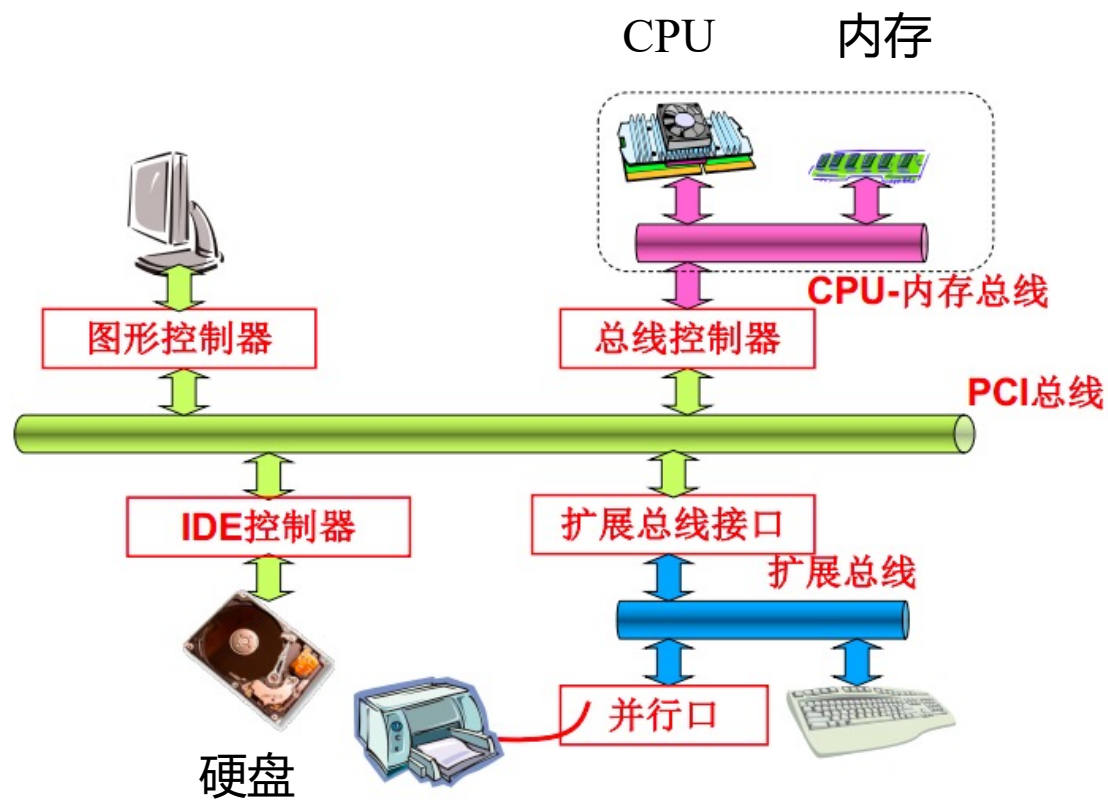
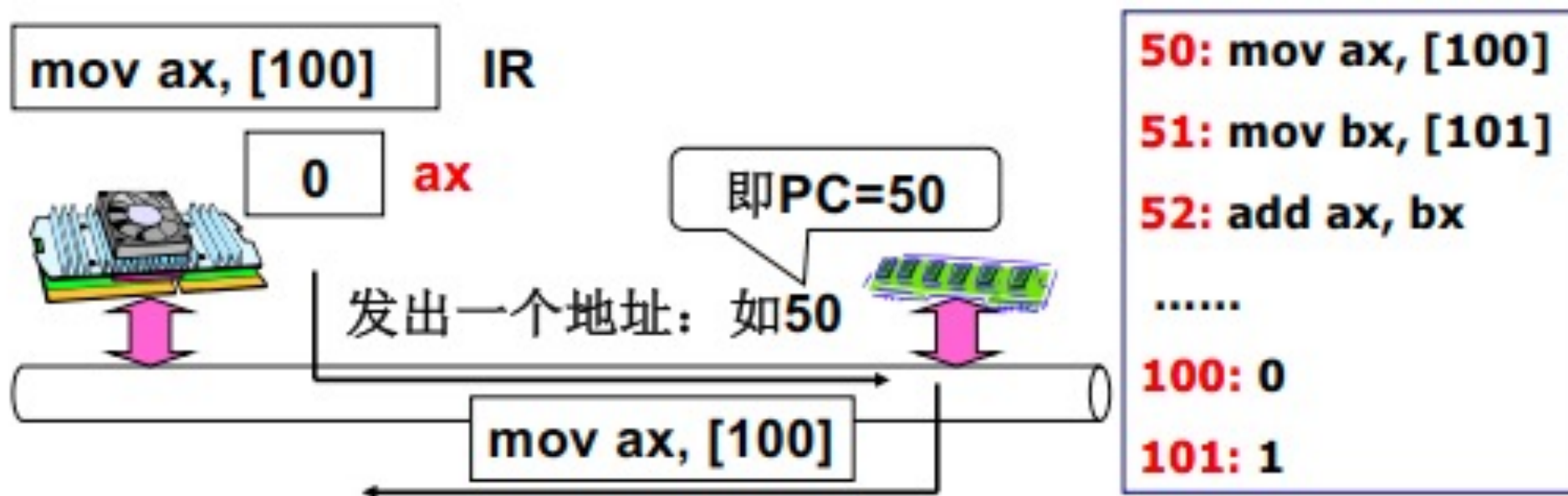


# 第二章 进程的描述与控制



## 第二章 进程的描述与控制



- 操作系统要管理 CPU
- CPU怎么工作？自动取指－执行
- CPU怎么被管理？

## 第二章 进程的描述与控制



```
int main(int argc, char* argv[])
{
    int i, to, *fp, sum = 0;
    to = atoi(argv[1]);
    for(i=1; i<=to; i++)
    {
        sum = sum + i;
        fprintf(fp, "%d", sum);
    }
}
```



fprintf用一条其他计算语句代替

```
C:\>sum 10000000
0.015000 seconds
```

$0.015/10^7$

仅加法运算

有fprintf

```
C:\>sum 1000
0.859000 seconds
```

$0.859/10^3$

仅输出

$5.7 \times 10^5 : 1$

■ I/O 效率

■ CPU计算效率

## 第二章 进程的描述与控制



```
int main(int argc, char* argv[])
{
    int i, to, *fp, sum = 0;
    to = atoi(argv[1]);
    for(i=1; i<=to; i++)
    {
        sum = sum + i;
        fprintf(fp, "%d", sum);
    }
}
```



fprintf用一条其他计算语句代替

```
C:\>sum 10000000
0.015000 seconds
```

$0.015/10^7$

有fprintf

```
C:\>sum 1000
0.859000 seconds
```

$0.859/10^3$

$5.7 \times 10^5 : 1$

$5.7 \times 10^5$   
条计算语句

1  
条I/O语句

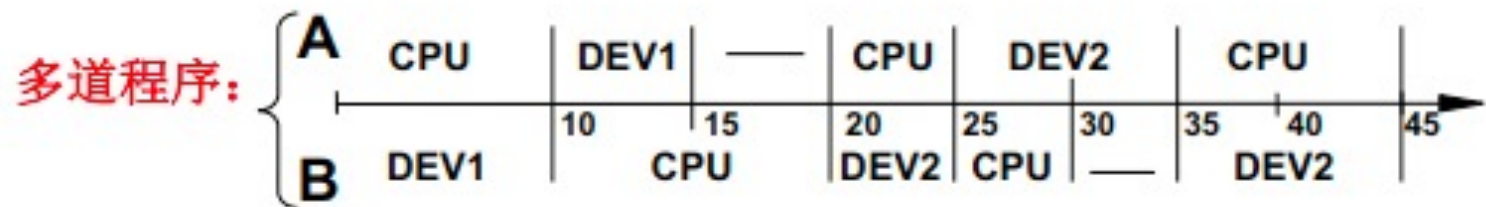
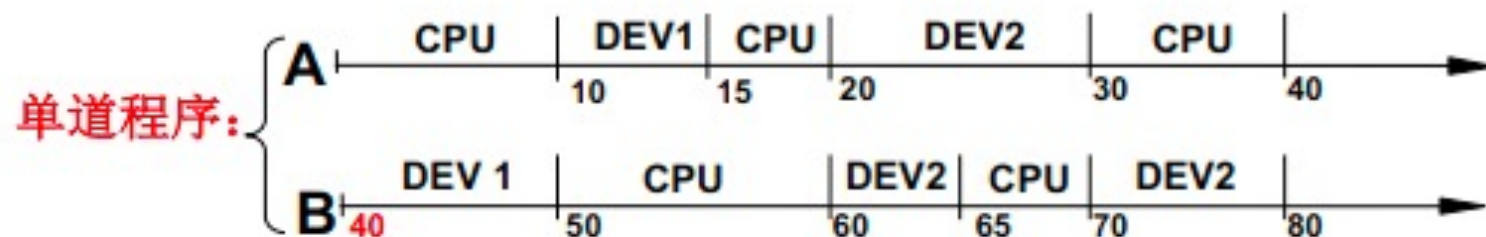
■ I/O 效率

■ CPU计算效率

■ CPU 利用率 ? ? ? ?

■ 如何提高CPU 利用率

## 第二章 进程的描述与控制



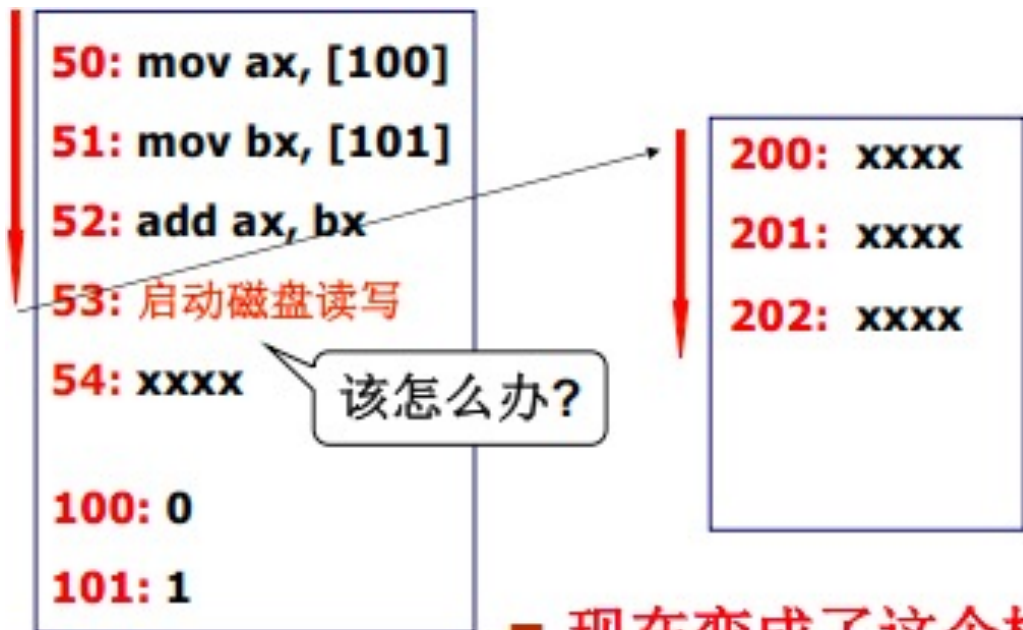
	单道程序	多道程序
CPU利用率	$40/80=50\%$	$40/45=89\%$
DEV1利用率	$15/80=18.75\%$	$15/45=33\%$
DEV2利用率	$25/80=31.25\%$	$25/45=56\%$

■ 进程切换

■ 多道程序同时在内存中

■ 多道程序、交替执行

# 第二章 进程的描述与控制



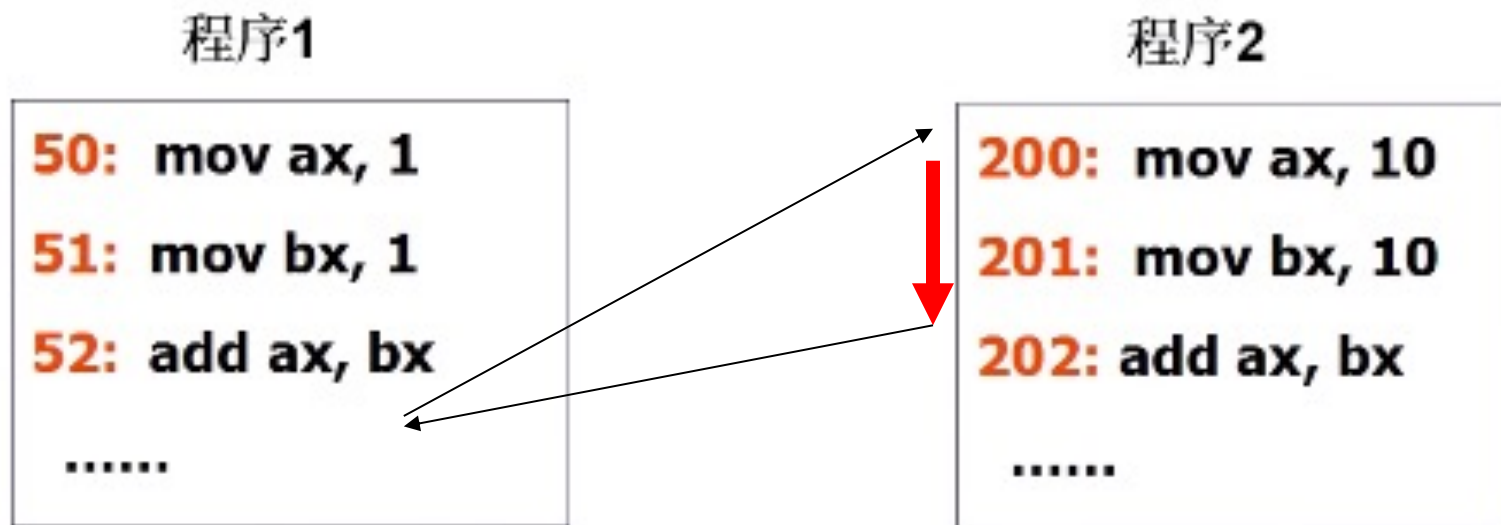
■ 现在变成了这个样子

- 进程切换
- 多道程序同时在内存中
- 多道程序、交替执行





## 第二章 进程的描述与控制



■ 一个CPU上交替的执行多个程序：**并发**

■ **如何并发**



## 第二章 进程的描述与控制



- 操作系统管理 CPU
- 运行单个程序 CPU 利用率低
- 运行多个程序 CPU 利用率高，多个程序交替执行
  - 进程（进行中的程序）
  - 管理好进程 → 管理好CPU
- 核心：如何管理好进程？？





# 第二章 进程的描述与控制



2.1 前趋图和程序执行

2.2 进程的描述

2.3 进程控制

2.4 进程同步

2.5 经典进程的同步问题

2.6 进程通信

2.7 线程(Threads)的基本概念

2.8 线程的实现

习题



## 2.1 前趋图和程序执行



在早期未配置OS的系统和单道批处理系统中，程序的执行方式是**顺序执行**，即在内存中仅装入**一道**用户程序，由它**独占系统中的所有资源**，只有在在一个用户程序执行完成后，才允许装入另一个程序并执行。

人工操作方式

脱机输入/输出方式

单道批处理系统

可见，这种方式浪费资源、系统运行效率低等缺点。



## 2.1 前趋图和程序执行

---

**我们来看看没有进程的概念，只有程序的概念时，情况是如何的？**

## 2.1 前趋图和程序执行

### 2.1.1 前趋图

为了能更好地描述程序的顺序和并发执行情况，我们先介绍用于描述程序执行先后顺序的前趋图。

**所谓前趋图(Precedence Graph)，是指一个有向无循环图，可记为 DAG(Directed Acyclic Graph)，它用于描述进程之间执行的先后顺序。**

图中的每个结点可用来表示一个进程或程序段，乃至一条语句，结点间的有向边则表示两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)。

## 2.1 前趋图和程序执行

程序之间的前趋关系可用“ $\rightarrow$ ”来表示，如果程序 $P_i$ 和 $P_j$ 存在着前趋关系，可表示为 $(P_i, P_j) \in \rightarrow$ ，也可写成 $P_i \rightarrow P_j$ ，表示在 $P_j$ 开始执行之前 $P_i$ 必须完成。此时称 $P_i$ 是 $P_j$ 的直接前趋，而称 $P_j$ 是 $P_i$ 的直接后继。在前趋图中，把没有前趋的结点称为初始结点(Initial Node)，把没有后继的结点称为终止结点(Final Node)。此外，每个结点还具有一个重量(Weight)，用于表示该结点所含有的程序量或程序的执行时间。

## 2.1 前趋图和程序执行

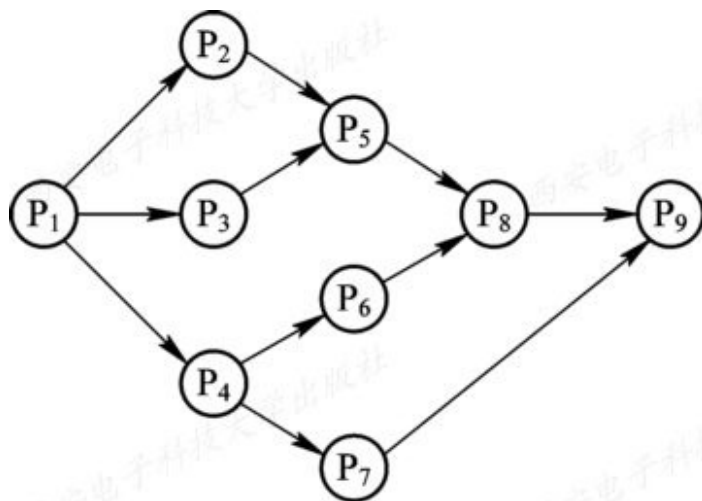
在图2-1(a)所示的前趋图中，存在着如下前趋关系：

$P_1 \rightarrow P_2$  ,  $P_1 \rightarrow P_3$  ,  $P_1 \rightarrow P_4$  ,  $P_2 \rightarrow P_5$  ,  $P_3 \rightarrow P_5$  ,  $P_4 \rightarrow P_6$  ,  $P_4 \rightarrow P_7$  ,  $P_5 \rightarrow P_8$  ,  $P_6 \rightarrow P_8$  ,  $P_7 \rightarrow P_9$  ,  $P_8 \rightarrow P_9$

或表示为

$P = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$

$= \{(P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_5), (P_3, P_5), (P_4, P_6), (P_4, P_7), (P_5, P_8), (P_6, P_8), (P_7, P_9), (P_8, P_9)\}$



(a) 具有九个结点的前趋图



(b) 具有循环的前趋图

图2-1 前趋图

## 2.1 前趋图和程序执行

应当注意，前趋图中是不允许有循环的，否则必然会产生不可能实现的前趋关系。如图2-1(b)所示的前趋关系中就存在着循环。它一方面要求在 $S_3$ 开始执行之前， $S_2$ 必须完成，另一方面又要求在 $S_2$ 开始执行之前， $S_3$ 必须完成。显然，这种关系是不可能实现的。

$$S_2 \rightarrow S_3, S_3 \rightarrow S_2$$



## 2.1 前趋图和程序执行

### 2.1.2 程序顺序执行

#### 1. 程序的顺序执行

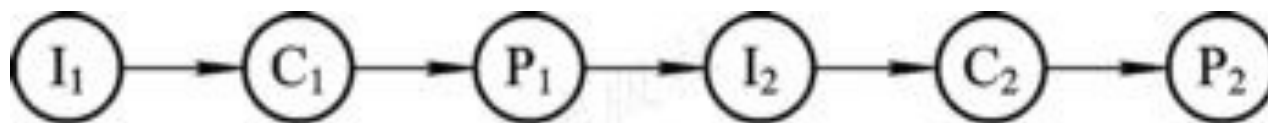
通常，一个应用程序由若干个程序段组成，每一个程序段完成特定的功能，它们在执行时，都需要按照某种先后次序顺序执行，仅当前一程序段执行完后，才运行后一程序段。

例如，在进行计算时，应先运行**输入程序**，用于输入用户的程序和数据；  
然后运行**计算程序**，对所输入的数据进行计算；  
最后才是运行**打印程序**，打印计算结果。

## 2.1 前趋图和程序执行

我们用结点(Node)代表各程序段的操作(在图2-1中用圆圈表示), 其中I代表输入操作, C代表计算操作, P为打印操作, 用箭头指示操作的先后次序。

这样, 上述的三个程序段间就存在着这样的前趋关系:  $I_1 \rightarrow C_1 \rightarrow P_1$ , 其执行的顺序可用前趋图2-2(a)描述。



(a) 程序的顺序执行

## 2.1 前趋图和程序执行

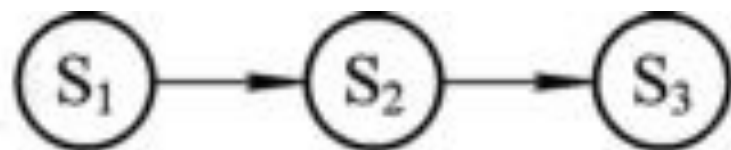
即使是一个程序段，也可能存在着执行顺序问题，下面示出了一个包含了三条语句的程序段：

$S_1: a := x + y;$

$S_2: b := a - 5;$

$S_3: c := b + 1;$

其中，语句 $S_2$ 必须在语句 $S_1$ 后(即 $a$ 被赋值)才能执行，语句 $S_3$ 也只能在 $b$ 被赋值后才能执行，因此，三条语句存在着这样的前趋关系： $S_1 \rightarrow S_2 \rightarrow S_3$ ，应按前趋图2-2(b)所示的顺序执行。



(b) 三条语句的顺序执行

## 2.1 前趋图和程序执行

### 2.1.2 程序顺序执行

#### 2. 程序顺序执行时的特征

由上所述可以得知，在程序顺序执行时，具有这样三个特征：

- ① **顺序性**：指处理机严格地按照程序所规定的顺序执行，即每一操作必须在下一个操作开始之前结束；
- ② **封闭性**：指程序在封闭的环境下运行，即程序运行时独占全机资源，资源的状态(除初始状态外)只有本程序才能改变它，程序一旦开始执行，其执行结果不受外界因素影响；
- ③ **可再现性**：指只要程序执行时的环境和初始条件相同，当程序重复执行时，不论它是从头到尾不停顿地执行，还是“停停走走”地执行，都可获得相同的结果。程序顺序执行时的这种特性，为程序员检测和校正程序的错误带来了很大的方便。

## 2.1 前趋图和程序执行

### 2.1.3 程序并发执行

#### 1. 程序的并发执行

我们通过一个常见的例子来说明程序的顺序执行和并发执行。在图2-2中的输入程序、计算程序和打印程序三者之间，存在着 $I_i \rightarrow C_i \rightarrow P_i$ 这样的前趋关系，以至对一个作业的输入、计算和打印三个程序段必须顺序执行。但若是对一批作业进行处理时，每道作业的输入、计算和打印程序段的执行情况如图2-3所示。

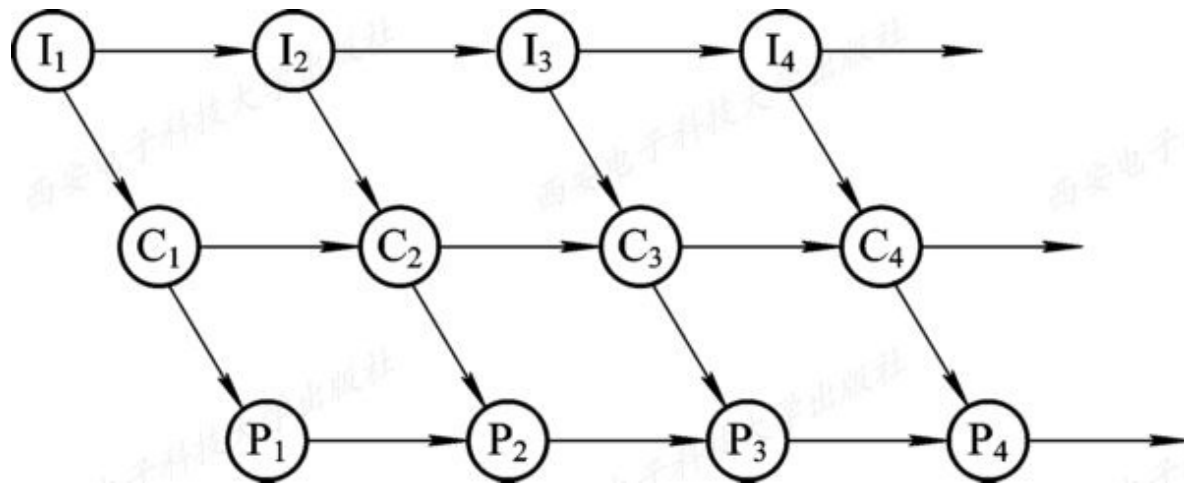


图2-3 程序并发执行时的前趋图

## 2.1 前趋图和程序执行

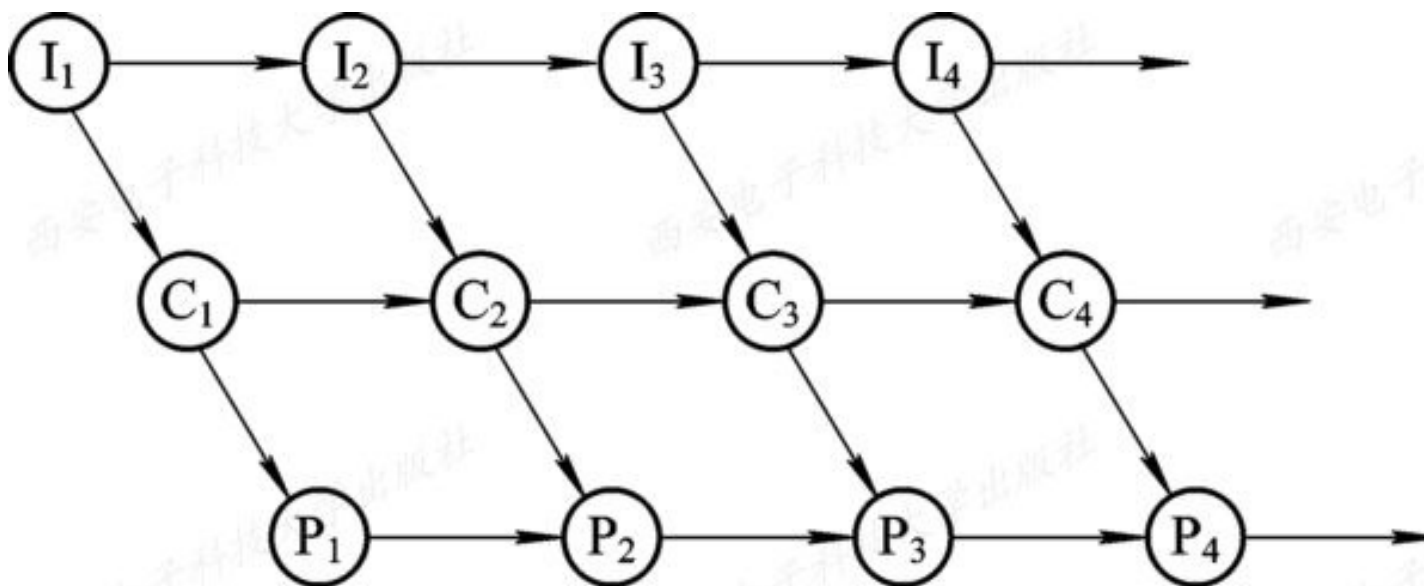


图2-3 程序并发执行时的前趋图

由图2-3可以看出，存在前趋关系 $I_i \rightarrow C_i$ ， $I_i \rightarrow I_{i+1}$ ， $C_i \rightarrow P_i$ ， $C_i \rightarrow C_{i+1}$ ， $P_i \rightarrow P_{i+1}$ ，而 $I_{i+1}$ 和 $C_i$ 及 $P_{i-1}$ 是重叠的，即在 $P_{i-1}$ 和 $C_i$ 以及 $I_{i+1}$ 之间，不存在前趋关系，可以并发执行。

## 2.1 前趋图和程序执行

对于具有下述四条语句的程序段：

$S_1: a := x + 2$

$S_2: b := y + 4$

$S_3: c := a + b$

$S_4: d := c + b$

可画出图2-4所示的前趋关系。可以看出： $S_3$ 必须在 $a$ 和 $b$ 被赋值后方能执行； $S_4$ 必须在 $S_3$ 之后执行；但 $S_1$ 和 $S_2$ 则可以并发执行，因为它们彼此互不依赖。

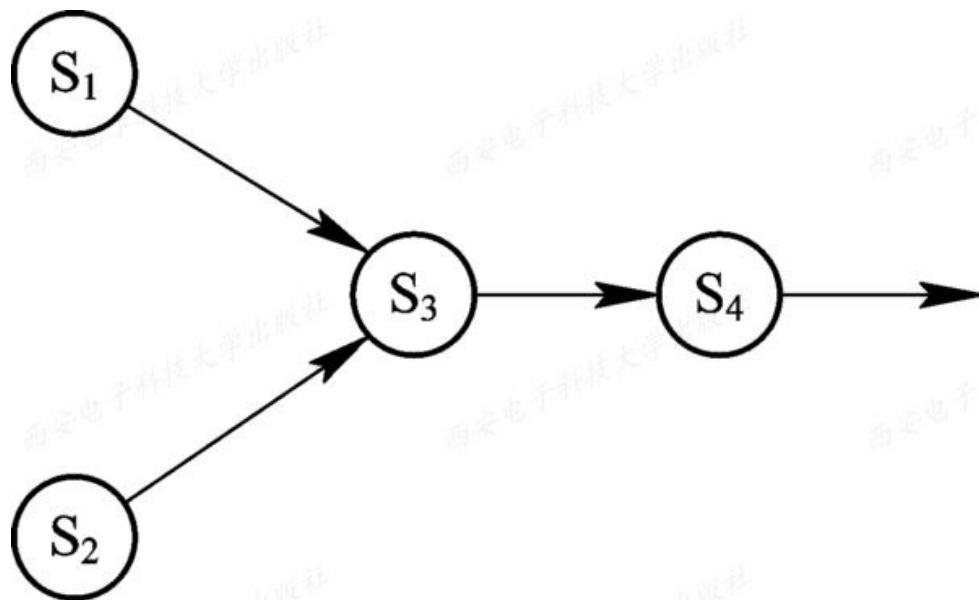


图2-4 四条语句的前趋关系



## 2.1 前趋图和程序执行

### 2.1.3 程序并发执行

#### 2. 程序并发执行时的特征

在引入了程序间的并发执行功能后，虽然提高了系统的吞吐量和资源利用率，但由于它们共享系统资源，以及它们为完成同一项任务而相互合作，致使在这些并发执行的程序之间必将形成相互制约的关系，由此会给程序并发执行带来新的特征。

- (1) 间断性。 走走停停执行。
- (2) 失去封闭性。 不独占全机。
- (3) 不可再现性。

**怎样可再现？？？？操作系统的异步性。**

## 2.1 前趋图和程序执行

- 运行的程序 和 静态程序
- 进程：进行中的程序
  - 进程有开始、有结束，程序没有
  - 进程会走走停停，程序没有
  - 进程需要记录现场信息（ax、bx），程序不用
- 不同的信息存放于PCB中



# 第二章 进程的描述与控制



2.1 前趋图和程序执行

2.2 进程的描述

2.3 进程控制

2.4 进程同步

2.5 经典进程的同步问题

2.6 进程通信

2.7 线程(Threads)的基本概念

2.8 线程的实现

习题

## 2.2 进程的描述

### ■ 如何使用CPU？

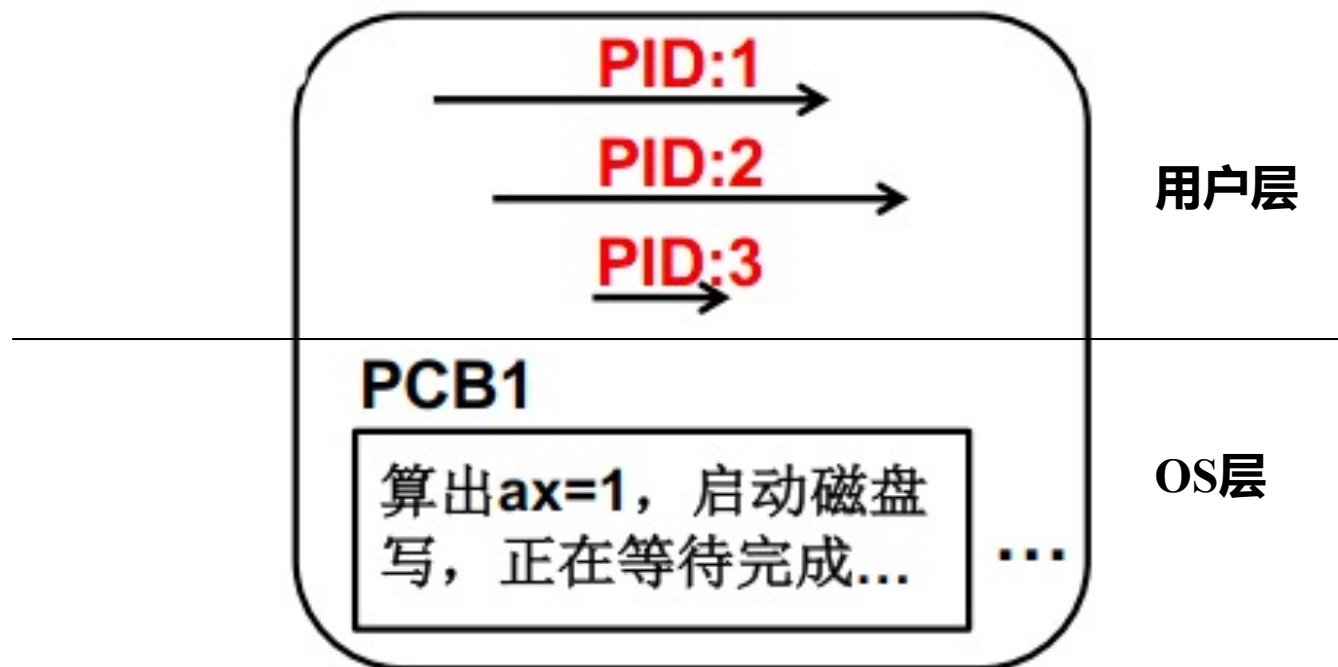
#### ■ 让程序运行起来

### ■ 如何充分使用CPU？

#### ■ 启动多个程序，交替执行

### ■ 启动了的程序就是进程，所以是多进程推进

#### ■ 操作系统记录好、按照合理的次序推进进程（分配资源、调度资源）



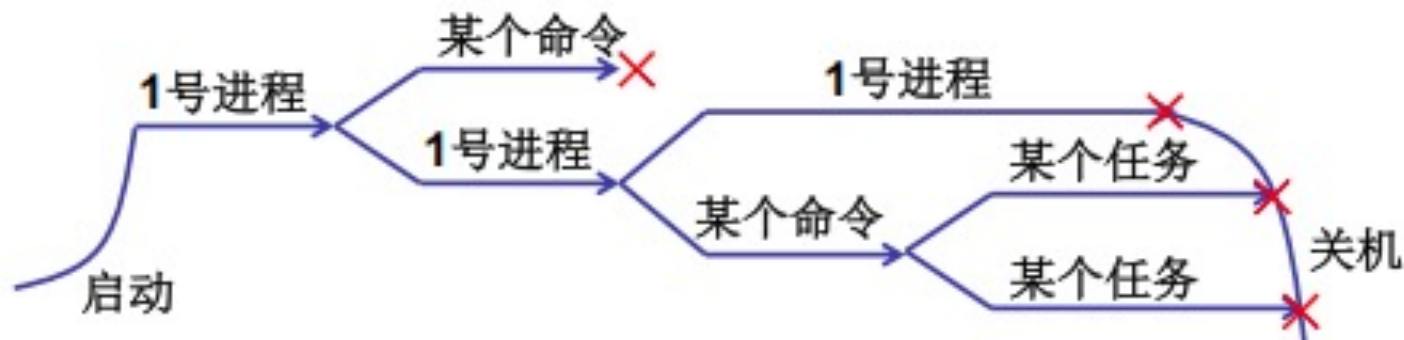
## 2.2 进程的描述

### ■ 进程的管理

```
login as: tnp
tnp@192.168.1.1's password:
Last login: Mon Nov 13 23:47:50 2006 from 192.168.1.2
[tnp@tevmull ~]$ ls
work.txt
[tnp@tevmull ~]$
```

Shell

### ■ Linux init 创建 shell , shell创建其它进程



## 2.2 进程的描述

### 2.2.1 进程的定义和特征

#### 1. 进程的定义

在多道程序环境下，程序的执行属于并发执行，此时它们将失去其封闭性，并具有间断性，以及其运行结果不可再现性的特征。

由此，决定了通常的程序是不能参与并发执行的，否则，程序的运行也就失去了意义。

为了能使程序并发执行，并且可以对并发执行的程序加以描述和控制，**人们引入了“进程”的概念。**

## 2.2 进程的描述

---

### 2.2.1 进程的定义和特征

#### 1. 进程的定义

进程实体（进程映像）：**程序段、相关数据段、PCB**



## 2.2 进程的描述

### 2.2.1 进程的定义和特征

#### 1. 进程的定义

对于进程的定义，从不同的角度可以有不同的定义，其中较典型的定义有：

(1) 进程是程序的一次执行。

(2) 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。

(3) 进程是具有独立功能的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。

## 2.2 进程的描述

### 2.2.1 进程的定义和特征

#### 2. 进程的特征

进程和程序是两个截然不同的概念，除了进程具有程序所没有的PCB结构外，还具有下面一些特征：

- (1) **动态性**。由创建而产生，由调度而执行，由撤销而消亡。
- (2) **并发性**。多道进程于同一时间段执行。
- (3) **独立性**。独立运行、独立获得资源和独立接受调度的单位。
- (4) **异步性**。以不可预知的速度推进，执行结果相同。

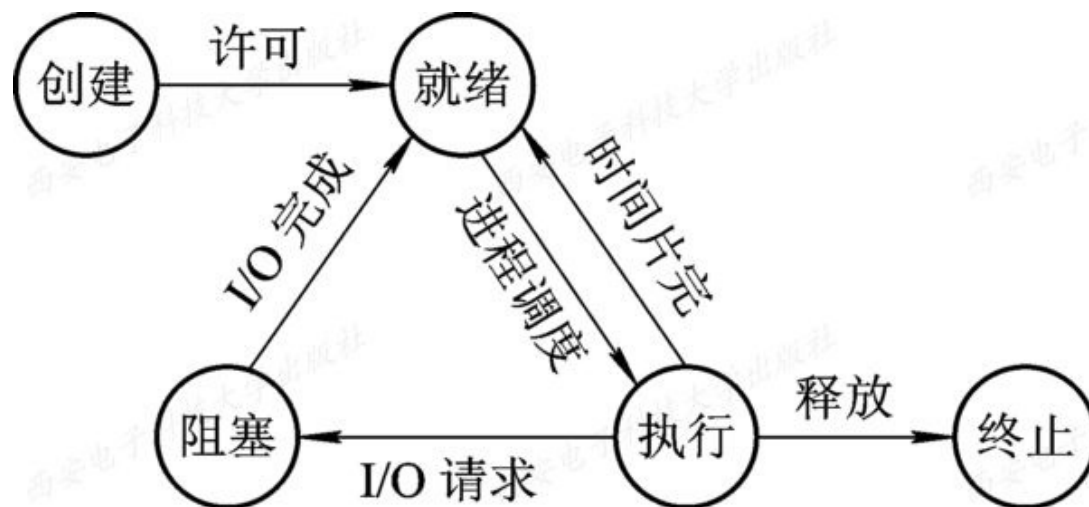
## 2.2 进程的描述

### 2.2.2 进程的基本状态及转换

#### 1. 进程的三种基本状态

由于多个进程在并发执行时共享系统资源，致使它们在运行过程中呈现间断性的运行规律，所以进程在其生命周期内可能具有多种状态。一般而言，每一个进程至少应处于以下三种基本状态之一：

- (1) 就绪(Ready)状态。
- (2) 执行(Running)状态。
- (3) 阻塞(Block)状态。

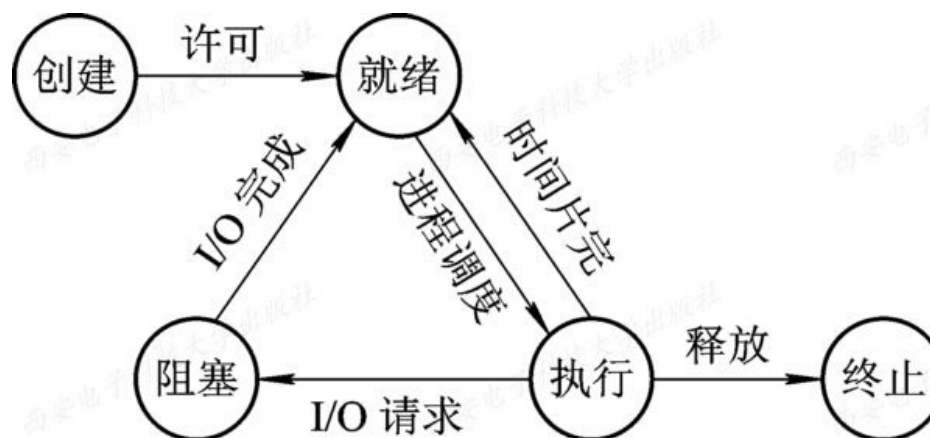


## 2.2 进程的描述

### 2.2.2 进程的基本状态及转换

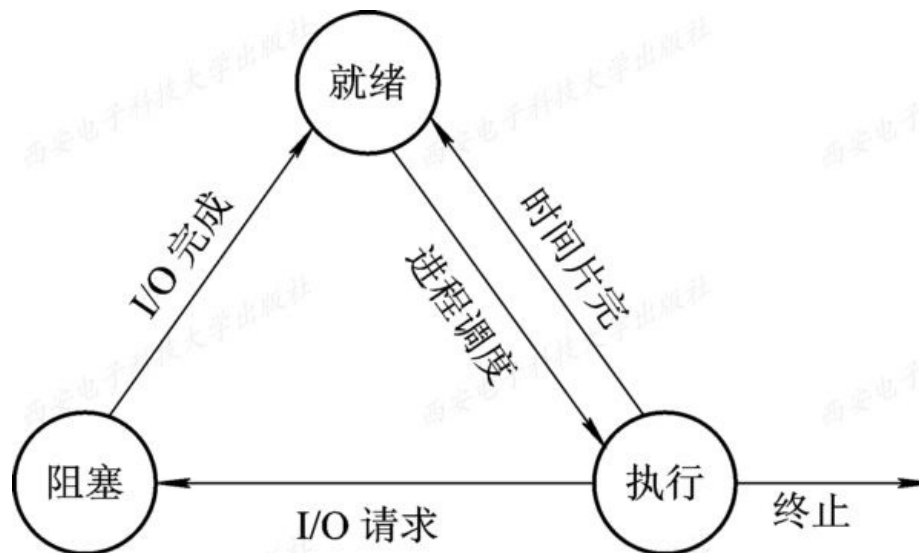
#### 2. 三种基本状态的转换

进程在运行过程中会经常发生状态的转换。例如，处于就绪状态的进程，在调度程序为之分配了处理机之后便可执行，相应地，其状态就由就绪态转变为执行态；正在执行的进程(当前进程)如果因分配给它的时间片已完而被剥夺处理机暂停执行时，其状态便由执行转为就绪；如果因发生某事件，致使当前进程的执行受阻(例如进程访问某临界资源，而该资源正被其它进程访问时)，使之无法继续执行，则该进程状态将由执行转变为阻塞。图2-5示出了进程的三种基本状态，以及各状态之间的转换关系。



## 2.2 进程的描述

- 运行→等待；运行→就绪；就绪→运行……



- 该图称为**进程状态图**
- 它能给出进程生存期的清晰描述
- 它是认识操作系统进程管理的一个窗口

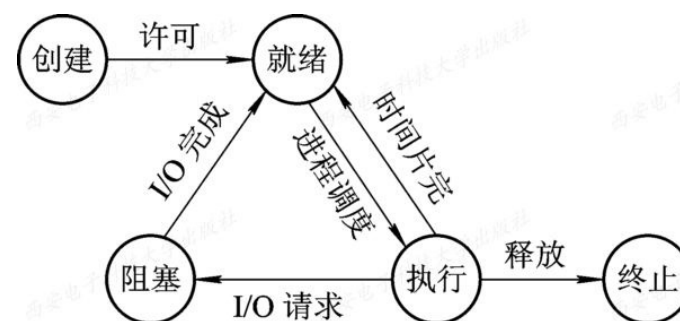
## 2.2 进程的描述

### 2.2.2 进程的基本状态及转换

#### 3. 创建状态和终止状态

##### 1) 创建状态

如前所述，进程是由创建而产生。创建一个进程是个很复杂的过程，一般要通过多个步骤才能完成：如首先由进程**申请一个空白PCB**，并向PCB中填写用于控制和管理进程的信息；然后为该进程**分配运行时所必须的资源**；最后，把该进程转入就绪状态并插入**就绪队列**之中。但如果进程所需的资源尚不能得到满足，**比如系统尚无足够的内存使进程无法装入其中**，此时创建工作尚未完成，进程不能被调度运行，于是把此时进程所处的状态称为**创建状态**。



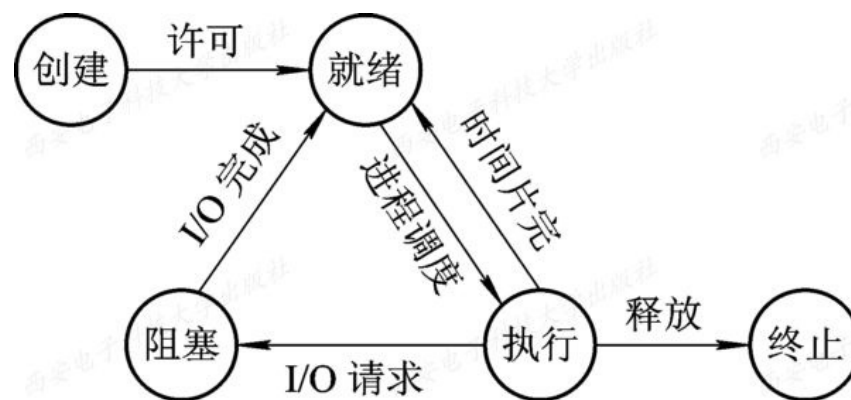
## 2.2 进程的描述

### 2.2.2 进程的基本状态及转换

#### 3. 创建状态和终止状态

##### 2) 终止状态

进程的终止也要通过两个步骤：首先，是等待操作系统进行善后处理，最后**将其PCB清零**，**并将PCB空间返还系统**。





## 2.2 进程的描述

### 2.2.2 进程的基本状态及转换

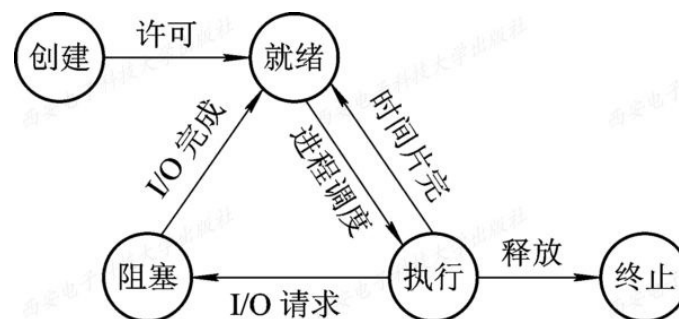
#### 3.创建状态和终止状态

##### 2) 终止状态

进程的终止也要通过两个步骤：首先，是等待操作系统进行善后处理，最后**将其PCB清零**，并**将PCB空间返还系统**。

#### 进入终止态的情况

- 当一个进程到达了**自然结束点**
- 出现了**无法克服的错误**
- 被**操作系统所终结**
- 被其他有**终止权的进程所终结**



## 2.2 进程的描述

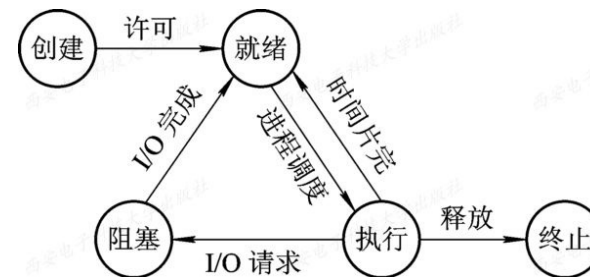
### 2.2.2 进程的基本状态及转换

#### 3.创建状态和终止状态

##### 2) 终止状态

进程的终止也要通过两个步骤：首先，是等待操作系统进行善后处理，最后**将其PCB清零**，并**将PCB空间返还系统**。

进入终止态的进程以后不能再执行，但在操作系统中依然保留一个记录，其中保存状态码和一些计时统计数据，供其他进程收集。一旦**其他进程完成了对其信息的提取之后**，操作系统将删除该进程，即将其PCB清零，并将该空白PCB返还系统。



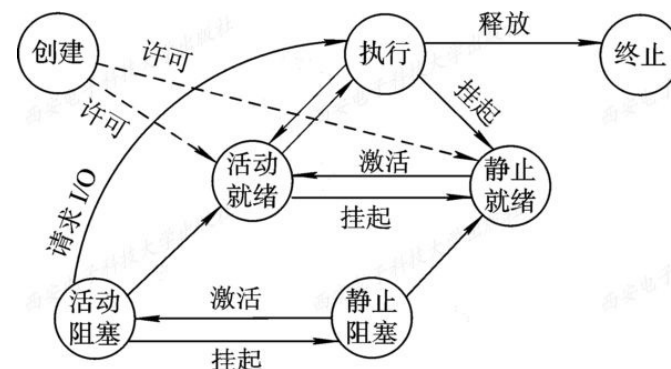
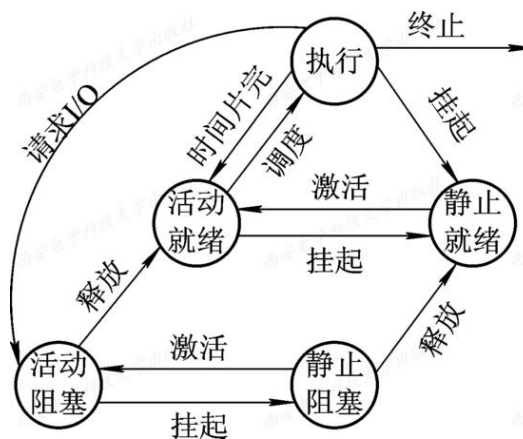
## 2.2 进程的描述

### 2.2.3 挂起操作和进程状态的转换

#### 1. 挂起操作的引入

引入挂起操作的原因，是基于系统和用户的如下需要：

- (1) 终端用户的需要。
- (2) 父进程请求。
- (3) 负荷调节的需要。
- (4) 操作系统的需要。



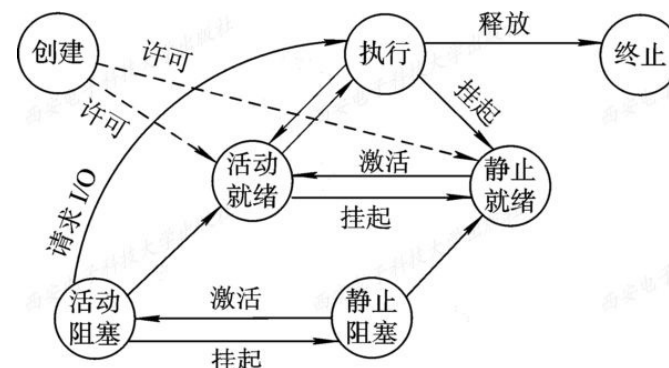
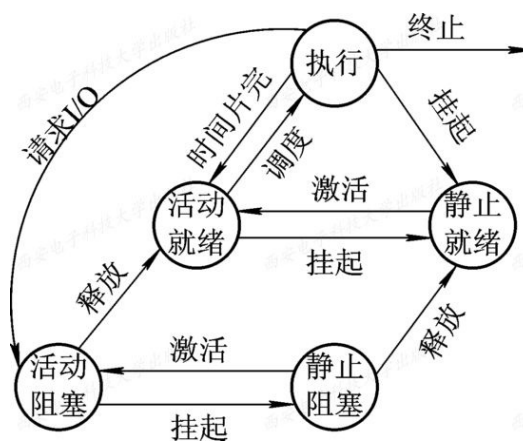
## 2.2 进程的描述

### 2.2.3 挂起操作和进程状态的转换

#### 2. 引入挂起原语操作后三个进程状态的转换

在引入挂起原语suspend和激活原语active后，在它们的作用下，进程将可能发生以下几种状态的转换：

- (1) 活动就绪→静止就绪。静止就绪不接受调度。
- (2) 活动阻塞→静止阻塞。
- (3) 静止就绪→活动就绪。
- (4) 静止阻塞→活动阻塞。



## 2.2 进程的描述

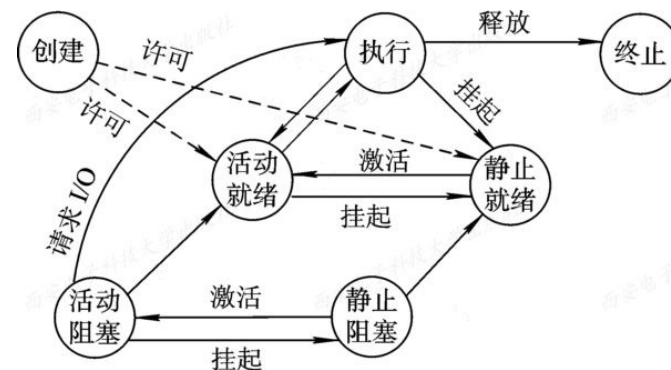
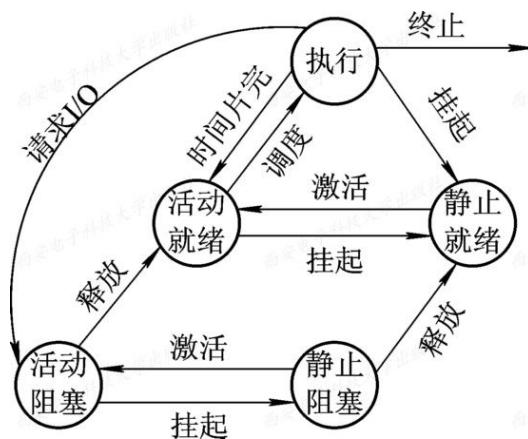
### 2.2.3 挂起操作和进程状态的转换

#### 3. 引入挂起操作后五个进程状态的转换

如图示出了增加了创建状态和终止状态后具有挂起状态的进程状态及转换图。

如图所示，引进创建和终止状态后，在进程状态转换时，与进程五状态转换相比较，要增加考虑下面的几种情况：

- (1) NULL→创建：
- (2) 创建→活动就绪：
- (3) 创建→静止就绪：
- (4) 执行→终止：



## 2.2 进程的描述

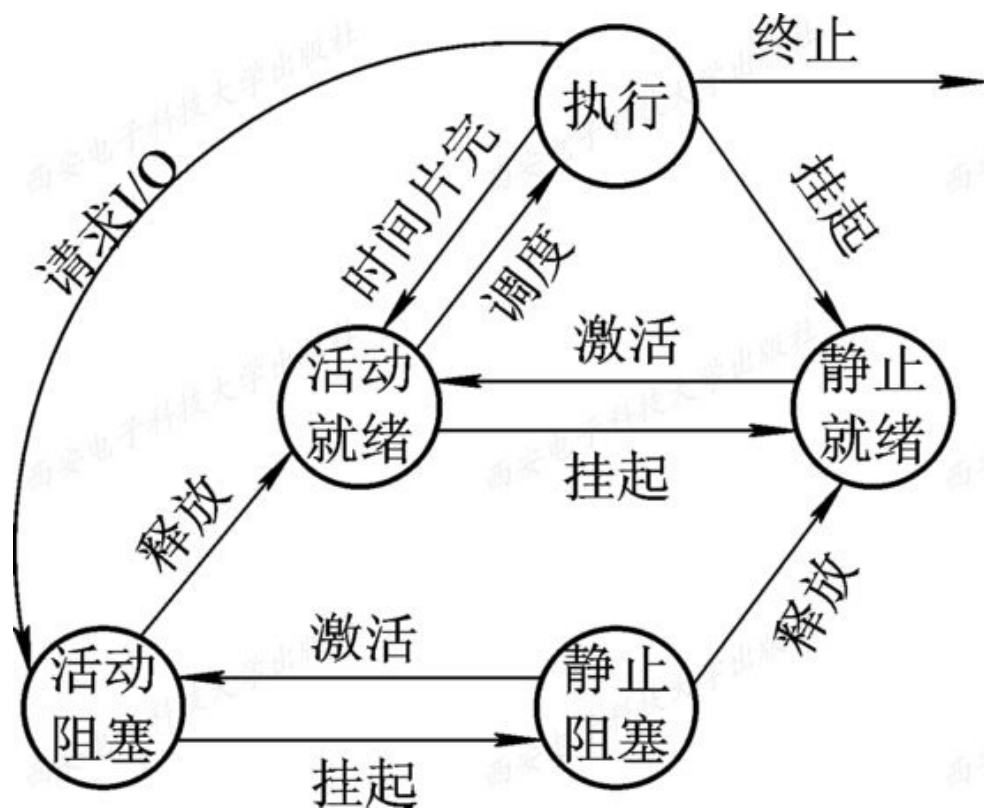


图2-7 具有挂起状态的进程状态图

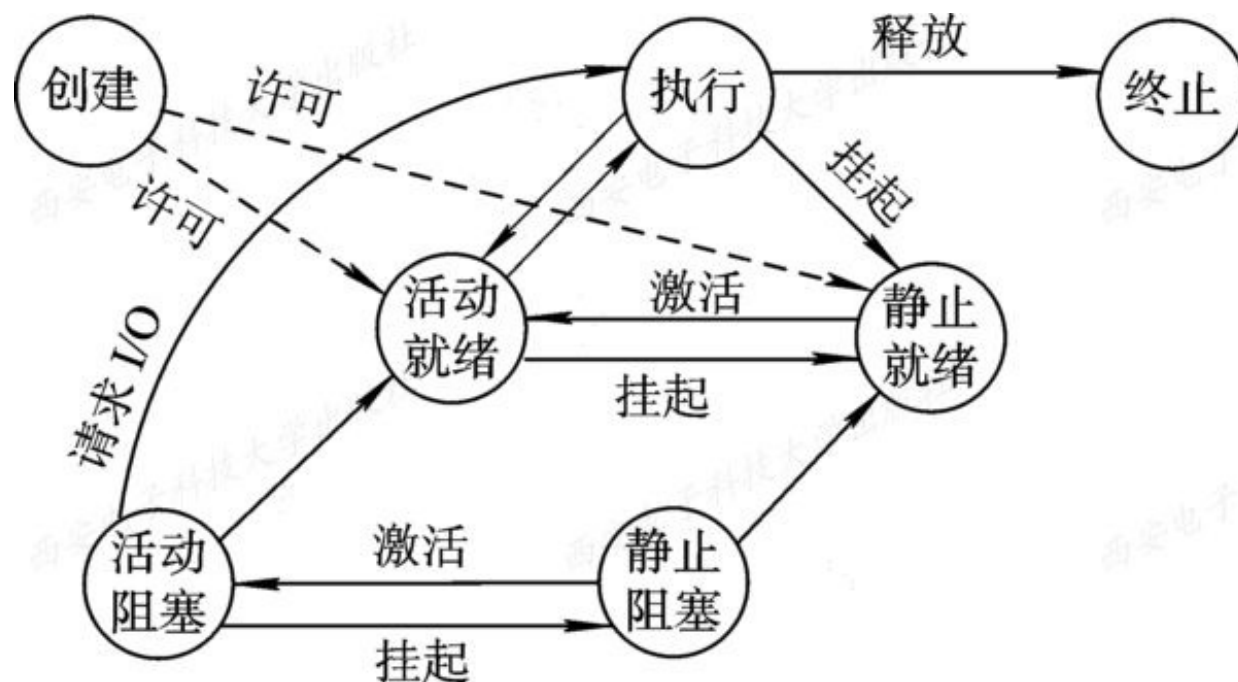


图2-8 具有创建、终止和挂起状态的进程状态图

## 2.2 进程的描述

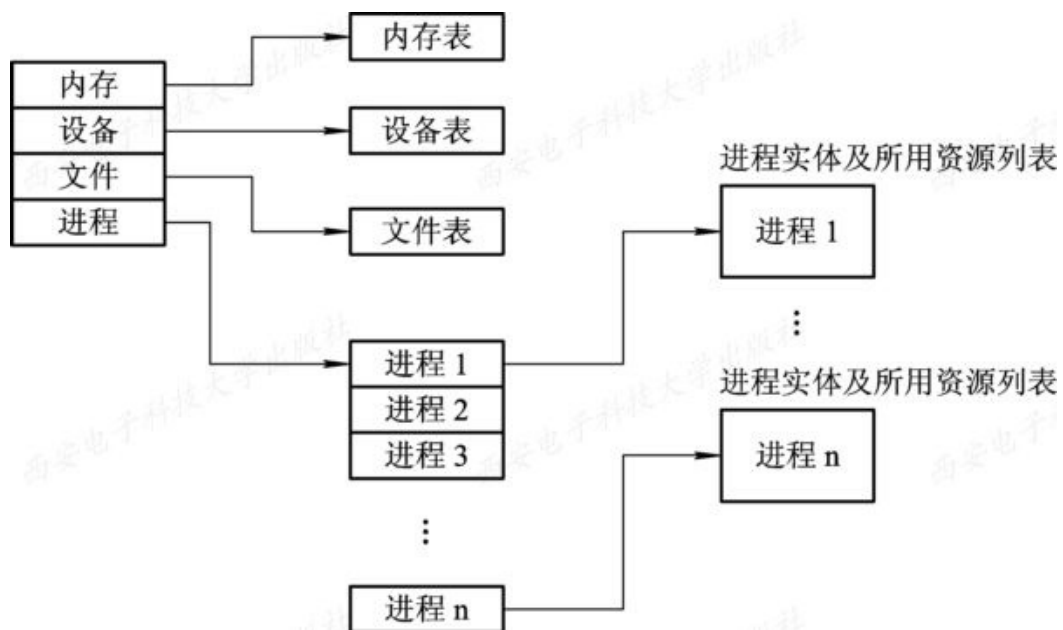
### 2.2.4 进程管理中的数据结构

#### 1. 操作系统中用于管理控制的数据结构

在计算机系统中，对于每个资源和每个进程都设置了一个数据结构，用于表征其实体，我们称之为**资源信息表或进程信息表**，其中包含了**资源或进程的标识、描述、状态等信息以及一批指针**。通过这些指针，可以将同类资源或进程的信息表，或者同一进程所占用的资源信息表分类链接成不同的队列，便于操作系统进行查找。

## 2.2 进程的描述

如图所示，OS管理的这些数据结构一般分为以下四类：内存表、设备表、文件表和用于进程管理的进程表，通常进程表又被称为进程控制块PCB。





## 2.2 进程的描述

### 2.2.4 进程管理中的数据结构

#### 2. 进程控制块PCB的作用

PCB作为进程实体的一部分，记录了操作系统所需的，用于描述进程的当前情况以及管

理进程运行的全部信息，是操作系统中**最**重要的记录型数据结构。

(1) 作为独立运行基本单位的标志。

创建进程时创建PCB，结束时回收PCB

(2) 能实现间断性运行方式。

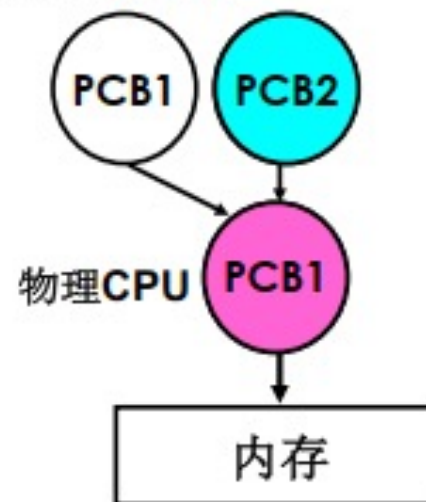
CPU现场保护

(3) 提供进程管理所需要的信息。

(4) 提供进程调度所需要的信息。

(5) 实现与其它进程的同步与通信。

一段数据(PCB)



## 2.2 进程的描述

### 2.2.4 进程管理中的数据结构

#### 2. 进程控制块PCB的作用

**PCB作为进程实体的一部分，记录了操作系统所需的，用于描述进程的当前情况以及管理进程运行的全部信息，是操作系统中最重要的记录型数据结构。**

- (1) 作为独立运行基本单位的标志。
- (2) 能实现间断性运行方式。
- (3) 提供进程管理所需要的信息。

调度时：PCB记录程序和数据在内存或者外存中的始指针

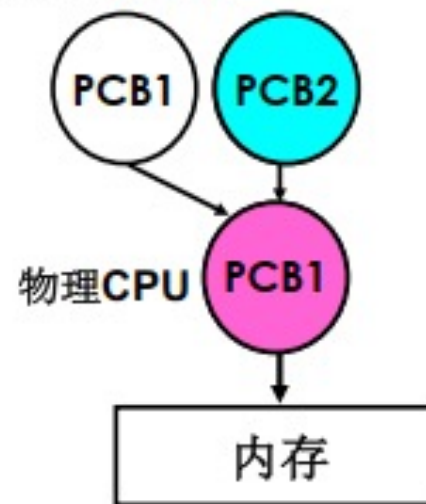
运行时的程序、数据、全部资源

- (4) 提供进程调度所需要的信息。

进程当前状态、优先级、等待时间、执行时间

- (5) 实现与其它进程的同步与通信。

一段数据(PCB)



## 2.2 进程的描述

### 2.2.4 进程管理中的数据结构

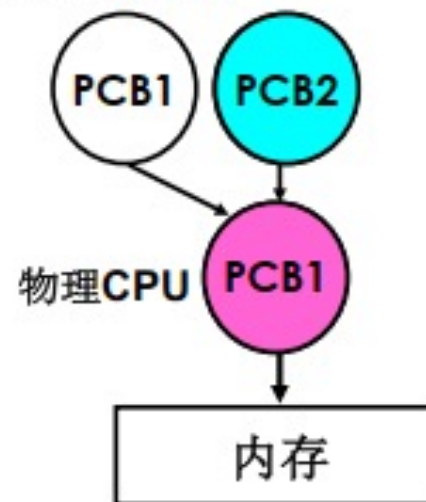
#### 2. 进程控制块PCB的作用

**PCB作为进程实体的一部分，记录了操作系统所需的，用于描述进程的当前情况以及管理进程运行的全部信息，是操作系统中最重要的记录型数据结构。**

- (1) 作为独立运行基本单位的标志。
- (2) 能实现间断性运行方式。
- (3) 提供进程管理所需要的信息。
- (4) 提供进程调度所需要的信息。
- (5) 实现与其它进程的同步与通信。

临界资源访问时信号量机制

一段数据(PCB)



## 2.2 进程的描述

### 2.2.4 进程管理中的数据结构

#### 3. 进程控制块中的信息

在进程控制块中，主要包括下述四个方面的信息。

##### 1) 进程标识符

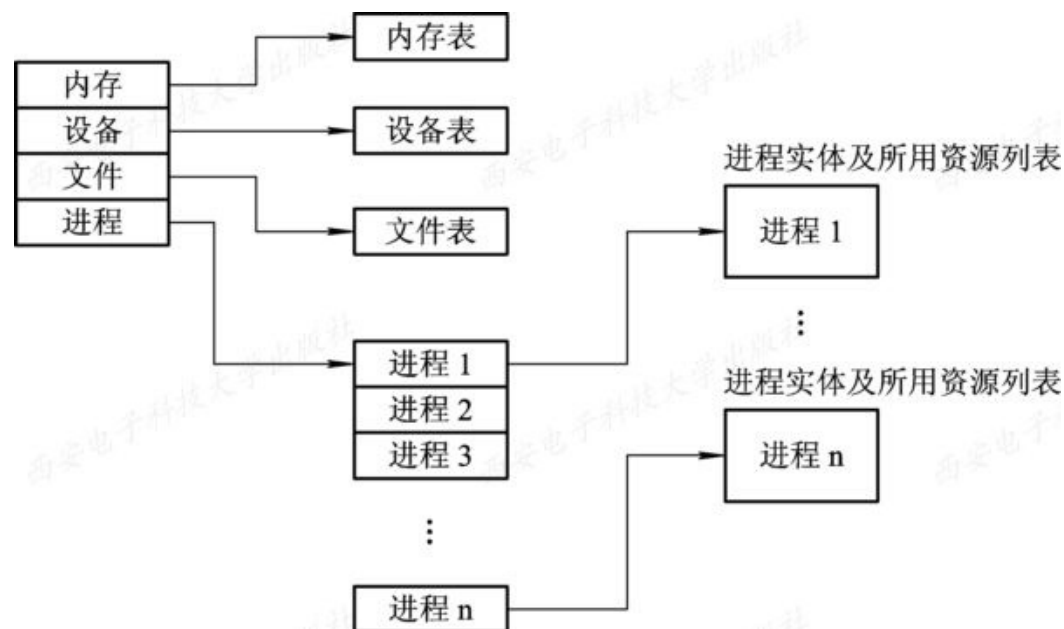
进程标识符用于唯一地标识一个进程。一个进程通常有两种标识符：

(1) 外部标识符。

通常由字母构成

(2) 内部标识符。

通常由数字构成



## 2.2 进程的描述

### 2.2.4 进程管理中的数据结构

#### 3. 进程控制块中的信息

在进程控制块中，主要包括下述四个方面的信息。

##### 2) 处理机状态

处理机状态信息也称为处理机的上下文，主要是由处理机的各种寄存器中的内容组成的。

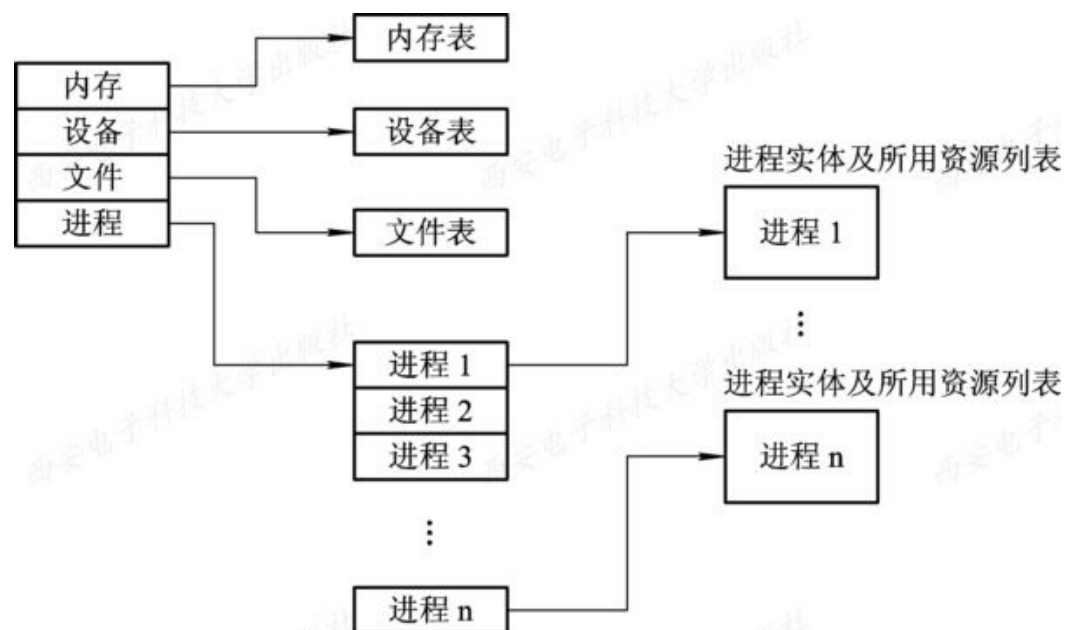
通用寄存器

指令计数器

程序状态字PSW

( 条件码、执行方式、中断屏蔽 )

用户栈指针



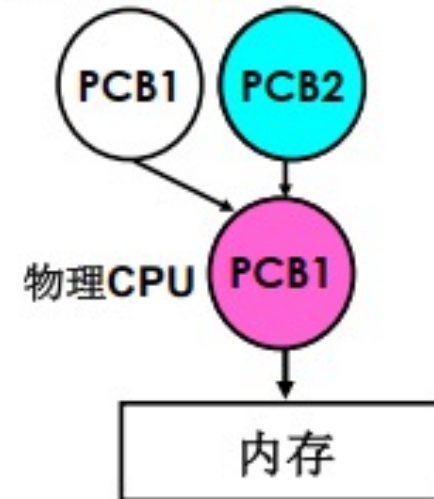
## 2.2 进程的描述

如图所示，OS管理的这些数据结构一般分为以下四类：内存表、设备表、文件表和用于进程管理的进程表，通常进程表又被称为进程控制块PCB。

```
switch_to(pCur,pNew) {  
    pCur.ax = CPU.ax;  
    pCur.bx = CPU.bx;  
    ...  
    pCur.cs = CPU.cs;  
    pCur.retpc = CPU.pc;  
  
    CPU.ax = pNew.ax;  
    CPU.bx = pNew.bx;  
    ...  
    CPU.cs = pNew.cs;  
    CPU.retpc = pNew.pc; }  

```

一段数据(PCB)



队列操作+调度+切换

## 2.2 进程的描述

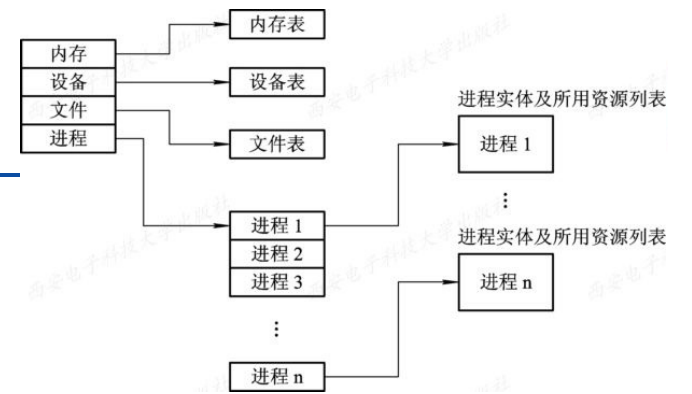
### 2.2.4 进程管理中的数据结构

#### 3. 进程控制块中的信息

##### 3) 进程调度信息

在OS进行调度时，必须了解进程的状态及有关进程调度的信息，这些信息包括：

- ① **进程状态**，指明进程的当前状态，它是作为进程调度和对换时的依据；
- ② **进程优先级**，是用于描述进程使用处理机的优先级别的一个整数，优先级高的进程应优先获得处理机；
- ③ **进程调度所需的其它信息**，它们与所采用的进程调度算法有关，比如，进程已等待CPU的时间总和、进程已执行的时间总和等；
- ④ **事件**，是指进程由执行状态转变为阻塞状态所等待发生的事件，即阻塞原因。



## 2.2 进程的描述

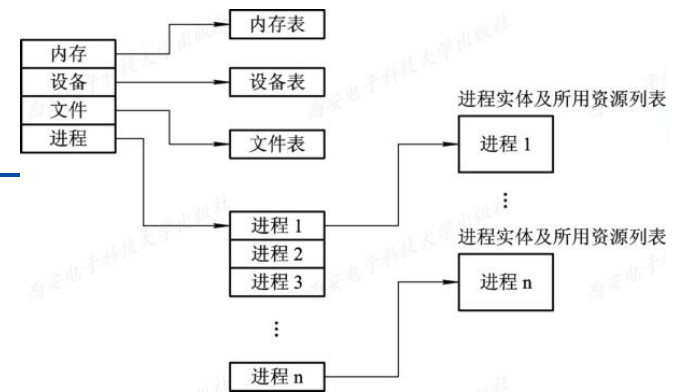
### 2.2.4 进程管理中的数据结构

#### 3. 进程控制块中的信息

#### 4) 进程控制信息

是指用于进程控制所必须的信息，它包括：

- ① **程序和数据地址**，进程实体中的程序和数据内存或外存地址(首址)，以便再调度到该进程执行时，能从PCB中找到其程序和数据；
- ② **进程同步和通信机制**，这是实现进程同步和进程通信时必需的机制，如消息队列指针、信号量等，它们可能全部或部分地放在PCB中；
- ③ **资源清单**，在该清单中列出了进程在运行期间所需的全部资源(除CPU以外)，另外还有一张已分配到该进程的资源的清单；
- ④ **链接指针**，它给出了本进程(PCB)所在队列中的下一个进程的PCB的首地址。





## 2.2 进程的描述

### 2.2.4 进程管理中的数据结构

#### 4. 进程控制块的组织方式

在一个系统中，通常可拥有数十个、数百个乃至数千个PCB。为了能对它们加以有效的管理，应该用适当的方式将这些PCB组织起来。目前常用的组织方式有以下三种。

**(1) 线性方式**，即将系统中所有的PCB都组织在一张线性表中，**将该表的首址存放在内存的一个专用区域中**。该方式实现简单、开销小，但每次查找时都需要扫描整张表，因此适合进程数目不多的系统。图2-10示出了线性表的PCB组织方式。

PCB1
PCB2
PCB3
⋮
PCBn

图2-10 PCB线性表示意图

## 2.2 进程的描述

**(2) 链接方式**，即把具有相同状态进程的PCB分别通过PCB中的链接字链接成一个队列。这样，可以形成就绪队列、若干个阻塞队列和空白队列等。对就绪队列而言，往往按进程的优先级将PCB从高到低进行排列，将优先级高的进程PCB排在队列的前面。同样，也可把处于阻塞状态进程的PCB根据其阻塞原因的不同，排成多个阻塞队列，如等待I/O操作完成的队列和等待分配内存的队列等。图2-11示出了一种链接队列的组织方式。

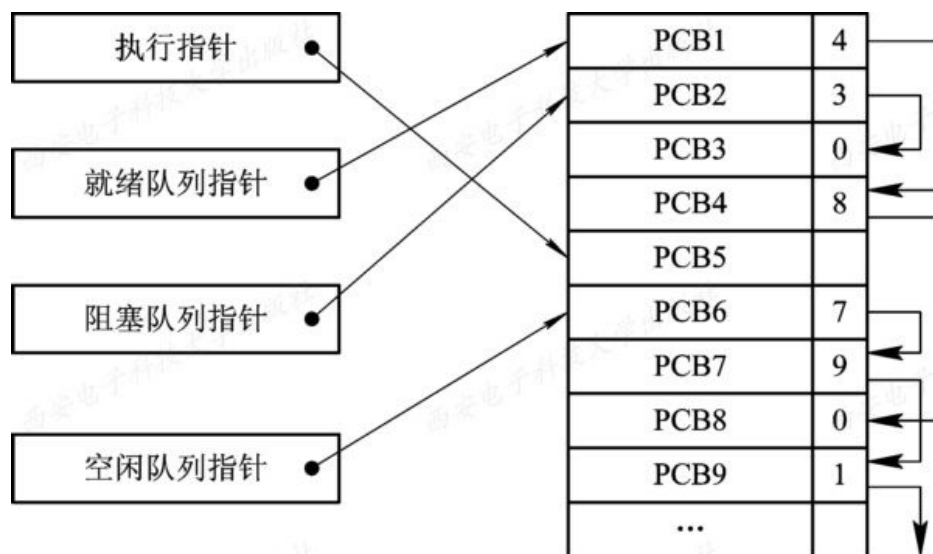


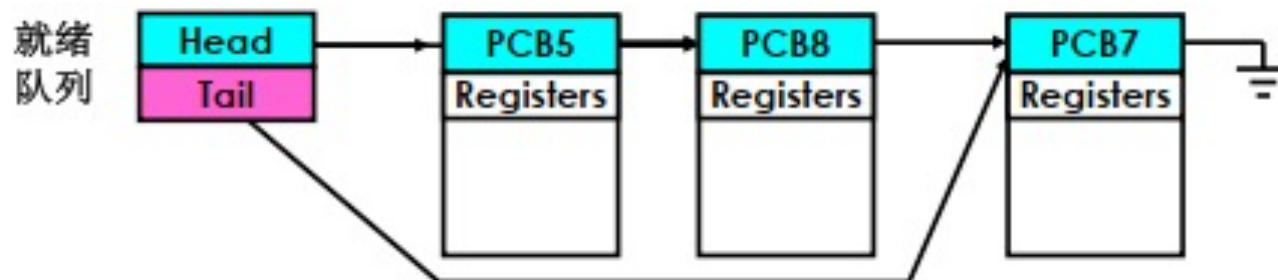
图2-11 PCB链接队列示意图

## 2.2 进程的描述

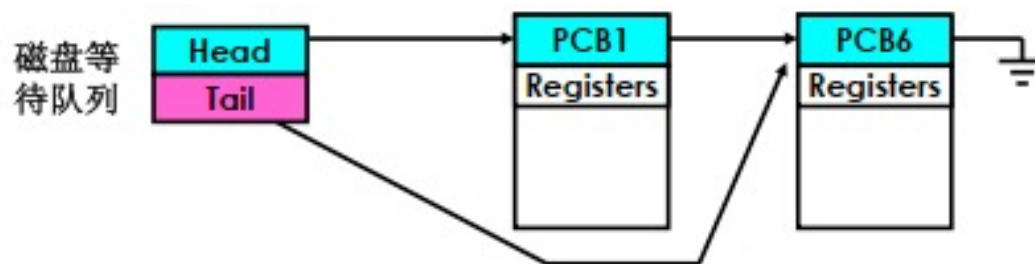
■ 有一个进程正在执行



■ 有一些进程等待执行



■ 有一个进程在等待某事件（如等待数据）



## 2.2 进程的描述

(3) **索引方式**，即系统根据所有进程状态的不同，建立几张索引表，例如，就绪索引表、阻塞索引表等，并把各索引表在**内存的首地址记录在内存的一些专用单元中**。在每个索引表的表目中，记录具有相应状态的某个PCB在PCB表中的地址。图2-12示出了索引方式的PCB组织。

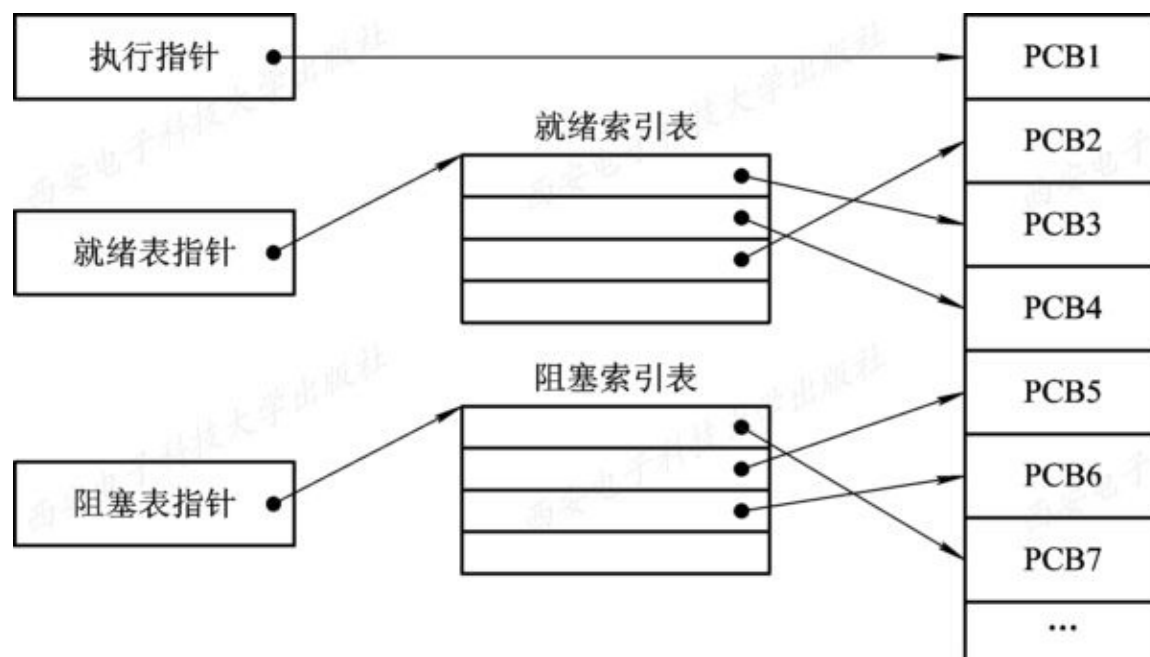


图2-12 按索引方式组织PCB



# 第二章 进程的描述与控制



2.1 前趋图和程序执行

2.2 进程的描述

2.3 进程控制

2.4 进程同步

2.5 经典进程的同步问题

2.6 进程通信

2.7 线程(Threads)的基本概念

2.8 线程的实现

习题



## 2.3 进程控制



进程控制是进程管理中**最**基本的功能，主要包括创建新进程、终止已完成的进程、将因发生异常情况而无法继续运行的进程置于阻塞状态、负责进程运行中的状态转换等功能。

如当一个正在执行的进程因等待某事件而暂时不能继续执行时，将其转变为阻塞状态，而在该进程所期待的事件出现后，又将该进程转换为就绪状态等。

**进程控制**一般是由OS的内核中的**原语**来实现的。

## 2.3 进程控制

### 2.3.1 操作系统内核

**OS内核：常驻内存的、紧靠硬件的软件层次**

- 中断处理程序
- 驱动程序
- 时钟管理模块

如微内核

- 进程调度模块 ...

如大内核

**处理机的执行状态**

- 系统态：管态
- 用户态：目态

**目的**

- 软件保护
- 提高运行效率

## 2.3 进程控制

### 2.3.1 操作系统内核

#### 1. 支撑功能

- (1) 中断处理。
- (2) 时钟管理。时间片
- (3) 原语操作。不可中断，常驻内存

#### 2. 资源管理功能

- (1) 进程管理。
- (2) 存储器管理。
- (3) 设备管理。



## 2.3 进程控制

### 2.3.2 进程的创建

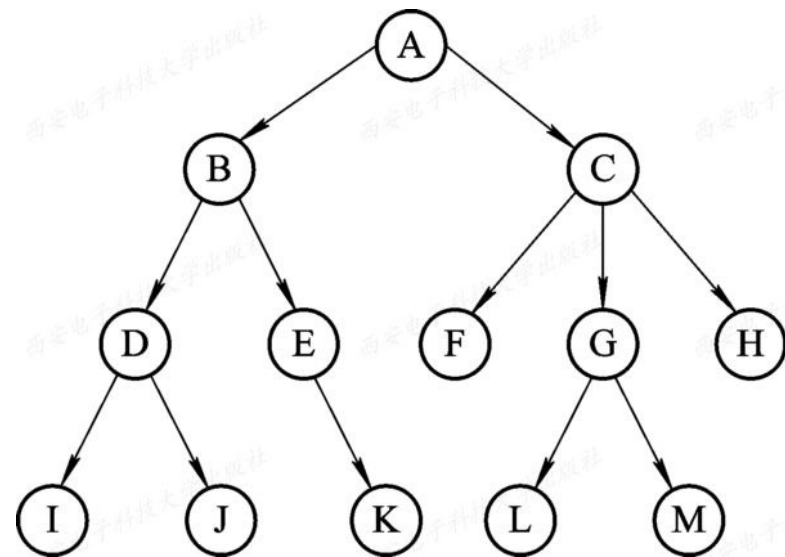
#### 1. 进程的层次结构

在OS中，允许一个进程创建另一个进程，通常把创建进程的进程称为**父进程**，而把被创建的进程称为**子进程**。子进程可继续创建更多的孙进程，由此便形成了一个进程的层次结构。如在UNIX中，进程与其子孙进程共同组成一个进程家族(组)。

## 2.3 进程控制

### 2. 进程图

为了形象地描述一个进程的家族关系而引入了进程图 (Process Graph)。所谓进程图就是用于描述进程间关系的一棵有向树，如图所示。



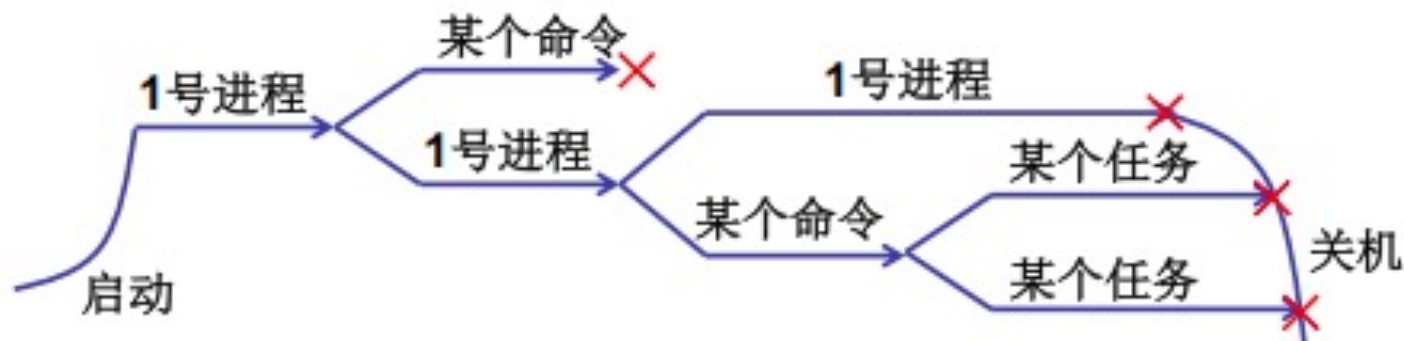
## 2.3 进程控制

### ■ 进程的管理

```
login as: tnp
tnp@192.168.1.1's password:
Last login: Mon Nov 12 23:47:50 2006 from 192.168.1.2
[tnp@tevmull ~]$ ls
work.txt
[tnp@tevmull ~]$
```

Shell

### ■ Linux init 创建 shell , shell创建其它进程



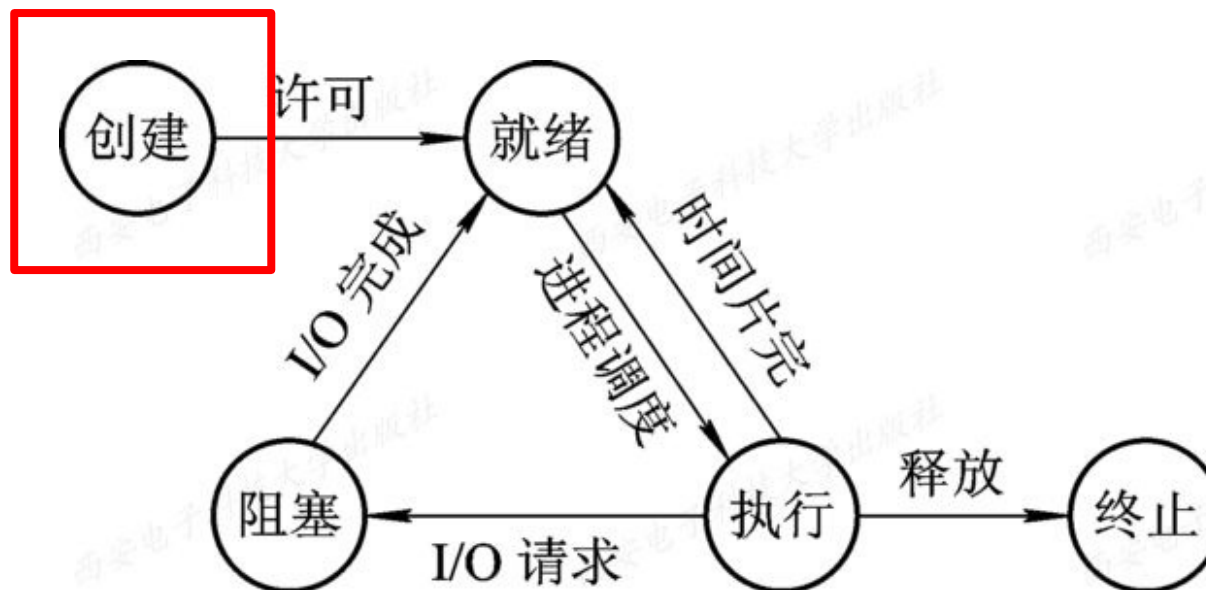
## 2.3 进程控制

### 2.3.2 进程的创建

#### 3. 引起创建进程的事件

为使程序之间能并发运行，应先为它们分别创建进程。导致一个进程去创建另一个进程的典型事件有四类：

- (1) 用户登录。
- (2) 作业调度。
- (3) 提供服务。
- (4) 应用请求。



## 2.3 进程控制

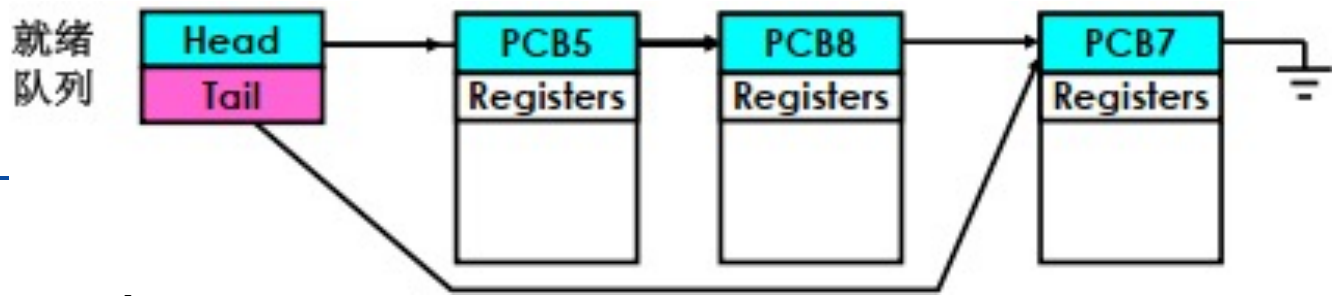
### 2.3.2 进程的创建

#### 4. 进程的创建(Creation of Process)

在系统中每当出现了创建新进程的请求后，OS便调用进程创建原语creat按下述步骤创建一个新进程：

**(1) 申请空白PCB**，为新进程申请获得唯一的数字标识符，并从PCB集合中索取一个空白PCB。

**(2) 为新进程分配其运行所需的资源**，包括各种物理和逻辑资源，如内存、文件、I/O设备和CPU时间等。



## 2.3 进程控制

### 2.3.2 进程的创建

#### 4. 进程的创建(Creation of Process)

在系统中每当出现了创建新进程的请求后，OS便调用进程创建原语Creat按下述步骤创建一个新进程：

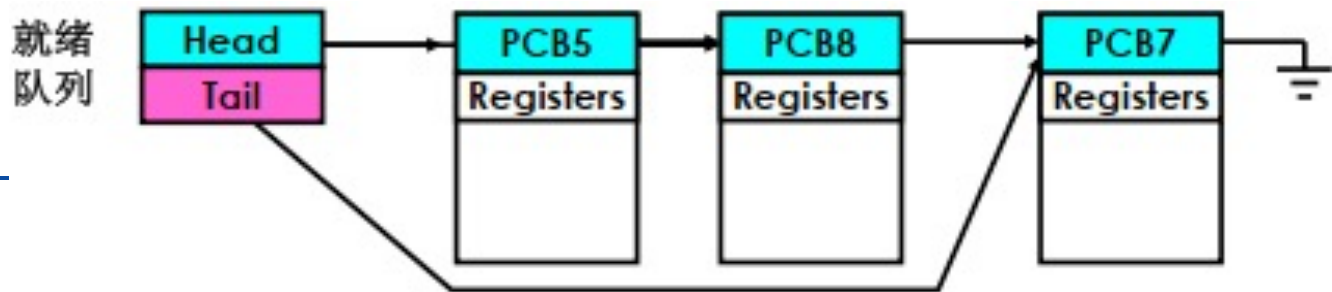
#### (3) 初始化进程控制块(PCB)。

初始化标识信息；

初始化处理机状态信息；

初始化处理机控制信息

(4) 如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。



## 2.3 进程控制

### 2.3.3 进程的终止

#### 1. 引起进程终止(Termination of Process)的事件

(1) 正常结束

(2) 异常结束

越界错误、保护错

非法指令、特权指令错

运行超时、等待超时

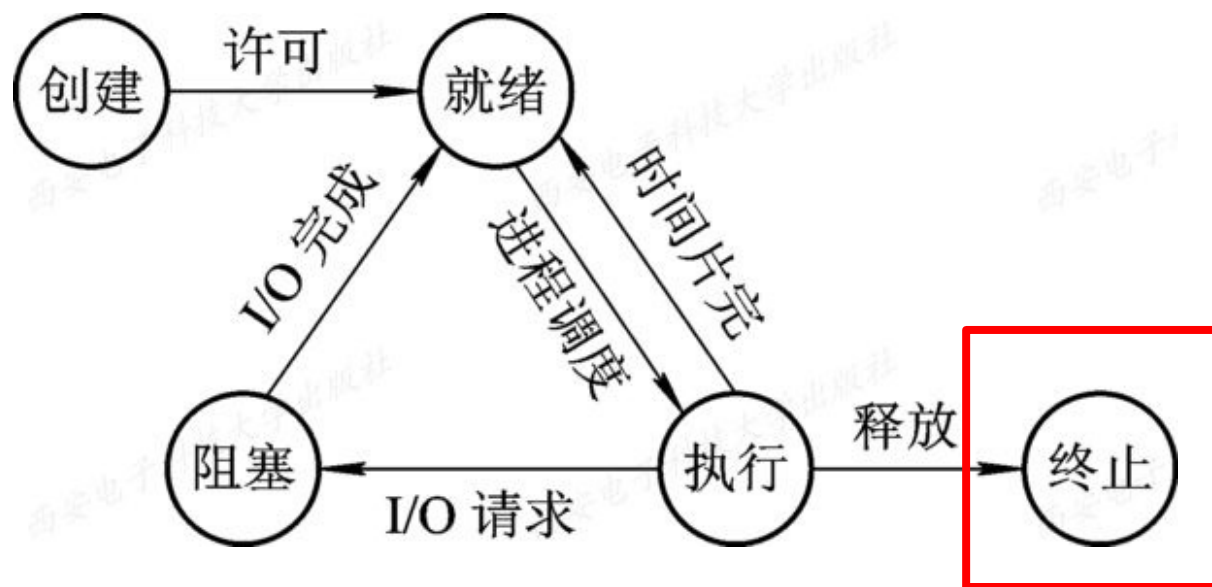
算数运行错、I/O 故障

(3) 外界干预

操作员或者操作系统干预

父进程请求

因父进程终止



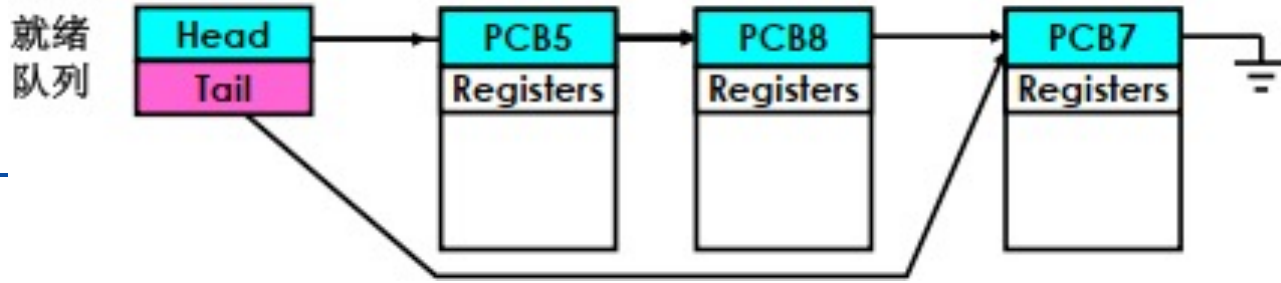
## 2.3 进程控制

### 2.3.3 进程的终止

#### 2. 进程的终止过程

如果系统中发生了要求终止进程的某事件，OS便调用进程终止原语，按下述过程去终止指定的进程：

- (1) **根据被终止进程的标识符**，从PCB集合中检索出该进程的PCB，从中读出该进程的状态；
- (2) 若被终止进程正处于执行状态，应立即终止该进程的执行，**并置调度标志为真**，用于指示该进程被终止后应重新进行调度；
- (3) **若该进程还有子孙进程**，还应将其所有子孙进程也都予以终止，以防它们成为不可控的进程；
- (4) 将被终止进程所拥有的全部资源或者归还给**其父进程**，**或者归还给系统**；
- (5) **将被终止进程(PCB)从所在队列(或链表)中移出**，等待其它程序来搜集信息。





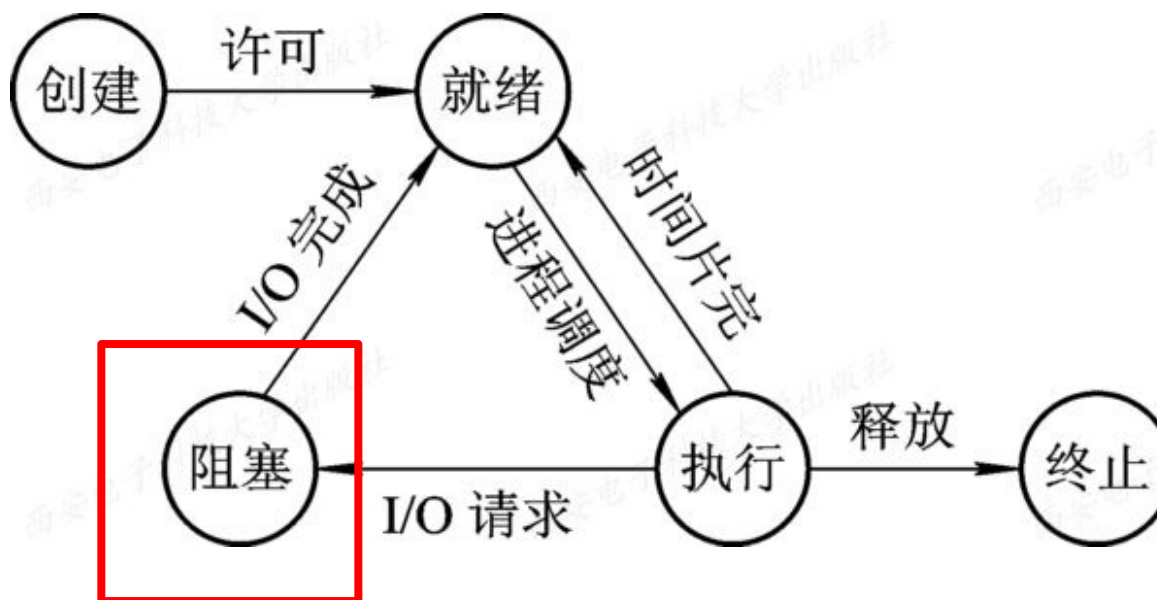
## 2.3 进程控制

### 2.3.4 进程的阻塞与唤醒

#### 1. 引起进程阻塞和唤醒的事件

有下述几类事件会引起进程阻塞或被唤醒：

- (1) 向系统请求共享资源失败。
- (2) 等待某种操作的完成。
- (3) 新数据尚未到达。
- (4) 等待新任务的到达。



## 2.3 进程控制

### 2.3.4 进程的阻塞与唤醒

#### 2. 进程阻塞过程

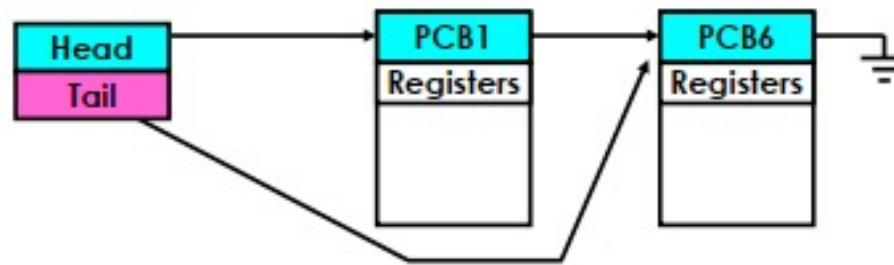
正在执行的进程，如果发生了上述某事件，进程便通过调用阻塞原语**block**将自己阻塞。

可见，**阻塞是进程自身的一种主动行为**。

进入block过程后，由于该进程还处于执行状态，所以应先立即停止执行，把进程控制块中的现行状态由“执行”改为阻塞，并将PCB插入阻塞队列。

如果**系统中设置了因不同事件而阻塞的多个阻塞队列**，则应将本进程插入到具有相同事件的阻塞队列。

最后，转调度程序进行重新调度，将处理机分配给另一就绪进程，并进行切换，亦即，保留被阻塞进程的处理机状态，按新进程的PCB中的处理机状态设置CPU的环境。



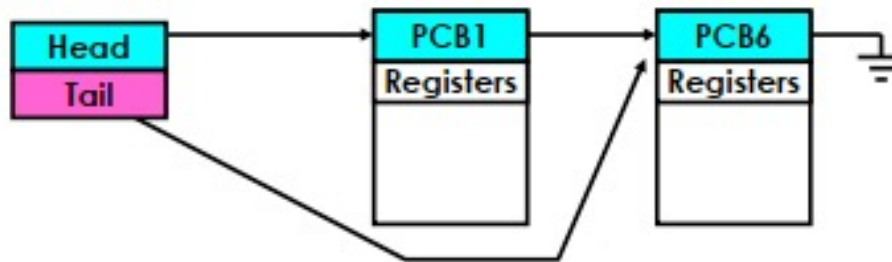
## 2.3 进程控制

### 2.3.4 进程的阻塞与唤醒

#### 3. 进程唤醒过程

当被阻塞进程所期待的事件发生时，比如它所启动的I/O操作已完成，或其所期待的数据已经到达，则由有关进程(比如提供数据的进程)调用唤醒原语**wakeup**，将等待该事件的进程唤醒。

wakeup执行的过程是：首先把被阻塞的进程从等待该事件的阻塞队列中移出，将其PCB中的现行状态由阻塞改为就绪，然后再将该PCB插入到就绪队列中。



## 2.3 进程控制

### 2.3.5 进程的挂起与激活

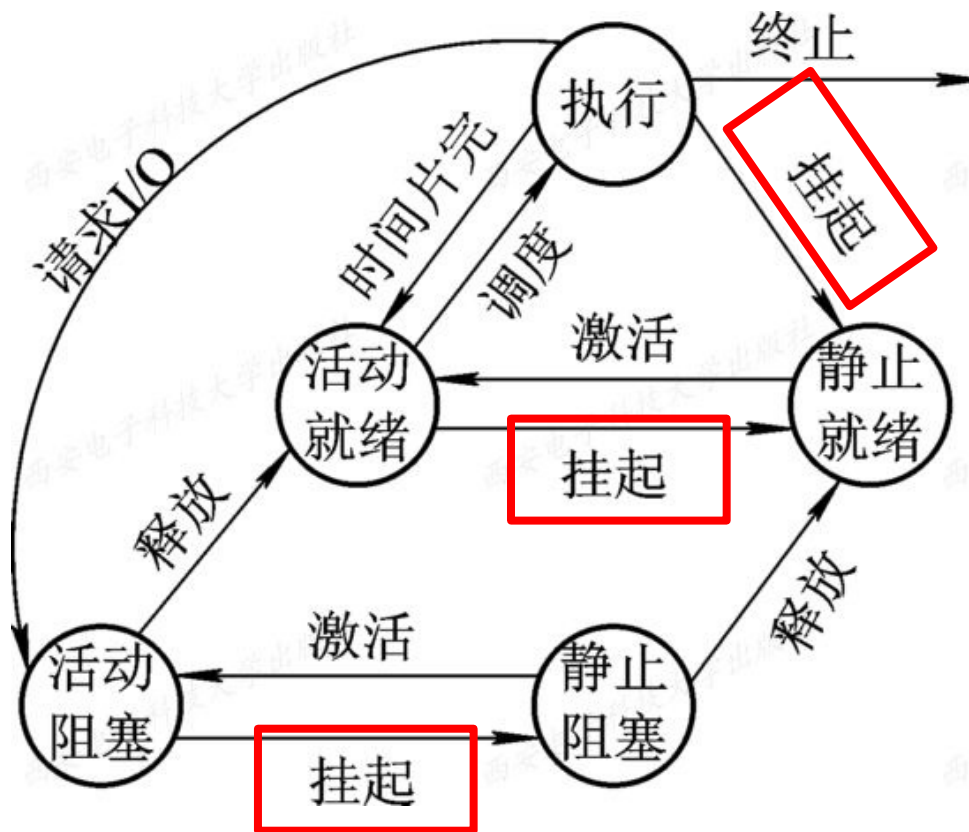
#### 1. 进程的挂起

挂起原语，suspend

首先，检查被挂起进程的状态，若处于活动就绪，便将其改为静止就绪；若为活动堵塞，则改为静止堵塞。

把PCB复制到指定的内存区域。

若被挂起的进程正在执行，那么重新调度新的进程。



## 2.3 进程控制

### 2.3.5 进程的挂起与激活

#### 1. 进程的挂起

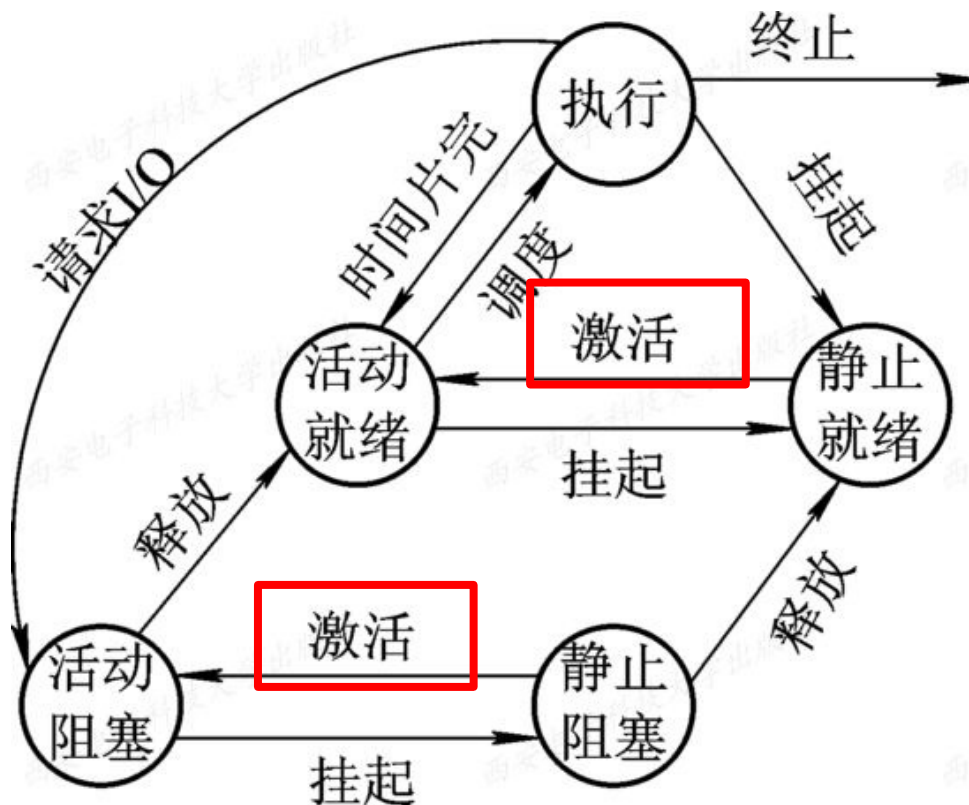
挂起原语，suspend

#### 2. 进程的激活过程

激活原语，active

从外存调入内存

变为活动就绪时，确定优先级，查看是否被重新调度。（抢占则重新调度，不抢占则插入队列）





# 第二章 进程的描述与控制



2.1 前趋图和程序执行

2.2 进程的描述

2.3 进程控制

2.4 进程同步

2.5 经典进程的同步问题

2.6 进程通信

2.7 线程(Threads)的基本概念

2.8 线程的实现

习题

## 2.4 进程同步



在OS中引入进程后，一方面可以使系统中的**多道程序并发执行**，这不仅能有效地改善资源利用率，还可显著地提高系统的吞吐量，但另一方面却使系统变得更加复杂。如果不能采取有效的措施，对多个进程的运行进行妥善的管理，必然会因为这些进程对系统资源的无序争夺给系统造成混乱。致使每次处理的结果存在着不确定性，即显现出其不可再现性。

**信号量机制**

**硬件同步机制**

**管程机制**

## 2.4 进程同步

### 2.4.1 进程同步的基本概念

#### 同步机制应遵循的规则

为实现进程互斥地进入自己的临界区，可用软件方法，更多的是在系统中设置专门的同步机构来协调各进程间的运行。所有同步机制都应遵循下述四条准则：

- (1) 空闲让进。若干进程要求进入空闲临界区时，应尽快使一进程进入临界区。
- (2) 忙则等待。临界区正在访问，则等待。
- (3) 有限等待。从进程发出进入请求到允许进入，不能无限等待。
- (4) 让权等待。当进程不能进入自己的临界区时，应释放处理机，以免陷入忙等。



## 2.4 进程同步

### 2.4.1 进程同步的基本概念

#### 1. 两种形式的制约关系

##### 1) 间接相互制约关系

共享资源，如I/O、CPU等

##### 2) 直接相互制约关系

进程的数据

#### 进程1代码

```
mov ax, 10100b
```

```
mov [100], ax
```

.....

#### 进程2代码

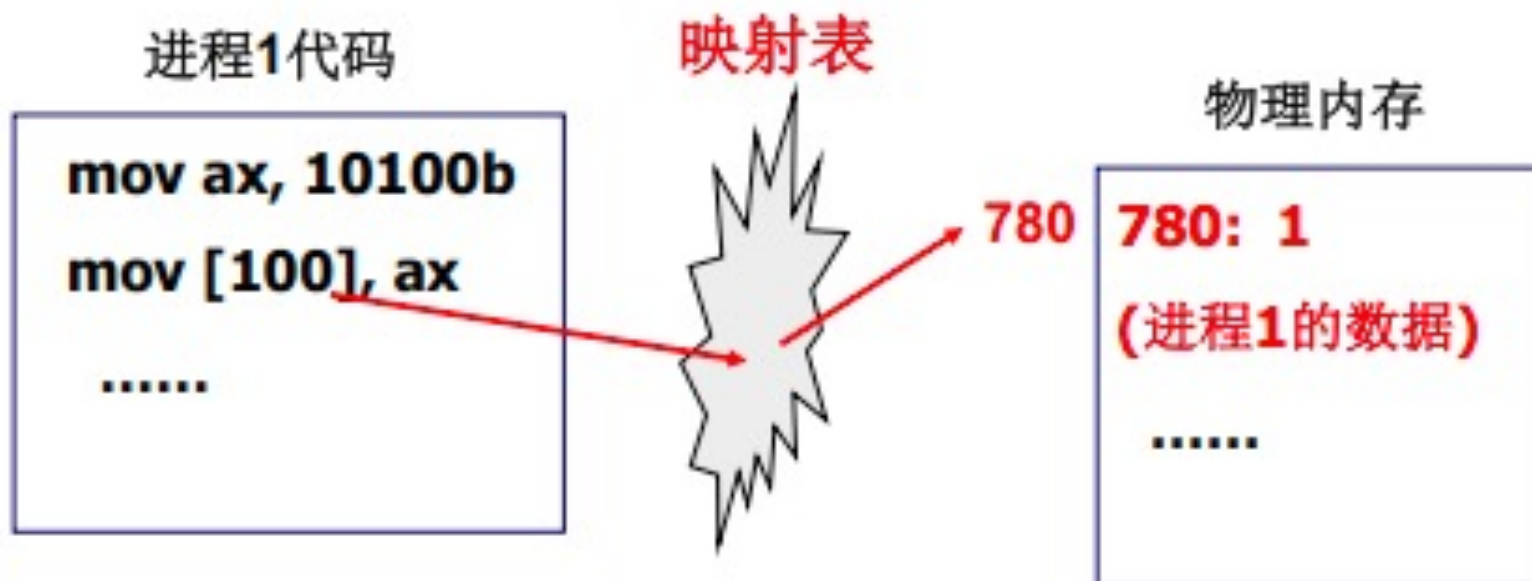
```
100: 00101
```

.....

#### 2. 临界资源(Critical Resouce)

在第一章中我们曾经介绍过，许多硬件资源如打印机、磁带机等，都属于临界资源，诸进程间应采取互斥方式，实现对这种资源的共享。

## 2.4 进程同步



- 进程1的映射表将访问限制在进程1的范围内
- 进程1根本访问不到其他进程的内容
- 内存管理 ... (其他设备的管理服务于CPU管理, 即服务于进程管理)

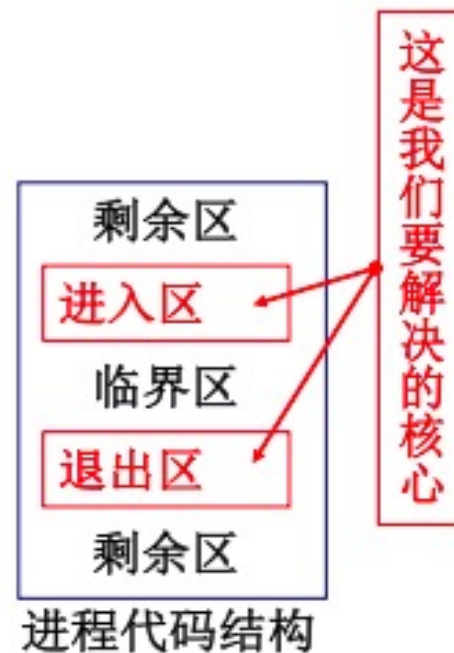
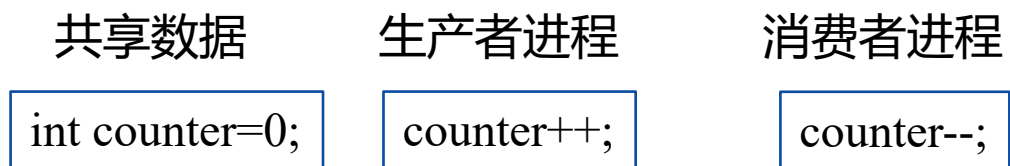
## 2.4 进程同步

### 2.4.1 进程同步的基本概念

#### 3. 临界区(critical section)

由前所述可知，不论是硬件临界资源还是软件临界资源，多个进程必须**互斥地对它进行访问**。人们把在每个进程中**访问临界资源的那段代码称为临界区**(critical section)。

**一次只允许一个进程进入该进程的临界区代码**



## 2.4 进程同步

### 生产者—消费者 问题

共享数据

```
int counter=0;
```

生产者进程

```
counter++;
```

消费者进程

```
counter--;
```

初始情况

```
counter=5;
```

生产者P

```
P.register = counter;  
P.register = register+1;  
counter = P.register;
```

消费者C

```
C.register = counter;  
C.register = register-1;  
counter = C.register;
```

时间片1

```
P.register = counter;  
P.register = register+1;
```

时间片2

```
C.register = counter;  
C.register = register-1;
```

时间片3

```
counter = P.register;
```

时间片4

```
counter = C.register;
```

**P进程与C进程并发执行，**

**执行的顺序不同，**

**counter 最终等于 6 ? 4 ? 5 ?**

**不满足异步性**

## 2.4 进程同步

### 生产者—消费者 问题

共享数据

```
int counter=0;
```

生产者进程

```
counter++;
```

消费者进程

```
counter--;
```

初始情况

```
counter=5;
```

生产者P

```
P.register = counter;  
P.register = register+1;  
counter = P.register;
```

消费者C

```
C.register = counter;  
C.register = register-1;  
counter = C.register;
```

时间片1

```
P1.register = counter;  
P1.register = register+1;
```

时间片2

```
P2.register = counter;  
P2.register = register+1;
```

时间片3

```
counter = P.register;
```

时间片4

```
counter = P2.register;
```

**P1进程与P2进程并发执行，  
执行的顺序不同，  
counter 最终等于 ???  
不满足异步性**

## 2.4 进程同步

生产者—消费者 问题

共享数据

```
int counter=0
```

初始情况

```
counter=5;
```

生产者P

消费者C

时间片1

```
P.register = counter;  
P.register = register+1;
```

**不允许counter被多个进程同时修改**

**为加一把锁（互斥信号），不让切换！！！！**

**合理推进进程**

消费者C

```
C.register = counter;  
C.register = register-1;  
counter = C.register;
```

执行的顺序不同，

counter 最终等于 6 ? 4 ? 5 ?

不满足异步性

## 2.4 进程同步

生产者和消费者

设置互斥信号量，mutex 初始值为1  
wait, signal成对出现；counter值确定

counter 作为临界区资源



### 2.4.3 信号量机制

#### 1. 整型信号量

最初由Dijkstra把整型信号量定义为一个用于表示资源数目的整型量S，它与一般整型量不同，除初始化外，仅能通过两个标准的**原子操作**(Atomic Operation) **wait(S)**和**signal(S)**来访问。很长时间以来，这两个操作一直被分别称为P、V操作。

```
wait(S){  
    while(S <= 0) ;  
    S--;  
}
```

```
signal(S){  
    S++;  
}
```

## 2.4 进程同步

### 生产者和消费者

#### counter 作为临界区资源



生产者P

wait(mutex) #检查并给counter加锁

P.register = counter;

P.register = register+1;

消费者C

wait(mutex) #检查counter的锁

生产者P

counter = P.register;

signal(mutex) #给counter开锁

消费者C

检查并给counter加锁

C.register = counter;

C.register = register-1;

counter = C.register;

signal(mutex) #给counter开锁

设置互斥信号量  
mutex初始值为1  
wait, signal成对出现  
counter值确定



## 2.4 进程同步

### 生产者和生产者

#### counter 作为临界区资源



生产者P1

wait(mutex) #检查并给counter加锁

P.register = counter;

P.register = register+1;

生产者P2

wait(mutex) #检查counter的锁

生产者P1

counter = P.register;

signal(mutex) #给counter开锁

生产者P2

检查并给counter加锁

P2.register = counter;

P2.register = register+1;

counter = P2.register;

signal(mutex) #给counter开锁

设置互斥信号量

mutex 初始值为1

wait, signal成对出现

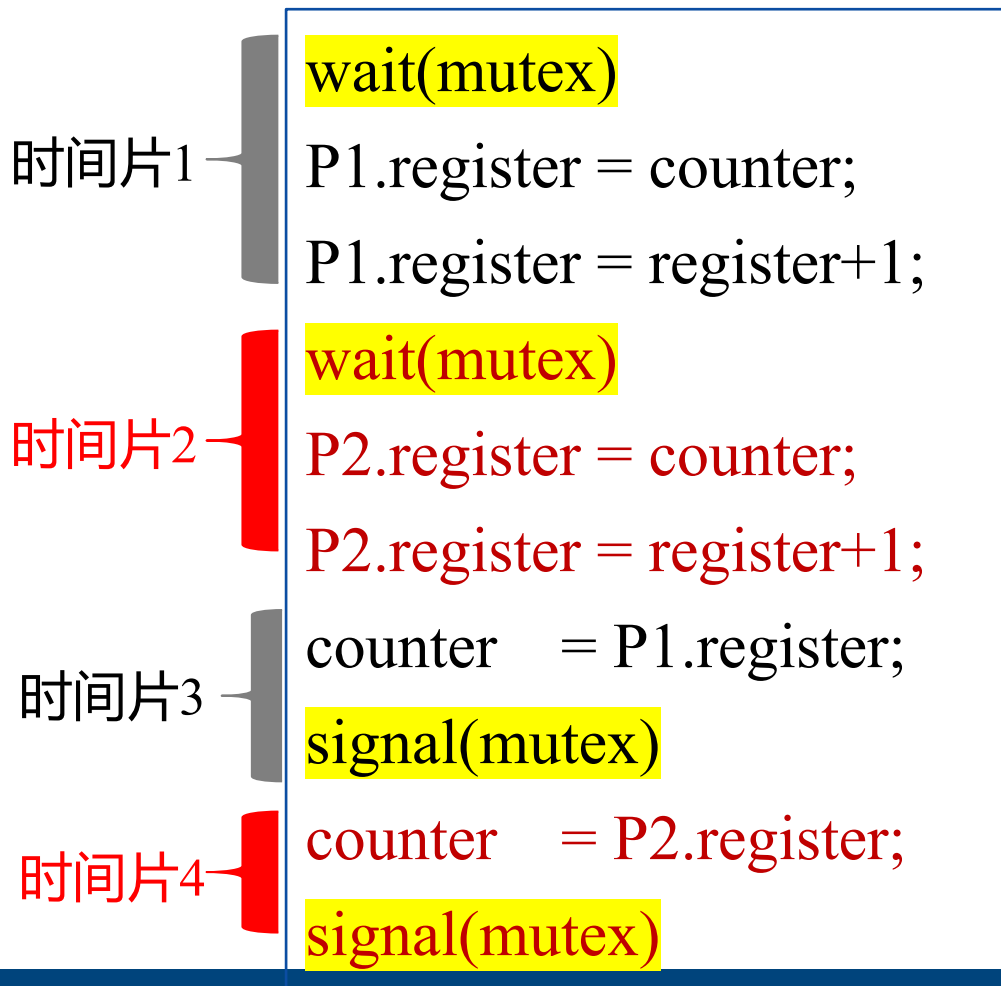
counter值确定

保证counter被加了两次

## 2.4 进程同步

### 生产者和生产者

#### counter 作为临界区资源



生产者P1

wait(mutex) #检查并给counter加锁

P.register = counter;

P.register = register+1;

生产者P2

wait(mutex) #检查counter的锁

生产者P1

counter = P.register;

signal(mutex) #给counter开锁

生产者P2

检查并给counter加锁

P2.register = counter;

P2.register = register+1;

counter = P2.register;

signal(mutex) #给counter开锁

忙等!!!

## 2.4 进程同步

生产者—消费者 问题

共享数据

counter;

同步机制应遵循的规则：

让权等待：

当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入“忙等”状态。

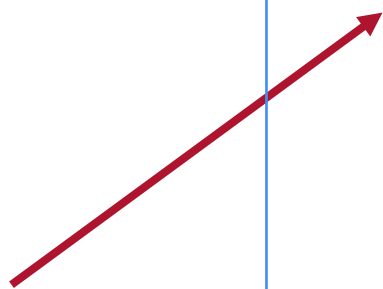
本例不满足**让权等待**！！！！

生产者进程 P1

```
wait(mutex)  ## 空闲让进
P1.register = counter;
P1.register = register+1;
counter  = P1.register;
signal(mutex)
```

生产者进程 P2

```
wait(mutex)  #忙则等待
P2.register = counter;
P2.register = register+1;
counter  = P2.register;
signal(mutex)
```



## 2.4 进程同步

生产者—消费者 问题

共享数据

counter;

**忙等！！！资源浪费**

**没有资源时，把自己block()**

**什么时候执行唤醒操作？**

生产者进程 P1

```
wait(mutex)
P1.register = counter;
P1.register = register+1;
counter = P1.register;
signal(mutex) #####wakeup()??????
```

生产者进程 P2

```
wait(mutex) ##### block()??????
P2.register = counter;
P2.register = register+1;
counter = P2.register;
signal(mutex)
```

## 2.4 进程同步

### 2.4.3 信号量机制

#### 2. 记录型信号量

在整型信号量机制中的wait操作，只要是信号量 $S \leq 0$ ，就会不断地测试。因此，**该机制并未遵循“让权等待”的准则，而是使进程处于“忙等”的状态**。记录型信号量机制则是一种不存在“忙等”现象的进程同步机制。但在采取了“让权等待”的策略后，又会出现多个进程等待访问同一临界资源的情况。为此，在信号量机制中，除了需要一个用于代表资源数目的整型变量value外，还应增加一个**进程链表指针list，用于链接上述的所有等待进程。**

```
typedef struct{
    int value;
    struct process_control_block* list;
}semaphore;
```

## 2.4 进程同步

### 2.4.3 信号量机制

#### 2. 记录型信号量

相应地，wait(S)和signal(S)操作可描述如下：

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) block(S->list);  
    #value==0时，减一等于负一，堵塞  
}
```

value ≥ 0 : 现有的资源量

value < 0 : 绝对值表示被堵塞的进程数量

list: 被堵塞的进程的PCB

```
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) wakeup(S->list);  
    #value<=0说明存在堵塞的进程  
}
```

## 2.4 进程同步

生产者—消费者 问题

共享数据

counter;

mutex  $\rightarrow$  value = 1

#资源信号量，堵塞数量

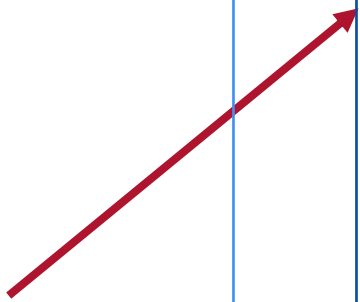
mutex  $\rightarrow$  list #堵塞队列

生产者进程 P1

```
wait(mutex)
P1.register = counter;
P1.register = register+1;
counter = P1.register;
signal(mutex) #####wakeup()?????
```

生产者进程 P2

```
wait(mutex) ##### block()?????
P2.register = counter;
P2.register = register+1;
counter = P2.register;
signal(mutex)
```



## 2.4 进程同步

生产者—消费者 问题

不会忙等，满足让权等待

共享数据 `counter;`

初始：mutex  $\rightarrow$  value=1

生产者进程 P1

```
wait(mutex)
# mutex  $\rightarrow$  value==0
P1.register = counter;
P1.register = register+1;
counter = P1.register;
signal(mutex)
##③ value== -1 wakeup()
```

生产者进程 P2

```
wait(mutex)
# ① value== -1 block()
P2.register = counter;
P2.register = register+1;
counter = P2.register;
signal(mutex)
#④ value== 0 wakeup()
```

消费者

```
wait(mutex)
# ② value== -2 block()
C.register = counter;
C.register = register-1;
counter = C.register;
signal(mutex)
#value == 1
```



## 2.4 进程同步

### 2.4.1 进程同步的基本概念

#### 4. 同步机制应遵循的规则

为实现进程互斥地进入自己的临界区，可用软件方法，更多的是在系统中设置专门的同步机构来协调各进程间的运行。所有同步机制都应遵循下述四条准则：

- (1) 空闲让进。若干进程要求进入空闲临界区时，应尽快使一进程进入临界区。
- (2) 忙则等待。临界区正在访问，则等待。
- (3) 有限等待。从进程发出进入请求到允许进入，不能无限等待。
- (4) 让权等待。当进程不能进入自己的临界区时，应释放处理机，以免陷入忙等。

## 2.4 进程同步

不满足有限等待举例：

初始：Dmutex==Emutex==1

```
process A:  
wait(Dmutex);  
wait(Emutex);
```

```
process B:  
wait(Emutex);  
wait(Dmutex);
```

死 锁 ！

若进程A和B按下述次序交替执行wait操作：

process A : wait(Dmutex); 于是Dmutex = 0

process B : wait(Emutex); 于是Emutex = 0

process A : wait(Emutex); 于是Emutex = -1 A阻塞

process B : wait(Dmutex); 于是Dmutex = -1 B阻塞

所需资源越多，发  
生死锁的概率越大

## 2.4 进程同步

### 2.4.3 信号量机制

#### 3. AND型信号量

前面所述的进程互斥问题针对的是多个并发进程仅共享一个临界资源的情况。在有些应用场合，是一个进程往往需要获得**两个或更多的共享资源**后方能执行其任务。假定现有两个进程A和B，它们都要求访问共享数据D和E，当然，共享数据都应作为临界资源。

## 2.4 进程同步

### 2.4.3 信号量机制

#### 3. AND型信号量

基本思想：将进程运行过程中所需要的资源，一次性全部分配给进程，使用完成后一起释放。

对若干个临界资源采取原子操作方式。

```
Swait(S1, S2, ..., Sn){
    while (TRUE) {
        if (Si >= 1 && ...&& Sn >= 1) {
            for (i = 1; i <= n; i++) Si--;
            break;
        }
        else {
            place the process in the waiting queue associated
            with the first Si found with Si<1, and set the
            program count of this process to the
            beginning of Swait operation
        }
    }
}
```

## 2.4 进程同步

### 2.4.3 信号量机制

#### 3. AND型信号量

基本思想：将进程运行过程中所需要的资源，一次性全部分配给进程，使用完成后一起释放。

对若干个临界资源采取原子操作方式。

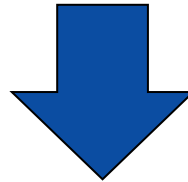
```
Ssignal(S1, S2, ..., Sn) {  
    while (TRUE) {  
        for (i = 1; i <= n; i++) {  
            Si++;  
            Remove all the process waiting  
                in the queue associated  
                with Si into the ready  
                queue  
        }  
    }  
}
```

## 2.4 进程同步

初始：Dmutex==Emutex==1

process A:  
wait(Dmutex);  
wait(Emutex);

process B:  
wait(Emutex);  
wait(Dmutex);



process A:  
Swait(Dmutex, Emutex);

process B:  
Swait(Emutex, Dmutex);

## 2.4 进程同步

### 2.4.1 进程同步的基本概念

#### 4. 同步机制应遵循的规则

为实现进程互斥地进入自己的临界区，可用软件方法，更多的是在系统中设置专门的同步机构来协调各进程间的运行。所有同步机制都应遵循下述四条准则：

- (1) 空闲让进。若干进程要求进入空闲临界区时，应尽快使一进程进入临界区。
- (2) 忙则等待。临界区正在访问，则等待。
- (3) 有限等待。从进程发出进入请求到允许进入，不能无限等待。
- (4) 让权等待。当进程不能进入自己的临界区时，应释放处理机，以免陷入忙等。

## 2.4 进程同步

### 不满足空闲让进举例

初始：mutex  $\rightarrow$  value == 1

生产者进程 P1

```
wait(mutex)
```

```
# ① mutex  $\rightarrow$  value == 0
```

```
P1.register = counter;
```

```
P1.register = register+1;
```

```
counter = P1.register;
```

```
signal(mutex)
```

```
# ② mutex  $\rightarrow$  value == 1
```

生产者进程 P2

```
wait(mutex)
```

```
# ③ mutex  $\rightarrow$  value == -1
```

```
P2.register = counter;
```

```
P2.register = register+1;
```

```
counter = P2.register;
```

资源未被释放

解决方法：

合理设置互斥信号



## 2.4 进程同步

### 不满足忙则等待举例

时间片1	P1.register = counter; P1.register = register+1;
时间片2	P2.register = counter; P2.register = register+1;
时间片3	counter = P1.register;
时间片4	counter = P2.register;

**解决方法：**

设置整形信号量

## 2.4 进程同步

### 不满足让权等待举例

生产者P1

**wait(mutex)** #检查并给counter加锁

P.register = counter;

P.register = register+1;

生产者P2

**wait(mutex)** #检查counter的锁

生产者P1

counter = P.register;

**signal(mutex)** #给counter开锁

生产者P2

检查并给counter加锁

P2.register = counter;

P2.register = register+1;

counter = C.register;

**signal(mutex)** #给counter开锁

**解决方法：**

设置记录型信号

## 2.4 进程同步

不满足有限等待举例：

初始：Dmutex==Emutex==1

```
process A:  
wait(Dmutex);  
wait(Emutex);
```

```
process B:  
wait(Emutex);  
wait(Dmutex);
```

若进程A和B按下述次序交替执行wait操作：

process A : wait(Dmutex); 于是Dmutex = 0

process B : wait(Emutex); 于是Emutex = 0

process A : wait(Emutex); 于是Emutex = -1 A阻塞

process B : wait(Dmutex); 于是Dmutex = -1 B阻塞

**解决方法：**

**设置AND型信号**

## 2.4 进程同步

### 2.4.3 信号量机制

#### 4. 信号量集

在前面所述的记录型信号量机制中，wait(S)或signal(S)操作仅能对信号量施以加1或减1操作，意味着每次只能对某类临界资源进行一个单位的申请或释放。**当一次需要N个单位时，便要进行N次wait(S)操作，这显然是低效的，甚至会增加死锁的概率。**此外，在有些情况下，为确保系统的安全性，当所申请的资源数量低于某一下限值时，还必须进行管制，不予以分配。因此，当进程申请某类临界资源时，在每次分配之前，都必须测试资源的数量，判断是否大于可分配的下限值，决定是否予以分配。

$\text{Swait}(S_1, t_1, d_1, \dots, S_n, t_n, d_n);$

$\text{Ssignal}(S_1, d_1, \dots, S_n, d_n);$

## 2.4 进程同步

### 2.4.3 信号量机制

#### 4. 信号量集

资源： $S_i$

资源的分配下限值： $t_i$ ， $S_i \geq t_i$  才会分配

进程对资源的需求量： $d_i$ ，分配后 $S_i = S_i - d_i$

$\text{Swait}(S_1, t_1, d_1, \dots, S_n, t_n, d_n);$

$\text{Signal}(S_1, d_1, \dots, S_n, d_n);$

1)  $\text{Swait}(s, d, d)$

2)  $\text{Swait}(S, 1, 1)$       若 $S > 1$ ，则等价于记录型信号量；若 $S = 1$ ，则等价于互斥信号量。

3)  $\text{Swait}(S, 1, 0)$       若 $S \geq 1$ ，允许多个进程进入某特定区域；若 $S = 0$ ，阻止任何进程进入某特定区域。

## 2.4 进程同步

### 2.4.4 信号量的应用

#### 1. 利用信号量实现进程互斥

为使多个进程能互斥地访问某临界资源，只需为该资源设置一互斥信号量mutex，并设其初始值为1，然后将各进程访问该资源的临界区CS置于wait(mutex)和signal(mutex)操作之间即可。

```
semaphore mutex = 1;
```

```
PA() {  
    while (1) {  
        wait(mutex);  
        临界区 ;  
        signal(mutex);  
        剩余区;  
    }  
}
```

```
PB() {  
    while (1) {  
        wait(mutex);  
        临界区 ;  
        signal(mutex);  
        剩余区;  
    }  
}
```

## 2.4 进程同步

### 2.4.4 信号量的应用

#### 2. 利用信号量实现前趋关系

还可利用信号量来描述程序或语句之间的前趋关系。设有两个并发执行的进程 $P_1$ 和 $P_2$ 。 $P_1$ 中有语句 $S_1$ ； $P_2$ 中有语句 $S_2$ 。我们希望在 $S_1$ 执行后再执行 $S_2$ 。为实现这种前趋关系，只需使进程 $P_1$ 和 $P_2$ 共享一个公用信号量 $S$ ，并赋予其初值为0，将 $\text{signal}(S)$ 操作放在语句 $S_1$ 后面，而在 $S_2$ 语句前面插入 $\text{wait}(S)$ 操作，即

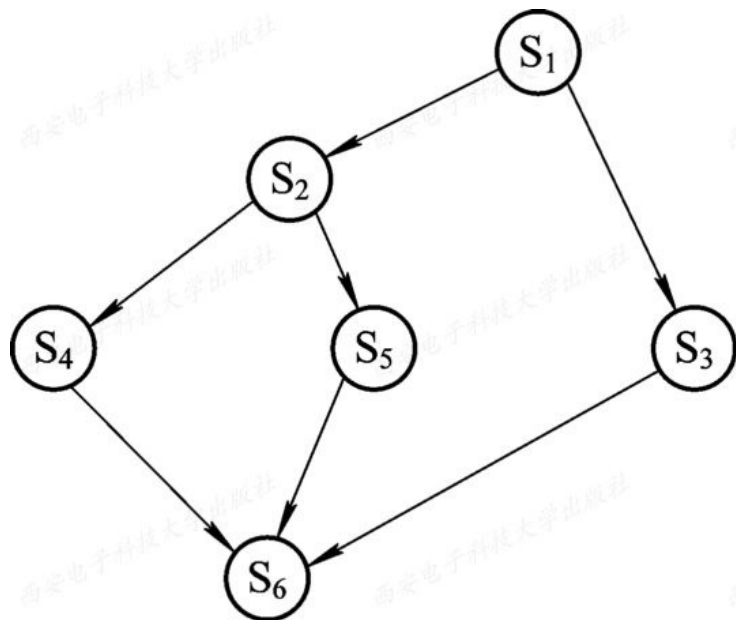
在进程 $P_1$ 中，用 $S_1$ ； $\text{signal}(S)$ ；

在进程 $P_2$ 中，用 $\text{wait}(S)$ ； $S_2$ ；

## 2.4 进程同步

### 2.4.4 信号量的应用

#### 2. 利用信号量实现前趋关系



```
p1() { S1; signal(a); signal(b); }
```

```
p2() { wait(a); S2; signal(c); signal(d); }
```

```
p3() { wait(b); S3; signal(e); }
```

```
p4() { wait(c); S4; signal(f); }
```

```
p5() { wait(d); S5; signal(g); }
```

```
p6() { wait(e); wait(f); wait(g); S6; }
```

```
main() {
```

```
    semaphore a, b, c, d, e, f, g;
```

```
    a.value = b.value = c.value = 0;
```

```
    d.value = e.value = 0;
```

```
    f.value = g.value = 0;
```

```
    cobegin
```

```
        p1(); p2(); p3(); p4(); p5(); p6();
```

```
    coend
```

```
}
```



## 2.4 进程同步

### 2.4.2 硬件同步机制

虽然可以利用软件方法解决诸进程互斥进入临界区的问题，但有一定难度，并且存在很大的局限性，因而现在已很少采用。相应地，目前许多计算机已提供了一些特殊的硬件指令，**允许对一个字中的内容进行检测和修正，或者是对两个字的内容进行交换**等。可利用这些特殊的指令来解决临界区问题。



## 2.4 进程同步

### 2.4.2 硬件同步机制

#### 1. 关中断

关中断是实现互斥的最简单的方法之一。**在进入锁测试之前关闭中断，直到完成锁测试并上锁之后才能打开中断。这样，进程在临界区执行期间，计算机系统不响应中断，从而不会引发调度，也就不会发生进程或线程切换。**由此，保证了对锁的测试和关锁操作的连续性和完整性，有效地保证了互斥。

## 2.4 进程同步

### 2.4.2 硬件同步机制

#### 1. 关中断

关中断是实现互斥的最简单的方法之一。

但是，关中断的方法存在许多缺点：

- ① 滥用关中断权力可能导致严重后果；
- ② 关中断时间过长，会影响系统效率，限制了处理器交叉执行程序的能力；
- ③ 关中断方法也不适用于多CPU 系统，因为在一个处理器上关中断并不能防止进程在其它处理器上执行相同的临界段代码。

## 2.4 进程同步

### 2.4.2 硬件同步机制

#### 2. 利用Test-and-Set指令实现互斥

这是一种借助一条硬件指令 —— “测试并建立” 指令TS(Test-and-Set)以实现互斥的方法。在许多计算机中都提供了这种指令。

#### 原子操作

## 2.4 进程同步

### 2.4.2 硬件同步机制

#### 2. 利用Test-and-Set指令实现互斥

##### 原子操作

每个临界值有一个\*lock, \*lock = FALSE表示资源空闲, \*lock = TRUE表示资源被占用。

##### TS 指令描述

```
boolean TS(boolean *lock) {  
    boolean old;  
    old = *lock;  
    *lock = TRUE;  
    return old;  
}
```

##### 利用TS指令实现互斥的循环进程结构

```
do {  
    ...  
    while TS(&lock); /*do skip*/  
    critical section;  
    lock = FALSE;  
    remainder section;  
} while (TRUE);
```

- \*lock==TRUE,\*lock被置为TRUE, 返回TRUE陷入循环等待;
- \*lock==FALSE,\*lock被置为TRUE, 返回FALSE, 进入临界区;

**FALSE→TRUE; TRUE→TRUE**

**忙等, 非让权等待**

## 2.4 进程同步

### 2.4.2 硬件同步机制

#### 3. 利用Swap指令实现进程互斥

该指令称为对换指令，在Intel 80x86中又称为XCHG指令，用于交换两个字的内容。

全局变量 lock，局部布尔变量key。

##### Swap处理过程描述

```
void swap(boolean *a, boolean *b) {  
    boolean temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

##### 利用Swap指令实现进程互斥的循环进程描述

```
do {  
    key = TRUE;  
    do {  
        swap(&lock, &key);  
    } while (key != FALSE);  
    临界区操作;  
    lock = FALSE;  
    ...  
} while (TRUE);
```

- \*key==TRUE,\*lock==TRUE,  
陷入循环等待；
- \*key==TRUE, \*lock==FALSE,  
\*lock被设置为TRUE，锁定，  
进入临界区

忙等，非让权等待

## 2.4 进程同步

### 2.4.5 管程机制

虽然信号量机制是一种既方便、又有效的进程同步机制，但每个要访问的临界资源的进程都必须自备同步操作wait(S)和Signal(S)，导致大量的同步操作分散在各个进程中。给系统管理带来麻烦，操作不当导致死锁。为解决上述问题产生新的同步工具管程（Monitors）。

## 2.4 进程同步

### 2.4.5 管程机制

生产者—消费者 问题

共享数据 `counter;`

初始：mutex  $\rightarrow$  value=1

生产者进程 P1

```
wait(mutex)
# mutex  $\rightarrow$  value==0
P1.register = counter;
P1.register = register+1;
counter = P1.register;
signal(mutex)
##③ value== -1 wakeup()
```

生产者进程 P2

```
wait(mutex)
# ① value== -1 block()
P2.register = counter;
P2.register = register+1;
counter = P2.register;
signal(mutex)
#④ value== 0 wakeup()
```

消费者

```
wait(mutex)
# ② value== -2 block()
C.register = counter;
C.register = register-1;
counter = C.register;
signal(mutex)
#value == 1
```

提取公共操作的部分作为新的  
数据结构与相应操作  $\rightarrow$  管程





## 2.4 进程同步

### 2.4.5 管程机制

生产者—消费者 问题

共享数据 `counter;`

初始: `mutex → value == 1`

`P1.put(x)`

`P2.put(x)`

`C.get(x)`



生产者进程 P1

`wait(mutex)`

`# mutex → value == 0`

`P1.register = counter;`

`P1.register = register + 1;`

`counter = P1.register;`

`signal(mutex)`

`## ③ wakeup() value == -1`

生产者进程 P2

`wait(mutex)`

`# ① block() value == -1`

`P2.register = counter;`

`P2.register = register + 1;`

`counter = P2.register;`

`signal(mutex)`

`# ④ wakeup() value == 0`

消费者

`wait(mutex)`

`# ② block() value == -2`

`C.register = counter;`

`C.register = register - 1;`

`counter = C.register;`

`signal(mutex)`

`# value == 1`

## 2.4 进程同步

### 2.4.5 管程机制

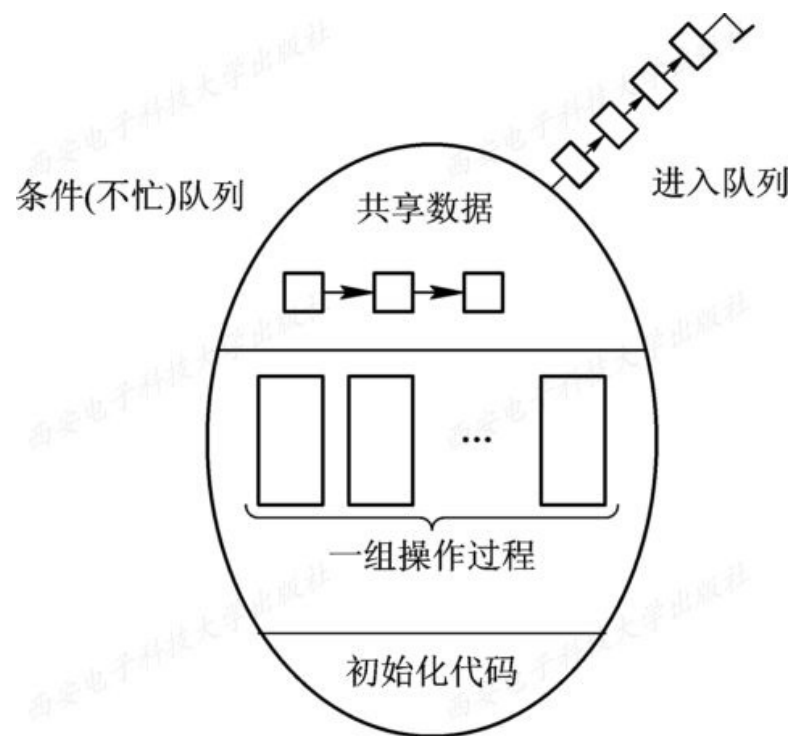
#### 1. 管程的定义

代表**共享资源的数据结构**以及由**对该共享数据结构实施操作的一组过程**所组成的**资源管理程序**共同构成了操作系统的资源管理模块，称之为管程。

**实现临界资源的统一管理。**

管程由四部分组成：

- ① 管程的名称；
- ② 局部于管程的共享数据结构说明；
- ③ 对该数据结构进行操作的一组过程；
- ④ 对局部于管程的共享数据设置初始值的语句。



## 2.4 进程同步

### 2.4.5 管程机制

#### 1. 管程的定义

```
Monitor monitor_name{           /*管程名*/
    share variable declarations; /*共享变量说明*/
    cond declarations;          /*条件变量说明*/
    public :                     /*能被进程调用的过程*/
    void P1(.....)              /*对数据结构操作的过程*/
    {.....}
    void P2(.....)
    {.....}
    .....
    void(.....)
    {.....}
    .....
    {                             /*管程主体*/
        initialization code; /*初始化代码*/
        .....
    }
}
```

## 2.4 进程同步

### 2.4.5 管程机制

#### 1. 管程

管程是一种程序设计语言的结构成分，他和信号量具有同等的表达能力，从语言的角度看，管程主要具有以下特性：

- ① **模块化**，即管程是一个基本程序单位，可以单独编译；
- ② **抽象数据类型**，指管程中不仅有数据，而且有对数据的操作；
- ③ **信息掩蔽**，指管程中的数据结构只能被管程中的过程访问，这些过程也是在管程内部定义的，供管程外的进程调用，而管程中的数据结构以及过程（函数）的具体实现外部不可见。

## 2.4 进程同步

### 2.4.5 管程机制

#### 1. 管程

管程和进程的不同：

- ① 虽然两者都定义了数据结构，但进程定义的是私有数据结构PCB，管程定义的是公共数据结构，如消息队列等；
- ② 二者都存在对各自数据结构上的操作，但进程是由顺序程序执行有关操作，而管程主要是进行同步操作和初始化操作；
- ③ 设置进程的目的在于实现系统的并发性，而管程的设置则是解决**共享资源的互斥**使用问题；

## 2.4 进程同步

### 2.4.5 管程机制

#### 1. 管程

管程和进程的不同：

- ④ 进程通过调用管程中的过程对共享数据结构实行操作，该过程就如通常的子程序一样被调用，因而管程为**被动工作方式**，进程则为**主动工作方式**；
- ⑤ 进程之间能并发执行，而管程则不能与其调用者并发；
- ⑥ 进程具有动态性，由“创建”而诞生，由“撤消”而消亡，而管程则是操作系统中的一个资源管理模块，供进程调用。

## 2.4 进程同步

### 2.4.5 管程机制

#### 2. 条件变量

在利用管程实现进程同步时，必须设置同步工具，如两个同步操作原语wait和signal。当某进程通过管程请求获得临界资源而未能满足时，管程便调用wait原语使该进程等待，并将其排在等待队列上。仅当另一进程访问完成并释放该资源之后，管程才又调用signal原语，唤醒等待队列中的队首进程。

条件：堵塞的进程、队列

```
Monitor monitor_name{           /*管程名*/
    share variable declarations; /*共享变量说明*/
    cond declarations;          /*条件变量说明*/
    public :                     /*能被进程调用的过程*/
    void P1(.....)              /*对数据结构操作的过程*/
    {.....}
    void P2(.....)
    {.....}
    .....
    void(.....)
    {.....}
    .....
    {                             /*管程主体*/
        initialization code; /*初始化代码*/
        .....
    }
}
```

## 2.4 进程同步

### 2.4.5 管程机制

#### 2. 条件变量

但是仅仅有上述的同步工具是不够的，考虑一种情况：当一个进程调用了管程，在管程中时被阻塞或挂起，直到阻塞或挂起的原因解除，而在此期间，如果该进程不释放管程，则其他进程无法进入管程，被迫长时间的等待。为了解决这个问题，引入了条件变量condition。通常，一个进程被阻塞或挂起的条件（原因）可以有多个，因此在管程中设置了多个条件变量，对这些条件变量的访问只能在管程中进行。

```
Monitor monitor_name{           /*管程名*/
    share variable declarations; /*共享变量说明*/
    cond declarations;          /*条件变量说明*/
    public :                     /*能被进程调用的过程*/
    void P1(.....)              /*对数据结构操作的过程*/
    {.....}
    void P2(.....)
    {.....}
    .....
    void(.....)
    {.....}
    .....
    {                             /*管程主体*/
        initialization code; /*初始化代码*/
        .....
    }
}
```



## 2.4 进程同步

### 2.4.5 管程机制

#### 2. 条件变量

管程中对每个条件变量都需要予以说明，其形式为：condition x, y; 对条件变量的操作仅是 wait 和 signal，因此条件变量也是一种抽象数据结构类型，**每个条件变量保存了一个链表，用于记录因该条件变量而阻塞的所有进程**，同时提供的两个操作即可表示为 x.wait 和 x.signal，其含义为：

① x.wait：正在调用管程的进程因x条件需要被阻塞或挂起，则调用 x.wait 将自己插入到 x 条件的等待队列上，并释放管程，直到x条件无变化。此时其他进程可以使用该管程。

② x.signal：正在调用管程的进程发现x条件发生了变化，则调用 x.signal，重新启动一个因 x 条件而阻塞或挂起的进程，如果存在多个这样的进程，则选择其中的一个，如果没有，继续执行原进程，而不产生任何结果。这与信号量机制中的signal操作不同。因为，后者总要执行  $s=s+1$  操作，因而总会改变信号量的状态。

```
Monitor monitor_name{           /*管程名*/
    share variable declarations; /*共享变量说明*/
    cond declarations;          /*条件变量说明*/
    public :                     /*能被进程调用的过程*/
    void P1(.....)              /*对数据结构操作的过程*/
    {.....}
    void P2(.....)
    {.....}
    .....
    void(.....)
    {.....}
    .....
    {                             /*管程主体*/
        initialization code; /*初始化代码*/
        .....
    }
}
```

## 2.4 进程同步

### 2.4.5 管程机制

#### 2. 条件变量

如果有进程Q因  $x$  条件处于阻塞状态，当正在调用管程的进程P执行了  $x.signal$  操作后，进程Q被重新启动，此时两个进程P和Q，如何确定哪个执行哪个等待，可采用下述两种方式之一进行处理：

- (1) P等待，直至Q离开管程或等待另一条件。
- (2) Q等待，直至P离开管程或等待另一条件。

采用哪种处理方式，当然是各执一词。Hoare采用第一种处理方式，而Hansan选择两者的折中，他规定管程中的过程所执行的signal操作是过程体的最后一个操作，于是，进程P执行signal操作后立即退出管程，因而，进程Q马上被恢复执行。

```
Monitor monitor_name{           /*管程名*/
    share variable declarations; /*共享变量说明*/
    cond declarations;           /*条件变量说明*/
    public :                      /*能被进程调用的过程*/
    void P1(.....)              /*对数据结构操作的过程*/
    {.....}
    void P2(.....)
    {.....}
    .....
    void(.....)
    {.....}
    .....
    {                             /*管程主体*/
        initialization code; /*初始化代码*/
        .....
    }
}
```



# 第二章 进程的描述与控制



2.1 前趋图和程序执行

2.2 进程的描述

2.3 进程控制

2.4 进程同步

2.5 经典进程的同步问题

2.6 进程通信

2.7 线程(Threads)的基本概念

2.8 线程的实现

习题



## 2.5 经典进程的同步问题



在多道程序环境下，进程同步问题十分重要，也是相当有趣的问题，因而吸引了不少学者对它进行研究，由此而产生了一系列经典的进程同步问题，其中较有代表性的是“生产者—消费者”问题、“读者—写者问题”、“哲学家进餐问题”等等。通过对这些问题的研究和学习，可以帮助我们更好地理解进程同步的概念及实现方法。

## 2.5 经典进程的同步问题

### 2.5.1 生产者-消费者问题

#### 1. 利用记录型信号量解决生产者-消费者问题

假定在生产者和消费者之间的公用缓冲池中具有 $n$ 个缓冲区，这时可利用互斥信号量mutex实现诸进程对缓冲池的互斥使用；

利用信号量empty和full分别表示缓冲池中空缓冲区和满缓冲区的数量。

又假定这些生产者和消费者相互等效，只要缓冲池未满，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。

## 2.5 经典进程的同步问题

## 标记法 + 轮转法

### 2.5.1 生产者-消费者问题

#### 1. 利用记录型信号量解决生产者-消费者问题

int in = 0, out = 0;   #位置指针  
item buffer[n];       #共享资源  
semaphore mutex = 1, empty = n, full = 0; #信号量

```
void producer() {  
    do {  
        produce an item nextp;  
        ...  
        wait(empty);   #判断是否有空间  
        wait(mutex);  
        buffer[in] = nextp;  
        in = (in + 1) % n;  
        signal(mutex);  
        signal(full);  
    } while (TRUE);  
}
```

```
void consumer() {  
    do {  
        wait(full);   #判断是否有资源  
        wait(mutex);  
        nextc = buffer[out];  
        out = (out + 1) % n;  
        signal(mutex);  
        signal(empty);  
        consumer the item in nextc;  
        ...  
    } while (TRUE)  
}  
  
void main() {  
    cobegin  
        producer(); consumer();  
    coend;  
}
```

## 2.5 经典进程的同步问题

## 标记法 + 轮转法

### 2.5.1 生产者-消费者问题

#### 1. 利用记录型信号量解决生产者-消费者问题

生产者-消费者问题可描述如下：

```
int in = 0, out = 0;
item buffer[n];
semaphore mutex = 1, empty = n, full = 0;

void producer() {
    do {
        produce an item nextp;
        ...
        wait(empty);
        wait(mutex);
        buffer[in] = nextp;
        in = (in + 1) % n;
        signal(mutex);
        signal(full);
    } while (TRUE);
}

void consumer() {
    do {
        wait(full);
        wait(mutex);
        nextc = buffer[out];
        out = (out + 1) % n;
        signal(mutex);
        signal(empty);
        consumer the item in nextc;
        ...
    } while(TRUE)
}

void main() {
    cobegin
        producer(); consumer();
    coend;
}
```

**先 wait(empty) 和 wait(full) 对buffer进行记录，两者一定有一个是可以满足的，即生产者消费者两个一定至少可以满足其中的一个。**

- **empty=n, full=0时,buffer空，可以进入producer**
- **empty=0, full=n,buffer满，可以进入consumer**
- **n>empty>0, n>full>0,buffer未满足空，都可进入，对buffer的读写控制权由mutex决定**

# 2.5 经典进程的同步问题

## 标记法 + 轮转法

### 2.5.1 生产者-消费者问题

#### 1. 利用记录型信号量解决生产者-消费者问题

生产者-消费者问题可描述如下：

```
int in = 0, out = 0;
item buffer[n];
semaphore mutex = 1, empty = n, full = 0;
```

```
void producer() {
    do {
        produce an item nextp;
        ...
        wait(mutex); #①mutex 1→0
        wait(empty); #②empty = 0 block
        buffer[in] = nextp;
        in = (in + 1) % n;
        signal(full);
        signal(mutex);
    } while (TRUE);
}
```

```
void consumer() {
    do {
        wait(mutex); #③mutex=0 block
        wait(full);
        nextc = buffer[out];
        out = (out + 1) % n;
        signal(empty);
        signal(mutex);
        consumer the item in nextc;
        ...
    } while (TRUE)
}

void main() {
    cobegin
        producer(); consumer();
    coend;
}
```

先wait资源信号量  
再wait互斥信号量  
否则具有死锁的可能性

P: wait(mutex) wait(empty)  
C: wait(mutex)

P等C的empty ; C等P的  
mutex。互等，死锁

满足：空闲让进。  
忙则等待。  
让权等待。  
不满足：有限等待。



## 2.5 经典进程的同步问题

### 2.5.1 生产者-消费者问题

#### 2. 利用AND信号量解决生产者-消费者问题

对于生产者-消费者问题，也可利用AND信号量来解决，

- 用Swait(empty, mutex)来代替wait(empty)和wait(mutex)；
- 用Signal(mutex, full)来代替signal(mutex)和signal(full)；
- 用Swait(full, mutex)代替wait(full)和wait(mutex)；
- 用Signal(mutex, empty)代替Signal(mutex)和Signal(empty)。

## 2.5 经典进程的同步问题

### 2.5.1 生产者-消费者问题

#### 2. 利用AND信号量解决生产者-消费者问题

利用AND信号量来解决生产者-消费者问题的算法中的生产者和消费者可描述如下：

```
int in = 0, out = 0;
item buffer[n];
semaphore mutex = 1, empty = n, full = 0;
```

```
void producer() {
    do {
        produce an item nextp;
        ...
        Swait(empty , mutex);
        buffer[in] = nextp;
        in = (in + 1) % n;
        Ssignal(mutex , full);
    } while (TRUE);
}
```

```
void consumer() {
    do {
        Swait(full , mutex);
        nextc = buffer[out];
        out = (out + 1) % n;
        Ssignal(mutex , empty);
        consumer the item in nextc;
        ...
    } while (TRUE) ;
}
```

## 2.5 经典进程的同步问题

### 2.5.1 生产者-消费者问题

#### 3. 利用管程解决生产者-消费者问题

在利用管程方法来解决生产者-消费者问题时，首先便是为它们建立一个管程，并命名为 `producerconsumer`，或简称为PC。其中包括两个过程：

- (1) `put(x)`过程。
- (2) `get(x)`过程。



## 2.5 经典进程的同步问题

### 2.5.1 生产者-消费者问题

#### 3. 利用管程解决生产者-消费者问题

PC管程可描述为：

对于条件变量notfull和notempty，分别有两个过程

cwait和csignal对它们进行操作：

(1) cwait(condition)过程：当管程被一个进程占用时，其他进程调用该过程时阻塞，并挂在条件condition的队列上。

(2) csignal(condition)过程：唤醒在cwait执行后阻塞在条件condition队列上的进程，如果这样的进程不止一个，则选择其中一个实施唤醒操作；如果队列为空，则无操作而返回。

```

Monitor producerconsumer{
    item buffer[N];
    int in, out;
    condition notfull, notempty, mutex;
    #因为notfull和notempty是链表，故需要count来辅助
    int count; #缓冲区中的资源数量
    public:
    void put(item x) {
        if (count >= N) cwait(notfull); #堵塞在链表中，等待被唤醒
        cwait(mutex);
        buffer[in] = x;
        in = (in + 1) % N;
        count++;
        csignal(mutex);
        csignal(notempty);
    }
    void get(item x) {
        if (count <= 0) cwait(notempty); #堵塞在链表中，等待被唤醒
        cwait(mutex);
        x = buffer[out];
        out = (out + 1) % N;
        count--;
        csignal(mutex);
        csignal(notfull);
    }

    {in = 0; out = 0; count = 0; }
}PC;

```

## 2.5 经典进程的同步问题

### 2.5.1 生产者-消费者问题

#### 3. 利用管程解决生产者-消费者问题

在利用管程解决生产者-消费者问题时，其中的生产者和消费者可描述为：

```
void producer() {
    item x;
    while (TRUE) {
        ...
        produce an item in nextp;
        PC.put(x);
    }
}

void consumer() {
    item x;
    while (TRUE) {
        PC.get(x);
        consume the item in nextc;
        ...
    }
}
```

```
void main() {
    cobegin
        producer(); consumer();
    coend
}
```

## 2.5 经典进程的同步问题

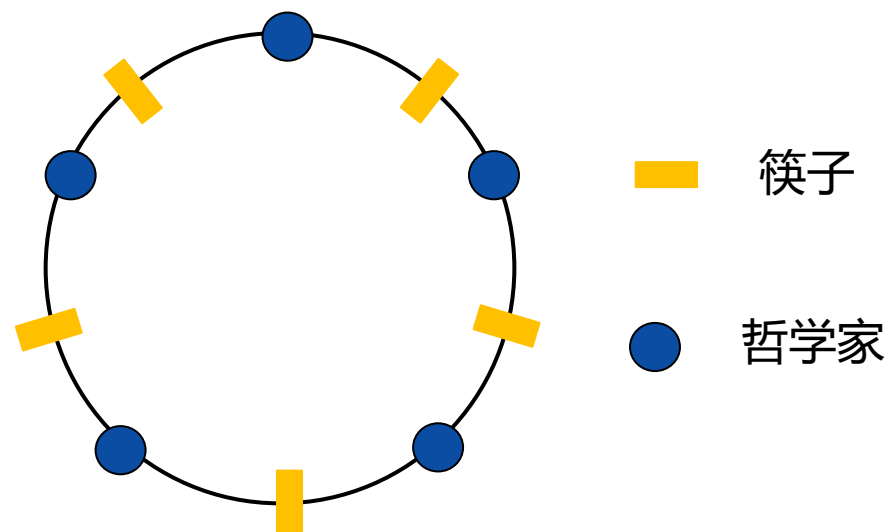
### 2.5.2 哲学家进餐问题

#### 1. 利用记录型信号量解决哲学家进餐问题

经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。其描述如下：

```
semaphore chopstick[5] = { 1,1,1,1,1 };
```

所有信号量均被初始化为1，第  $i$  位哲学家的活动可描述为：



## 2.5 经典进程的同步问题

### 2.5.2 哲学家进餐问题

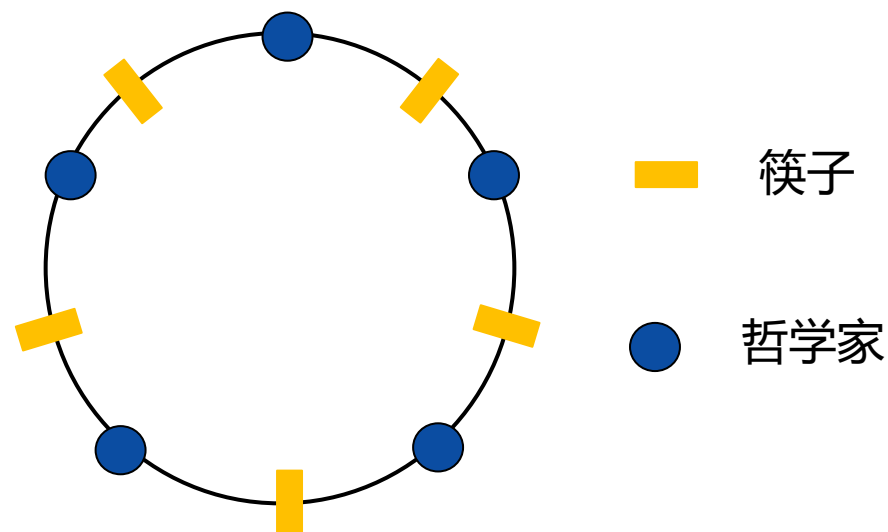
#### 1. 利用记录型信号量解决哲学家进餐问题

经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。

其描述如下：

```
void phi{  
    do {  
        wait(chopstick[i]);  
        wait(chopstick[(i + 1) % 5]);  
        ...  
        //eat  
        ...  
        signal(chopstick[i]);  
        signal(chopstick[(i + 1) % 5]);  
        ...  
        //think  
        ...  
    } while (TRUE);  
}
```

具有死锁的风险



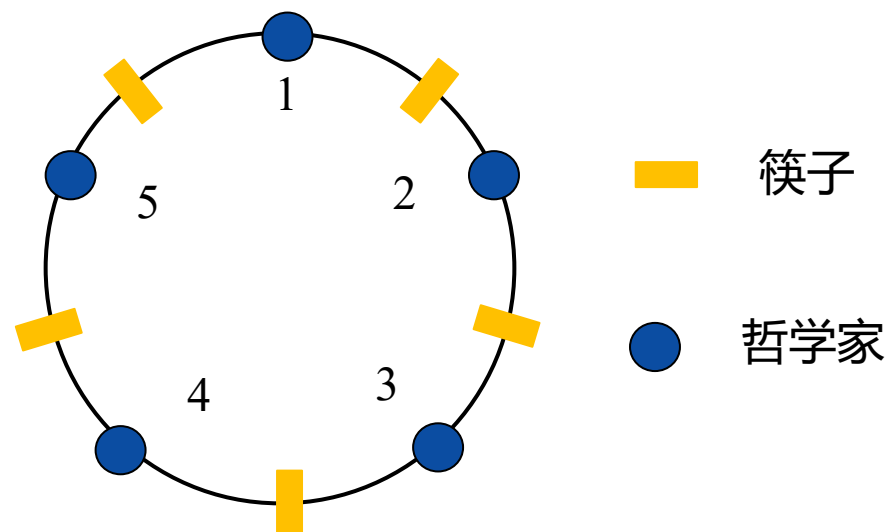
## 2.5 经典进程的同步问题

### 2.5.2 哲学家进餐问题

#### 1. 利用记录型信号量解决哲学家进餐问题

解除死锁的方法

- ①至多只允许四位哲学家同时进餐，保证至少一位能够同时获得两只筷子；
- ②仅当哲学家左、右两只筷子同时可用时，才允许哲学家拿起筷子进餐；
- ③规定奇数号的哲学家竞争左边的筷子，  
然后再拿右边的筷子，偶数号哲学家相反。





## 2.5 经典进程的同步问题

### 2.5.2 哲学家进餐问题

#### 2. 利用AND信号量机制解决哲学家进餐问题

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源(筷子)后方能进餐，这在本质上就是前面所介绍的AND同步问题，故用AND信号量机制可获得最简洁的解法。

```
semaphore chopstick[5] = { 1,1,1,1,1 };  
do {  
    ...  
    //think  
    ...  
    Swait(chopstick[(i + 1) % 5], chopstick[i]);  
    ...  
    //eat  
    ...  
    Ssignal(chopstick[(i + 1) % 5], chopstick[i]);  
} while(TRUE);
```

## 2.5 经典进程的同步问题

**题目：**试利用记录型信号量写出一个，至多只有四位哲学家去拿左边的筷子，不会死锁的哲学家进餐问题的算法。

分析：此题有多种解决方法。其中之一是只允许4个哲学家同时进餐，以保证至少有一个哲学家可以进餐，最终才可能由他释放出其所用过的两只筷子，从而使更多的哲学家可以进餐。为此，需设置一个信号量 $S_m$ 来限制同时进餐的哲学家数目，使它不超过4，故 $S_m$ 的初值也应置成4（也可想象成桌上有4块令牌，只有拿到令牌的哲学家才能进餐，那样，令牌就是一种临界资源，故可为它们设置一个初值为4的信号量 $S_m$ ）

`semaphore sm=4`

## 2.5 经典进程的同步问题

**题目：**试利用记录型信号量写出一个，至多只有四位哲学家去拿左边的筷子，不会死锁的哲学家进餐问题的算法。

答：除了为每只筷子设置一个初值为{1, NULL}的记录型信号量chopstick[i](i=0, ..., 4)外，还需再设置一个初值为{4, NULL}的记录型信号量Sm，以限制同时就餐的哲学家人数不超过4。第i个哲学家的活动可描述为:

```
semaphore Sm = 4, chopstick=[1,1,1,1,1];

Pi() {
    while (1) {
        wait(Sm);
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
        eat;
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
        signal(Sm);
        think;
    }
}
```

## 2.5 经典进程的同步问题

### 2.5.3 读者-写者问题

#### 1. 利用记录型信号量解决读者-写者问题

写者间、写者与读者均互斥，读者间可共享

- 互斥信号量wmutex：Reader与Writer进程间在读或写时的互斥；
- 整型变量readcount：正在读的进程数目；
- 互斥信号量rmutex：readcount是一个可被多个Reader访问的临界资源，**保护readcount不被多个读者并发改乱**
- 仅当readcount=0，表示尚无Reader进程在读时，Reader进程才需要执行Wait(wmutex)操作。若wait(wmutex)操作成功，Reader进程便可去读，相应地，做readcount+1操作。
- 仅当Reader进程执行了readcount-1操作后其值为0时，才必须执行signal(wmutex)操作，以便让Write进程写操作。

## 2.5 经典进程的同步问题

### 2.5.3 读者-写者问题

#### 1. 利用记录型信号量解决读者-写者问题

读者-写者问题可描述如下：

```
semaphore rmutex = 1, wmutex = 1;
int readcount = 0;
void Reader() {
    do {
        wait(rmutex);
        if (readcount == 0) wait(wmutex);
        readcount++;
        signal(rmutex);
        ...
        perform read operation;
        ...
        wait(rmutex);
        readcount--;
        if (readcount == 0) signal(wmutex);
        signal(rmutex);
    } while (TRUE);
}
```

```
void Writer() {
    do {
        wait(wmutex);
        perform write operation;
        signal(wmutex);
    } while (TRUE);
}

void main() {
    cobegin
        Reader(); Writer();
    coend
}
```

## 2.5 经典进程的同步问题

### 2.5.3 读者-写者问题

#### 2. 利用信号量集机制解决读者-写者问题

这里的读者—写者问题，与前面的略有不同，它增加了一个限制，即**最多只允许RN个读者同时读**。为此，又引入了一个信号量L，并赋予其初值为RN，通过执行Swait(L, 1, 1)操作来控制读者的数目，每当有一个读者进入时，就要先执行Swait(L, 1, 1)操作，使L的值减1。当有RN个读者进入读后，L便减为0，第RN + 1个读者要进入读时，必然会因wait(L, 1, 1)操作失败而阻塞。

## 2.5 经典进程的同步问题

### 2.5.3 读者-写者问题

#### 2. 利用信号量集机制解决读者-写者问题

利用信号量集来解决读者-写者问题的描述如下：

```
semaphore L = RN, mx = 1;
void Reader() {
    do {
        Swait(L, 1, 1);
        #L≥1时, 消耗1, 进入
        Swait(mx, 1, 0);
        #mx≥1时, 允许进入
        ...
        perform read operation;
        ...
        Ssignal(L, 1);
    } while (TRUE);
}

void Writer() {
    do {
        Swait(mx, 1, 1; L, RN, 0);
        # mx≥1时, 消耗1, 进入
        # L≥RN时, L的名额还有RN个,
        即无读者, 允许进入
        perform write operation;
        Ssignal(mx, 1);
    } while (TRUE);
}

void main() {
    cobegin
        Reader(); Writer();
    coend
}
```

mx：控制写者的互斥信号量

L：控制读者的整形信号量，记录有几个reader

## 2.5 经典进程的同步问题

**题目：**桌上有一个能盛得下五个水果的空盘子。爸爸不停的向盘子中放苹果或者桔子，儿子不停的从盘子中取出桔子享用，女儿不停的从盘中取出苹果享用。规定三人不能同时使用盘子。试用信号量实现爸爸、儿子、女儿这三个循环进程之间的同步。

**分析：**本题是生产者-消费者问题的变形，相当于一个能生产两种产品的生产者（爸爸）向两个消费者（儿子和女儿）提供产品的同步问题，因此，需要设置两个不同的full信号量apple和orange，初值均为0。

```
semaphore empty = 5, orange = 0, apple = 0, mutex = 1;
```



## 2.5 经典进程的同步问题

**题目：**桌上有一个能盛得下五个水果的空盘子。爸爸不停的向盘子中放苹果或者桔子，儿子不停的从盘子中取出桔子享用，女儿不停的从盘中取出苹果享用。规定三人不能同时使用盘子。试用信号量实现爸爸、儿子、女儿这三个循环进程之间的同步。

**答：**定义如下信号量：

semaphore empty = 5, orange = 0, apple = 0, mutex = 1;

爸爸、儿子和女儿的算法可描述为：

```
Dad() {  
    while (1) {  
        wait(empty);  
        wait(mutex);  
        将水果放入盘中;  
        signal(mutex);  
        if (放入的是桔子) signal(orange);  
        else signal(apple);  
    }  
}
```

```
Son() {  
    while (1) {  
        wait(orange);  
        wait(mutex);  
        从盘中取一个桔子;  
        signal(mutex);  
        signal(empty);  
        享用桔子;  
    }  
}
```

```
Daughter() {  
    while (1) {  
        wait(apple);  
        wait(mutex);  
        从盘中取一个苹果;  
        signal(mutex);  
        signal(empty);  
        享用苹果;  
    }  
}
```

## 2.5 经典进程的同步问题

**题目：**请用信号量解决以下的“过桥”问题：同一方向的行人可连续过桥，当某一方向有人过桥时，另一方向的行人必须等待；当某一方向无人过桥时，另一方向的行人可以过桥。

分析：独木桥问题是读者-写者问题的一个变形，同一个方向的行人可以同时过桥，这相当于读者可以同时读。因此，可将两个方向的行人看做是两类不同的读者，同类读者（行人）可以同时读（过桥），但不同类读者（行人）之间必须互斥地读（过桥）。

与“生产者-消费者”问题不同，“生产者-消费者”可以同时并发生产者与消费者，此处不能同时包含两个方向的人。

```
int countA = 0, countB = 0;           //countA、countB分别表示A、B两个方向过桥的行人数量
semaphore bridge = 1;                 //用来实现不同方向行人对桥的互斥共享
semaphore mutexA = mutexB = 1;        //分别用来实现对countA、countB的互斥共享
```

## 2.5 经典进程的同步问题

**题目：**请用信号量解决以下的“过独木桥”问题：同一方向的行人可连续过桥，当某一方向有人过桥时，另一方向的行人必须等待；当某一方向无人过桥时，另一方向的行人可以过桥。

**答：**可为独木桥问题定义如下的变量：

```
int countA = 0, countB = 0;           //countA、countB分别表示A、B两个方向过桥的行人数量
semaphore bridge = 1;                 //用来实现不同方向行人对桥的互斥共享
semaphore mutexA = mutexB = 1;        //分别用来实现对countA、countB的互斥共享
```

A方向的所有行人对应相同的算法，B方向的所有行人也对应相同的算法，他们的动作的算法可描述为：

```
PA() {
    wait(mutexA);
    if (countA == 0) wait(bridge);
    countA++;
    signal(mutexA);
    过桥;
    wait(mutexA);
    countA--;
    if (countA == 0) signal(bridge);
    signal(mutexA);
}
```

```
PB() {
    wait(mutexB);
    if (countB == 0) wait(bridge);
    countB++;
    signal(mutexB);
    过桥;
    wait(mutexB);
    countB--;
    if (countB == 0) signal(bridge);
    signal(mutexB);
}
```

## 2.5 经典进程的同步问题

**题目：**嗜睡的理发师问题。一个理发店由一个有N张沙发的等候室和一个放有一张理发椅的理发室组成。没有顾客要理发时，理发师便去睡觉。当一个顾客走进理发店时，如果所有的沙发都已经被占用，他便离开理发店；否则，如果理发师正在为其他顾客理发，则该顾客就找一张空沙发坐下等待；如果理发师因无顾客正在睡觉，则由新到的顾客唤醒理发师为其理发。在理发完成后，顾客必须付费，直到理发师收费后才能离开理发店。试用信号量实现这一同步问题。

答：为解决上述问题，

int count	用来对占用沙发的顾客进行计数；
semaphore mutex	用来实现顾客对count变量的互斥访问，其初值为1；
semaphore empty	表示是否有空闲的理发椅，其初值为1；
semaphore full	表示理发椅上是否坐有等待理发的顾客，即沙发上是否有等待的顾客，其初值为0；
semaphore payment	用来等待付费，其初值为0；
semaphore receipt	用来等待收费，其初值为0。

## 2.5 经典进程的同步问题

```
int count = 0;
semaphore mutex = 1, empty = 1, full = 0;
semaphore payment = 0, receipt = 0;
```

```
Barber() {
    while (1) {
        wait(full); //如没顾客就在此睡觉
        替顾客理发;
        wait(payment); //等待顾客付费
        收费;
        signal(receipt); //通知顾客收费完成
    }
}
```

```
guest() {
    wait(mutex);
    if (count >= N) { //沙发已被全部占用
        signal(mutex);
        离开理发店;
    } else {
        count++;
        signal(mutex);
        在沙发中就座;
        wait(empty); //等待理发椅变空
        离开沙发, 坐到理发椅上;
        wait(mutex);
        count--;
        signal(mutex);
        signal(full); //唤醒理发师
        理发;
        付费;
        signal(payment); //通知理发师付费
        wait(receipt); //等待理发师收费
        signal(empty); //离开理发椅
        离开理发店;
    }
}
```

## 2.5 经典进程的同步问题

问题1：定义信号量

问题2：区分变量和信号量，变量是临界资源 → 保护变量

问题3：先wait资源信号量，再wait互斥信号量

问题4：写入动作

问题5：桔子苹果分开写

```
int countA = 0, countB = 0;           //countA、countB分别表示A、B两个方向过桥的行人数量
semaphore bridge = 1;                 //用来实现不同方向行人对桥的互斥共享
semaphore mutexA = mutexB = 1;        //分别用来实现对countA、countB的互斥共享
```

```
PA() {
    wait(mutexA);
    if (countA == 0) wait(bridge);
    countA++;
    signal(mutexA);
    过桥;
    wait(mutexA);
    countA--;
    if (countA == 0) signal(bridge);
    signal(mutexA);
}
```



## 第二章 进程的描述与控制



2.1 前趋图和程序执行

2.2 进程的描述

2.3 进程控制

2.4 进程同步

2.5 经典进程的同步问题

2.6 进程通信

2.7 线程(Threads)的基本概念

2.8 线程的实现

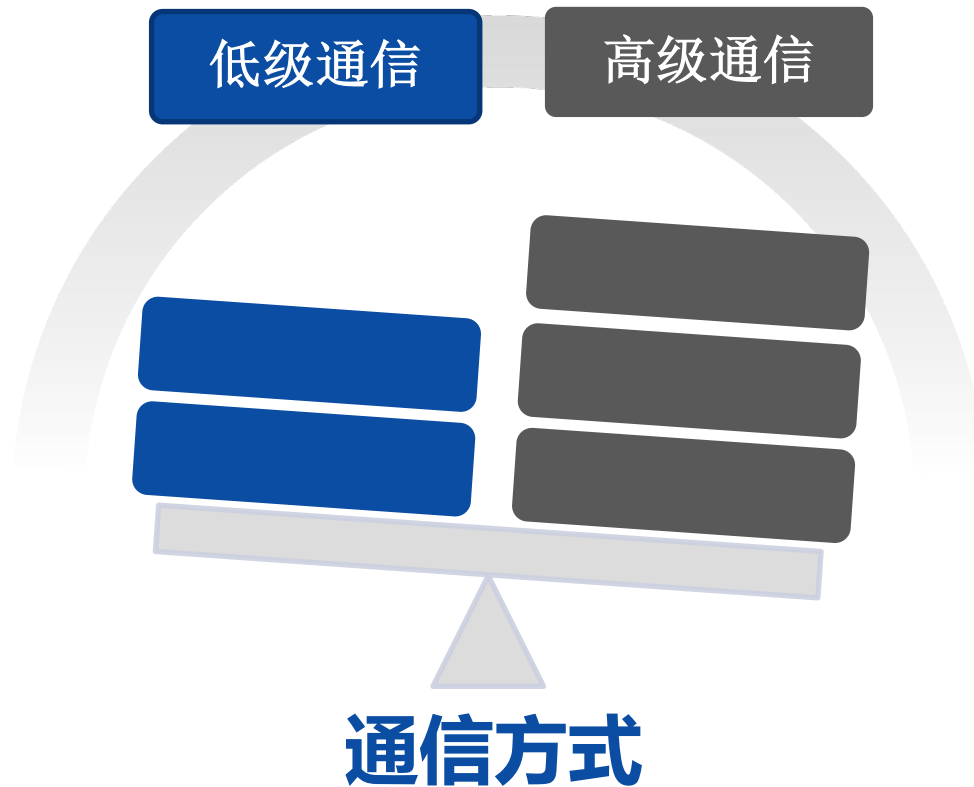
习题



# 进程通信是指进程之间的**信息交换**。

### (以信号量机制为例)

- 效率低 (通信量少)
- 通信对用户不透明 (程序员实现, 操作系统只提供共享存储器供代码操作)

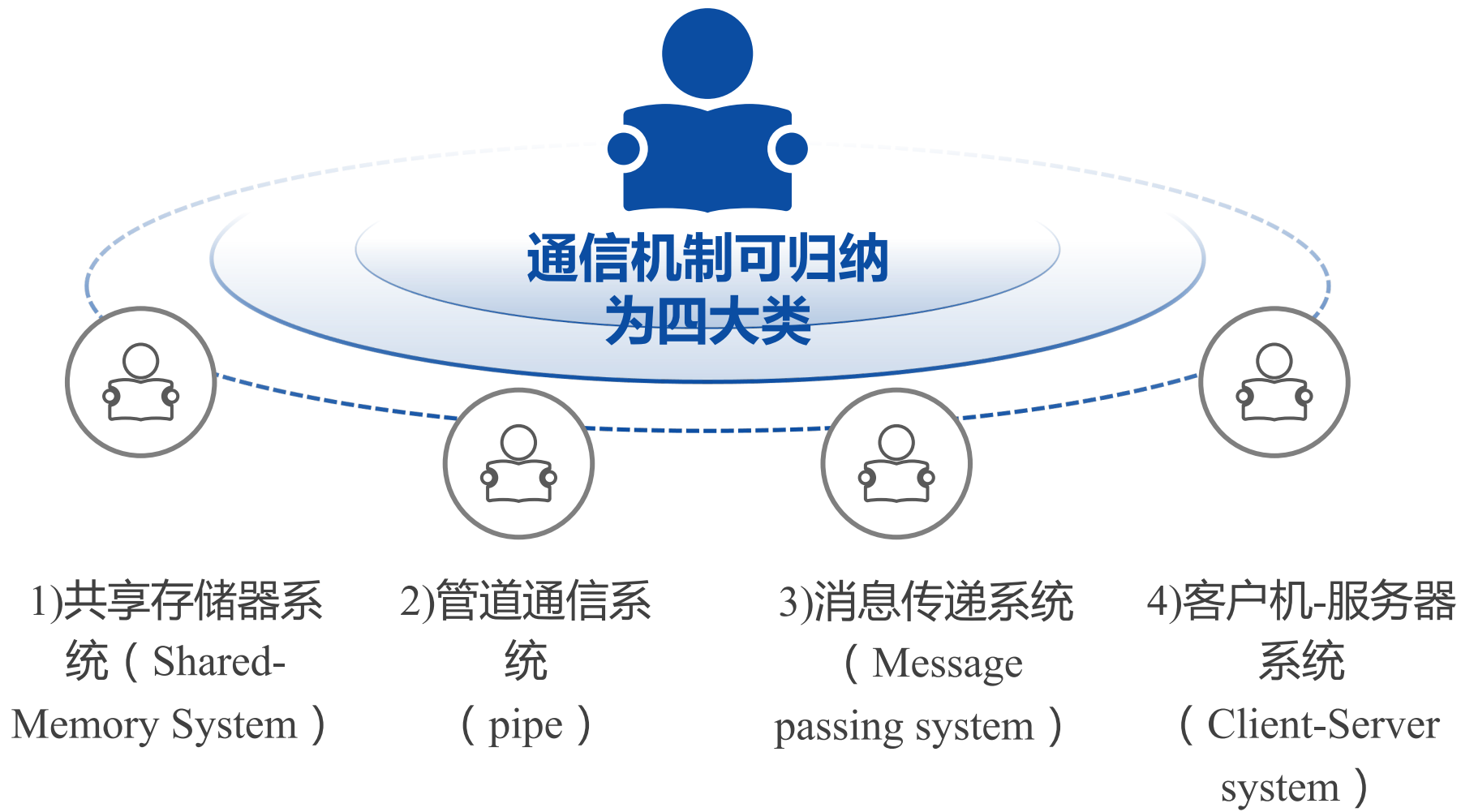


### (当进程之间要传送大量数据时)

- 用户直接利用操作系统提供的一组通信命令, 高效地传送大量数据的通信方式。
- 操作系统隐藏了进程通信的细节, 对用户透明, 减少了通信程序编制上的复杂性。



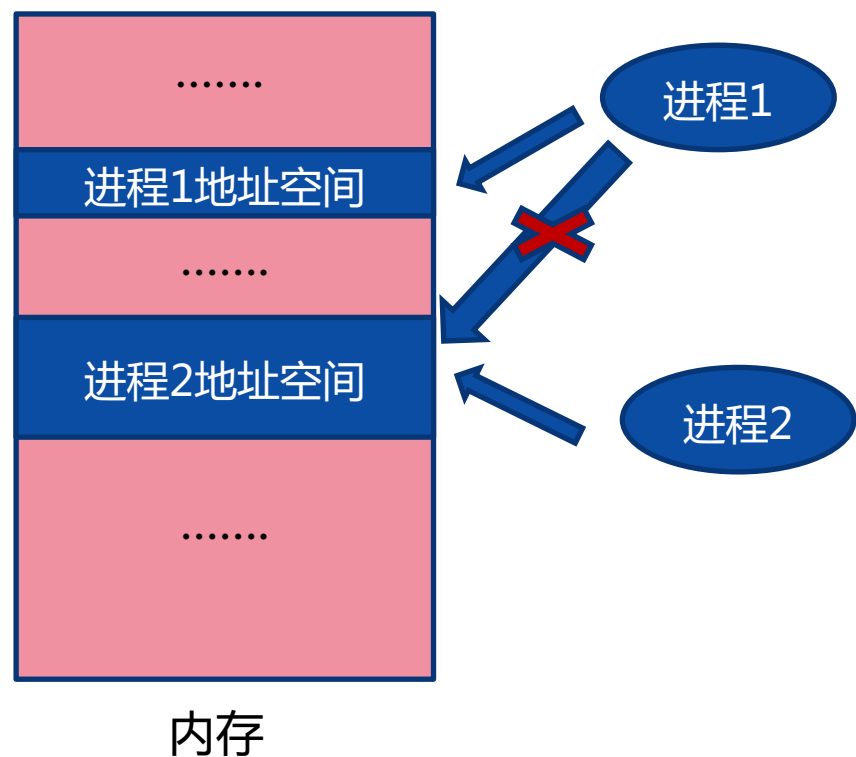
## 02 进程通信的类型



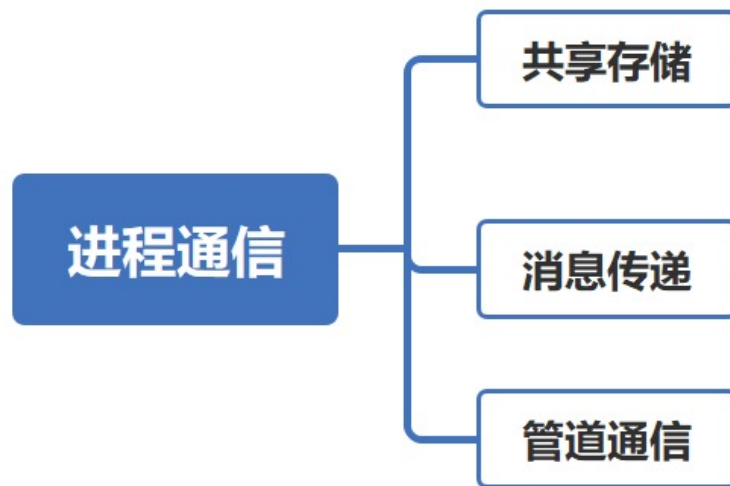
# 共享存储器

顾名思义，进程通信就是指进程之间的信息交换。

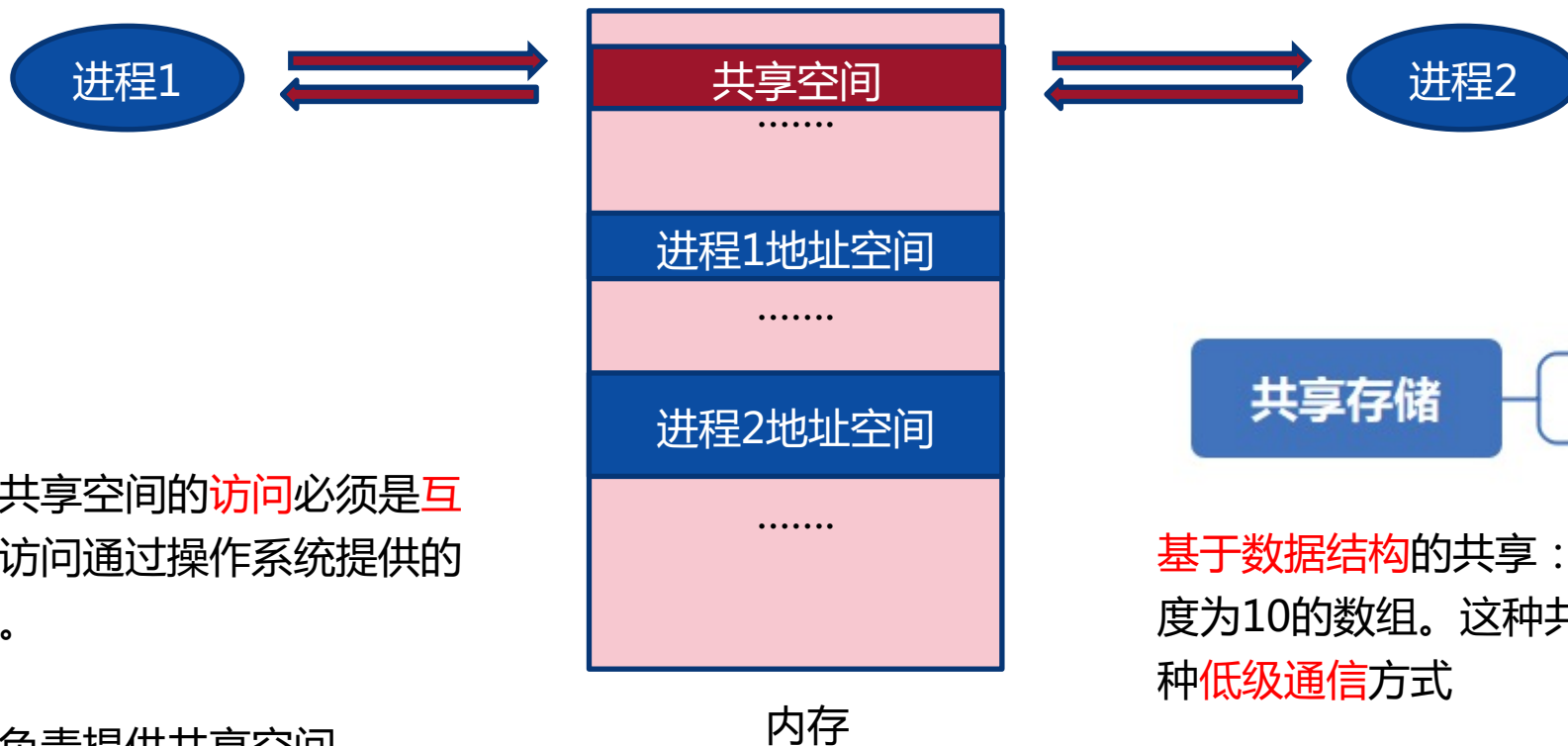
进程是分配系统资源的单位（包括内存地址空间），因此各进程拥有的内存地址空间相互独立。



为了保证安全，一个进程不能直接访问另一个进程的地址空间。但是进程之间的信息交换又是必须实现的。为了保证进程间的安全通信，操作系统提供了一些方法。



# 共享存储器



两个进程对共享空间的访问必须是互斥的（互斥访问通过操作系统提供的工具实现）。

操作系统只负责提供共享空间和同步互斥工具（如P、V操作）

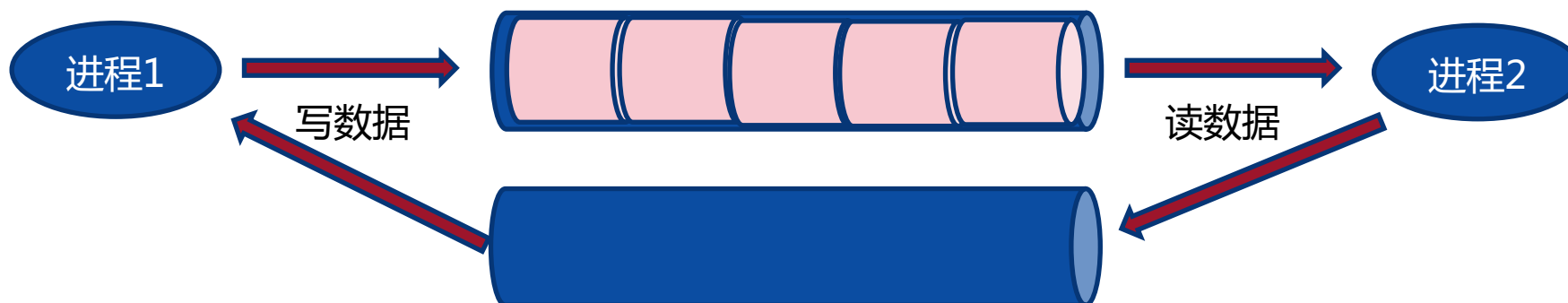


**基于数据结构的共享**：比如共享空间里只能放一个长度为10的数组。这种共享方式速度慢、限制多，是一种**低级通信**方式

**基于存储区的共享**：在内存中画出一块共享存储区，数据的形式、存放位置都由进程控制，而不是操作系统。相比之下，这种共享方式速度更快，是一种**高级通信**方式。

# 进程通信——管道通信

“管道”是指用于连接读写进程的一个共享文件，又名pipe文件。其实就是在内存中开辟一个大小固定的缓冲区



1. 管道只能采用**半双工通信**，某一时间段内只能实现单向的传输。如果要实现**双向同时通信**，则需要设置两个管道。
2. 各进程要**互斥**地访问管道。
3. 数据以字符流的形式写入管道，当**管道写满**时，**写进程**的write()系统调用将被**阻塞**，等待读进程将数据取走。当读进程将数据全部取走后，**管道变空**，此时**读进程**的read()系统调用将被**阻塞**。
4. 如果**没写满**，就不允许读。如果**没读空**，就不允许写。(严格意义上是这样，有些情况会放松)
5. 数据一旦被读出，就从管道中被抛弃，这就意味着**读进程最多只能有一个**，否则可能会有读错数据的情况。

## 2.6 进程通信

### 2.6.1 进程通信的类型

#### 3. 消息传递系统(Message passing system)

在该机制中，进程不必借助任何共享存储区或数据结构，而是以**格式化的消息** (message) 为单位，将通信的数据封装在消息中，并利用操作系统提供的一组通信命令(原语)，在进程间进行消息传递，完成进程间的数据交换。

基于消息传递系统的通信方式属于**高级通信方式**，因其实现方式的不同，可进一步分成两类：

##### (1) 直接通信方式

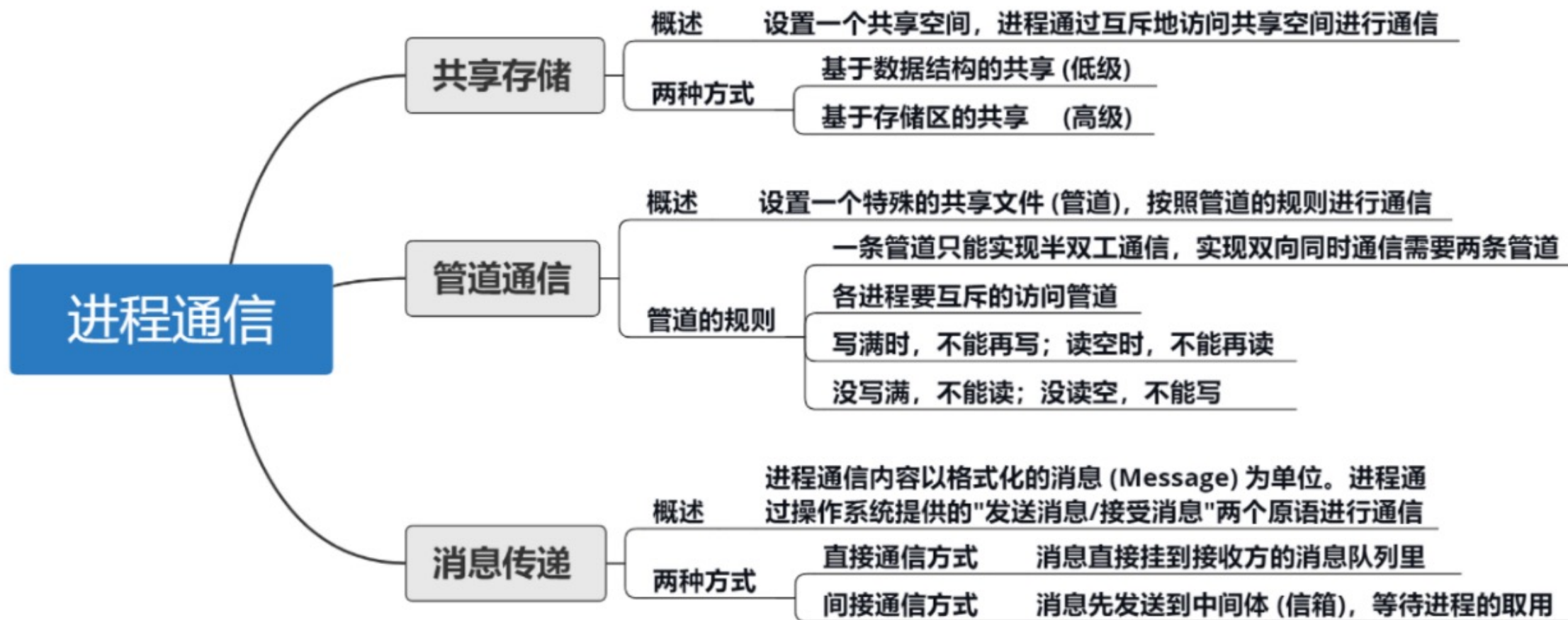
使用发送原语，直接把消息发送给目标进程

##### (2) 间接通信方式

发送和接收进程，通过共享中间实体（邮箱）的方式进行消息的发送和接收

## 02 进程通信的类型

前三种类型较为多见，可以用下图概括：



客户机——服务器的通信机制，其主要的实现方式分为三类：

**套接字(Socket)**、**远程过程调用**和**远程方法调用**。



## 套接字Socket

一个套接字就是一个**通信标识类型的数据结构**：包含通信目的地，通信使用的端口号，通信网络的传输层协议，进程所在的网络地址，以及针对客户或者服务器所提供的不同的系统调用。通常套接字包括两类：

**基于文件型**：通信进程都运行在一台机器的环境中，套接字是基于本地文件系统的支持，一个套接字关联到一个特殊的文件，双方文件基于这个特殊文件进行读写来实现通信

**基于网络型**：采用非对称方法通信，即发送者需要提供接收者的命名。通信双方的进程运行的在不同主机的网络环境下，被分配了一对套接字。

套接字的**优势**在于，它不仅适用于同一台计算机内部的进程通信，也适用于网络环境中不同计算机间的进程通信。

## 远程过程调用和远程方法调用

远程过程调用RPC ( Remote Procedure Call )，是一个通信协议，用于通过网络连接的系统。该协议允许运行一台主机系统上的进程调用另一台主机系统上的进程，而对程序员表现为常规的进程调用，无需为此额外编程。如果涉及的软件采用**面向对象编程**，那么**远程过程调用**亦可称做**远程方法调用**。



## 2.6 进程通信-消息传递系统

### 背景

**消息传递系统 ( Message passing system )**，当前应用最为广泛的一种进程间的通信机制。在该机制中，进程间的数据交换是以格式化的消息 ( message ) 为单位的。在计算机网络中，又把message称为报文。程序员直接利用操作系统提供的一组通信命令 ( 原语 )，不仅能实现大量数据的传递，而且还隐藏了通信的实现细节，使通信过程对用户是透明的，从而大大减化通信程序编制的复杂性，因而获得广泛的应用。

当今最为流行的微内核操作系统中，微内核与服务器之间的通信，无一例外地采用了消息传递机制。它能很好地支持多处理机系统、分布式系统和计算机网络，因此成为这些领域最主要的通信工具。

消息传递的实现方式：**直接通信方式、间接通信方式。**



## 2.6 进程通信

### 2.6.2 消息传递通信的实现方式

#### 1. 直接消息传递系统

在直接消息传递系统中采用直接通信方式，即发送进程利用OS所提供的发送命令(原语)，直接把消息发送给目标进程。

##### 1) 直接通信原语

- (1) 对称寻址方式。双方都需指定地址。
- (2) 非对称寻址方式。接收方无法指定地址。

## 2.6 进程通信

### 2.6.2 消息传递通信的实现方式

#### 1. 直接消息传递系统

##### 2) 消息的格式

在消息传递系统中所传递的消息，必须具有一定的消息格式。在**单机系统环境中**，由于发送进程和接收进程处于同一台机器中，有着相同的环境，所以消息的格式比较简单，可采用**比较短的定长消息格式，以减少对消息的处理和存储开销。**

该方式可用于**办公自动化系统**中，为用户提供快速的便笺式通信。但这种方式对于需要发送较长消息的用户是不方便的。为此，可**采用变长的消息格式，即进程所发送消息的长度是可变的。对于变长消息**，系统无论在处理方面还是存储方面，都可能会付出更多的开销，但其优点在于方便了用户。

## 2.6 进程通信

### 2.6.2 消息传递通信的实现方式

#### 1. 直接消息传递系统

#### 3) 进程的同步方式

在进程之间进行通信时，同样需要有进程同步机制，以使诸进程间能协调通信。不论是发送进程还是接收进程，在完成消息的发送或接收后，都存在两种可能性，即进程或者继续发送(或接收)或者阻塞。

#### 4) 通信链路

为使在发送进程和接收进程之间能进行通信，必须在两者之间建立一条通信链路。有两种方式建立通信链路。第一种方式是：由发送进程在通信之前用显式的“建立连接”命令(原语)请求系统为之建立一条通信链路，在链路使用完后拆除链路。

## 2.6 进程通信

### 2. 间接通信方式-信箱通信

信箱通信属于间接通信方式，即进程之间的通信，需要通过某种中间实体（如共享数据结构等）来完成。

#### 1) 信箱的结构

信箱定义为一种数据结构，通常从逻辑上分为两部分。

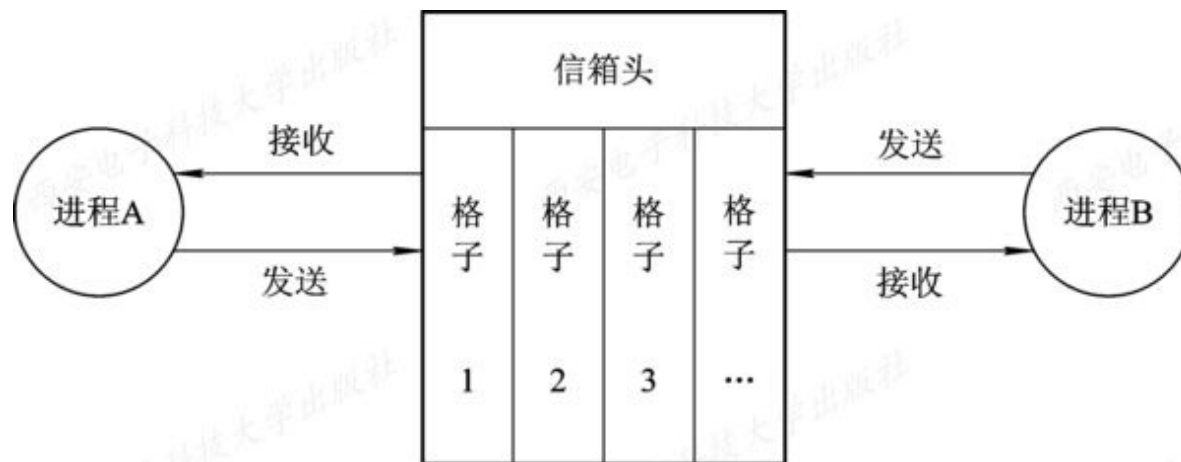
**信息头**：存放有关信箱的**描述信息**，如信箱标识符，信箱的拥有者，信箱的口令，信箱的空格数。

**信箱体**：由若干个可以**存放消息(或者消息头)**的信箱格子组成。

消息传递方式上，最简单的情况是**单向**传递。消息的传递也可以是**双向**的。下图为书本上的双向链路的通信方式。



## 2.6 进程通信



### 2) 信箱的通信原语

- (1) **邮箱的创建和撤消**。进程可利用信箱创建原语来建立一个新信箱。创建者进程应给出信箱名字、信箱属性（公用、私用或共享）。对于共享信箱还应给出共享者的名字。当进程不需要读信箱时，可用信箱撤消原语将之撤消。
- (2) **消息的发送和接收**。当进程之间要利用信箱进行通信时，必须使用共享信箱，并利用通信原语通信。

Send(mailbox , message)将一个消息发送到指定邮箱。

Receive(mailbox , message)从指定邮箱中接收一个消息。

## 2.6 进程通信

信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。由此可分为三类：

### 私用邮箱

用户进程可为自己建立一个新信箱，并作为该进程的一部分，**信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。**这种私用信箱可采用单向通信链路的信箱来实现。当拥有该信箱的进程结束时，信箱也随之消失。

### 公用邮箱

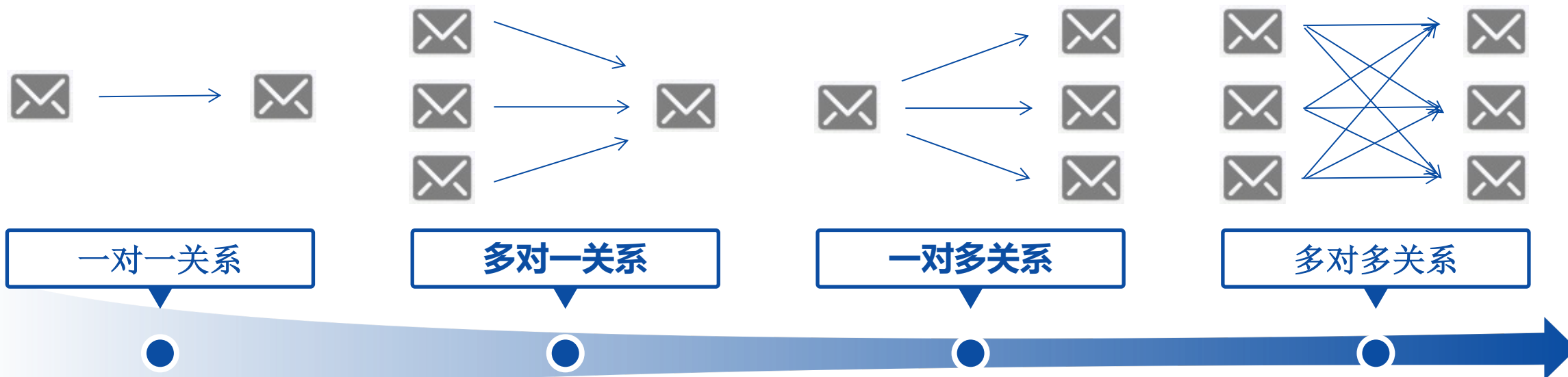
**由操作系统创建，并提供给系统中的所有核准进程使用。**核准进程即可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。显然公用信箱应采用双向通信链路的信箱来实现。通常公用信箱在系统运行期间始终存在。

### 共享邮箱

**由某进程创建，在创建时或创建后指明它是可共享的，同时须指出共享进程（用户）的名字。**信箱的拥有者和共享者都有权从信箱中取走发送给自己的消息。

## 2.6 进程通信

利用邮箱通信，发送进程和接收进程之间存在以下**四种关系**：



请可为发送进程和接收进程建立一条两者专用的通信链路，使两者之间的交互不受其他进程的干扰。

允许提供服务的进程与多个用户进程之间进行交互，也称为客户/服务器交互 (client/server interaction)。

允许一个发送进程与多个接收进程进行交互，使发送进程可用广播方式向接收者 (多个) 发送消息。

允许建立一个公用信箱，让多个进程都能向信箱中投递消息，也可从信箱中取走属于自己的消息。

## 2.6 进程通信

### 2.6.3 直接消息传递系统实例

消息缓冲队列通信机制首先由美国的Hansan提出，并在RC 4000系统上实现，后来被广泛应用于本地进程之间的通信中。在这种通信机制中，发送进程利用Send原语将消息直接发送给接收进程；接收进程则利用Receive原语接收消息。

#### 1. 消息缓冲队列通信机制中的数据结构

- (1) 消息缓冲区。
- (2) PCB中有关通信的数据项。



## 2.6 进程通信

### 1.消息缓冲队列通信机制中的数据结构

(1)消息缓冲区。在消息缓冲队列通信方式中，主要利用的数据结构是消息缓冲区。它可描述如下：

```
typedef struct message_buffer {  
    int sender;           发送者进程标识符  
    int size;             消息长度  
    char *text;           消息正文  
    struct message_buffer *next; 指向下一个消息缓冲区的指针  
}
```

(2)PCB中有关通信的数据项。

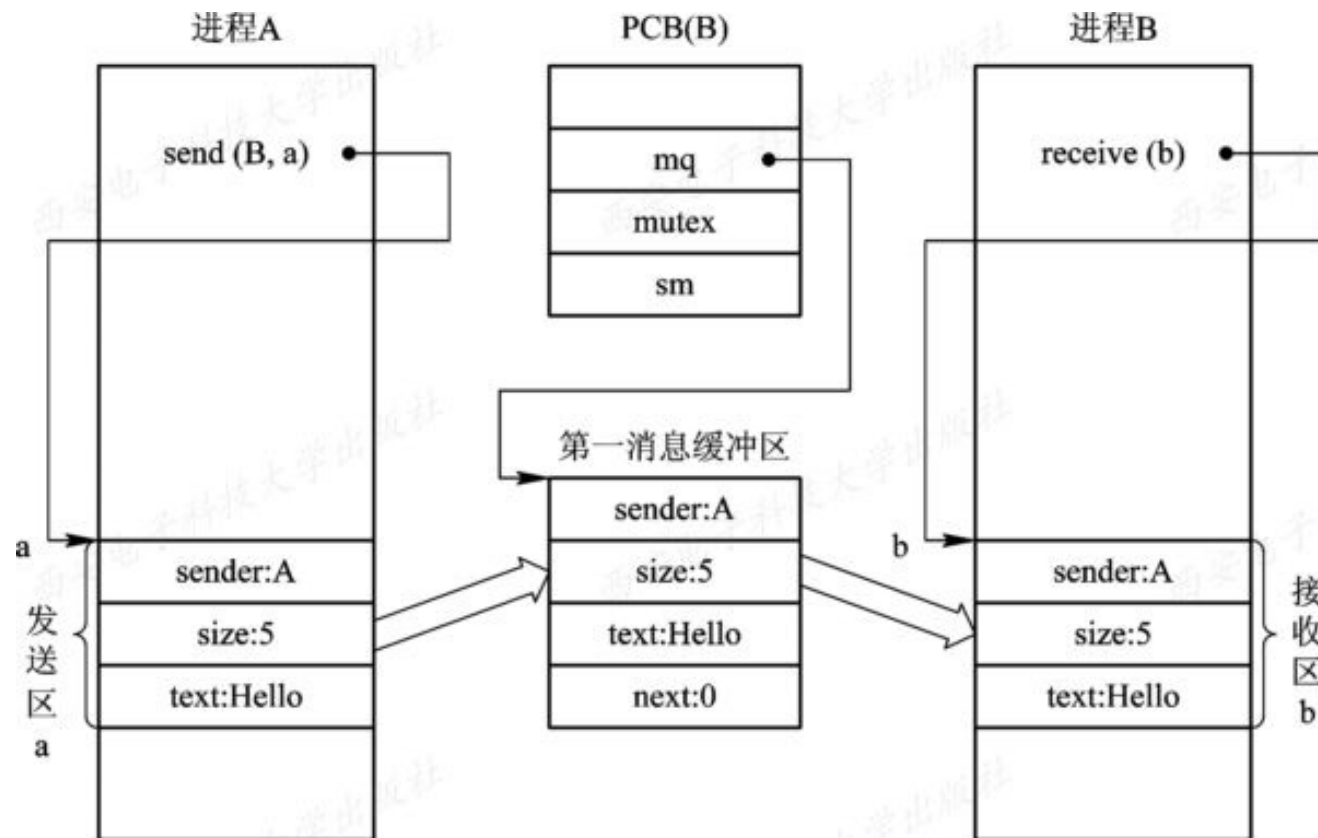
```
typedef struct processcontrol_block {  
    ...  
    struct message_buffer *mq;  消息队列队首指针  
    semaphore mutex;           消息队列互斥信号量  
    semaphore sm;              消息队列资源信号量  
}
```

## 2.6 进程通信

### 2.6.3 直接消息传递系统实例

#### 2. 发送原语

发送进程在利用**发送原语发送消息**之前，应先在在自己的内存空间设置一发送区a，如图所示，把待发送的消息正文、发送进程标识符、消息长度等信息填入其中，然后调用发送原语，把消息发送给目标(接收)进程。发送原语首先根据发送区a中所设置的消息长度a.size来申请一缓冲区i，接着，把发送区a中的信息复制到缓冲区i中。为了能将i挂在接收进程的消息队列mq上，应先获得接收进程的内部标识符j，然后将i挂在j.mq上。由于该队列属于临界资源，故在执行insert操作的前后都要执行wait和signal操作。



## 2.6 进程通信

发送原语可描述如下：

<pre>void send(receiver, a) {</pre>	
<pre>    getbuf(a.size, i);</pre>	receiver为接收进程标识符，a为发送区首； 根据a.size申请缓冲区；
<pre>    i.sender = a.sender;</pre>	
<pre>    i.size = a.size;</pre>	
<pre>    copy(i.text, a.text);</pre>	将发送区a中的信息复制到消息缓冲区i中；
<pre>    i.next = 0;</pre>	
<pre>    getid(PCBset, receiver.j);</pre>	获得接收进程内部的标识符；
<pre>    wait(j.mutex);</pre>	
<pre>    insert(&amp;j.mq, i);</pre>	将消息缓冲区插入消息队列；
<pre>    signal(j.mutex);</pre>	
<pre>    signal(j.sm);</pre>	先释放消息才能接收消息
<pre>}</pre>	



## 2.6 进程通信

### 2.6.3 直接消息传递系统实例

#### 3. 接收原语

接收进程调用接收原语receive(b)，从自己的消息缓冲队列mq中摘下第一个消息缓冲区i，并将其中的数据复制到以b为首址的指定消息接收区内。

## 2.6 进程通信

接收原语可描述如下：

```
void receive(b) {  
    j = internal name;    j为接收进程内部的标识符;  
    wait(j.sm);           接收已经释放的消息  
    wait(j.mutex);  
    remove(j.mq, i);      将消息队列中第一个消息移出;  
    signal(j.mutex);  
    b.sender = i.sender;  
    b.size = i.size;  
    copy(b.text, i.text);  将消息缓冲区i中的信息复制到接收区b;  
    releasebuf(i);        释放消息缓冲区;  
}
```



## 2.6 进程通信

总结：

### 2.6.1. 进程通信的方式

共享存储器系统

管道通信系统

消息传递系统

客户机-服务器系统

### 2.6.2. 消息传递通信实现的方式

直接消息传递系统

间接消息传递系统（邮箱通信）

原语：send , receive



## 第二章 进程的描述与控制



2.1 前趋图和程序执行

2.2 进程的描述

2.3 进程控制

2.4 进程同步

2.5 经典进程的同步问题

2.6 进程通信

2.7 线程(Threads)的基本概念

2.8 线程的实现

习题

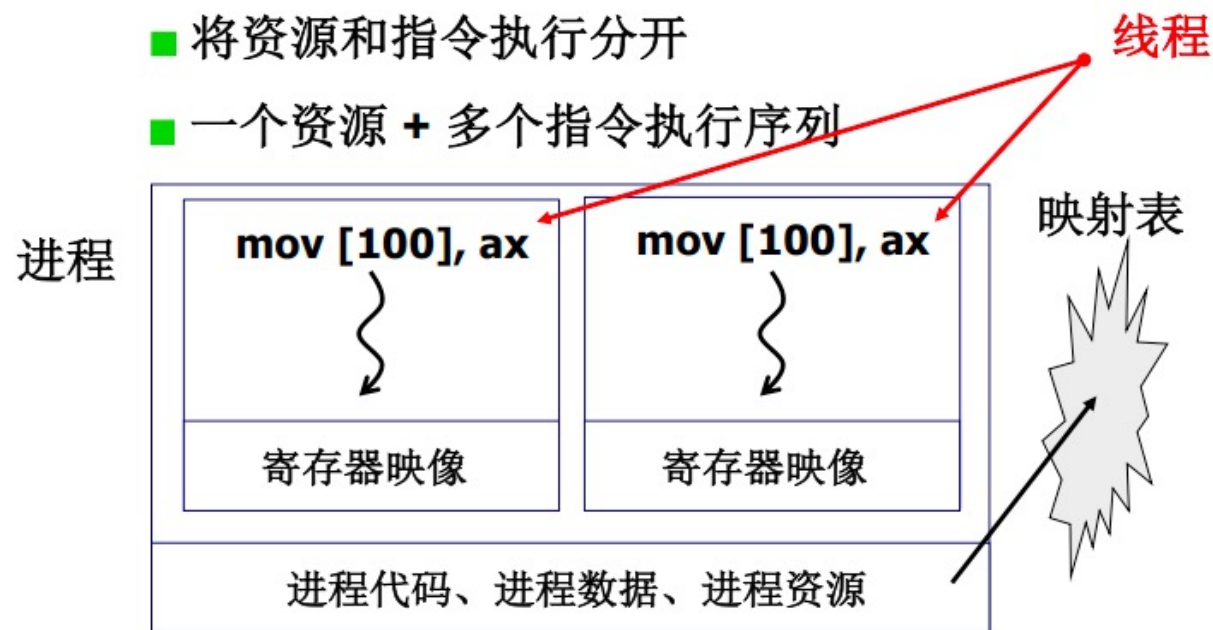
# 2.7 线程(Threads)的基本概念

## 2.7.1 线程的引入

如果说，在OS中引入进程的目的是为了使多个程序能并发执行，以提高资源利用率和系统吞吐量，那么，在操作系统中再引入线程，则是为了减少程序在并发执行时所付出的时空开销，使OS具有更好的并发性。

■ 将资源和指令执行分开

■ 一个资源 + 多个指令执行序列



可否只切换指令不切换资源？

即保留并发的特性

减少上下文切换的开销：例如不切换映射表



## 2.7 线程(Threads)的基本概念

### 2.7.1 线程的引入

#### 1. 进程的两个基本属性

首先让我们来回顾进程的两个基本属性：

① 进程是一个可拥有资源的独立单位，一个进程要能独立运行，它必须拥有一定的资源，包括用于存放程序正文、数据的磁盘和内存地址空间，以及它在运行时所需要的I/O设备、已打开的文件、信号量等；

② 进程同时又是一个可独立调度和分派的基本单位，一个进程要能独立运行，它还必须是一个可独立调度和分派的基本单位。每个进程在系统中有唯一的PCB，系统可根据其PCB感知进程的存在，也可以根据其PCB中的信息，对进程进行调度，还可将断点信息保存在其PCB中。反之，再利用进程PCB中的信息来恢复进程运行的现场。

正是由于进程有这两个基本属性，才使进程成为一个能独立运行的基本单位，从而也就构成了进程并发执行的基础。

## 2.7 线程(Threads)的基本概念

### 2.7.1 线程的引入

#### 2. 程序并发执行所需付出的时空开销

为使程序能并发执行，系统必须进行以下的一系列操作：

- (1) 创建进程，系统在创建一个进程时，必须为它分配其所必需的、除处理机以外的所有资源，如内存空间、I/O设备，以及建立相应的PCB；
- (2) 撤消进程，系统在撤消进程时，又必须先对其所占有的资源执行回收操作，然后再撤销PCB；
- (3) 进程切换，对进程进行上下文切换时，需要保留当前进程的CPU环境，设置新选中进程的CPU环境，因而须花费不少的处理机时间。

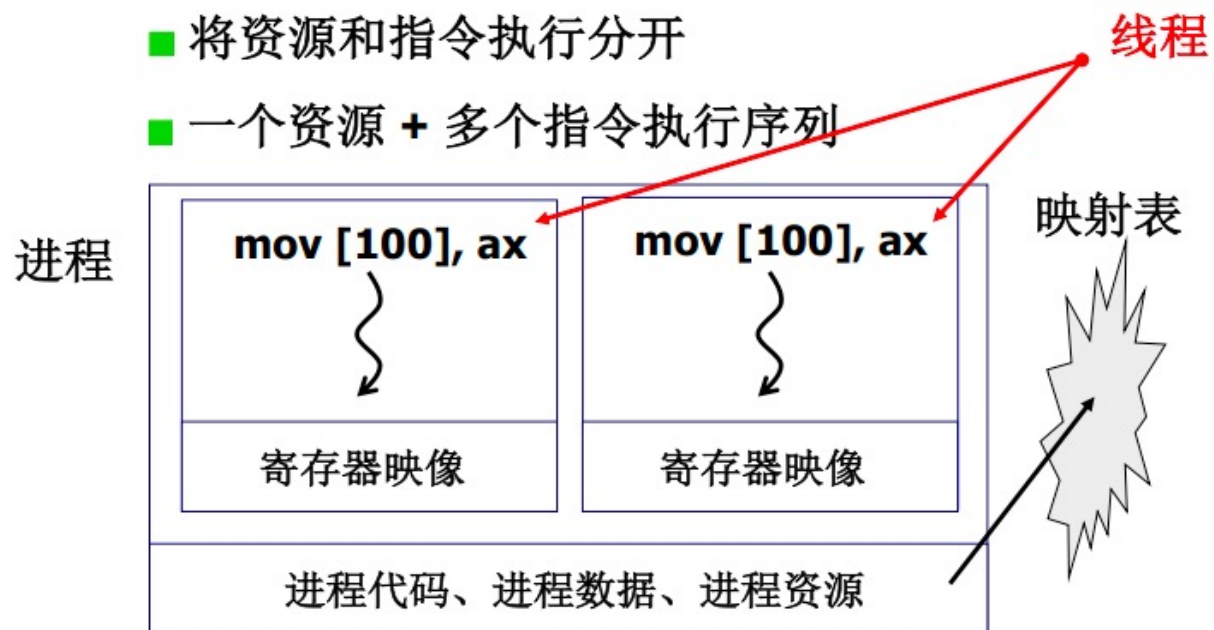
## 2.7 线程(Threads)的基本概念

### 2.7.1 线程的引入

#### 3. 线程——作为调度和分派的基本单位

如何能使多个程序更好地并发执行，同时又尽量减少系统的开销，已成为近年来设计操作系统时所追求的重要目标。有不少研究操作系统的学者们想到，要设法将进程的上述两个属性分开，由OS分开处理，亦即并不把作为调度和分派的基本单位也同时作为拥有资源的单位，以做到“轻装上阵”；而对于拥有资源的基本单位，又不对之施以频繁的切换。正是在这种思想的指导下，形成了线程的概念。

## 2.7 线程(Threads)的基本概念

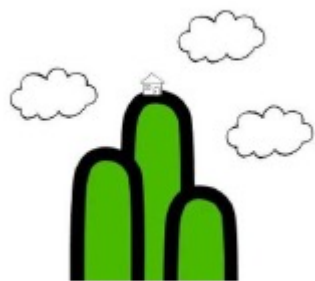


- 线程：保留了并发的优点，避免了进程切换的代价
- 实质就是映射表不变而 PC 指针变

# 什么是线程，为什么要引入线程？

还没引入进程之前，系统中各个程序只能串行执行。

故事发生在很久以前……



引入线程之后



进程是程序的一次执行。但这些功能显然不可能由一个程序顺序处理就能实现的

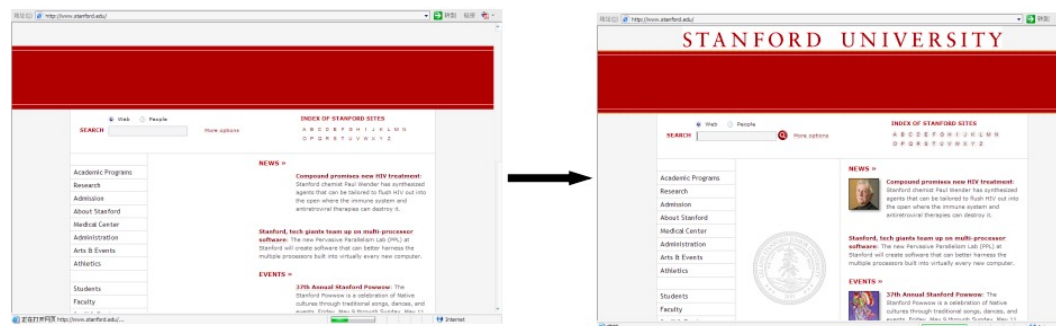
But...Think QQ 可以do what?

## 2.7 线程(Threads)的基本概念

### ■ 线程同步

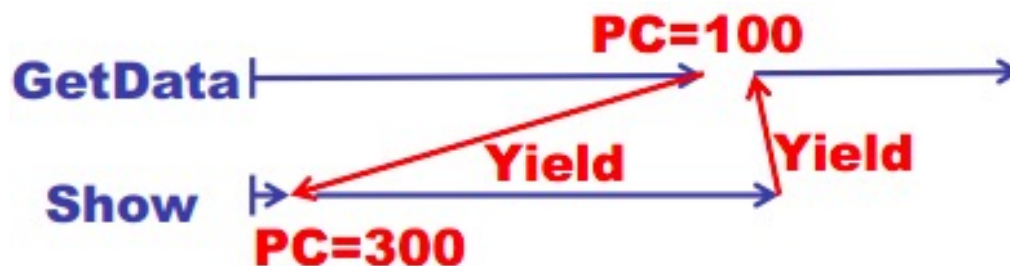
### ■ 一个网页浏览器

- 一个线程用来从服务器接收数据
- 一个线程用来显示文本
- 一个线程用来处理图片（如解压缩）
- 一个线程用来显示图片



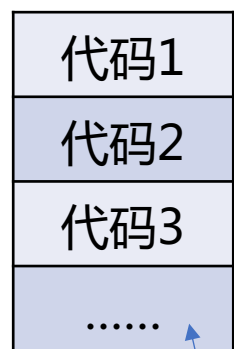
### ■ 这些线程要共享资源么？

- 接收数据放在存储区，显示时要读
- 共享同一个屏幕

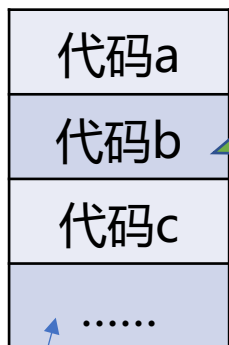


# 什么是线程，为什么要引入线程？

有的进程可能需要“同时”做很多事，而传统的进程只能串行地执行一系列程序。为此，引入了“线程”，来增加并发度。



进程1

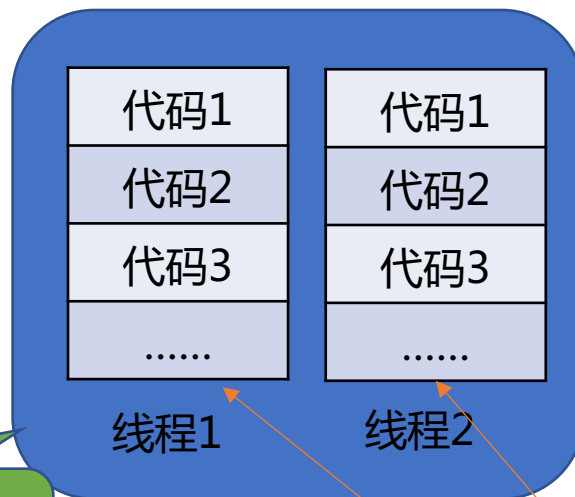


进程2



传统的进程是  
程序执行流的  
最小单位

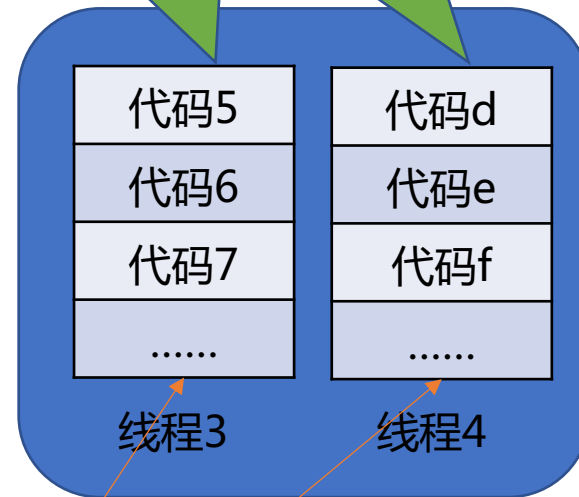
引入线程后，线程成为了  
程序执行流的最小单位



进程

QQ视频聊  
天处理程序

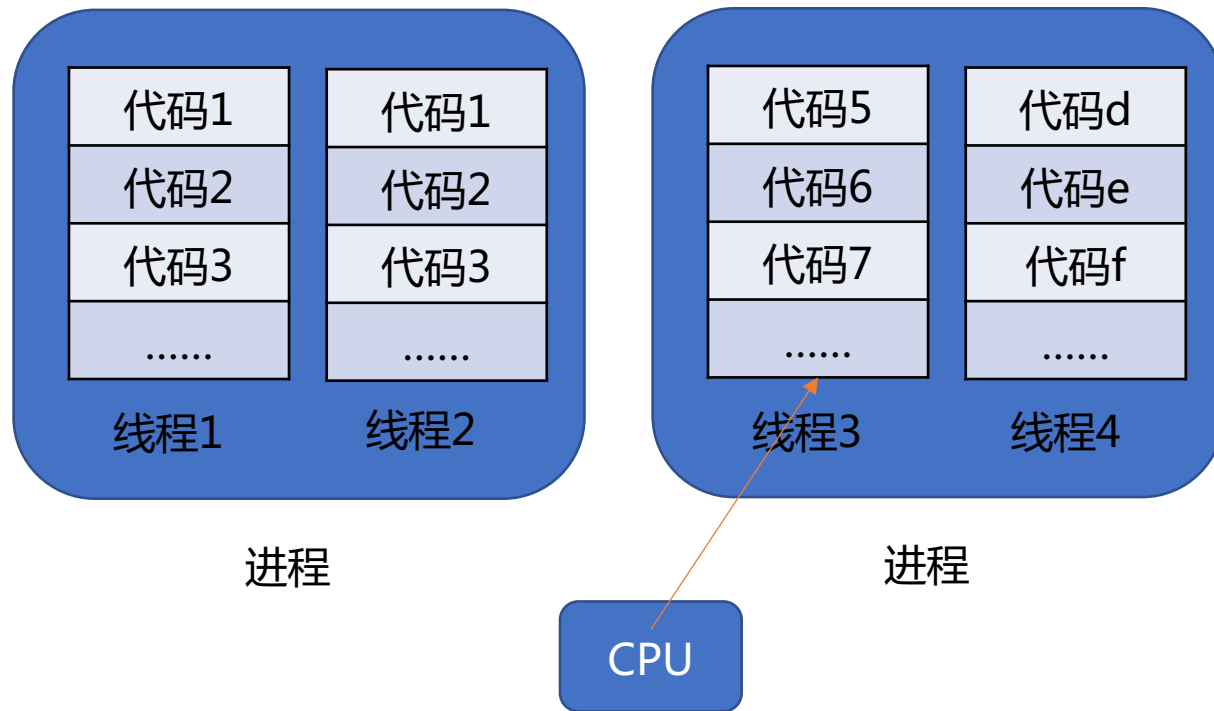
QQ传送文件处  
理程序



进程



# 什么是线程，为什么要引入线程？



可以把线程理解为“轻量级进程”。

线程是一个基本的CPU执行单元，也是程序执行流的最小单位。引入线程之后，不仅是进程之间可以并发，进程内的各线程之间也可以并发，从而进一步提升了系统的并发度，使得一个进程内也可以并发处理各种任务（如QQ视频、文字聊天、传文件）

引入线程后，进程只作为除CPU之外的系统资源的分配单元（如打印机、内存地址空间等都是分配给进程的）。

线程则作为处理机的分配单元。



## 2.7 线程(Threads)的基本概念

### 2.7.2 线程与进程的比较

#### 1. 调度的基本单位

进程 → 线程

#### 2. 并发性

进程并发 → 进程并发+线程并发

#### 3. 拥有资源

TCB：程序计数器、局部变量、少数状态参数、返回地址

多个线程可访问进程的资源，属于同一进程的线程具有相同的地址空间

## 2.7 线程(Threads)的基本概念

### 2.7.2 线程与进程的比较

#### 4. 独立性

共享进程的地址空间和资源，比进程的独立性要低

#### 5. 系统开销

大 → 小

#### 6. 支持多处理机系统

同一个进程的线程可在多个处理机上运行

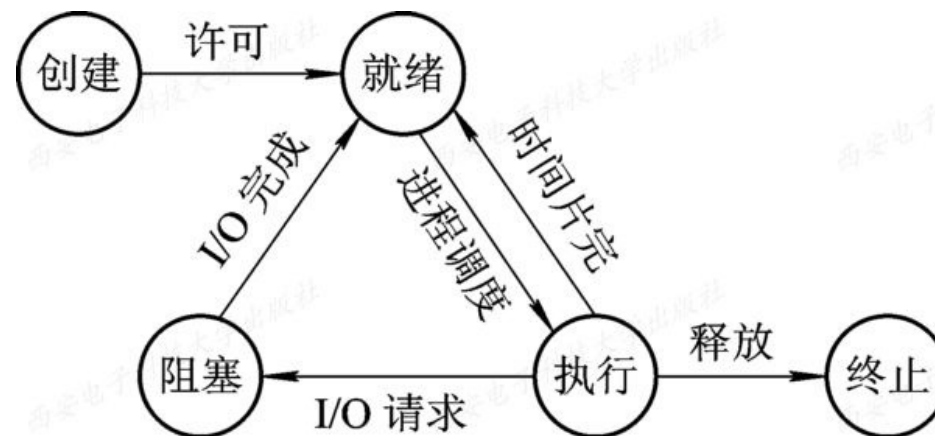
## 2.7 线程(Threads)的基本概念

### 2.7.3 线程的状态和线程控制块

#### 1. 线程运行的三个状态

与传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。相应地，线程在运行时也具有下述三种基本状态：

- (1) 执行状态，表示线程已获得处理机而正在运行；
- (2) 就绪状态，指线程已具备了各种执行条件，只须再获得CPU便可立即执行；
- (3) 阻塞状态，指线程在执行中因某事件受阻而处于暂停状态，例如，当一个线程执行从键盘读入数据的系统调用时，该线程就被阻塞。



## 2.7 线程(Threads)的基本概念

### 2.7.3 线程的状态和线程控制块

#### 2. 线程控制块TCB

如同每个进程有一个进程控制块一样，系统也为每个线程配置了一个线程控制块TCB，将所有用于控制和管理线程的信息记录在线程控制块中。

- ① 线程标识
- ② 寄存器信息：程序计数器PC、状态寄存器、通用寄存器
- ③ 线程运行状态
- ④ 优先级
- ⑤ 线程专有存储区
- ⑥ 屏蔽信号
- ⑦ 堆栈指针

例如初始地址、内存空间等

## 2.7 线程(Threads)的基本概念

### 2.7.3 线程的状态和线程控制块

#### 3. 多线程OS中的进程属性

通常多线程OS中的进程都包含了多个线程，并为它们提供资源。OS支持在一个进程中的多个线程能并发执行，但此时的进程就不再作为一个执行的实体。多线程OS中的进程有以下属性：

- (1) 进程是一个可拥有资源的基本单位。
- (2) 多个线程可并发执行。
- (3) 在多线程OS中，把线程作为独立运行（调度）的基本单位。

对进程状态有关的操作也对线程起作用。

## 2.7 线程(Threads)的基本概念

### 2.7.2 线程与进程的比较

	调度的基本单位	并发性	拥有资源	独立性	系统开销	多处理机
无线程的OS	进程	✓	以进程为单位	进程高	大	不支持
有线程的OS	线程	进程+线程	线程仅拥有必不可少的部分资源	线程低	小	支持



## 第二章 进程的描述与控制



2.1 前趋图和程序执行

2.2 进程的描述

2.3 进程控制

2.4 进程同步

2.5 经典进程的同步问题

2.6 进程通信

2.7 线程(Threads)的基本概念

2.8 线程的实现

习题



## 2.8 线程的实现



### 2.8.1 线程的实现方式

线程已在许多系统中实现，但各系统的实现方式并不完全相同。在有的系统中，特别是一些数据库管理系统，如infomix所实现的是**用户级线程**；而另一些系统(如Macintosh和OS/2操作系统)所实现的是**内核支持线程**；还有一些系统如Solaris操作系统，则同时实现了这两种类型线程的**组合方式**。



## 2.8 线程的实现

### 2.8.1 线程的实现方式

#### 1. 内核支持线程KST(Kernel Supported Threads)

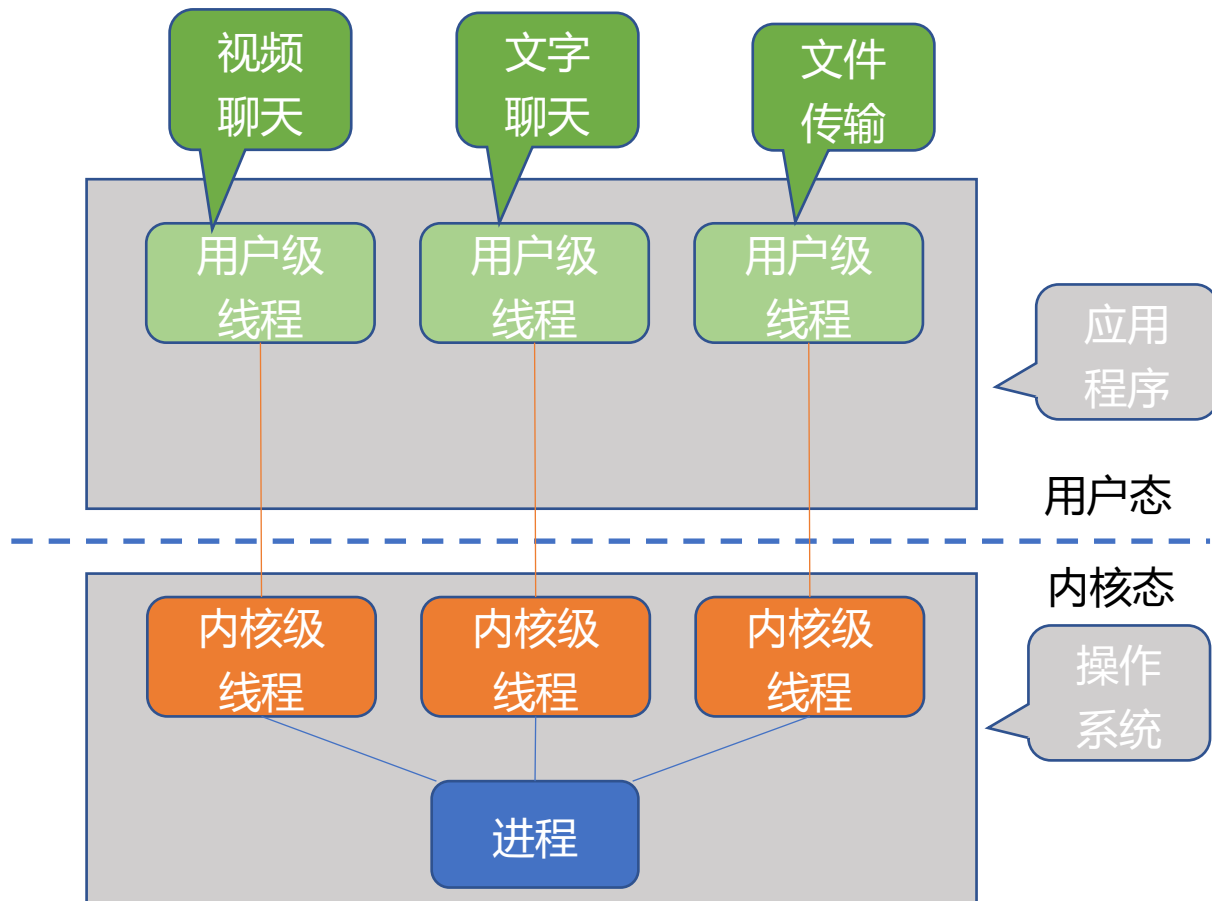
在OS中的所有进程，无论是系统进程还是用户进程，都是在操作系统内核的支持下运行的，是与内核紧密相关的。而内核支持线程KST同样也是在内核的支持下运行的，它们的创建、阻塞、撤消和切换等，也都是在内核空间实现的。

为了对内核线程进行控制和管理，在内核空间也为每一个内核线程设置了一个线程控制块，内核根据该控制块而感知某线程的存在，并对其加以控制。当前大多数OS都支持内核支持线程。

# 线程的实现方式

内核级线程 ( Kernel-Level Thread, KLT, 又称 “内核支持的线程” )

由操作系统支持的线程



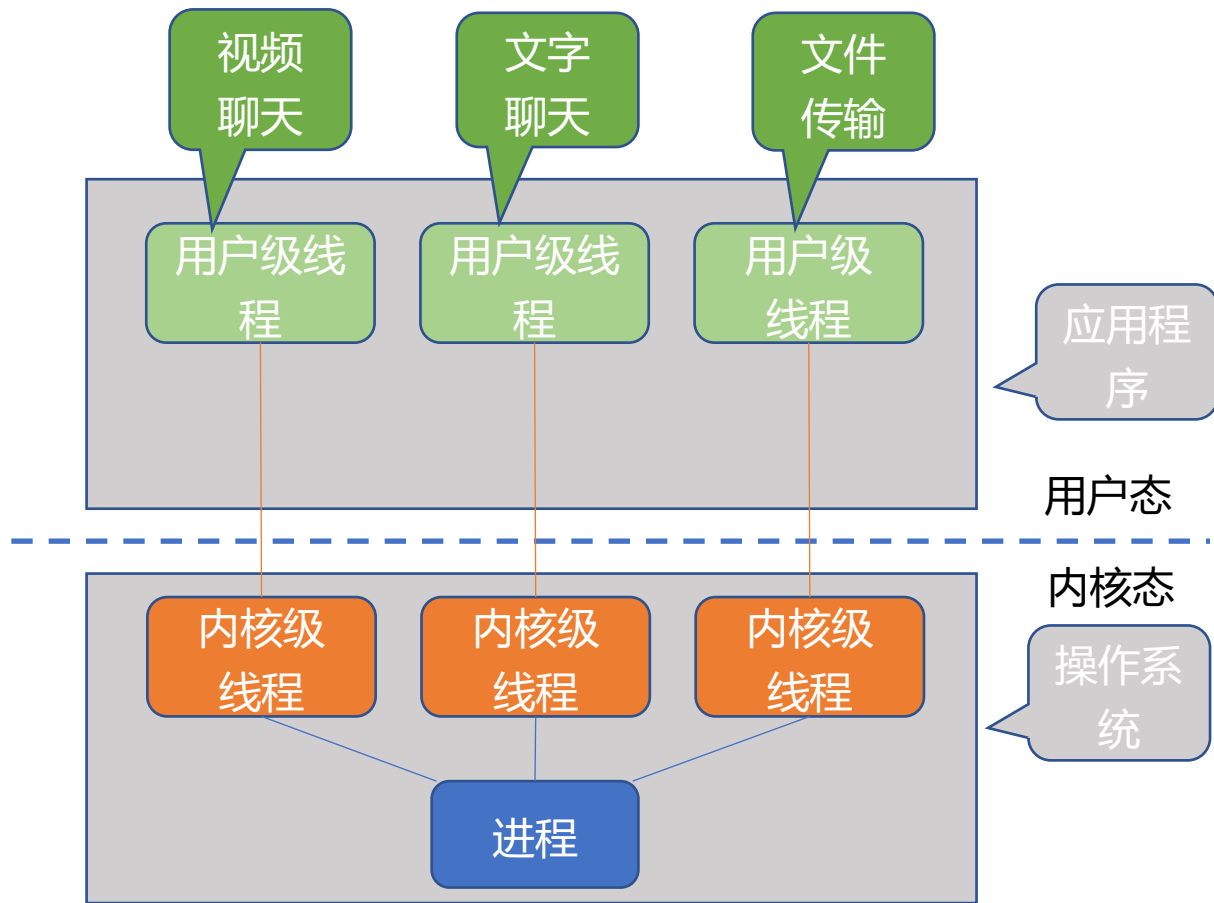
大多数现代操作系统都实现了内核级线程，如 Windows、Linux

1. 线程的管理工作由谁来完成？
2. 线程切换是否需要CPU变态？
3. 操作系统是否能意识到内核级线程的存在？
4. 这种线程的实现方式有什么优点和缺点？

# 线程的实现方式

内核级线程 ( Kernel-Level Thread, KLT, 又称 “内核支持的线程” )

由操作系统支持的线程



1. 内核级线程的管理工作由操作系统内核完成。
2. 线程调度、切换等工作都由内核负责，因此内核级线程的切换必然需要在核心态下才能完成。
3. 操作系统会为每个内核级线程建立相应的TCB ( Thread Control Block , 线程控制块 ) ，通过TCB对线程进行管理。  
“内核级线程”就是“从操作系统内核视角看能看到的线程”
4. 优缺点  
优点：当一个线程被阻塞后，别的线程还可以继续执行，并发能力强。多线程可在多核处理机上并行执行。  
缺点：一个用户进程会占用多个内核级线程，线程切换由操作系统内核完成，需要切换到核心态，因此线程管理的成本高，开销大。

## 2.8 线程的实现

### 2.8.1 线程的实现方式

#### 1. 内核支持线程KST(Kernel Supported Threads)

这种线程实现方式主要有四个主要优点：

- (1) 在多处理器系统中，内核能够同时调度同一进程中的多个线程并行执行；
- (2) 如果进程中的一个线程被阻塞了，内核可以调度该进程中的其它线程占有处理器运行，也可以运行其它进程中的线程；
- (3) 内核支持线程具有很小的数据结构和堆栈，线程的切换比较快，切换开销小；
- (4) 内核本身也可以采用多线程技术，可以提高系统的执行速度和效率。

## 2.8 线程的实现

### 2.8.1 线程的实现方式

#### 2. 用户级线程ULT(User Level Threads)

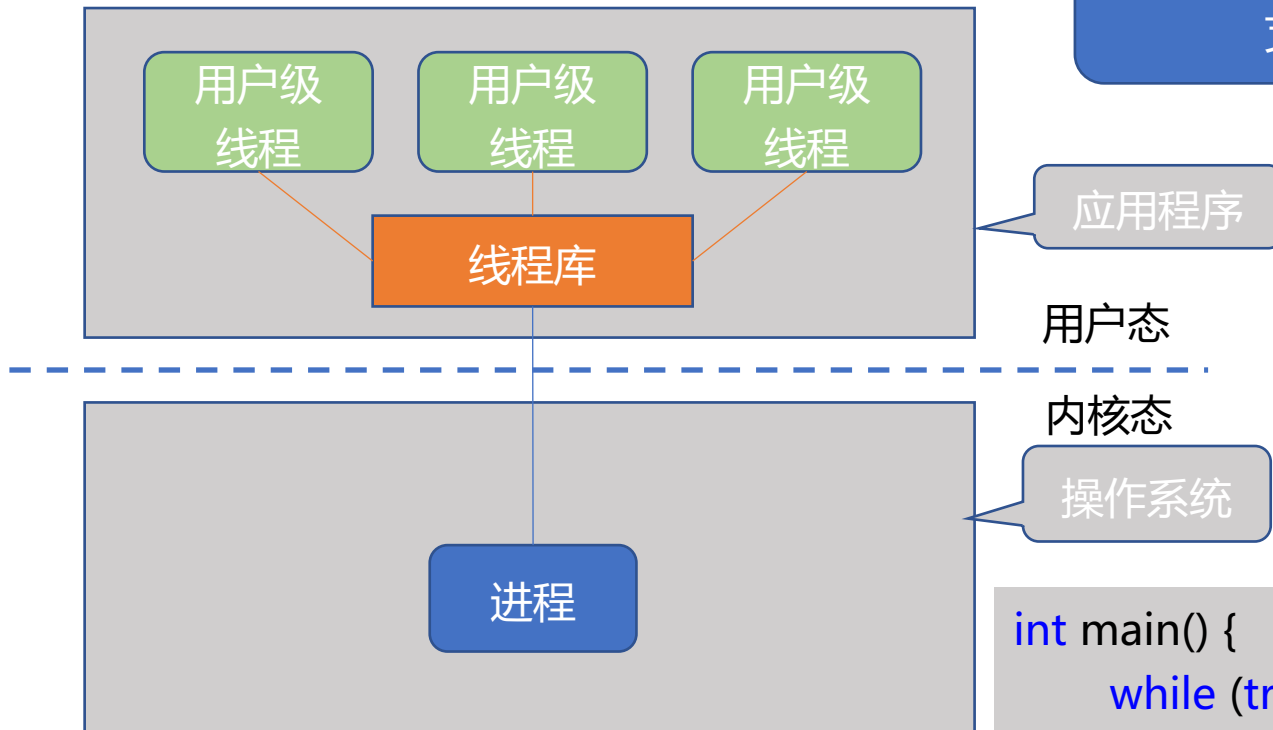
用户级线程是在用户空间中实现的。对线程的创建、撤消、同步与通信等功能，都无需内核的支持，即用户级线程是与内核无关的。

在一个系统中的用户级线程的数目可以达到数百个至数千个。

由于这些线程的任务控制块都是设置在用户空间，而线程所执行的操作也无需内核的帮助，因而内核完全不知道用户级线程的存在。

# 线程的实现方式

用户级线程 ( User-Level Thread, ULT )



历史背景：早期的操作系统（如：早期Unix）只支持进程，不支持线程。当时的“线程”是由线程库实现的



```
int main() {  
    while (true) {  
        处理视频聊天的代码;  
    }  
}
```

进程1

```
int main() {  
    while (true) {  
        处理文字聊天的代码;  
    }  
}
```

进程2

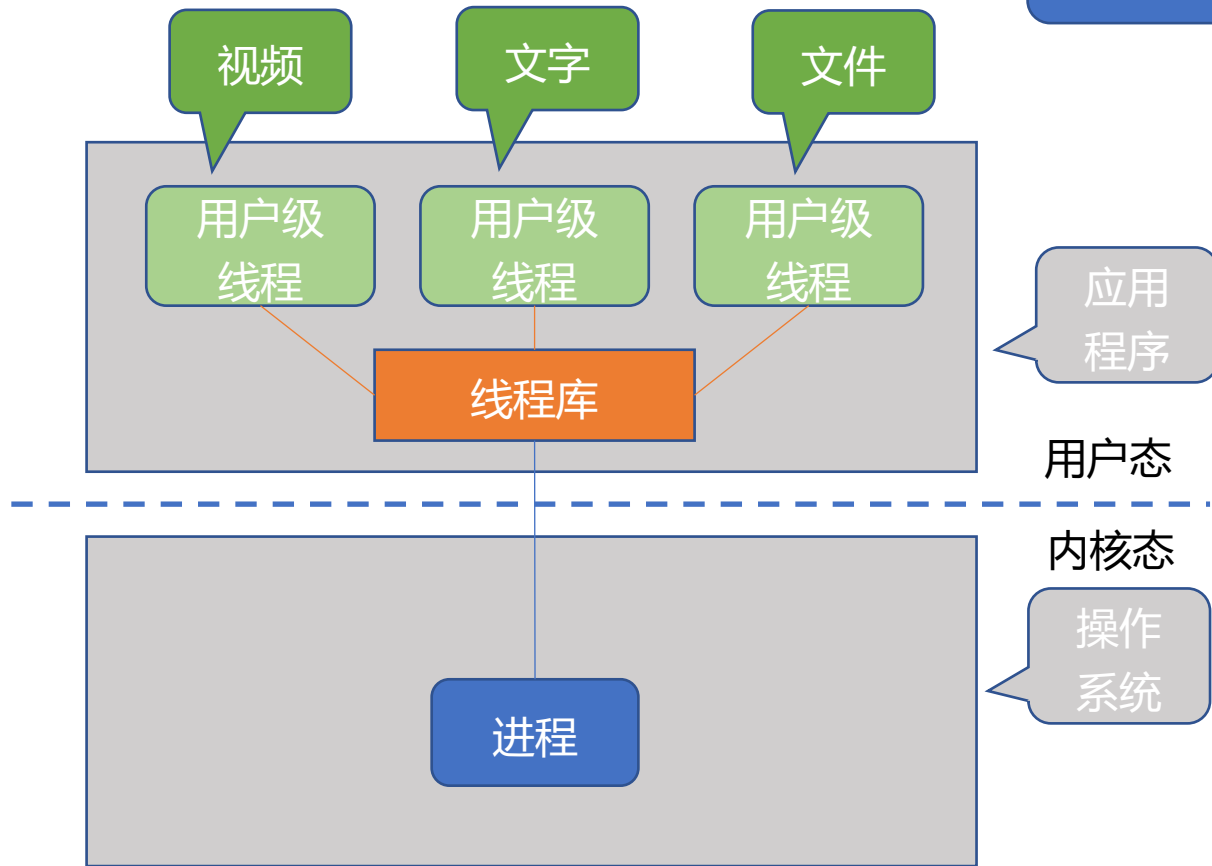
```
int main() {  
    while (true) {  
        处理文件传输的代码;  
    }  
}
```

进程3

# 线程的实现方式

用户级线程 ( User-Level Thread, ULT )

历史背景：早期的操作系统（如：早期Unix）只支持进程，不支持线程。当时的“线程”是由线程库实现的



```
int main() {  
    while (true) {  
        if (i == 0) { 处理视频聊天的代码; }  
        if (i == 1) { 处理文字聊天的代码; }  
        if (i == 2) { 处理文件传输的代码; }  
        i = (i + 1) % 3; //i的值为0,1,2,0,1,2...  
    }  
}
```

QQ进程

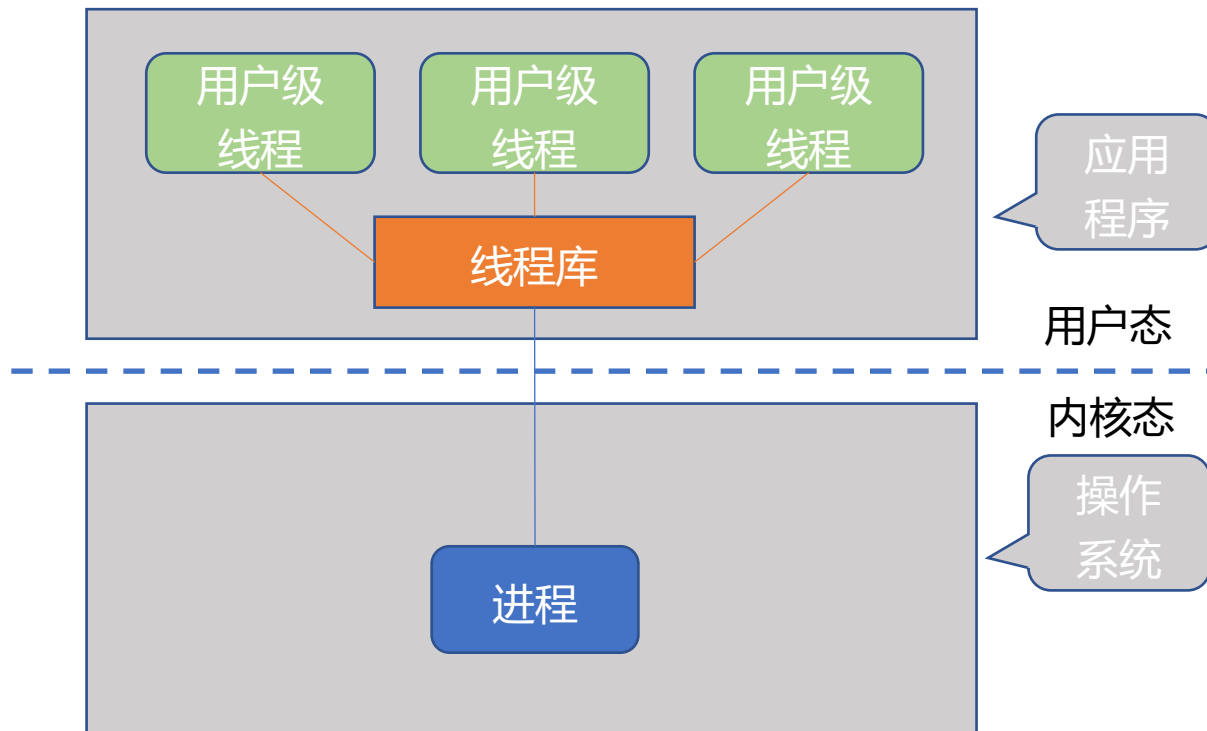
从代码的角度看，线程其实就是一段代码逻辑。上述三段代码逻辑上可以看作三个“线程”。while 循环就是一个最弱智的“线程库”，线程库完成了对线程的管理工作（如调度）。

# 线程的实现方式



历史背景：早期的操作系统（如：早期Unix）只支持进程，不支持线程。当时的“线程”是由线程库实现的

用户级线程（User-Level Thread, ULT）



很多编程语言提供了强大的线程库，可以实现线程的创建、销毁、调度等功能。

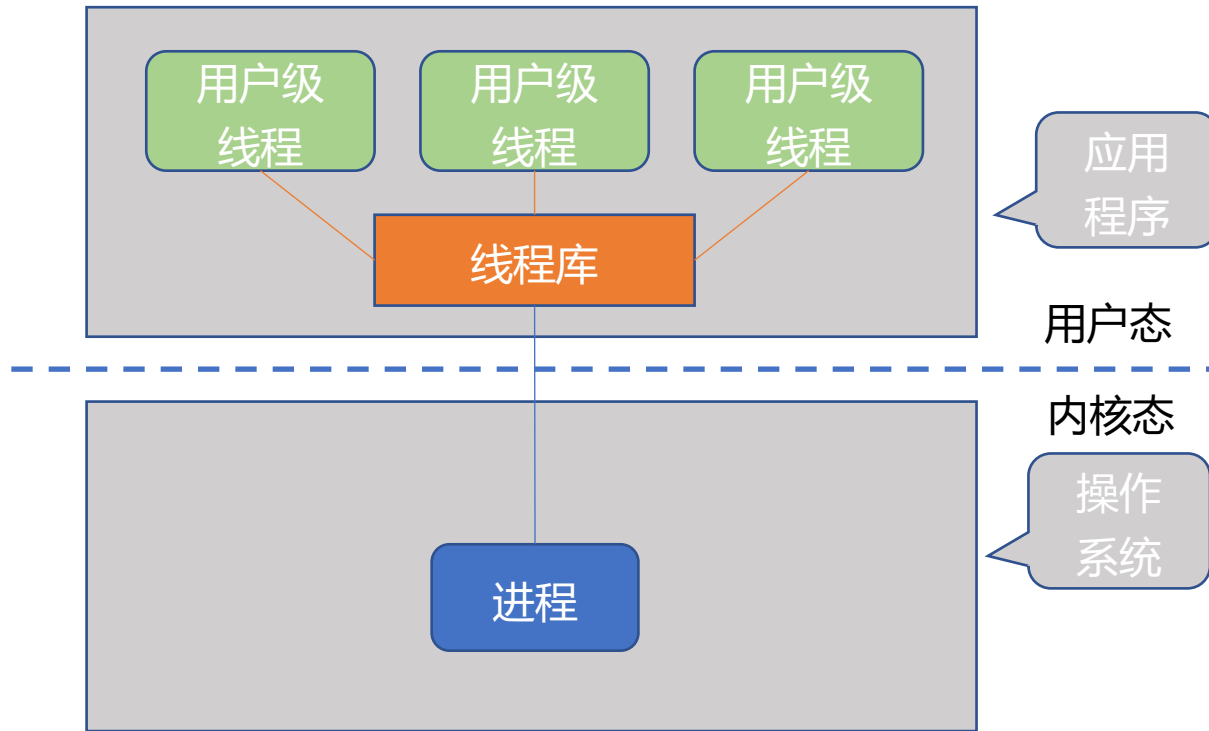
1. 线程的管理工作由谁来完成？
2. 线程切换是否需要CPU变态？
3. 操作系统是否能意识到用户级线程的存在？
4. 这种线程的实现方式有什么优点和缺点？



# 线程的实现方式



用户级线程 ( User-Level Thread, ULT )



1. 用户级线程由应用程序通过线程库实现，所有的线程管理工作都由应用程序负责（包括线程切换）
2. 用户级线程中，线程切换可以在用户态下即可完成，无需操作系统干预。
3. 在用户看来，是有多个线程。但是在操作系统内核看来，并意识不到线程的存在。“用户级线程”就是“从用户视角看能看到的线程”
4. 优缺点  
    优点：用户级线程的切换在用户空间即可完成，不需要切换到核心态，线程管理的系统开销小，效率高  
    缺点：当一个用户级线程被阻塞后，整个进程都会被阻塞，并发度不高。多个线程不可在多核处理机上并行运行。

## 2.8 线程的实现

### 2.8.1 线程的实现方式

#### 2. 用户级线程ULT(User Level Threads)

使用用户级线程方式有许多优点：

- (1) 线程切换不需要转换到内核空间。
- (2) 调度算法可以是进程专用的。
- (3) 用户级线程的实现与OS平台无关，因为对于线程管理的代码是属于用户程序的一部分，所有的应用程序都可以对之进行共享。

## 2.8 线程的实现

### 2.8.1 线程的实现方式

#### 2. 用户级线程ULT(User Level Threads)

而用户级线程方式的主要缺点则在于：

(1) 系统调用的阻塞问题。在基于进程机制的OS中，大多数系统调用将使进程阻塞，因此，当线程执行一个系统调用时，**不仅该线程被阻塞，而且，进程内的所有线程会被阻塞。而在内核支持线程方式中，则进程中的其它线程仍然可以运行。**

(2) 在单纯的用户级线程实现方式中，多线程应用不能利用多处理机进行多重处理的优点，**内核每次分配给一个进程的仅有一个CPU**，因此，进程中仅有一个线程能执行，在该线程放弃CPU之前，其它线程只能等待。

## 2.8 线程的实现

### 2.8.1 线程的实现方式

#### 3. 组合方式

有些OS把用户级线程和内核支持线程两种方式进行组合，提供了组合方式ULT/KST 线程。在组合方式线程系统中，内核支持多个内核支持线程的建立、调度和管理，同时，也允许用户应用程序建立、调度和管理用户级线程。

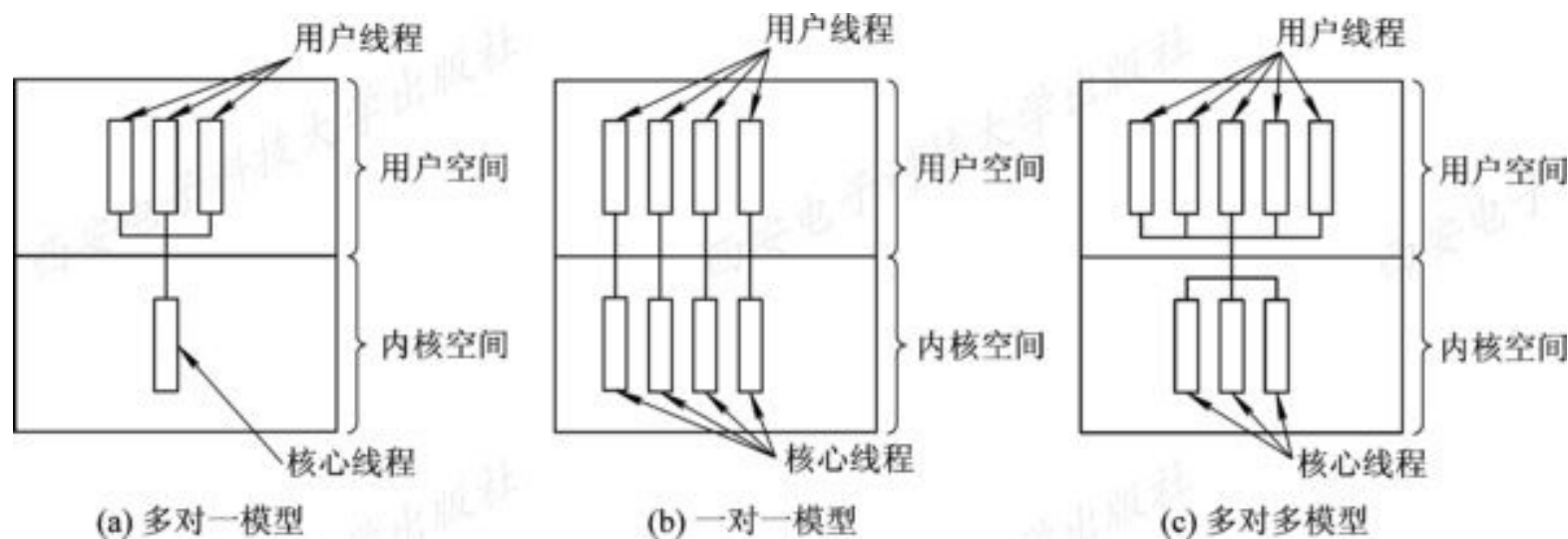
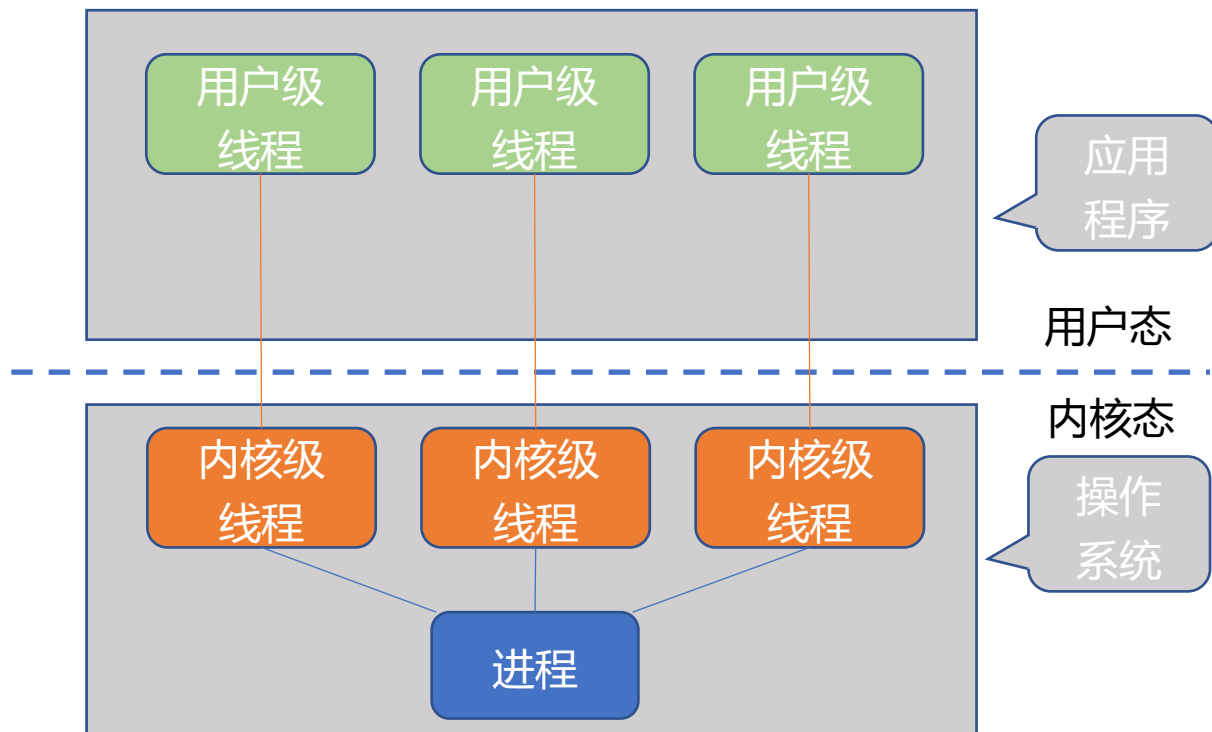


图2-18 多线程模型

# 多线程模型

在支持内核级线程的系统中，根据用户级线程和内核级线程的映射关系，可以划分为几种多线程模型



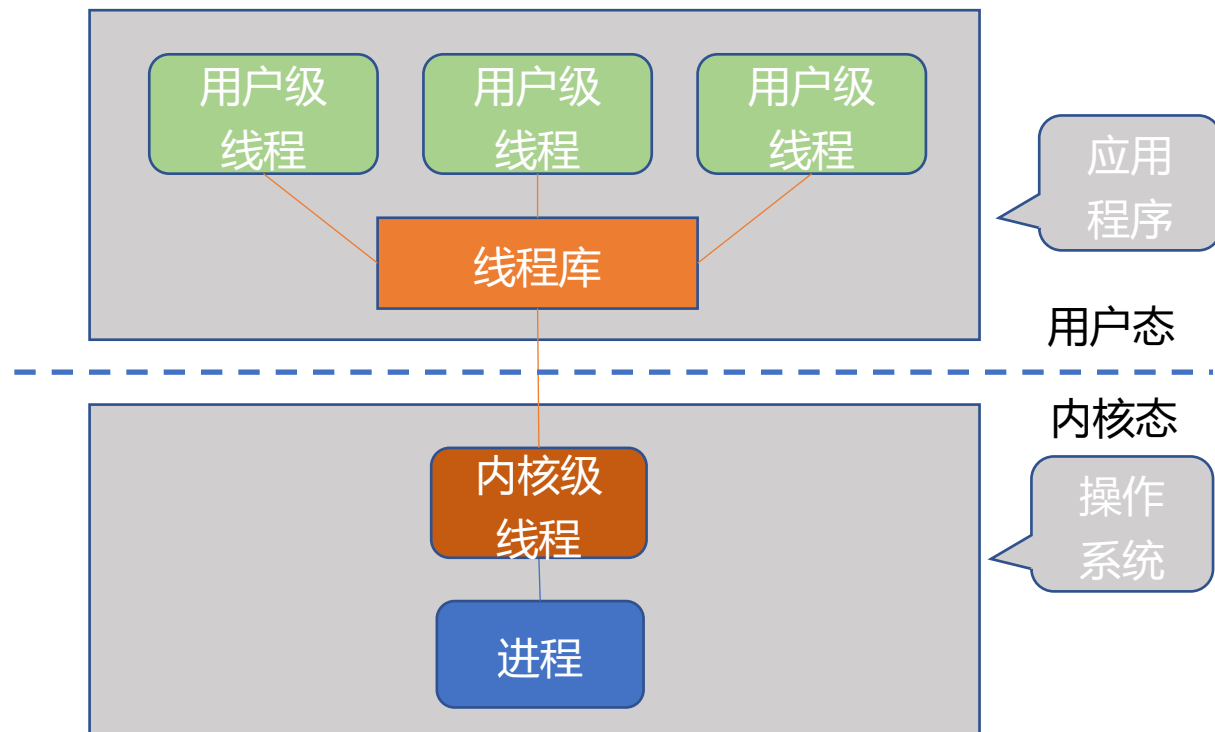
**一对一模型：**一个用户级线程映射到一个内核级线程。每个用户进程有与用户级线程同数量的内核级线程。

优点：当一个线程被阻塞后，别的线程还可以继续执行，并发能力强。多线程可在多核处理机上并行执行。

缺点：一个用户进程会占用多个内核级线程，线程切换由操作系统内核完成，需要切换到核心态，因此线程管理的成本高，开销大。

# 多线程模型

在支持内核级线程的系统中，根据用户级线程和内核级线程的映射关系，可以划分为几种多线程模型



**多对一模型**：多个用户级线程映射到一个内核级线程。且一个进程只被分配一个内核级线程。

优点：用户级线程的切换在用户空间即可完成，不需要切换到核心态，线程管理的系统开销小，效率高

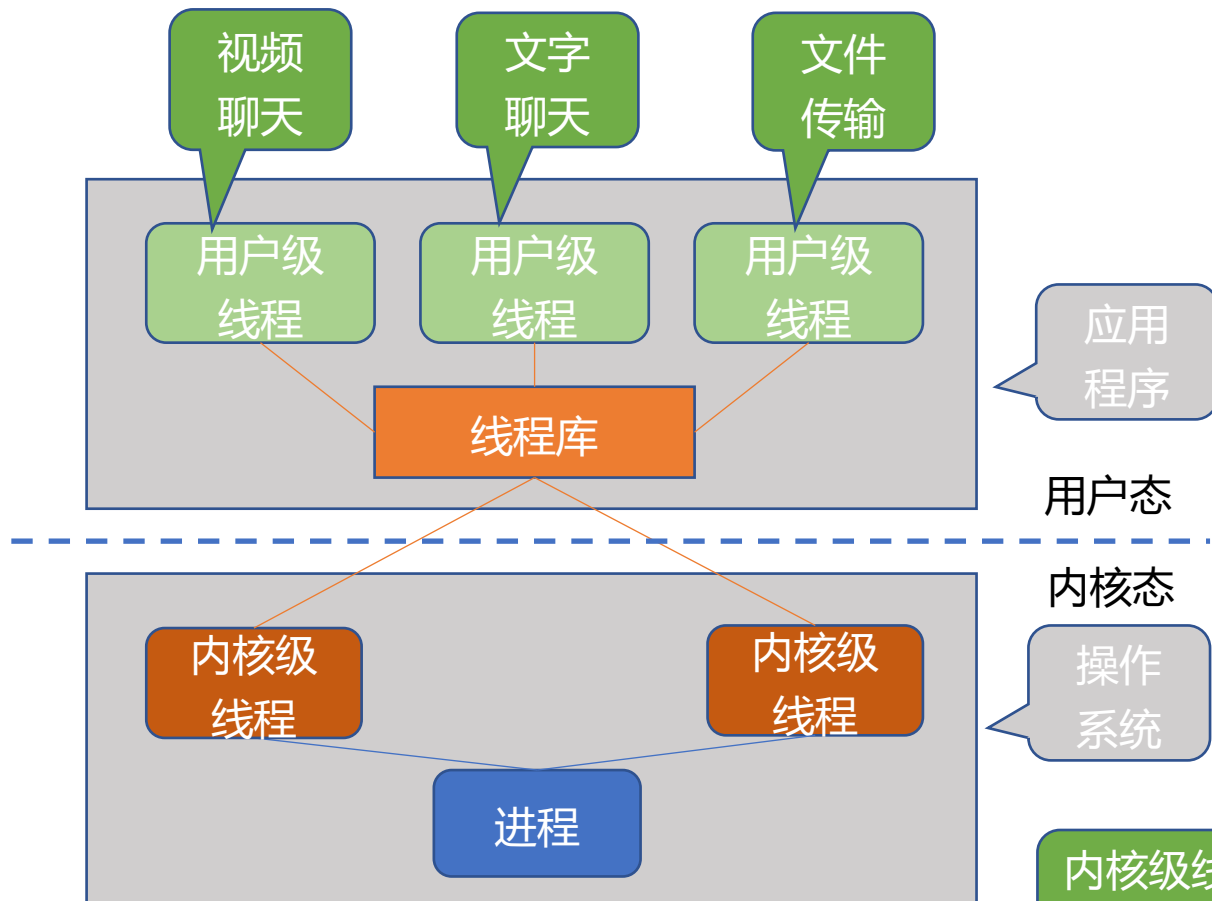
缺点：当一个用户级线程被阻塞后，整个进程都会被阻塞，并发度不高。多个线程不可在多核处理机上并行运行

**重点重点重点：**

操作系统只“看得见”内核级线程，因此只有**内核级线程才是处理机分配的单位**。

# 多线程模型

在支持内核级线程的系统中，根据用户级线程和内核级线程的映射关系，可以划分为几种多线程模型



**多对多模型**：n 用户级线程映射到m 个内核级线程 ( $n \geq m$ )。每个用户进程对应m 个内核级线程。

克服了对一模型并发度不高的缺点（一个阻塞全体阻塞），又克服了一对一模型中一个用户进程占用太多内核级线程，开销太大的缺点。

可以这么理解：

用户级线程是“代码逻辑”的载体  
内核级线程是“运行机会”的载体

**内核级线程才是处理机分配的单位。**例如：多核CPU环境下，左边这个进程最多能被分配两个核。

一段“代码逻辑”只有获得了“运行机会”才能被CPU执行

内核级线程中可以运行任意一个有映射关系的用户级线程代码，只有两个内核级线程中正在运行的代码逻辑都阻塞时，这个进程才会阻塞

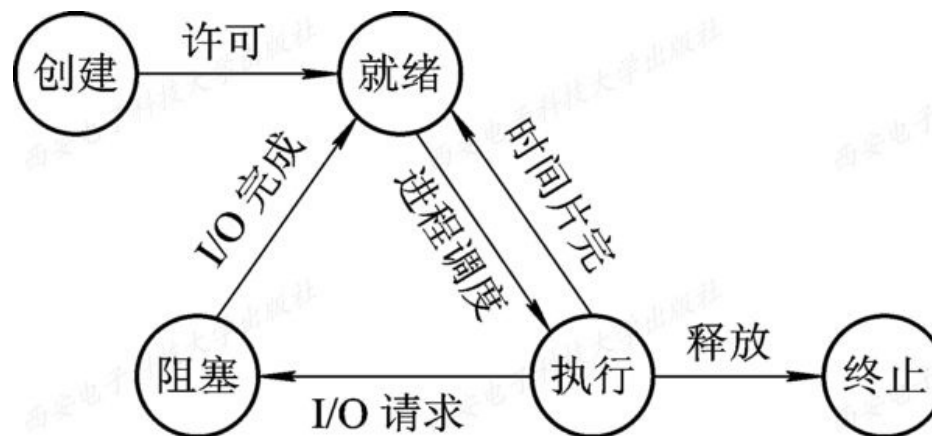
## 2.8 线程的实现

### 2.8.3 线程的创建和终止

线程跟进程一样，由创建而产生、由调度而执行、由终止而消亡。

#### 1. 线程的创建

应用程序在启动时，通常仅有一个线程在执行，人们把线程称为“初始化线程”，它的主要功能是用于创建新线程。在创建新线程时，需要利用一个线程创建函数(或系统调用)，并提供相应的参数，如指向线程主程序的入口指针、堆栈的大小，以及用于调度的优先级等。在线程的创建函数执行完后，将返回一个线程标识符供以后使用。





## 2.8 线程的实现

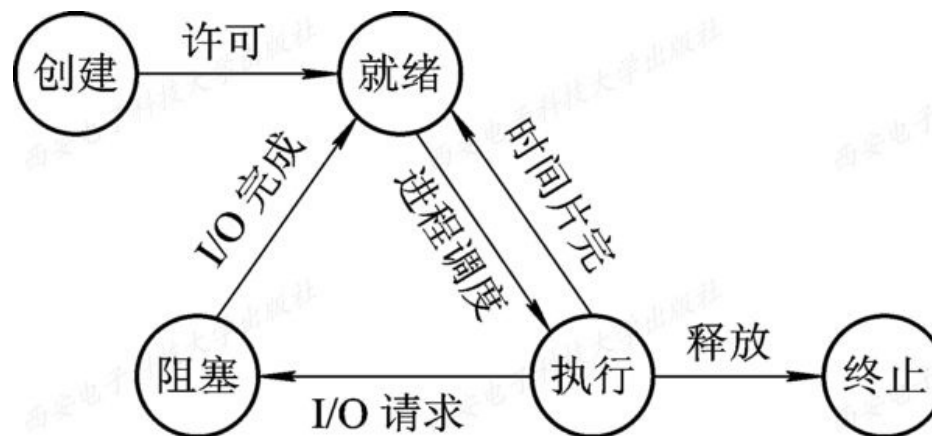
### 2.8.3 线程的创建和终止

#### 2. 线程的终止

当一个线程完成了自己的任务(工作)后，或是线程在运行中出现异常情况而须被强行终止时，由终止线程通过调用相应的函数(或系统调用)对它执行终止操作。

但有些线程(主要是系统线程)，它们一旦被建立起来之后，便一直运行下去而不被终止。

在大多数的OS中，线程被中止后并不立即释放它所占有的资源，只有当进程中的其它线程执行了分离函数后，被终止的线程才与资源分离，此时的资源才能被其它线程利用。



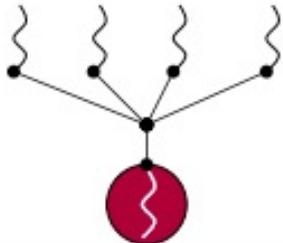
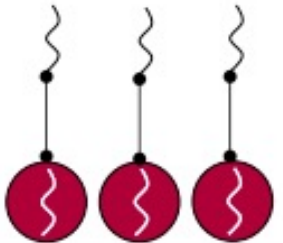
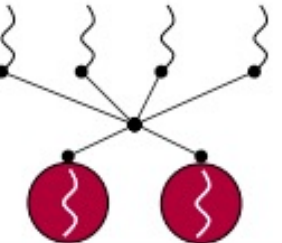
## 2.7 线程(Threads)的基本概念

### 2.7.2 线程与进程的比较

	调度的基本单位	并发性	拥有资源	独立性	系统开销	多处理机
无线程的OS	进程	✓	以进程为单位	进程高	大	不支持
有线程的OS	线程	进程+线程	线程仅拥有必不可少的部分资源	线程低	小	支持

## 2.8 线程的实现

### 用户级线程、核心级线程的对比

	用户级线程	核心级线程	用户+核心级
实现模型			
利用多核	差	好	好
并发度	低	高	高
内核改动	无	大	大
用户灵活性	大	小	大

1. 什么是前趋图？为什么要引入前趋图？
2. 试画出下面四条语句的前趋图：  
S1:  $a = x + y$  ;  
S2:  $b = z + 1$  ;  
S3:  $c = a - b$  ;  
S4:  $w = c + 1$  ;
3. 为什么程序并发执行会产生间断性特征？
4. 程序并发执行时为什么会失去封闭性和可再现性？
5. 在操作系统中为什么要引入进程的概念？它会产生什么样的影响？
6. 试从动态性、并发性和独立性上比较进程和程序。
7. 试说明PCB的作用具体表现在哪几个方面，为什么说PCB是进程存在的唯一标志？
8. PCB提供了进程管理和进程调度所需要的哪些信息？

# 习题



9. 进程控制块的组织方式有哪几种?
10. 何谓操作系统内核? 内核的主要功能是什么?
11. 试说明进程在三个基本状态之间转换的典型原因。
12. 为什么要引入挂起状态? 该状态有哪些性质?
13. 在进行进程切换时, 所要保存的处理机状态信息有哪些?
14. 试说明引起进程创建的主要事件。
15. 试说明引起进程被撤消的主要事件。
16. 在创建一个进程时所要完成的主要工作是什么?
17. 在撤消一个进程时所要完成的主要工作是什么?
18. 试说明引起进程阻塞或被唤醒的主要事件是什么?
19. 为什么要在OS中引入线程?
20. 试说明线程具有哪些属性?

# 习题



21. 试从调度性、并发性、拥有资源及系统开销方面对进程和线程进行比较。
22. 线程控制块TCB中包含了哪些内容？
23. 何谓用户级线程和内核支持线程？
24. 试说明用户级线程的实现方法。
25. 试说明内核支持线程的实现方法。
26. 多线程模型有哪几种类型？多对一模型有何优缺点？