CSCI 3753
Operating Systems

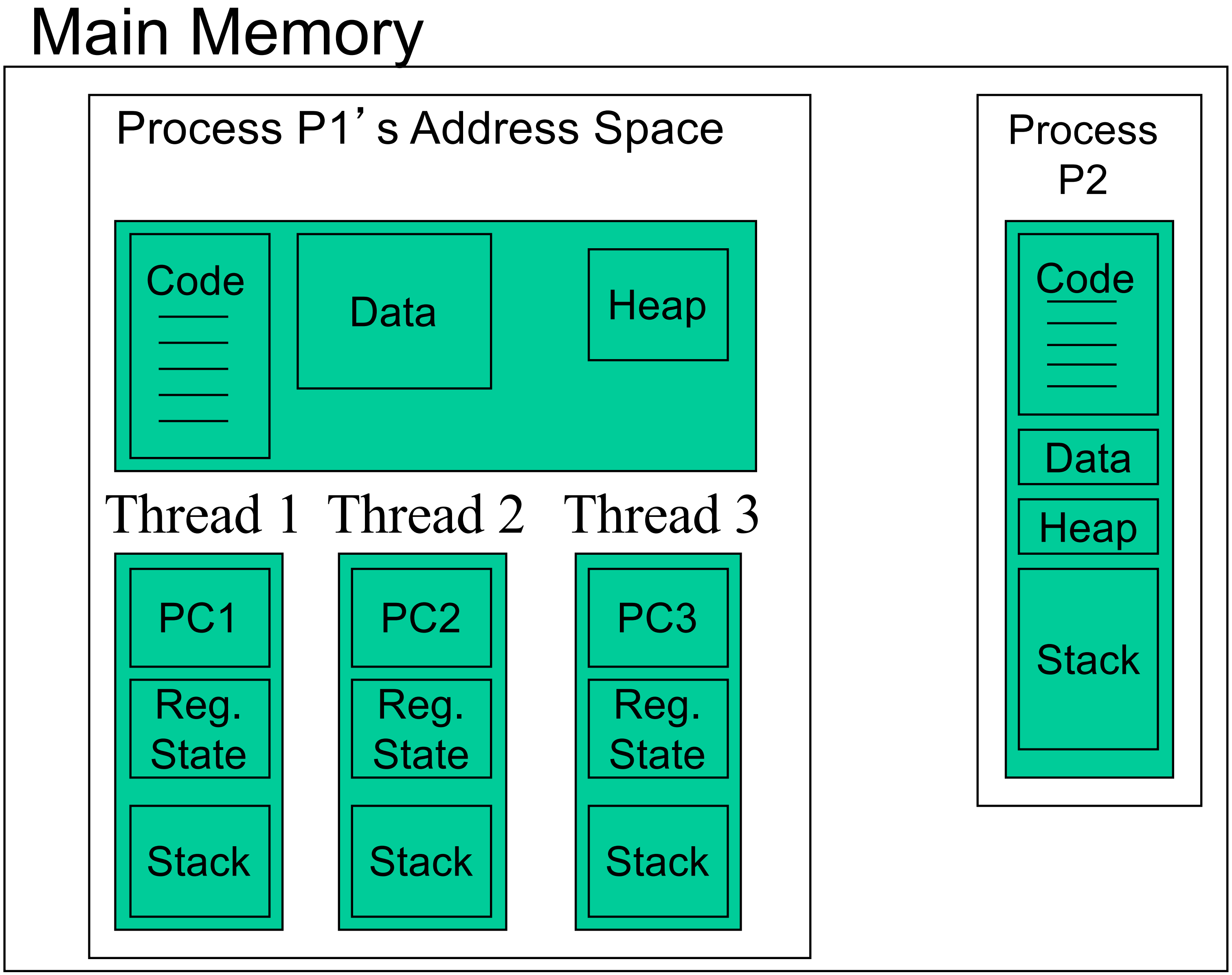
Threads

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 09/07/2016

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

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

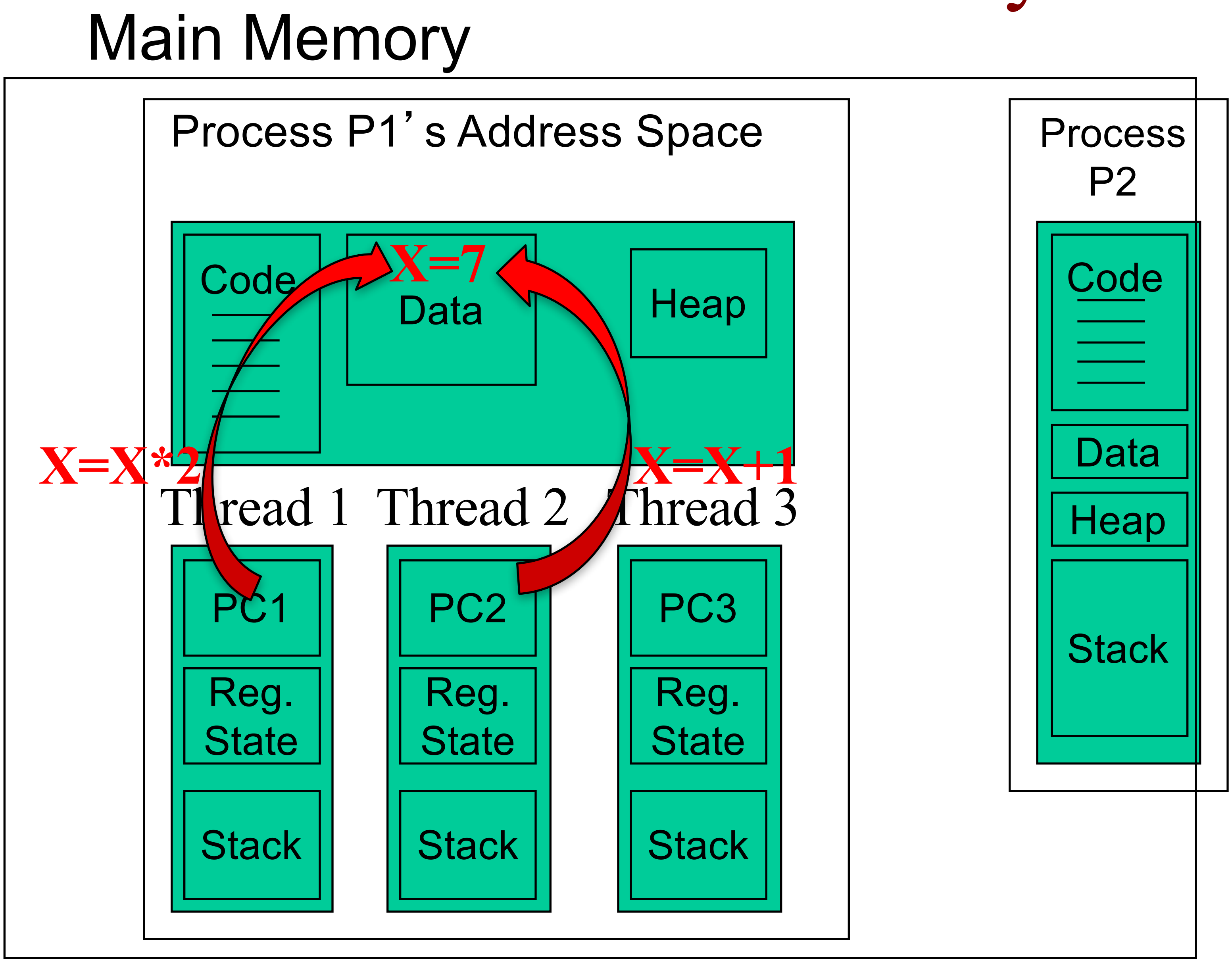
Motivation for Threads (2)

- 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

- 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 Safety



- Suppose Thread 1 wants to multiply X by 2
- Thread 2 wants to decrement 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

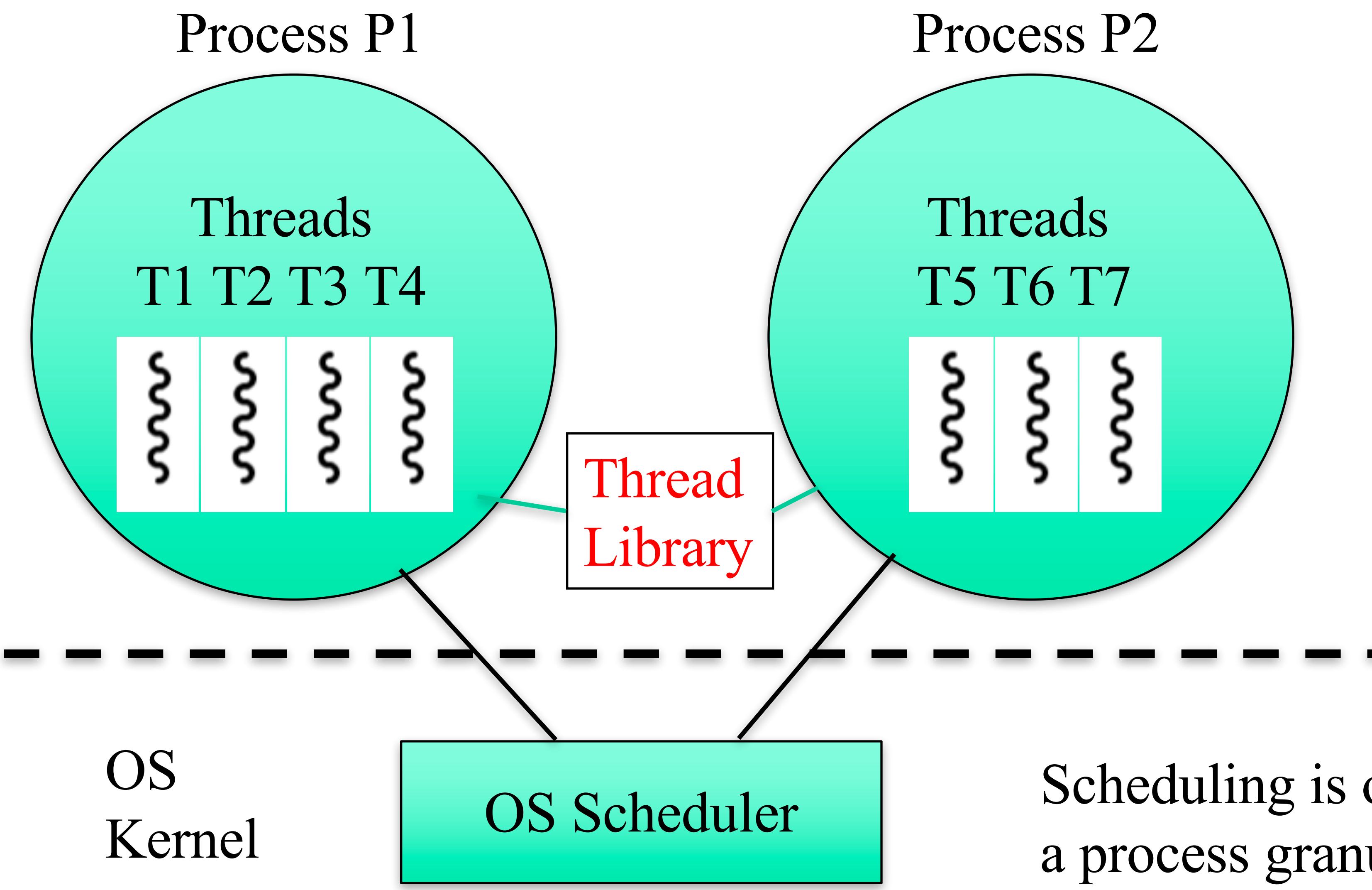
- Why are processes still used when threads bring so many advantages?
  1. Some tasks are sequential and not easily parallelizable, and hence are single-threaded by nature
  2. No fault isolation between threads
     - If a thread crashes, it can bring down other threads
     - If a process crashes, 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

User-Space Threads



Scheduling is on a process granularity

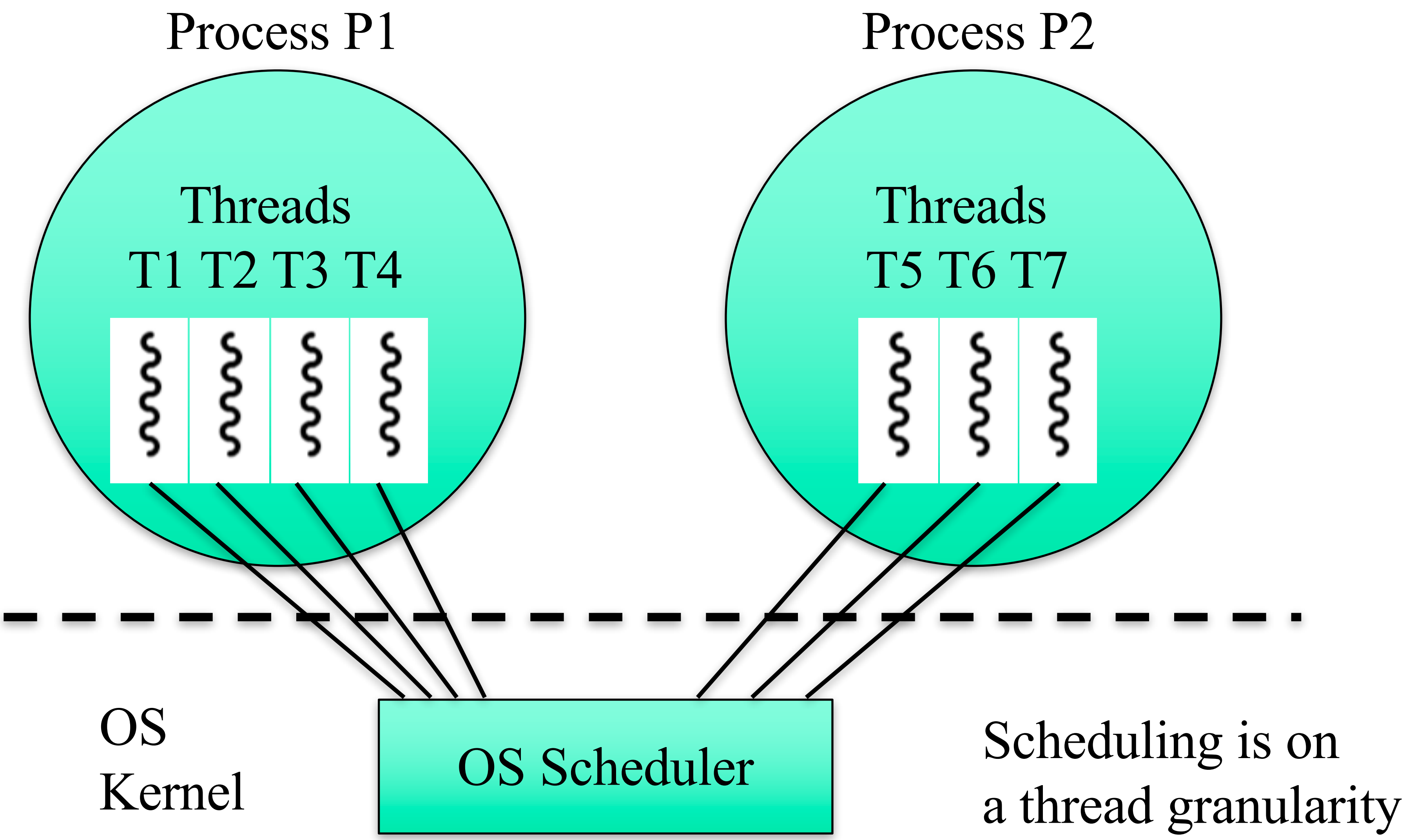
User-Space Threads

- *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

- *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

Kernel Threads



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: