



---

第2章 进程与线程

## 2.9 线程



## 2.9 线程

---

- 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。



## 2.9.1 线程的概念

- 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



# 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

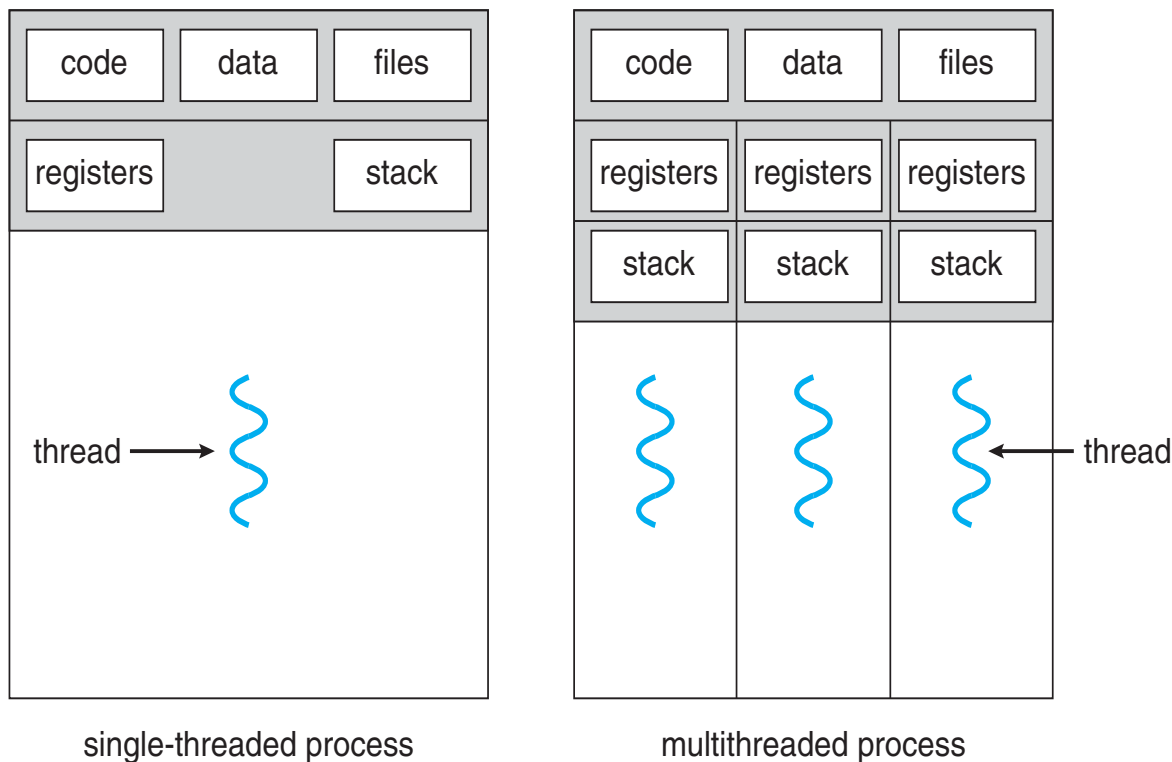
- 进程是拥有资源的基本单位，由一个或多个线程及相关资源构成。

## ■ 系统开销

- 进程创建、撤销及切换均涉及资源管理，开销大
- 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

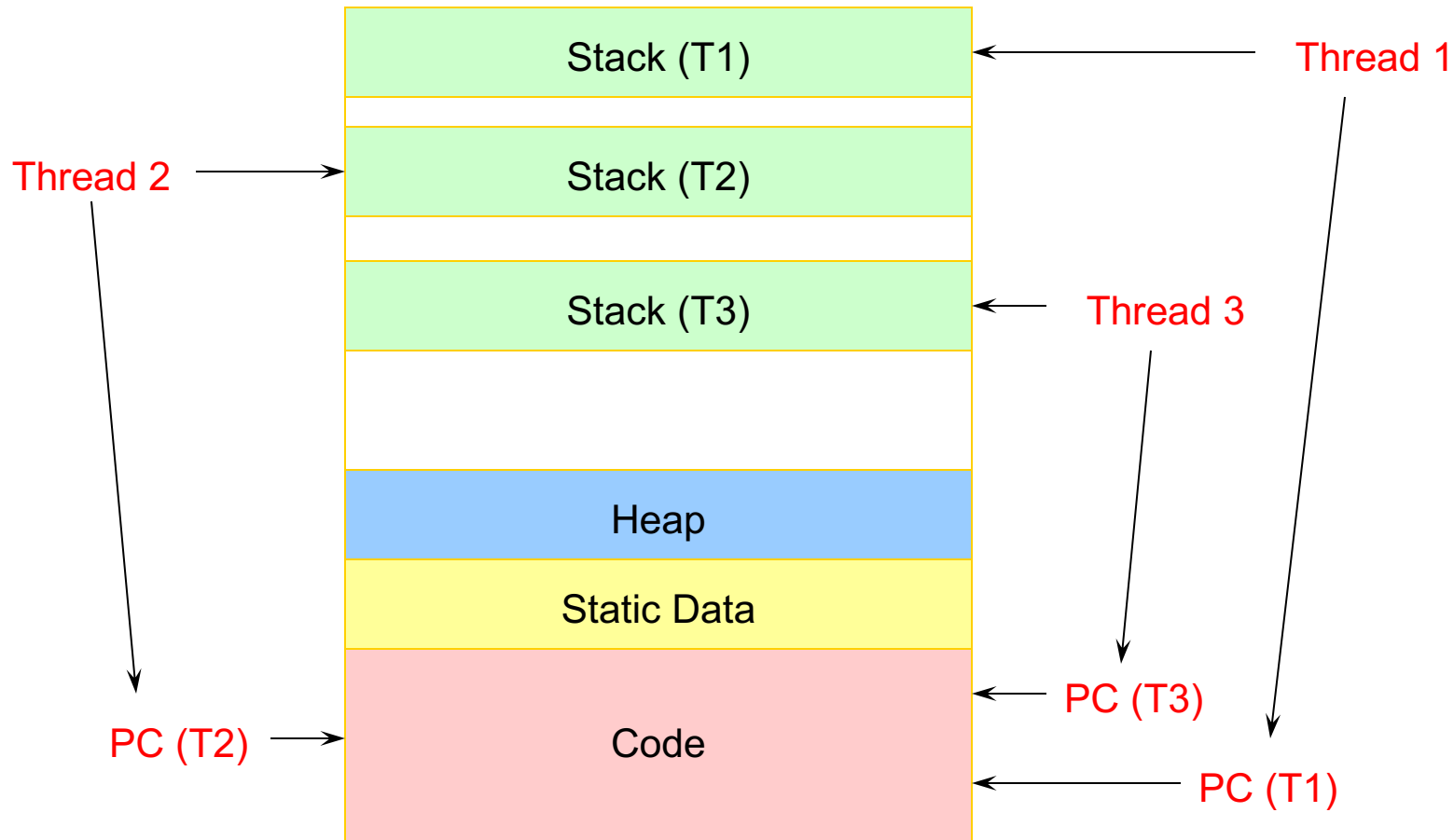
# 多线程

- **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





# 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

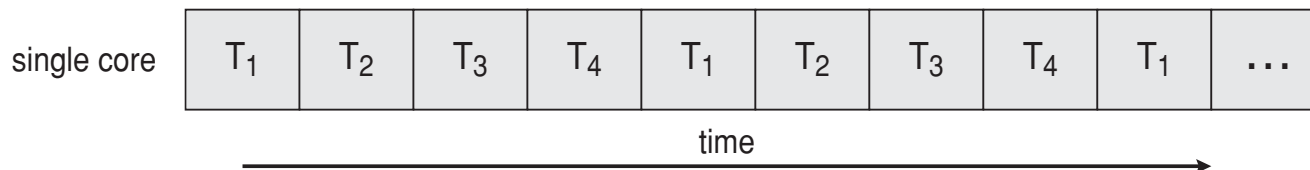
- 比进程创建更简单，上下文切换的负载小
- 如**Solaris**，创建：进程比线程慢**30**倍，切换：慢**5**倍

## ■ 可伸缩性

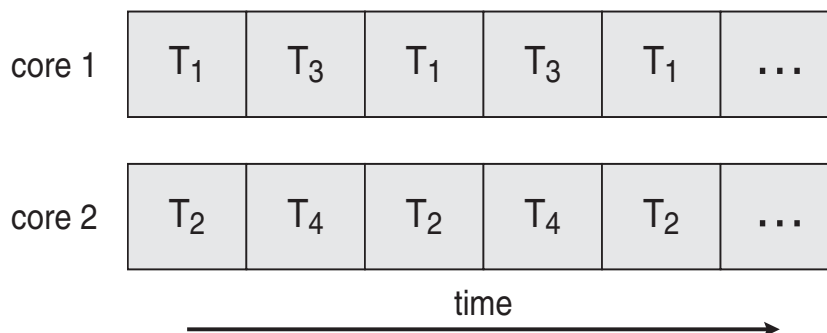
- 能够更好地利用多核体系结构优势，可以使得多线程能够并行在不同处理器核上运行。而单线程进程则只能运行在一个处理器上。

# 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:  $T_{new} = S \times T_{old} + (1 - S) \times \frac{T_{old}}{N}$
- 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



## 2.9.3 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()



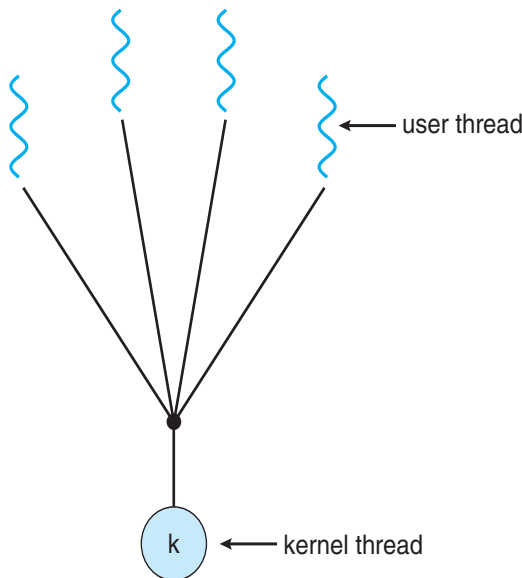
# 两种方法的组合

---

- 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。

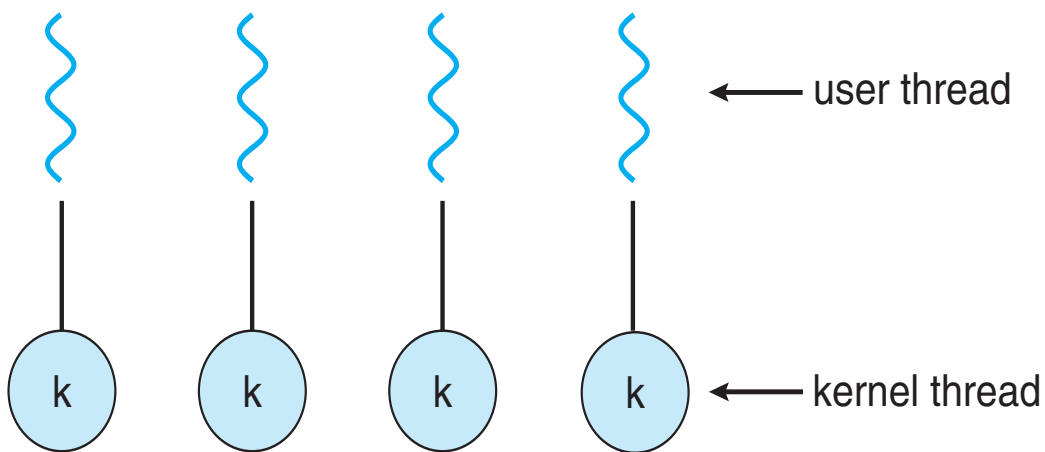
# 多对一模型

- 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



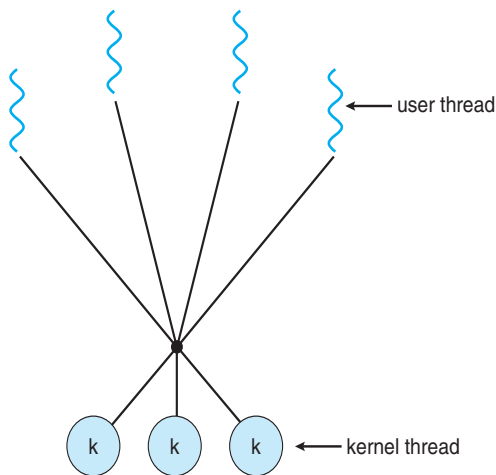
# 一对一模型

- 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





## 2.9.4 线程库

---

- 线程库（Thread library）提供了创建和管理多线程的编程接口
  - POSIX Pthreads
  - Windows threads
  - Java threads
- 线程库的实现方式
  - 纯用户态的实现
  - OS支持的内核级线程库
    - 对线程库中的API函数调用，会映射到特定的系统调用
    - 内核级线程库也能够响应部分中断





# 1. Pthreads库

---

- 可以提供用户级和内核级的线程库支持
- 遵从 POSIX 标准 (IEEE 1003.1c)的 API规定，是一种API规范，而非实现
- API定义了线程库的行为
- 常见于UNIX家族OS
  - Solaris, Linux, Mac OS X



# 常用Pthreads函数API

---

- 创建: `pthread_create`
- 退出: `pthread_exit`
- 等待: `pthread_join`
- 产生信号: `pthread_kill`
- 主动让出处理器: `pthread_yield`
- 异步撤销: `pthread_cancel`
- 延迟撤销: `pthread_testcancel`



# Pthreads 例程

```
#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;
    }

    #define NUM_THREADS 10

    /* an array of threads to be joined upon */
    pthread_t workers[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++)
        pthread_join(workers[i], NULL);
```

```
/* 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);
}

/* 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);
}
```

等待多个线程



# 非抢占调度

- 利用 `pthread_yield` 主动的让出 CPU

Ping Thread

```
while (1) {  
    printf("ping\n");  
    pthread_yield();  
}
```

Pong Thread

```
while (1) {  
    printf("pong\n");  
    pthread_yield();  
}
```

- 输出?

# pthread\_yield() 的模拟实现

```
pthread_yield() {  
    thread_t old_thread = current_thread;  
    current_thread = get_next_thread();  
    append_to_queue(ready_queue, old_thread);  
    context_switch(old_thread, current_thread);  
    return;  
}
```

As old thread

As new thread

■ append\_to\_queue()

■ context\_switch()



# 多线程的抢占调度

- 非抢占调度只有在当前运行的线程主动放弃时才会让出CPU
  - 长时间运行的线程会持续占用CPU
  - 只有`pthread_yield()`, `pthread_cancel()`, `pthread_exit()` 才引起上下文切换
- 抢占调度：被动的上下文切换（OS主动回收）
  - 异步的回收CPU控制权
  - 定时器
  - 线程库需要内核的支持，迫使当前线程主动调用`thread_yield`



## 2. 隐式多线程

---

- 将复杂的线程创建管理交给编译器和运行时库来完成
- 主要方法
  - 线程池
  - OpenMP
  - Grand Central Dispatch



# 线程池

---

- 创建一定数量的线程，并加到池中等待工作
- 优点
  - 使用已有的线程，可以减少创建新线程代价
  - 有效限制系统内的并发线程数量
  - 将执行任务从创建任务的机制中分离，允许采用不同策略来运行任务
- **Windows API支持线程池**





# OpenMP

---

- 一种广泛支持C、C++、FORTRAN的编译指令和API
- 使用并行编译指令定义可并行的区块

## #pragma omp parallel

```
#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;
}
```



# Grand Central Dispatch

---

- Mac OS X、iOS中采用的技术
- 支持C, C++, API, 运行时库的扩展
- 增加了一个形如“`^{ }`”的块定义
  - `^{ printf("I am a block"); }`
- 代码块会被放置到调度队列中
  - 当分配到合适线程时，就会从队列中被移除



## 2.9.6 操作系统实例分析

---

- Windows Threads
- Linux Thread

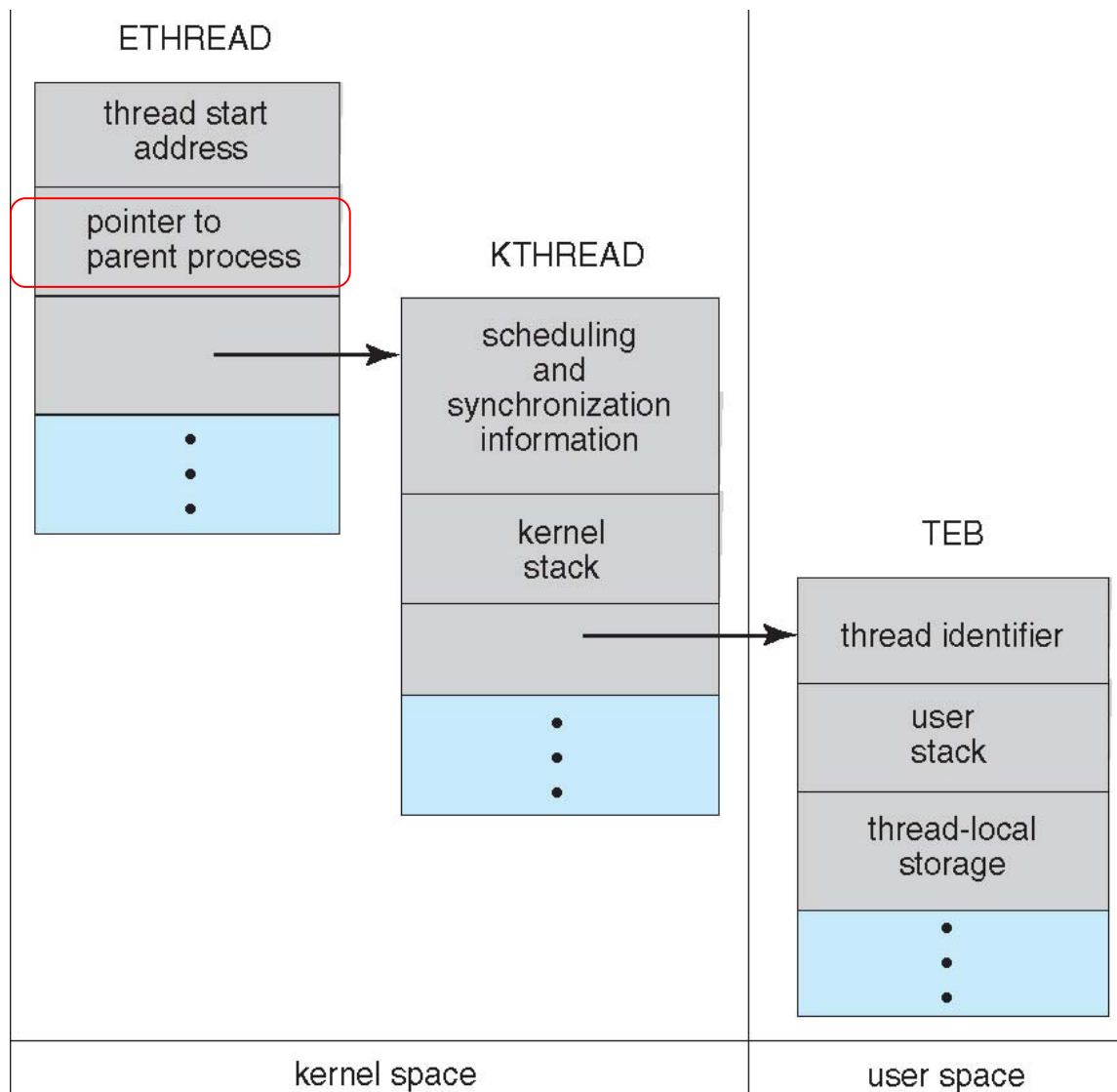


# Windows 线程

---

- 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
- 线程的主要数据结构：
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

# Windows 线程的数据结构





# 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
<code>CLONE_FS</code>	File-system information is shared.
<code>CLONE_VM</code>	The same memory space is shared.
<code>CLONE_SIGHAND</code>	Signal handlers are shared.
<code>CLONE_FILES</code>	The set of open files is shared.



# 作业-3

---

- 小作业-3

- V7版： 4.7

- V7版： 4.8

- 大作业-3

- V9版： 第4章， 编程项目任选1题