第4章 线程

进程与线程

线程分类

多核与多线程

对称多处理

微内核

现代操作系统相关技术

4.1 进程与线程

- 进程（process）=>进程映像=代码、数据、栈、PCB
- 进程概念的两种属性
  - **资源分配的单位：**资源的控制或所有权属于进程。操作系统执行保护功能，以防止进程之间发生不必要的与资源相关的冲突
  - **调度/执行的单位：**进程沿着一条执行路径（轨迹）执行。可能与其他进程的执行过程交替进行。

4.1 进程与线程

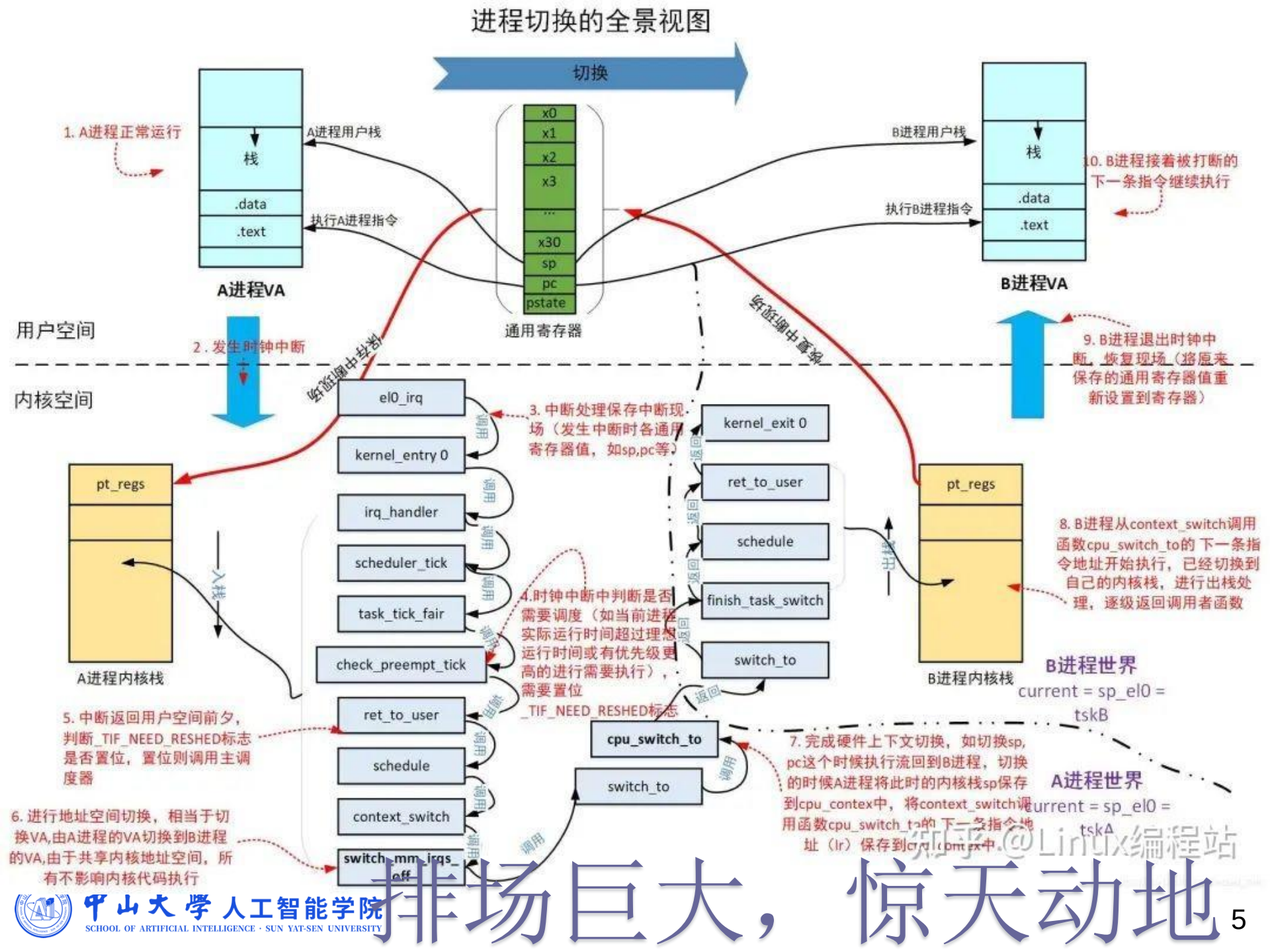
- 传统操作系统中进程机制同时呈现 资源分配单位 和 调度/执行单位两种属性

- 两种属性本质上独立，可分开处理：用进程（任务）作为资源所有权单位，用线程（thread）/ LWP（Light Weight Process，轻量级进程）作为**CPU调度/执行/分派单位**

为什么需要线程（thread）

在没有线程概念的系统中，进程是

- 资源分配的单位
- 调度/执行的单位

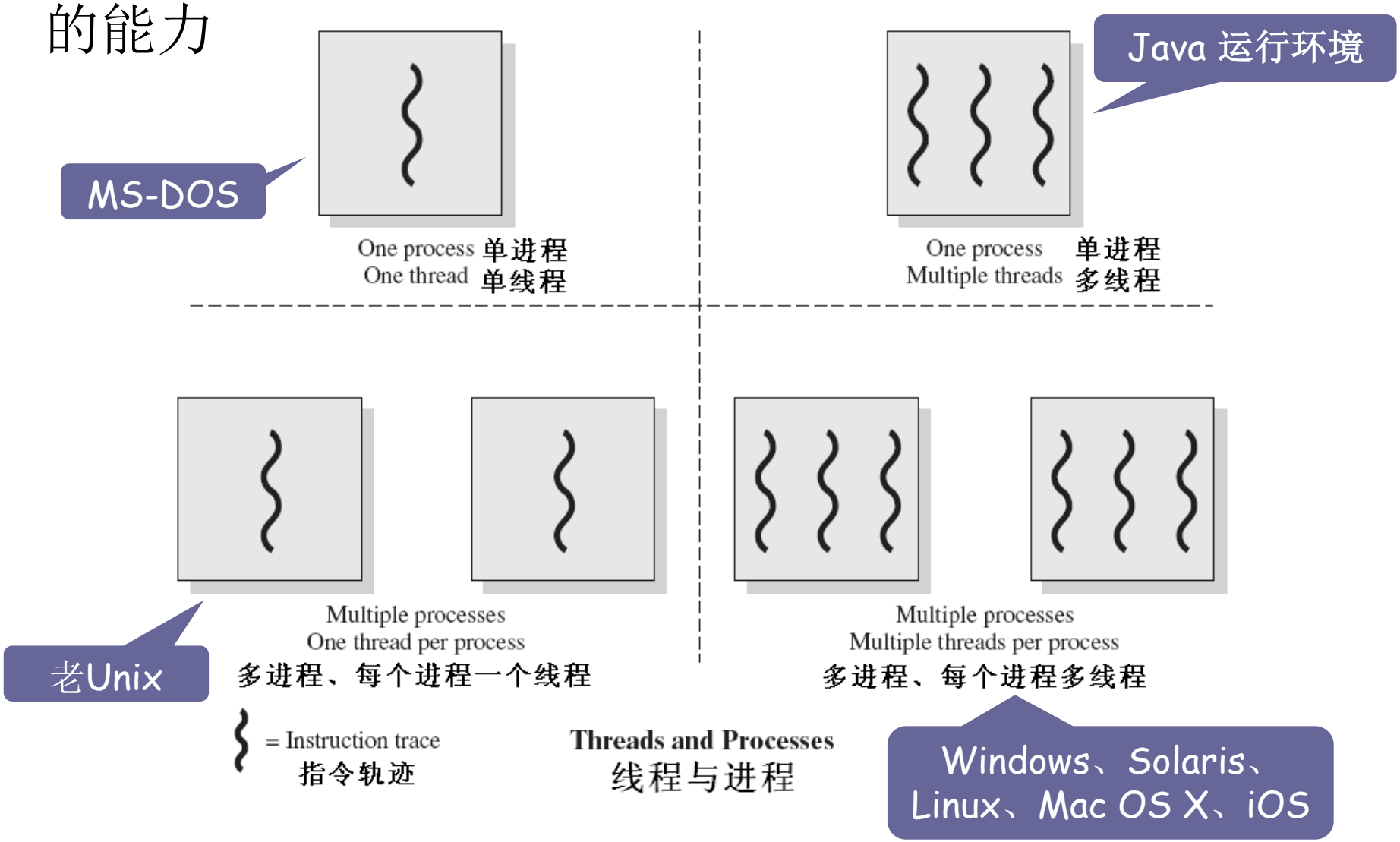

进程切换的全景视图
切换
1. A进程正常运行
栈
.data
.text
A进程用户栈
执行A进程指令
A进程VA
x0
x1
x2
x3
...
x30
sp
pc
pstate
通用寄存器
B进程用户栈
执行B进程指令
栈
.data
.text
B进程VA
10. B进程接着被打断的下一条指令继续执行
用户空间
2. 发生时钟中断
内核空间
异常返回中存储
异常返回中重载
9. B进程退出时钟中断，恢复现场（将原来保存的通用寄存器值重新设置到寄存器）
el0_irq
kernel_entry 0
irq_handler
scheduler_tick
task_tick_fair
check_preempt_tick
ret_to_user
schedule
context_switch
switch_mm_irqs_off
调用
3. 中断处理保存中断现场（发生中断时各通用寄存器值，如sp,pc等）
kernel_exit 0
ret_to_user
schedule
finish_task_switch
switch_to
返回
pt_regs
入栈
A进程内核栈
4.时钟中断中判断是否需要调度（如当前进程实际运行时间超过理想运行时间或有优先级更高的进行需要执行），需要置位_TIF_NEED_RESHED标志
pt_regs
出栈
B进程内核栈
8. B进程从context_switch调用函数cpu_switch_to的 下一条指令地址开始执行，已经切换到自己的内核栈，进行出栈处理，逐级返回调用者函数
B进程世界
current = sp_el0 = tskB
5. 中断返回用户空间前夕，判断_TIF_NEED_RESHED标志是否置位，置位则调用主调度器
cpu_switch_to
switch_to
7. 完成硬件上下文切换，如切换sp,pc这个时候执行流回到B进程，切换的时候A进程将此时的内核栈sp保存到cpu_contex中，将context_switch调用函数cpu_switch_to的 下一条指令地址（lr）保存到cpu_contex中
A进程世界
current = sp_el0 = tskA
6. 进行地址空间切换，相当于切换VA,由A进程的VA切换到B进程的VA,由于共享内核地址空间，所有不影响内核代码执行

为什么需要线程（thread）

- 主要问题：
  - 进程切换开销大：每次切换都要保存和恢复进程所拥有的全部信息(PCB、有关程序段和相应的数据集等)
  - 进程占用资源多：多个同类进程需占用多份资源，而一个进程中的多个同类线程则共享一份资源
- **线程**：一个进程内的***基本调度单位***
  - 一个进程可有一个或多个线程

4.1.1 多线程（Multithreading）

- **多线程**：操作系统支持在一个进程中执行多个线程的能力



多线程环境下的进程和线程

进程/任务

资源分配和保护单位

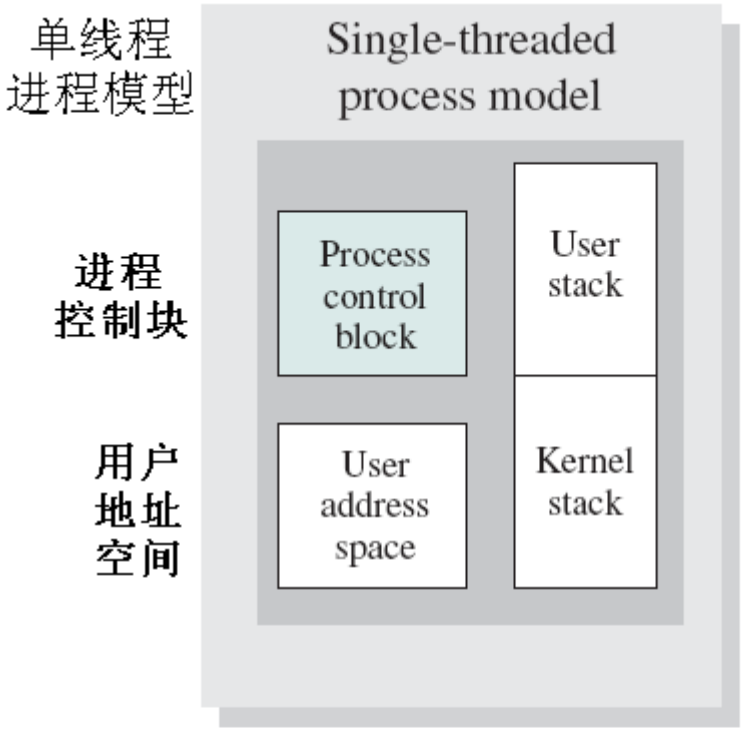
- 拥有用于保存进程映像的虚地址空间
- 受保护地访问处理器、其他进程、文件和I/O资源

线程

分派的执行单位

- 执行状态（运行、就绪等）
- 线程上下文（非运行时）
- 一个执行栈
- 存储局部变量的静态存储空间
- 对进程的内存和其他资源的访问（同进程其他线程共享）

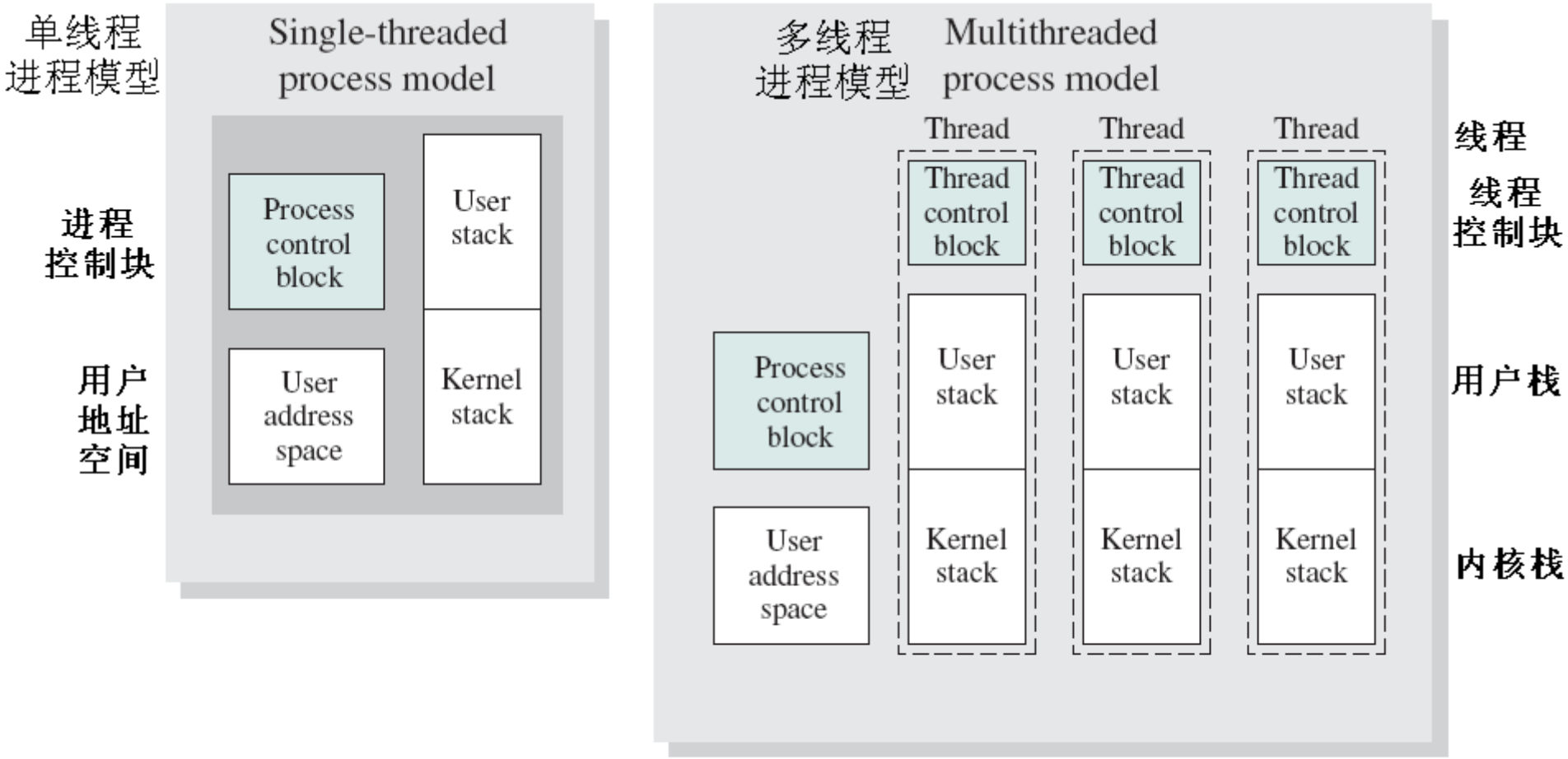
单线程和多线程的进程模型



Single-Threaded and Multithreaded Process Models

单线程与多线程进程模型

单线程和多线程的进程模型



Single-Threaded and Multithreaded Process Models

单线程与多线程进程模型

线程的优点(与进程比较)

- 创建速度快（在已有进程内）
- 终止所用的时间少
- 切换时间少（保存和恢复工作量小）
- 通信效率高（在同一进程内，无需调用OS内核，可利用共享的存储空间）

线程的应用

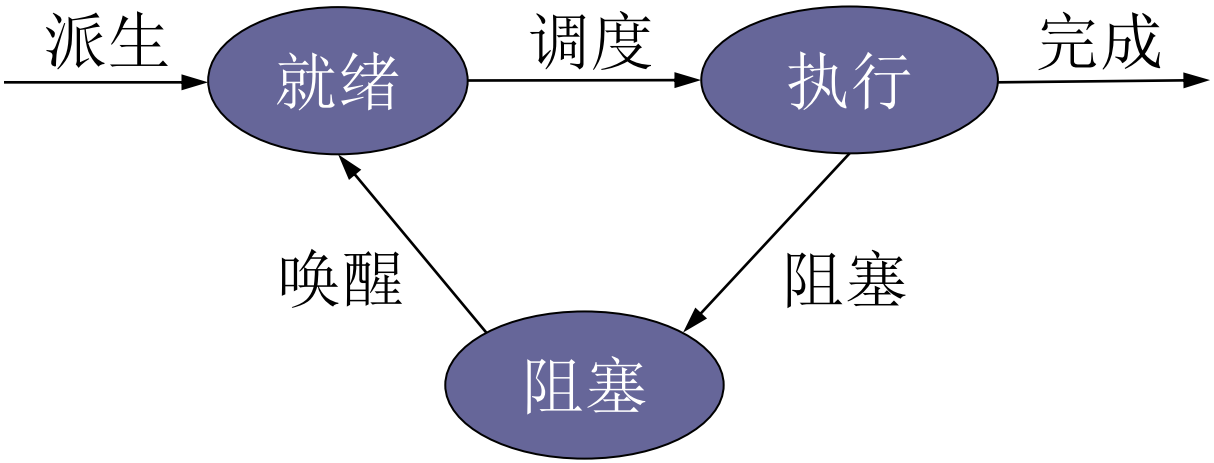
- 若应用程序*可按功能划分成不同的小段*，或*可划分成一组相关的执行实体*
- 则用一组线程（比用一组进程）可提高执行效率（尤其是在多处理器和多核系统中）

线程的应用

- 单用户多核/多处理器系统的典型应用：
  - 服务器中的文件管理或通信控制
  - 前台和后台操作（如前台用户交互、后台数据处理）
  - 异步处理（如字处理与周期性备份）
  - 加速执行（在多核/多处理器系统中的并行）
  - 模块化程序结构（涉及多种活动或多个I/O源和目的地的程序）

4.1.2 线程的功能特性（执行特征）

- 线程状态
  - 运行
  - 就绪
  - 阻塞



4.1.2 线程的功能特性（执行特征）

- 线程不拥有资源，且进程与其所有线程共享代码和地址空间。
- 线程拥有自己的寄存器上下文和栈空间（用来存储局部变量和调用参数）
- **挂起**状态、**终止状态**是进程级的概念

（共享地址空间）

  - 挂起一个进程，则该进程的所有线程也挂起
  - 终止一个进程，则该进程的所有线程也终止

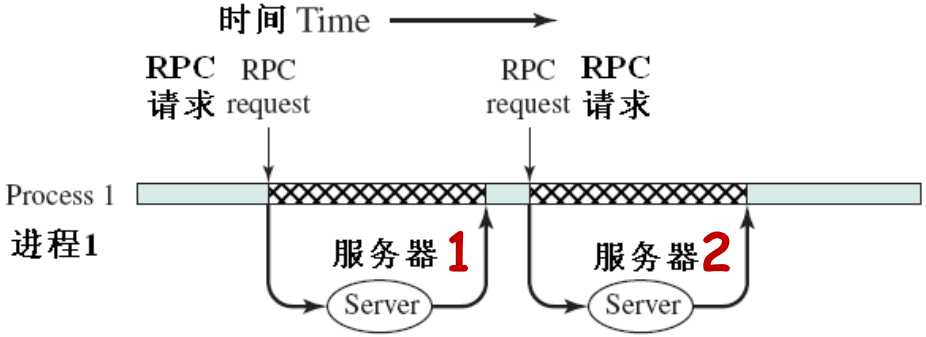
（共享代码段）

线程状态

- 与线程状态变化有关的操作：
  - **派生（spawn，产卵）**——线程可派生新线程
  - **阻塞（block）**——等待事件发生
  - **唤醒/解除阻塞（unblock）**——事件发生后被唤醒，转换到就绪态
  - **调度（schedule）**——由操作系统将就绪线程调度到处理器（核）中执行
  - **结束（finish）**——释放寄存器上下文和栈

例1：使用线程的RPC

- **远程过程调用**：在不同机器上执行两个程序的技术，它使用过程调用/返回的语法和语义。



(a) RPC using single thread
使用单线程的RPC

Blocked, waiting for response to RPC 阻塞，等待RPC响应

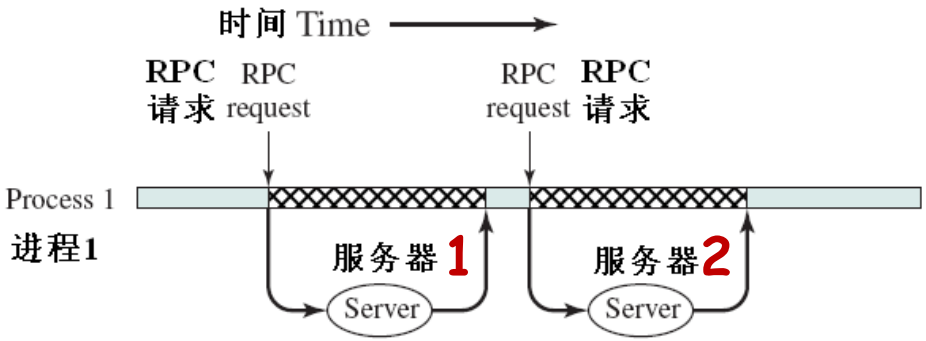
Blocked, waiting for processor, which is in use by Thread B 阻塞，等待正在被线程B使用的处理器

Running 运行

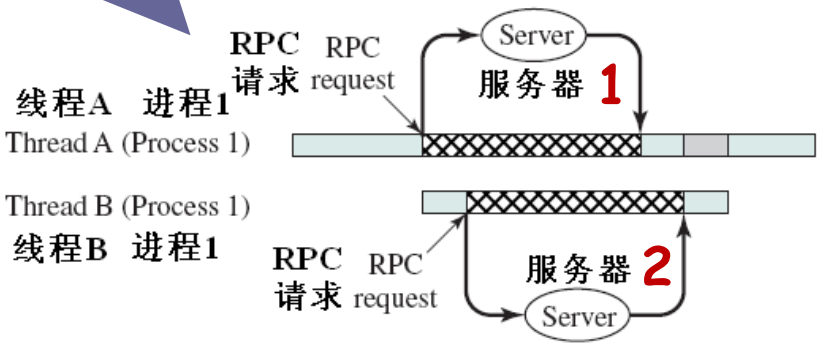
Remote Procedure Call (RPC) Using Threads
使用线程的远程过程调用（RPC）

例1：使用线程的RPC

一个线程阻塞不会
阻塞整个进程



(a) RPC using single thread
使用单线程的RPC



(b) RPC using one thread per server (on a uniprocessor)
每个服务器使用一个线程的RPC（在单处理器上）

Blocked, waiting for response to RPC 阻塞，等待RPC响应

Blocked, waiting for processor, which is in use by Thread B 阻塞，等待正在被线程B使用的处理器

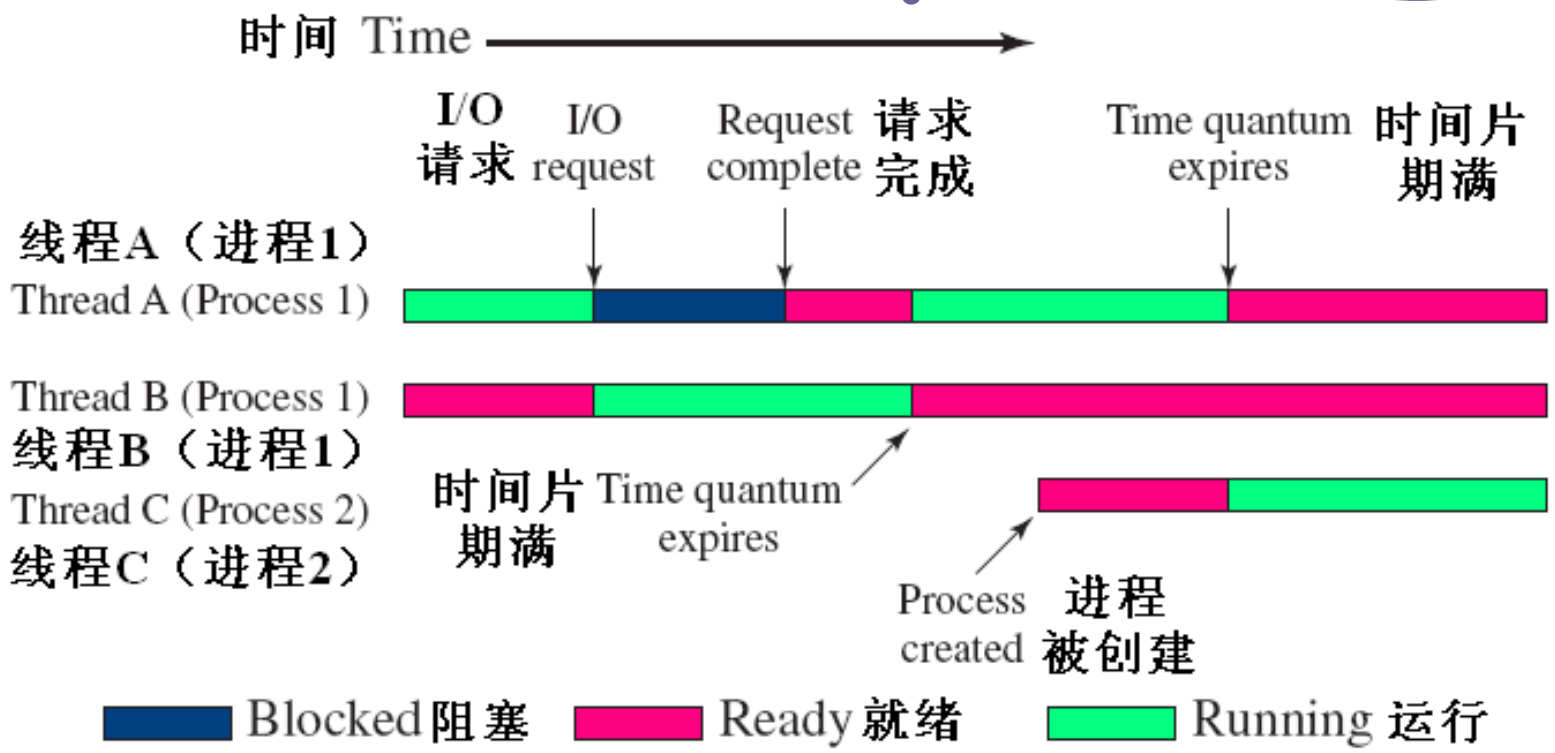
Running 运行

Remote Procedure Call (RPC) Using Threads

使用线程的远程过程调用（RPC）

例2：单个单核处理器上的多线程

多个进程中的多个线程可交替执行



Multithreading Example on a Uniprocessor

单处理器上的多线程例子

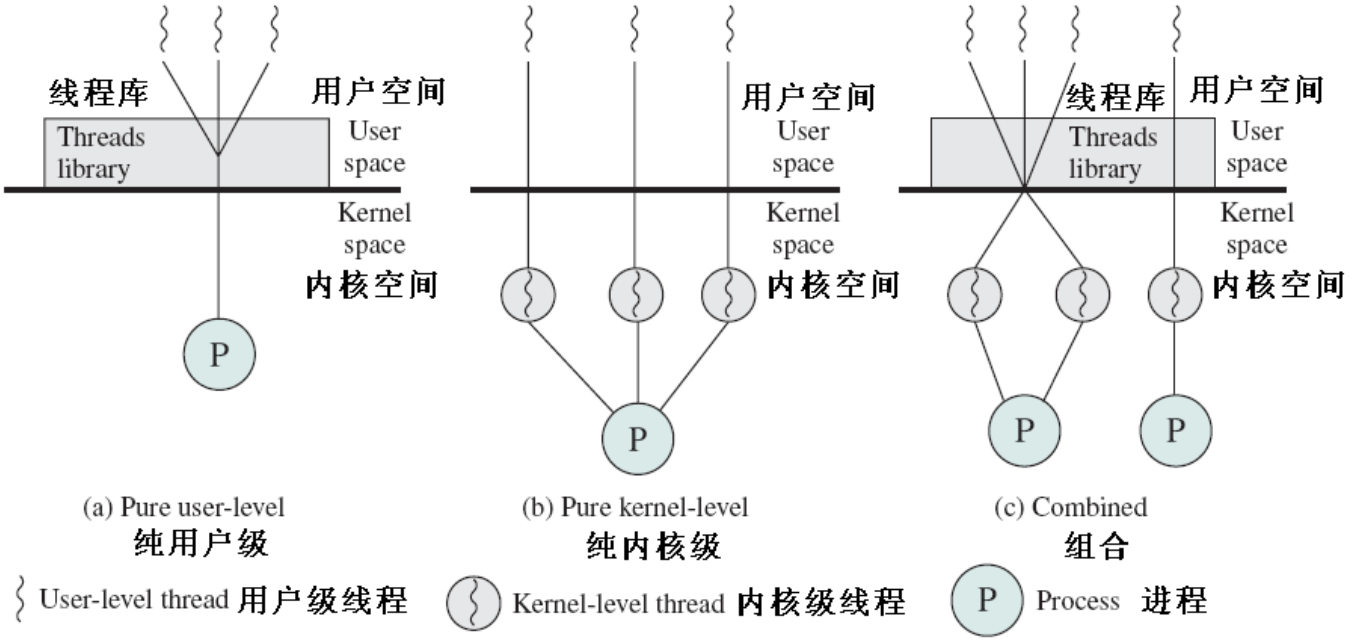
线程同步

- 同一进程的多个线程共享同一个地址空间和其他资源

- 线程同步
  - 需要同步各个线程的活动，使它们互不干涉且不破坏数据结构
  - 线程同步机制与进程同步机制相同

4.2 线程分类

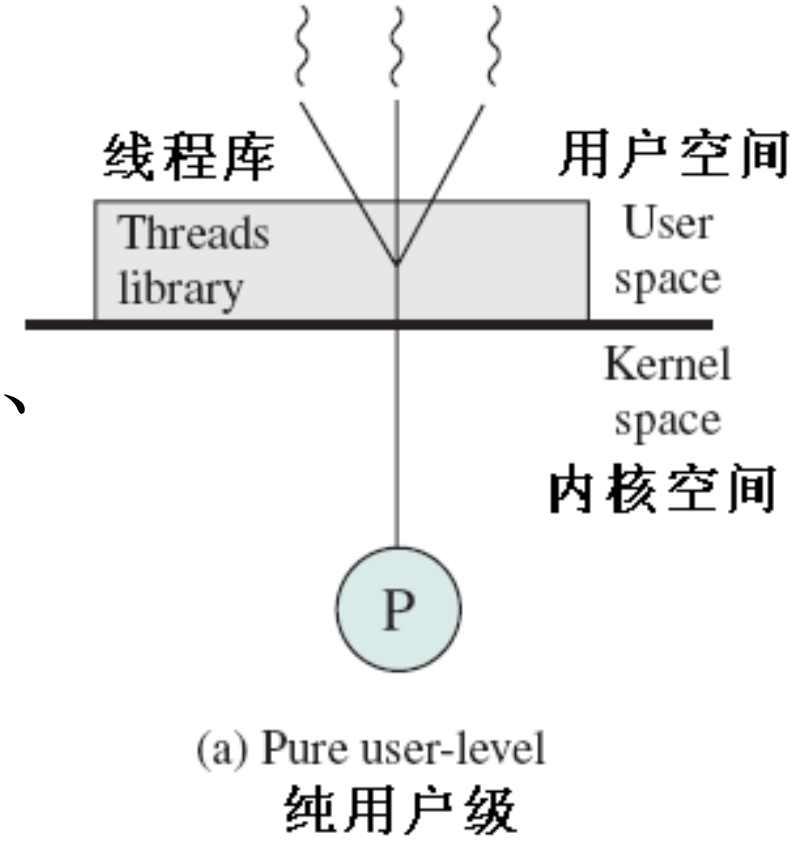
- 用户级线程(User-level Threads, ULT)
- 内核级线程(Kernel-level Threads, KLT)
- 混合方法/组合途径(Combined Approaches)



User-Level and Kernel-Level Threads
用户级与内核级线程

用户级线程(ULT)

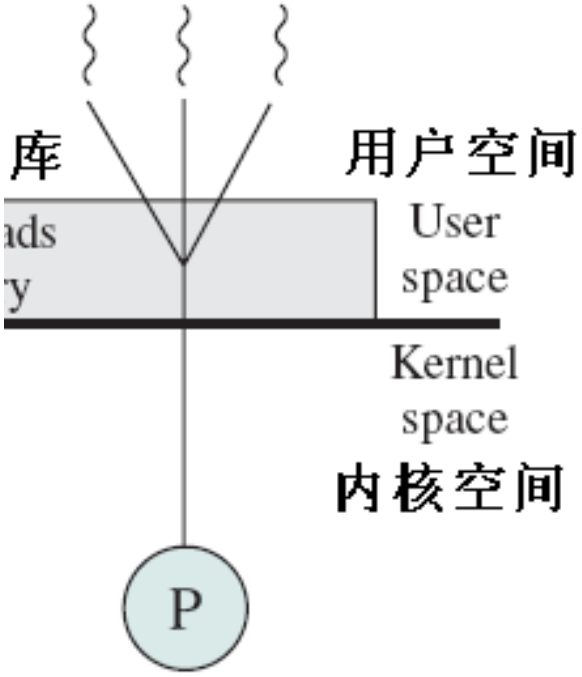
- 线程管理均由应用程序完成（线程库）
- 线程库含有创建和销毁线程、在线程间传递消息和数据的代码、调度线程和保存恢复线程上下文的代码
- 内核不知道线程的存在



(a) Pure user-level
纯用户级

用户级线程(ULT)

- 优点：
  - 线程切换不需要模式切换
  - 调度算法可应用程序专用
  - 用户级别线程不需内核支持，线程库可在任何操作系统上运行
- 缺点：
  - 一个线程阻塞会导致整个进程阻塞
  - 不能利用多核和多处理器技术



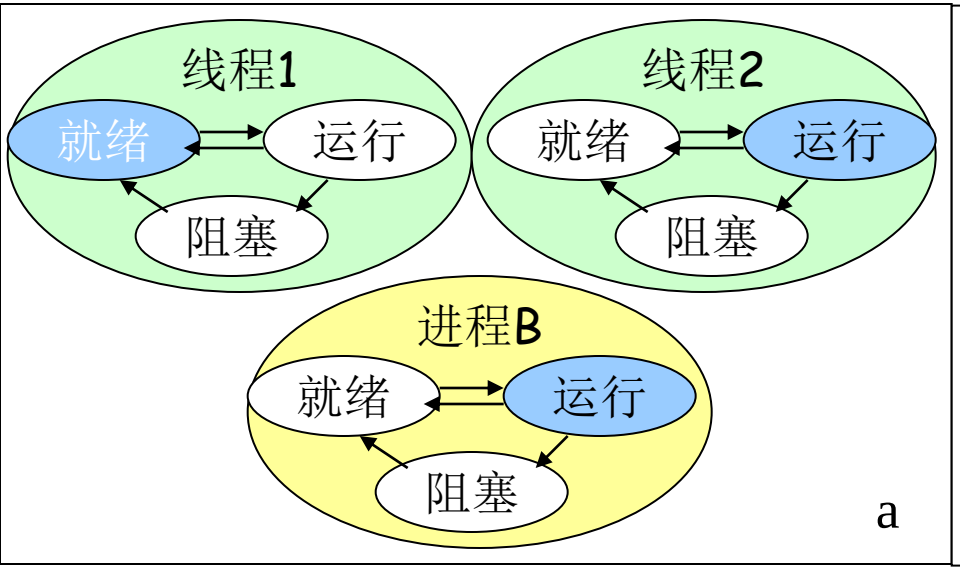
a) Pure user-level
纯用户级

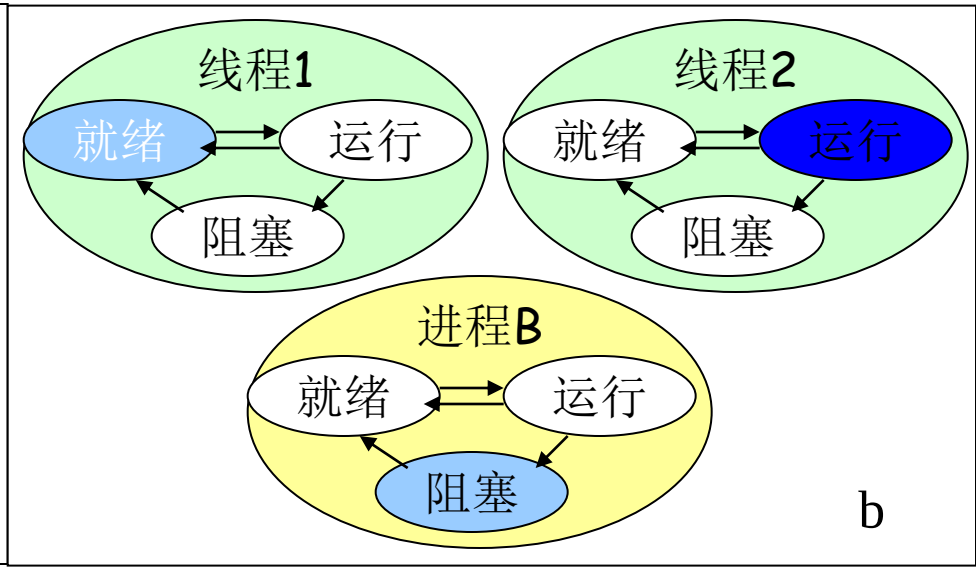
```c
#include <pthread.h>
#include <stdio.h>
#include <windows.h>
#include<string.h>
#define NUM 5
void *print_msg(void *m)
{
    char *cp = (char *)m;
    int i;
    for (i = 0; i < NUM; i++)
    {
        for (int j = 0; j < strlen(cp);j++)
            printf("%c", cp[j]);
        fflush(stdout);
        Sleep(1);
    }
    return NULL;
}
int main()
{
    pthread_t t1, t2; /* two threads */
    pthread_create(&t1, NULL, print_msg, (void *)"hello");
    pthread_create(&t2, NULL, print_msg, (void *)"world\n");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

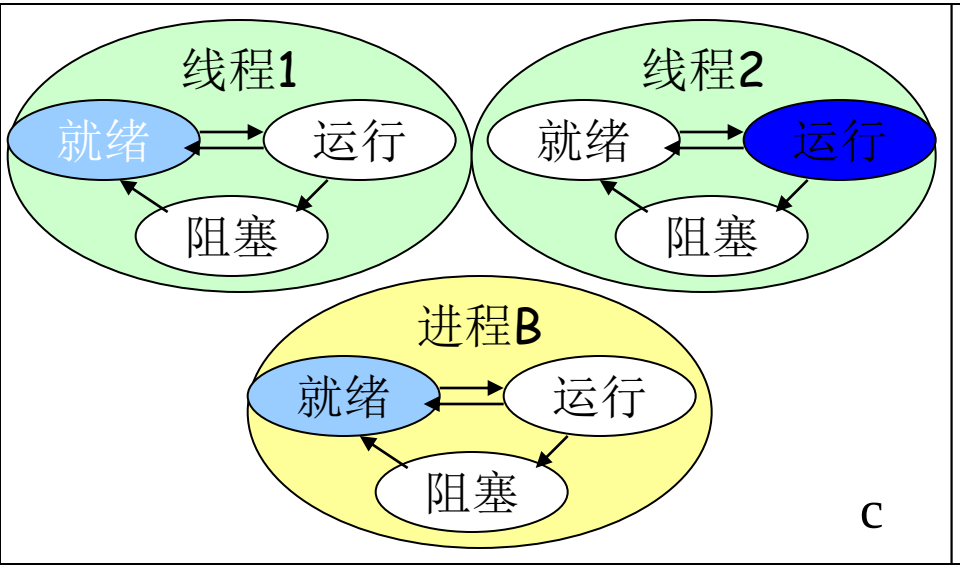
运行结果

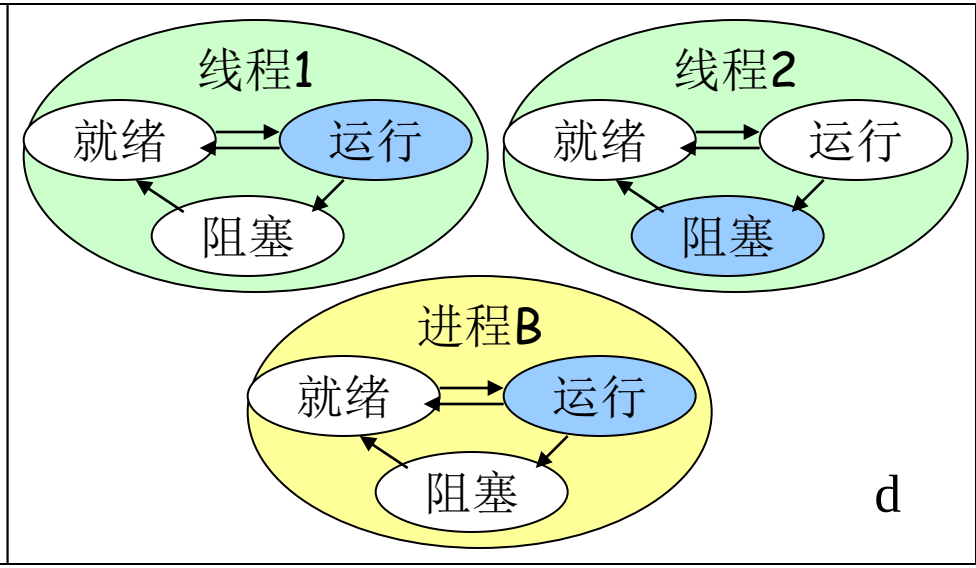
```
PS C:\Users\lz\Desktop\c++> .\a.exe
hworld
elloworld
hellohelwlorldo
hellwoorld
whoerlllod
PS C:\Users\lz\Desktop\c++> .\a.exe
hewlolrold
world
hellowoherllld
owohrelldl
ohweolrllod
```

用户级线程状态与进程状态之间的关系







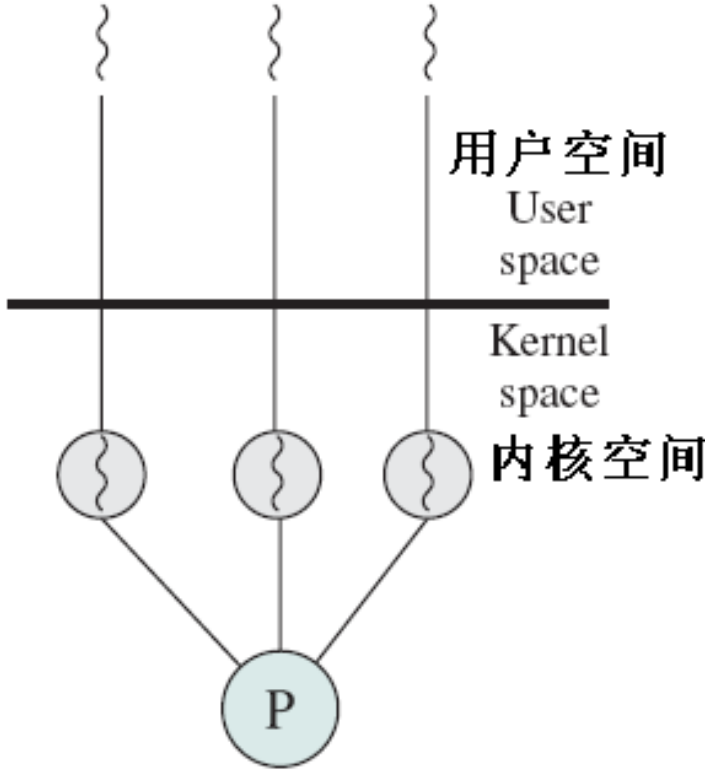


线程状态变化与进程状态变化

- 线程2中执行的应用程序代码执行了系统调用，阻塞了进程B。图b
- 系统调用完成后，时钟中断把控制权交给内核，内核确定当前正在运行的进程B的时间片用完，将进程B转为就绪态，并切换到另一个进程。图c
- 线程2执行到某处，它需要进程B的线程1所执行的某些动作的结果。线程2进入阻塞状态，线程1从就绪转换为运行，进程B自身仍保留在运行状态。图d

内核级线程(KLT)

- 线程管理由内核完成（提供API）
- 调度基于线程进行
- 实例：Windows、Linux、Mac OS X、iOS

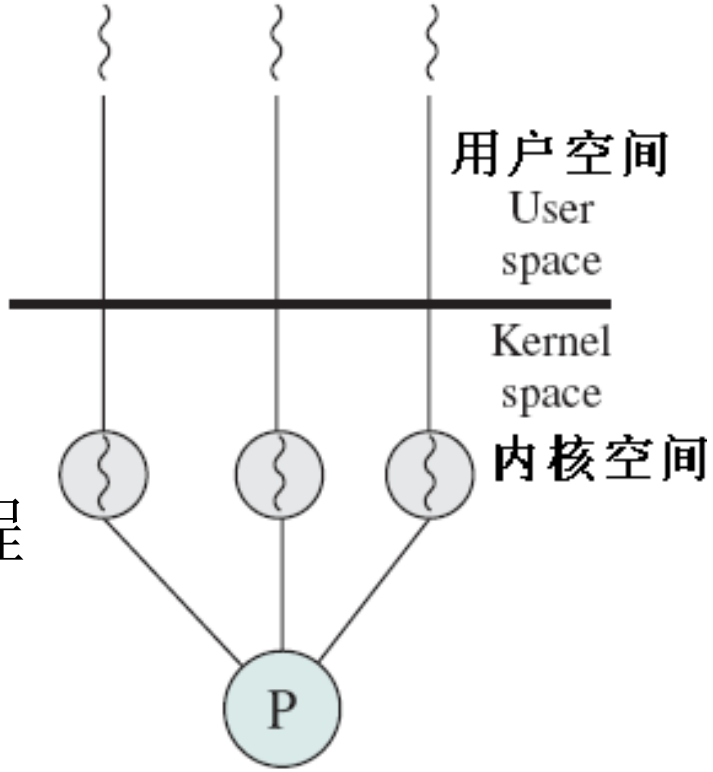


(b) Pure kernel-level

纯内核级

内核级线程(KLT)

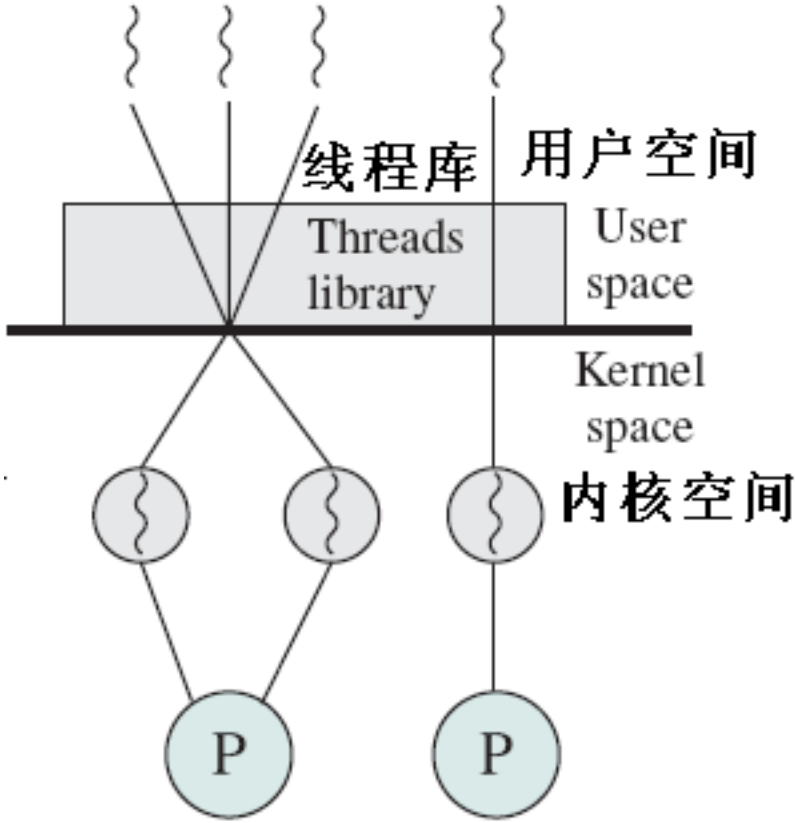
- 优点：
  - 线程阻塞不会导致进程阻塞
  - 可以利用多核和多处理器技术
  - 内核例程本身也可以使用多线程
- 缺点：
  - 线程切换需要模式切换



(b) Pure kernel-level
纯内核级

组合方法

- 线程创建在用户空间完成
- 线程调度和同步在内核空间进行
- 应用程序的m个ULT被映射到n（≤m）个KLT
- 实例：Solaris、Windows 7



(c) Combined

组合

4.2.2 其他方案

线程与进程间的关系

线程：进程	描 述	示例系统
1:1	每个进程有唯一线程	DOS、传统Unix
M:1	一个进程可拥有多个线程	Windows NT、Solaris、Linux、Mac OS X、iOS
1:M	一个线程可在多个进程环境中迁移	RS(Clouds)、Emerald
M:N	M:1+1:M	Trix

4.3 多核与多线程

- 处理器进入多核（CPU）甚至众核（GPU）阶段
  - 2013年中大信科院提供理论峰值达9.7万亿次的GPU高性能集群计算服务，集群使用了16个6核12线程的Intel至强E5645 CPU处理器、2个4核8线程的Intel至强E5620 CPU处理器、9个448核的NVIDIA Tesla C2050 GPU运算卡

4.3 多核与多线程

- 对称多处理(SMP)
  - 多个处理器可以执行相同功能（故称“对称”），内核可运行在任一处理器上
  - 每个处理器可从可用进程和线程池完成自身的调度工作

4.3.1 多核系统上的软件性能

- 阿姆德尔（Amdahl）定律

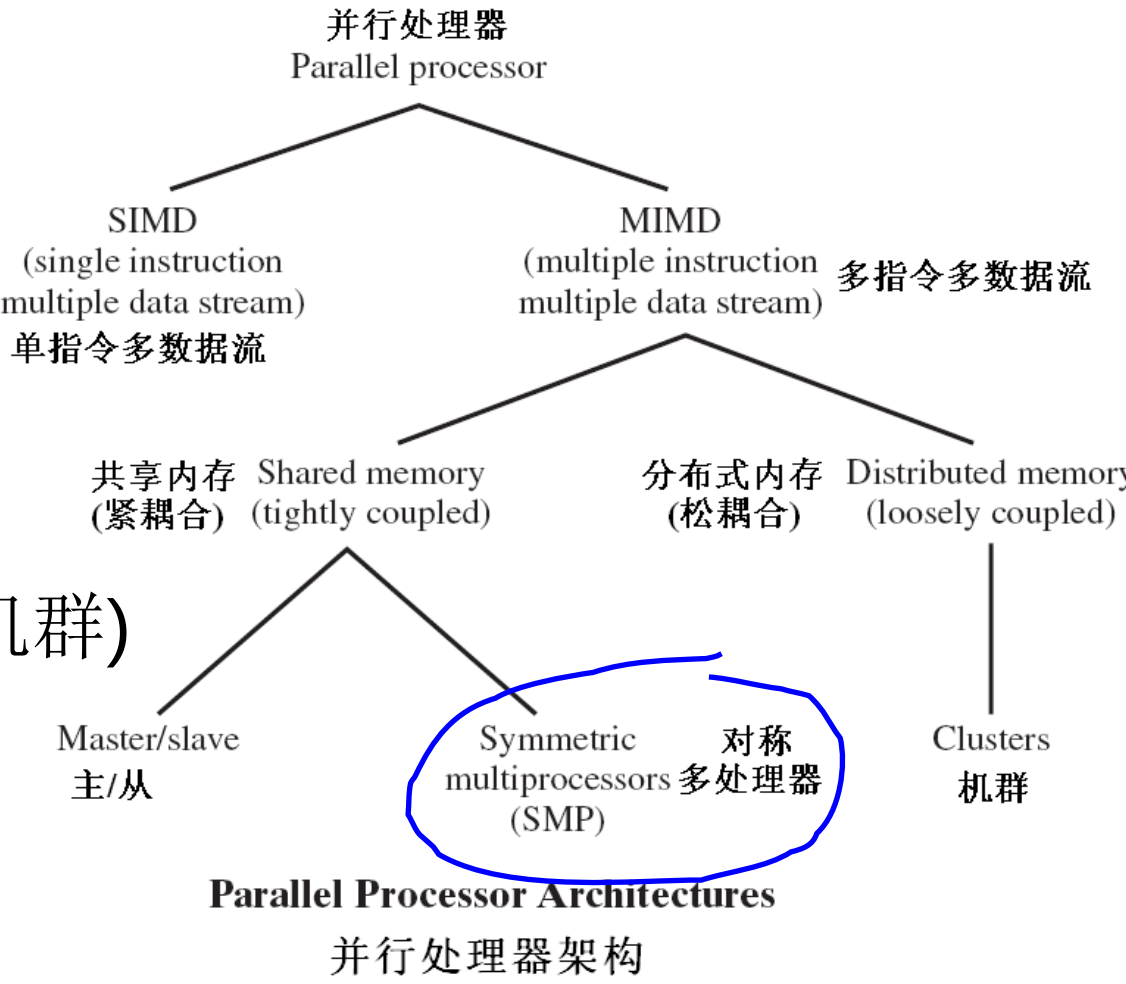
$$\text{加速比}=\frac{\text{单处理器上程序的运行时间}}{\text{在}N\text{个并行处理器上程序的运行时间}}=\frac{1}{(1-f)+\frac{f}{N}}$$

- 其中：N 为处理器/核数、f 为并行的代码占比
- $N \rightarrow \infty$，加速比 $\rightarrow 1/(1-f)$

  - f=0.9（10%代码串行执行）、N=8（8核），则加速比≈4.7
  - f=0.9、N=∞时，加速比=10

并行处理器体系结构

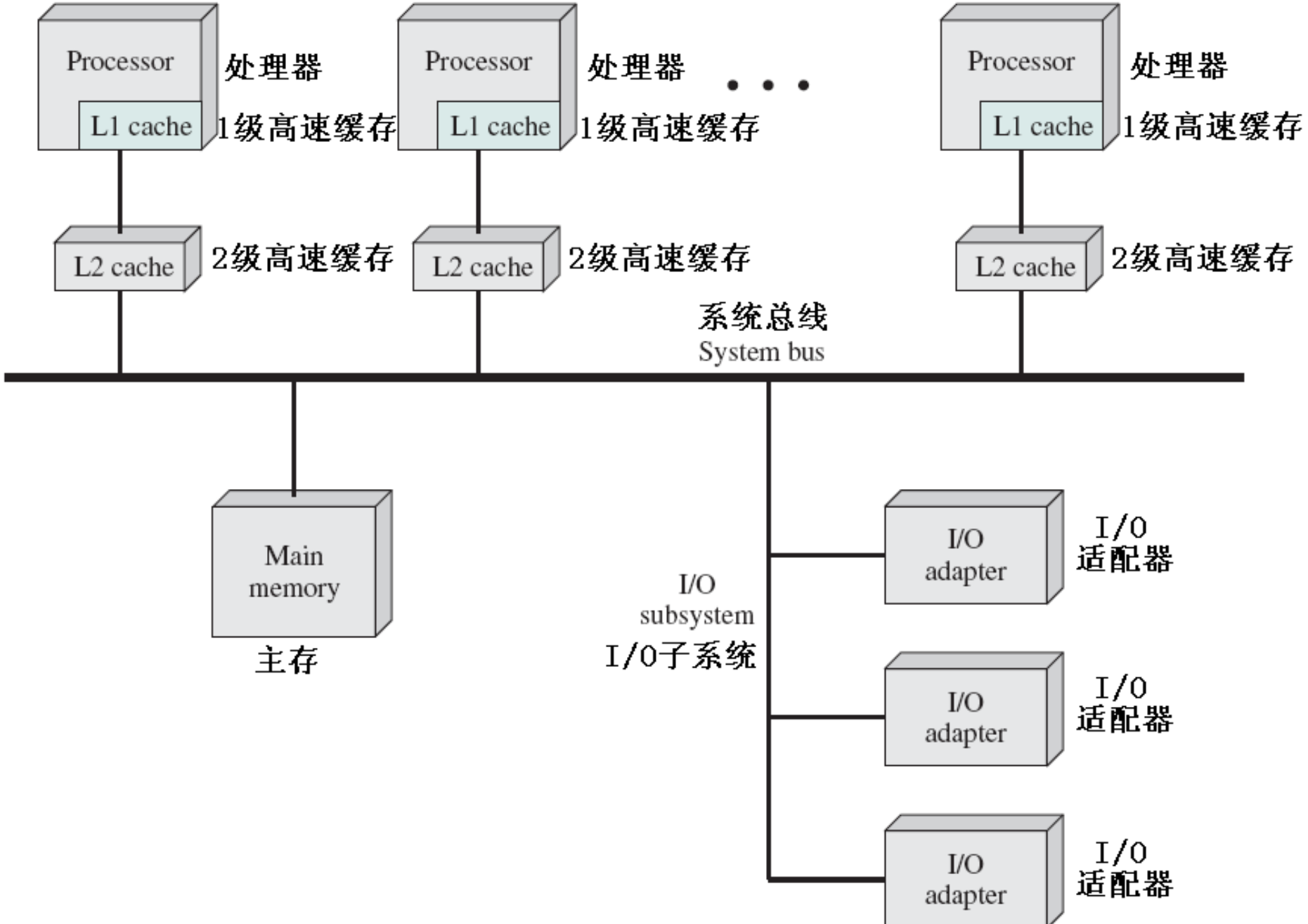
- 计算机系统分类：
  - SISD(单处理器)
  - SIMD(向量机)
  - MISD(GPU)
  - MIMD(多核/SMP/机群)



Parallel Processor Architectures

并行处理器架构

对称多处理的组织结构图

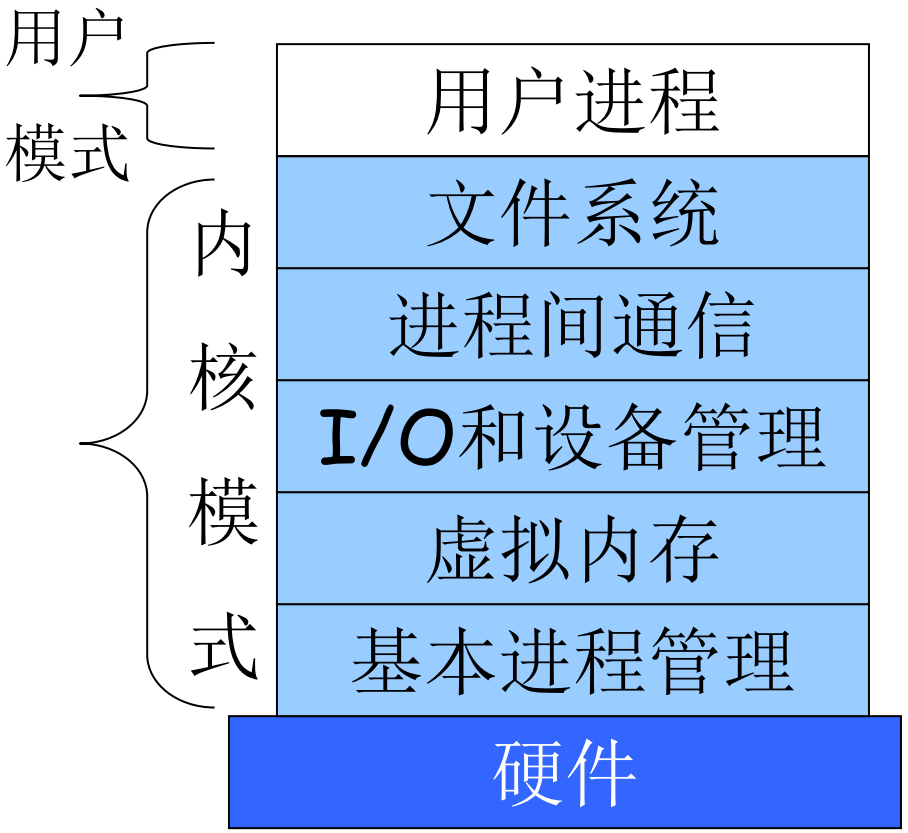


多处理器操作系统设计

- 除多道程序系统的全部功能，仍需适配附加功能
- 关键问题
  - 并发进程/线程：内核例程可重入
  - 调度：避免多处理器的调度冲突
  - 同步：控制共享资源的访问
  - 存储器管理
  - 可靠性和容错

分层内核

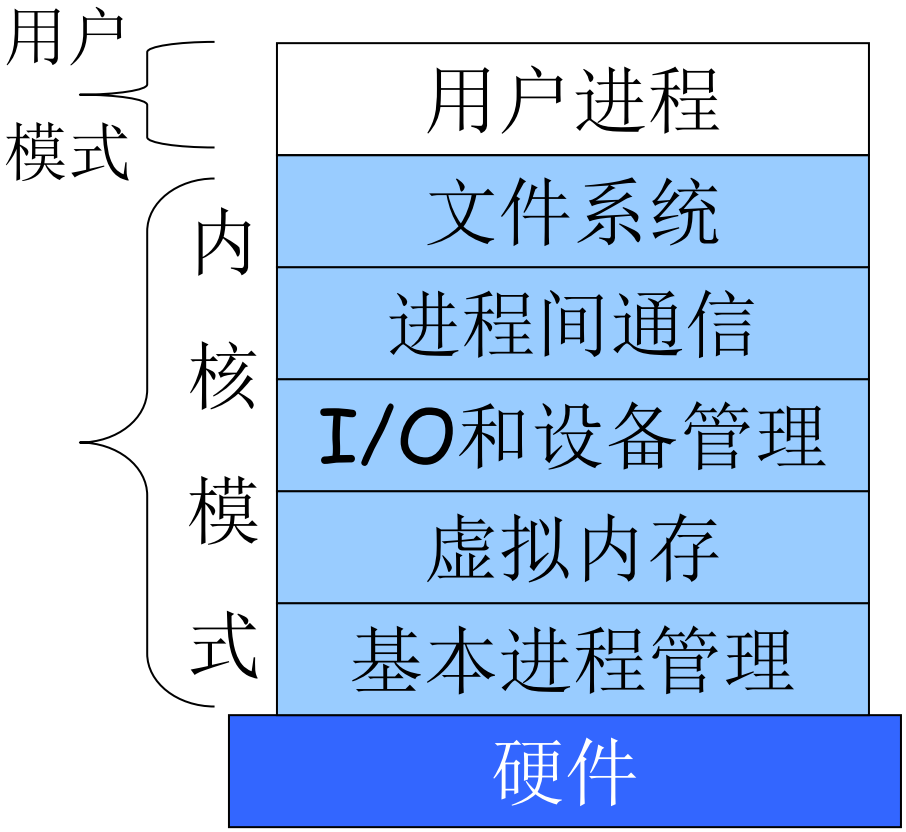
- 分层内核（宏内核/单体内核）
  - 所有功能按层次组织
  - 只在相邻层之间发生交互



分层内核

分层内核

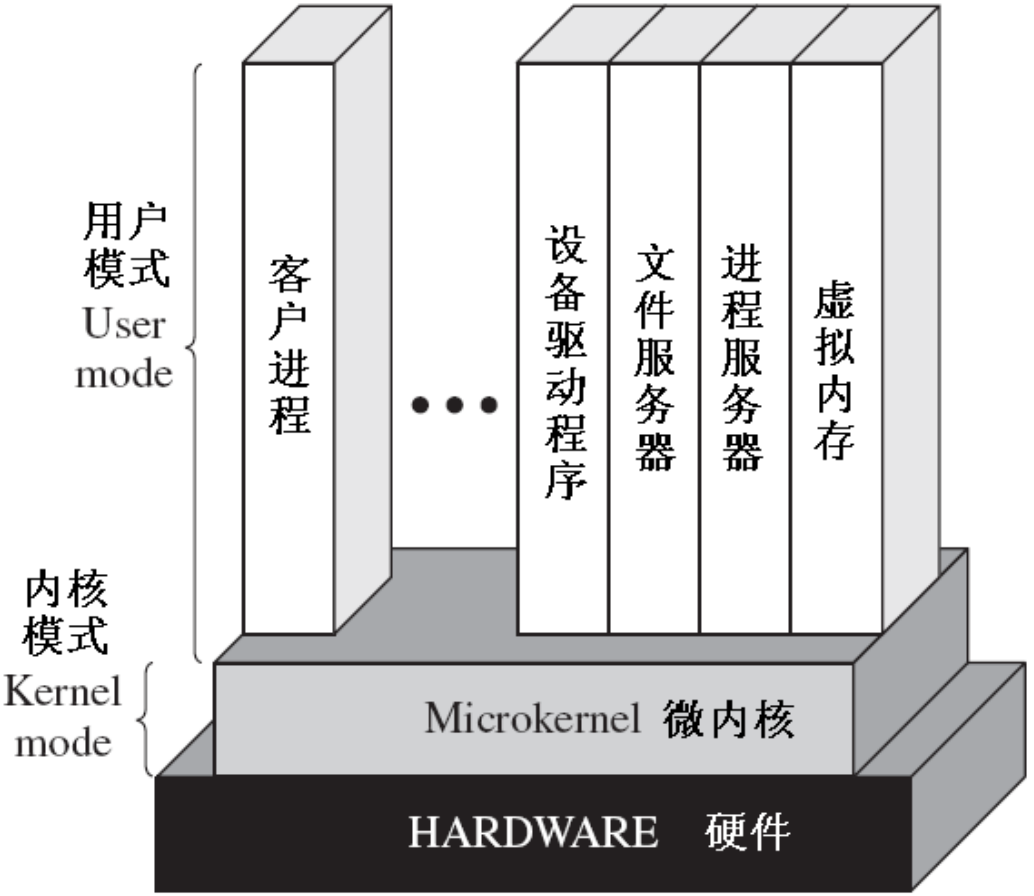
- 缺点:
  - 某一层中的主要变化可能会对相邻层中的代码产生巨大影响
  - 相邻层之间有大量交互，很难保证安全性



分层内核

微内核（续）

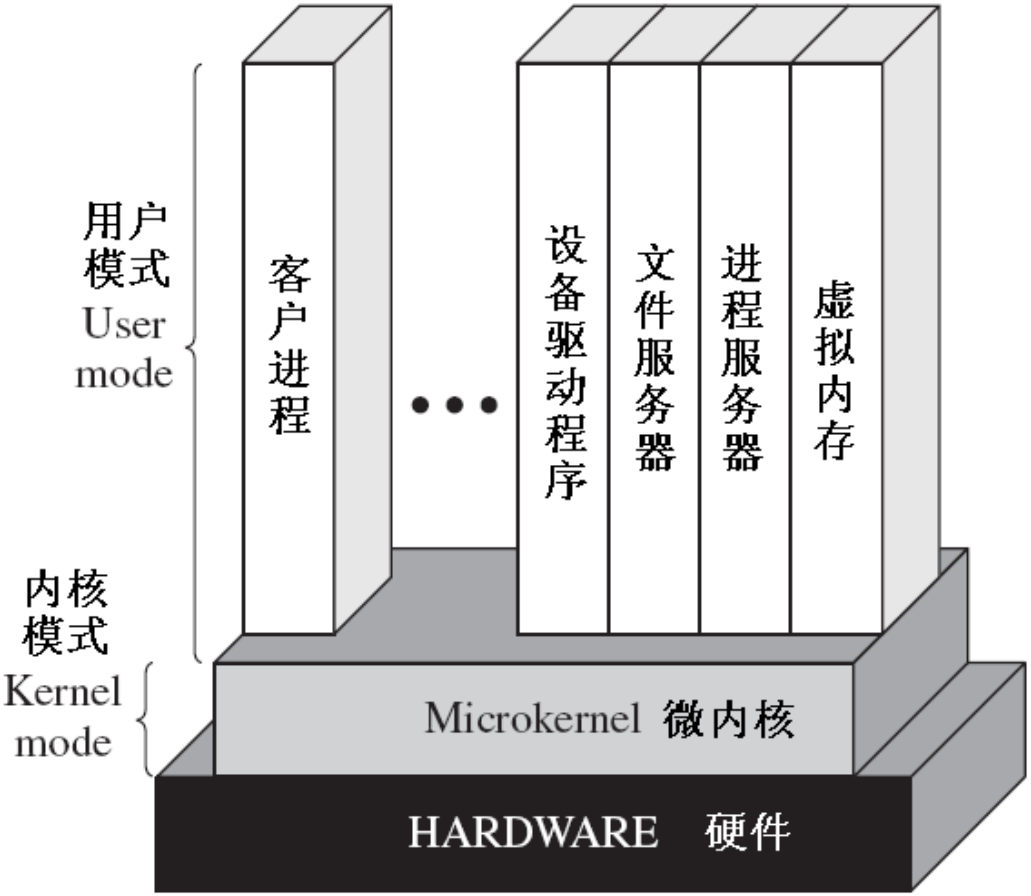
- 基本思想
  - 只有最基本操作系统和功能放在内核中，运行在内核模式下
  - 其他放在内核之外，运行在用户模式



(b) Microkernel
微内核

微内核（续）

- 基本思想
  - 只有最基本操作系统和功能放在内核中，运行在内核模式下
  - 其他放在内核之外，运行在用户模式



(b) Microkernel
微内核

垂直分层

客户/服务器结构

微内核组织结构的优点

- 一致接口
  - 所有服务都以消息的形式提供
- 可扩展性(Extensibility)
  - 允许增加新的服务
- 灵活性(Flexibility)
  - 可以增加新的功能、删除现有功能
- 可移植性(Portability)
  - 移植系统只需要对内核而不需要对其他服务修改

微内核组织结构的优点

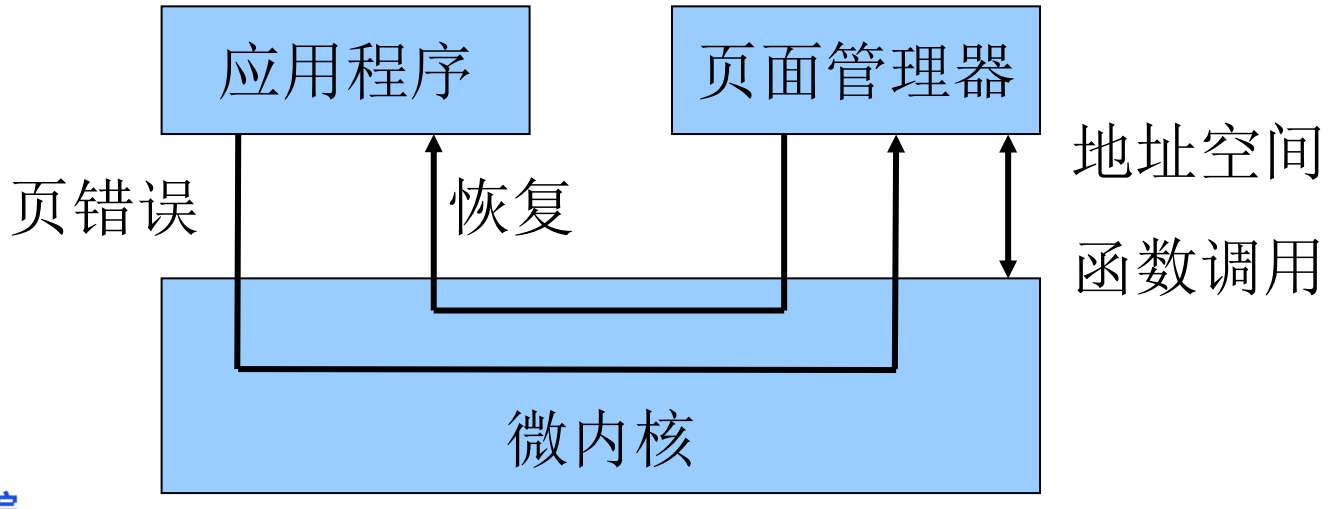
- 可靠性(Reliability)
  - 模块化设计
  - 小的微内核可以被严格地测试
- 分布式系统支持
  - 消息传送不需要知道目标机器的位置
- 对面向对象操作系统(OOOS)的支持
  - 组件是具有明确定义接口的对象，可互连构造软件

微内核设计

- 微内核功能与服务的最小集
  - 低级存储器管理
  - 进程间的通信
  - I/O和中断管理

低级存储器管理

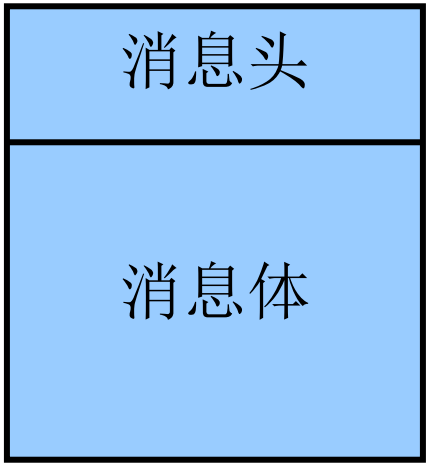
- 微内核控制硬件概念上的地址空间，使得操作系统可以在进程级实现保护
- 微内核只负责将每个虚页映射到一个物理页框上
- 内核外实现页替换算法等分页工作
  - 虚页调入
  - 页换出



三种微内核操作

- **授权**：一个地址空间的所有者可以授权另一个进程使用它的某些页
- **映射**：一个进程可以把它的任何页映射到另一个进程的地址空间，使得两个进程者都可以访问这些页
- **刷新**：进程可以回收授权给其它进程或映射到另一个进程的任何页

进程间的通信



- **消息**是进程间通信的基本形式
- **消息结构**：消息头和消息体

- **端口**：发往某个特定进程的消息序列
- **消息传递**：将消息从一个进程的地址空间内复制到另一个进程的地址空间（速度远低于处理器的），效率比共享存储方案低

I/O与中断管理

- 微内核结构中，硬件中断可能会当作消息处理
  - 微内核捕捉和识别中断，但可以不处理中断消息头
  - 产生一条消息发给相关的用户级进程，并由它进行中断处理

- 用户级进程代码通用结构

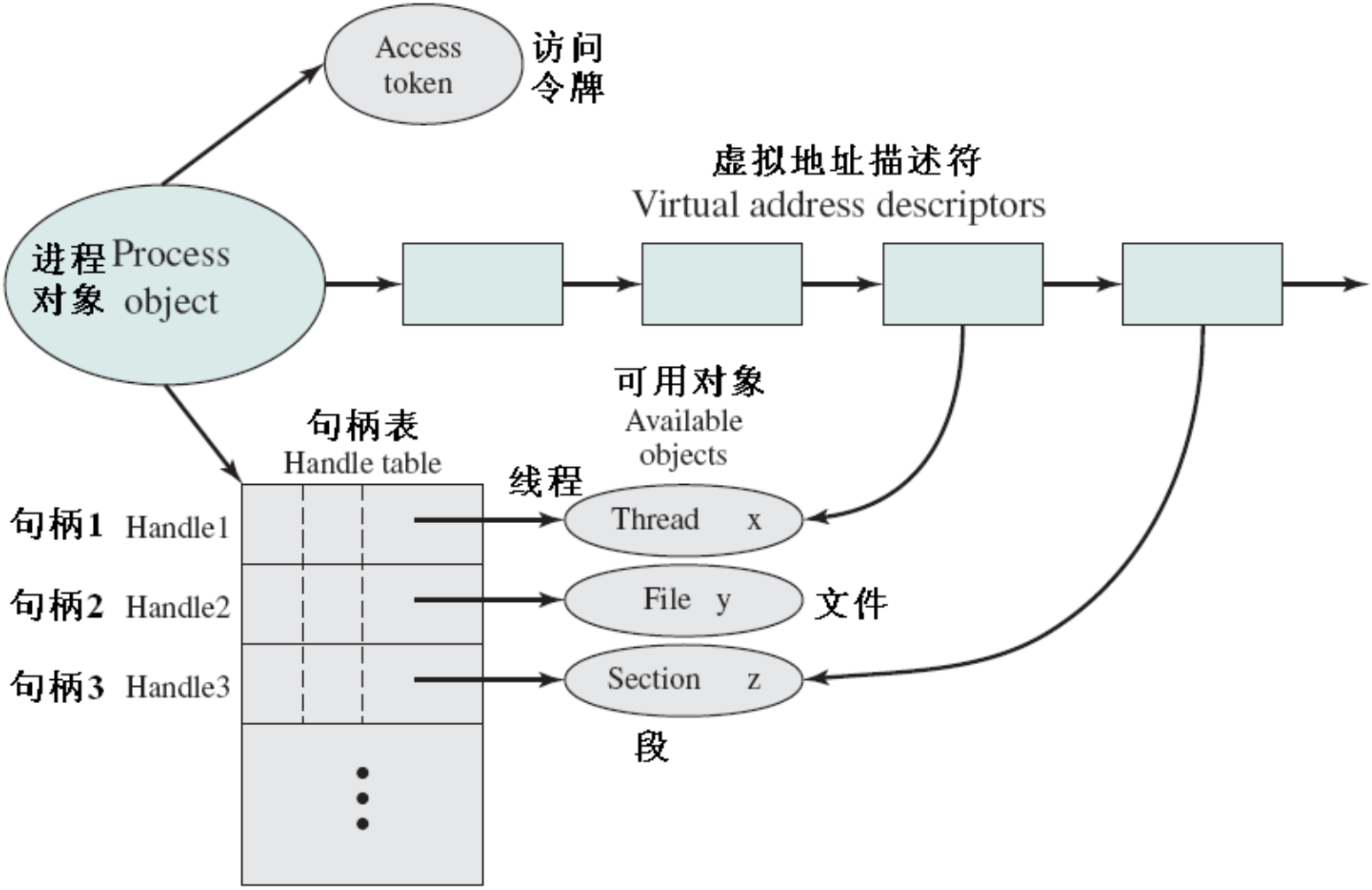
```
driver thread:
 do
   waitFor(msg,sender);
  if (sender==my_hardware_interrupt)
  {
     read/write I/O ports;
     reset hardware interrupt
   }
  else …
While (true);
```

问 答

4.4 Windows 7的线程与SMP管理

- Windows进程和线程的重要特点：
  - Windows进程/线程作为对象实现
  - 一个可执行的进程可能包含一个或多个线程
  - 进程对象和线程对象都具有内置的同步能力

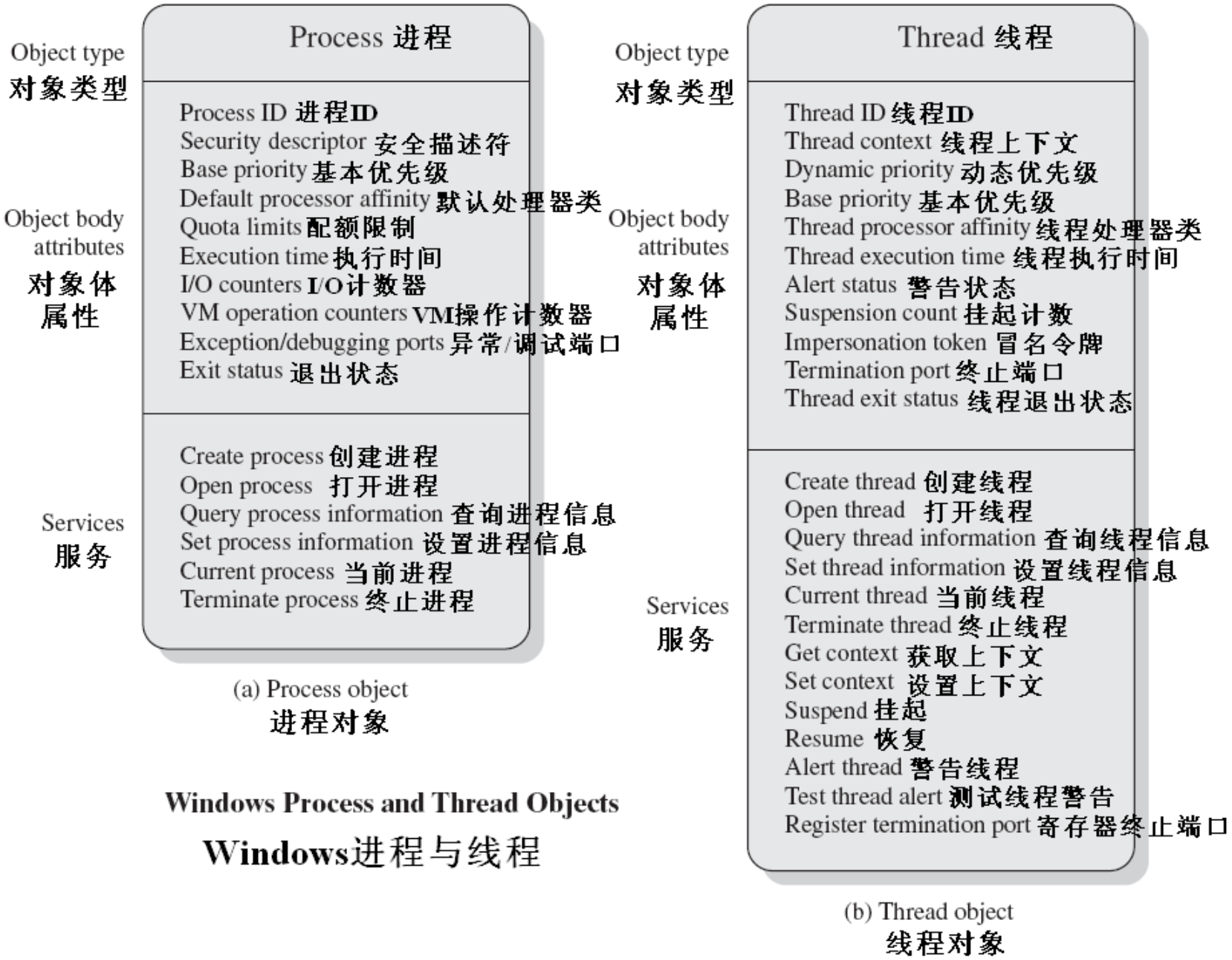
Windows进程及其资源



A Windows Process and Its Resources

一个Windows进程及其资源

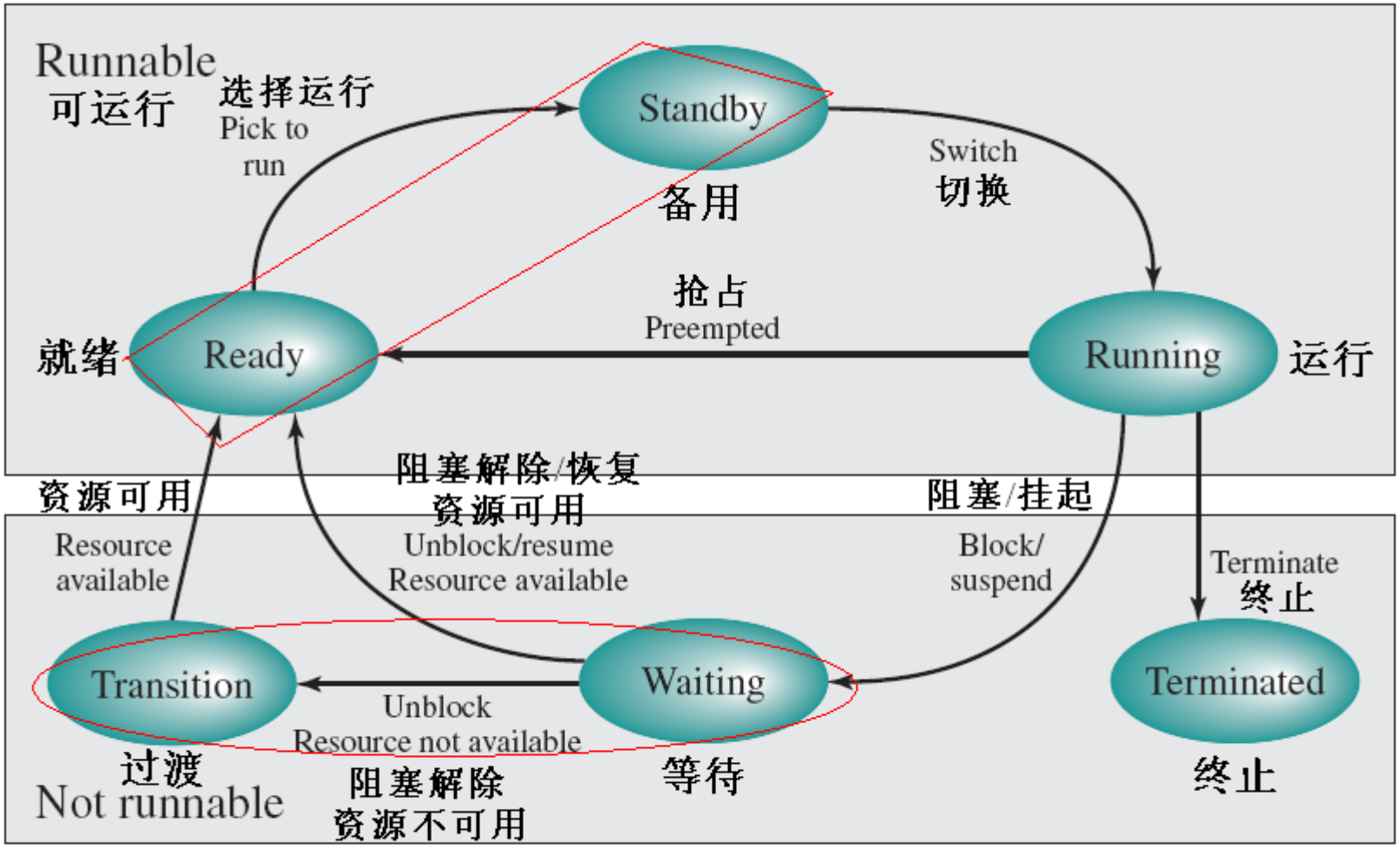
4.4.1 Windows进程和线程对象



Windows Process and Thread Objects

Windows进程与线程

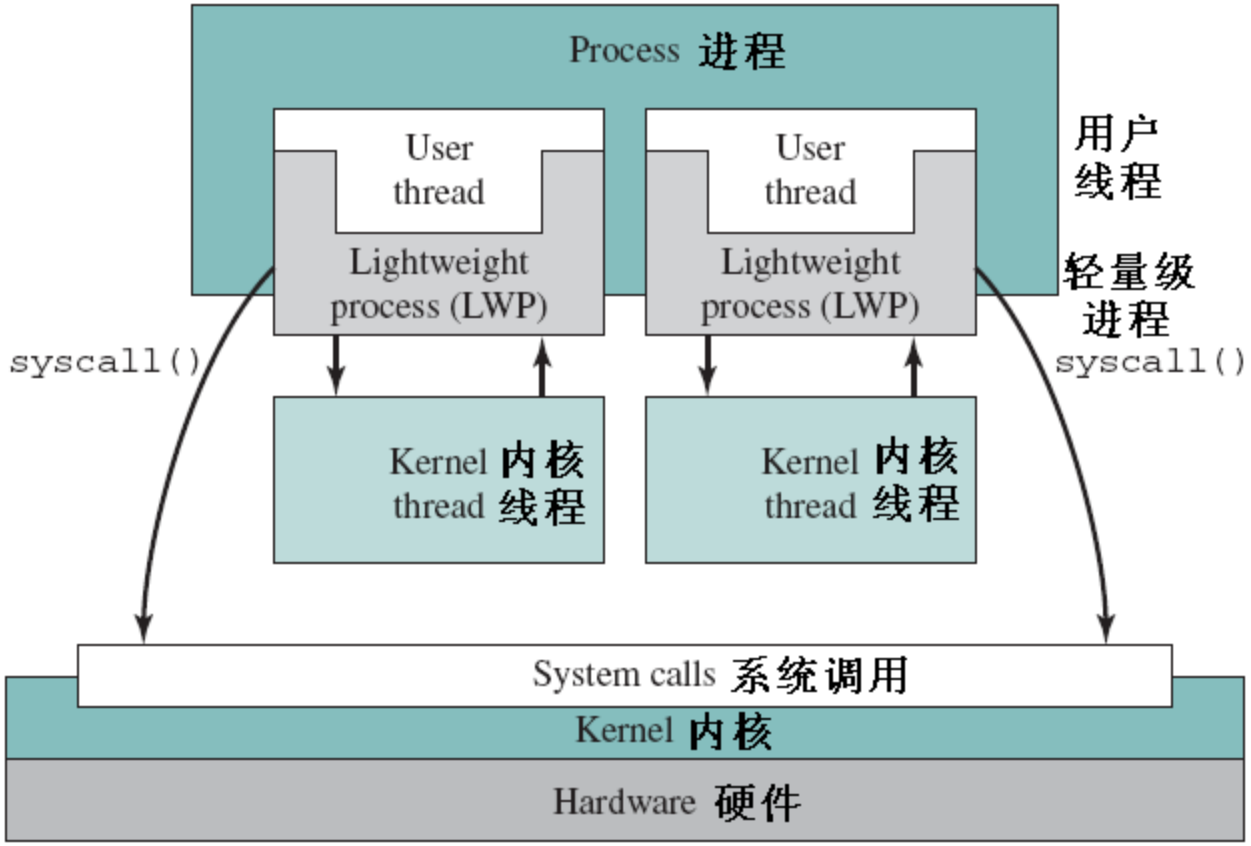
4.4.3 Windows线程状态及转换



Windows Thread States

Windows线程状态

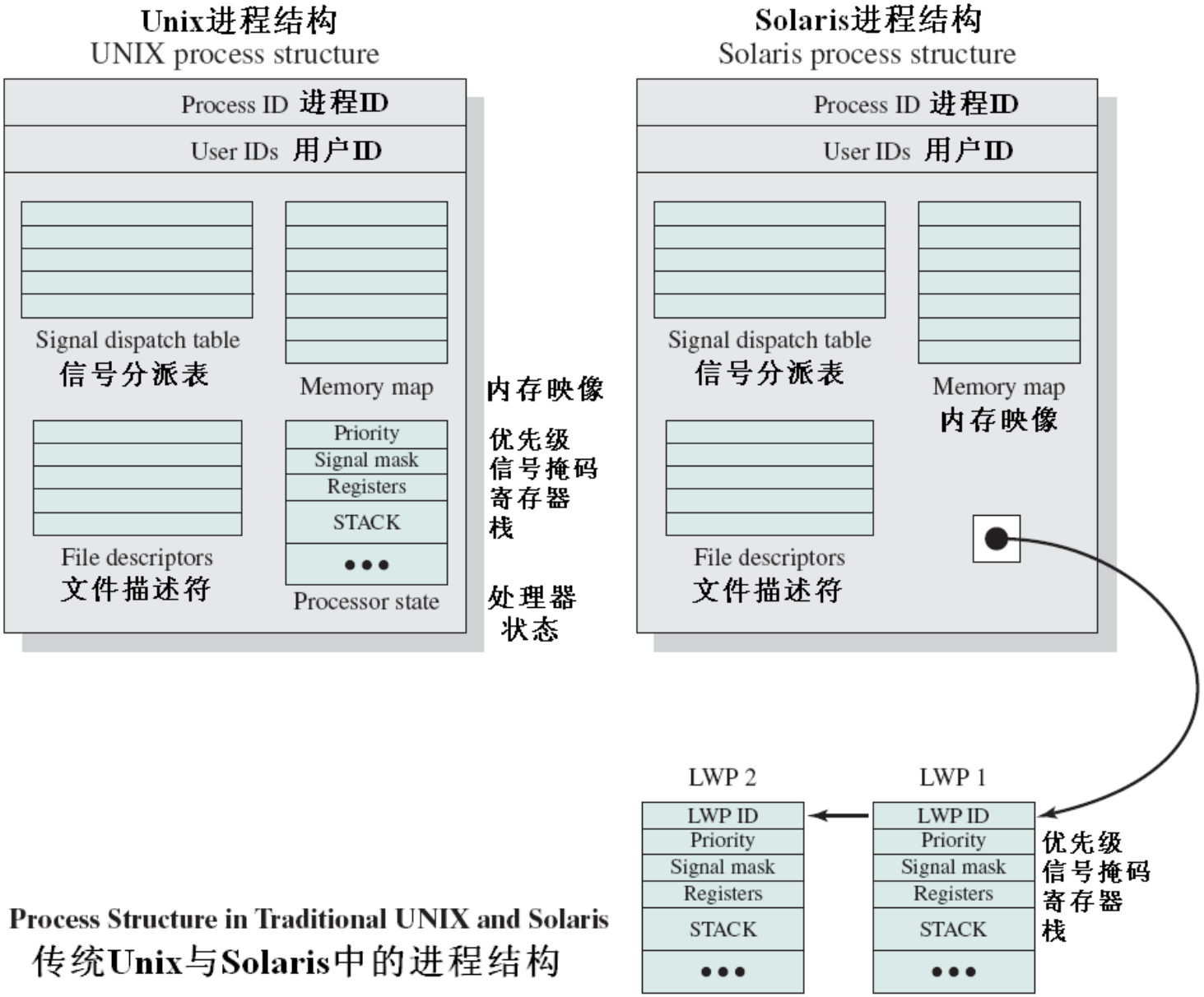
4.5 Solaris的线程和SMP管理



Processes and Threads in Solaris

Solaris中的进程与线程

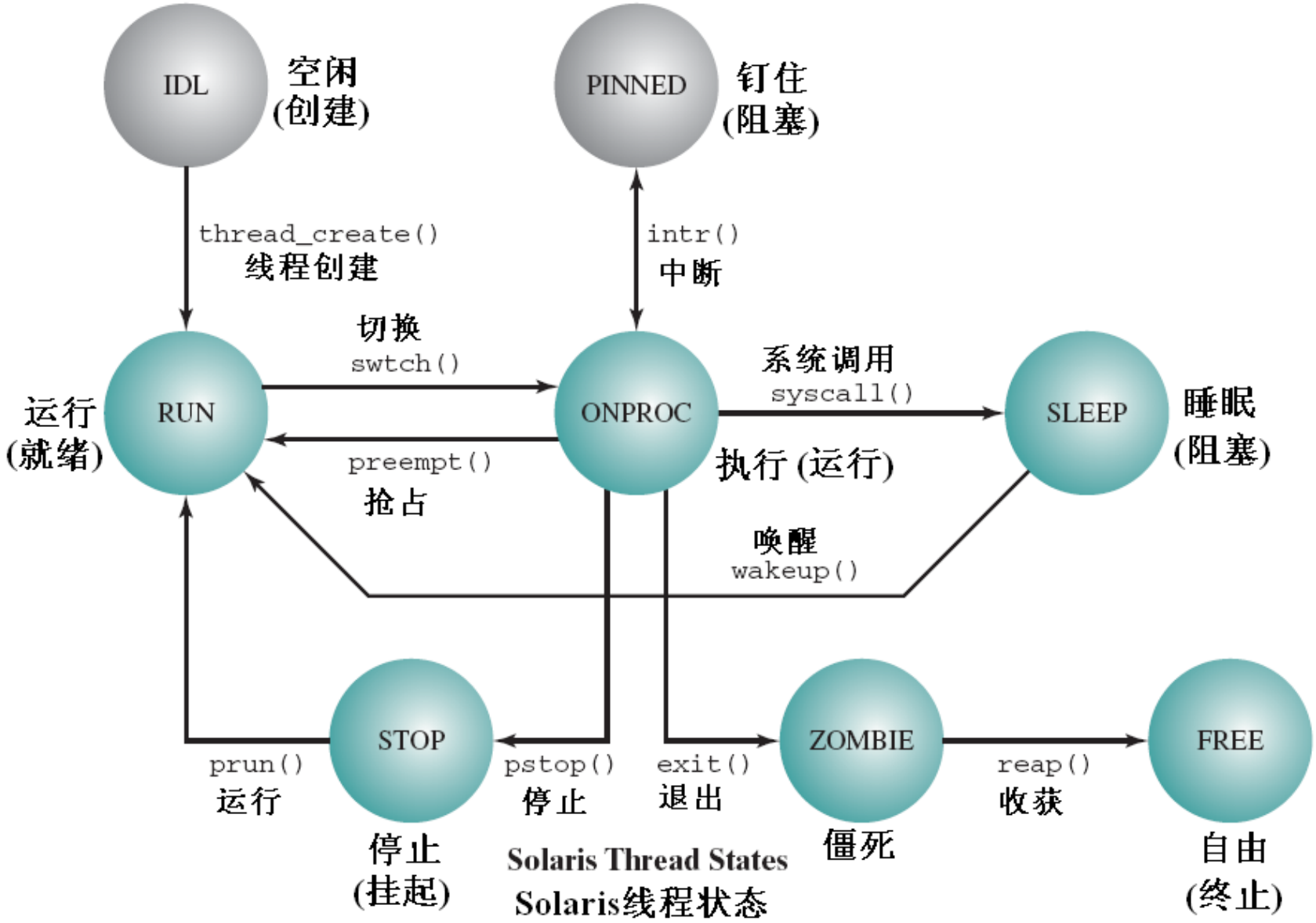
传统Unix与Solaris进程结构的比较



Process Structure in Traditional UNIX and Solaris

传统Unix与Solaris中的进程结构

Solaris的线程状态

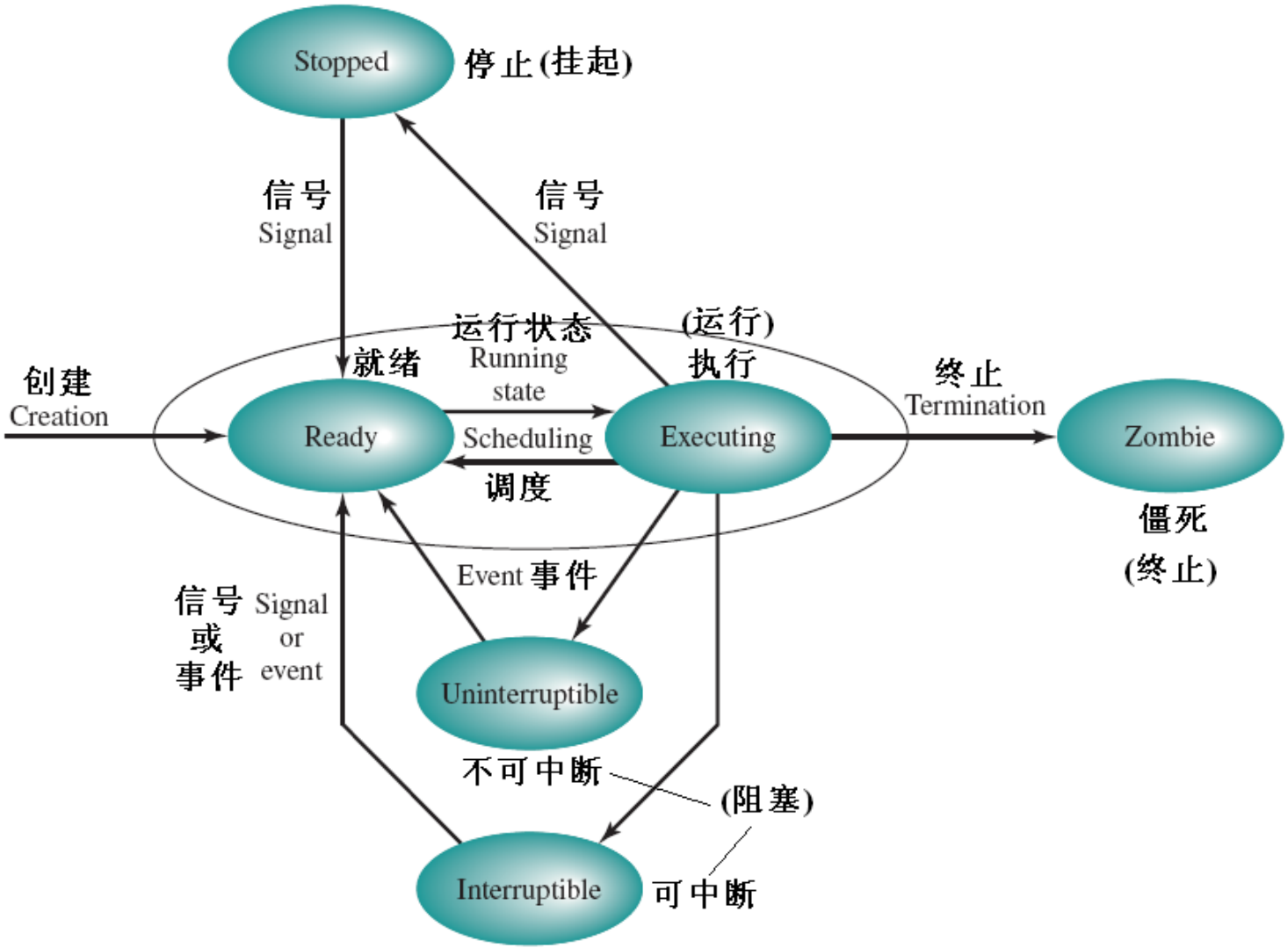


4.6 Linux的进程和线程管理

4.6.1 Linux任务

- 进程的数据结构task_struct
  - 状态：执行、就绪、挂起、停止、僵死
  - 调度信息：普通/实时、优先级、计数器
  - 标识符：进程标识符、用户标识符、组标识符
  - 进程间通信（IPC）
  - 链接：父进程链接、兄弟进程链接、子进程链接
  - 时间和计时器：进程创建时间、进程消耗处理器总时间、计时器
  - 文件系统：打开文件指针
  - 虚存：分配给该进程的虚存空间
  - 处理器专用上下文环境

Linux的进程/线程模型



Linux Process/Thread Model

Linux进程/线程模型

4.6.2 Linux线程

- Linux提供一种不区分进程和线程的解决方案
- 用户级线程被映射到内核级进程上。具体来说，组成一个用户进程的多个用户级线程被映射到共享同一组ID的多个Linux内核级进程上。这些进程可以共享文件和内存等资源，且同一级中的进程调度切换时不需要切换上下文
- Linux用clone()系统调用实现进程创建，与传统的fork()使用相同

4.7 Mac OS X和iOS的GCD技术

- GCD（Grand Central Dispatch，大中央分派，源自美国纽约市曼哈顿中城的大中央车站[Grand Central Terminal]）是Mac OS X和iOS所采用的一种多线程技术，从（2009年8月28日推出的）Mac OS X 10.6和（2010年6月21日推出的）iOS 4开始支持
- 为简化程序员（在多核系统上）的多线程编程，GCD将多线程操作交给操作系统本身，而不再是交给应用程序。GCD可根据处理器核的数量和系统当前的运行情况，自动调整应用程序的工作负荷，自动确定任务所需的线程数量，从而提高应用程序的效率
- GDC是操作系统底层的C库，通过块（block）、队列（queue）和线程池（thread pool）来实现

GCD技术（续）

- 块（block）（x=^{……}）是C、C++和Object-C等编程语言块语句（{……}）的简单扩展，程序员可以用块来封装程序中的部分代码和数据。块可以被独立调度和并发执行
- 块以队列（queue）的方式调度和分派。在程序执行过程中所遇到的块，都会被放入队列中。GCD利用队列来实现并发（concurrency）、序列化（serialization ）和回调（callbacks）。使用队列远比用手工来管理线程和锁更加有效
- GDC提供了一个可用的线程池（thread pool），操作系统会根据处理核数量和系统的线程容量，从线程池中自动取出适当数量的线程来运行队列中的块
- 除了直接对块作调度外，GCD还可让应用程序将一个块和队列与某个事件源（event source，如时钟、网络套接字、文件描述符等）关联起来。每当源产生一事件时，关联的块就会被调度执行。这样既实现了快速响应，又避免了轮询或让线程阻塞在事件源上所付出的代价
- 虽然其他操作系统也有类似的线程池，但是GCD在易用性和效率方面有了质的提升

GDC中的块、队列、线程池和事件源