Threads

Thread

- Thread: single sequential flow of control within a program
- Single-threaded program can handle one task at any time.
- Multitasking allows single processor to run several concurrent threads.
- Most modern operating systems support multitasking.

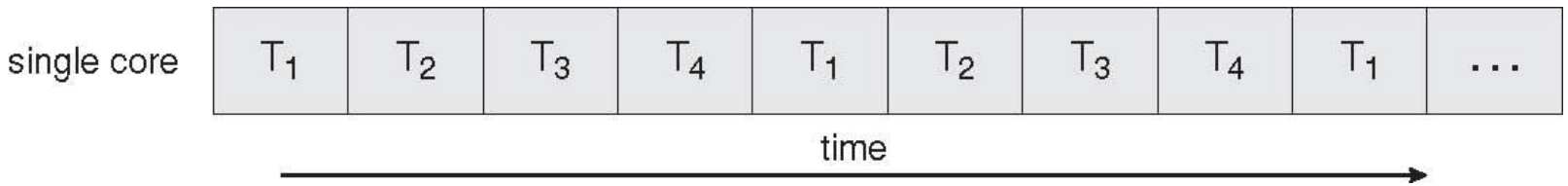
Thread Concepts

- Traditionally a process has a single address space and a single thread of control to execute a program within that address space.
- To execute a program, a process has to initialize and maintain state information.
- The state information is comprised of page tables, swap image, file descriptors, outstanding I/O requests, saved register values etc. This information is maintained on a per program basis and thus a per process basis.
- The volume of this information makes it expensive to create and maintain processes as well as to switch between them.
- Threads or light weight processes have been proposed to handle situations where creating, maintaining and switching between processes occur frequently.

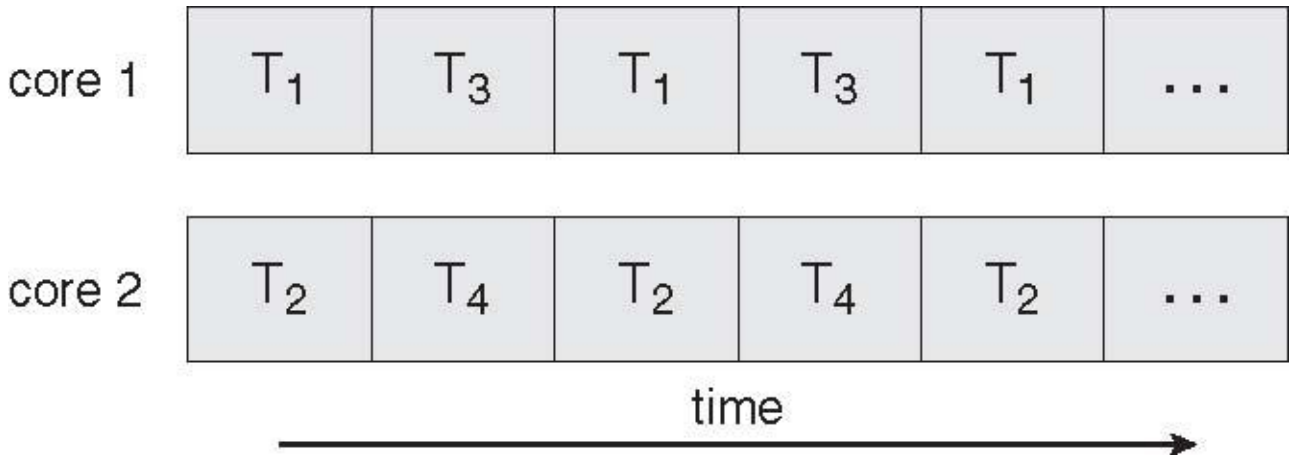
Multicore Programming

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

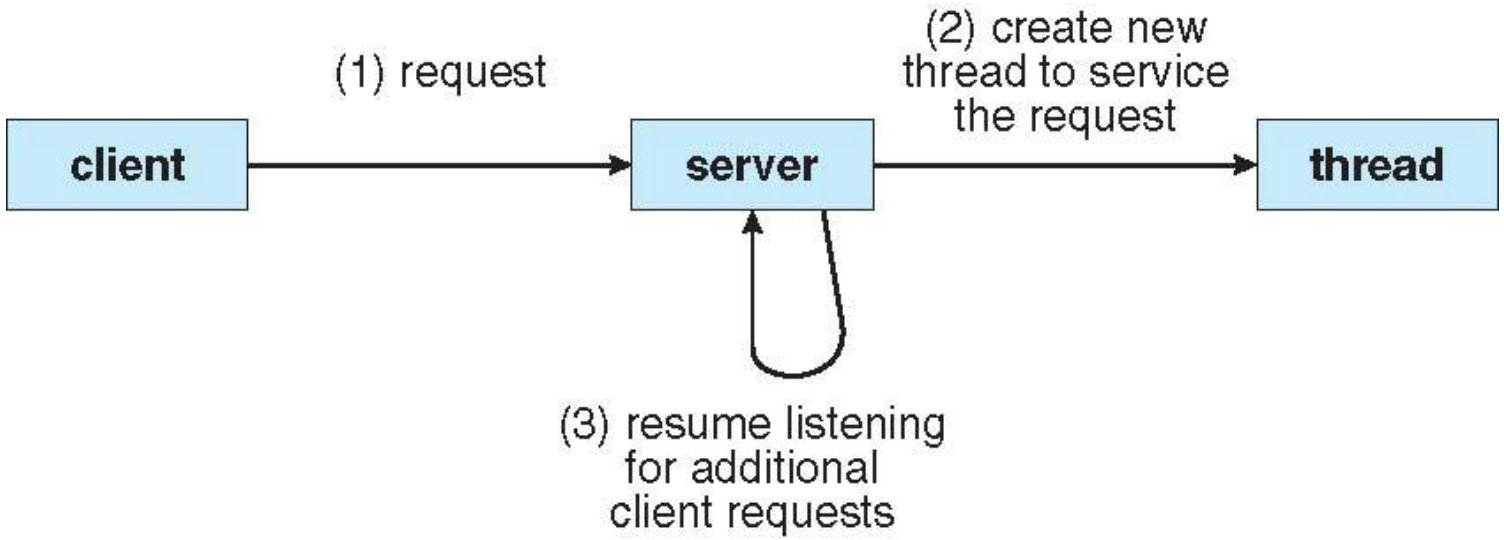
Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System

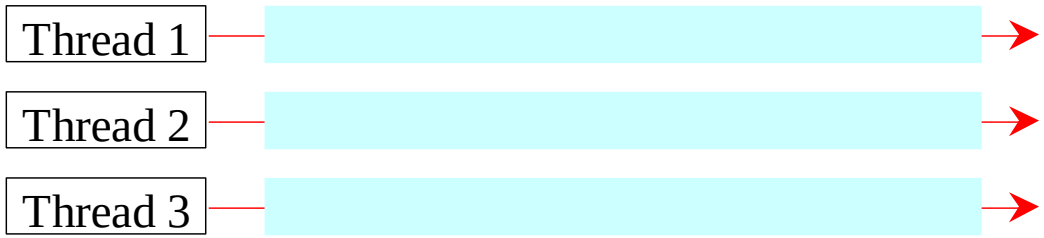


Multithreaded Server Architecture

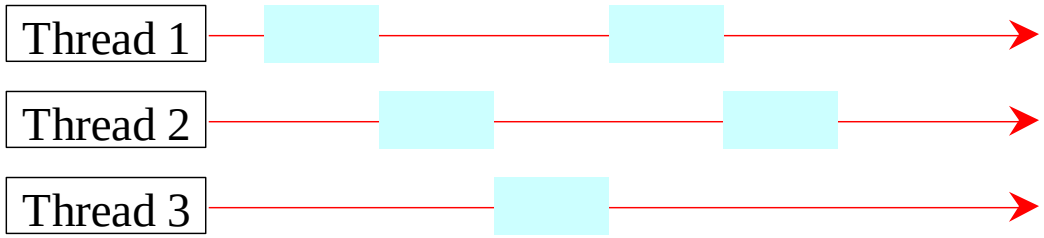


Threads Concept

Multiple threads on multiple CPUs



Multiple threads sharing a single CPU



- A process is an abstraction for representing resource allocation.
- A *thread* is an abstraction for execution: a thread represents the execution of a particular sequence of instructions in a program's code, or equivalently a particular path through the program's flow of control.
- A process may have multiple threads, each sharing the resources allocated to the process.

Threads vs Processes

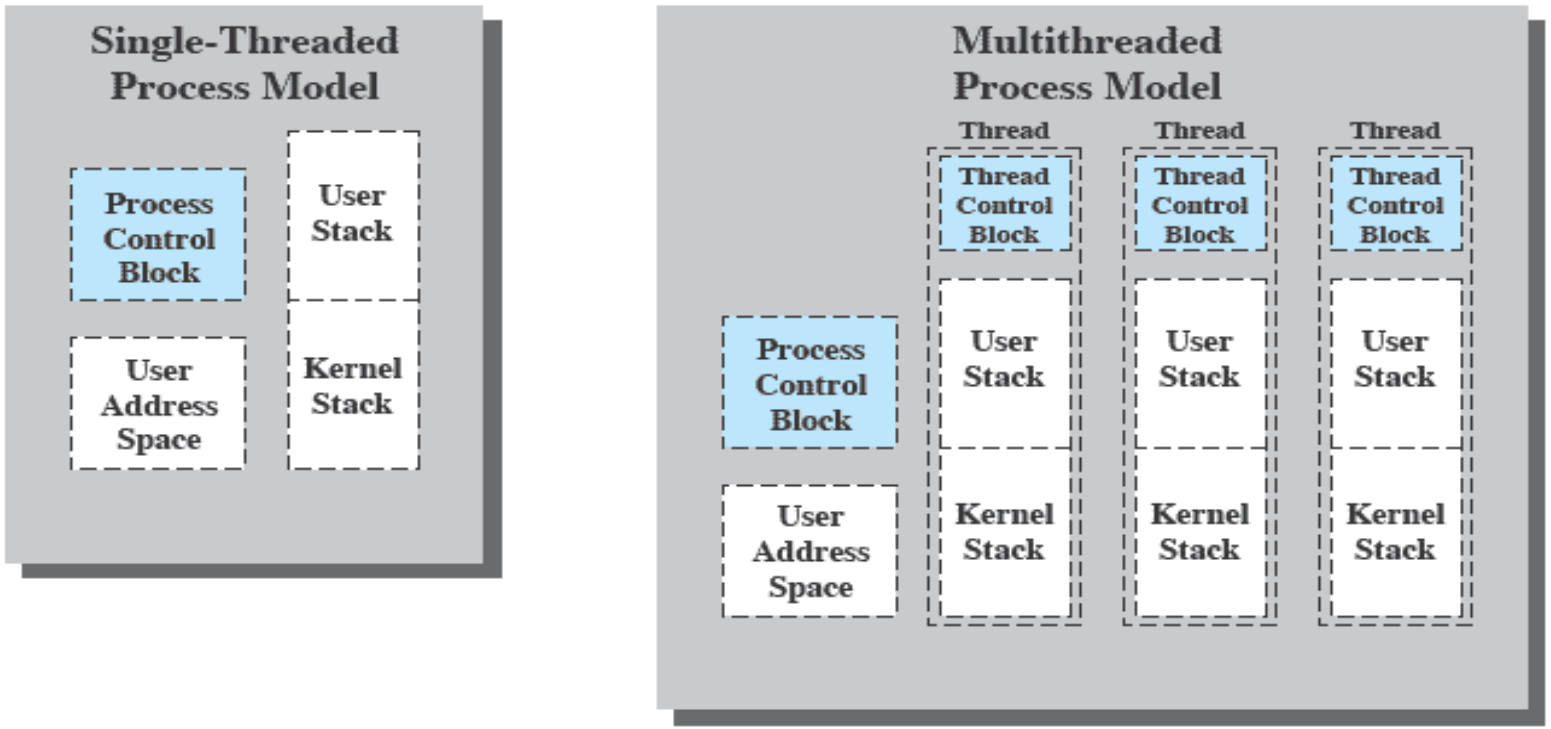
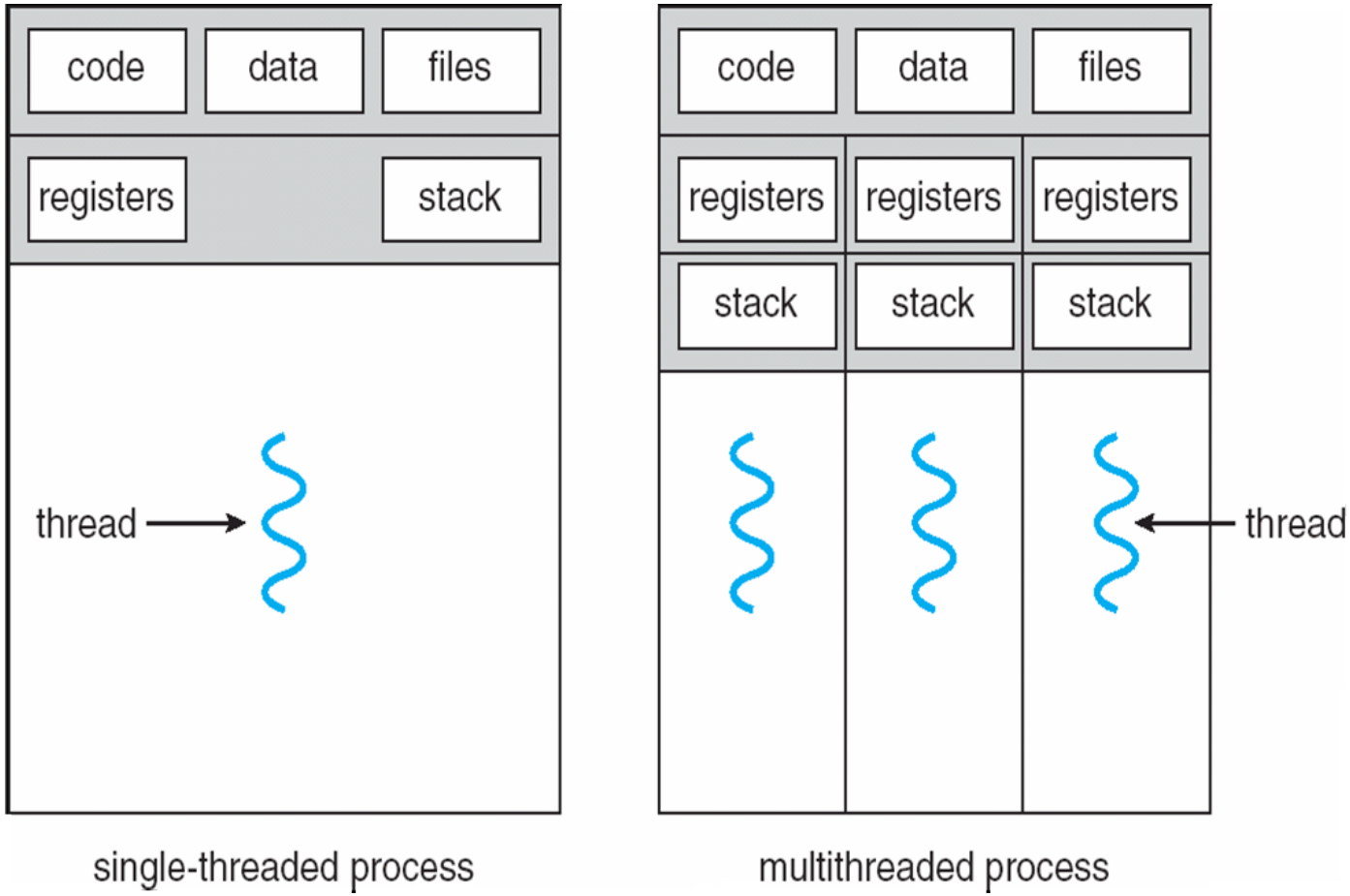


Figure 4.2 Single Threaded and Multithreaded Process Models

Single and Multithreaded Processes



- A single thread executes a portion of a program, cooperating with other threads concurrently executing within the same address space. Each thread makes use of a separate program counter and a stack of activation records and a thread control block.
- The control block contains the state information necessary for thread management, such as putting a thread into a ready list and for synchronizing with other threads.
- Threads share all resources (memory, open files, etc.) of their parent process *except* the CPU.

- The purpose of the thread abstraction is to simplify programming of logically parallel, cooperating activities; threads enable each activity to be implemented largely as a sequential program that shares resources with its peer threads.
- But, since all threads in the same process share the process's resources, communication (and hence cooperation) among threads is easier to achieve than if each thread was a separate process.
- Most of the information that is part of a process is common to all the threads executing within a single address space and hence maintenance is common to all threads.
- By sharing common information overhead incurred in creating and maintaining information and the amount of information that needs to be saved when switching between threads of the same program is reduced significantly.

Each thread requires its own context:

- program counter
- stack
- Registers

Each Thread has

- an execution state (Running, Ready, etc.)
- saved thread context when not running (TCB)
- an execution stack
- some per-thread static storage for local variables
- access to the shared memory and resources of its process (all threads of a process share this)

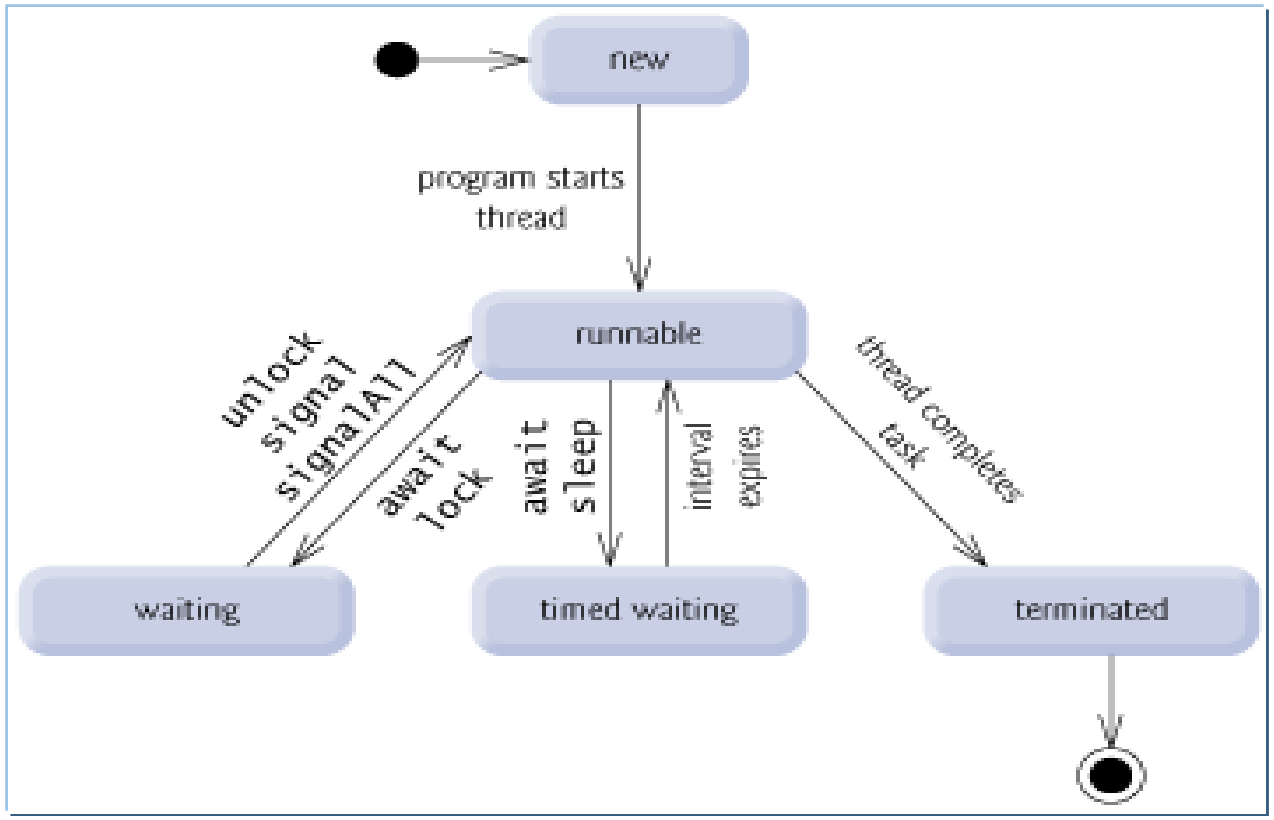
Thread Execution States

Similar to a process a thread can be in any of the primary states: Running, Ready and Blocked.

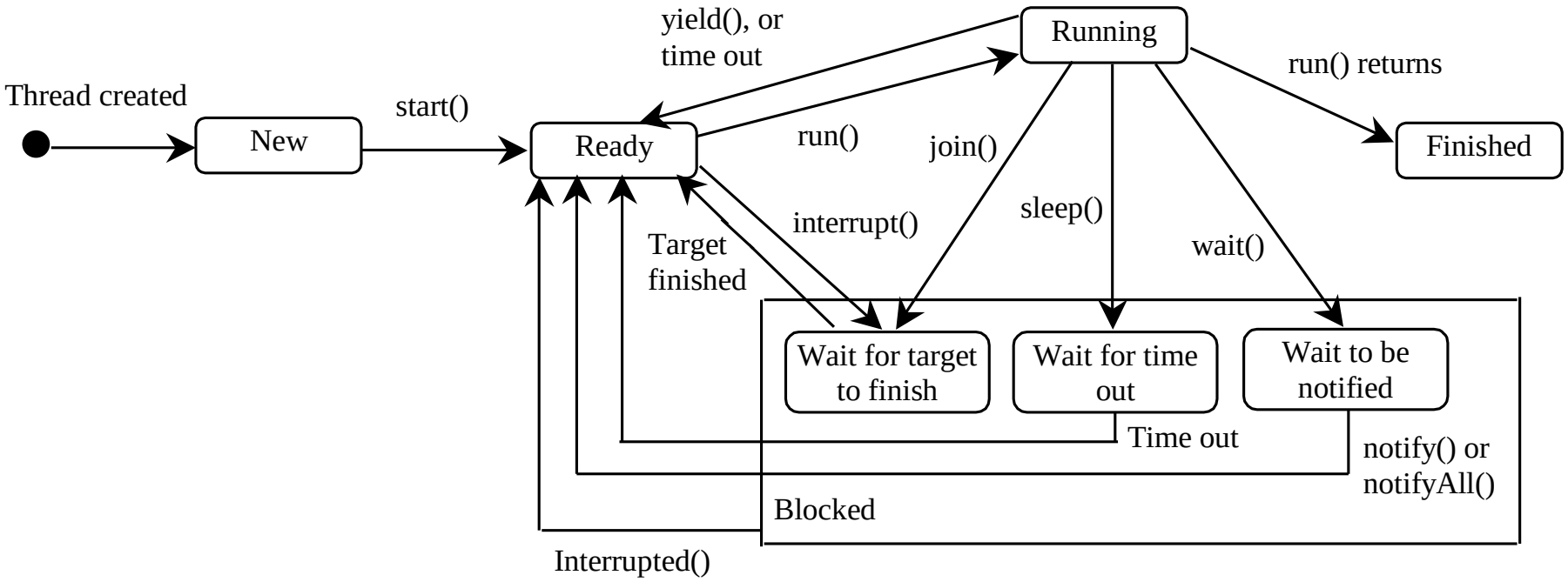
The operations needed to change state are:

- Spawn: new thread provided register context and stack pointer.
- Block: event wait, save user registers, PC and stack pointer
- Unblock: moved to ready state
- Finish: deallocate register context and stacks.

Thread States



Thread States



Thread Not Runnable

A thread becomes Not Runnable when one of these events occurs:

- Its sleep method is invoked.
- The thread calls the wait method to wait for a specific condition to be satisfied.
- The thread is blocking on I/O.

Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
  - suspending a process involves suspending all threads of the process
  - termination of a process terminates all threads within the process

Thread Scheduling

- An operating system's thread scheduler determines which thread runs next.
- Most operating systems use *timeslicing* for threads of equal priority.
- *Preemptive scheduling*: when a thread of higher priority enters the running state, it preempts the current thread.
- *Starvation*: Higher-priority threads can postpone (possible forever) the execution of lower-priority threads.

Thread Synchronization

- It is necessary to synchronize the activities of the various threads
  - all threads of a process share the same address space and other resources
  - any alteration of a resource by one thread affects the other threads in the same process

Common Thread Models

- User level threads

  These threads implemented as user libraries. Thread library provides programmer with API for creating and managing threads. Benefits of this are no kernel modifications, flexible and low cost. The drawbacks are thread may block entire process and no parallelism.

- Kernel level threads

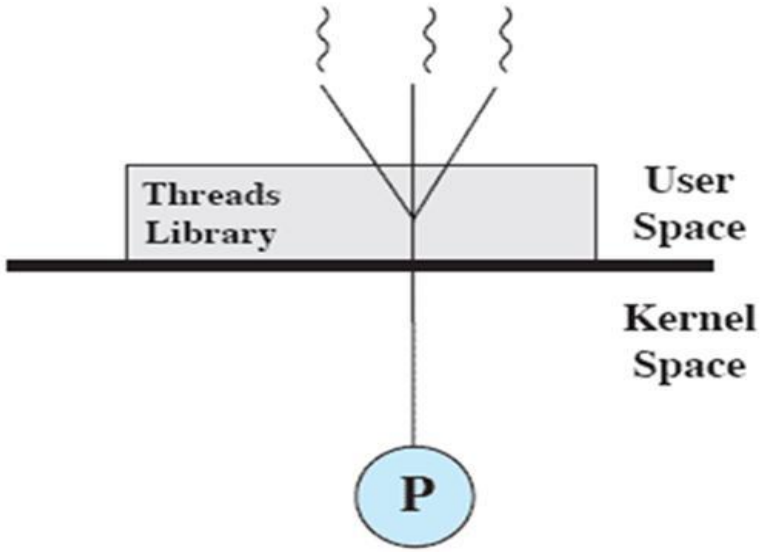
  Kernel directly supports multiple threads of control in a process. Benefits are scheduling/synchronization coordination, less overhead than process, suitable for parallel application. The drawbacks are more expensive than user-level threads and more overhead.

- Light-Weight Processes (LWP)

  It is a kernel supported user thread. LWP bound to kernel thread but a kernel thread may not be bound to an LWP. It is scheduled by kernel and user threads scheduled by library onto LWPs, so multiple LWPs per process.

User-Level Threads (ULTs)

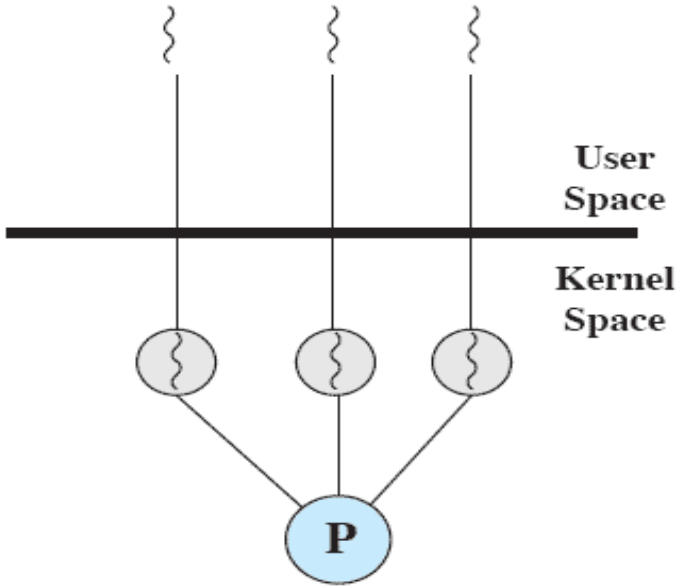
- Thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level

Kernel-Level Threads (KLTs)

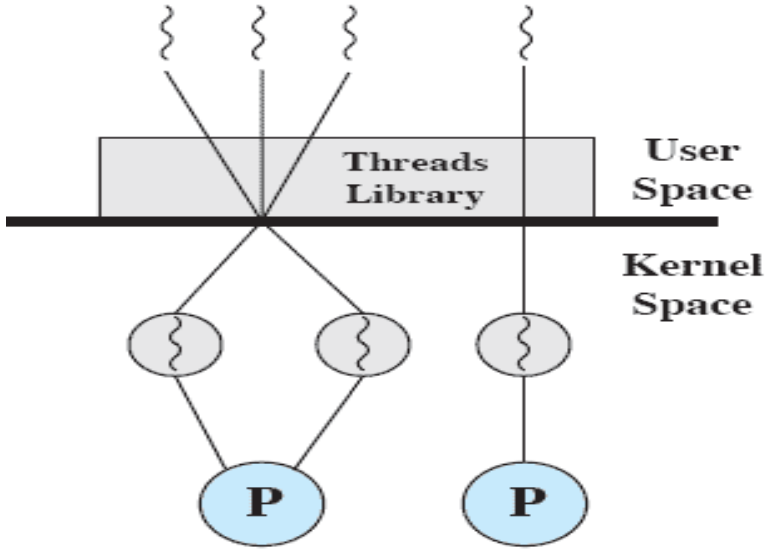
- Thread management is done by the kernel (could call them K*M*T)
- No thread management is done by the application; Windows is an example of this approach



(b) Pure kernel-level

Combined Approaches

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads is by the application
- Solaris is an example



(c) Combined

Multithreading

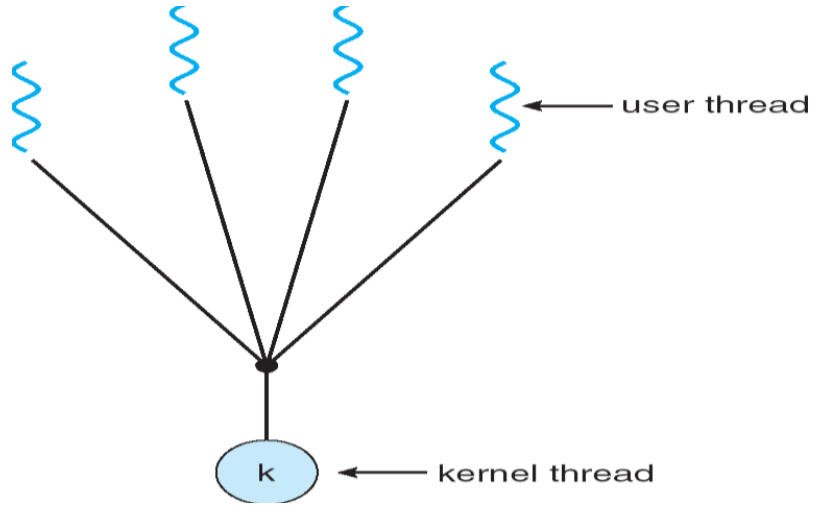
Kernels are generally multi-threaded.

Multi-threading models include

- Many-to-One: Many user-level threads mapped to single kernel thread
- One-to-One: Each user-level thread maps to kernel thread
- Many-to-Many: Many user-level threads mapped to many kernel threads.

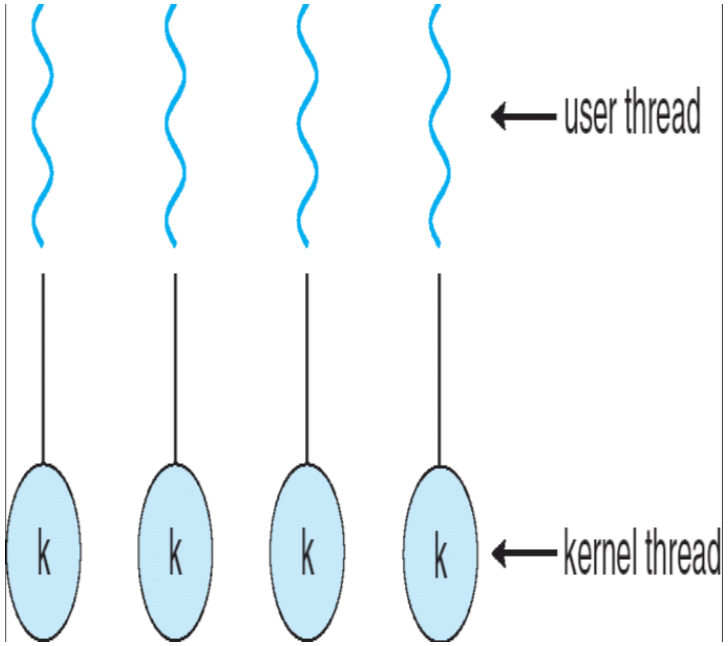
Many-to-One

- Thread management is done by the thread library in user space
- The entire process will block if a thread makes a blocking system call
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Mainly used in language systems, portable libraries
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



- Advantages:
  - totally portable
  - easy to do with few systems dependencies
- Disadvantages:
  - cannot take advantage of parallelism
  - may have to block for synchronous I/O

One-to-One

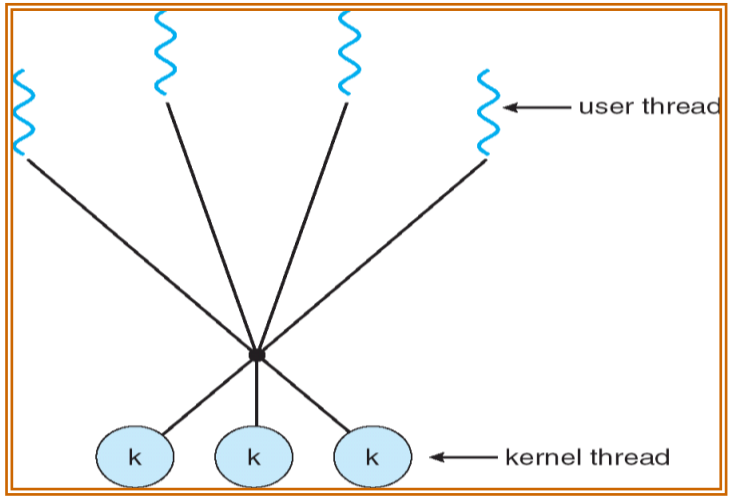


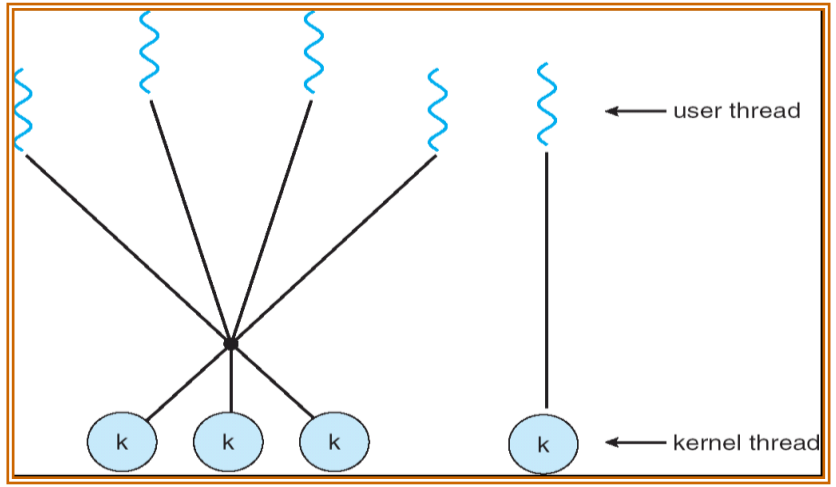
- Each **user-level thread** maps to **kernel thread**
  - Creating a **user-level** thread **creates** a **kernel thread**
  - *This is a drawback to this model; creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may **burden** the performance of a system.*
- More concurrency than many-to-one
  - Allows another thread to run when a thread makes a blocking system call
  - It also allows multiple threads to run in parallel on multiprocessors.
- Number of threads per process sometimes restricted due to overhead
- Used in LinuxThreads and other systems where LWP creation is not too expensive

- Advantages:
  - can exploit parallelism, blocking system calls
- Disadvantages:
  - thread creation involves LWP creation
  - each thread takes up kernel resources
  - limiting the number of total threads

Many-to-many

- In this model, the library has two kinds of threads: *bound* and *unbound*
  - bound threads are mapped each to a single lightweight process
  - unbound threads *may* be mapped to the same LWP





- 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 a particular application or a particular machine
  - *An application may be allocated more kernel threads on a system with eight processing cores than a system with four cores*
- Used in the Solaris implementation of Pthreads (and several other Unix implementations)
- Windows with the *ThreadFiber* package
- Otherwise not very common

- Issues in multithreading include
  - Thread Creation
  - Thread Cancellation
  - Signal Handling (synchronous / asynchronous),
  - Handling thread-specific data and scheduler activations.

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 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
    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 threat to receive all signals for 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)

Scheduler activations

- Many : Many 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

Relationship Between Threads and Processes

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

User Threads

- Thread management done by user-level threads library

- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

Supported by the Kernel

- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux

Various Implementations

PThreads

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

Windows 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

Various Implementations

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