Chapters 3 & 4: Process Management, Threads

CSCI 3753 Operating Systems

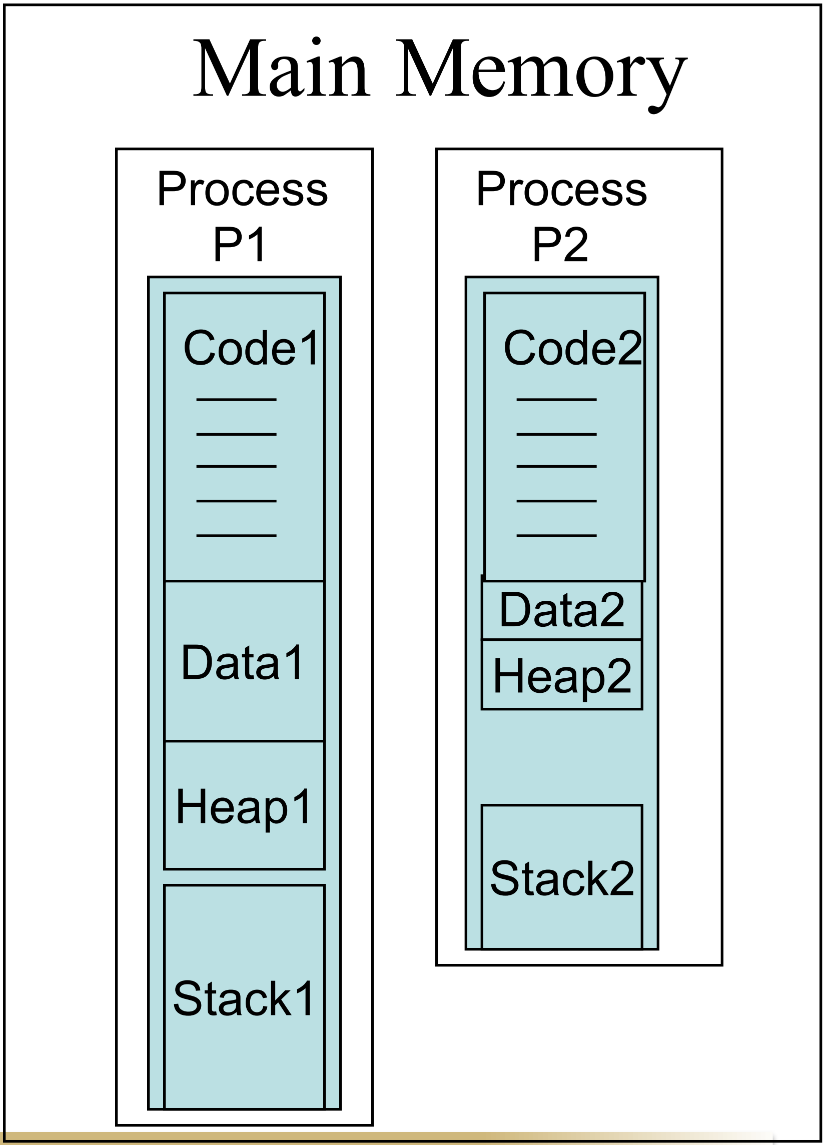
William Mortl, MS & Rick Han, PhD

Spring 2016

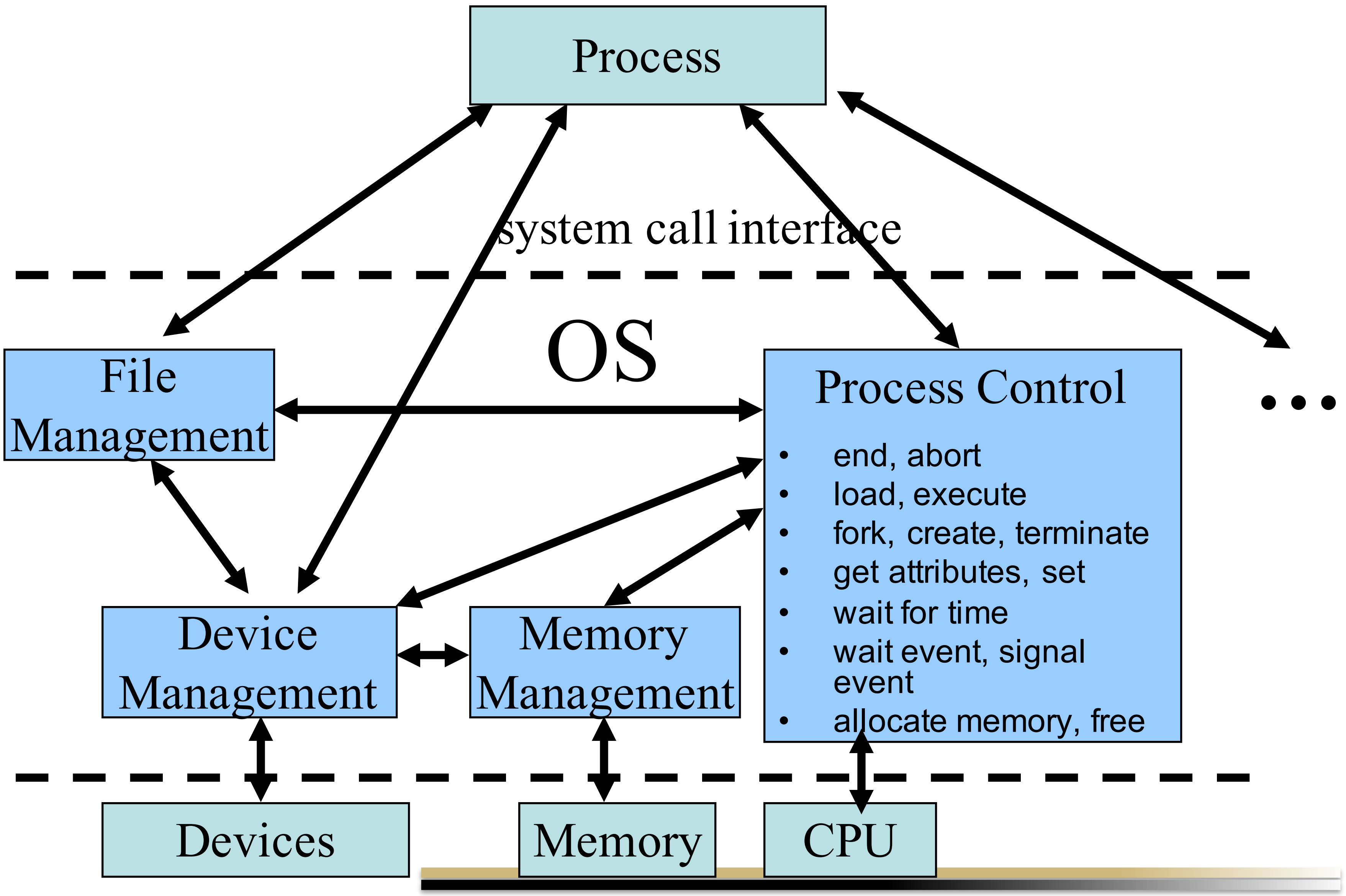
University of Colorado Boulder

Recap

- Process Management
  - Definition of a Process: an actively executing program in its own address space, plus the values it stores in the CPU's registers
  - Each process has its own code, data, heap and stack (caveats: threads and shared libraries)



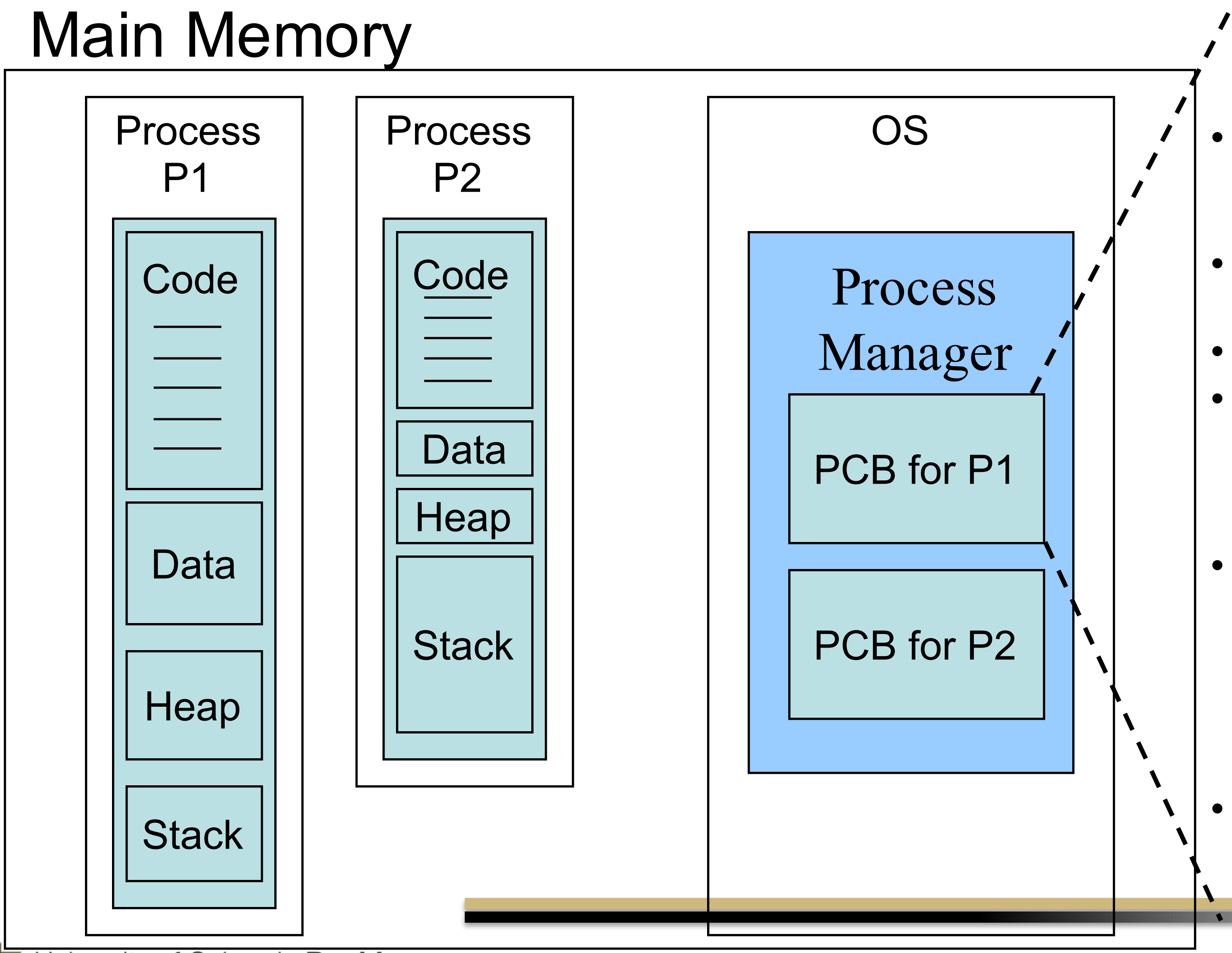
Process Management



Process Manager

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

Process Control Block (PCB)



- 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:
    1. allocate space in memory for the process
    2. Set up PCB state for process, assigns PID, etc.
    3. Copy code of program name from hard disk to main memory, sets PC to entry point in main
    4. Schedule the process for execution
  - As we will see, this combines UNIX's fork() and exec() and achieves the same effect

Creating Processes in UNIX

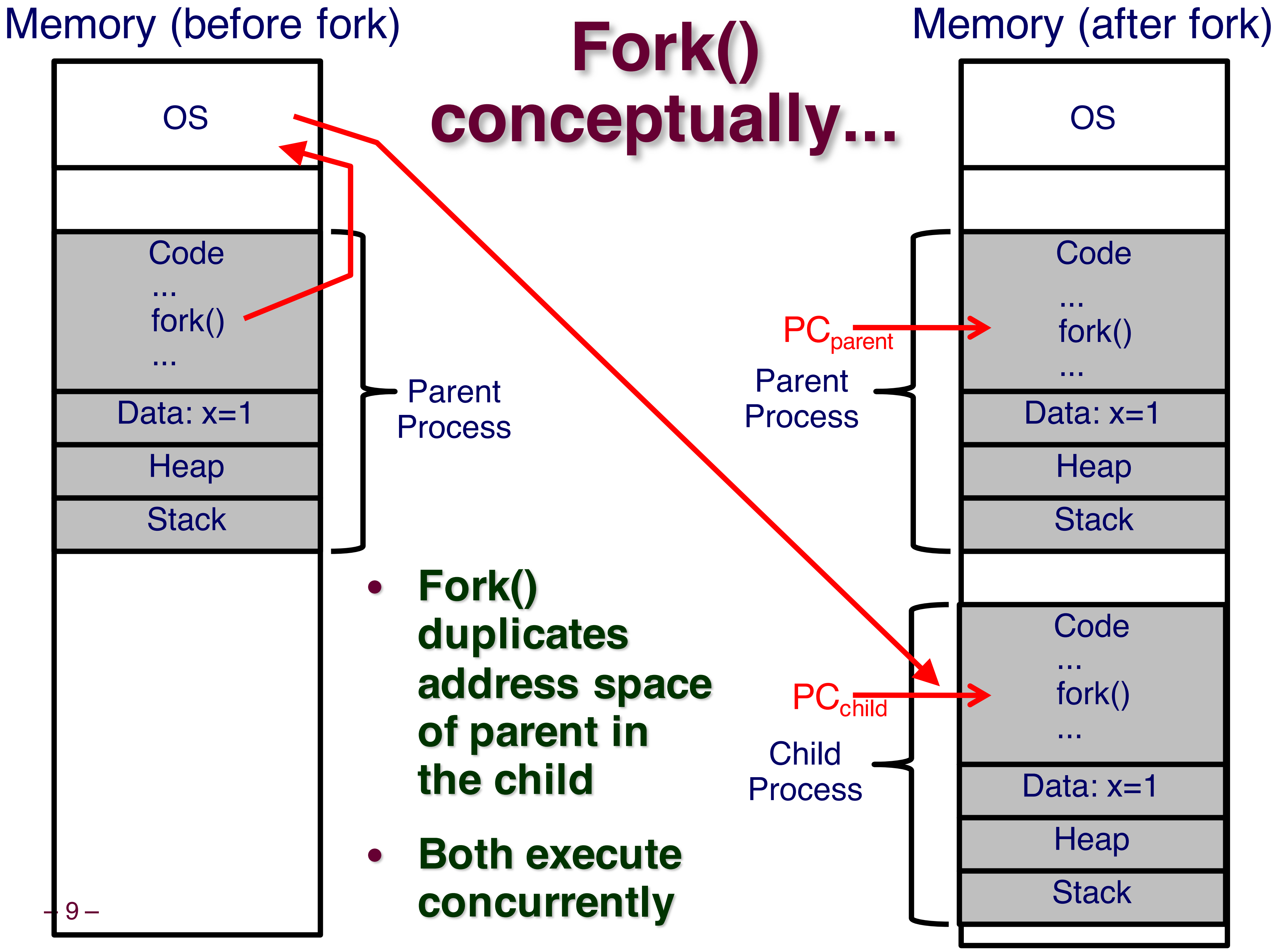
- Use fork() command to create/spawn new processes from within a process
  - When a *parent* process calls fork(), this creates a *child* process that inherits a copy of the parent's code;
  - In UNIX, the child starts executing at the same point as the parent, namely just after returning from the fork() call
  - Typically, only the child then calls exec() to copy in code from the new program to run

Forking Processes

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

- In UNIX, the fork() call returns
  - The child’s PID in the parent
  - Zero in the child

Fork() conceptually...

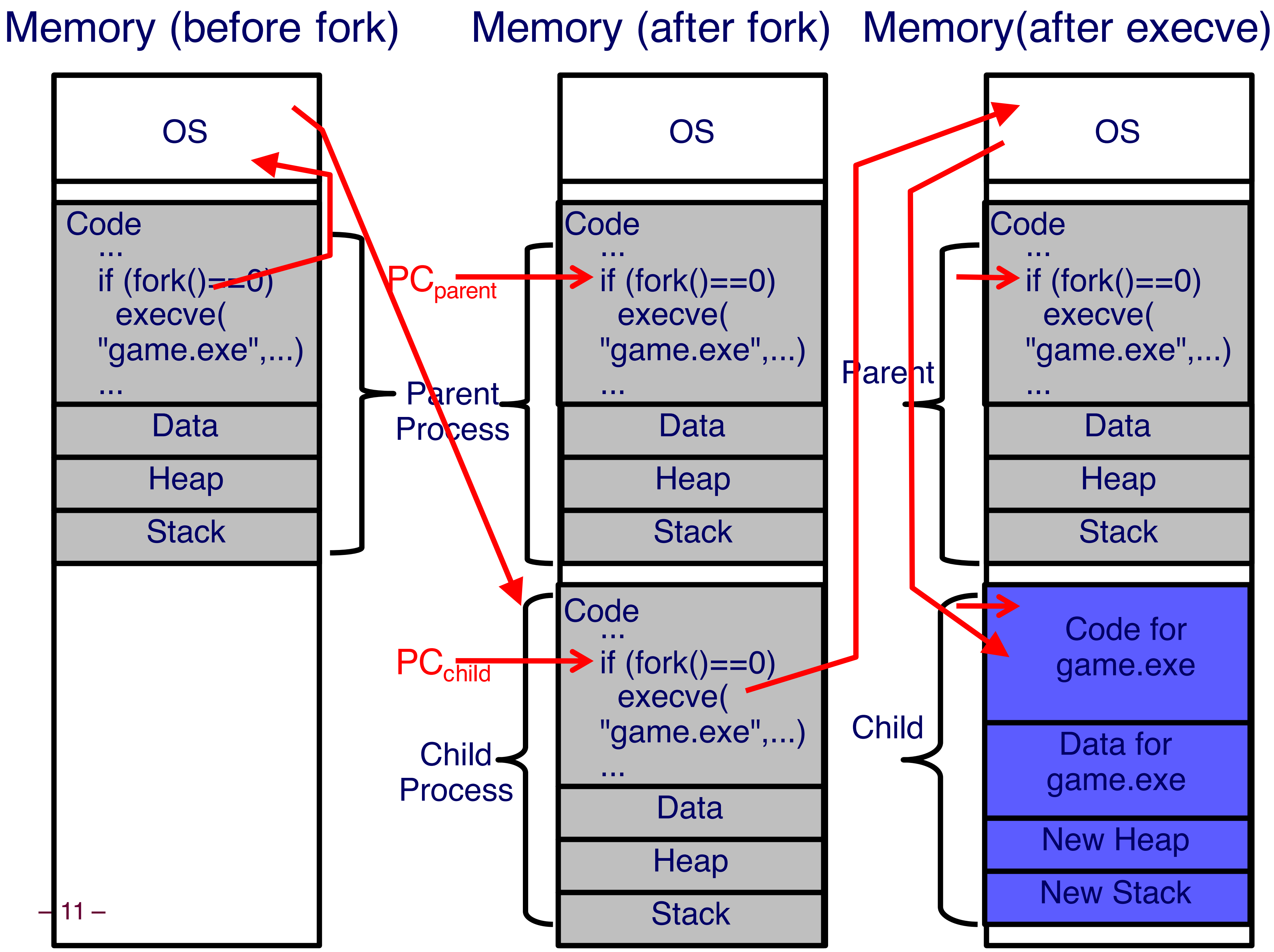


- **Fork() duplicates address space of parent in the child**
- **Both execute concurrently**

Loading New Processes' Code

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

- the exec() system call
  - loads program code into the calling process's memory (same address space!) - the calling code is erased!
  - clears the stack, and
  - begins executing the new code at its main entry point



```
Code
  ...
  if (fork()==0)
    execve(
  "game.exe",...)
  ...
```

Copy-On-Write

- copying the entire code of a parent into a child can be expensive on a fork() or CreateProcess()
- Better: let the child share the parent’s code pages
  - Accomplish this by having the page tables point to the same physical pages in memory
- Only create a copy of a page on a write.
  - Since the code is read-only, then never have to create a 2nd copy of the code pages.
- This speeds up creation of new processes

Deleting Processes

- In UNIX, the wait() 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
  - This is also called reaping
  - There is also waitpid() to wait on a particular child process to finish

Process Context Switches

- A context switch can occur because of
  - a system call
  - an I/O interrupt, e.g. disk has finished reading
  - a timer interrupt
- Context switch time is pure overhead
  - Have to save the state of the process 1 in the PCB1
  - Then have to load the state of process 2 from PCB2
  - Typically on the order of microseconds vs time slices of tens of milliseconds

Accessing Process State

- One way is through standard system calls
- Another way is through the /proc "pseudo"-file system interface on Linux systems
  - 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

Using /proc

- Can read and write status variables
  - Most /proc files are read-only
  - `sysctl` can be used to change a limited # of kernel variables
    - can tune kernel at run-time
- Many system utilities like `ps` (process status) and `top` are simply calls to files in the /proc directory

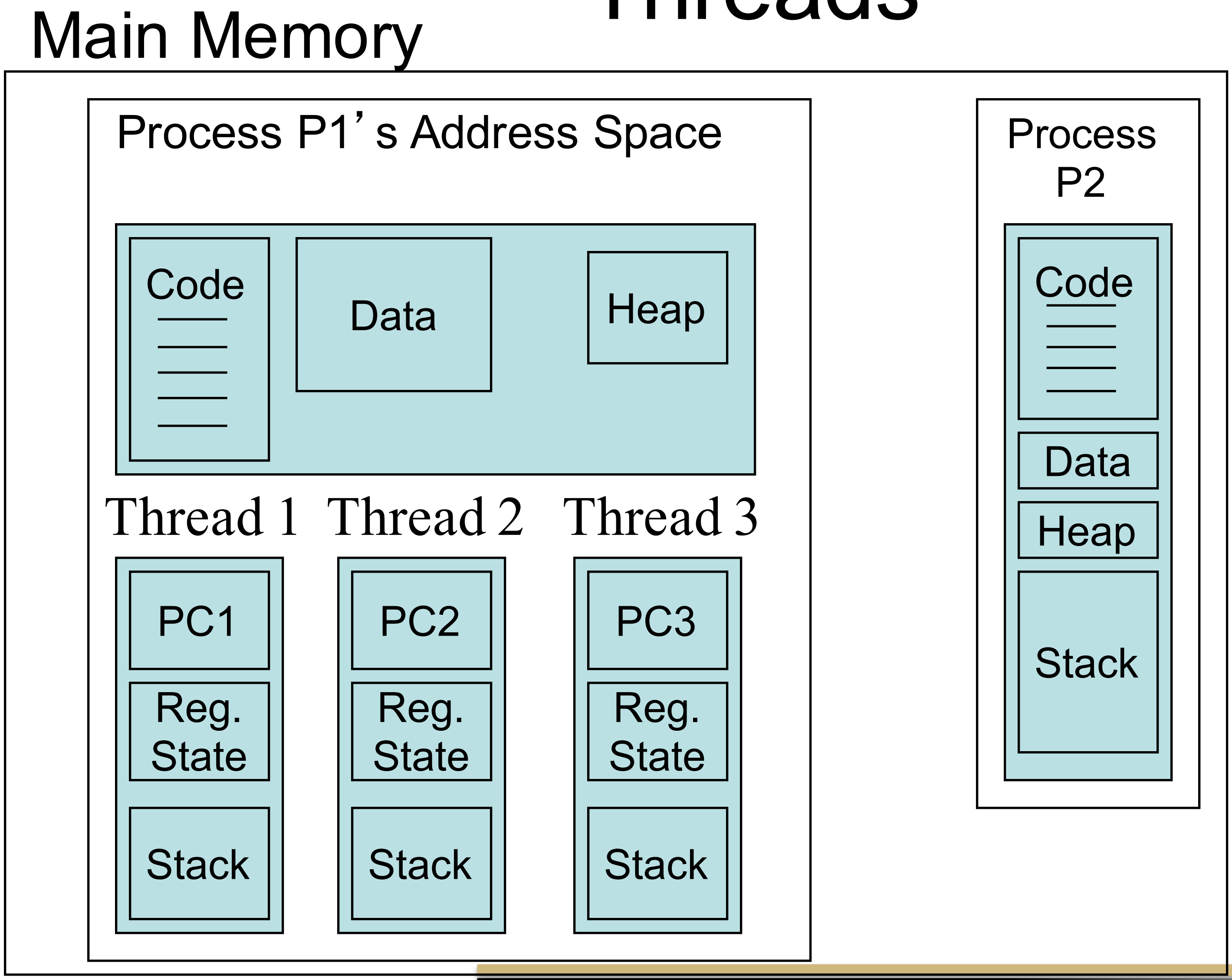
/proc to Access System State

- /proc also contains information about
  - hardware I/O devices
  - overall system status
  - not just process state
- Examples:
  - /proc/interrupts shows which interrupts are in use, and how many of each there have been
  - /proc/devices lists the device drivers configured into the currently running kernel
  - /proc/net contains networking information

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*

Threads



- Process P1 is *multithreaded*
- Process P2 is single threaded
- The OS is *multiprogrammed*
- If there is preemptive timeslicing, the system is *multitasked*

Motivation for Threads

- reduced context switch overhead vs multiple processes
  - 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

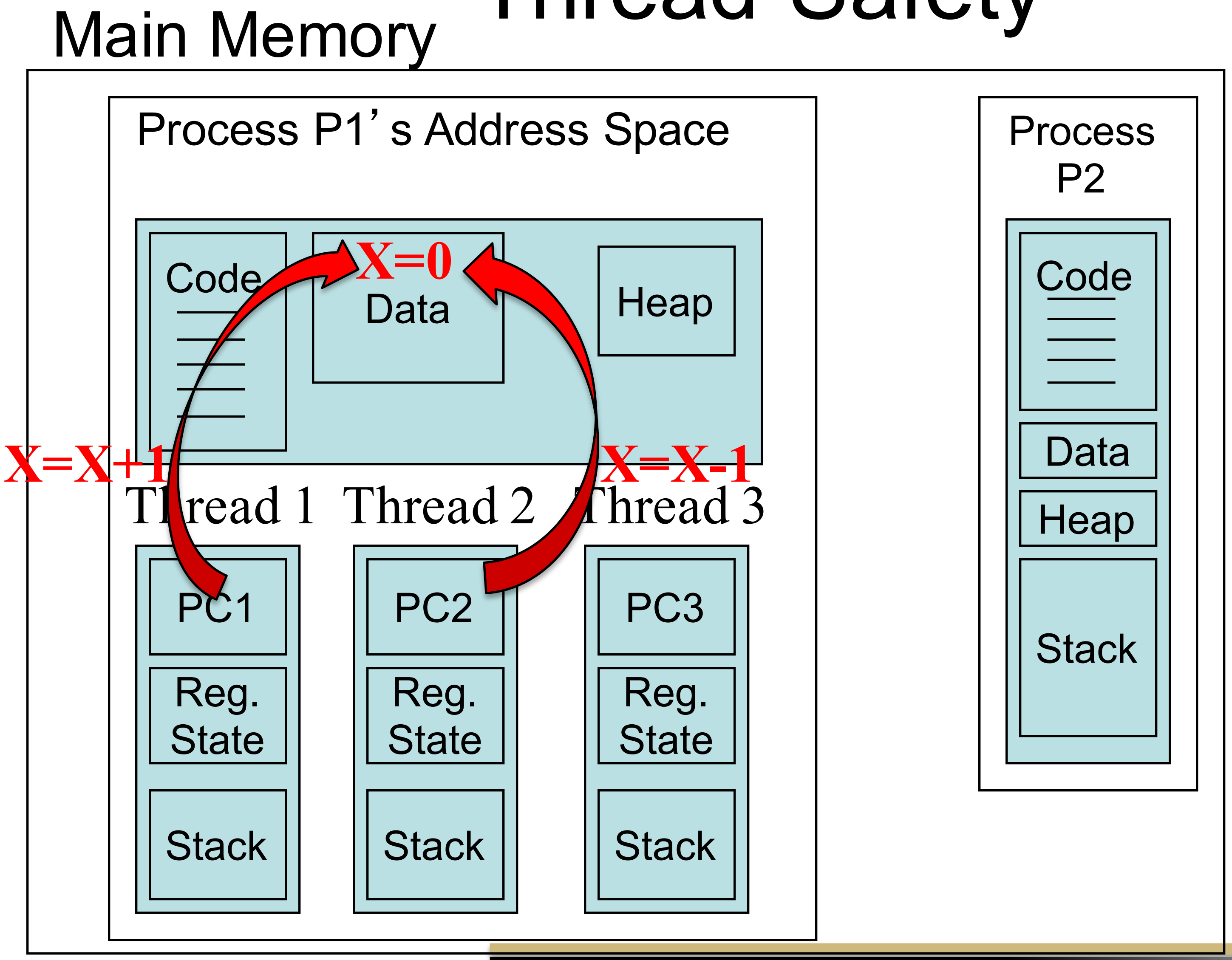
Motivation for Threads (2)

- 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

Applications, Processes, and Threads

- Application = $\sum$ Processes$_i$
  - e.g. a server could be split into multiple processes, each one dedicated to a specific task (UI, computation, communication, etc.)
- Process = $\sum$ Threads$_j$
  - e.g. a Web server process could spawn a thread to handle each new http request for a Web page
- An application could thus consist of many processes and threads

Thread Safety



- Suppose Thread 1 wants to increment global variable X
- Thread 2 wants to decrement variable X
- *Could have a race condition (see Chapter 6)*

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

Thread-Safe Code (2)

- The use of local variables is thread safe
- To make a thread-unsafe code thread-safe,
  1. add locking mechanisms to protect access to shared resources
     - thread-safe code protects and synchronizes access to global and static variables, i.e. persistent data, with locking and synchronization mechanisms
  2. Rewrite code to only use local variables and parameters, and make sure any called functions do the same thing, and so on down the call chain…

Thread-Safe Code (3)

- Example of thread-safe code:

```
function f() {
   lock();
   change global variable G;
   unlock();
}
```

The lock() function's semantics:

- once it's called, the caller has the lock until they call unlock().
- Any other task that calls lock() before the 1st caller has called unlock() will block until the lock is released

Reentrant Code

- Reentrancy is a concept related to but different from thread safety
  - Code is reentrant if a *single* thread (really, a sequence of execution) can be interrupted in the middle of executing the code and then can reenter the same code later in a safe manner (before the first entry has been completed)
  - In contrast, code is thread safe if *multiple* threads can operate safely in the same code at the same time

Reentrant Code (2)

- Reentrancy was developed to describe interrupt service routines (ISRs) in early OSs, i.e. interrupt handlers
  - in the middle of an OS processing an interrupt, its ISR can be interrupted to process a 2nd interrupt
  - The same OS ISR code may be reentered a 2nd time before the 1st interrupt has been fully processed
  - If the ISR code is not well-written, i.e. reentrant, then the system could hang or crash

Reentrant Code Example

Neither Reentrant Nor Thread-safe

```
int global1 = 1;
int f() {
   global1 = global1 + 2;
   return global1;
}
int g() {
    return f() + 2;
}
```

Race condition if two threads call f() (or g()). If a single thread is interrupted in "ISR" f before return, & reenters f, global1 grows by 2, so resumed thread returns wrong value (bigger by 2)

Reentrant and Thread-safe

```
int f(int i) {
    int local = i;
    local = local + 2;
    return local;
}
int g(int j) {
    int local = j;
    return f(local) + 2;
}
```

No race conditions in f() or g()

Thread-safe but not reentrant

- Earlier example of code below was thread-safe but not reentrant:

```
function f() {
  lock();
  change global variable G;
  unlock();
}
```

Not reentrant because if f() is interrupted just before the unlock(), and f() is called a 2nd time, the system will hang, because the 2nd call will try to lock, and be unable to lock, because the 1st call had not yet unlocked the system

Reentrant Code (2)

– Code can be:

Thread-Safe And Reentrant	Thread-Safe And Not Reentrant
Not Thread-Safe And Reentrant	Neither Thread-Safe Nor Reentrant

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.

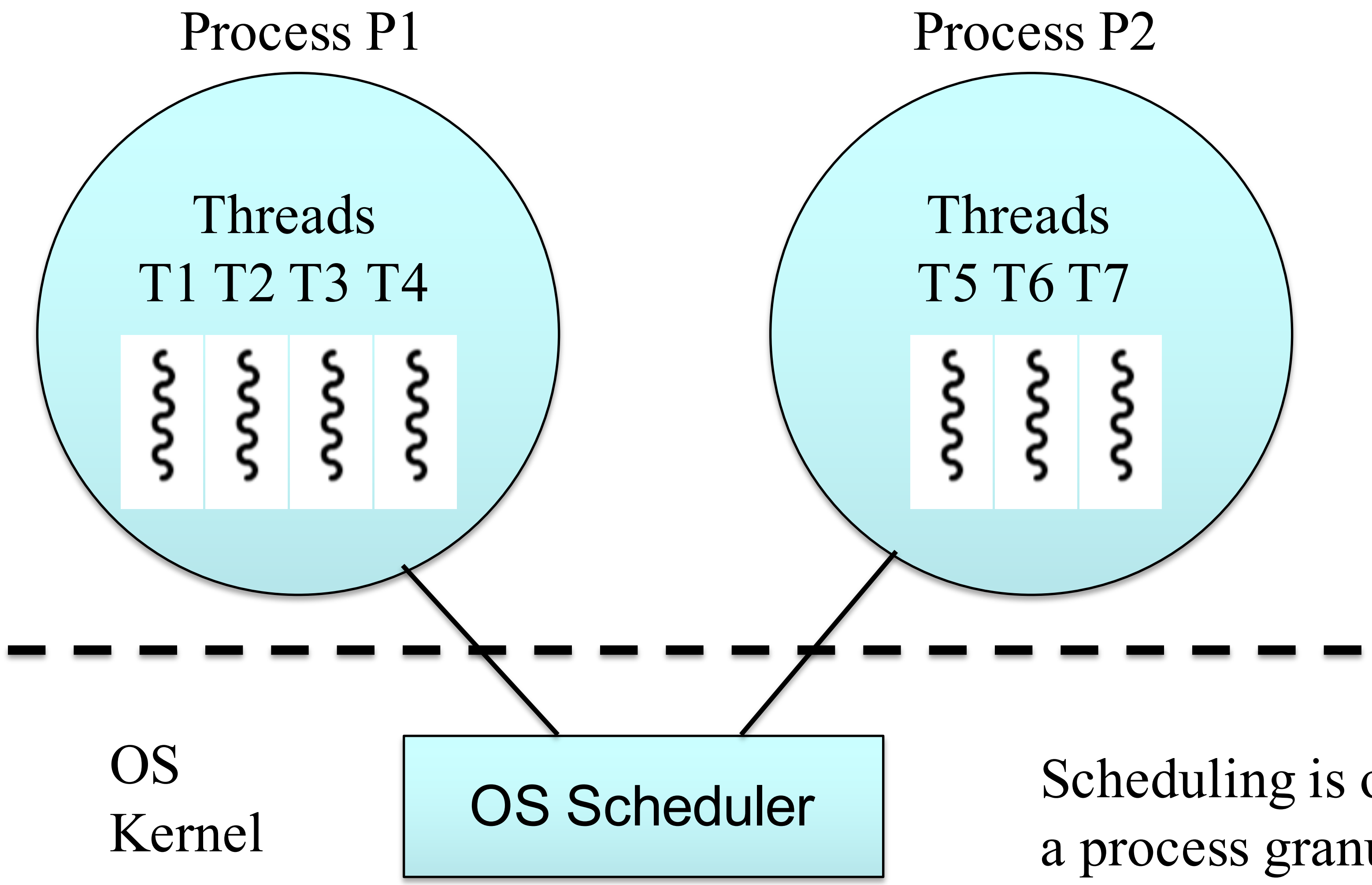
Processes vs. Threads (2)

- 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

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
  - provides interface to create, delete threads in the same process
  - threads will synchronize with each other via the user space threading package or library
  - 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*

User-Space Threads

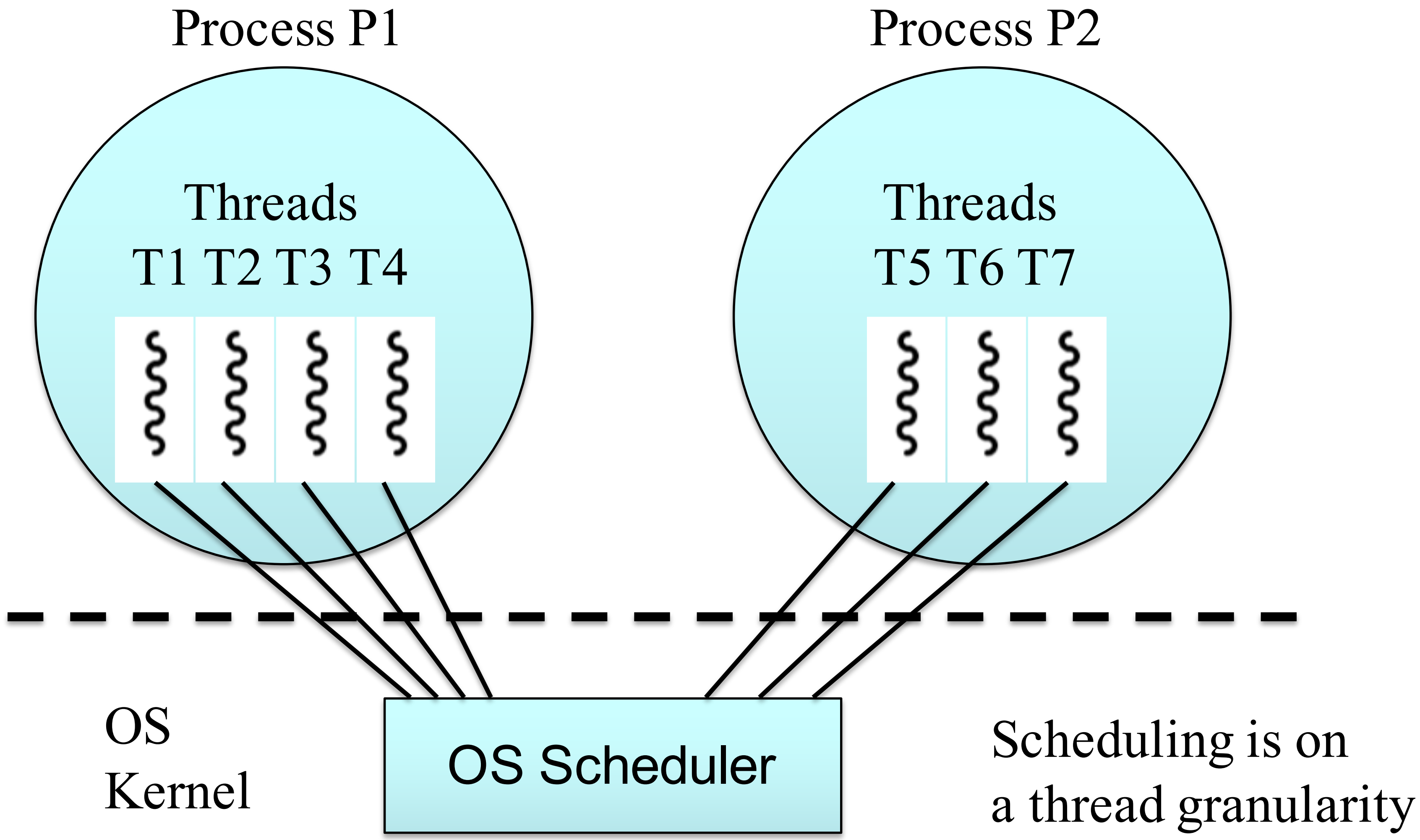


Scheduling is on a process granularity

Kernel Threads

- *Kernel threads* are supported by the OS
  - kernel sees threads and schedules at the granularity of threads
    - In user space threads, the kernel doesn't see threads, only processes, and only schedules processes
  - 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

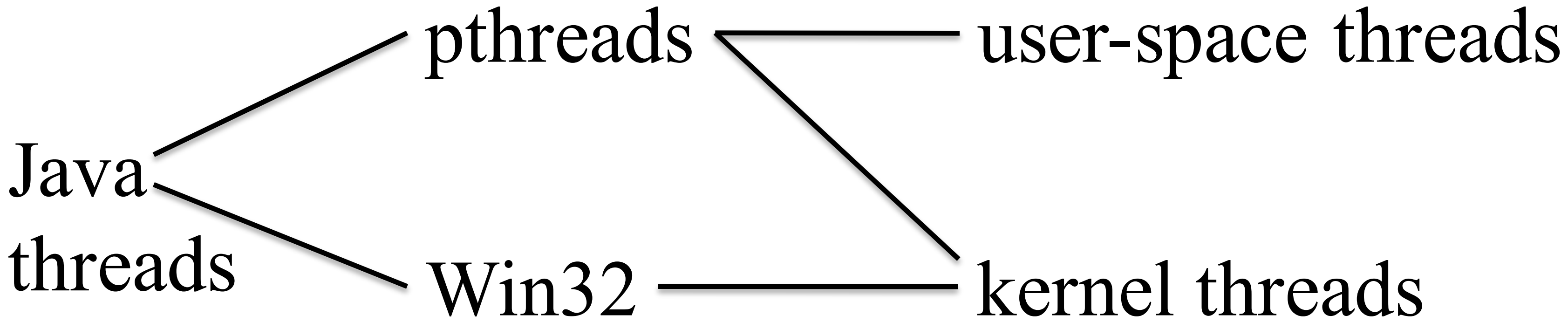
Kernel Threads



Scheduling is on a thread granularity

User-Space & Kernel Threads

- Java thread library is running in Java VM on top of host OS, so on Windows it's implemented on top of Win32 threading, while on Linux/Unix it's implemented on top of pthreads
- Possible scenarios:


Java threads
pthreads
Win32
user-space threads
kernel threads