

Mac OS

VxWorks

OS2

UNIX

LINUX

WINDOWS



操作系统结构分析

第3章同步、通信与死锁

南京邮电大学

计算机学院 信息安全系

曹晓梅

手机: 189-0518-4599

QQ: 757375652

email: caoxm@njupt.edu.cn

□ 并发的进程间为何要对话？

- 竞争或协作
- 掌握进程同步的机制，解决经典同步问题

□ 并发进程间的对话方式有主要哪些？

- ★• 信号量
- 共享文件
- 共享存储区
- 消息传递

□ 并发的进程间如果沟通不好的话会咋样？

- 答：会死锁！
- 掌握如何解决死锁问题

重点

- 实现进程同步和互斥的各种机制
- 死锁避免、检测和解除方法

难点

- 解决经典同步问题的方法

通过信号量机制解决：
I 生产者-消费者问题
II 读者-写者问题

- 死锁避免方法

银行家算法

内容纲要

Contents Page



1 进程同步

2 进程通信

3 死锁

1.1 并发进程

- I 并发进程之间的关系
- II 异步产生的问题
- III 进程的同步与互斥

1.2 临界区管理

1.3 信号量与P、V操作



此后课程中将不再刻意区分并发进程和并发线程，它们都可以理解为运行的实体和调度的单位。

并发进程之间的关系(1/3)

□ 无关进程

- 并发进程分别在自己的变量集合上运行
- 例如： word进程和foxmail进程
- 无关进程可能会**竞争**使用独占资源

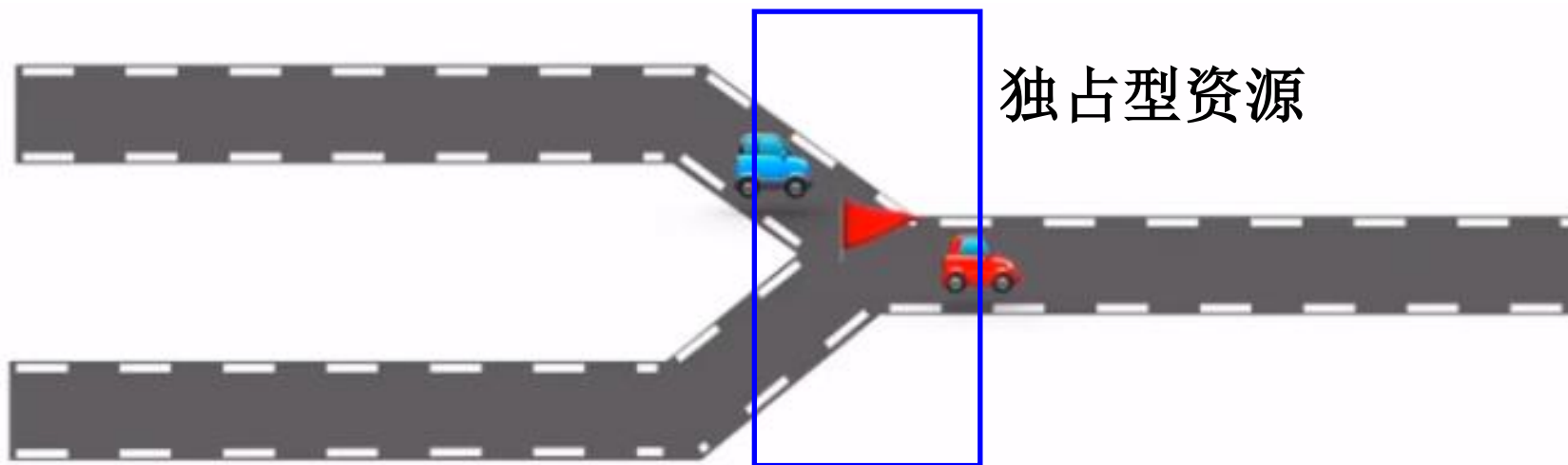
□ 交互进程

- 并发进程执行过程中需要共享和交换数据
- 例如： 银行交易服务器上receiver进程和handler进程
- 交互的并发进程之间存在**竞争**和**协作**两种关系

并发进程之间的关系(2/3)

□ 竞争关系(race)

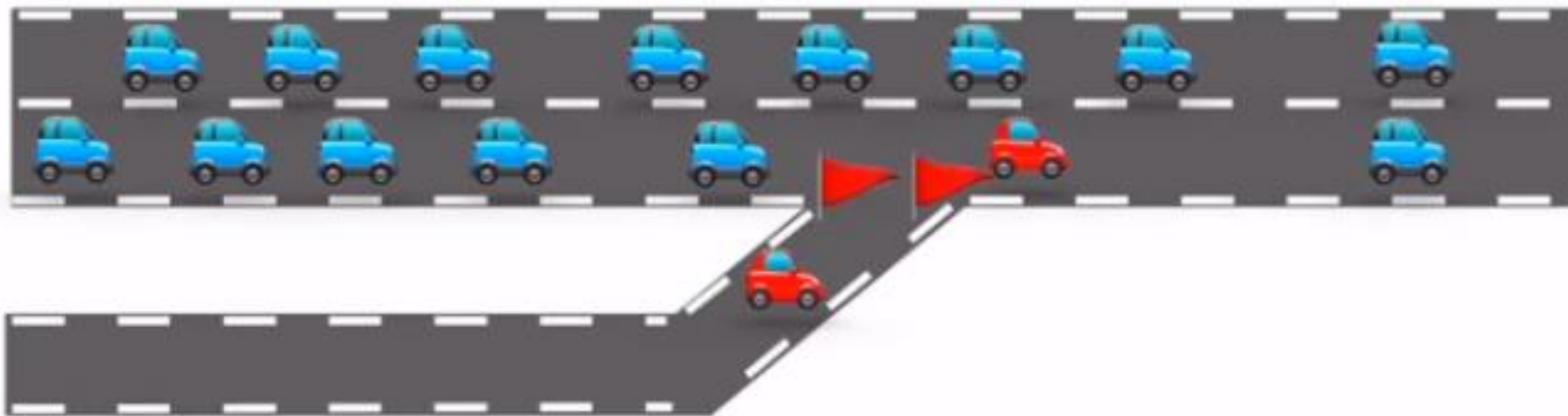
现实世界中的异步和同步



并发进程之间的关系(3/3)

□ 协作关系(cooperation)

现实世界中的异步和同步



主干道每过两辆车才允许匝道通过一辆车

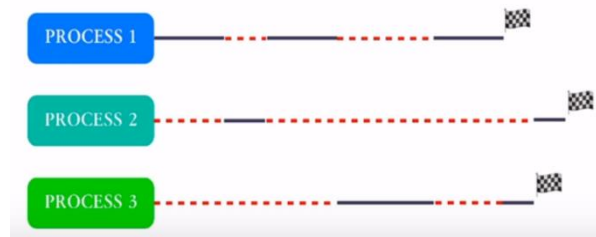
1.1 并发进程

并发执行的程序中须有一定机制避免竞争条件的发生！

异步(Asynchronous)产生的问题

□ 引发竞争条件(race condition)

- 多个进程并发操作同一个共享数据，导致执行结果依赖于特定的进程执行顺序。



例：Ti是订票终端，共享变量 $x=2$ 代表剩余票数，为所有终端共享

T_1 :

...

$T=x;$ ① ①

if $T \geq 1$ then ②③

$x = T - 1;$

...

T_2 :

...

$T=x;$ ③ ②

if $T \geq 1$ then ④

$x = T - 1;$

...

剩余票数最终结果不确定！可能是0，也可能是1.

进程的同步与互斥(1/2)

□ 概念

- 互斥(mutual exclusion->mutex)

并发进程间因**竞争独占型资源**而引起的，为一个进程等待另一个进程已经占有的、必须互斥使用的资源时的一种制约关系。

间接制约

- 同步(synchronization)

并发进程之间需要**协调完成同一任务**时引起的一种关系，为一个进程等待另一个进程向它直接发送消息或数据时的一种制约关系。

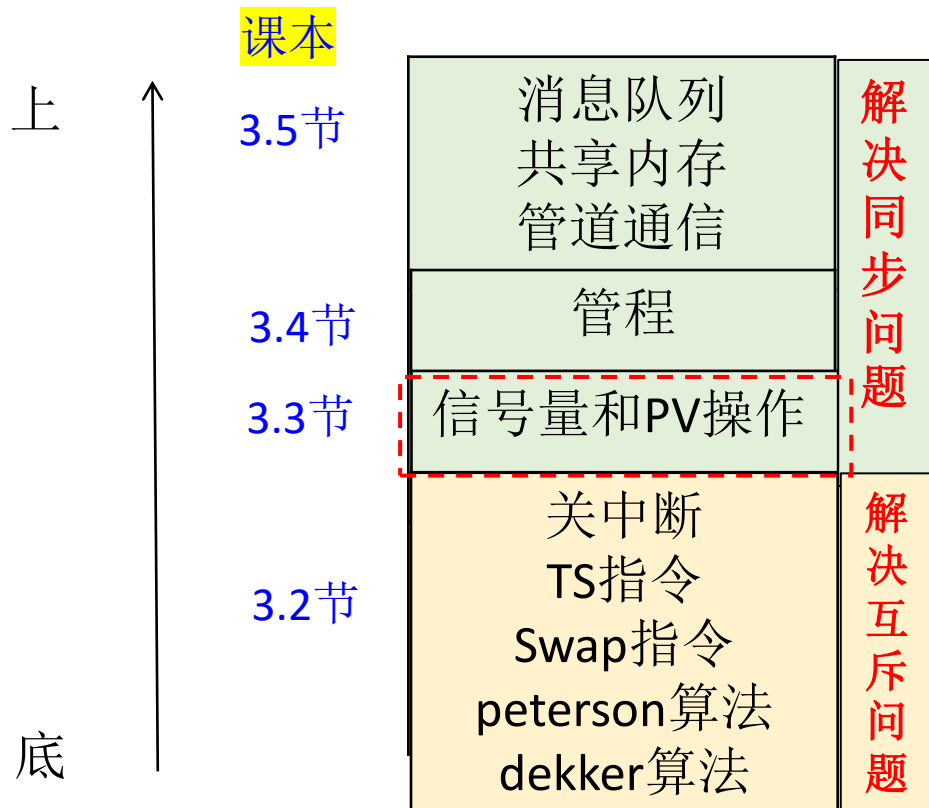
直接制约

1.1 并发进程

1. 进程同步»

进程的同步与互斥(2/2)

□ 实现机制



- 互斥是一种特殊的同步，即逐次使用互斥资源。因此，解决同步问题的方法同样能解决互斥问题，反之并不成立。

如果并发进程交互不当则会发生死锁，解决死锁有三套方案：
防止死锁、避免死锁、检测死锁。

3.6节

CQ1.1.1 在操作系统中，要对并发进程进行同步的原因是（ ）。

- ☐ A 进程必须在有限时间内完成
- ☐ B 进程具有动态性
- ☒ C 并发进程的异步性
- ☐ D 进程具有结构性

提交

CQ1.1.2 进程A和进程B通过共享缓冲区协作完成数据处理，进程A负责产生数据并放入缓冲区，进程B从缓冲区读取数据并输出。进程A和进程B之间的制约关系是（ ）。

- ☐ A 互斥关系
- ☐ B 同步关系
- ☒ C 互斥和同步关系
- ☐ D 无制约关系

提交

1.1 并发进程

1.2 临界区管理

- I 临界区及其使用
- II 实现临界区互斥的软件方法
- III 实现临界区互斥的硬件方法

1.3 信号量与PV操作

临界区及其使用(1/2)

❑ 临界资源(critical resource)

- 一次只能供一个进程使用的资源
 - ✓ 许多物理设备都属于临界资源，如打印机、扫描仪等
 - ✓ 许多可被多个进程共享的变量和文件也属于临界资源
- 对于临界资源的访问必须采用互斥方式。

❑ 临界区(critical section)

互斥区

- 在并发进程中，访问临界资源的程序段；
- 为了保证临界资源的正确使用，可以把临界资源的访问过程分为4个部分：

进入区

临界区

退出区

剩余区

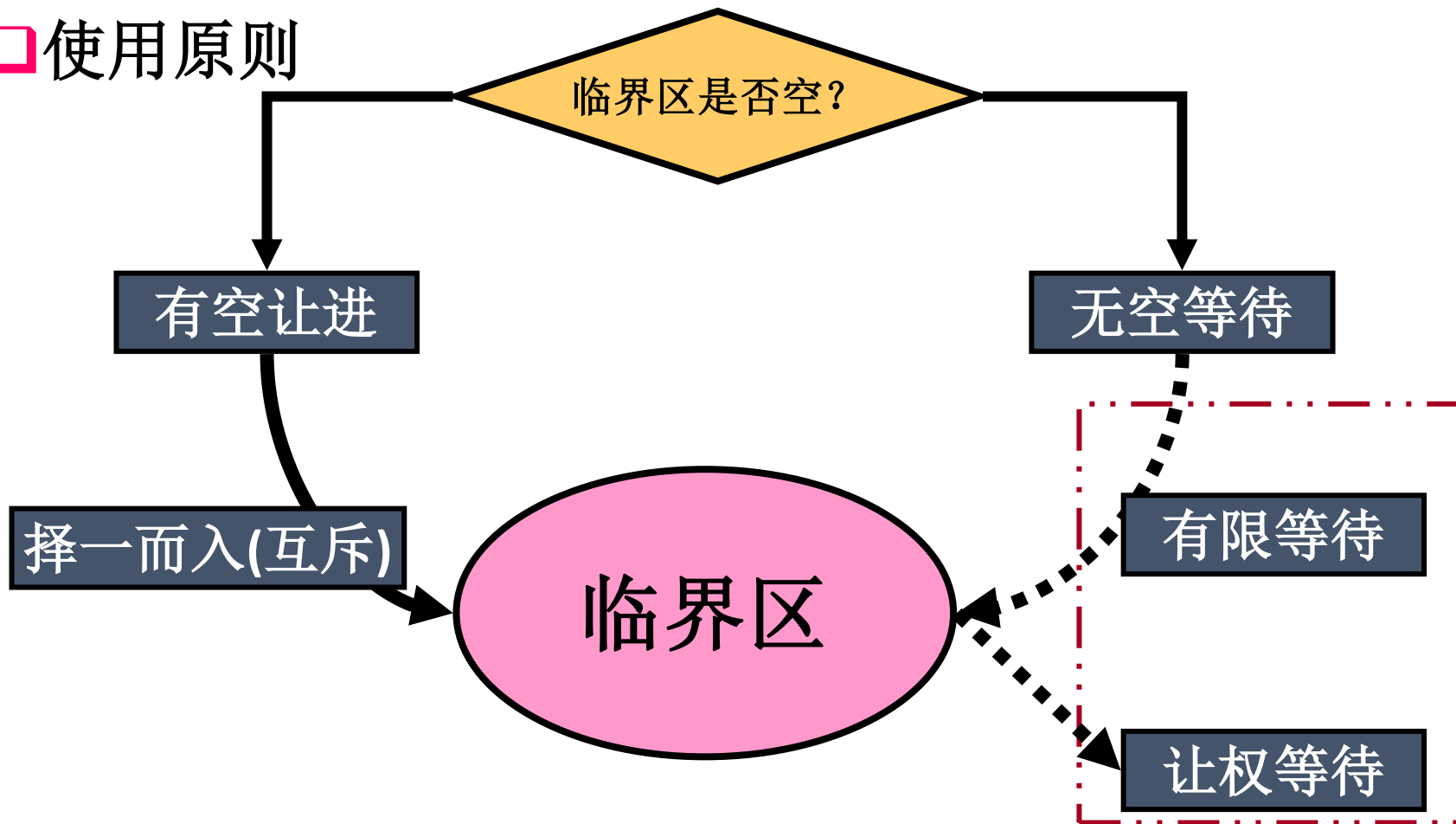


1.2 临界区管理

1. 进程同步》

临界区及其使用(2/2)

□ 使用原则



有限等待: 对要求访问临界资源的进程, 应保证有限时间内能进入临界区。以免陷入“死等”状态。

让权等待: 当进程不能进入临界区时, 应释放CPU, 以免进程陷入“忙等”状态。




1.2 临界区管理



1. 进程同步»

临界区互斥的软件方法(1/7)

❑ 临界区管理的尝试½

```
int turn = 1 or 2;    /*表示谁可以进入临界区*/

process P1  
{
    while (turn != 1);
    
    turn = 2;
    
}

process P2
{
    while (turn != 2);
    
    turn = 1;
    
}
```

进入区
退出区

- 问题：P1和P2必须交替执行！若某时刻P1执行结束，turn=2，但P2暂不要求访问该临界资源，则P1无法再次进入临界区。不满足有空让进。

总结：本算法可以保证两个进程互斥访问临界资源，但存在的问题是强制两个进程以交替次序进入临界区，造成资源利用不充分的情况。

1.2 临界区管理

1. 进程同步

总结：本算法不能解决临界资源互斥访问的问题。

临界区互斥的软件方法(2/7)

□ 临界区管理的尝试1

```
BOOL inside1, inside2;  
inside1 = false; /* P1不在其临界区内 */  
inside2 = false; /* P2不在其临界区内 */
```

process P ₁	process P ₂
{ while (inside2); ①	{ while (inside1); ②
inside1 = true; ③	inside2 = true; ④
临界区;	临界区;
inside1 = false;	inside2 = false;
剩余区;	剩余区;
}	}

① ② ③ ④ 进入区 退出区

问题：当两个进程都未进入临界区时，它们各自的访问标志值均为false，若此时两个进程几乎同时都想进入临界区，并且发现对方标志值为false，于是两个进程同时置自己标志值为true，并进入各自的临界区，不满足择一而入。

1.2 临界区管理

1. 进程同步»

临界区互斥的软件方法(3/7)

总结：本算法可以防止两个进程同时进入临界区，但存在两个进程都进不了临界区的问题。

❑ 临界区互斥的尝试2

```

    BOOL inside1, inside2;
    inside1 = false;      /* P1不想进其临界区 */
    inside2 = false;      /* P2不想进其临界区 */

process P1
{
    inside1 = true; ①
    while (inside2); ③
    // 临界区;
    inside1 = false;
    // 剩余区;
}

process P2
{
    inside2 = true; ②
    while (inside1); ④
    // 临界区;
    inside2 = false;
    // 剩余区;
}
// 退出区
```

问题：当两个进程几乎同时都想进入临界区时，它们分别将自己的标志值置为 **true**，并且同时去检查对方的状态，发现对方也要进入临界区，从而出现死循环，没有进程能在有限时间内进入临界区，造成**永远等待**。**“饥饿”现象**

临界区互斥的软件方法(4/7)

□ 两次尝试失败的原因

- 两进程总是想争先进入临界区，导致：
 - 尝试1: 先测试再上锁，两个都进去了
(不满足择一而入)
 - 尝试2: 先上锁再测试，两个都进不去
(不满足有空让进、有限等待)
- 进入临界区的两个必要步骤：
 - 测试/等待: `while(inside[i])`
 - 上锁: `inside[i] = true`



• 如何改进?

让双方变得更礼貌点，总是让对方先进!

临界区互斥的软件方法(5/7)

❑ 临界区互斥的尝试3

- `inside[i]`表示进程*i*想进入临界区，而不表示它已经在临界区；
- 加入一个令牌`turn`，表示该由谁进入；
- *i*若想进入临界区先提交申请，并将令牌丢给对方，让对方优先进入：

`inside[i] = true ; turn = j ;`

- 然后测试对方*j*是否能进入：

`while ((turn == j) and (inside[j] == true)) ;`

Peterson算法

1.2 临界区管理

1. 进程同步»

课本P131

临界区互斥的软件方法(6/7)

❑ 临界区互斥的尝试3——Peterson算法

```
int turn = 1 or 2;           /* 令牌归谁所有 */
BOOL inside[2] = {false, false}; /* P1 P2不想进其临界区 */
```

Process P₁

```
{ inside[1] = true;
```

```
  turn = 2;
```

```
  while (turn==2 && inside[2]);
```

临界区;

```
inside[1] = false;
```

剩余区;

```
}
```

Process P₂

```
{ inside[2] = true;
```

```
  turn = 1;
```

```
  while (turn==1 && inside[1]);
```

临界区;

```
inside[2] = false;
```

剩余区;

```
}
```

进入区

退出区

总结：本算法结合了第1次尝试和第2次尝试，由于任何一个时刻只有一个进程掌握令牌，总有一个进程的while循环条件为假，进入临界区执行，因此避免了尝试2中的“永远等待”现象。

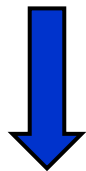
临界区互斥的软件方法(7/7)

❑ 软件方法的缺点

- 忙式等待：空循环在浪费CPU时间片。
- 实现复杂，需要较高的编程技巧，不同的进程实现代码要分别设计。

❑ 产生问题的根本原因

测试和上锁没有一气呵成



硬件方法

I. 开、关中断方法

II. 硬件指令方法

原语

test_and_set()

compare_and_swap()

1.2 临界区管理

1. 进程同步»

临界区互斥的硬件方法(1/5)

❑ 开、关中断方法

- 典型模式

```
...  
关中断;          // cli( )  
临界区;  
开中断;          // sti( )  
...
```

- 局限性

- 效率降低
- 安全风险

不适合作为通用的互斥机制

临界区互斥的硬件方法(2/5)

□ 硬件指令方式

• 思路

许多系统提供了专门的、不可中断的硬件指令，完成对一个字或者字节的内容进行检查、修改，或者交换两个字或字节的内容，利用这些硬件指令解决临界区的互斥问题。

atomic operation

互斥锁(mutex lock)

• 管理临界区的硬件指令

-- test_and_set()指令 (TS指令)

-- compare_and_swap()指令

自学，P133



1.2 临界区管理

1. 进程同步



临界区互斥的硬件方法(3/5)

□ TS指令的描述

TS指令管理临界区时，把一个临界区与一个布尔变量相连，该变量为true表示临界区空闲，进程进入临界区前要保证TS(s)的返回值为true。

```
BOOL TS (BOOL &x)
{
    if (x)                //测试
    { x = false;           //上锁
      return true;         //形成条件码
    }
    else
      return false;       //形成条件码
}
```

原语

说明:

- 如果临界区空闲则可以进入，进入后修改布尔变量的值为false，表明临界区被占用，之后返回条件码true;
- 如果临界区被占，则返回条件码false，表明暂时不能进入临界区。

1.2 临界区管理

1. 进程同步》



临界区互斥的硬件方法(4/5)

❑ TS指令的应用

- 利用TS指令实现临界区的互斥访问

```
BOOL s = true; // free
cobegin
process Pi()
{
    while ( !TS(s) ); //loop if busy, lock
    临界区;
    s = true; //unlock
    剩余区;
}
coend
```

说明

- 如果临界区空闲 ($s=true$) , TS返回true并将s设为false, 此时临界区为设为占用状态, 同时跳出while循环进入临界区;
- 如果临界区占用 ($s=false$), 则TS返回false并且s值不变, 进程陷入循环测试, 直至s值为true!

临界区互斥的硬件方法(5/5)

- ❑ 硬件指令互斥锁存在**忙式等待**(busy waiting)，即占用CPU执行空循环实现等待。
- ❑ 这种类型的互斥锁也被称为“**自旋锁**”(spin lock)
 - 缺点：
 - 浪费CPU周期
 - 优点：
 - 进程在等待时没有上下文切换，对于使用锁时间不长的进程，自旋锁还是可以接受的；
 - 在多处理器系统中，自旋锁的优势更加明显。

CQ1.2.1 在操作系统中，临界区是()

- ☐ A 一个缓冲区
- ☐ B 一段共享数据区
- ☒ C 一段程序
- ☐ D 一个互斥资源

提交

CQ1.2.2 以下（ ）不属于临界资源。

- ☐ A 打印机
- ☒ B 非共享数据
- ☐ C 共享变量
- ☐ D 共享缓冲区

提交

CQ1.2.3 下述选项中，体现原语特点的是（ ）。

- ☐ A 并发性
- ☐ B 共享性
- ☐ C 结构性
- ☒ D 不可分割性

提交

1.1 并发进程

1.2 临界区管理

★ 1.3 信号量与P、V操作

I 信号量机制

概念

II 用信号量机制实现进程互斥

III 用信号量机制实现进程同步

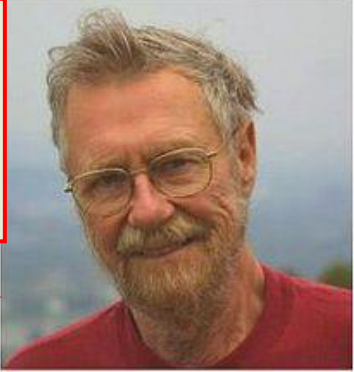
应用

消息队列 共享内存 管道通信	解决同步问题
信号量机制	
关中断 TS指令 Swap指令 peterson算法 dekker算法	解决互斥问题

1.3 信号量与P、V操作

信号量机制(1/5)

1930年5月11日~2002年8月6日
1972 图灵奖
1989 ACM计算机科学教育教学
杰出贡献奖



广义的锁

□ 信号量的提出

- 1965年荷兰学者Dijkstra提出**信号量(semaphore)**机制，这是一种**比互斥锁更强大的同步工具**，它可以提供更高级的方法来同步并发进程。

□ 信号量的定义

- 信号量是一个**数据结构**，负责协调各个进程，以保证它们能够正确、合理的使用公共资源。

```
typedef struct semaphore
{
    int value;           //信号量的值
    struct pcb* list;    //信号量等待队列指针
} semaphore;
```

semaphore s ;



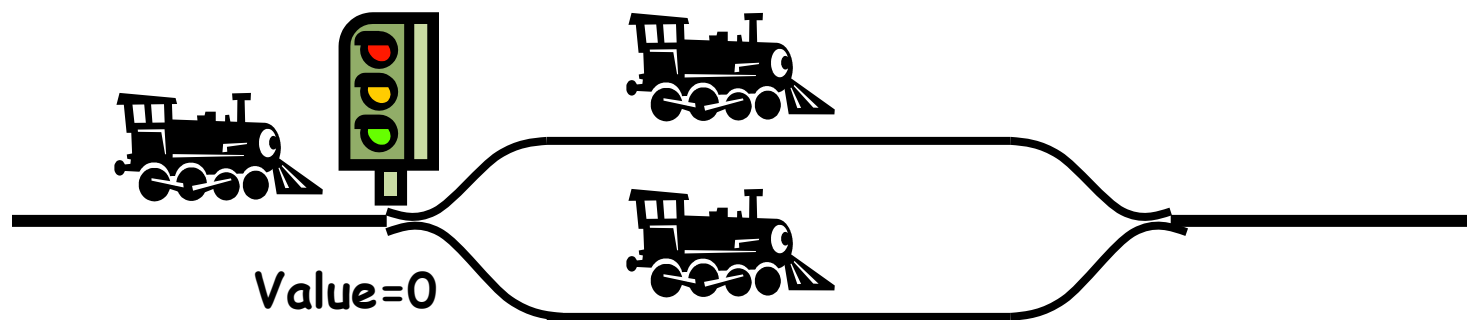
- (1) 信号量值的含义?
- (2) 信号量等待队列的用途?

1.3 信号量与P、V操作

信号量机制(2/5)

□ 信号量的特性

- 信号量代表对某种公共资源的管理，其初值是一个非负整数，表征资源的数量。
- 要使用资源的进程需通过信号量申请：有资源时顺利通过；若没有资源可用，则进程将进入对应信号量的等待队列。
- 当一个进程归还资源时，此时若信号量队列有等待进程则唤醒一个。



1.3 信号量与P、V操作

信号量机制(3/5)

□ P、V操作的功能

- P操作用于申请信号量管理的资源。
 - 信号量的值减1
 - 有条件阻塞自己
- V操作用于释放资源。
 - 信号量的值加1
 - 有条件唤醒其它进程

P、V操作均是低级进程通信原语，执行过程不可中断。

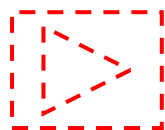
除了初始化之外，P、V操作是两个唯一能对信号量的值进行处理的操作。

1.3 信号量与P、V操作

信号量机制(4/5)

□ P、V原语的实现

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



```
void P(semaphore s)
{
    s.value = s.value - 1;
    if (s.value < 0)
        sleep(s.list);
}
```

资源不足(或事件未到)则阻塞自己

```
void V(semaphore s)
{
    s.value = s.value + 1;
    if (s.value <= 0)
        wakeup(s.list);
}
```

有进程等待资源(或事件)则唤醒“他人”



信号量和PV操作通过**有条件阻塞和唤醒**，巧妙地避免了“忙式等待”，这种机制被广泛应用于解决并发进程的竞争问题，以及并发进程的协作问题！

1.3 信号量与P、V操作

信号量机制(5/5)

□ 总结 信号量机制用于资源共享时的含义

若 s 代表某个资源对应的信号量，则

- ① $s.value > 0$ 表示有 $value$ 个资源可用
- ② $s.value = 0$ 表示无资源可用
- ③ $s.value < 0$ 则 $|s.value|$ 表示 s 等待队列中的进程个数
- ④ $P(s)$ 表示申请一个资源
- ⑤ $V(s)$ 表示释放一个资源

1.1 并发进程

1.2 临界区管理

1.3 信号量与P、V操作

I 信号量机制

II 用信号量机制实现进程互斥

III 用信号量机制实现进程同步

- 互斥信号量
- 计数信号量
- 哲学家就餐问题

用信号量机制实现互斥

□互斥信号量

semaphore s.value=2;

cobegin

process Pi { /*i=1,2,...,n*/

P(s);

临界区;

V(s);

}

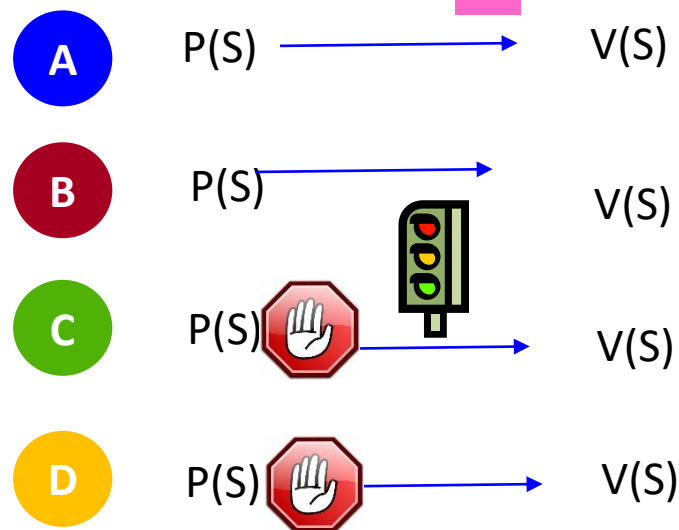
coend



- s是公有信号量，用于实现资源的互斥访问，初值置为1，表示临界区一次仅能让一个进程进入。

DO	Operation	s.value
1	Initialize	2
2	A:P(S)	0
3	A:V(S)	1
4	B:P(S)	0
5	C:P(S)	-1 阻塞
6	D:P(S)	-2 阻塞
7	B:V(S)	-1 唤醒
8	C:V(S)	0 唤醒
9	D:V(S)	1

S.value=2



□ 计数信号量(counting semaphore)

- 初始值可以是任意非负整数的信号量，用于控制并发进程对一类、多个共享资源的访问。

```
semaphore road.value = 2;
```

```
cobegin
```

```
process Cari{
```

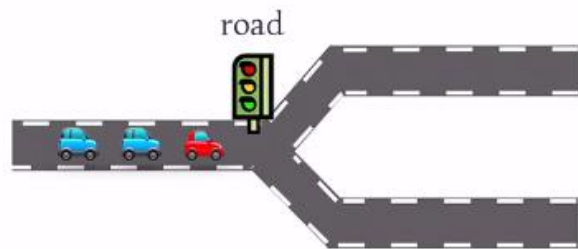
```
    P(road);
```

```
    占用一条道路通行;
```

```
    V(road);
```

```
}
```

```
coend
```



信号量s初始值的含义 **s.value ≥ 0**

s.value = 1 临界资源互斥访问

s.value > 1 同类可用资源数量

s.value = 0 **P(s) → sleep**
强制进程等待事件发生

用于互斥访问的信号量s，竞争相应资源的每个进程在其进入区执行**P(s)**，退出区执行**V(s)**。
用于同步的信号量却非如此！

□ 哲学家就餐问题(1/3)

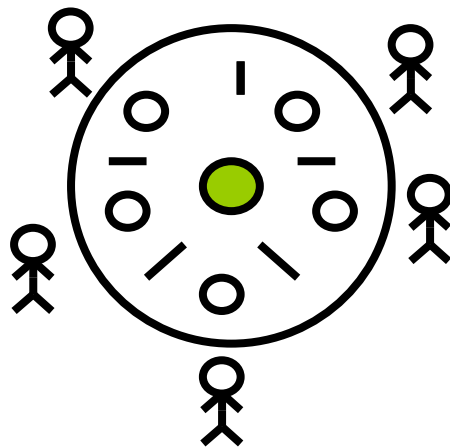
• 问题描述

5个哲学家围在桌旁，每人面前有一盘子，每两人之间放一把叉子。每个哲学家思考、饥饿，然后想吃通心面。只有拿到两把叉子的人才可以吃面，并且每人只能直接拿自己左手或右手边的叉子。

• 解题思路

5位哲学家看作5个进程，需要互斥使用5把叉子，这5把叉子是共享资源，用5个信号量表示。

每个叉子一次只能由一个哲学家使用，初值为1。

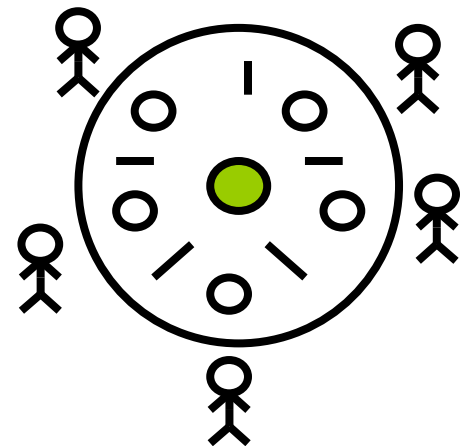


用信号量机制实现互斥

□ 哲学家就餐问题(2/3)

- 核心实现

课本139



简单起见，信号量赋初值操作不再使用成员变量名value.

1.3 信号量与P、V操作

```
semaphore fork[5];  
  
for( int i=0; i<5; i++)  
    fork[i]= 1;  
  
cobegin  
    process Pi {                //i=0,1,2,3,4  
        while(true){  
            think( );  
  
            P( fork[i] );  
            P( fork[(i+1) % 5] );  
  
            eat( );  
  
            V( fork[i] );  
            V( fork[(i+1) % 5] );  
        }  
    }  
coend
```

□ 哲学家就餐问题(3/3)

- 死锁问题及其解决方案

- 上述解法可能出现永远等待

死锁!

当五个人同时都取到右边（或左边）的叉子，企图再拿起其左边（或右边）的叉子时发生。

- 有若种办法可避免死锁：

- 1) 至多允许四个哲学家同时吃；
- 2) 奇数号先取左手边的叉子，偶数号先取右手边的叉子；
- 3) 每个哲学家取到手边的两把叉子才吃，否则一把叉子也不取。

1.1 并发进程

1.2 临界区管理

1.3 信号量与P、V操作

I 信号量机制

II 用信号量机制实现进程互斥

III 用信号量机制实现进程同步

- 基本思路
(司机-售票员问题)
- 生产者-消费者问题
- 苹果-桔子问题
- 读者-写者问题
- 理发师问题

□ 基本思路 (1/5)

- 同步问题实质上是让那些随机发生的事件变得有序。
- 实现的方法是调节并发进程的执行速度，让某些进程必须等待另一些进程的执行而得以继续。
- 在同步问题里，我们利用执行P原语会引发等待的效果来实现调节并发进程执行速度的目标。

为了让进程A等待，信号量s的初值置为0，进程A执行P(s)操作进入等待状态，当满足运行条件时，由其他进程执行V(s)，将进程A唤醒。通常此类同步信号量被称为进程A的私有信号量。

用信号量机制实现同步

1.3 信号量与P、V操作



若进程P1也要同步
等待P2?该如何是好?

□ 基本思路 (2/5)

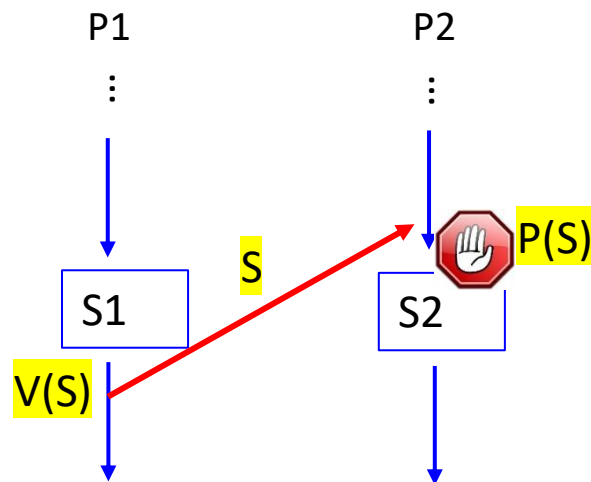
· 最基本的同步实例

P1执行的程序中有一条语句S1，P2执行的语句中有一条语句S2，只有当P1执行了S1语句之后，P2才能开始执行语句S2。

设S为控制进程并发执行的信号量，初值为0。

```
semaphore S=0;  P1 ()      P2 ()
main()          { ...      { ...
{  cobegin      S1;        P (S) ;
    P1 () ; P2 () ;    V (S) ;    S2;
    coend        ...}      ...}
}
```

对协作进程间的执行顺序进行同步

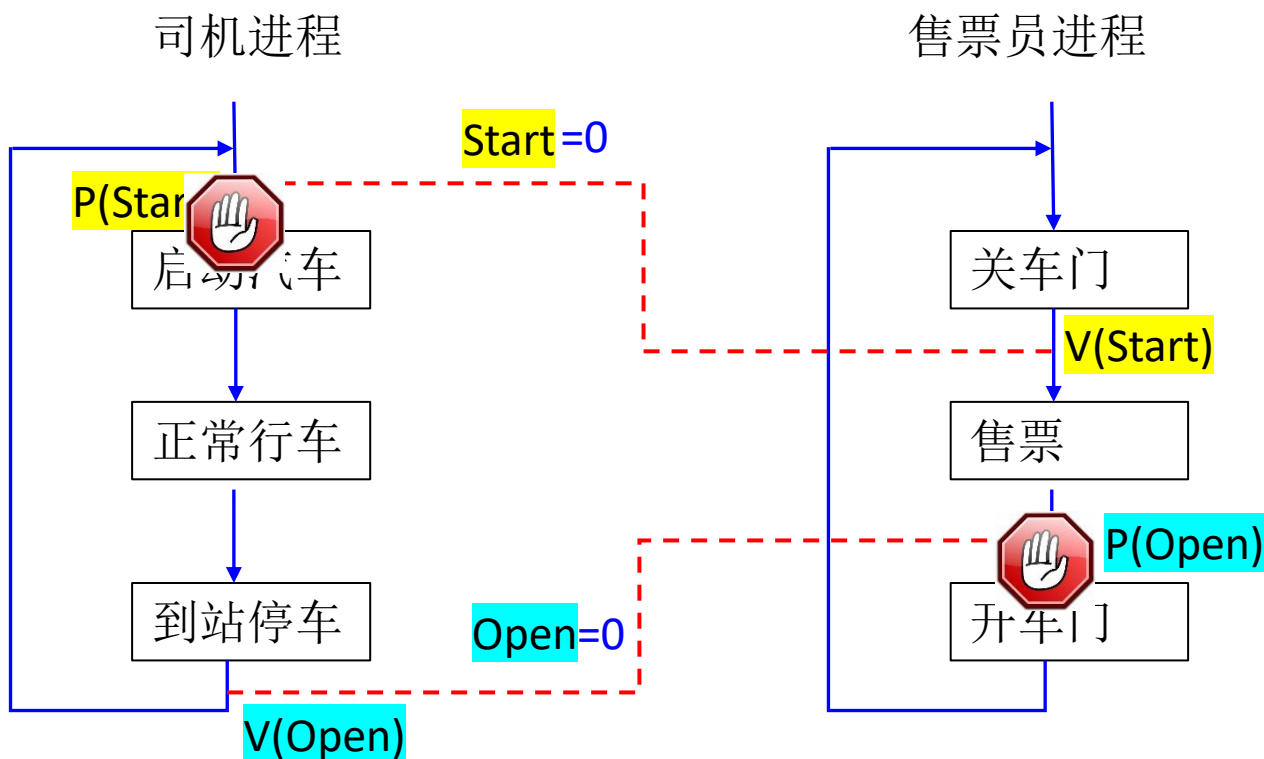


□ 基本思路 (3/5)

• 交替执行进程的同步问题实例

规则：

- (1) 司机要等车门关闭才能启动汽车；
- (2) 售票员要等到站停车才能开车门。



□基本思路 (4/5)

semaphore Start=0; //司机私有信号量, 表示是否可以开车

semaphore Open=0; //售票员的私有信号量, 表示是否可以开门

P司机

```
{ while(true)
{ P(Start); //司机申请启动汽车
  启动汽车;
  正常行车;
  到站停车;
  V(Open); //允许售票员开门
}
```

P售票员

```
{ while(true)
{ 关车门;
  V(Start); //允许司机启动汽车
  售票;
  P(Open); //售票员申请开门
  开车门
}
```



- 同步进程之间具有某种合作关系, 如在执行时间上必须按一定的顺序协调进行, 或者共享某资源。
- 互斥是由于并发进程之间竞争临界资源引起的, 互斥进程彼此在逻辑上完全无关, 它们的运行不具有次序的特征。

□基本思路 (5/5)

• 控制进程执行次序的实例

- P_1 、 P_2 、 P_3 、 P_4 、 P_5 、 P_6 为一组合作进程，其前趋图如下，试用信号量及P、V操作实现这6个进程的同步。

解题思路：

- (1) 为每条直接前趋关系 $P_i \rightarrow P_j$ 定义一个信号量，该信号量初值为0；
- (2) 在 P_i 结束后执行对该信号量的V操作，在 P_j 开始之前执行对该信号量的P操作。

```
semaphore a=0,b=0,c=0,d=0,e=0,f=0,g=0;
```

```
main()
```

```
{ cobegin
```

```
  P1(); P2(); P3(); P4(); P5(); P6();
```

```
  coend
```

```
}
```

```
P1( ) { ... .. V(a); V(b); }
```

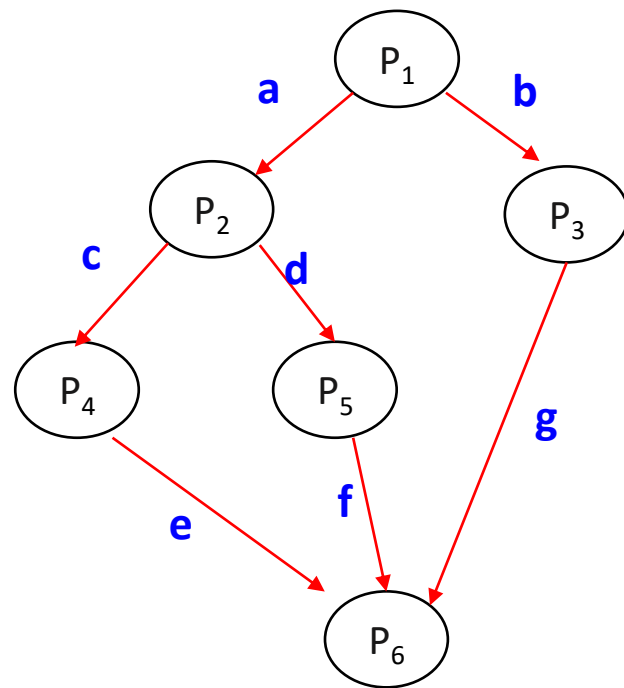
```
P2( ) { P(a); ..... V(c); V(d); }
```

```
P3( ) { P(b); ..... V(g); }
```

```
P4( ) { P(c); ..... V(e); }
```

```
P5( ) { P(d); ..... V(f); }
```

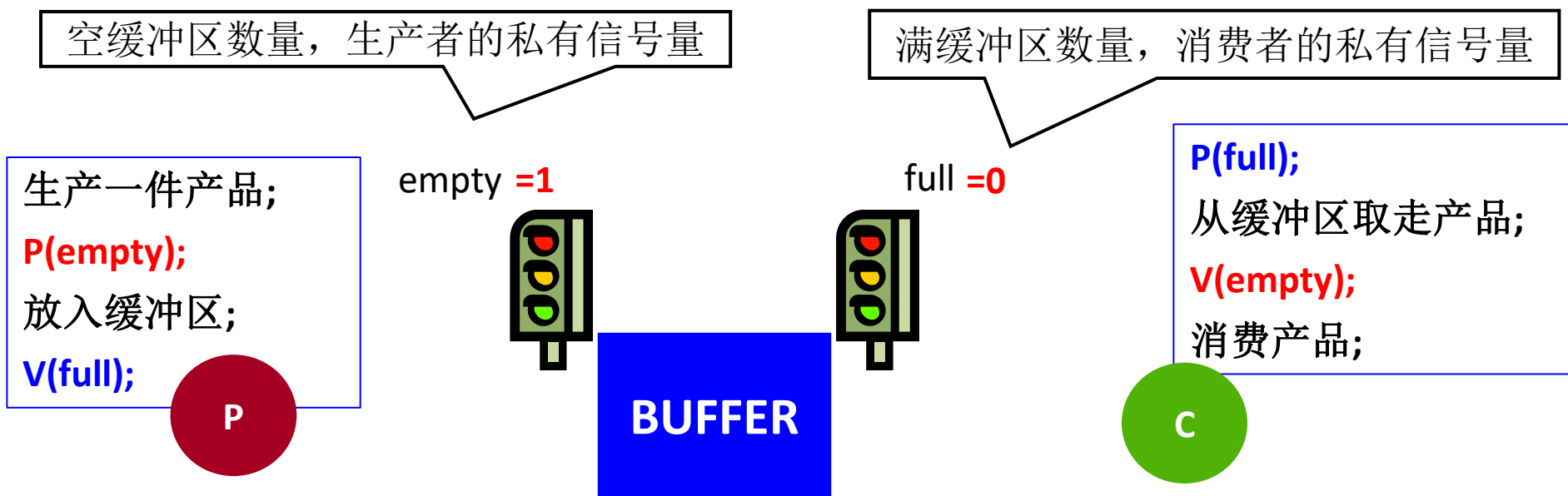
```
P6( ) { P(e); P(f); P(g); ..... }
```



生产者-消费者问题(1/5)

单缓冲区问题

- 生产者(P)与消费者(C)各一人共用一个缓冲区，当缓冲区“满”时P进程必须等待，当缓冲区为“空”时，C进程必须等待。



- (1) 初始状态 : BUFFER为空
- (2) 设置信号量 几个? 初值?

□ 生产者-消费者问题(2/5)

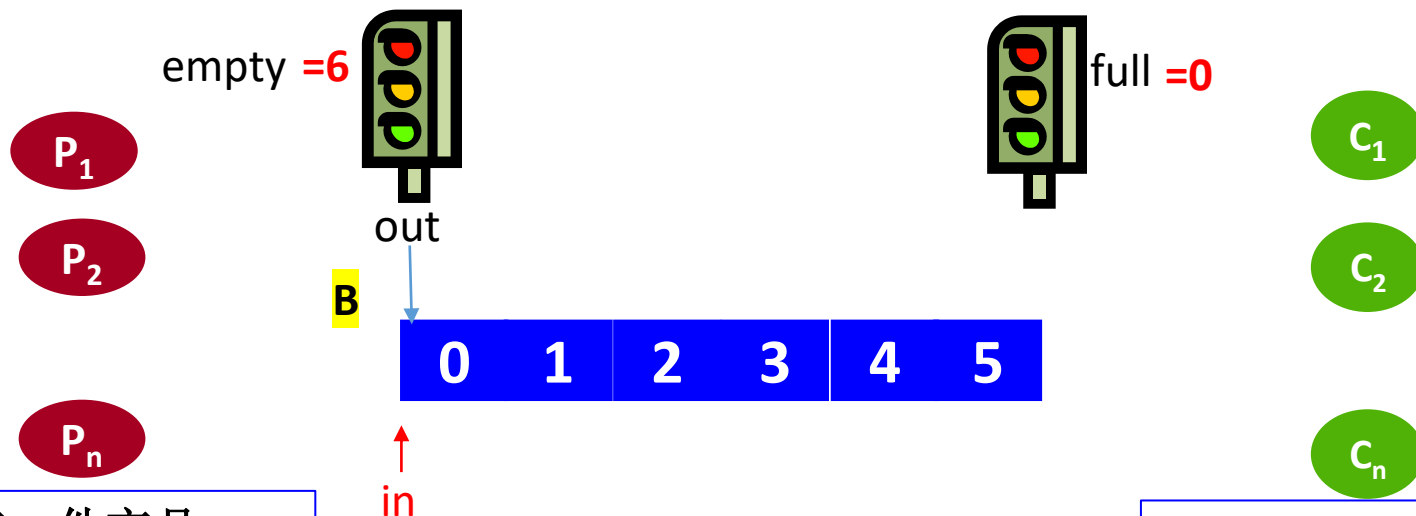
单缓冲区问题

```
item B;                                //缓冲区数量为1个
semaphore empty=1;                      //生产者的私有信号量, 空缓冲区数量
semaphore full=0;                       //消费者的私有信号量, 满缓冲区数量
cobegin
    process producer() {                process consumer() {
        while (true)                    while (true)
        {
            生产产品;                    P(full);
            P(empty);                    take from B;
            append to B;                 V(empty);
            V(full);                     消费产品;
        }                                }
    }                                    }
coend
```

□ 生产者-消费者问题(3/5)

有界缓冲区问题

- 生产者们(P)与消费者们(C)共用多个缓冲区，生产者进程不能往“满”的缓冲区中放产品，消费者进程不能从“空”的缓冲区中取产品。



生产一件产品;
P(empty);
append to B[in];
in=(in+1) % 6;
V(full);

P(full);
take from B[out];
out=(out+1) % 6;
V(empty);
消费产品;

用信号量机制实现同步

生产者-消费者问题(4/5)

item B[k]; int in=0,out=0;

semaphore empty = k, full = 0;

semaphore mutex=1; //互斥信号量

cobegin

```
process producer_i() {
```

```
while (true)
```

```
{ 生产一个产品;
```

```
  P(empty);
```

```
  P(mutex);
```

```
  append to B[in];  
  in=(in+1) % k;
```

```
  V(mutex);
```

```
  V(full);
```

```
}
```

```
}
```

```
process consumer_i() {
```

```
while (true)
```

```
{  P(full);
```

```
  P(mutex);
```

```
  take from B[out];  
  out=(out+1) % k;
```

```
  V(mutex);
```

```
  V(empty);
```

```
  消费一个产品;
```

```
}
```

```
}
```

```
coend
```



1.3

是否会有问题?



OUT



是否存在多个生产者/消费者同时进入临界区的可能? ✓

竞争条件

临界区



能否交换进程内P操作的顺序?

生产者-消费者问题(5/5)

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

Time

0

```
C1  P(mutex); mutex=0
      P(full);  C1 sleep

P1  P(mutex);  P1 sleep

P2  P(mutex);  P2 sleep
⋮
```

死锁

cobegin

```
process producer_i() {
    while (true)
    {  生产一个产品;
      P(empty);
      P(mutex);
      append to B[in];
      in=(in+1) % k;
      V(full);
      V(mutex);
    }
}

process consumer_i() {
    while (true)
    {  P(full);
      P(mutex);
      take from B[out];
      out=(out+1) % k;
      V(empty);
      V(mutex);
      消费一个产品;
    }
} coend
```

□ 信号量机制使用注意事项

- 对某一信号量 s 的P、V操作必须成对出现，有一个 $P(s)$ 操作就一定有一个对应的 $V(s)$ 操作。
 - 实现互斥操作时， $P(s)$ 、 $V(s)$ 同处于同一进程
 - 实现同步操作时， $P(s)$ 、 $V(s)$ 不在同一进程中出现
- 如果 $P(\text{full})$ 或 $P(\text{empty})$ 和 $P(\text{mutex})$ 两个操作在一起，那么P操作的顺序至关重要,一个同步P操作与一个互斥P操作在一起时 **同步P操作在互斥P操作前**，而两个V操作无关紧要。

□ 同步问题的分析过程

- 有几个并发进程，公用缓冲区的个数。
- 分析每个进程的执行流程，必要时画出它的流程图，找出在何时何处需要等待！
- 设置几个信号量？信号量初值设成多少？需要其它变量辅助吗？
- 公用缓冲区是否需要用互斥信号量保护？

□ 苹果-桔子问题(1/3)

• 问题描述

- 桌上有一只盘子，每次只能放入一只水果。
- 妈妈专向盘子中放桔子(orange)，爸爸专向盘子中放苹果(apple)。
- 一个儿子专等吃盘子中的桔子，一个女儿专等吃盘子里的苹果。

• 问题分析

- 盘子→缓冲区；爸、妈→生产者；儿、女→消费者
- 这是多个生产/消费者、一个缓冲区、多种产品的问题。
- 设置三个信号量：
 - empty用于指示盘子能放几个水果，初值=? **1**
 - orange、apple分别用于指示桔子和苹果的个数，初值=? **0**

用信号量机制实现同步

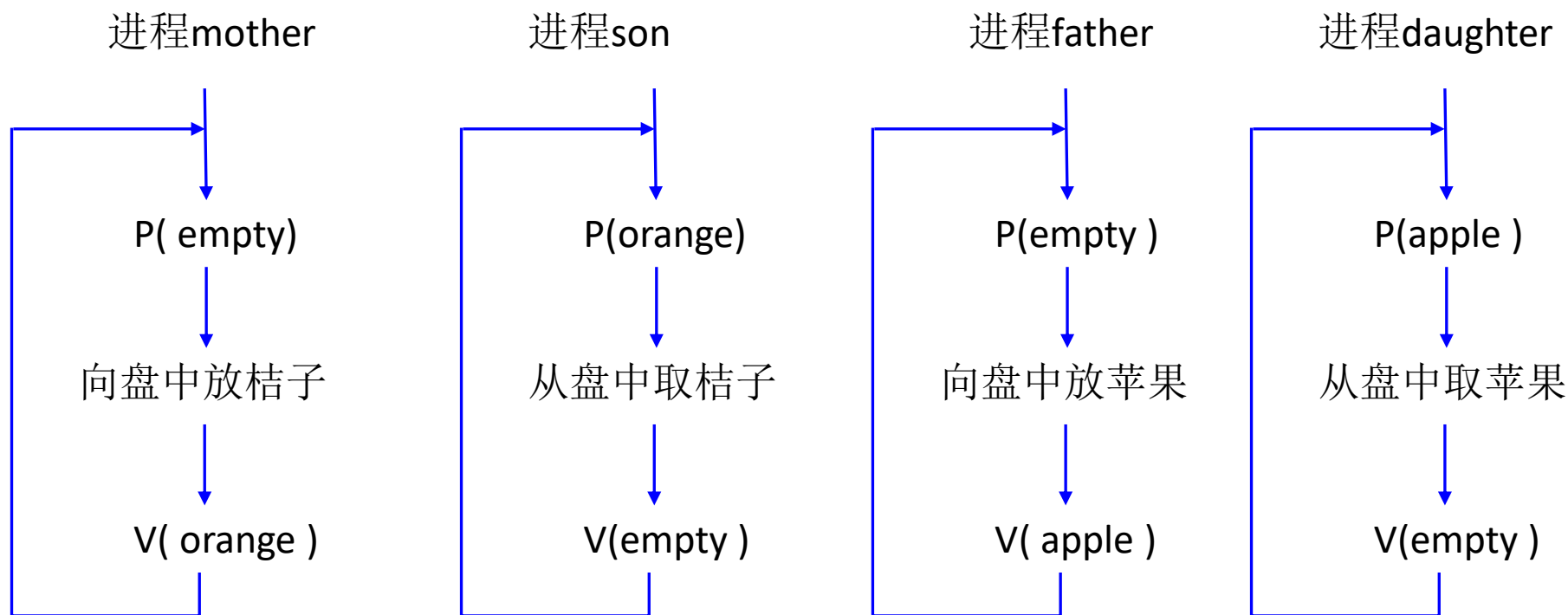
1.3 信号量与P、V操作

□ 苹果-桔子问题(2/3)

- 4个进程的同步

```
empty = 1;  
orange = 0;  
apple = 0;
```

```
/* 盘子里允许放一个水果 */  
/* 盘子里没有桔子 */  
/* 盘子里没有苹果 */
```



□ 苹果-桔子问题(3/3)

```
semaphore empty    = 1;    /* 盘子里允许放一个水果*/  
semaphore orange   = 0;    /* 盘子里没有桔子 */  
semaphore apple    = 0;    /* 盘子里没有苹果*/
```

```
Process mother {  
    剥一个桔子;  
    P(empty);  
    把桔子放入plate;  
    V(orange);  
}
```

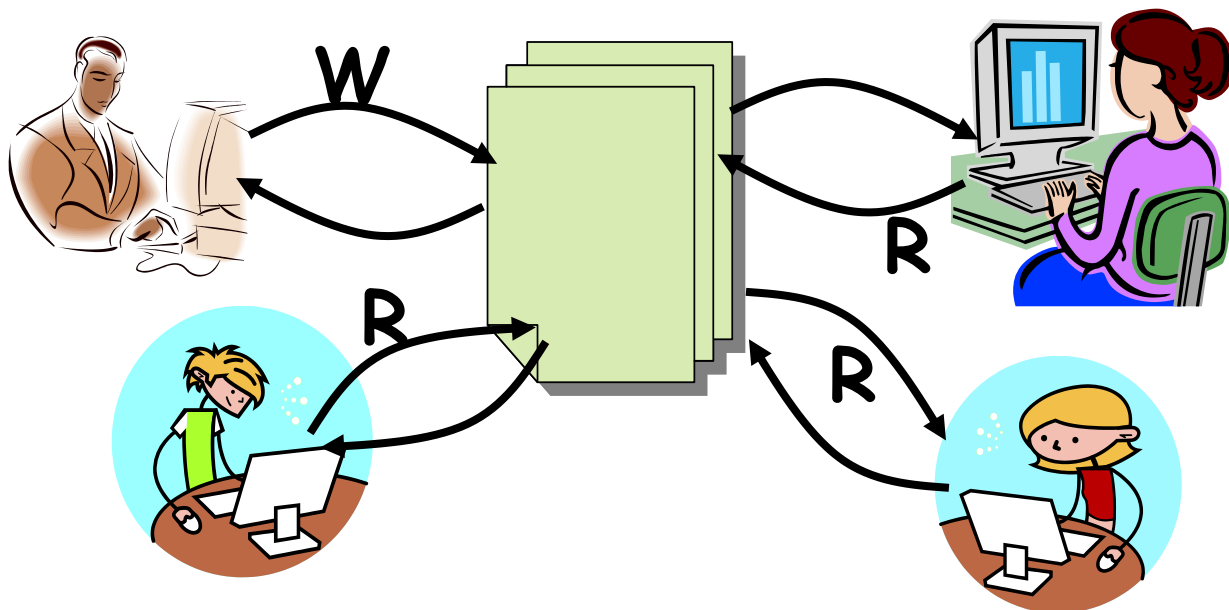
```
Process son {  
    P(orange);  
    从plate中取桔子;  
    V(empty);  
    吃桔子;  
}
```

```
Process father {  
    削一个苹果;  
    P(empty);  
    把苹果放入plate;  
    V(apple);  
}
```

```
Process daughter {  
    P(apple);  
    从plate中取苹果;  
    V(empty);  
    吃苹果;  
}
```

❑ 读者-写者问题(1/3)

• 问题描述



-- 有两组并发进程：读进程和写进程，共享一个数据集/文件，要求：

- 允许多个读进程同时执行读操作；
- 写进程执行写操作前，应让已有的写进程和读进程全部退出；
- 任一写进程在完成写操作之前不允许其它读进程或写进程工作。

-- 简单一句话：某一时刻，允许存在多个读进程，但仅能有一个写进程。

❑ 读者-写者问题(2/3)

• 解题思路

- (1) 多个读进程可以并发读数据集，用一个**整型变量rc**记录并发读数据集的进程数(初值为0)，当 $rc \geq 1$ 时禁止写进程对数据集进行写操作，当 $rc=0$ 时允许写进程对数据集进行写操作。
- (2) 所有读进程维护计数变量rc，**该变量对读进程而言是一个临界资源**，为此设置**互斥信号量mutex**，防止几个读进程同时修改rc的值。
- (3) 数据集对写进程而言是个临界资源，设置**互斥信号量db**。

- **整型变量rc**: 公共变量，记录读进程的个数，初值为0。
- **互斥信号量mutex**: 保护对rc的互斥访问，初值为1。
- **互斥信号量db**: 控制写进程对数据集的互斥访问，初值为1。

用信号量机制实现同步

□ 读者-写者问题(3/3)

```
semaphore db = 1;
```

```
semaphore Mutex = 1;
```

```
int rc = 0;
```

```
void Writer_j( )
```

```
{ while(true)
```

```
{
```

```
    P(db) ;
```

```
    /*写数据集;*/
```

```
    V(db) ;
```

```
}
```

```
}
```



写者优先呢?

1.3 信号量与P、V操作

```
void Reader_i( )
```

```
{ while(true)
```

```
{ P(Mutex) ;
```

```
    rc = rc + 1;
```

```
    if (rc == 1)
```

```
        then P(db) ;
```

```
    V(Mutex) ;
```

```
    /*读数据集;*/
```

```
    P(Mutex) ;
```

```
    rc = rc - 1;
```

```
    if (rc == 0)
```

```
        then V(db) ;
```

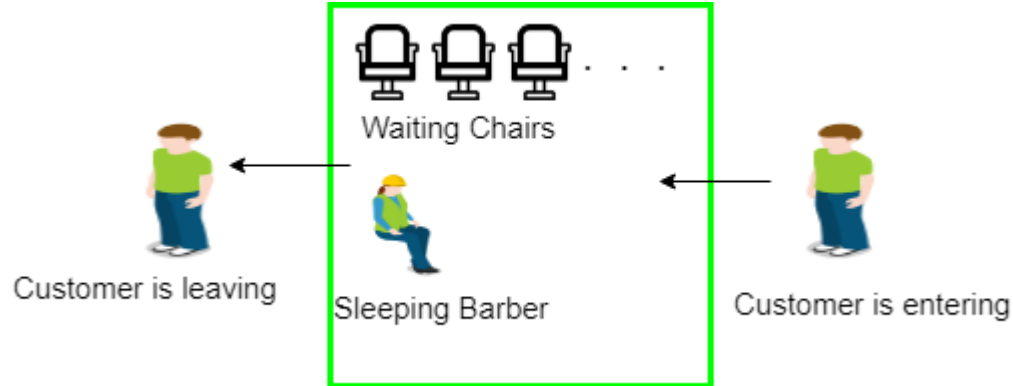
```
    V(Mutex) ;
```

```
}
```

读者优先

用信号量机制实现同步

理发师问题(1/3)



问题描述

- 理发店里有一位理发师、一把理发椅和N(常量)把供等候理发的顾客坐的椅子;
- 如果没有顾客, 理发师便在理发椅上睡觉;
- 一个顾客到来时, 他必须叫醒理发师;
- 如果理发师正在理发时又有顾客来到, 则如果有空椅子可坐, 顾客就坐下来等待, 否则就离开。

变量设置

设置1个计数变量waiting、1个互斥信号量和2个同步信号量。

- 整型变量waiting记录等候理发的顾客数, 初值为0
- 互斥信号量mutex用于互斥修改waiting, 初值为1
- 同步信号量customers表示等候理发的顾客数, 可阻塞理发师进程, 初值为0
- 同步信号量barbers表示正在等候顾客的理发师数, 可阻塞顾客进程, 初值为1

□ 理发师问题(2/3)

• 模拟实现

```
int waiting = 0;           /*等候理发的顾客数*/
semaphore customers,barbers,mutex;
customers = 0; barbers = 1; mutex = 1;

void barber() {
    while(TRUE){
        P(customers); //试图为一位顾客服务，如果没有他就睡觉（进程阻塞）
        P(mutex); //如果有顾客，这时理发师进程被唤醒，互斥修改空椅子的数量
        waiting = waiting - 1;
        V(mutex); /*释放椅子互斥量，使得进店的顾客可以访问椅子的数量以决定是否进店等待*/
        cut_hairs();
        V(barbers); //唤醒其他顾客
    }
}
```


□ 理发师问题(3/3)

• 模拟实现（续）

```
void customer_i() {  
    P(mutex); //想坐到一把椅子上  
    if (waiting < N){ //如果还有空着的椅子的话  
        waiting = waiting + 1; //顾客坐到一张椅子上了  
        V(mutex); //顾客已经坐在椅子上等待了，访问椅子结束，释放互斥量  
        V(customers); //通知理发师，有一位顾客来了  
        P(barbers); /*该这位顾客理发了，如果理发师还在忙，那么他就等着（顾客进程阻塞）*/  
        get_haircut();  
    }else //没有空着的椅子  
        V(mutex); //释放被锁定的椅子，离开  
}
```

CQ1.3.1 在下列同步机制中，可以实现让权等待的是（ ）。

- ☐ A Peterson方法
- ☐ B swap指令
- ☒ C 信号量方法
- ☐ D TestAndSet指令

提交

CQ1.3.2 进程从运行态到等待态可能是由于（ ）。

- ☐ A 进程调度程序的调度
- ☐ B 当前运行进程的时间片用完
- ☒ C 当前运行的进程执行了P操作
- ☐ D 当前运行的进程执行了V操作

提交

CQ1.3.3 对于两个并发进程，设互斥信号量为mutex（初值为1），若 $\text{mutex} = -1$ ，则（ ）。

- ☐ A 表示没有进程进入临界区
- ☐ B 表示有一个进程进入临界区
- ☒ C 表示有一个进程进入临界区，另一个进程等待进入
- ☐ D 表示有两个进程进入临界区

提交

CQ1.3.4 如果有4个进程共享同一程序段，则每次允许3个进程进入该程序段，若用P、V操作作为同步机制，则信号量的取值范围是（ ）。

- ☐ A -1~4
- ☐ B -2~2
- ☒ C -1~3
- ☐ D -3~2

提交

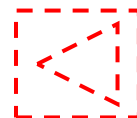
CQ1.3.5 在9个生产者、6个消费者共享8个单元缓冲区的生产者/消费者问题中，互斥使用缓冲区的信号量的初始值为（ ）。

- ☒ A 1
- ☐ B 6
- ☐ C 8
- ☐ D 9

提交

CQ1.3.6 对信号量X执行V操作时，若（ ）则唤醒等待队列中的队首进程，使之进入就绪队列排队。

- ☐ A $X+1 < 0$
- ☒ B $X+1 \leq 0$
- ☐ C $X+1 > 0$
- ☐ D $X+1 \geq 0$



提交

CQ1.3.7 用P、V操作实现进程同步时，信号量的初值为（ ）

- ☐ A -1
- ☐ B 0
- ☐ C 1
- ☒ D 由用户确定

提交

CQ1.3.8 若信号量S的初值为3，当前值为-2，则表示有（ ）个等待进程。

☒ A 2个

☐ B 3个

☐ C 4个

☐ D 5个

提交

CQ1.3.9 若一个系统中共有5个并发进程涉及某个相同的变量A，则变量A相关的临界区是由（ ）个临界区构成。

- ☐ A 1
- ☐ B 3
- ☒ C 5
- ☐ D 6

提交

CQ1.3.10 下列关于PV操作的说法中，正确的是（ ）。

- I PV操作是一种系统调用命令
- II PV操作是一种低级进程通信原语
- III PV 操作由一个不可被中断的过程组成
- IV PV 操作由两个不可被中断的过程组成

- ☐ A III
- ☒ B II、IV
- ☐ C I、II、IV
- ☐ D I、IV

提交

内容纲要

Contents Page



1 进程同步

2 进程通信

2.1 概述

2.2 管道

2.3 共享内存

2.4 消息传递

3 死锁



Inter-Process Communication, IPC

□ 进程通信概念

- 协作进程间的信息交换。

□ 分类

- 低级进程通信：效率低，主要针对控制信息的传送。典型实例为信号量机制。 semaphore
- 高级进程通信：能以较高速率传输大量数据；进程通信实现细节由操作系统提供，整个通信过程对用户透明，通信程序编制简单，分为以下三类：

-- 管道通信机制 pipe

-- 共享内存通信机制 shared memory

-- 消息传递通信机制 message passing

□ 何谓管道？

- 管道(Pipe)是用于连接读进程和写进程以实现它们之间通信的共享文件(shared file)，是UNIX的传统进程通信方式。
- 管道是单向的，发送进程（即写进程）视管道文件为输出文件，以字符流的形式把大量数据送入管道；接收进程（即读进程）是管道文件为输入文件，从管道中接收数据。

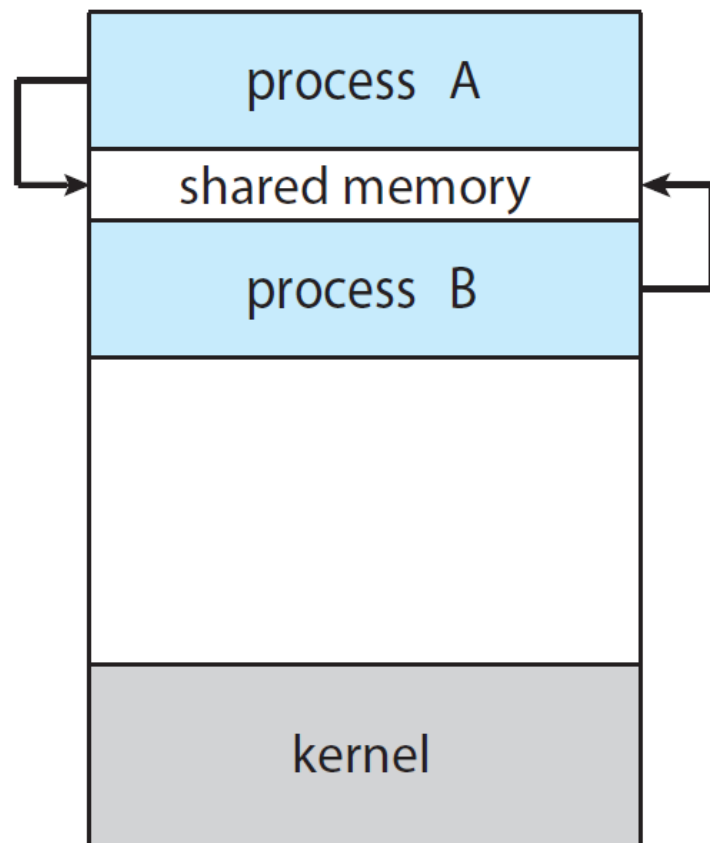


□ 管道的实现

- 管道通信借助于文件系统的机制实现，包括管道文件的创建、打开、关闭和读写。
- 管道通信机制还要提供读、写进程间的同步：
 - 双方必须知道对方是否存在，只有确定对方存在时，才能进行通信。
 - 互斥：一次只能一个进程读/写管道。
 - 同步：
 - 1) 当写进程把一定数量的数据写入管道，便去睡眠等待，直到读进程取走数据后，再把它唤醒；
 - 2) 当读进程读一空管道时，也应睡眠等到，直到写进程将数据写入管道后，才将它唤醒。

□ 原理

互相通信的协作进程建立一块共享的内存区域，进程通过向此共享区域读出或写入数据来交换信息。



□ 实现

一个进程首先在自己的地址空间中创建一块内存区作为通信使用，而其余进程则将该块内存区映射到自己的虚存空间。各进程通过读写自己虚拟内存空间中对应的共享内存区实现通信。

`shmget()`: 获得共享内存并返回shm_id

`#include <sys/shm.h>`

`shmat()`: 将共享内存映射到自己的进程空间

`shmdt()`: 解除映射

`shmctl()`: 对共享内存进行控制，如删除

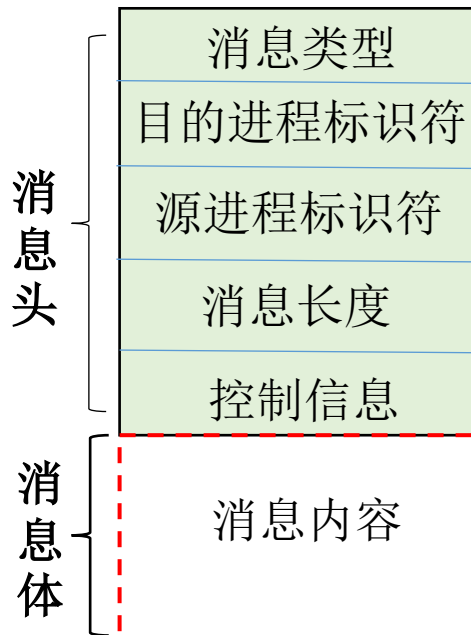
□ 优缺点

- IPC机制中最快捷的通信机制
- 未提供同步机制，需要借助加锁、信号量等机制

□ 原理

进程间的数据交换以**消息(message)**为单位，程序员直接利用系统提供的一组**通信原语**来实现通信。

□ 消息的格式：



消息传递方式

直接（同步）通信

一个进程可以在任何时刻向另一个进程发送或者请求一条消息。

`send(receiver, message)`

`receive(sender, message)`

间接（异步）通信

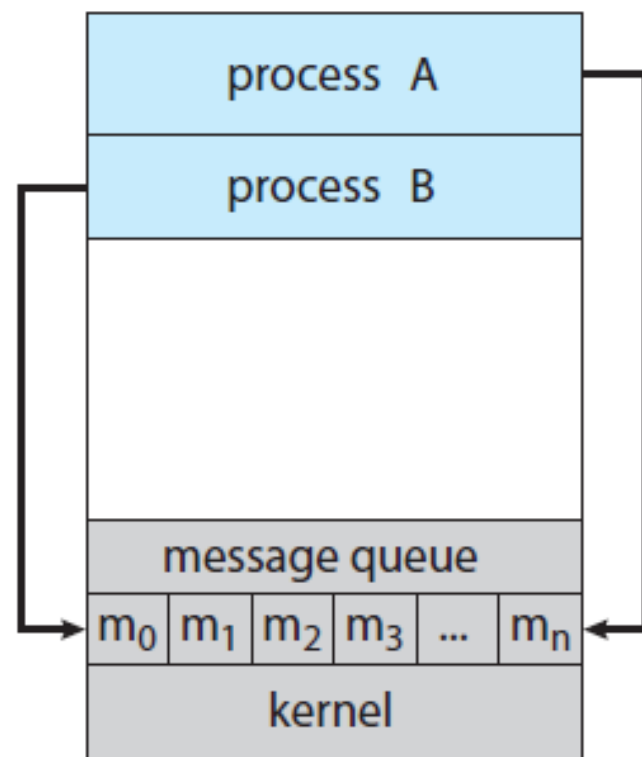
借助于收发双方进程之外的共享数据结构作为通信中转，该共享的内存称为**信箱**。

`send(mailbox, message)`

`receive(mailbox, message)`

优缺点

- 方便、灵活
- 效率



CQ2.1 信箱通信是一种（ ）通信方式。

- ☐ A 直接
- ☒ B 间接
- ☐ C 低级
- ☐ D 信号量

提交

CQ2.2 在消息传递通信机制中，消息队列属于（ ）资源。

- ☒ A 临界
- ☐ B 共享
- ☐ C 永久
- ☐ D 可剥夺

提交

内容纲要

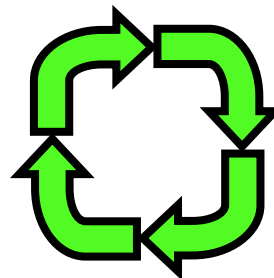
Contents Page



1 进程同步

2 进程通信

3 死锁



3.1 死锁概述

- I 什么是死锁？
- II 死锁产生的四个必要条件
- III 死锁与饥饿
- IV 死锁的解决方案

3.2 死锁的预防

3.3 死锁的避免和银行家算法

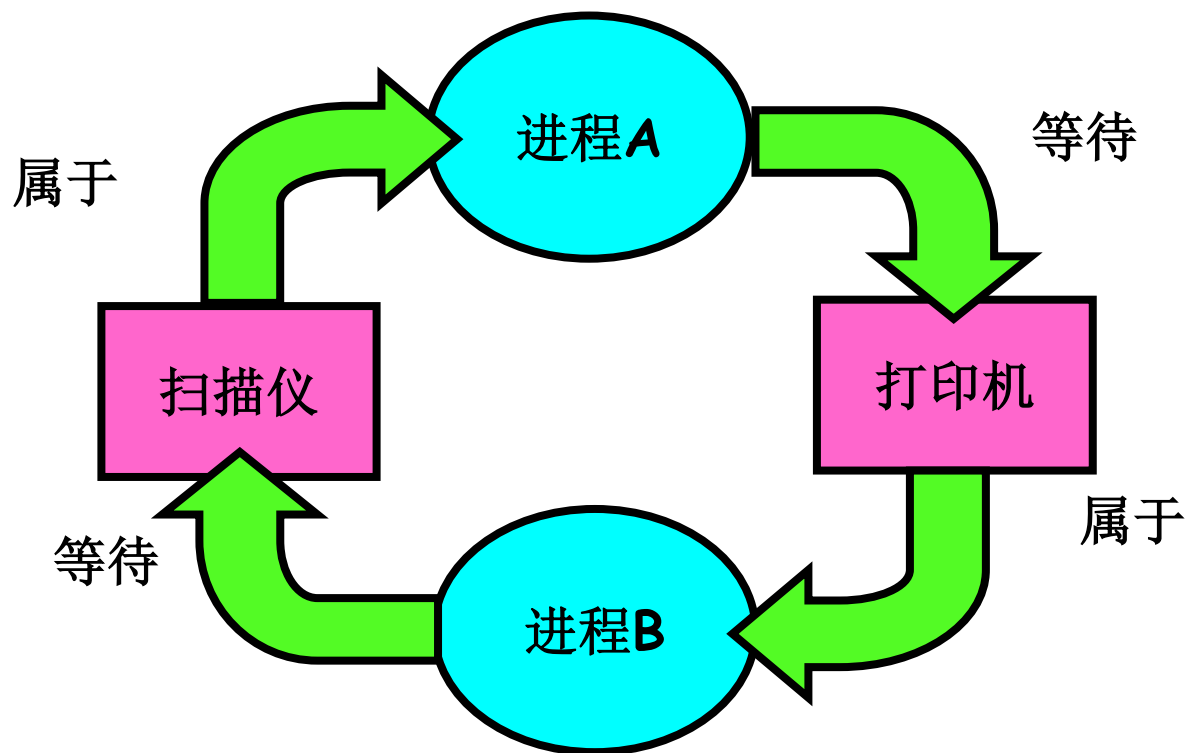


3.4 死锁的检测和恢复

什么是死锁？

❑ 死锁（Deadlock）

- 多个进程因竞争系统资源或互相通信而处于永久等待状态，若无外力作用，这些进程都将无法向前推进。



3.1 死锁概述



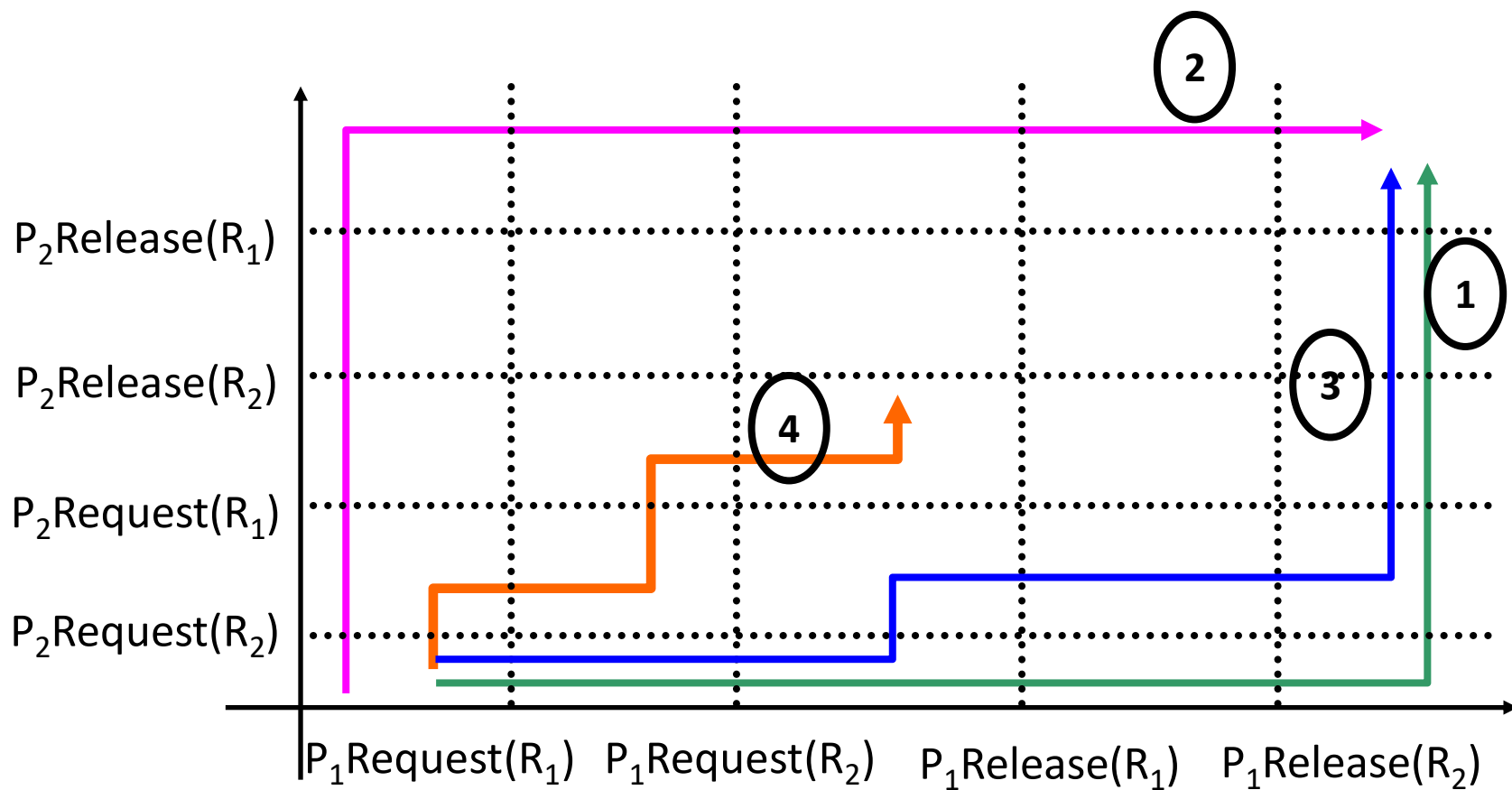
3. 死锁»

死锁的成因

(1) 系统资源不足

(2) 进程推进次序不当

死锁与进程的推进速度、资源的数量以及资源分配策略有关！



3.1 死锁概述

产生死锁的必要条件

四个条件同时成立时，死锁才可能发生；如果这四个条件不同时成立，则必然不会引发死锁！

- ❑ 互斥条件(**mutual exclusion**)

一个时刻，一个资源仅能被一个进程占有。

- ❑ 不可剥夺条件(**no preemption**)

除了资源占有进程主动释放资源，其它进程都不可抢夺其资源。

- ❑ 占有和等待条件(**hold and wait**)

请求资源未果进程虽阻塞但保持占有资源不放。

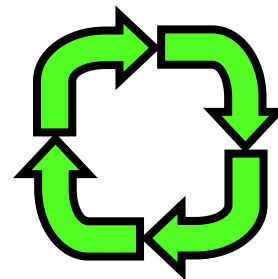
- ❑ 循环等待条件(**circular wait**)

前三个条件同时存在产生的结果

存在一个进程等待队列 $\{P_1, P_2, \dots, P_n\}$ ，其中 P_1 等待 P_2 占有的资源， P_2 等待 P_3 占有的资源， \dots ， P_n 等待 P_1 占有的资源，形成一个进程等待环路。

死锁 vs 饥饿

❑ 死锁 循环等待资源



- 死锁的进程数必须大于或等于两个；
- 处于死锁的进程必定是等待态。

❑ 饥饿 进程长时间的等待

- e.g.低优先级进程总是等待高优先级进程所占有的资源；
- 进入“饥饿”的进程可能只有一个；
- 处于“饥饿”的进程可以是一个就绪进程。

❑ 对比 死锁 \Rightarrow 饥饿 (反之不亦然)

- 饥饿可能自行解除；
- 如果无外部干涉，死锁无法终止。

死锁的解决方案

❑ 死锁的预防 (Prevention)

静态方法

- 破坏四个必要条件之一。

❑ 死锁的避免 (Avoidance)

动态方法

- 允许四个必要条件同时存在，看是否能找到一个资源分配的**安全序列**，进而决定是否分配。 **银行家算法**

❑ 死锁的检测和恢复 (Detection & Recovery)

- 允许死锁的发生，系统及时地检测死锁并解除它。



Windows和Linux等大多数通用操作系统采用的处理死锁问题的方法并没有列在上面，请大家猜一猜。

无视！

3.1 死锁概述

3.2 死锁的预防

3.3 死锁的避免和银行家算法

3.4 死锁的检测和恢复

❑ 思路

破坏产生死锁的四个必要条件之一！

❑ 可行性分析

- 破坏“互斥条件” 不可行

- 破坏“不可剥夺条件”

增加系统开销，降低系统吞吐量，仅适用于状态易于保存和恢复的资源，如CPU和内存。

- 破坏“占有和等待条件”

静态资源分配，安全且易于实现，但资源浪费、进程延迟。

- 破坏“循环等待条件”

有序资源分配，避免几个进程对资源的请求形成环路。

资源次序的不灵活性限制了新设备的增加，增加了程序员设计和实现程序的难度，浪费了系统资源。

3.1 死锁概述

3.2 死锁的预防

3.3 死锁的避免和银行家算法

3.4 死锁的检测和恢复

死锁的避免

□ 引入及描述

- 死锁的预防对资源分配加以诸多限制，降低了进程运行和资源使用的效率。
- 死锁的避免允许同时存在四个必要条件，试图在系统运行过程中避免死锁的发生。

□ 基本思路

- 系统对进程每次发出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。

一个典型的动态检查算法：银行家算法

3.3 死锁的避免和银行家算法

银行家算法初探(1/3)

□ 问题描述

- ① 银行家拥有一笔周转资金；
- ② 客户要求分期贷款，如果客户能够得到各期贷款，就一定能够归还贷款，否则就一定不能归还贷款；
- ③ 银行家应谨慎地贷款，防止出现坏账；
- ④ 银行家采用的具体方法是看他是否有足够的剩余资金满足某一个需求的客户，如此反复下去；
- ⑤ 如果所有投资最终都被收回，则该状态是安全的，最初的请求可以批准。



capital: 10



credit used

3 + 2

credit limit: 5



credit used

4 + 4

credit limit: 8



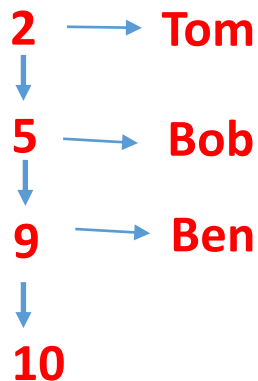
credit used

2



credit limit: 7

3. 死锁



银行家算法初探(2/3)

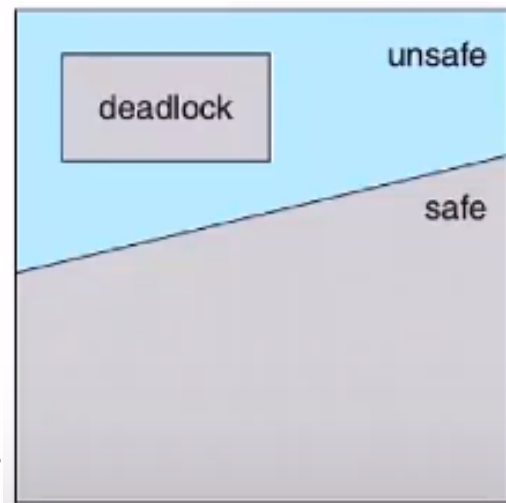
□ 安全状态

- 如果系统能按某种顺序如 $\langle P_1, P_2, \dots, P_n \rangle$ 来为每个进程分配资源，直至最大需求，使每个进程都可以顺利运行完成，则称此时的系统状态为**安全状态**，称 $\langle P_1, P_2, \dots, P_n \rangle$ 为**安全序列**。

若某一时刻系统中不存在一个安全序列，则称此时的系统状态为**不安全状态**。

□ 安全状态与死锁的关系

- 当系统进入不安全状态后，便**可能**陷入死锁，（并非所有不安全状态都是死锁状态）。
- 只要保证系统**处于安全状态**，便可避免死锁。

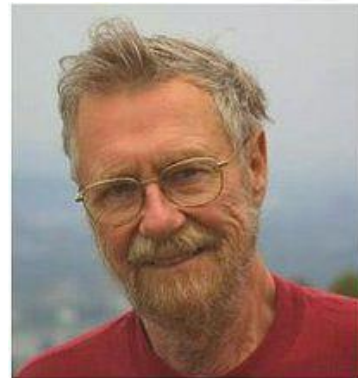


3.3 死锁的避免和银行家算法

3. 死锁»

银行家算法初探(3/3)

Banker's Algorithm



□ 提出

银行家算法是一个避免死锁的著名算法，是由荷兰学者Dijkstra在1965年为T.H.E系统设计的一种避免死锁产生的算法。它以银行借贷系统的分配策略为基础，判断并保证系统的安全运行。

□ 前提和目标

- 已知系统中所有资源的种类和数量；
- 已知进程所需各类资源的最大需求量；
- 对进程的每一次资源申请进行详细的计算，当存在安全序列时进行资源分配，确保系统始终处于安全状态。

3.3 死锁的避免和银行家算

银行家算法进阶(1/11)

□ 数据结构

- 可利用资源向量 Available

-- $Available[j]=k$: 表示系统现有 k 个 R_j 类空闲资源

- 最大需求矩阵 Max

-- $Max[i,j]=k$: 表示进程 P_i 最多需要 k 个 R_j 类资源

- 分配矩阵 Allocation

-- $Allocation[i,j]=k$: 表示进程 P_i 已有 k 个 R_j 类资源

- 需求矩阵 Need $Need(i, j) = Max(i, j) - Allocation(i, j)$

-- $Need[i,j]=k$: 表示进程 P_i 尚需 k 个 R_j 类资源

请求向量: $Request[i,j]=k$: 表示进程 P_i 请求 k 个 R_j 类资源

工作向量: $Work[j]=k$: 表示系统“可”提供 k 个 R_j 类资源

完成向量: $Finish[i]=True$: 表示进程 P_i 有足够资源完成运行 (布尔向量)

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1



银行家算法进阶(2/11)

□ 主体算法 若：进程 P_i 发出资源请求 $Request_i$

- ① 若 $Request_i \leq Need[i]$ ，则转向②，否则出错返回
- ② 若 $Request_i \leq Available$ ，则转向③，否则应使 P_i 等待并返回
- ③ 系统试探性地满足 P_i 请求，并作以下修改：

$$Available = Available - Request_i$$

$$Allocation[i] = Allocation[i] + Request_i$$

$$Need[i] = Need[i] - Request_i$$

- ④ 系统调用安全性算法进行资源分配检查，若存在安全序列，则执行分配，否则恢复试探分配前状态，并使 P_i 等待

银行家算法进阶(3/11)

□ 安全性算法

1. 令 $Work = Available$, $Finish = \mathbf{FALSE}$
2. 从进程集合中查找一个满足 $Finish[i] = \mathbf{FALSE}$ 且 $Need[i] \leq Work$ 的进程 P_i 。若找到, 则可假定 P_i 能获得所需资源并顺利执行, 故有:

$$Work = Work + Allocation[i]$$

$$Finish[i] = \mathbf{True}$$

然后重复执行第2步; 否则转至第3步执行

3. 如果所有进程 $Finish[i] = \mathbf{True}$, 则表示系统处于安全状态; 否则系统处于不安全状态

3.3 死锁的避免和银行家算法

3. 死锁»

银行家算法进阶(4/11)

□ 例1 - (1/6)

系统资源总量 Available[A,B,C] = {10, 5, 7}

进程	Max A B C	Allocation A B C	Need A B C	Work A B C	Allocation + Work A B C	Finish
P ₀	7 5 3	0 1 0	7 4 3			
P ₁	3 2 2	3 0 2	0 2 0			
P ₂	9 0 2	3 0 2	6 0 0			
P ₃	2 2 2	2 1 1	0 1 1			
P ₄	4 3 3	0 0 2	4 3 1			

- T1时刻 Available[A,B,C] = {3, 3, 2} => {2, 3, 0} ③
- 进程P1发出资源请求Request1(1,0,2) < Need1(1,2,2) ①
- 安全分配序列 ? ④ < Available(3,3,2) ②



银行家算法进阶(5/11)

□ 例1-(2/6)

系统资源总量 Available[A,B,C] = {10, 5, 7}

进程	Max <u>A B C</u>	Allocation <u>A B C</u>	Need <u>A B C</u>	Work <u>A B C</u>	Allocation + Work <u>A B C</u>	<u>Finish</u>
P ₀	7 5 3	0 1 0	7 4 3			
P ₁	3 2 2	3 0 2	0 2 0			
P ₂	9 0 2	3 0 2	6 0 0			
P ₃	2 2 2	2 1 1	0 1 1			
P ₄	4 3 3	0 0 2	4 3 1			

- T1时刻 Available[A,B,C] = {2, 3, 0}
- 安全分配序列 ?

3.3 死锁的避免和银行家算法

3. 死锁»

银行家算法进阶(6/11)

□ 例1 - (3/6)

系统资源总量 Available[A,B,C] = {10, 5, 7}

进程	Max <u>A B C</u>	Allocation <u>A B C</u>	Need <u>A B C</u>	Work <u>A B C</u>	Allocation + Work <u>A B C</u>	<u>Finish</u>
P ₀	7 5 3	0 1 0	7 4 3			
P ₁	3 2 2	3 0 2	0 2 0	2 3 0	5 3 2	① True
P ₂	9 0 2	3 0 2	6 0 0			
P ₃	2 2 2	2 1 1	0 1 1			
P ₄	4 3 3	0 0 2	4 3 1			

3.3 死锁的避免和银行家算法

3. 死锁»

银行家算法进阶(7/11)

□ 例1 - (4/6)

系统资源总量 Available[A,B,C] = {10, 5, 7}

进程	Max <u>A B C</u>	Allocation <u>A B C</u>	Need <u>A B C</u>	Work <u>A B C</u>	Allocation + Work <u>A B C</u>	<u>Finish</u>
P ₀	7 5 3	0 1 0	7 4 3			
P ₁	3 2 2	3 0 2	0 2 0	2 3 0	5 3 2	① True
P ₂	9 0 2	3 0 2	6 0 0			
P ₃	2 2 2	2 1 1	0 1 1	5 3 2	7 4 3	② True
P ₄	4 3 3	0 0 2	4 3 1			

3.3 死锁的避免和银行家算法

3. 死锁»

银行家算法进阶(8/11)

□ 例1 - (5/6)

系统资源总量 Available[A,B,C] = {10, 5, 7}

进程	Max A B C	Allocation A B C	Need A B C	Work A B C	Allocation + Work A B C	Finish
P ₀	7 5 3	0 1 0	7 4 3			
P ₁	3 2 2	3 0 2	0 2 0	2 3 0	5 3 2	① True
P ₂	9 0 2	3 0 2	6 0 0			
P ₃	2 2 2	2 1 1	0 1 1	5 3 2	7 4 3	② True
P ₄	4 3 3	0 0 2	4 3 1	7 4 3	7 4 5	③ True

银行家算法进阶(9/11)

例1 - (6/6)

系统资源总量 Available[A,B,C] = {10, 5, 7}

进程	Max A B C	Allocation A B C	Need A B C	Work A B C	Allocation + Work A B C	Finish
P ₀	7 5 3	0 1 0	7 4 3	7 4 5	7 5 5	④ True
P ₁	3 2 2	3 0 2	0 2 0	2 3 0	5 3 2	① True
P ₂	9 0 2	3 0 2	6 0 0			
P ₃	2 2 2	2 1 1	0 1 1	5 3 2	7 4 3	② True
P ₄	4 3 3	0 0 2	4 3 1	7 4 3	7 4 5	③ True

- 存在安全分配序列 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$
- 可以满足进程P₁的资源申请

银行家算法进阶(10/11)

□ 例2

系统资源总量 $Available[A,B,C] = \{10, 5, 7\}$

进程	Max <u>A B C</u>	Allocation <u>A B C</u>	Need <u>A B C</u>	Work <u>A B C</u>	Allocation + Work <u>A B C</u>	<u>Finish</u>
P ₀	7 5 3	0 1 0	7 4 3			
P ₁	3 2 2	3 0 2	0 2 0			
P ₂	9 0 2	3 0 2	6 0 0			
P ₃	2 2 2	2 1 1	0 1 1			
P ₄	4 3 3	0 0 2	4 3 1			

- T2时刻 $Available[A,B,C] = \{2, 3, 0\}$
- 进程P4发出资源请求Request₄(3,3,0) < Need₄(4,3,1)
- Request₄(3,3,0) > Available(2,3,0), 故让P4等待

银行家算法进阶(11/11)

例3

系统资源总量 Available[A,B,C] = {10, 5, 7}

进程	Max A B C	Allocation A B C	Need A B C	Work A B C	Allocation + Work A B C	Finish
P ₀	7 5 3	0 3 0	7 2 3			
P ₁	3 2 2	3 0 2	0 2 0			
P ₂	9 0 2	3 0 2	6 0 0			
P ₃	2 2 2	2 1 1	0 1 1			
P ₄	4 3 3	0 0 2	4 3 1			

- T3时刻 Available[A,B,C] = {2, 3, 0}
- 进程P₀发出资源请求Request₀(0,2,0)
 - < Need₀(7,4,3)
 - < Available(2,3,0)
- 尝试分配,

银行家算法的优缺点

□ 优点

允许死锁必要条件同时存在。

□ 缺点 缺乏实用价值

- 进程运行前就要求知道其所需要资源的最大数量；
- 要求进程是无关的，若考虑同步情况，可能会打乱安全序列；
- 要求进入系统的进程个数和资源数固定。

3.1 死锁概述

3.2 死锁的预防

3.3 死锁的避免和银行家算法

3.4 死锁的检测和恢复

死锁的检测(1/5)

□ 基本策略

- 允许死锁发生，操作系统不断监视系统进展情况，判断死锁是否发生。
- 一旦死锁发生，则采取专门的措施解除死锁，并以最小的代价恢复操作系统运行。

□ 检测时机

- 当进程等待时检测死锁（系统开销大）
- 定时检测
- 系统资源利用率下降时检测死锁

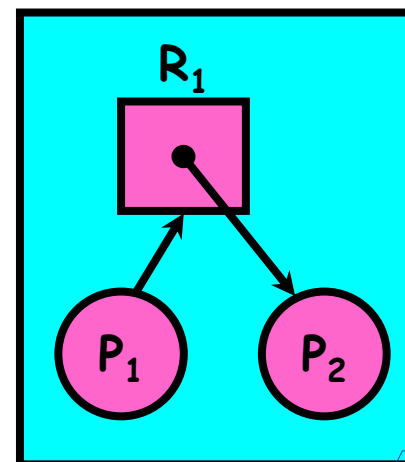
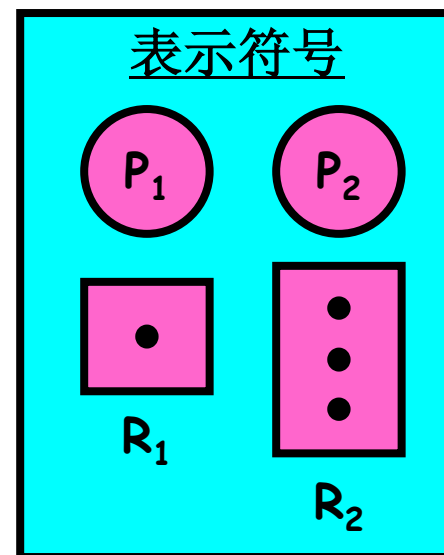
3.4 死锁的检测与恢复

3. 死锁»

死锁的检测(2/5)

▣ 进程-资源分配图的概念

- 资源类（资源的不同类型）
 - 用方框表示
- 资源实例（存在于每个资源中）
 - 用方框中的黑圆点表示
- 进程
 - 用圆圈中加进程名表示
- 申请边
 - 进程指向资源类的一条有向边， $P \rightarrow R$
- 分配边
 - 资源实例指向进程的一条有向边， $R \rightarrow P$

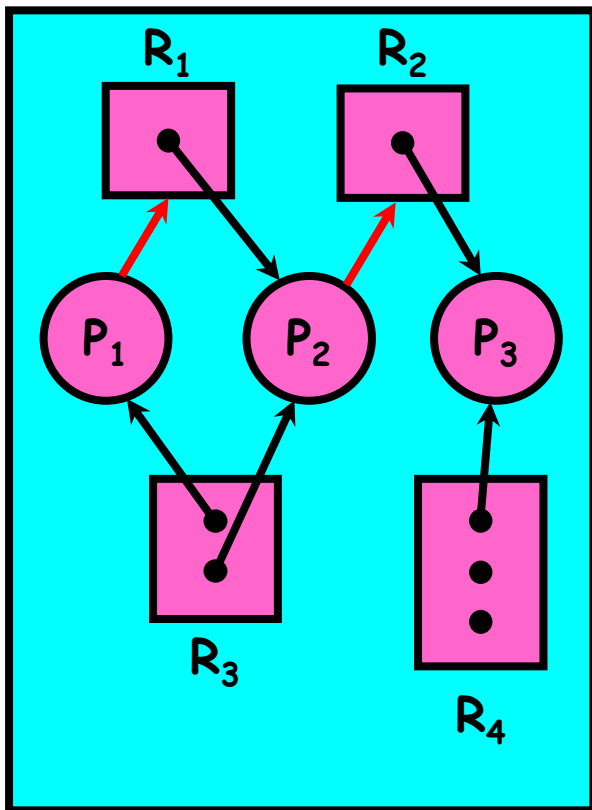


3.4 死锁的检测与恢复

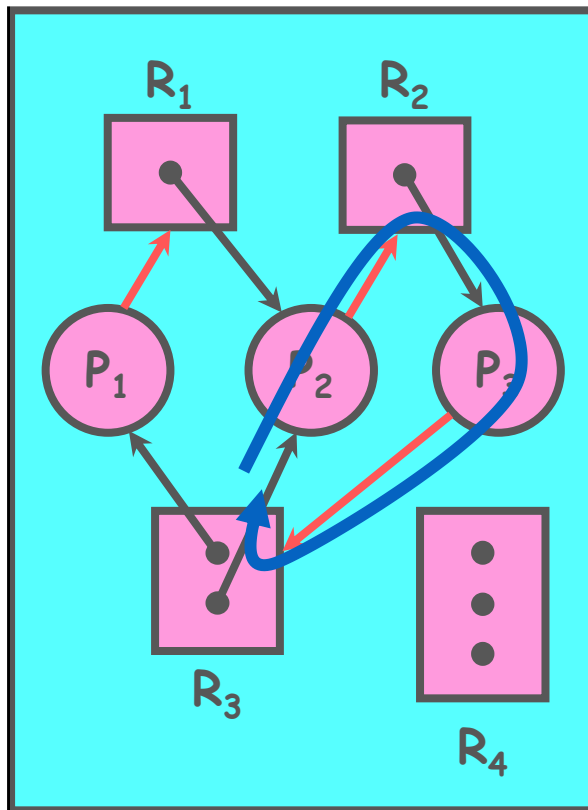
3. 死锁»

死锁的检测(3/5)

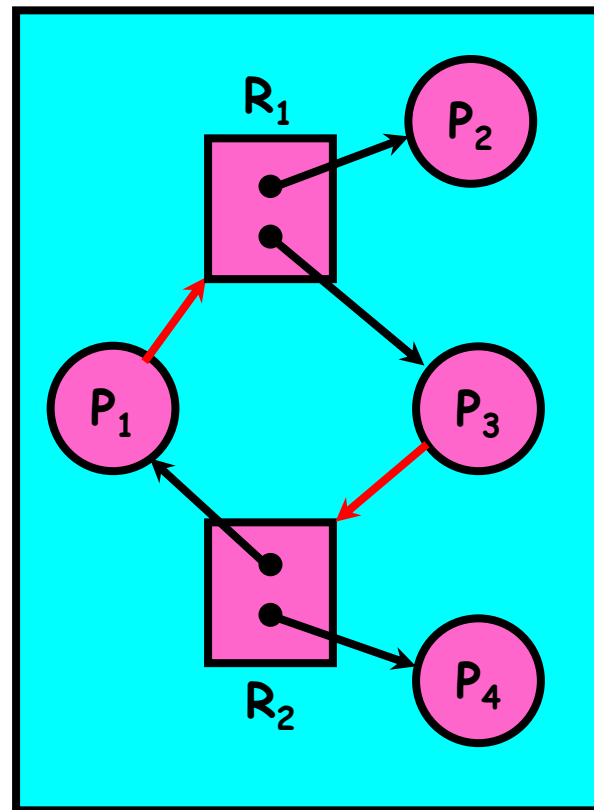
□ 进程-资源分配图举例



无环无死锁



有环死锁



有环但无死锁

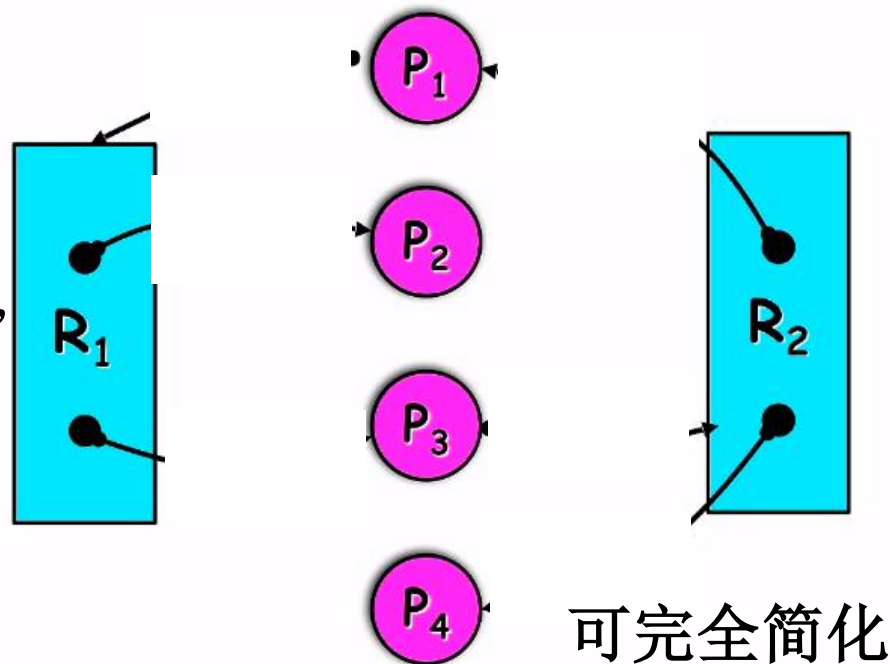
3.4 死锁的检测与恢复

3. 死锁

死锁的检测(4/5)

□ 进程-资源分配图的化简

- 不断寻找图中没有资源请求或资源请求能够得到满足的进程节点，消除其分配边和申请边。
- 如果能消去某进程的所有请求边和分配边，则称该进程为孤立结点。
- 如果经一系列简化，使所有进程都成为孤立结点，则该图是可完全简化的；否则则称该图是不可完全简化的。



□ 死锁定理

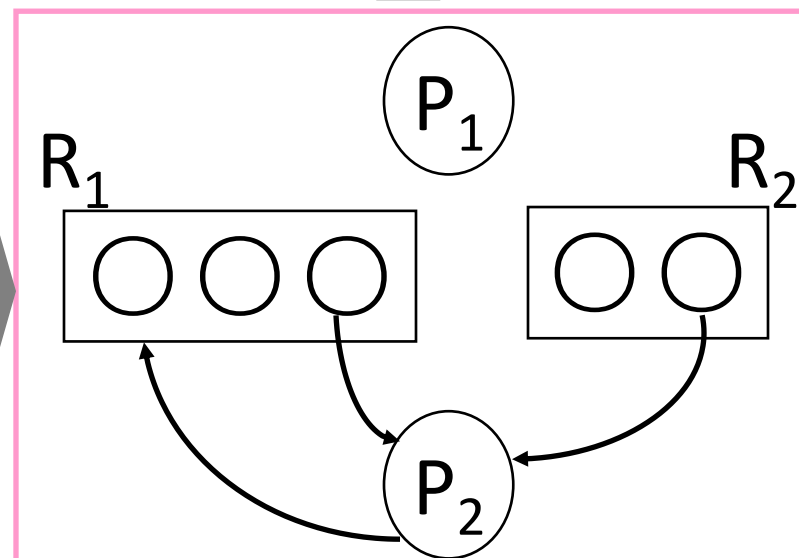
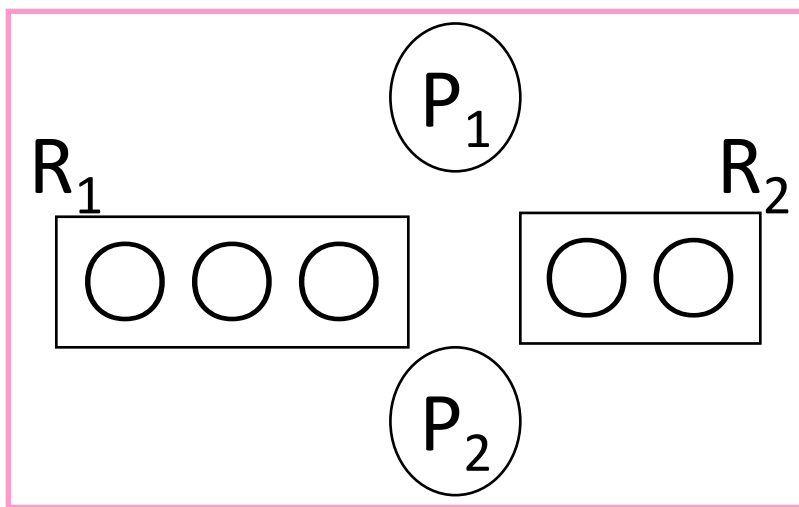
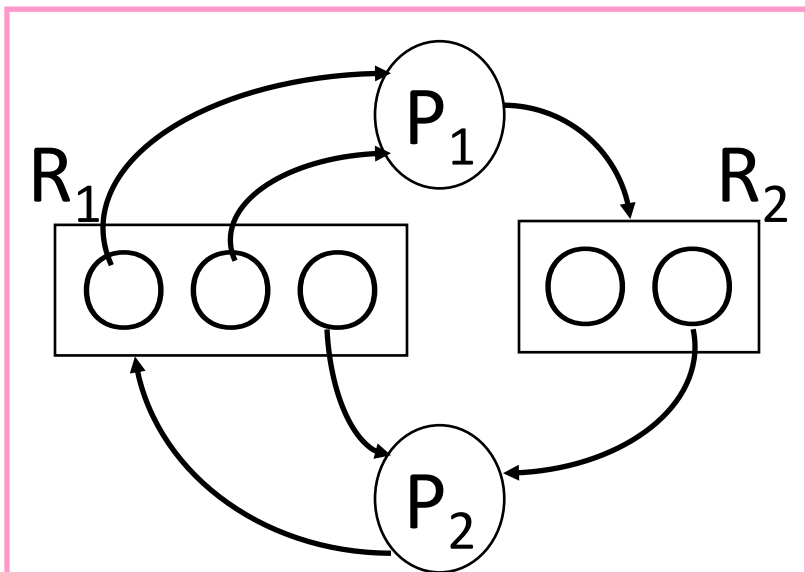
系统为死锁状态的充分条件是当且仅当该状态的“进程—资源分配图”是不可完全简化的。

3.4 死锁的检测与恢复

3. 死锁»

死锁的检测(5/5)

可完全简化
不存在死锁！



死锁的恢复

- ❑ 撤销进程，强制回收资源
- ❑ 剥夺资源，但不中止进程
- ❑ 进程回滚(roll back)
- ❑ 重新启动

Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems!

CQ3.1 操作系统中，死锁出现是指（ ）

- ☐ A 计算机系统发生重大故障
- ☐ B 进程同时申请的资源数远远超过资源总数
- ☒ C 若干进程因竞争资源而无限等待其他进程释放已占有的资源
- ☐ D 资源个数远远小于进程数

提交

CQ3.2 产生死锁的必要条件是：互斥、（ ）、循环等待和不剥夺。

- ☐ A 请求与阻塞
- ☒ B 请求与保持
- ☐ C 请求与释放
- ☐ D 释放与阻塞

提交

CQ3.3 死锁的预防是根据（ ）而采取措施实现的。

- ☐ A 配置足够的系统资源
- ☐ B 使进程的推进顺序合理
- ☒ C 破坏死锁的四个必要条件之一
- ☐ D 防止系统进入不安全状态

提交

CQ3.4 资源的有序分配策略可以破坏死锁的（ ）条件。

- ☐ A 互斥
- ☐ B 请求与保持
- ☐ C 不剥夺
- ☒ D 循环等待

提交

CQ3.5 银行家算法在解决问题中是用于（ ）的。

- ☐ A 预防死锁
- ☐ B 检测死锁
- ☒ C 避免死锁
- ☐ D 解除死锁

提交

CQ3.6 死锁定理是用于（ ）的方法。

- ☐ A 预防死锁
- ☐ B 避免死锁
- ☒ C 检测死锁
- ☐ D 解除死锁

提交

CQ3.7 某系统中有11台打印机，N个进程共享打印机资源，每个进程要求3台。当N的取值不超过（ ）时，系统不会发生死锁。

- ☐ A 4
- ☒ B 5
- ☐ C 6
- ☐ D 7

提交

CQ3.8 某系统中有3个并发进程，都需要同类资源4个，试问该系统不会发生死锁的最少资源数是（ ）。

- ☐ A 9
- ☒ B 10
- ☐ C 11
- ☐ D 12

提交

本章总结(1/2)

□进程的互斥

任一时刻只能由一个进程进入**临界区**，为了实现这个目标，可以用软件方法和硬件指令的方法，统称这种技术叫“**锁**”。

□进程的同步

为了调节各进程之间的执行速度，引入**信号量**和**PV操作**让进程在各自的**等待点**上等待信号的到来。

互斥访问是一种特殊的同步，所以信号量也适用于解决资源的竞争和互斥访问。

本章总结(2/2)

□ 进程间除了通过信号量交互外，还有如下机制：

- 管道
- 共享内存
- 消息传递

□ 死锁及其解决方案

- 预防死锁：破坏产生死锁的四个必要条件之一
- 避免死锁：利用银行家算法，每次进程申请资源时都尝试找出一个进程执行序列保证不会发生死锁。
- 检测死锁：在适当的时机检测系统中是否发了死锁，如果发生了则采取措施解除它。检测的方法有：
 - 化简进程—资源分配图

Thank You

H a v e A N i c e D a y

南京邮电大学计算机学院、
软件学院、网络空间安全学院
