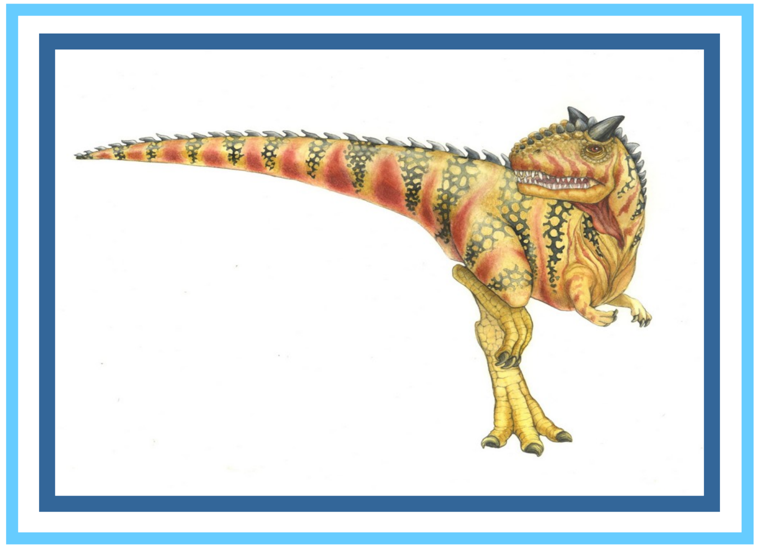
Threads



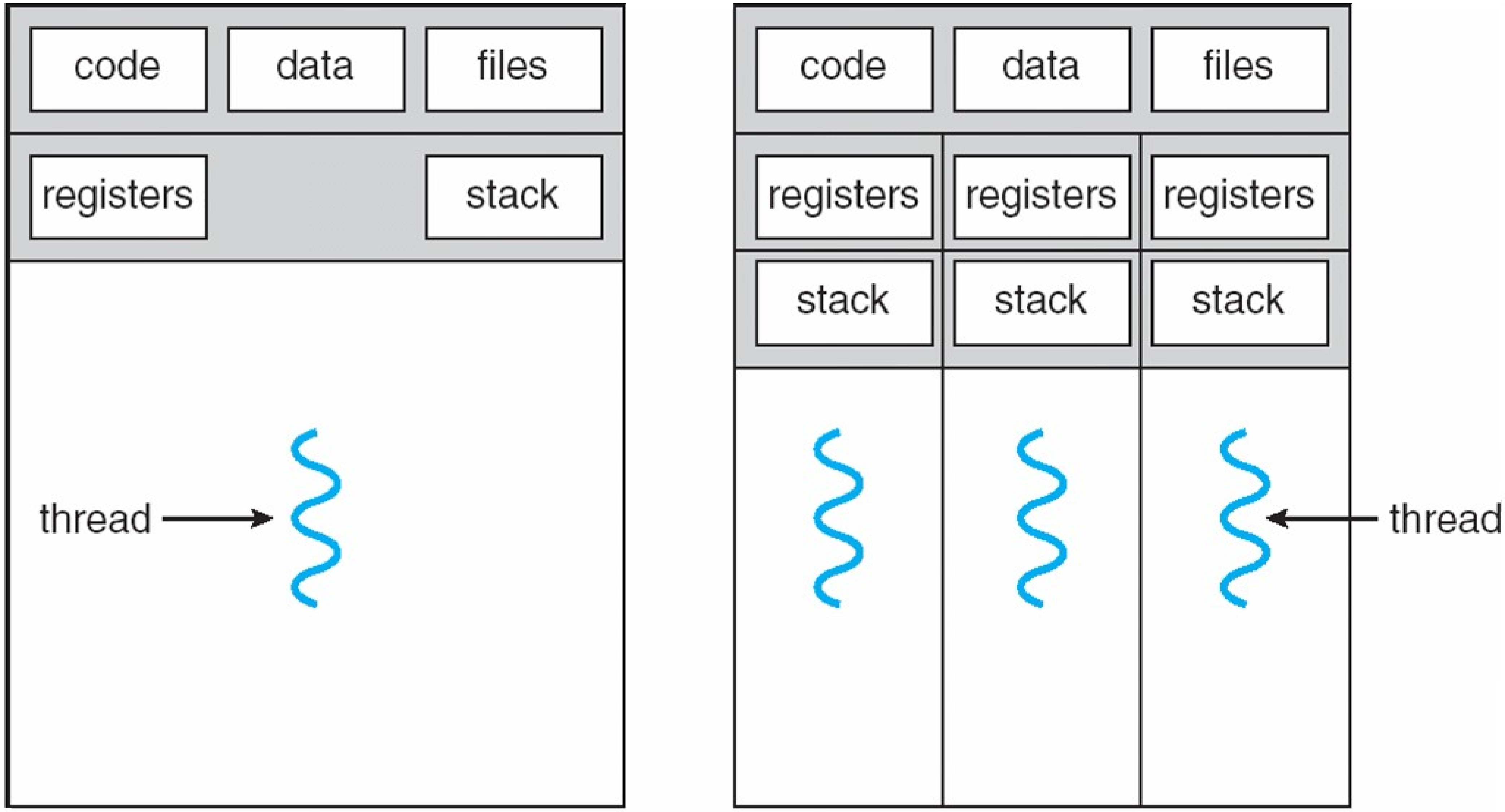
- A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

- A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

- A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

- Each thread belongs to exactly one process and no thread can exist outside a process.
- Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server.
- They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.

Single and Multithreaded Processes



single-threaded process

multithreaded process

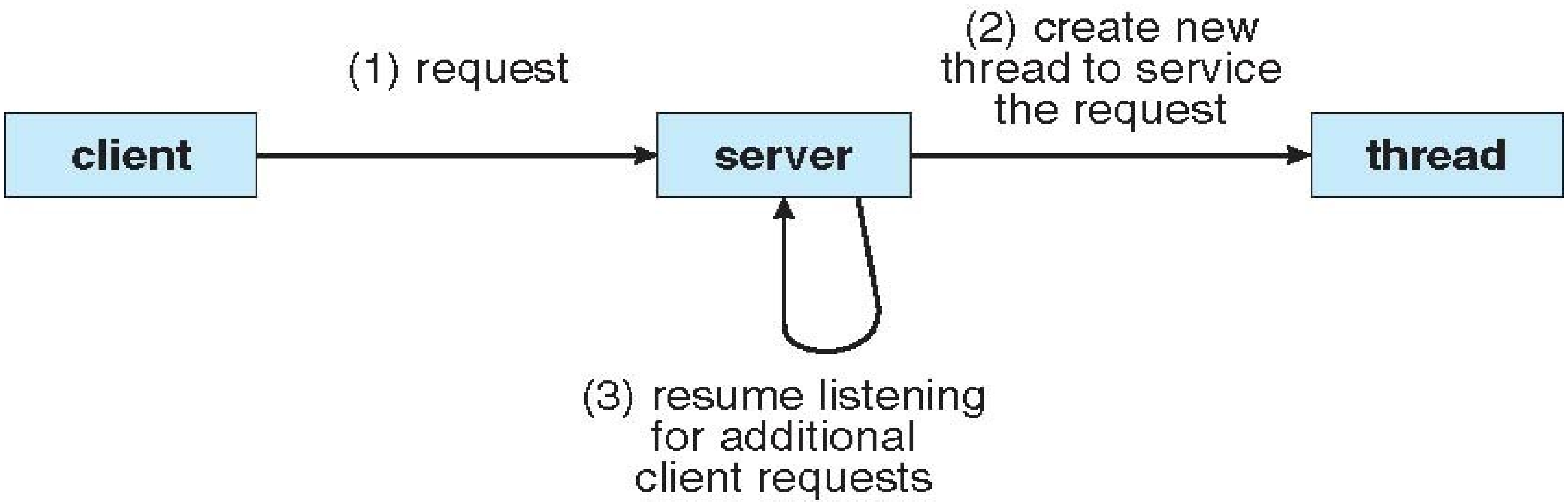
Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability

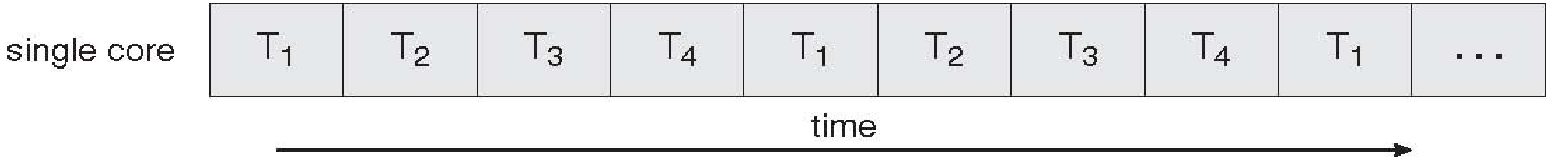
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

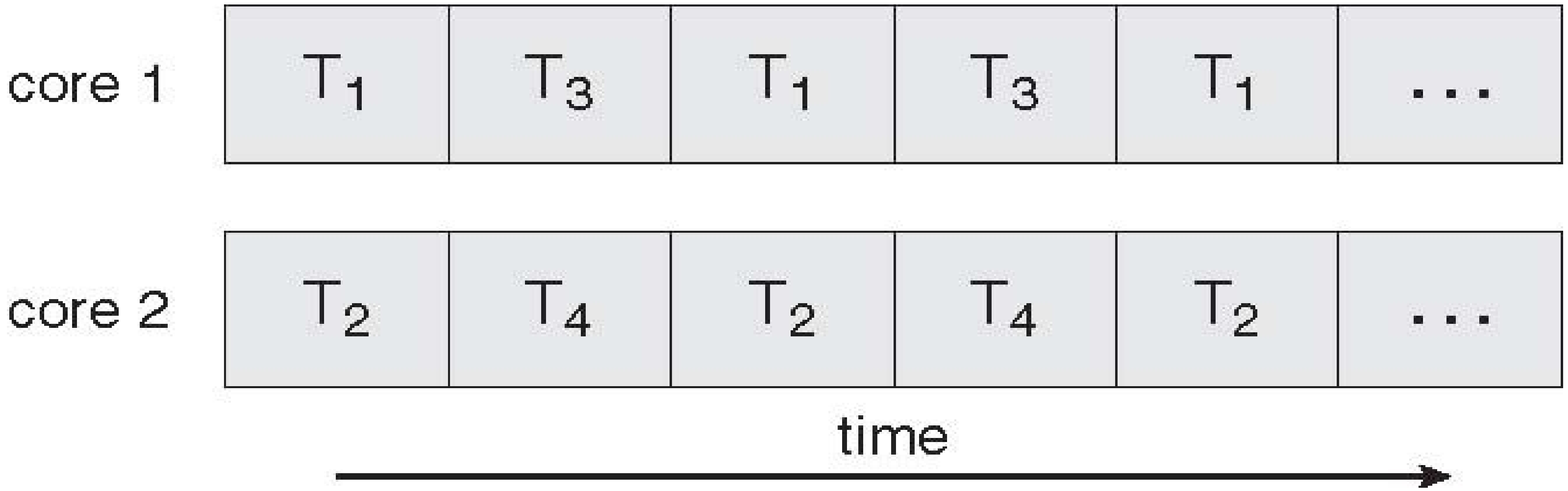
Multithreaded Server Architecture



Concurrent Execution on a Single-core System

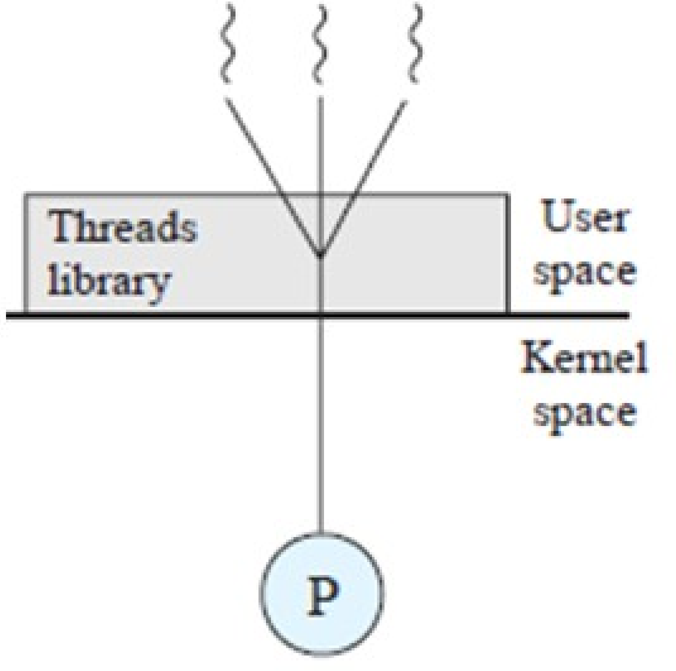


Parallel Execution on a Multicore System



User Threads

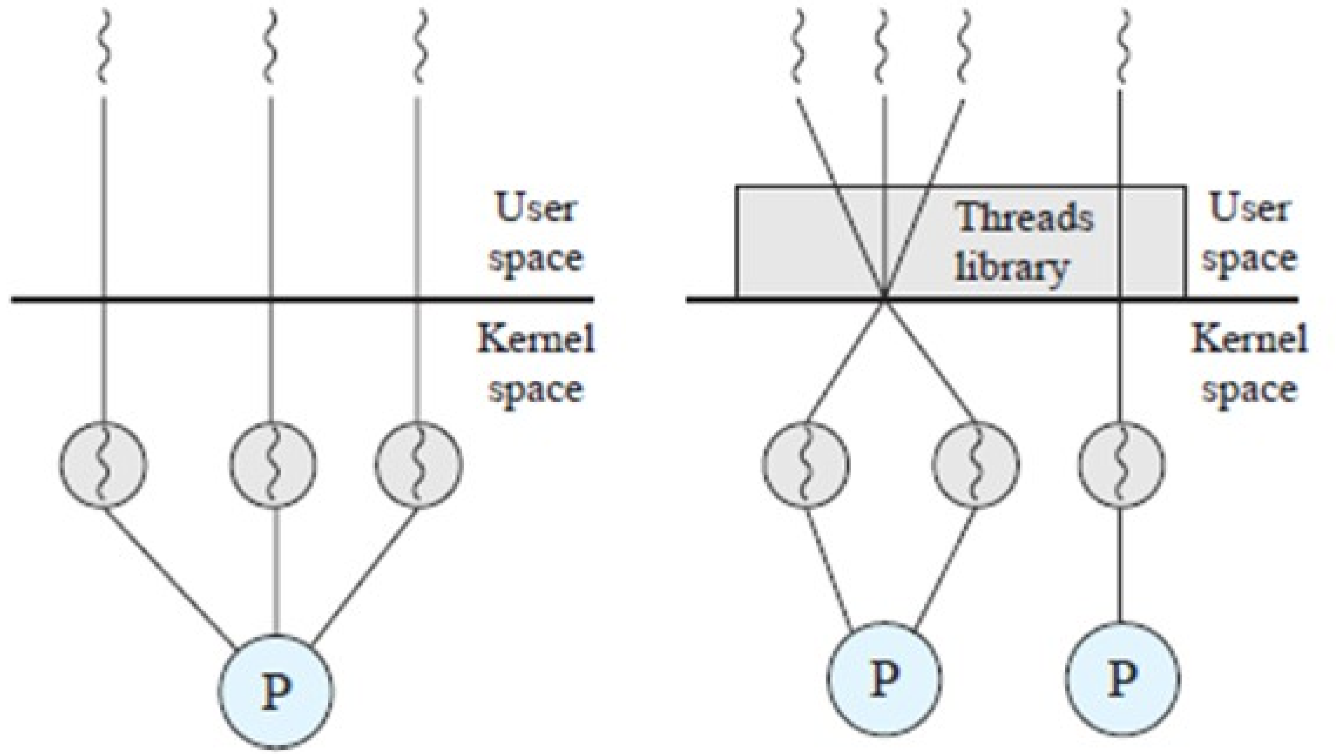
- Thread management done by user-level threads library
- All of the thread management is done by the application and the kernel is not aware of the existence of threads.
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- **Advantages of user level threads:**
  - There are a number of advantages to the use of ULTs instead of KLTs
  - Thread switching does not require kernel mode privileges
  - Scheduling can be application specific
  - ULTs can run on any OS.



- **Disadvantages of user level threads:**
  - When a ULT executes a blocking system call, not only is that thread blocked, but also all of the threads within the process are blocked.
  - A Multithreaded application cannot take advantage of multiprocessing. A kernel assigns one process to only one processor at a time.

Kernel Threads

- Supported by the Kernel
- There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility.
- The kernel maintains context information for the process as a whole and for individual threads within the process.
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X



- **Advantages of Kernel level threads:**
  - The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
  - If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- **Disadvantages of kernel level threads:**
  - The transfer of control from one thread to another within the same process requires a mode switch to the kernel.

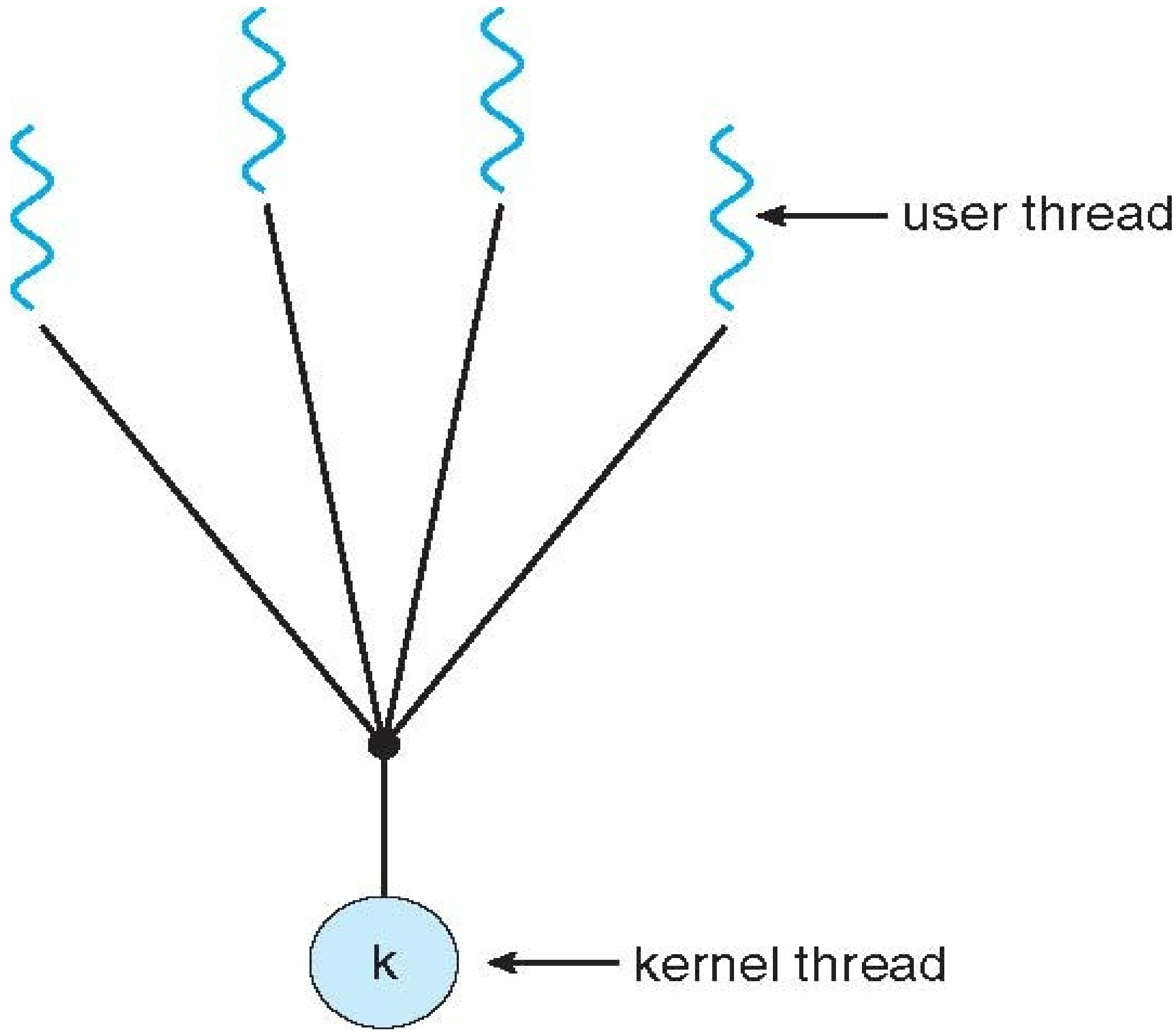
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

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

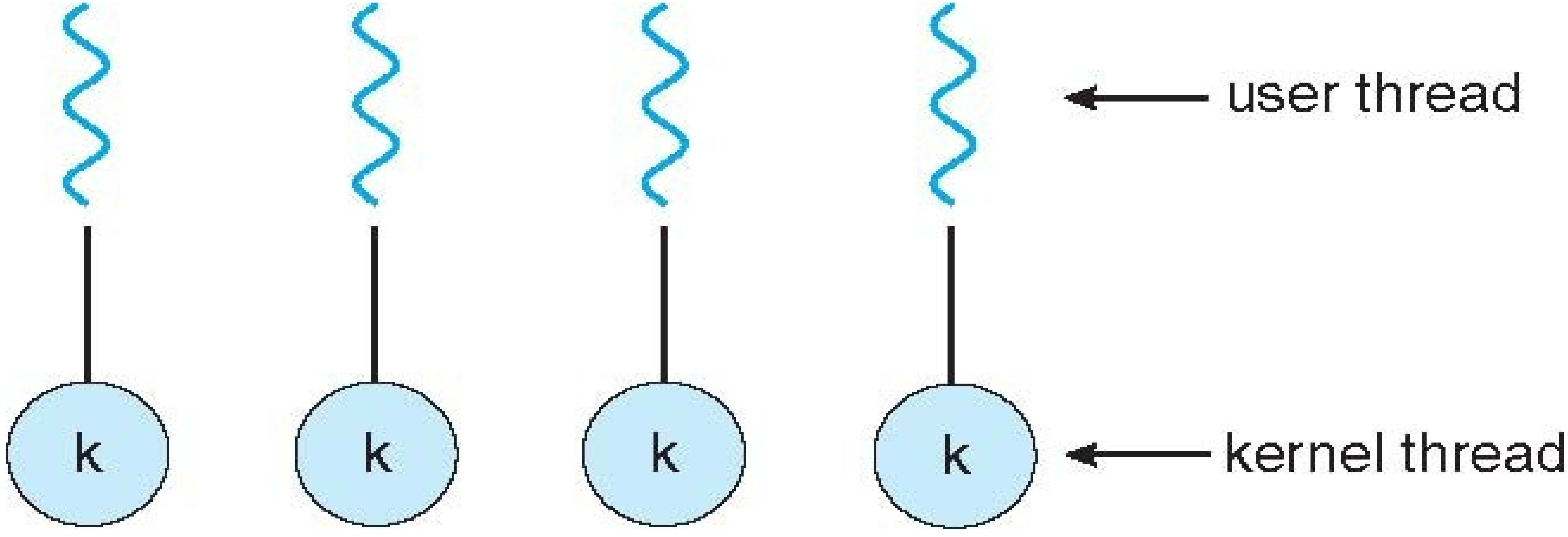
Many-to-One Model



One-to-One

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

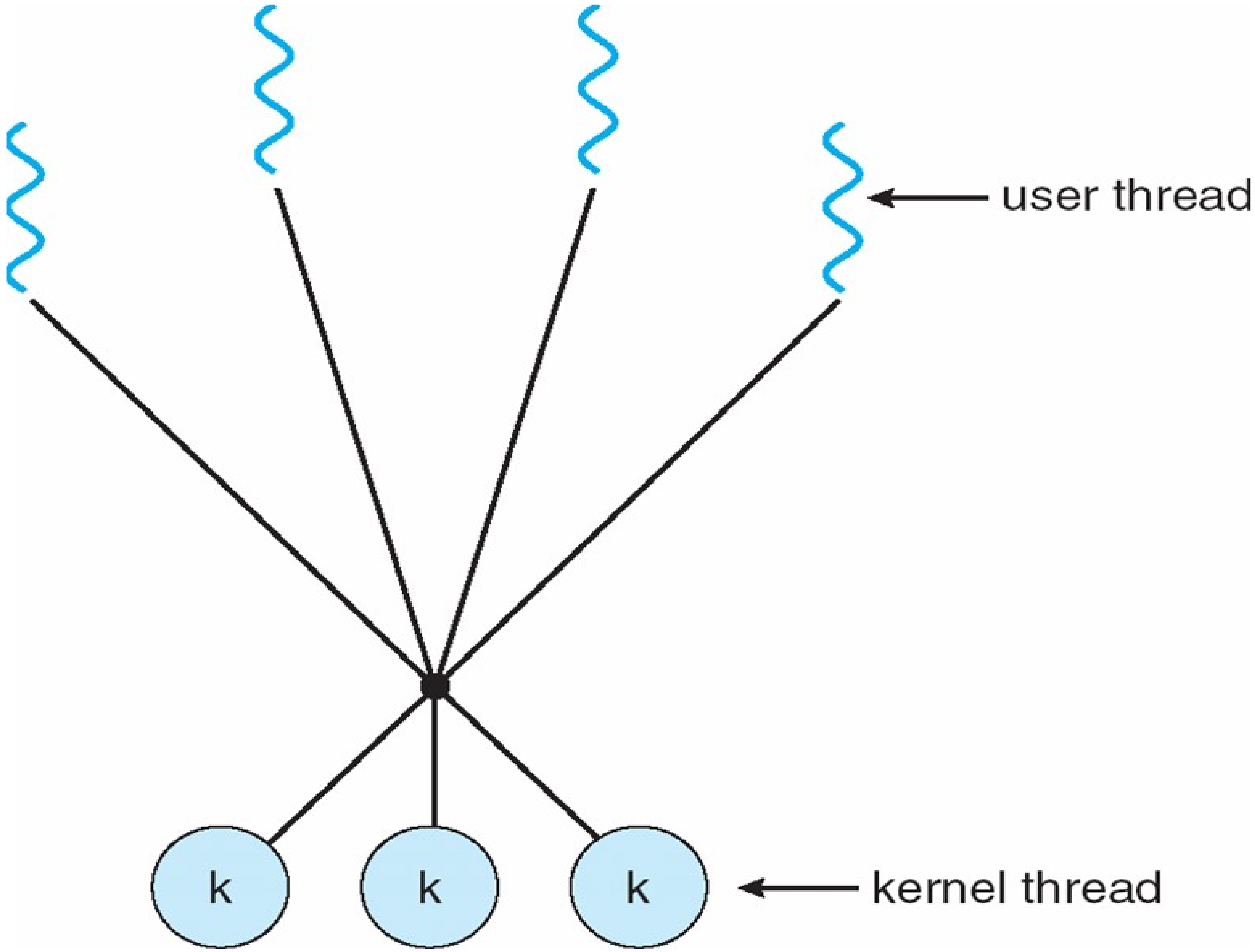
One-to-one Model



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

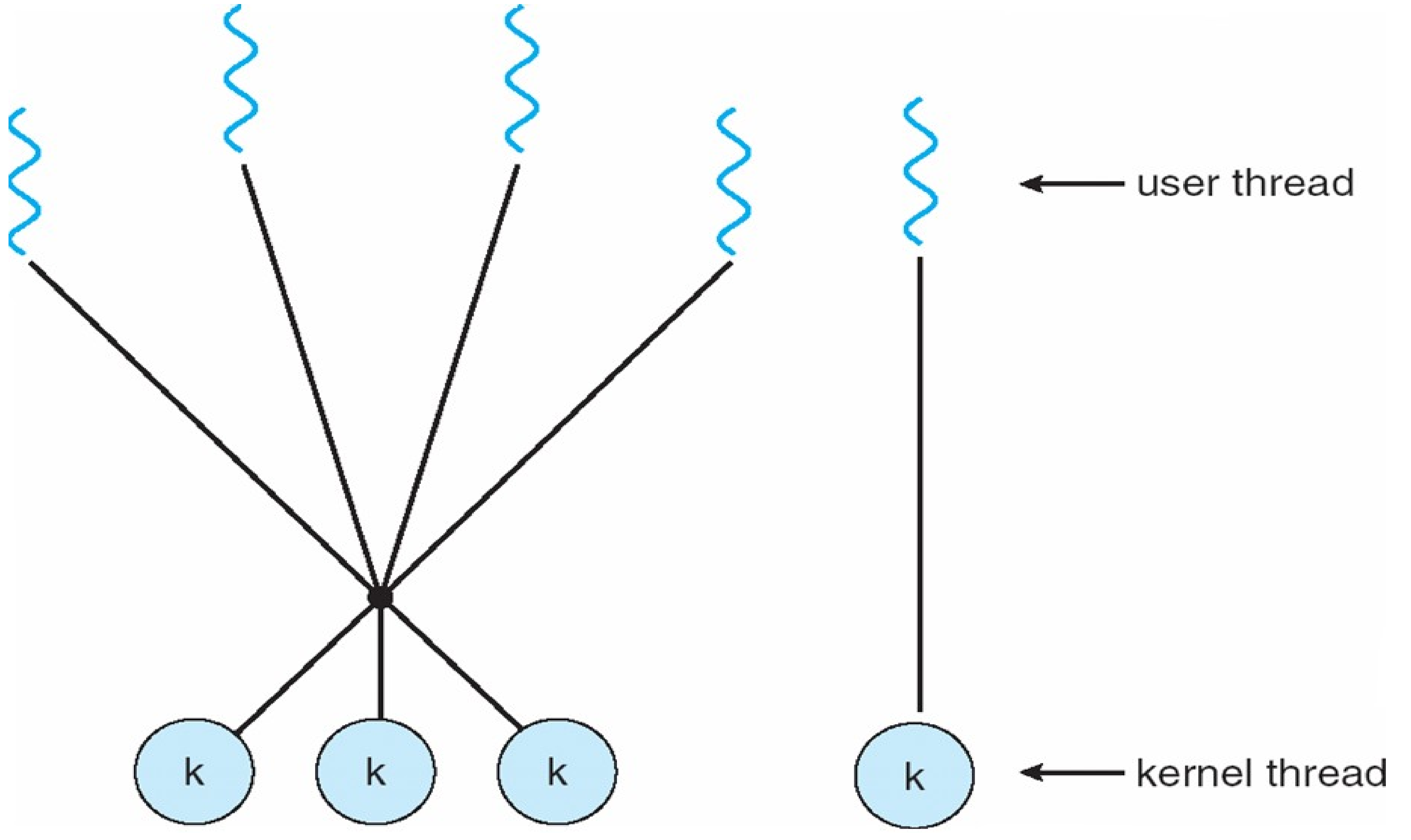
Many-to-Many Model



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

Two-level Model



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

Comparison Between Process and Threads

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Why Pthreads

- The primary motivation for using Pthreads is to realize potential program performance gains.
- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.
- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, other threads can perform CPU intensive work.
- Priority/real-time scheduling: tasks that are more important can be scheduled to supersede or interrupt lower priority tasks.
- Multi-threaded applications will work on a uni-processor system; yet naturally take advantage of a multiprocessor system, without recompiling.

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
  - Synchronous and asynchronous

Threading Issues (Cont.)

- **Thread pools**
- **Thread-specific data**
  - Create Facility needed for data private to thread
- **Scheduler activations**

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
- exec() - replace the running process including all threads

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately.
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.

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

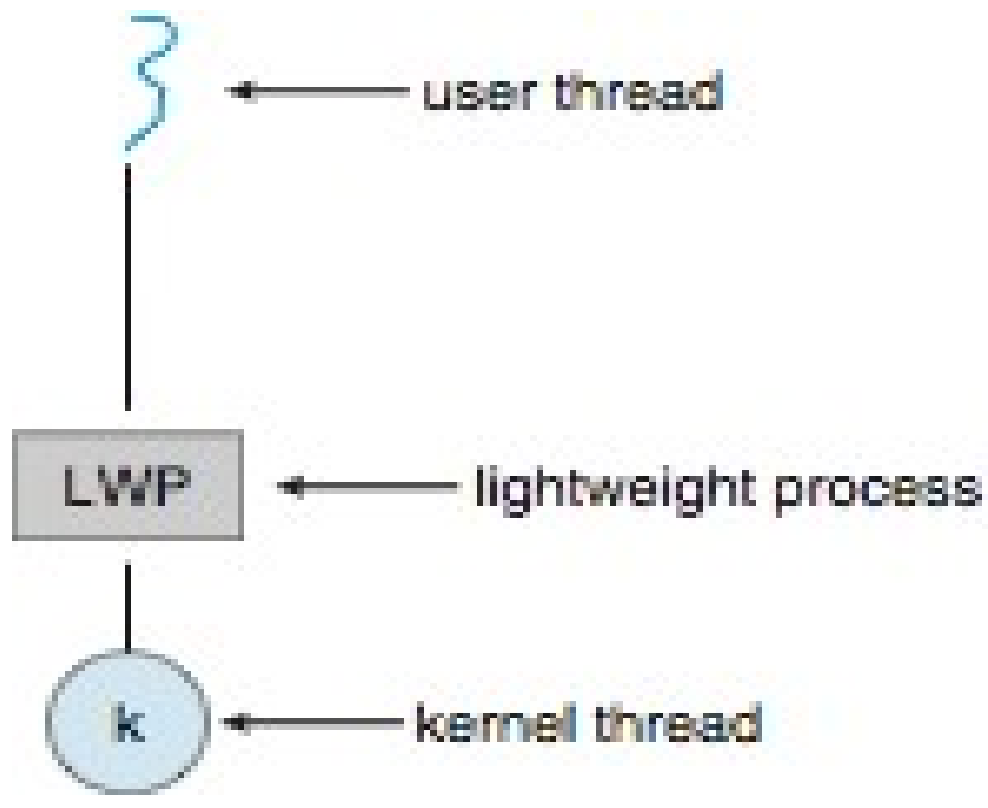
Thread Specific Data

- 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

Lightweight Processes



Operating System Examples

- Windows XP Threads
- Linux Thread

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)
- `struct task_struct` points to process data structures (shared or unique)

Linux Threads

- `fork()` and `clone()` system calls
- Doesn't distinguish between process and thread
  - Uses term *task* rather than thread
- `clone()` takes options to determine sharing on process create
- `struct task_struct` points to process data structures (shared or unique)

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.