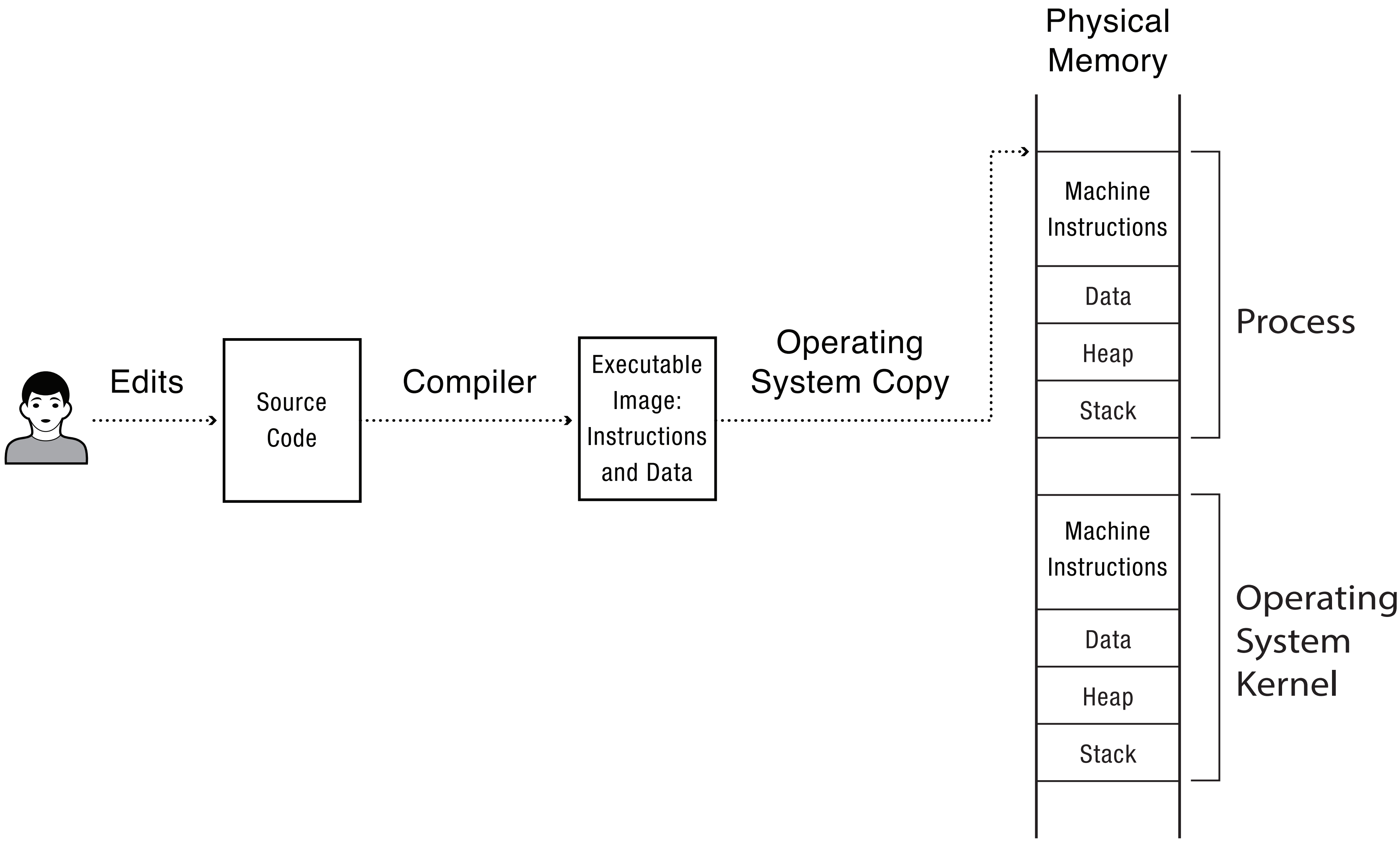
The Kernel Abstraction

The 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

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

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)

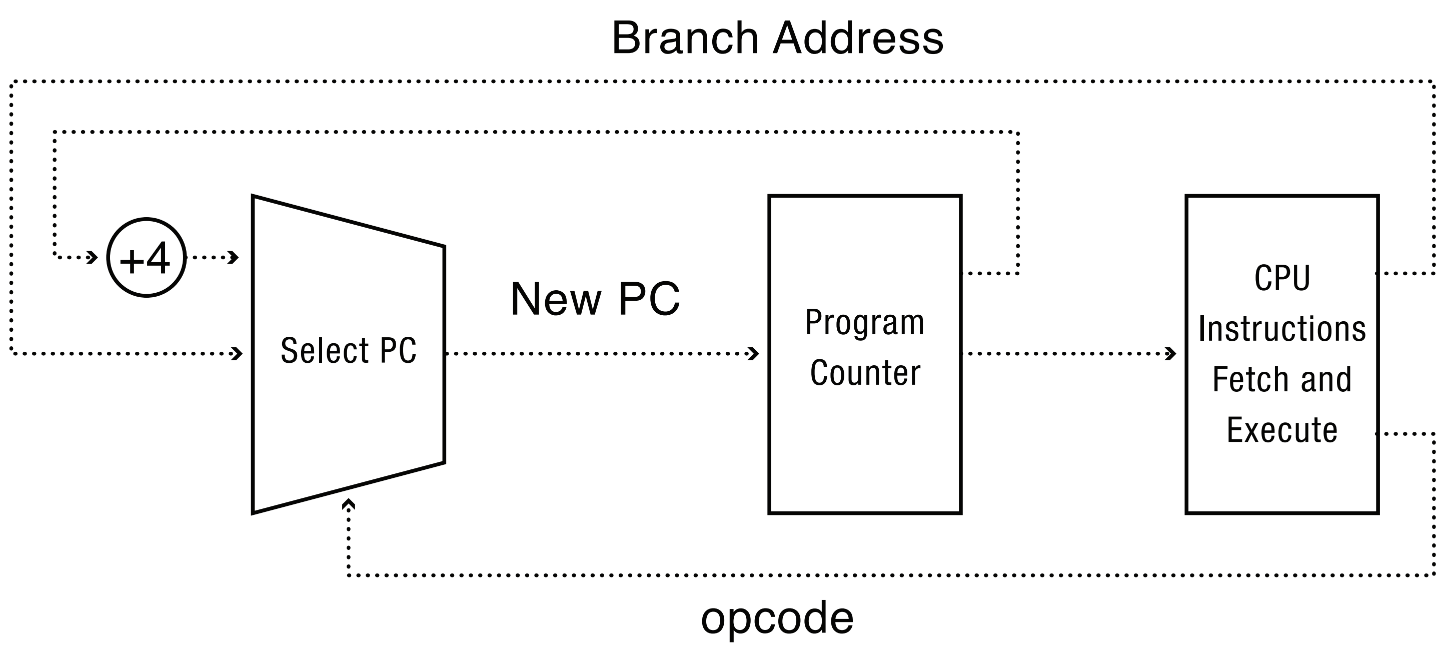
Thought Experiment

- 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, …
- How do we go faster?
  – Run the unprivileged code directly on the CPU?

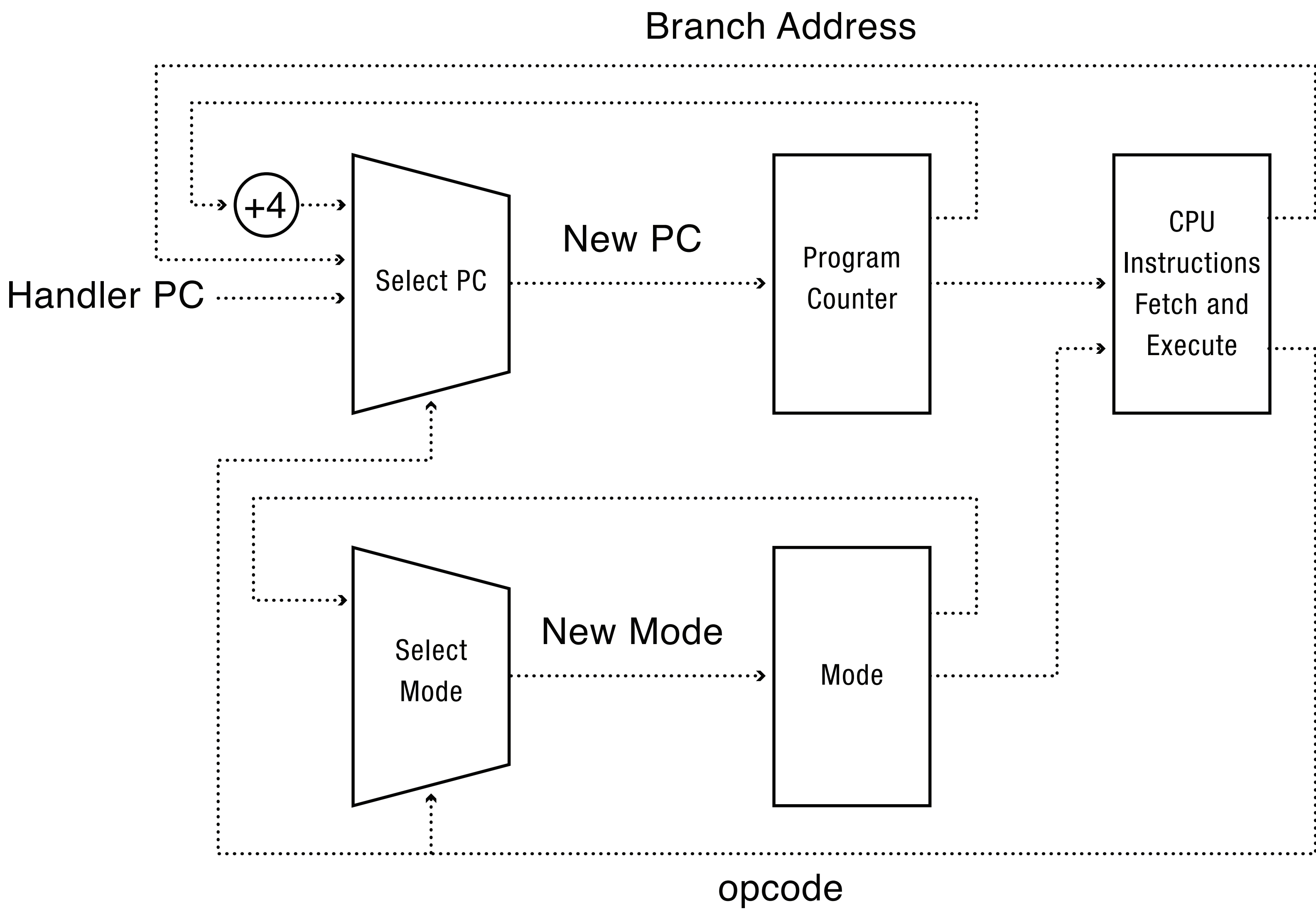
Hardware Support: Dual-Mode Operation

- 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

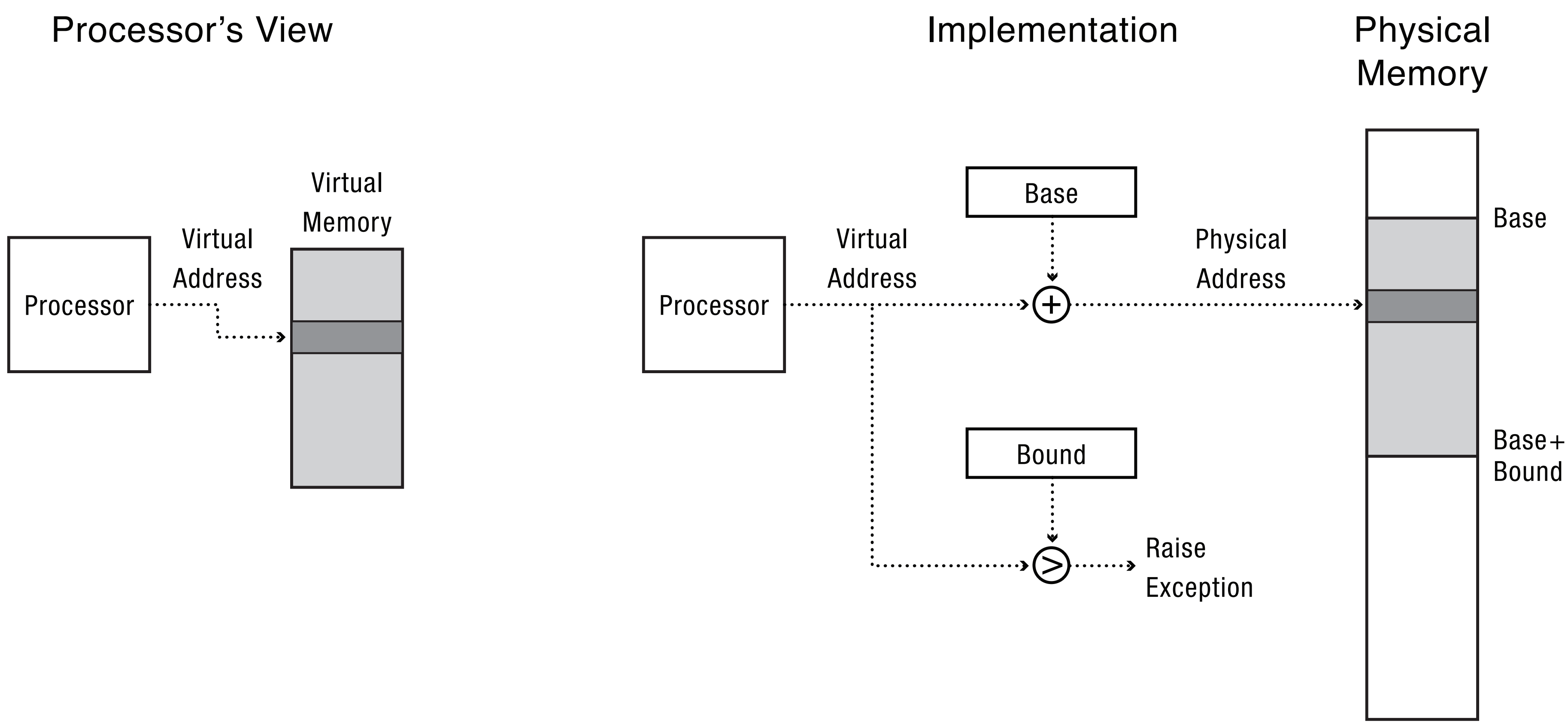
- Examples?

- 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

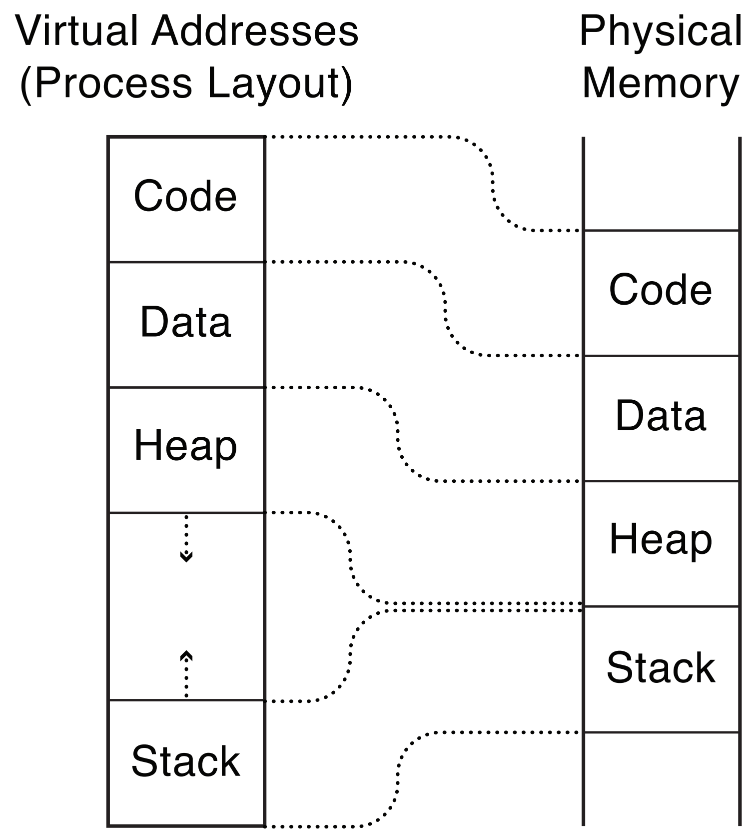


Towards Virtual Addresses

- 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() {
   int localVar = 0;   // a procedure local variable

   staticVar += 1; localVar += 1;

   sleep(10);  // sleep causes the program to wait for x seconds
   printf ("static address: %x, value: %d\n", &staticVar, staticVar);
   printf ("procedure local address: %x, value: %d\n", &localVar, localVar);
}
```

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

Question

- Suppose we had a perfect object-oriented language and compiler, so that only an object's methods could 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?

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

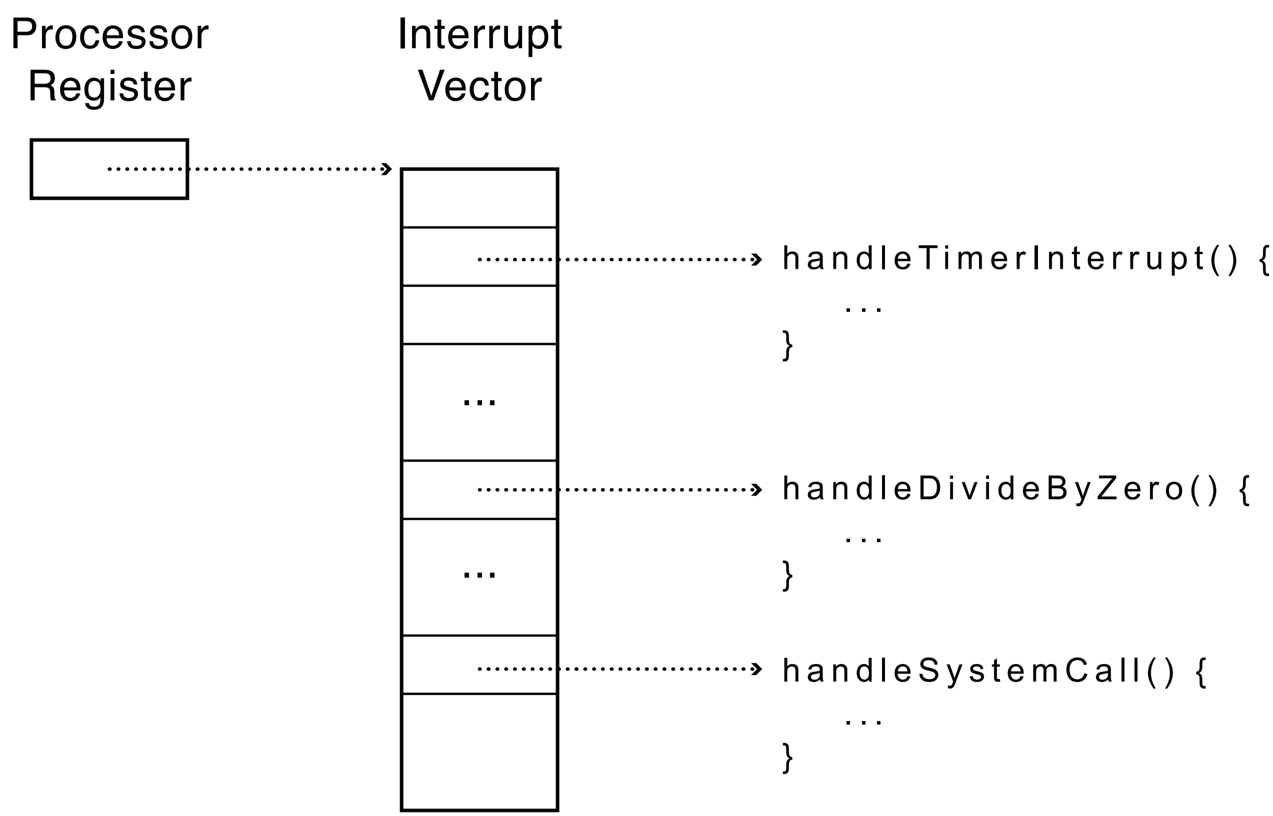
- From kernel mode to user mode
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall (UNIX signal)
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

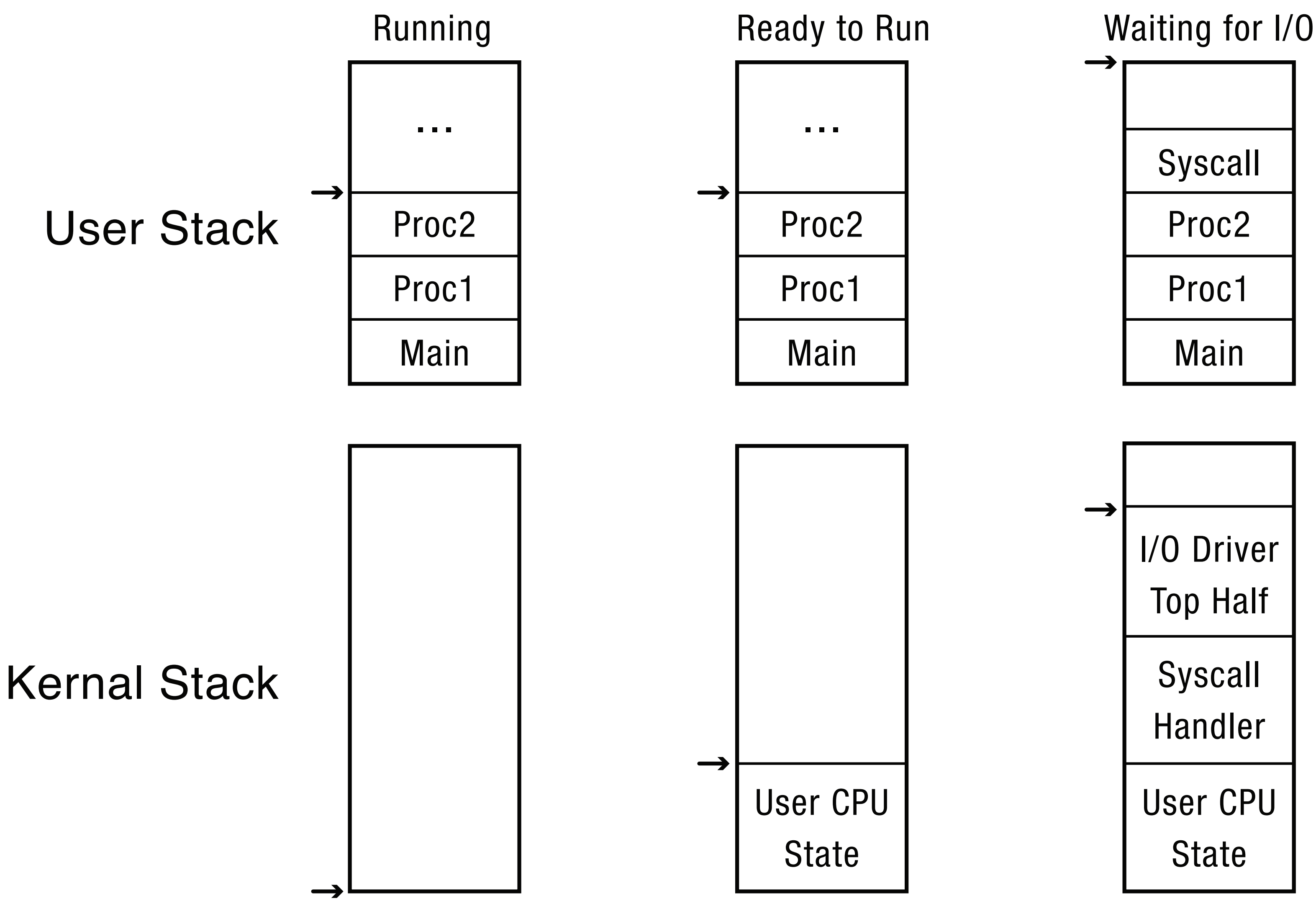
- Table set up by OS kernel; pointers to code to run on different events



Interrupt Stack

- Per-processor, located in kernel (not user) memory
  - Usually a process/thread has both: kernel and user stack
- Why can't 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)
- Cf. implementing synchronization, chapter 5

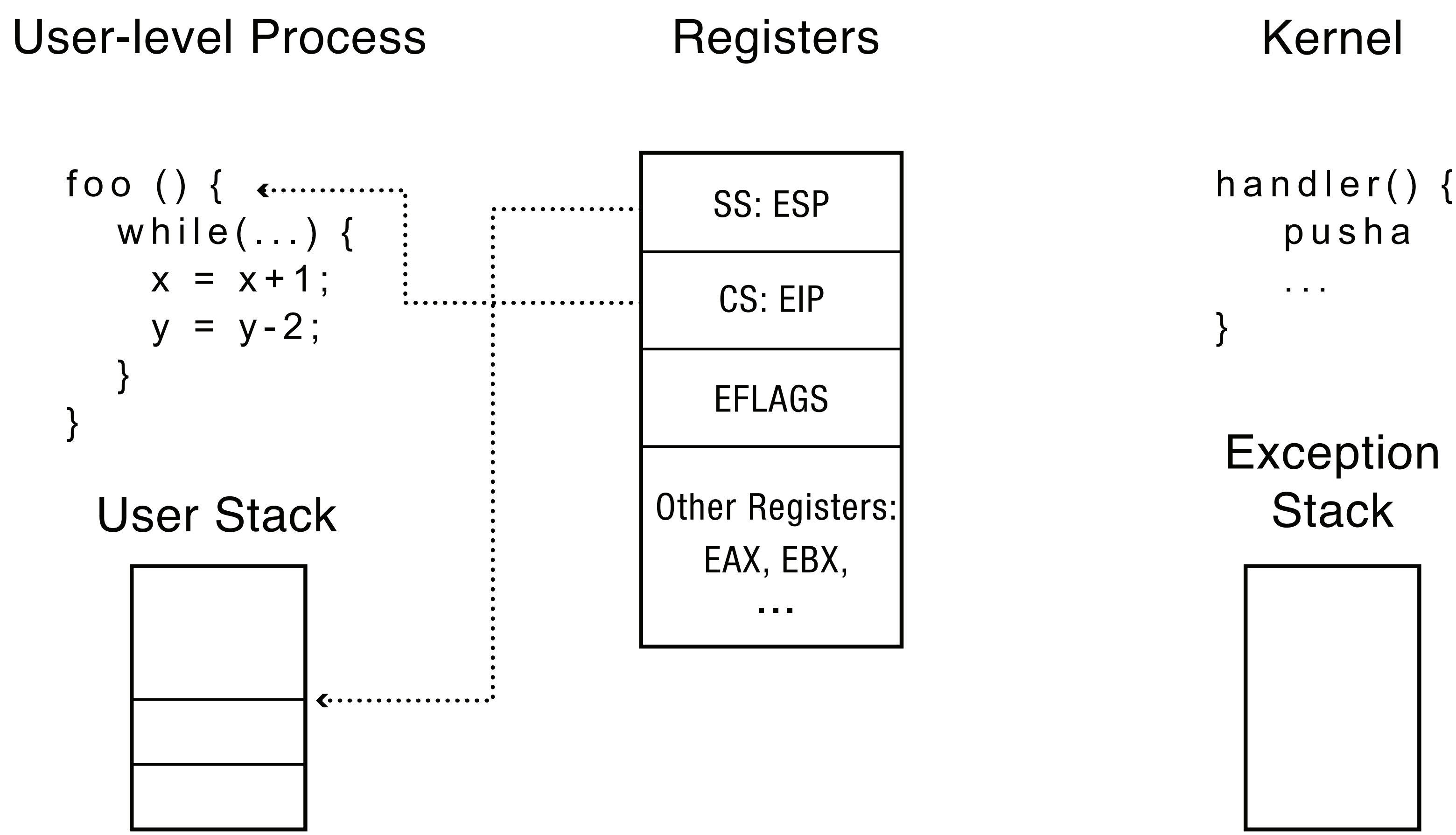
Interrupt Handlers

- 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

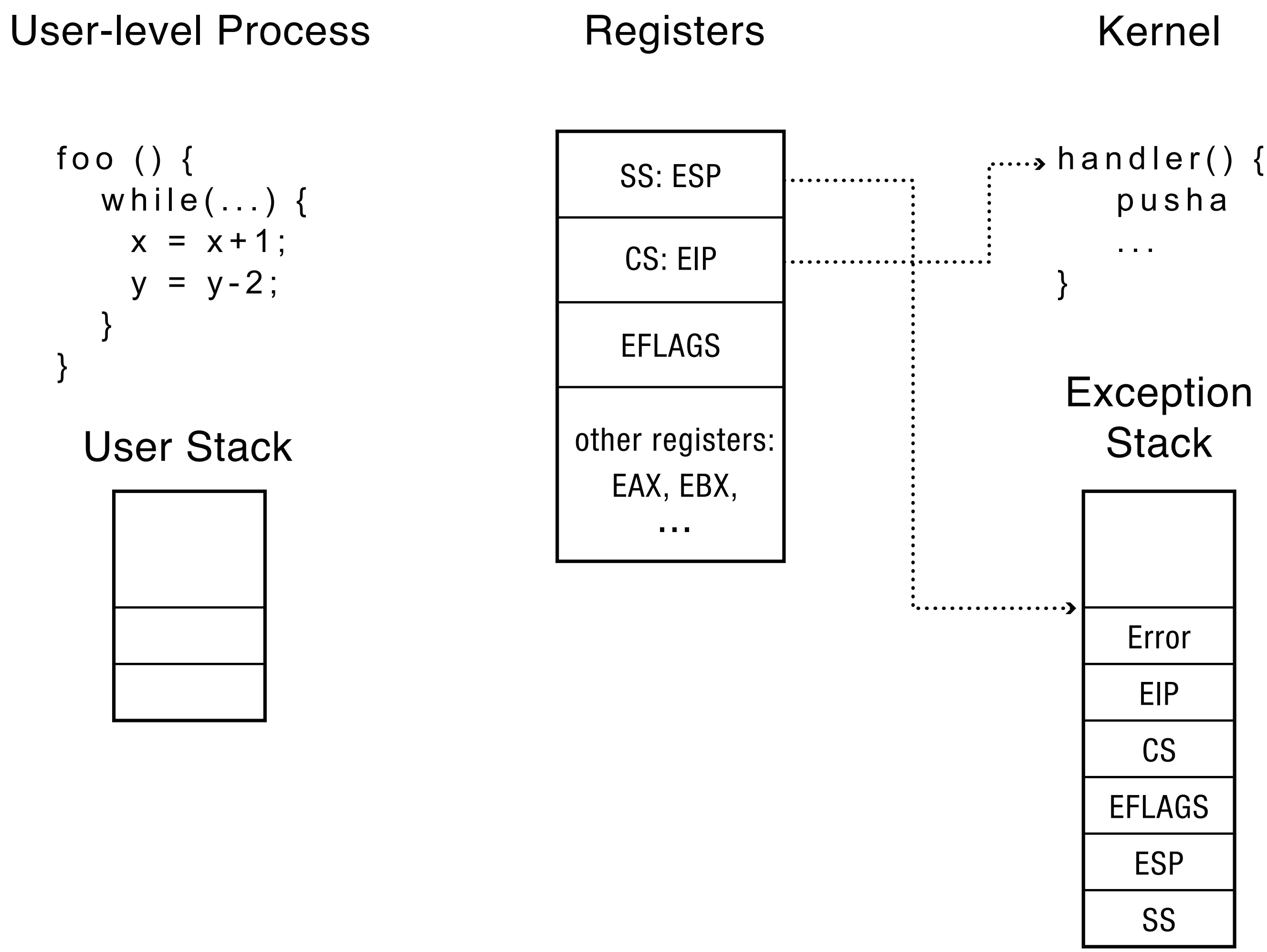
Atomic Mode Transfer

- On interrupt (x86)
  - 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

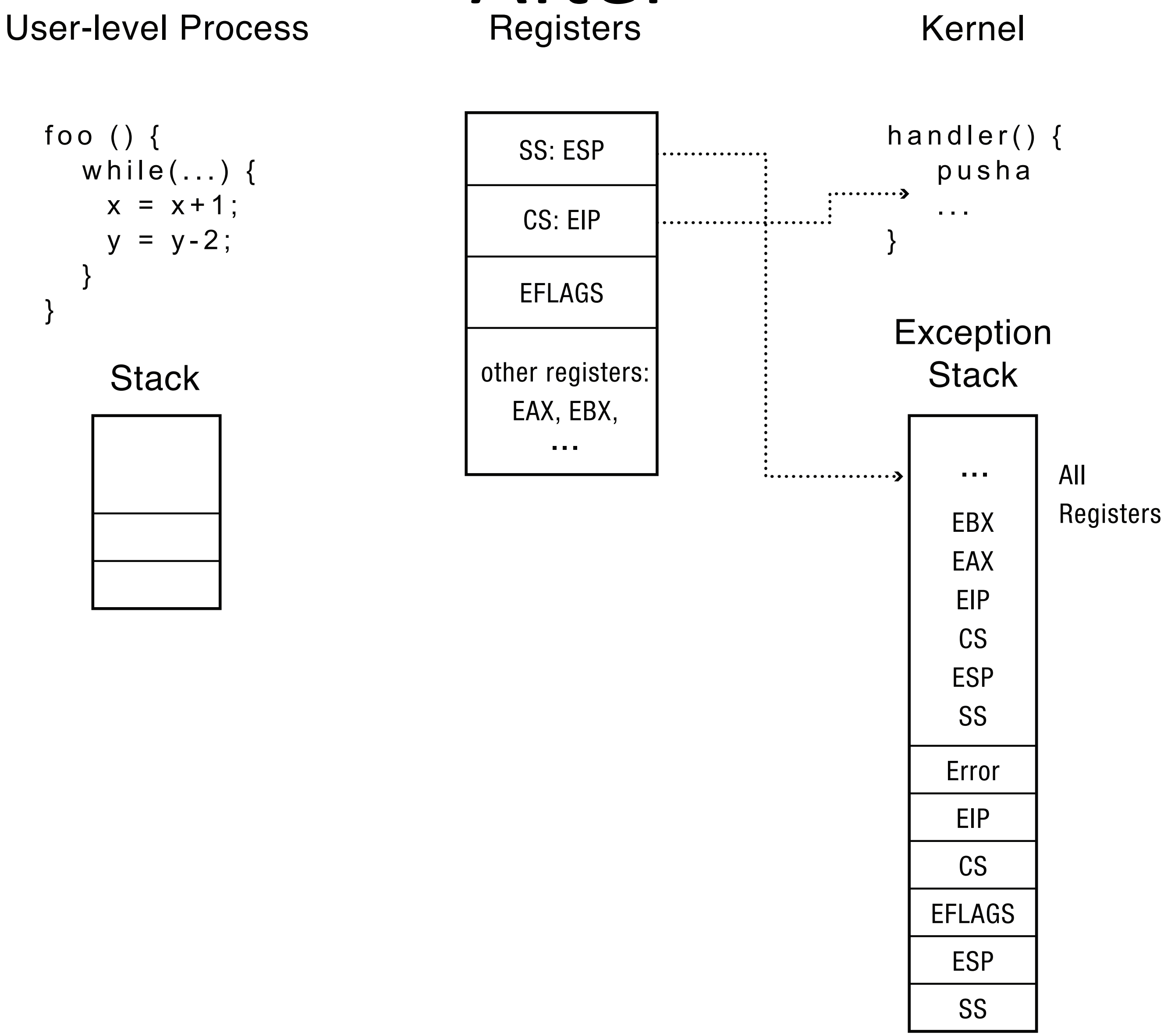
Before


User-level Process
Registers
Kernel
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
SS: ESP
CS: EIP
EFLAGS
Other Registers:
EAX, EBX,
...
handler() {
pusha
...
}
User Stack
Exception Stack

During


User-level Process
Registers
Kernel
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
User Stack
SS: ESP
CS: EIP
EFLAGS
other registers:
EAX, EBX,
...
handler() {
pusha
...
}
Exception
Stack
Error
EIP
CS
EFLAGS
ESP
SS

After


User-level Process
Registers
Kernel
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
Stack
SS: ESP
CS: EIP
EFLAGS
other registers:
EAX, EBX,
...
handler() {
pusha
...
}
Exception Stack
...
EBX
EAX
EIP
CS
ESP
SS
All Registers
Error
EIP
CS
EFLAGS
ESP
SS

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

System Calls

User Program

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

(1) (6)

User Stub

```
syscall (arg1, arg2) {
  trap
  return
}
```

(2)

Hardware Trap

Trap Return

(5)

Kernel

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

(3) (4)

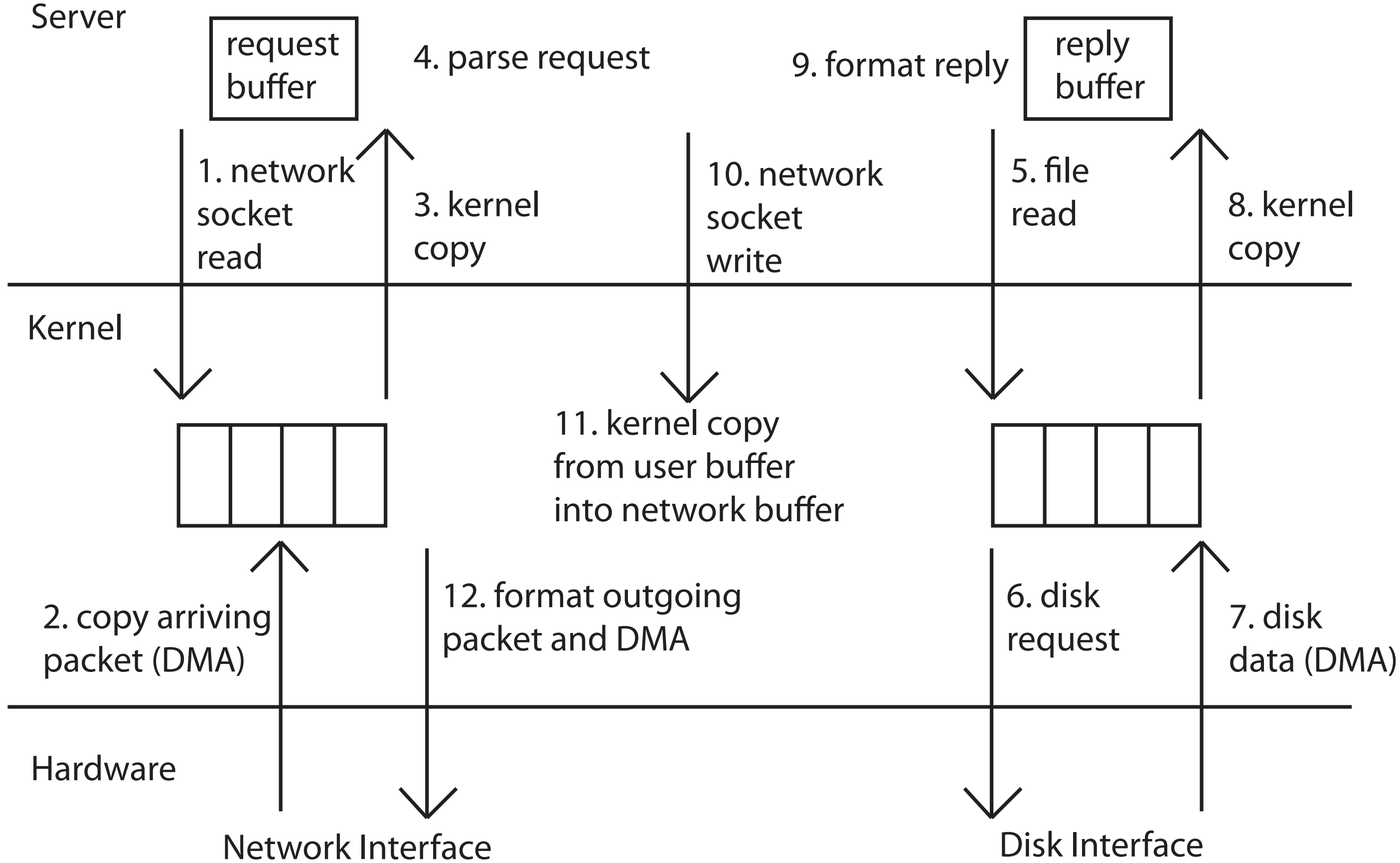
Kernel Stub

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

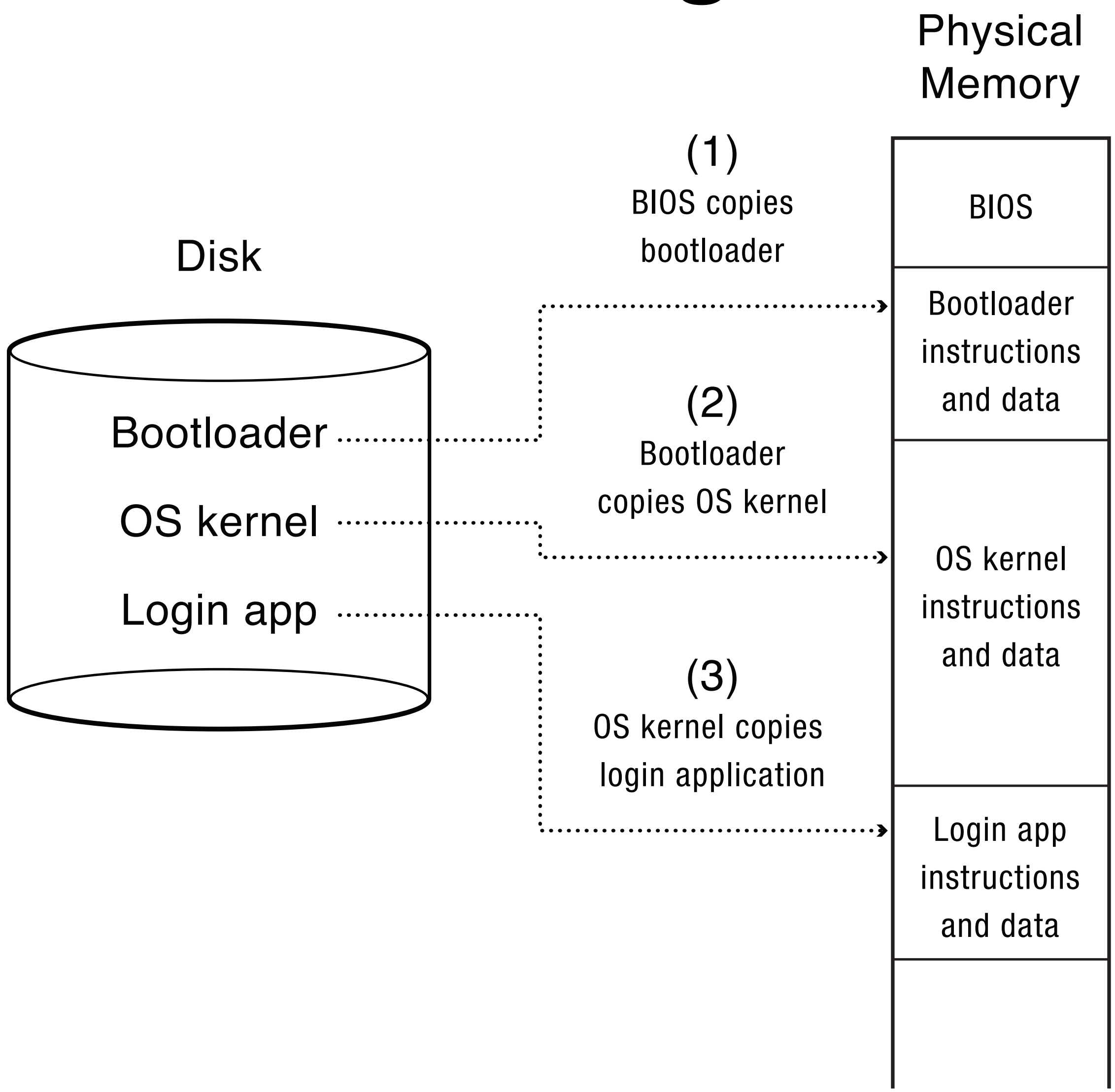
Kernel System Call Handler

- Locate arguments
  - In registers or on user(!) stack
- 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

Web Server Example

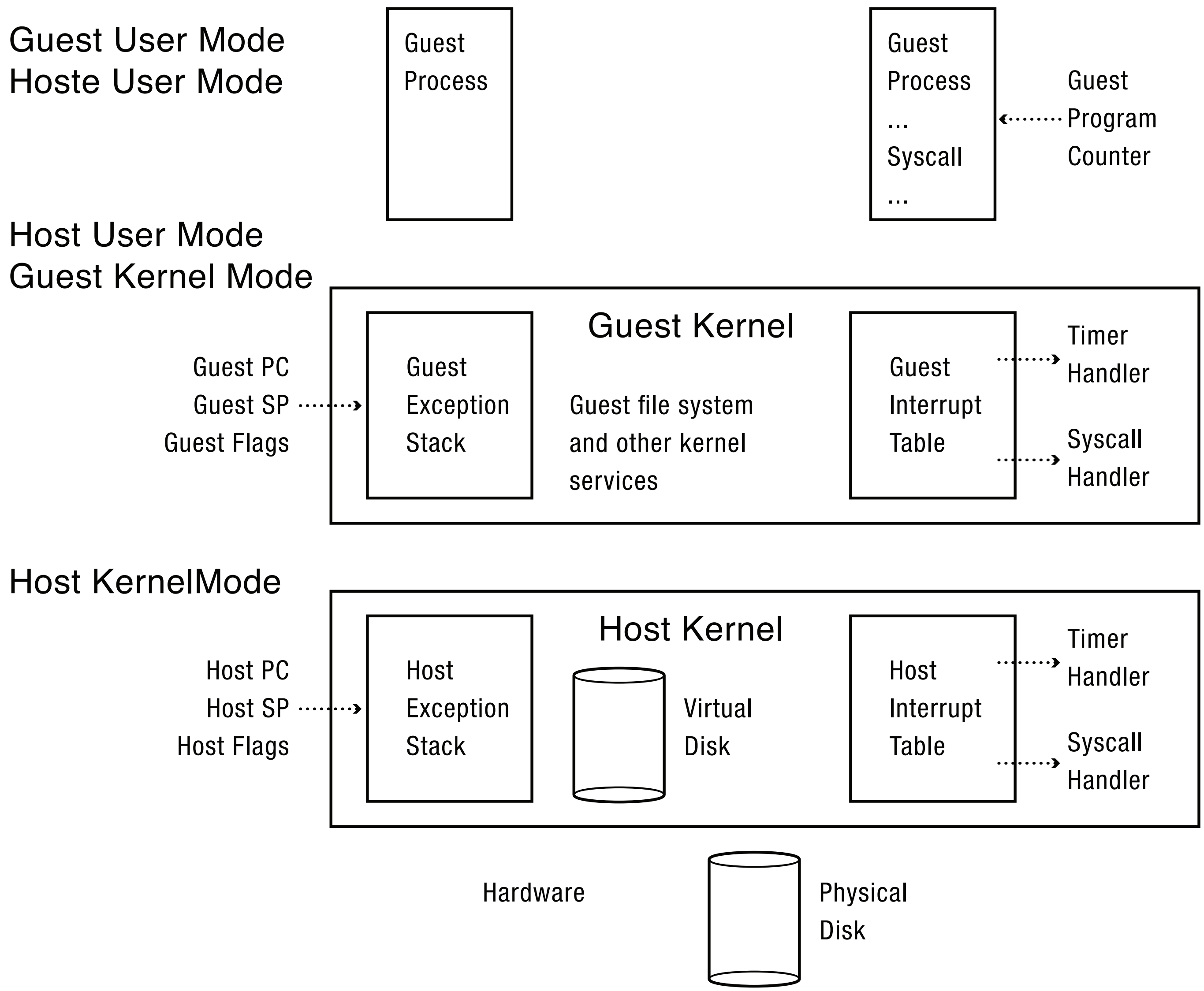


Booting



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

Virtual Machine



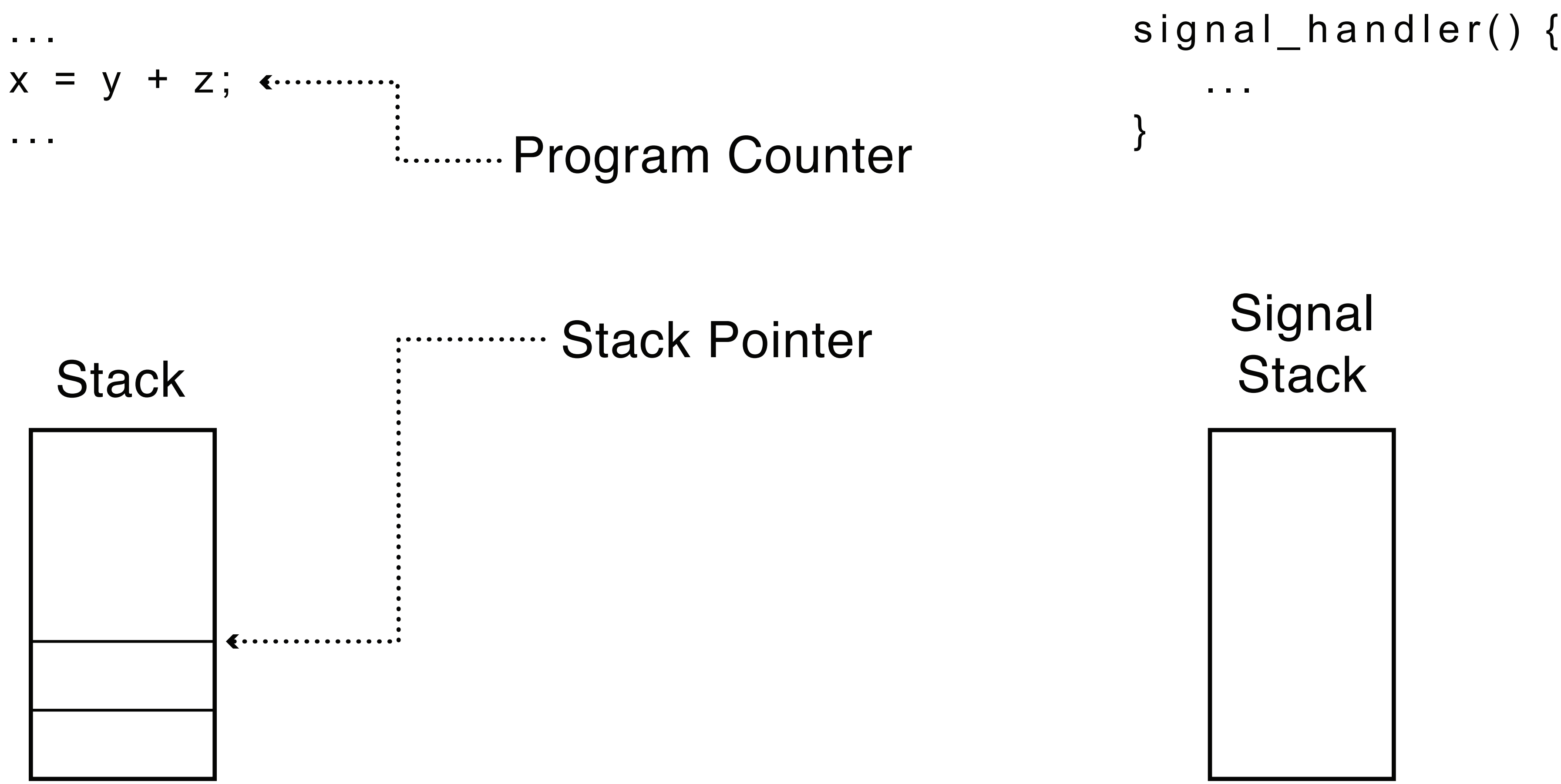
User-Level Virtual Machine

- 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

Upcall: User-level interrupt

- AKA UNIX signal
  - Notify user process of event that needs to be handled right away
    - Time-slice for user-level thread manager
    - Interrupt delivery for VM player
- Direct analogue of kernel interrupts
  - Signal handlers – fixed entry points
  - Separate signal stack
  - Automatic save/restore registers – transparent resume
  - Signal masking: signals disabled while in signal handler

Upcall: Before



Upcall: After