Chapter 3: Introduction to Processes

CSCI 3753 Operating Systems

William Mortl, MS & Rick Han, PhD

Spring 2016

University of Colorado Boulder

Announcements

- Quiz #3 is up early for hackathon kids (deadline is still February 2nd
- Programming Assignment #1 is due on February 2nd… email me or TA's to use the 1 week extension (20% penalty)
- Computer architecture is not the focal point of this course, but we had to cover in order to talk about OS's... Boot loading started the REAL part of this course

Recap…

- Booting
  1. POST – Power On Self Test
  2. BIOS – Basic Input Output System loads the 512 byte boot loader
  3. 512 byte boot loader loads secondary stage boot loader (GRUB, LILO)
  4. Second stage boot loader loads the OS kernel

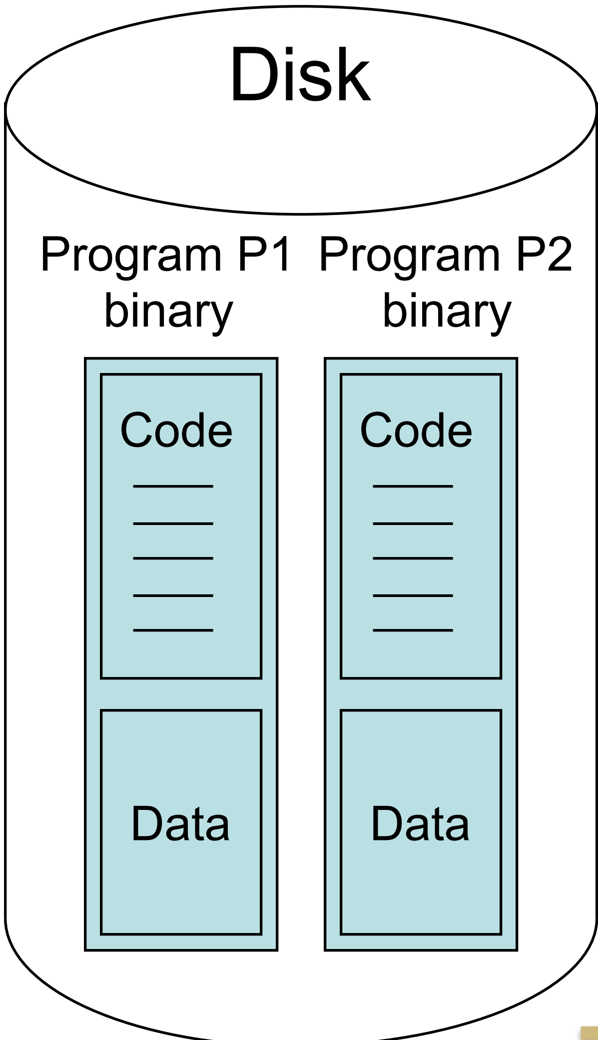
Bootstrapping the OS in PCs

- Multi-stage procedure:
  1. Power On Self Test (POST) from ROM
     - Check hardware, e.g. CPU and memory, to make sure it's OK
  2. BIOS (Basic Input/Output System) looks for a device to boot from…
     - May be prioritized to look for a USB flash drive or a CD/DVD-ROM drive before a hard disk drive
     - Can also boot from network
  3. BIOS finds a hard disk drive to boot from
     - Looks at Master Boot Record (MBR) in sector 0 of disk
     - Only 512 bytes long (Intel systems), contains primitive code for later stage loading and a partition table listing an active partition, or the location of the bootloader

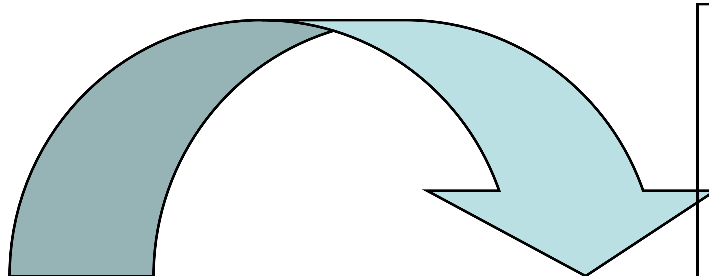
Bootstrapping the OS in PCs

- Multi-stage procedure: (continued)
  4. Primitive loader then loads the secondary stage bootloader
     - Examples of this bootloader include LILO (Linux Loader), and GRUB (Grand Unified Bootloader)
     - Can select among multiple OS's (on different partitions) – i.e. dual booting
     - Once OS is selected, the bootloader goes to that OS's partition, finds the boot sector, and starts loading the OS's kernel

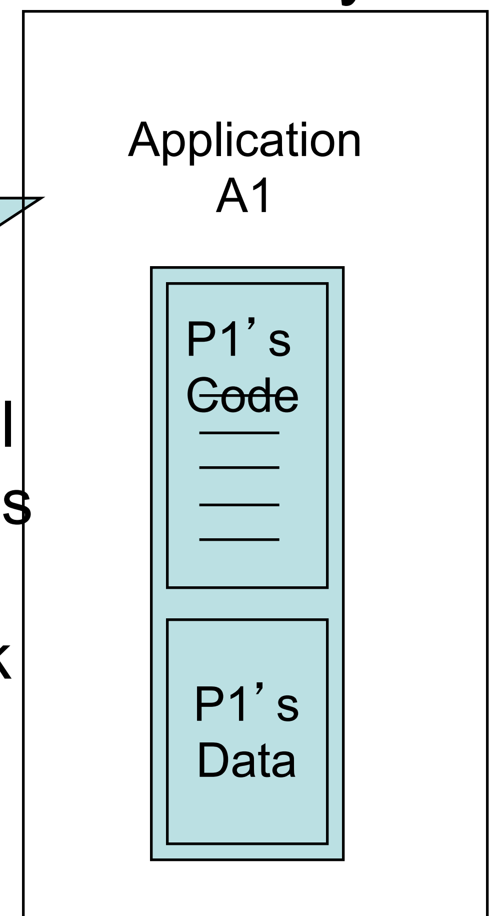
Loading a Program into Memory



OS Loader



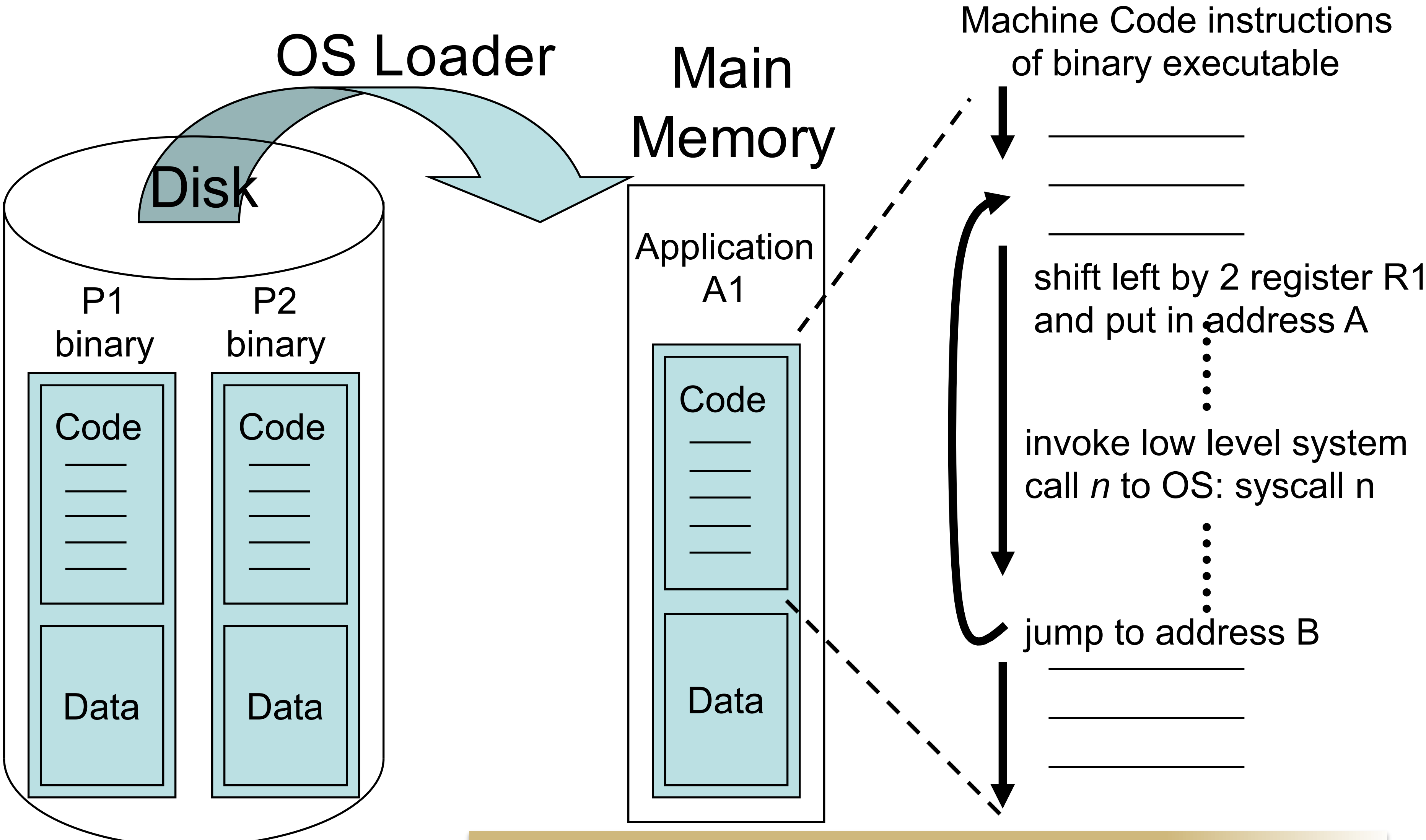
Main Memory



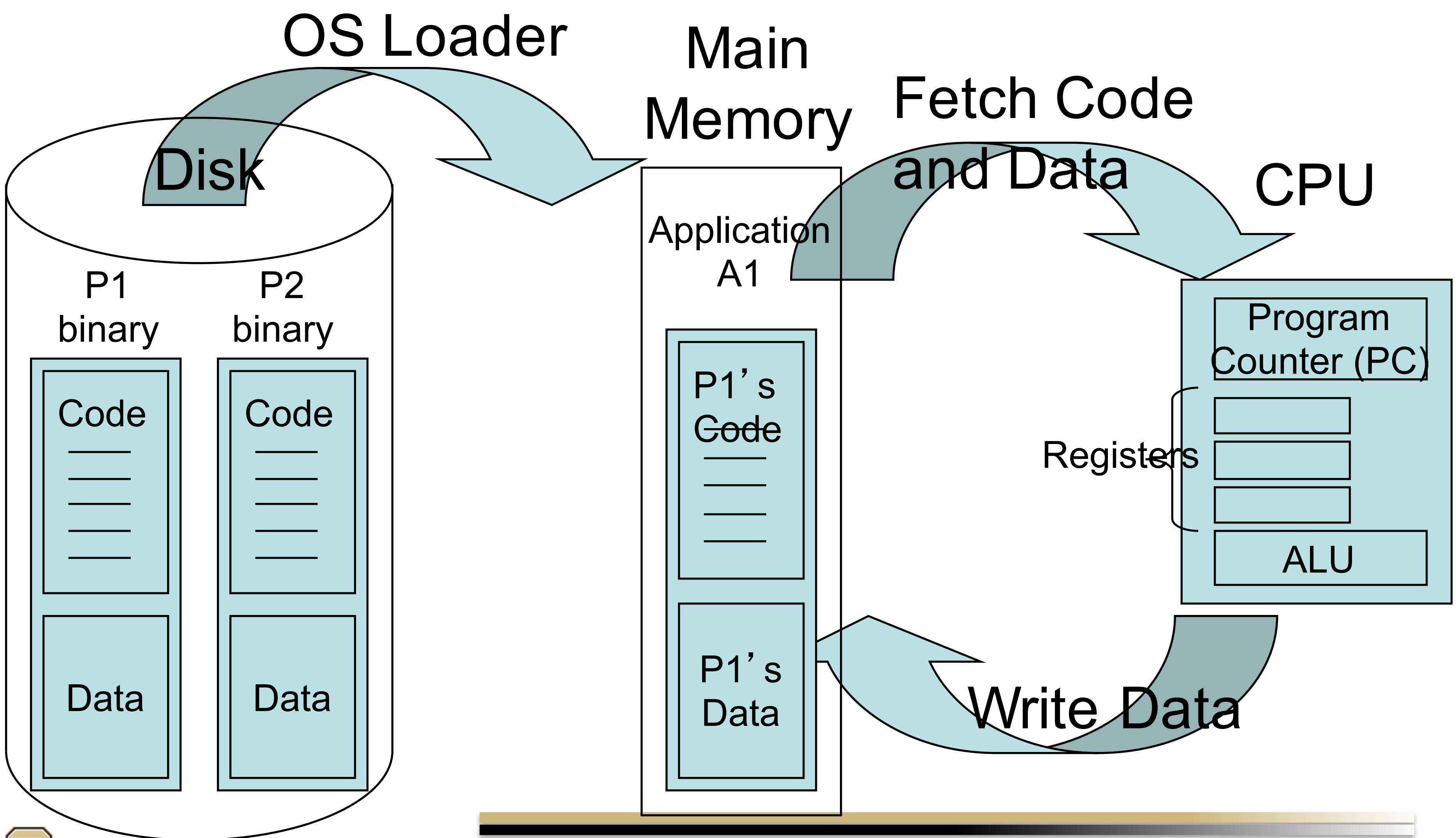
- Invoked by typing a program name in shell or doubleclicking on its icon
- Copies P1 from disk to RAM

In reality, more complex, execve system call and paging are involved

Loading and Executing a Program

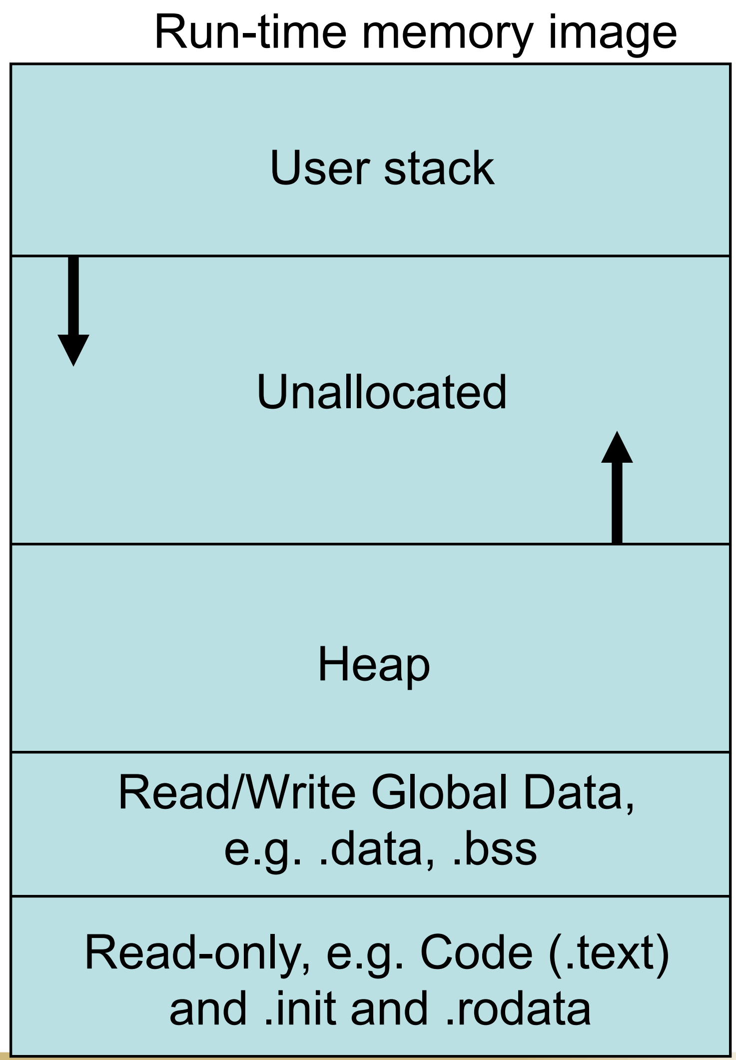


Loading and Executing a Program



Loading Executable Object Files

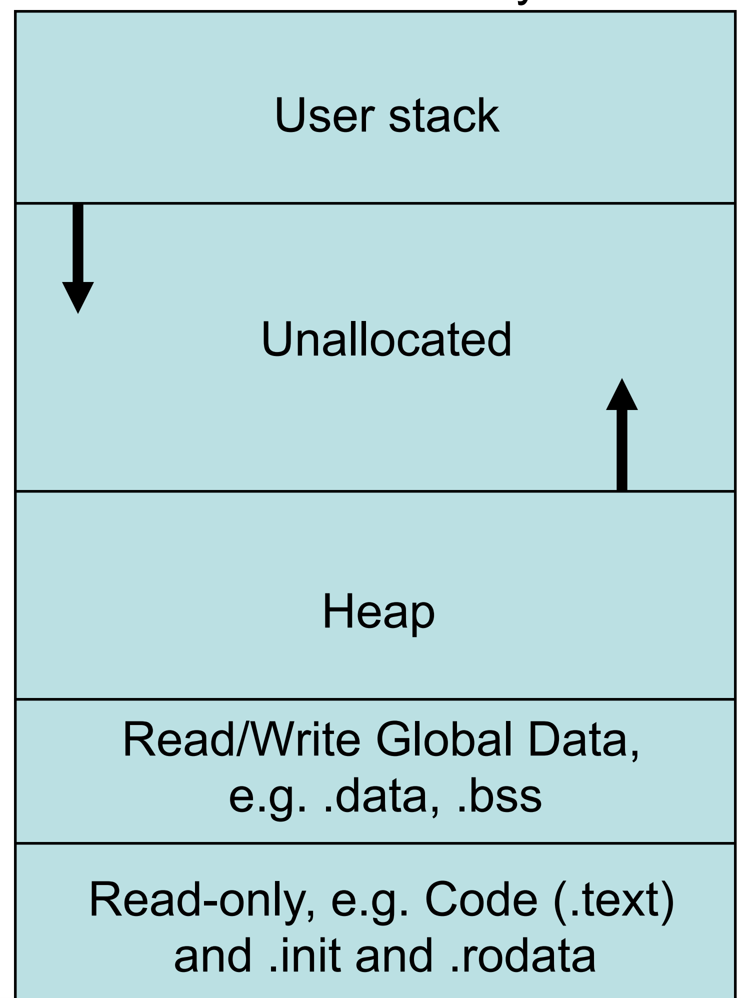
- When a program is loaded into RAM, it becomes an actively executing application
- The OS allocates a stack and heap to the app in addition to code and global data.
  - A call stack is for local variables
  - A heap is for dynamic variables, e.g. malloc()
  - Usually, stack grows downward from high memory, heap grows upward from low memory, but this is architecture-specific



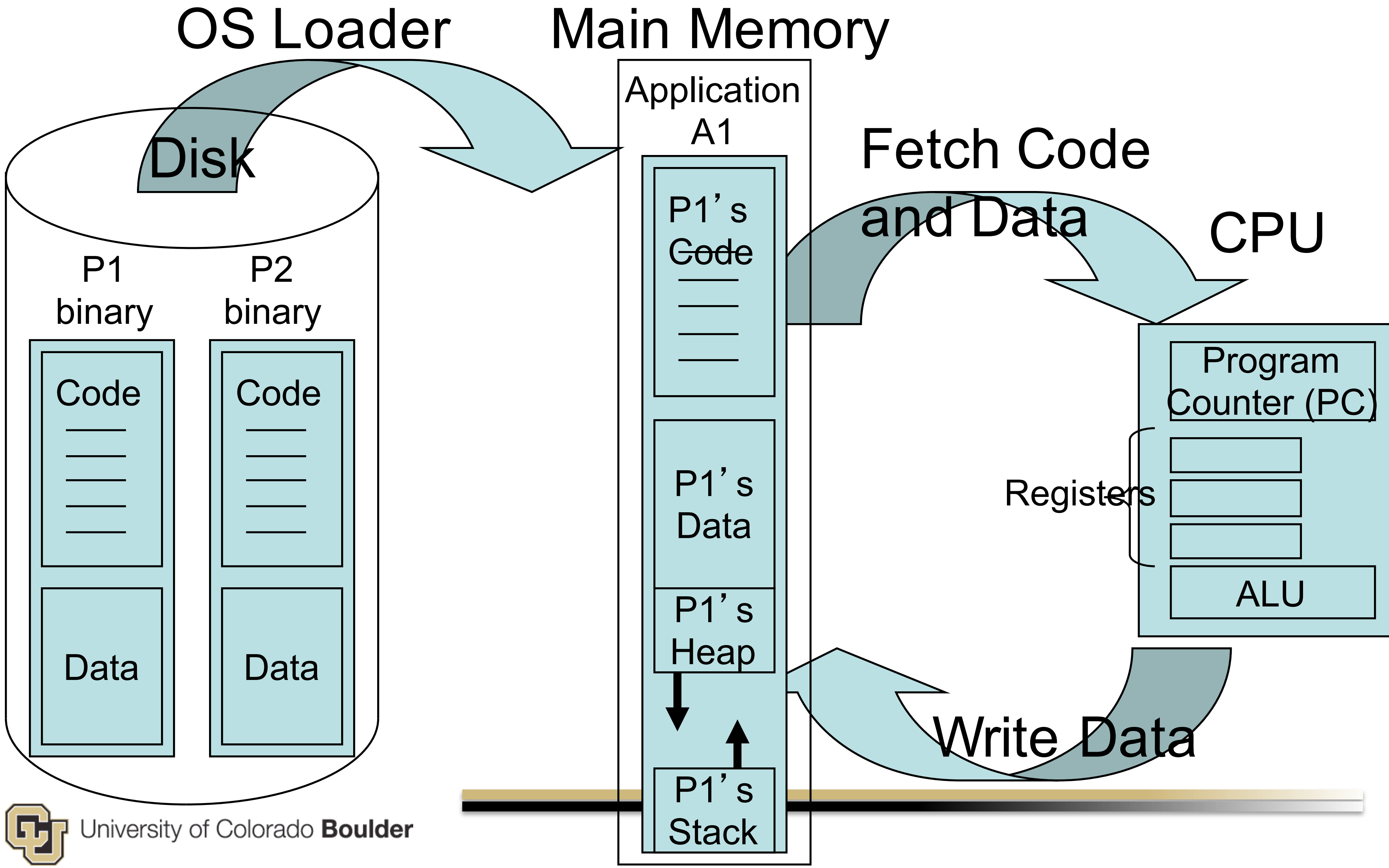
Running Executable Object Files

- Stack contains local variables
  - As main() calls function f1, we allocate f1's local variables on the stack
  - If f1 calls f2, we allocate f2's variables on the stack below f1's, thereby growing the stack, etc…
  - When f2 is done, we deallocate f2's local variables, popping them off the stack, and return to f1
- Stack dynamically expands and contracts as program runs and different levels of nested functions are called
- Heap contains run-time variables/buffers
  - Obtained from malloc()
  - Program should free() the malloc'ed memory
- Heap can also expand and contract during program execution

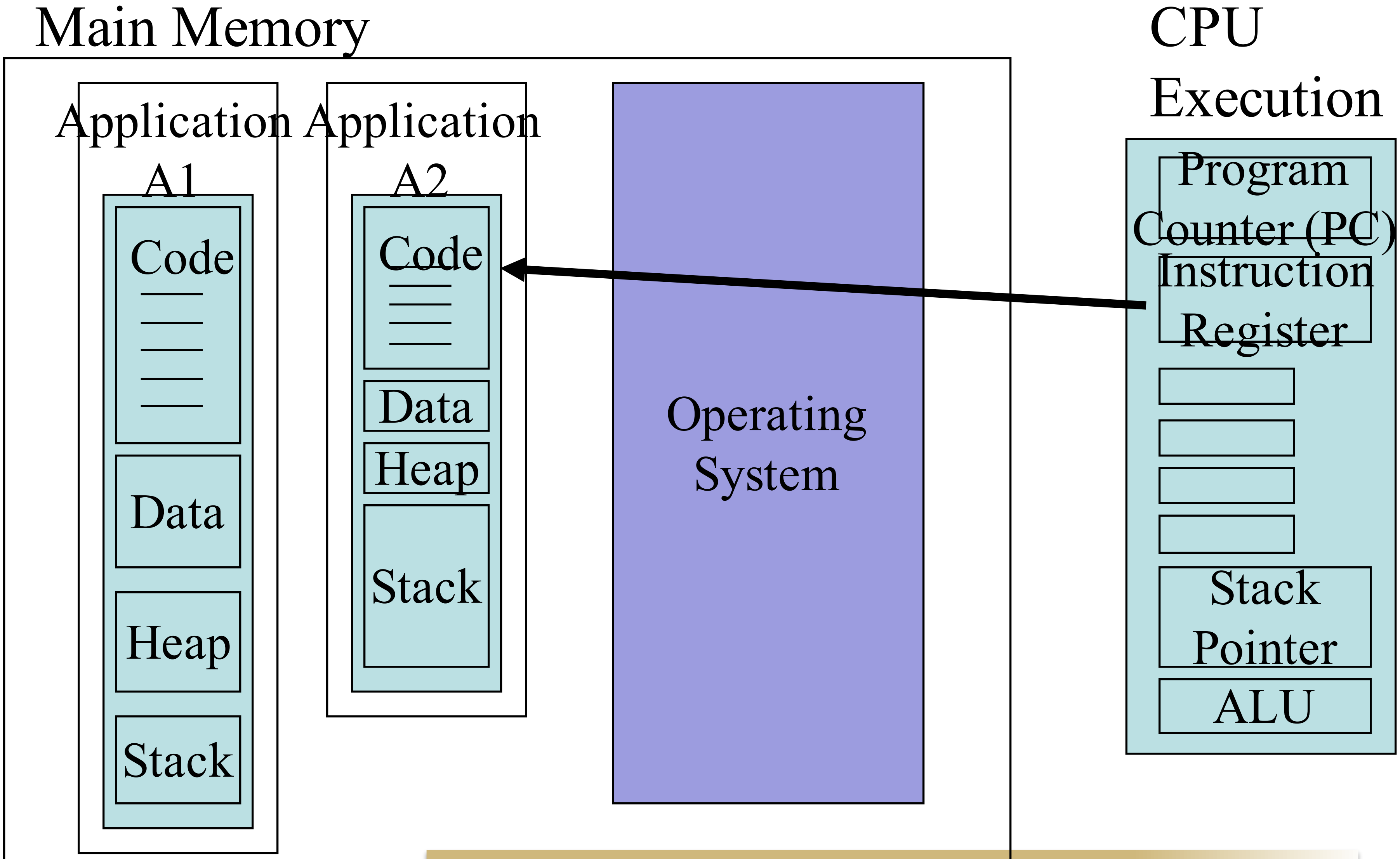
Run-time memory



Loading and Executing a Program – a more complete picture



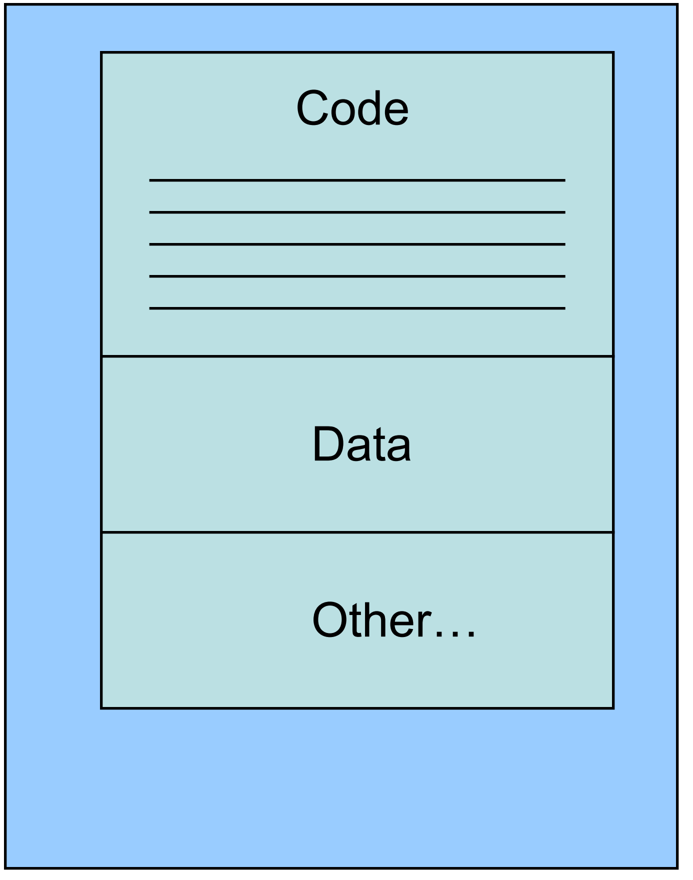
Multiple Applications + OS



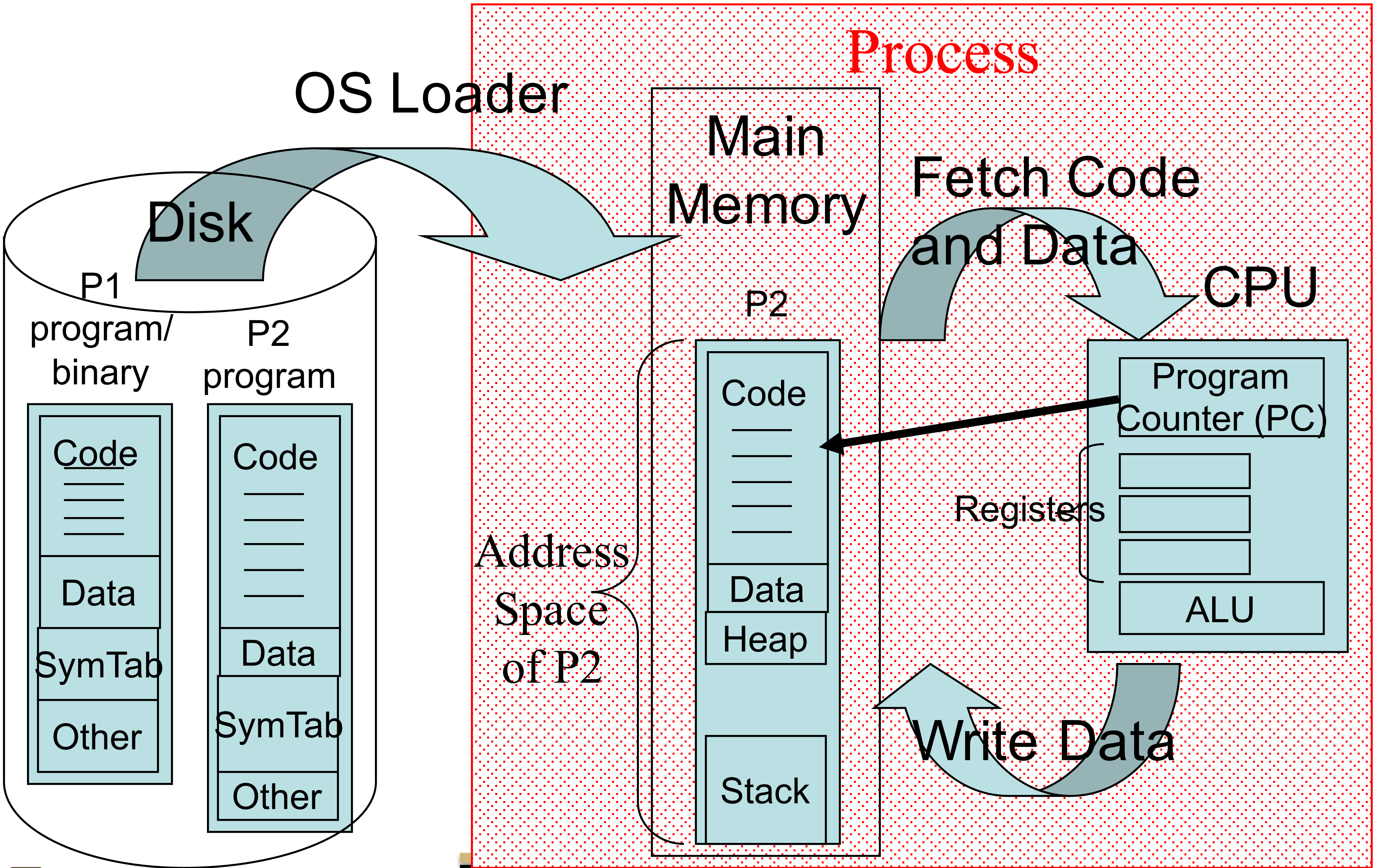
Chapter 3: What is a Process?

- A software *program* consist of a sequence of code instructions and data stored on disk
  - A program is a *passive* entity
- A *process* is a program *actively executing* from main memory within its *own address space*

Program P1

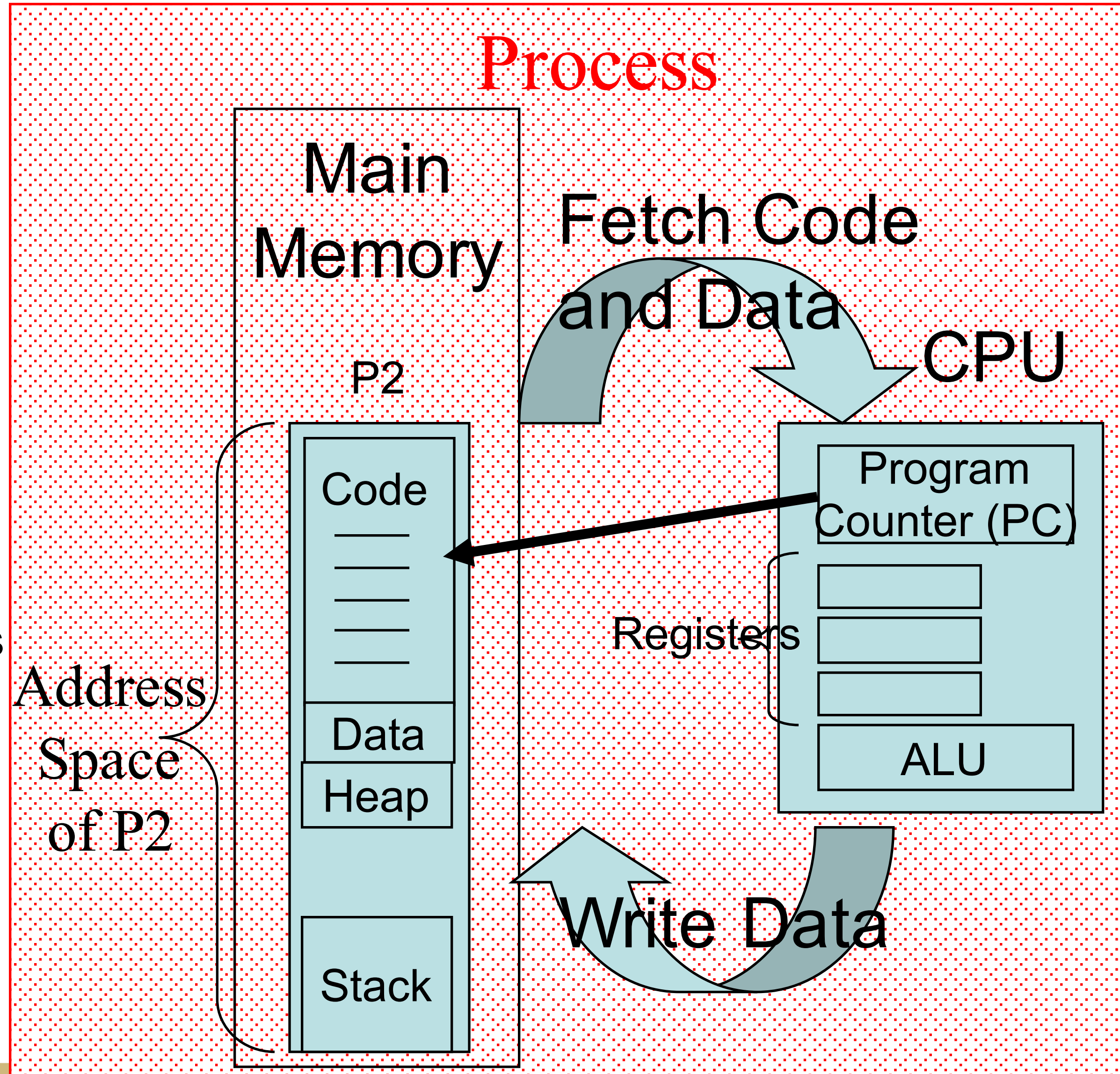


What Is a Process? (2)



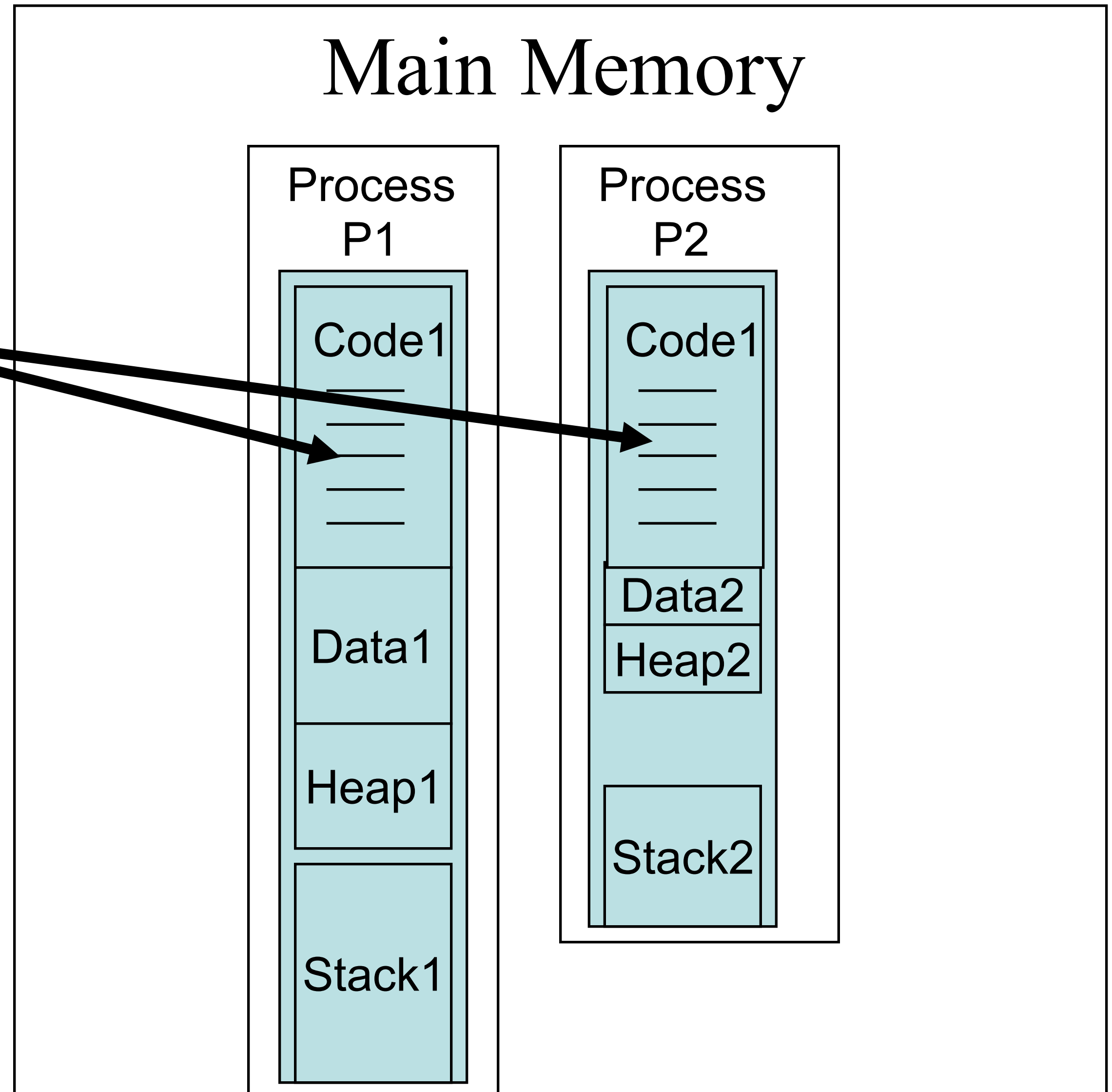
What is a Process? (3)

- A *process* is a program *actively executing* from main memory
  - has a Program Counter (PC) and execution state associated with it
    - CPU registers keep state
    - OS keeps process state in memory
    - it's alive!
  - Owns its own *address space*
    - a limited set of (virtual) addresses that can be accessed by the executing code



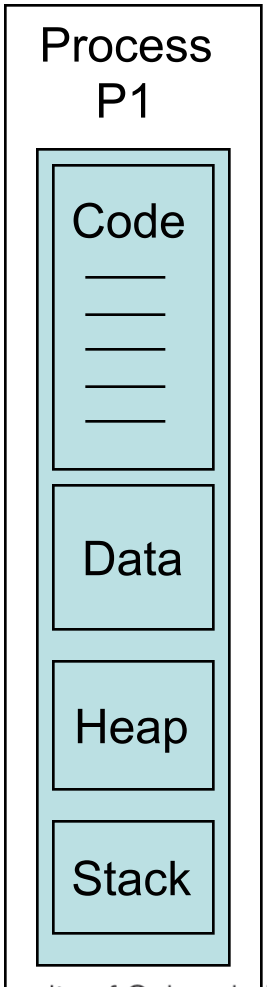
What is a Process? (4)

– 2 processes may execute the same program code, but they are considered *separate execution* sequences
  - e.g. two shell terminals



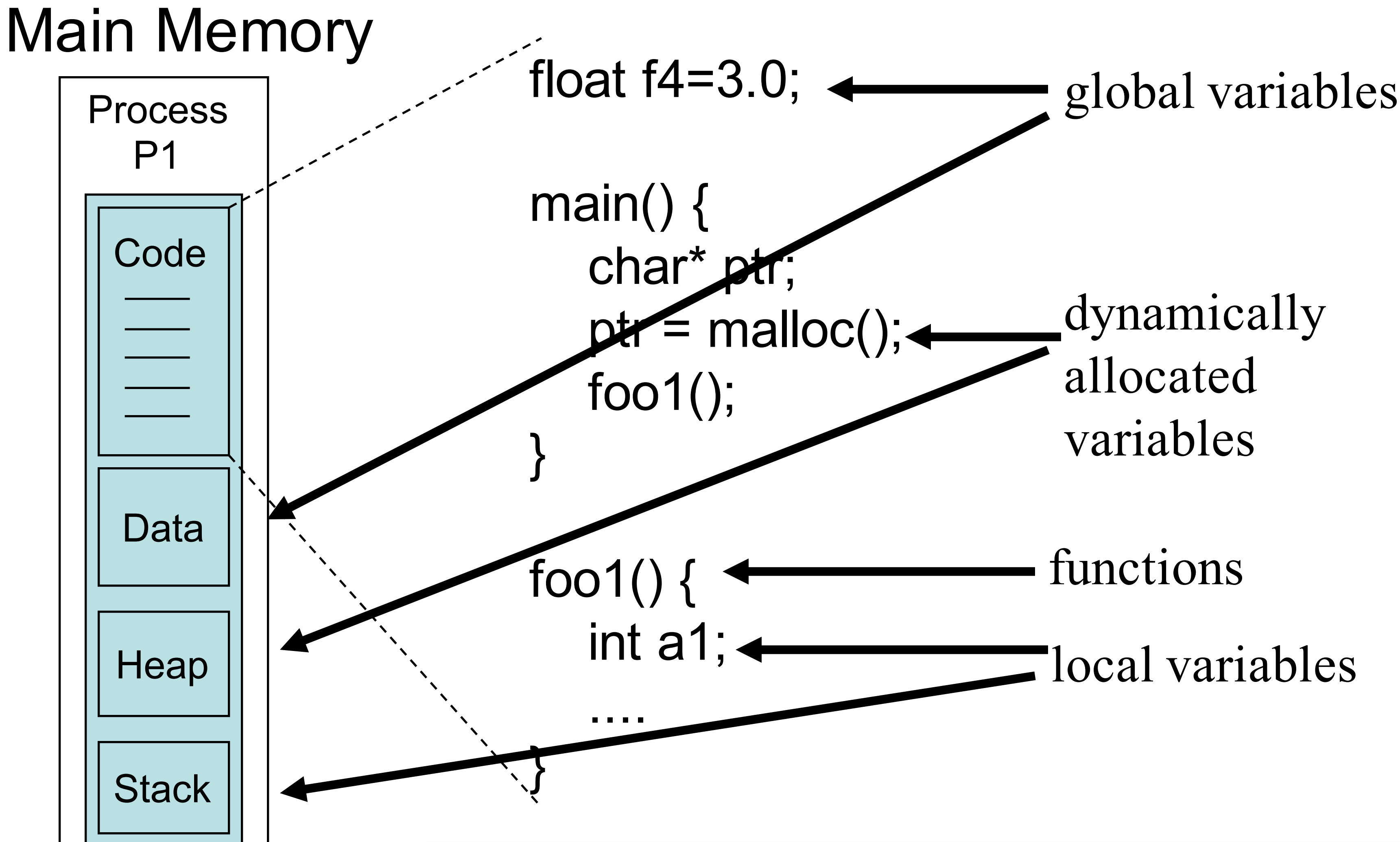
A Process Executes in its Own Address Space

Main Memory



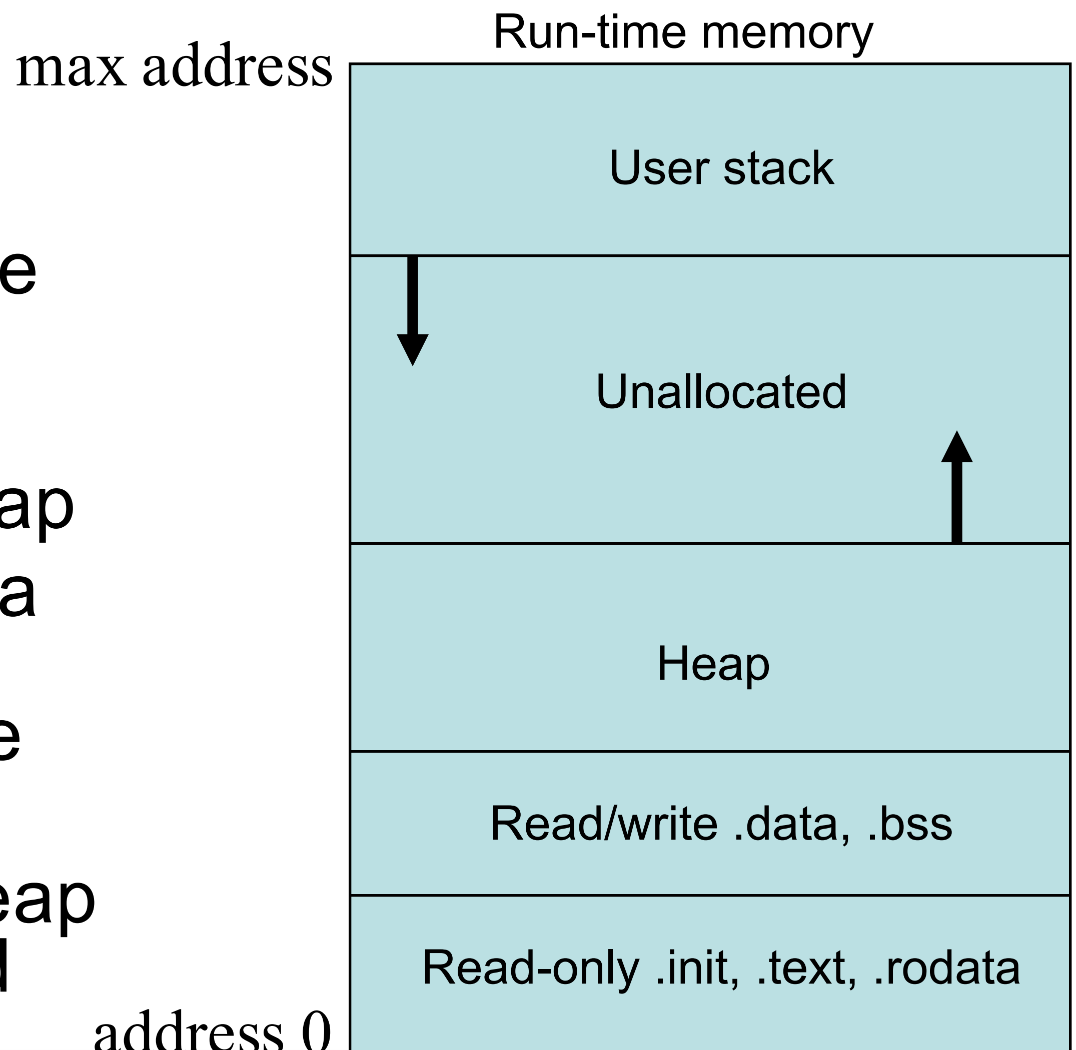
- OS tries to provide the illusion or *abstraction* to the process that it executes on its own abstract machine
  - in its own subset of RAM, i.e. its own address space – achieved using virtual memory paging
  - on its own subset (time slice) of the CPU – achieved by preemptive multitasking

How is a Process Structured in Memory?



How is a Process Structured in Memory?

- Run-time memory image
- Essentially code, data, stack, and heap
- Code and data loaded from executable file
- Stack grows downward, heap grows upward



Applications and Processes

- Application = $\sum_i$ Processes$_i$
  - e.g. a server could be split into multiple processes, each one dedicated to a specific task (UI, computation, communication, etc.)
  - The Application's various processes talk to each other using Inter-Process Communication (IPC). We'll see various forms of IPC later.