The Kernel Abstraction

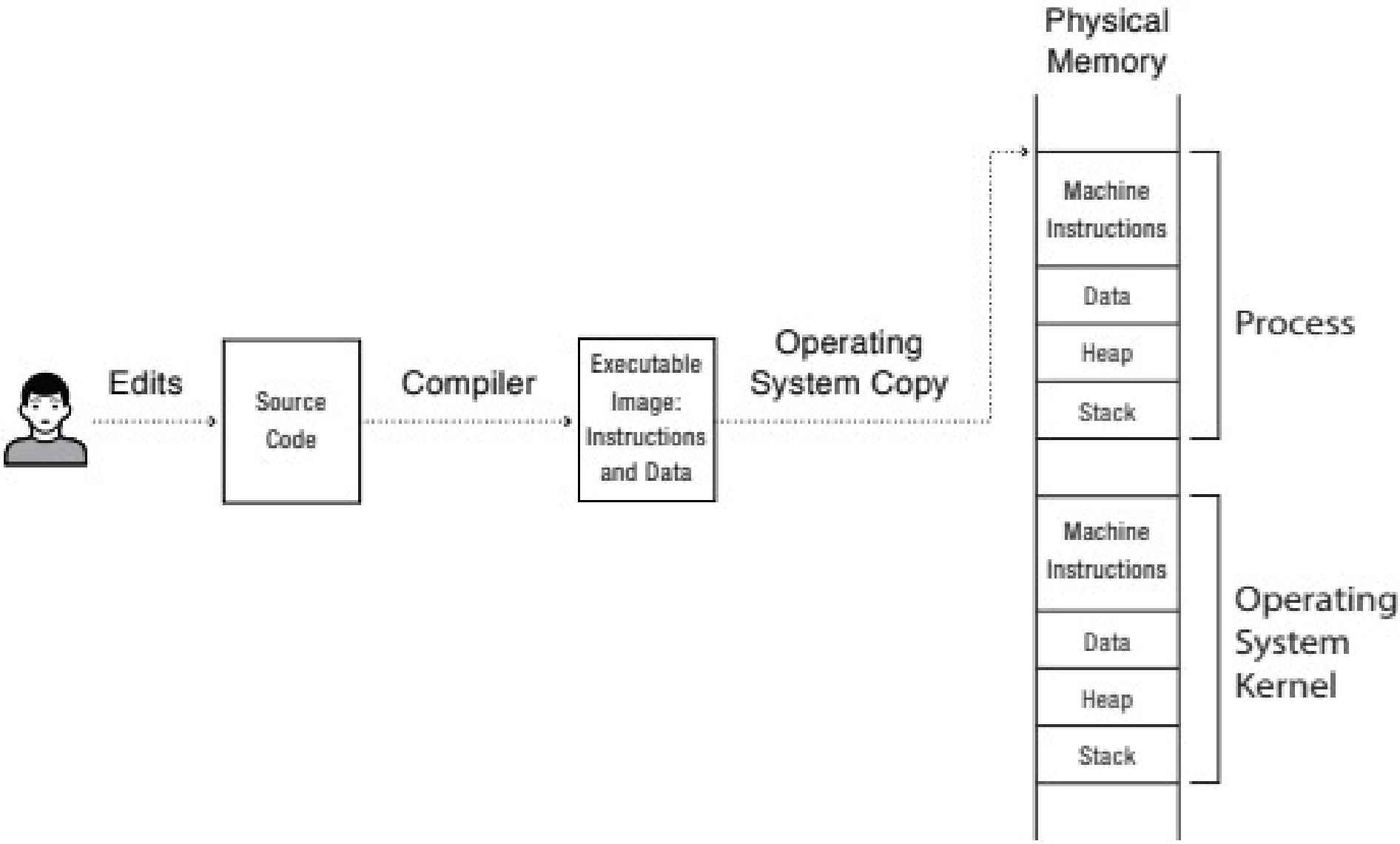
Main Points

- 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
- Types of Mode Transfer
  - What causes the processor to switch control from a user-level program to the kernel
- 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
  - Process vs Program
  - Process Control Block (PCB)
  - 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)

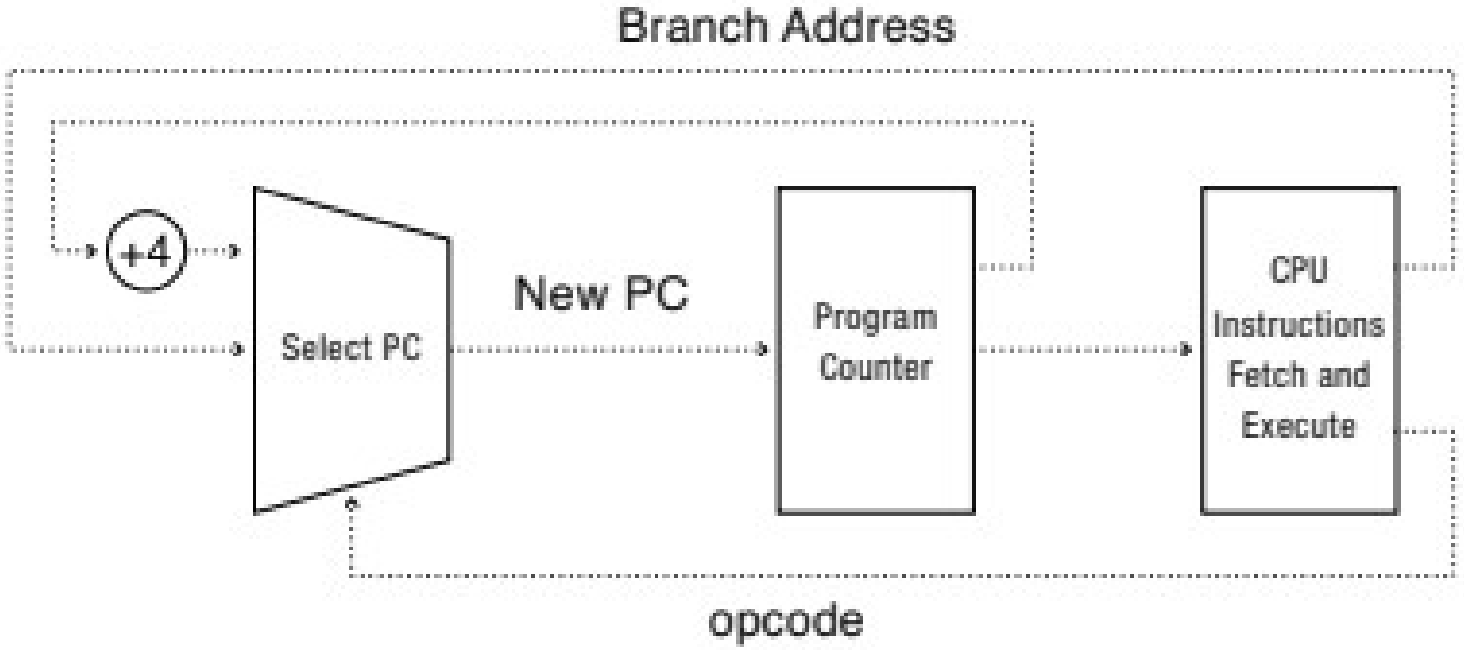
Process Creation



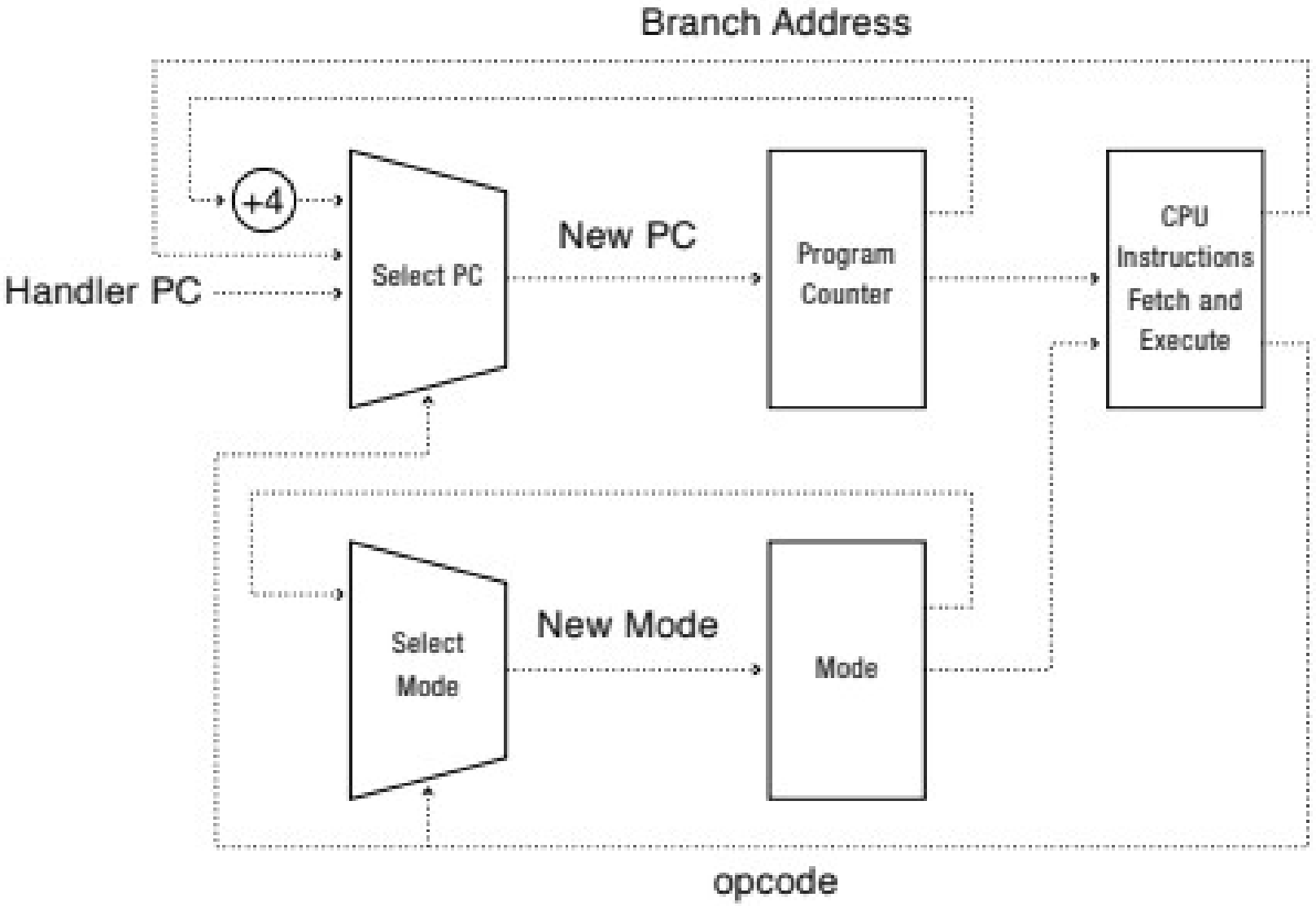
Hardware Support: Dual-Mode Operation

- 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
  - The interpreter checks if a process has permission to perform each instruction

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

- Should programs directly change their privilege level?
  - No! for process isolation
  - Possible indirectly through *system calls*
    - But can transfer control into the OS kernel
- Privileged Instructions
  - Run by OS kernel
    - Change privilege levels
    - Adjust memory access
    - Disable/enable interrupts

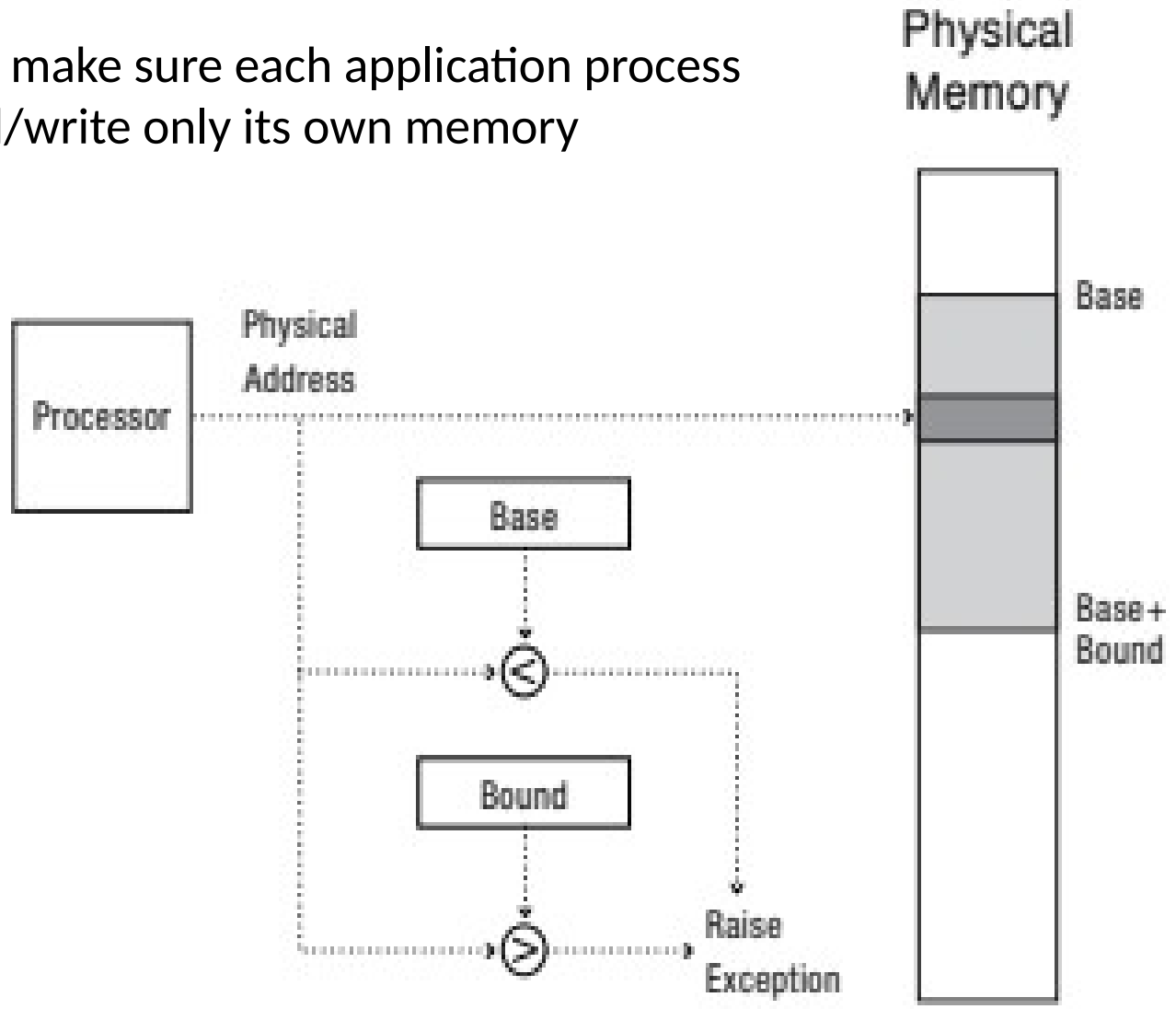
Privileged instructions

- 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 could happen if an application attempts to access restricted memory or change its privilege?
  - Processor exception->kernel exception handler

- What could happen if applications were allowed to jump into kernel mode at any location in the kernel?
  - Kernel verifies whether the user has the permission

Simple Memory Protection

OS must make sure each application process can read/write only its own memory



Towards Virtual Addresses

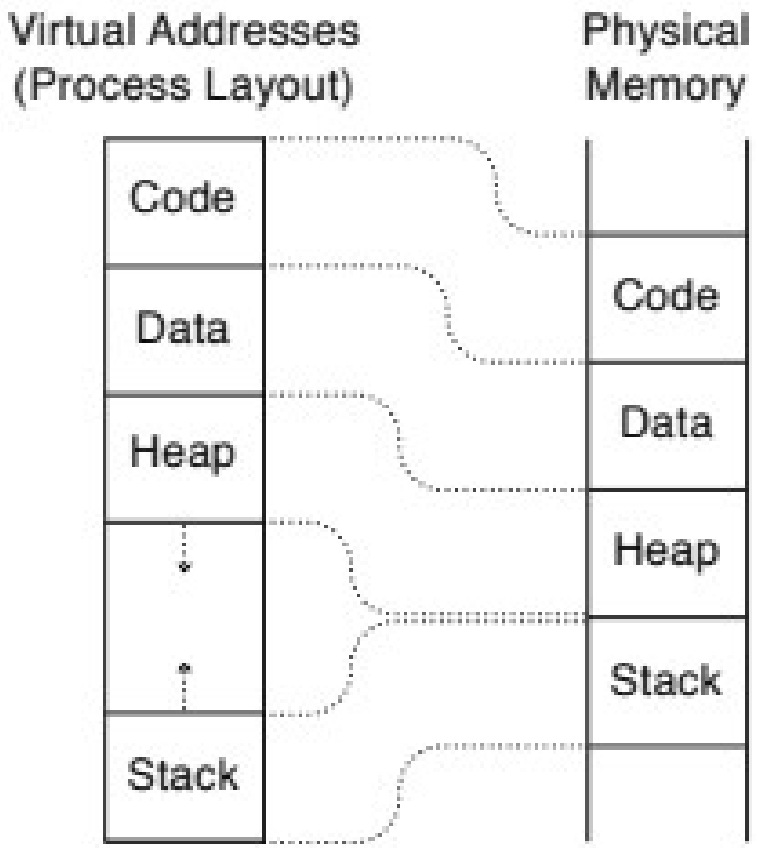
- Problems with base and bounds?
  - Expandable heap?
  - Expandable stack?
  - Memory sharing between processes?
    - Multiple processes running the same program or using the same library
  - Physical memory addresses
    - Must change every instruction and data location each time the program is loaded into memory
  - Memory fragmentation
    - Difficult to relocate programs once started

Virtual Addresses

- All process memory starts at the same place, e.g., zero
  - Virtual addresses never change
- Each process thinks that it has the entire machine -> Transparent!
- Actual physical memory locations determined at run-time

Virtual Addresses

- Translation
  hardware, u
  table
- Table set up
  operating sy
  kernel
- Stack and he
  start from
  separate ends



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?

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?
  - In theory, no
  - In practice, yes
  - Google Chrome example

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

  How does the kernel know if an application is in an infinite loop?

Mode Switch

- From user mode to kernel mode
  - Interrupts
    - Asynchronous signal to the processor that some external event has occurred
    - Saves the current execution state before executing at the interrupt handler at the kernel
    - Interrupt handers are also specially designated
      - Timer handler: Infinite loop alert
      - I/O handler
      - Interrupt vector table
      - Inter-processor interrupts
    - Triggered by timer and I/O devices
    - Polling: kernel loops and checks events

Mode Switch

- From user mode to kernel mode
  - Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
    - Divide by 0, access of non-existed or read-only memory
    - OS halts process and return error coce
  - 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
    - Transparent: user applications do not know mode switch

Mode Switch

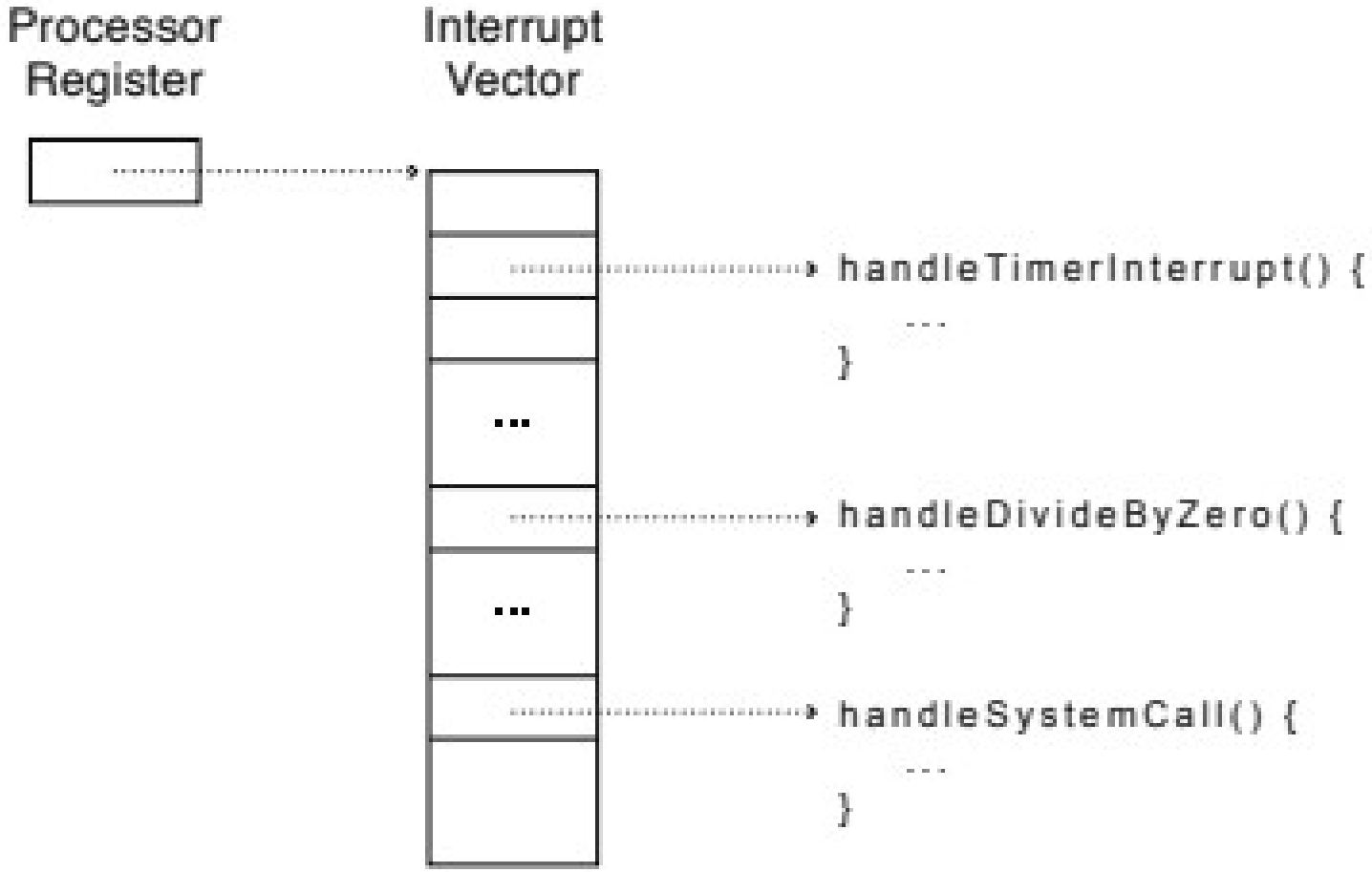
- From kernel mode to user mode
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall (UNIX signal)
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



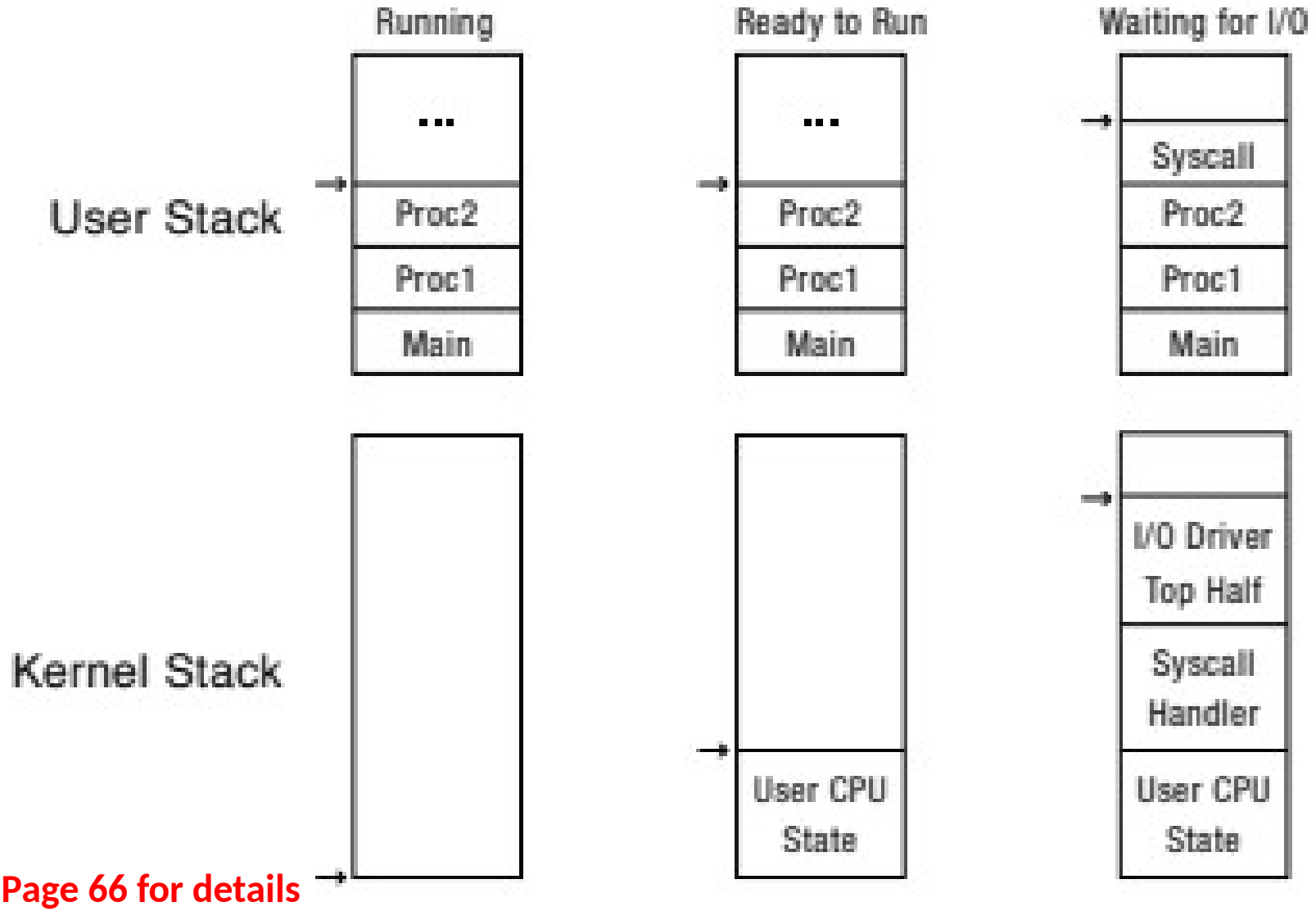
Interrupt Stack

- When context switch happens, the hardware changes its stack pointer to the base of the kernel's interrupt stack and saves interrupted process's registers by pushing them into the *interrupt stack*.
  - Reverse operation when returning from the interrupt

- 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



Page 66 for details

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

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

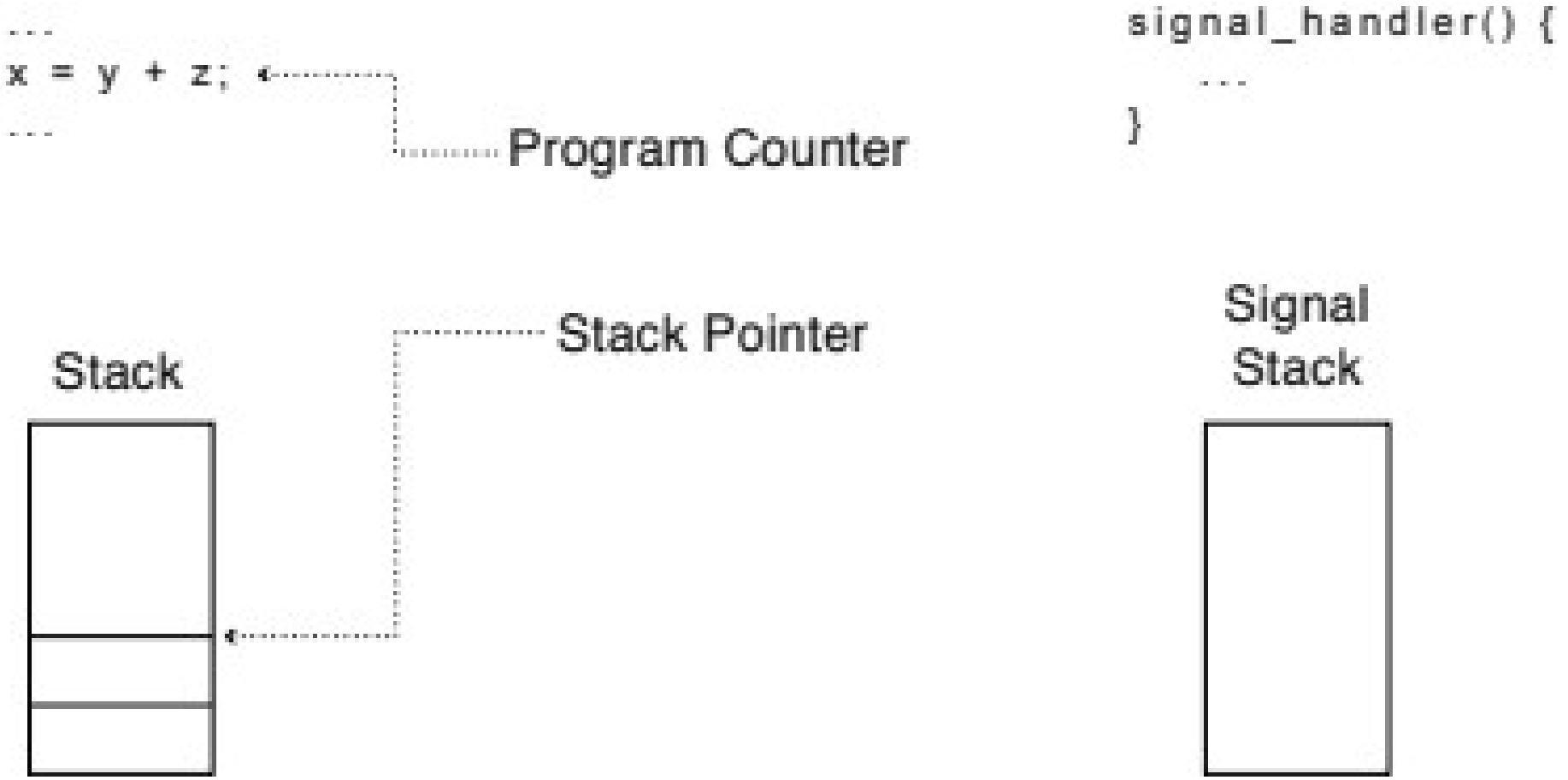
Upcall: User-level event delivery

- 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

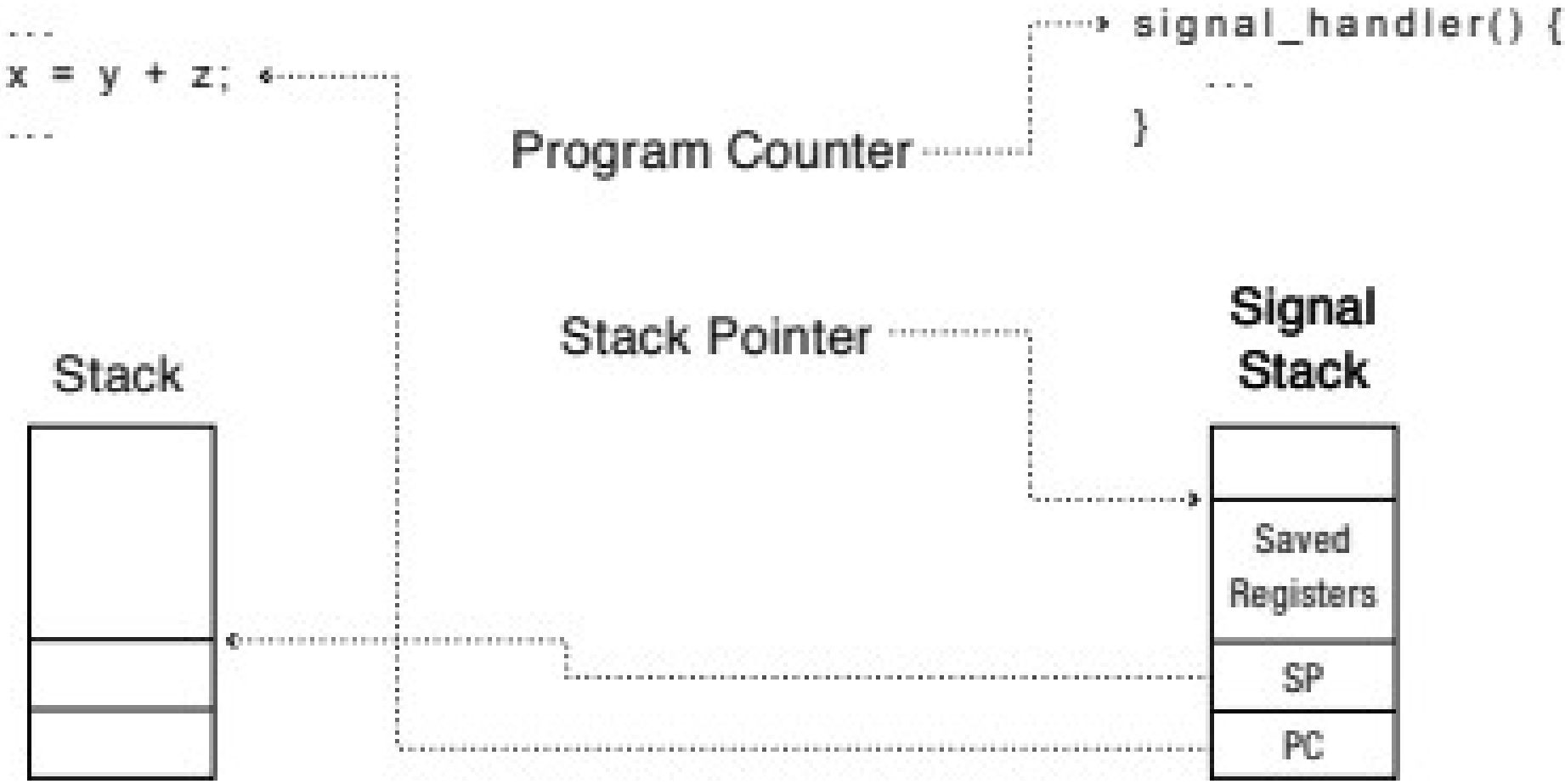
Upcalls vs Interrupts

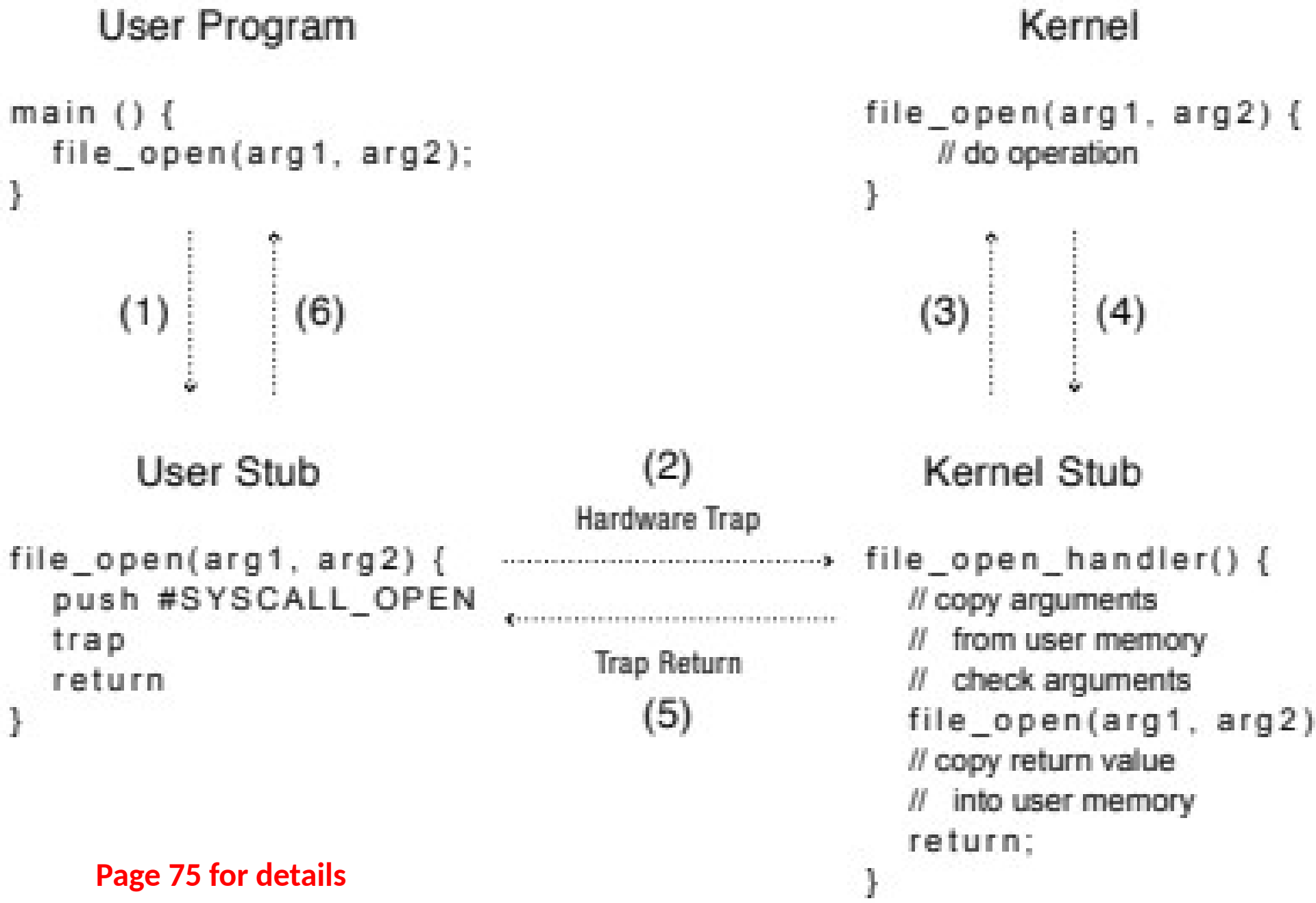
- Signal handlers = interrupt vector
- Signal stack = interrupt stack
- Automatic save/restore registers = transparent resume
- Signal masking: signals disabled while in signal handler

Upcall: Before

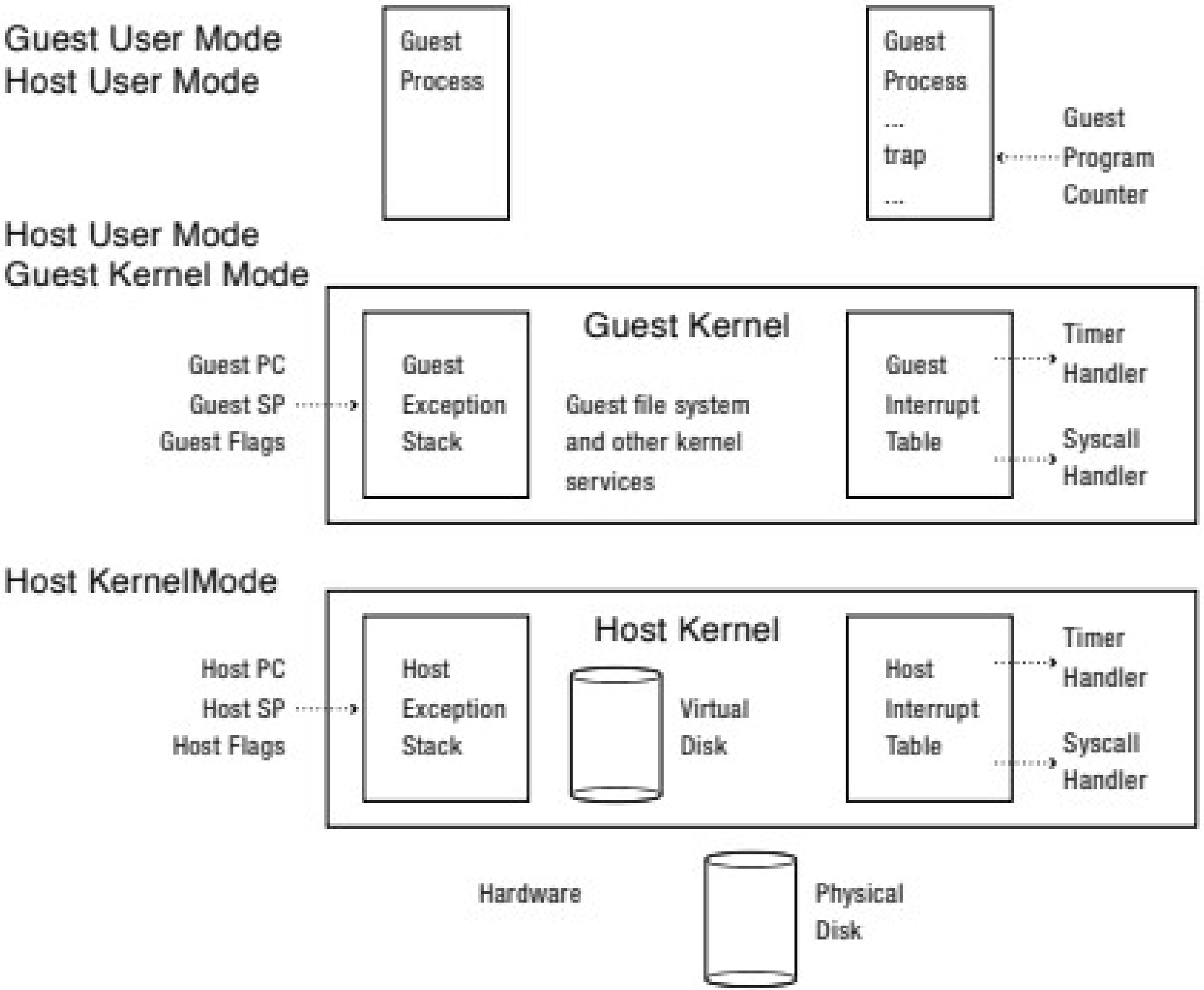


Upcall: During





Page 75 for details


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