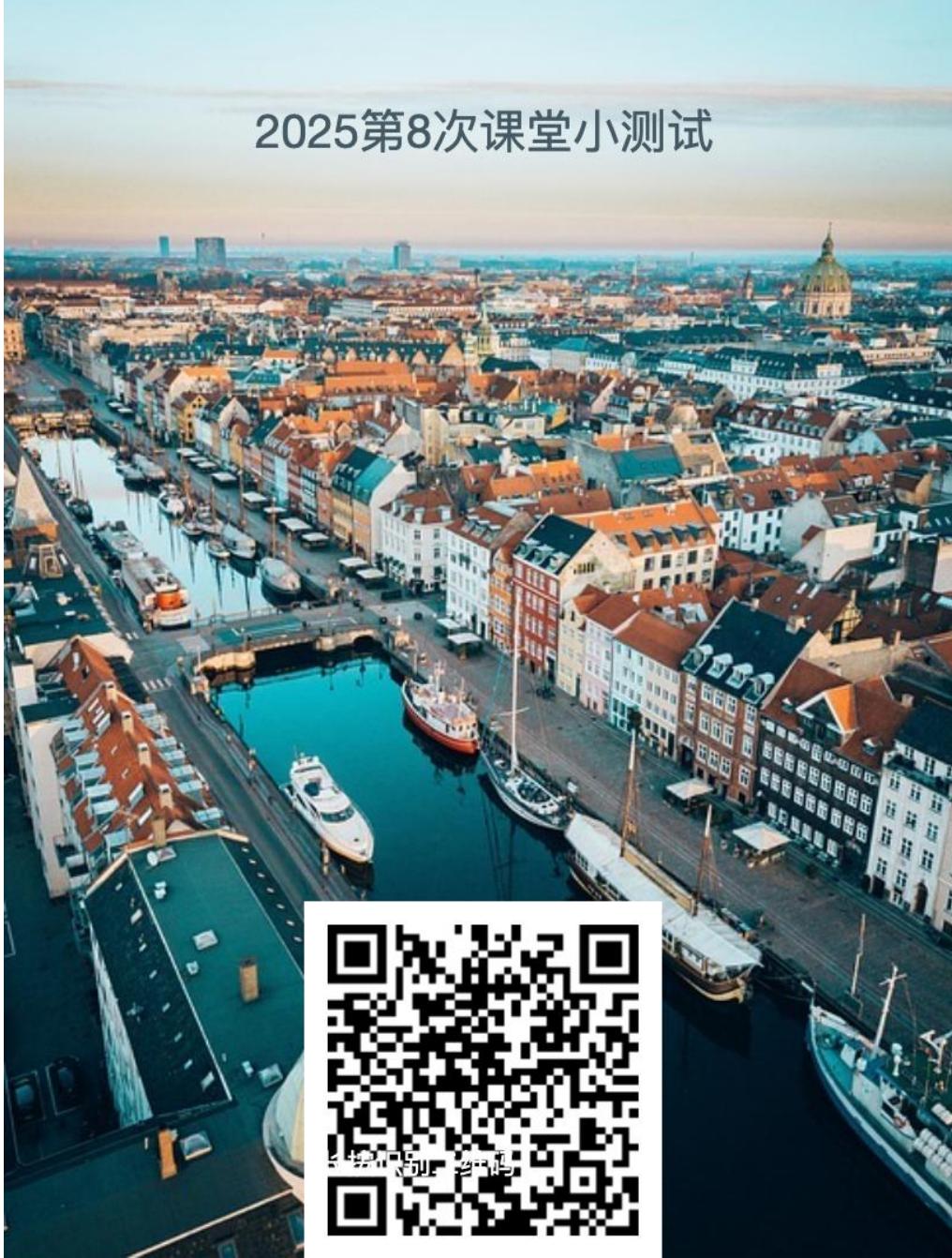
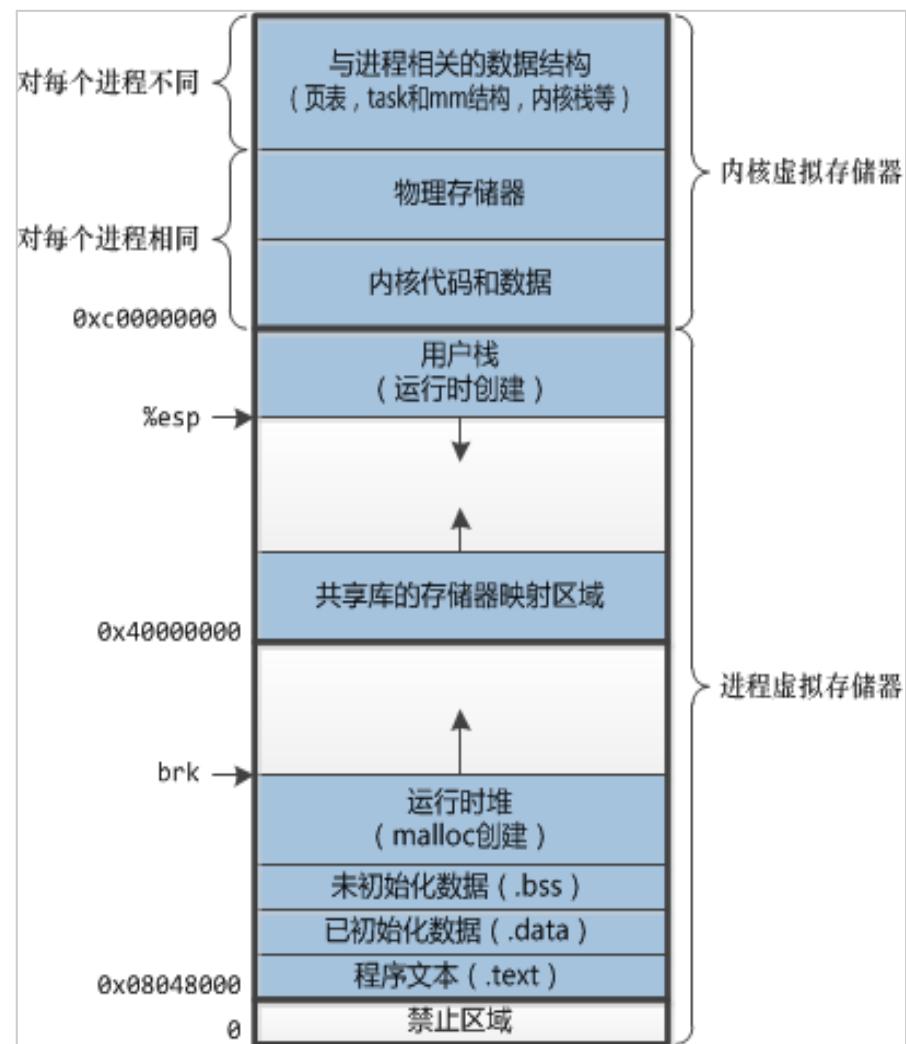
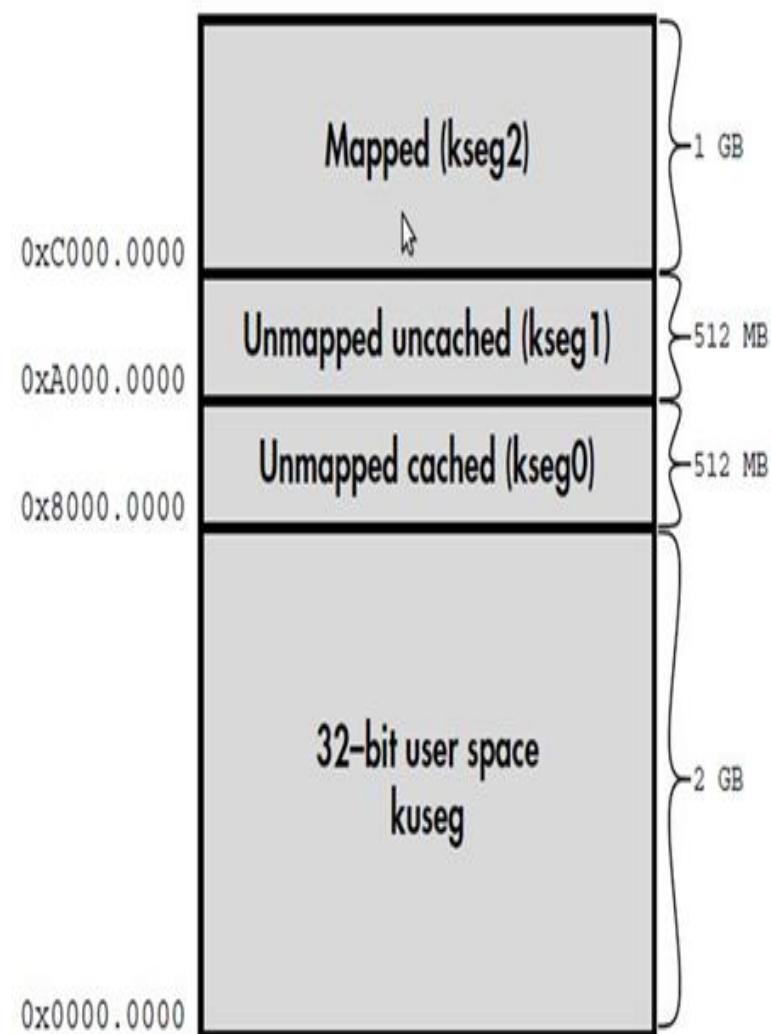


1

2025第8次课堂小测试



虚拟地址、进程空间



实验OS的虚拟内存布局和页表位置

实验OS的PCB

Listing 8: 进程控制块

```
struct Env {  
    struct Trapframe env_tf;           // Saved registers  
    LIST_ENTRY(Env) env_link;          // Free LIST_ENTRY  
    u_int env_id;                     // Unique environment identifier  
    u_int env_parent_id;              // env_id of this env's parent  
    u_int env_status;                 // Status of the environment  
    Pde *env_pgdir;                  // Kernel virtual address of page dir  
    u_int env_cr3;  
    LIST_ENTRY(Env) env_sched_link;    //  
    u_int env_pri;  
};
```



操作系统 *Operating System*

第四章 进程管理

4.1 进程与线程

原仓周 82338521

yuancz@buaa.edu.cn

内容提要

- 进程概念的引入
- 进程状态与控制
- 线程概念的引入
- 线程的实现方式
- 小结

从一个例子说起

- 在X年X月X日， X人， 家中：
 - 电话铃响 → 要接电话。
 - 有人敲门 → 要去开门。
 - 孩子哭了 → 要哄孩子。
 - 要下雨了 → 收衣服。
 - 水龙头开着 → 要关水。
- 怎么办？



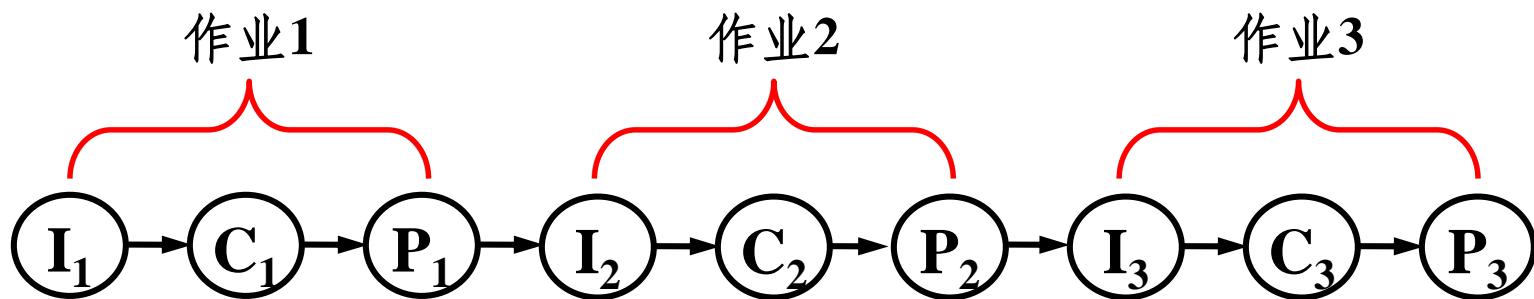
操作系统也是如此！

多个程序间的运行关系

- 顺序执行 **Sequential**
- 并发 **Concurrent**：设有两个活动 a_1 和 a_2 ，如果在某一指定的时间 t ，无论 a_1 和 a_2 是在同一处理机上还是在不同的处理机上执行，只要 a_1 和 a_2 都处在各自的起点和终点之间的某一处，则称 a_1 和 a_2 是并发执行的
- 并行 **Parallel**：如果考虑两个程序，它们在同一时间度量下同时运行在不同的处理机上，则称这两个程序是并行执行的

程序的顺序执行与特征

- **顺序性**: 按照程序结构所指定的次序（可能有分支或循环）
- **封闭性**: 独占全部资源，计算机的状态只由于该程序的控制逻辑所决定
- **可再现性**: 初始条件相同则结果相同。



程序的并发执行

- 程序的并发执行是指若干个程序（或程序段）同时在系统中运行，这些程序（或程序段）的执行在时间上是重叠的。
- 所谓执行在时间上是重叠的，是指执行一个程序（或程序段）的第一条指令是在执行另一个程序（或程序段）的最后一条指令完成之前开始。

前趋图

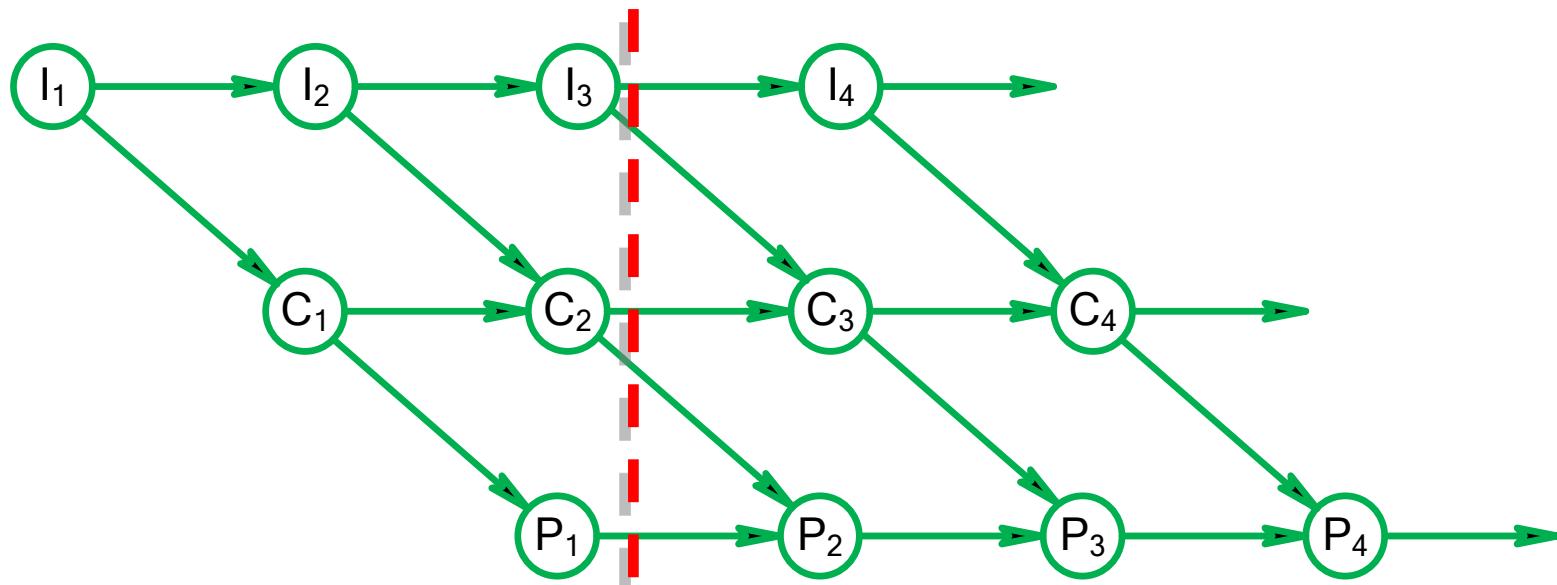
- 前趋图是一个有向无循环图，图中的每个结点可以表示一条语句、一个程序段或进程，结点间的有向边表示语句或程序段的执行次序。



表示A和B的执行顺序为：先A后B，可写成：
 $A \rightarrow B$

并发执行例子

- 例如：程序1、2、3…并发执行。对每个程序而言，其输入、计算和输出这三个操作必须顺序执行；三个程序间存在如下关系：
 - $I_i \rightarrow C_i, I_i \rightarrow I_{i+1}, C_i \rightarrow P_i, C_i \rightarrow C_{i+1}, P_i \rightarrow P_{i+1}$
 - I_{i+1} 和 C_i 及 P_{i-1} 可并发。



程序并发执行时的特征

- **间断性**: 并发程序具有“执行---暂停----执行”这种间断性的活动规律。
- **非封闭性**: 多个程序共享系统中的资源，这些资源的状态将由多个程序来改变，致使程序之间相互影响。
- **不可再现性**: 在初始条件相同的情况下，程序的执行结果依赖于执行的次序。

不可再现性的例子

- 程序并发执行时可能出现与时间有关的错误。
- 例如：

进程A:



.....
a1 N := N+1;
.....

进程B:



.....
b1 Print(N);
b2 N := 0;
.....

- 设在两进程运行之前，N的值为2。则两进程运行结束后，打印出的结果及N值可为：
 - 若顺序为a1,b1,b2 → 打印结果为3, N=0;
 - 若顺序为b1,b2,a1 → 打印结果为2, N=1;
 - 若顺序为b1,a1,b2 → 打印结果为2, N=0;

什么情况？

发生了竞争：

- 竞争：多个进程在读写一个共享数据时结果依赖于它们执行的相对时间，这种情形叫做竞争
- 竞态：多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关，称为竞态。

什么条件下两个进程不会发生竞争？

- 并发进程的无关性是进程与时间无关的一个充分条件，这一条件在1966年首先由Bernstein提出，称为 Bernstein 条件

程序正确并发执行的Bernstein条件

定义：

- $R(S_i)$: S_i 的读子集, 其值在 S_i 中被引用的变量的集合
- $W(S_i)$: S_i 的写子集, 其值在 S_i 中被改变的变量的集合

Bernstein条件：

- 两个进程 S_1 和 S_2 可并发, 当且仅当下列条件同时成立：
 - $R(S_1) \cap W(S_2) = \emptyset$
 - $W(S_1) \cap R(S_2) = \emptyset$
 - $W(S_1) \cap W(S_2) = \emptyset$

Bernstein条件举例

- S1: $c := a + b;$
- S2: $d := a - b;$
- 读集: $R(S1) = \{a, b\}$, $R(S2) = \{a, b\};$
- 写集: $W(S1) = \{c\}$, $W(S2) = \{d\};$

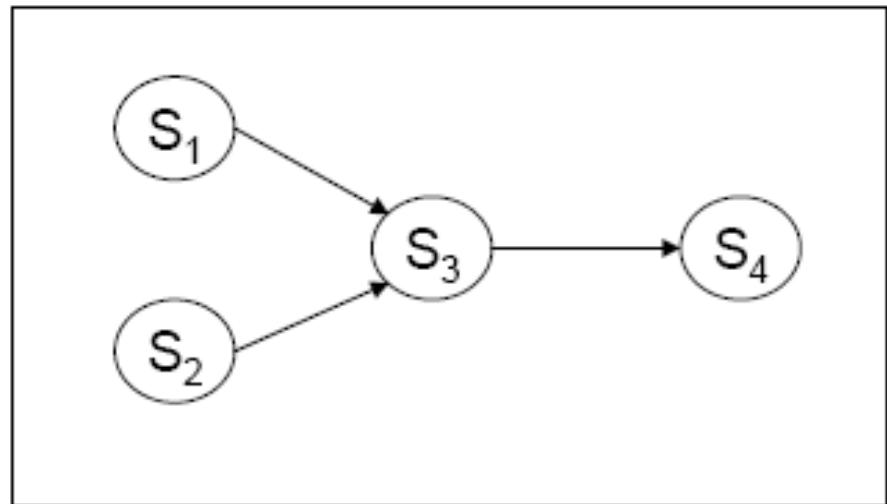
Bernstein 条件:

$$\begin{aligned} & [R(S1) \cap W(S2)] \cup [R(S2) \cap W(S1)] \cup [W(S1) \cap W(S2)] \\ & = [\{a, b\} \cap \{d\}] \cup [\{a, b\} \cap \{c\}] \cup [\{c\} \cap \{d\}] = \Phi \end{aligned}$$

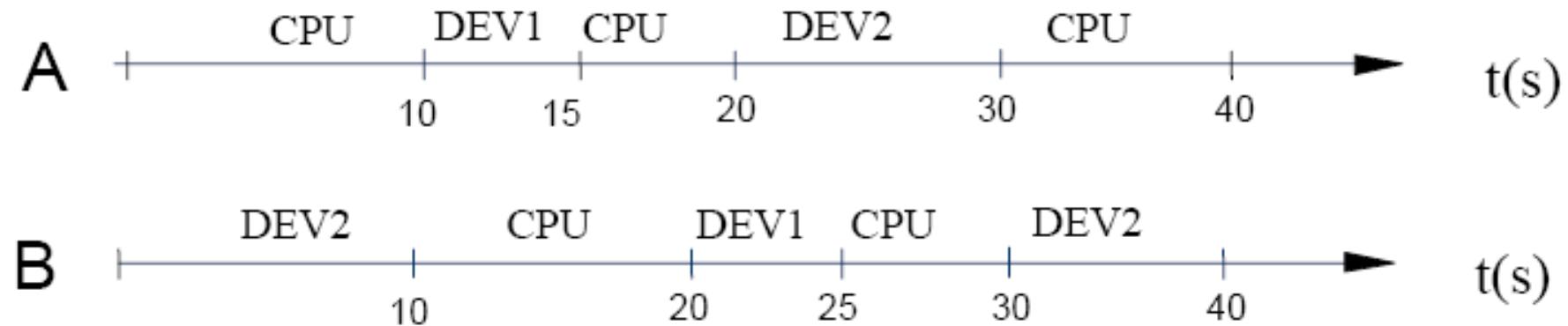
判断程序并发执行结果是否可再现的充分条件

是否能确定执行?

- $S_1 : a := x + y$
- $S_2 : b := z + 1$
- $S_3 : c := a - b$
- $S_4 : w := c + 1$



举例：A和B程序顺序执行的结果



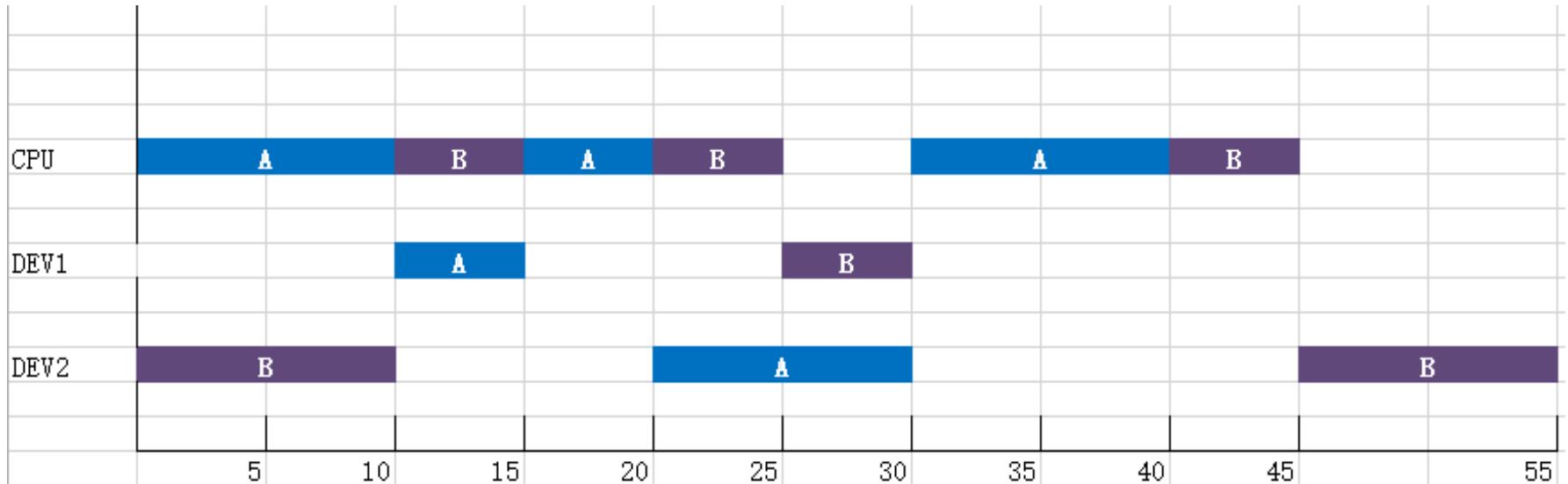
在顺序环境下

$$\text{CPU利用率} = 40/80 = 50\%$$

$$\text{DEV1利用率} = 10/80 = 12.5\%$$

$$\text{DEV2利用率} = 30/80 = 37.5\%$$

A和B程序并发执行的结果



CPU利用率： $40/55=72.7\%$ (50%)

DEV1利用率： $10/55=18.1\%$ (12.5%)

DEV2利用率： $30/55=54.6\%$ (37.5%)

进程的引入

- “程序”与“计算”不是一一对应的关系：一个程序段可能对应多个“计算”
- 多道程序+资源的限制：执行-暂停-执行
 - 直接制约：逻辑上相互依赖
 - 间接制约：等待资源
- 使用“程序”不能揭示多道程序、分时系统引发的动态特性，因此引入“进程”（Process）

MULTICS和IBM的CTSS360中首先
引入了进程的概念

进程的定义和特征

- 进程可以表示程序的一次执行
- 进程是一个程序及其数据，在处理机上顺序执行时所发生的活动
- 进程是可以和别的计算并发执行的计算
- 进程可定义为一个数据结构，及能在其上进行操作的一个程序
- 进程是程序在一个数据集合上运行的过程，它是系统进行**资源分配**和**调度**的一个**独立单位**

进程的特征

- 动态性：进程是程序的一次执行过程。动态性还表现为它因创建而产生，因调度而执行，因无资源而暂停，因撤销而消亡。而程序是静态实体
- 并发性：多个进程实体同时存在于内存中，能在一段时间内同时运行
- 独立性：在传统OS中，进程是独立运行的基本单位
- 异步性：也叫制约性，进程之间相互制约，进程以各自独立的不可预知的速度向前推进
- 结构特征：程序段，数据段，**进程控制块PCB**

一个进程的组成

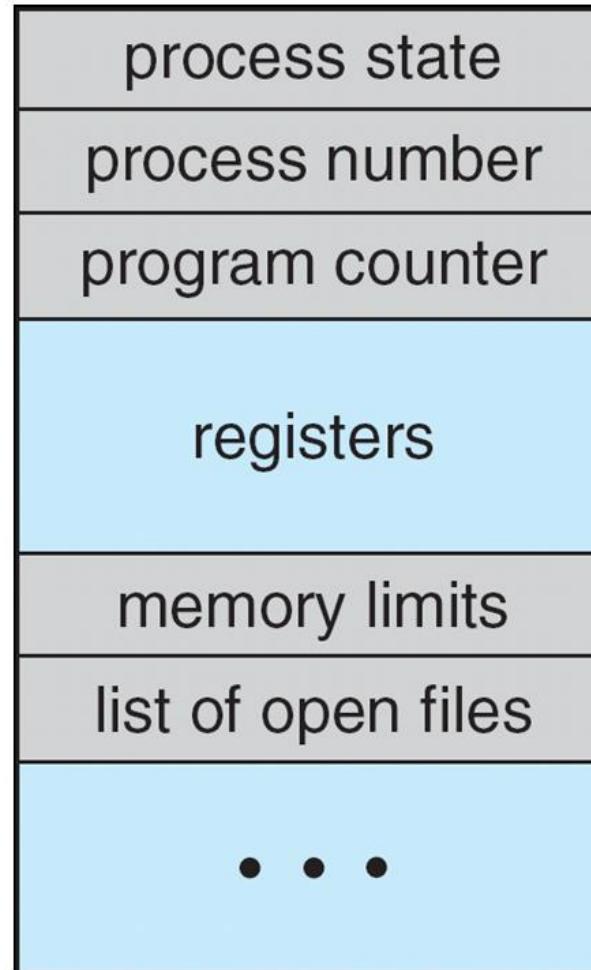
- 程序的代码
- 程序的数据
- PC中的值，用来指示下一条将运行的指令
- 一组通用的寄存器的当前值，堆、栈
- 一组系统资源（如打开的文件）

进程控制块（PCB）

- 系统为每个进程定义了一个数据结构：进程控制块PCB（Process Control Block）
- 作用：
 - 进程创建、撤销
 - 进程唯一标志
 - 限制系统进程数目
- 进程控制块是进程管理和控制的最重要的数据结构，每一个进程均有一个PCB，在创建进程时，建立PCB，伴随进程运行的全过程，直到进程撤销而撤销

PCB的内容 (1/4)

- 进程标识符
- 程序和数据地址
- 现行状态
- 现场保留区
- 互斥和同步机制
- 进程通信机制
- 优先级
- 资源清单
- 链接字
- 家族关系 ...



PCB的内容 (2/4)

进程标识符:

- 每个进程都必须有一个唯一的标识符，可以是字符串，也可以是一个数字。Linux系统中就是一个整型数。在进程创建时由系统赋予。

程序和数据地址:

- 把PCB与其程序和数据联系起来。

当前状态:

- 为了管理的方便，系统设计时会将相同的状态的进程组成一个队列，如就绪进程队列，等待进程则要根据等待的事件组成多个等待队列，如等待打印机队列、等待磁盘I/O完成队列等等。

PCB的内容 (3/4)

现场保护区：

- 当进程因某种原因不能继续占用CPU时（等待打印机），释放CPU，这时就要将CPU的各种状态信息保护起来，为将来再次得到处理机恢复CPU的各种状态，继续运行。

同步与同步机制：

- 用于实现进程间互斥、同步和通信所需的信号量等。

优先级：

- 进程的优先级反映进程的紧迫程度，通常由用户指定和系统设置。Linux系统采用用户设置和系统计算相结合的方式确定进程的优先级。

PCB的内容 (4/4)

资源清单：

- 列出所拥有的除CPU外的资源记录，如拥有的I/O设备，打开的文件列表等

链接：

- 根据进程所处的现行状态，进程相应的PCB参加到不同队列中
- PCB链接指出该进程所在队列中下一个进程PCB的首地址

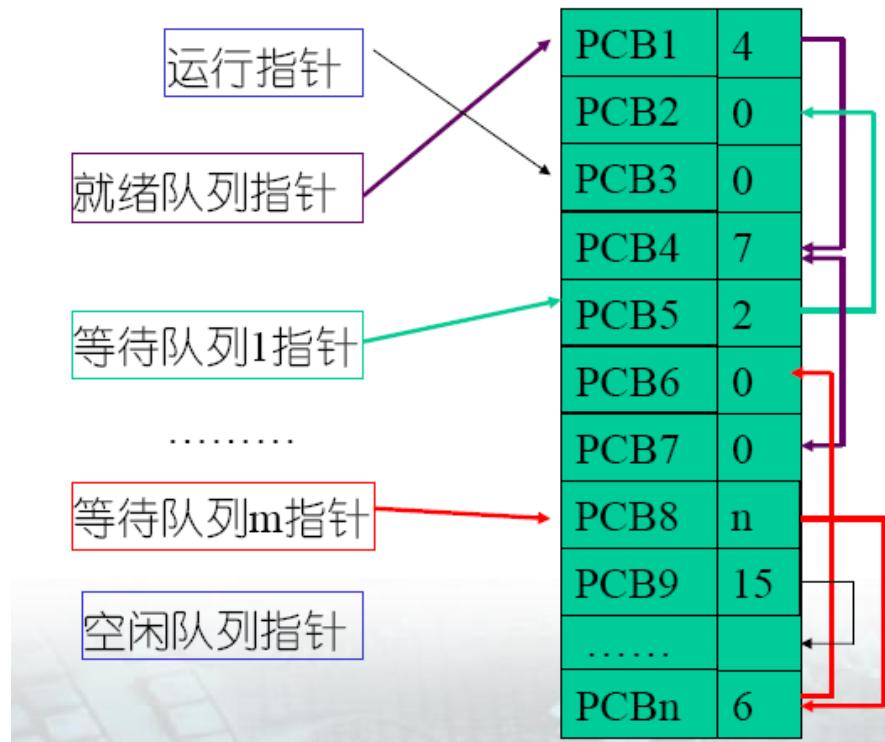
其他信息：

- 如进程记账信息，进程占用CPU的时间等。

在Linux中每一个进程都由**task_struct**数据结构来定义，**task_struct**就是我们通常所说的**PCB**

PCB的组织方式 (1/3)

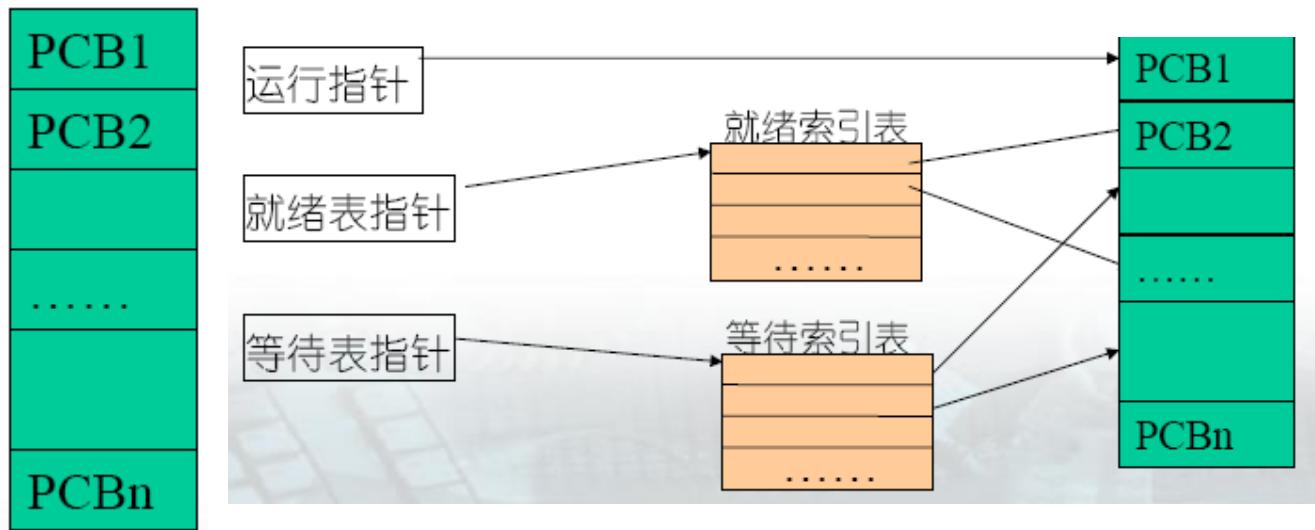
- 线性表
- 链接方式
- 索引方式



线性表方式：不论进程的状态如何，将所有的**PCB**连续地存放在内存的系统区。这种方式适用于系统中进程数目不多的情况。

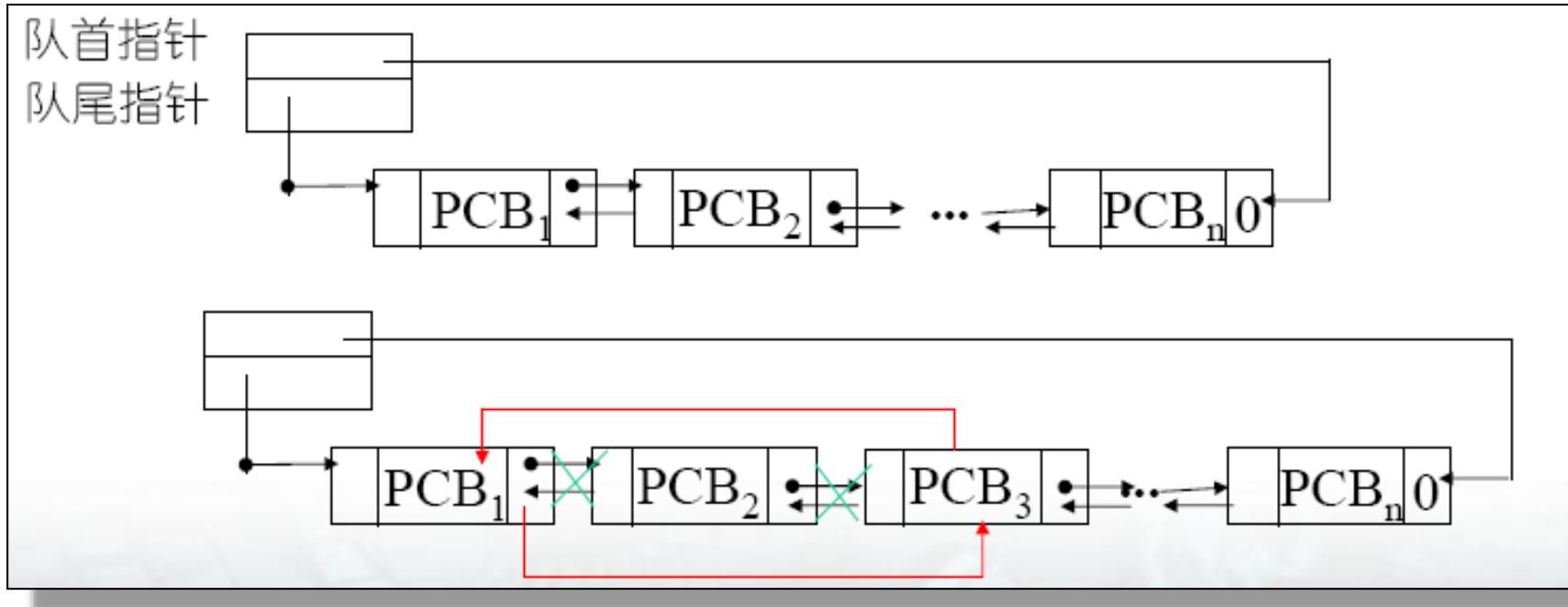
PCB的组织方式 (2/3)

- 线性表
- 索引方式
- 链接方式



索引表方式：该方式是线性表方式的改进，系统按照进程的状态分别建立就绪索引表、阻塞索引表等。

PCB的组织方式 (3/3)



链接表方式：系统按照进程的状态将进程的PCB组成队列，从而形成就绪队列、阻塞队列、运行队列等。

引入进程的利弊

- 利：提高效率
- 弊：空间开销、时间开销

进程与程序的区别

- 进程是动态的，程序是静态的：程序是有序代码的集合；进程是程序的执行。通常进程不可在计算机之间迁移；而程序通常对应着文件、静态和可以复制
- 进程是暂时的，程序的永久的：进程是一个状态变化的过程，程序可长久保存
- 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）
- 进程与程序的对应关系：通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序

有一个计算机科学家，有一天女儿过生日，想亲手给女儿做一个生日蛋糕。所以他就找了一本有关做蛋糕的食谱，买了一些原料，面粉、鸡蛋、糖、香料等，然后边看边学边做。

食谱=程序； 科学家=CPU；

原料=数据； 做蛋糕=进程；

这时小儿子哭着跑进来，说手被蜜蜂蛰了。教授只好把蛋糕先放在一边。他在食谱上做了个标记，把状态信息记录了起来。然后又去找了一本医疗手册，查到了相关的内容，按照上面的指令一步步地执行。当伤口处理完之后，又回到厨房继续做蛋糕。

CPU从一个进程（做蛋糕）切换到另一个进程（医疗救护）。

内容提要

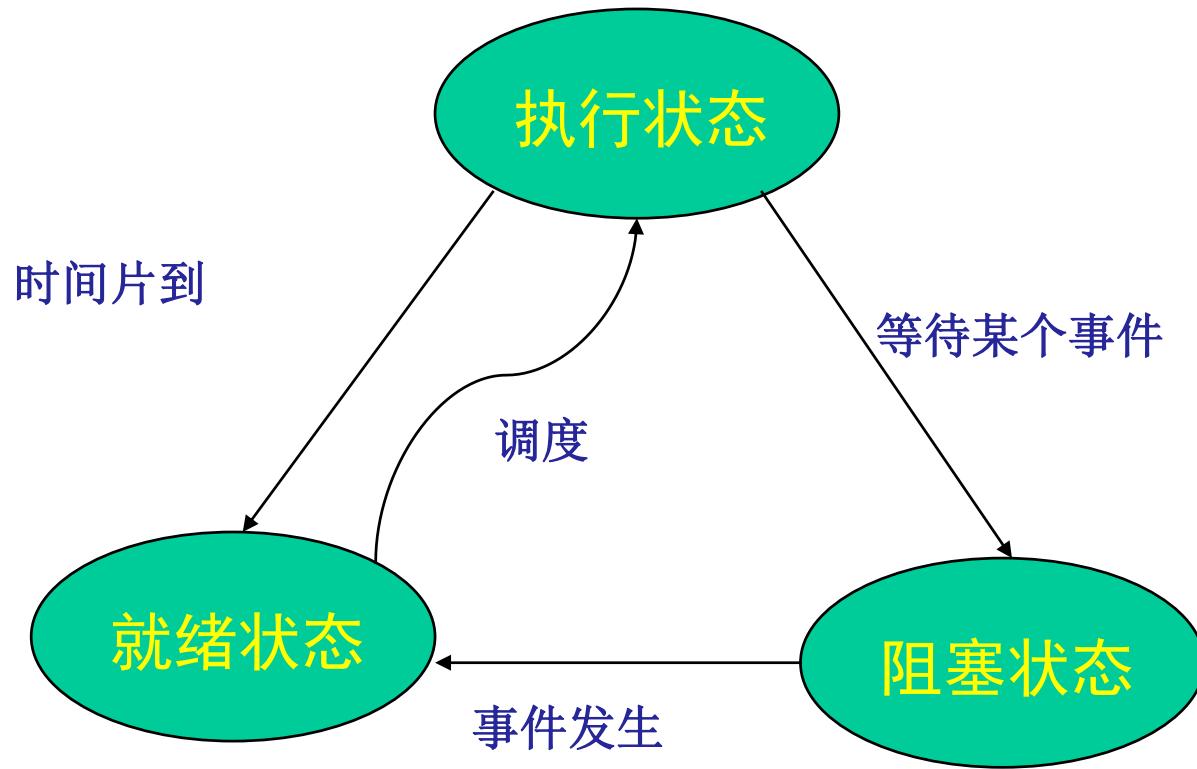
- 进程概念的引入
- 进程状态与控制
- 线程概念的引入
- 线程的实现方式
- 小结

进程状态及其演变

进程的三种基本状态

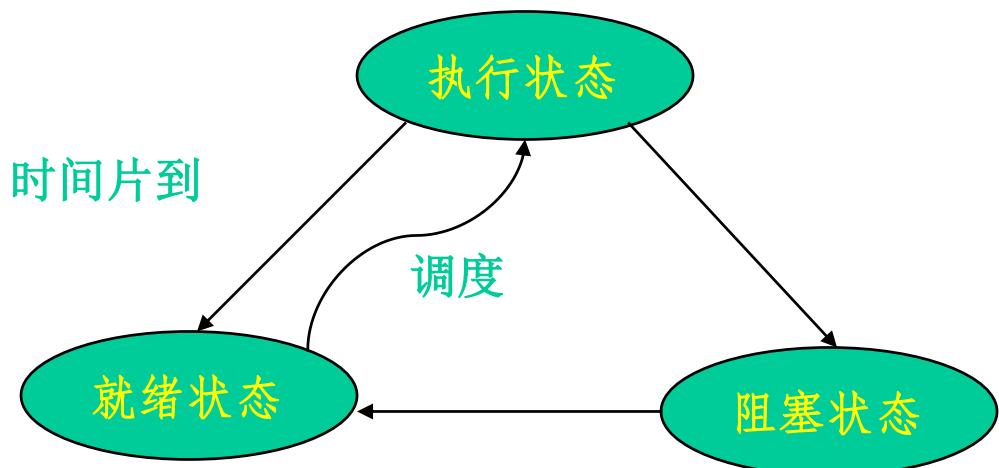
- **就绪状态**: 进程已获得除处理机外的所需资源，等待分配处理机资源；只要分配CPU就可执行。
- **执行状态**: 占用处理机资源；处于此状态的进程的数目小于等于CPU的数目。在没有其他进程可以执行时（如所有进程都在阻塞状态），通常会自动执行系统的idle进程（相当于空操作）。
- **阻塞状态**: 正在执行的进程，由于发生某种事件而暂时无法执行，便放弃处理机处于暂停状态。

进程状态转换图



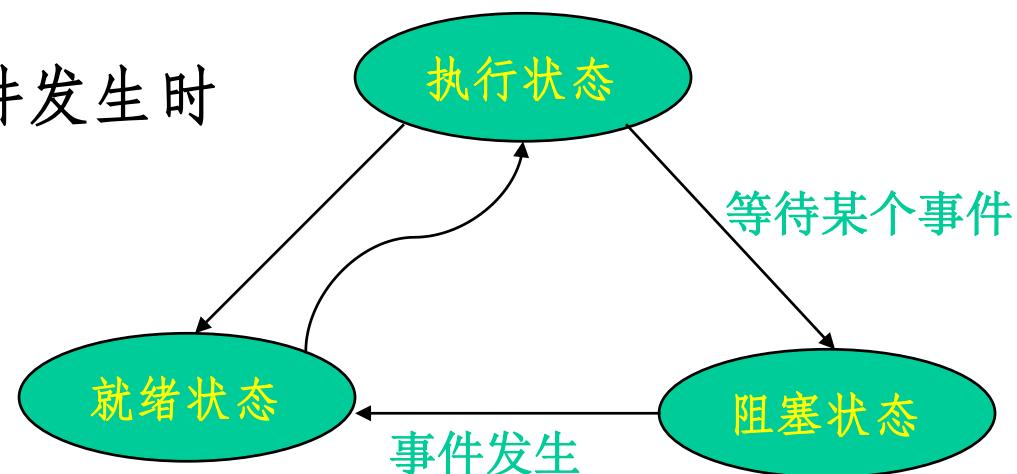
进程状态转换(1/2)

- 就绪--> 运行
 - 时间一到，调度程序选择一个进程运行
- 运行--> 就绪
 - 运行进程用完了时间片
 - 运行进程被中断，因为一高优先级进程处于就绪状态

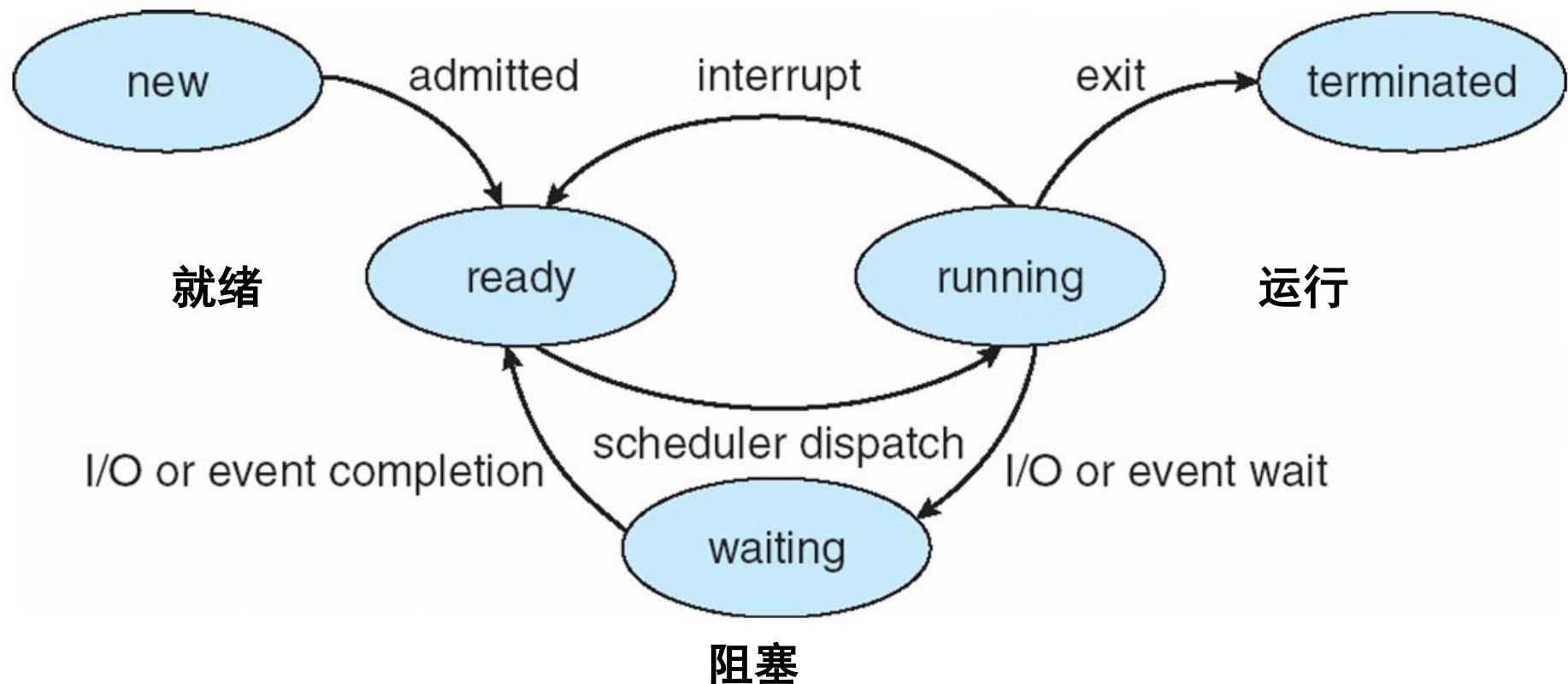


进程状态转换(2/2)

- 运行--> 阻塞
 - 当一进程所需的东西必须等待时
 - OS尚未完成服务
 - 对一资源的访问尚不能进行
 - 初始化I/O 且必须等待结果
 - 等待某一进程提供输入(IPC)
- 阻塞--> 就绪
 - 当所等待的事件发生时



比较完整的进程状态转换模型



进程控制

进程控制的主要任务是

- 创建和撤销进程
- 实现进程的状态转换

由内核来实现

进程的创建 情形

- 提交一个批处理作业
- 用户登录
- 由OS创建，用以向一用户提供服务
- 由已存在的一进程创建

进程撤消 的情形

- 用户退出登录
- 进程执行一中止服务请求
- 出错及失败因素
- 正常结束
- 给定时限到
-

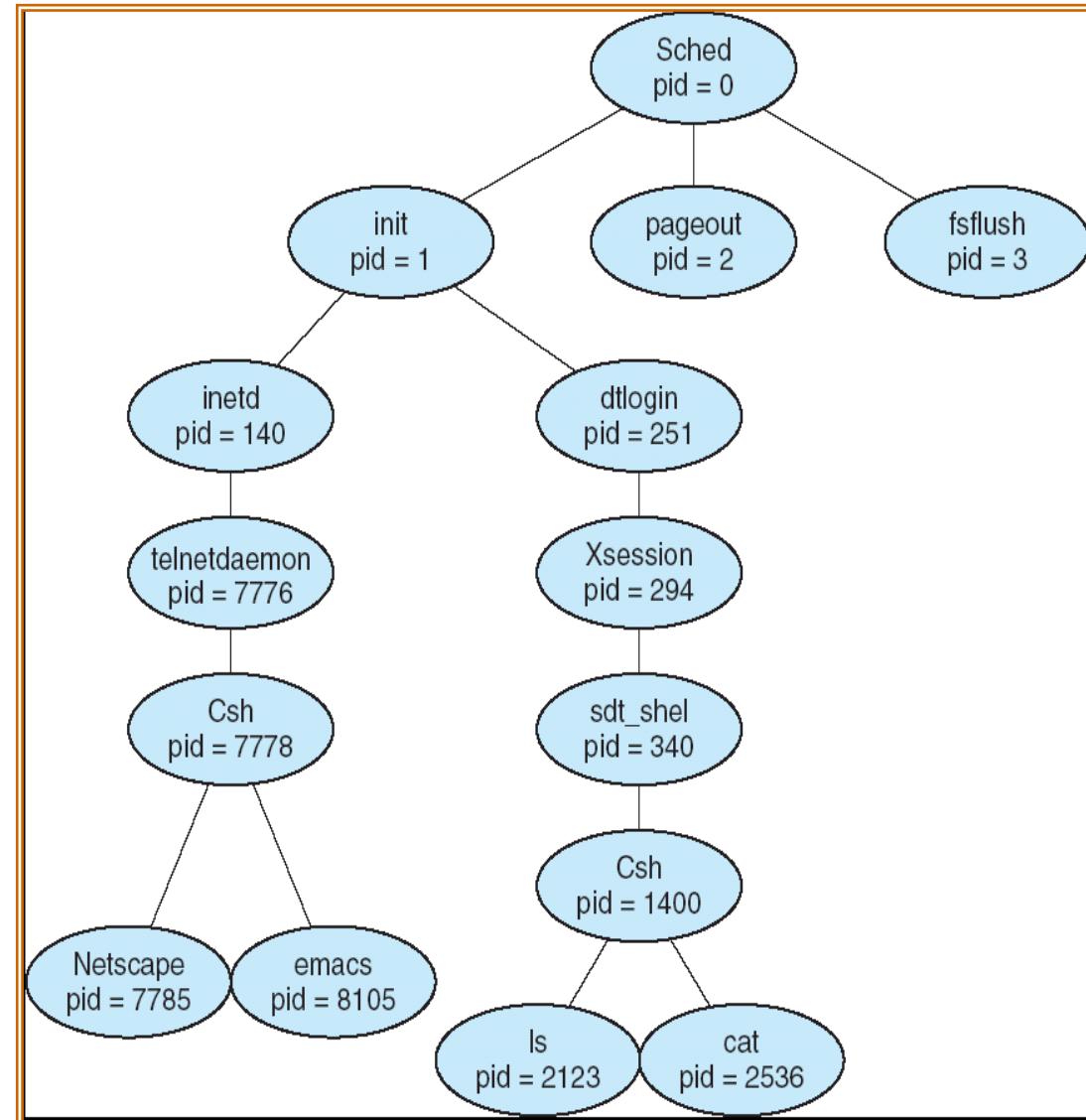
原语

- 原语：由若干条指令所组成的指令序列，来实现某个特定的操作功能
 - 指令序列执行是连续的，不可分割
 - 是操作系统核心组成部分
 - 必须在管态（内核态）下执行，且常驻内存
- 与系统调用的区别

进程树

- 进程图
- 创建原语
 - (fork, exec)
- 撤消原语(kill)
 - 释放资源、撤消子进程、重新调度。

进程图是一棵有向树，结点代表进程，一棵树表示一个家族，根结点为该家族的祖先 (Ancestor)



进程创建实例

```
#include <sys/types.h>
int glob = 6; /* external variable in initialized data */

int
main(void)
{
    int var; /* automatic variable on the stack */
    pid_t fpid;

    var = 88;
    printf("before fork\n"); /* we don't flush stdout */

    if ((fpid = fork()) < 0)
        err_sys("fork error");
    else if (fpid == 0) { /* child */
        glob++;
        /* modify variables */
        var++;
    } else
        sleep(2); /* parent */

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

fork()函数使用举例

- 在语句`fpid=fork()`之前，只有一个进程在执行这段代码，但在这条语句之后，就变成两个进程在执行了
- 在`fork`函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，`fork`函数返回0，在父进程中，`fork`返回新创建子进程的进程ID。我们可以通过`fork`返回的值来判断当前进程是子进程还是父进程

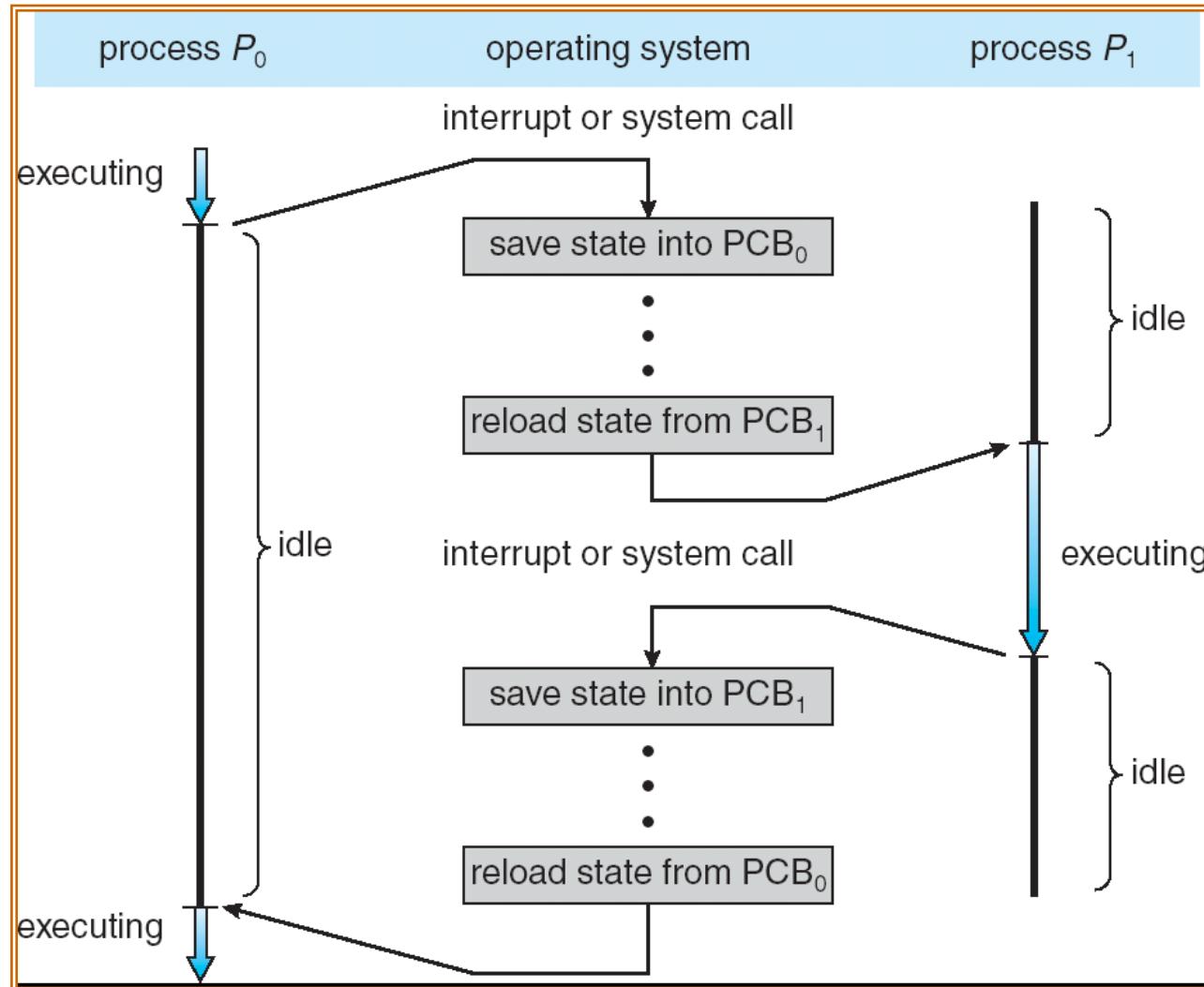
fork()函数使用举例

- 为什么两个进程的fpid不同呢，这与fork函数的特性有关。fork调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：
 - a) 在父进程中，fork返回新创建子进程的进程ID；
 - b) 在子进程中，fork返回0；
 - c) 如果出现错误，fork返回一个负值；
- fpid的值为什么在父子进程中不同。其实就相当于链表，进程形成了链表，父进程的fpid指向子进程的进程id, 因为子进程没有子进程，所以其fpid为0.

思考？？？

- **问题：**如果你要设计一个OS，怎么样来实现其中的进程机制？
 - 包括进程的创建
 - 进程的管理
 - 进程的终止
 - 进程的状态转换等等

进程的状态与控制：示例



辨析：进程上下文切换 vs 陷入内核

- 进程上下文切换（Process Context Switch）
 - 通常由调度器执行
 - 保存进程执行断点
 - 切换内存映射（页表基址、flush TLB）
- 陷入/退出内核（也称为模态切换, Mode Switch）
 - CPU状态改变
 - 由中断、异常、Trap指令（系统调用）引起
 - 需要保存执行现场（寄存器、堆栈等）

辨析：进程上下文切换 vs 陷入内核

- 系统调用涉及到进程从用户态到内核态的切换 (mode switch)
 - 这个时候涉及到的切换主要是寄存器上下文的切换
- 和通常所说的进程上下文切换 (Process Context Switch) 不同， mode switch 的消耗相对要小很多

内容提要

- 进程概念的引入
- 进程状态与控制
- 线程概念的引入
- 线程的实现方式
- 小结

从一个实例讲起

【案例】 编写一个MP3播放软件。

核心功能模块有三个：

- (1) 从MP3音频文件当中读取数据；
- (2) 对数据进行解压缩；
- (3) 把解压缩后的音频数据播放出来。

实例的伪代码

```
main() {  
    while(TRUE) {  
        Read();  
        Decompress();  
        Play();  
    }  
}  
Read() { ... }  
Decompress() { ... }  
Play() { ... }
```

■ 问题：

- 播放出来的声音能否连贯？
- 各个函数之间不是并发执行，影响资源的使用效率；

多进程方案

程序1

```
main()
{
    while(TRUE)
    {
        Read();
    }
}
Read() { ... }
```

程序2

```
main()
{
    while(TRUE)
    {
        Decompress();
    }
}
Decompress() { ... }
```

程序3

```
main()
{
    while(TRUE)
    {
        Play();
    }
}
Play() { ... }
```

- 进程拥有各自的空间……

这样做有什么问题吗？



线程 (thread)

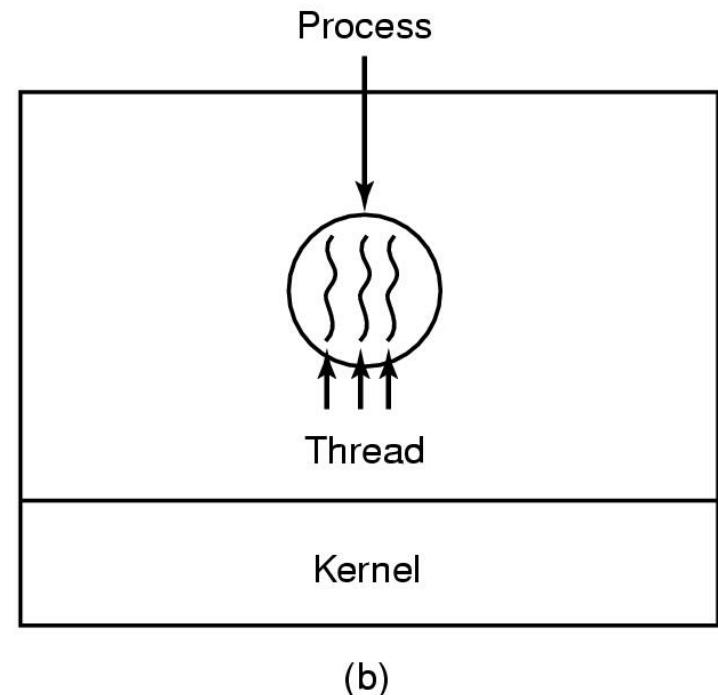
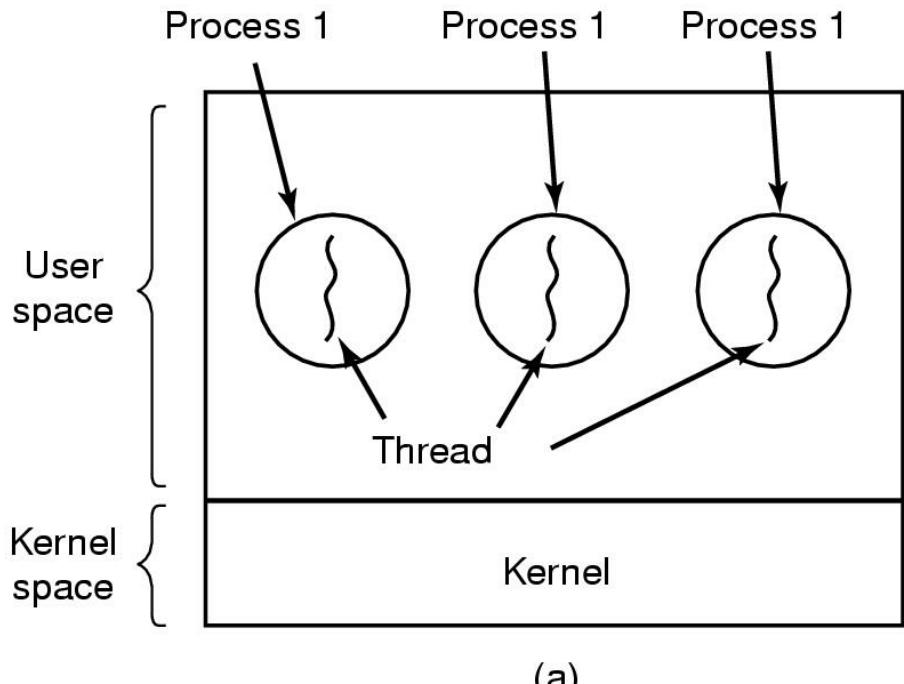
- 进程的不足：
 - 进程只能在一个时间干一件事，如果想同时干两件事或多件事，进程就无能为力了
 - 进程在执行的过程中如果阻塞，例如等待输入，整个进程就会挂起，即使进程中有些工作不依赖于输入的数据，也将无法执行
- 需要提出一种新的实体，满足以下特性：
 - 实体之间可以并发地执行
 - 实体之间共享相同的地址空间

进程与线程

- 进程抽象了两个概念：
 - 资源拥有者
 - 可执行单元
- 现代操作系统将资源拥有者称为进程
 - process, task
- 可执行单元称为线程（Thread）。

线程：将 资源 与 计算 分离
提高并发效率。

多进程 v.s. 多线程



引入线程的目的

- 减小进程切换的开销
- 提高进程内的并发程度
- 共享资源

进程线程对比

■ 进程：

- 创建进程
- 撤消进程
- 进程切换

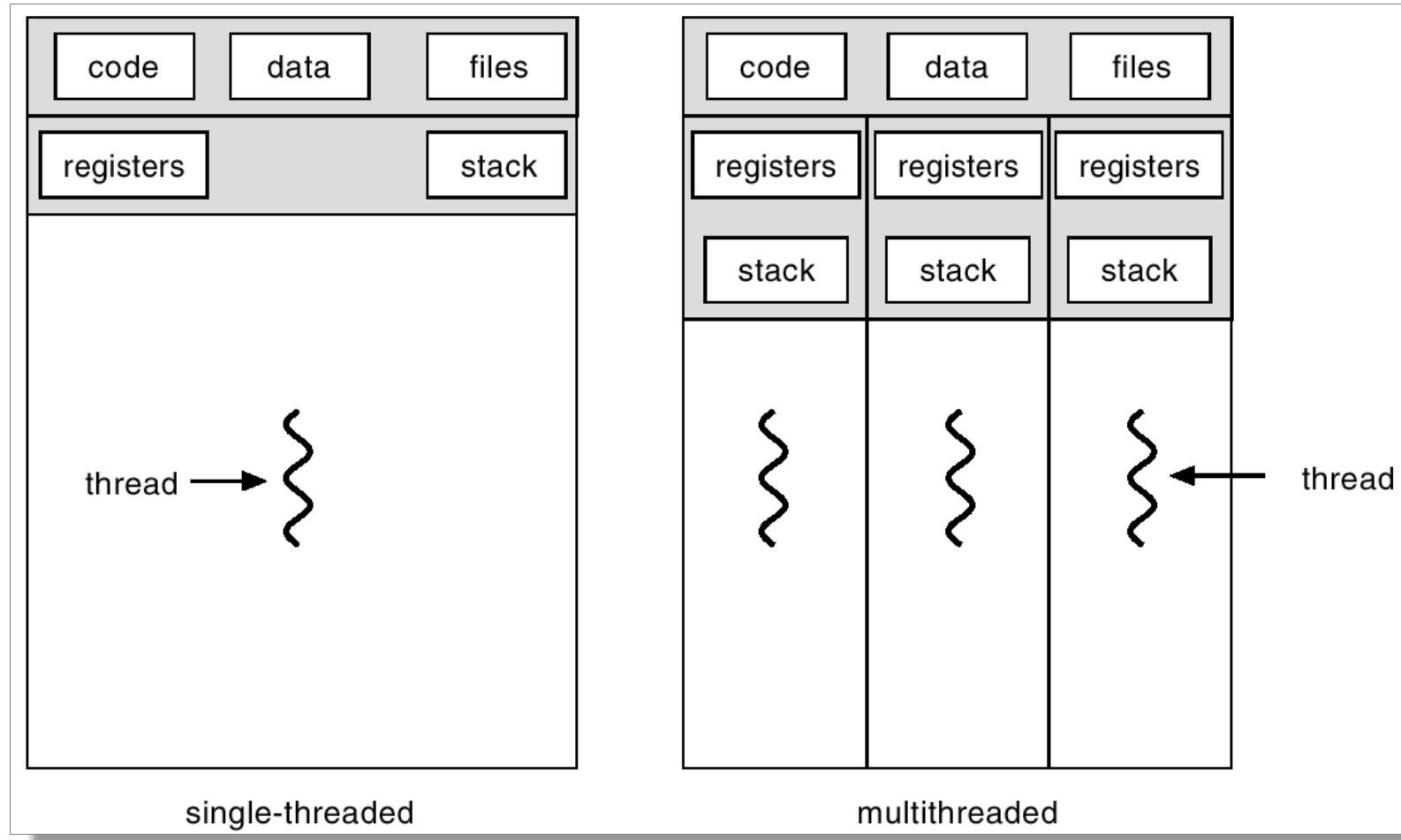
■ 线程：

- 进程中的一个实体
- 是一个CPU调度和分派的单位
- 基本上不拥有资源，只有必不可少的少量资源
- 可以与其他同进程的线程共享进程拥有的所有资源

进程线程对比

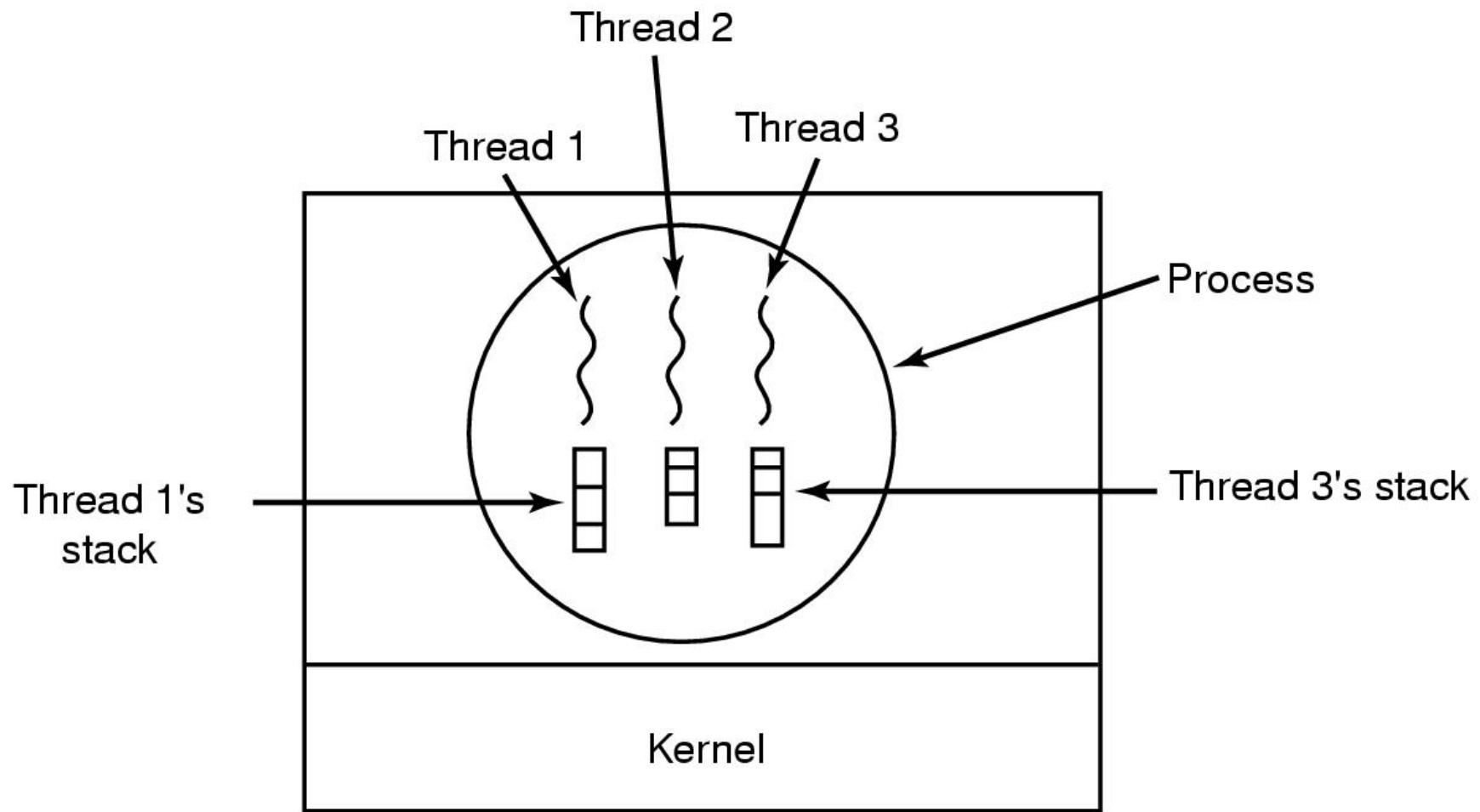
- 引入进程好处
 - 多个程序可以并发执行，改善资源使用率，提高系统效率
- 引入线程好处
 - 减少并发程序执行时所付出的时空开销，使得并发粒度更细、并发性更好

进程线程对比图



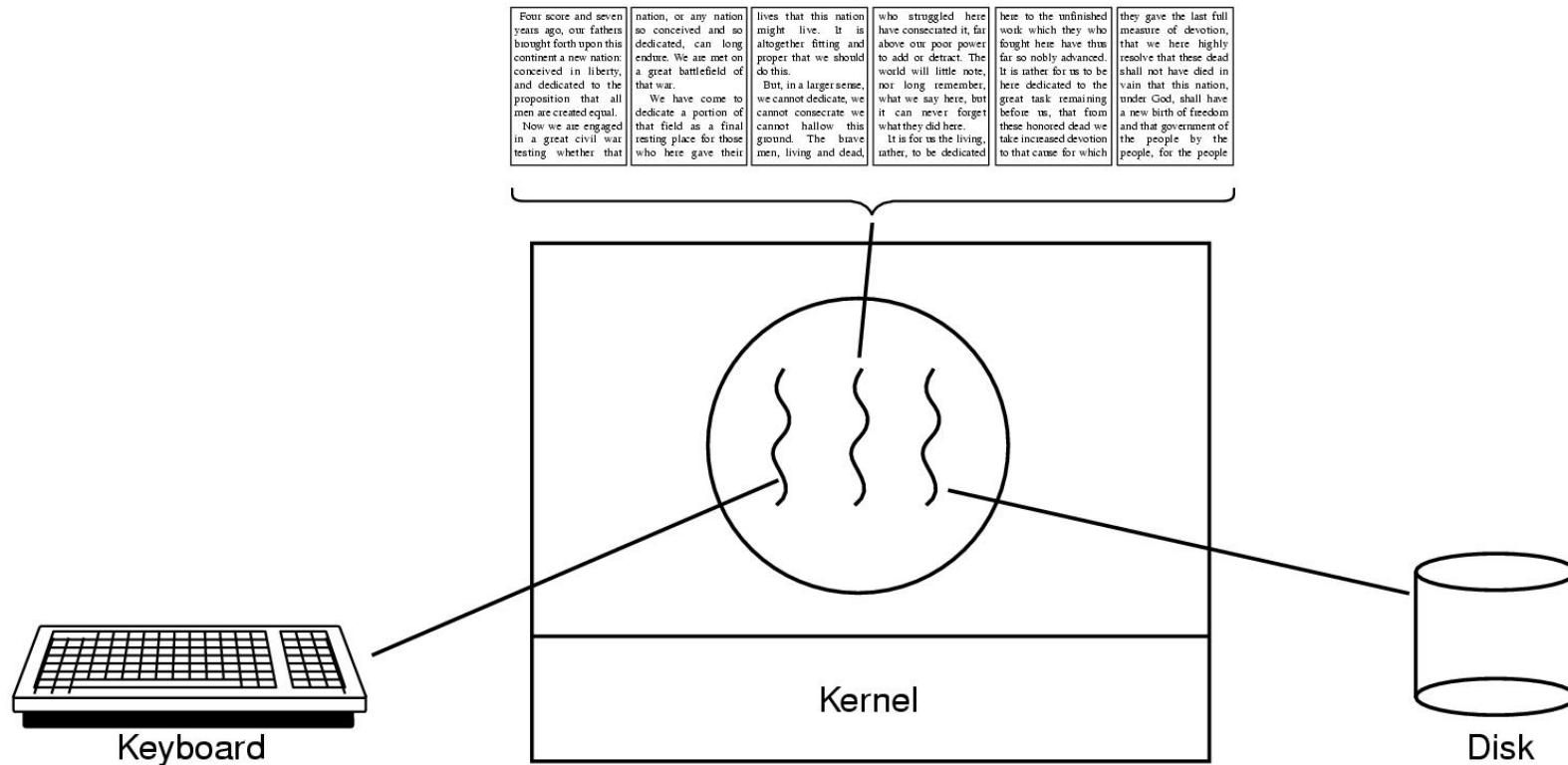
进程拥有：虚空间、进程映像、处理器保护、文件、I/O空间
线程拥有：运行状态、保存上下文（程序计数器）、栈、资源共享机制

每个线程有自己的栈

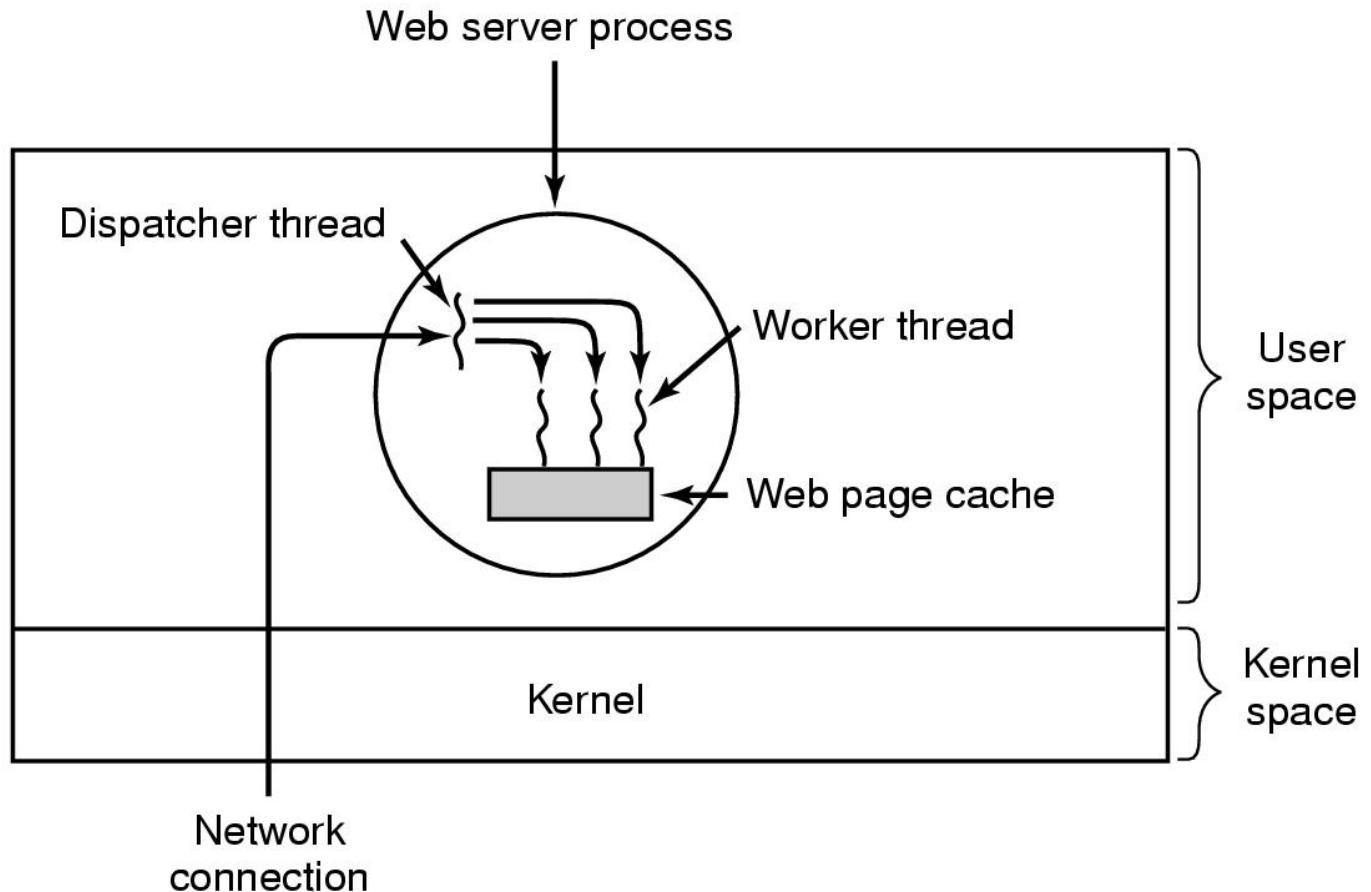


为什么要一定要线程？字处理软件

- 同时进行字符输入、拼写检查（分词检查）、按排版格式显示、定时存盘、打印等事情。



Web Server



实际操作系统与线程

- MS DOS是单进程、单线程系统；
- UNIX是多进程系统，但每个进程一个线程；
- JAVA是单进程、多线程系统；
- NT、OS/2、Solaris、Linux是多进程和多线程系统，每个进程有多个线程。

进程与线程的关系

- 1: 1 (UNIX系统)
- 1: M (NT、OS/2、Solaris、Linux)
- M: M (TRIX)
- M: 1

小结：引入线程的好处

- 线程比进程轻量：容易创建、撤销
 - 有些应用要求并行实体共享同一个地址空间和所有可用数据的能力
 - 创建一个线程比一个进程快10-100倍
 - 对于存在大量计算和大量I/O处理的应用，大幅度提高性能
 - 在多CPU/多核CPU系统中更有优势

小结：线程 v.s. 进程

- 一个进程可以拥有多个线程，而一个线程同时只能被一个进程所拥有
- 进程是资源分配的基本单位，线程是处理器调度的基本单位，所有的线程共享其所属进程的所有资源与代码
- 线程执行过程之中很容易进行协作同步，而进程需要通过消息通信进行同步
- 线程的划分尺度更小，并发性更高
- 线程共享进程的数据的同时，有自己私有的的栈
- 线程不能单独执行，但是每一个线程都有程序的入口、执行序列以及程序出口。它必须组成进程才能被执行

内容提要

- 进程概念的引入
- 进程状态与控制
- 线程概念的引入
- 线程的实现方式
- 小结

线程的实现方式

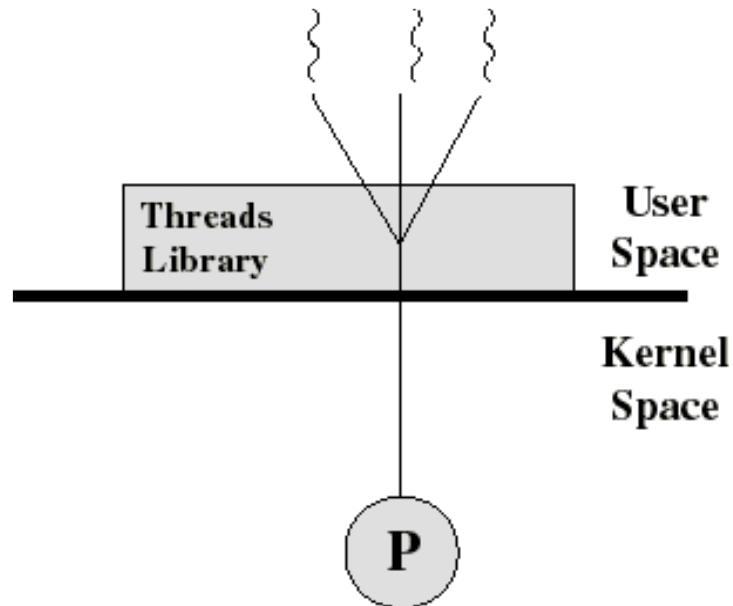
- 用户级线程：**User level threads(ULT)**
- 内核级线程：**Kernel level threads (KLT)**
- 混合实现方式

线程的实现方式

- 用户级线程：**User level threads(ULT)**
- 内核级线程：**Kernel level threads (KLT)**
- 混合实现方式

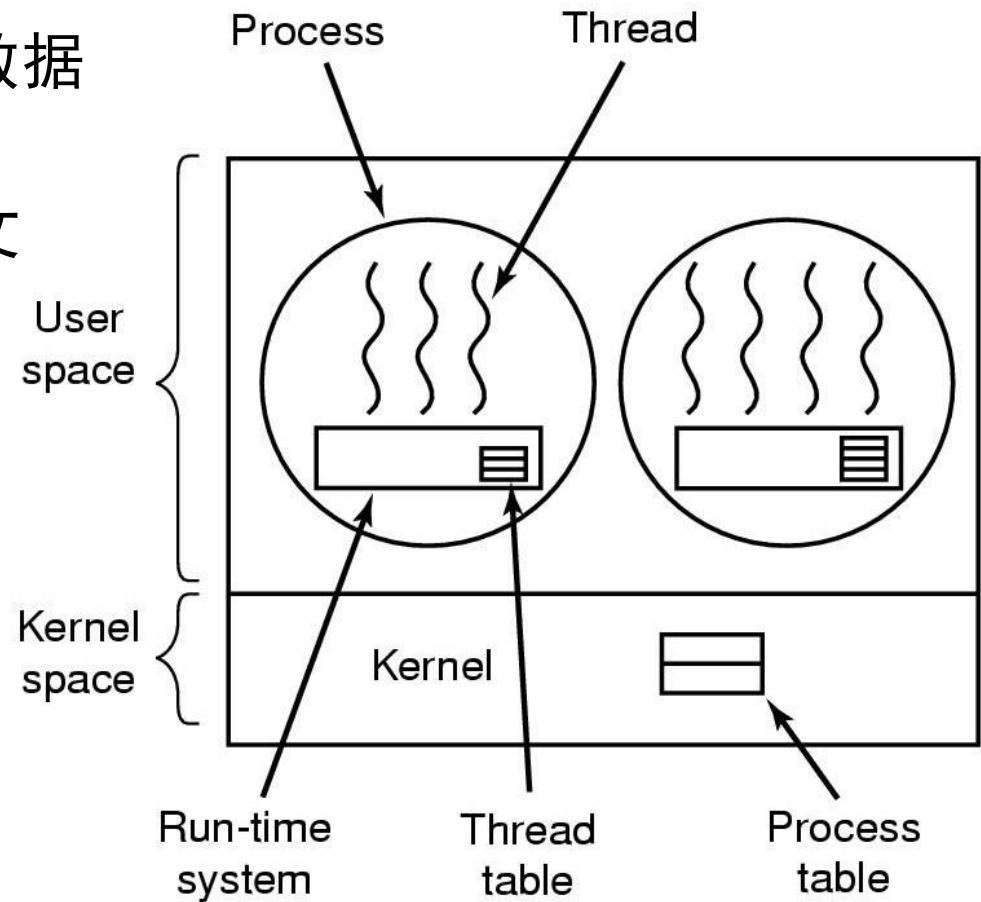
用户级线程

- 线程在用户空间，通过 **library** 模拟的 **thread**，不需要或仅需要极少的 **kernel** 支持
- 上下文切换比较快，因为不用更改 **page table** 等，使用起来较为轻便快速
- 提供操控视窗系统的较好的解决方案



用户级的线程库的主要功能

- 创建和销毁线程
- 线程之间传递消息和数据
- 调度线程执行
- 保存和恢复线程上下文
- 典型的例子
 - *POSIX Pthreads*
 - *Mach C-threads*
 - *Java Threads*



POSIX Pthreads

- 用于线程创建和同步的POSIX 标准API (IEEE 1003.1c).
- 可在用户级或者内核级实现.
- API规定了线程库的行为，但不限定实现方法
- 类Unix操作系统中很常见： Solaris, Linux, Mac OS X.

典型的Pthreads API

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

用户级线程的优缺点

■ 优点

- 线程切换与内核无关
- 线程的调度由应用决定，容易进行优化
- 可运行在任何操作系统上，只需要线程库的支持

■ 不足

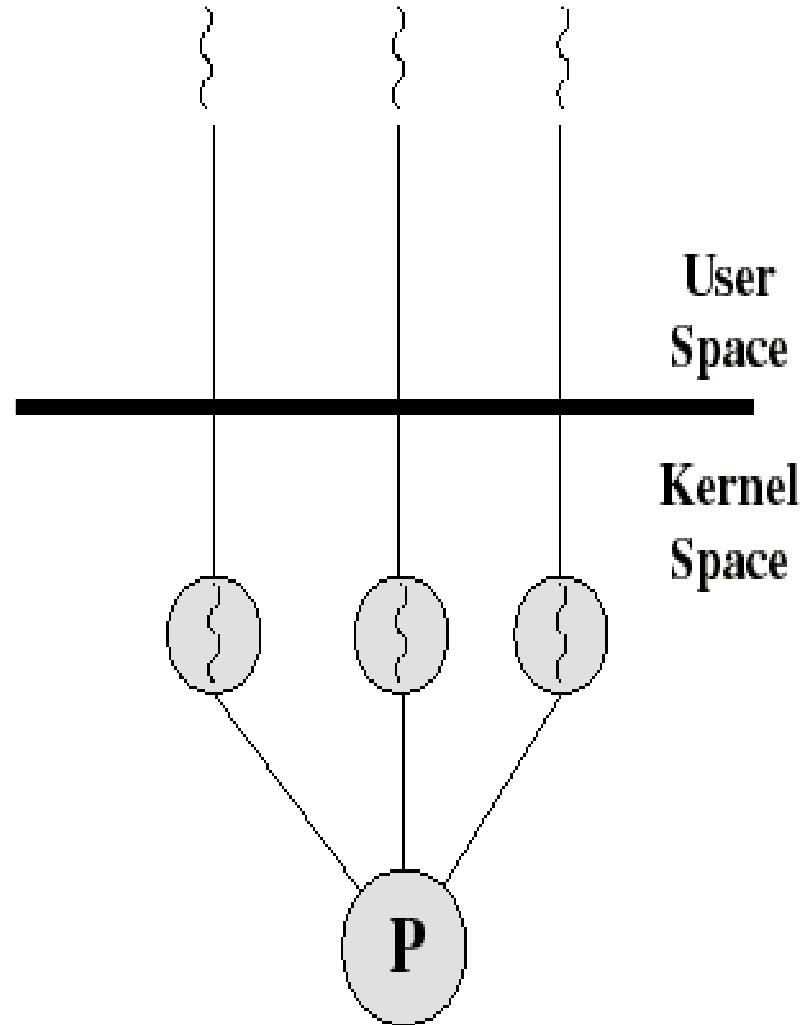
- 很多系统调用会引起阻塞，内核会因此而阻塞所有相关的线程。
- 内核只能将处理器分配给进程，即使有多个处理器，进程也无法实现一个进程中的多个线程的并行执行。

线程的实现方式

- 用户级线程：**User level threads(ULT)**
- 内核级线程：**Kernel level threads (KLT)**
- 混合实现方式

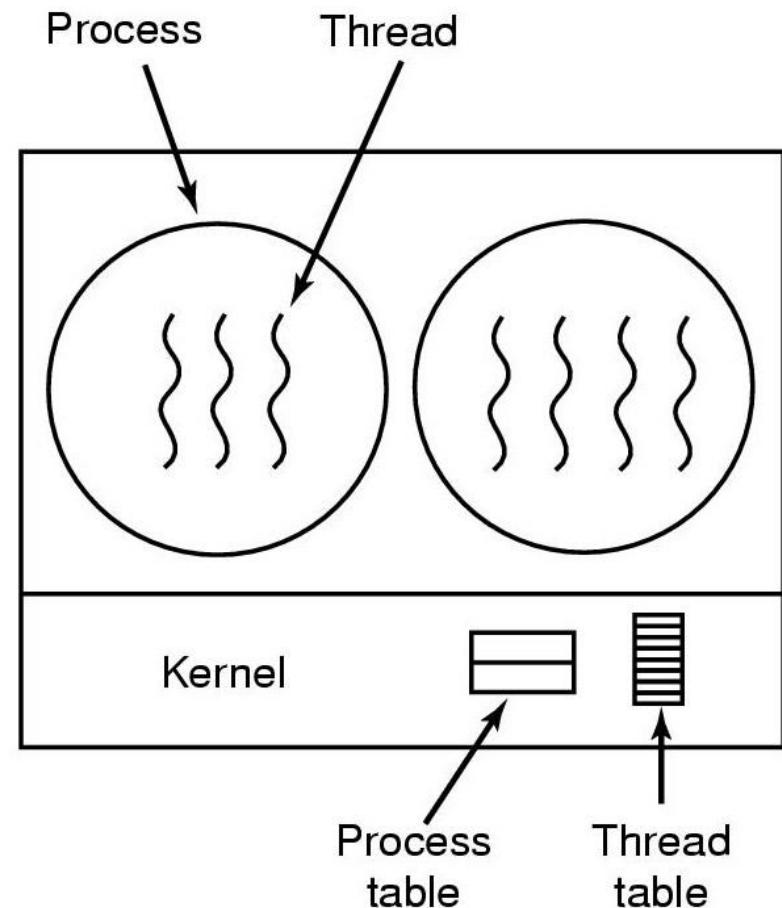
内核级线程

- 内核级线程就是kernel有好几个分身,一个分身可以处理一件事.
- 这用来处理非同步事件很有用,kernel可以对每个非同步事件产生一个分身来处理.
- 支持内核线程的操作系统内核称作多线程内核.



典型实现

- Windows 2000/XP
- OS/2
- Linux
- Solaris
- Tru64 UNIX
- Mac OS X



内核级线程的优缺点

■ 优点

- 内核可以在多个处理器上调度一个进程的多个线程实现同步并行执行
- 阻塞发生在线程级别
- 内核中的一些处理可以通过多线程实现

■ 缺点

- 一个进程中的线程切换需要内核参与，
- 线程的切换涉及到两个模式的切换
 - 进程-进程
 - 线程-线程
- 降低效率

线程操作的延迟

μs

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Source: Anderson, T. et al, “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”, ACM TOCS, February 1992.

用户级线程和内核级线程的比较

- 内核支持线程是OS内核可感知的，而用户级线程是OS内核不可感知的
- 用户级线程的创建、撤销和调度不需要OS内核的支持，是在语言或用户库这一级处理的；而内核支持线程的创建、撤销和调度都需OS内核提供支持，而且与进程的创建、撤销和调度大体是相同的
- 用户级线程执行系统调用指令时将导致其所属进程被中断，而内核支持线程执行系统调用指令时，只导致该线程被中断

用户级线程和内核级线程的总结

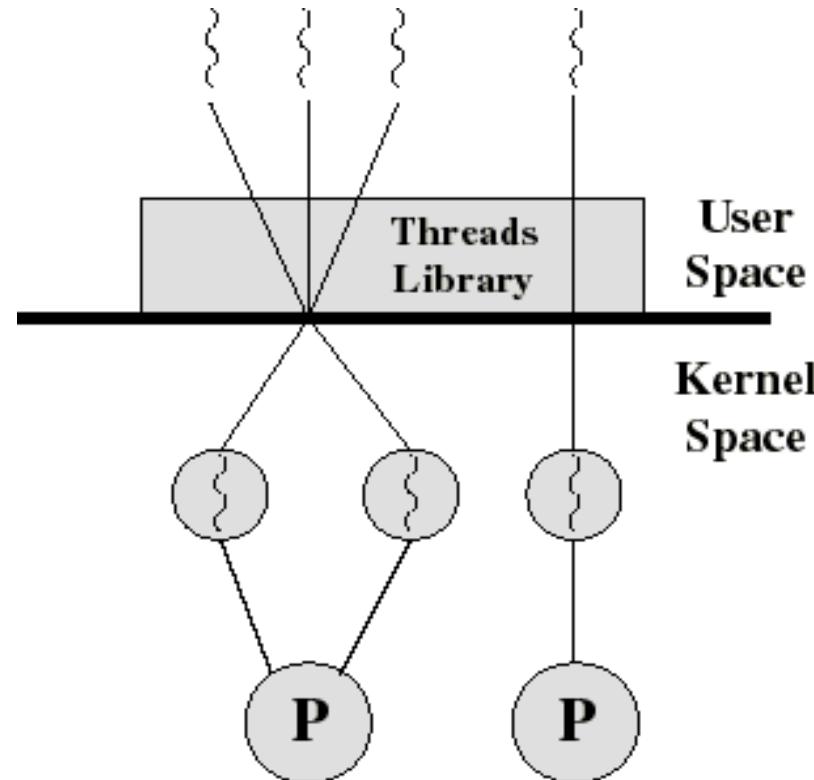
- 在只有用户级线程的系统内，CPU调度还是以进程为单位，处于运行状态的进程中的多个线程，由用户程序控制线程的轮换运行；在有内核支持线程的系统内，CPU调度则以线程为单位，由OS的线程调度程序负责线程的调度
- 用户级线程的程序实体是运行在用户态下的程序，而内核支持线程的程序实体则是可以运行在任何状态下的程序

线程的实现方式

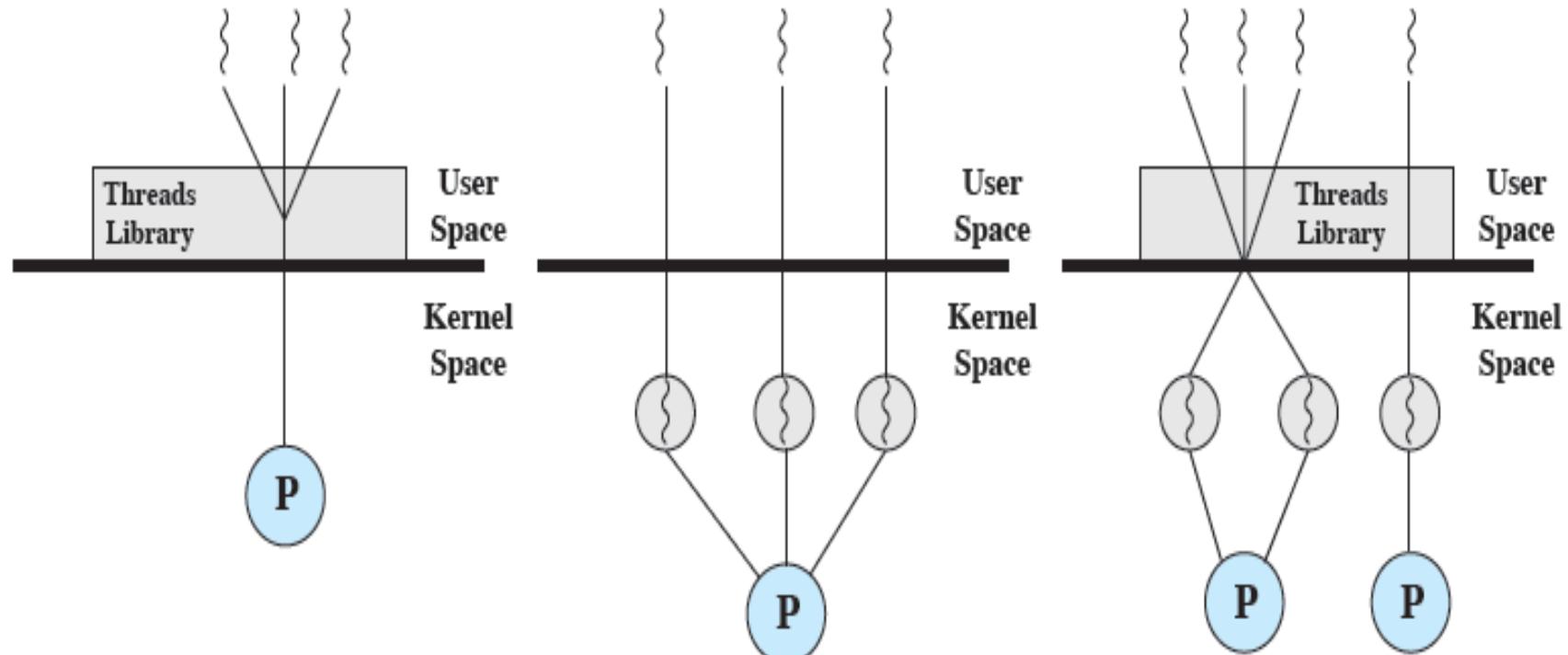
- 用户级线程：**User level threads(ULT)**
- 内核级线程：**Kernel level threads (KLT)**
- 混合实现方式

混合的线程实现方式

- 线程在用户空间创建和管理
- 需要实现从用户空间的线程到内核空间线程（轻量级进程）的映射



三种线程实现方式的对比



(a) Pure user-level

(b) Pure kernel-level

(c) Combined

{ User-level thread

{ Kernel-level thread

P Process

内容提要

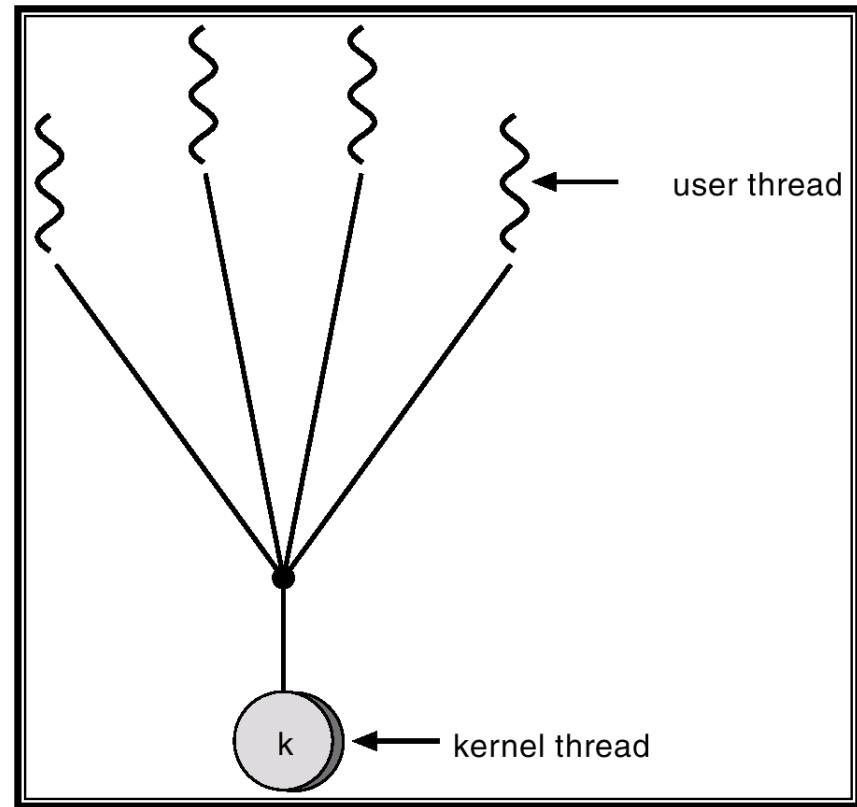
- 进程概念的引入
- 进程状态与控制
- 线程概念的引入
- 线程的实现方式
- 线程模型
- 小结

线程模型

- 有些系统同时支持用户线程和内核线程由此产生了不同的多线程模型，即实现用户级线程和内核级线程的连接方式。
 - Many-to-One
 - One-to-One
 - Many-to-Many

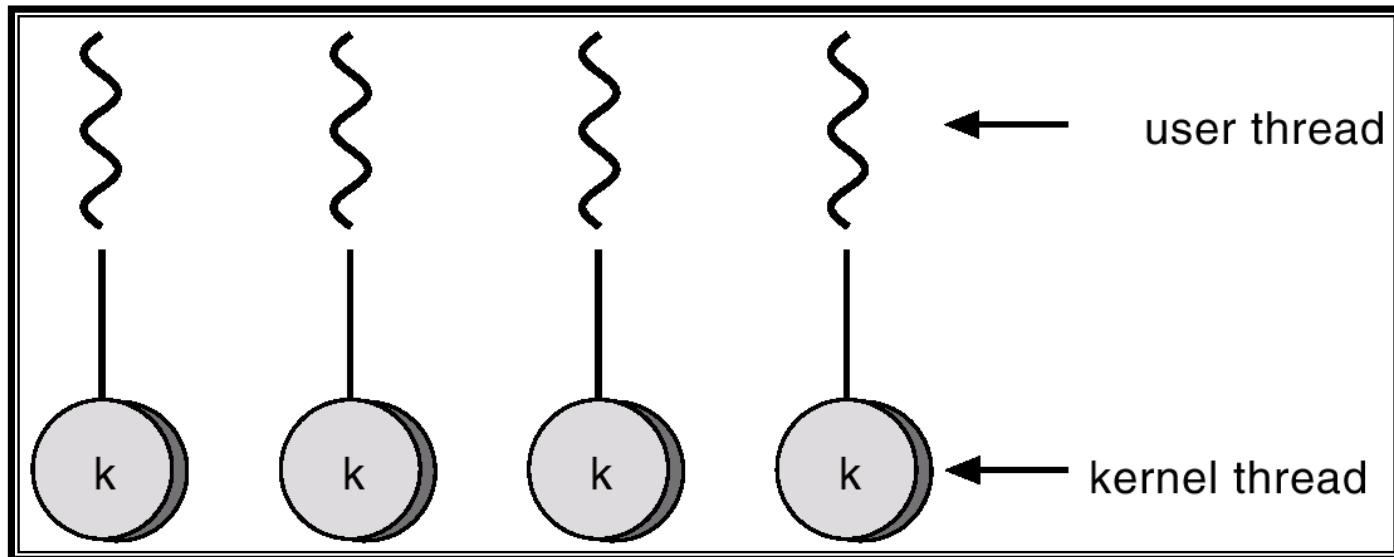
Many-to-One Model

- 将多个用户级线程映射到一个内核级线程，线程管理在用户空间完成。此模式中，用户级线程对操作系统不可见（即透明）。
 - 优点：线程管理是在用户空间进行的，因而效率比较高。
 - 缺点：当一个线程在使用内核服务时被阻塞，那么整个进程都会被阻塞；多个线程不能并行地运行在多处理机上。



One-to-one Model

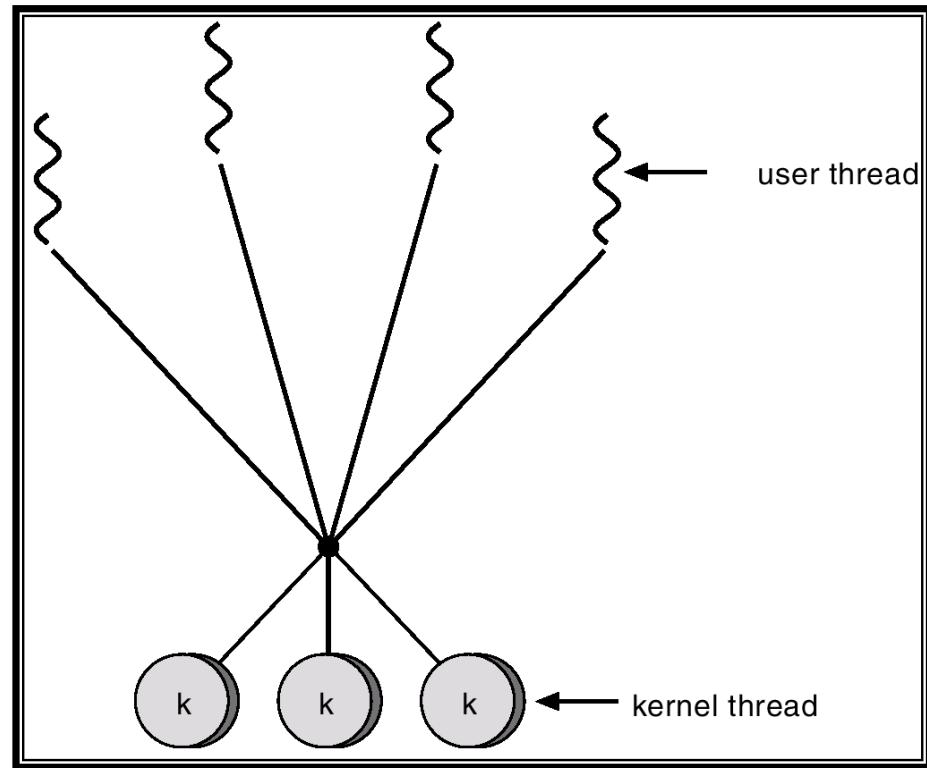
- 将每个用户级线程映射到一个内核级线程。
 - 优点：当一个线程被阻塞后，允许另一个线程继续执行，所以并发能力较强。
 - 缺点：每创建一个用户级线程都需要创建一个内核级线程与其对应，这样创建线程的开销比较大，会影响到应用程序的性能。



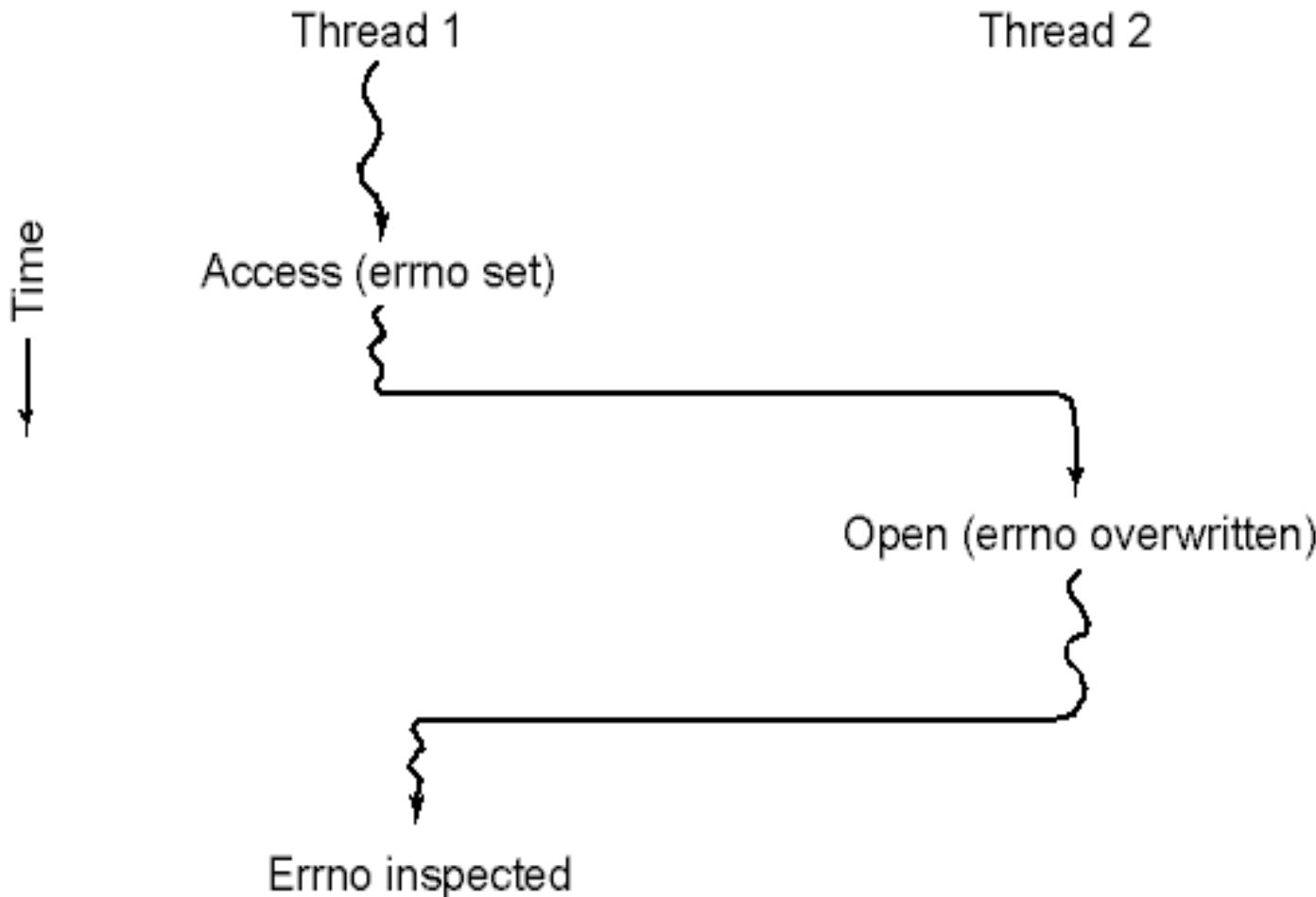
Many-to-Many Model

将 n 个用户级线程映射到 m 个内核级线程上，要求 $m \leq n$ 。

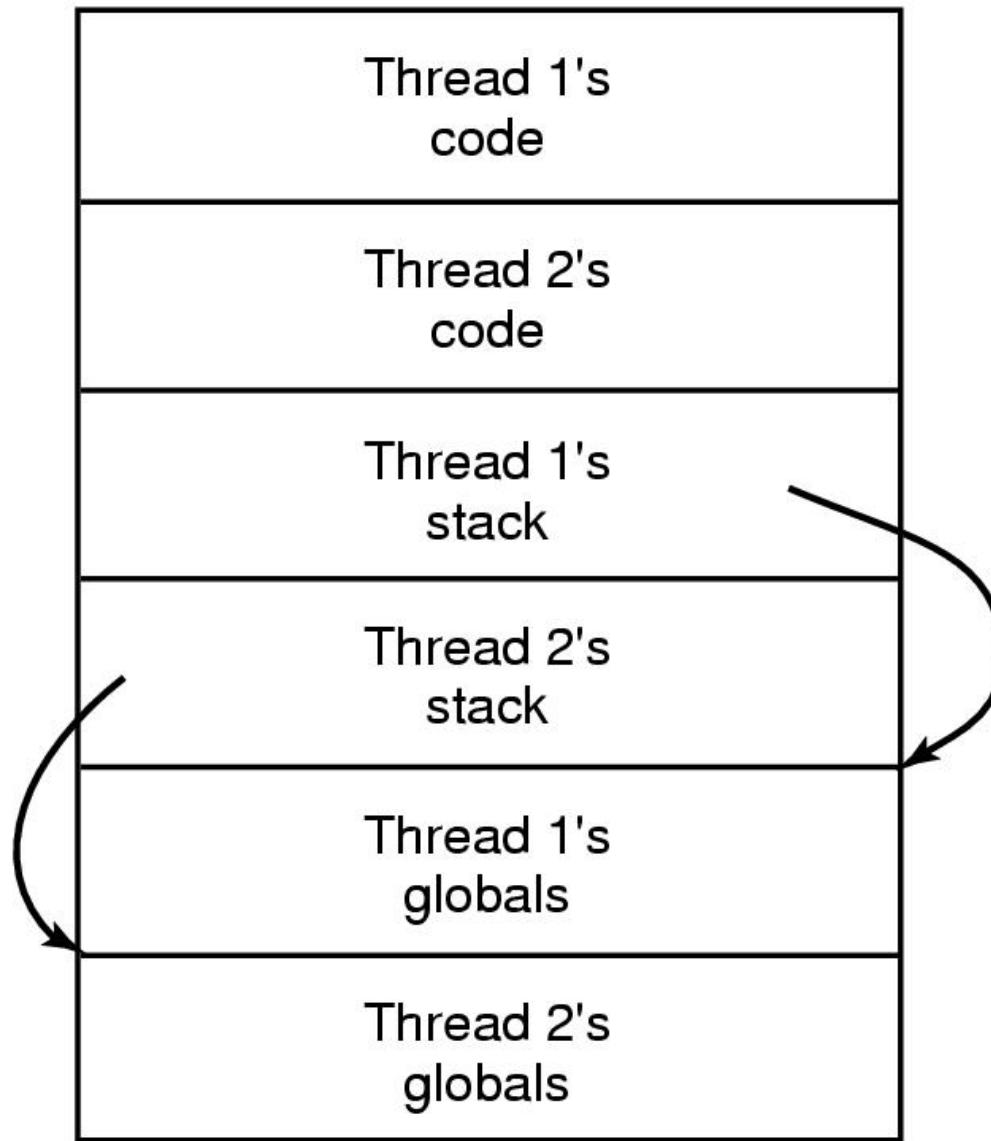
- 特点：在多对一模型和一对多模型中取了个折中，克服了多对一模型的并发度不高的缺点，又克服了一对一模型的一个用户进程占用太多内核级线程，开销太大的缺点。又拥有多对一模型和一对一模型各自的优点，可谓集两者之所长。



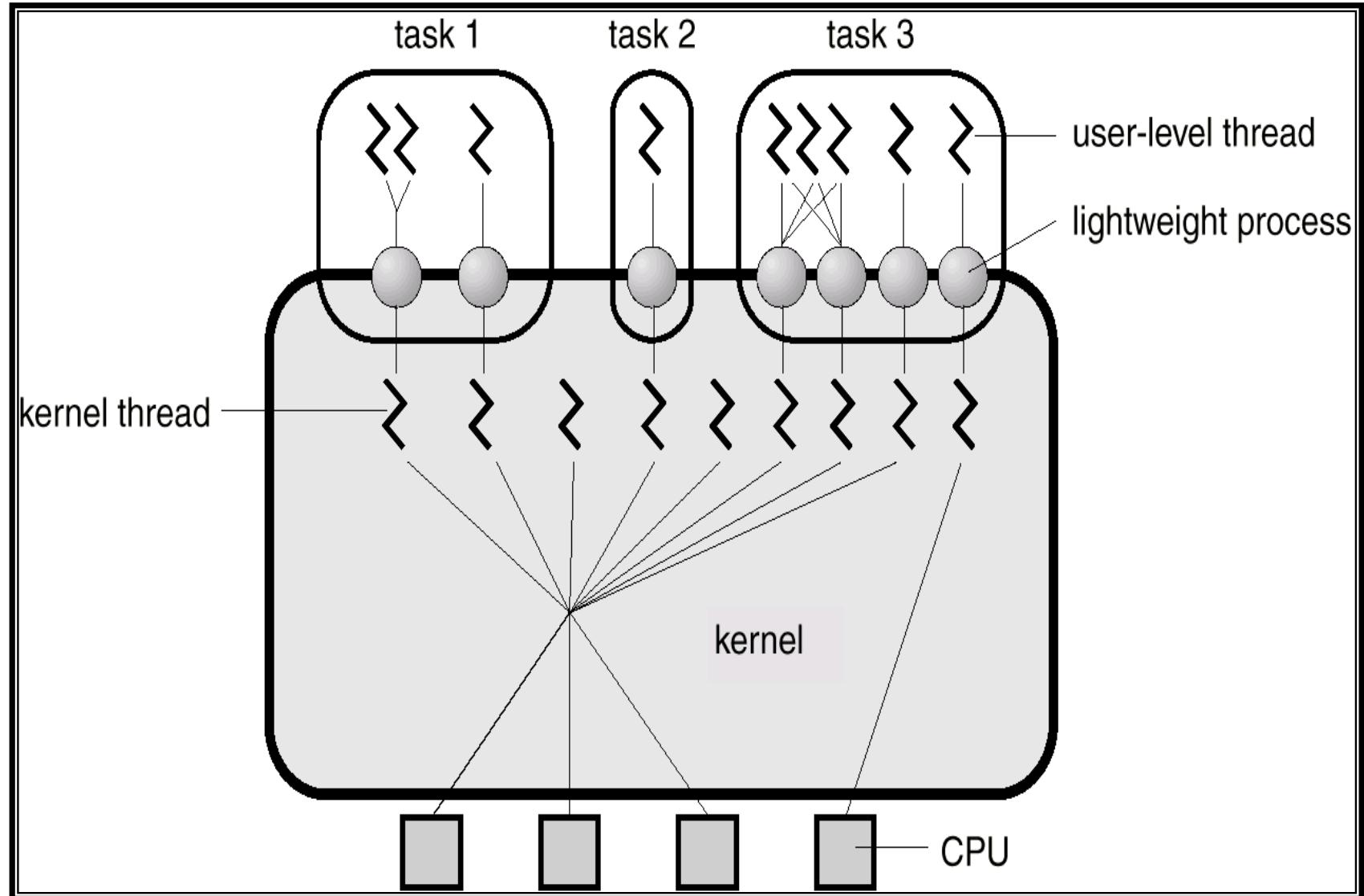
线程编程问题—全局变量



线程局部变量



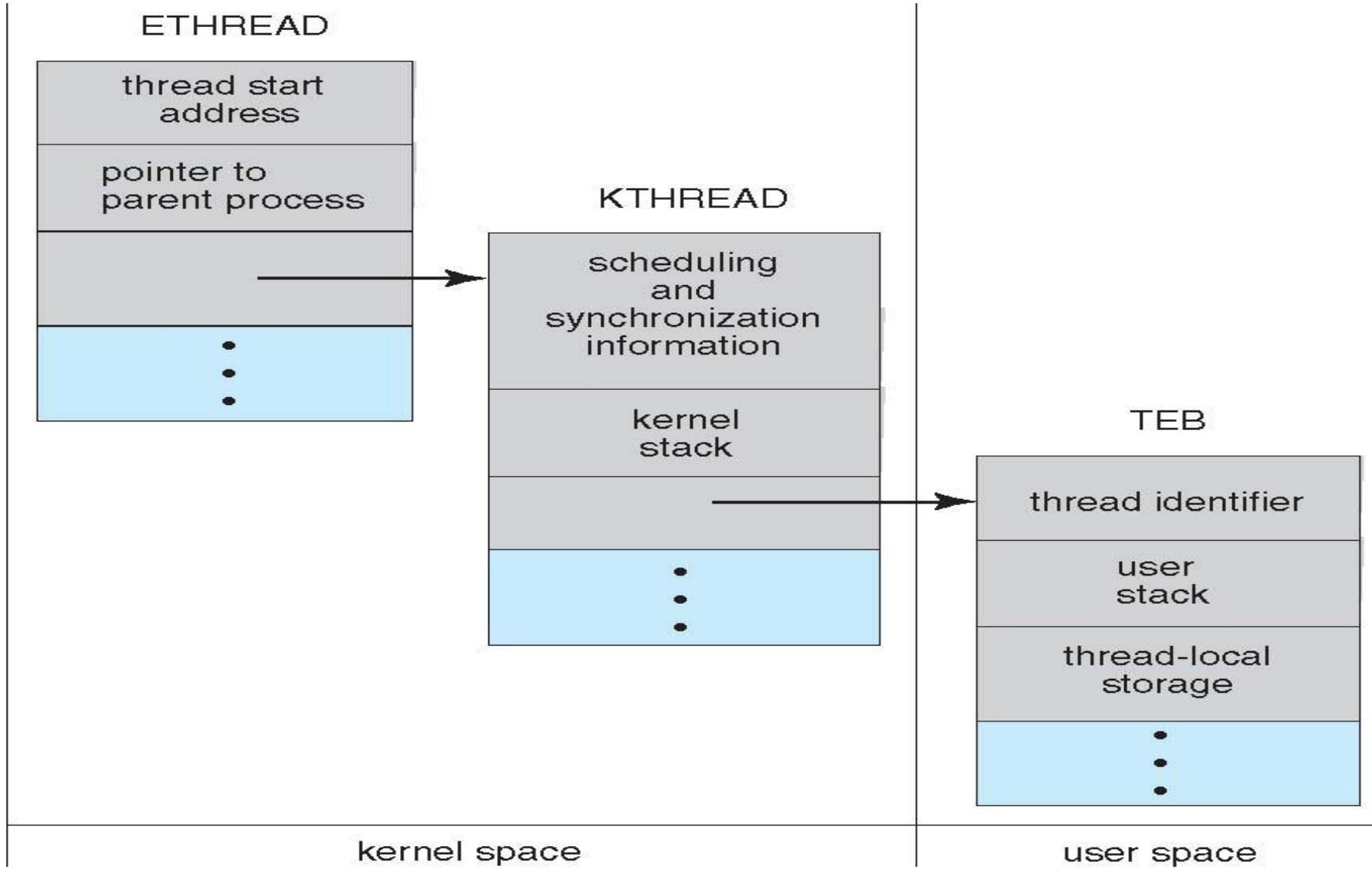
Solaris 2 Threads



Windows XP 线程

- 实现one-to-one模型
 - 每个线程包括的内容
 - 线程id
 - 若干寄存器
 - 用户和内核线程的堆栈
 - 私有的数据区
 - 线程的主要数据结构包括
 - ETHREAD (executive thread block)
 - KTHREAD (kernel thread block)
 - TEB (thread environment block)
- 
- 线程的上下文

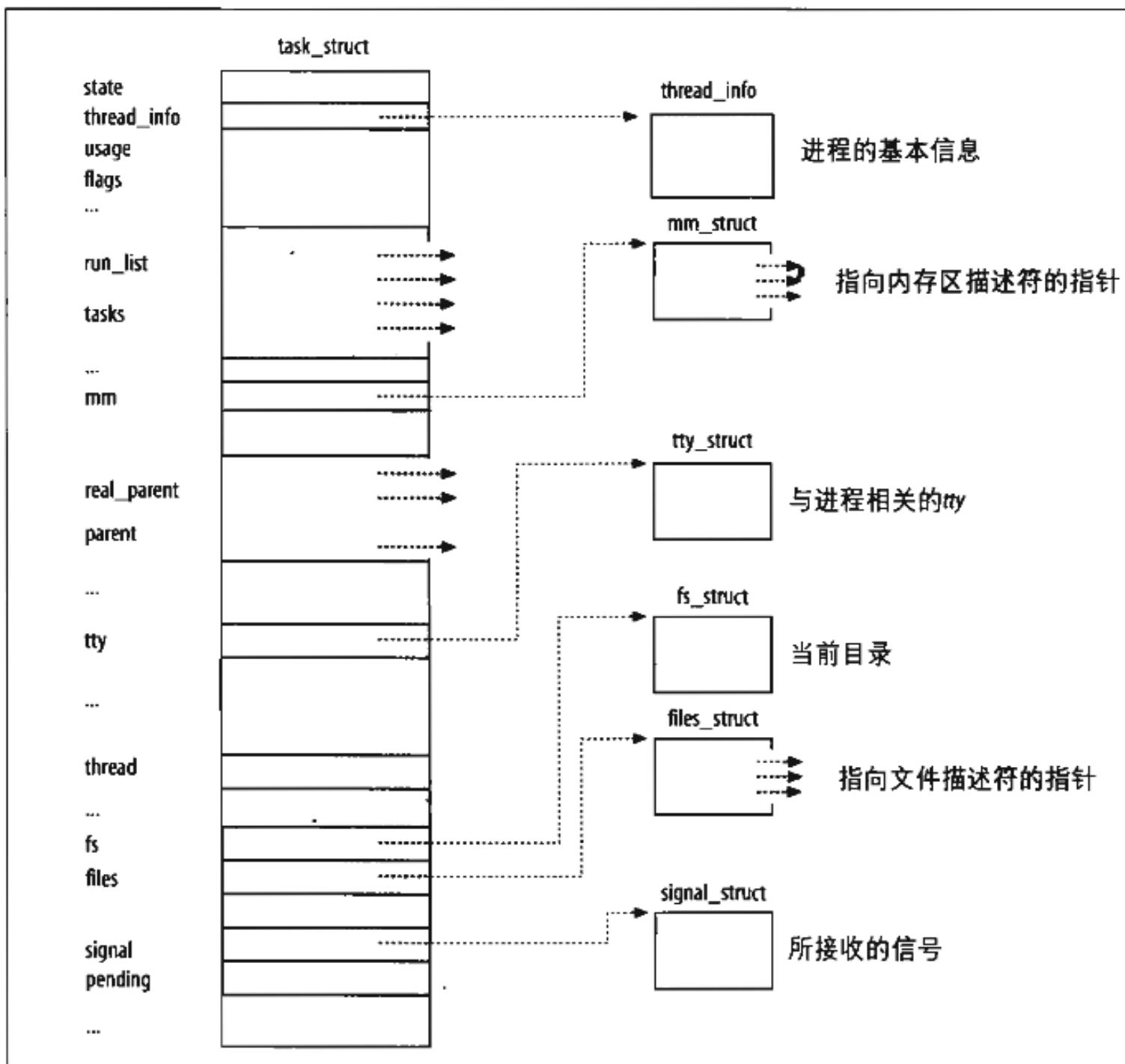
Windows XP Threads



Linux并不确切区分进程与线程

- Linux并不确切区分进程与线程，而将线程定义为“执行上下文”，它实际只是同一个进程的另外一个执行上下文而已。
- 对于调度，仍然可以使用进程的调度程序。
- Linux的内核进程，使用kernel_thread创建，一般被称作内核线程。

Linux 的PCB



fork 和 clone系统调用

- 有两个系统调用建立新进程： fork 与 clone
 - fork一般用以创建普通进程
 - 而 clone 可用以创建线程， kernel_thread便是通过sys_clone来创建新的内核进程
 - fork 与 clone都调用 do_fork 函数执行创建进程的操作
- fork并不指定克隆标志，而clone可由用户指定克隆标志
 - 克隆标志有 CLONE_VM 、 CLONE_FS 、 CLONE_FILES 、 CLONE_SIGHAND 与 CLONE_PID等
 - 而 fork 创建普通进程则使用 SIGCHLD 标志

克隆标志

- CLONE_VM。父子进程共享同一个mm_struct结构，这个克隆标志用以创建一个线程。由于两个进程都使用同一个mm_struct结构，于是这两个进程的指令、数据都共享，也就是将线程视为同一个进程的不同执行上下文。
- CLONE_FS：父子进程共享同一个文件系统。
- CLONE_FILES：父子进程共享打开的文件。
- CLONE_SIGHAND。父子进程共享信号处理句柄。
- CLONE_PID。父子进程共享pid

思考题问题

- 什么情况下不适合用多线程？

小结

- 并发与并行的区别
- 引入进程的目的
- 进程与程序的区别
- 进程的状态与控制
- 引入线程的目的
- 线程与进程的区别
- 线程的实现方法

```
#include <unistd.h>
#include <stdio.h>
int main ()
{
    pid_t fpid; //fpid表示fork函数返回的值
    int count=0;
    fpid=fork();
    if (fpid < 0)
        printf("error in fork!");
    else if (fpid == 0) {
        printf("i am the child process, my process id is %d/n",getpid());
        printf("我是爹的儿子/n");
        count++;
    }
    else {
        printf("i am the parent process, my process id is %d/n",getpid());
        printf("我是孩子他爹/n");
        count++;
    }
    printf("统计结果是: %d/n",count);
    return 0;
}
```

fork()函数 使用举例

fork()函数使用举例

运行结果是：

i am the child process, my process id is 5574

我是爹的儿子

统计结果是: 1

i am the parent process, my process id is 5573

我是孩子他爹

统计结果是: 1

2025第9次课堂小测试

