slides kindly provided by:

Tam Vu, Ph.D
Professor of Computer Science
Director, Mobile & Networked Systems Lab
Department of Computer Science
University of Colorado Boulder

CSCI 3753 Operating Systems Summer 2019

Christopher Godley

PhD Student

Department of Computer Science

University of Colorado Boulder

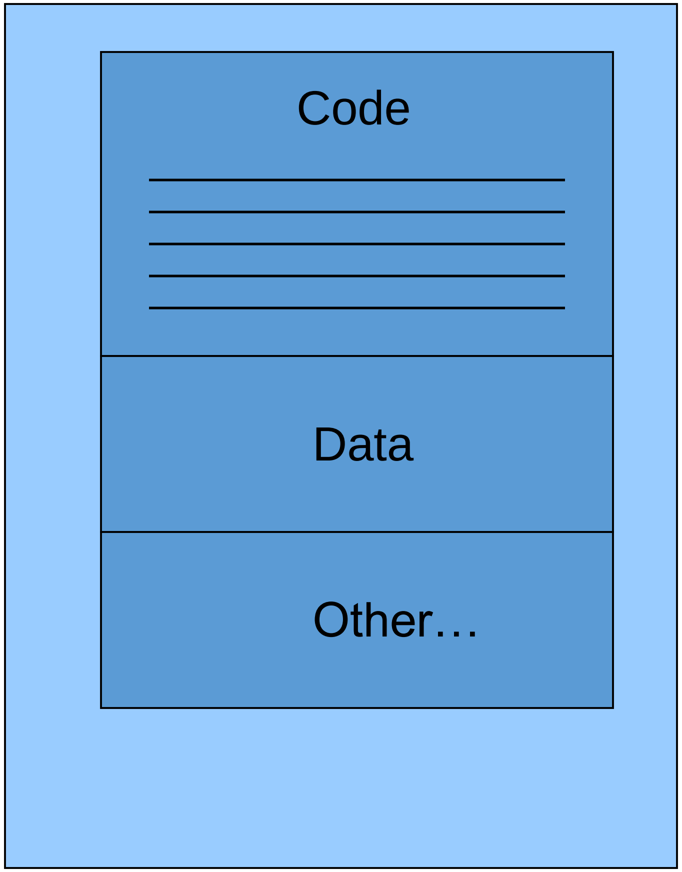
Processes

What is a Process?

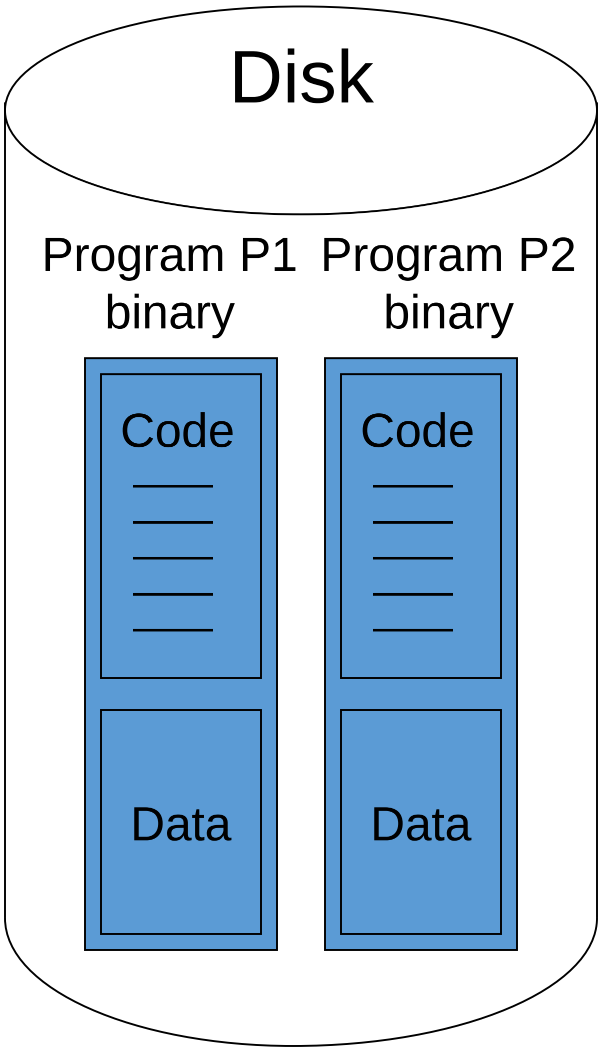
- A software *program* consist of a sequence of code instructions and data stored on disk
  - A program is a *passive* entity

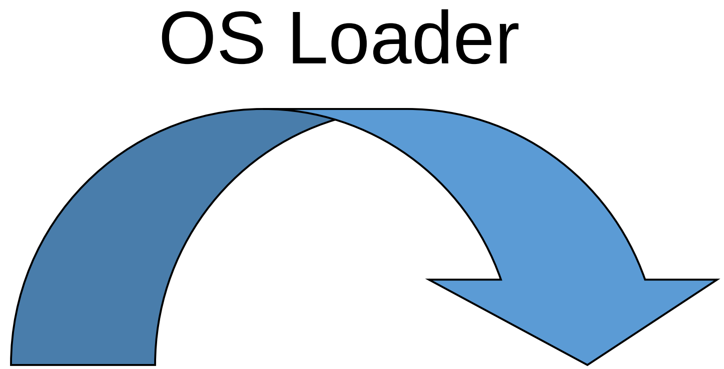
- A *process* is a program ***actively executing*** from main memory within its ***own address space***

Program P1



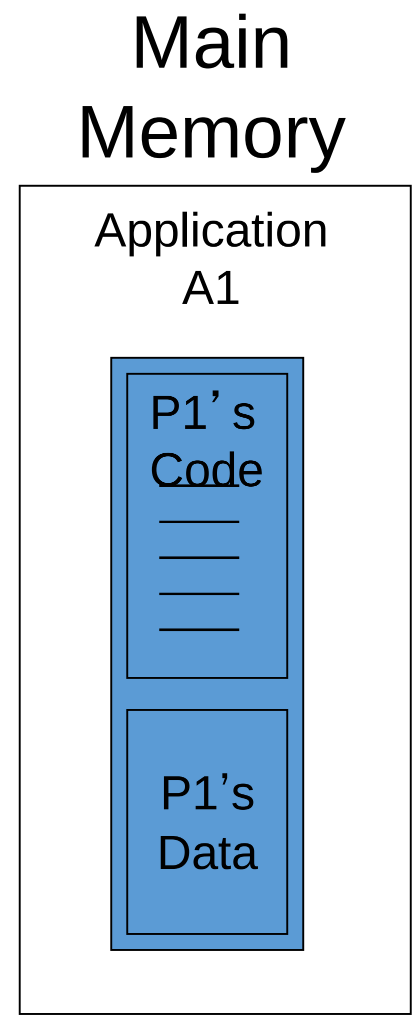
Loading a Program into Memory



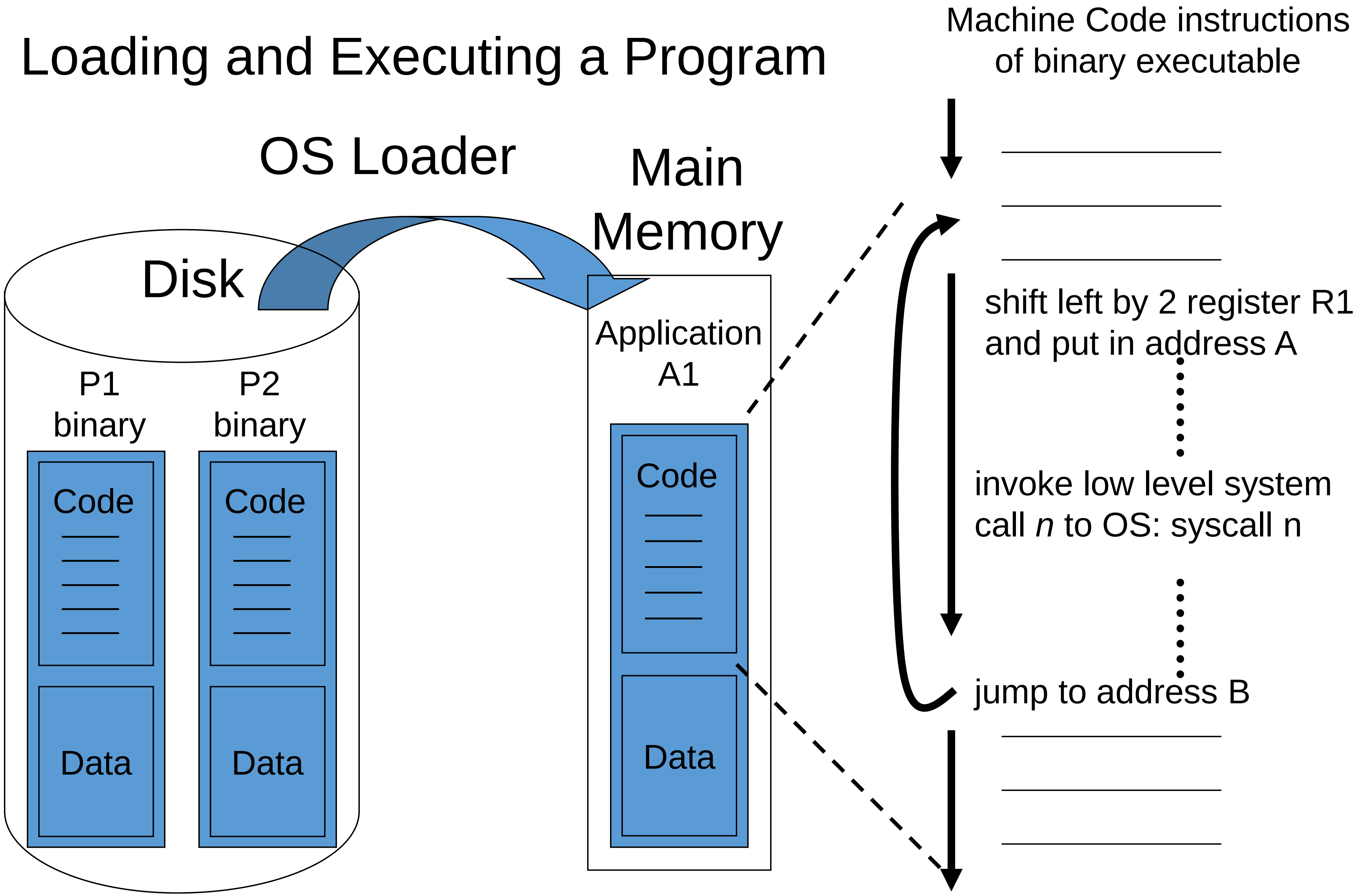


- Invoked by typing a program name in shell or double-clicking on its icon
- Copies P1 from disk to RAM

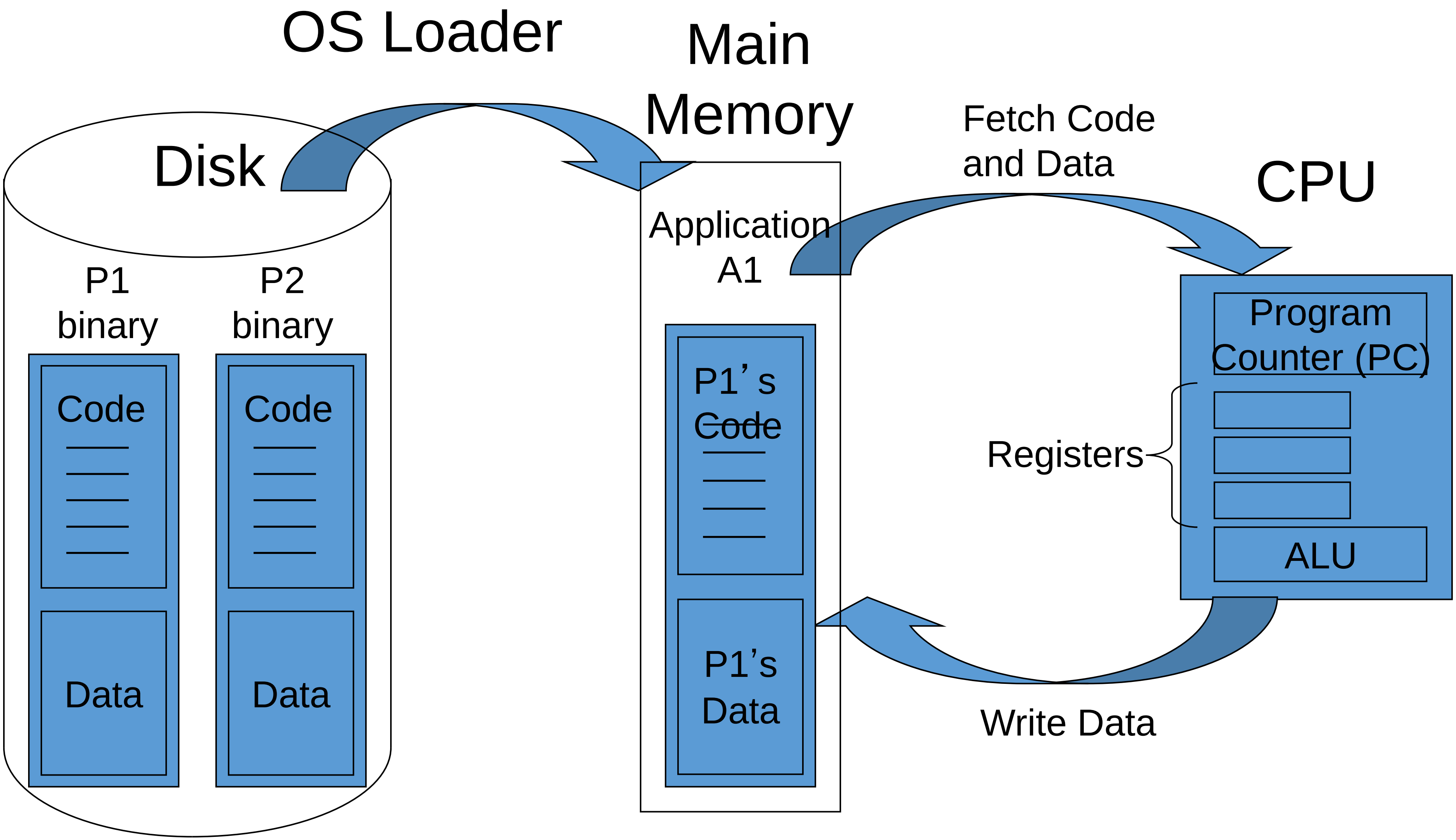
In reality, more complex, *execve* system call and paging are involved



Loading and Executing a Program



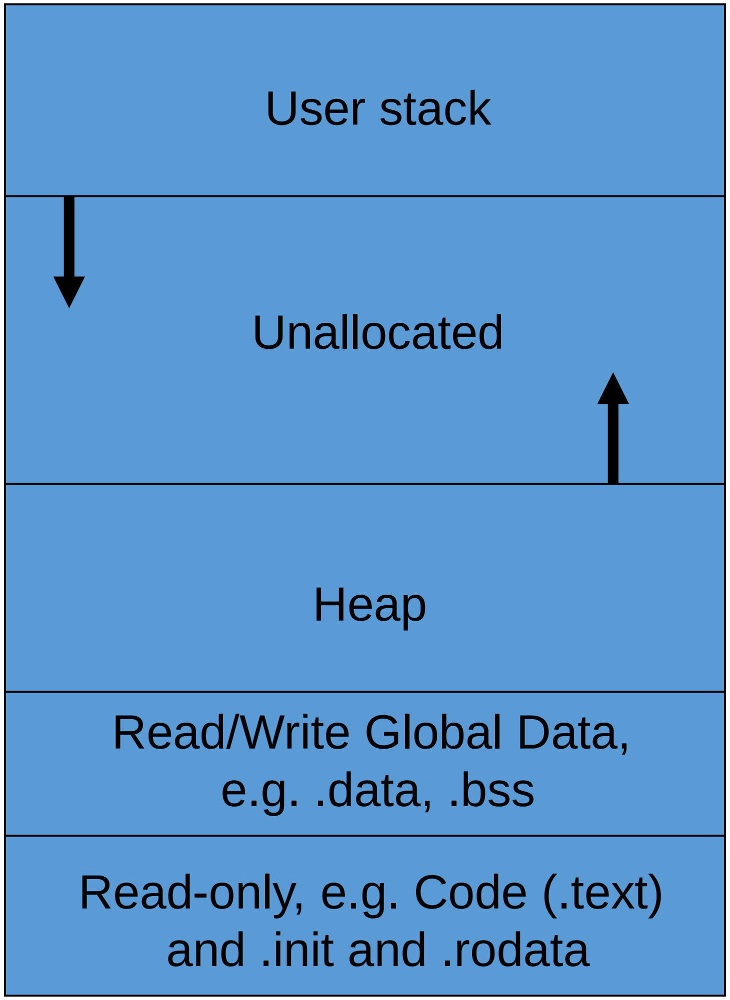
Loading and Executing a Program



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

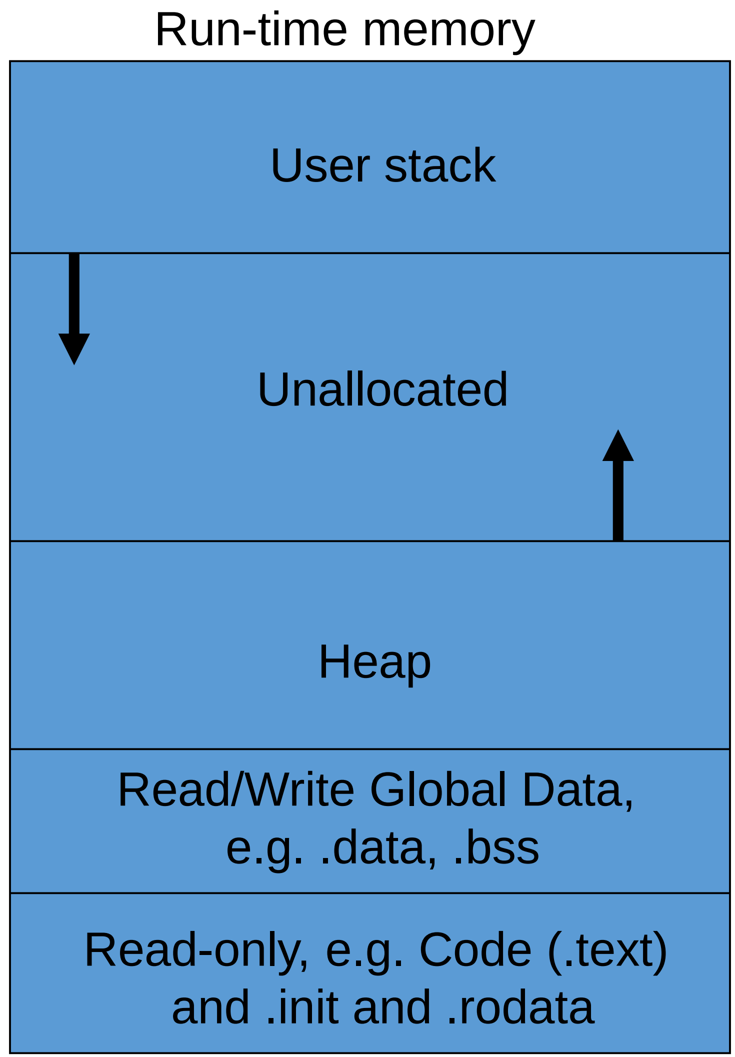
Run-time memory image



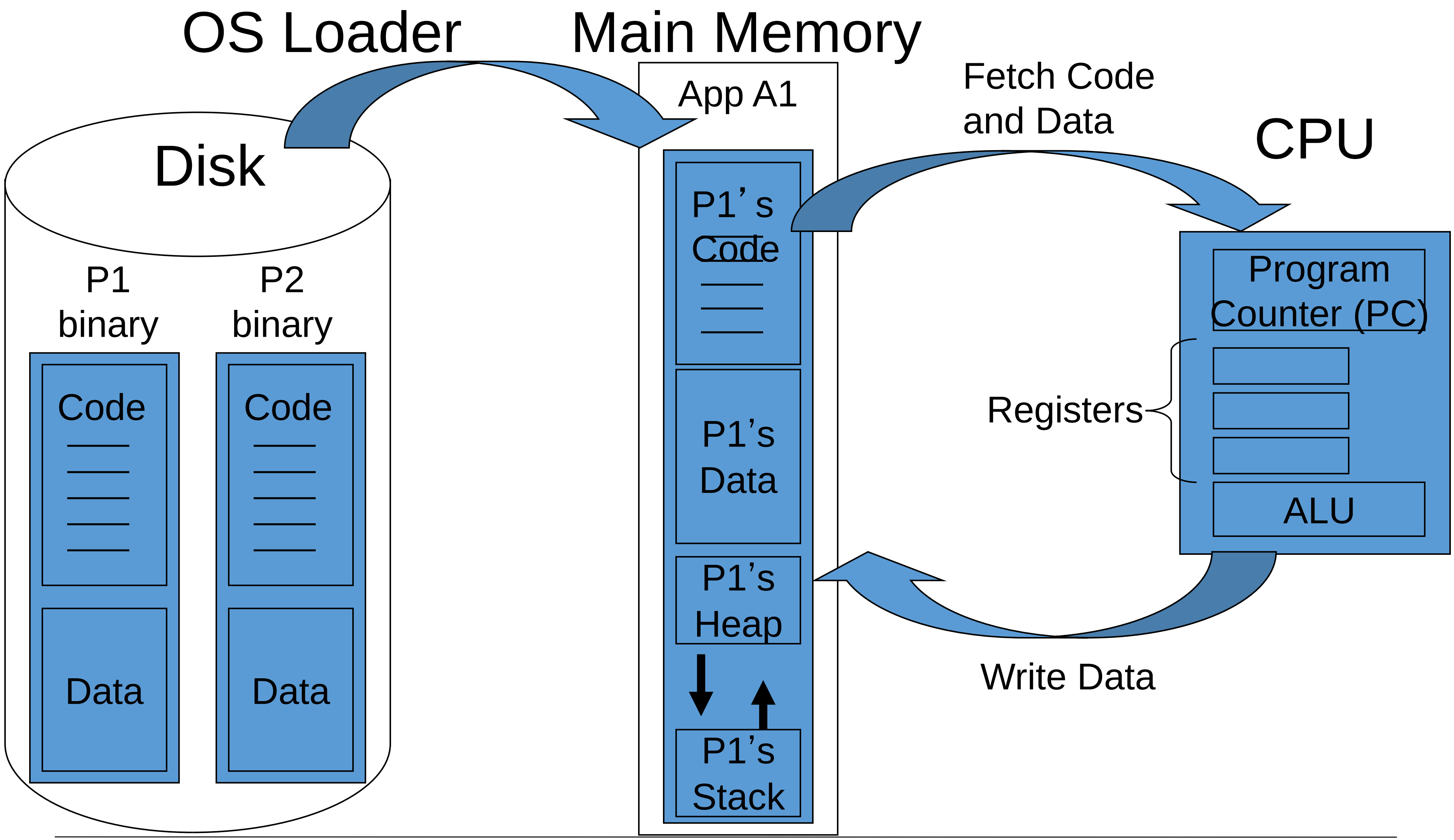
But this is architecture-specific

Running Executable Object Files

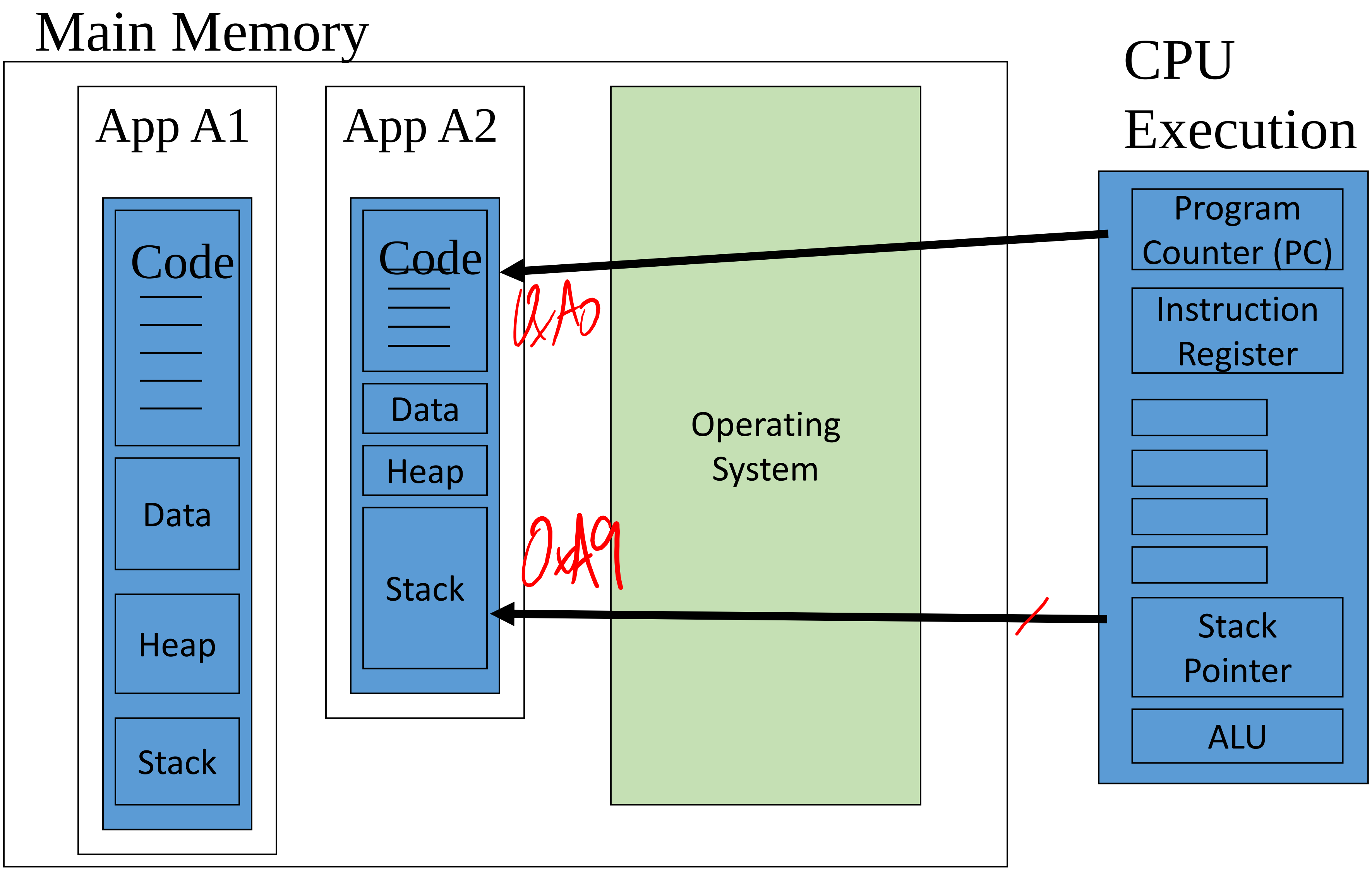
- Stack contains local variables
  - As main() calls function *f1*, we allocate *f1's* local variables on the stack
  - If f1 calls *f2*, we allocate *f2's* variables on the stack below *f1's*, thereby growing the stack, etc…
  - When *f2* is done, we deallocate *f2's* local variables, popping them off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



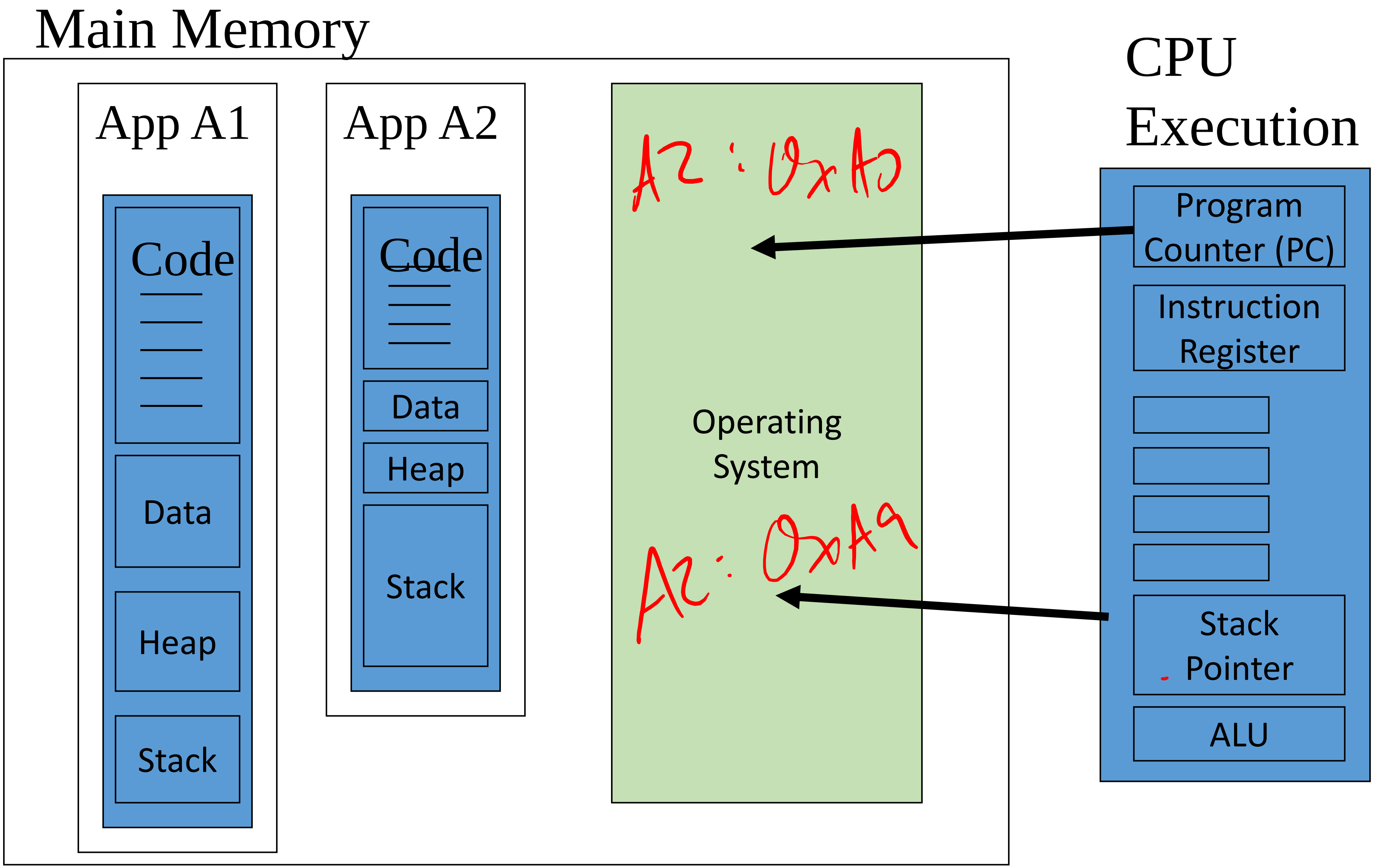
Loading and Executing a Program
– a more complete picture –



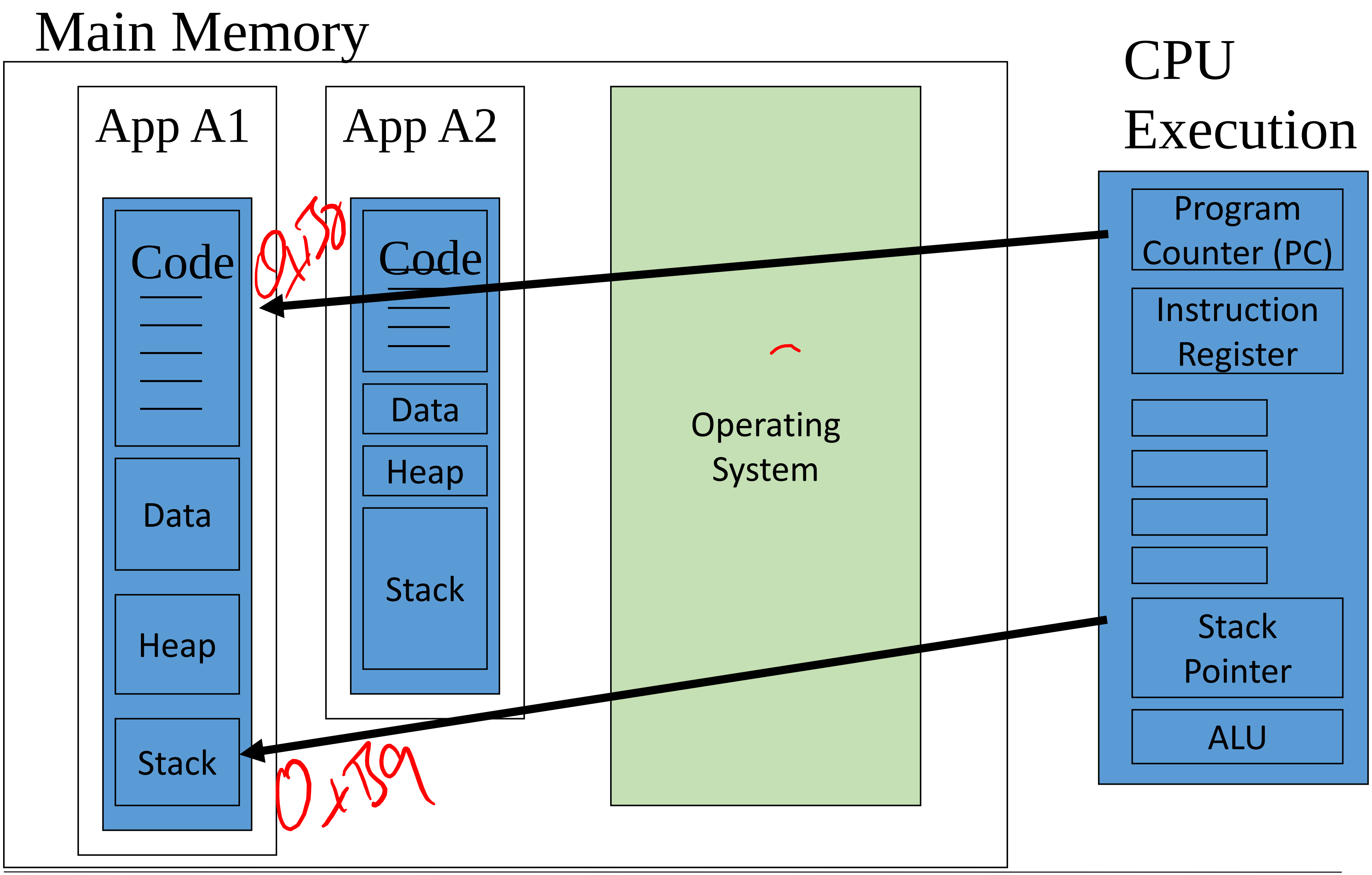
Multiple Applications + OS



Multiple Applications + OS



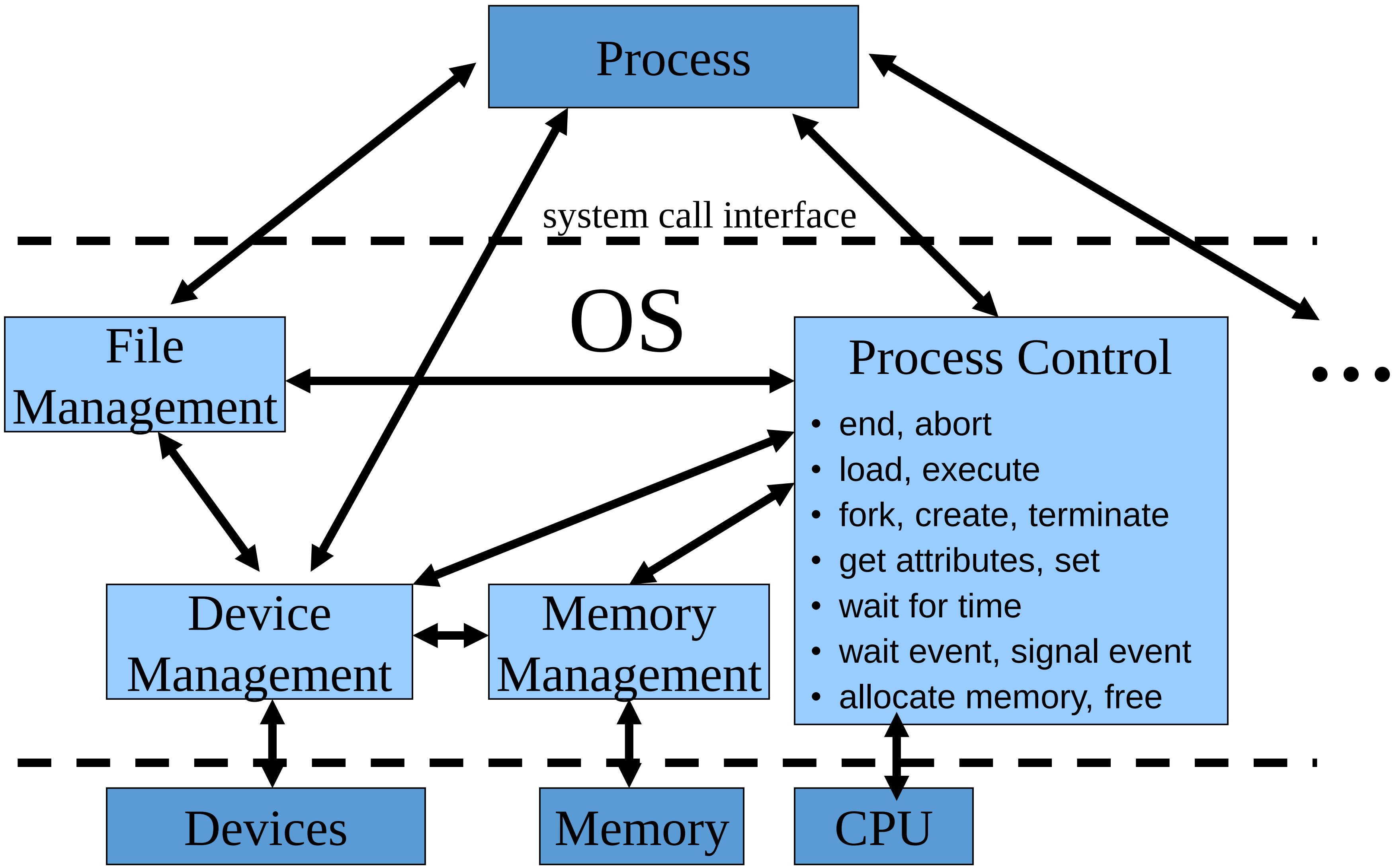
Multiple Applications + OS



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



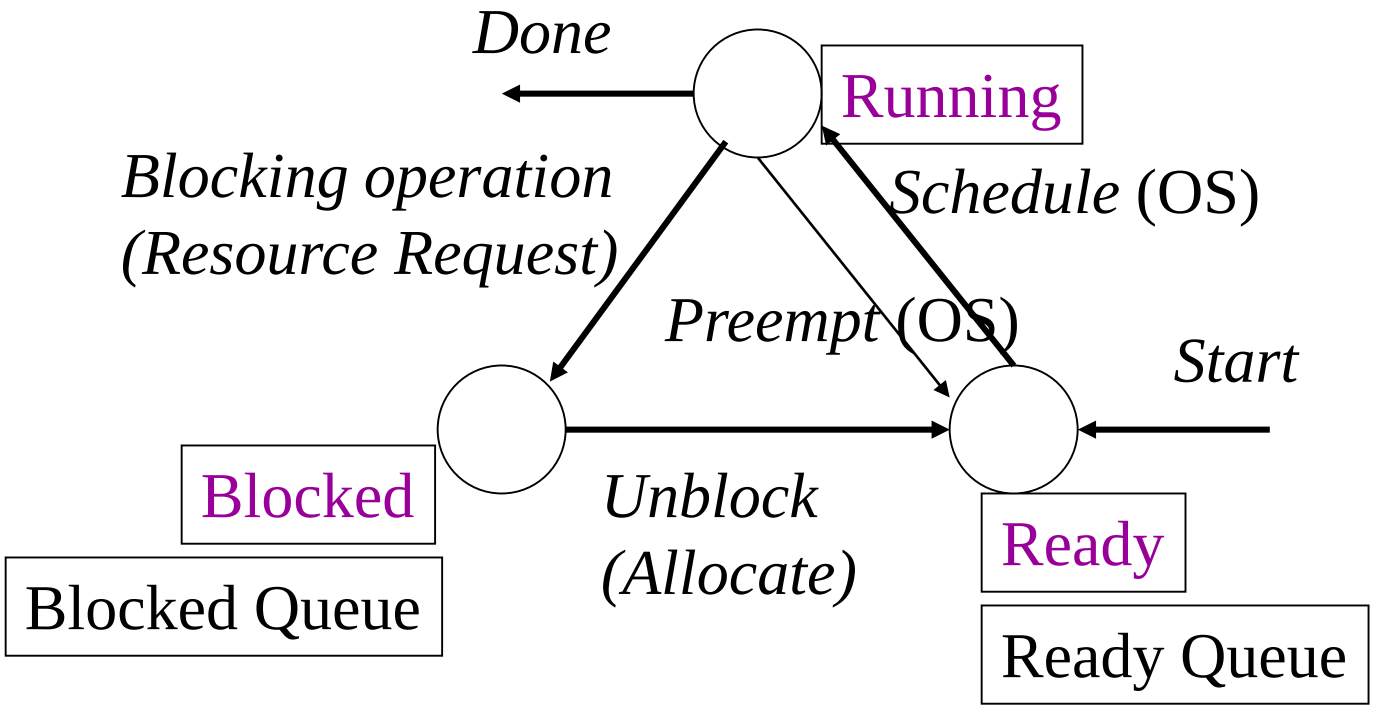
Process Manager functionalities

- 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

what defines the State of a Process

- Memory image: Code, data, heap, stack
- Process state, e.g. ready, running, or waiting
- Accounting info, e.g. process ID, privilige
- Program counter (PC) 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

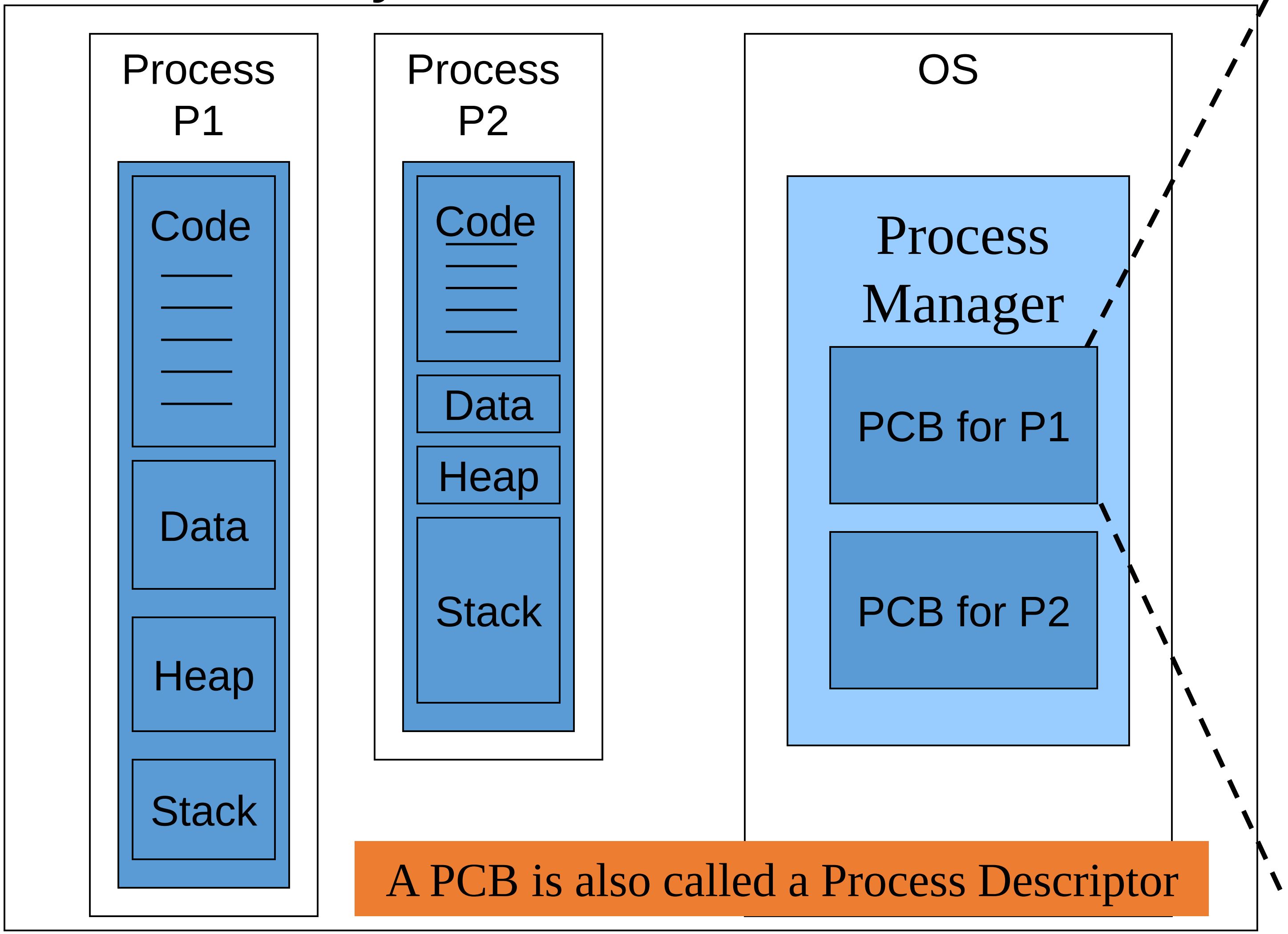
- OS maintains a PCB table containing one entry for every process in the system.

- 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

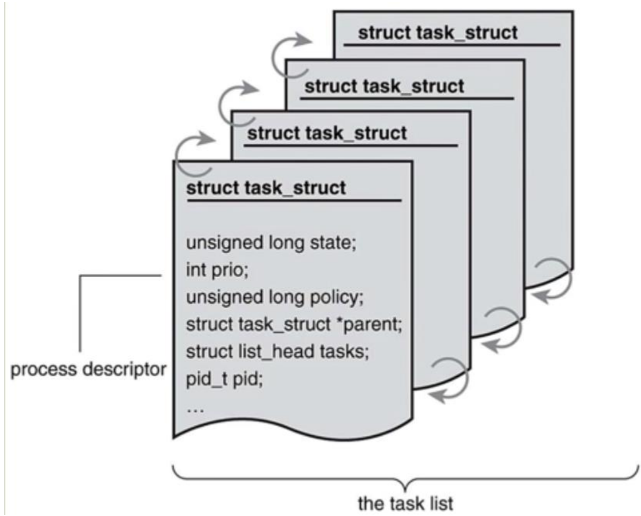


- Process state, e.g. ready, running, or waiting
- Accounting info, e.g. process ID
- Program Counter
- CPU registers
- CPU-scheduling info, e.g. priority
- Memory management info, e.g. base and limit registers, page tables
- I/O status info, e.g. list of open files

Process Descriptor

- Process – dynamic, program in motion
  - Kernel data structures to maintain "state"
  - Descriptor, task_struct
  - Complex struct with pointers to others

- Type of info in task_struct
  - state,
  - id,
  - priorities,
  - locks,
  - files,
  - signals,
  - memory maps,
  - queues, list pointers, …

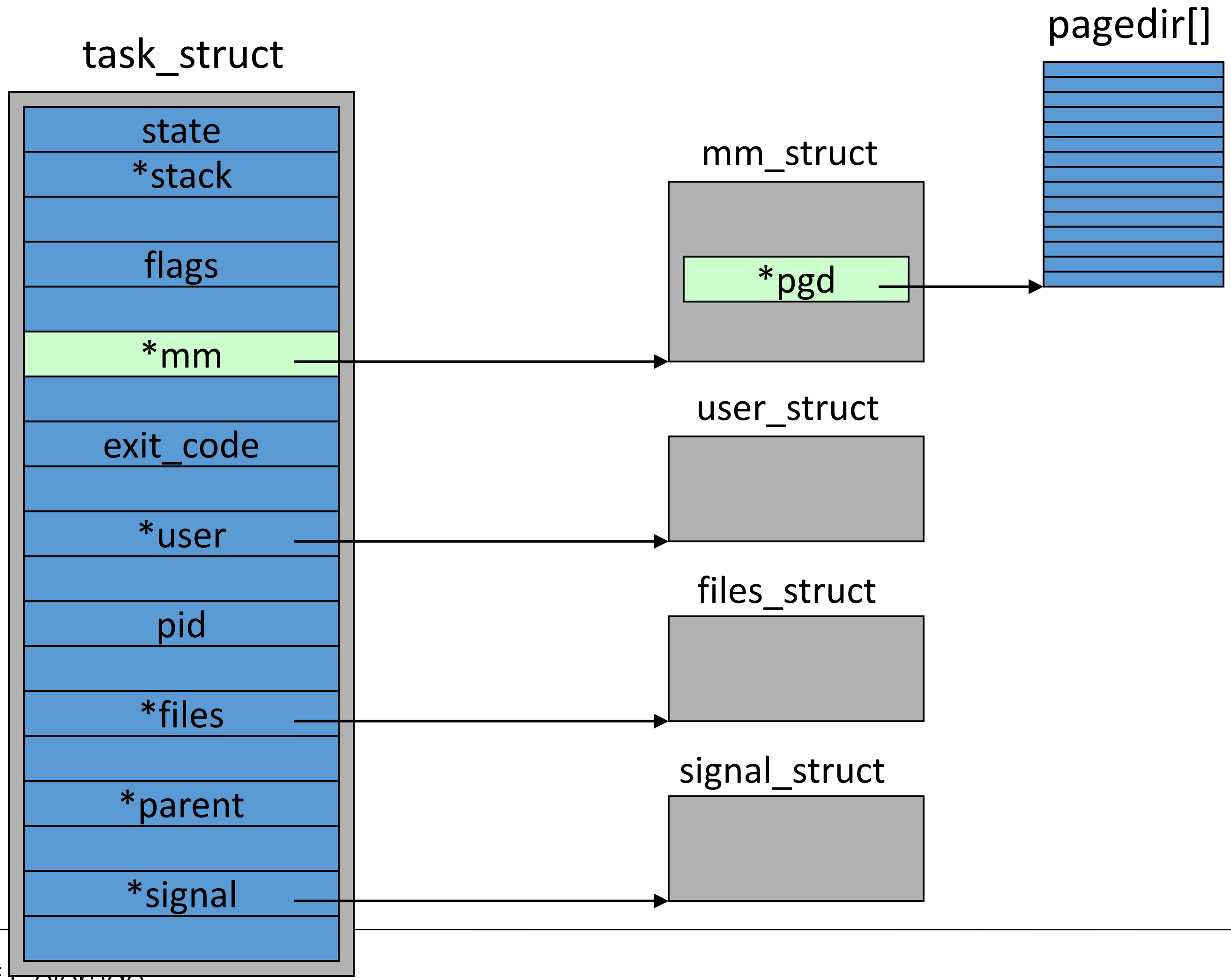


The Linux process descriptor

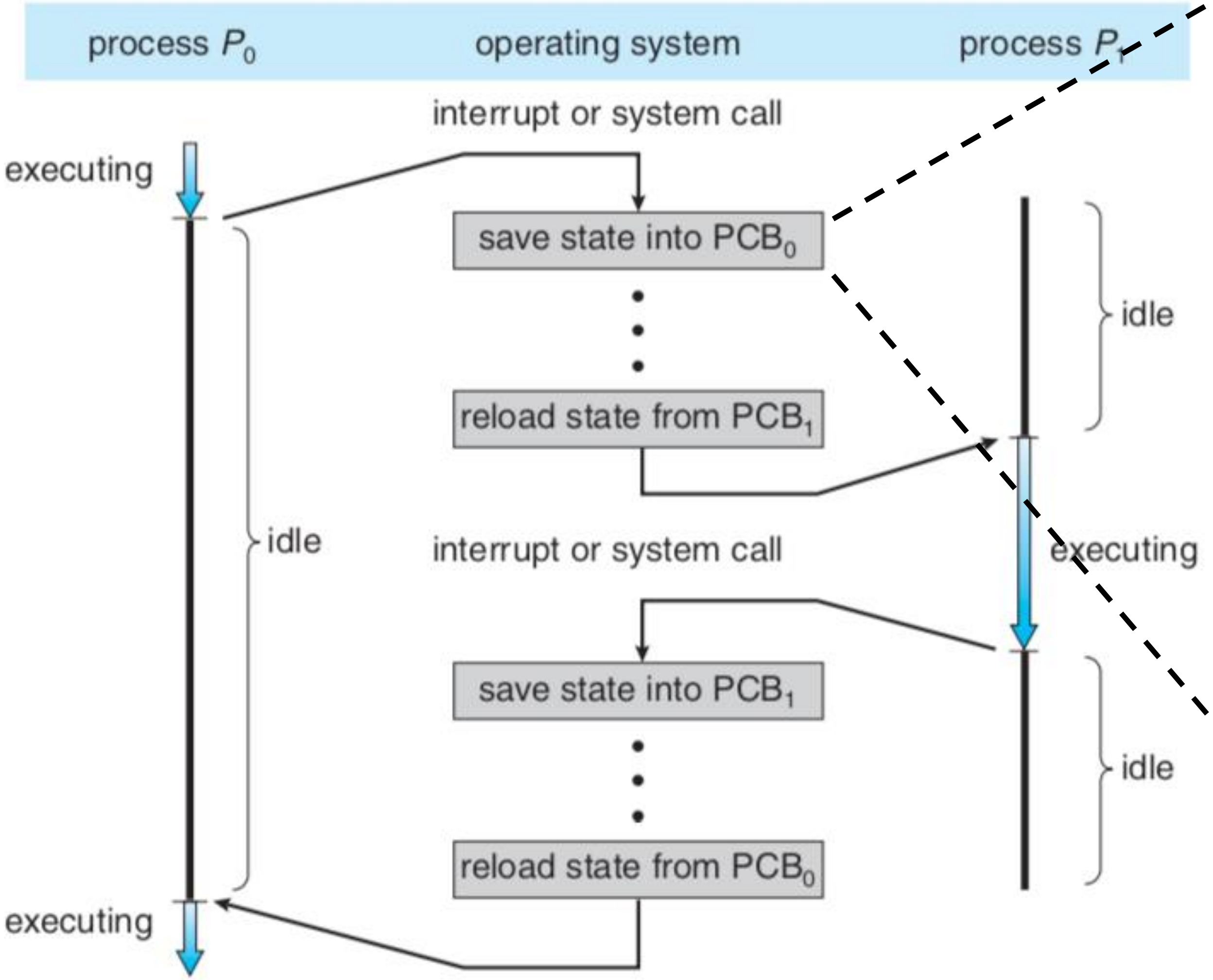
Each process descriptor contains many fields

and some are pointers to other kernel structures

which may themselves include fields that point to structures



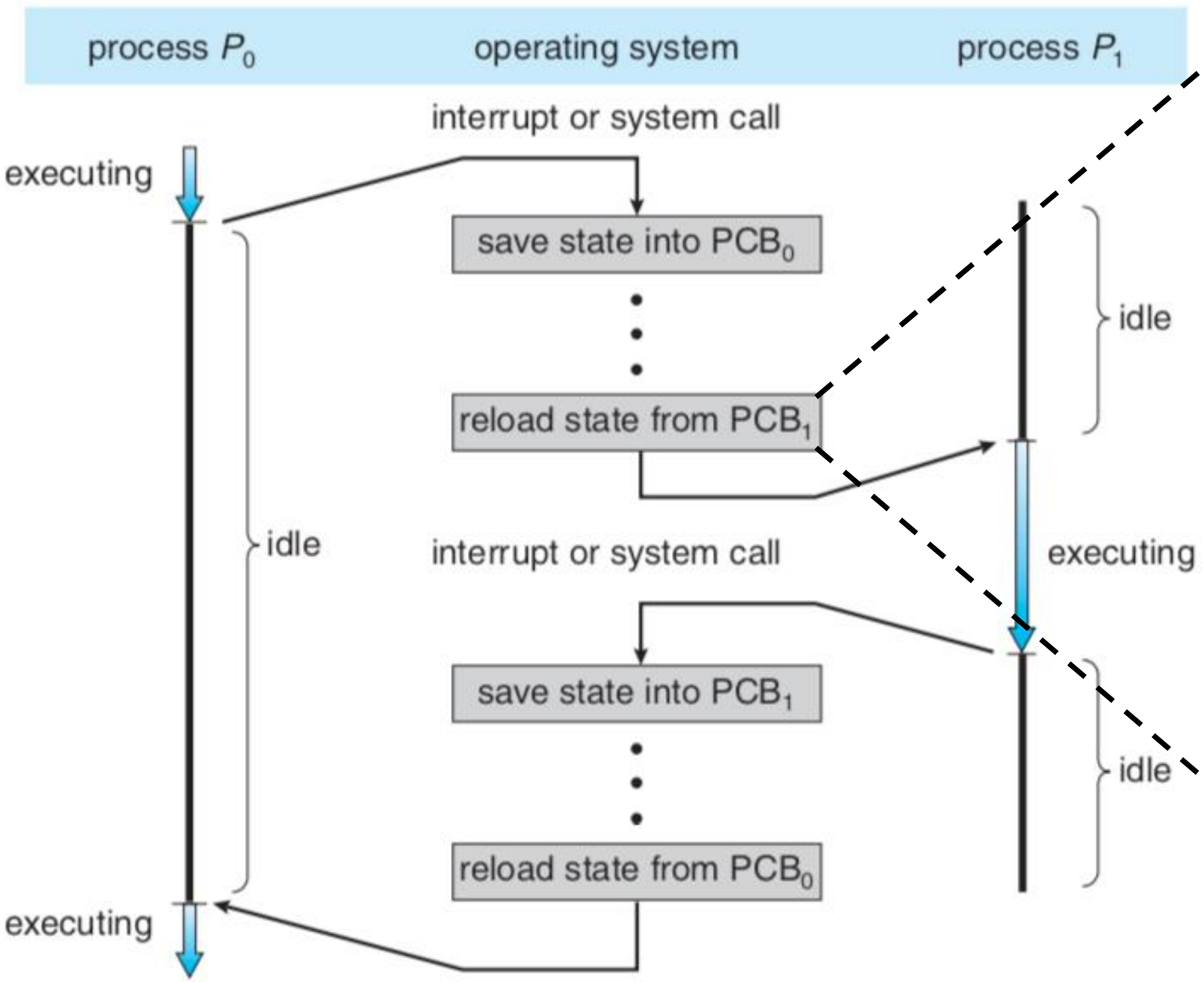
Context switch from one processes to another



- 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

Context switch from one processes to another

(102



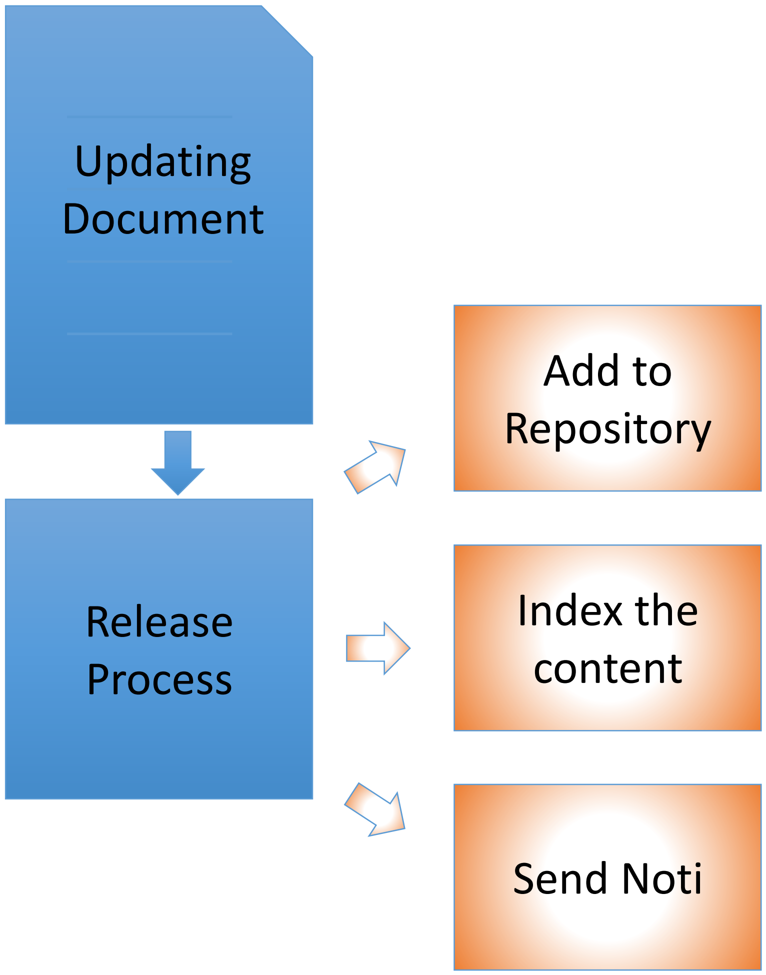
- Process state, e.g. ready, running, or waiting
- Accounting info, e.g. process ID
- Program Counter
- CPU registers
- CPU-scheduling info, e.g. priority
- Memory management info, e.g. base and limit registers, page tables
- I/O status info, e.g. list of open files

Process 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

Process pipeline Example

- Example: When updated document is being released (Dropbox)
  - Place into the repository
  - Add to context retrieval system
  - Send notification all copies



Creating Processes in Windows

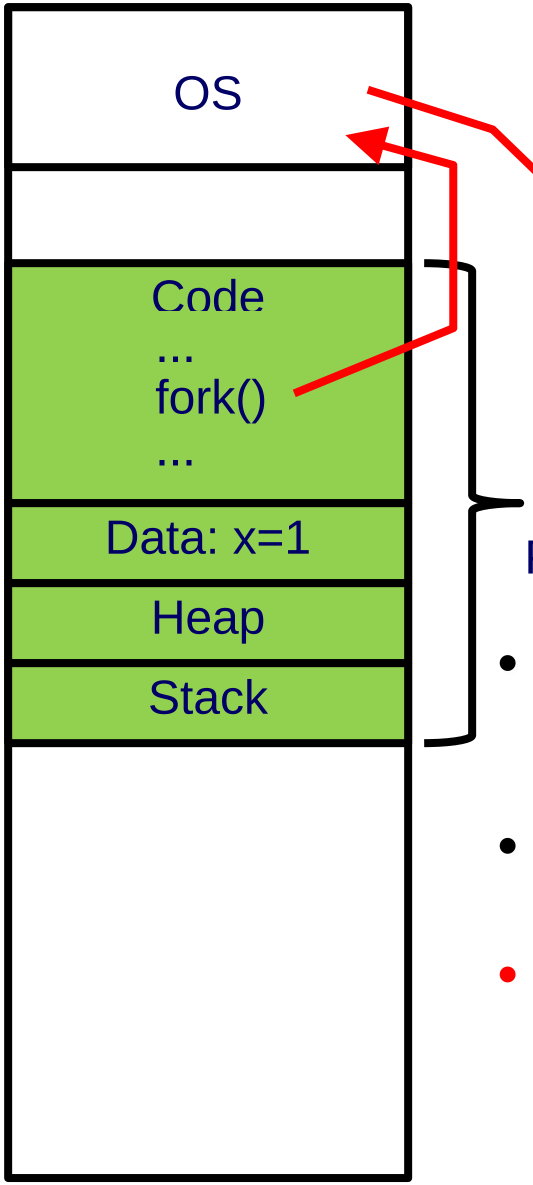
- CreateProcess() call, in Windows
  - Pass an argument to *CreateProcess()* indicating which program to start running
  - Invokes a system call to OS that then invokes process manager to:
    - allocate space in memory for the process
    - Set up PCB state for process, assigns PID, etc.
    - Copy code of program name from hard disk to main memory, sets PC to entry point in *main()*
    - Schedule the process for execution
  - Combines *fork()* and *exec*() system calls in UNIX/Linux 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 code and data are exactly the same
    - All addresses are appropriately mapped – more details on this later during memory management
  - The child starts executing at the same point as the parent, namely just after returning from the *fork()* call

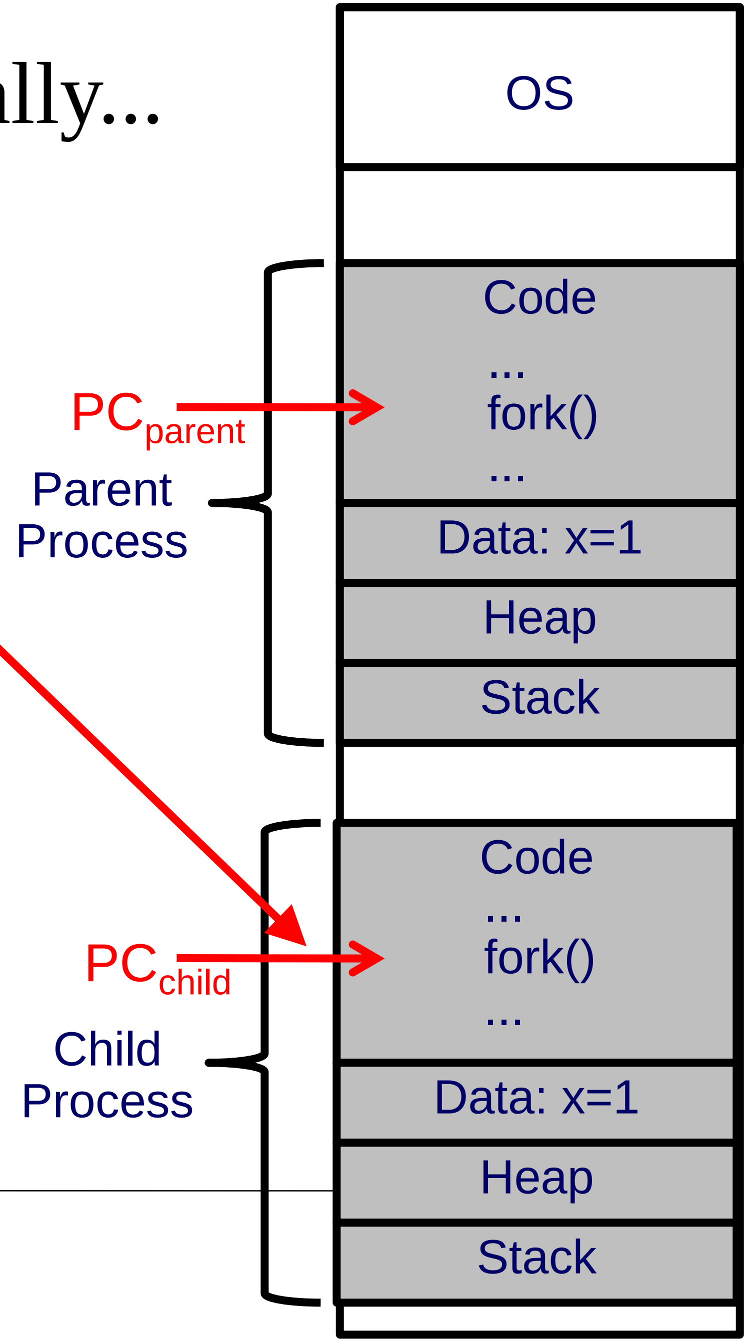
Fork() conceptually...

Memory (before fork)



- Fork() duplicates address space of parent in the child
- Both execute **concurrently**
- How does a process know if it is the parent or the child?

Memory (after fork)



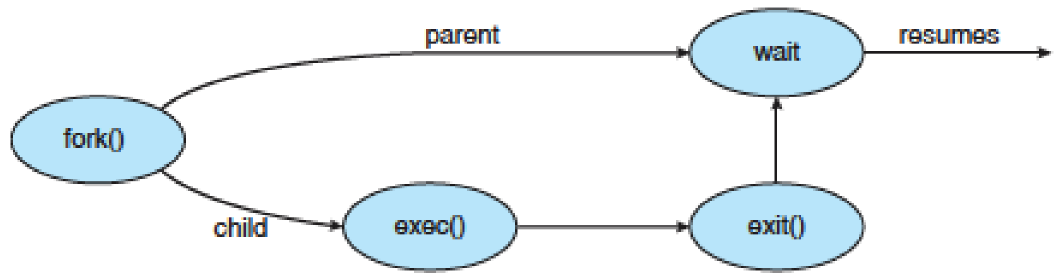
fork ()

- 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 new program code into the calling process' s memory

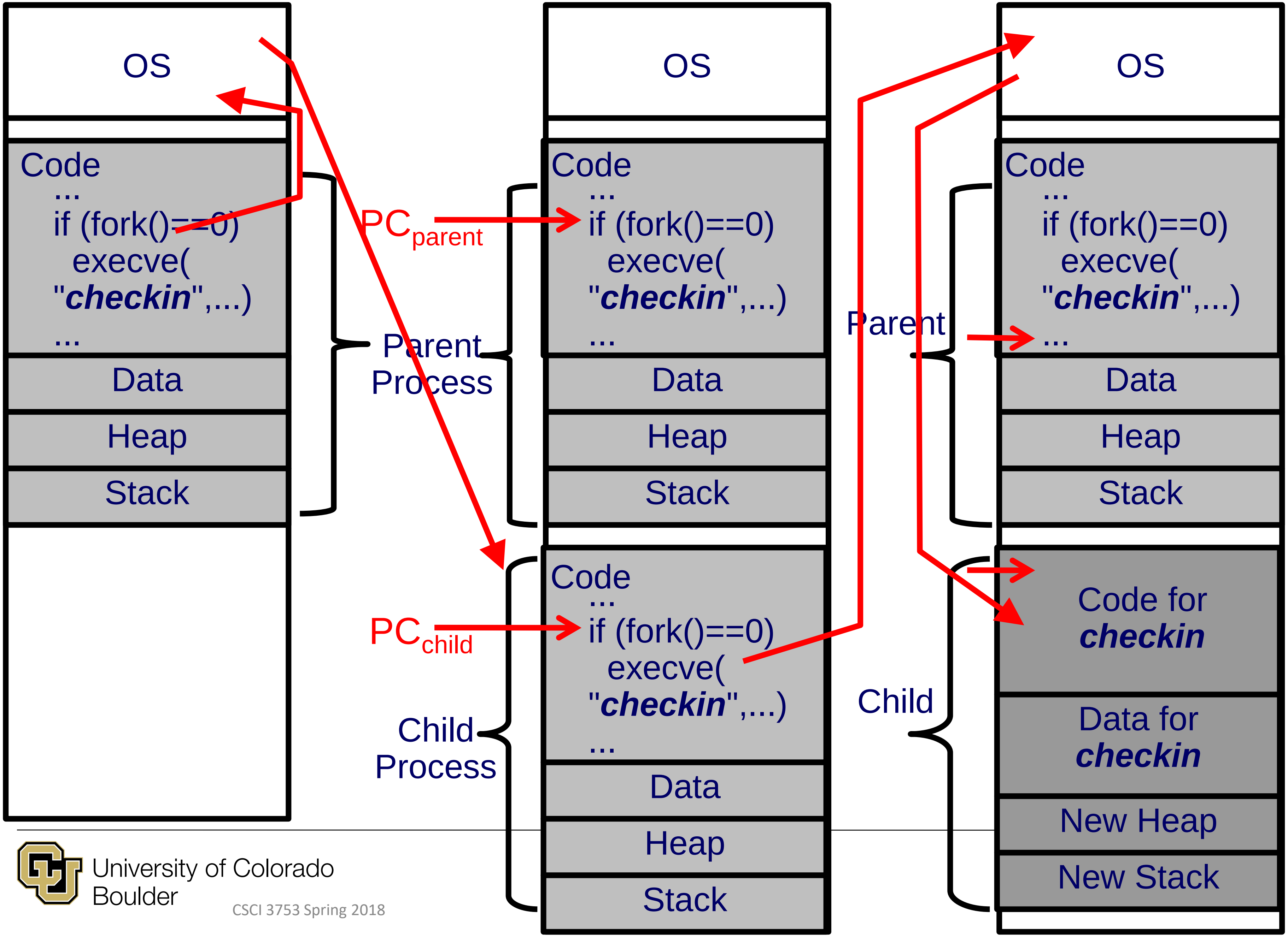


The calling code is erased!

- Use *fork()* and *exec()* (actually *execve()*)
- to create a new process executing a new program in a new address space

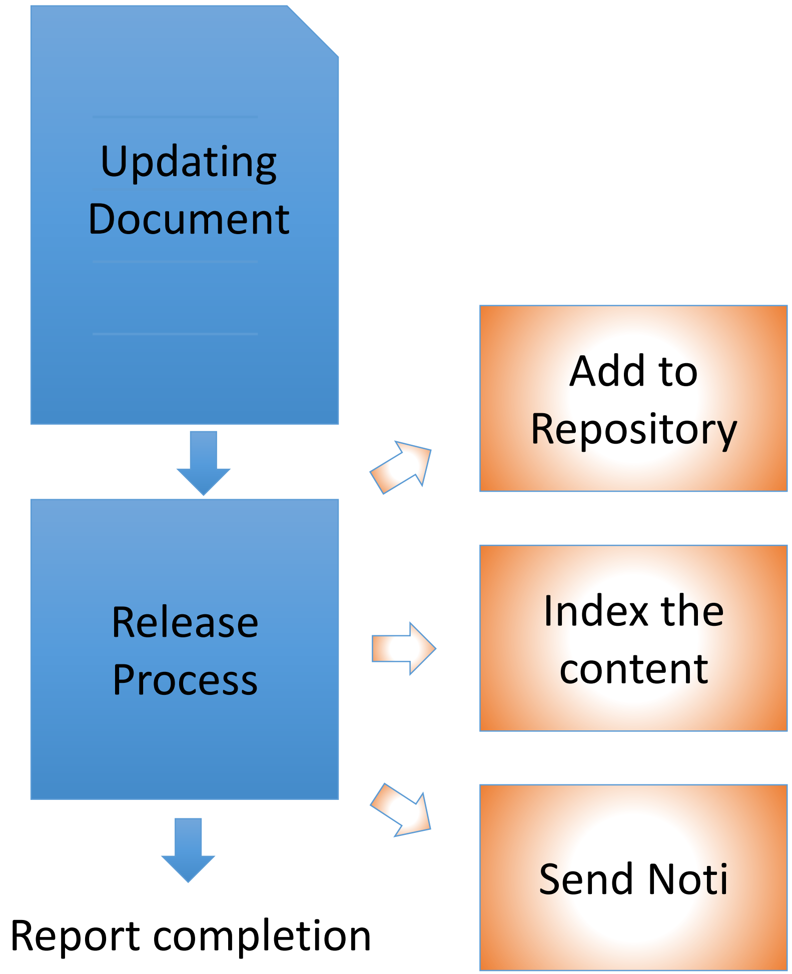
```
PID = fork();
if (PID==0) { /* child */
   exec("checkin");
   exit(0);
}

/* the parent's code here */
```

Memory (before fork) Memory (after fork) Memory(after execve)



Process pipeline Example

- Example: When updated document is being released (Dropbox)
  - Place into the repository
  - Add to context retrieval system
  - Send notification all copies



Wait for each process to complete before starting the next process

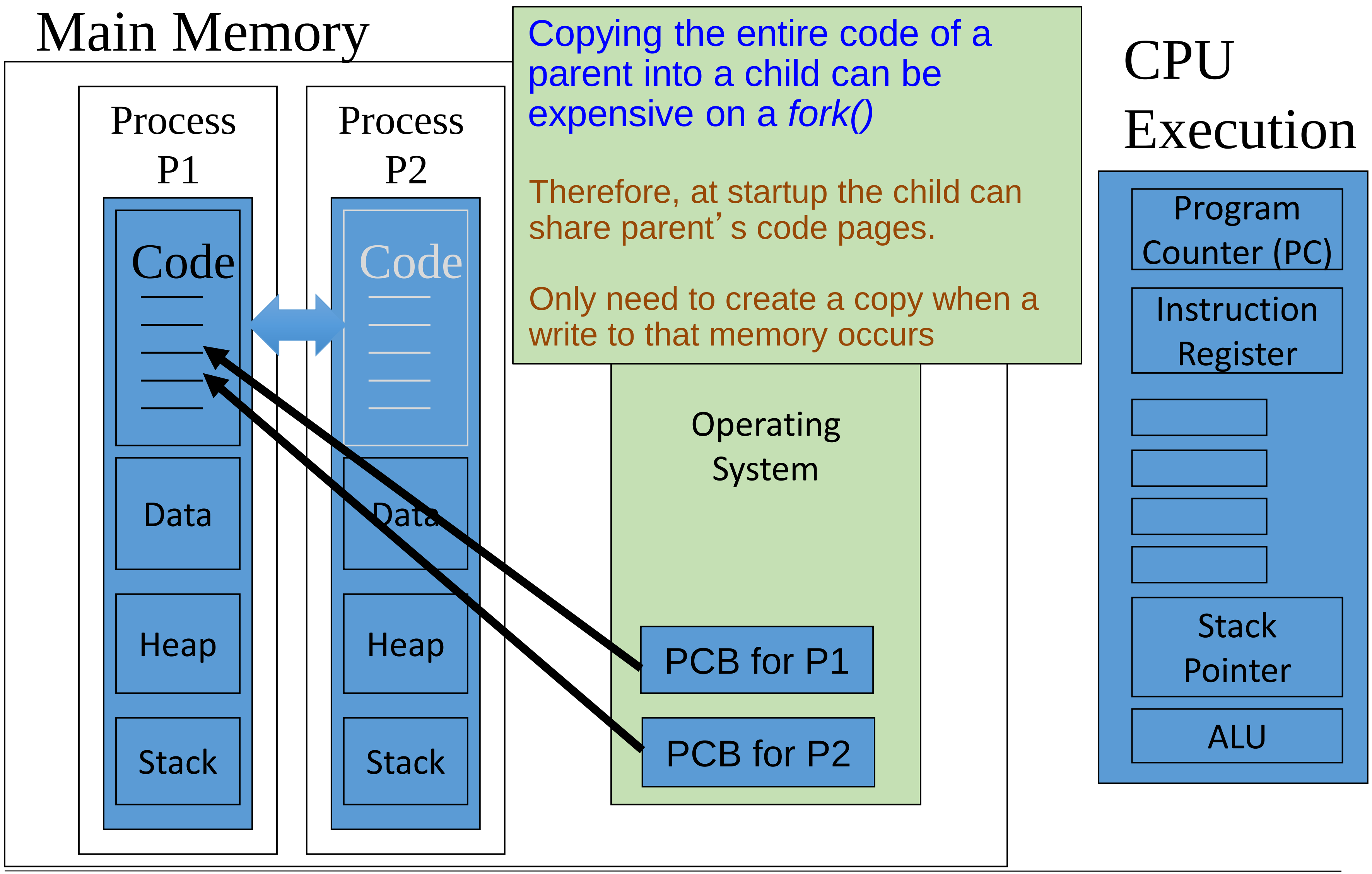
Waiting on Processes

- 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

```
PID = fork();
if (PID==0) { /* child */
   exec("checkin");
   exit(0);
}

/* the parent's code here */
```

```
/* the parent's code here */
child_pid = wait();
/* child has completed */
. . .
```

Multiple Processes + OS



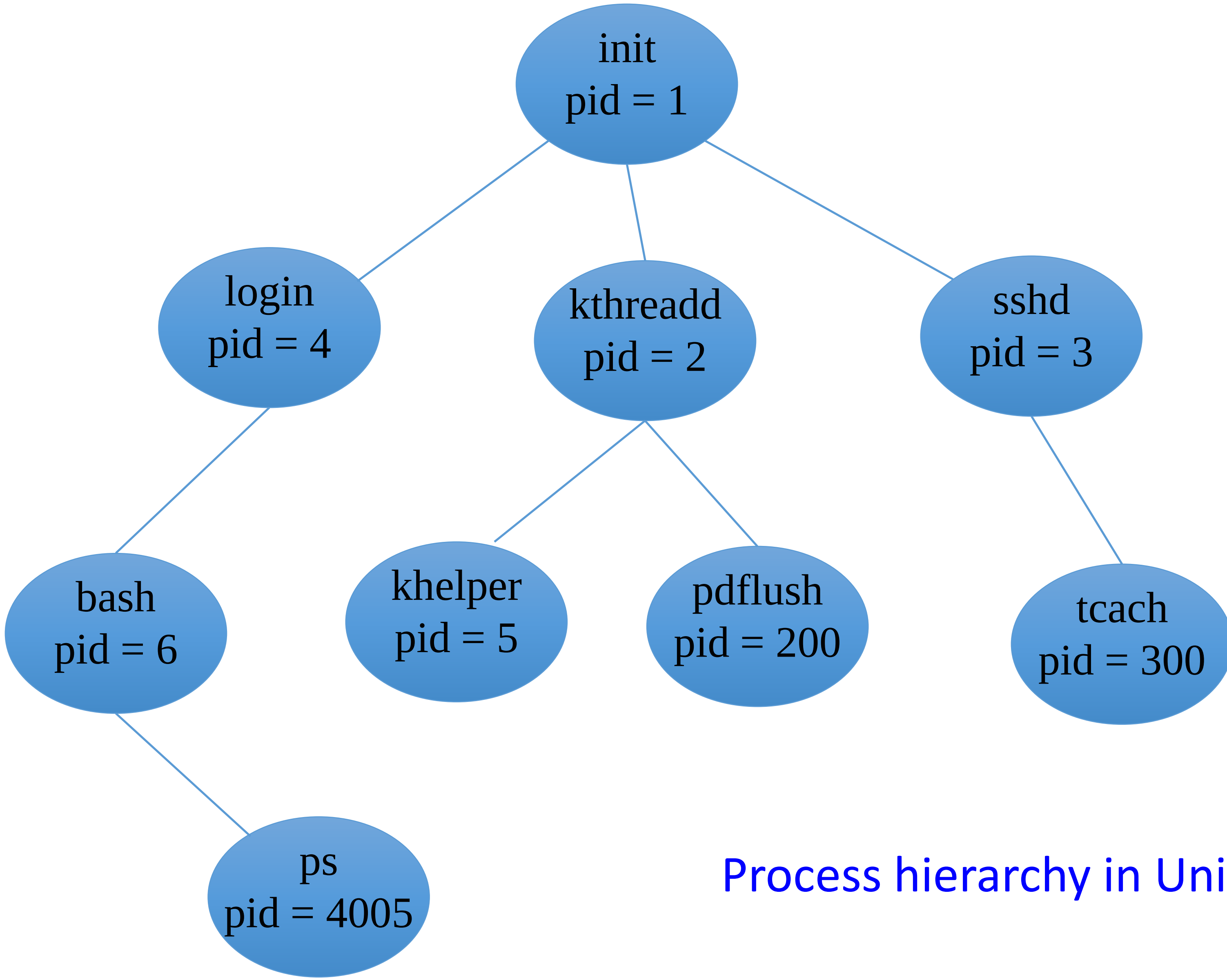
Process Hierarchy

- OS creates a single process at the start up

  init

- 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.

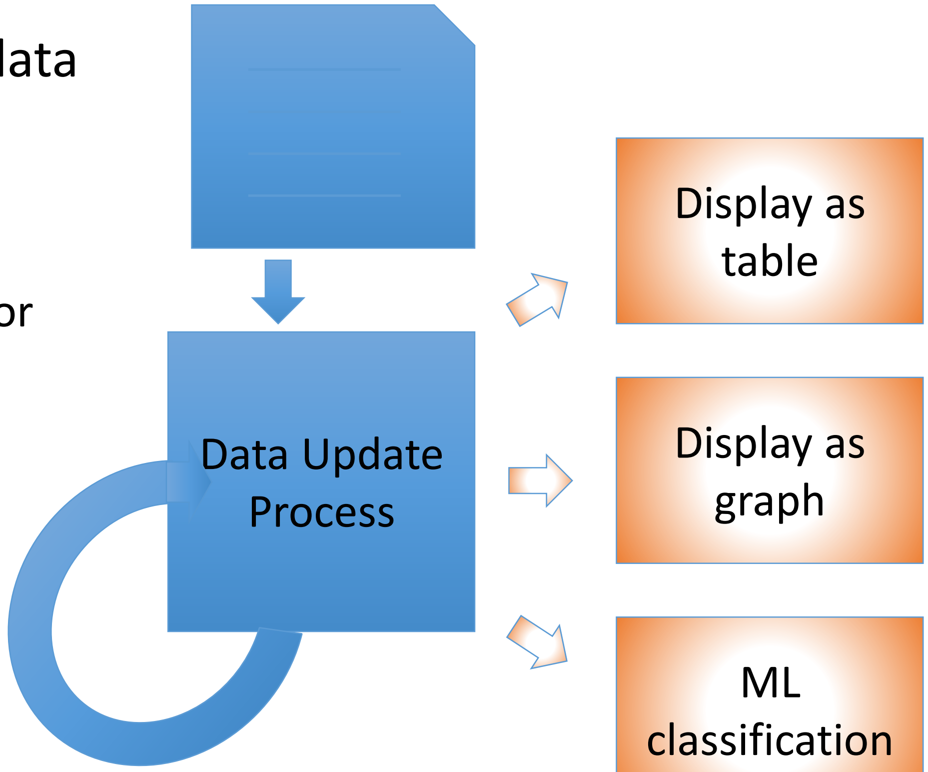
- A tree-structured hierarchy of processes



Process hierarchy in Unix

Another Process Example

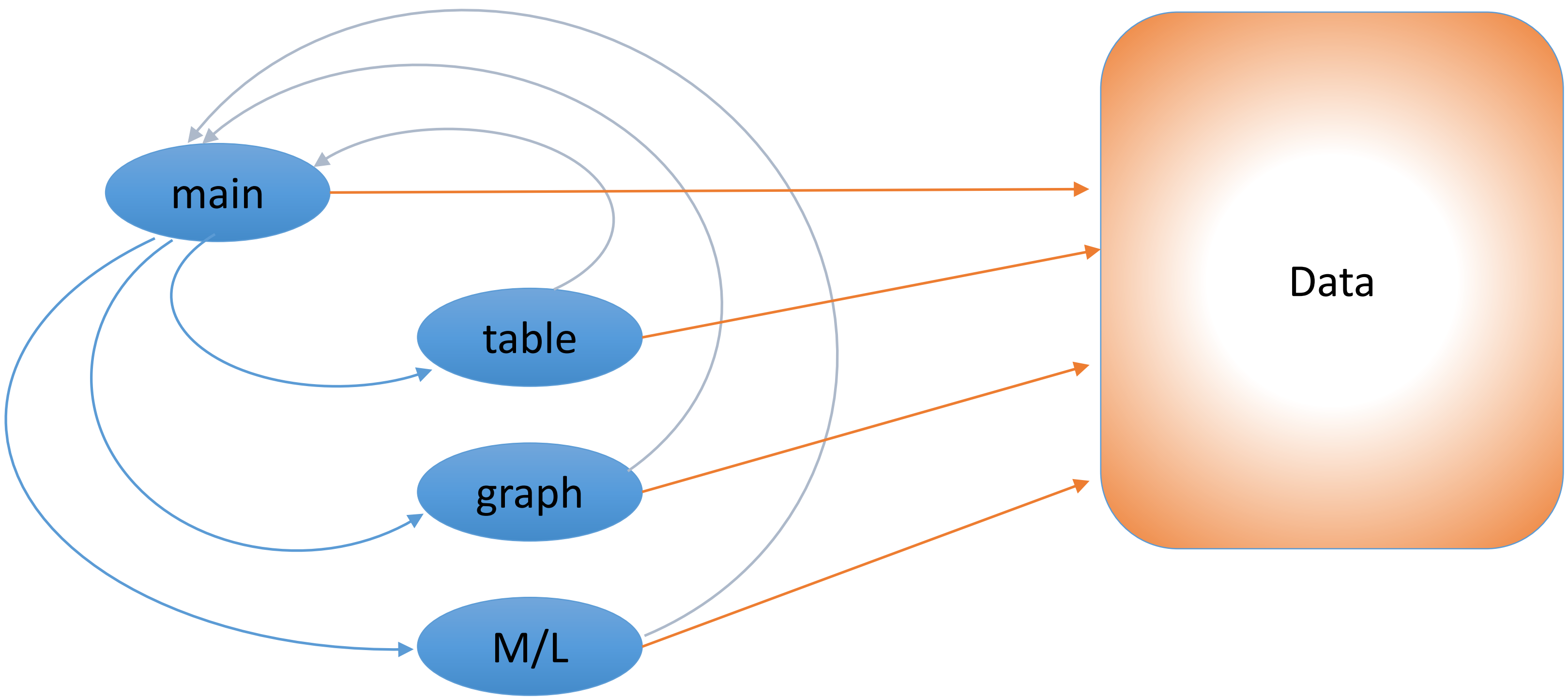
- When a new / updated data is available
  - Display as table
  - Display as graph
  - Perform a ML algorithm for classification



Repeat every time the data is updated

All processes are independent, but need to share the same data

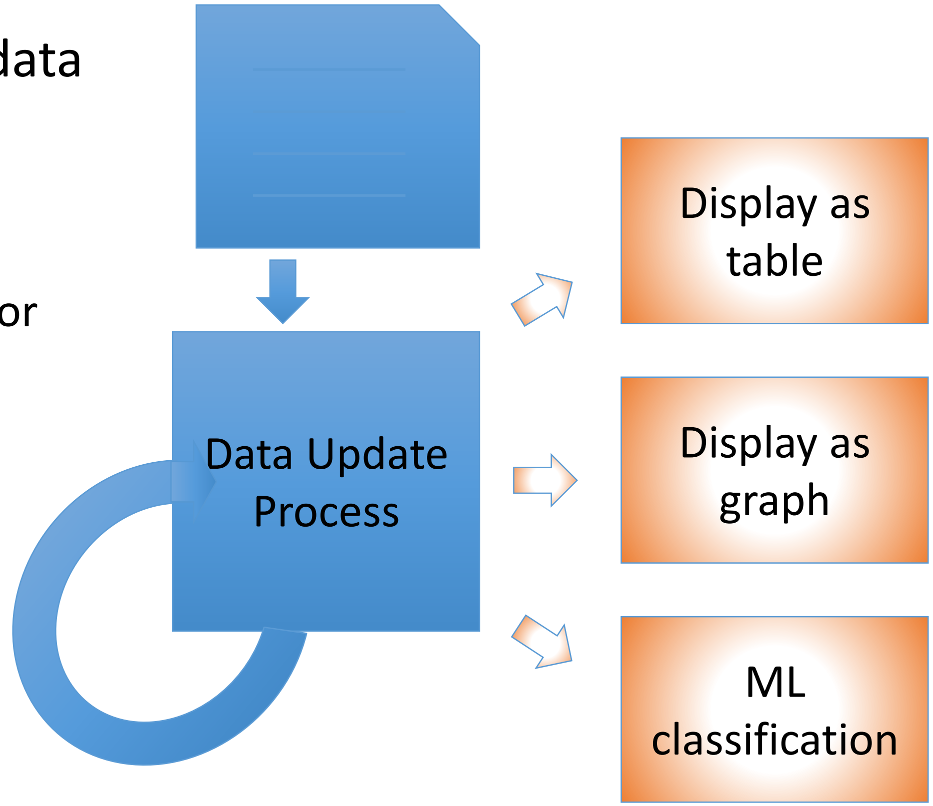
Parallel Pipeline



- All processes need to access the same data

Another Process Example

- When a new / updated data is available
  - Display as table
  - Display as graph
  - Perform a ML algorithm for classification



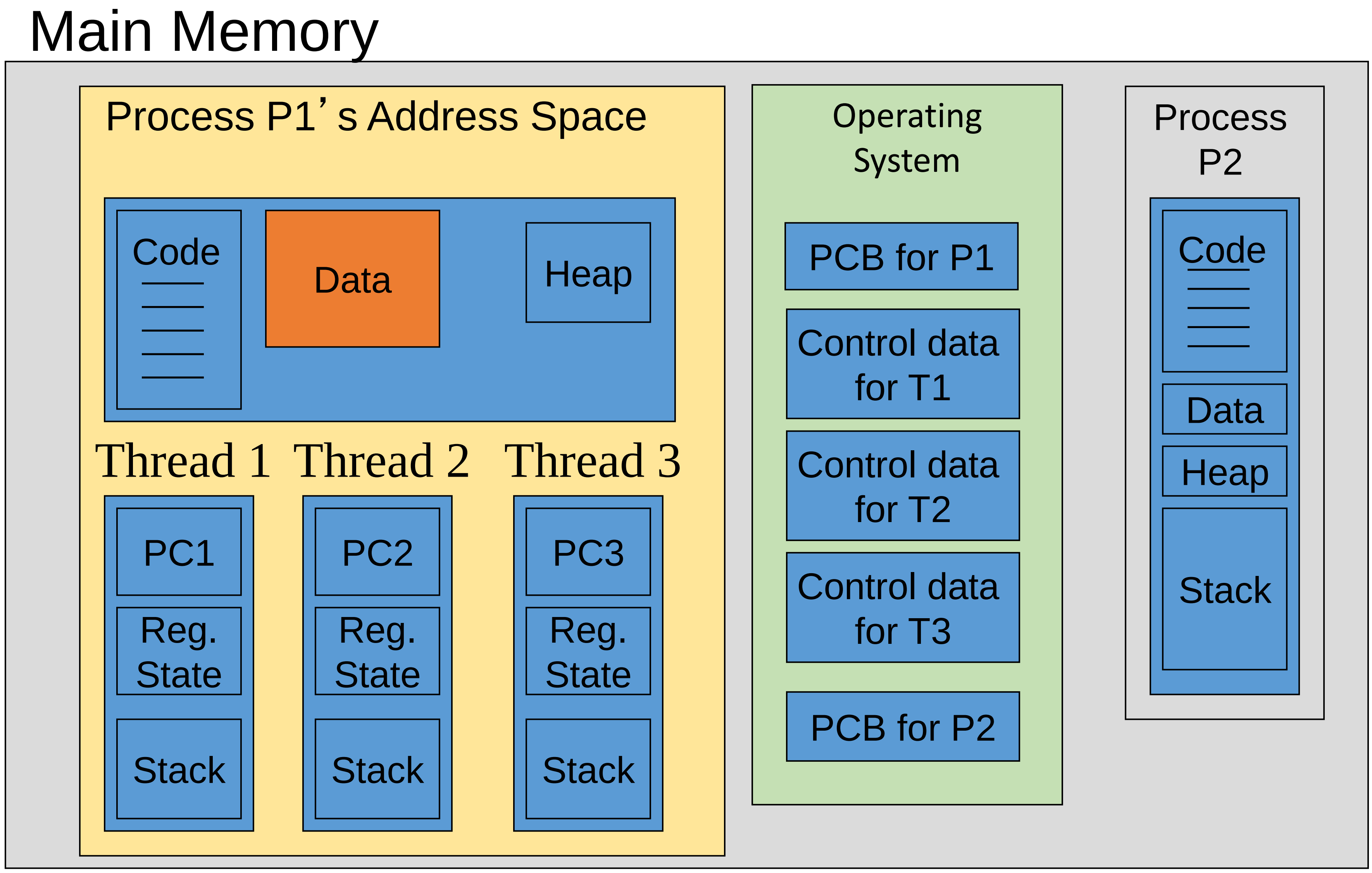
Repeat every time the data is updated

All processes are independent, but need to share the same data

Threads

- A thread is a logical flow or unit of execution that runs within the context of a process
  - has its own program counter (PC), register state, and stack
  - shares the memory address space with other threads in the same process,
    - share the same code and data and resources (e.g. open files)
  - A thread is also called a *lightweight process*
    - *Low overhead compared to a separate process*

Multiple Threads



Why do we want to use Threads

- Reduced context switch overhead vs multiple processes
  - E.g. In Solaris, context switching between processes is 5x slower than switching between threads
  - Don't have to save/restore context, including base and limit registers and other MMU registers, also TLB cache doesn't have to be flushed
- Shared resources => less memory consumption
  - Don't duplicate code, data or heap or have multiple PCBs as for multiple processes
  - Supports more threads – more scalable, e.g. Web server must handle thousands of connections

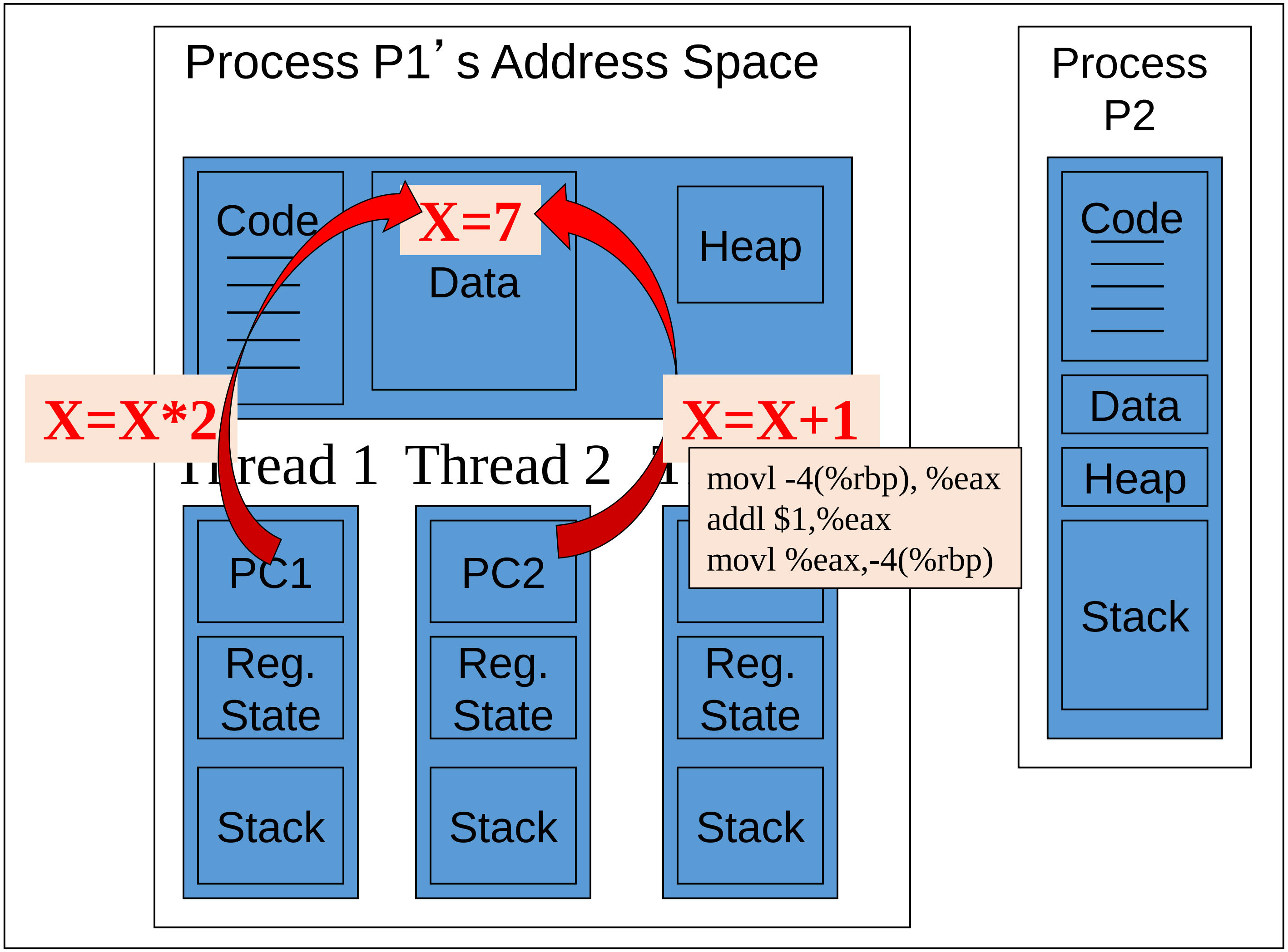
More reasons for

Why do we want to use Threads

- Inter-thread communication is easier and faster than inter-process communication
  - threads share the same memory space, so just read/write from/to the same memory location ! ! !
  - IPC via message passing uses system calls to send/receive a message, which is slow
    - IPC using shared memory may be comparable to inter-thread communication

Thread Safety

Main Memory



Suppose:

- Thread1 wants to multiply X by 2
- Thread2 wants to increment X
- *Could have a race condition (see Chapter 5)*

Thread-Safe Code

- A piece of code is **thread-safe** if it functions correctly during simultaneous or *concurrent* execution by multiple threads.
  - In particular, it must satisfy the need for multiple threads to access the same shared data, and the need for a shared piece of data to be accessed by only one thread at any given time.

- If two threads share and execute the same code, then unprotected use of shared
  - global variables is not thread safe
  - static variables is not thread safe
  - heap variables is not thread safe

We will learn how to write thread-safe code in Chapter 5

Processes vs. Threads

- Why are processes still used when threads bring so many advantages?
  1. Some tasks are sequential and not easily parallelizable, and hence are single-threaded by nature
  2. No fault isolation between threads
     - If a thread crashes, it can bring down other threads
     - If a process crashes, other processes continue to execute, because each process operates within its own address space, and so one crashing has limited effect on another
       - Caveat: a crashed process may fail to release synchronization locks, open files, etc., thus affecting other processes . But, the OS can use PCB's information to help cleanly recover from a crash and free up resources.

Processes vs. Threads (2)

- Why are processes still used when threads bring so many advantages? (cont.)

    3. Writing thread-safe/reentrant code is difficult. Processes can avoid this by having separate address spaces and separate copies of the data and heap

Threads vs. Processes

- Advantages of multithreading
  - Sharing between threads is easy
  - Faster creation
- Disadvantages of multithreading
  - Ensure threads-safety
  - Bug in one thread can bleed to other threads, since they share the same address space
  - Threads must compete for memory
- Considerations
  - Dealing with signals in threads is tricky
  - All threads must run the same program
  - Sharing of files, users, etc

Applications, Processes, and Threads

- An application can consist of multiple processes, each one dedicated to a specific task (UI, computation, communication, etc.)

- Each process can consists of multiple threads

- An application could thus consist of many processes and threads

Thread Libraries

- Thread library provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

- Three main thread libraries in use today:
  - POSIX Pthreads
  - Win32
  - Java

Thread Libraries

- Three main thread libraries in use today:
  - **POSIX pthreads**
    - May be provided either as user-level or kernel-level
    - A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
    - API specifies behavior of the thread library, implementation is up to development of the library
  - **Win32**
    - Kernel-level library on Windows system
  - **Java**
    - Java threads are managed by the JVM
    - Typically implemented using the threads model provided by underlying OS

The **pthreads** API

- ***Thread management:*** The first class of functions work directly on threads - creating, terminating, joining, etc.
- ***Semaphores:*** provide for create, destroy, wait, and post on semaphores.
- ***Mutexes:*** provide for creating, destroying, locking and unlocking mutexes.
- ***Condition variables:*** include functions to create, destroy, wait and signal based upon specified variable values.

Thread Creation

pthread_create (tid, attr, start_routine, arg)

- It returns the new thread ID via the *tid* argument.
- The *attr* parameter is used to set thread attributes, NULL for the default values.
- The *start_routine* is the C routine that the thread will execute once it is created.
- A single argument may be passed to *start_routine* via *arg*. It must be passed by reference as a pointer cast of type void.

Thread Termination and Join

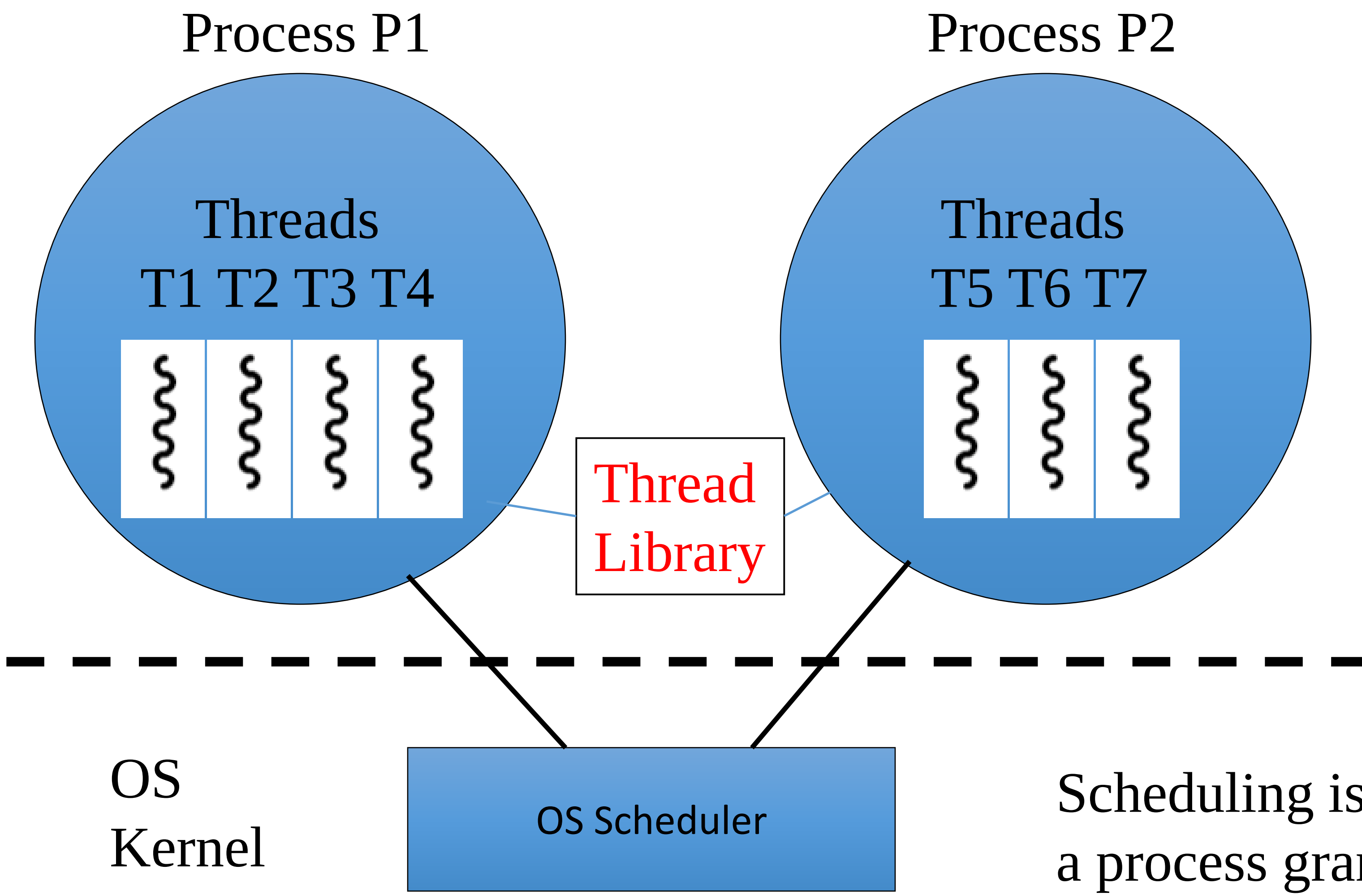
pthread_exit (value) ;

- This Function is used by a thread to terminate.
- The return value is passed as a pointer.

pthread_join (tid, value_ptr);

- The pthread_join() subroutine blocks the calling thread until the specified *threadid* thread terminates.
- Return 0 on success, and negative on failure. The returned value is a pointer returned by reference. If you do not care about the return value, you can pass NULL for the second argument.

User-Space Threads



Scheduling is on
a process granularity

User-Space Threads

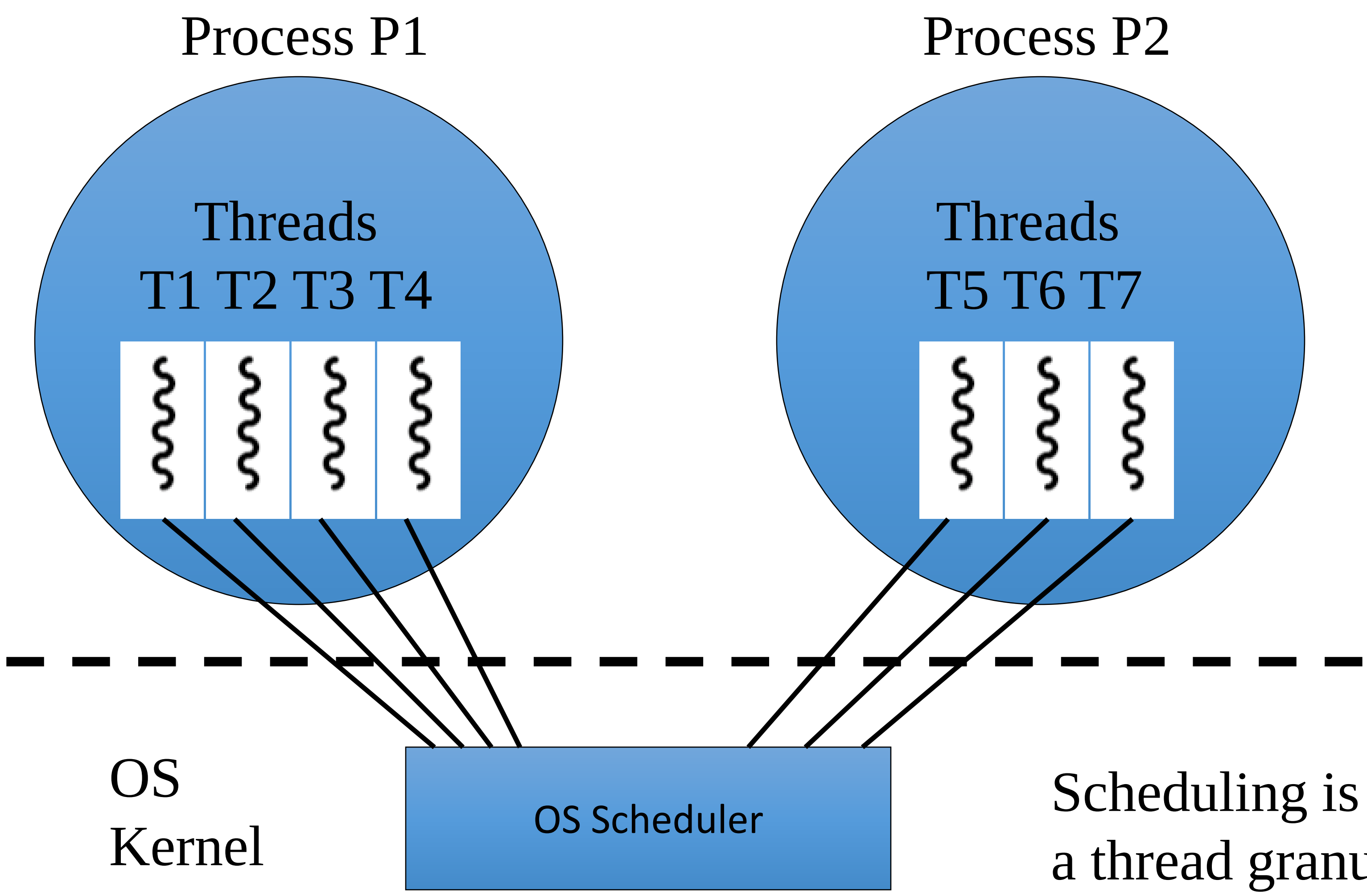
- *User space threads* are usually cooperatively multitasked, i.e. user threads within a process voluntarily give up the CPU to each other
  - threads will synchronize with each other via the user space threading package or library
  - Thread library: provides interface to create, delete threads in the same process

- OS is unaware of user-space threads – only sees user-space processes
  - If one user space thread blocks, the entire process blocks in a many-to-one scenario (see text)

- *pthreads* is a POSIX threading API
  - implementations of pthreads API differ underneath the API; could be user space threads; there is also pthreads support for kernel threads as well

- User space thread also called a *fiber*

Kernel Threads

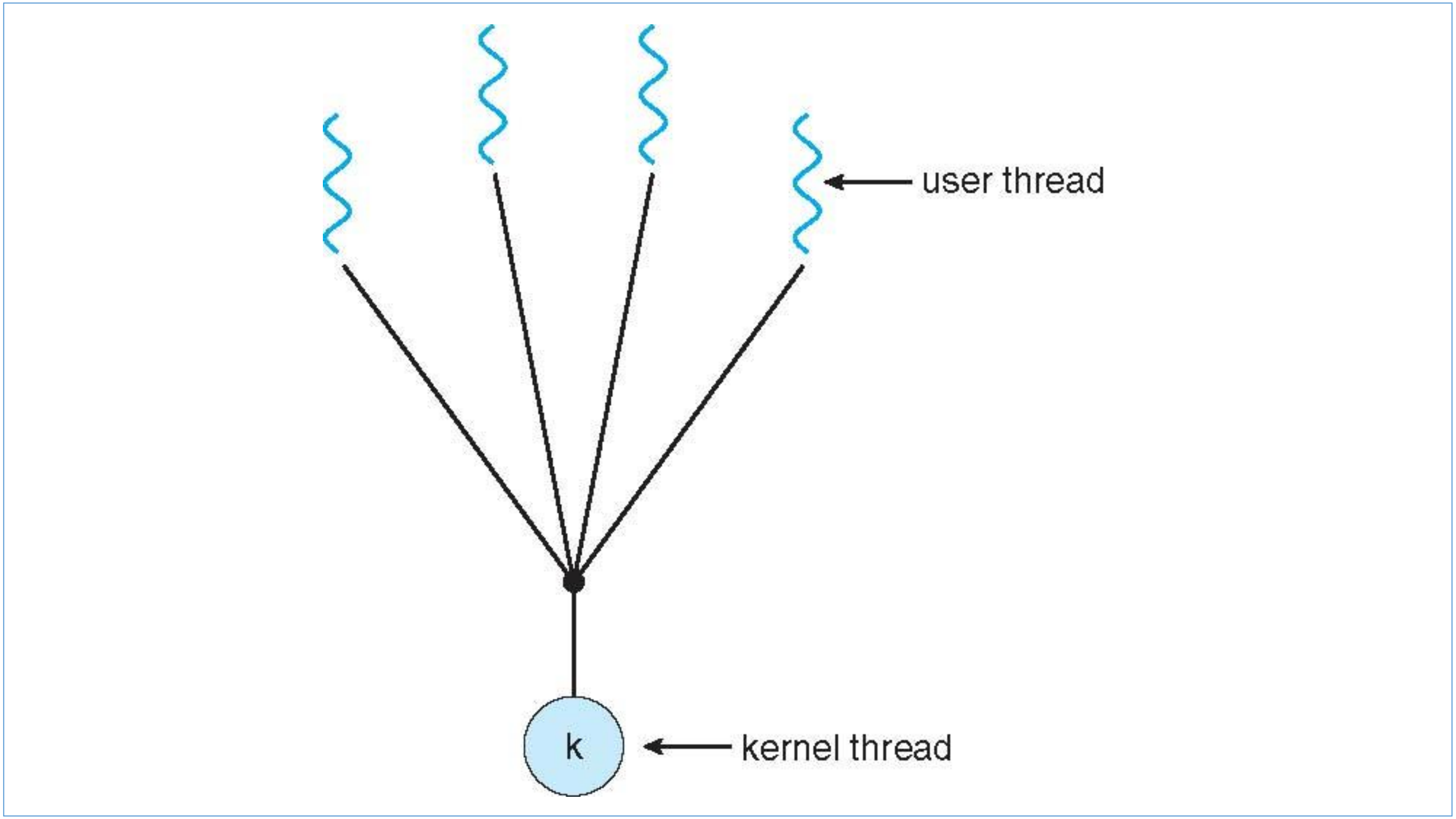


Scheduling is on a thread granularity

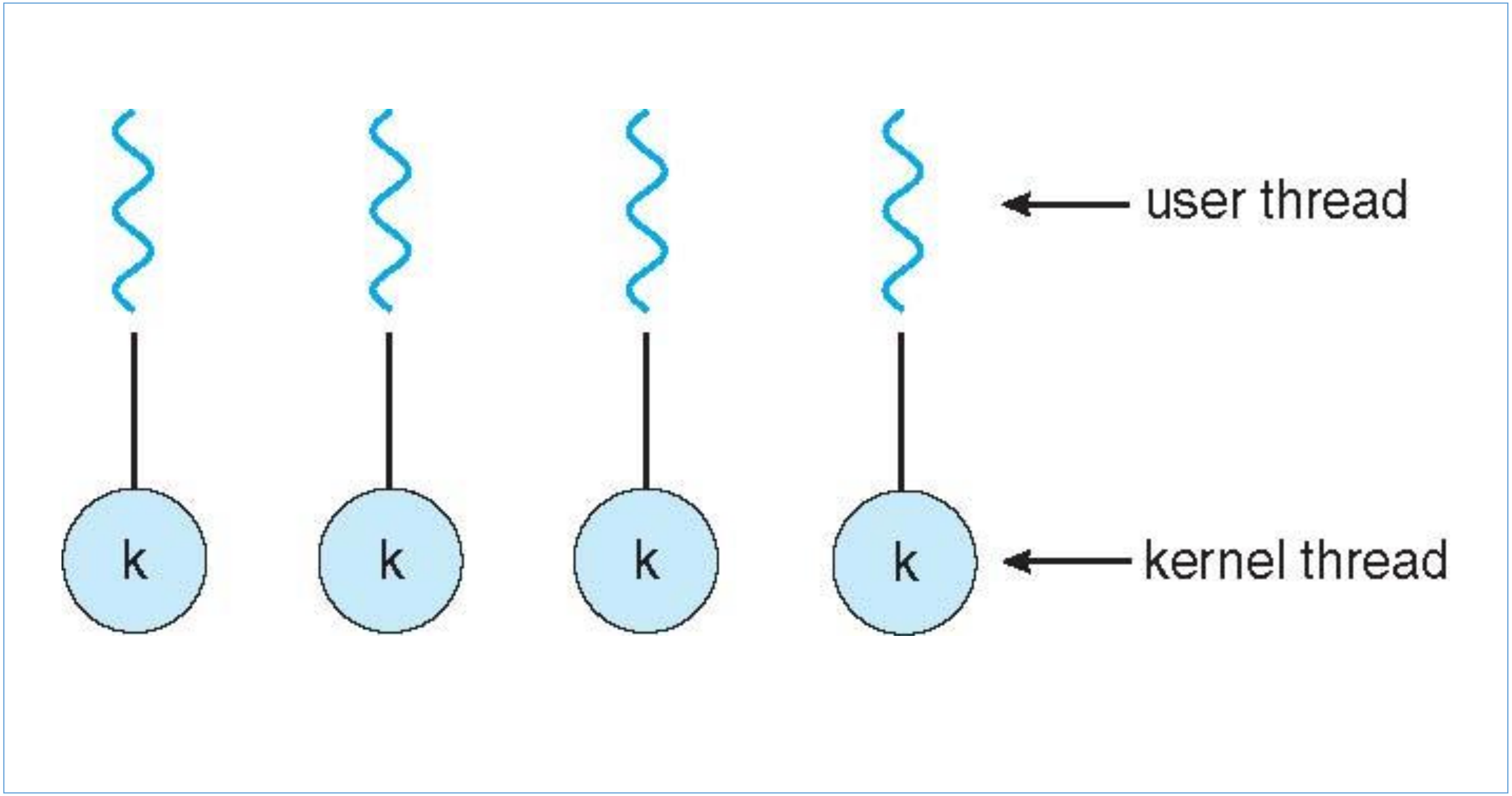
Kernel Threads

- *Kernel threads* are supported by the OS
  - kernel sees threads and schedules at the granularity of threads
  - Most modern OSs like Linux, Mac OS X, Win XP support kernel threads
  - Mapping of user-level threads to kernel threads is usually one-to-one, e.g. Linux and Windows, but could be many-to-one, or many-to-many
  - Win32 thread library is a kernel-level thread library

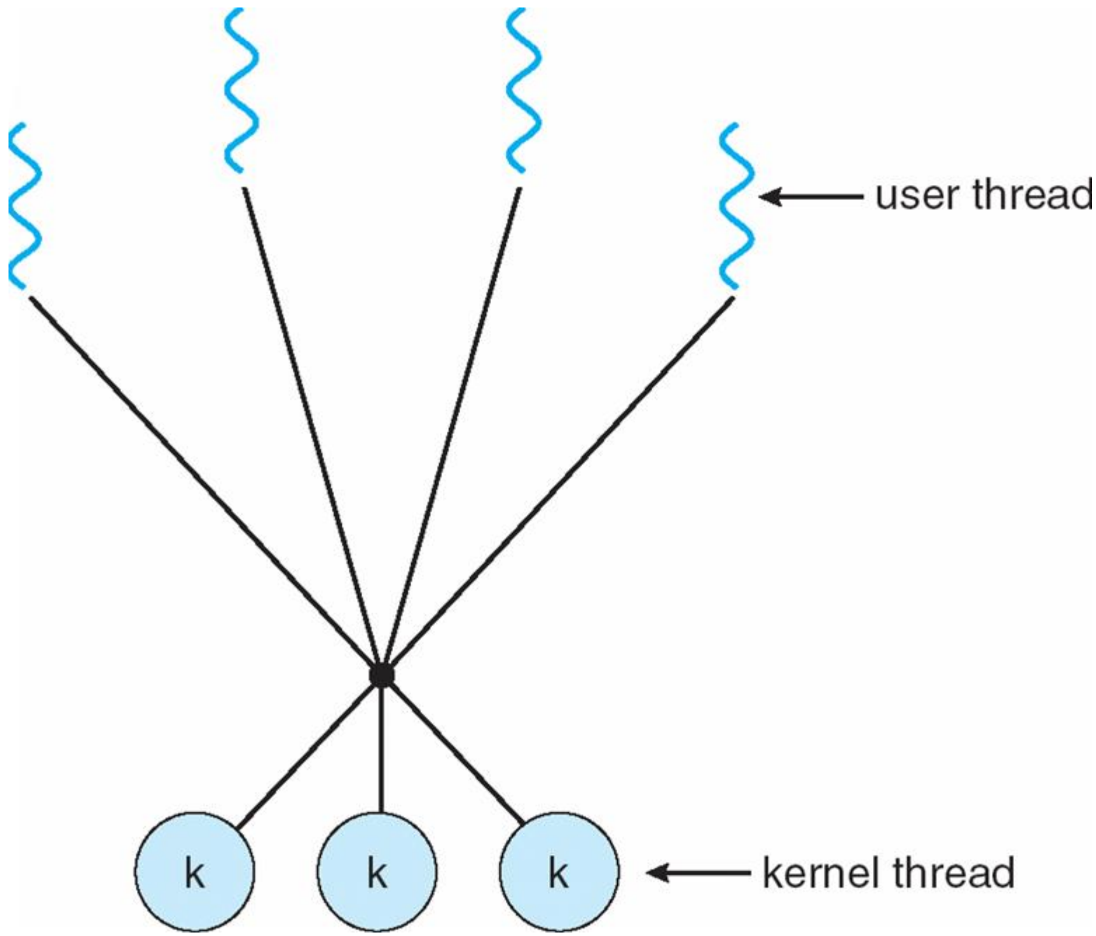
Many-to-One Model



One-to-one Model



Many-to-Many Model



Stop here – Below are backup slide

printk( ): Logging levels

- printk has an optional prefix string, Loglevel, that specifies the type of message being sent to the kernel message log

KERN_EMERG	Emergency condition, system is probably dead
KERN_ALERT	Some problem has occurred, immediate attention is needed
KERN_CRIT	A critical condition
KERN_ERR	An error has occurred
KERN_WARNING	A warning
KERN_NOTICE	Normal message to take note of
KERN_INFO	Some information
KERN_DEBUG	Debug information related to the program