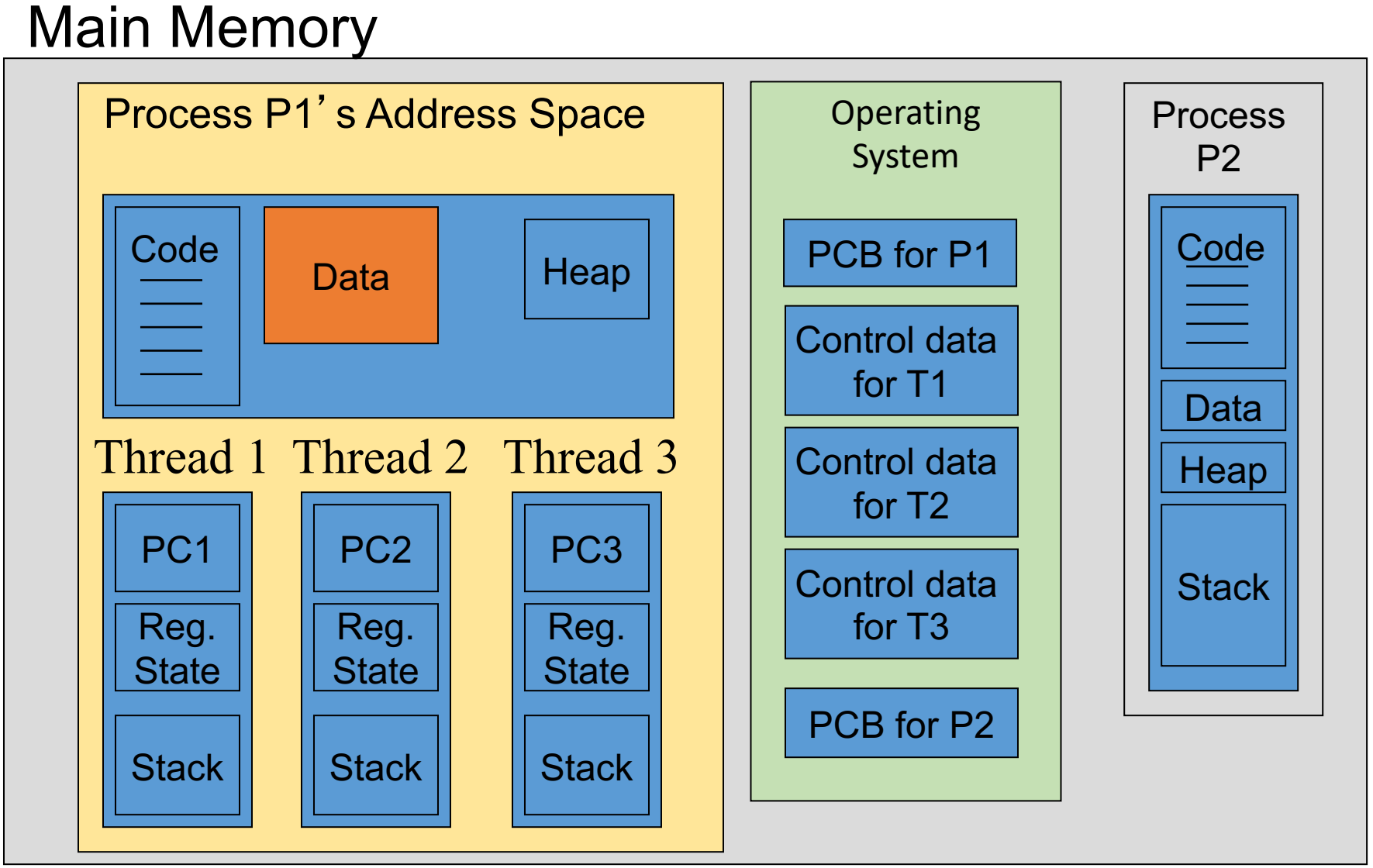
Lecture 7
Threads

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

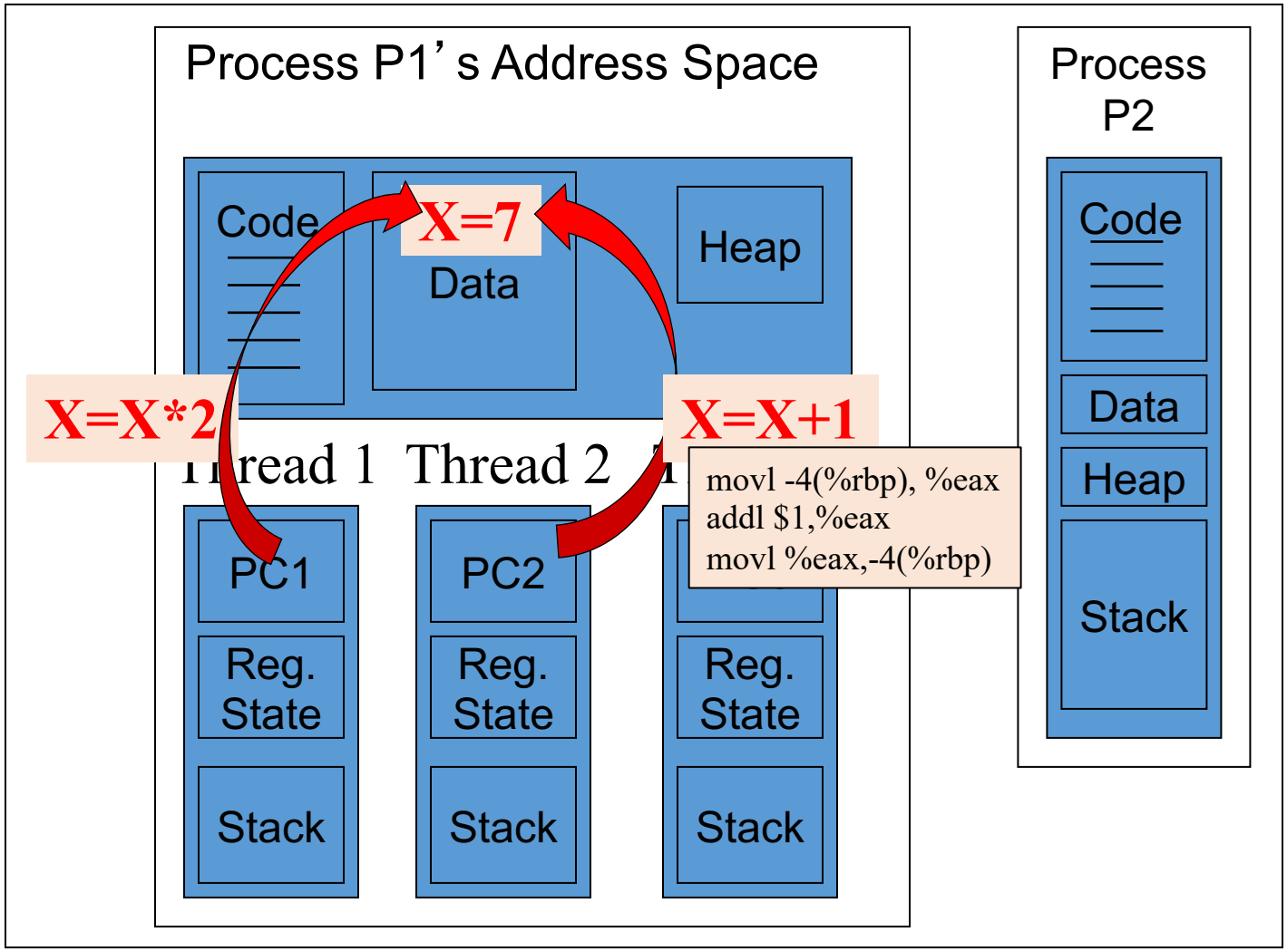
More reasons for

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:
  - POSIC 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

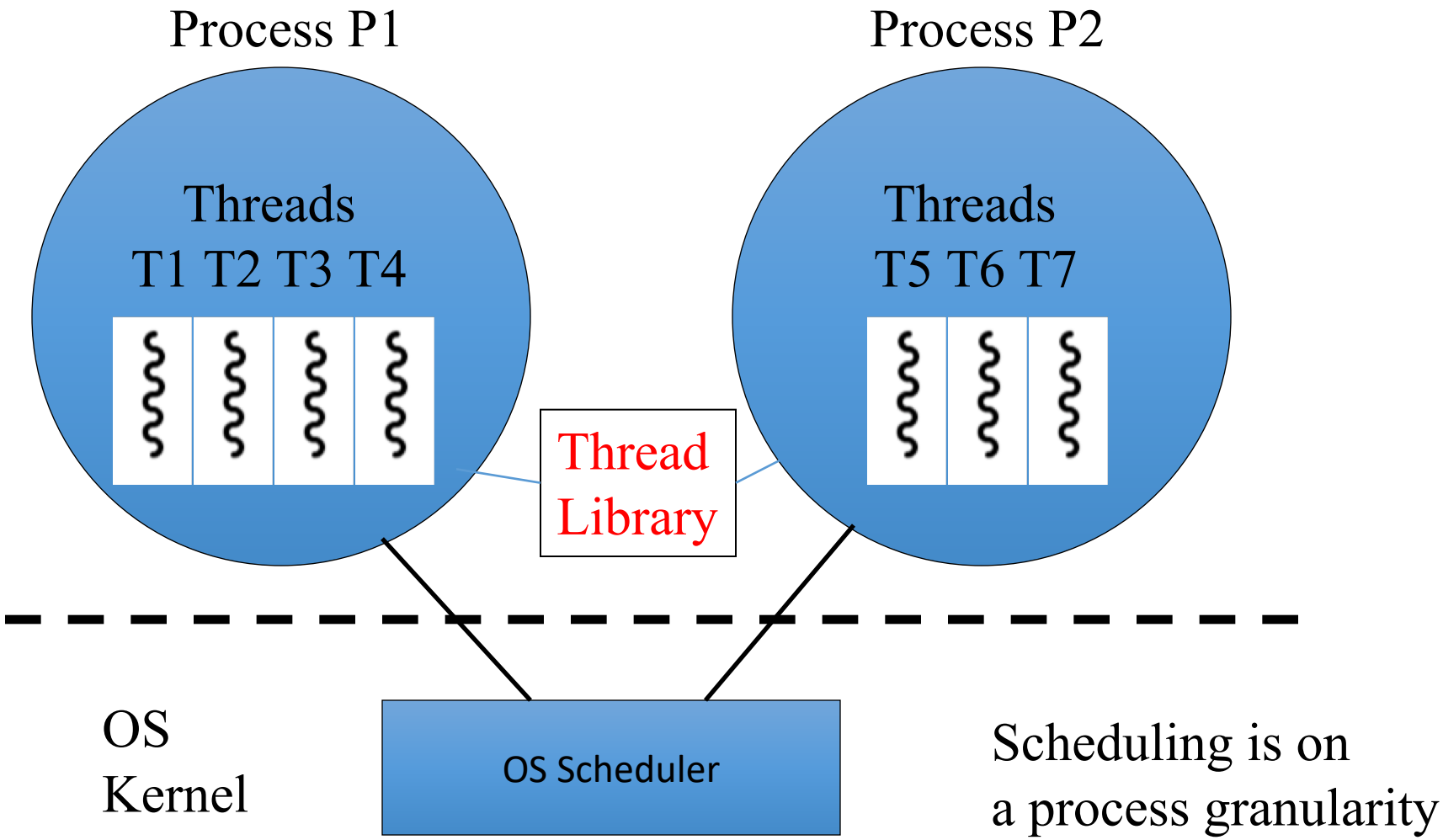
pthread_exit (value) ;

- 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



User-Space Threads

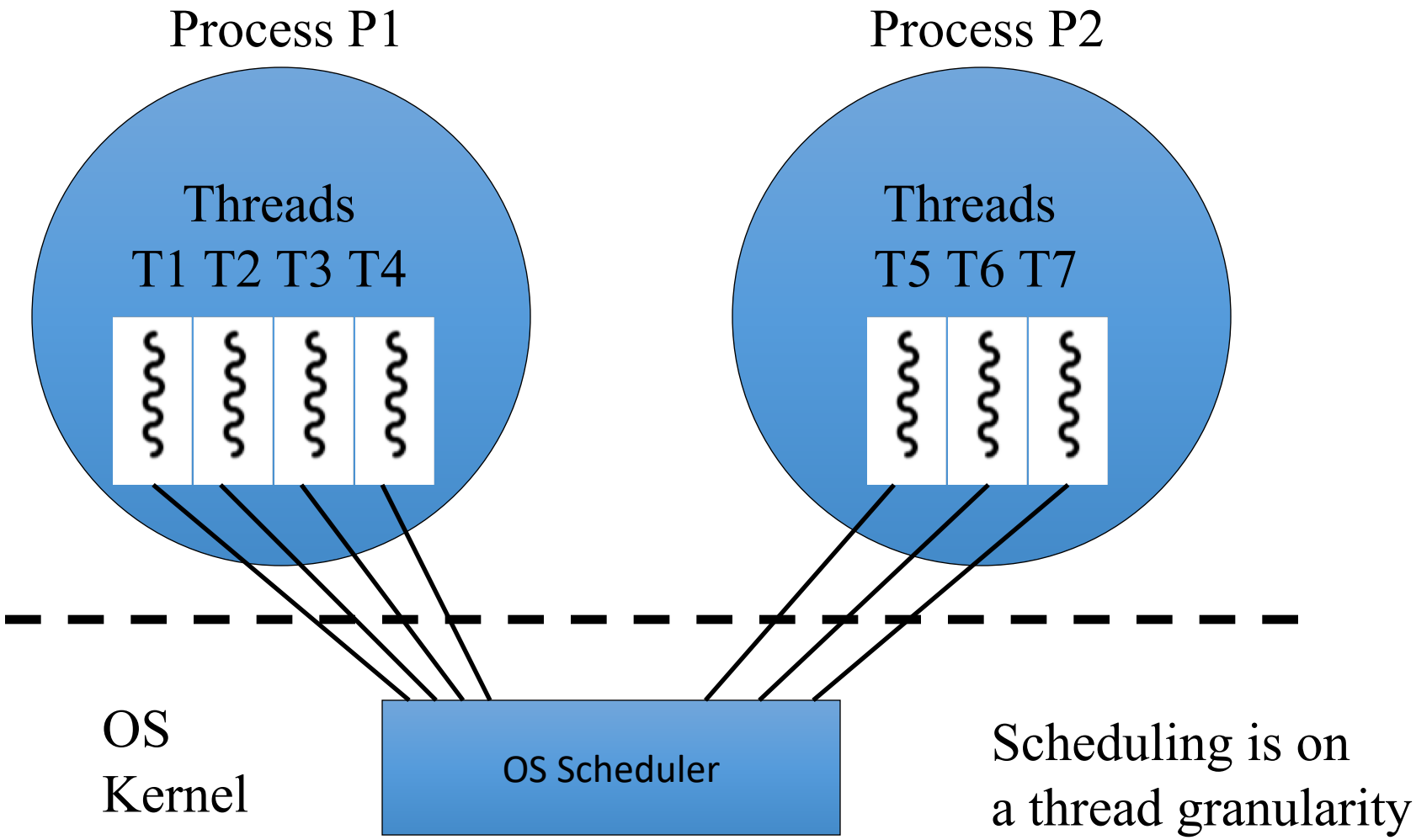
- *User space threads* are usually cooperatively multitasked, i.e. user threads within a process voluntarily give up the CPU to each other
  - threads will synchronize with each other via the user space threading package or library
  - Thread library: provides interface to create, delete threads in the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

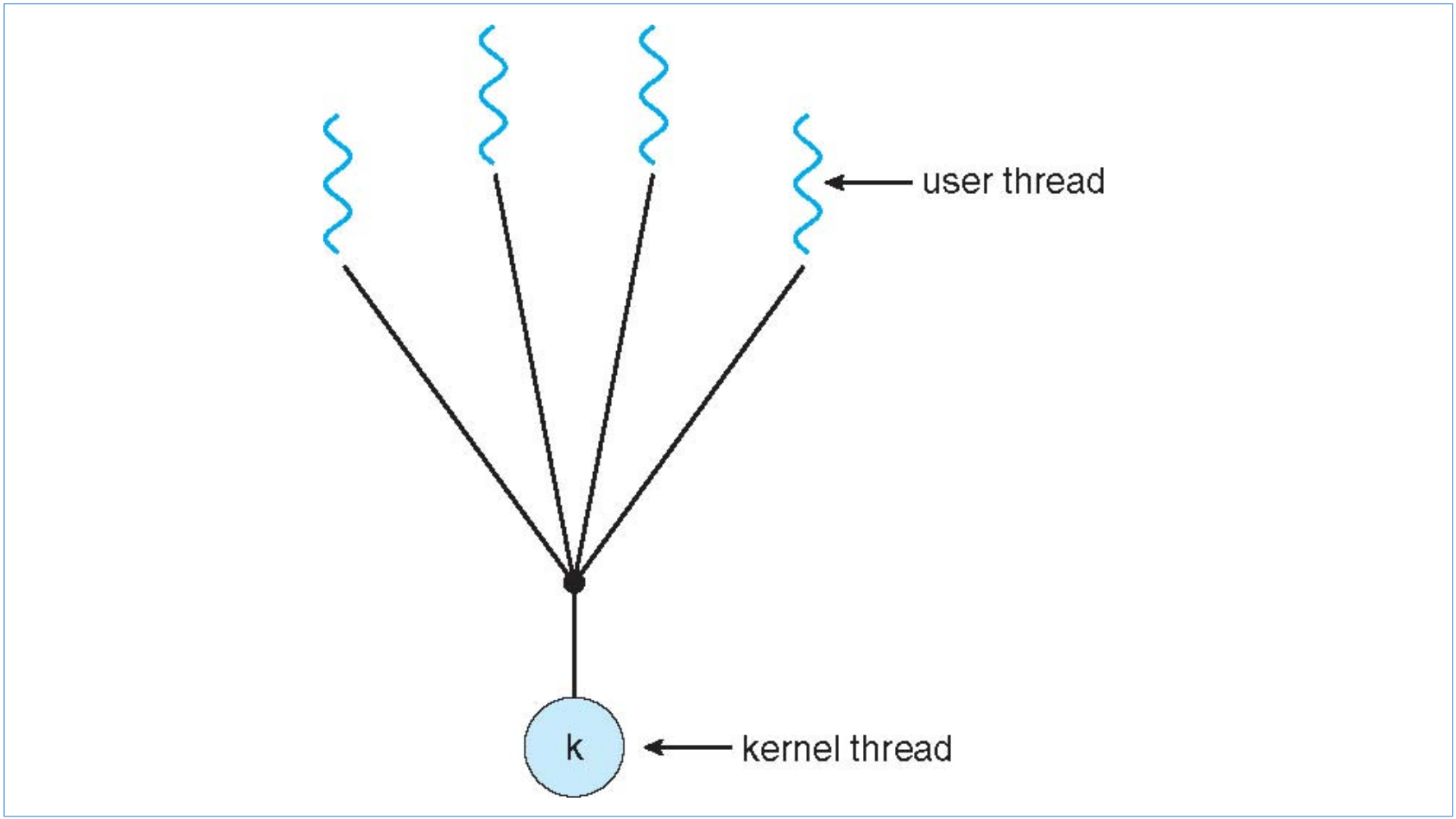
Kernel Threads



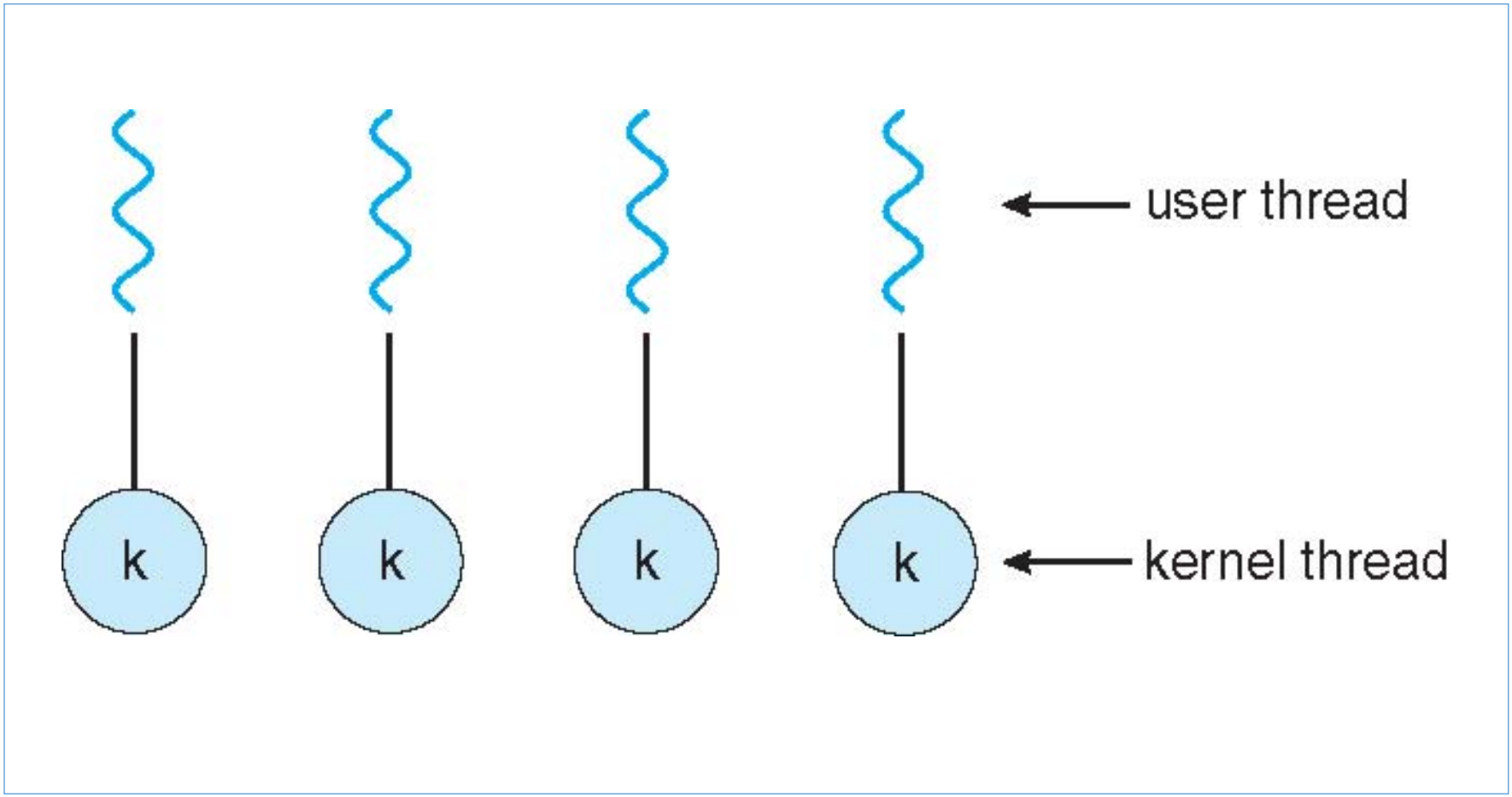
Kernel Threads

- *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