Ch 2: The Kernel Abstraction

Bill Cheng

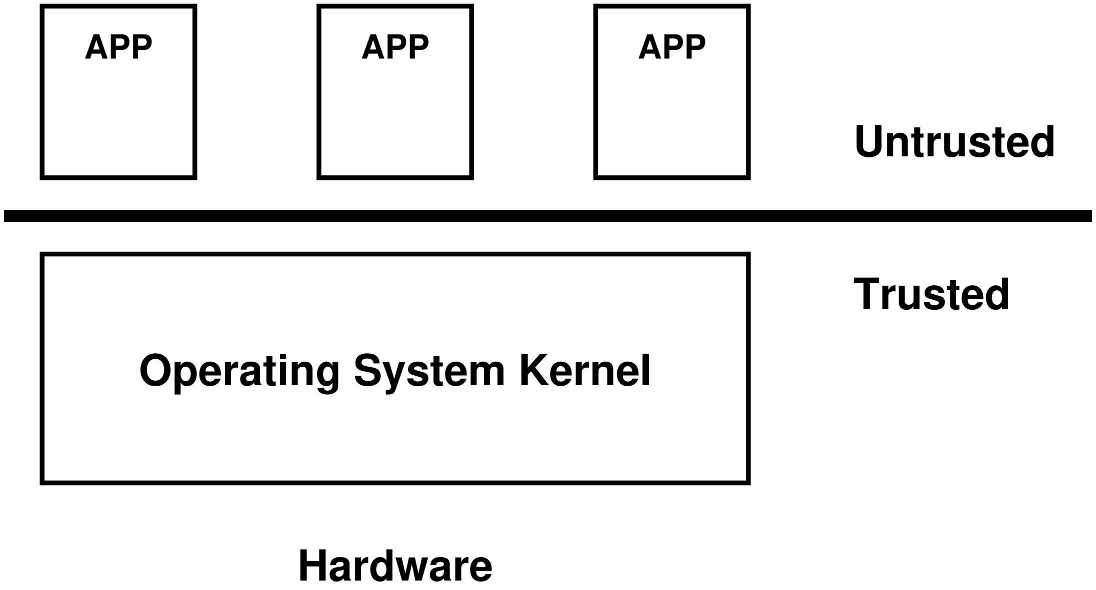
http://merlot.usc.edu/william/usc/

Challenge: Protection

- A central role of OS is *protection*
  - isolating bad applications and users so that they do not corrupt other applications or the OS (which is the protector of other applications)
- Protection is essential to achiving some of the OS goals
  - *reliability* - when an application crashes, it must not affect the OS
  - *security* - protect other applications and the OS from malicious applications
    - trusted code vs. untrusted code
  - *privacy* - on a multi-user system, one user must not be able to access information of another user
  - *fair resource allocation* - an application must not be allowed to use an unfair amount of shared resources (e.g., CPU time, memory, disk space, etc.)
- Implementing protection is the job of the OS kernel

Challenge: Protection

- The OS kernel is the lowest level of software running on the system and has full access to all machine hardware
  - must trust the OS kernel to do anything with the hardware
  - everything else is untrusted and must run in a restricted environment



APP

APP

APP

Untrusted

Trusted

Operating System Kernel

Hardware

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 Concept

- A process is the execution of an application program with restricted rights
  - the process is the abstraction for protected execution provided by the OS kernel
  - a process needs permission from the OS kernel before accessing memory of any other process, before reading/writing to disk, before changing hardware settings, etc.

- The OS kernel runs directly on the processor with unlimitted rights
  - what about applications?
  - in order to have *good performance*, applications also need to run directly on the processor
    - but with all potentially dangerous operations *disabled*
  - hardware can help to improve performance
    - in general, the more hardware help the better/faster
      - how much can and should hardware help?

Dual-mode Operation

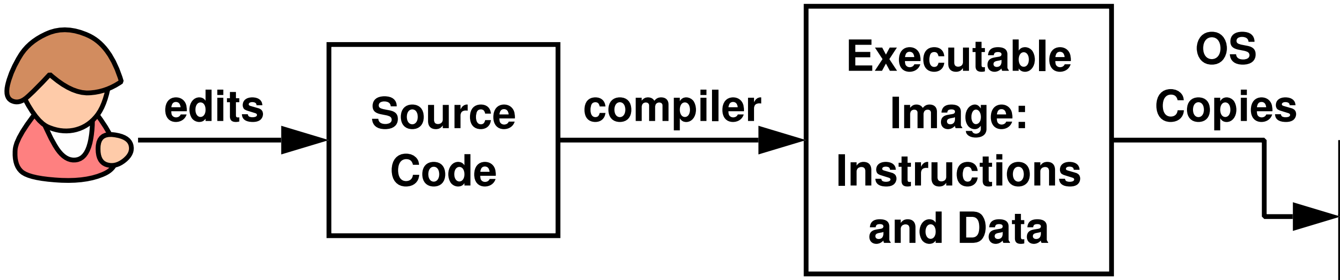
- Split personality of a process
  - when running the OS kernel, it's in charge of everything and can do anything it wants
  - when running applicatino code, it needs to ask permissions to do anything potentially harmful to other applications or the OS kernel
  - remember that they are running on the same processor, sometimes completely trustworthy and other times completely untrusted

Safe Control Transfer

- Application to OS kernel: make *system call*
  - return from system call

(2.1) The Process Abstraction

The Process Abstraction



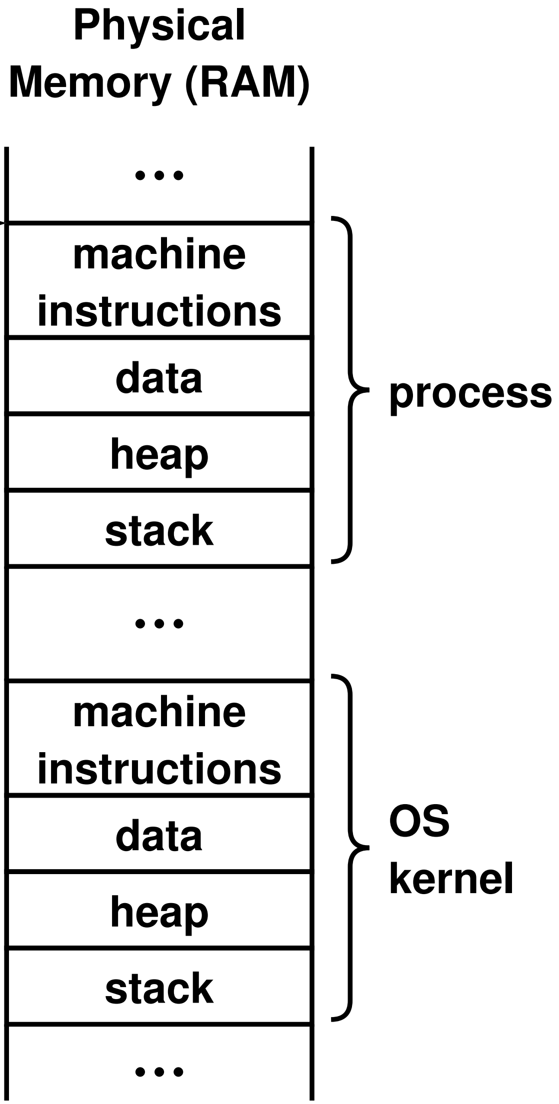
edits

Source Code

compiler

Executable Image: Instructions and Data

OS Copies



Physical Memory (RAM)

...

machine instructions

data

heap

stack

...

machine instructions

data

heap

stack

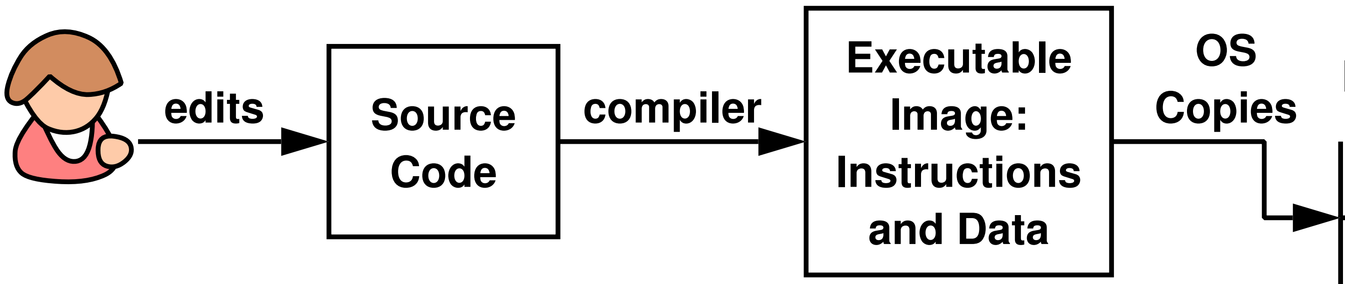
...

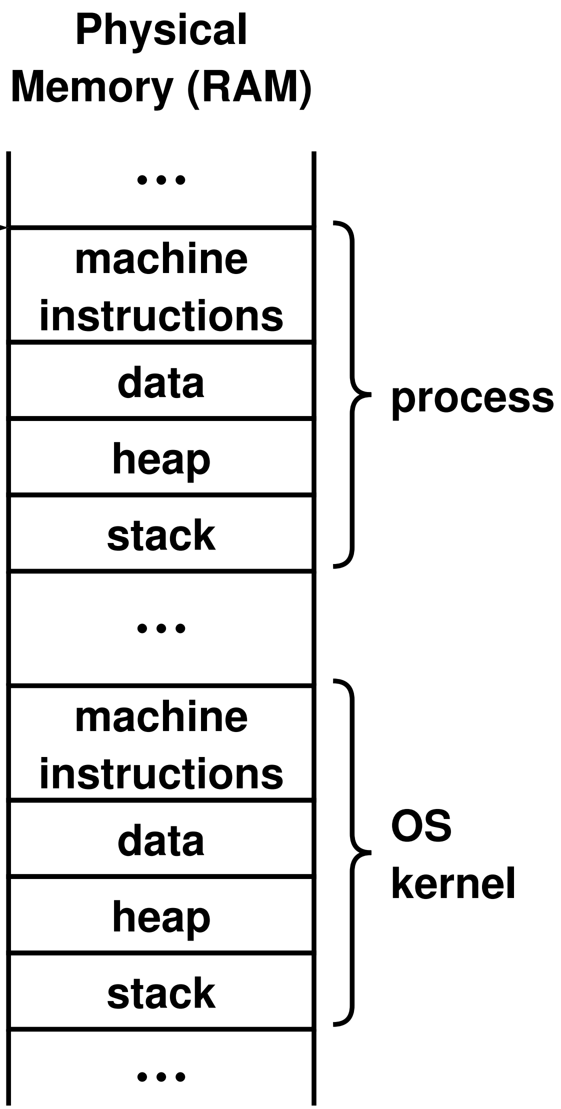
process

OS kernel

- A process is a running program
  - it has an *address space* that's made up of *memory segments*
    - machine instructions are kept inside the *text* segment
    - global variables are kept inside the *data* segment
    - the *heap* holds dynamically allocated data structures that the process might need
    - the *stack* holds the state of local variables and function arguments during procedure calls
  - conceptually, the OS kernel has its own address space

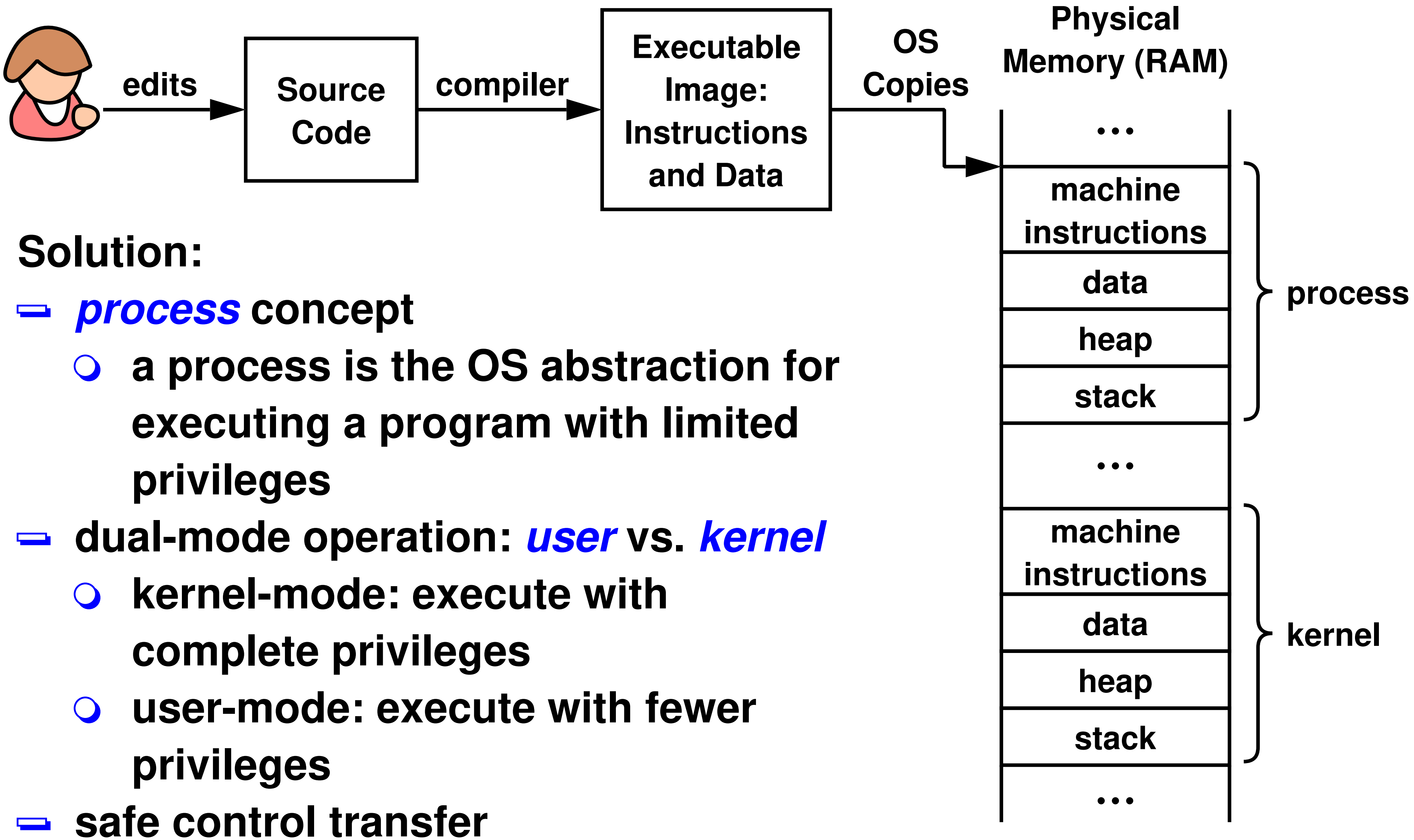
The Process Abstraction





- What's to keep the process from modifying data in the OS kernel or other processes?
- What's to keep the process from modifying data on disk?

The Process Abstraction



Solution:

- ***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?

The 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 a process can access
    - other permissions the process has (e.g., what memory is shared with another process)

Process Control Block (PCB)

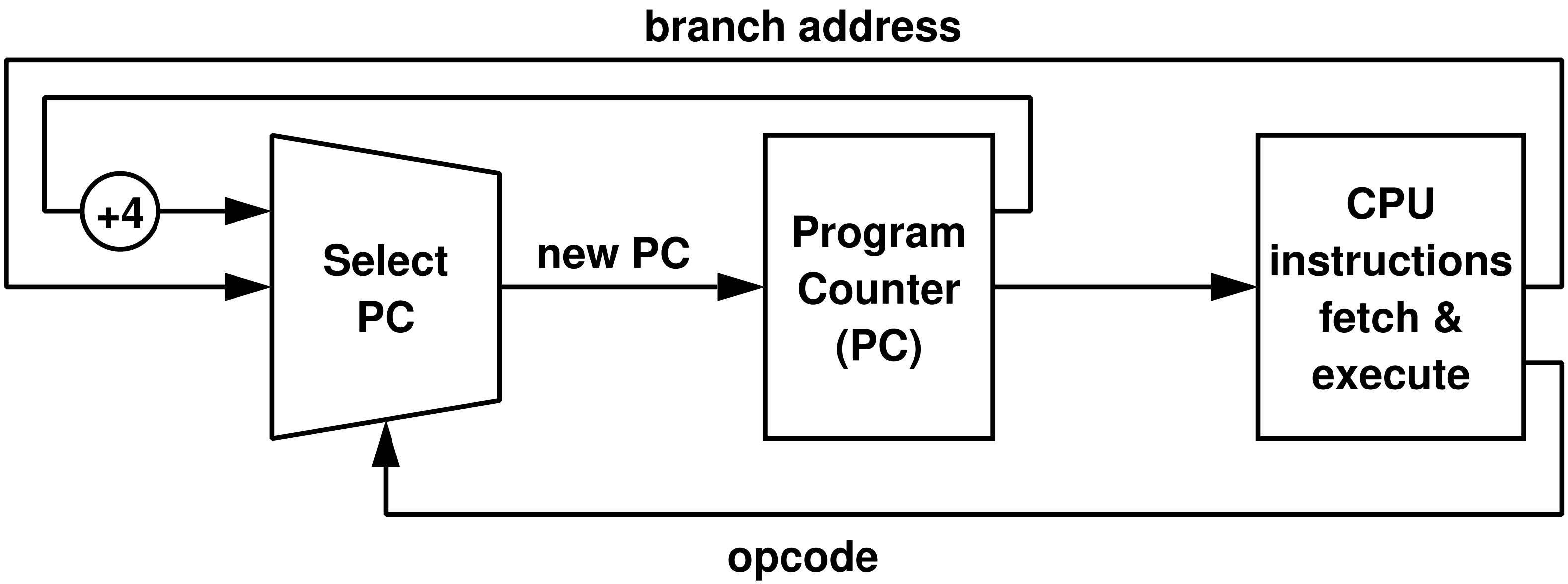
- **OS maintains information about every process in a data structure called a *Process Control Block (PCB)***
- **PCB contains information such as:**
    - **where the process data (e.g., code, global variabls, stack, heap) is stored in memory**
    - **where its executable image resides on disk**
    - **which user asked to execute the program**
    - **what privileges the process has**
    - **...**

(2.2) Dual-Mode Operation

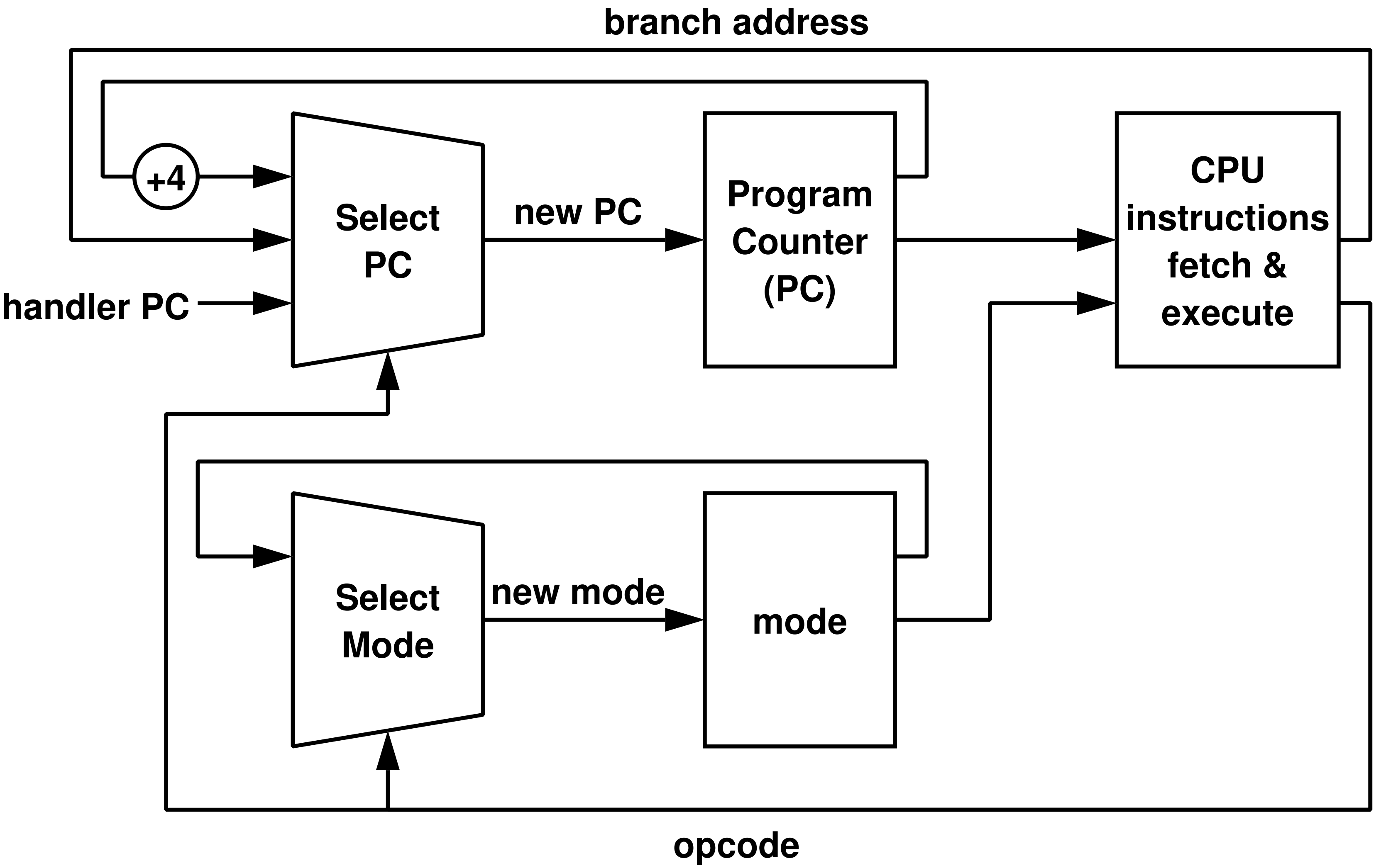
(2.2) Dual-Mode Operation

- 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
- On the x86, mode stored in EFLAGS register
- On the MIPS, mode in the status register

A Model of a CPU



A CPU with Dual-Mode Operation



A CPU with 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

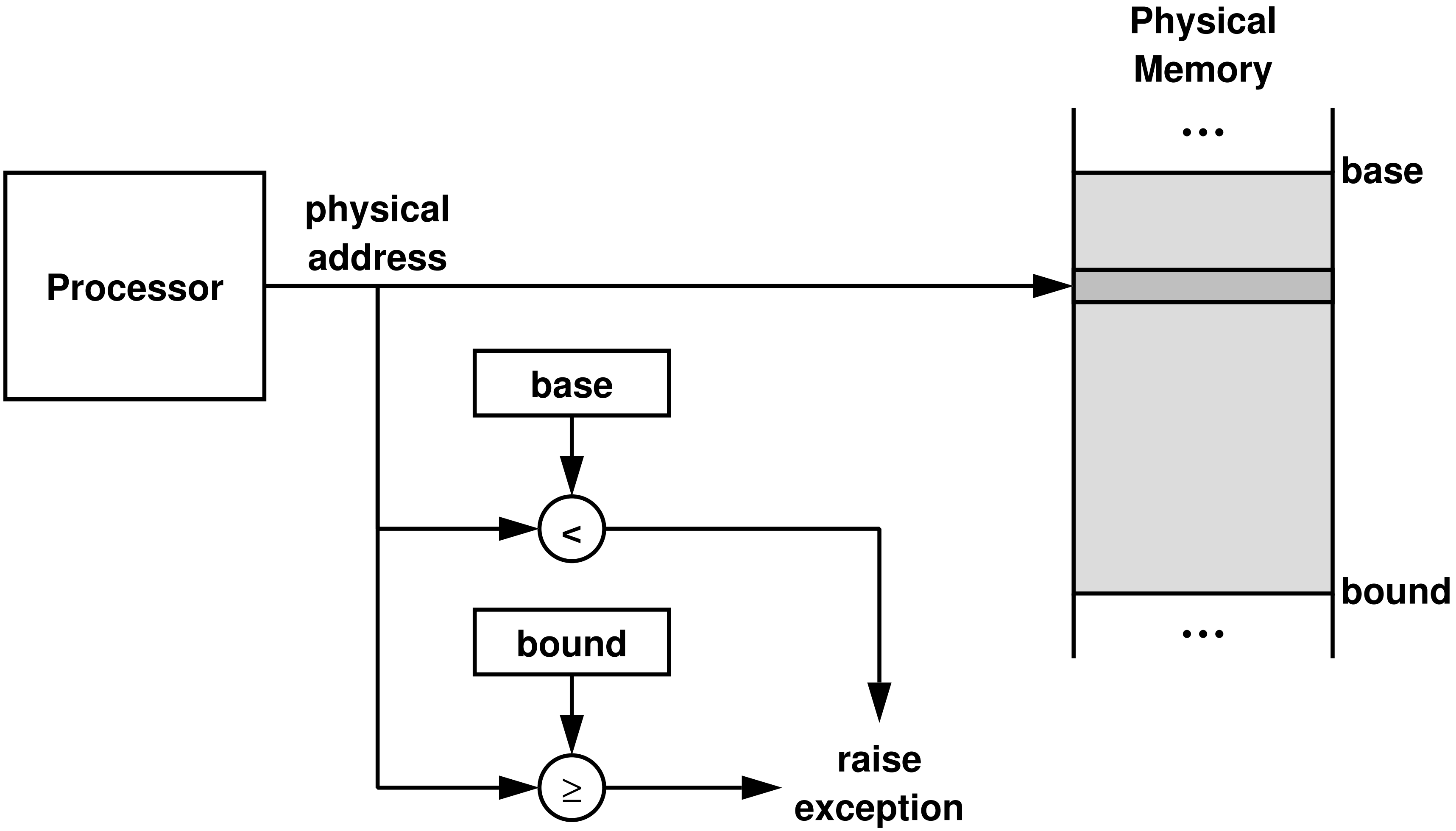
- What would be an example of a privileged instruction?
  - change mode bit in EFLAGS register
  - change which memory location a user program can access
  - send commands to I/O devices
  - read data from or write data to I/O devices
  - jump into kernel code

Privileged Instructions

- What should happen if a user program attempts to execute a privileged instruction?
  - would cause a *processor exception* (in hardware)
    - which would cause the processor to transfer control to an *exception handler* in the OS kernel
      - usually, the kernel simply halts the process after a privilege violation

- What bad thing can happen if an application can jump into kernel mode at any location in the kernel?
  - it may crash the kernel
  - it may allow the application to access privileged data
  - it may allow the application to bypass security checks

Simple Memory Protection: Base & Bound Registers



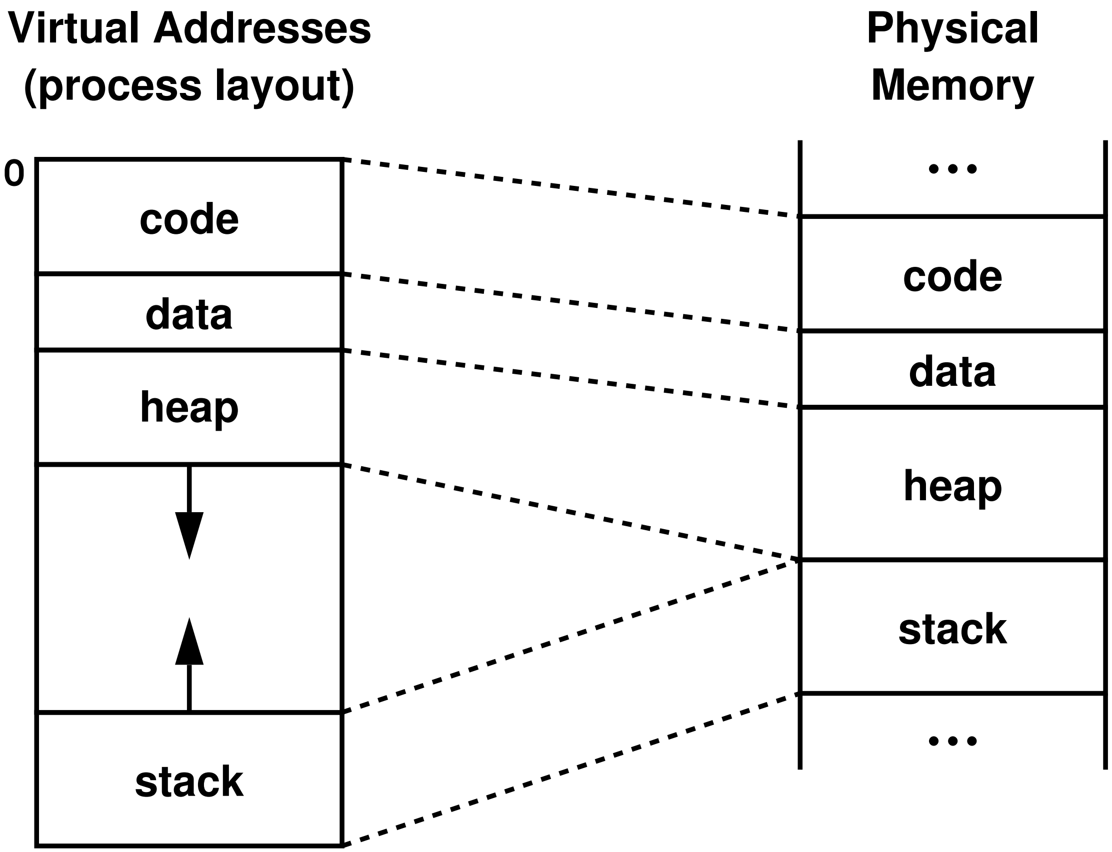
- Can only modify these registers using privileged instructions
  - otherwise, application can access data that belongs to the kernel or other processes

Towards Virtual Addresses

- What’s are the problems with base and bound?
  - addresses used by an application must be contiguous (cannot have gaps)
    - it would be nice if the stack and heap can grow and shrink
  - cannot share code between processes
  - absolute addresses are difficult to use
    - how to load the same program at two different memory locations?
      - e.g., "`jmp 0x12345678`"
  - memory fragmentation

Virtual Addresses

- Translation done in hardware (on every address), using a table set up by the OS kernel
  - the (virtual) memory of every process starts at the same place, i.e., 0
    - these memory "segments" can be located anywhere in physical memory



Virtual Address Example

```
int staticVar = 0; // a static variable
main() {
  staticVar += 1;
  sleep(10); // sleep for 10 seconds
  printf("static address: %x, value: %d\n",
      &staticVar, staticVar);
}
```

- If you run two instances of this program simultaneously, you would get the same printout from them if *virtual addresses* are used
  - if you don’t have support for virtual addresses and have to use physical addresses, the printout will be different

Virtual Address Example

```
int staticVar = 0; // a static vari
main() {
  staticVar += 1;
  sleep(10); // sleep for 10 second
  printf("static address: %x, value
      &staticVar, staticVar);
}
```

- my color codes for code
  - reserved/key words are in blue
  - numeric and string constants are in red
  - comments are in green
  - black otherwise

- If you run two instances of this program simultaneously, you would get the same printout from them if *virtual addresses* are used
  - if you don't have support for virtual addresses and have to use physical addresses, the printout will be different

Hardware Timer

- What if a programming bug causes a user process to go into an infinite loop and never give up the processor?
  - we need a way for the OS kernel to gain control periodically
- Hardware timer is a device that periodically interrupts the processor
  - returns control to the kernel handler
  - interrupt frequency set by the kernel
    - not by user code
    - expires every few milliseconds (human reaction time is a few hundred of milliseconds)
  - interrupts can be temporarily deferred
    - not by user code
    - interrupt deferral crucial for implementing mutual exclusion

(2.3) Types of Mode Transfer

Types of Mode Transfer

- User Mode To Kernel Mode
- Kernel Mode To User Mode

Mode Switch: User Mode To Kernel Mode

- Interrupts (if we don't say "software interrupt", we mean "hardware interrupt")
  - triggered by timer and I/O devices
- Exceptions
  - triggered by unexpected program behavior
  - or malicious behavior!
  - e.g., divide by zero, execute a privileged instruction
  - when such an exception occurs, the thread in the user process *"traps"* into the kernel
- System calls (aka protected procedure call)
  - request by program for kernel to do some operation on its behalf
  - only limited number of very carefully coded *kernel entry points*
  - e.g., read data from disk, create another process
  - also *"trap"* into the kernel
  - when a thread in a user process makes a system call, it *"traps"* into the kernel

Mode Switch: User Mode To Kernel Mode

- **Interrupts (if we don't say "software interrupt", interrupt")**
  - triggered by timer and I/O devices

- *trap* is a *synchronous* event (to transfer from user mode to kernel mode
- *interrupt* is an *asynchronous* event (can happen at any time)

- **Exceptions**
  - triggered by unexpected program behavior
  - or malicious behavior!
  - e.g., divide by zero, execute a privileged instruction
  - when such an exception occurs, the thread in the user process *"traps"* into the kernel
- **System calls (aka protected procedure call)**
  - request by program for kernel to do some operation on its behalf
  - only limited number of very carefully coded *kernel entry points*
  - e.g., read data from disk, create another process
  - also *"trap"* into the kernel
  - when a thread in a user process makes a system call, it *"traps"* into the kernel

Mode Switch: Kernel Mode To User Mode

- New process (or new thread) starts
  - jump to first instruction in program (or thread)
- Return from interrupt, exception, system call
  - resume suspended execution
- Process (or thread) context switch
  - resume some other process (or thread)
- User-level upcall (UNIX signal)
  - asynchronous notification to user program

(2.4) Implementing Safe Mode Transfer

Implementing Safe Mode Transfer

- Context switch code must be carefully crafted
  - relies on hardware support
- Most OS has a common sequence of instructions for enter the kernel and for returning to user level, regardless of cause
  - at a minimum, this common sequence must provide
    - limited entry into the kernel
      - an entry point must be set up by the kernel and not allow entry into the kernel at arbitrary points
    - *atomic* changes to processor state
      - processor mode, program counter, stack pointer, memory protection registers all change at the same time
    - transparent, restartable execution
      - an interrupt must be invisible to the user process (i.e., serviced transparently)
      - if an interrupt is serviced in the middle of an instruction execution, the CPU needs to be able to restart or finish the execution of that instruction seamlessly

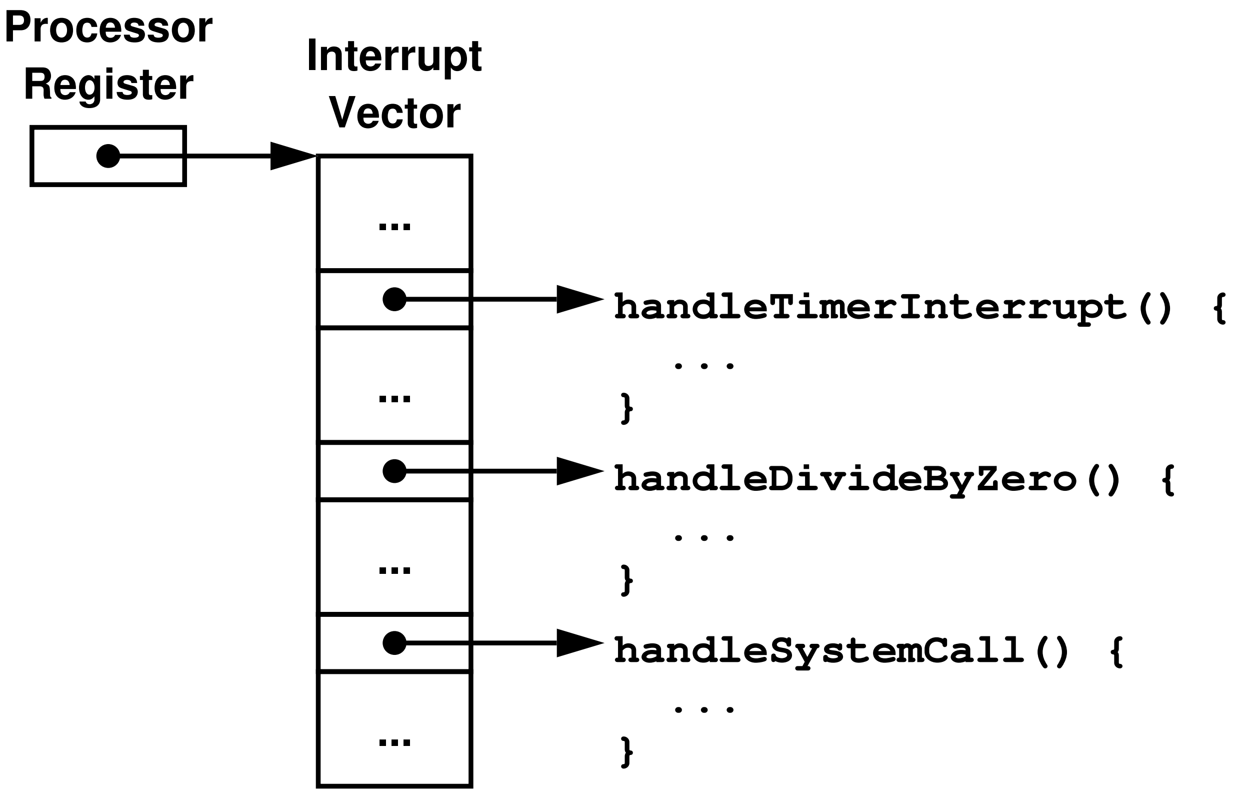
Common Interrupt / Exception Handling

- On an interrupt (or exception), the following happens
  1) the processor saves its current state to memory
  2) further events are deferred
  3) changes to kernel mode
  4) jump to the interrupt or exception handler

- When the handler finishes, the steps are reversed and the processor state is restored from its saved location
  - the interrupted entity has no idea that an interrupt has occurred

Interrupt Vector (Table)

- **Table set up by OS kernel**
  - **the interrupt vector is an array of function pointers, pointing to code to run on different events**
  - **a special purpose processor register stores the address of this array**



Processor Register

Interrupt Vector

...

...

...

...

```
handleTimerInterrupt() {
   ...
}
handleDivideByZero() {
   ...
}
handleSystemCall() {
   ...
}
```

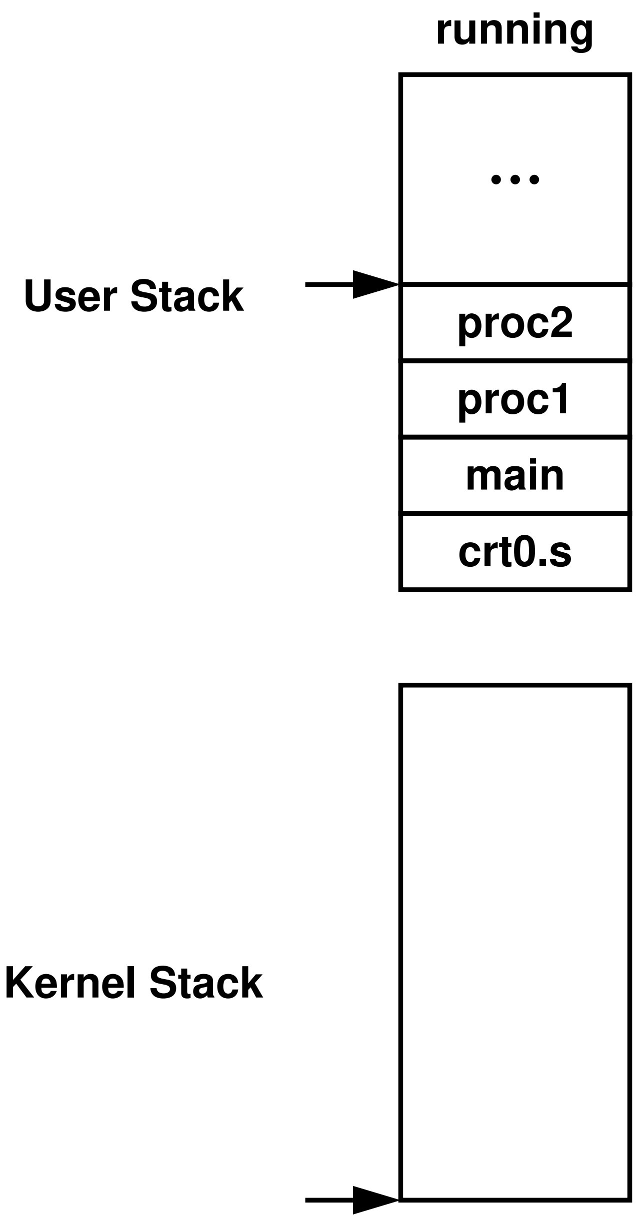
Interrupt Stack

- On most processors, a special hardware register points to an *interrupt stack*
  - when an interrupt or a trap causes a context switch into the kernel, the hardware changes the stack pointer to point to the interrupt stack
    - the hardware automatically saves some of the interrupted thread’s registers by pushing them onto the interrupt stack before calling the handler
- Why can’t you use the process’s user-level stack to store the saved state?
  - user-level stack pointer may be invalid (malicious user)
  - on a multi-processor system, another thread in the same process that runs in a different processor may modify the saved state

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

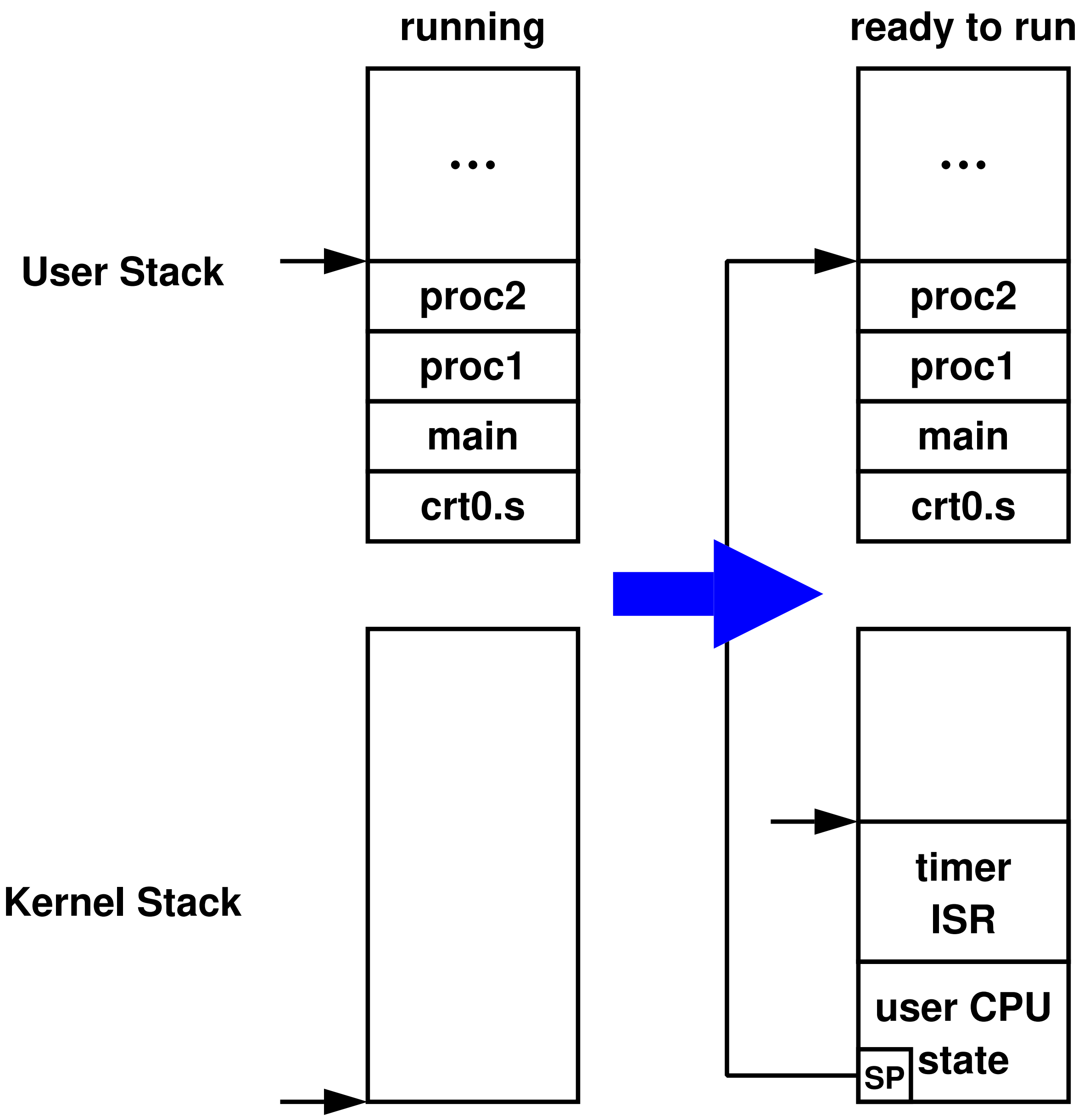


- `main()` is called by `crt0.s`
  - some would call `crt0.s` the *startup function*
  - the code for `crt0.s` is simply:

```
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



running

ready to run

User Stack

...

proc2

proc1

main

crt0.s

Kernel Stack

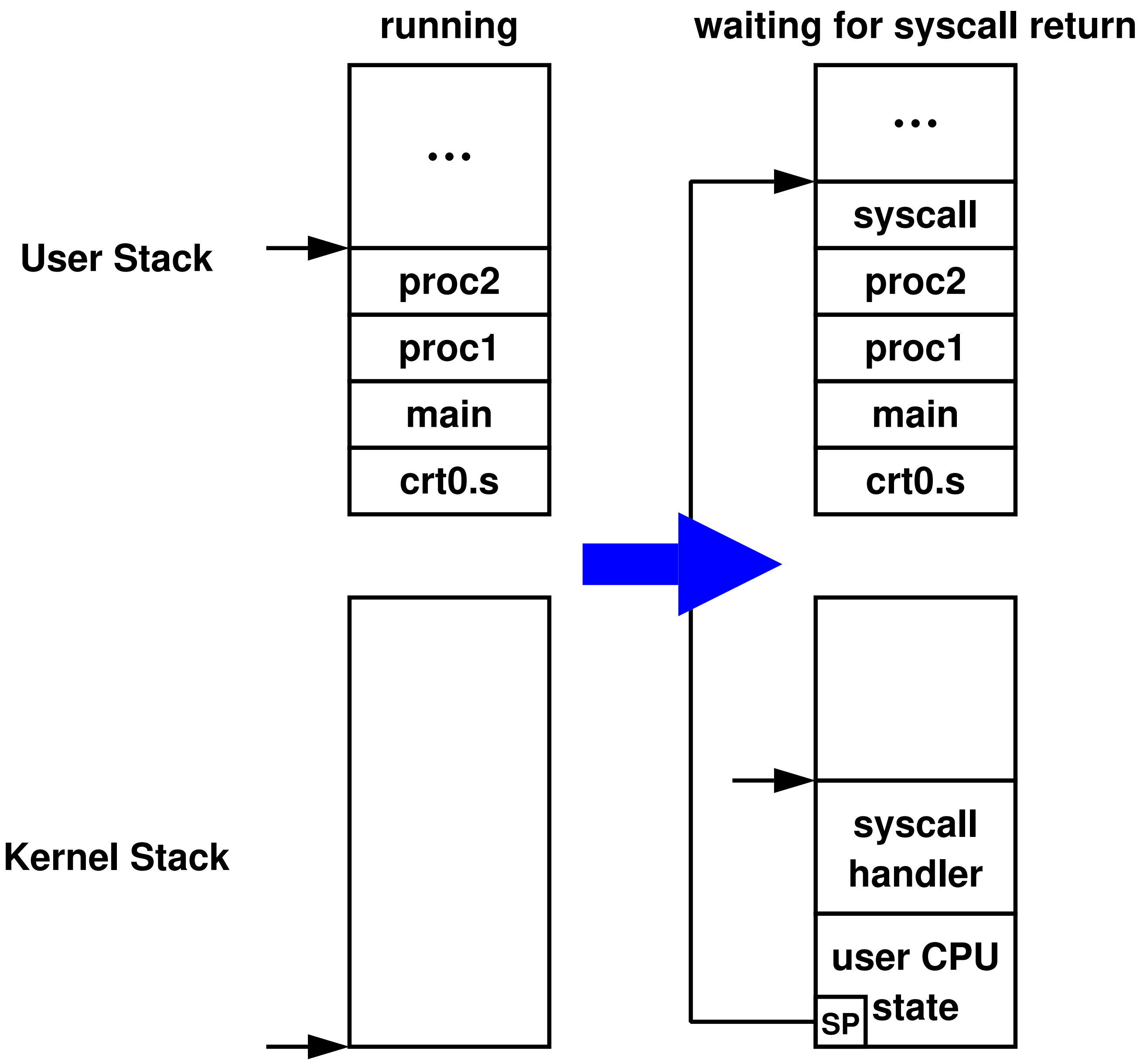
timer ISR

user CPU state

SP

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

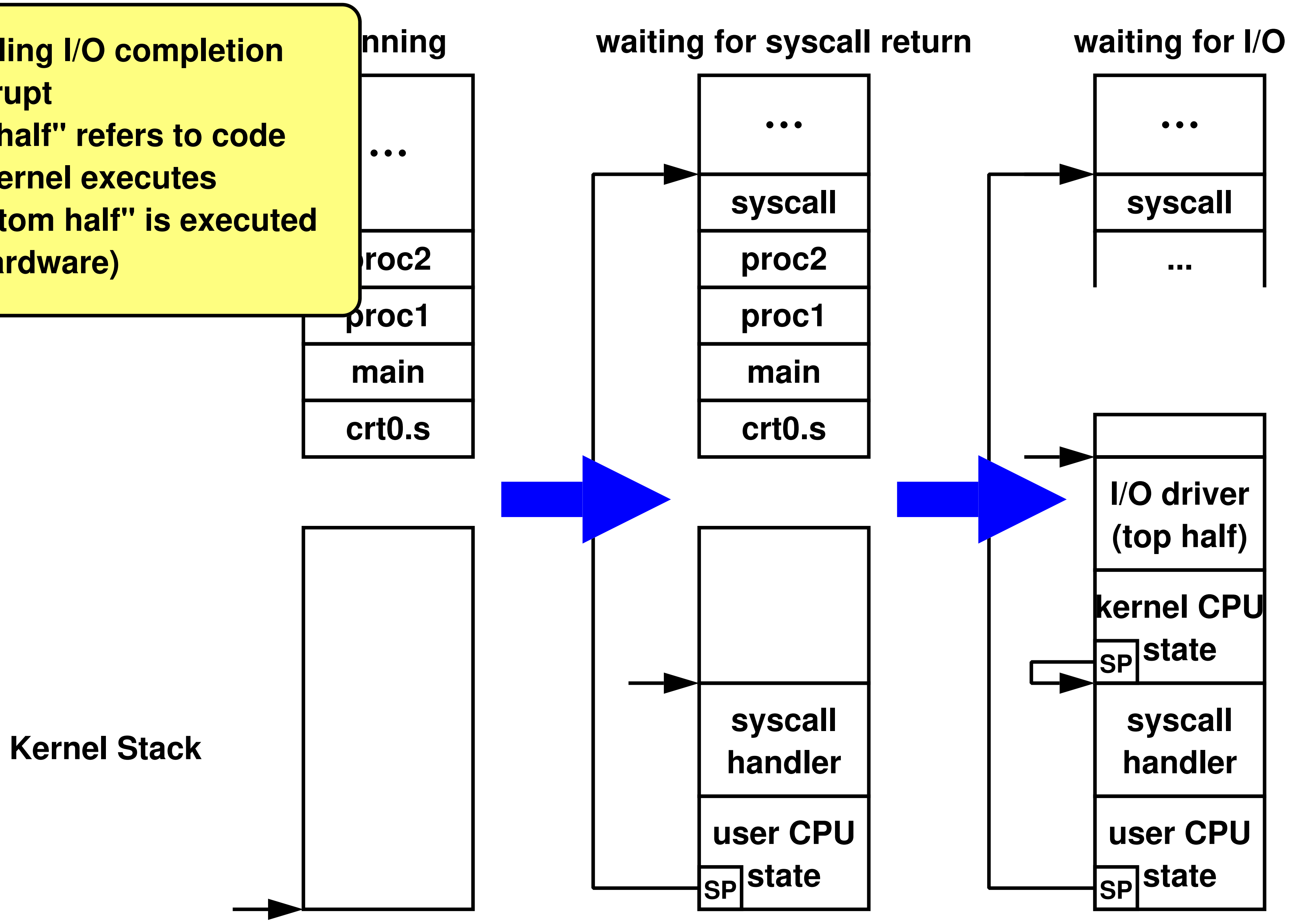
Two Stacks Per Process



- 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)



nning

waiting for syscall return

waiting for I/O

...

roc2

proc1

main

crt0.s

syscall

proc2

proc1

main

crt0.s

syscall

...

Kernel Stack

syscall handler

user CPU state

SP

I/O driver (top half)

kernel CPU state

SP

syscall handler

user CPU state

SP

Interrupt Masking

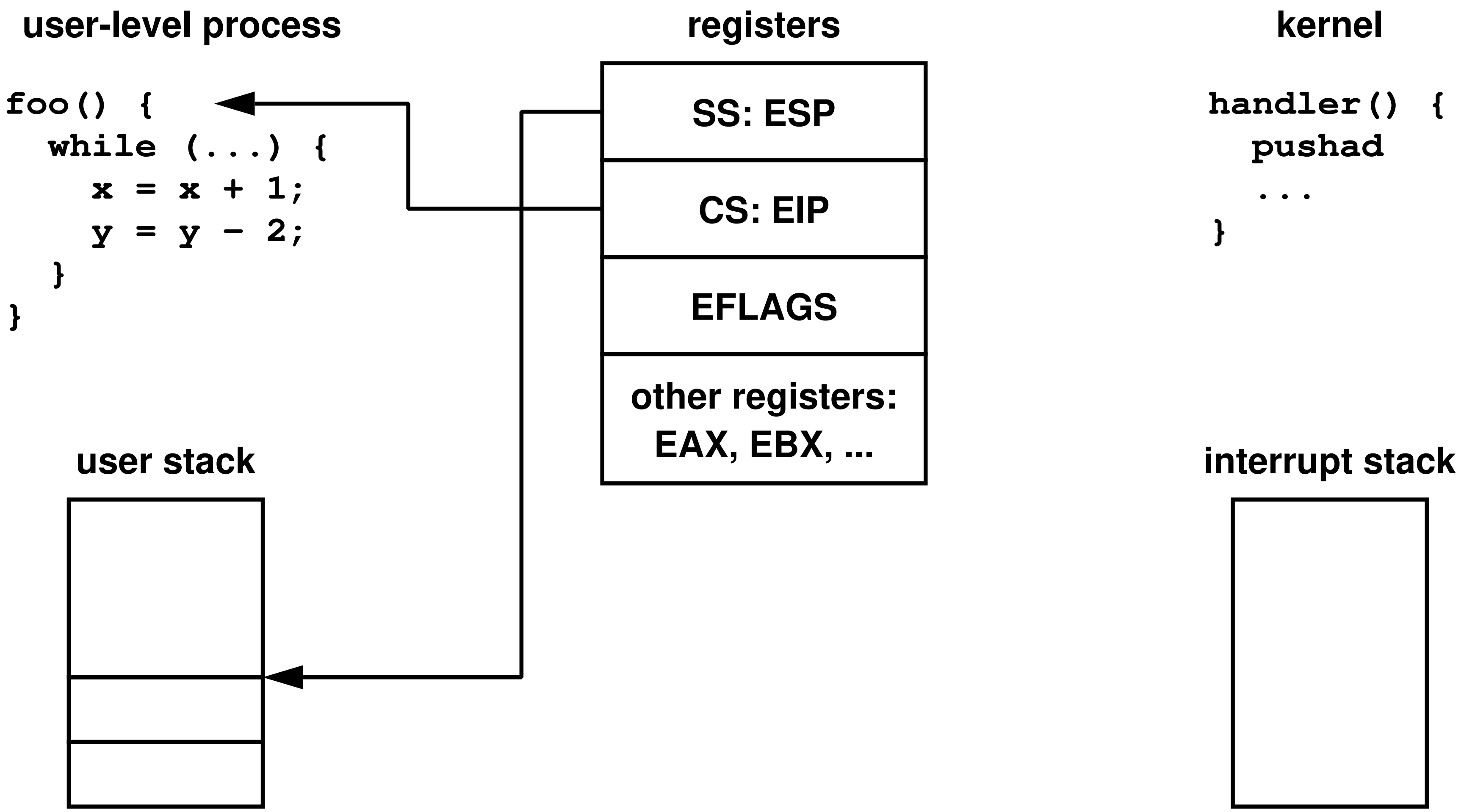
- Interrupt handler runs with interrupts disabled
  - re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - e.g., when determining the next process/thread to run
  - on x86
    - CLI: disable interrrupts
    - STI: enable interrrupts
    - only applies to the current CPU (on 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



user-level process

```
foo() {
  while (...) {
    x = x + 1;
    y = y - 2;
  }
}
```

user stack

registers

SS: ESP

CS: EIP

EFLAGS

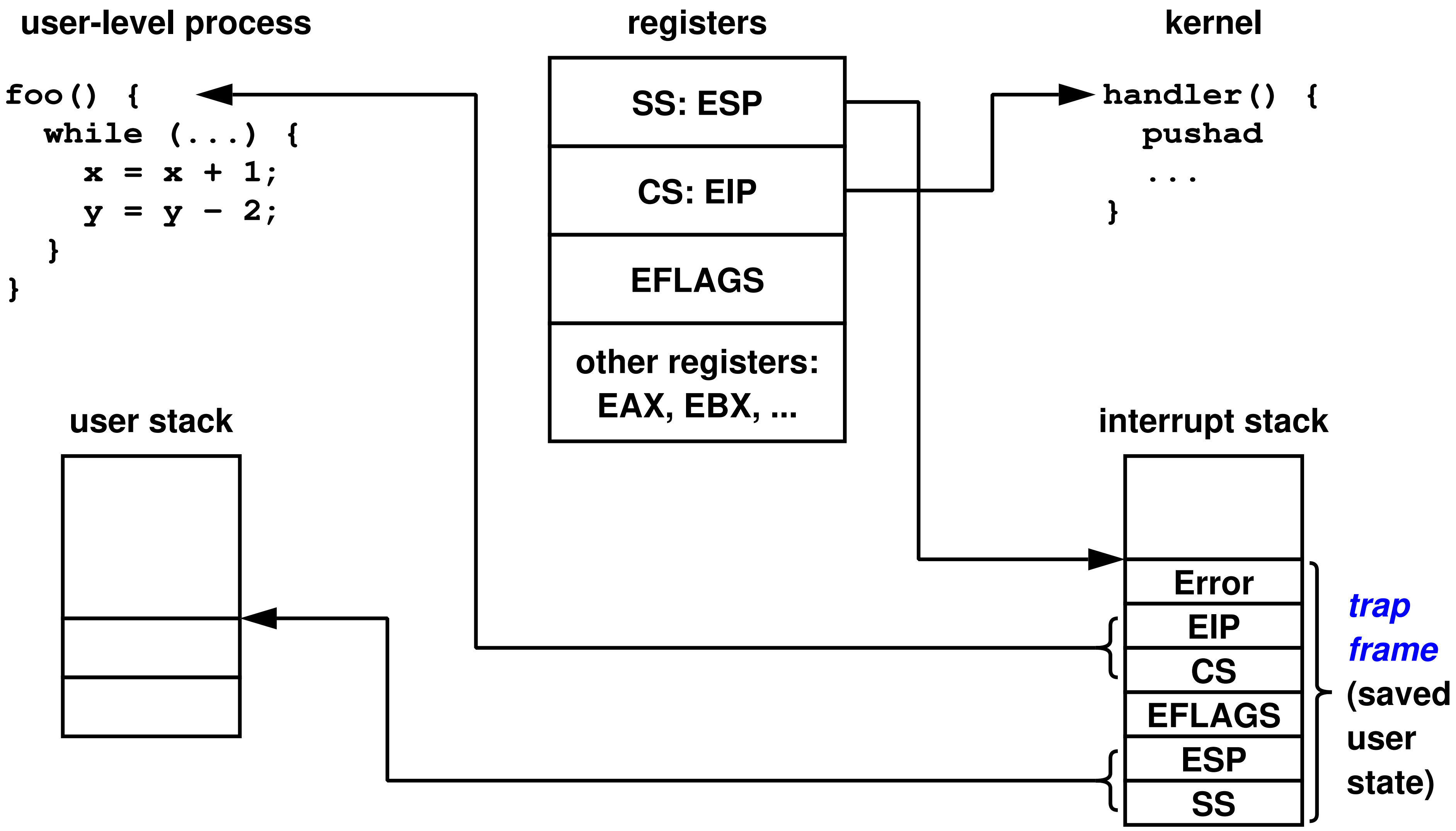
other registers:
EAX, EBX, ...

kernel

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

interrupt stack

Got Interrupt



user-level process

```
foo() {
  while (...) {
    x = x + 1;
    y = y - 2;
  }
}
```

registers

- SS: ESP
- CS: EIP
- EFLAGS
- other registers: EAX, EBX, ...

kernel

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

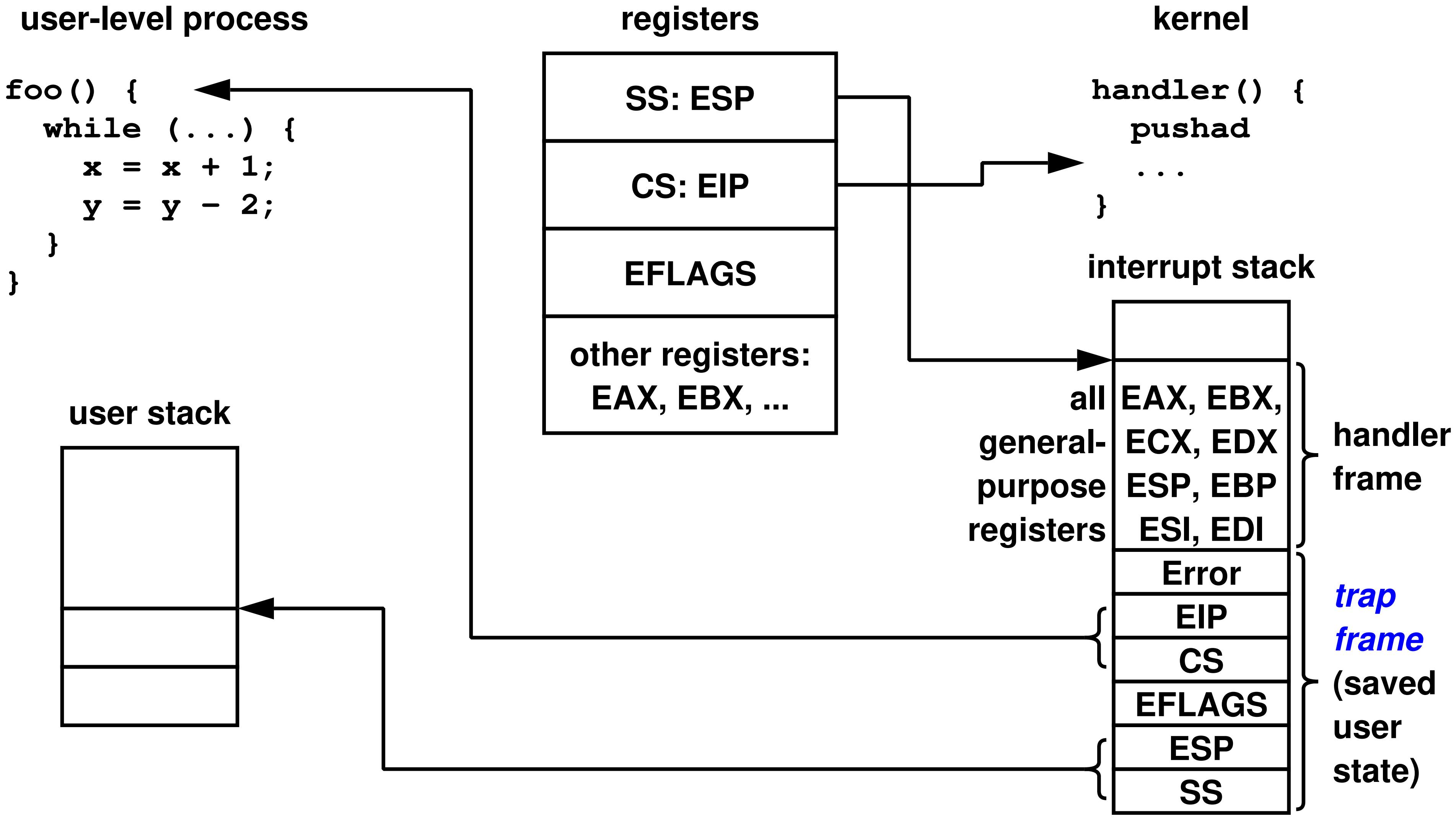
user stack

interrupt stack

- Error
- EIP
- CS
- EFLAGS
- ESP
- SS

trap frame (saved user state)

After Interrupt Handler Starts



user-level process

```
foo() {
  while (...) {
    x = x + 1;
    y = y - 2;
  }
}
```

user stack

registers

SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...

kernel

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

interrupt stack

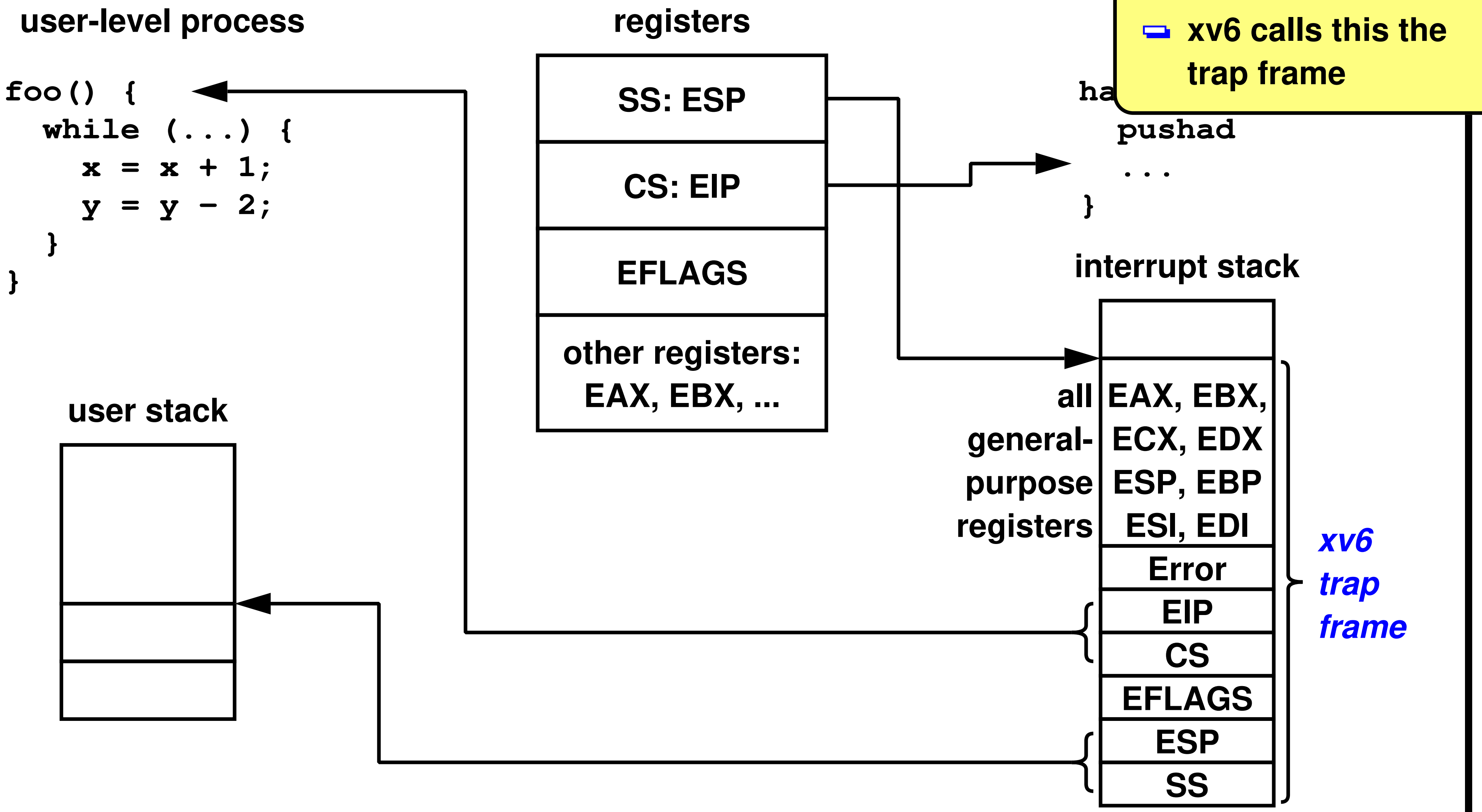
all general-purpose registers	EAX, EBX, ECX, EDX ESP, EBP ESI, EDI
	Error
	EIP
	CS
	EFLAGS
	ESP
	SS

handler frame

trap frame (saved user state)

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

After Interrupt Handler Starts



- xv6 calls this the trap frame

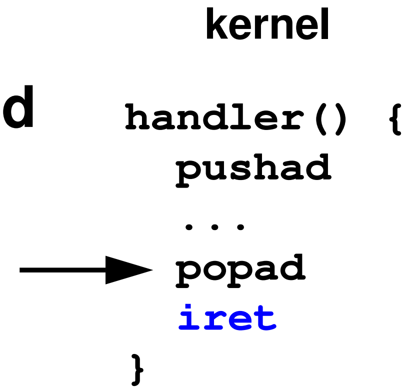
```
foo() {
  while (...) {
    x = x + 1;
    y = y - 2;
  }
}
```

```
ha
  pushad
  ...
}
```

- **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
  iret
}
```

(2.6) Implementing Secure System Calls

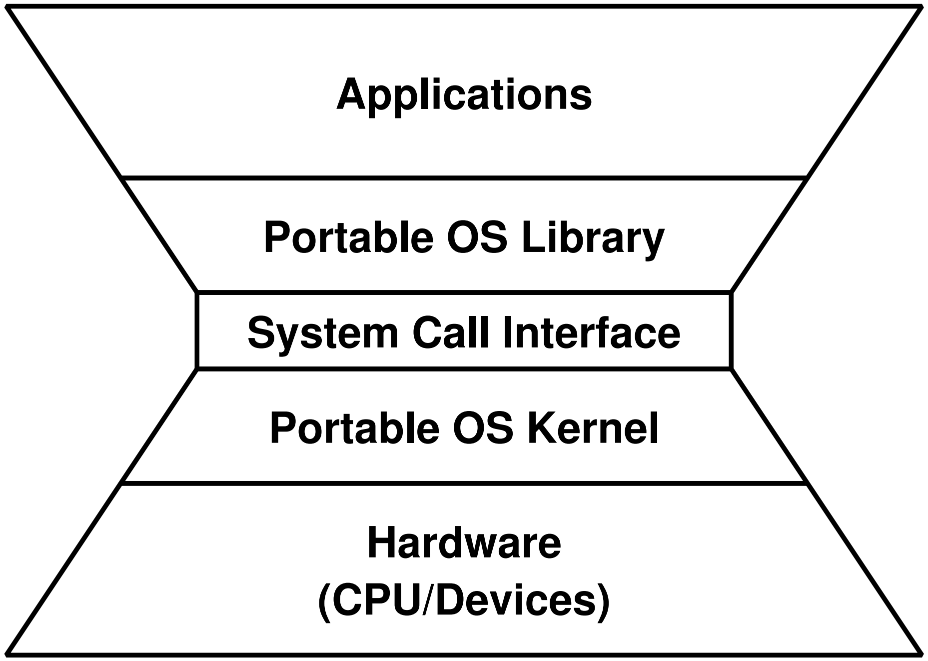
Implementing Secure System Calls

- 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

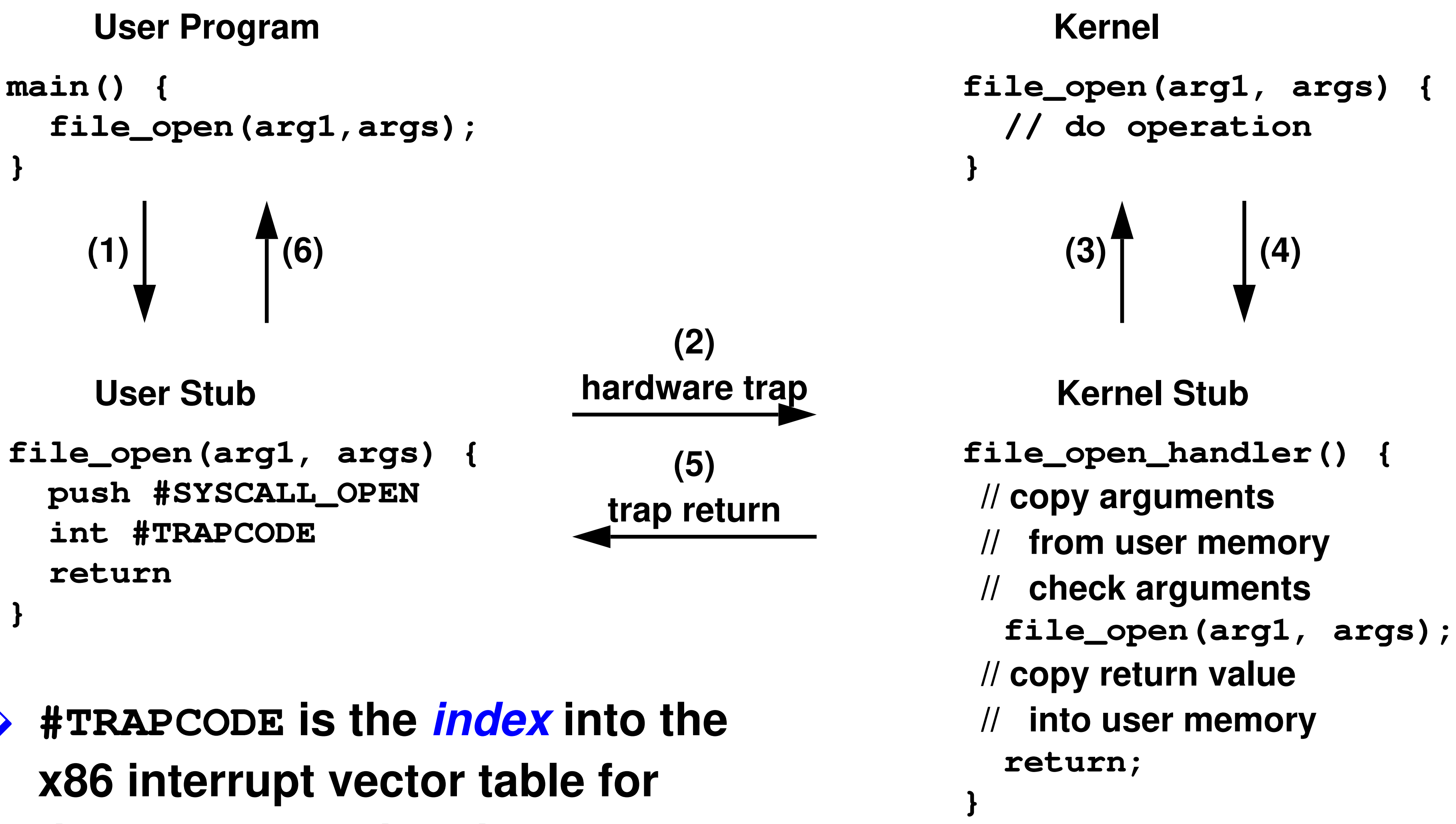
- All system calls follow a certain calling convention
  - e.g., how to name them, how to pass arguments, how to receive return values
  - information is passed in registers and on the executiong stack
  - a trap instruction is eventually invoked to transfer control to the kernel
    - for x86, the machine 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);
}
```

Kernel

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

User Stub

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

(2)
hardware trap

(5)
trap return

Kernel Stub

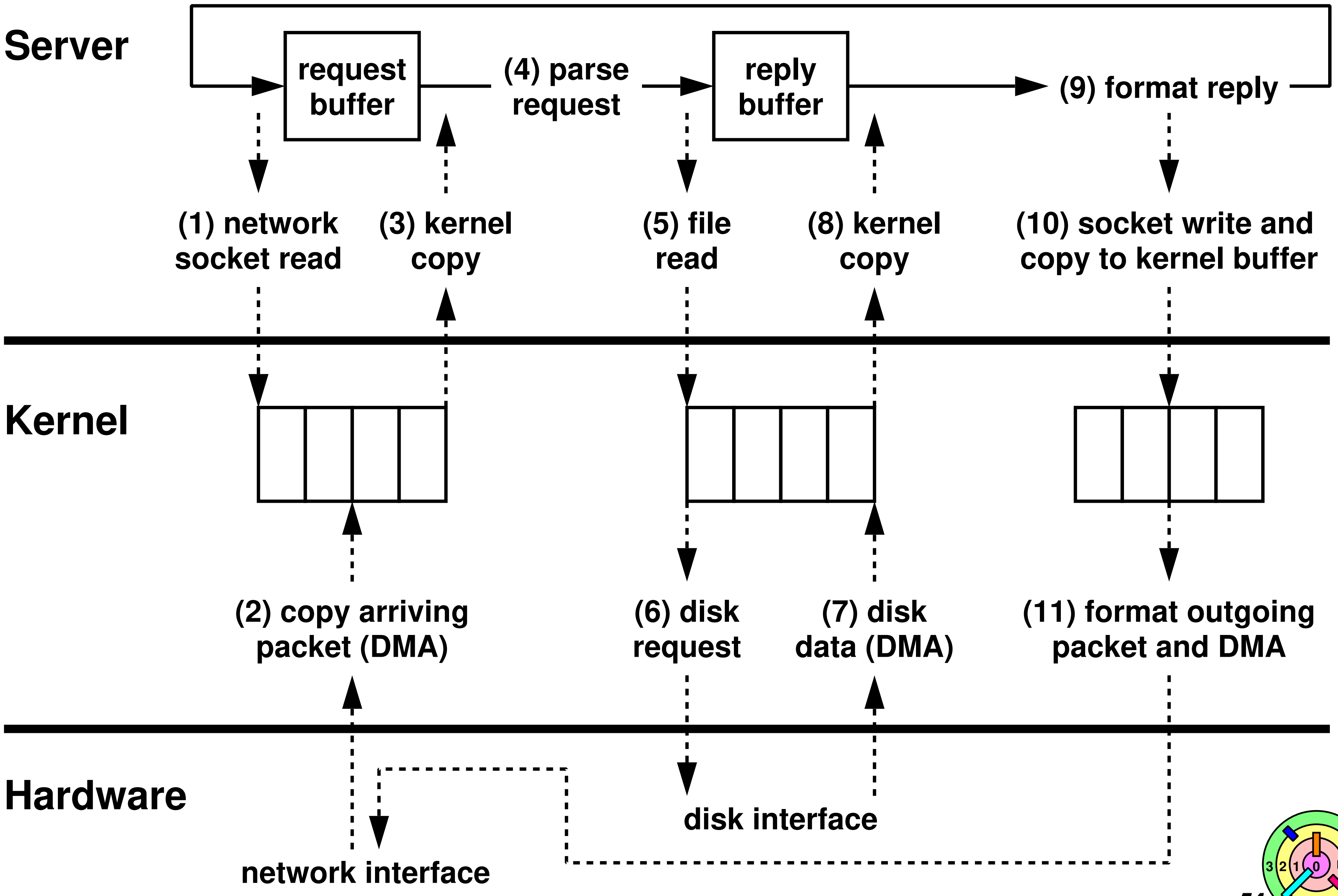
```
file_open_handler() {
 // copy arguments
 //   from user memory
 //   check arguments
  file_open(arg1, args);
 // copy return value
 //   into user memory
  return;
}
```

- `#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

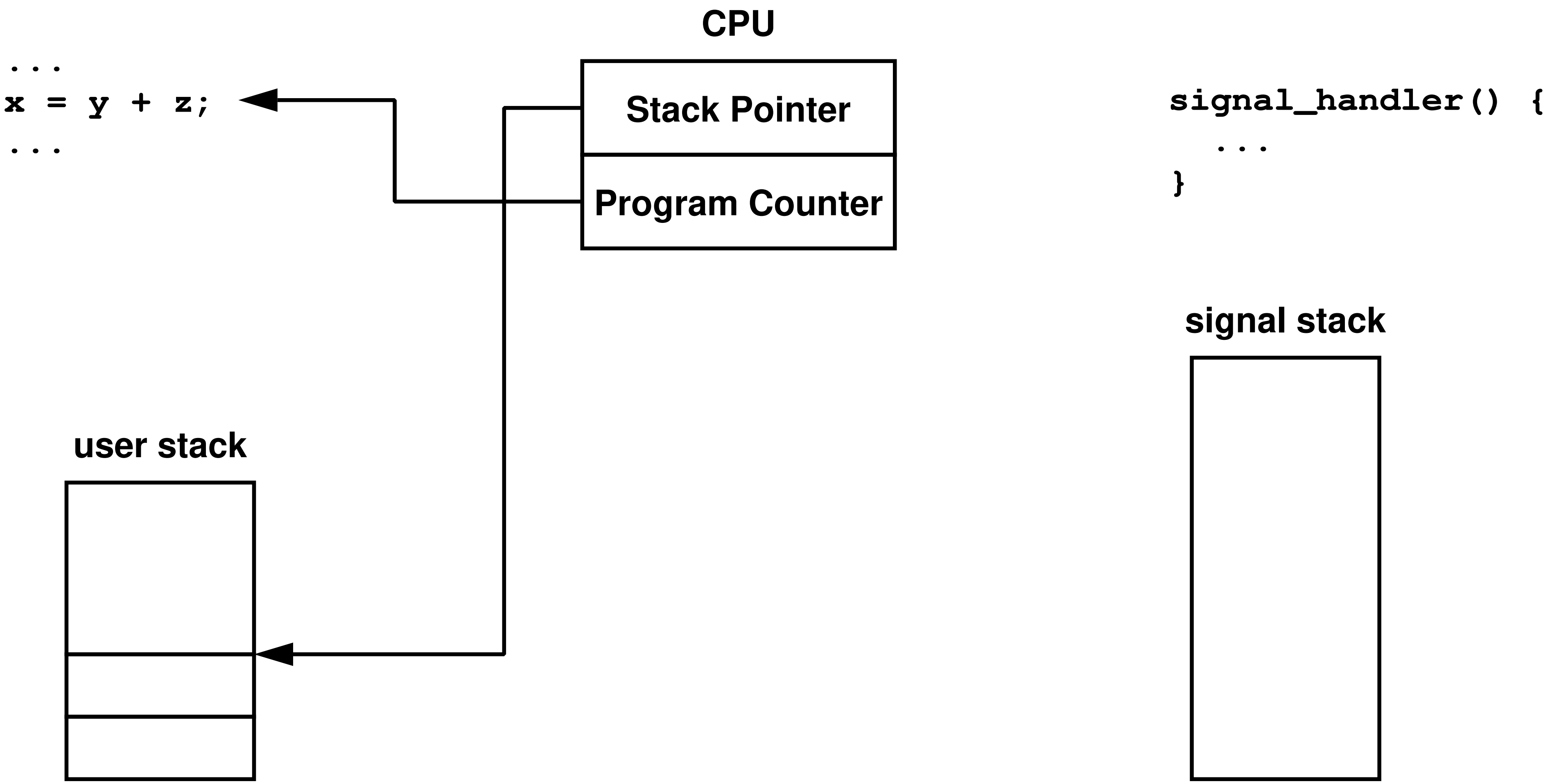
- There are several uses for immediate event delivery with upcalls
  - preemptive user-level threads (e.g., timer upcall)
  - asynchronous I/O notification (e.g., I/O completion upcall)
    - used in *asynchronous I/O*
  - interprocess communication (e.g., debugger upcall to suspend or resume a process, 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



```
...
x = y + z;
...
```

CPU

Stack Pointer

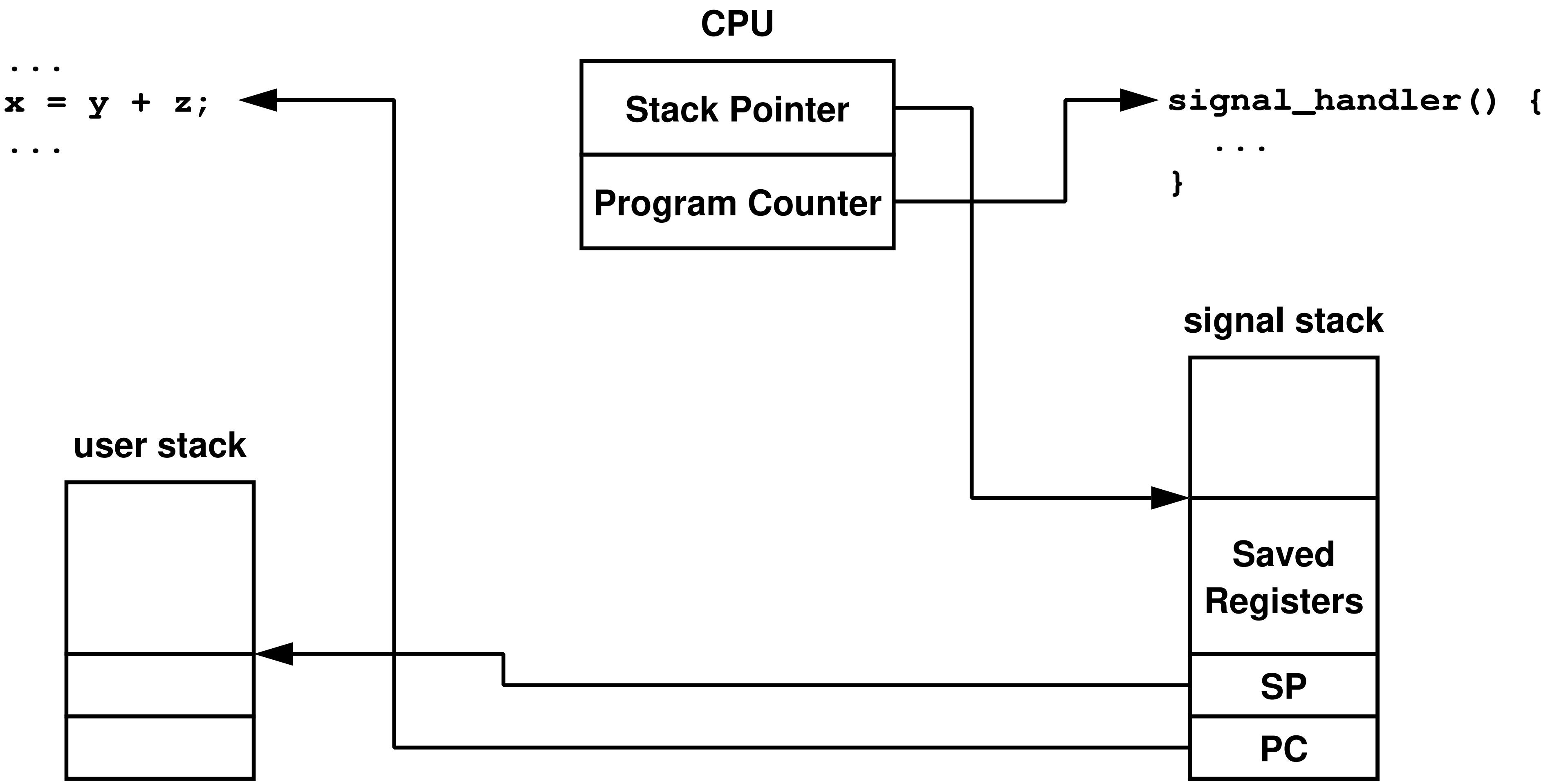
Program Counter

```
signal_handler() {
   ...
}
```

signal stack

user stack

Upcall: During



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

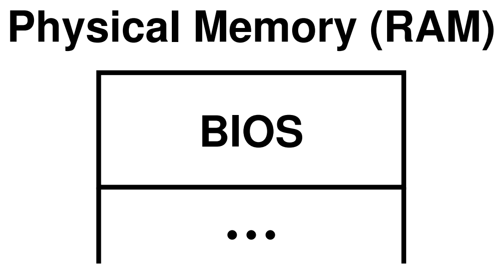
Upcall: Implementation

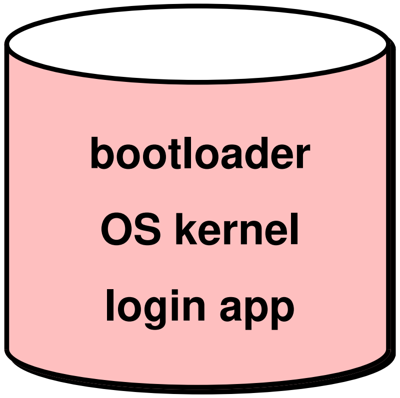
- To implement upcall only requires a small modification to the "return from system call" or "return from interrupt" mechanism
  - e.g., timer interrupt upcall
    - the hardware and the interrupt handler save the state of the user-level computation
    - kernel copies save state to the bottom of the 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

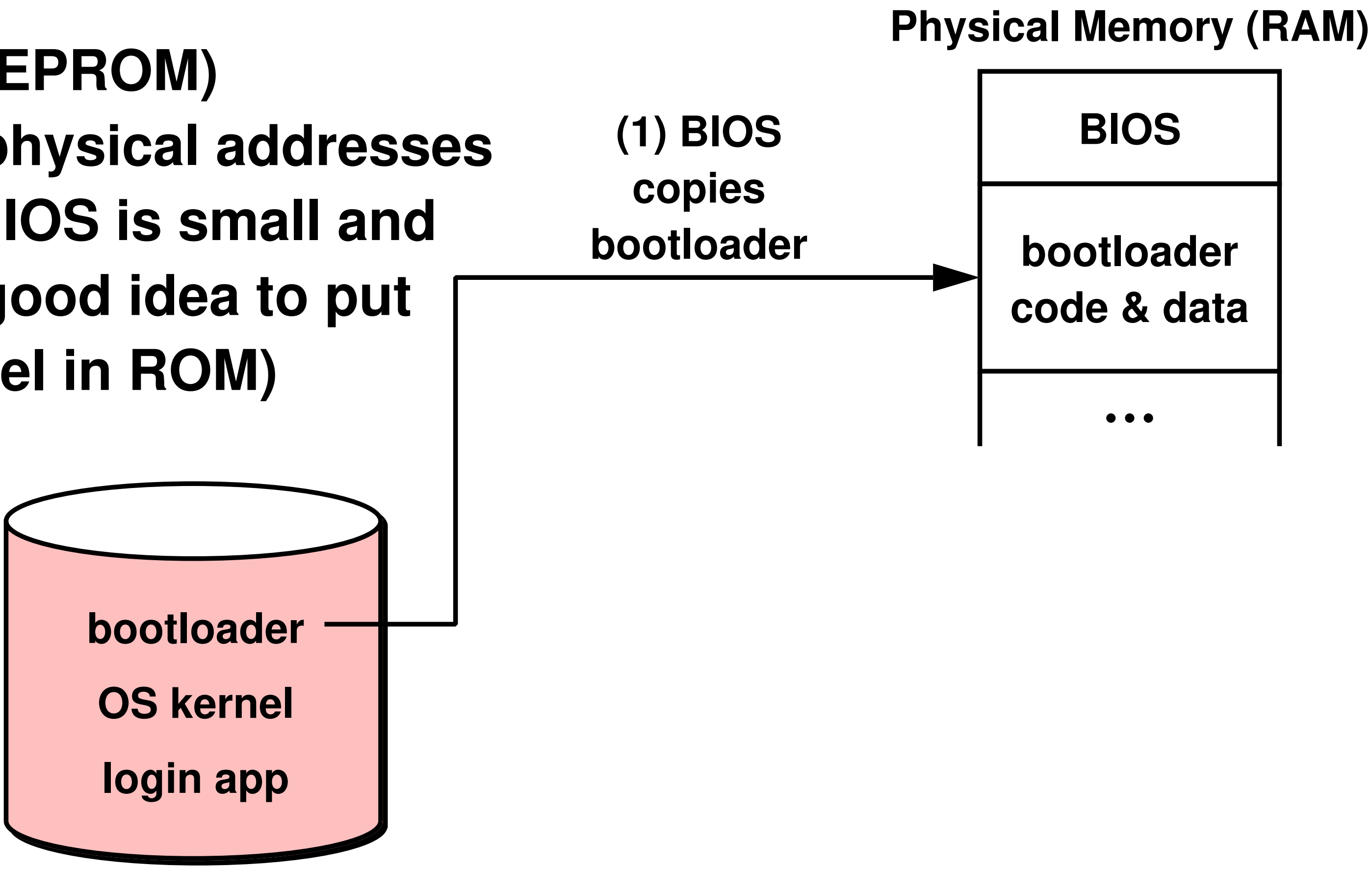
- How does the OS *bootstrap* itself?
  - Ex: running Linux or Windows on a PC
- BIOS in ROM (or EPROM)
  - accessed via physical addresses
  - boot code in BIOS is small and simple (not a good idea to put the entire kernel in ROM)





Case Study: Booting An OS Kernel

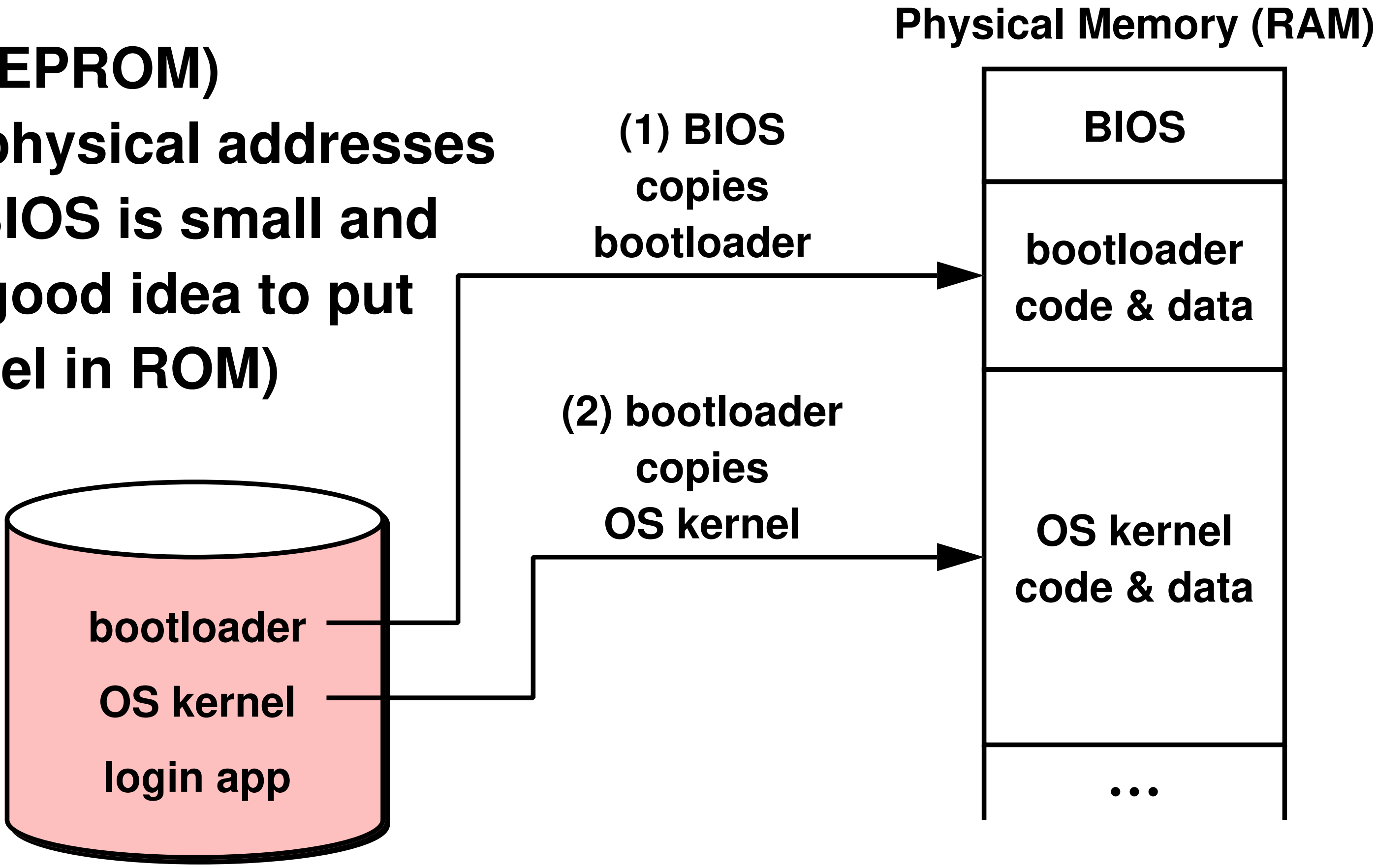
- How does the OS *bootstrap* itself?
  - Ex: running Linux or Windows on a PC
- BIOS in ROM (or EPROM)
  - accessed via physical addresses
  - boot code in BIOS is small and simple (not a good idea to put the entire kernel in ROM)



- 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

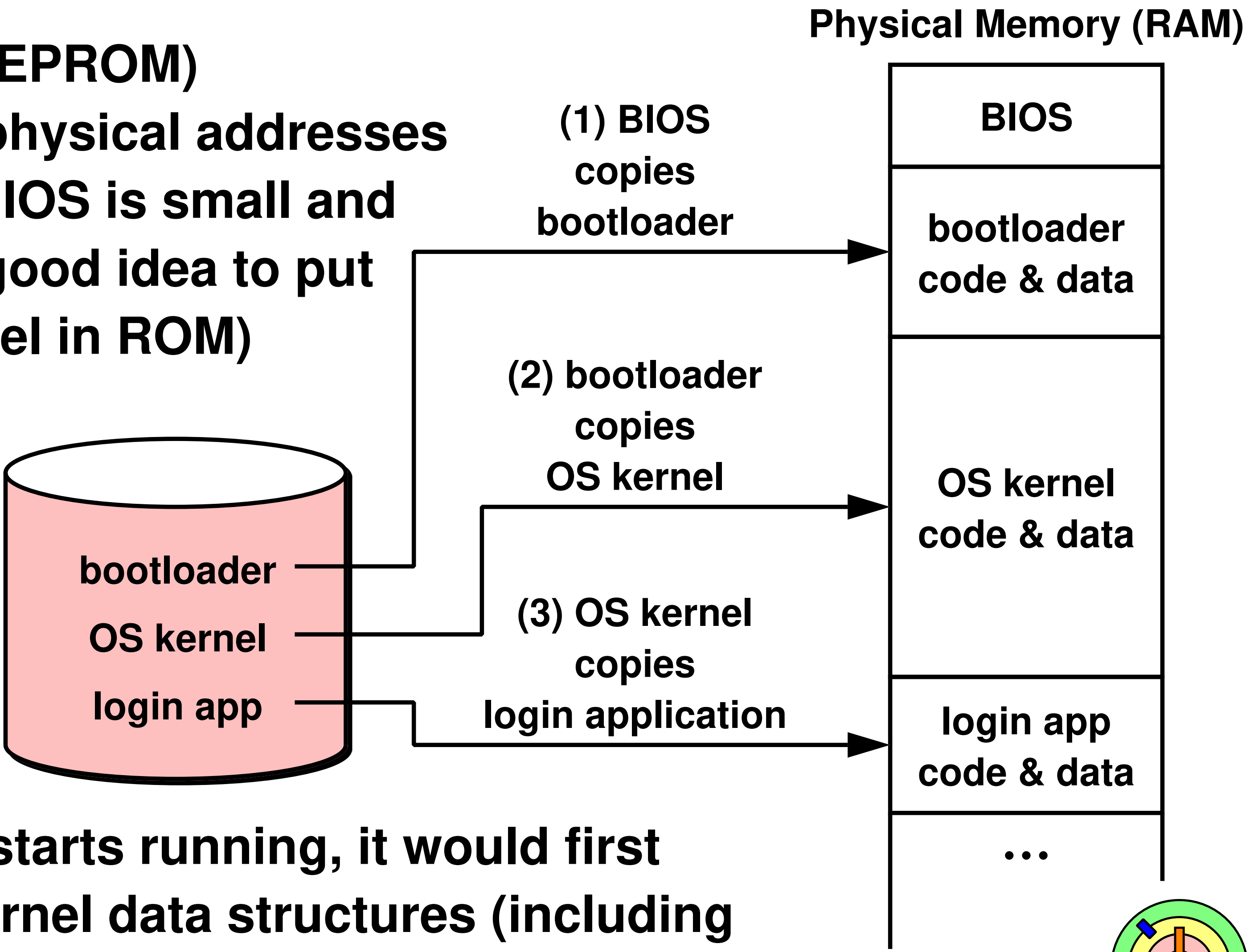
- How does the OS *bootstrap* itself?
  - Ex: running Linux or Windows on a PC
- BIOS in ROM (or EPROM)
  - accessed via physical addresses
  - boot code in BIOS is small and simple (not a good idea to put the entire kernel in ROM)



- 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

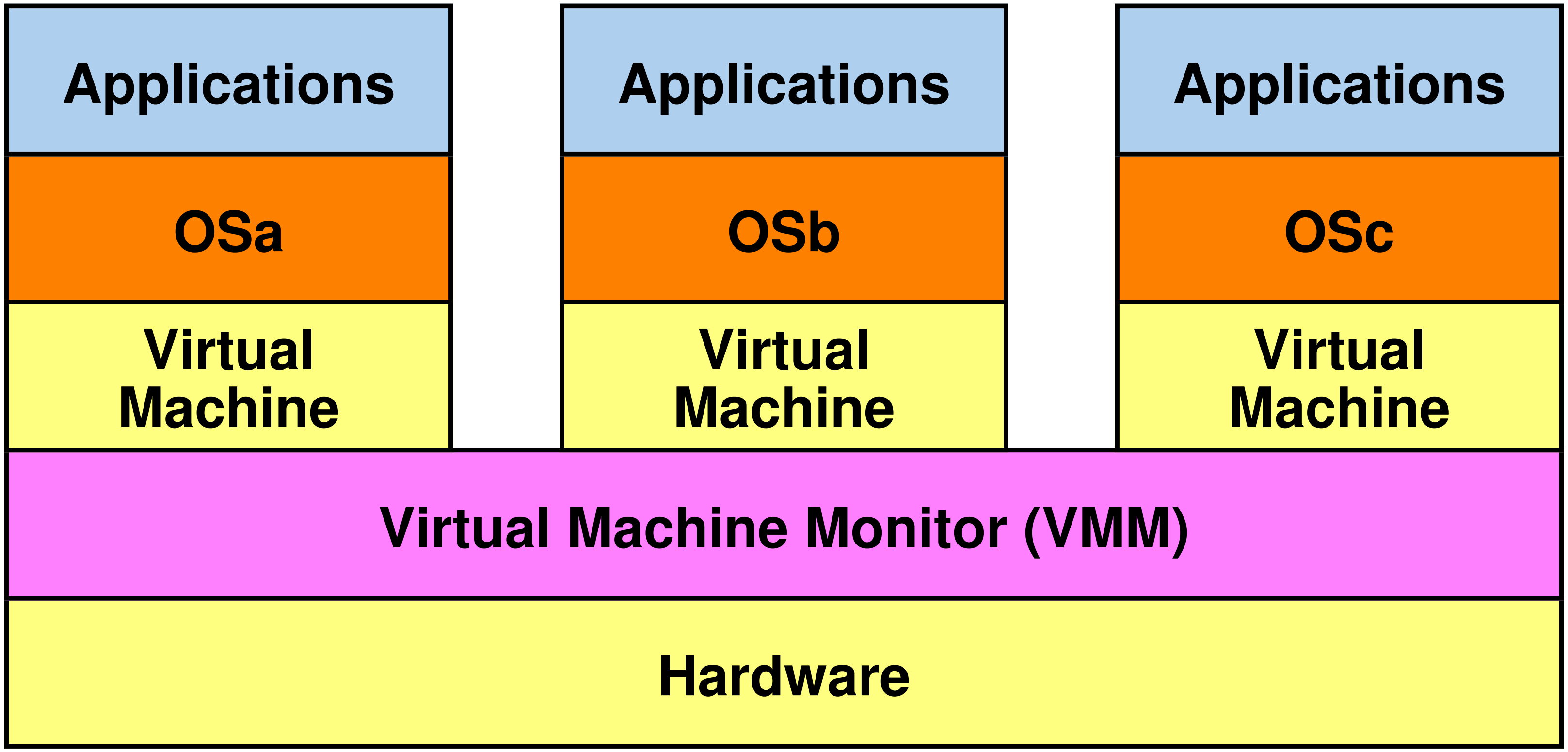
- OS kernel needs to communicate with physical devices
- Devices operate asynchronously from the CPU
    - polling: kernel polls to see if I/O is done
    - interrupts: kernel can do other work in the meantime
- Device access to memory
    - Programmed I/O (PIO): CPU reads and writes to device
    -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

(2.10) Case Study: Virtual Machines

Case Study: Virtual Machines

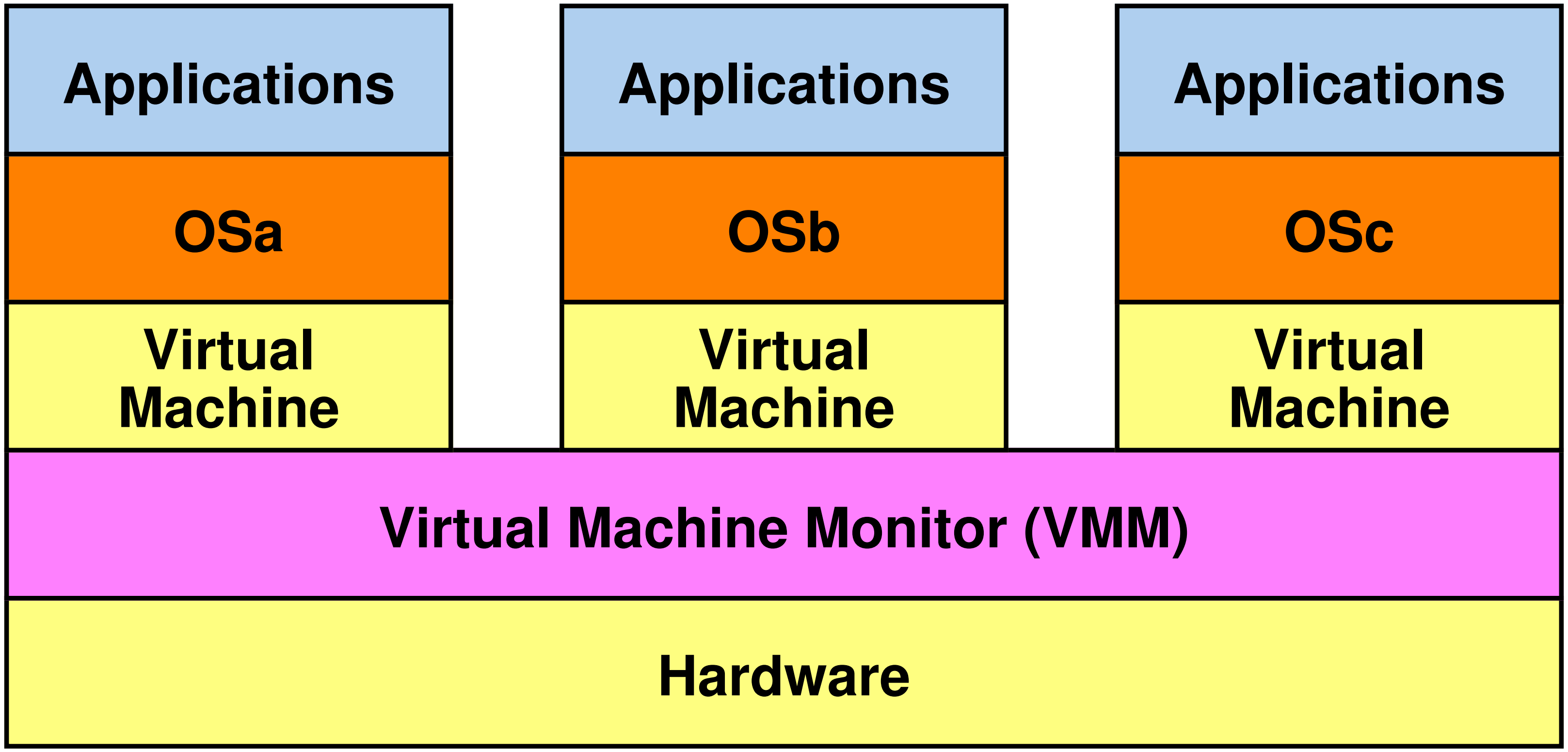
- In the 60s, IBM had a *single-user time-sharing system* called CMS
  - IBM wants to build a *multiuser time-sharing system*
- TSS (Time-Sharing System) project
  - it's a very difficult system to build
  - large, monolithic system
  - lots of people working on it
  - for years
  - total, complete flop
- 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



Applications

Applications

Applications

OSa

OSb

OSc

Virtual Machine

Virtual Machine

Virtual Machine

Virtual Machine Monitor (VMM)

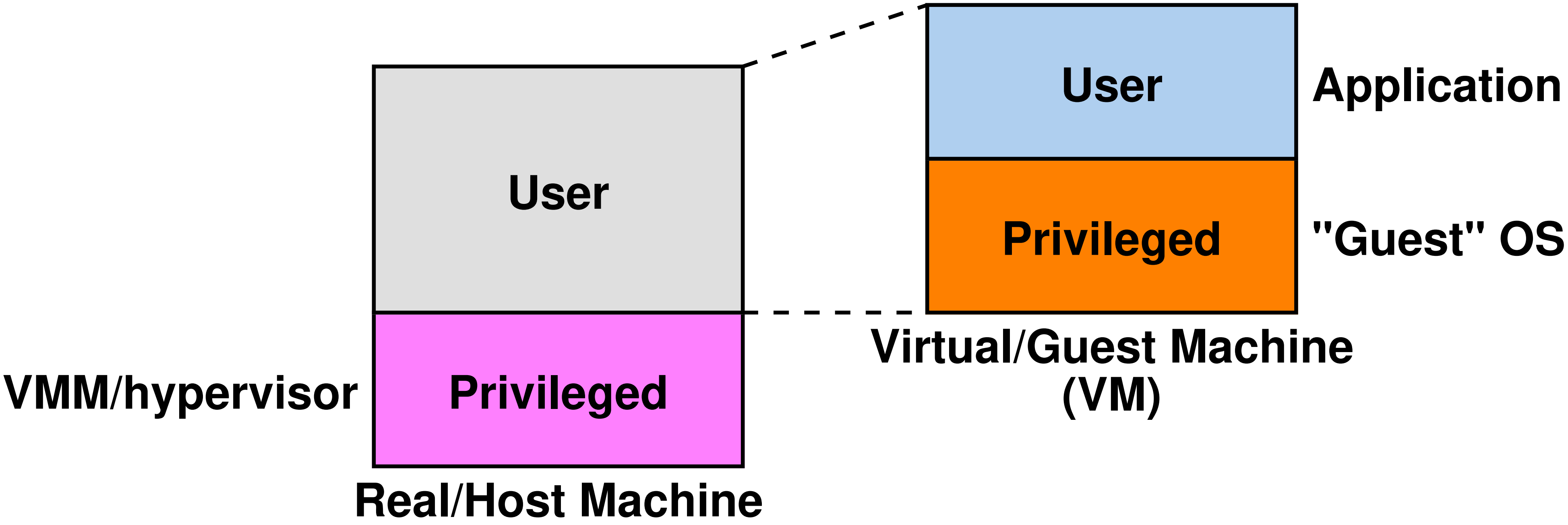
Hardware

- **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

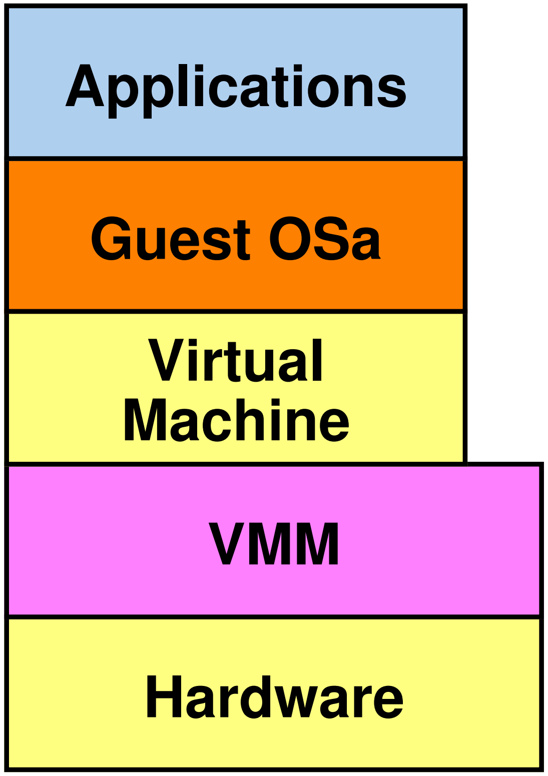
- Virtual Machine: run one OS inside (or on-top-of) another OS
  - *run* (not emulate/simulate) OSx on-top-of OSy
    - we will refer to OSx as the *guest OS* and OSy as the *host OS* (a host can have multiple guests)
    - a virtual machine is *not* an OS *emulator*
      - must *execute* guest OS code 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)

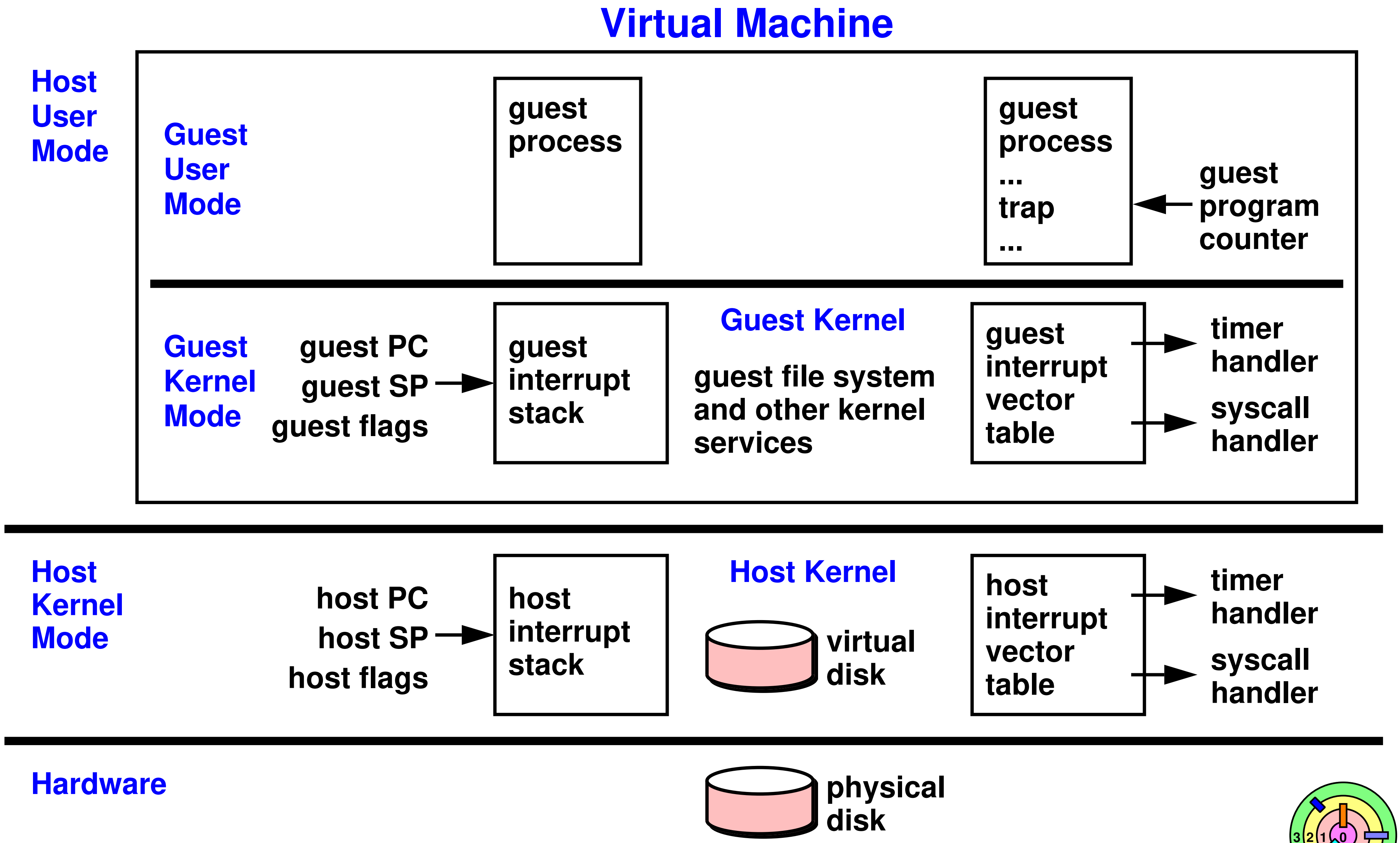
Virtual Machines: Mode Transfer Example #2

- **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 Machine

Host User Mode

Guest User Mode

guest process

guest process … trap …

guest program counter

Guest Kernel Mode

guest PC

guest SP

guest flags

guest interrupt stack

Guest Kernel

guest file system and other kernel services

guest interrupt vector table

timer handler

syscall handler

Host Kernel Mode

host PC

host SP

host flags

host interrupt stack

Host Kernel

virtual disk

host interrupt vector table

timer handler

syscall handler

Hardware

physical disk

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

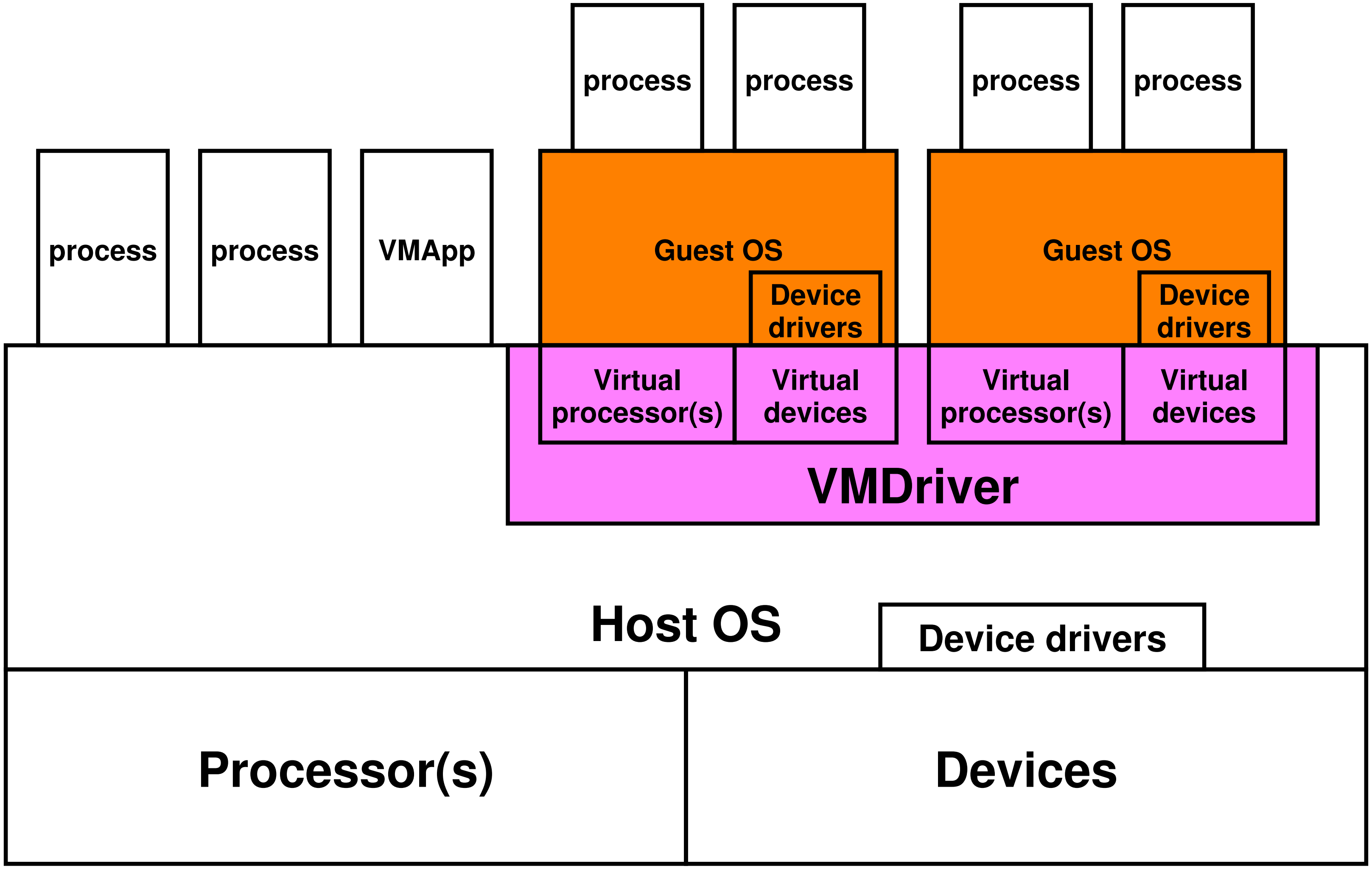
Virtual Machines: I/O Interrupt Handling

- 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

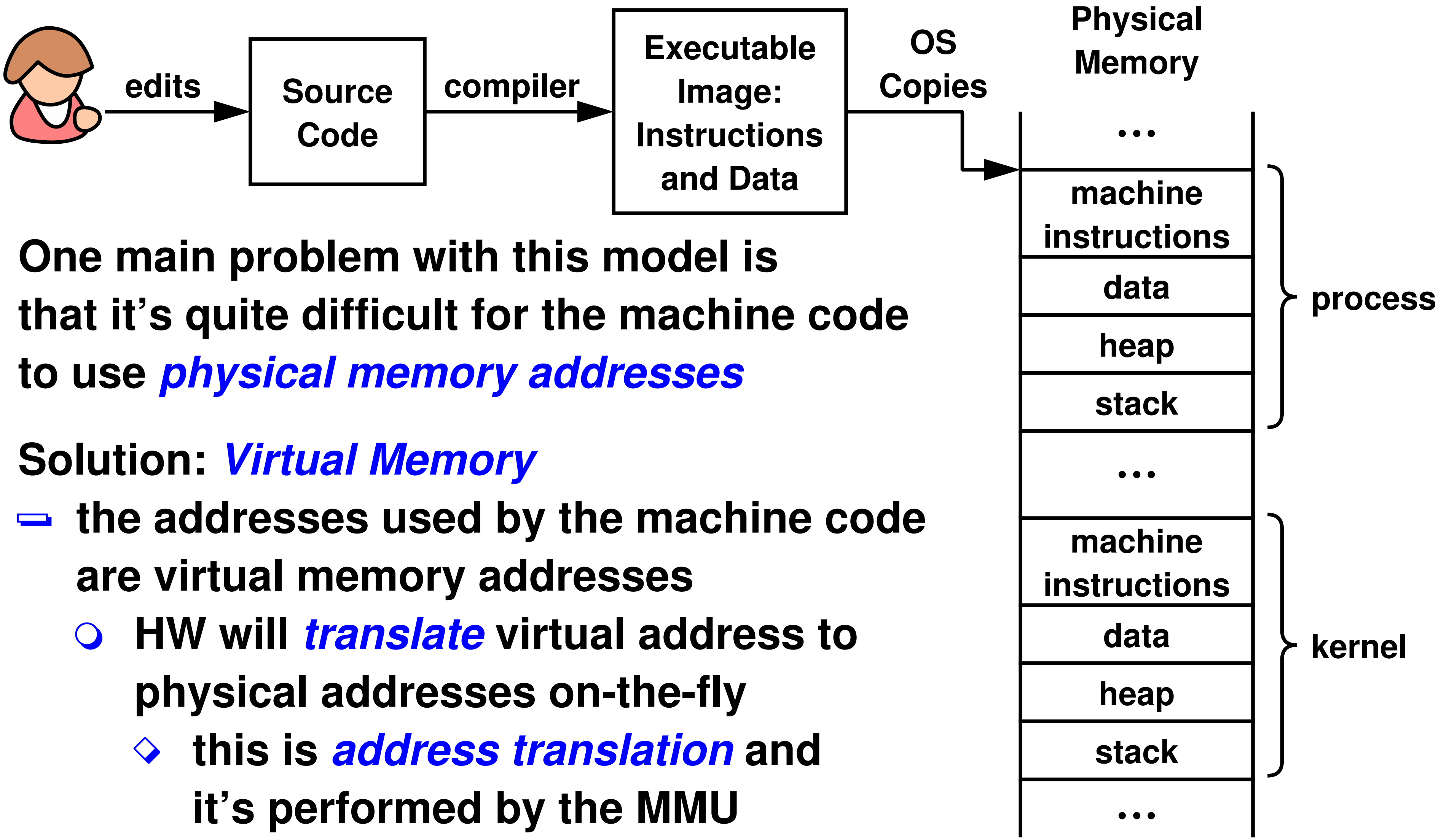


Extra Slides

Challenge: Protection

- How do we execute code with restricted privileges?
  - either because the code is buggy or if it might be malicious
- Some examples:
  - a script running in a web browser
  - a program you just downloaded off the Internet
  - a program you just wrote that you haven’t tested yet

Edit, Compile, Load, and Run



edits

Source Code

compiler

Executable Image: Instructions and Data

OS Copies

Physical Memory

...

machine instructions

data

heap

stack

...

machine instructions

data

heap

stack

...

process

kernel

- One main problem with this model is that it's quite difficult for the machine code to use *physical memory addresses*
- Solution: *Virtual Memory*
  - the addresses used by the machine code are virtual memory addresses
    - HW will *translate* virtual address to physical addresses on-the-fly
      - this is *address translation* and it's performed by the MMU
  - both the kernel and the application uses virtual addresses to run code and access data

Booting

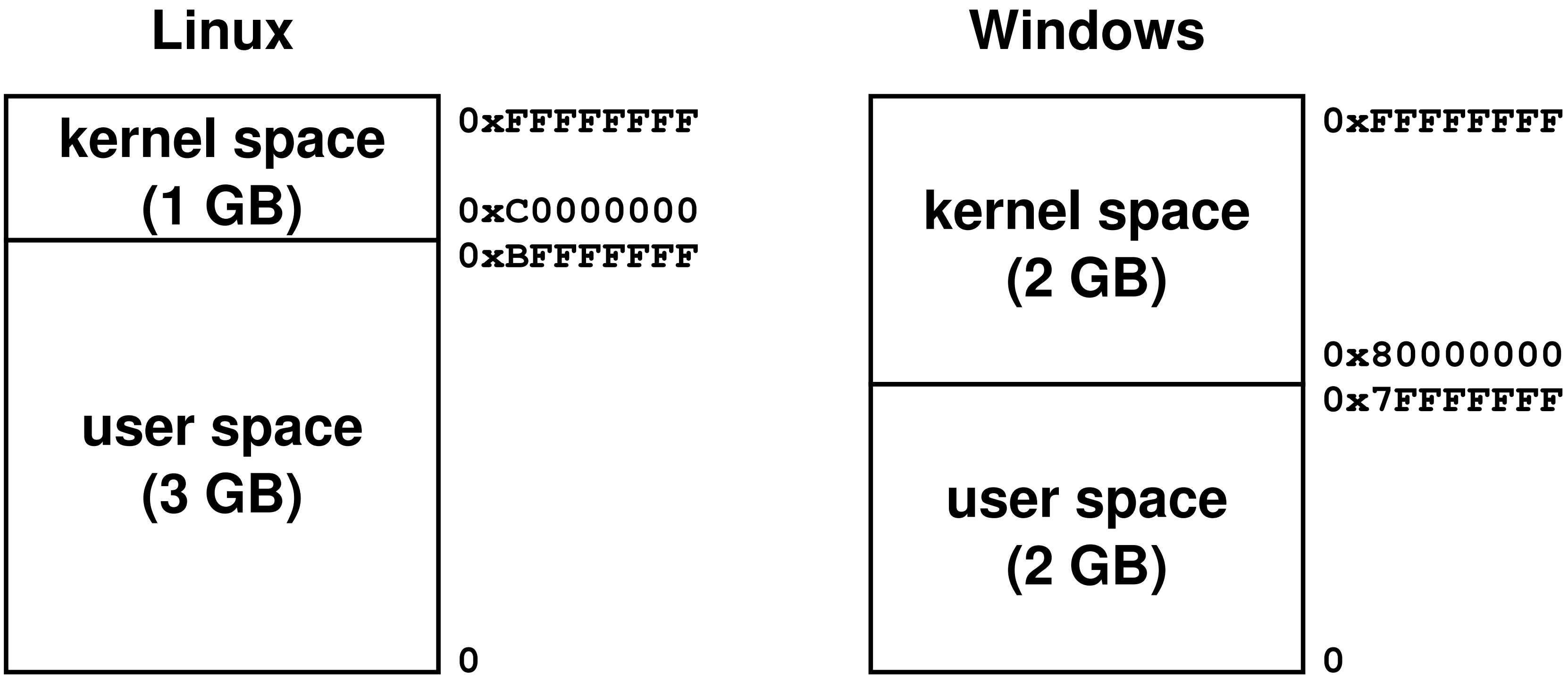
- OS kernel needs to communicate with physical devices
- Devices operate asynchronously from the CPU
  - polling: kernel polls to see if I/O is done
  - interrupts: kernel can do other work in the meantime
- Device access to memory
  - Programmed I/O (PIO): CPU reads and writes to device
  - Direct Memory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

(Virtual) Address Space Abstraction

- *Abstraction* of physical memory
- Goal of address space abstraction:
  - give each process a private memory area for code, data, stack
  - prevent one process from reading/writing outside its address space
  - allow sharing, when needed
- The kernel has its own address space

Address Space: Implementation

- Usually the implementation is split between the OS and HW
  - OS manages address spaces
    - allocates physical memory (for creation, growth, deletion)
  - HW performs *address translation* and *protection*
    - translates user addresses to physical addresses



The Process Abstraction

- Process ≠ Program
- Process is an instance of a program, running with lmited rights
- Process consists of two parts
    1) address space which contains *memory segments*
        - code
        - data
        - heap
        - stack
        - kernel stack
    2) process control block (PCB) which contains all the information the kernel needs about the process, e.g.,
        - where the address space is stored in memory
        - where the executable image resides on disk
        - which user the process belongs to
        - what privileges the process has
        - etc.

The Process Abstraction

- **Process ≠ Program**
- **Process is an instance of a program, running with lmited rights**
- **Process consists of two parts**
  1) address space which contains *memory segments*
     - code
     - data
     - heap
     - stack

     contents of these are stored in user space

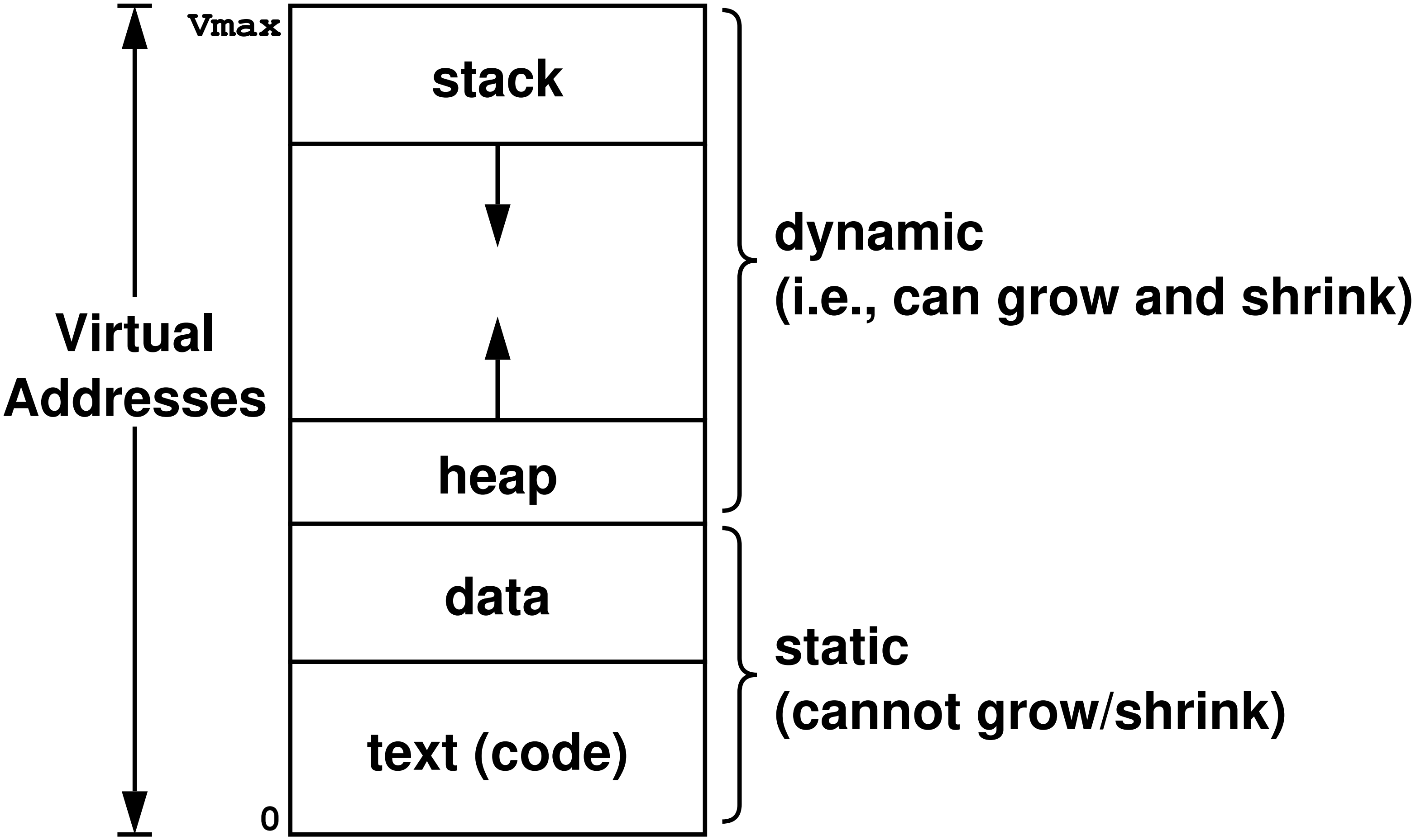
     - kernel stack

     contents of these are stored in kernel space

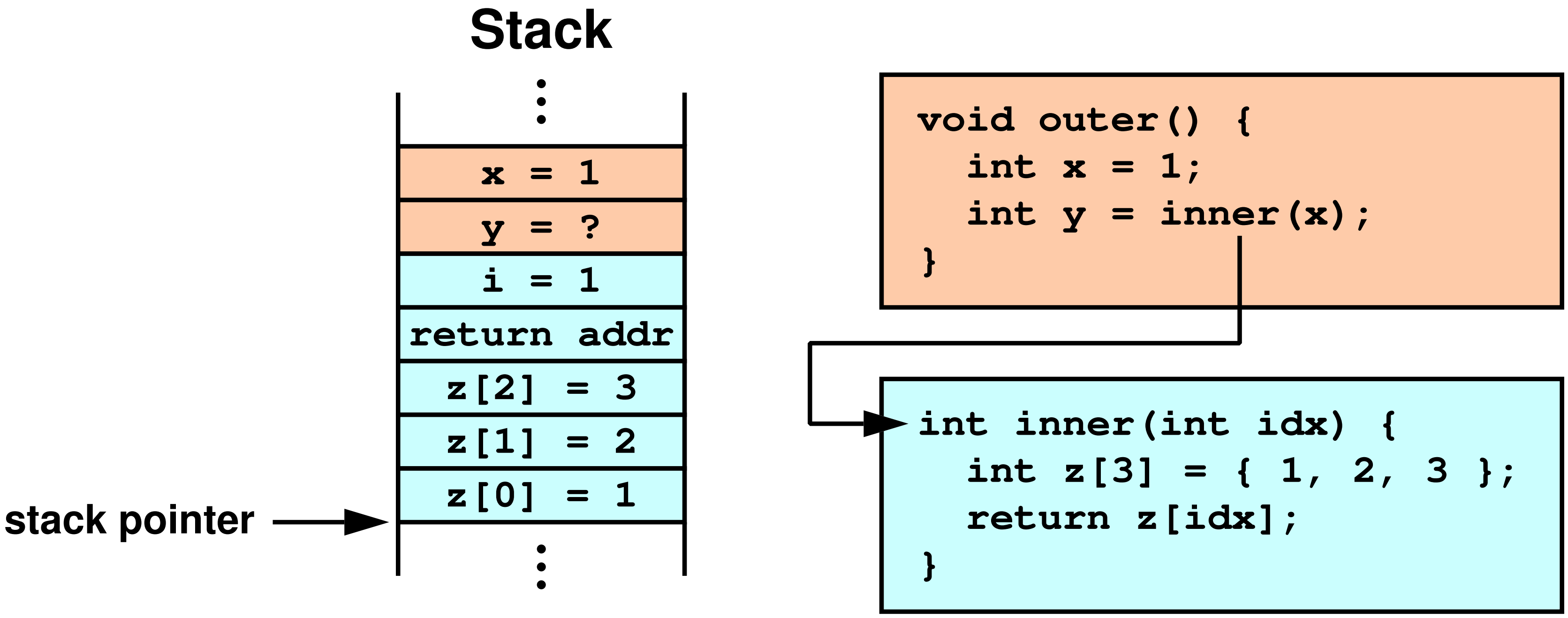
  2) process control block (PCB) which contains all the information the kernel needs about the process, e.g.,
     - where the address space is stored in memory
     - where the executable image resides on disk
     - which user the process belongs to
     - what privileges the process has
     - etc.

Process Address Space

- All (virtual) memory a process can address



Example: Function Invocation



Stack

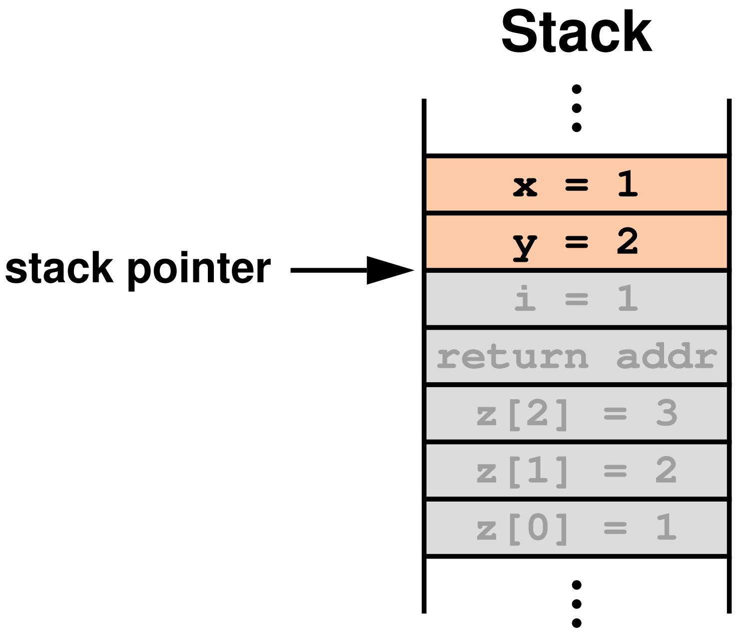
```
x = 1
y = ?
i = 1
return addr
z[2] = 3
z[1] = 2
z[0] = 1
```

stack pointer

```
void outer() {
  int x = 1;
  int y = inner(x);
}
```

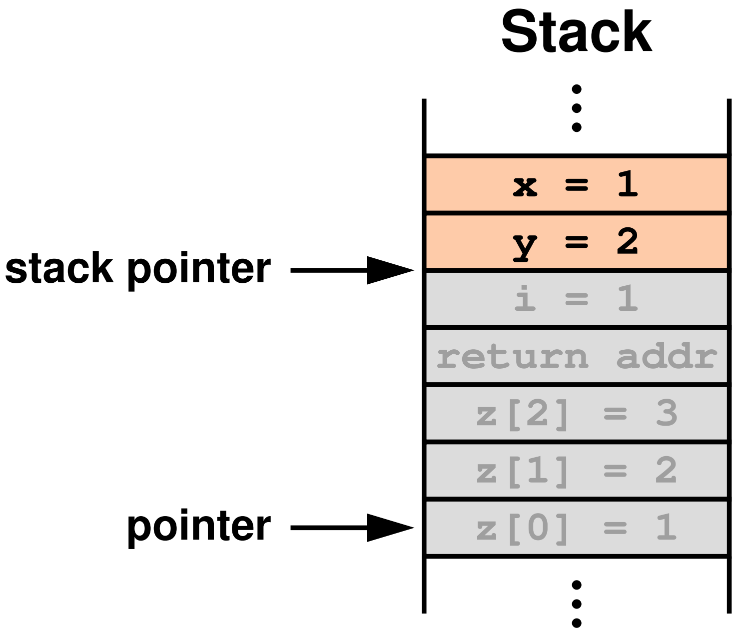
```
int inner(int idx) {
  int z[3] = { 1, 2, 3 };
  return z[idx];
}
```

Example: Function Invocation



```
void outer() {
  int x = 1;
  int y = inner(x);
}
```

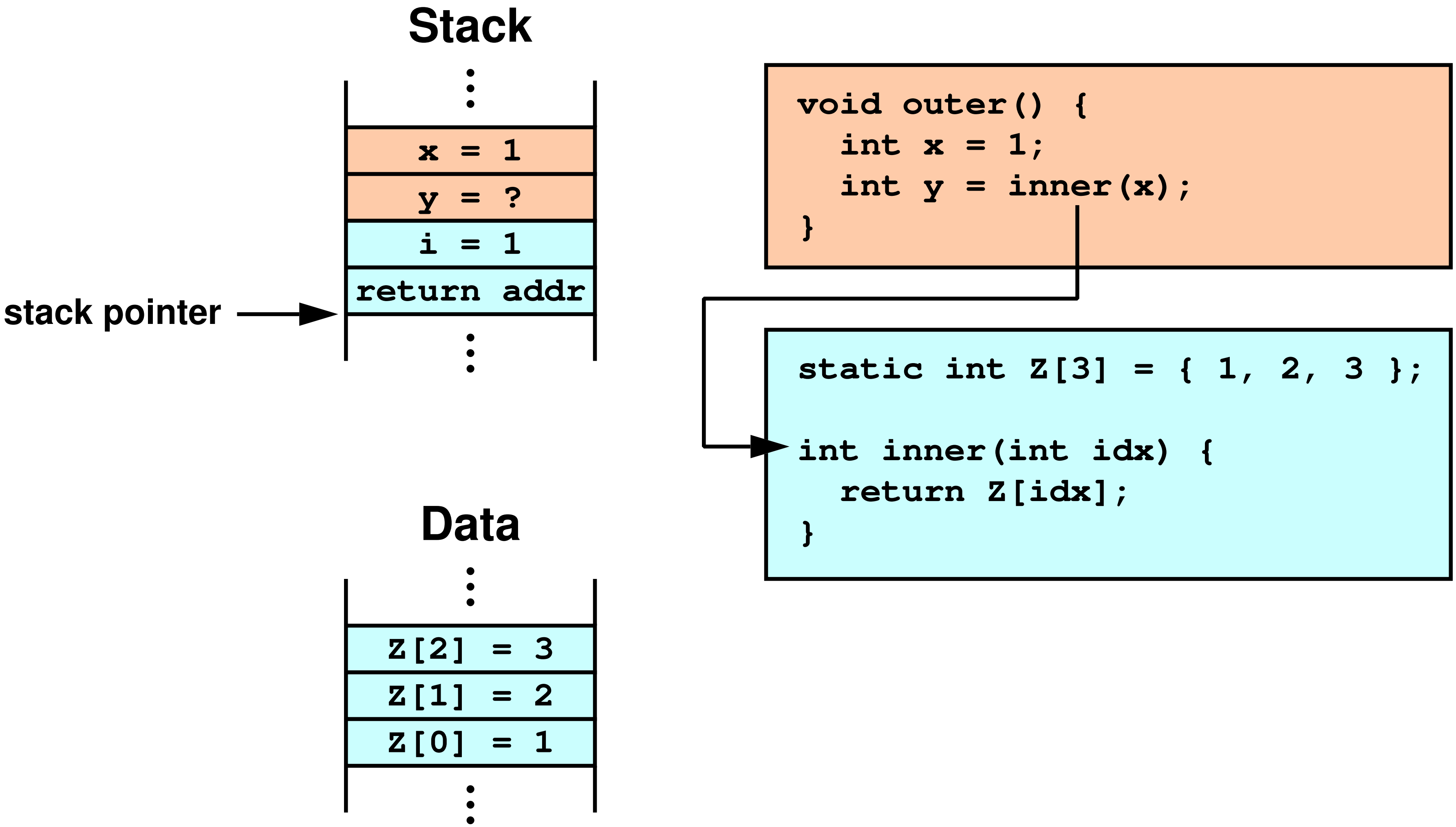
Example: Function Invocation



```
void outer() {
  int x = 1;
  int y = inner(x);
}
```

- If you have a pointer pointing to the old/osbolete z somehow, this pointer is now pointing to "garbage" (although it doesn't look like garbage at this moment)
  - when will it turn into garbage?
    - when you make another function call, you would build a stack frame and may wipe out the values in old z

Example: Static Variables



Stack

x = 1
y = ?
i = 1
return addr

stack pointer

Data

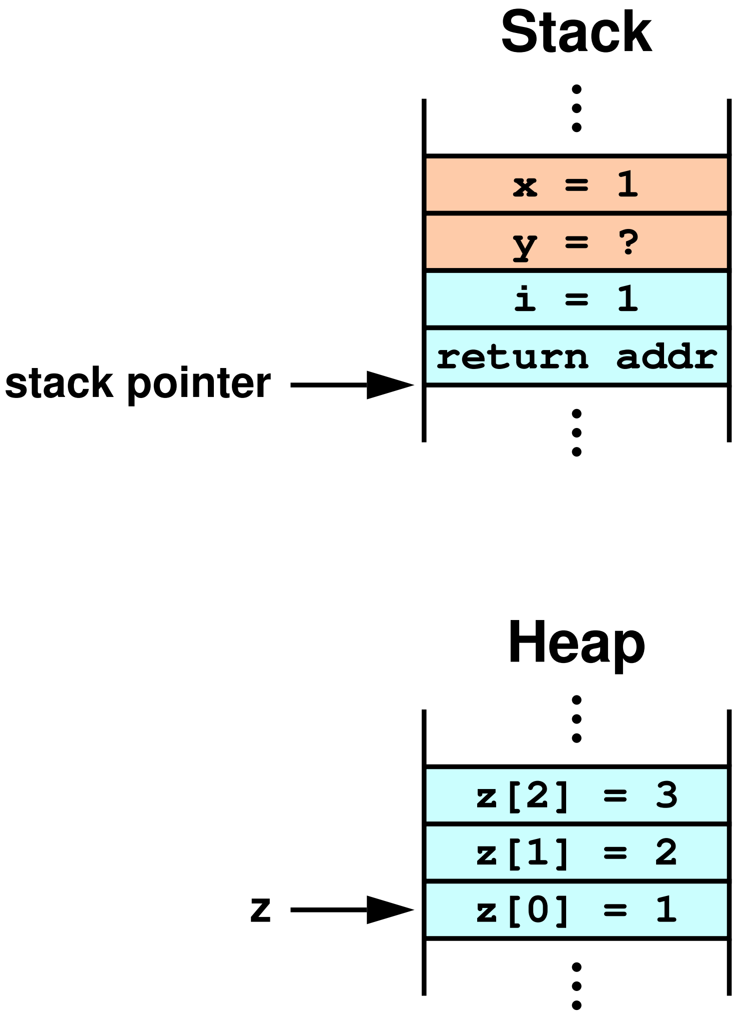
Z[2] = 3
Z[1] = 2
Z[0] = 1

```
void outer() {
  int x = 1;
  int y = inner(x);
}
```

```
static int Z[3] = { 1, 2, 3 };

int inner(int idx) {
  return Z[idx];
}
```

Example: Dynamic/Heap Memory



Stack

```
x = 1
y = ?
i = 1
return addr
```

stack pointer

Heap

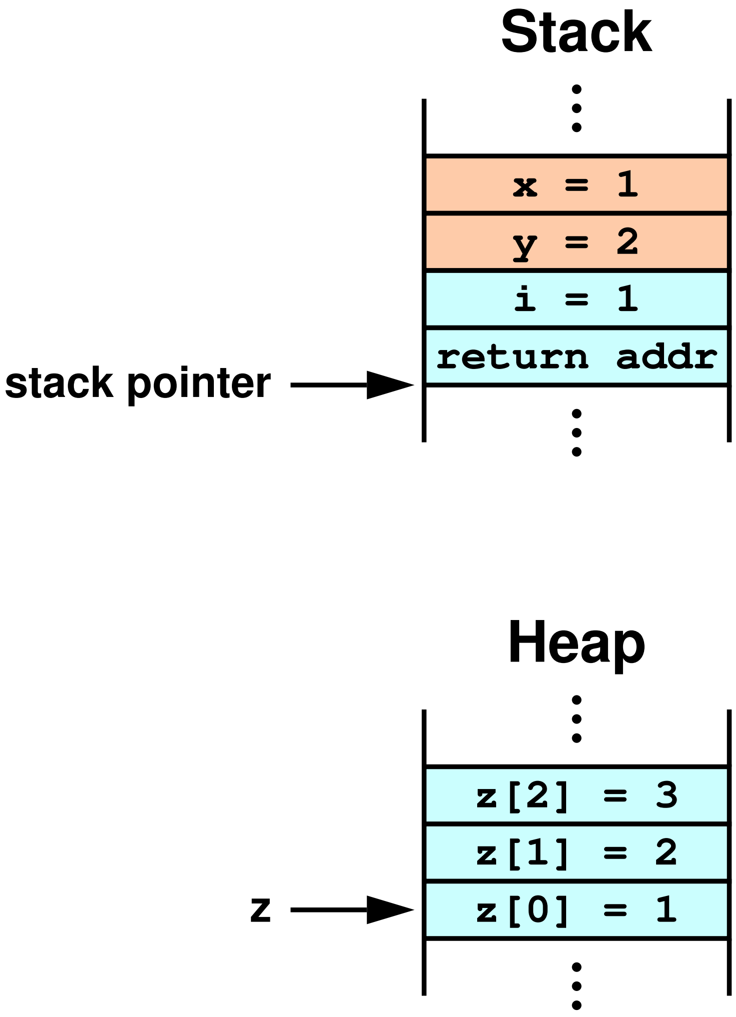
```
z[2] = 3
z[1] = 2
z[0] = 1
```

z

```
void outer() {
  int x = 1;
  int y = inner(x);
}
```

```
int inner(int idx) {
  z = malloc(3*sizeof(int));
  z[0] = 1;
  z[1] = 2;
  z[3] = 3;
  return z[idx];
}
```

Example: Dynamic/Heap Memory

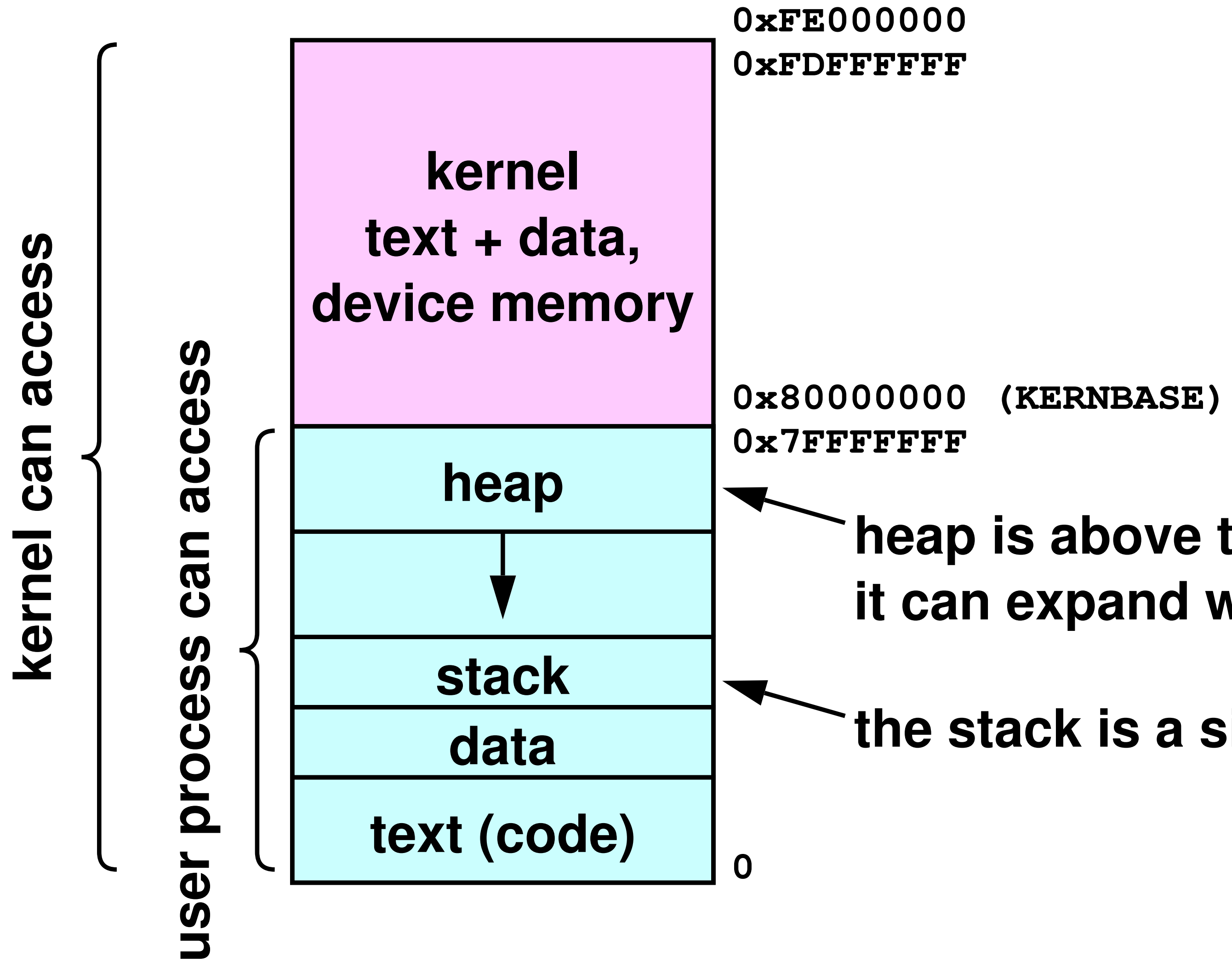


```
void outer() {
  int x = 1;
  int y = inner(x);
}
```

- Memory leak
  - no way to figure out where the dynamically allocated z was
  - when you use up your heap (maybe because too many memory leaks), `malloc()` will fail and return `NULL`)

Process Address Space in xv6

- A process's user memory starts at virtual address 0, can grow up to `KERNBASE`, allowing a process to address up to 2 GB of memory



heap is above the stack so that it can expand with `sbrk()`

the stack is a single page (4 KB)

Process Control Block (PCB)

- OS maintains information about every process in a data structure called a *Process Control Block (PCB)*
- PCB contains:
  - unique process identifier (PID)
  - process state (running, ready, etc.)
  - CPU state (program counter, registers, flags)
  - memory management information (page table, segment table, base & bound registers)
  - CPU scheduling & accounting information
  - parent process
  - child processes
  - ...

PCB Example xv6: "`proc.h`"

```
// Per-process state
struct proc {
  uint sz;                      // Size of process memory (bytes)
  pde_t* pgdir;                 // Page table
  char *kstack;                 // Bottom of kernel stack for this process
  enum procstate state;         // Process state
  int pid;                      // Process ID
  struct proc *parent;          // Parent process
  struct trapframe *tf;         // Trap frame for current syscall
  struct context *context;      // swtch() here to run process
  void *chan;                   // If non-zero, sleeping on chan
  int killed;                   // If non-zero, have been killed
  struct file *ofile[NOFILE];   // Open files
  struct inode *cwd;            // Current directory
  char name[16];                // Process name (debugging)
};
```

PCBs of Multiple Processes

- OS has a Process Table to manage all the PCBs
  - for each process, there is one entry in the table

Process Table

PID	PCB
1	●
2	●
...	...
N	●