

CS162
Operating Systems and
Systems Programming
Lecture 2

Four Fundamental OS Concepts

January 18th, 2024

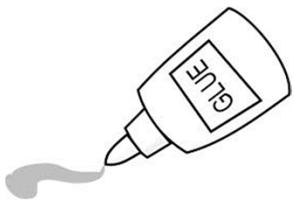
Prof. John Kubiatowicz

<http://cs162.eecs.Berkeley.edu>

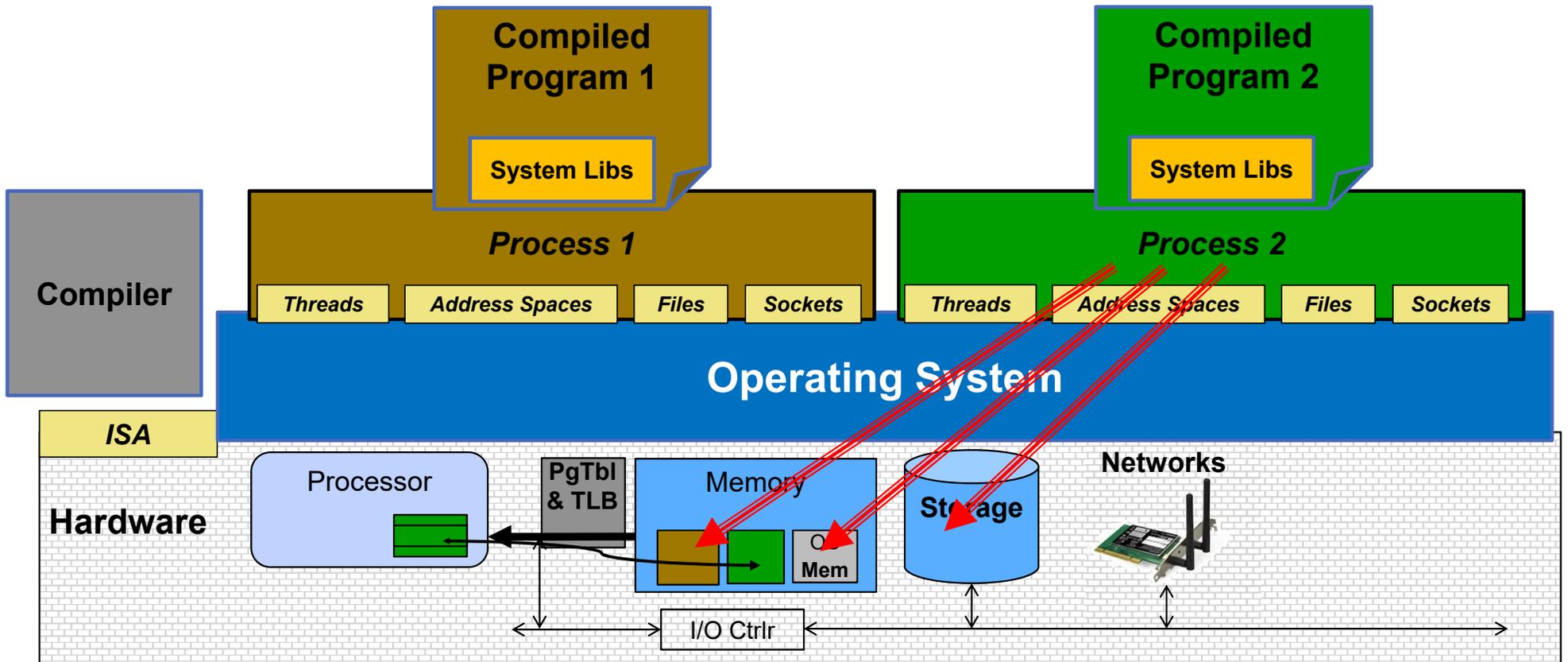
Recall: What is an Operating System?



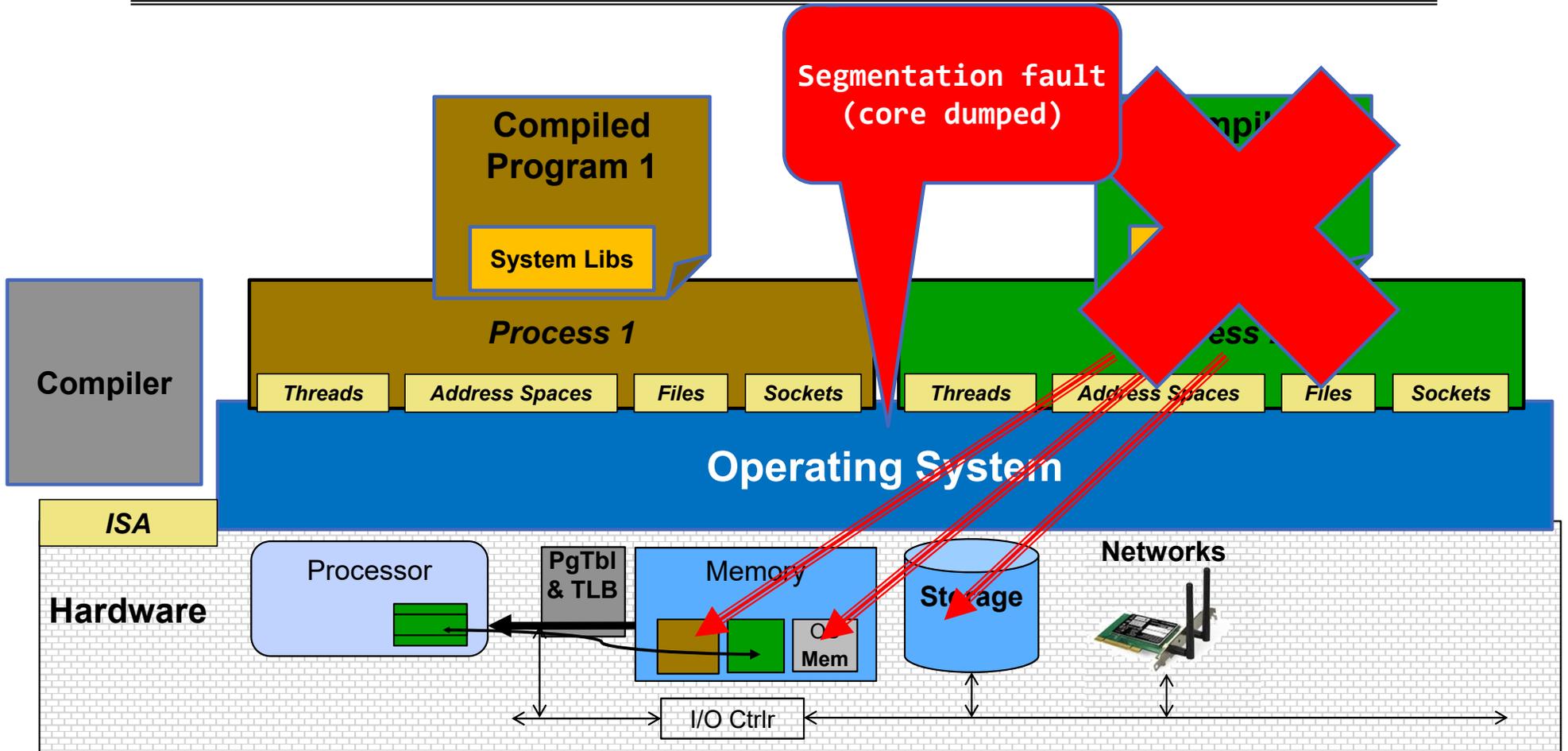
- Referee
 - Manage protection, isolation, and sharing of resources
 - » Resource allocation and communication
- Illusionist
 - Provide clean, easy-to-use abstractions of physical resources
 - » Infinite memory, dedicated machine
 - » Higher level objects: files, users, messages
 - » Masking limitations, virtualization
- Glue
 - Common services
 - » Storage, Window system, Networking
 - » Sharing, Authorization
 - » Look and feel



Recall: OS Protection



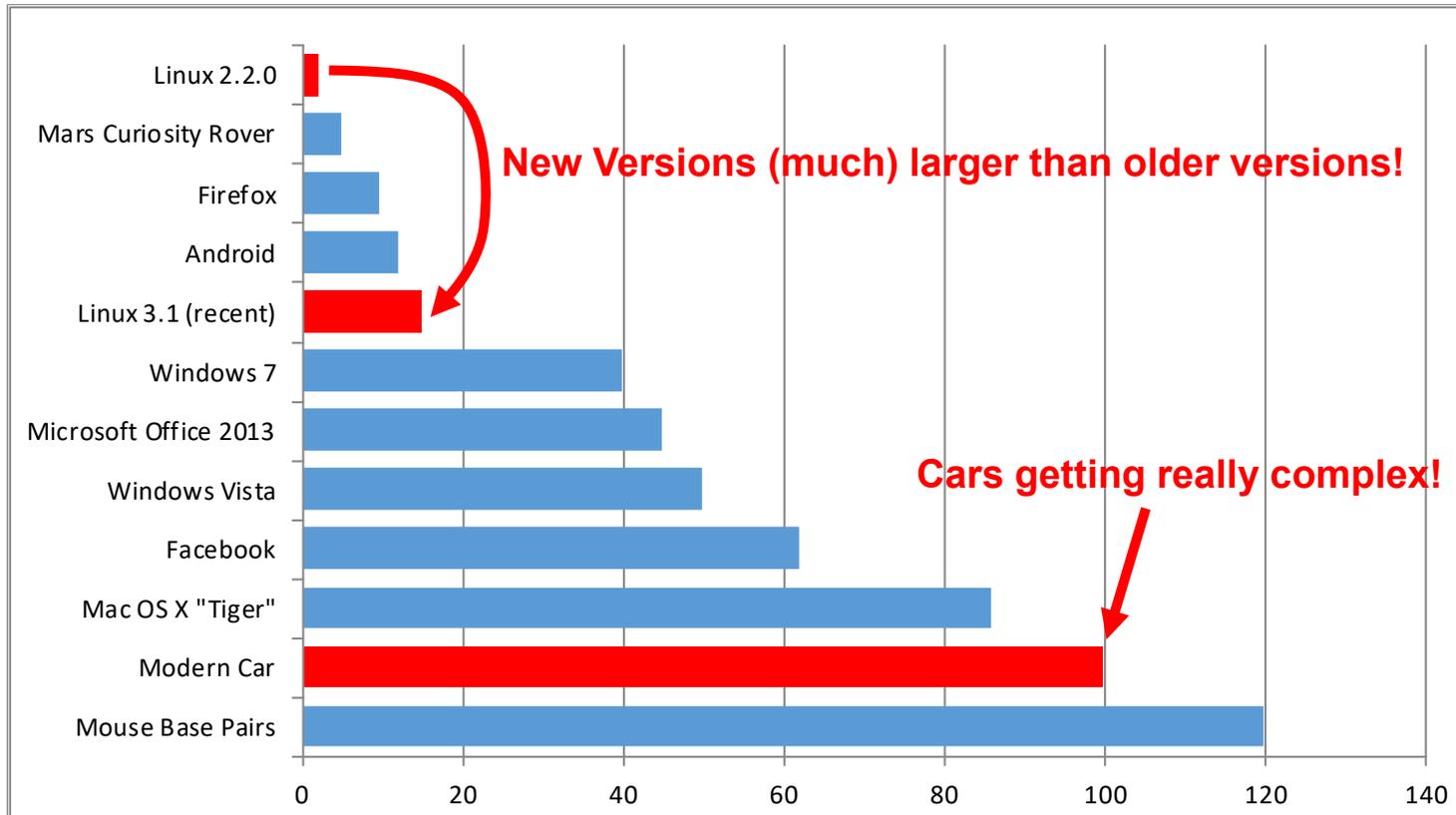
Recall: OS Protection



Challenge: Complexity

- Applications consisting of...
 - ... a variety of software modules that ...
 - ... run on a variety of devices (machines) that
 - » ... implement different hardware architectures
 - » ... run competing applications
 - » ... fail in unexpected ways
 - » ... can be under a variety of attacks
- Not feasible to test software for all possible environments and combinations of components and devices
 - The question is not whether there are bugs but how serious are the bugs!
- Complexity of software keeps growing!

For Instance: Software Complexity keeps growing!

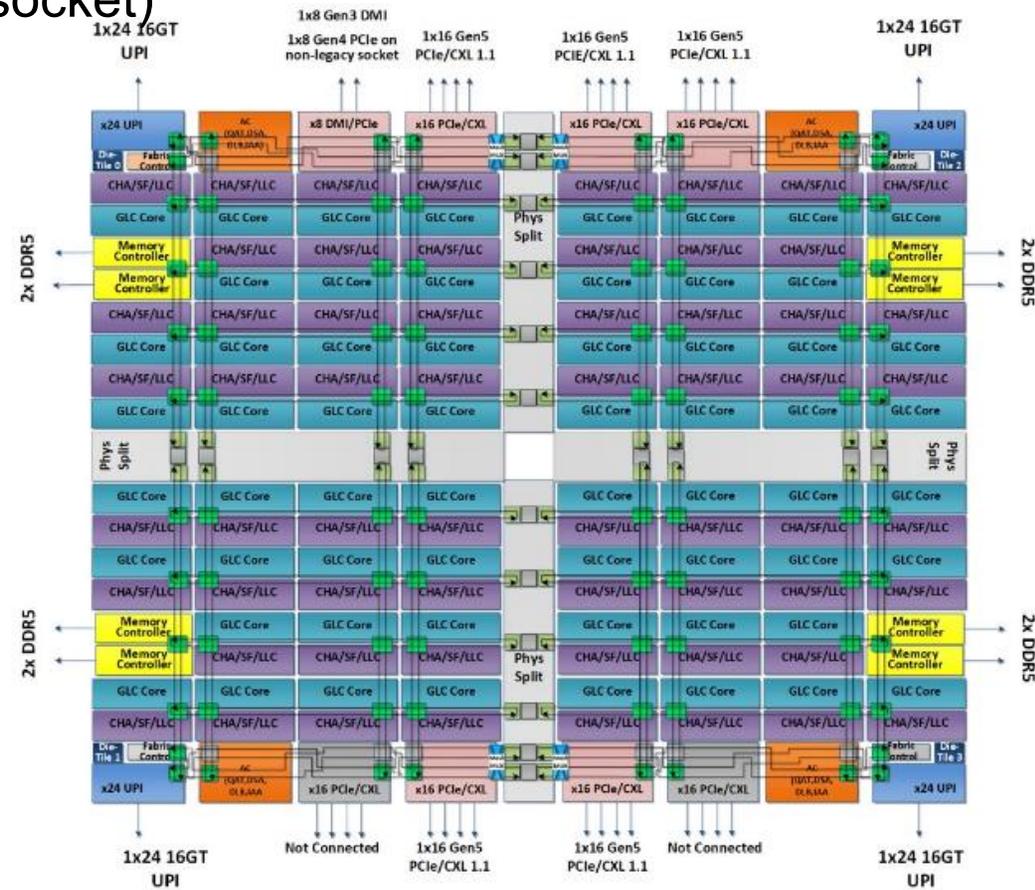


Millions of Lines of Code

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

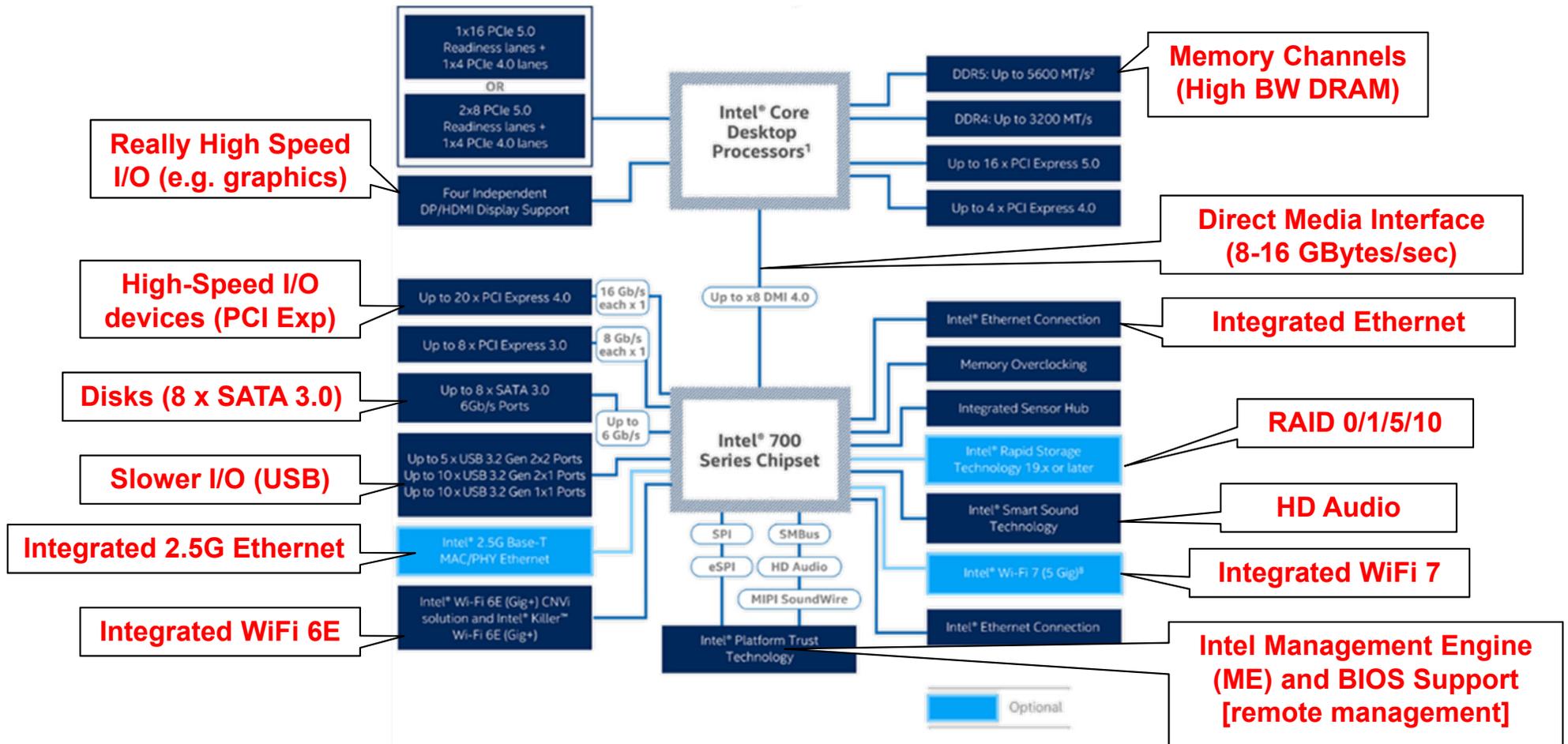
The World Is Parallel: e.g. Intel Sapphire Rapids (2023)

- Up to 60 cores, 120 threads/package (socket)
 - Up to 4 “chipllets” bonded together
- Network:
 - On-chip Mesh Interconnect
 - Fast off-chip network (UPI): directly connects 8-chips
 - 480 cores/shared memory domain!
- Each Core Has:
 - 80 KB L1 Cache
 - 2 MB L2 Cache
 - Fraction of up to 112.5 MB L3 Cache
- DRAM/chips
 - Up to 4 TiB of DDR5 memory
- Many Accelerators of different types
 - Graphics, Encryption, AI, Security



Sapphire Rapids 4-chiplet single package

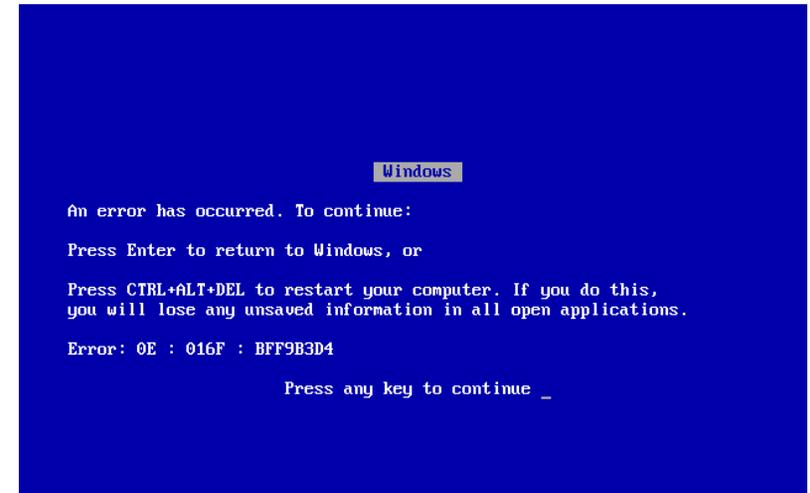
HW Functionality comes with great complexity!



Intel 700 Chipset I/O Configuration

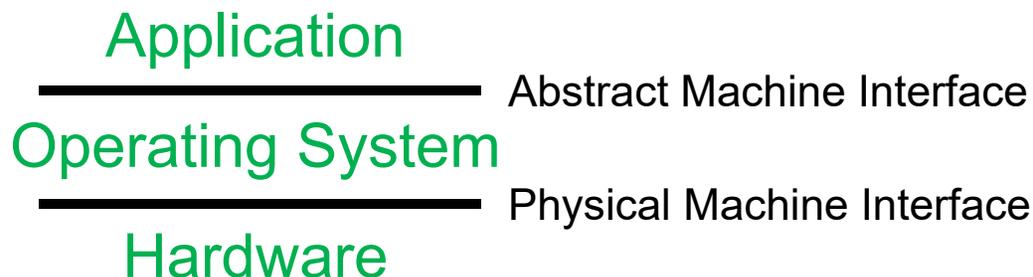
Complexity leaks into OS if not properly designed:

- Third-party device drivers are one of the most unreliable aspects of OS
 - Poorly written by non-stake-holders
 - Ironically, the attempt to provide clean abstractions can lead to crashes!
- Holes in security model or bugs in OS lead to instability and privacy breaches
 - Great Example: Meltdown (2017)
 - » Extract data from protected kernel space!
- Version skew on Libraries can lead to problems with application execution
- Data breaches, DDOS attacks, timing channels....
 - Heartbleed (SSL)



OS Abstracts Underlying Hardware to help Tame Complexity

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

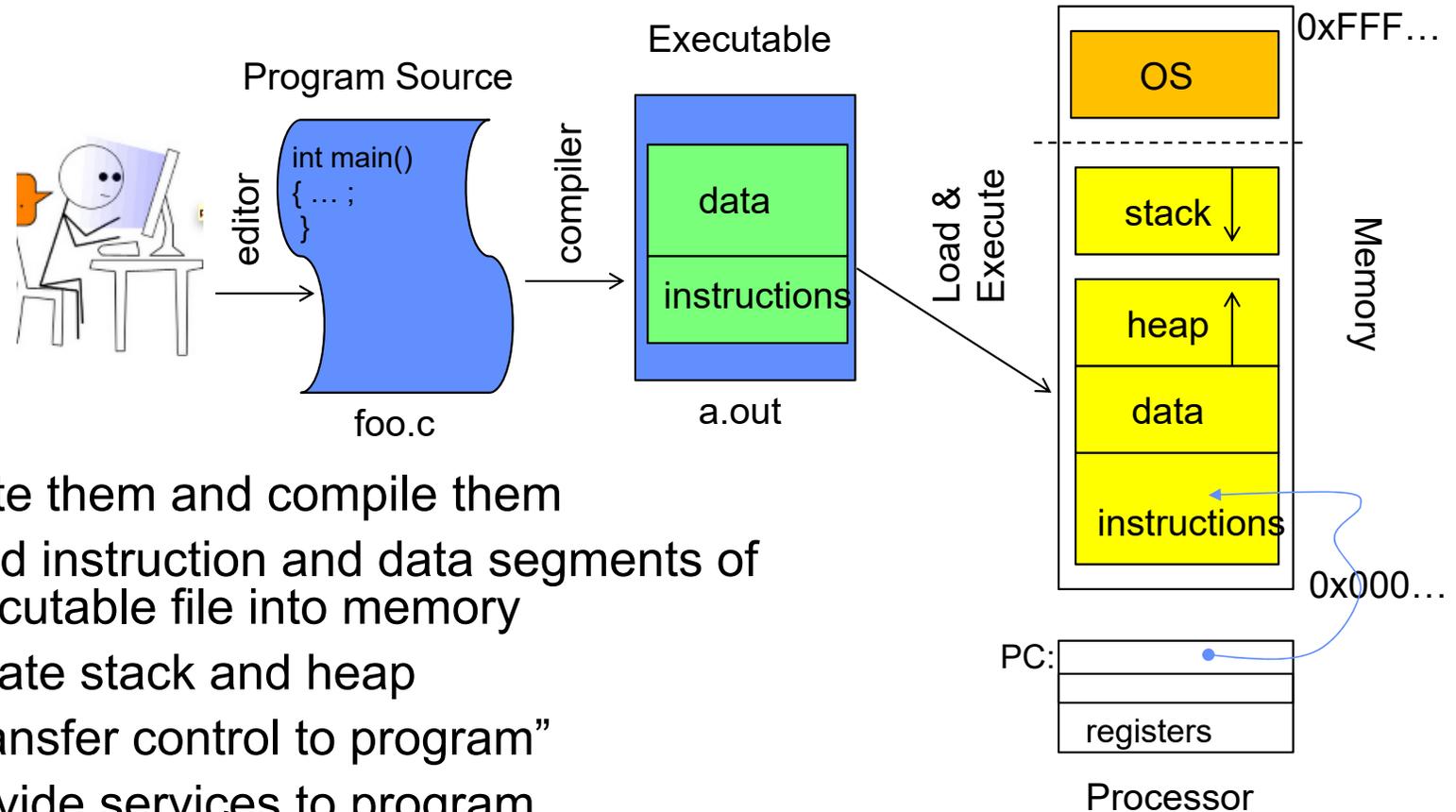


- OS as an *Illusionist*:
 - Remove software/hardware quirks (*fight complexity*)
 - Optimize for convenience, utilization, reliability, ... (*help the programmer*)
- For any OS area (e.g. file systems, virtual memory, networking, scheduling):
 - What hardware interface to handle? (physical reality)
 - What's software interface to provide? (nicer abstraction)

Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
 - Fully describes program state
 - 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

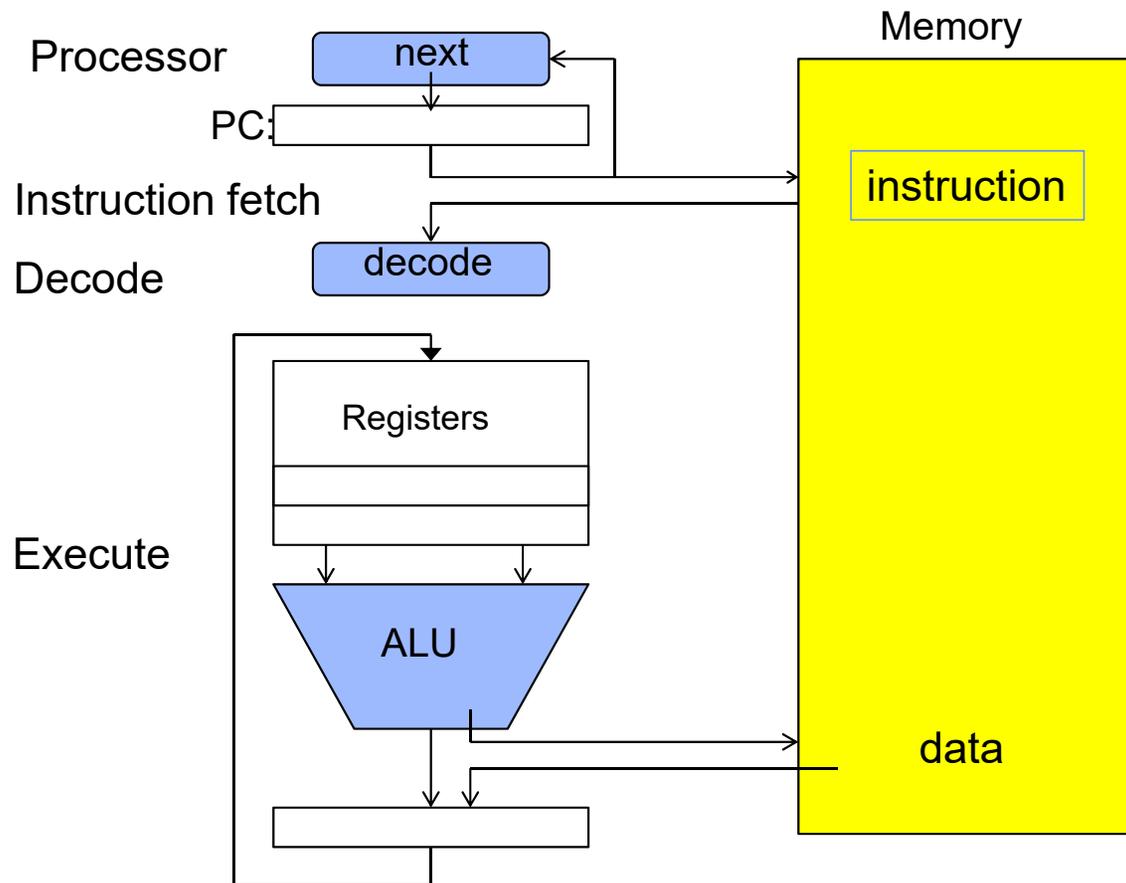
OS Bottom Line: Run Programs



- Write them and compile them
- 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 (61C): Instruction Fetch/Decode/Execute

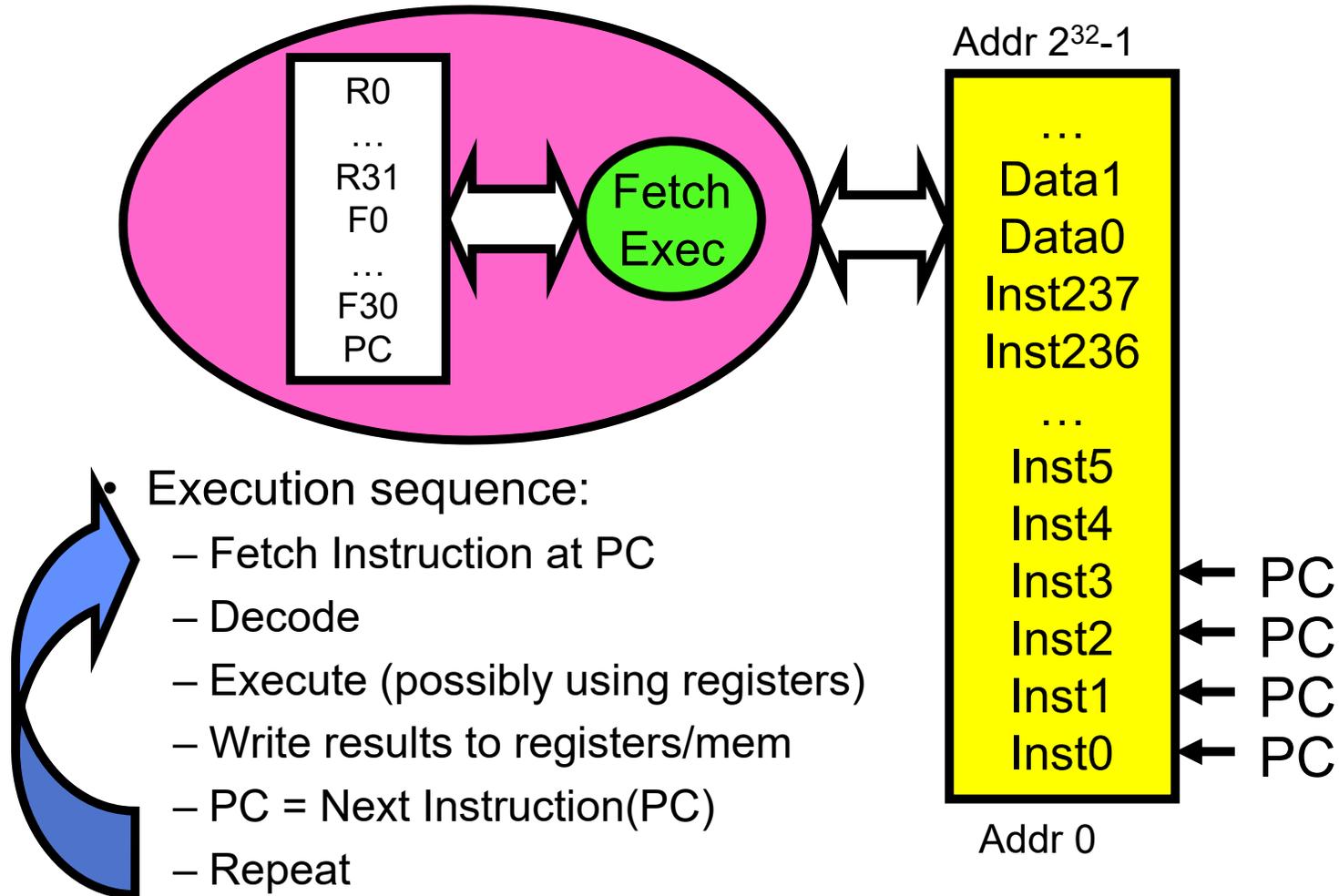
The instruction cycle



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

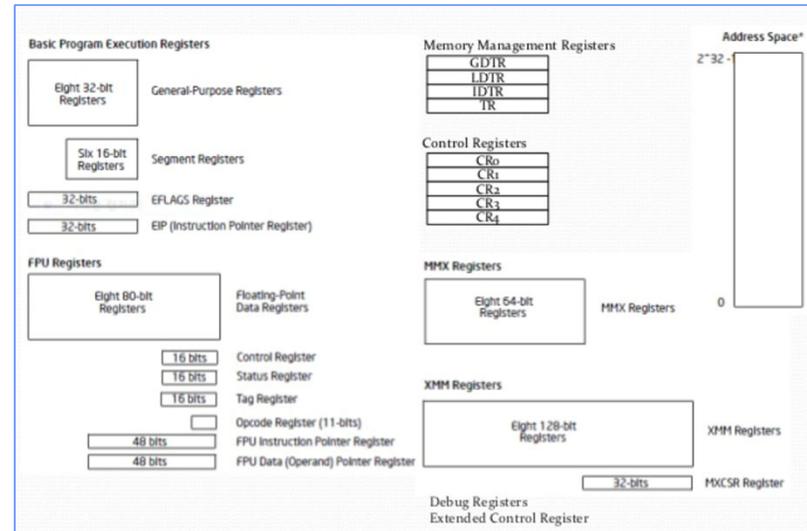
Recall (61C): What happens during program execution?



Registers: RISC-V \Rightarrow 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

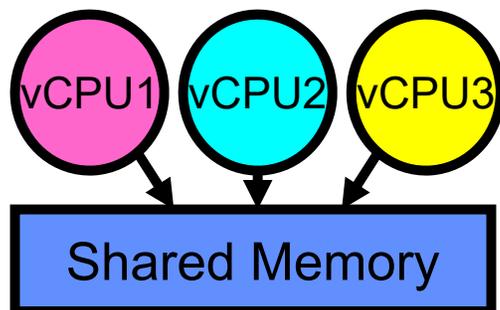
**Load/Store Arch (RISC-V)
with software conventions**



**Complex mem-mem arch (x86) with
specialized registers and “segments”**

- CS61C does RISC-V. Will need to learn x86...
- Section will cover this architecture

Illusion of Multiple Processors with single core



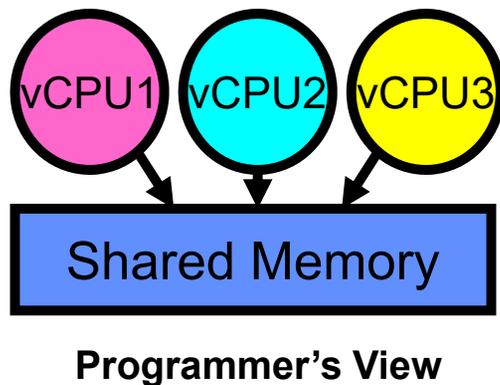
Programmer's View



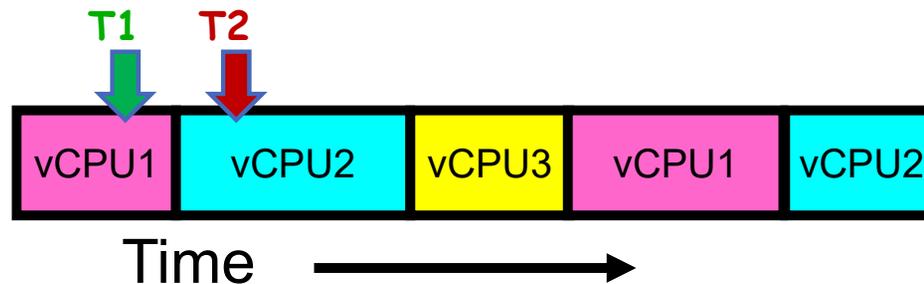
- Assume a **single processor (core)**. How do we provide the illusion of multiple processors?
 - Multiplex in time!
- Threads are **virtual cores**

- 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)

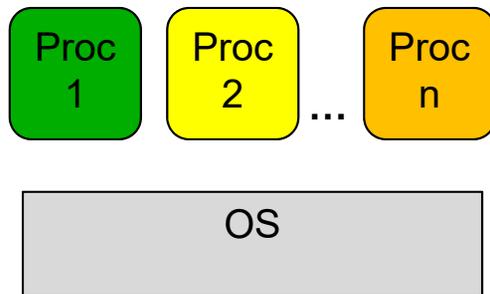


- Consider:
 - At T1: vCPU1 on real core, vCPU2 in memory
 - At T2: vCPU2 on real core, vCPU1 in memory

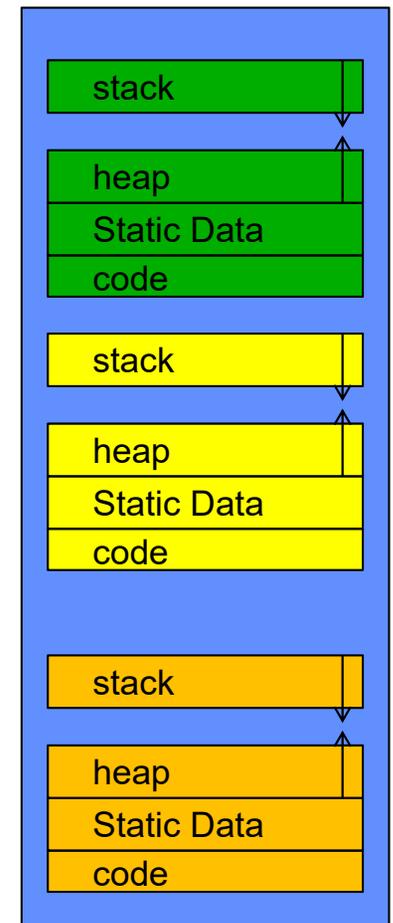


- What happened?
 - OS Ran [how?]
 - Saved PC, SP, ... in vCPU1's thread control block (memory)
 - Loaded PC, SP, ... from vCPU2's TCB, jumped to PC
- What triggered this switch?
 - Timer, voluntary yield, I/O, other things we will discuss

Multiprogramming - Multiple Threads of Control



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



Administrivia: Getting started

- Should be working on Homework 0 already! ⇒ **Due Wednesday (1/24)**
 - cs162-xx account, Github account, registration survey
 - VM environment for the course
 - Get familiar with all the cs162 tools, submitting to autograder via git
- **Start Project 0 Monday!**
 - **To be done on your own – like a homework**
- Reminder of Resources for you (look at Resources tab on homepage!)
 - Quick tutorial about C: Ladder
 - O'Reilly Books on: C language, Git environment, Rust (for later)
- Slip days: I'd bank these and not spend them right away!
 - **You have 4 slip days for homework**
 - **You have 5 slip days for projects**
 - (Very) Limited credit when late and run out of slip days
- Please don't use slip days on Homework 0 / Project 0!
 - You will start off on wrong foot!

Administrivia (Con't)

- Monday is an optional REVIEW session for C
 - Time and location/Zoom link TBA
 - May be recorded
- Class size is fixed at 404 students, 13 sections
 - Will be moving more students from waitlist \Rightarrow class as students drop
 - If you are on waitlist, *assume you could get into the class at any time in next week or so!*
 - Keep up with work (until you drop or we close the class)
 - » If cannot get credentials for autograder, contact us (I believe we have an Ed thread for that!)
- **Friday (1/26) is drop day for this class!**
 - **Very hard to drop afterwards...**
 - **Please drop sooner if you are going to anyway \Rightarrow Let someone else in!**

CS 162 Collaboration Policy



- Explaining a concept to someone in another group
- Discussing algorithms/testing strategies with other groups
- Discussing debugging approaches with other groups
- Searching online for generic algorithms (e.g., hash table)

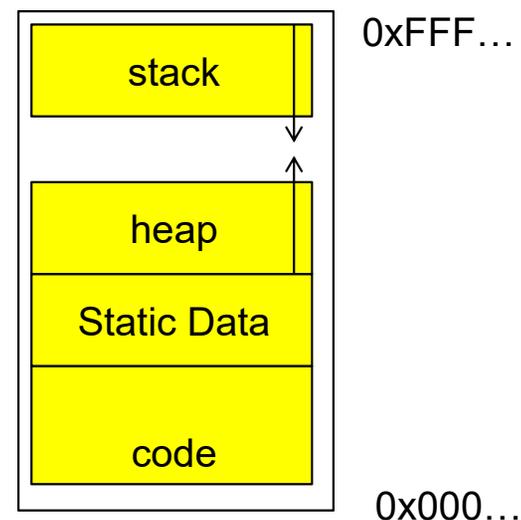


- Sharing code or test cases with another group
- Copying OR reading another group's code or test cases
- Copying OR reading online code or test cases from prior years
- Helping someone in another group to debug their code

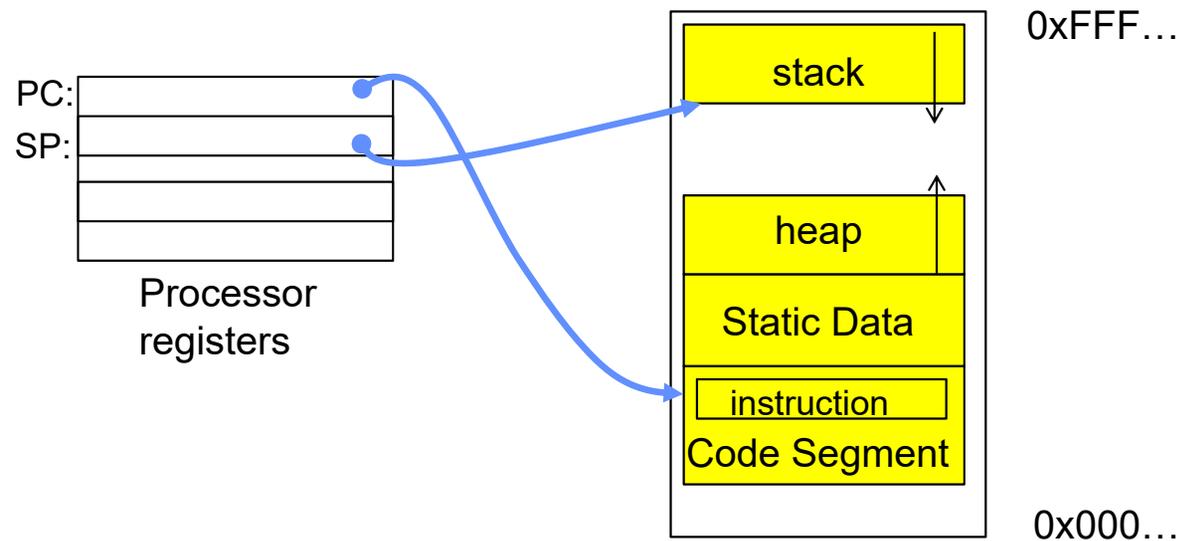
- We compare all project submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders
- Don't put a friend in a bad position by asking for help that they shouldn't give!

Second OS Concept: Address Space

- **Address space** \Rightarrow 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 ignores writes
 - Perhaps causes I/O operation
 - » (Memory-mapped I/O)
 - Perhaps causes exception (fault)
 - Communicates with another program
 -



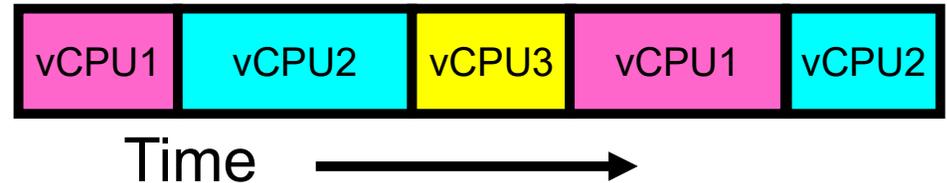
Address Space: In a Picture



- 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...**



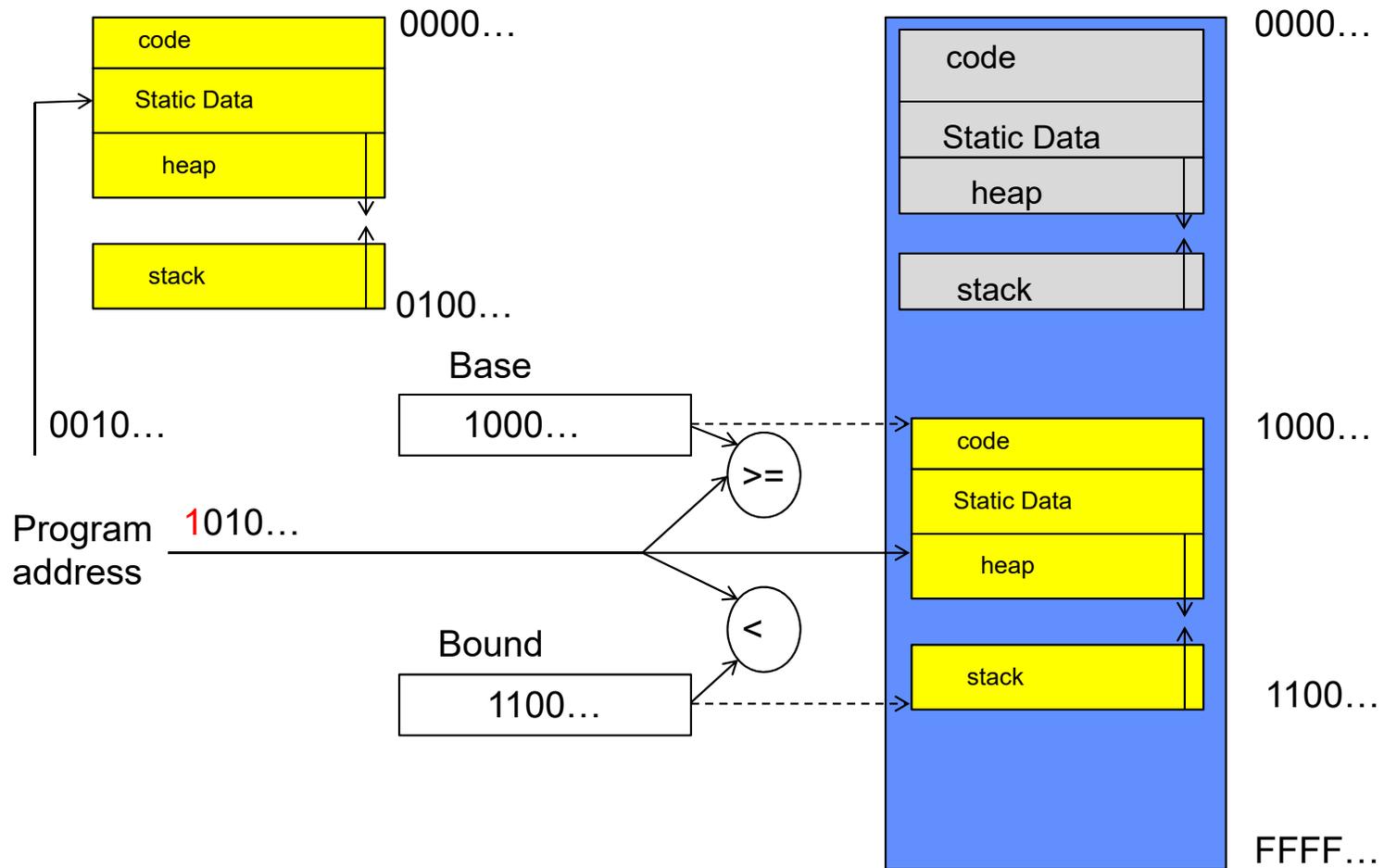
Simple Multiplexing has no Protection!

- 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

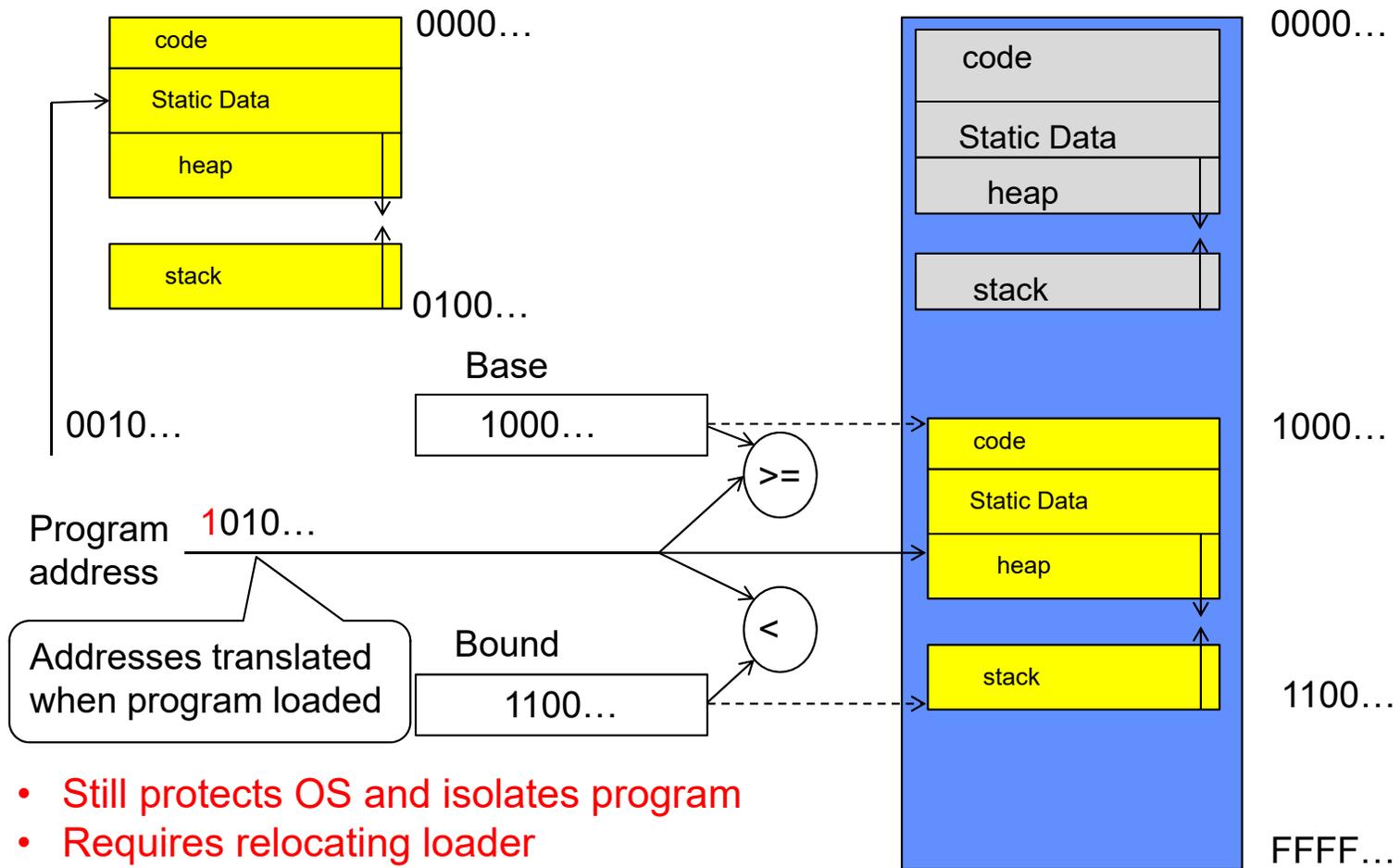
What can the hardware do to help the
OS protect itself from programs???

How about: Limit the *ACCESS* that threads have to memory

Simple Protection: Base and Bound (B&B)

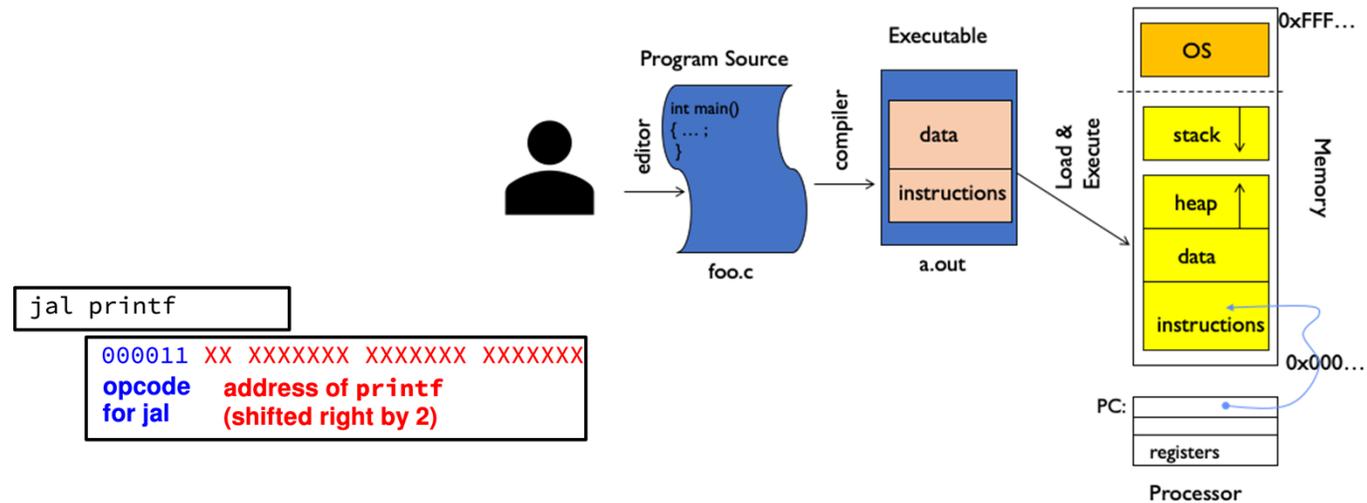


Simple Protection: Base and Bound (B&B)



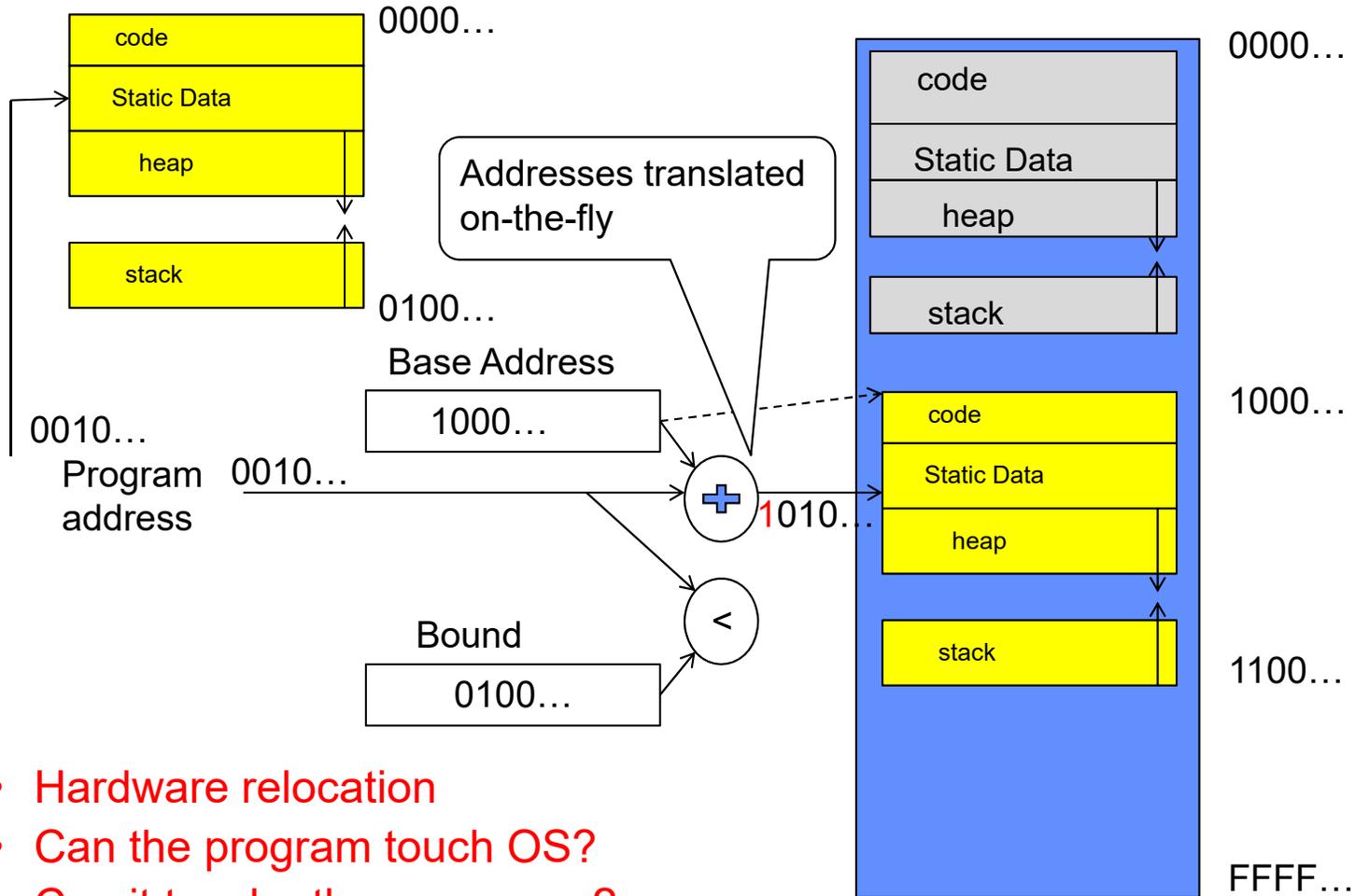
- Still protects OS and isolates program
- Requires relocating loader
- No addition on address path

61C Review: Relocation



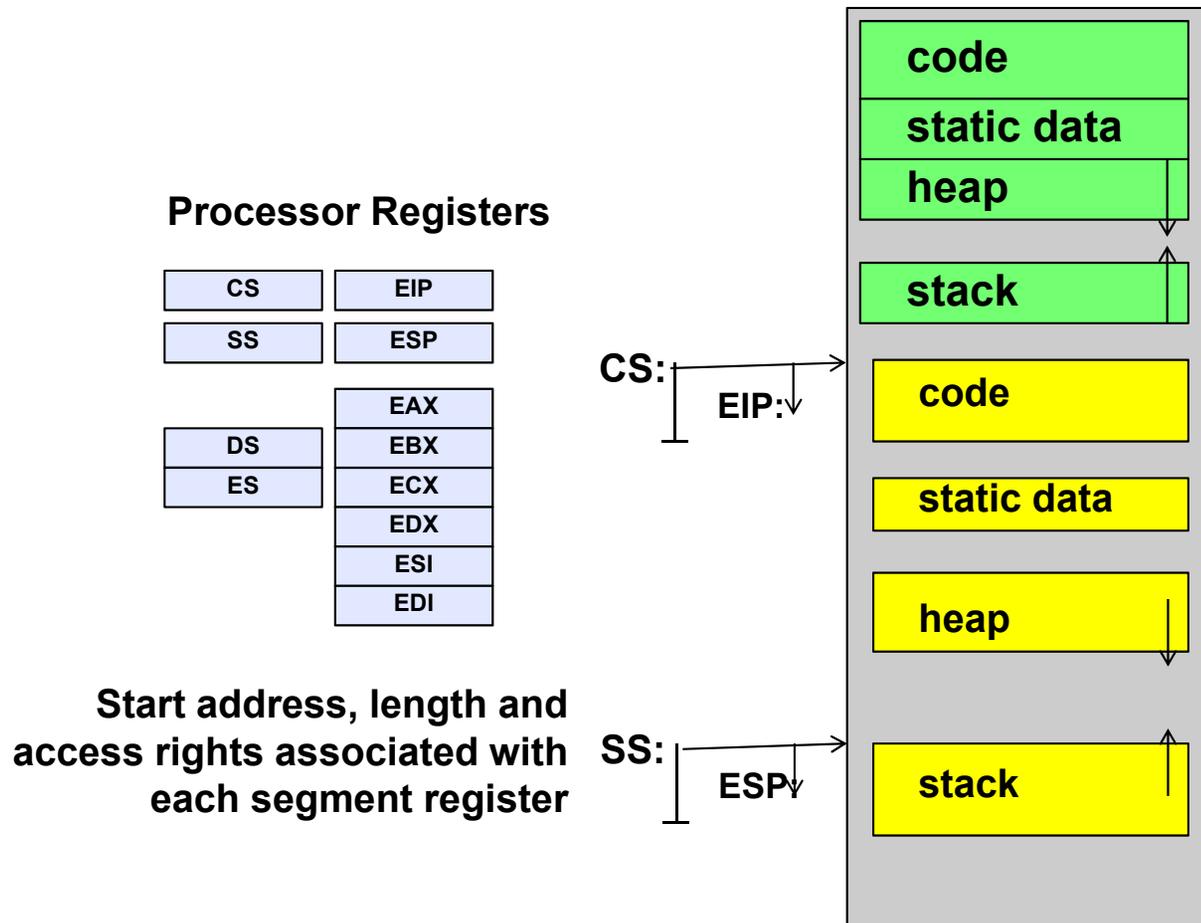
- Compiled `.obj` file linked together in an `.exe`
- All address in the `.exe` are as if it were loaded at memory address `00000000`
- File contains a list of all the addresses that need to be adjusted when it is “relocated” to somewhere else.

Simple address translation with Base and Bound



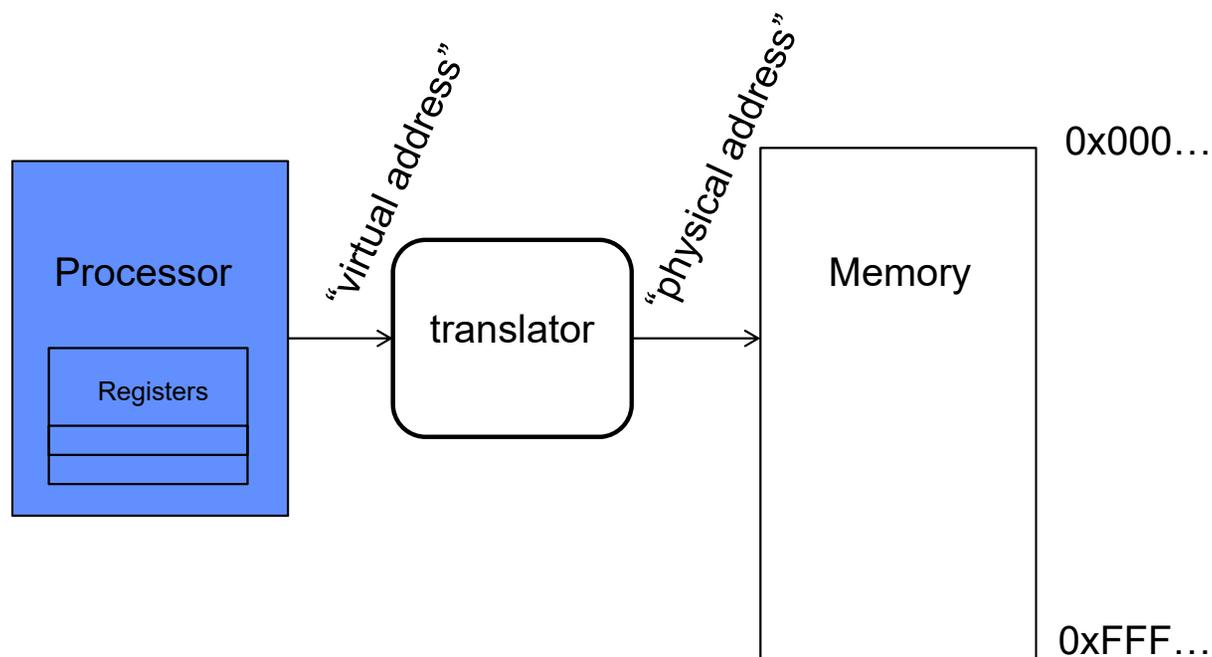
- Hardware relocation
- Can the program touch OS?
- Can it touch other programs?

x86 – segments and stacks



Another idea: Address Space Translation

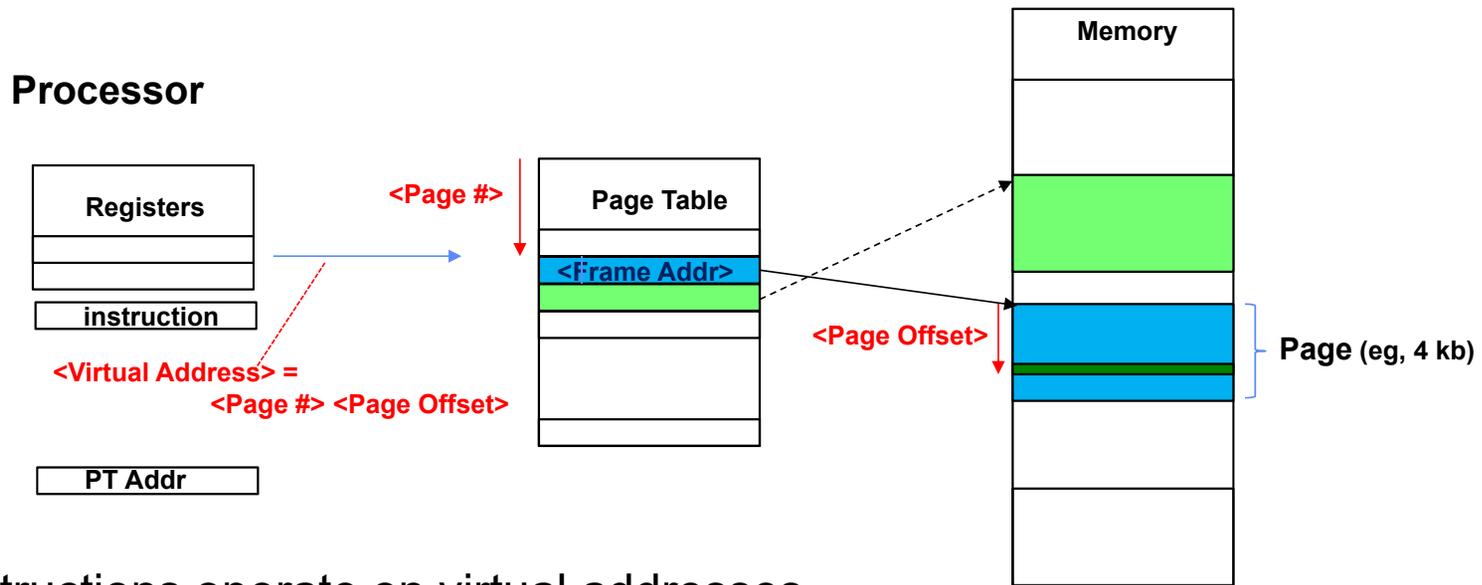
- Program operates in an address space that is distinct from the physical memory space of the machine



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?
- 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
 - Treat memory as page size frames and put any page into any frame ...
- Another cs61C review...

Paged Virtual Address (from 61C)

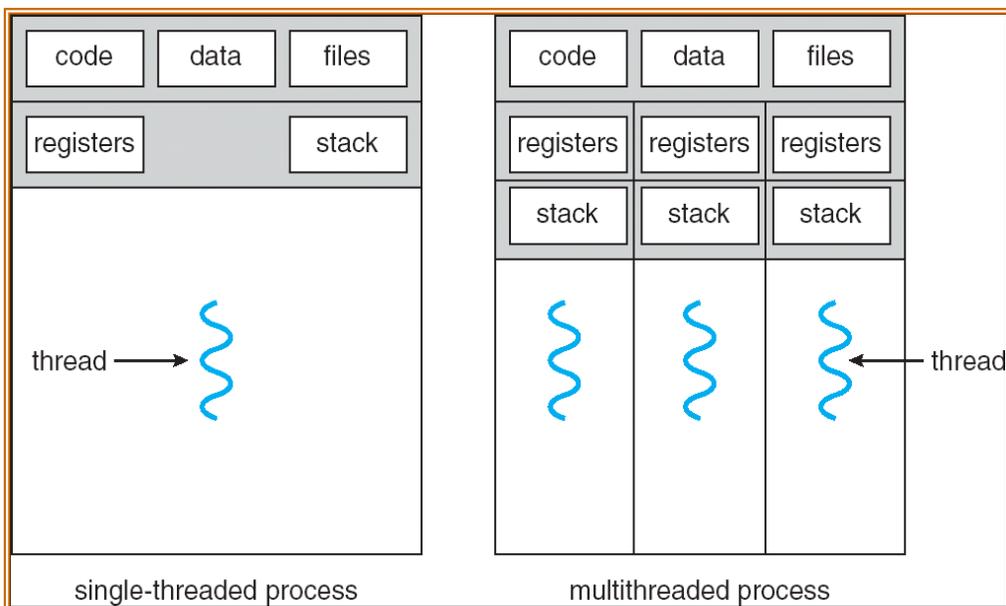


- Instructions operate on virtual addresses
 - Instruction address, load/store data address
- Translated to a physical address through a Page Table by the hardware
- Any Page of address space can be in any (page sized) frame in memory
 - Or not-present (access generates a page fault)
- Special register 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)
 - 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
- Fundamental tradeoff between protection and efficiency
 - Communication easier *within* a process
 - Communication harder *between* processes

Single and Multithreaded Processes



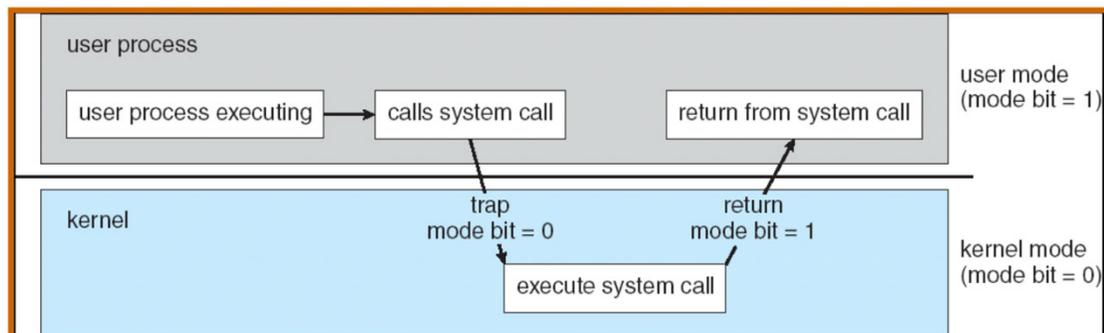
- Threads encapsulate **concurrency**:
 - “Active” component
- Address spaces encapsulate **protection**:
 - “Passive” component
 - Keeps buggy programs from crashing the system
- Why have multiple threads per address space?
 - Parallelism: take advantage of actual hardware parallelism (e.g. multicore)
 - Concurrency: ease of handling I/O and other simultaneous events

Protection and Isolation

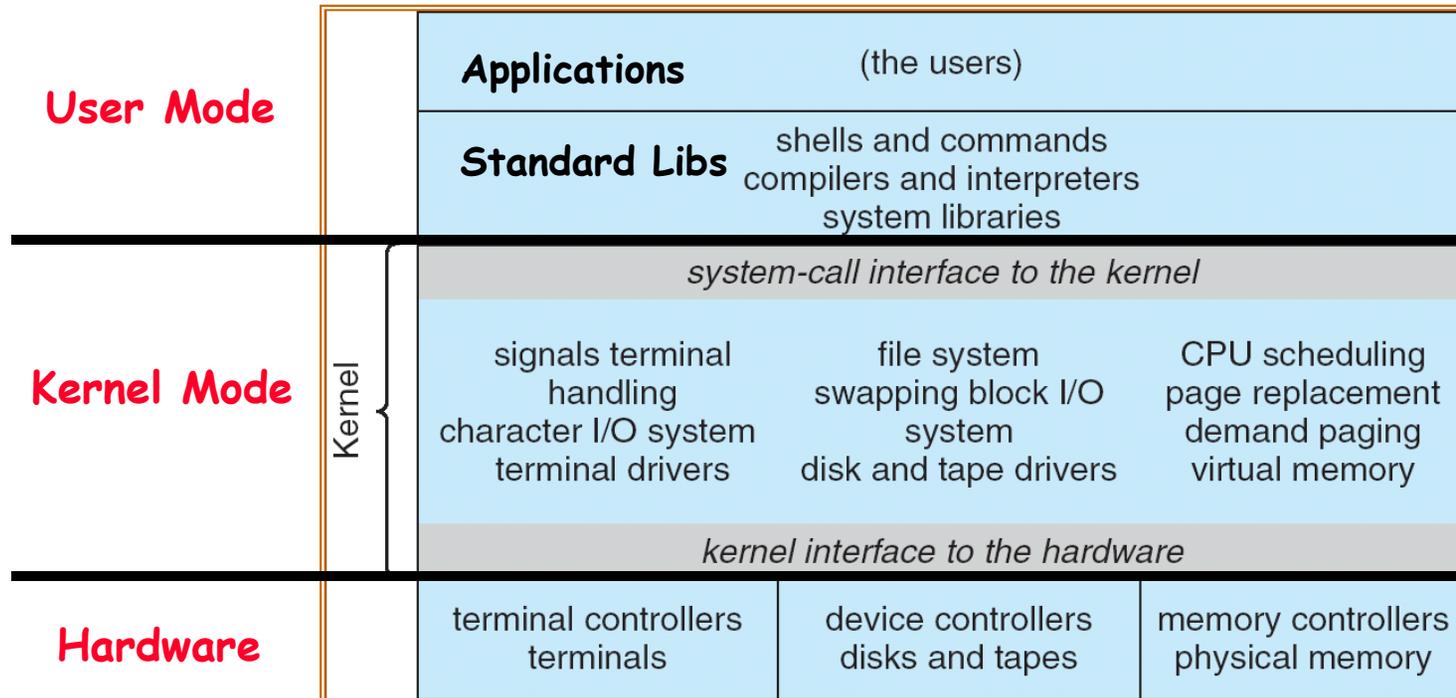
- Why Do We Need Processes??
 - Reliability: bugs can only overwrite memory of process they are in
 - Security and privacy: malicious or compromised process can't read or write other process' data
 - (to some degree) Fairness: enforce shares of disk, CPU
- Mechanisms:
 - Address translation: address 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

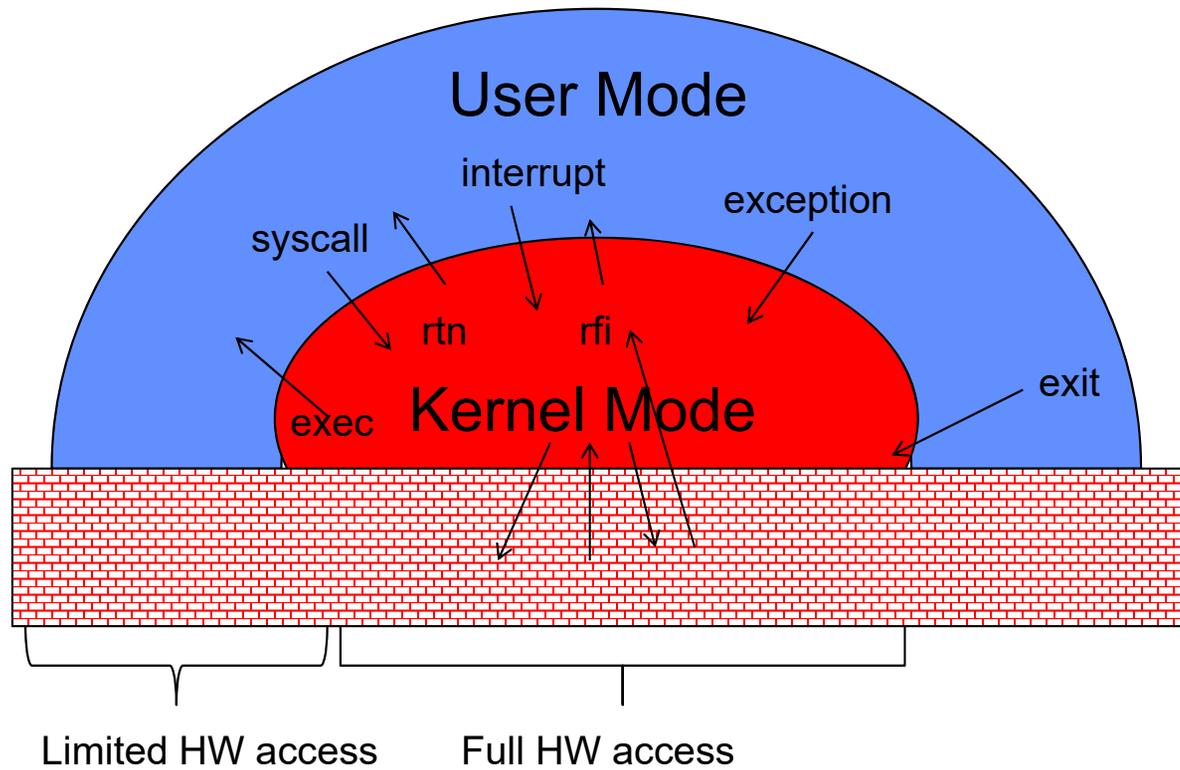
- **Hardware** provides at least two modes (at least 1 mode bit):
 1. **Kernel Mode** (or “supervisor” mode)
 2. **User Mode**
- Certain operations are **prohibited** when running in user mode
 - Changing the page table pointer, disabling interrupts, interacting directly w/ hardware, writing to kernel memory
- Carefully controlled transitions between user mode and kernel mode
 - System calls, interrupts, exceptions



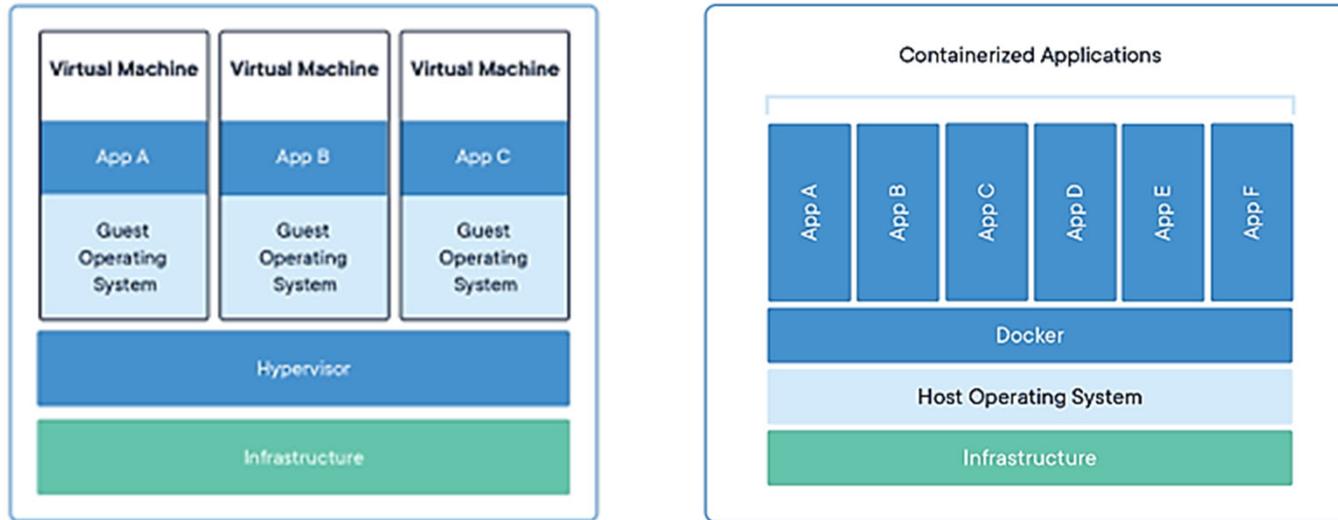
For example: UNIX System Structure



User/Kernel (Privileged) Mode

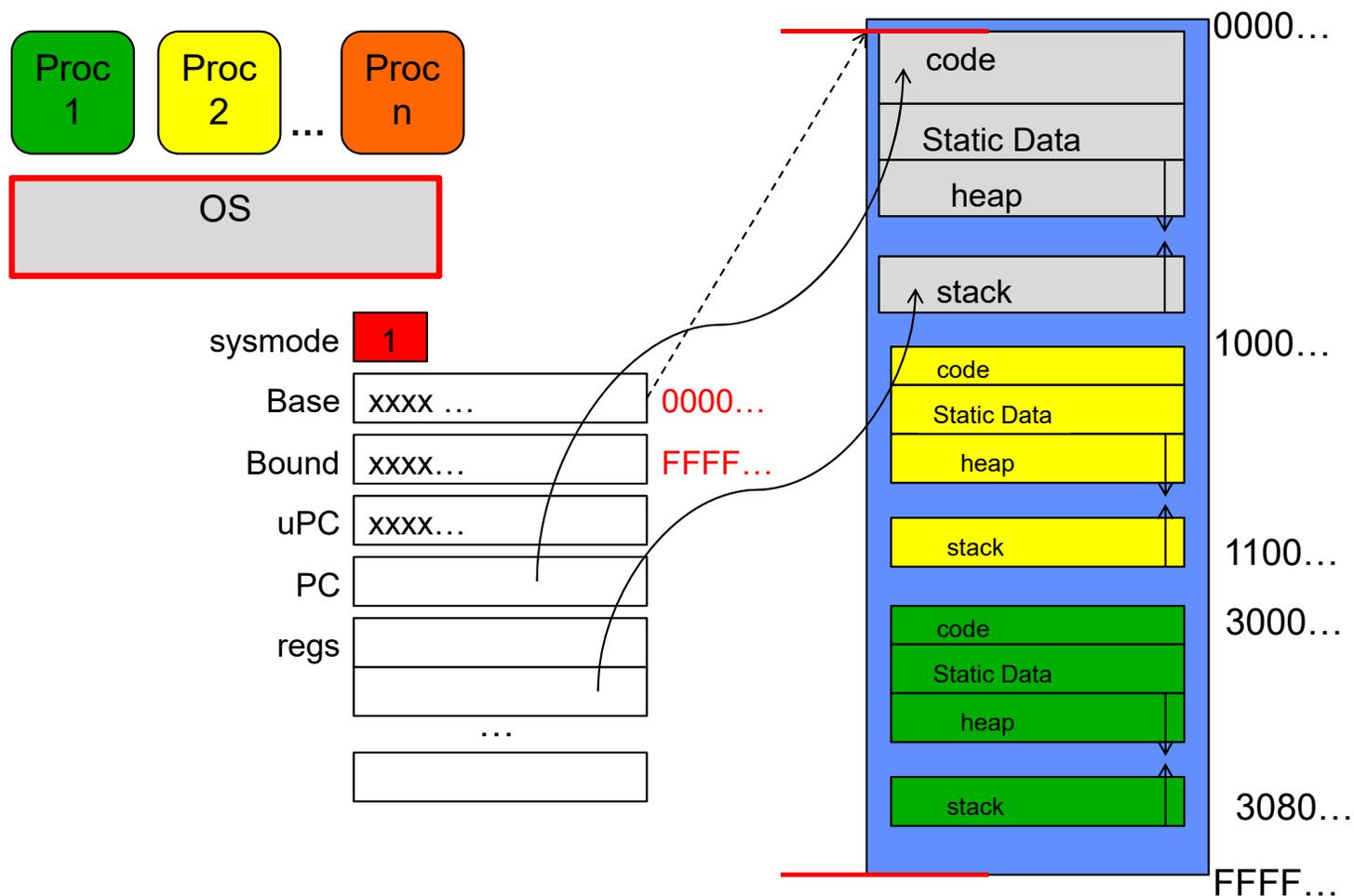


Additional Layers of Protection for Modern Systems

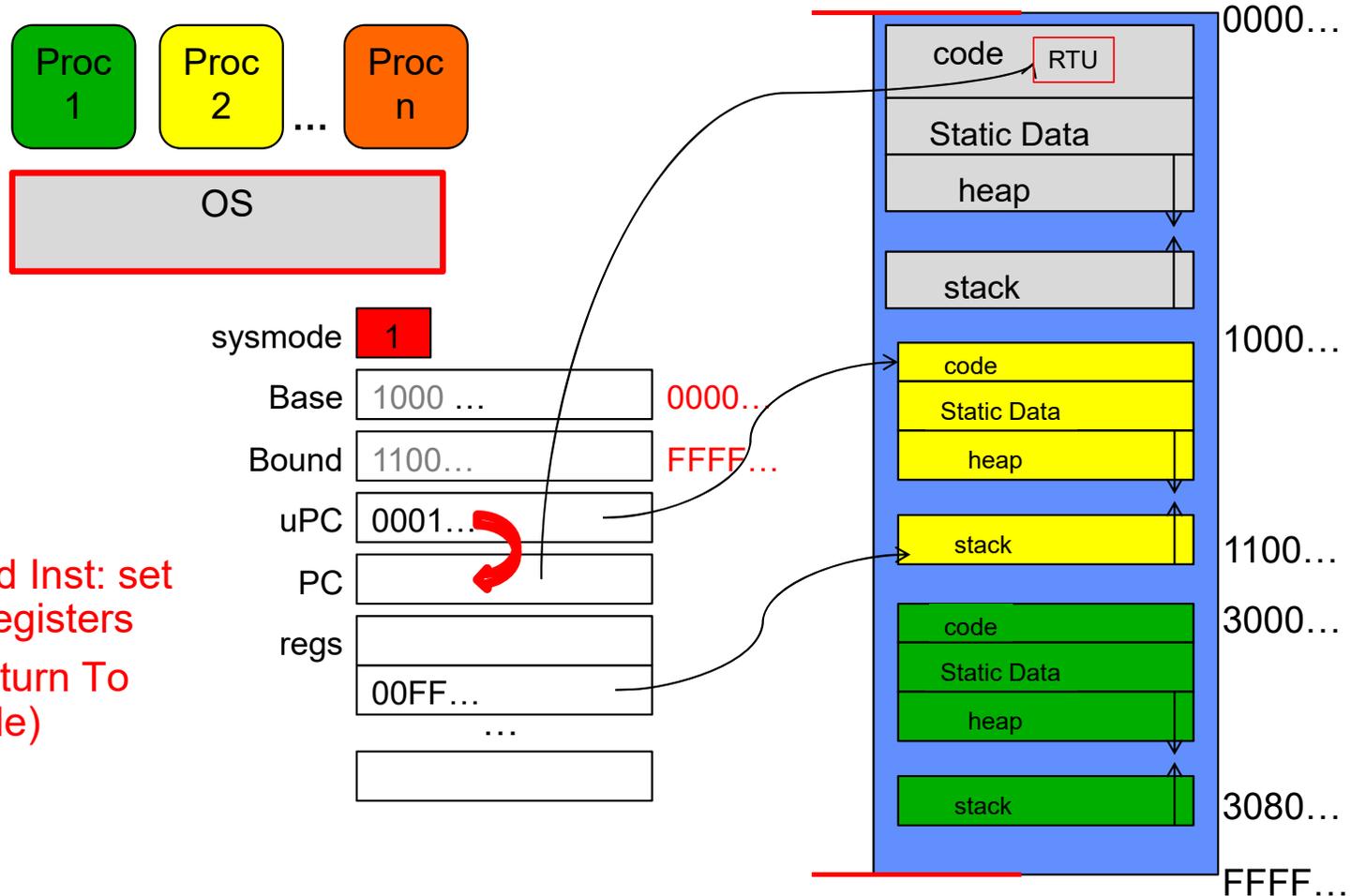


- Additional layers of protection through virtual machines or containers
 - Run a complete operating system in a virtual machine
 - Package all the libraries associated with an app into a container for execution
- More on these ideas later in the class

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

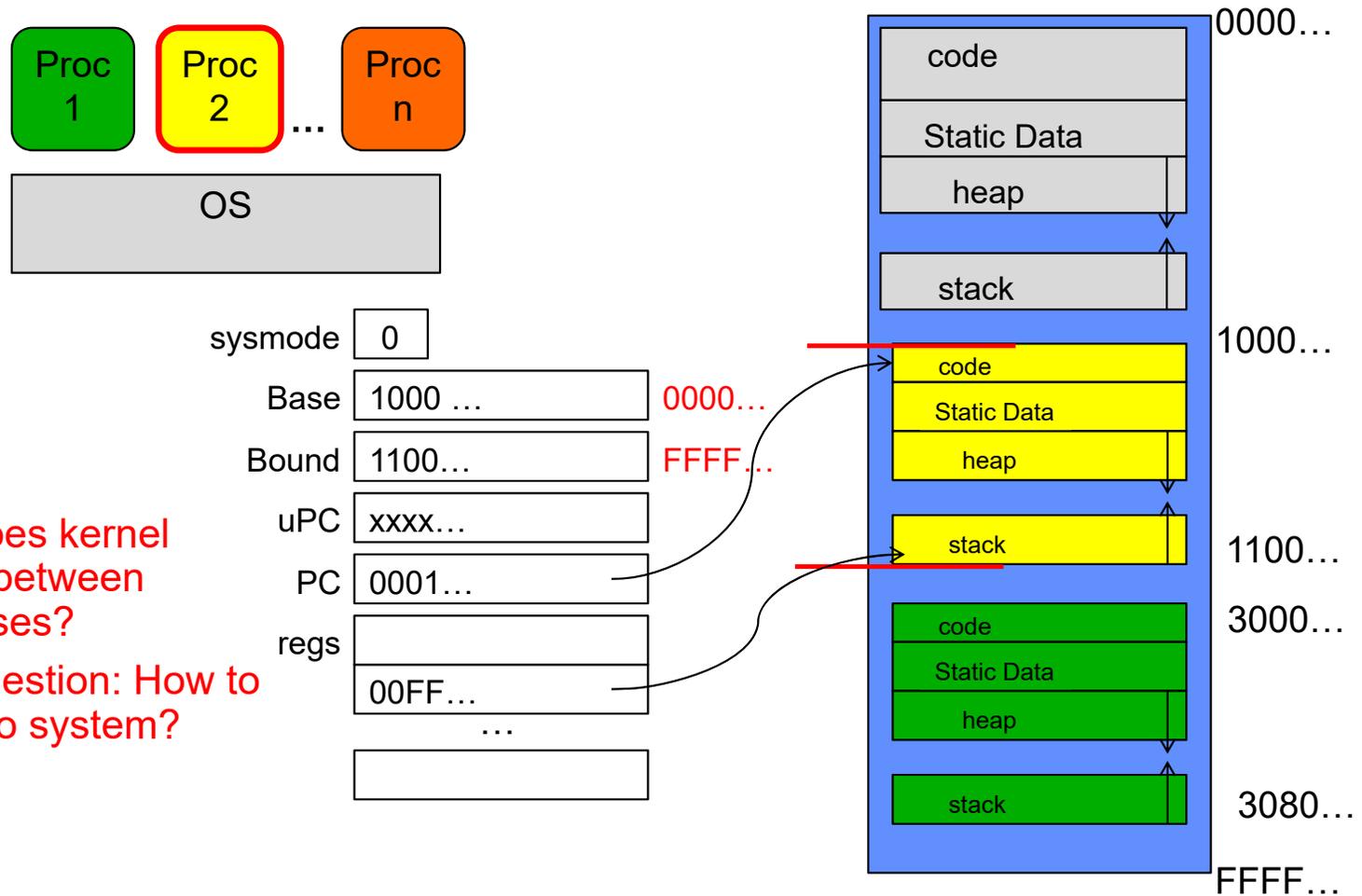


Simple B&B: OS gets ready to execute process



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

Simple B&B: User Code Running



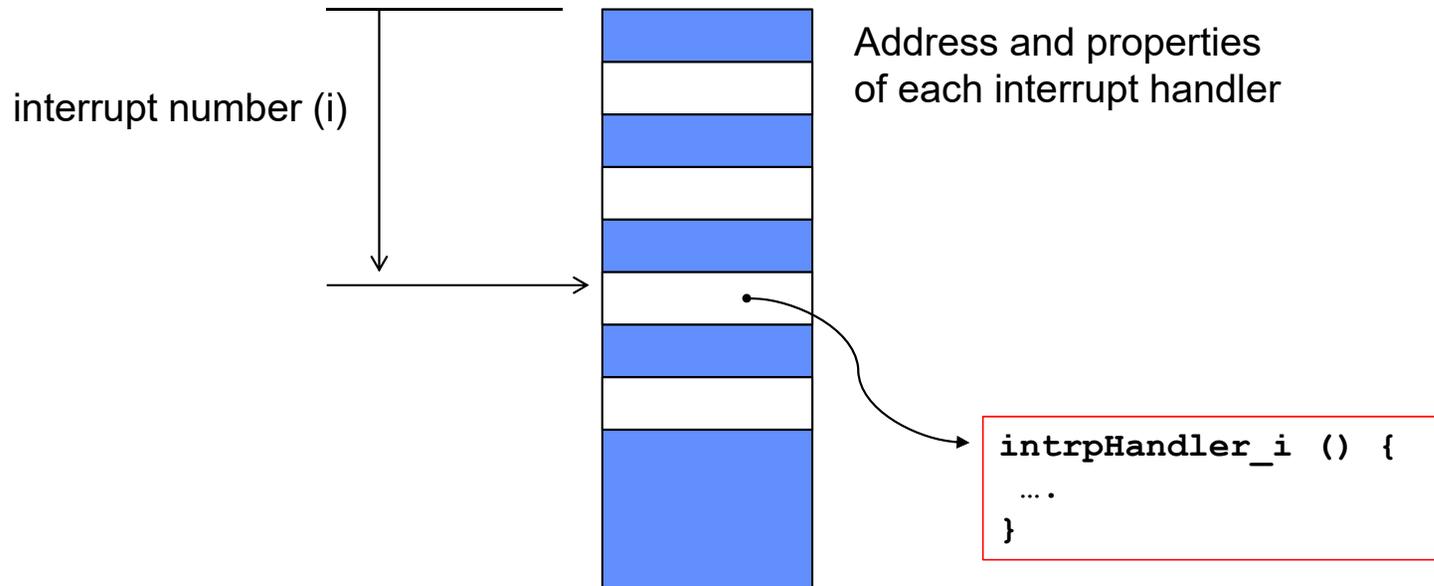
- How does kernel switch between processes?
- First question: How to return to system?

3 types of User \Rightarrow Kernel 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?

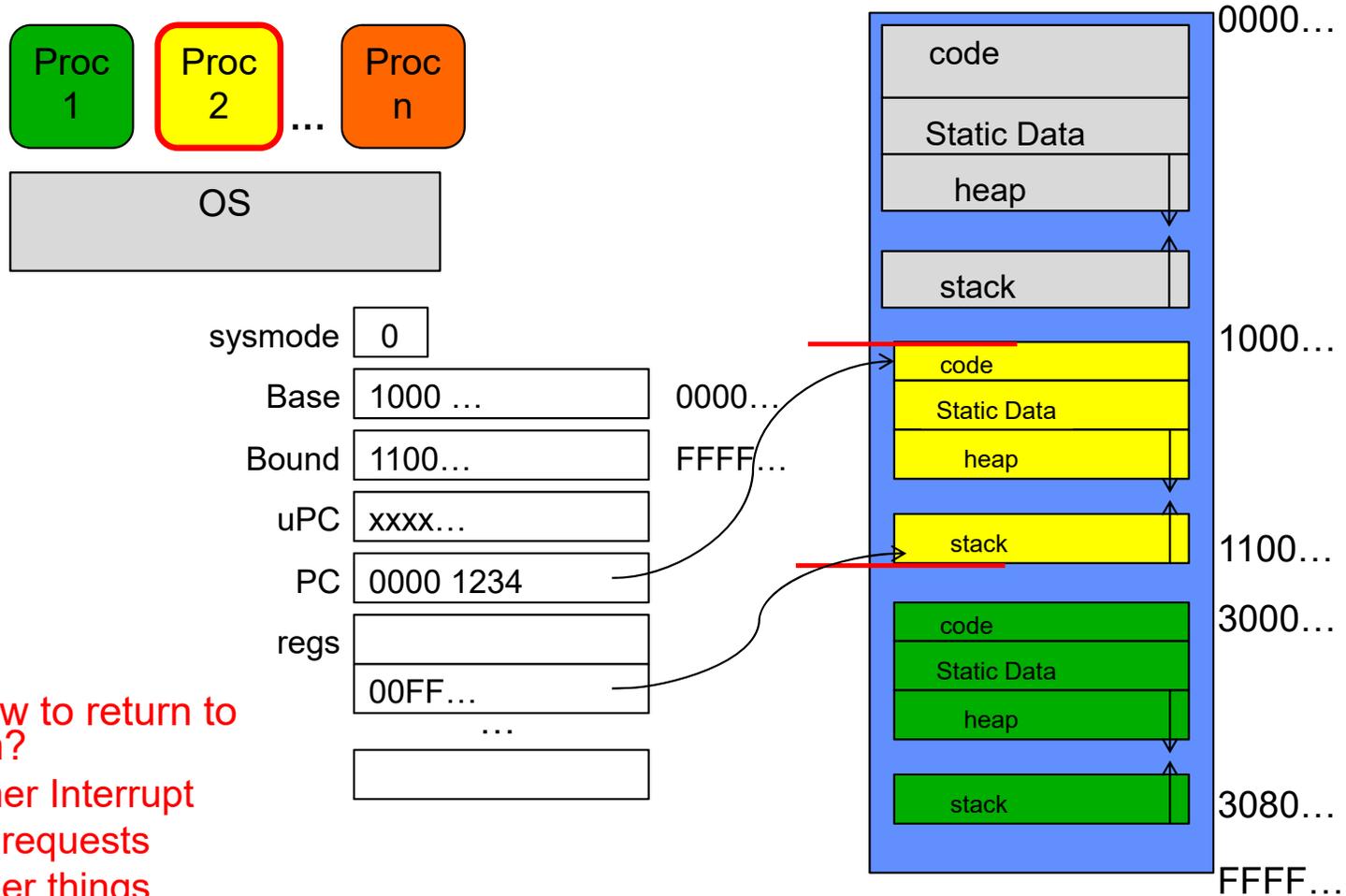
How do we get the system target address of the “unprogrammed control transfer?”

Interrupt Vector



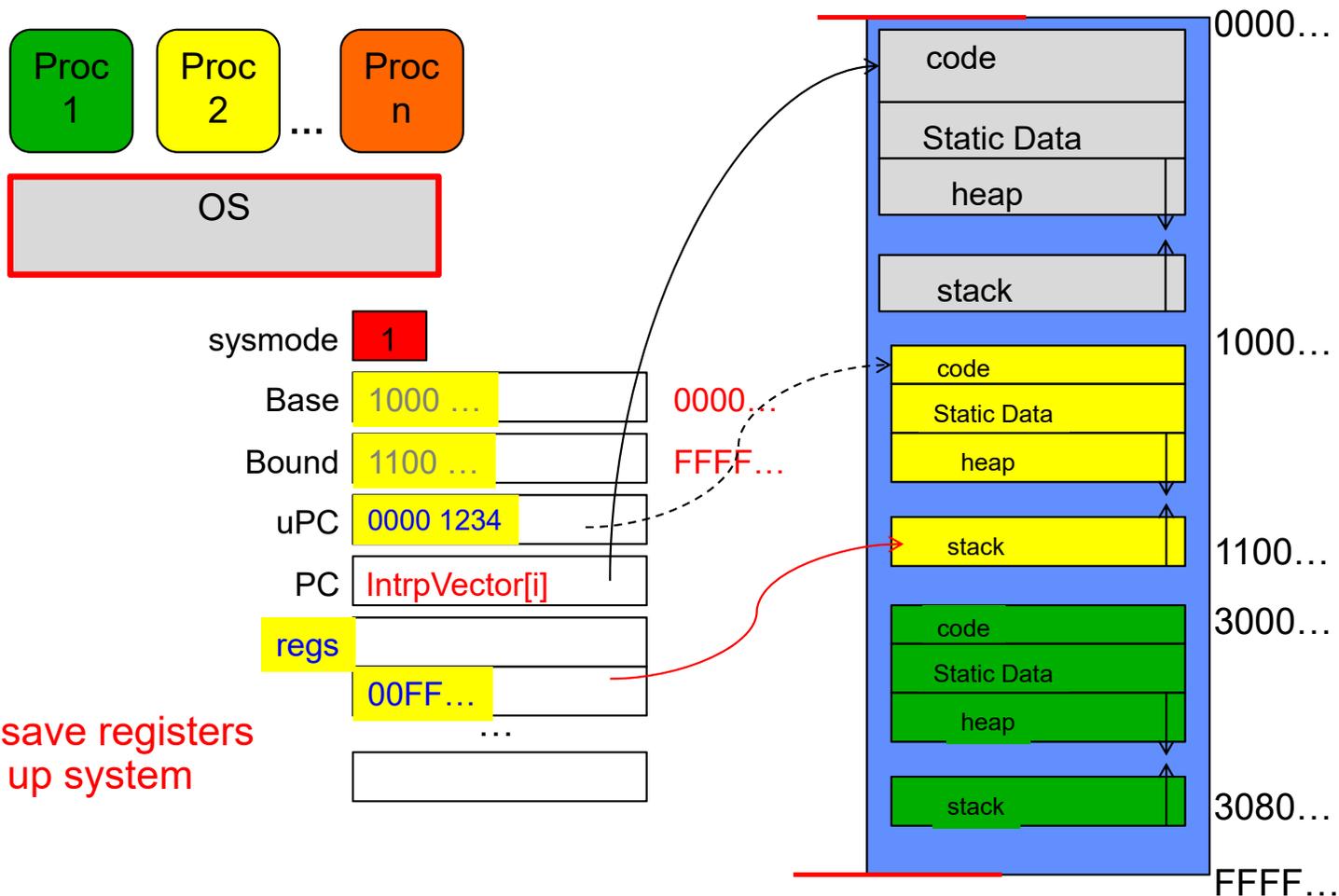
- Where else do you see this dispatch pattern?

Simple B&B: User => Kernel



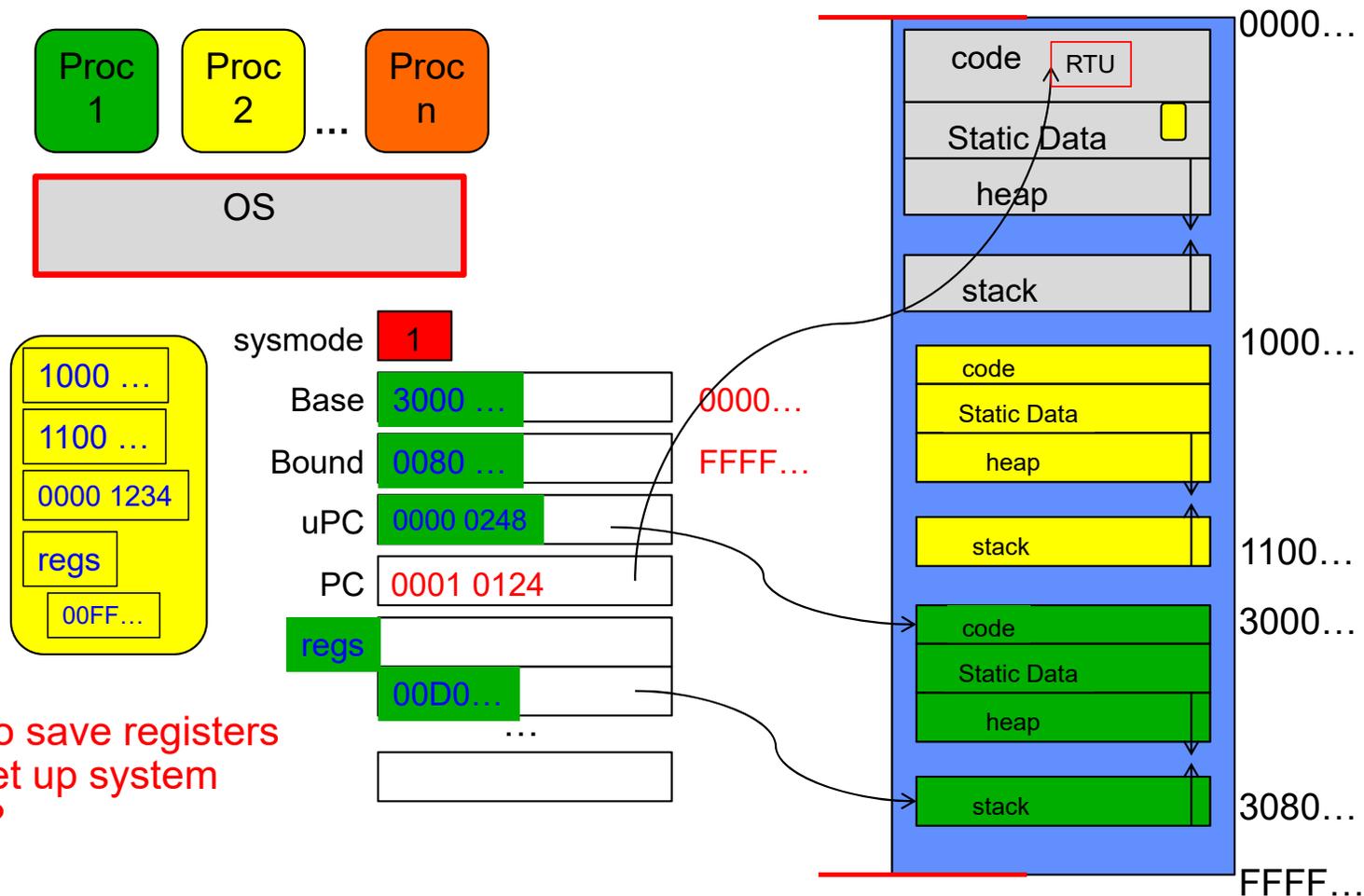
- So: How to return to system?
 - Timer Interrupt
 - I/O requests
 - Other things

Simple B&B: Interrupt



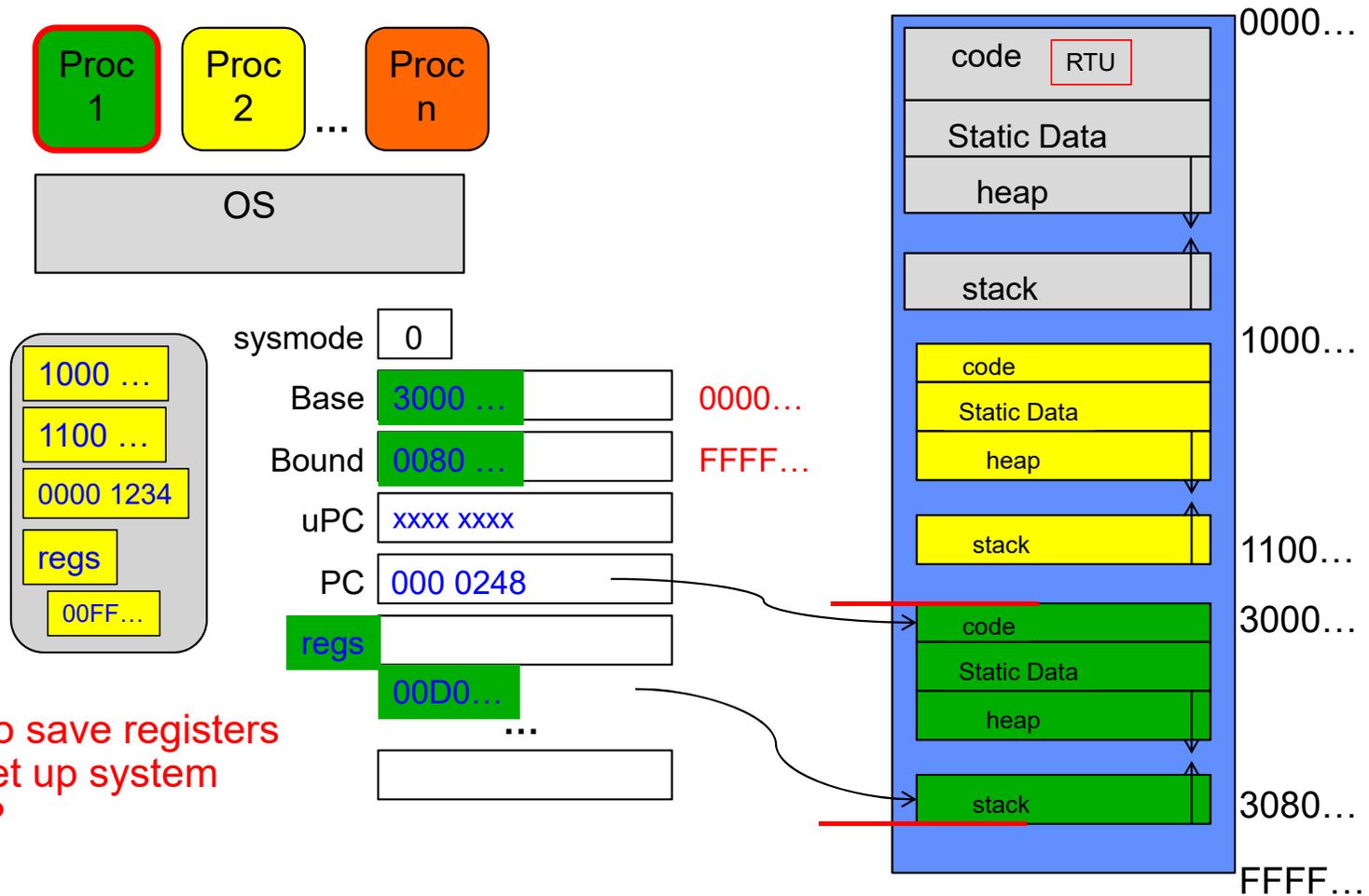
- How to save registers and set up system stack?

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?

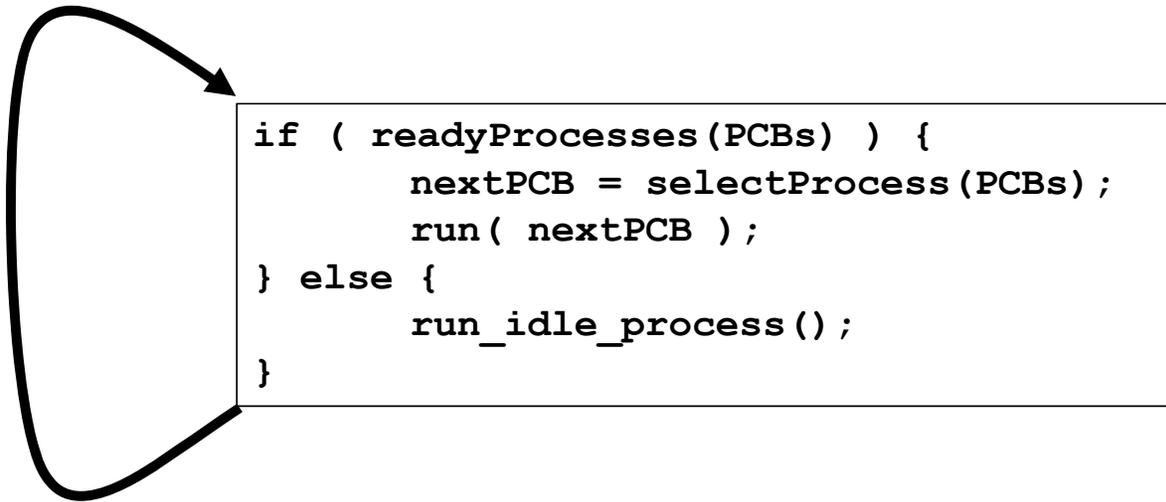
Running Many Programs ???

- 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 a 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



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