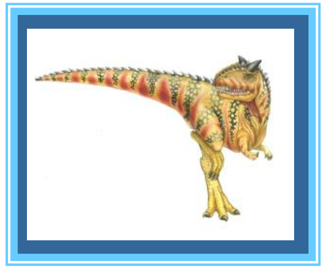
Chapter 4: Threads & Concurrency
（线程和并发）



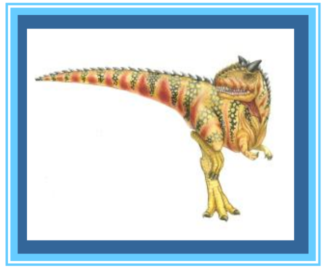
Chapter 4: Threads & Concurrency

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

Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designng multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux operating systems represent threads
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs

4.1 Overview



Overview

- The concept of a process as embodying two characteristics :
  - Unit of Resource ownership （资源拥有单位）
    - 给每个进程分配一虚拟地址空间，保存进程映像，控制一些资源（文件，I/O设备），有状态、优先级、调度
  - Unit of Dispatching （调度单位）
    - 进程是由一个或多个程序的一次执行
    - 可能会与其他进程交替执行
- These two characteristics are treated independently by the operating system
  - Dispatching is referred to as a thread or lightweight process （轻型进程LWP）
  - Resource of ownership is referred to as a processor task

线程(Thread)概念

- These two characteristics are treated independently by the operating system
  - 资源拥有单元称为进程（或任务），调度的单位称为**线程**、又称**轻型进程**（light weight process）。
  - 线程只拥有一点在运行中必不可省的资源（程序计数器、一组寄存器和栈），但它可与同属一个进程的其它线程共享进程拥有的全部资源。
- **线程定义为进程内一个执行单元或一个可调度实体。**

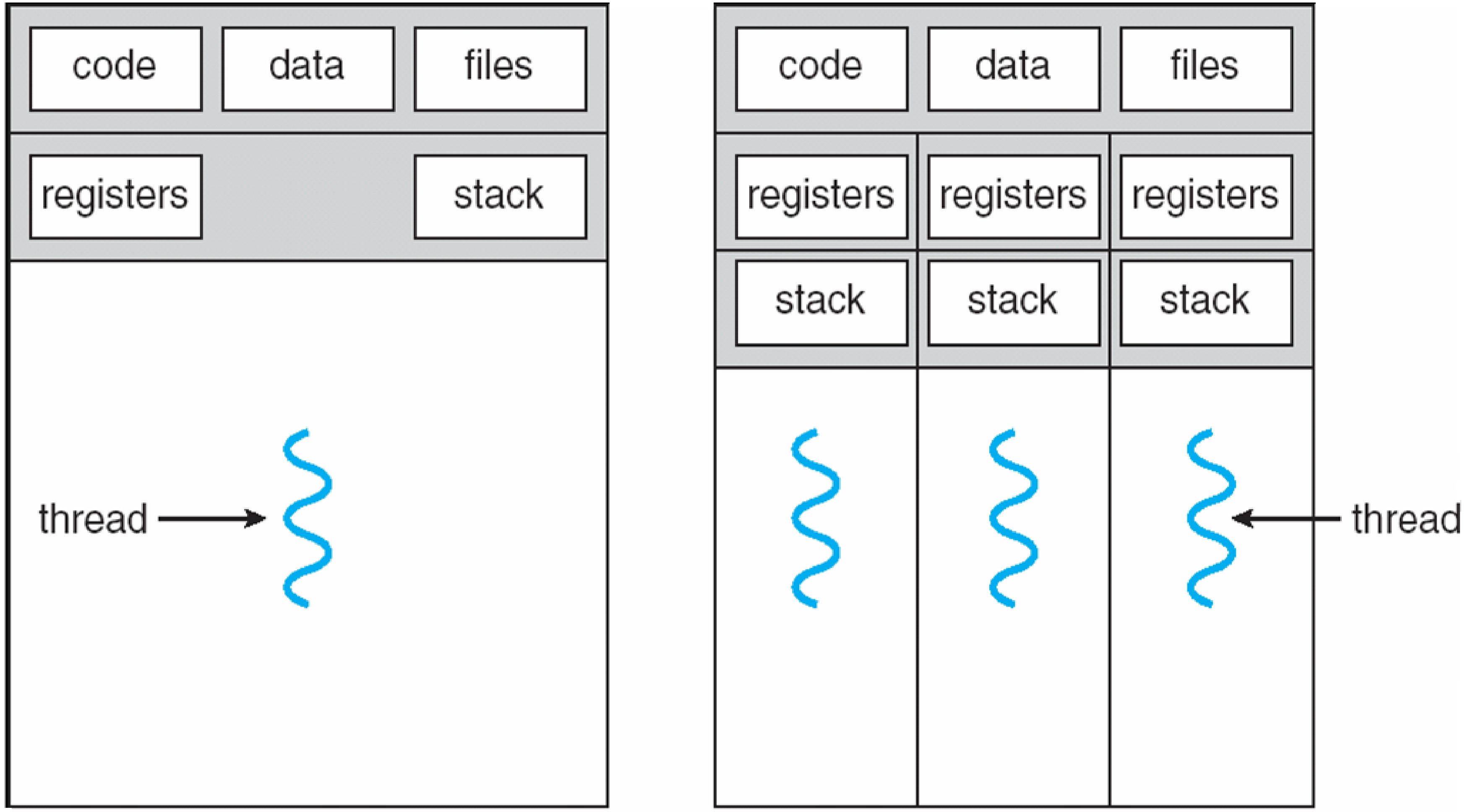
线程

- 线程：
  - 有执行状态（状态转换）
  - 不运行时保存上下文
  - 有一个执行栈
  - 有一些局部变量的静态存储
  - 可存取所在进程的内存和其他资源
  - 可以创建、撤消另一个线程

线程的特点

- 拥有少量的**系统资源**（资源是分配给进程）
- 一个进程中的多个线程可并发执行（进程可创建线程执行同一程序的不同部分）
- 系统开销小、切换快。（进程的多个线程都在进程的地址空间活动）

Single and Multithreaded Processes



single-threaded process

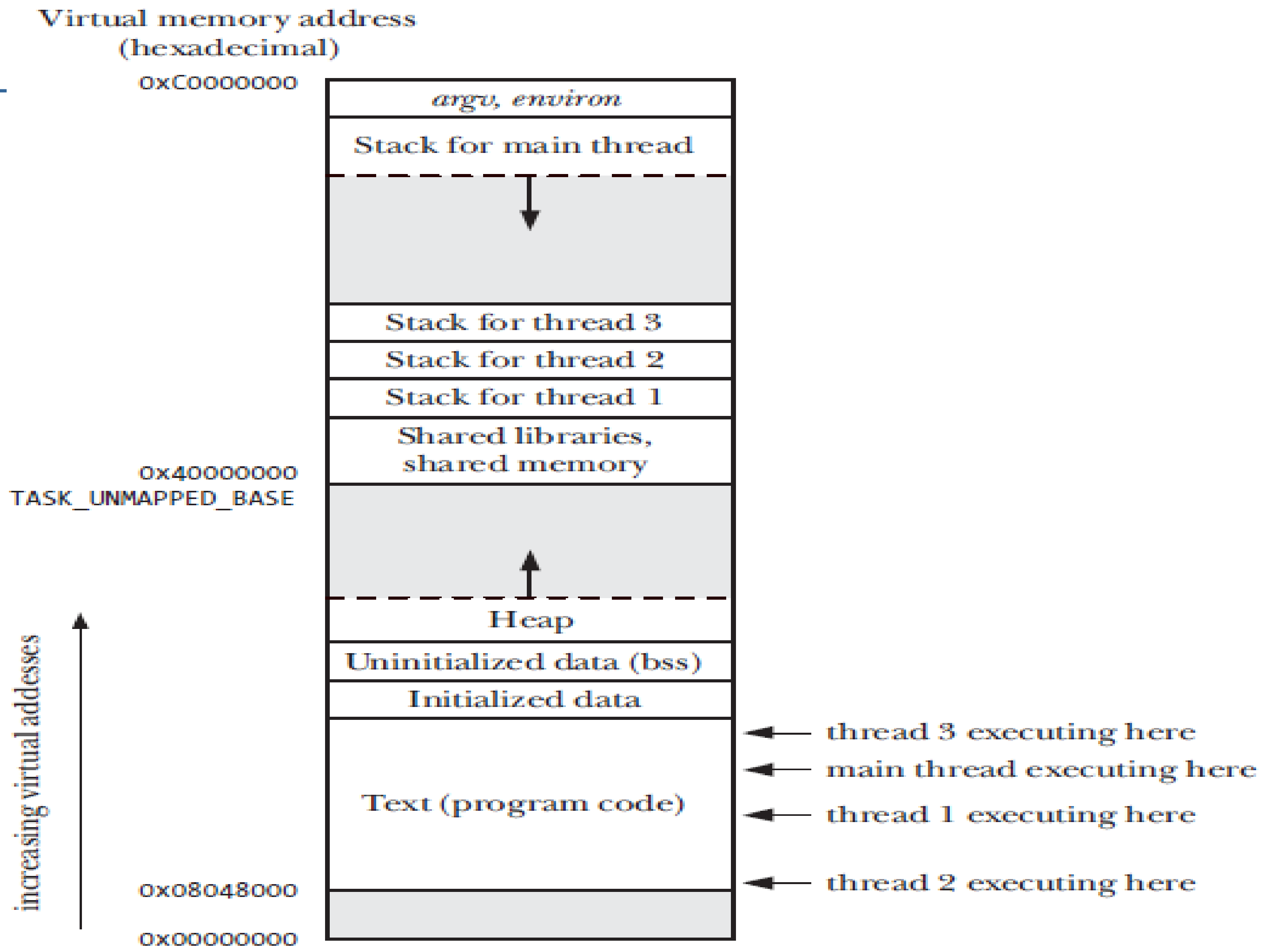
multithreaded process

Thread

- **A thread shares with threads belonging to the same process its:**
  - **code section**
  - **data section**
  - **operating-system resources**

(Process Have a virtual address space which holds the process image Protected access to processors, other processes, files, and I/O resources)

- **A traditional or *heavyweight* process （重型进程）is equal to a task with one thread**



Four threads executing in a process (Linux/x86-32)

Benefits

- **创建一个新线程花费时间少**（结束亦如此）
- **两个线程的切换花费时间少**

（如果机器设有“存储[恢复]所有寄存器”指令，则整个切换过程用几条指令即可完成）

- 因为同一进程内的**线程共享内存和文件**，因此它们之间相互通信无须调用内核
- **适合多处理机系统**

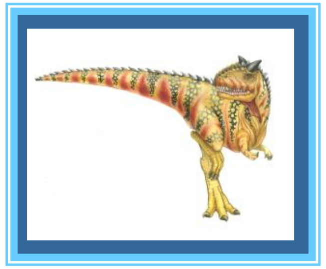
例子1：

- LAN中的一个文件服务器，在一段时间内需要处理几个文件请求
  - 有效的方法是：为每一个请求创建一个线程
  - 在一个SMP机器上：多个线程可以同时在不同的处理器上运行

例子2：

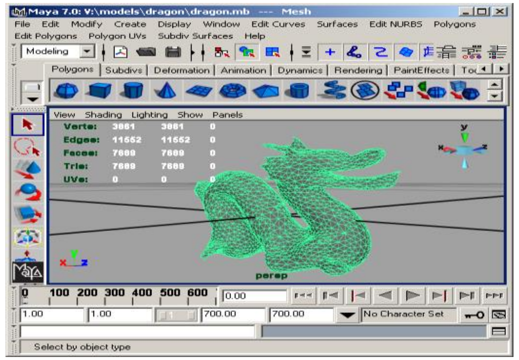
- 一个线程显示菜单，并读入用户输入；另一个线程执行用户命令
  - 考虑一个应用：由几个独立部分组成，这几个部分不需要顺序执行，则每个部分可以以线程方式实现
  - 当一个线程因I/O阻塞时，可以切换到同一应用的另一个线程

4.2 Multicore Programming



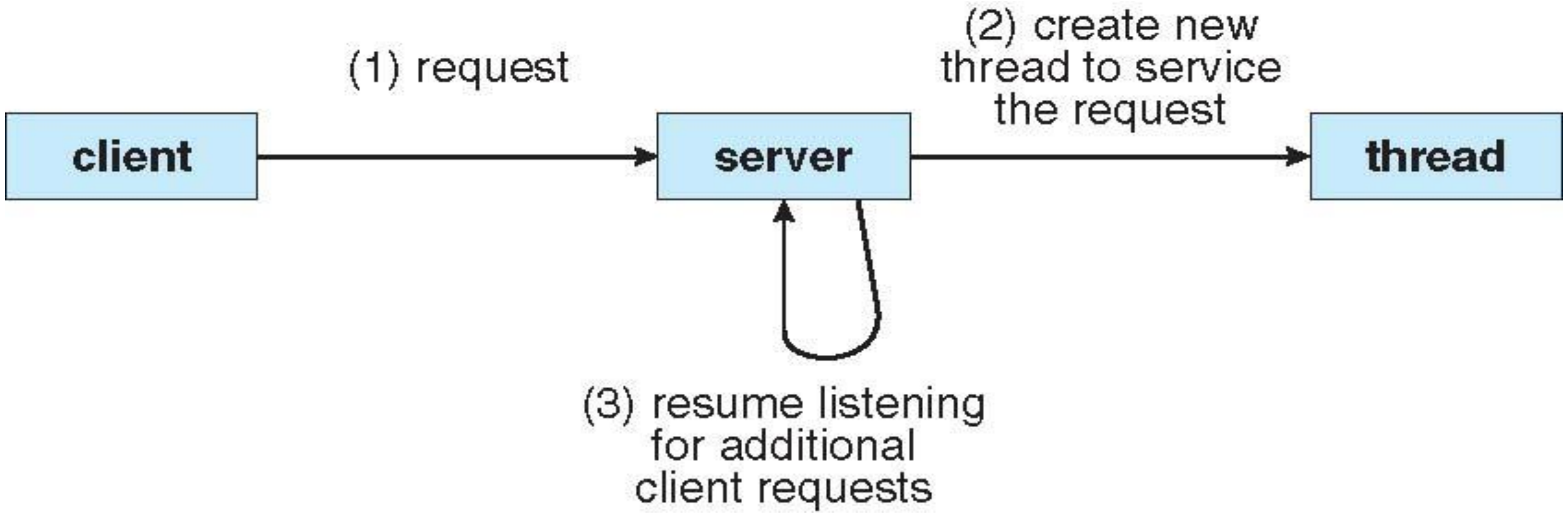
Multicore Programming

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

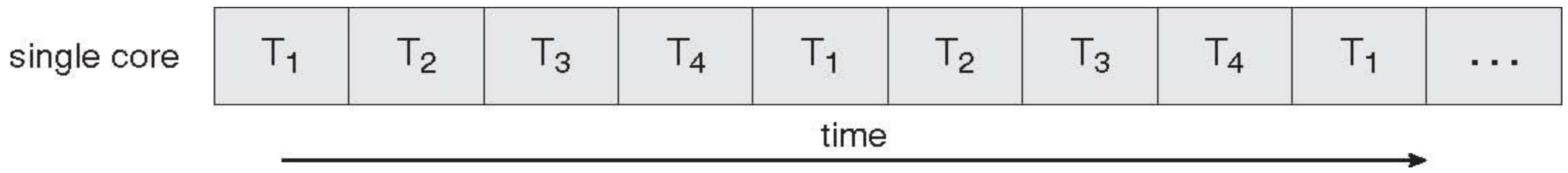


Each can run on its own core

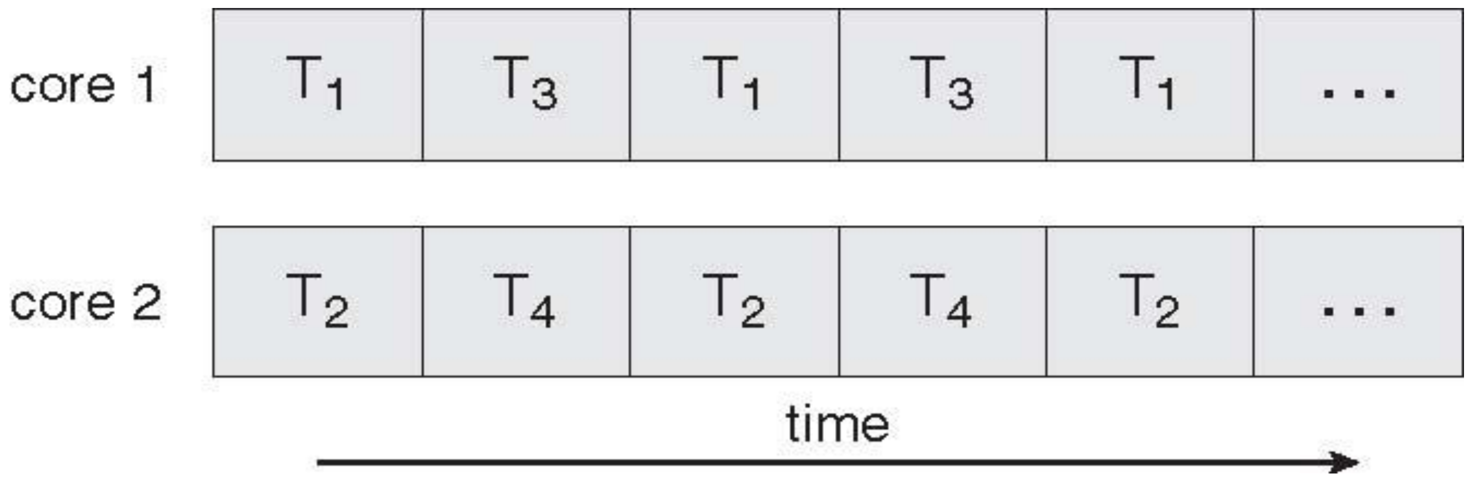
Multithreaded Server Architecture



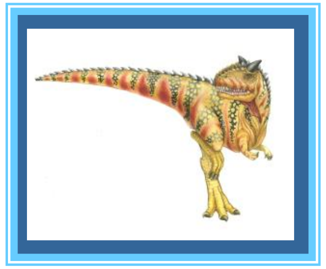
Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



4.3 Multithreading Models



线程的实现机制

- 用户级线程 user-level thread
- 内核级线程 kernel-level thread
- 两者结合方法

User Threads（用户级线程）

- **用户级线程：**不依赖于OS内核（内核不了解用户线程的存在），应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。

  如：数据库系统informix、图形处理Aldus，调度由应用软件内部进行，通常采用非抢先式和更简单的规则，也无需用户态/核心态切换，所以速度特别快。一个线程发起系统调用而阻塞，则整个进程在等待。

  - 用户线程的维护由应用进程完成；
  - 内核不了解用户线程的存在；
  - 用户线程切换不需要内核特权；
  - 用户线程调度算法可针对应用优化；
  - 一个线程发起系统调用而阻塞，则整个进程在等待。（一对多模型中）

- Three primary thread libraries:
  - POSIX Pthreads 、 Win32 threads、 **Java threads**

Kernel Threads （内核级线程）

- 内核级线程：依赖于OS核心，由内核的内部需求进行创建和撤销，用来执行一个指定的函数。一个线程发起系统调用而阻塞，不会影响其他线程。时间片分配给线程，所以多线程的进程获得更多CPU时间。
  - 内核维护进程和线程的上下文信息；
  - 线程切换由内核完成；
  - 时间片分配给线程，所以多线程的进程获得更多CPU时间；
  - 一个线程发起系统调用而阻塞，不会影响其他线程的运行。
- **Examples**
  - Windows XP/2000 及以后
  - Solaris
  - Linux
  - POSIX Pthreads
  - Mac OS X

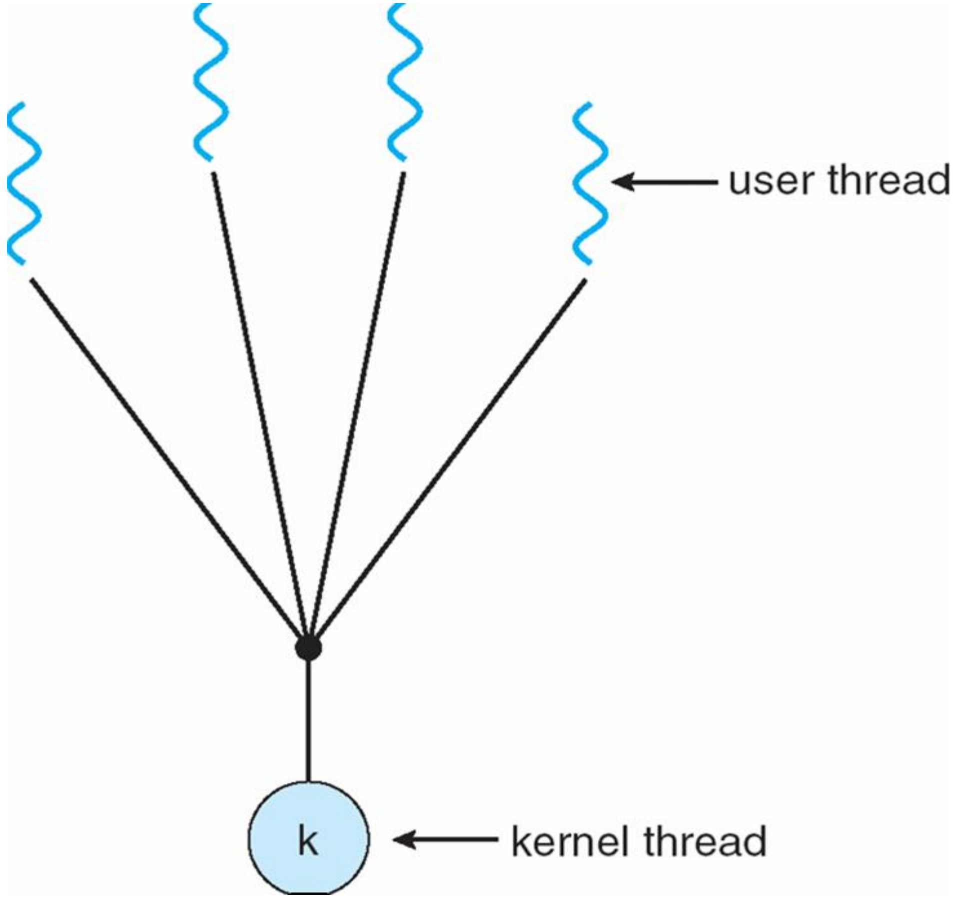
Multithreading Models

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

Many-to-One

- Many user-level threads mapped to single kernel thread
- Implemented by user-level runtime libraries
  - Create, schedule, synchronize threads at user-level
- OS is not aware of user-level threads
  - OS thinks each process contains only a single thread of control
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

Many-to-One Model



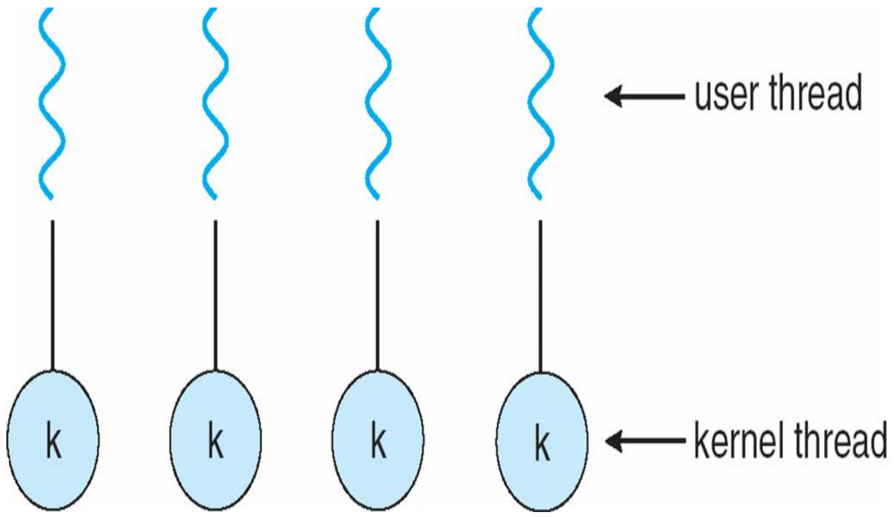
- **Advantages**
  - **Does not require OS support**
  - **Can tune scheduling policy to meet application (user level) demands**
  - **Lower overhead thread operations since no system calls**
- **Disadvantages**
  - **Cannot leverage multiprocessors (no true parallelism)**
  - **Entire process blocks when one thread blocks**

One-to-One

- Each user-level thread maps to kernel thread
- OS provides each user-level thread with a kernel thread
- Each kernel thread scheduled independently
- Thread operations (creation, scheduling, synchronization) performed by OS
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

One-to-one Model

- **Advantages**
  - **Each kernel-level thread can run in parallel on a multiprocessor**
  - **When one thread blocks, other threads from process can be scheduled**
- **Disadvantages**
  - **Higher overhead for thread operations**
  - **OS must scale well with increasing number of threads**



Many-to-Many Model

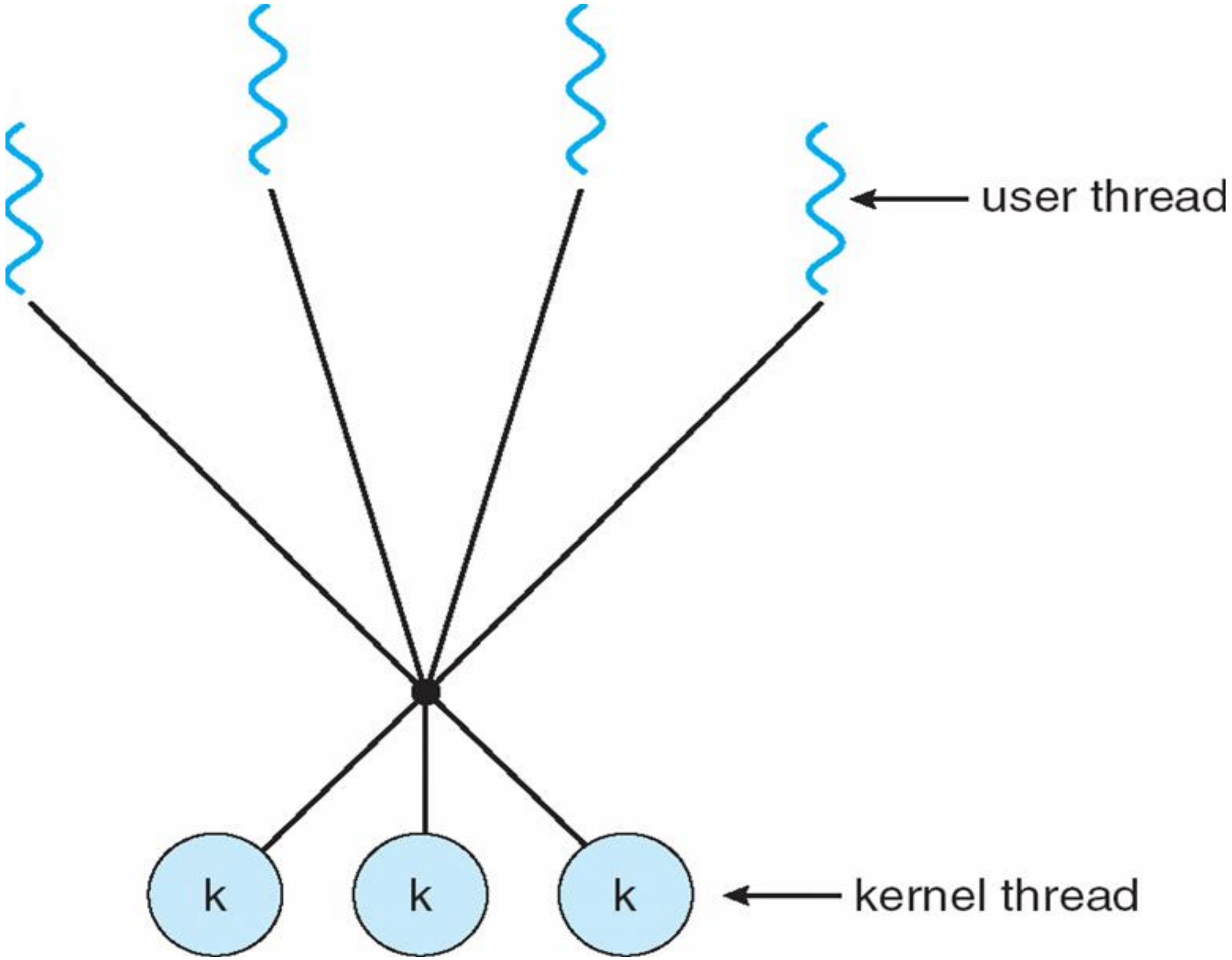
- **Allows many user level threads to be mapped to many kernel threads**
- **Allows the operating system to create a sufficient number of kernel threads**

- **Examples：**

  Solaris prior to version 9

  Windows NT/2000 with the ThreadFiber package

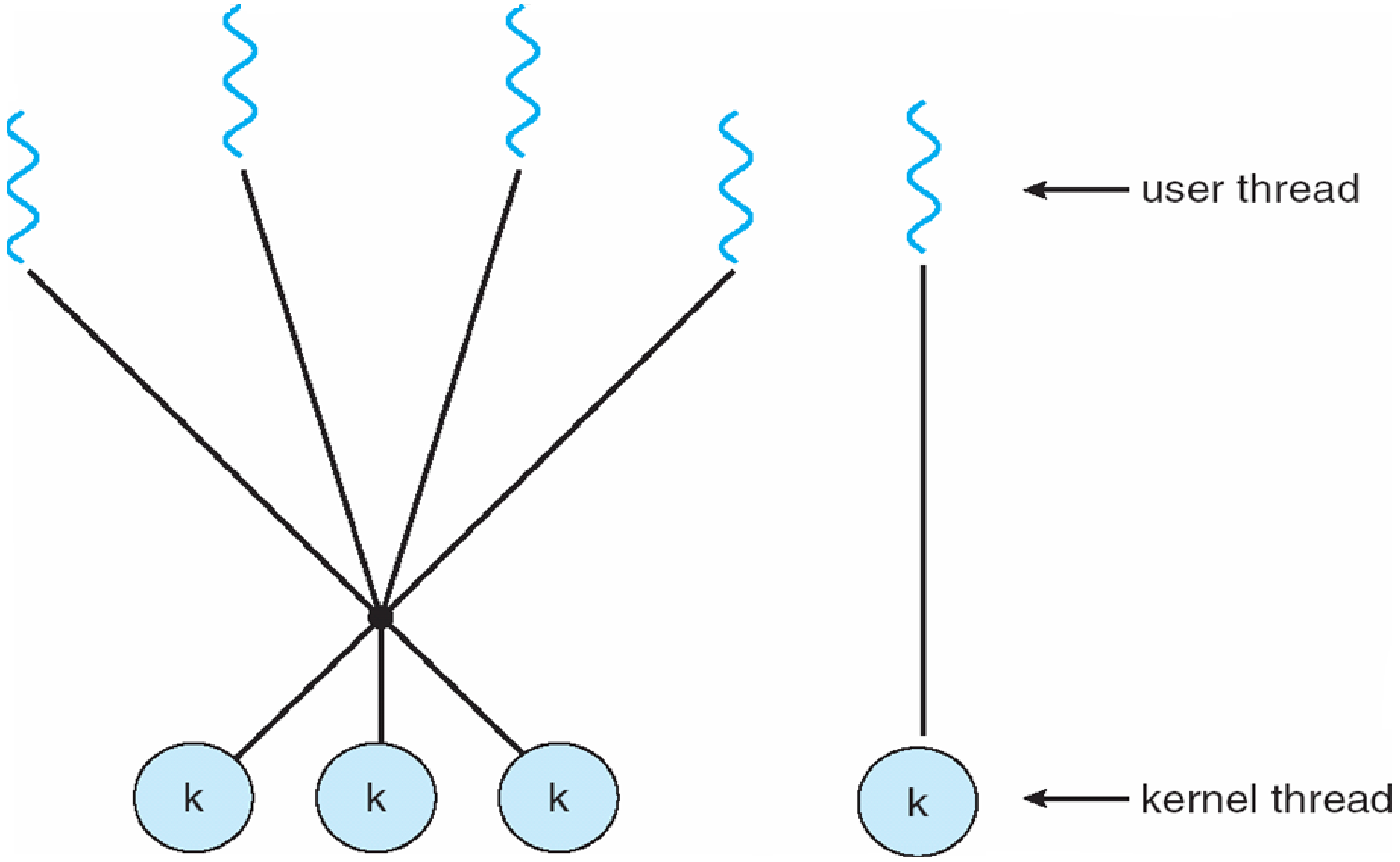
Many-to-Many Model



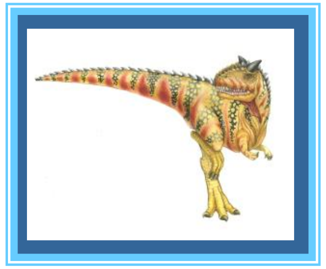
Two-level Model

- 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



4.4 Thread Libraries



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

Thread Libraries

- Three main thread libraries are in use today:
  - Windows
  - POSIX Pthreads
  - Java

- The Windows thread library is a kernel-level library available on Windows systems.

Windows thread library

- The Windows thread library is a kernel-level library available on Windows systems.
- Threads are created in the Windows API using the CreateThread() function

Example： **createthread.cpp**

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX （Portable Operating System Interface） standard (IEEE 1003.1c) API for thread creation and synchronization

  http://standards.ieee.org/reading/ieee/stad_public/description/posix
- 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)

POSIX线程库Pthreads

Pthreads

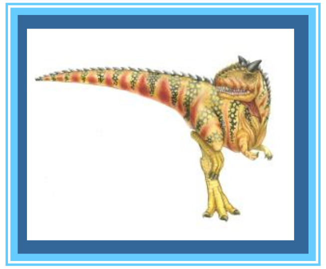
- POSIX线程库(Pthreads):
  - pthread_create(): 创建线程函数;
  - pthread_exit(): 主动退出线程;
  - pthread_join(): 用于将当前线程挂起来等待线程的结束。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源就被收回。
  - pthread_cancel(): 终止另一个线程的执行。

例：**thread.c**

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

4.5 Implicit Threading(隐式多线程)



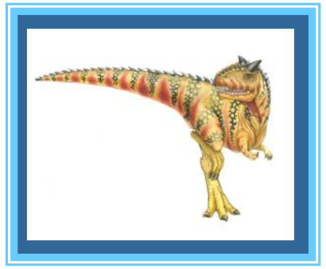
Implicit Threading

- 多核系统多线程编程，一个应用程序有可能有几百个甚至上千的线程，这样的程序面临许多挑战：
  - 编程挑战：任务分解、任务的工作量平衡、数据分割、数据依赖、测试与调试
  - 程序执行顺序的正确性问题：同步、互斥
- **策略：** 隐式线程Implicit Threading，当前一种流行趋势。
  - 将线程的创建与管理交给**编译器**和**运行时**库来完成

Implicit Threading

- 几种隐式线程的设计方法：
  - **Thread Pools**（线程池）
  - **Fork Join**
  - **OpenMP**，用于共享内存并行系统的多处理器程序设计的一套指导性编译处理方案。OpenMP支持的编程语言包括C、C++和Fortran。
  - **Grand Central Dispatch**（GCD，大中央调度），Apple for its macOS and iOS operating systems.
  - **Intel Thread Building Blocks**（TBB），Intel开发的构建多线程库，open source；TBB是一个可移植的C++库，能够运行在Windows、Linux、Macintosh以及UNIX等系统上。
  - Java

4.6 Threading Issues



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()

fork()

- duplicate only the calling thread or all threads?

exec()

- Replaces the process - including all threads?

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（信号处理）

- 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 threa to receive all signals for the process

Thread Pools（线程池）

- 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

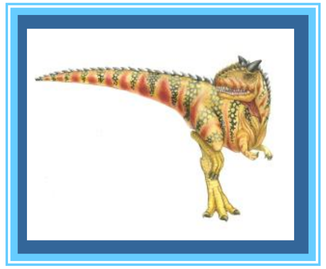
Thread Specific Data（线程特有数据）

- 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
- 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

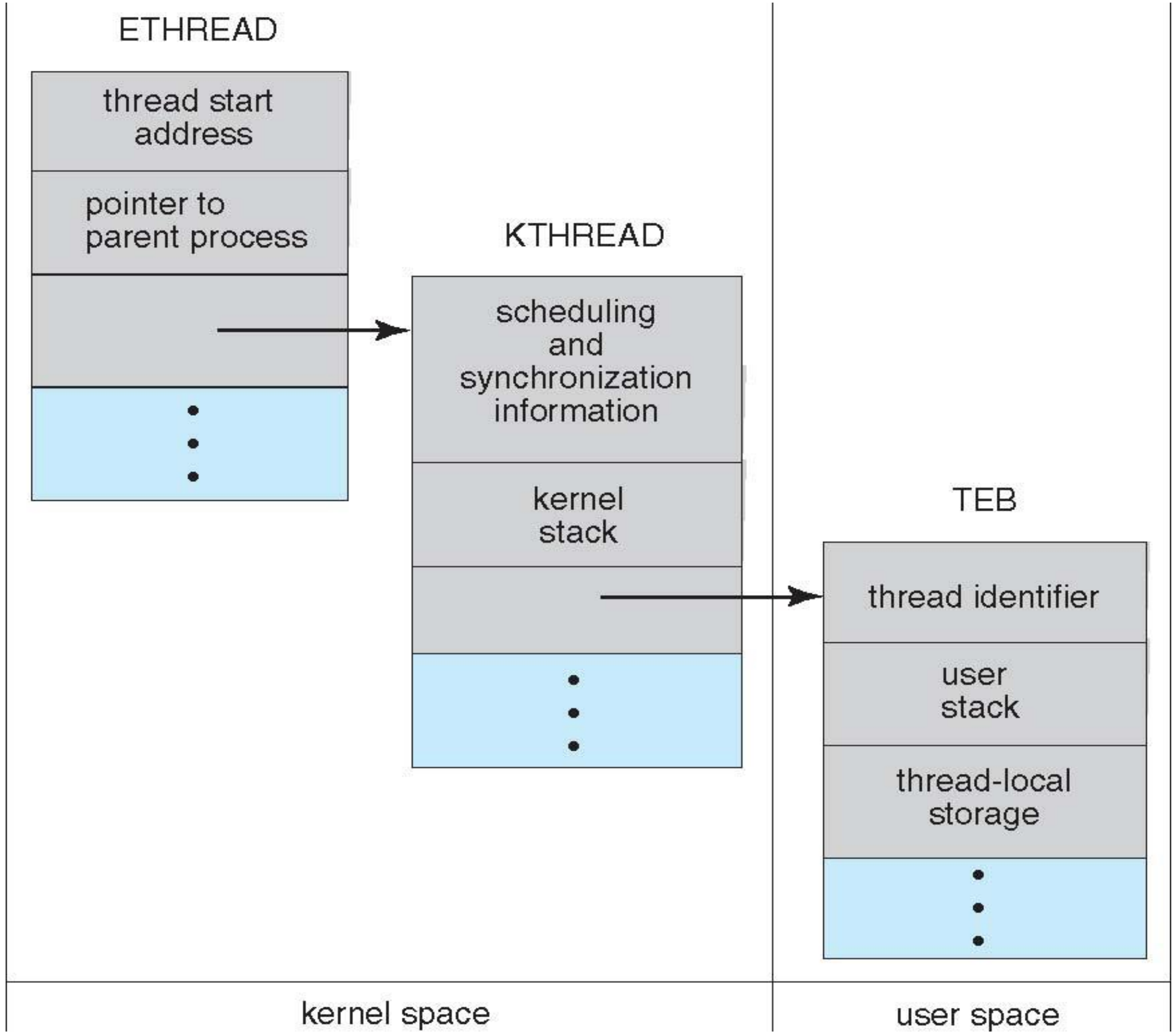
4.7 Operating-System Examples



Operating System Examples

- **Windows XP Threads**
- **Linux Thread**

Windows XP Threads



Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- 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)

CreateThread.cpp

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)

Linux Threads

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.

Exercise

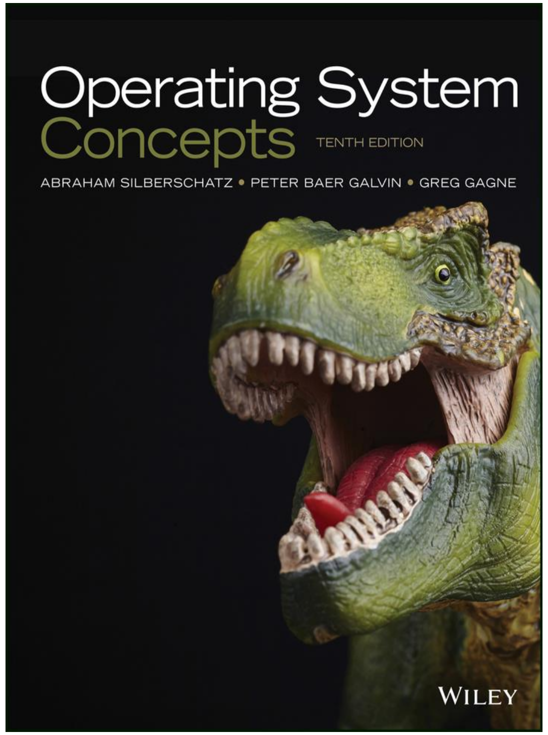
- 进程、线程习题分析

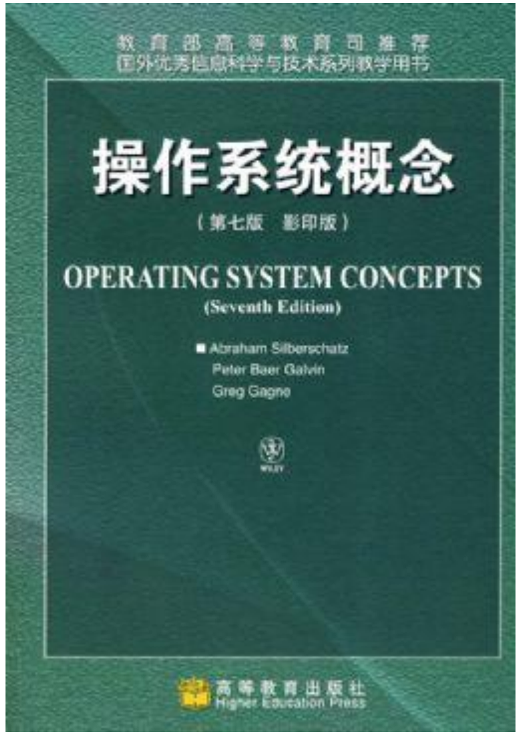
Homework

- 按时完成“学在浙大”中的作业

Reading Assignments

- Read for this week:
  - Chapters 4
    of the text book:
- Read for next week:
  - Chapters 5
    of the text book:





End of Chapter 4