上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

L3-1. 线程

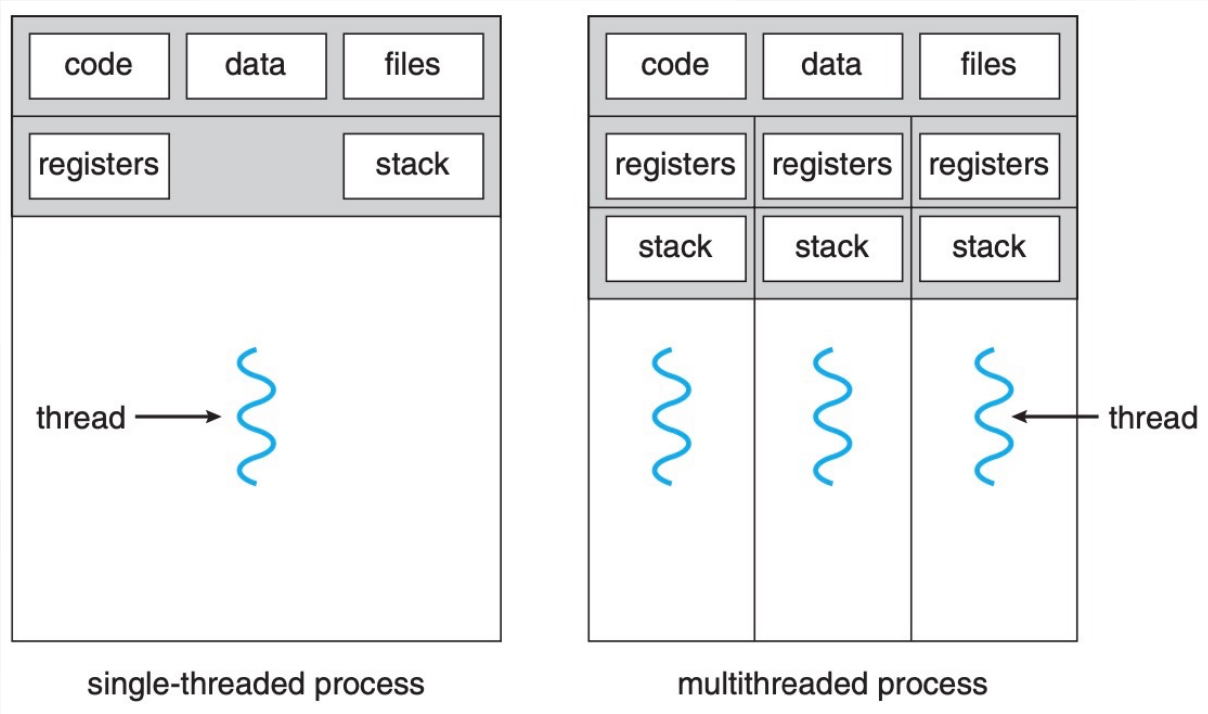
宋卓然

上海交通大学计算机系

songzhuoran@sjtu.edu.cn

什么是线程

- 每个线程是CPU使用的一个基本单元；它包括线程ID（tid）、程序计数器、寄存器组和堆栈。
- 它与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



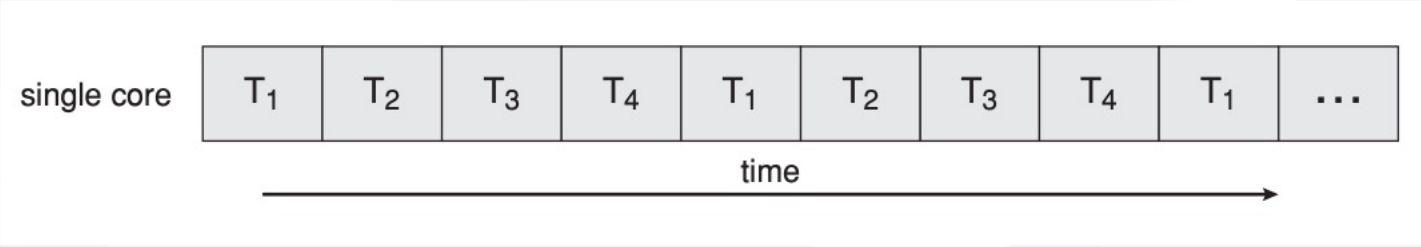
为什么要有线程

- 现代计算机运行的大多数应用软件都是多线程的
- 一个应用程序通常作为具有多个控制线程的一个进程来实现
  - 显示图像、文本
  - 接收数据
  - 拼写检查
- 进程创建往往是重量级的，而线程创建是轻量级的
- 操作系统的内核往往都是多线程的

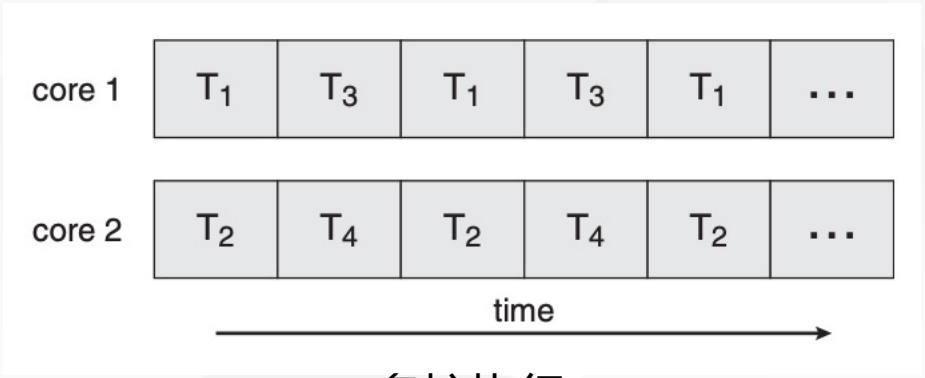
多线程的优点

- 响应性:如果一个交互程序采用多线程，那么即使部分阻塞或者执行冗长操作，它仍可以继续执行，从而增加对用户的响应程度
- 资源共享:进程只能通过如共享内存和消息传递之类的技术共享资源；而线程默认共享它们所属进程的内存和资源
- 经济:进程创建所需的内存和资源分配非常昂贵。由于线程能够共享它们所属进程的资源，所以创建和切换线程更加经济
- 可伸缩性:对于多处理器体系结构，多线程的优点更大，因为线程可在多处理核上并行运行

多核系统



单核执行



多核执行

多核速度更快

注意**并行性与并发性**的区别：并行系统可以同时执行多个任务。相比之下，并发系统支持多个任务，允许所有任务都能取得进展。因此，没有并行，并发也是可能的。在SMP 和多核架构出现之前，大多数计算机系统只有单个处理器。CPU调度器通过快速切换系统内的进程，以便允许每个进程取得进展，从而提供并行假象。这些进程并发运行，而非并行运行

Amdahl定律

- Amdahl定律是一个公式。对于既有串行也有并行组件的应用程序，该公式确定由于额外计算核的增加而潜在的性能改进。如果S是应用程序的一部分，它在具有N个处理核的系统上可以串行执行，那么该公式如下：

$$\text{加速比} \leqslant \frac{1}{S + \frac{1-S}{N}}$$

- 一个例子，假设我们有一个应用程序，其75%为并行而25%为串行。如果我们在具有两个处理核的系统上运行这个程序，我们能得到1.6 倍的加速比。如果我们再增加两核 (一共有4个)， 加速比是2.28倍。
  - S=0.25 N=2

Amdahl定律

- Amdahl定律是一个公式。对于既有串行也有并行组件的应用程序，该公式确定由于额外计算核的增加而潜在的性能改进。如果S是应用程序的一部分，它在具有N个处理核的系统上可以串行执行，那么该公式如下：

$$\text{加速比} \leqslant \frac{1}{S + \frac{1-S}{N}}$$

- Amdahl定律的一个有趣事实是，当N趋于无穷大时，加速比收敛到1/S。例如，如果应用程序的40%为串行执行，无论我们添加多少处理核，那么最大加速比为2.5倍。这是Amdahl定律背后的根本原则:对于通过增加额外计算资源而获得的性能，应用程序的串行部分可能具有不成比例的效果

并行类型

- 数据并行
  - 将数据分布于多个计算核，并在每个核进行相同操作
- 任务并行
  - 将任务（线程）分布于多个计算核，每个线程都执行一个独特的操作
  - 不同线程可以操作相同的数据，或者也可以操作不同的数据
- 通常混合两种并行方式进行计算

进程 vs. 线程

进程

- 独立
- 运行时携带更多状态信息
- 拥有独立的地址空间
- 上下文切换通常较慢

线程

- 为进程的子集
- 共享进程状态、存储、资源
- 共享进程的地址空间
- 上下文切换往往较快

线程类别

- 用户级线程
  - 由用户级线程库完成线程管理
  - 三种主要的线程库：POSIX Pthreads、Win32 threads、Java threads
- 内核级线程
  - 由操作系统内核完成线程管理
  - 现代操作系统Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X 都支持内核级线程
- **用户级线程与内核级线程的区别是?**

线程类别

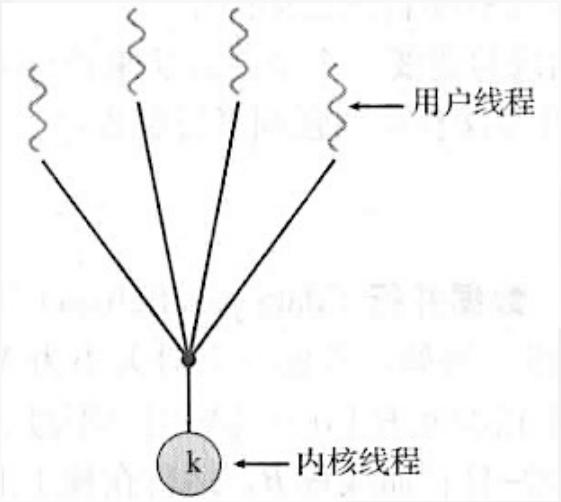
- 用户级线程可以认为是用户“定义”出的线程
  - 这种线程的创建、调度和撤销完全由用户空间的线程库控制，而不涉及操作系统内核的直接支持
  - 用户级线程的切换也由用户/线程库定义
- 内核级线程为操作系统内核管理的线程
  - 内核级线程是由操作系统内核直接管理和支持的线程
  - 相较用户级线程更重，线程间切换由操作系统进行调度

多线程模型

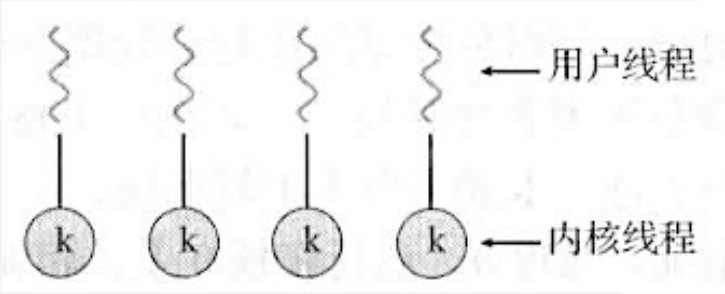
- 用户级线程与内核级线程的关系
  - 多对一模型
  - 一对一模型
  - 多对多模型
  - 双层模型

多对一模型

- 多对一模型映射多个用户级线程到一个内核线程
- 优点
  - 线程管理是由用户空间的线程库来完成的，因此效率更高
- 缺点
  - 如果一个线程执行阻塞系统调用，那么整个进程将会阻塞
  - 因为任一时间只有一个线程可以访问内核，所以多个线程不能并行运行在多处理核系统



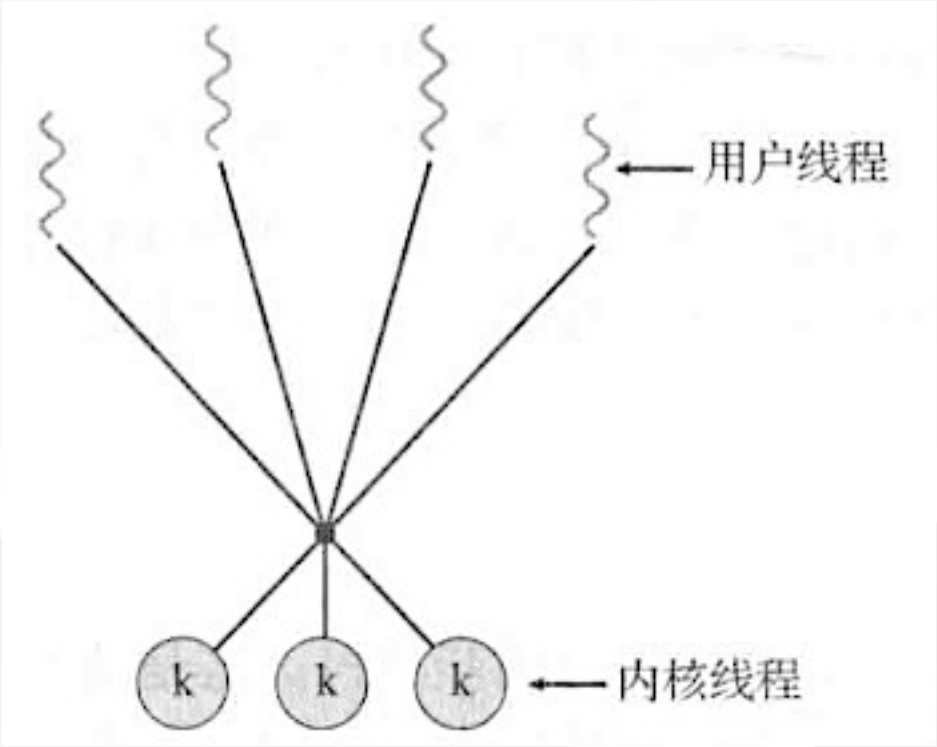
一对一模型



- 一对一模型映射每个用户线程到一个内核线程
- 优点
  - 该模型在一个线程执行阻塞系统调用时，能够允许另一个线程继续执行，所以它提供了比多对一模型更好的并发功能
- 缺点
  - 创建一个用户线程就要创建一个相应的内核线程，有一定的开销，所以这种模型的大多数实现限制了系统支持的线程数量
- 例子：Linux、Window XP/2000

多对多模型

- 多对多模型映射多个用户级线程到同样数量或更少数量的内核线程，其中，内核线程的数量可能与特定应用程序或特定机器有关
- 例子：Windows NT/2000

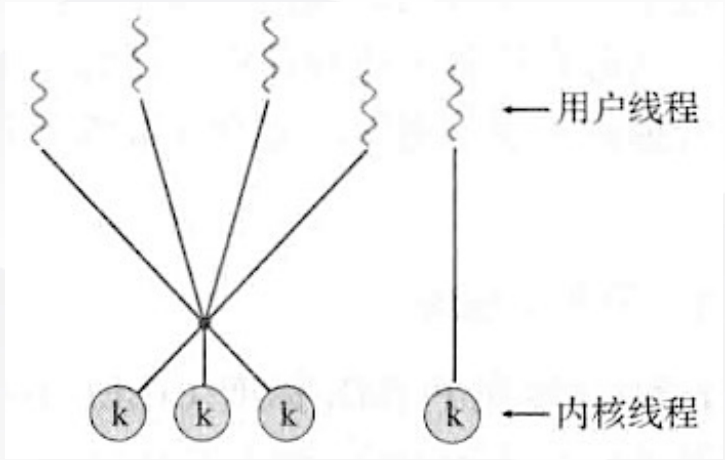


对并发性的影响

- 各模型对并发性的影响
  - 多对一模型允许开发人员创建任意多的用户线程，但是由于内核只能一次调度一个线程，所以并未增加并发性
  - 虽然一对一模型提供了更大的并发性，但是开发人员应小心，不要在应用程序内创建太多线程
  - 多对多模型没有这两个缺点:开发人员可以创建任意多的用户线程，并且相应内核线程能在多处理器系统上并发执行

双层模型

- 与多对多模型相似，同时可以允许一个用户线程与一个内核线程绑定
- 例子
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



线程库

- 线程库（thread library）为程序员提供创建和管理线程的API
- 两种实现线程库的方法
  - 用户级线程库
    - 针对用户级线程库的所有代码、数据结构都存在于用户空间
    - 调用库内的一个函数只是导致了用户空间内的一个本地函数的调用，而不是系统调用
  - 内核级线程库
    - 针对内核级线程库的所有代码、数据结构都存在于内核空间
    - 调用库中的一个API函数通常会导致对内核的系统调用
- 三种主要线程库：POSIX Pthreads, Win32 threads, Java threads

线程库

- Pthreads是POSIX标准定义的线程创建与同步API
  - 可提供用户级或内核级的库
  - 用于线程创建、同步
  - 广泛用于UNIX类型的操作系统
    - Solaris, Linux, Mac OS X
- Windows线程库用于Windows操作系统的内核级线程库
- JAVA 线程API允许线程在Java 程序中直接创建和管理

Pthreads实例 多线程

- 完成求和运算：

```c
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    pthread_mutex_init(&mutex_sum, NULL);

    // Initialize array with values
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i + 1;
    }

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, sum_array, (void *)&thread_ids[i]);
    }

    // Join threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Sum: %d\n", sum);

    pthread_mutex_destroy(&mutex_sum);

    return 0;
}
```

互斥锁

当调用pthread_join(thread, NULL)时，调用线程将被阻塞，直到指定的线程thread执行结束

Pthreads实例 多线程

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

#define NUM_THREADS 4
#define ARRAY_SIZE 10000

int array[ARRAY_SIZE];
int sum = 0;
pthread_mutex_t mutex_sum;

void *sum_array(void *arg) {
    int tid = *((int *)arg);
    int partial_sum = 0;

    for (int i = tid * (ARRAY_SIZE / NUM_THREADS); i < (tid + 1) * (ARRAY_SIZE / NUM_THREADS); i++) {
        partial_sum += array[i];
    }

    pthread_mutex_lock(&mutex_sum);
    sum += partial_sum;
    pthread_mutex_unlock(&mutex_sum);

    pthread_exit(NULL);
}
```

Windows线程

```
#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 parameter is required\n")
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      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);
   }
}
```

若要等待多个线程，可使用WaitForMultipleObject()

线程池

- 多线程程序存在潜在问题：无限制地创建线程将耗尽系统资源
- 使用线程池
  - 在进程开始时创建一定量的线程，并加到池中以等待工作
  - 当服务器收到请求时，唤醒池内的一个线程
  - 若没有可用线程，则阻塞
- 优点
  - 避免创建线程所带来的开销
  - 限制了可用线程的数量

OpenMP

- OpenMP为一组编译指令和API，用于编写C、C++、Fortran等语言的程序，支持共享内存环境下的并行编程
- OpenMP识别并行区域，即可并行运行的代码块：“隐式多线程”，无需直接管理线程的创建和同步

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

创建与系统处理核一样多的线程

OpenMP

- OpenMP提供各种指令，例如循环并行化
  - 累加，使用parallel for自动地将循环分配到多个线程并行执行
  - 支持多种操作系统：Linux、Windows、Mac OS

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
  c[i] = a[i] + b[i];
}
```

大中央调度

- 大中央调度（GCD）是苹果为Mac OS X和iOS操作系统提供的一种技术，允许将部分代码区域并行运行
- 增加了“块”的扩展，使用花括号将代码括起来，并在前面加上字符

```
^{ printf("I am a block"); }
```

- 通过将这些块放在调度队列上，GCD调度块以便执行
  - 当GCD从队列中移除一块后，该块将被分配给线程池中可用线程
  - 调度队列可以分为串行和并发两类
    - 串行：块按照先进先出的顺序删除
    - 并行：同时删除多个块

```
dispatch_queue_t queue = dispatch_get_global_queue
  (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

向队列提交一个块

Intel Threading Building Block

- Threading Building Block（TBB）是Intel商用的并行库，隐式使用多线程

```
for (int i = 0; i < n; i++) {
   apply(v[i]);
}
```

- 使用parallel_for并行化apply(v[i])

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```

多线程的问题-信号处理

- UNIX信号用于通知进程某个特定事件发生，接收可以是同步或异步的，遵循相同的模式：
  - 信号是由特定事件的发生而产生
  - 信号被传递到某个进程
  - 信号一旦收到就应处理
- 同步信号：非法访问内存、被0所除，需要发送到执行该操作而导致这个信号的进程
- 异步信号：运行程序以外的事件所产生，如使用特殊键（control+C）终止进程、定时器到期，该信号需要发送到另一进程

多线程的问题-信号处理

- 信号的传递规则
  - 传递信号到信号所适用的进程
  - 传递信号到进程内所有线程
  - 传递信号到进程内某些线程
  - 规定一个特定线程以接收进程的所有信号
- 传递取决于信号的类型
  - 如同步信号要传递到产生这一信号的线程，而非其他线程
  - 异步信号，如control+C，应传递到所有线程

```
kill(pid_t pid, int signal)
```

多线程的问题-线程撤销

- 线程撤销是在线程完成之前终止线程，如网页浏览器中按键停止加载
- 目标线程的撤销有两种情况：
  - 异步撤销：一个线程立即终止目标线程
    - 当线程已被分配了部分资源，或正在更新与某些线程共享的数据，异步撤销可能不会释放必要的资源，导致浪费
  - 延迟撤销：目标线程不断检查它是否应该被终止
    - 更安全，保证资源有序释放

多线程的问题-线程撤销

- 对于Pthreads，通过函数pthread_cancel()可以发起线程撤销

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

- Pthreads支持三种模式以撤销线程：

模式	状态	类型
关闭	禁用	-
延迟	启用	延迟
异步	启用	异步

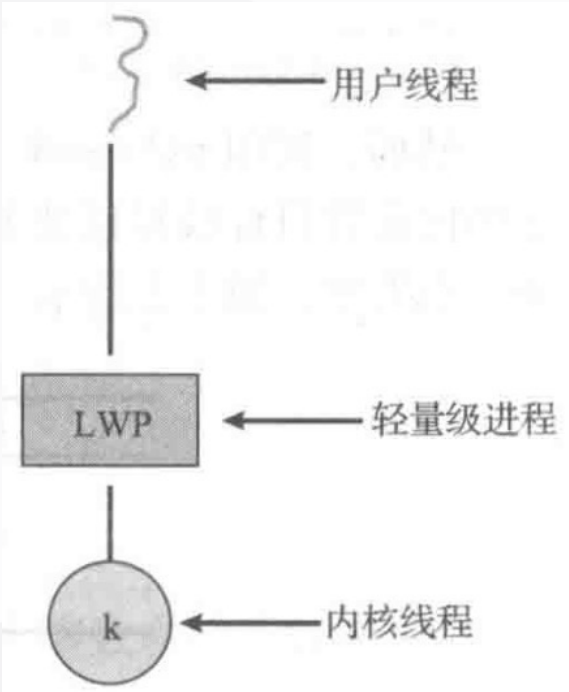
  - 禁用
    - 该线程当前无法撤销，指导启用撤销并响应该请求
  - 缺省撤销类型为延迟撤销，即当线程到达撤销点时，发生撤销

多线程的问题-线程本地存储

- 通常，多线程共享进程数据
- 但线程也需要它自己的某些数据，称为线程本地存储（TLS）
  - 例如存储线程的唯一标识符
- 注意TLS与局部变量的区别
  - 局部变量只在单个函数调用才可见；TLS在多个函数调用时都可见（由一个线程所调用）

多线程的问题-调度程序激活

- 在实现多对多、双层模型时，在用户和内核线程间增加一个中间数据结构，称为轻量级进程（LWP），实现内核级线程与用户级线程通信
- 每个LWP与一个内核线程相连，当内核线程阻塞，LWP也会阻塞，而LWP上的用户级线程也会阻塞
- 为了高效运行，LWP需要达到一定数量
  - 对于CPU密集型任务，LWP一个就够了
  - 对于I/O密集型任务，LWP需要多个，因为阻塞了一个内核线程，还可以通过调度其他内核级线程保证并发



课堂习题

1.设有一个应用，其60%为并行部分，而处理核数量分别为(a)2个和(b)4个。利用Amdahl定律，计算加速增益

2.有可能有并发但无并行吗?请解释