Chapter 4: Threads

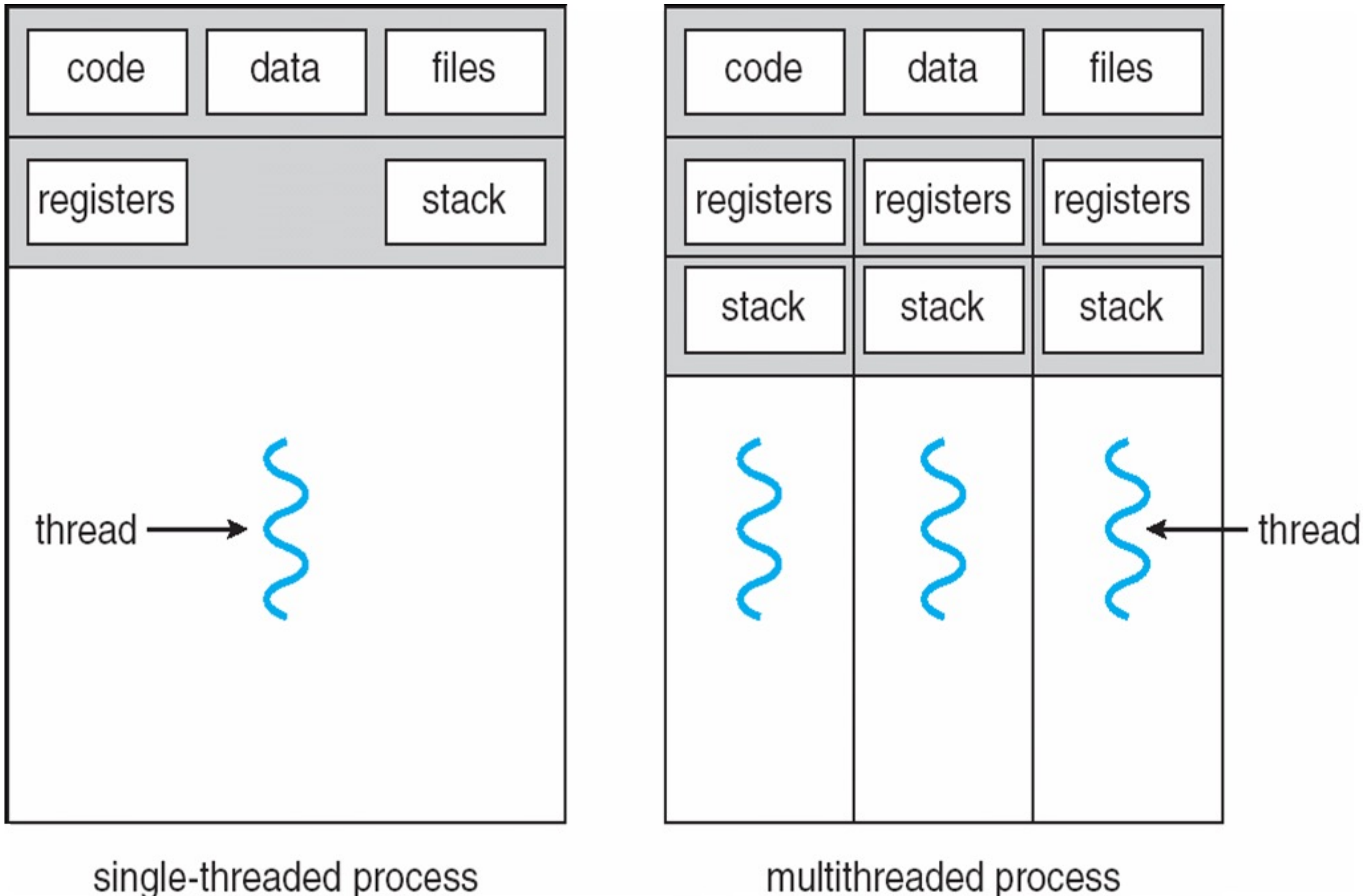
Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Windows XP Threads
- Linux Threads

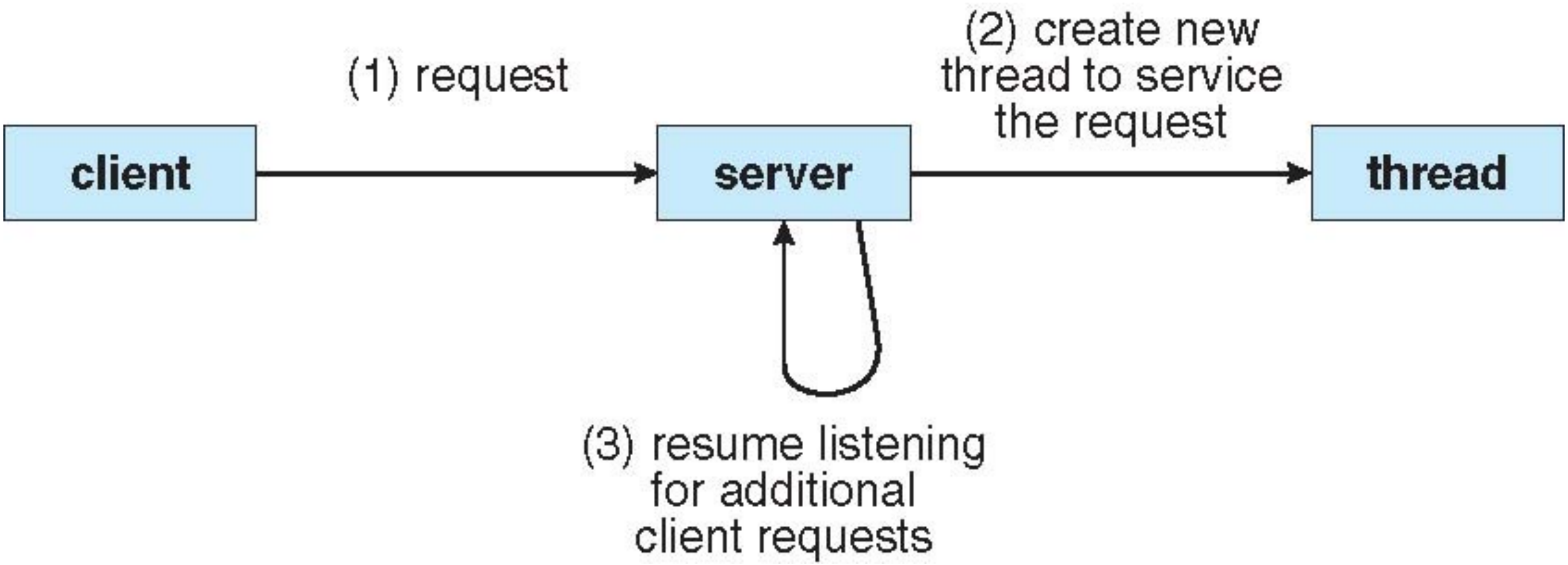
Objectives

- To introduce the notion of **a thread** — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming

Single and Multithreaded Processes



Multithreaded Server Architecture



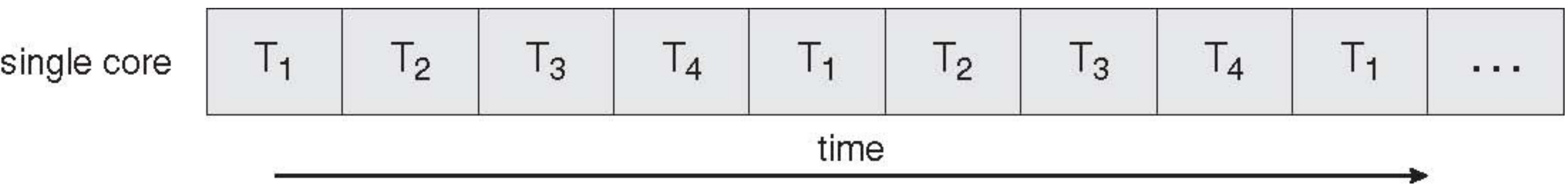
Benefits

- Responsiveness
  - Allows a program continue running if part of it is blocked or its is performing a lengthy operation
- Resource Sharing
  - Threads share the memory and the resources of the parent process
- Economy
  - In Solaris, creating a process is 30 times slower than creating a thread, context switching is 5 times slower.
- Scalability
  - Multithreading on a multi-CPU machine increase parallelism

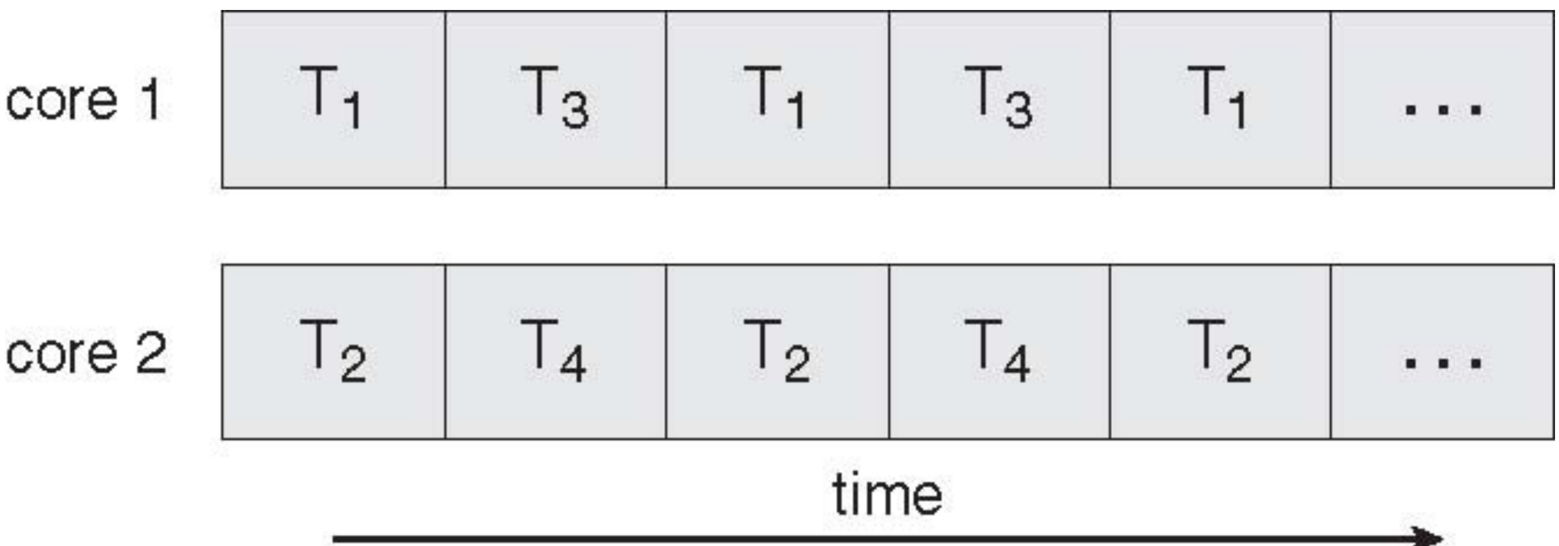
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
    - Find area that can be divided into separate, concurrent tasks
  - **Balance**
    - Ensure concurrent tasks perform equal work of equal value
  - **Data splitting**
    - Data accessed must be divided to run on speparate cores.
  - **Data dependency**
    - One task depends on data from another, ensure synchrornization
  - **Testing and debugging**
    - Test many execution path

Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

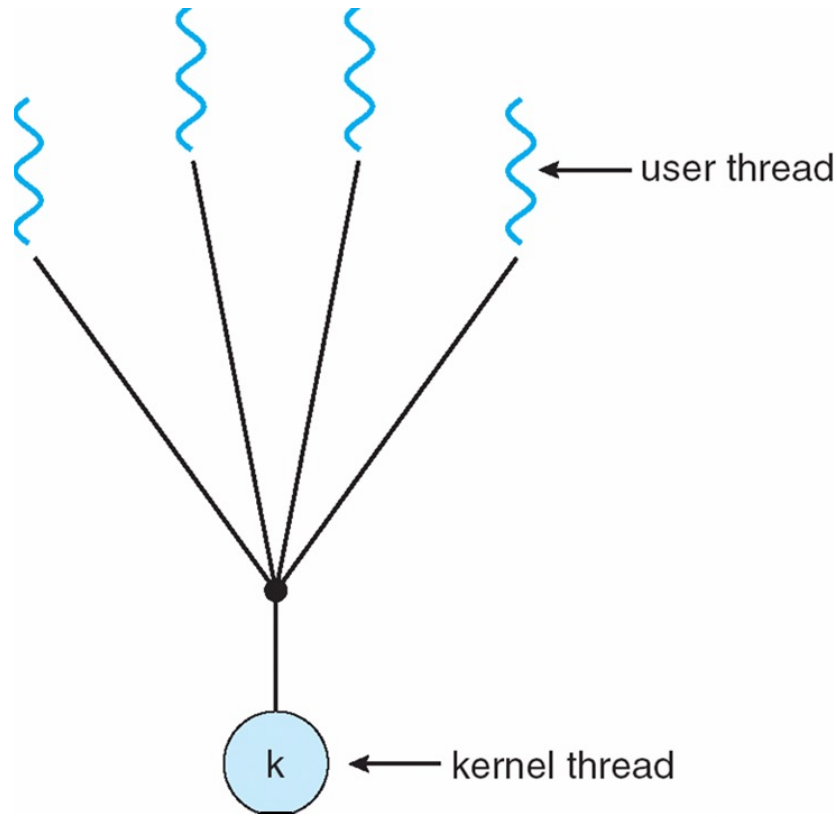
- Supported by the Kernel/OS
- All contemporary OS support kernel threads
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- Many-to-One: map **many user-level** threads to **one kernel** thread
- One-to-One: map each user-level thread to a kernel thread
- Many-to-Many: multiplexes many user-level threads to a smaller or equal number of kernel theads

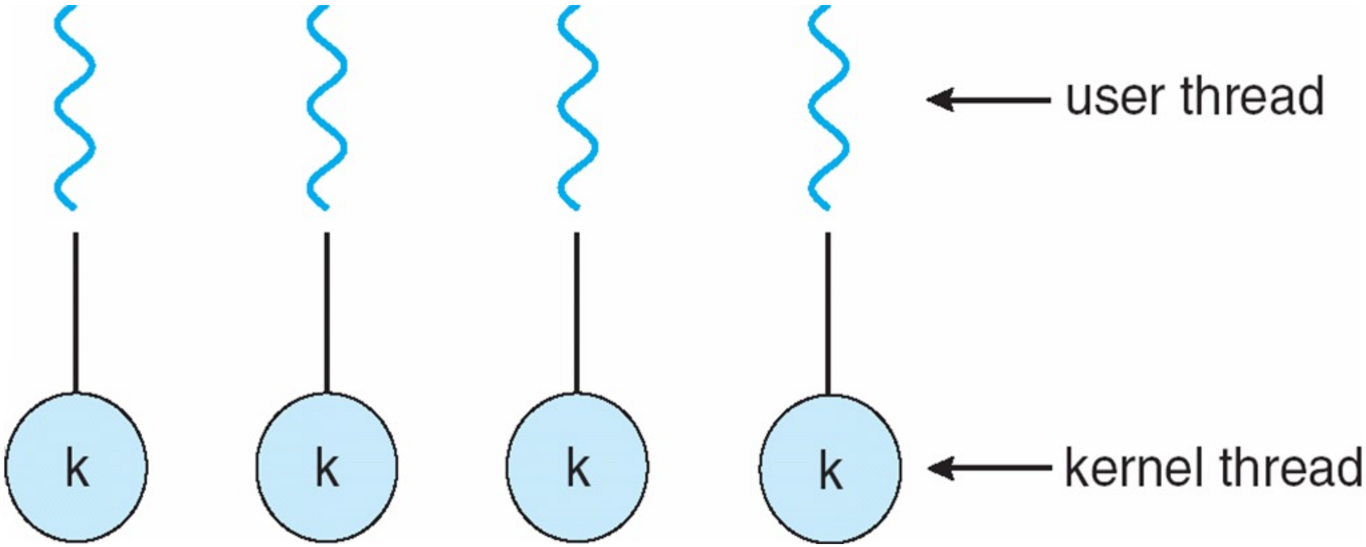
Many-to-One Model

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



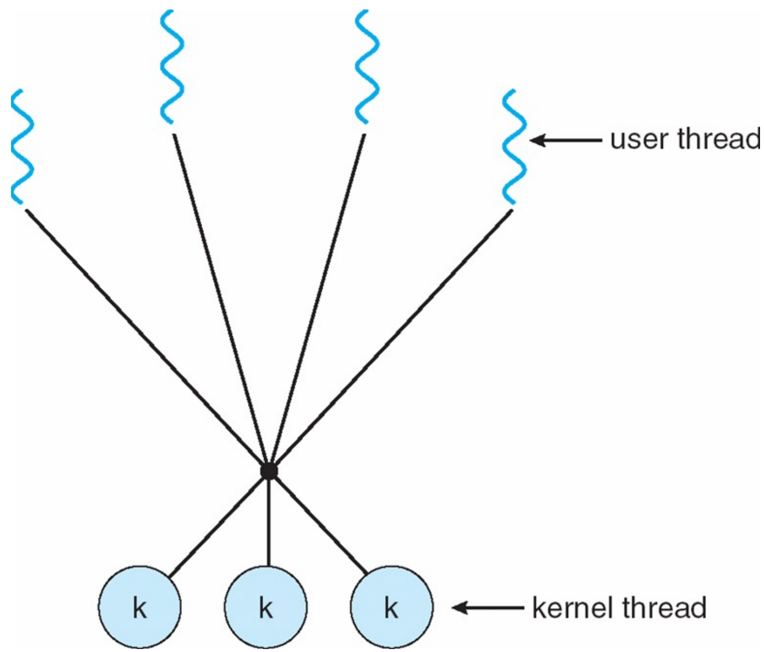
One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



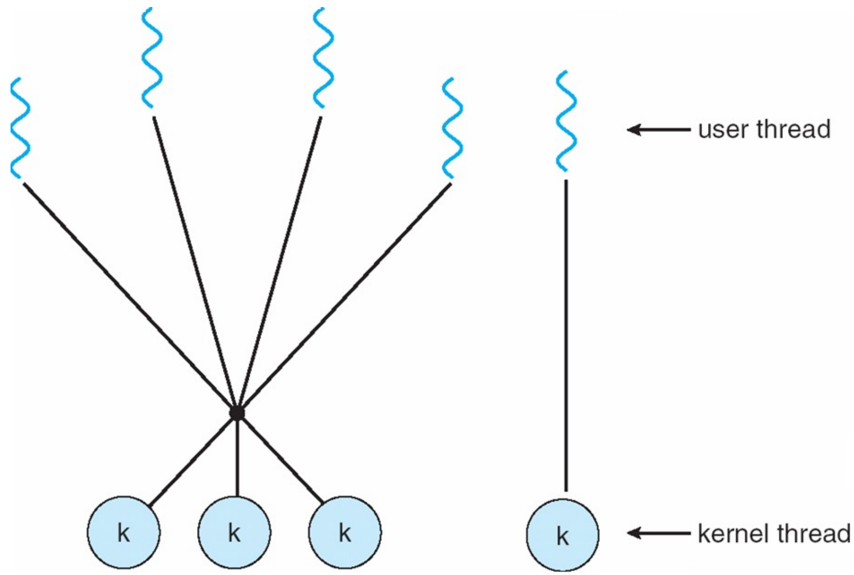
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



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

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
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - **Extending** Thread class
  - **Implementing** the Runnable interface

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Threading Issues-Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
- One that duplication all threads – the child thread does not call exec() after forking
- Only the thread that invoked the fork() system call is duplicated – exec() is called immediately after forking

Threading Issues-Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately – it is troublesome if a thread to be canceled is in the middle of updating shared data
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled – allow threads to be canceled safely

Threading Issues-Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to **the thread** to which the signal applies
  - Deliver the signal to **every thread** in the process
  - Deliver the signal to **certain threads** in the process
  - Assign **a specific thread** to receive all signals for the process

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly **faster** to service a request with an existing thread than create a new thread
  - Allows **the number of threads in the application(s) to be bound to the size of the pool**

Thread Specific Data

- Allows each thread to have its own copy of data
- Useful **when you do not have control over the thread creation process** (i.e., when using a thread pool)

Scheduler Activations

- Both M:M and Two-level models require communication **to maintain the appropriate number of kernel threads** allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

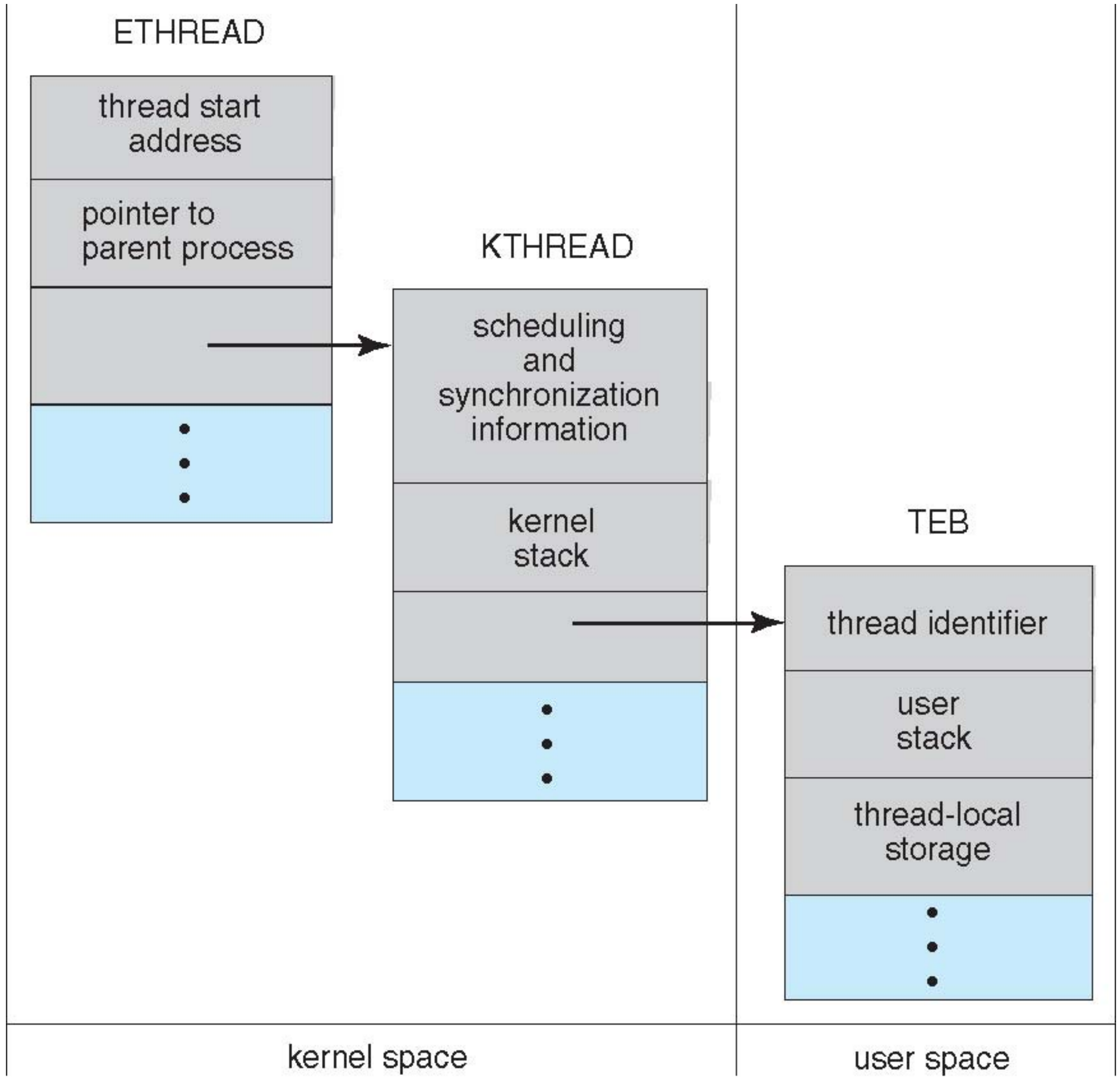
Operating System Examples

- A lightweight process (LWP) – an intermediate data structure
- To the user thread, it is a virtual processor that schedule a user thread to run.
- Each LWP is attached to a kernel thread, and OS schedules kernel thread to run.
- Example:
  - Windows XP Threads
  - Linux Thread, not distinguish b/w processes and threads

Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block) - LWP
  - TEB (thread environment block)

Windows XP Threads



Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

Linux Threads

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

End of Chapter 14