Four Fundamental Operating System Concepts

Sam Kumar

CS 162: Operating Systems and System Programming

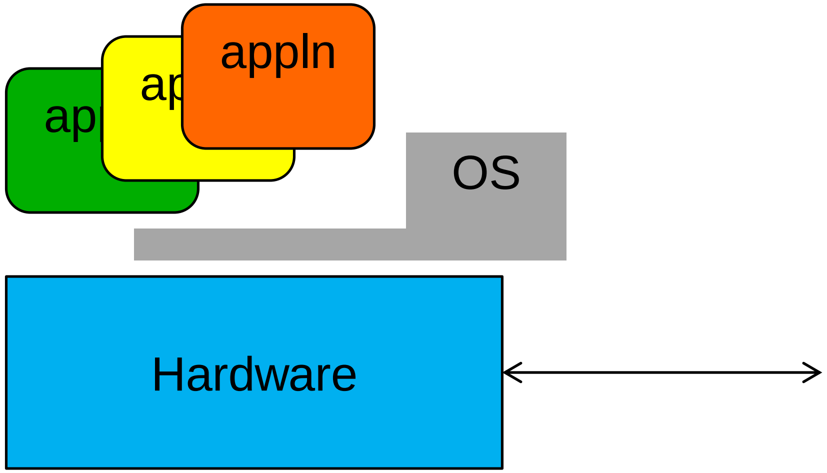
Lecture 2

https://inst.eecs.berkeley.edu/~cs162/su20

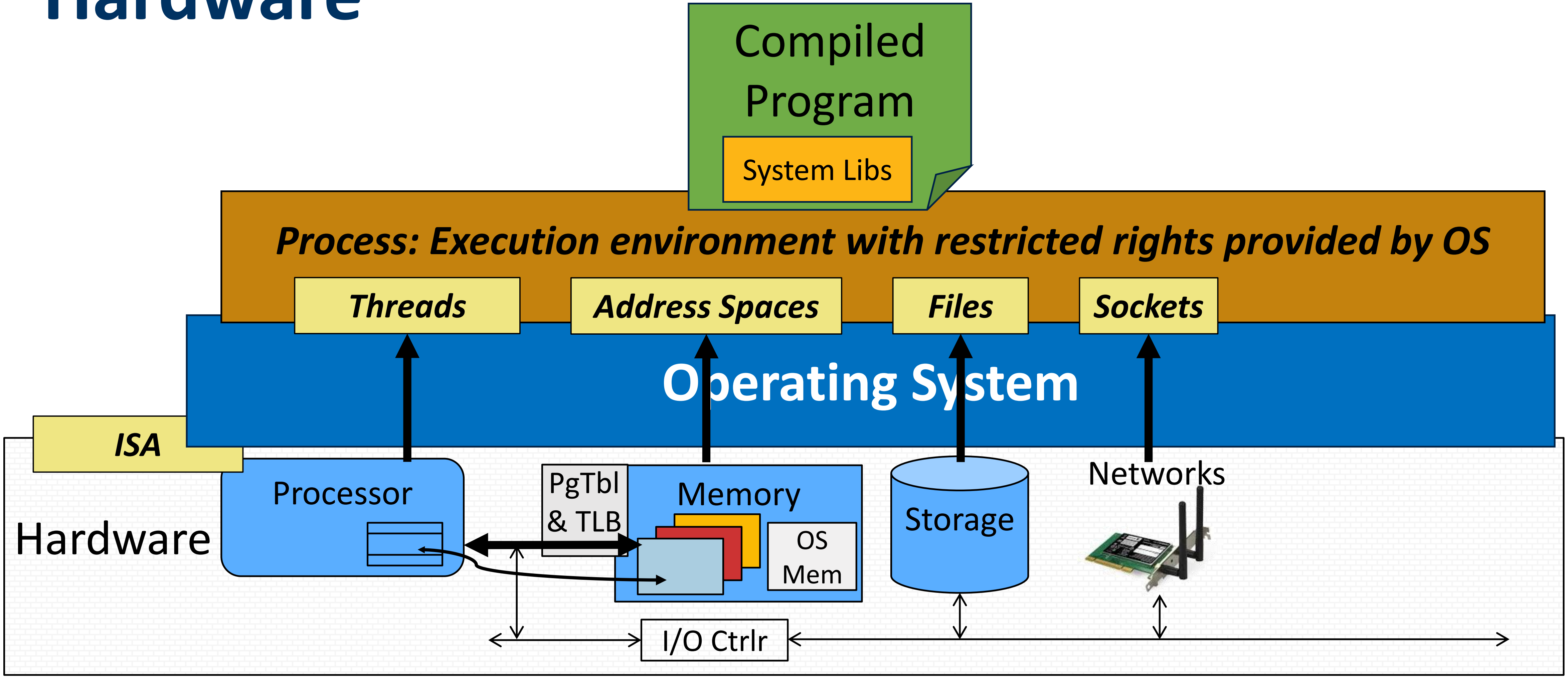
Read: A&D, 2.1-7

Recall: What is an Operating System?

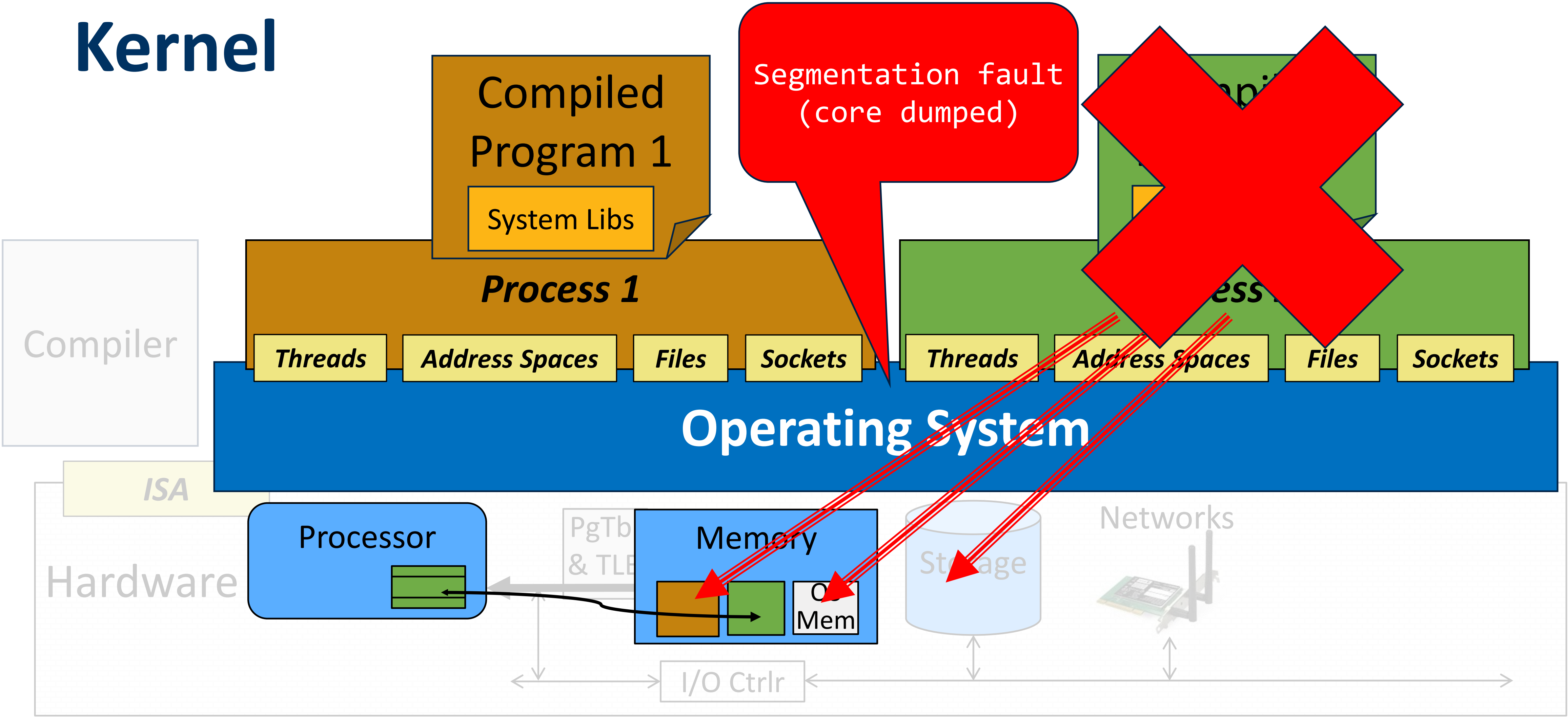
- Special layer of software that provides application software access to hardware resources
  - Convenient abstraction of complex hardware devices
  - Protected access to shared resources
  - Security and authentication
  - Communication



Recall: OS *Abstracts* the Underlying Hardware



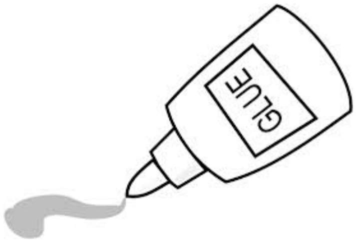
Recall: OS *Protects* Processes and the Kernel



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

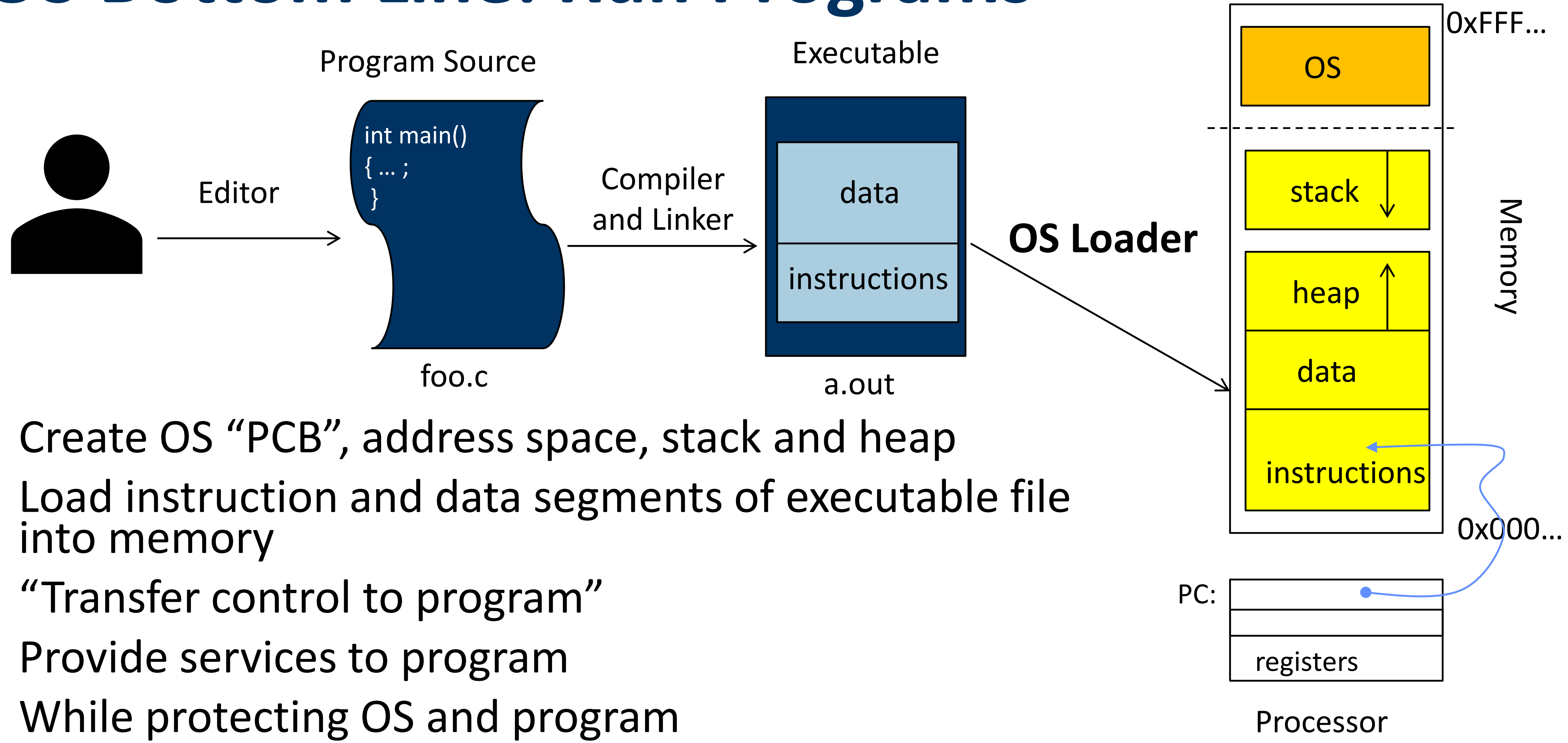


- Glue
  - Common services
    - Storage, Window system, Networking
    - Sharing, Authorization
    - Look and feel

Today: Four Fundamental OS Concepts

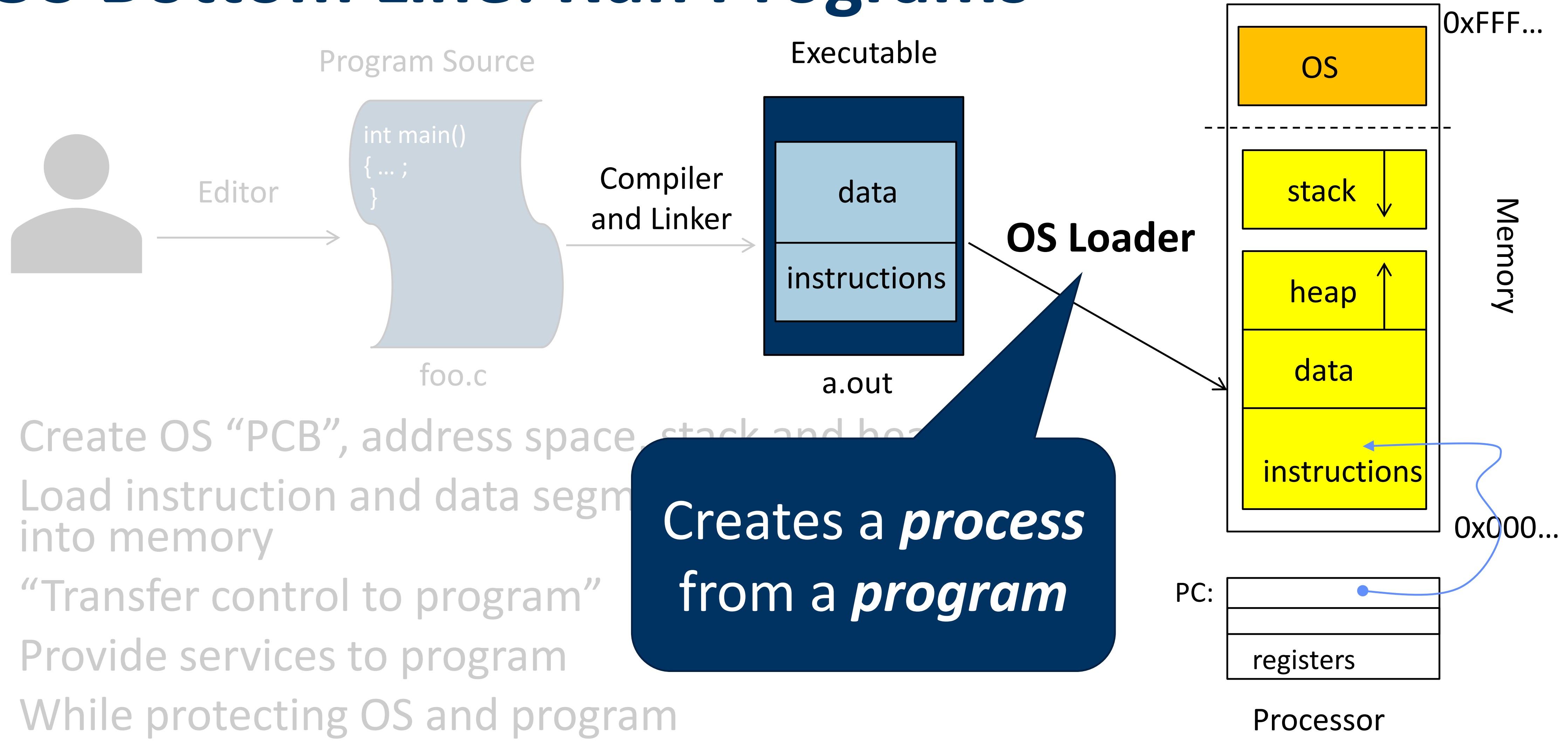
- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address Space** (with **Translation**)
  - Program's view of memory is distinct from physical machine
- **Process: Instance of a Running Program**
  - Address space + one or more threads + …
- **Dual-Mode Operation and Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

OS Bottom Line: Run Programs



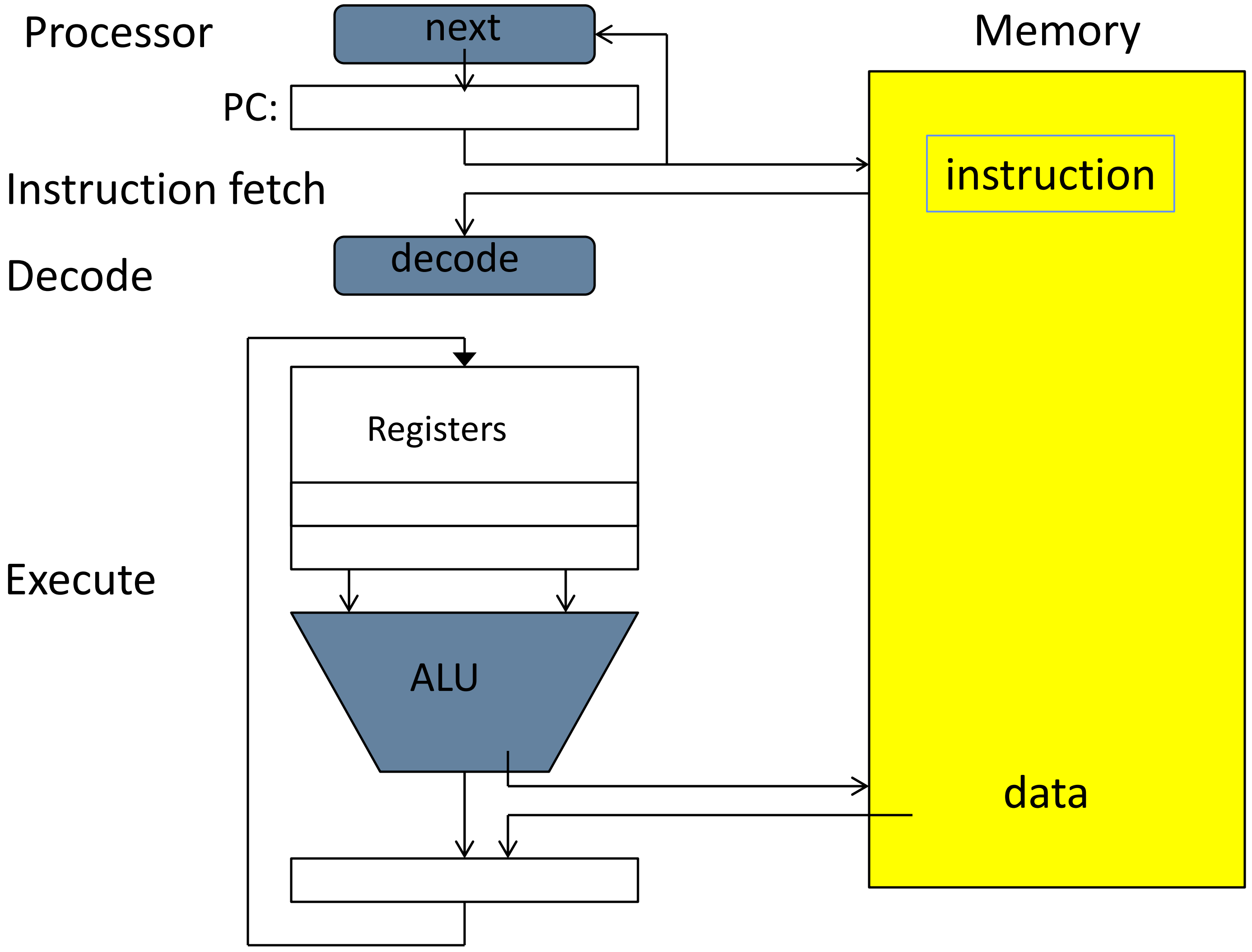
- Create OS "PCB", address space, stack and heap
- Load instruction and data segments of executable file into memory
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

OS Bottom Line: Run Programs

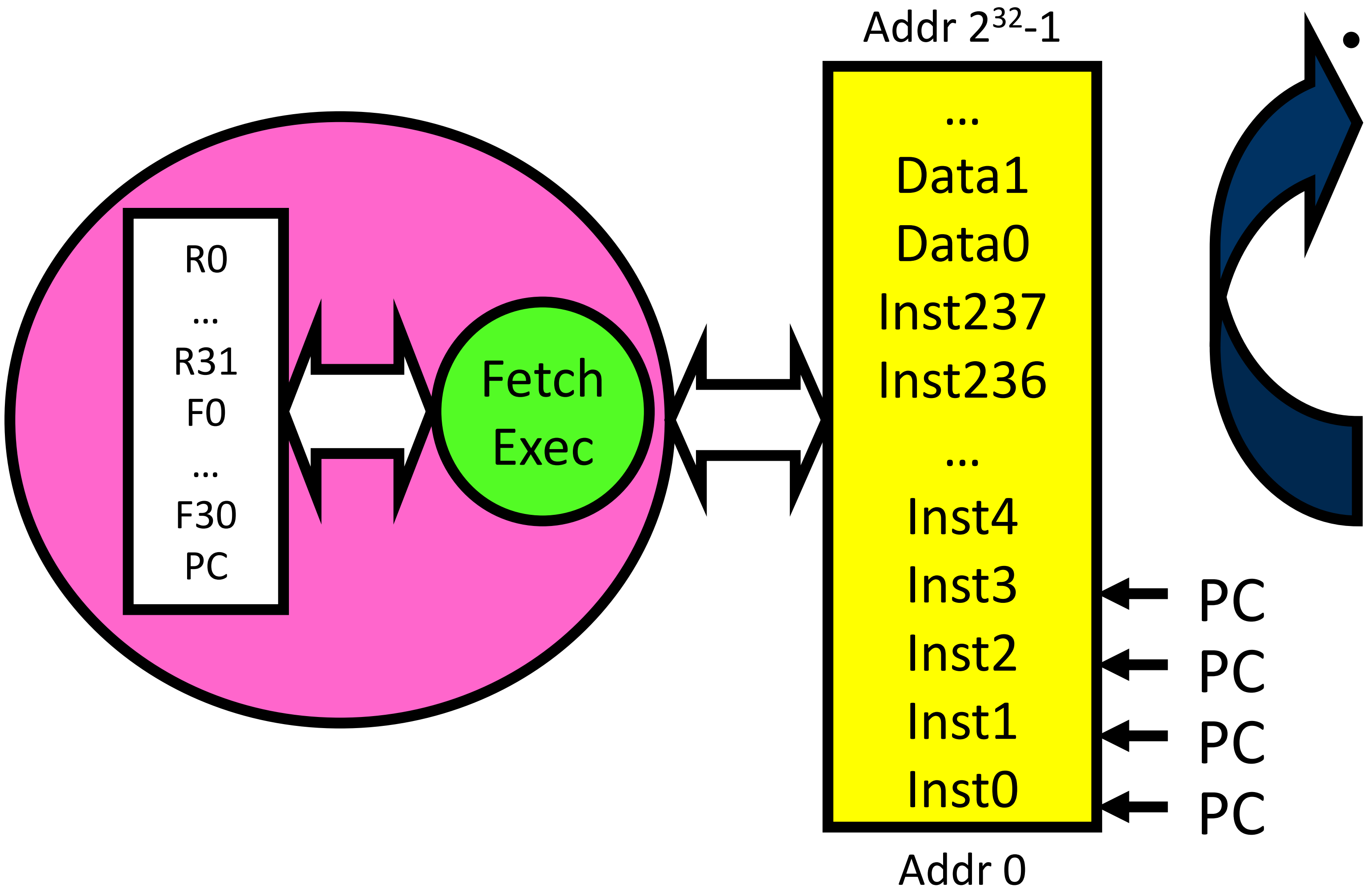


- Create OS “PCB”, address space, stack and hea
- Load instruction and data segm into memory
- “Transfer control to program”
- Provide services to program
- While protecting OS and program

Review (61C): How Programs Execute



Review (61C): How Programs Execute



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

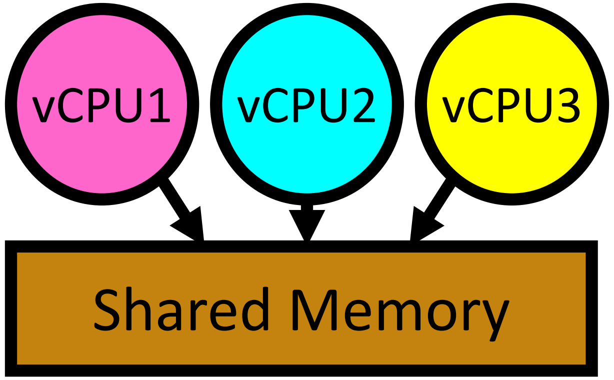
Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address Space** (with **Translation**)
  - Program’s view of memory is distinct from physical machine
- **Process: Instance of a Running Program**
  - Address space + one or more threads + …
- **Dual-Mode Operation and Protection**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other

Key OS Concept: Thread

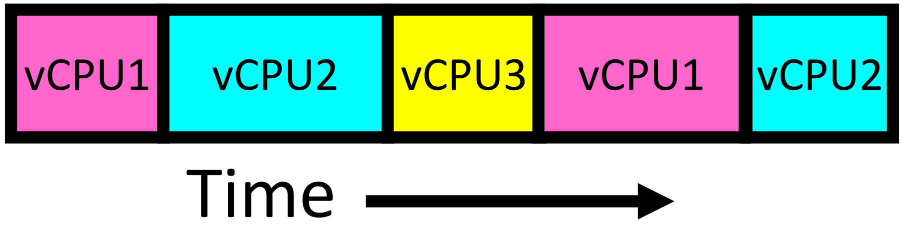
- Definition: **A single, unique execution context**
  - Program counter, registers, stack
- **A thread is the OS abstraction for a CPU core**
  - A “virtual CPU” of sorts
- Registers hold the root state of the thread:
  - Including program counter – pointer to the currently executing instruction
  - The rest is “in memory”
- Registers point to thread state in memory:
  - Stack pointer to the top of the thread’s (own) stack

Illusion of Multiple Processors



- Threads are **virtual cores**
- Multiple threads: **Multiplex** hardware in time
- **A thread is *executing* on a processor when it is resident in that processor's registers**

On a single physical CPU:



- Each virtual core (thread) has PC, SP, Registers
- Where is it?
  - On the real (physical) core, or
  - Saved in memory – called the Thread Control Block (TCB)

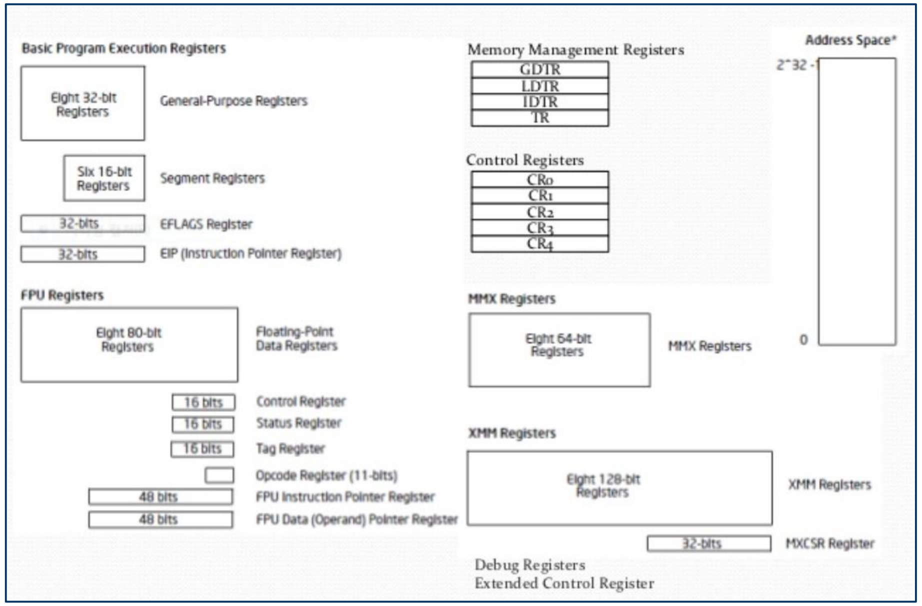
OS Object Representing a Thread

- Traditional term: Thread Control Block (TCB)
- Holds contents of registers when thread is not running…
- … And other information the kernel needs to keep track of the thread and its state.

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

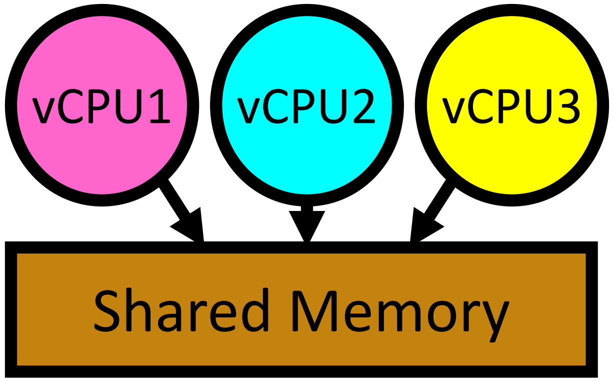
Load/Store Arch with software conventions



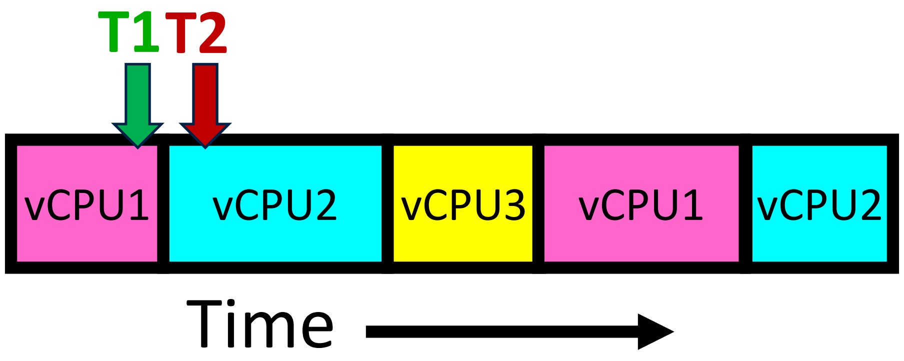
Complex mem-mem arch with specialized registers and “segments”

- In CS 61C you learned RISC-V
- In section tomorrow you’ll learn x86

Illusion of Multiple Processors



On a single physical CPU:



- At T1: vCPU1 on real core
- At T2: vCPU2 on real core

- What happened?
  - OS ran [how?]
  - Saved PC, SP, … in vCPU1's thread control block
  - Loaded PC, SP, … from vCPU2's thread control block

- This is called **context switch**

Very Simple Multiprogramming

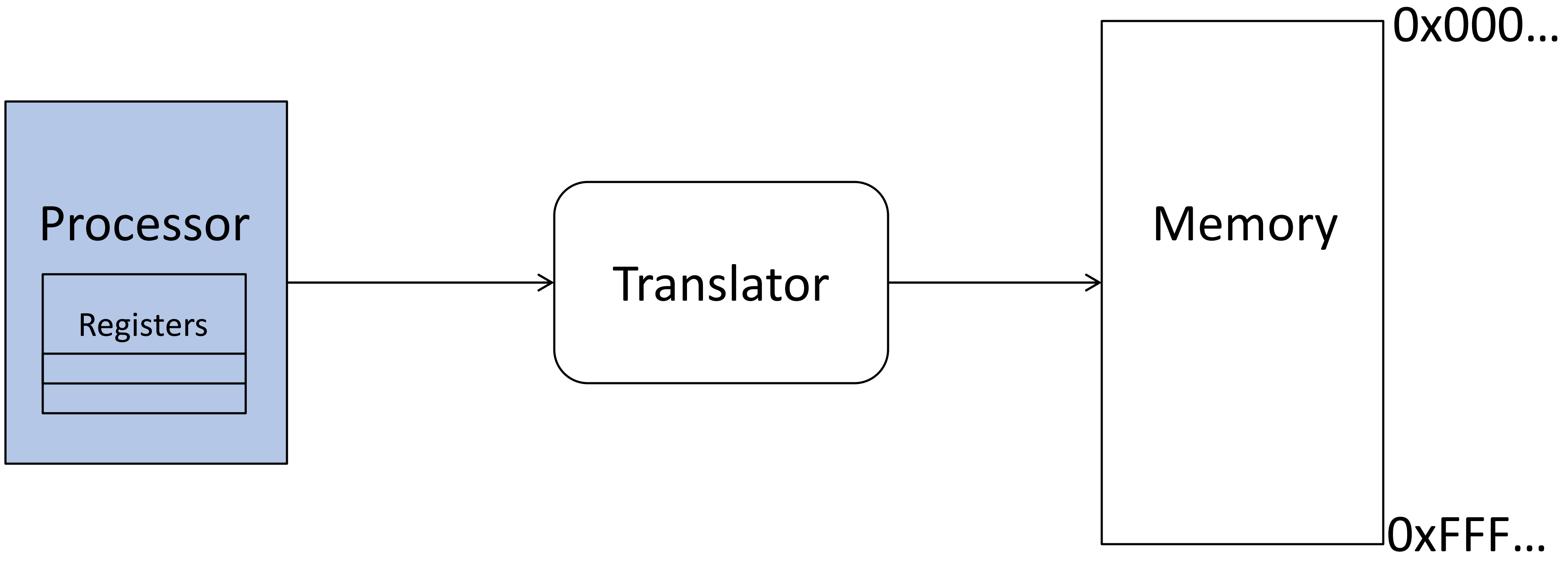
- All vCPUs share non-CPU resources
  - Memory, I/O Devices
- Each thread can **read/write memory**
  - Including data of others
  - And the OS!
- Unusable?
- This approach is used in:
  - Very early days of computing
  - Embedded applications
  - MacOS 1-9/Windows 3.1 (switch only with voluntary yield)
  - Windows 95-ME

Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address Space** (with **Translation**)
  - Program's view of memory is distinct from physical machine
- **Process: Instance of a Running Program**
  - Address space + one or more threads + …
- **Dual-Mode Operation and Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

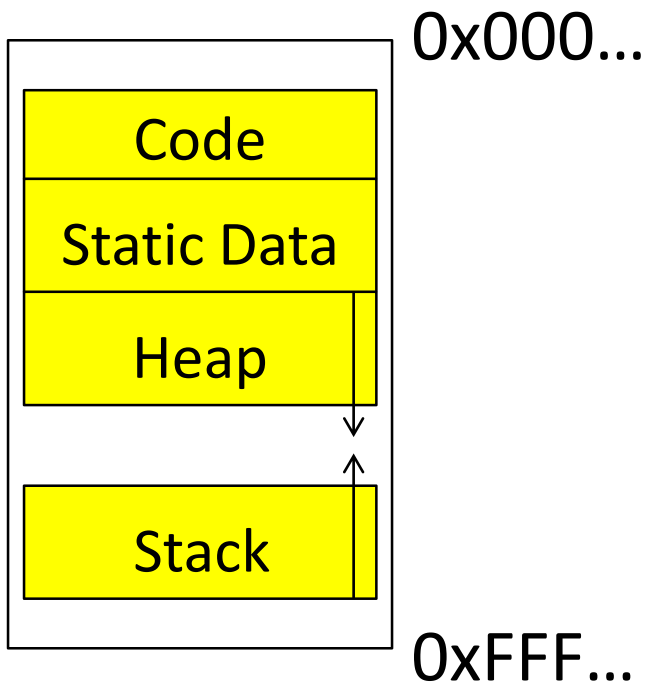
Key OS Concept: Address Space

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

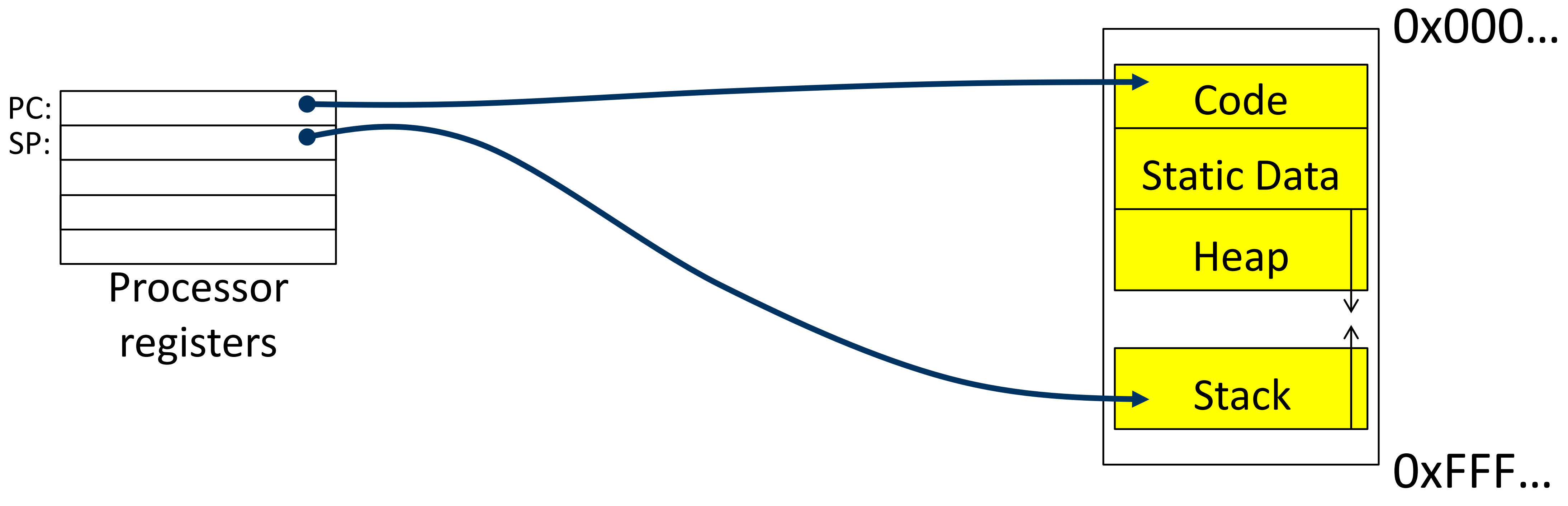


Address Space

- Definition: **Set of accessible addresses and the state associated with them**
  - 2^{32} = ~4 billion on a 32-bit machine
- What happens when you read or write to an address?
  - Perhaps acts like regular memory
  - Perhaps causes I/O operation
    - (Memory-mapped I/O)
  - Causes program to abort (segfault)?
  - Communicate with another program
  - …



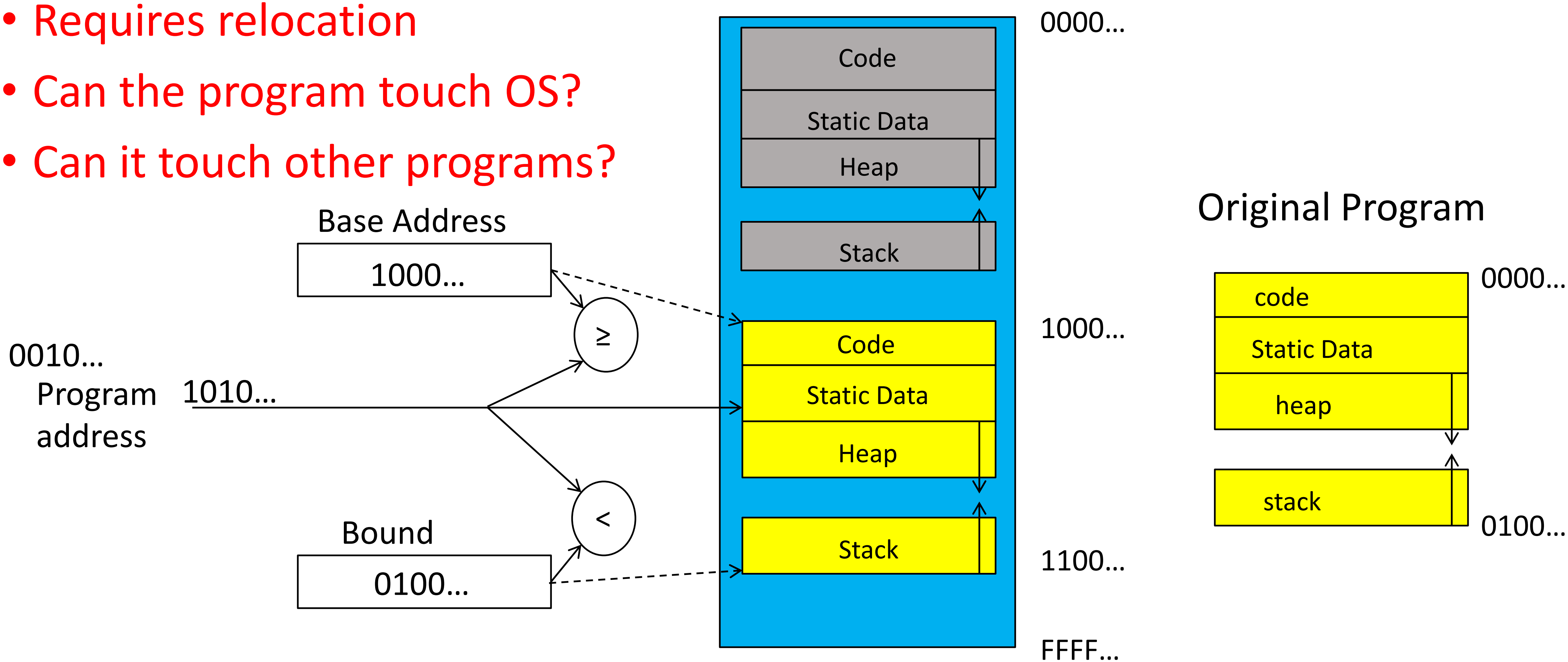
Typical Address Space Structure



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

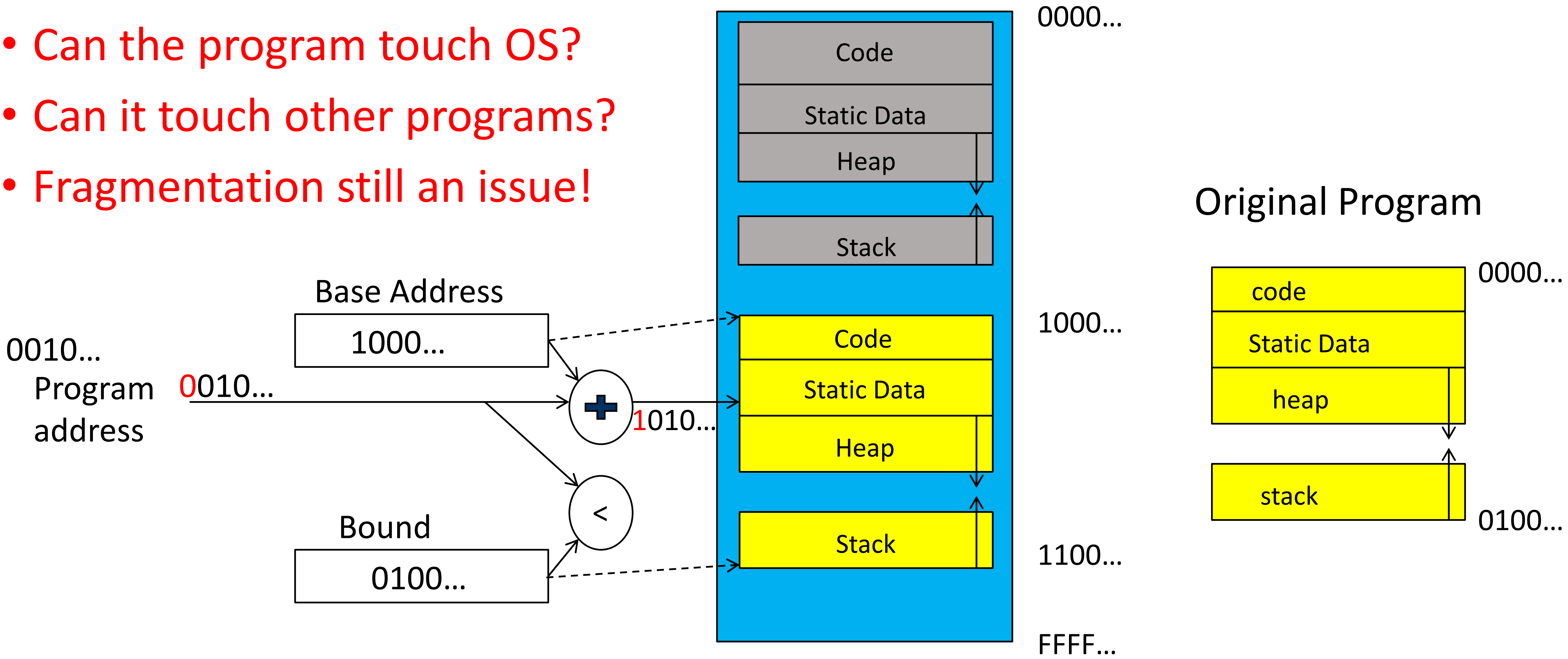
Base and Bound (no Translation)

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



Base and Bound (with Translation)

- Can the program touch OS?
- Can it touch other programs?
- Fragmentation still an issue!

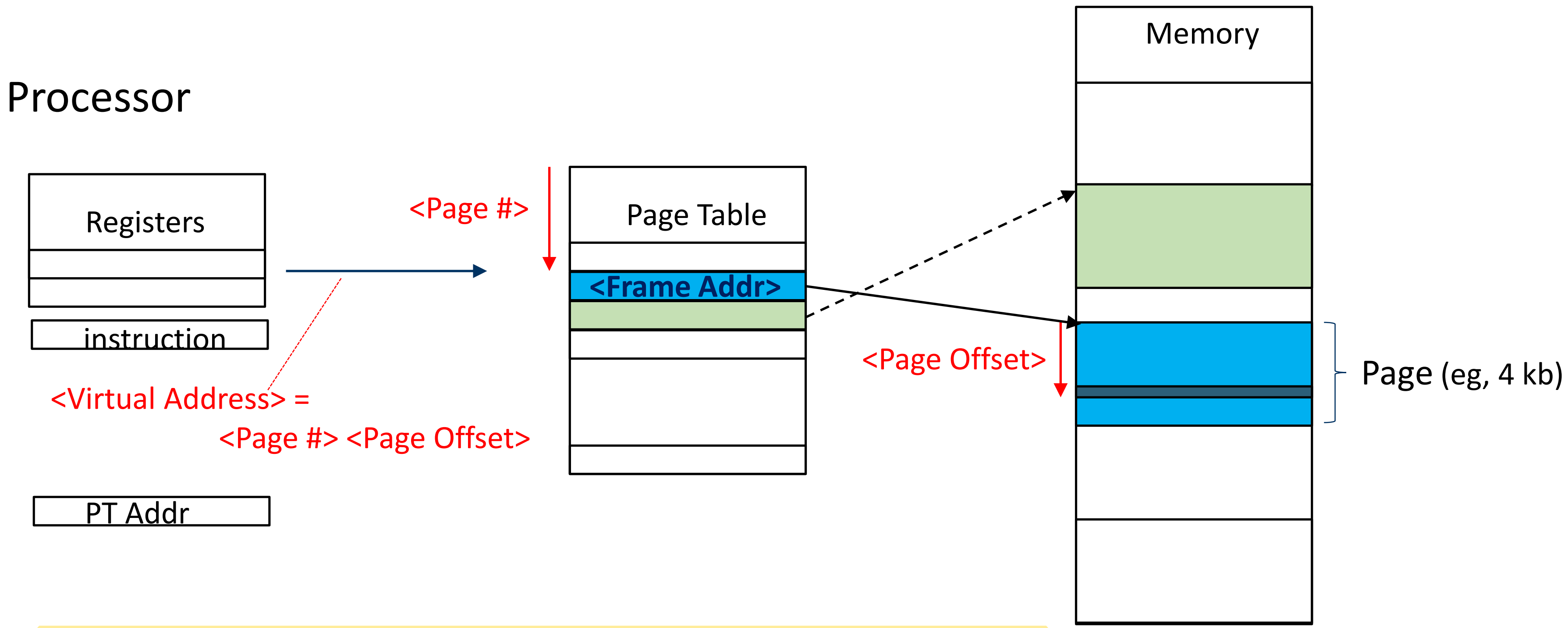


Paged Virtual Address Space

- What if we break the entire virtual address space into equal-size chunks (i.e., pages) and have a base and bound 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

Paged Virtual Address Space



- Instructions operate on virtual addresses
- Translated at runtime to physical addresses via a page table
- Special register holds page table base address of current process' page table

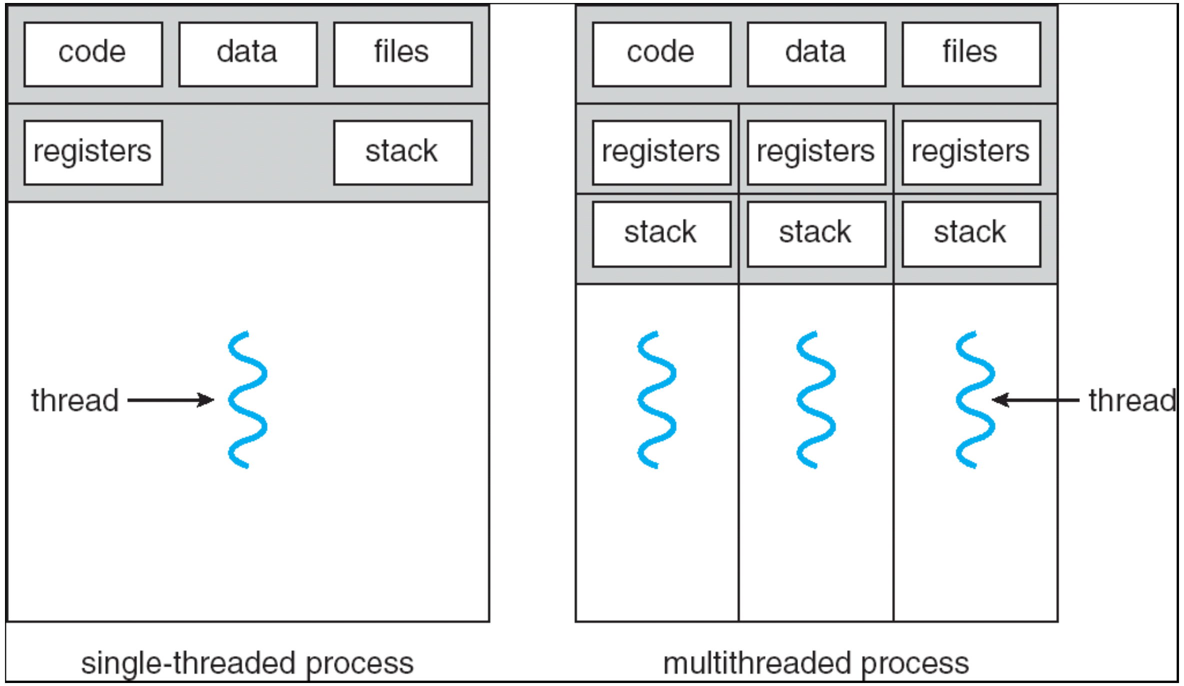
Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address Space** (with **Translation**)
  - Program's view of memory is distinct from physical machine
- **Process: Instance of a Running Program**
  - Address space + one or more threads + …
- **Dual-Mode Operation and Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

Key OS Concept: Process

- Definition: execution environment with restricted rights
  - One or more threads executing in a single address space
  - Owns file descriptors, network connections
- Instance of a running program
  - When you run an executable, it runs in its own process
  - Application: one or more processes working together
- Protected from each other; OS protected from them
- **In modern OSes, anything that runs outside of the kernel runs in a process**

Single and Multithreaded Processes



- Threads encapsulate concurrency
  - “Active” component
- Address space encapsulate protection:
  - “Passive” component
  - Keeps bugs from crashing the entire system
- Why have multiple threads per address space?

Protection and Isolation

- Why?
  - 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**

Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address Space** (with **Translation**)
  - Program's view of memory is distinct from physical machine
- **Process: Instance of a Running Program**
  - Address space + one or more threads + …
- **Dual-Mode Operation and Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

Dual-Mode Operation

- **One bit** of state: processor is either in (user mode or kernel mode)
- Certain actions are only permitted in kernel mode
  - e.g., changing the page table pointer
  - Certain entries in the page table
  - Hardware I/O instructions

Announcements

- Homework 0 is out!
  - Due Thursday
  - Register for the autograder and get the class VM running ASAP
- Quiz 0 is tomorrow
  - Optional, ungraded
  - Opportunity to get familiar with online exam format
- If you have a conflict with any of the exams, then fill out the Exam Conflict Form linked on Piazza

Dual-Mode Operation

- Processes (i.e., programs you run) execute in **user mode**
  - To perform privileged actions, processes request services from the OS kernel
  - Carefully controlled transition from user to kernel mode
- Kernel executes in **kernel mode**
  - Performs privileged actions to support running processes
  - … and configures hardware to properly protect them (e.g., address translation)

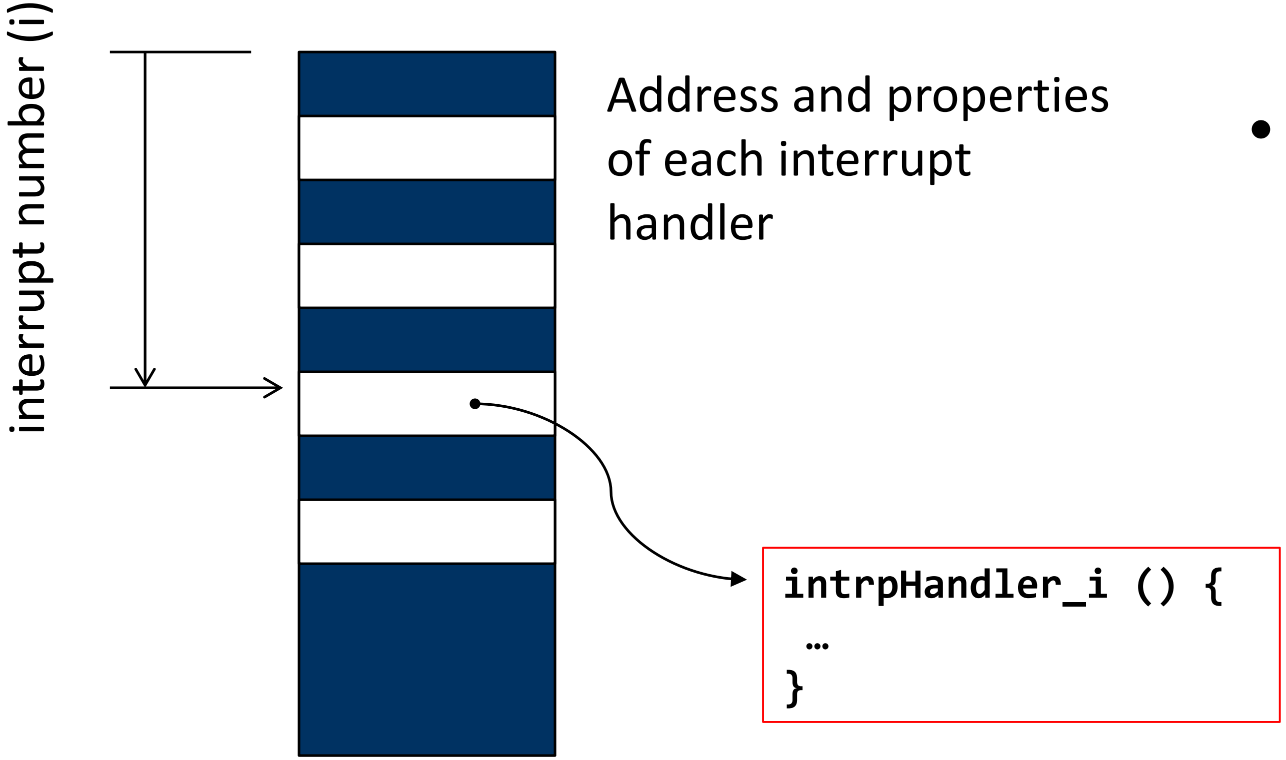
Three Types of User → Kernel Mode Transfer

- System Call ("syscalls")
  - Process requests a system service (e.g., open a file)
  - Like a function call, but "outside" the process
- Interrupt
  - External asynchronous event, independent of the process
  - e.g., Timer, I/O device
- Trap
  - Internal synchronous event in process triggers context switch
  - E.g., Divide by zero, bad memory access (segmentation fault)

All 3 *exceptions* are UNPROGRAMMED CONTROL TRANSFER

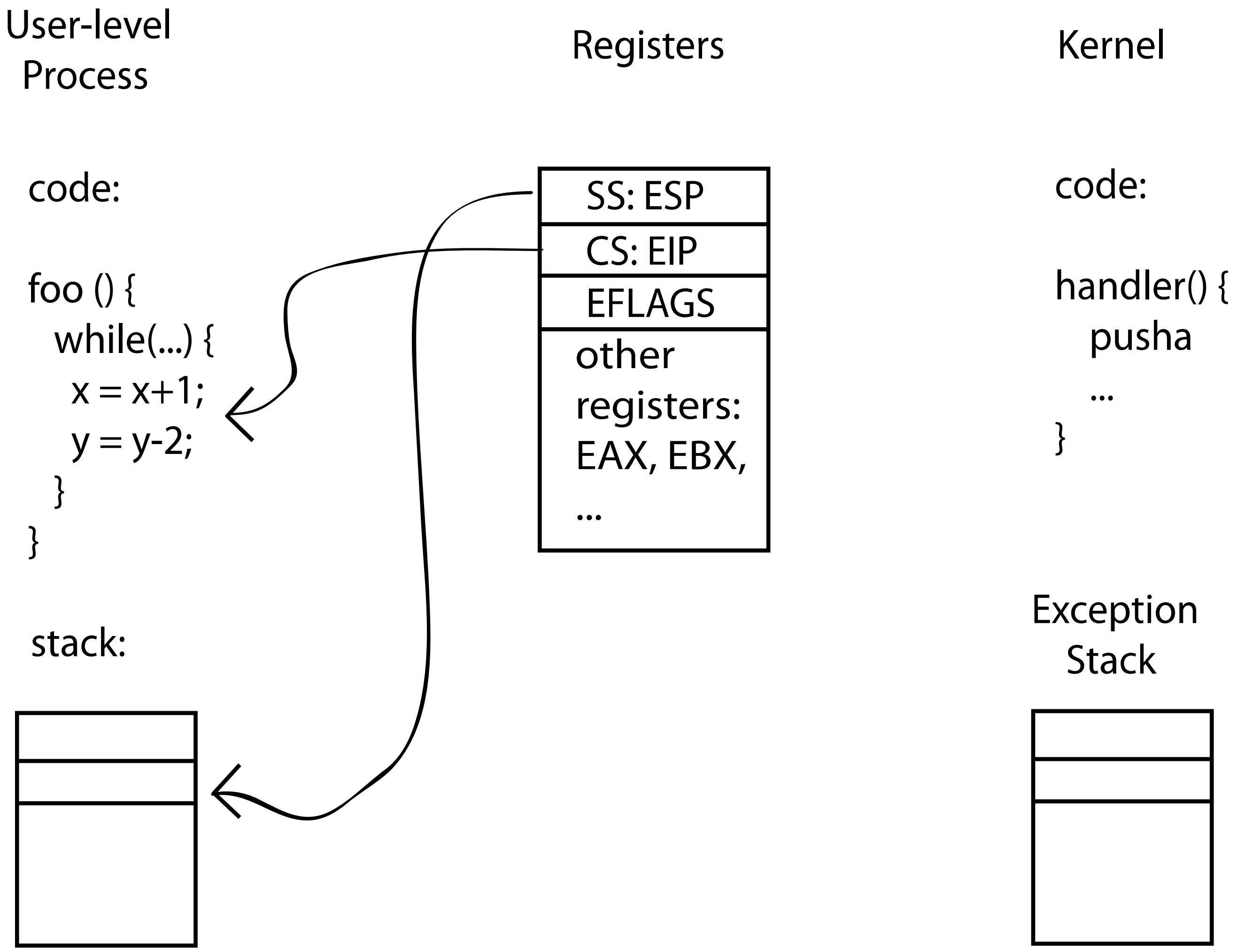
- User process can't jump to arbitrary instruction address in kernel!
- Why not?

Where do User → Kernel Mode Transfers Go?



- Cannot let user programs specify the exact address!
- Solution: ***Interrupt Vector***
  - OS kernel specifies a set of functions that are *entrypoints* to kernel mode
  - Appropriate function is chosen depending on the type of transition
    - Interrupt Number (i)
    - OS may do additional *dispatch*

Example: Before Exception



Example: After Exception

User-level Process

code:

```
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

stack:

Registers

SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...

Kernel

code:

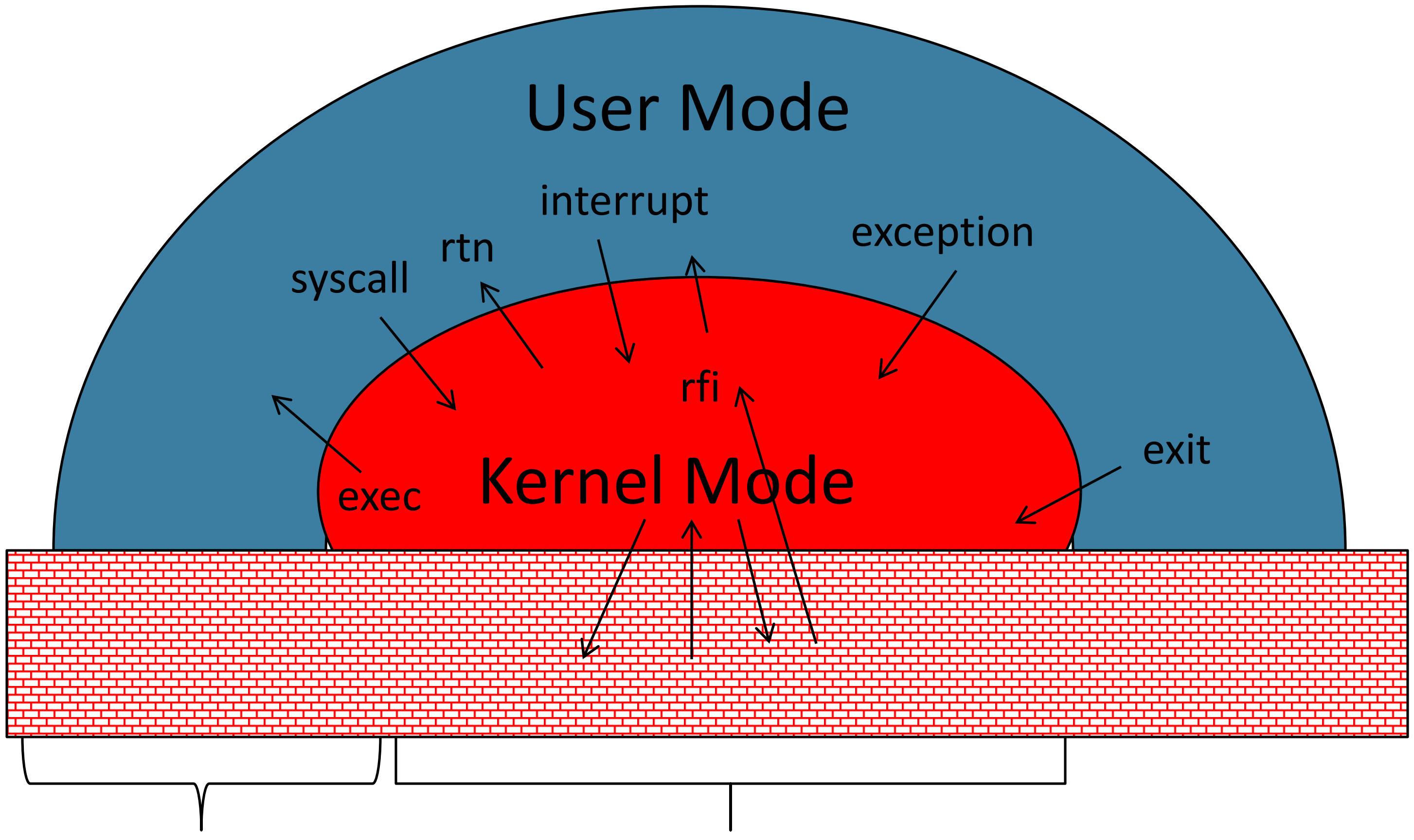
```
handler() {
  pusha
  ...
}
```

Exception Stack

SS
ESP
EFLAGS
CS
EIP
error

Why don't we just use the user stack?

Life of a Process



Implementing Safe User → Kernel Mode Transfers

- *Carefully* constructed kernel code packs up the user process state and sets it aside
- Must handle weird/buggy/malicious user state
  - Syscalls with null pointers
  - Return instruction out of bounds
  - User stack pointer out of bounds
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself
- User program should not know that an interrupt has occurred *(transparency)*

Kernel System Call Handler

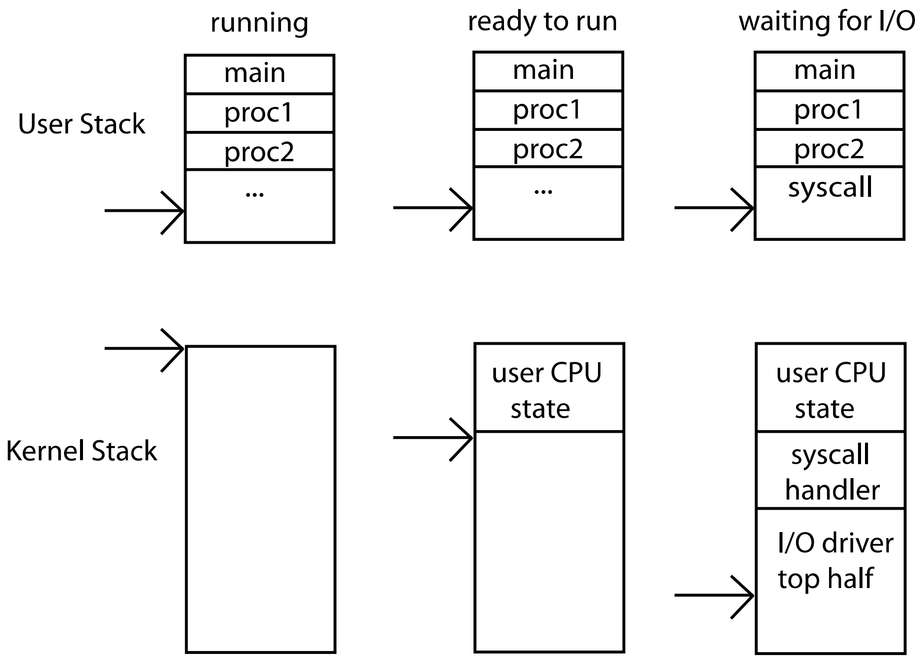
- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments
  - From user memory into kernel memory – carefully checking locations!
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory – carefully checking locations!

Kernel Stacks

- Interrupt handlers want a stack
- System call handlers want a stack
- Can't just use the user stack [why?]

Kernel Stacks

- One Solution: two-stack model
  - Each thread has user stack *and* a kernel stack
  - Kernel stack stores users registers during an exception
  - Kernel stack used to execute exception handler in the kernel



Hardware Support: Interrupt Control

- Interrupt processing not visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state
  - Happens *transparently* to the process—user program does not know it was interrupted
- Interrupt Handler invoked with interrupts ‘disabled’
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack up in a queue and pass off to an OS thread for hard work
    - wake up an existing OS thread

Hardware Support: Interrupt Control

- Interrupt processing not visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?

- Interrupt Handler invoked with interrupts 'disabled'
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack up in a queue and pass off to an OS thread for hard work
    - wake up an existing OS thread

How do we take Interrupts Safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Kernel interrupt stack
  - Handler works regardless of state of user code
- Interrupt masking
  - Handler is non-blocking
- Atomic transfer of control
  - “Single instruction”-like to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

Kernel → User Mode Transfers

- “Return from interrupt” instruction
- Drops mode from kernel to user privilege
- Restores user PC and stack

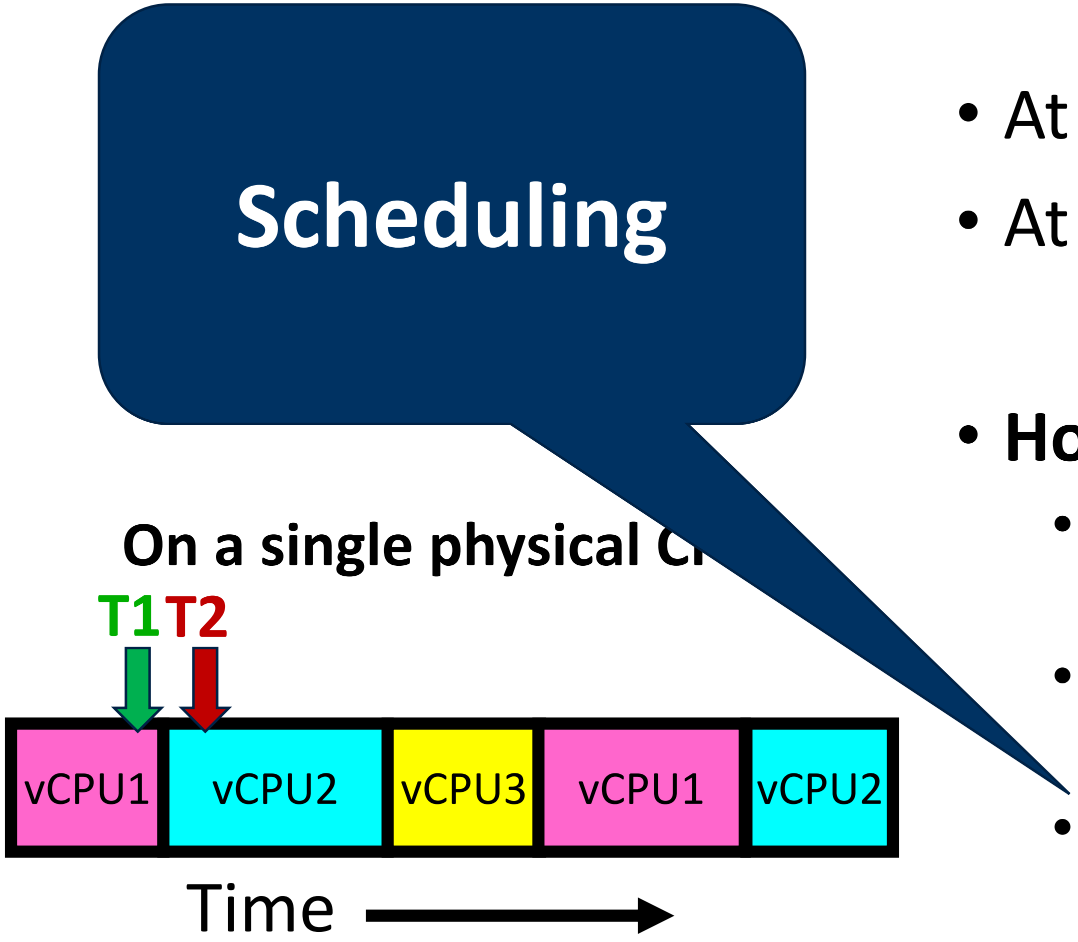
Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address Space** (with **Translation**)
  - Program’s view of memory is distinct from physical machine
- **Process: Instance of a Running Program**
  - Address space + one or more threads + …
- **Dual-Mode Operation and Protection**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other

Break (If Time)

Now, let's put it all together!

Illusion of Multiple Processors



- At T1: vCPU1 on real core
- At T2: vCPU2 on real core

- **How did the OS get to run?**
  - Earlier, OS configured a hardware timer to periodically generate an interrupt
  - On the interrupt, the hardware switches to kernel mode and the OS's timer interrupt handler runs
  - Timer interrupt handler decides whether to switch threads or not **according to a policy**

Scheduling

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

- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or …

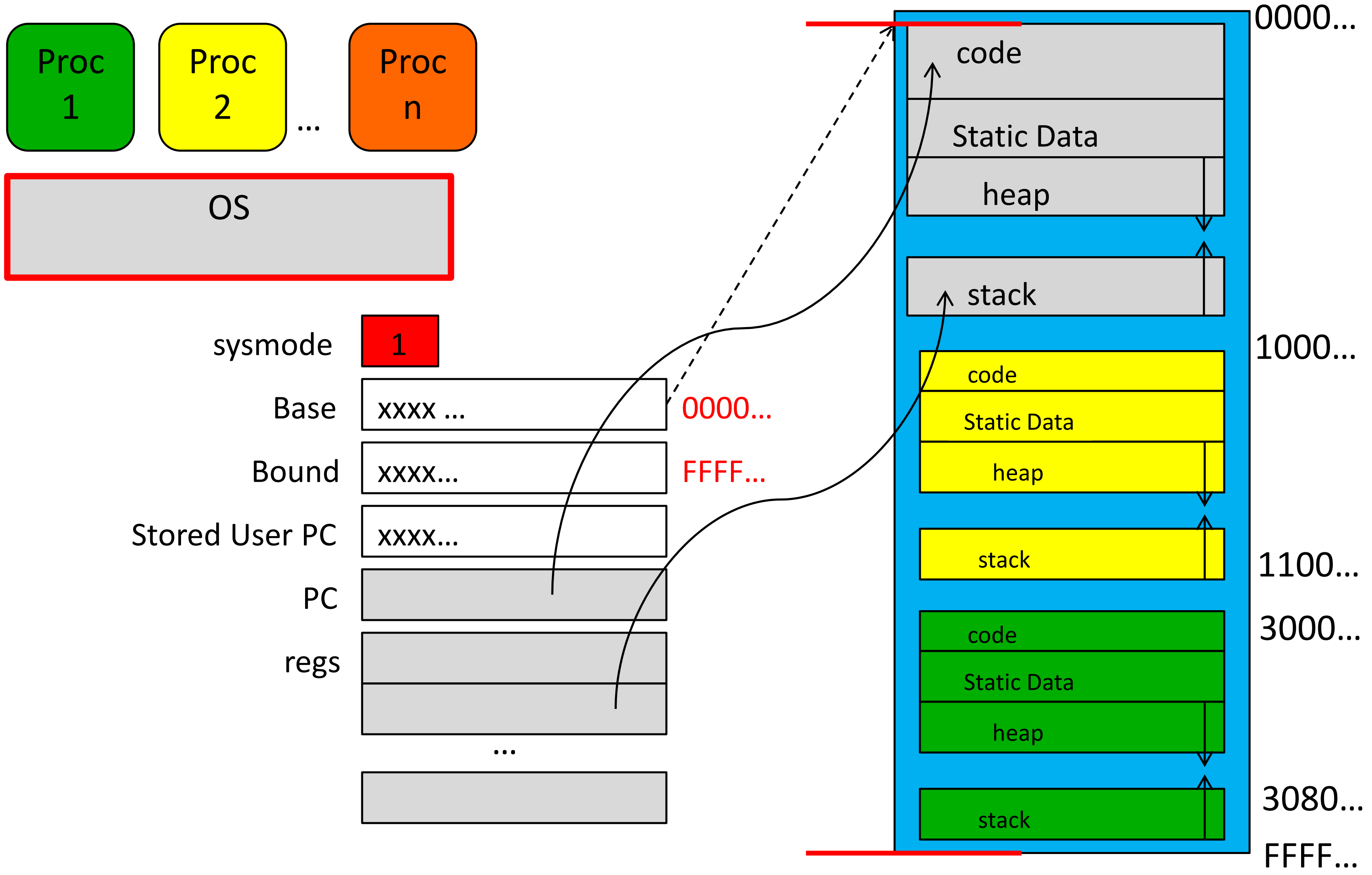
What's in a Process?

- Process Control Block (PCB): Kernel representation of each process
  - Process ID
  - Thread control block(s)
    - Program pointer, stack pointer, and registers for each thread
  - Page table (information for address space translation)
  - Necessary state to process system calls
    - Which files are open and which network connections are accessible to the process

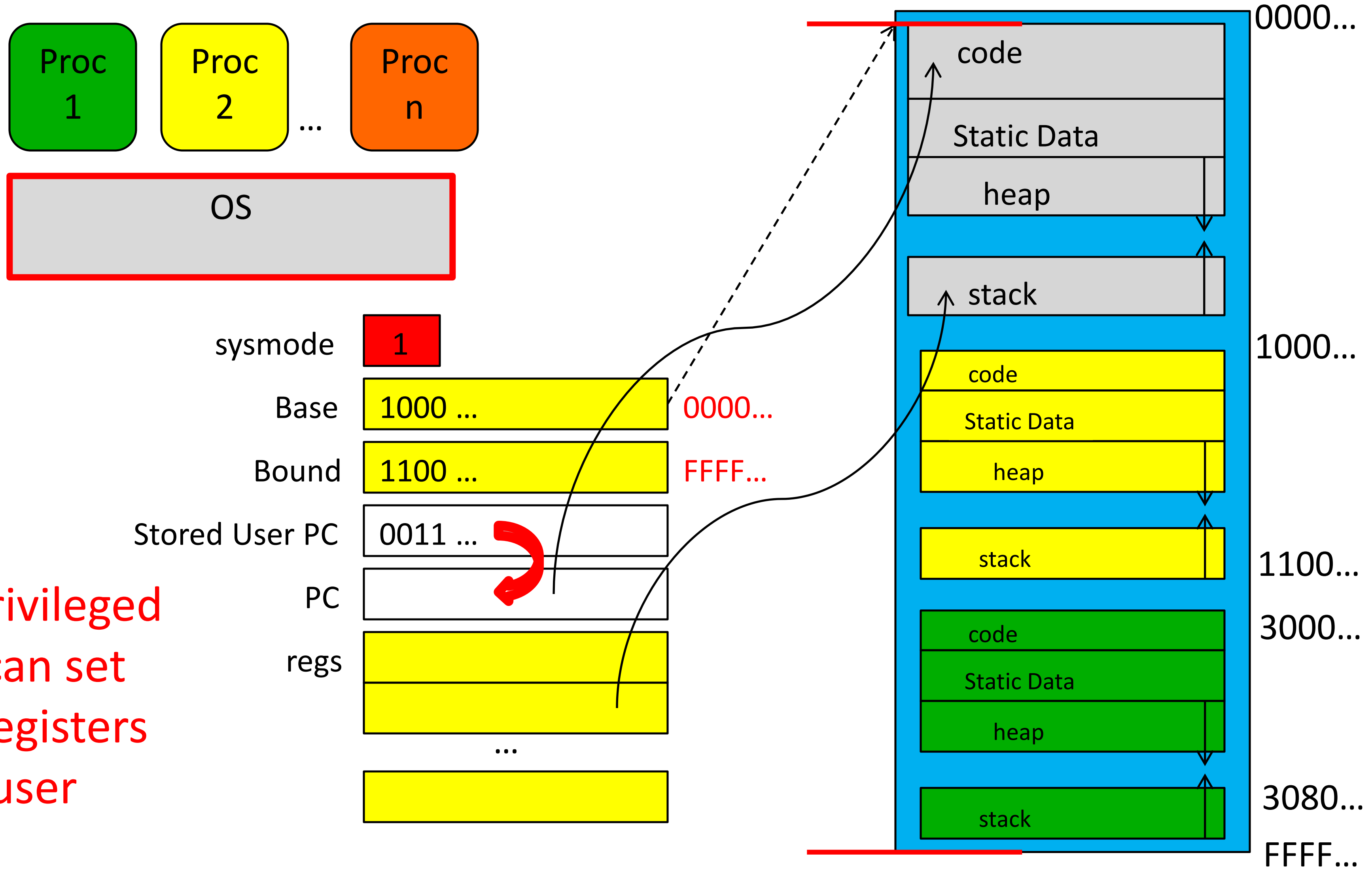
Mode Transfer and Translation

- Mode transfer should change address translation mapping
- Examples:
  - Ignore base and bound in kernel mode
  - Page tables:
    - Either switch to kernel page table…
    - Or mark some pages as only accessible in kernel mode

Base and Bound: OS Loads Process

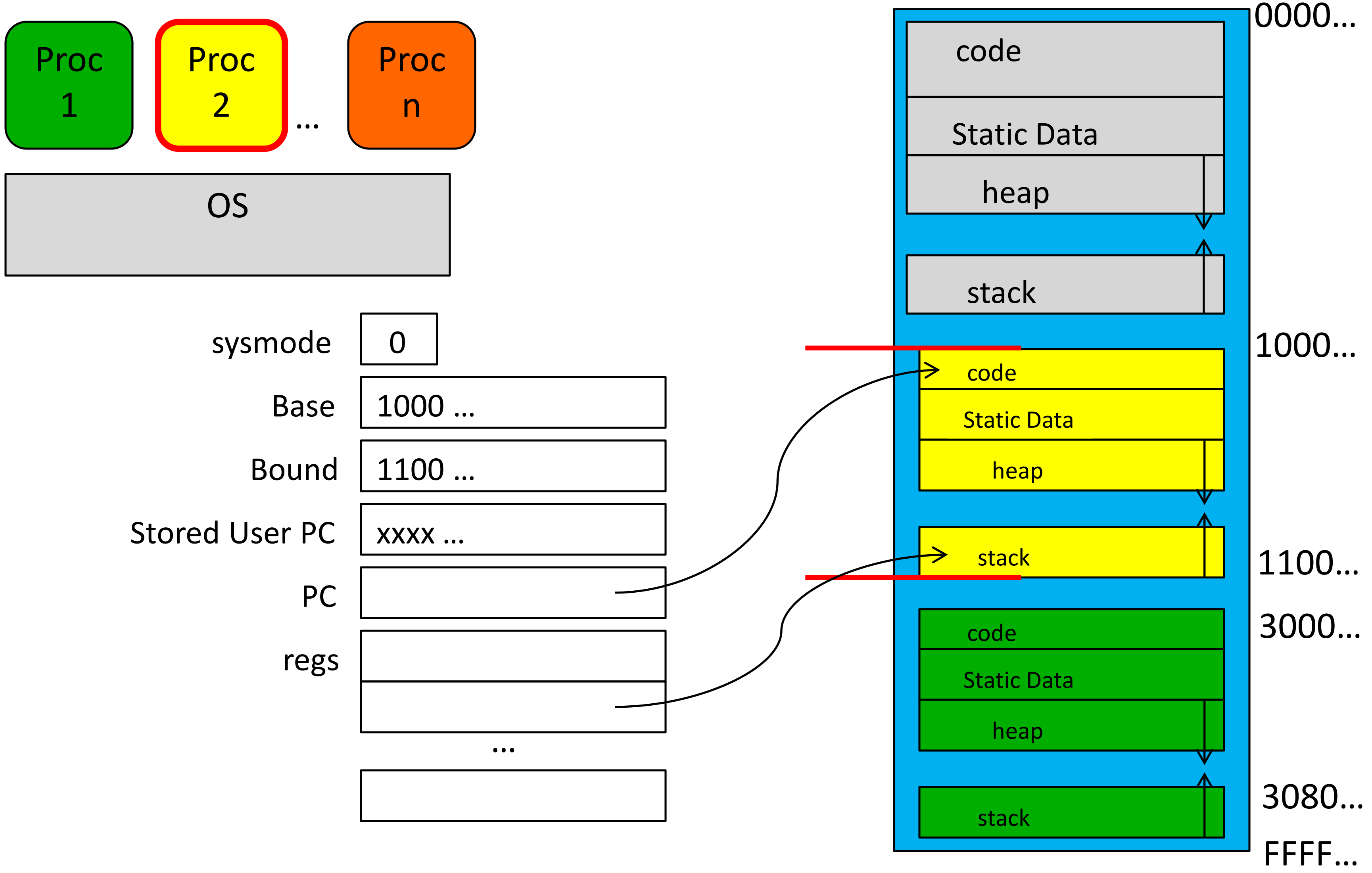


Base and Bound: About to Switch

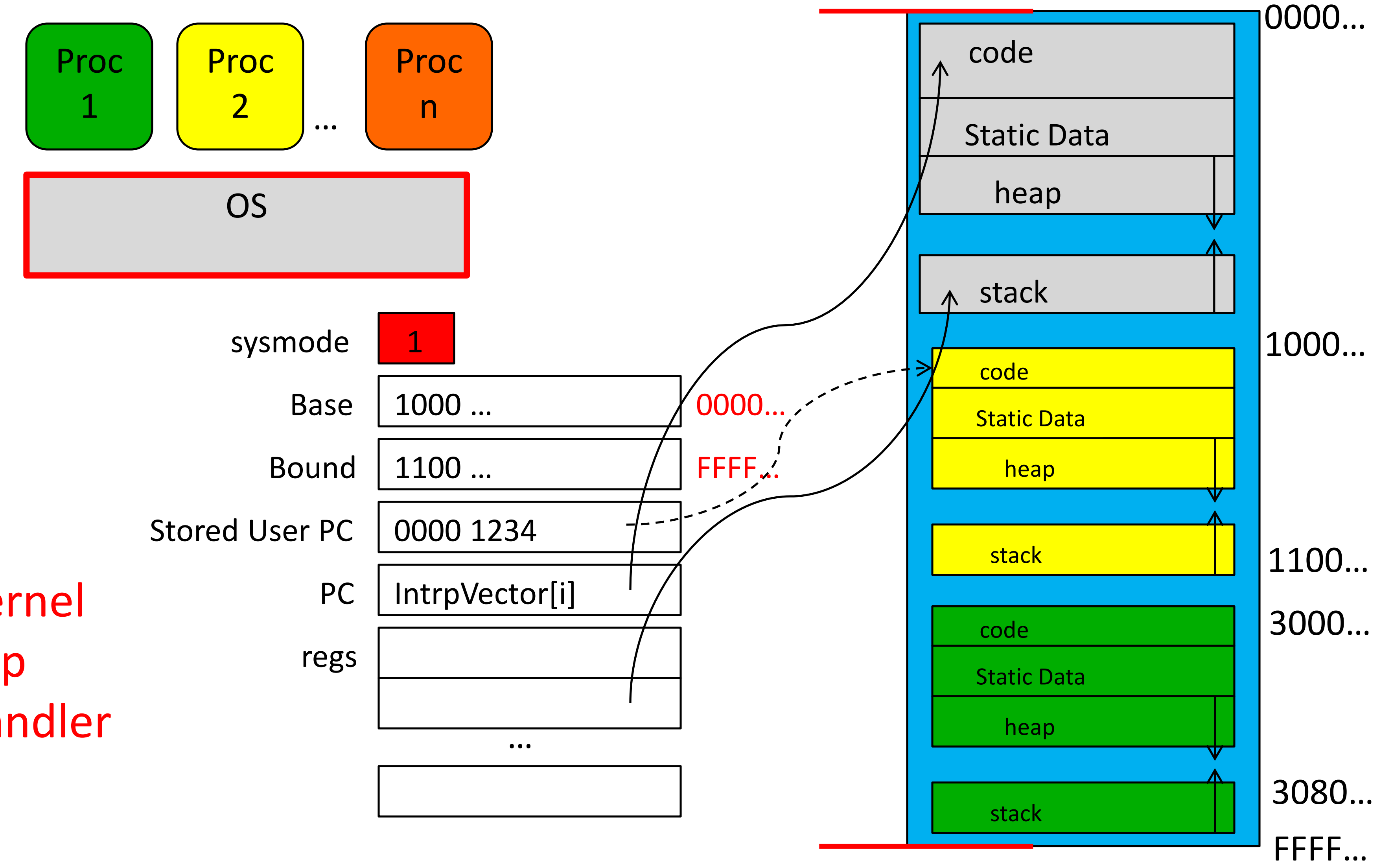


- OS runs in privileged mode, so it can set the special registers
- "Return" to user

Base and Bound: User Code Running

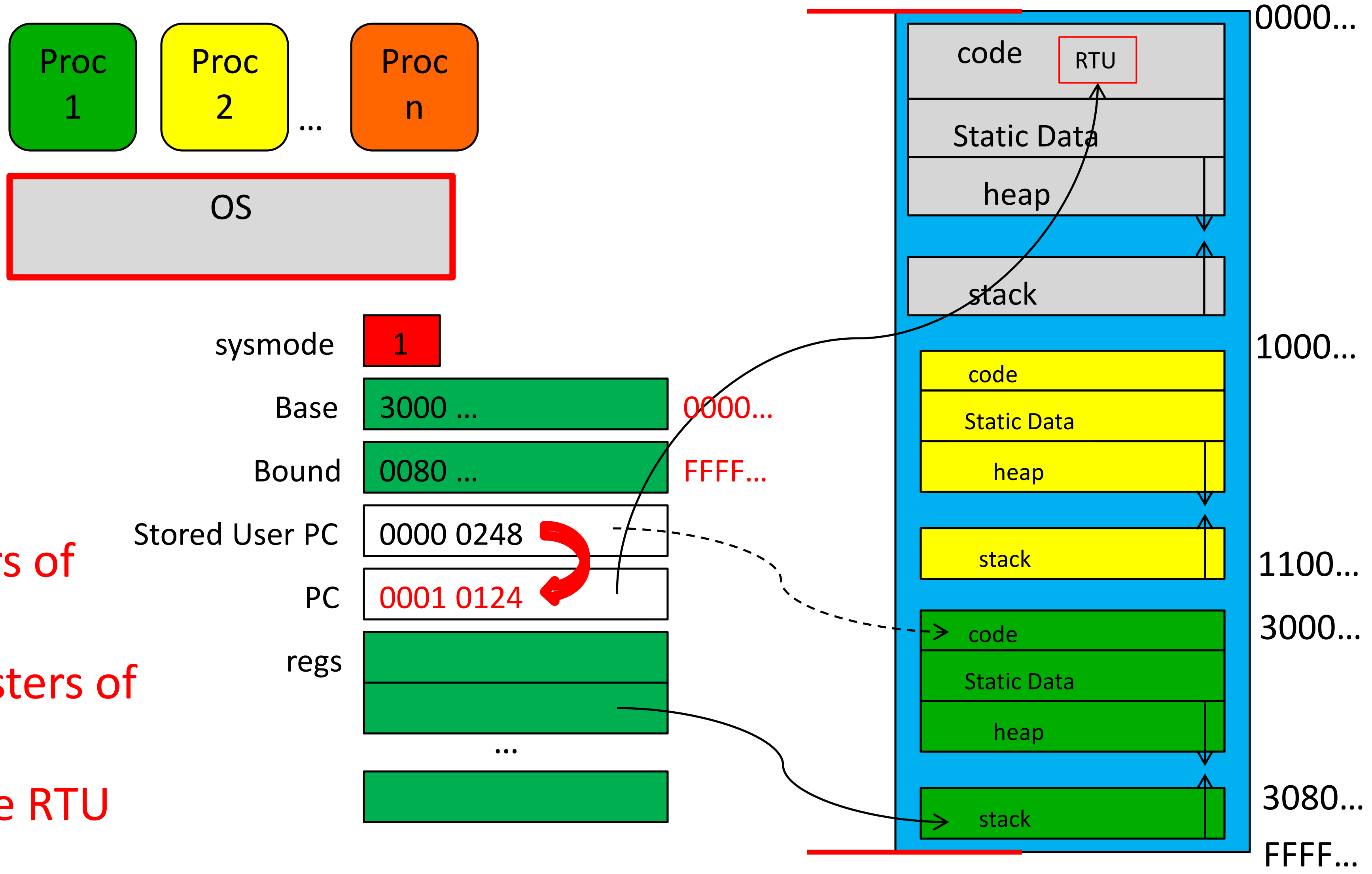


Base and Bound: Handle Interrupt



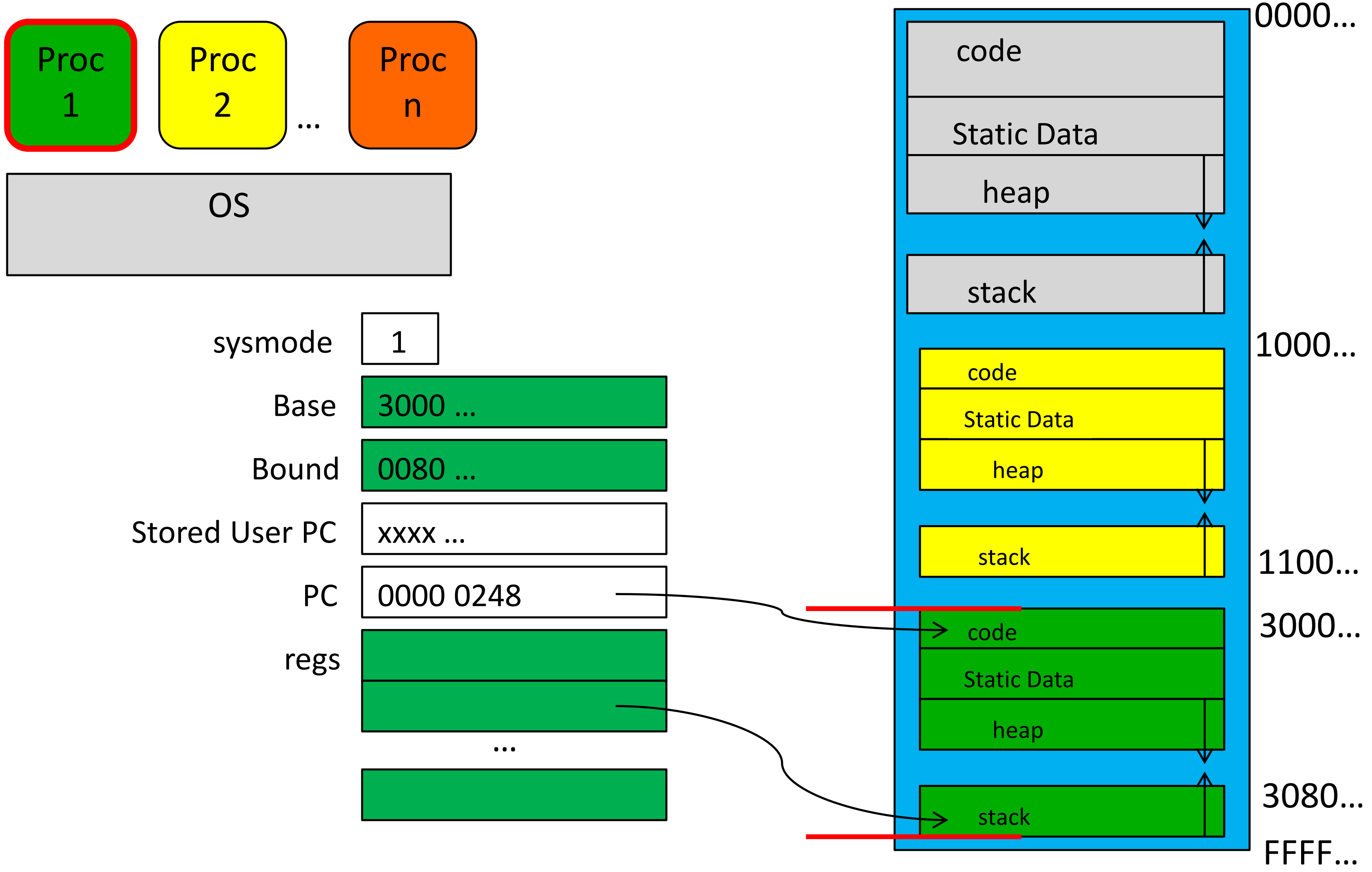
- Switch to kernel mode, set up interrupt handler

Base and Bound: Switch to Process 1

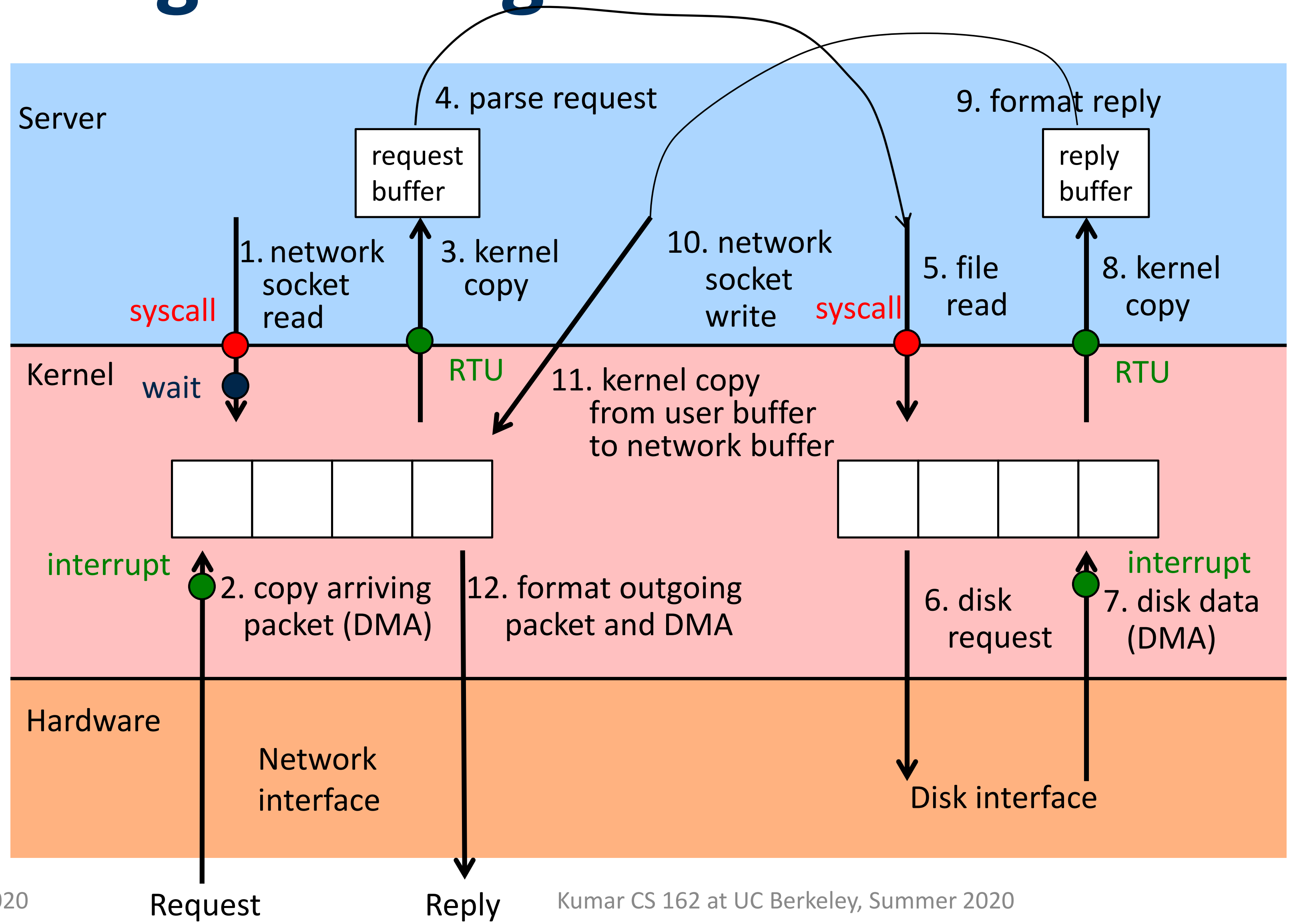


- Save registers of Process 2
- Restore registers of Process 1
- Then execute RTU

Base and Bound: Switch to Process 1



Putting it all Together: Web Server



Conclusion: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address Space** (with **Translation**)
  - Program's view of memory is distinct from physical machine
- **Process: Instance of a Running Program**
  - Address space + one or more threads + …
- **Dual-Mode Operation and Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other