Housekeeping (Lecture 3 - 5/29/2025)

- **PA1 is due at 11:45pm on Tuesday, 6/3/2025**
  - **if you have code from current or a previous semester, *do not look at/copy/share* any code from it**
    - **it's best if you just get rid of it**
  - **get started soon**
    - **if you are stuck, make sure you come to see me during office hours, send me email, or post in the class Piazza Forum**

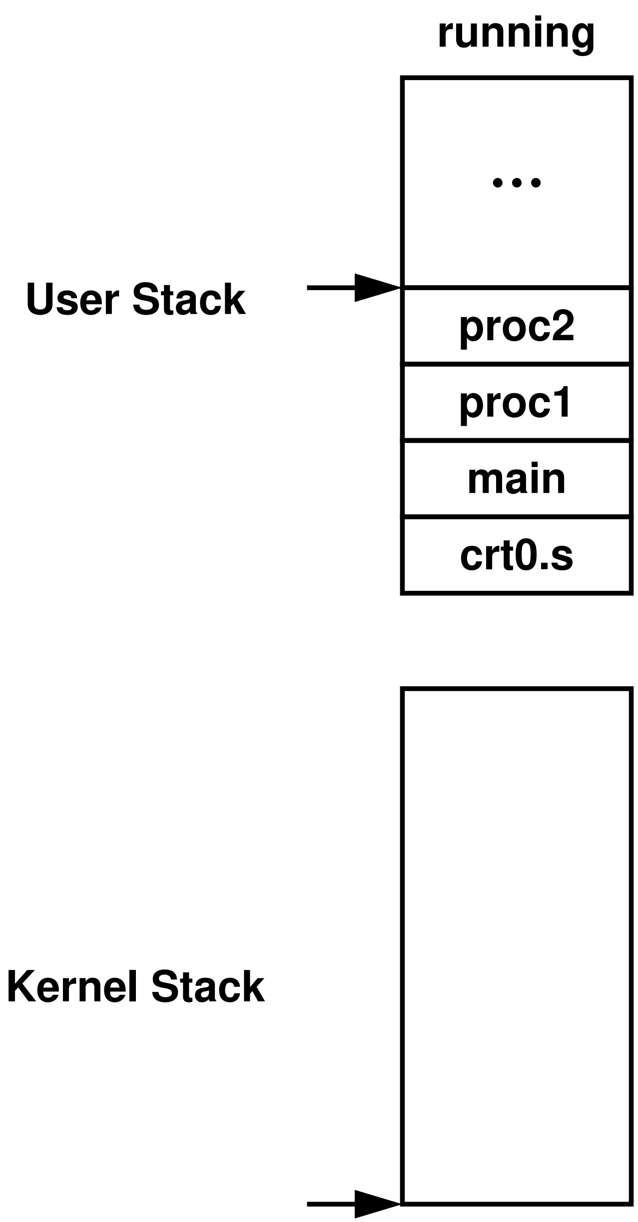
Housekeeping (Lecture 3 - 5/29/2025)

- *Grading guidelines* is the *ONLY* way we will grade and we can only grade on a *standard 32-bit Ubuntu Linux 16.04 inside VirtualBox/UTM* or on AWS Free Tier
  - due to our *fairness* policy
  - the *grading guidelines* is *part of the spec*
  - although not recommended, you can do your development on a different platform
    - you must test your code on the "standard" platform because those are the *only* platforms the grader is allowed to grade on
  - if you make submission, make sure you run through the *Verify Your Submission* procedure as if you are the grader

Two Stacks Per Thread

- **Most OS allocate a kernel interrupt stack for every user-level thread**
  - **when a user-level thread is running, the hardware interrupt stack points to that thread's kernel stack and the kernel stack is empty**
    - **we refer to the interrupt stack simply as the *"kernel stack"***

Two Stacks Per Process

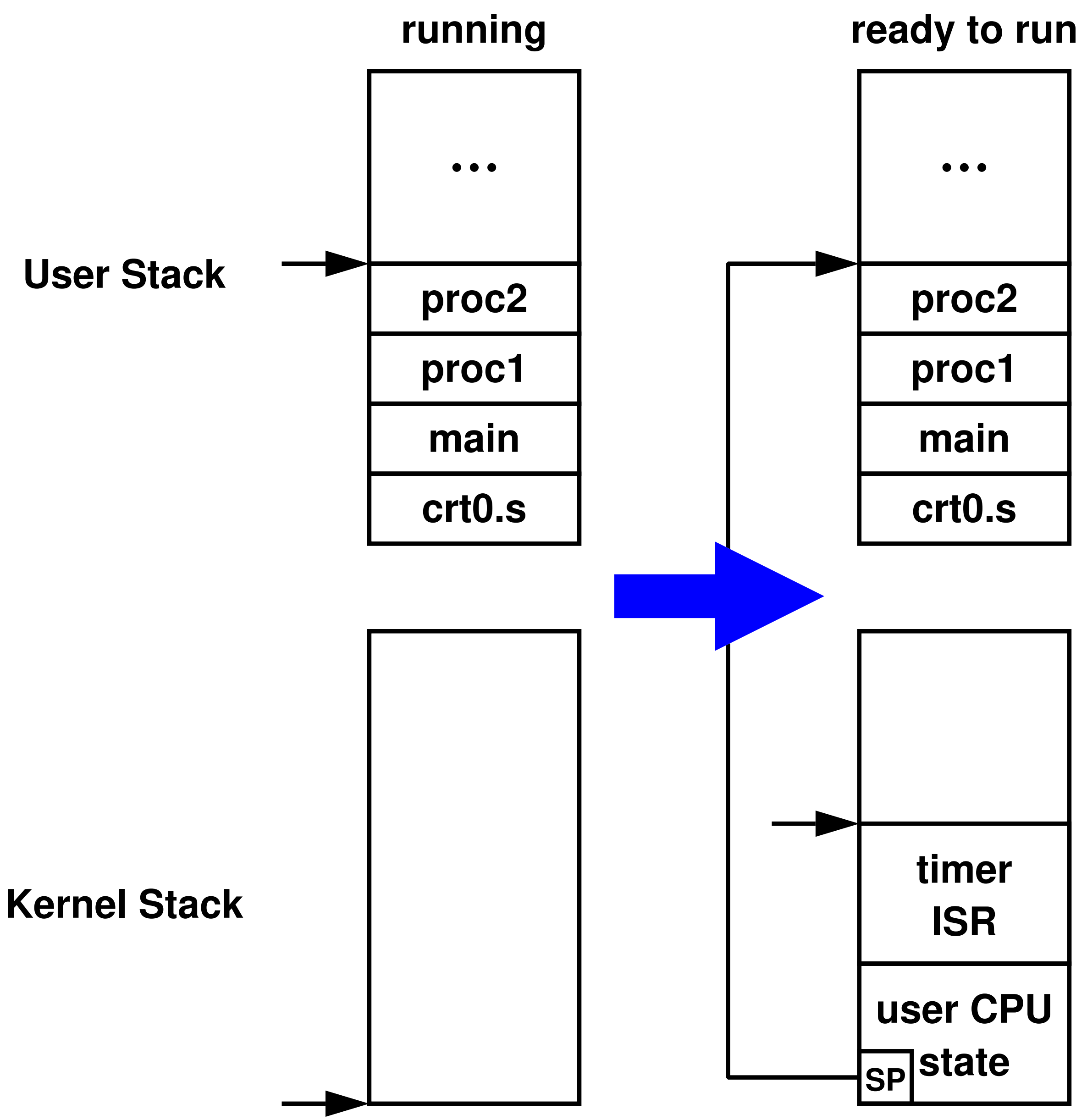


- `main()` is called by `crt0.s`
  - some would call `crt0.s` the ***startup function***
  - the code for `crt0.s` is simply:

    ```
    exit(main());
    ```

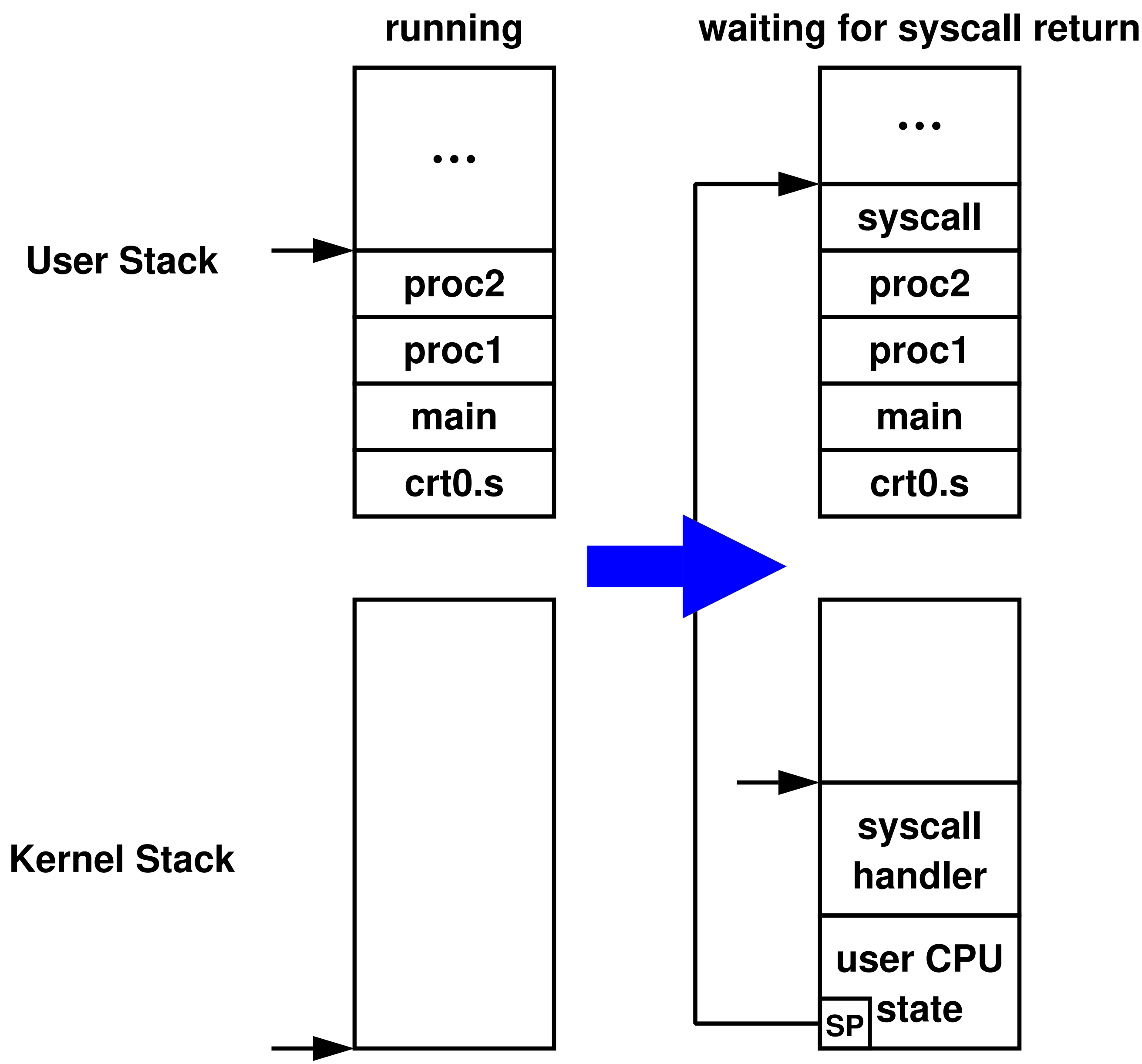
  - if `main()` returns `N`, then the main thread would call `exit(N)` to terminate the process
  - you don't see `crt0.s` because it's written for you already

Two Stacks Per Process



- in this example, the user thread was suspended due to timer expiration, (i.e., hardware interrupt)

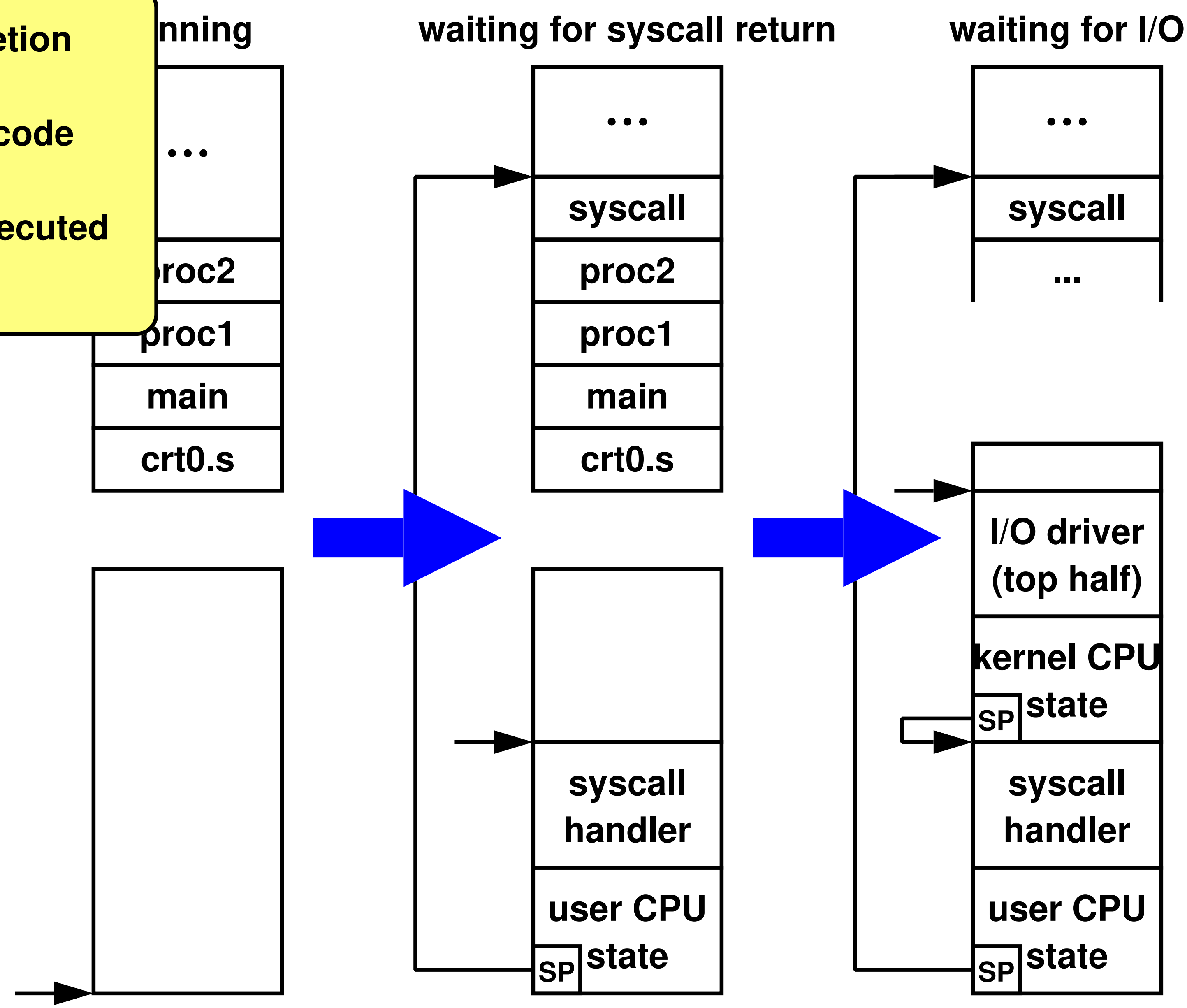
Two Stacks Per Process



- in this example, the user thread made a system call and the system call started an I/O operation

Two Stacks Per Process

- handling I/O completion interrupt
- "top half" refers to code the kernel executes ("bottom half" is executed by hardware)



Kernel Stack

Interrupt Masking

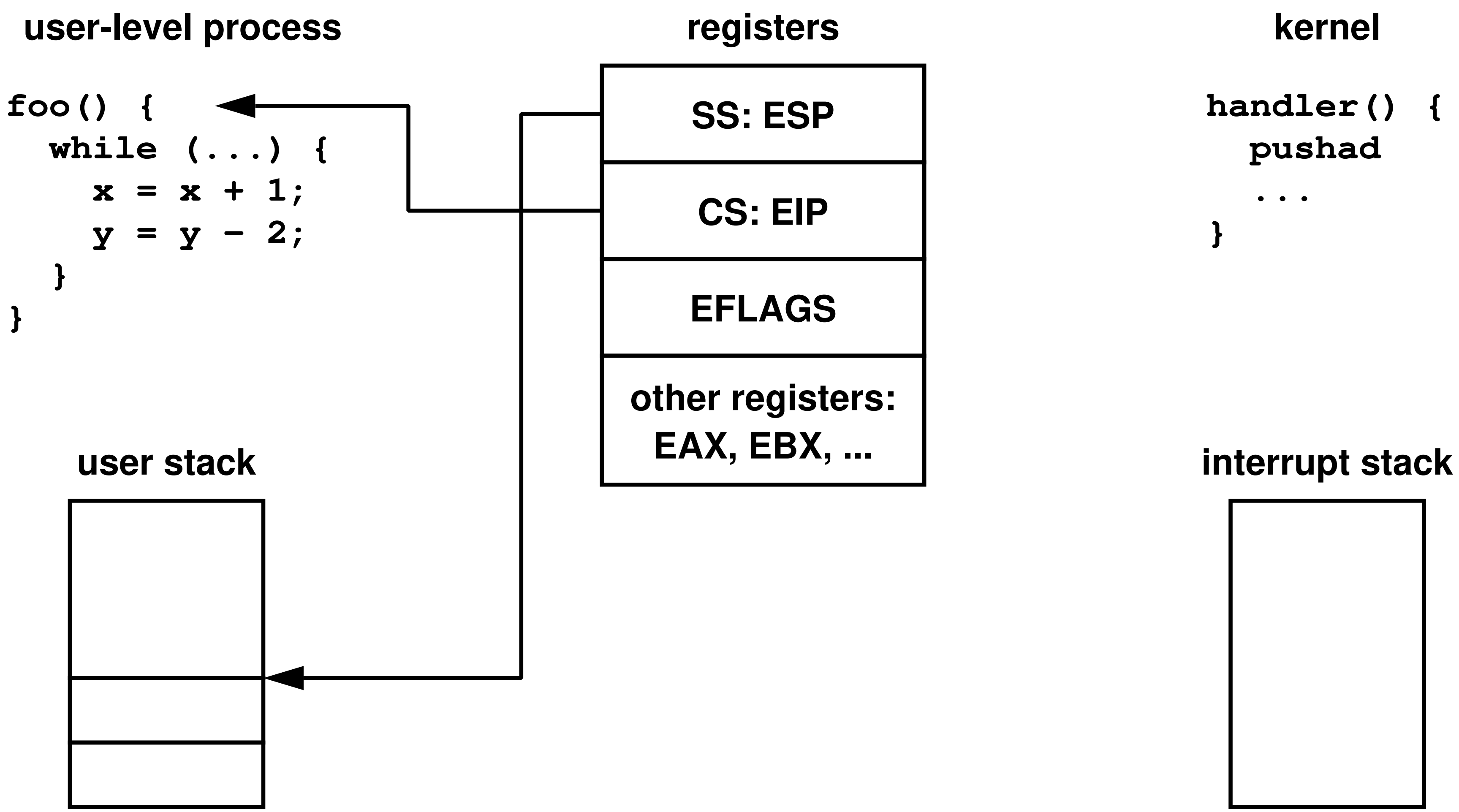
- **Interrupt handler runs with interrupts disabled**
  - **re-enabled when interrupt completes**
- **OS kernel can also turn interrupts off**
  - **e.g., when determining the next process/thread to run**
  - **on x86**
    - **CLI: disable interrrupts**
    - **STI: enable interrrupts**
    - **only applies to the current CPU (on a multicore)**
- **We will need this to implement synchronization in Ch 5**
- **If an interrupt is generated when interrupt is disabled, the new interrupt becomes *pending* (and deferred, but not lost)**
  - **when interrupt is enabled, all pending interrupt will be *delivered* in a certain sequence**
  - **usually, the hardware will buffer one interrupt of each type**
    - **interrupt handler needs to check with the device to see if multiple interrupts of the same type has occurred**

(2.5) Putting It All Together: x86 Mode Transfer

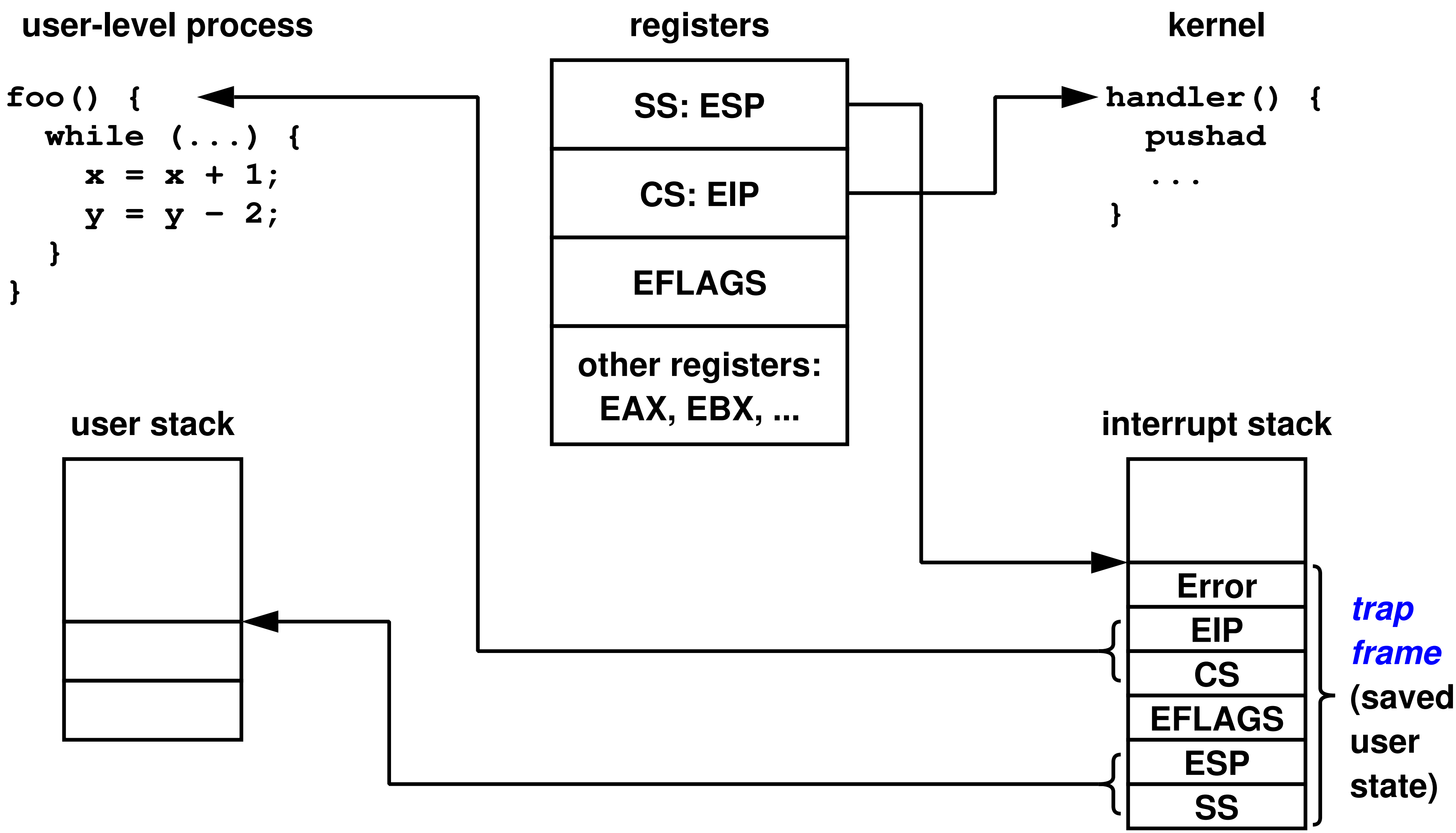
x86 Mode Transfer

- **Case Study: x86 Interrupt**
  1) mask interrupts and switch to kernel mode
  2) save current stack pointer, program counter, and Processor Status Word (condition codes) to internal registers
  3) the hardware switchs to interrupt/kernel stack (information stored in a special hardware register)
  4) pushd saved SP, PC, PSW from internal registers on to stack
  5) save error code that caused the interrupt)
  6) invoke the interrupt handler
     - vector through interrupt table
     - interrupt handler saves registers it might clobber

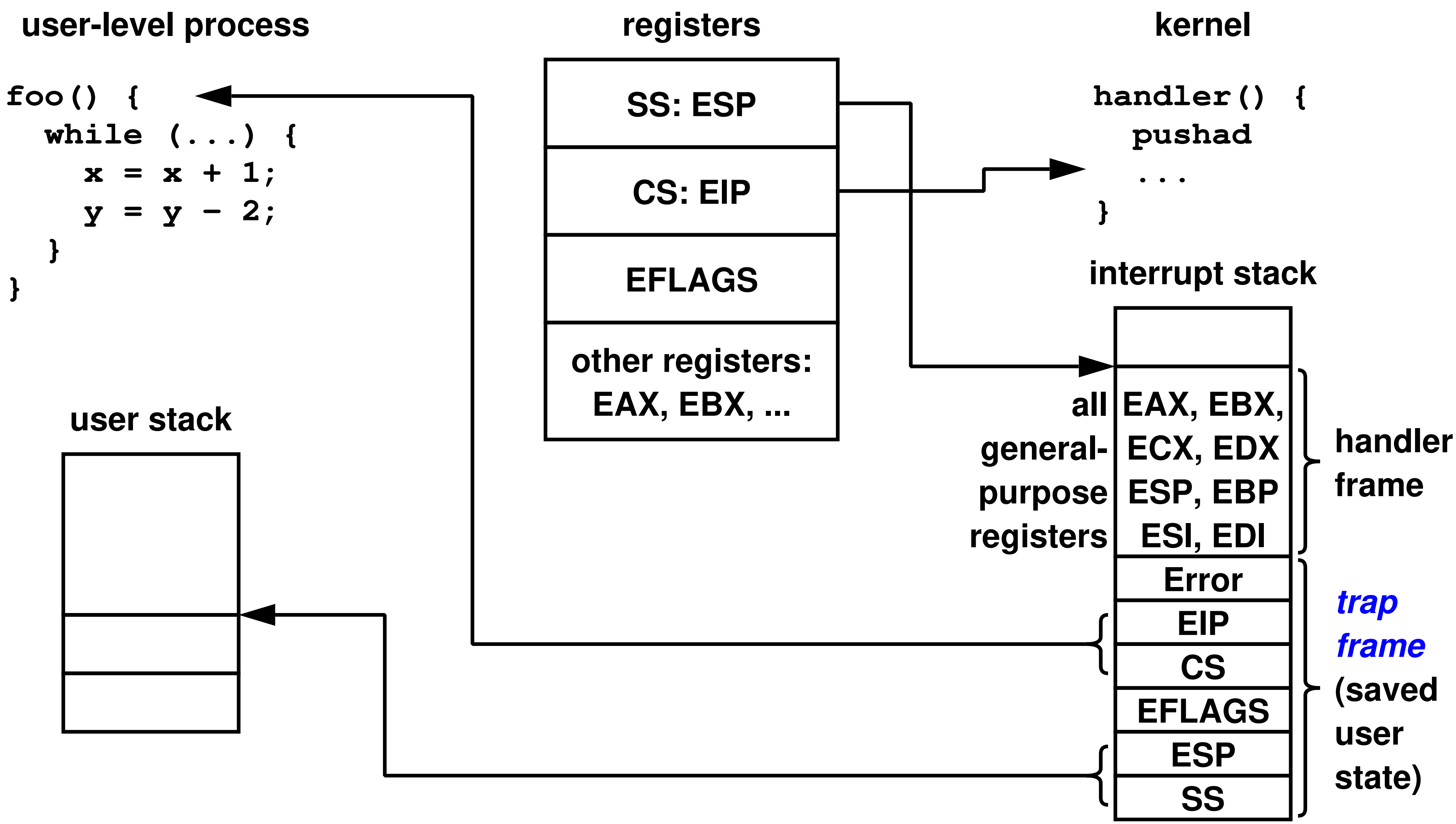
Before Interrupt



Got Interrupt

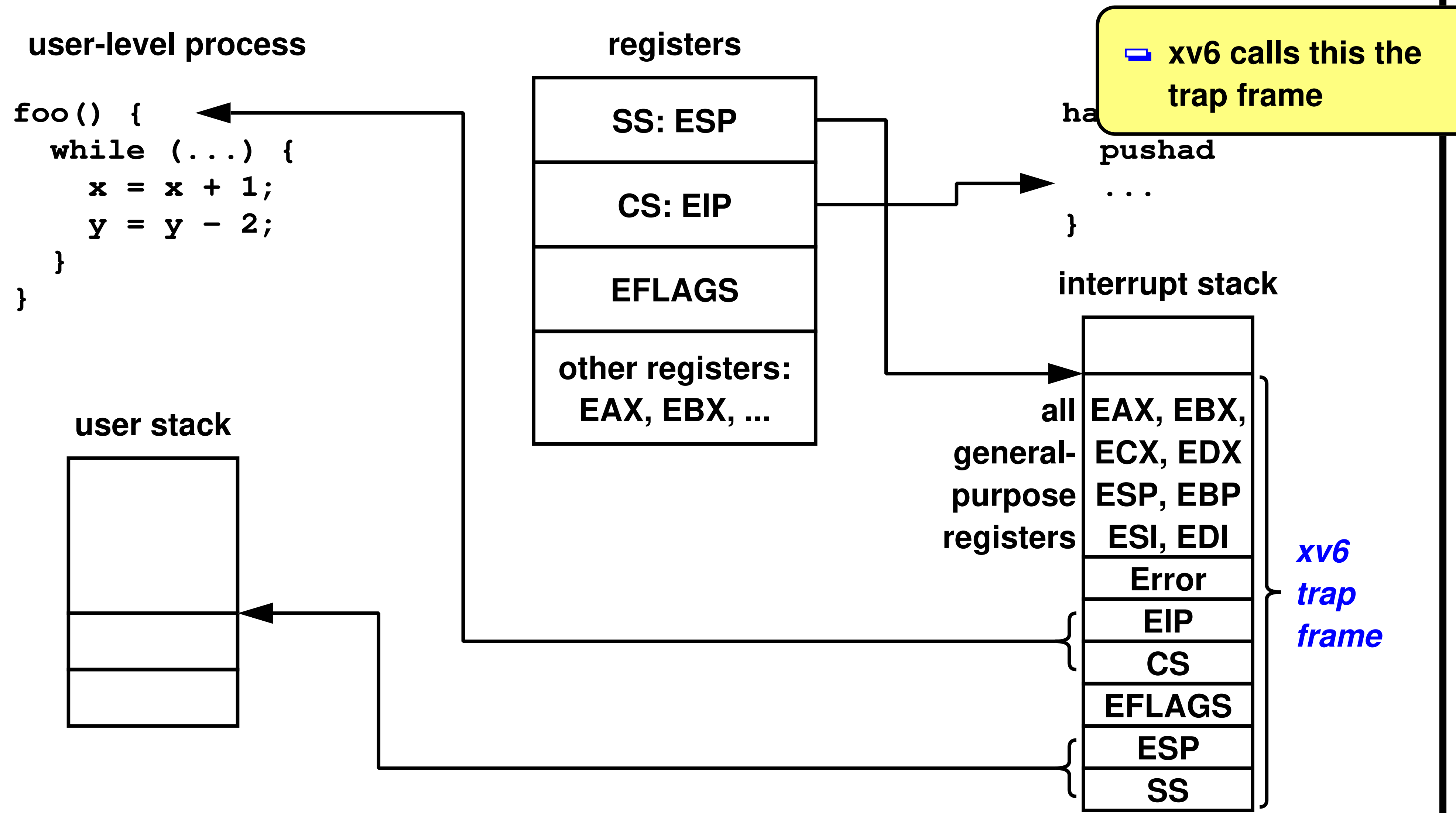


After Interrupt Handler Starts



- **Note: two ESPs saved inside the interrupt stack**
  - one points to interrupt stack, one points to user stack

After Interrupt Handler Starts



- Note: two ESPs saved inside the interrupt stack
  - one points to interrupt stack, one points to user stack

At End Of Handler

- Handler restores saved registers
- `iret`: atomically return to interrupted process/thread
  - restore program counter
  - restore program stack
  - restore processor status word/condition codes
  - switch to user mode
- `iret` is the *only way* to go/return from kernel to user mode for the x86 CPU

kernel

```
handler() {
  pushad
  ...
  popad      <-- (arrow)
  iret
}
```

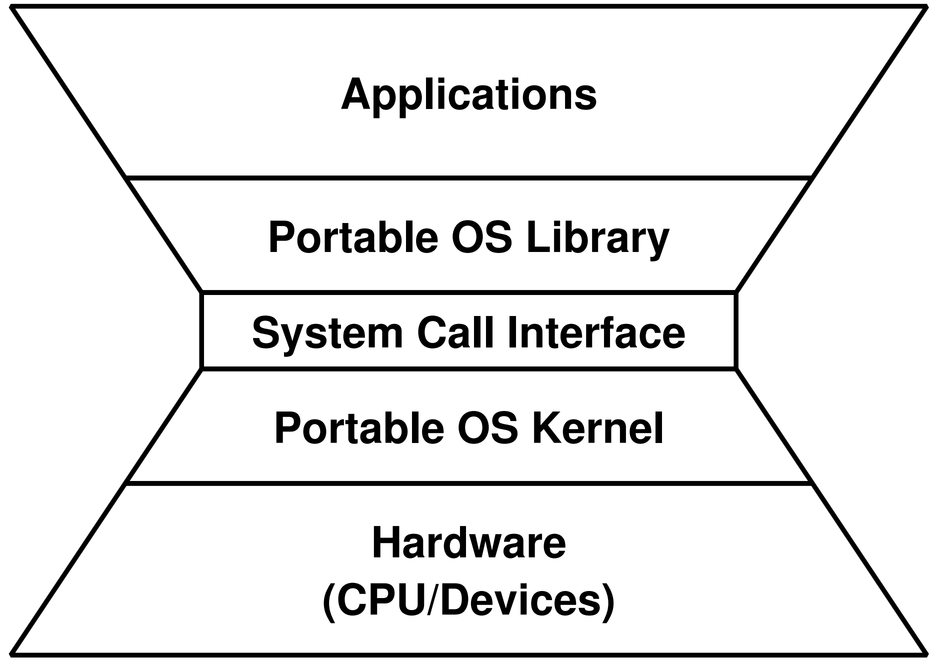
(2.6) Implementing Secure System Calls

Implementing Secure System Calls

- System calls provide the illusion that the OS kernel is simply a set of user space library routines
  - user space program needs not be concerned itself with how the kernel implements system calls
- All system calls follow a certain calling convention
  - e.g., how to name them, how to pass arguments, how to receive return values
  - information is passed in registers and on the executiong stack
  - a trap instruction is eventually invoked to transfer control to the kernel
    - for x86, the machine instruction to trap into the kernel is a *software interrupt* machine instruction

Typical Structure Of System Call Implementation

- In the OS kernel, each system call is implemented by a different function
  - one main difference between this type of function and other OS kernel functions is that it must not trust the values passed from user space
    - bad arguments must not crash the kernel
    - computer virus must not be able to user a system call to take control of the OS kernel



User Stub & Kernel Stub In A System Call

User Program

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

(1) ↓ ↑ (6)

User Stub

```
file_open(arg1, args) {
  push #SYSCALL_OPEN
  int #TRAPCODE
  return
}
```

(2) hardware trap →

(5) trap return ←

Kernel

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

(3) ↑ ↓ (4)

Kernel Stub

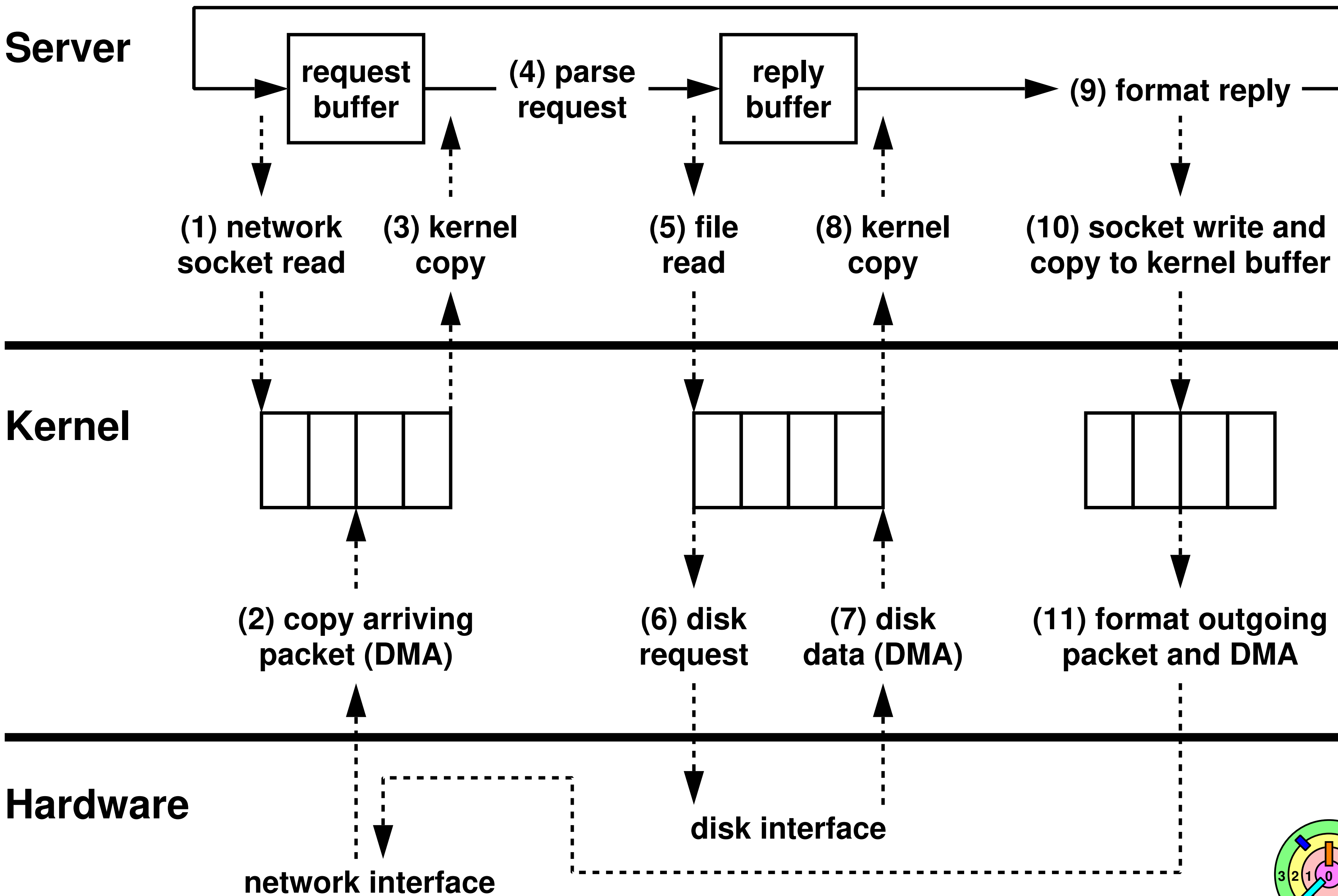
```
file_open_handler() {
 // copy arguments
 //   from user memory
 //   check arguments
  file_open(arg1, args);
 // copy return value
 //   into user memory
  return;
}
```

- `#TRAPCODE` is the *index* into the x86 interrupt vector table for the system call handler
  - for xv6, it's 0x40

Kernel Stub

- Kernel stub has four tasks
  - locate system call arguments
    - in registers or on user stack
      - user stack pointer may be bad (must not trust user)
    - translate user addresses into kernel addresses (again, must not trust user addresses)
  - copy arguments from user memory into kernel memory
    - protect kernel from malicious code evading checks (must do this *before validating arguments* to protect the kernel from a TOCTOU attack)
  - validate arguments
    - protect kernel from errors (or attacks) in user code
      - every byte of user data must be valid and file access rights must be verified
  - copy results back from the kernel into user memory
    - must verify user space addresses before copying

Example: Network Server



(2.7) Starting A New Process

Starting A New Process

- **Step 1: Create a new process**
  - allocate and initialize a new PCB
  - allocate memory for the new process
  - copy program data from disk into the newly allocated memory
  - allocate user-level stack for user-level code execution
  - allocate kernel-level stack to handle system calls, interrupts, and processor exceptions

Starting A New Process

- Step 2: Start running the new process
  - copy arguments into memory
    - by convention, arguments of a process are copied to the base of the user-level stack (i.e., pushed onto the stack)
      - in C, set up `argv[]` to point there
  - transfer control to user mode
    - as if it's returning from a system call (set up the bottom of the kernel stack just right then execute `popad` and `iret`)
  - the starting point of a user program is not `main()`

```
start(argc, argv) {
  exit(main(argc, argv));
}
```

    - the `start()` function doesn't return and it's identical for all programs (and that's why you don't need to write code for this function)

(2.8) Implementing Upcalls

Implementing Upcalls

- It would be nice to have OS-like functionality in user space
  - e.g., be notified about I/O completion interrupt
    - of course, user space program should not be allowed to provide actual interrupt handler (or should it?!)
  - there is a need to "virtualize" some part of the OS kernel so that applications can behave more like the OS

- We call virtualized interrupts and exceptions *upcalls*
  - in Unix/Linux, they are called *signals*
  - in Windows, they are called *asynchronous events*

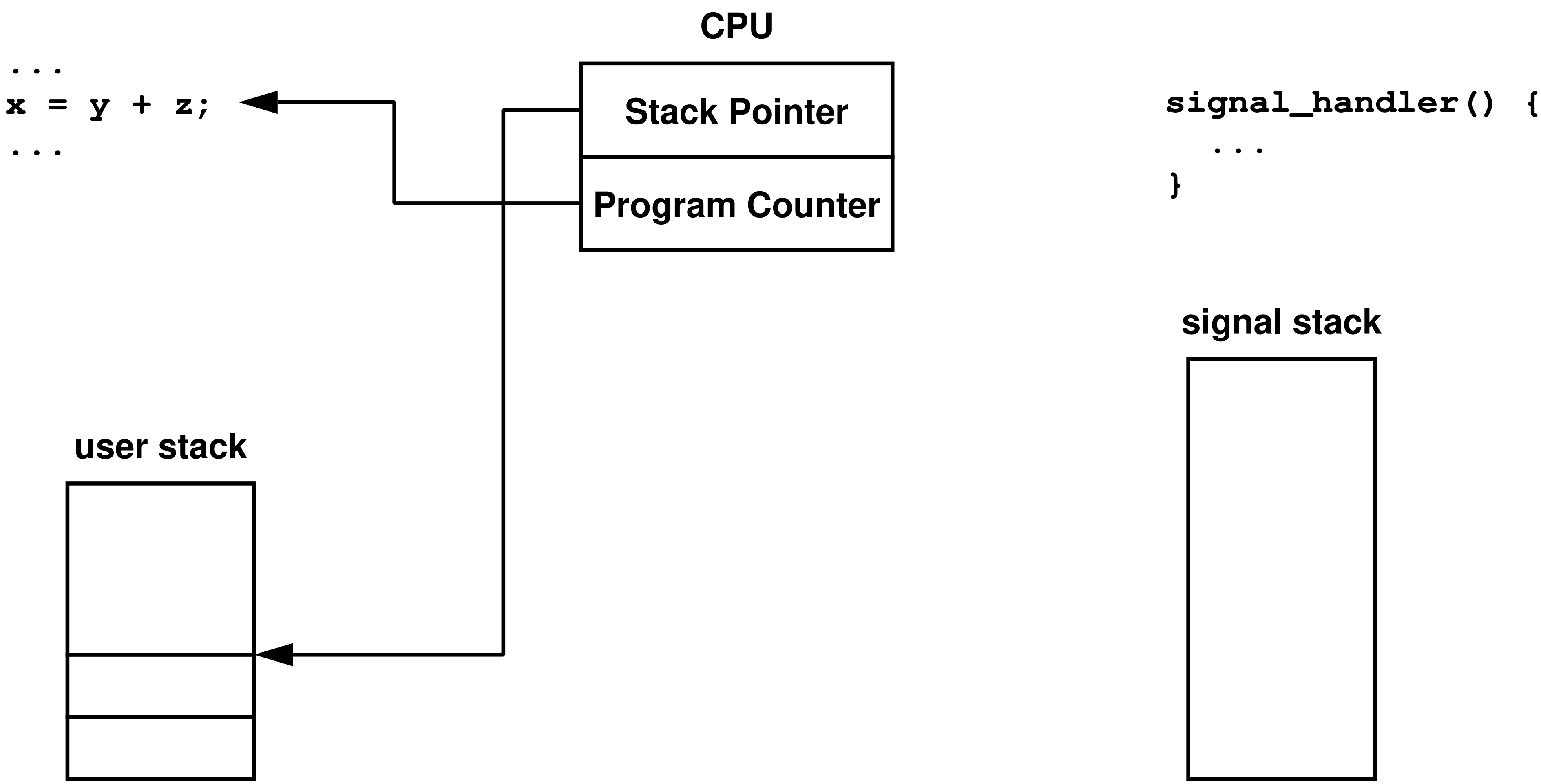
Upcalls

- There are several uses for immediate event delivery with upcalls
  - preemptive user-level threads (e.g., timer upcall)
  - asynchronous I/O notification (e.g., I/O completion upcall)
    - used in *asynchronous I/O*
  - interprocess communication (e.g., debugger upcall to suspend or resume a process, logout upcall to safely self-terminate)
  - user-level exception handling (e.g., divide-by-zero upcall to safely self-terminate)
  - user-level resource allocation (e.g., Java garbage collection upcall when amount of available memory changes for a process)
- Upcalls from kernel to user processes are not always needed
  - event-driven applications don't need upcalls since OS events can be virtualized
    - until recently, Microsoft Windows had no support for immediate delivery of upcalls to user-level programs since application programs are all event-driven

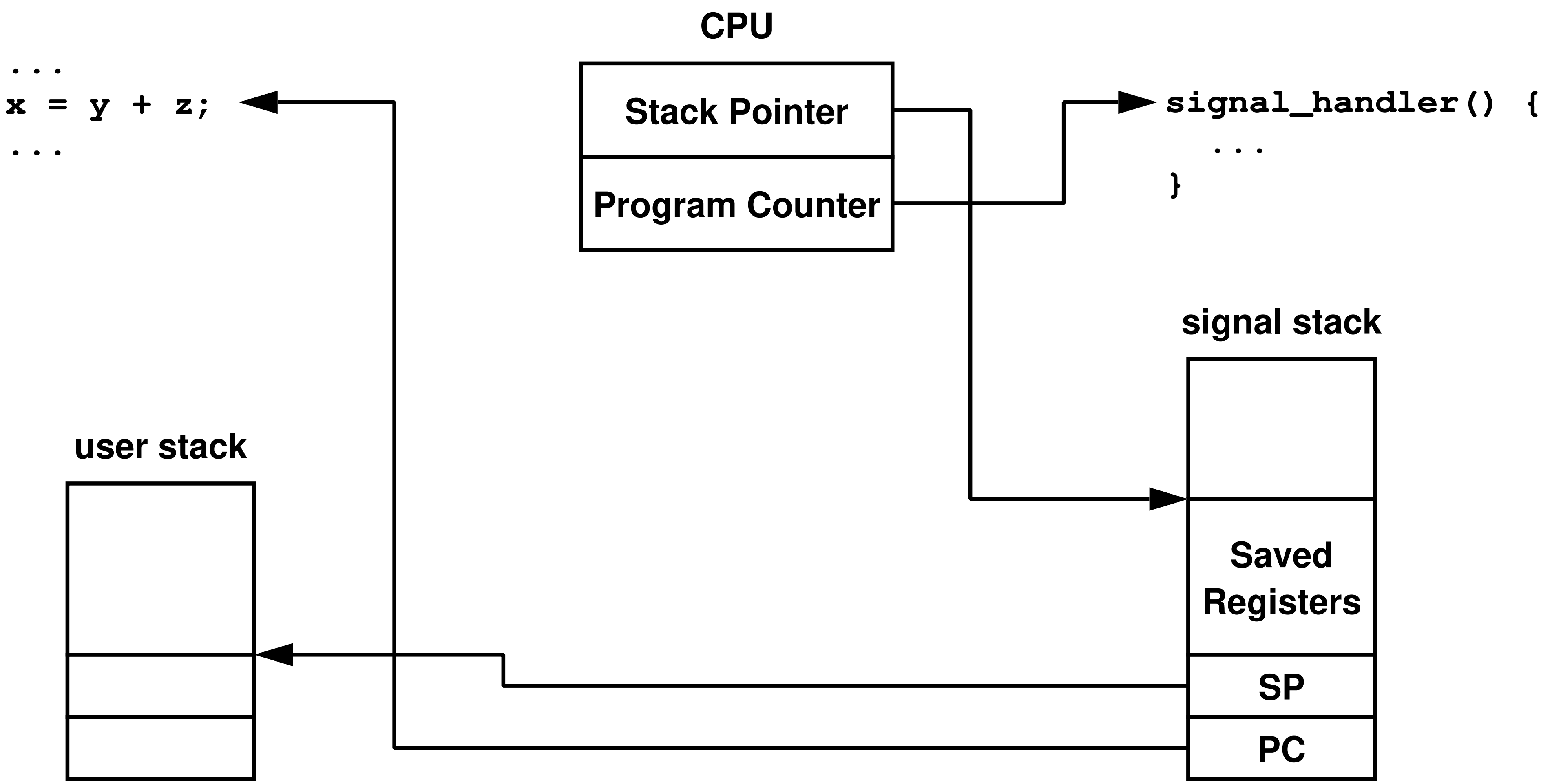
Unix Signals

- Signal delivery to a user space program is similar to hardware interrupt delivery to the kernel
  - instead of *interrupt vector*, Unix has *signal handlers*
  - instead of using an *interrupt stack*, some OSes use a *signal stack*
    - this is a design choice; alternatively, can use a normal execution stack
      - difficult to modify the stack you are using
  - registers are automatically saved and restored, transparent to user processes or kernel
  - *signal masking:* signals disabled while in signal handler (since there is only one signal stack per process)
  - *processor state:* kernel copies onto the signal stack the saved state (i.e., PC, SP, general purpose registers at the point when the user process stopped)
    - the signal handler can modify the saved state (e.g., so that the kernel can resume a different user-level task when the signal handler returns)

Upcall: Before



Upcall: During



- the bottom of the signal stack is set up by the kernel (e.g., copied from the interrupt stack)

Upcall: Implementation

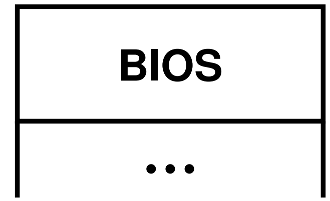
- To implement upcall only requires a small modification to the "return from system call" or "return from interrupt" mechanism
  - e.g., timer interrupt upcall
    - the hardware and the interrupt handler save the state of the user-level computation
    - kernel copies save state to the bottom of the signal stack
    - reset the saved state to point to the signal handler and and signal stack
    - use `iret` to exit the kernel handler and resume user-level execution at the signal handler
    - when signal handler returns, these steps are unwound (i.e., processor state is copied back from the singal handler into the interrupt stack)
    - use `iret` to resume original user-level computation

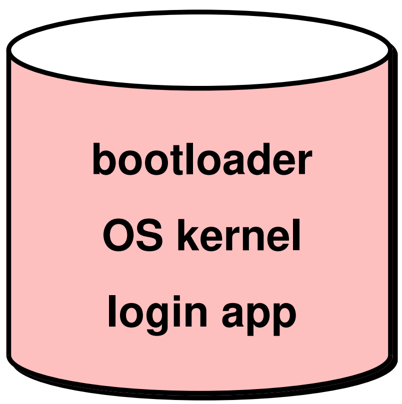
(2.9) Case Study: Booting An OS Kernel

Case Study: Booting An OS Kernel

- How does the OS ***bootstrap*** itself?
  - Ex: running Linux or Windows on a PC
- BIOS in ROM (or EPROM)
  - accessed via physical addresses
  - boot code in BIOS is small and simple (not a good idea to put the entire kernel in ROM)

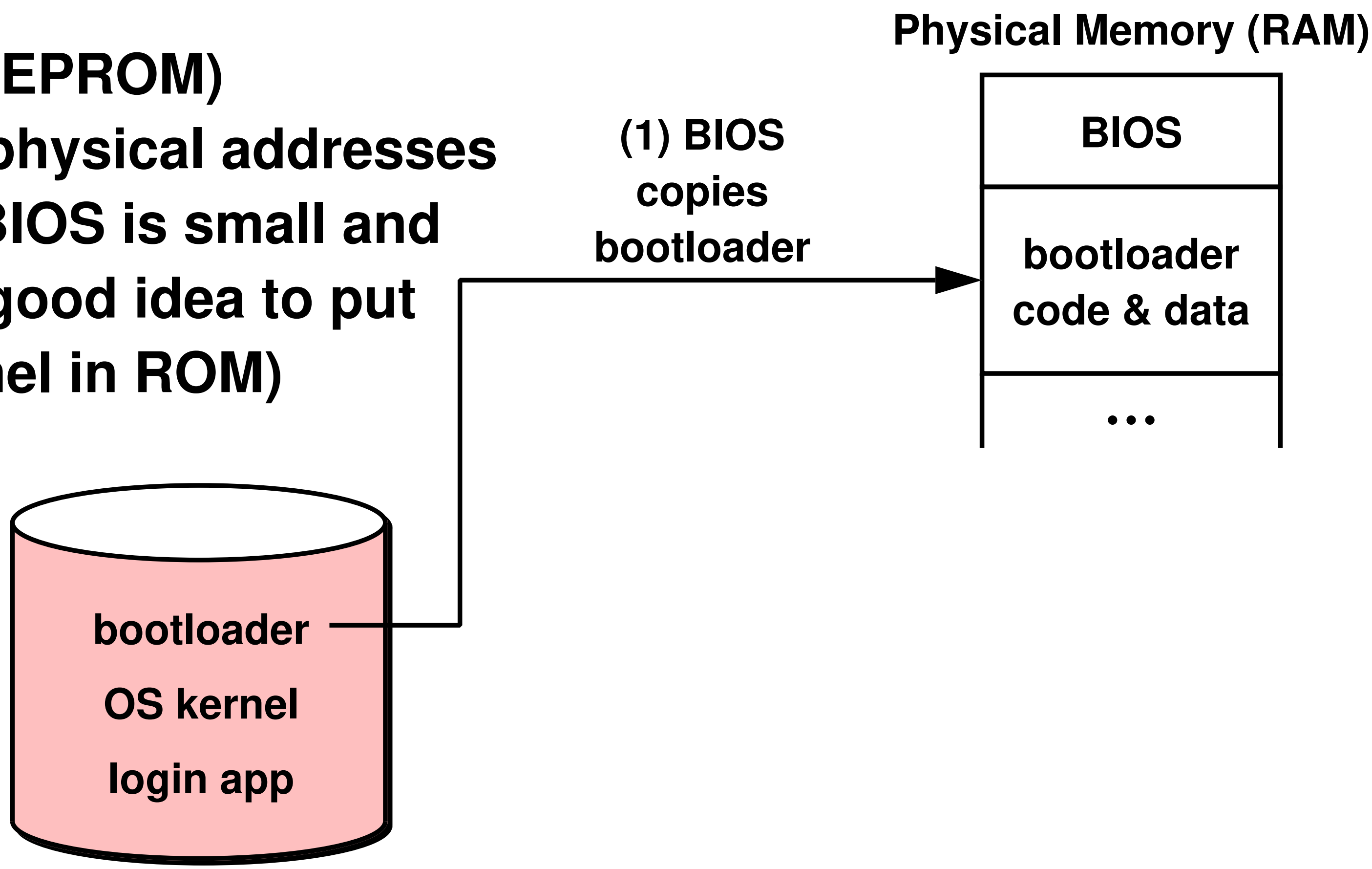
Physical Memory (RAM)





Case Study: Booting An OS Kernel

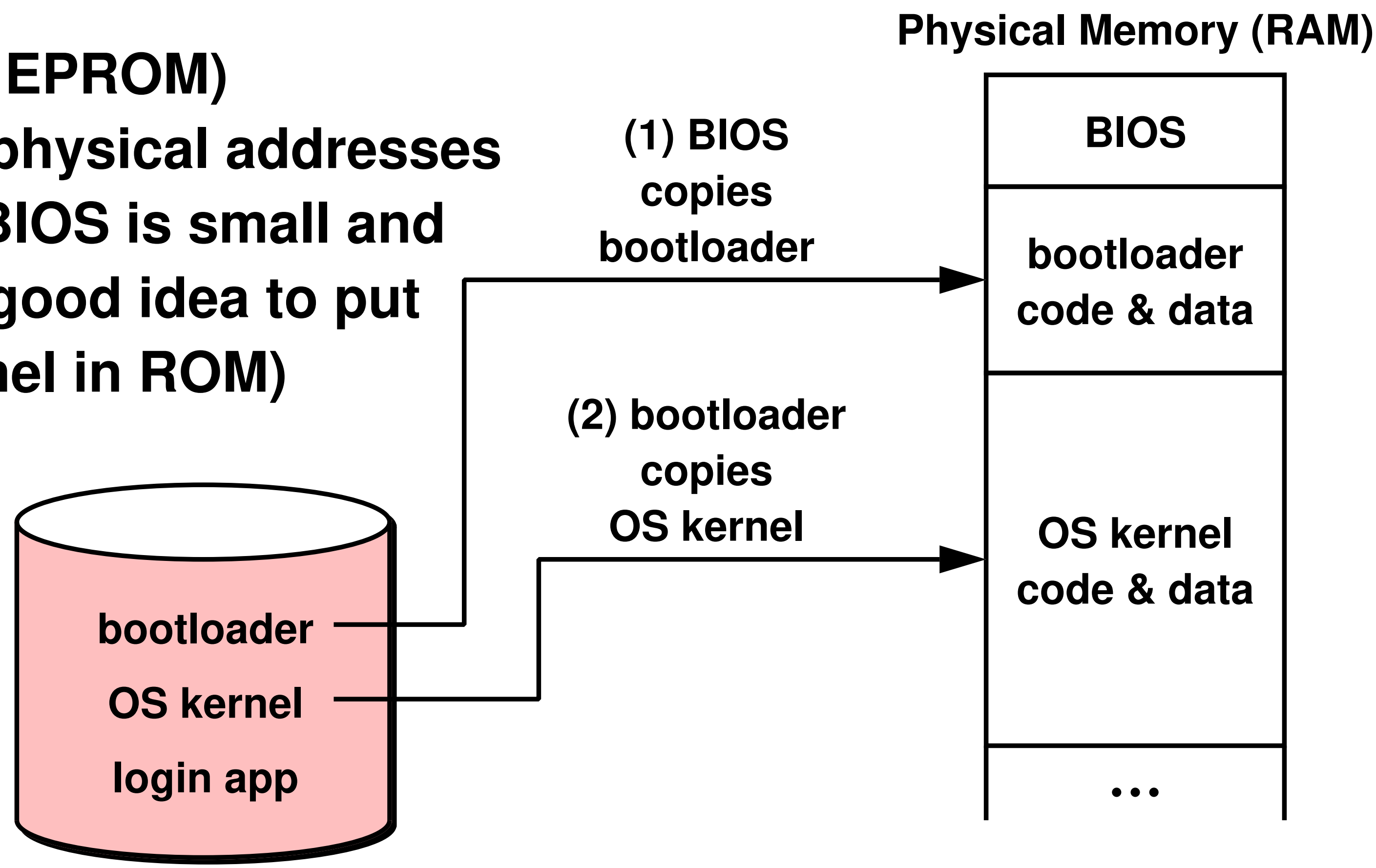
- How does the OS ***bootstrap*** itself?
  - Ex: running Linux or Windows on a PC
- BIOS in ROM (or EPROM)
  - accessed via physical addresses
  - boot code in BIOS is small and simple (not a good idea to put the entire kernel in ROM)



- BIOS ***loads*** the bootloader into memory and ***jumps*** to it
  - on newer hardware, BIOS would first verify the integrity of bootloader

Case Study: Booting An OS Kernel

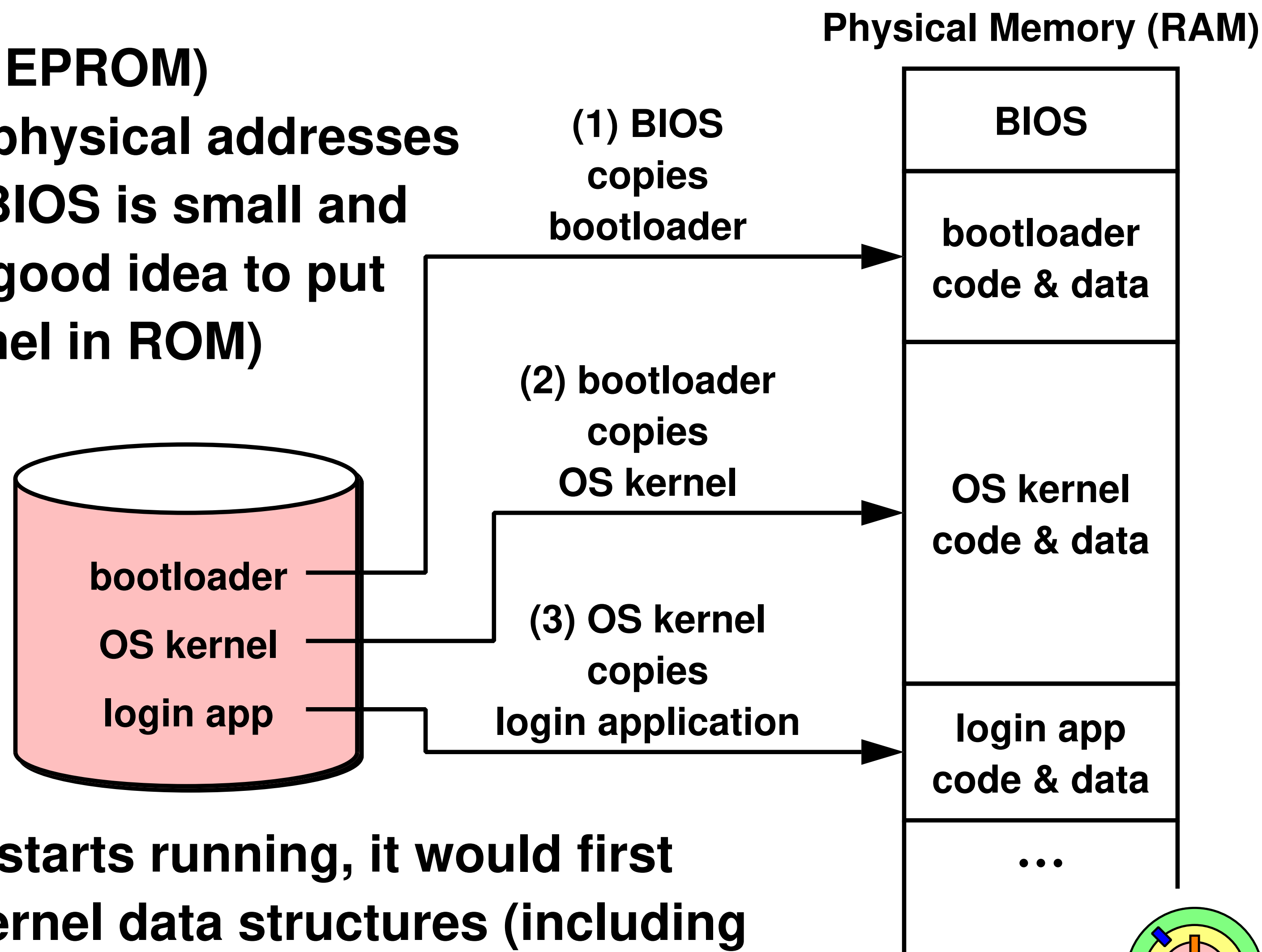
- How does the OS *bootstrap* itself?
  - Ex: running Linux or Windows on a PC
- BIOS in ROM (or EPROM)
  - accessed via physical addresses
  - boot code in BIOS is small and simple (not a good idea to put the entire kernel in ROM)



- Bootloader *loads* the OS kernel into memory and *jumps* to it
  - bootloader would first verify the integrity of OS kernel
  - bootloader knows how to access file system on disk

Case Study: Booting An OS Kernel

- How does the OS ***bootstrap*** itself?
  - Ex: running Linux or Windows on a PC
- BIOS in ROM (or EPROM)
  - accessed via physical addresses
  - boot code in BIOS is small and simple (not a good idea to put the entire kernel in ROM)



- When OS kernel starts running, it would first initialize some kernel data structures (including setting up interrupt vector table)

Booting

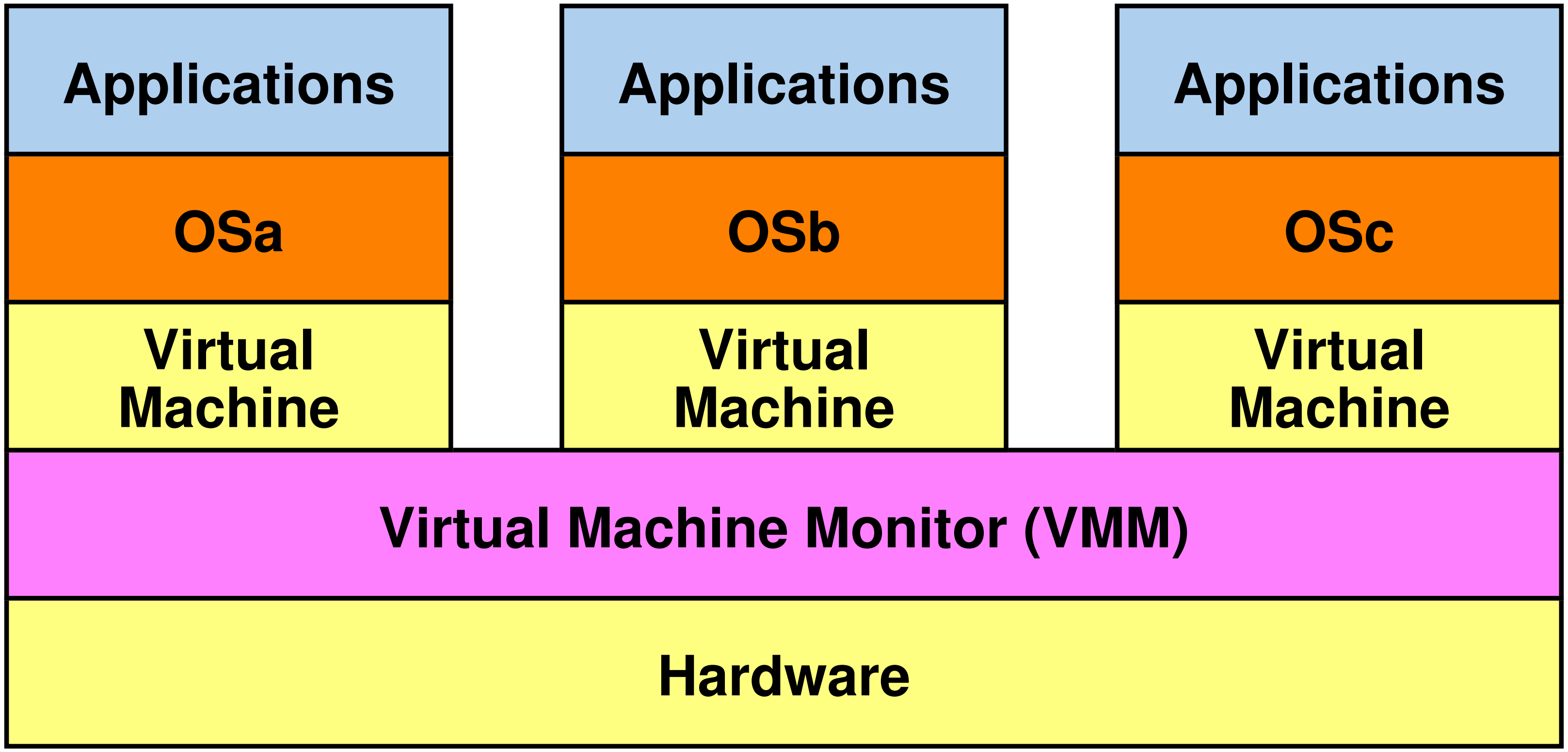
- OS kernel needs to communicate with physical devices
- Devices operate asynchronously from the CPU
  - polling: kernel polls to see if I/O is done
  - interrupts: kernel can do other work in the meantime
- Device access to memory
  - Programmed I/O (PIO): CPU reads and writes to device
  - Direct Memory Access (DMA) by device
  - buffer descriptor: sequence of DMA's

(2.10) Case Study: Virtual Machines

Case Study: Virtual Machines

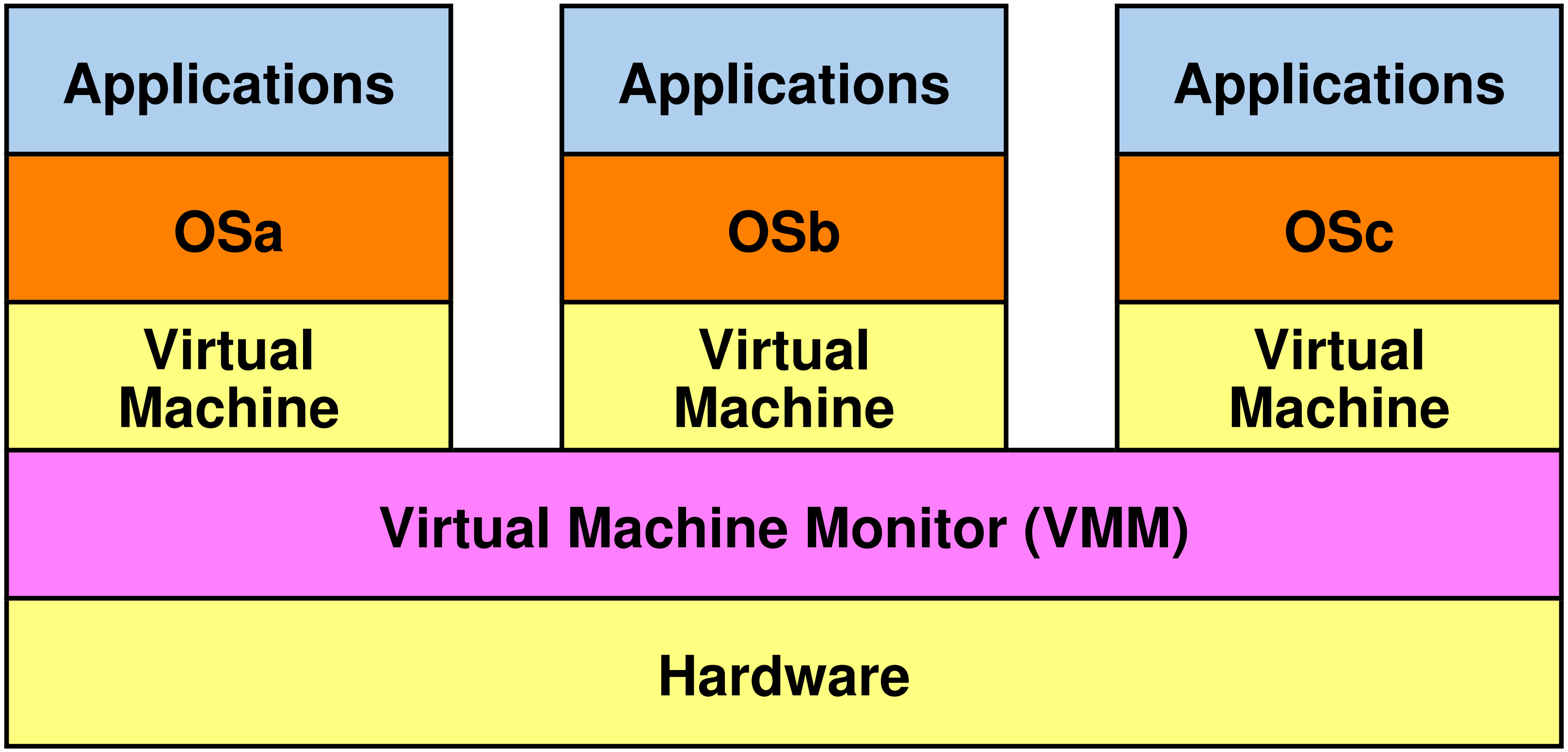
- In the 60s, IBM had a *single-user time-sharing system* called CMS
  - IBM wants to build a *multiuser time-sharing system*
- TSS (Time-Sharing System) project
  - it's a very difficult system to build
  - large, monolithic system
  - lots of people working on it
  - for years
  - total, complete flop
- CP67
  - *virtual machine monitor (VMM)*
  - supports multiple virtual IBM 360s
- Put the two together ...
  - a (working) *multiuser time-sharing system*

Virtual Machines



- **A "*monitor*" is a *synchronization construct* that allows executing entities to have both *mutual exclusion* and the ability to *wait* (block) for a certain condition to become true**
  - **today, we call the VMM a *hypervisor***
- **What abstraction does a *virtual machine* provide?**
  - ***hardware***

Virtual Machines

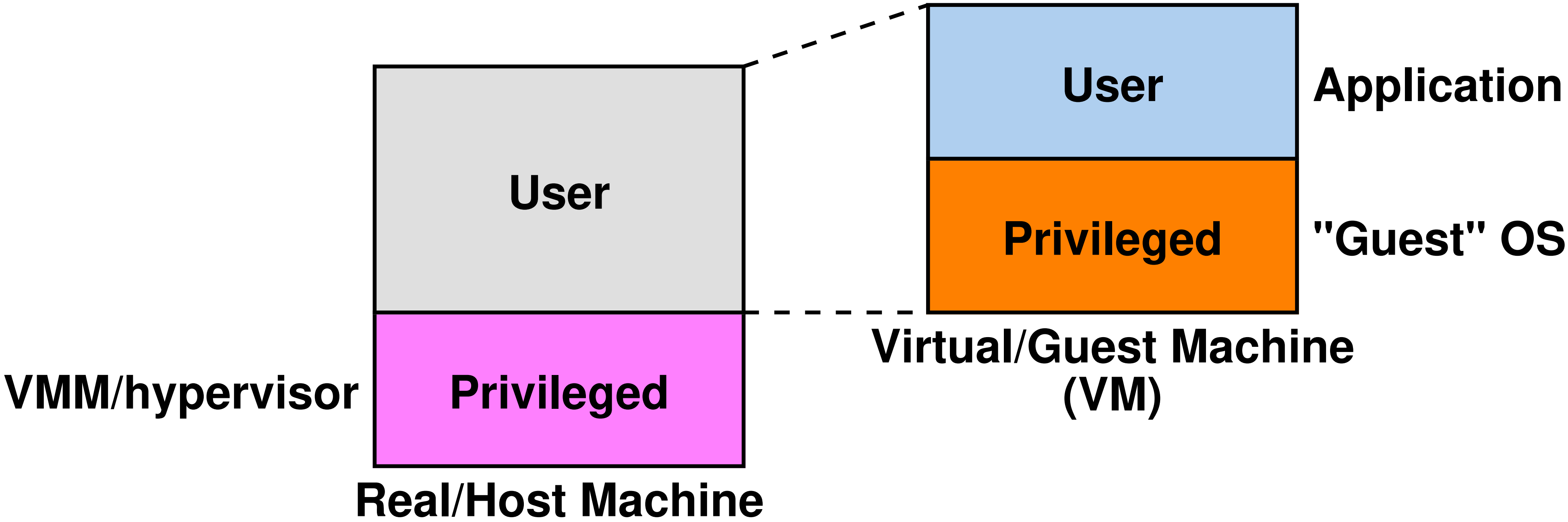


- **A single user time-sharing system could be developed independently of the VMM**
  - **and it can be tested on a real machine (which behaves identical to the VM)**
  - **no ambiguity about the interface VMM must provide to its applications - *identical* to the *real machine*!**

Virtual Machines

- Virtual Machine: run one OS inside (or on-top-of) another OS
  - *run* (not emulate/simulate) OSx on-top-of OSy
    - we will refer to OSx as the *guest OS* and OSy as the *host OS* (a host can have multiple guests)
    - a virtual machine is *not* an OS *emulator*
      - must *execute* guest OS code on the real processor *directly*
  - make the guest OS think that it's running on hardware, but in reality, it is running inside a *virtual machine*

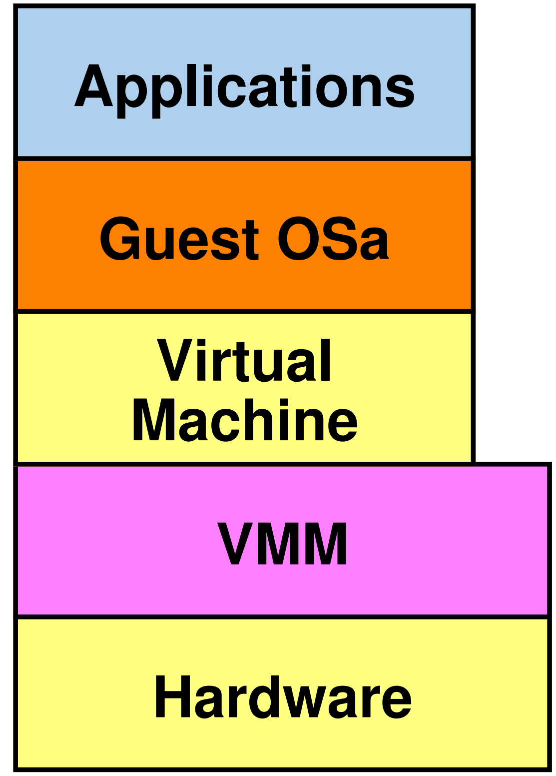
How?



- Run the *entire VM* in *user mode* of the real machine
  - *VMM/hypervisor* runs in the *privileged* mode of the *real* machine
- VMM keeps track of whether each VM is in the *virtual/guest privileged mode* or in the *virtual/guest user mode*
  - *guest OS* runs in the *(virtual) privileged* mode of the VM
  - applications runs in the *(virtual) user* mode of the VM

HOW?

- VMM/hypervisor provides the illusion that the guest OS is running on real hardware
  - e.g., VMM must manage mode transfer between guest processes and guest OS
  - e.g., to provide a guest disk, VMM can simulate a virtual disk as a file on real disk
  - e.g., to provide network access to guest OS, VMM can simulate a virtual network using physical network packets
  - e.g., host kernel must manage memory to provide the illusion that the guest kernel is managing its own memory protection



- "Virtual Machine" in the picture contains: virtual CPU, virtual disk, virtual display, virtual keyboard, etc.
  - *data structures* and *code* that *represent real hardware components*

Virtual Machines: Mode Transfer Example #1

- **Mode transfer example #1:**
  - during boot the host kernel initializes its interrupt vector table as usual
  - when host kernel starts the virtual machine, the guest kernel starts running as if it's being booted:
    1) host loads the guest bootloader from the virtual disk and starts it running
    2) guest bootloader loads the guest kernel from the virtual disk and starts it running
    3) guest kernel initializes its interrupt vector table as it normally would
    4) guest kernel loads a process from the virtual disk into guest memory
    5) to start a process, guest kernel issues instructions to resume execution at user level (use `iret` on x86) and traps into host kernel (since this is a privileged instruction)
    6) ...

Virtual Machines: Mode Transfer Example #1

- **Mode transfer example #1:**
  6) host kernel simulates the requested mode transfer as if the processor had directly executed the `iret` instruction
     - it restores the PC, SP, and processor status word exactly as the guest kernel had intended
- host kernel must protect itself from bugs in guest OS
  - needs to verify the validity of the mode transfer (i.e., make sure that the mode transfer will not end up in the host kernel)

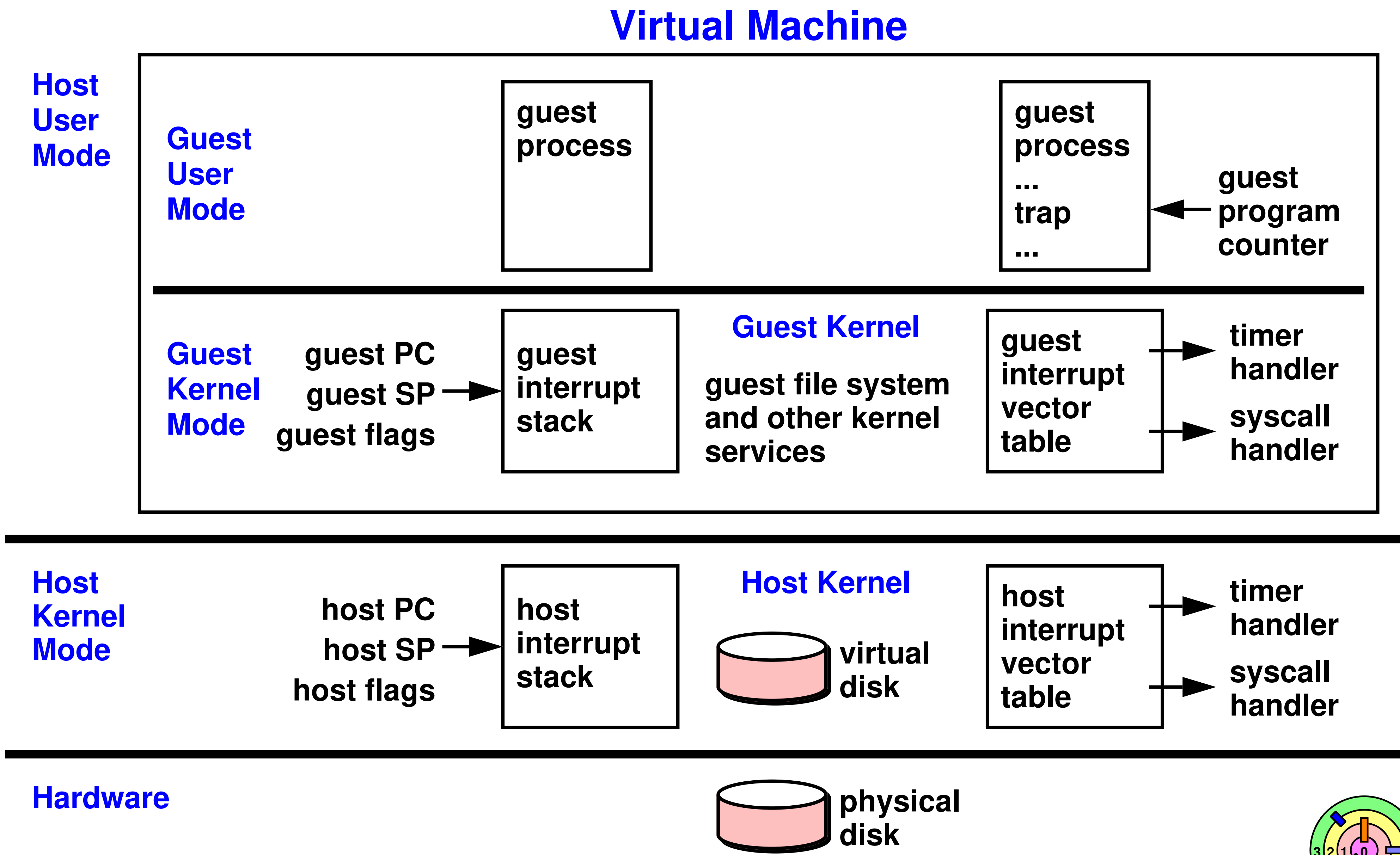
Virtual Machines: Mode Transfer Example #2

- Mode transfer example #2:
  - guest user process makes a *system call*
  - trap machine instruction would trap into the host kernel (and it needs to be *delivered* to the *trap handler* in the guest kernel)
  - host kernel simulates what would have happened had the system call instruction occurred on real hardware running the guest OS:
    1) host kernel saves user space IP, SP, and processor status register on the interrupt stack of the guest kernel
    2) host kernel transfers control to the guest kernel at the beginning of the interrupt handler, but with the guest kernel running with user-mode privilege
    3) guest kernel performs the system call, starts with saving user state and checking arguments
    4) when guest kernel attempts to return from the system call back to user level, this causes a processor exception, dropping back into the host kernel
    5) cont...

Virtual Machines: Mode Transfer Example #2

- **Mode transfer example #2:**
  5) host kernel can then restore the state of the user process, running at user level, as if the guest OS had been able to return there directly

Virtual Machines: Mode Transfer Example #2



Virtual Machines: Exception Handling

- **Host kernel handles processor exceptions similarly**
  - **exceptions generated in the guest user mode needs to be delivered to the guest kernel**
  - **exceptions generated in the guest kernel mode needs to be simulated by the host kernel (if they do not have handlers in the guest kernel)**
  - **therefore, the host kernel must track whether the virtual machine is in the virtual/guest user mode or virtual/guest kernel mode**

- **If you got into the host kernel, think about whether there is a *handler in the guest kernel* or not**
  - **if yes, the job of the host kernel is to *deliver* trap/interrupt to the guest kernel handler**
  - **if no, the job of the host kernel is to *emulate* the trap/interrupt**

Virtual Machines: Timer Interrupt Handling

- Timer interrupts need special handling
  - while servicing a timer interrupt in host kernel, enough virtual time may have passed that the guest kernel is due for a timer interrupt
    - in this case, host kernel needs to invoke the interrupt handler for the guest kernel
    - guest kernel may switch guest user processes
      - this would cause a processor exception (since `iret` is executed) and returning to the host kernel
      - host kernel can then resume the correct guess user process

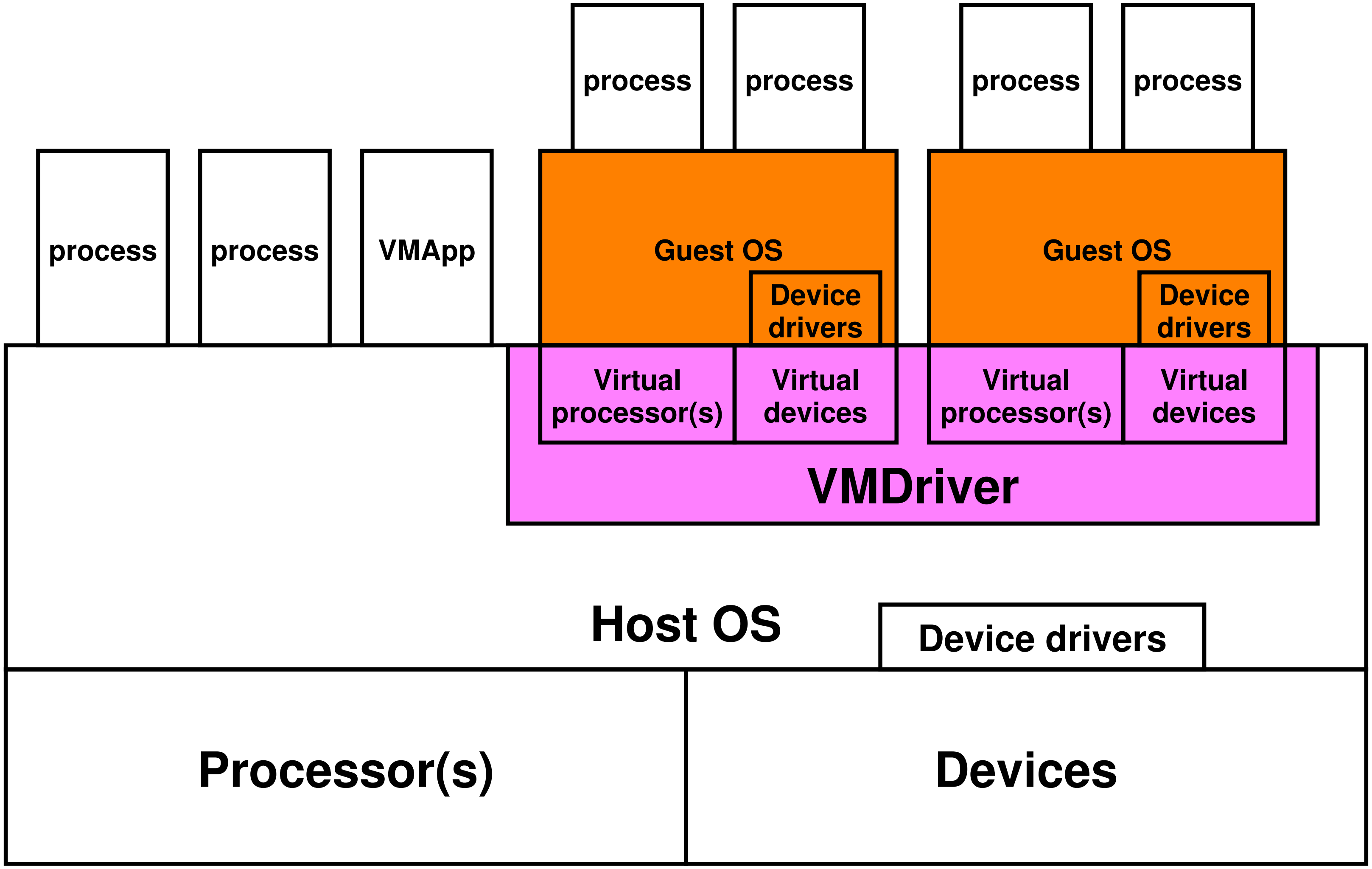
Virtual Machines: I/O Interrupt Handling

- Handling I/O interrupts is similar to handling timer interrupts
  - when guest kernel makes a request to a virtual disk, it would write instructions to the buffer descriptor ring for the virtual disk device
    - in this case, host kernel would translate and perform these instructions on the virtual disk
    - guest kernel expects to receive I/O completion interrupt
      - when the host kernel finishes performing operations on the virtual disk, it needs to invoke the disk interrupt handler for the guest kernel

User-Level Virtual Machine

- How does VMware Workstation Player work?
  - run as a user-level application
  - how does it catch privileged instructions, interrupts, I/O operations?
- Modern OSes allow 3rd party kernel drivers (kind of like device drivers with no corresponding devices)
  - these drivers can intercept hardware and software interrupts to execute its own ISR
    - interpositioning: they can even call the original ISR (similar to DLL injection attack in Windows)
- VMware installs a kernel driver (called VMDriver) into host kernel
  - requires administrator privileges
  - modifies interrupt table to redirect to VMDriver code
  - if interrupt is for VM, upcall
  - if interrupt is for another process, reinstalls interrupt table and resumes kernel

User-Level Virtual Machine



Extra Slides