The Kernel Abstraction

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

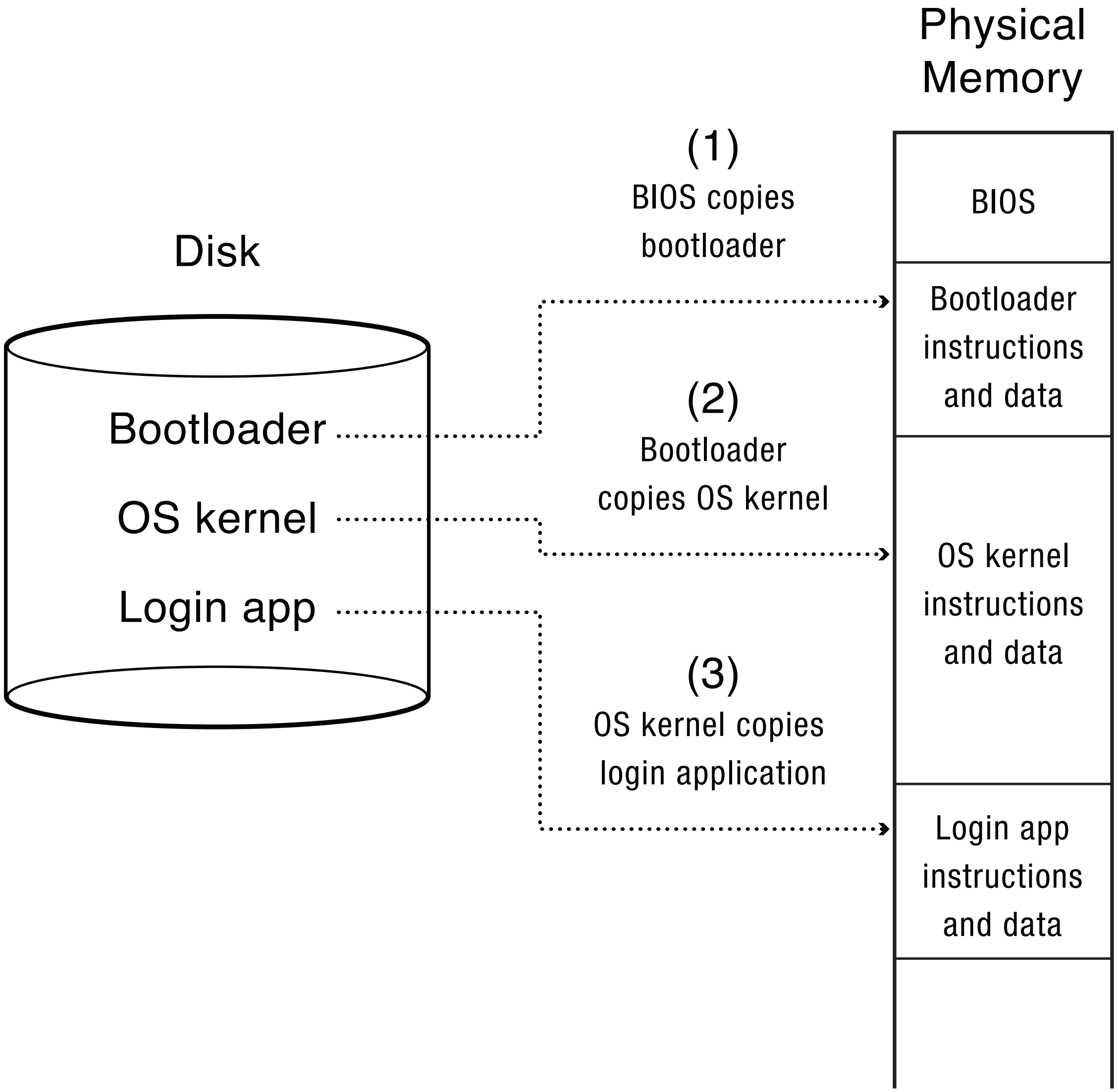
- Understanding -> design -> code
  - not the opposite
- Form a hypothesis that explains the bug
  - Which tests work, which don't. Why?
  - Add tests to narrow possible outcomes
- Use best practices
  - Always walk through your code line by line
  - Module tests – narrow scope of where problem is
 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

ABET

You can't debug effectively without this:

b. Ability to design and conduct experiments, analyze and interpret data.

Booting


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

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
  - Buffer descriptor: sequence of DMA's
    - E.g., packet header and packet body
  - Queue of buffer descriptors
    - Buffer descriptor itself is DMA'ed

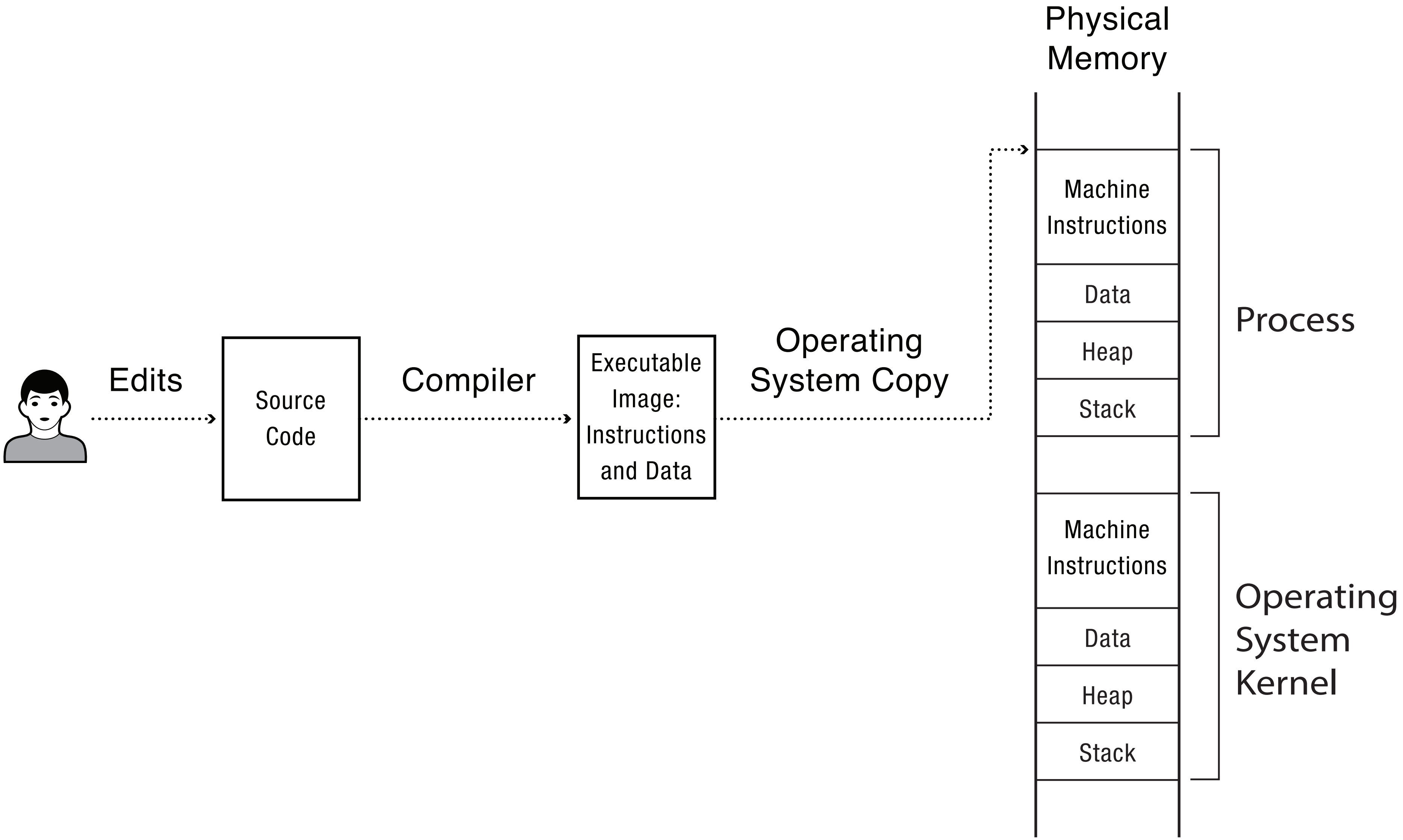
Device Interrupts

- 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?

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

A Problem


Physical Memory
Edits
Source Code
Compiler
Executable Image: Instructions and Data
Operating System Copy
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
Operating System Kernel

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?

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)

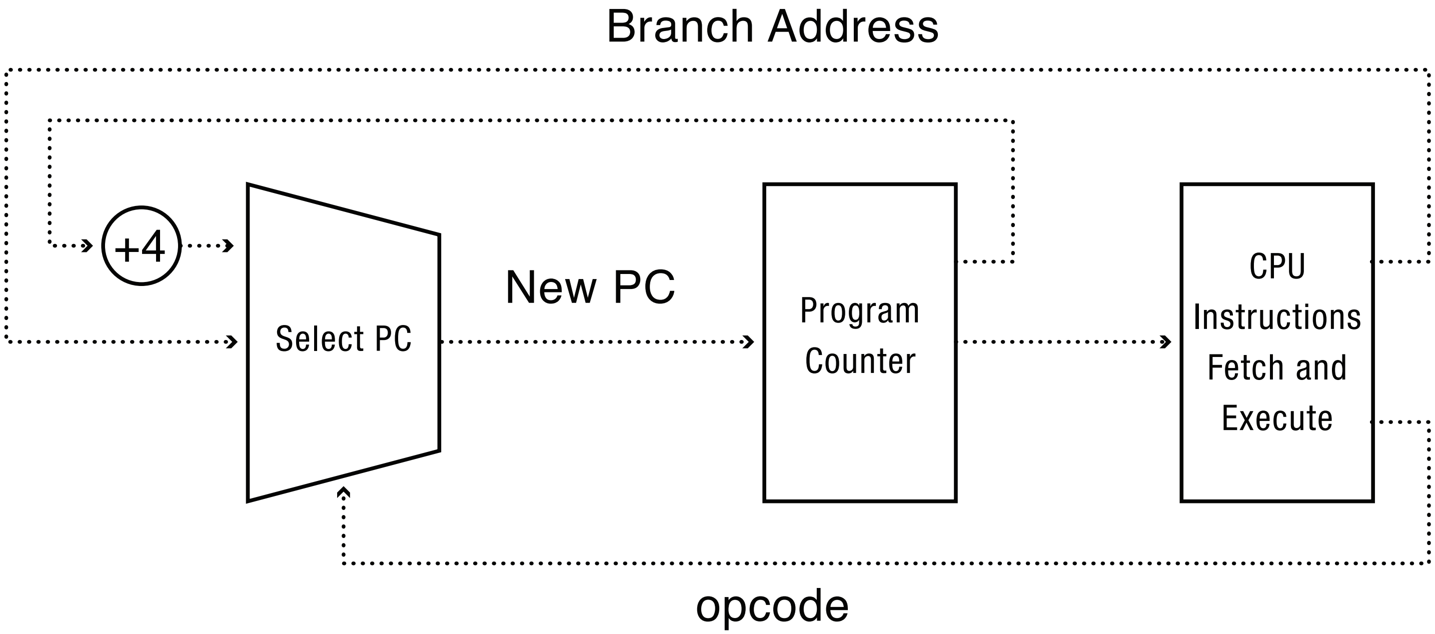
Thought Experiment

- 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!

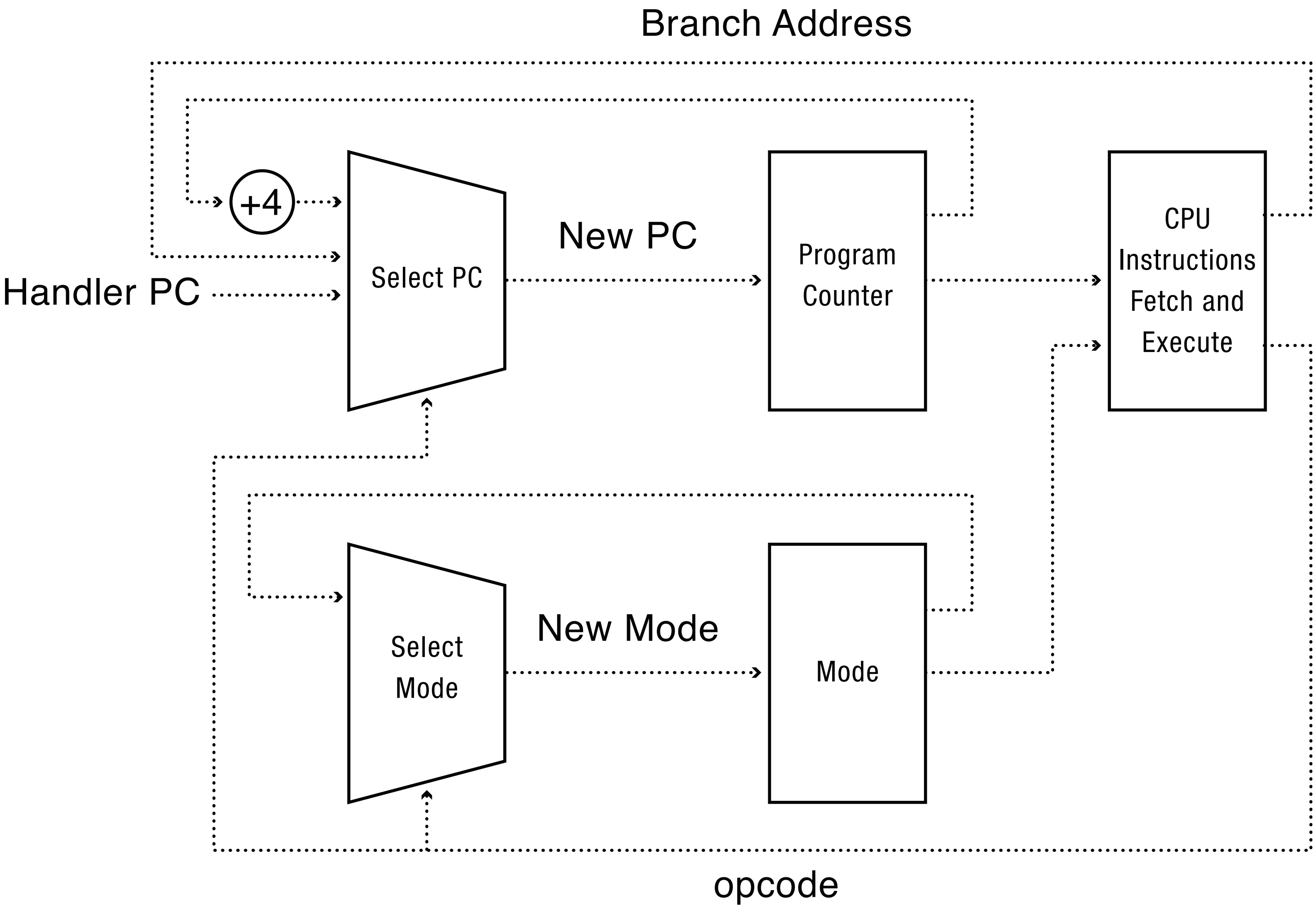
Hardware Support: Dual-Mode Operation

- 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 Model of a CPU



A CPU with Dual-Mode Operation



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

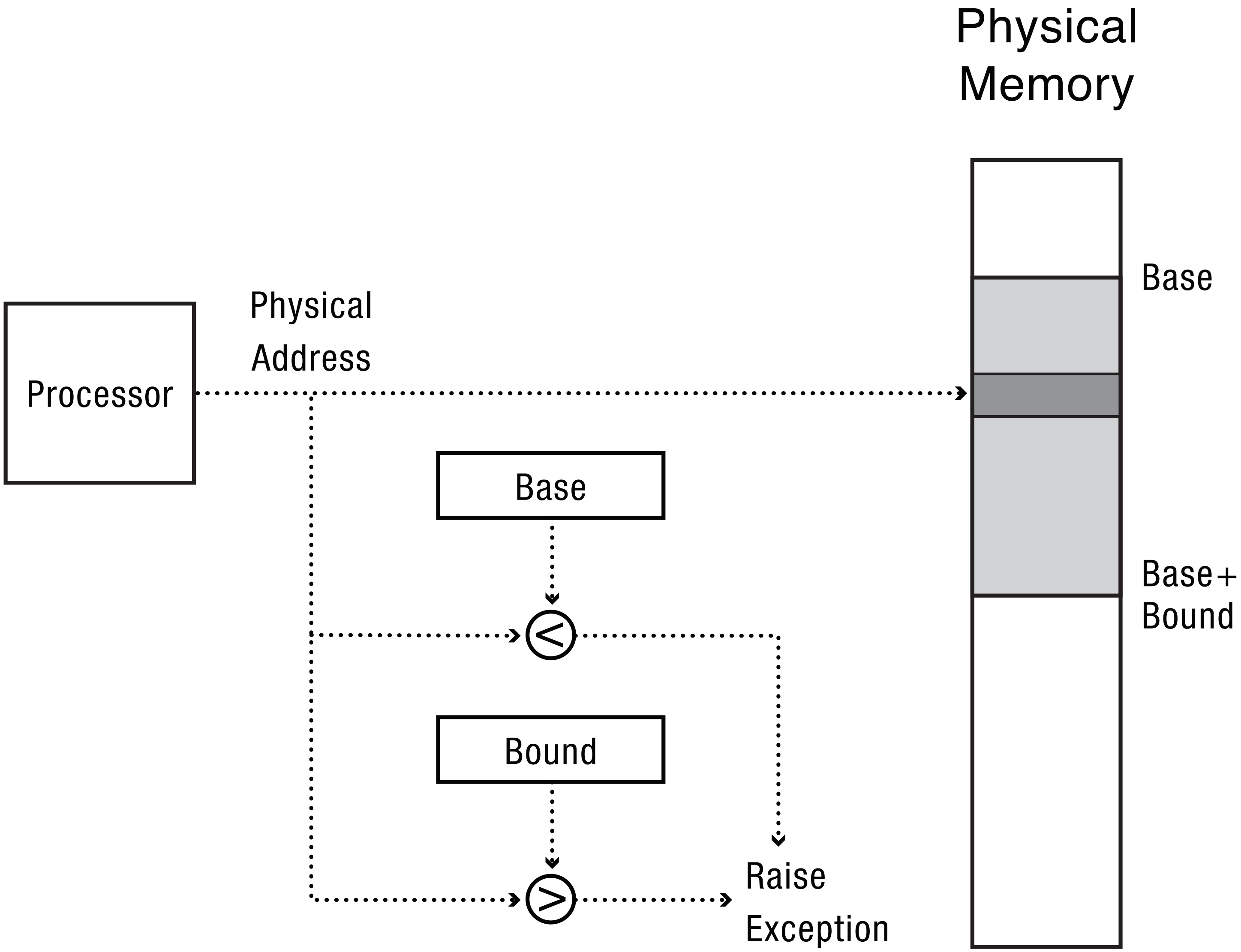
- Examples?

- 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

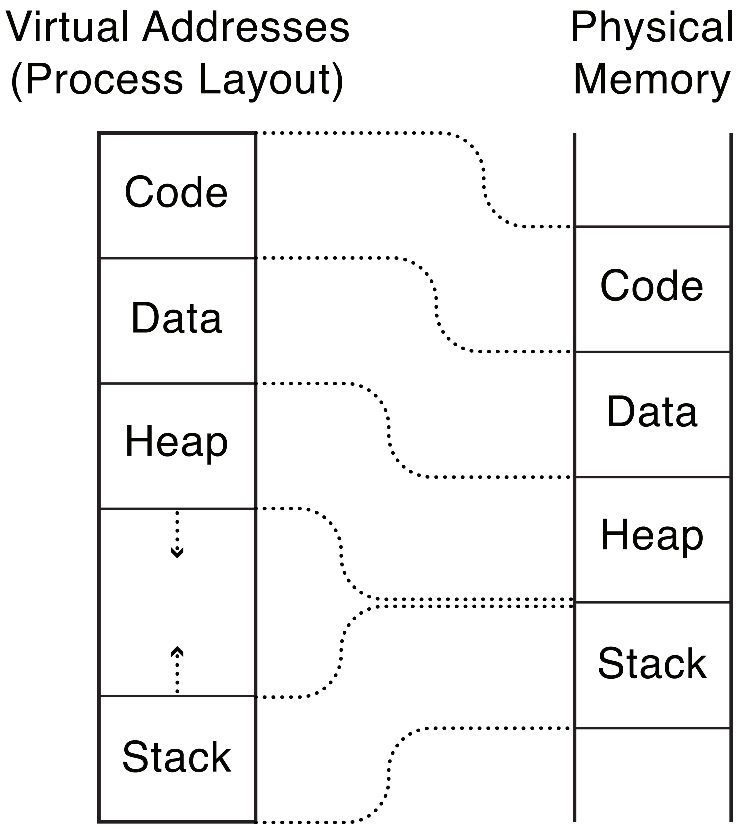


Towards Virtual Addresses

- Problems with base and bounds?

Virtual Addresses

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



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?

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

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

Question

- Examples of exceptions

- Examples of system calls

Mode Switch

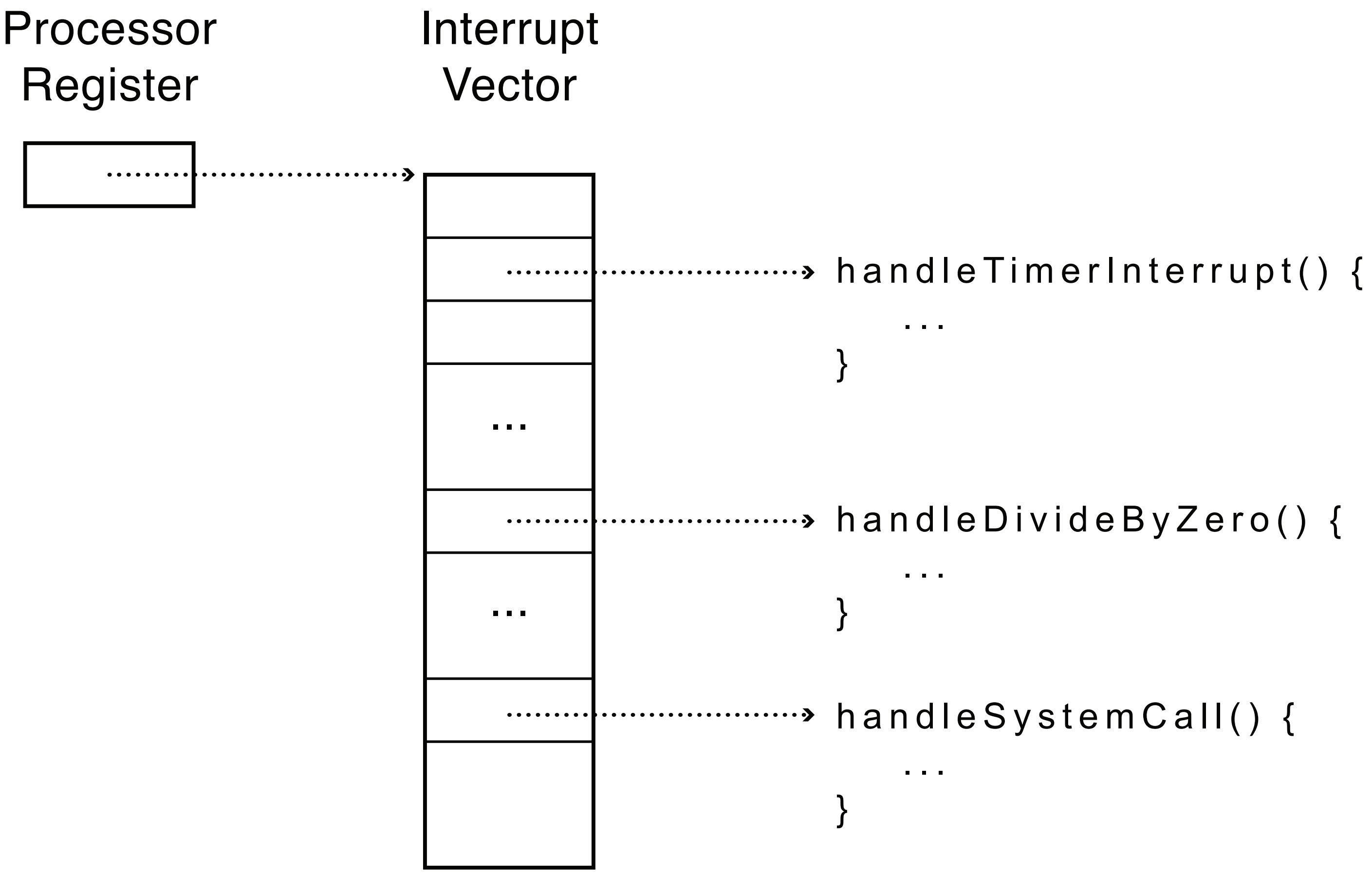
- From kernel mode to user mode
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall (UNIX signal)
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

How do we take 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

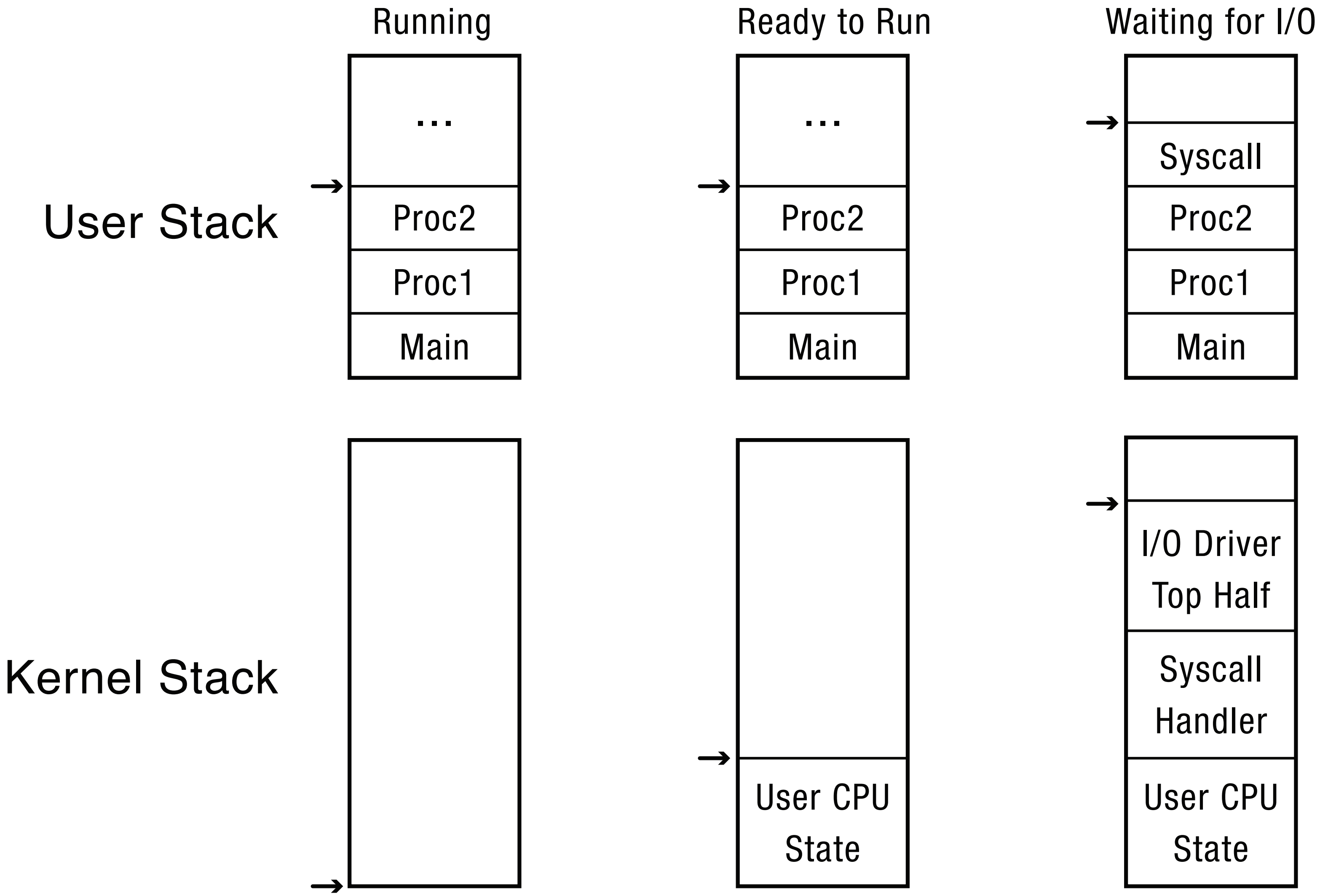
- Table set up by OS kernel; pointers to code to run on different events



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



Running

Ready to Run

Waiting for I/O

User Stack

…

Proc2

Proc1

Main

…

Proc2

Proc1

Main

Syscall

Proc2

Proc1

Main

Kernel Stack

User CPU State

I/O Driver Top Half

Syscall Handler

User CPU State

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

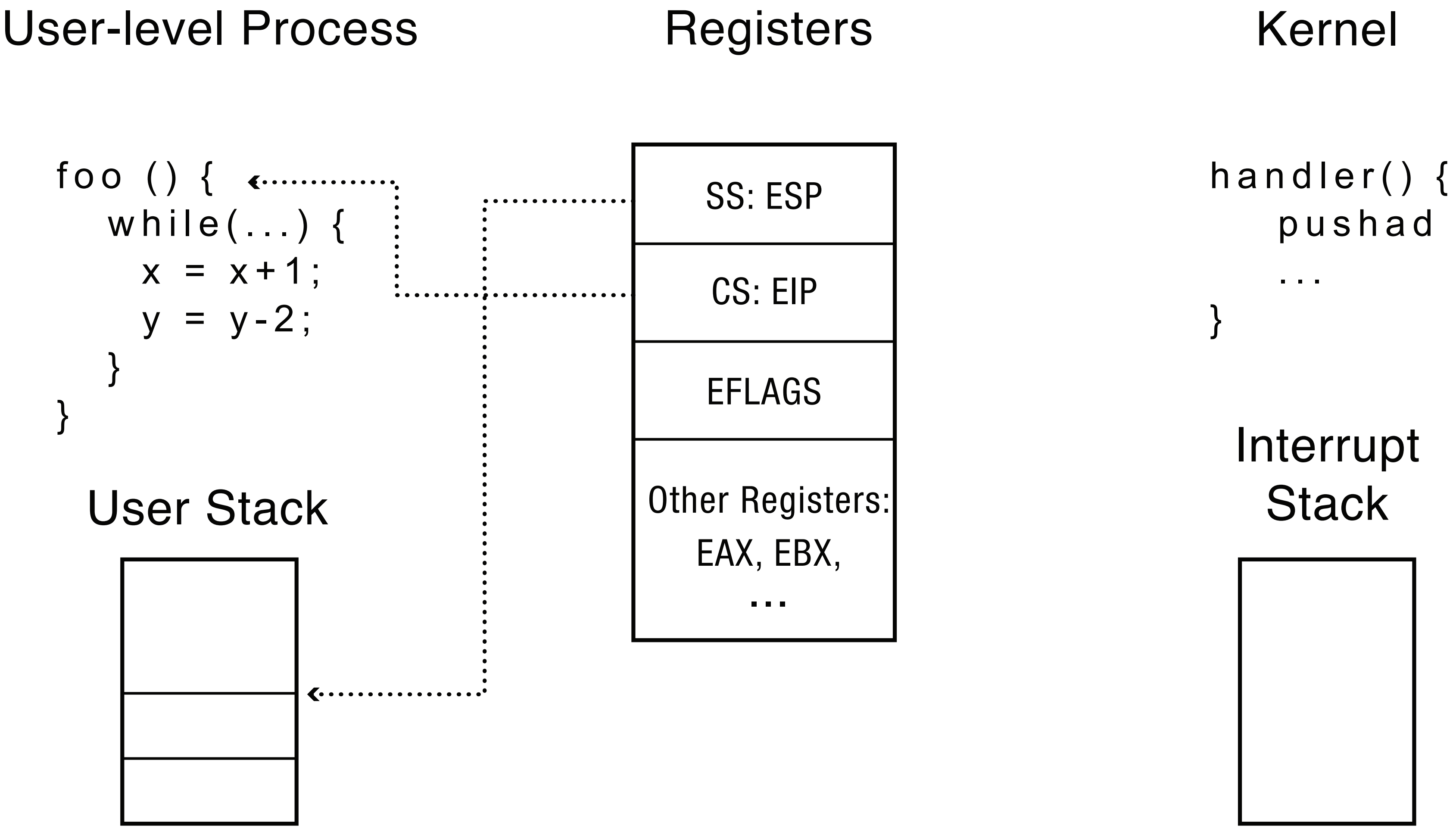
Case Study: MIPS Interrupt/Trap

- Two entry points: TLB miss handler, everything else
- Save type: syscall, exception, interrupt
  - And which type of interrupt/exception
- Save program counter: where to resume
- Save old mode, interruptable bits to status register
- Set mode bit to kernel
- Set interrupts disabled
- For memory faults
  - Save virtual address and virtual page
- Jump to general exception handler

Case Study: x86 Interrupt

- Save current stack pointer
- Save current program counter
- Save current processor status word (condition codes)
- Switch to kernel stack; put SP, PC, PSW on stack
- Switch to kernel mode
- Vector through interrupt table
- Interrupt handler saves registers it might clobber

Before Interrupt



User-level Process

```
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

User Stack

Registers

SS: ESP

CS: EIP

EFLAGS

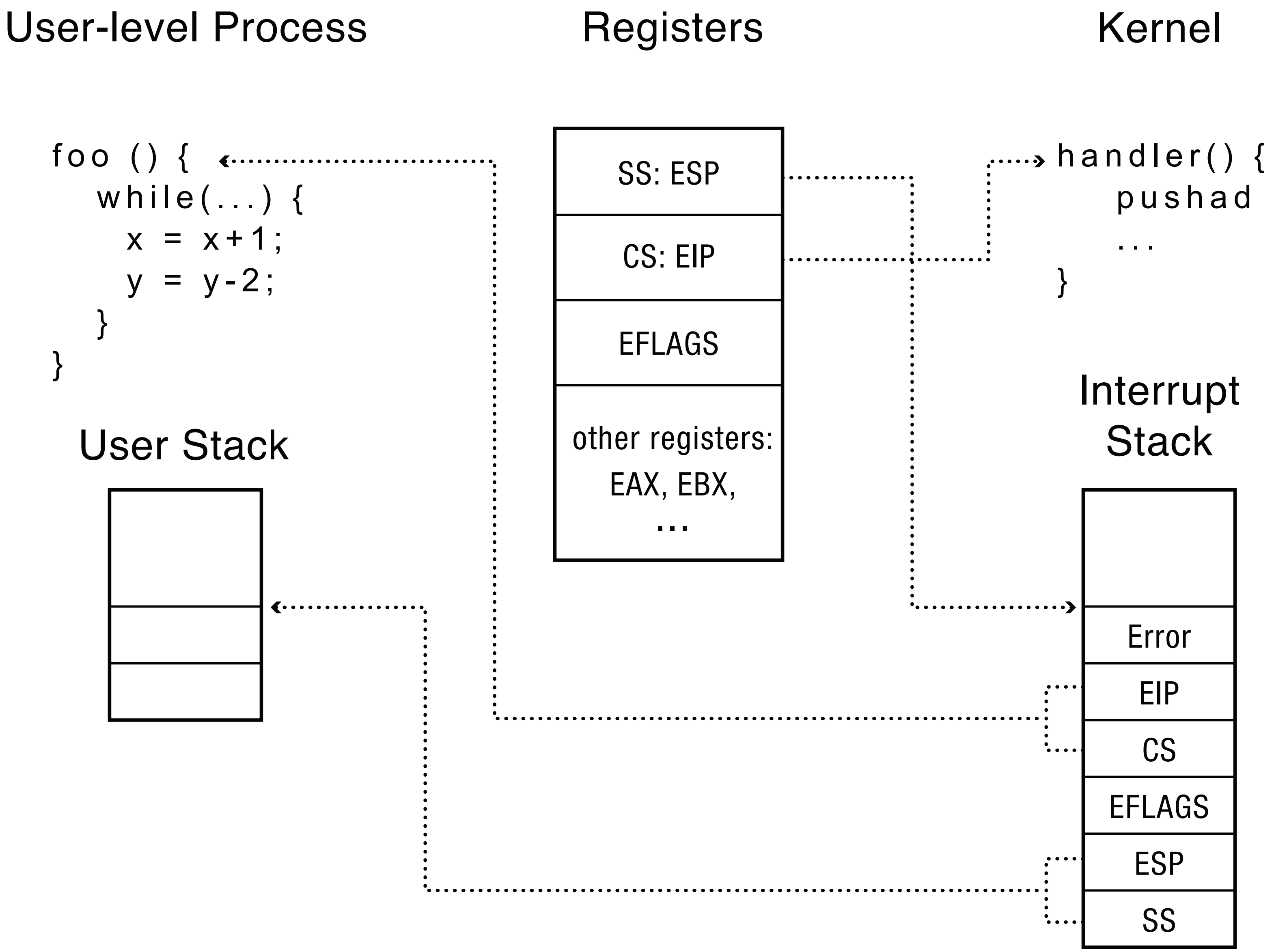
Other Registers:
EAX, EBX,
...

Kernel

```
handler() {
   pushad
   ...
}
```

Interrupt Stack

During Interrupt



User-level Process

```
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

User Stack

Registers

SS: ESP

CS: EIP

EFLAGS

other registers:
EAX, EBX,
...

Kernel

```
handler() {
   pushad
   ...
}
```

Interrupt Stack

Error

EIP

CS

EFLAGS

ESP

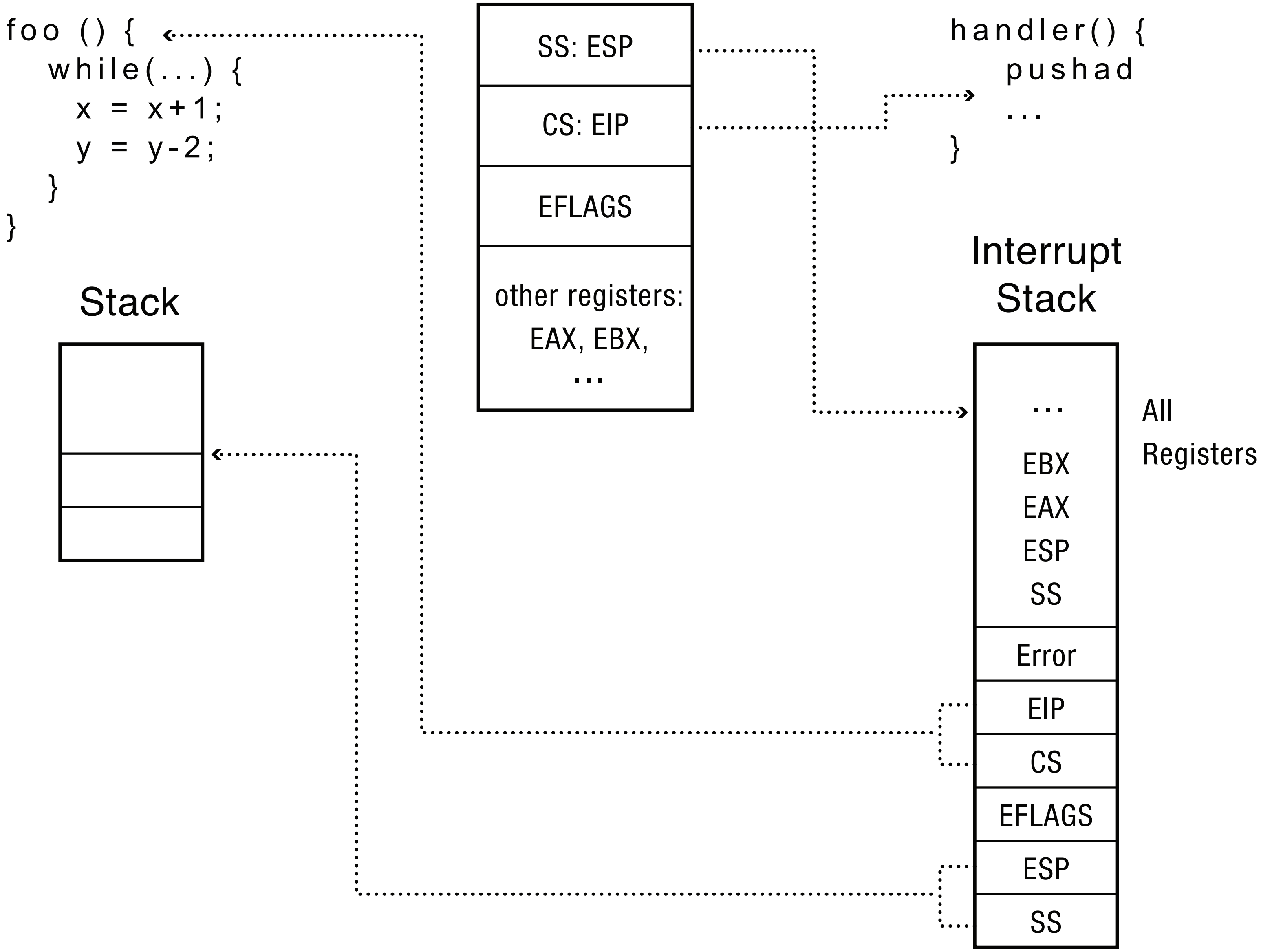
SS

After Interrupt

User-level Process

Registers

Kernel



```
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

```
handler() {
  pushad
  ...
}
```

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

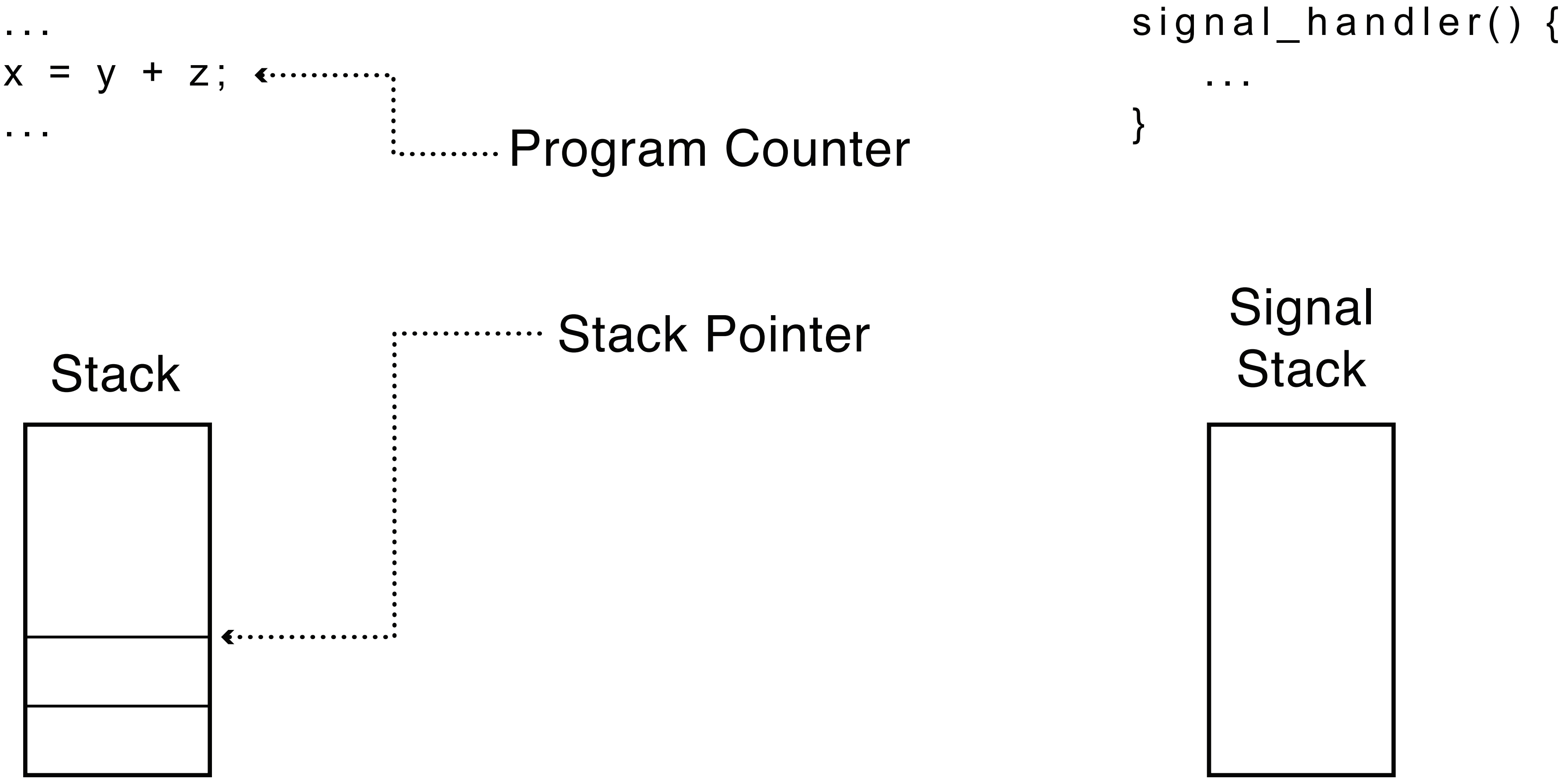
Upcall: User-level event delivery

- 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



```
...
x = y + z;
...
```

Program Counter

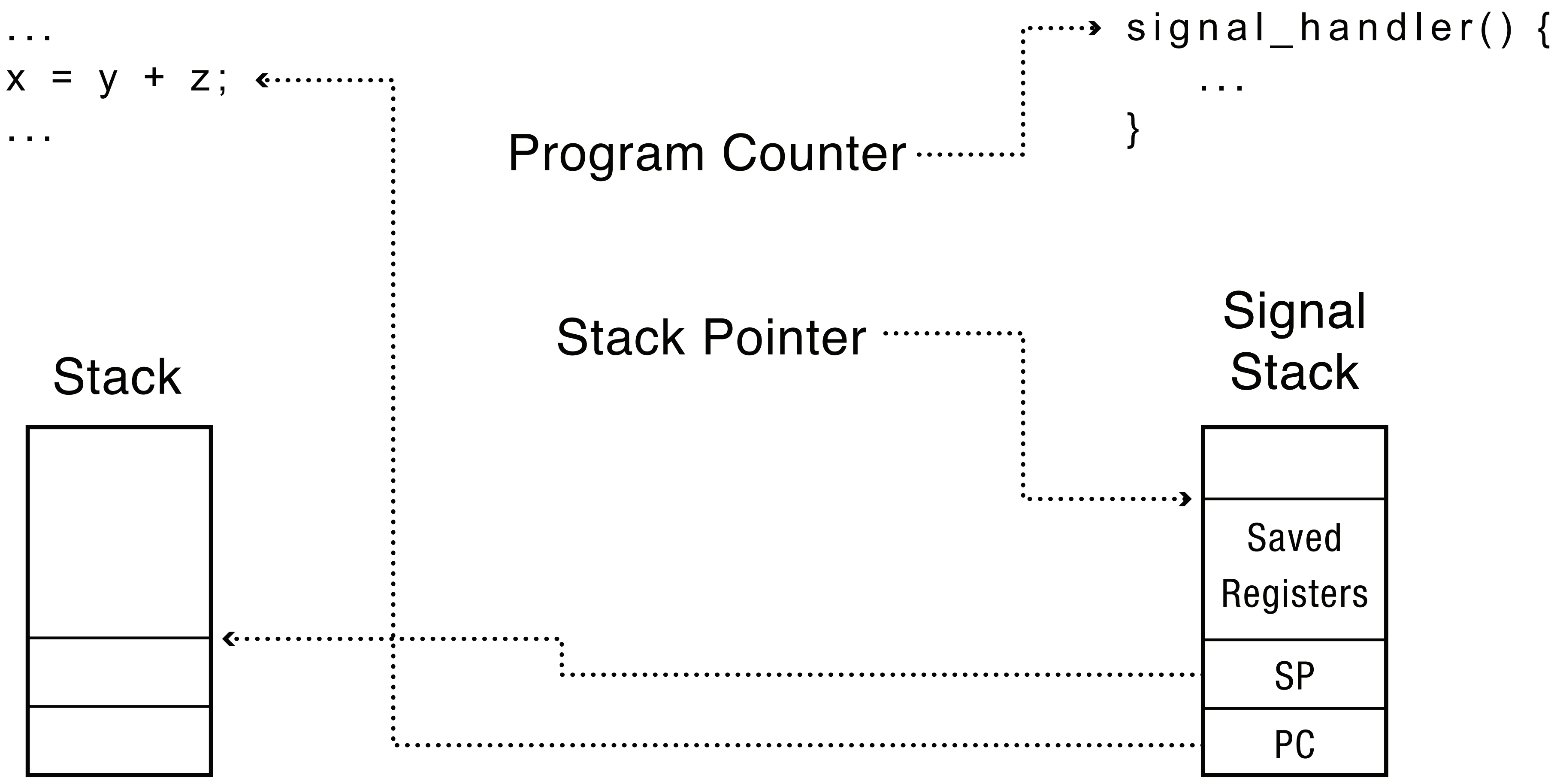
```
signal_handler() {
    ...
}
```

Stack Pointer

Stack

Signal Stack

Upcall: During



```
...
x = y + z;
...
```

```
signal_handler() {
    ...
}
```

Program Counter

Stack Pointer

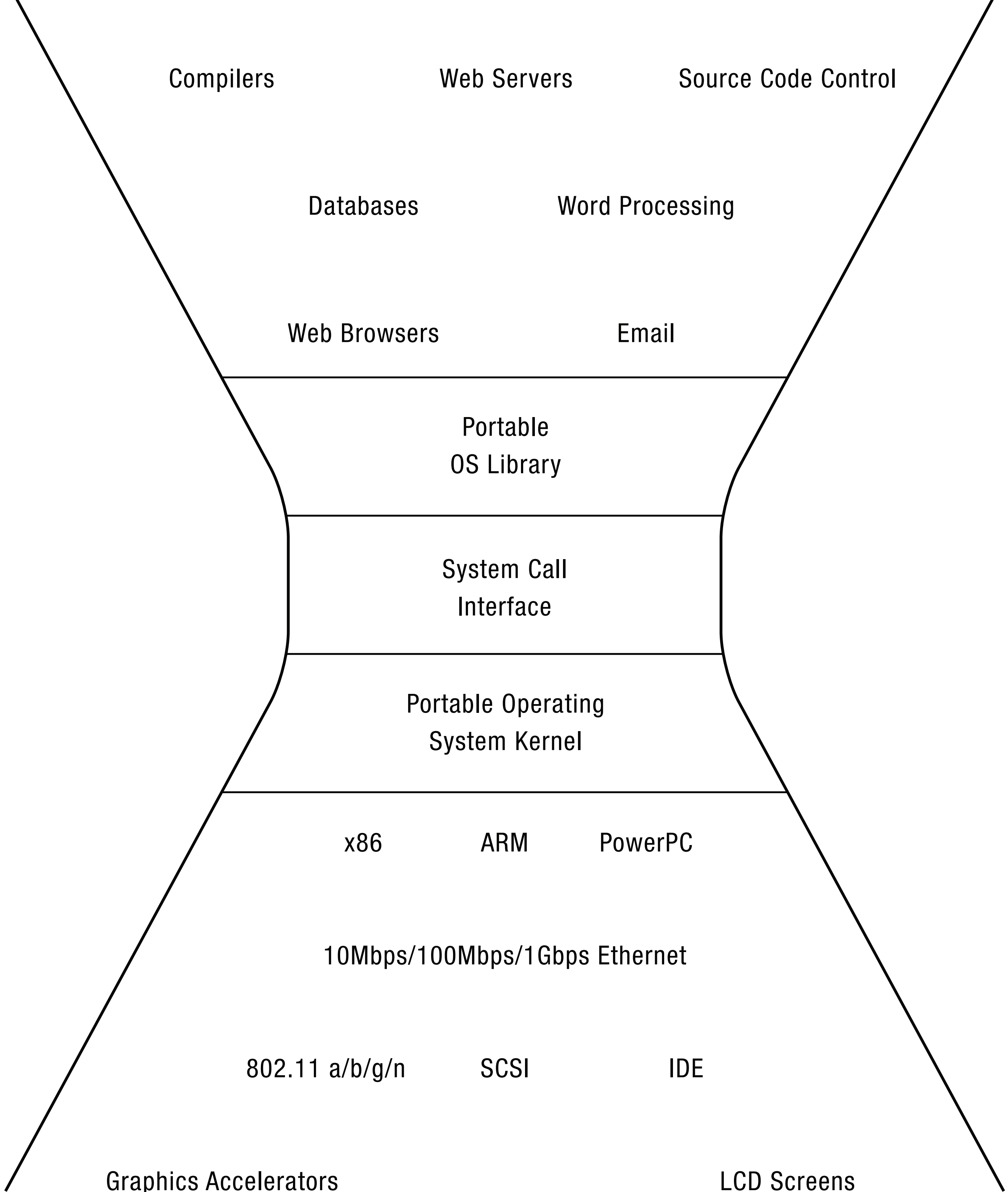
Stack

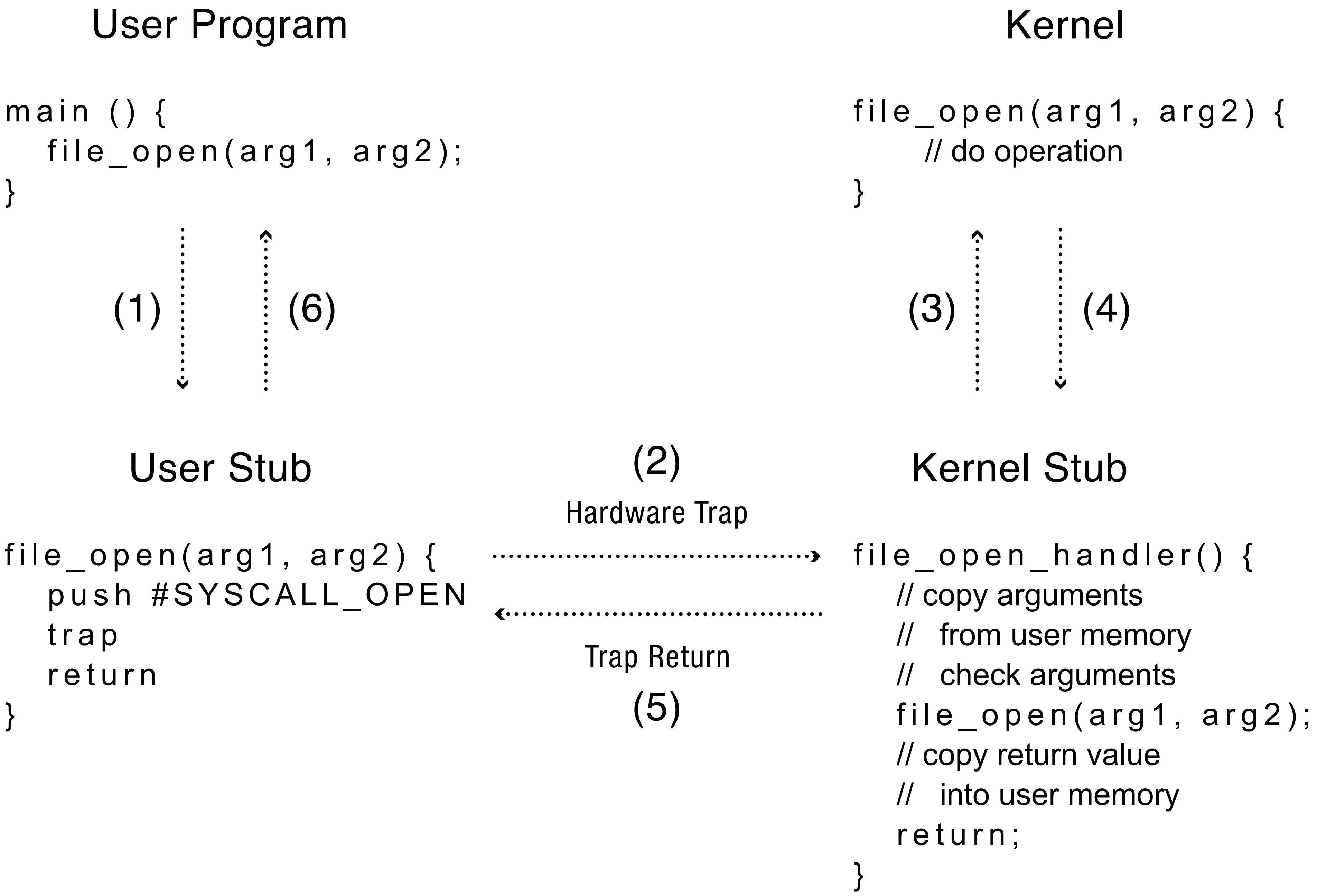
Signal Stack

Saved Registers

SP

PC


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



User Program

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

(1)

(6)

Kernel

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

(3)

(4)

User Stub

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

(2)

Hardware Trap

Trap Return

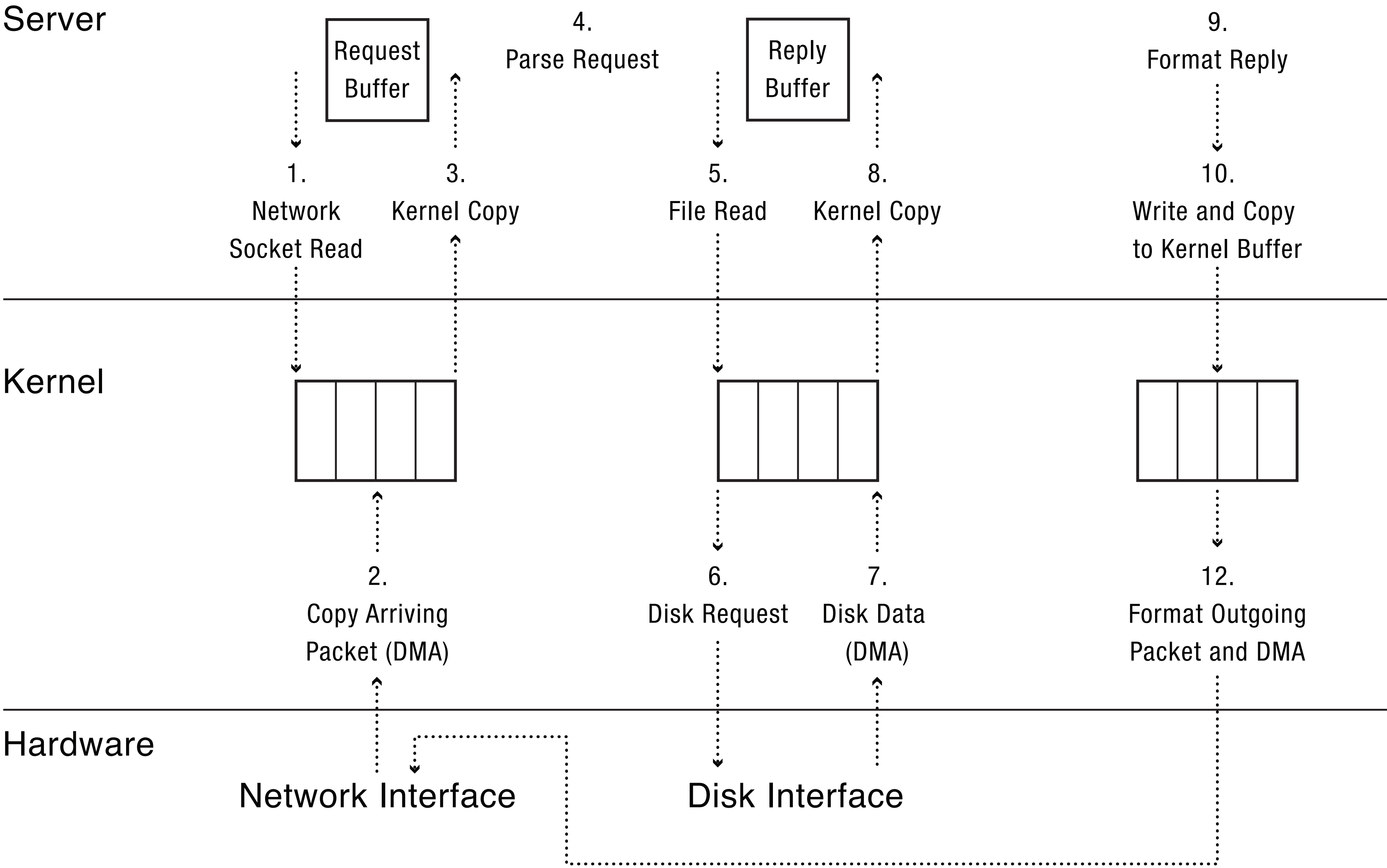
(5)

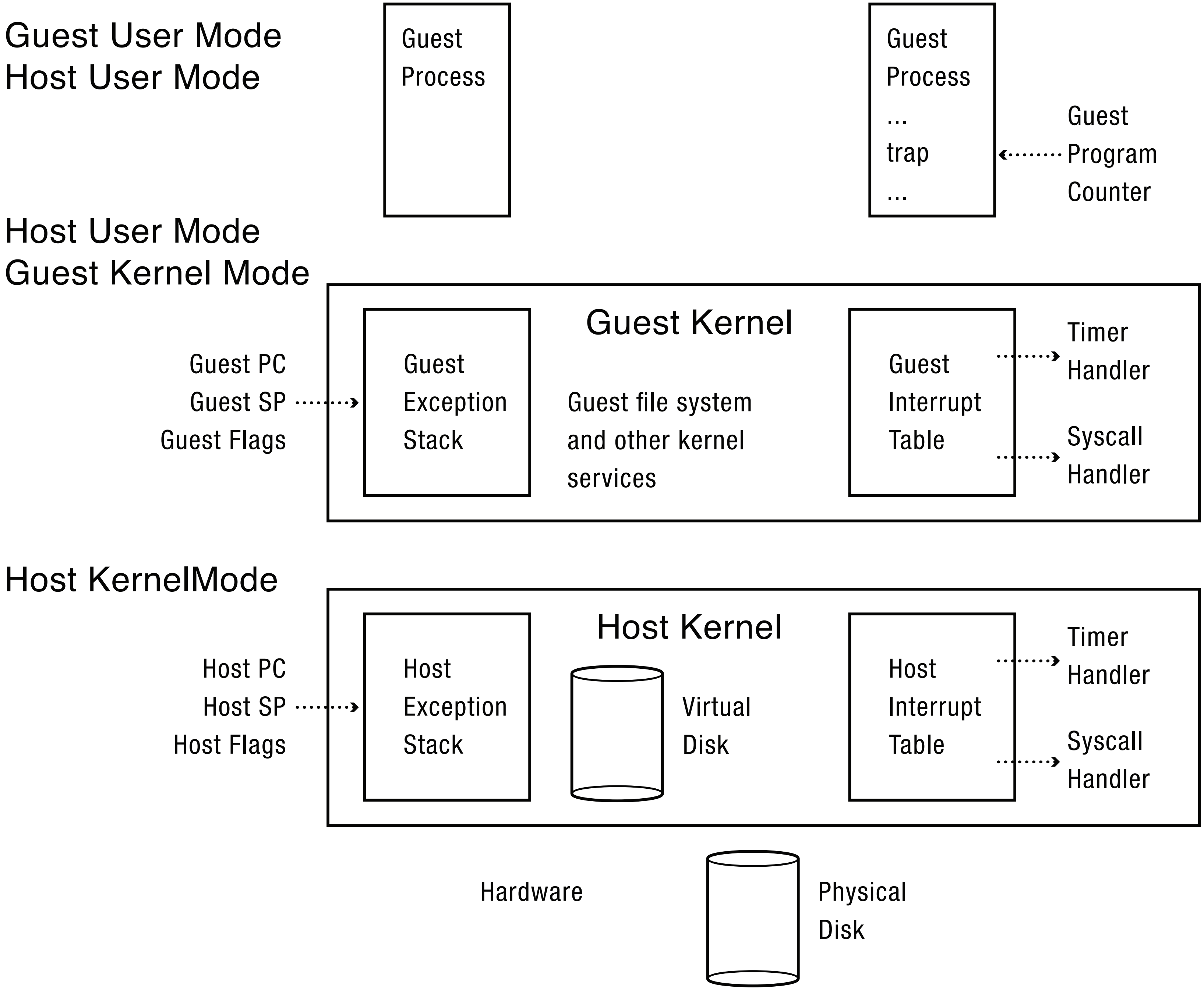
Kernel Stub

```
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


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



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