第4章 线程

目录

线程的引入

- 在操作系统中引入进程的目的是使多道程序能并发执行，以改善资源利用率及提高系统吞吐量；

- 在操作系统中再引入线程，则是为了减少程序并发执行所付出的时空开销，使操作系统具有更好的并发性。

线程的引入

- 早期进程概念中，进程具有两个属性：
  - 拥有资源的独立单位
  - 调度和分派的基本单位
- 线程的引入：**寻找最经济的并发**
  - 一个应用程序往往可分解为多个子任务
    - 浏览器：更新显示内容+网络数据接收
    - Word：响应用户输入+后台拼写检查
  - 一个应用程序也可能执行多个类似任务
    - Web Server：面对大量类似的网络请求处理
  - 进程的并发执行(创建、切换等)，涉及到资源管理，花费很大的时空开销
  - 现代OS的内核大多采用了多线程而非多进程

4.1 线程定义

- 线程的定义情况与进程类似，存在多种不同的提法。下面列出一些较认可的定义：
  - 线程是进程内的一个执行单元。
  - 线程是进程内的一个可调度实体。
  - 线程是程序（或进程）中相对独立的一个控制流序列。
  - 线程是执行的上下文，是执行的现场数据和其他调度所需的信息（这种观点来自Linux系统）。

线程定义(Cont.)

- 本书的定义
  - 线程是CPU使用的基本单位
  - 它由线程ID、程序计数器、寄存器集合和栈组成。
  - 它与属于同一进程的其他线程共享代码段、数据段和其他操作系统资源。

  - 线程是进程内一个相对独立的、可调度的执行单元。
  - 线程自己基本上不拥有资源，只拥有一点在运行时必不可少的资源（如程序计数器、一组寄存器和栈），但它可以与同属一个进程的其他线程共享进程拥有的全部资源。

线程与进程的比较

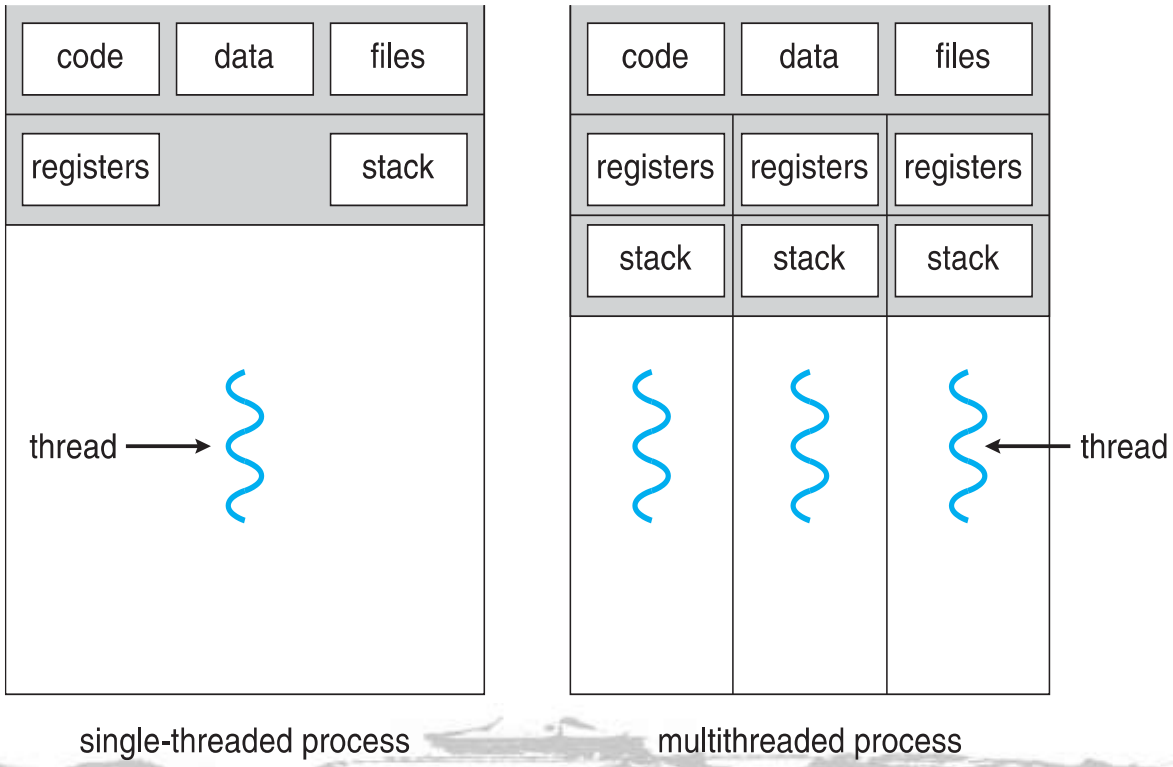
- 进程中的线程具有
  - 执行栈：用于切换时存储上下文
  - 寄存器及对所属进程资源的访问
  - ~~代码段~~
  - ~~数据段（静态数据段）~~

线程与进程的比较

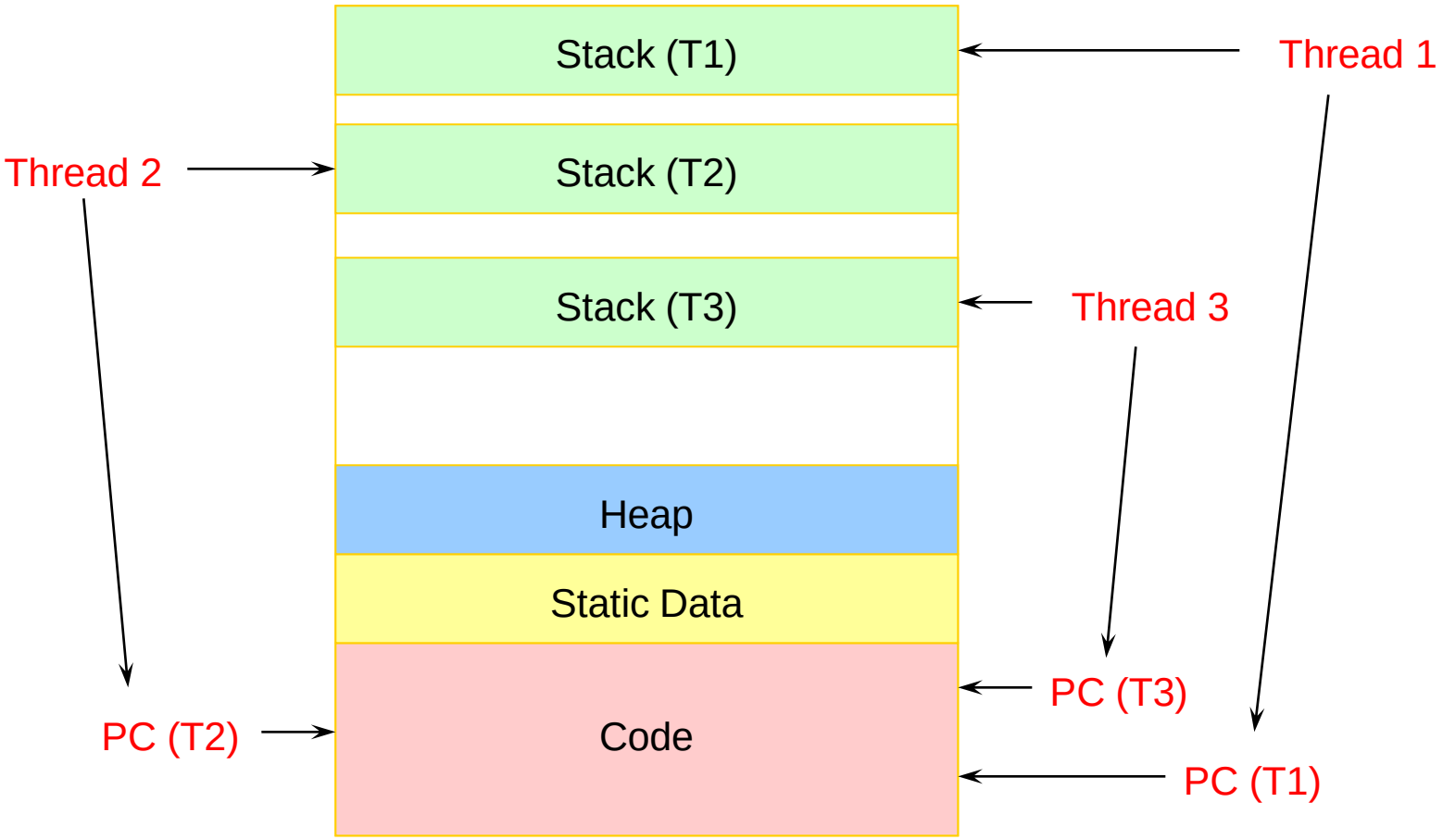
- 调度
  - 在传统OS中，进程是调度和分配资源的基本单位；
  - 引入线程后，线程是调度和分派的基本单位，进程是拥有资源的基本单位。
- 拥有资源
  - 进程是拥有资源的基本单位，由一个或多个线程及相关资源构成。
- 系统开销
  - 进程创建、撤销及切换均涉及资源管理，开销大程
  - 线程创建只涉及很少一部分的资源管理；同一进程的线程间同步与通信开销小

多线程

- **多线程**是指一个进程中有多个线程，这些线程共享该进程的状态和资源，它们驻留在同一地址空间，并且可以访问到相同的数据。



进程中的多线程


Stack (T1)
Stack (T2)
Stack (T3)
Heap
Static Data
Code
Thread 1
Thread 2
Thread 3
PC (T3)
PC (T2)
PC (T1)

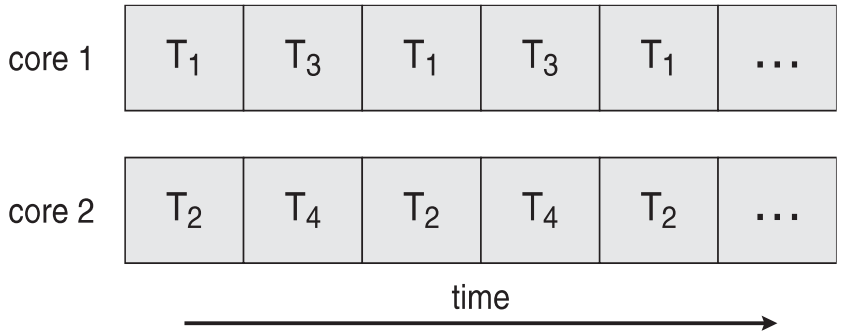
多线程的优势

- 响应度好
  - 如果进程部分阻塞，可以允许这个程序继续执行，如多线程浏览Web时候，一个线程装载图片，可以利用另外一个线程接受用户交互
- 资源共享
  - 线程默认共享进程的内存和资源，代码、数据的共享
  - 允许在同样一个空间上，有不同的活动线程，方便消息传递
- 经济性
  - 比进程创建更简单，上下文切换的负载小
  - 如Soloaris，创建：进程比线程慢30倍，切换：慢5倍
- 可伸缩性
  - 能够更好地利用多核体系结构优势，可以使得多线程能够并行在不同处理器核上运行。而单线程进程则只能运行在一个处理器上。

Concurrency vs. Parallelism

- 在单核心系统上的并发执行

- 在多核心系统上的并行执行



Amdahl’s Law

- 额外的计算核心数量的增加能够对应用程序带来潜在性能的改善
  - S： 应用程序中串行执行的比例
  - N： 处理器核心

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

  - 简单推导：$\mathrm{T_{new}} = S \times \mathrm{T_{old}} + (1 - S) \times \frac{\mathrm{T_{old}}}{N}$

- 当应用程序并行与串行比例3:1时
  - 处理器核心2个，1.6倍提速
  - 处理器核心4个，2.28倍
- 当N无穷大，加速接近 1/S，串行执行部分对应用程序并行性能影响很大
- 局限性：理想情况，没有考虑到现实硬件发展

多核编程的挑战

- 多核编程的挑战
  - 任务分解：识别哪些任务是可独立、并发的
  - 平衡：识别任务的重要价值，平衡资源使用
  - 数据分割：把数据分配到独立的核
  - 数据依赖：分析任务依赖，确保多任务间的同步
  - 测试与调试：多核更复杂
- 并行类型
  - 数据并行：把数据分布到多个核心，在每个核心执行相同操作
  - 任务并行：让每个任务执行各自的操作

4.2 多线程模型

- 操作系统中有多种方式实现对线程的支持：
  - 内核级线程
  - 用户级线程
  - 两种方法的组合实现

内核级线程

- 内核级线程是指依赖于内核，由操作系统内核完成创建和撤消工作的线程。
- 在支持内核级线程的OS中，内核维护进程和线程的上下文信息并完成线程切换。
- 一个内核级线程阻塞时不会影响同一进程的其他线程的运行。*Why?*
- 处理机时间分配对象是线程，所以有多个线程的进程将获得更多处理机时间。

内核级线程

- 内核级线程的限制
  - 内核级线程的管控需要通过系统调用来实现，过细粒度的内核级线程并发会带来性能的下降（频繁的模式切换）

用户级线程

- 用户级线程是指不依赖于操作系统核心，由应用进程利用**用户级线程库**提供创建、同步、调度和管理线程的函数来控制的线程。

- 用户级线程的维护由应用进程完成，可以用于不支持内核级线程的操作系统
  - 用户级线程库实施了用户级线程创建、调度等

用户级线程

- 用户级线程对OS不可见，OS调度的依然是进程
- 在调度时，由线程库来切换TCB, PC, regs, stack
  - 均是过程调用来实现，不涉及模式切换
- 优势：速度快100x
- 限制
  - 当一个线程阻塞时，整个进程都必须等待， Why?
  - 处理机时间分配对象是进程，每个用户级线程的执行时间相对少一些

讨论：

1. 纯用户级的线程如何调度？
2. 纯用户级线程间会存在抢占吗？（定时器能用于在用户级线程之间回收控制和调度吗？）
3. 如果一个用户级线程出现了阻塞，同进程的其他线程有机会运行吗，怎么办？

纯用户级线程不支持抢占，除非主动的放弃处理器（完成退出或通过调度函数）

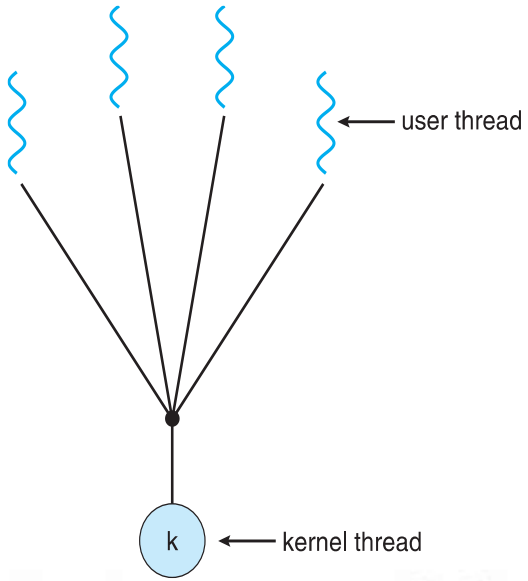
Thread_yield()

两种方法的组合

- 在有些系统中，提供了上述两种方法的组合实现。
- 在这种系统中，内核支持多线程的建立、调度与管理；同时，系统中又提供使用线程库的便利，允许用户应用程序建立、调度和管理用户级的线程。
- 因此可以很好地将内核级线程和用户级线程的优点结合起来。由此产生了不同的多线程模型。

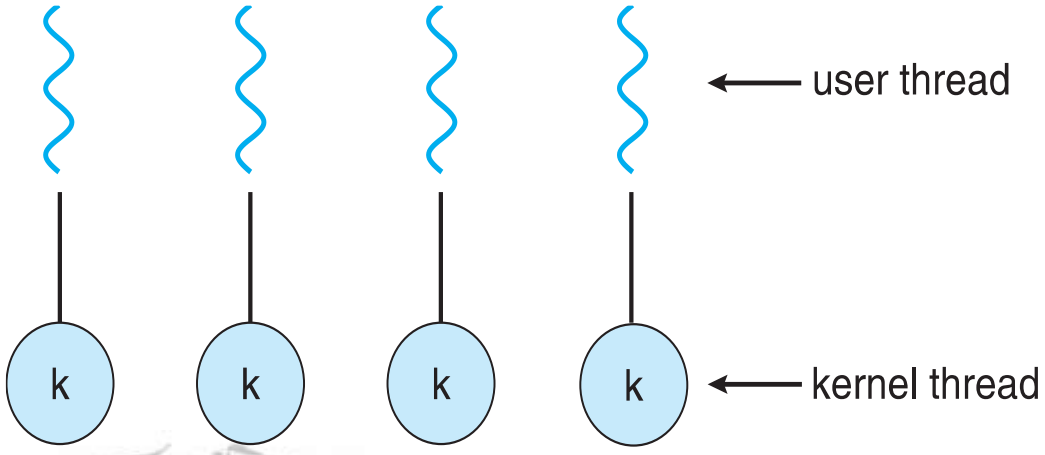
多对一模型

- 多个用户级线程映射到一个内核级线程上
  - 线程管理由线程库在用户空间进行
  - 一个用户线程若执行了阻塞系统调用，则整个进程会阻塞，Why?
  - 任一时刻一次只有一个线程能够访问内核
  - 无法利用多核处理器目前很少系统采用这种模型
  - 典型例子
    - Solaris Green Threads
    - GNU Portable Threads



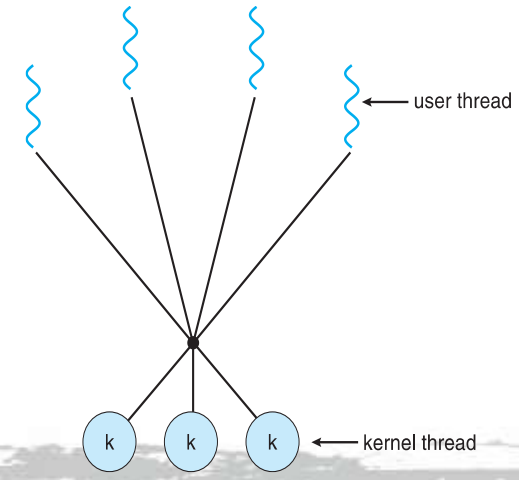
一对一模型

- 每个用户级线程映射到一个内核级线程上
  - 一个用户线程阻塞时，允许其他线程继续执行
  - 允许多线程并发运行在多处理器系统
  - 但，大量创建内核线程开销很大，因此实现中需要限制创建数量
  - Linux、Windows、Solaris 9系列都实现了一对一模型



多对多模型

- 多个用户级线程映射到较少或相等个数的内核级线程上
  - 可创建任意多的用户线程
  - 相应的内核线程能在多处理器系统上并发执行
  - 当一个线程执行阻塞系统调用，内核能调度另一个线程执行
  - 变种（二层模型）：既允许多对多，也允许一对一绑定



4.3 线程库

- 线程库为程序员提供了一套创建和管理线程的API
  - POSIX Pthreads、Win32 threads‘Java threads
- 通常有两种主要的方法来实现
  - 用户空间提供一套无内核支持的库
    - 库的所有代码和数据结构都在User spaces
    - 调用函数API后，只在用户空间调用，而不涉及系统调用
  - 操作系统支持的内核库
    - 库的代码和数据结构存在于内核空间
    - 调用函数API后，会触发System call

Pthreads

- POSIX线程（POSIX threads），简称Pthreads，是线程的POSIX标准。
- 该标准定义了创建和操纵线程的一整套API。在类Unix操作系统（Unix、Linux、Mac OS X等）中，都使用Pthreads作为操作系统的线程。
- Pthreads 可以提供用户和内核级的库。
- 常用的线程控制的函数有：
  - pthread_create 创建一个线程
  - pthread_join 等待一个线程的结束

```c
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1])
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

```c
/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

Win32 线程

- Win32 线程库是用于Windows系统的内核级线程库
- 常用的线程控制的函数有
  - CreateThread 创建线程
  - WaitForSingleObject 等待线程结束

```c
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer param
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0
      return -1;
   }

   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

   if (ThreadHandle != NULL) {
      /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

      /* close the thread handle */
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

Figure 4.11 Multithreaded C program using the Windows API.

Java

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}
```

4.4 操作系统例子

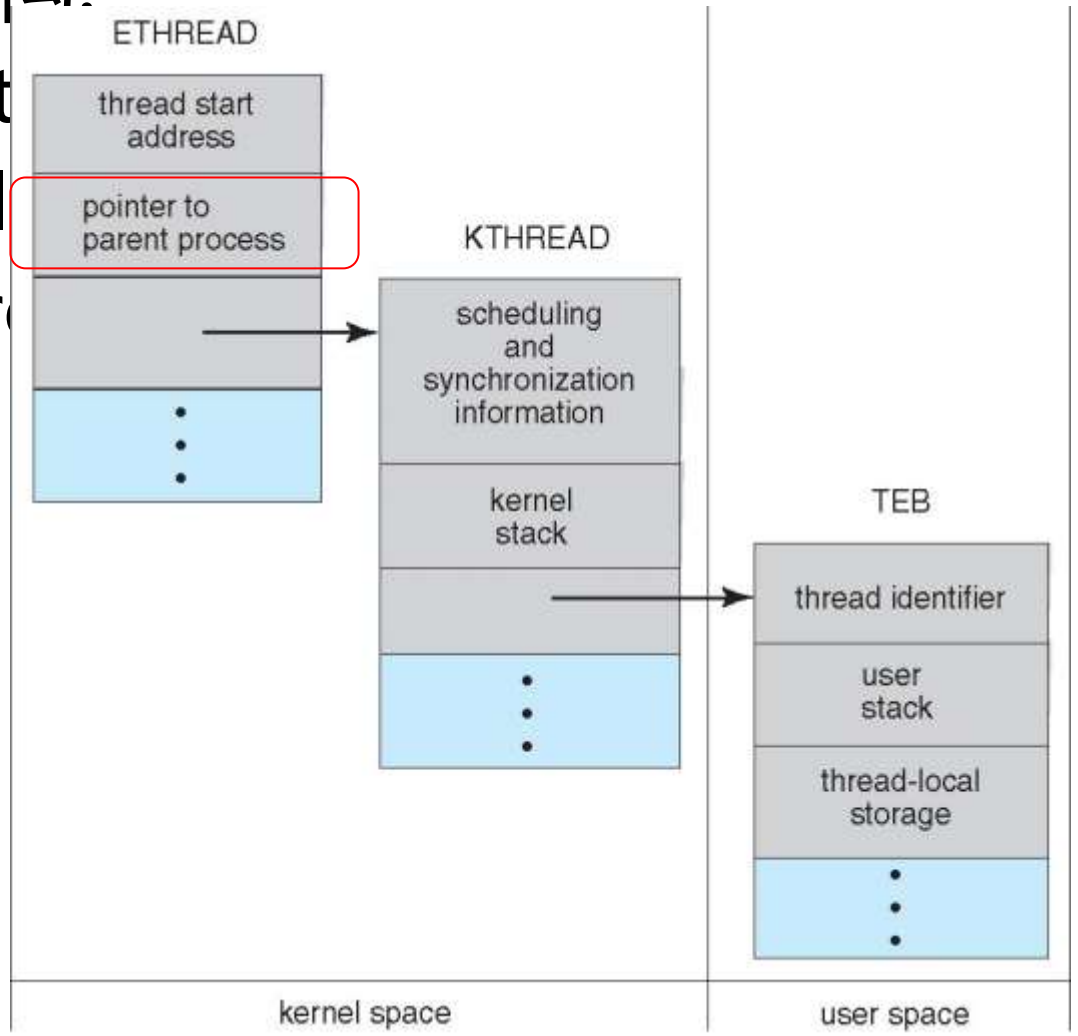
- Windows Threads
- Linux Thread

Windows 线程

- 一对一模型
- 每个线程包括
  - Thread id
  - 寄存器组
  - 用户栈和内核栈
  - 私有存储区

Windows 线程的数据结构

- 线程的主要数据结构:
  - ETHREAD (execut
  - KTHREAD (kernel
  - TEB (thread envir



Linux Threads

- Linux中 使用“任务”而不是 “线程”

- 线程的创建是通过系统调用 clone()来实现的
  - Fork与clone的区别，就在于是否共享地址空间和资源

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.

小作业3

- 课本P132　4.12

大作业3

- 在国产操作系统环境下，实现第4章课后编程项目项目1 or 项目2