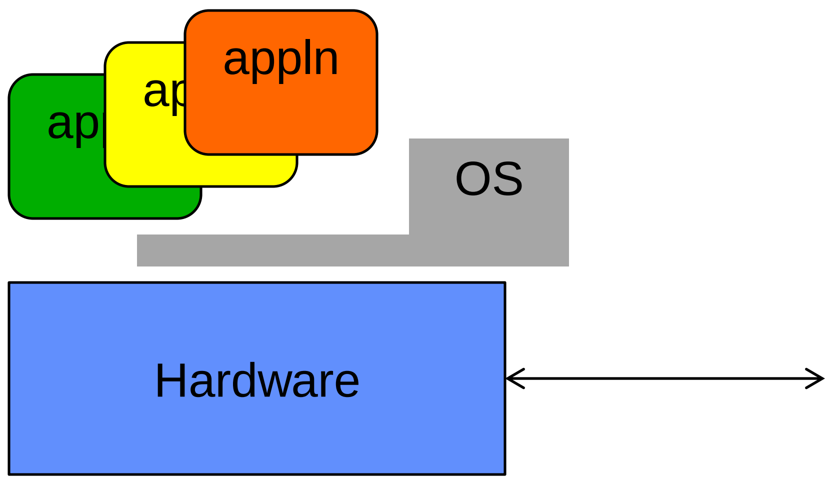
CSE150
Operating Systems
Lecture 2

Processes, Threads and Address Spaces

Recall: What is an operating system?

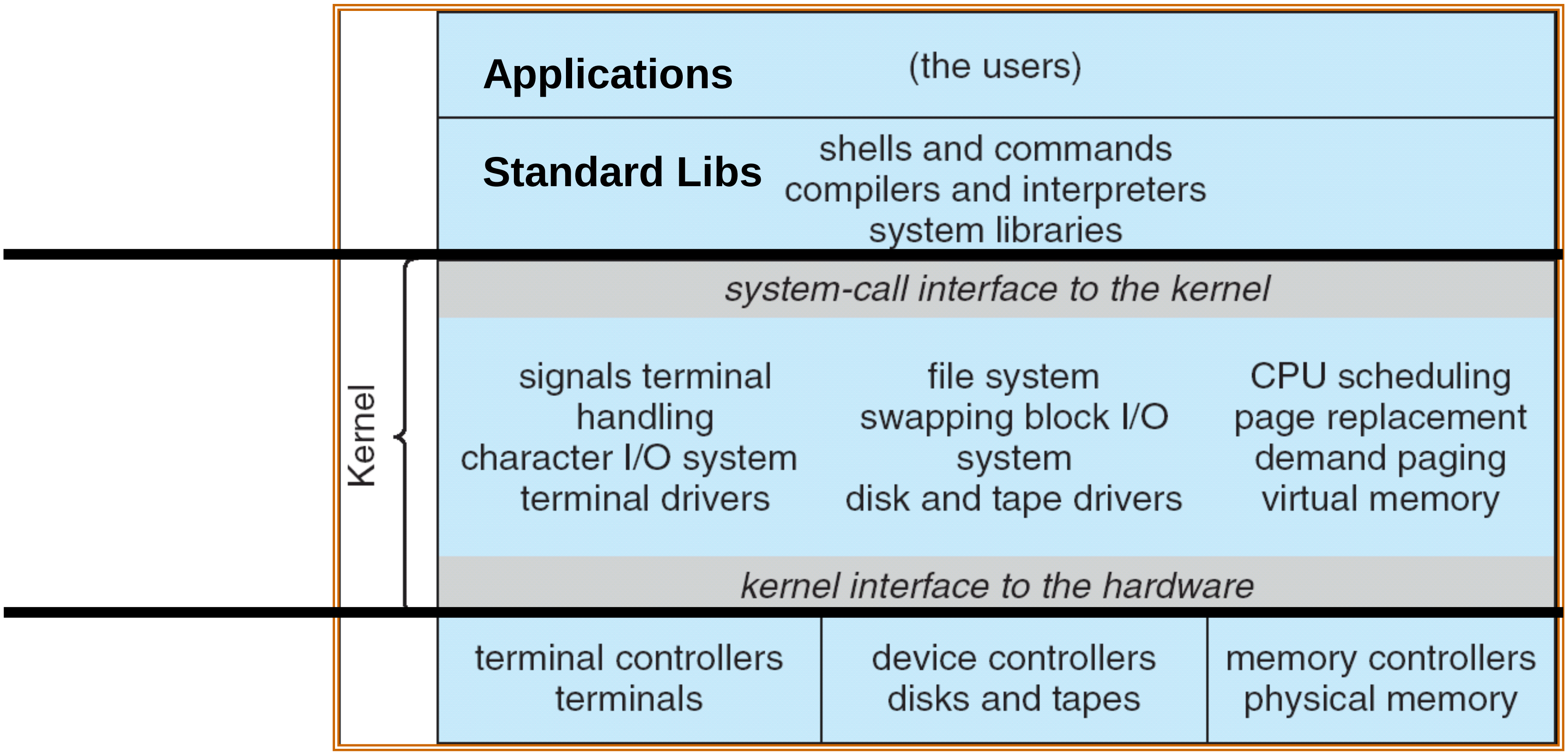
- Special layer of software that provides application software access to hardware resources
  - – Convenient abstraction of complex hardware devices
  - – Protected access to shared resources
  - – Security and authentication
  - – Communication amongst logical entities



Today: Four Fundamental OS Concepts

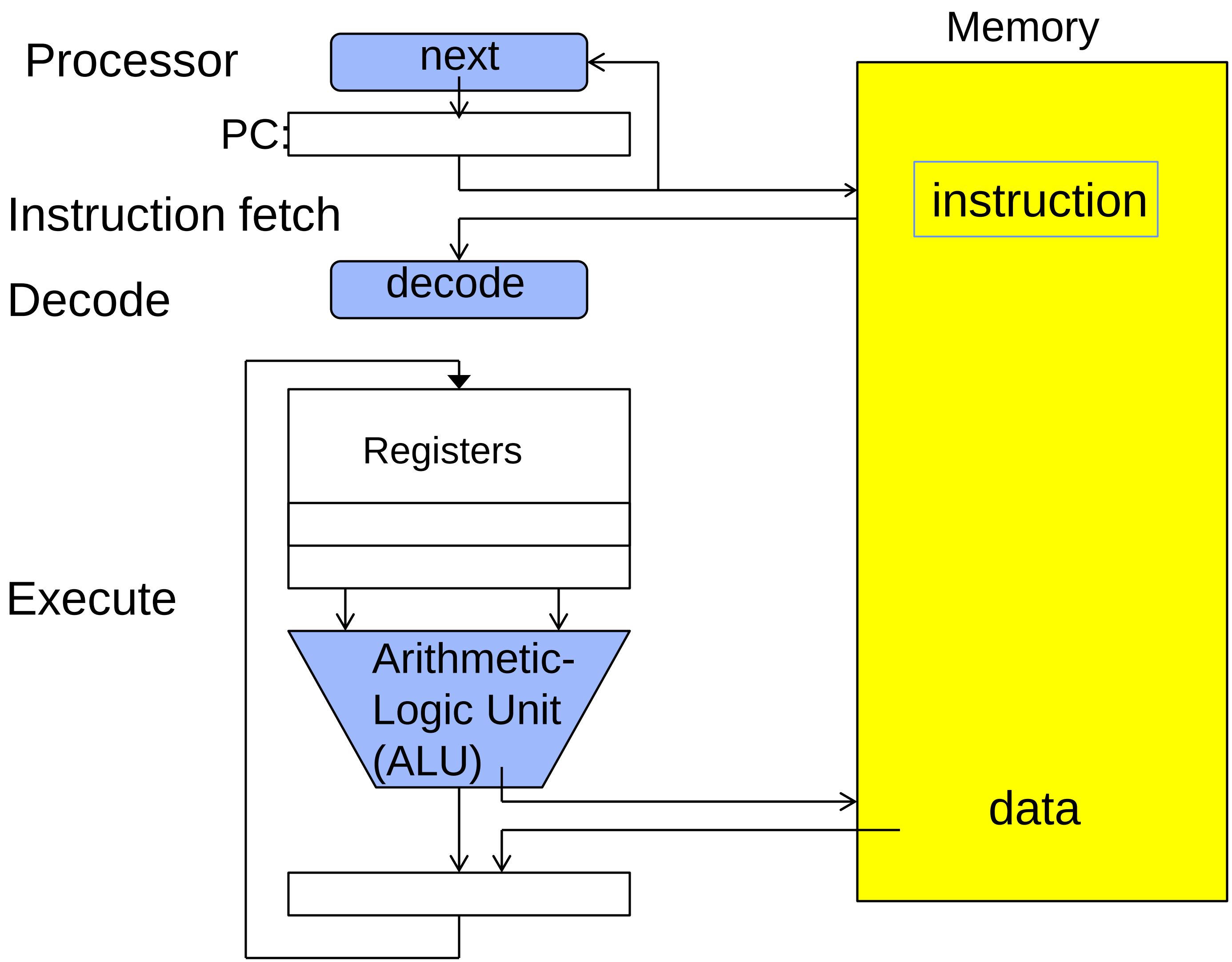
- Thread
  - Single unique execution context: fully describes program state
  - Program counter, Registers, Stack
- Address space (with translation)
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine
- Process
  - An instance of an executing program is *a process consisting of an address space and one or more threads*
- 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

UNIX System Structure

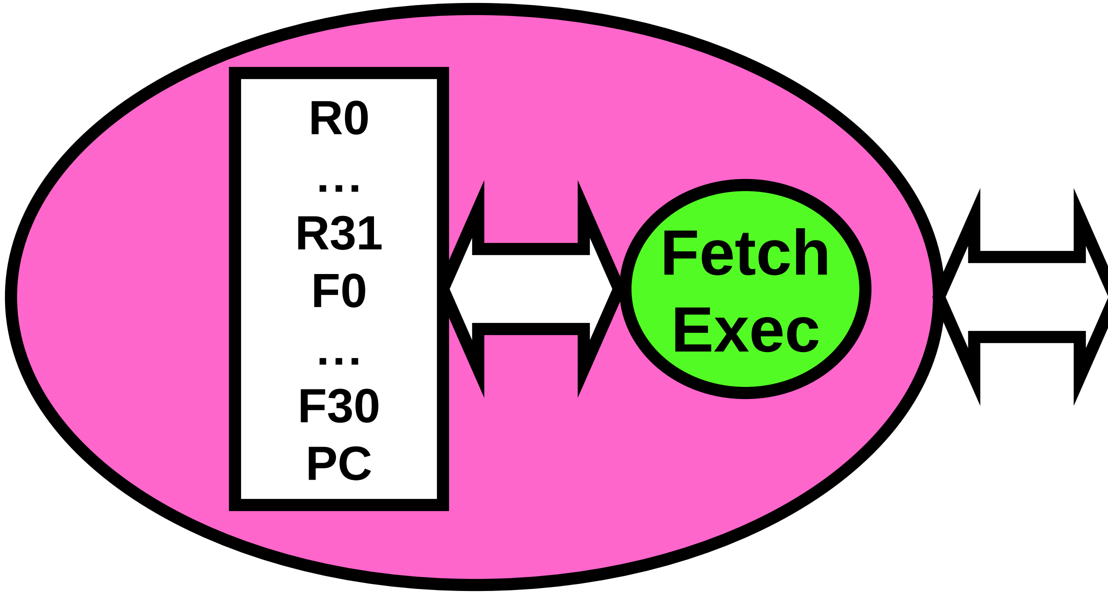


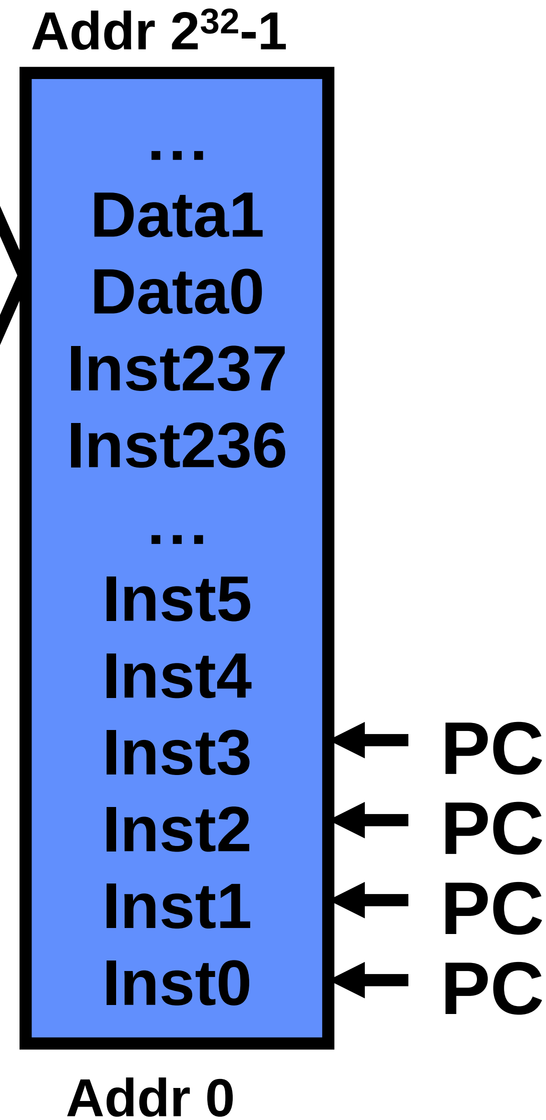
Recall (31): Instruction Fetch/Decode/Execute

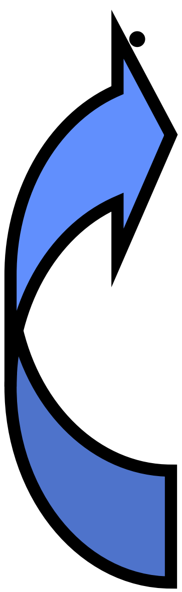
The instruction cycle



Recall (31): What happens during execution?

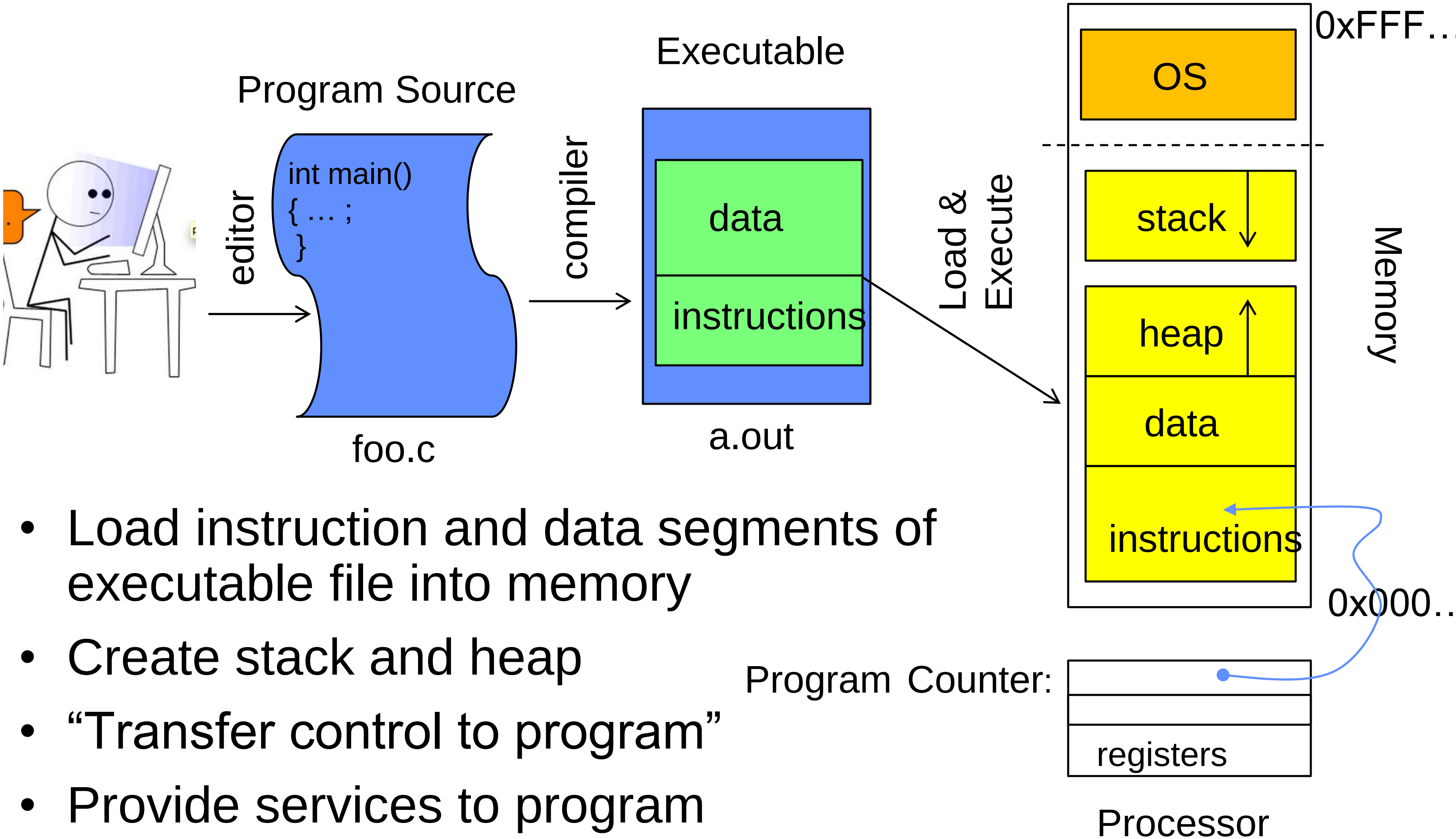






- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

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

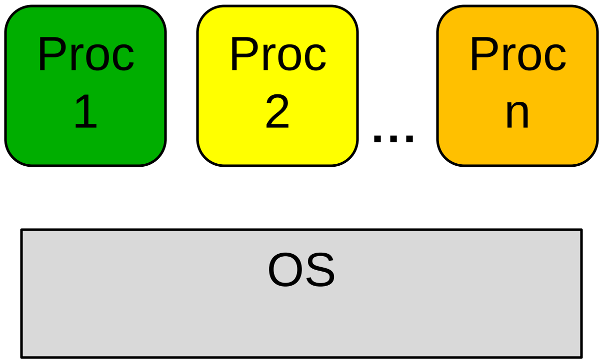
First OS Concept: Thread of Control

- Thread: Single unique execution context
  - Program Counter, Registers, Stack
- A thread is executing on a processor when it is resident in the processor registers.
- Program Counter (PC) register holds the address of executing instruction in the thread
- 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
- Registers hold the root state of the thread.
  - The rest is "in memory"

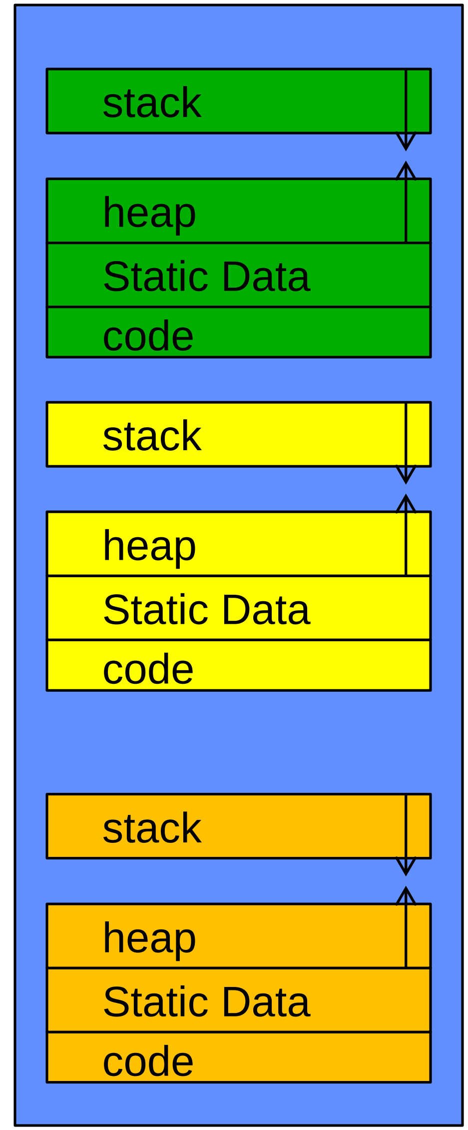
Concurrency

- “Thread” of execution
  – Independent Fetch/Decode/Execute loop
  – Operating in some Address space
- Uniprogramming: *one thread at a time*
  – MS/DOS, early Macintosh, Batch processing
  – Easier for operating system builder
  – Get rid concurrency by defining it away
  – Does this make sense for personal computers?
- Multiprogramming: *more than one thread at a time*
  – Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Mac OS X
  – Often called “multitasking”, but multitasking has other meanings (talk about this later)
- ManyCore $\Rightarrow$ Multiprogramming, right?

Multiprogramming - Multiple Threads



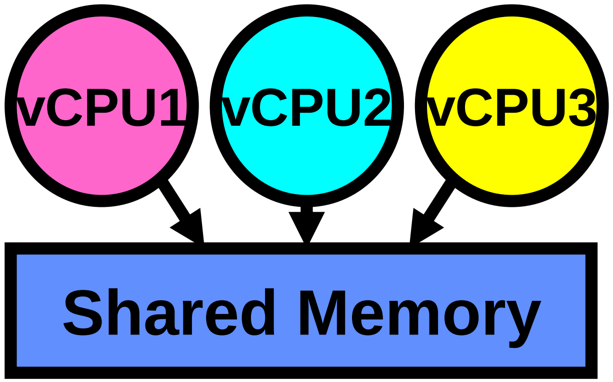
- Easy when multiple # processor >= # threads
- But typically not the case
  – Challenge: How do we make this work?
  – Extreme case: 1 processor

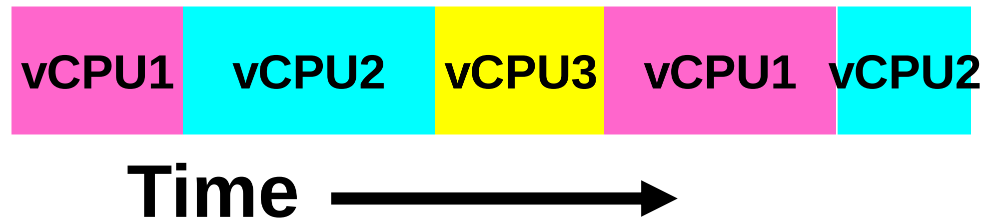


The Basic Problem of Concurrency

- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: users think they have exclusive access to shared resources
- OS Has to coordinate all activity
  - Multiple users, I/O interrupts, …
  - How can it keep all these things straight?
- Basic Idea: Use Virtual Machine abstraction
  - Simple machine abstraction for processes
  - Then, worry about multiplexing these abstract machines

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 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

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 common in:
  - Embedded applications
  - Windows 3.1/Machintosh (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)
  - Separation: 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, I/O instructions, special registers
  - syscall processing, subsystem implementation
    - » (e.g., file access rights, etc)

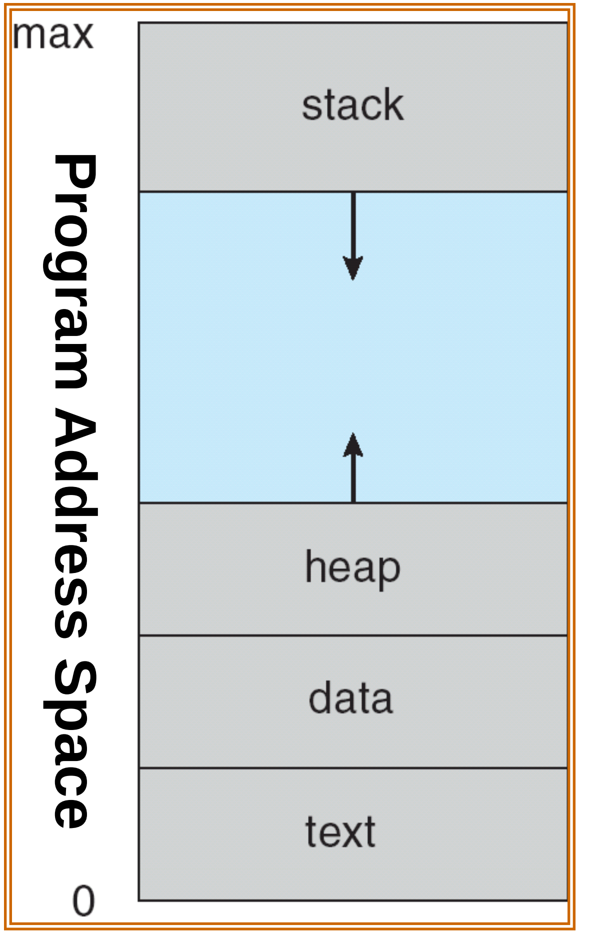
How to protect threads from one another?

- Need three important things:

1. Protection of memory
   - » Every task does not have access to all memory
2. Protection of I/O devices
   - » Every task does not have access to every device
3. Protection of Access to Processor:
   Preemptive switching from task to task
   - » Use of timer
   - » Must not be possible to disable timer from usercode

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)



Summary

- Two OS concepts so far
  – Threads (Concurrency)
  – Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  – Unloading current thread (PC, registers)
  – Loading new thread (PC, registers)
  – Such context switching may be voluntary (yield(), I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished by restricting access:
  – Memory mapping isolates processes from each other
  – Dual-mode for isolating I/O, other resources (later)