第4章 线程、SMP和微内核

4.1* 进程和线程

4.2* 线程分类

4.3 多核和多线程

4.4～4.5 Windows8和Solaris线程

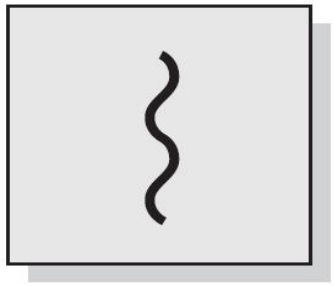
4.6～4.8 Linux和Android线程和Mac OS X的GCD技术(略)

4.1 进程process和线程thread

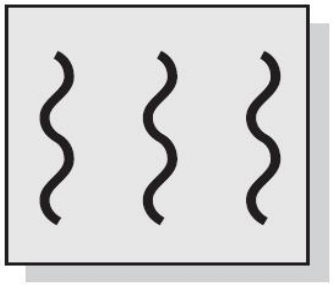
- 进程的两个特性：分配资源，被调度执行。

内存、设备、文件等　　CPU时间片

- 进程：分配资源的单位，不频繁地进行切换；
- 线程：被调度运行的单位，不拥有资源，可频繁调度切换，轻装运行。也称轻量级进程LWP。
- 传统：单线程进程；　　现在：多线程进程。



One process
One thread

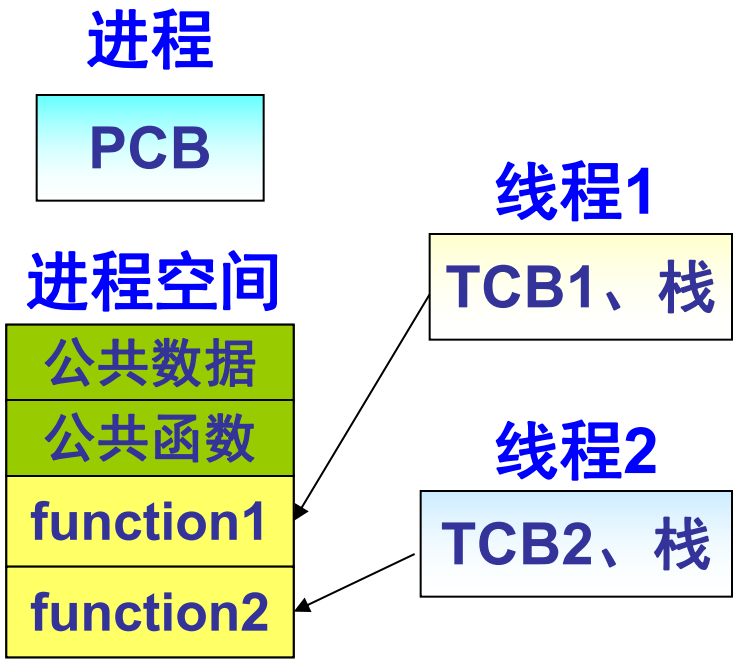


One process
Multiple threads

多个线程并发、甚至并行运行

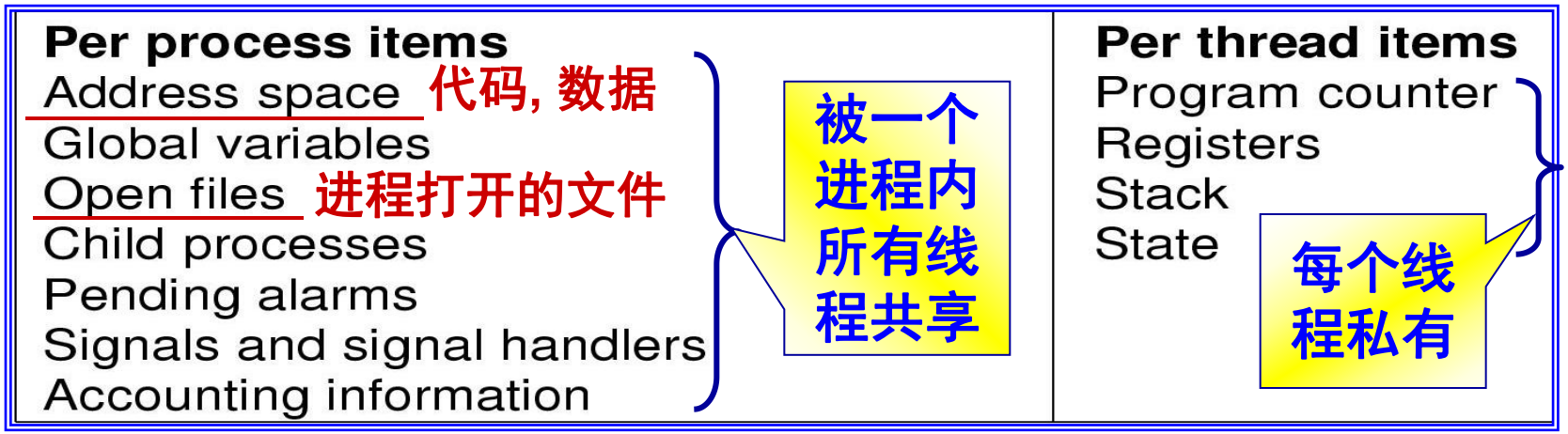
4.1.1 多线程－线程≠子进程

- 线程≠子进程：
  - 多个线程共享进程空间（内存、文件等），通信快。
  - 子进程空间各自独立，通信时需借助于IPC机制(进程间通信，如消息、管道、信号、共享内存区等)，慢。



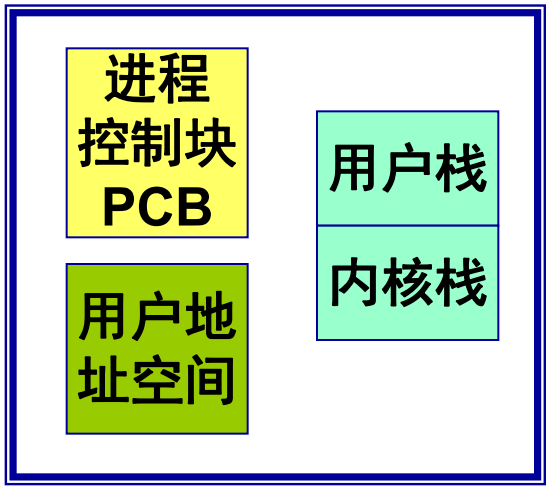
4.1.1 多线程－进程和线程区别

- 进程是资源分配和抢占CPU的单位，进程的资源及地址空间供其所有线程共享。
- 线程不拥有系统资源，线程执行环境小，同一进程的不同线程间切换和通信时开销少。
- 线程是一个进程内部的基本调度单位。一个进程可派生多个线程（执行多条代码流），线程间并发运行。
  - Java：继承Thread类，重写线程对象的run()方法，加入本线程要执行的代码，调用start()方法启动线程。

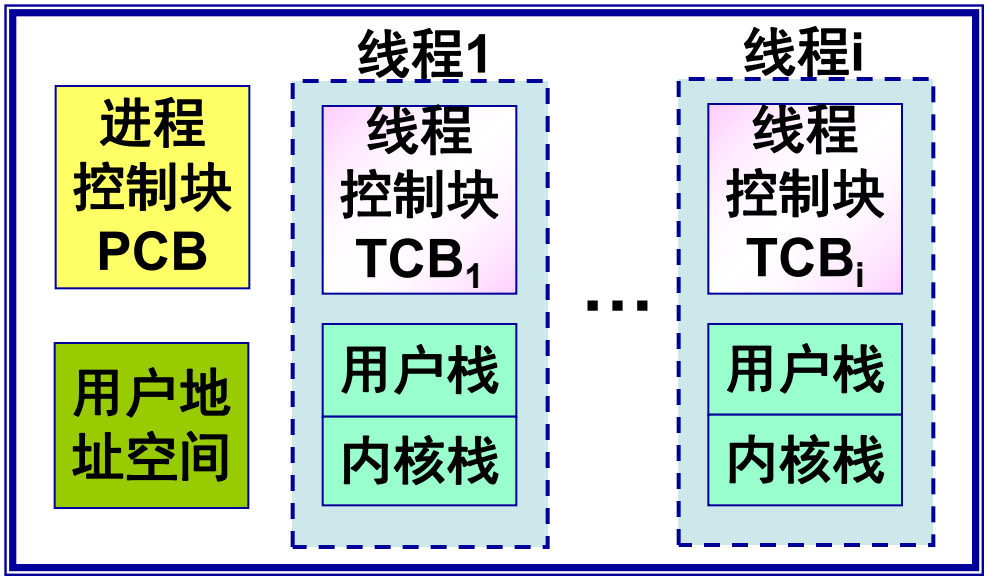


4.1.1 多线程—进程模型

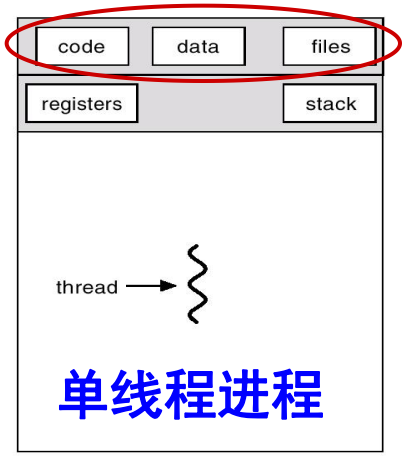
- 单线程进程模型：

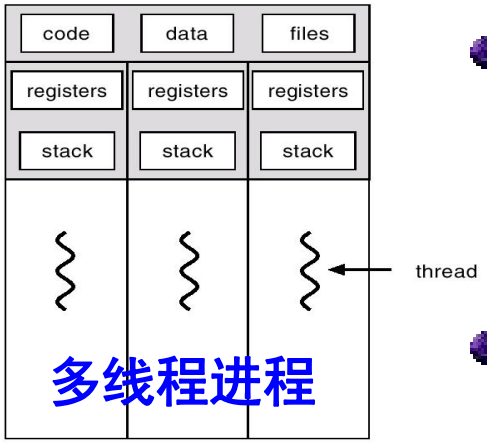


- 多线程进程模型：



用户地址空间

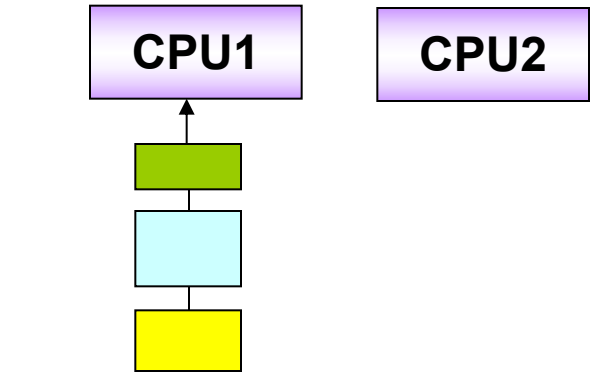




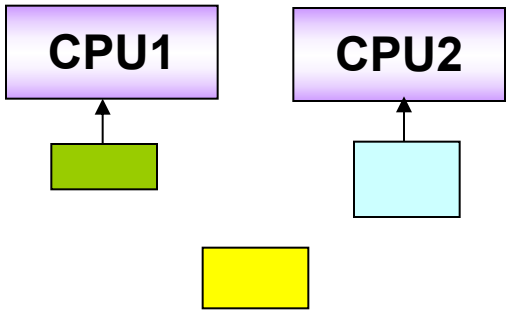
- 进程的所有线程共享该进程的用户地址空间和资源：所有线程可以访问同样的代码、数据和文件。
- 资源利用率高。多线程并发执行。可以利用多核处理器。

4.1.1 多线程－多线程的优点

- 创建/终止一个线程比创建/终止进程的时间少。
- 上下文切换开销少：对进程不频繁地进行切换；同一进程内各线程的切换速度快。
- 资源共享：进程的资源供其所有线程共享；线程间共享内存，通信方便。
- 并发程度高，可利用多处理器结构。



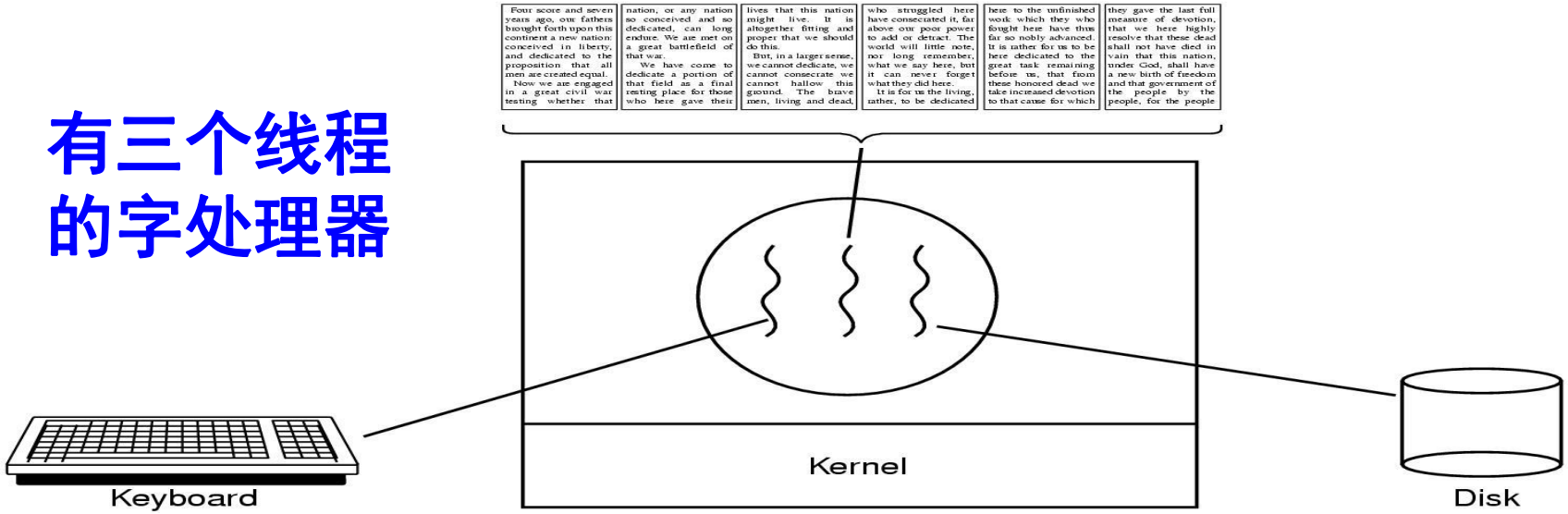
单线程进程：多个模块串行执行



多线程进程：多个线程代码模块
并发/并行执行

4.1.1 多线程－多线程的应用例1

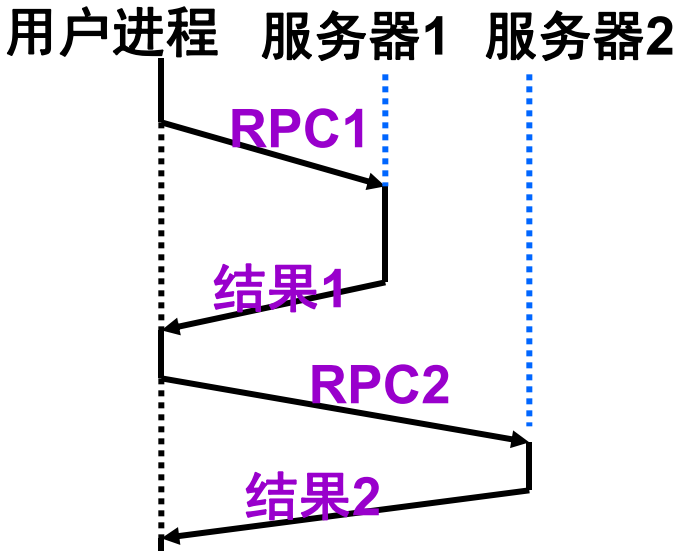
有三个线程
的字处理器



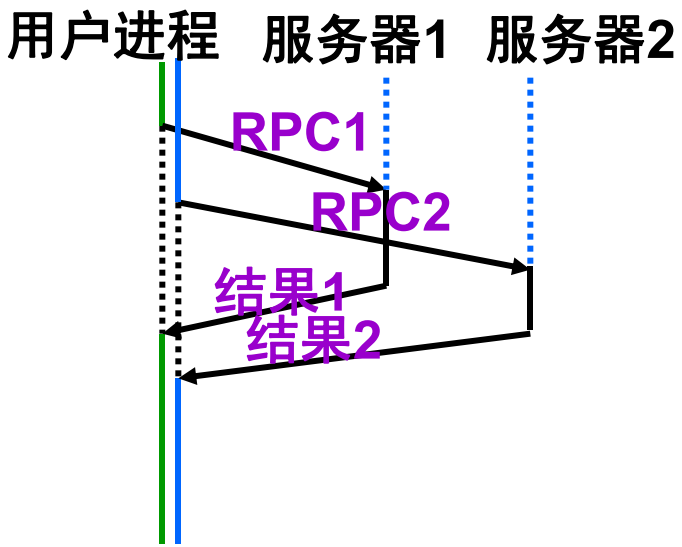
- 这个例子中，线程有两个作用：
  - 前台/后台处理：一个进程产生多个线程来接收用户的不同请求，并分别放在前台和后台处理，后台任务在处理机空闲时进行。
  - 异步处理：将程序中可同时执行的若干部分或异步的外部事件用不同的线程来同时执行或处理，加快响应。

4.1.1 多线程－多线程的应用例2

- **远程过程调用RPC：**对客户端发出的每个RPC使用一个独立的线程，获取服务器的响应。
- 这个例子里，并发等待两个应答，速度快。



用户进程只用单线程时

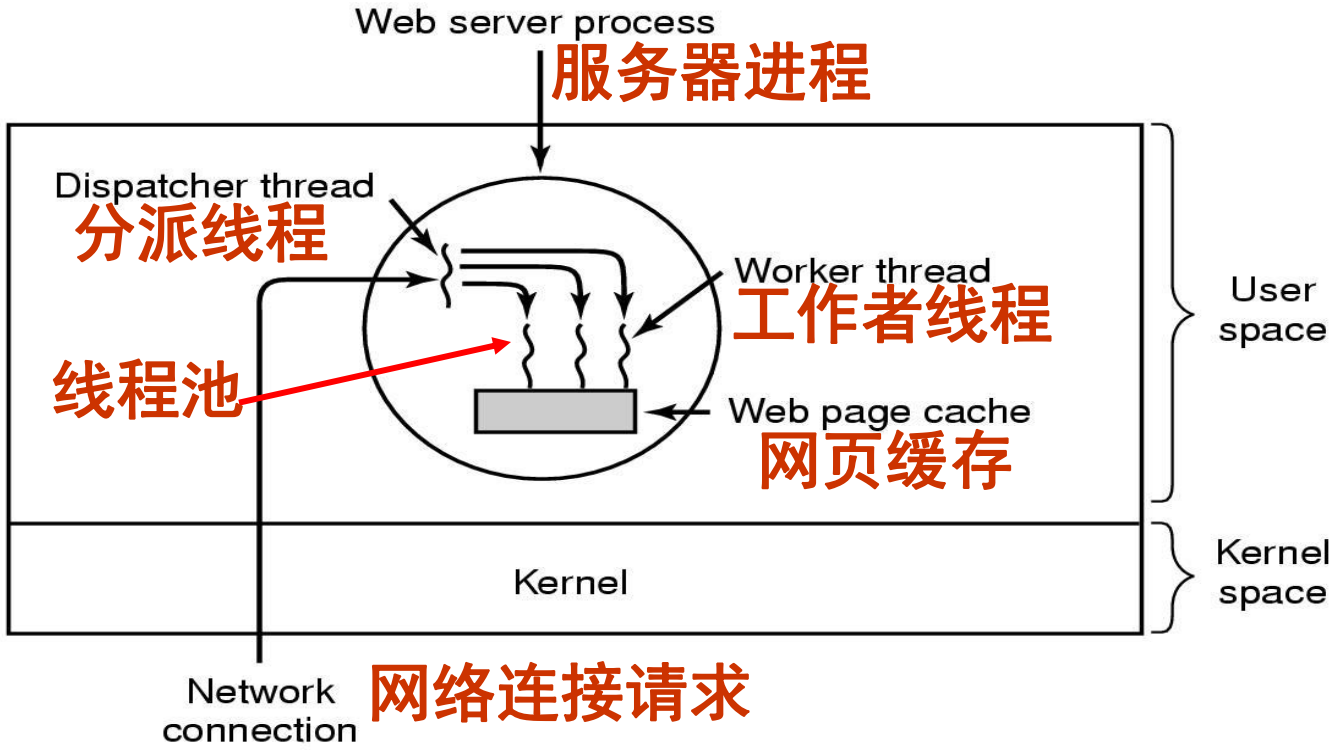


用户进程采用多线程时

4.1.1 多线程－多线程的应用例3

- Web服务器中，一个文件/通信服务进程可派生出多个线程，构成线程池。有请求时，唤醒一个工作者线程提供服务，服务完毕后放回池中睡眠。
  - 用一个已存在的线程响应请求要比新建一个线程更快。但线程池限制了线程的数量(可能发生拒绝服务式攻击DoS)。

多线程的Web服务器



4.1.2 线程功能－线程的三种状态

- 线程基本状态：运行态、就绪态、阻塞态。
- 挂起/终止一个进程会导致其所有线程被同时挂起/终止。
- 与进程类似，多个线程并发执行可以提高系统效率，但需要线程间的同步互斥。

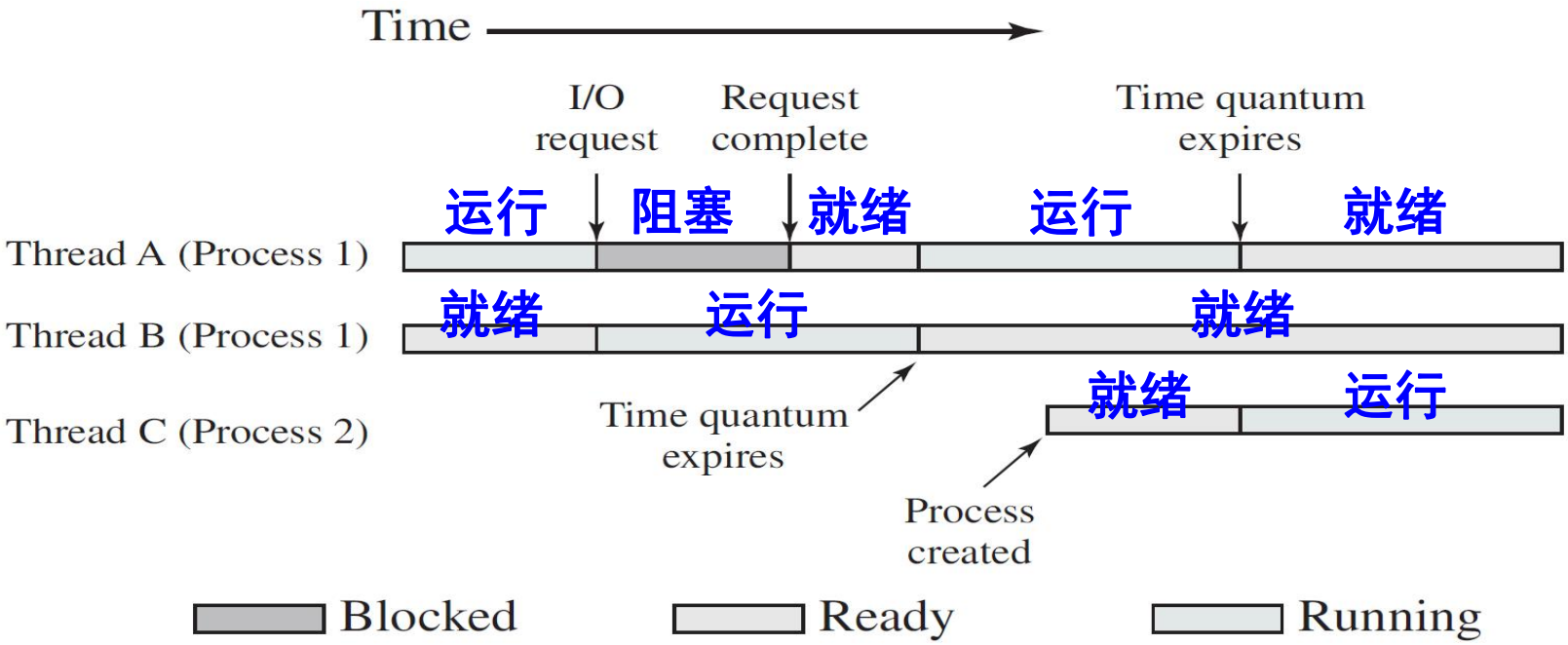


图4.4 单处理器上的多线程状态例子

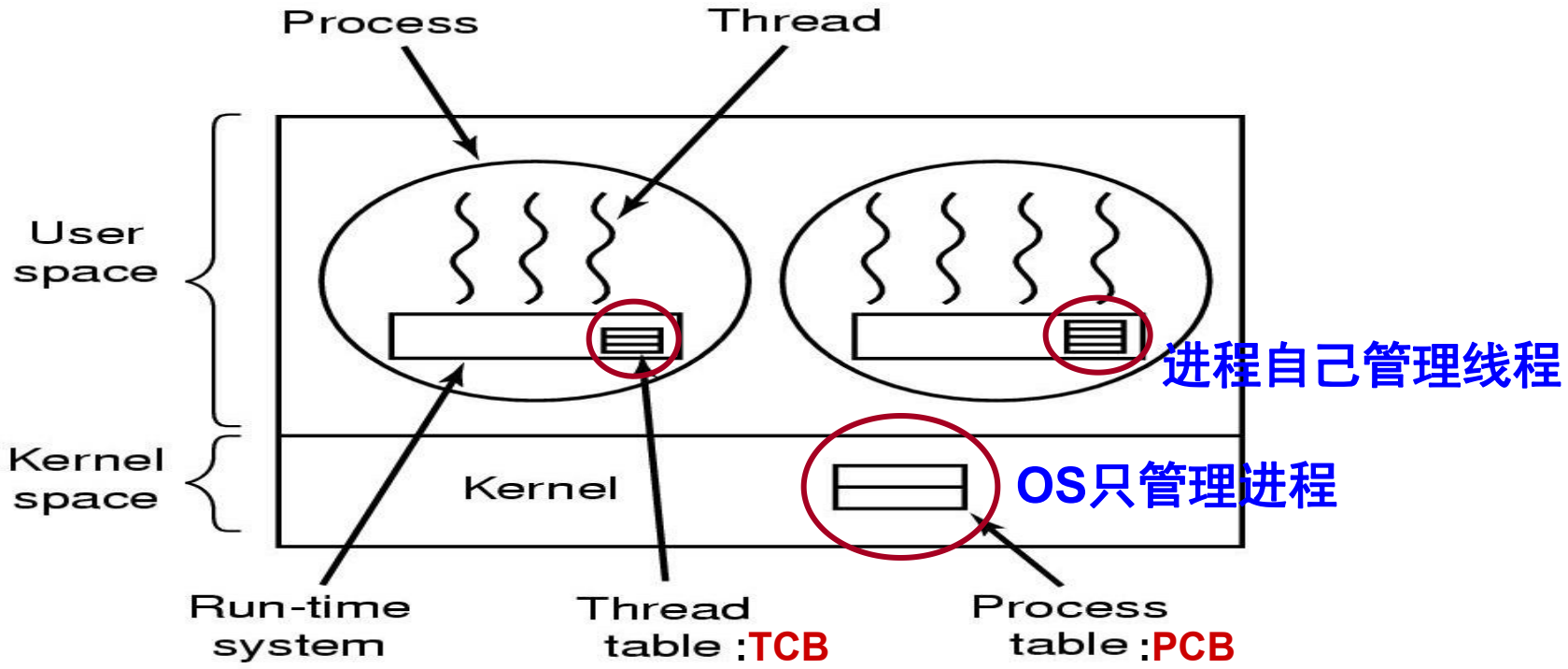
4.2 线程分类

4.2.1 用户级线程和内核级线程

- 线程可分为两类：
  - 用户级线程（User-Level Thread，ULT）
  - 内核级线程（Kernel-Level Thread，KLT），也称内核支持的线程、轻量级进程。
  - Solaris既支持ULT也支持KLT（混合方法）。

4.2.1 用户级线程和内核级线程—ULT

- **用户级线程**的创建、撤消和调度与OS内核无关，由用户空间中的**线程库**完成。OS内核并不知道用户级线程的存在，进程自己管理各自的线程。

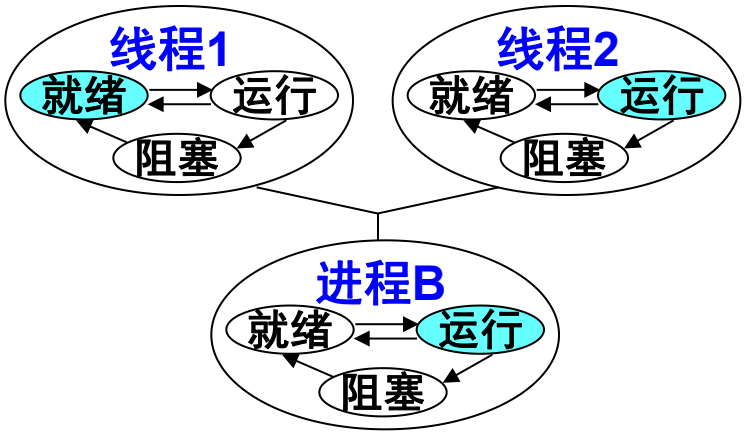


- 例如: POSIX Pthreads, Informix, JavaVM, Solaris

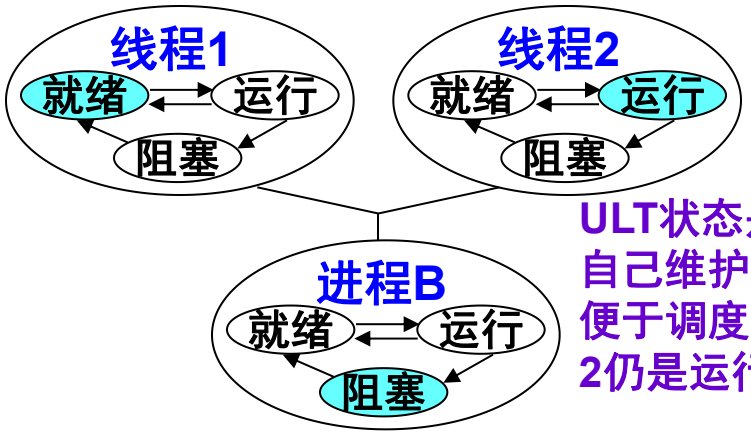
4.2.1 用户级线程和内核级线程—ULT

- 内核为进程分配CPU，进程调度其某个ULT运行。

a) 进程B在运行它的线程2

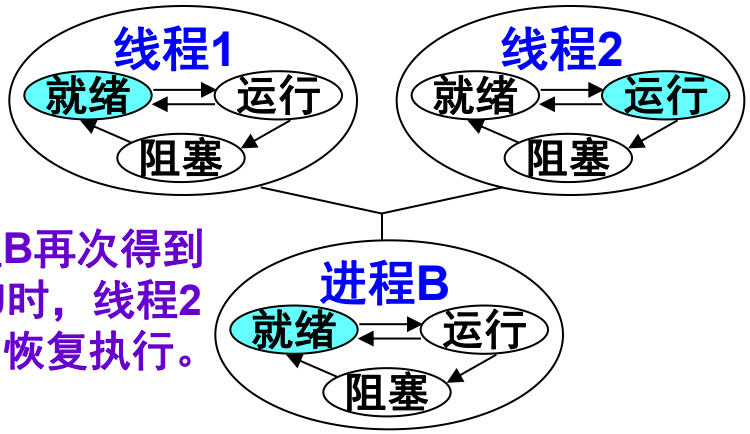


b) 线程2 I/O，阻塞了进程B



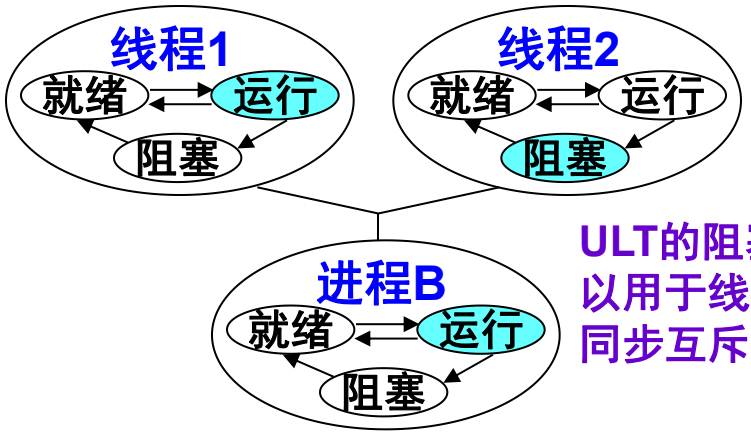
ULT状态是进程自己维护的，为便于调度，线程2仍是运行态。

c) 进程BI/O完毕或时间片超时



进程B再次得到CPU时，线程2立即恢复执行。

d) 线程2需要等线程1先运行



ULT的阻塞态可以用于线程间的同步互斥。

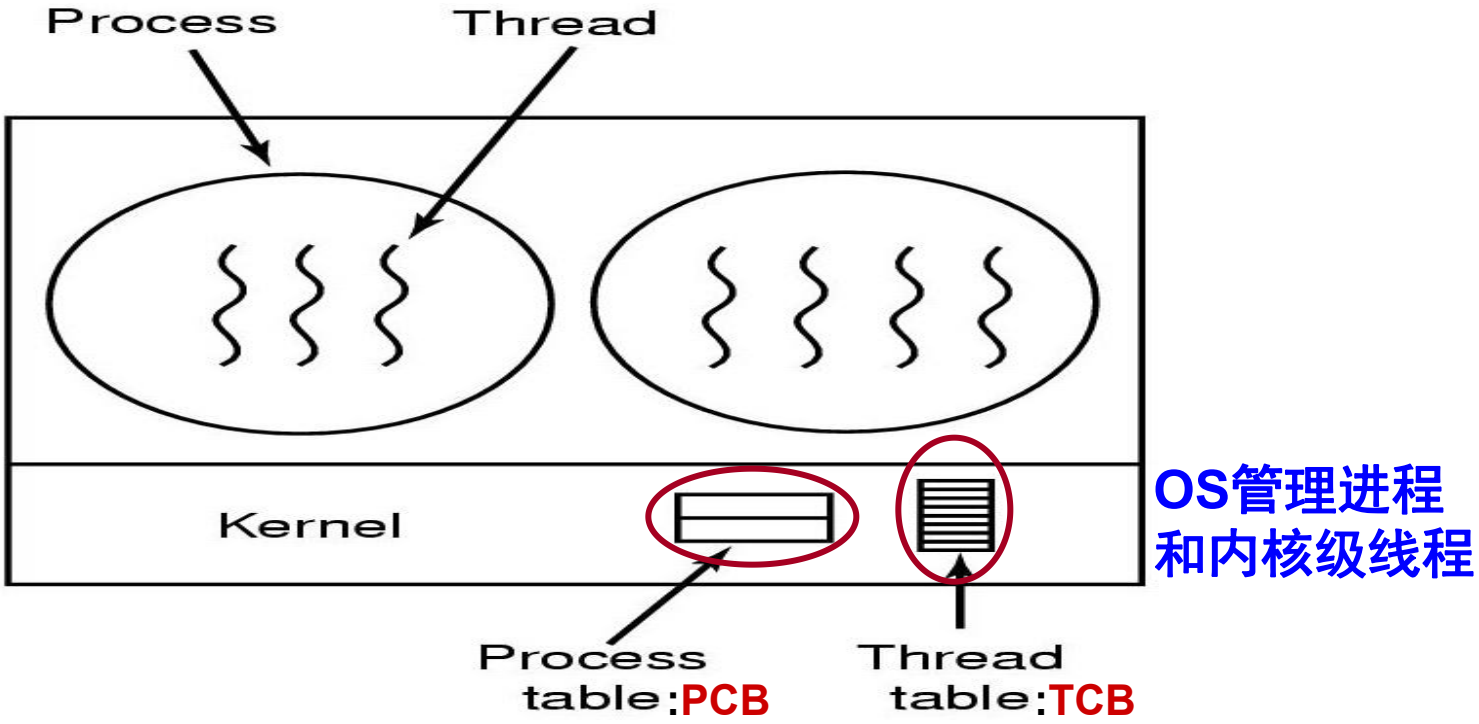
图4.6 ULT状态与进程状态例子

4.2.1 用户级线程和内核级线程—ULT

- 用户级线程的优点：
  - 一个进程中的ULT切换时，不需要切换到系统态。
  - 进程自己决定如何调度线程，更灵活。
  - 只要有线程库，用户级线程可以在任何OS上运行，而不需要修改底层的OS内核代码。
- 用户级线程的缺点：
  - ULT提出阻塞式系统调用时，将阻塞所属进程。
  - ULT对OS不可见，CPU分配以进程为单位，因此不能利用多CPU结构。

4.2.1 用户级线程和内核级线程－KLT

- **内核级线程**的创建、撤消、调度及同步互斥由OS内核完成，OS通过TCB控制内核级线程。
  对内核级线程的管理类似于进程。



- 例如: Windows, Solaris, Tru64 UNIX, BeOS, Linux

4.2.1 用户级线程和内核级线程－KLT

- 内核级线程的优点：
  - 可以同时把一个进程的多个KLT调度到多个CPU上。
  - 一个KLT阻塞时，OS可调度该进程的另一个KLT。
- 内核级线程的缺点：
  - 在调度KLT时，需要CPU切换到系统态，开销大。

单CPU VAX机，UNIX，两种操作所需时间 (μs)			
操作	用户级线程	内核级线程	进程
Null fork	34	948	11300
signal waiting	37	441	1840

4.2.1 用户级线程和内核级线程

- 用户级线程和内核级线程的比较：

	用户级线程	内核级线程
调度和切换	在一个进程的线程间切换，快。以进程为单位分配CPU。不能利用多CPU结构。	OS管理内核级线程和进程，以内核级线程为单位分配CPU。可利用多CPU结构。
当调用阻塞式系统调用时	阻塞该用户级线程所属的进程。	只阻塞该内核级线程本身。
运行时间	一个进程内所有用户级线程分享一个时间片。	每个内核级线程运行一个时间片。

4.2.2 其他方案

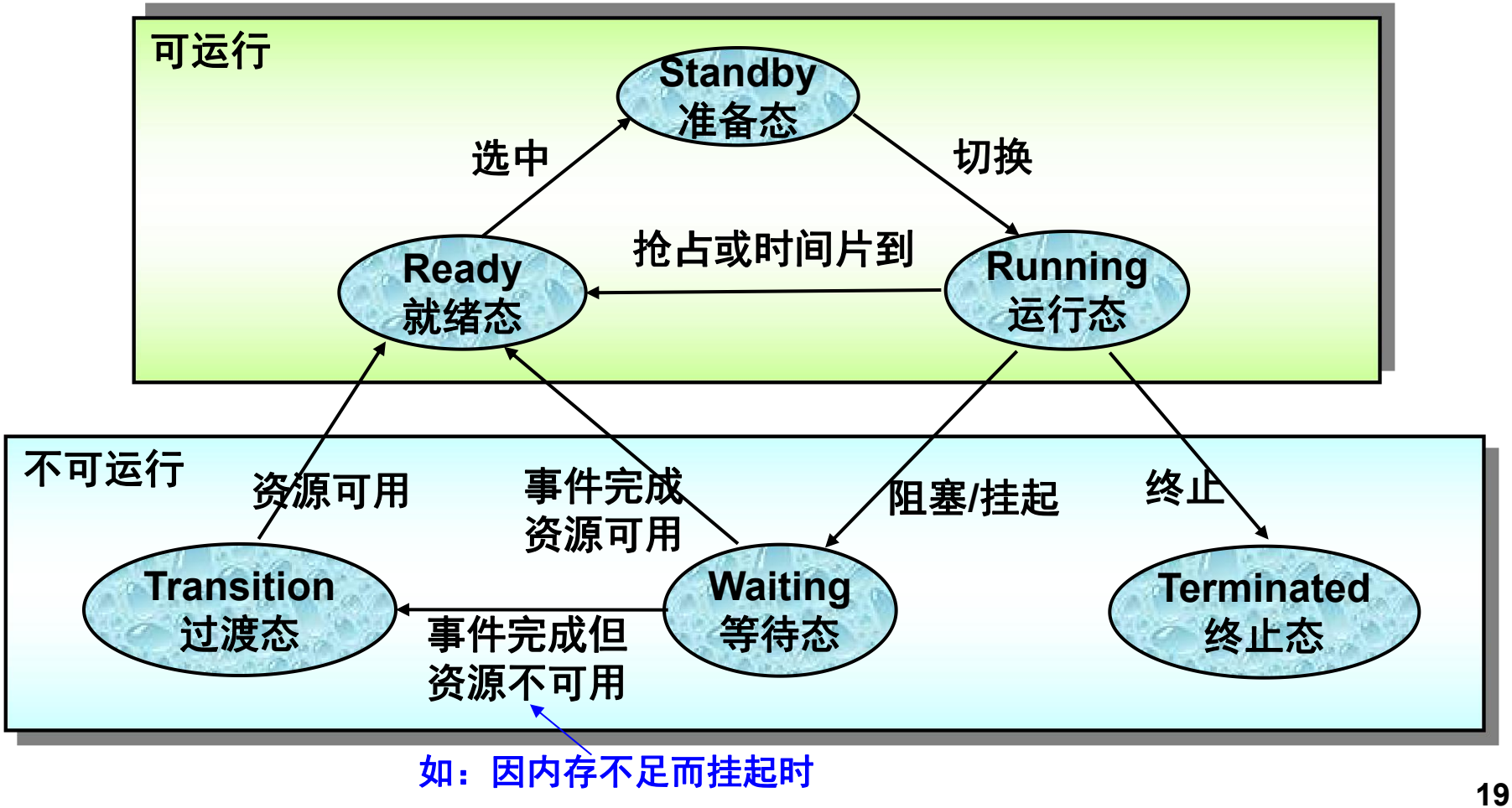
- 线程与所属进程间的数量关系：

线程:进程	描述	实例系统
1:1 一对一	传统的单线程进程。一个进程只产生一个线程。	传统UNIX
M:1 多对一	一个进程拥有地址空间和资源。在该进程中创建和执行多个线程。	Windows NT, Solaris, Linux等
1:M 一对多	一个线程可从一个进程环境迁移到另一个进程环境，甚至横跨不同计算机。线程迁移时携带状态信息。	RS(Clouds), Emerald
M:N 多对多	多个线程可在一个域（地址空间）中执行，应用程序可能在多个域中执行(产生多个进程)。线程可在域间迁移。	TRIX

4.3 在多核系统上，多线程可提升程序加速比和数据库负载及吞吐量。

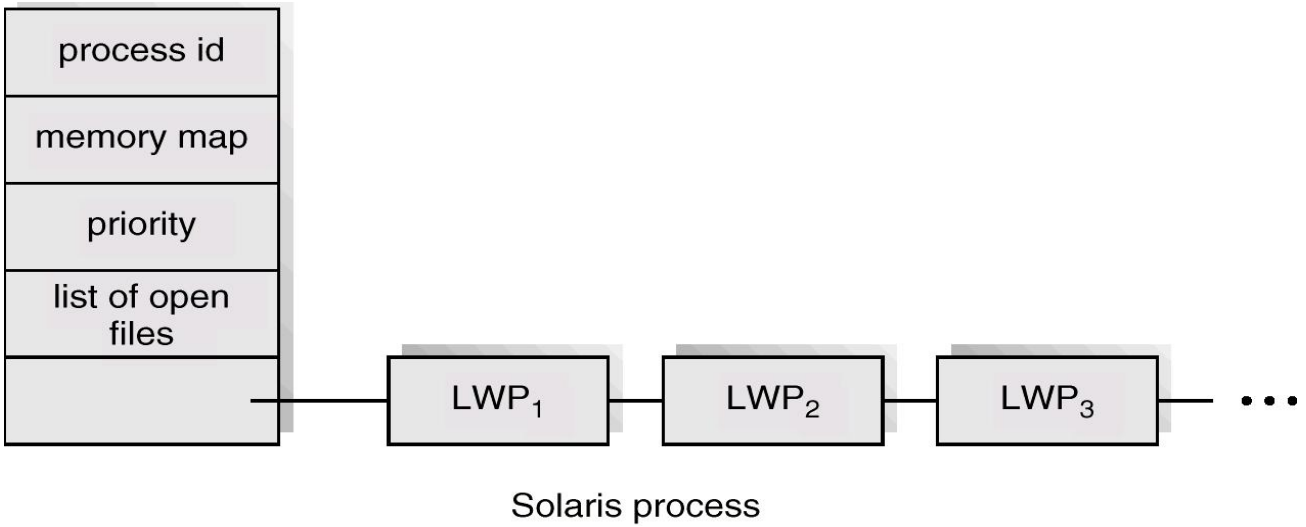
4.4 Windows 8进程和线程管理

- Windows 中处理器的调度单位是KLT。
- Windows线程状态：

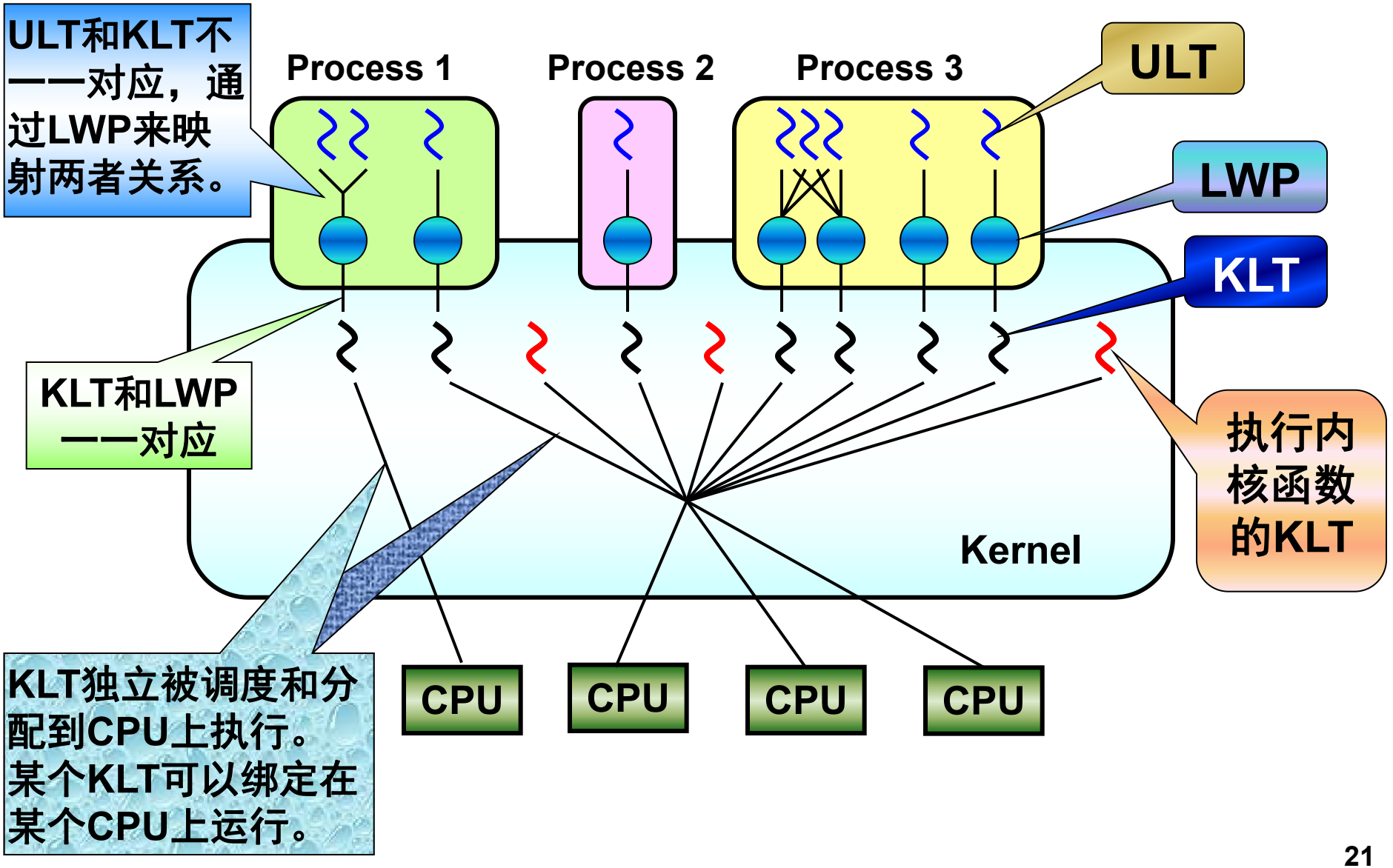


4.5 Solaris线程和SMP管理

- Solaris支持以下4种进程和线程概念：
  - 进程
  - 用户级线程ULT
  - 轻量级进程LWP（Light Weight Process）
  - 内核级线程KLT



4.5 Solaris线程和SMP管理



4.6 Linux的进程和线程管理（略）

4.7 Android的进程和线程管理（略）

4.8 Mac OS X的GCD技术（略）

作业：

- 复习题：5，6，7