Threads

Threads

- A **thread** is a basic unit of CPU utilization
- We have considered only the single-threaded processes (heavyweight processes) so far
- We have not separated out a thread from a process
- However, a process can be multithreaded

Threads

- What is a thread?
- Why the multithreaded processes have a number of advantages in comparison with the single-threaded processes?
- How threads are organized?

Processes and Threads

- Process abstraction combines two concepts
  - Concurrency
    - Each process is a sequential execution stream of instructions
  - Protection
    - Each process defines an address space
    - Address space identifies all addresses that can be touched by the program

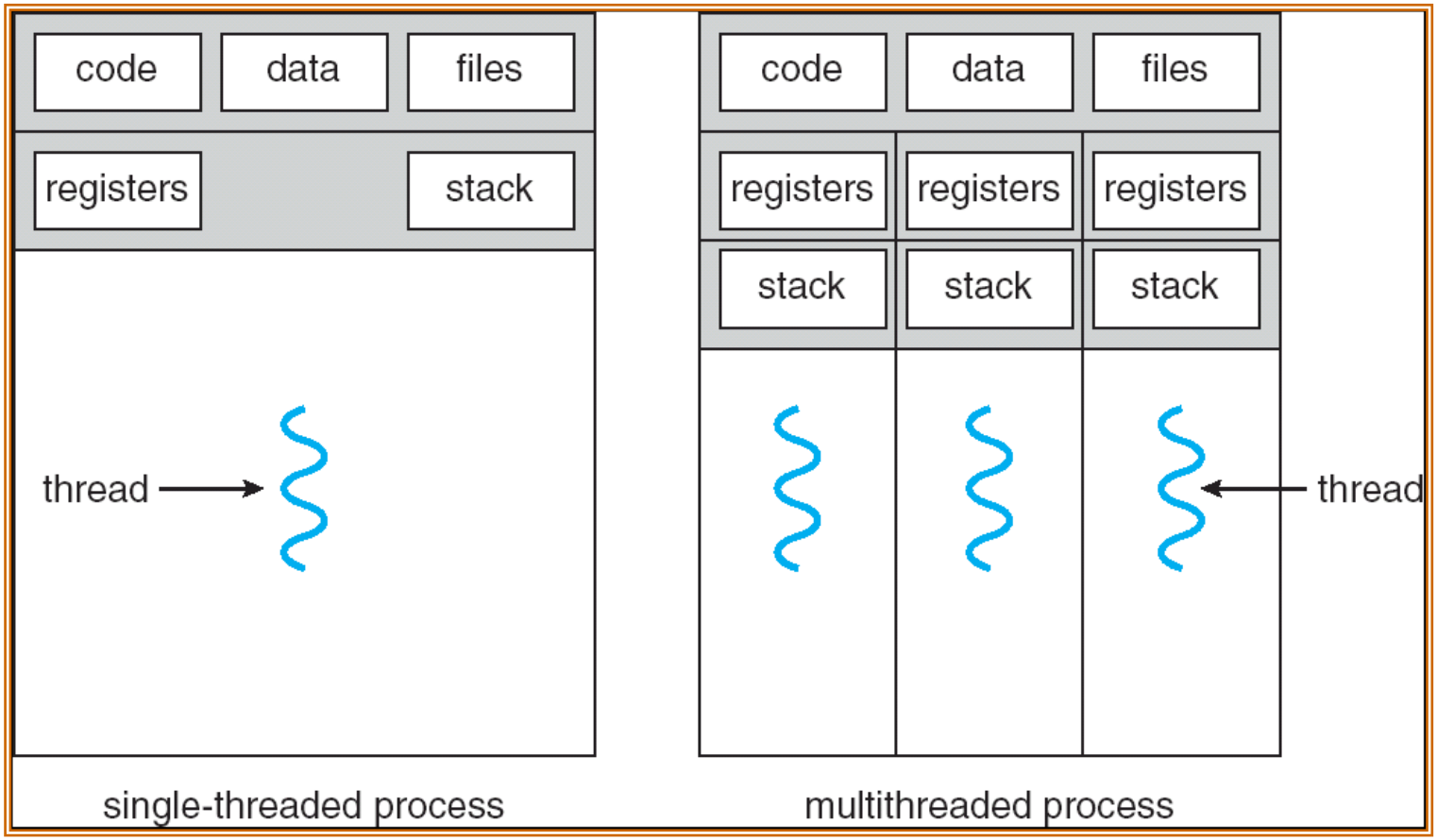
- **Threads**
  - **Key idea**: separate the concepts of concurrency from protection
  - A thread represents a sequential execution: stream of instructions
  - A process defines the address space that may be shared by multiple threads that **share code/data/heap**

Introducing Threads

- A **thread** represents an abstract entity that executes a sequence of instructions
  - **It has its own ID**
  - **It has its own Program Counter**
  - **It has its own set of CPU registers**
  - **It has its own stack**
  - There is no thread-specific heap or data segment, code section and other operating system resources (unlike process)

- Threads are lightweight
  - Creating a thread more efficient than creating a process.
  - Communication between threads easier than btw. processes.
  - Context switching between threads requires fewer CPU cycles and memory references than switching processes.
  - Threads only track a subset of process state (share list of open files, pid, …)

Single and Multithreaded Processes

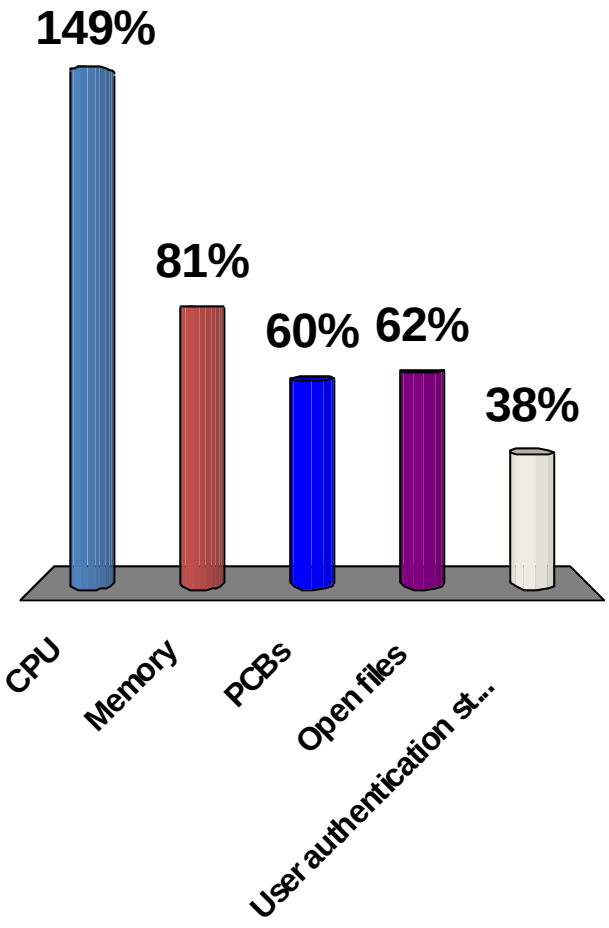


Benefits

- **Responsiveness**. Multithreading may allow a program to continue running even if part of it is blocked or is performing a lengthy operation.
- **Resource Sharing**. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
- **Economy**. Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Utilization of Multiprocessor Architectures**. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.

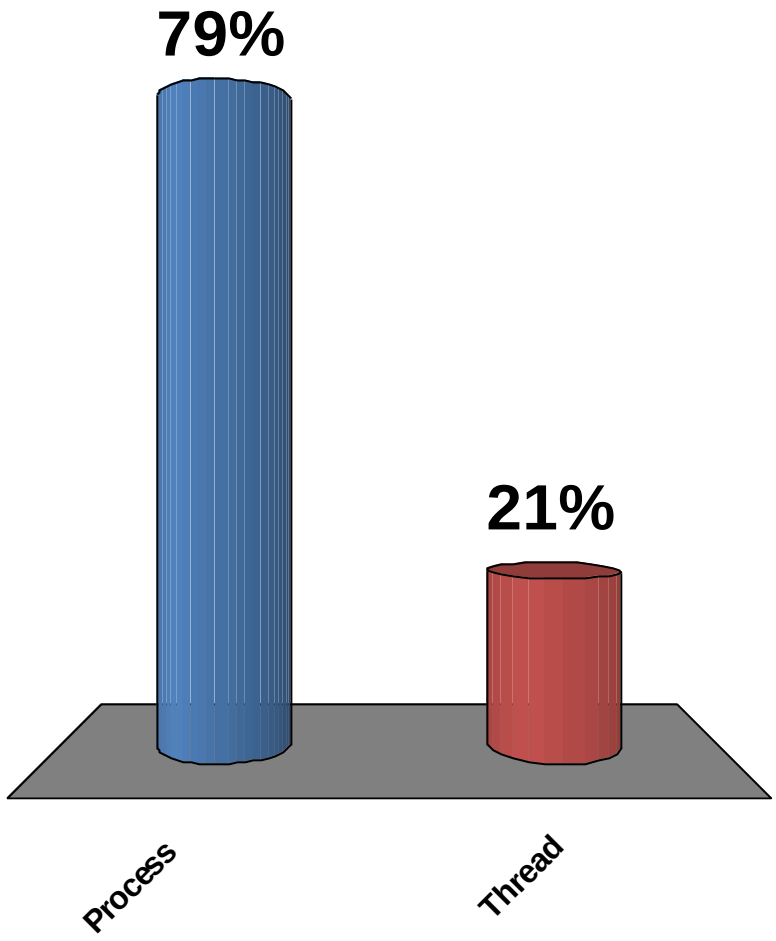
Threads allow you to multiplex which resources?

1. CPU
2. Memory
3. PCBs
4. Open files
5. User authentication structures



Context switch time for which entity is greater?

1. Process
2. Thread



Threads vs. Processes

Threads

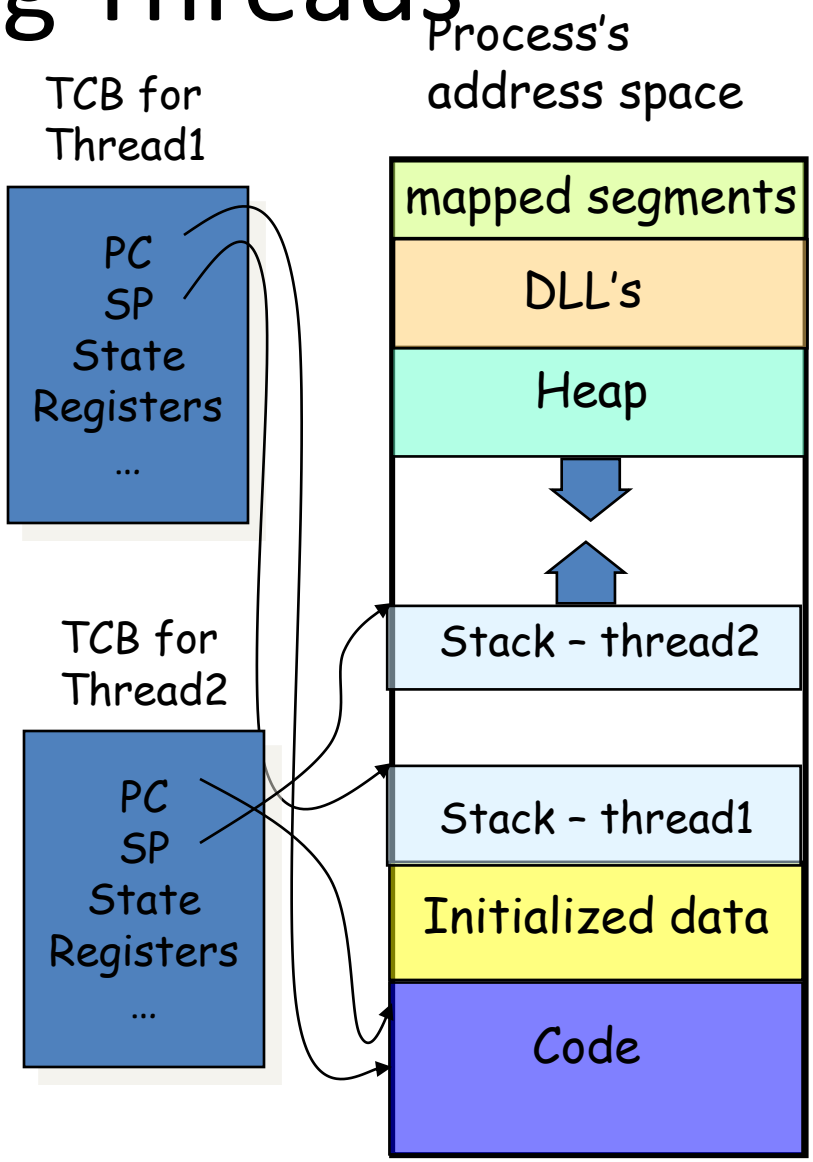
- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one thread in a process, the first thread calls main & has the process's stack
- Inexpensive creation
- Inexpensive context switching
- If a thread dies, its stack is reclaimed
- Inter-thread communication via memory.

Processes

- A process has code/data/heap & other segments
- There must be at least one thread in a process
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- Expensive creation
- Expensive context switching
- If a process dies, its resources are reclaimed & all threads die
- Inter-process communication via OS and data copying.

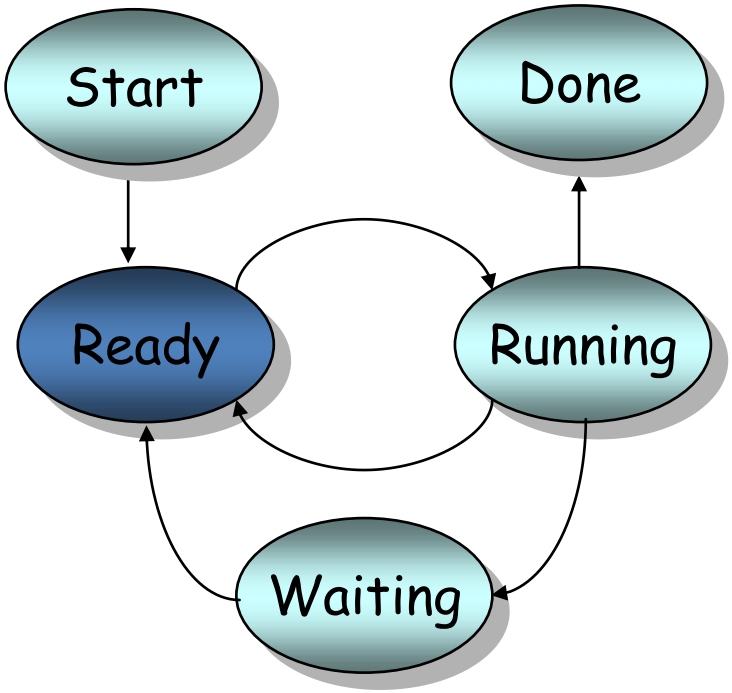
Implementing Threads

- Processes define an address space; threads share the address space
- Process Control Block (PCB) contains process-specific information
  - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- Thread Control Block (TCB) contains thread-specific information
  - Stack pointer, PC, thread state (running, …), register values, a pointer to PCB, …



Threads' Life Cycle

- Threads (just like processes) go through a sequence of *start*, *ready*, *running*, *waiting*, and *done* states



User and Kernel Threads

- Thread management done by user-level threads library above the kernel
- A thread library provides the programmer an API for creating and managing threads.
- Two approaches of implementing a thread library:
  - the library is in user space with no kernel support (all code and data structures for the library exist in the user space, above the kernel)
  - a kernel-level library supported directly by the operating system.
- Three primary thread libraries:
  - POSIX Pthreads (user- or kernel- level - UNIX, Linux)
  - Win32 threads (kernel-level - Windows)
  - Java threads (thread creation and management in Java programs)

Multithreading Models

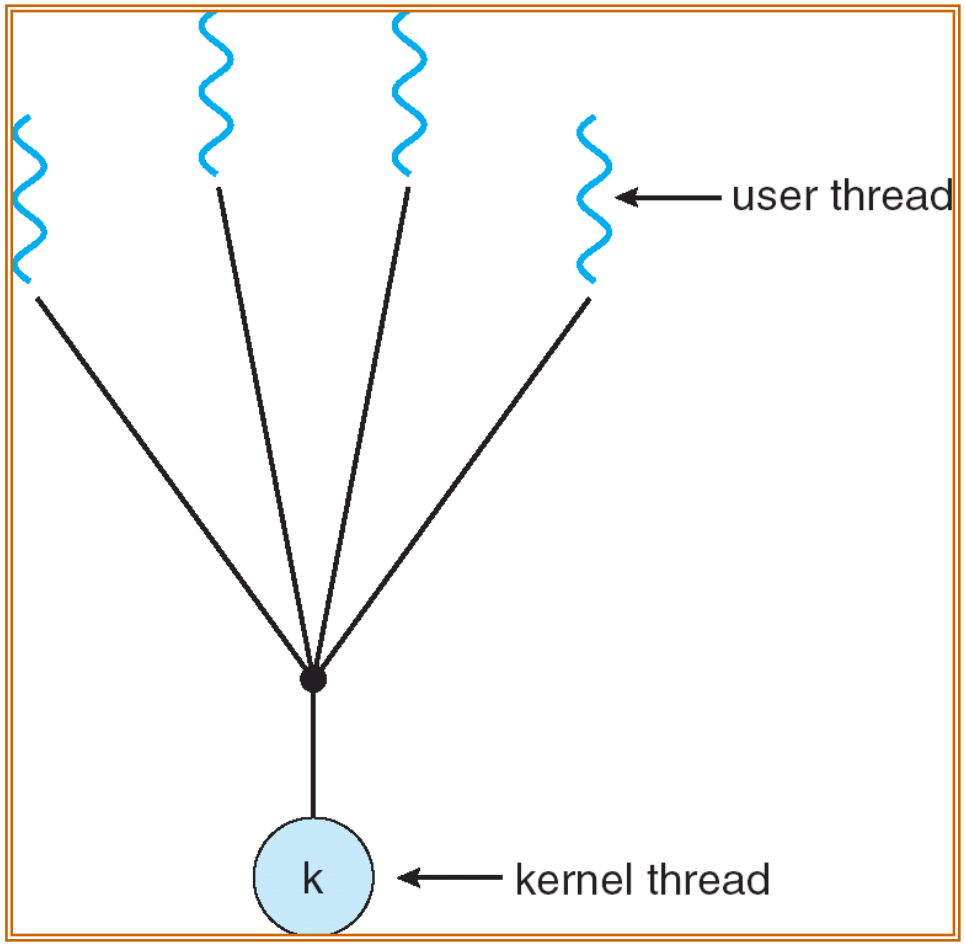
- A relationship between user threads and kernel threads

- Many-to-One model maps many user-level threads to one kernel thread. Multiple threads are unable to run in parallel, but the entire process will block if any thread makes a blocking system call. Since only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors
- One-to-One model maps each user thread to a separate kernel thread and allows multiple threads to run in parallel on multiprocessors and another thread to run when a thread makes a blocking system call
- Many-to-Many model multiplexes many user-level threads to a smaller or equal number of kernel threads that can run in parallel on a multiprocessor

Many-to-One

- Many user-level threads mapped to a single kernel thread
- Multiple threads are unable to run in parallel, but the entire process will block if any thread makes a blocking system call.
- Since only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

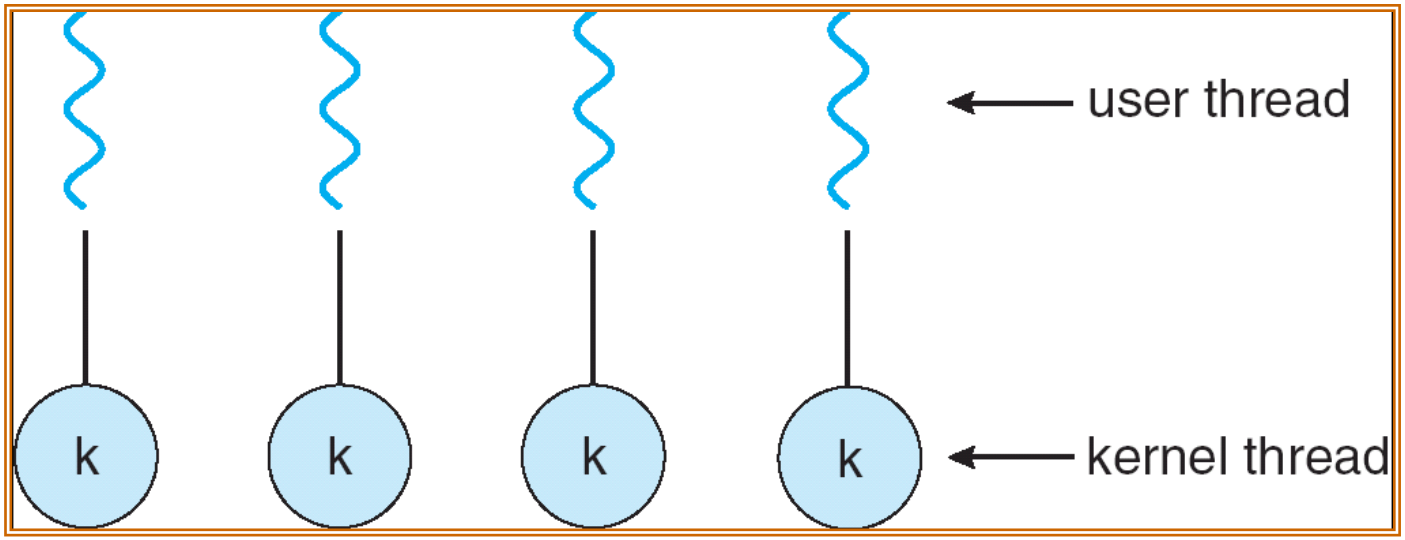
Many-to-One Model



One-to-One

- Each user-level thread maps to a separate kernel thread
- allows multiple threads to run in parallel on multiprocessors
- allows another thread to run when a thread makes a blocking system call
- **Drawback**: because the overhead of creating kernel threads can burden the performance of the application, most implementations of this model restrict the number of threads supported by the system
- Examples
  - Windows NT/XP/2000, 95, 98
  - Linux
  - Solaris 9 and later

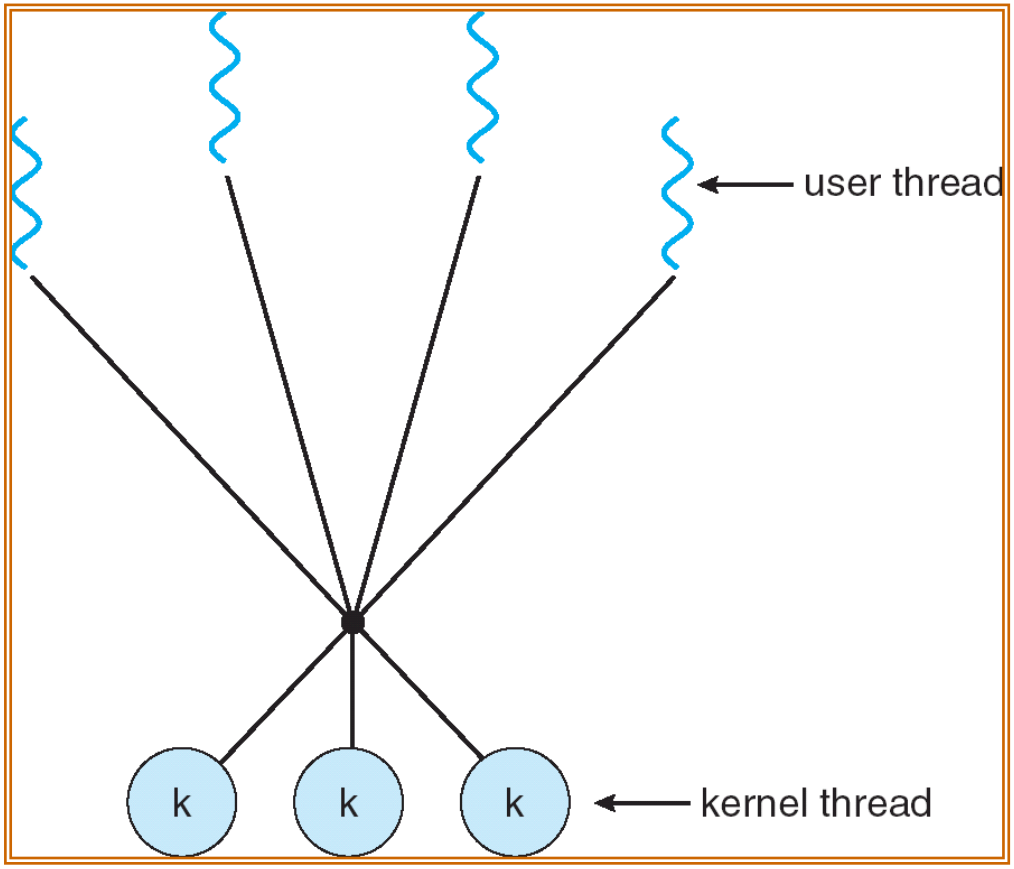
One-to-one Model



Many-to-Many Model

- Multiplexes many user-level threads to a smaller or equal number of kernel threads that can run in parallel on a multiprocessor
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- The number of kernel threads may be specific to either particular application or a particular computer
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

Many-to-Many Model



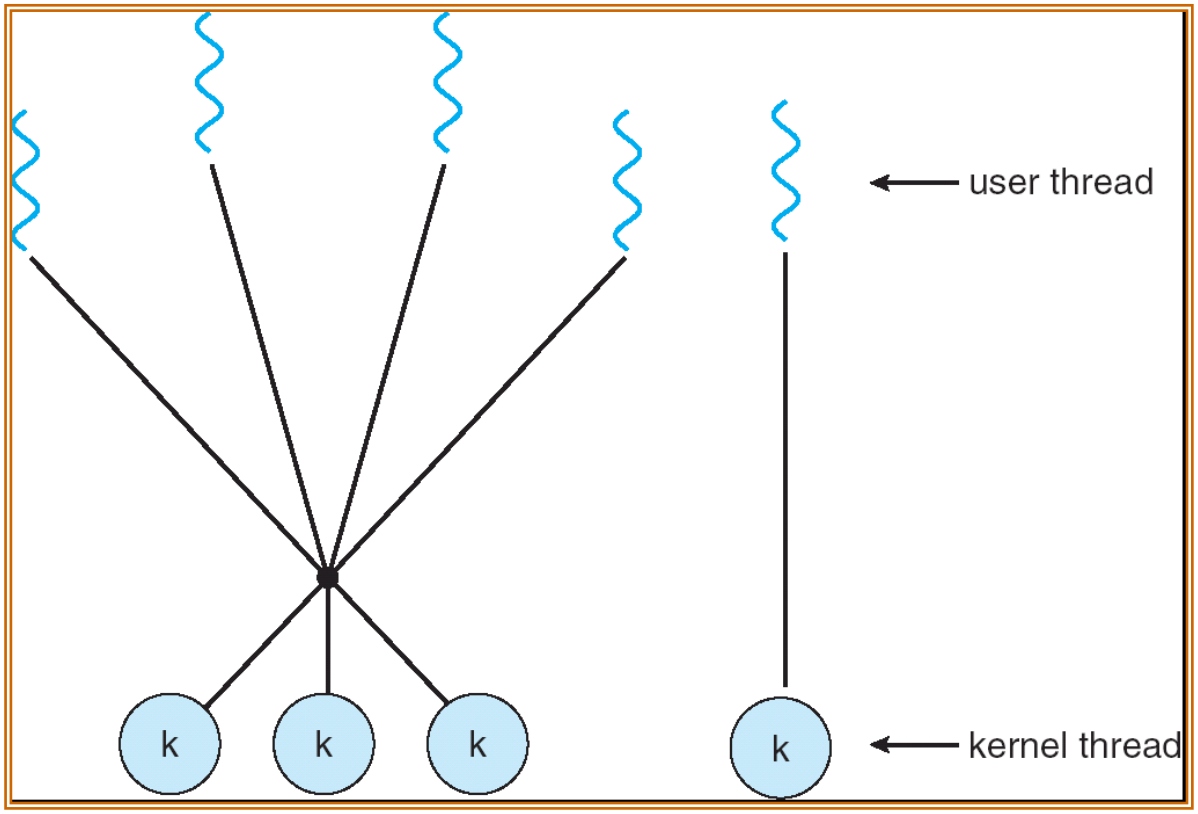
Multithreading Models: Comparison

- The **many-to-one** model allows the developer to create as many user threads as he/she wishes, but true concurrency can not be achieved because only one kernel thread can be scheduled for execution at a time
- The **one-to-one** model allows more concurrence, but the developer has to be careful not to create too many threads within an application
- The **many-to-many** model does not have these disadvantages and limitations: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor

Two-level Model

- Similar to **many-to-many**, 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



Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
  - Two versions of fork() in UNIX:, one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.
  - If a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process – including all threads.
  - If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate.
  - If the separate process does not call exec() after forking, the separate process should duplicate all threads.

Thread Cancellation

- Thread cancellation is the task of 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

Thread Pools

- The general idea is to create a number of threads at process startup and place them into a **pool** where they sit and wait for 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)

End of Chapter 4