Threads

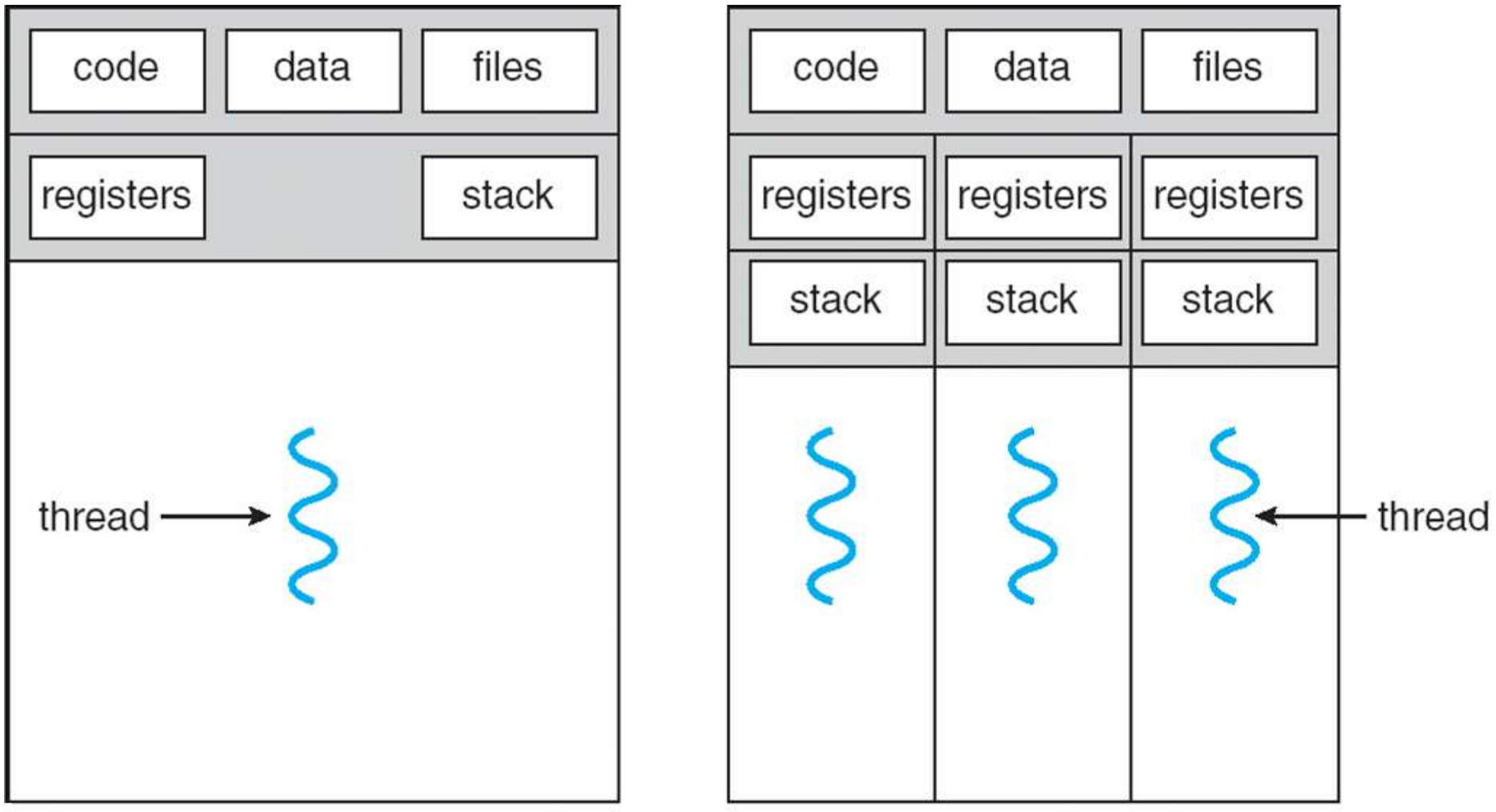
Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues

Threads

- 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.
- Thread is a basic units of CPU Utilization process.

Single and Multithreaded Processes



single-threaded process

multithreaded process

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Benefits

- **Responsiveness**
- Program responsiveness allows a program to run even if part of it is blocked using multithreading. This can also be done if the process is performing a lengthy operation. For example - A web browser with multithreading can use one thread for user contact and another for image loading at the same time.

- **Resource Sharing**
- All the threads of a process share its resources such as memory, data, files etc. A single application can have different threads within the same address space using resource sharing.

Benefits

- **Utilization of Multiprocessor Architecture**
- In a multiprocessor architecture, each thread can run on a different processor in parallel using multithreading. This increases concurrency of the system. This is in direct contrast to a single processor system, where only one process or thread can run on a processor at a time.
- **Economy**
- It is more economical to use threads as they share the process resources. Comparatively, it is more expensive and time-consuming to create processes as they require more memory and resources. The overhead for process creation and management is much higher than thread creation and management.

Types of Thread

- Threads are implemented in following two ways
- **User Level Threads** – Usermanaged threads.
- **Kernel Level Threads** – Operating System managed threads acting on kernel, an operating system core.
- Ultimately their must exist relationship between user threads and kernel threads.

User Threads

- Thread management done by user-level threads library.
- the thread management kernel is not aware of the existence of threads.
- The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

Kernel Threads

- Supported by the Kernel.
- thread management is done by the Kernel. There is no thread management code in the application area.
- Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

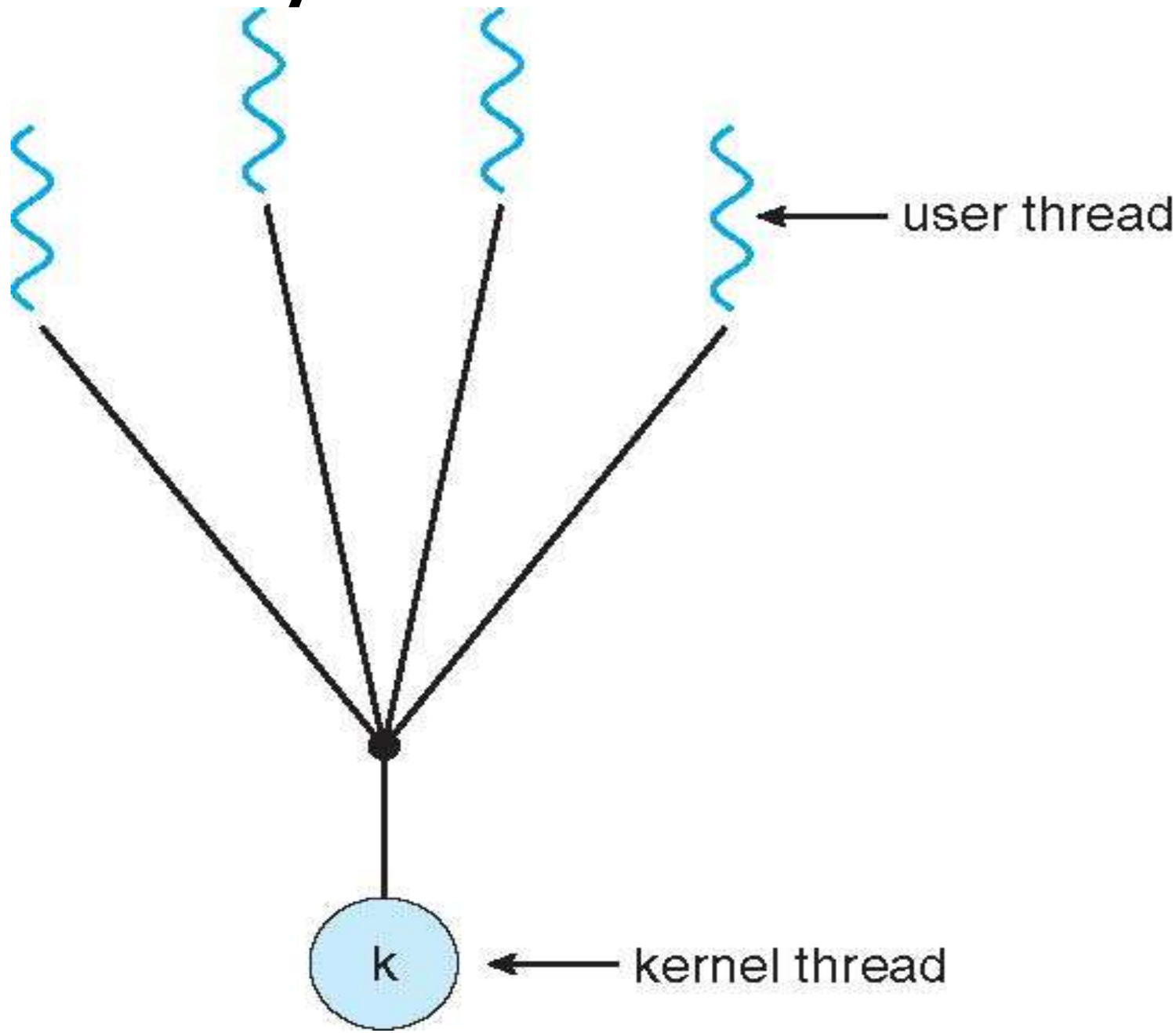
Multithreading Models

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

Many-to-One

- Many user-level threads mapped to single kernel thread
- Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library.
- When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

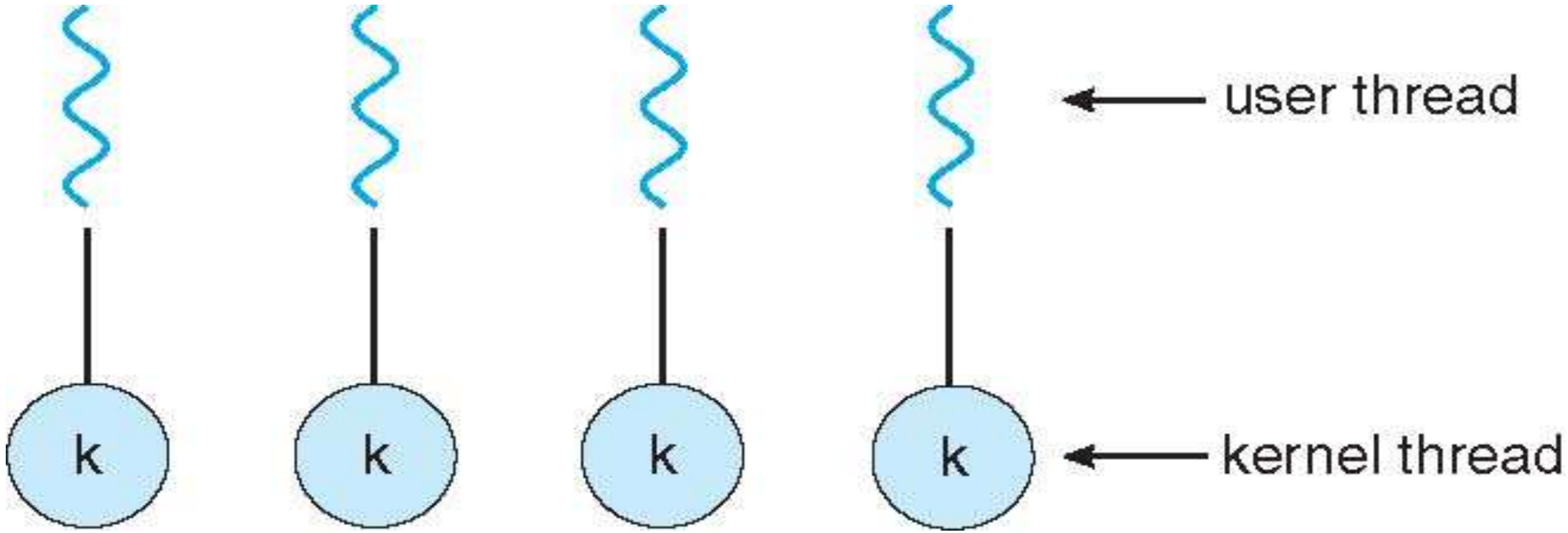
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model.
- It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

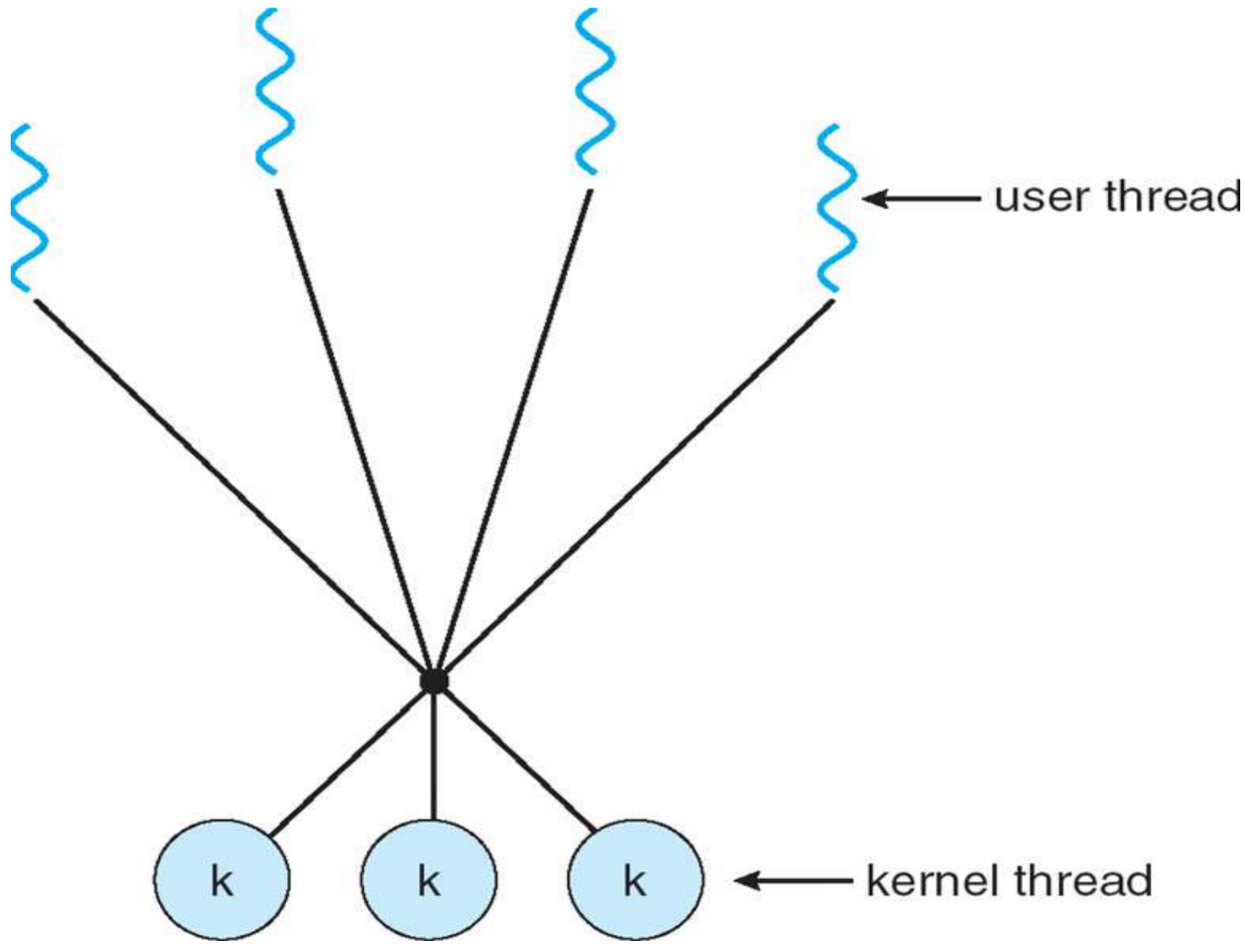
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.
- n this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9

Many-to-Many 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

Thread Issues

- **Thread Cancellation**

Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is **Asynchronous cancellation**, which terminates the target thread immediately. The other is **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. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all, or a single thread.

Thread Issues

- **Security Issues**

There can be security issues because of extensive sharing of resources between multiple threads.

- **fork() and exec () System Call**

fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in Multithreaded process is, if one thread forks, will the entire process be copied.