OS History and OS Structures

Karthik Dantu

CSE 421/521: Operating Systems

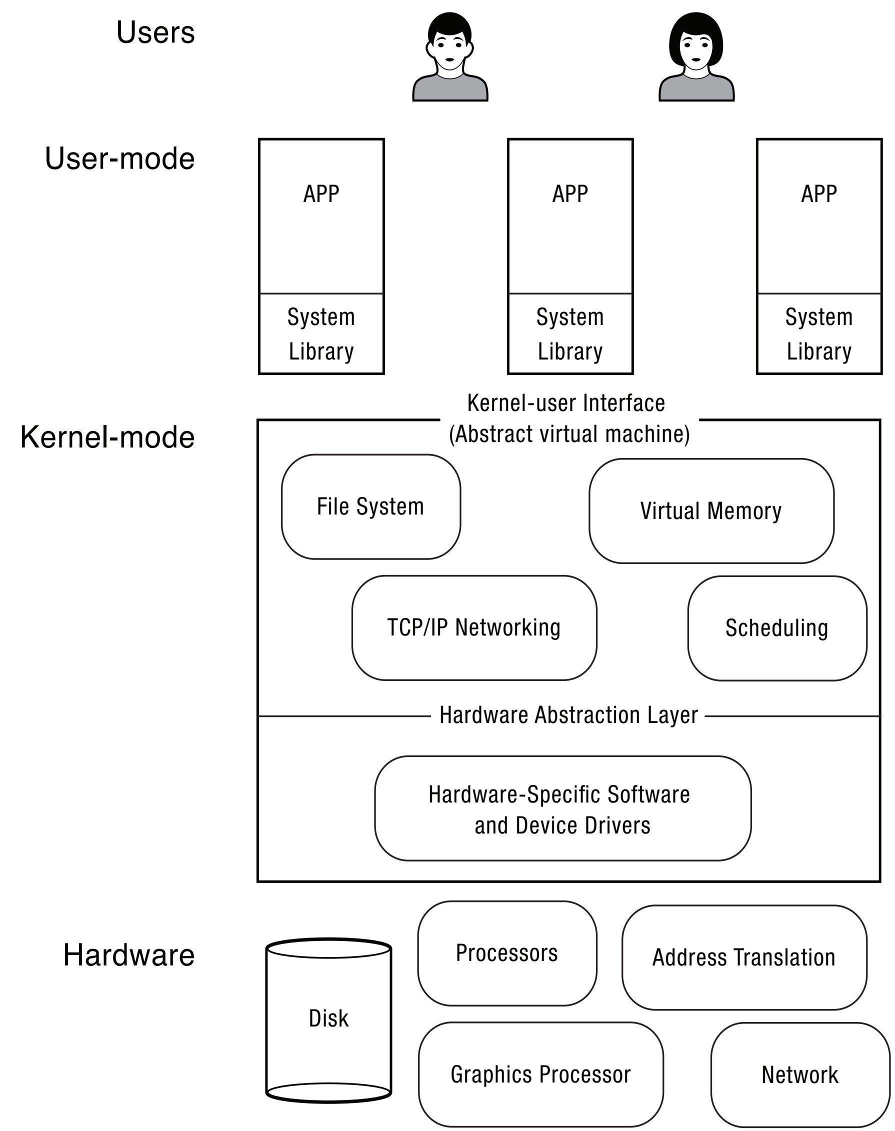
Slides adopted from CS162 class at Berkeley, CSE 451 at U-Washington and CSE 421 by Prof Kosar at UB

Action Items From Last Class

- Join Piazza
- Look through assignment#0
- Set up development environment: VirtualBox + Ubuntu 16.04
- Implement assignment and test in the environment
- Form groups

What is an OS?

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



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

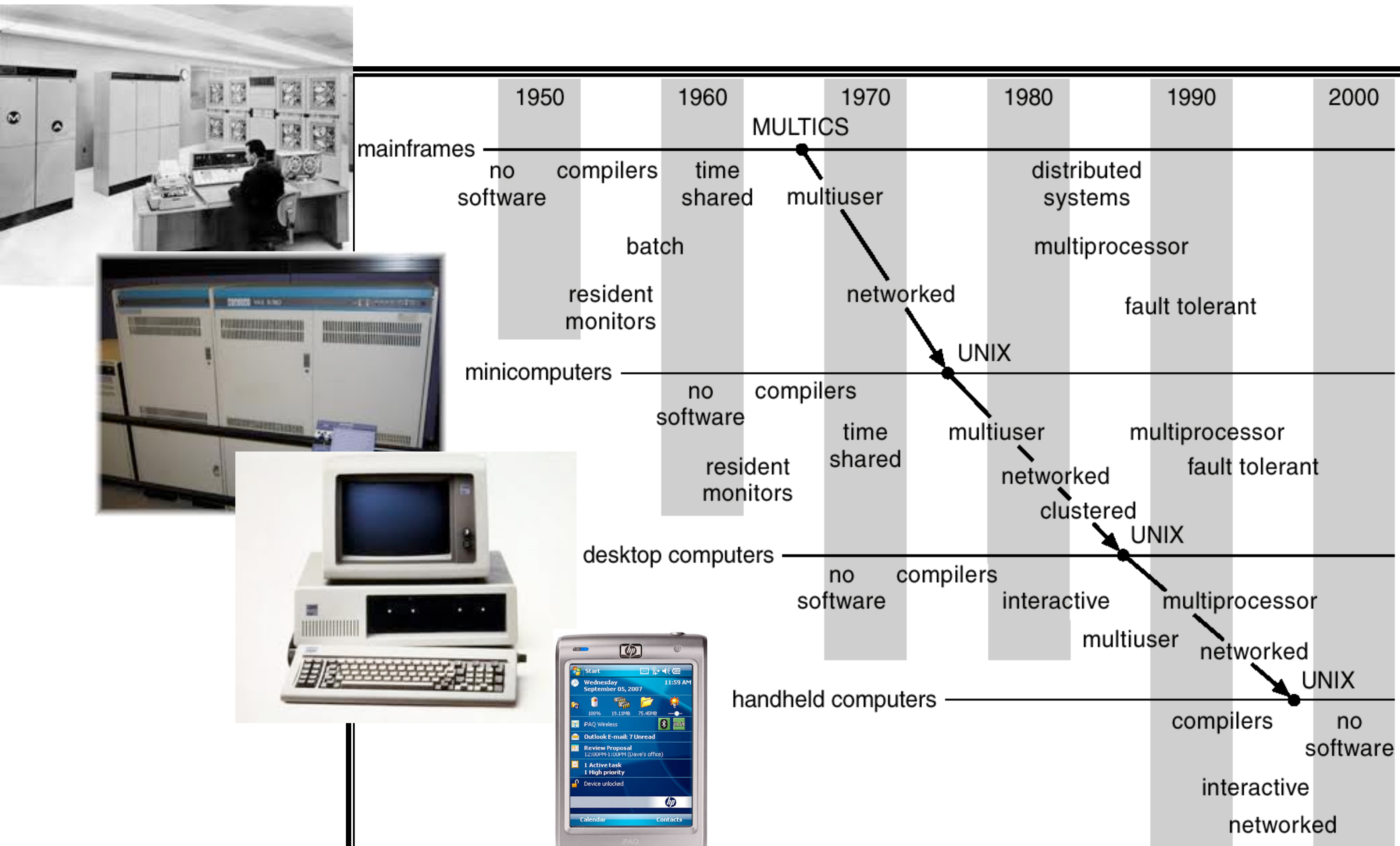
Today's Operating Systems: Computers Cheap

- 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

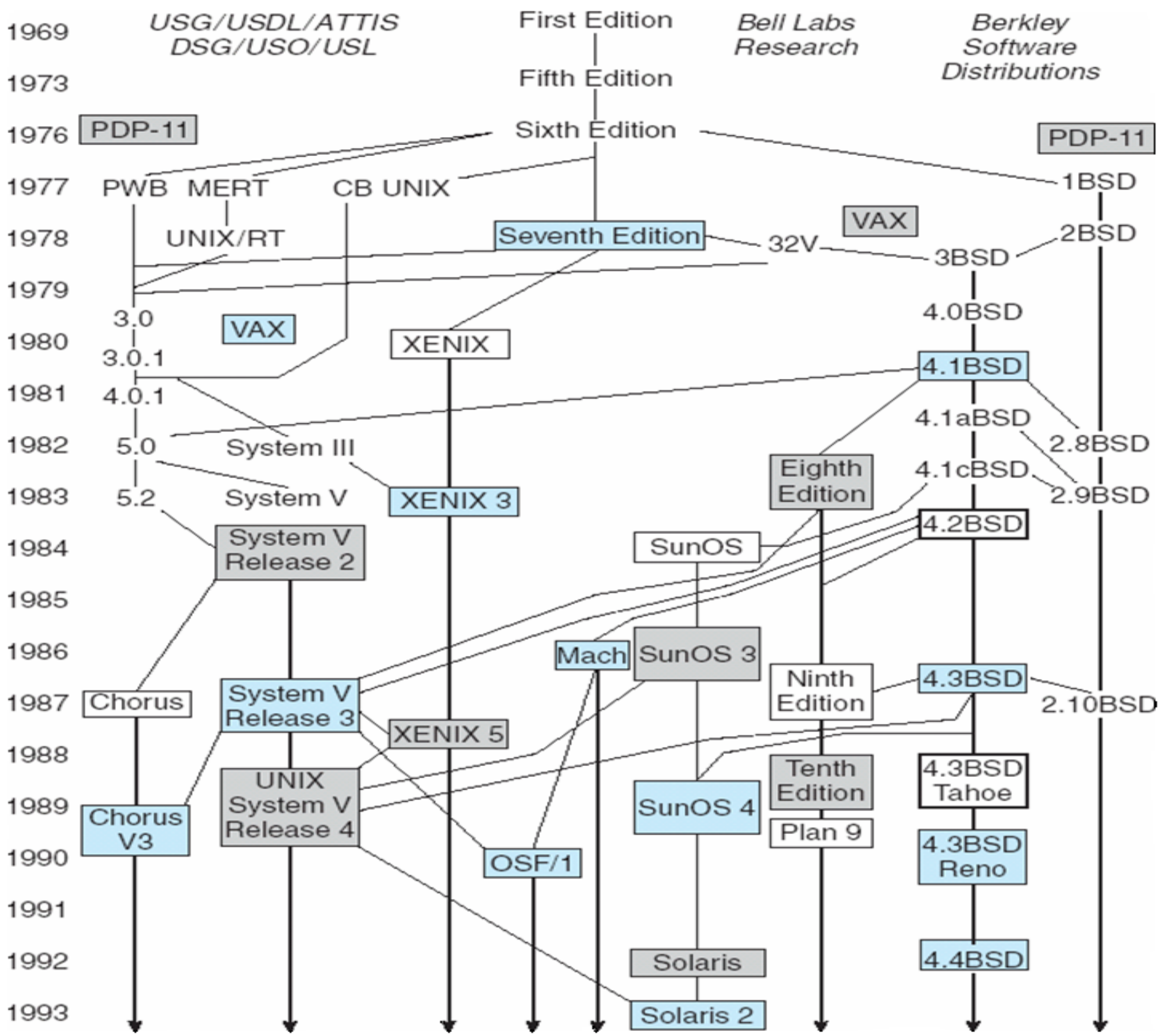
OS History


1950
1960
1970
1980
1990
2000
MULTICS
mainframes
no software
compilers
time shared
multiuser
distributed systems
batch
multiprocessor
resident monitors
networked
fault tolerant
UNIX
minicomputers
no software
compilers
time shared
multiuser
multiprocessor
resident monitors
networked
fault tolerant
clustered
UNIX
desktop computers
no software
compilers
interactive
multiprocessor
multiuser
networked
UNIX
handheld computers
compilers
no software
interactive
networked

Unix History

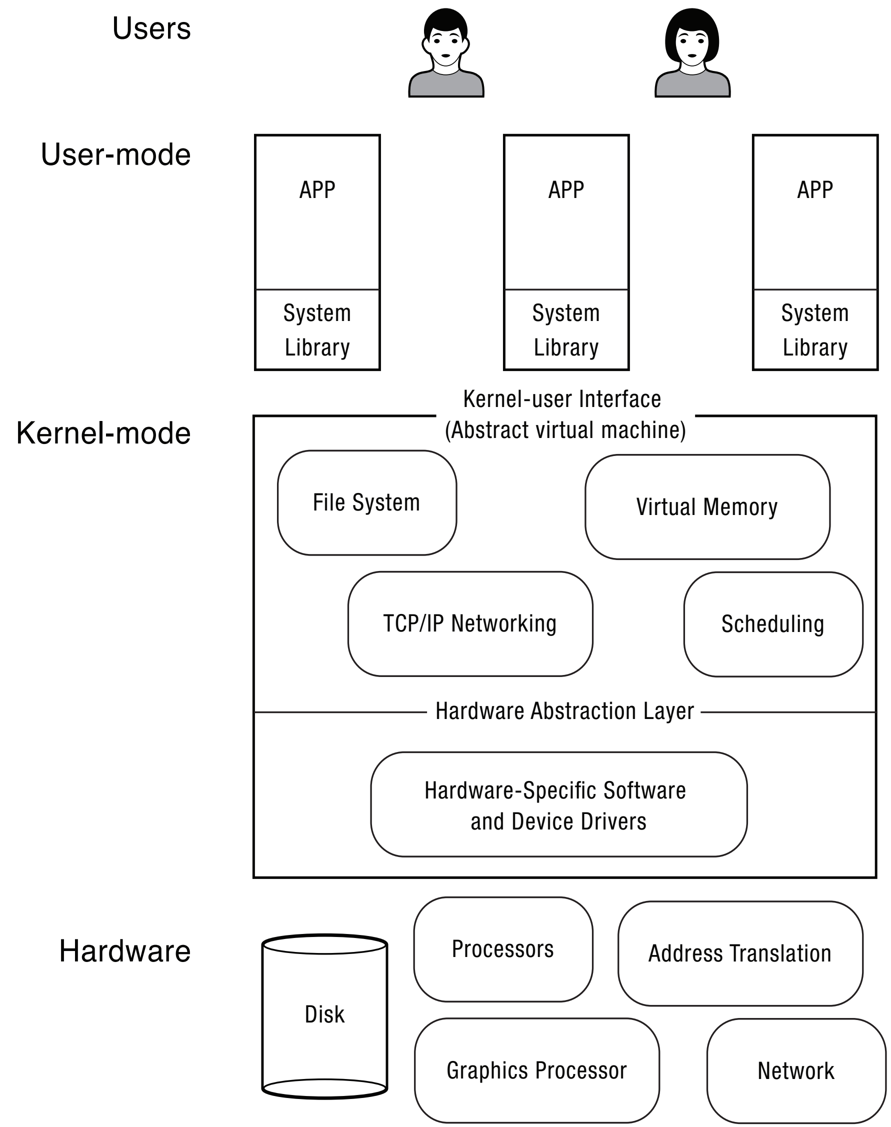
- First developed in 1969 by Ken Thompson and Dennis Ritchie of the Research Group at Bell Laboratories; incorporated features of other operating systems, especially MULTICS
- The third version was written in C, which was developed at Bell Labs specifically to support UNIX
- The most influential of the 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



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

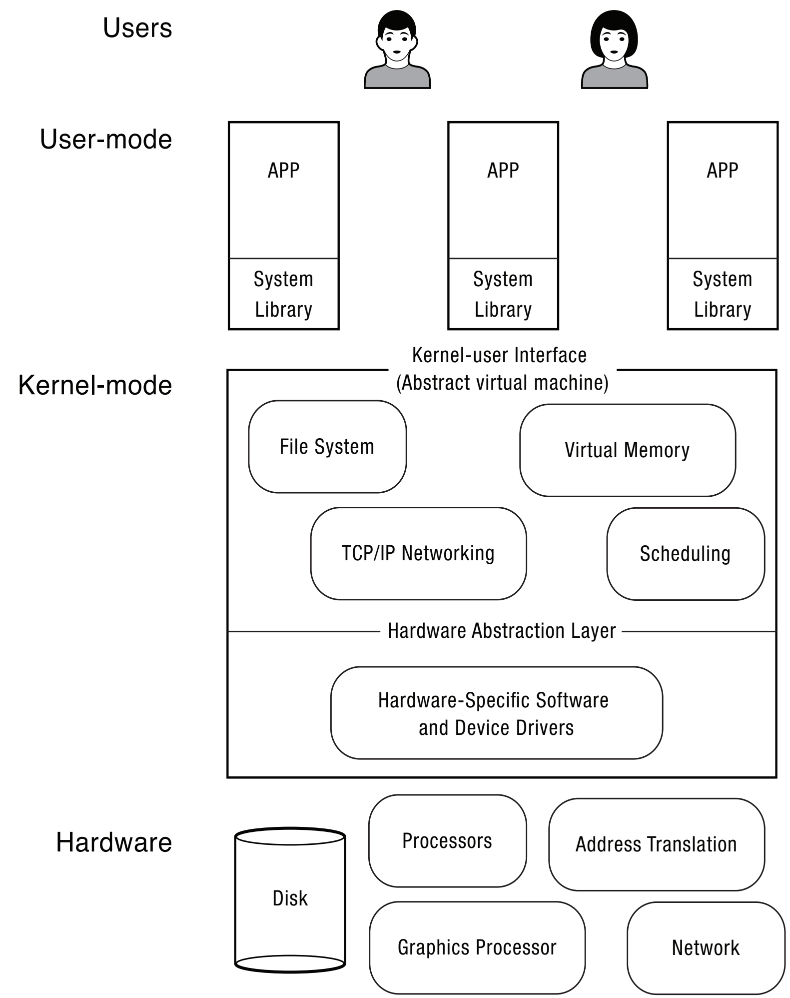
- What (hardware, software) do you need to be able to run an untrustworthy application?

OS Challenges - Correctness

- 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 – Wide Applicability

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



OS Challenges - Performance

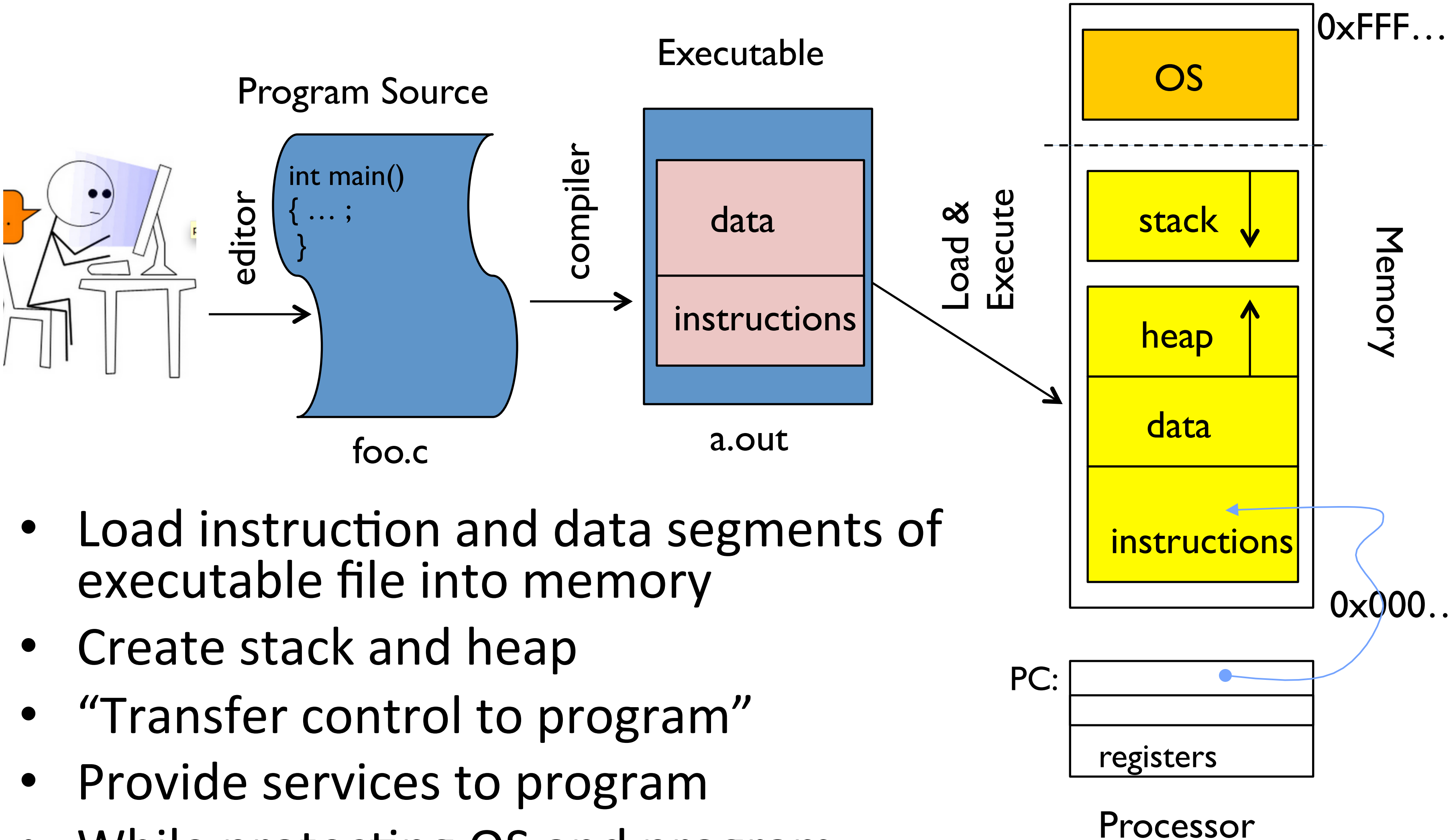
- Latency/response time
  - How long does an operation take to complete?
- Throughput
  - How many operations can be done per unit of time?
- Overhead
  - How much extra work is done by the OS?
- Fairness
  - How equal is the performance received by different users?
- Predictability
  - How consistent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

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

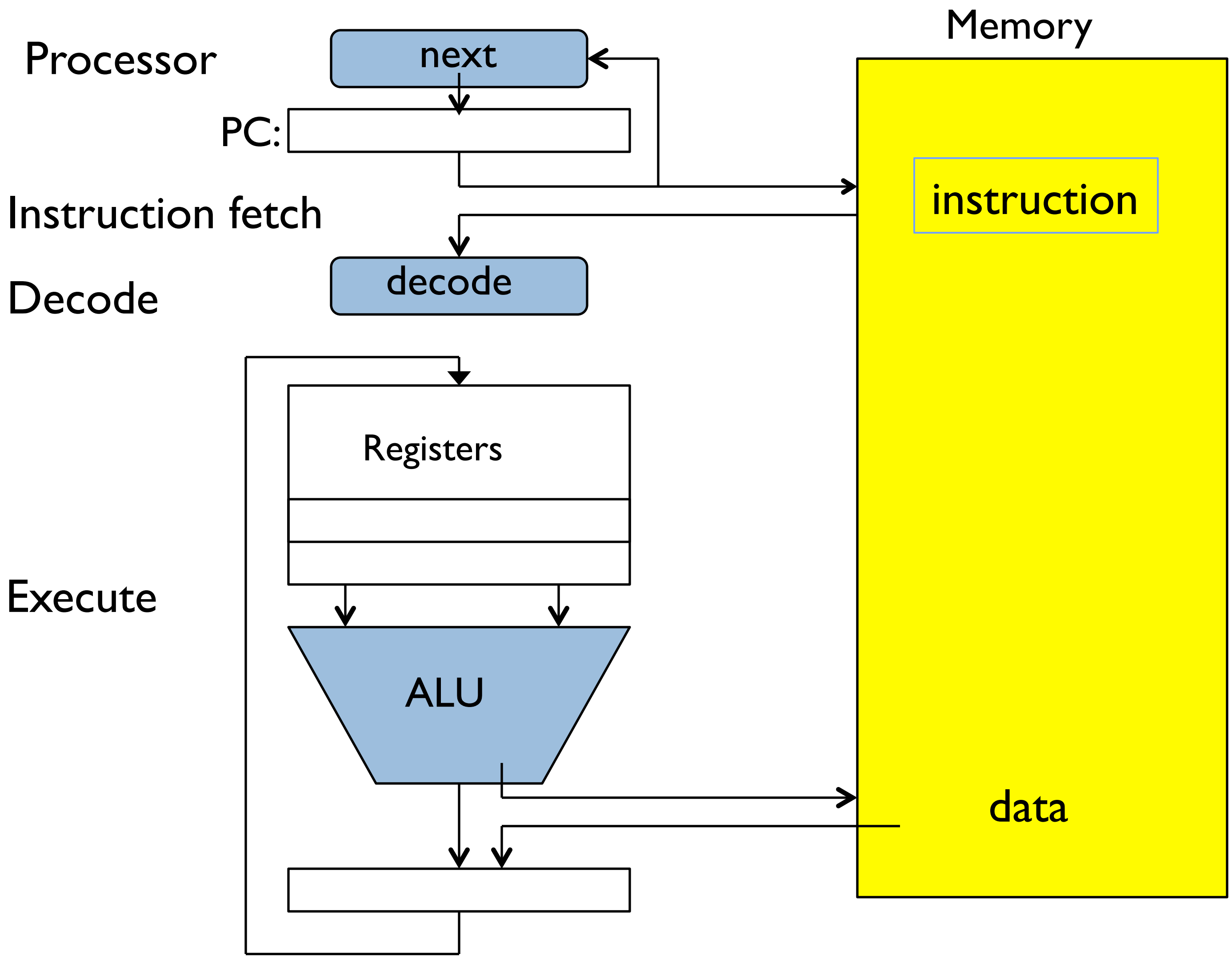
OS Bottom Line: Run Programs



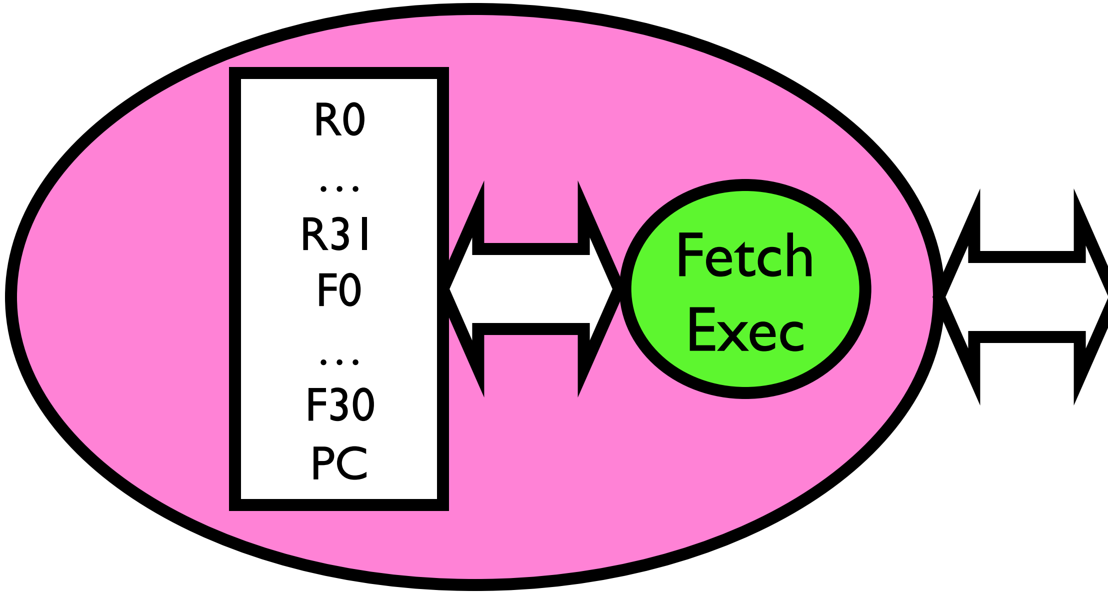
- Load instruction and data segments of executable file into memory
- Create stack and heap
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

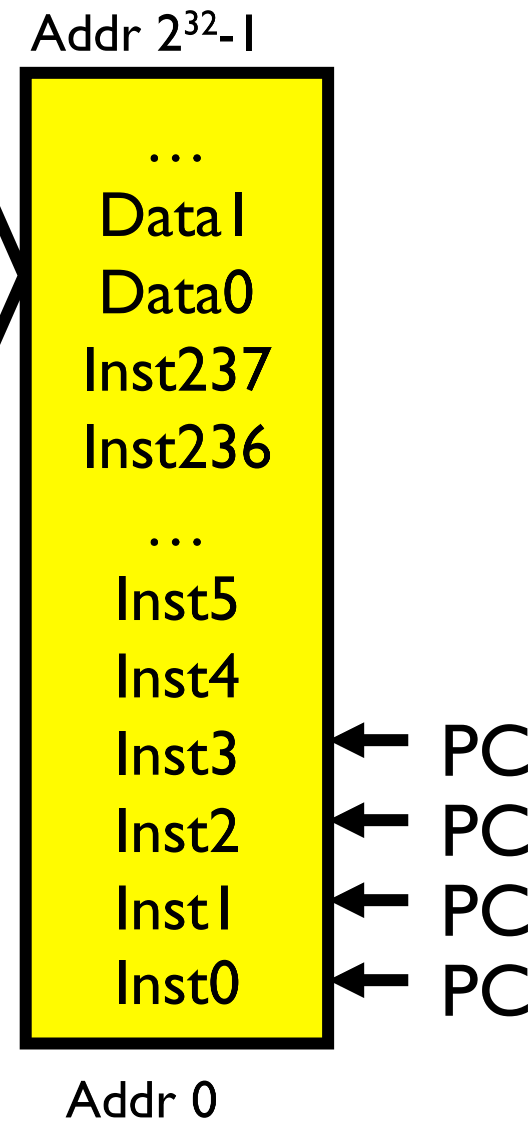
Instruction Fetch/Decode/Execute Cycle

The instruction cycle



What happens during program execution?





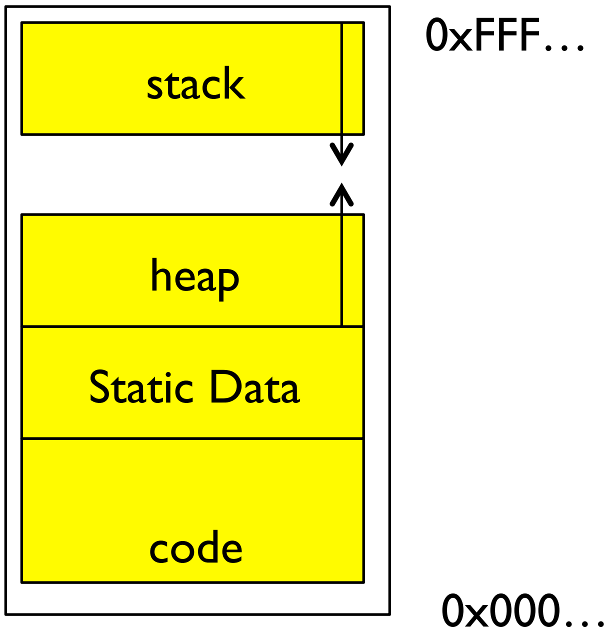
- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

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"

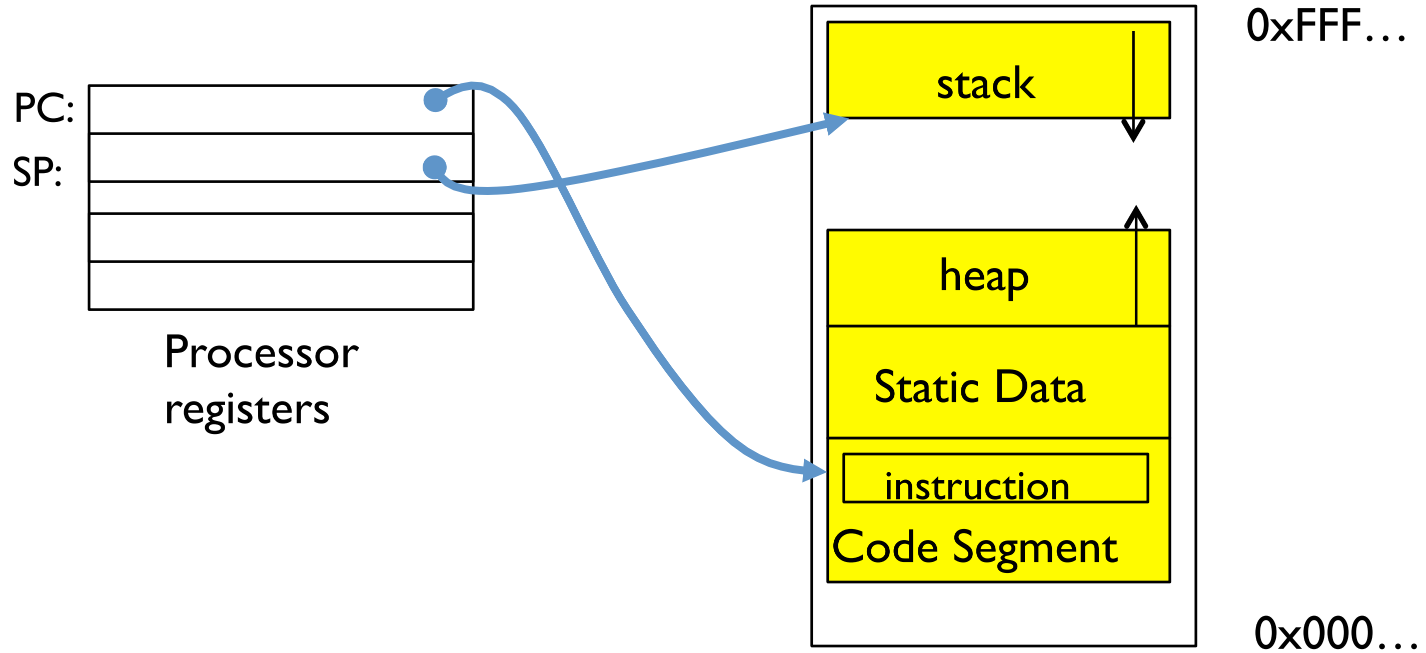
Second OS Concept: Program's Address Space

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



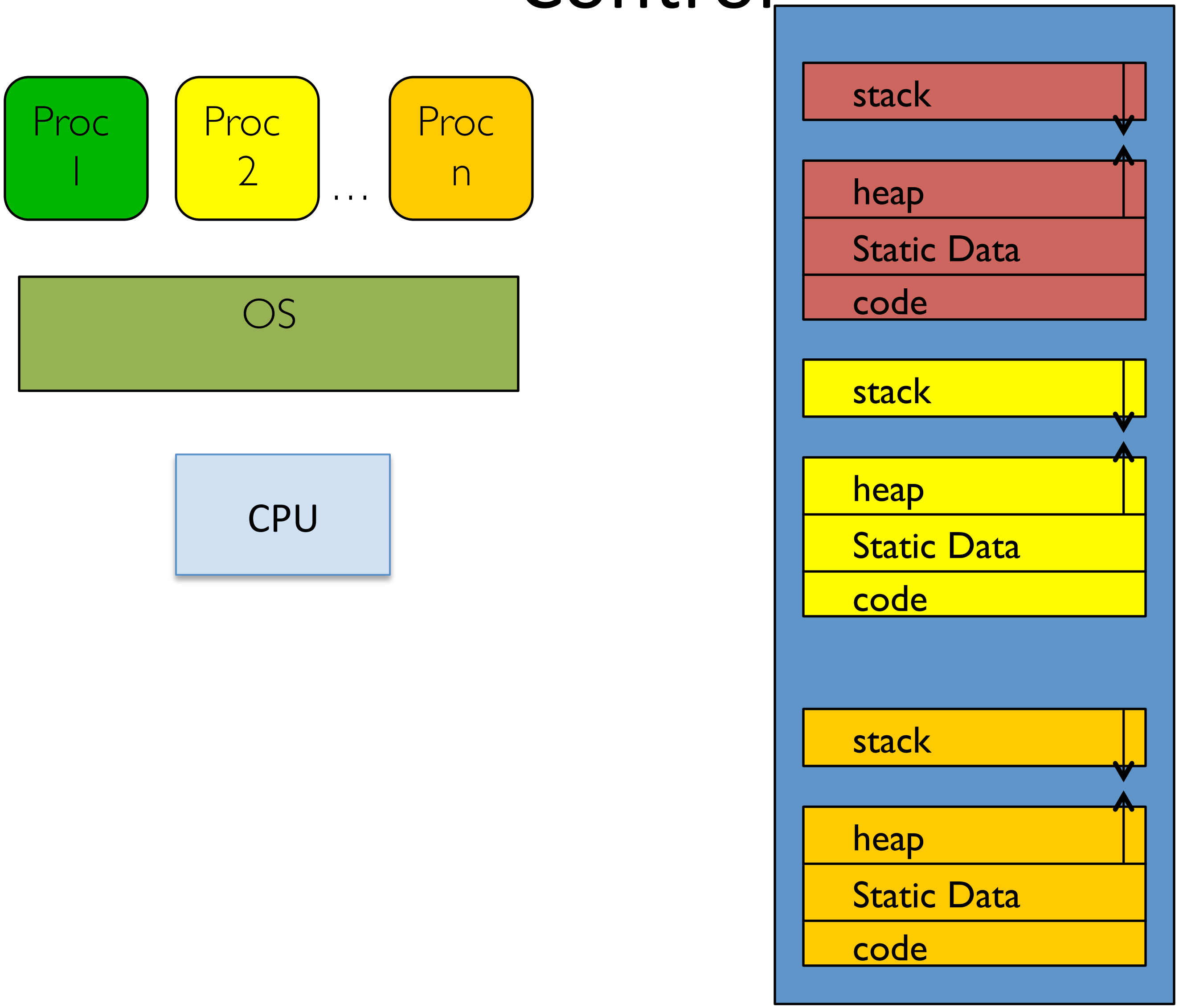
- What happens when you read or write to an address?
  - Perhaps nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - (Memory-mapped I/O)
  - Perhaps causes exception (fault)

Address Space: In a Picture

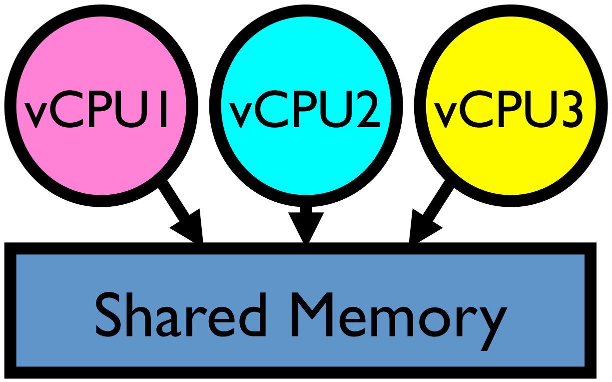


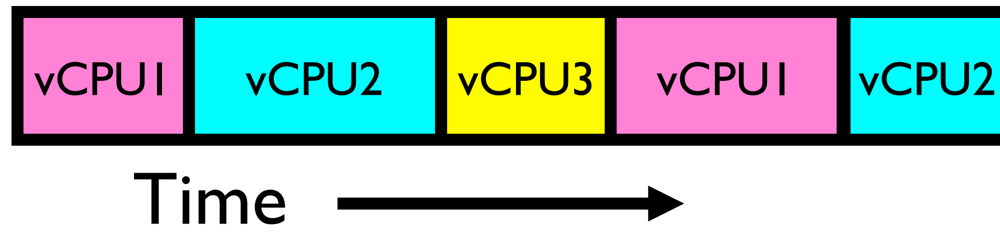
- What's in the code segment? Static data segment?
- What's in the Stack Segment?
  - How is it allocated? How big is it?
- What's in the Heap Segment?
  - How is it allocated? How big?

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)

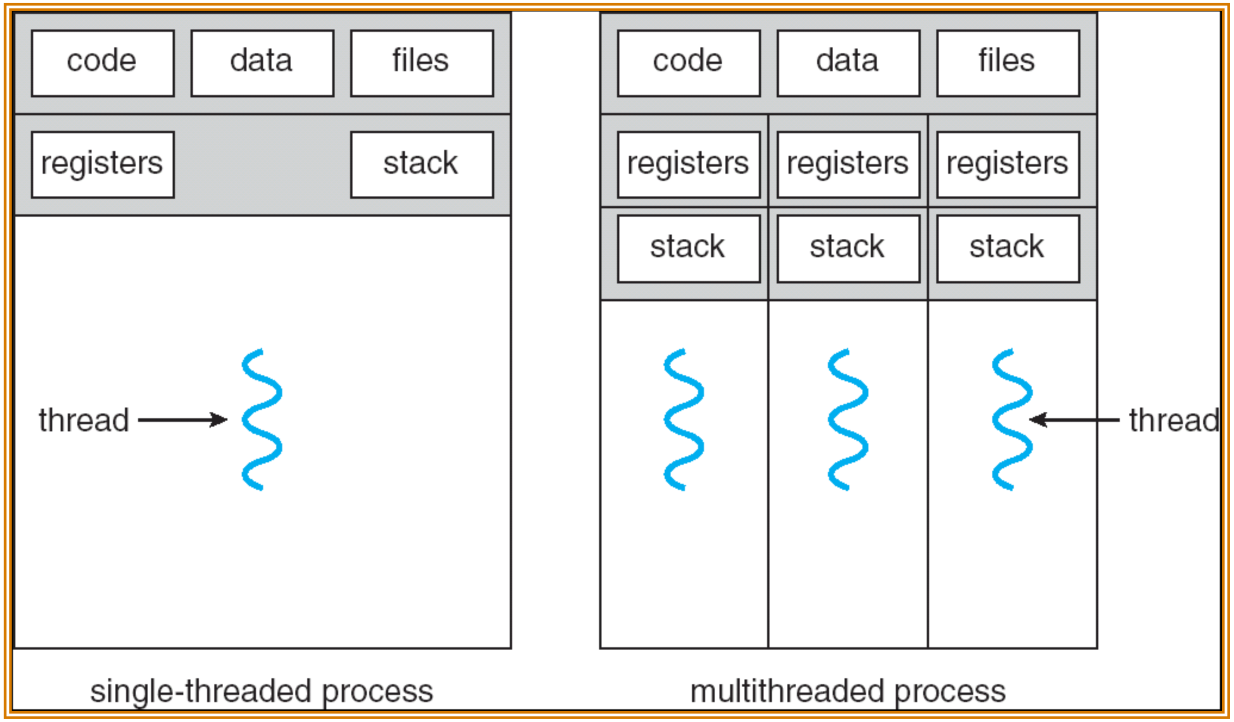
Protection

- 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

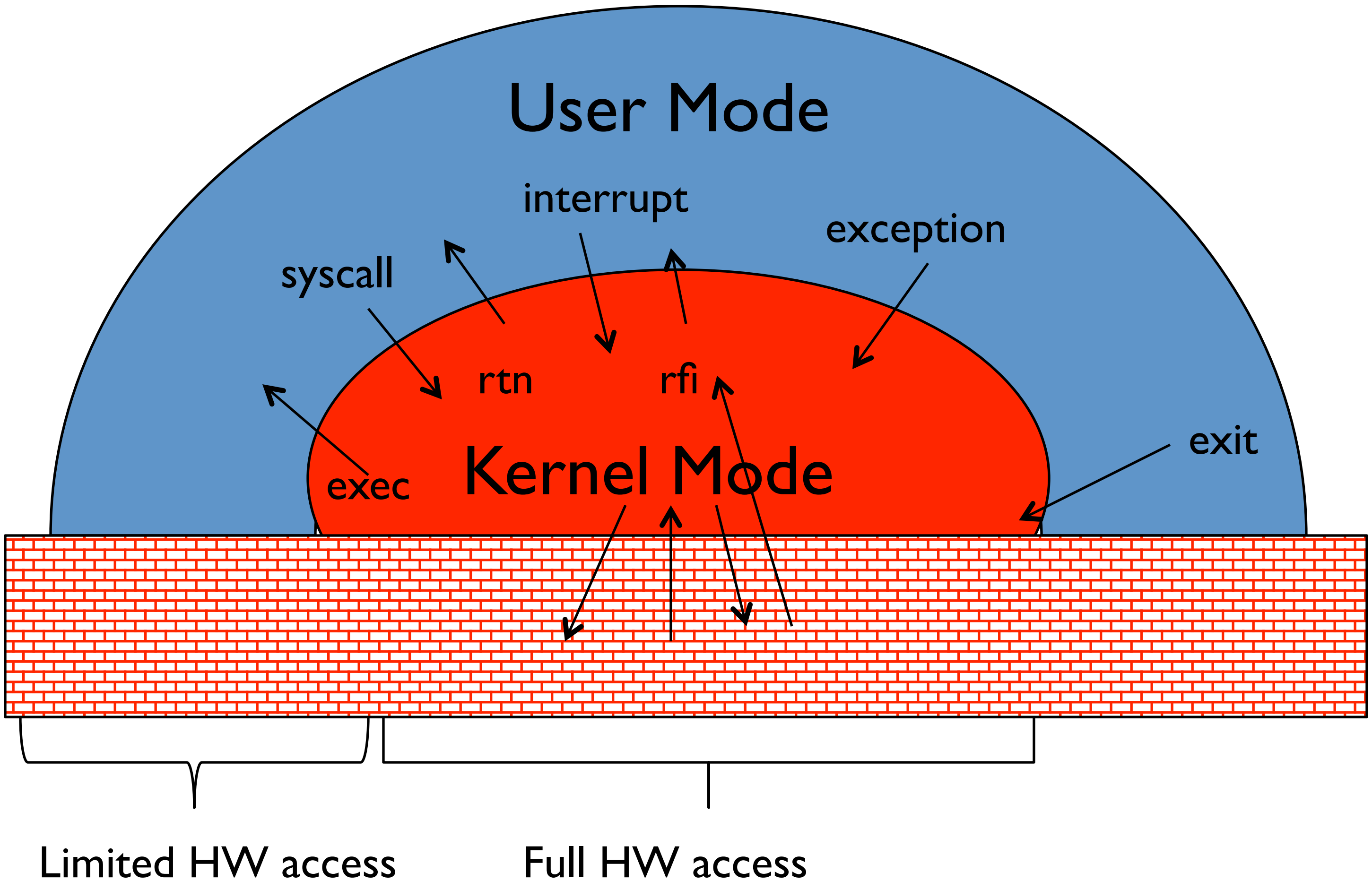


- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?
  - E.g., web server

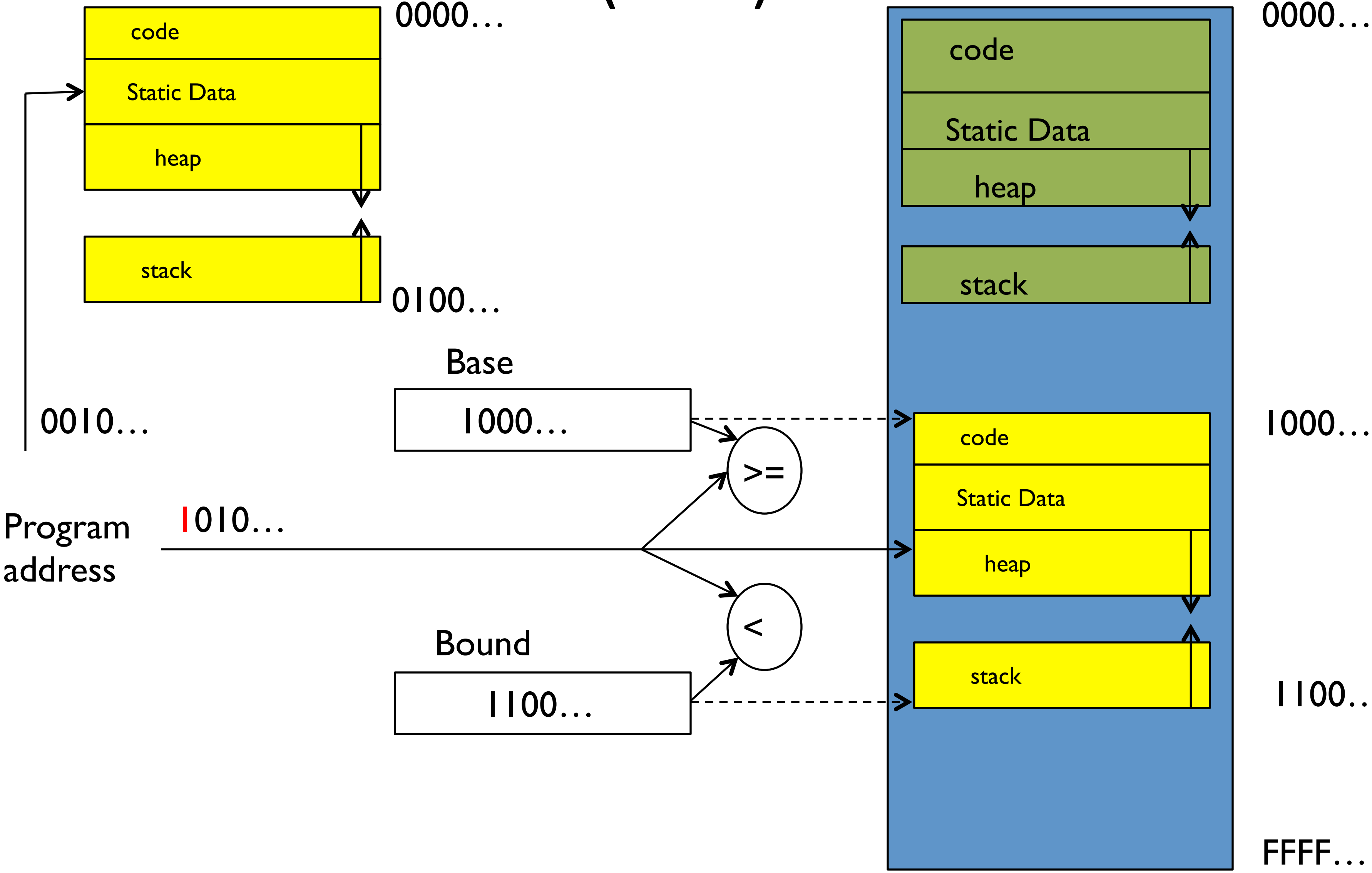
Fourth OS Concept: Dual Mode Operation

- 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

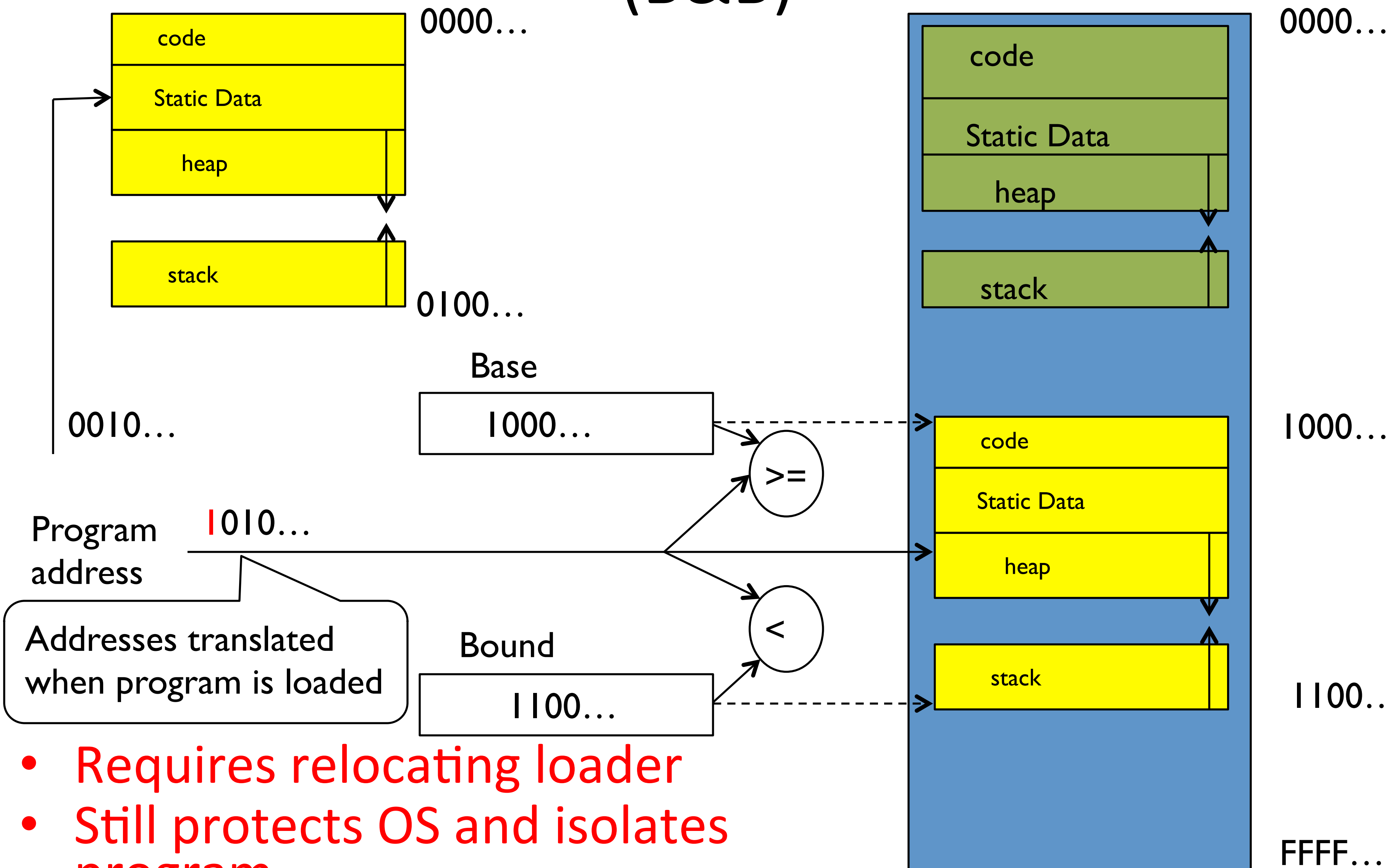
User/Kernel (Privileged) Mode



Simple Protection: Base and Bound (B&B)



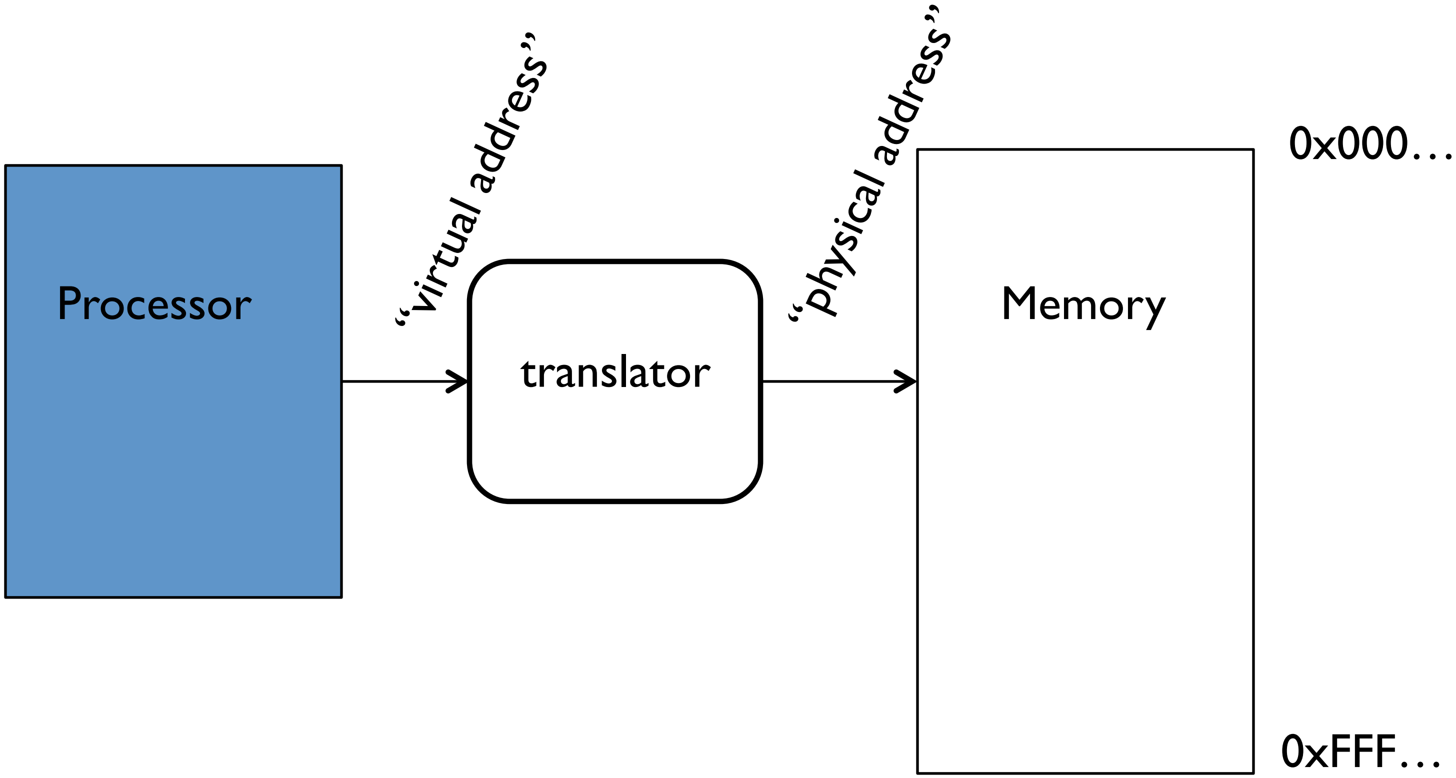
Simple Protection: Base and Bound (B&B)



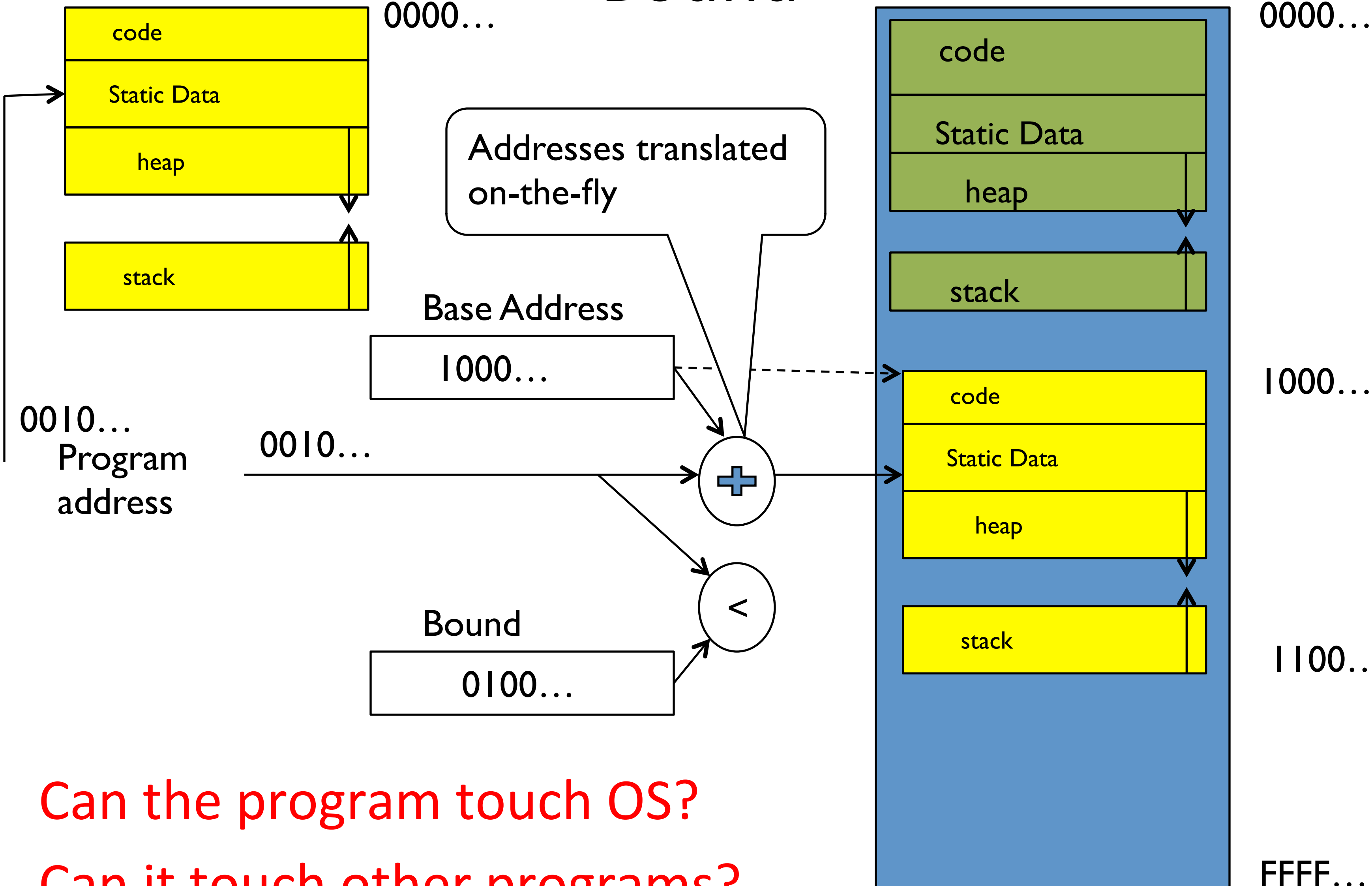
- Requires relocating loader
- Still protects OS and isolates program
- No addition on address path

Another idea: Address Space Translation

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

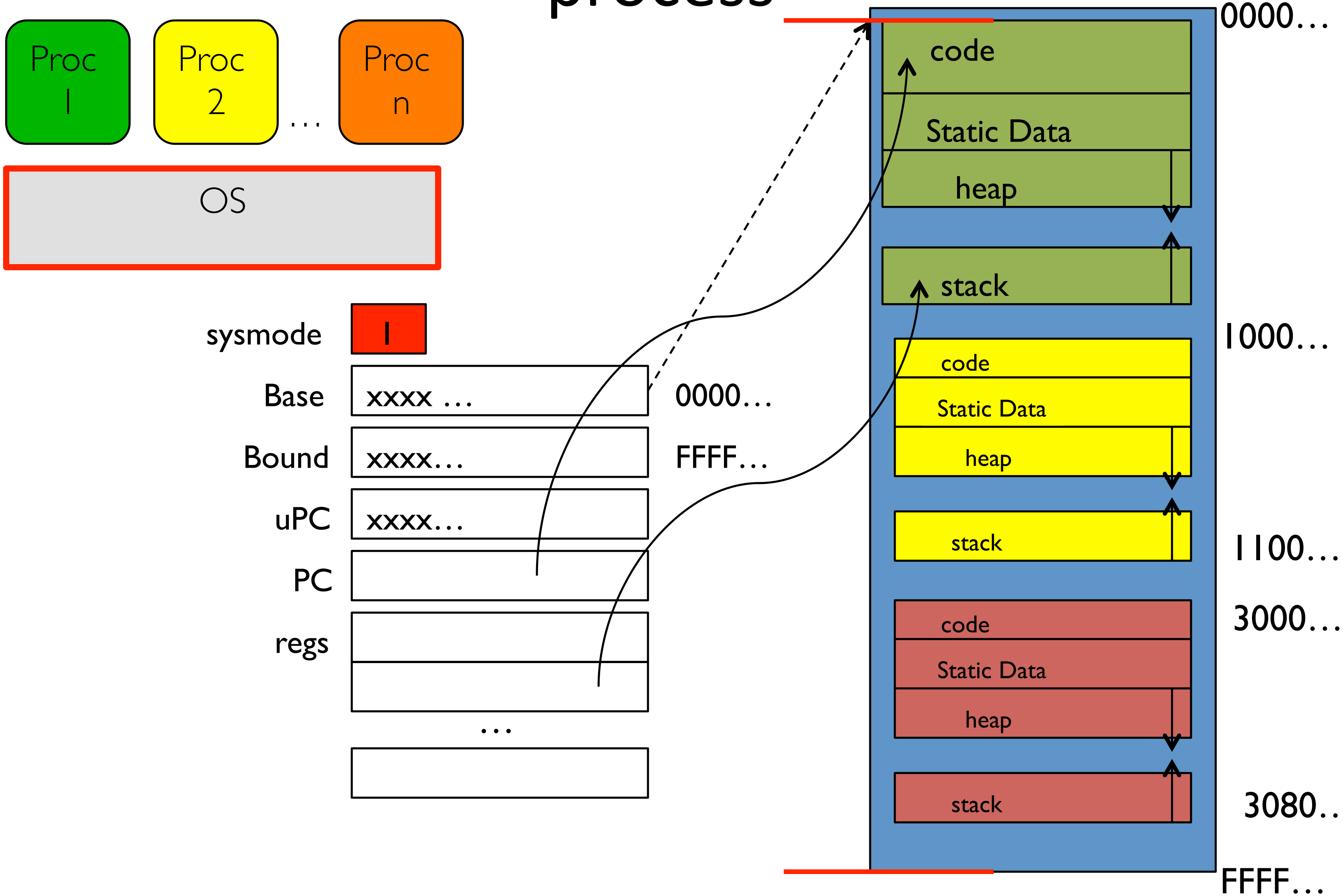


A simple address translation with Base and Bound

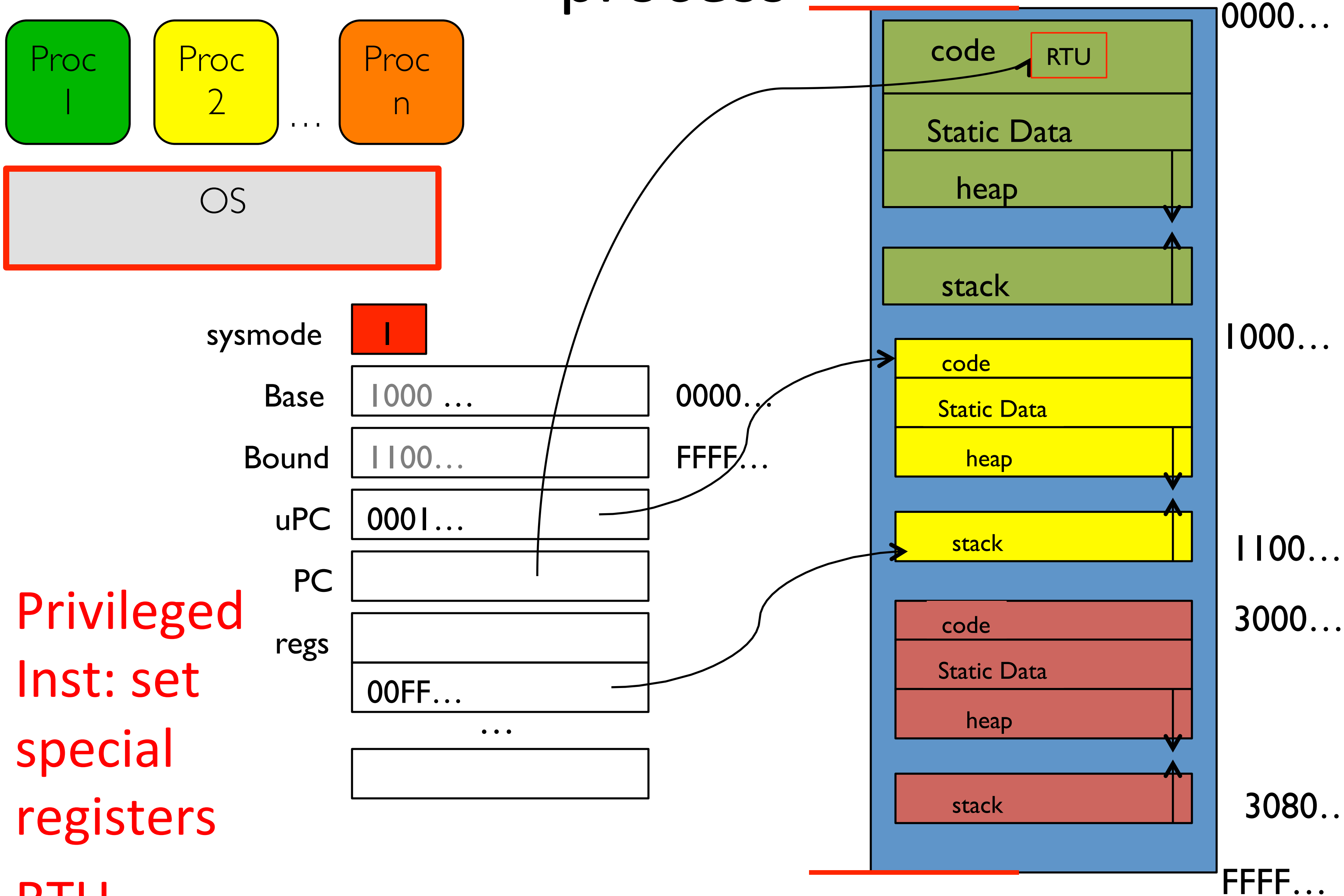


- Can the program touch OS?
- Can it touch other programs?

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

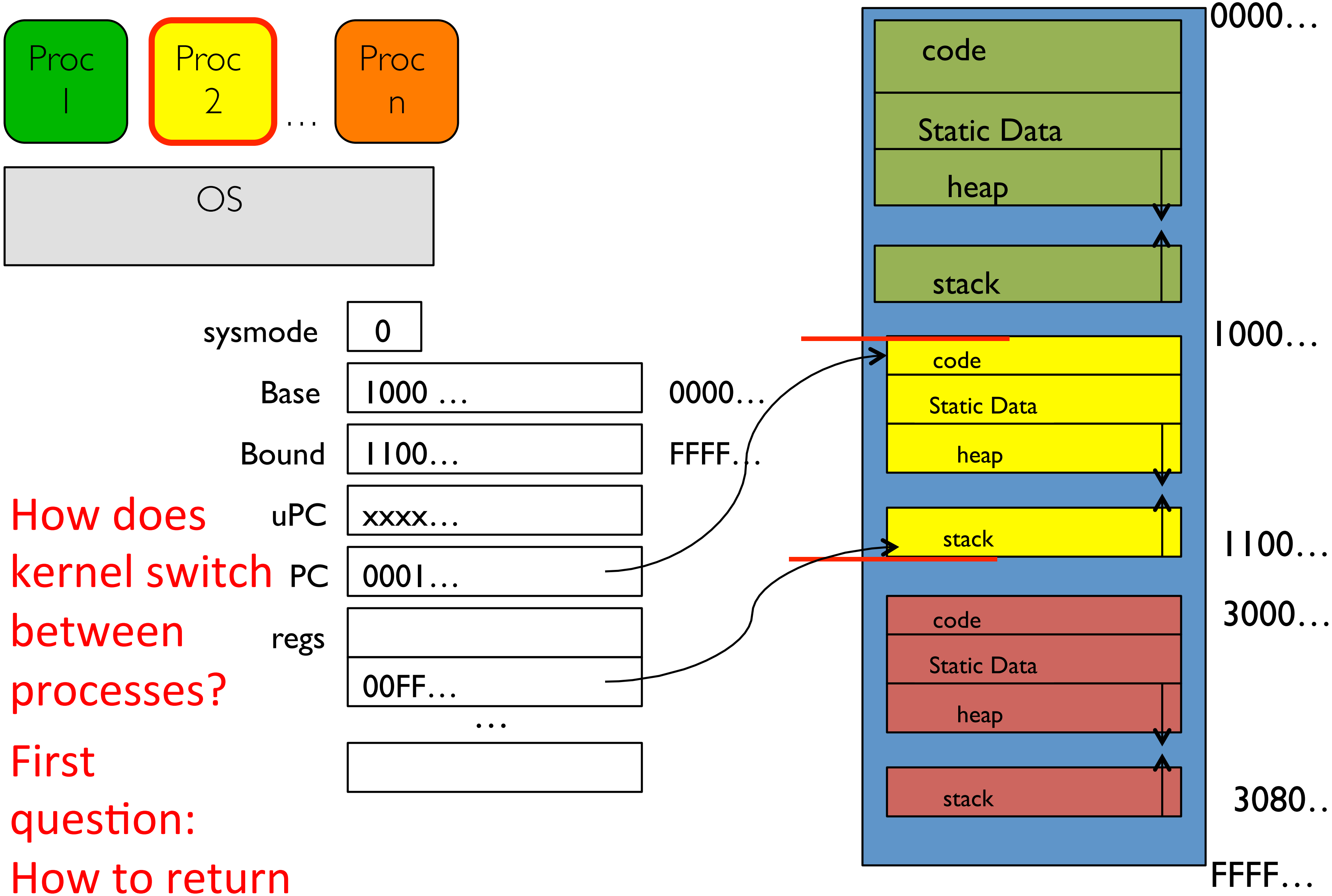


Simple B&B: OS gets ready to execute process



- Privileged Inst: set special registers
- RTU

Simple B&B: User Code Running



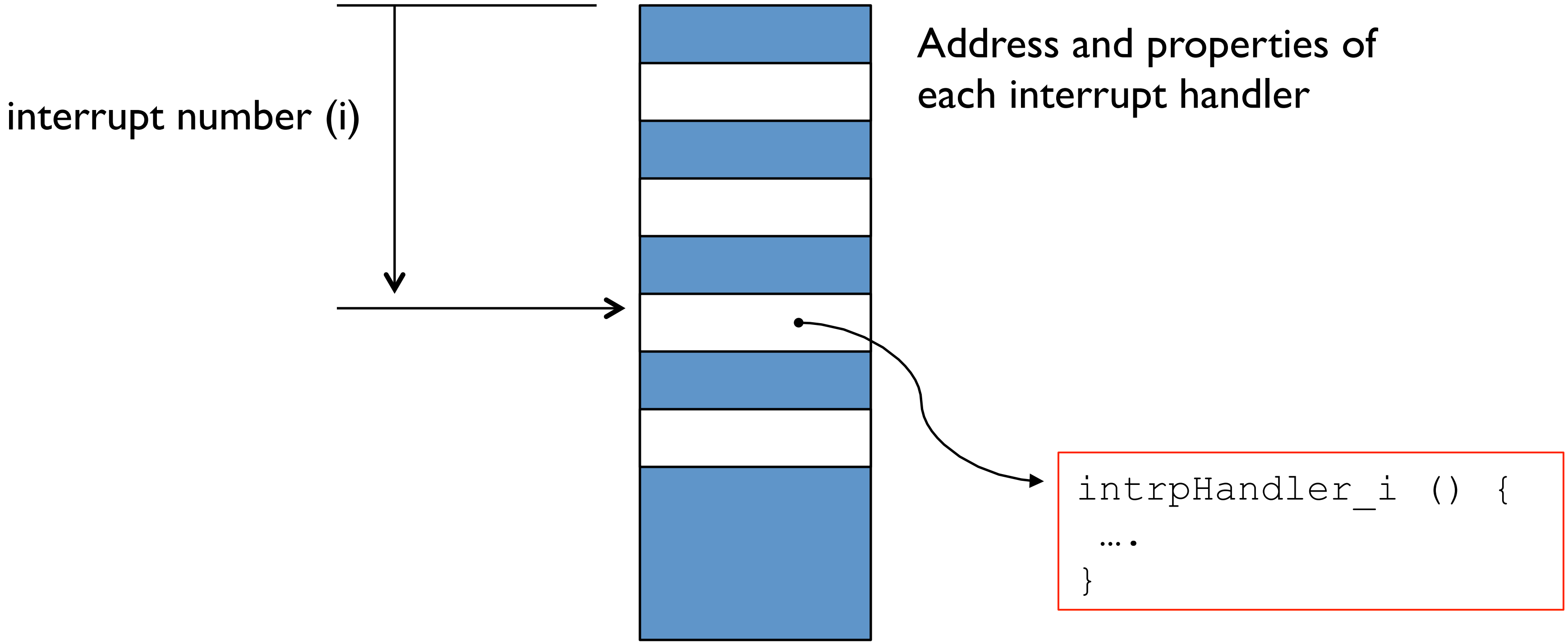
- How does kernel switch between processes?
- First question: How to return to system?

3 types of Mode Transfer

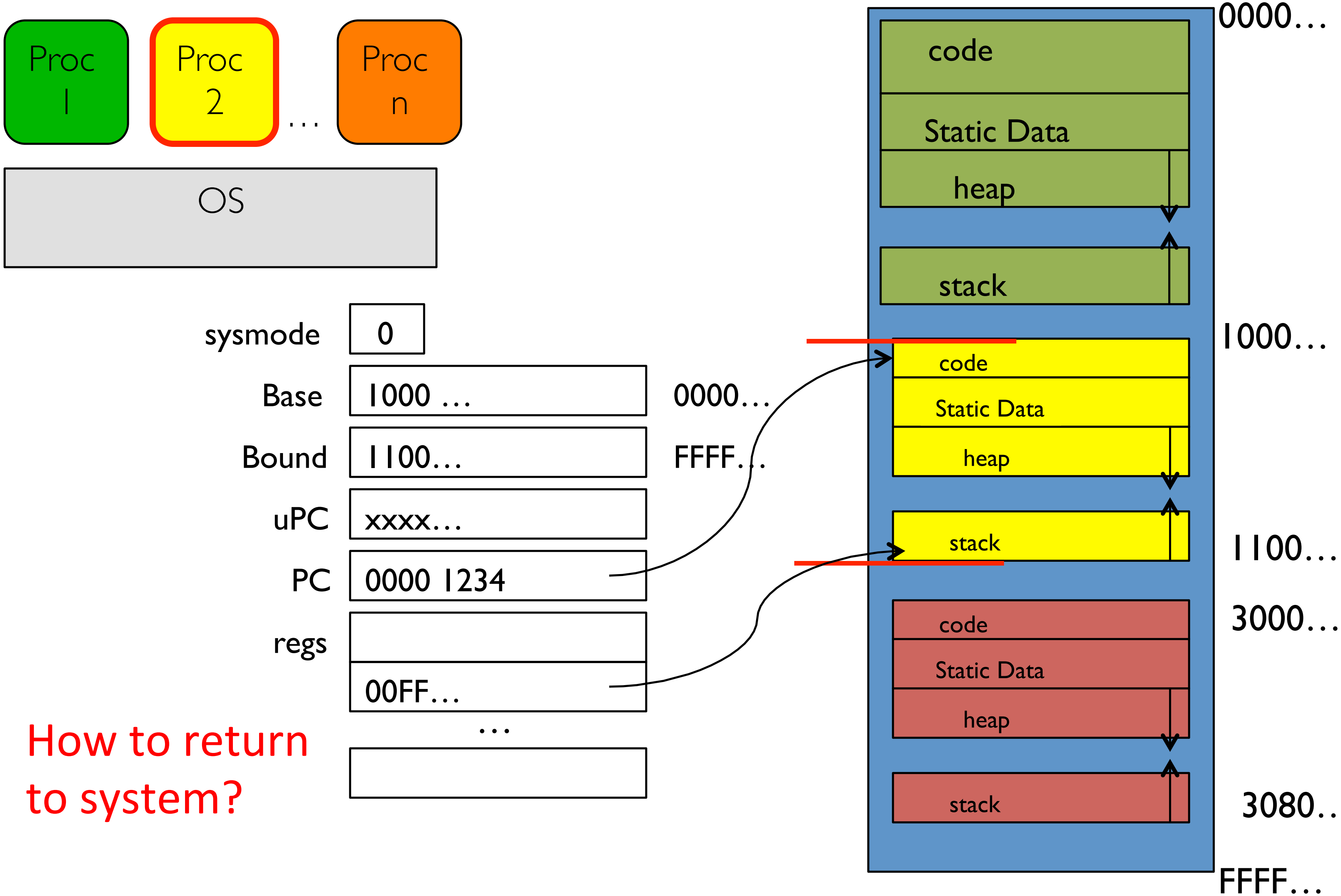
- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but “outside” the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt
  - 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

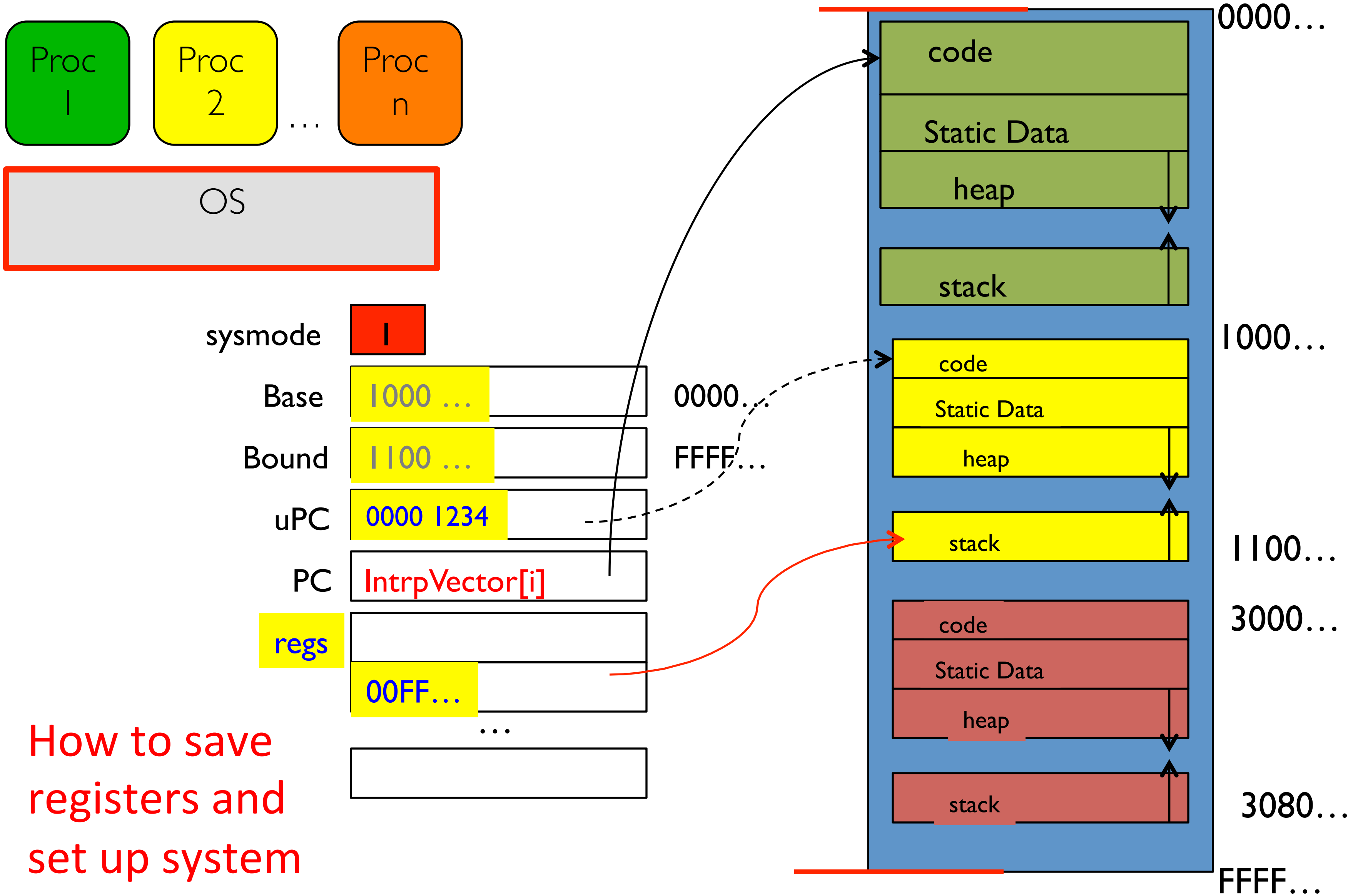


Simple B&B: User => Kernel



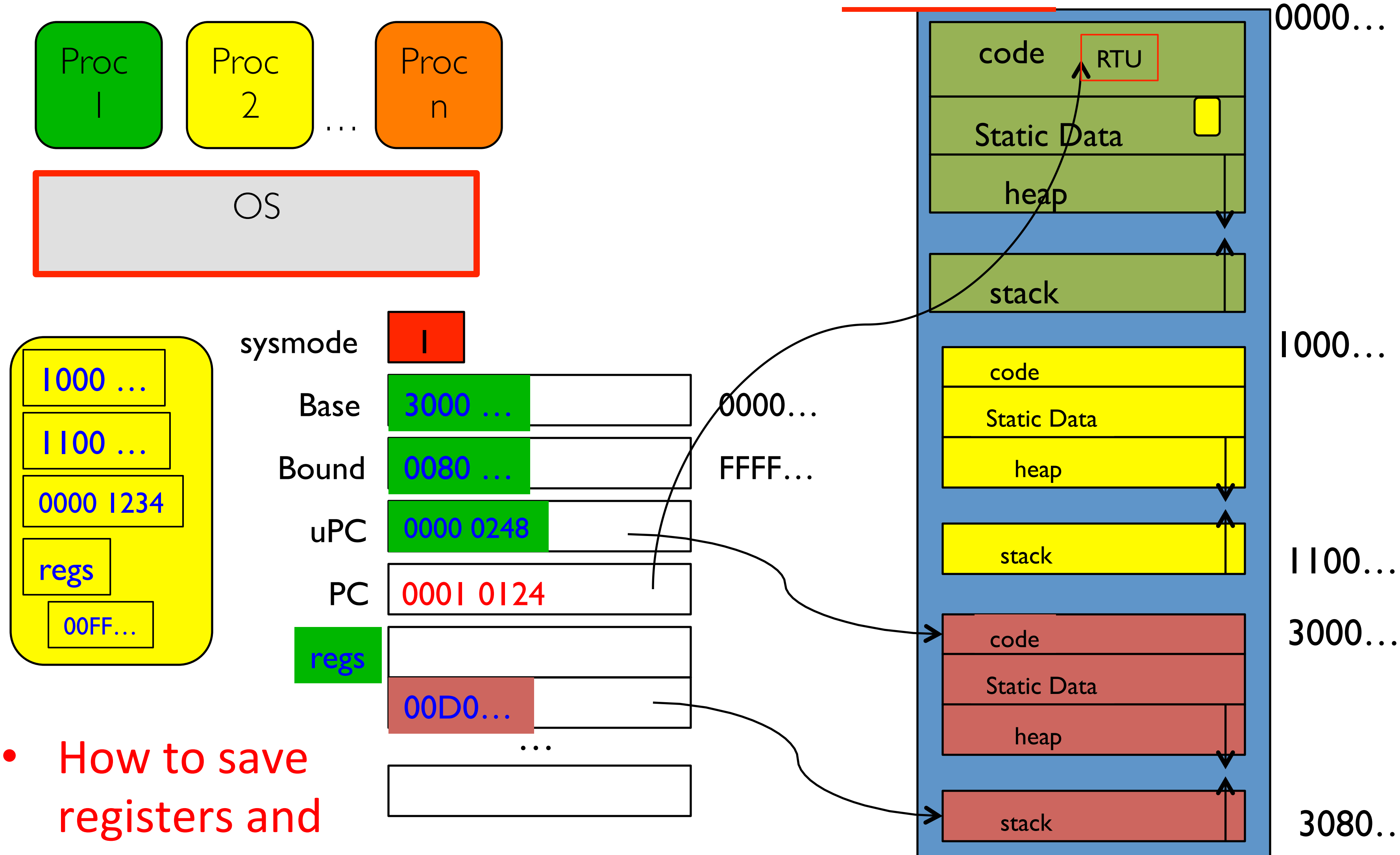
- How to return to system?

Simple B&B: Interrupt



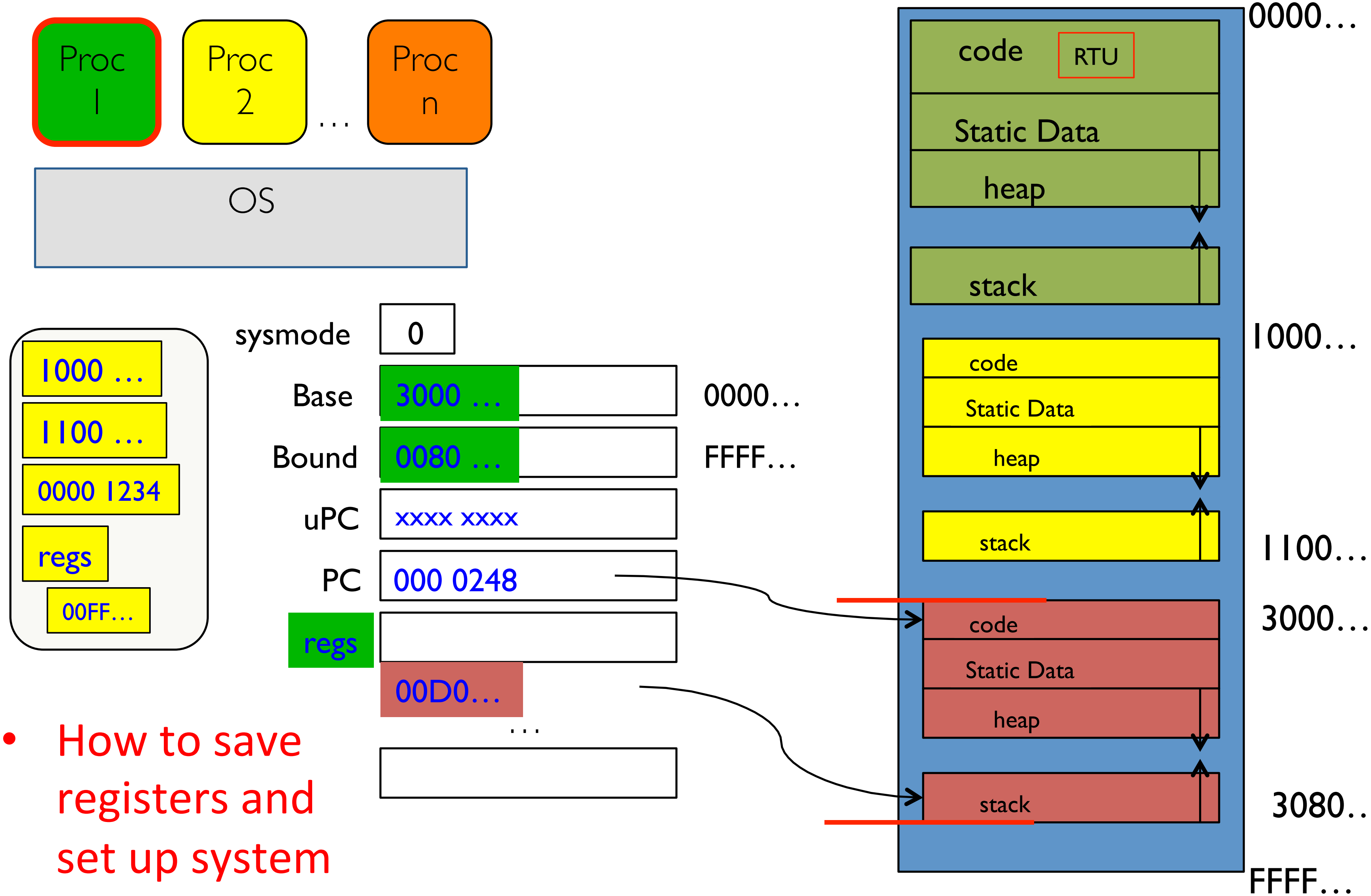
- How to save registers and set up system stack?

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