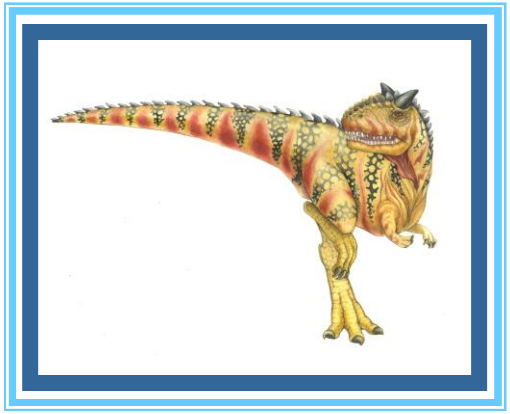
Chapter 4: Threads



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

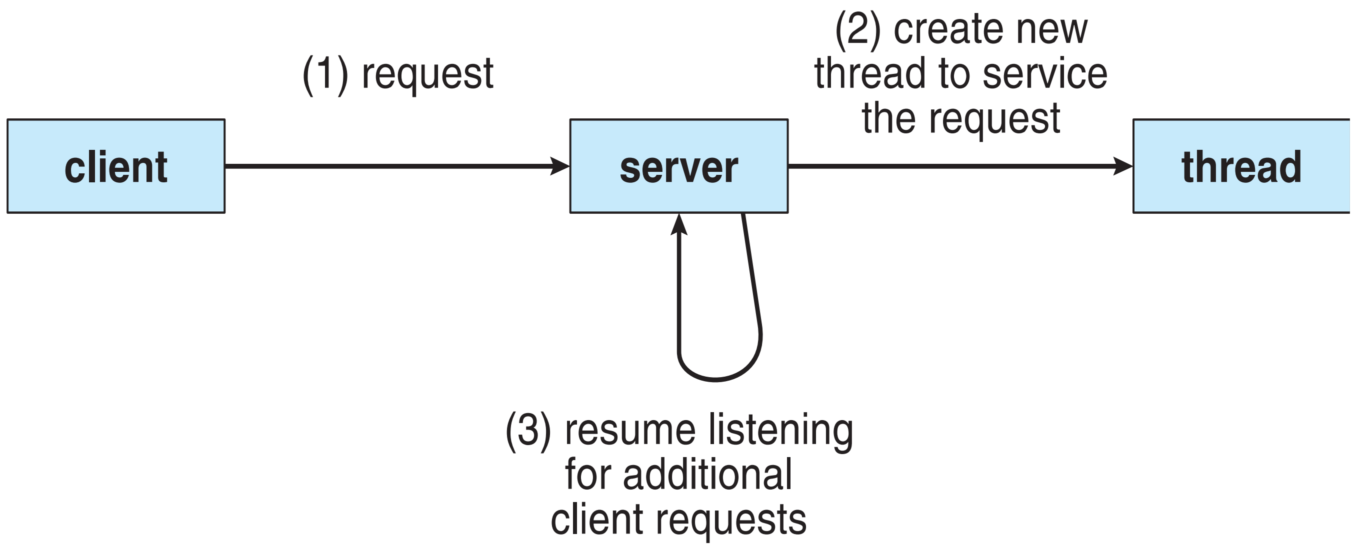
Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Multithreaded Server Architecture



Benefits

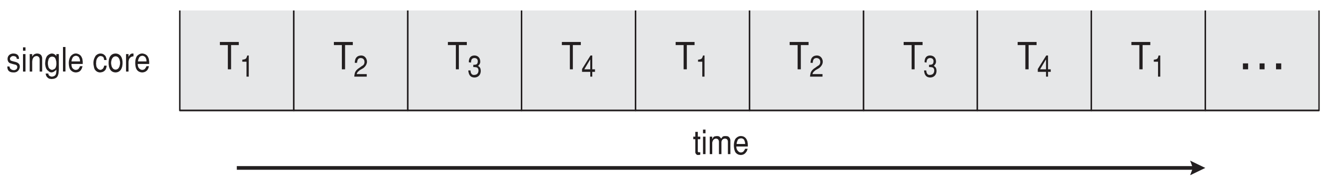
- **Responsiveness**
  - may allow continued execution if part of process is blocked
  - especially important for user interfaces
- **Resource Sharing**
  - threads share resources of process: easier than shared memory or message passing
- **Economy**
  - Thread creation is faster than process creation
    - Less new resources needed vs a new process
    - Solaris: 30x faster
  - Thread switching lower overhead than context switching
    - 5x faster
- **Scalability**
  - Threads can run in parallel on many cores

Multicore Programming

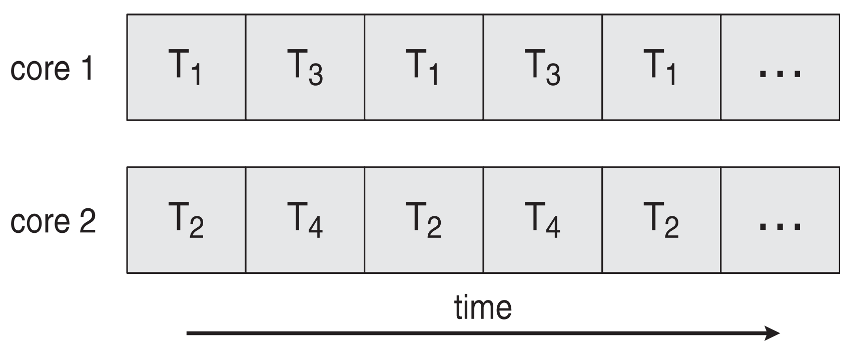
- **Multicore** or **multiprocessor** systems
  - Putting pressure on programmers
  - How to load all of them for efficiency
  - Challenges include:
    - **Dividing activities**
    - **Balance**
    - **Data splitting**
    - **Data dependency**
    - **Testing and debugging**

Concurrency vs. Parallelism

- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - True parallelism or an illusion of parallelism
  - Single processor / core, scheduler providing concurrency
- **Concurrent execution on single-core system**



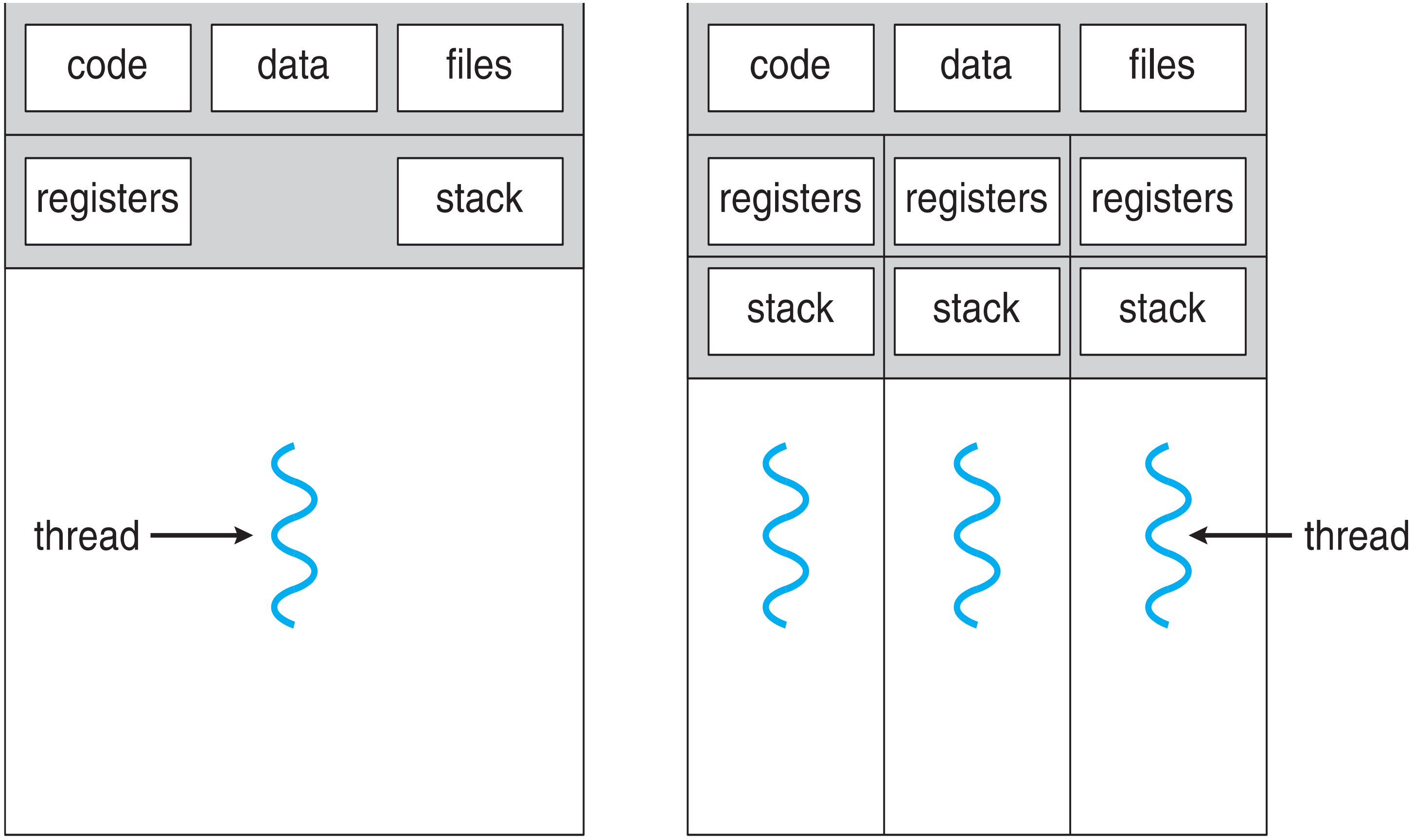
- **Parallelism on a multi-core system:**



Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation/task on each
    - Example: the task of incrementing elements by one of an array can be split into two: incrementing its elements in the 1^{st} and 2^{nd} halfs
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
  - In practice, people often follow a hybrid of the two
- Architectural support for threading grows
  - CPUs have cores as well as ***hardware threads***
    - N hardware threads per core
      - Means N threads can be loaded into the core for fast switching.
  - Consider Oracle SPARC T4:
    - 8 cores
    - 8 hardware threads per core

Single and Multithreaded Processes



single-threaded process

multithreaded process

User Threads and Kernel Threads

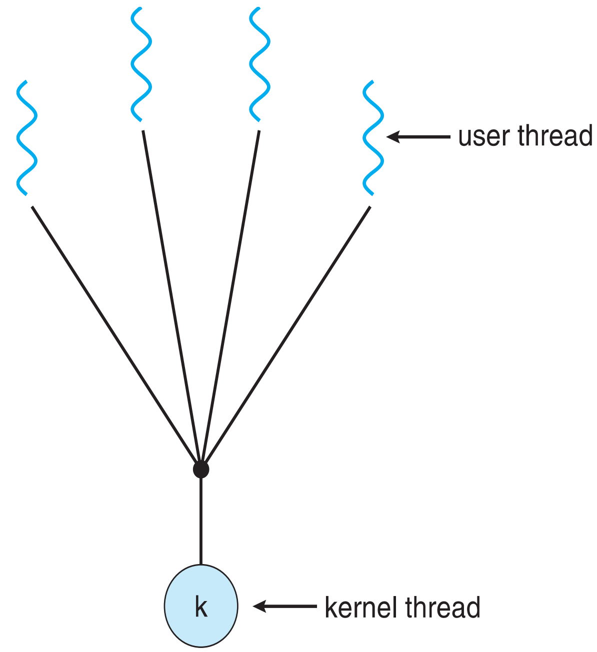
- **User threads**
  - Support provided at the user-level
  - Managed above the kernel
    - without kernel support
  - Management is done by thread library
    - Three primary thread libraries:
    - POSIX **Pthreads,** Windows threads, Java threads
- **Kernel threads**
  - Supported and managed by OS
  - Virtually all modern general-purpose operating systems support them

Multithreading Models

- A relationship exists between user threads and kernel threads
- Three common ways of establishing this relationships
  - Many-to-One model
  - One-to-One model
  - Many-to-Many model

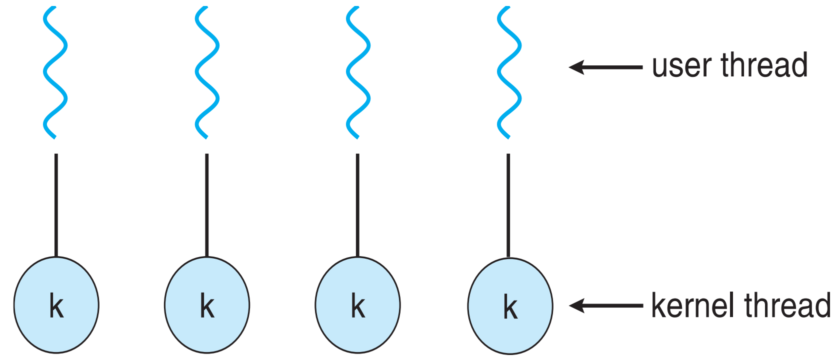
Many-to-One

- Many user-level threads mapped to single kernel thread
- **Advantage:**
  - Thread management in user space
  - Hence, efficient
- **Disadvantages:**
  - One thread blocking causes all to block
  - Multiple threads may not run in parallel on multicore system
    - Since only 1 may be in kernel at a time
    - So, few systems currently use this model



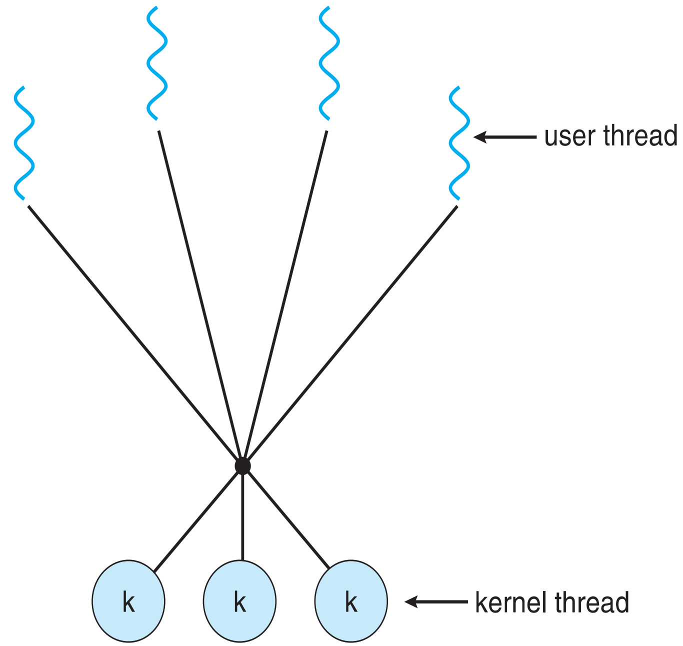
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- **Advantages:**
  - More concurrency than many-to-one
- **Disadvantages:**
  - High overhead of creating kernel threads
  - Hence, number of threads per process sometimes restricted
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



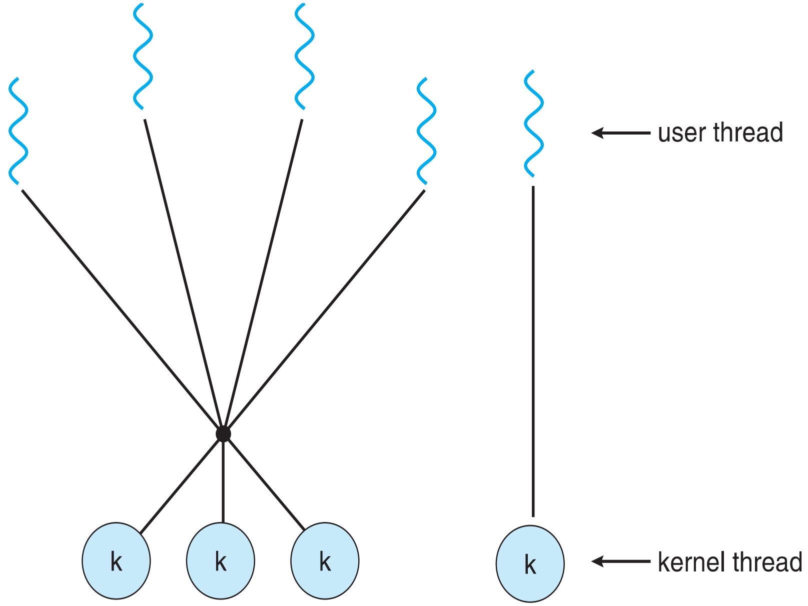
Many-to-Many Model

- Allows many user-level threads to be mapped to (smaller or equal number of) kernel threads
- Allows the OS to create a sufficient number of kernel threads
  - The number is dependent on specific machine or application
  - It can be adjusted dynamically
- Many-to-one
  - Any number of threads is allowed, but low concurrency
- One-to-one
  - Great concurrency, but the number of threads is limited
- Many-to-many
  - Gets rid of the shortcomings of the precious two



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



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
- Examples of thread libraries
  - Pthreads, Java Threads, Widnows threads

Threading Issues

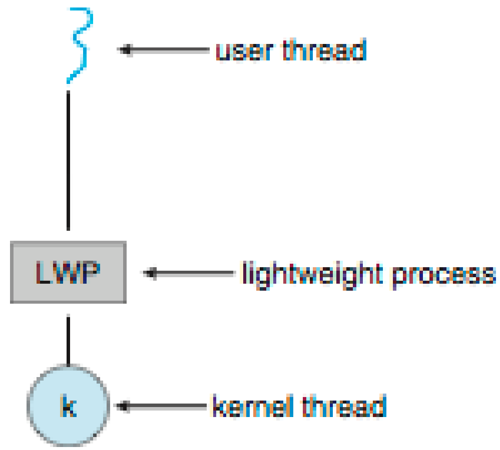
- Semantics of **fork()** and **exec()** system calls
- Many other issues (we will not consider their details)
  - Signal handling
    - Synchronous and asynchronous
  - Thread cancellation of target thread
    - Asynchronous or deferred
  - Thread-local storage
  - Scheduler Activations

Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal – replace the running process including all threads

Lightweight process

- **Lightweight Process** (**LWP**)
  - An intermediate data structure between user and kernel threads
  - To user-level thread library, it appears as a virtual processor on which process can schedule user thread to run
  - Each LWP attached to a kernel thread
  - LWP are used, for example, to implement Many-to-many and two-level models



Operating System Examples

- Windows Threads
- Linux Threads

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)
  - Flags control behavior

flag	meaning
`CLONE_FS`	File-system information is shared.
`CLONE_VM`	The same memory space is shared.
`CLONE_SIGHAND`	Signal handlers are shared.
`CLONE_FILES`	The set of open files is shared.

- **`struct task_struct`** points to process data structures (shared or unique)

End of Chapter 4