Chapter 4 线程

内容

1. 什么是线程？
2. 多线程模型
3. 线程库

1、概述

线程引入的原因
线程概念
线程和进程
线程结构
线程优点
Windows线程
Linux线程

引入原因

- 性能
  - 进程是重量级的，负载了程序运行的所有内容，进程操作开销大
  - Unix的轻型进程（fork）
- 应用
  - 进程代码有并行执行的需求
  - 例子：PPT编辑（输入/拼写检查/存盘）
- 硬件
  - 多核处理器已经是主流硬件
  - 加速进程的运行

线程

线程（轻型进程light weight process, LWP ）

- 可在CPU上运行的**基本执行单位**
- 进程内的一个代码片段可以被创建为一个线程
- 线程状态：就绪、运行、等待等
- 线程操作：创建、撤销、等待、唤醒
- 进程依旧是资源分配的基本单位
- 线程自己不拥有完整的系统资源，通过进程申请资源

- 传统进程：
  - 可称为重型进程(heavy weight process)
  - 等价于只有一个线程的任务，主线程
  - 单线程模型

- 操作系统引入线程后，资源分配的基本单位是（ ）
  - A. 线程
  - B. 进程
  - C. 超线程
  - D. 以上都不是

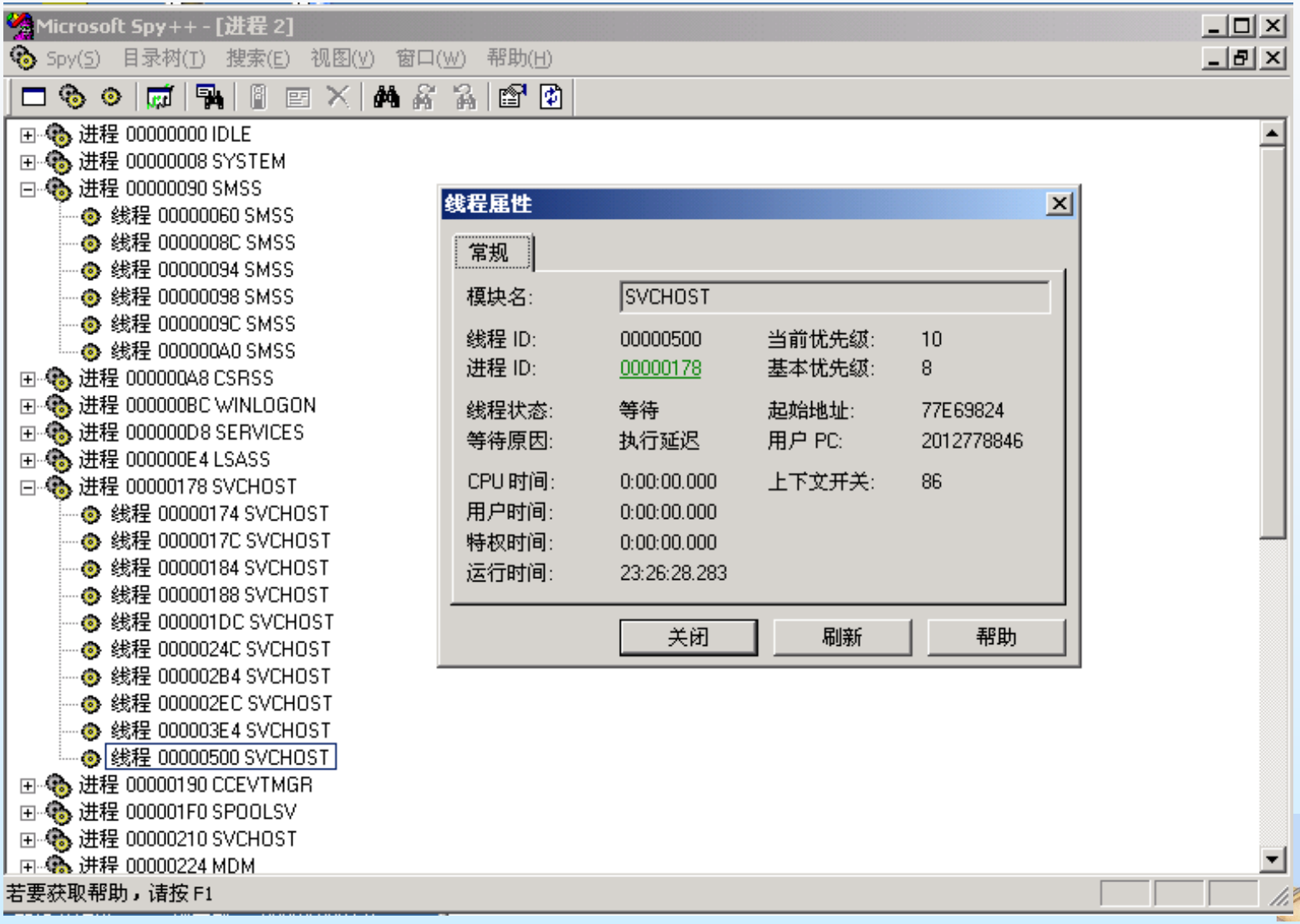
线程和进程

代码	资源	调度	切换	生命期
进程包含线程	进程是资源分配基本单位	同一进程中的线程切换不会引起进程切换	进程：重量级，上下文切换代价大	进程撤销会导致它的所有线程被撤销
线程是进程中的一段代码	线程不拥有资源，共享进程的资源	线程是基本调度单位	线程：轻量级，代价小	线程撤销不会影响进程

线程结构

- 代码和数据：来自进程
- 各类资源：来自进程
- 线程控制块（Thread Control Block,TCB）
  - 线程ID
  - 线程计数器PC
  - 寄存器集
  - 栈空间

Windows 2000 Process and Thread
Windows 2000进程和线程



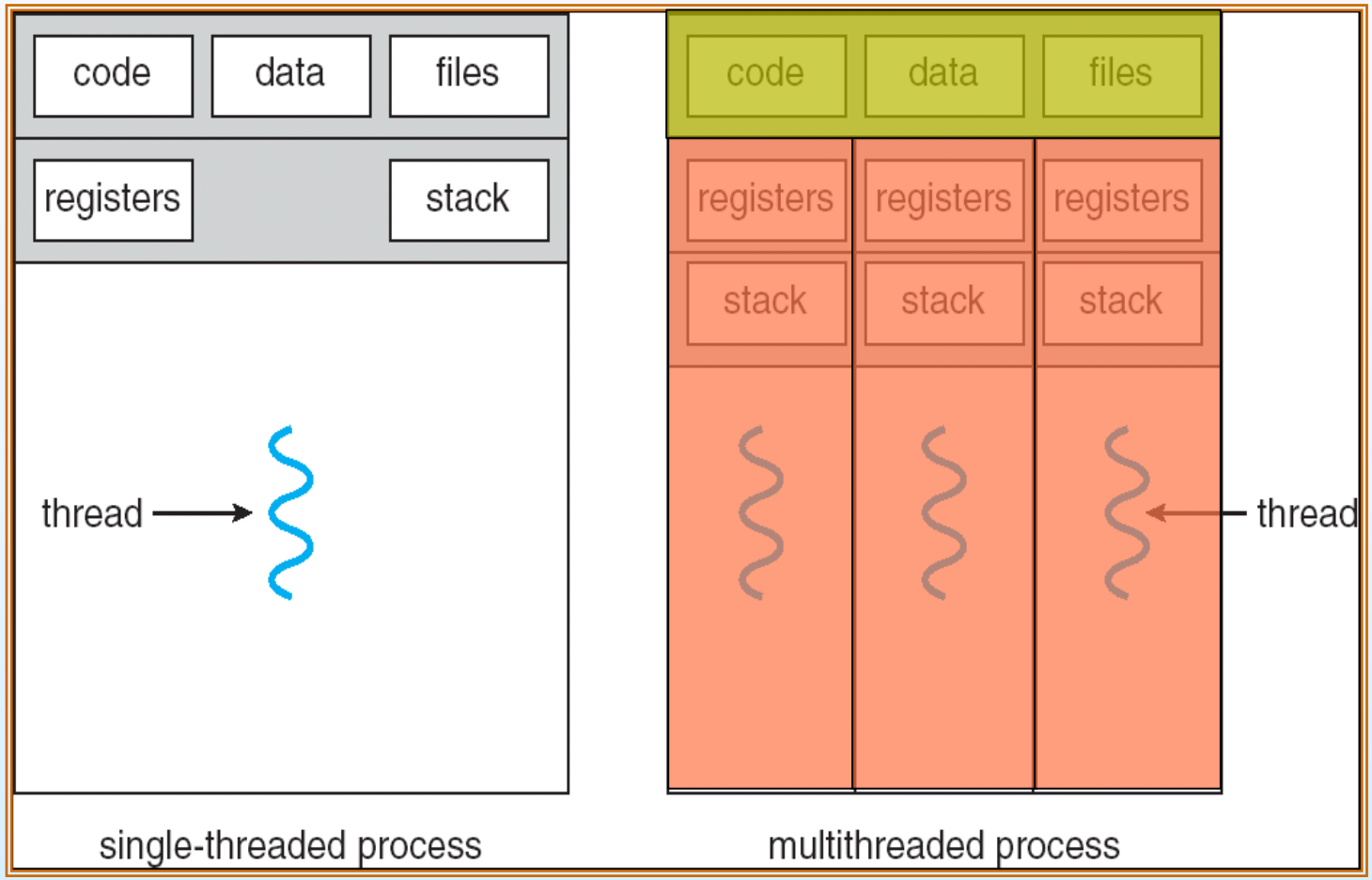
单个线程和多线程进程

单线程

- 一个进程只有一个线程
- 早期操作系统

多线程

- 一个进程有多个线程
- 支持线程的操作系统



线程优点

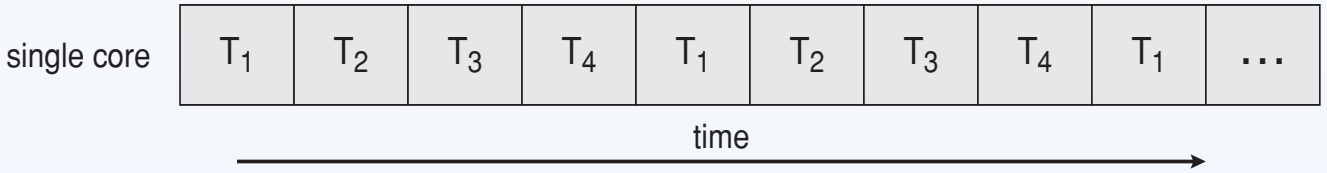
- 响应度高
  - 线程创建开销小
  - 例子：Web服务器
- 资源共享
  - 进程中的线程可以共享进程资源
- 经济性
  - 线程创建、上下文切换比进程快
  - Solaris：创建线程比进程快30倍，线程切换比进程切换快5倍
- MP体系结构的运用
  - 一个进程中的线程在不同处理器上并行运行，缩短了运行时间

Web服务器

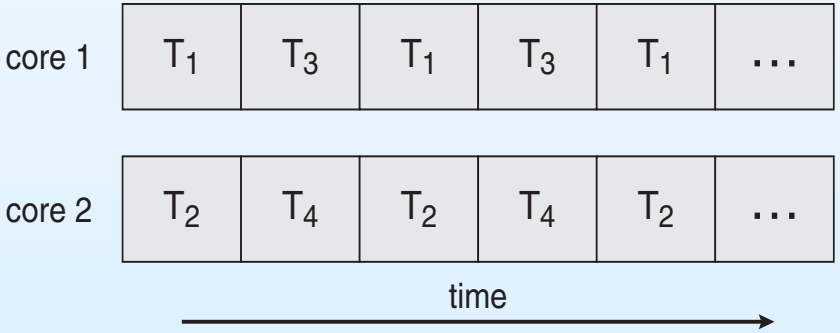
<html>….<img src=1.jpg><img src=2.jpg>…</html>

并发（Concurrency）和并行（Parallelism）

- 单核系统并发:

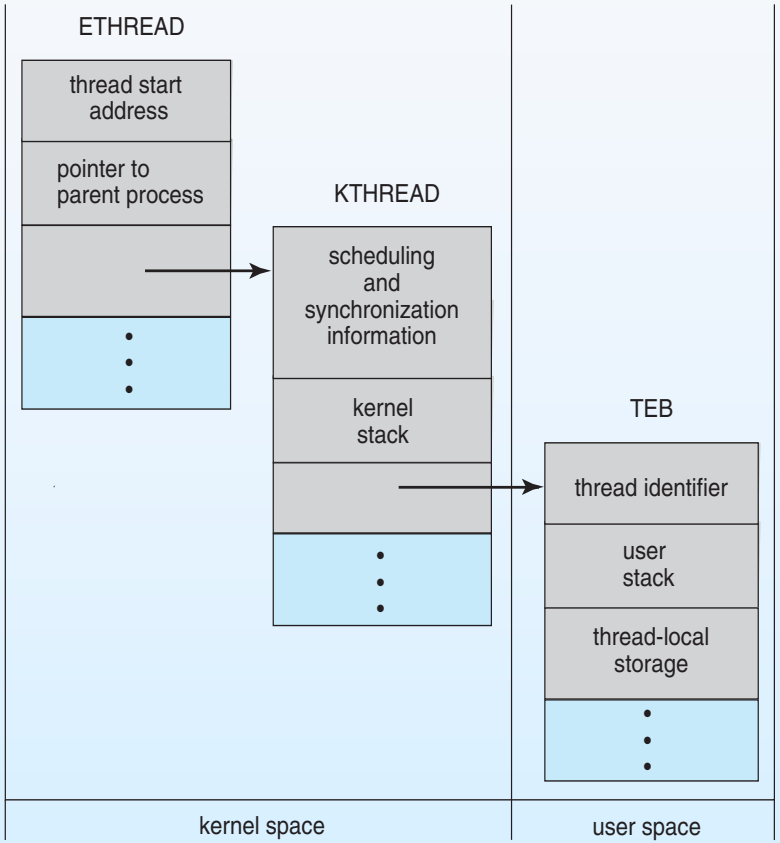


- 多核系统并行:



Windows 线程

- 操作系统支持线程
- 线程主要数据结构:
  - 执行线程块ETHREAD
    - Executive thread block
    - 对应程序地址
    - 指向KTHREAD指针
    - 内核空间
  - 核心线程块KTHREAD
    - Kernel thread block
    - 线程调度和同步信息
    - 内核空间
  - 线程环境块TEB
    - Thread environment block
    - 用户空间的数据结构，线程本地存储



Linux 线程

- Linux 2.2版引入线程
- 通过 **clone()** 系统调用创建线程
  - 类似fork
  - Linux只支持克隆线程
- 用术语“任务”表示进程和线程，一般不用“线程”
- 用户线程：PThreads

Clone

- 创建进程或者线程
- Clone可有选择性继承父进程的资源
  - 不和父进程共享

int clone(int (*fn)(void *), void *child_stack, int flags, void *arg)

  - fn为函数指针，此指针指向一个函数体，即想要创建进程的静态程序；
  - child_stack为给子进程分配系统堆栈的指针
  - flags为要复制资源的标志，描述你需要从父进程继承那些资源

2、多线程模型

用户线程
内核线程
多对一模型
一对一模型
多对多模型

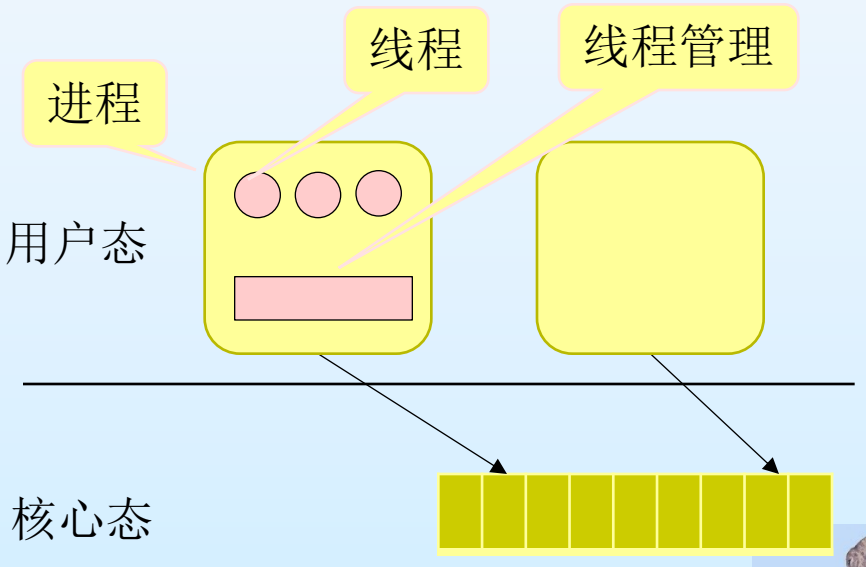
用户（管理）线程

- 用户线程：由用户线程库进行管理的线程
  - 内核看不到用户线程
  - 用户线程的创建和调度在用户空间中，不需要内核的干预
  - 应用于传统的只支持进程的操作系统
- 例子

  -POSIX *Pthreads*

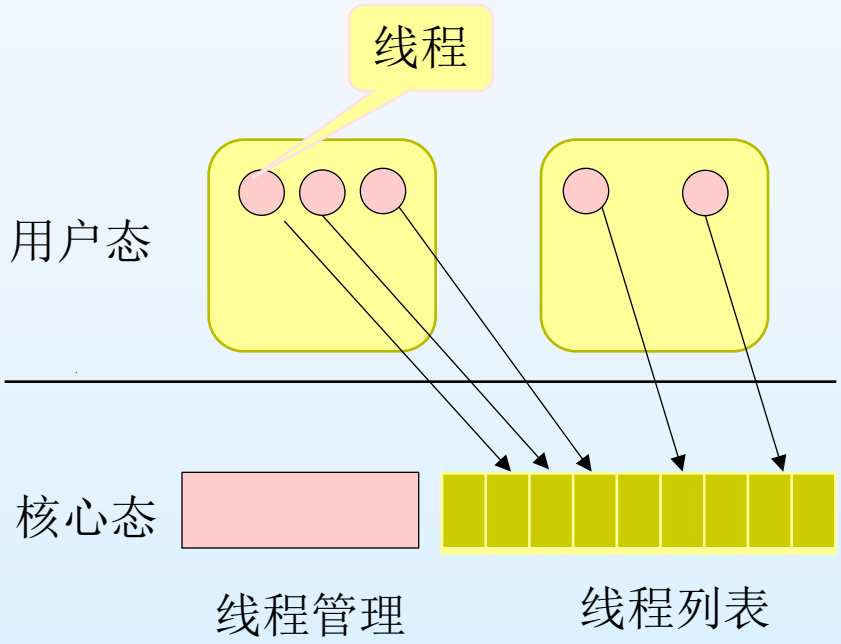
  - Win32 *threads*

  - Java *threads*



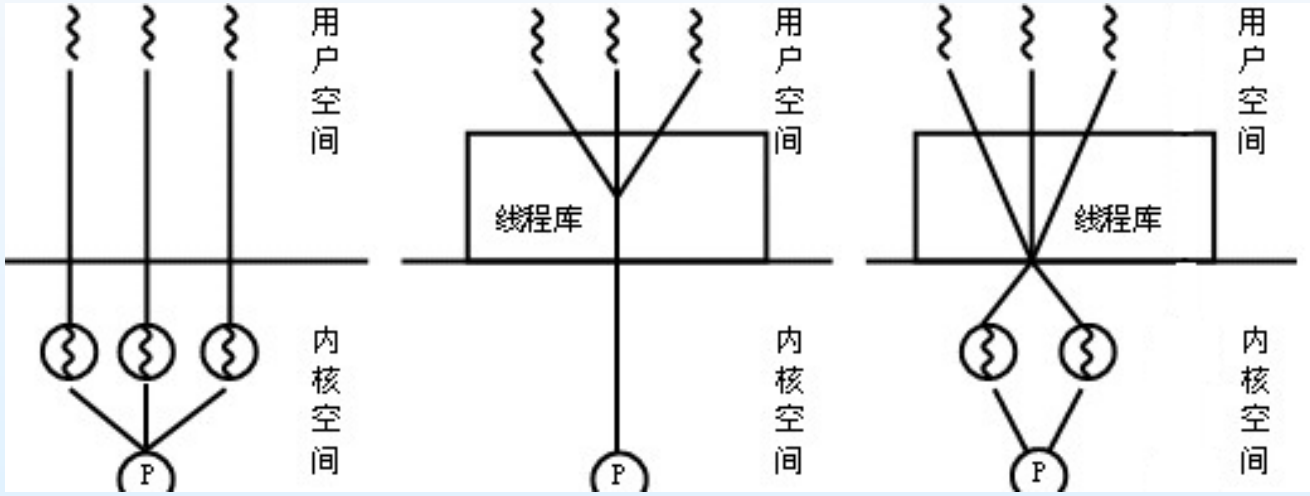
内核（管理）线程

- 内核线程：内核进行管理的线程
  - 需要内核支持
  - 由内核完成线程调度
  - 由内核进行创建和撤销
- 例子
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X



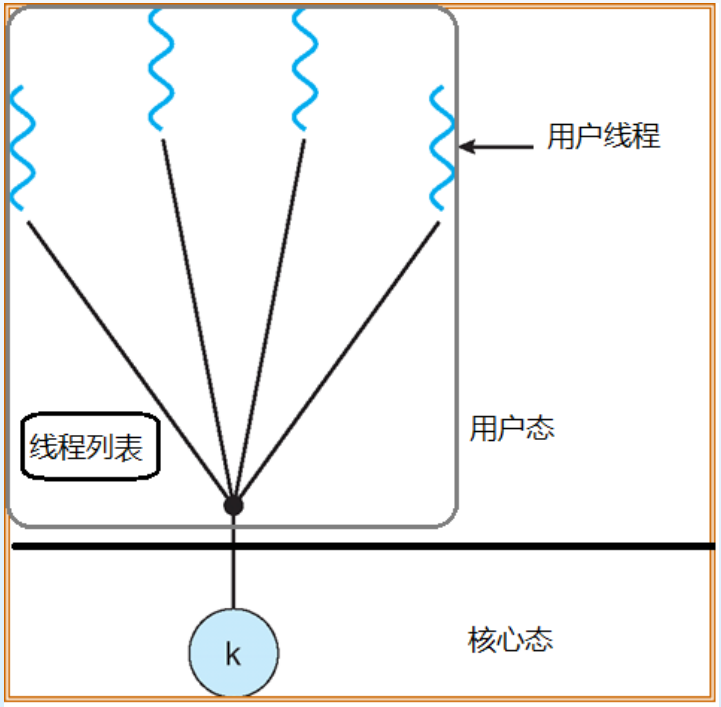
多线程模型

- 多对一模型
- 一对一模型
- 多对多模型



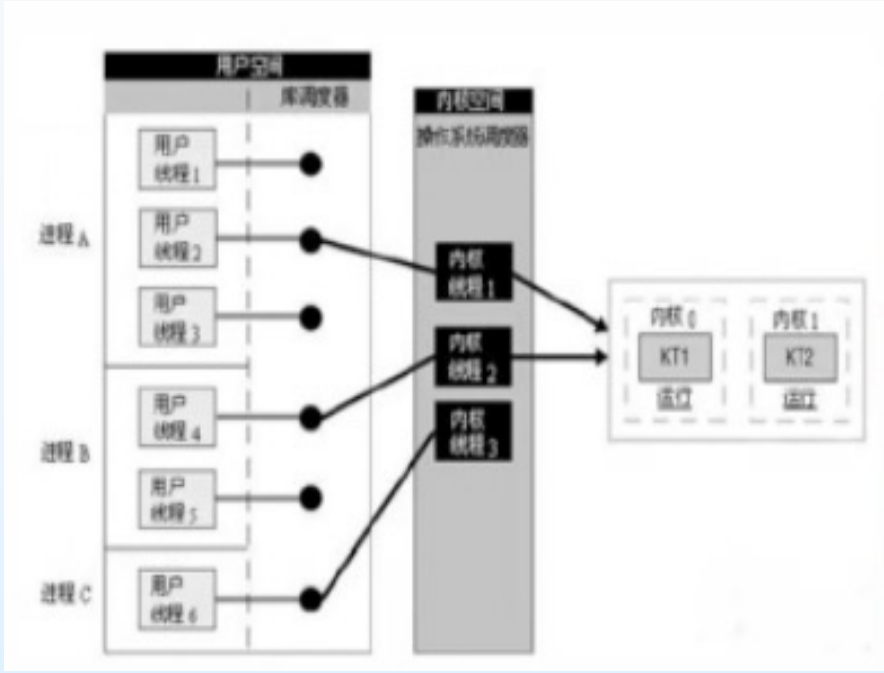
多对一模型

- 不支持内核线程的操作系统
  - 内核只有进程
- 内核只看到一个进程
  - 多个线程不能并行运行在多个处理器上（缺点）
- 进程中的用户线程由进程自己管理
  - 进程内线程切换不会导致进程切换（优点）
  - 一个线程的系统调用会导致整个进程阻塞
- 例子:
  - Solaris Green Threads
  - GNU Portable Threads



例子

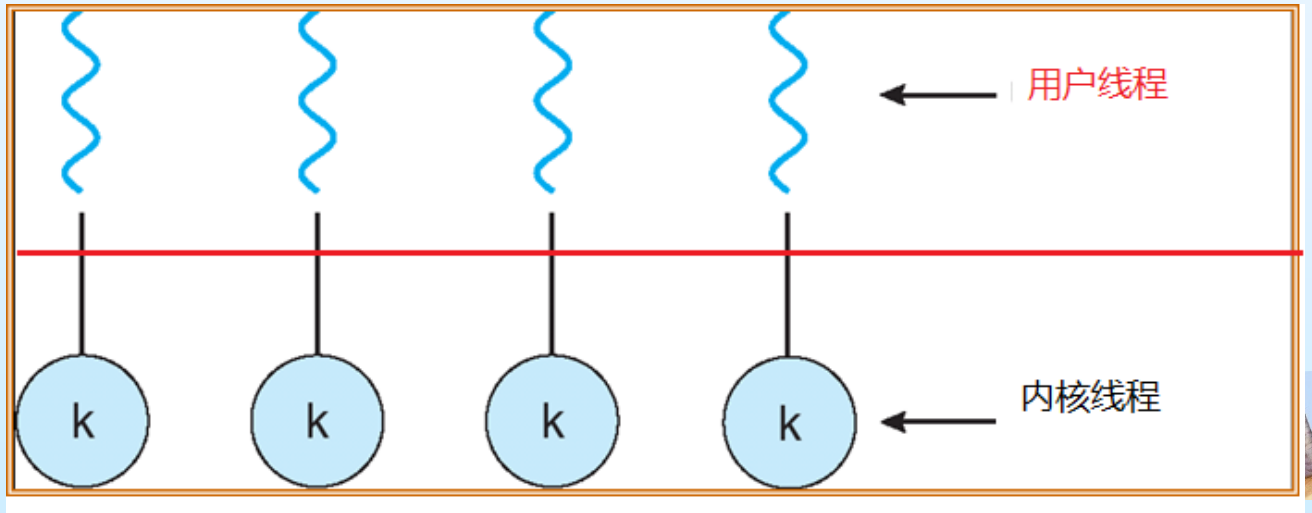
- 优点
  - 可运行在不支持线程的操作系统中
  - 线程管理开销小
  - 允许进程定制调度算法，线程管理灵活
  - 线程调度不需要内核参与，控制简单
- 缺点
  - 同一进程中只能同时有一个线程运行，并行性差
  - 线程发生系统调用而阻塞时，会阻塞整个进程
  - 多处理器下，同一个进程的线程只能在1个处理器上运行



- 判断：多对一模型中内核只能看到一个进程，看不到进程中的线程。

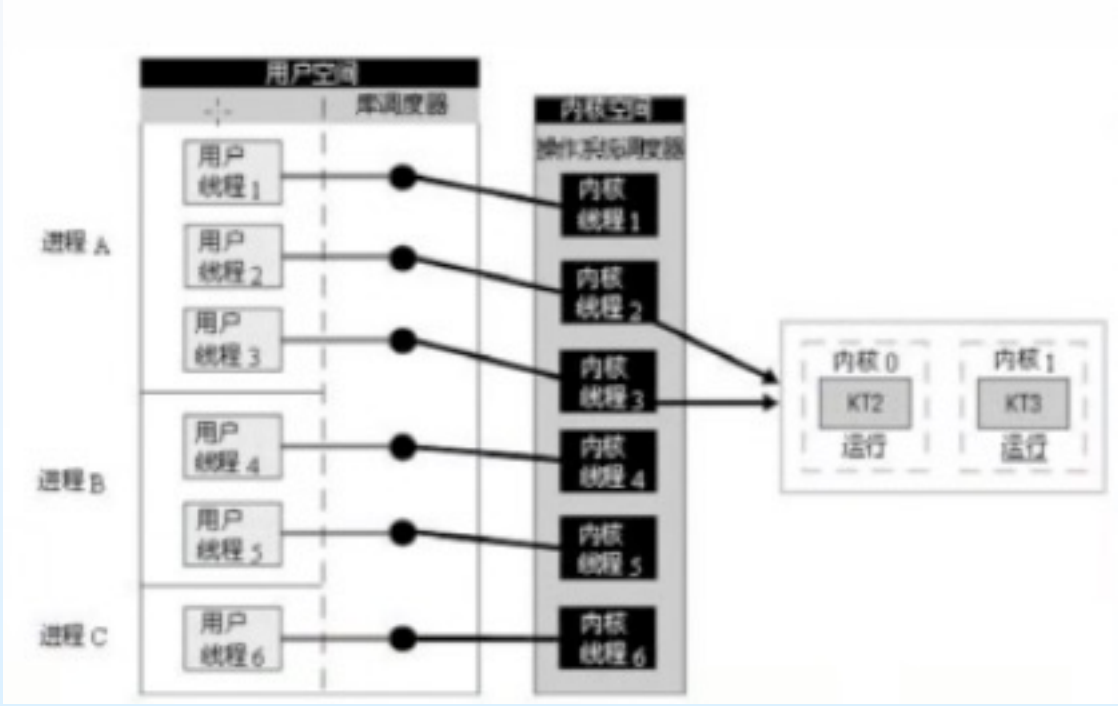
一对一模型

- 用于支持线程的操作系统
  - 用户线程映射到内核线程
  - 操作系统管理这些线程
- 并发性好：多个线程可并行运行在多个处理器上
- 内核开销大
- 例子
  - Windows
  - OS/2



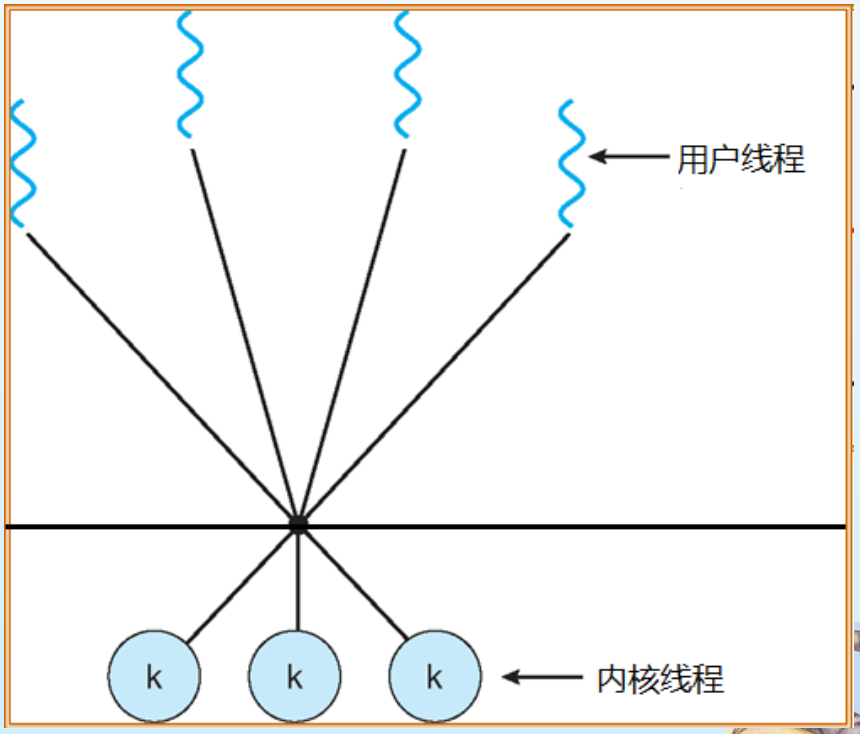
例子

- 优点
  - 多处理器系统中，同一进程中的线程能够并行在多个处理器上运行
  - 进程中一个线程被阻塞，能切换同一进程内的其它线程继续执行
- 缺点
  - 线程管理开销大
  - 线程调度可能会引起进程调度
  - 需要操作系统支持

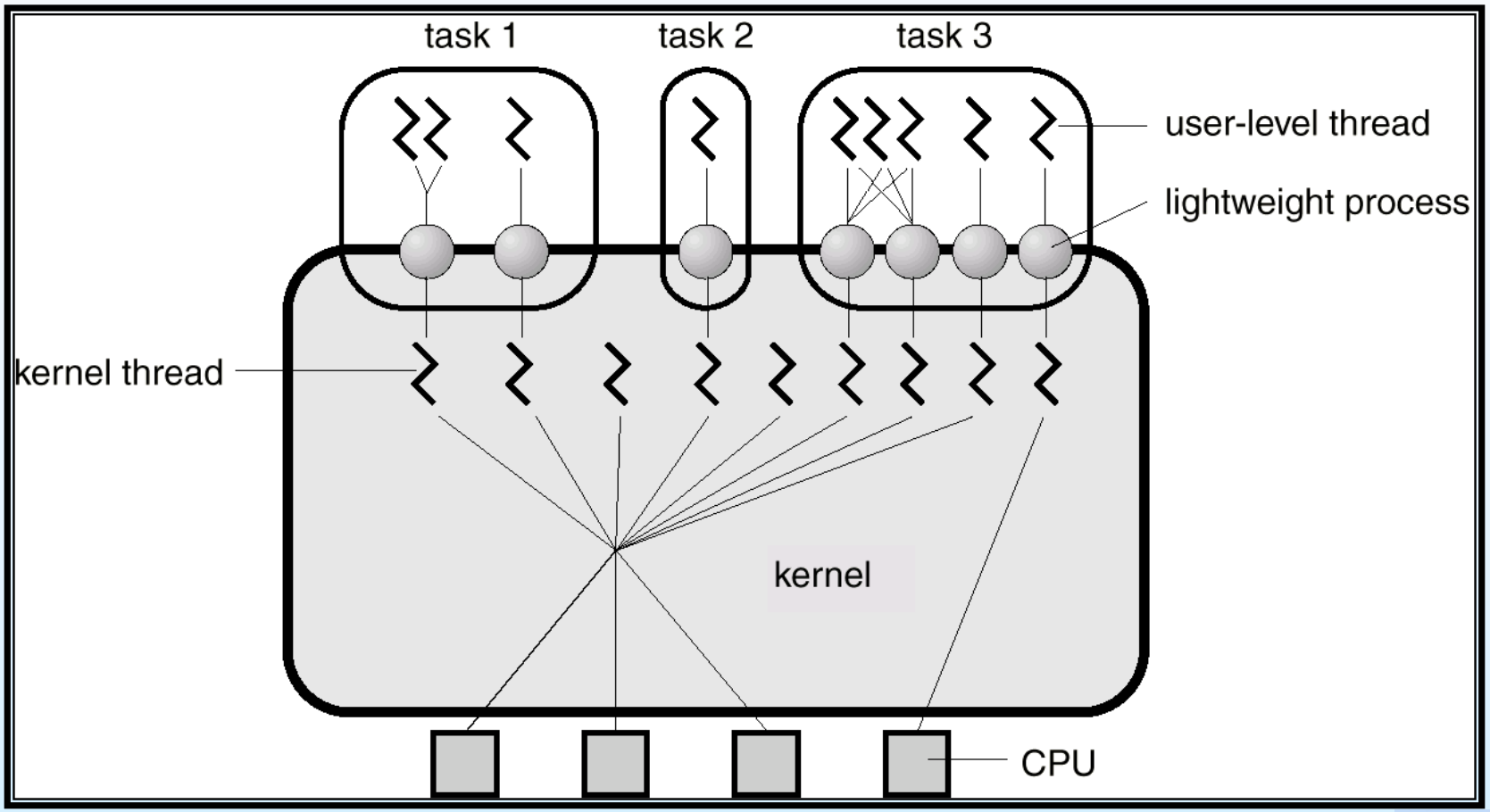


多对多模型

- 多个用户线程映射为相等或更小数目的内核线程
  - 并发性和效率兼顾
  - 增加复杂度
- 例子
  - Solaris 9 以前的版本

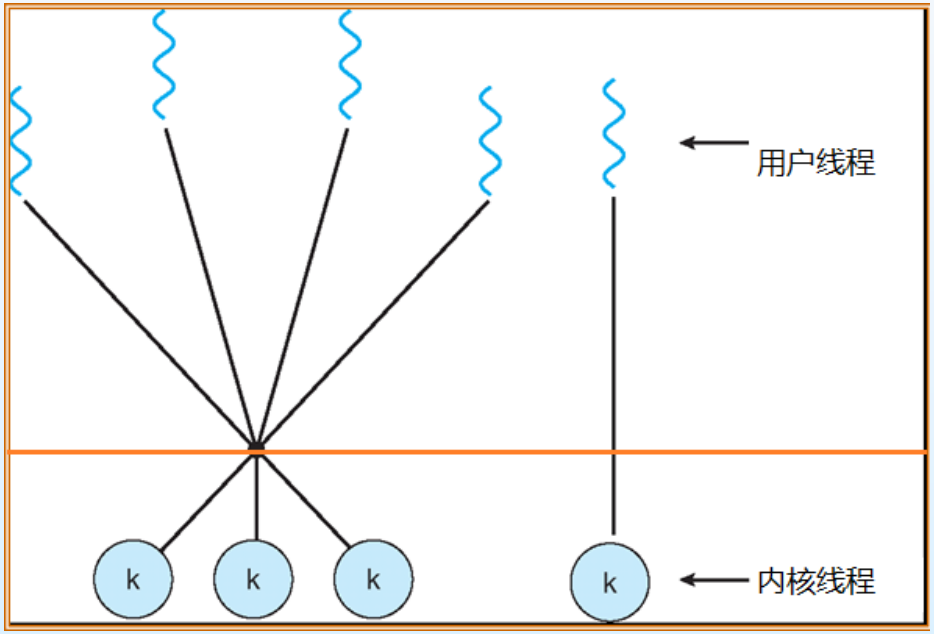


Solaris 2多对多模型



两级模型

- 类似于 M:M, 只不过它允许一个用户线程绑定到内核线程
- 例子
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



3、线程库

线程库的概念

Pthreads 线程库

JAVA线程库

Win32线程库

线程库

- 为程序员提供了创建和管理线程的API
- 两种模式
  - 用户库（用户线程）
    - 存在于用户空间
    - 不需要内核支持
    - 调用线程库不会产生系统调用
  - 内核库（内核线程）
    - 存在于内核
    - 操作系统支持
    - 调用线程库会产生系统调用

线程库

- 常用线程库
  - Windows 线程库： 内核级
  - Pthreads 线程库：用户级
  - JAVA 线程库： 用户级

Win32线程库

- Windows操作系统支持线程技术，所以Win32线程库实际上包含在Win32 API中。
  - Win32线程库是内核库
  - 内核线程
  - 线程创建方法
    - Win 32 API
    - MFC
    - .Net Framework

Windows 线程

- 1对1映射
- 每个线程包括：
  - 线程 id
  - 寄存器集合
  - 堆栈
  - 私有数据
- 线程主要的数据结构:
  - ETHREAD (executive thread block)：执行线程块
  - KTHREAD (kernel thread block)：核心线程块
  - TEB (thread environment block)：线程环境块

Win32线程库

- 线程创建

  HANDLE **CreateThread**(LPSECURITY_ATTRIBUTES lpThreadAttributes, DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId);

  线程函数原型：**DWORD WINAPI threadfunc(LPVOID param);**

- 线程挂起

  DWORD **SuspendThread**(HANDLE hThread);

- 线程恢复

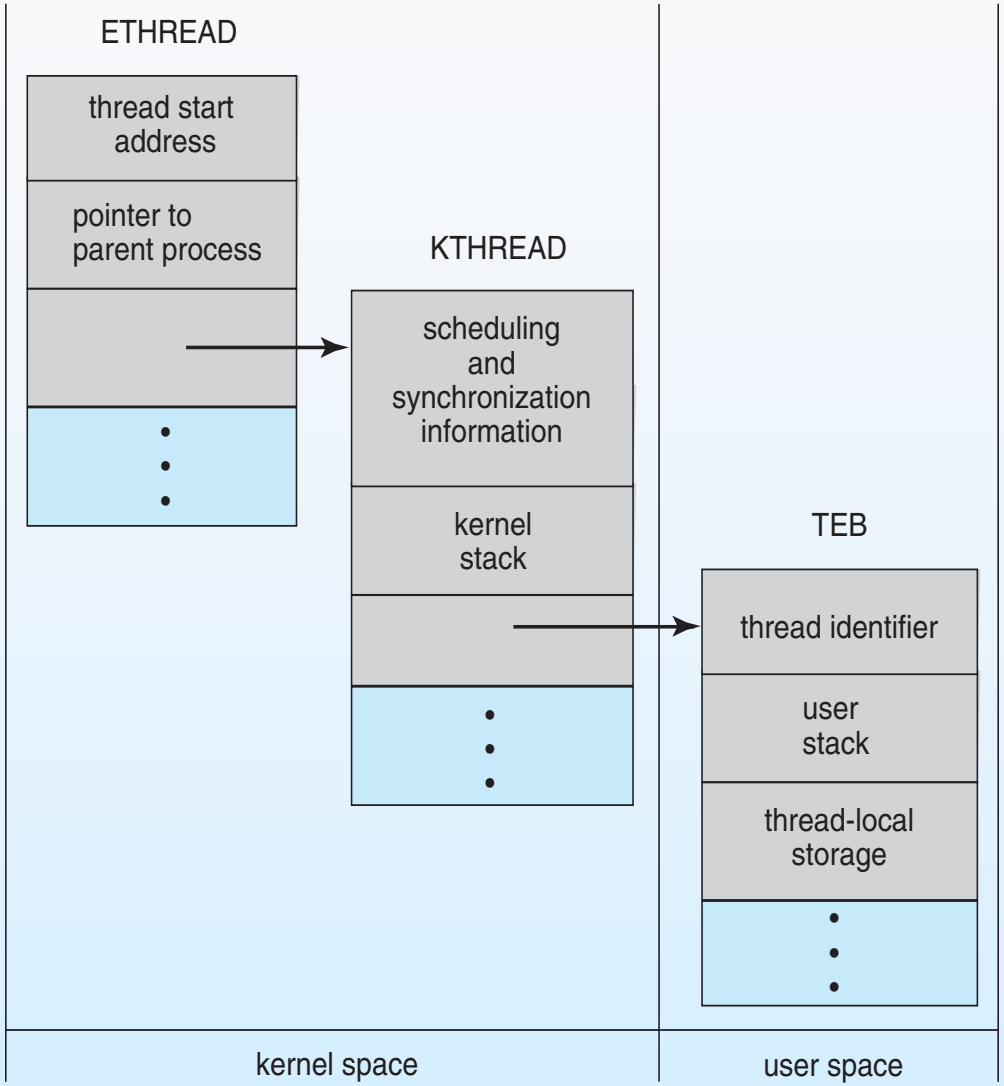
  DWORD **ResumeThread**(HANDLE hThread);

- 线程退出

  VOID **ExitThread**(DWORD dwExitCode);

  BOOL **TerminateThread**(HANDLE hThread,DWORD dwExitCode);

Windows 线程结构



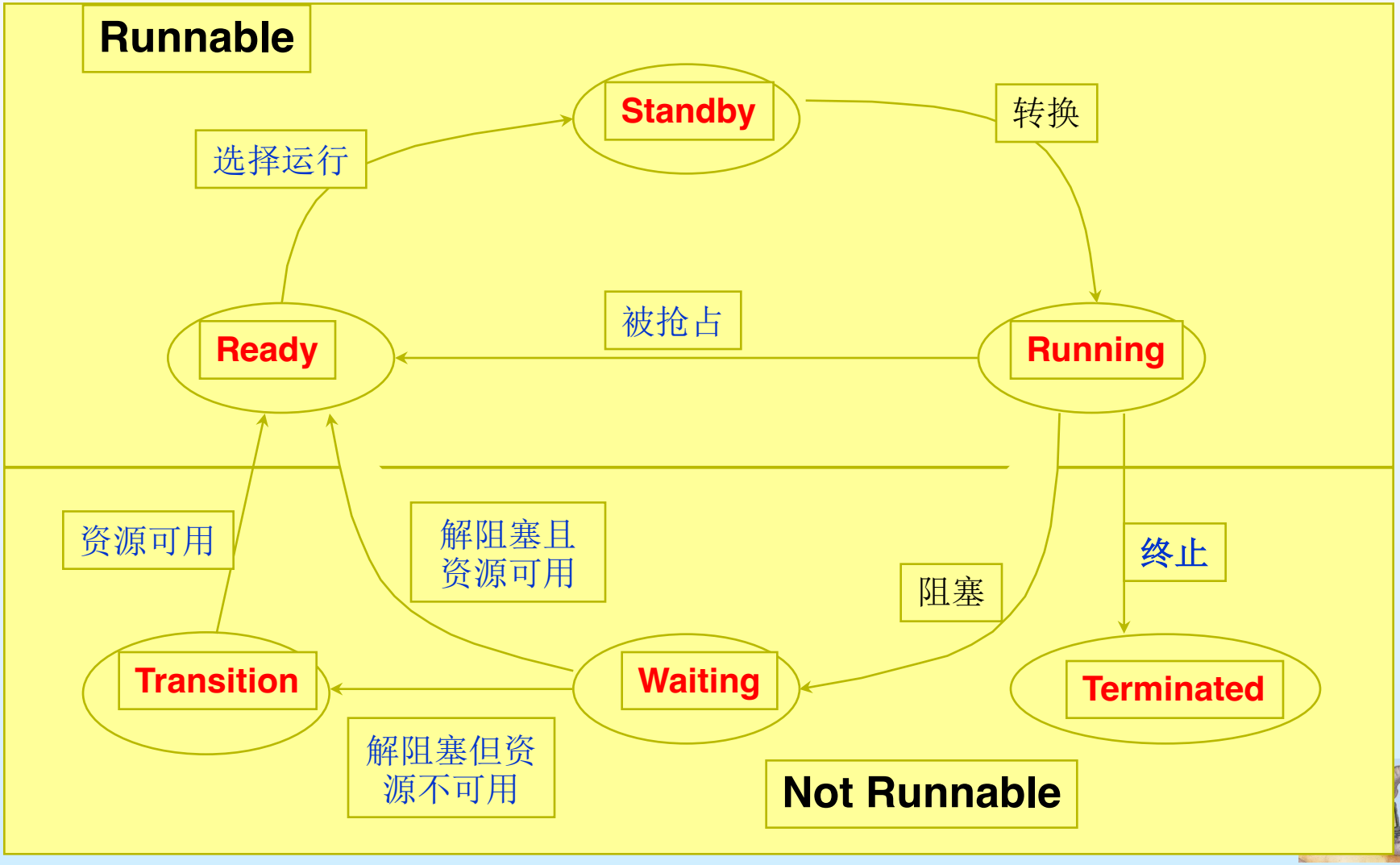
例子

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
using namespace std;
DWORD WINAPI FunOne(LPVOID param){
  while(true)  {
    Sleep(1000);
    cout<<"hello! ";
  }
  return 0;
}
DWORD WINAPI FunTwo(LPVOID param){
  while(true)  {
    Sleep(1000);
    cout<<"world! ";
  }
  return 0;
}
```

例子

```
int main(int argc, char* argv[]) {
        int input=0;
        HANDLE hand1=CreateThread (NULL, 0, FunOne, (void*)&input,
             CREATE_SUSPENDED, NULL);
        HANDLE hand2=CreateThread (NULL, 0, FunTwo, (void*)&input,
             CREATE_SUSPENDED, NULL);
        while(true){
          cin>>input;
          if(input==1)  {
                        ResumeThread(hand1);
                        ResumeThread(hand2);
          }
          else     {
                        SuspendThread(hand1);
                        SuspendThread(hand2);
          }
         }
        TerminateThread(hand1,1);
        TerminateThread(hand2,1);
        return 0;
}
```

Windows 线程状态



Pthreads线程库

- POSIX 标准：可移植操作系统接口（Portable Operating System Interface）
  - IEEE 定义了操作系统为应用程序提供的接口标准
  - 为各种Unix软件定义的一系列API标准总称为POSIX
- Pthreads：POSIX线程（POSIX threads）
  - 线程的POSIX 标准
  - 定义了创建和管理线程的一整套API
  - 常用于UNIX类操作系统 (Solaris, Linux, Mac OS X)
  - Windows也有移植版pthreads-win32
  - 一般为用户线程

POSIX 线程库Pthreads

- 使用fork()创建进程
  - 代价昂贵
  - 进程间通信方式复杂
  - 操作系统在实现进程间的切换比线程切换更费事
- 使用pthreads库创建线程
  - 创建线程比创建进程更快
  - 线程间的通信方式更容易
  - 操作系统对线程的切换比对进程的切换更容易和快速

常用线程操作

- pthread_create()：创建一个线程
- pthread_exit()：终止当前线程
- pthread_cancel()：中断另外一个线程的运行
- pthread_join()：阻塞当前的线程，直到另外一个线程运行结束
- pthread_attr_init()：初始化线程的属性

- pthread_t：线程ID
- pthread_attr_t：线程属性

线程的创建

```
#include <pthread.h>
   int  pthread_create(pthread_t  *  thread,
                       pthread_attr_t * attr,
                       void *(*start_routine)(void *),
                       void * arg);
```

- 第一个参数为指向线程标识符的指针
- 第二个参数用来设置线程属性
- 第三个参数是线程运行函数的起始地址
- 最后一个参数是运行函数的参数

一个简单例子

```
#include<stdio.h>
#include<pthread.h>
#include<string.h>
#include<sys/types.h>
#include<unistd.h>

pthread_t ntid;


void *thr_fn(void *arg){
  printf("new thread:%u\n", ntid);
  return ((void *)0);
}

int main(){
  printf("main: %u\n", pthread_self());
  int err;

  err = pthread_create(&ntid,NULL,thr_fn,NULL);
  if(err != 0){
   printf("can't create thread: %s\n",strerror(err));
   return 1;
  }

  sleep(1);
  return 0;
}
```

- gcc -o mypthread -lpthread mypthread.c

线程结束

- 调用pthread_exit()结束线程执行
  - void pthread_exit(void *retval);
- 使用 pthread_cancel() 函数终止其他线程的执行
  - int pthread_cancel(pthread_t thread);
  - 向线程t发送取消请求，默认情况下线程thread自己调用pthread_exit(PTHREAD_CANCELED)，

线程等待

- pthread_join() 函数等待被创建的线程结束
- pthread_join() 函数会挂起创建线程的线程的执行，直到等待到想要等待的子线程
- 函数原型：

```
int pthread_join(pthread_t th, void **thread_return);
```

例子

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

#define THREAD_NUMBER 2
int retval_hello1= 2, retval_hello2 = 3;

void* hello1(void *arg)
{
   char *hello_str = (char *)arg;
   sleep(1);
   printf("%s\n", hello_str);
   pthread_exit(&retval_hello1);
}
```

```
void* hello2(void *arg)
{
   char *hello_str = (char *)arg;
   sleep(2);
   printf("%s\n", hello_str);
   pthread_exit(&retval_hello2);
}
```

例子

```
int main(int argc, char *argv[])
{
   int i;     int ret_val, ret_val2;     int *retval_hello[2];

   pthread_t pt[THREAD_NUMBER];
   const char *arg[THREAD_NUMBER];
   arg[0] = "hello world from thread1";
   arg[1] = "hello world from thread2";

  printf("Begin to create threads...\n");
   ret_val1 = pthread_create(&pt[0], NULL, hello1, (void *)arg[0]);
   ret_val2 = pthread_create(&pt[1], NULL, hello2, (void *)arg[1]);
```

例子

```
    printf("Begin to wait for threads...\n");
    for(i = 0; i < THREAD_NUMBER; i++) {
        ret_val = pthread_join(pt[i], (void **)&retval_hello[i]);
        if (ret_val != 0) {
           printf("pthread_join error!\n");
         exit(1);
        } else {
           printf("return value is %d\n", *retval_hello[i]);
        }
    }

    printf("Now, the main thread returns.\n");
    return 0;
}
```

Java 线程库

- Java 线程由JAVA虚拟机JVM管理
  - 用户线程
  - JAVA线程操作系统不可见
  - 定义了创建和操纵线程的一整套API
  - 跨操作系统平台
- Java 线程创建
  - 扩展java.lang.Thread类(书本例子)
  - 实现Runnable接口

线程例子

```
public class DoSomething implements Runnable {
    private String name;

    public DoSomething(String name) {
        this.name = name;
    }

    public void run() {
            System.out.println("name:"  + name);
        }
}
```

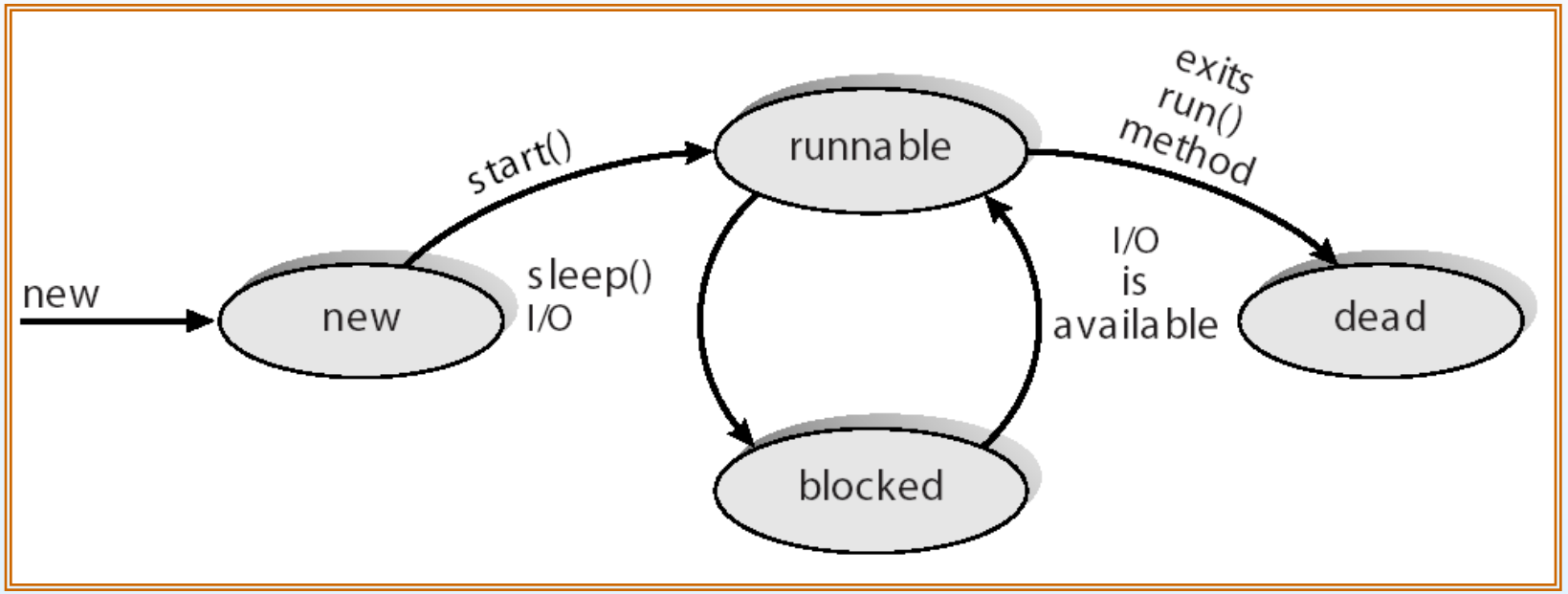
运行线程例子

```
public class TestRunnable {
    public static void main(String[] args) {
        DoSomething ds1 = new DoSomething("1");
        DoSomething ds2 = new DoSomething("2");

        Thread t1 = new Thread(ds1);
        Thread t2 = new Thread(ds2);

        t1.start();
        t2.start();
    }
}
```

Java线程状态


new
new
start()
sleep()
I/O
runnable
blocked
exits
run()
method
I/O
is
available
dead

4、隐式线程

隐式线程

- 随着进程中线程数量的增加，编程越来越大困难
- 新方法：由编译器或运行库创建或管理线程
- 三种模式：
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- 其它方法
  - Microsoft Threading Building Blocks (TBB)
  - `java.util.concurrent` package

Thread 池

- 在池中创建一批线程，等到任务
- 优点：
  - 利用线程池中的线程来响应请求比创建一个线程更加快速
  - 允许一个应用程序中的线程数量达到线程池的上限

- Windows API支持线程池：

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```

OpenMP

- C, C++, FORTRAN 支持
- 并行程序设计方法
- 识别并行区间

```
#pragma omp parallel
```

- 例子

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

```
#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（GCD）

- Apple在Mac OS X 和 iOS 引用的技术
- C，C++, Swift，API，运行库的扩展
- 识别并行区间
  - “^{}”- ^{ printf(“I am a block”); }
- GCD提供调度队列（dispatch queues）来处理提交的任务
  - 管理想GCD提交的任务并且以先进先出的顺序执行任务
- 顺序调度队列
  - 顺序队列中的任务同一时间只执行一件
- 并发调度队列
  - 并发队伍中的多个任务可以并行执行

Grand Central Dispatch（GCD）

- Apple在Mac OS X 和iOS引用的技术
- C, C++ ,swift, API, 运行库的扩展
- 识别并行区间
  - “^{}” - `^{ printf("I am a block"); }`
- GCD提供调度队列（dispatch queues）来处理提交的任务
  - 管理向GCD提交的任务并且以先进先出（FIFO）的顺序来执行任务

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

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

Grand Central Dispatch

- 顺序调度队列
  - 顺序队列中的任务同一时间只执行一件任务，每件任务只有在先前的任务完成后才开始
- 并发调度队列
  - 并发队列中的多个任务可以同时并行执行