



经典教材《计算机操作系统》**最新版**

第4章 进程同步

主讲教师：陆丽萍



 **人民邮电出版社**
POSTS & TELECOM PRESS



第4章知识导图

第1章 操作系统引论

第2章 进程的描述与控制

第3章 处理机调度与死锁

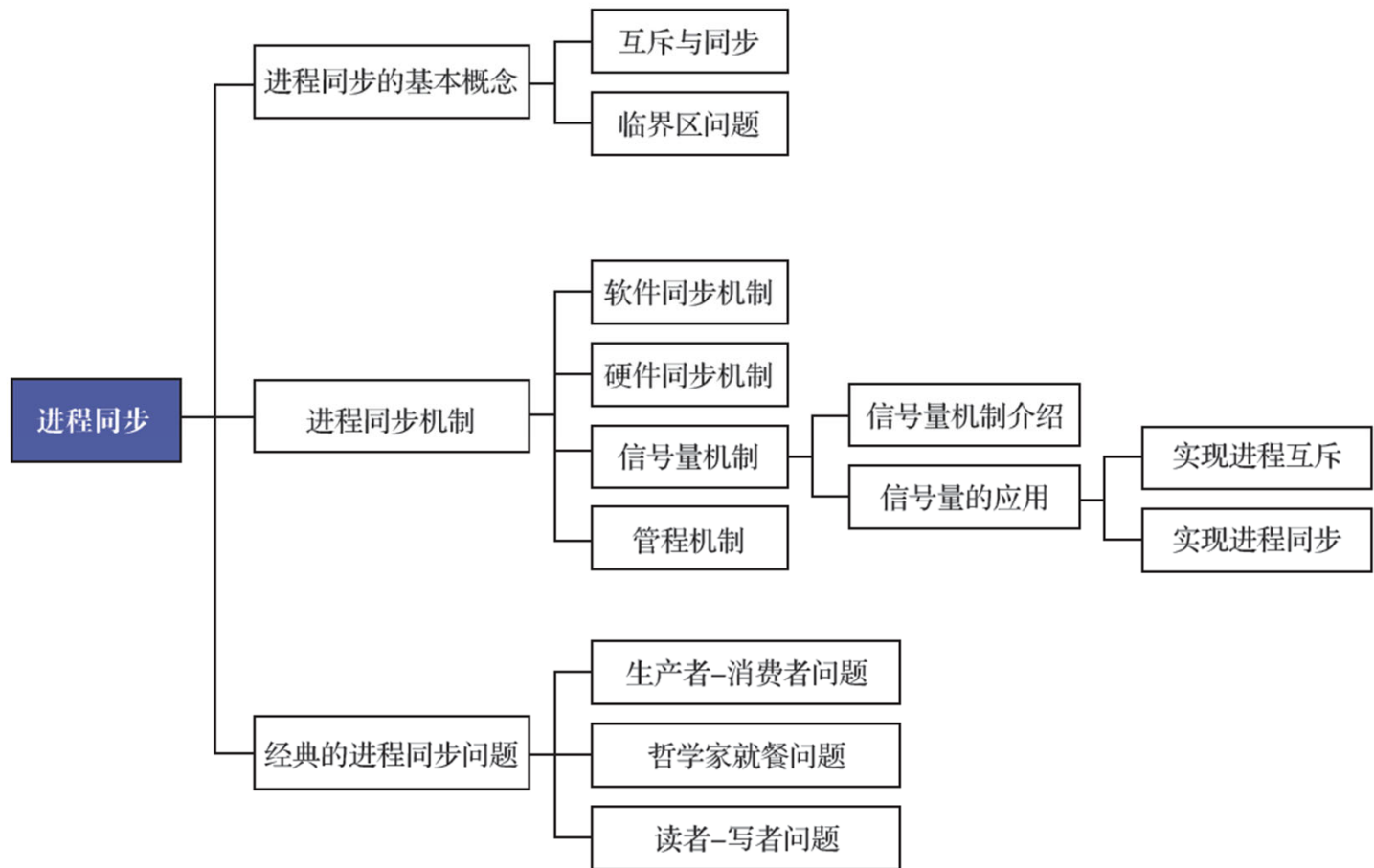
第4章 进程同步

第5章 存储器管理








第6章 虚拟存储器

第7章 输入/输出系统

第8章 文件管理



内容导航:

-  4.1 进程同步的概念
-  4.2 软件同步机制
-  4.3 硬件同步机制
-  4.4 信号量机制
-  4.5 管程机制
-  4.6 经典进程的同步问题
-  4.7 Linux进程同步机制

第4章 进程同步



进程同步的概念



主要任务

- 使并发执行的诸进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性。



进程间的制约关系

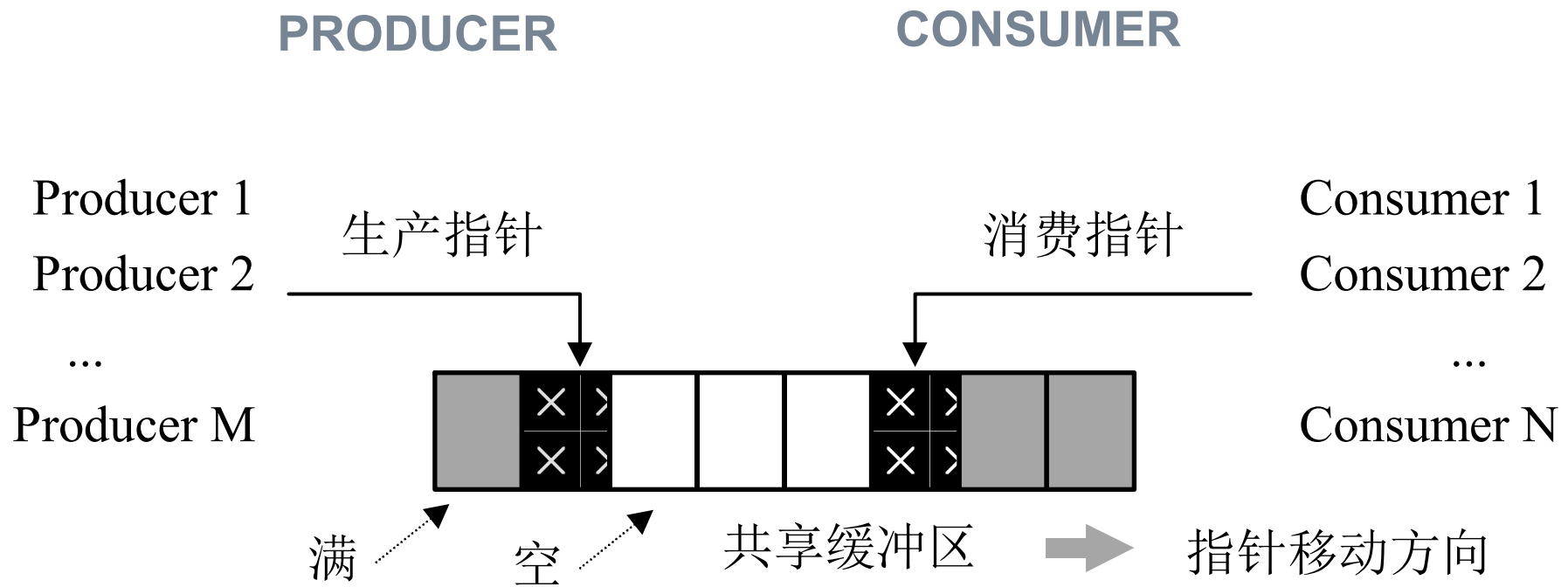
- 间接相互制约关系(互斥关系)
 - 进程互斥使用临界资源
- 直接相互制约关系 (同步关系)
 - 进程间相互合作



临界资源

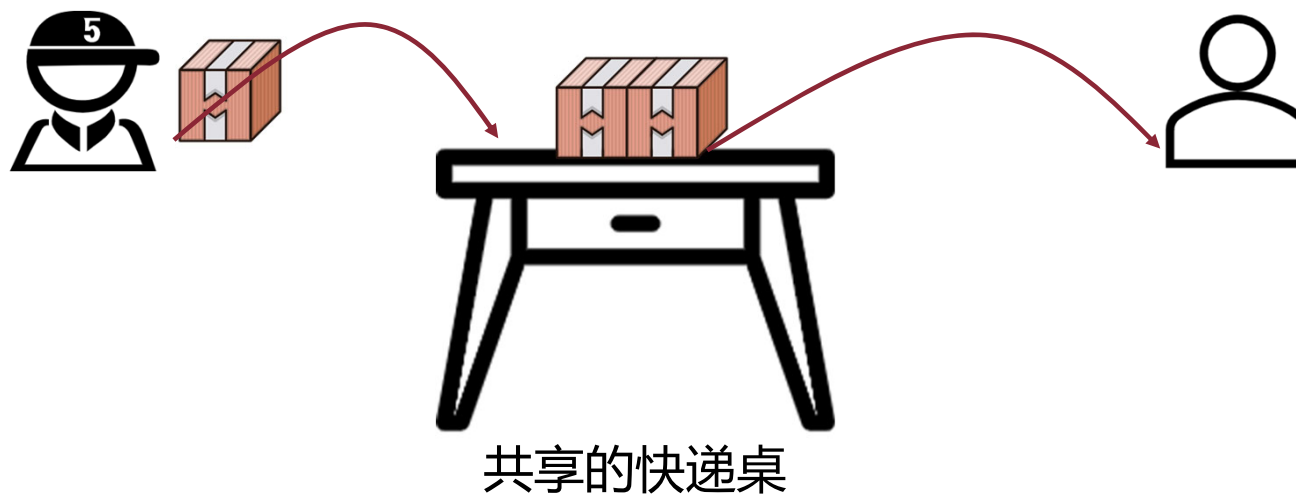
- 系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源或共享变量。
- 诸进程间应采取互斥方式，实现对这种资源的共享。

OS 数据不一致例子：生产者-消费者问题



回顾：新冠疫情下快递员送货问题

- 小区门口的桌子上允许临时放置快递
- 快递员在桌上放置快递，小明从桌上取快递



送货（生产者消费者）问题的基础实现

- 基础实现: 生产者(快递员)

```
while (true) {  
    /* Produce an item */  
    while (prodCnt - consCnt == BUFFER_SIZE)  
        ; /* do nothing -- no free buffers */  
    buffer[prodCnt % BUFFER_SIZE] = item;  
    prodCnt = prodCnt + 1;  
}
```

当没有空间时，发送者盲目等待
(快递员等待桌子有空闲空间)

发送者放置消息
(快递员将快递放在桌上空闲空间)

送货（生产者消费者）问题的基础实现

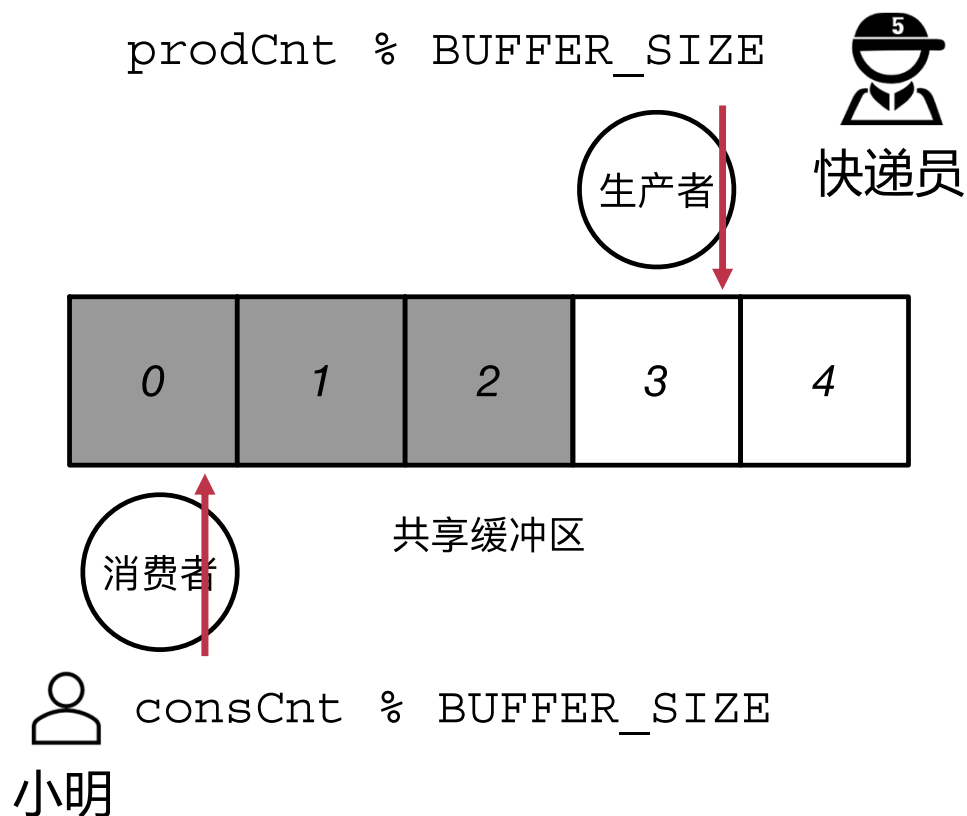
- 基础实现: 消费者(小明)

当没有新消息时，接收者盲目等待
(小明盲目查看桌上状态和等待)

```
while (true) {  
    while (prodCnt == consCnt)  
        ; /* do nothing */  
    item = [consCnt % BUFFER_SIZE] ;  
    consCnt = consCnt + 1;  
}
```

接收者获取消息
(小明拿到最先到达的一个快递)

送货（生产者消费者）问题方案总结



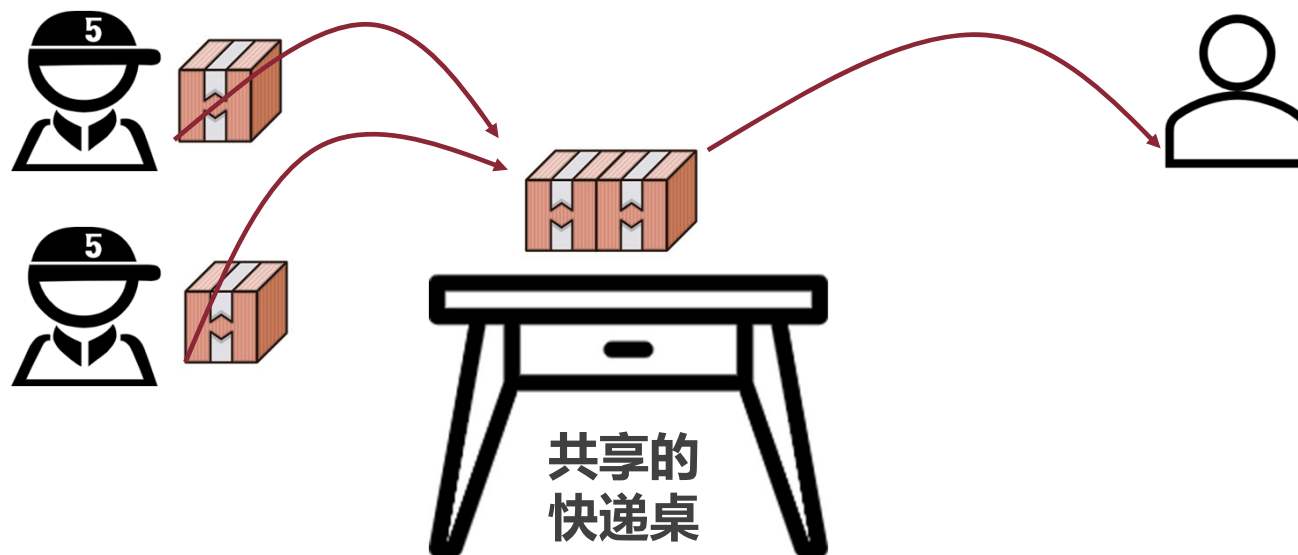
通过两个计数器来协调

生产者（快递员）与消费者（小明）

问：如果有多个生产者和消费者呢？

多生产者消费者问题

假设同一时刻有多个生产者：

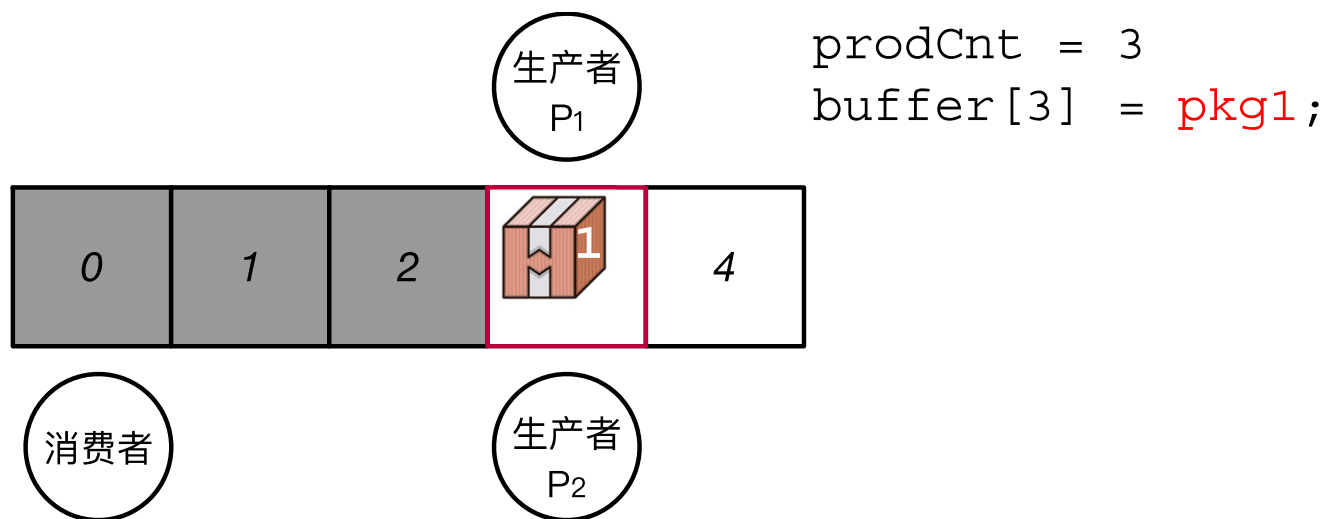


多生产者消费者问题

考虑这样一个**执行流程**

*其他流程也是可能的

```
while (prodCnt - consCnt == BUFFER_SIZE)
    ;    /* do nothing -- no free buffers */
buffer[prodCnt % BUFFER_SIZE] = item;
prodCnt = prodCnt + 1;
```

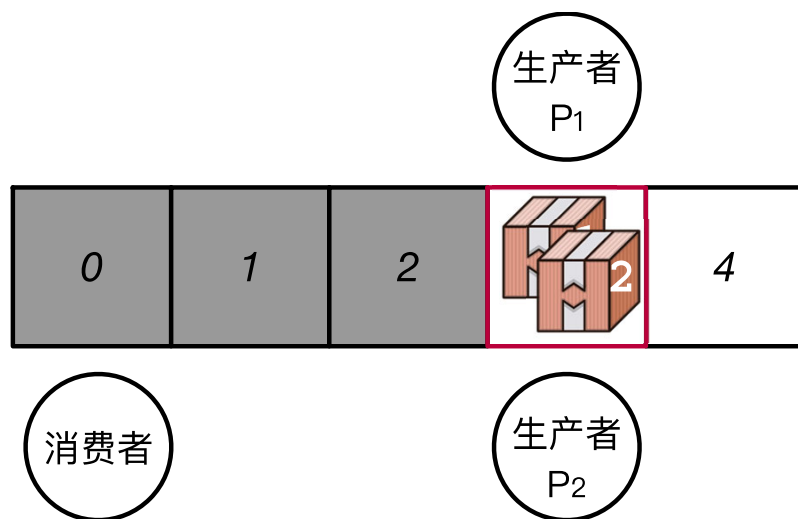


多生产者消费者问题

考虑这样一个**执行流程**

*其他流程也是可能的

```
while (prodCnt - consCnt == BUFFER_SIZE)
    ; /* do nothing -- no free buffers */
buffer[prodCnt % BUFFER_SIZE] = item;
prodCnt = prodCnt + 1;
```



prodCnt = 3
buffer[3] = pkg1;

prodCnt = 3
buffer[3] = pkg2;

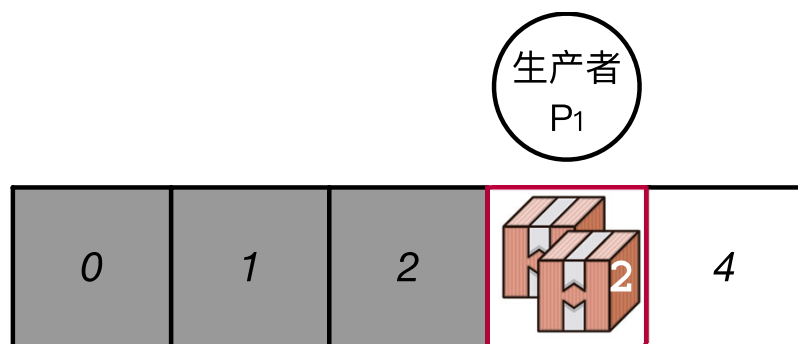
将快递碰到地上

多生产者消费者问题

考虑这样一个**执行流程**

*其他流程也是可能的

```
while (prodCnt - consCnt == BUFFER_SIZE)
    ; /* do nothing -- no free buffers */
buffer[prodCnt % BUFFER_SIZE] = item;
prodCnt = prodCnt + 1;
```



生产者
P₁

prodCnt = 3
buffer[3] = pkg1;
prcdCnt = 4

消费者

生产者
P₂

prodCnt = 3
buffer[3] = pkg2;

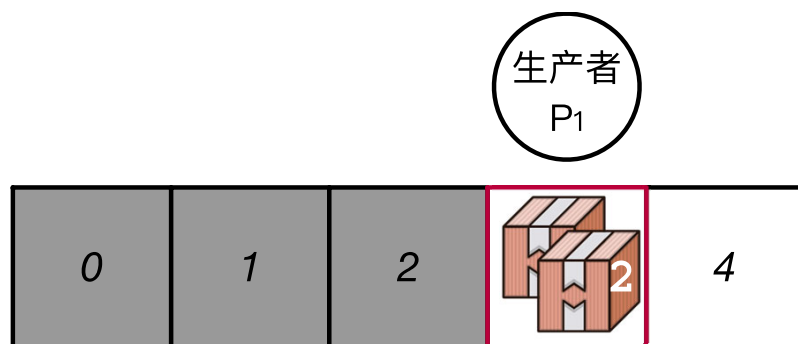
将快递碰到地上

多生产者消费者问题

考虑这样一个**执行流程**

*其他流程也是可能的

```
while (prodCnt - consCnt == BUFFER_SIZE)
    ; /* do nothing -- no free buffers */
buffer[prodCnt % BUFFER_SIZE] = item;
prodCnt = prodCnt + 1;
```



prodCnt = 3
buffer[3] = pkg1;
prodCnt = 4

prodCnt = 3
buffer[3] = pkg2;
prodCnt = 5

将快递碰到地上

如何确保他们**不会**将新产生的数据放入到同一个缓冲区中，造成**数据覆盖**？

缓冲池buffer:
用数组来表示具有n个缓冲区的缓冲池

输入指针in/**prodCnt** :
指示下一个可投放产品的缓冲区, 每当**生产者进程**生产并投放一个产品后, 输入指针加1, 初值为0



输出指针out/**consCnt** :
指示下一个可获取产品的缓冲区, 每当**消费者进程**取走一个产品后, 输出指针加1, 初值为0

整型变量count/
 $\text{prodCnt} - \text{consCnt}$:
初值为0, 表示缓冲区中的产品个数



生产者进程: producer()

```
void producer( ) {  
    while(1){  
        produce an item in nextp;  
        ...  
        while (count == n)  
            ; // do nothing  
        // add an item to the buffer  
        buffer[in] = nextp;  
        in = (in + 1) % n;  
        count++;  
    }  
}
```




消费者进程: consumer ()

```
void consumer() {  
    while(1){  
        while (count == 0)  
            ; // do nothing  
        // remove an item from the buffer  
        nextc = buffer[out];  
        out = (out + 1) % n;  
        count --;  
        consumer the item in nextc;  
        ...  
    }  
}
```



count++和count--的机器语言

■ count++:

- R1=count; 1
- R1=R1+1; 2
- count=R1; 3

■ count--:

- R2=count; 4
- R2=R2-1; 5
- count=R2; 6

Count初值为4,并发执行produce()和consume()进程

执行次序	结果	是否正确
123456	4	是
142536	3	否
145263	5	否

解决方法: 下列语句必须被原子性地执行

counter++;

counter--;



临界区

- 临界区：进程中涉及临界资源的代码段
- 进入区：用于检查是否可以进入临界区的代码段。
- 退出区：将临界区正被访问的标志恢复为未被访问标志。
- 剩余区：其他代码

一个访问临界资源的循环进程的描述

```
While(TRUE){
```

进入区

临界区

退出区

剩余区

```
}
```

临界资源

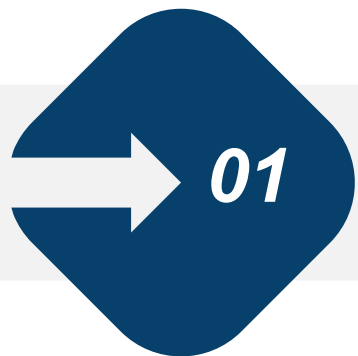
```
item nextConsumed;  
while (1) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;
```

临界区



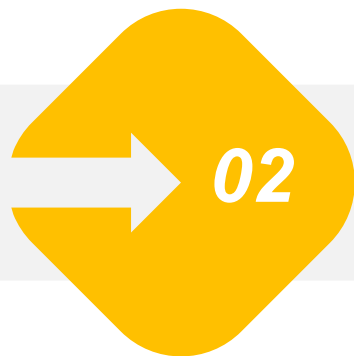
同步机制应遵循的准则

空闲让进



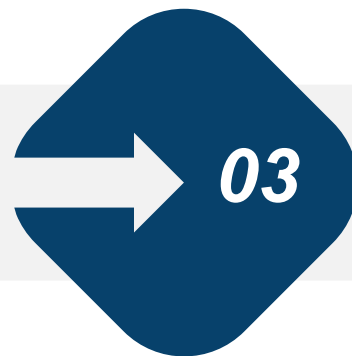
- 当无进程处于临界区，应允许一个请求进入临界区的进程立即进入自己的临界区；

忙则等待



- 已有进程处于其临界区，其它试图进入临界区的进程必须等待；

有限等待



- 等待进入临界区的进程不能“死等”；

让权等待



- 不能进入临界区的进程，应释放CPU（如转换到阻塞状态）

解决临界区问题的三个要求

- **互斥访问**：在同一时刻，**有且仅有一个进程**可以进入临界区。 **(一个快递员放快递)**
- **有限等待**：当一个进程申请进入临界区之后，必须在**有限的时间**内获得许可进入临界区而不能无限等待。 **(快递员还要送其他的)**
- **空闲让进**：当没有进程在临界区中时，必须在申请进入临界区的进程中选择一个进入临界区，保证执行临界区的**进展**。 **(没人放快递时要选一个快递员去放)**

```
while(TRUE) {
```

申请进入临界区

临界区部分

通知退出临界区

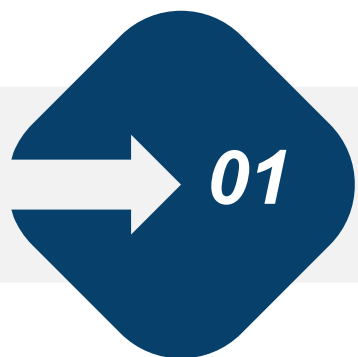
其他代码

```
}
```



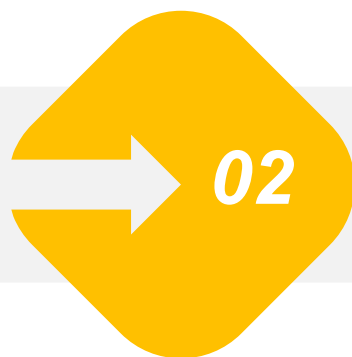
进程同步机制

软件同步机制



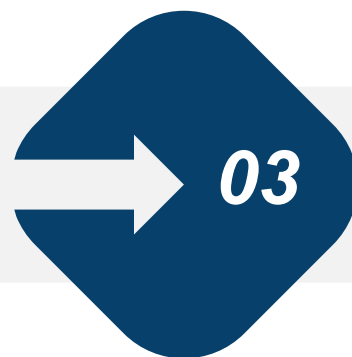
- 使用编程方法解决临界区问题
- 有难度、具有局限性，现在很少采用

硬件同步机制



- 使用特殊的硬件指令，可有效实现进程互斥

信号量机制





- 一种有效的进程同步机制，已被广泛应用

管程机制



- 新的进程同步机制

内容导航:

-  4.1 进程同步的概念
-  **4.2 软件同步机制**
-  4.3 硬件同步机制
-  4.4 信号量机制
-  4.5 管程机制
-  4.6 经典进程的同步问题
-  4.7 Linux进程同步机制

第4章 进程同步

软件解决方案：皮特森算法



flag[2] : 用于记录进程0或进程1是否申请进入临界区

turn : 用于决定如果两个进程都希望进入临界区时谁能进入临界区

进程-0

进程-1

申请进入临界区:

设置状态:

flag[0] = true

flag[1] = true

??? ???

等待条件:

通知退出临界区:

flag[0] = false

flag[1] = false

软件解决方案：皮特森算法



flag[2] : 用于记录进程0或进程1是否申请进入临界区

turn : 用于决定如果两个进程都希望进入临界区时谁能进入临界区

进程-0

进程-1

申请进入临界区:

设置状态:

flag[0] = true

flag[1] = true

等待条件:

flag[1] == true

flag[0] == true

通知退出临界区:

flag[0] = false

flag[1] = false

软件解决方案：皮特森算法



flag[2] : 用于记录进程0或进程1是否申请进入临界区

turn : 用于决定如果两个进程都希望进入临界区时谁能进入临界区

进程-0

进程-1

申请进入临界区:

设置状态:

flag[0] = true

flag[1] = true

等待条件:

flag[1] == true

flag[0] == true

够了吗? 如果进程0判断之前进程1也设置了?

通知退出临界区:

flag[0] = false

flag[1] = false

软件解决方案：皮特森算法



flag[2] : 用于记录进程0或进程1是否申请进入临界区

turn : 用于决定如果两个进程都希望进入临界区时谁能进入临界区

进程-0

进程-1

申请进入临界区:

设置状态:

flag[0] = true

flag[1] = true

turn = 1

turn = 0

等待条件:

flag[1] == true

flag[0] == true

&& turn = 1

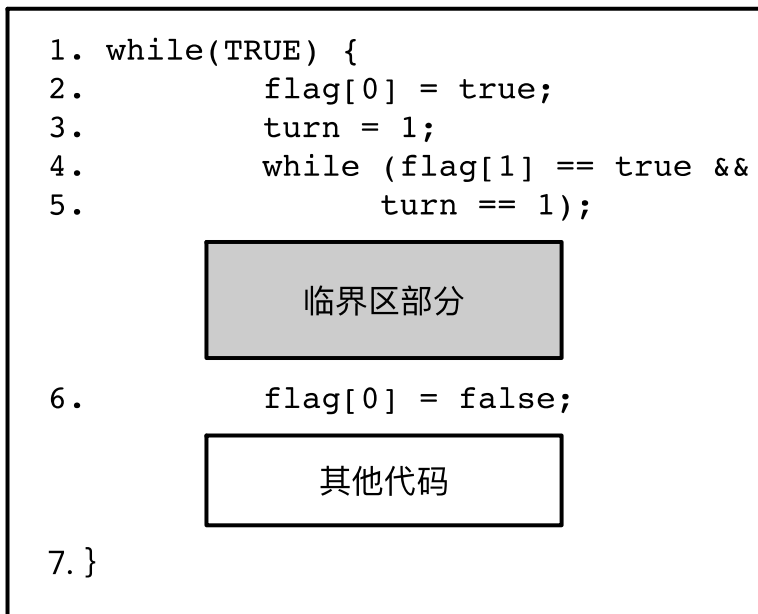
&& turn = 0

思考：为何turn要设置为对方？

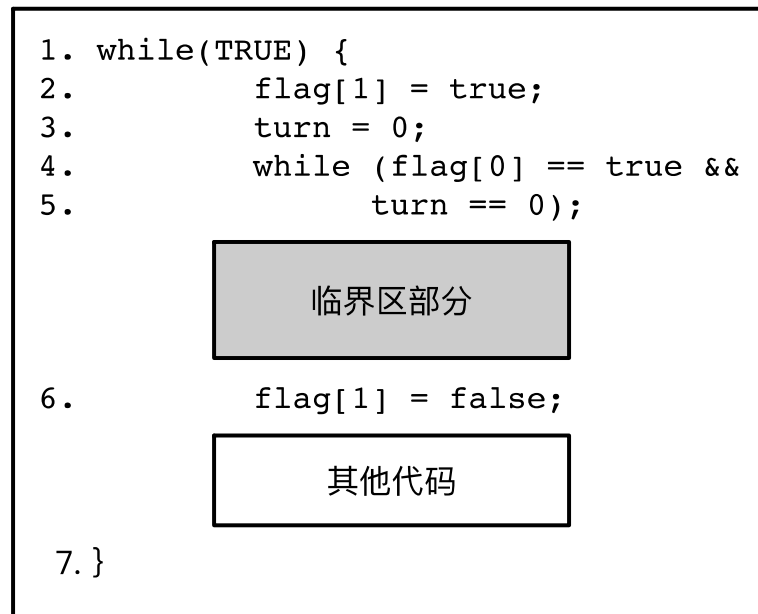
以进程0为例，只有当flag[1] == false（即线程1没有申请进入临界区）或turn == 0（即线程1也申请进入临界区，但turn决定线程0可以进入临界区）

软件解决方案：皮特森算法

线程 - 0



线程 - 1



思考：是否满足解决临界区问题的三个必要条件？

互斥访问

有限等待

空闲让进

软件解决方案：皮特森算法

	进程-0	进程-1
申请进入临界区:		
设置状态:	<code>flag[0] = true</code> <code>turn = 0</code>	<code>flag[1] = true</code> <code>turn = 1</code>
等待条件:	<code>flag[1] == true</code> <code>&& turn = 1</code>	<code>flag[0] == true</code> <code>&& turn = 0</code>

思考：如果turn要设置为自己



软件同步机制-Peterson解决方案

共享变量:








```
int turn = 0;  
turn == i ; //表示 Pi 能进入临界区  
j = 1-i;  
boolean flag[2];  
flag [0] = flag [1] = false;    //初值  
flag [i] = true ; //表示Pi 准备进入临界区
```

满足三个需求；解决了两个进程的临界区问题。

进程 P_i

```
do {  
    flag [i]:= true;  
    turn = j;  
    while (flag [j] and turn = j) ;  
        临界区  
    flag [i] = false;  
        剩余区  
} while (1);
```

内容导航:

-  4.1 进程同步的概念
-  4.2 软件同步机制
-  **4.3 硬件同步机制**
-  4.4 信号量机制
-  4.5 管程机制
-  4.6 经典进程的同步问题
-  4.7 Linux进程同步机制

第4章 进程同步



硬件同步机制



关中断

- 进入锁测试之前关闭中断，完成锁测试并上锁之后才打开中断
- 可有效保证互斥，但存在许多缺点

```
while(TRUE) {
```

申请进入临界区

临界区部分

通知退出临界区

其他代码

```
}
```

这样能解决临界区问题吗？

关闭所有核心的中断

可以解决单个CPU

核上的临界区问题

如果在多个核心中，

关闭中断不能阻塞

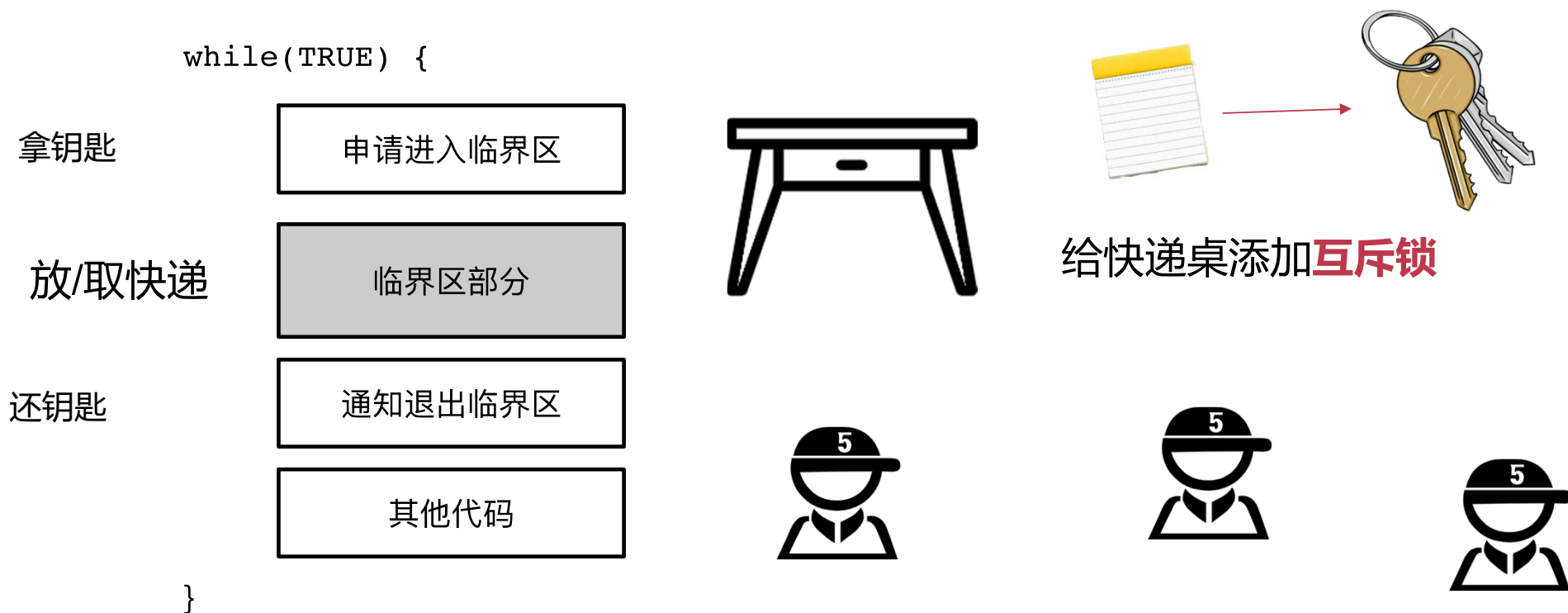
开启所有核心的中断

其他进程执行

并不能阻止多个CPU核同时进入临界区

有没有更简单的方法？

如何确保他们**不会**将新产生的数据放入到同一个缓冲区中，造成**数据覆盖**？



互斥锁

设临界区的类名为 S 。为了保证每一次临界区中只能有一个程序段被执行，又设锁定位 $key[S]$ 。 $key[S]$ 表示该锁定位属于类名为 S 的临界区。加锁后的临界区程序描述如下：

$key[S] = 1$ 时表示类名为 S 的临界区可用，
位 $key[S] = 0$ 时表示类名为 S 的临界区不可用。



```
while(TRUE) {
```

申请进入临界区

```
lock(key[S]);
```

lock操作

临界区部分

通知退出临界区

```
unlock(key[S]);
```

unlock操作: $key[S] = 1$

其他代码

```
}
```

多个核心同时执行这个操作会产生什么问题？

上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

lock操作

一种简便的实现方法是：

```
lock(key[S])
{
test: if(key[S]==0)
        goto test;    //测试锁标志
    else // key[S]==1
        key[S]==0;    //上锁
}
```

这种实现方法是不能保证并发进程互斥执行所要求的准则(3)的。因为当同时有几个进程调用lock(key[S])时，在x←0语句执行之前，可能已有两个以上的多个进程由于key[S]=1而进入临界区。为解决这个问题有些机器在硬件中设置了“测试与设置指令，保证第一步和第二步执行不可分离。



硬件同步机制



Test-and-Set指令

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。



Swap指令

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。



Test-and-Set指令实现互斥

➤ 原子地检查和修改字的内容

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

➤ 共享数据:

```
boolean lock = FALSE;
```

➤ 进程 P_i

```
do {
    while (TestAndSet(lock)) ;
    critical section
    lock = FALSE;
    remainder section
}
```



Swap指令实现互斥

➤ 原子地交换两个变量

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

➤ 共享数据 (初始化为 false):

```
boolean lock;  
boolean waiting[n];
```

➤ 进程 P_i

```
do {  
    key = true;  
    do {  
        Swap(lock, key);  
    } while (key != false)  
    临界区  
    lock = false;  
    剩余区  
} while(true)
```









硬件同步机制



缺点:

- 不符合“让权等待”原则，浪费CPU时间
- 很难解决复杂的同步问题

内容导航:

-  4.1 进程同步的概念
-  4.2 软件同步机制
-  4.3 硬件同步机制
-  **4.4 信号量机制**
-  4.5 管程机制
-  4.6 经典进程的同步问题
-  4.7 Linux进程同步机制

第4章 进程同步



信号量机制



1965年，由荷兰学者迪科斯彻Dijkstra提出（P、V分别代表荷兰语的Proberen (test)和Verhogen (increment)），是一种卓有成效的进程同步机制。

信号量-软件解决方案：

- 保证两个或多个代码段不被并发调用
- 在进入关键代码段前，进程必须获取一个信号量，否则不能运行
- 执行完该关键代码段，必须释放信号量
- 信号量有值，为正说明它空闲，为负说明其忙碌



类型

- 整型信号量
- 记录型信号量
- AND型信号量
- 信号量集



艾兹格·W·迪科斯彻(Edsger Wybe Dijkstra)

- 信号量和PV原语发明者
- 解决了“哲学家就餐”问题
- 最短路径算法(SPF)和银行家算法的创造者
- 结构程序设计之父
- THE操作系统设计者和开发者



与D. E. Knuth并称为这个时代最伟大的计算机科学家





- 信号量S-整型变量
- 提供两个不可分割的[原子操作]访问信号量

wait(S):

```
while s<=0 ; /*do no-op*/
```

```
s:=s-1;
```

signal(S):

```
s:=s+1;
```

- Wait(s)又称为P(S)
- Signal(s)又称为V(S)
- 缺点：进程忙等



记录型信号量:去除忙等的信号量

每个信号量S除一个整数值S.value外, 还有一个进程等待队列S.list, 存放阻塞在该信号量的各个进程PCB

- 信号量只能通过初始化和两个标准的原语PV来访问 - - 作为OS核心代码执行, 不受进程调度的打断
- 初始化指定一个非负整数值, 表示空闲资源总数 (又称为"资源信号量") - - 若为非负值表示当前的空闲资源数, 若为负值其绝对值表示当前等待临界区的进程数

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



wait、signal操作定义

```
wait(semaphores *S) {  
    S->value --;  
    if (S->value < 0) block(S->list)  
}
```

//请求一个单位的资源

//资源减少一个

//进程自我阻塞

```
signal(semaphores *S)  
{  
    S->value++;  
    if (S->value <= 0) wakeup(S->list);  
}
```

//释放一个单位资源

//资源增加一个

//唤醒等待队列中的一个进程



AND型信号量



AND型信号量同步的基本思想：将进程在整个运行过程中需要的所有资源，一次性全部分配给进程，待进程使用完后再一起释放。



对若干个临界资源的分配，采用原子操作。



在wait(S)操作中增加了一个“AND”条件，故称之为AND同步，或同时wait(S)操作，即Swait(Simultaneous wait)。



AND型信号量操作定义

```
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  
        }  
    }  
}
```

```
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.  
        }  
    }  
}
```



信号量集



在记录型信号量中，wait或signal仅能对某类临界资源进行一个单位的申请和释放，当需要对N个单位进行操作时，需要N次wait/signal操作，效率低下



扩充AND信号量：对进程所申请的所有资源以及每类资源不同的资源需求量，在一次P、V原语操作中完成申请或释放

- 进程对信号量 S_i 的测试值是该资源的分配下限值 t_i ，即要求 $S_i \geq t_i$ ，否则不予分配。一旦允许分配，进程对该资源的需求值为 d_i ，即表示资源占用量，进行 $S_i = S_i - d_i$ 操作
- $Swait(S1, t1, d1, \dots, Sn, tn, dn)$
- $Ssignal(S1, d1, \dots, Sn, dn)$
- $Swait(S, d, d)$ 。此时在信号量集中只有一个信号量 S ，但允许它每次申请 d 个资源，当现有资源数少于 d 时，不予分配。
- $Swait(S, 1, 1)$ 。此时的信号量集已蜕化为一般的记录型信号量($S > 1$ 时)或互斥信号量($S = 1$ 时)
- $Swait(S, 1, 0)$ 。这是一种特殊且很有用的信号量操作。当 $S \geq 1$ 时，允许多个进程进入某个特定的临界区；当 $S = 0$ 时，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。



信号量的应用



利用信号量实现**进程互斥**

➤ 设置互斥信号量



利用信号量实现**前趋关系**



利用信号量实现**进程同步**

➤ 设置同步信号量



利用信号量实现进程互斥

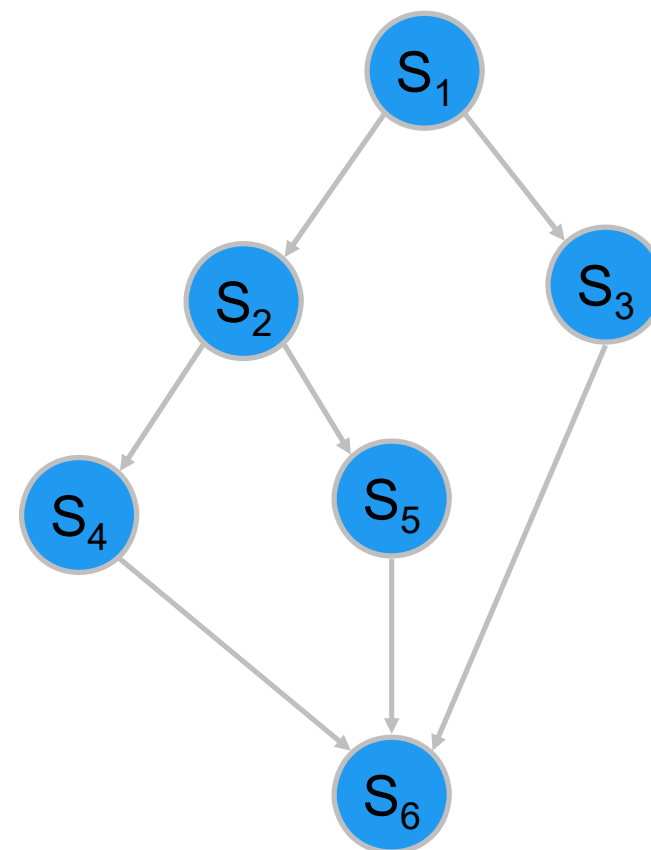
```
semaphore mutex;  
mutex=1; // 初始化为 1
```

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



利用信号量实现前趋关系

```
main(){  
    Semaphore a,b,c,d,e,f,g;  
    a.value=0;b.value=0;c.value=0;  
    d.value=0;e.value=0;f.value=0;g.value=0;  
    cobegin  
        { S1;signal(a);signal(b); }  
        { wait(a);S2;signal(c) ;signal(d);}  
        { wait(b);S3;signal(e); }  
        { wait(c);S4;signal(f); }  
        { wait(d);S5;signal(g); }  
        { wait(e);wait(f);wait(g);S6; }  
    corend  
}
```



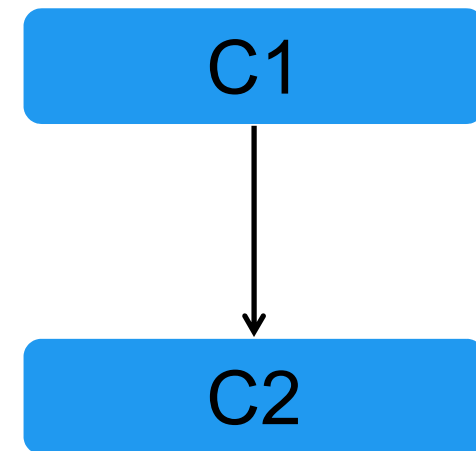


利用信号量实现进程同步







- 实现各种同步问题
- 例子：P1和 P2 需要代码段 C1 比C2先运行
semaphores s=0; //主要用于传递消息

```
P1()  
{  
    C1;  
    signal(s);  
    ...  
}
```

```
P2()  
{  
    ...  
    wait(s);  
    C2;  
}
```



内容导航:

-  4.1 进程同步的概念
-  4.2 软件同步机制
-  4.3 硬件同步机制
-  4.4 信号量机制
-  **4.5 管程机制**
-  4.6 经典进程的同步问题
-  4.7 Linux进程同步机制

第4章 进程同步



信号量机制的问题



问题

- 需要程序员实现，编程困难
- 维护困难
- 容易出错
 - wait/signal位置错
 - wait/signal不配对



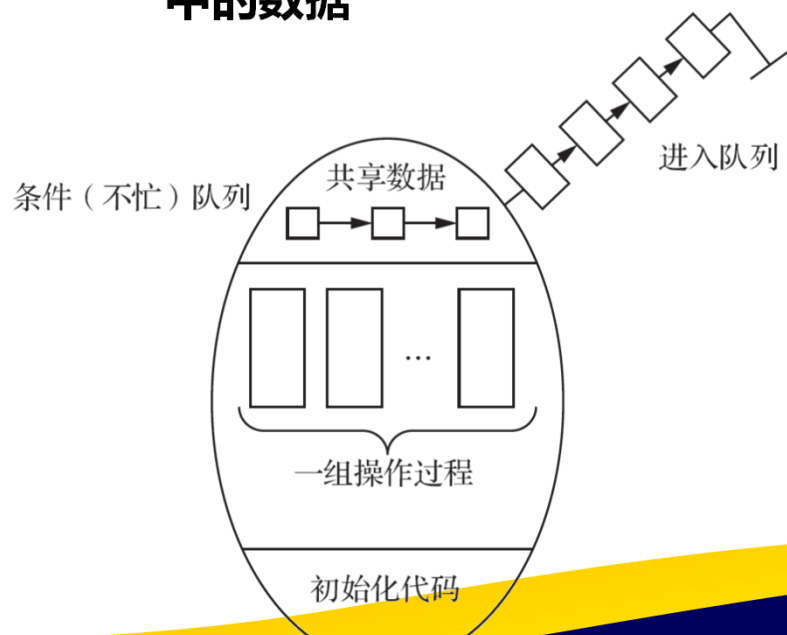
解决方法

- 由**编程语言**解决同步互斥问题
- 管程 (1970s, Hoare和Hansen)

信号量：**分散式**
管 程：**集中式**

OS 管程定义

- 一个管程定义了一个**数据结构**和能为并发进程所执行（在该数据结构上）的**一组操作**，这组操作能**同步进程**和**改变管程中的数据**



OS 语法描述如下：

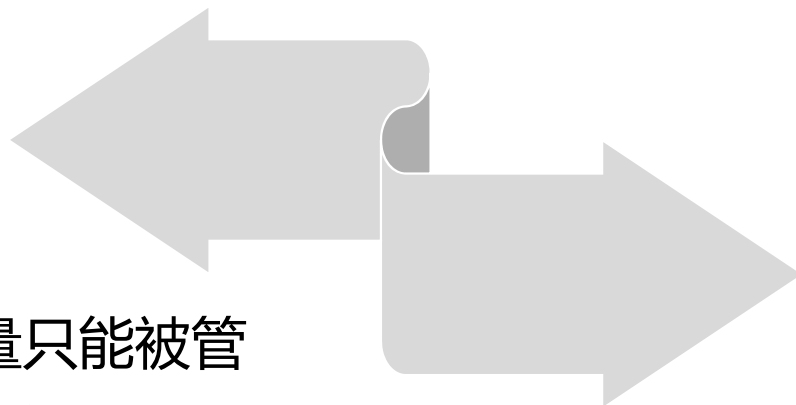
```

Monitor monitor_name {
    share variable declarations; /*①④管程名*/
    cond declarations; /*② 共享变量说明 */
    public: /*条件变量说明*/
        void P1(.....) {.....} /*③能被进程调用的过程*/
        void P2(.....) {.....}
        .....
        void (.....) {.....}
        .....
    {
        initialization code;
        .....
    }
}
    
```



互斥

- 管程中的变量只能被管程中的操作访问
- 任何时候只有一个进程在管程中操作
- 类似临界区
- 由编译器完成



同步

- 条件变量
- 唤醒和阻塞操作


```
init(){
    S=5;//初始资源=5
    condition x;
}
// 申请一个资源
take_away(){
    if(S<=0)x.wait();
    S--;//可用资源数-1
}
// 归还一个资源
give_back(){
    S++;//可用资源数+1
    if(有进程在等待)x.signal;
}
```

当一个进程进入管程后被阻塞，直到阻塞被解除的这个期间，如果进程不释放管程，那么其他进程就无法进入管程。

为了解决这种特殊情况，我们将阻塞的原因定义为条件变量 condition，通常引起阻塞的原因可以有多个，因此在管程中设置了多个条件变量。每个条件变量保存了一个等待队列，当发生阻塞的时候，将进程插入到相应的队列，并释放管程，这样就可以避免上述说的那种情况了。



condition x, y;



条件变量的操作

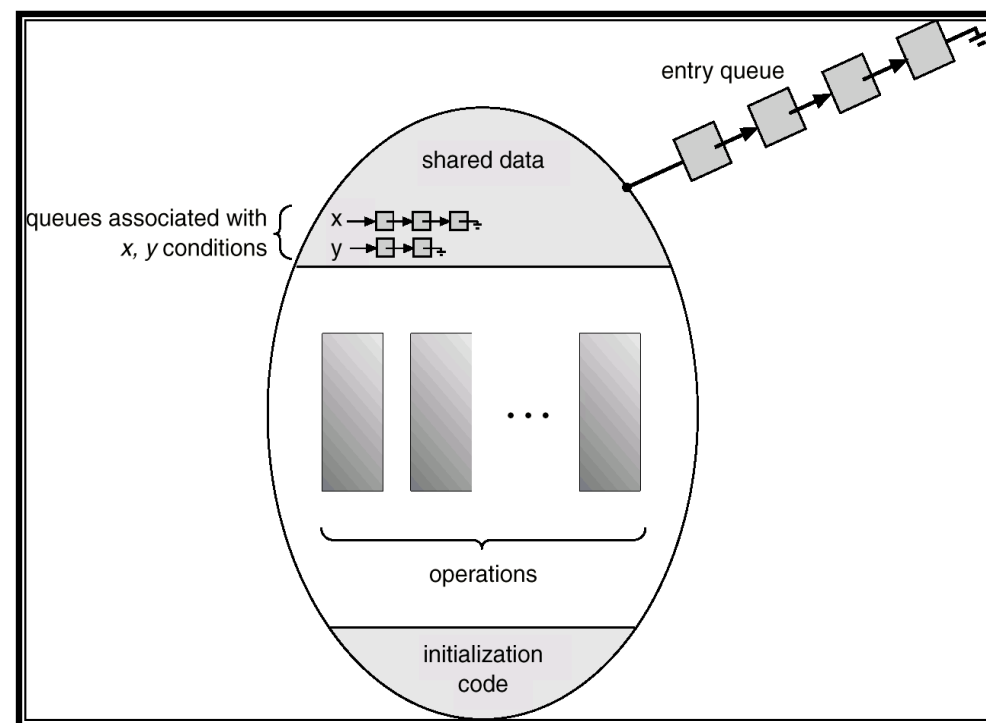
- 阻塞操作: wait
- 唤醒操作: signal



x.wait(): 进程阻塞直到另外一个进程调用x.signal()



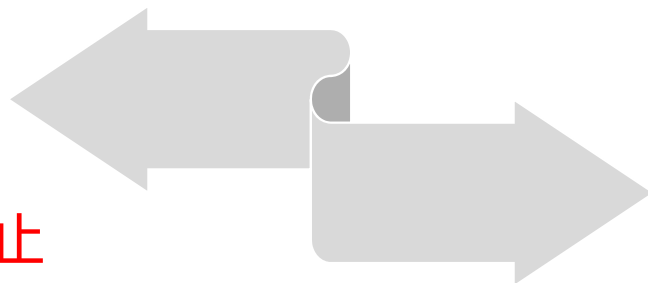
x.signal(): 唤醒另外一个进程





管程内可能存在不止
1个进程。

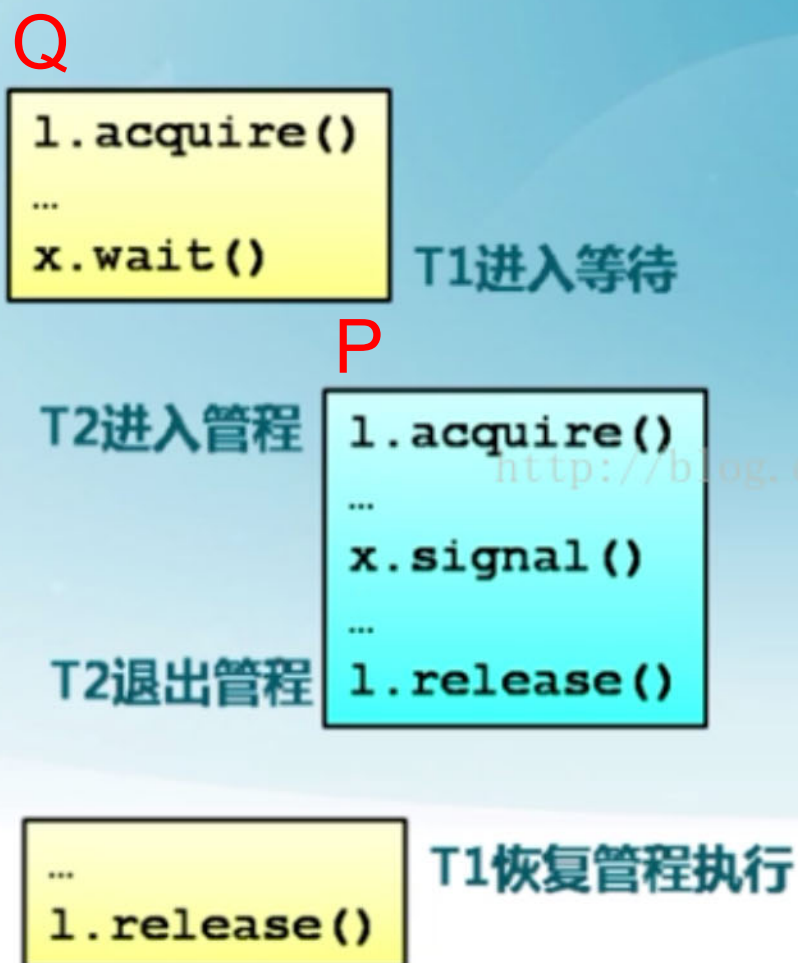
- 例如：进程P调用
signal操作唤醒进程
Q后。



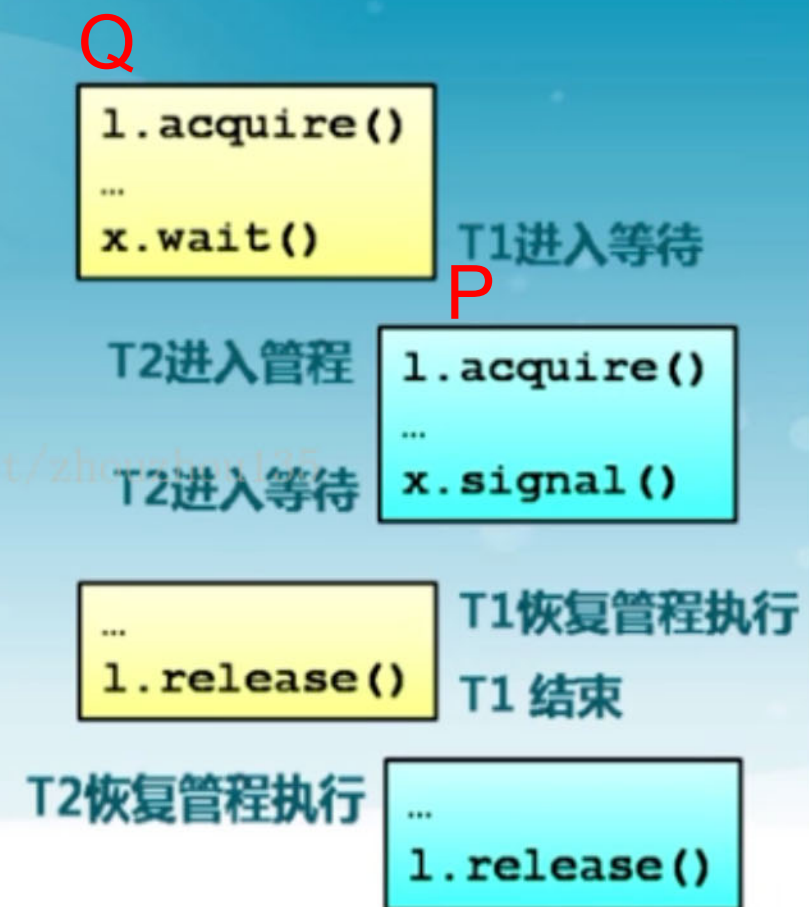
存在的可能处理方式：

- Q等待，直到P离开管程或
等待另一条件（Hansen）。
- P等待，直到Q离开管程或
等待另一条件（Hoare）。

■ Hansen管程










■ Hoare管程



http://blog.csdn.net/Dylan_Frank

内容导航:

-  4.1 进程同步的概念
-  4.2 软件同步机制
-  4.3 硬件同步机制
-  4.4 信号量机制
-  4.5 管程机制
-  **4.6 经典进程的同步问题**
-  4.7 Linux进程同步机制

第4章 进程同步



内容导航:



4.1 进程同步的概念



4.2 软件同步机制



4.3 硬件同步机制



4.4 信号量机制



4.5 管程机制



4.6 经典进程的同步问题 →



4.7 Linux进程同步机制

第4章 进程同步

■ 4.6.1 生产者-消费者问题

■ 4.6.2 哲学家进餐问题

■ 4.6.3 读者-写者问题



生产者-消费者问题



生产者-消费者问题是相互合作进程关系的一种抽象



利用记录型信号量实现：

- 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，可利用互斥信号量mutex使诸进程实现对缓冲池的互斥使用；
- 利用资源信号量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



其它解决方案：AND信号集、管程

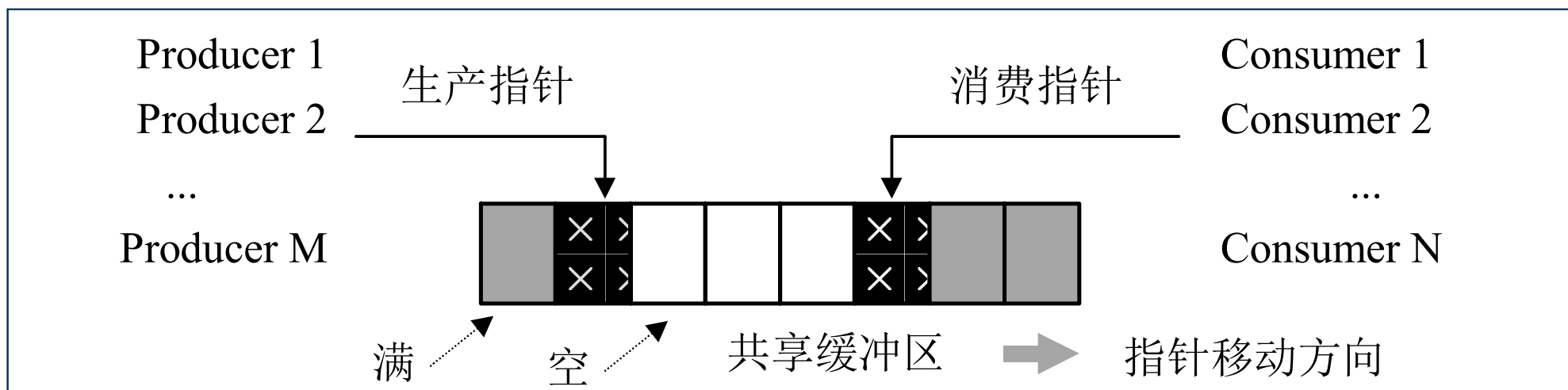


生产者-消费者问题描述

- 生产者 (M个) : 生产产品, 并放入缓冲区
- 消费者 (N个) : 从缓冲区取产品消费
- 问题: 如何实现生产者和消费者之间的同步和互斥?

PRODUCER

CONSUMER





生产者消费者流程

生产者:

```
{  
  ...  
  生产一个产品  
  ...  
  
  ...  
  把产品放入指定缓冲区  
  ...  
}
```

消费者:

```
{  
  ...  
  
  ...  
  从指定缓冲区取出产品  
  ...  
  
  ...  
  消费取出的产品  
  ...  
}
```



生产者消费者的互斥分析

临界资源



生产者

- 把产品放入指定缓冲区
- **in**: 所有的生产者对in指针需要互斥
- **counter**: 所有生产者消费者进程对counter互斥

```
buffer[in] = nextp;  
in = (in + 1) % N;  
counter++;
```



消费者

- 从指定缓冲区取出产品
- **out**: 所有的消费者对out指针需要互斥
- **counter**: 所有生产者消费者进程对counter互斥

```
nextc = buffer[out];  
out = (out + 1) % N;  
counter--;
```



划分临界区

生产者:

```
{  
  ...  
  生产一个产品  
  ...  
  ...  
  ...  
  把产品放入指定缓冲区  
  ...  
}
```

临界区

把产品放入指定缓冲区

消费者:

```
{  
  ...  
  ...  
  ...  
  从指定缓冲区取出产品  
  ...  
  ...  
  消费取出的产品  
  ...  
}
```

从指定缓冲区取出产品

临界区



增加互斥机制

```
semaphore *m;    m->vaule = 1;
```

生产者:

```
{  
    ...  
    生产一个产品  
    ...  
    wait(m);  
    ...  
    把产品放入指定缓冲区  
    ...  
    signal(m);  
}
```

临界区

把产品放入指定缓冲区

消费者:

```
{  
    ...  
    wait(m);  
    ...  
    从指定缓冲区取出产品  
    ...  
    signal(m);  
    ...  
    消费取出的产品  
    ...  
}
```

临界区

从指定缓冲区取出产品



生产者消费者的同步分析



两者需要协同的部分

- **生产者**: 把产品放入指定缓冲区 (关键代码C1)
- **消费者**: 从满缓冲区取出一个产品 (关键代码C2)



三种运行次序 (不同条件下不同运行次序)

- 所有缓冲区空时:
- 所有缓冲区满时:
- 缓冲区有空也有满时:





算法描述：生产者



生产者

...

生产一个产品

...

同步：判断

1) 判断是否能获得一个空缓冲区，如果不能则阻塞

C1:把产品放入指定缓冲区

临界区

2) 满缓冲区数量加1，如果有消费者由于等消费产品而被阻塞，则唤醒该消费者

同步：通知



算法描述：消费者



消费者

同步：判断

1) 判断是否能获得一个满缓冲区，如果不能则阻塞

从满缓冲取出一个产品

临界区

2) 空缓冲区数量加1，如果有生产者由于等空缓冲区而阻塞，则唤醒该生产者

同步：通知



同步信号量定义



共享数据

semaphore ***full**, ***empty**, *m; //full:满缓冲区数量 empty: 空缓冲区数量

初始化:

full->value = 0; **empty->vaule = N;** m->vaule = 1;

生产者:

{

...

生产一个产品

...

wait(empty);

wait(m);

...

C1: 把产品放入指定缓冲区

...

signal(m);

signal(full);

}

当empty大于0时, 表示有空缓冲区, 继续执行; 否则, 表示无空缓冲区, 当前生产者阻塞。

把full值加1, 如果有消费者等在full的队列上, 则唤醒该消费者

消费者:

{

...

wait(full);

wait(m);

...

C2: 从指定缓冲区取出产品

...

signal(m);

signal(empty);

...

消费取

...

}

当full大于0时, 表示有满缓冲区, 继续执行; 否则, 表示无满缓冲区, 当前消费者阻塞。

把empty值加1, 如果有生产者等在empty的队列上, 则唤醒该生产者。



在生产者-消费者问题中应注意

- 在每个进程中用于实现互斥的wait(mutex)和signal(mutex)必须成对出现;
- 对资源信号量empty和full的wait和signal操作, 同样需要成对出现, 但它们分别处于不同的进程中。
- 每个程序中的多个wait操作的顺序不能颠倒, 应先执行对资源信号量的wait操作, 再执行对互斥信号量的wait操作, 否则可能会引起进程死锁。

利用AND信号量解决生产者-消费者问题

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

```
void producer() {  
    do {  
        producer 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,emph);  
        comsumer the item in nextc;  
        ...  
    }while(TRUE); }
```



利用管程解决生产者-消费者问题

```
Monitor producerconsumer {  
    item buffer[N];  
    int in,out;  
    condition notfull,notempty;  
    int count;  
    public:  
    void put(item x) {  
        if (count>=N) cwait(notfull);  
        buffer[in] = x;  
        in = (in+1) % N;  
        count++;  
        csignal(notempty);  
    }  
}
```



```
void get(item x) {  
    if (count<=0) cwait(notempty);  
    x = buffer[out];  
    out = (out+1) % N;  
    count--;  
    csignal(notfull);  
}  
  
{  
    in=0;out=0;count=0;  
}  
}PC;
```



利用管程解决生产者-消费者问题

```
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  
    proceducer(); consumer();  
    coend  
}
```


内容导航:


 4.1 进程同步的概念

 4.2 软件同步机制

 4.3 硬件同步机制

 4.4 信号量机制

 4.5 管程机制

 4.6 经典进程的同步问题 ➡

 4.7 Linux进程同步机制

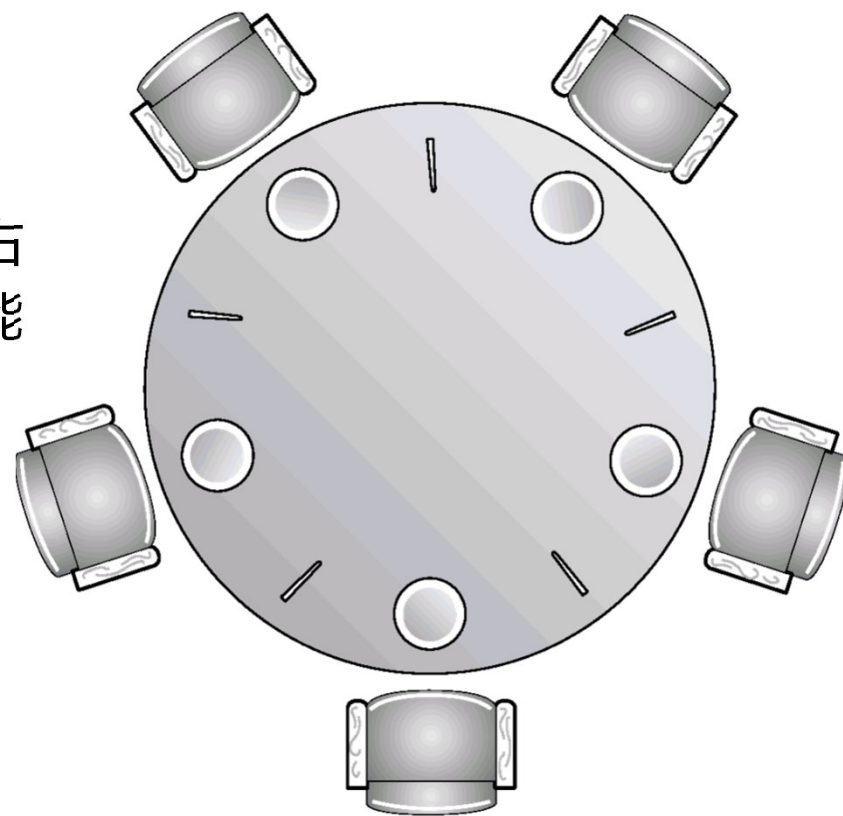
第4章 进程同步

■ 4.6.1 生产者-消费者问题

■ **4.6.2 哲学家进餐问题**

■ 4.6.3 读者-写者问题

- 五个哲学家的生活方式：交替思考和进餐
共用一张圆桌，分别坐在五张椅子上
在圆桌上有五个碗和五支筷子
平时哲学家思考，饥饿时便试图取用其左、右
最靠近他的筷子，只有在拿到两支筷子时才能
进餐
进餐毕，放下筷子又继续思考
- 解决方案：
 - ❑ 记录型信号量；
 - ❑ AND信号量集、管程。





利用记录型信号量解决

Semaphore chopstick[5] = {1,1,1,1,1};

Philosopher i:

do {

wait(chopStick[i]); // get left chopstick

wait(chopStick[(i + 1) % 5]); // get right chopstick

...

// eat for awhile

...

signal(chopStick[i]); //return left chopstick

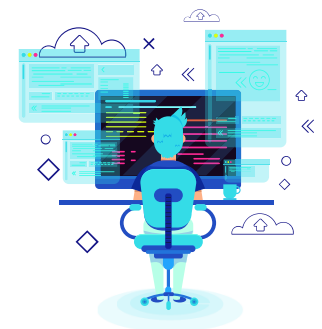
signal(chopStick[(i + 1) % 5]); // return right chopstick

...

// think for awhile

...

} while (true)





存在问题及解决方案



可能引起死锁，如五个哲学家同时饥饿而各自拿起左筷子时，会使信号量 chopstick 均为 0；因此他们试图去拿右筷子时，无法拿到而无限期等待。



解决方法：

- ① 最多允许 4 个哲学家同时坐在桌子周围
- ② 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。
- ③ 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



利用AND信号量机制解决哲学家就餐问题

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





利用管程解决哲学家就餐问题

```
monitor dp{  
    enum { thinking, hungry, eating} state [5];  
    condition self [5];  
    initialization_code() {  
        for (int i = 0; i < 5; i++)  
            state[i] = thinking;  
    }  
    void pickup (int i) {  
        state[i] = hungry;  
        test(i);  
        if (state[i] != eating) self[i].wait ();  
    }  
}
```

```
void putdown (int i) {  
    state[i] = thinking;  
    // test left and right neighbors  
    test((i + 4) % 5);  
    test((i + 1) % 5);  
}
```



利用管程解决哲学家就餐问题

```
void test (int i) {  
    if ( (state[(i + 4) % 5] != eating) &&(state[i] == hungry)  
        &&(state[(i + 1) % 5] != eating) ) {  
        state[i] = eating ;  
        self[i].signal() ;//  
    }  
}  
}
```

哲学家i的活动可描述为:


```
do {  
    dp.pickup (i);  
    ...  
    eat  
    ...  
    dp.putdown (i);  
} while[TRUE];
```


内容导航:


 4.1 进程同步的概念

 4.2 软件同步机制

 4.3 硬件同步机制

 4.4 信号量机制

 4.5 管程机制

 4.6 经典进程的同步问题 →

 4.7 Linux进程同步机制

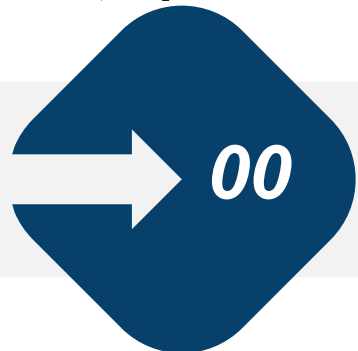
第4章 进程同步

■ 4.6.1 生产者-消费者问题

■ 4.6.2 哲学家进餐问题

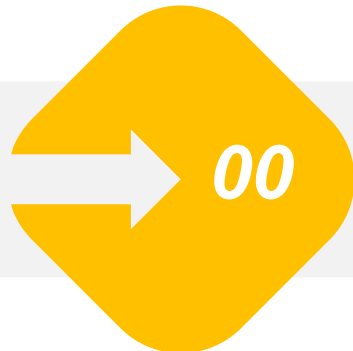
■ **4.6.3 读者-写者问题**

有两组并发
进程：



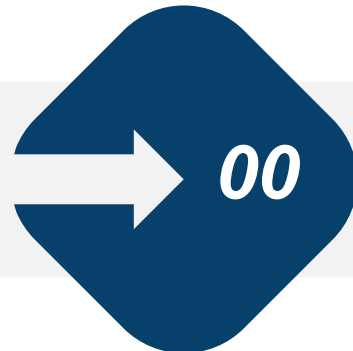
- 读者和写者，共享一组数据区。

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。

分类：



- 读者优先(第一类读者写者问题)
- 写者优先(第二类读者写者问题)

解决方案：



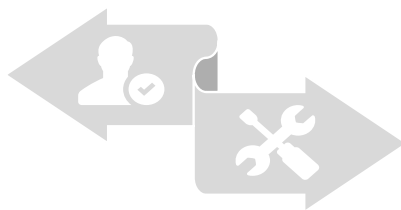
- 记录型信号量
- 信号量集



读者优先(第一类读者写者问题)

如果**读者**来:

- 无读者、写者, 新读者可以读。
- 有写者等, 但有其它读者正在读, 则新读者也可以读。
- 有写者写, 新读者等。



如果**写者**来:

- 无读者, 新写者可以写。
- 有读者, 新写者等待。
- 有其它写者, 新写者等待。

- 为了实现reader与writer进程在读/写时的互斥, 设置了一个互斥信号量wmutex。
- 设置了一个整型变量readcount, 用于表示正在读的进程数目。
- readcount是一个可被多个reader进程访问的临界资源, 因此, 也应该为它设置一个互斥信号量rmutex。

互斥量（互斥锁）和信号量的区别

1. 互斥量(**Mutex**) 用于线程的互斥，信号量(**Semaphore**)用于线程的同步。 这是互斥量和信号量的根本区别，也就是互斥和同步之间的区别。
 - 互斥：是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。
 - 同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源
2. 互斥量值只能为0/1，信号量值可以为非负整数。 也就是说，一个互斥量只能用于一个资源的互斥访问，它不能实现多个资源的多线程互斥问题。信号量可以实现多个同类资源的多线程互斥和同步。当信号量为单值信号量是，也可以完成一个资源的互斥访问。
3. 互斥量（互斥锁）的加锁和解锁必须由同一线程分别对应使用，信号量可以由一个线程释放，另一个线程得到。

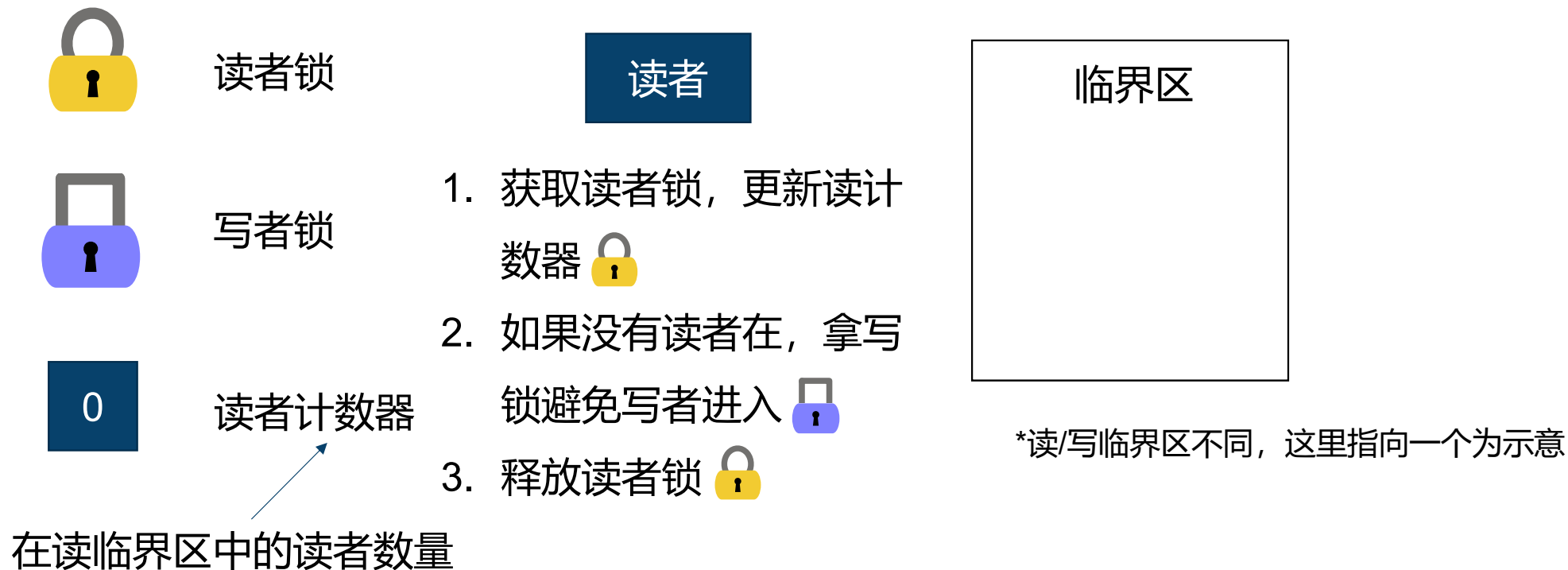


信号量机制实现进程互斥

1. 分析**并发**进程的关键活动，划分临界区（如：对临界资源打印机的访问就应放在临界区）
2. 设置互斥信号量mutex，**初值为1**
3. 在临界区之前执行P(mutex)
4. 在临界区之后执行V(mutex)

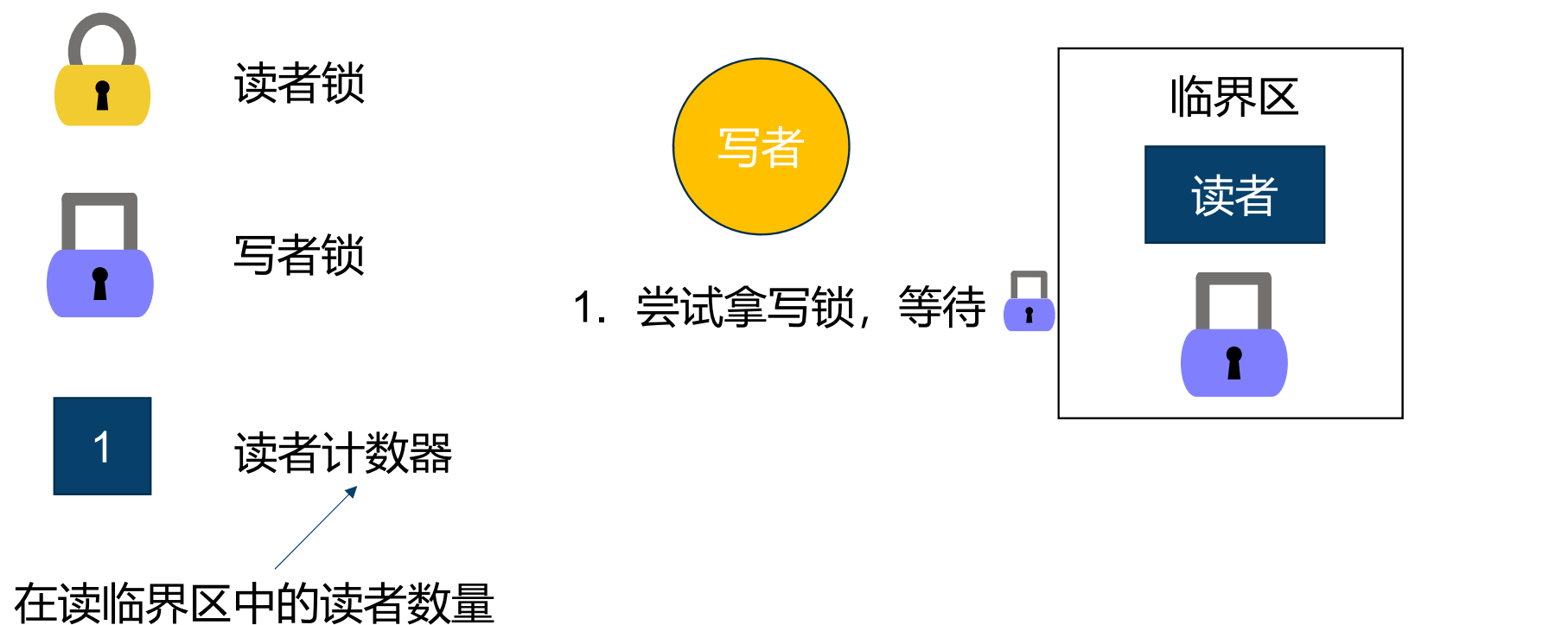
如果多个线程被阻塞在该互斥量上，将 **等待队列中**一个线程并允许它获得改信号量。

读写锁的实现：偏向读者为例



- ❑ 为了实现reader与writer进程在读/写时的互斥，设置了一个互斥信号量wmutex（写锁）。
- ❑ 设置了一个整型变量readcount，用于表示正在读的进程数目。
- ❑ readcount是一个可被多个reader进程访问的临界资源，因此，也应该为它设置一个互斥信号量rmutex（读锁）。

读写锁的实现：偏向读者为例



*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



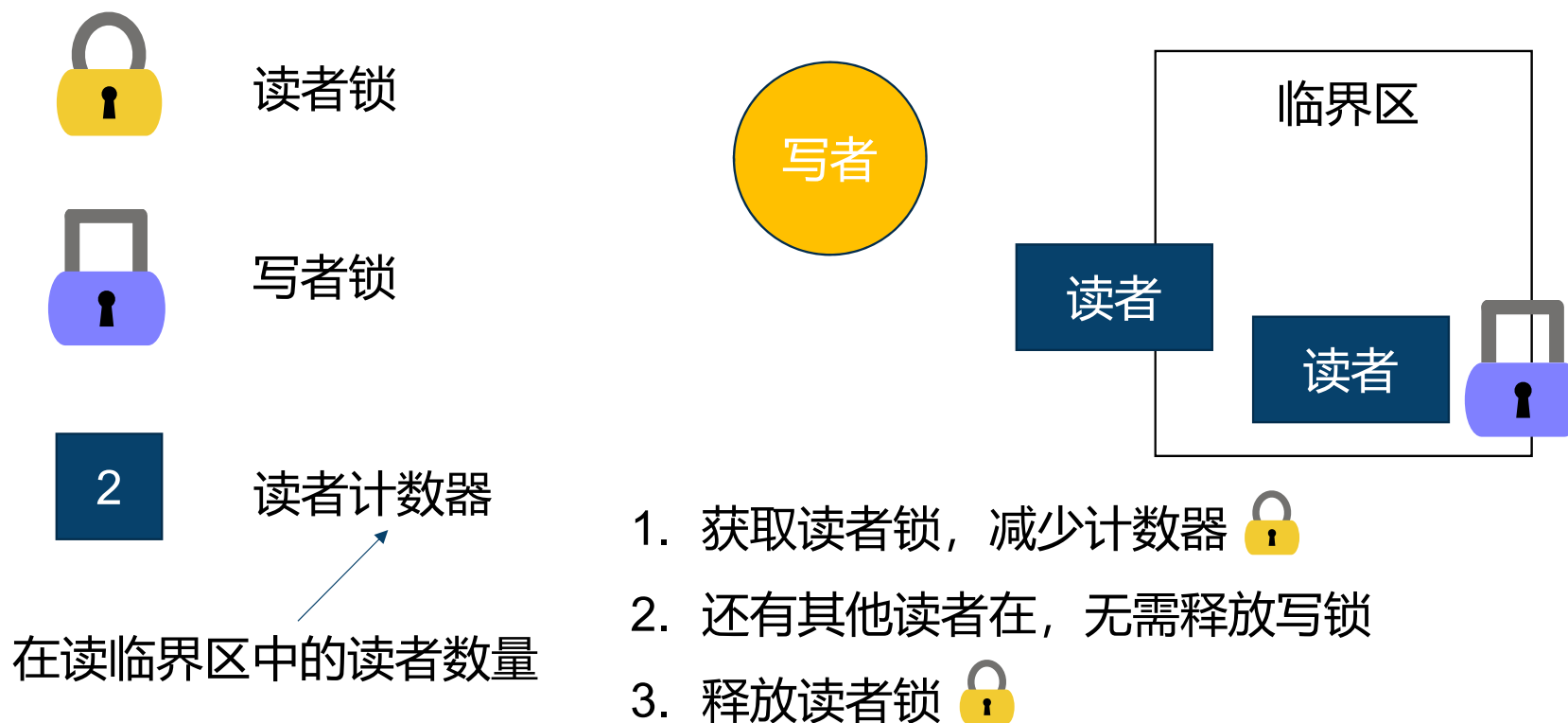
*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



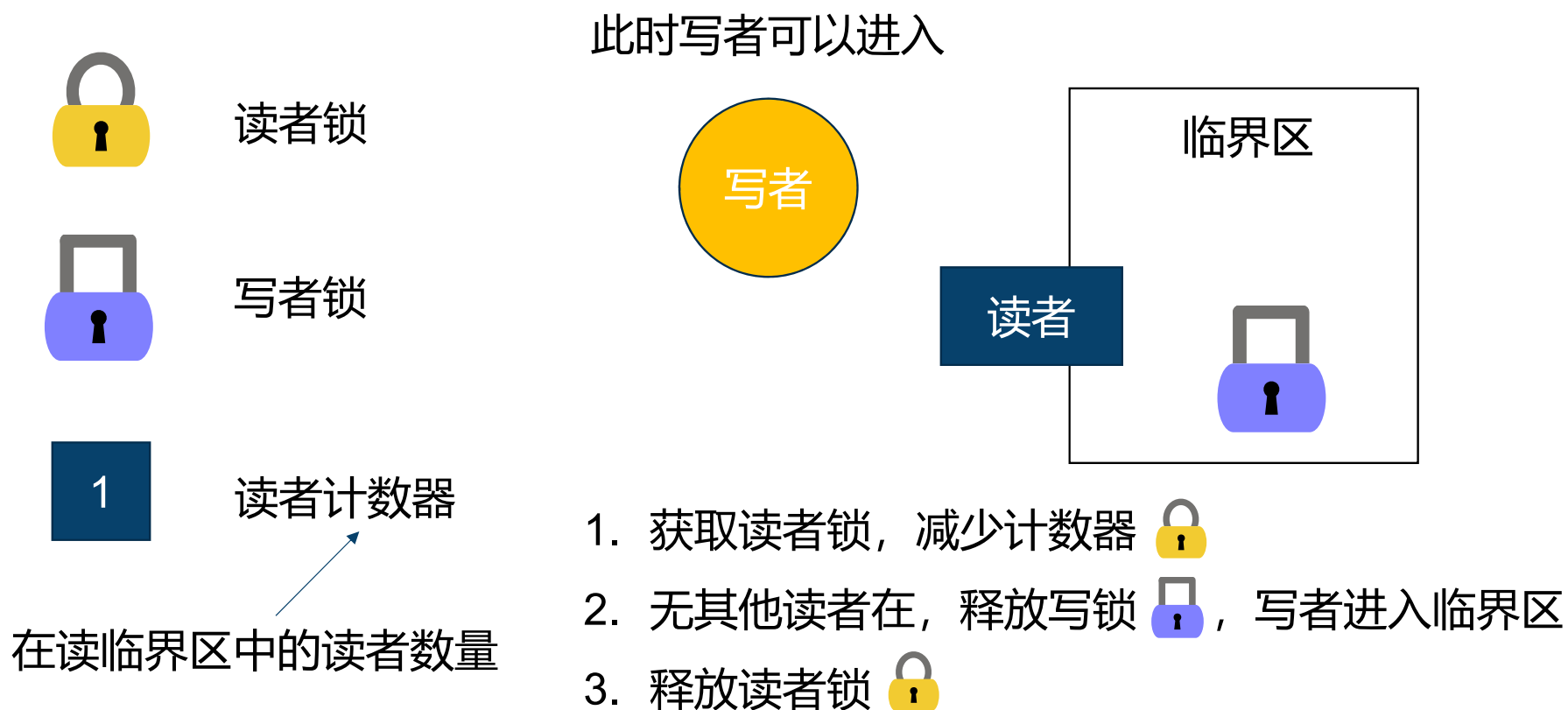
*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



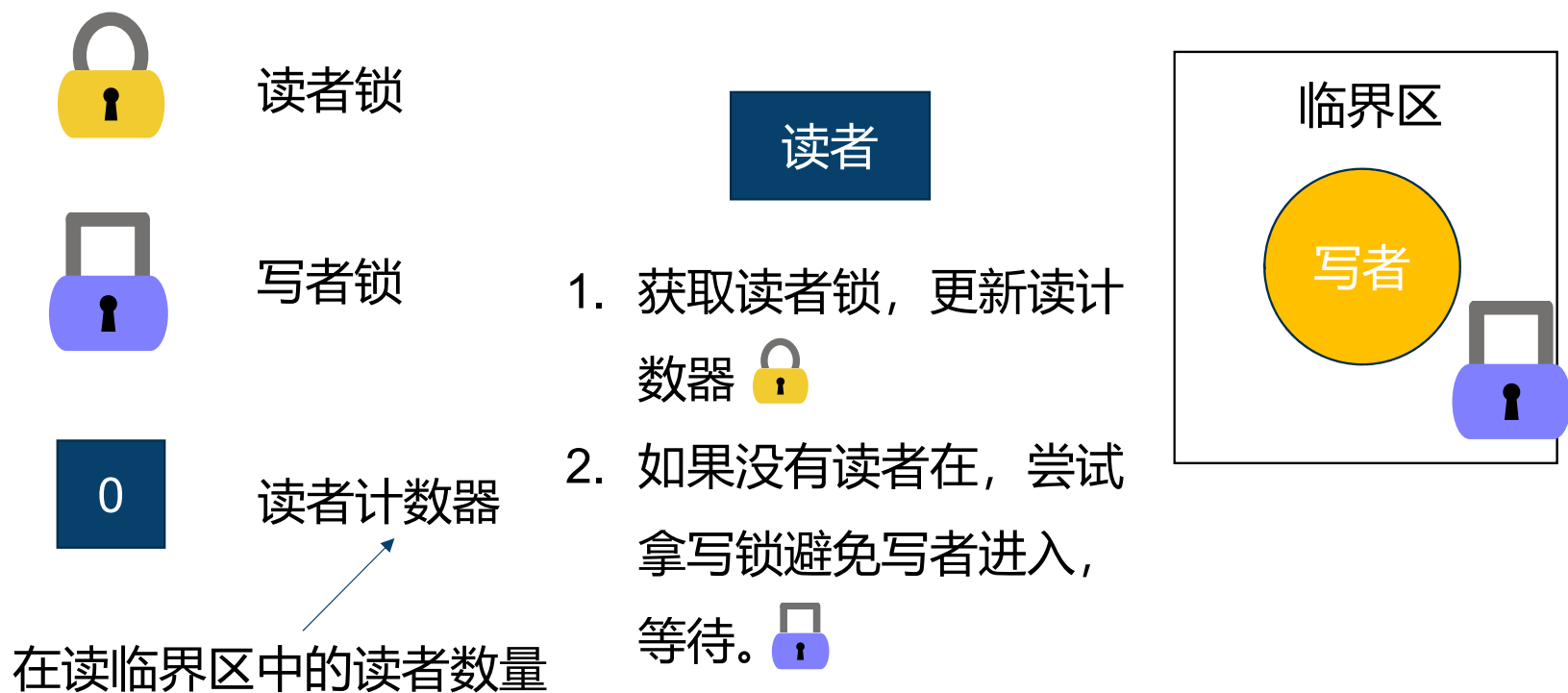
*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



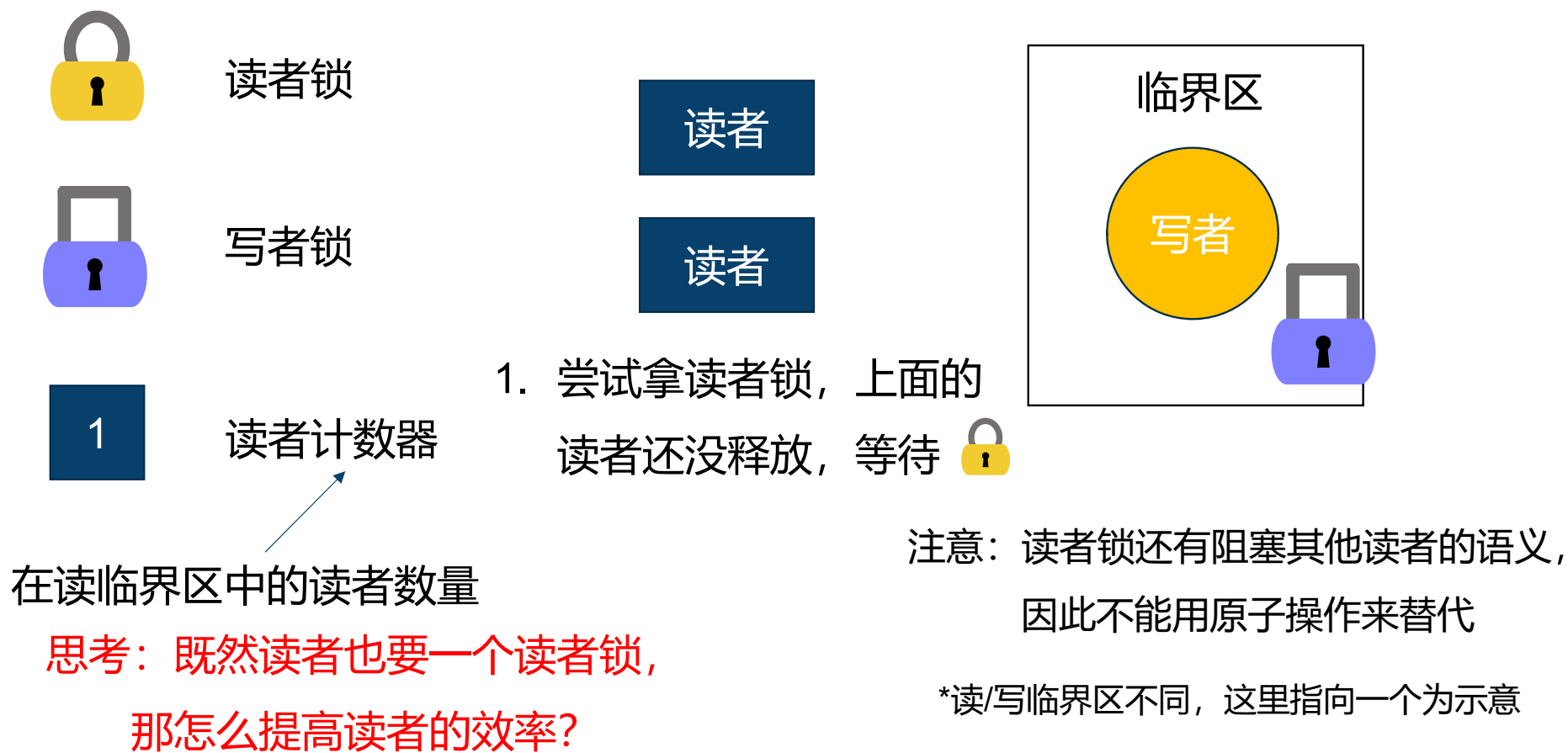
*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例





读者优先(第一类读者写者问题)

```
void reader() {  
    do {  
        wait(rmutex); // 先获得读锁  
        if (readcount == 0) wait(wmutex);  
        readcount++;  
        signal(rmutex);  
        ...  
        读  
        ...  
        wait(rmutex);  
        readcount--;  
        if (readcount == 0) signal(wmutex);  
        signal(rmutex);  
    } while(TRUE);  
}
```

初始化:

```
semaphore rmutex=1, wmutex=1;  
int readcount=0;
```

```
void writer() {  
    do {  
        wait(wmutex);  
        ...  
        写  
        ...  
        signal(wmutex);  
    }  
} while(TRUE)
```



写者优先(第二类读者写者问题)



问题描述

- 多个读者可以同时进行读
- 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
- 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）



如何用PV操作实现？（思考题）



读者优先(第一类读者写者问题)

```
void reader() {  
    do {  
        wait(mutex);  
        if (readcount==0) wait(w);  
        readcount++;  
        signal(mutex);  
        ...  
        读  
        ...  
        wait(mutex);  
        readcount--;  
        if (readcount==0) signal(w);  
        signal(mutex);  
    }while(TRUE);  
}
```

```
初始化:  
semaphore mutex=1, w=1;  
int readcount=0;  
  
void writer() {  
    do {  
        wait(w);  
        ...  
        写  
        ...  
        signal(w);  
    }  
}while(TRUE)
```



写者优先(第二类读者写者问题)



问题描述

- 多个读者可以同时进行读
- 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
- 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）



如何用PV操作实现？（思考题）



利用信号量集解决读者-写者问题

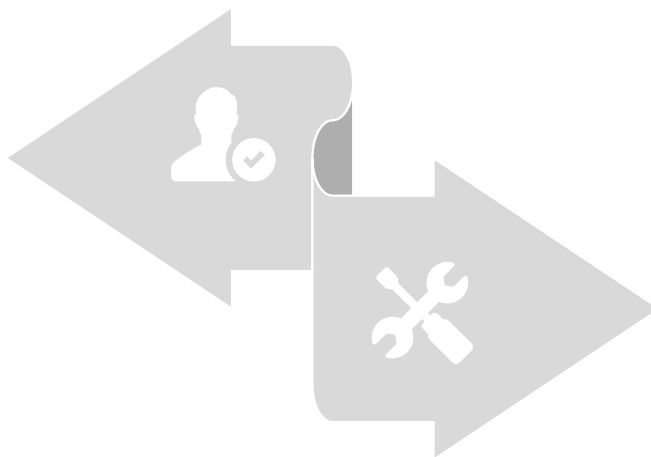
```
void reader() {  
    do {  
        Swait(L,1,1)  
        Swait(mx,1,0);  
        m³...  
        perform read operation;  
        m³...  
        Ssignal(L,1);  
    }while(TRUE);  
}
```

```
void writer() {  
    do {  
        Swait(mx,1,1; L,RN,0);  
        perform write operation;  
        Ssignal(mx,1);  
    }while(TRUE);  
}  
  
int RN;  
semaphore L=RN, mx=1;
```



信号量的使用:

- 信号量必须置一次且只能置一次初值，初值不能为负数
- 除了初始化，只能通过执行P、V操作来访问信号量



使用中存在的问题

- 死锁
- 饥饿



死锁和饥饿



死锁：两个或多个进程无限期地等待一个事件的发生，而该事件正是由其中的一个等待进程引起的。

例如：S和Q是两个初值为1的信号量

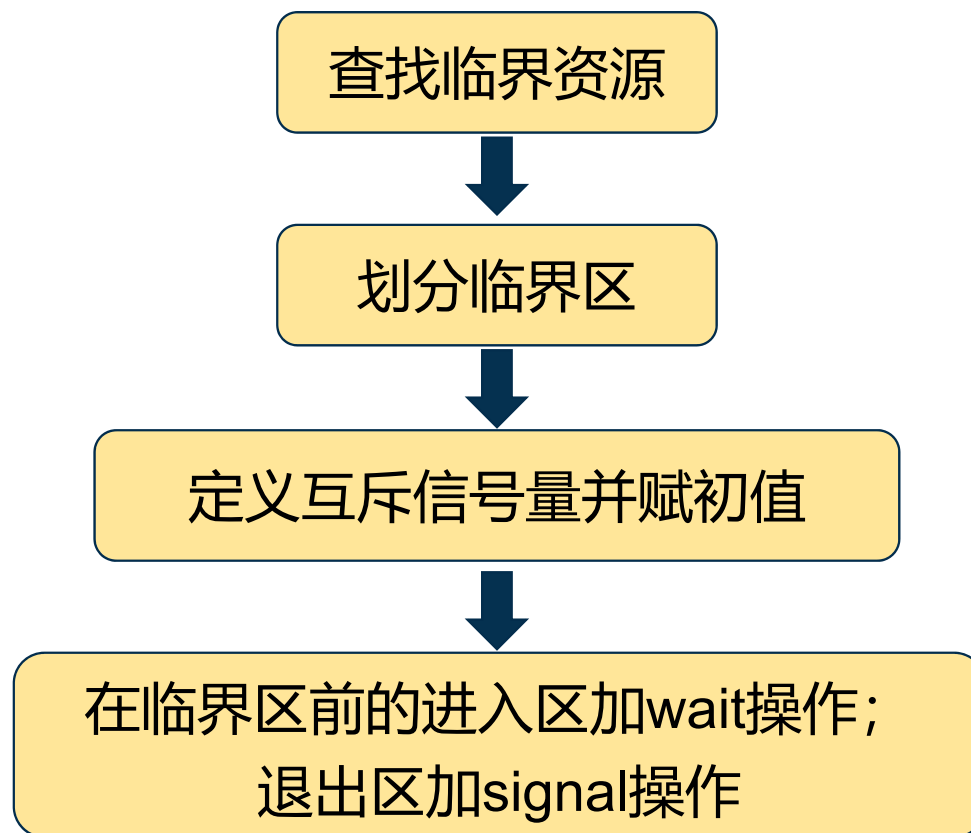
P_0	P_1
P(S);	P(Q);
P(Q);	P(S);
\vdots	\vdots
V(S);	V(Q);
V(Q)	V(S);



饥饿：无限期地阻塞，进程可能永远无法从它等待的信号量队列中移去（只涉及一个进程）。



互斥分析基本方法





①找出需要同步的代码片段（关键代码）

②分析这些代码片段的执行次序

③增加同步信号量并赋初始值

④在关键代码前后加wait和signal操作

同步分析较为困难！



信号量的物理含义

- $S > 0$ 表示有 S 个资源可用;
- $S = 0$ 表示无资源可用;
- $S < 0$ 则 $|S|$ 表示 S 等待队列中的进程个数。
- $P(S)$: 表示申请一个资源;
- $V(S)$ 表示释放一个资源。信号量的初值应该大于等于 0



PV操作的使用

- P.V 操作必须成对出现, 有一个 P 操作就一定有一个 V 操作
- 当为互斥操作时, 它们同处于同一进程
- 当为同步操作时, 则不在同一进程中出现
- 如果 $P(S1)$ 和 $P(S2)$ 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



信号量同步的缺点

同步操作分散：信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁（如P、V操作的次序错误、重复或遗漏）








不利于修改和维护：各模块的独立性差，任一组变量或一段代码的修改都可能影响全局；



易读性差：要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发程序；

正确性难以保证：操作系统或并发程序通常很大，很难保证这样一个复杂的系统没有逻辑错误；

内容导航:

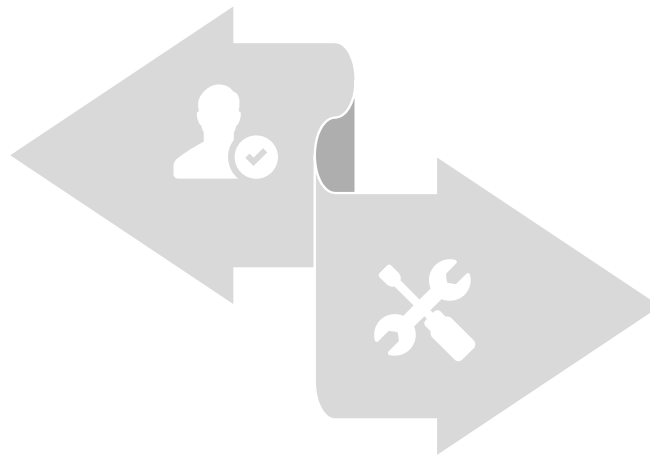
-  4.1 进程同步的概念
-  4.2 软件同步机制
-  4.3 硬件同步机制
-  4.4 信号量机制
-  4.5 管程机制
-  4.6 经典进程的同步问题
-  **4.7 Linux进程同步机制**

第4章 进程同步



Linux并发的主要来源:

- 中断处理、内核态抢占、多处理器的并发。



同步方法:

- 原子操作
- 自旋锁 (spin lock)
 - 不会引起调用者阻塞
- 信号量 (Semaphore)
- 互斥锁 (Mutex)
- 禁止中断 (单处理器不可抢占系统)



习题 1

现有5个操作A、B、C、D和 E，操作C必须在A和B完成后执行，操作E必须在 C和D完成后执行，请使用信号量的P、V（或 wait()、signal()）操作描述上述操作之间的同步关系，并说明所用信号量及其初值。



习题 1 答案

分别为A、B、C、D、E这5个操作进程设置对应的同步信号量为a、b、c、d、e，初始值均为0。

```
A(){  
    完成动作A;  
    V(a);  
}
```

```
B(){  
    完成动作B;  
    V(b);  
}
```

```
C(){ //C必须在A、B都完成后才能完成  
    P(a);  
    P(b);  
    完成动作C;  
    V(c); }
```

```
D(){  
    完成动作D;  
    V(d);  
}
```

```
E(){ //E必须在完成C、D之后执行P(c);  
    P(c);  
    P(d);  
    完成动作E;  
    V(e); }
```




习题 2

系统中有多个生产者进程和消费者进程，共享用一个可以存1000个产品的缓冲区（初始为空），当缓冲区未满时，生产者进程可以放入1件其生产的产品，否则等待；当缓冲区不空时，消费者进程可以取走1件产品，否则等待。要求1个消费者进程从缓冲区连续取出10件产品后，其他消费者进程才可以取产品，请用信号量P，V（或wait()、signal()）操作实现进程间的互斥和同步，要求写出完整的过程；并指出所用信号量的含义和初值。



习题 2 答案

生产者之间设互斥信号量mutex1，消费者之间设互斥信号量mutex2。上述进程的同步问题，要设置3个信号量。其中empty对应空闲的缓冲单元，初值为1000；full对应缓冲区中待取走的产品数，初值为0；另外，还须定义2个整型变量in、out，初值均为0，分别指示下一个可存放产品的缓冲单元、下一个取走的缓冲单元。过程如下：

Producer() //生产者进程

{

 Produce an item put in nextp; //生产一个产品，存在临时缓冲区

 P(empty); //申请一个空缓冲区

 P(mutex1); //生产者申请使用缓冲区

 array[in]=nextp; //将产品存入缓冲区

 in=(in+1)%1000; //指针后移

 V(mutex1); //生产者缓冲区使用完毕，释放互斥信号量

 V(full); //增加一个满缓冲区

}

Consumer() //消费者进程

{

 P(mutex2); //消费者申请使用缓冲区

 for(int i = 0;i<10;i++) //一个消费者进程须从缓冲区连续取走 10 件产品

 {

 P(full); //申请一个满缓冲区

 nextc[i]=array[out]; //将产品取出，存于临时缓冲区

 out=(out+1)%1000; //指针后移

 V(empty); //增加一个空缓冲区

 }

 V(mutex2); //消费者缓冲区使用完毕，释放互斥信号量

 Consume the items in nextc; //消费掉这 10 个产品

}

某银行提供了1个服务窗口和10个供顾客等待时使用的座位。顾客到达银行时，若有空座位，则到取号机上领取一个号，等待叫号。取号机每次仅允许一位顾客使用。当营业员空闲时，通过叫号选取一位顾客，并为其服务。顾客和营业员的活动过程描述如下。

```
cobegin {  
    process顾客{  
        从取号机上获得一个号码;  
        等待叫号;  
        获得服务;  
    }  
    process营业员{  
        while (TRUE) {  
            叫号;  
            为顾客服务;  
        }  
    }  
} coend
```

请添加必要的信号量和P、V操作或wait)、signal()操作，实现上述过程中的互斥与同步。要求写出完整的过程，说明信号量的含义并赋初值。

semaphore numget=1,seats=10,custom=0;//numget是关于取号机互斥的信号量; 信号量 seats是座位的个数;信号量custom是顾客的个数

```
process顾客{
    P(seats); //看有没有空座位
    P(numget); //取号
    取号;
    V(numget); //取完号后释放取号机11
    V(custom);
    等待叫号;
    V(seats);
    接受服务;
}
```

```
process营业员{
    P(custom);
    叫号;
    为顾客服务;
}
```



学而时习之（第4章总结）

第1章 操作系统引论

第2章 进程的描述与控制

第3章 处理机调度与死锁

第4章 进程同步

第5章 存储器管理

第6章 虚拟存储器

第7章 输入/输出系统

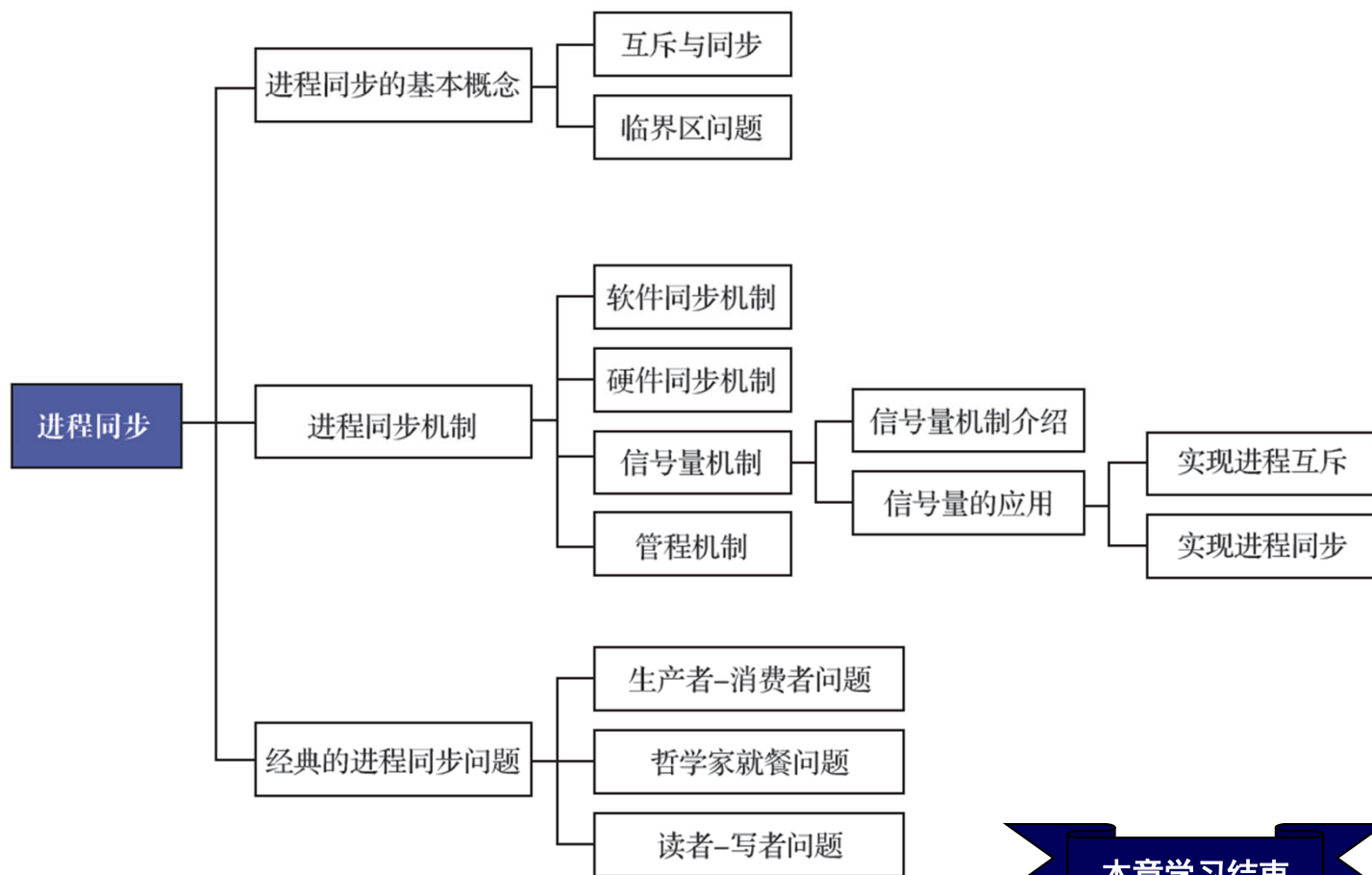
第8章 文件管理

第9章 磁盘存储器管理

第10章 多处理机操作系统

第11章 虚拟化和云计算

第12章 保护和安全



本章学习结束



第一次作业

简答题

1	2	3	4	5	6	7	8
9	10	11	12				

计算题

13	14	15	
----	----	----	--

综合应用题

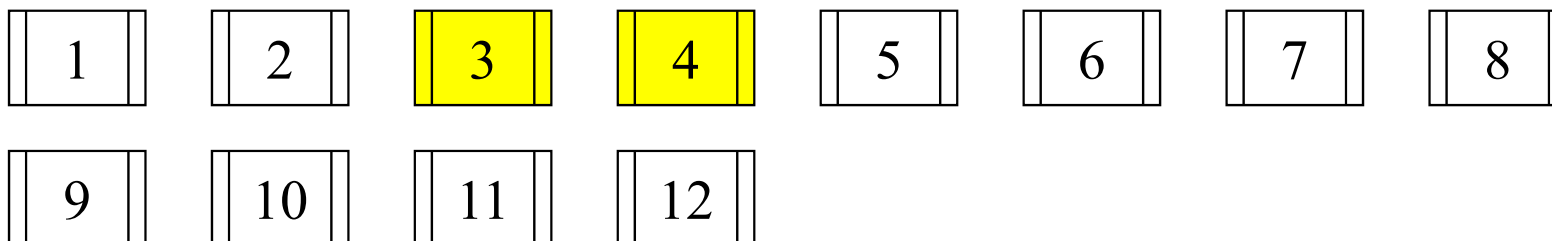
16	17	18	19	20	21
----	----	----	----	----	----

标黄色为本次作业



第二次作业

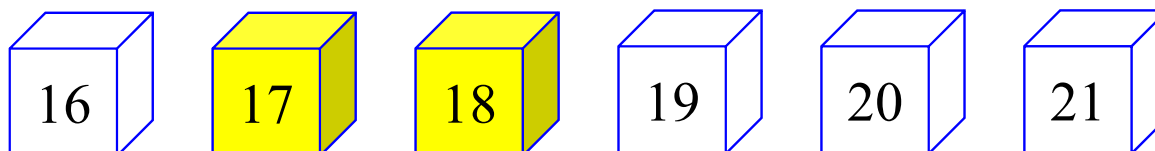
简答题



计算题



综合应用题



标黄色为本次作业



Windows XP系统宣布“退役”，国产操作系统迎来新机遇

2001年10月25日，由微软公司研发的Windows XP系统正式发布。

Windows XP系统是一款面向个人计算机和平板电脑的操纵系统，具有界面华丽、兼容性好、功能丰富、管理方便等特点，集成了微软公司的防火墙技术，支持在第一时间为用户所要升级的硬件提供相关程序下载等功能。相比于Windows系列之前的操作系统，Windows XP系统的用户体验得到了极大程度的提升。这也就引出了该操作系统名称中“XP”的由来，其即“experience”（体验）一词的缩写。

2014年4月8日，这款功能多、体验好，在各个领域经过13年广泛应用的操作系统正式宣布“退役”，即微软公司终止对Windows XP系统继续提供技术支持。



Windows XP系统宣布“退役”，国产操作系统迎来新机遇

该事件实际上对我国工薪阶层以及行动迟缓的单位机关造成了一定的影响，但是同时也给国产操作系统的发展带来了新机遇。虽说当时我国还没有真正意义上自己的操作系统（没有自己的操作系统，就得继续受制于人），但是，计算机技术对国人来说具有独到优势（在计算机相关技术国际知名企业中有大量国人就业），而且我国的整体经济状况呈良好发展态势，因此，无论从哪个层面讲，我国都有必要投入人力、物力来研发自己的操作系统。

我们坚信，在众多知识分子与科技人才的共同努力下，我国必定能够早日研发成功自主可控的国产操作系统，从此不再受制于人！



经典教材《计算机操作系统》**最新版**



学习进步！

作者：汤小丹、王红玲、姜华、汤子瀛