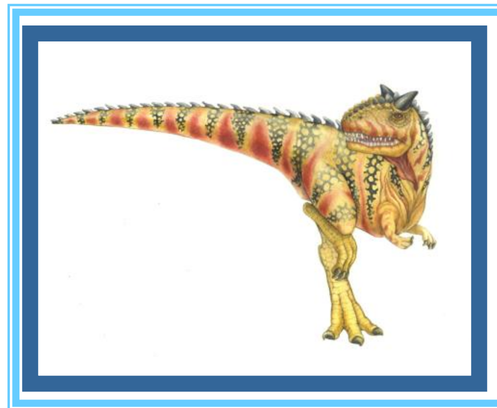
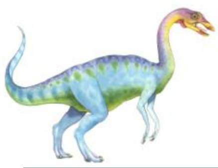
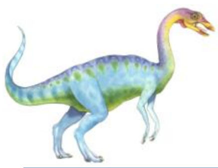
Chapter 4: Multithreaded Programming





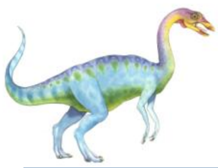
Chapter 4: Multithreaded Programming

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues

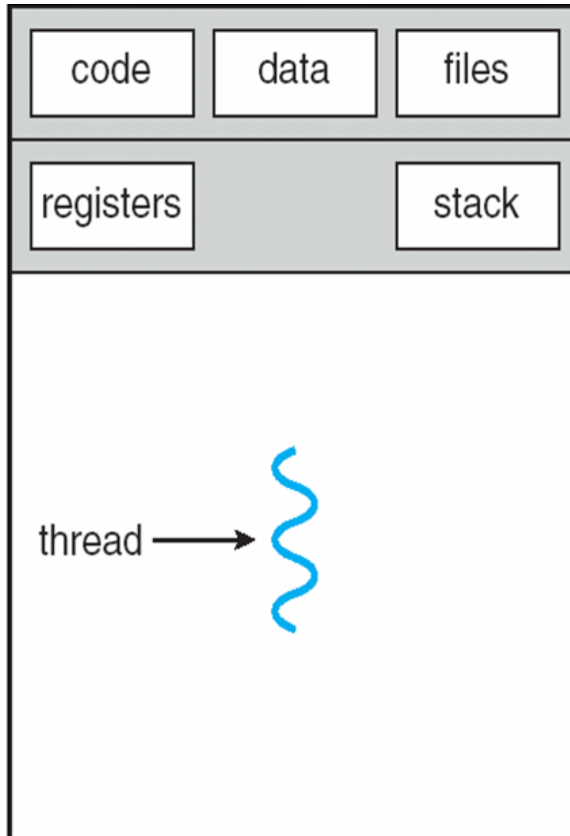


Objectives

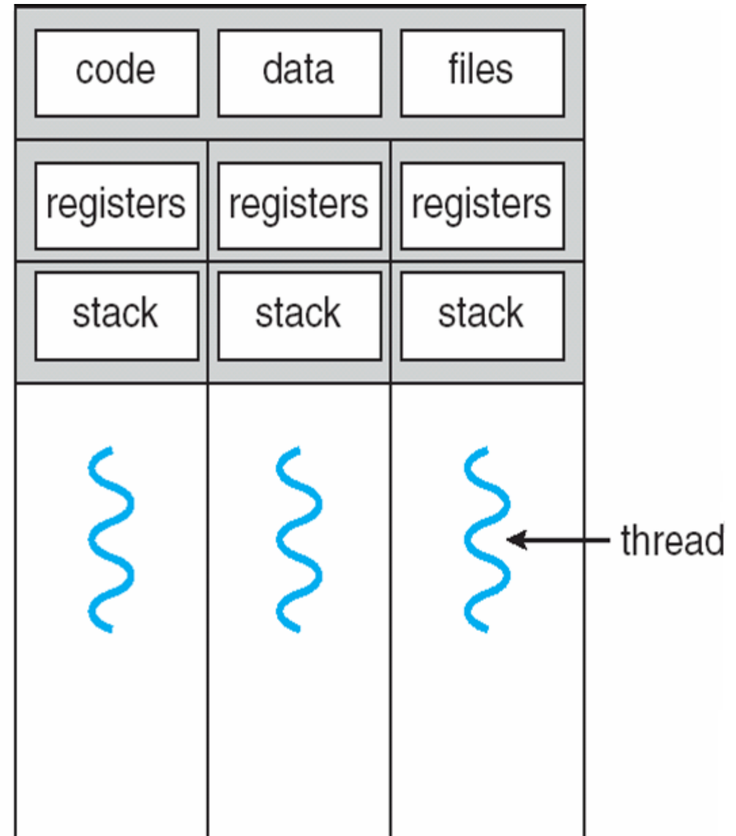
- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming



Single and Multithreaded Processes



single-threaded process

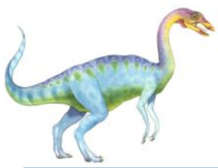


multithreaded process

Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability

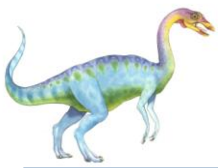
See Text Book



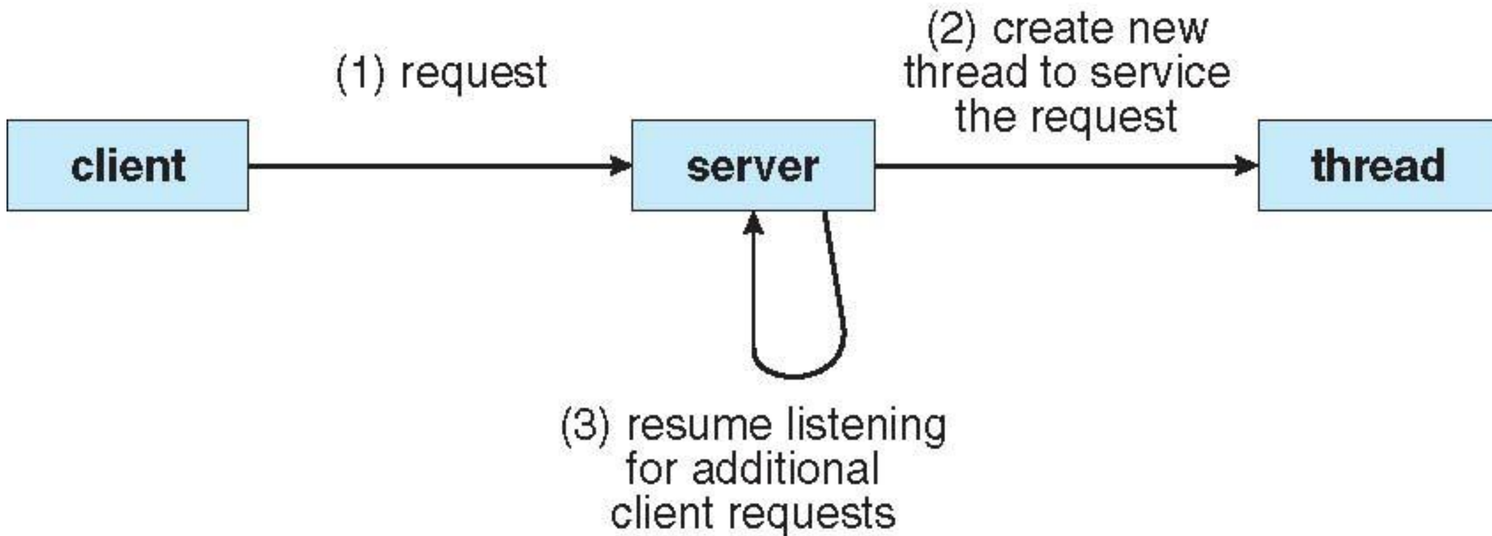
Multicore Programming

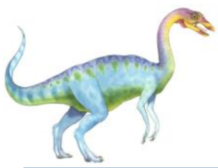
- Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

See Text Book

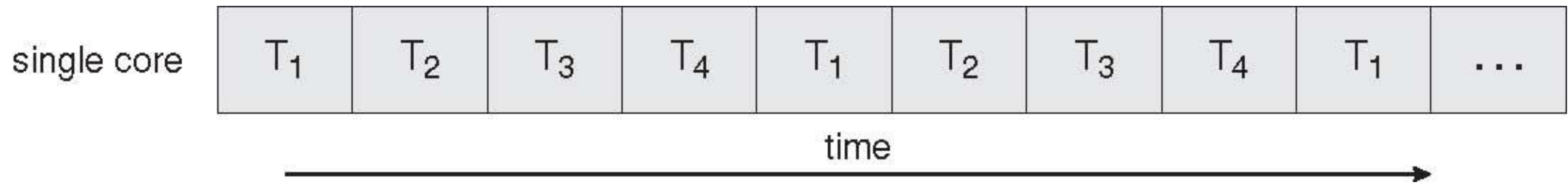


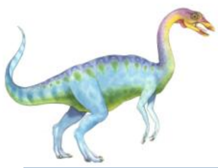
Multithreaded Server Architecture



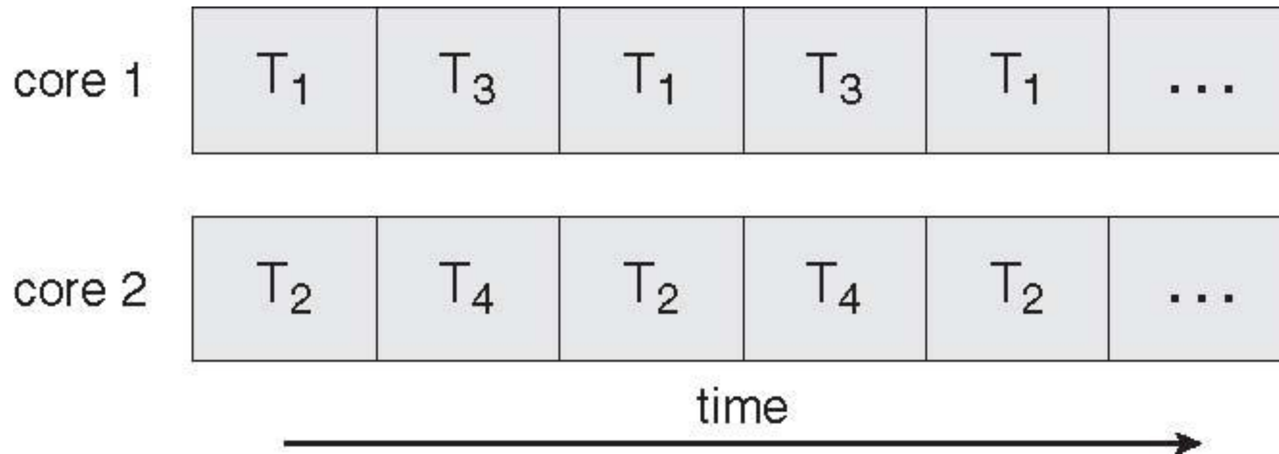


Concurrent Execution on a Single-core System





Parallel Execution on a Multicore System

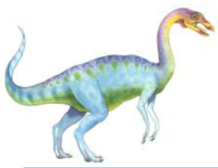


User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X



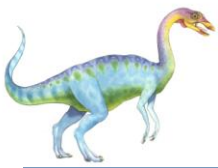
Multithreading Models

A relationship must exist between user threads and kernel threads in three common ways:

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

Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management is done by the thread library in user space
- Disadvantages:
  - The entire process will block if a thread makes a blocking system call
  - Multiple threads unable to run in parallel on multiprocessors because only one thread can access the kernel at a time.
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



Many-to-One Model

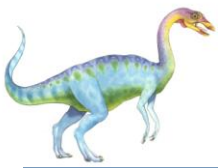
user thread

k

kernel thread

One-to-One

- Each user-level thread maps to kernel thread
- Allowing another thread to run when a thread makes a blocking system call
- Allowing multiple threads to run in parallel on multiprocessors
- Disadvantage:
  - Creating a user thread requires creating the corresponding kernel thread. (overhead of creating kernel threads can burden the application performance)
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

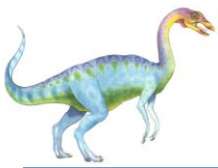


One-to-one Model

user thread

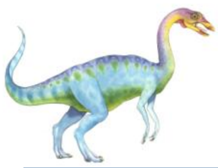
k k k k

kernel thread



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
- When a thread performs a blocking system call, the kernel can schedule another thread for execution
- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package



Many-to-Many Model

Two-level Model

- One popular variation on the M:M model, 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

user thread

k k k k

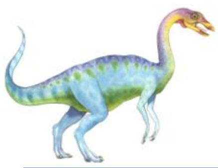
kernel thread

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
- Three main thread libraries are using today:
  - ***POSIX Pthreads*** (provided either as user-level or kernel-level)
    - Common in UNIX operating systems (Solaris, Linux, Mac OS X)
  - ***Win32 thread librarary*** (kernel-level library)
    - Available on Windows system
  - ***Java thread API*** (on the host OS)
    - are managed by the JVM
    - On Windows systems, it implemented using the Win32 API

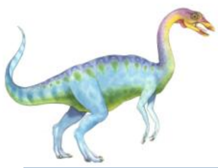
Threading Issues

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



Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
- Some UNIX systems have chosen two versions of fork():
  - Duplicate all threads
  - Duplicate only the thread that invoked the fork() system call
- The **exec()** system call typically works in the same way as described in chapter 3
- Depending on the Application:
  - If exec() is called immediately after forking, then duplicating only the calling thread.
  - If not call exec() after forking, the separate process should duplicate all threads.



Thread Cancellation

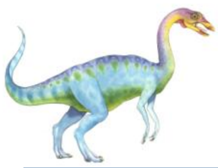
- Terminating a thread before it has finished
- A thread that is to be canceled is often referred to as the **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- 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 of delivering signals in multithreaded programs:
  - 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

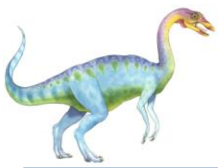
Thread Pools

- Create a number of threads at process startup and place them into a pool where they sit and wait for work
- When a server receives a request,
  1. It awakens a thread from the pool – if one is available – and passes it the request for service.
  2. Once the thread completes its service, it returns to the pool and awaits more work.
  3. If the pool contains no available thread, the server waits until one becomes free
- 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)



Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- One scheme for communication between user-thread library and kernel known as Scheduler activations
- Scheduler activations provide upcall procedure
  1. The kernel provides an application with a set of virtual processors (LWPs)
  2. The application schedule user threads onto an available virtual processor
  - The kernel must inform an application about certain events
- Upcall are handled by the thread library with upcall handler, which running on a virtual processor.
- This communication allows an application to maintain the correct number kernel threads

End of Chapter 4