
  - Most have their own microprocessor
  - Example: Apple Watch OS runs laptop keyboard

Device I/O

- How does the OS communicate with the device?
  - I/O devices assigned a range of memory addresses
  - Separate from main DRAM memory
  - To issue commands/read results:
    - Special instructions (e.g., inb/outb)
    - Read/write memory locations

Synchronous I/O

- Polling
  - I/O operations take time (physical limits)
  - OS pokes I/O memory on device to issue request
  - While device is working, kernel polls I/O memory to wait until I/O is done
  - Device completes, stores data in its buffers
  - Kernel copies data from device into memory

Faster I/O: Interrupts

- Interrupts
  - OS pokes I/O memory on device to issue request
  - CPU goes back to work on some other task
  - Device completes, stores data in its buffers
  - Triggers CPU interrupt to signal I/O completion
  - Device specific handler code runs
  - When done, resume previous work

Faster I/O: DMA

- Programmed I/O
  - I/O results stored in the device
  - CPU reads and writes to device memory
  - Each CPU instruction is an uncached read/write (over the I/O bus)
- Direct memory access (DMA)
  - I/O device reads/writes the computer's memory
  - After I/O interrupt, CPU can access results in memory

Faster I/O: Buffer Descriptors

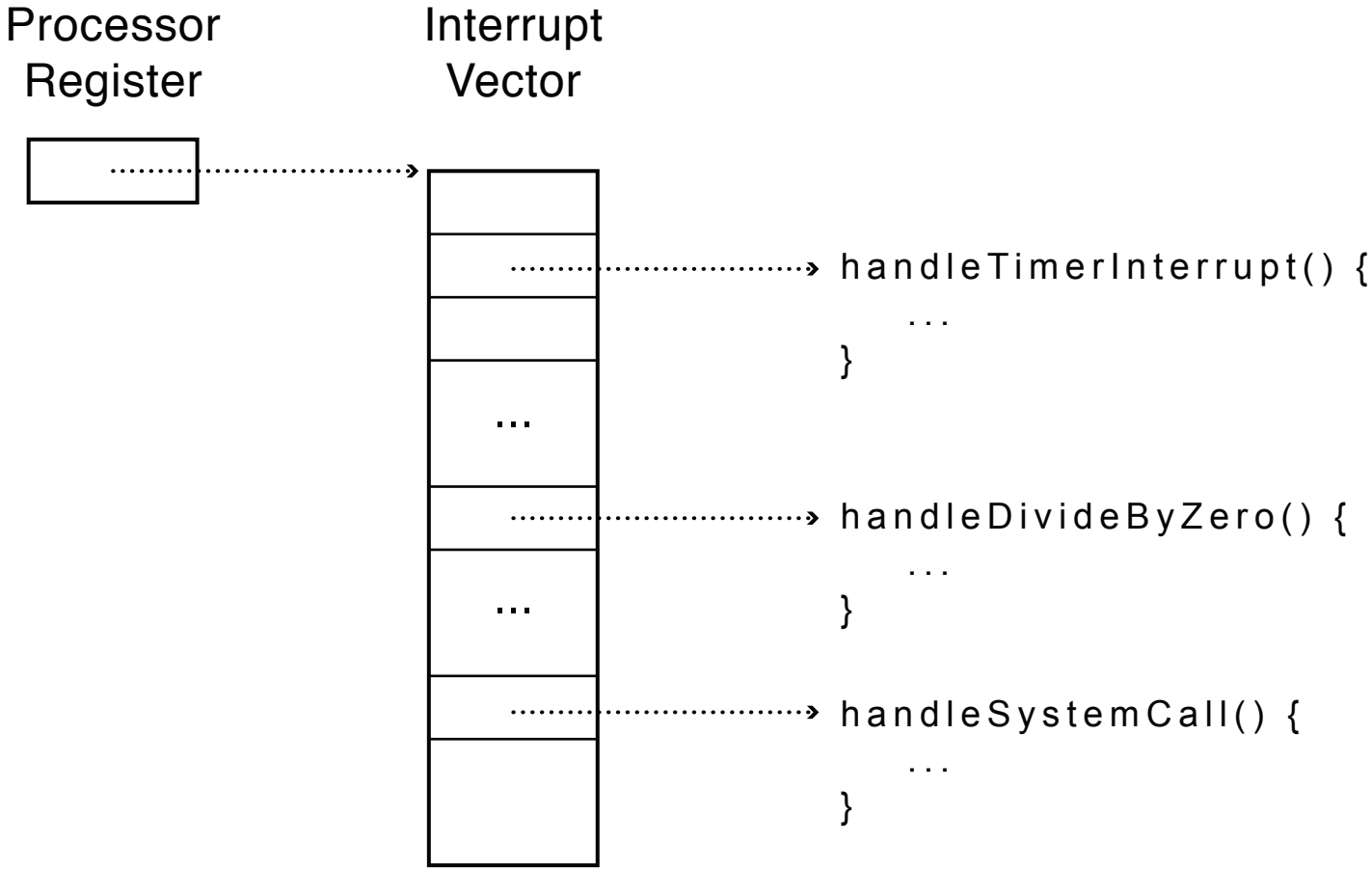
- Buffer descriptor: data structure to specify where to find the I/O request
  - E.g., packet header and packet body
  - Buffer descriptor itself is DMA'ed!
- CPU and device I/O share a queue of buffer descriptors
  - I/O device reads from front
  - CPU fills at tail
- Interrupt only if buffer empties/fills

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?

Interrupt Vector

- Table set up by OS kernel; pointers to code to run on different events (in xv6, vectors.pl)



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

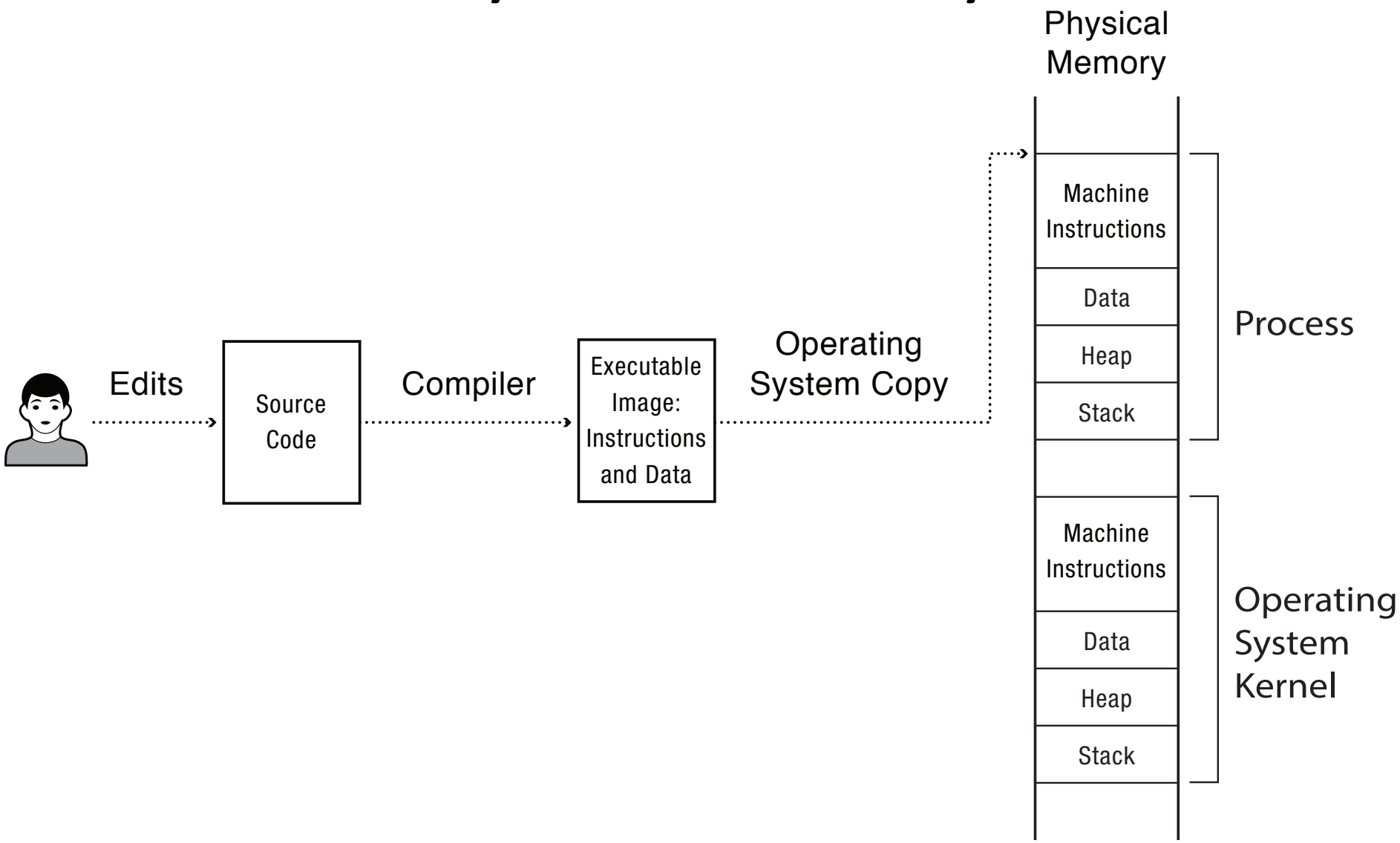
Challenge: Saving/Restoring State

- We need to be able to interrupt and transparently resume execution
  - I/O device signals I/O completion
  - Periodic hardware timer to check if app is hung
  - Multiplexing multiple apps on a single CPU
  - Code unaware it has been interrupted!
- Not just the program counter
  - Condition codes, registers used by interrupt handler, …

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

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

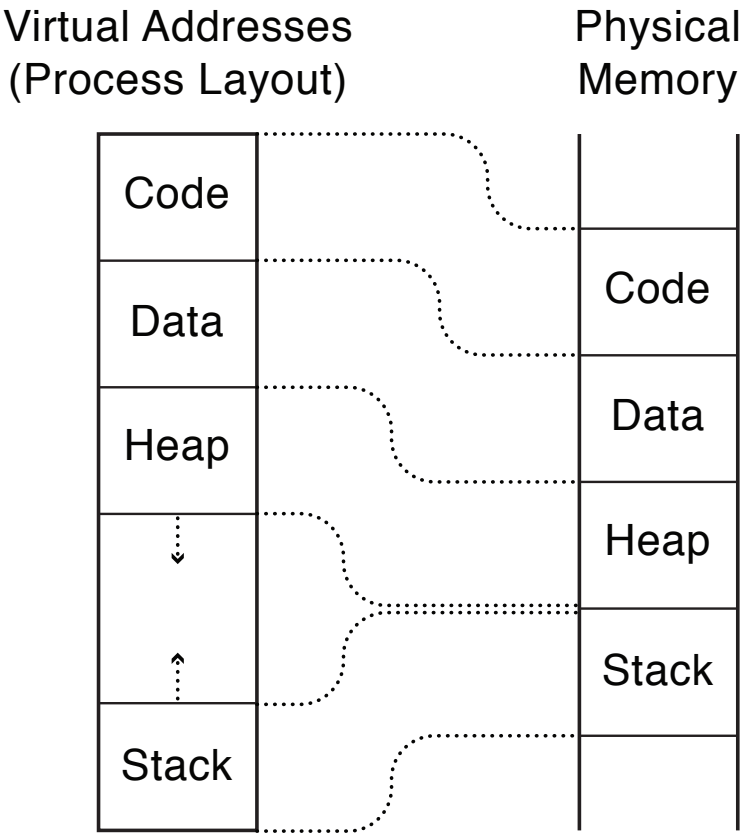
Privileged instructions

- Examples?

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

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

- With an object-oriented language and compiler, only an object's methods can access the internal data inside an object. If the operating system only ran programs written in that language, would it still need hardware memory address protection?
- What if the contents of every object were encrypted except when its method was running, including the OS?

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

Kernel->User Mode Switch

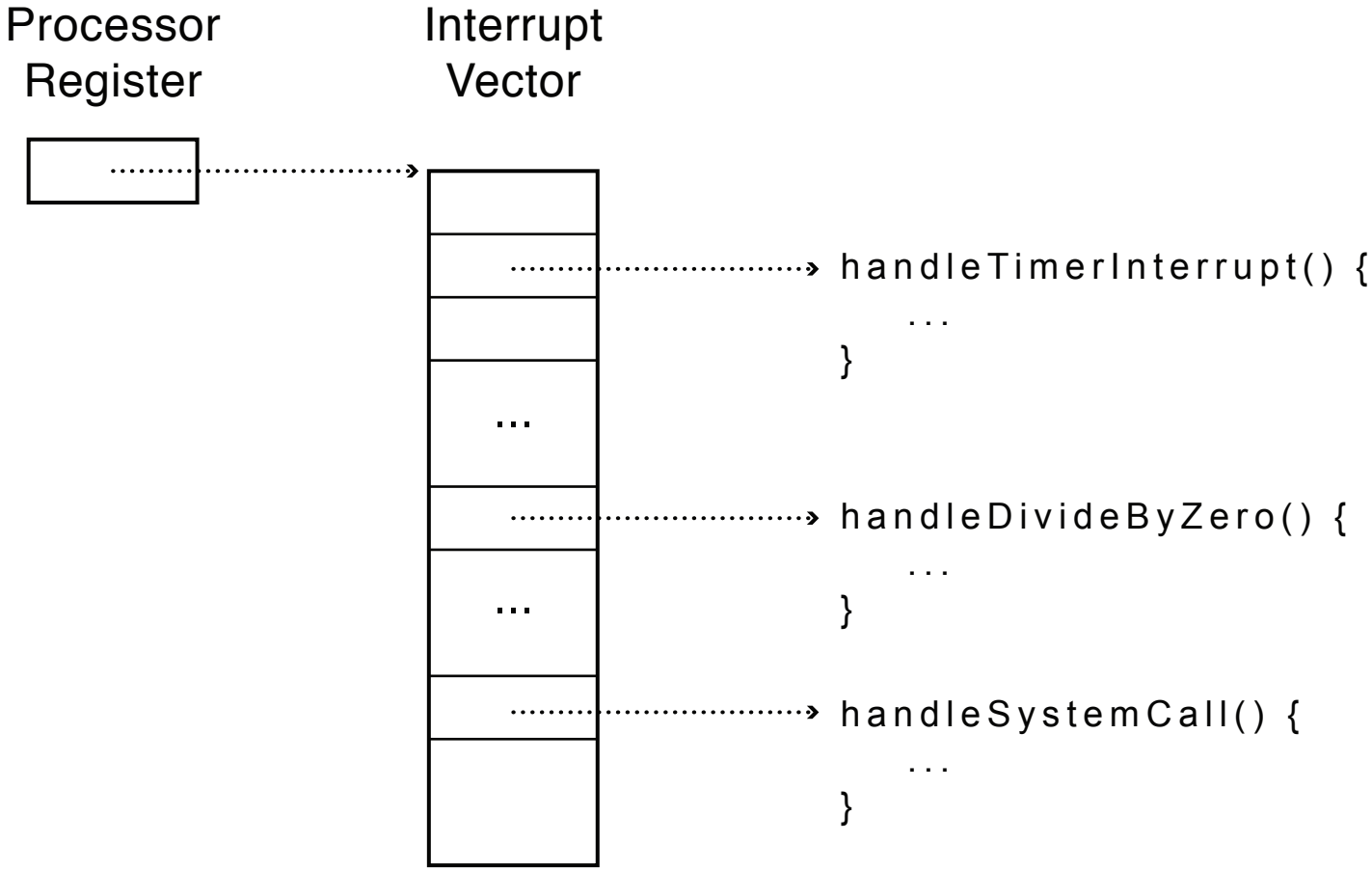
- From kernel mode to user mode
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall (UNIX signal)
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

How do we take traps safely?

- Interrupt or trap 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

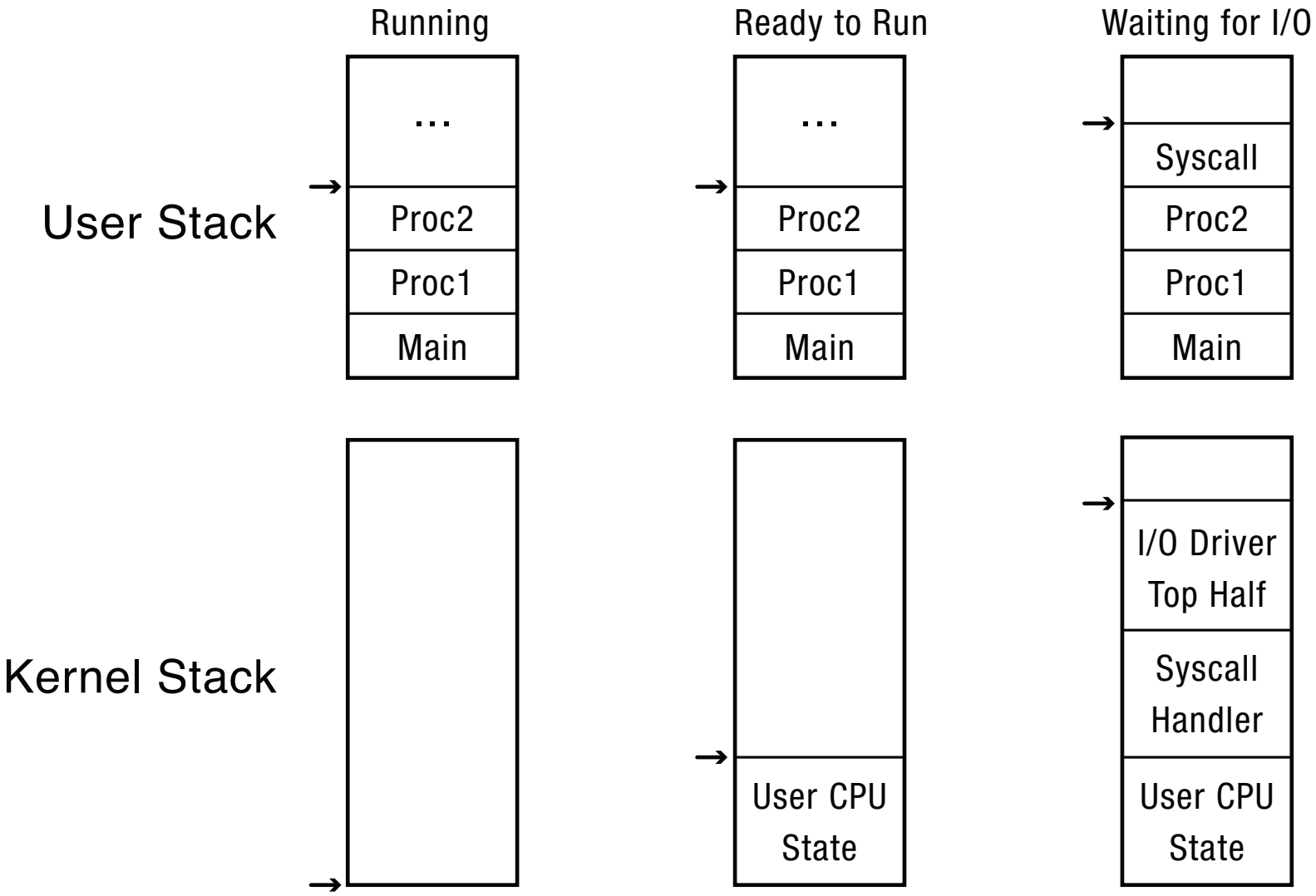
- Table set up by OS kernel; pointers to code to run on different events



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

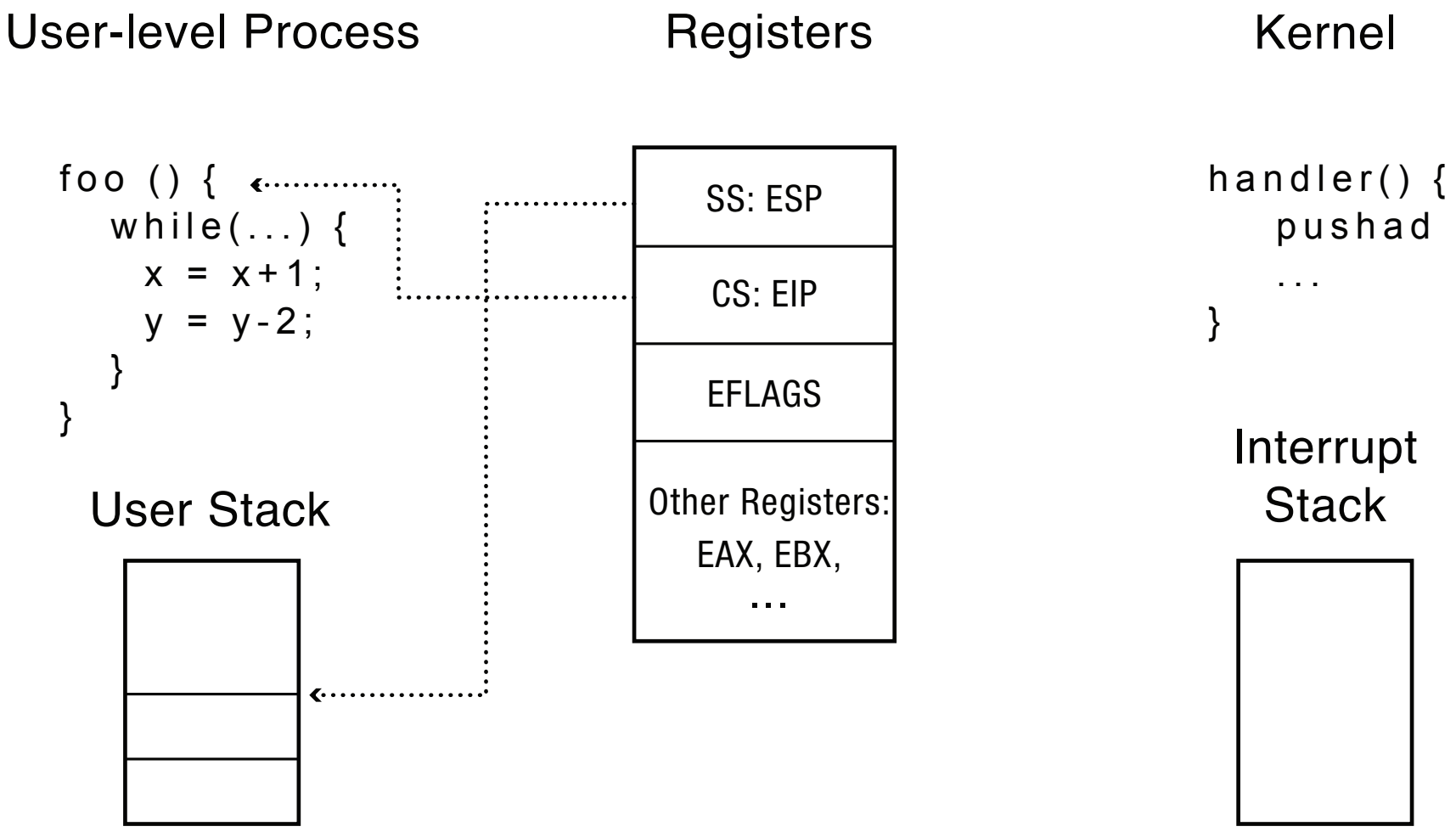
Case Study: MIPS Interrupt/Trap (Hardware Support)

- 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)

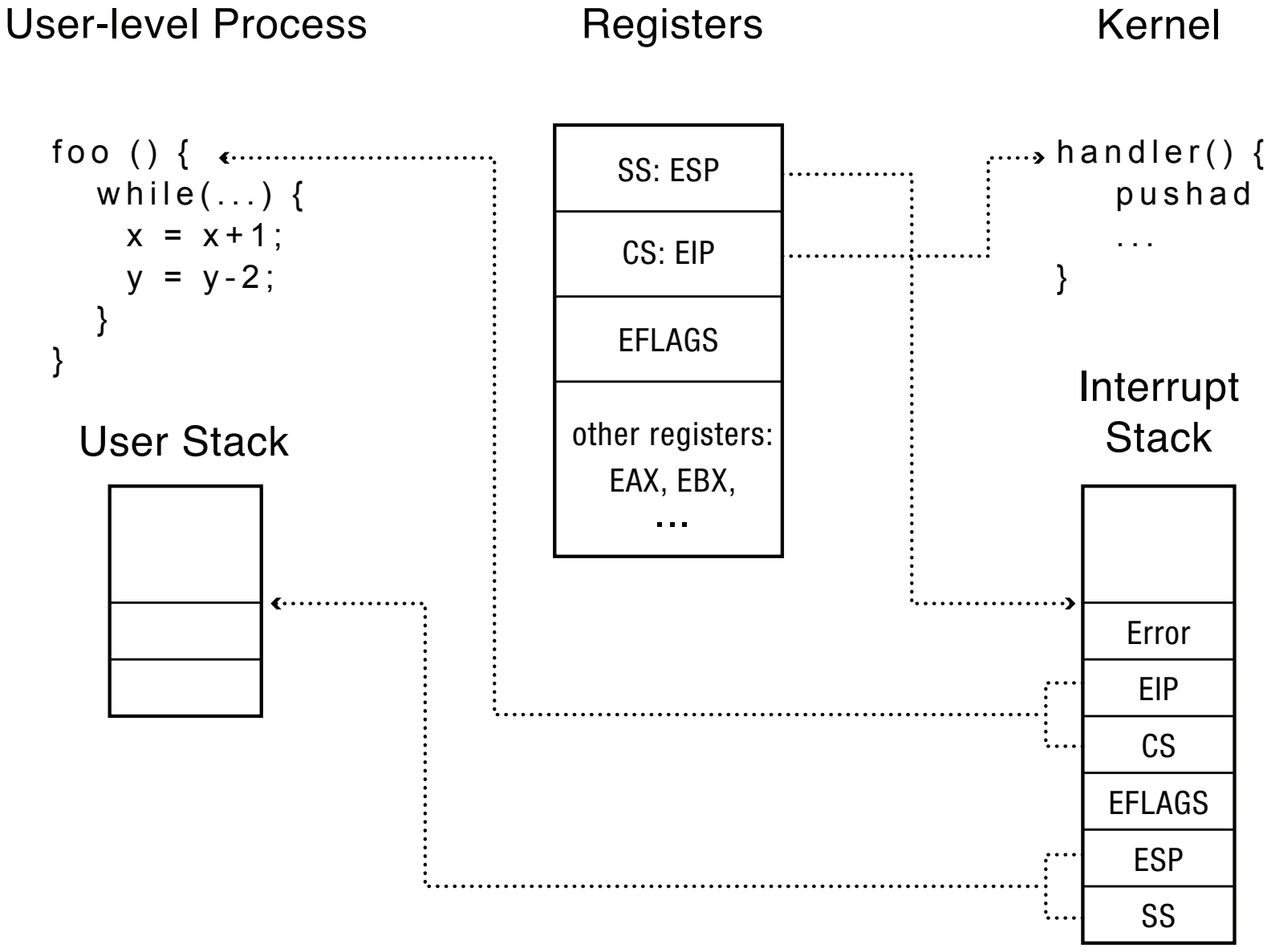
Case Study: x86 Interrupt (Hardware Support)

- 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

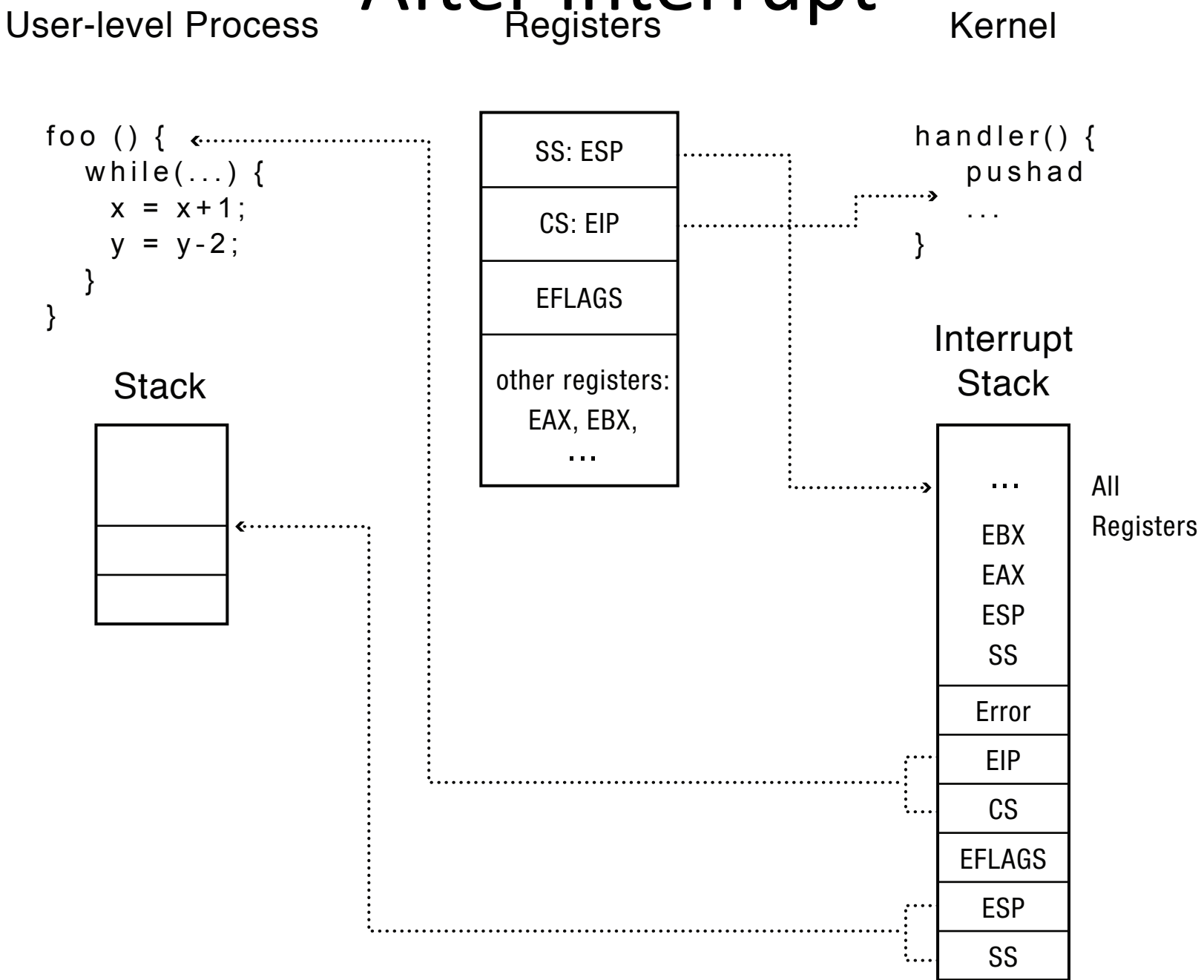
Before Interrupt



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?

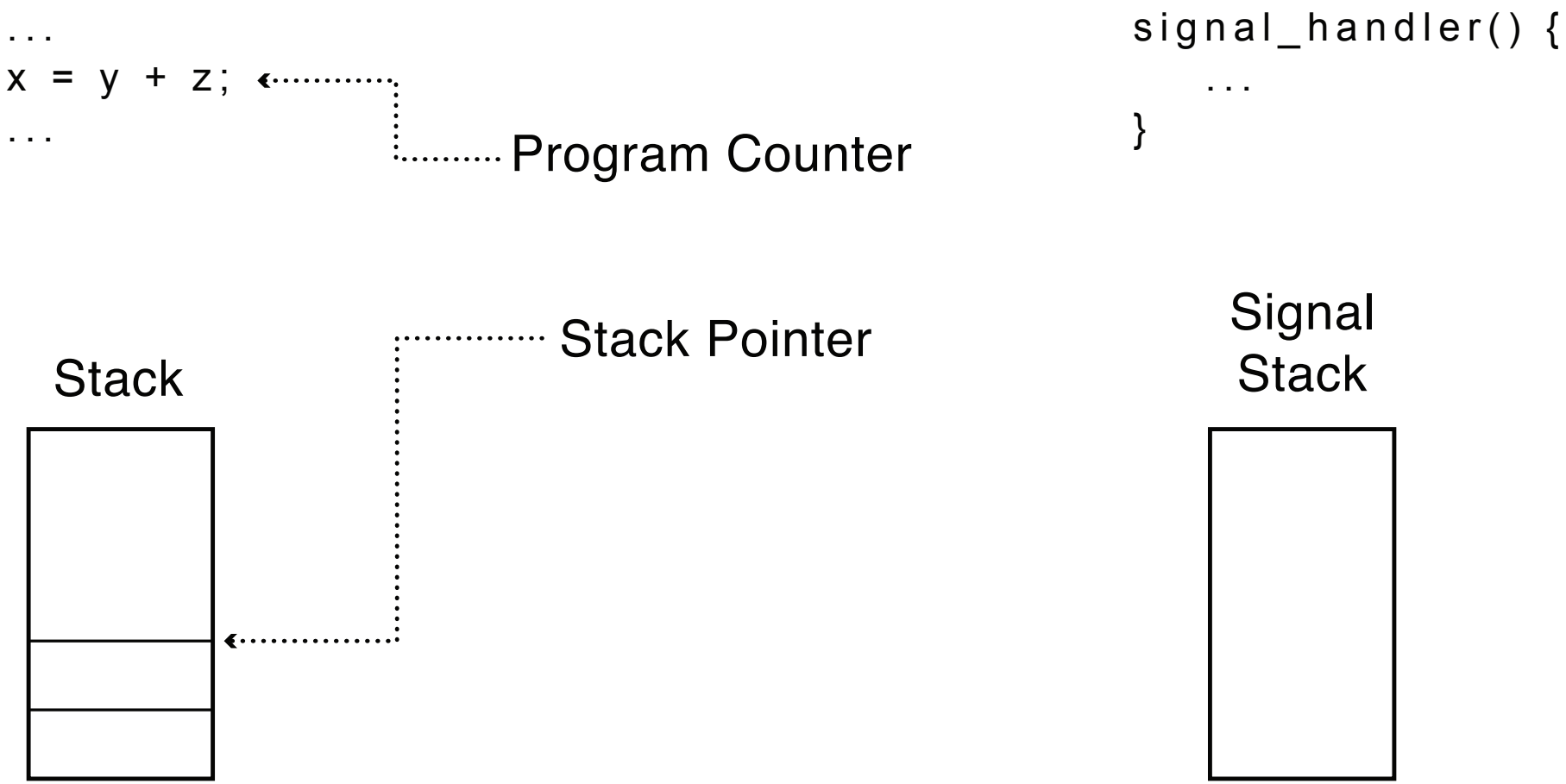
Upcall: User-level event delivery

- 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

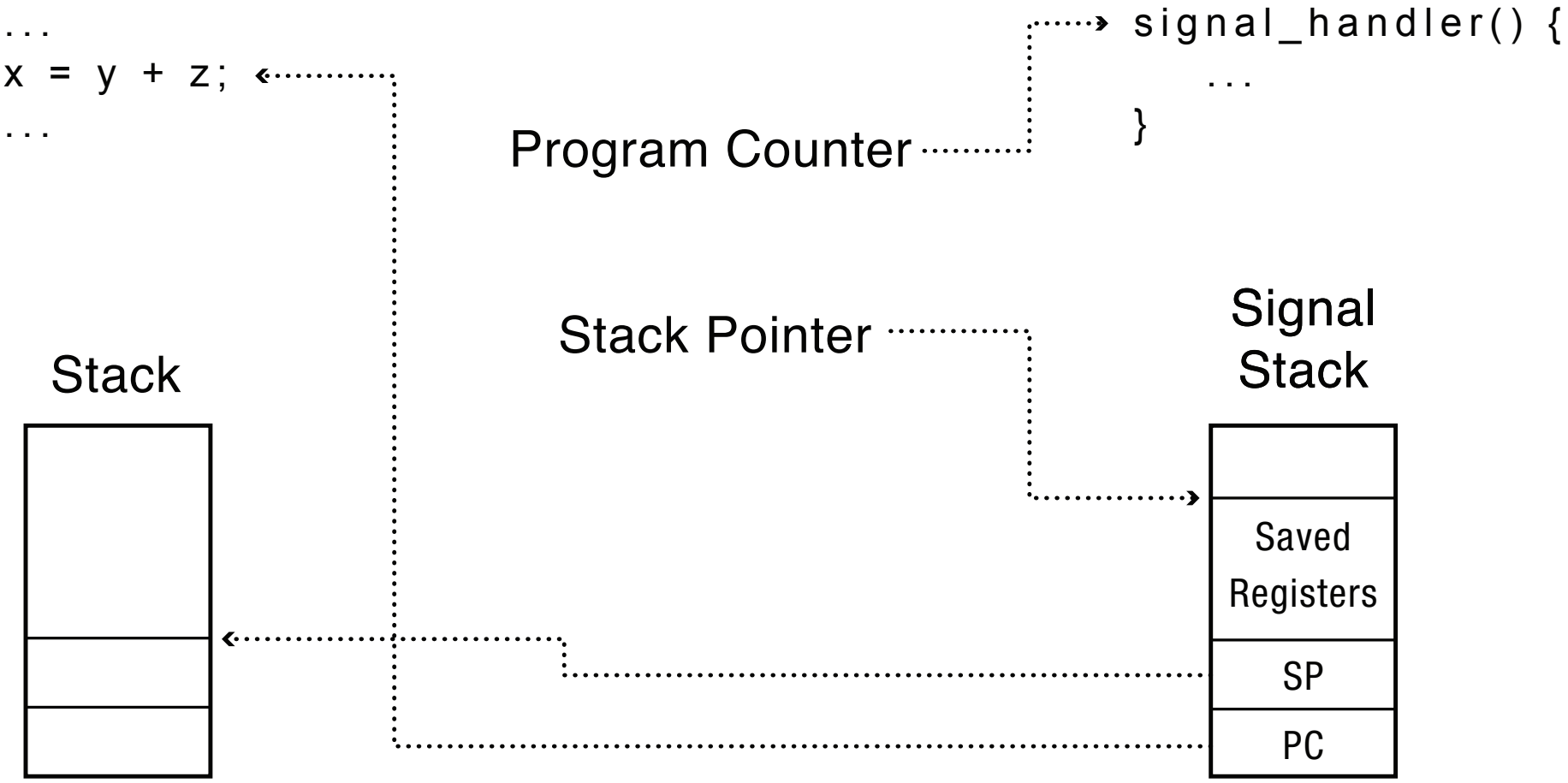
Upcalls vs Interrupts

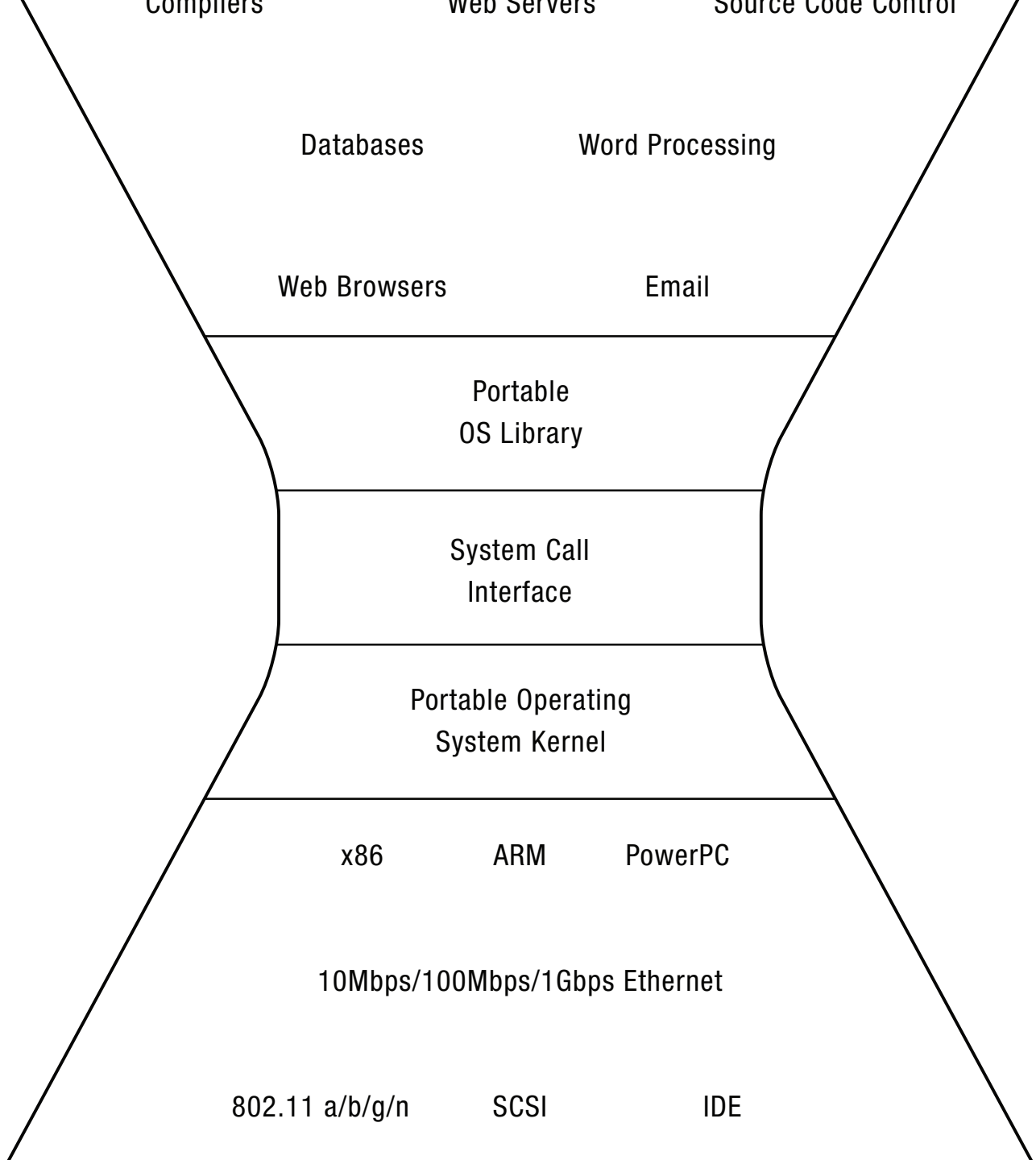
- Signal handlers = interrupt vector
- Signal stack = interrupt stack
- Automatic save/restore registers = transparent resume
- Signal masking: signals disabled while in signal handler

Upcall: Before



Upcall: During




Compilers
Web Servers
Source Code Control
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

```
main () {
   file_open(arg1, arg2);
}
```

(1) (6)

User Stub

```
file_open(arg1, arg2) {
   push #SYSCALL_OPEN
   trap
   return
}
```

(2)

Hardware Trap

Trap Return

(5)

Kernel

```
file_open(arg1, arg2) {
      // do operation
}
```

(3) (4)

Kernel Stub

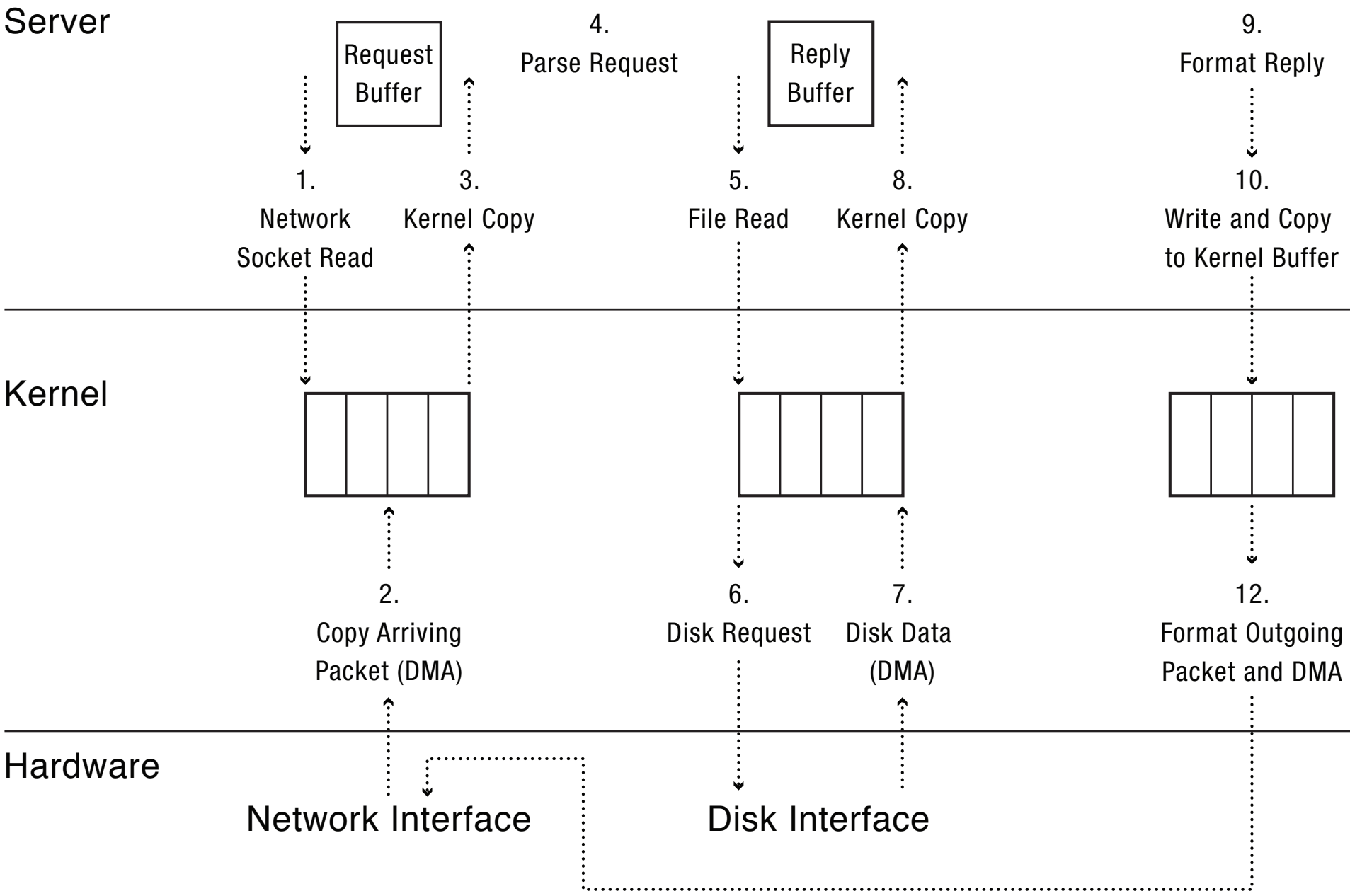
```
file_open_handler() {
   // copy arguments
   //   from user memory
   //   check arguments
   file_open(arg1, arg2);
   // copy return value
   //   into user memory
   return;
}
```

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

QEMU and xv6

- 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

The Programming Interface

Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells

- Example: to compile a C program

```
cc –c sourcefile1.c
cc –c sourcefile2.c
ln –o program sourcefile1.o sourcefile2.o
```

Question

- If the shell runs at user-level, what system calls does it make to run each of the programs?
  - Ex: cc, ln

Windows CreateProcess

- System call to create a new process to run a program
  - Create and initialize the process control block (PCB) in the kernel
  - Create and initialize a new address space
  - Load the program into the address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''
  - Inform the scheduler that the new process is ready to run

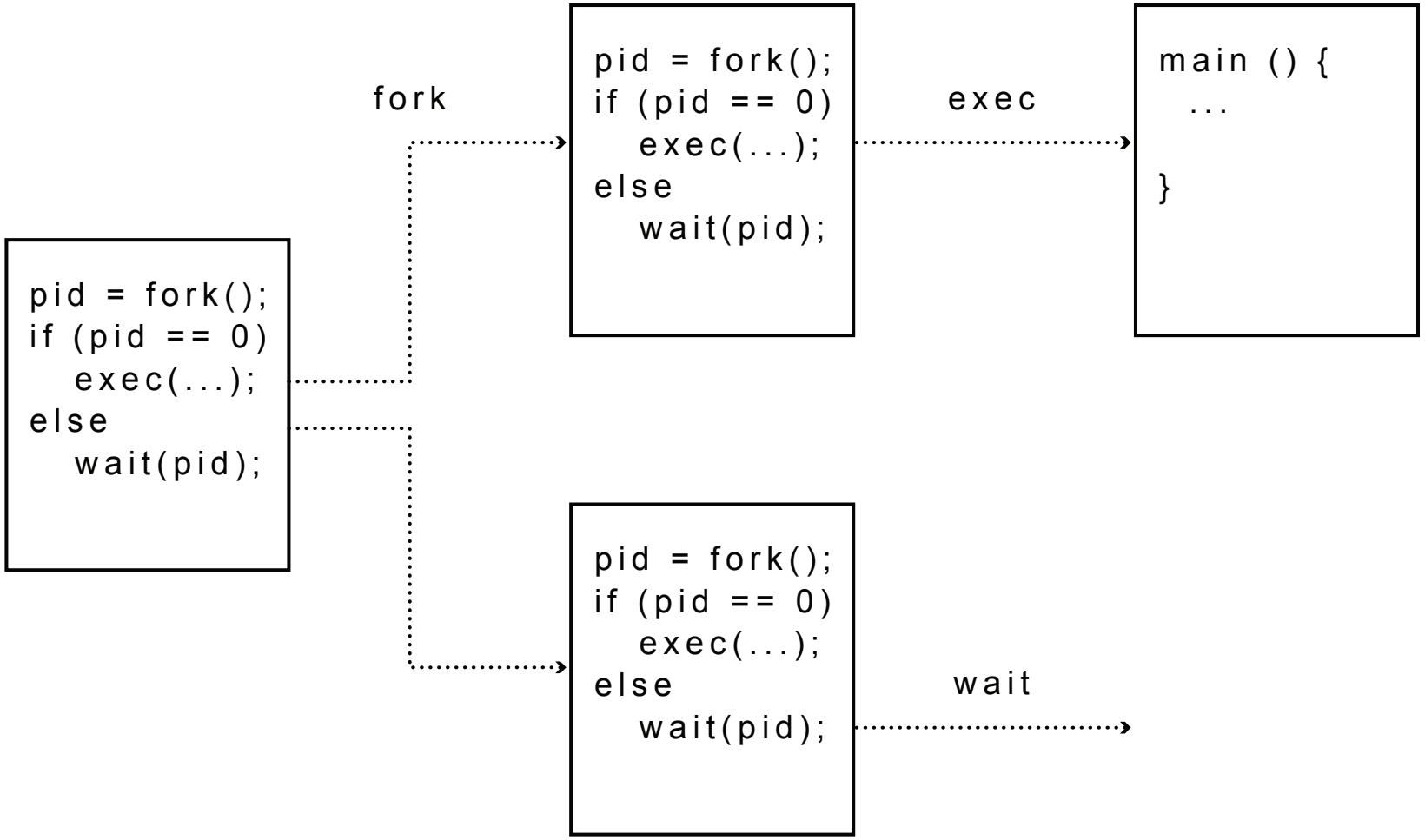
Windows CreateProcess API (simplified)

```
if (!CreateProcess(
    NULL,           // No module name (use command line)
    argv[1],        // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,              // No creation flags
    NULL,           // Use parent's environment block
    NULL,           // Use parent's starting directory
    &si,            // Pointer to STARTUPINFO structure
    &pi )           // Pointer to PROCESS_INFORMATION structure
)
```

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 Process Management



Question: What does this code print?

```
int child_pid = fork();
if (child_pid == 0) {          // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                       // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

Questions

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

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
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

Implementing UNIX exec

- Steps to implement UNIX fork
  - 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''

UNIX I/O

- Uniformity
  - All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use
  - Open returns a handle (file descriptor) for use in later calls on the file
- Byte-oriented
- Kernel-buffered read/write
- Explicit close
  - To garbage collect the open file descriptor

Implementing a Shell

```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
   child_pid = fork();      // create a child process
    if (child_pid == 0) {
       exec(prog, args);       // I'm the child process.  Run program
      // NOT REACHED
    } else {
       wait(child_pid);       // I'm the parent, wait for child
       return 0;
    }
}
```

UNIX File System Interface

- UNIX file open is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn't exist, return an error
    - If file doesn't exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn't empty, nix it then open
    - If file exists but isn't empty, return an error
    - …

Interface Design Question

- Why not separate syscalls for open/create/exists?

```
if (!exists(name))
    create(name);   // can create fail?
fd = open(name);   // does the file exist?
```

UNIX Retrospective

- Designed for computers 10^-8 slower than today
- Radical simplification relative to Multics
- Is UNIX a Christensen disruptive change?
  - Definition of disruptive?
- Are other technical changes disruptive?
  - Internet? Web?
  - Multicore? Mapreduce?
  - SQL? C? Python?

UNIX Retrospective

- Key ideas behind the project
  1. ease of programming and interactive use
  2. size constraint: underpowered machine, small memory.
  3. Eat your own dogfood. UNIX dev on UNIX.
- Any missing goals?
  - Performance. What if they had put performance first?
  - Portability. What if they had put portability first?

UNIX Retrospective

- What were the principal technical innovations in UNIX?
- Are those still applicable in an age of laptops, smartphones, and cloud servers?
- Were those technical innovations responsible for UNIX's success?
- Retrospectively, what's missing from UNIX, if anything?

UNIX Design Choices

- How does UNIX do X? How did earlier and later systems do them? How might they be done in the future?
- What's good about the UNIX approach? What are the downsides?

UNIX I/O

- All I/O done as a byte stream
  - TCP hadn't been invented yet, but same idea
- Talk to files the same way you talk to:
  - process, disk, tape drive, keyboard, network …
  - What if one side is slow and the other is fast? One side needs to wait; other needs a kernel buffer
- Allows programs to be composed easily, and to be kept simple: bytes in/bytes out
  - Compile app as many components, via pipes

UNIX File System

- Hierarchical naming
- Flat byte storage
- Directories as files
- Dynamic extents

UNIX Security Model

- Kernel/user mode
- Kernel protects itself from user mistakes (e.g., user can’t modify directories)
- Superuser can do anything

UNIX Process Management

- Fork/exec
- Inherit open files from parent
- Shell: redirection, pipes

UNIX Memory Management

- Process swapping, no demand paging
- In earlier systems
  - segmented paging, virtual machines
- In later systems
  - segmented paging, virtual machines
  - hardware abstraction layer, for portability