

Reading Reference: Textbook 1 Chapter 2

# ARCHITECTURAL SUPPORT FOR OS

Tanzir Ahmed  
CSCE 313 Spring 2020

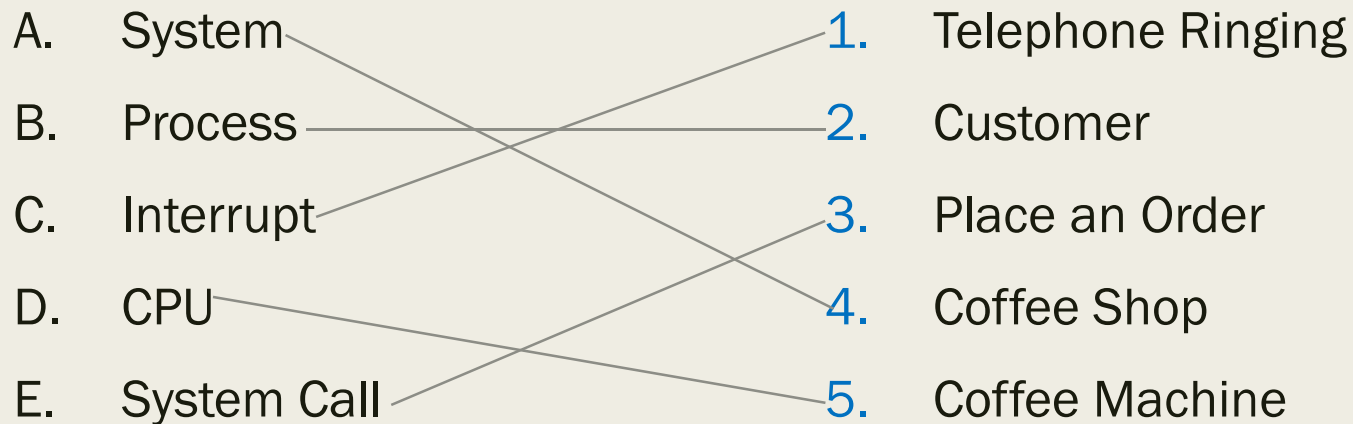
# Theme of Today's Discussion

- Understand **Dual-Mode Operation** (User Mode versus Kernel Mode)
  - *Also called Limited-Direct Execution*
- Combine that understanding with that of Exception Control Flow

# Quick Recap – Find Best Match for the Coffee Shop Analogy

## Computer Systems

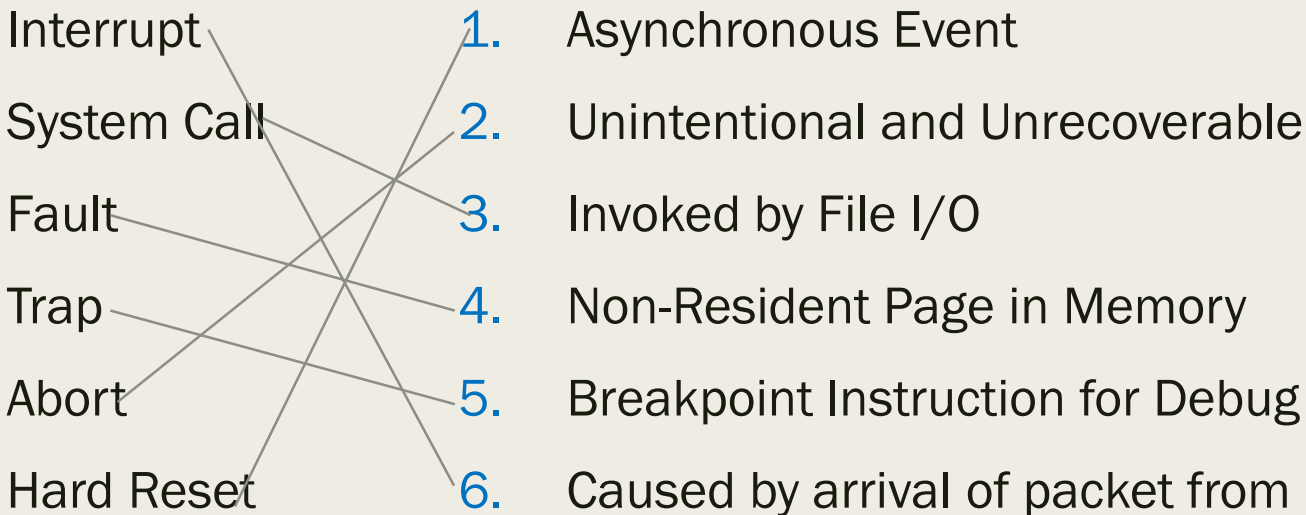
## Analogy



# Exceptions Recap – Find Best Match

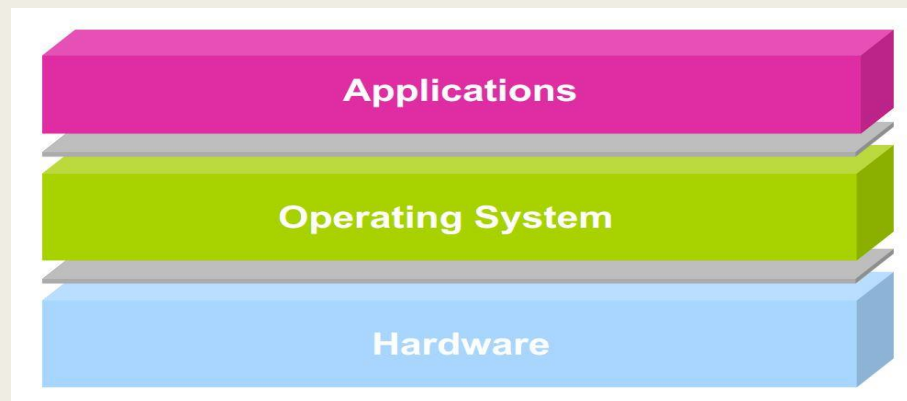
## Exception Type

## Definition

- |                |   |
|----------------|---|
| A. Interrupt   | 1. Asynchronous Event                       |
| B. System Call | 2. Unintentional and Unrecoverable          |
| C. Fault       | 3. Invoked by File I/O                      |
| D. Trap        | 4. Non-Resident Page in Memory              |
| E. Abort       | 5. Breakpoint Instruction for Debug         |
| F. Hard Reset  | 6. Caused by arrival of packet from network |
- 

# Architectural Support for OS

- Operating systems mediate between applications and the physical hardware of the computer
  - *Key goals of an OS are to enforce **protection** and **resource sharing***
  - *If done well, applications can be oblivious to HW details*



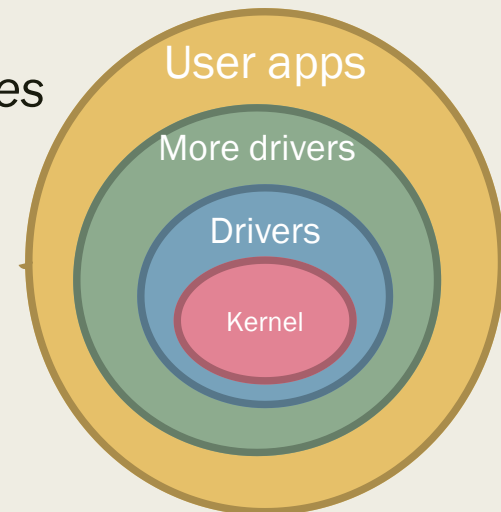
# Architectural Features

The features we design in HW to facilitate the OS to meet some key challenges

- Protection Modes
  - *Protection Ring / 2 modes: User/kernel*
  - *Privileged Instructions*
- Interrupts and Exceptions
- System Calls
- Timers (clock)
- Memory Protection Mechanisms
- I/O Control and Operation
- Synchronization Primitives (e.g., atomic instructions)

# Dual-mode Execution

- Every CPU has at least 2 modes of execution (the CPU alternates between the modes)
  - **Kernel-mode:** Execution with the **full privileges** of the hardware
    - E.g. Read/write to any memory, access any I/O device, read/write any disk sector, send/receive any packet
  - **User-mode:** Execution with **Limited privileges**
    - Only those granted by the operating system kernel
- Some architectures support more than 2 modes
  - Example: Both Intel and AMD support 4 modes
  - Together they are called **protection ring**
- However, both Windows and Linux support only 2 modes of operation
  - Called **Dual-mode operation**



# Dual-mode Operation

- Depending on the architecture, execution mode is stored either in the **Program Status Word (PSW)** register or scattered on multiple registers (e.g., **EFLAGS** in Intel)
- The **mode** tells us whether the instruction should be **checked** or not
  - *If set (i.e., in user mode), each instruction is checked to see if **allowed***
    - E.g., CLF (Clear Interrupt Flag) is never allowed in user mode; otherwise apps could ignore all Interrupts
  - *If not set (i.e., in kernel mode), no check is performed*

Allowed in both modes

Only allowed in  
Kernel mode

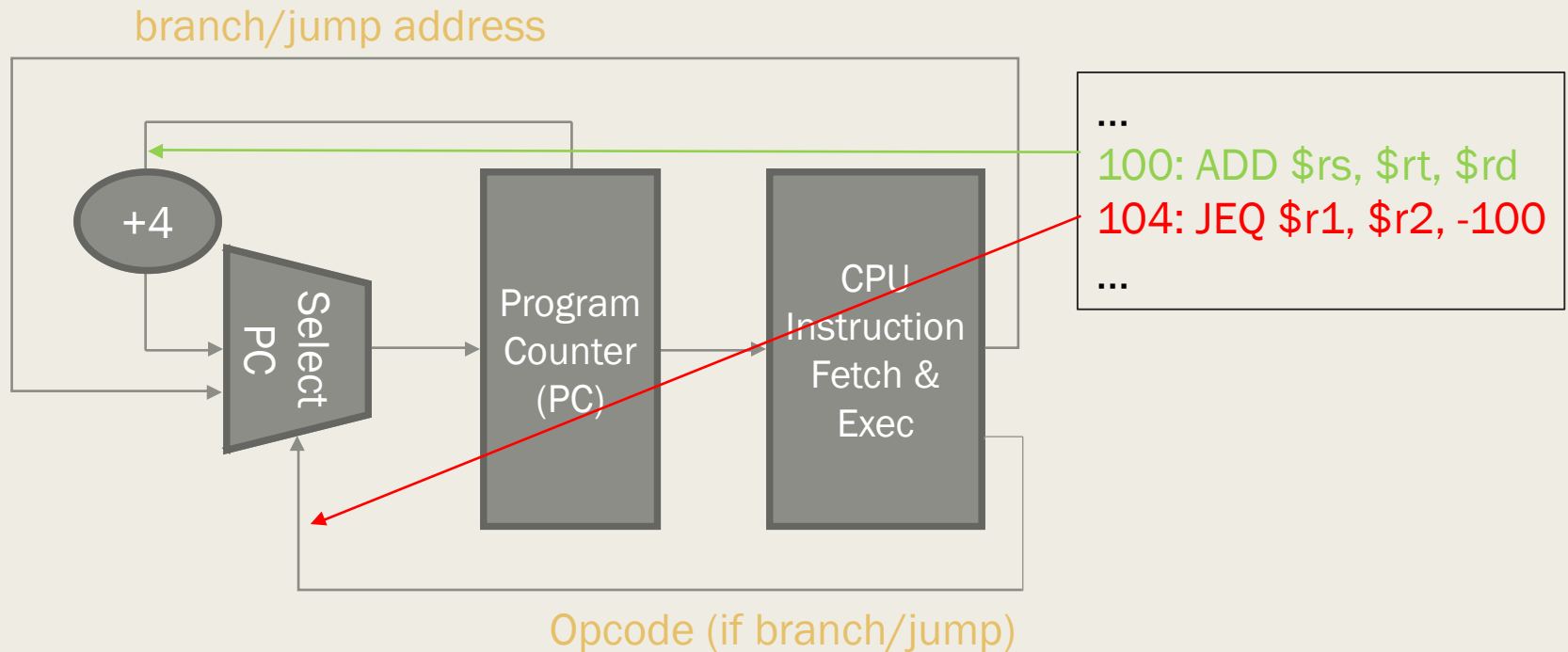
Full Instruction Set



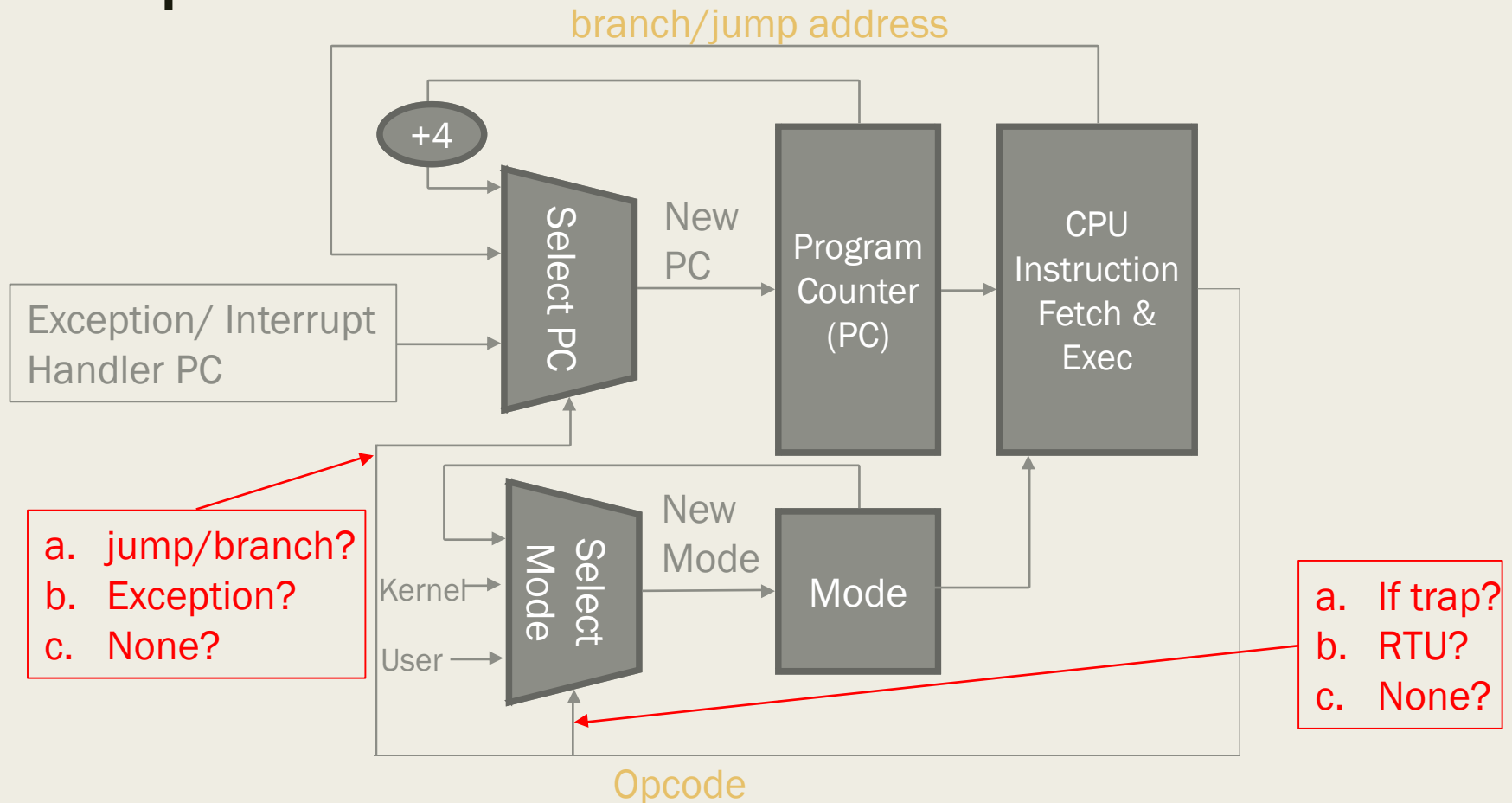
# Mode Switch

- Safe control transfer
  - *Mode separation does NOT mean that a user program cannot request a Kernel-mode operation*
    - User mode to kernel mode switch is very common (e.g., printf/cout, writing to a file)
  - *How do we switch from one mode to the other?*
    - Such that the protection is not compromised

# A Simple CPU Model



# A CPU with Dual-Mode Operation



# Dual-Mode Operation: Minimal Hardware Requirement

1. Privileged instructions
  - *Subsect of instructions available only to the kernel mode*
2. Limits on memory accesses
  - *To prevent user code from reading/overwriting the kernel*
3. Timer
  - *To regain control from a user program periodically*

# Privileged Instructions - Examples

- Only the OS should be able to
  - *Directly access I/O devices (disks, printers..)*
    - Allows OS to enforce security and fairness
    - User programs cannot possibly be fair to each other
  - *Manipulate memory management state*
    - E.g., page tables (Virtual-> Physical), protection bits, TLB entries, etc.
    - Processes use them, but cannot modify – that would defeat the protection
  - *Adjust protected control registers*
    - User  $\leftarrow \rightarrow$  Kernel modes or Raise/Lower interrupt level
  - *Execute CLF instruction*

A small cache  
for page  
tables



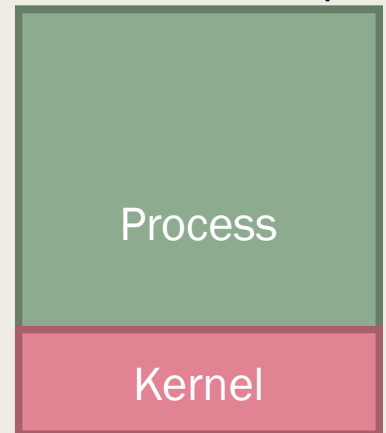
# Question

- What should happen if a user program attempts to execute a privileged instruction?
  - *Will be treated by the Kernel as an attempt to go around protection measures, and will result in terminating the user application*

# Memory Protection

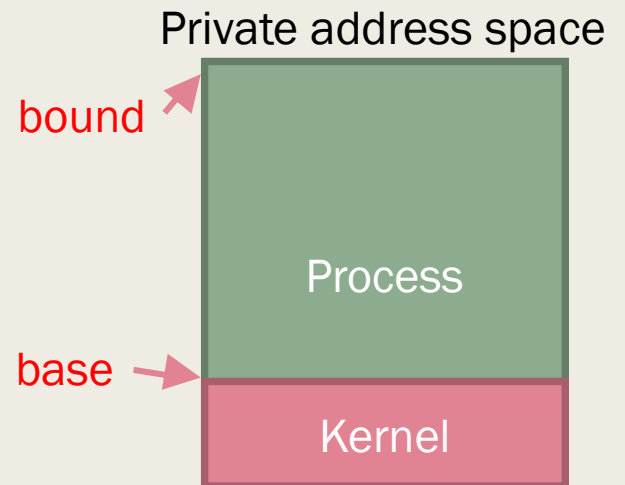
- Because of Virtual Memory, we know processes cannot see each other
- Then, why do we need any more memory protection?
  - *Because in the address space of each process, the kernel is mapped*
- Why is Kernel also included?
  - *To load and run the process in the first place*
  - *Handle Interrupts, exceptions, and system calls*
- Therefore, kernel must be protected from a faulty or malicious user program

Private address space



# Memory Protection (2)

- Could we just make Kernel memory read-only?
  - *No. Because kernel contain sensitive data that should not even be read*
  - *Example: page tables*
- A primitive but effective solution in hardware:
  - A **Base** and a **Bound** register for each process
  - Cannot access (read or write) anything below **base**
- This check is done only in User mode
  - *Kernel mode has unlimited access*





# Hardware Timer

- Operating system timer is a critical building block
  - *Many resources are time-shared; e.g., CPU*
  - *Allows OS to prevent infinite loops*
- Fallback mechanism by which OS regains control
  - *When timer expires, generates an interrupt*
  - *Handled by kernel, which controls resumption context*
    - **Basis for OS scheduler**; more later...
  - *Setting (and clearing) a timer is a privileged instruction*

# User → Kernel Mode Switch

- From user-mode to kernel-mode

- *Interrupts*

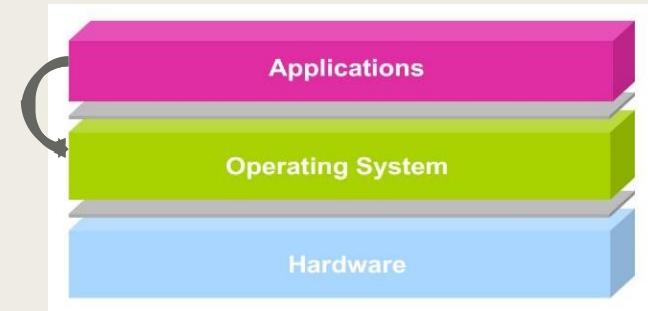
- Triggered by timer and I/O devices
    - Checked by the CPU after every instruction

- *(Synchronous) Exceptions (Faults and Aborts)*

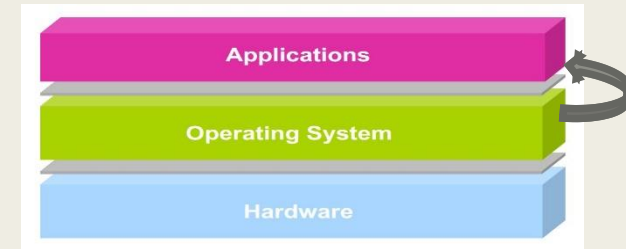
- Triggered by unexpected program behavior
    - Or malicious behavior!

- *System calls (traps) (aka 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

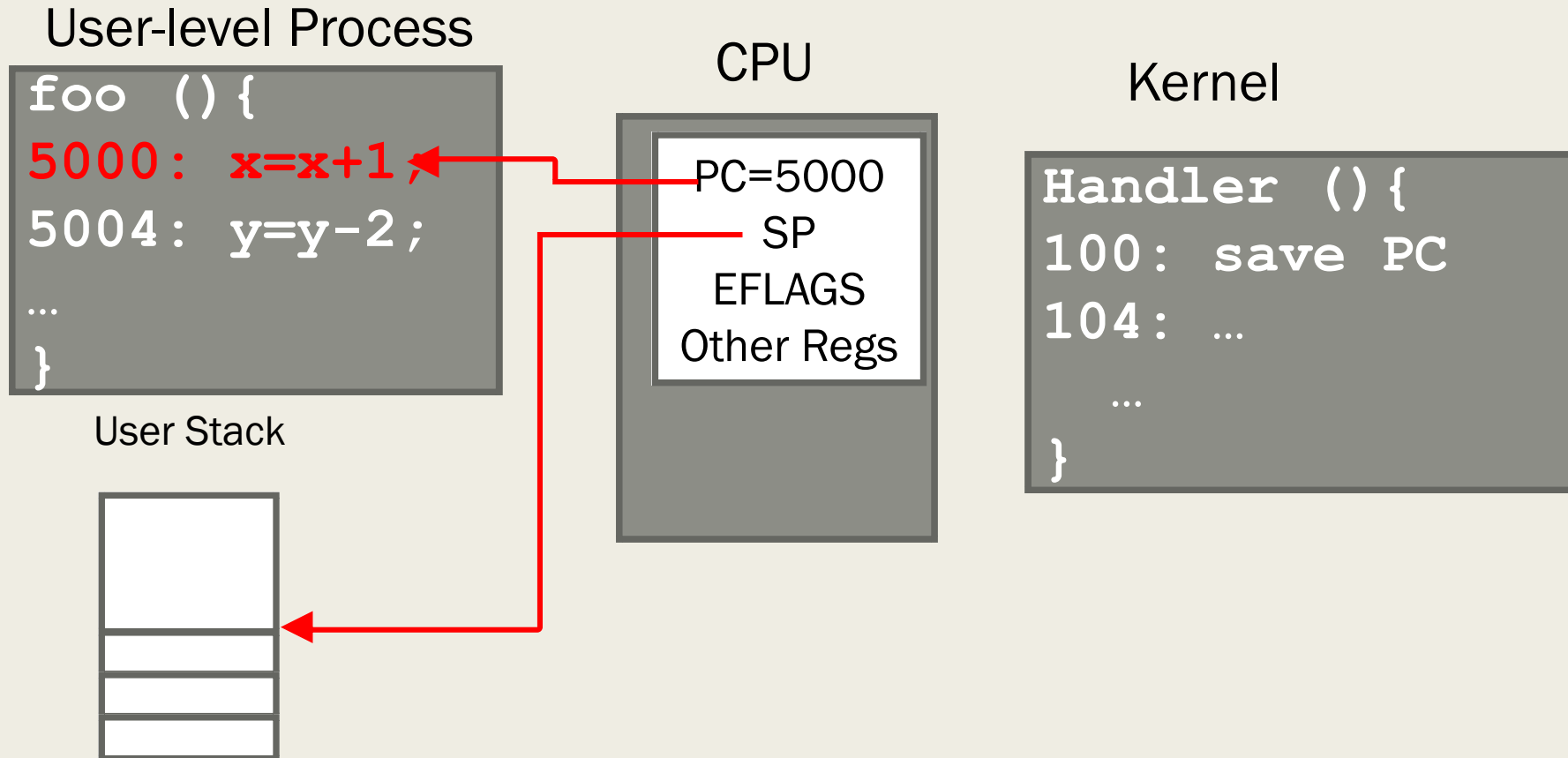


- From kernel-mode to user-mode
  - *New process/new thread start*
    - Jump to first instruction in program/thread
  - *Return from interrupt, exception, system call*
    - Resume suspended execution
  - *Process/thread context switch*
    - Resume some other process
  - *User-level upcall*
    - Asynchronous notification to user program by the kernel
    - Example: Writing customized Page Fault handler

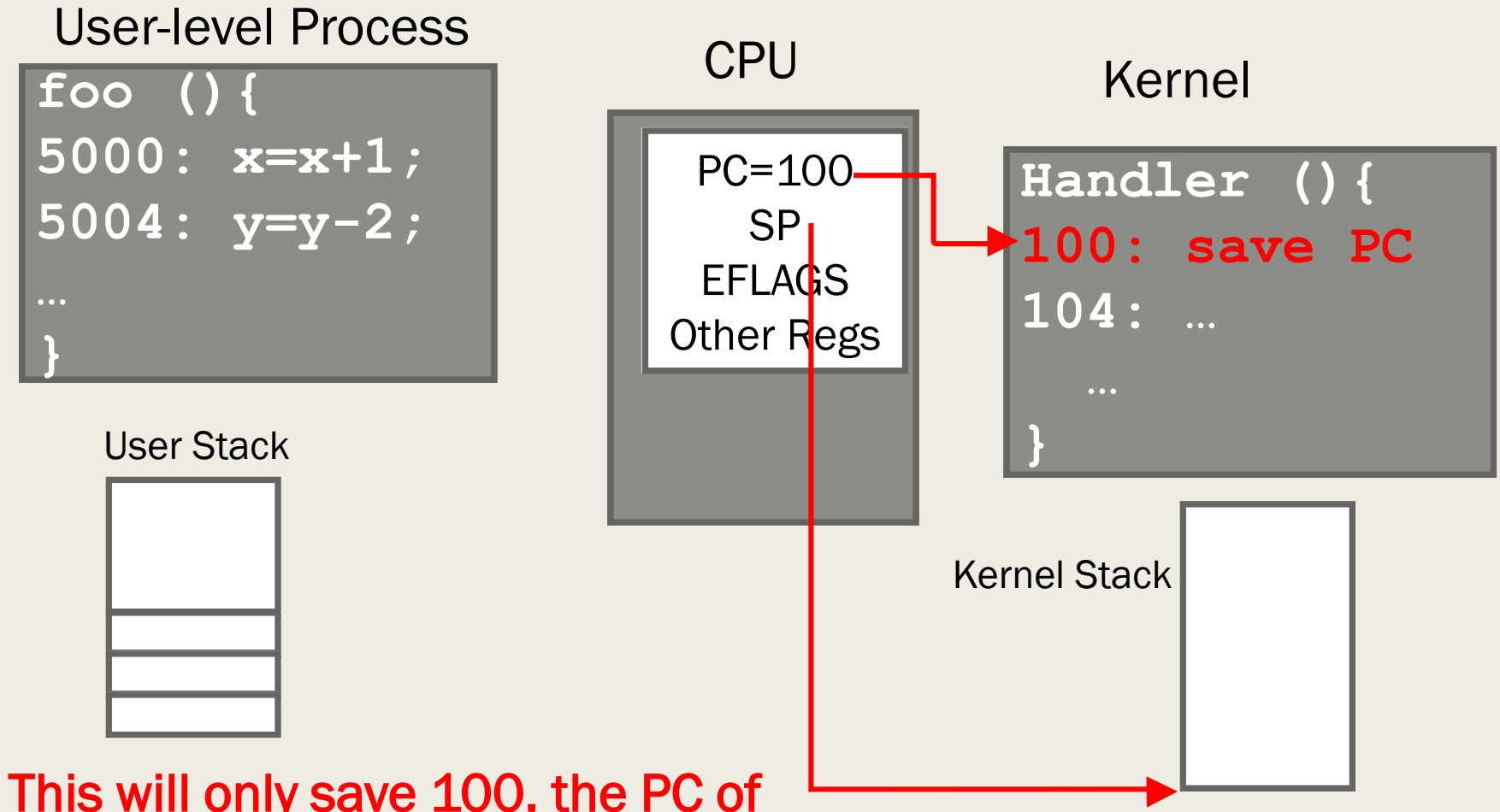
# Safe Mode Transfer – Interrupt Handling

- First, User->Kernel. The main idea is rather simple
  - *Store the state of the running process (so that it can be resumed later)*
  - *Execute the handler*
  - *Return to the interrupted process by restoring the saved state*
- But the actual implementation is a bit more complicated
- We first need to know which process info must be stored
  - *Basically, the Process Control Block (PCB)*

# Saving Process State: Difficulty



# Saving Process State: Difficulty



This will only save 100, the PC of the handler. The original PC=5000 of user app is lost forever!!

# To Summarize

- The processor has **only 1 set of SP, PC, EFLAGS etc.**
- Any piece of code (e.g. handler code as well) will require its own PC (and also SP and others) first loaded into the CPU
- Switching from User code to Handler code means **overwriting** PC, SP etc. with the handler PC, SP etc.
  - *But ALAS!!! We just lost the PC, SP for the user code*
  - *How can we ever recover those??*
- Quoting the Anderson book: *“This is akin to rebuilding the car’s transmission while it barrels down the road 60mph”*
- Solution: Take hardware help
  - *Clearly, any other code will also need PC, SP,..*
  - *Hardware does not need to use SP, PC to implement a logic*

# User to Interrupt Handler – Mechanism on x86

## ■ Hardware does the following:

1. *Mask further interrupts*
  - they are stored, not thrown away
  - Getting interrupted within handling an interrupt is problematic but possible
2. *Save PC, SP, EFLAGS in some special registers in CPU*
3. *Change SP to point to the **Kernel Interrupt Stack***
4. *Change mode to Kernel*
5. *Push PC, SP, EFLAGS in the special registers into the new stack the SP now points to (i.e., the Kernel Interrupt Stack)*
6. *Invoke the interrupt handler by vectoring through the Interrupt Vector Table (i.e., overwrite PC with the handler PC)*

## ■ Software (i.e., the handler code) does the following:

1. *Stores the rest of the general purpose registers being used by the interrupted process*
2. *Does the interrupt handling operation*

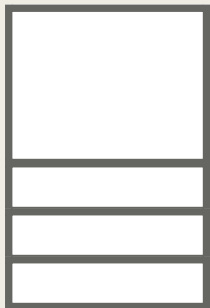


# Before

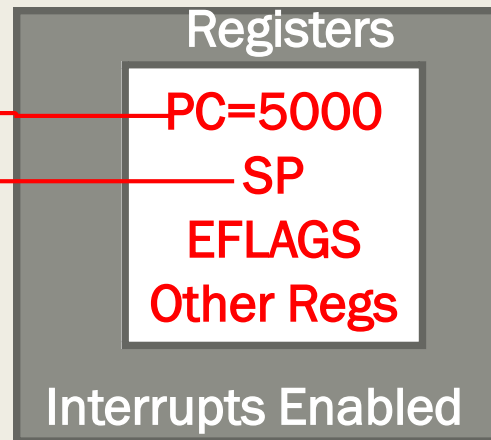
## User-level Process

```
foo () {  
  5000 : x = x + 1 ;  
  5004 : y = y - 2 ;  
}
```

## User Stack

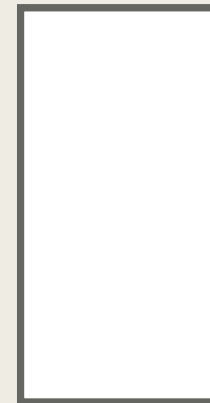


## CPU



## Kernel

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



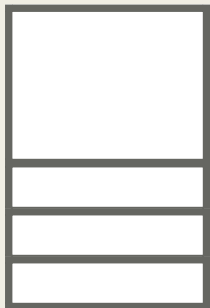
## Kernel Interrupt Stack

# Hardware Action

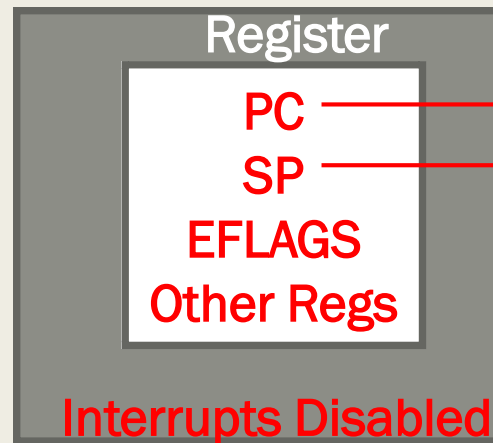
## User-level Process

```
foo () {  
  5000 : x = x + 1 ;  
  5004 : y = y - 2 ;  
}
```

## User Stack



## CPU



## Kernel

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



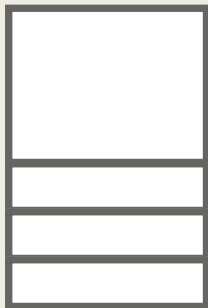
## Kernel Interrupt Stack

# As Handler Starts

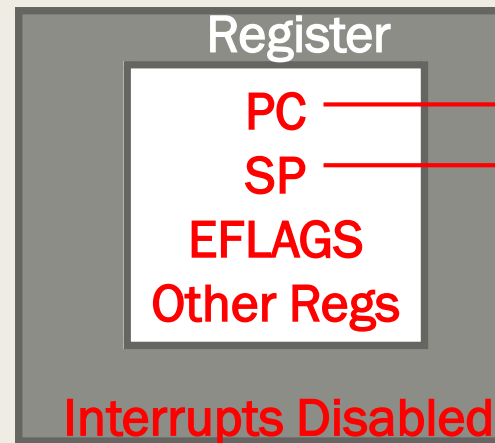
## User-level Process

```
foo () {  
  5000 : x = x + 1 ;  
  5004 : y = y - 2 ;  
}
```

## User Stack



## CPU



## Kernel

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



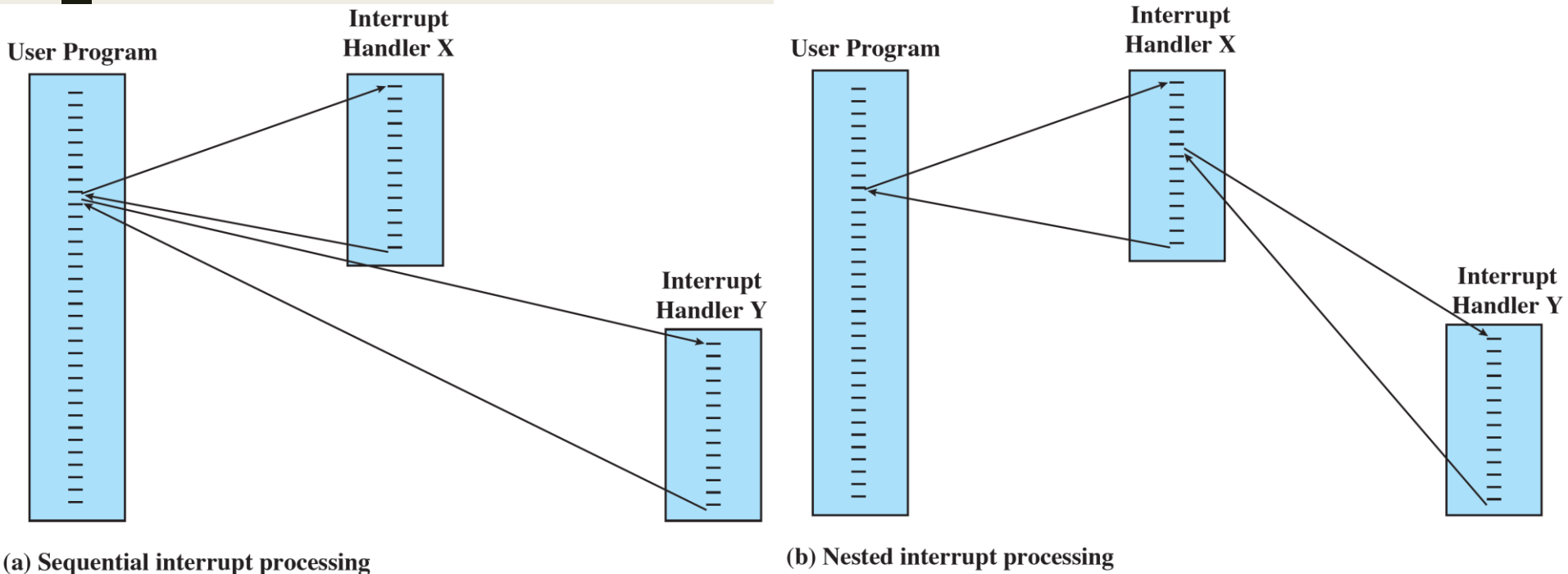
## Kernel Interrupt Stack

# When Handler Ends

- Software does the following:
  - *Handler code restores the saved general registers*
- Hardware does the following:
  - *Restores PC, SP, EFLAGS from Kernel Stack*
  - *Reenable Interrupts*
  - *Switch to user mode*

# Sequential vs Nested Interrupt Handling

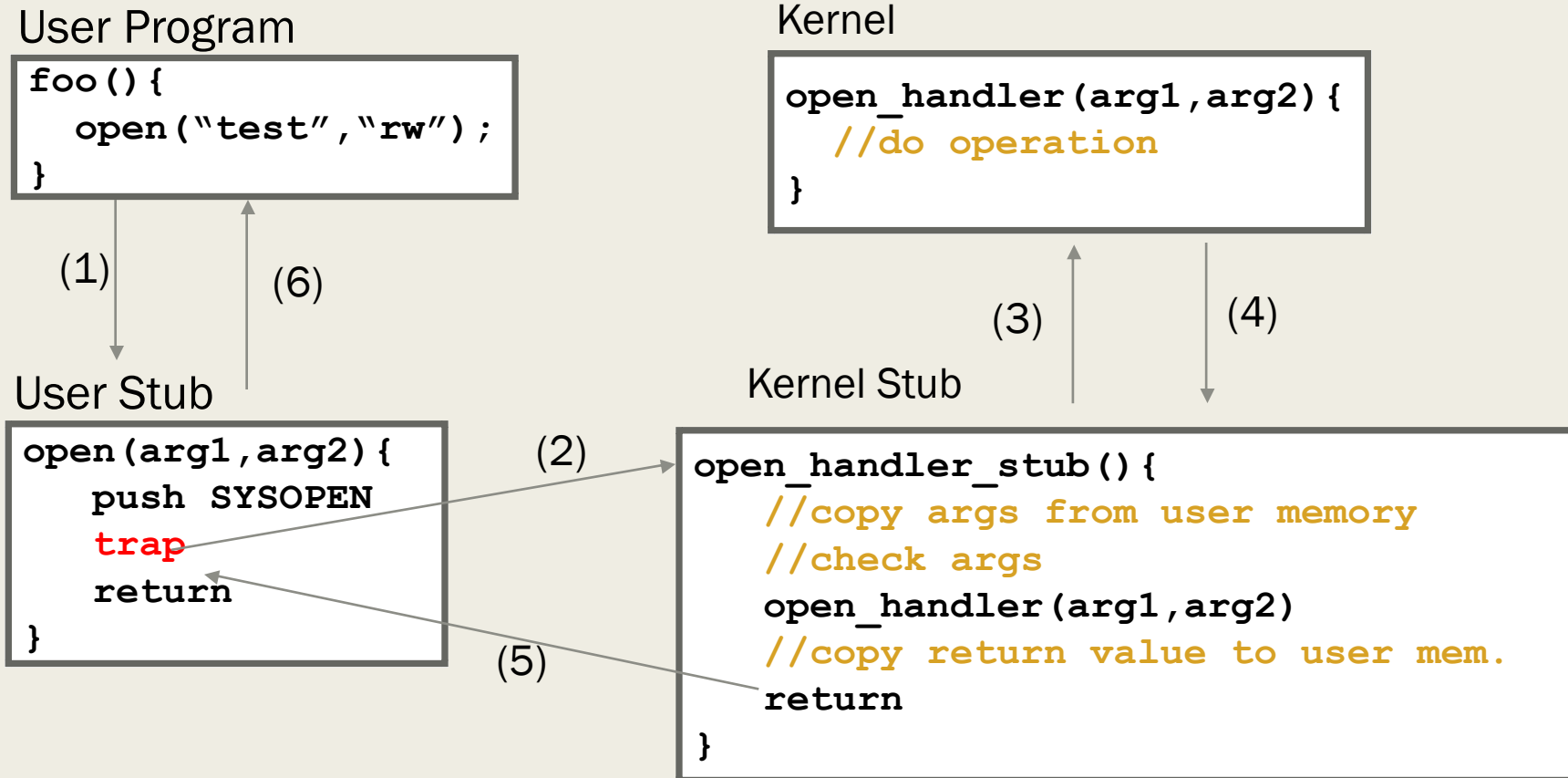
- In x86, other interrupts are disabled to avoid confusion
  - *This keeps things simple, however, no levels of priority among interrupts*
- Therefore, many systems support nested interrupt handler
  - *Another interrupt handler will run if that is higher priority*
  - *Can be implemented with a little bit of more complexity*



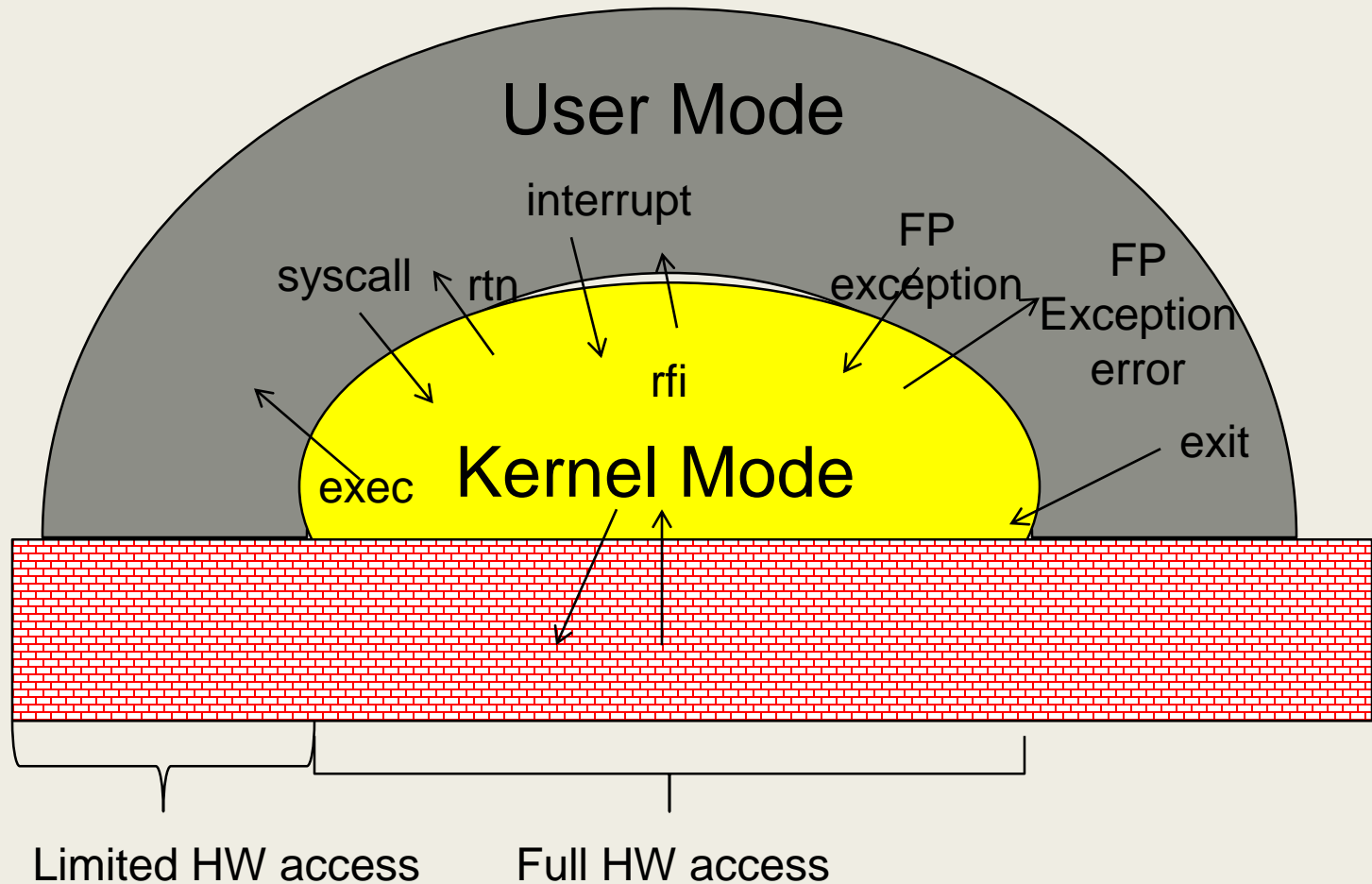
# User-to-Kernel Mode Switch in System Calls

- 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 provided arguments*
- Copy results back
  - *into user memory so that user can use them*
  - *Note, Kernel Stack is not accessible by user*

# System Calls



# Summary: User/Kernel (Privileged) Mode





# Today's Learnings

- Architectural support for user and kernel modes in CPU execution, especially
  - *Privileged Instructions*
  - *Protection*
  - *Timer*
- Interrupt/System Call handling
  - *Most of the path switching User<->Kernel*