2.4 线程及其实现

- 2.4.0 简单了解线程
- 2.4.1 引入多线程技术的动机
- 2.4.2 多线程环境中的进程和线程
- 2.4.3 线程的实现

河海大学 计算机与信息学院

```
1.  # include <windows.h>
2.  #include <iostream.h>
3.  DWORD WINAPI Fun1Proc(LPVOID lpParameter);

4.  void main()
5.  {
6.      HANDLE hThread1;
7.      hThread1 = CreateThread(NULL,0,Fun1Proc,NULL,0,NULL);
8.      cout << "main thread is running“ << endl;
9.      Sleep(1000); //ms
10. }

11. DWORD WINAPI Fun1Proc(LPVOID lpParameter)
12. {
13.     cout <<"thread1 is running“ <<endl;
14. }
```

通过CreateThread执行Fun1Proc，和直接调用该函数有什么区别?

```
1.  # include <windows.h>
2.  #include <iostream.h>
3.  DWORD WINAPI Fun1Proc(LPVOID lpParameter);

4.  void main()
5.  {
6.      HANDLE hThread1;
7.      hThread1 = CreateThread(NULL,0,Fun1Proc,NULL,0,NULL);
8.      cout << "main thread is running“ << endl;
9.      Sleep(1000); //ms
10. }

11. DWORD WINAPI Fun1Proc(LPVOID lpParameter)
12. {
13.     cout <<"thread1 is running“ <<endl;
14. }
```


简单了解线程

- 线程用于执行包含在进程的地址空间中的代码
- 进程作为线程的容器，提供线程运行的环境
- 单个进程可能包含若干个线程，这些线程都“同时” 执行进程地址空间中的代码
- 每个进程至少拥有一个线程，来执行进程的地址空间中的代码。当创建一个进程时，操作系统会自动创建这个进程的第一个线程，称为主线程。此后，该线程可以创建其他的线程

简单了解线程（续）

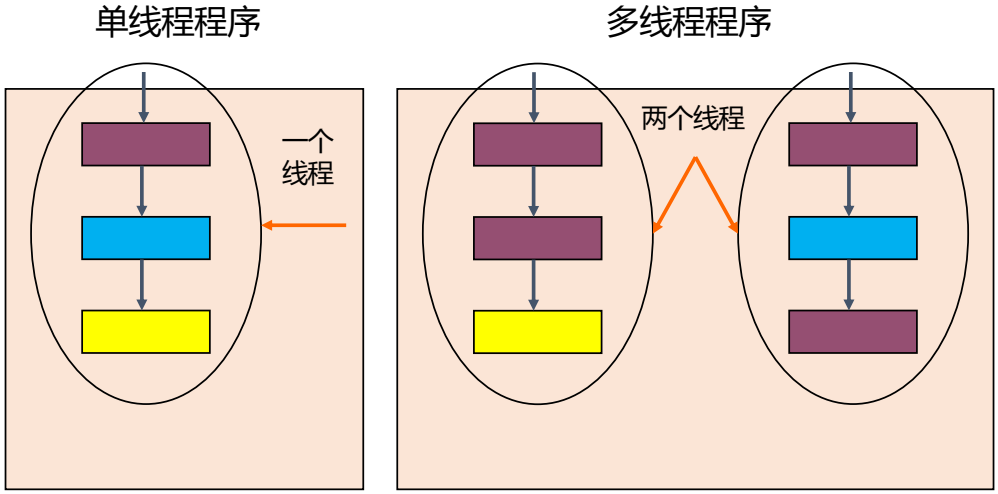
- Linux和Windows2000中线程时间片基本上在10ms左右
- 如果计算机拥有多个CPU，线程就能真正意义上同时运行了

线程也称为轻量进程
LWP（Light-Weight Process）

河海大学 计算机与信息学院

简单了解线程（续）

- Linux和Windows2000中线程时间片基本上在10ms左右
- 如果计算机拥有多个CPU，线程就能真正意义上同时运行



举例（1）

- 某大厦的建筑工程可作为一个“进程”运行，下有许多工程队，如瓦工队、木工队、水电工队、油漆工队等，每个工程队作为一个“线程”运行。
  - “进程”负责采购资源（原料）和工程管理，
  - 有原料时这些工程队可以按进度齐头并进同时工作（多线程并行执行），以加快装潢进度。
  - 缺少原料时，如缺少水泥、木料、水管、油漆之一时，相应工程队等待（线程被阻塞），而可以调度其他工程队（线程）工作

河海大学 计算机与信息学院

2.4 线程及其实现

2.4.1：单线程（结构）进程

- 在传统的操作系统中，进程是系统进行资源分配的基本单位，同时，进程也是处理器调度的基本单位，进程在任一时刻只有一个执行控制流，通常将这种结构的进程称单线程（结构）进程（Single Threaded Process）

如网络文件服务功能的应用，若采用单线程编程方法，需要循环检查网络的连接、磁盘驱动器的状况，并在适当的时候显示这些数据，必须等到一遍查询后才能刷新数据的显示，对使用者来说，延迟可能很长

2.4.1：单线程进程（续）

- 而一个多线程的应用可以把这些任务分给多个线程
  - 一个线程可检查网络
  - 另一个线程管理磁盘驱动器
  - 还有一个线程负责显示数据
  - 三个线程结合起来共同完成文件服务的功能，使用者也可以及时看到网络的变化

2.4.1：单线程进程（续）

- 单线程结构进程给并发程序设计效率带来问题
  - **进程切换开销大，特别是涉及到内外存交换时**
  - 进程通信代价大，需要通过内核切换进程来实现数据共享
  - 并发度不高，过多的进程切换和通信延迟使得细粒度并发得不偿失

2.4.1：基本思路

- 把进程的两项功能 ——

  “独立分配资源”与“被调度分派执行”分离开来

- 进程作为系统资源分配和保护的独立单位，无需频繁切换
- 线程作为系统调度和分派的基本单位，能轻装运行，会被频繁地调度和切换
- 在这种指导思想下，产生了线程的概念

2.4.1：多线程（结构）进程

- 并发多线程程序设计
  - 在同一进程中设计出多条控制流（每一个控制流称为一个线程），多条控制流之间可以并行执行
  - 多控制流切换不需要通过进程调度
  - 多控制流之间可以通过内存区直接通信，降低通信开销

2.4.1：总结

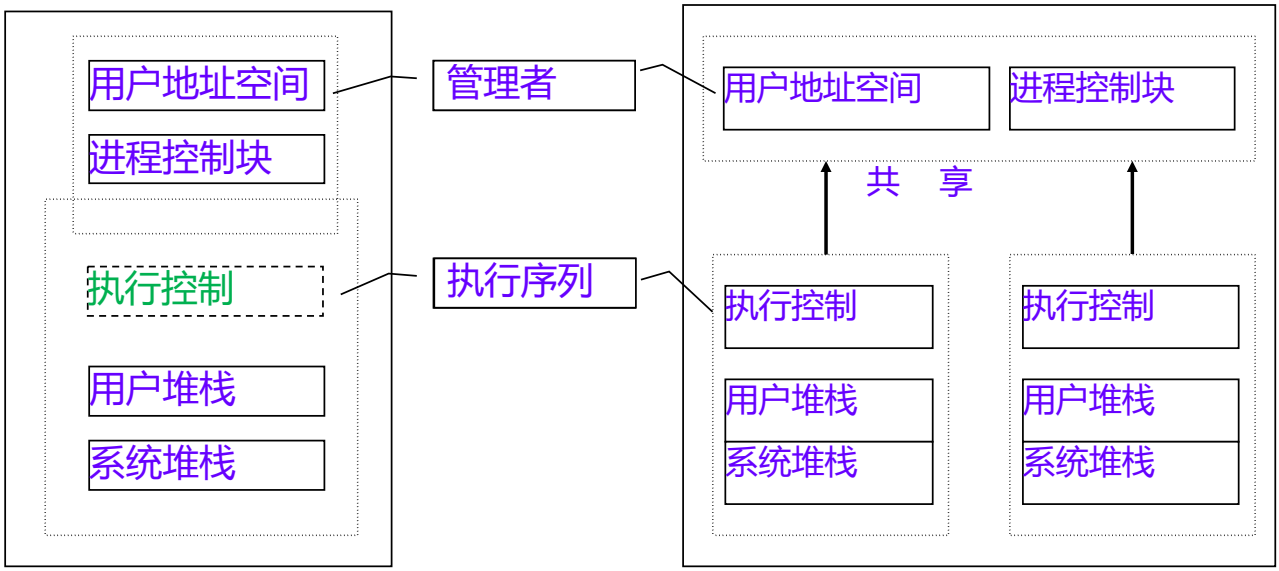
- 引入进程的目的：
  - 为了使多个程序并发执行，以改善资源使用率和提高系统效率
- 引入线程的目的：
  - 降低程序并发执行时的时空开销，使得并发粒度更细、并发性更好

2.4 线程及其实现

河海大学 计算机与信息学院

2.4.2：多线程环境中的进程

- 在单线程进程中：
  - 进程和线程概念可以不加区别
  - 一个进程的结构可以划分成两个部分：
    - 对资源的管理
    - 实际的指令执行序列
- 在多线程进程中，每个线程具有：
  - 独立堆栈
  - 现场信息
  - 其它状态信息与线程控制块TCB（Thread Control Block）
  - 一个进程中的所有线程共享其所属进程拥有的资源，它们驻留在相同的地址空间，可以存取相同的数据



进 程

单线程进程的内存布局和运行

进 程

管理和执行相分离的进程模型

2018/10/15

河海大学 计算机与信息学院

2.4.2：多线程中的进程（续）

- 进程是操作系统中进行保护和资源分配的基本单位，具有：
  - 一个虚拟地址空间，用来容纳进程的映像
  - 对处理器、其他(通信的)进程、文件和I/O资源等有控制有保护的访问
- 传统进程原先所承担的控制流执行任务交给称作线程的部分完成

2.4.2：多线程中的进程属性

- 作为系统资源分配的单位：进程包括用户地址空间、用于实现同步和通信的机制、已打开的文件和已申请到的I/O设备以及一张由核心进程维护的地址映射表
- 单个进程包含了多个线程
- 进程不是一个可执行的实体，但仍具有与执行相关的状态
- 对进程所施加的与进程状态相关的操作对其线程也起作用（阻塞、唤醒、挂起、激活）

2.4.2：多线程环境中的线程

- 线程是操作系统进程中能够独立执行的实体（控制流）
  - 是处理器调度和分派的基本单位
  - 是进程的组成部分，每个进程内允许包含多个并发执行的实体（控制流），这就是多线程
  - 同一进程中的所有线程共享进程的主存空间和资源，但不拥有资源

2.3.1（续）：进程的属性（vs. 程序）

1） 结构性

- 进程包含了数据集合和运行于其上的程序。每个进程至少包含三个组成要素：程序块、数据块和进程控制块

2） 共享性

- 同一程序运行于不同数据集合上时，构成不同的进程。多个不同的进程可以共享相同的程序，所以进程和程序不是一一对应的
- 进程之间可以共享某些公用变量
- 进程的运行环境不再是封闭的

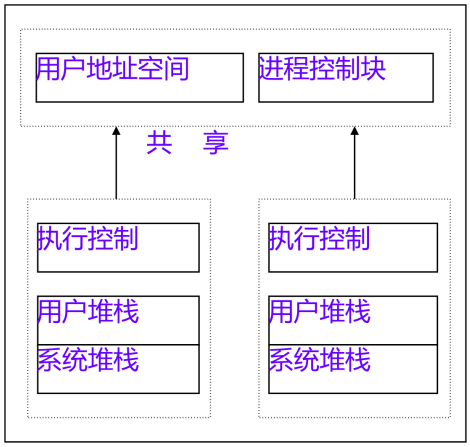
河海大学 计算机与信息学院

2.4.2：多线程环境中的线程属性

- 结构性：线程具有唯一的标识符和线程控制块，其中包含调度所需的一切私有信息
- 共享性：同一进程中的所有线程共享但不拥有进程的状态和资源，且驻留在进程的同一个主存地址空间中，可以访问相同的数据

2.4.2：多线程中的线程组成

- 每个线程拥有受保护的线程上下文
  - 一个独立的程序指令计数器
  - 一个执行堆栈
  - 一个容纳局部变量的主存存储区



当线程不运行时，用于存储现场信息

2.3.1（续）：进程的属性（vs. 程序）

3） 动态性
- 进程由创建而产生，由调度而执行，由撤销而消亡
- 程序是一组有序指令序列，作为一种系统资源是永久存在的

4） 独立性
- 进程是系统中资源分配和保护的基本单位，也是系统调度的独立单位（单线程进程）

2.4.2：多线程环境中的线程属性

- 动态性：线程是程序在相应数据集上的一次执行过程，由创建而产生，直到由撤消而消亡，有其生命周期。每个进程被创建时，至少同时为其创建一个线程，需要时线程可以再创建其它线程
- 并行性：同一进程的多个线程可在一个/多个处理器上并发或并行地执行，而进程之间的并发执行演变为不同进程的线程之间的并发执行

2.4.2：多线程中的进程vs. 线程

- 进程可以划分为两个部分：资源集合和线程集合
- 进程封装了管理信息，进程要支撑线程运行，为线程提供地址空间和各种资源。包括对指令代码、全局数据和I/O状态数据等共享部分的管理
- 线程封装了执行信息，包括对CPU寄存器、执行栈（用户栈、内核栈）和局部变量、过程调用参数、返回值等线程私有部分的管理

2.4.2：并发多线程设计优点

- 并发多线程程序设计的优点体现在：
  - 进程：具有独立的虚地址空间，以进程为单位进行任务调度，系统必须交换（虚）地址空间，切换时间长
  - 线程：同一进程中的多线程共享同一地址空间，能使快速切换

- 减少（系统）管理开销
- 线程通信易于实现
- 并行程度提高
- (相对于多进程)节省内存空间

2.4.2：线程状态

- 由于进程不是调度单位，不必将进程状态划分过细
  - 如Windows操作系统中仅把进程分成可运行和不可运行状态
  - 挂起状态属于不可运行状态

在多线程进程环境下，进程不再是调度单位，线程才是

2.4.2：线程状态

- 线程的状态有：运行、就绪和阻塞，线程的状态转换也类似于进程
  - 挂起状态对线程是没有意义的
  - 挂起状态是进程级状态，不作为线程级状态
  - 类似地，进程的终止会导致所有线程的终止

河海大学 计算机与信息学院

2.4.2：线程状态

- 线程的两种阻塞方式：
  - 阻塞进程方式：当一个线程被阻塞，所在进程也转换为阻塞态，即使这个进程存在另一个处于就绪态的线程
  - 阻塞线程方式：当一个线程被阻塞，如果存在另外一个处于就绪态的线程，则调度该线程进入运行状态，否则进程才转换为阻塞态

2.4.2：线程状态

- 以阻塞进程方式，假设进程B正在执行线程3，可能出现下列情况：
  - Eg.1：进程B的线程3发出一个封锁B的系统调用(如I/O操作)，这导致控制转移到内核，内核启动I/O操作，将进程B置为阻塞状态，并切换到另一个进程。
    - 由于由线程库维护线程状态，此时进程B的线程3仍处在运行态，进程B中的其他线程处于可运行的就绪态，但因进程B被阻塞而也都被阻塞

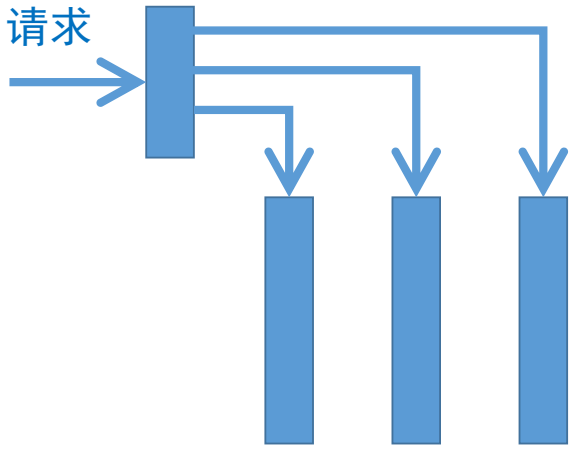
河海大学 计算机与信息学院

2.4.2：线程状态

- Eg.2：一个时钟中断传送控制给内核，内核中止当前时间片用完的进程B，并把它放入就绪队列，切换到另一个就绪进程。此时，进程B的线程3仍处于运行态，进程B却已处于就绪态

- Eg.3：线程3执行到某处，它需要进程B的线程1的某些操作，于是让线程3变成阻塞态，而线程1从就绪态转为运行态，进程始终处于运行态

2.4.2：线程的组织方式（1）

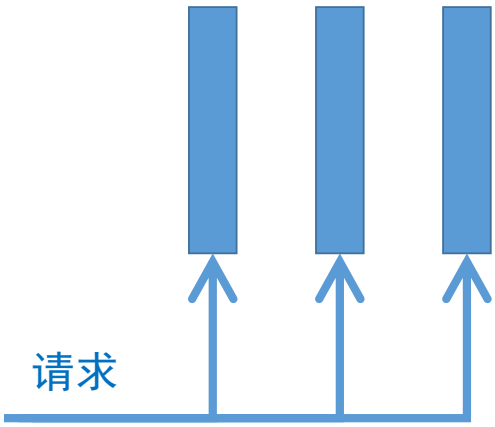


- 调度员 / 工作者模式：
  - 进程中的一个线程担任调度员的角色接受和处理工作请求，其他线程为工作者线程。由调度员线程分配任务和唤醒工作者线程工作

调度者的线程读取工作请求，稍做检查后选择一个空闲的工作者线程，将请求交给它并唤醒该工作者

（C/S模式，数据库线程池）

2.4.2：线程的组织方式（2）

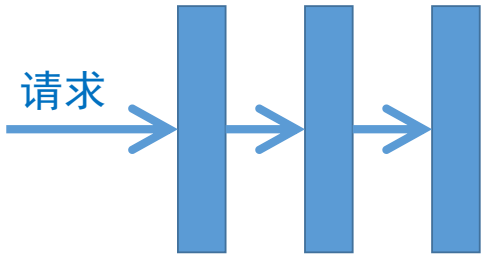


- 组模式：
  - 进程中的各个线程都可以取得并处理该请求，不存在调度者线程。有时每个线程被设计成专门处理特定的任务，同时建立相应的任务队列

所有的线程都是平等的，将等待处理的工作放入一个工作队列，有时每个线程都设计成专门处理某种特定的工作

（异步、并发的多线程执行）

2.4.2：线程的组织方式（3）



- 流水线模式：
  - 线程排成一个次序，第一个线程产生的数据传送给下一个线程处理，依次类推

第一个线程产生一些数据，并传给下一个线程去处理，数据沿着线程依次传递

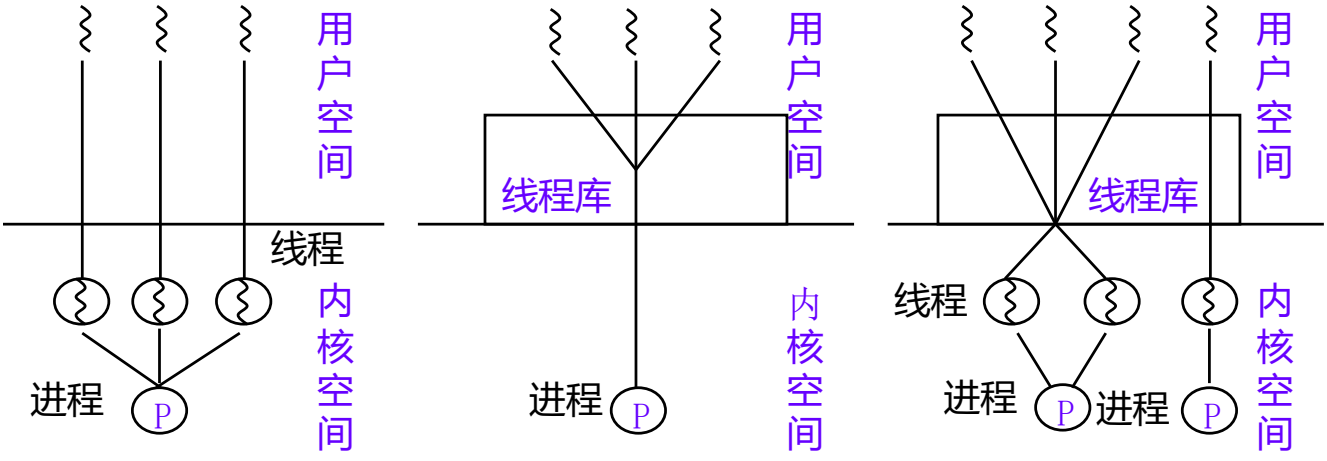
（生产者/消费者问题）

2.4 线程及其实现

2.4.3 线程的实现

- 从实现的角度看，线程可以分成：
  - 内核级线程 KLT(如Windows2000/XP、OS/2)，分别在用户空间和内核空间实现
  - 用户级线程 ULT(如Java，POSIX)
  - 混合式线程(如Solaris)，同时支持两种线程实现

2.4.3 线程的实现（续）



1）内核级线程　2）用户级线程　3）混合式线程

各种线程实现方法

2.4.3：内核级线程：实现原理

- 在纯内核级线程设施中，线程管理的所有工作由操作系统内核做。内核专门提供KLT 应用程序设计接口API供开发者使用，应用程序区不需要有线程管理代码。
- 内核要为整个进程及进程中的单个线程维护现场信息，应在内核中建立和维护PCB及TCB,内核的调度是在线程的基础上进行的
- Windows NT 和 OS/2都是采用这种方法的例子。线程执行中可通过内核创建线程原语来创建其他线程，这个应用的所有线程均在一个进程中获得支持。

2.4.3：内核级线程：优缺点

- 多处理器上，内核能同时调度同一进程中多个线程并行执行进程中的一个线程被阻塞了，内核能调度同一进程的其它线程或其他进程中的线程占有处理器运行
- 内核线程数据结构和堆栈很小，KLT切换快，内核自身也可用多线程技术实现，能提高系统的执行速度和效率
- 应用程序线程在用户态运行，而线程调度和管理在内核实现，在同一进程中，控制权从一个线程传送到另一个线程时需要用户态-内核态-用户态的模式切换，系统开销较大

2.4.3：用户级线程：实现原理

- 在纯ULT环境中，线程管理工作由应用程序做，在用户空间实现，内核不知道线程的存在。任何应用程序均需通过线程库进行程序设计，再与线程库连接后运行来实现多线程。
- 线程库：一个ULT管理的例行程序包，是线程的运行支撑环境

2.4.3：用户级线程：实现原理（续）

- ULT线程“孵化”过程：
  - 进程开始只有一个线程，由线程库为新线程创建一个TCB，并置为就绪态。
  - 按一定的调度算法把控制权传递给进程中处于就绪态的一个线程。
  - 当控制权传送到线程库时，当前线程的现场信息应被保存。
  - 当线程库调度一个线程执行时，要恢复它的现场信息
- 上述活动均发生在用户空间，且在单个进程中。内核并不知道这些活动。内核按进程为单位调度，并赋予一个进程状态（就绪、运行、阻塞. . .）

2.4.3：用户级线程：优点

- 线程切换不需要内核特权方式
  - 因为所有线程管理数据结构均在单个进程的用户空间中，管理线程切换的线程库也在用户地址空间中运行，因而进程不需要切换到内核方式来做线程管理
- 按应用特定需要允许进程选择调度算法
  - 线程库的线程调度算法与操作系统的低级调度算法是无关的
- ULT能运行在任何OS上，内核在支持ULT方面不需要做任何改变。
- 线程库是可被所有应用共享的应用级实用程序，许多当代操作系统和语言均提供了线程库

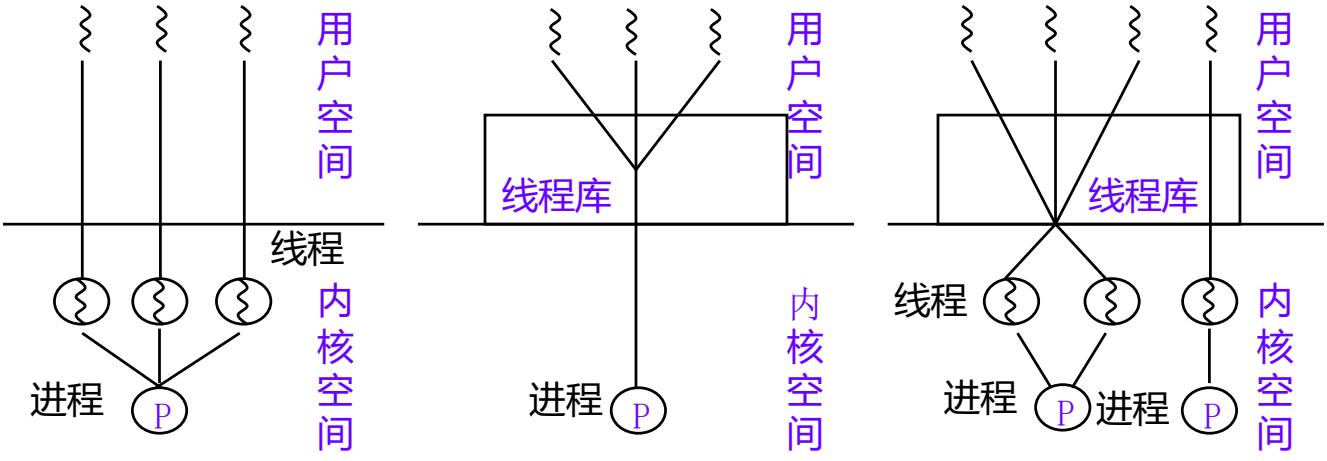
2.4.3：用户级线程：缺点

- 线程执行系统调用被阻塞时，不仅该线程被阻塞，且进程内的所有线程会被阻塞
- 纯ULT中，多线程应用不能利用多重处理的优点。内核在一段时间里，分配一个进程仅占用一个CPU，进程中仅有一个线程能执行

2.4.3：混合式线程

- 在混合式系统中，内核支持KLT多线程的建立、调度和管理，同时也提供线程库，允许应用程序建立、调度和管理ULT。应用程序的多个ULT映射成一些KLT，程序员可按应用需要和机器配置调整KLT数目，以达到较好效果
- 在混合式系统中，一个应用中的多个线程能同时在多处理器上并行运行，且阻塞一个线程时并不需要封锁整个进程

2.4.3 线程的实现（续）



1）内核级线程　2）用户级线程　3）混合式线程

ULT　KLT　P Process

各种线程实现方法