CS162
Operating Systems and Systems Programming
Lecture 2

Four Fundamental OS Concepts

January 23th, 2020

Prof. John Kubiatowicz

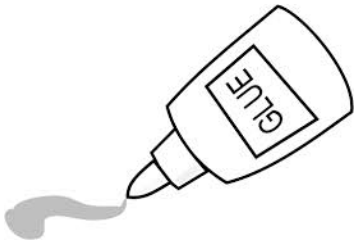
http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. *When slides are obtained from other sources, a a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.*

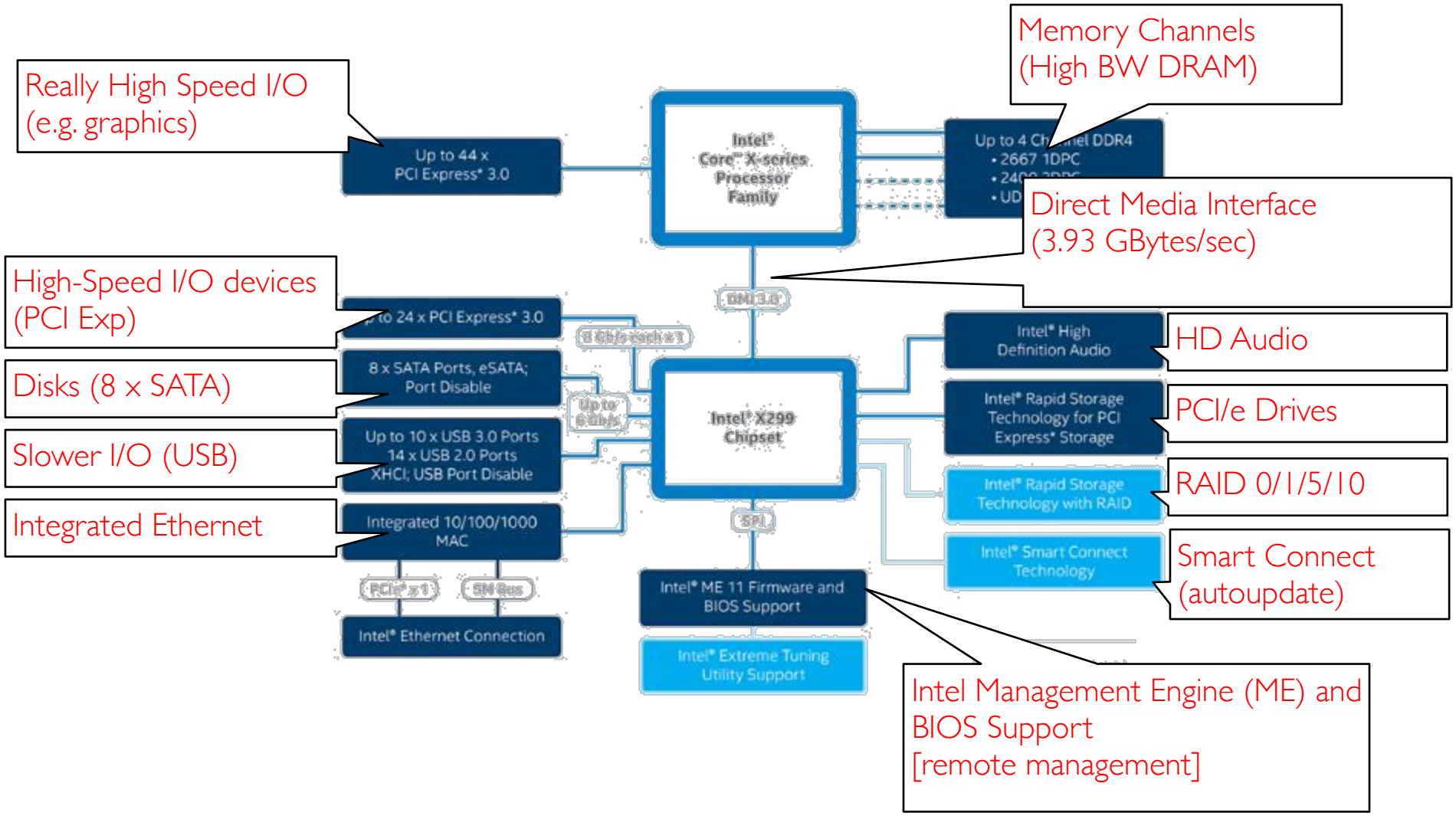
Review: What is an Operating System?

- Referee
  - Manage sharing of resources, Protection, Isolation
    - » Resource allocation, isolation, 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

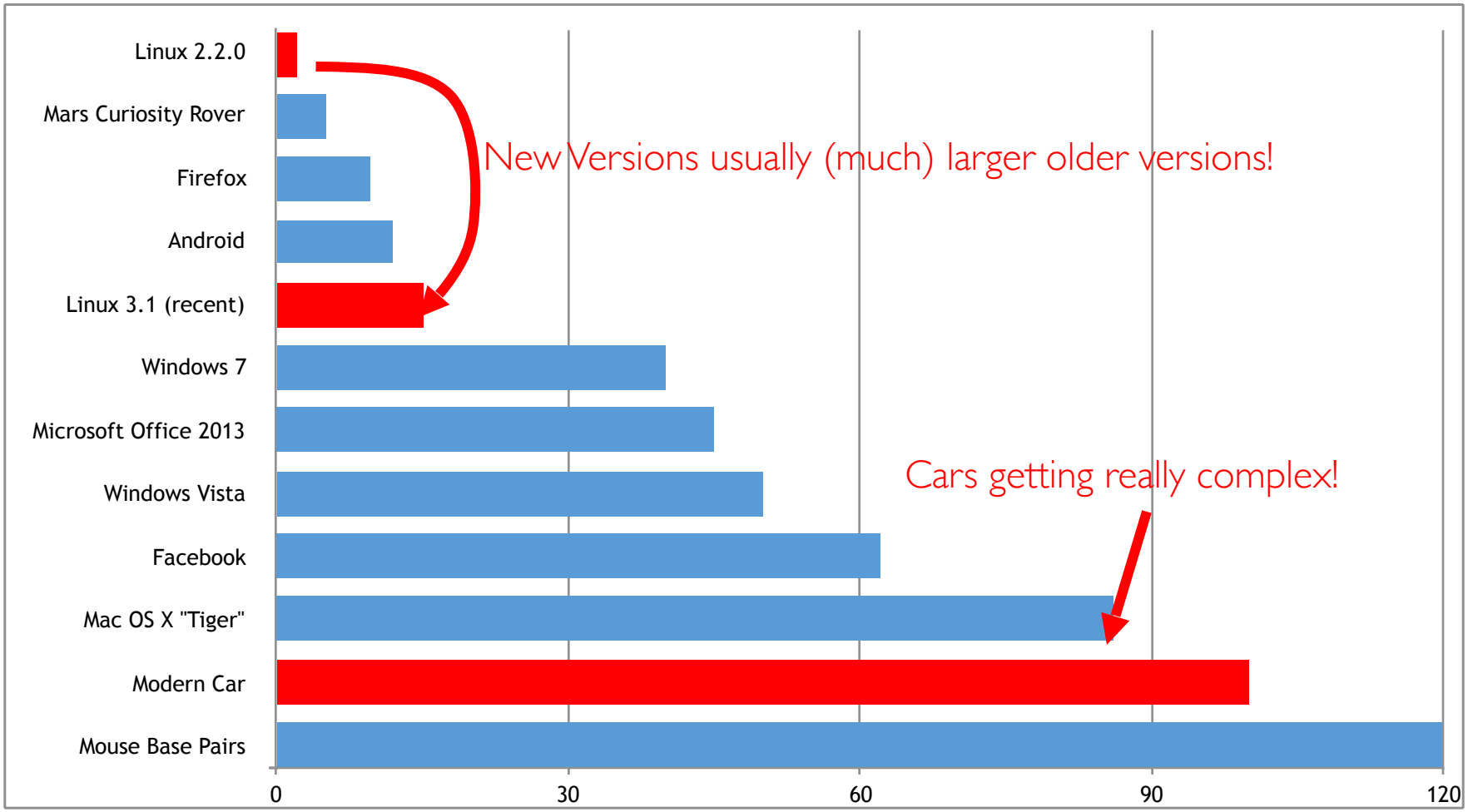


Recall: HW Functionality ⇒ great complexity!



Intel Skylake-X I/O Configuration

Recall: Increasing Software Complexity



Millions of Lines of Code

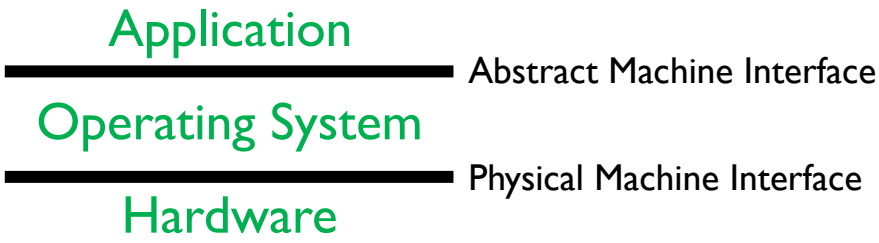
(source https://informationisbeautiful.net/visualizations/million-lines-of-code/)

Recall: How do we tame complexity?

- Every piece of computer hardware different
  - Different CPU
    - » Pentium, PowerPC, ColdFire, ARM, MIPS
  - Different amounts of memory, disk, …
  - Different types of devices
    - » Mice, Keyboards, Sensors, Cameras, Fingerprint readers
  - Different networking environment
    - » Cable, DSL, Wireless, Firewalls,…
- Questions:
  - Does the programmer need to write a single program that performs many independent activities?
  - Does every program have to be altered for every piece of hardware?
  - Does a faulty program crash everything?
  - Does every program have access to all hardware?

OS *Abstracts* underlying hardware

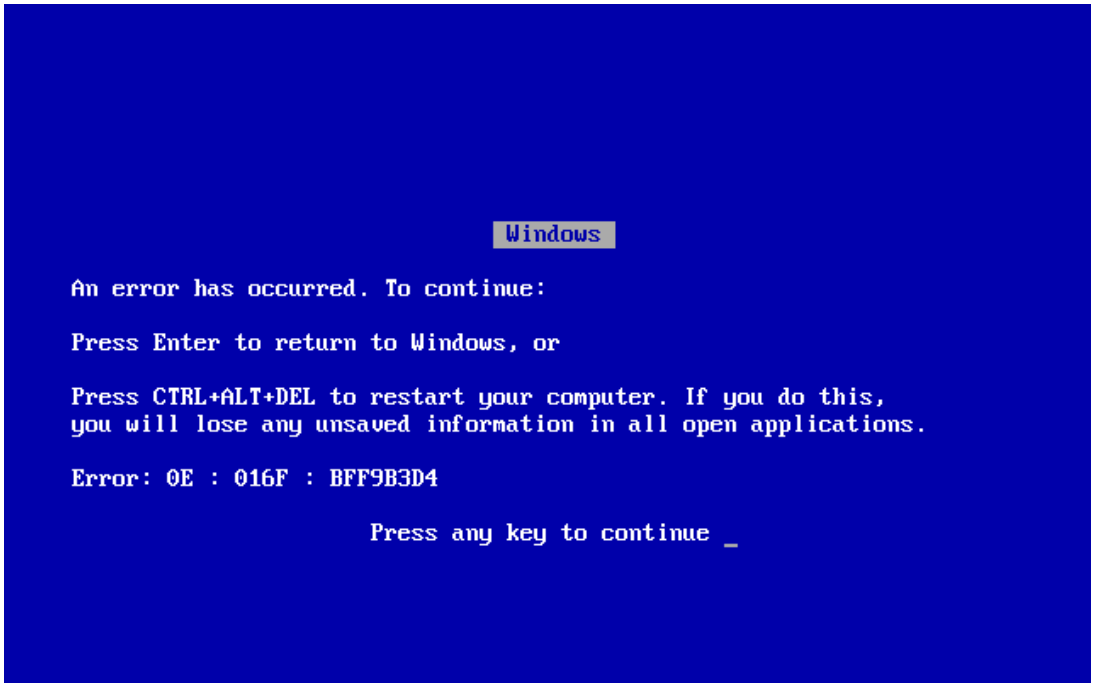
- Processor => Thread
- Memory => Address Space
- Disks, SSDs, … => Files
- Networks => Sockets
- Machines => Processes



- OS Goals:
  - 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)

OS Goal: Protecting Processes & The Kernel

- Run multiple applications and:
  - Keep them from interfering with or crashing the operating system
  - Keep them from interfering with or crashing each other

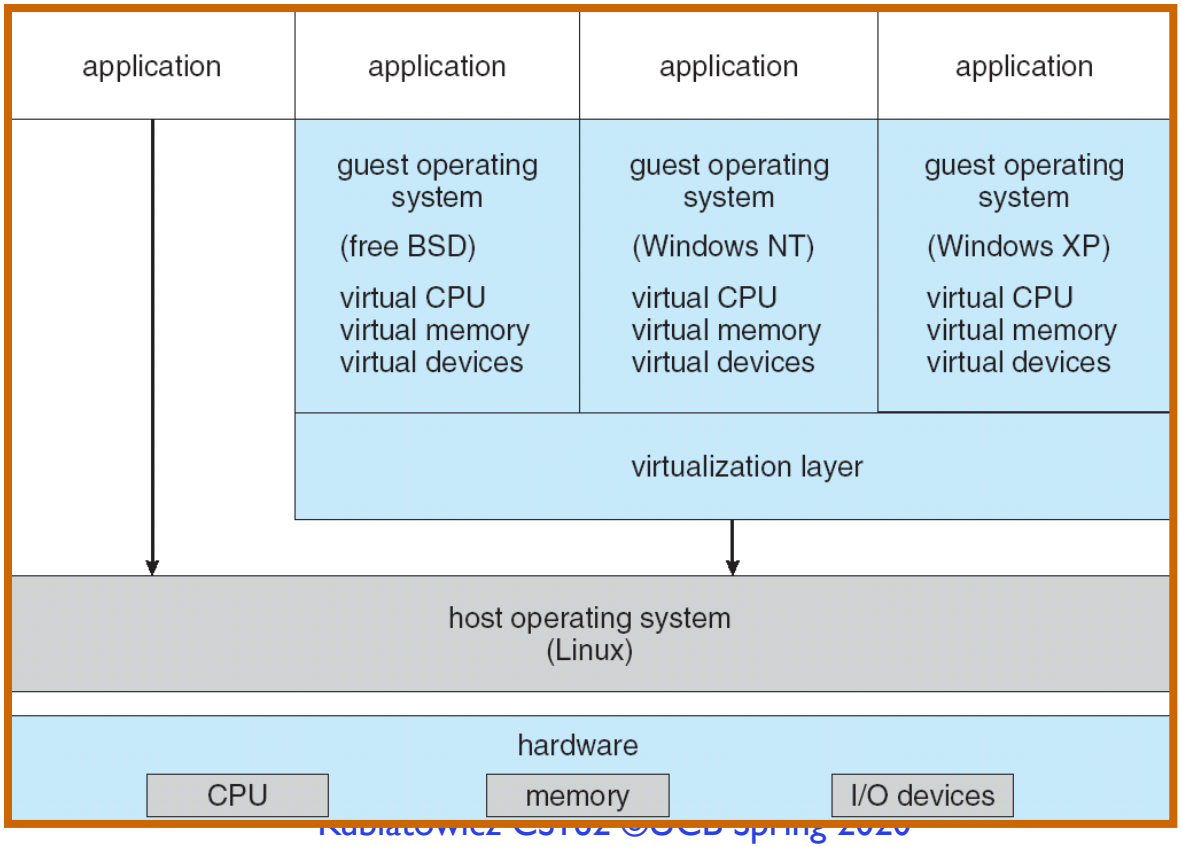


Virtual Machines

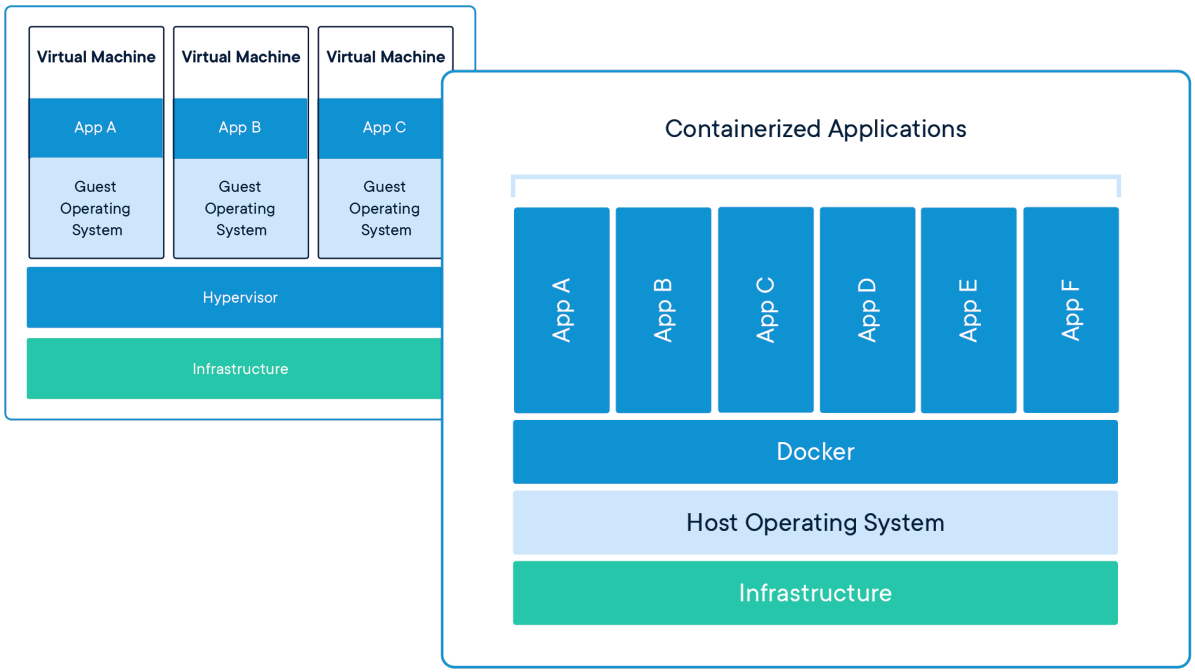
- Virtualize every detail of a hardware configuration so perfectly that you can run an operating system (and many applications) on top of it.
  - VMWare Fusion, Virtual box, Parallels Desktop, Xen, Vagrant
- Provides isolation
- Complete insulation from change
- The norm in the Cloud (server consolidation)
- Long history (60's in IBM OS development)
- All our work will take place INSIDE a VM
  - Vagrant (new image just for you)

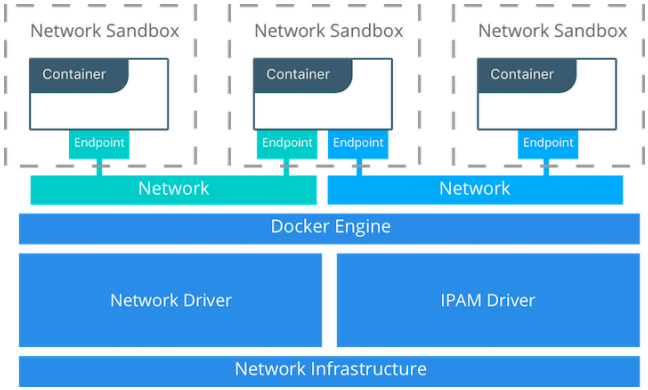
System Virtual Machines: Layers of OSs

- Useful for OS development
  - When OS crashes, restricted to one VM
  - Can aid testing/running programs on other Oss
- Use for deployment
  - Running different OSes at the same time



Containers *virtualize the OS*

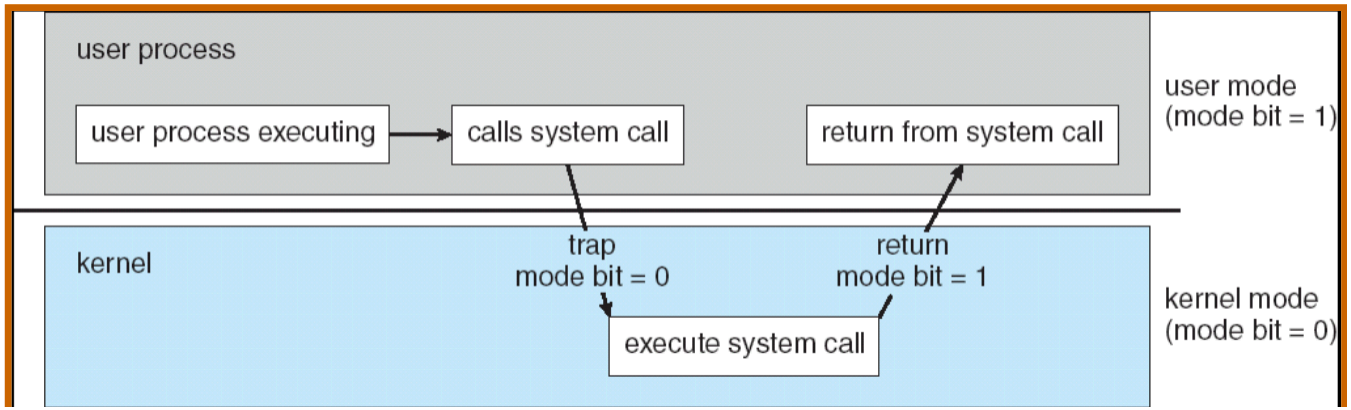




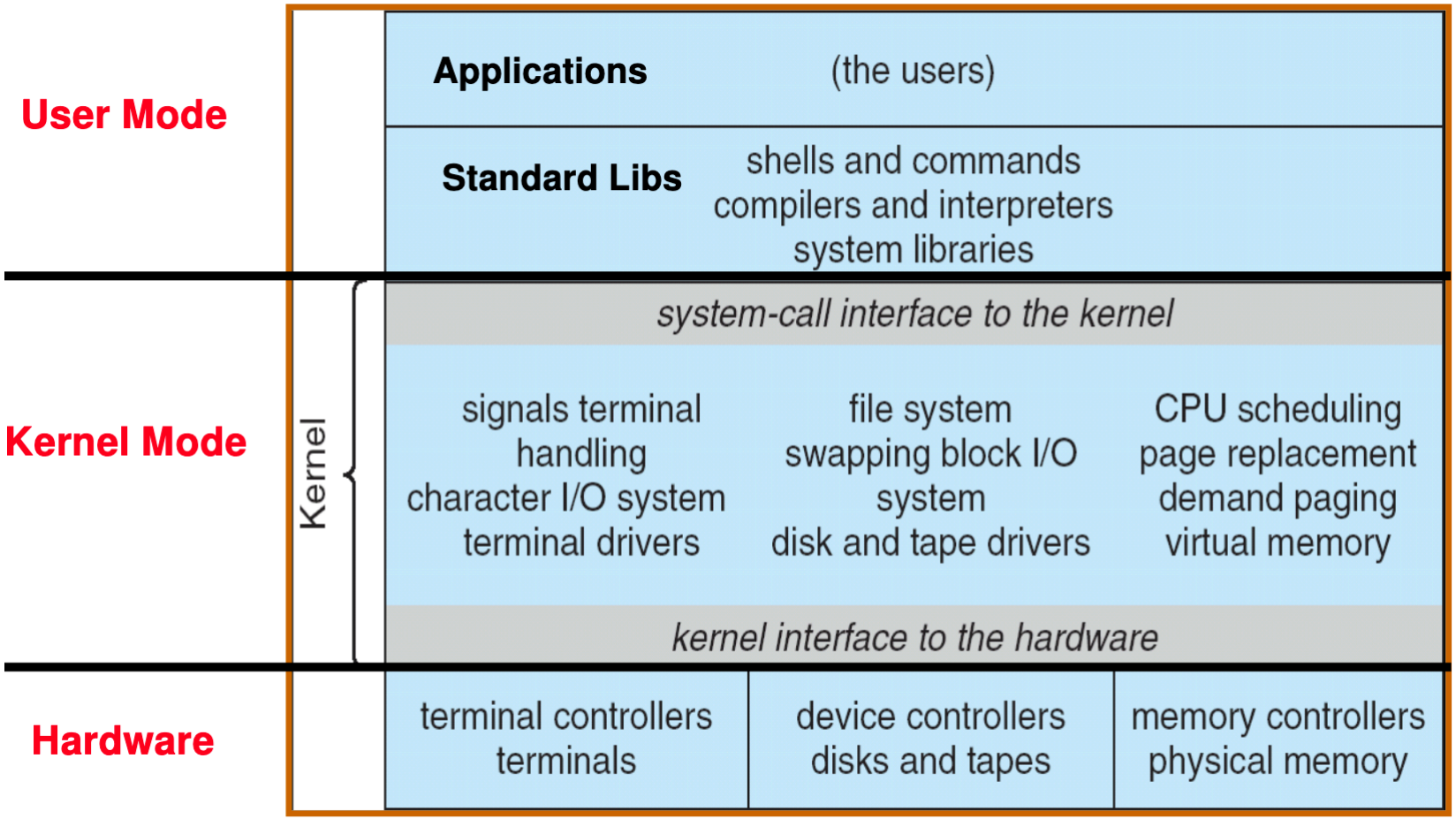
- Roots in OS developments to provide protected systems abstraction, not just application abstraction
  - User-level file system (route syscalls to user process)
  - Cgroups – predictable, bounded resources (CPU, Mem, BW)

Basic tool: Dual Mode Operation

- Hardware provides at least two modes:
  1. Kernel Mode (or "supervisor" / "protected" mode)
  2. User Mode
- Certain operations are prohibited when running in user mode
  - Changing the page table pointer
- Carefully controlled transitions between user mode and kernel mode
  - System calls, interrupts, exceptions



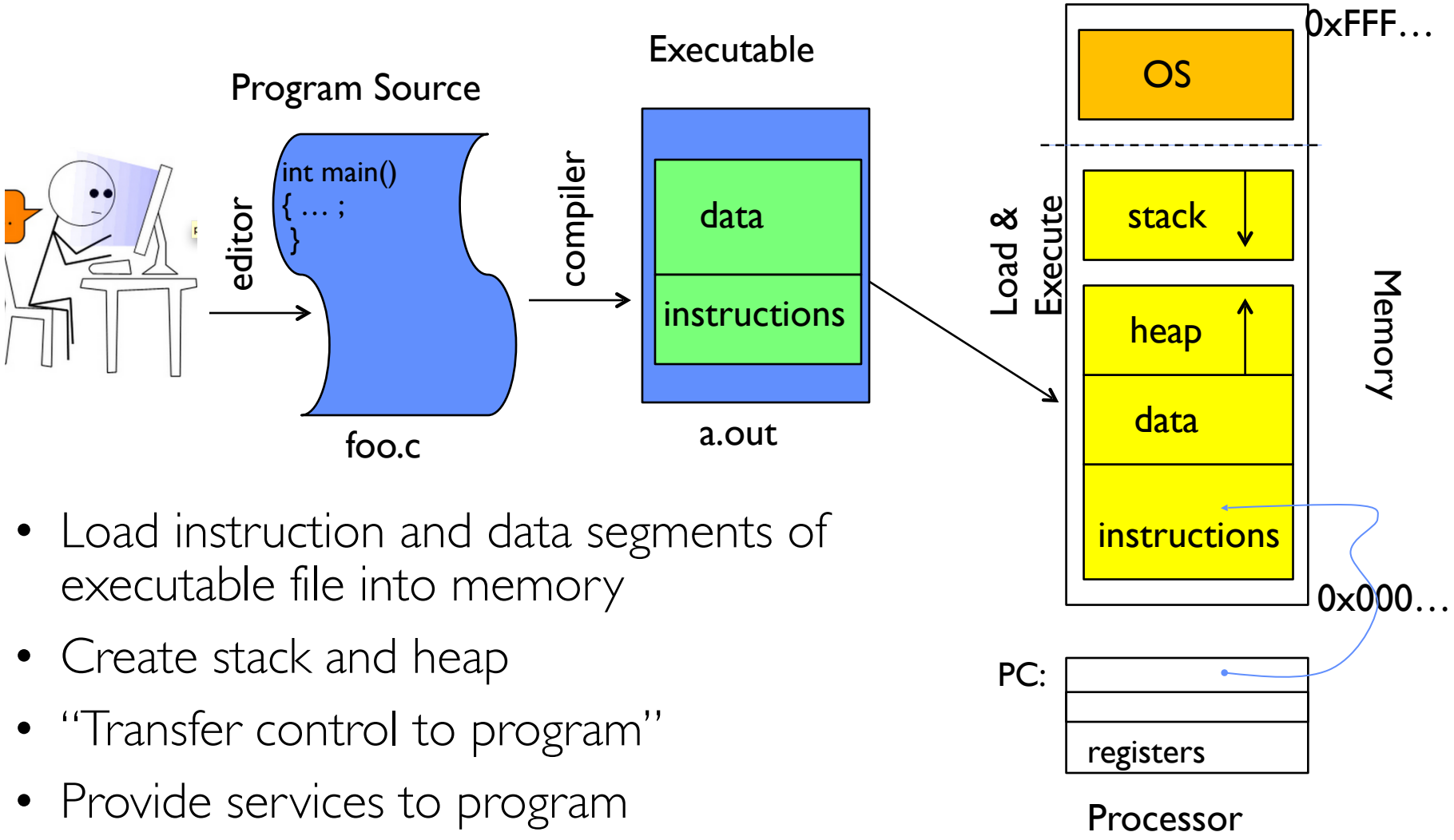
UNIX OS Structure



Today: Four Fundamental OS Concepts

- Thread: Execution Context
  - Fully describes program state
  - Program Counter, Registers, Execution Flags, Stack
- Address space (with or w/o translation)
  - Set of memory addresses accessible to program (for read or write)
  - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- 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
  - Combined with translation, isolates programs from each other and the OS from programs

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

Stack vs. Heap

Stack

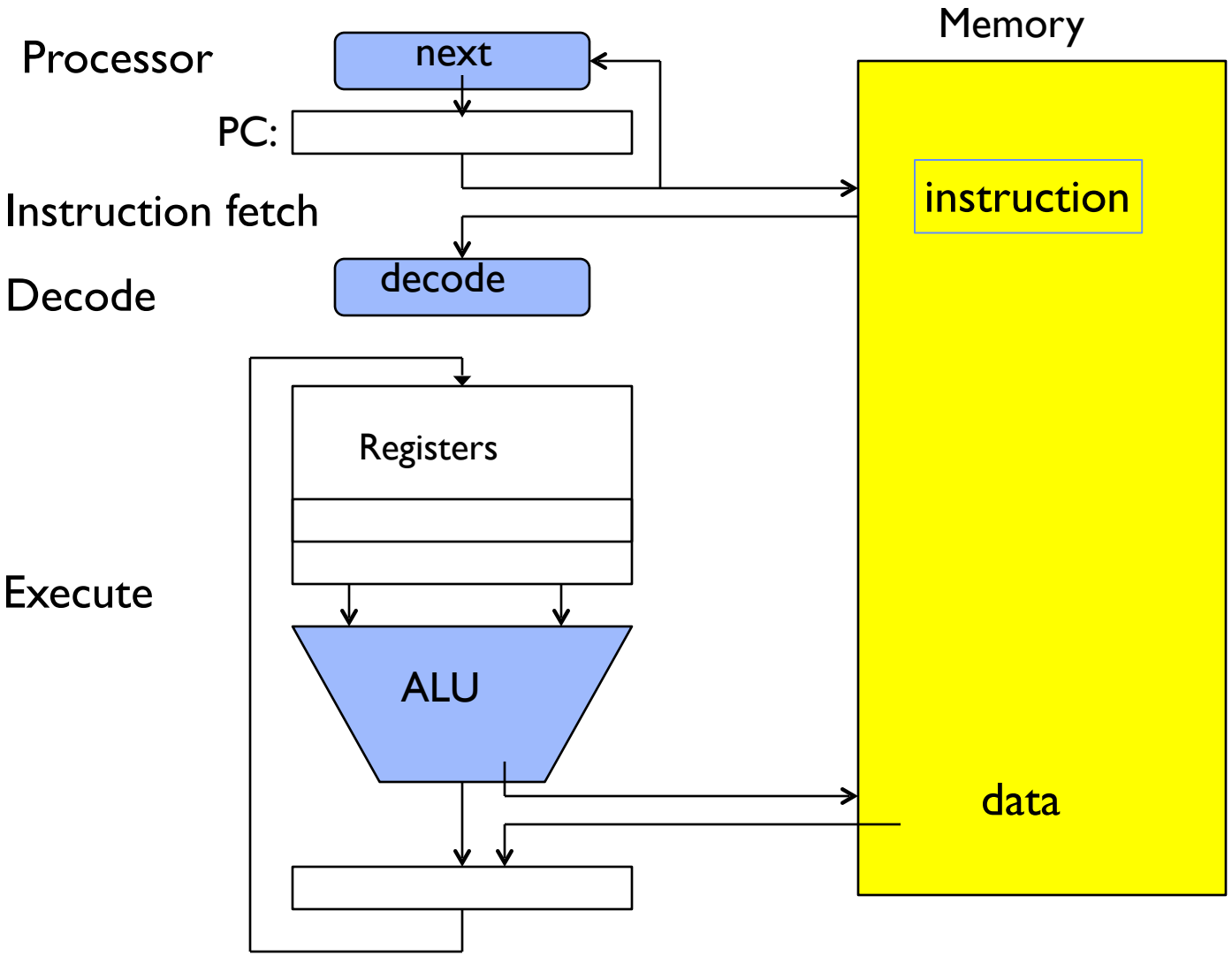
Heap

Google

@fhinkel

Recall (61C): Instruction Fetch/Decode/Execute

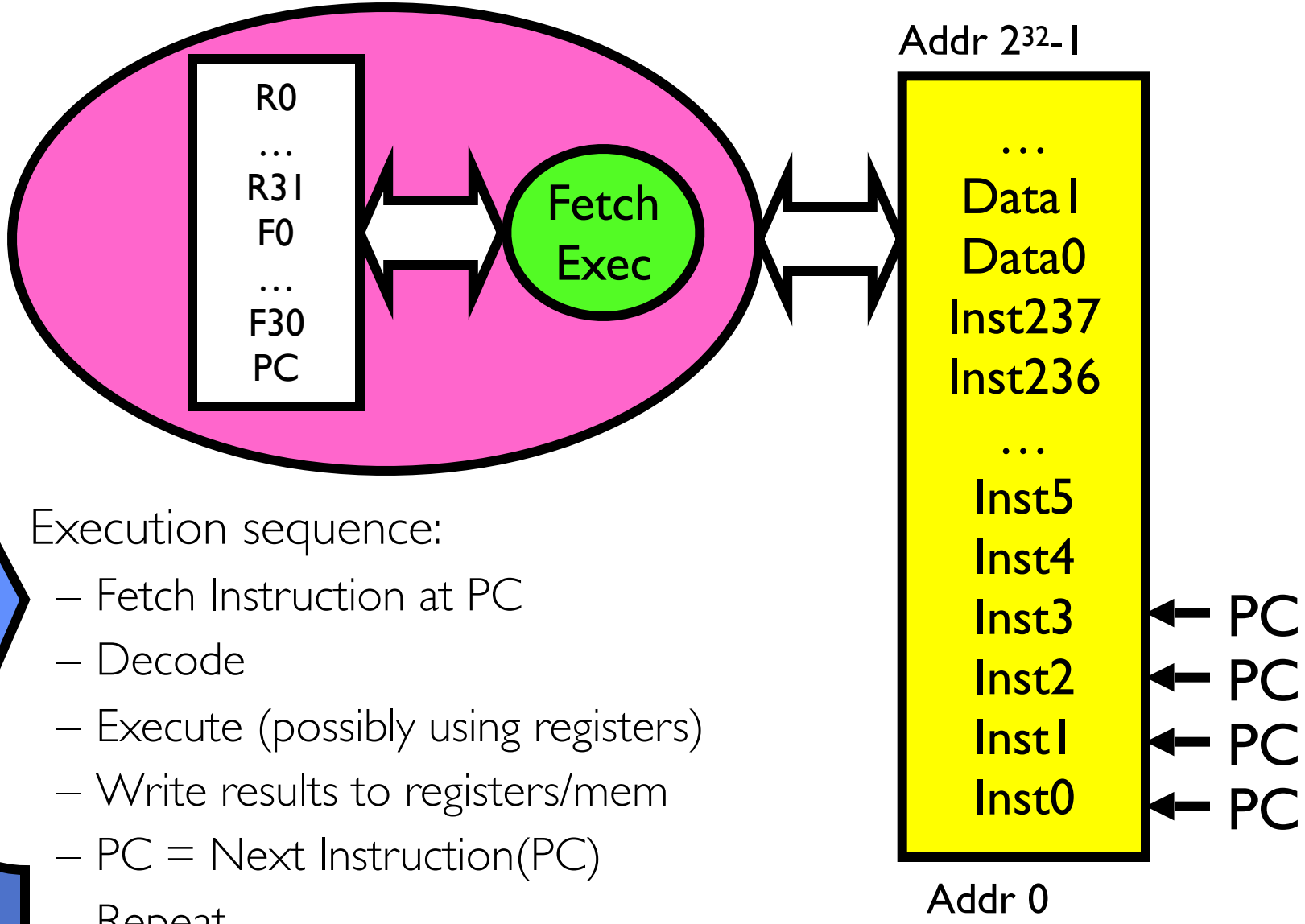
The instruction cycle



First OS Concept: Thread of Control

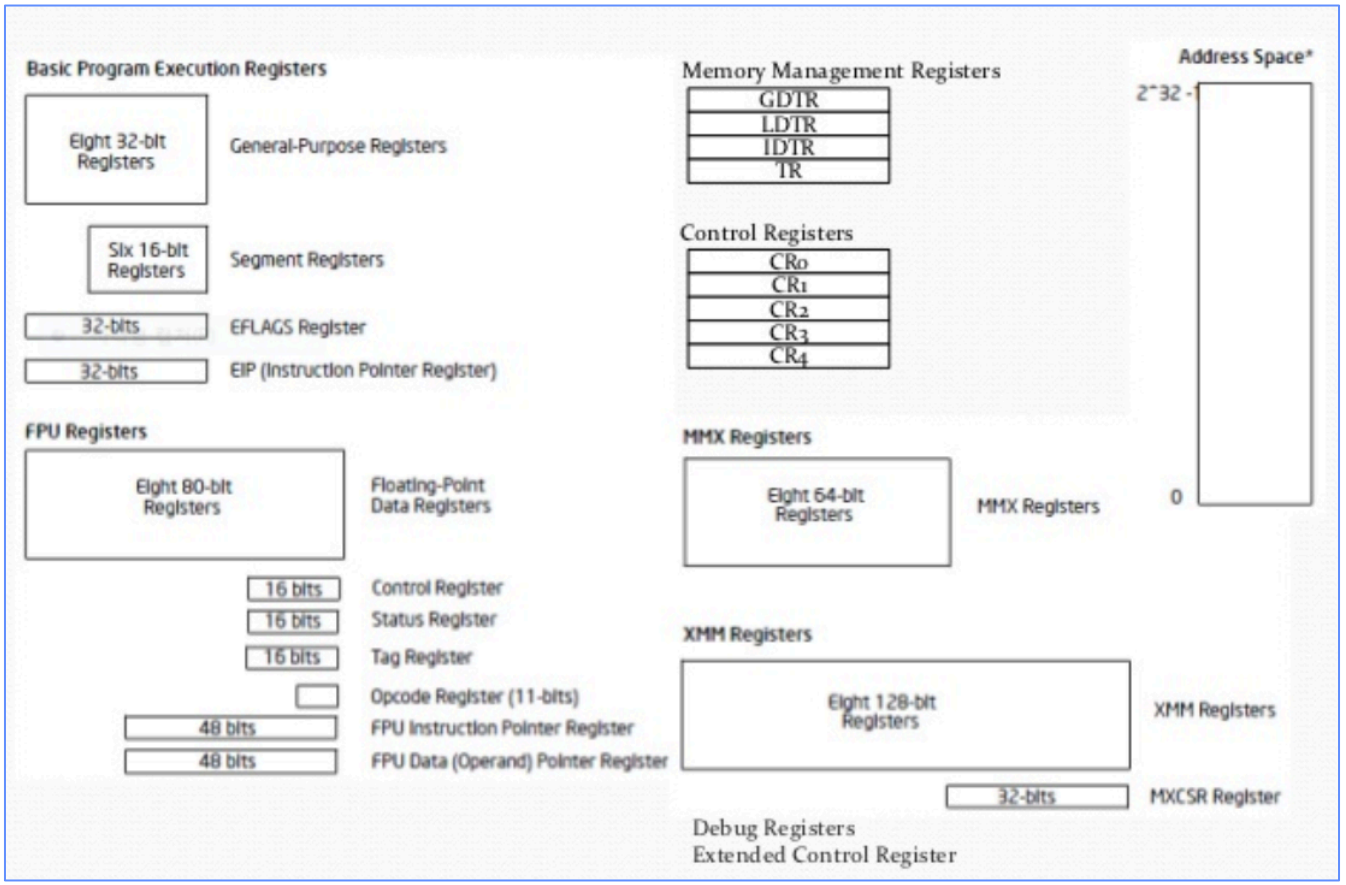
- Thread: Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack, Memory State
- A thread is *executing* on a processor (core) when it is *resident* in the processor registers
- Resident means: Registers hold the root state (context) of the thread:
  - Including program counter (PC) register & currently executing instruction
    - » PC points at next instruction in memory
    - » Instructions stored in memory
  - Including intermediate values for ongoing computations
    - » Can include actual values (like integers) or pointers to values in memory
  - Stack pointer holds the address of the top of stack (which is in memory)
  - The rest is "in memory"
- 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

Recall (61C): What happens during program execution?



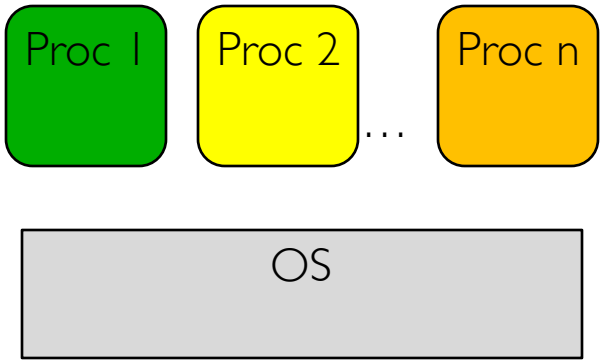
- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

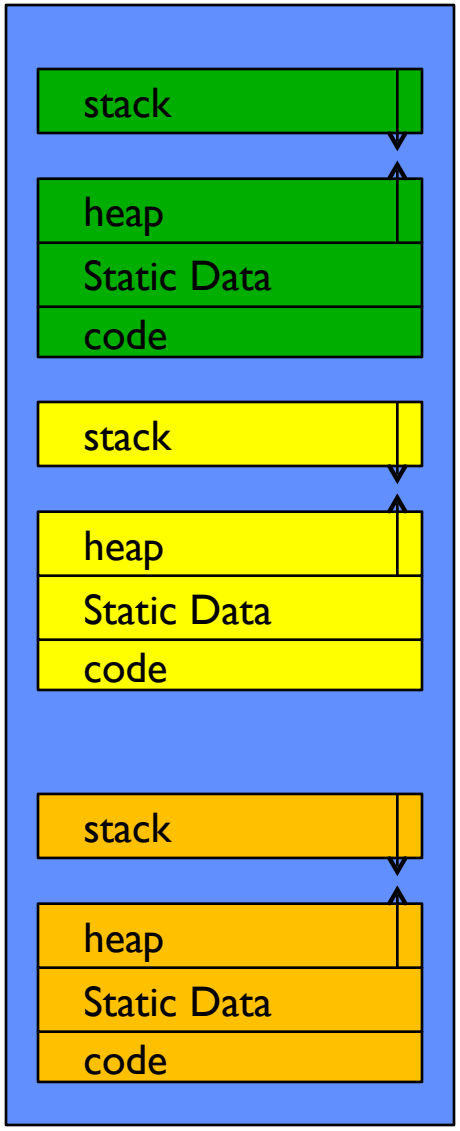
x86 Registers



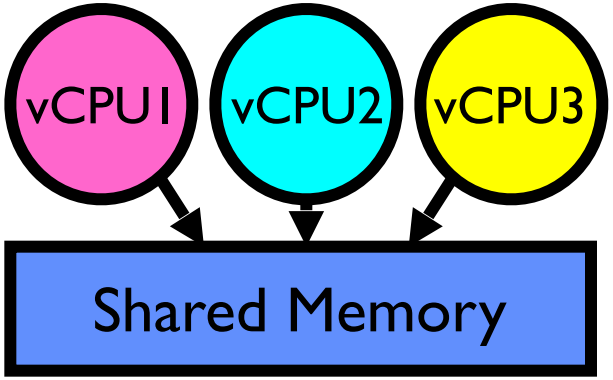
Complex mem-mem arch (x86) with specialized registers and "segments"

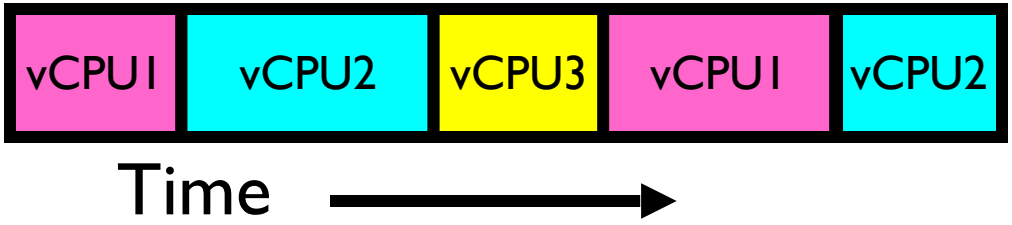
Multiprogramming - Multiple Threads of Control





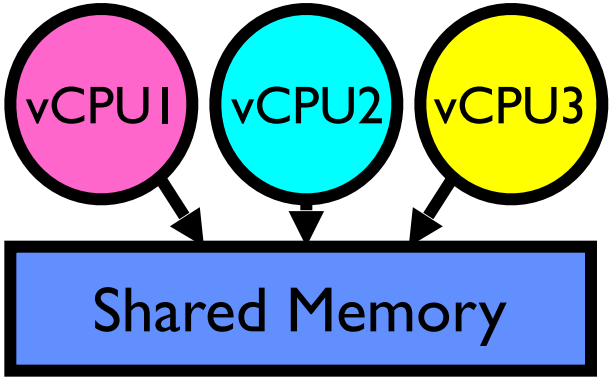
Illusion of Multiple Processors

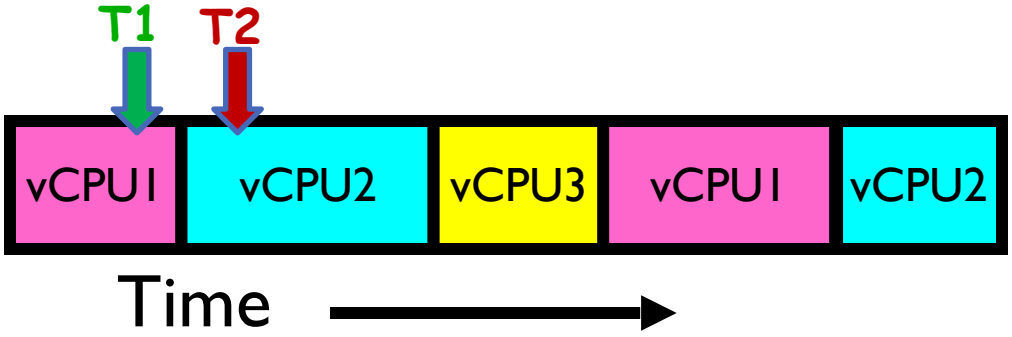




- Assume a single processor (core). How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Threads are *virtual cores*
- Contents of virtual core (thread):
  - Program counter, stack pointer
  - Registers
- Where is "it" (the thread)?
  - On the real (physical) core, or
  - Saved 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
- What happened?
  - OS Ran [how?]
  - Saved PC, SP, … in vCPU1's thread control block (memory)
  - Loaded PC, SP, … from vCPU2's TCB, jumped to PC
- What triggered this switch?
  - Timer, voluntary yield, I/O, other things we will discuss

OS object representing a thread?

- Traditional term: Thread Control Block (TCB)
- Holds contents of registers when thread is not running
- What other information?

- PINTOS? – read thread.h and thread.c

Administrivia: Getting started

- Start homework 0 immediately ⇒ Due next Monday (11/28)!
  - Vagrant and VirtualBox – VM environment for the course
    - » Consistent, managed environment on your machine
  - Get familiar with all the tools, submit via git
- TA Class
  - Saturdy 12:30 to 13:30, RM 204
    - Will announce when there is a class
- Any questions on class rules and regulations?
- Midterm Date:
  - Any issues?

CE424 Collaboration Policy

Explaining a concept to someone in another group

Discussing algorithms/testing strategies with other groups

Helping debug someone else's code (in another group)

Searching online for generic algorithms (e.g., hash table)

Sharing code or test cases with another group

Copying OR reading another group's code or test cases

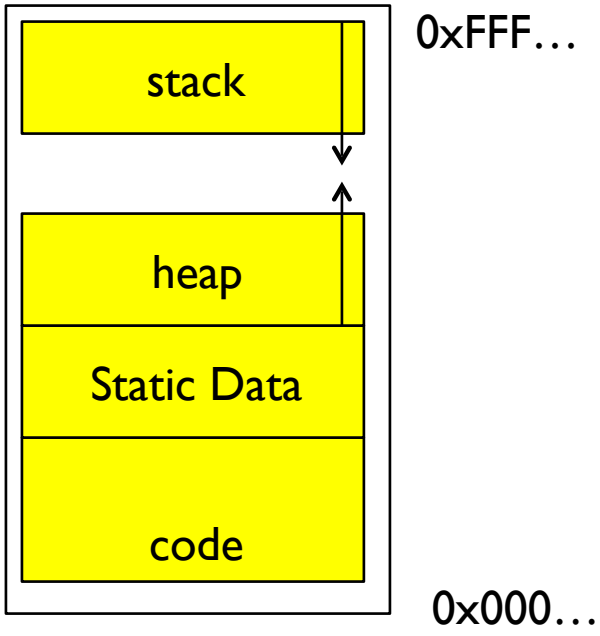
Copying OR reading online code or test cases from from prior years

We compare all project submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders

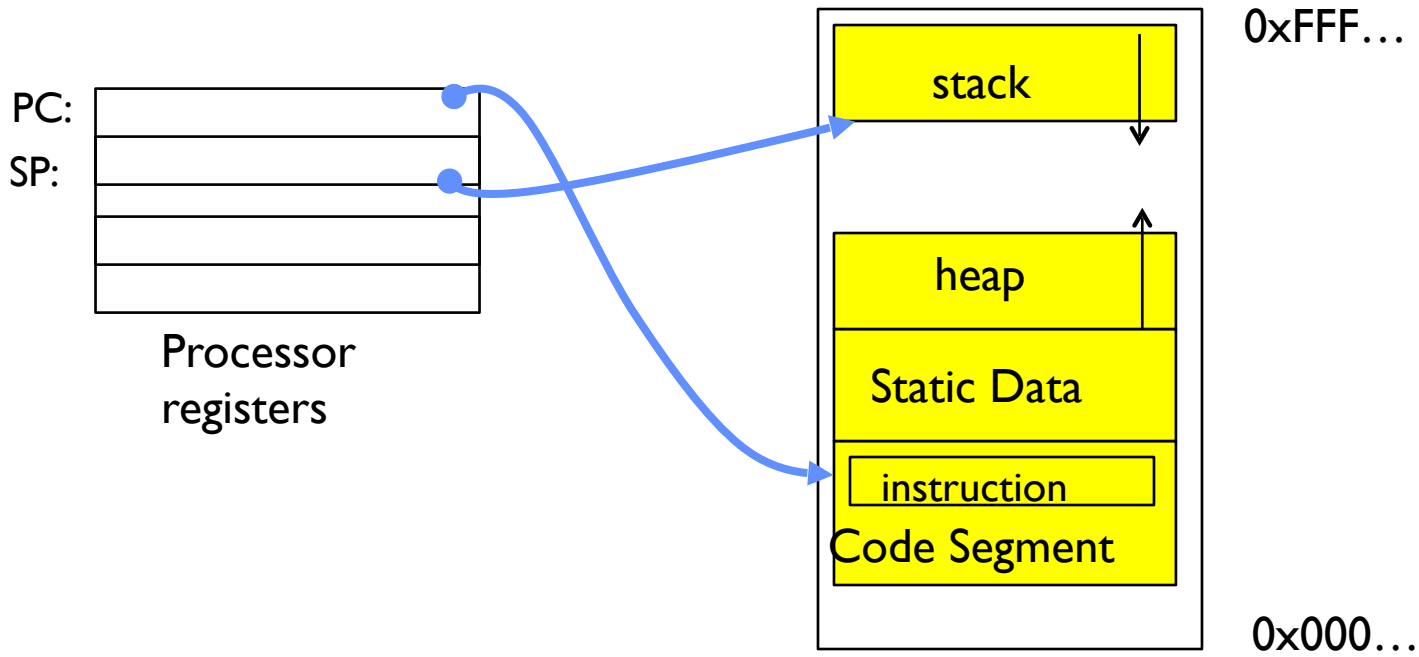
Second OS Concept: Address Space

- 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 acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - » (Memory-mapped I/O)
  - Perhaps causes exception (fault)
  - Communicates with another program
  - ….



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?

Previous discussion of threads: Very Simple Multiprogramming

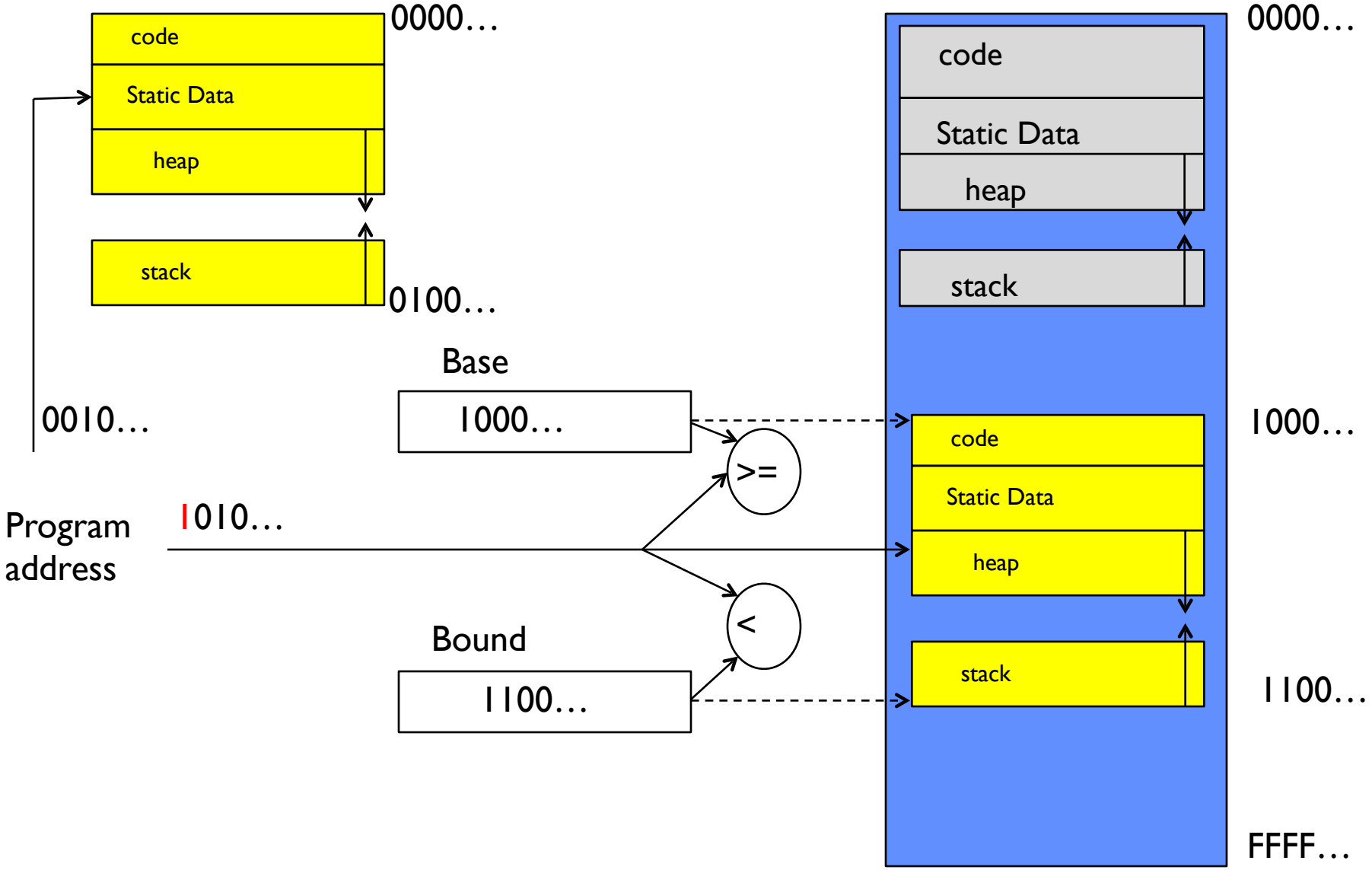
- All vCPU's share non-CPU resources
  - Memory, I/O Devices
- Each thread can read/write memory
  - Perhaps data of others
  - can overwrite OS ?
- Unusable?
- This approach is used in
  - Very early days of computing
  - Embedded applications
  - MacOS 1-9/Windows 3.1 (switch only with voluntary yield)
  - Windows 95-ME (switch with yield or timer)
- However it is risky…

Simple Multiplexing has no Protection

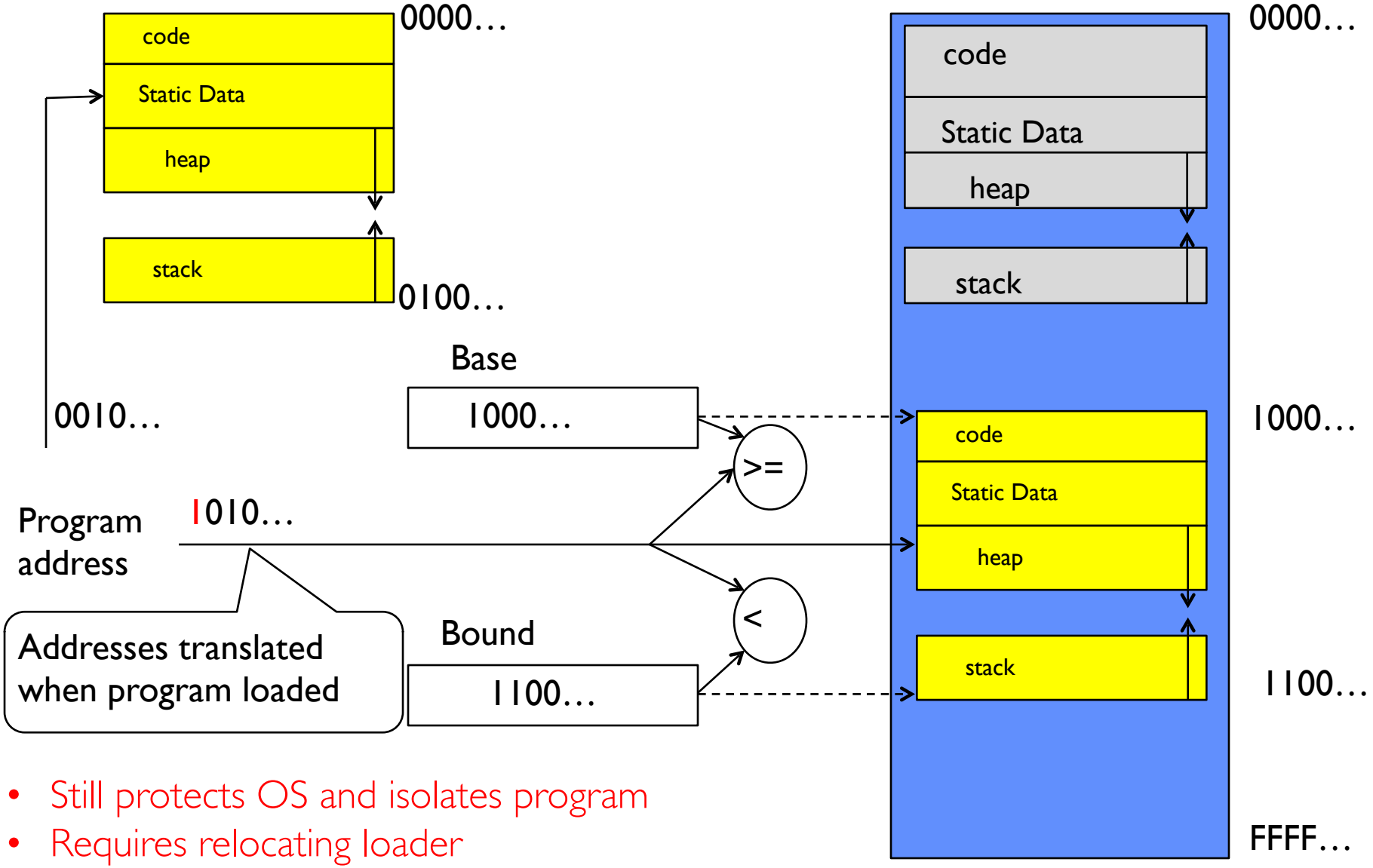
- Operating System must protect itself from user programs
  - Reliability: compromising the operating system generally causes it to crash
  - Security: limit the scope of what threads can do
  - Privacy: limit each thread to the data it is permitted to access
  - Fairness: each thread should be limited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

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

Simple Protection: Base and Bound (B&B)

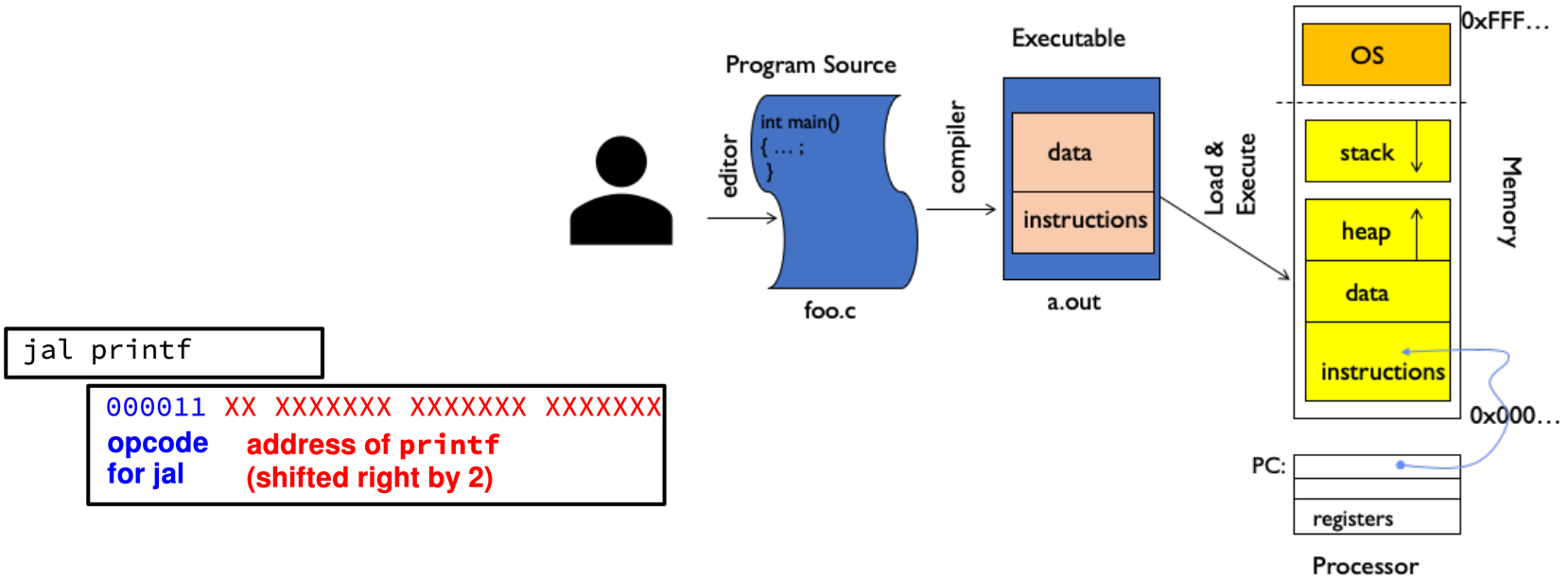


Simple Protection: Base and Bound (B&B)



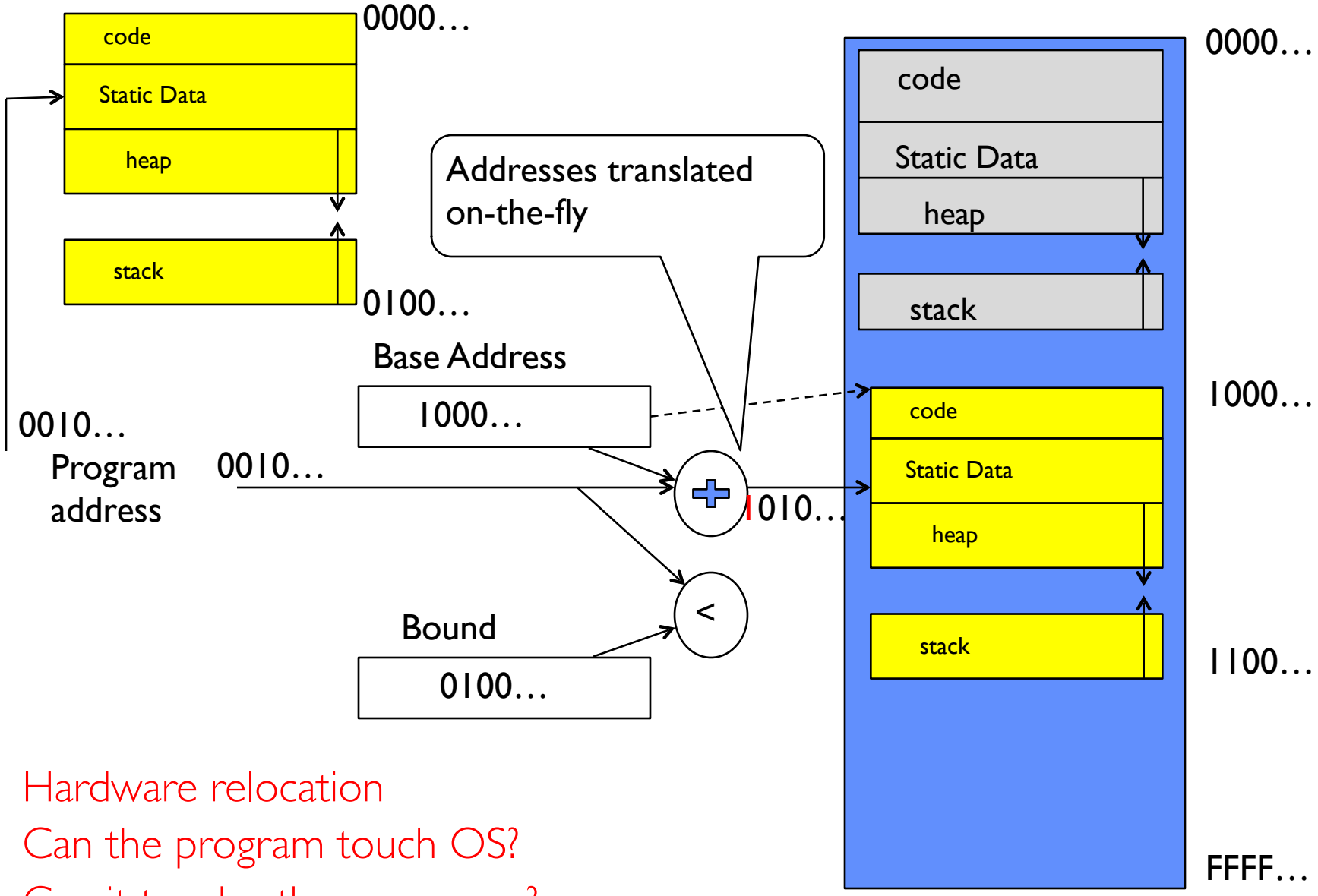
- Still protects OS and isolates program
- Requires relocating loader
- No addition on address path

61C Review: Relocation



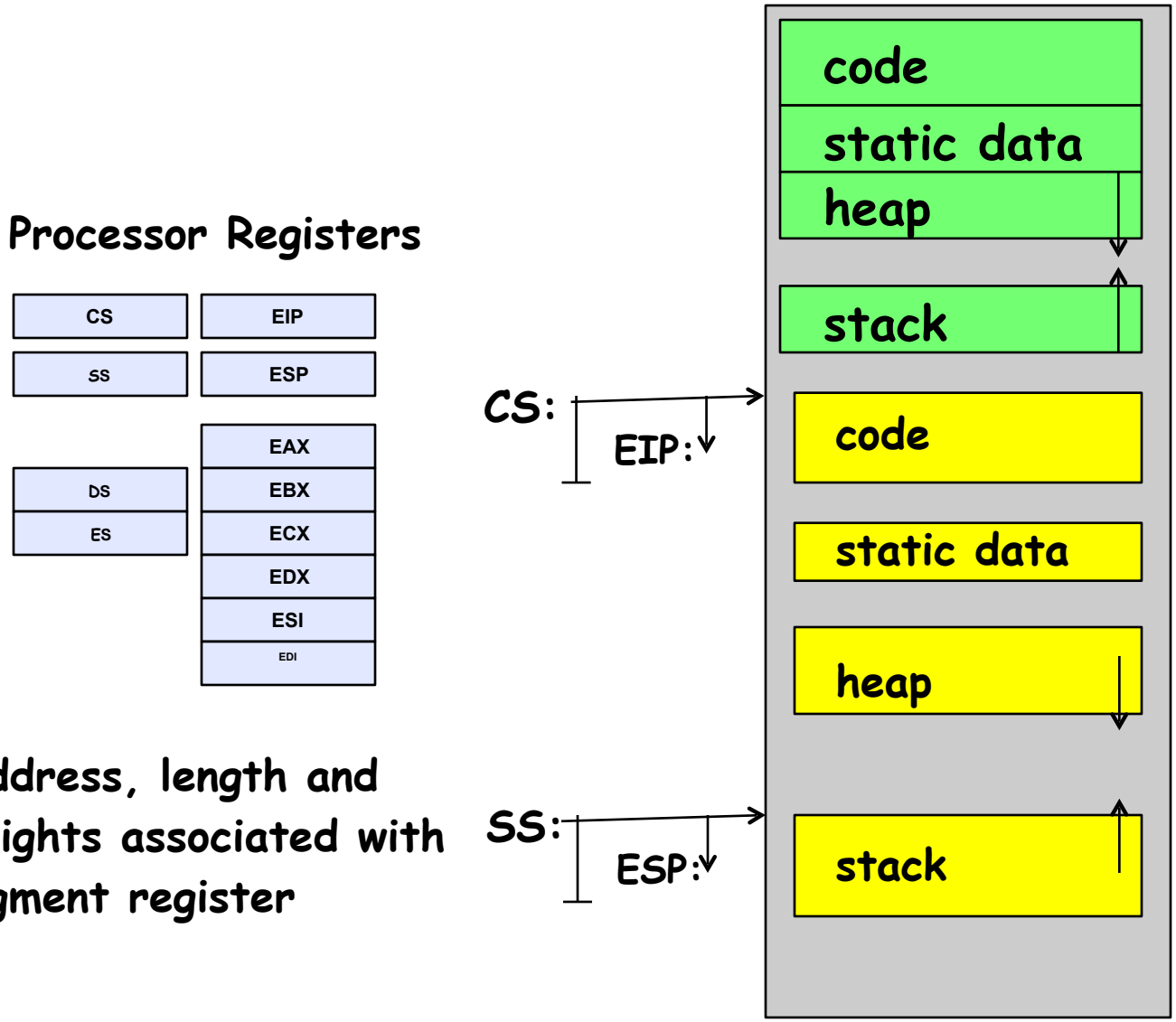
- Compiled .obj file linked together in an .exe
- All address in the .exe are as if it were loaded at memory address 00000000
- File contains a list of all the addresses that need to be adjusted when it is "relocated" to somewhere else.

Simple address translation with Base and Bound



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

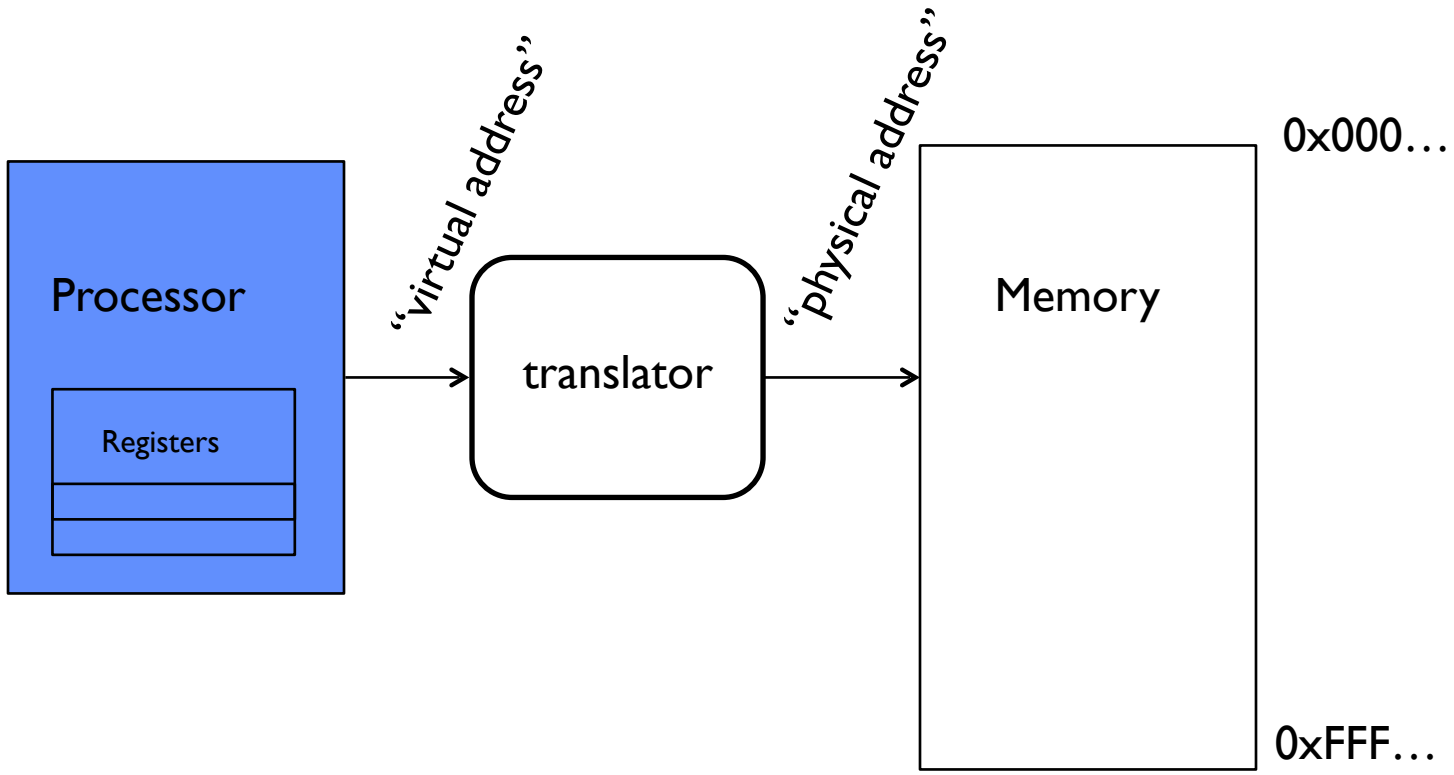
x86 – segments and stacks



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

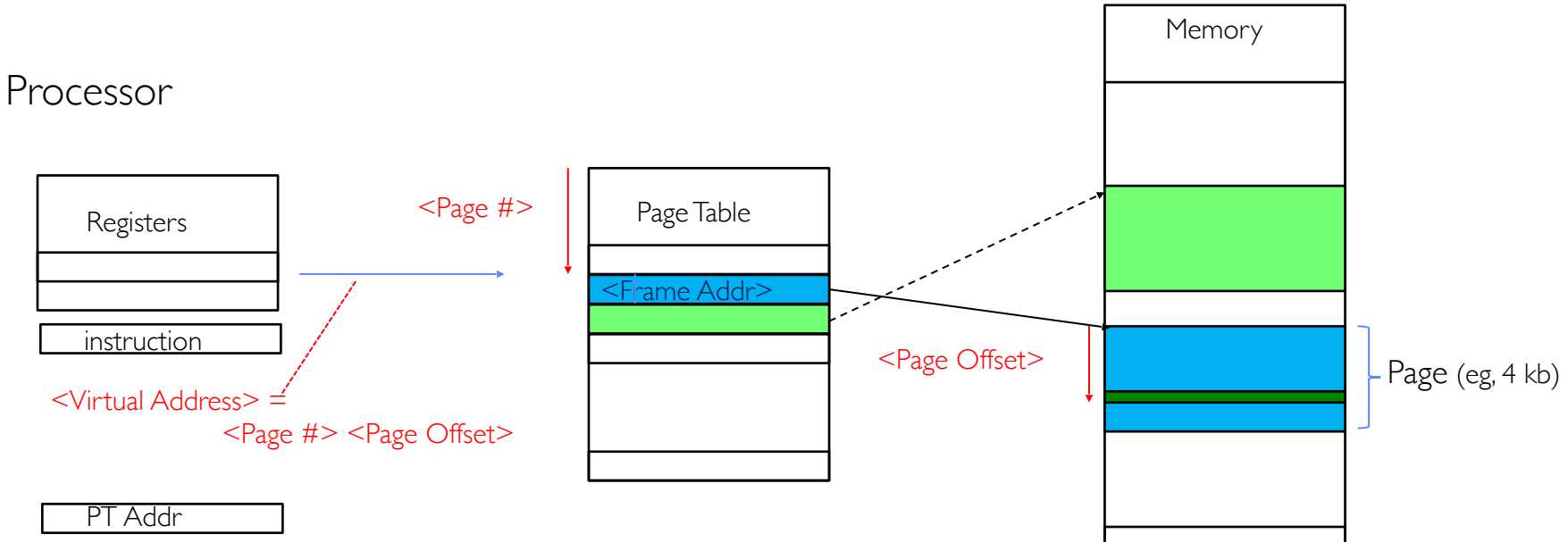


Paged Virtual Address Space

- What if we break the entire virtual address space into equal size chunks (i.e., pages) have a base for each?
- Treat memory as page size frames and put any page into any frame …

- Another cs61C review…

Paged Virtual Address

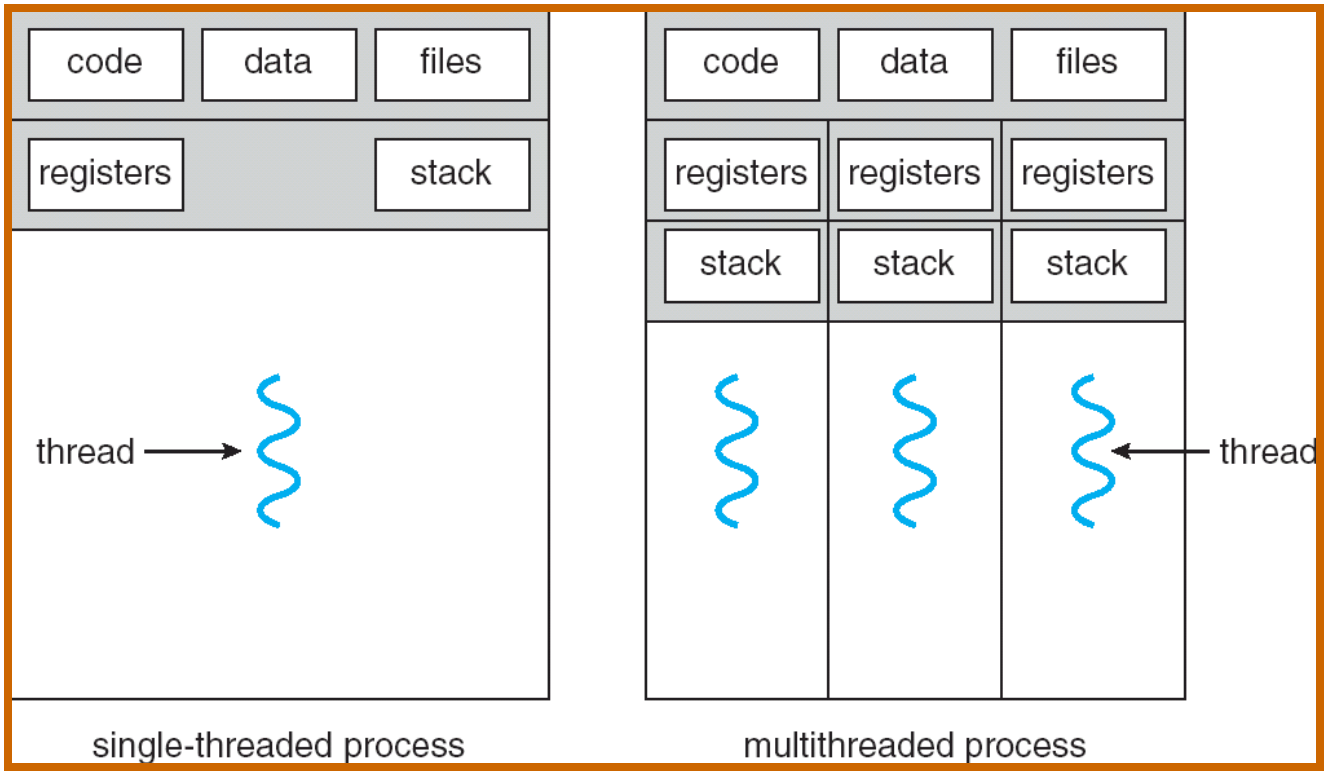


- Instructions operate on virtual addresses
  - Instruction address, load/store data address
- Translated to a physical address (or Page Fault) through a Page Table by the hardware
- Any Page of address space can be in any (page sized) frame in memory
  - Or not-present (access generates a page fault)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

Third OS Concept: Process

- **Process:** execution environment with Restricted Rights
  - **(Protected) 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
- Application program executes as a process
  - Complex applications can fork/exec child processes [later!]
- Why **processes**?
  - Protected from each other!
  - OS Protected from them
  - Processes provides memory protection
  - Threads more efficient than processes for parallelism (later)
- Fundamental tradeoff between protection and efficiency
  - Communication easier *within* a process
  - Communication harder *between* processes

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?

Kernel code/data in process Virtual Address Space?

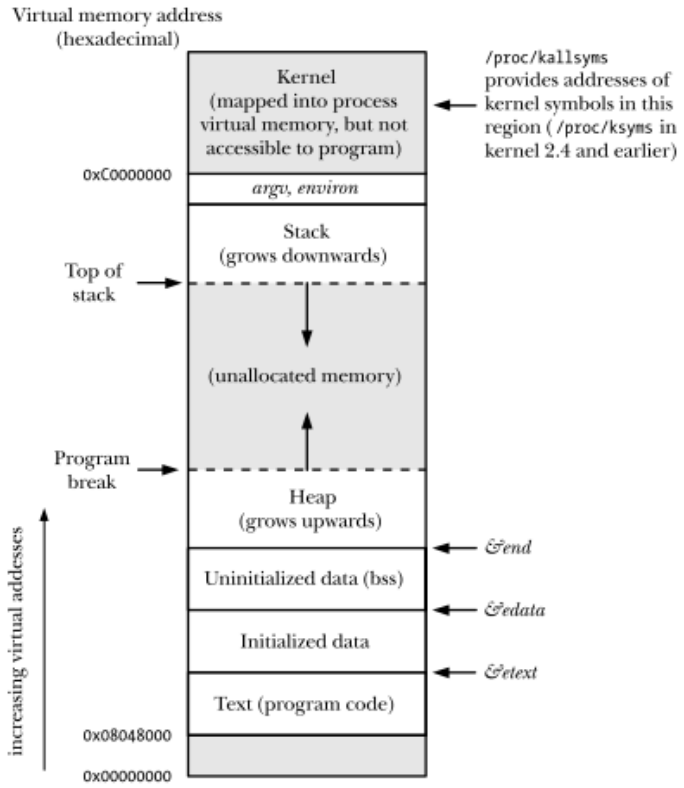


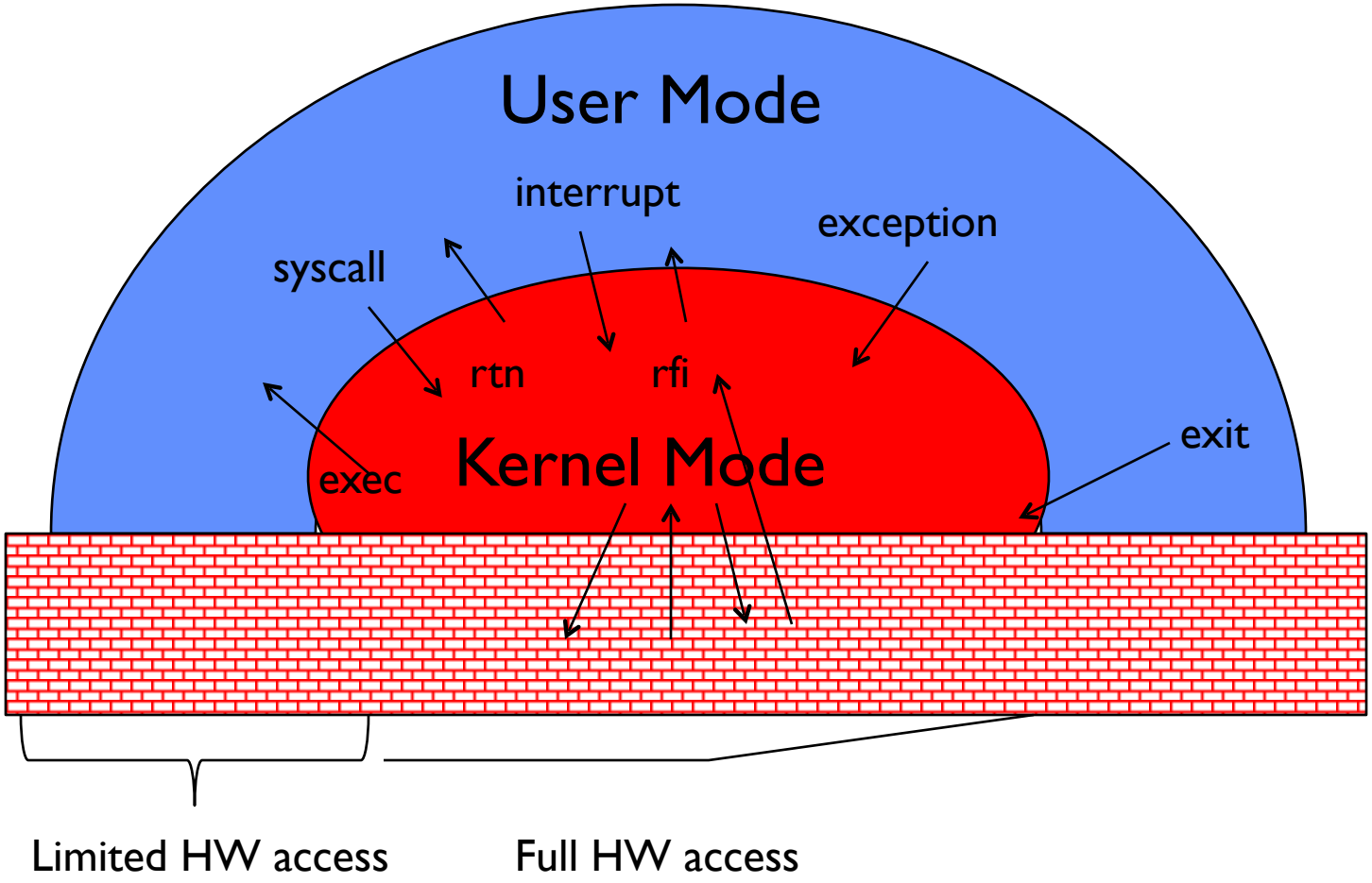
Figure 6-1: Typical memory layout of a process on Linux/x86-32

- Unix: Kernel space is mapped in high - but inaccessible to user processes

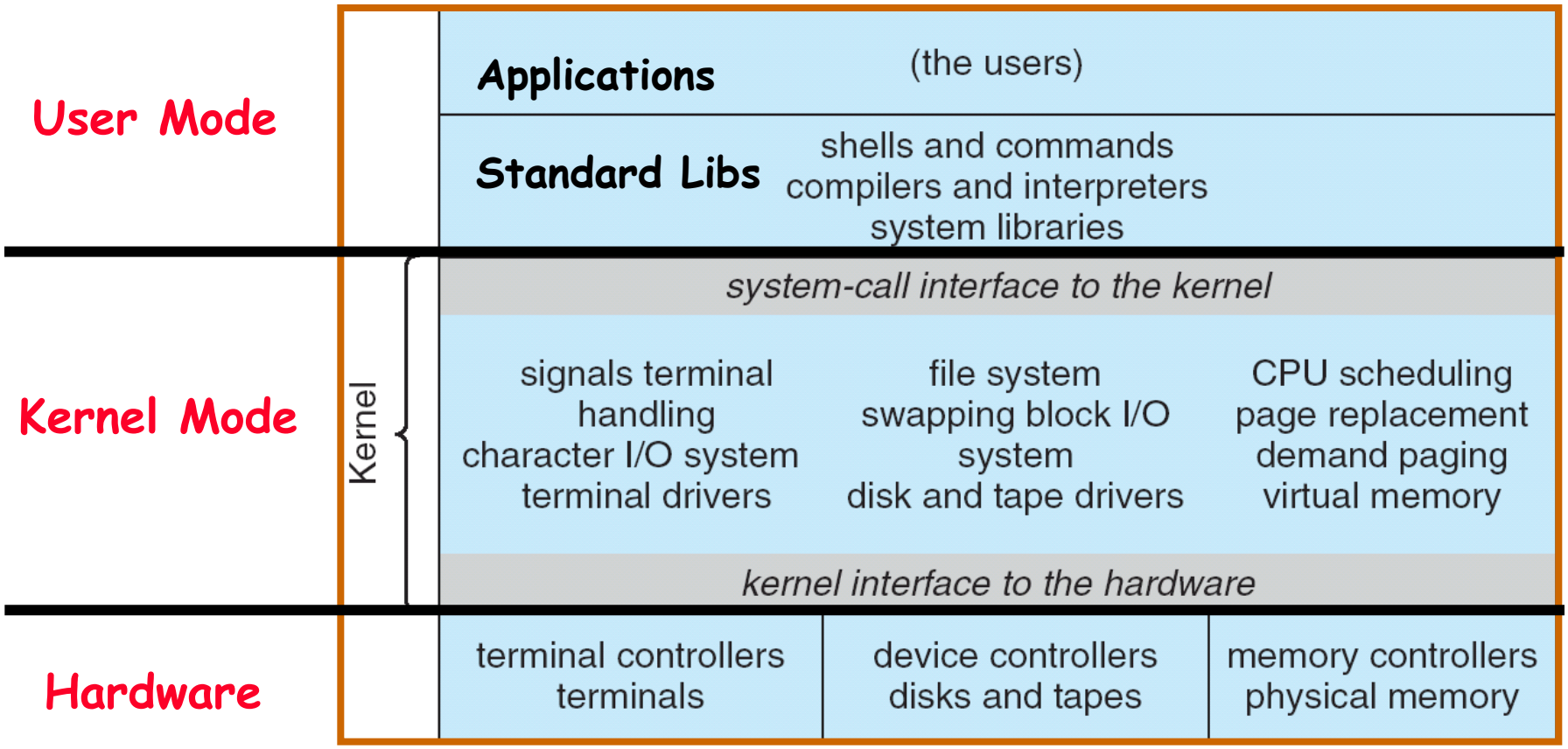
Fourth OS Concept: Dual Mode Operation

- 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

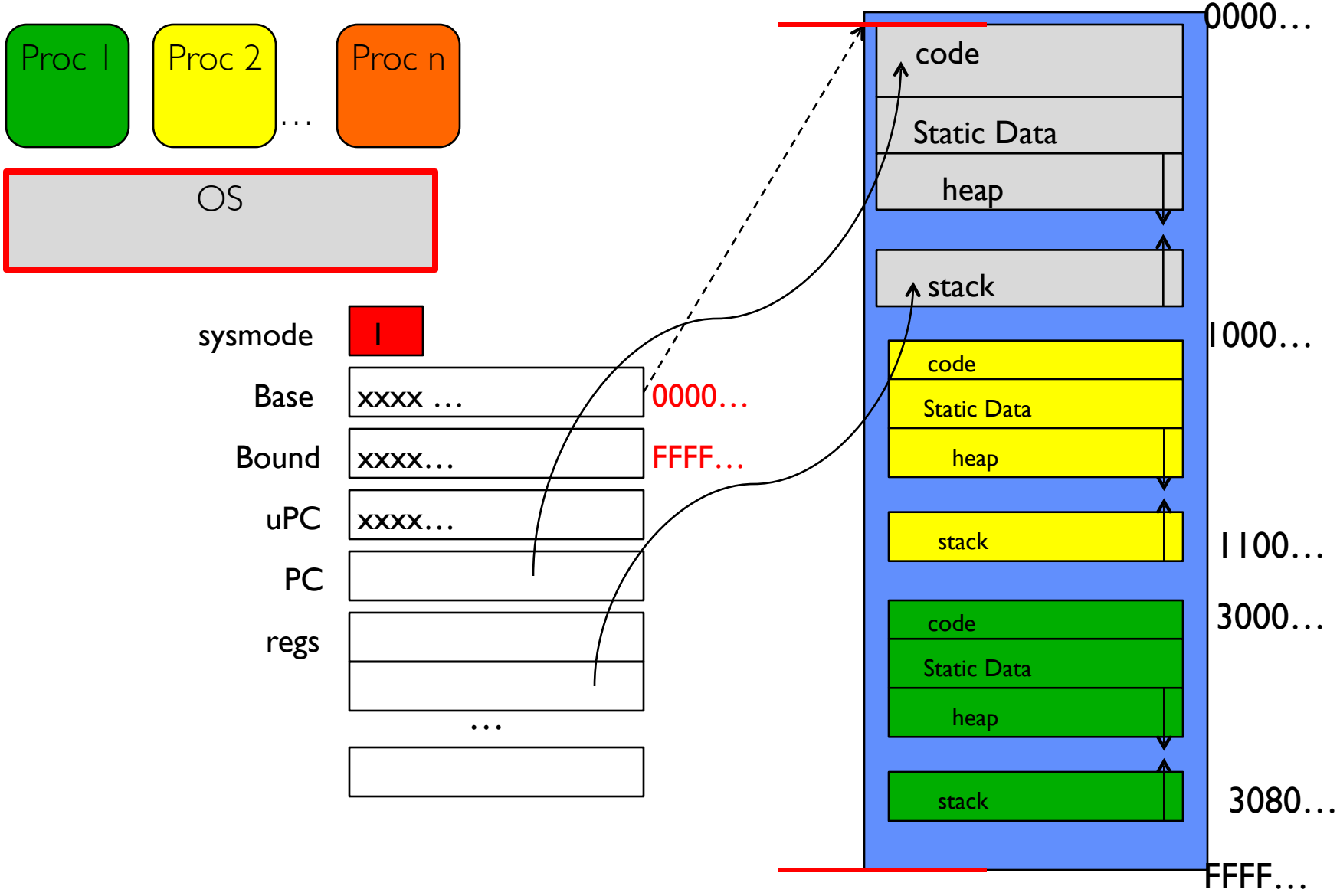


For example: UNIX System Structure

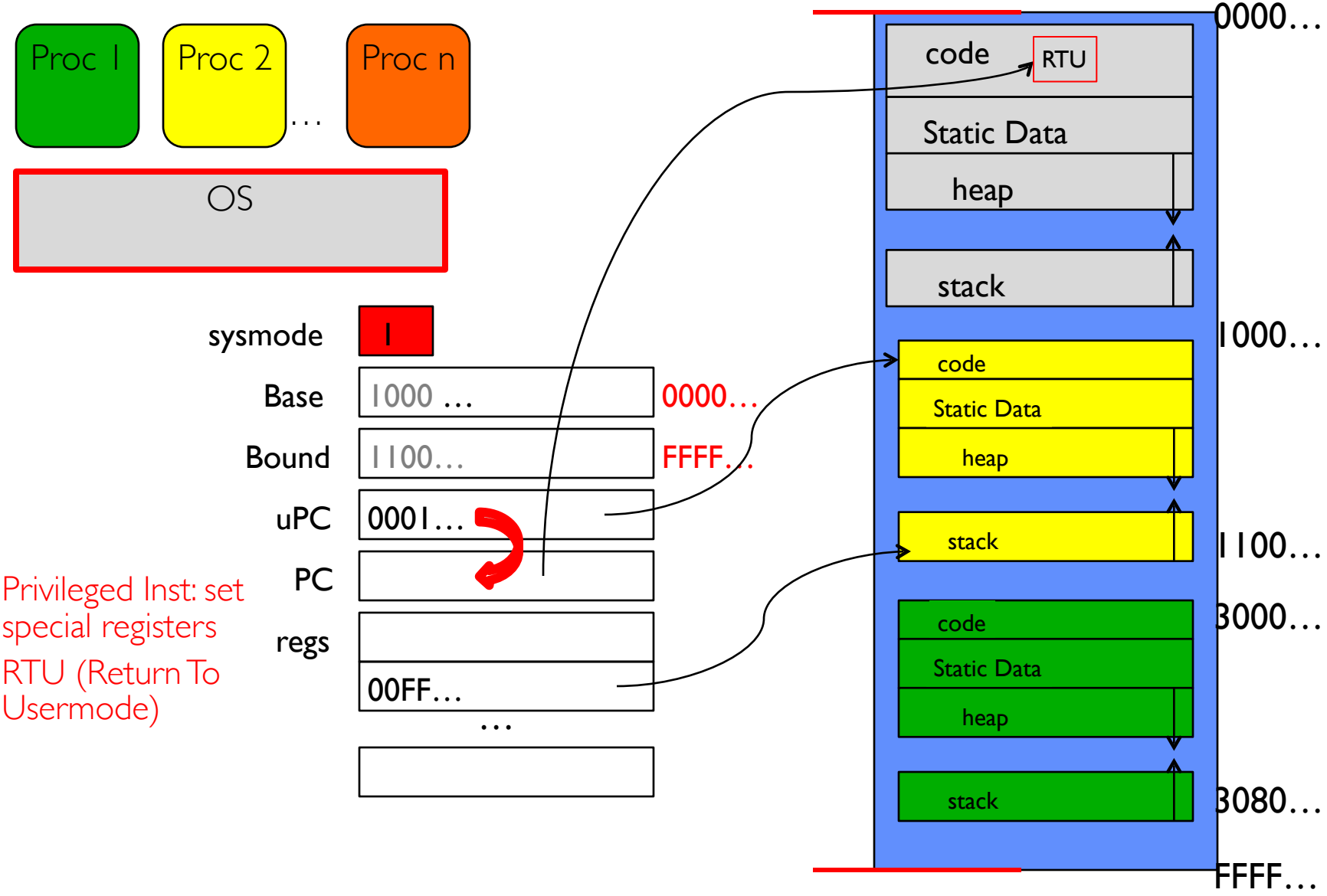


Break!

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

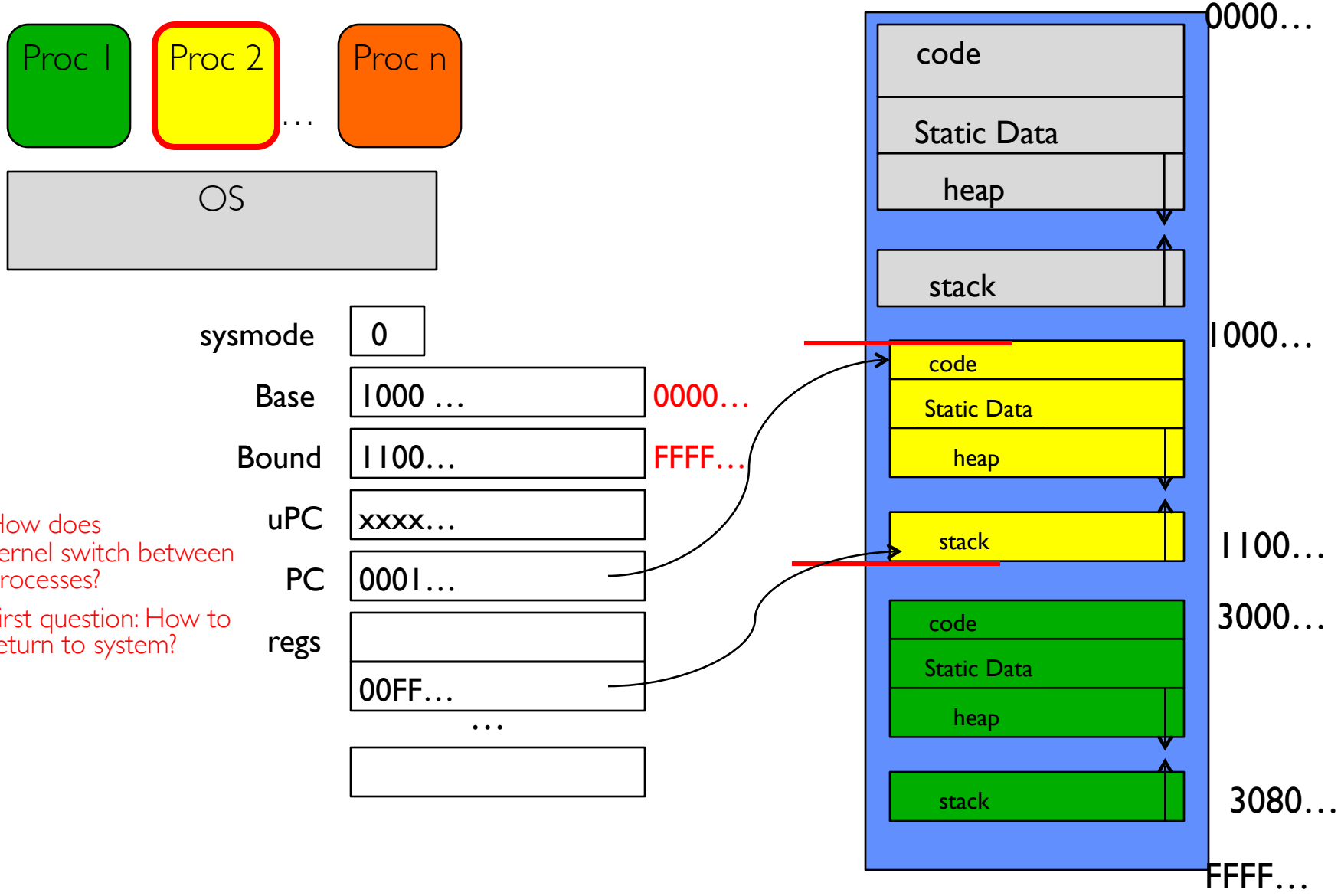


Simple B&B: OS gets ready to execute process



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

Simple B&B: User Code Running



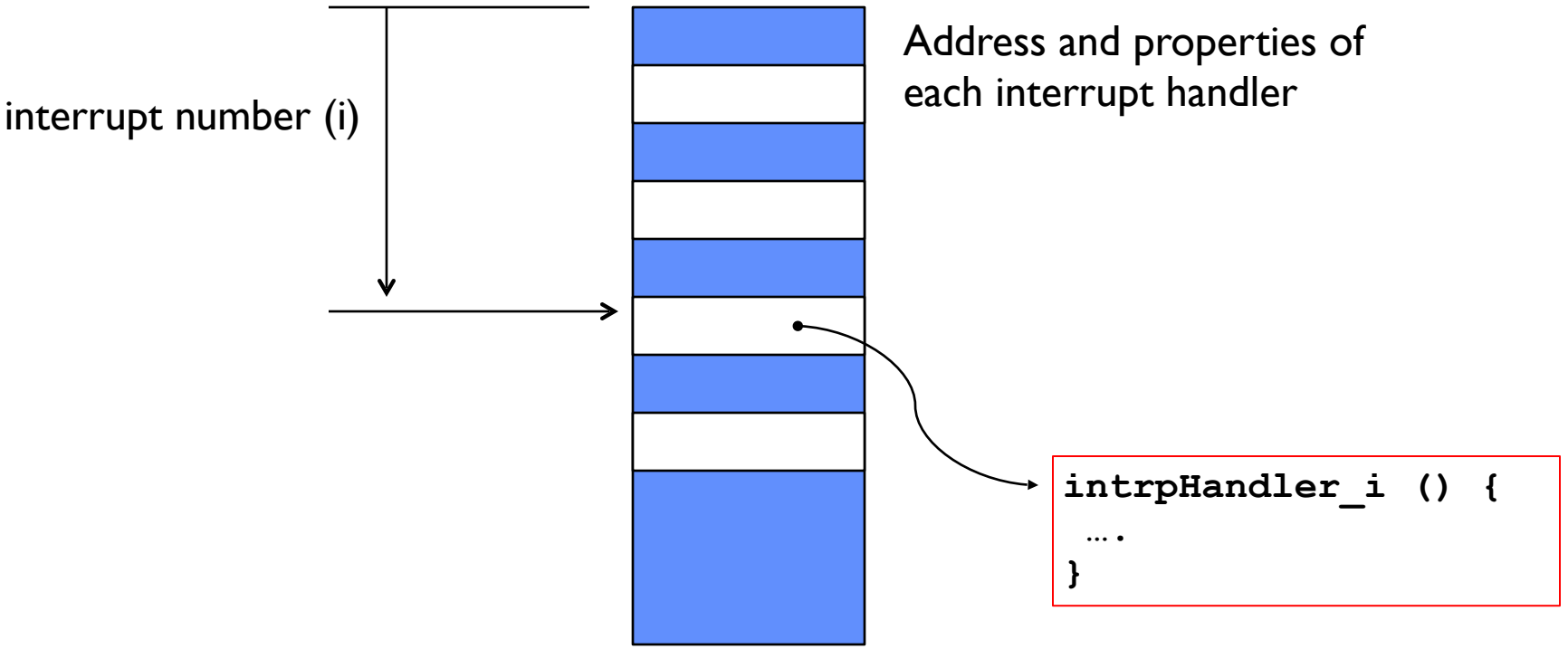
- How does kernel switch between processes?
- First question: How to return to system?

3 types of Mode Transfer

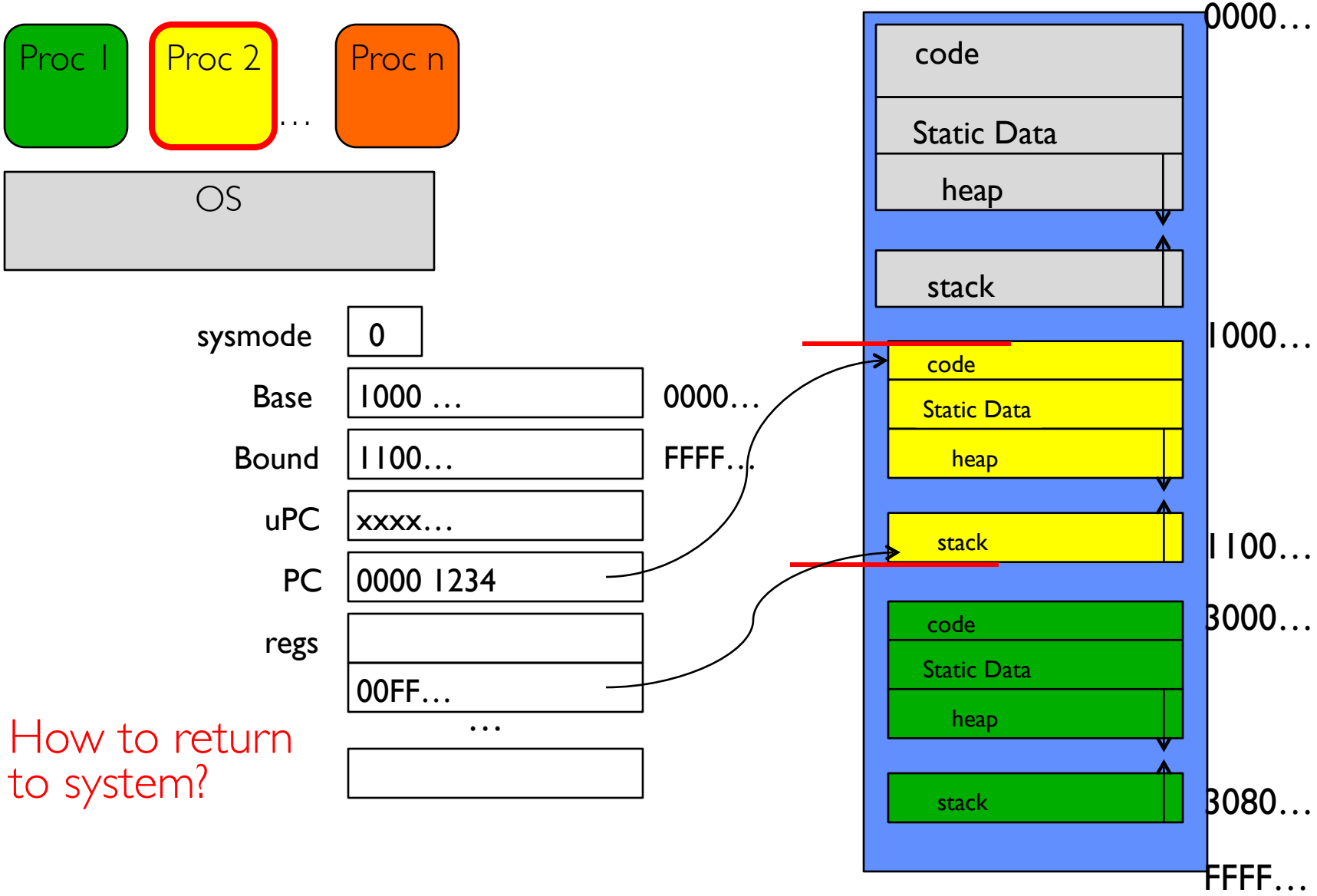
- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but "outside" the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt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

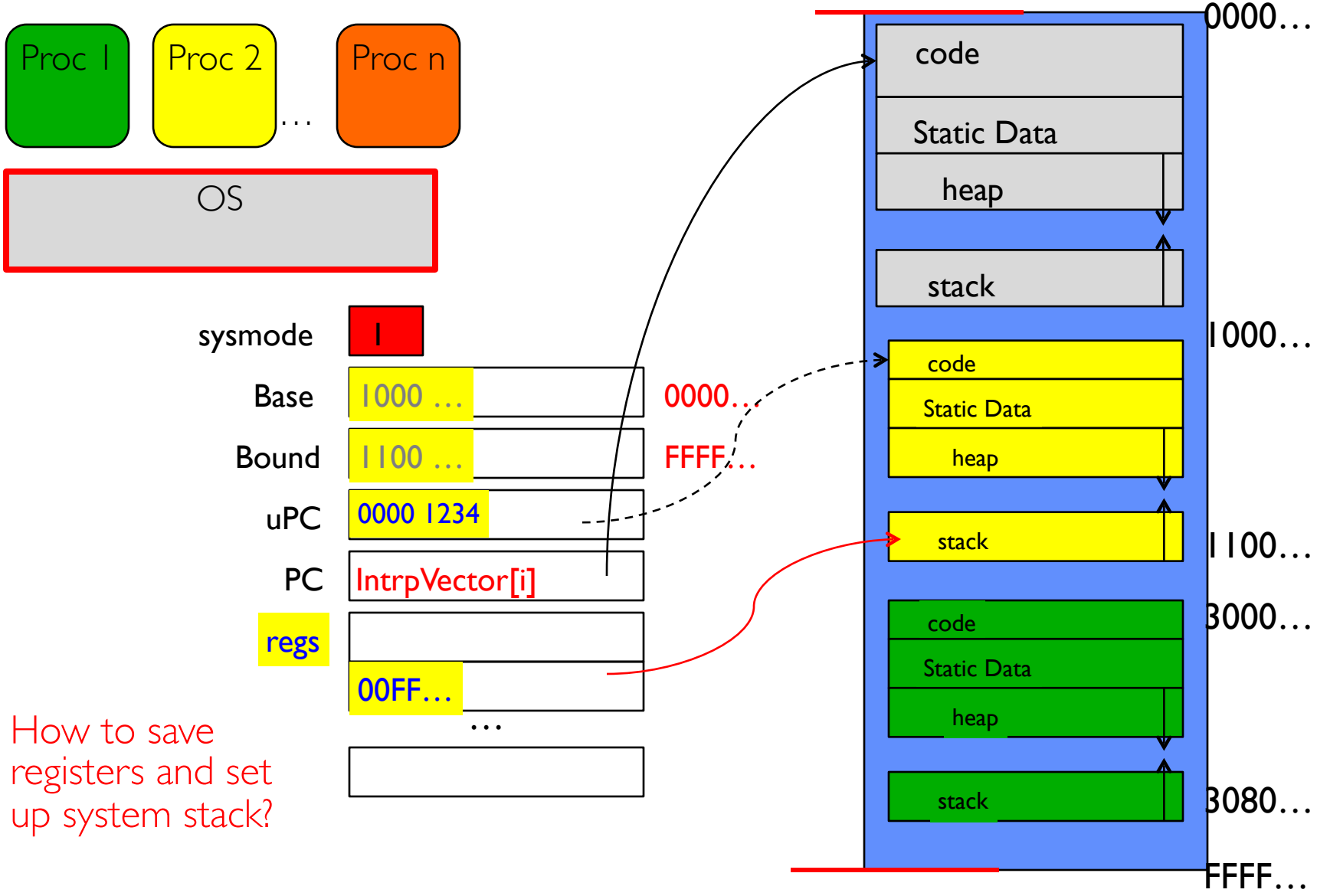


Simple B&B: User => Kernel



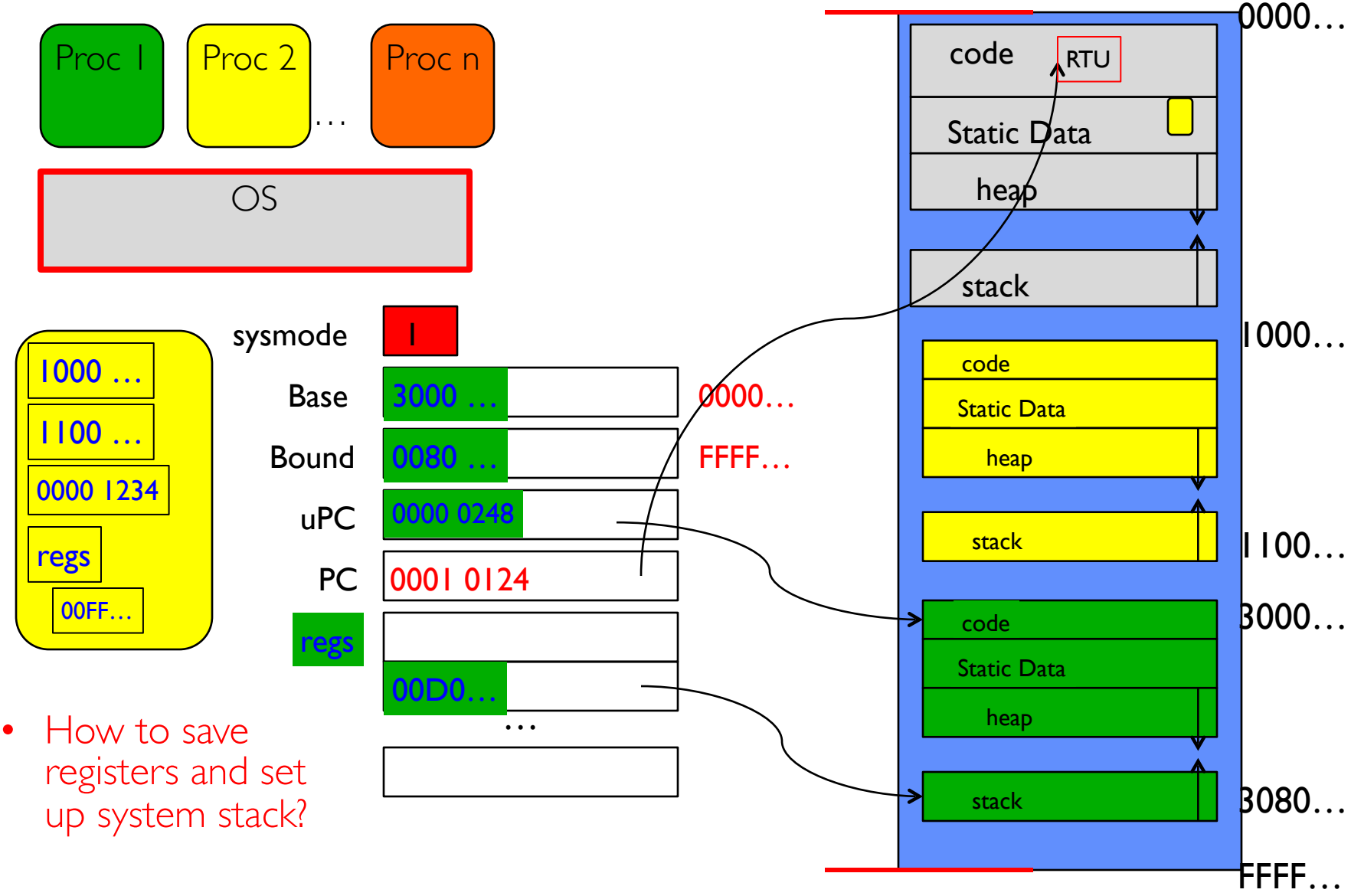
- How to return to system?

Simple B&B: Interrupt



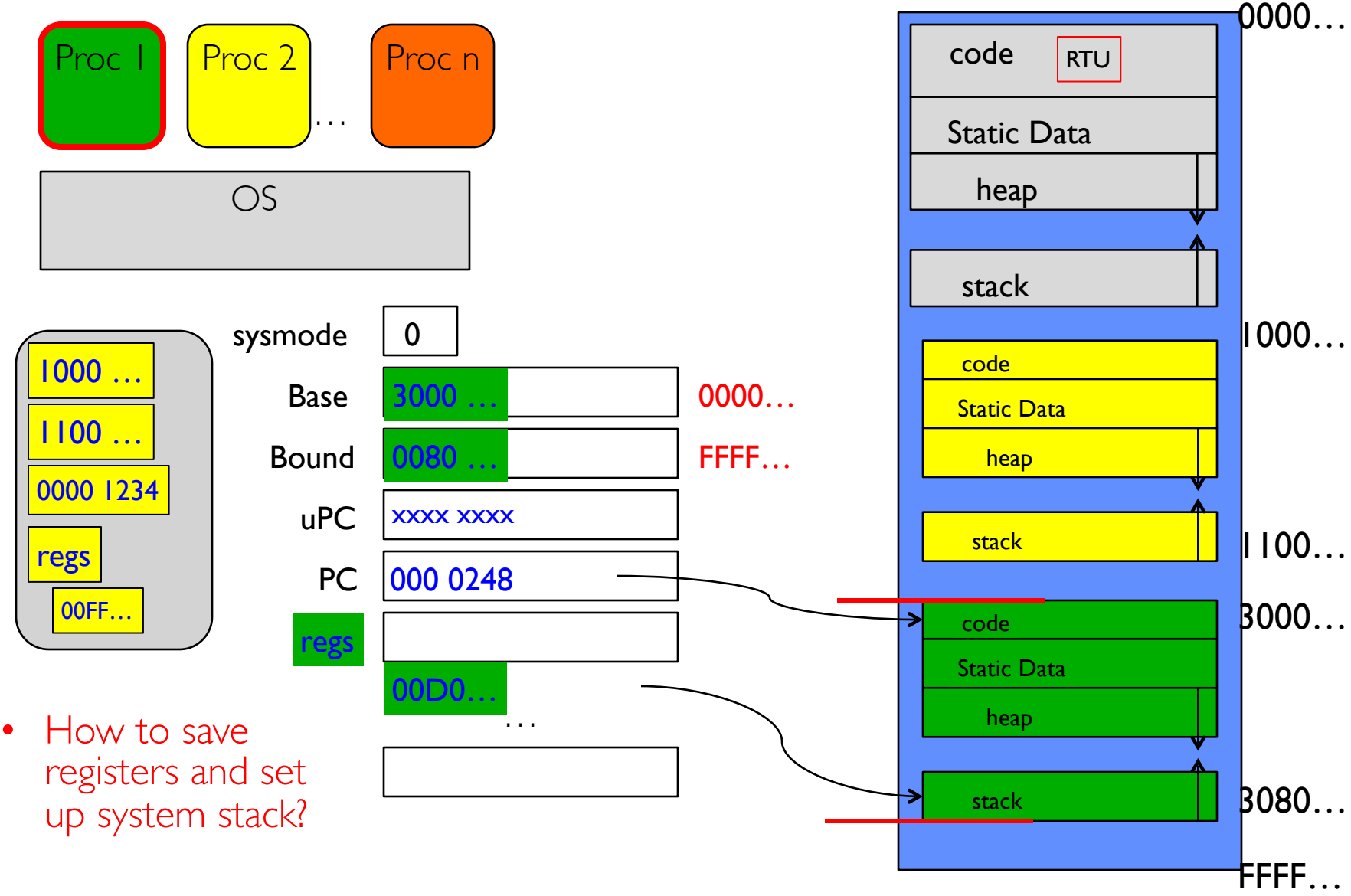
- How to save registers and set up system stack?

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?

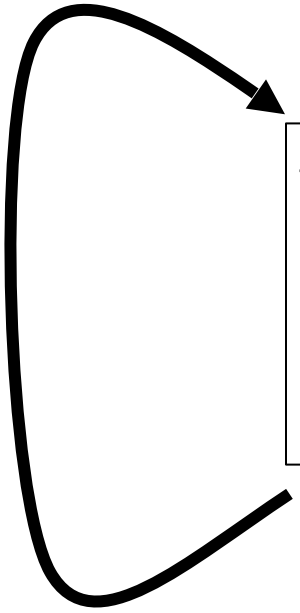
Running Many Programs ???

- 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
- Questions ???
- 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?
- Aren't we wasting are lot of memory?
- …

Process Control Block

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel Scheduler maintains a data structure containing the PCBs
- Scheduling algorithm selects the next one to run

Scheduler



```
if ( readyProcesses(PCBs) ) {
        nextPCB = selectProcess(PCBs);
        run( nextPCB );
} else {
        run_idle_process();
}
```

Conclusion: Four Fundamental OS Concepts

- Thread: Execution Context
  - Fully describes program state
  - Program Counter, Registers, Execution Flags, Stack
- Address space (with or w/o translation)
  - Set of memory addresses accessible to program (for read or write)
  - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- 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
  - Combined with translation, isolates programs from each other and the OS from programs