OS History and OS Structures

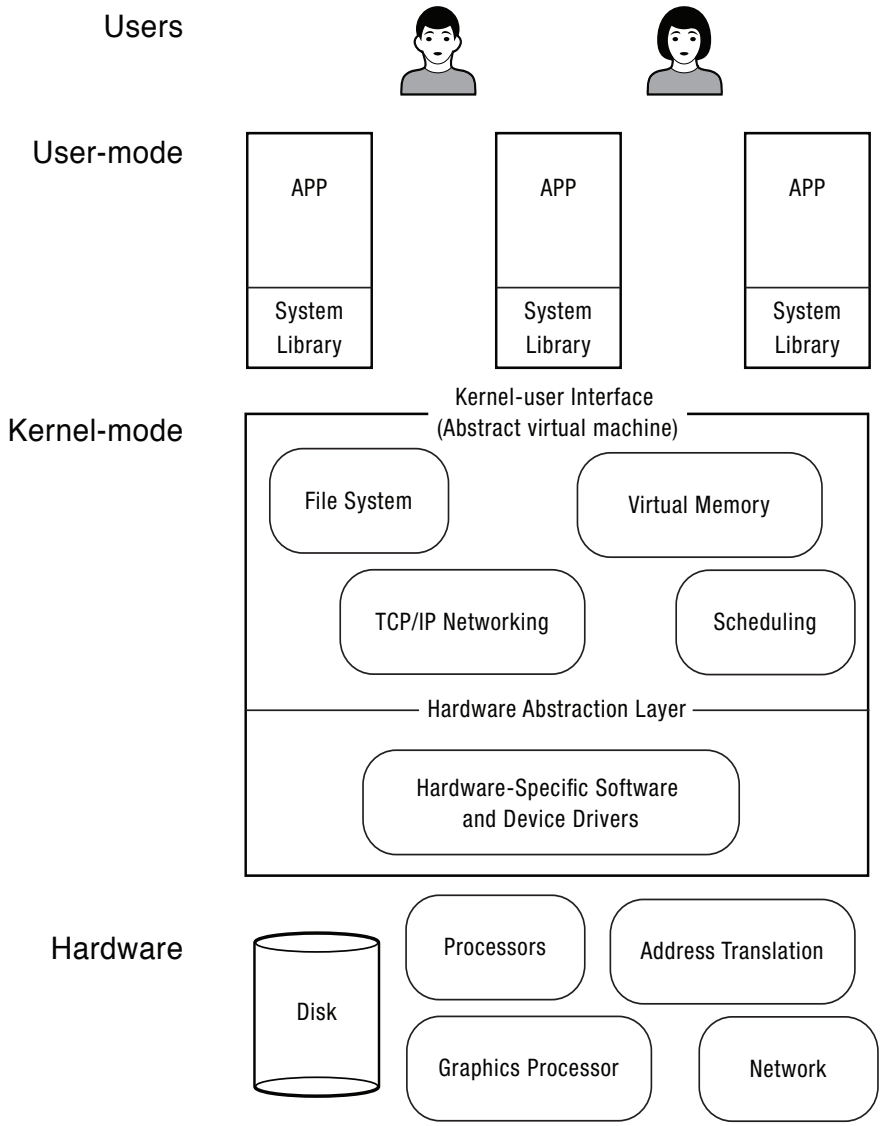
Karthik Dantu

CSE 421/521: Operating Systems

Slides adopted from CS162 class at Berkeley, CSE 451 at U-Washington and CSE 421 by Prof Kosar at UB

What is an OS?

- Software to manage a computer's resources for its users and applications



Operating System Roles

- Referee:
  - Resource allocation among users, applications
  - Isolation of different users, applications from each other
  - Communication between users, applications
- Illusionist
  - Each application appears to have the entire machine to itself
  - Infinite number of processors, (near) infinite amount of memory, reliable storage, reliable network transport
- Glue
  - Libraries, user interface widgets, …

Example: File Systems

- Referee
  - Prevent users from accessing each other's files without permission
  - Even after a file is deleting and its space re-used
- Illusionist
  - Files can grow (nearly) arbitrarily large
  - Files persist even when the machine crashes in the middle of a save
- Glue
  - Named directories, printf, …

Question

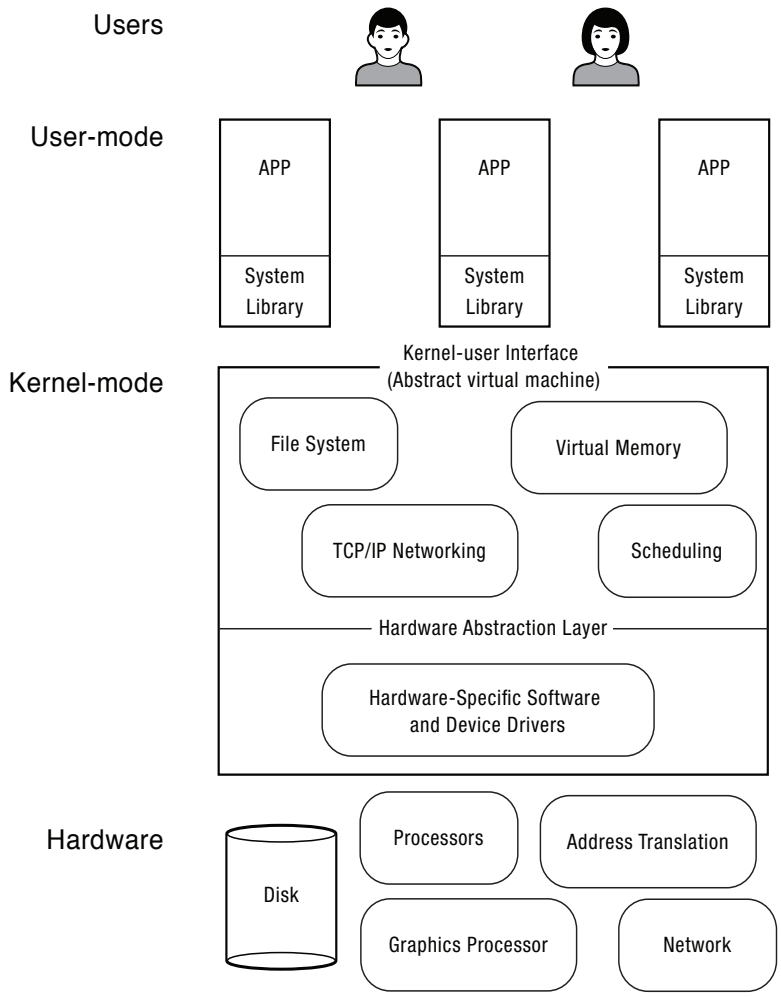
- What (hardware, software) do you need to be able to run an untrustworthy application?

OS Challenges

- Reliability
  - Does the system do what it was designed to do?
- Availability
  - What portion of the time is the system working?
  - Mean Time To Failure (MTTF), Mean Time to Repair
- Security
  - Can the system be compromised by an attacker?
- Privacy
  - Data is accessible only to authorized users

OS Challenges

- Portability
  - For programs:
    - Application programming interface (API)
    - Abstract virtual machine (AVM)
  - For the operating system
    - Hardware abstraction layer



OS Challenges

- Performance
  - Latency/response time
    - How long does an operation take to complete?
  - Throughput
    - How many operations can be done per unit of time?
  - Overhead
    - How much extra work is done by the OS?
  - Fairness
    - How equal is the performance received by different users?
  - Predictability
    - How consistent is the performance over time?

Computer Performance Over Time

	1981	1997	2014	Factor (2014/1981)
Uniprocessor speed (MIPS)	1	200	2500	2.5K
CPUs per computer	1	1	10+	10+
Processor MIPS/$	$100K	$25	$0.20	500K
DRAM Capacity (MiB)/$	0.002	2	1K	500K
Disk Capacity (GiB)/$	0.003	7	25K	10M
Home Internet	300 bps	256 Kbps	20 Mbps	100K
Machine room network	10 Mbps (shared)	100 Mbps (switched)	10 Gbps (switched)	1000
Ratio of users to computers	100:1	1:1	1:several	100+

Early Operating Systems: Serial Operations

- One application at a time
  - Had complete control of hardware
  - OS was runtime library
  - Users would stand in line to use the computer
- Batch systems
  - Keep CPU busy by having a queue of jobs
  - OS would load next job while current one runs
  - Users would submit jobs, and wait, and wait, and

Time-Sharing Operating Systems: Client-Server Age

- Multiple users on computer at same time
  - Multiprogramming: run multiple programs at same time
  - Interactive performance: try to complete everyone's tasks quickly
  - As computers became cheaper, more important to optimize for user time, not computer time

Today's Operating Systems: Computers Cheap

- Smartphones
- Embedded systems
- Laptops
- Tablets
- Virtual machines
- Data center servers

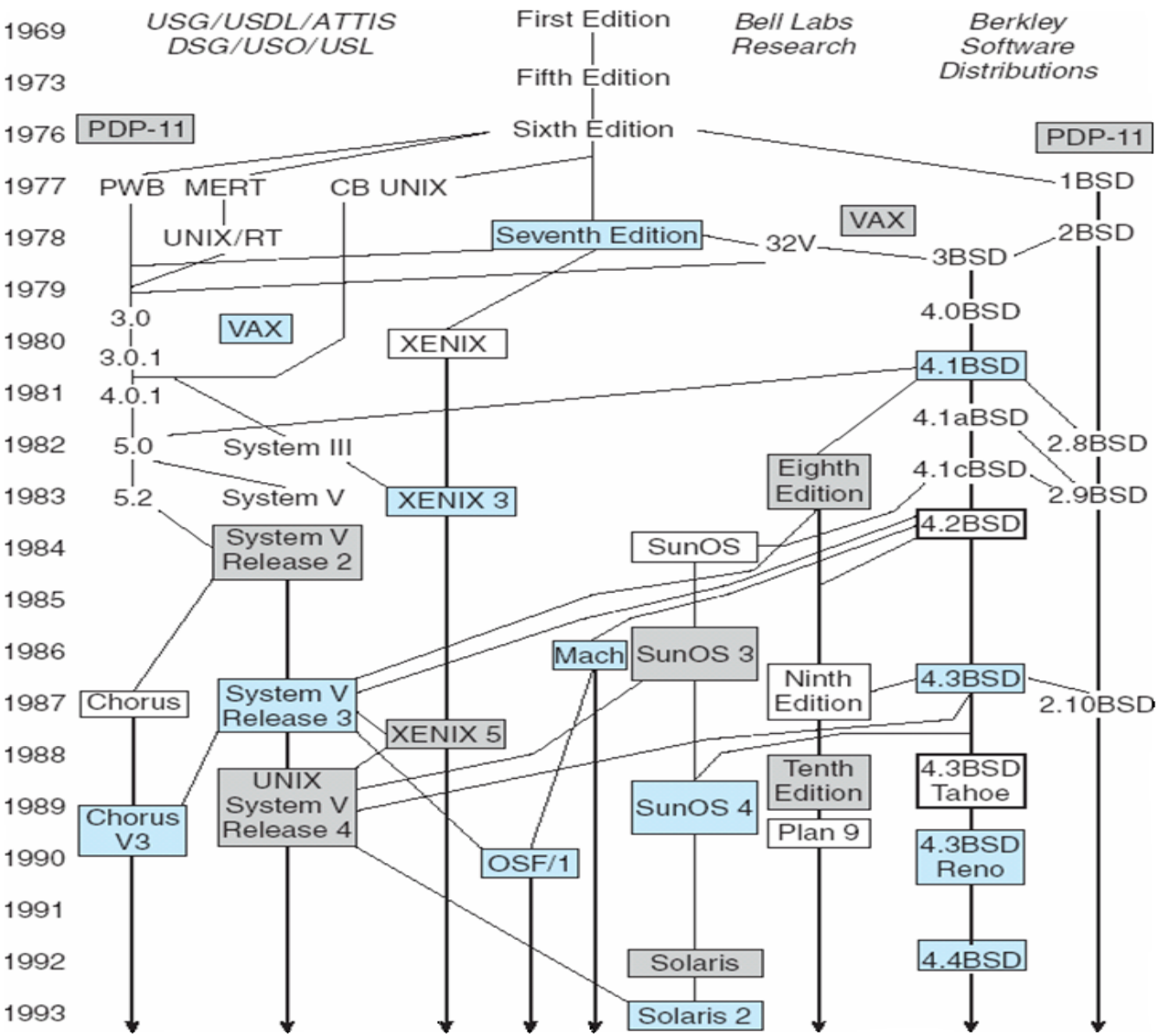
Tomorrow's Operating Systems

- Giant-scale data centers
- Increasing numbers of processors per computer
- Increasing numbers of computers per user
- Very large scale storage
- Mark Weiser: Ubiquitous and Pervasive Computing?

Unix History

- First developed in 1969 by Ken Thompson and Dennis Ritchie of the Research Group at Bell Laboratories; incorporated features of other operating systems, especially MULTICS
- The third version was written in C, which was developed at Bell Labs specifically to support UNIX
- The most influential of the non-Bell Labs and non-AT&T UNIX development groups — University of California at Berkeley (Berkeley Software Distributions - BSD)
- 4BSD UNIX resulted from DARPA funding to develop a standard UNIX system for government use
- Developed for the VAX, 4.3BSD is one of the most influential versions, and has been ported to many other platforms
- Several standardization projects seek to consolidate the variant flavors of UNIX leading to one programming interface to UNIX

Timeline of Unix versions

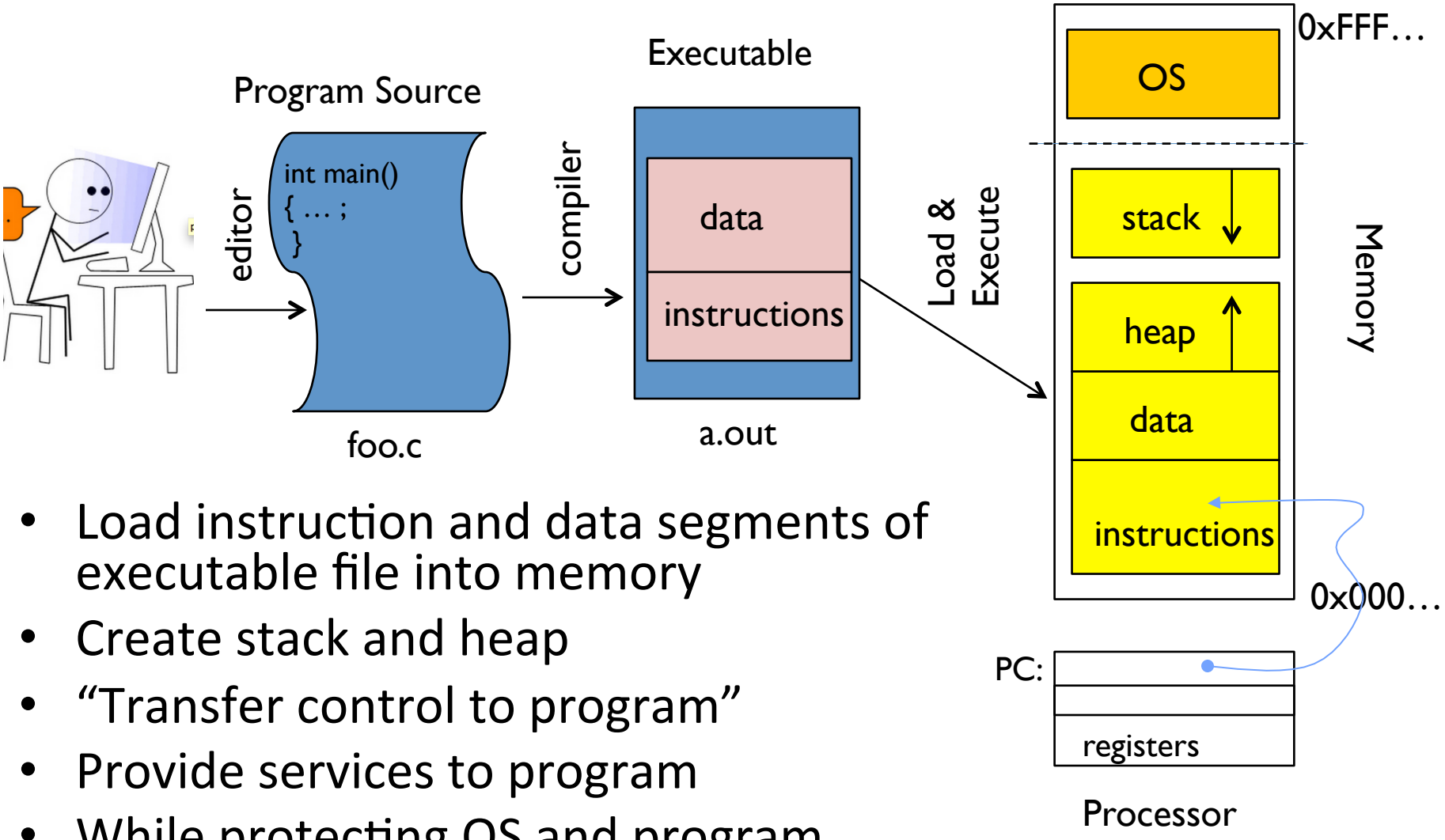


OPERATING SYSTEMS STRUCTURES

Today: Four Fundamental OS Concepts

- Thread
  - Single unique execution context: fully describes program state
  - Program Counter, Registers, Execution Flags, Stack
- Address space (with translation)
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine
- Process
  - An instance of an executing program is *a process consisting of an address space and one or more threads of control*
- Dual mode operation / Protection
  - Only the "system" has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses

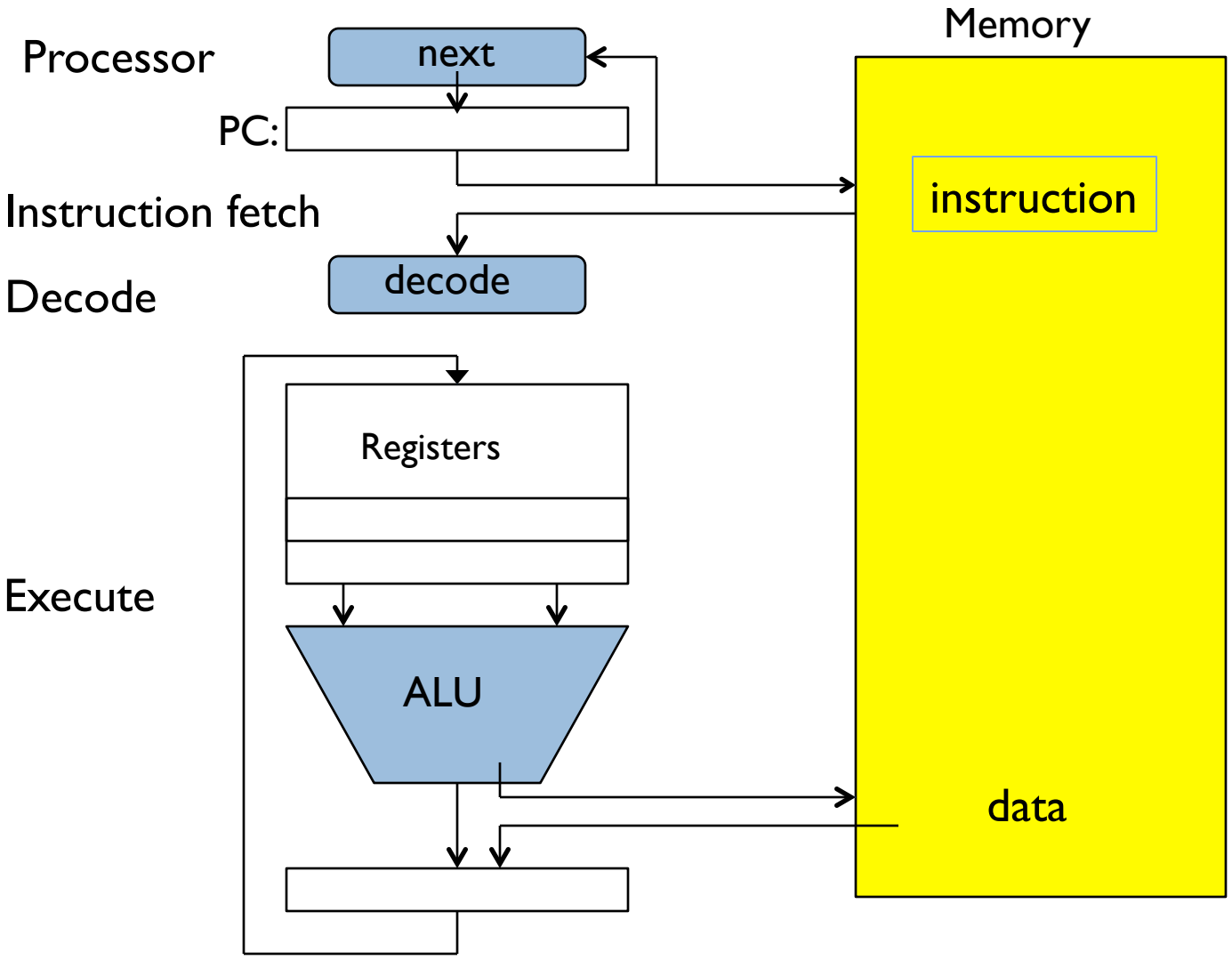
OS Bottom Line: Run Programs



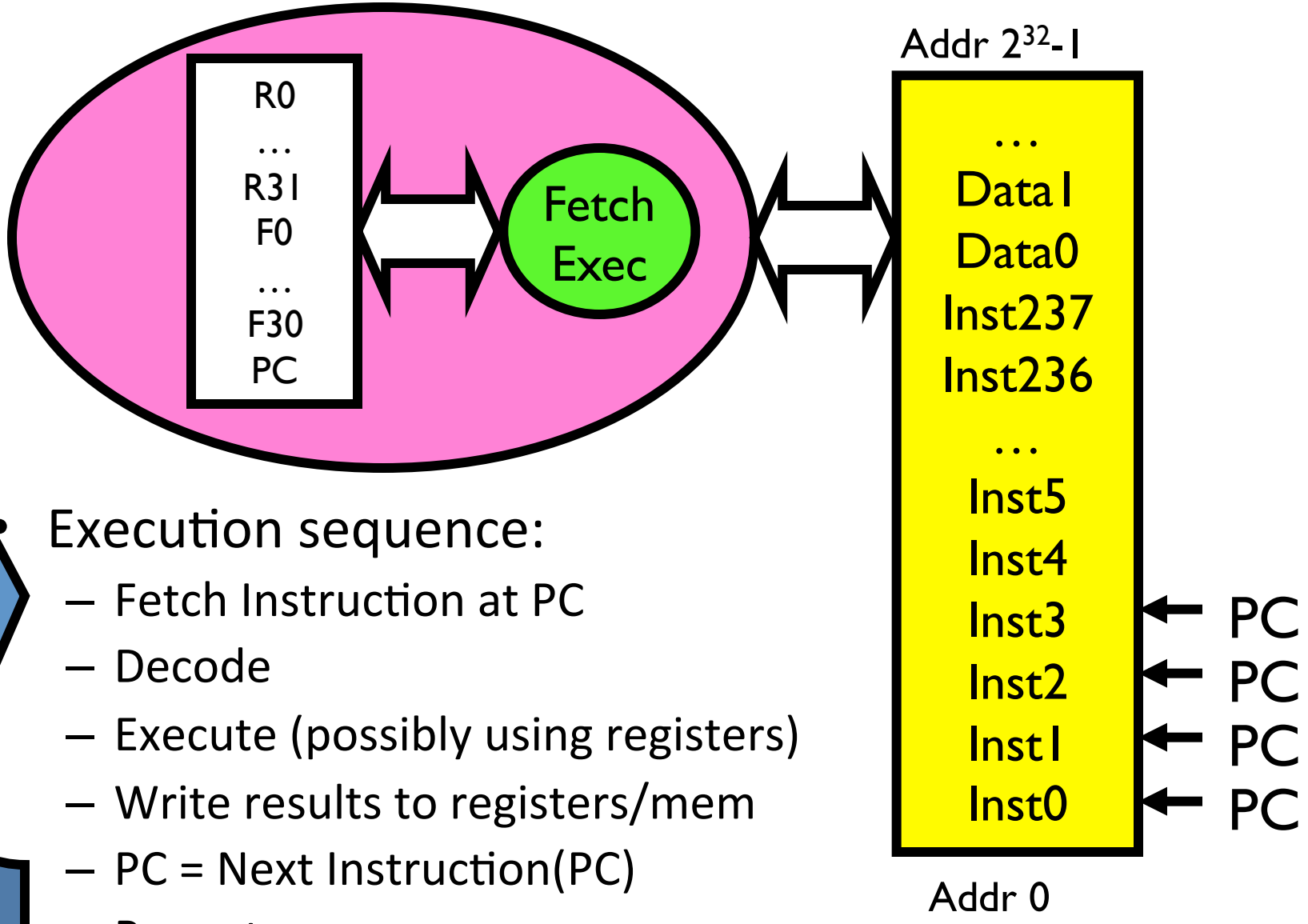
- Load instruction and data segments of executable file into memory
- Create stack and heap
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

Instruction Fetch/Decode/Execute Cycle

The instruction cycle



What happens during program execution?



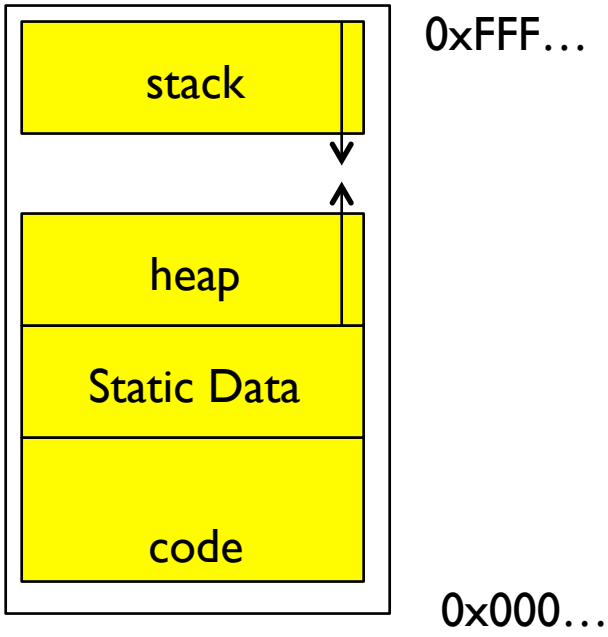
- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

First OS Concept: Thread of Control

- Certain registers hold the *context* of thread
  - Stack pointer holds the address of the top of stack
    - Other conventions: Frame pointer, Heap pointer, Data
  - May be defined by the instruction set architecture or by compiler conventions
- Thread: Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack
- A thread is executing on a processor when it is resident in the processor registers.
- PC register holds the address of executing instruction in the thread
- Registers hold the root state of the thread.
  - The rest is "in memory"

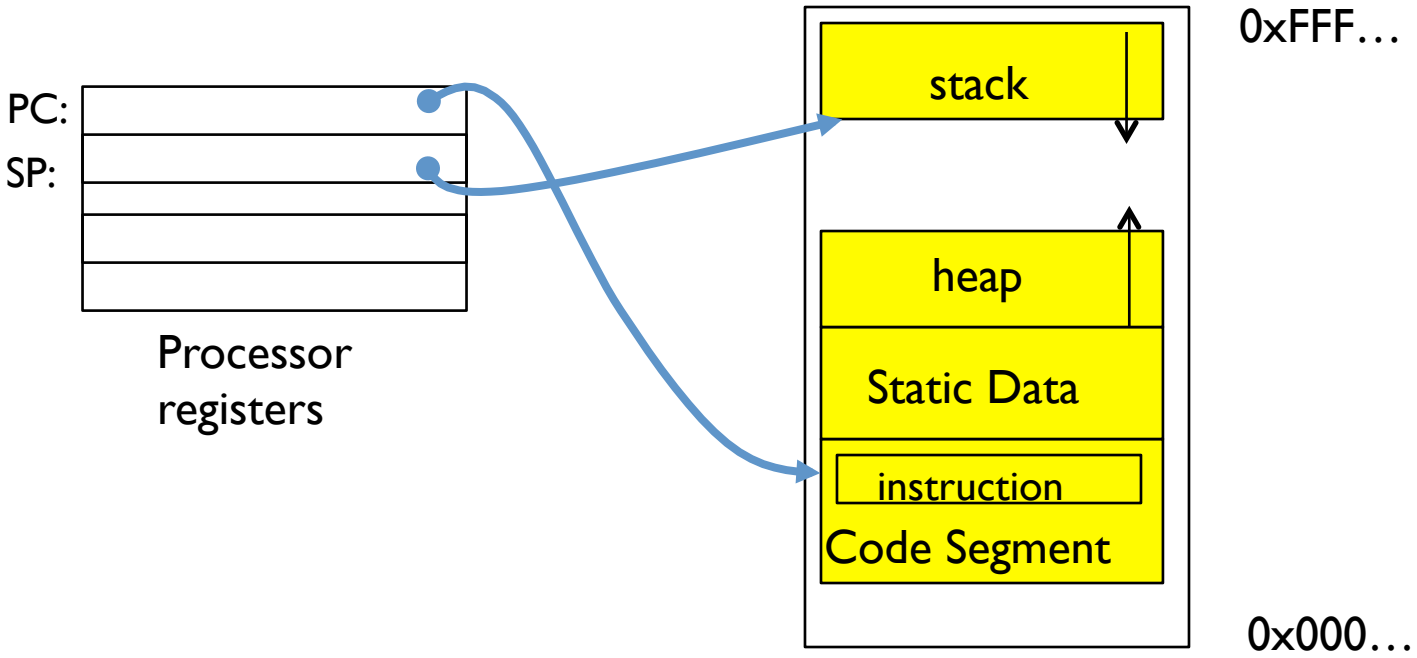
Second OS Concept: Program's Address Space

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are 2^{32} = 4 billion addresses



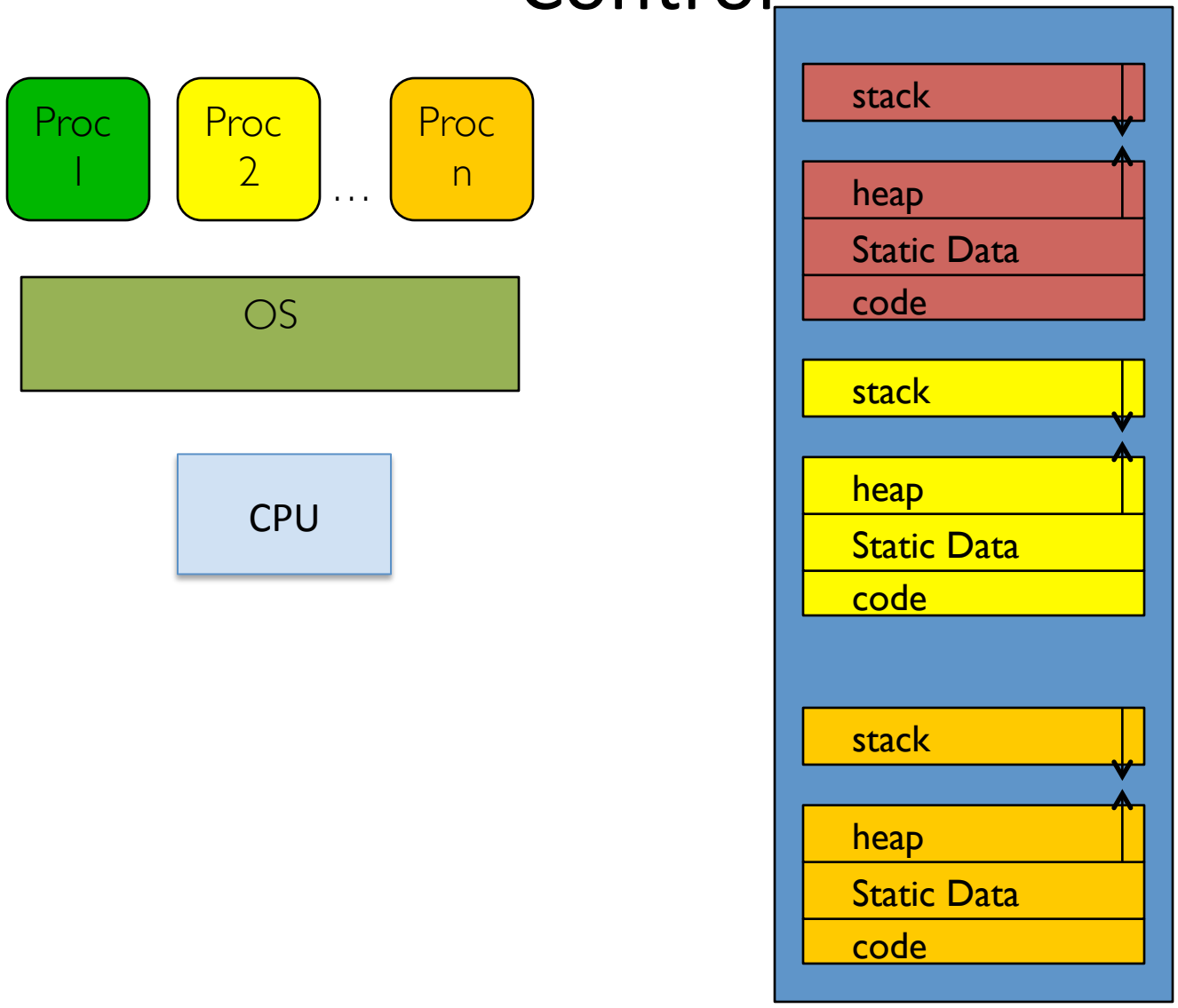
- What happens when you read or write to an address?
  - Perhaps nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - (Memory-mapped I/O)
  - Perhaps causes exception (fault)

Address Space: In a Picture

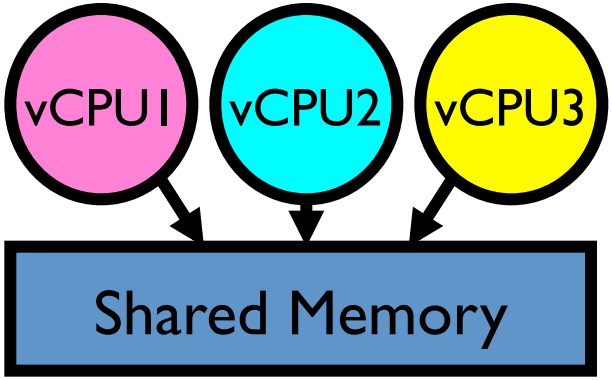


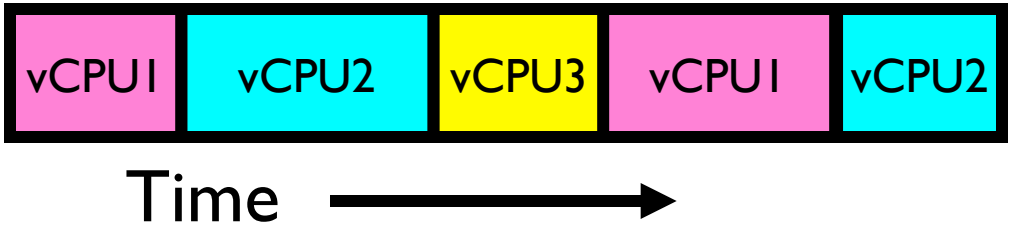
- What's in the code segment? Static data segment?
- What's in the Stack Segment?
  - How is it allocated? How big is it?
- What's in the Heap Segment?
  - How is it allocated? How big?

Multiprogramming - Multiple Threads of Control



How can we give the illusion of multiple processors?





- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one virtual CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

The Basic Problem of Concurrency

- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: processes think they have exclusive access to shared resources
- OS has to coordinate all activity
  - Multiple processes, I/O interrupts, …
  - How can it keep all these things straight?
- Basic Idea: Use Virtual Machine abstraction
  - Simple machine abstraction for processes
  - Multiplex these abstract machines

Properties of this simple multiprogramming technique

- All virtual CPUs share same non-CPU resources
  - I/O devices the same
  - Memory the same
- Consequence of sharing:
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
- This (unprotected) model is common in:
  - Embedded applications
  - Windows 3.1/Early Macintosh (switch only with yield)
  - Windows 95—ME (switch with both yield and timer)

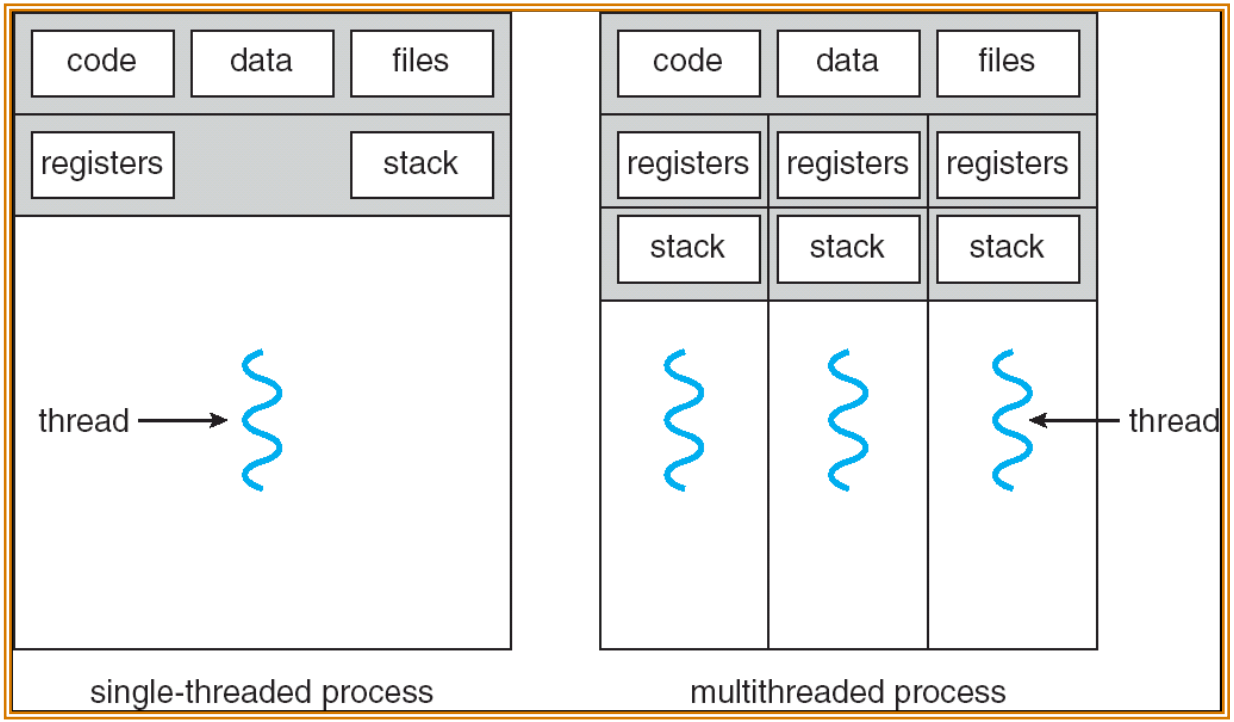
Protection

- Operating System must protect itself from user programs
  - Reliability: compromising the operating system generally causes it to crash
  - Security: limit the scope of what processes can do
  - Privacy: limit each process to the data it is permitted to access
  - Fairness: each should be limited to its appropriate share of system resources (CPU time, memory, I/O, etc)
- It must protect User programs from one another
- Primary Mechanism: limit the translation from program address space to physical memory space
  - Can only touch what is mapped into process *address space*
- Additional Mechanisms:
  - Privileged instructions, in/out instructions, special registers
  - syscall processing, subsystem implementation
    - (e.g., file access rights, etc)

Third OS Concept: Process

- Process: execution environment with Restricted Rights
  - Address Space with One or More Threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Encapsulate one or more threads sharing process resources
- Why processes?
  - Protected from each other!
  - OS Protected from them
  - Processes provides memory protection
  - Threads more efficient than processes (later)
- Fundamental tradeoff between protection and efficiency
  - Communication easier *within* a process
  - Communication harder *between* processes
- Application instance consists of one or more processes
  - E.g., Facebook app on your phone

Single and Multithreaded Processes

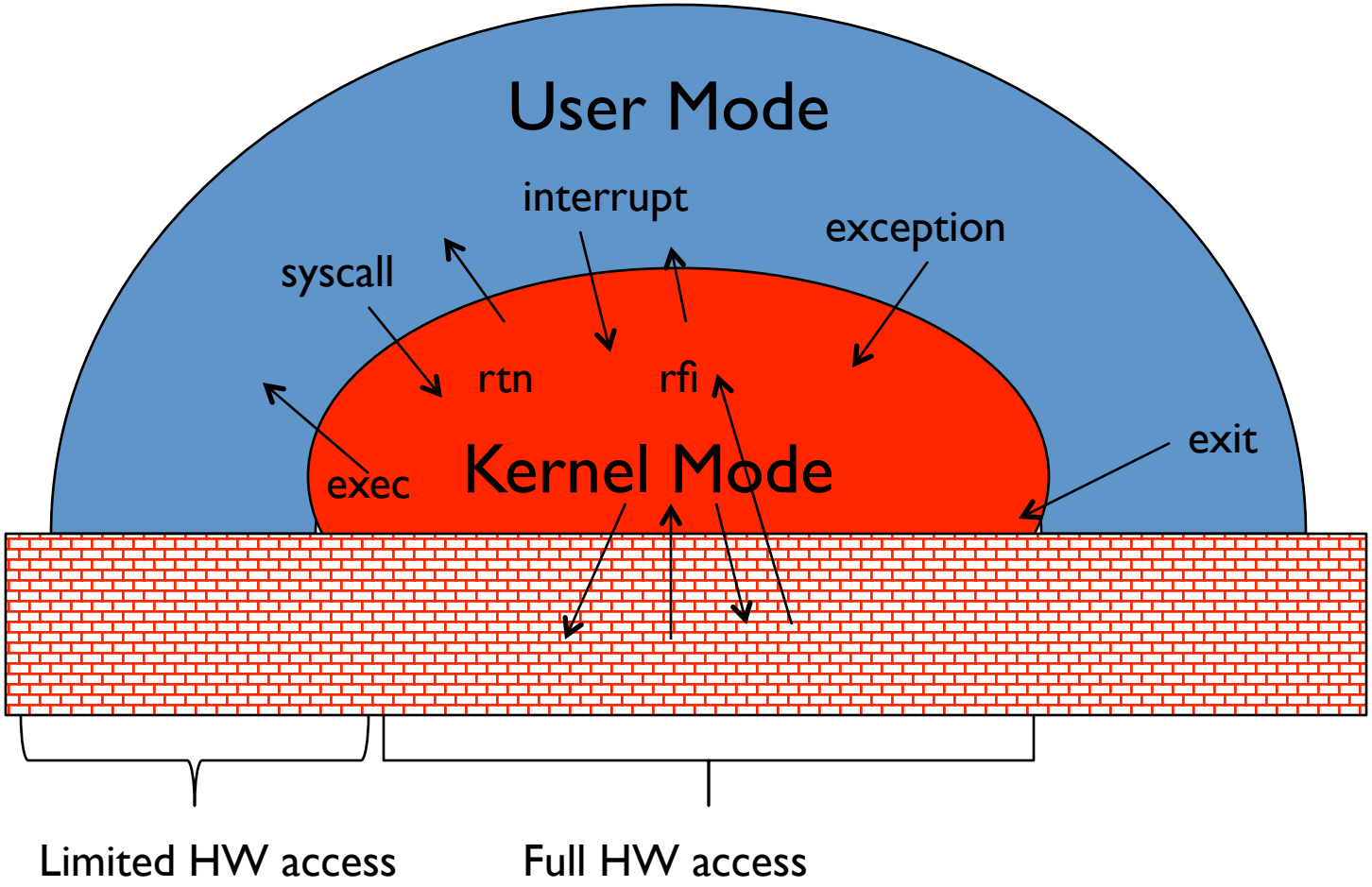


- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?
  - E.g., web server

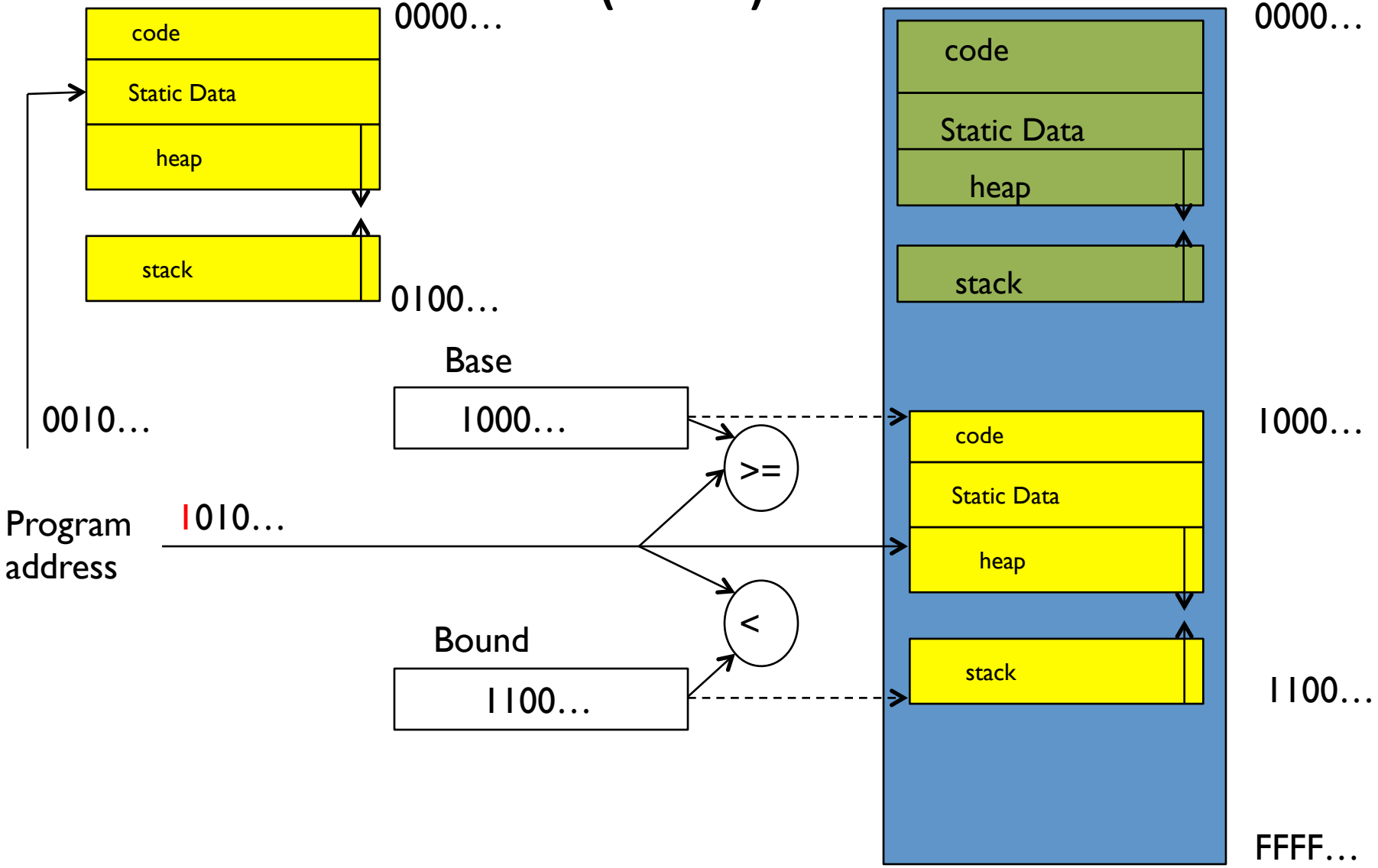
Fourth OS Concept: Dual Mode Operation

- Hardware provides at least two modes:
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode: Normal programs executed
- What is needed in the hardware to support "dual mode" operation?
  - A bit of state (user/system mode bit)
  - Certain operations / actions only permitted in system/kernel mode
    - In user mode they fail or trap
  - User → Kernel transition *sets* system mode AND saves the user PC
    - Operating system code carefully puts aside user state then performs the necessary operations
  - Kernel → User transition *clears* system mode AND restores appropriate user PC
    - return-from-interrupt

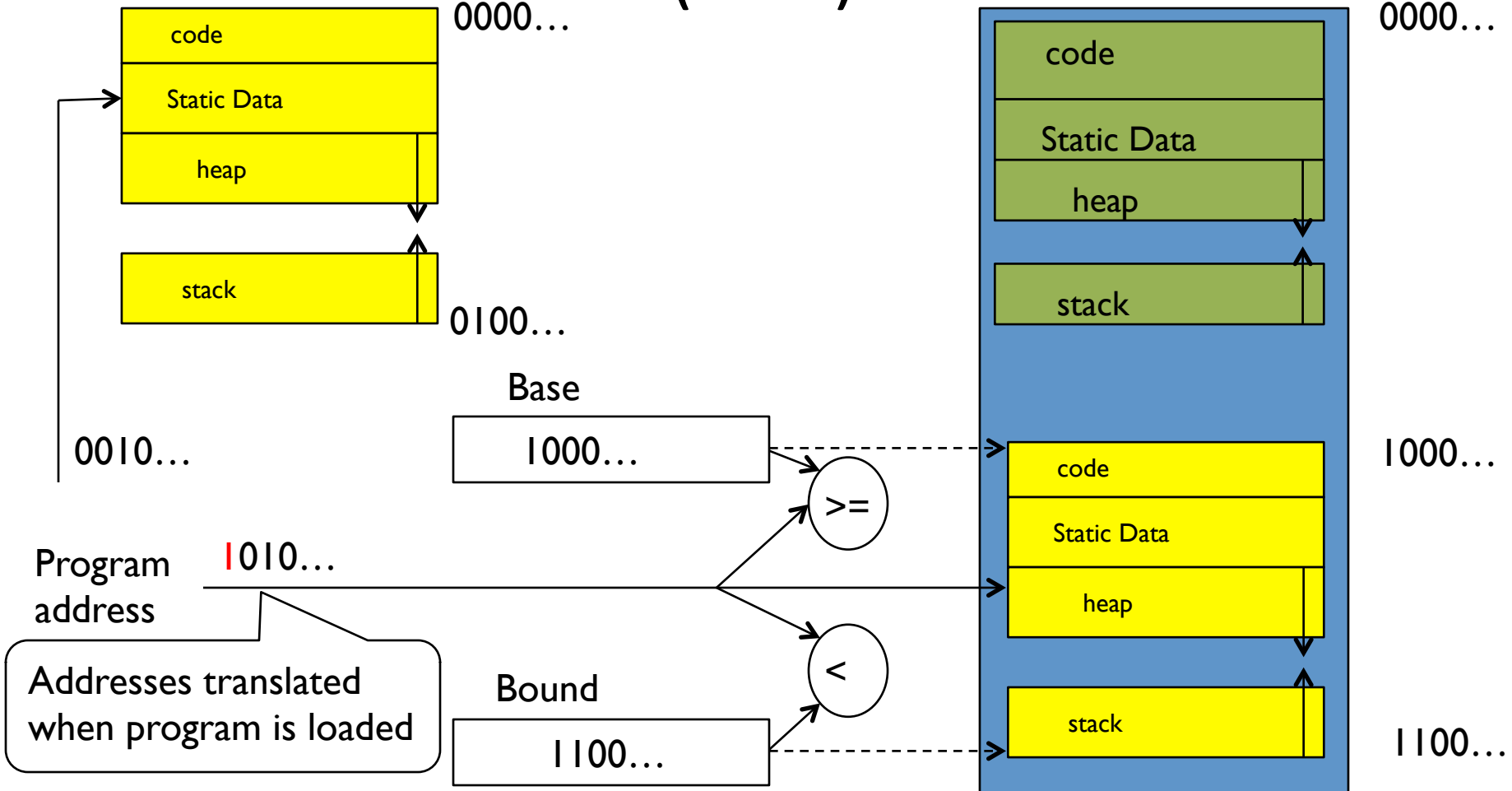
User/Kernel (Privileged) Mode



Simple Protection: Base and Bound (B&B)



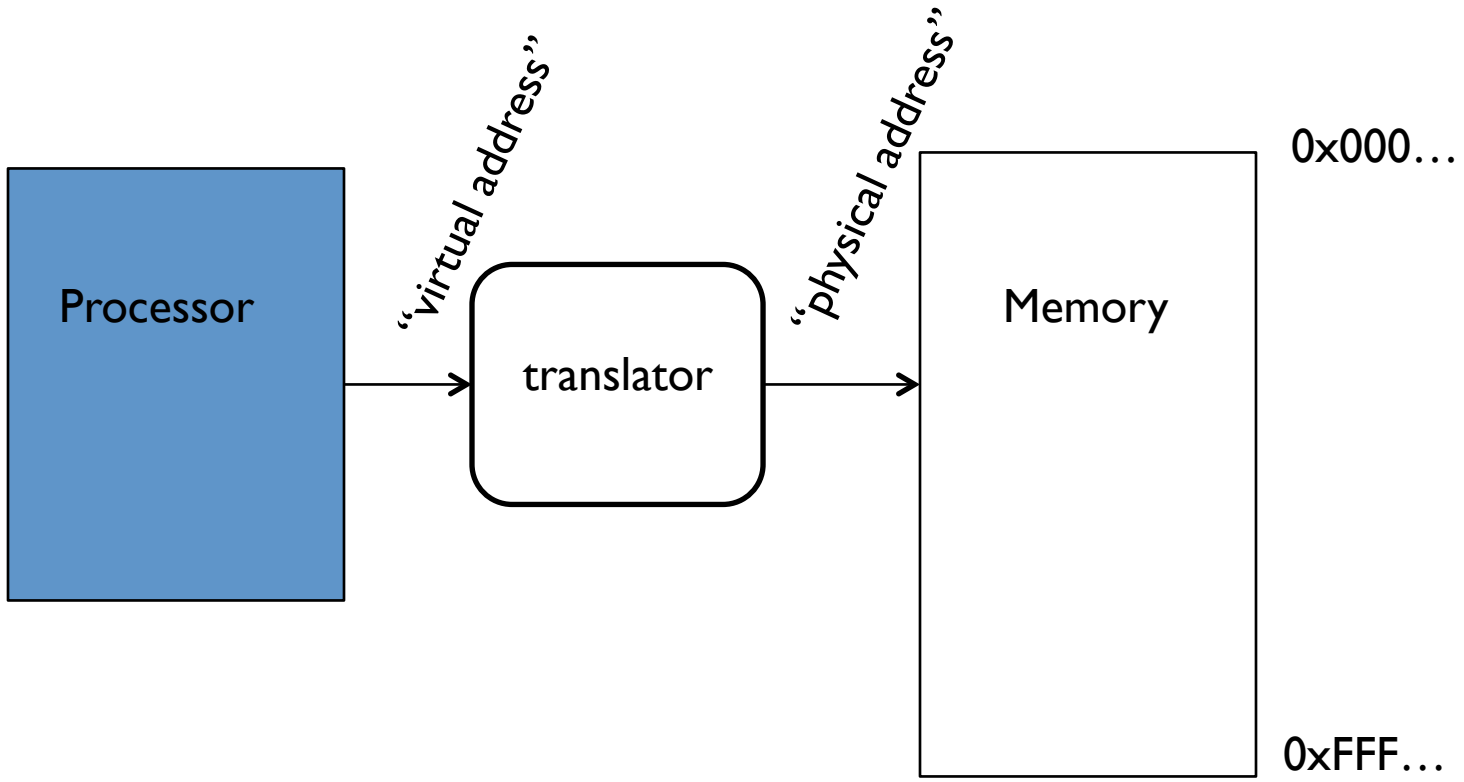
Simple Protection: Base and Bound (B&B)



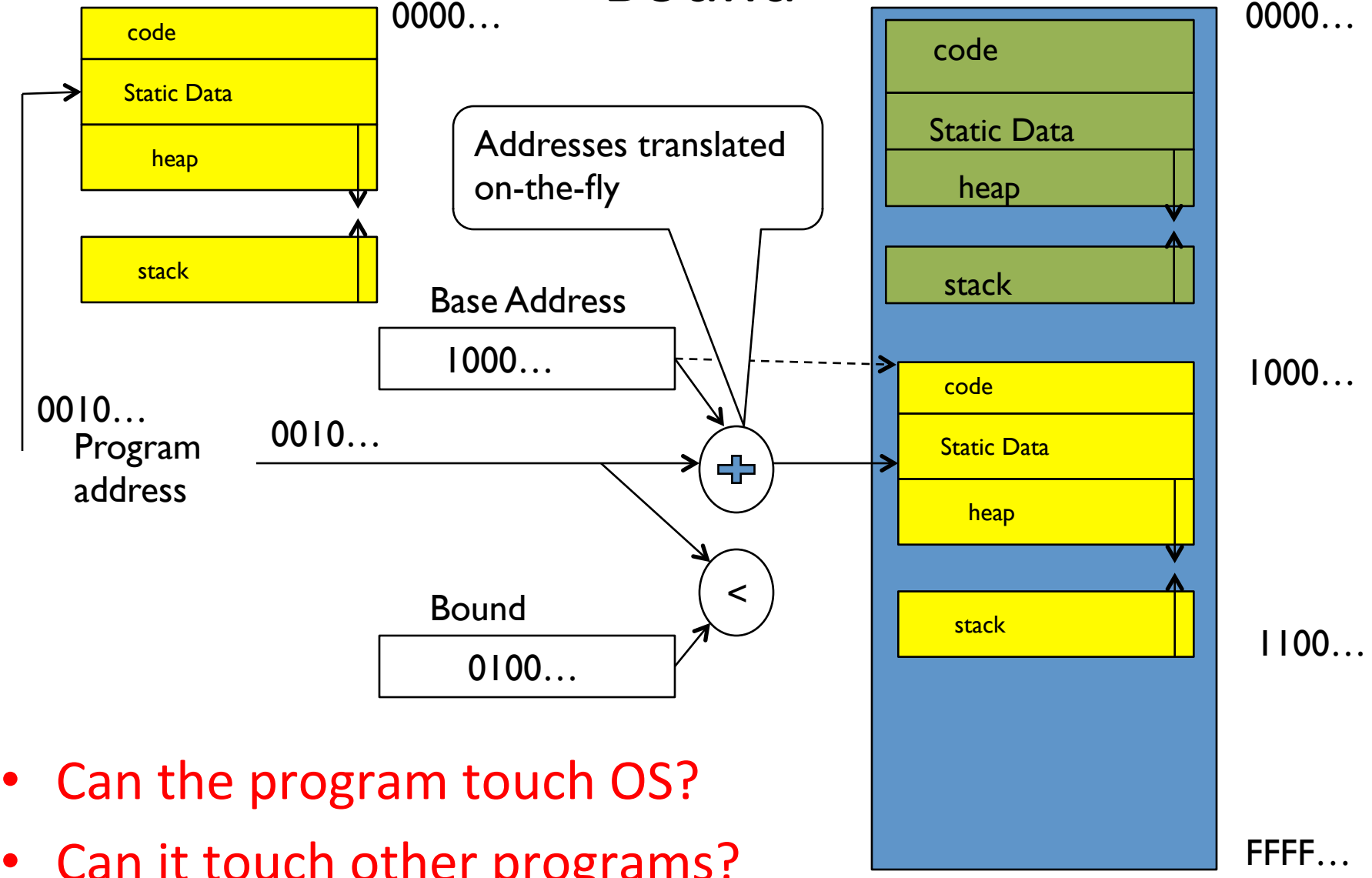
- Requires relocating loader
- Still protects OS and isolates program
- No addition on address path

Another idea: Address Space Translation

- Program operates in an address space that is distinct from the physical memory space of the machine

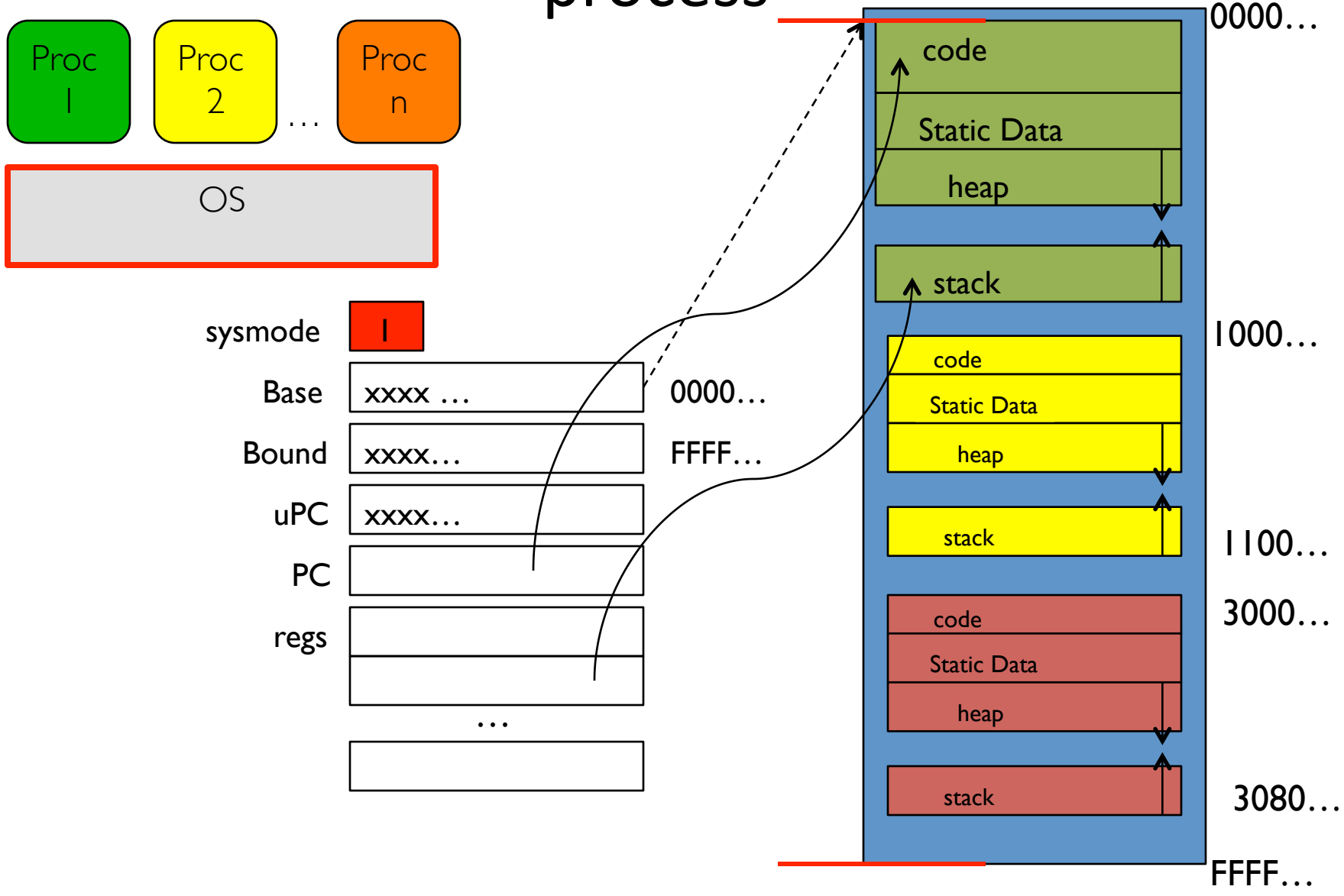


A simple address translation with Base and Bound

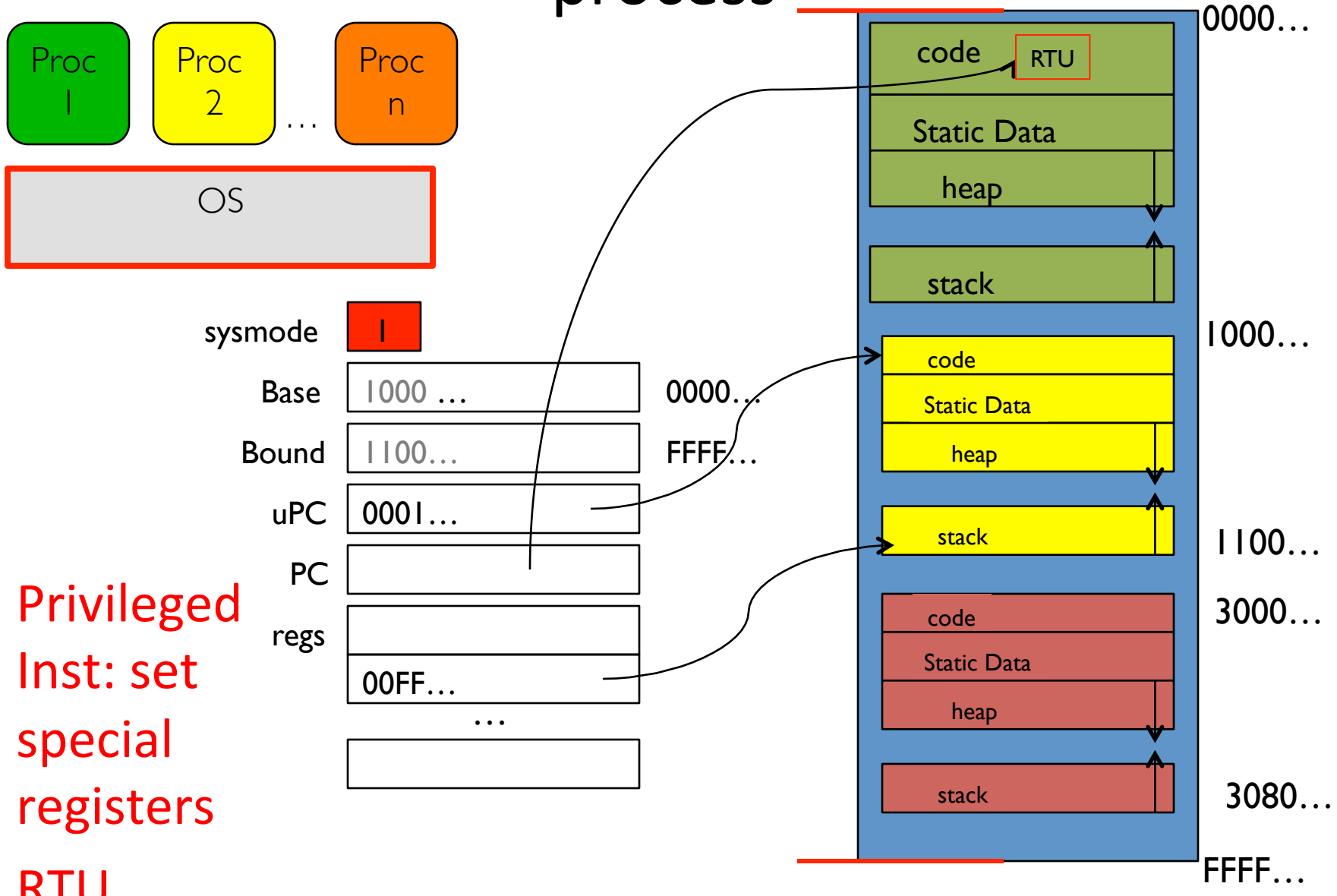


- Can the program touch OS?
- Can it touch other programs?

Tying it together: Simple B&B: OS loads process

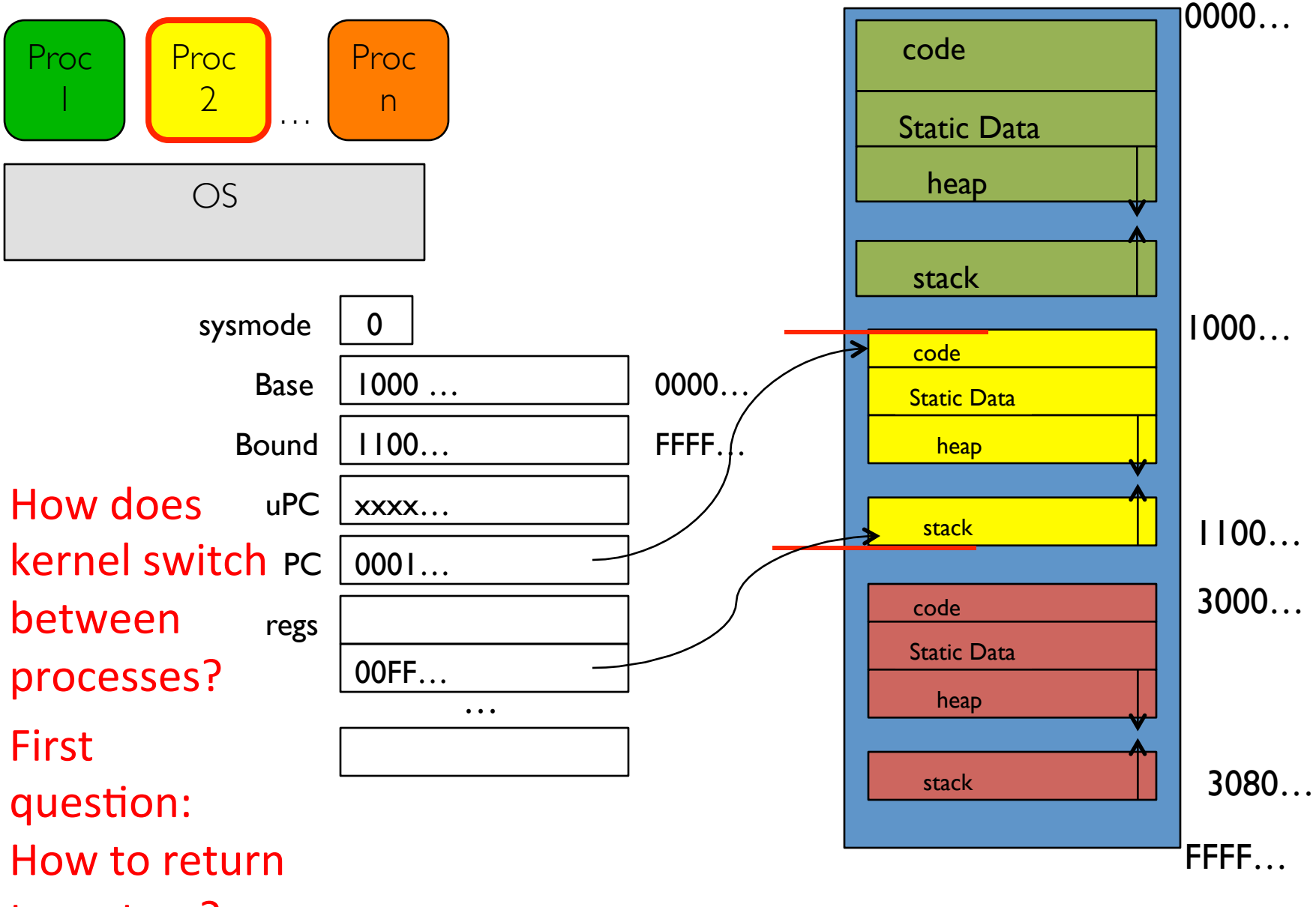


Simple B&B: OS gets ready to execute process



- Privileged Inst: set special registers
- RTU

Simple B&B: User Code Running



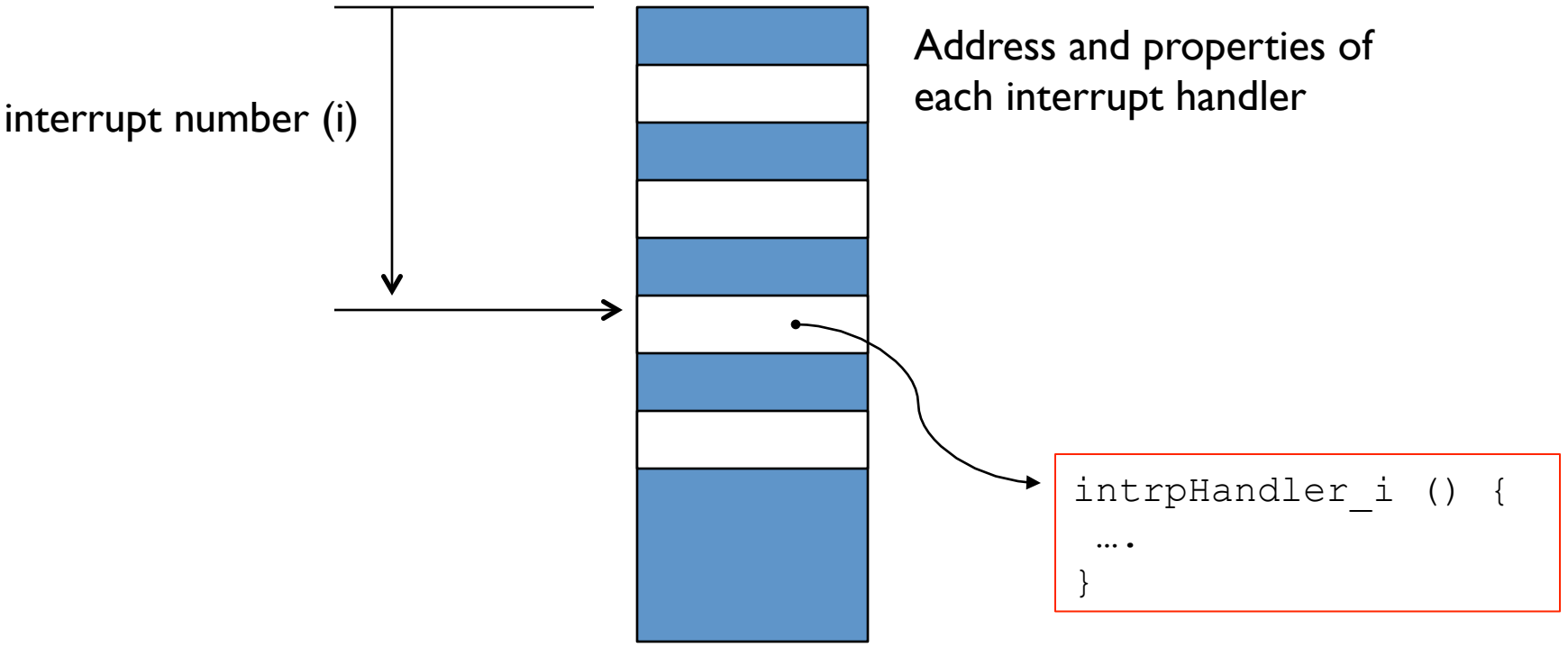
- How does kernel switch between processes?
- First question: How to return to system?

3 types of Mode Transfer

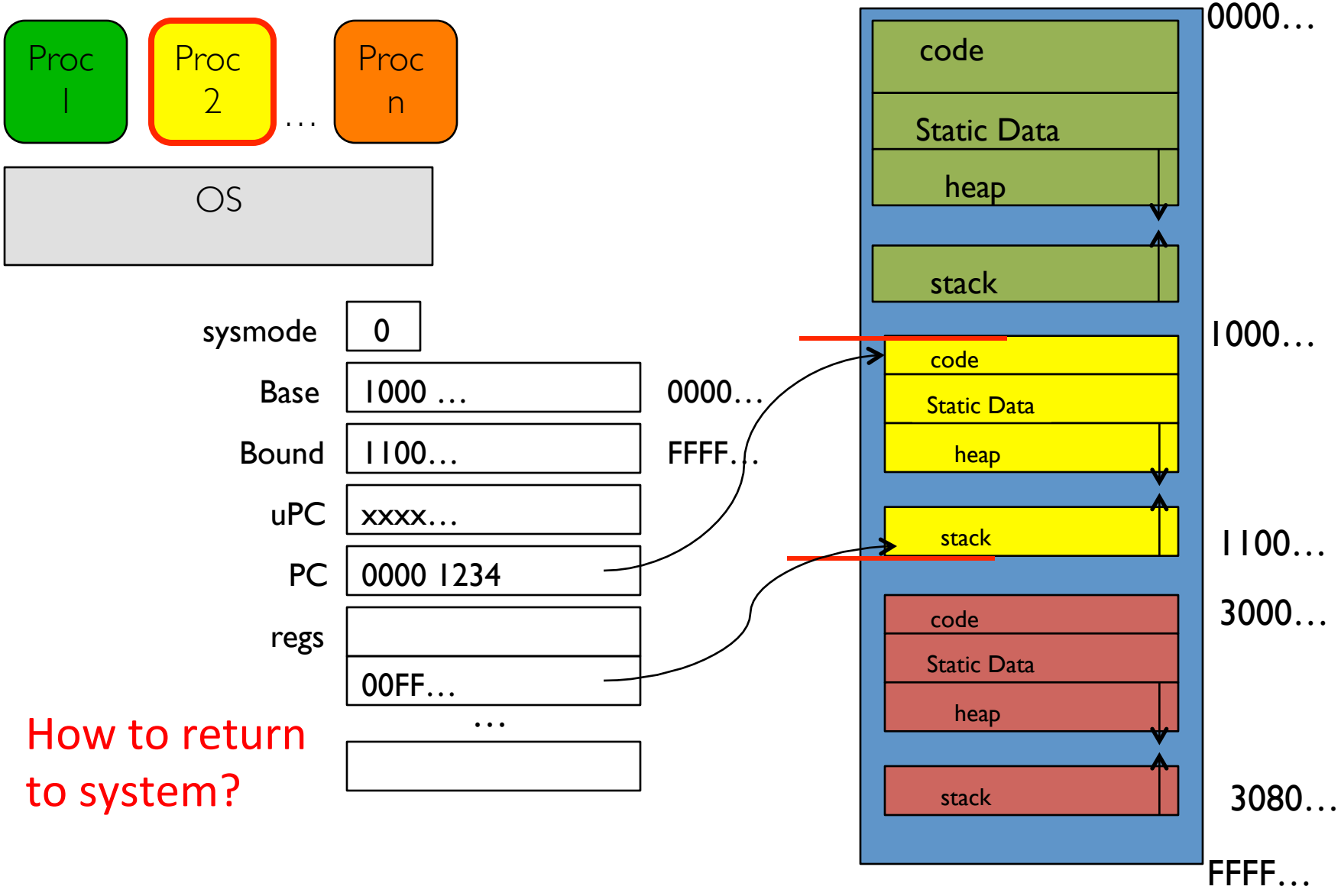
- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but "outside" the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt
  - External asynchronous event triggers context switch
  - e. g., Timer, I/O device
  - Independent of user process
- Trap or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …
- All 3 are an UNPROGRAMMED CONTROL TRANSFER
  - Where does it go?

How do we get the system target address of the “unprogrammed control transfer?”

Interrupt Vector

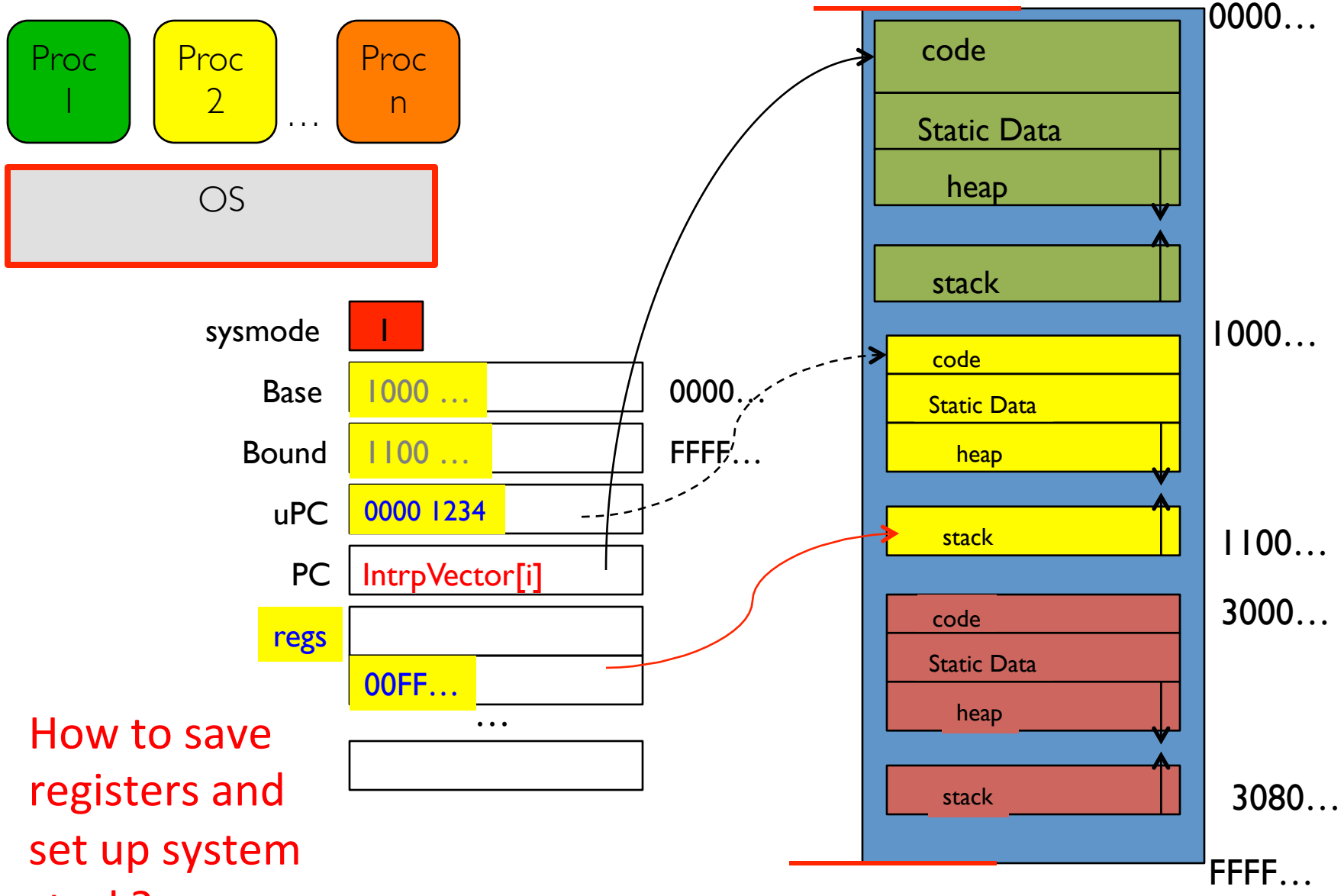


Simple B&B: User => Kernel



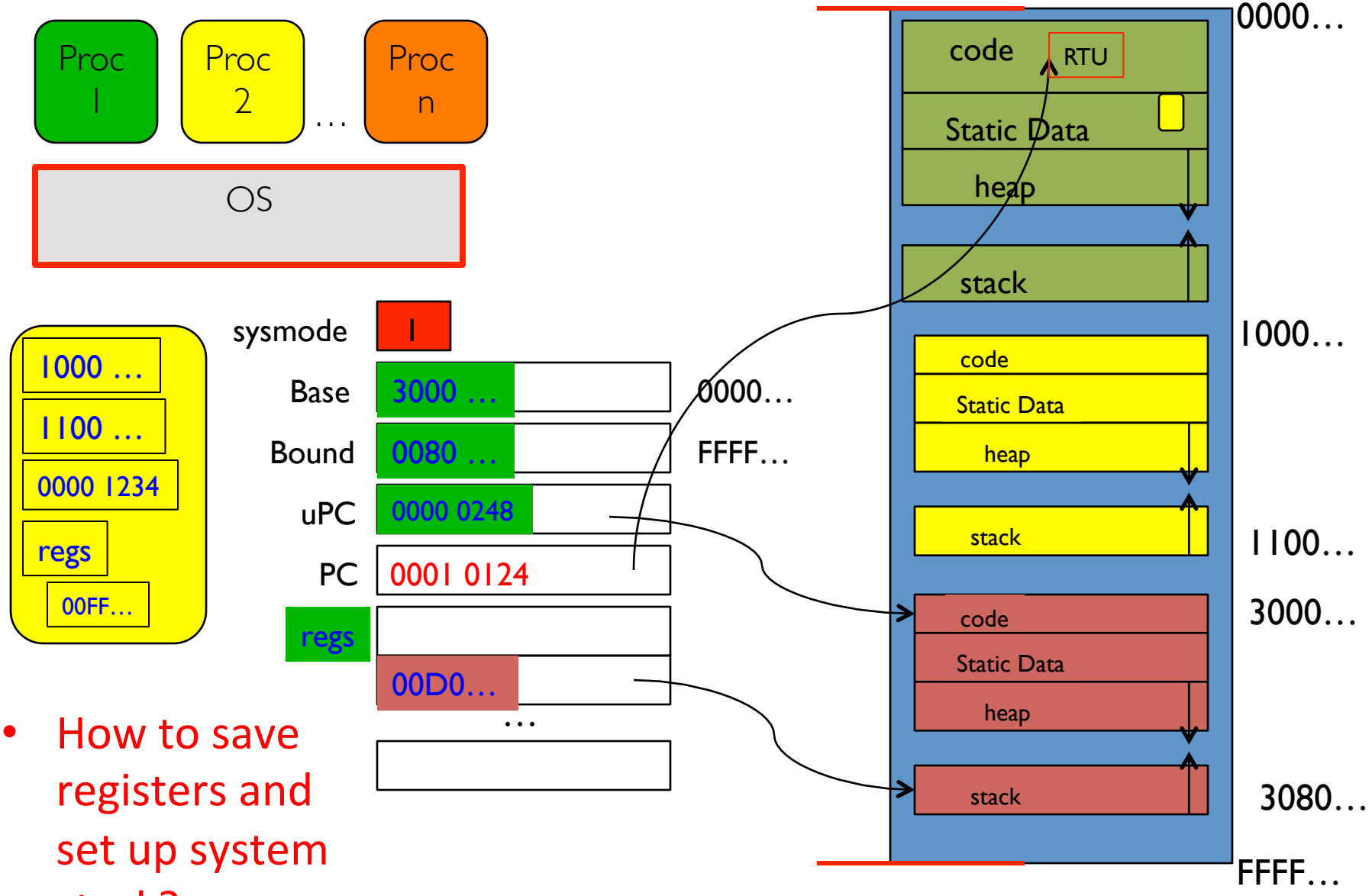
- How to return to system?

Simple B&B: Interrupt



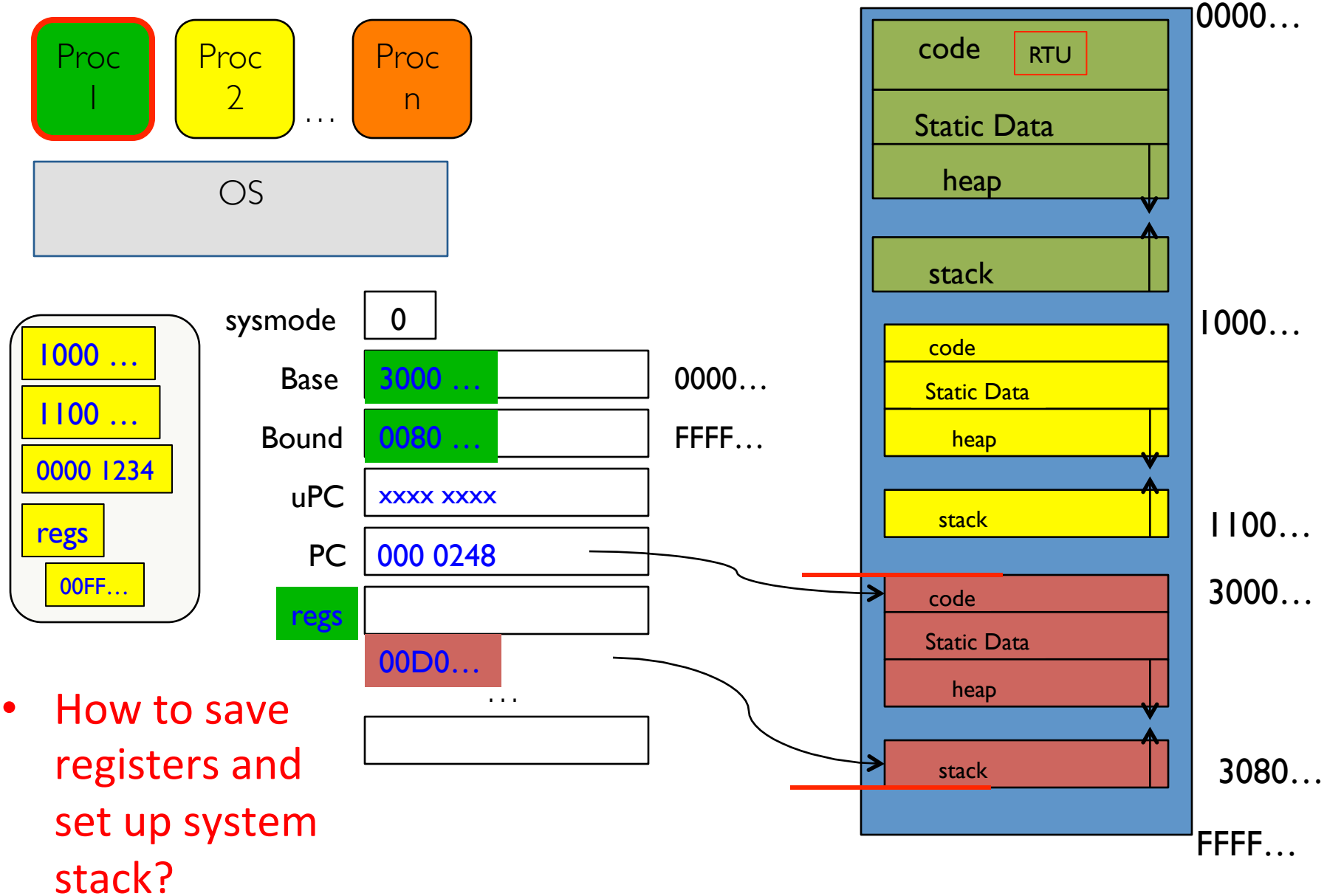
- How to save registers and set up system stack?

Simple B&B: Switch User Process



- How to save registers and set up system stack?

Simple B&B: "resume"

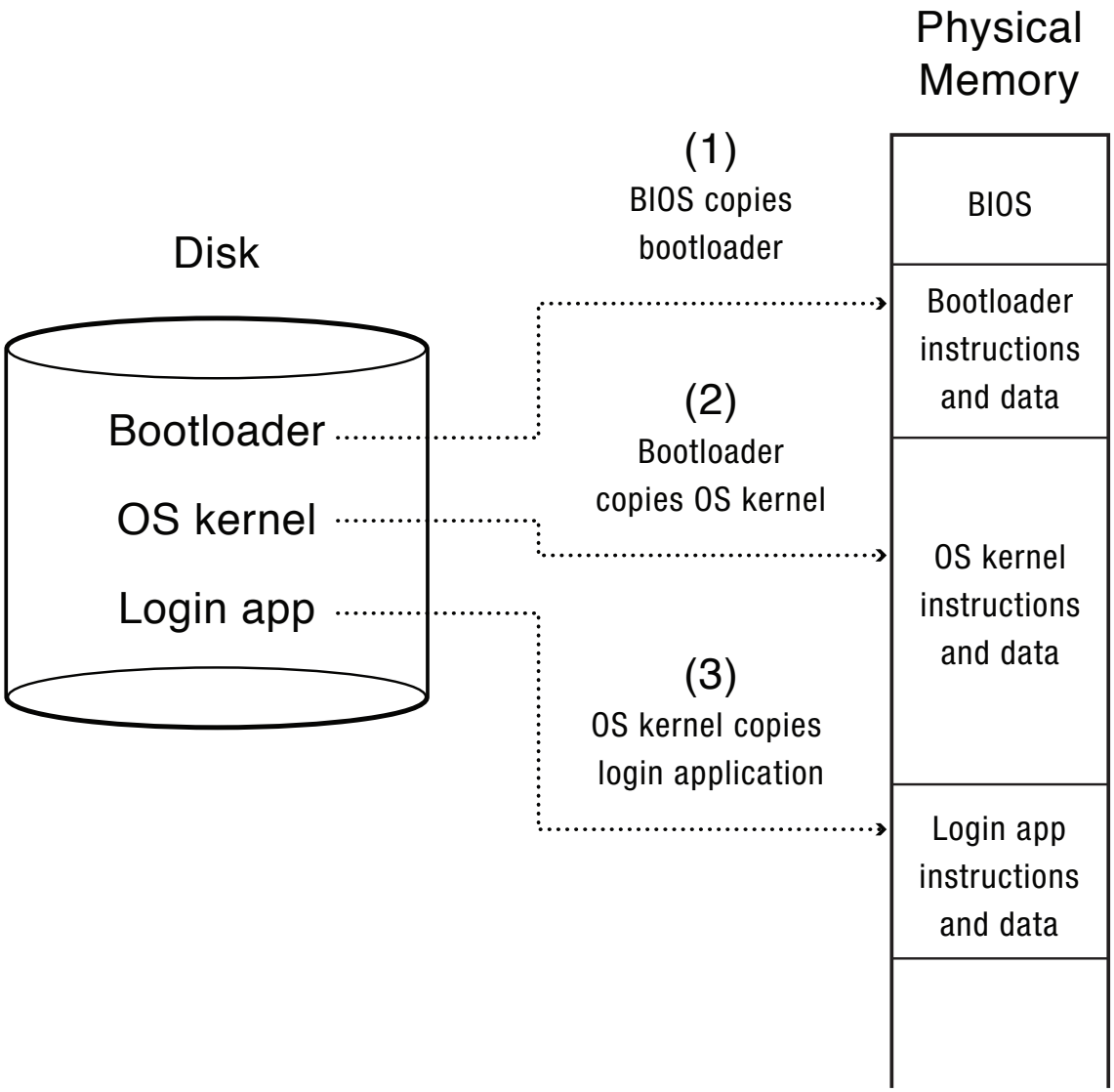


- How to save registers and set up system stack?

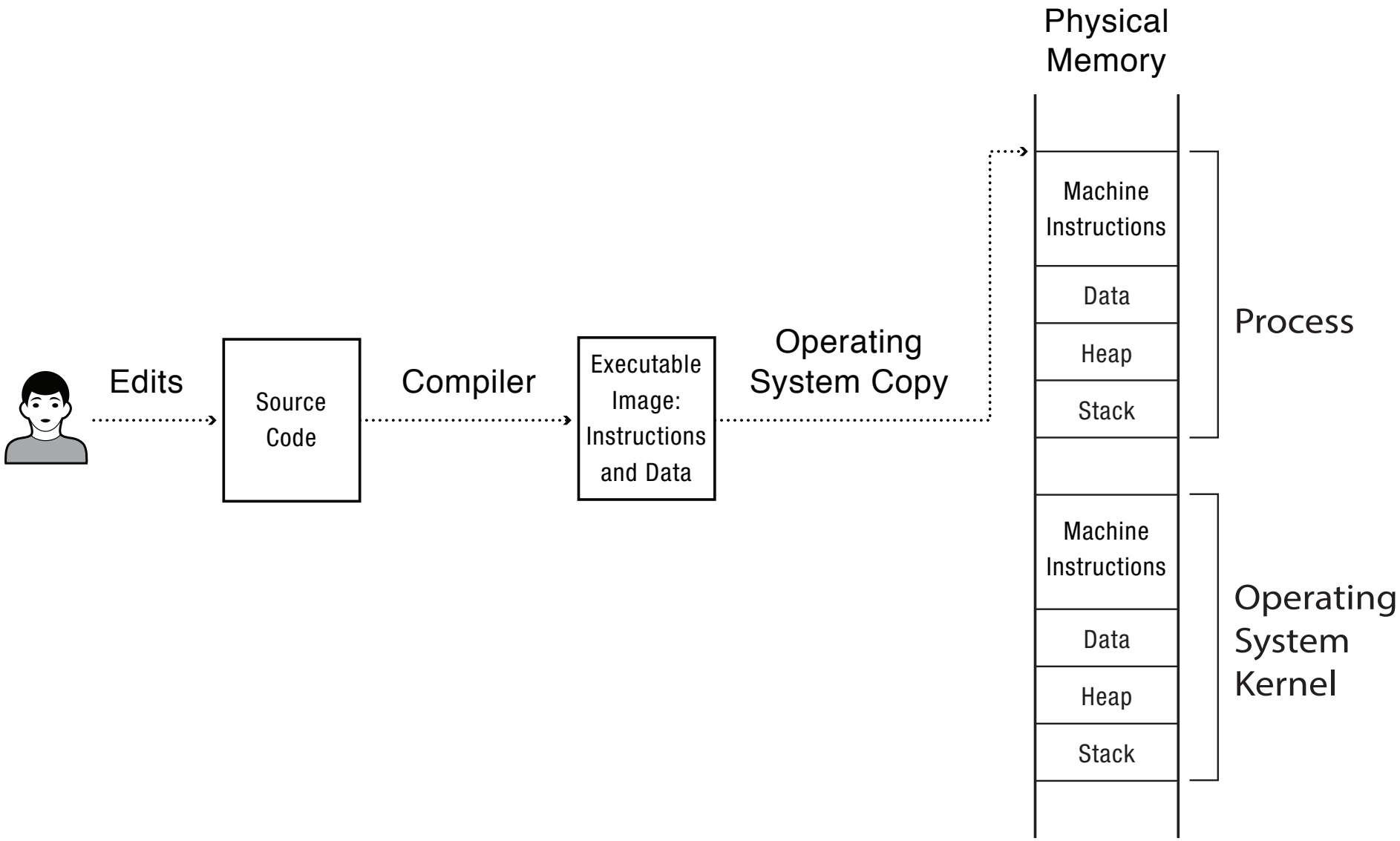
Conclusion: Four fundamental OS concepts

- Thread
  - Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack
- Address Space with Translation
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine
- Process
  - An instance of an executing program is *a process consisting of an address space and one or more threads of control*
- Dual Mode operation/Protection
  - Only the "system" has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses

Booting


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

A Problem


Physical Memory
Edits
Source Code
Compiler
Executable Image: Instructions and Data
Operating System Copy
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
Operating System Kernel

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

Basic OS Concepts

- Thread
  - Single execution context: fully describes program state
  - PC, registers, execution flag, stack
- Address space (with translation)
  - Programs execute in address space that is different from memory space on the physical machine
- Process
  - An instance of an executing program is a process consisting of an address space and one or more threads of control
- Dual mode operation / Protection
  - Only the “system” has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by controlling the translation from program virtual addresses to machine physical addresses

Process Abstraction

- Process: an *instance* of a program, running with limited rights
  - Thread: a sequence of instructions within a process
    - Potentially many threads per process (for now 1:1)
  - Address space: set of rights of a process
    - Memory that the process can access
    - Other permissions the process has (e.g., which system calls it can make, what files it can access)

Thought Experiment

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

Hardware Support: 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

Hardware Support: 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

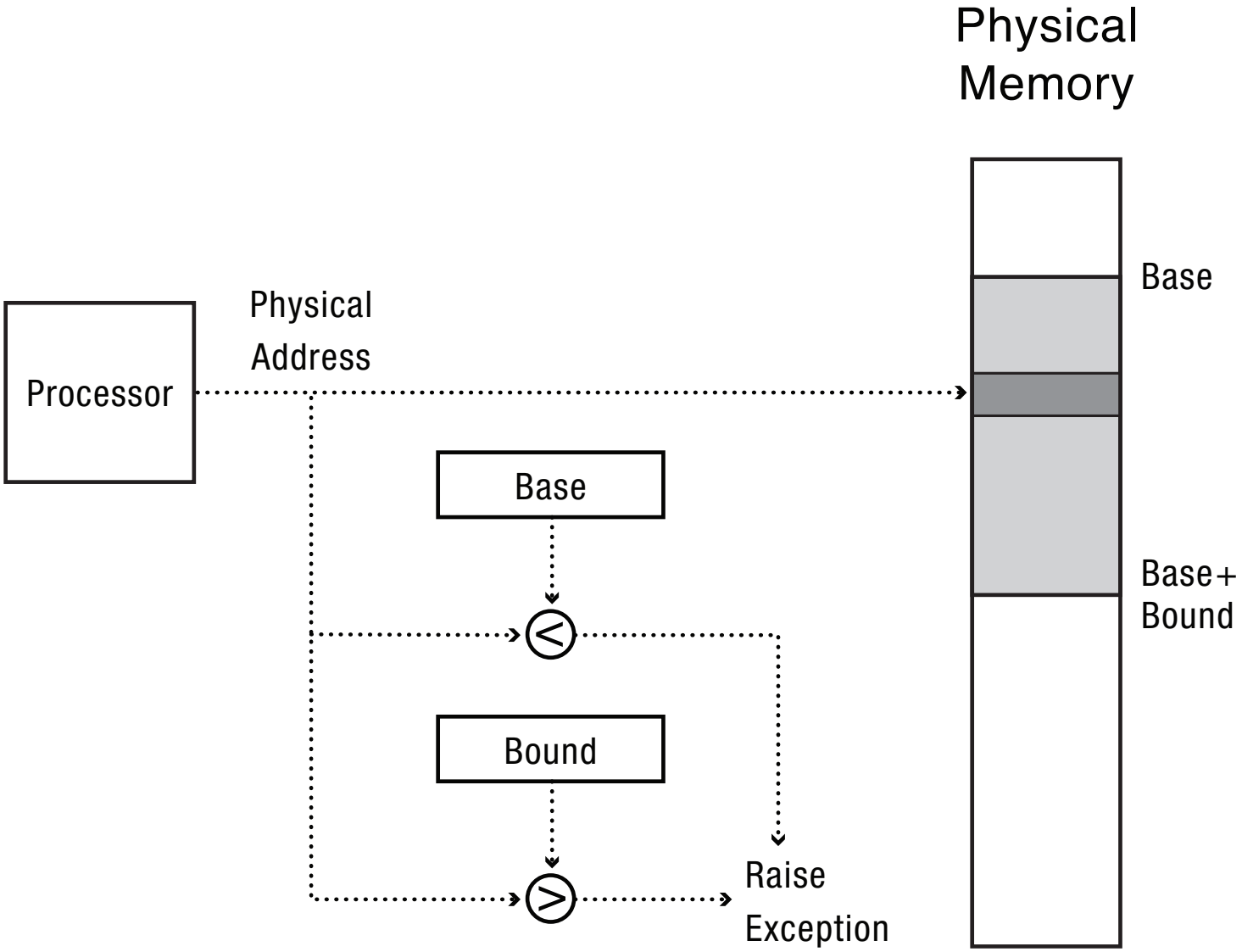
- Examples?

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

Question

- For a “Hello world” program, the kernel must copy the string from the user program memory into the screen memory.
- Why not allow the application to write directly to the screen’s buffer memory?

Simple Memory Protection

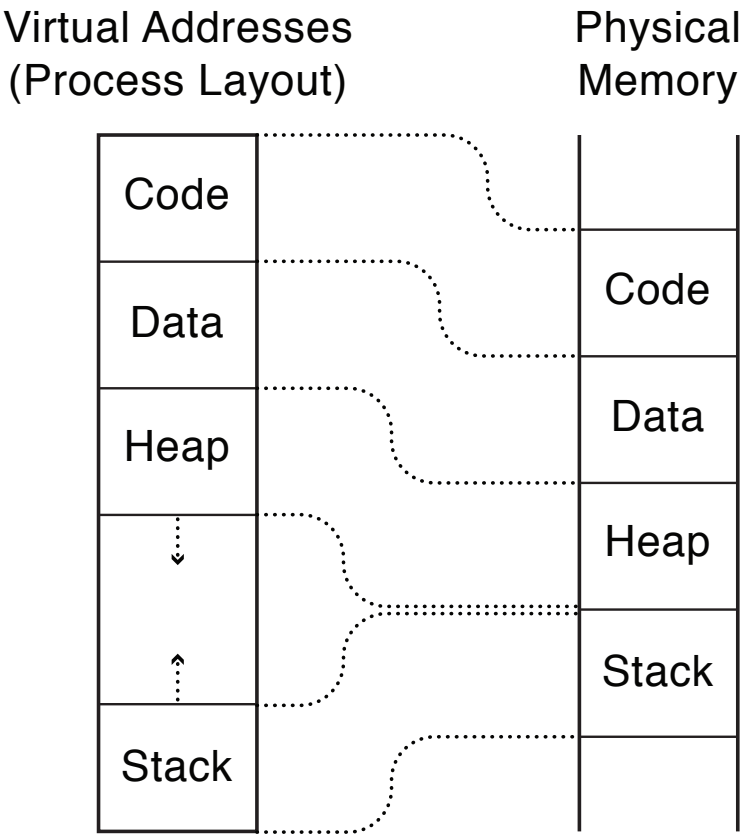


Towards Virtual Addresses

- Problems with base and bounds?

Virtual Addresses

- Translation done in hardware, using a table
- Table set up by operating system kernel



Example

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

What happens if we run two instances of this program at the same time?

What if we took the address of a procedure local variable in two copies of the same program running at the same time?

Question

- With an object-oriented language and compiler, only an object's methods can access the internal data inside an object. If the operating system only ran programs written in that language, would it still need hardware memory address protection?
- What if the contents of every object were encrypted except when its method was running, including the OS?

Hardware Timer

- Hardware device that periodically interrupts the processor
  - Returns control to the kernel handler
  - Interrupt frequency set by the kernel
    - Not by user code!
  - Interrupts can be temporarily deferred
    - Not by user code!
    - Interrupt deferral crucial for implementing mutual exclusion

Mode Switch

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

Question

- Examples of exceptions

- Examples of system calls

Mode Switch

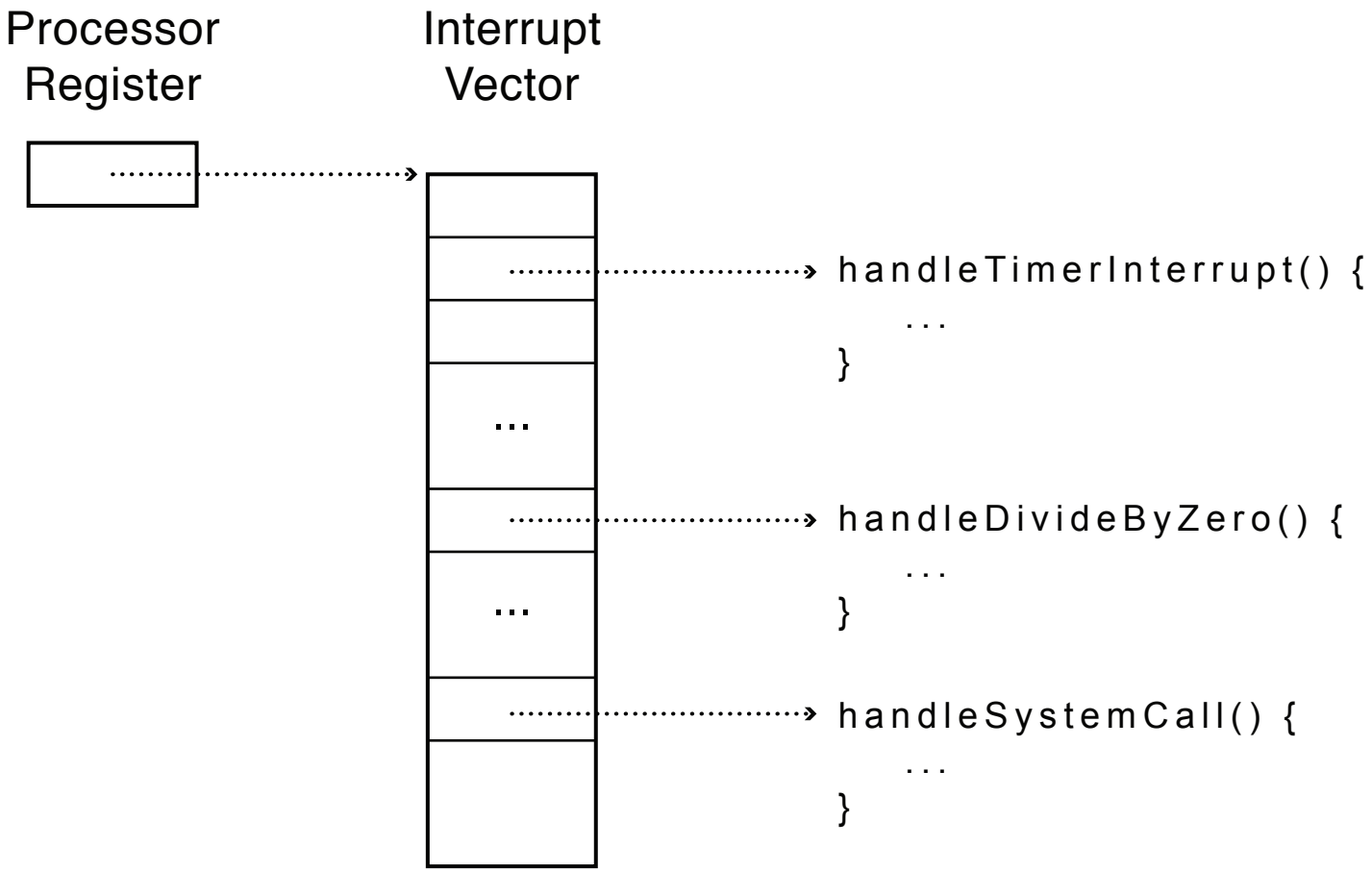
- From kernel mode to user mode
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall (UNIX signal)
    - Asynchronous notification to user program

How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Atomic transfer of control
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

Interrupt Vector

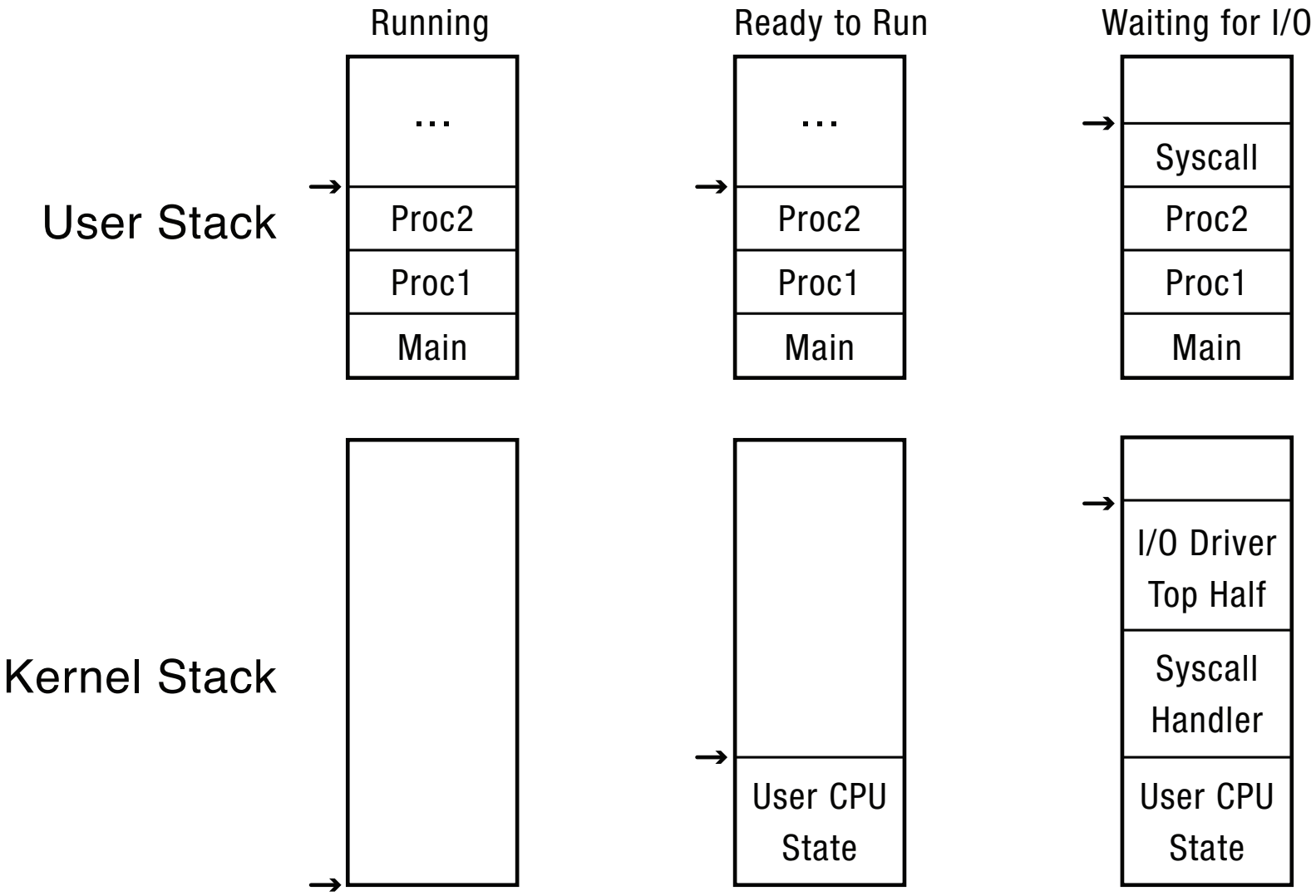
- Table set up by OS kernel; pointers to code to run on different events



Interrupt Stack

- Per-processor, located in kernel (not user) memory
  - Usually a process/thread has both: kernel and user stack
- Why can’t the interrupt handler run on the stack of the interrupted user process?

Interrupt Stack



Interrupt Masking

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - On x86
    - CLI: disable interrrupts
    - STI: enable interrupts
    - Only applies to the current CPU (on a multicore)
- We'll need this to implement synchronization in chapter 5

Interrupt Handlers

- Non-blocking, run to completion
  - Minimum necessary to allow device to take next interrupt
  - Any waiting must be limited duration
  - Wake up other threads to do any real work
    - Linux: semaphore
- Rest of device driver runs as a kernel thread

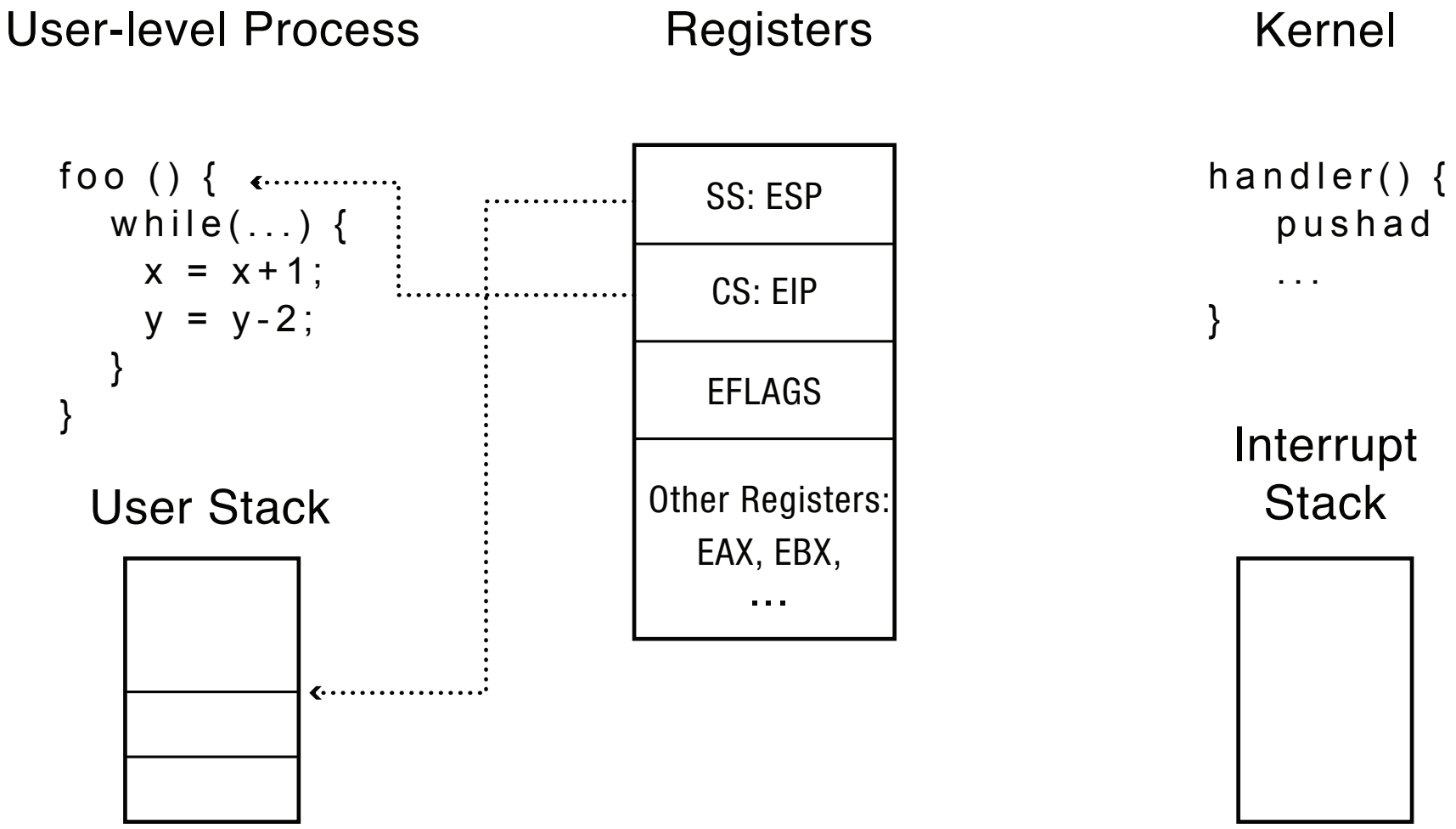
Case Study: MIPS Interrupt/Trap

- Two entry points: TLB miss handler, everything else
- Save type: syscall, exception, interrupt
  - And which type of interrupt/exception
- Save program counter: where to resume
- Save old mode, interruptable bits to status register
- Set mode bit to kernel
- Set interrupts disabled
- For memory faults
  - Save virtual address and virtual page
- Jump to general exception handler

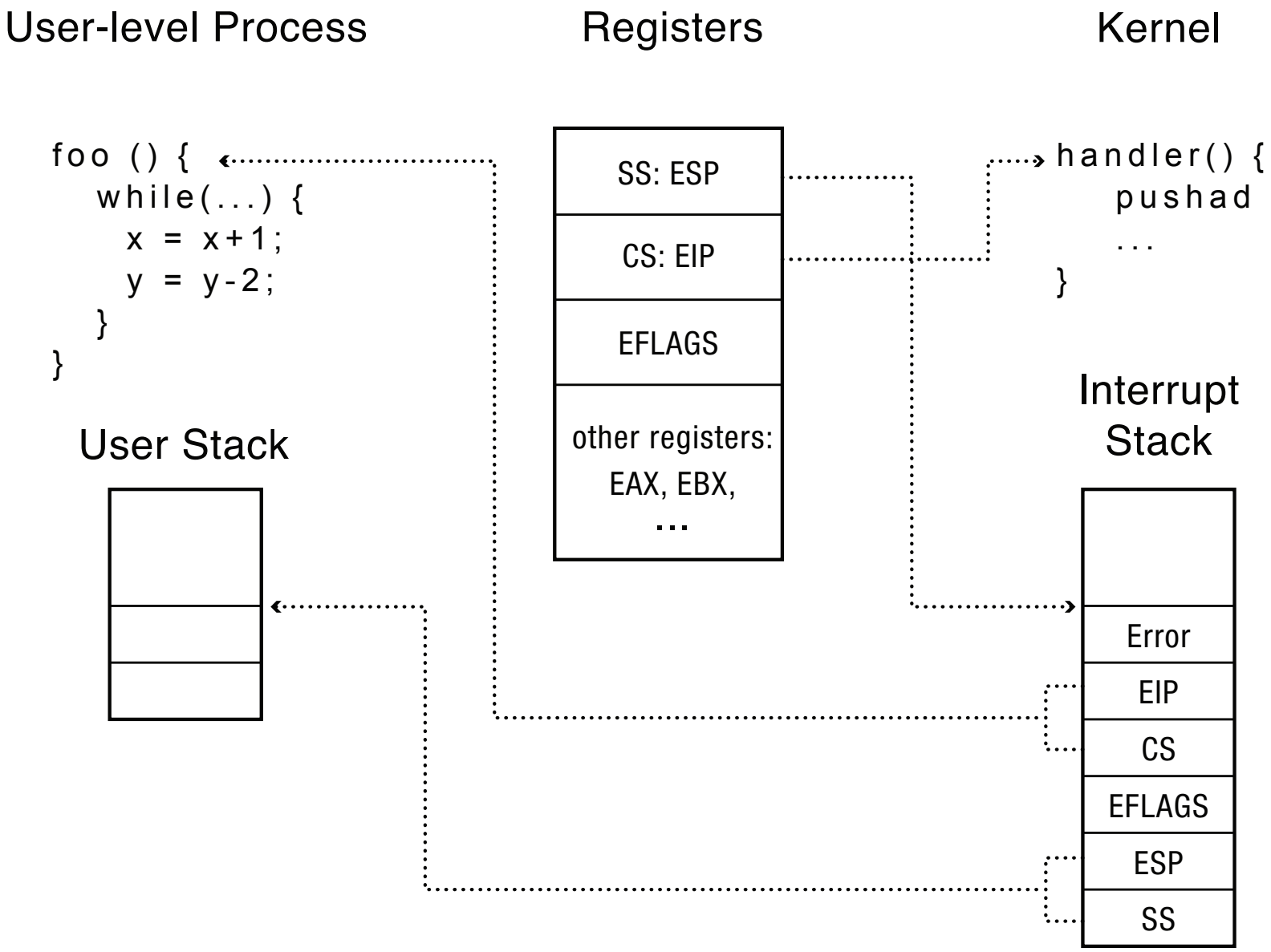
Case Study: x86 Interrupt

- Save current stack pointer
- Save current program counter
- Save current processor status word (condition codes)
- Switch to kernel stack; put SP, PC, PSW on stack
- Switch to kernel mode
- Vector through interrupt table
- Interrupt handler saves registers it might clobber

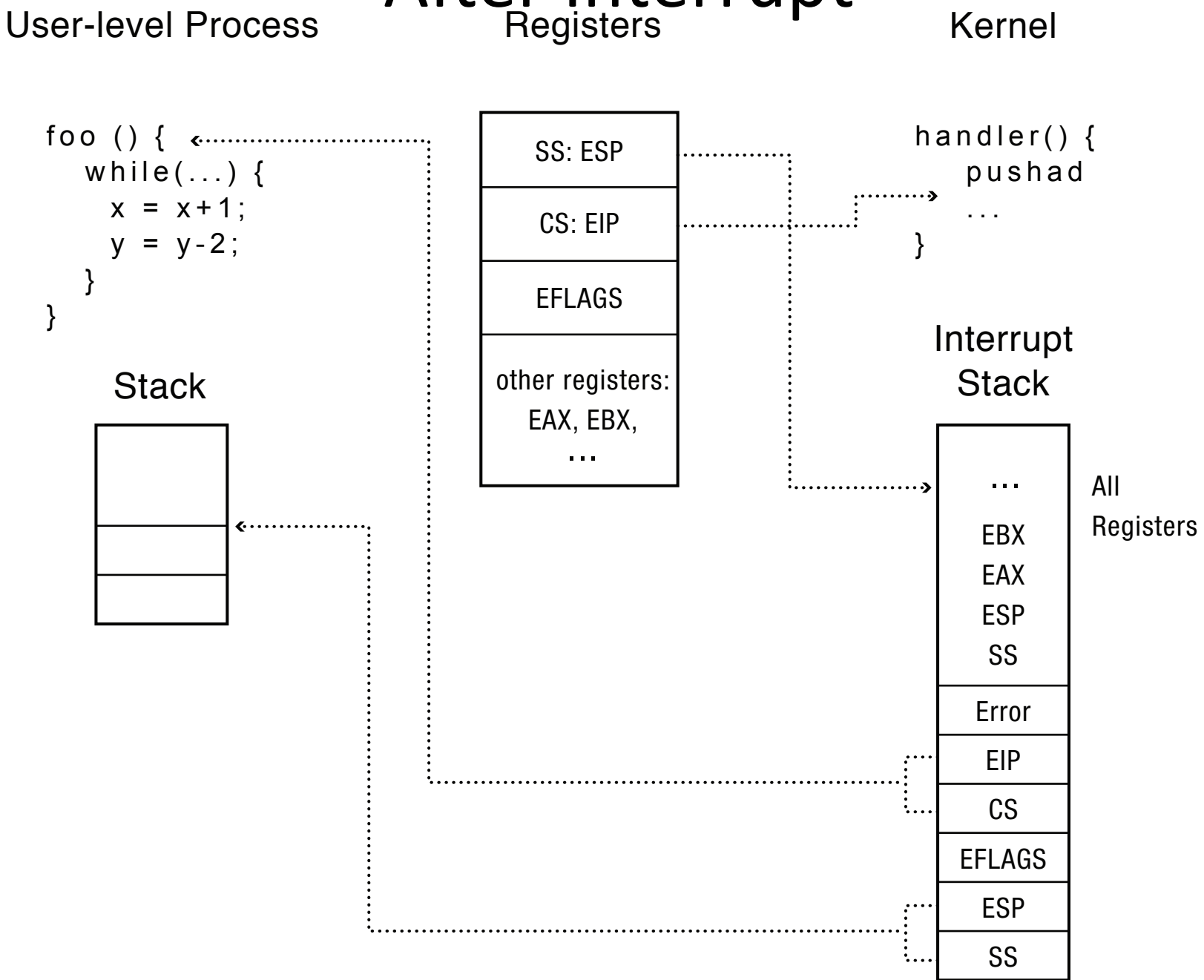
Before Interrupt



During Interrupt



After Interrupt



Question

- Why is the stack pointer saved twice on the interrupt stack?
  - Hint: is it the same stack pointer?

At end of handler

- Handler restores saved registers
- Atomically return to interrupted process/thread
  - Restore program counter
  - Restore program stack
  - Restore processor status word/condition codes
  - Switch to user mode

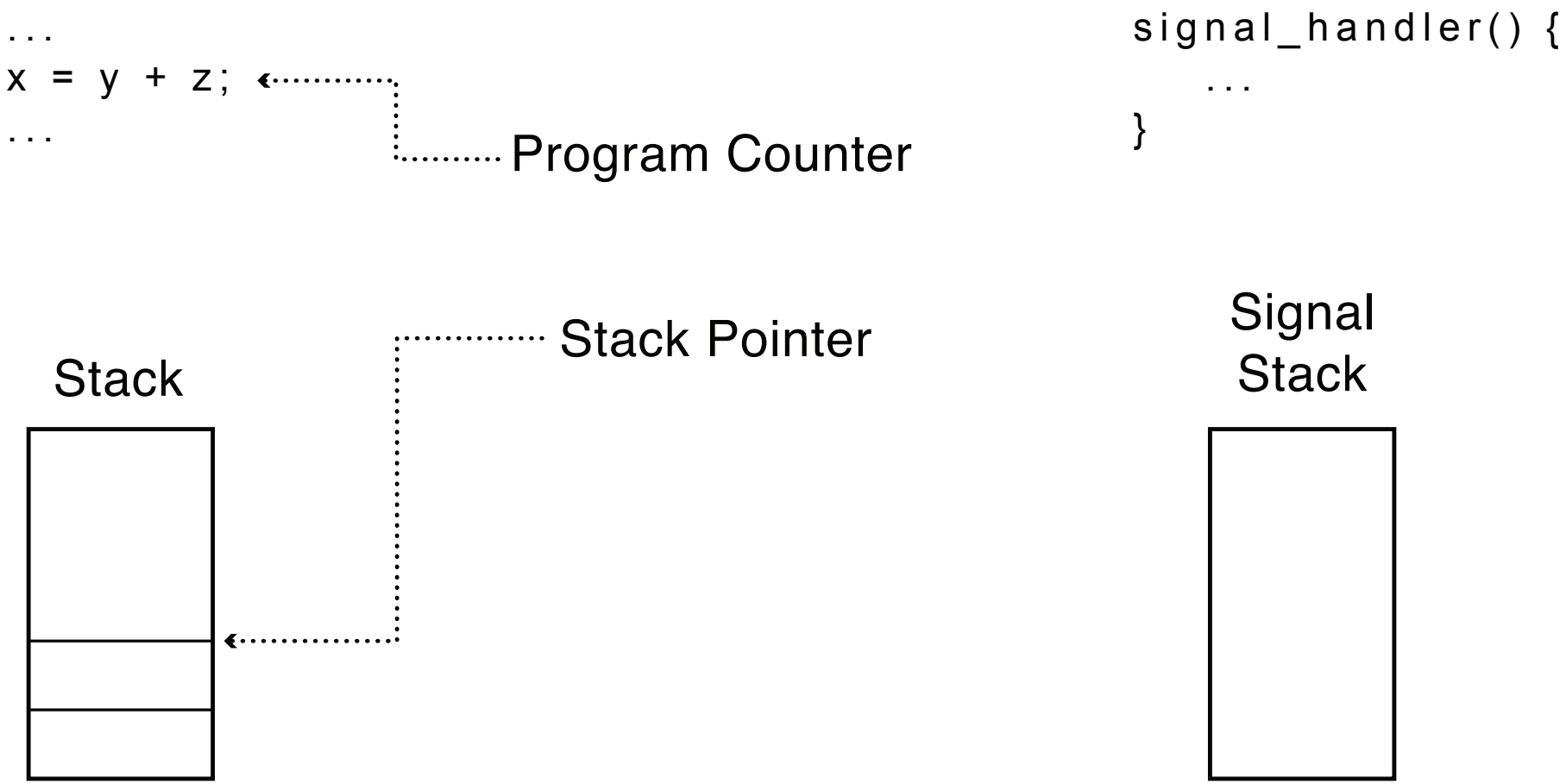
Upcall: User-level event delivery

- Notify user process of some event that needs to be handled right away
  - Time expiration
    - Real-time user interface
    - Time-slice for user-level thread manager
  - Interrupt delivery for VM player
  - Asynchronous I/O completion (async/await)
- AKA UNIX signal

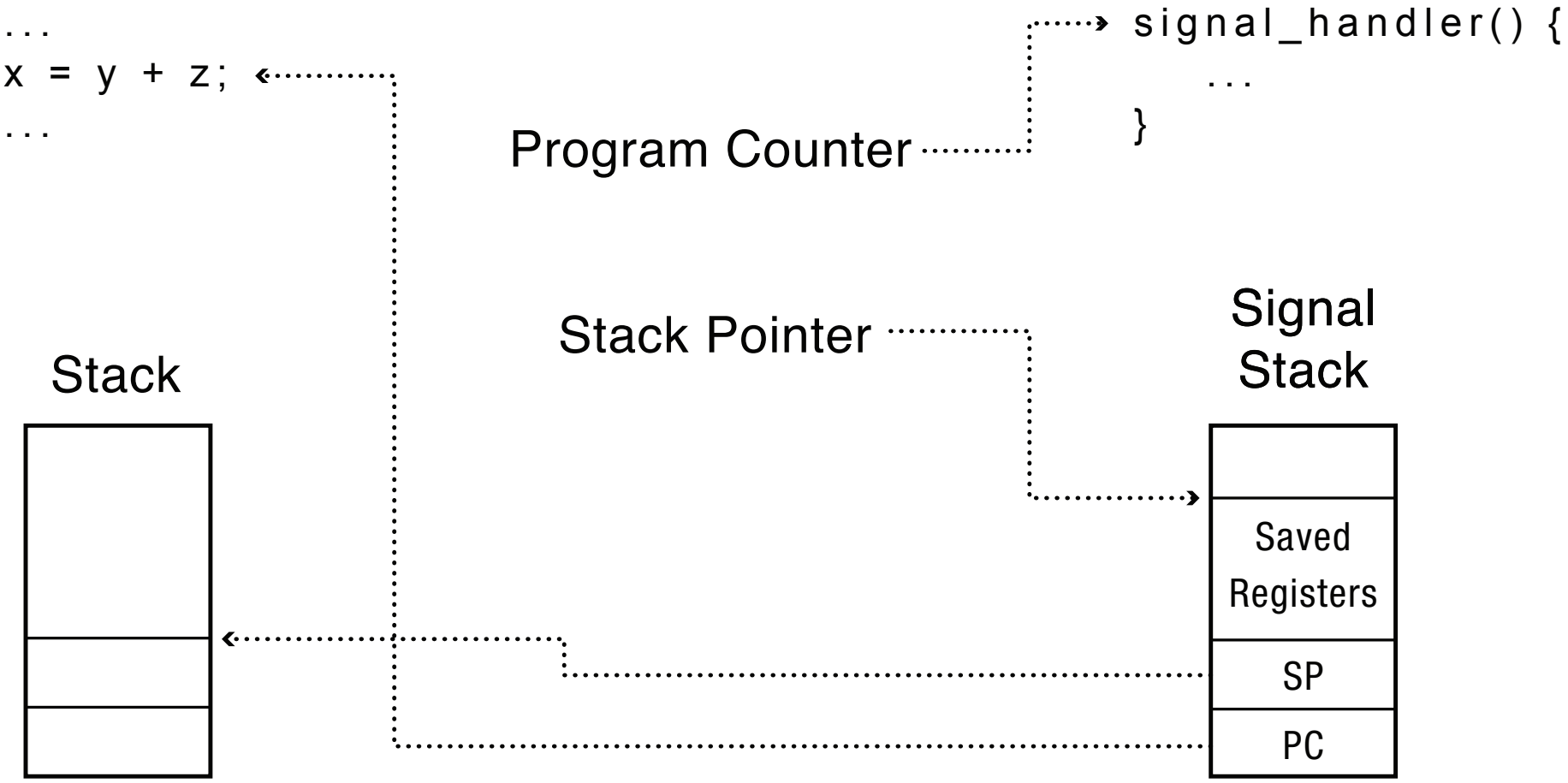
Upcalls vs Interrupts

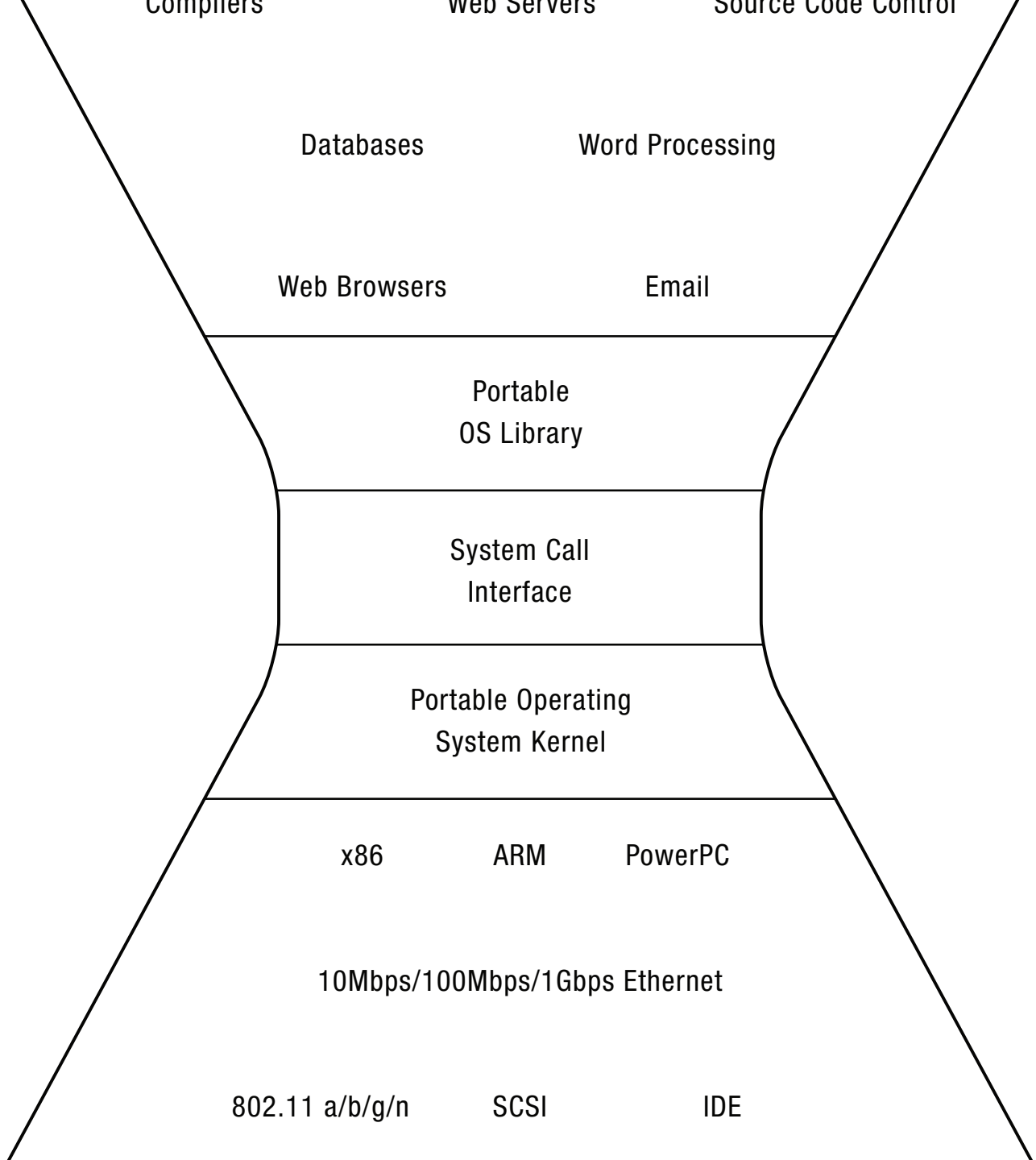
- Signal handlers = interrupt vector
- Signal stack = interrupt stack
- Automatic save/restore registers = transparent resume
- Signal masking: signals disabled while in signal handler

Upcall: Before



Upcall: During




Compilers
Web Servers
Source Code Control
Databases
Word Processing
Web Browsers
Email
Portable
OS Library
System Call
Interface
Portable Operating
System Kernel
x86
ARM
PowerPC
10Mbps/100Mbps/1Gbps Ethernet
802.11 a/b/g/n
SCSI
IDE

User Program

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

(1) (6)

User Stub

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

(2)

Hardware Trap

Trap Return

(5)

Kernel

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

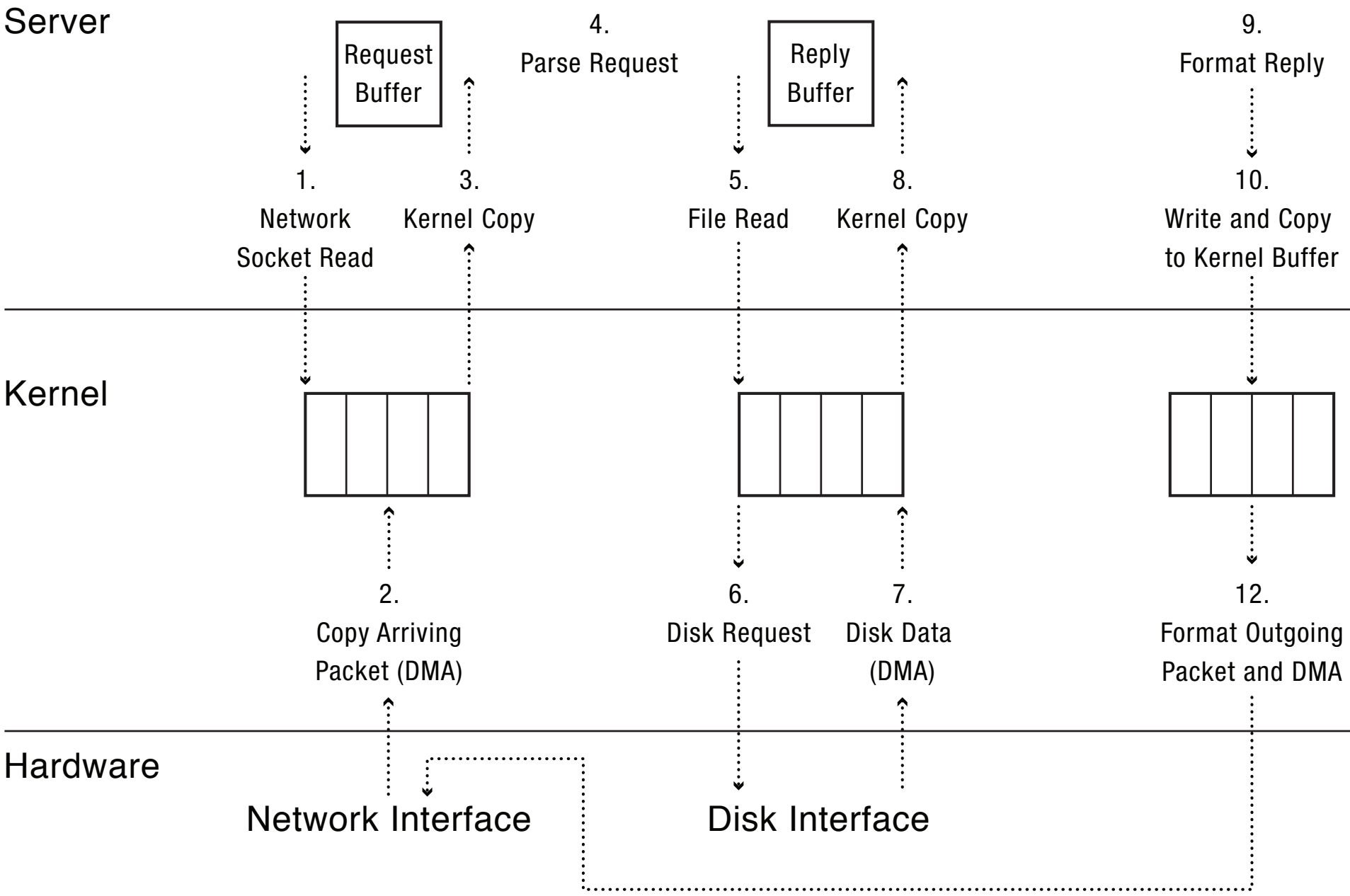
(3) (4)

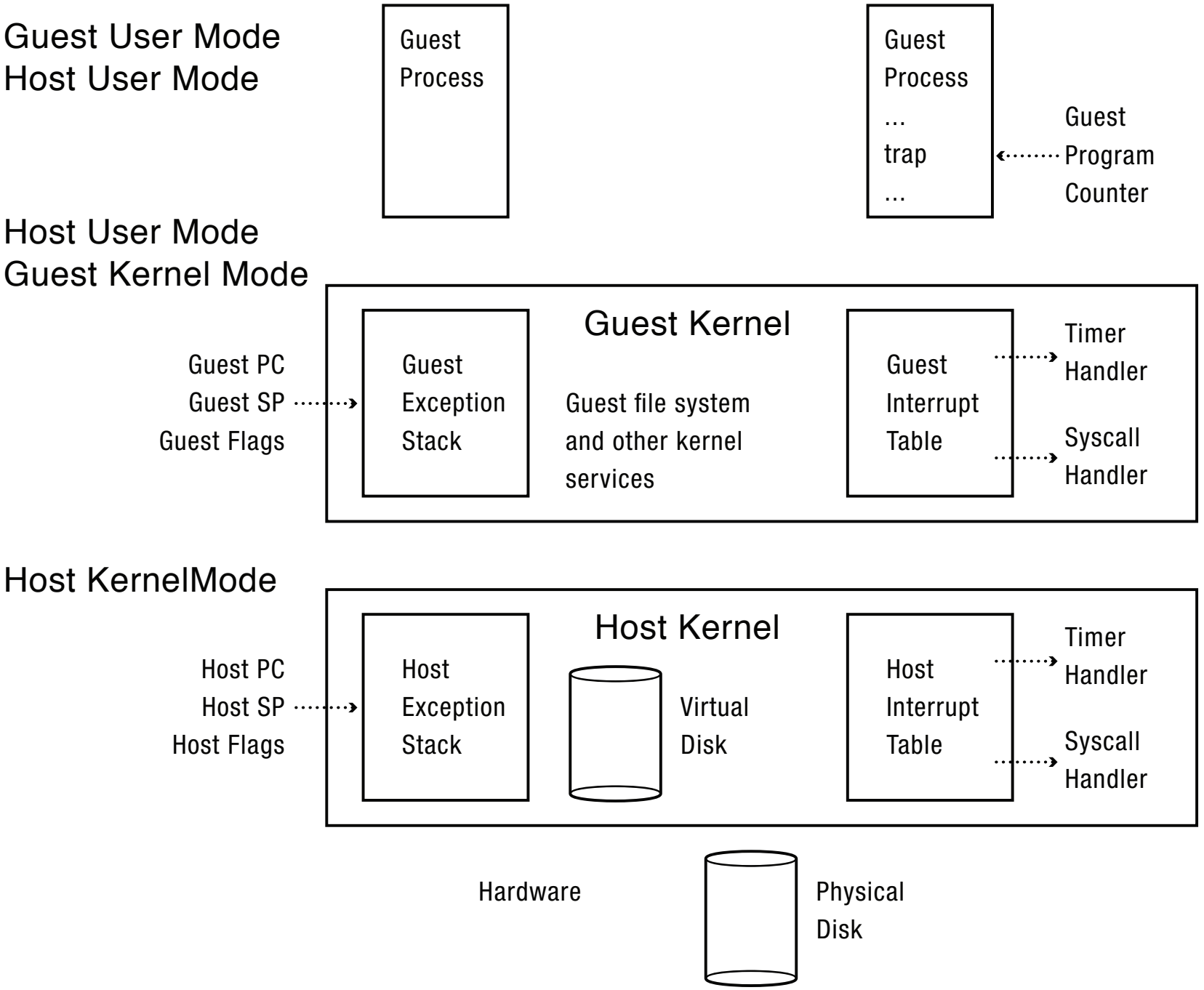
Kernel Stub

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

Kernel System Call Handler

- Locate arguments
  - In registers or on user stack
  - *Translate* user addresses into kernel addresses
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back into user memory
  - *Translate* kernel addresses into user addresses


Server
Request Buffer
4.
Parse Request
Reply Buffer
9.
Format Reply
1.
Network Socket Read
3.
Kernel Copy
5.
File Read
8.
Kernel Copy
10.
Write and Copy to Kernel Buffer
Kernel
2.
Copy Arriving Packet (DMA)
6.
Disk Request
7.
Disk Data (DMA)
12.
Format Outgoing Packet and DMA
Hardware
Network Interface
Disk Interface


Guest User Mode
Host User Mode
Guest Process
Guest Process
...
trap
...
Guest Program Counter
Host User Mode
Guest Kernel Mode
Guest Kernel
Guest PC
Guest SP
Guest Flags
Guest Exception Stack
Guest file system and other kernel services
Guest Interrupt Table
Timer Handler
Syscall Handler
Host KernelMode
Host Kernel
Host PC
Host SP
Host Flags
Host Exception Stack
Virtual Disk
Host Interrupt Table
Timer Handler
Syscall Handler
Hardware
Physical Disk

User-Level Virtual Machine

- How does VM Player work?
  - Runs as a user-level application
  - How does it catch privileged instructions, interrupts, device I/O?
- Installs kernel driver, transparent to host kernel
  - Requires administrator privileges!
  - Modifies interrupt table to redirect to kernel VM code
  - If interrupt is for VM, upcall
  - If interrupt is for another process, reinstalls interrupt table and resumes kernel