Computer

操作系统原理与实践

第二章 进程管理

高等教育出版社

北京汉众信息科技有限责任公司策划

第二章　进程管理

- 目的与要求：掌握进程的定义、进程描述、进程控制块和进程控制、进程的同步和互斥及用信号量机制解决进程同步问题的方法，掌握进程的调度、死锁的定义及解决死锁的方法。理解线程的概念及线程的管理、进程通信。
- 重点与难点：进程的定义、进程的同步和互斥及用信号量机制解决进程同步问题的方法、进程的调度、死锁的定义及解决死锁的方法。

第二章　进程管理

- 2.1　进程
- 2.2　线程
- 2.3　同步
- 2.4　信号量
- 2.5　进程间通信
- 2.6　进程调度
- 2.7　死锁

2.1 进程

- 2.1.1 进程概念
- 2.1.2 进程特性
- 2.1.3 进程状态与转换
- 2.1.4 进程组成
- 2.1.5 进程控制块和进程队列
- 2.1.6 进程控制

2.1.1 进程概念

- 进程概念
  - 进程是执行中的程序的抽象
- 关于进程概念有不同的说法：
  - 进程是程序的一次执行
  - 进程是可以与其他计算并发执行和计算
  - 进程是一个程序及其数据在处理器上顺序执行时发生的活动
  - 进程是进程实体的一次活动
  - 进程是支持程序运行的机制
- 进程定义：
  - 进程是具有一定功能的程序在一个数据集合上的运行过程，它是系统进行资源分配和调度管理的一个可并发执行的基本单位

2.1.2 进程特性

- 进程特性：
  - 动态性
  - 并发性
  - 独立性
  - 异步性
  - 结构特性

进程的特征

- 动态性：进程的实质是程序的一次执行过程。

进程是动态产生，动态消亡的，进程在其生命周期内，在三种基本状态之间转换。

- 并发性：任何进程都可以同其他进程一起向前推进。
- 独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位。
- 异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进。
- 结构特征：为了控制和管理进程，系统为每个进程设立一个进程控制块— PCB。

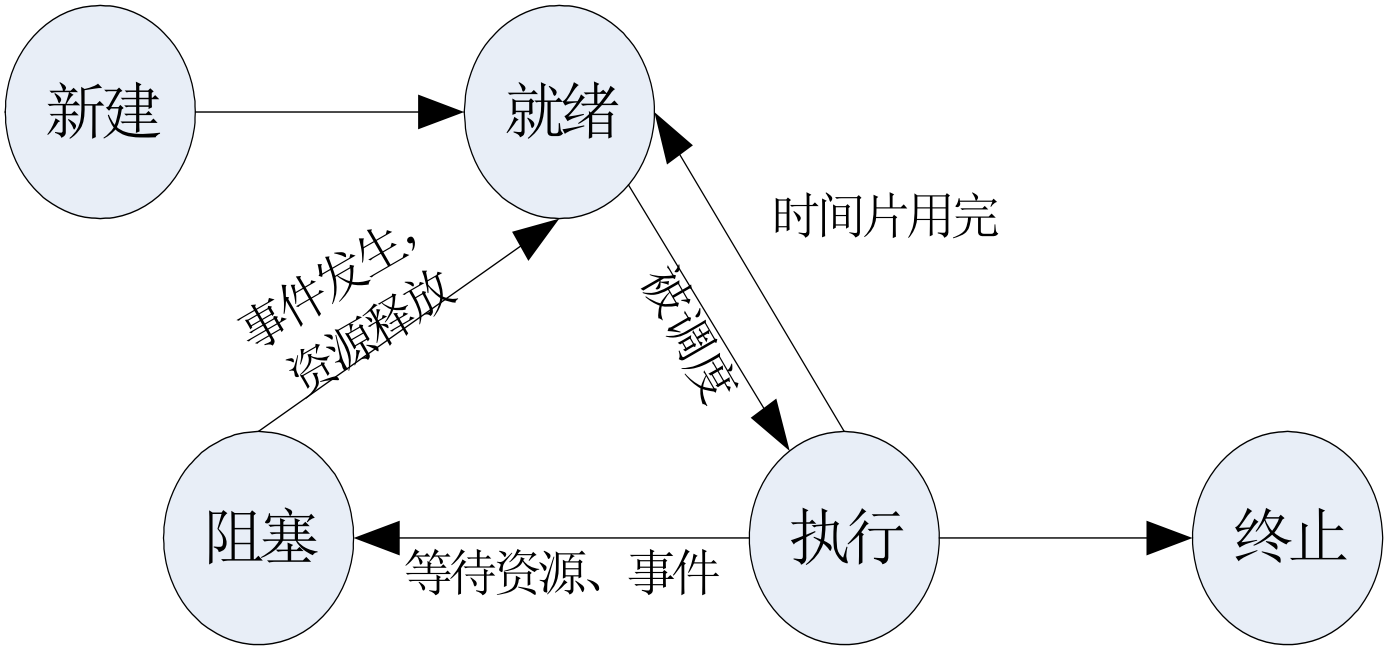
2.1.2进程特性

- 进程与程序
  - 进程是动态的，而程序是静态的
  - 1个程序可以对应多个进程；
  - 而1个进程只能对应1个程序
- 系统进程
- 用户进程

2.1.3进程状态和转换

- 进程状态和转换
  - 就绪(Ready)状态
  - 执行(Running) 状态
  - 阻塞(Blocked)状态

2.1.3进程状态和转换



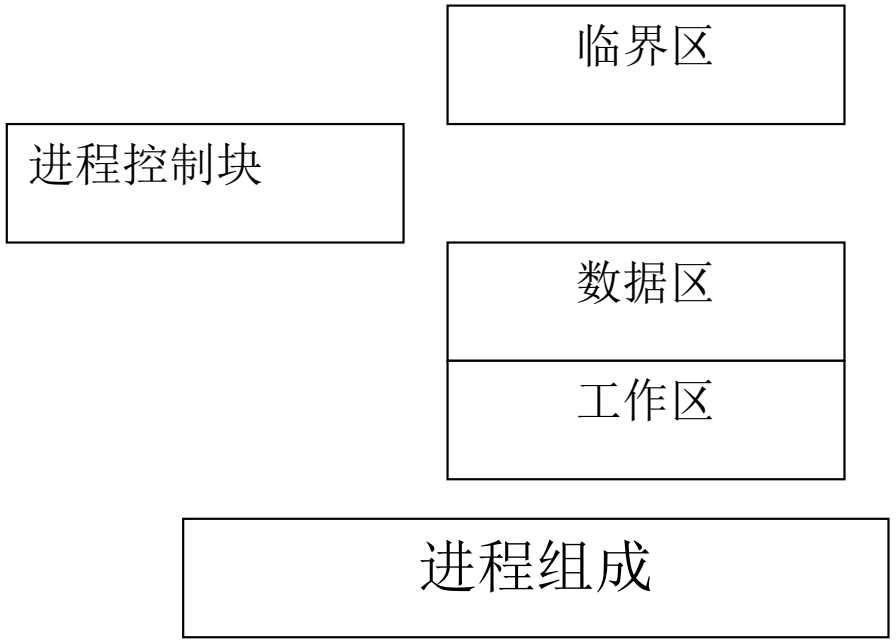
进程状态的转换

2.1.3进程状态和转换

- 进程状态和转换
  - 就绪→执行 ： 被调度
  - 执行→阻塞：等待资源、事件
  - 阻塞→就绪：事件发生，资源释放
  - 执行→就绪：时间片用完

2.1.4进程的组成

- 进程的组成
- **PCB**
  - 临界区
  - 工作区
  - 数据区



2.1.4进程的组成

- 进程控制块
  - 一个进程只有一个**PCB**
  - 是进程存在与否的唯一标记
- 描述信息
- 管理信息

2.1.4进程的组成

- 进程控制块
  - 进程标识信息
  - 进程状态
  - 进程特征
  - 进程位置及大小信息
  - 处理器现场保留区
  - 进程资源清单
  - 进程同步与通信机制
  - 进程间联系

2.1.5进程控制块和进程队列

进程控制块与进程队列

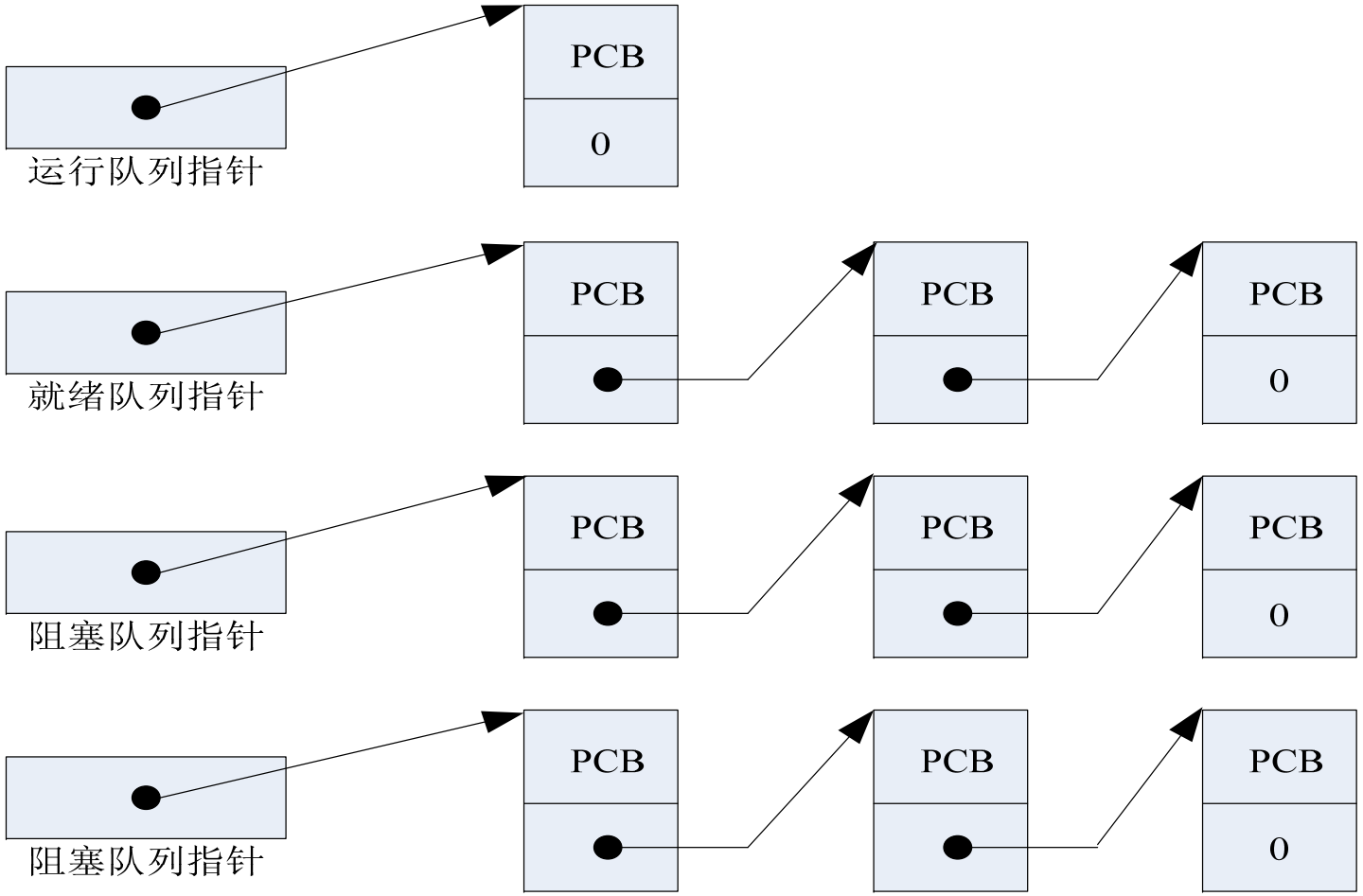
- 线性方式

PCB1	PCB2	PCB3	...	...	PCBn-2	PCBn-1	PCBn

- 链接方式

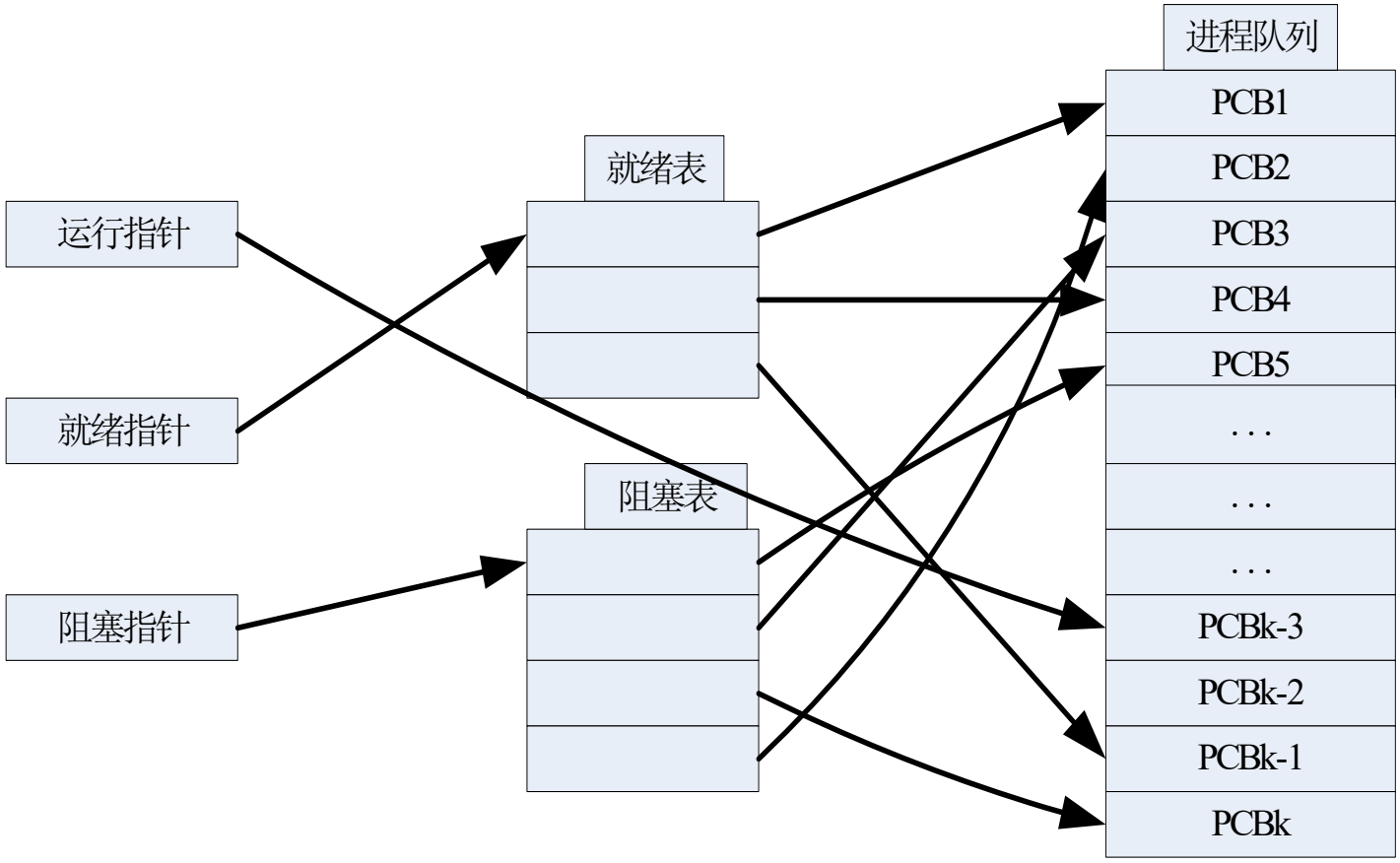
- 索引方式

2.1.5进程控制块和进程队列



连接方式

2.1.5进程控制块和进程队列



索引方式

2.1.6进程控制

- 进程控制的主要任务

进程控制是对系统中所有进程从产生、存在到消亡的全过程实行有效的管理和控制。进程控制一般是由操作系统的内核来实现，内核在执行操作时，往往是通过执行各种原语操作来实现的。

2.1.6进程控制

创建、撤消进程以及完成进程各状态之间的转换。由具有特定功能的原语完成。

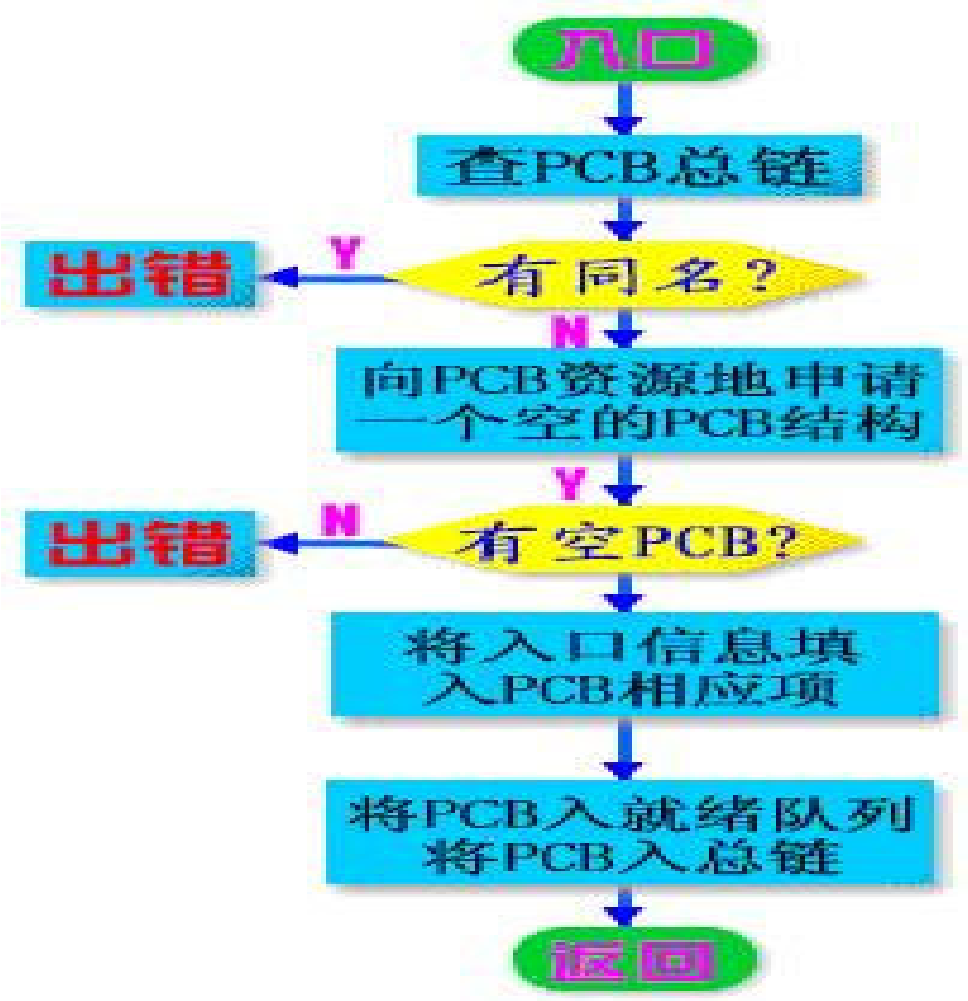
- ✓ 进程创建原语
- ✓ 进程撤消原语
- ✓ 阻塞原语
- ✓ 唤醒原语

- 原语：操作系统内核中用于完成特定功能的一个过程 ，此过程在执行过程中呈现原子特征，不可中断。

进程创建过程

- 创建一个PCB
- 赋予一个统一进程标识符
- 为进程映象分配空间
- 初始化进程控制块
  - 许多默认值（如：状态为 New，无I/O设备或文件...）
- 设置相应的链接
  - 如：把新进程加到就绪队列的链表中

创建原语的实现过程


入口
查PCB总链
有同名？
Y
出错
N
向PCB资源地申请
一个空的PCB结构
Y
有空PCB？
N
出错
将入口信息填
入PCB相应项
将PCB入就绪队列
将PCB入总链
返回

引起撤消的原因

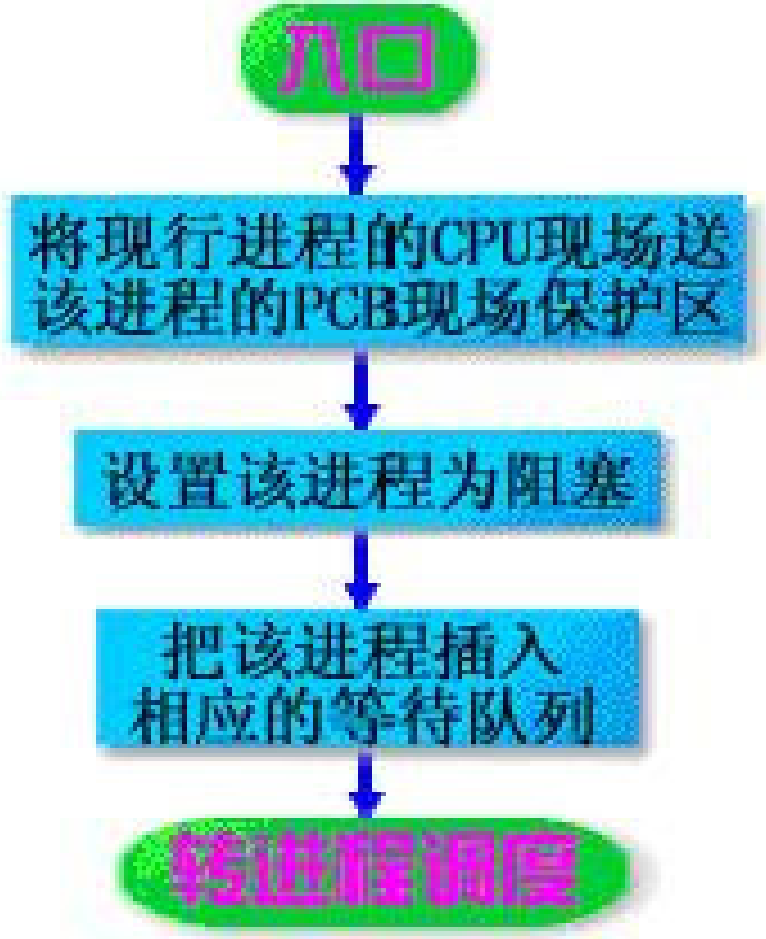
- 正常结束
- 异常结束（越界错、保护错、特权指令错、非法指令、运行超时、I/O故障等）
- 外界干预（操作员干预、死锁、父进程请父进程终止）

撤消原语的实现过程


入口
以撤消名查PCB总链
有此PCB?
N
出错
Y
将该进程从现行队列的总链中摘除
将该进程的资源回收
有子进程?
N
Y
将子进程的PCB从有关链中摘除，并回收
回收PCB
返回

进程的阻塞原语

- 功能：停止调用进程的执行，变为等待。
- 入口信息：可省
- 阻塞原语的实现过程



进程的唤醒原语

- 功能：唤醒某一处于等待队列当中的进程。
- 入口信息：被唤醒进程的名字
- **引起唤醒的原因**

  系统服务由不满足到满足

  I/O完成

  新数据到达

  进程提出新请求（服务）

唤醒原语的实现过程

2.2 线程

- **2.2** 线程
- **2.2.1** 线程引入和线程概念
- **2.2.2** 线程的实现
- **2.2.3** 线程池
- **2.2.4** 线程优势

2.2.1 线程引入和线程概念

1. 线程引入

进程的两个基本属性：

- 资源的拥有者：

  给每个进程分配一虚拟地址空间，保存进程映像，控制一些资源（文件，I/O设备），有状态、优先级、调度

- 调度单位：

  进程是一个执行轨迹

  以上两个属性构成进程并发执行的基础

2.2.1 线程引入和线程概念

对进程系统必须完成的操作：

- 创建进程
- 撤消进程
- 进程切换

缺点：

时间空间开销大，限制并发度的提高。

2.2.1 线程引入和线程概念

2. 线程概念：

- 线程（**Thread)**是进程中实施调度和分派的基本单位

3. 线程状态：

- 运行状态、阻塞状态、就绪状态、终止状态

2.2.1 线程引入和线程概念

4. 线程管理：

- 线程创建
- 线程终止
- 线程等待
- 线程让权

2.2.1 线程引入和线程概念

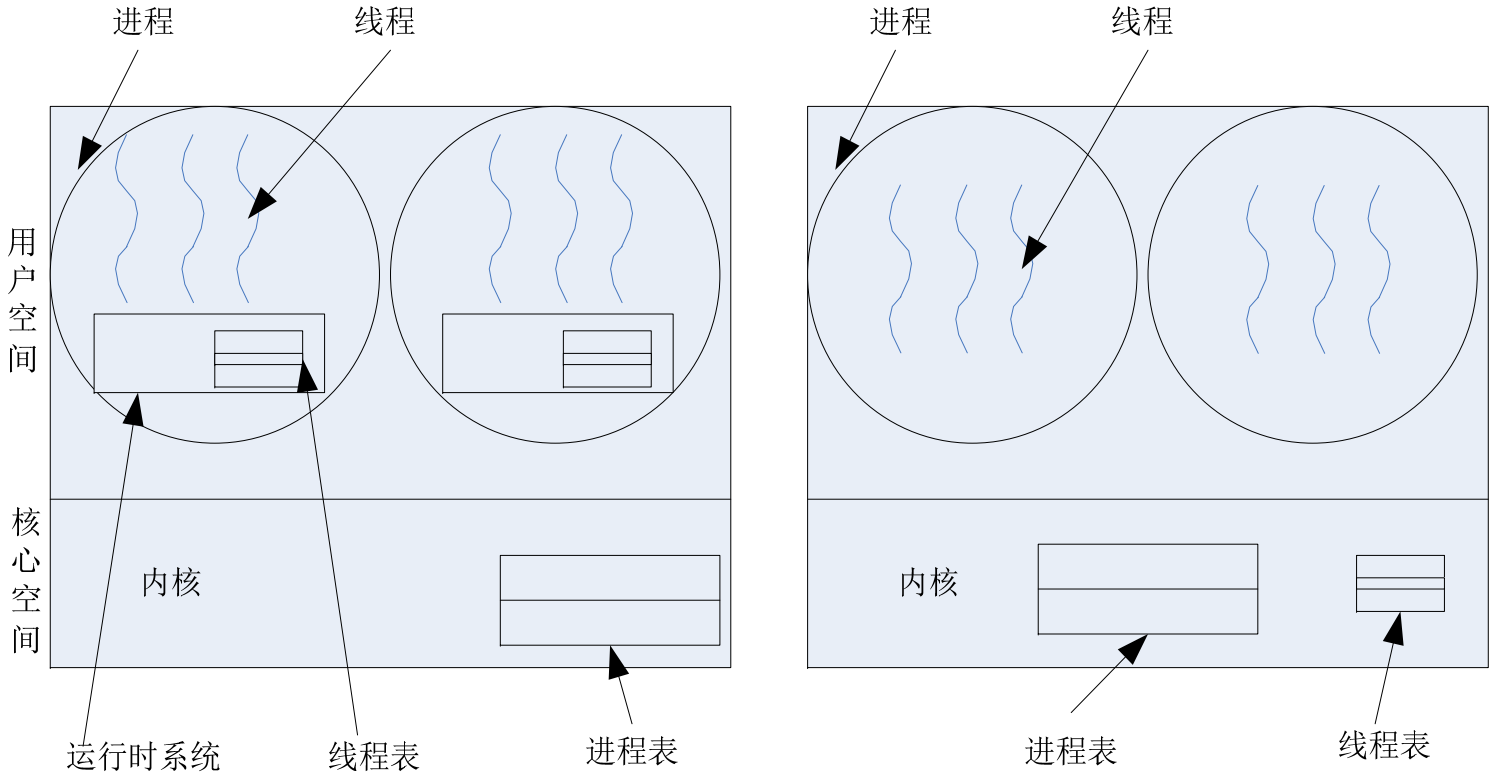
5. 线程与进程的关系：

- 一个进程对应多个线程；一个线程只能在一个进程的地址空间内活动
- 资源分配给进程；同一进程的线程共享资源

- **CPU分配给线程**
- 线程在执行过程中需要协作同步；不同进程则利用通信实现同步

2.2.2 线程的实现

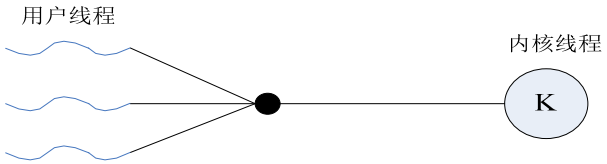
- 用户级线程与核心级线程
- 用户级线程与核心级线程实现方式



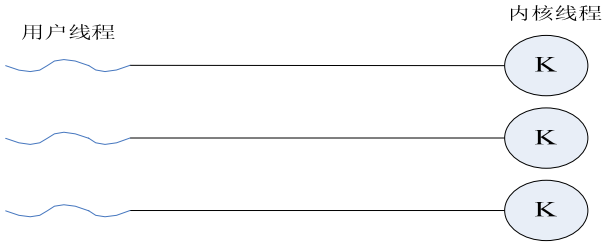
用户级线程和核心级线程的实现方式

2.2.2 线程的实现

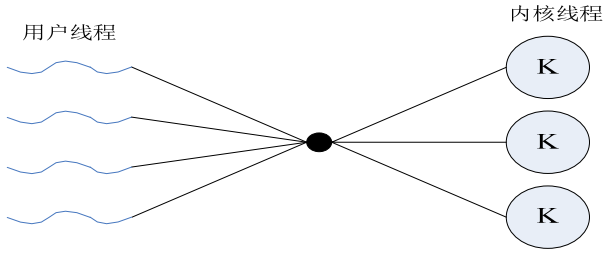
- 用户级线程与核心级线程的比较
  - 多对一模型



  - 一对多模型



  - 多对多模型



2.2.3 线程池

- 线程池定义
- 线程池优势
- 线程池设计原则

2.2.4 线程优势

- 线程优势
  - 响应度高
  - 资源共享
  - 经济
  - 效能高

2.3 同步

- 2.3.1 进程同步和进程间通信
- 2.3.2 互斥的实现方式

2.3.1 进程同步和进程间通信

- 进程间相互关系
  - 同步

所谓同步是指一组相互协同的进程，在完成同一任务，对某些共享资源进行操作时，为协调资源占用而相互等待、相互交换信息所产生的制约关系。

  - 互斥

所谓互斥是指并发进程间因相互竞争使用独占资源所产生的制约关系。

  - 通信

进程间交换信息称为通信。

2.3.1 进程同步和进程间通信

- 临界资源与临界区
  - 临界资源:就是一次只能允许一个进程使用。
  - 临界区:每个进程中访问或者使用临界资源的那段代码程序就叫做临界区。
  - 进程的通用结构

Repeat
entry section 进入区
critical section 临界区
exit section 退出区
remainder section 剩余区
Until false

5. 使用临界区的原则

- 空闲让进：当无进程在互斥区时，任何有权使用互斥区的进程可进入
- 忙则等待：不允许两个以上的进程同时进入互斥区
- 有限等待：任何进入互斥区的要求应在有限的时间内得到满足
- 让权等待：处于等待状态的进程应放弃占用CPU，以使其他进程有机会得到CPU的使用权

2.3.2 互斥的实现方式

- 解决进程互斥进入临界区有两种方式：
  - 硬件
  - 软件
- 硬件解决方式：
  - 禁止中断
  - 专用机器指令
- 软件算法实现：
  - 进入区、退出区、剩余区

2.4 信号量

- **2.4.1** 整形信号量
- **2.4.2** 记录型信号量
- **2.4.3** 信号量的应用
- **2.4.4** 经典同步问题

2.4.1　整形信号量

- 信号量方法提出：
  - 荷兰学者E.W.Dijkstra整形信号量
  - 1、信号量可以初始为一个非负值
  - 2、只能由P和V操作来访问信号量
- P操作：测试；DOWN操作
- V操作：增加；UP操作
- **(P) Wait(s):　while s<=0 do no-op;**
  s:=s-1;

- **(V)Signal(s):　s:=s+1;**

2.4.1 整形信号量

- 进程Pi利用信号量实现互斥的伪代码形式：

```
do{
        P(mutex)
        临界区（Critical Section）
       V(mutex)
        剩余区（remainder section）
} while(1);
```

2.4.2 记录型信号量

- 为解决忙等待引入记录型信号量
- 记录型信号量定义:由两个成员组成的数据结构，其中一个成员是整型变量，表示该信号的值；另一个是指向PCB的指针。当多个进程都要等待同一信号量时，它们就排成一个队列，由信号量的指针项指示该队列的队首，进程控制块PCB是通过PCB自身所包含的指针项进行链接的，最后一个PCB（队尾）的链接指针为0。

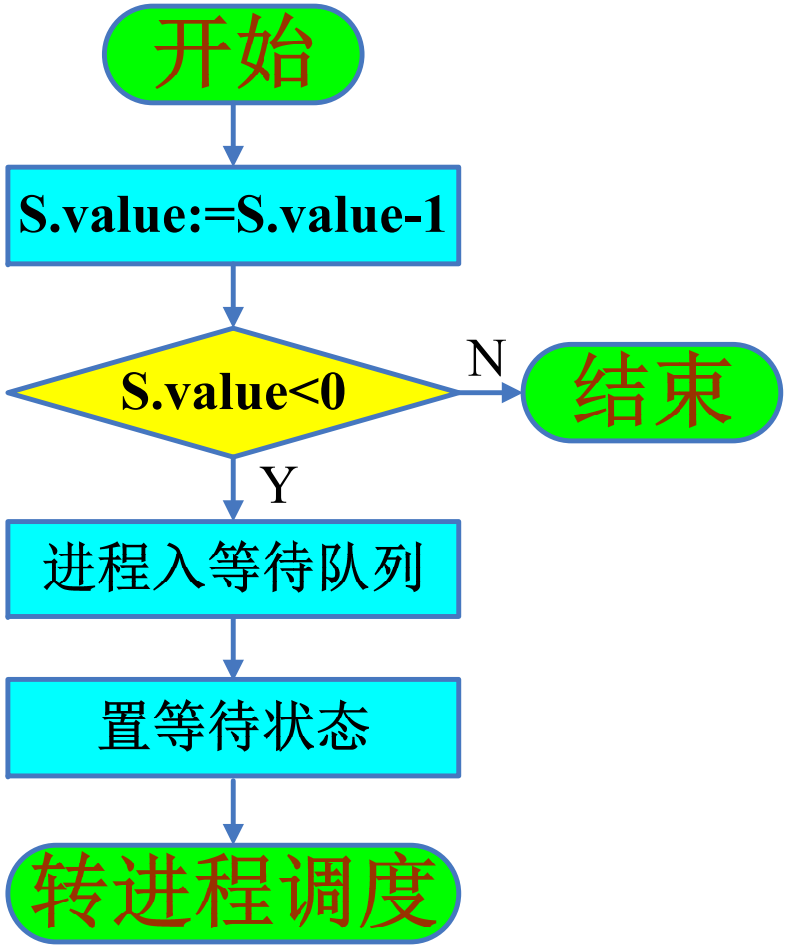
2.4.2 记录型信号量

- 记录型信号量值和相应资源有关
  - 大于0：表示当前可用资源数量
  - 小于0：绝对值表示等待进程个数
  - P（S）操作
  - V（S）操作

2.4.2 记录型信号量

- P操作的定义如下：

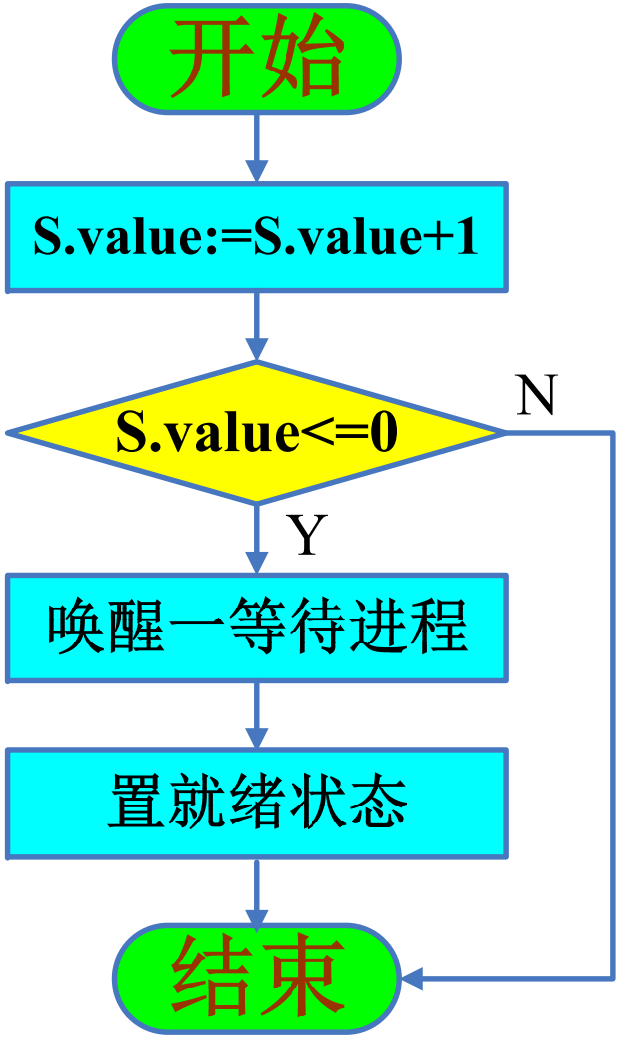
```
Void P(S){
    S．value --;
    if ( S．value<0)
    {将这个进程加到S．list队列；
    block()；  }
}
```



2.4.2 记录型信号量

- V操作的定义如下：

```
Void V(S){
        S．value ++;
      if ( S．value<=0){
     从S．list队列中将Q移走；
      wakeup(Q)；  }
      }
```



2.4.3 信号量的应用

- 用信号量实现进程的互斥
  - 设置一个互斥信号量mutex，其初值为1。PI（分配进程）和PO（释放进程）的临界区代码可描述成：

PI

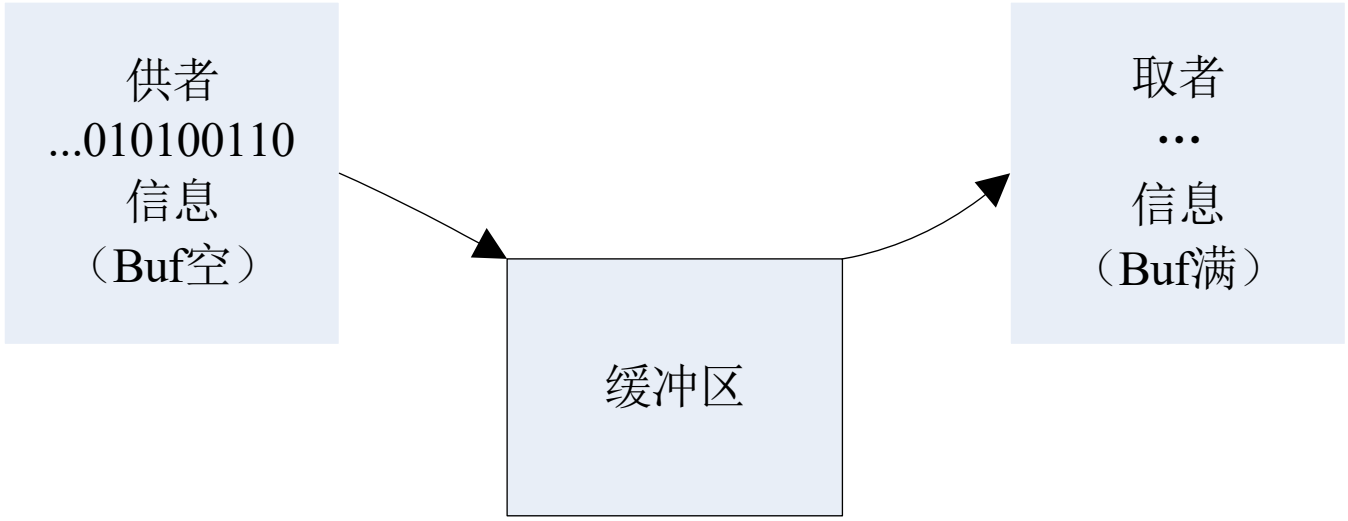
```
...
P（mutex）
分配打印机
（读写分配表）
V（mutex）
...
```

PO

```
...
P（mutex）
释放打印机
（读写分配表）
V（mutex）
...
```

2.4.3 信号量的应用

- 用信号量实现进程间同步
- 以短信缓冲池的同步使用为例，进程A（供者）和进程B（取者）对短信缓冲池的使用关系如图所示



2.4.3 信号量的应用

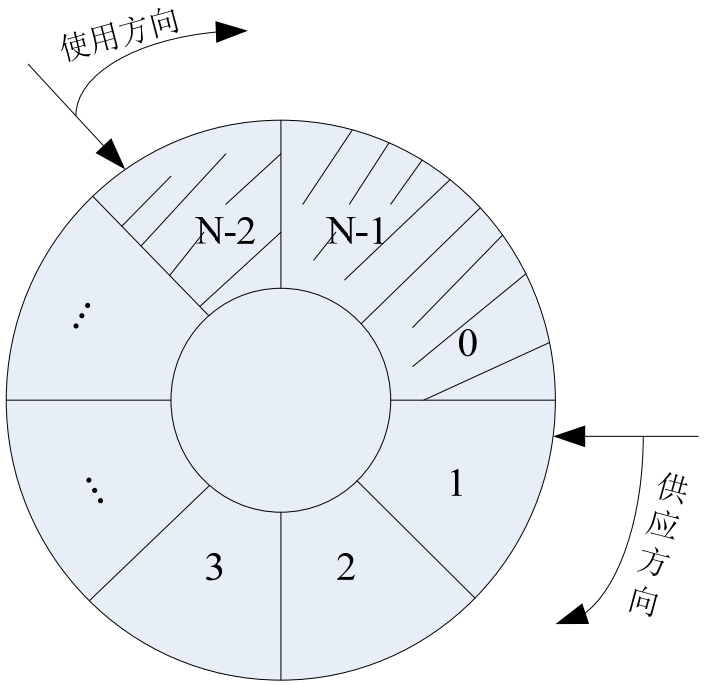
- 规定S1和S2的初值分别为1和0，对缓冲池的供者进程和取者进程的同步关系用下述方式实现：

供者进程	取者进程
repeat	repeat
P(S1)；	。 。 。
放入信息到缓冲池中	P（S2）；
。 。 。	从缓冲池中取
出信息	
信息放入结束	V（S1）；
V（S2）；	将短信信息发
出	
until false	until false

2.4.4 经典同步问题

- 生产者和消费者问题

生产者—消费者(Producer-Consumer)问题是著名的进程同步问题。它描述一组生产者向一组消费者提供消息，它们共享一个有界缓冲池，生产者向其中投放消息，消费者从中取得消息。以下用信号量解决生产者—消费者问题。



生产者和消费者问题缓冲池

生产者-消费者问题

- 同步
- 互斥
- 问题限制

- 假设缓冲池中有n个缓冲区，每个缓冲区存放一个消息，可利用互斥信号量mutex使诸进程对缓冲池实现互斥访问；利用empty和full计数信号量分别表示空缓冲及满缓冲的数量。又假定这些生产者和消费者互相等效，只要缓冲池未满，生产者可将消息送入缓冲池；只要缓冲池未空，消费者可从缓冲池取走一个消息。

2.4.4 经典同步问题

- 生产者进程（Producer）　消费者进程(Consumer)

```
while(true) {                    while(true){
P（empty）;                      P（full）;
P（mutex）;                      P（mutex）;
产品放入缓冲区buffer(in);        缓冲区取出一产品 buffer(out);
in=(in+1)mod N; /*以N为模*/      out=(out+1)mod N; /*以N为模*/
V（mutex）;                      V（mutex）;
V（full）;                       V（empty）;
```

2.4.4 经典同步问题

- 注意三点：
  - 1、每个进程先执行P操作，后执行V操作;
  - 2、同步信号量的PV操作要成对出现；应该在不同的进程中;
  - 3、两个进程的P操作次序不能颠倒.

2.4.4 经典同步问题

- 读者与写者问题

有两组并发进程:

读者和写者, 共享一组数据区

要求:

允许多个读者同时执行读操作

不允许读者、写者同时操作

不允许多个写者同时操作

2.4.4 经典同步问题

- 如果读者来：

1）无读者、写者，新读者可以读

2）有写者等，但有其它读者正在读，则新读者也可以读

3）有写者写，新读者等

- 如果写者来：

1）无读者，新写者可以写

2）有读者，新写者等待

3）有其它写者，新写者等待

2.4.4 经典同步问题

- 设置两个信号量：读互斥信号量**rmutex**和写互斥信号量**wmutex**。另外设立一个读者计数器**readcount**，它是一个整型变量，初值为**0**。
- **rmutex**用于用户进程互斥访问**readcount**，初值为**1**。
- **wmutex**用于保证一个写者与其它用户互斥地访问共享资源，初值为**1**。

2.4.4 经典同步问题

读者：

```
while (true) {
P(rmutex);
    readcount ++;
    if (readcount==1)
        P (wmutex);
V(rmutex);
   读
P(rmutex);
    readcount --;
    if (readcount==0)
        V(wmutex);
 V(rmutex);
 };
```

写者：

```
    while (true) {

     P(wmutex);
      写
     V(wmutex);

      };
```

2.4.4 经典同步问题

- 理发师问题

理发店里有一位理发师和一把理发椅，还有 N 张座椅供等候的顾客休息。在没有顾客的时候，理发师就会躺在理发椅上睡觉。因此，当有顾客来到理发店时，他必须先叫醒理发师。如果理发师正在理发时又有顾客来到，此时，如果有空椅子可坐，他们就会坐下来等；如果没有空椅子，就会离开。

问题：用信号量和P、V原语写出理发师和顾客行为的程序描述。

理发师问题

理发师问题分析

顾客的做法

S1 判断有多少人在等待，有多少把椅子，若椅子不够，转S5；
S2 如果是第一个顾客，就去叫醒理发师；
S3 走到一把空椅子处坐下等，等理发师叫自己；
S4 理发中…；
S5 离开理发店。

理发师的做法

S1 判断是否有人在等，若没有，就去睡觉；若有，转S2；
S2 去唤醒一个正在等待的顾客；
S3 理发中…；
S4 转S1。

理发师问题

- 分析：
- 理发师和每位顾客都分别是一个进程。
- 理发师开始工作时，先观察一下店内是否有顾客，如果没有他就在理发椅子上休息；如果有顾客，他就为等待时间最长的顾客服务，且顾客等待人数减1。
- 每位顾客进程开始执行时，先看店内是否有空位，如果没有空位，就不等了，离开理发店；如果有空位，则排队，顾客等待人数加1；如果理发师在休息，则唤醒理发师工作。

理发师问题

- 解答：
- 引入3个信号量和一个控制变量：
- 1)控制变量waiting用来记录等候理发的顾客数，初值设为0；
- 2)信号量customers用来记录等候理发的顾客数（不包括正在理发的顾客），并用作阻塞理发师进程，初值为0；
- 3)信号量barbers用来记录正在等候顾客的理发师数，并用作阻塞顾客进程，初值为0；
- 4)信号量mutex用于互斥，初值为1；

理发师问题

- var waiting : integer； /*等候理发的顾客数*/
- CHAIRS:integer； /*为顾客准备的椅子数*/
- Customers, barbers， mutex : semaphore；
- customers := 0； barbers := 0；
- waiting := 0； mutex := 1;

睡着的理发师问题

```
Procedure barber;
 {
   while(TRUE)；/*理完一人,还有顾客吗?*/
    P(cutomers)；/*若无顾客,理发师休息*/
    P(mutex)；  /*进程互斥*/
    waiting ： = waiting-1 /*等候顾客数少一个*/
    V(barbers)；   /*理发师去为一个顾客理发*/
    V(mutex)；   /*退出临界区*/
     cut-hair( )；   /*正在理发*/
   }
```

睡着的理发师问题

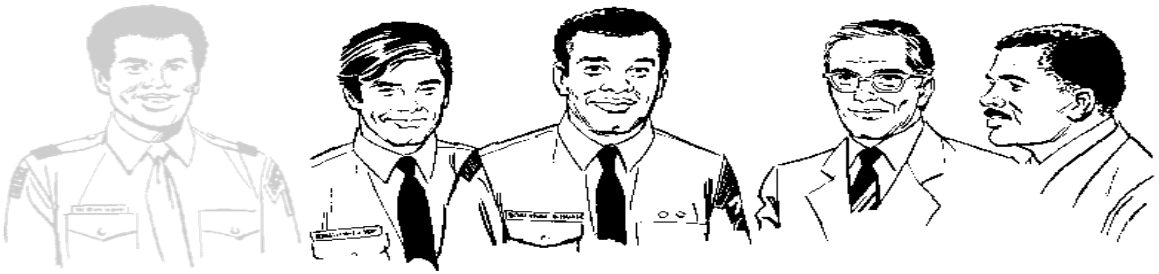
```
Procedure customer
 {
   P(mutex);        /*进程互斥*/
   if (waiting <CHAIRS)
   {waiting：= waiting+1；    /* 等候顾客数加1*/
    V(customers);           /*必要的话唤醒理发师*/
    V(mutex);               /*退出临界区*/
   P(barbers);             /*无理发师，顾客坐着养神*/
   get-haircut( );         /*一个顾客坐下等理发*/
  } else
 V(mutex);      /*人满了,走吧!*/
}
```

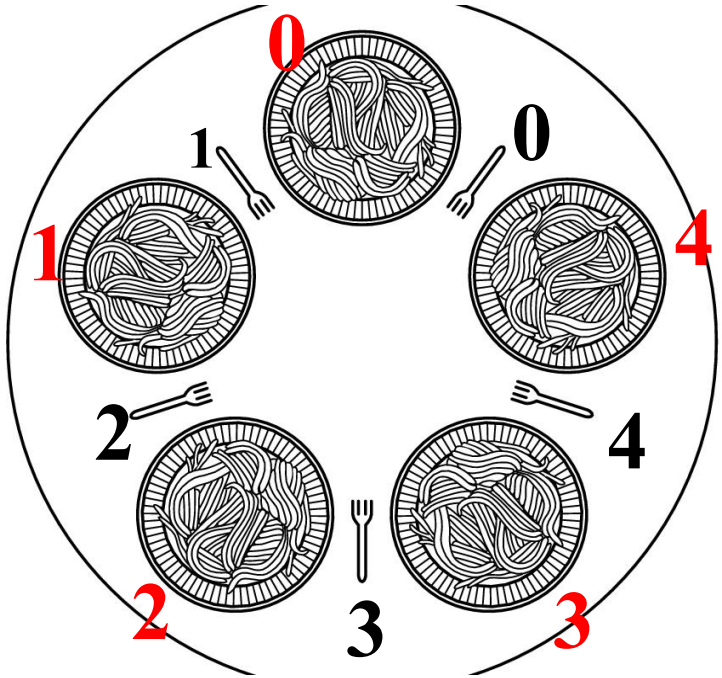
哲学家就餐问题

有五个哲学家围坐在一圆桌旁，桌中央有一盘通心粉，每人面前有一只空盘子，每两人之间放一只筷子

每个哲学家的行为是思考，感到饥饿，然后吃通心粉

为了吃通心粉，每个哲学家必须拿到两只筷子，并且每个人只能直接从自己的左边或右边去取筷子





哲学家就餐问题

```
#define  N  5
 void philosopher (int i)
 {
    while (true)  {
    思考;
    取fork[i]; 取fork[(i+1)
% 5];
    进食;
    放fork[i]; 放fork[(i+1)
% 5];
    }
}
```

```
Var chopstick:array [0,…,4] of
semaphore;
第i个哲学家的活动可描述为:
repeat
  wait(chopstick[i]);
  wait(chopstick[(i+1)  mod  5]);
  ...
  eat;
  ...
  signal(chopstick[i]);
  signal(chopstick[(I+1)  mod  5]);
  ...
  think;
until  false;
```

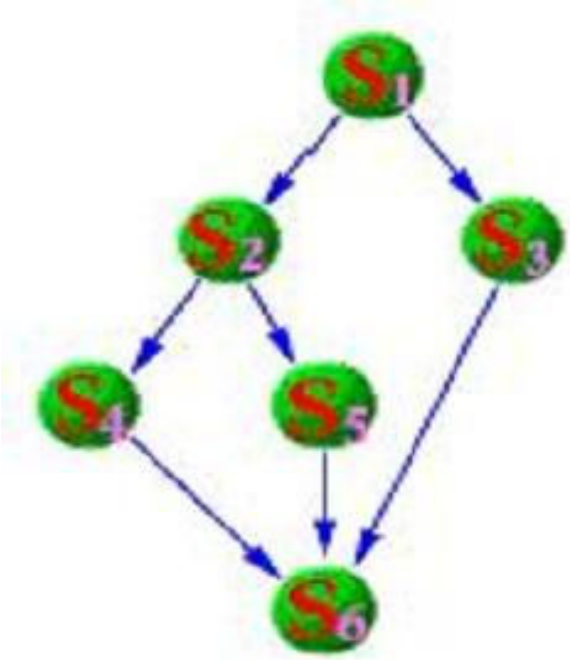
哲学家就餐问题

为防止死锁发生可采取的措施：

- 最多允许4个哲学家同时坐在桌子周围
- 仅当一个哲学家左右两边的筷子都可用时，才允许他拿筷子（√）
- 给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之。
- 为了避免死锁，把哲学家分为三种状态，思考，饥饿，进食，并且一次拿到两只筷子，否则不拿

用P-V操作描述前趋关系的例子

- 信号量还可以描述程序或语句之间的前趋关系。



用P-V操作描述前趋关系（续）

描述如下：

```
Var a, b, c, d, e, f, g: semaphore: =0, 0, 0,  0, 0,
   0, 0;
begin
parbegin
begin  S1; V(a); V(b); end;
begin  P(a); S2; V(c); V(d); end;
begin  P(b); S3; V(e); end;
begin  P(c); S4; V(f); end;
begin  P(d); S5; V(g); end;
begin  P(e); P(f); P(g); S6; end;
parend
end
```

用P.V操作解决司机与售票员的问题

```
司机进程：
while
(true){
    启动车辆
    正常驾驶
    到站停车
}...
```

```
售票员进程：
while
(true){
        关门
        售票
        开门
}...
```

用P.V操作解决司机与售票员的问题

```
semaphore  S_Door;      // 初始化为0
semaphore  S_Stop;      // 初始化为0
```

```
司机:
while(上班时间)
{
    P(S_Door);
    发动汽车;
    正常运行;
    到站停车;
    V(S_Stop);
}
```

先关门后开车

```
售票员:
while(上班时间)
{
    关闭车门;
    V(S_Door);
    售票;
    P(S_Stop);
    打开车门;
}
```

先停车后开门

信号量和P、V原语的小结

对信号量和P、V原语的使用可以归纳为三种情形：

- 第一，把信号量视为一个加锁标志位，其目的是为了实现对某个唯一的共享数据的互斥访问，如数据库中的某个记录，各个进程间的某个共享变量。该共享数据的取值与信号量本身的取值并没有什么直接的关系，信号量的作用仅仅是作为一个加锁标志位。其特征是信号量的初始值为1，然后在一个进程内部对它进行配对的P、V操作。

```
P(mutex);              // mutex的初始值为1
访问该共享数据;
V(mutex);
非临界区
```

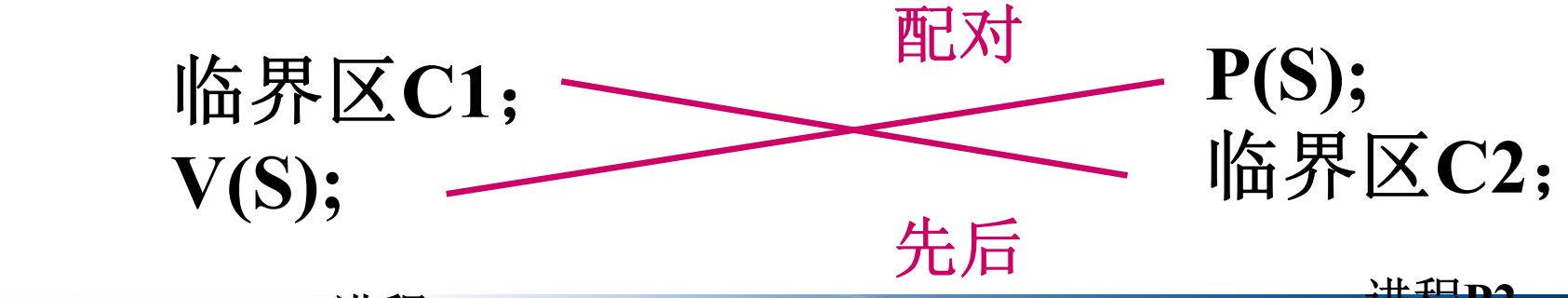
信号量和P、V原语的小结

- 第二，把信号量视为是某种类型的共享资源的剩余个数，其目的是为了实现对这种类型的共享资源的访问，如各种I/O设备。信号量的取值具有实际的意义，就等于空闲资源的个数。多个进程可以同时使用这种类型的资源，直到所有空闲资源均已用完。其特征是信号量的初始值为N(N≥1)，然后在一个进程内部对它进行配对的P、V操作

```
P(resource);          // resource的初始值为N
使用该资源;
V(resource);
非临界区
```

信号量和P、V原语的小结

- 第三，把信号量作为进程间同步的工具，利用它来设定两个进程在运行时的先后顺序。比如说，它可以是某个共享资源的当前个数，但是由一个进程负责生成该资源，而另一个进程负责消费该资源，由此引发了两个进程之间的先后顺序。其特征是信号量的初始值为N（N ≥ 0），然后在一个进程里面用对它使用V原语，增加资源个数，而在另外一个进程里面对它使用P原语，减少资源个数，从而实现两个进程之间的同步关系。



课后练习

图书馆阅览室问题

- 问题描述：假定阅览室最多可同时容纳100个人阅读，读者进入时，必须在阅览室门口的一个登记表上登记，内容包括姓名、座号等，离开时要撤掉登记内容。用P、V操作描述读者进程的同步算法。

2.5 进程通信

- **2.5.1** 进程通信的定义
- **2.5.2** 消息传递
- **2.5.3** 共享存储
- **2.5.4** 管道通信

2.5.1进程通信的定义

- 进程通信是指进程之间可以直接以较高的速率传输较多的数据和信息的信息交换方式。
- 三种基本进程通信方式:
  - 消息传递系统
  - 共享存储器系统
  - 管道通信系统

2.5.2消息传递

- 假如需要通信的进程之间不存在可直接访问的共享空间，必须利用操作系统提供的通信类系统调用来实现进程间通信。
  - 发送原语：**send(**接收者进程名字，发送区首地址**)** 。
  - 接收原语：**receive**（发送者进程名字，接收区首地址）。
- 发送原语和接收原语是系统提供给用户实现进程通信的最基本的原语，也是构成一种具体通信系统的主要内容。

2.5.2消息传递

1、一对一通信：发送消息的进程明确指定接受消息的进程，接受消息的进程也明白消息来自哪个进程，谓之一对一通信。

- **send和receive的形式如下：**
  - **send (p message)** 发送消息到进程**p**
  - **receive (q message)** 从进程**q**接受消息

2.5.2消息传递

- 将两个进程合起来就直接命名了一条信道**pq**，消息通过**pq**信道传递。

(1)消息缓冲区的结构

```
TYPE
 MESSAGE_BUFFER=RECORD
        sender: INETGER; 发送进程号
        size: INETGER; 消息长度(字符数)
        text: STRING; 消息正文
        next: POINTER; 消息缓冲区链接指针
        END;
```

2.5.2消息传递

(2)消息发送原语

```
PROCEDURE send(p, m)
        VAR p: INTEGER；
            m: MESSAGE；
        BEGIN
         IF p<>receiver THEN return("接收进程不存在")；
         getbuf(i: MESSAGE_BUFFER)；/*申请缓冲区*/
         copy(m, i)；                          /*把m复制到i*/
         getpid(q)；    /*获得本进程号，作为发送进程号*/
         i.sender := q；
         P(p.mutex)；    /*消息链是临界资源，必须互斥*/
         insert(p.MessageQuene, i)；/* 把i插入到消息链尾*/
         V(p.mutex)；
         V(p.Sm)；/*消息链长进1，若有等待消息的进程则唤醒它*/
        END；
```

2.5.2消息传递

(3)消息接受原语

```
PROCEDURE receive(q, m)
        VAR q: INTEGER;   /*发送消息的进程号为q*/
            m: MESSAGE;
        BEGIN
         getpid(p);              /* 获得接收进程号p*/
         P(p.Sm);               /* 从消息链申请1个消息*/
         P(p.mutex);        /* 对消息链互斥*/
         remove(p.Message_quene, i);  /* 从消息链取消息*/
         V(p.mutex);
         IF q<>i.sender THEN return("一对一有违例");
         copy(i, m);              /* 把i复制到m*/
         release(i: MESSAGE);   /* 释放i到空间区*/
        END;
```

2.5.2消息传递

2、客户/服务器系统中的通信

用户要访问的数据可能存放在网络的某个服务器，目前大多数用户应用系统都是这种应用形式，人们常称之**Client/Server**方式，简称**C/S**模式。

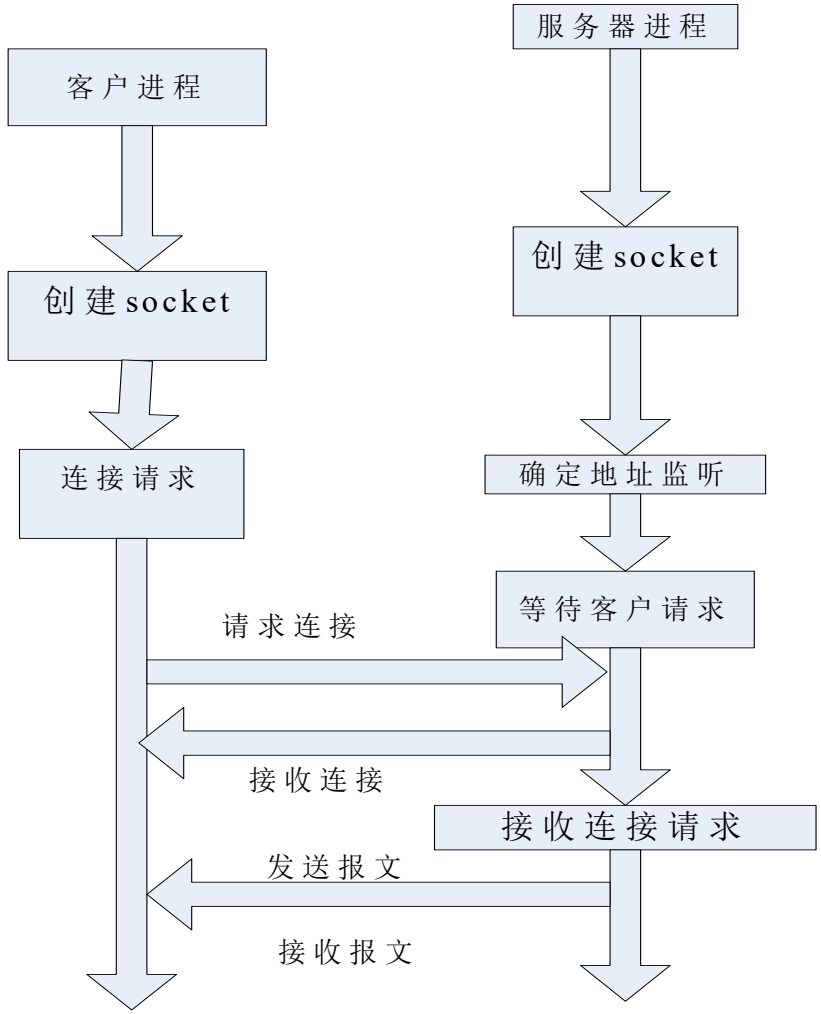
客户/服务器系统中，常用的进程通信方式有**socket**编程和远程过程调用。

2.5.2消息传递

（1）socket 编程

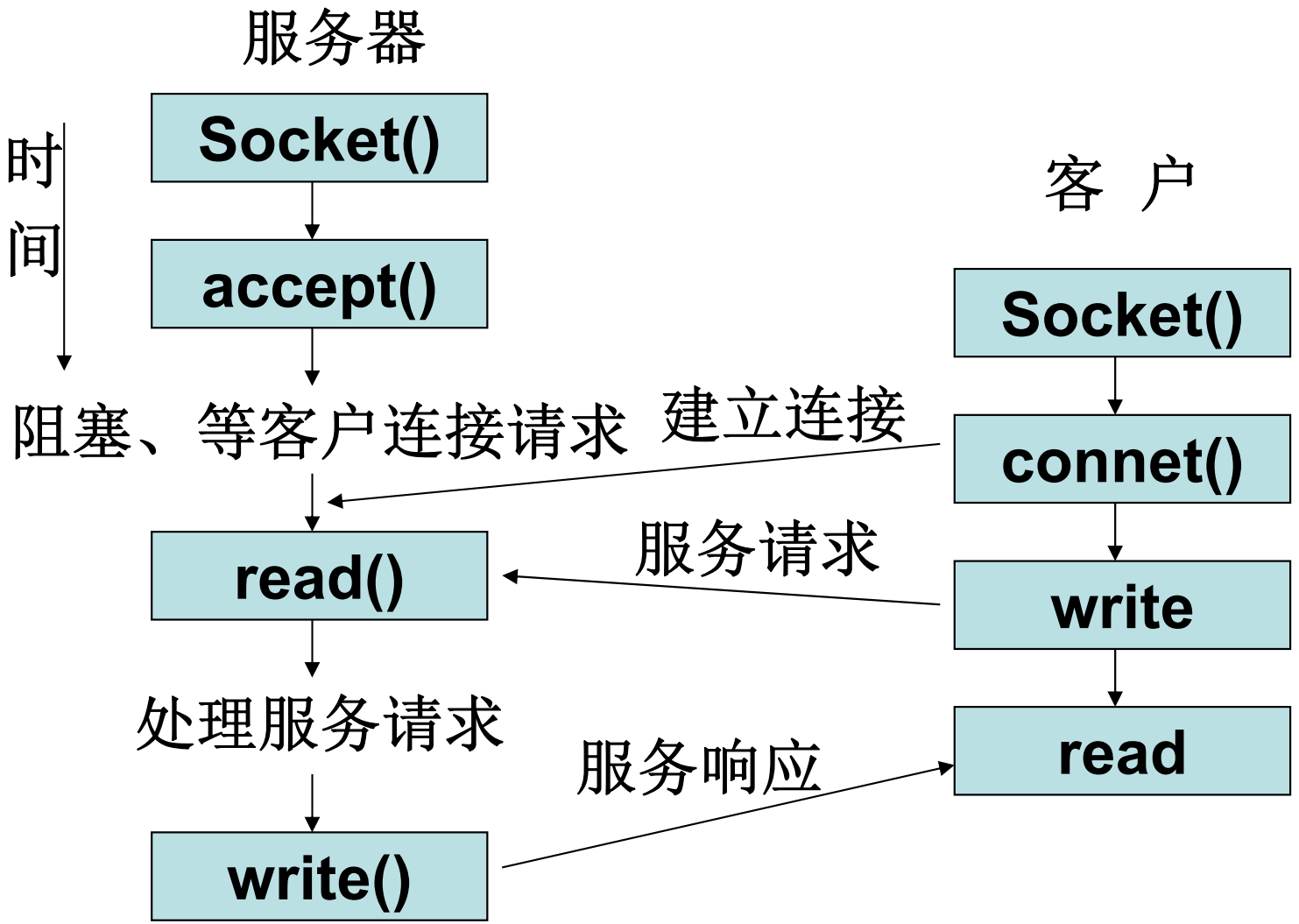
- socket也称套接字或插口，好像一条通信线路两头的接插口。
- 过网络进行通信要用一对socket，每个进程一个。一个socket在逻辑上有网络地址，连接类型和网络规程三个要素。

2.5.2消息传递



Socket通信流程

基于连接的服务器、客户程序流程



2.5.2消息传递

（2）远程过程调用

远程过程调用的思想很简单：允许程序调用另外机器上的过程。当机器A的一个客户（或者线程）调用机器B上的一个过程，A上的调用进程阻塞，被调用过程在B上开始执行。调用者以参数形式将信息传送给被调用者，被调用者将过程执行结果回送给调用者。

2.5.2消息传递

- 远程过程调用的消息是有一定结构的，不再是简单的数据包。
- 远程过程调用的具体步骤有以下几点：

  A、客户过程以通常方式调用客户代理；

  B、客户代理构造一个消息，并且陷入内核；

  C、本地内核发送消息给远程内核；

  D、远程内核将消息送给服务器代理；

2.5.2消息传递

E、服务器代理从信息包中取出参数，并调用服务器；

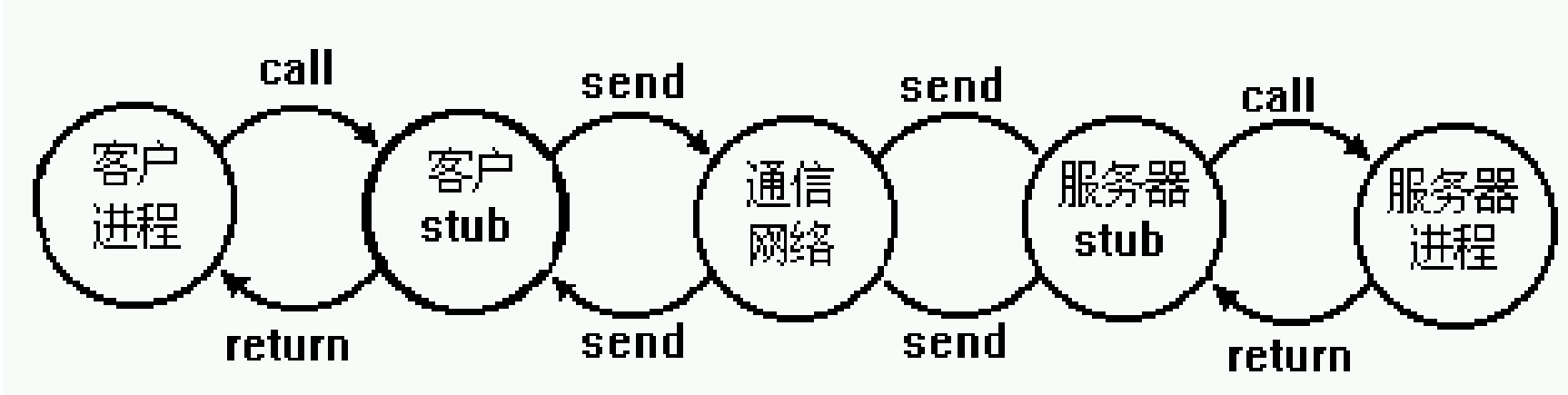
F、服务器完成相应服务，并将结果送给服务器代理；

G、服务器代理将结果打包形成消息，并且陷入内核；

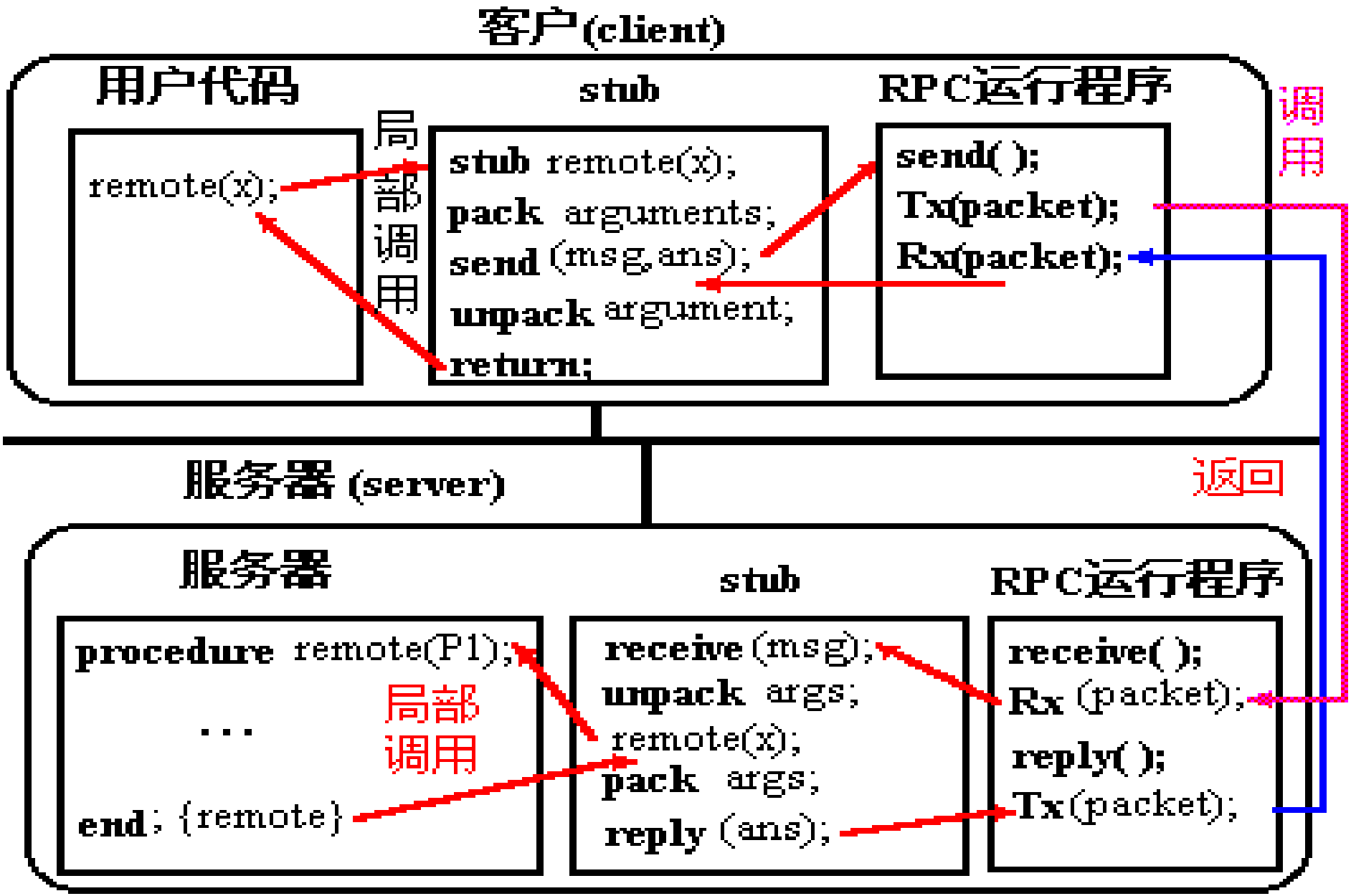
H、远程内核发送消息给客户机内核；

I、客户机内核将消息传给客户代理；

J、客户代理取出结果，并返回客户调用程序。



RPC的实现概况


客户(client)
用户代码
stub
RPC运行程序
remote(x);
局部调用
stub remote(x);
pack arguments;
send (msg,ans);
unpack argument;
return;
send( );
Tx(packet);
Rx(packet);
调用
返回
服务器 (server)
服务器
stub
RPC运行程序
procedure remote(P1);
...
end; {remote}
局部调用
receive (msg);
unpack args;
remote(x);
pack args;
reply (ans);
receive( );
Rx (packet);
reply( );
Tx(packet);

2.5.2消息传递

（3）信 箱

信箱由信箱头和包括若干信格的信箱体组成。每个信箱必须有自己唯一的标识符。利用信箱，允许向不知名的进程发送消息。信息始终安全地保存在信箱中，允许目标进程随时读取。信箱被广泛地用于多机系统和计算机网络中。信箱大多支持双向通信。

2.5.2消息传递

- 为了支持信箱通信，操作系统通常提供创建、撤消、获取信箱名等管理命令，以及发送信件与接受信件的命令。
- 信箱通信在实践上也存在一些应当明确的问题：

  A、信件的格式、类型如何？

  B、信格的大小如何？是可变的还是固定的？

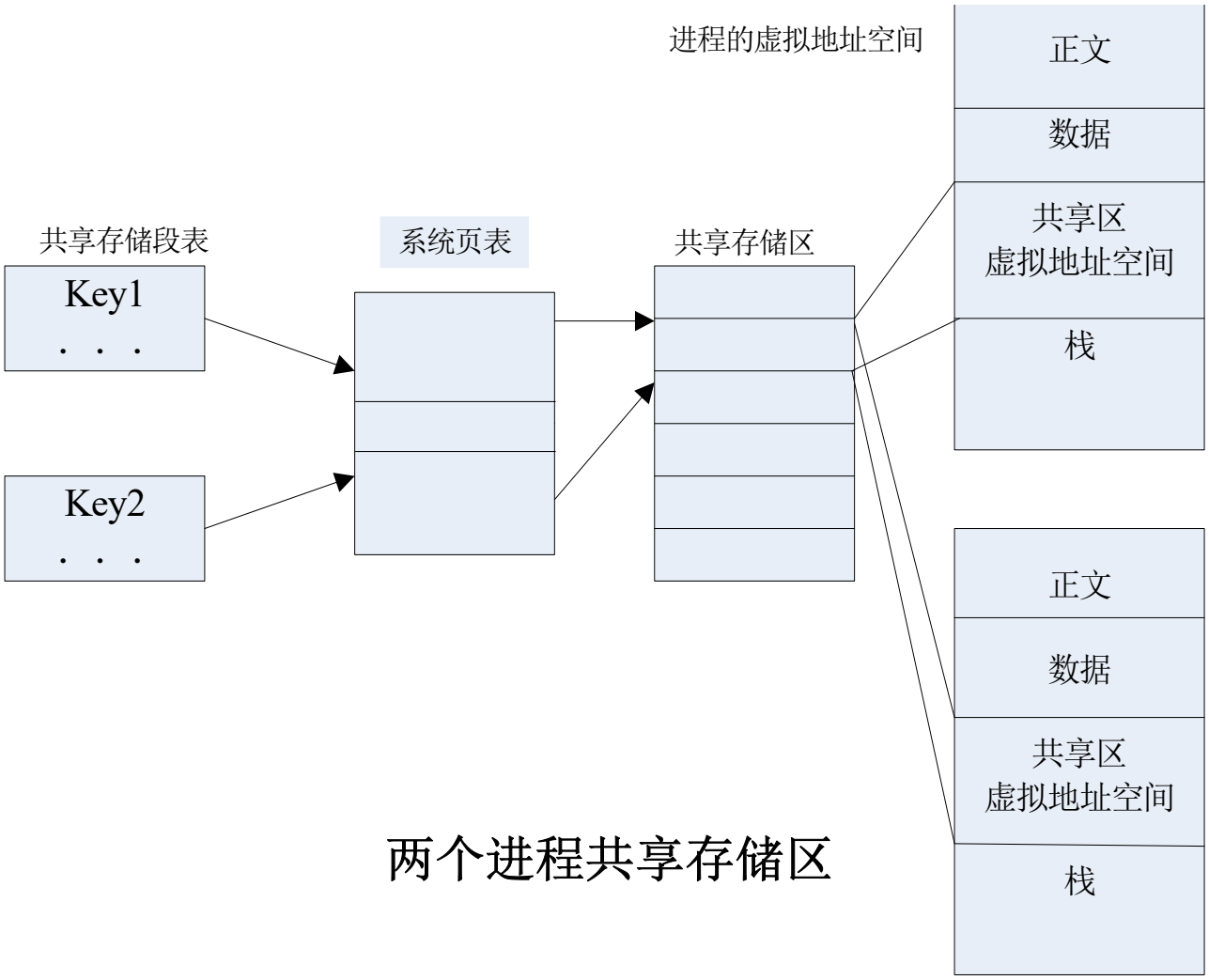
  C、信格数如何确定？是可变的还是固定的？

  D、信箱本身存放在操作系统空间还是存放在用户进程空间？信箱的所有权归谁？

2.5.3共享存储

- 把需要交换的信息发送到某一约定的存储区域，接受进程从该区域读取信息，从而实现2个或2个以上进程间的通信。这种通信方式称为共享存储。
- 共享存储区方式是目前系统进程通信中最高效的方式。
- 该机制可以将内存中的一个区域连入多个进程的虚拟地址空间。
- **UNIX, Windows 95, OS/2**等都采用这种方式。

2.5.3共享存储



两个进程共享存储区

2.5.3共享存储

- 涉及共享区的系统调用通常有创建、附接、断接、状态查询。

  (1)创建 shmget(key, size, flag)

  (2)附接 shmat(shmid, addr, flag)

  (3)断接 shmdt(viraddr)

  (4)状态查询 shmctl(shmid, cmd, buf)

- 共享存储区的好处在于为通信进程提供直接通信的手段，使得通信进程通过写(发送消息)、读(接受消息)对方的虚空间完成相互通信，通信的效率很高。

2.5.4管道通信

- 管道是一种信息流缓冲机构，它用于连接发送进程和接收进程，以实现它们之间的数据通信。管道不同一般的数据缓冲，它以先进先出（**FIFO**）的方式组织数据的传输。
- 在**UNIX**系统中，管道是以文件为基础的，利用共享文件进行通信的方式实现（有些书称之为文件通信）。

2.5.4管道通信

- 管道通信优点：是交换的信息量大，信息的保存期长。
- 管道通信缺点：是I/O操作的次数较多，同步和控制机构也较为复杂。
- 管道机制由管道文件创建、管道文件的读写以及读写操作时的同步机构3部分程序组成 。
- 管道通常用于有共同祖先的进程间的信息交换。

2.6进程调度

2.6.1 调度概念的引入

2.6.2 CPU调度程序

2.6.3调度准则

2.6.4 调度的策略

2.6.1 调度概念的引入

- 处理机管理是影响操作系统的主要功能之一。处理机管理的实现策略决定了操作系统的类型，其算法好坏直接影响整个系统的性能。
- 所谓进程调度，就是通过某种规则或算法从就绪(等待)进程队列中选出一个进程投入运行。
- 调度是一个基本的操作系统功能。**CPU** 调度是操作系统设计的核心问题。

2.6.2 CPU调度程序

1、CPU调度程序任务。

调度程序的任务是负责在CPU上切换进程。

当正运行的进程应该从CPU移除（改变到等待或阻塞状态）时，从处于等待状态的多进程中选择一个不同的进程。CPU被分配给被选择的进程，新的进程的状态从就绪改变为运行。

2.6.2 CPU调度程序

2、调度策略。

调度策略确定进程何时从CPU移除并且应当接着将CPU分配给某个等待的进程。

CPU调度决策可在如下四种情况下发生：

(1)当一个进程从运行状态切换到阻塞状态。

(2)当一个进程从运行状态切换到就绪状态。

(3)当一个进程从阻塞状态切换到就绪状态。

(4)当一个进程终止。

2.6.2 CPU调度程序

- 当出现第一种和第四种情况时，人们称调度方案是非抢占（**nonpreemptive**）的，与之对应，当出现另外两种情况时（第二和第三），调度方案是可抢占（**preemptive**）的。

2.6.2 CPU调度程序

- 非抢占调度方案，一旦CPU被分配给一个进程，该进程就将被执行到底，一直将CPU使用到进程结束或切换到等待状态再释放。

- 可抢占式调度方案，允许调度程序根据某种策略中止当前运行进程的执行，将其移入就绪队列，并选择另外一个进程投入运行。

2.6.2 CPU调度程序

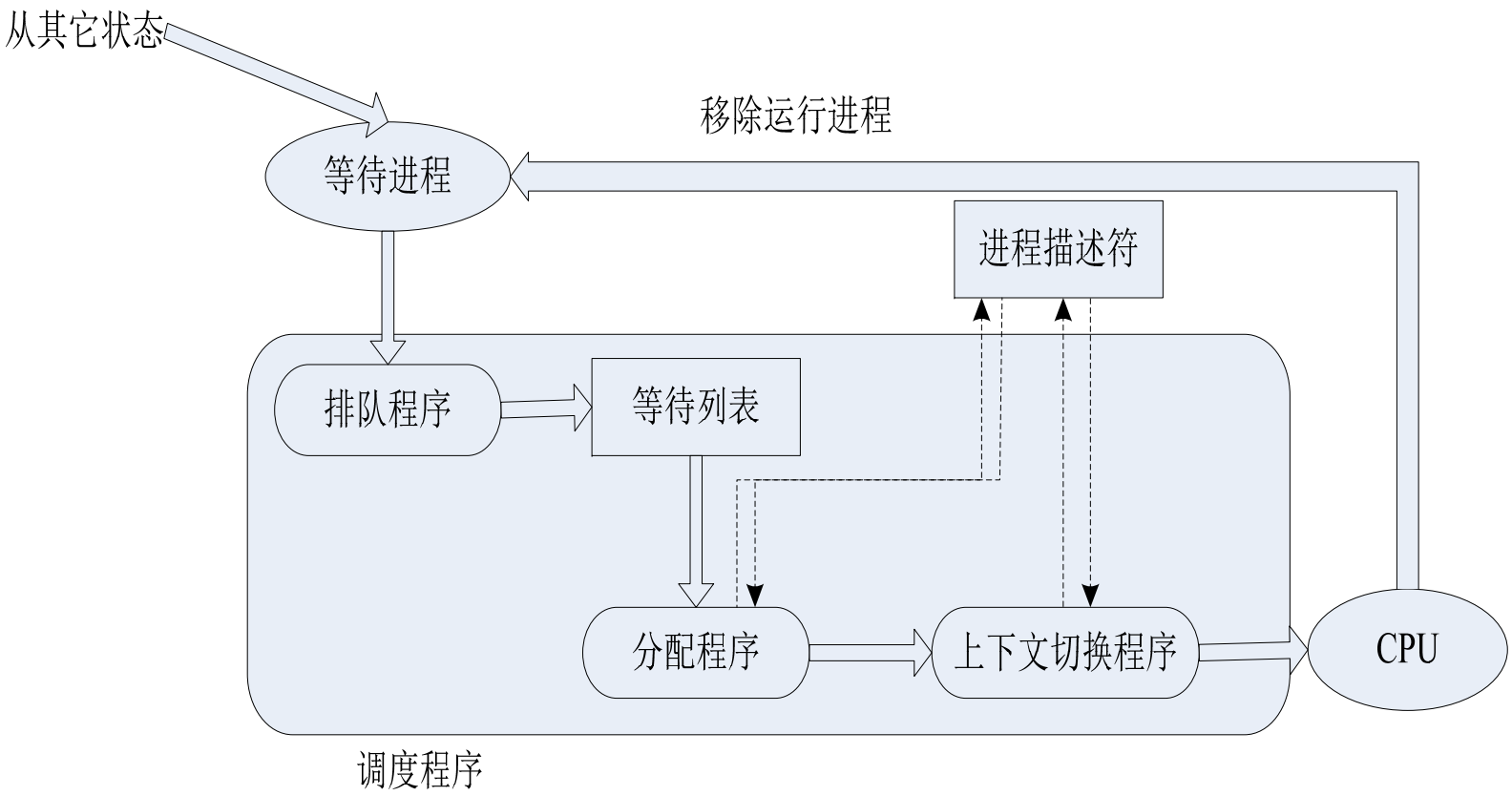
3、CPU调度程序主要模块。

- 三个主要部分：排队程序、分配程序、上下文切换程序。

（1）排队程序。

当一个进程改变到阻塞状态时，更新进程描述符并且排队程序通过指针关联进程描述符，该进程进入阻塞CPU的进程列表（称为阻塞列表）。

2.6.2 CPU调度程序



调度程序主要模块

2.6.2 CPU调度程序

（2）分配程序。

分配程序用来将CPU的控制交给由短期调度程序选择（根据上面提到的选择算法）的进程。

这部分程序要包括以下功能模块：

- 切换上下文。
- 切换到用户模式。
- 跳转到用户程序的合适位置以重新启动用户这个程序。

2.6.2 CPU调度程序

（3）上下文切换程序。

当调度程序切换CPU，从执行一个进程到执行另一个进程时，上下文切换程序保存从CPU移除进程的所有处理器寄存器（PC、IR、条件状态、处理器状态、和ALU状态）的内容到进程的PCB中。

2.6.2 CPU调度程序

4、性能

调度程序对于多道程序计算机的性能有很大的影响，因为其完全控制一个进程何时能分配有**CPU**。

- 关于一个进程一旦变为等待必须等待多久而产生的性能问题由调度策略确定。

2.6.3调度准则

1.影响调度算法选择的主要因素。

- 下面是一些在确定调度策略和算法时应该注意的主要因素：

（1）设计目标

（2）公平性

（3）均衡性

（4）综合平衡

（5）基于相对优先级

2.6.3调度准则

2.调度性能评价

① CPU使用率

② 吞吐量

③ 周转时间

④ 就绪等待时间

⑤ 响应时间

2.6.4调度策略

最常用的CPU调度策略。

- 先来先服务调度算法
- 短作业优先调度算法
- 优先级调度算法
- 轮转调度算法
- 多级队列调度算法

2.6.4调度策略

1、先来先服务调度算法

先来先服务（FCFS）先请求CPU 的进程先获得CPU。

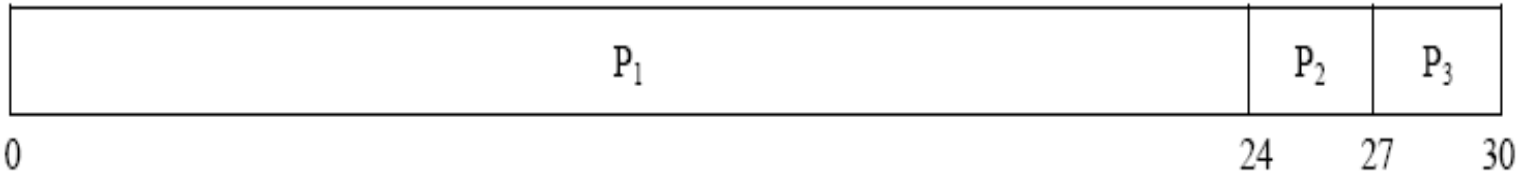
- 最简单的CPU 调度算法
- 对CPU繁忙型进程比较有利
- FCFS 调度算法是非抢占性的

2.6.4调度策略

如下的进程组合1

进程	占用时间
P1	24
P2	3
P3	3

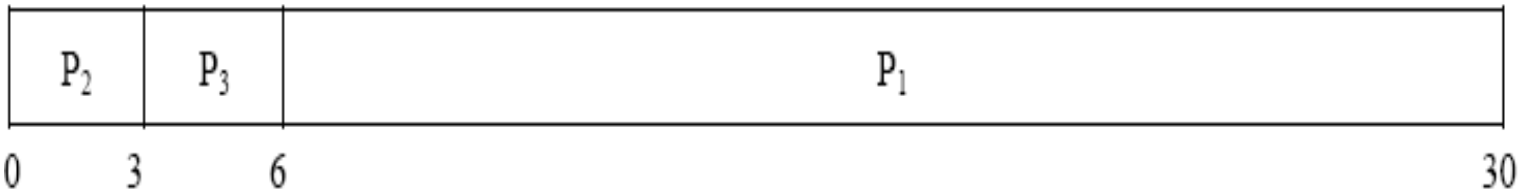
- 如果进程以P1 、P2、P3 的顺序到达，并且以FCFS 规则服务：



进程P1 的等待时间是0 毫秒，进程P2 是24 毫秒，P3 是27 毫秒。这样，平均等待时间是(0 + 24 + 27)/3 = 17 毫秒。

2.6.4调度策略

- 如果进程到达的顺序是**P2、P3、P1，**
  那么结果是甘特图2



现在的平均等待时间是**(6 + 0 + 3)/3 = 3** 毫秒。平均等待时间很明显的减少了。

FCFS 策略下的平均等待时间通常不是最小的，而且如果进程的**CPU** 占用 时间有明显的变化时平均时间也会有很明显的变化。

2.6.4调度策略

- **FCFS** 调度算法对**CPU**繁忙型进程比较有利，而对于**I/O** 繁忙型进程则不利。
- **FCFS** 调度算法是非抢占性的。一旦**CPU** 被分配给一个进程，该进程将持有**CPU** 直到它释放**CPU**（通过终止或请求**I/O**）。

2.6.4调度策略

2、短作业优先调度算法

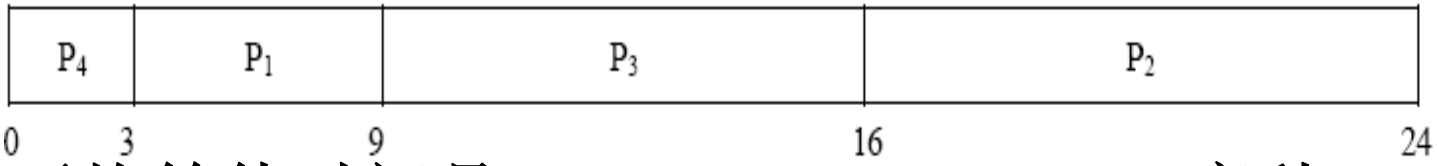
短作业优先（**Shorttest-job-first，SJF**）调度算法它将每个进程与其下一个**CPU**占用时间长度相关联。当**CPU**可用时，它将优先被分配给具有最短后续**CPU**占用的进程。这种算法假定事先知道了每个进程下次运行的**CPU** 占用长度。如果两个进程的下一个**CPU** 占用相同，就使用**FCFS** 调度。

2.6.4调度策略

考虑如下表所示的一组进程组合2

进程	占用时间（毫秒）
P1	6
P2	8
P3	7
P4	3

利用SJF 调度，我们将依照甘特图3来调度这些进程：



平均等待时间是(3 + 16 + 9 + 0)/4 = 7 毫秒。

如果使用FCFS 调度策略，那么平均等待时间是10.25 毫秒 。

2.6.4 调度策略

- **SJF** 调度算法为指定的进程组给出了最小的平 均等待时间
- 短作业优先也是算法是非抢占性的

- 最短剩余时间优先调度（**Shortest-Remaining-Time-First，SRTF**）

抢占性短作业优先算法

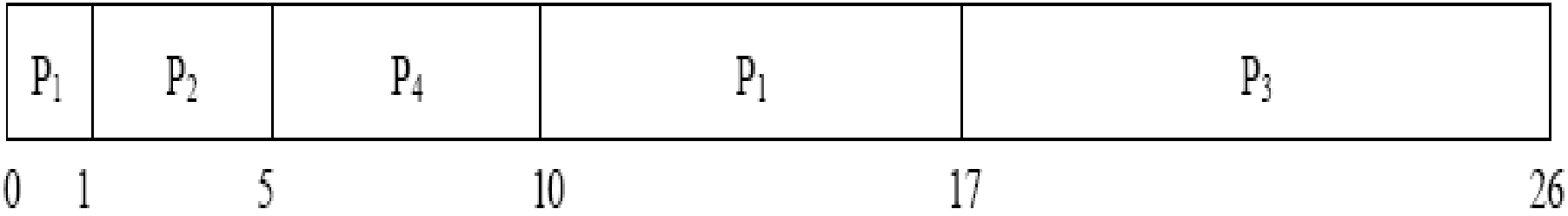
2.6.4 调度策略

下面给出一个例子说明抢占式**SJF**调度，考虑下表所示四个进程，给定如下的**CPU** 占用长度：

进程	到达时间	占用时间
P1	**0**	**8**
P2	**1**	**4**
P3	**2**	**9**
P4	**3**	**5**

2.6.4 调度策略

- 那么按照抢占式**SJF** 调度会产生如下图的结果：



这个例子的平均等待时间是**((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5** 毫秒。而采用非抢占式**SJF** 调度的平均等待时间为**7.75** 毫秒。

2.6.4 调度策略

3.优先级调度算法

- 优先级调度算法是从就绪队列中选出优先级别最高的进程。让它占用**CPU**运行
- 静态优先级：静态优先级调度算法是指在创建进程时就确定下来的，而且在进程的整个运行期间其优先级是维持不变的
- 动态优先级：动态优先级是随着进程的推进而不断变化的

2.6.4 调度策略

- 静态优先级调度算法
  - 静态优先级调度算法易于实现，系统的开销也小。
  - “饥饿”现象，某些优先级别比较低的进程可能 无限期等待**CPU**调用，如果高级别的进程很多，形成一个稳定的进程队列，可能使低级别进程任何时候也得不到**CPU**资源，会“饿死了”。

2.6.4 调度策略

- 动态优先级调度算法
  - 为解决“饥饿”现象，而产生的一种解决方案。
  - 解决“饥饿”现象的一种方法就是“老化”处理。

2.6.4 调度策略

- 高响应比优先调度算法（HRN）

最高响应比作业优先算法是对FCFS方式和SJF方式的一种综合平衡响应比R定义为系统对作业的响应时间与作业要求运行时间的比值

R＝响应时间 / 要求运行时间

＝(作业等待时间＋需运行时间)/ 需运行时间

＝1＋已等待时间 / 需运行时间

＝1＋W/T

2.6.4 调度策略

- 响应比R不仅是要求运行时间的函数，而且还是等待时间的函数。
- 由于R与要求运行时间成反比，故对短作业是有利的，另一方面，因R与等待时间成正比，故长作业随着其等待时间的增长，也可获的较高的相应比。这就克服了短作业优先数法的缺点，既照顾了先来者，又优待了短作业，是上述两种算法的一种较好的折中。

2.6.4 调度策略

作业	进入时刻	运行时间	开始时刻	完成时刻	周转时间	带权周转
1	8.00	2.00	8.00	10.00	2.00	1.00
2	8.50	0.50	10.10	10.60	2.10	4.20
3	9.00	0.10	10.00	10.10	1.10	11.00
4	9.50	0.20	10.60	10.80	1.30	6.50

平均周转时间1.625h

带权周转时间 5.675h

2.6.4 调度策略

- 优先级调度算法中，对应也有两种不同的处理方式
  - **1**、非抢占式优先级法。当前占用**CPU**的进程一直运行下去，直到完成任务或因为等待某些资源而主动让出**CPU**，系统才让另外一个优先级高的进程占用**CPU**。
  - **2**、抢占式优先级法。在当前进程运行过程中，一旦有另外一个优先级更高的进程出现在就绪队列中，进程调度程序就停止当前进程运行，强行将**CPU**分配给此优先级高的进程。

2.6.4 调度策略

4.轮转（Round-Robin，RR）调度算法

- 将所有就绪进程按照先入先出的原则排成队列，新来的进程加到就绪队列的末尾，但是轮转调度添加了进程间的抢占切换。
- 轮转调度算法特点：
  - 1、RR 策略的平均等待时间通常会相当长
  - 2、RR调度算法尤其适用于分时系统
  - 3、RR 算法的性能依赖于时间片的大小

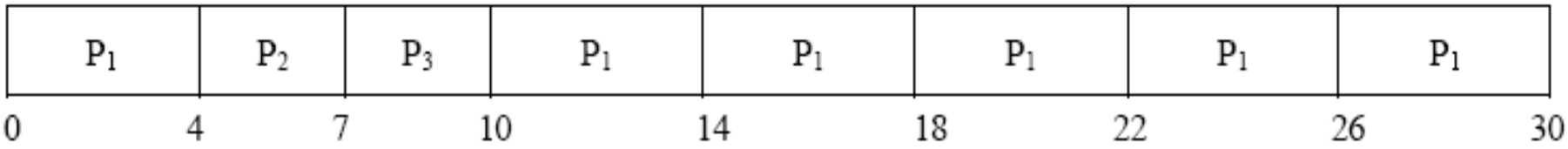
2.6.4 调度策略

- **RR** 算法定义了一个小的时间单元，被称为时间片。一个时间片通常在**10** 毫秒到**100** 毫秒之间。
- 把就绪队列作为循环队列对待。**CPU** 调度程序环绕这个就绪队列，将**CPU** 分配到每个进程，每隔一个时间片转换一次。

2.6.4 调度策略

进程	占用时间
P1	24
P2	3
P3	3

如果我们使用4 毫秒的时间片,结果， RR 调度得到甘特图6：



平均等待时间是17/3 = 5.66 毫秒

2.6.4 调度策略

- 时间片的大小通常由以下几个因素决定：

（1）时间片大小和上下文切换有关。

（2）系统的响应时间。

（3）进程周转时间。

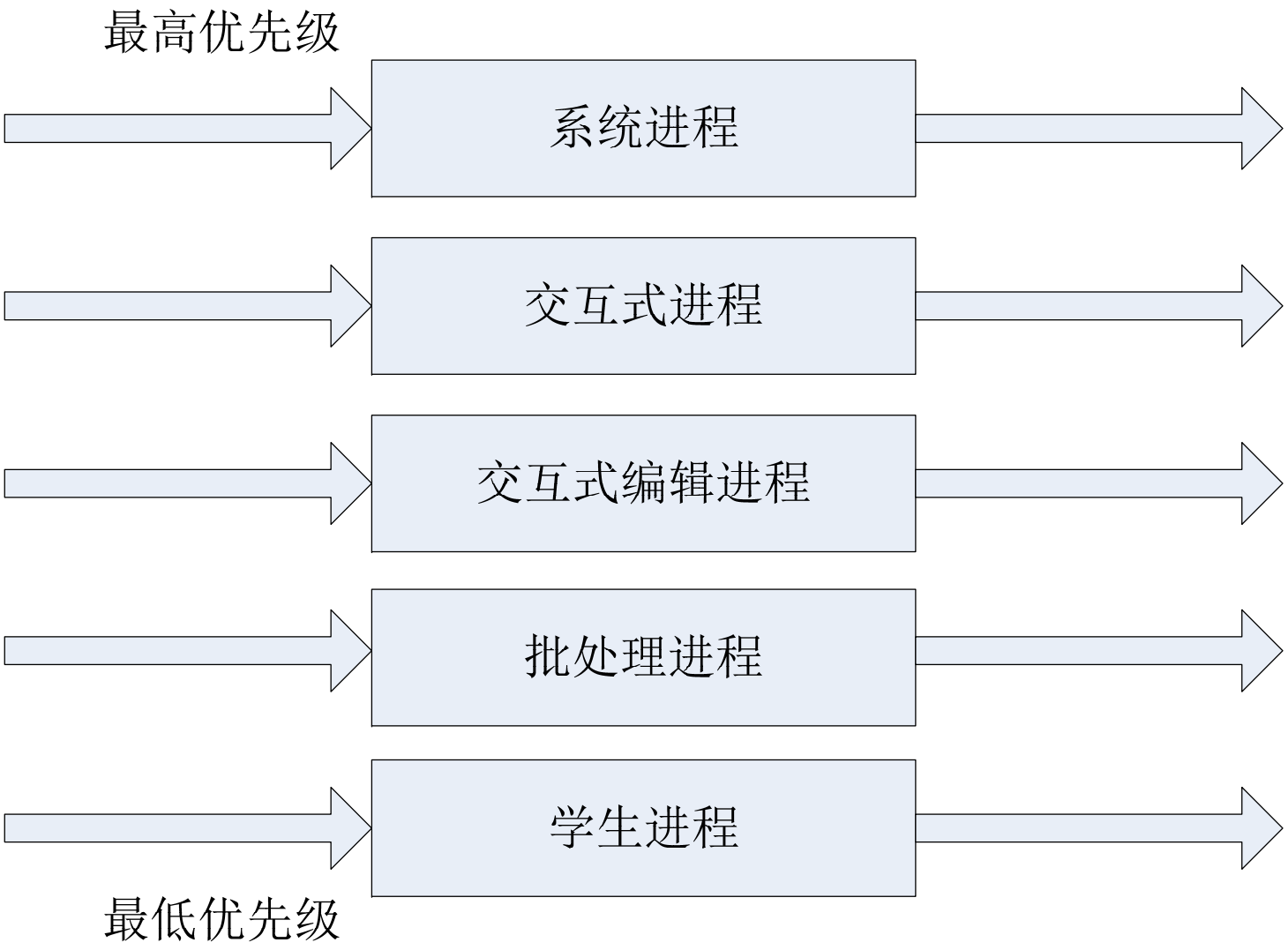
（4）时间片大小和CPU主频有关。

2.6.4 调度策略

5.多级队列(Multilevel-Queue,MQ)调度算法

- 把就绪队列划分为多个独立的队列
- 将进程分成前台或后台
- 每个队列有自己的调度策略
- 每个队列之间可以按优先级调度，也可以按时间片轮转调度。

2.6.4 调度策略



2.7 死锁

- 2.7.1 死锁的背景
- 2.7.2 产生死锁的必要条件
- 2.7.3 资源使用方式
- 2.7.4 死锁的处理方法
- 2.7.5 预防死锁
- 2.7.6 避免死锁
- 2.7.7 死锁的检测与解除
- 作业：40、42、43

2.7.1 死锁的背景

- 资源共享问题产生
- 两个或多个进程无限期地等待永远都不会发生的条件，系统处在停滞状态
- 死锁问题是荷兰学者**E.W.Dijkstra**在**1965**年研究银行家算法时首次提出的
- 死锁不仅发生在硬件资源上，由于设计不当，也有可能发生在软件资源的使用上。

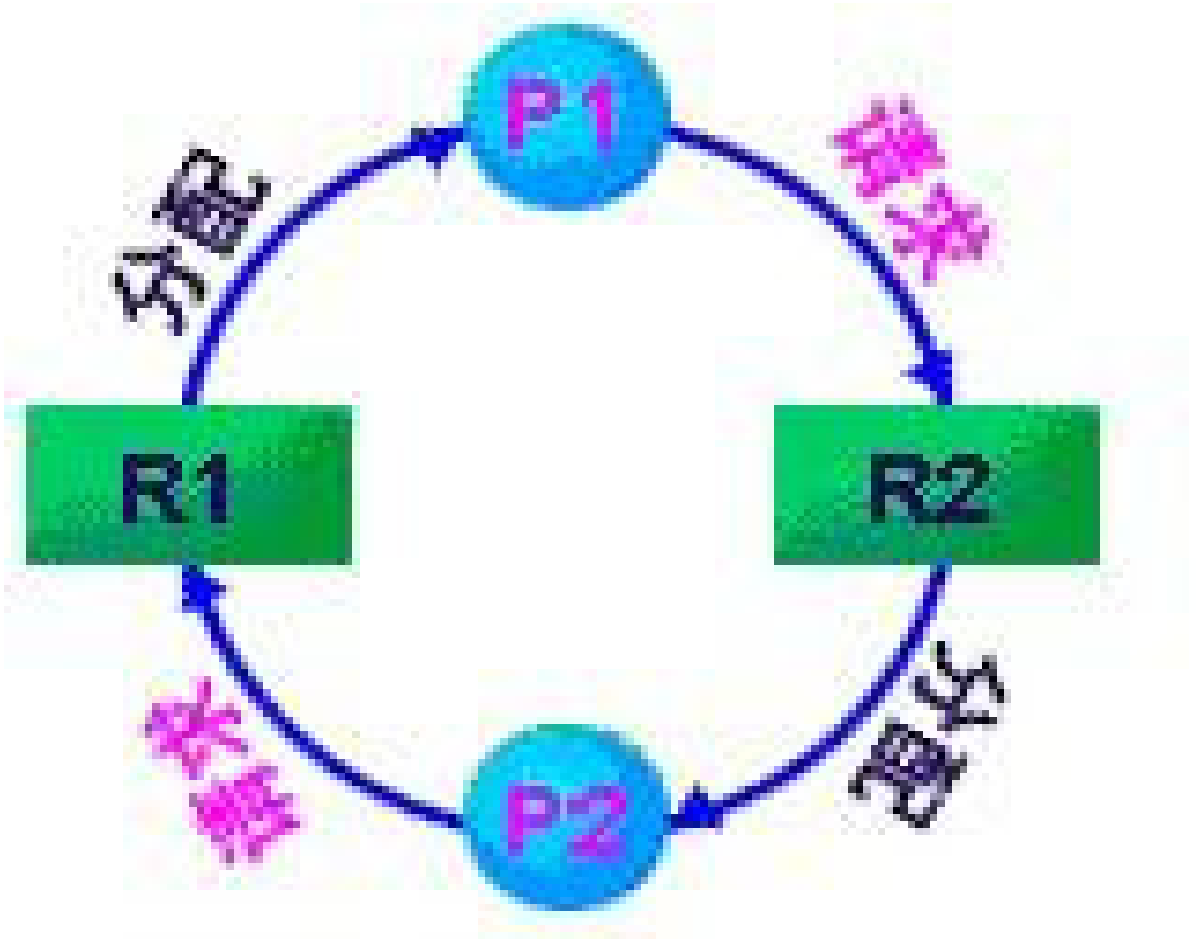
2.7.2 产生死锁的必要条件

- 在系统中，如果如下四个条件同时成立，那么死锁就会发生：
  - （1）互斥条件
  - （2）持有并等待条件
  - （3）不可抢占条件
  - （4）循环等待条件

2.7.2 产生死锁的必要条件

- 资源分配图
- 该图由一组有向的顶点**V**和一组有向边**E**组成。顶点**V**分为两个不同类型的节点：由系统中所有的活动进程组成的节点**P = { P0, P1, …, Pn }**和由系统中所有的资源组成的节点**R = { R0, R1, …, Rm }**。有向边**Pi → R**被称为请求边（**request edge**）；有向边**Rj → Pi**被称为分配边（**assignment edge**）。

资源分配图



2.7.3 资源使用方式

- 在通常操作方式下，进程只能通过如下的顺序使用资源：
  - 1. 请求：如果请求不能够立刻得到允许（例如，所请求的资源正为另一个进程所用），那么发出请求的进程必须等待，直到它可以获得该资源。
  - 2. 使用：进程可以对资源进行操作（例如，如果资源是打印机，那么该进程可以在这个打印 机上打印）。
  - 3. 释放：进程释放它以前请求且分到的资源。

2.7.4 死锁的处理方法

- 死锁的处理方法
  - 1、预防死锁
  - 2、避免死锁
  - 3、检测与解除死锁
  - 4、将死锁忽略不计，认为死锁不可能在系统内发生。

2.7.5 预防死锁

- 如果我们能够确保产生死锁的四个必要条件有一个不成立，就可以预防死锁。
  - 1、互斥条件
  - 2、持有并等待条件
  - 3、不可抢占条件
  - 4、循环等待条件

2.7.5 预防死锁

- 资源一次性分配：（破坏请求和保持条件）
- 可剥夺资源：即当某进程新的资源未满足时，释放已占有的资源或剥夺等待进程的资源（破坏不可剥夺条件）
- 资源有序分配法：做法：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反(破坏环路等待条件)

2.7.6 避免死锁

- 死锁预防算法通过对约束资源请求来预防死锁的发生。这种约束确保了至少有一个死锁的必要条件不会发生，从而达到预防死锁的目的。这种方法人们常称之为静态策略，显而易见，采用静态分配策略来预防死锁的也造成了较低的设备利用率并降低了系统吞吐率。
- 避免死锁方法

2.7.6 避免死锁

- **死锁避免定义**:在系统运行过程中，对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配。

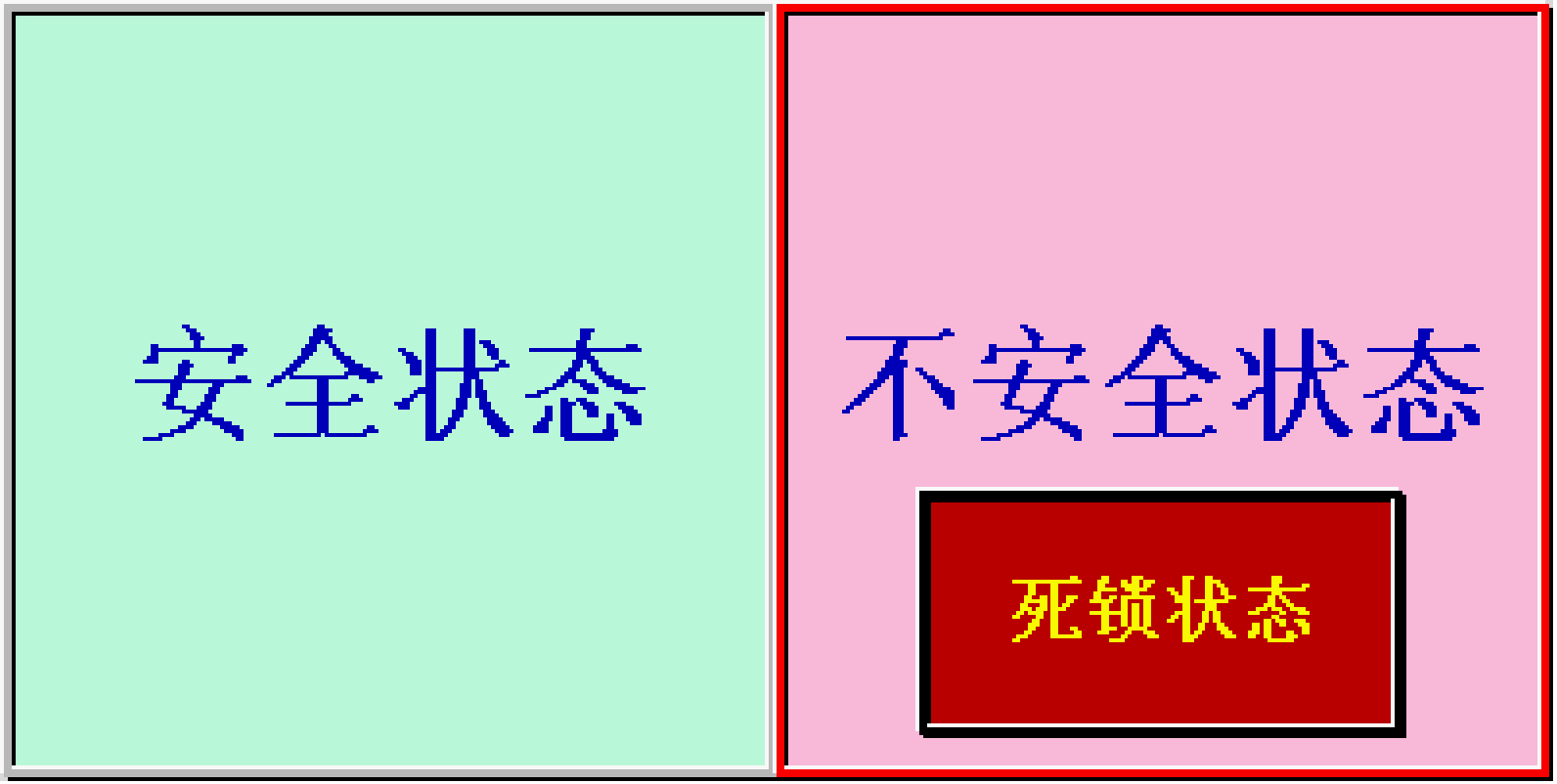
2.7.6 避免死锁

1、安全状态

- 如果系统能够以某些顺序为每个进程分配资源（可满足进程的最大需求）并依然可以避免死锁，那么系统的状态是安全的。更严格的说，系统只有在存在一个安全序列时才处于安全状态。对于进程序列**< P1, P2, …, Pn >**，如果每个**Pi** 对资源的请求都能够由当前有效的资源加上**Pj**（**j < i**）释放的资源来满足，那么这个序列是安全的。

2.7.6 避免死锁

- 不安全状态:不存在一个安全序列，不安全状态不一定导致死锁



2.7.6 避免死锁

我们通过一个例子来说明安全性。假定系统中有三个进程P_0、P_1和P_2，共有12台磁带机。进程P_0总共要求10台磁带机，P_1和P_2分别要求4台和9台。假设在T_0时刻，进程P_0、P_1和P_2已分别获得5台、2台和2台磁带机，尚有3台空闲未分配，如下表所示：

进 程	最 大 需 求	已 分 配	可 用
P_0	10	5	3
P_1	4	2	
P_2	9	2	

安全序列{P1 ， P0 ， P2}

2.7.6 避免死锁

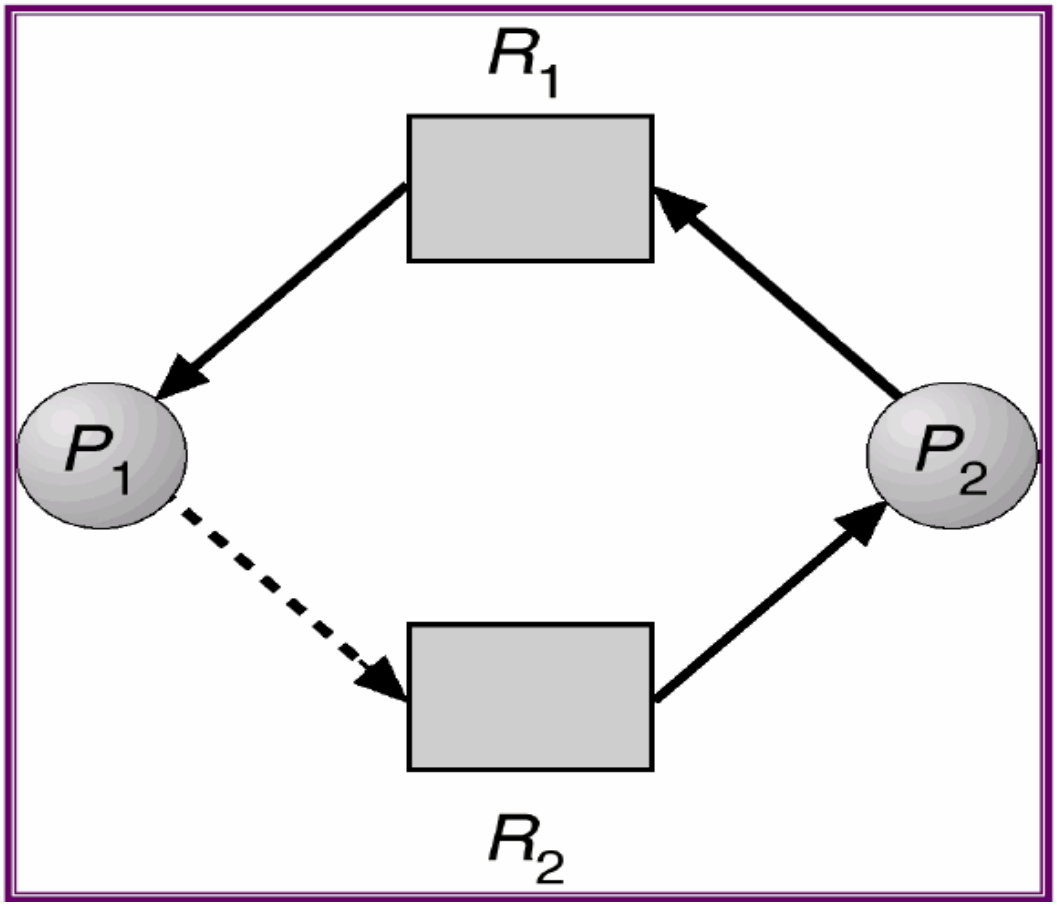
如果不按照安全序分配资源，则系统可能会由安全状态进入不安全状态。例如，在T_0时刻以后，P_2又请求1台磁带机，若此时系统把剩余3台中的1台分配给P_2，则系统便进入不安全状态。 因为，此时也无法再找到一个安全序列，例如，把其余的2台分配给P_1，这样，在P_1完成后只能释放出4台，既不能满足P_0尚需5台的要求，也不能满足P_2尚需6台的要求，致使它们都无法推进到完成，彼此都在等待对方释放资源，即陷入僵局，结果导致死锁。

2.7.6 避免死锁

2、资源分配图算法

- 如果资源分配系统的每种资源只有一个，可以使用资源分配图方法有所不同的新的资源分配图来避免死锁。
- 除请求边和分配边之外，再加上一种新边，称为声明边。声明边**Pi** → **Rj** 表明：进程**Pi** 可能会在稍后某些时候请求资源**Rj**。这条边与请求边的方向一致，但是用虚线表示。当进程**Pi** 请求资源**Rj** 时，声明边**Pi** → **Rj** 转换成请求边。类似的，当进程**Pi**，释放资源**Rj** 时，分配边**Rj** → **Pi** 转换成声明边**Pi** → **Rj**。注意：在系统中必须要预先声明资源。更确切的说，在进程**Pi** 开始执行之前，它所有的声明边必须要出现在资源分配图中。

2.7.6 避免死锁



资源分配图中的一个不安全状态

2.7.6 避免死锁

3、银行家算法

- 银行家算法中的数据结构

(1) 可利用资源向量**Available**。这是一个含有***m***个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果**Available［j］=K**，则表示系统中现有**Rj**类资源***K***个。

银行家算法

(2) 最大需求矩阵Max。这是一个$n \times m$的矩阵，它定义了系统中n个进程中的每一个进程对m类资源的最大需求。如果Max［i,j］=K，则表示进程i需要R_j类资源的最大数目为K。

银行家算法

(3) 分配矩阵**Allocation**。这也是一个$n \times m$的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果**Allocation［i,j］=K**，则表示进程**i**当前已分得R_j类资源的数目为K。

银行家算法

(4) 需求矩阵**Need**。这也是一个$n \times m$的矩阵，用以表示每一个进程尚需的各类资源数。如果**Need［i,j］=*K***，则表示进程**i**还需要**Rj**类资源***K***个，方能完成其任务。

Need［i,j］=Max［i,j］-Allocation［i,j］

银行家算法

- 银行家算法

设$Request_i$是进程P_i的请求向量，如果$Request_i$［j］=*K*，表示进程P_i需要*K*个R_j类型的资源。当P_i发出资源请求后，系统按下述步骤进行检查：

(1) 如果$Request_i$［j］≤Need［i,j］，便转向步骤2；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。

(2) 如果$Request_i$［j］≤Available［j］，便转向步骤(3)；否则，表示尚无足够资源，P_i须等待。

银行家算法

(3) 系统试探着把资源分配给进程P_i，并修改下面数据结构中的数值：

Available［j］：=Available［j］-Request$_i$［j］；

Allocation［i,j］：=Allocation［i,j］+Request$_i$［j］；

Need［i,j］：=Need［i,j］-Request$_i$［j］；

银行家算法

(4) 系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式将资源分配给进程**Pi**，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程**Pi**等待。

银行家算法

- 安全性算法

(1) 设置两个向量：① 工作向量**Work:** 它表示系统可提供给进程继续运行所需的各类资源数目，它含有*m*个元素，在执行安全算法开始时，**Work：=Available;** ② **Finish:** 它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做**Finish［i］：=false;** 当有足够资源分配给进程时， 再令**Finish［i］：=true**。

银行家算法

(2) 从进程集合中找到一个能满足下述条件的进程：

① Finish［i］=false; ② Need［i,j］≤Work［j］； 若找到， 执行步骤(3)， 否则，执行步骤(4)。

(3) 当进程P_i获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

Work［j］：= Work［i］+Allocation［i,j］;

Finish［i］：= true;

go to step 2;

(4) 如果所有进程的Finish［i］=true都满足， 则表示系统处于安全状态；否则，系统处于不安全状态。

银行家算法

- 银行家算法例子

  假定系统中有五个进程{P0，P1，P2，P3，P4}和三种类型的资源{A，B，C}，每一种资源的数量分别为10、5、7，在T0时刻的资源分配情况如图所示

	MAX	ALLO	NEED	AVAI
P0	7 5 3	0 1 0	7 4 3	3 3 2
P1	3 2 2	2 0 0	1 2 2	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

现在是安全的吗？

银行家算法

1. **T0时刻的安全序列：** **P1-P3-P4-P0-P2**

	MAX	ALLO	NEED	AVAI	Work	Finish
P0	7 5 3	0 1 0	7 4 3	3 3 2	7 5 5	True
P1	3 2 2	2 0 0	1 2 2		5 3 2	True
P2	9 0 2	3 0 2	6 0 0		10 5 7	True
P3	2 2 2	2 1 1	0 1 1		7 4 3	True
P4	4 3 3	0 0 2	4 3 1		7 4 5	True

银行家算法

2. P1请求资源：

P1发出请求向量Request1（1，0，2），系统按银行家算法进行检查：

（1）Request1（1，0，2）<=Need1(1,2,2)

（2）Request1（1，0，2）<=Available(3,3,2)

（3）系统先假定可为P1分配资源，并修改Available, Allovation1和Need1向量，由此形成的资源变化情况，如图中圆括号所示

银行家算法

2. **P1请求（1,0,2）**

	MAX	ALLO	NEED	AVAI	Work	Finish
P0	7 5 3	0 1 0	7 4 3	2 3 0	7 5 5	True
P1	3 2 2	3 0 2	0 2 0		5 3 2	True
P2	9 0 2	3 0 2	6 0 0		10 5 7	True
P3	2 2 2	2 1 1	0 1 1		7 4 3	True
P4	4 3 3	0 0 2	4 3 1		7 4 5	True

有安全序列**P1-P3-P4-P0-P2**，可以实际进行分配

银行家算法

3．P4请求资源：

P4发出请求向量Request4（3，3，0），系统按银行家算法进行检查：

（1）Request4（3，3，0)<=Need4(4,3,1)

（2）Request4(3, 3, 0)不小于或等于Available(2,3,0),让P4等待。

银行家算法

4. P_0请求资源：

P_0发出请求向量$Request_0$（0，2，0），系统按银行家算法进行检查：

（1）$Request_0$（0，2，0）<=$Need_0$(7,4,3)

（2）$Request_0$（0，2，0）<=Available(2,3,0)

（3）系统先假定可为P_0分配资源，并修改有关数据。如下图所示。

银行家算法

	Allocation A B C	Need A B C	Available A B C
P_0	0 3 0	7 2 3	2 1 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

进行安全性检查：

可用资源**Available(2,1,0)**已不能满足任何进程的需要，故系统进入不安全状态，此时系统不分配资源。

2.7.7 死锁的检测与解除

1、死锁的检测

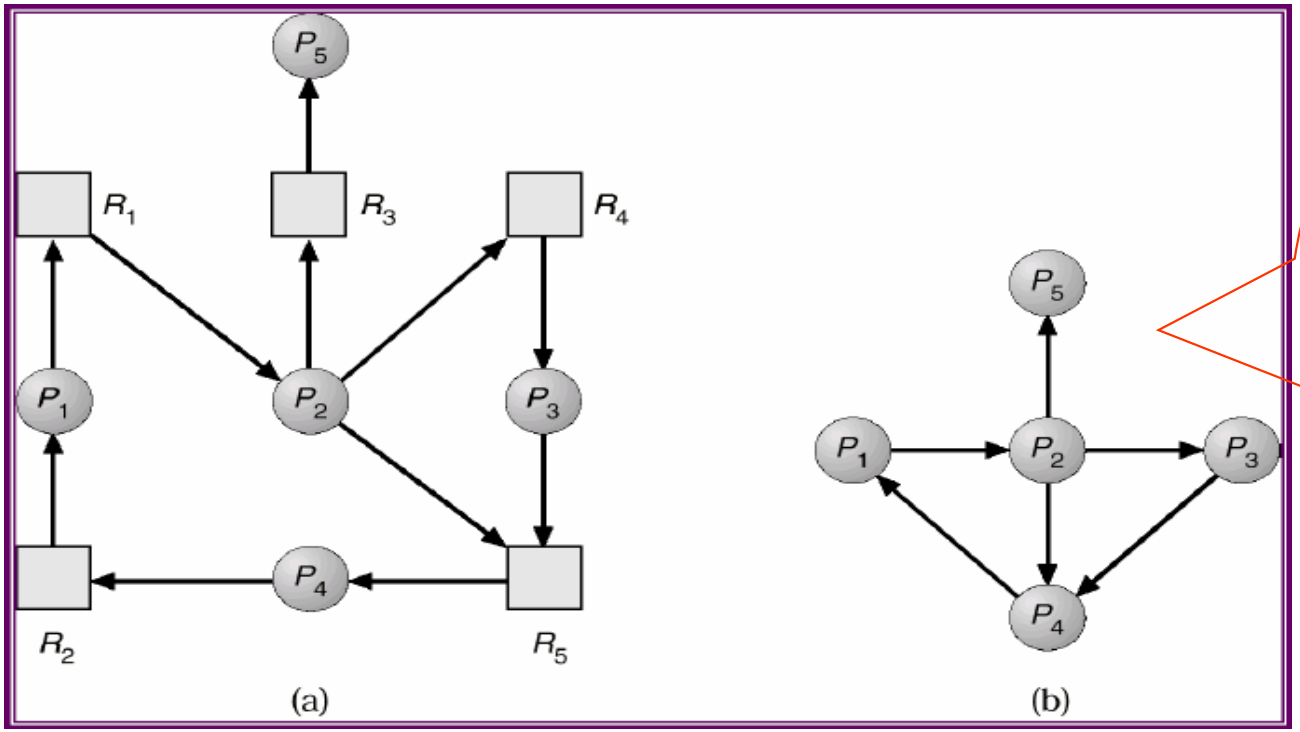
如果系统没有采用死锁预防或死锁避免算法那么就有可能发生死锁。在这种环境下，系统必须提供：

- 用于检测系统状态以确定是否发生死锁的算法。
- 从死锁中恢复的算法。

2.7.7 死锁的检测与解除

- 资源单一

  如果每种资源只有一个，修改资源分配图，从资源分配图中移除资源节点和相应的边，形成等待图。



如果等待图有环，说明系统处于不安全状态，有可能死锁也可能不会死锁。

2.7.7 死锁的检测与解除

- 多个资源

1. 令**Work** 和**Finish** 的长度各为**m** 和**n**。初始化**Work := Available** 。对于**i = 1, 2, ..., n**，如果**Allocationi ≠ 0**，那么**Finish[i] :=false**；否则，**Finish[i] := true**。

2.7.7 死锁的检测与解除

2. 找到这样的一个下标i：

a. Finish[i] =false.

b. Requesti ≤ Work.

如果不存在，跳到第**4** 步。

3. Work := Work + Allocationi

Finish[i] := true

跳到第**2** 步。

2.7.7 死锁的检测与解除

4. 对于i，1 < i < n，如果Finish[i] = false，那么系统处于死锁状态。此外，如果Finish[i] =false，那么进程Pi 死锁。

2.7.7 死锁的检测与解除

- 死锁的检测应用：
  - 考虑因素：
    - 1、可能多长发生一次死锁
    - 2、死锁发生会影响多少进程

2.7.7 死锁的检测与解除

- 死锁的解除
  - 1、进程终止
  - 2、抢占资源

2.7.7　死锁的检测与解除

- 进程终止两种方案
  - 1、异常终止所有的死锁进程
  - 2、每次异常终止一个进程直到消除死锁循环

2.7.7 死锁的检测与解除

- 选择终止进程的因素：
  - 1、进程的优先级是什么？
  - 2、进程计算了多长时间？进程在完成任务前还要计算多长时间？
  - 3、进程使用的资源数量和种类（例如，是否资源被抢占）？
  - 4、进程尚需多少资源？
  - 5、需要终止多少进程？
  - 6、进程是交互式的还是分批的

2.7.7 死锁的检测与解除

- 抢占资源
- 用抢占来解除死锁，解决三件事
  - 1、选择一个进程
  - 2、回滚
  - 3、如何避免“饥饿”