The Kernel Abstraction

***Throughout the course we will use overheads that were adapted from those distributed from the textbook website. Slides are from the book authors, modified and selected by Jean Mayo, Shuai Wang and C-K Shene.**

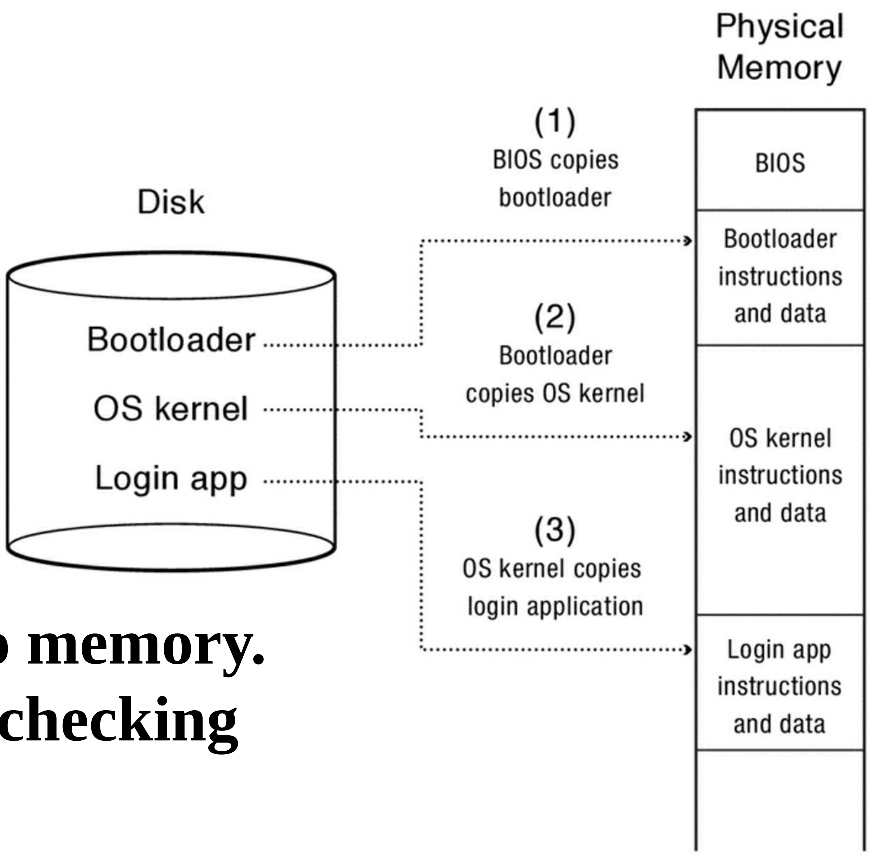
I don't know what the programming language of the year 2000 will look like, but I know it will be called FORTRAN.

Charles Anthony Richard Hoare

Spring 2019

Booting / Initial Program Loader

- When a computer is powered up, the booting procedure starts.
- A small record on the boot device is read (by the BIOS).
- This small record may load more records from the boot device, making a larger "program".
- Then, this program executes, maybe loading more modules into memory.
- This program may clear memory, checking for available devices, etc.
- Moreover, the kernel is loaded into memory. When the kernel starts running, interrupts are enabled.
- ***Remember***: OS is an interrupt-driven program.



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

Dual-Mode Operation

- **Modern CPUs have two execution modes: the *user* mode and the *supervisor* (or system, kernel, privileged) mode, controlled by a mode bit.**
- **The OS runs in the supervisor mode and all user programs run in the user mode.**
- **Some instructions that may do harm to the OS (e.g., I/O and CPU mode change) are *privileged instructions*. Privileged instructions, for most cases, can only be used in the supervisor model.**
- **When execution switches to the OS (resp., a user program), execution mode is changed to the supervisor (resp., user) mode.**

Hardware Support: Dual-Mode Operation 1/2

- ***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 the `EFLAGS` register.
- On the MIPS, mode in the status register.

Hardware Support: Dual-Mode Operation 2/2

- ❑ 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

Mode Switch (User -> Kernel)

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

Mode Switch (Kernel -> User)

- ***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

Thought Experiment

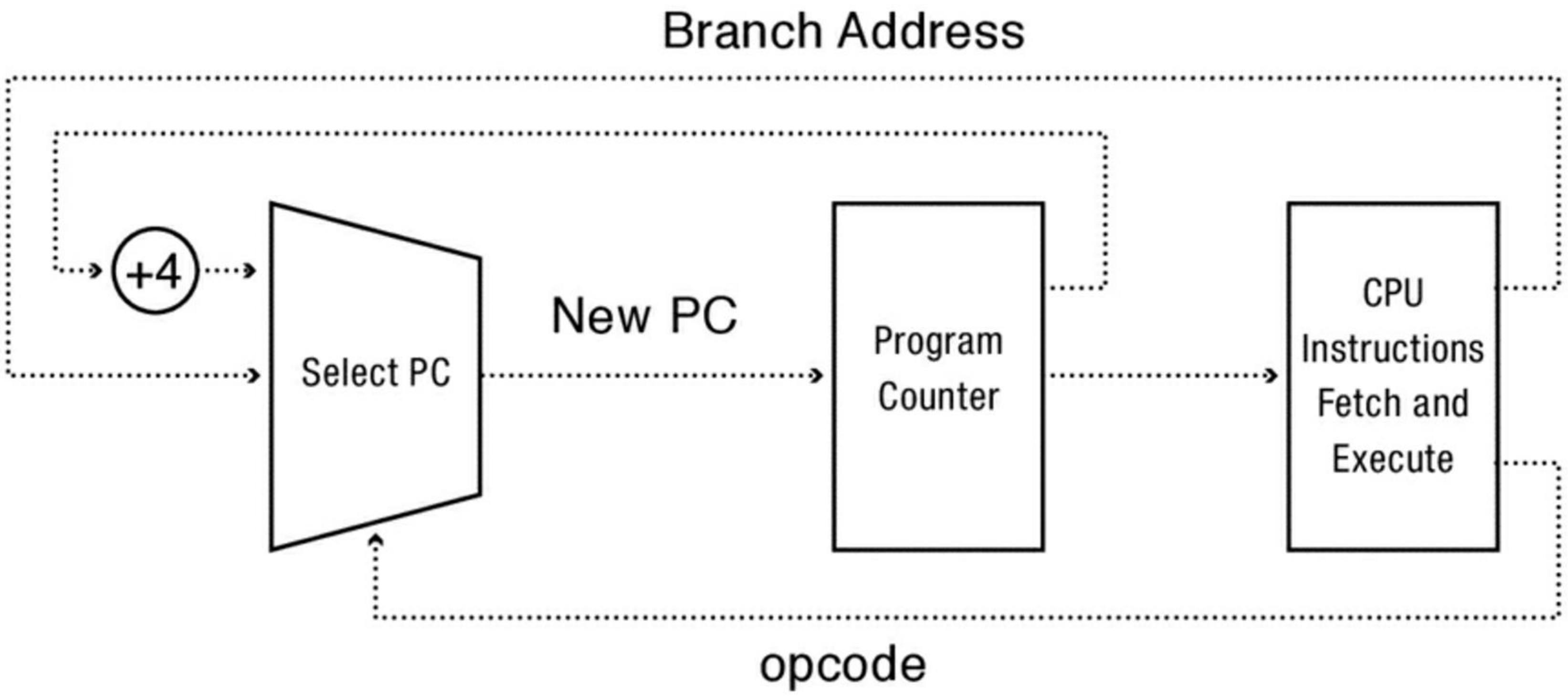
- How can we implement execution with limited privilege?
  - Execute each program instruction in a simulator
  - If the instruction is permitted, do the instruction
  - Otherwise, stop the process
  - Basic model in Javascript and other interpreted languages
- How do we go faster?
  - Run the unprivileged code directly on the CPU!

Privileged Instructions

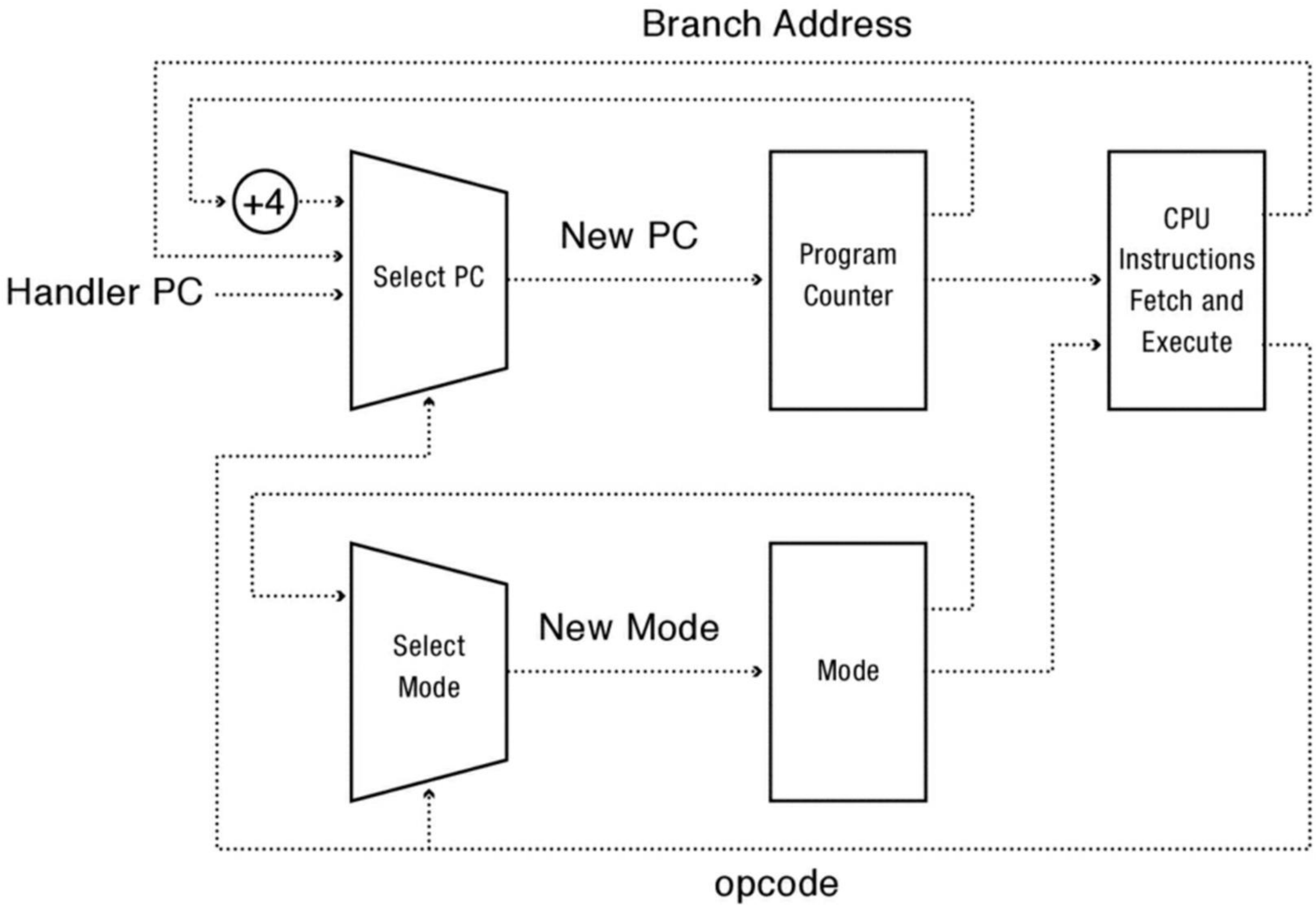
- ***Examples?***

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

A Model of a CPU



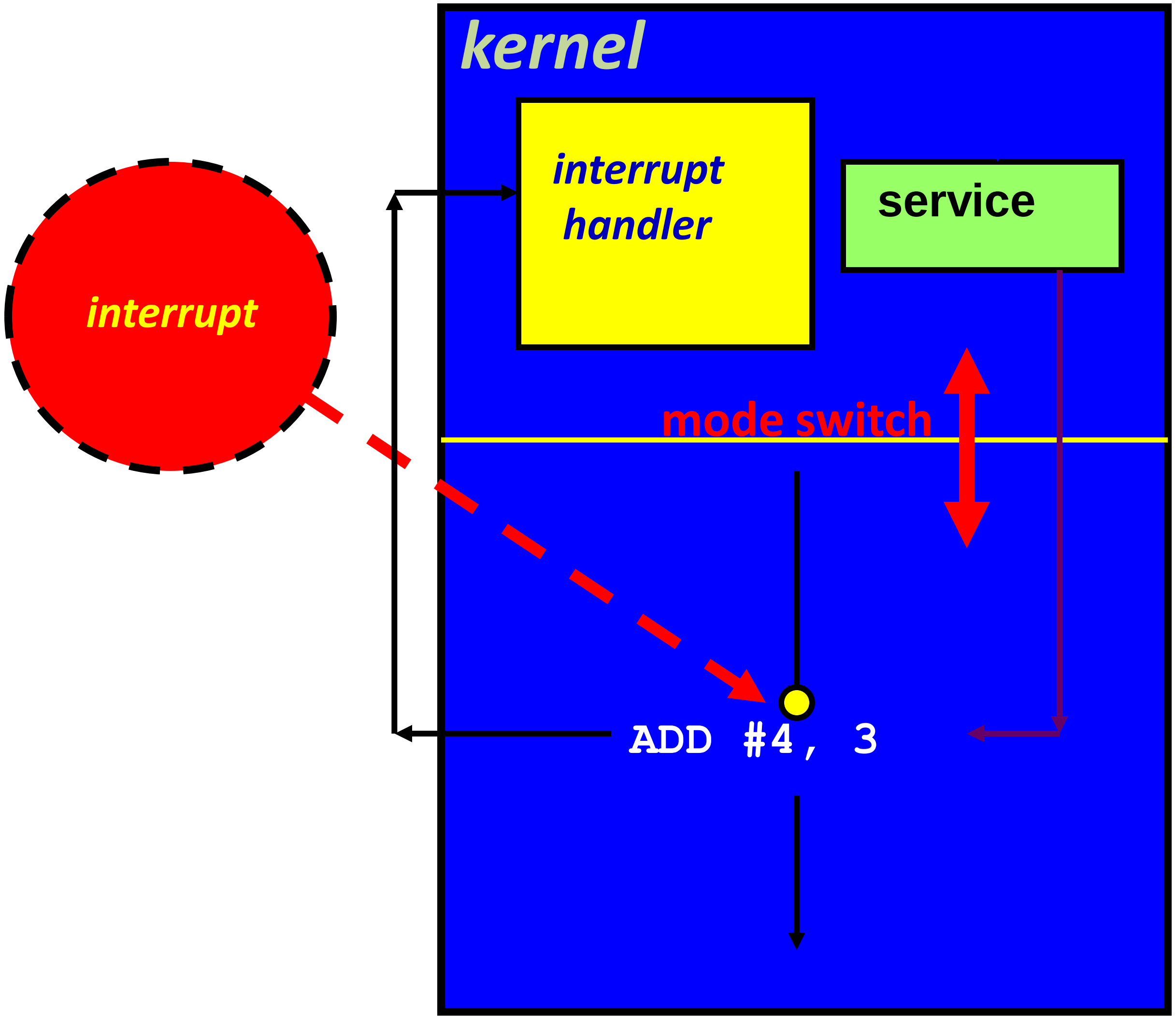
A CPU with Dual-Mode Operation



Interrupt and Trap

- **An event that requires the attention of the OS is an interrupt. These events include the completion of an I/O, a keypress, a request for service, a division by zero and so on.**
- **Interrupts may be generated by hardware or software.**
- **An interrupt generated by software (*i.e.*, division by 0) is referred to as a *trap* or an *exception*.**
- **Modern operating systems are *interrupt driven*, meaning the OS is in action only if an interrupt occurs.**

What Is Interrupt-Driven?

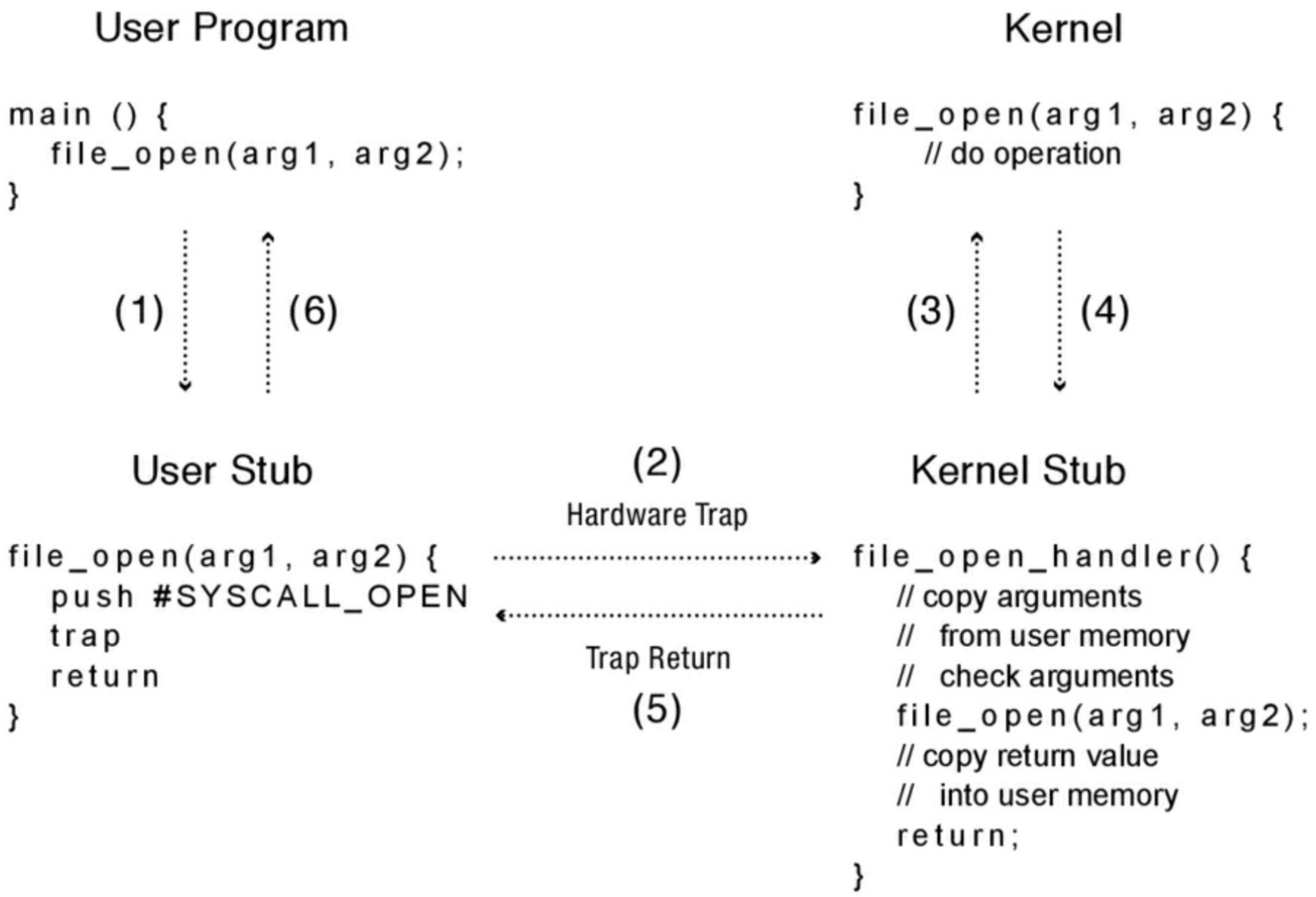


- The OS is activated by an interrupt.
- The executing program is suspended.
- Control is transferred to the OS.
- A program will be resumed when the service completes.

System Calls

- System calls provide an interface to the services made available by an operating system.
- A system call generates an interrupt (actually a trap), and the caller is suspended.
- Type of system calls:
  - Process control (e.g., create and destroy processes)
  - File management (e.g., open and close files)
  - Device management (e.g., read and write operations)
  - Information maintenance (e.g., get time or date)
  - Communication (e.g., send and receive messages)

Sequence of Steps Involved in a System Call



User Program

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

Kernel

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

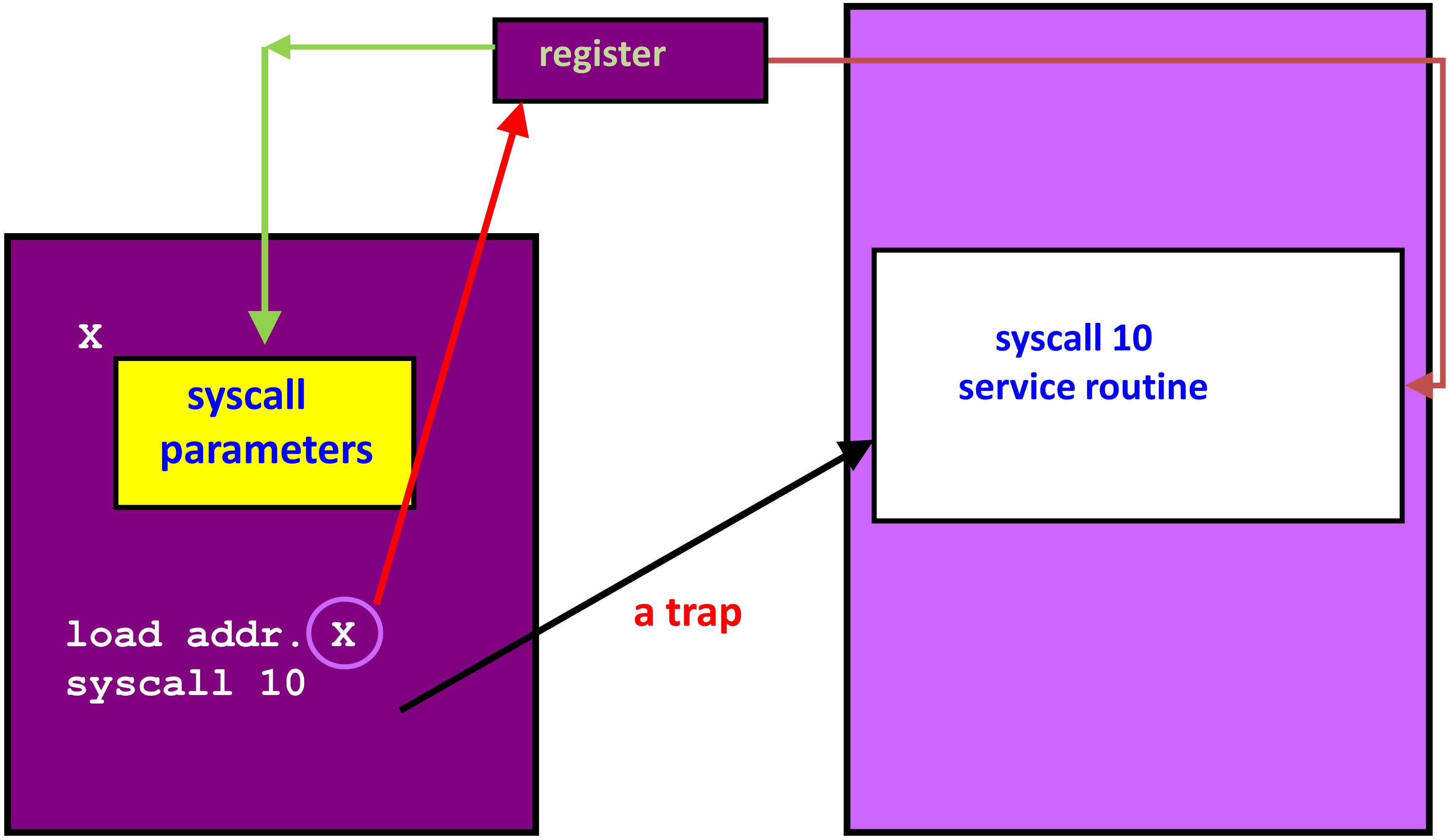
User Stub

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

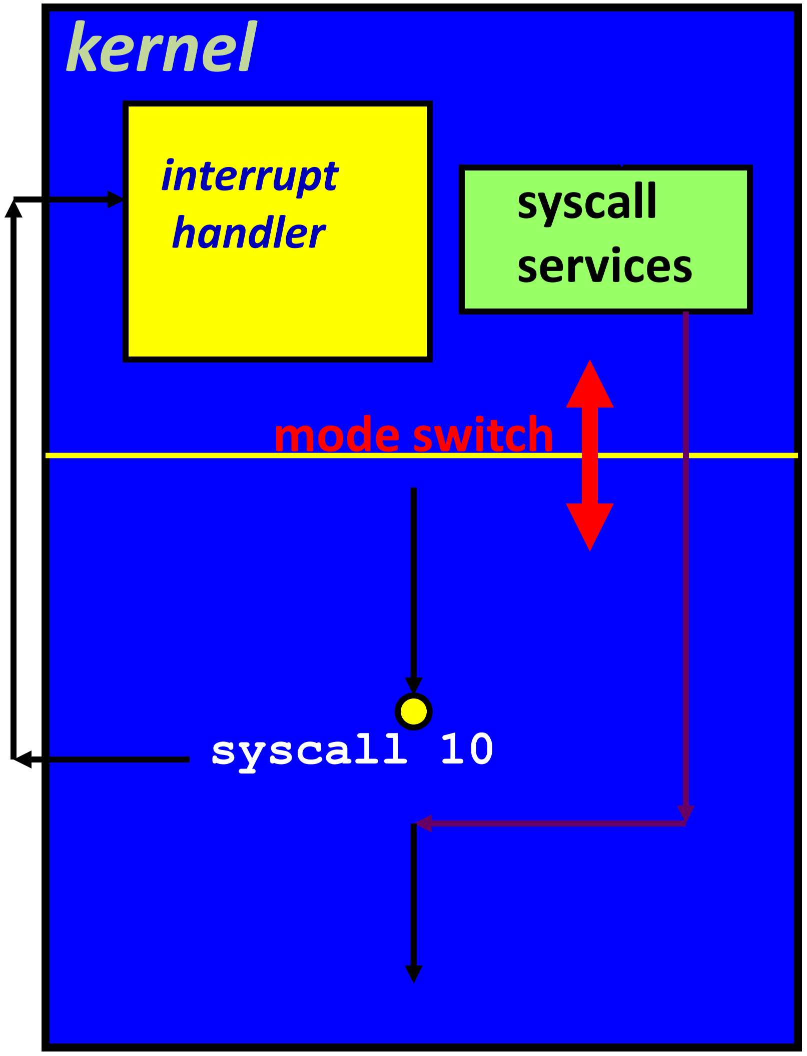
Kernel Stub

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

System Call Mechanism: 1/2



System Call Mechanism: 2/2

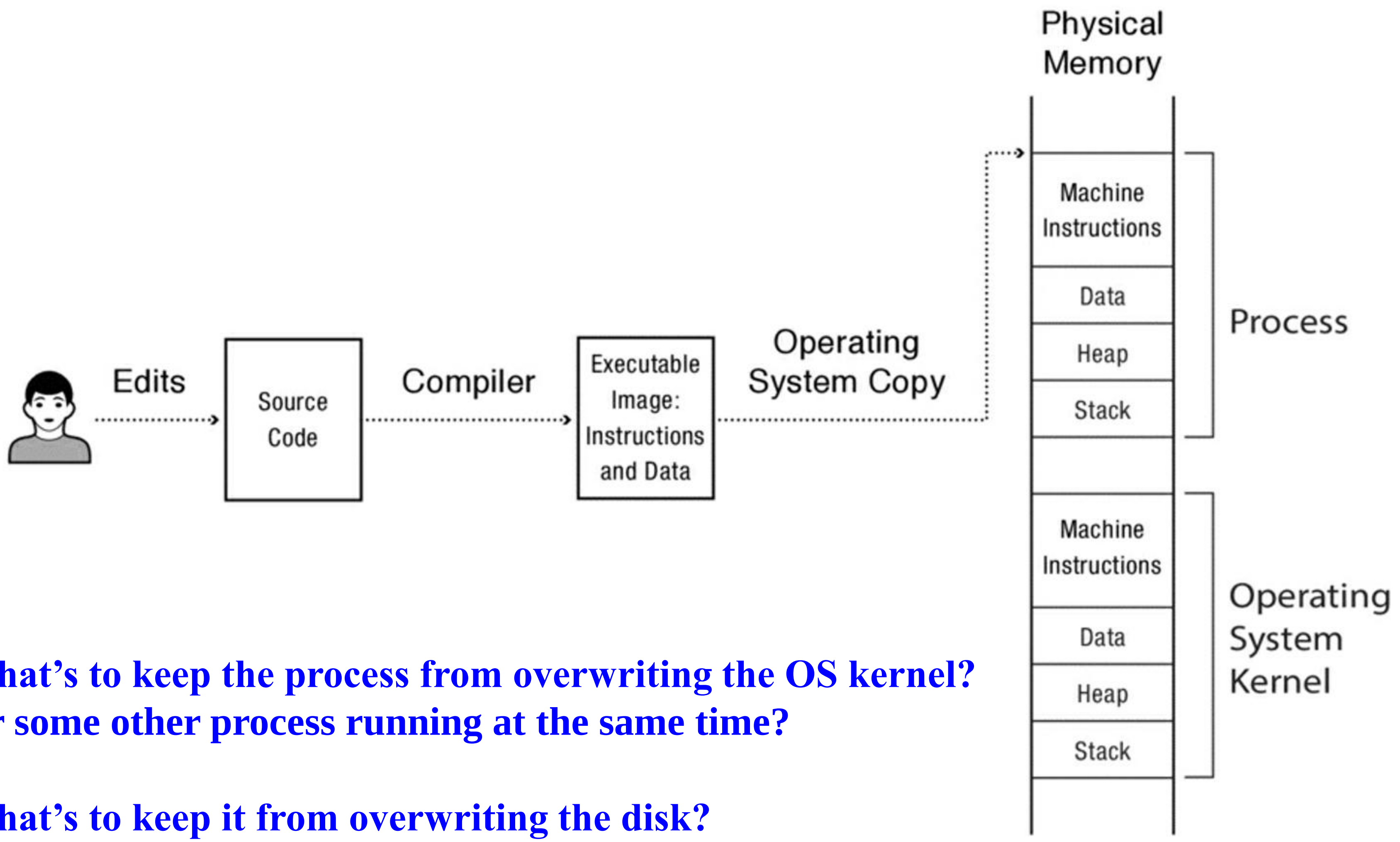


- A system call generates a *trap*.
- The executing program (i.e., caller) is suspended.
- Control is transferred to the OS.
- A program will be resumed when the system call service completes.

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

A Problem



What's to keep the process from overwriting the OS kernel?
Or some other process running at the same time?

What's to keep it from overwriting the disk?
From reading someone else's files that are stored on disk?

Main Points

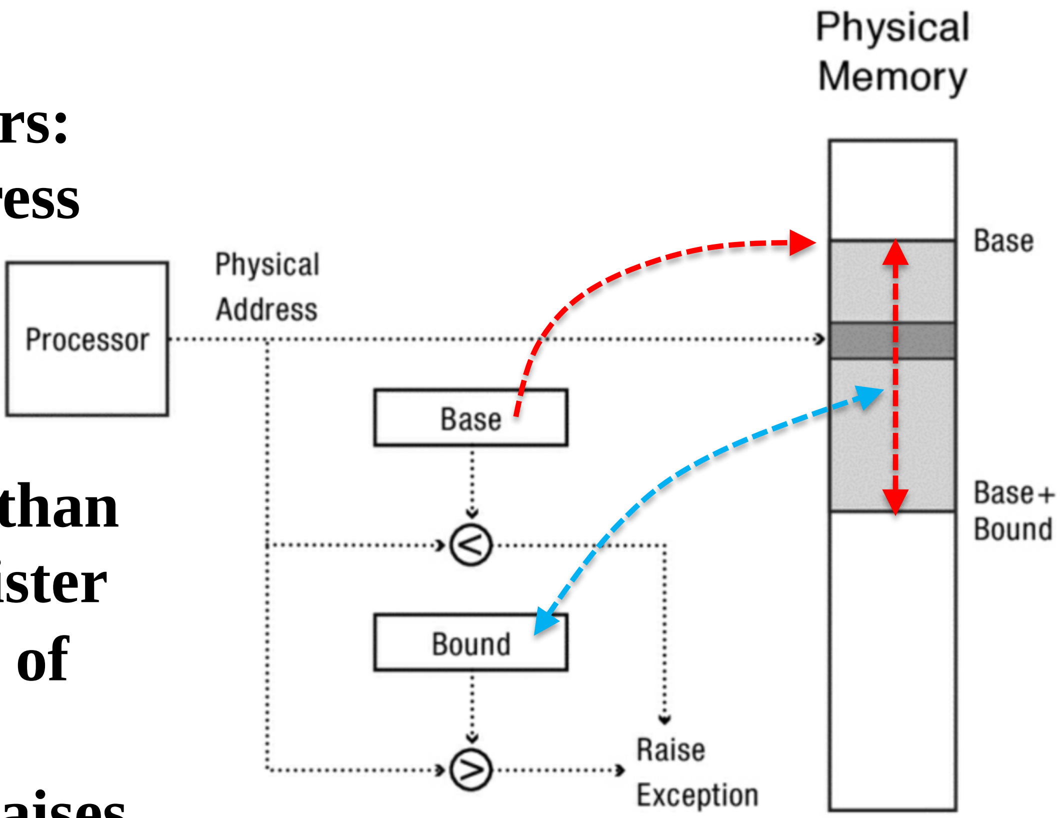
- Process concept
  - A process is the OS abstraction for executing a program with limited privileges
- Dual-mode operation: user vs. kernel
  - ***Kernel-mode***: execute with complete privileges
  - ***User-mode***: execute with fewer privileges
- Safe control transfer
  - How do we switch from one 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)

Simple Memory Protection

- Each process has two registers:
  Base: points to the 1st address
  Bound: length of a process
- The processor generates a physical address.
- This address must be larger than the value in the ***Base*** register and smaller than the value of ***Base*** + ***Bound***.
- If a test fails, the hardware raises an exception via an interrupt.
- There is no "relocation" here.
  Will address this issue later in this semester.



Towards Virtual Addresses

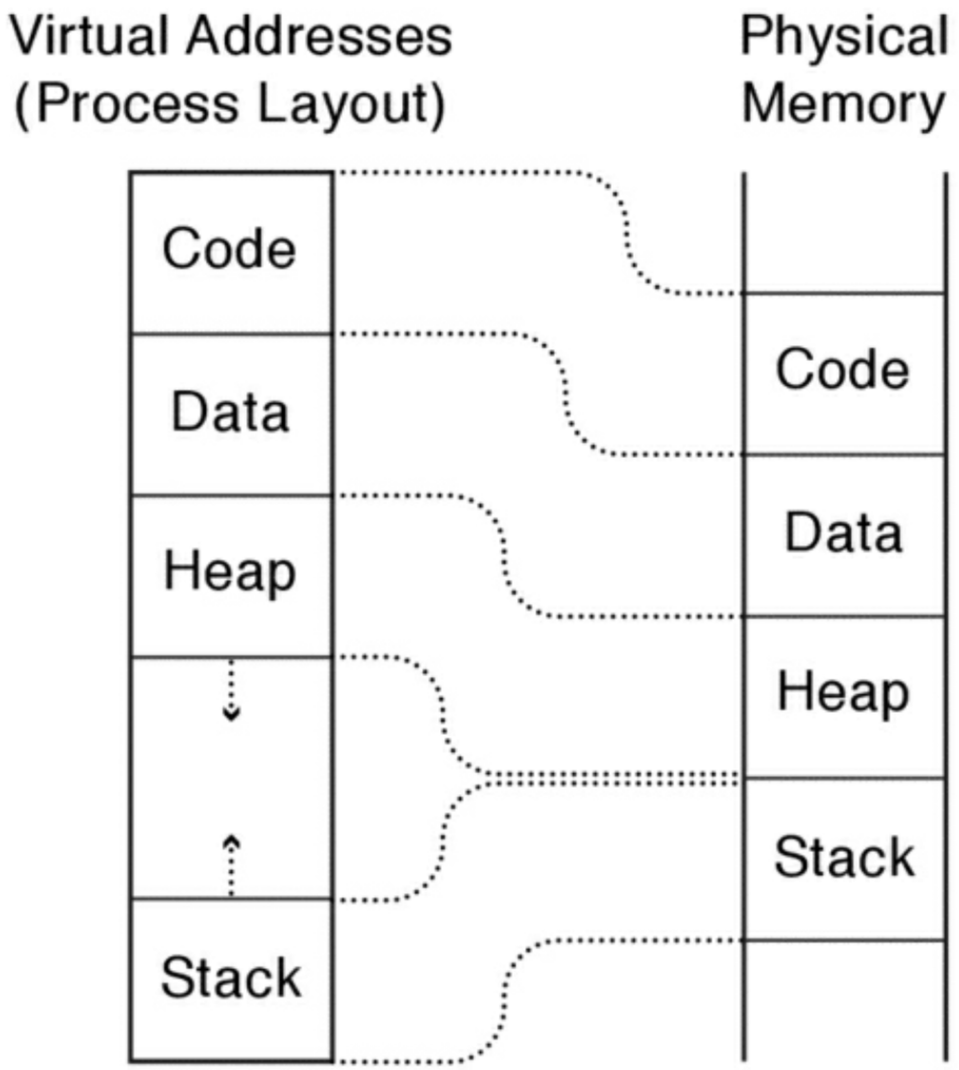
- Problems with base and bounds?
  - Expandable heap and/or stack?
  - Memory sharing between processes (e.g., shared memory segments)
  - Memory fragmentation
  - What if some memory segments should be moved around?

Logical, Virtual, Physical Address

- ❑ **Logical Address: the address generated by the CPU.**
- ❑ **Physical Address: the address seen and used by the memory unit.**
- ❑ **Virtual Address: Run-time binding may generate different logical address and physical address. In this case, logical address is also referred to as virtual address. (Logical = Virtual in this course)**

Virtual Addresses

- Translation done in hardware, using a table.
- Table set up by operating system kernel.
- Each section may be further cut into smal pages scattering all over the physical memory.



Hardware Timer: 1/2

- Because the operating system must maintain the control over the CPU, it has to prevent a user program from getting the CPU forever without calling for system service (i.e., I/O).
- Use an interval timer! An interval timer is a count-down timer.
- Before a user program runs, the OS sets the interval timer to certain value. Once the interval timer counts down to 0, an interrupt is generated and the OS can take appropriate action.

Hardware Timer: 2/2

- 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

Device Interrupts: 1/2

- 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: 2/2

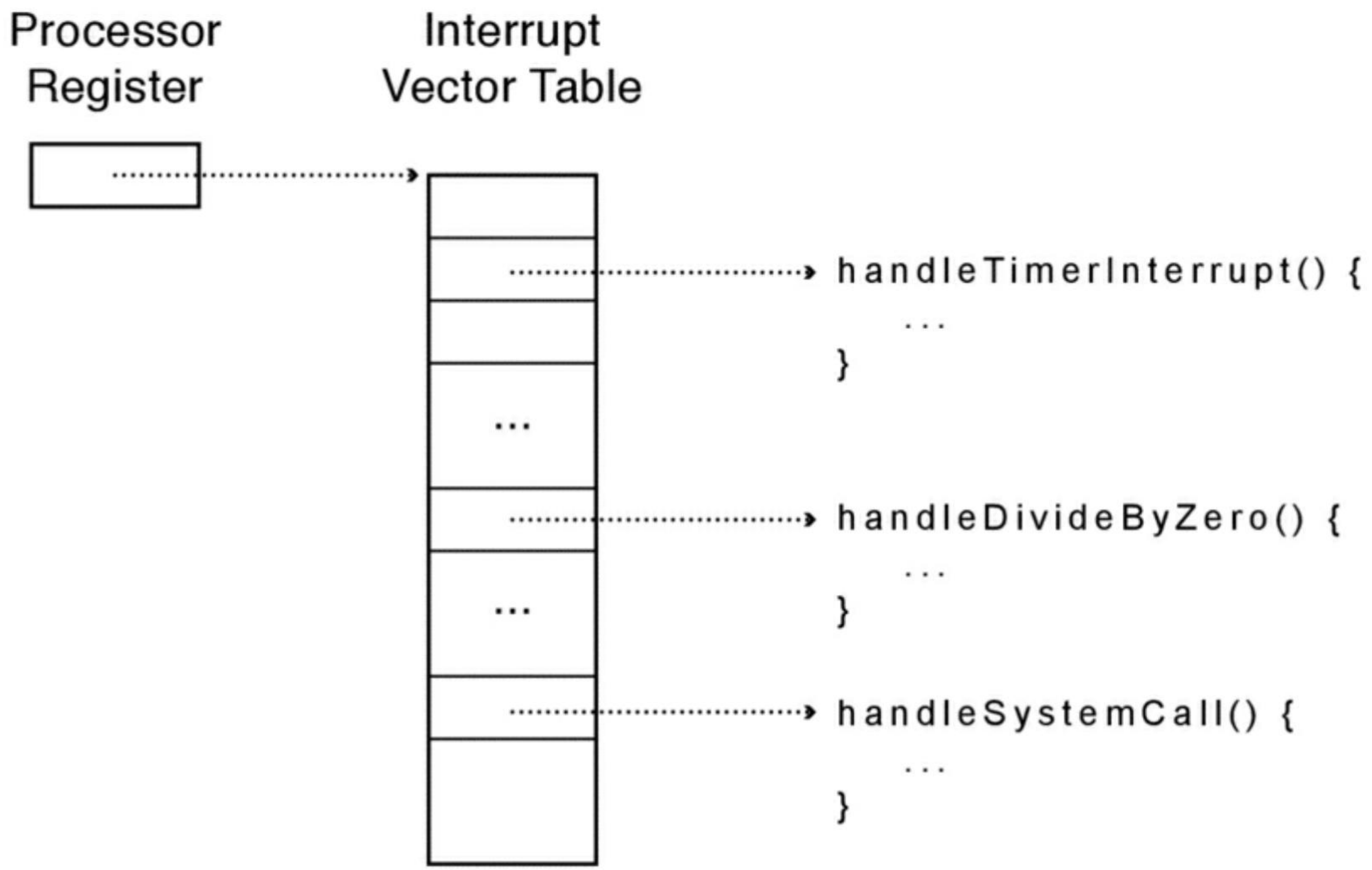
- How do device interrupts work?
  - Where does the CPU run after an interrupt?
  - What is the interrupt handler written in? C? Java?
  - What stack does it use?
  - Is the work the CPU had been doing before the interrupt lost forever?
  - If not, how does the CPU know how to resume that work?

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

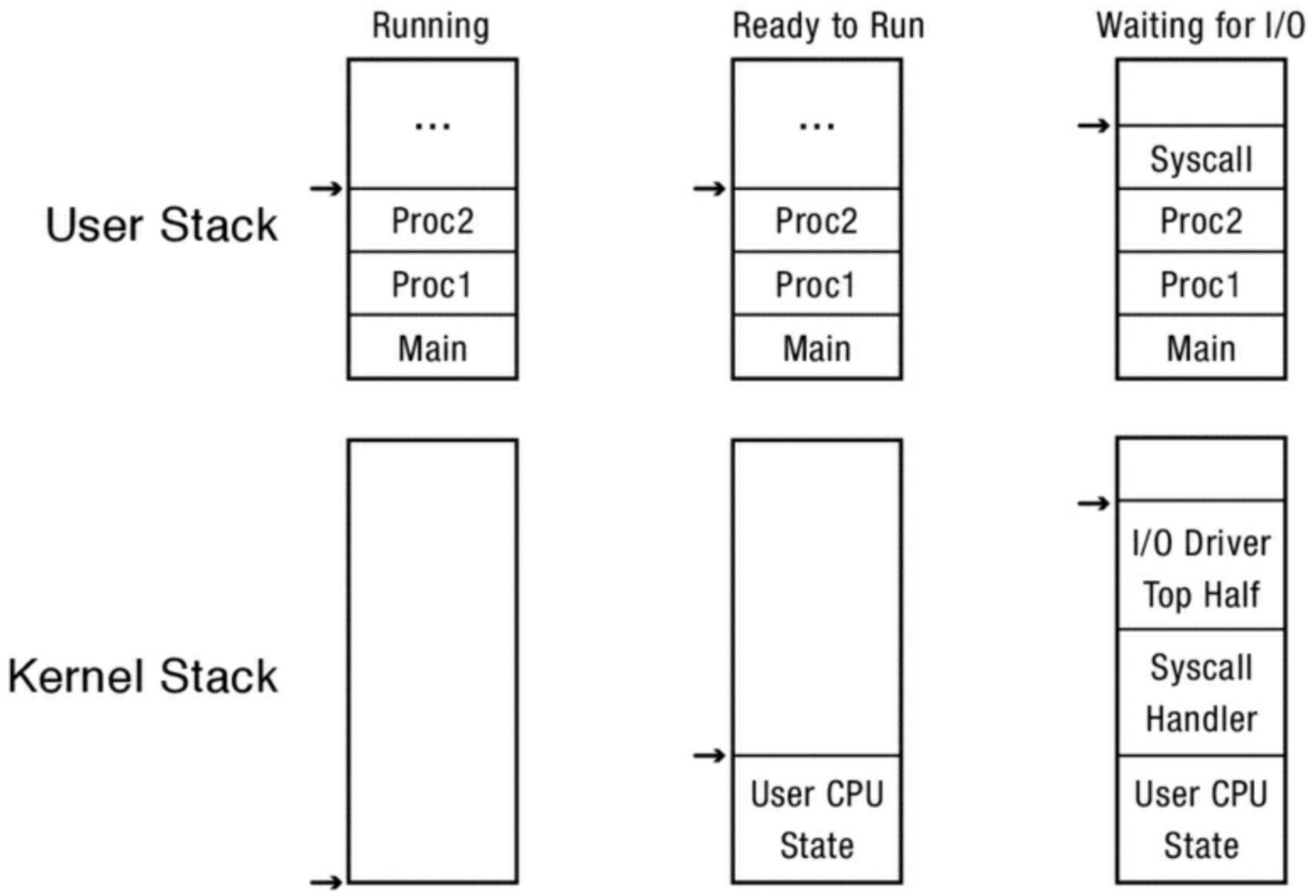


```
handleTimerInterrupt() {
    ...
}

handleDivideByZero() {
    ...
}

handleSystemCall() {
    ...
}
```

Interrupt Stack: 1/2

- 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: 2/2



Interrupt Masking

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Example: when determining the next process or thread to run
  - On x86
    - ✓ CLI (clear the interrupt flag in the `EFLAGS`): disable interrupts
    - ✓ STI (set the interrupt flag): enable interrupts
    - ✓ Only applies to the current CPU (on a multicore)

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

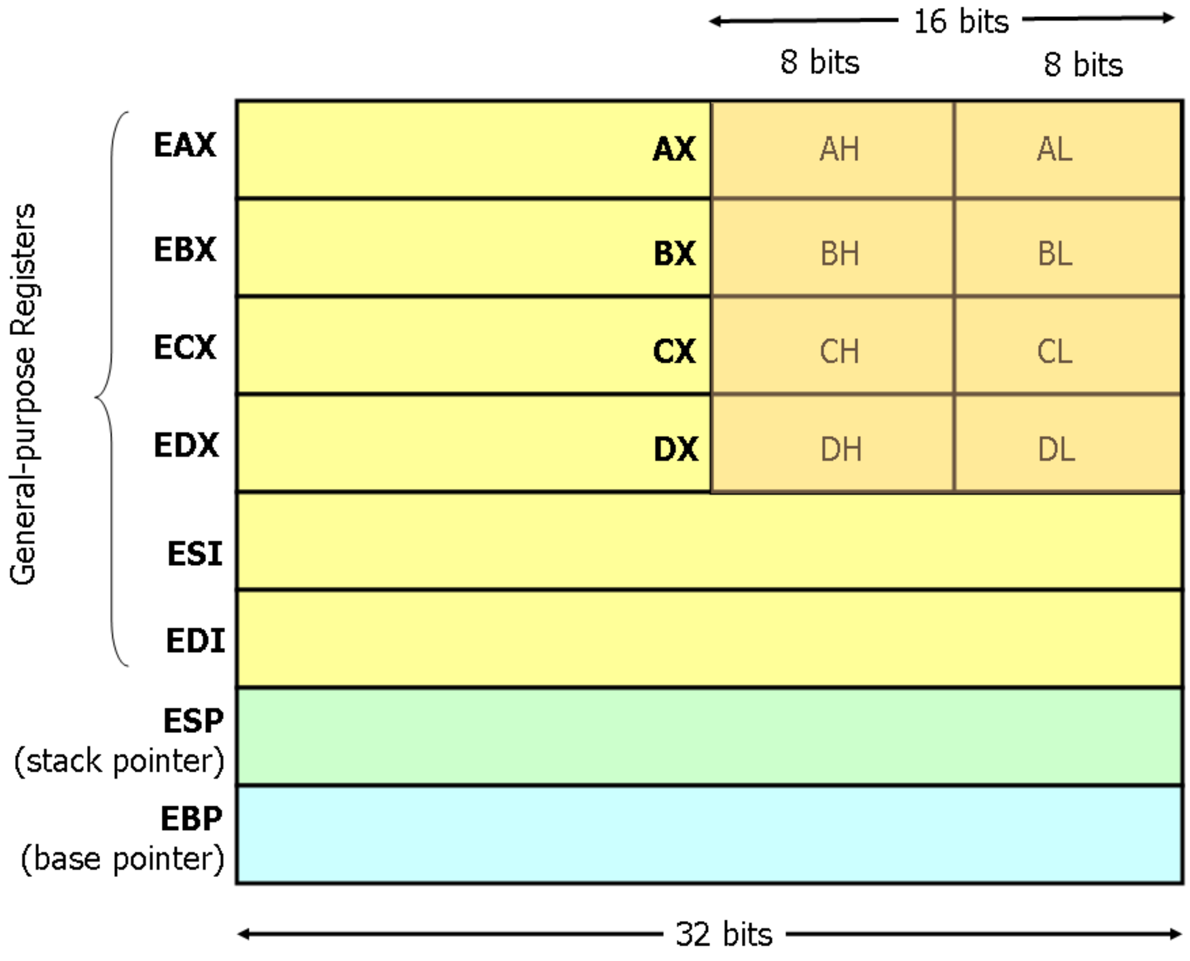
Case Study: MIPS Interrupt/Trap

- 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, interruptible 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

x86 Registers

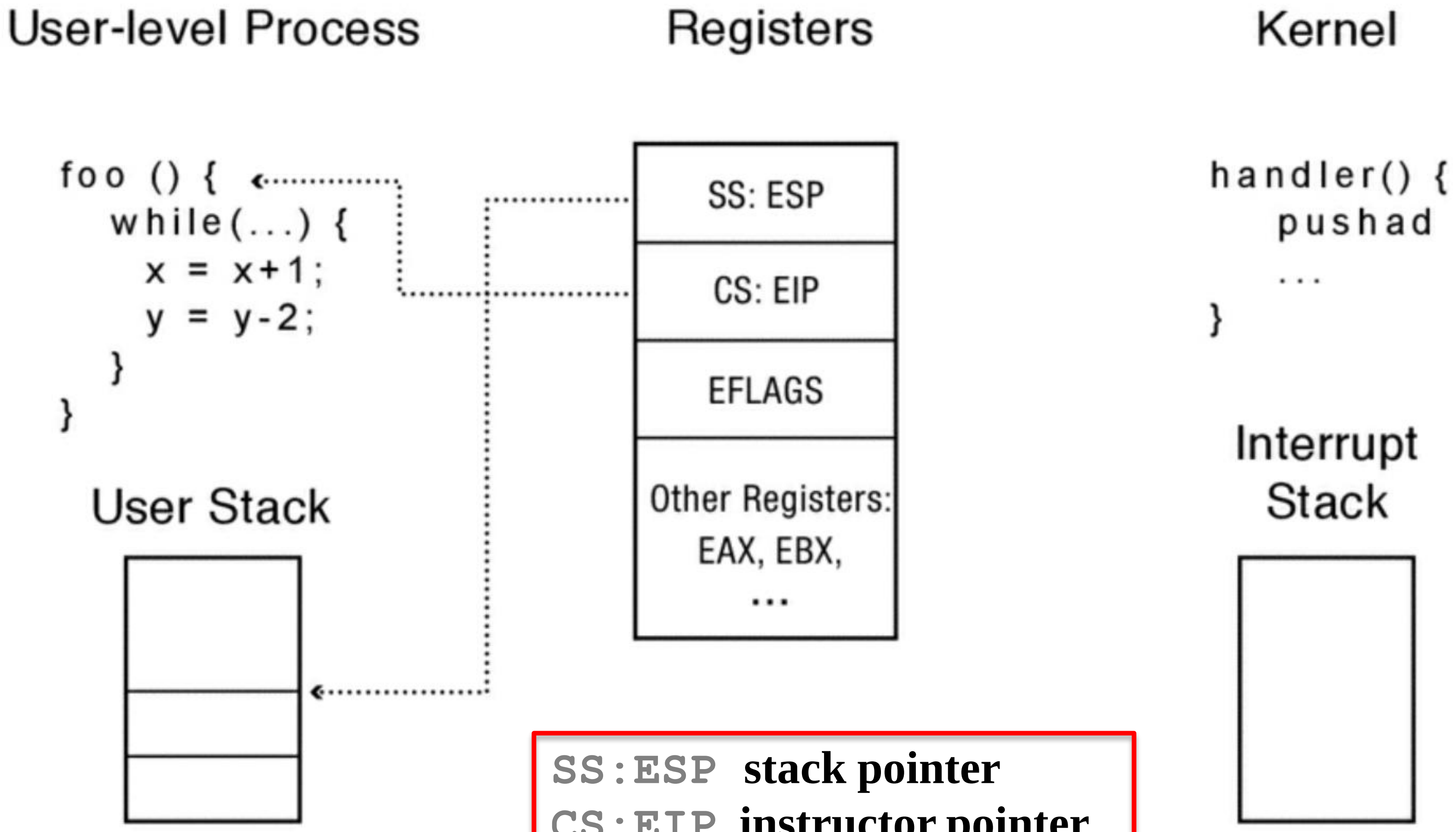


80286 introduced 4 segments:

- ❑ CS – code segment
- ❑ DS – data segment
- ❑ SS – stack segment
- ❑ ES – extra (E) segment
- ❑ FS – pointer to more extra data. F comes after E
- ❑ GS – pointer to more extra data. G comes after F

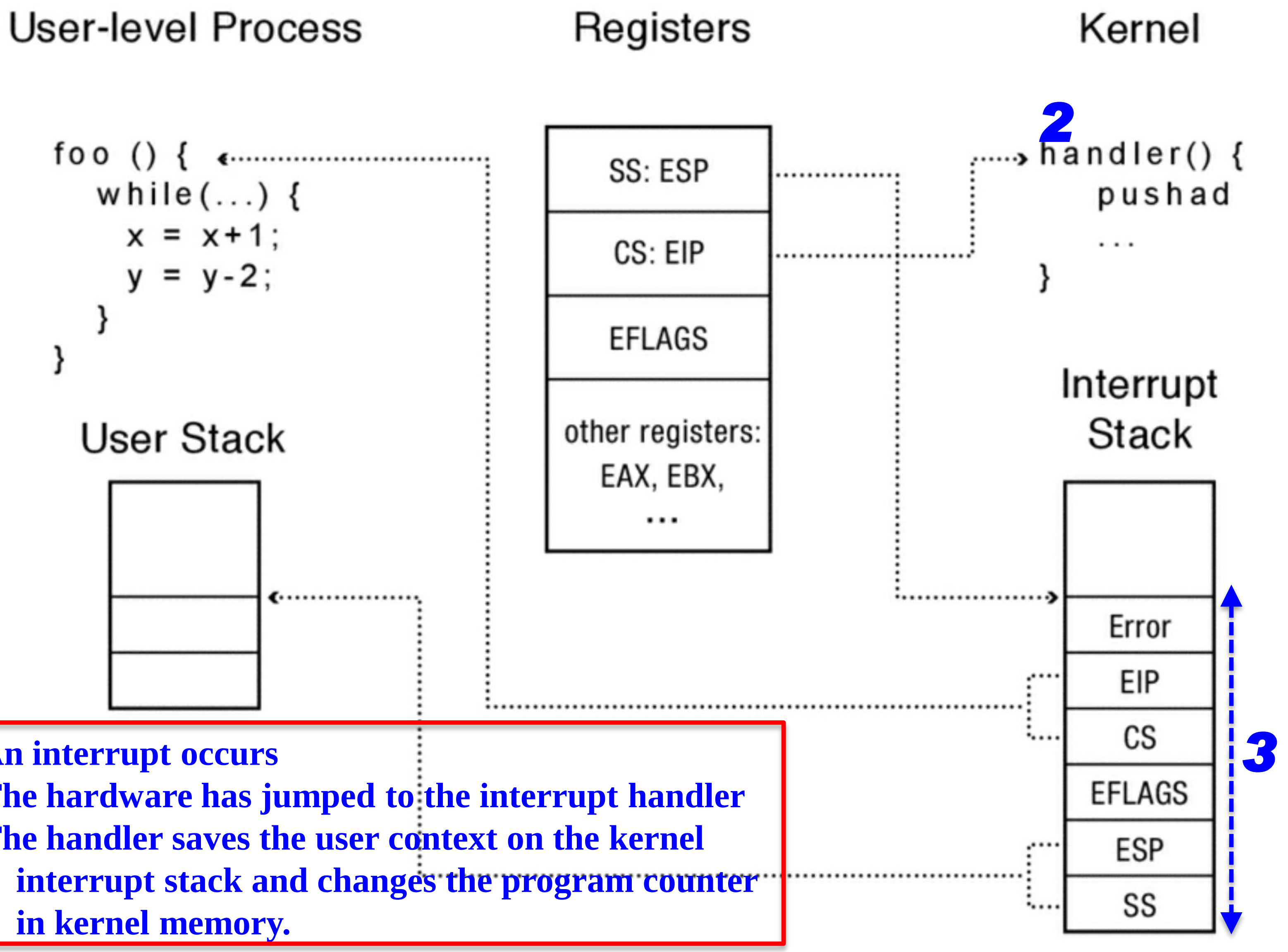
EFLAGS – a 32-bit register for storing status of processor

Before Interrupt



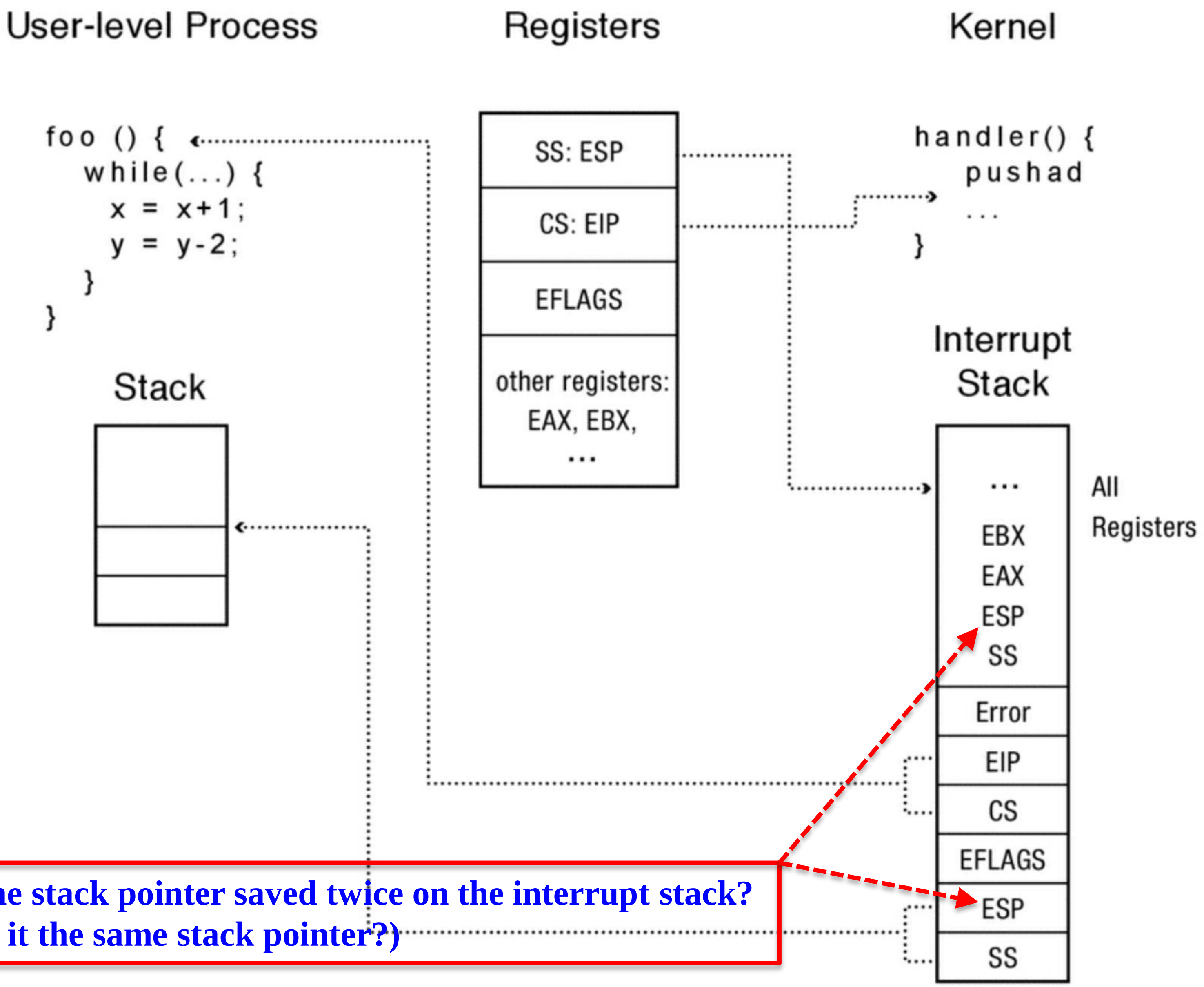
SS:ESP stack pointer
CS:EIP instructor pointer
(program counter)

During Interrupt



1. An interrupt occurs
2. The hardware has jumped to the interrupt handler
3. The handler saves the user context on the kernel interrupt stack and changes the program counter in kernel memory.

After Interrupt



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

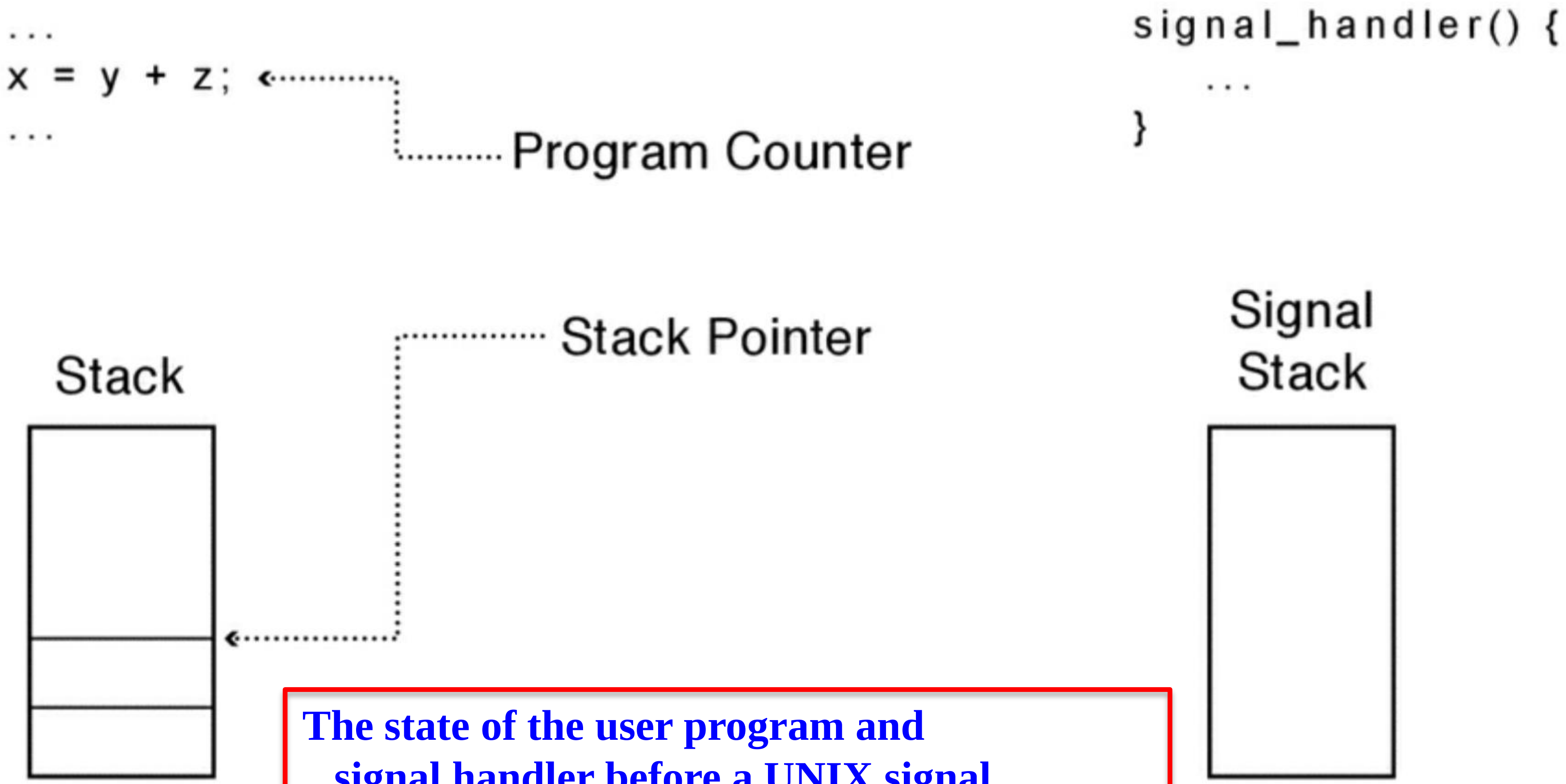
Upcall: User-level Event Delivery

- Many operating systems provide user programs with the ability to receive asynchronous notification of event.
- This mechanism is similar to kernel interrupt handling, except at the user level.
- It notifies user process of some event that needs to be handled right away
  - Time expiration
  - Interrupt delivery for VM player
  - Asynchronous I/O completion (async/await)
- AKA UNIX signal

Upcalls vs Interrupts

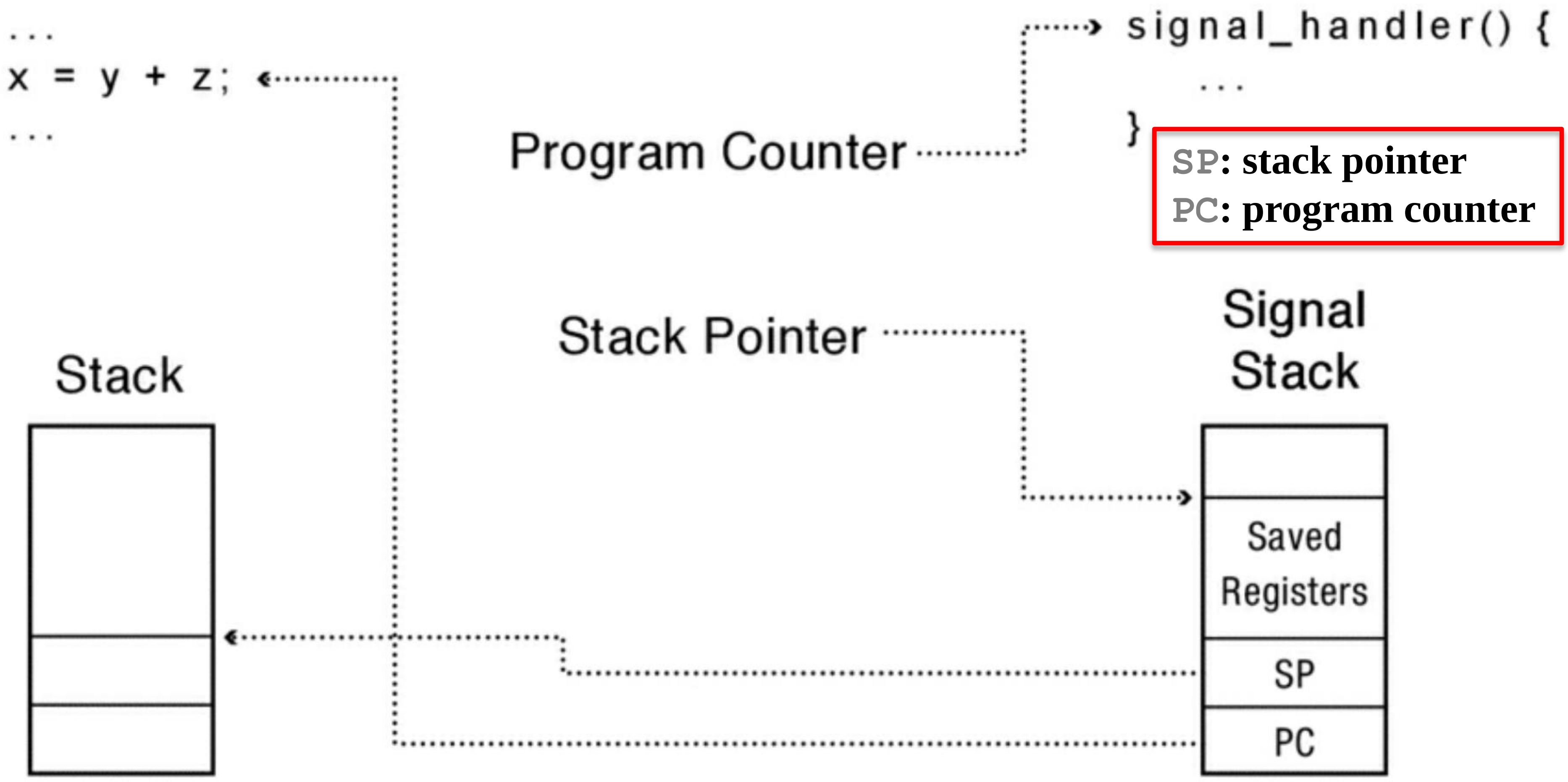
- Signal handlers ≈ interrupt vector
- Signal stack ≈ interrupt stack
- Automatic save/restore registers ≈ transparent resume
- Signal masking: signals disabled while in signal handler

Upcall: Before



The state of the user program and
signal handler before a UNIX signal.
UNIX signals behave like processor exception,
nut at user level.

Upcall: During



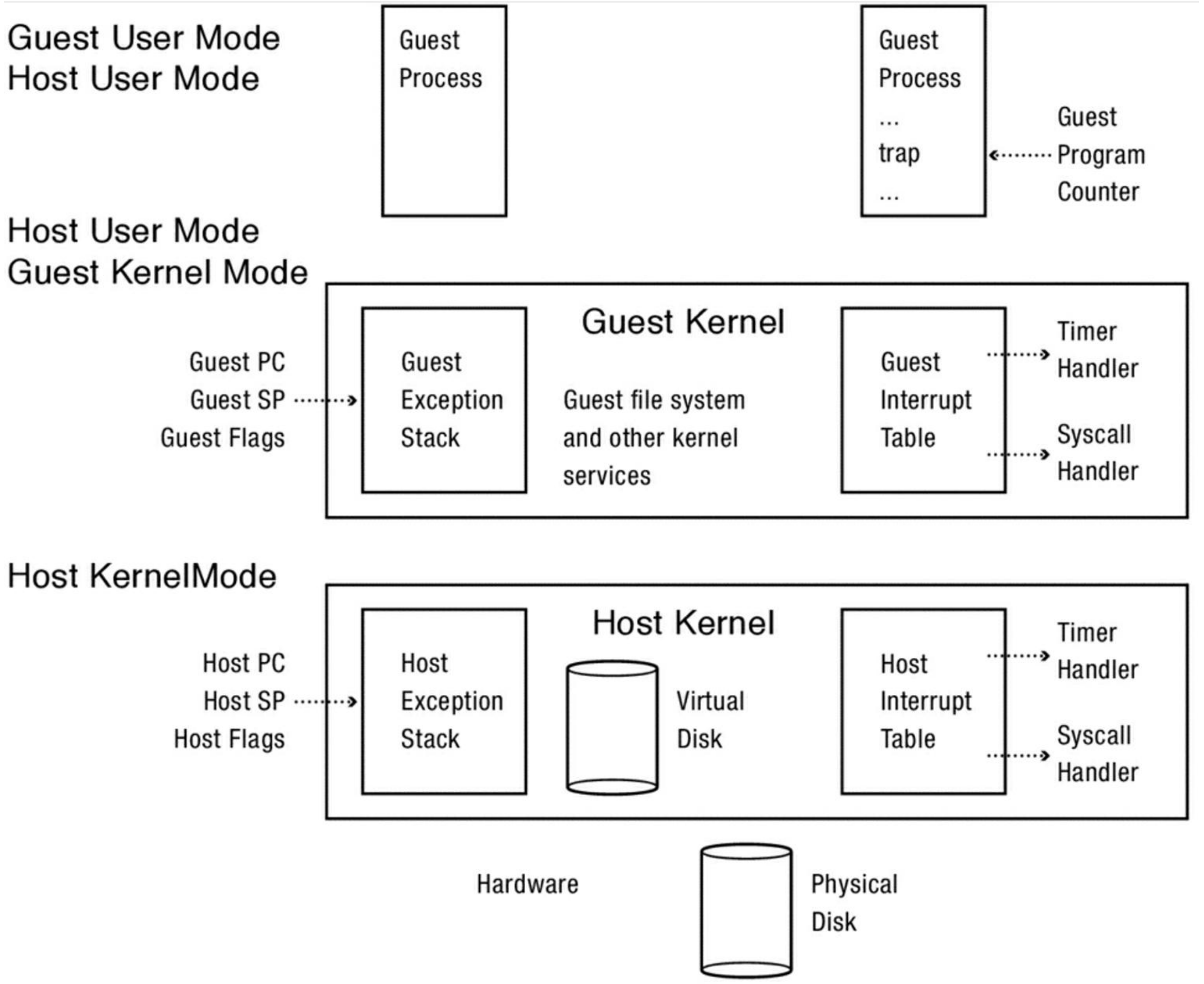
SP: stack pointer
PC: program counter

The signal stack stores the state of the hardware registers at the point where the process was interrupted, with room for the signal handler to execute on the signal stack.

User-Level Virtual Machine: 1/6

- The host OS provides the illusion that the guest kernel is running on real hardware.
- The guest kernel provides a guest disk and the host kernel simulates a virtual disk as a file on the physical disk.
- The host kernel must manage memory protection to provide the illusion that the guest kernel is managing its own memory protection even though it is running with virtual address.

User-Level Virtual Machine: 2/6



User-Level Virtual Machine: 3/6

- How does the host kernel manage mode transfer between guest processes and the guest kernel?
  1. During boot, the host kernel initializes its interrupt vector to its own interrupt handlers in host kernel memory.
  2. When the host kernel starts the VM, the guest kernel starts running as if it is being booted.
  3. The host loads the guest bootloader from the virtual disk and starts it running.
  4. The guest bootloader loads the guest kernel from the virtual disk into memory and starts it running.
  5. The guest kernel initializes its interrupt vector table to point to the guest interrupt handlers.

User-Level Virtual Machine: 4/6

- ***Continue from the previous slide:***
  6. **The guest kernel loads a process from the virtual disk into guest memory.**
  7. **To start a process, the guest kernel issues instruction to resume execution at user level. Because changing the privilege level is a privileged operation, this instruction traps into the host kernel.**
  8. **The host kernel simulates the requested mode transfer as if the processor had directly executed it.**

User-Level Virtual Machine: 5/6

- **How does the host kernel manage system call by the guest kernel?**
  1. **When the guest kernel executes a system call, this causes a trap into the host kernel.**
  2. **The host kernel saves the instruction counter, processor status register, and user stack pointer on the interrupt stack of the guest kernel.**
  3. **The host kernel transfers control to the guest kernel at the beginning of the interrupt handler, but with the guest kernel running in user mode.**
  4. **The guest kernel performs the system call – saving user states and checking arguments.**

User-Level Virtual Machine: 6/6

- ***Continue from the previous slide:***
  5. **When the guest kernel attempts to return from the system call back to user level, this causes a processor exception, dropping back to the host kernel.**
  6. **The host kernel can restore the state of the user process, running at user level, as if the guest OS had been able to return there directly.**

The End