Chapter 4: Threads

Chapter 4: Threads

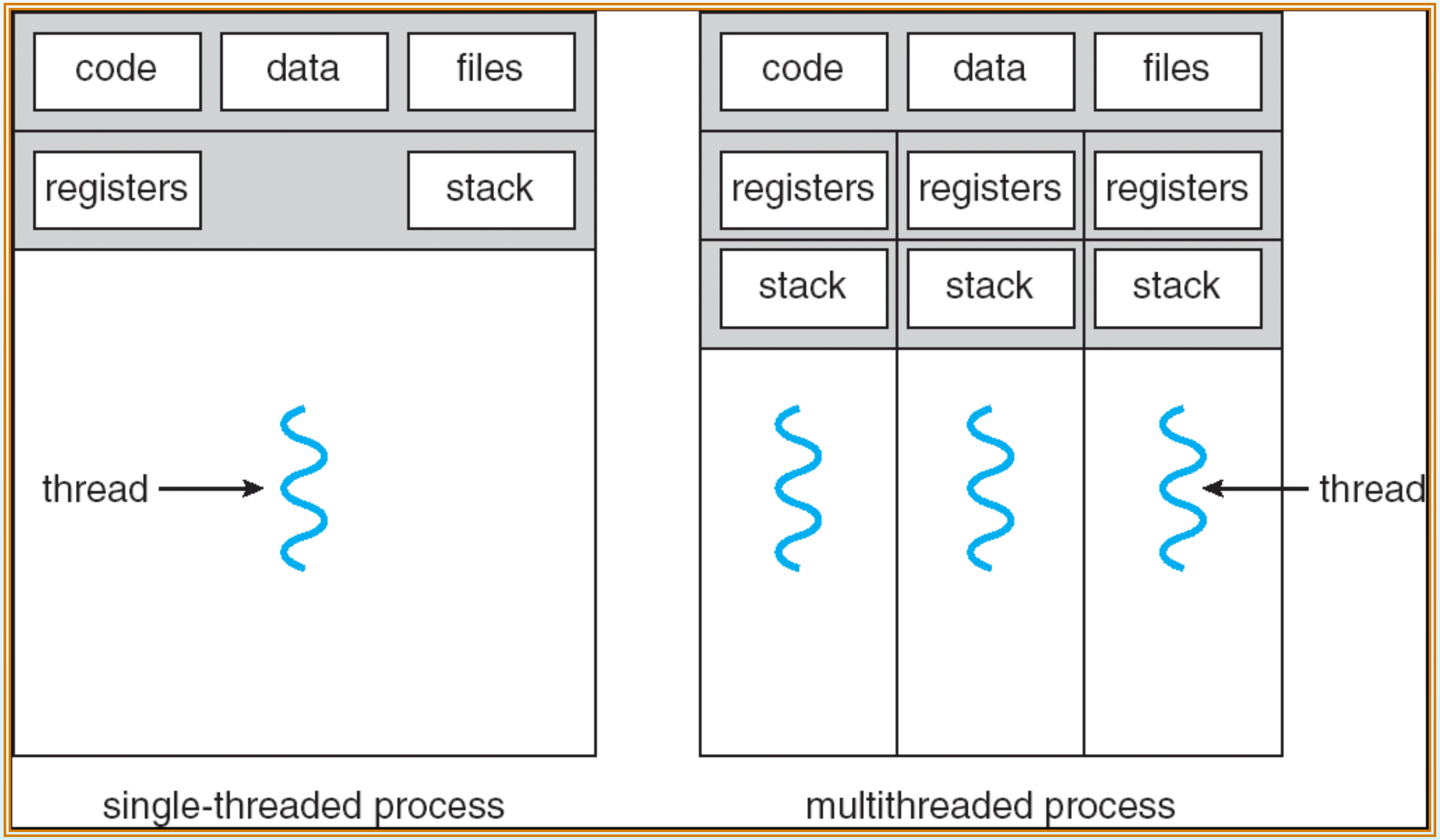
- Overview
- Multithreading Models
- Threading Issues
- Pthreads
- Windows XP Threads
- Linux Threads
- Java Threads

Single and Multithreaded Processes

What are threads?

- Sometimes called a **lightweight process** (LWP), a thread is the basic unit of CPU utilisation
- Each thread comprises:
  - Thread ID
  - Program counter
  - Register set
  - Stack
- It shares with other threads belonging to the same process, its code section, data section and other OS resources, such as open files and signals
- A traditional (or **heavyweight**) process has a single thread of control.
- A multi-threaded process has multiple threads of control, which means it can do more than one task at a time

Single and Multithreaded Processes



Benefits

- Responsiveness
  - Multithreading an interactive app. may allow a program to continue running even if part of it is blocked, or is performing a lengthy operation
- Resource Sharing
  - Threads share the memory & resources of the process to which they belong. Code sharing allows an app. to have several different threads of activitiy all within the same address space
- Economy
  - Because threads share resources, it's more economical to create and context switch threads; For eg: in Solaris 2
    - process creation time = 30 x thread creation time
    - Process context switch time = 5 x thread context switch time
- Utilization of Multi-Processor (MP) Architectures
  - Individual threads of one process can run in parallel on separate processors => increased concurrency and faster execution

User Threads

- Thread management done by user-level threads library which provides support for thread creation, scheduling and management, with no support from the kernel
  - User threads are supported *above* the kernel
- Advantage:
  - Since there's no kernel intervention, user-level threads are generally fast to create and manage
- Disadvantage:
  - If kernel is single threaded, and a user thread performs a blocking system call, then all other user threads will also be blocked => entire process will be blocked
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Supported directly by the OS: i.e. supported by the Kernel
  - The kernel performs thread creation, scheduling and management in **kernel space**
- **Disadvantage:**
  - Since thread management is carried out by the OS, kernel threads are generally slower to create and manage than user threads
- **Advantage**
  - If a thread performs a blocking system call, the kernel can schedule another thread in the application for execution
  - In multi-processor systems, the kernel can schedule threads on different processors
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

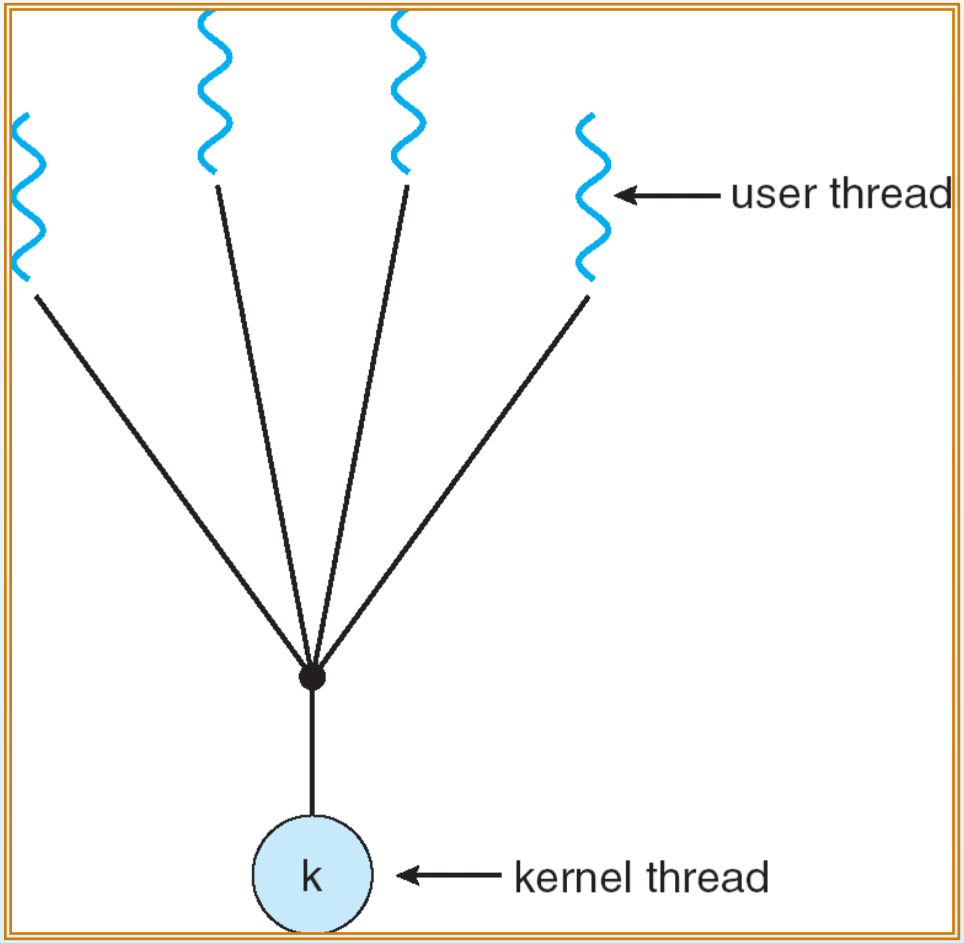
Many systems provide support for both user and kernel threads, resutling in different multithreading models. Three common types of thread implementation are:

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

Many-to-One

- Many user-level threads mapped to single kernel thread
- Used on systems that do not support kernel threads.
- Disadvantages:
  - The entire process will be blocked if one user-level thread belonging to it makes a blocking system call
  - Since only one thread can access the kernel at a time, multiple threads (from the same process) cannot run in parallel on multiprocessors
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

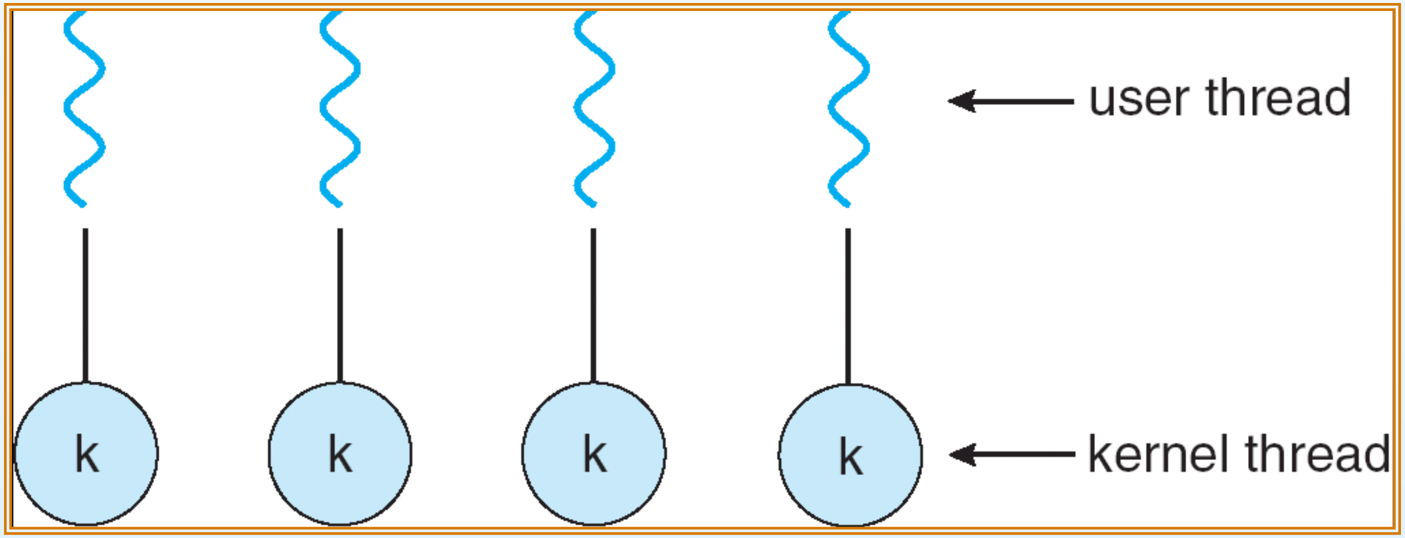
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- Advantages:
  - Provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call
  - Allows multiple threads to run in paralllel on multiprocessors
- Disadvantages
  - Creating a user thread requires creating the corresponding kernel thread. Because of the overhead involved in creating kernel threads, most implementations restrict the number of threads supported by the system
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

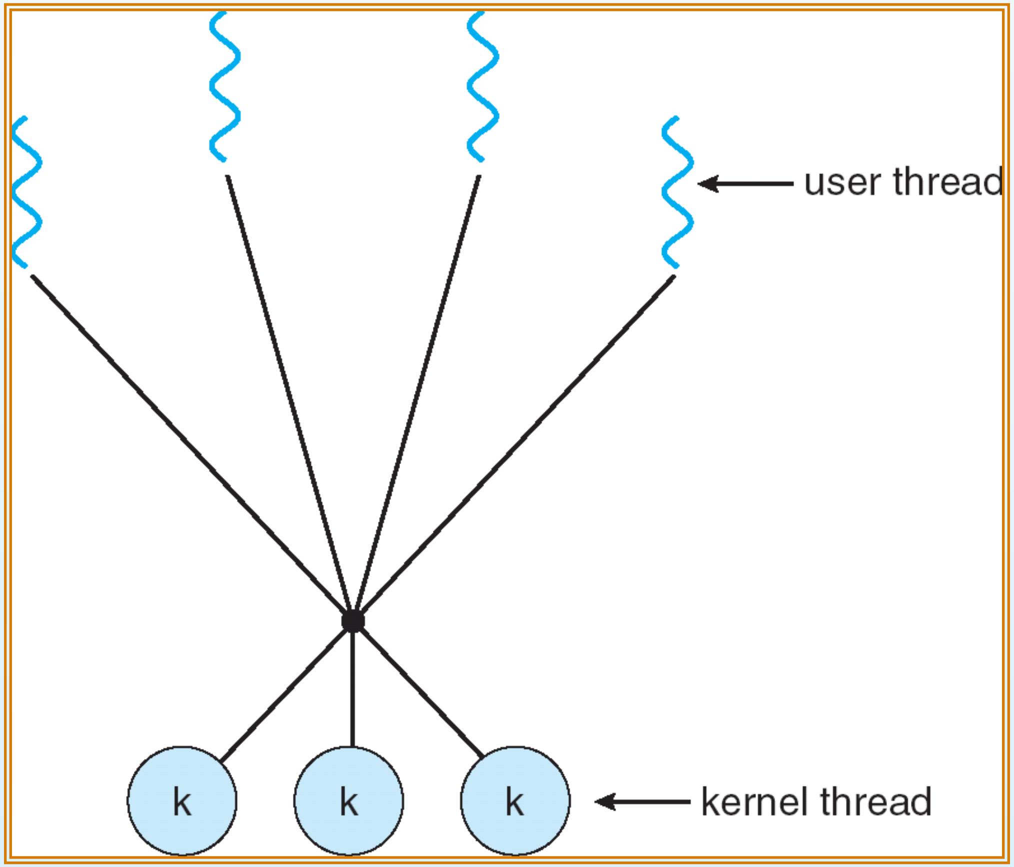
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to as many or a smaller number of kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- Advantage:
  - Doesn't suffer lack of concurrency seen in Many-to-One model
  - Doesn't suffer the overhead burden of the One-to-One model, because kernel threads can run in parallel on a multiprocessor
- Allows the operating system to create a sufficient number of kernel threads
- Examples:
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package

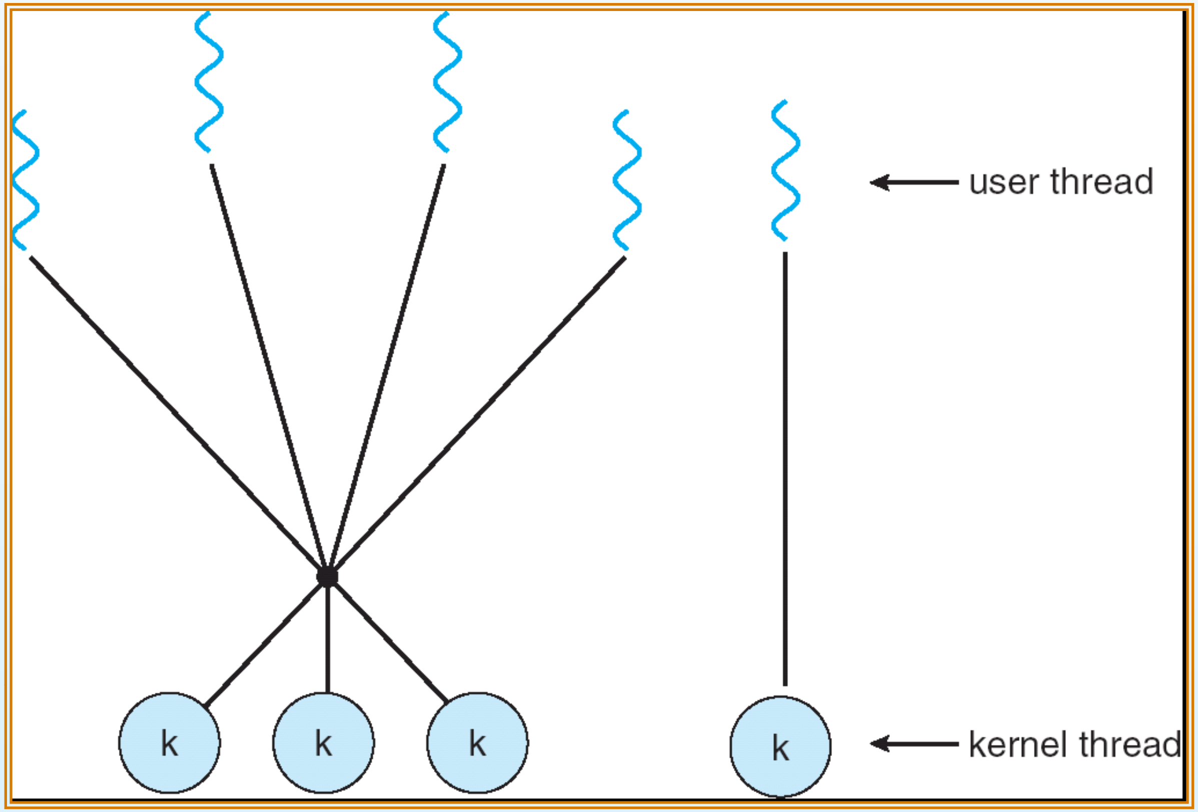
Many-to-Many Model



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



Threading Issues

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

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?

<u>Question</u>: If one thread in a process calls `fork`, does the new process duplicate all threads or is the new process single-threaded?

- Some UNIX systems have two versions of fork:
  - One version duplicates all threads
  - Other version duplicates only the thread that invoked `fork`

- The `exec` system call, typically works the same way as in single-threaded processes:
  - If a thread invokes the `exec` system call, the program specified in the parameter to `exec` will replace the *entire* process---including all threads and LWPs

Thread Cancellation

- Thread cancellation is the task of terminating a thread before it has completed
- Example:
  - Multiple threads concurrently searching through a database for one result
- Thread to be cancelled is called a **target thread**
- Cancellation may occur in two different scenarios:
  - **Asynchronous cancellation**: One thread immediately terminates the target thread
  - **Deferred cancellation**: The target thread can periodically check if it should terminate, allowing the target thread the opportunity to terminate itself in an orderly fashion
- Difficulties with cancellation may arise in situations where resources have been allocated to a cancelled thread or if a thread is cancelled during the updating of data it shares with other threads

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
- Signals may be received in either of two ways (depending on the source and the reason for the event being signalled):
  - Synchronously => signal delivered to the *same* process that performed the operation causing the signal; for eg: illegal memory access by a process generates a signal to that process
  - Asynchronously => signal delivered is generated by an event *external* to a running process; for eg: CTRL-C by the user!

Signal Handling

- Every signal may be handled by one of two possible handlers:
  - A default signal handler
  - A user-defined signal handler
- **Default signal handler**: run by the kernel when handling the signal
- **User-defined signal handler**: user-defined function which over-rides default signal handler

Thread Pools

- Consider the example of multithreading a web server. A simple implementation might be as follows:
  - When the server receives a request, it simply creates a separate thread to service the request
- There are some concerns associated with this approach
  - It takes time to create a thread, prior to servicing the request
  - This thread will probably be discarded, once it completes its work.
  - With no bound on the number of threads, system resources could be exhausted by unlimited threads being created to service requests
- <u>Alternative approach</u>: Create a number of threads at process startup and place them into a ***thread pool***, where they sit and wait for work.
  - When a server receives a request, it awakens a thread from the pool, if one is available
  - Once the thread completes its service, it returns to the pool for use by another request!
- 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

- Recall: Threads belonging to a process share the data of the process
- 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)
- For example: In a transaction-processing system:
  - we might service each transaction in a separate thread
  - Each transaction may be assigned a unique ID
  - To associate each thread with it unique ID, we could use thread-specific data
- Most thread libraries, and some languages, provide some form of support for thread-specific data:
  - Win32 thread library
  - Pthread library
  - Java programming language

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

Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, but implementation is up to developers of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- (Mac OS X uses FreeBSD UNIX)

Windows XP Threads

- Implements the one-to-one mapping
- 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)
  - TEB (thread environment block)

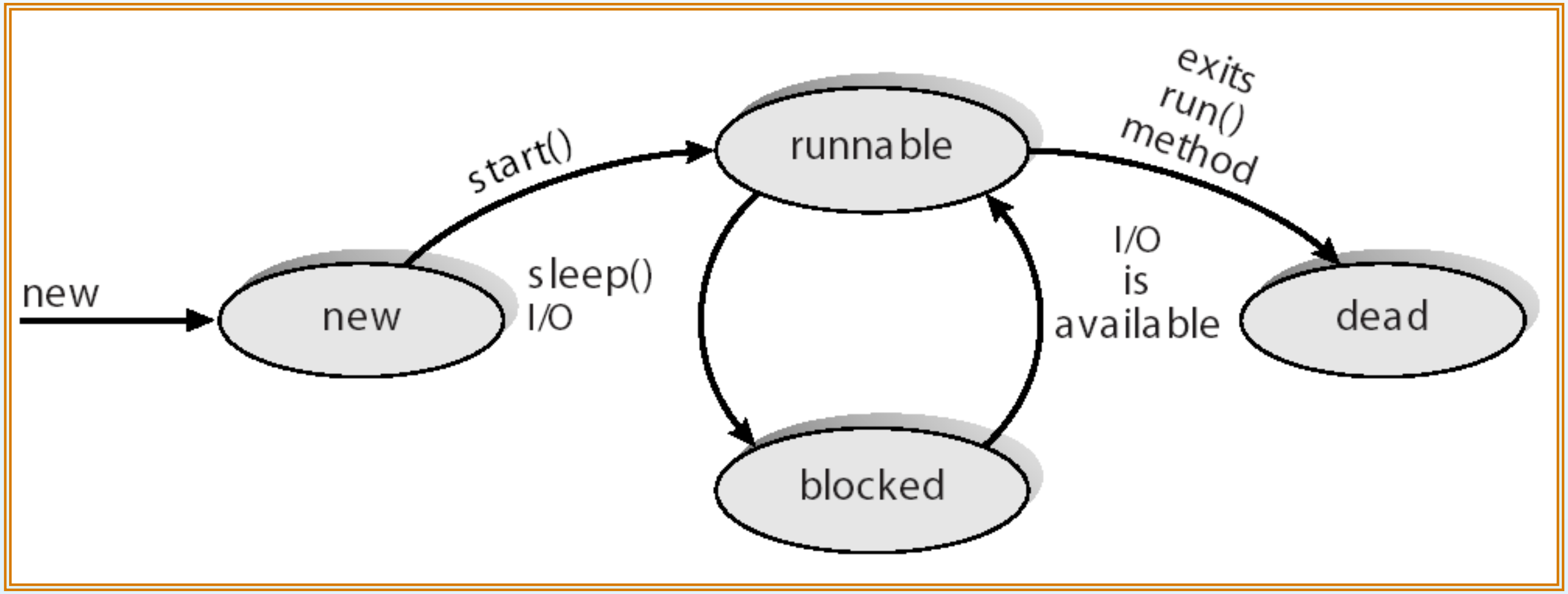
Linux Threads

- 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)

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Java Thread States



End of Chapter 4