CSC 112: Computer Operating Systems
Lecture 2

Four Fundamental OS Concepts

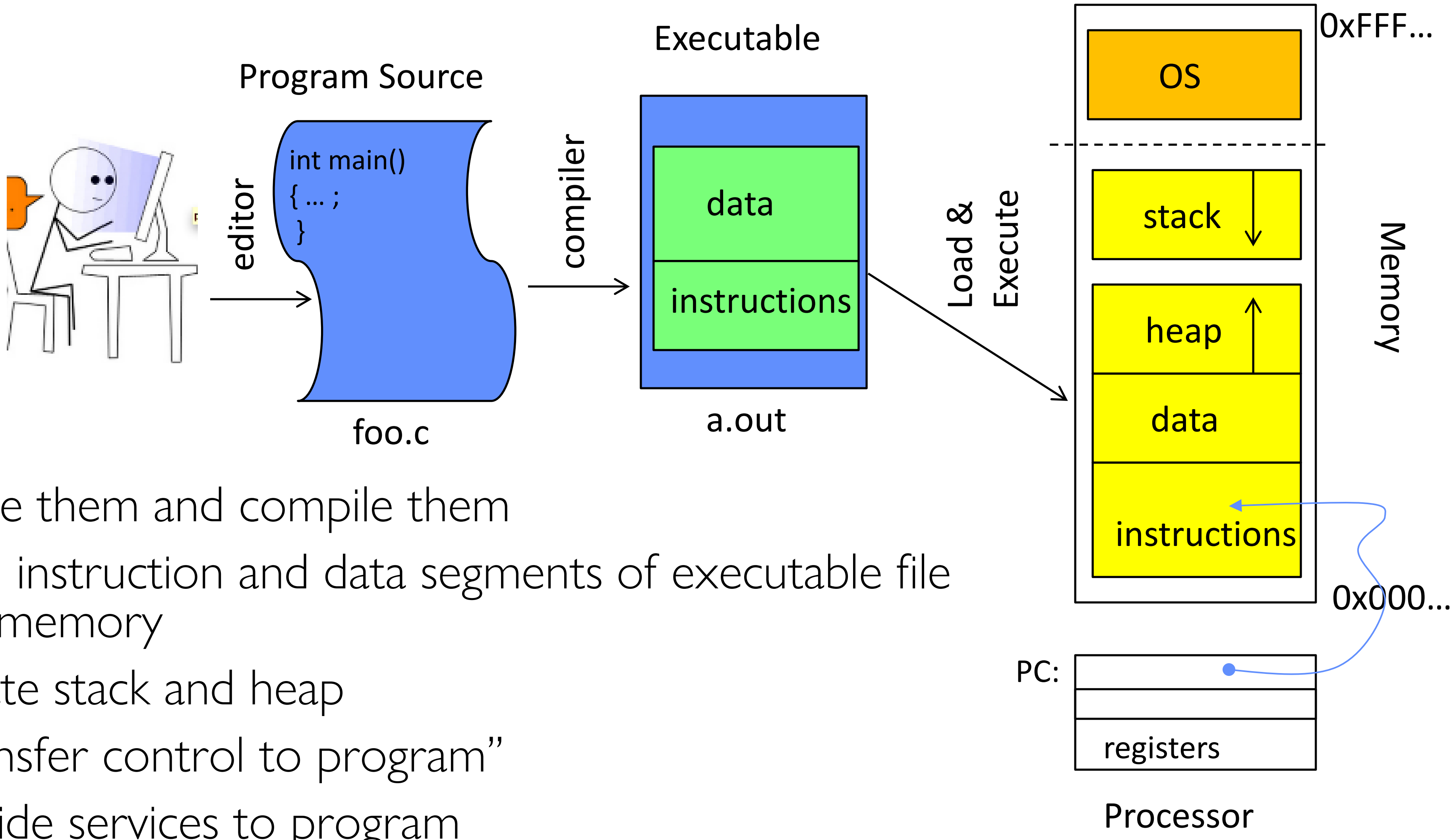
Department of Computer Science,
Hofstra University

Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming

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

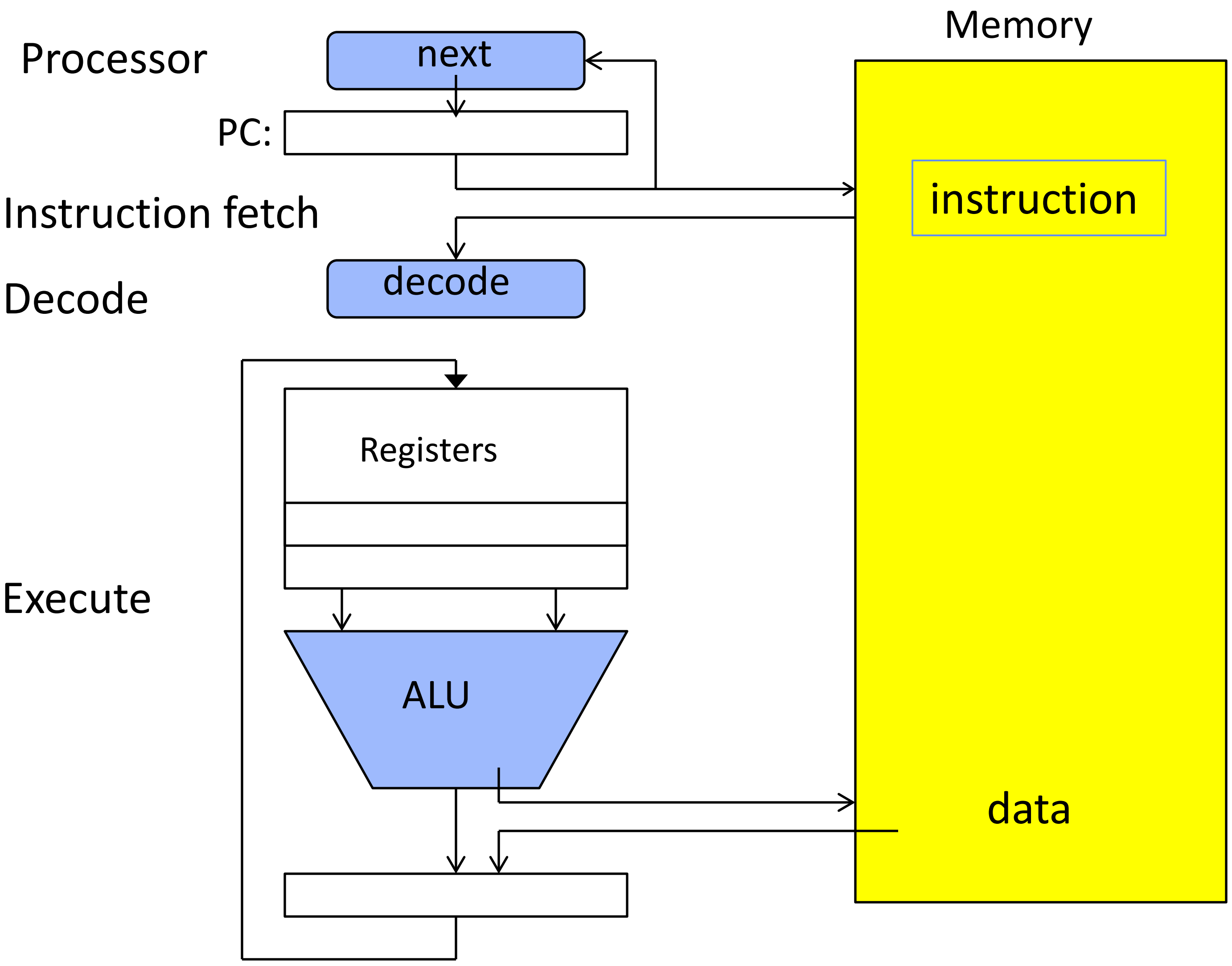
OS Bottom Line: Run Programs



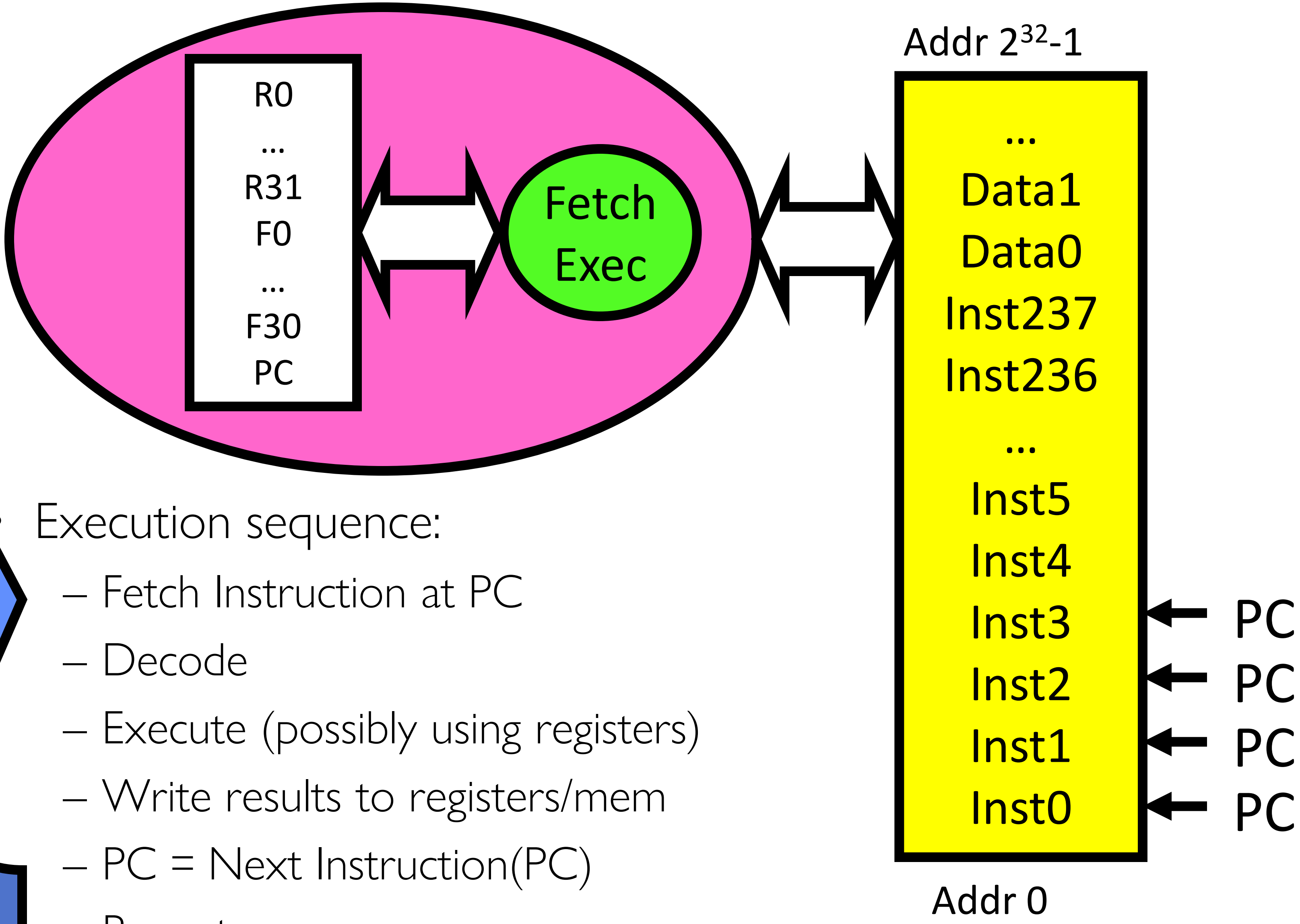
- Write them and compile them
- Load instruction and data segments of executable file into memory
- Create stack and heap
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

Instruction Fetch/Decode/Execute

The instruction cycle



What happens during program execution?

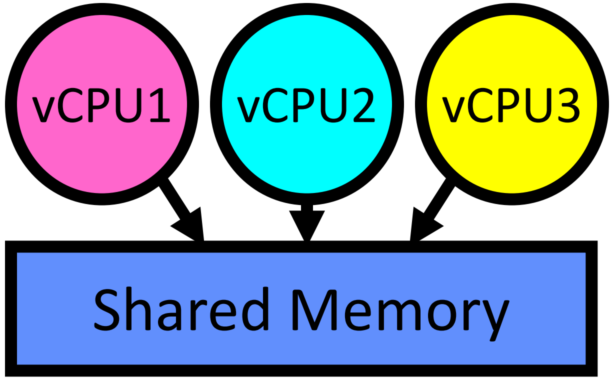


- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

First OS Concept: Thread of Control

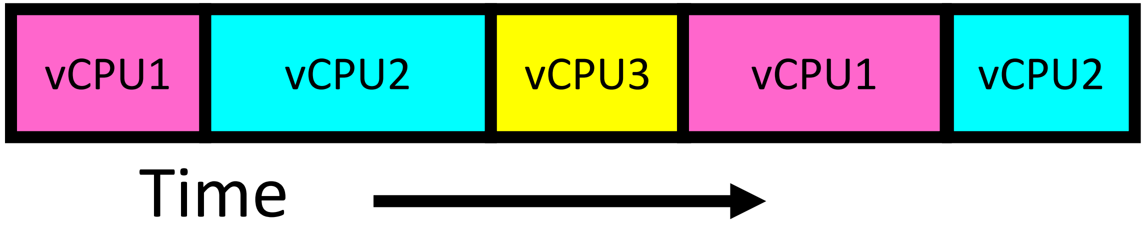
- **Thread**: Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack, Memory State
- A thread is *executing* on a processor (core) when it is *resident* in the processor registers
- Resident means: Registers hold the root state (context) of the thread:
  - Including program 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

Illusion of Multiple Processors



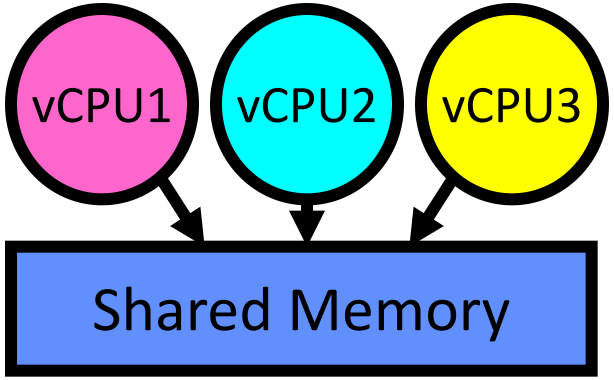
Programmer's View

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



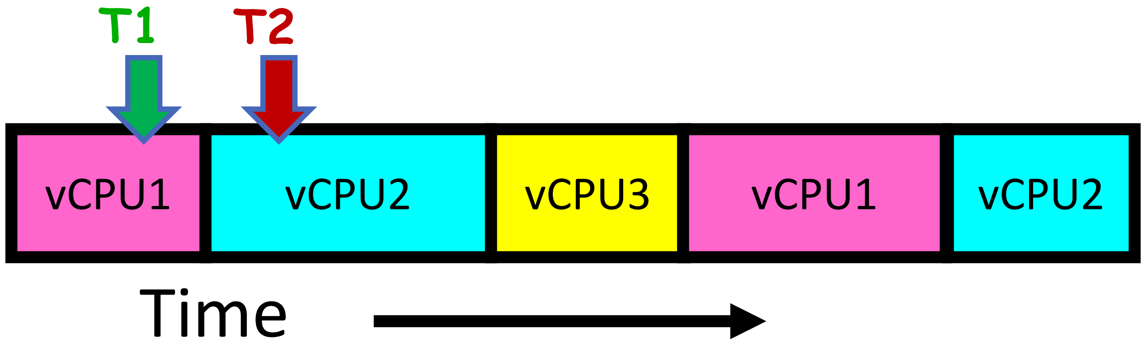
- Contents of virtual core (thread):
  - Program counter, stack pointer
  - Registers
- Where is "it" (the thread)?
  - On the real (physical) core, or
  - Saved in chunk of memory – called the *Thread Control Block (TCB)*

Illusion of Multiple Processors (Continued)



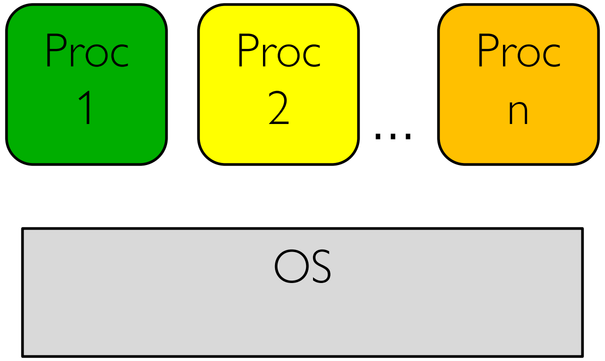
Programmer's View

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

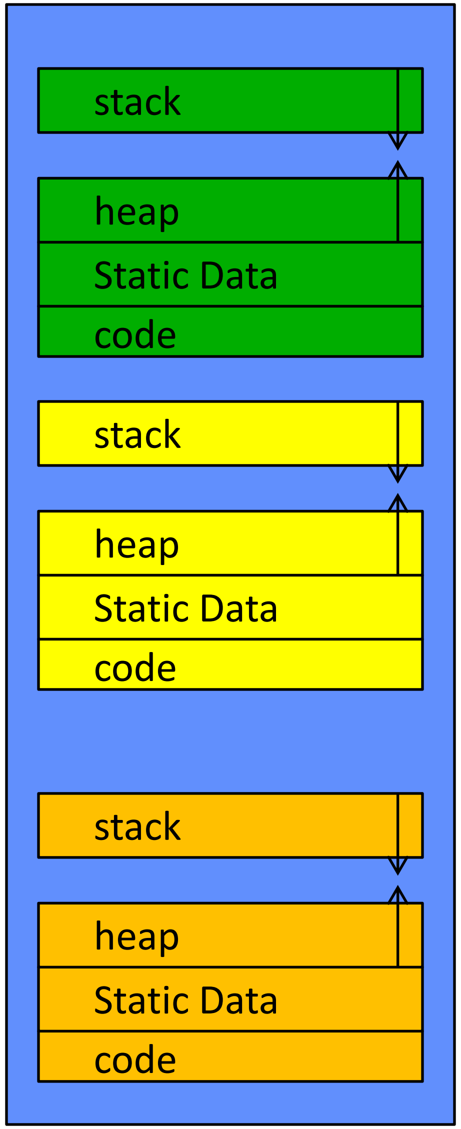


- What happened?
  - OS ran
  - Saved program counter (PC), stack pointer (SP), … in vCPU1's TCB (in memory)
  - Loaded PC, SP, … from vCPU2's TCB, jumped to PC
- What triggered this switch?
  - Timer, voluntary yield, I/O…

Multiprogramming - Multiple Threads of Control



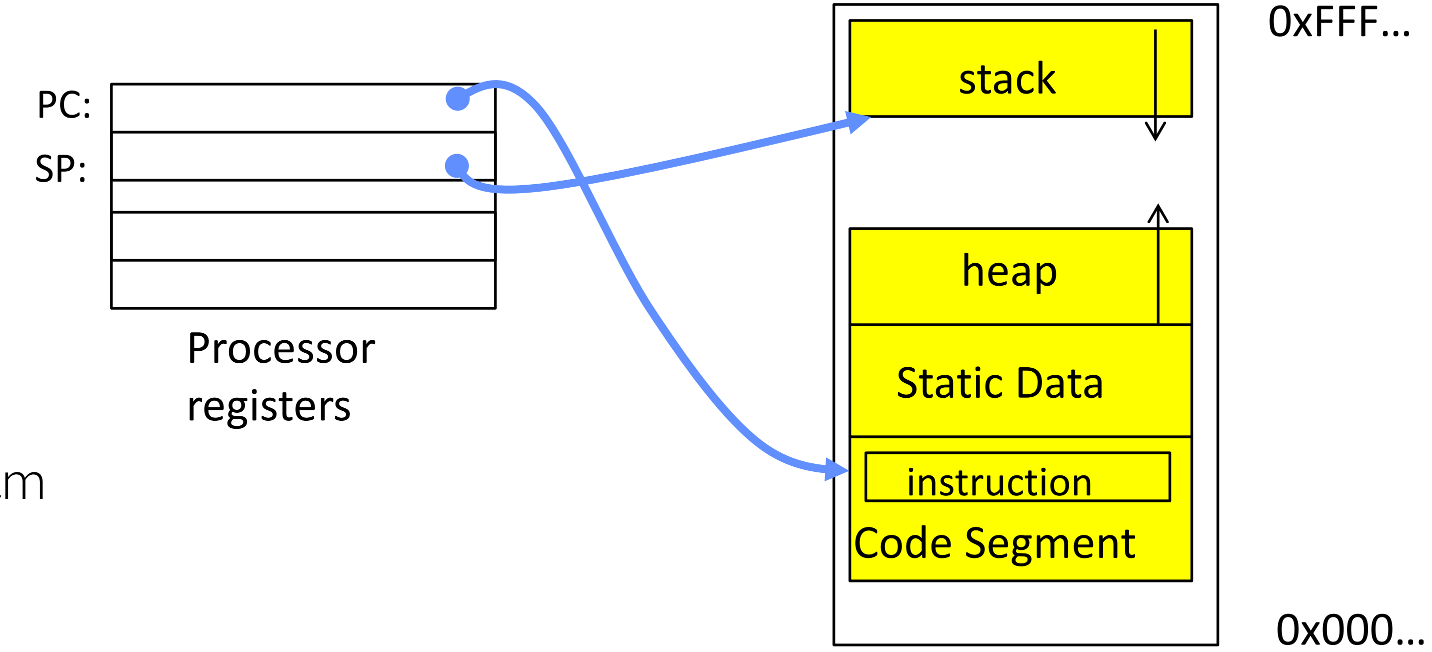
- Thread Control Block (TCB)
  – Holds contents of registers when thread not running
- Where are TCBs stored?
  – In the kernel



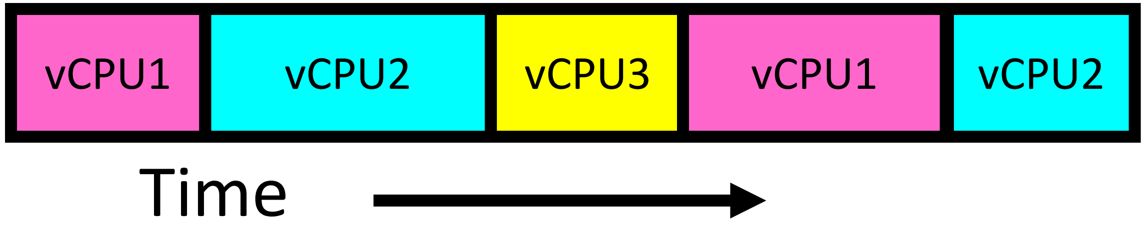
Second OS Concept: Address Space

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

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



Previous discussion of threads: Very Simple Multiprogramming

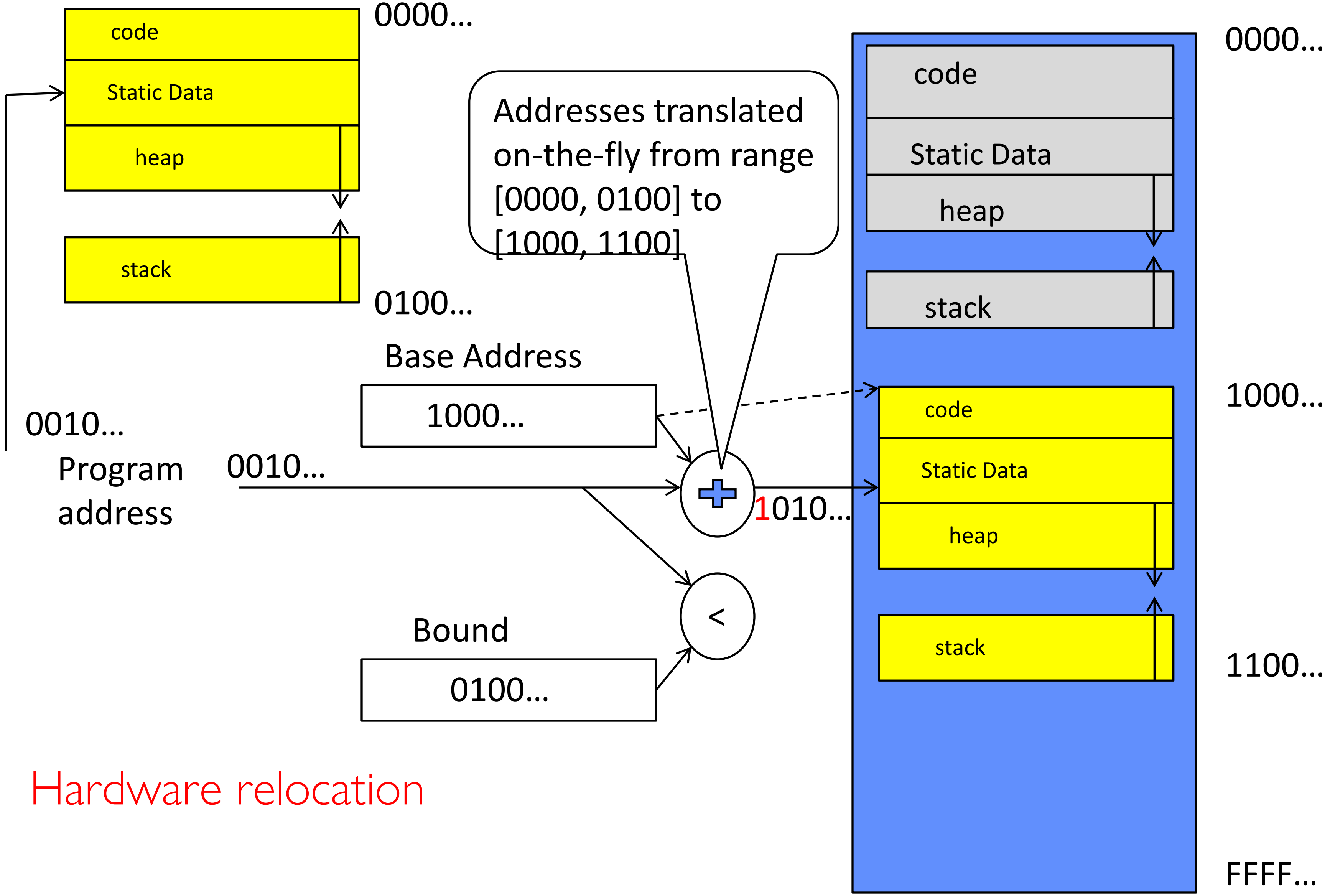


- All vCPU's share non-CPU resources
  - Memory, I/O Devices
- Each thread can read/write memory
  - Perhaps data of others
  - can overwrite OS ?
- This approach is used in
  - Very early days of computing
  - Embedded applications
  - MacOS 1-9/Windows 3.1 (switch only with voluntary yield)
  - Windows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

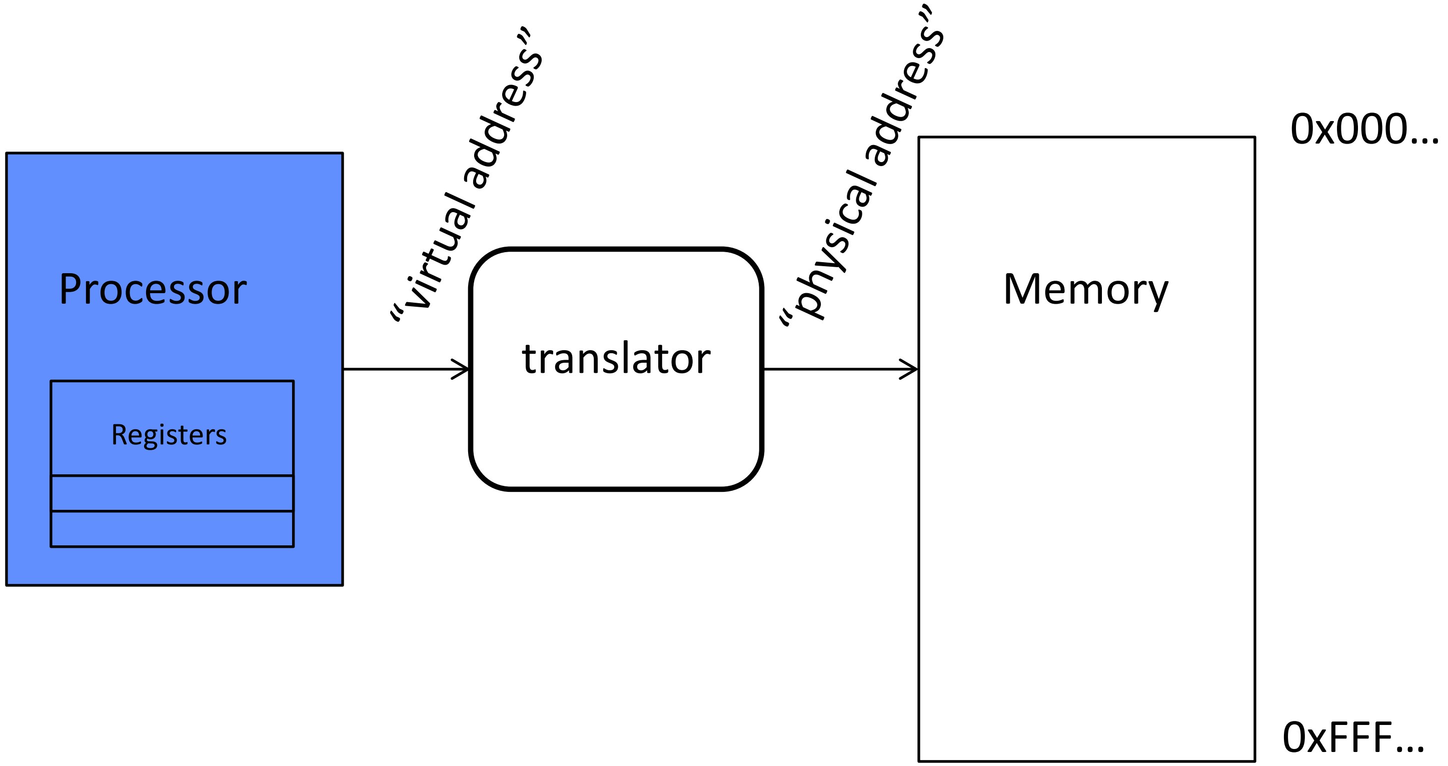
Simple address translation with Base and Bound



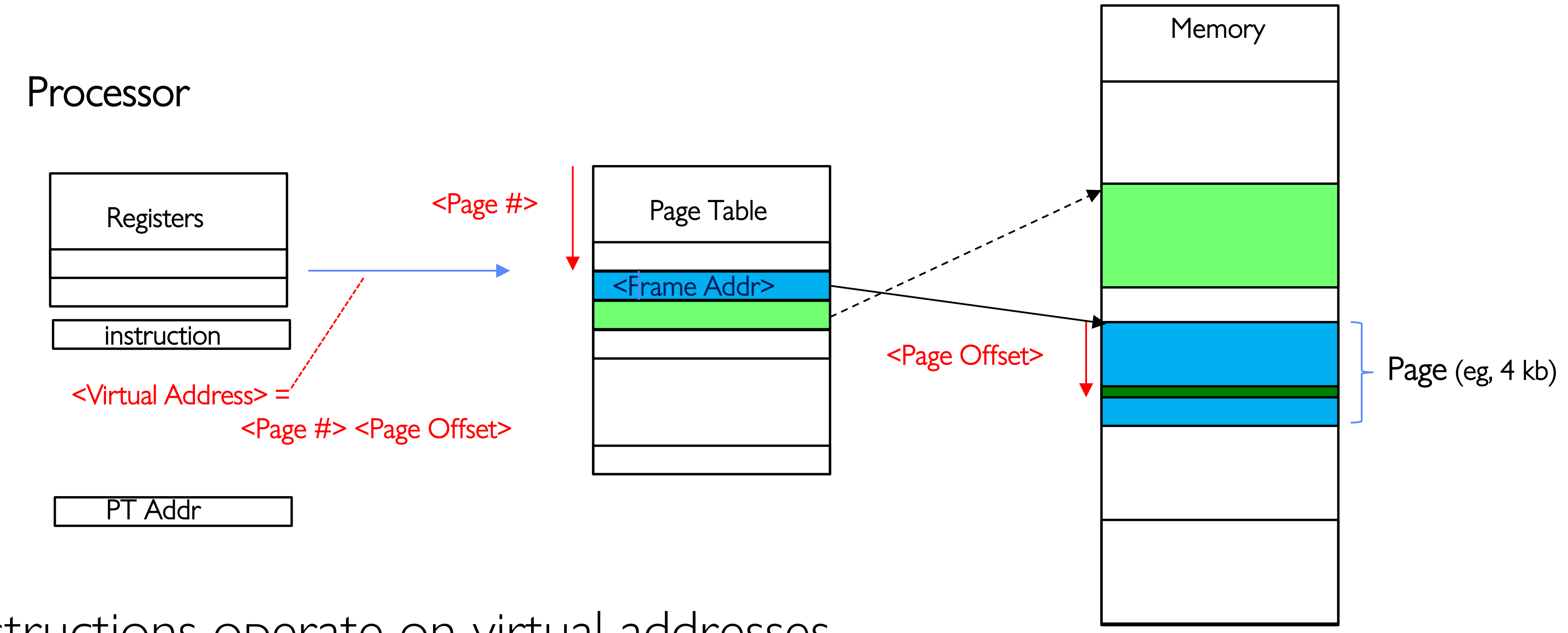
- Hardware relocation

Virtual-to-Physical Address Space Translation

- Program operates in a virtual address space that is distinct from the physical memory space of the machine
  - Break the entire virtual address space into equal size chunks (i.e., pages)
- Hardware translates address using a **page table**
  - Special hardware register stores pointer to page table
  - Treat memory as page size frames and put any page into any frame …



Paged Virtual Address

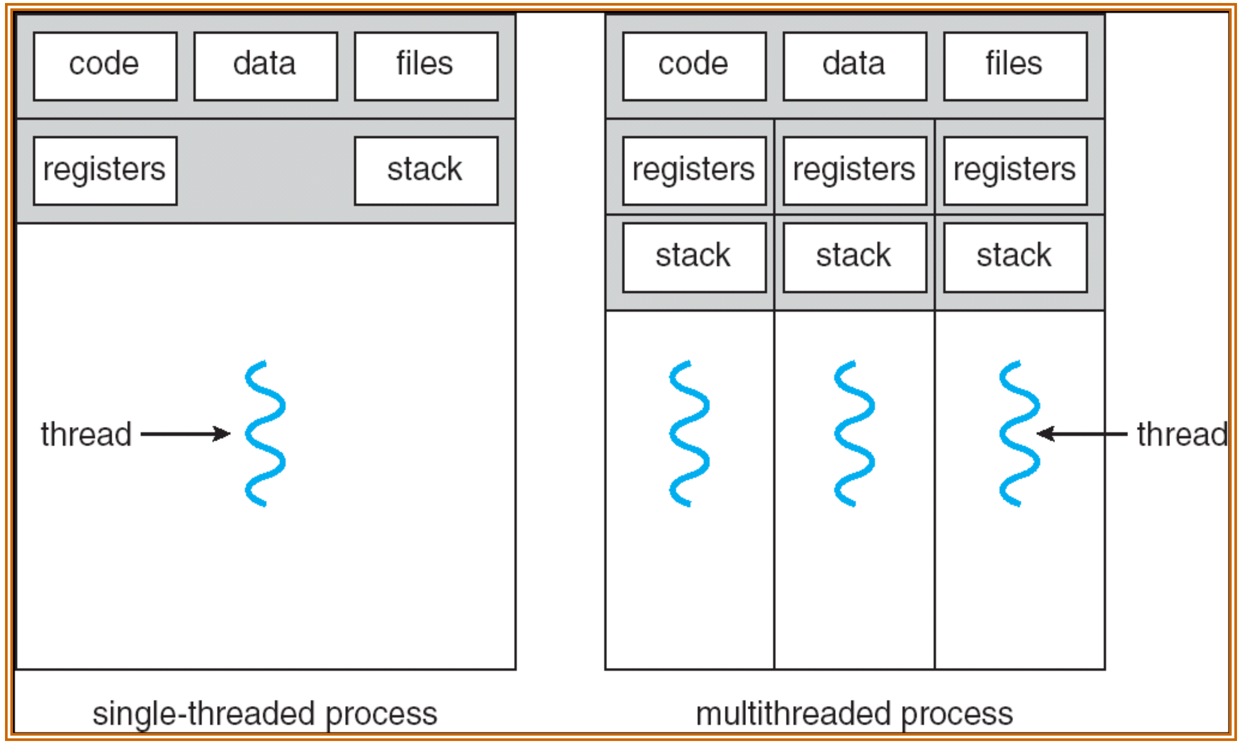


- Instructions operate on virtual addresses
  - Instruction address, load/store data address
- Translated to a physical address through a Page Table by the hardware
- Any Page of address space can be in any (page sized) frame in memory
  - Or not-present (access generates a page fault)

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
- 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



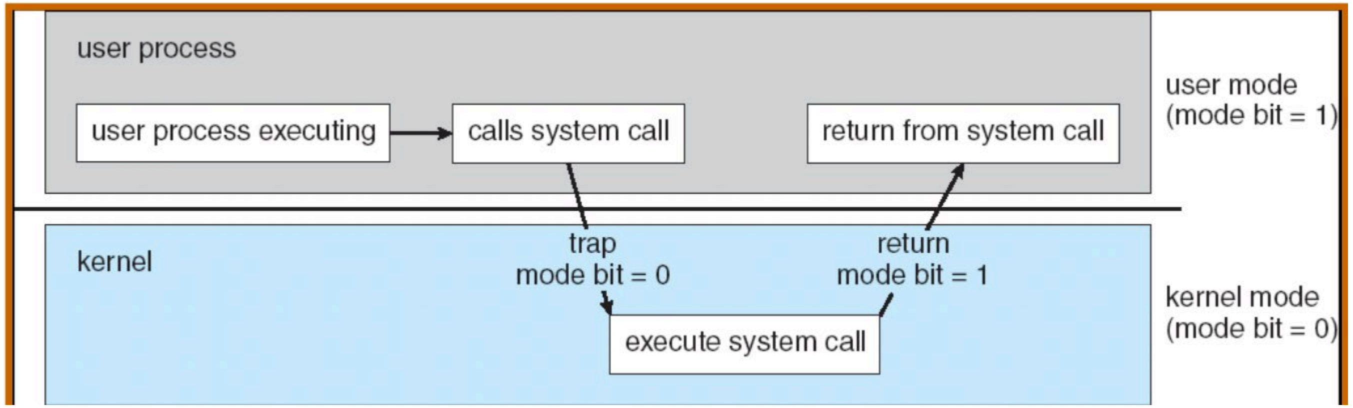
- Threads encapsulate concurrency:
  - "Active" component
- Address spaces encapsulate protection:
  - "Passive" component
  - Keeps buggy programs from crashing the system
- Why have multiple threads per address space?
  - Parallelism: take advantage of actual hardware parallelism (e.g. multicore)
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

Protection and Isolation

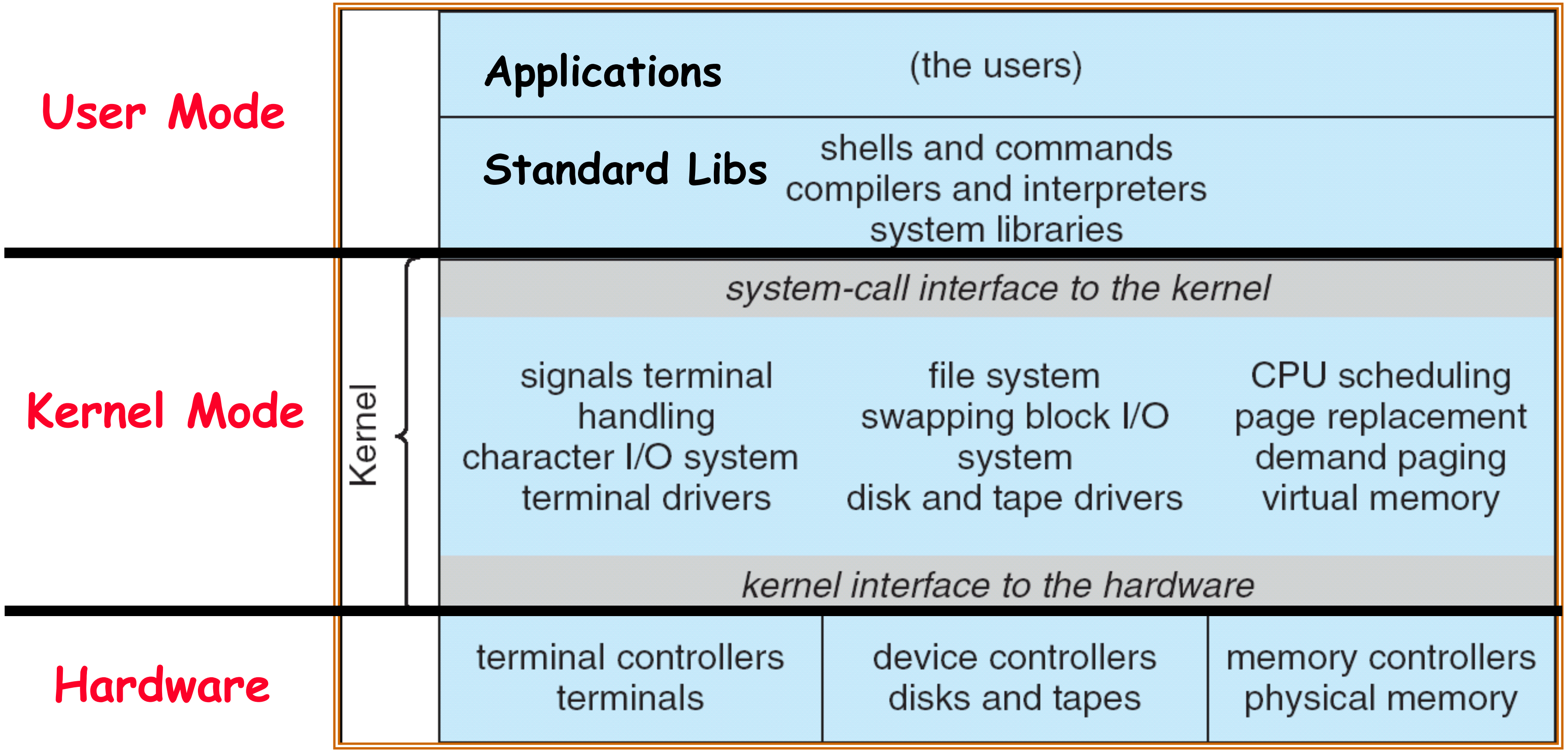
- Why Do We Need Processes??
  - Reliability: bugs can only overwrite memory of process they are in
  - Security and privacy: malicious or compromised process can't read or write other process' data
  - Fairness: enforce shares of disk, CPU
- Mechanisms:
  - Address translation: address space only contains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

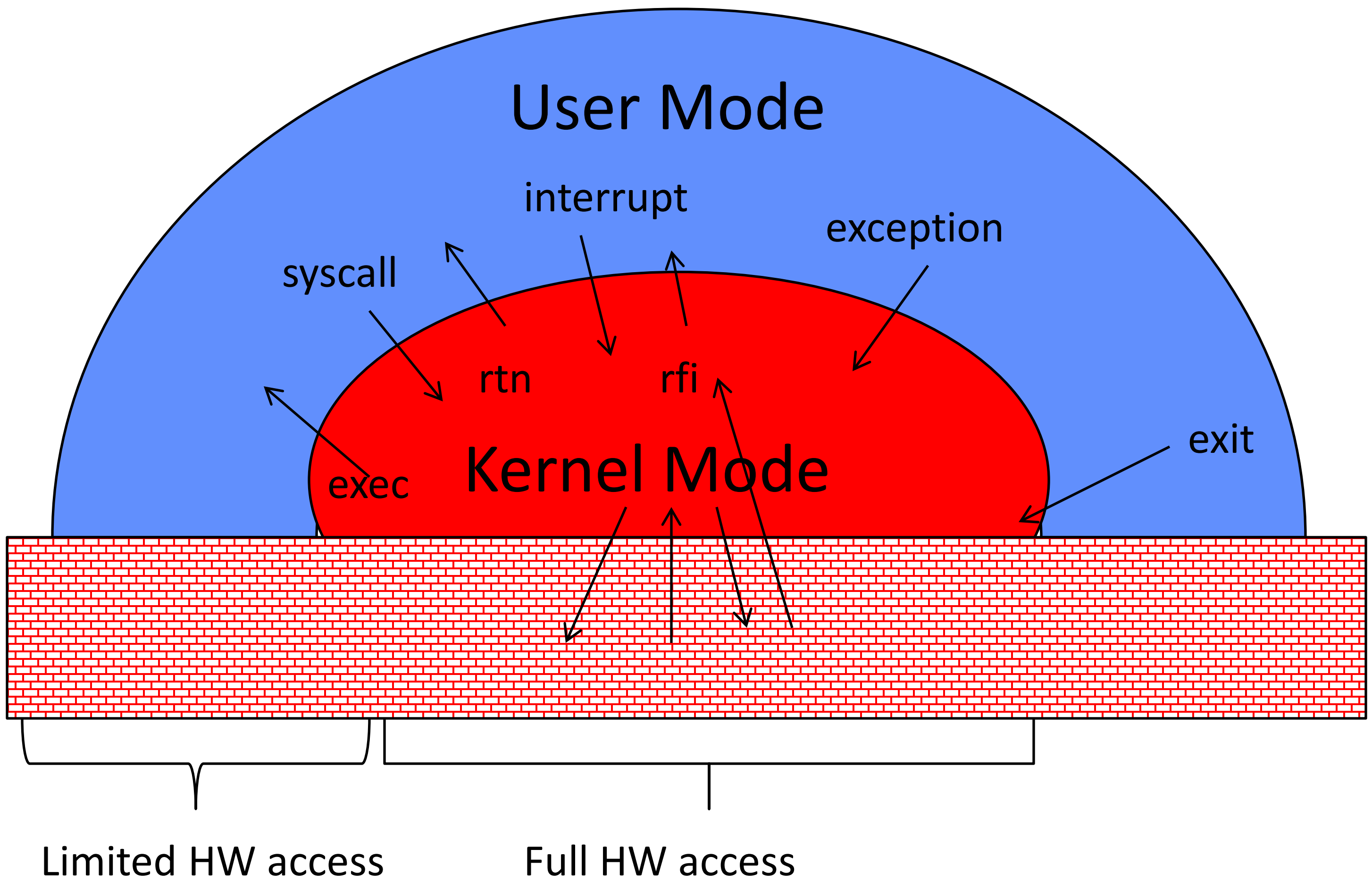
- Hardware provides at least two modes (at least 1 mode bit):
  1. Kernel Mode (or "supervisor" mode)
  2. User Mode
- Certain operations are prohibited when running in user mode
  – Changing the page table pointer, disabling interrupts, interacting directly w/ hardware, writing to kernel memory
- Carefully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



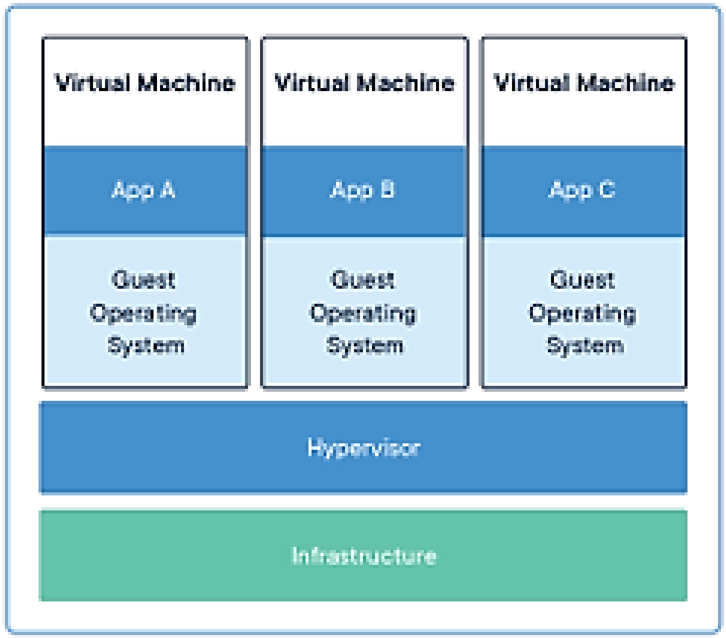
For example: UNIX System Structure

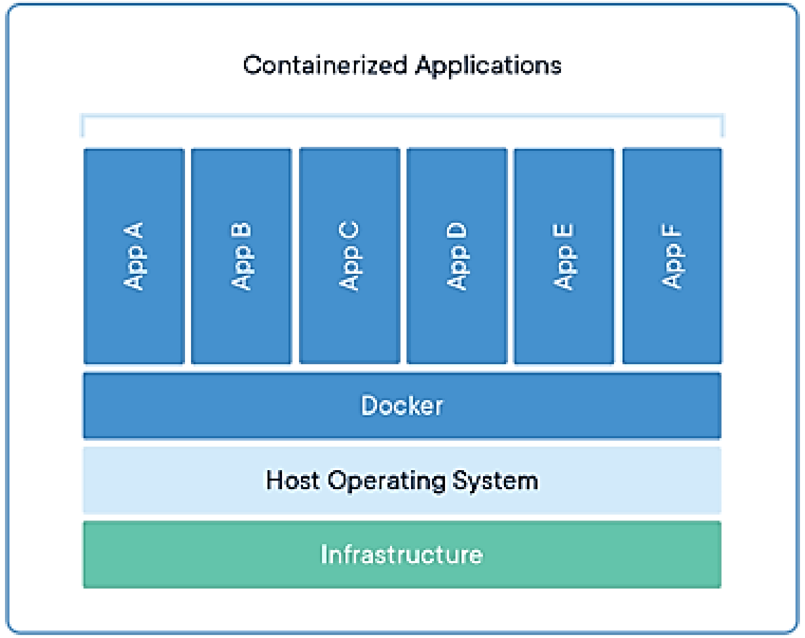


User/Kernel (Privileged) Mode



Additional Layers of Protection for Modern Systems

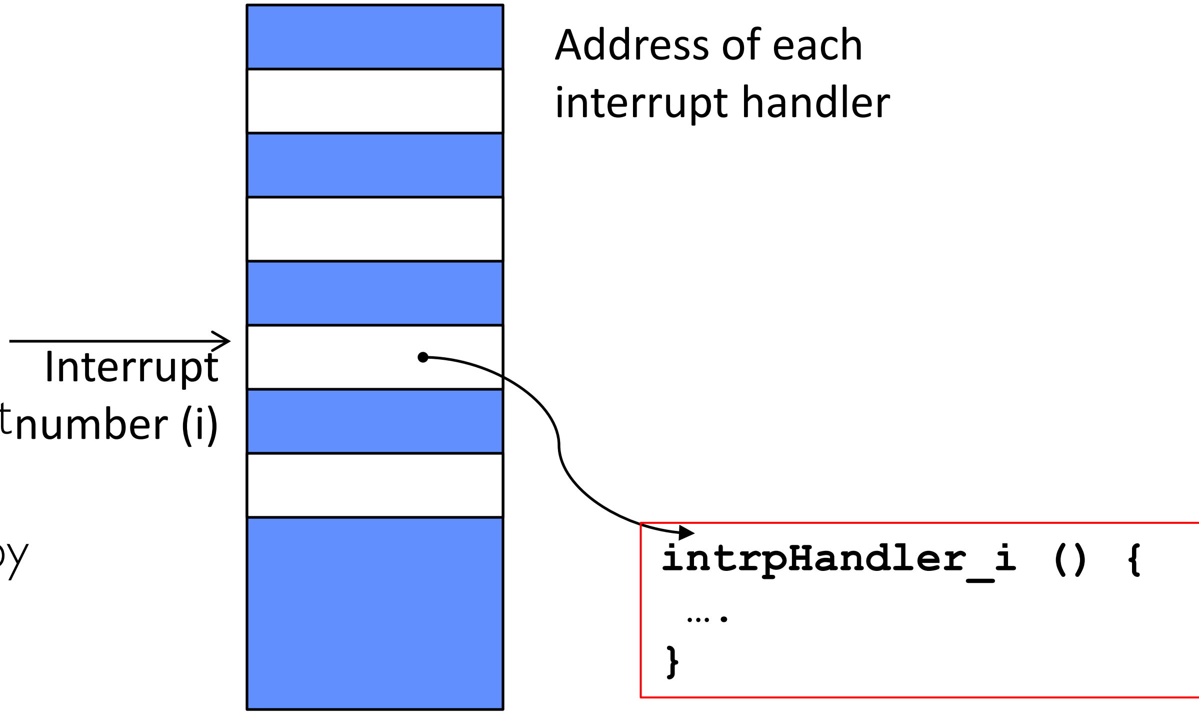




- Additional layers of protection through virtual machines or containers
  - Run a complete operating system in a virtual machine
  - Package all the libraries associated with an app into a container for execution
- More on these ideas later in the class

3 types of User ⇒ Kernel Mode Transfer

- Syscall
  - Process requests a system service, e.g., exit
  - Put 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?



Interrupt Vector

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?
- …

Process Control Block and Scheduler

- 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

```
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)
  - Virtual address space distinct from physical address space of the machine
- **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