

CSCI 3753

Operating Systems

Processes

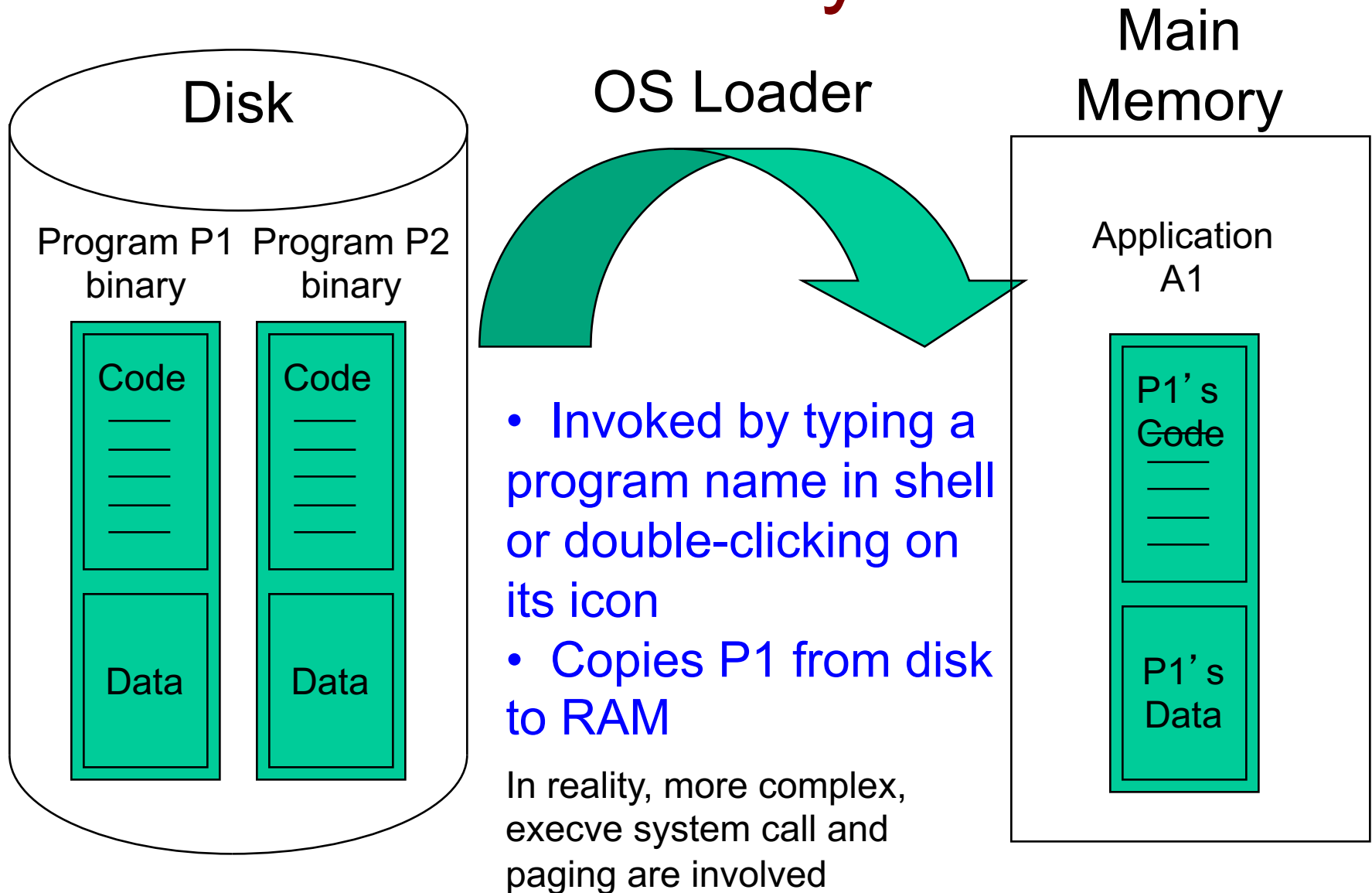
Lecture Notes By

Shivakant Mishra

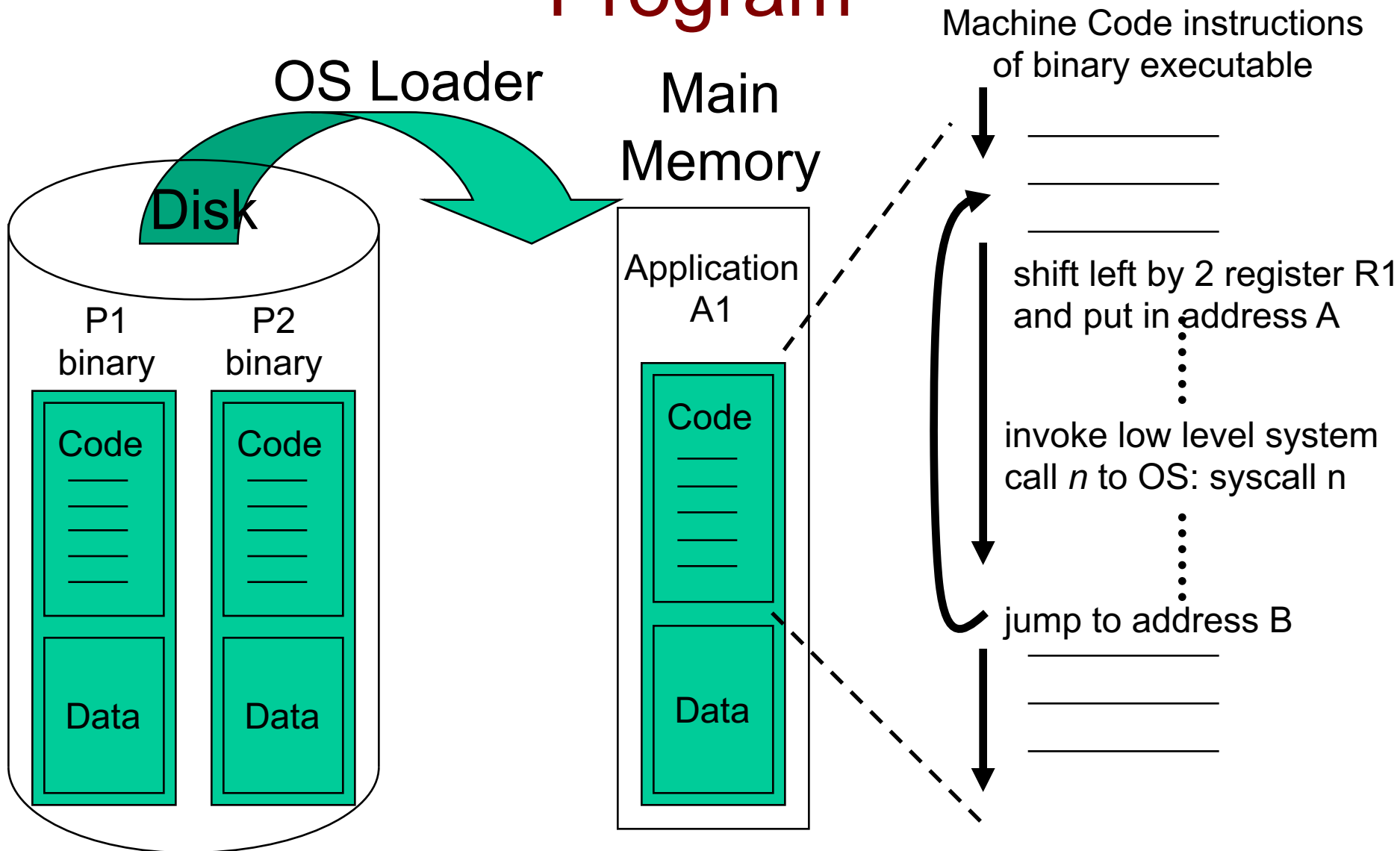
Computer Science, CU-Boulder

Last Update: 09/14/2017

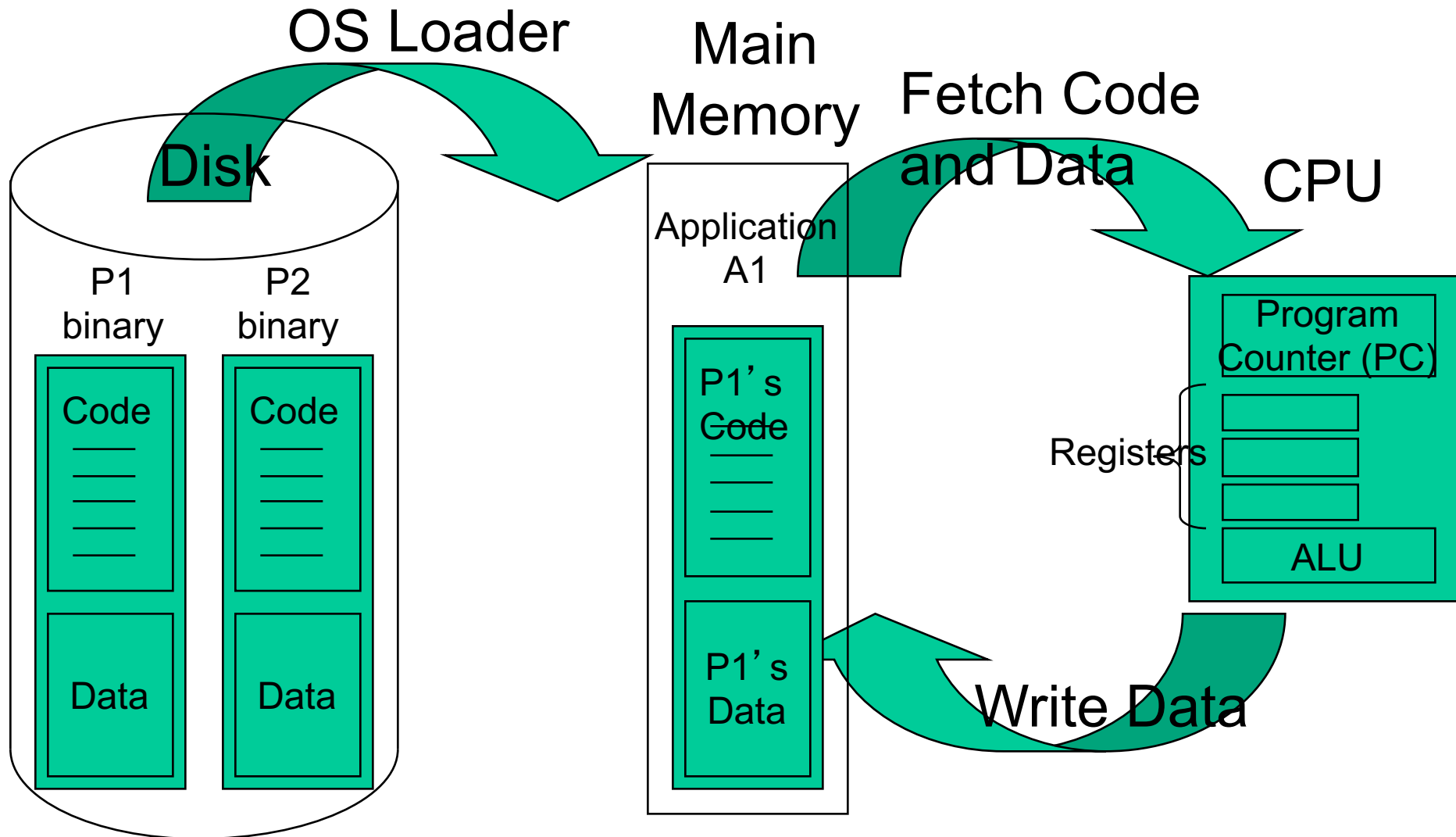
Loading a Program into Memory



Loading and Executing a Program

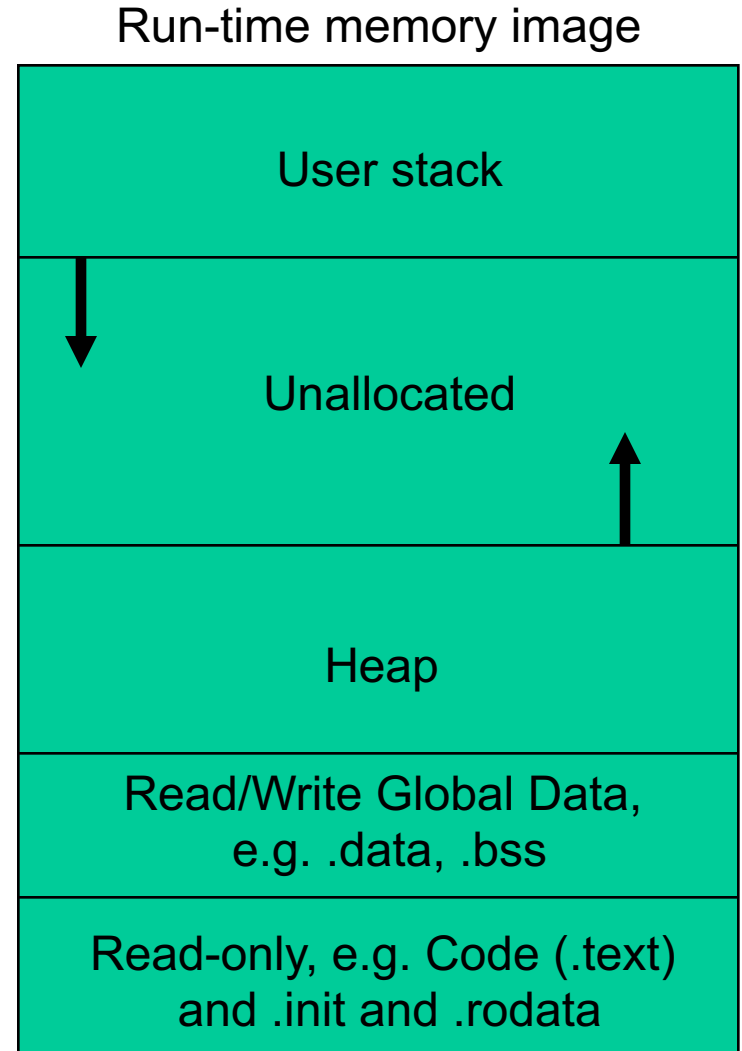


Loading and Executing a Program



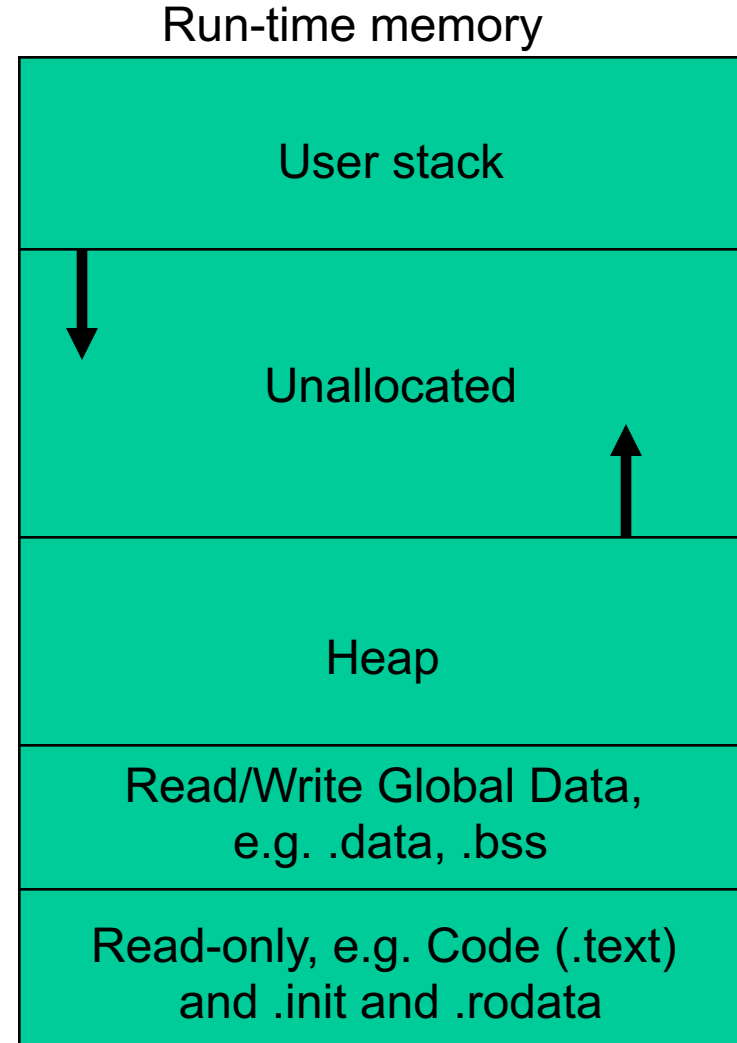
Loading Executable Object Files

- 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, function parameters and return addresses
 - A heap is for dynamic variables, e.g. *malloc()*, *new*

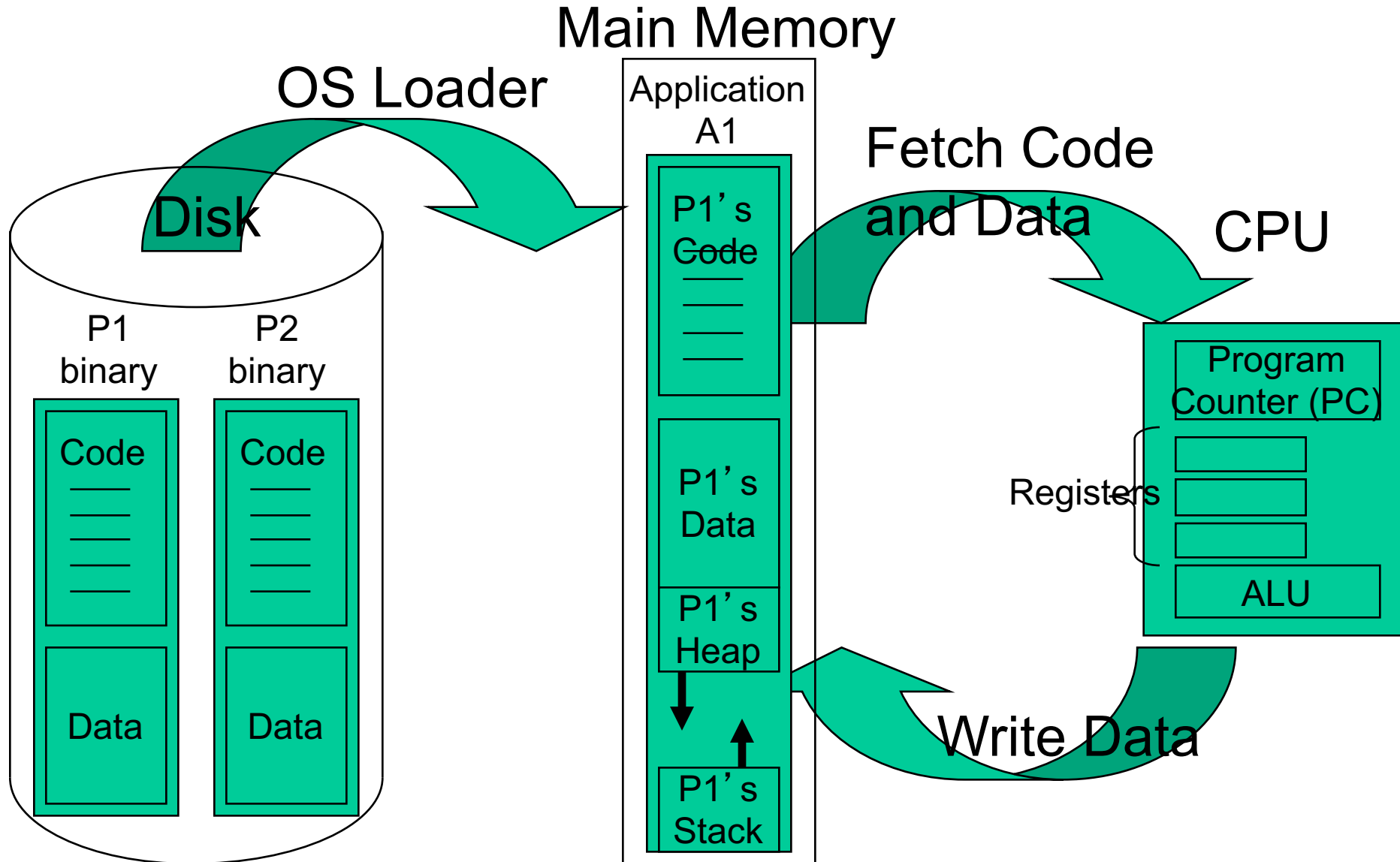


Running Executable Object Files

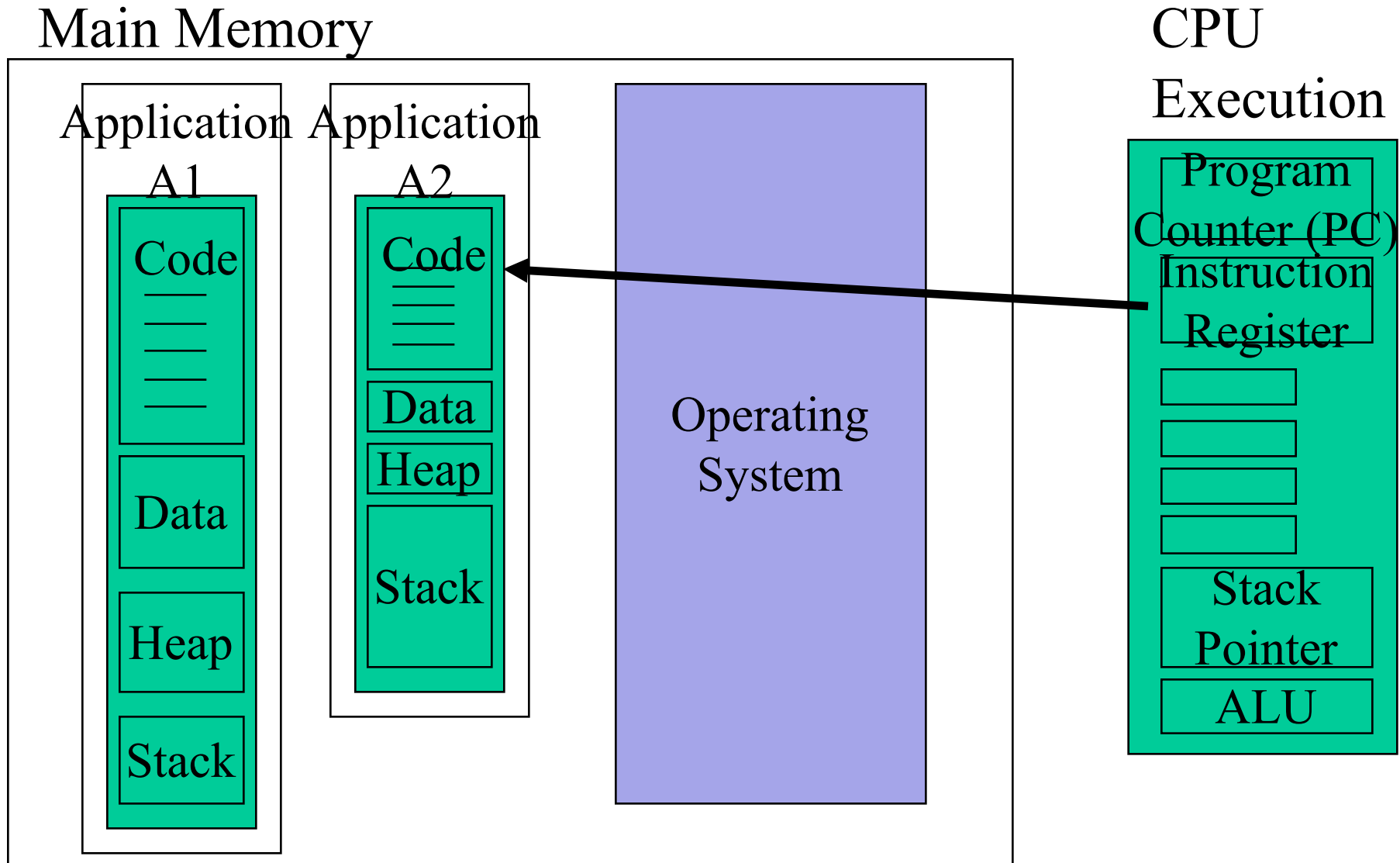
- Stack contains local variables
 - As `main()` calls function `f1`, we allocate `f1`'s local variables on the stack
 - If `f1` calls `f2`, we allocate `f2`'s variables on the stack below `f1`'s, thereby growing the stack, etc...
 - When `f2` is done, we deallocate `f2`'s local variables, popping them 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



Loading and Executing a Program – a more complete picture

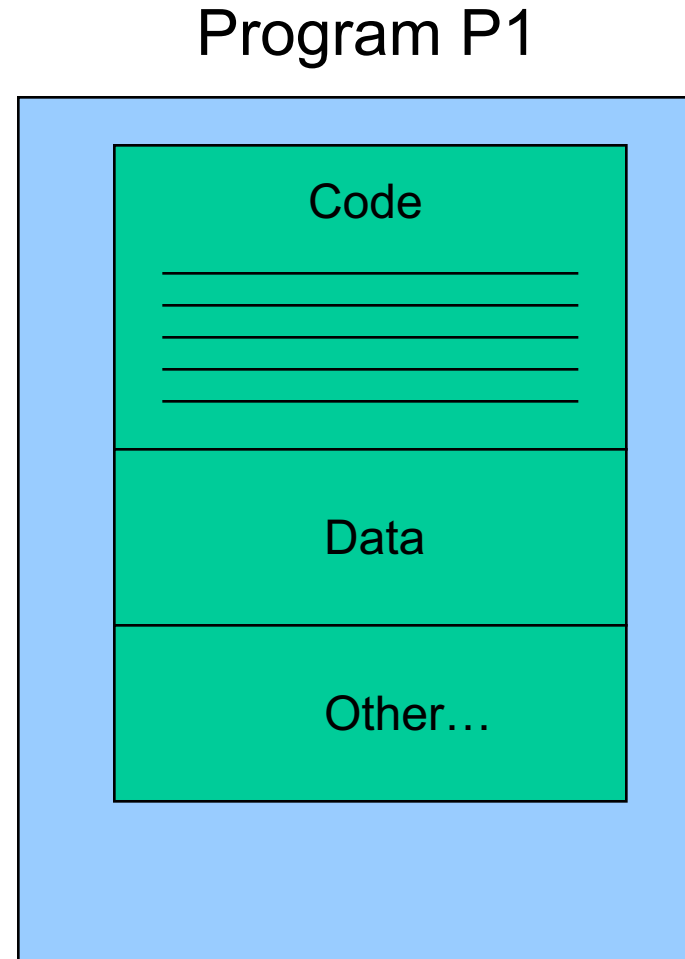


Multiple Applications + OS

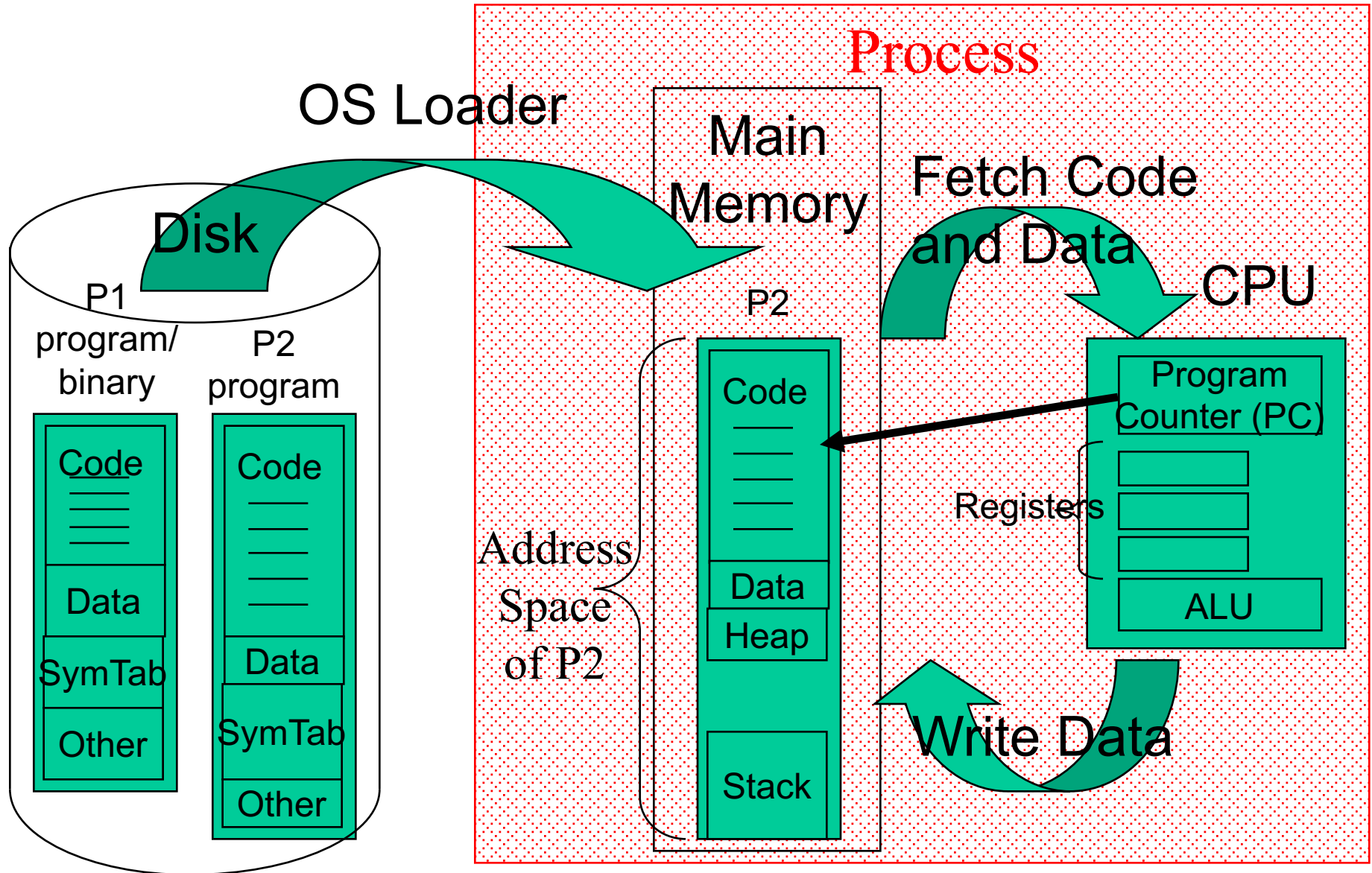


Chapter 3: What is a Process?

- 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

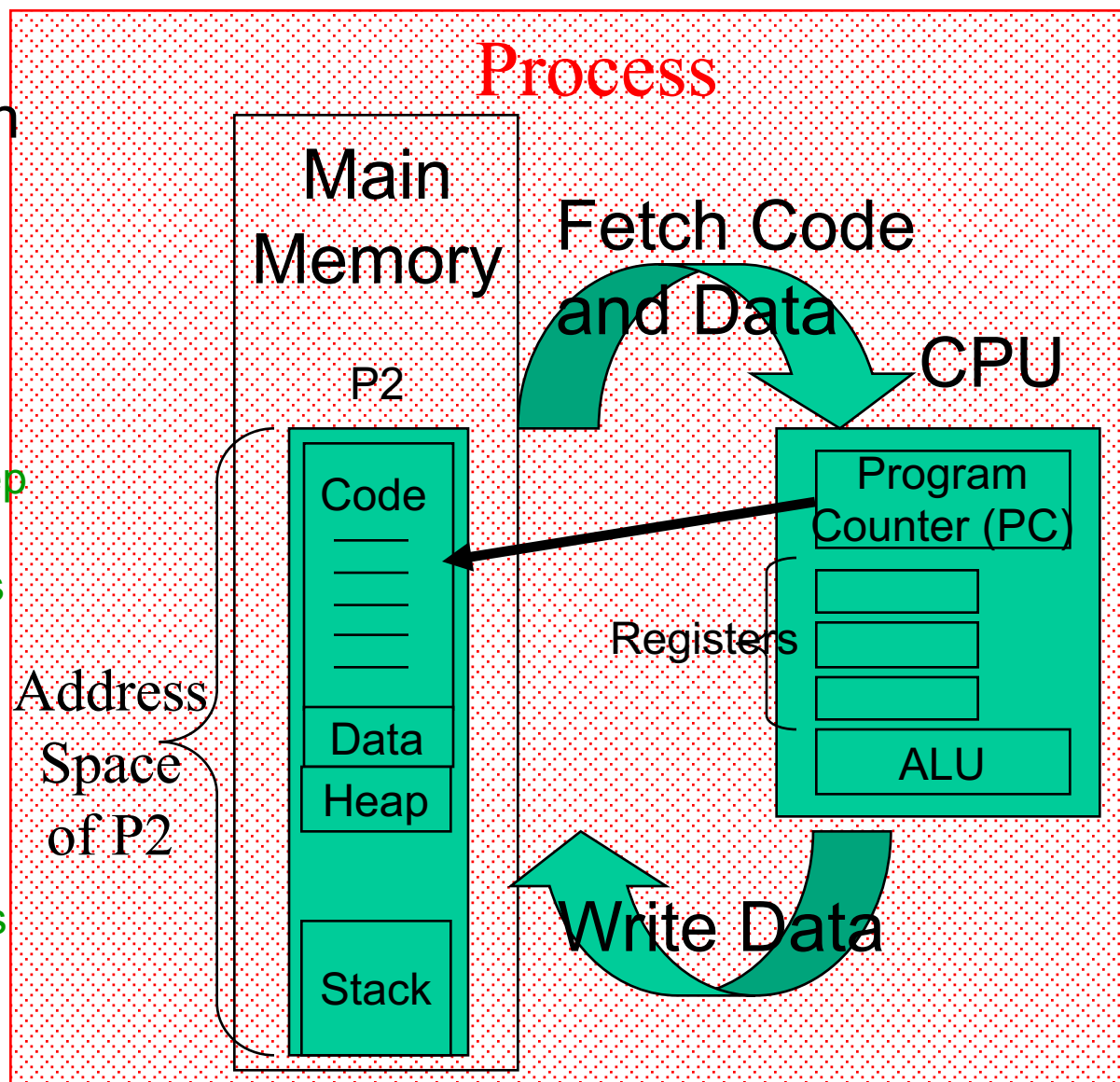


What Is a Process? (2)



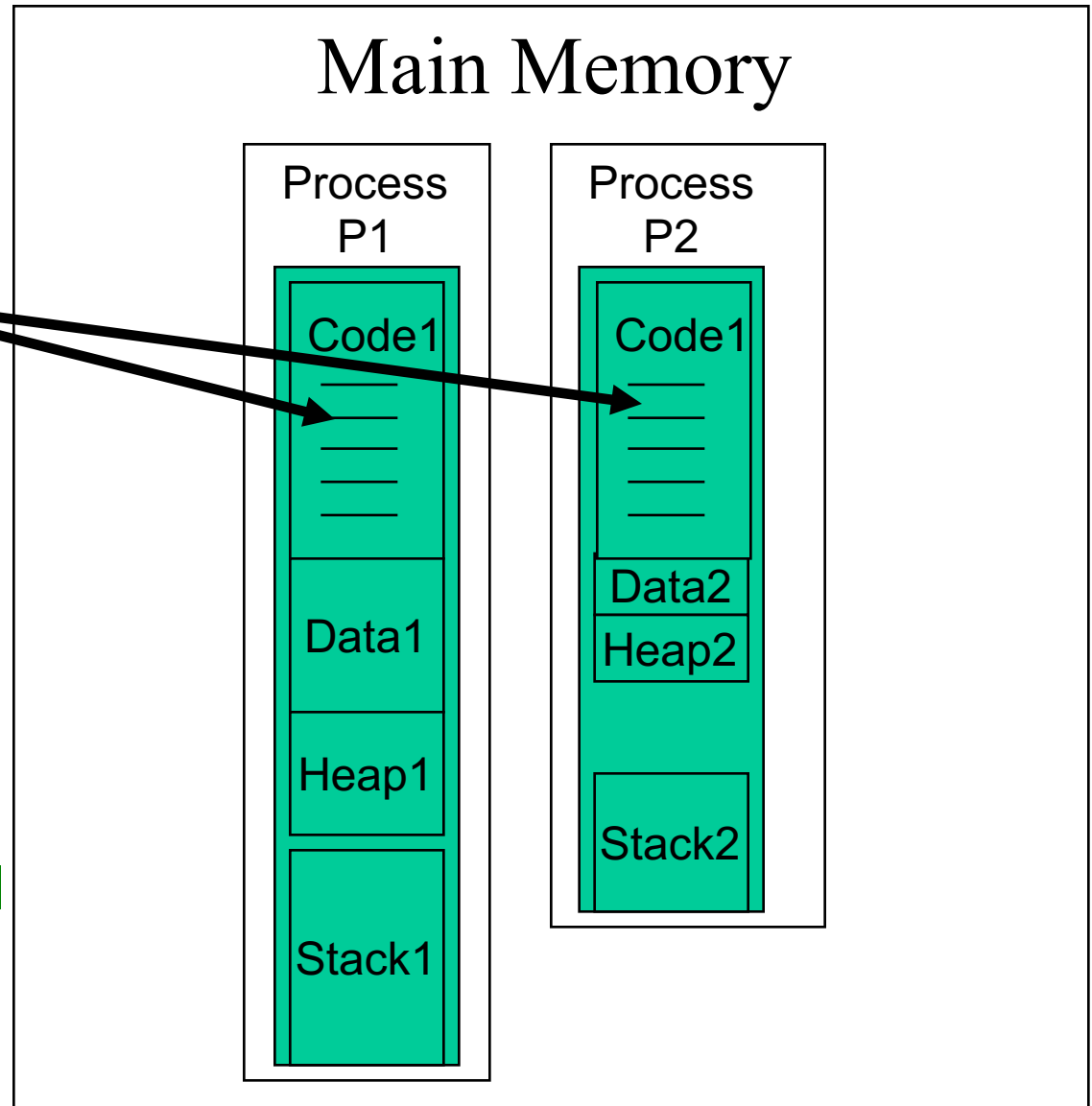
What is a Process? (3)

- 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



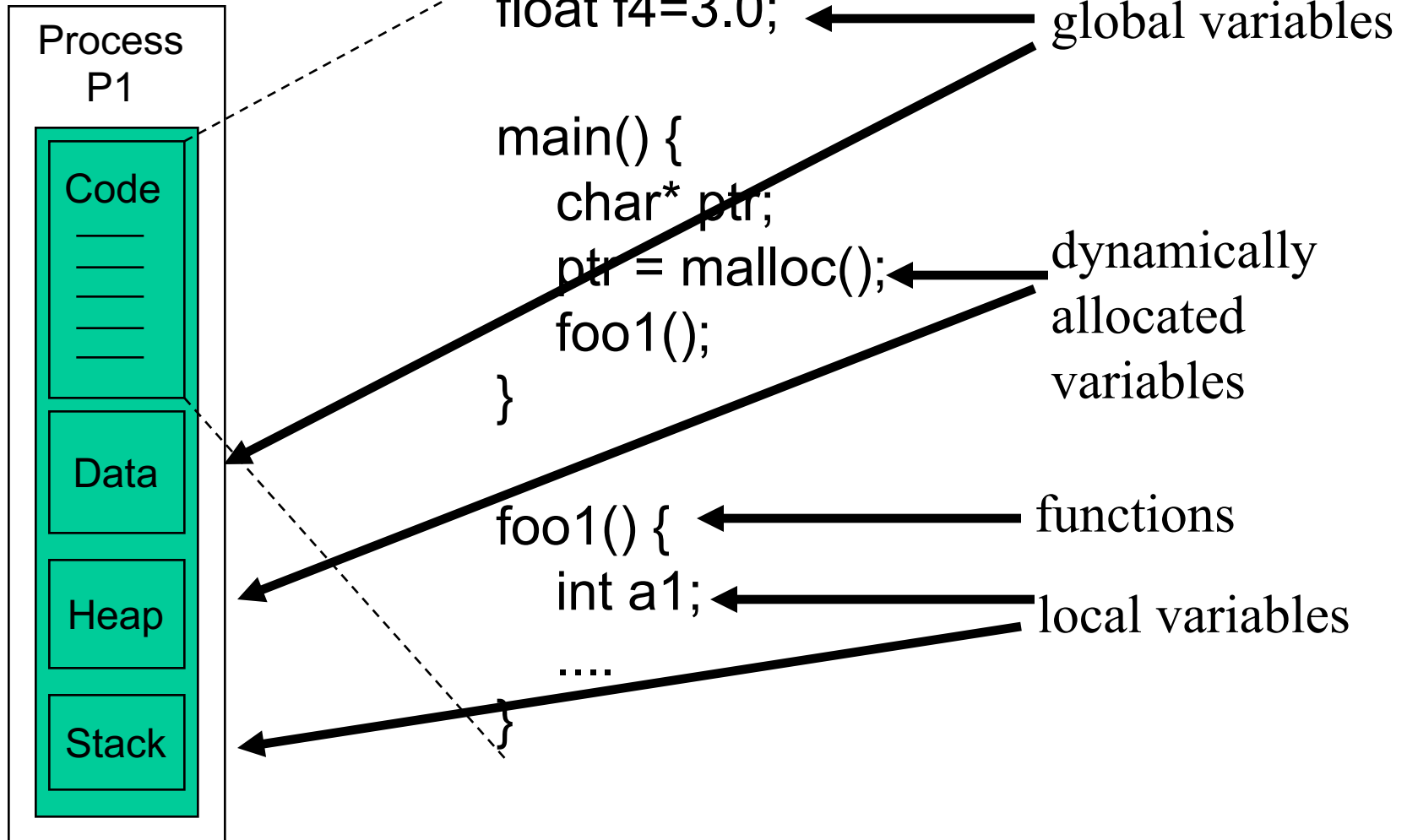
What is a Process? (4)

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



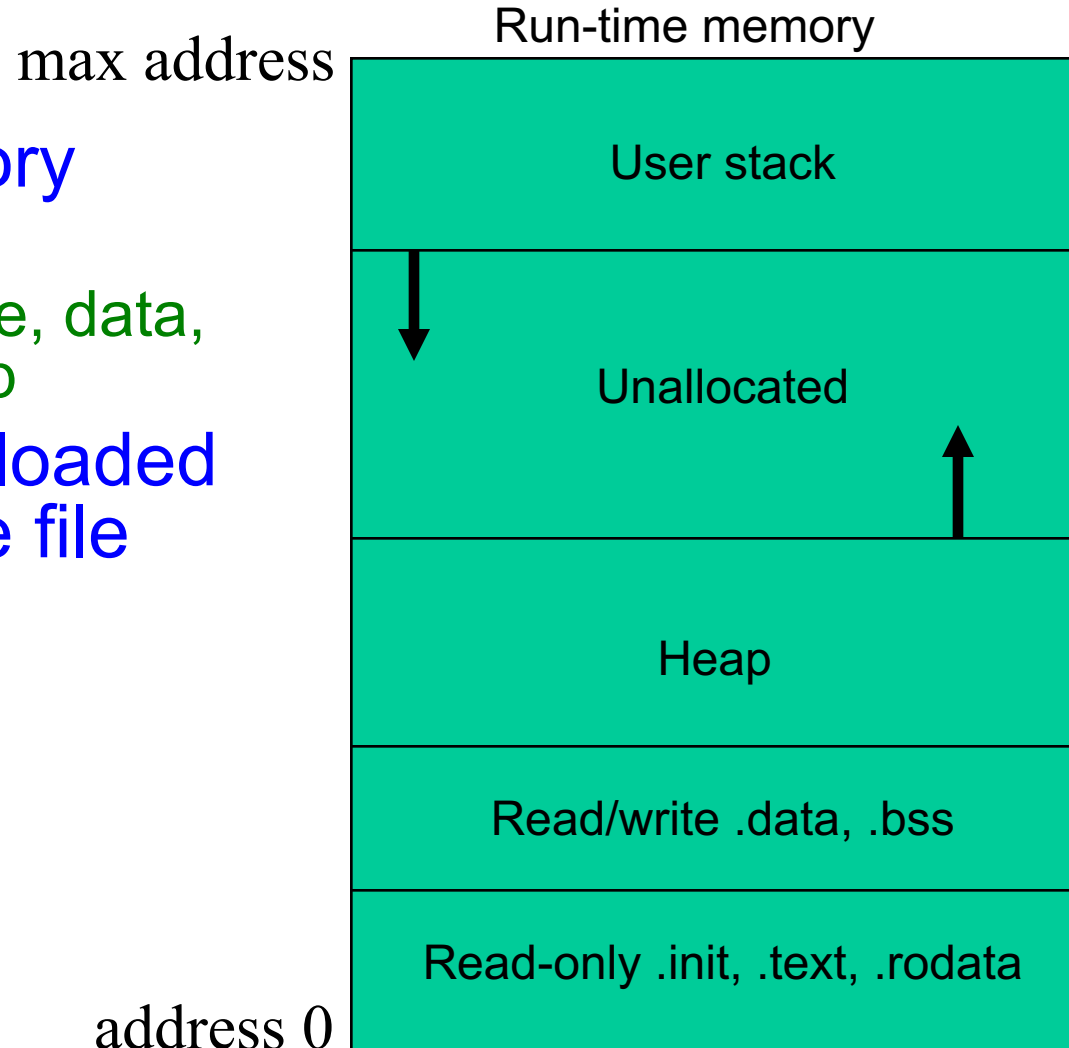
How is a Process Structured in Memory?

Main Memory



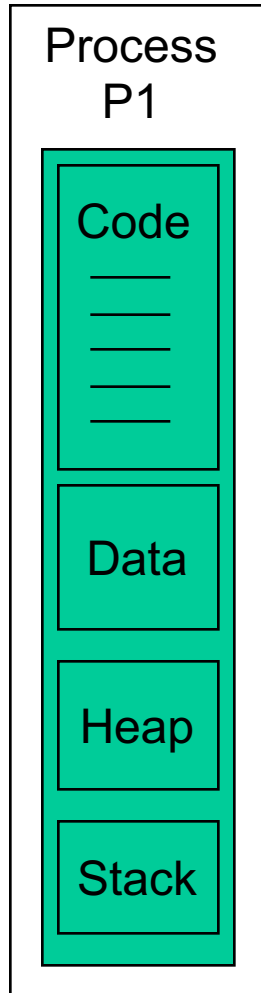
How is a Process Structured in Memory?

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



A Process Executes in its Own Address Space

Main Memory

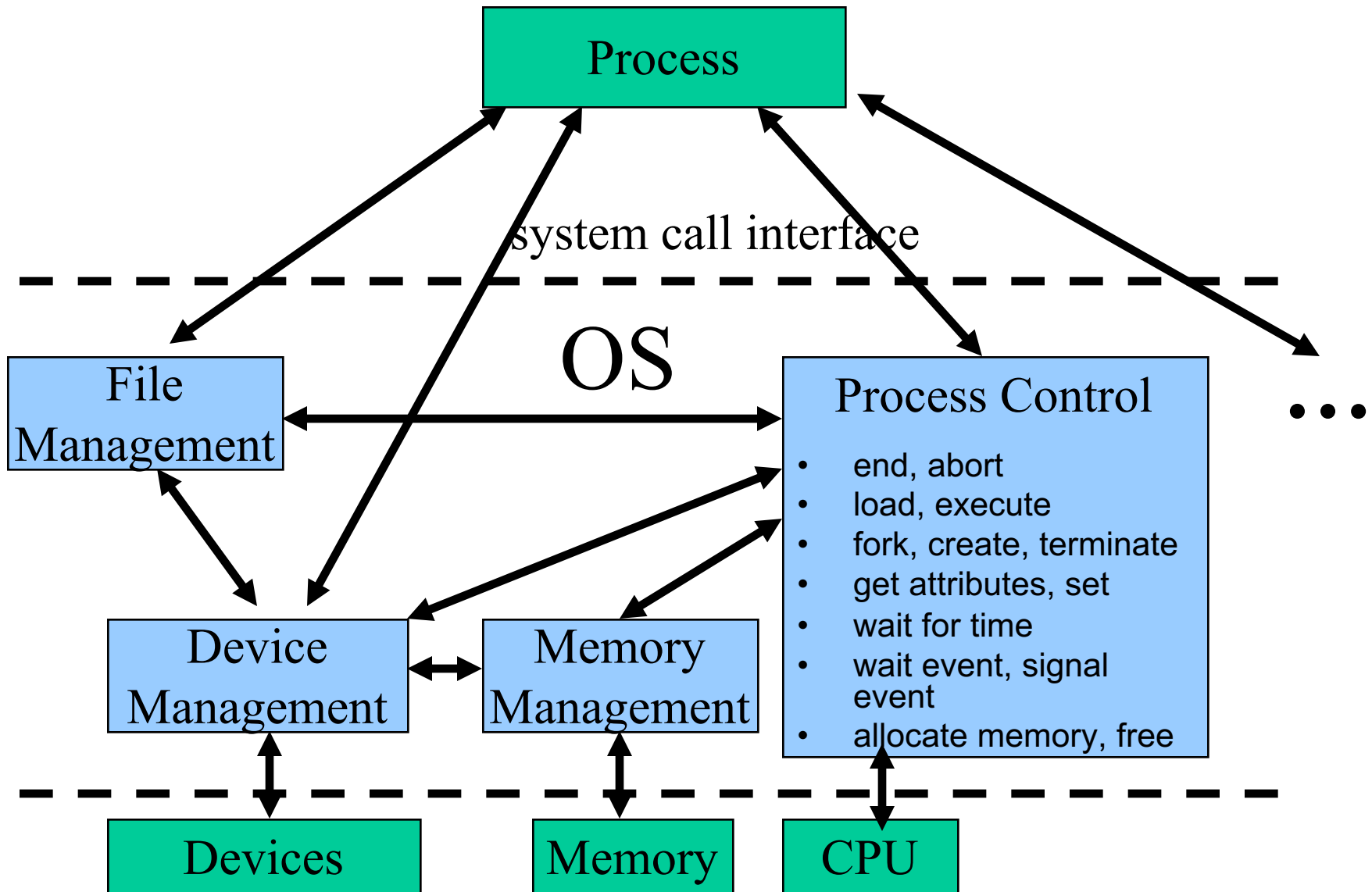


- OS tries to provide the illusion or *abstraction* to the process that it executes
 - in its own subset of RAM, i.e. its own address space
 - on its own subset (time slice) of the CPU

Applications and Processes

- An application may consist of multiple processes, each executing in its own address space
 - 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.

Process Management



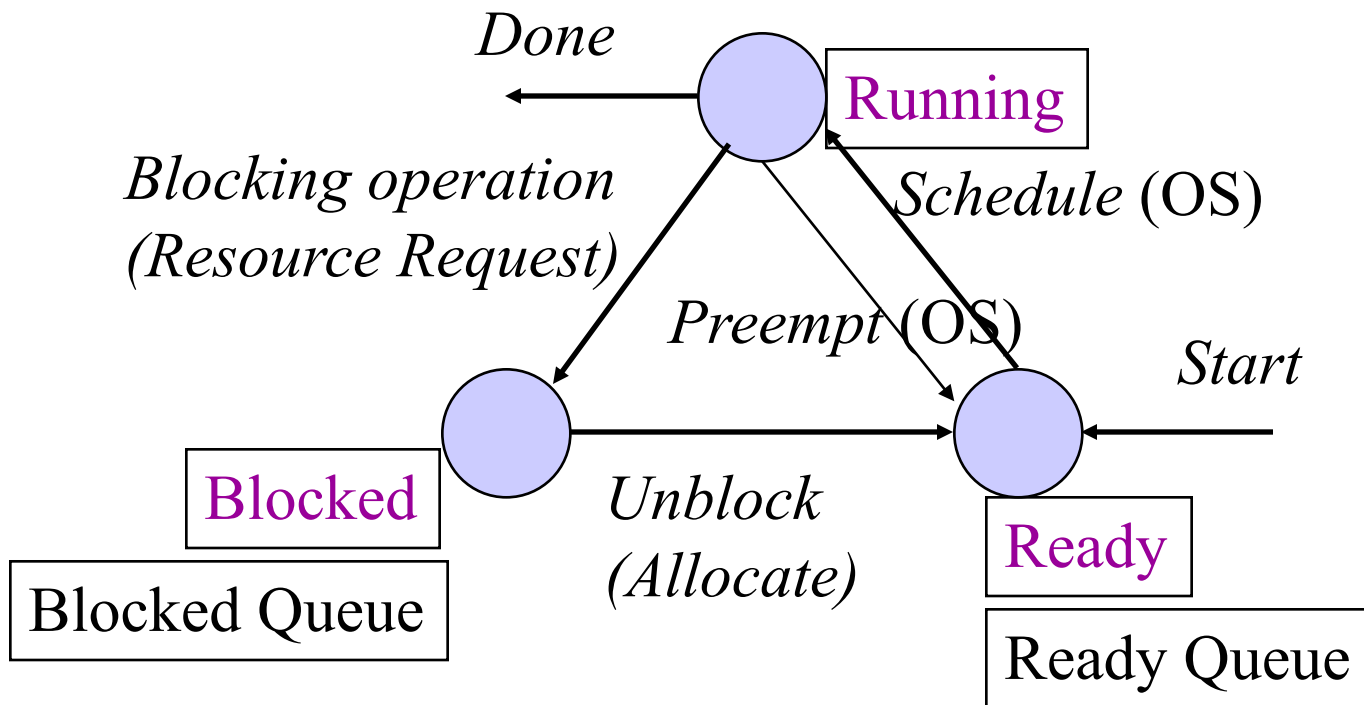
Process Manager

- Creation/deletion of processes (and threads)
- Synchronization of processes (and threads)
- Managing process state
 - Process state like PC, stack ptr, etc.
 - Resources like open files, etc.
 - Memory limits to enforce an address space
- Scheduling processes
- Monitoring processes
 - Deadlock, protection

State of a Process

- Memory image: Code, data, heap, stack
- Process state, e.g. ready, running, or waiting
- Accounting info, e.g. process ID
- Program counter value
- CPU registers' values
- CPU-scheduling info, e.g. priority
- Memory management info, e.g. base and limit registers, page tables
- I/O status info, e.g. list of open files

Process State Diagram

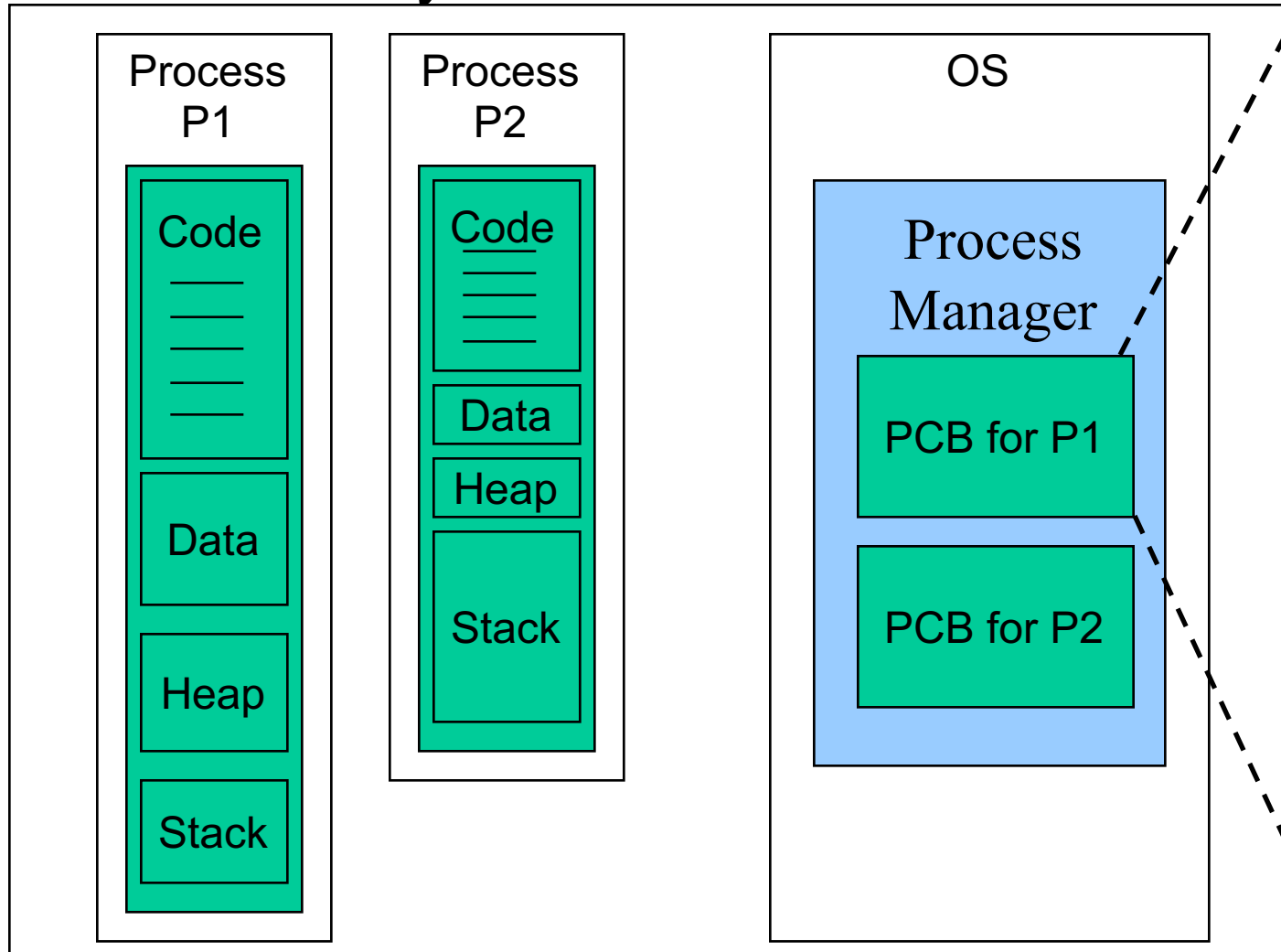


Process Control Block

- Each process is represented in OS by a process control block (PCB).
- PCB: Complete information of a process
- OS maintains a PCB table containing one entry for every process in the system.
- PCB table is typically of fixed size. This size determines the maximum number of processes an OS can have
 - The actual maximum may be less due to other resource constraints, e.g. memory.

Process Control Block (PCB)

Main Memory



- Process state, e.g. ready, running, or waiting
- accounting info, e.g. process ID
- Program Counter
- CPU registers
- CPU-scheduling info, e.g. priority
- Memory management info, e.g. base and limit registers, page tables
- I/O status info, e.g. list of open files

A PCB is also called a Process Descriptor

Creating Processes

- In Windows, there is a `CreateProcess()` call
 - Pass an argument to *CreateProcess()* indicating which program to start running
 - Invokes a system call to OS that then invokes process manager to:
 - allocate space in memory for the process
 - Set up PCB state for process, assigns PID, etc.
 - Copy code of program name from hard disk to main memory, sets PC to entry point in *main()*
 - Schedule the process for execution
 - As we will see, this combines UNIX's *fork()* and *exec()* system calls and achieves the same effect

Creating Processes in UNIX/Linux

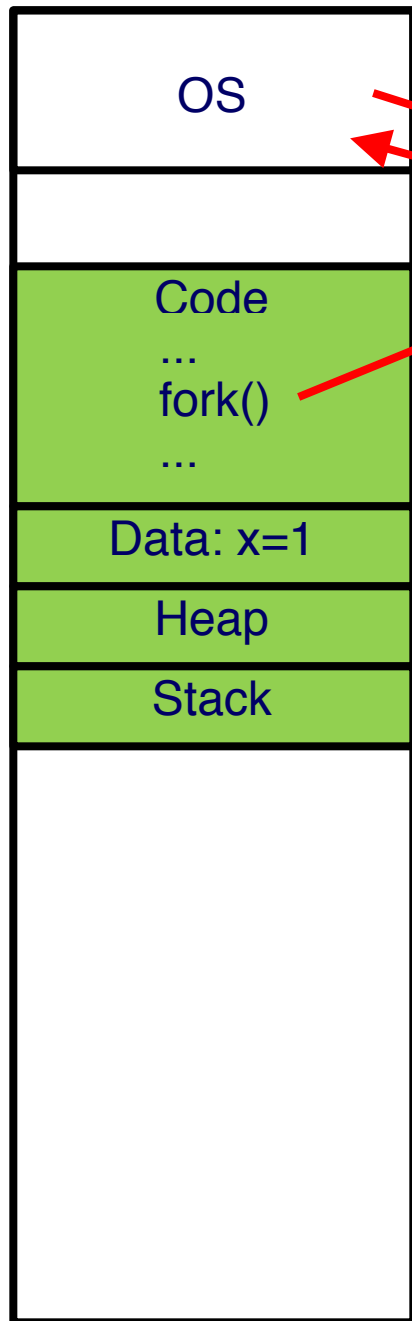
- Use *fork()* command to create/spawn new processes from within a process
 - When a process (called parent process) calls *fork()*, a new process (called child process) is created
 - Child process is an exact copy of the parent process
 - All addresses are appropriately mapped – We'll see this later under memory management
 - The child starts executing at the same point as the parent, namely just after returning from the *fork()* call

fork ()

- The `fork()` call returns an *int* value
 - In the parent process, returned value is child's PID
 - In the child, returned value is 0
 - Since both parent and child execute the same code starting from the same place, i.e. just after the `fork()`, then to differentiate the child's behavior from the parent's, you could add code:

```
PID = fork();  
if (PID==0) { /* child */  
    codeforthechild();  
    exit(0);  
}  
/* parent's code here */
```

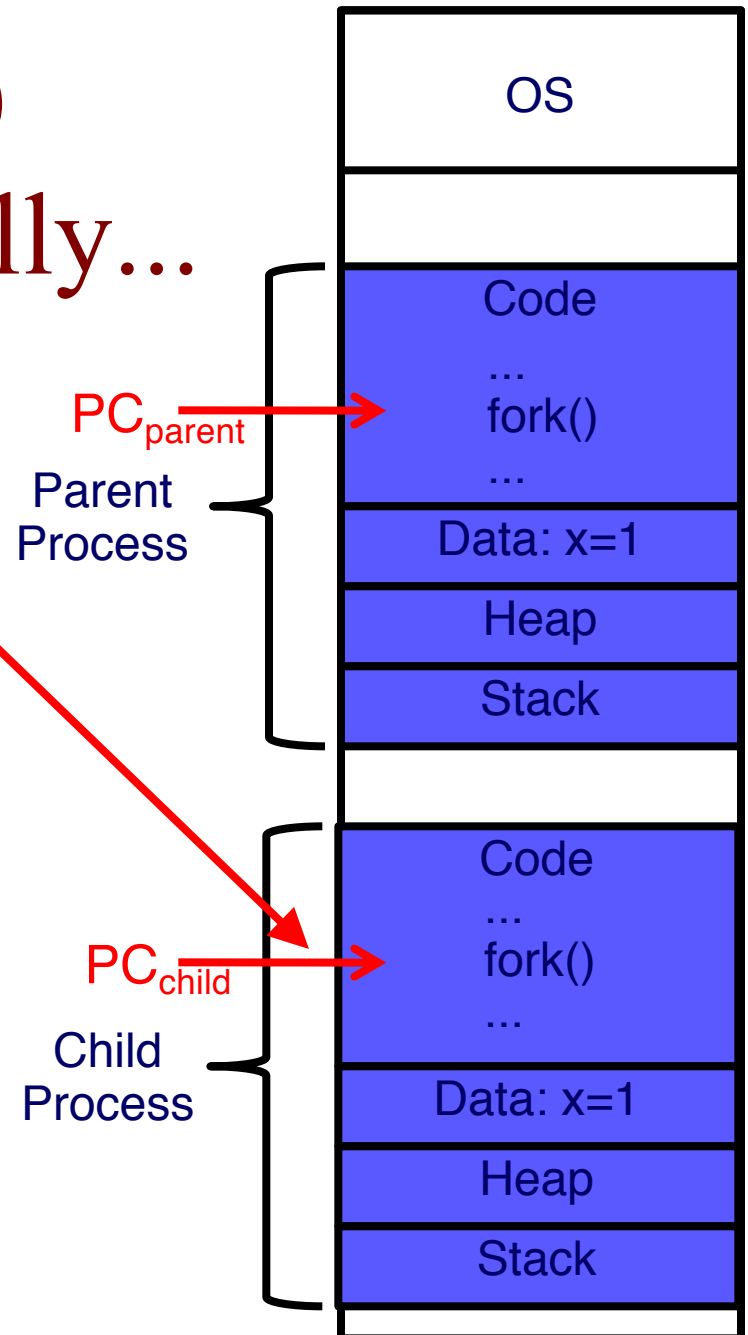
Memory (before fork)



Fork()
conceptually...

- Fork() duplicates address space of parent in the child
- Both execute concurrently

Memory (after fork)



Loading Processes

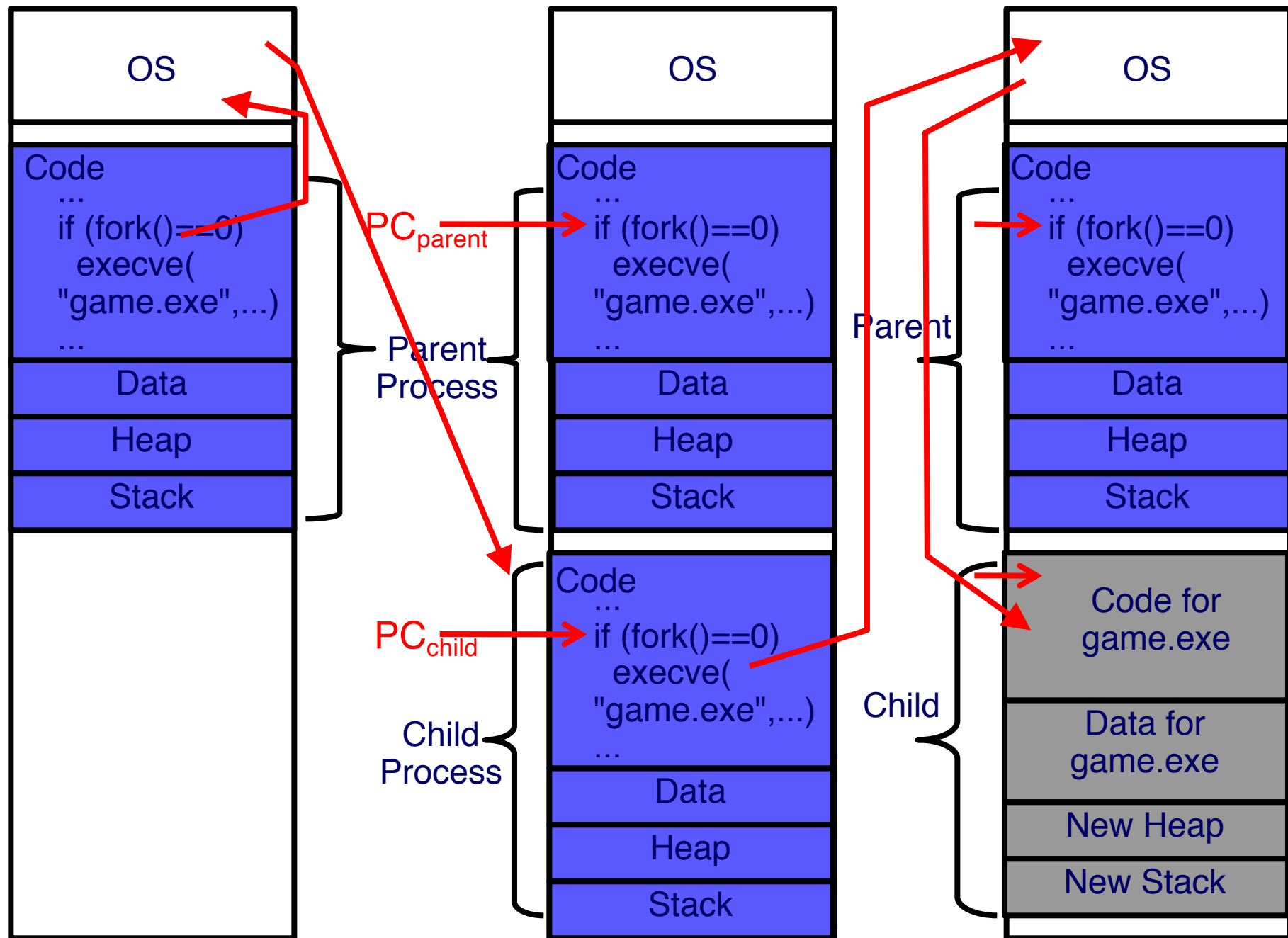
- The `exec()` system call loads program code into the calling process's memory (same address space!), clears the stack, and begins executing the new code at its main entry point
 - The calling code is erased!
 - Use *fork()* and *exec()* (actually *execve()*) to create a new process executing a new program in a new address space

```
PID = fork();
if (PID==0) { /* child */
    exec("/bin/ls");
    exit(0);
}
/* the parent's code here */
```

Memory (before fork)

Memory (after fork)

Memory(after execve)



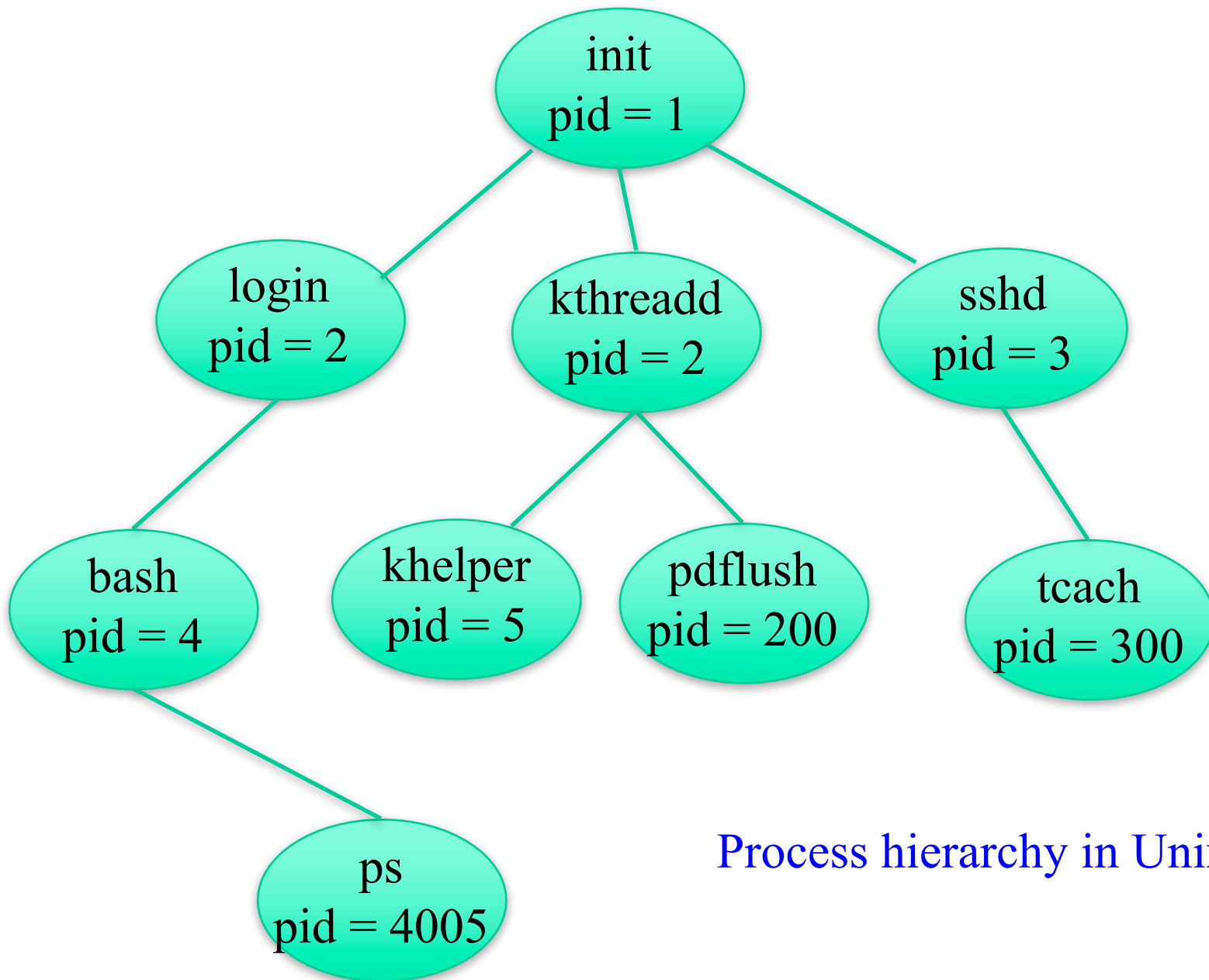
More on Processes

- Copying the entire code of a parent into a child can be expensive on a *fork()*, so
 - at start, child can share parent's code pages. Only create a copy on a write.
- The *wait()* system call is used by a parent process to be informed of when a child has completed, i.e. called *exit()*
 - Once the parent has called *wait()*, the child's PCB and address space can be freed
- There is also *waitpid()* to wait on a particular child process to finish

More about this in recitation

Process Hierarchy

- OS creates a single process at the start up
- An existing process can spawn one or more new processes during execution
 - Parent-child relationship
 - A parent process may have some control over its child process(es): suspend/activate execution; wait for termination; etc.
- A tree-structured hierarchy of processes



Process hierarchy in Unix

Accessing Process State

- One way is through standard system calls
- Another way is through the proc file system
 - Linux exports process status through /proc
 - Each process is listed by its process ID in the /proc directory
 - To inspect a given variable of a process, look up its corresponding file name
 - e.g. /proc/processID/stat gives the process' status

Announcements

- Midterm exam
 - Thursday, 10/26 in class
 - Closed-book, closed-notes
 - Topics will be announced in the 2nd week of Oct
- Friday recitation
 - Recitation material posted on Moodle
 - there will a quiz
- Readings: Chapters 1, 2, 13 and 3

Recap ...

- Process: actively executing program from memory
 - State: PC, CPU registers, stack, heap, ...
 - Own address space
- Process state diagram: Ready, Running, Blocked
- PCB and PCB table
- Process creation
 - Windows: `CreateProcess()`
 - Linux: `fork()` and `exec()`
- An application may consist of multiple processes

Recap ...

```
PID = fork();  
if (PID==0) { /* child */  
    exec("/bin/ls");  
    exit(0);  
}  
/* the parent's code here */
```

- Process hierarchy: a tree-structured hierarchy of processes

Context Switch: more details

- Linux allocates two stacks for each process: a user stack that resides in the user address space and a kernel stack that resides in the kernel
 - Kernel stack is used when the process is executing in the kernel (supervisor mode)
 - Kernel stack is needed for security purposes
 - OS allocates 8 KB for each kernel stack
 - Process's PCB is actually stored at one end of this space and the (kernel) stack starts from the other end
- Linux OS provides *schedule()*, which is invoked by the timer interrupt to schedule a new process on the CPU
- The *schedule()* function calls another function *switch_to()*, which does the actual context switching

Schedule() and switch_to() function

- Here is an outline of schedule and switch_to functions
(*actual implementations are in assembly code*)

```
schedule( )
```

```
{
```

```
    disable_interrupts;
```

```
    prev_proc = process id of running process
```

```
    next_proc = process id of the next process to run
```

```
    update ready_queue, system_queue, etc.
```

```
    switch_to (prev_proc, next_proc);
```

```
    running_process = prev_proc;
```

```
    enable_interrupts;
```

```
}
```

```
switch_to (prev_proc, next_proc)
```

```
{
```

```
    save the state of prev_proc in prev_proc's kernel stack
```

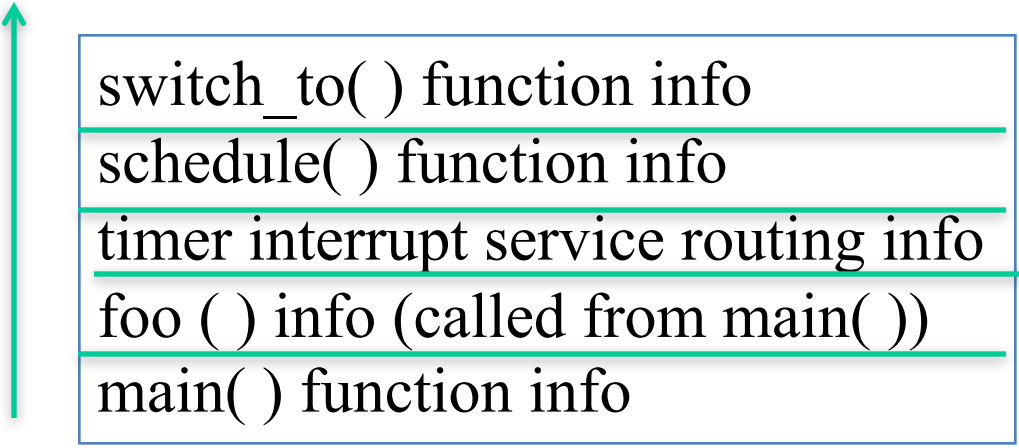
```
    load the state of next_proc from next_proc's kernel stack
```

```
}
```

Context switch Example

- For simplicity, we will assume a single stack
- Suppose P1 and P2 alternate their execution on CPU
- Lets assume P1 is running and a timer interrupt occurs to preempt this process

P1's stack



A vertical stack of five frames, each separated by a horizontal line. To the left of the stack is a green arrow pointing upwards. The frames contain the following text from top to bottom: 'switch_to() function info', 'schedule() function info', 'timer interrupt service routing info', 'foo () info (called from main())', and 'main() function info'.

switch_to() function info
schedule() function info
timer interrupt service routing info
foo () info (called from main())
main() function info

```
switch_to (p1, p2)
{
    save p1's state
    load p2's state
}
```

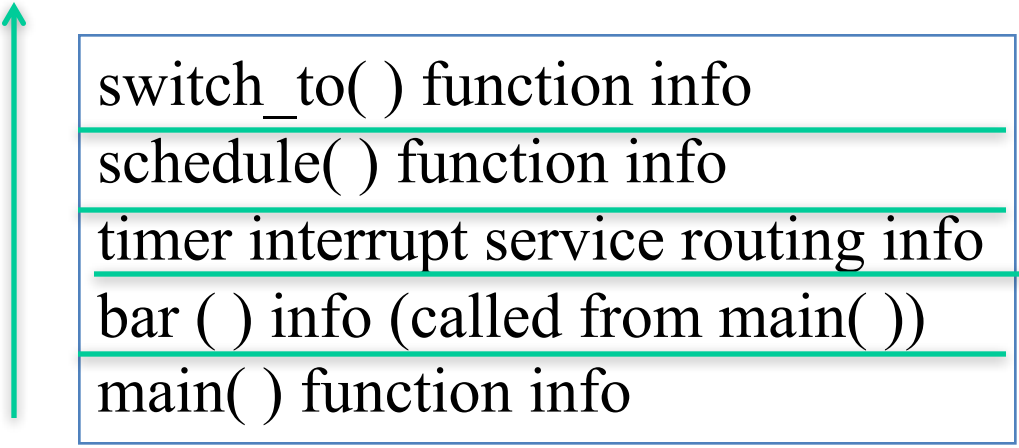
P1's instruction ptr



A green arrow points from the text 'P1's instruction ptr' to the closing curly brace of the 'switch_to' function.

- Since P2's state is loaded, P2 starts running
- Lets assume a timer interrupt occurs to preempt this process
- Now P1 starts running, since its state is loaded

P2's stack



A diagram showing a stack structure. On the left, a vertical green arrow points upwards. To its right is a rectangular box divided into five horizontal sections, each containing text. The sections from top to bottom are: 'switch_to() function info', 'schedule() function info', 'timer interrupt service routing info', 'bar () info (called from main())', and 'main() function info'.

switch_to() function info
schedule() function info
timer interrupt service routing info
bar () info (called from main())
main() function info

```
switch_to (p2, p1)
{
    save p2's state
    load p1's state
}
```

P2's instruction ptr



A green arrow points from the text 'P2's instruction ptr' to the closing curly brace '}' of the switch_to function code block.

- Now P1 starts running, since its state is loaded
- P1's instruction pointer is pointing to the end of its `switch_to()`, so that function will return
- Next, last statement of the schedule function will execute and that function will return
- Next, the timer interrupt service routine completes and returns
- Finally, the execution of the `foo()` function resumes from the point where it was executing before P1 was preempted

