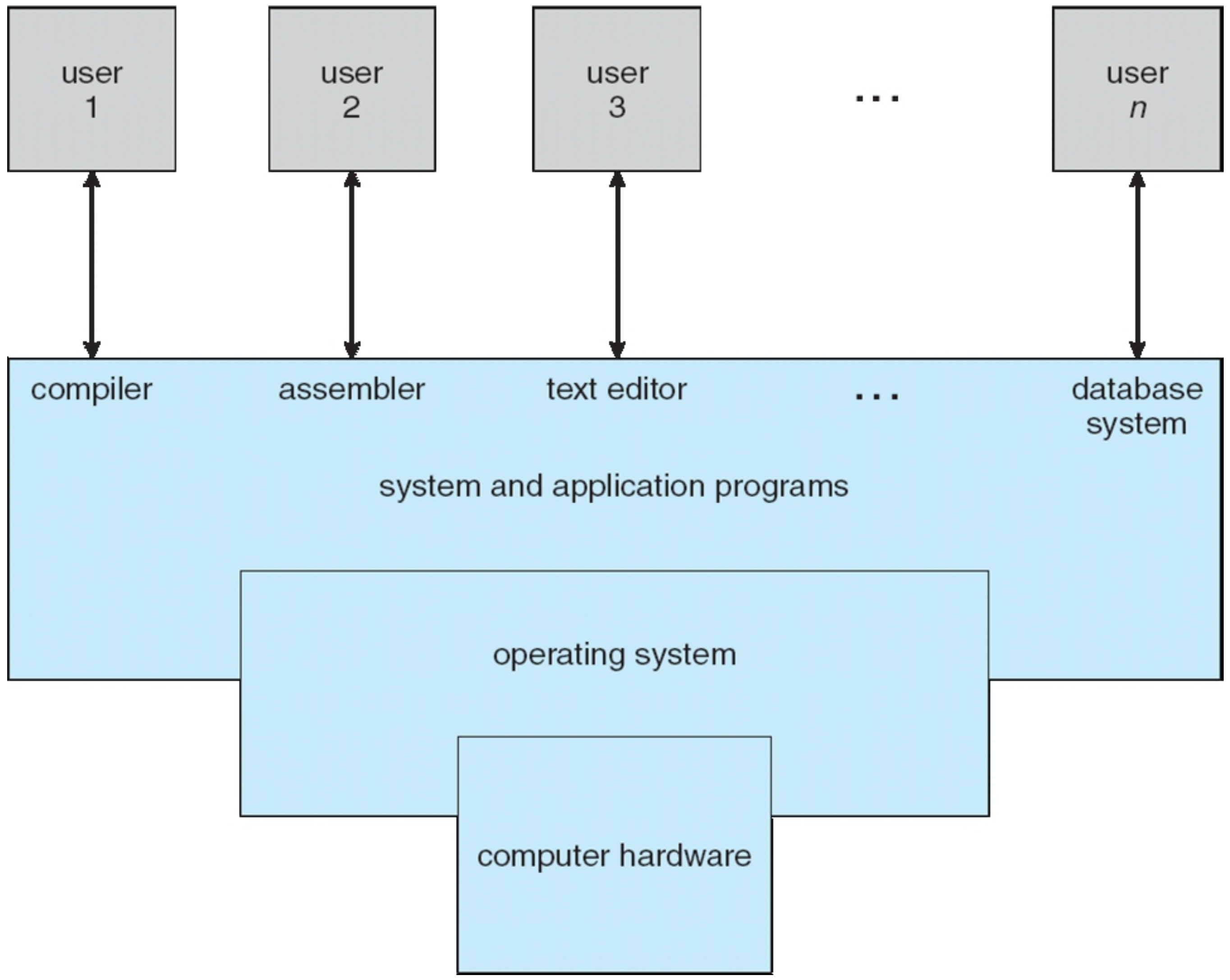
Lecture 2
Fundamental OS Concepts

Yinqian Zhang@ 2021, Spring

Our Roadmap

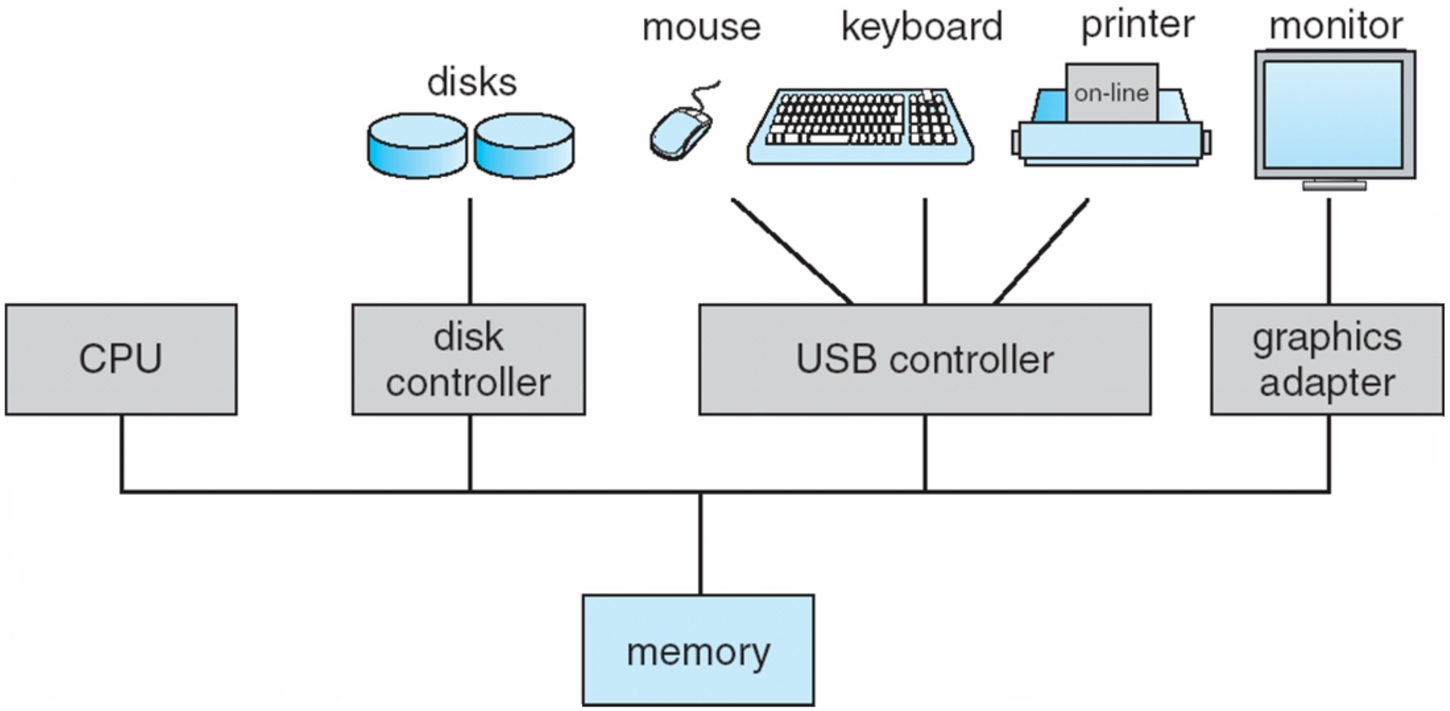
- Computer organization revision
- Kernel data structures in OS
- OS history
- Four fundamental OS concepts
  - Thread
  - Address space (with translation)
  - Process
  - Dual mode operation / protection

Four Components of a Computer System

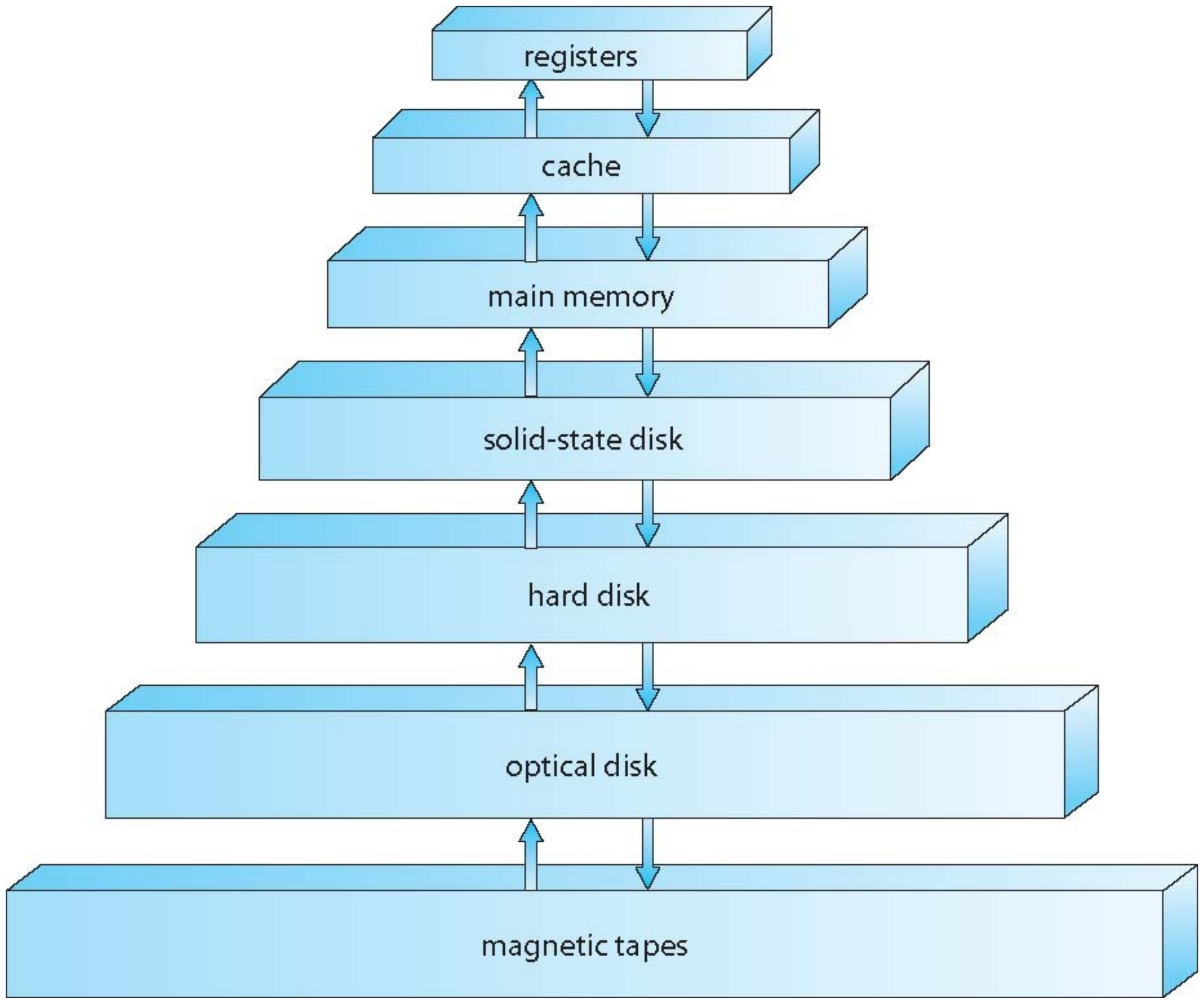


Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Storage Device Hierarchy

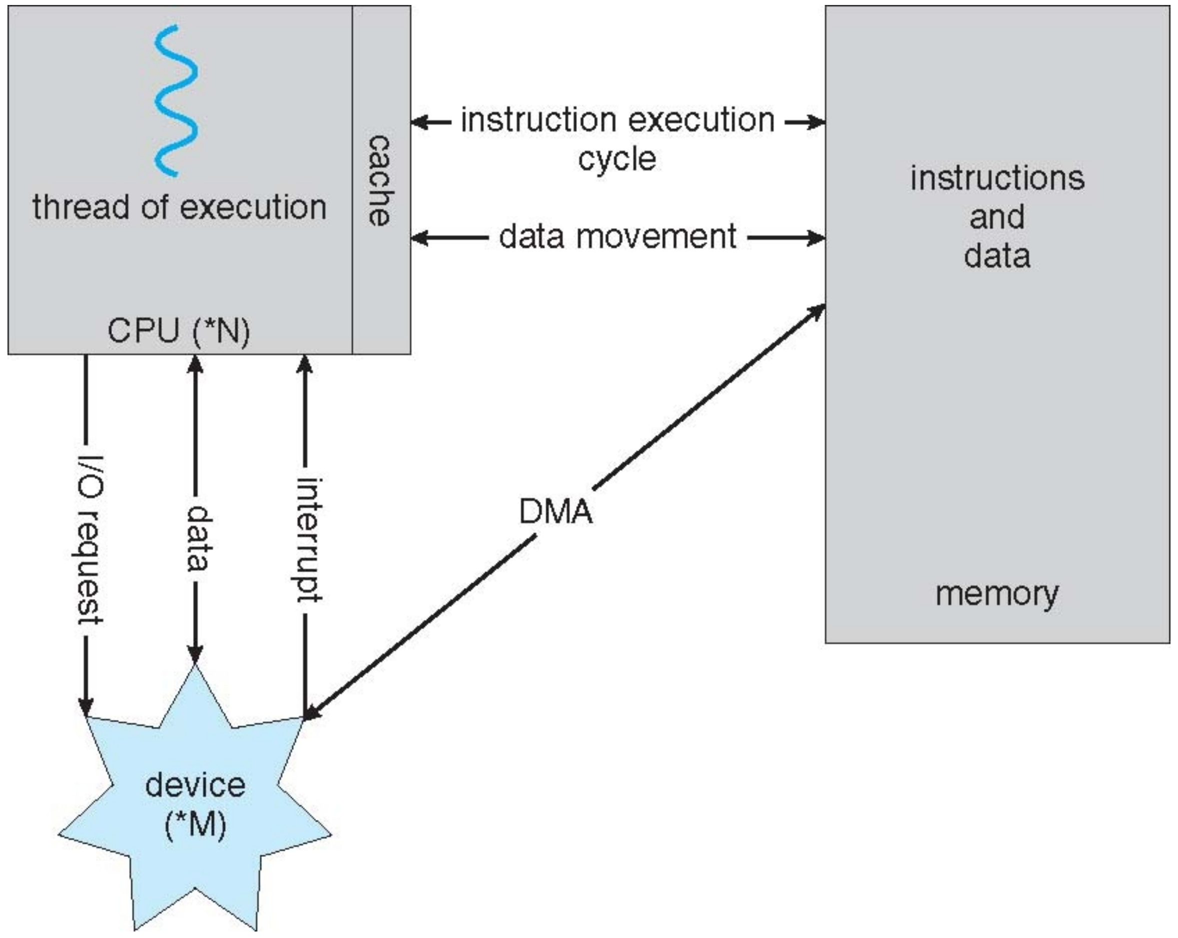


Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Movement between levels of storage hierarchy can be explicit or implicit

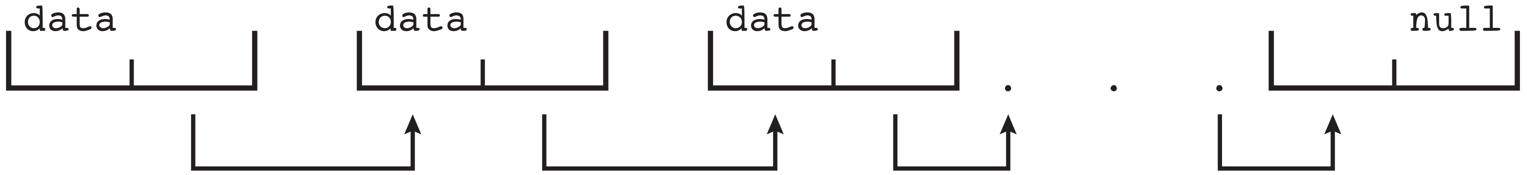
How a Modern Computer Works



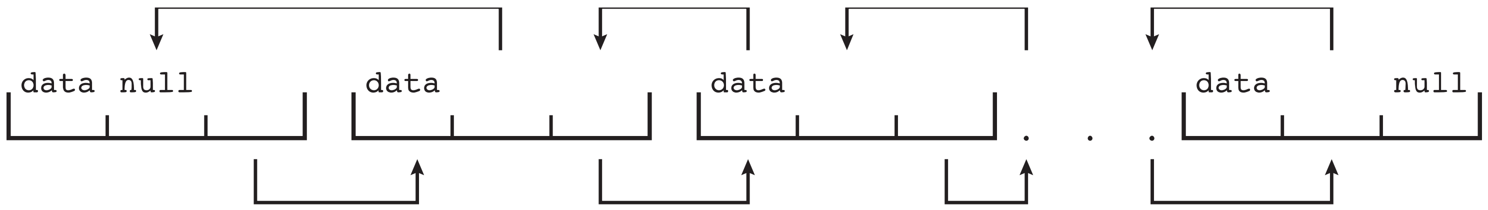
A von Neumann architecture

Kernel Data Structures

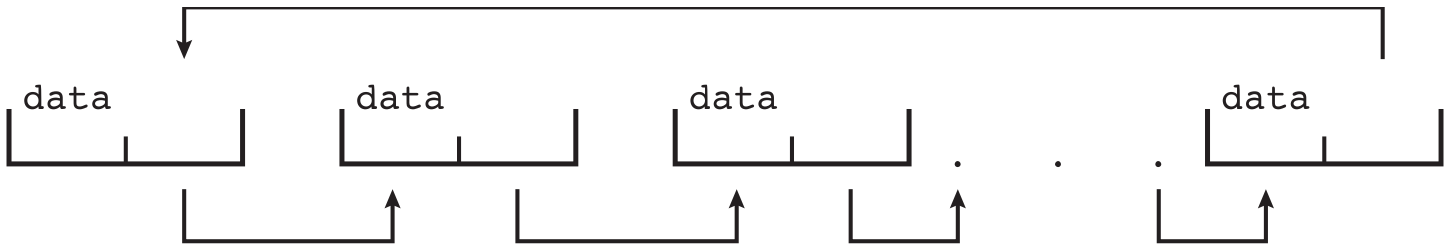
- Many similar to standard programming data structures
- Singly linked list



- Doubly linked list

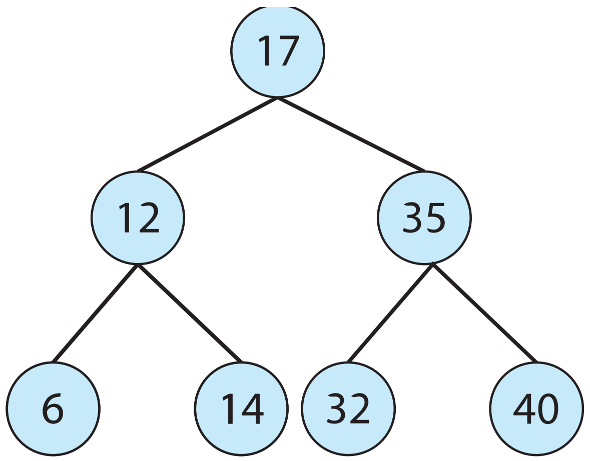


- Circular linked list



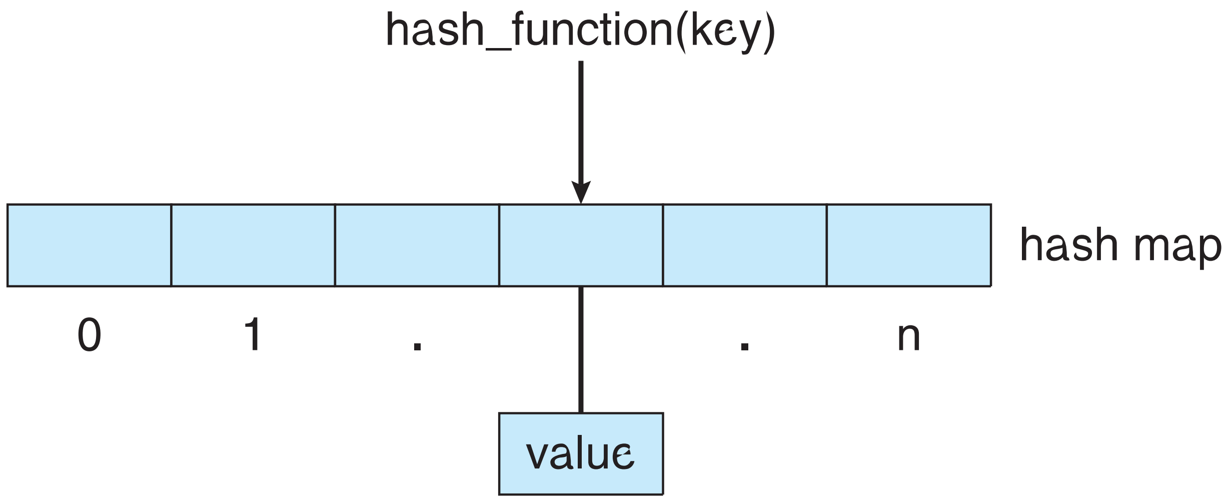
Kernel Data Structures

- **Binary search tree** left <= right
  - Search performance is *O(n)*
  - **Balanced binary search tree** is *O(lg n)*



Kernel Data Structures

- **Hash function** can create a **hash map**



- **Bitmap** – string of *n* binary digits representing the status of *n* items
- Linux data structures defined in ***include*** files <linux/list.h>, <linux/kfifo.h>, <linux/rbtree.h>

Software complexity

Millions of Lines of Code

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

Very Brief History of OS

- Several Distinct Phases:
    - Hardware Expensive, Humans Cheap
        - Eniac, … Multics

Very Brief History of OS

- Several Distinct Phases:

"I think there is a world market for maybe five computers." – Thomas Watson, chairman of IBM, 1943

Very Brief History of OS

- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - Eniac, … Multics

(Electronic Numerical Integrator And Computer)
(Multiplexed Information and Computing Service)

"I think there is a world market for maybe five computers." – Thomas Watson, chairman of IBM, 1943

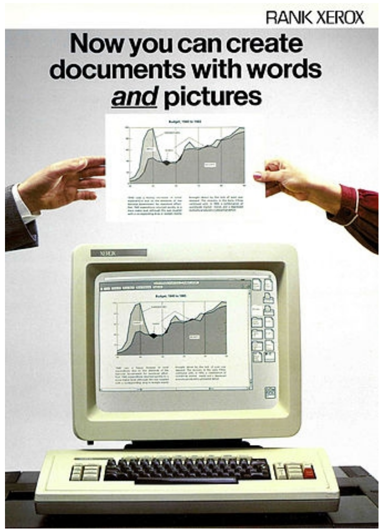
Very Brief History of OS

- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - Eniac, … Multics

Thomas Watson was often called "the worlds greatest salesman" by the time of his death in 1956

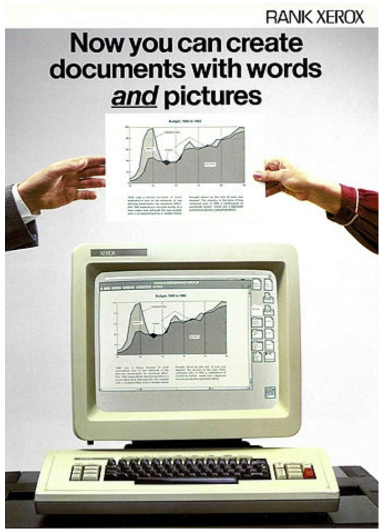
Very Brief History of OS

- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - Eniac, … Multics
  - Hardware Cheaper, Humans Expensive
    - PCs, Workstations, Rise of GUIs



Very Brief History of OS

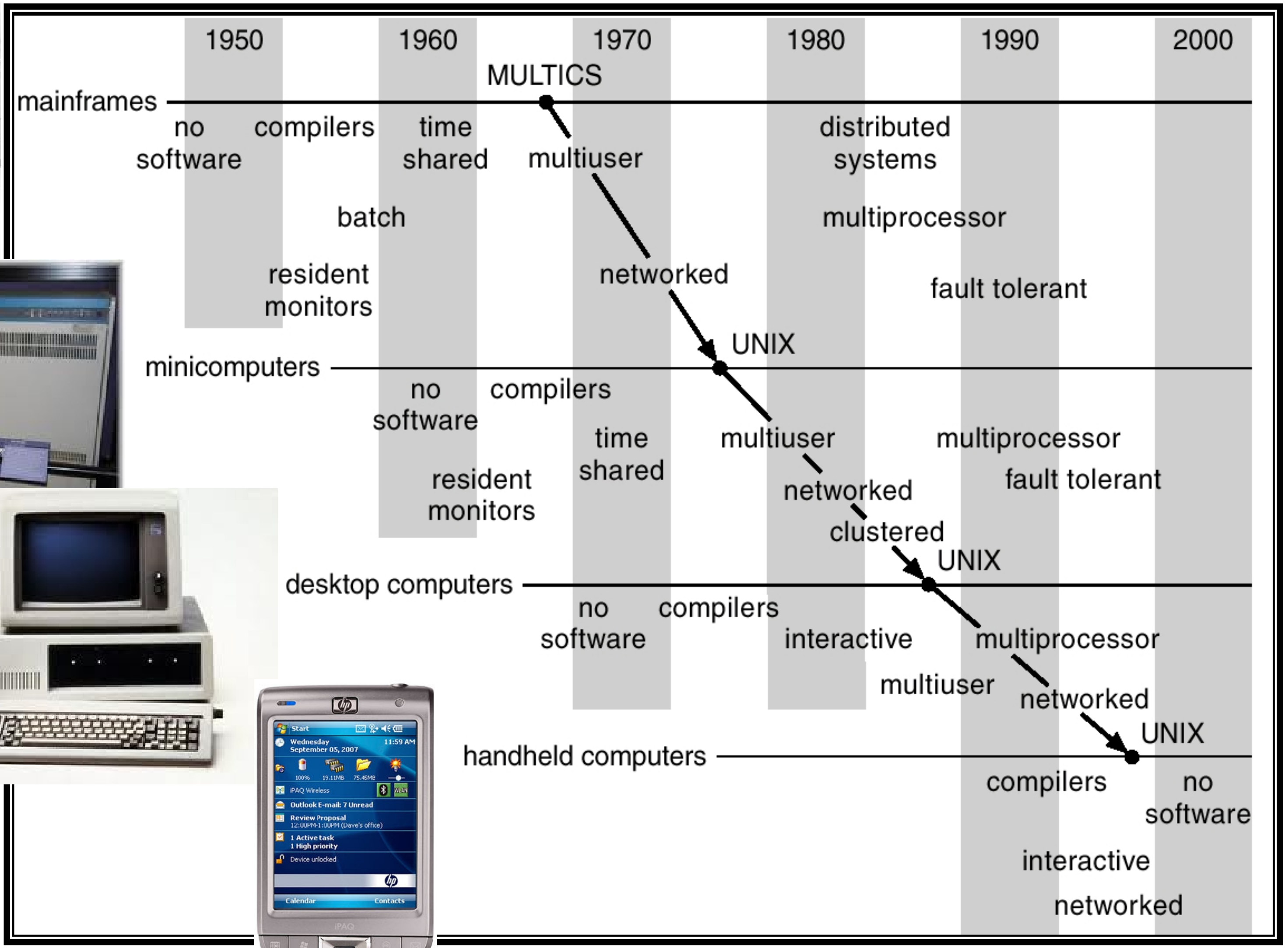
- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - Eniac, … Multics
  - Hardware Cheaper, Humans Expensive
    - PCs, Workstations, Rise of GUIs
  - Hardware Really Cheap, Humans Really Expensive
    - Ubiquitous devices, Widespread networking



OS Archaeology

- Because of the cost of developing an OS from scratch, most modern OSes have a long lineage:
- Multics → AT&T Unix → BSD Unix → Ultrix, SunOS, NetBSD,…
- Mach (micro-kernel) + BSD → NextStep → XNU → Apple OS X, iPhone iOS
- MINIX → Linux → Android OS, Chrome OS, RedHat, Ubuntu, Fedora, Debian, Suse,…
- CP/M → QDOS → MS-DOS → Windows 3.1 → NT → 95 → 98 → 2000 → XP → Vista → 7 → 8 → 10 → phone → …

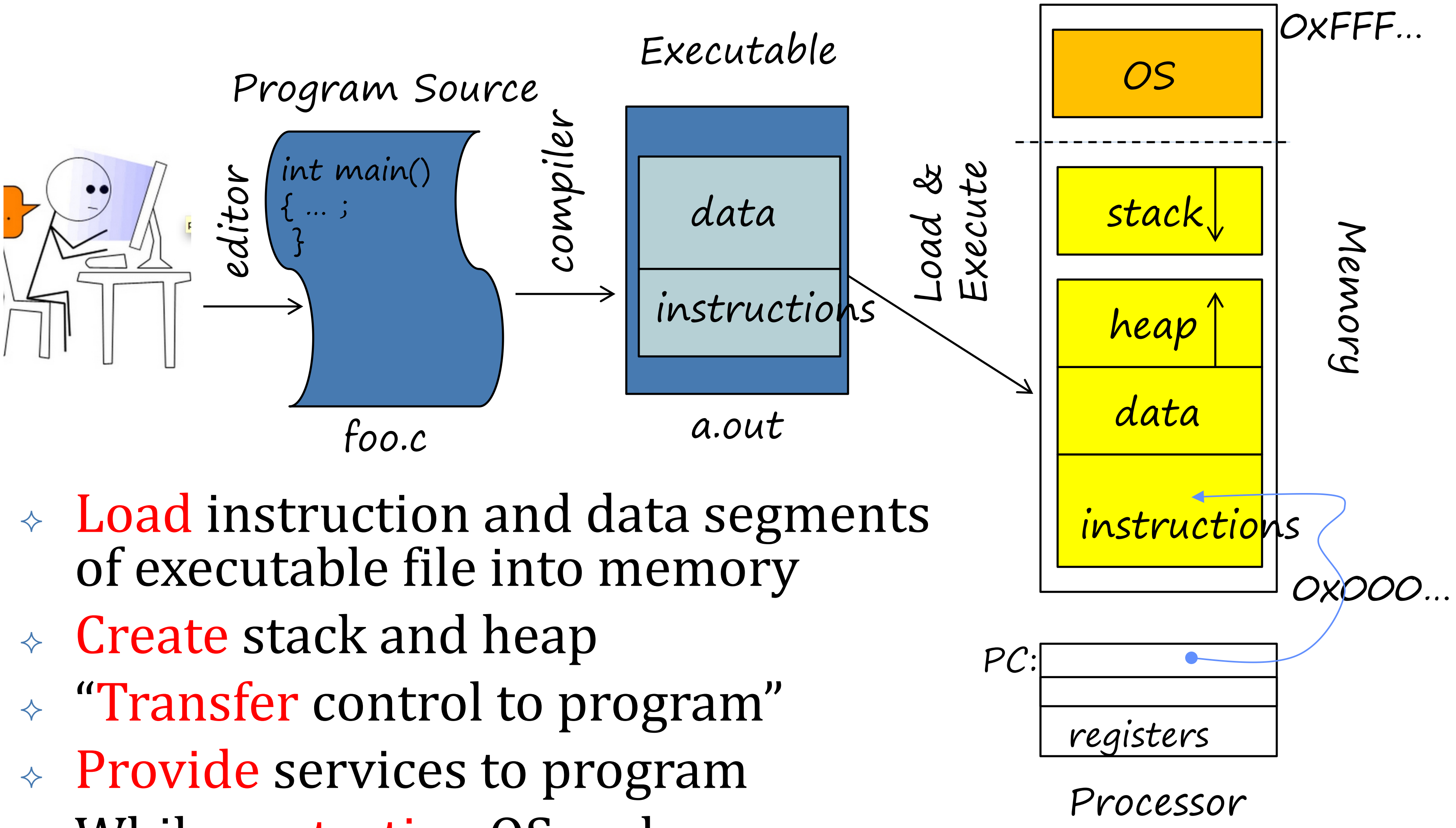
Migration of OS Concepts and Features



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

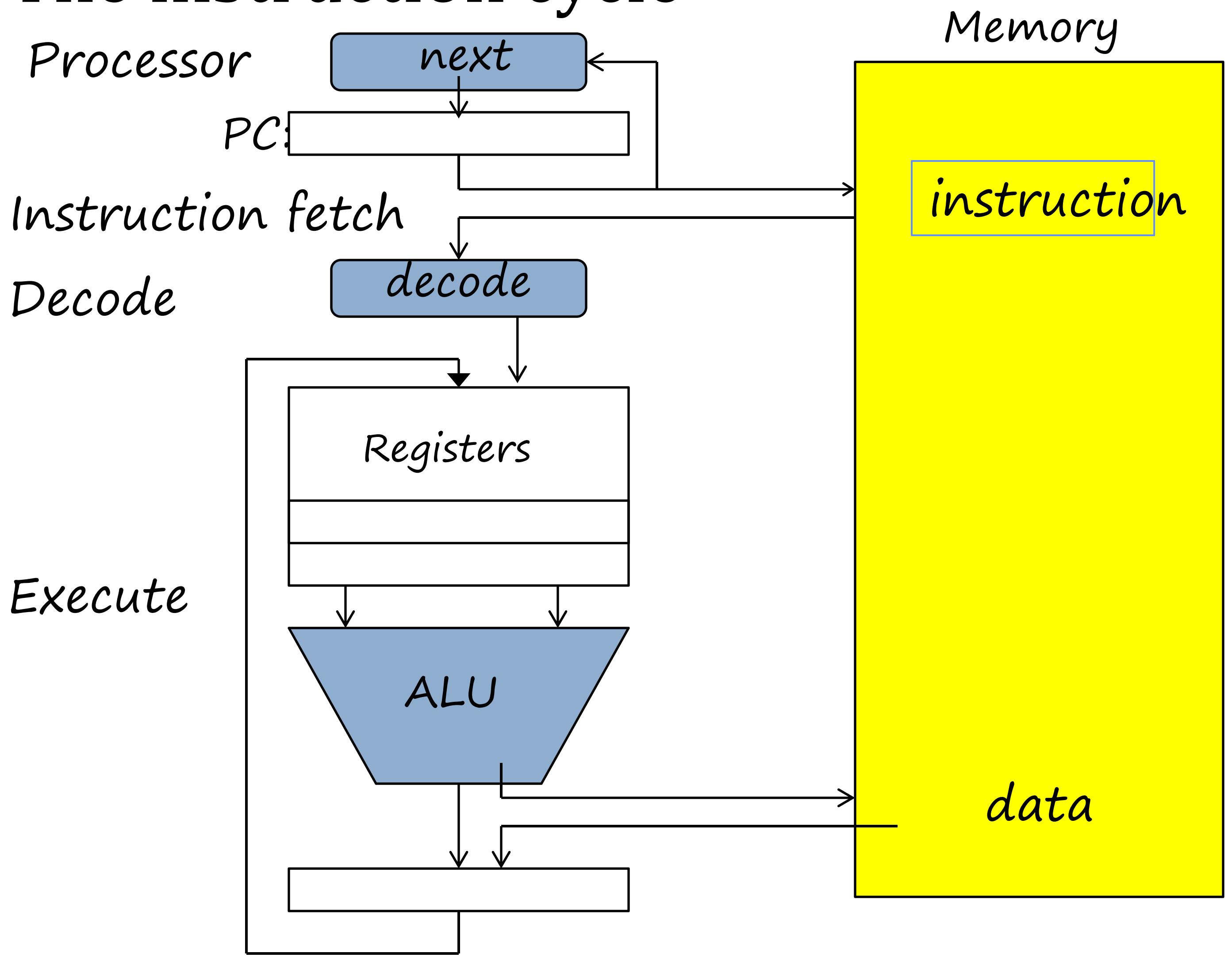
OS Bottom Line: Run Programs



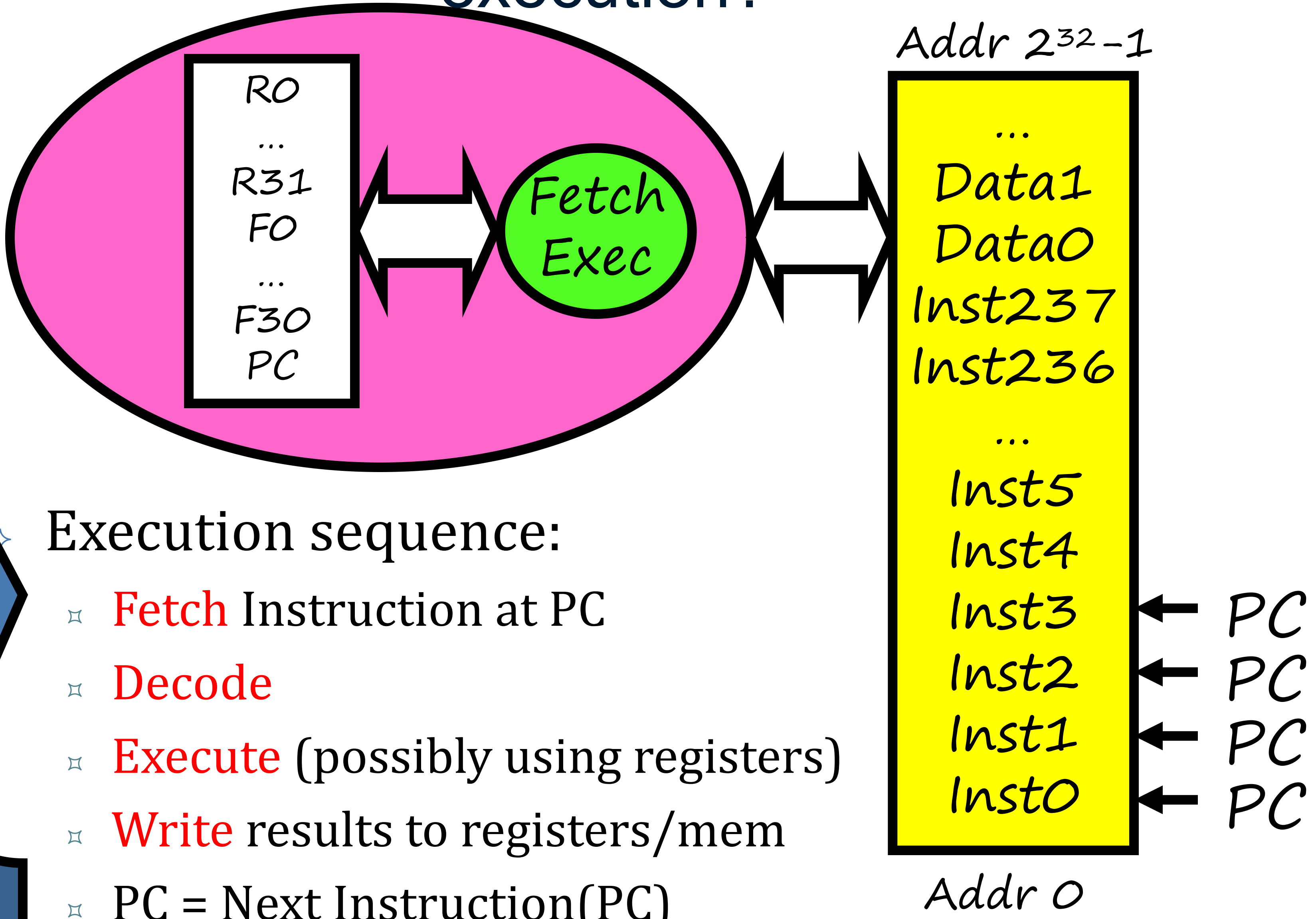
- **Load** instruction and data segments of executable file into memory
- **Create** stack and heap
- "**Transfer** control to program"
- **Provide** services to program
- While **protecting** OS and program

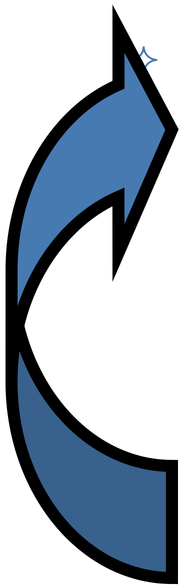
Recall: Instruction Fetch/Decode/Execute

The instruction cycle



Recall: What happens during program execution?





Execution sequence:

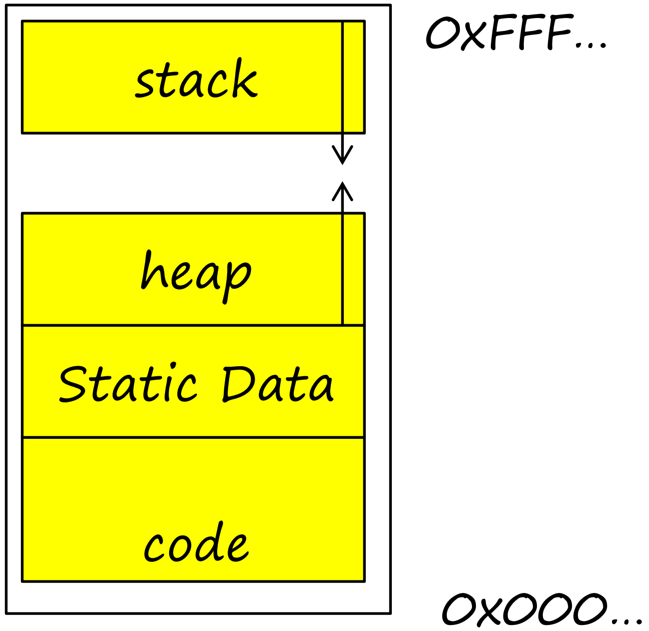
- Fetch Instruction at PC
- Decode
- Execute (possibly using registers)
- Write results to registers/mem
- PC = Next Instruction(PC)
- Repeat

First OS Concept: Thread of Control

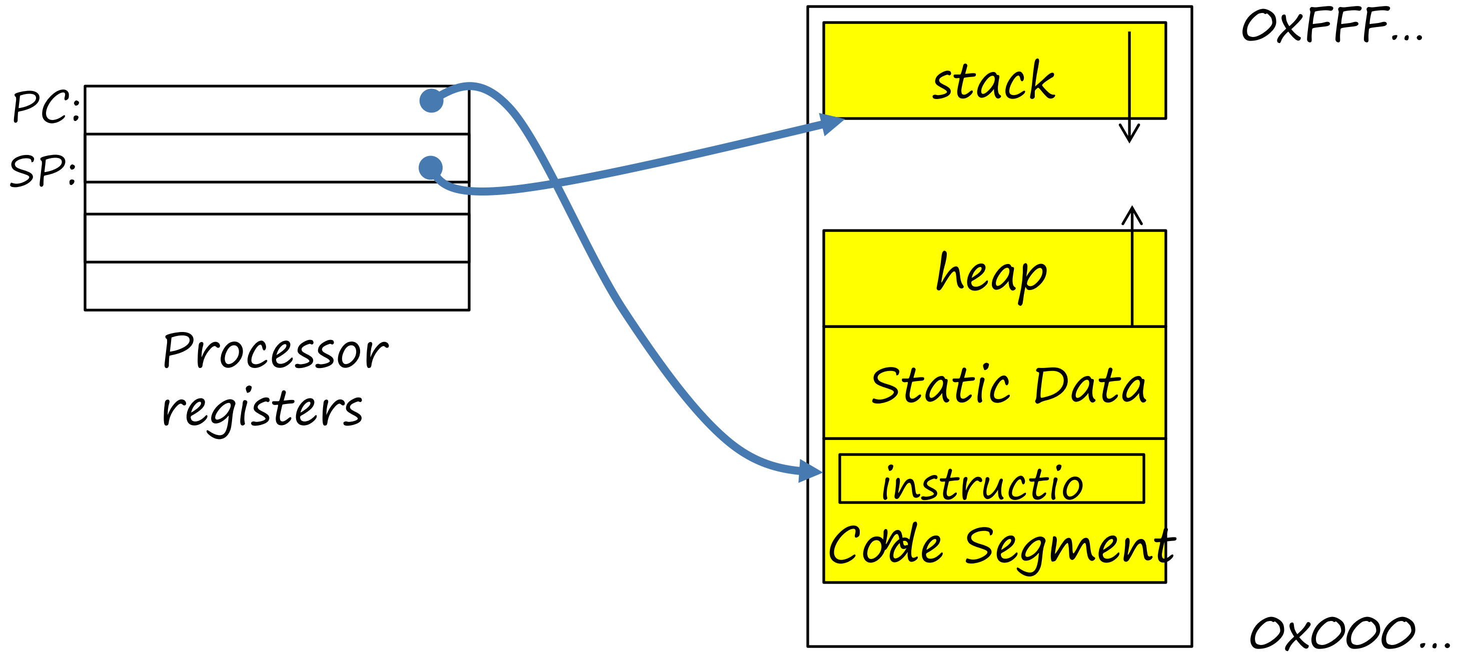
- Certain registers hold the *context* of thread
  - Stack pointer holds the address of the top of stack
    - Other conventions: Frame pointer, Heap pointer, Data
  - May be defined by the instruction set architecture or by compiler conventions
- Thread: Single unique execution context
  - Program Counter, 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)

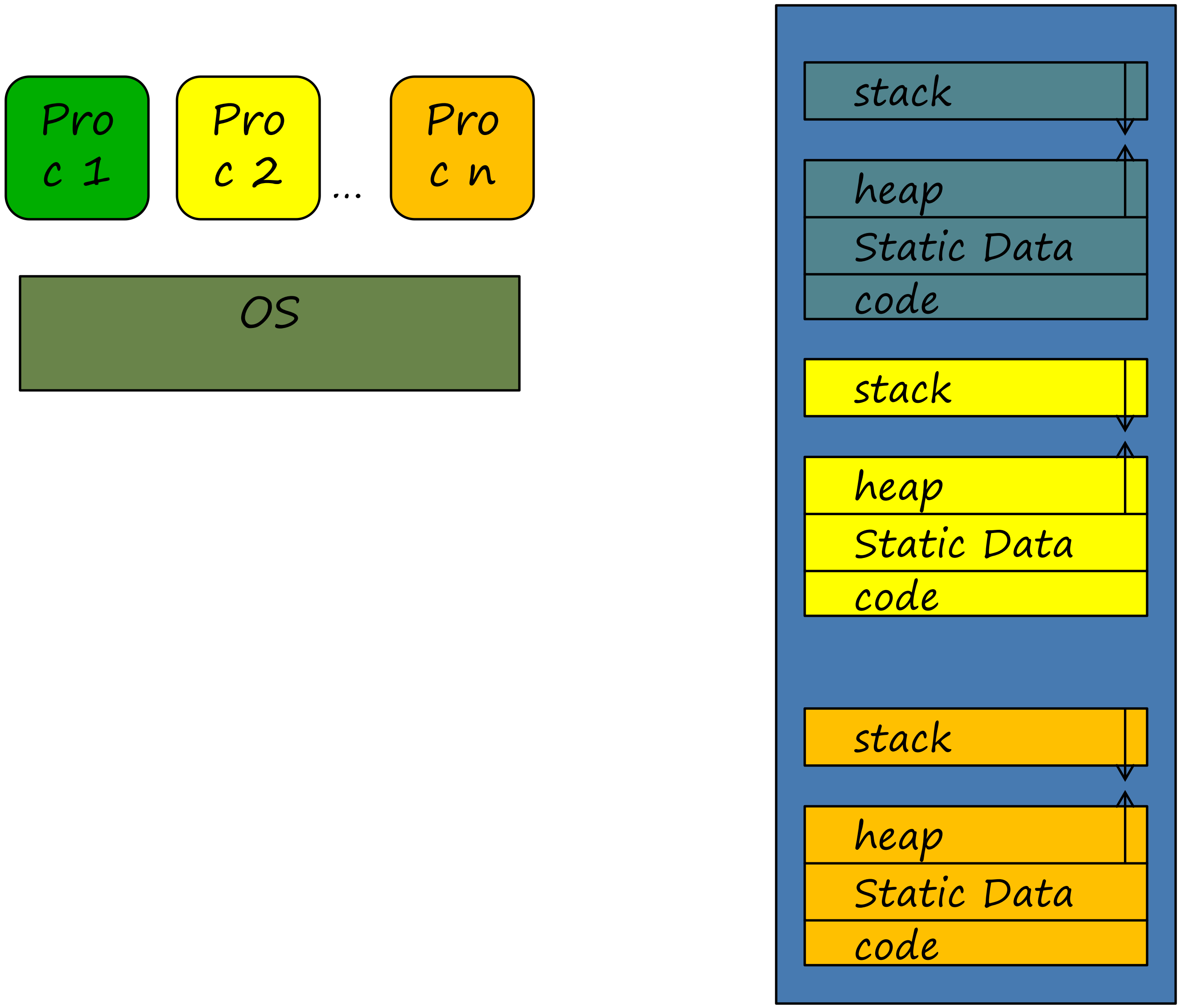


Address Space: In a Picture

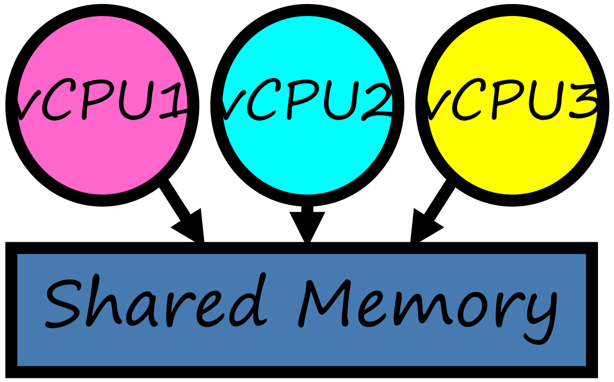


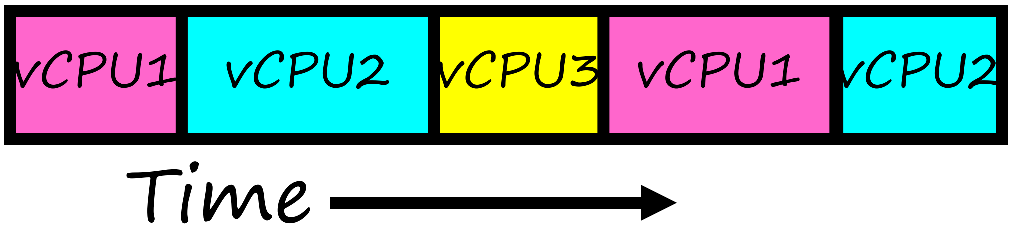
- What is in the code segment? Static data segment?
- What is in the Stack Segment?
    - How is it allocated? How big is it?
- What is in the Heap Segment?
    - How is it allocated? How big?

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 activities
  - 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)

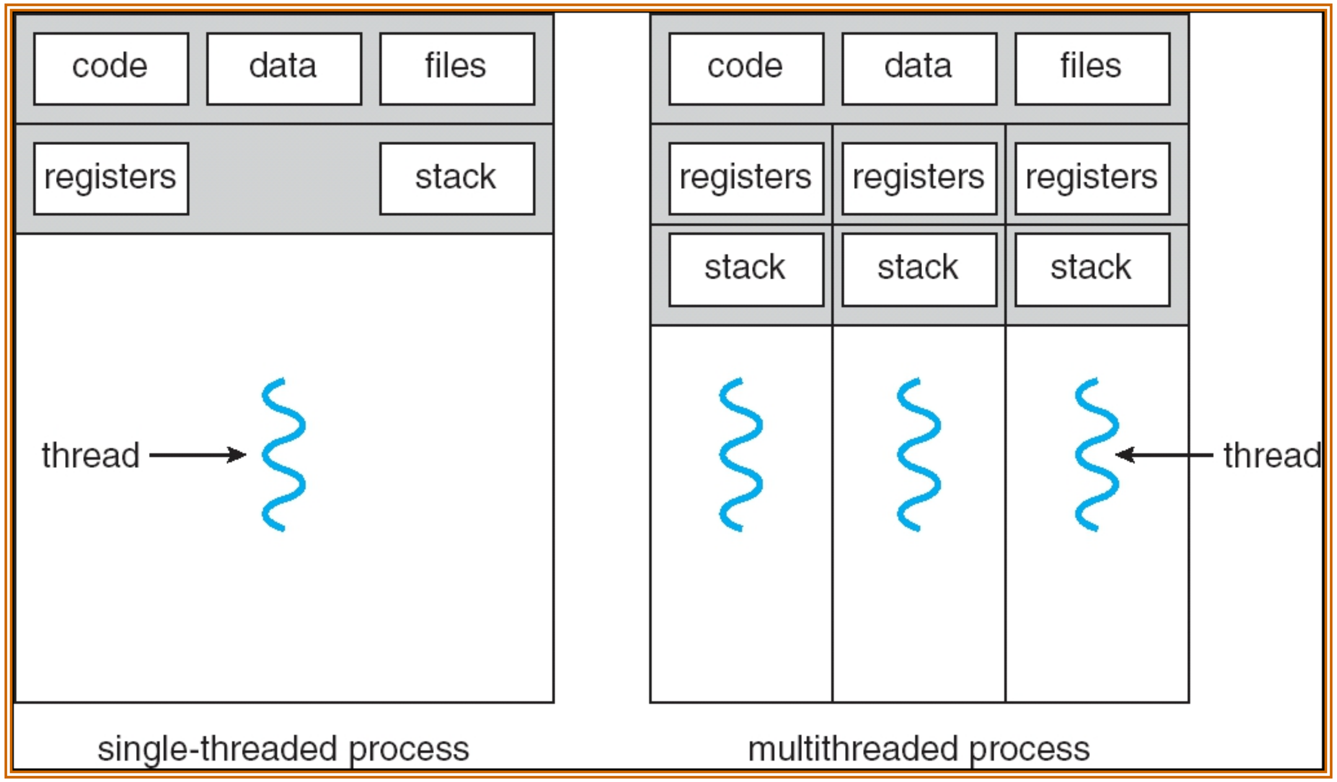
Protection

- 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

Single and Multithreaded Processes

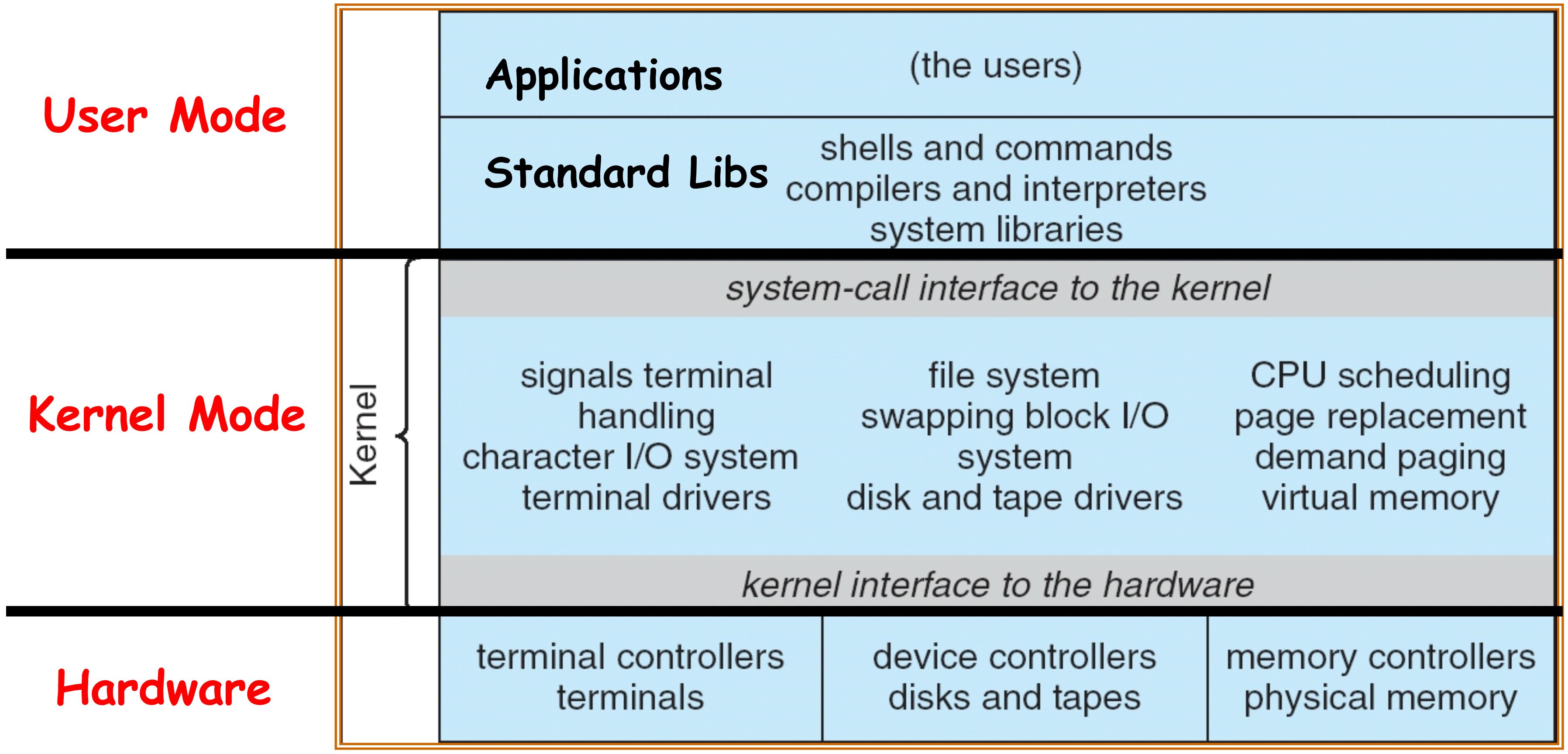


- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

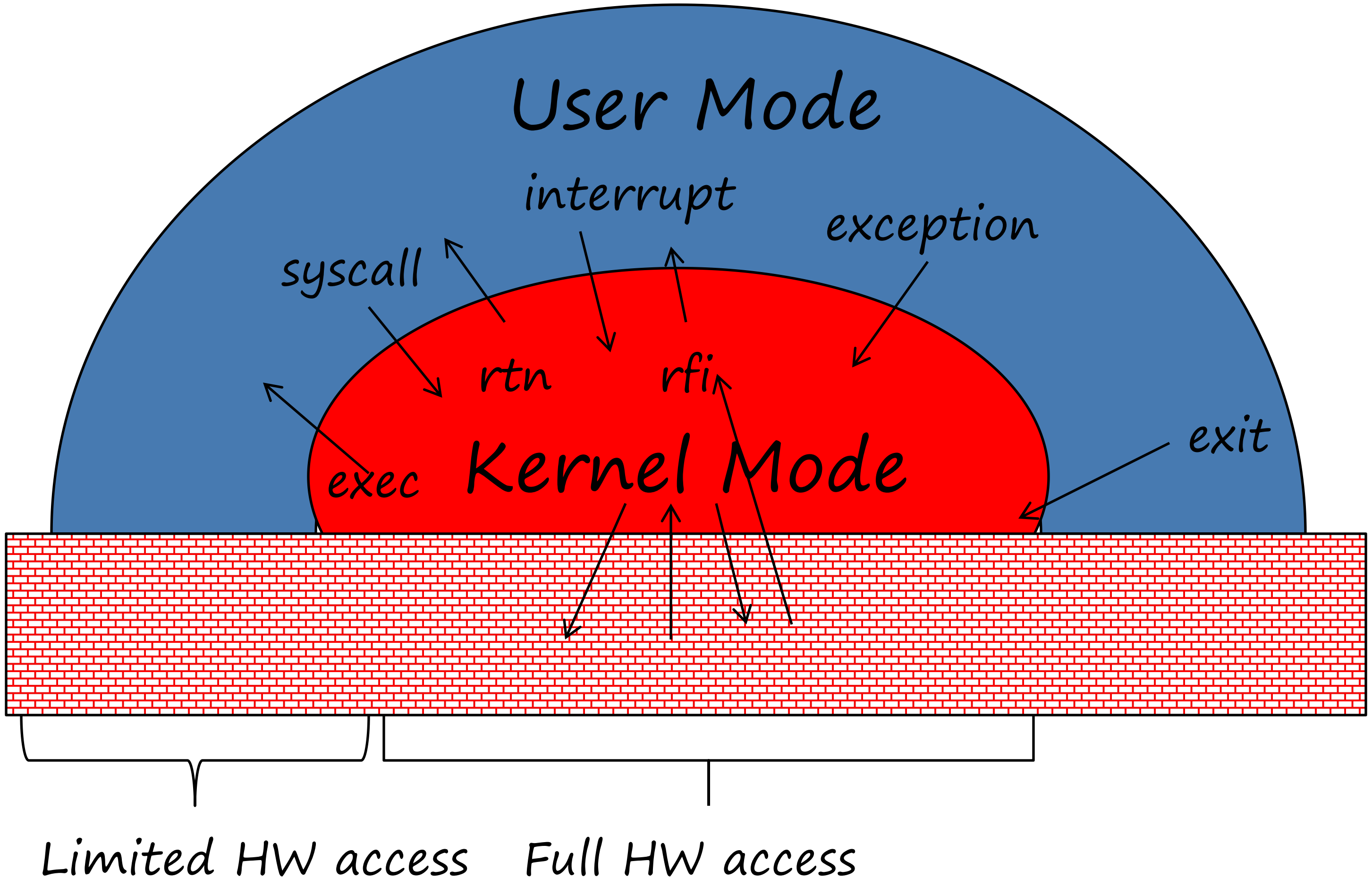
Fourth OS Concept: Dual Mode Operation

- 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

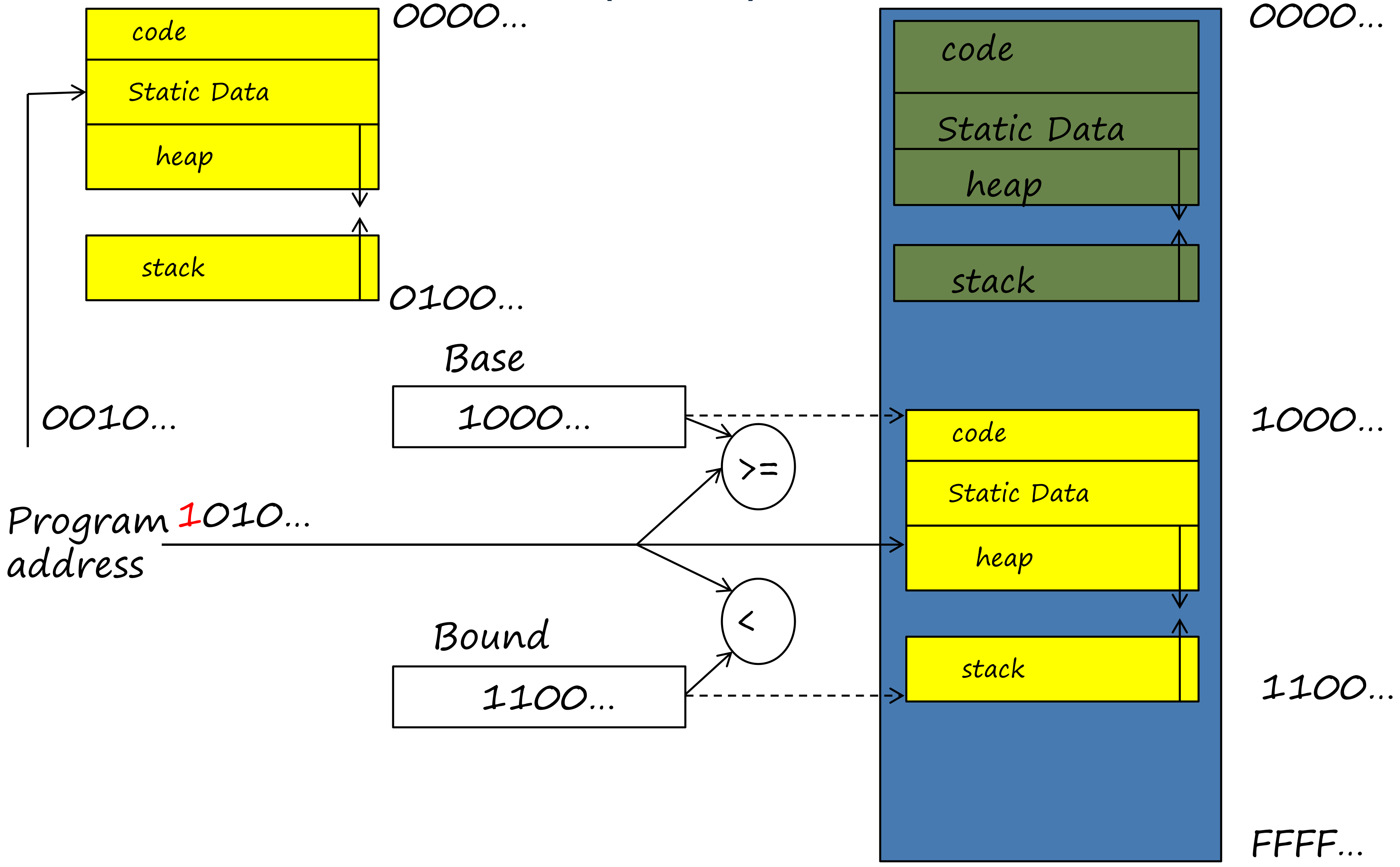
Unix System Structure



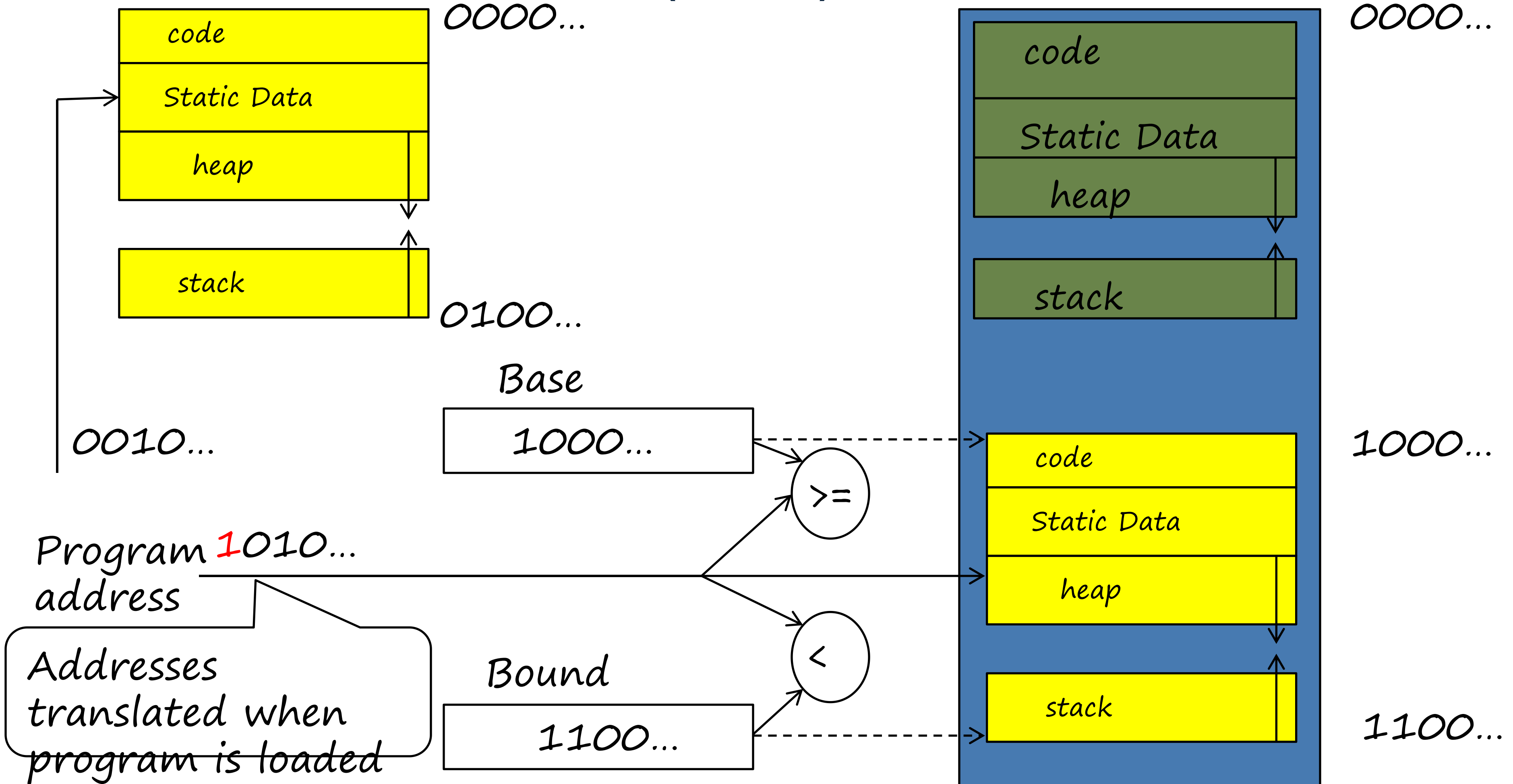
User/Kernel (Privileged) Mode



Simple Protection: Base and Bound (B&B)



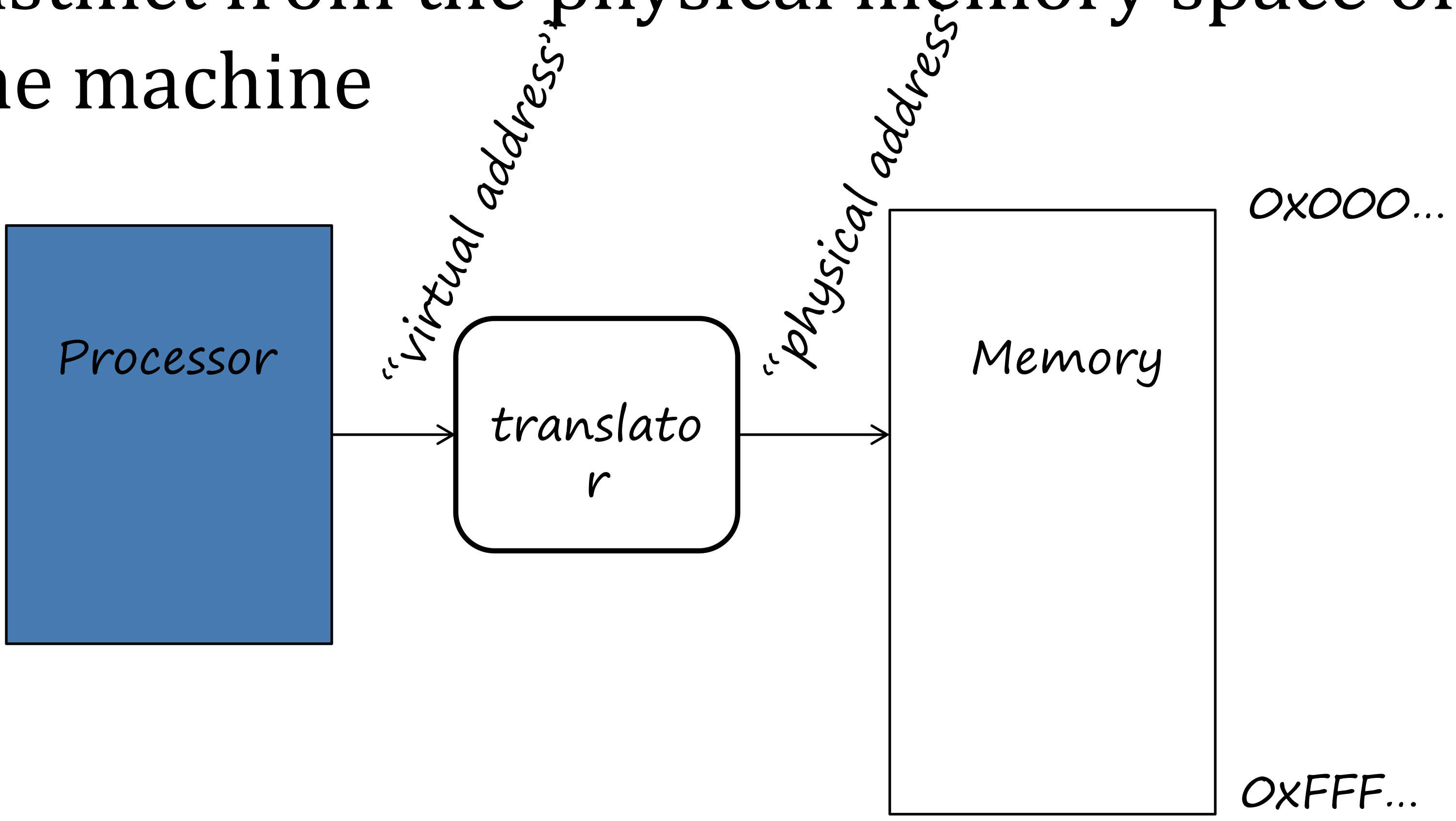
Simple Protection: Base and Bound (B&B)



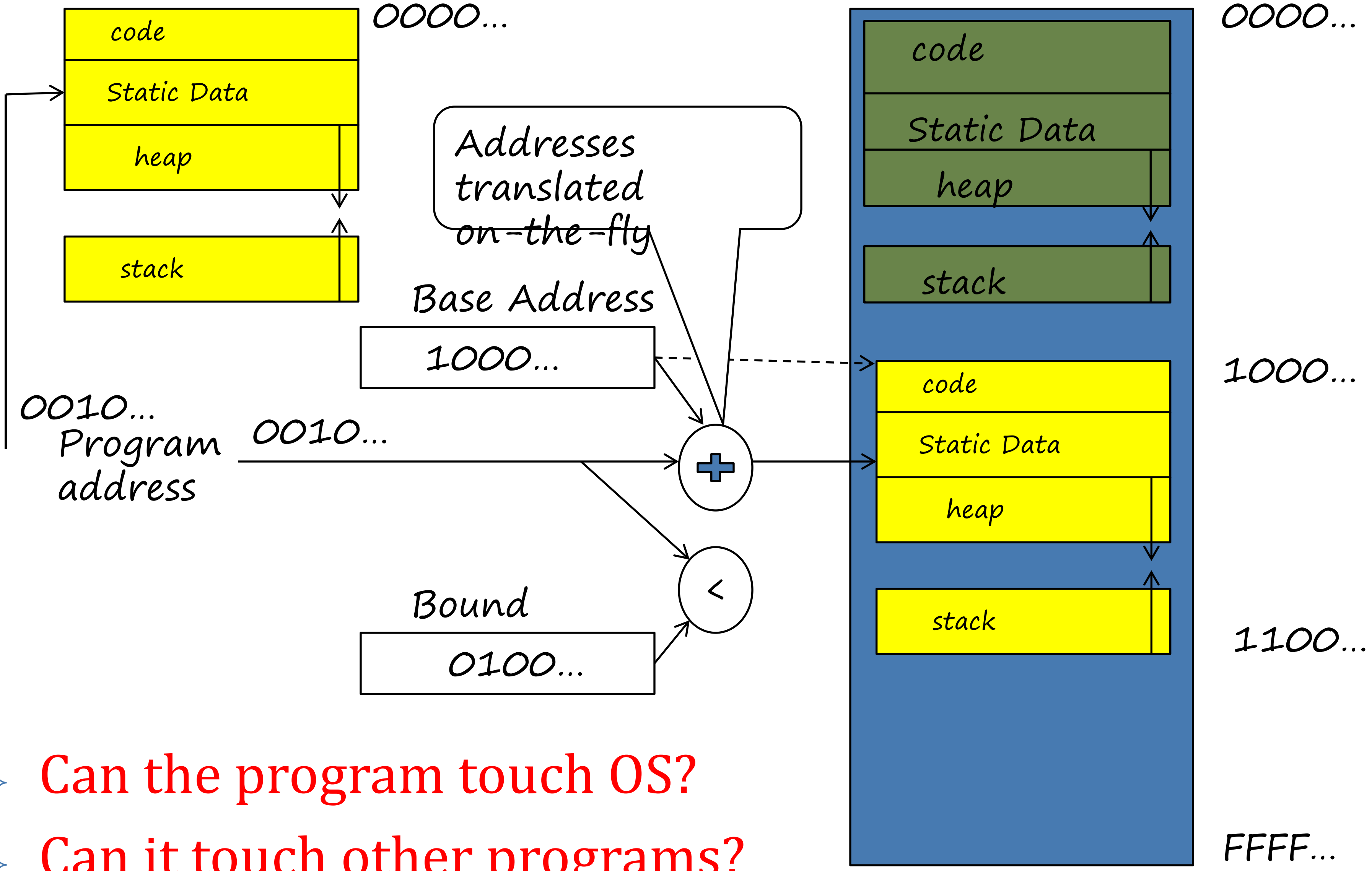
- Requires relocating loader
- Still protects OS and isolates program
- No addition on address path

Another idea: Address Space Translation

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

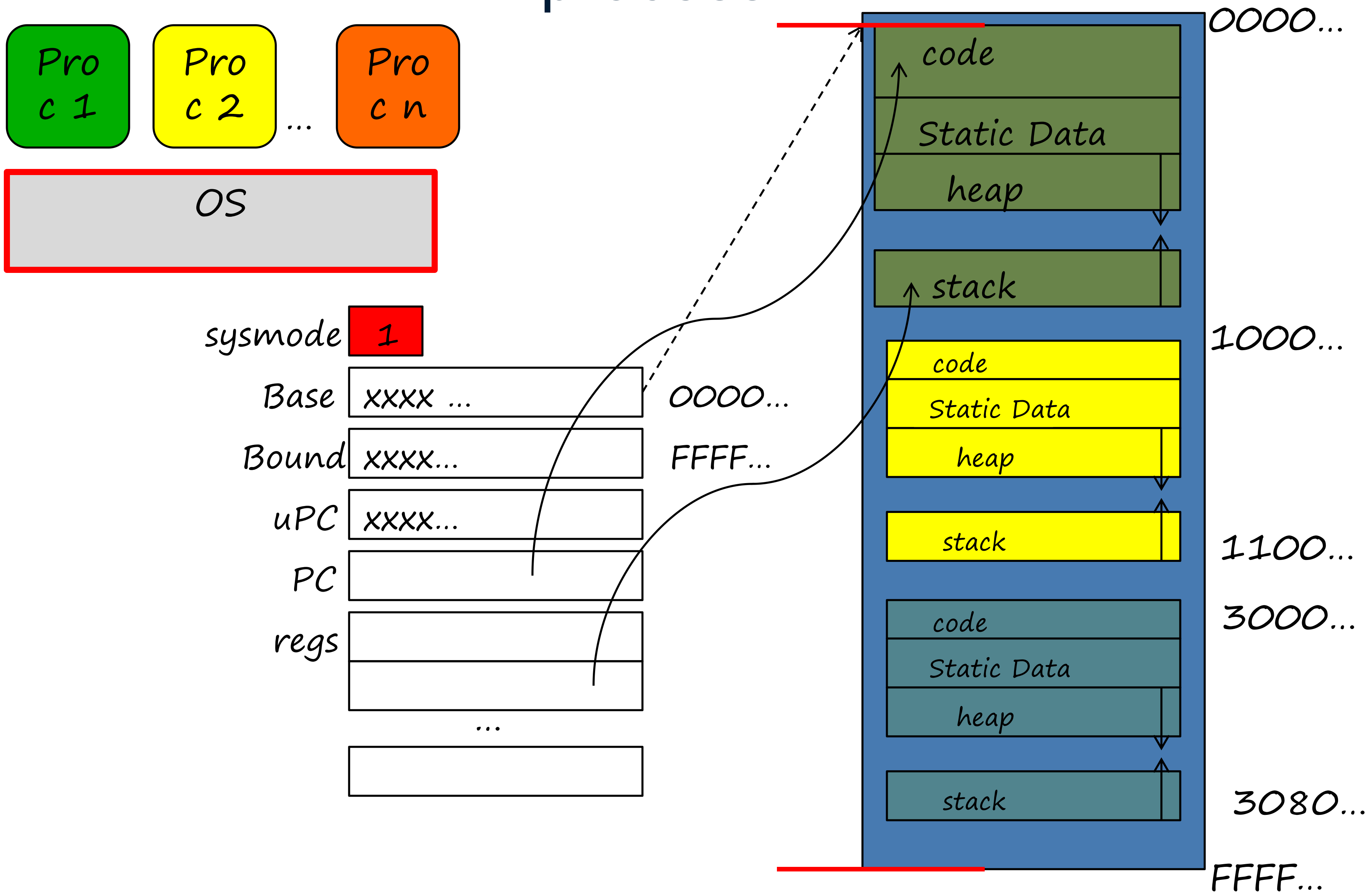


A simple address translation with Base and Bound

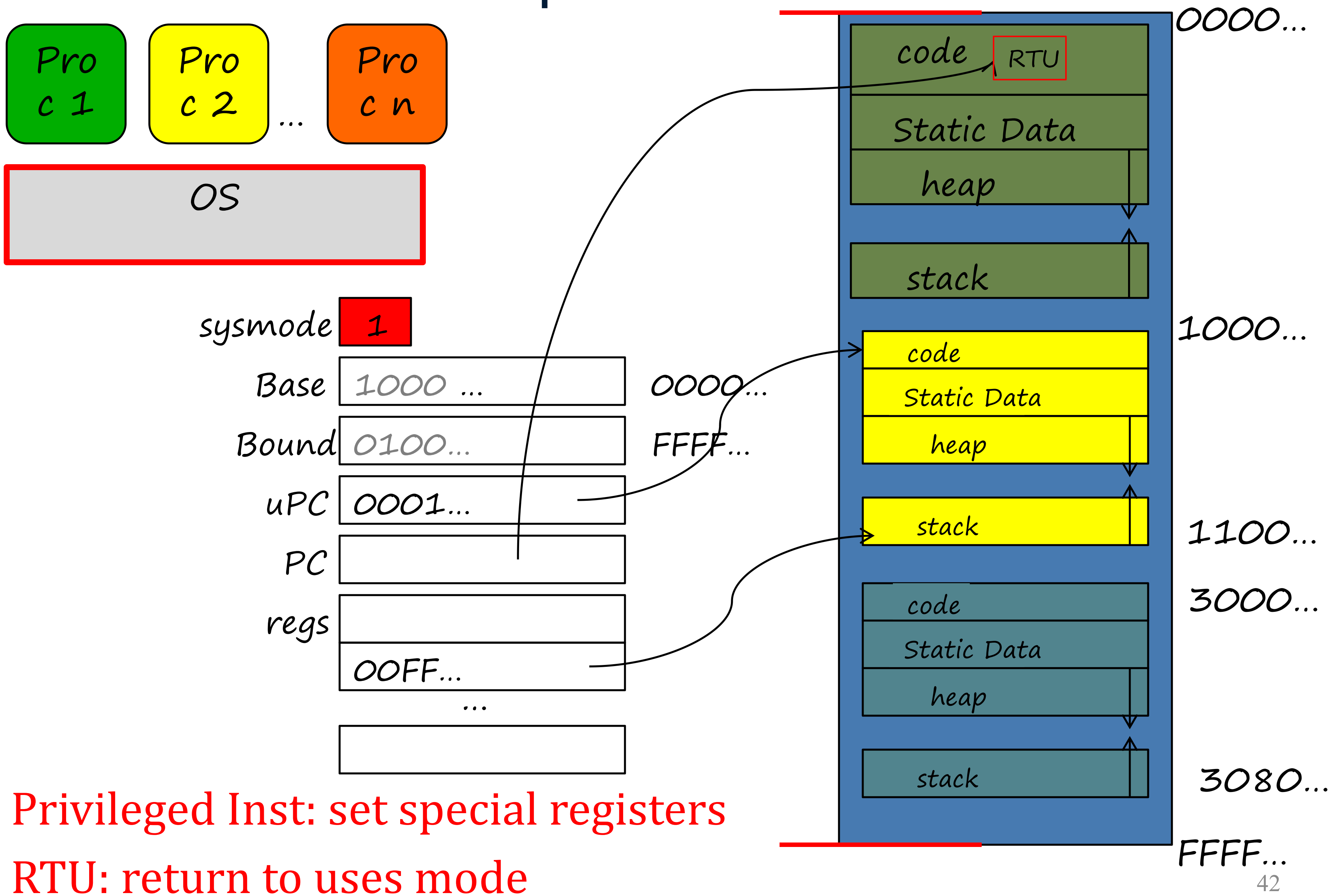


- Can the program touch OS?
- Can it touch other programs?

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

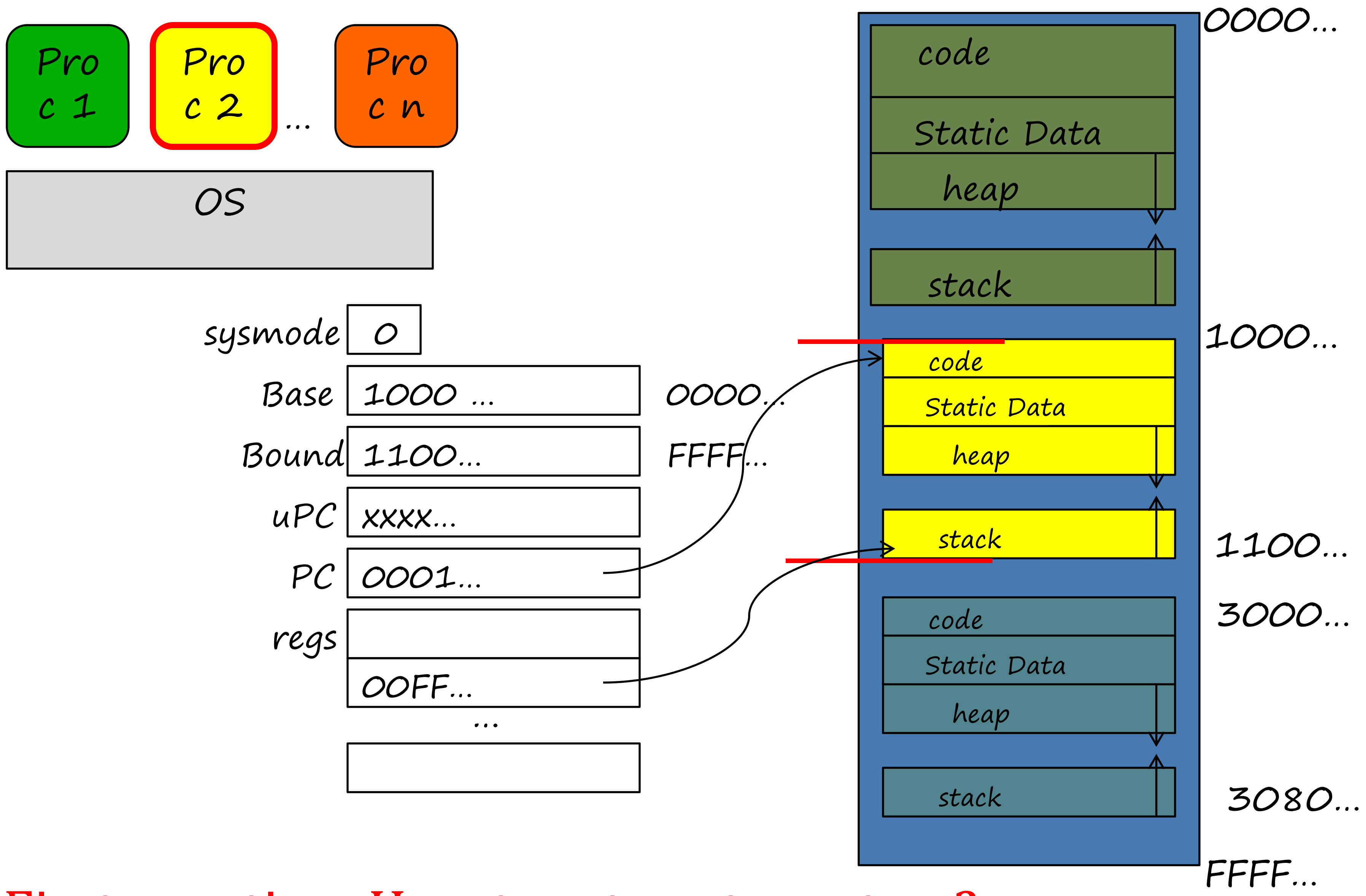


Simple B&B: OS gets ready to execute process



- Privileged Inst: set special registers
- RTU: return to uses mode

Simple B&B: User Code Running



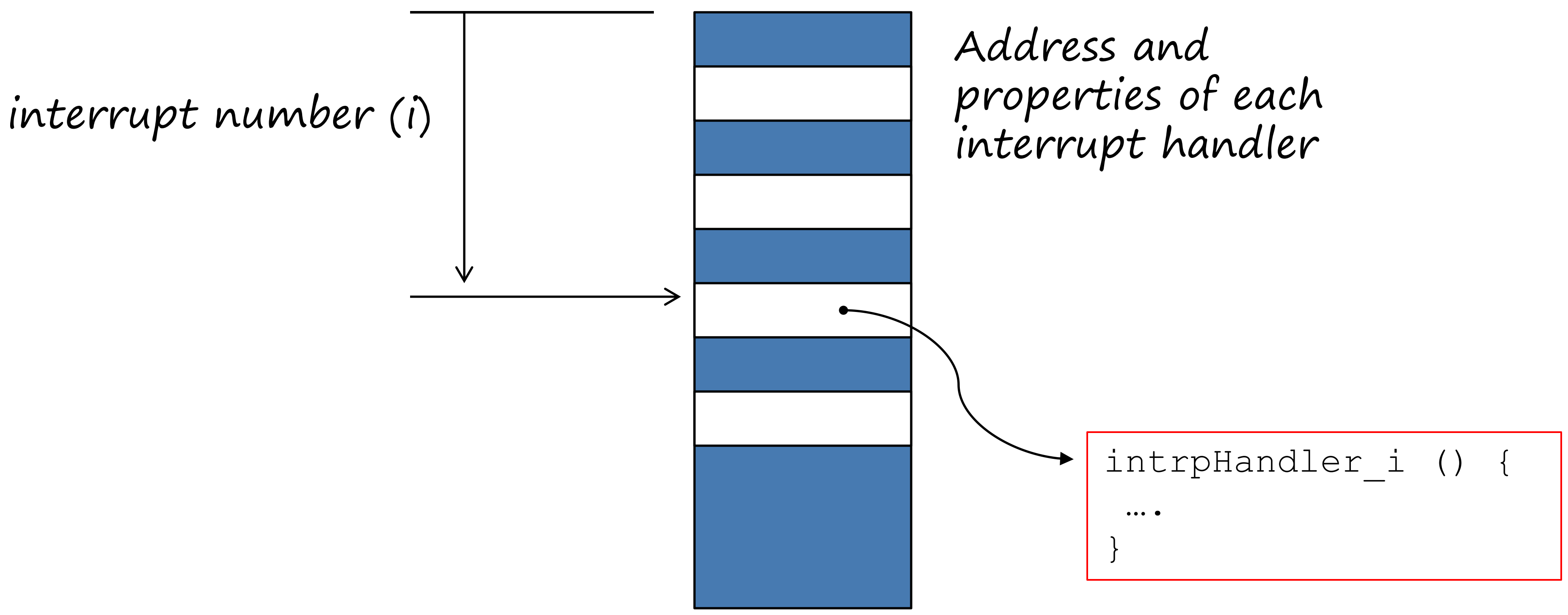
- First question: How to return to system?

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?”

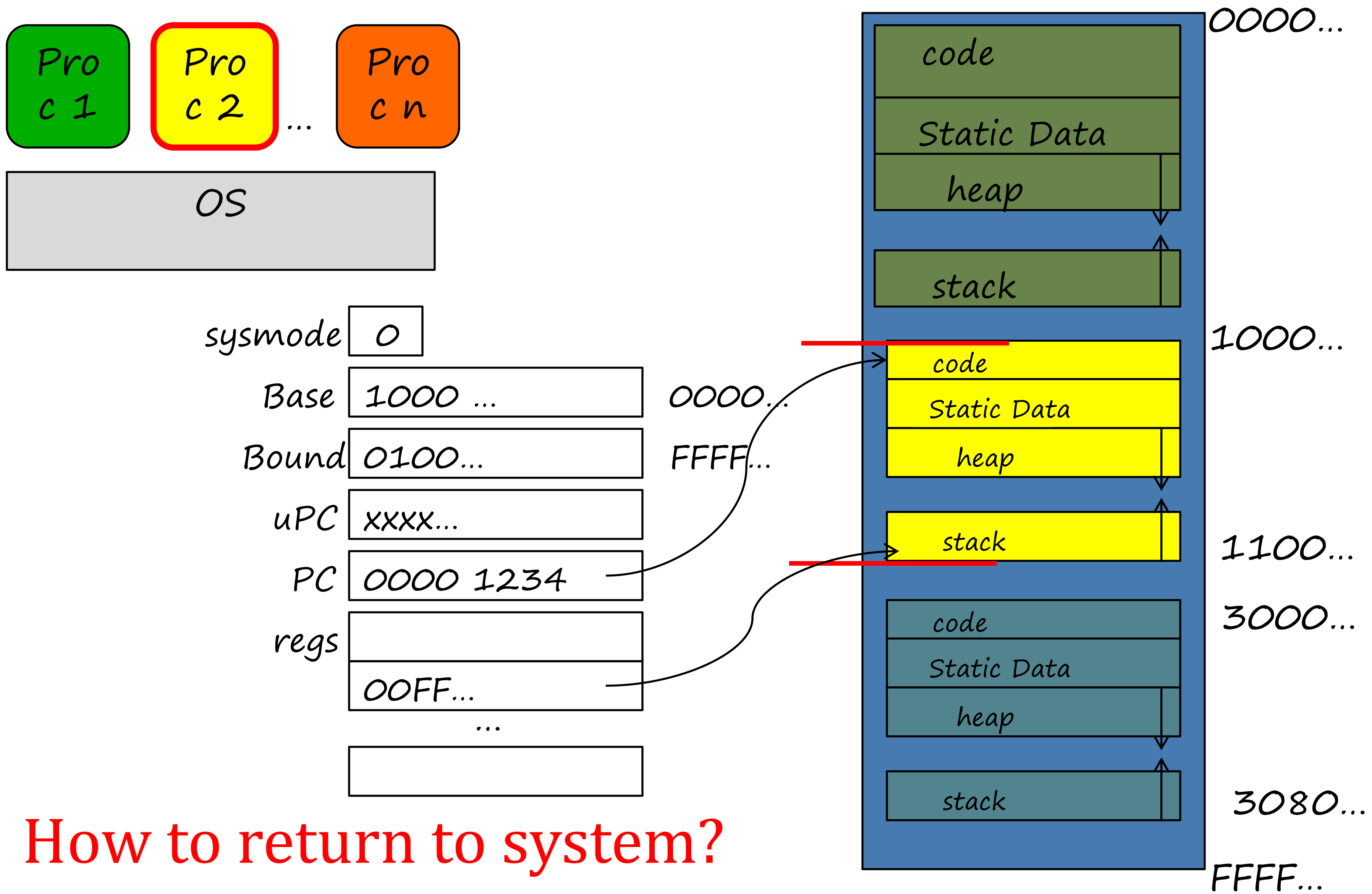
Interrupt Vector



```
intrpHandler_i () {
  ….
}
```

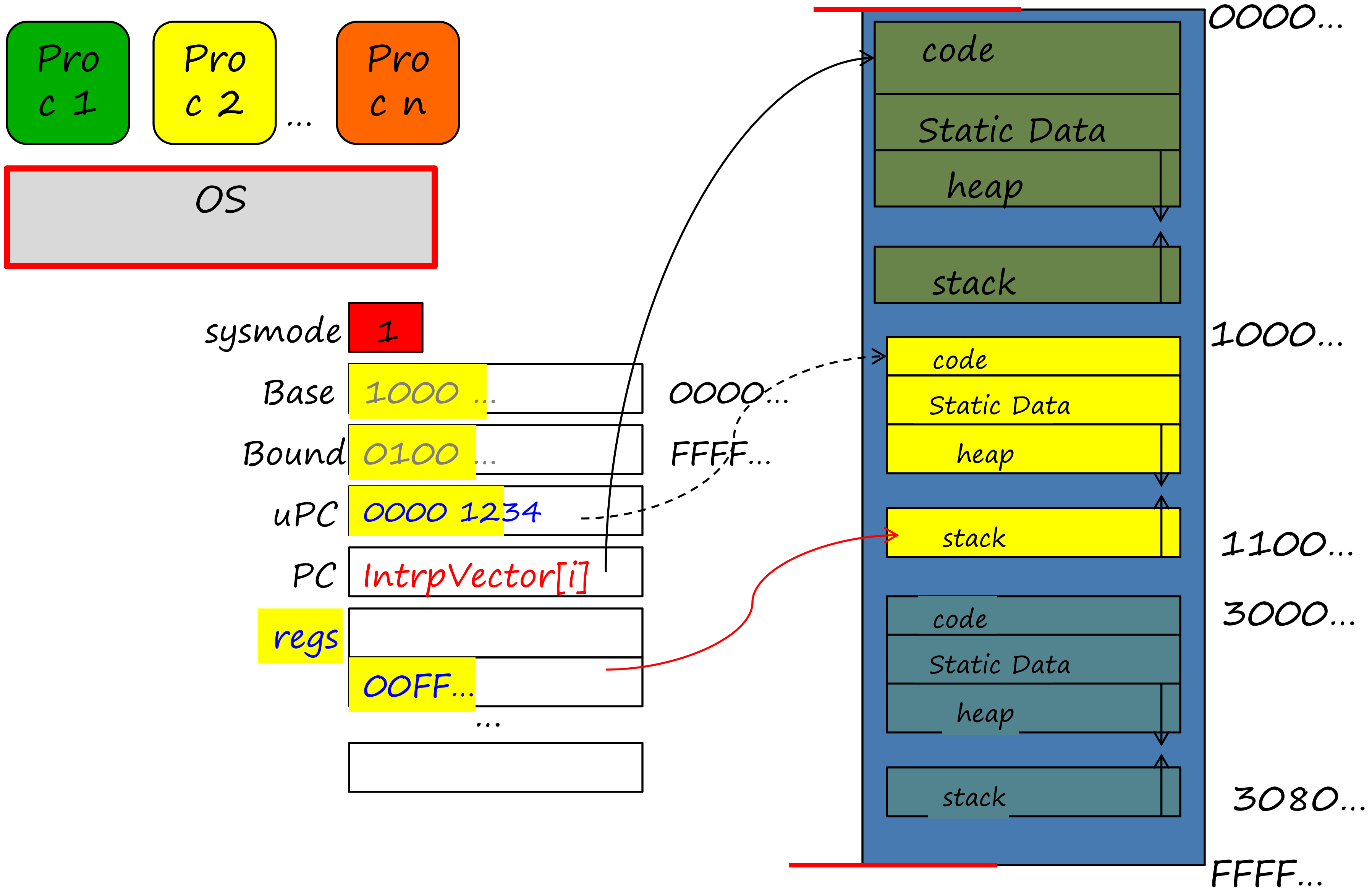
- Where else do you see this dispatch pattern?

Simple B&B: User => Kernel

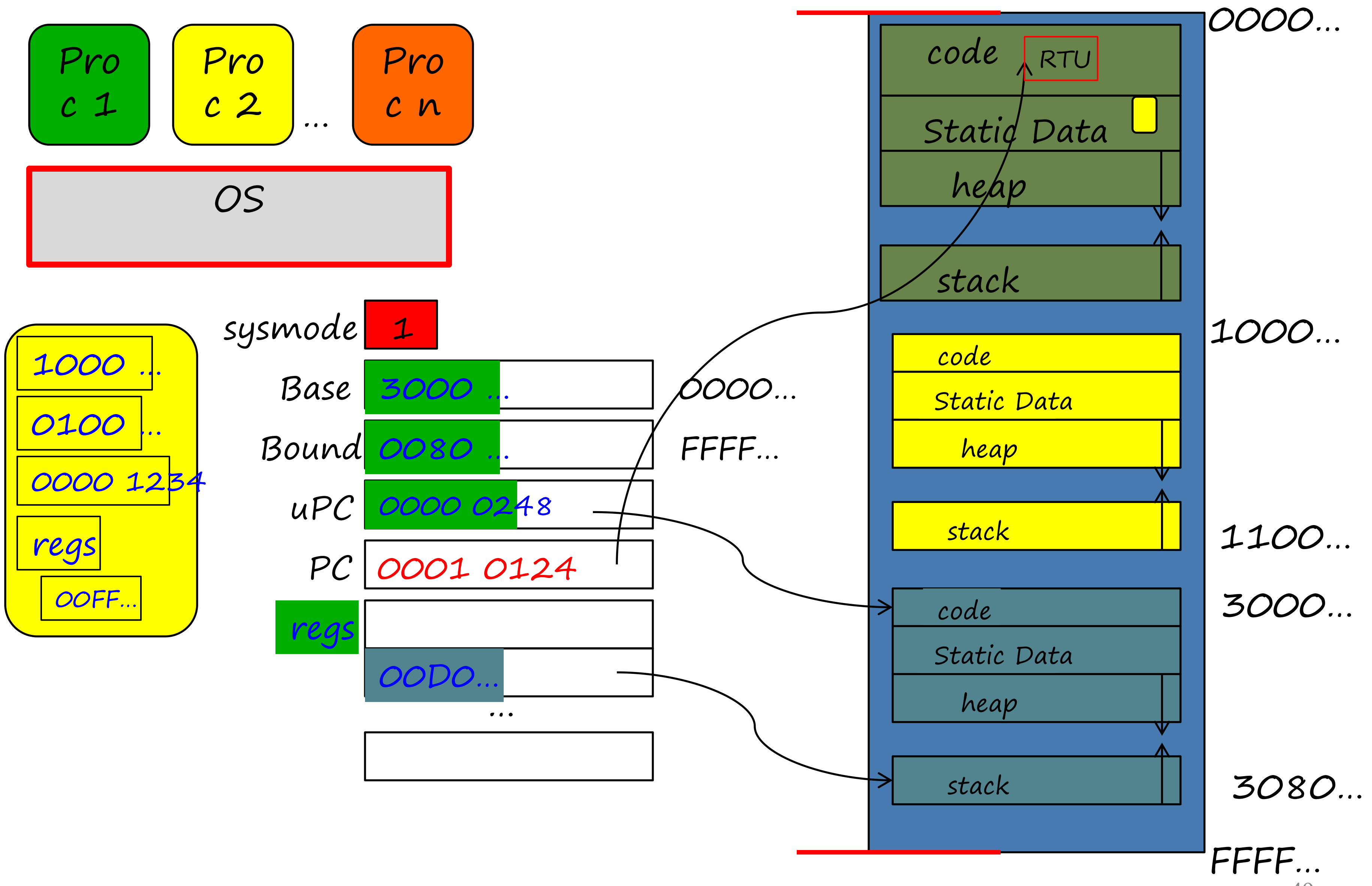


- How to return to system?

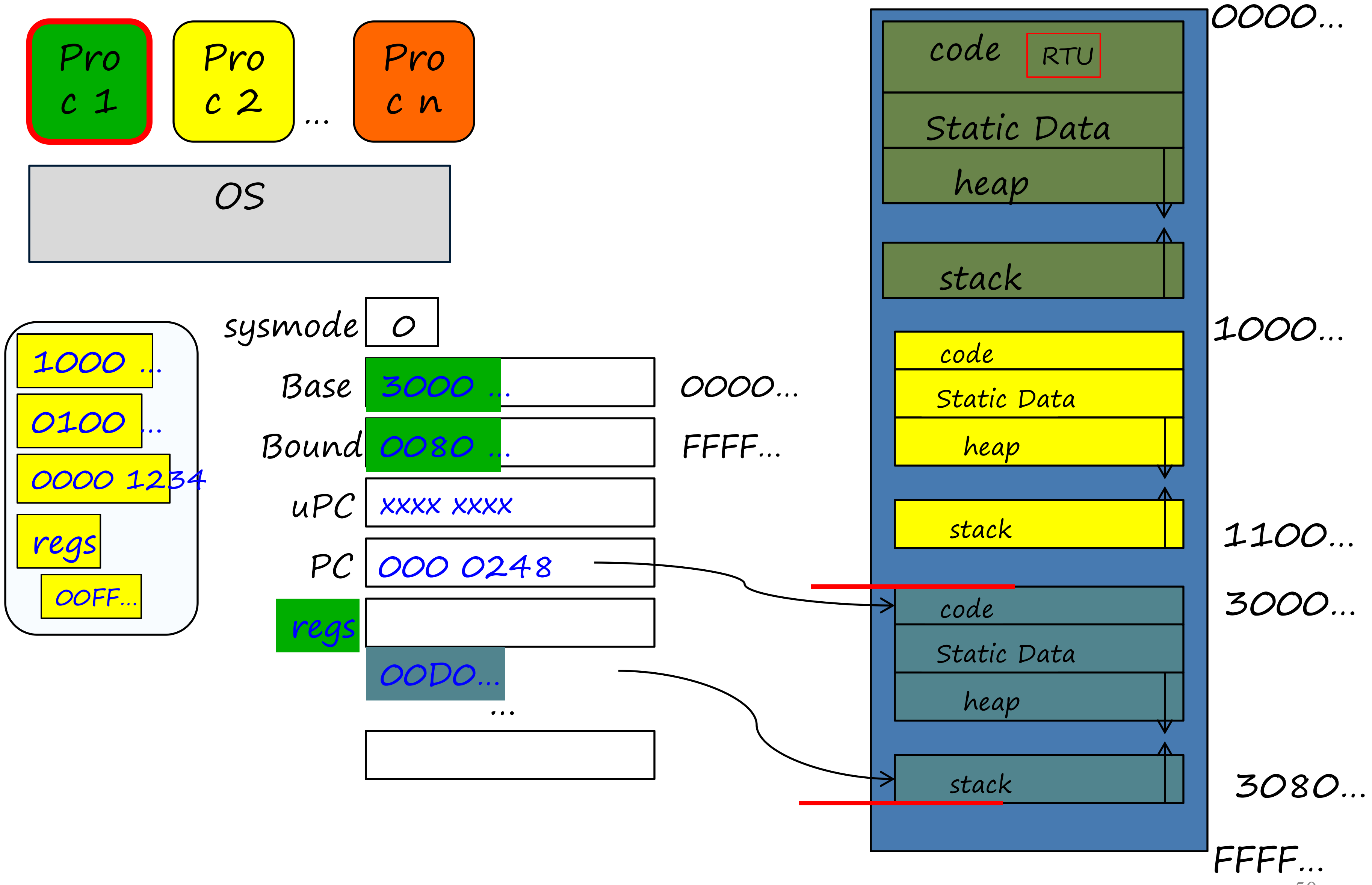
Simple B&B: Interrupt



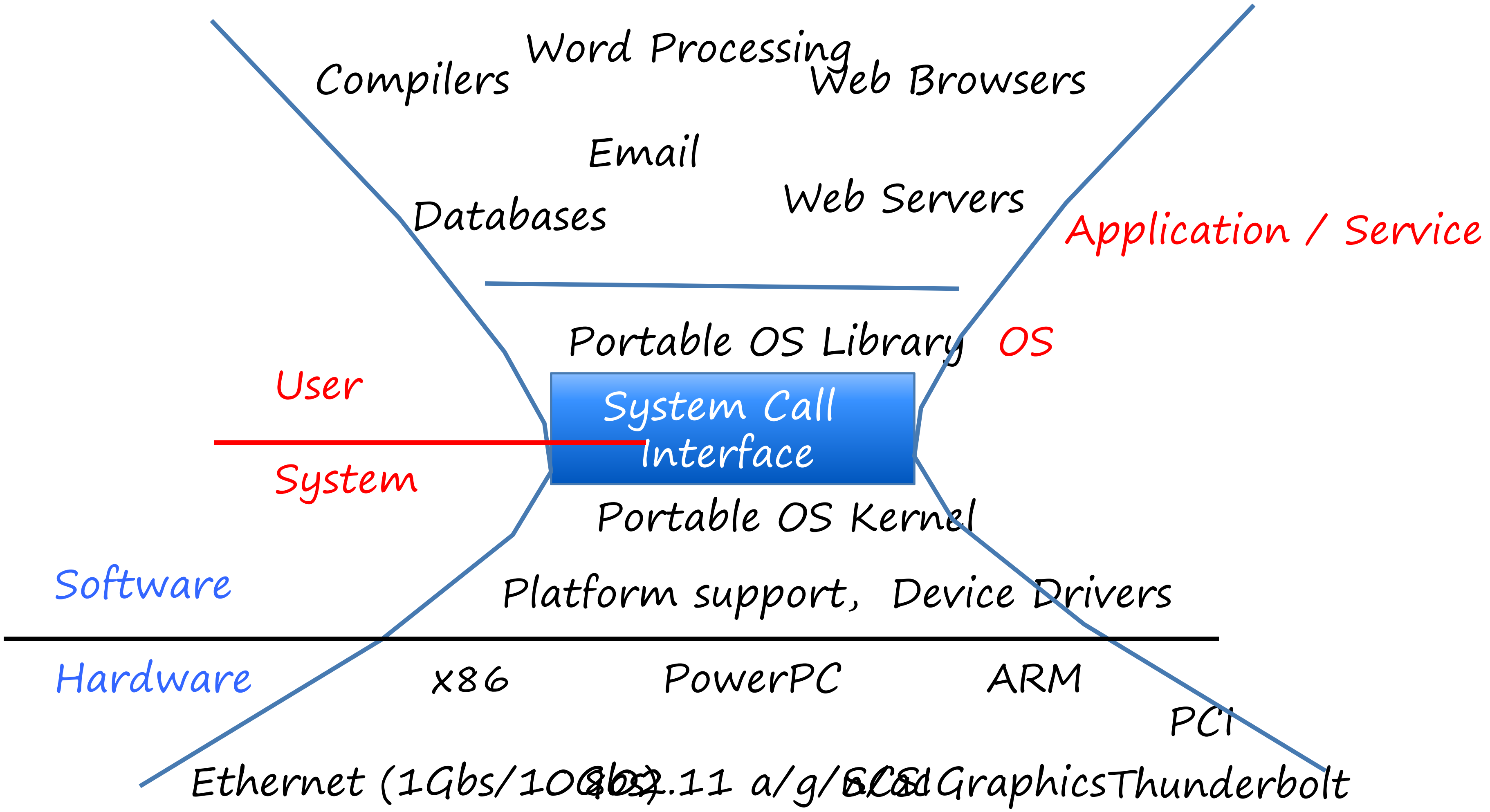
Simple B&B: Switch User Process



Simple B&B: "resume"



Dual Mode Operation



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

Thank You!