

CSCI 3753

Operating Systems

Processes

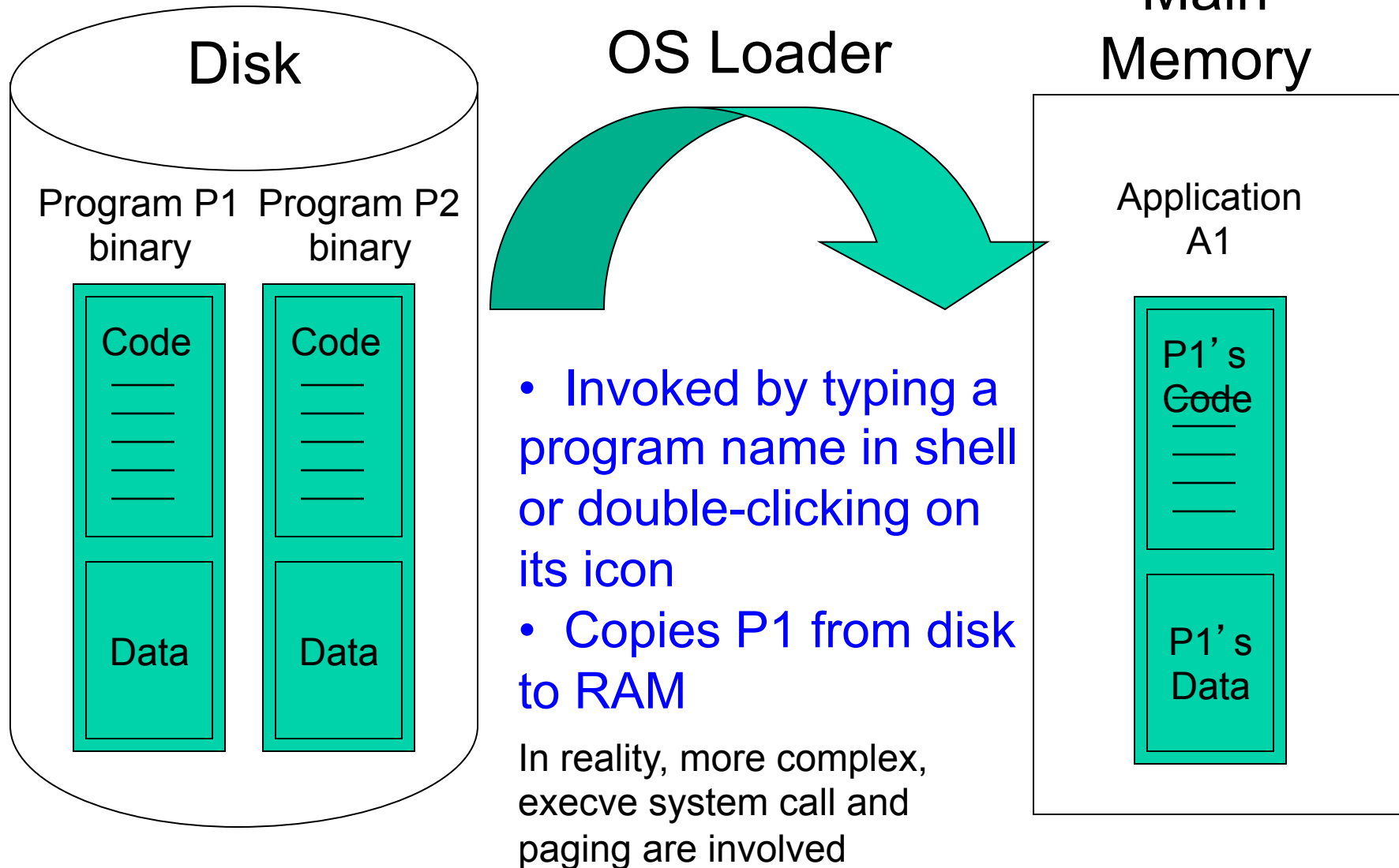
Lecture Notes By

Shivakant Mishra

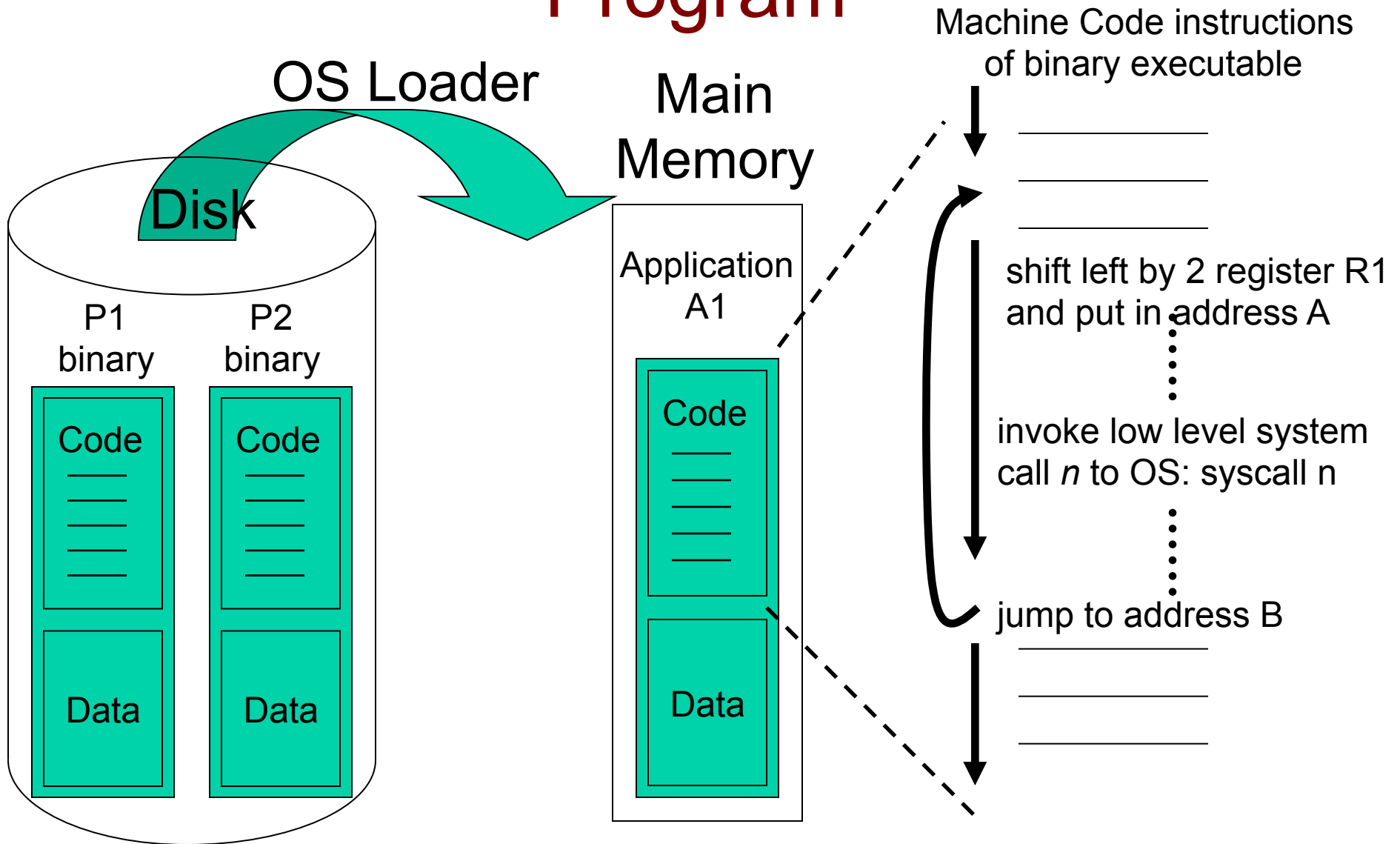
Computer Science, CU-Boulder

Last Update: 01/31/13

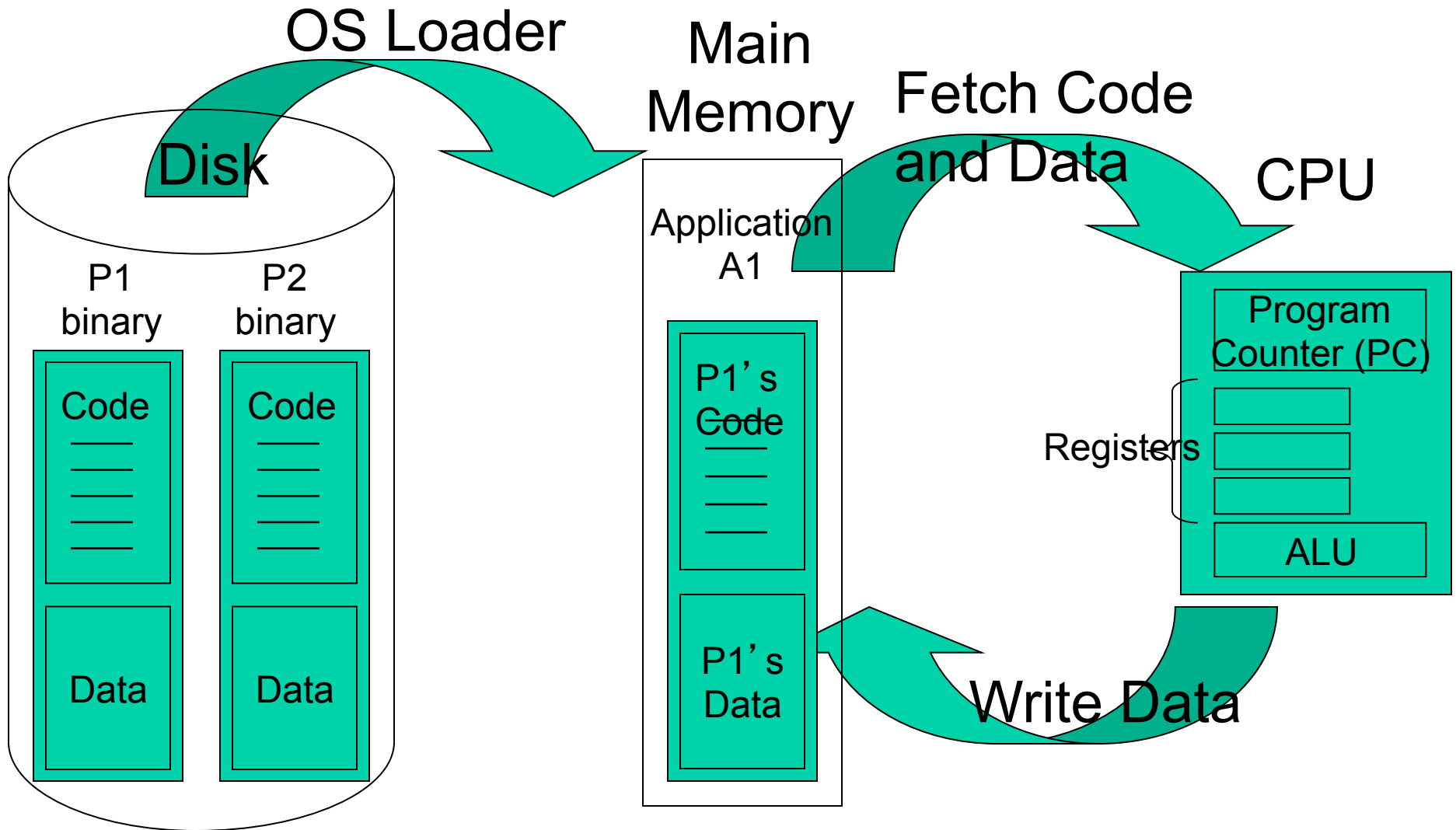
Loading a Program into Memory



Loading and Executing a Program

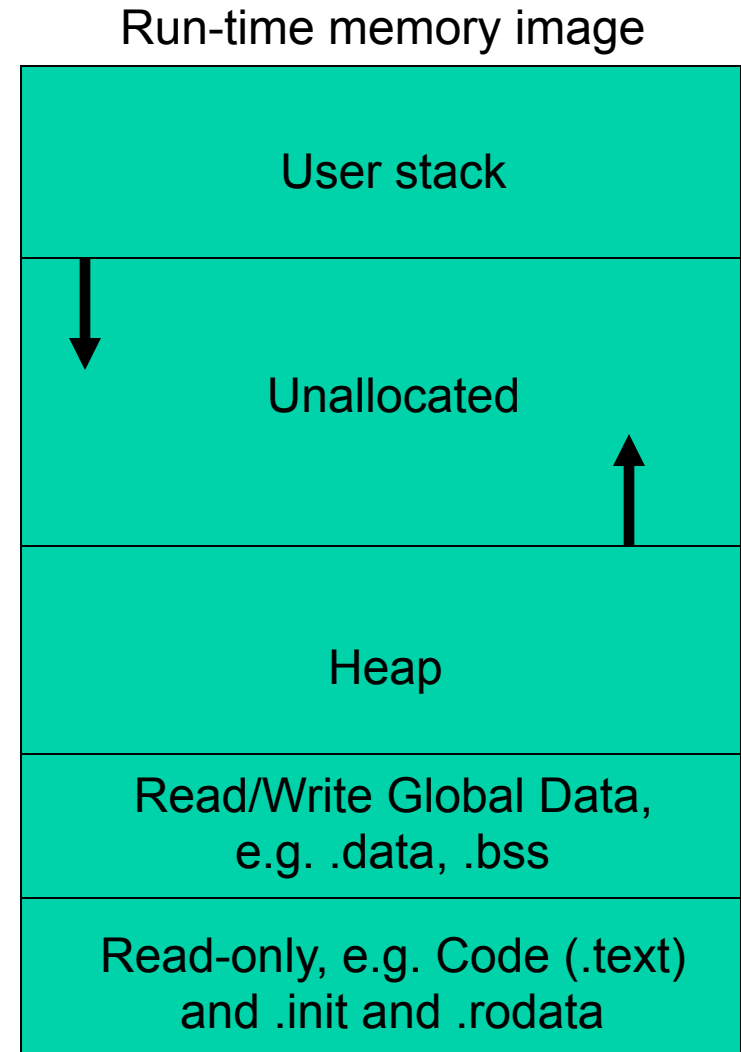


Loading and Executing a Program



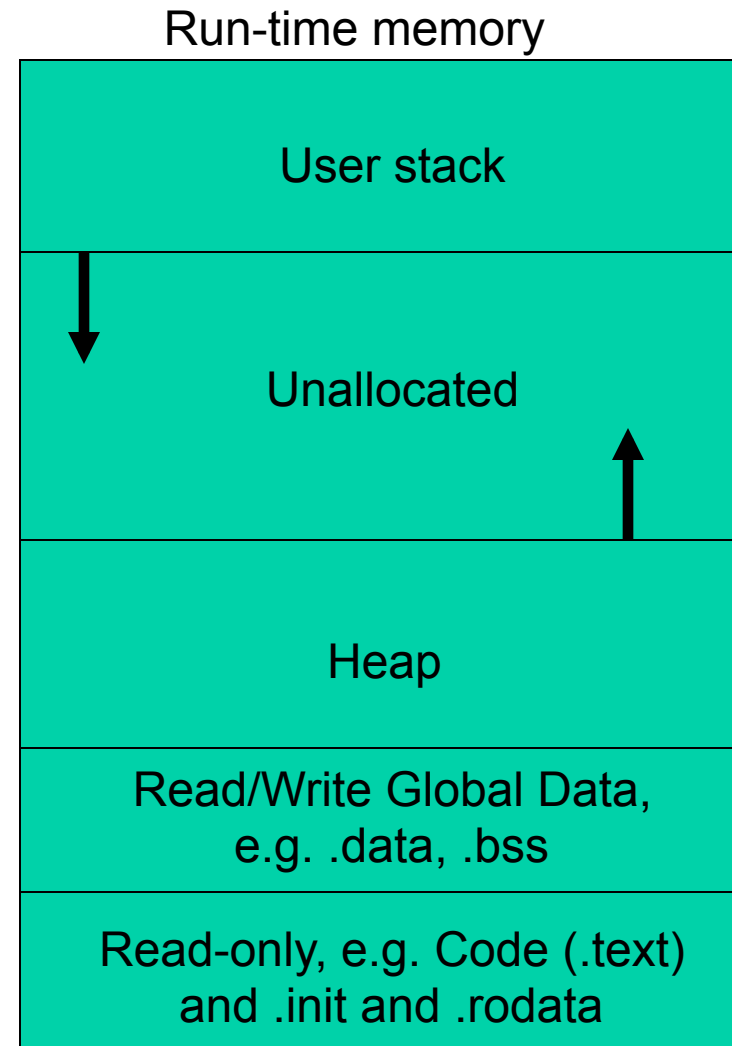
Loading Executable Object Files

- 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*
 - Usually, stack grows downward from high memory, heap grows upward from low memory, but this architecture-specific

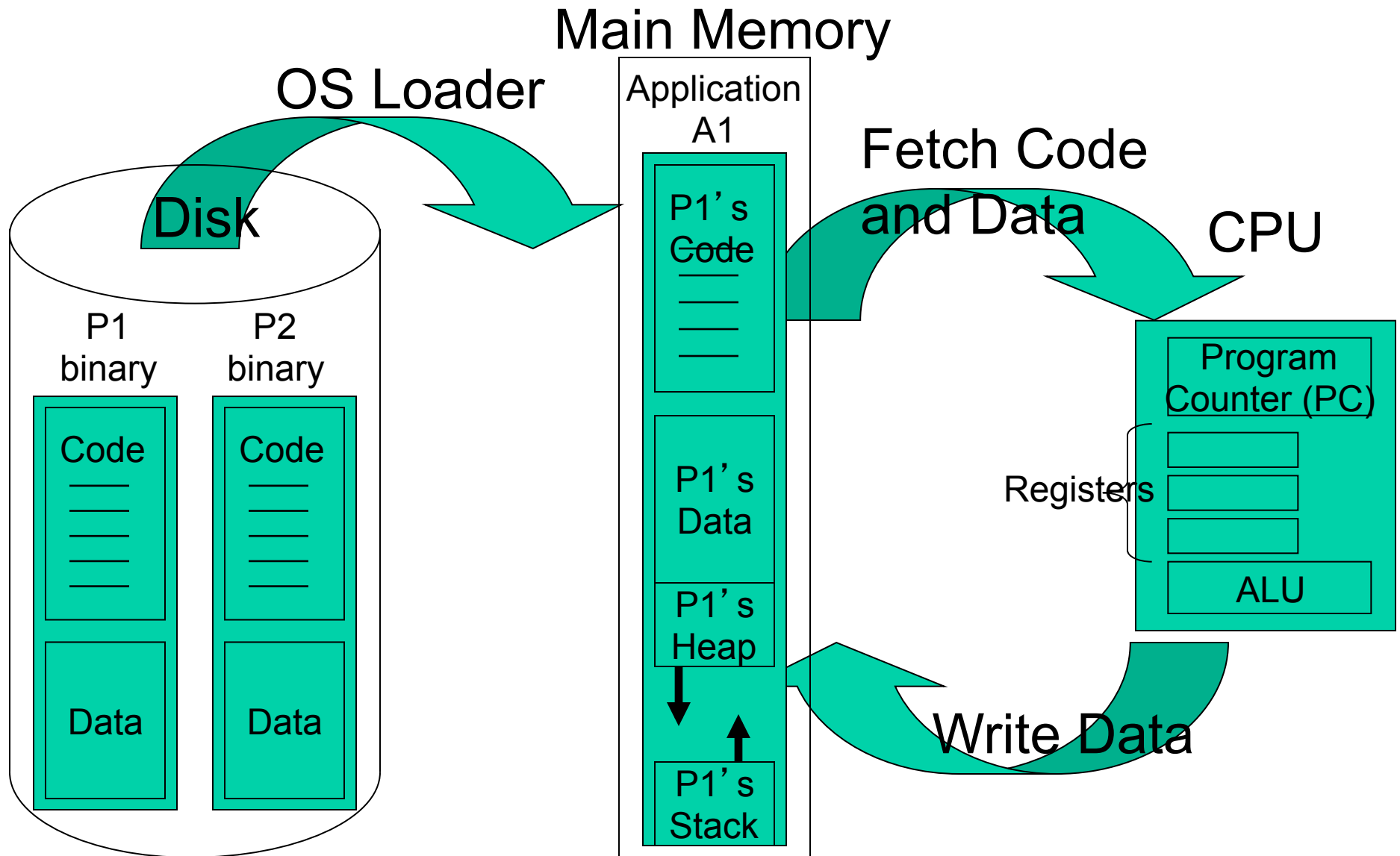


Running Executable Object Files

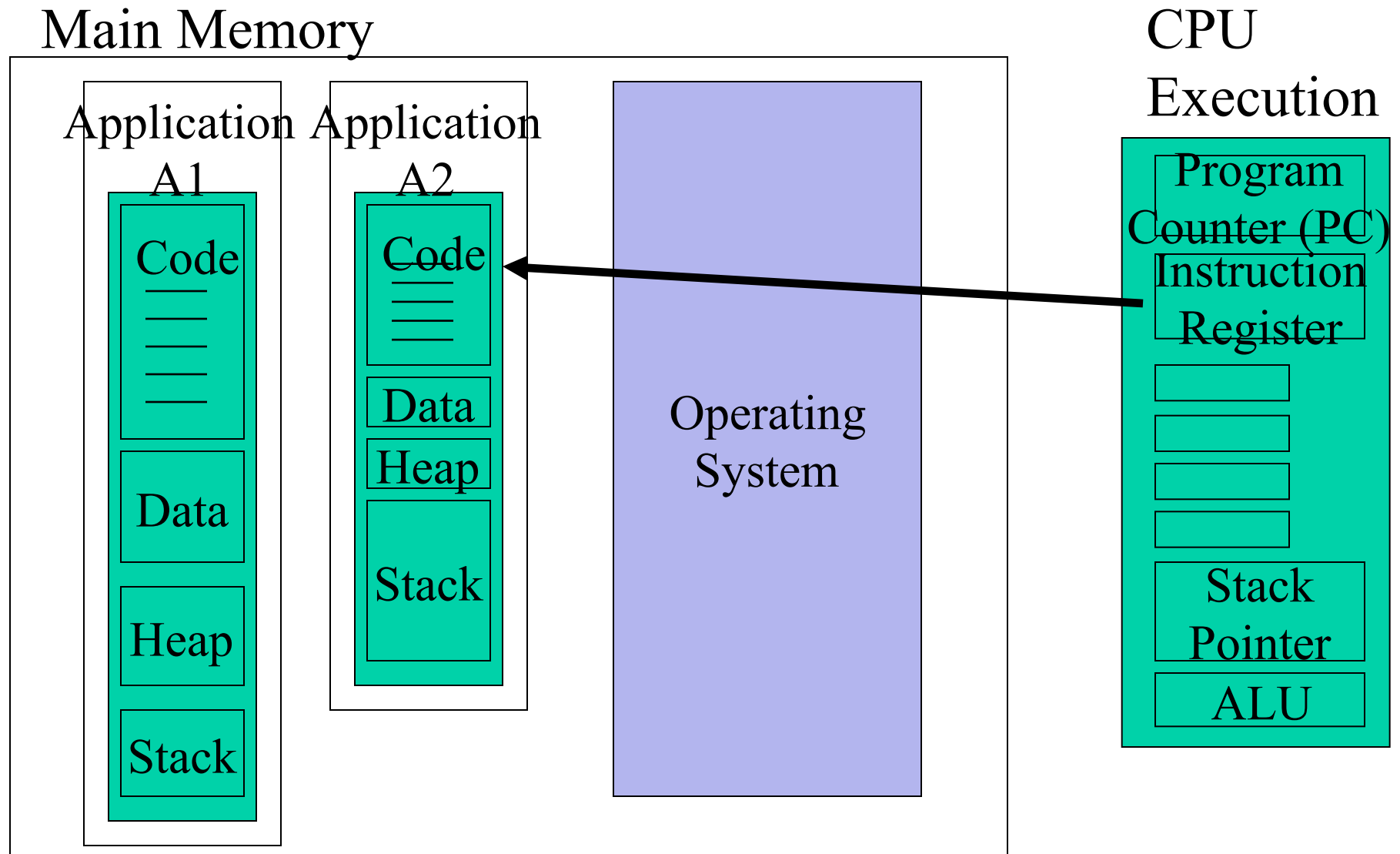
- Stack contains local variables
 - As `main()` calls function `f1`, we allocate `f1`'s local variables on the stack
 - If `f1` calls `f2`, we allocate `f2`'s variables on the stack below `f1`'s, thereby growing the stack, etc...
 - When `f2` is done, we deallocate `f2`'s local variables, popping them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



Loading and Executing a Program – a more complete picture

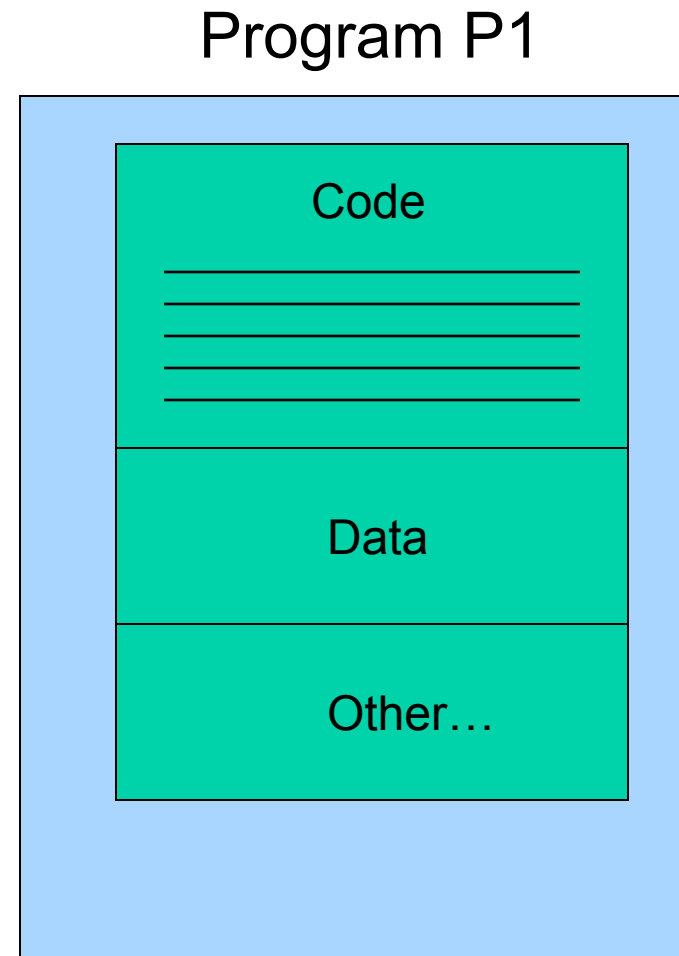


Multiple Applications + OS

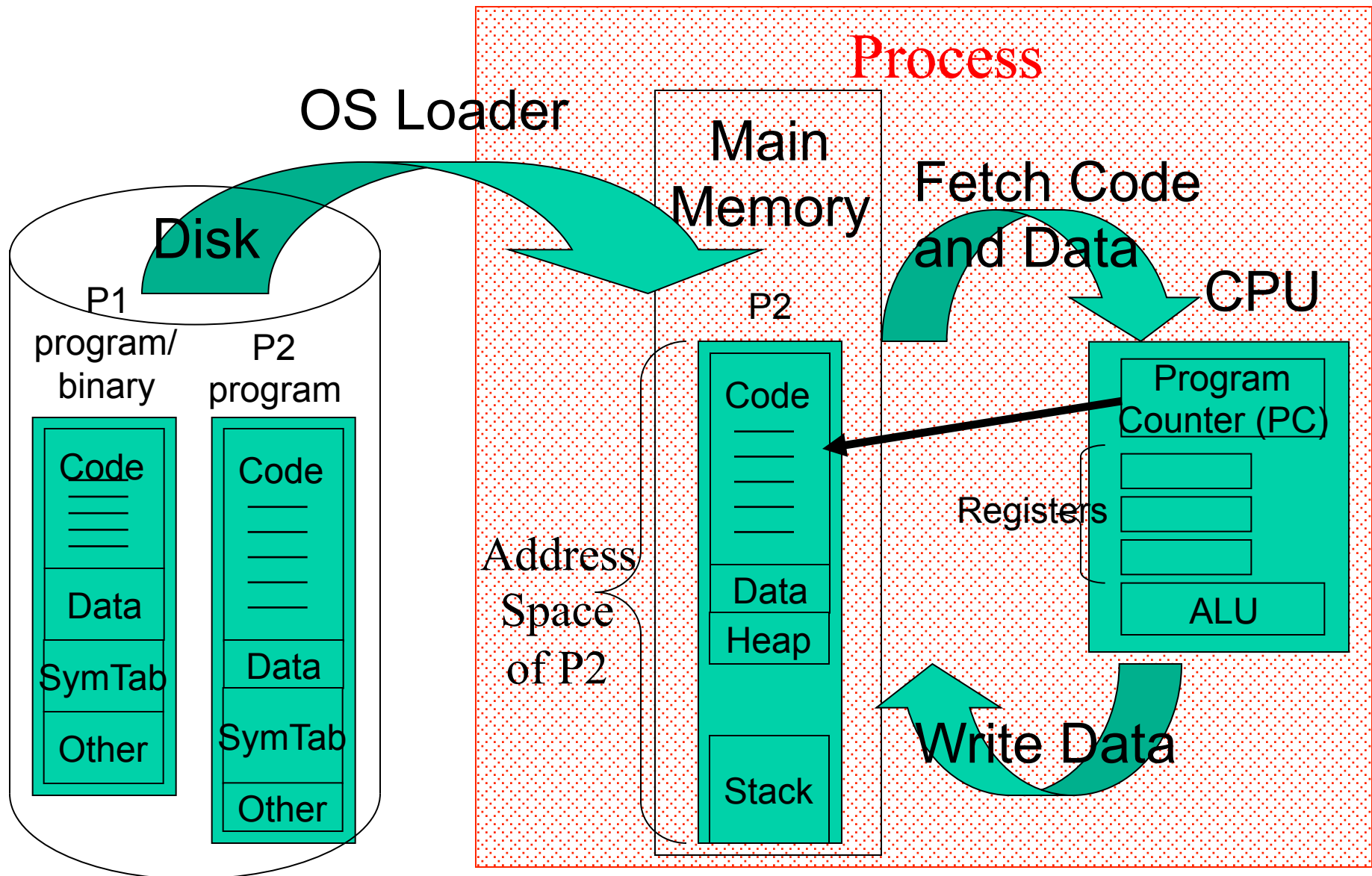


Chapter 3: What is a Process?

- 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*

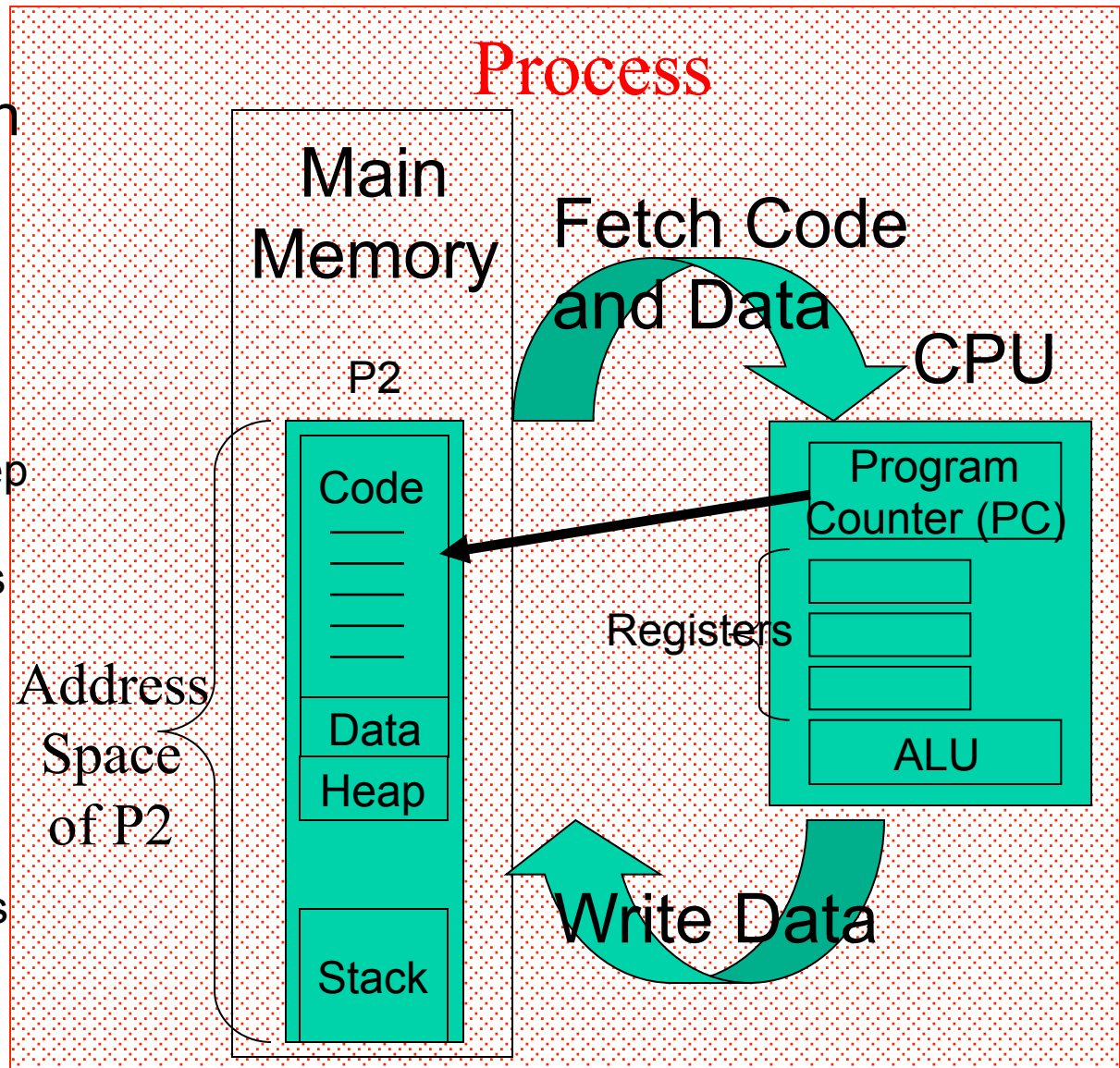


What Is a Process? (2)



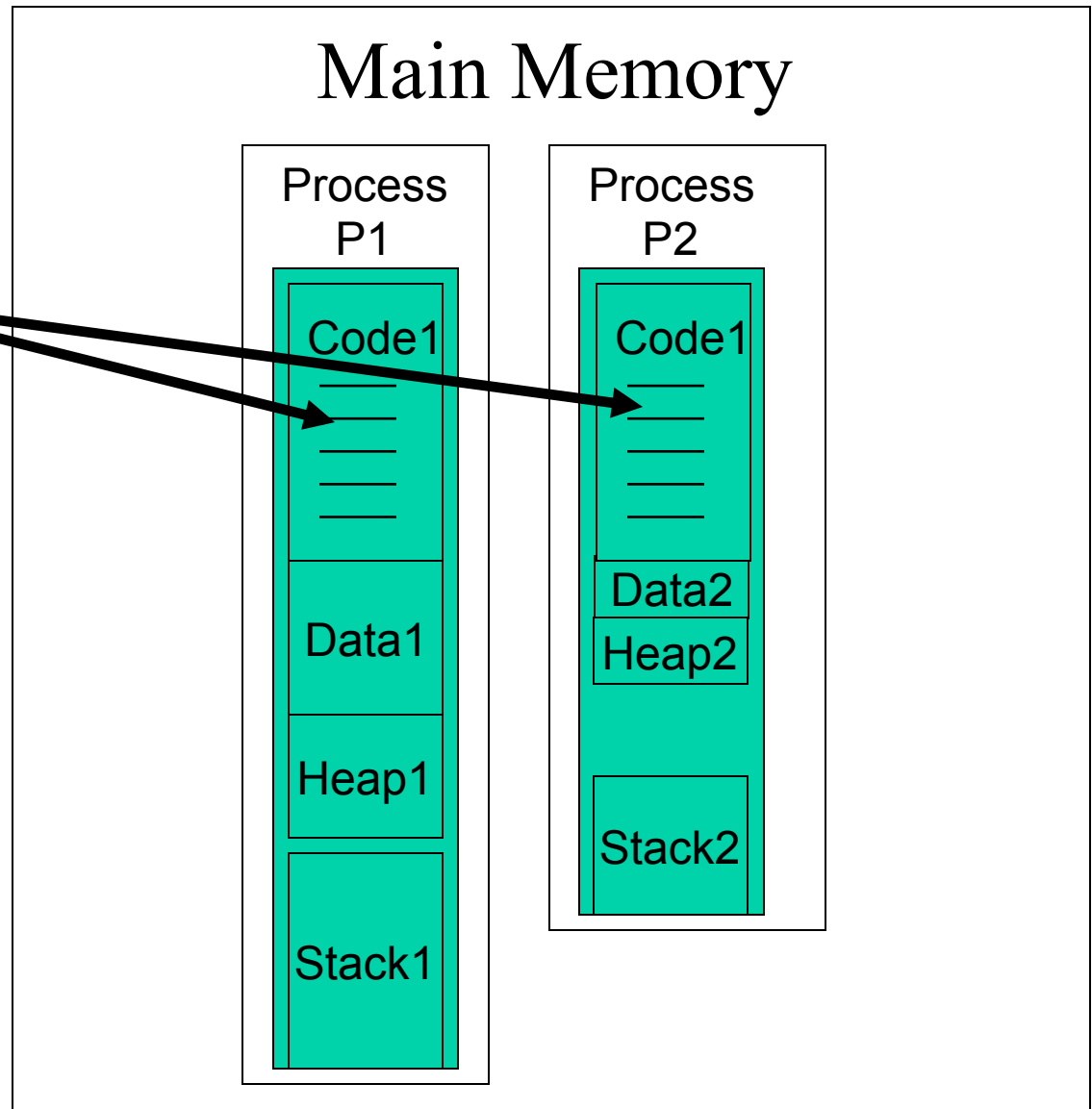
What is a Process? (3)

- 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



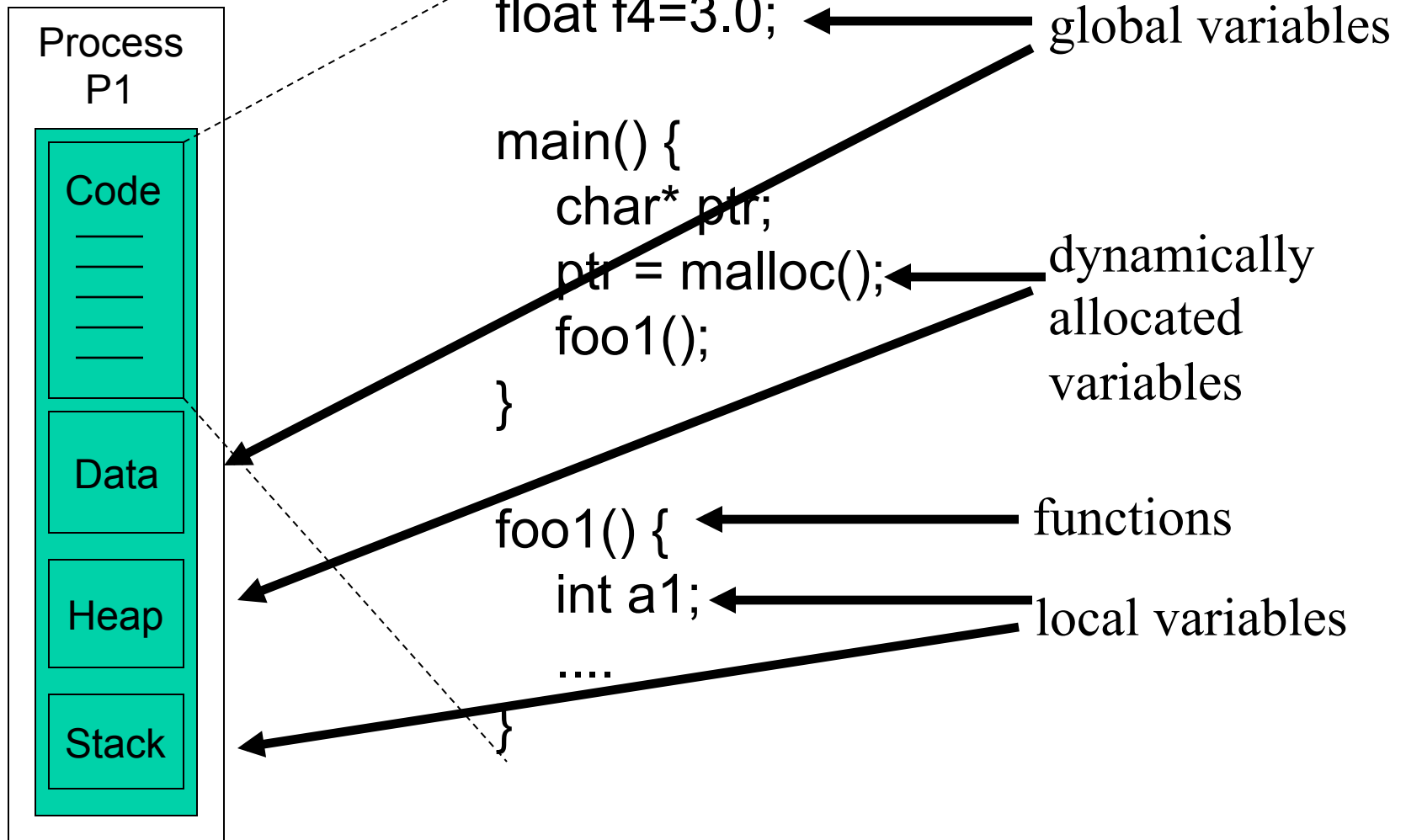
What is a Process? (4)

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



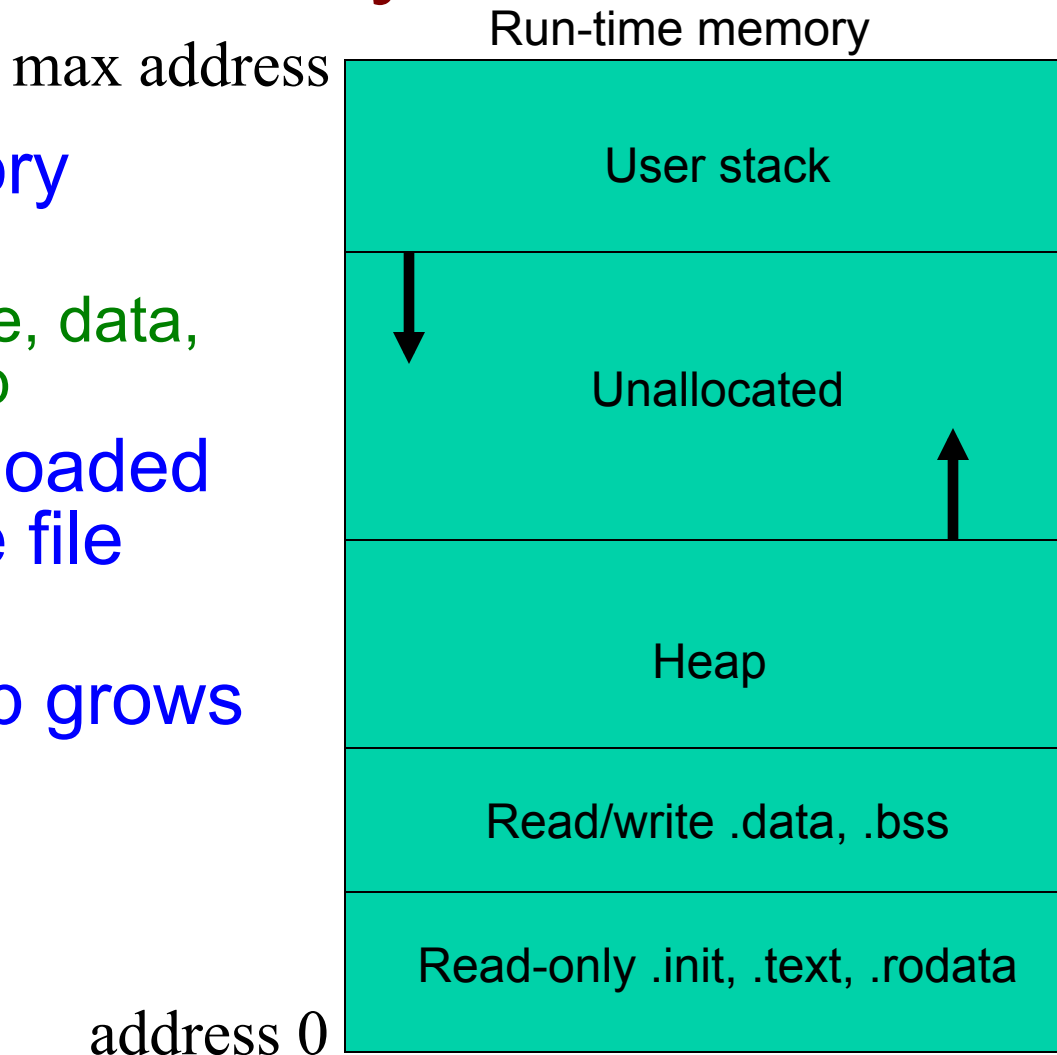
How is a Process Structured in Memory?

Main Memory



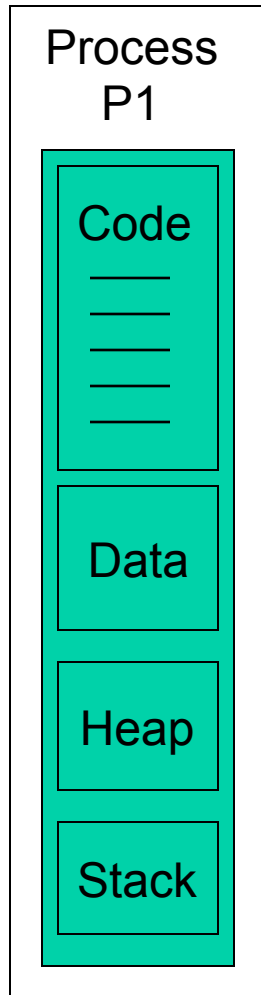
How is a Process Structured in Memory?

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



A Process Executes in its Own Address Space

Main Memory

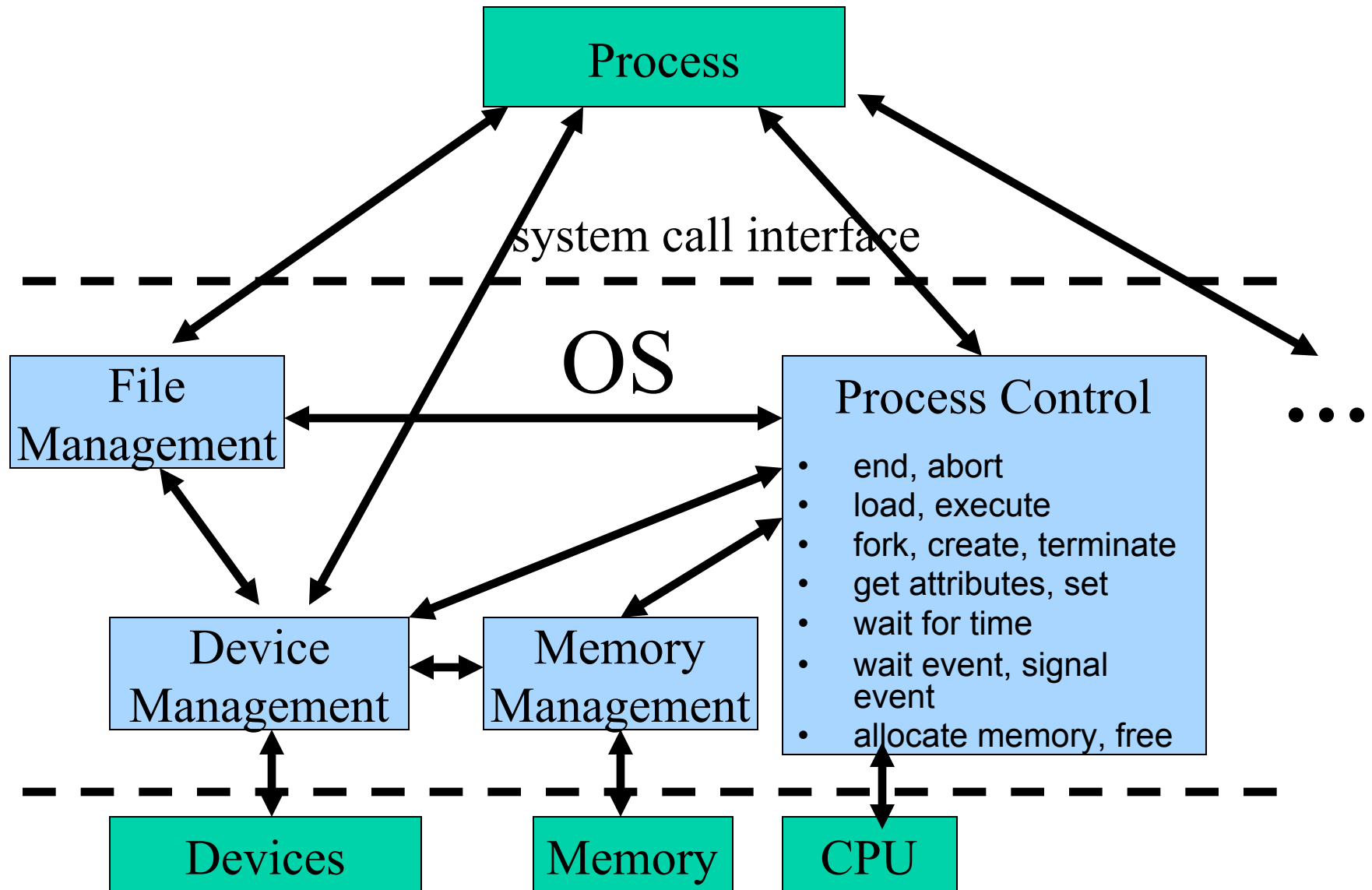


- OS tries to provide the illusion or *abstraction* to the process that it executes
 - in its own subset of RAM, i.e. its own address space
 - on its own subset (time slice) of the CPU

Applications and Processes

- 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.

Process Management



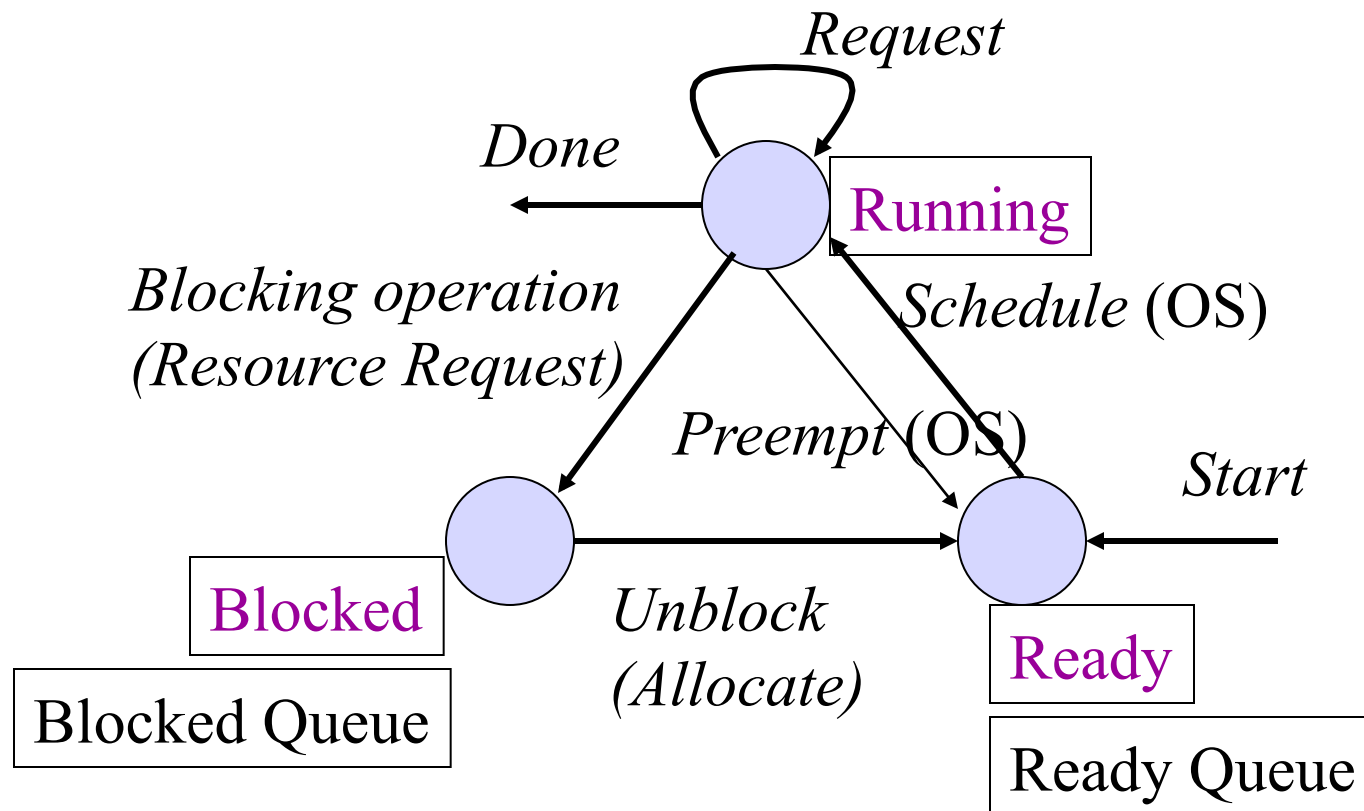
Process Manager

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

Process State

- Memory image: Code, data, heap, stack
- 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

Process State Diagram

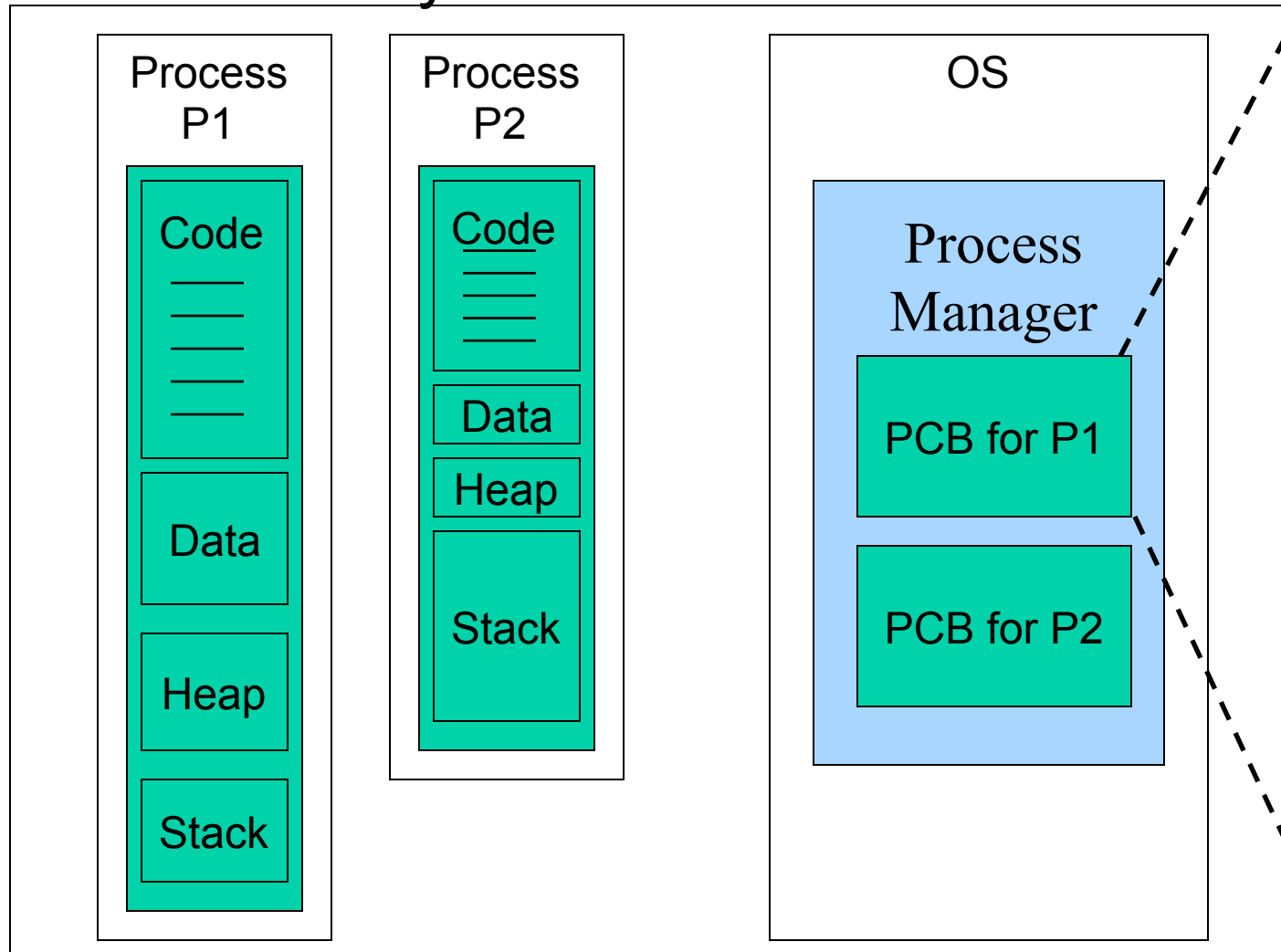


Process Control Block

- 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

Context Switch

- Running state → Ready state
- Running state → Blocked state
- Switching the CPU from currently running process to another process
 - Save the state of the currently running process in its PCB
 - Load the saved state of new process scheduled to run from its PCB
 - Context switch time is pure overhead: 1 – 1000 microseconds.
 - An important goal in OS design is to minimize context switch time.

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

- 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 */
```

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 */
```

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

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

<Figure>

Communicating Between Processes

- **Inter-Process Communication (IPC):** Want to communicate between two processes because (why?)
 - An application may split its tasks into two or more processes for reasons of convenience and/or performance
 - e.g. Web server and cgi-bin (Common Gateway Interface) – for certain URLs, Web server creates a new cgi-bin process to service the request, e.g. dynamic content generation
 - Creating a new process is expensive
 - Sharing information
 - Improved fault isolation

Communicating Between Processes

- Two types of IPC

1. *shared memory* - OS provides mechanisms that allow creation of a shared memory buffer between processes

- `shmid = shmget (key name, size, flags)` is part of the POSIX API that creates a shared memory segment, using a name (key ID)
 - All processes sharing the memory need to agree on the key name in advance.
 - Creates a new shared memory segment if no such shared memory with the same name exists and returns handle to the shared memory. If it already exists, then just return the handle.

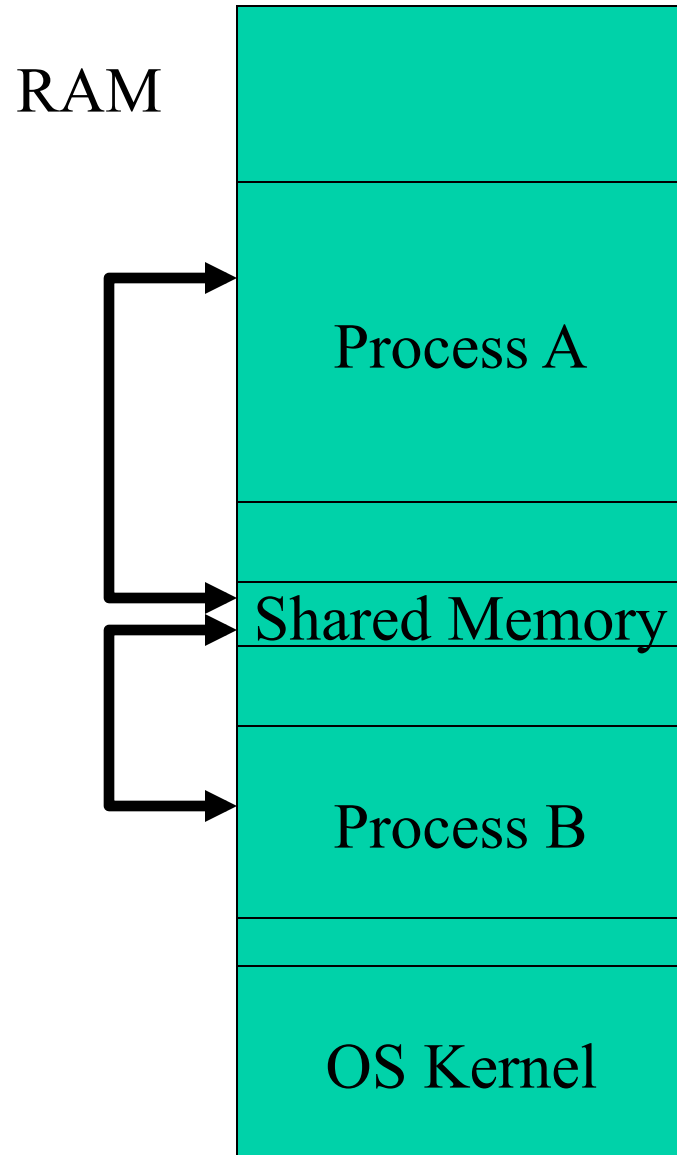
Communicating Between Processes (2)

- Two types of IPC

- 1. *shared memory* (cont.)

- `shm_ptr = shmat (shmid, NULL, 0)` to attach a shared memory segment to a process's address space
 - This association is also called binding
 - Reads and writes now just use `shm_ptr`
 - `shmctl()` to modify control information and permissions related to a shared memory segment, & to remove a shared memory segment

IPC Shared Memory

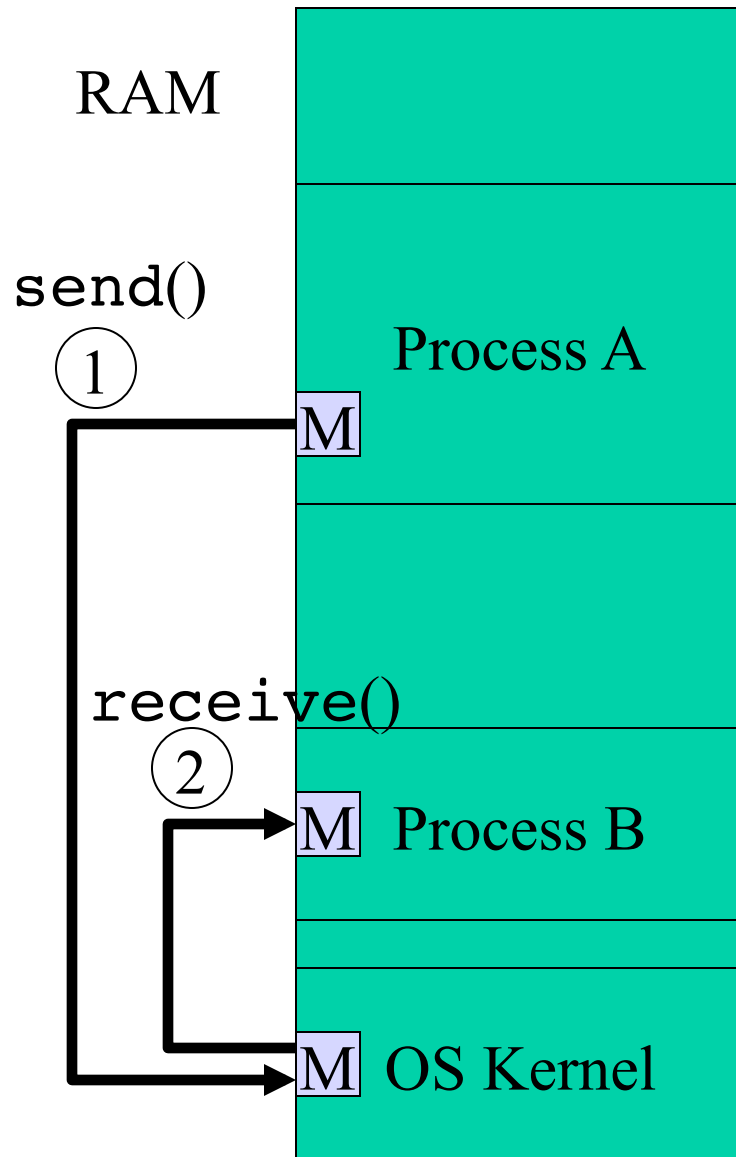


- allows fast and high volume reading/writing of buffer in memory
- applies to processes on the same machine
- Problem: shared access to the same memory introduces complexity
 - need to synchronize access
 - Producer-Consumer example
 - if two producers write at the same time to shared memory, then they can overwrite each other's data
 - if a producer writes while a consumer is reading, then the consumer may read inconsistent data

IPC Message Passing

- Two types of IPC
 2. *message passing* - OS provides constructs that allow communication via buffers
 - Basic primitives are `send()` and `receive()`
 - typically implemented via system calls, and is hence slower than shared memory
 - Sending process has a buffer to send/receive messages, as does the receiving process and OS
 - In direct message-passing, processes send directly to each other's buffers
 - In indirect message-passing, sending process sends a message to OS by calling `send()`. OS acts as a buffer or mailbox for relaying the message to the receiving process, which calls `receive()` to retrieve message

IPC Message Passing



Example of indirect message-passing

- `send()` and `receive()` can be blocking/synchronous or non-blocking/asynchronous
- used to pass small messages
- Advantage: doesn't require synchronization
- Disadvantage: Slow - OS is involved in each IPC operation for control signaling and possibly data as well
- Message Passing IPC types: pipes, UNIX-domain sockets, Internet domain sockets, message queues, and remote procedure calls (RPC)

Using Sockets for UNIX IPC

Sockets are an example of message-passing IPC. Created in UNIX using `socket()` call:

```
sd = socket(int domain, int type, int protocol);
```

socket descriptor

- = `PF_UNIX` for local Unix domain sockets (local IPC)
- = `PF_INET` for Internet sockets (but can still achieve local communication by specifying localhost address as destination)
- = `SOCK_STREAM` for reliable in-order delivery of a byte stream
- = `SOCK_DGRAM` for delivery of discrete messages
- = 0 usually to select default protocol associated with a type

Using Sockets for UNIX IPC (2)

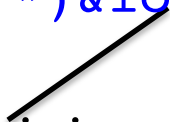
- Each communicating process will first create its own socket, usually `SOCK_STREAM`.
- Now let's consider only the case of UNIX domain sockets (`PF_UNIX` domain):
 - Used only for local communication only among a computer's processes
 - Emulates reading/writing from/to a file
 - Each process `bind()`'s its socket to a filename:

```
bind(sd, (struct sockaddr *)&local, length);
```

socket
descriptor



data structure containing unique unused file
name, e.g. `"/users/shiv/myipcsocketfile"`



Using Sockets for UNIX IPC (3)

- UNIX domain sockets (cont.):
 - Usually, one process acts as the server, and the other processes connect to it as clients

Server code:

```
sd = socket  
    (PF_UNIX, SOCK_STREAM, 0)
```

```
bind(sd, ...)
```

```
listen() for connect requests
```

```
sd2 = accept() a connect  
      request
```

```
recv(sd2, ...) and  
send(sd2, ...)
```

Client code:

```
sd = socket  
    (PF_UNIX, SOCK_STREAM, 0)
```

```
connect(sd, ...) to server
```

```
recv(sd, ...) and  
send(sd, ...)
```

bind and connect must
use same file name!

IPC

IPC via Internet domain sockets

- Set up and connect Internet sockets in a way very similar to Unix domain sockets, but...
 - Configure the socket with domain `PF_INET` instead and set destination to localhost (say `127.0.0.1`) instead of the usual remote Internet IP address
 - And need to choose a well-known port # that is shared between processes, i.e. P1 and P2 know this port # in advance – this is similar to the well-known file name
 - Both processes then `send()` and `receive()` messages via this port and socket
 - Advantage of writing applications to use this type of IPC via Internet domain sockets is that it is arguably more portable than UNIX-domain sockets