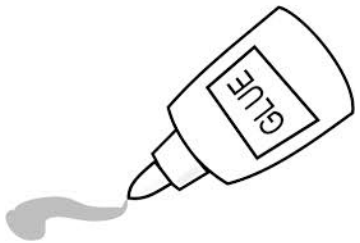
CS162
Operating Systems and Systems Programming
Lecture 2

Four Fundamental OS Concepts

Recall: What is an Operating System?

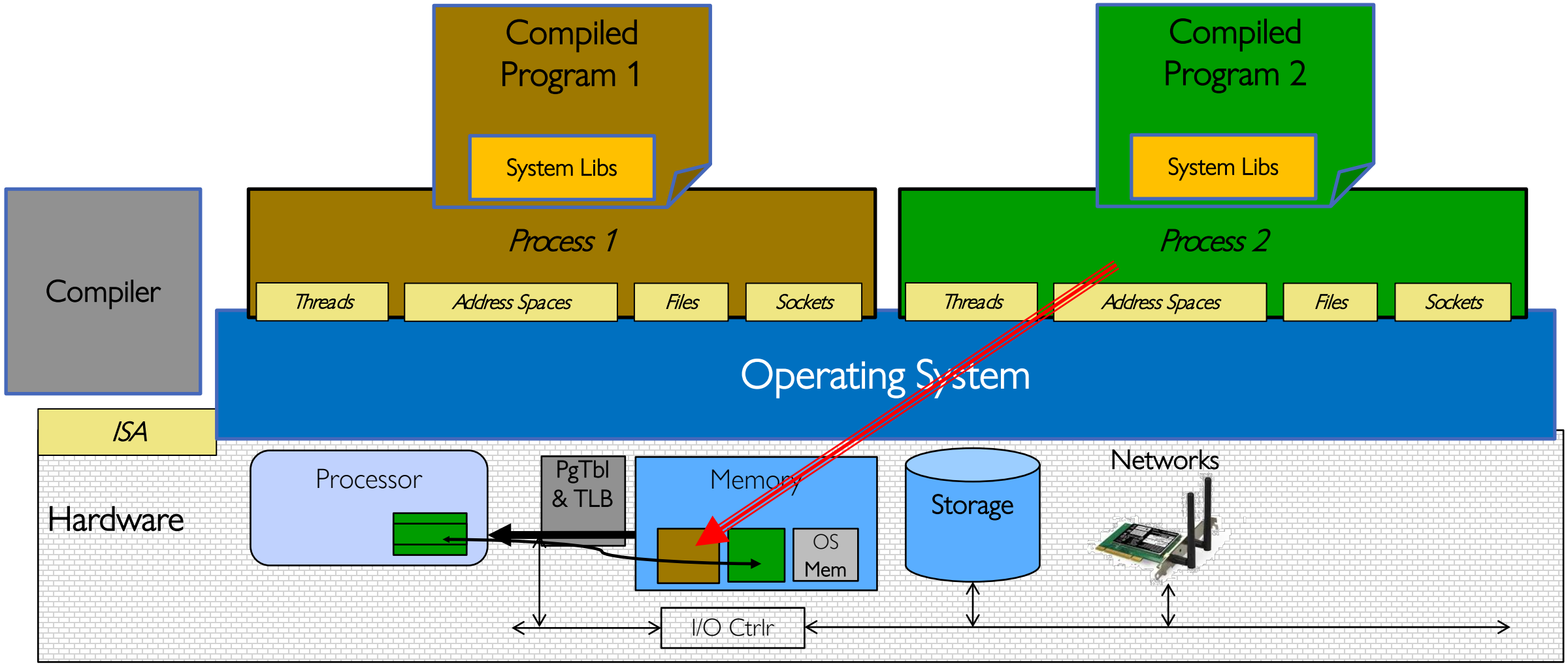
- Referee
  - Manage protection, isolation, and sharing of resources
    - » Resource allocation and communication

- Illusionist
  - Provide clean, easy-to-use abstractions of physical resources
    - » Infinite memory, dedicated machine
    - » Higher level objects: files, users, messages
    - » Masking limitations, virtualization

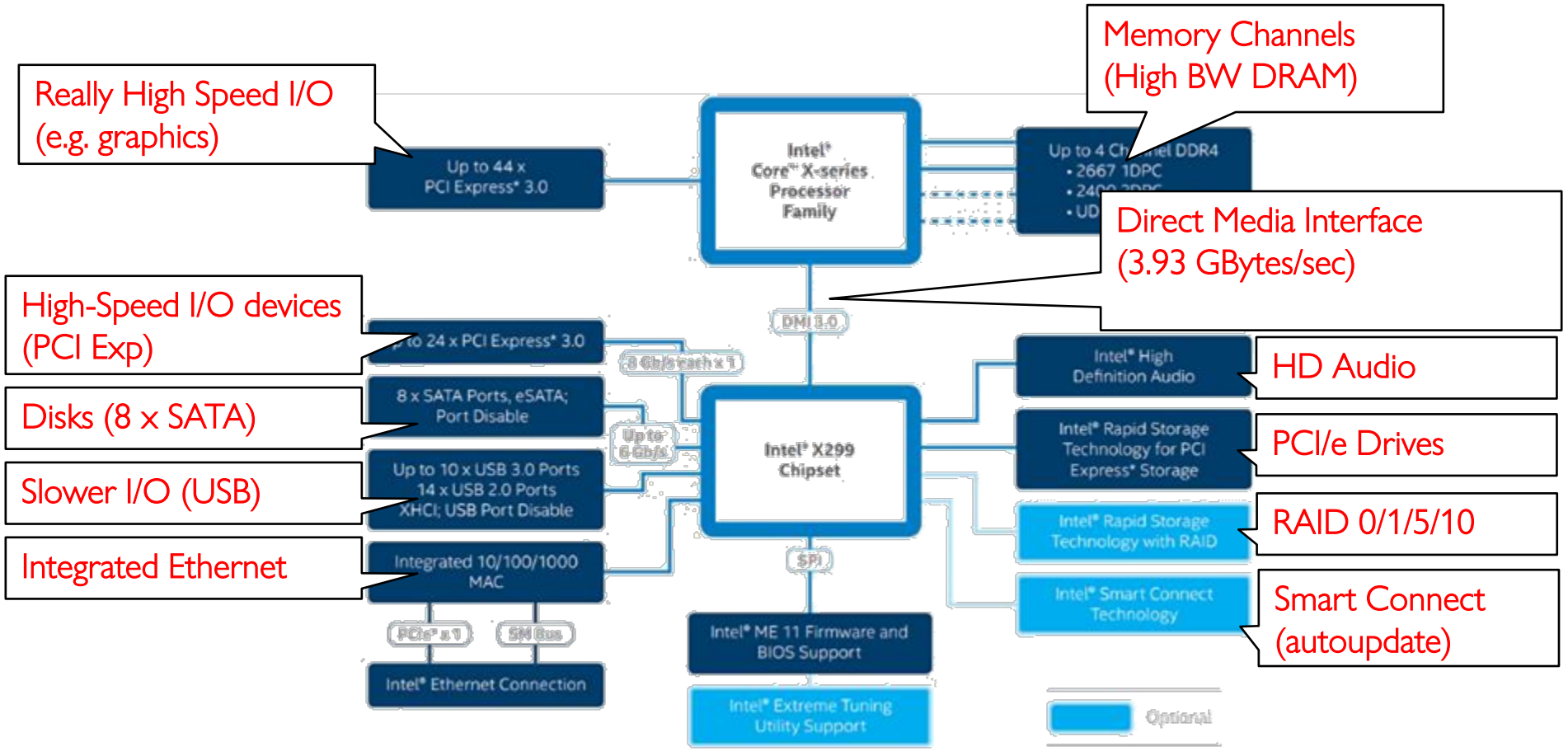


- Glue
  - Common services
    - » Storage, Window system, Networking
    - » Sharing, Authorization
    - » Look and feel

Recall: OS Protection

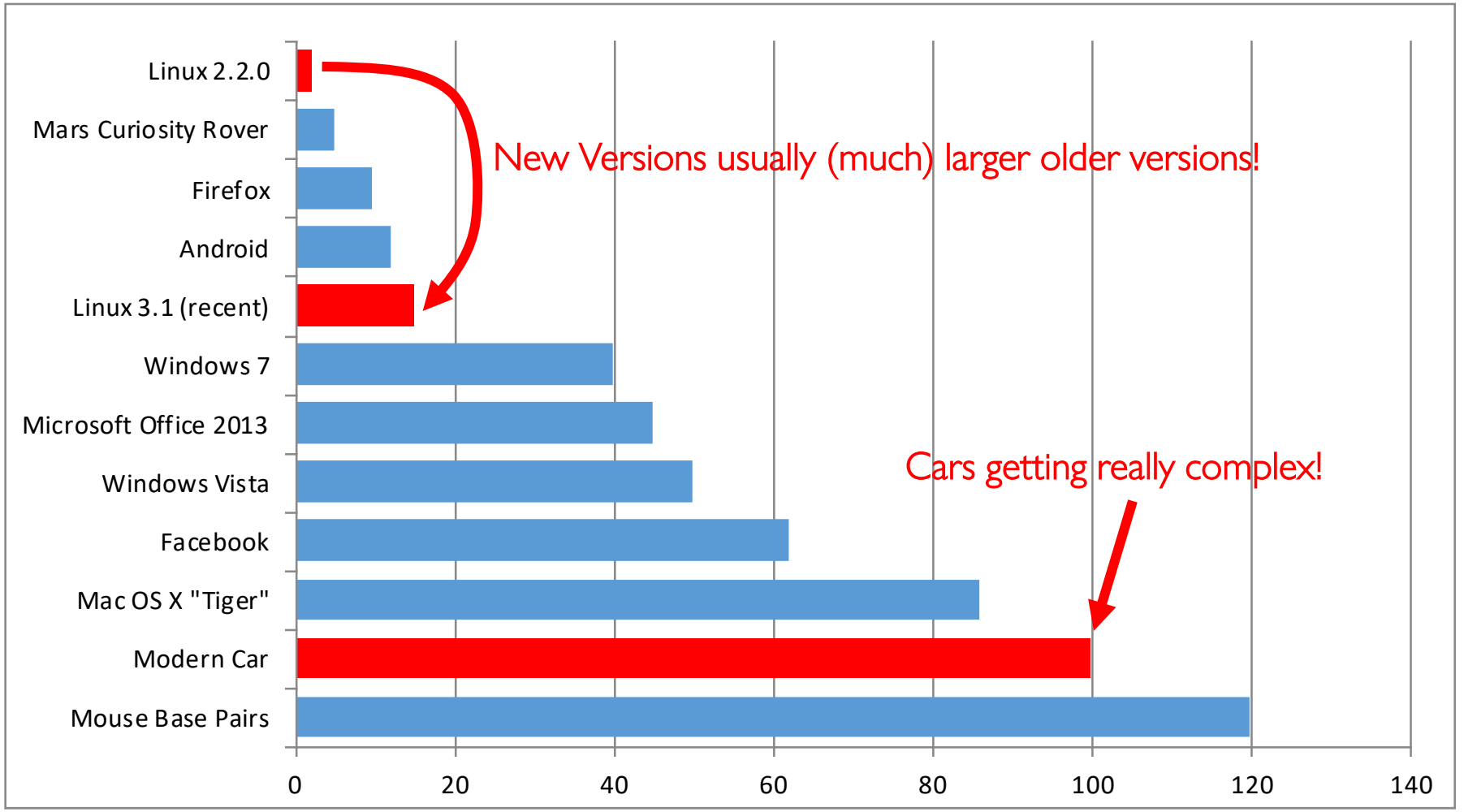


Recall: HW Functionality ⇒ great complexity!



Intel Skylake-X I/O Configuration

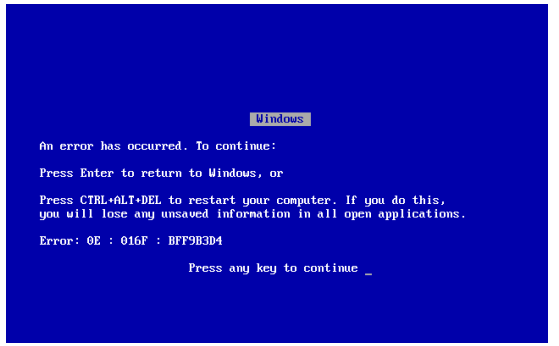
Recall: Increasing Software Complexity



(source https://informationisbeautiful.net/visualizations/million-lines-of-code/)

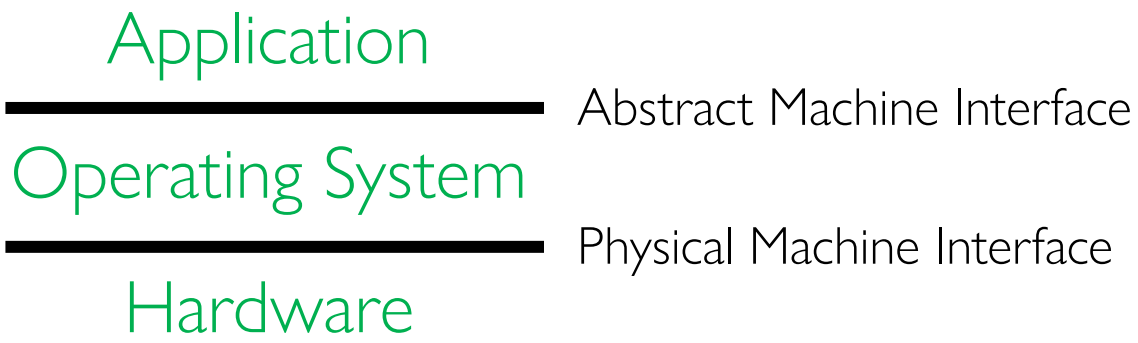
Complexity leaks into OS if not properly designed:

- Buggy device drivers
- Holes in security model or bugs in OS lead to instability and privacy breaches
  - Meltdown (2017)
  - Spectre (2017)
- Version skew of libraries can lead to problems with application execution



OS *Abstracts* Underlying Hardware to help Tame Complexity

- Processor → Thread
- Memory → Address Space
- Disks, SSDs, … → Files
- Networks → Sockets
- Machines → Processes



- OS as an *Illusionist*:
  - Remove software/hardware quirks (*fight complexity)*
  - Optimize for convenience, utilization, reliability, … *(help the programmer)*

- For any OS area (e.g. file systems, virtual memory, networking, scheduling):
  - What hardware interface to handle? (physical reality)
  - What's software interface to provide? (nicer abstraction)

Today: Four Fundamental OS Concepts

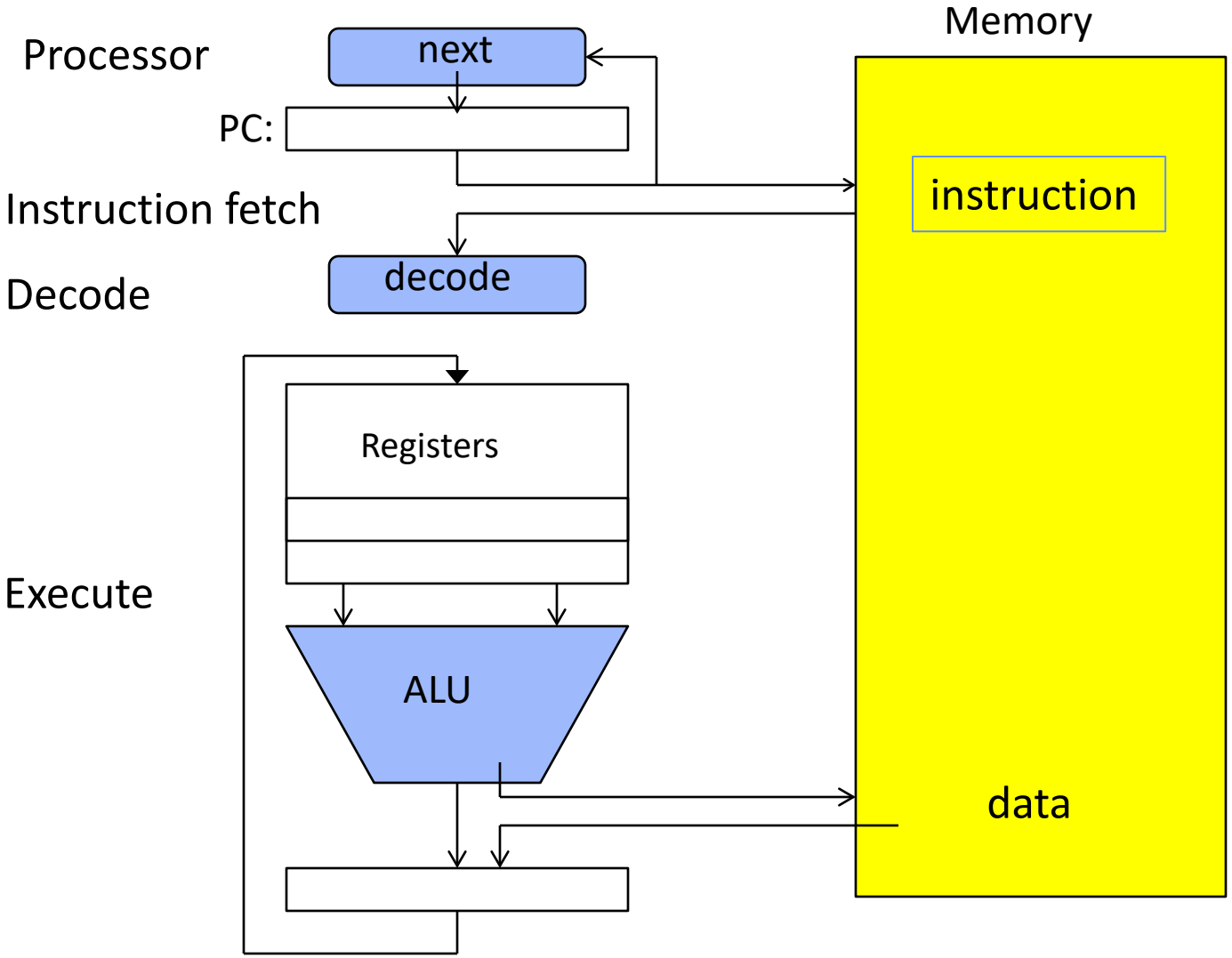
- Thread: Execution Context
  - Fully describes program state
- Address space
  - Set of memory addresses accessible to program (for read or write)
- Process: an instance of a running program
  - Protected Address Space + One or more Threads
- Dual mode operation / Protection
  - Only the "system" has the ability to access certain resources

First OS Concept: Thread of Control

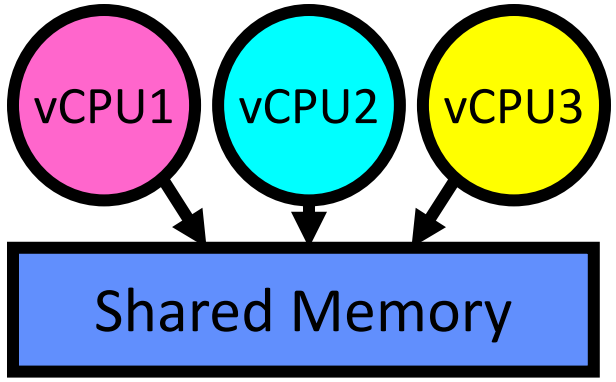
- Thread: Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack, Memory State
- A thread is *executing* on a processor (core) when it is *resident* in the processor registers
- A thread is *suspended* (not *executing)* when its state *is not* loaded (resident) into the processor
  - Processor state pointing at some other thread
  - Program counter register *is not* pointing at next instruction from this thread
  - Often: a copy of the last value for each register stored in memory

61 is back! Instruction Fetch/Decode/Execute

The instruction cycle

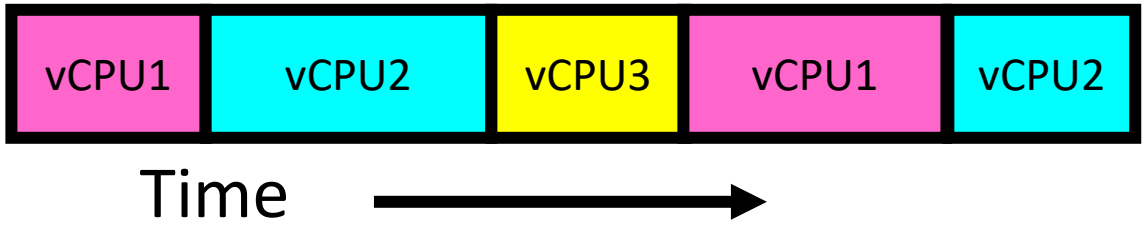


Illusion of Multiple Processors



Programmer's View

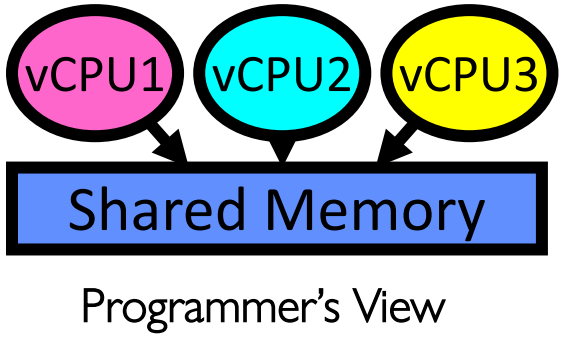
- Multiplex in time
- Threads are *virtual cores*



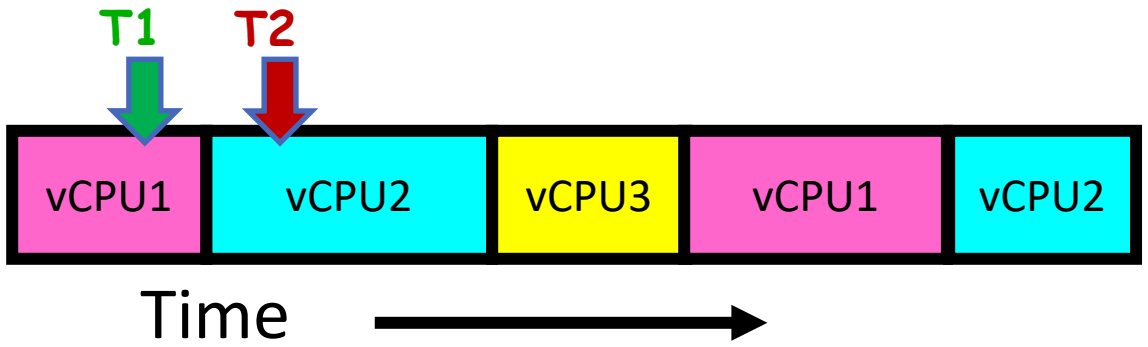
- Contents of virtual core (thread):
  - Program counter, stack pointer
  - Registers
- Where is "it" (the thread)?
  - On the real (physical) core, or
  - Saved in chunk of memory – called the *Thread Control Block (TCB)*

Illusion of Multiple Processors (Continued)

- Consider:
  - At T1: vCPU1 on real core, vCPU2 in memory
  - At T2: vCPU2 on real core, vCPU1 in memory

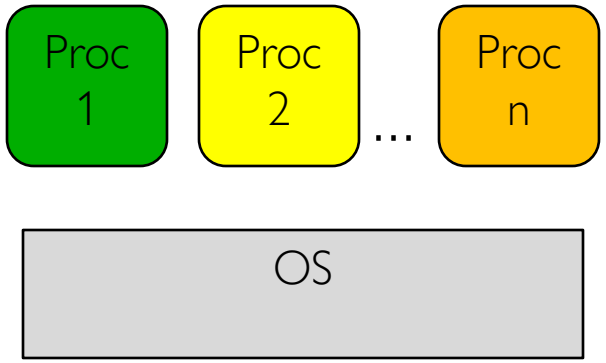


Programmer's View

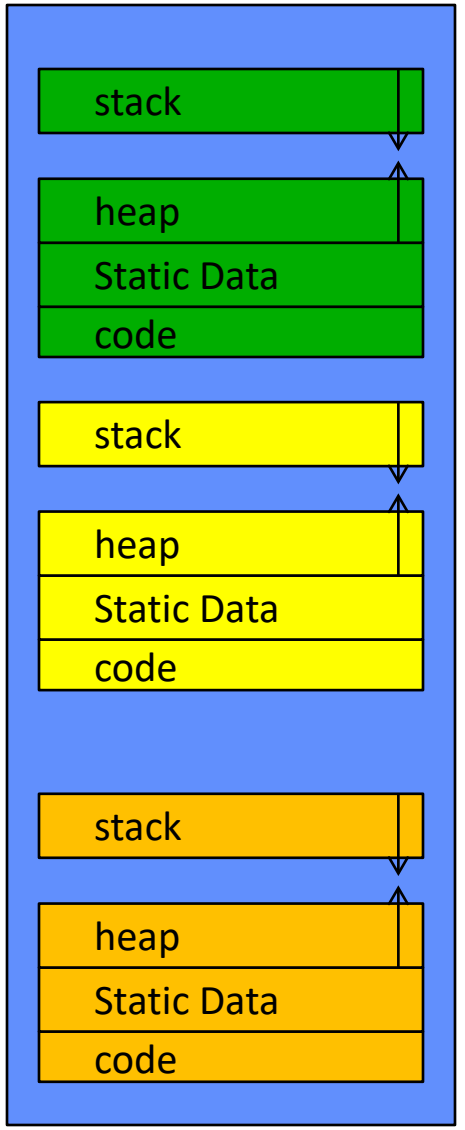


- What happened?
  - OS Ran, triggering a *context switch*
  - Saved PC, SP, … in vCPU1's thread control block (memory)
  - Loaded PC, SP, … from vCPU2's TCB, jumped to PC

Multiprogramming - Multiple Threads of Control



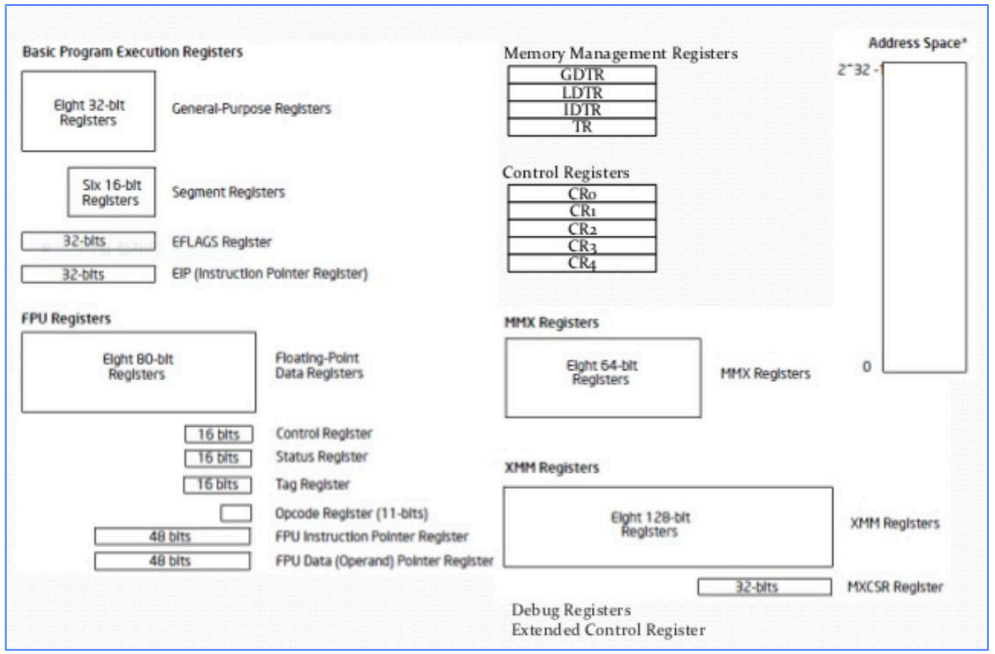
- Thread Control Block (TCB)
  - Holds contents of registers when thread not running
- Where are TCBs stored?
  - For now, in the kernel
- PINTOS? – read thread.h and thread.c



Registers: RISC-V ⇒ x86

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Load/Store Arch (RISC-V) with software conventions



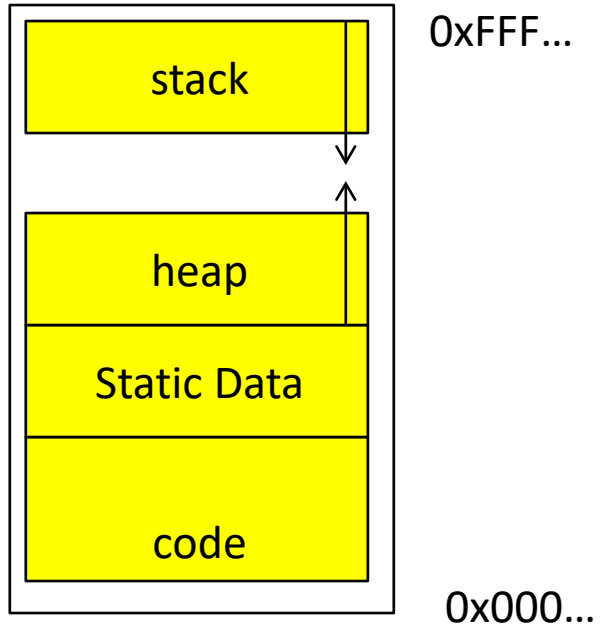
Complex mem-mem arch (x86) with specialized registers and "segments"

- cs61C does RISC-V. Will need to learn x86…
- Section will cover this architecture

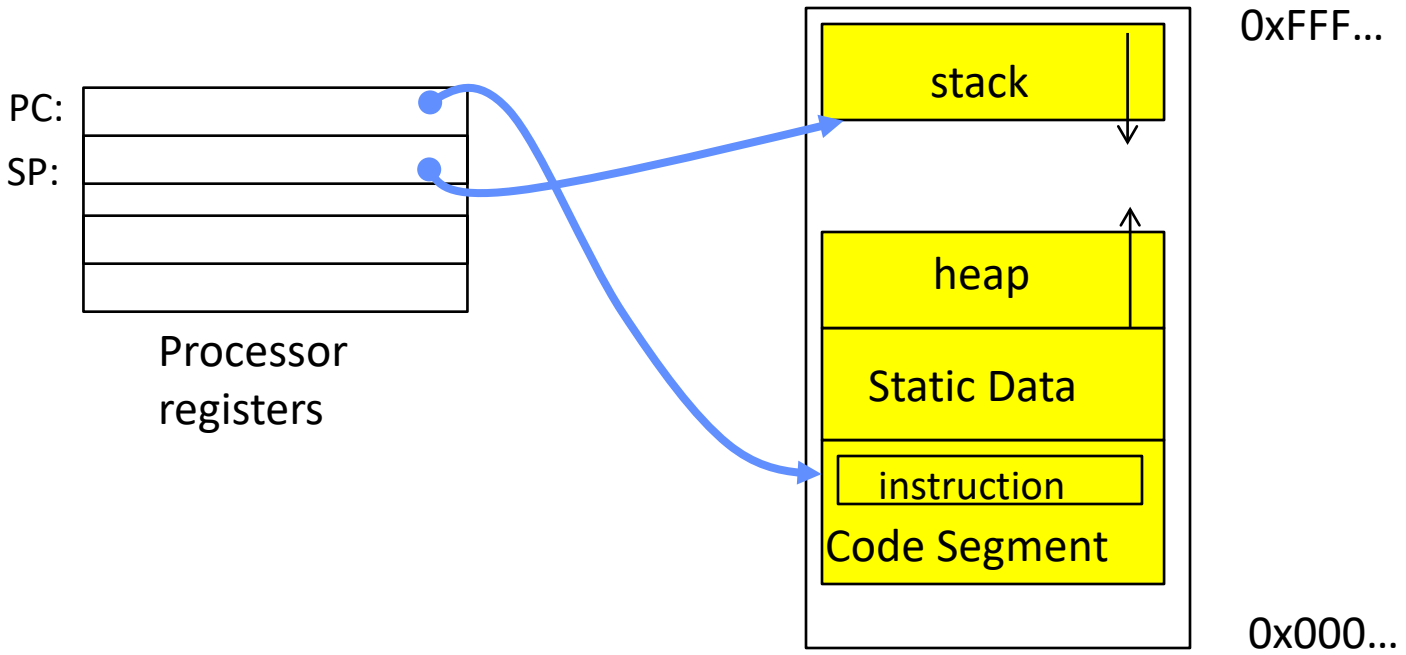
Second OS Concept: Address Space

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For 32-bit processor: 2^{32} = 4 billion (10^9) addresses
  - For 64-bit processor: 2^{64} = 18 quintillion (10^{18}) addresses

- What happens when you read or write to an address?
  - Perhaps acts like regular memory
  - Perhaps causes I/O operation
    - » (Memory-mapped I/O)
  - Perhaps causes exception (fault)
  - Communicates with another program
  - ….



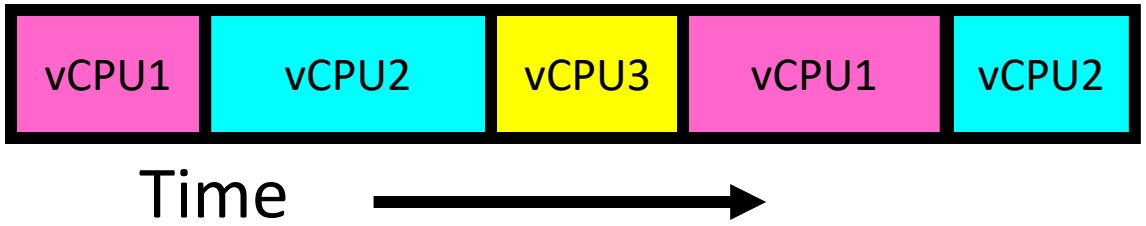
Address Space: In a Picture



Very Simple Multiprogramming

- All vCPU's share non-CPU resources
  - Memory, I/O Devices

- Each thread can read/write memory
  - Perhaps data of others, including OS!



- Used in early days of computing or embedded systems.

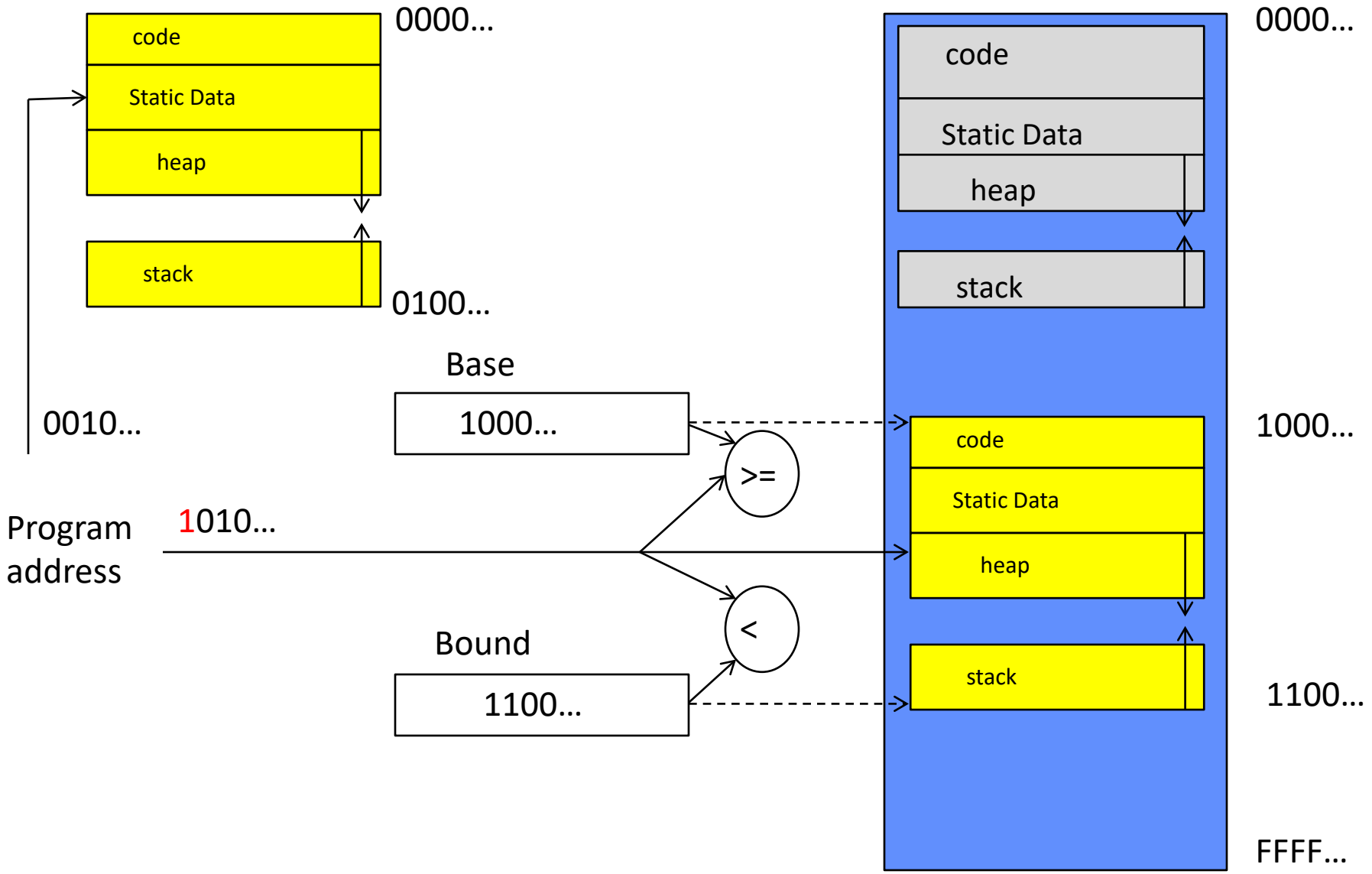
Simple Multiplexing has no Protection!

- OS must protect user programs from one another
  - Prevent threads owned by one user from impacting threads owned by another user
  - Example: prevent one user from stealing secret information from another user

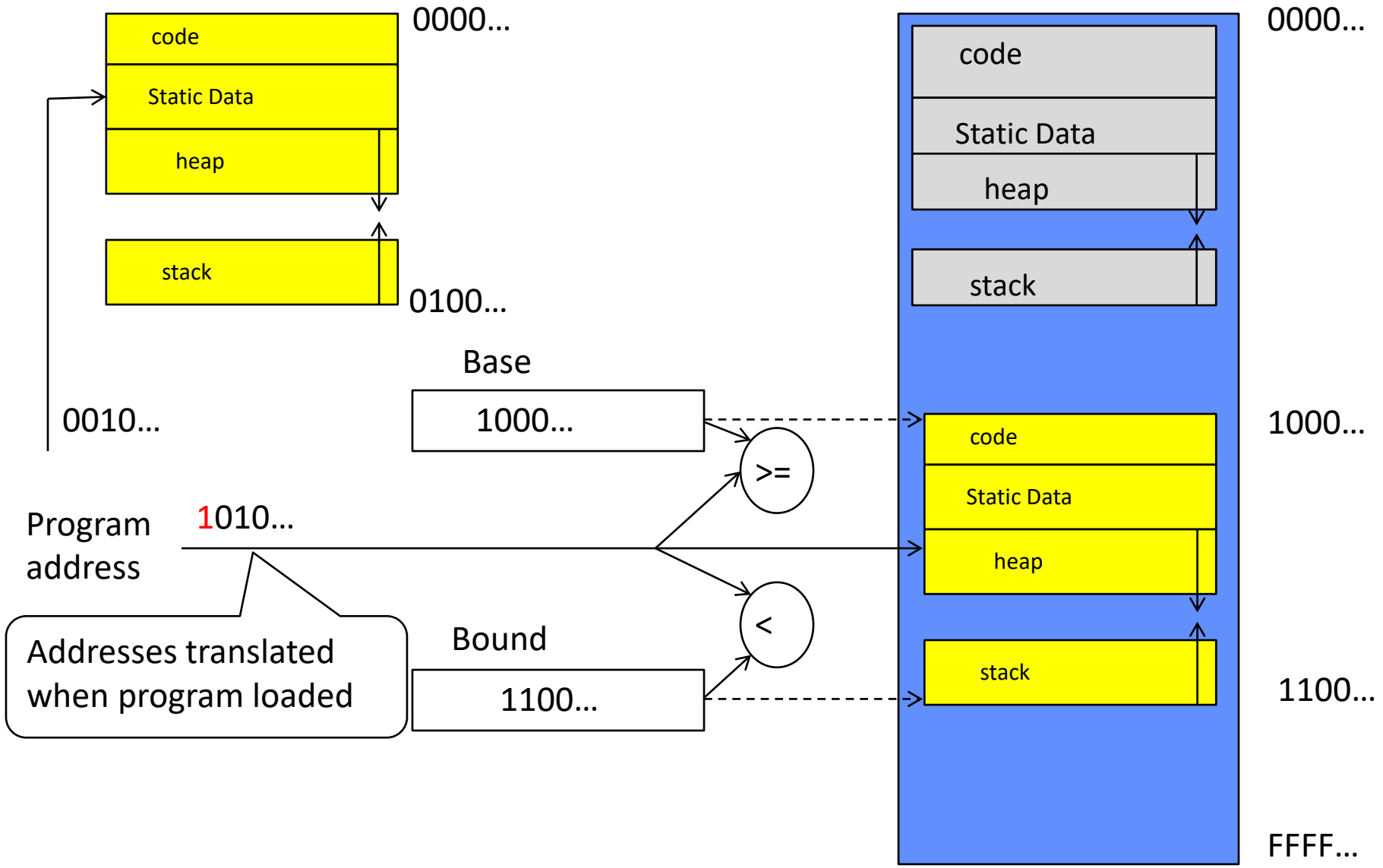
- OS must protect itself from user programs
  - Reliability: compromising the operating system generally causes it to crash
  - Security: limit the scope of what threads can do
  - Privacy: limit each thread to the data it is permitted to access
  - Fairness: each thread should be limited to its appropriate share of system resources (CPU, memory)

What can the hardware do to help the OS protect itself from programs???

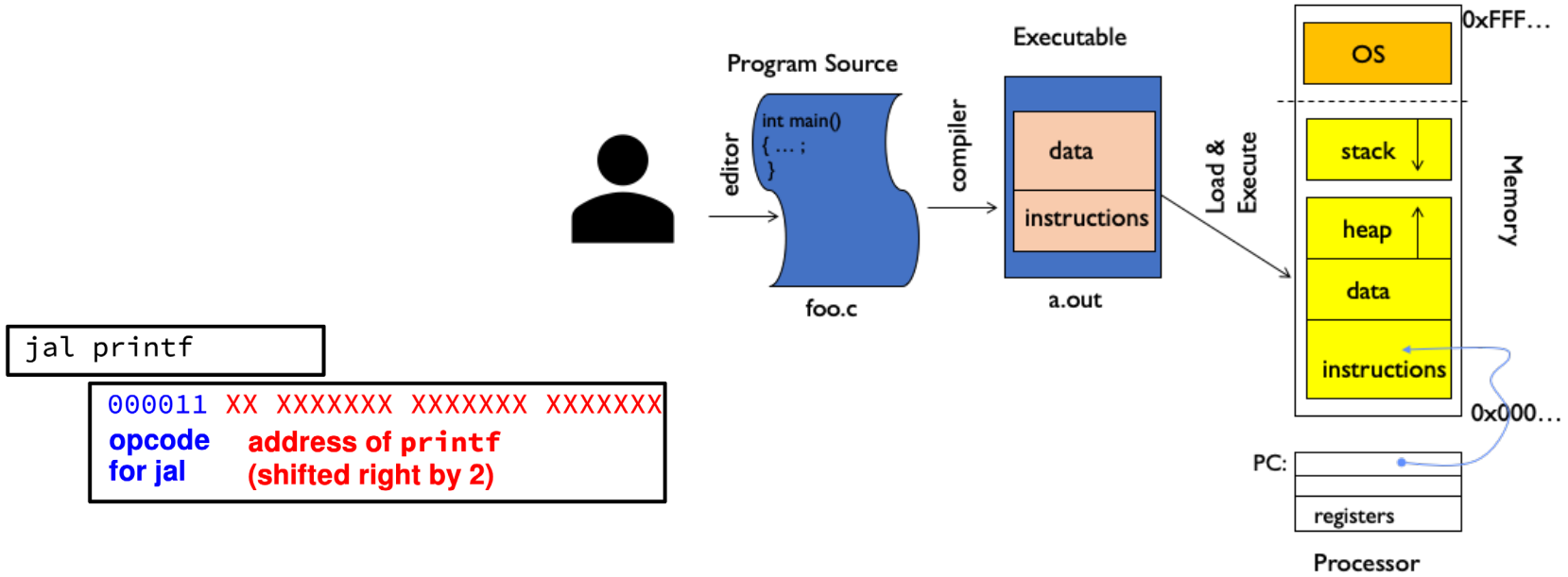
Simple Protection: Base and Bound (B&B)



Simple Protection: Base and Bound (B&B)

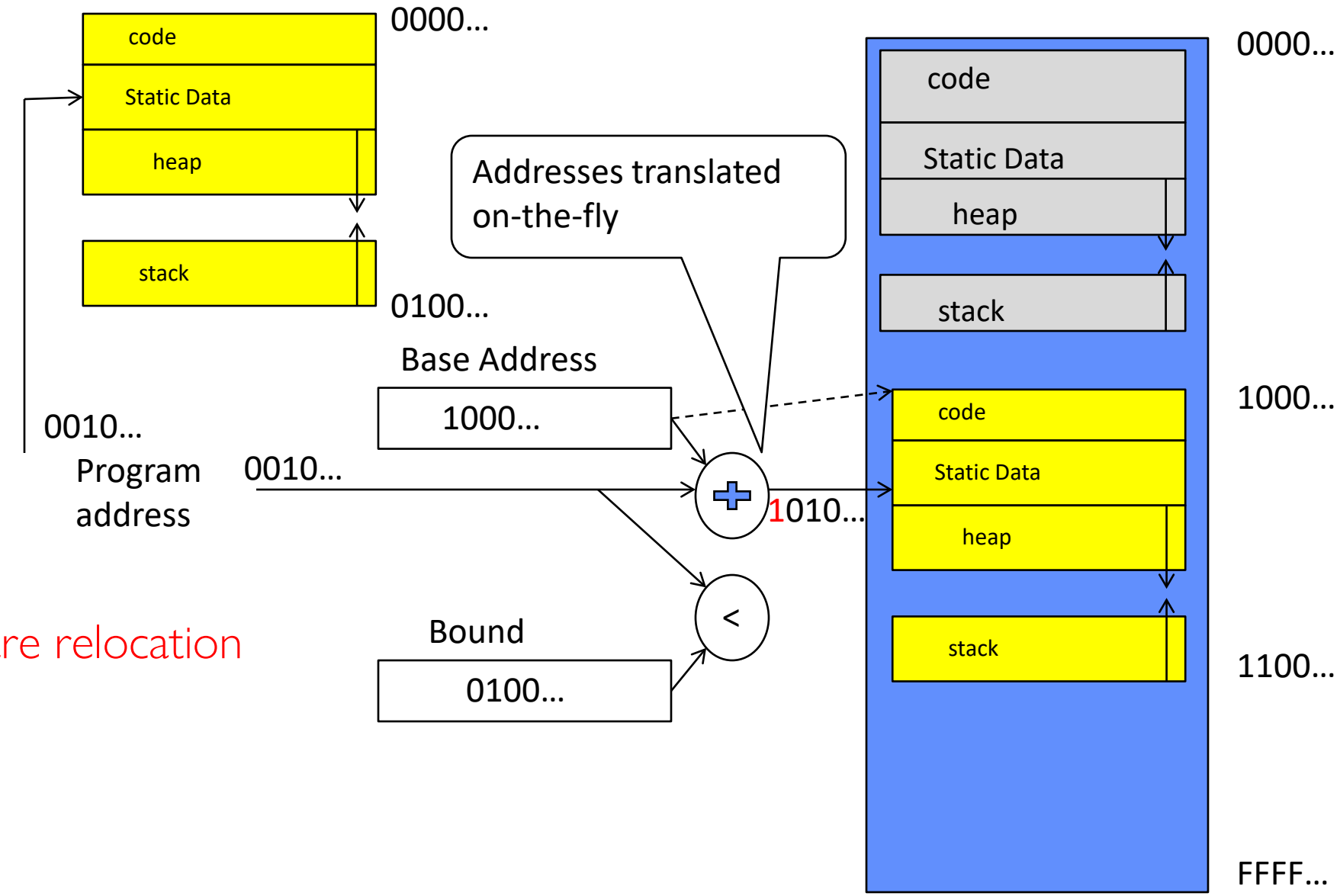


61C Review: Relocation



- Compiled .obj file linked together in an .exe
- All address in the .exe are as if it were loaded at memory address 00000000
- File contains a list of all the addresses that need to be adjusted when it is "relocated" to somewhere else.

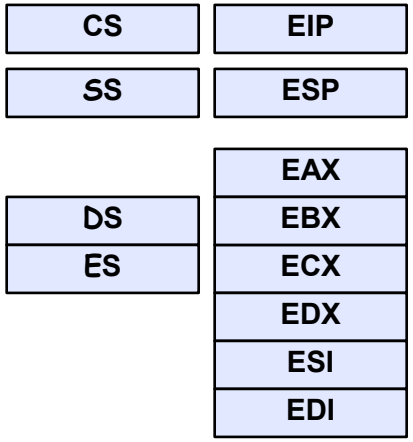
Simple address translation with Base and Bound



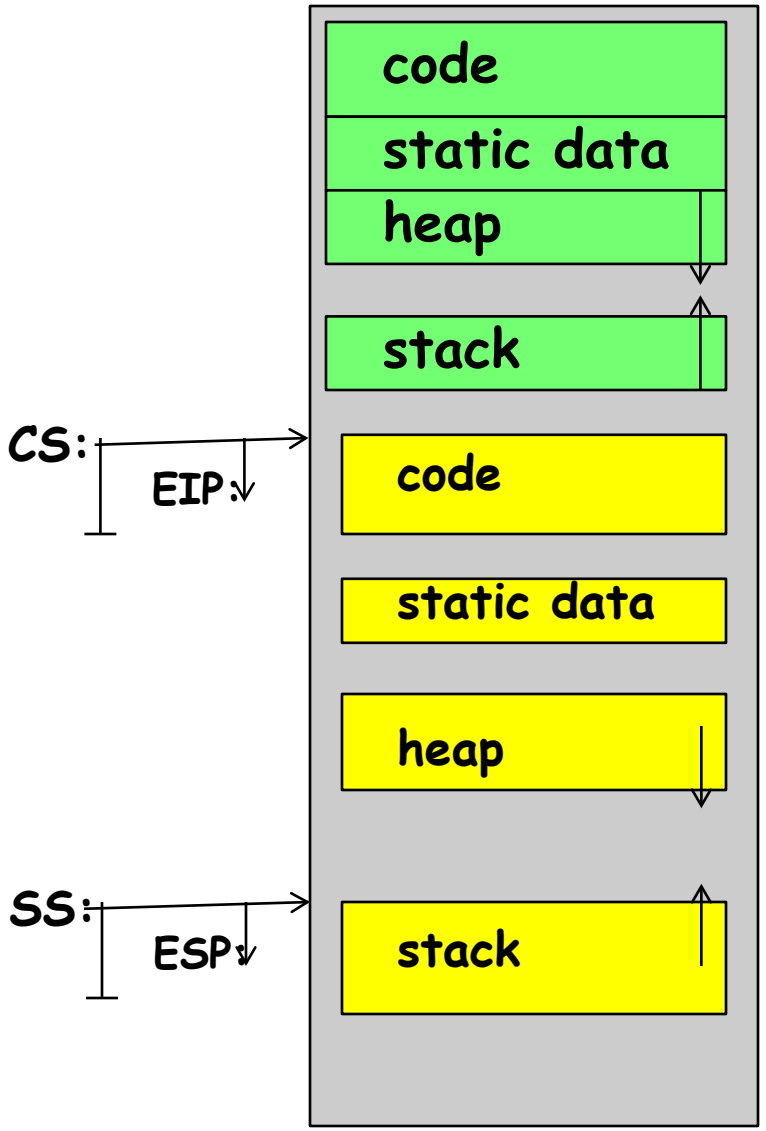
- Hardware relocation

x86 – segments and stacks

Processor Registers

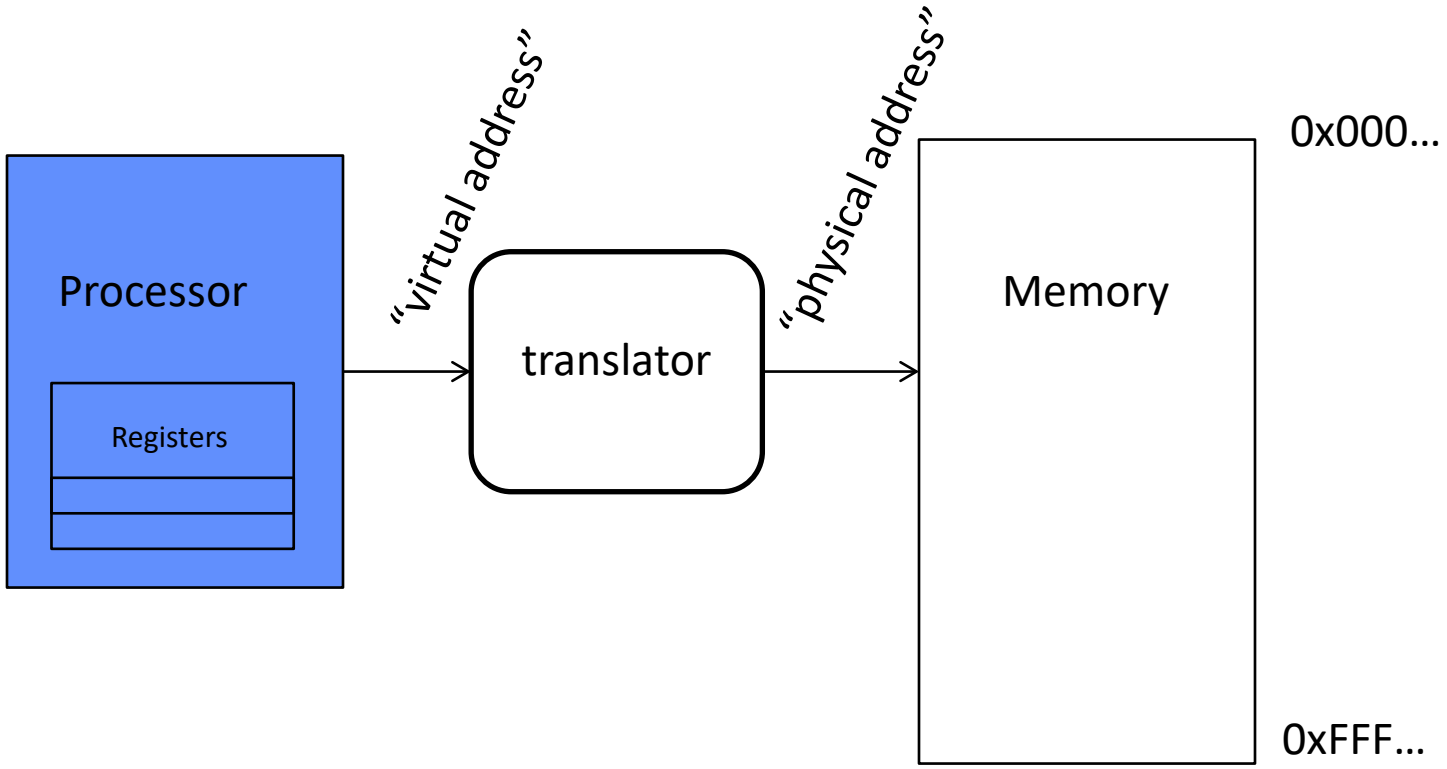


Start address, length and access rights associated with each segment register



Another idea: Address Space Translation

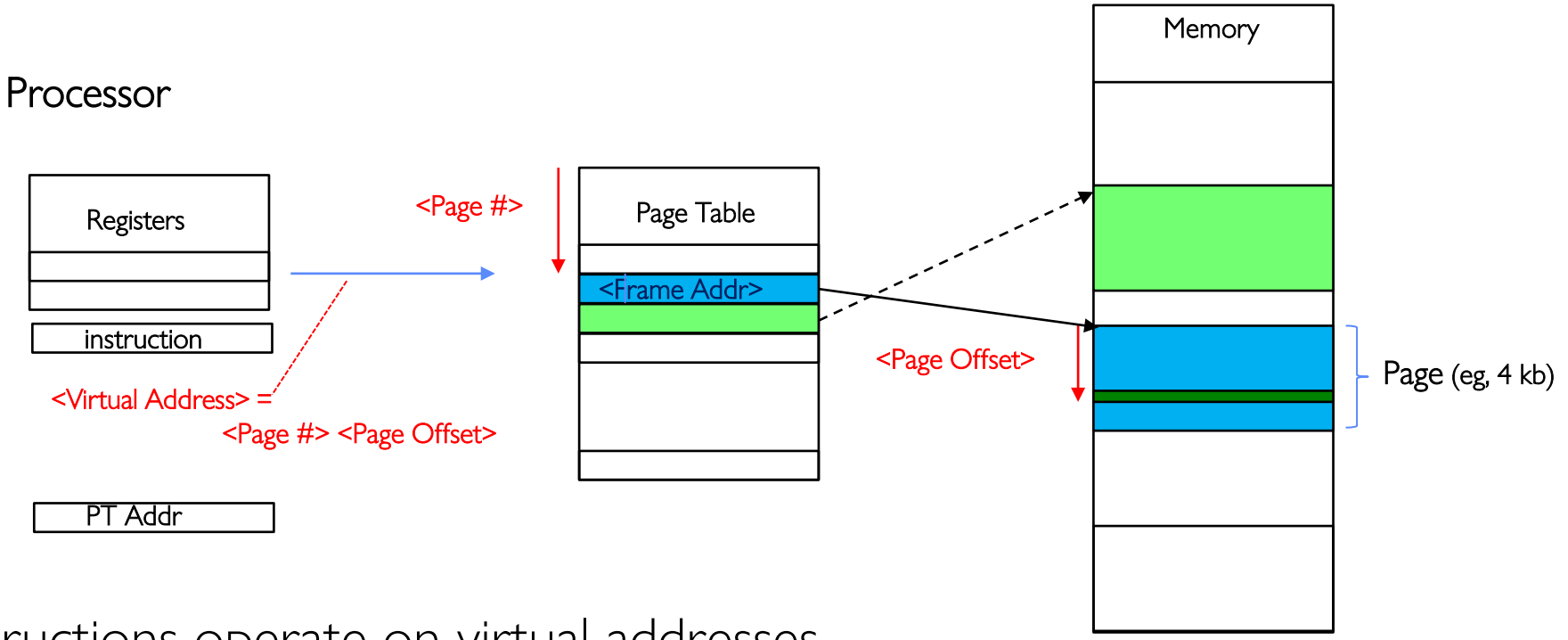
- Program operates in an address space that is distinct from the physical memory space of the machine



Paged Virtual Address Space

- Break the entire virtual address space into equal size chunks (i.e., pages)
- All pages same size, so easy to place each page in memory!
- Hardware translates address using a **page table**
  - Each page has a separate base
  - The "bound" is the page size
  - Special hardware register stores pointer to page table

Paged Virtual Address



- Instructions operate on virtual addresses
- Translated to a physical address through a Page Table by the hardware
- Any Page of address space can be in any (page sized) frame in memory
  - Or not-present (access generates a page fault)
- Special register holds page table base address (of the process)

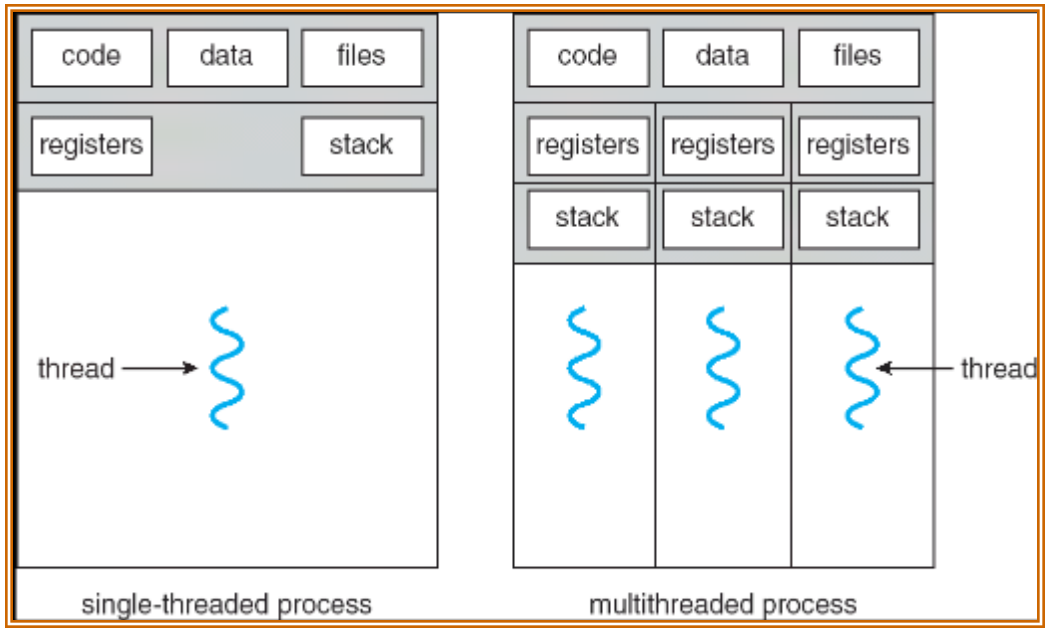
Third OS Concept: Process

- **Definition:** execution environment with restricted rights
  - (Protected) Address Space with One or More Threads
  - Owns memory (address space), file descriptors, sockets
  - Encapsulate one or more threads sharing process resources

- Why **processes**?
  - Protected from each other! OS Protected from them
  - Processes provides memory protection

- A process is a running program, with protection

Single and Multithreaded Processes



- Threads encapsulate concurrency
- Address spaces encapsulate protection
- Why have multiple threads per address space?
  - Parallelism: take advantage of actual hardware parallelism (e.g. multicore)
  - Concurrency: ease of handling I/O and other simultaneous events

Protection and Isolation

- Processes provide protection and isolation
  - Reliability: bugs can only overwrite memory of process they are in
  - Security and privacy: malicious or compromised process can't read or write other process' data

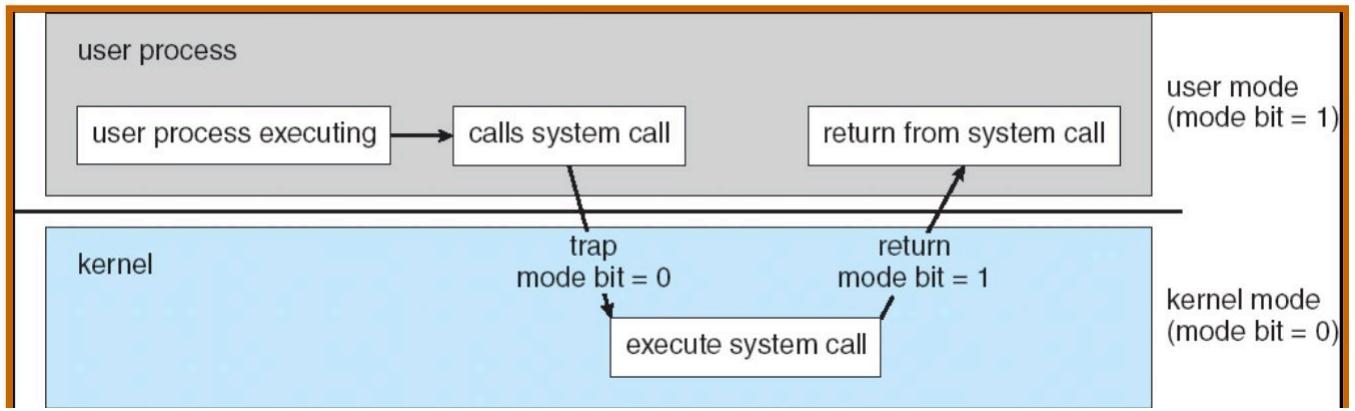
- Mechanisms:
  - Address translation: address space only contains its own data
  - BUT: why can't a process change the page table pointer?
    - » Or use I/O instructions to bypass the system?
  - Hardware must support **privilege levels**

Fourth OS Concept: Dual Mode Operation

- Hardware provides at least two modes
  1. Kernel Mode (or "supervisor" mode)
  2. User Mode

- Certain operations are prohibited when running in user mode (privileged instructions)

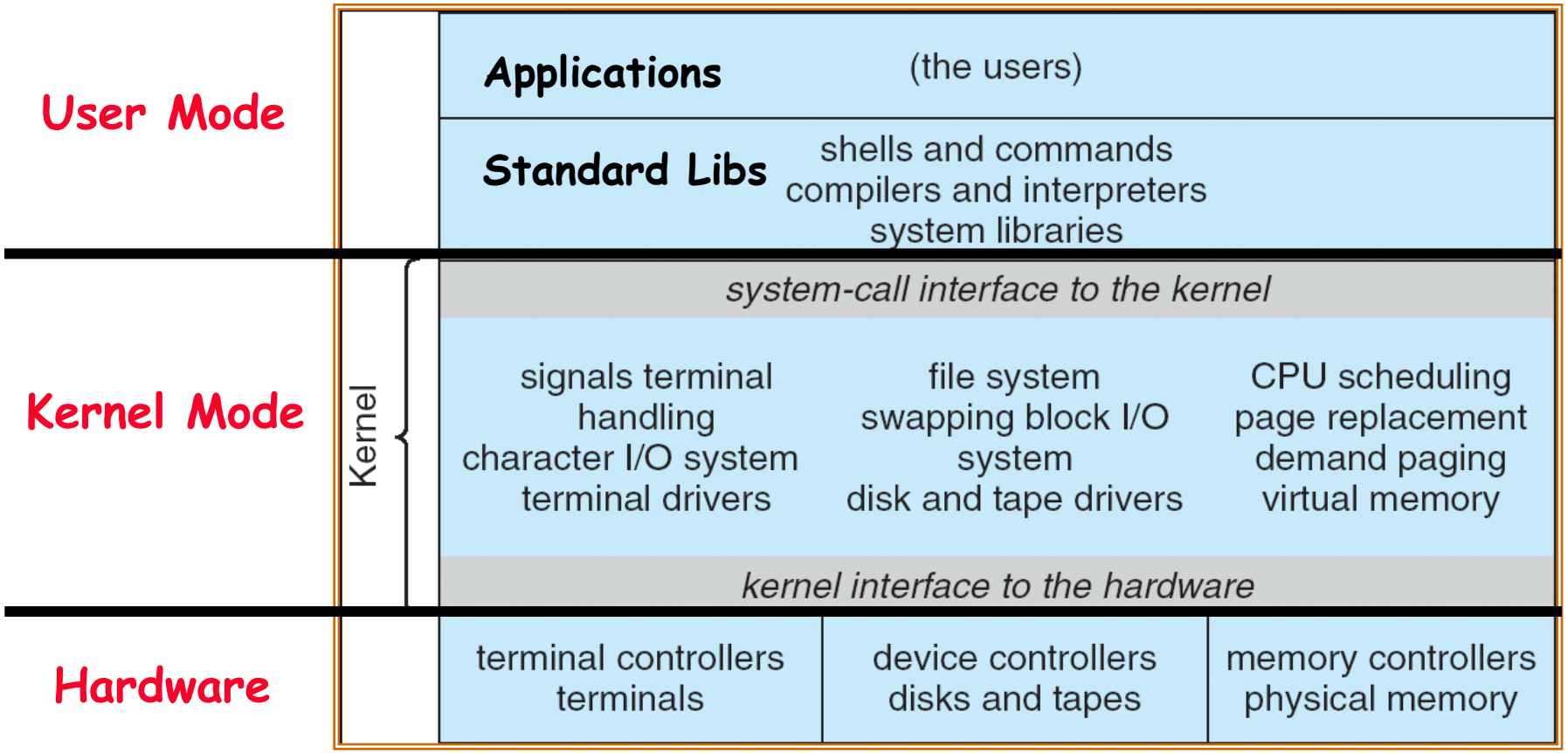
- Carefully controlled transitions between user mode and kernel mode



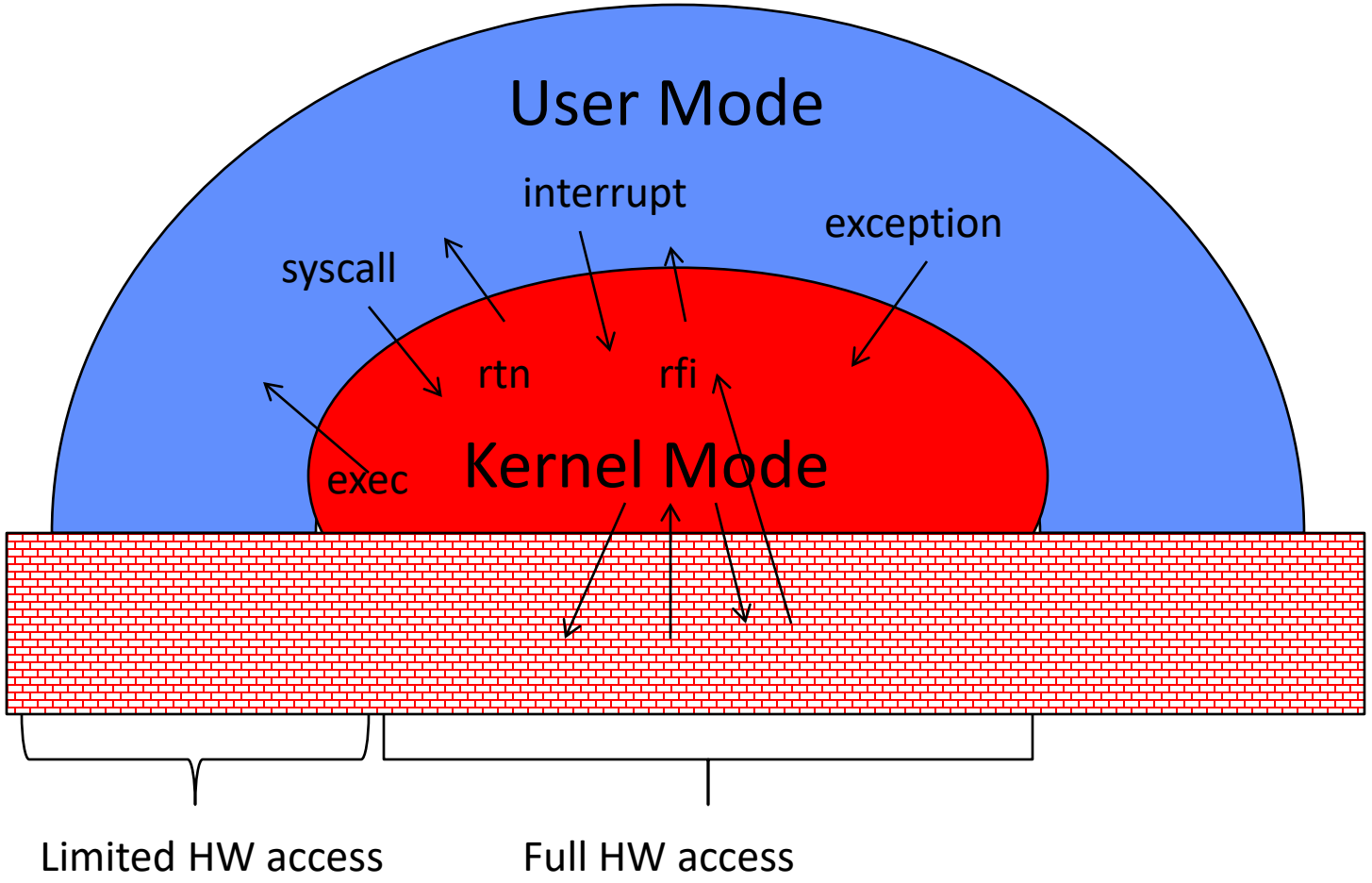
3 types of User ⇒ Kernel Mode Transfer

- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but "outside" the process
- Interrupt
  - External asynchronous event triggers context switch
  - e. g., Timer, I/O device
- Trap or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …

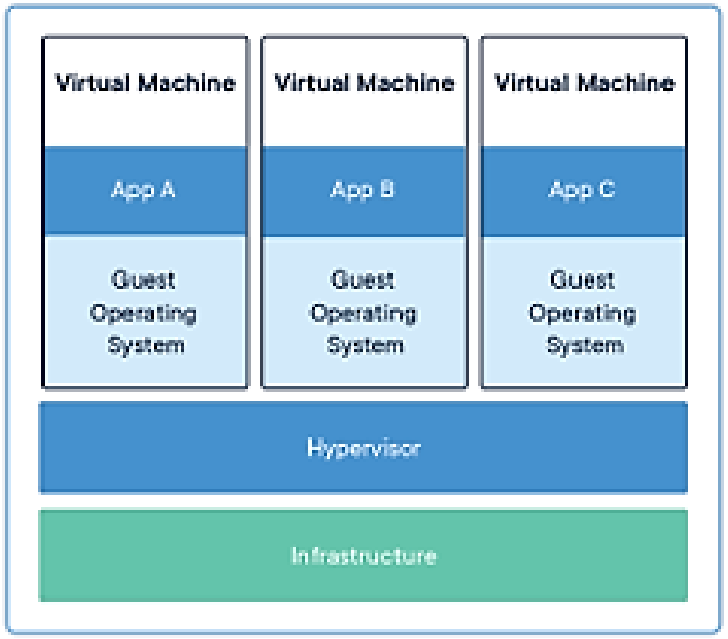
For example: UNIX System Structure

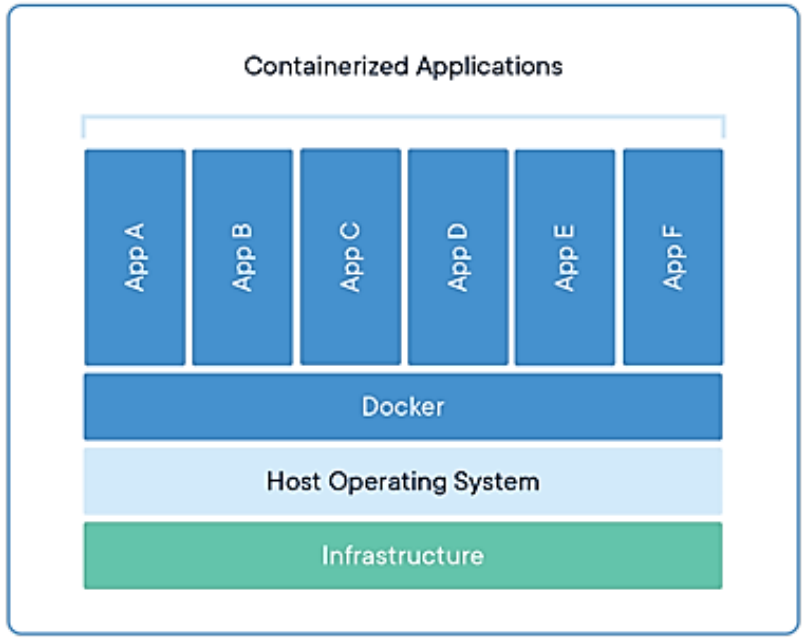


User/Kernel (Privileged) Mode



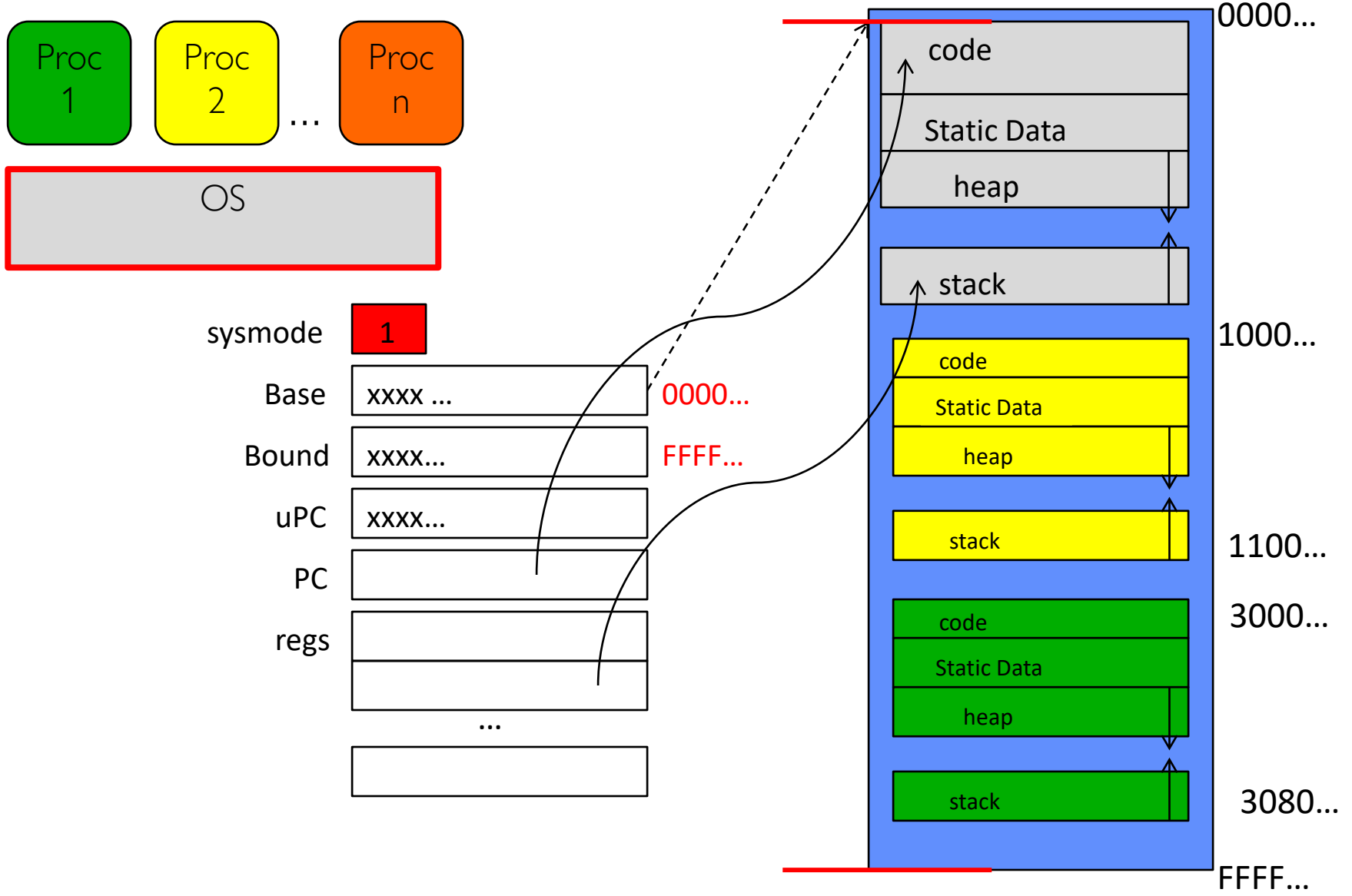
Additional Layers of Protection for Modern Systems



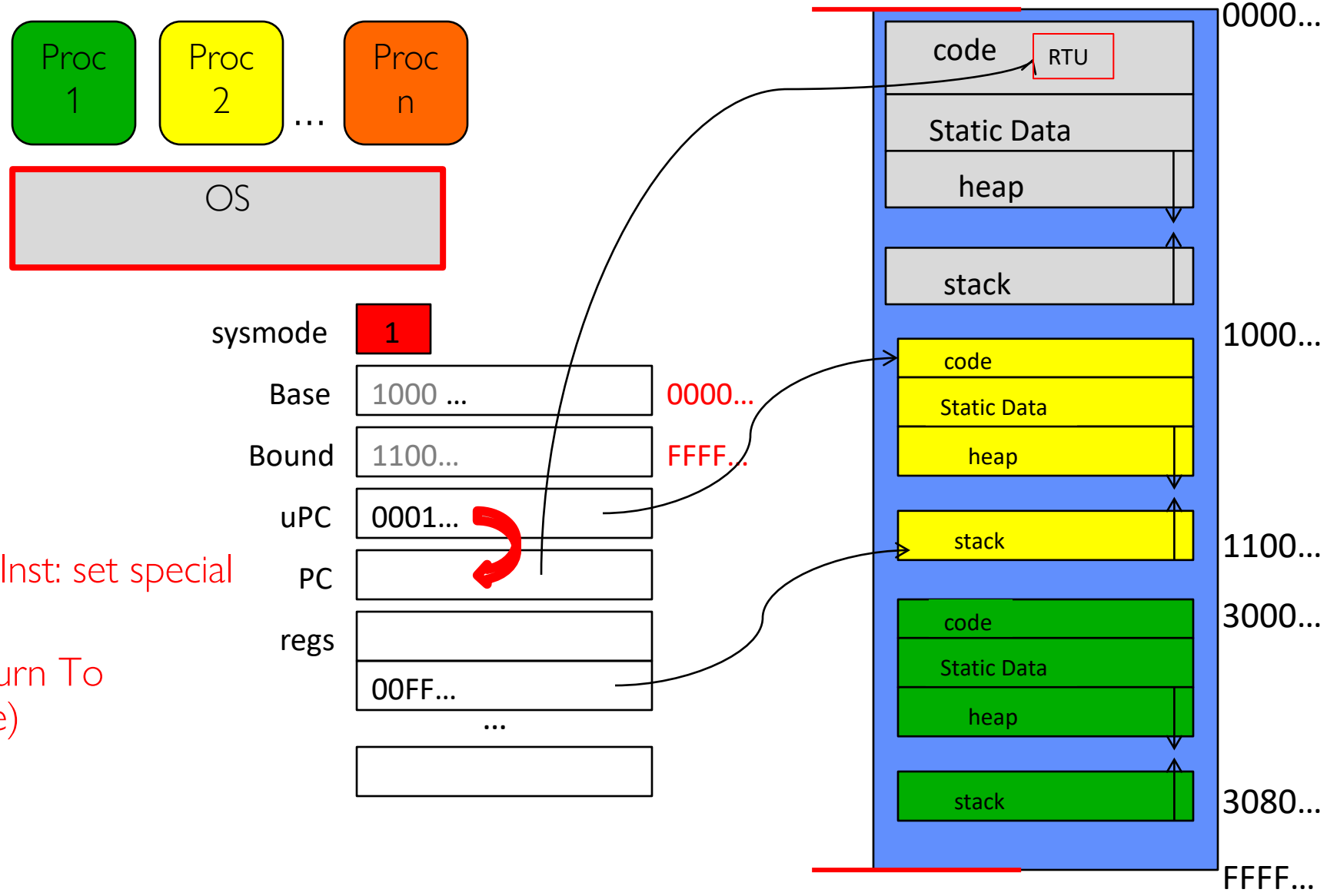


- Additional layers of protection through virtual machines or containers
  - Run a complete operating system in a virtual machine
  - Package all the libraries associated with an app into a container for execution
- More on these ideas later in the class

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



Simple B&B: OS gets ready to execute process

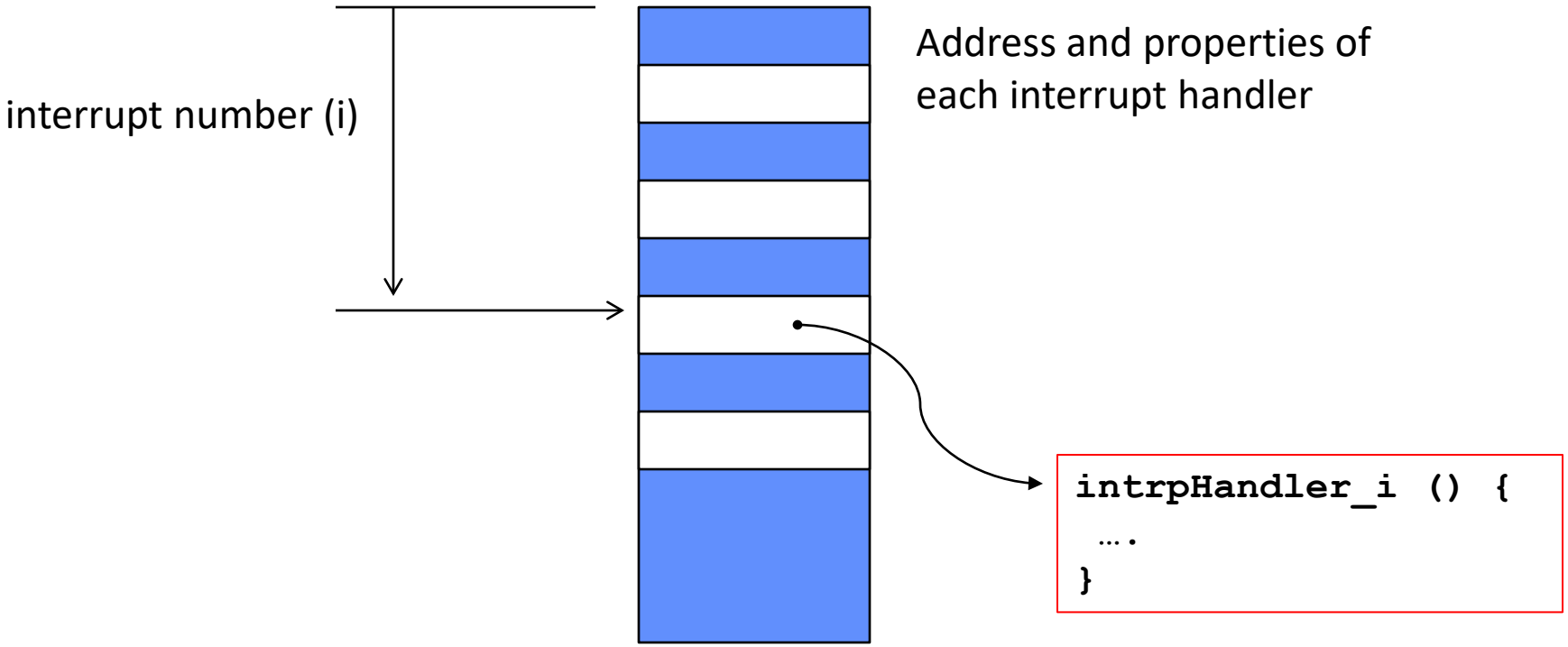


- Privileged Inst: set special registers
- RTU (Return To Usermode)

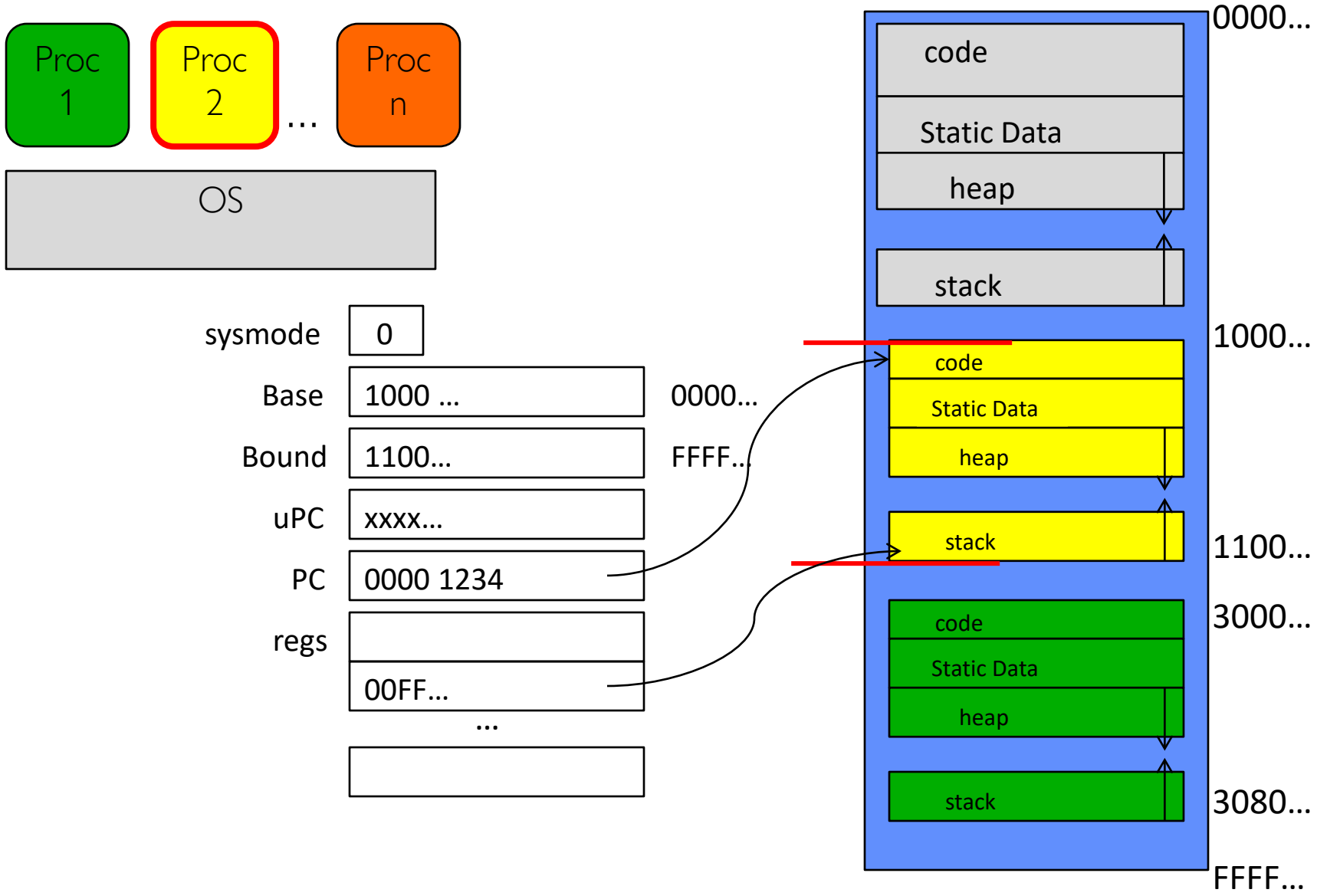
Unprogrammed control transfers

- User => Kernel mode transitions are examples of "unprogrammed control transfers"
- How do we know what the address of the next instruction should be?
- Will require support of lookup tables

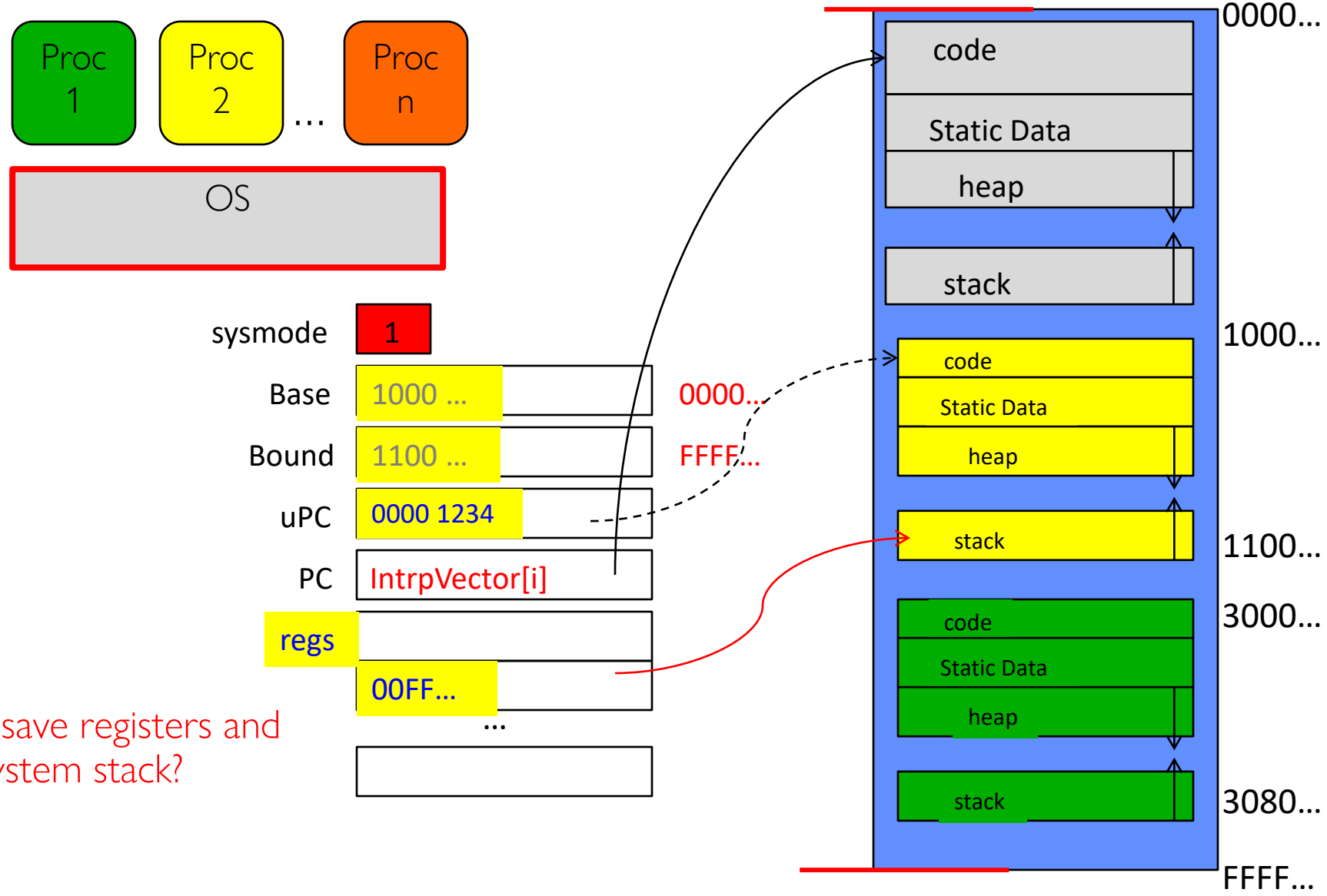
Interrupt Vector



Simple B&B: User => Kernel

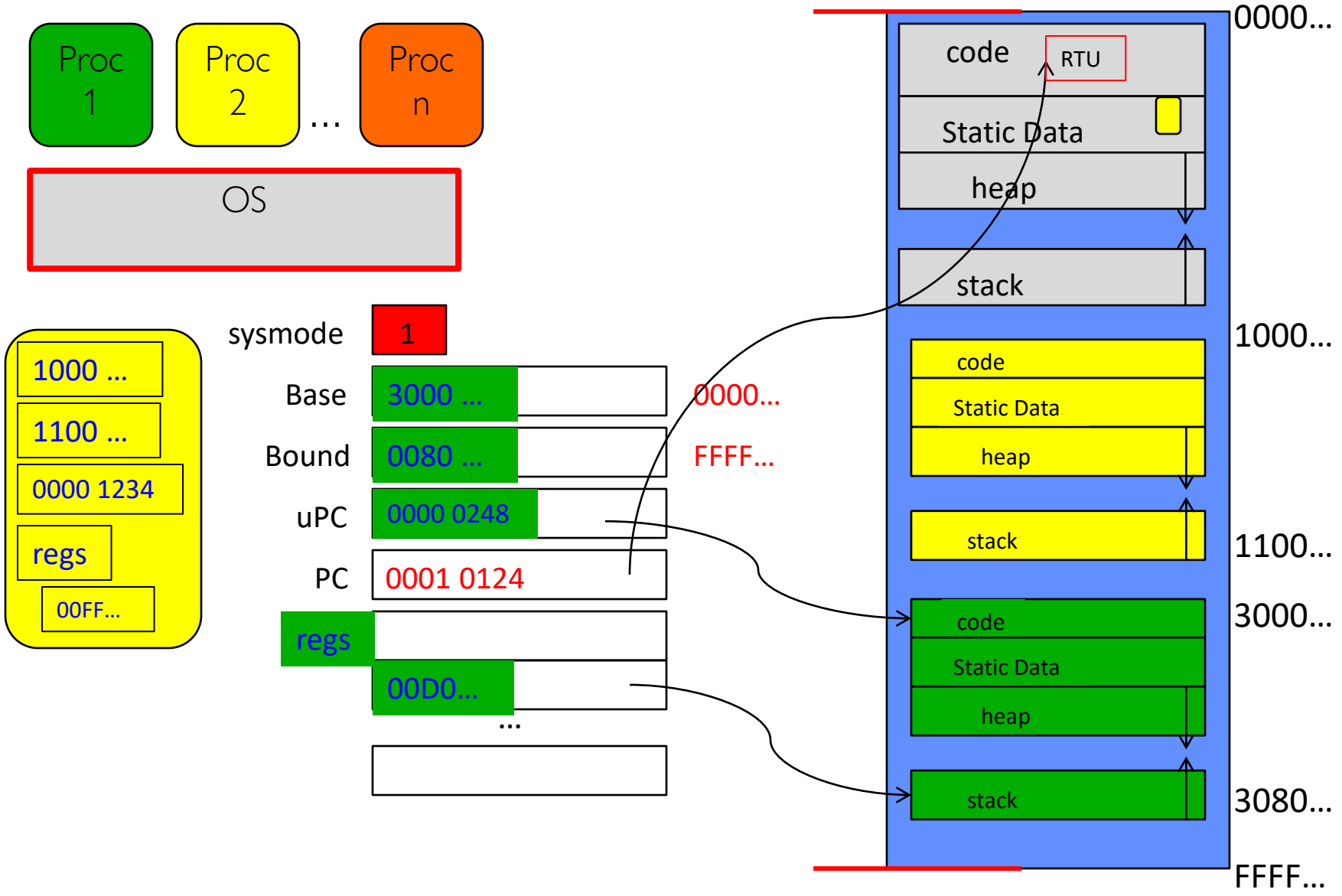


Simple B&B: Interrupt

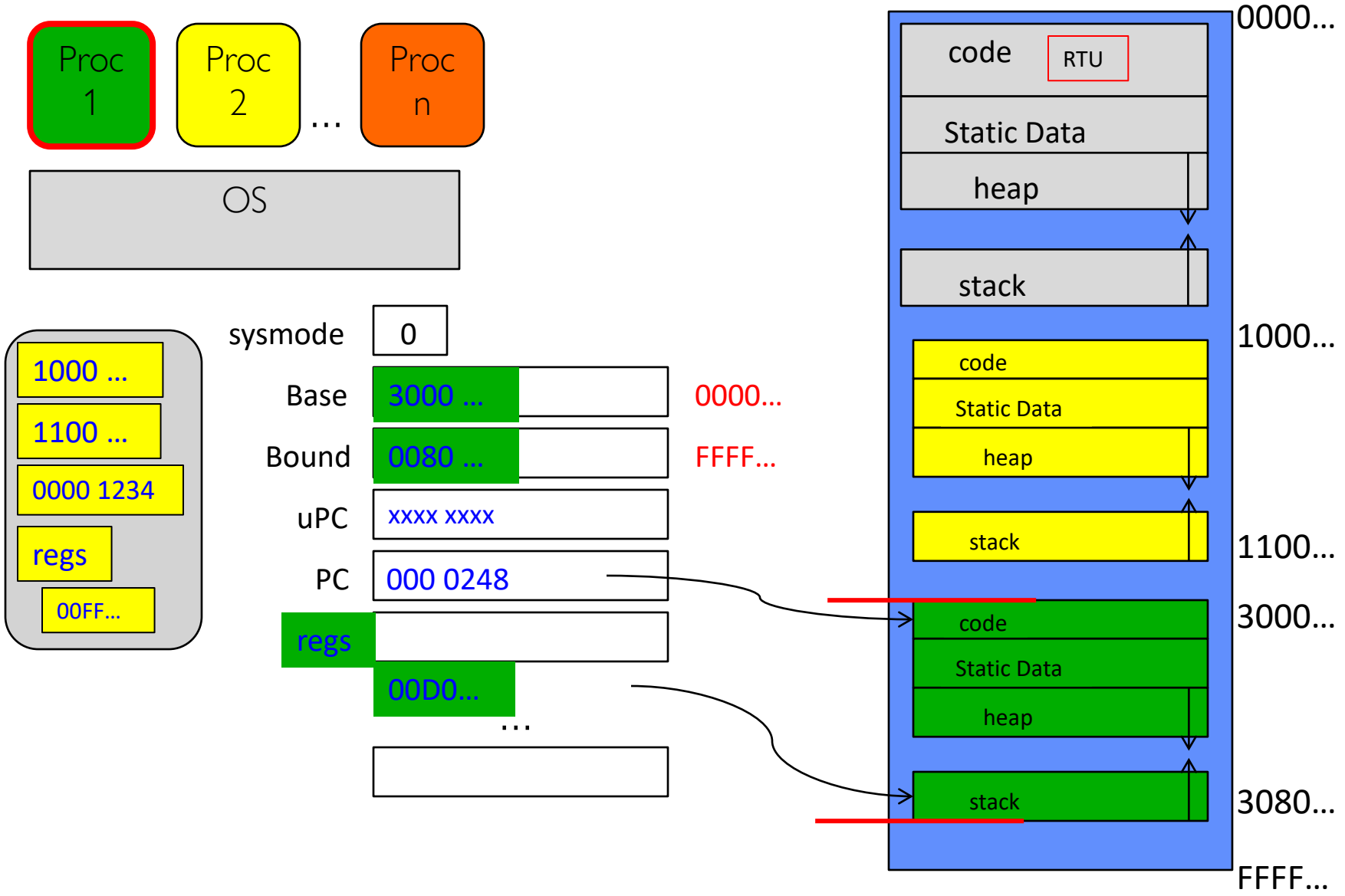


- How to save registers and set up system stack?

Simple B&B: Switch User Process



Simple B&B: "resume"



Running Many Programs ???

- We have the basic mechanism to
  - switch between user processes and the kernel,
  - the kernel can switch among user processes,
  - Protect OS from user processes and processes from each other

- How do we decide which user process to run?
- How do we represent user processes in the OS?
- How do we pack up the process and set it aside?
- How do we get a stack and heap for the kernel?

Conclusion: Four Fundamental OS Concepts

- Thread: Execution Context
  - Single thread of execution
- Address space (with or w/o translation)
  - Set of memory addresses accessible to program (for read or write)
- Process: an instance of a running program
  - Protected Address Space + One or more Threads
- Dual mode operation / Protection
  - Only the "system" has the ability to access certain resources