Reading Reference: Textbook 1 Chapter 8
Text 2: Direct Execution

EXCEPTIONS & HARDWARE SUPPORT

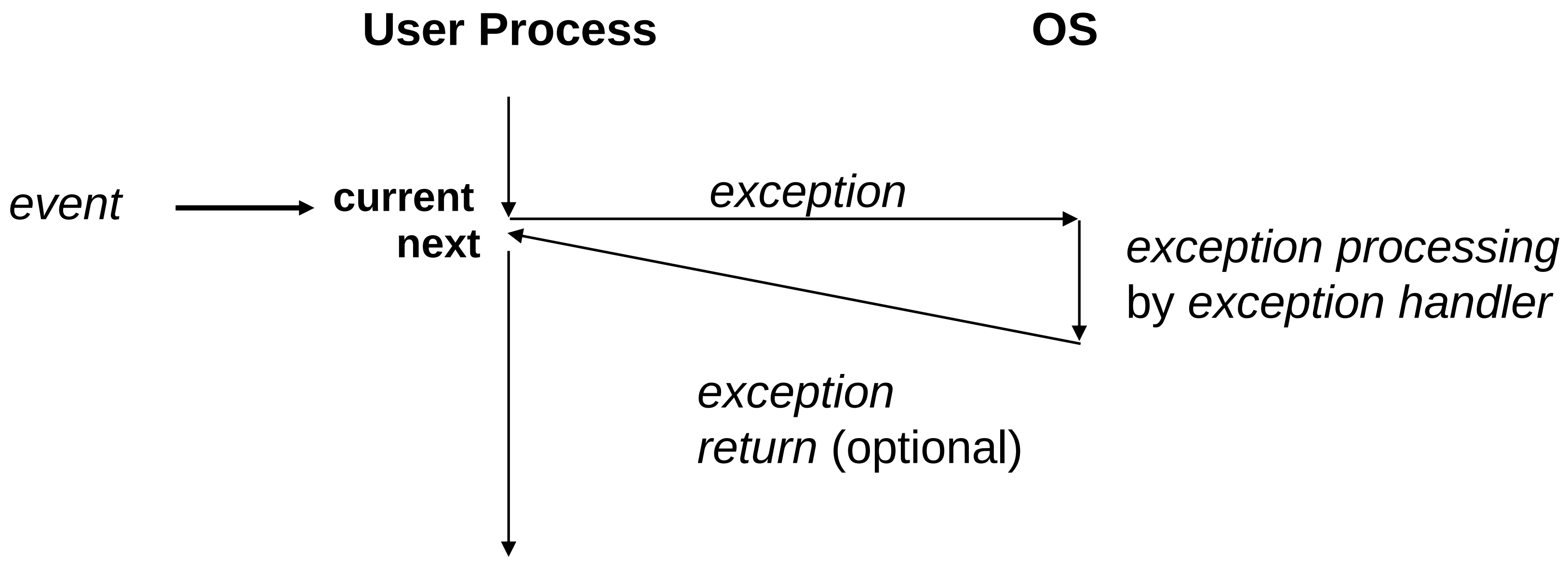
TANZIR AHMED
CSCE 313 Spring 2021

Altering the Control Flow

- CPU executes instructions 1-at-a-time mostly sequentially
- Program-assisted mechanisms for changing control flow:
  - ***Jumps and branches****—react to changes in program state*
  - ***Function call and return*** *using stack discipline—react to program state*
- Insufficient for a useful system
  - *The user application is the central thing – how to let OS into the CPU unless the app gives up control?*
  - *Thus, difficult for the CPU to react to other changes in system state*
    - Data arrives from a network adapter
    - Instruction divides by zero
    - User hits control-C at the keyboard
- System needs mechanisms for “exception control flow”

Exception Control Flow

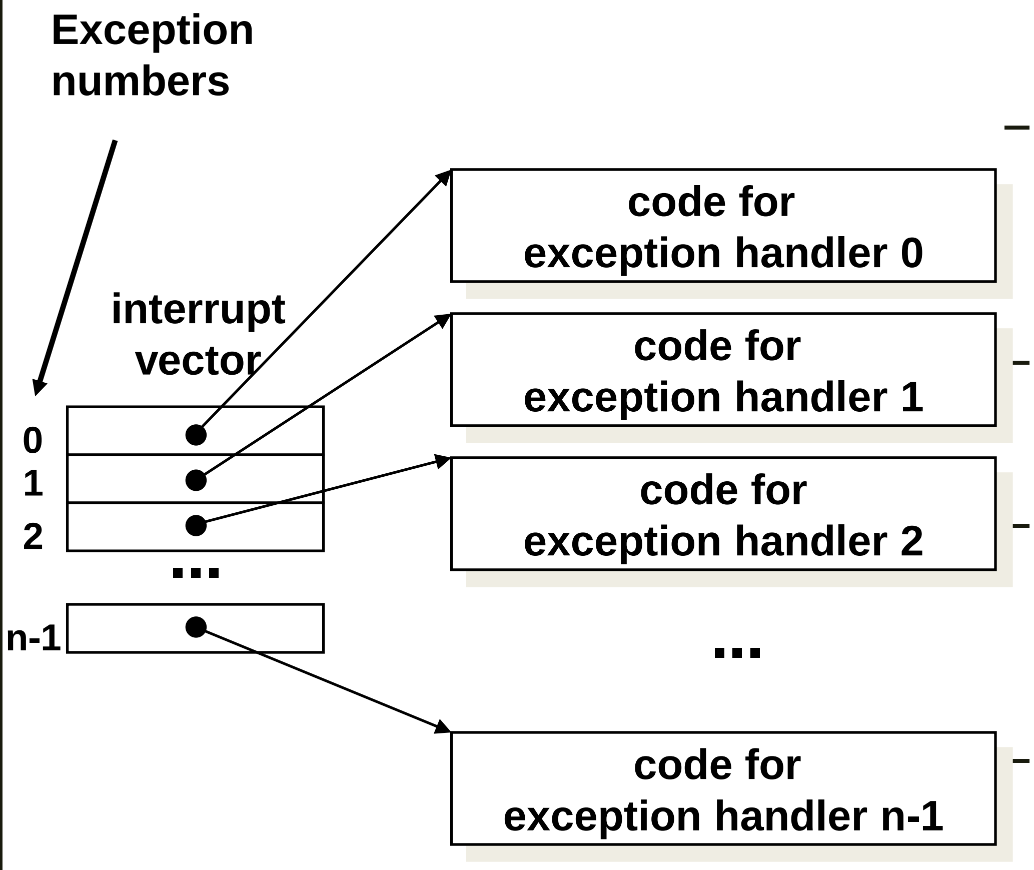
- An *exception* is a transfer of control to the OS in response to some *event* (i.e., change in processor state)
  - *It also gives "asynchronous" behavior*



Asynchronous Exceptions (Interrupts)

- Caused by events **external** to processor (i.e., outside the current program)
  - *Indicated by setting the processor's interrupt pin(s)*
  - *Handler returns to "next" instruction after servicing*
- **Examples:**
  - *I/O interrupts*
    - Key pressed on the keyboard
    - Arrival of packet from network, or disk
  - *Hard-reset interrupt*
    - Hitting reset button
  - *Soft-reset interrupt*
    - Hitting control-alt-delete to initiate restart on a PC

Interrupt Vectors


Exception numbers
interrupt vector
0
1
2
...
n-1
code for exception handler 0
code for exception handler 1
code for exception handler 2
...
code for exception handler n-1

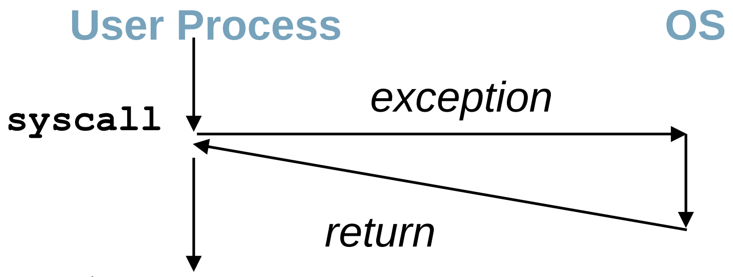
- *Each type of event has a unique exception number k*
- *Index into jump table (a.k.a., interrupt vector)*
- *Jump table entry k points to a function (exception handler).*
- *Handler k is called each time exception k occurs.*

Synchronous Exceptions: Traps, Faults, Aborts

- Caused as result of executing an instruction
  - *From within the CPU*
- 3 types:
  - ***System Calls***
  - ***Faults***
  - ***Aborts***

System Calls

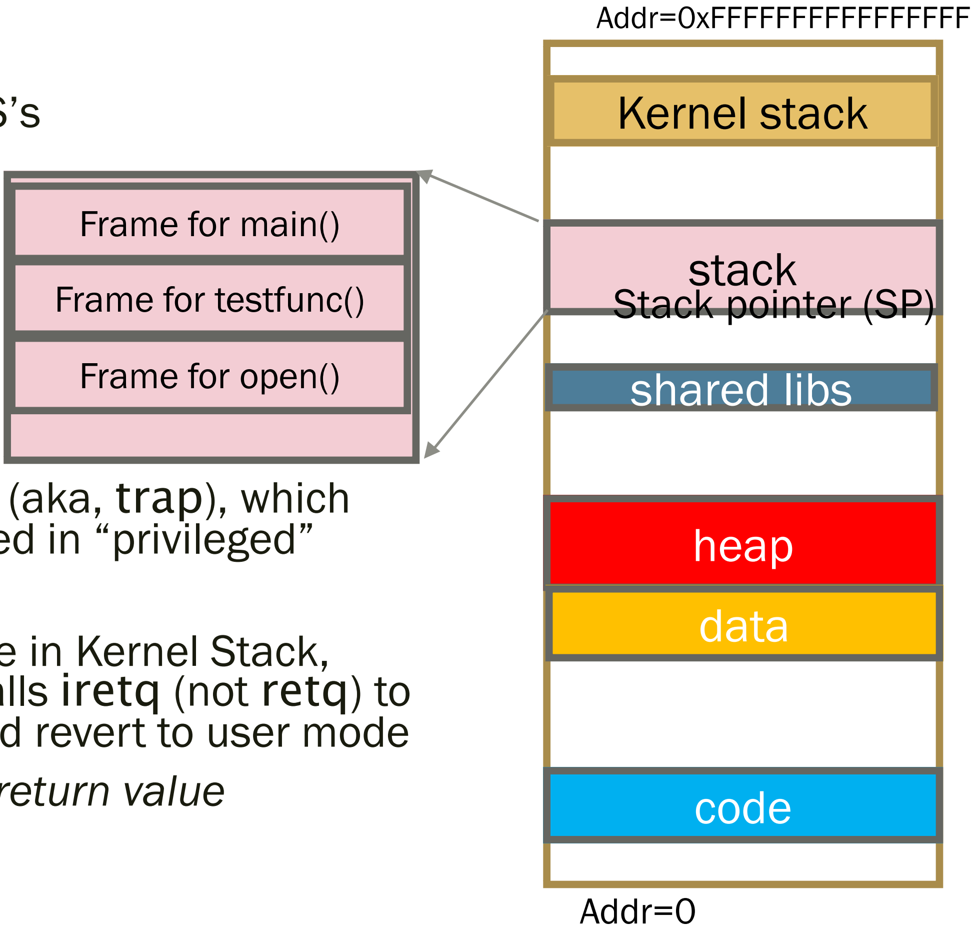
- Functions called by user programs to interact with the OS or request core OS services
- However, unlike traditional functions, system calls must execute in some “privileged” mode so that has access to OS’s internal information and data structures etc.
  - *Hence, these are very different from regular function calls*
  - *Example: Opening a File*
  - *User calls* `open(filename, options)`
  - *OS must find (if reading) or create (when writing) the file, consult permissions and get it ready*



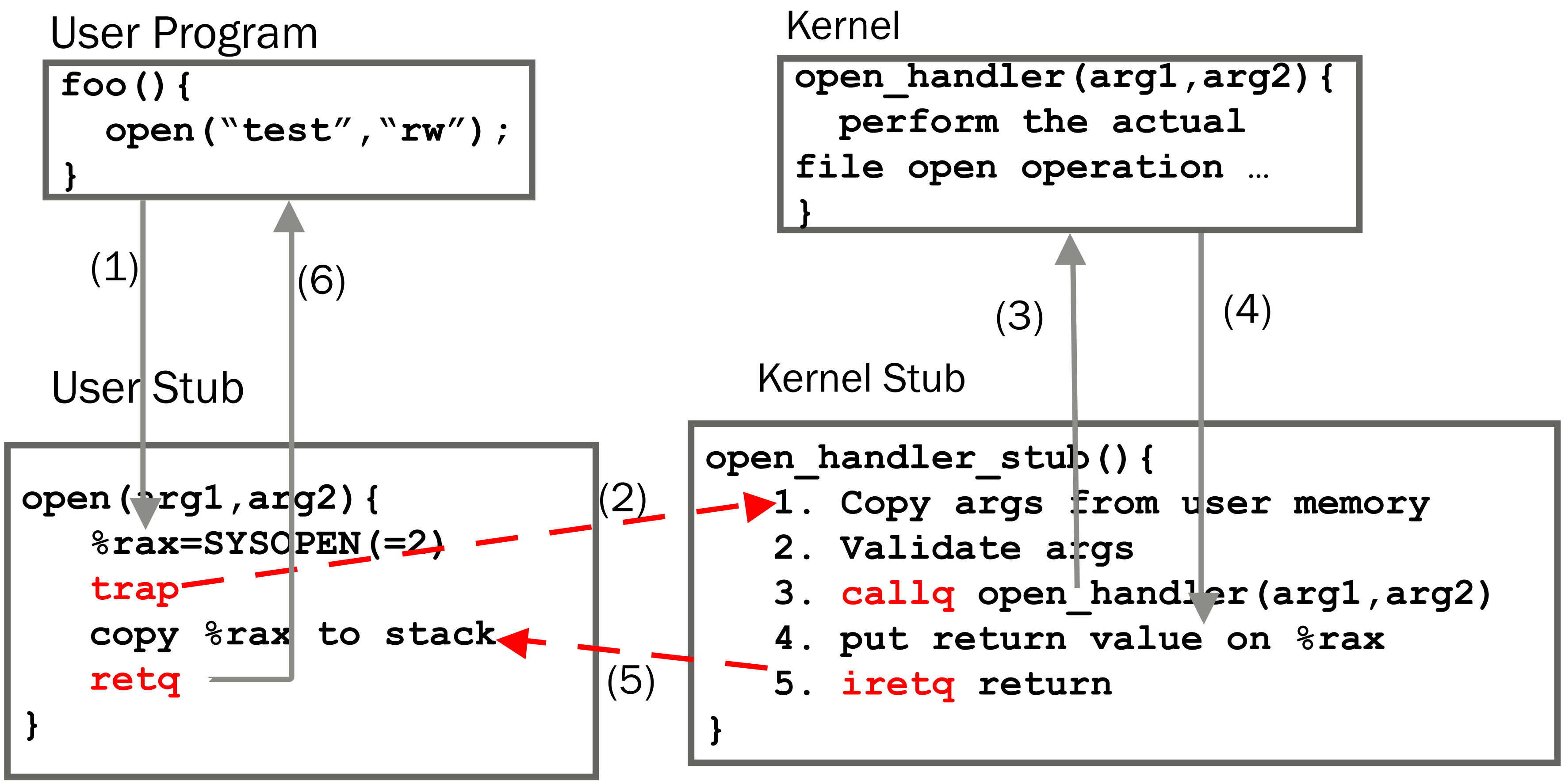
- However, when everything checks out, it must also act like a regular function that returns a file descriptor
- Let us see how a system call is implemented then

Mechanism of System Calls

- We need to take a closer look at the address space layout of program
- The user stack maintains a collection of stack frames, but this is insecure for syscalls because someone could manipulate the SP and navigate all frames
  - *E.g., gdb can do that*
- Therefore, the solution is to keep OS's internal data separately in a place called kernel stack
- The desired system call's opcode is placed in reg %rax (e.g., 0 for read, 1 for write, 2 for open, 3 for close)
- The CPU issues a software interrupt (aka, trap), which invokes an interrupt handler executed in "privileged" mode
- The interrupt handler makes a frame in Kernel Stack, places return value in %rax, then calls iretq (not retq) to return from the interrupt handler and revert to user mode
  - *%rax is then put in place of a return value*



Control Flow in System Calls



```
foo(){
   open("test","rw");
}
```

```
open_handler(arg1,arg2){
   perform the actual
file open operation …
}
```

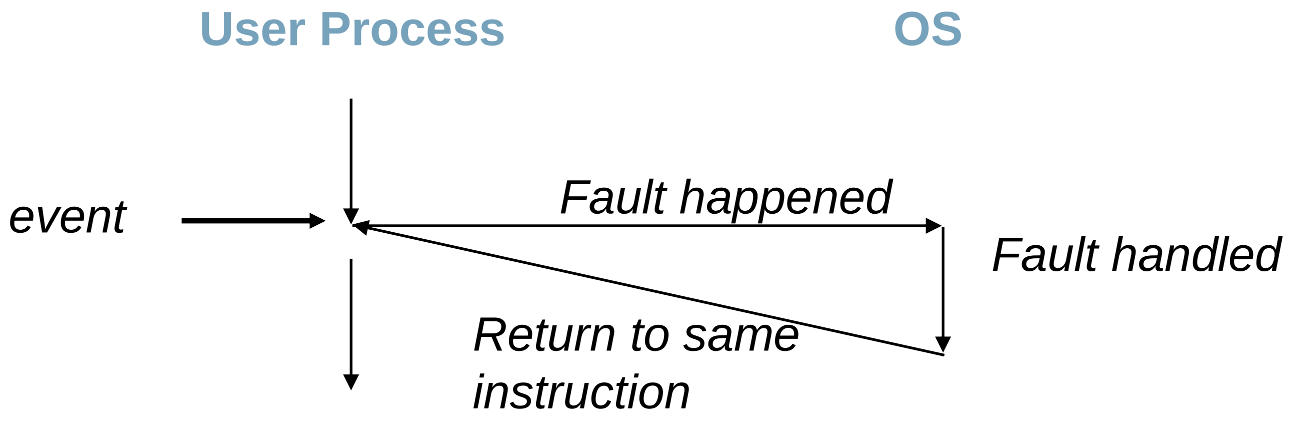
```
open(arg1,arg2){
    %rax=SYSOPEN(=2)
    trap
    copy %rax to stack
    retq
}
```

```
open_handler_stub(){
    1. Copy args from user memory
    2. Validate args
    3. callq open_handler(arg1,arg2)
    4. put return value on %rax
    5. iretq return
}
```

Dashed lines mean context switch due to software interrupt,
This is also where execution mode changes to/from "privileged"

Faults

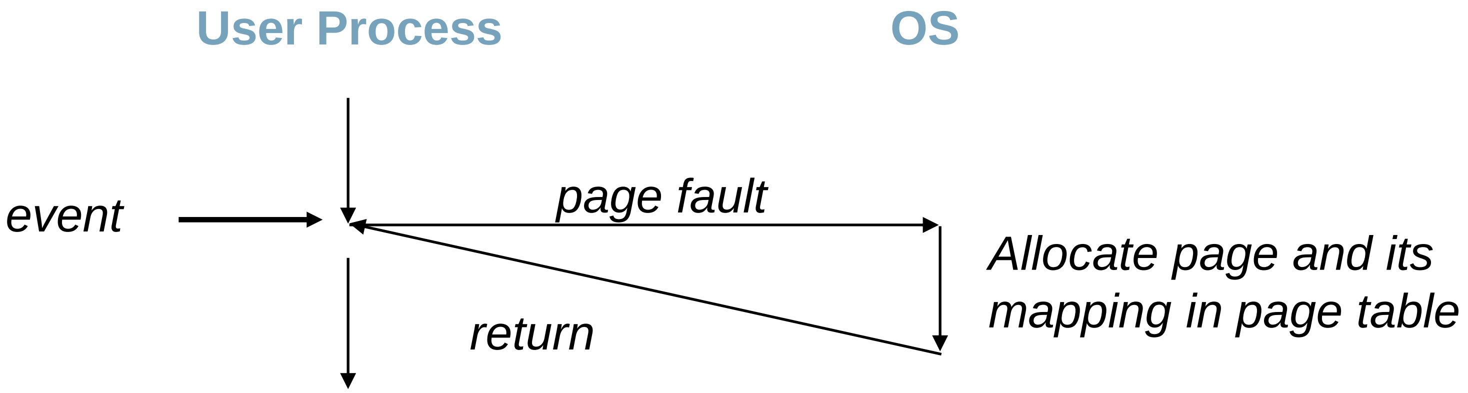
- Attributes
  - *Unintentional but possibly recoverable*
  - *Examples: Page Faults*
  - *Either re-executes faulting ("current") instruction or aborts*



Fault Example #1

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

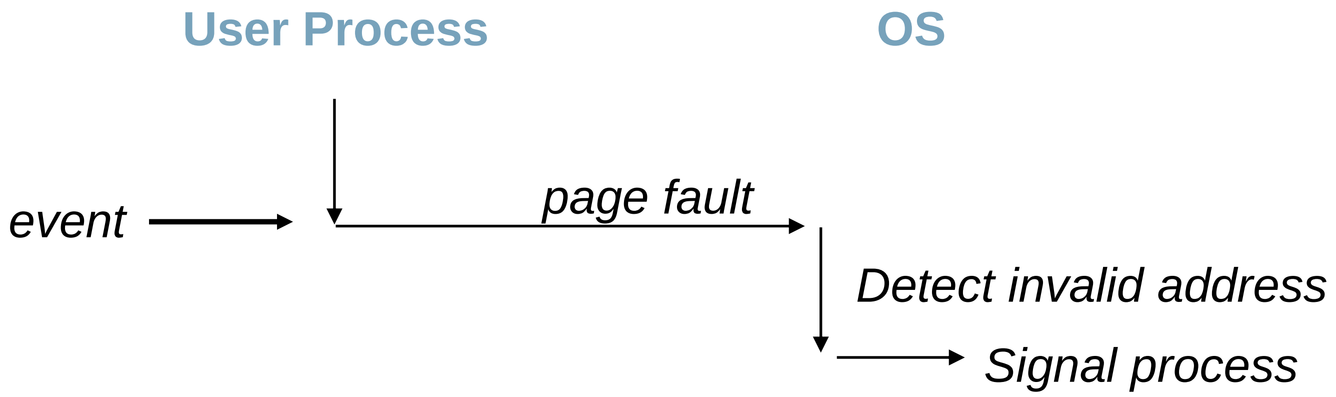
- **Memory Reference**
  - *User writes to memory location*
  - *That portion (page) of user's memory is not mapped yet (because memory pages are mapped only when necessary)*
  - *Page handler must load page into physical memory*
  - *Returns to faulting instruction*
  - *Successful on second try*



Fault Example #2

- **Illegal Memory Reference**
  - *User writes to memory location*
  - *Address is not valid*
  - *Page handler detects invalid address*
  - *Sends* `SIGSEGV` *signal to user process*
  - *User process exits with "segmentation fault"*

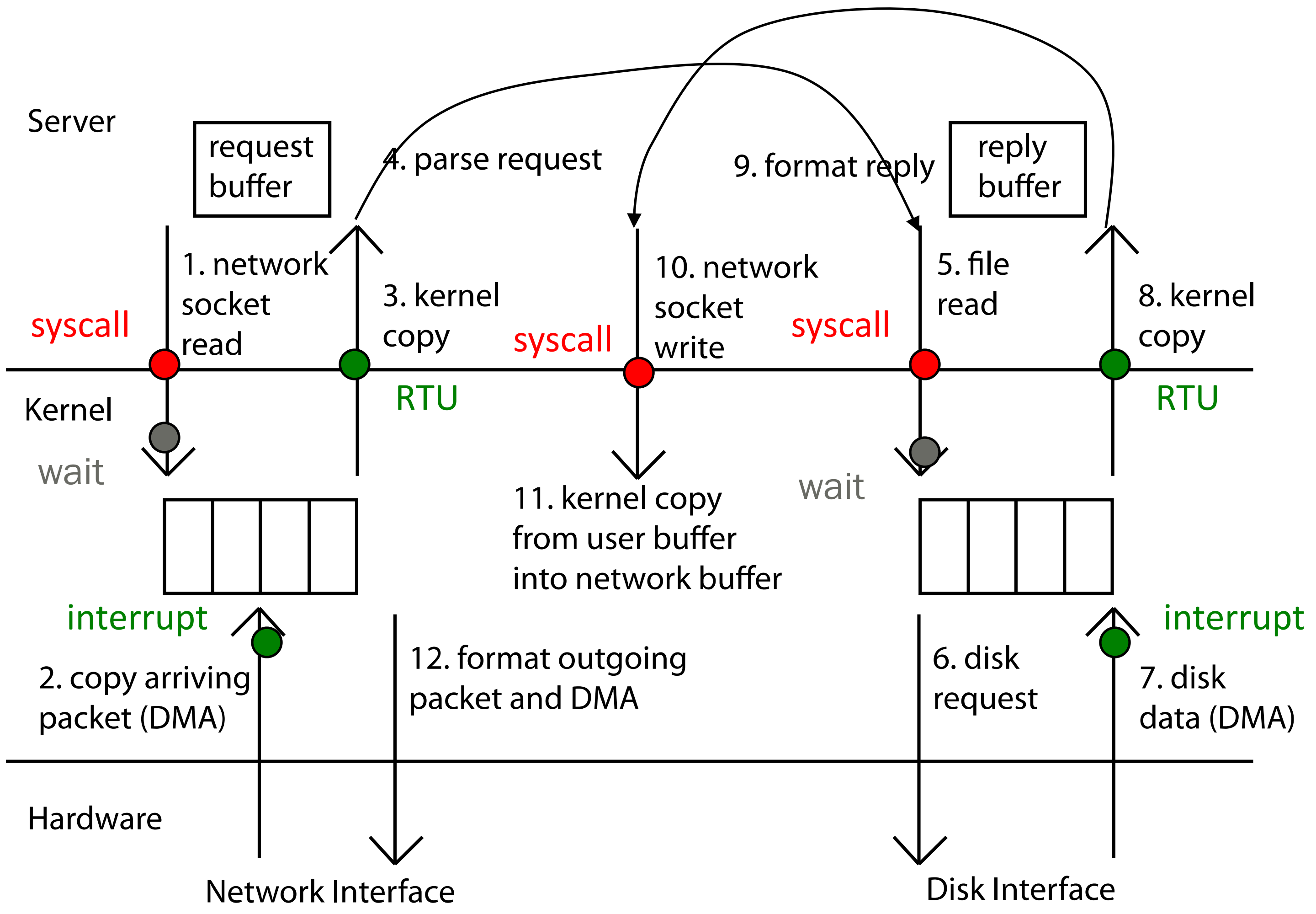
```
int a[1000];
main ()
{
    a[5000] = 13;
}
```



Aborts

- Attributes
  - *Unintentional and unrecoverable*
  - *Examples: parity error, machine check, divide by zero*
  - *Aborts current program and hands control over to the OS*
    - This is the way for the OS to put essential error checking
    - Also, an excellent way to make OS resilient when applications are failing or crashing

Example: Web Server

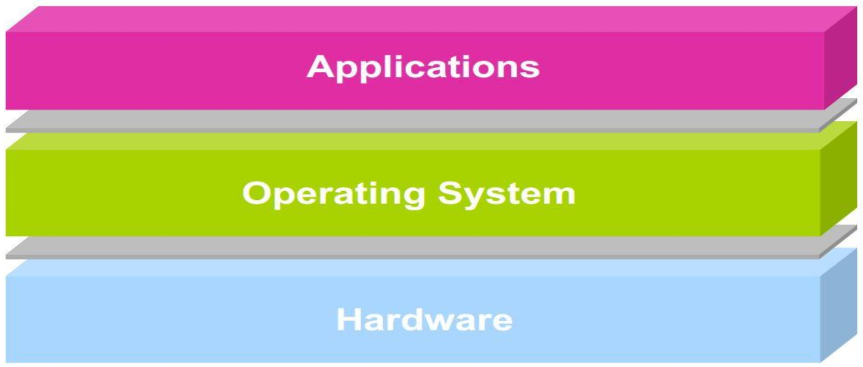


Summarizing Control Flow Exceptions

- User programs are not in charge of (and therefore not burdened with) handling every exception
  - *Not every program will check for divide- by-0 error*
- Mechanism is used by OS to do things beyond error handling
  - *E.g., page faults are used to enable "lazy" physical memory allocation*
- Are Synchronous/Internal (Traps, Faults, Aborts) OR Asynchronous/External (I/O Interrupts, Hard or Soft Reset etc.)

Architectural Support for OS

- A modern with all its features cannot run on a hardware that does not support it
  - *For example, the CPU must be Interrupt-ready if you want to run multiprogramming, time-sharing*

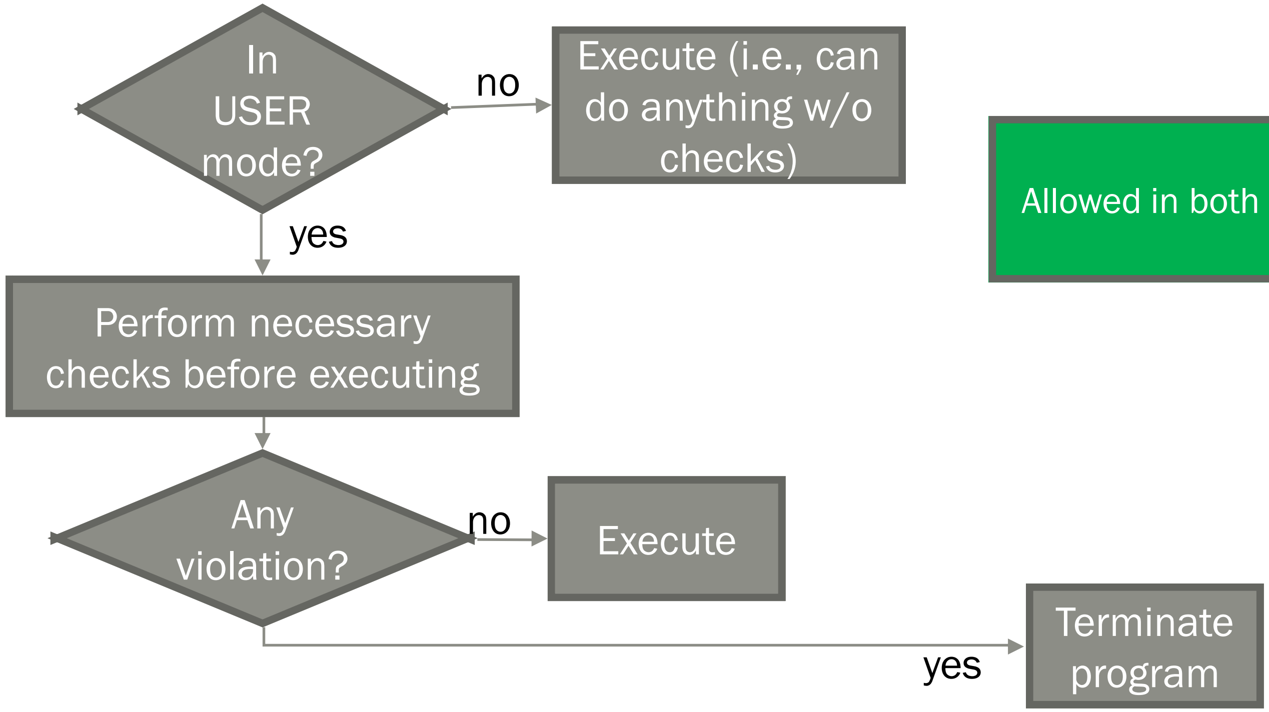


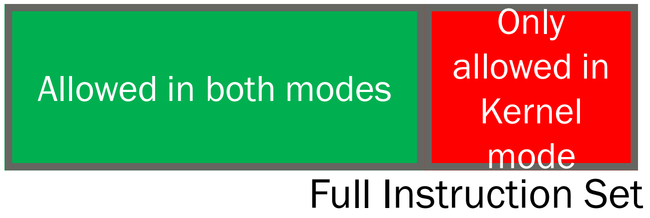
Architectural Features

- 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*
  - ***User-mode:*** *Execution with **Limited privileges***
    - Only those granted by the operating system kernel

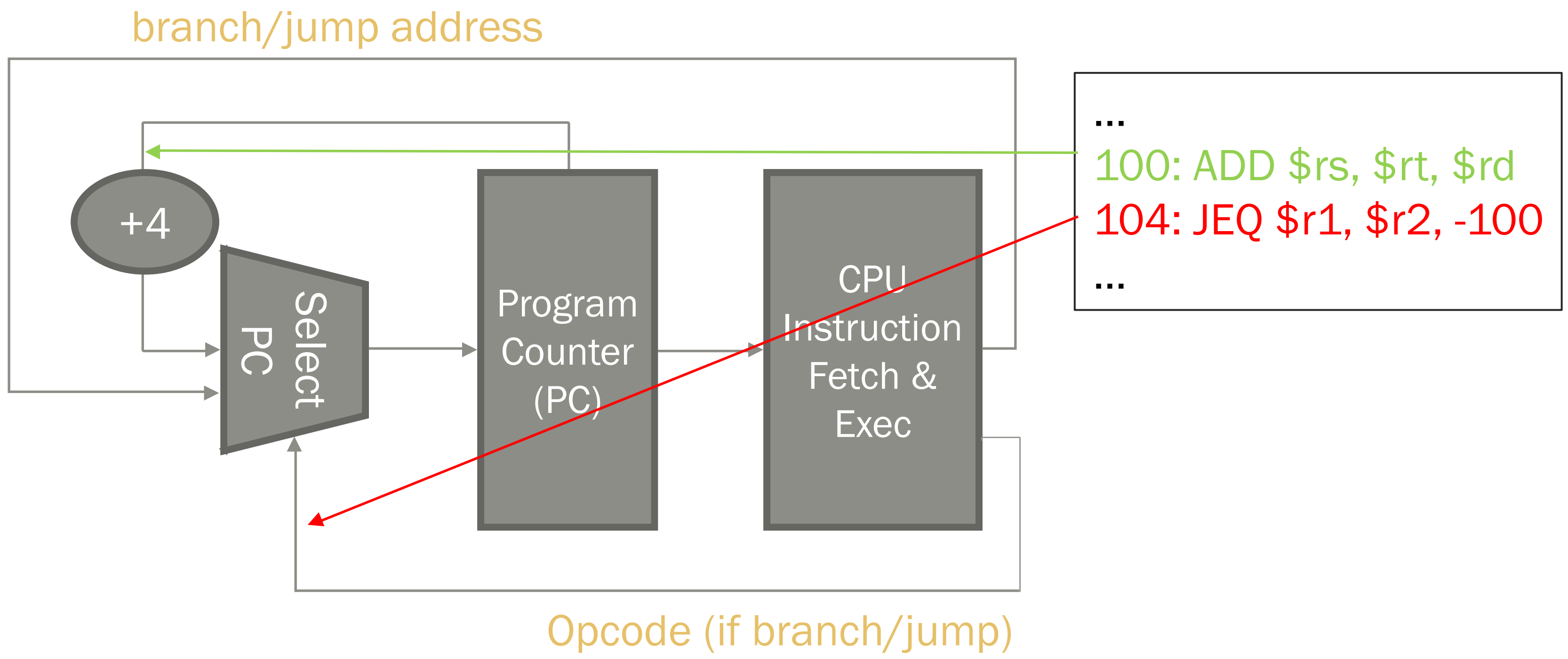

In USER mode?
no
Execute (i.e., can do anything w/o checks)
yes
Perform necessary checks before executing
Any violation?
no
Execute
yes
Terminate program



Privileged Instructions - Examples

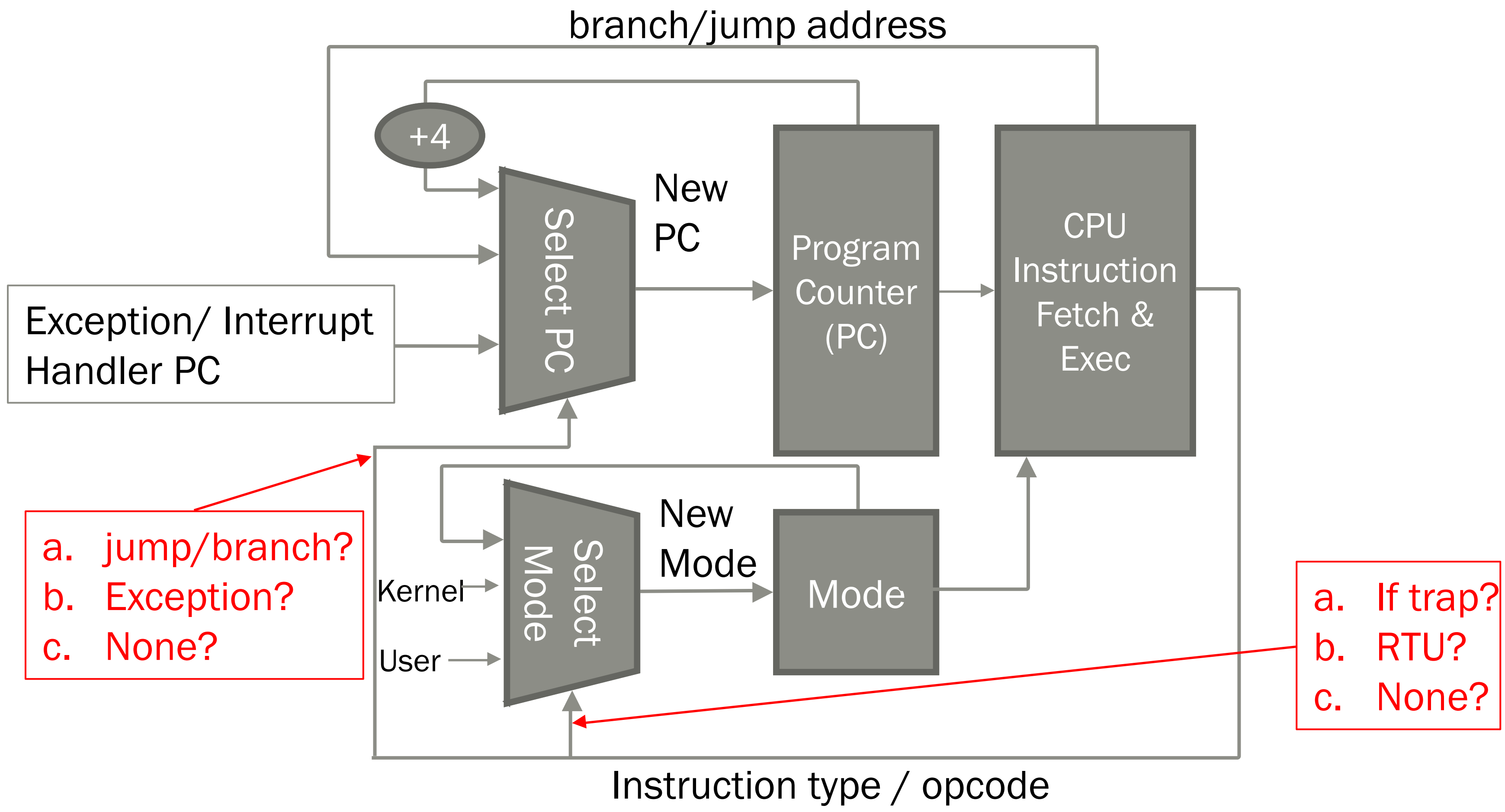
- Only the OS should be able to
  - *Directly access I/O devices (disks, printers..)*
    - Allows OS to enforce security and fairness
    - User programs cannot possibly be fair to each other
  - *Manipulate memory management state*
    - E.g., page tables (Virtual-> Physical), protection bits, TLB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

A Simple CPU Model – Needs Some Change

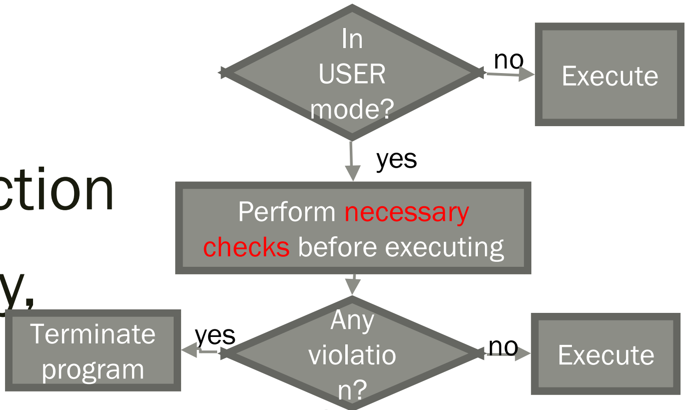


```
…
100: ADD $rs, $rt, $rd
104: JEQ $r1, $r2, -100
…
```

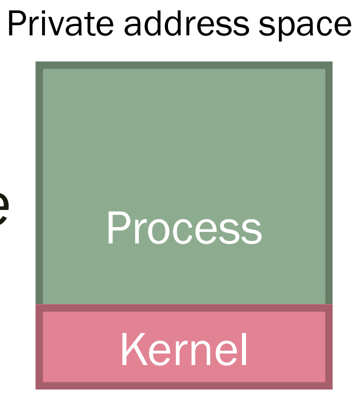
A CPU with Dual-Mode Operation



What are “Necessary” Checks?



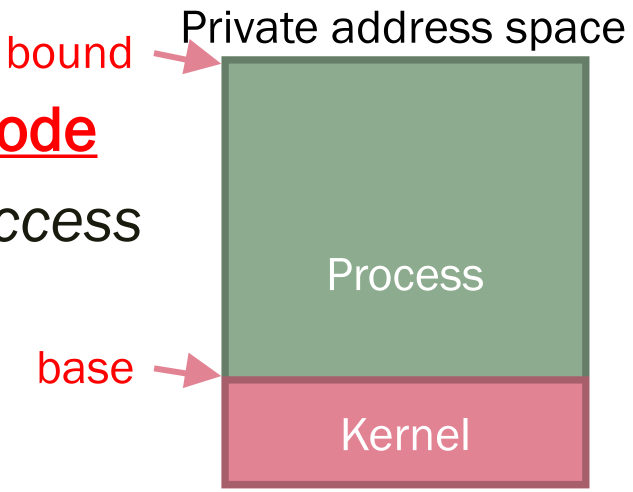
- One example: Kernel Memory Protection
- Note that because of Virtual Memory, no process can harm another
  - *Then, why bother about memory protection?*
- However, the kernel is mapped into lower part of each process’s address space like in the picture:



- Why is Kernel also included?
  - *To load and run the process in the first place*
  - *Handle Interrupts, exceptions, and system calls efficiently w/o address space switch*
- Therefore, kernel must be protected from a faulty or malicious user program

Memory Protection

- 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*



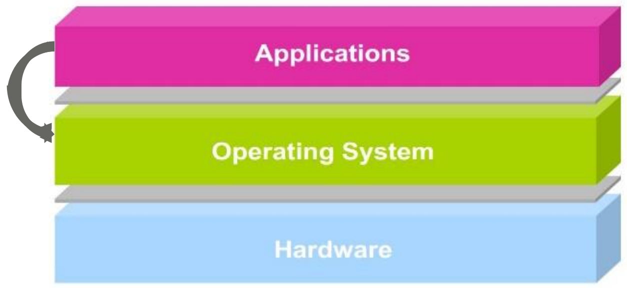
Hardware Timer

- 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*

Mode Switch

- Mode separation does NOT mean that a user program cannot request a Kernel-mode operation
  - *User mode to kernel mode switch is very common*
  - *How do we switch from one mode to the other?*
    - Such that the protection is not compromised

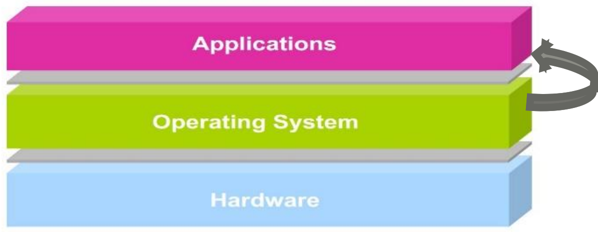
User → Kernel Mode Switch



- **From user-mode to kernel-mode**
  - *Interrupts*
  - *(Synchronous) Exceptions (Faults and Aborts)*
  - *System calls (traps) (aka protected procedure call)*

Kernel → User Mode Switch

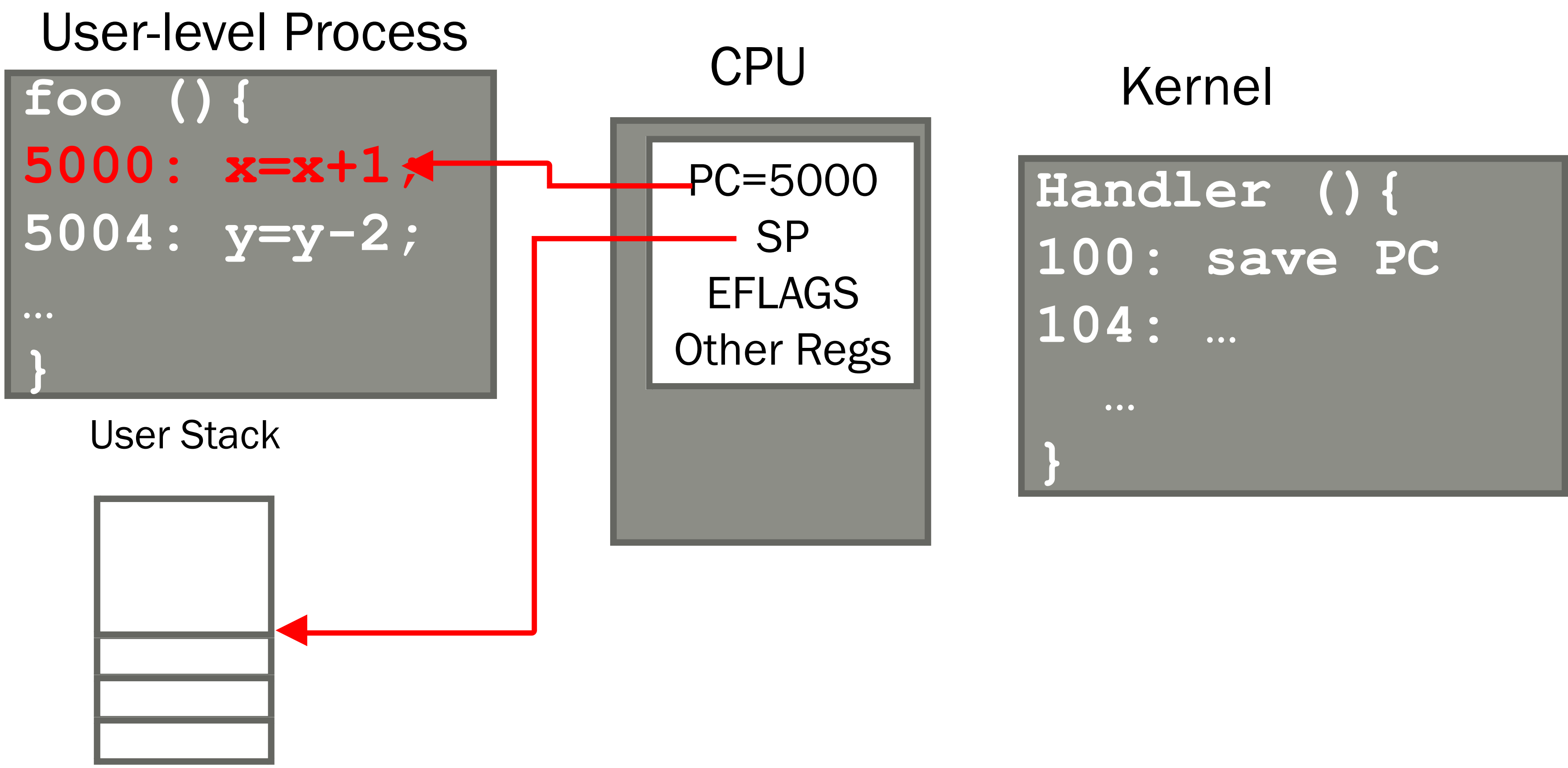
- From kernel-mode to user-mode
  - *New process/new thread start*
    - Jump to first instruction in program/th
  - *Return from interrupt, exception, system call*
    - Resume suspended execution
  - *Process/thread context switch*
    - Resume some other process



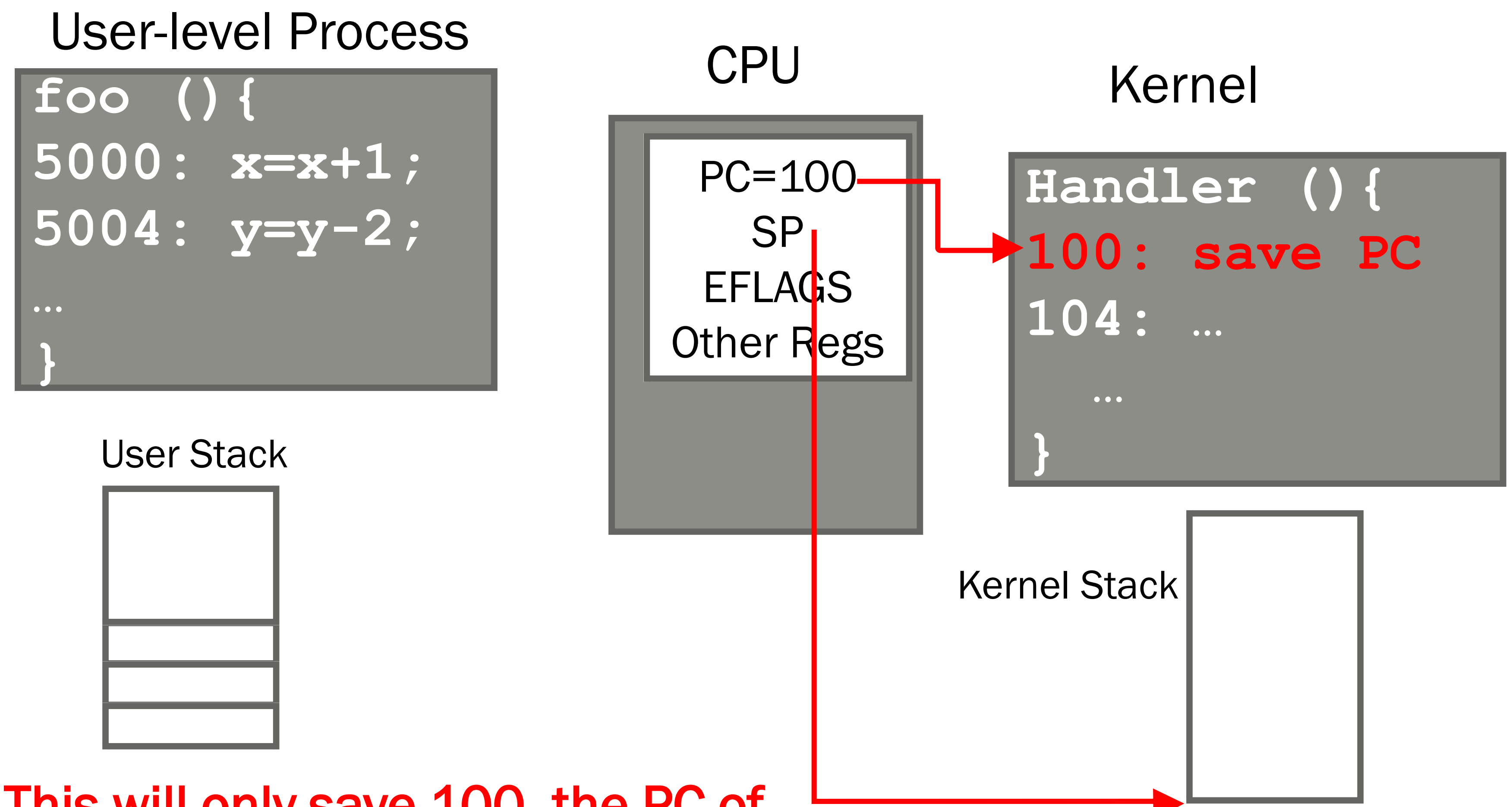
Safe Mode Transfer – Interrupt Handling

- 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 entire program/process state*

Saving Process State: Difficulty



Saving Process State: Difficulty

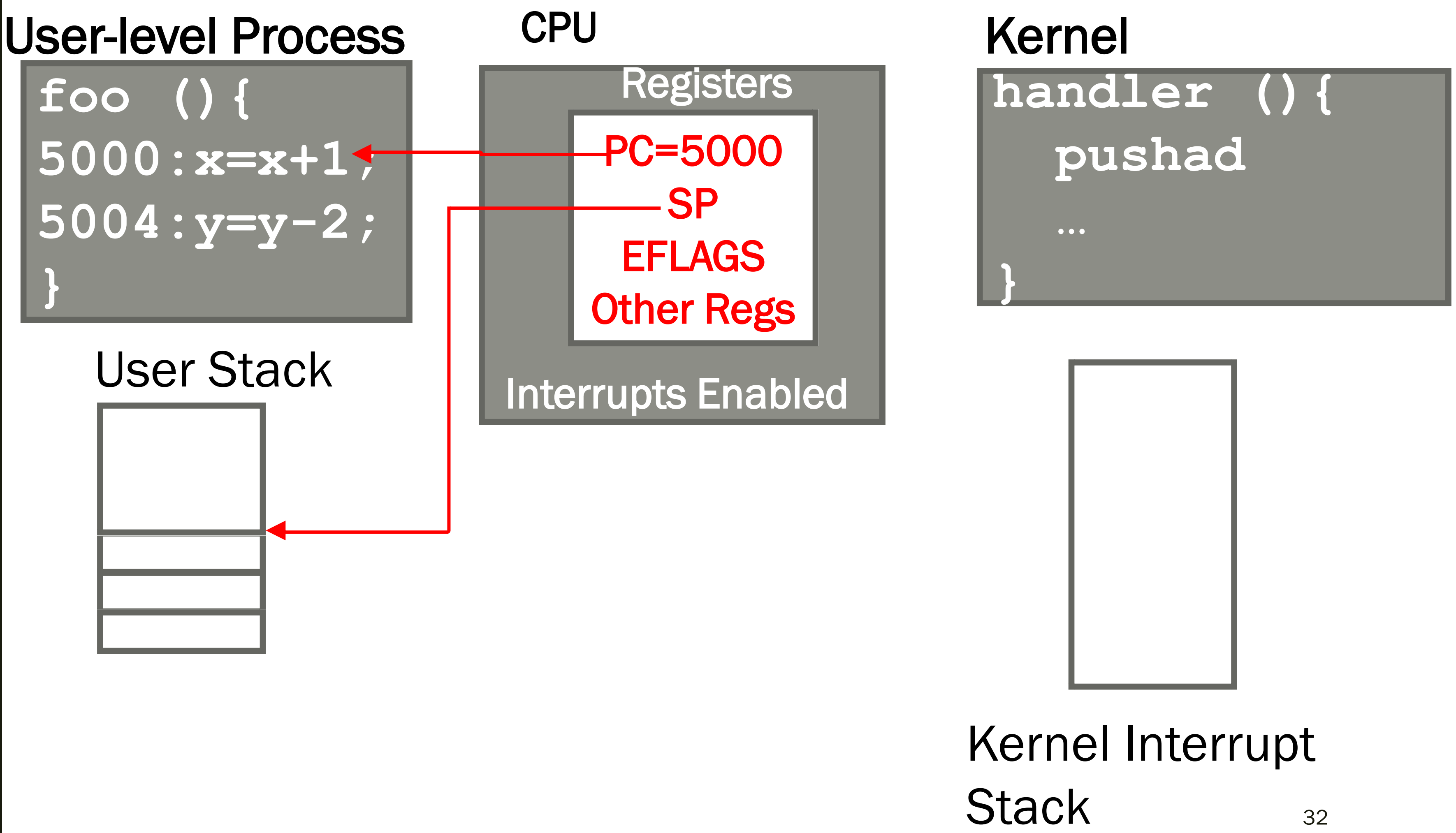


This will only save 100, the PC of the handler. The original PC=5000 of user app is lost forever!!

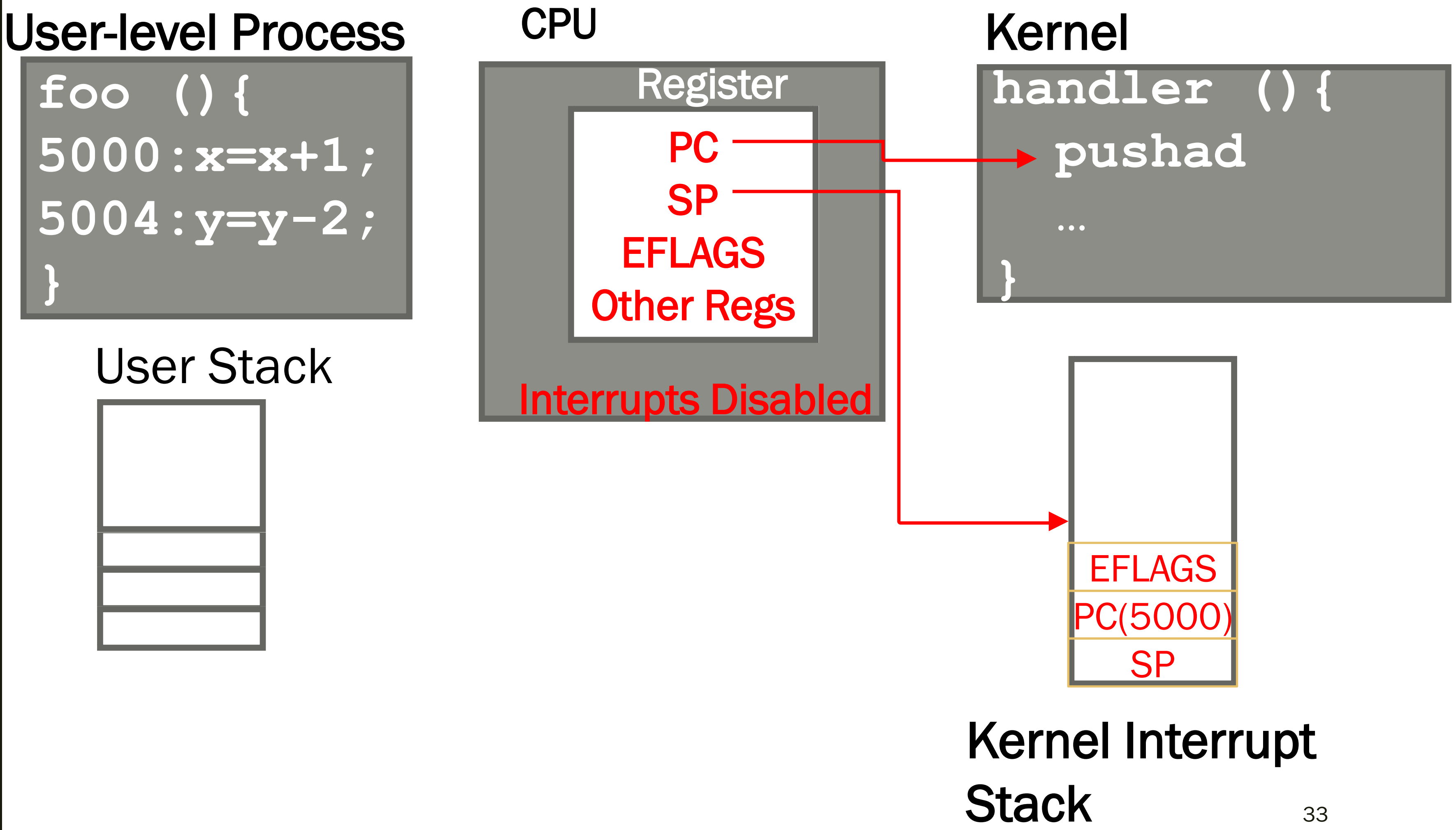
To Summarize

- 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*

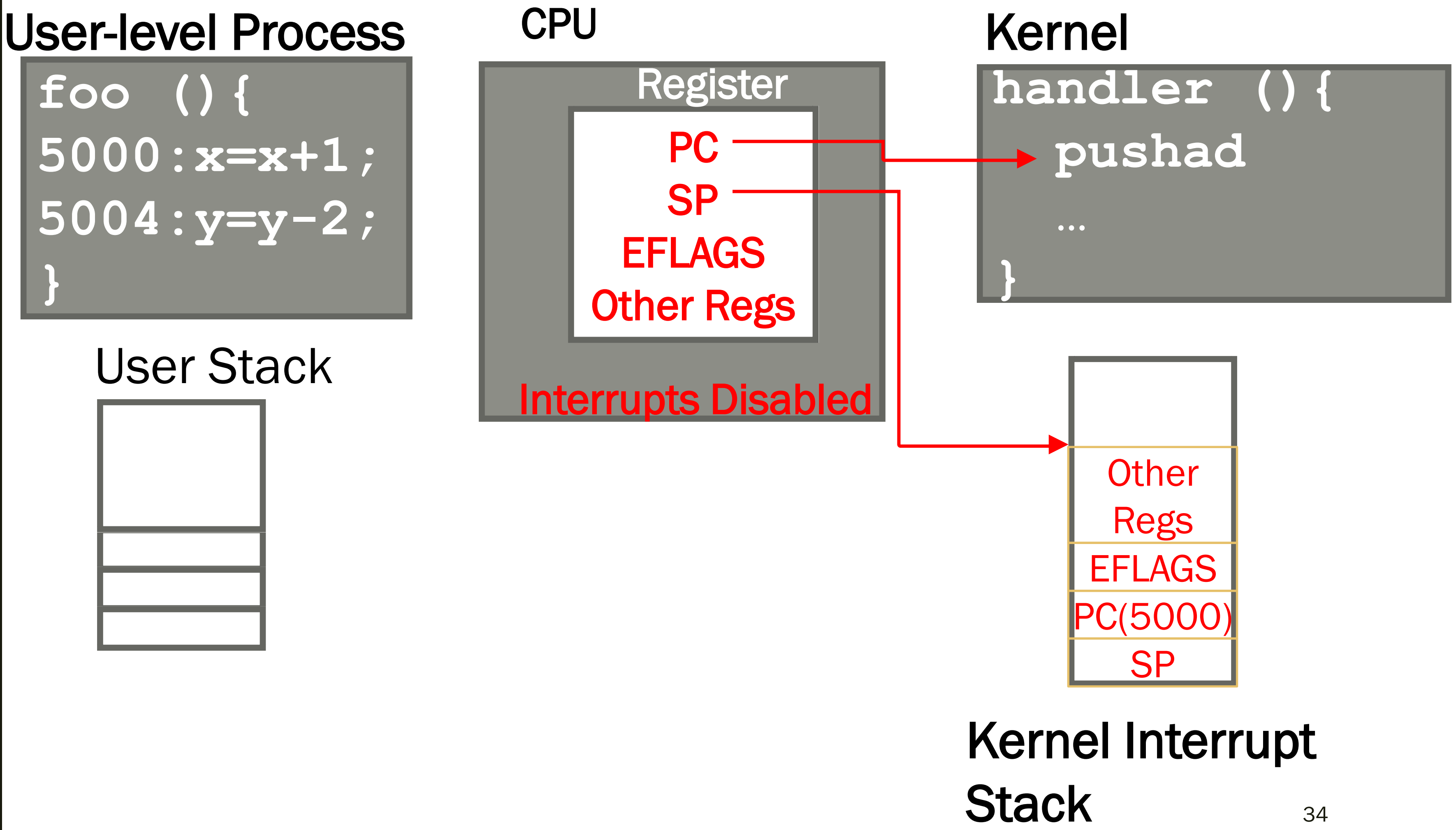
Plan of Action: Before Interrupt



Plan of Action: Hardware Action



Plan of Action: As Handler Starts



User to Interrupt Handler – Mechanism on x86

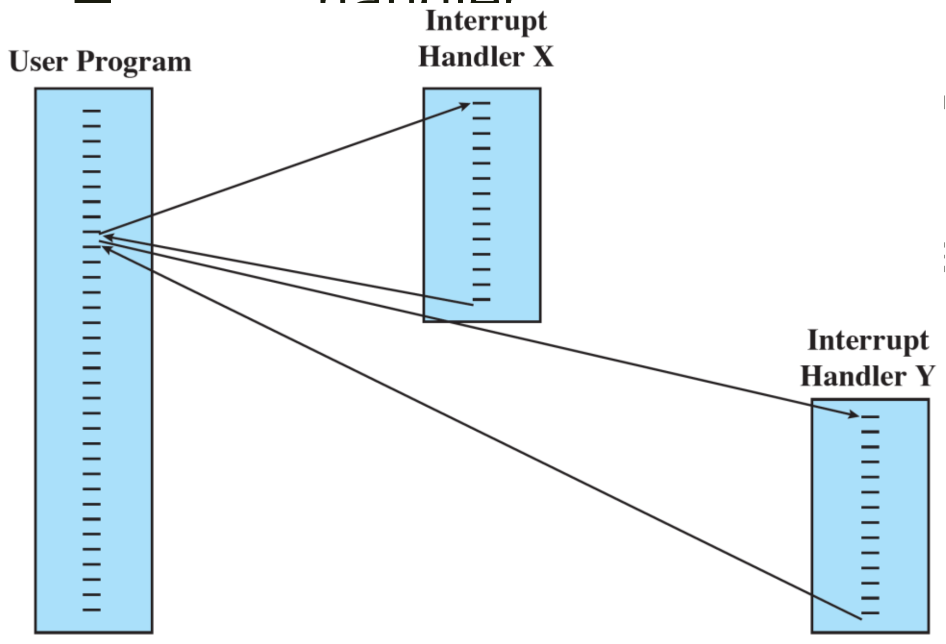
- **Hardware** does the following:
  1. *Mask further interrupts*
     - they are stored, not thrown away
     - If another interrupt arrives in steps 2-5 while the interrupted context is only partially saved, it is lost. Int. Disabled thus provides a "concurrency safety"
  2. *Change mode to Kernel*
  3. *Copy PC, SP, EFLAGS to the* ***Kernel Interrupt Stack*** *(KIS)*
  4. *Change SP: to the KIS (above the stored PC, SP, EFLAGS)*
  5. *Change PC: Invoke the interrupt handler from 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 rest of interrupt handling operation*

When Handler Ends

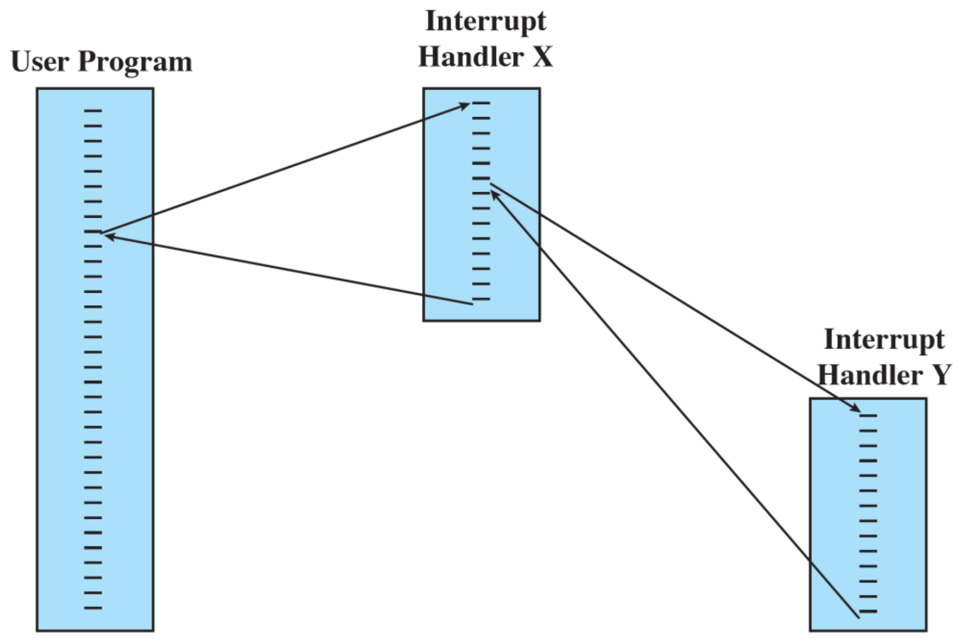
- Software does the following:
  - *Handler code restores the saved general registers*
- Hardware does the following:
  - *Restores PC, SP, EFLAGS from Kernel Stack*
  - *Reenable Interrupts*
  - *Switch to user mode*

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

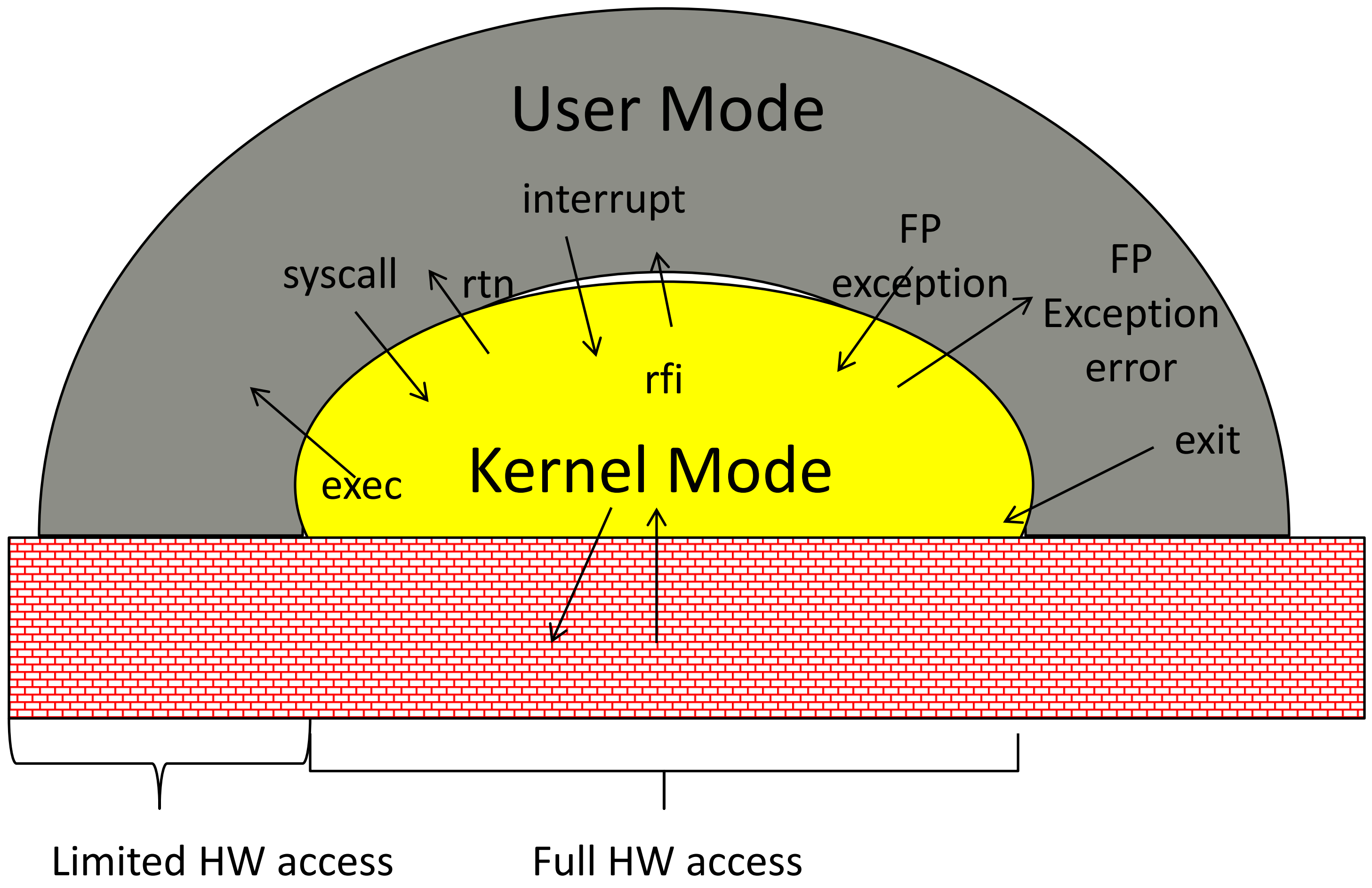


(a) Sequential interrupt processing



(b) Nested interrupt processing

Summary: User/Kernel (Privileged) Mode


User Mode
interrupt
syscall
rtn
FP exception
FP Exception error
rfi
exit
exec
Kernel Mode
Limited HW access
Full HW access