Inter Process Communication and Threads

Tahira Alam
University of Asia Pacific

Inter Process Communication (IPC)

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data

Inter Process Communication (IPC)(2)

- Processes do not share any memory with each other
- Some processes might want to work together for a task, so need to communicate information
- IPC mechanisms to share information between processes

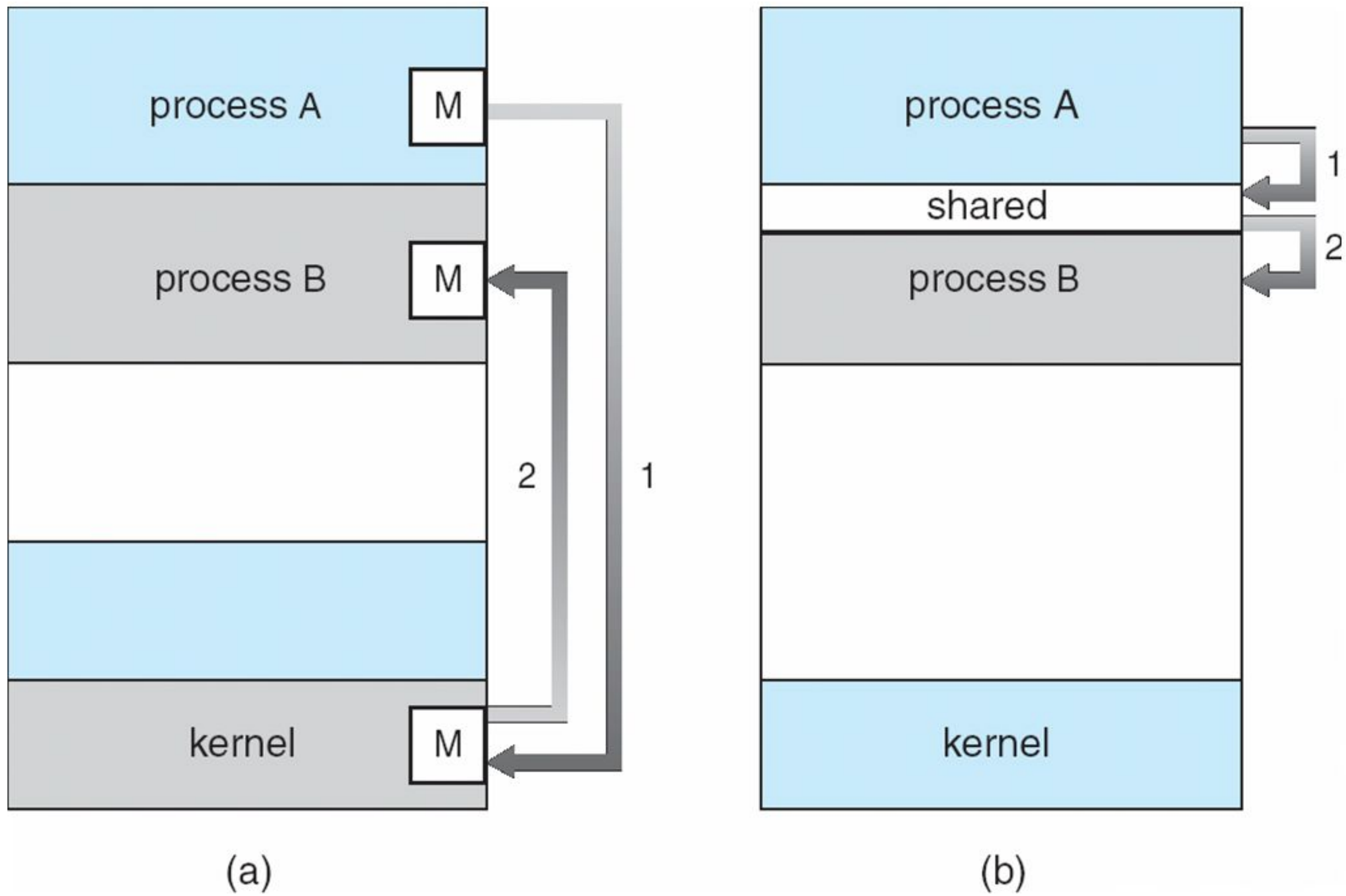
Reasons for IPC

- Information sharing
- Computation speedup
- Modularity
- Convenience

Two models of IPC

– Shared memory
– Message passing

Communications Models



Shared Memory

- Processes can both access same region of memory via shmget() system call
- int shmget (key_t key, int size, int shmflg)
- By providing same key, two processes can get same segment of memory
- Can read/write to memory to communicate
- Need to take care that one is not overwriting other's data: how?

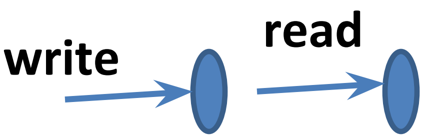
Signals

- A certain set of signals supported by OS
  - Some signals have fixed meaning (e.g., signal to terminate process)
  - Some signals can be user-defined
- Signals can be sent to a process by OS or another process (e.g., if you type Ctrl+C, OS sends SIGINT signal to running process)
- Signal handler: every process has a default code to execute for each signal
  - Exit on terminate signal
- Some signal handlers can be overridden to do other things

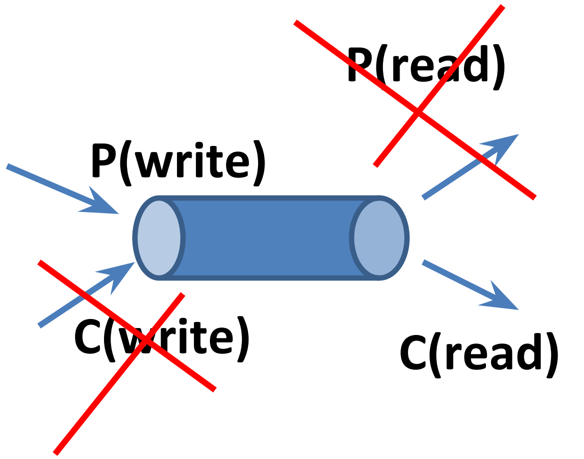
Sockets

- Sockets can be used for two processes on same machine or different machines to communicate
  - TCP/UDP sockets across machines
  - Unix sockets in local machine
- Communicating with sockets
  - Processes open sockets and connect them to each other
  - Messages written into one socket can be read from another
  - OS transfers data across socket buffers

Pipes



- Pipe system call returns two file descriptors
  - Read handle and write handle
  - A pipe is a half-duplex communication
  - Data written in one file descriptor can be read through another
- Regular pipes: both fd are in same process (how it is useful?)
  - Parent and child share fd after fork
  - Parent uses one end and child uses other end
- Named pipes: two endpoints of a pipe can be in different processes
- Pipe data buffered in OS buffers between write and read



Message Queues

- Mailbox abstraction
- Process can open a mailbox at a specified location
- Processes can send/receive messages from mailbox
- OS buffers messages between send and receive

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server

Blocking vs. non-blocking communication

- Some IPC actions can block
  - Reading from socket/pipe that has no data, or reading from empty message queue
  - Writing to a full socket/pipe/message queue
- The system calls to read/write have versions that block or can return with an error code in case of failure
  - A socket read can return error indicating no data to be read, instead of blocking

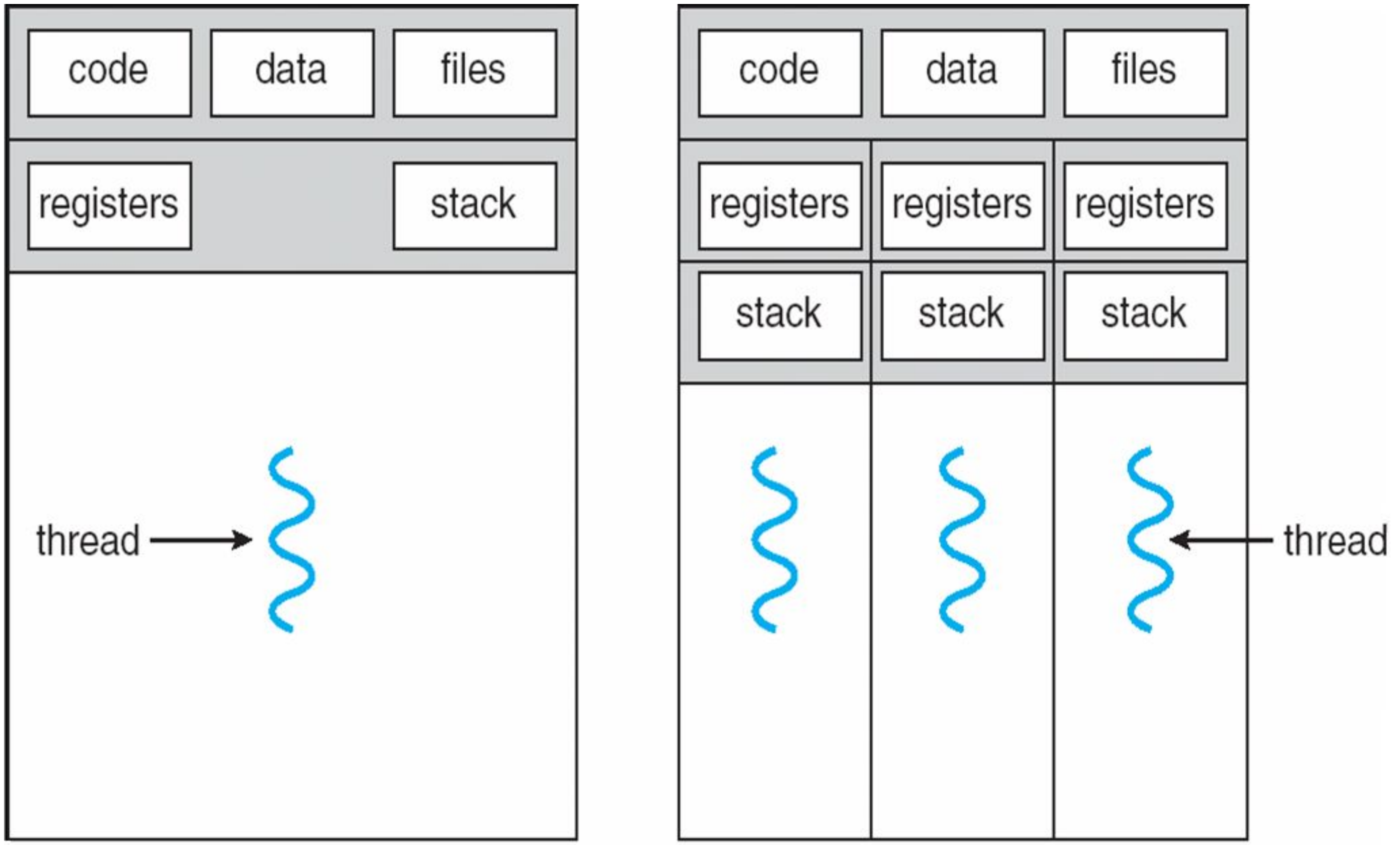
Now,

Threads

Threads

- A **thread** is the smallest sequence of programmed instructions that can be managed independently by a OS scheduler.

Single and Multithreaded Processes



single-threaded process

multithreaded process

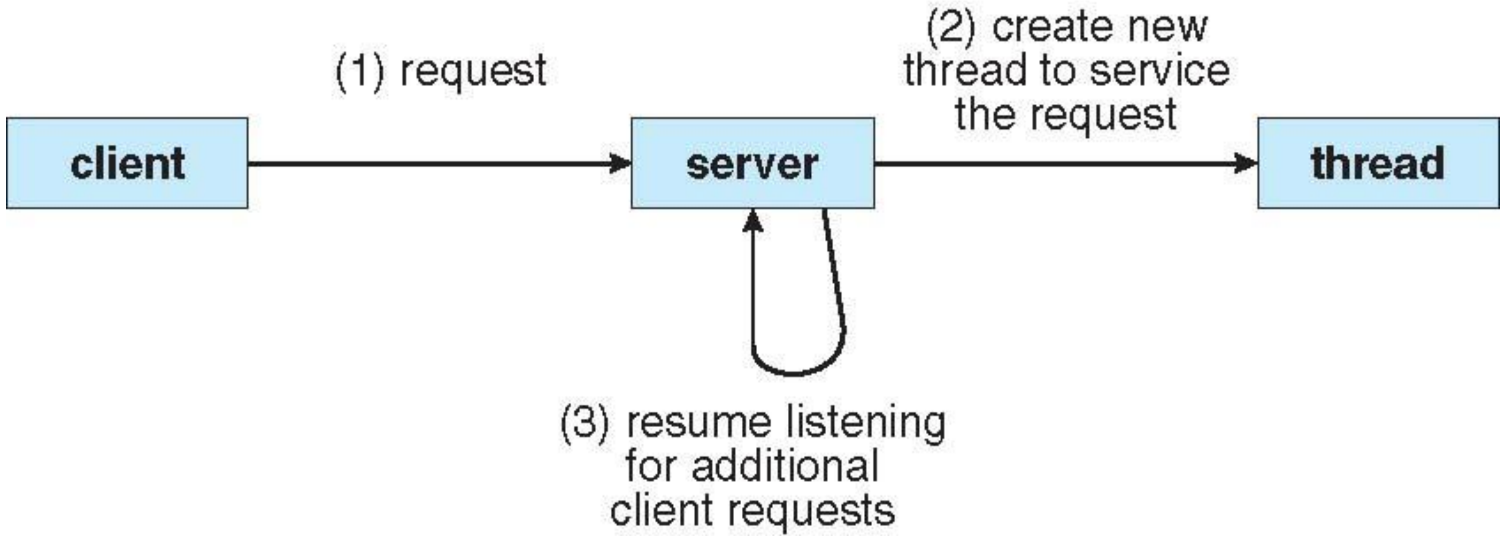
Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability

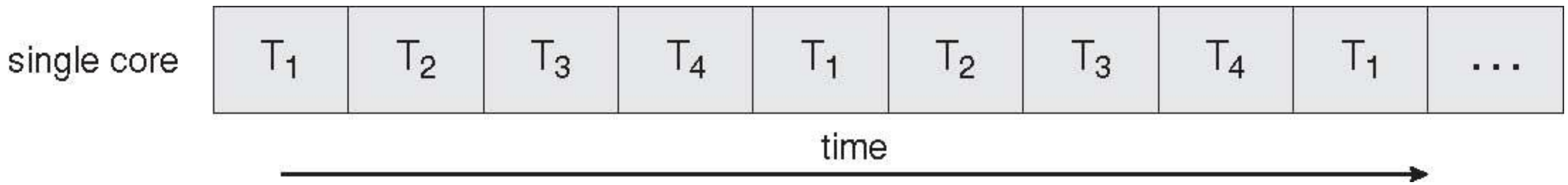
Multicore Programming

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

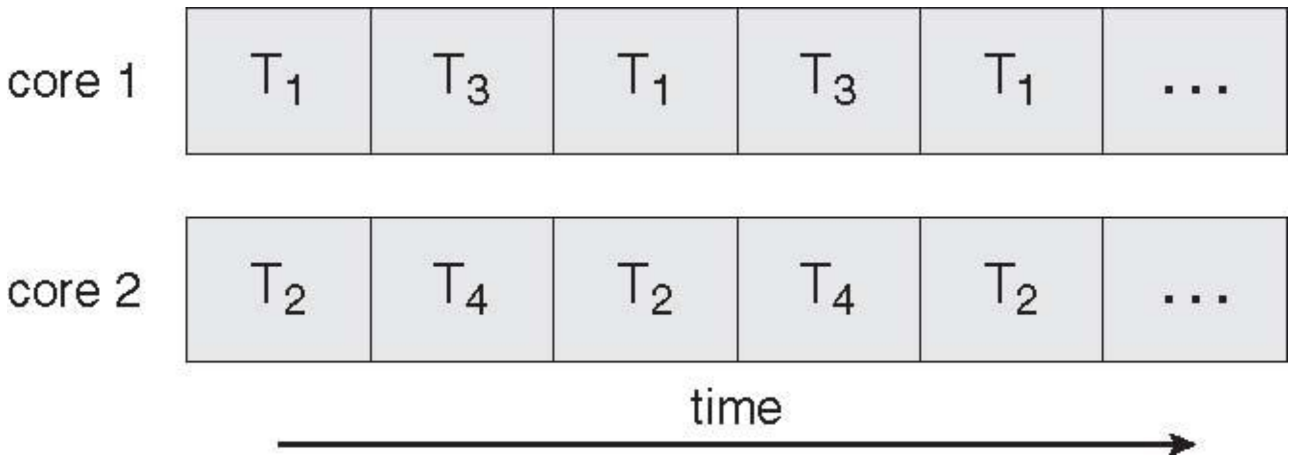
Multithreaded Server Architecture



Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



User Threads and Kernal Thread

- User Threads: Thread management done by user-level threads library
- Kernel Threads: Supported by the Kernel

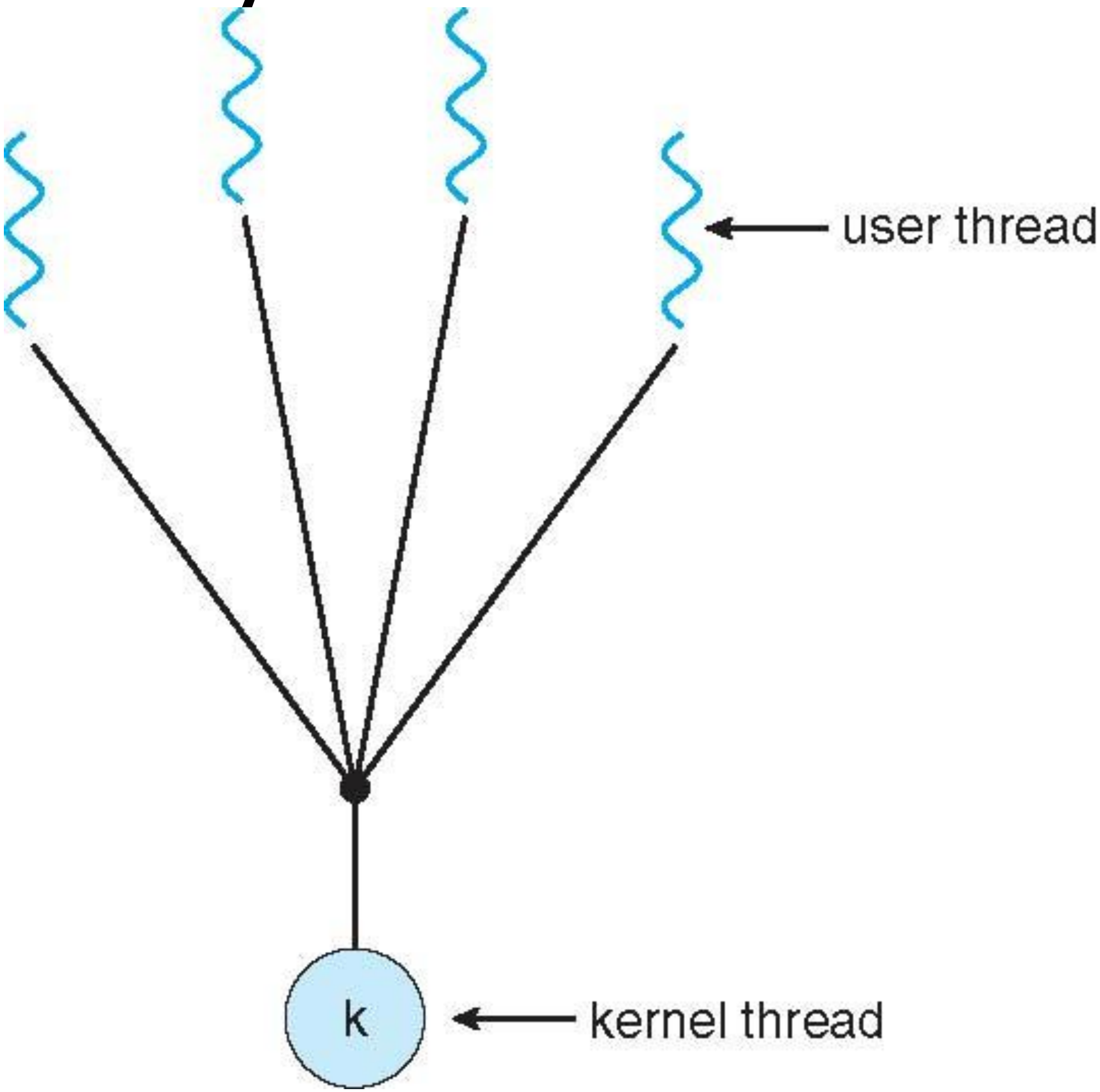
Multithreading Models

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

Many-to-One

- Many user-level threads mapped to single kernel thread

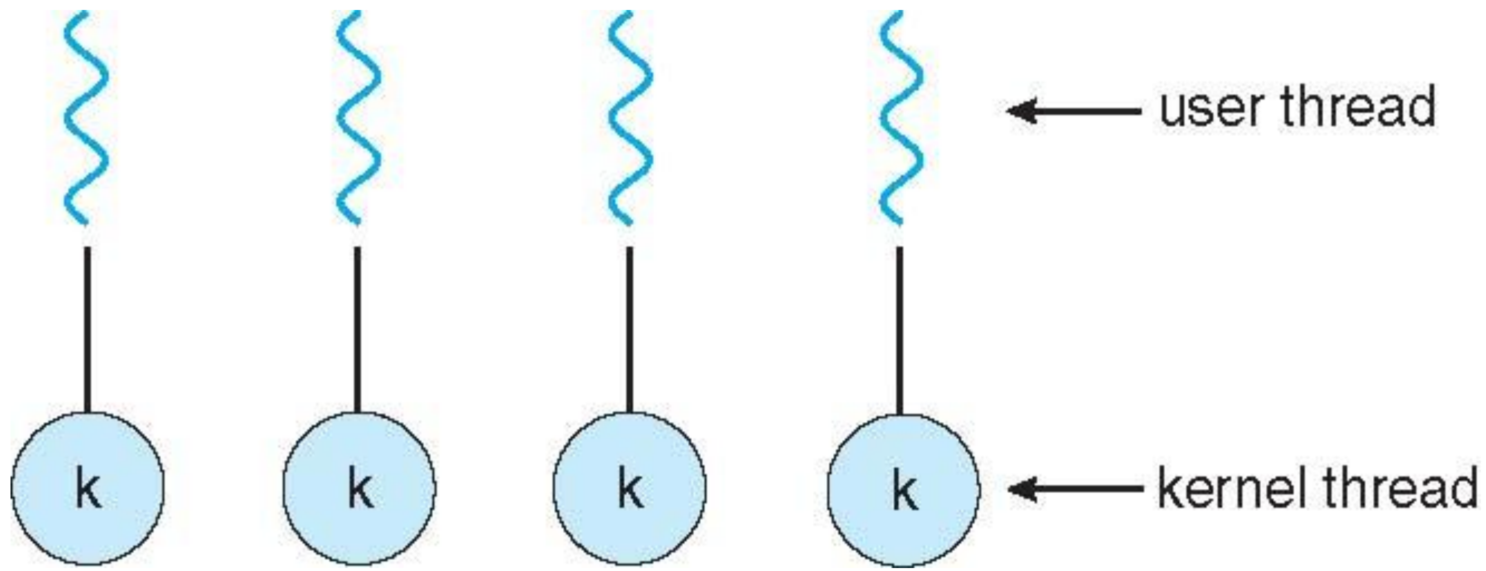
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread

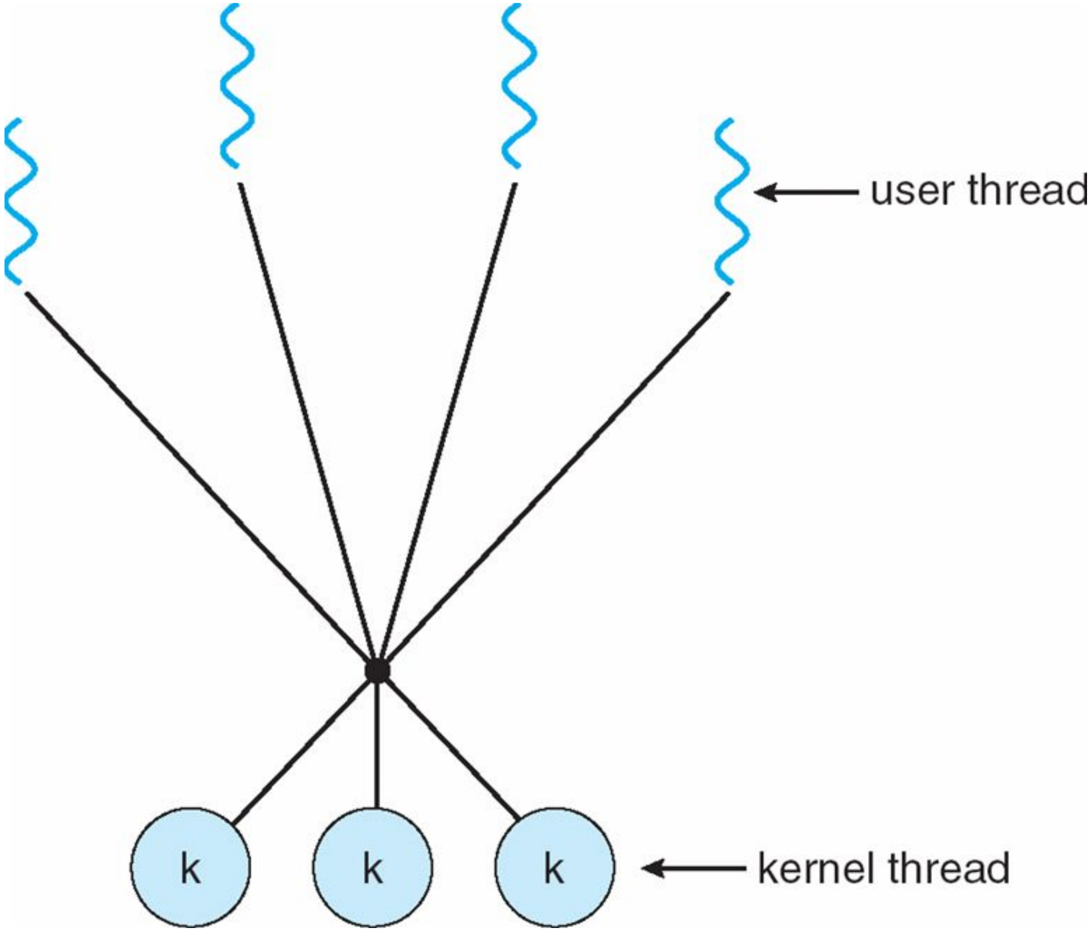
One-to-one Model



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

Many-to-Many Model



Thank you!