



操作系统 Operating System

第四章 进程与并发程序设计(2) ——同步与互斥

沃天宇

woty@buaa.edu.cn

2021年4月13日





REVIEW & EXERCISES

- 在单用户系统中，有 n 个进程，排在就绪队列和等待队列中的进程个数的取值范围是： $0 \sim (n-1)$, $0 \sim n$
- 判断：
 - Y – 若系统中没有Running进程，则一定没有Ready进程
 - N – 若系统中既没有Running进程，也没有Ready进程，则系统中没有进程
 - N – 不同的进程必然对应不同的程序
 - N – 用户进程可以自行修改PCB
 - Y – 进程在运行中可以将自身状态变为阻塞状态
 - N – 子进程可以继承父进程拥有的全部资源
 - N – 中断是进程切换的充分必要条件
- 区分：父进程、子进程，主程序、子程序
- 一个进程进入阻塞状态，其执行断点保存在什么地方？如何恢复执行？

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

程序的并发执行

并发是OS的设计基础，也是所有（如，同步互斥）问题产生的原因。

- 进程的三个特征：

- **并发**：体现在进程的执行是间断性的；进程的相对执行速度是不可测的。（间断性）
- **共享**：体现在进程/线程之间的制约性（如共享打印机）（非封闭性）。
- **不确定性**：进程执行的结果与其执行的相对速度有关，是不确定的（不可再现性）。

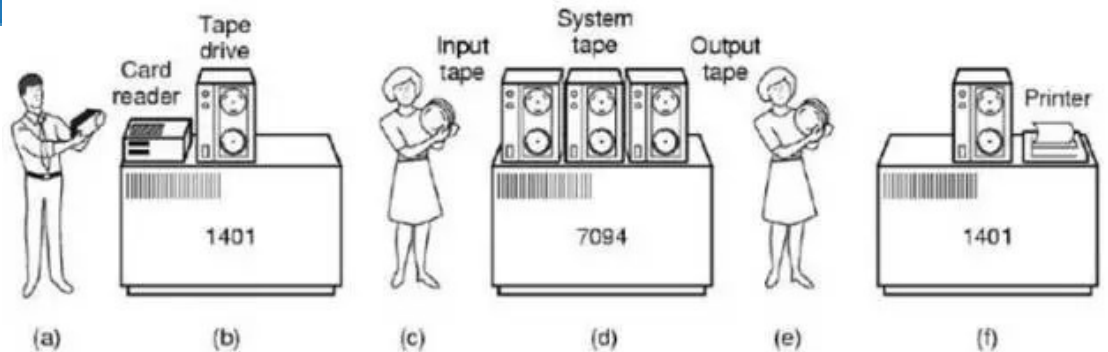
程序的并发执行

并发执行，不可避免地产生了多个进程对同一个共享资源访问，造成了资源的争夺。

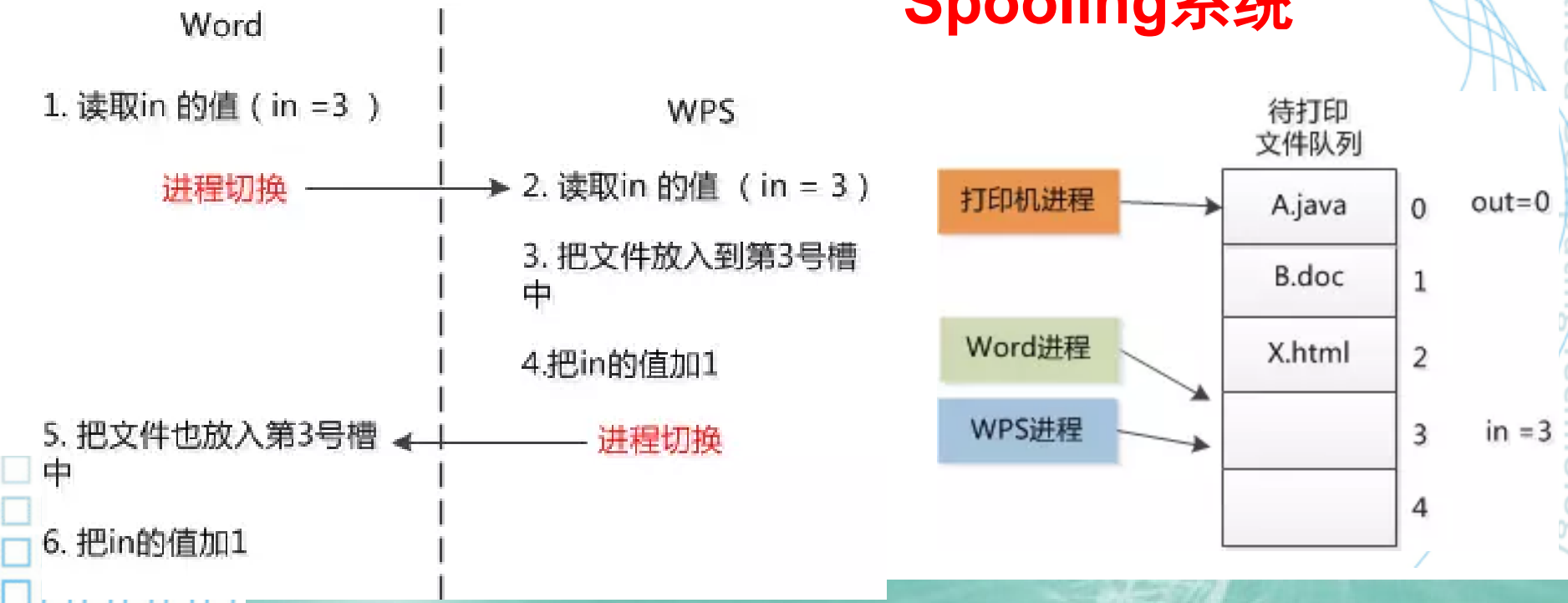
- **竞争**：两个或多个进程对同一共享数据同时进行访问，而最后的结果是不可预测的，它取决于各个进程对共享数据访问的相对次序。这种**情形**叫做竞争。
- **竞争条件**：多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关。
- **临界资源**：我们将一次仅允许一个进程访问的资源称为临界资源。
- **临界区**：每个进程中访问临界资源的那段代码称为临界区。

临界资源

- 一次只允许一个进程使用的资源，如打印机。



Spooling系统





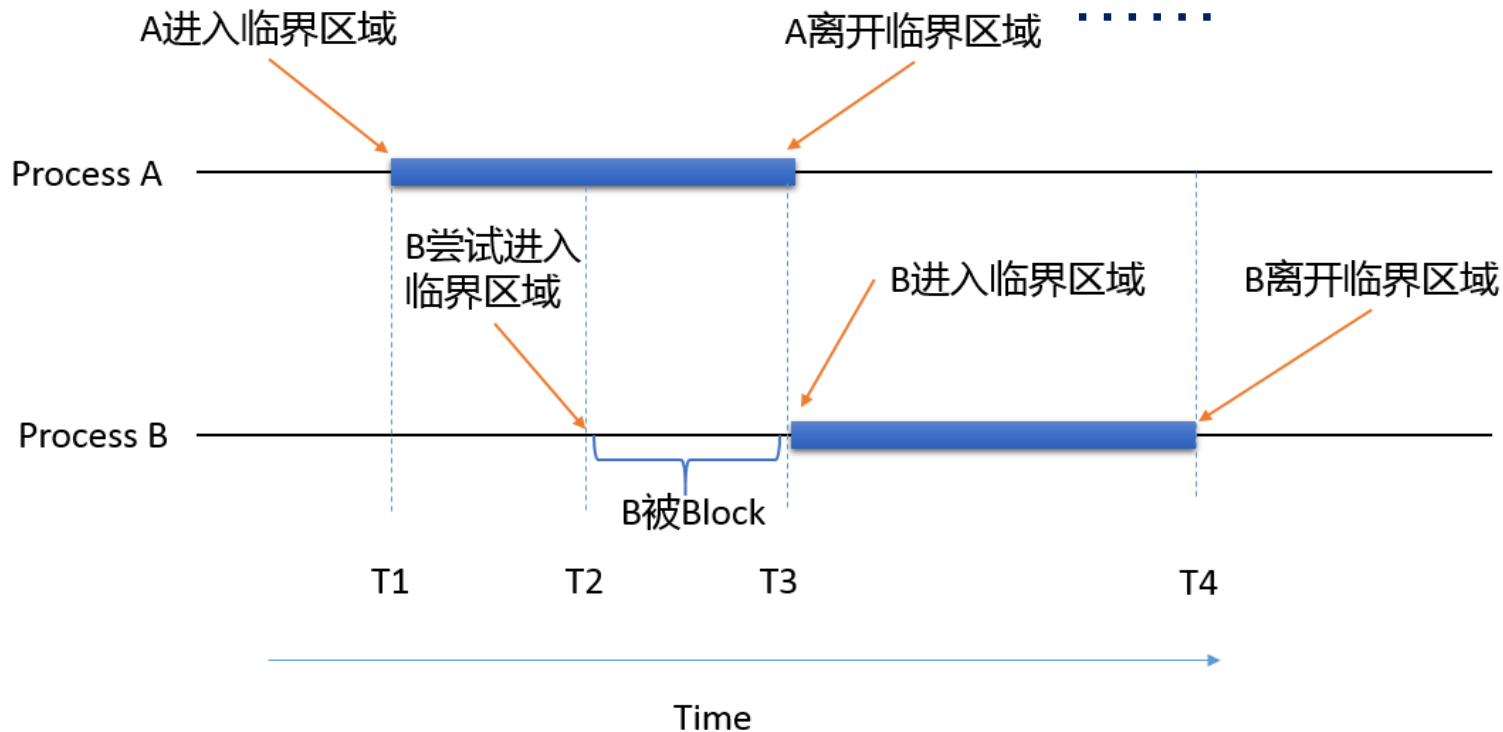
临界区

- 每个进程中访问临界资源的那段代码称为临界区。

P:

.....

Entry section
critical section
exit section
remainder section



进程的同步与互斥

• 进程互斥（间接制约关系）：

- 两个或两个以上的进程，不能同时进入关于同一组共享变量的临界区域，否则可能发生与时间有关的错误，这种现象被称作进程互斥。
- 进程互斥是进程间发生的一种间接性作用，一般是程序不希望的。

• 进程同步（直接制约关系）：

- 系统中各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的过程称为进程同步。
- 进程同步是进程间的一种刻意安排的直接制约关系。即为完成同一个任务的各进程之间，因需要协调它们的工作而相互等待、相互交换信息所产生的制约关系。

同步与互斥的区别与联系

- **互斥**：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。互斥无法限制访问者对资源的访问顺序，即访问是**无序访问**。
- **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，特别是所有对资源的写入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。



互斥区管理应满足的条件

1. 没有进程在临界区时，想进入临界区的进程可进入。
2. 任何两个进程都不能同时进入临界区（Mutual Exclusion）；
3. 当一个进程运行在它的临界区外面时，不能妨碍其他的进程进入临界区（Progress）；
4. 任何一个进程进入临界区的要求应该在有限时间内得到满足（Bounded Waiting）。

正确、公平、效率

机制设计上应遵循的准则

- **空闲让进**：临界资源处于空闲状态，允许进程进入临界区。如，临界区内仅有一个进程执行
- **忙则等待**：临界区有正在执行的进程，所有其他进程则不可以进入临界区
- **有限等待**：对要求访问临界区的进程，应在保证在有限时间内进入自己的临界区，避免死等。
- **让权等待**：当进程（长时间）不能进入自己的临界区时，应立即释放处理机，尽量避免忙等。



内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
 - 软件方法
 - 硬件方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题



软件方法尝试1

P:

.....

while(Occupied); ①

Occupied=true;

临界区

Occupied=false;

.....

Q:

.....

while(Occupied); ②

Occupied=true;

临界区

Occupied=false;

.....

Occupied: 临界区空满标志
true: 临界区内有进程
false: 临界区内无进程
Occupied的初值为false

问题：先检查有无标志，后留标志，造成一个空挡，不能实现互斥。



软件方法尝试2

P:

.....

```
while(turn==Q);  
turn=P;
```

临界区

```
turn=Q;
```

.....

Q:

.....

```
while(turn==P);  
turn=Q;
```

临界区

```
turn =P;
```

.....

turn: 进程进入临界区的优先标志。
turn的初值为P或Q
(对两个进程而言就是0或1)

问题: 固然实现了互斥, 但要求两进程严格交替进入临界区。否则, 一个临界区外的进程会阻止另一个进程进入临界区, 违反了progress原则。



软件方法尝试3

After you
问题

P:

.....

pturn=true;
while(qturn);

临界区

pturn=false;

.....

Q:

.....

qturn=true;
while(pturn);

临界区

qturn=false;

.....

pturn, qturn: 进程需要进入临界区的标志, 初值为 false

P进入临界区的条件: $pturn \wedge \text{not } qturn$ 。

Q进入临界区的条件: $qturn \wedge \text{not } pturn$

问题: 竞争时都可能无法进入临界区, 违反了progress原则。



软件方法尝试4

P:

.....

pturn=true;

while(qturn)

pturn=false;

pturn=true;

临界区

pturn=false;

.....

Q:

.....

qturn=true;

while(pturn)

qturn=false;

qturn=true;

临界区

qturn=false;

.....

问题：增加了让权等待，导致互斥规则被破坏。



Dekker算法

P:

.....

pturn=true;

while(qturn) {

if(turn == 1) {

pturn=false;

while(turn==1);

pturn=true;

} }

临界区

turn = 1;

pturn=false;

.....

Q:

.....

qturn=true;

while(pturn) {

if(turn == 0) {

qturn=false;

while(turn==0);

qturn=true;

} }

临界区

turn = 0;

qturn=false;

.....

引入变量 *turn*, 以便在竞争时逃出进入临界区的进程

让权

缺点：忙等
浪费CPU时间

1965年第一个用软件方法解决了临界区问题



Peterson算法

```
#define FALSE 0
#define TRUE 1
#define N      2           // 进程的个数
int turn;                 // 轮到谁?
int interested[N];
// 兴趣数组, 初始值均为FALSE
```

```
void enter_region ( int process)
    // process = 0 或 1
{
    int other;
    // 另外一个进程的进程号
    other = 1 - process;
    interested[process] = TRUE;
    // 表明本进程感兴趣
    turn = process;
    // 设置标志位
    while( turn == process &&
interested[other] == TRUE);
}
```

循环

```
void leave_region ( int process)
{
    interested[process] = FALSE;
    // 本进程已离开临界区
}
```

进程i:

```
... ..
enter_region ( i );
    临界区
leave_region ( i );
... ..
```

Peterson算法解决了互斥访问的问题, 而且克服了强制轮流法的缺点, 可以完全正常地工作 (1981)



N个进程互斥的软件算法

- 实现进程互斥的软件的结构框架是：

Repeat

entry section

critical section

exit section

remainder section

Until false

- 进程互斥的软件实现算法有：Lamport面包店算法和Eisenberg算法。这两种算法均假定系统中进程的个数有限，如n个。

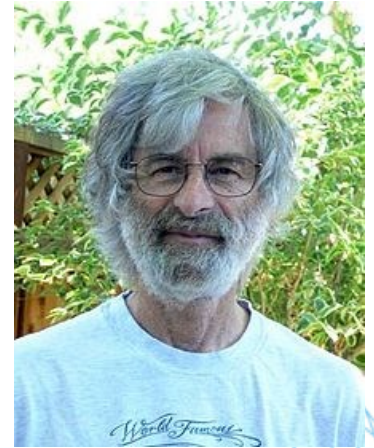
Lamport Bakery Algorithm

面包店算法（ Bakery Algorithm ）的基本思想来源于顾客在面包店购买面包时的排队原理。顾客进入面包店前，首先抓取一个号码，然后按号码从小到大的次序依次进入面包店购买面包，这里假定：

- (1) — 面包店按由小到大的次序发放号码，且两个或两个以上的顾客有可能得到相同号码（要使顾客的号码不同，需互斥机制）；
- (2) — 若多个顾客抓到相同号码，则按顾客名字的字典次序排序 *i.e.*, 1,2,3,3,3,3,4,5... （假定顾客没有重名）。

Lamport Bakery Algorithm

- 计算机系统中，顾客相当于进程，每个进程有一个唯一的标识，用 P_i 表示，对于 P_i 和 P_j ，若有 $i < j$ ，即 P_i 先进入临界区，则先为 P_i 服务。
- 基本思想：设置一个发号器，按由小到大的次序发放号码。进程进入临界区前先抓取一个号码，然后按号码从小到大的次序依次进入临界区。若多个进程抓到相同的号码则按进程编号依次进入。
- 面包店算法是1974年由莱斯利·兰波特给出的。著名的排版软件LaTeX也是他的贡献。



Leslie Lamport
2013 Turing Award



数据结构

`int choosing[n];` //表示进程是否正在抓号，初值为0。
若进程i正在抓号，则`choosing[i]=1`。

`int number[n];` //记录进程抓到的号码，初值为0。
若`number[i]=0`，则进程i没有抓号

排序方法：字典序

$(a, b) < (c, d) \rightarrow (a < c) \text{ or } ((a == c) \text{ and } (b < d))$

初始值：

Choosing , Number : array [1..N] of integer = {0};



Bakery Algorithm (for P_i)

```
Entry Section (i) { // i → process i
    while (true) {
        Choosing[i] = 1;
        Number[i] = 1 + max(Number[1], ..., Number[N]);
        Choosing[i] = 0;
        for (j=1; j ≤ N; ++j) {
            while (Choosing[j] != 0) { }
            // wait until process j receives its number
            while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) { }
            // wait until processes with smaller numbers, or with the
            // same number, but with higher priority, finish their work
        }
        // critical section...
        Number[i] = 0;
        // non-critical section...
    }
}
```

Bakery算法的说明

- 当进程 P_i 计算完 $\max(\dots)+1$ 但尚未将值赋给 $\text{number}[i]$ 时，进程 P_j 中途插入，计算 $\max(\dots)+1$ ，得到相同的值。在这种情况下， $\text{Choosing}[j]$ 的使用可保证编号较小的进程先进入临界区，并不会出现两个进程同时进入临界区的情况。
- 忙式等待：上述Lamport面包店算法中，若 while 循环的循环条件成立，则进程将重复测试，直到条件为假。实际上，当 while 循环条件成立时，进程 P_i 不能向前推进，而在原地踏步，这种原地踏步被称为忙式等待。忙式等待空耗CPU资源，其它进程无法使用，因而是低效的。

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
 - 软件方法
 - 硬件方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

硬件方案1：中断屏蔽

中断屏蔽方法：使用“开关中断”指令。

- 执行“关中断”指令，进入临界区操作；
- 退出临界区之前，执行“开中断”指令。

优缺点：

- 简单。
- 不适用于多**CPU**系统：往往会带来很大的性能损失；
- 单处理器使用：很多日常任务，都是靠中断的机制来触发的，比如时钟，如果使用屏蔽中断，会影响时钟和系统效率，而且用户进程的使用可能很危险！
- 使用范围：内核进程（少量使用）。

硬件方案2：使用test and set指令

- TS (test-and-set) 是一种不可中断的基本原语（指令）。它会写值到某个内存位置并传回其旧值。在多进程可同时存取内存的情况下，如果一个进程正在执行检查并设置，在它执行完成前，其它的进程不可以执行检查并设置。
- Test and Set指令
 - IBM370系列机器中称为TS;
 - 在INTEL8086中称为TSL。
- 语义：

```
TestAndSet(boolean_ref lock) {  
    boolean initial = lock;  
    lock = true;  
    return initial; }
```



自旋锁Spinlocks

- 利用test_and_set硬件原语提供互斥支持
- 通过对总线的锁定实现对某个内存位置的原子读与更新

```
acquire(lock) {  
    while(test_and_set(lock) == 1);  
}
```

critical section

```
release(lock) {  
    lock = 0;  
}
```



Spinlocks

0

```
acquire(int *lock) {  
    while(test_and_set(lock) == 1)  
        /* do nothing */;  
}  
release(int *lock) { *lock = 0; }
```

Let me in!!!

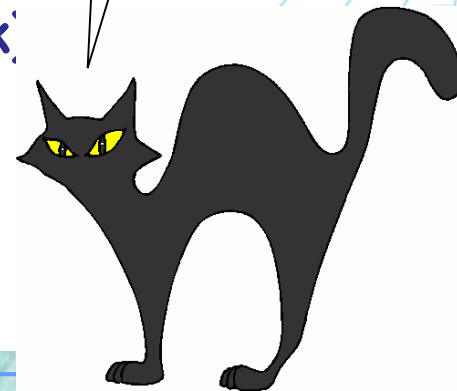
```
acquire(houselock);  
Jump_on_the_couch();  
Be_goofy();  
release(houselock);
```

1

```
acquire(houselock);  
Nap_on_couch();  
Release(houselock);
```

1

No, Let me in!!!





Spinlocks

1

```
acquire(int *lock) {  
    while(test_and_set(lock) == 1)  
        /* do nothing */;  
}  
release(int *lock) { *lock = 0; }
```

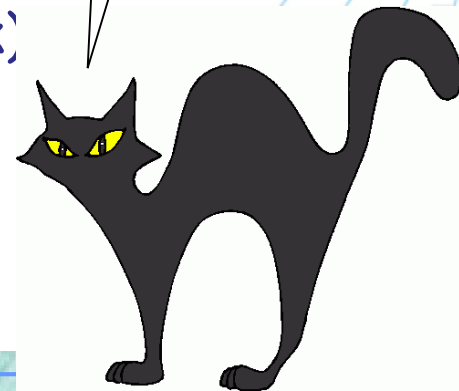
Yay, couch!!!

```
acquire(houselock);  
Jump_on_the_couch();  
Be_goofy();  
release(houselock);
```

```
acquire(houselock);  
Nap_on_couch();  
Release(houselock);
```

I still want in!

1



Spinlocks

1

```
acquire(int *lock) {  
    while(test_and_set(lock) == 1)  
        /* do nothing */;  
}  
release(int *lock) { *lock = 0; }
```

Oooh, food!

```
acquire(houselock);  
Jump_on_the_couch();  
Be_goofy();  
release(houselock);
```

```
acquire(houselock);  
Nap_on_couch();  
Release(houselock);
```

It's cold here!

1

X86基于TSL的自旋锁汇编代码

enter_region:

TSL REGISTER, LOCK

| copy lock to register and set lock to 1

CMP REGISTER, #0

| was lock zero?

JNE enter_region

| if it was non zero, lock was set, so loop

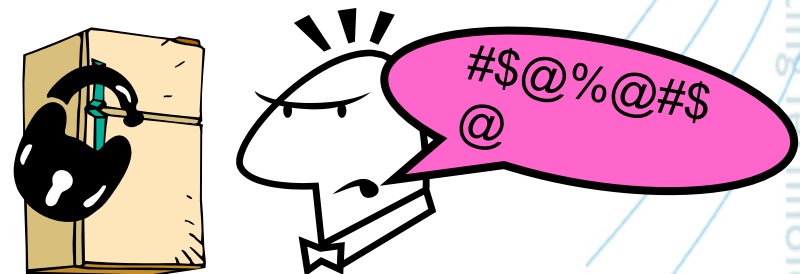
RET | return to caller; critical region entered

leave_region:

MOVE LOCK, #0

| store a 0 in lock

RET | return to caller



硬件方案3：使用swap指令

- Swap（对换）指令与TSL指令一样，是不会被中断的原子指令，其功能是交换两个字的内容，其语义如下：

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

- 在INTEL8086中为XCHG。

硬件方案3：使用swap指令

使用Swap指令实现进程互斥的描述如下：

```
Boolean k = true;           // 初始化为1
Boolean use = false;        // 初始资源空闲
while (k != 0)
    Swap (&use, &k); //进入区
    Critial_region (); //临界区
    Use = 0;           //退出区
    Other_region ();   //剩余区
```

- 采用Swap指令与采用TSL指令类似，也会由于循环对换两个变量，造成忙等的情况。



MIPS中的spinlock

指令说明	指令定义	Alpha	MIPS64
寄存器和内存之间原子交换（用在锁和信号量上）	Temp<---Rd; Rd<---Mem[x]; Mem[x]<---Temp	LDL/Q_L; STL/Q_C	LL; SC

- MIPS 提供了 LL（Load Linked Word）和 SC（Store Conditional Word）这两个汇编指令来实现Spinlock(对共享资源的保护)。
- Read-modify-write 操作:当我们对某个Mem进行操作时, 我们首先从该 Mem中读回(read) data, 然后对该 data进行修改(modify)后, 再写回(write)该Mem。



MIPS中的spinlock

- LL 指令的功能是从内存中读取一个字,以实现接下来的 RMW (Read-Modify-Write) 操作;
- SC 指令的功能是向内存中写入一个字,以完成前面的 RMW 操作。
- LL/SC 指令的独特之处:
 - 当使用 LL 指令从内存中读取一个字后,如 $LL\ d, off(b)$, 处理器会记住 LL 指令的这次操作 (会在 CPU 的寄存器中设置一个不可见的 bit 位), 同时 LL 指令读取的地址 $off(b)$ 也会保存在处理器的寄存器中。
 - SC 指令,如 $SC\ t, off(b)$, 会检查上次 LL 指令执行后的 RMW 操作是否是原子操作 (即不存在其它对这个地址的操作), 如果是原子操作, 则 t 的值将会被更新至内存中, 同时 t 的值也会变为1, 表示操作成功; 反之, 如果 RMW 的操作不是原子操作 (即存在其它对这个地址的访问冲突), 则 t 的值不会被更新至内存中, 且 t 的值也会变为0, 表示操作失败。



MIPS中spinlock

Spin_Unlock(lockkey)

sync

sw zero, lockkey //给 lockKey 赋值为 0,
表示这个锁被释放

Spin_Lock(lockkey) // lockKey 是共享资源锁

l:

```
ll t0, lockkey    //将 lockKey 读入t0寄存器中
bnez t0, 1b        //比较 lockKey 是否空闲，如果该锁不可
                   //用的话则跳转到1:
li t0, 1           //给 t0 寄存器赋值为1
sc t0, lockkey     //将 t0 寄存器的值保存入 lockKey 中，
                   //并返回操作结果于 t0 寄存器中。
beqz t0, 1b        //判断 t0 寄存器的值，为0表示 li 的操作失
                   //败，则返回 1:重新开始；为1表示 li 的操
                   //作成功。
sync              //Sync 是内存操作同步指令，用来保证 sync
                   //之前对于内存的操作能够在 sync 之后的指
                   //令开始之前完成。
```

几个算法的共性问题

- 无论是软件解法（如Peterson）还是硬件（如TSL或XCHG）解法都是正确的，它们都有一个共同特点：当一个进程想进入临界区时，先检查是否允许进入，若不允许，则该进程将原地等待，直到允许为止。

1. 忙等待：浪费CPU时间

2. 优先级反转：低优先级进程先进入临界区，高优先级进程一直忙等



操作系统 Operating System

第四章 进程与并发程序设计(3) ——同步与互斥

沃天宇

woty@buaa.edu.cn

2021年4月15日





几个算法的共性问题

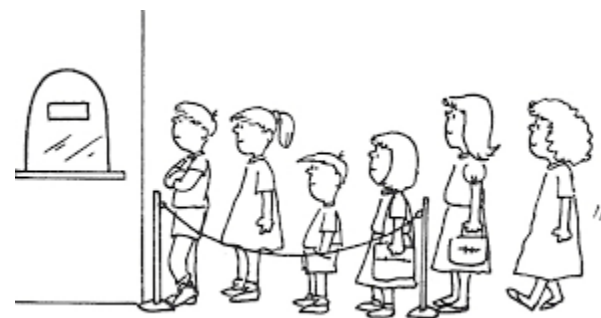
- 无论是软件解法（如Peterson）还是硬件（如TSL或XCHG）解法都是正确的，它们都有一个共同特点：当一个进程想进入临界区时，先检查是否允许进入，若不允许，则该进程将原地等待，直到允许为止。

1. 忙等待：浪费CPU时间

2. 优先级反转：低优先级进程先进入临界区，高优先级进程一直忙等

解决之道

- 忙等->阻塞
- Sleep
- Wakeup



• E.g. 银行排队

- 忙等：看到队很长，坚持排队
- 阻塞：看到队很长，先回家歇会儿（**sleep**），有空柜台了，大堂经理（**scheduler**）电话通知再过来（**wakeup**）

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

信号量机制

- 1965年Dijkstra。P (S) , V (S) 操作。P、V分别是荷兰语的test(proberen)和increment(verhogen))
- 其基本思路是使用一种新的变量类型(semaphore)
- 信号量只能通过初始化和两个标准的原语来访问，作为OS核心代码执行，不受进程调度的打断



经典信号量机制

ACT

**P(S): while $S \leq 0$ do skip
 $S := S - 1$;**

V(S): $S := S + 1$;





计数信号量机制

Type semaphore = record
 value : integer;
 L : list of process;
end

Procedure P(S)
 var S : semaphore;
 begin
 S.value := S.value - 1;
 if S.value < 0 then block(S.L);
 end

procedure V(S)
 var S : semaphore;
 begin
 S.value := S.value + 1;
 if S.value <= 0 then wakeup(S.L)
 end



信号量的使用：

- 必须置一次且只能置一次初值
- 只能由P、V操作来改变





物理意义

- S.value为正时表示资源的个数
- S.value为负时表示等待进程的个数
- P操作分配资源
- V操作释放资源。



信号量机制的实现

- 原子性问题；
 - 关中断、TS指令等
- PCB链表形式。

《现代操作系统》 P74



信号量的应用

- 互斥
 - 利用信号量实现进程互斥 ($S=1$)
- 同步
 - 利用信号量实现进程同步 ($S=0$)
 - 例如：描述进程执行的前趋/后继关系



互斥

P(S)
临界区
V(S)

P(S)
临界区
V(S)



同步

$P(S)$

代码

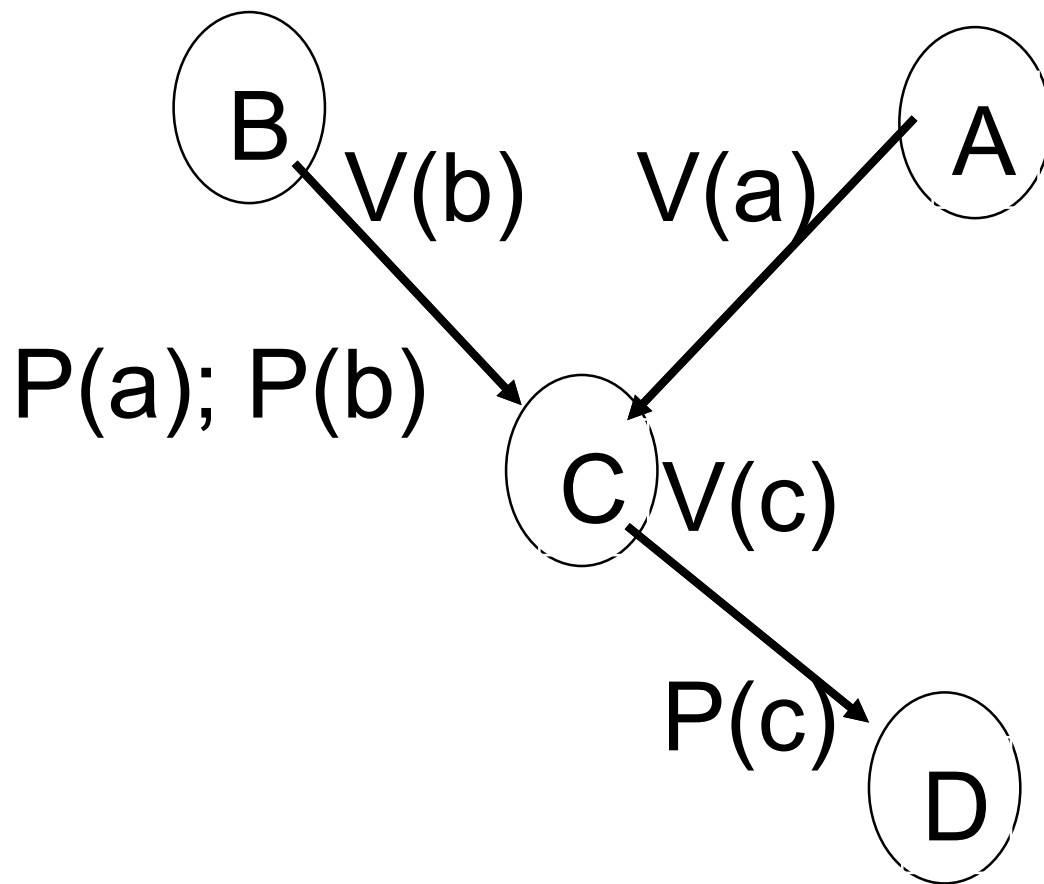
$V(S)$

代码





前趋关系



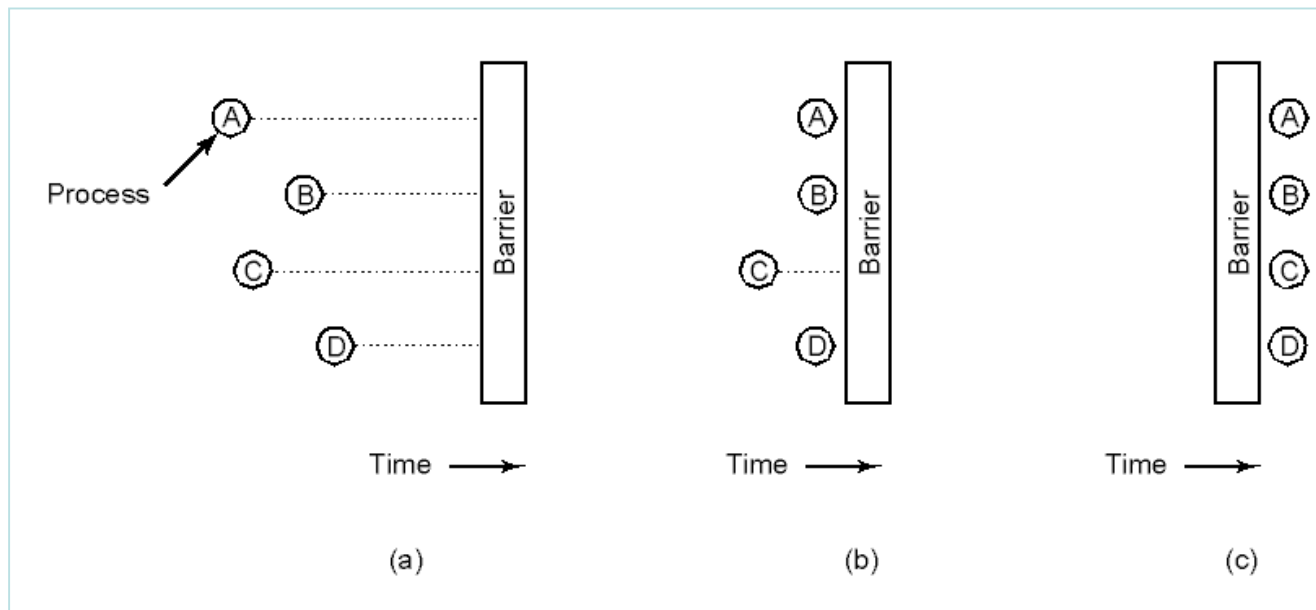


信号量在并发中的典型应用

应用	描述
互斥 (Mutual exclusion)	可以用初始值为1的信号量来实现进程间的互斥。一个进程在进入临界区之前执行semWait操作，退出临界区后再执行一个semSignal操作。这是实现临界区资源互斥使用的一个二元信号量。
有限并发 (Bounded concurrency)	是指有 n ($1 \leq n \leq c$, c 是一个常量) 个进程并发的执行一个函数或者一个资源。一个初始值为 c 的信号量可以实现这种并发。
进程同步 (Synchronizati on)	是指当一个进程 P_i 想要执行一个 a_i 操作时，它只在进程 P_j 执行完 a_j 后，才会执行 a_i 操作。可以用信号量如下实现：将信号量初始为0， P_i 执行 a_i 操作前执行一个semWait操作；而 P_j 执行 a_j 操作后，执行一个semWait操作。

多进程同步原语：屏障Barriers

• 用于进程组的同步



• 思考：如何使用信号量实现Barrier？

多进程同步原语：屏障Barriers

- 对rendezvous进行泛化，使其能够用于多个线程，用于进程组的同步
 - 科学计算中的迭代
 - 深度学习的卷进神经网络迭代
 - GPU编程中的渲染算法迭代
- 思考：如何使用信号量实现Barrier？

多进程同步原语：屏障Barriers

- 对rendezvous进行泛化，使其能够用于多个线程或进程组的同步
- 提示：

```
1 n = the number of threads
2 count = 0
3 mutex = Semaphore(1)
4 barrier = Semaphore(0)
```

- Count记录到达汇合点的线程数。mutex保护count，barrier在当所有线程到达之前都是0或者负值。到达后取正值。
- 思考：如何使用信号量实现Barrier？



一种低级通信原语：屏障Barriers

• 思考：如何使用PV操作实现Barrier？

- n = the number of threads
- `count = 0` //到达汇合点的线程数
- `mutex = Semaphore(1)` //保护count
- `barrier = Semaphore(0)` //线程到达之前都是0或者负值。到达后取正值
- `mutex.wait()`
- `count = count + 1`
- `mutex.signal()`
- `if count == n: barrier.signal()` # 唤醒一个线程
- `barrier.wait()`
- `barrier.signal()` # 一旦线程被唤醒，有责任唤醒下一个线程



“信号量集” 机制

Process A:

P(Dmutex);

P(Emutex);

Process B:

P(Emutex);

P(Dmutex);

Dmutex, Emutex = 1;

Process A: P(Dmutex);

Process B: P(Emutex);

Process A: P(Emutex);

Process B: P(Dmutex);



“信号量集” 机制

当利用信号量机制解决了单个资源的互斥访问后，我们讨论如何控制同时需要多个资源的互斥访问。信号量集是指同时需要多个资源时的信号量操作。

- “AND型” 信号量集
- 一般信号量集



AND型信号量集机制

- 基本思想：将进程需要的所有共享资源一次全部分配给它；待该进程使用完后在一起释放。

```
SP(S1, S2, ..., Sn)
  if S1=>1 and ... and Sn=>1 then
    for I :=1 to n do
      Si := Si - 1;
    endfor
  else
    wait in Si;
  endif
```

```
SV(S1, S2, ..., Sn)
  for I :=1 to n do
    Si := Si + 1;
    wake waited process
  endfor
```


一般“信号量集”机制

- 基本思想：在AND型信号量集的基础上进行扩充：进程对信号量 S_i 的测试值为 t_i （用于信号量的判断，即 $S_i \geq t_i$ ，表示资源数量低于 t_i 时，便不予分配），占用值为 d_i （用于信号量的增减，即 $S_i = S_i - d_i$ 和 $S_i = S_i + d_i$ ）

```
SP(S1, t1, d1, ... , Sn, tn, dn)
  if S1=>t1 and ... and Sn=>tn then
    for l :=1 to n do
      Si := Si - di;
    endfor
  else
    wait in Si;
  endif
```

```
SV(S1, d1, ... ,Sn, dn)
  for l :=1 to n do
    Si := Si + di;
    wake waited process
  endfor
```



一般“信号量集”机制

几个例子：

- $SP(S, d, d)$: 表示每次申请 d 个资源，当资源数量少于 d 个时，便不予分配。
- $SP(S, 1, 1)$: 表示互斥信号量。
- $SP(S, 1, 0)$: 可作为一个可控开关(当 $S \geq 1$ 时，允许多个进程进入临界区；当 $S=0$ 时禁止任何进程进入临界区)。



P.V操作的优缺点

- 优点:

- 简单，而且表达能力强（用P.V操作可解决任何同步互斥问题）

- 缺点:

- 不够安全；P.V操作使用不当会出现死锁；遇到复杂同步互斥问题时实现复杂

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

管程 (Monitor)

- 用信号量可实现进程间的同步，但：(1)加重了编程的负担；(2)同步操作分散在各个进程中，使用不当就可能死锁（如P、V操作的次序错误、重复或遗漏）。
- 管程：把分散的临界区集中起来，为每个可共享资源设计一个专门机构来统一管理各进程对该资源的访问，这个专门机构称为管程。
- 管程可以函数库的形式实现。相比之下，管程比信号量好控制。
- **管程是一种高级同步原语。**



管程的引入

- 1972年前后，在**E.W. Dijkstra**等工作的基础上，由**Brinch Hansen**和 **Tony Hoare**所提出，**Hansen**在并发的**Pascal**语言中首先实现了管程。
- 一个管程是由**过程、变量及数据结构**等组成的一个集合，它们组成一个特殊的模块或者软件包。
 - 局部于该管程的共享数据：资源状态
 - 局部于该管程的若干过程：对数据的操作
- **互斥**：**任一时刻，管程中只能有一个活跃进程**
- **管程是一种语言概念，由编译器负责实现互斥**



管程的实现

- 目标：为每个共享资源设立一个管程，对共享资源及其操作进行封装，从而简化对共享资源的互斥访问过程。类似于“面向对象”的观点。
- 局部控制变量（临界资源）：一组局部于管程的控制变量
- 初始化代码：对控制变量进行初始化的代码
- 操作原语（互斥）：对控制变量和临界资源进行操作的一组原语过程（程序代码），是访问该管程的唯一途径。
- 条件变量（同步）：每个独立的条件变量是和进程需要等待的某种原因相联系的，当定义一个条件变量x时，系统就建立一个相应的等待队列
 - **wait(x)**:把调用者进程放入x的等待队列
 - **signal(x)**: 唤醒x等待队列中的一个进程

管程的条件变量

- 由于管程通常是用于管理资源的，因而在管程内部，应当存在某种等待机制。当进入管程的进程因资源被占用等原因不能继续运行时使其等待。为此在管程内部可以说明和使用一种特殊类型的变量----条件变量。
- 每个条件变量表示一种等待原因，并不取具体数值——相当于每个原因对应一个队列。



条件变量与信号量的区别

- 条件变量的值不可增减，P-V操作的信号量值可增减
 - wait操作一定会阻塞当前进程；但P操作只有当信号量的值小于0时才会阻塞。
 - 如果没有等待的进程，signal将丢失；而V操作增加了信号量的值，不会丢失。
- 访问条件变量必须拥有管程的锁

多个进程同时在管程中出现

• 场景：

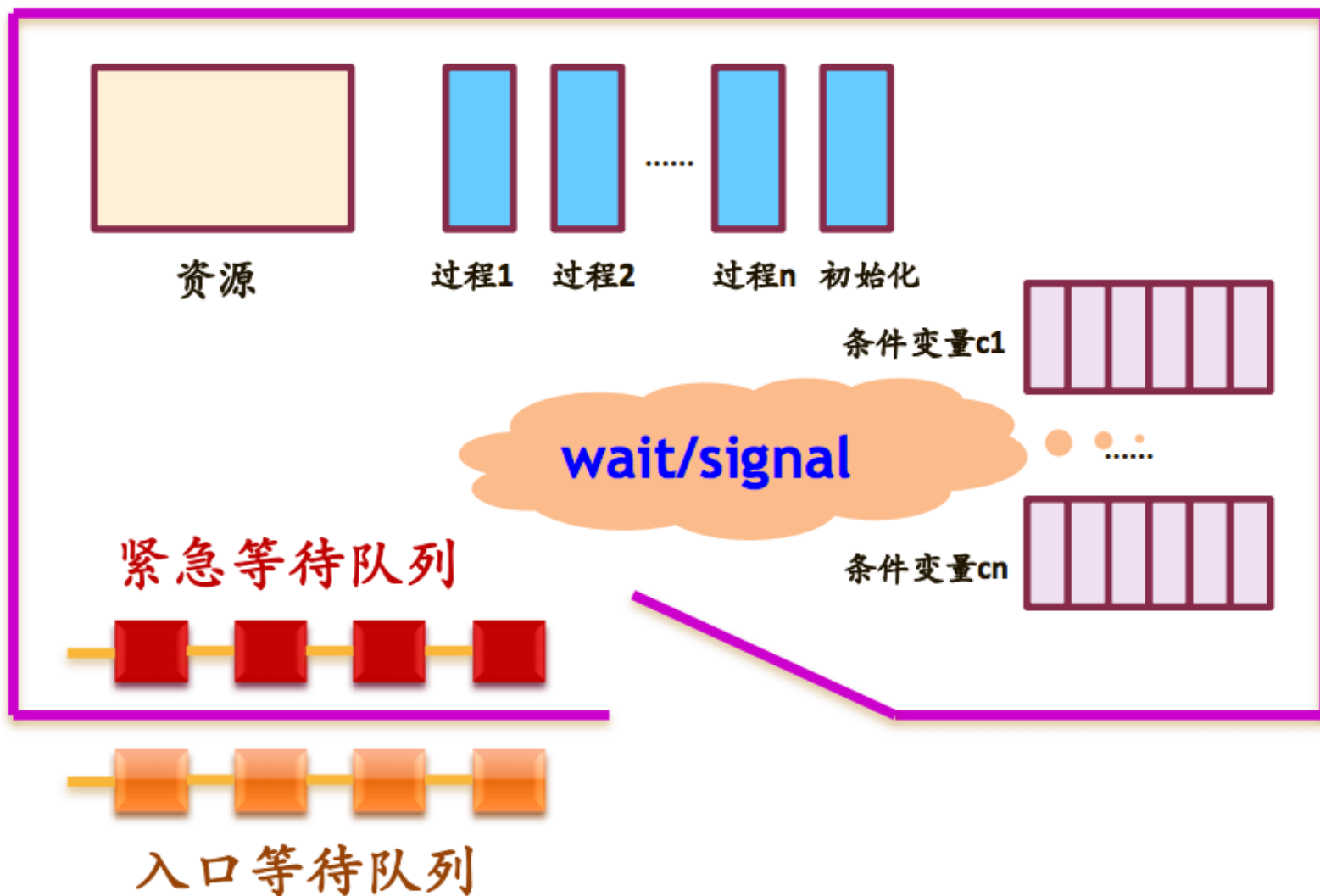
- 当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权。
- 当后面进入管程的进程执行唤醒操作时（例如**P**唤醒**Q**），管程中便存在两个同时处于活动状态的进程。

Hoare管程与Hansen管程

- 当 $\text{signal}(x)$ 时，如何避免管程中有两个活跃的进程
 - Hoare管程（阻塞式条件变量）：执行 signal 的进程等待，直到被释放进程退出管程或等待另一个条件
 - Mesa管程（非阻塞式条件变量）：被释放进程等待，直到执行 signal 的进程退出管程或者等待另一个条件
 - Hansen管程：执行 signal 的进程立即退出管程，即 signal 操作必须是管程中的过程体的最后一个操作



Hoare管程



Hoare管程

- **入口等待队列**(entry queue): 因为管程是互斥进入的, 所以当有一个进程试图进入一个已被占用的管程时它应当在管程的入口处等待, 因而在管程的入口处应当有一个进程等待队列, 称作入口等待队列。
- **紧急等待队列**: 如果进程 P 唤醒进程 Q, 则 P 等待 Q 继续, 如果进程 Q 在执行又唤醒进程 R, 则 Q 等待 R 继续, ..., 如此, 在管程内部, 由于执行唤醒操作, 可能会出现多个等待进程 (已被唤醒, 但由于管程的互斥进入而等待), 因而还需要有一个进程等待队列, 这个等待队列被称为紧急等待队列。它的优先级应当高于入口等待队列的优先级。

Hoare管程的同步原语

- 同步操作原语wait和signal：针对条件变量x，**x.wait()**将自己阻塞在x队列中，**x.signal()**将x队列中的一个进程唤醒。
 - **x.wait()**：如果紧急等待队列非空，则唤醒第一个等待者；否则释放管程的互斥权，执行此操作的进程排入x队列尾部（**紧急等待队列与x队列的关系：紧急等待队列是由于管程的互斥进入而等待的队列，而x队列是因资源被占用而等待的队列**）。
 - **x.signal()**：如果x队列为空，则相当于空操作，执行此操作的进程继续；否则唤醒第一个等待者，执行**x.signal()**操作的进程排入紧急等待队列的尾部。

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- **进程通信的主要方法**
- 经典的进程同步与互斥问题

进程间通信(Inter-Process-Comm)

- 低级通信：只能传递状态和整数值（控制信息），包括进程互斥和同步所采用的信号量和管程机制。缺点：
 - 传送信息量小：效率低，每次通信传递的信息量固定，若传递较多信息则需要进行多次通信。
 - 编程复杂：用户直接实现通信的细节，编程复杂，容易出错。
- 高级通信：适用于分布式系统，基于共享内存的多处理机系统，单处理机系统，能够传送任意数量的数据，可以解决进程的同步问题和通信问题，主要包括三类：管道、共享内存、消息系统。



IPC概述

- 管道（Pipe）及命名管道（Named pipe或FIFO）
- 消息队列（Message）
- 共享内存（Shared memory）
- 信号量（Semaphore）
- 套接字（Socket）
- 信号（Signal）



无名管道 (Pipe)

- 管道是**半双工**的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；
- **只能用于父子进程或者兄弟进程**之间（具有亲缘关系的进程）；
- 单独构成一种**独立的文件系统**：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且**只存在在内存中**。
- 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。



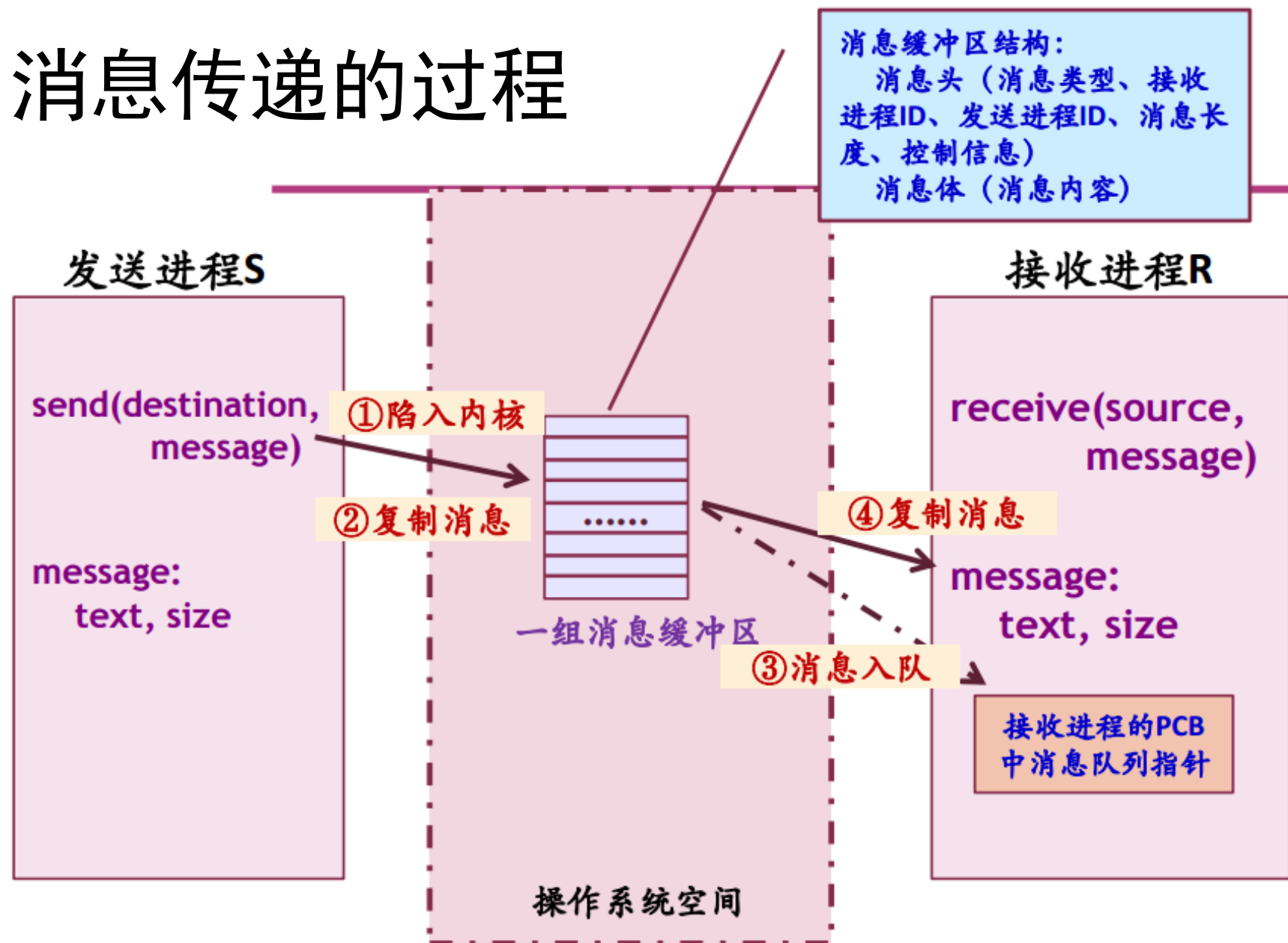
有名管道（Named Pipe或FIFO）

- 无名管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在**有名管道**提出后，该限制得到了克服。
- FIFO不同于管道之处在于它提供一个路径名与之关联，以FIFO的文件形式存在于文件系统中。这样，即使与FIFO的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过FIFO相互通信（能够访问该路径的进程以及FIFO的创建进程之间），因此，通过FIFO不相关的进程也能交换数据。
- FIFO严格遵循先进先出（first in first out），对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾。

消息传递 (message passing)

- 消息传递——两个通信原语 (OS系统调用)
 - **send (destination, &message)**
 - **receive(source, &message)**
- 调用方式
 - 阻塞调用
 - 非阻塞调用
- 主要问题：
 - 解决消息丢失、延迟问题 (TCP协议)
 - 编址问题: mailbox

消息传递的过程

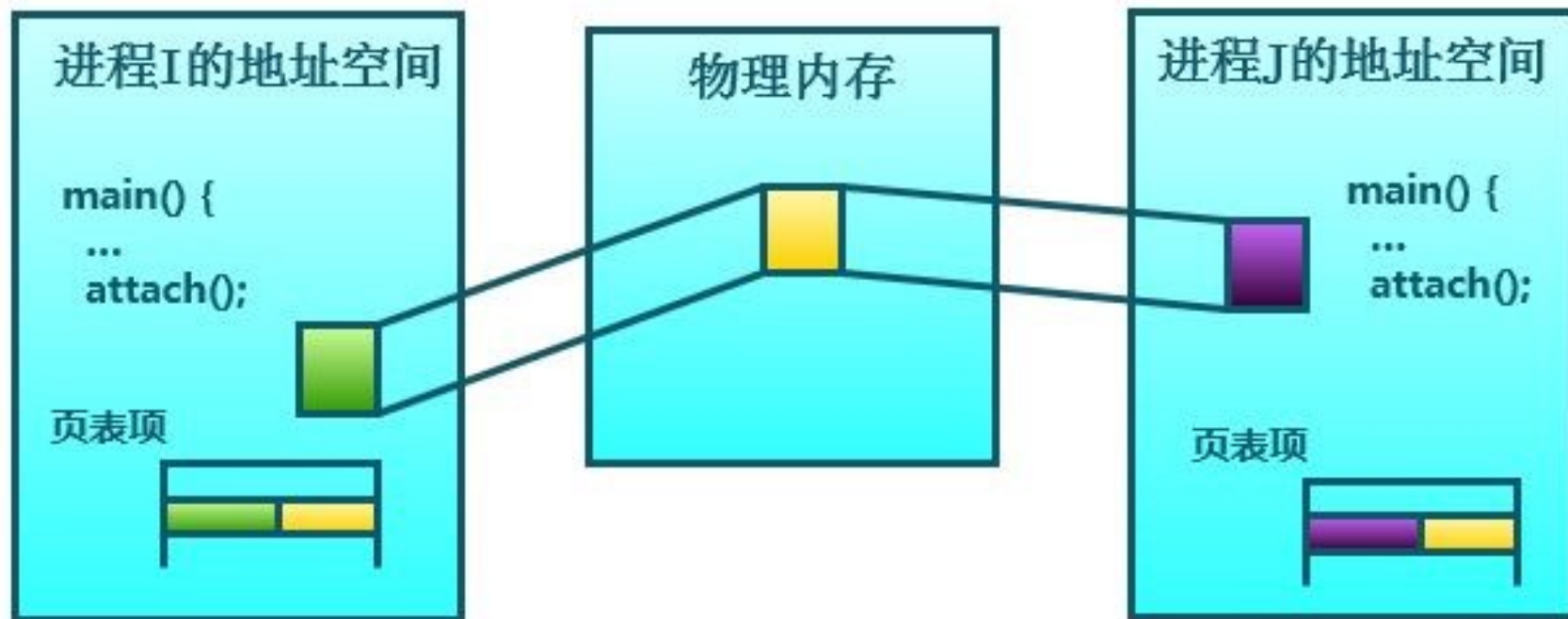




共享内存

- 共享内存是最有用的进程间通信方式，也是最快的IPC形式（因为它避免了其它形式的IPC必须执行的开销巨大的缓冲复制）。
- 两个不同进程A、B共享内存的意义是，**同一块物理内存被映射到进程A、B各自的进程地址空间**。
- 当多个进程共享同一块内存区域，由于**共享内存可以同时读但不能同时写**，则需要同步机制约束（互斥锁和信号量都可以）。
- 共享内存通信的效率 high（因为进程可以直接读写内存）。
- 进程之间在共享内存时，保持共享区域直到通信完毕。

共享内存机制





操作系统 Operating System

第四章 进程与并发程序设计(4) ——调度

沃天宇

woty@buaa.edu.cn

2021年4月20日



内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

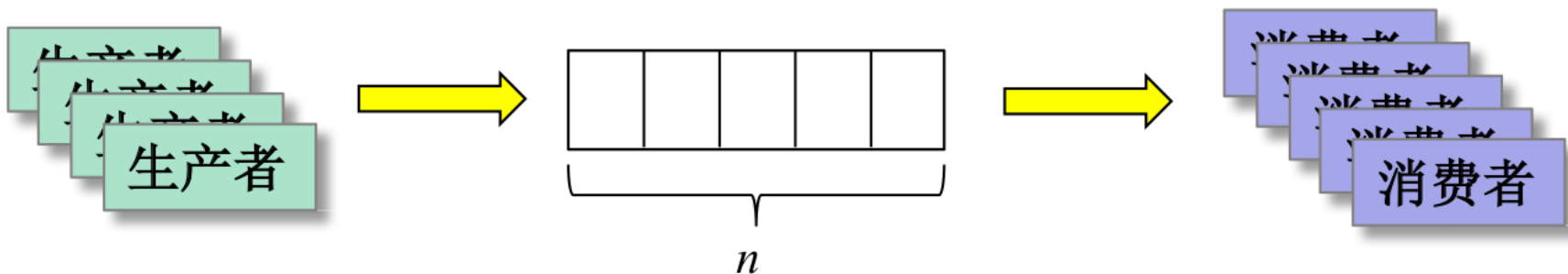


经典进程同步问题

- 生产者—消费者问题(the producer-consumer problem)
- 读者—写者问题(the readers-writers problem)
- 哲学家进餐问题(the dining philosophers problem)

生产者—消费者问题

- 问题描述：若干进程通过有限的共享缓冲区交换数据。其中，“生产者”进程不断写入，而“消费者”进程不断读出；共享缓冲区共有 N 个；任何时刻只能有一个进程可对共享缓冲区进行操作。



生产者—消费者问题

- 两个隐含条件：
 1. 消费者和生产者数量不固定。
 2. 消费者和生产者不能同时使用缓存区。
- 行为分析：
 - 生产者：生产产品，放置产品(有空缓冲区)。
 - 消费者：取出产品(有产品)，消费产品。
- 行为关系：
 - 生产者之间:互斥(放置产品)
 - 消费者之间:互斥(取出产品)
 - 生产者与消费者之间：互斥(放/取产品) 同步(放置——取出)

用PV操作解决生产者/消费者问题

- 信号量设置:

```
semaphore mutex =1;    //互斥
semaphore empty=N;     //空闲数量
semaphore full=0;      //产品数量
```

- 实际上，full和empty是同一个含义：
$$\text{full} + \text{empty} == N$$





用PV操作解决生产者/消费者问题

生产者

```
P(empty);  
P(mutex);  
    one >> buffer  
V(mutex)  
V(full)
```

消费者

```
P(full);  
P(mutex);  
    one << buffer  
V(mutex)  
V(empty)
```



用PV操作解决生产者/消费者问题

生产者

P(mutex);

P(empty);

one >> buffer

V(full)

V(mutex)

消费者

P(mutex);

P(full);

one << buffer

V(empty)

V(mutex)



如果两个V交换顺序呢？





完整的伪代码写法

```
Semaphore full = 0;  
Semaphore empty = n;  
Semaphore mutex = 1;  
ItemType buffer[0..n-1];  
int in = 0, out = 0;
```

```
main() {  
    Cobegin  
        producer();  
        consumer();  
    Coend  
}
```

```
producer() {  
    while(true){  
        生产产品nextp;  
        P(empty);  
        P(mutex);  
        buffer[in] = nextp;  
        in = (in + 1) MOD n;  
        V(mutex);  
        V(full);  
    }  
}
```

```
consumer() {  
    while(true){  
        P(full);  
        P(mutex);  
        nextc = buffer[out];  
        out = (out + 1) MOD n;  
        V(mutex);  
        V(empty);  
        消费nextc中的产品  
    }  
}
```




用管程解决生产者消费者问题

```
monitor ProducerConsumer
condition full, empty;
integer count;

procedure insert (item: integer);
begin
    if count == N then wait(full);
    insert_item(item); count++;
    if count == 1 then signal(empty);
end;

function remove: integer;
begin
    if count == 0 then wait(empty);
    remove = remove_item; count--;
    if count == N-1 then signal(full);
end;

count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item);
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item);
    end
end;
```

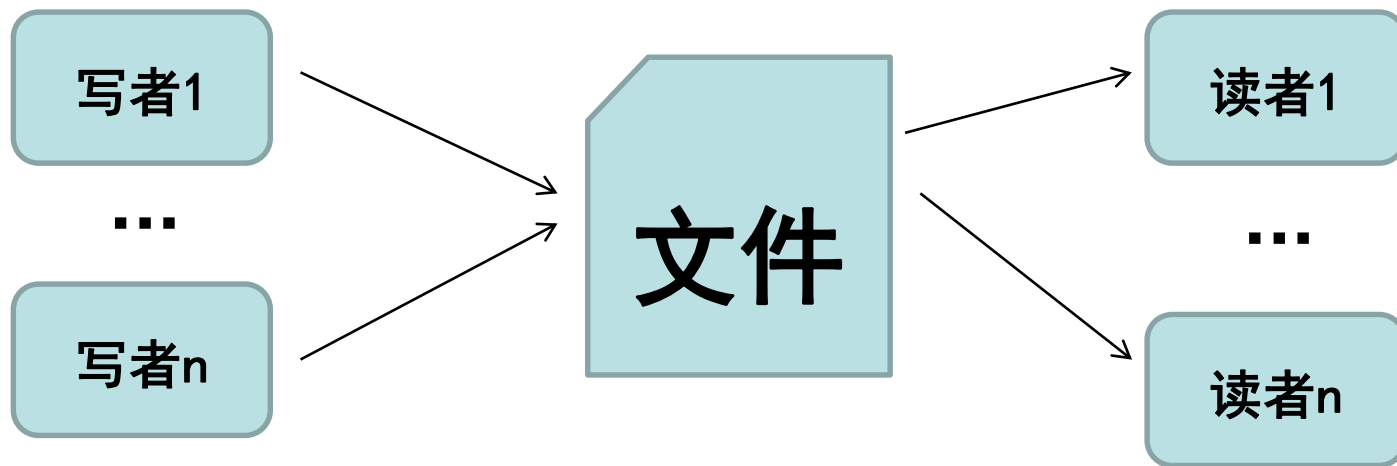


采用AND信号量集

- SP(empty, mutex),
- SP(full, mutex)
- 同学们可自行练习

读者—写者问题

- 问题描述：对共享资源的读写操作，任一时刻“写者”最多只允许一个，而“读者”则允许多个，即“读—写”互斥，“写—写”互斥，“读—读”允许



读者-写者问题分析

- 生活中的实例：12306订票
 - 读者：？
 - 写者：？
- 多个线程/进程共享内存中的对象
 - 有些进程读，有些进程写
 - 同一时刻，只有一个激活的写进程
 - 同一时刻，可以有多个激活的读进程
- 当读进程激活时，是否允许写进程进入？
- 当写进程激活时，？



采用信号量机制

- 基本框架

Writer

```
P(wmutex);  
写数据  
V(wmutex);
```

Reader

```
if first_reader? then P(wmutex);  
读数据  
if last_reader? then V(wmutex);
```

采用信号量机制

- wmutex表示“允许写”，初值是1。
- 公共变量readcount表示“正在读”的进程数，初值是0；
- mutex表示对readcount的互斥操作，初值是1。



Writer

```
P(wmutex);  
write  
V(wmutex);
```

Reader

```
P(mutex);  
if readcount=0 then P(wmutex);  
readcount := readcount + 1;  
V(mutex);  
read  
P(mutex);  
readcount := readcount - 1;  
if readcount=0 then V(wmutex);  
V(mutex);
```

- 如果有一个写者，第一个读者将阻塞，其他读者也会阻塞
- 如果一个读者激活，所有的读者都会陆续激活，哪个最先激活？
- 最后一个读者退出时，唤醒一个写者；如果没有写者呢？
- 如果读者和写者均阻塞在wmutex上，写者退出后，哪一个先激活？

完整伪代码，
参见教材P128
页。

Writer

```
P(wmutex);  
write  
V(wmutex);
```

Reader

```
P(mutex);  
if readcount=0 then P(wmutex);  
readcount := readcount + 1;  
V(mutex);  
read  
P(mutex)  
readcount := readcount - 1;  
if readcount=0 then V(wmutex);  
V(mutex)
```

回顾 “读者-
写者” 问题

该算法是对读者有利，还是对写者有利？

之前（对读者有利）算法

- 从读者、写者自身角度总结几条规则
- 写者：
 - 开始后，没有其他写者、读者可以进入
 - 必须确保没有正在读的读者，才能开始写
 - 写完之后允许其他读写者进入
- 读者开始读之前需要确保：
 - 第一个读者开始读，到最后最后一个读者结束读之间不允许有写进程进入写过程



对写者有利的算法？

- 在之前的算法基础上增加什么？
 - 写者出现后，新读者不允许抢在该写者之前进行读操作。
 - 也就是说：写者出现后，后续读者不允许执行 $rcount = rcount + 1$
 - 因此需要增加一个互斥信号量

读写公平的算法

Writer

```
P(rwmutex);  
P(wmutex);  
写数据  
V(wmutex);  
V(rwmutex);
```

Reader

```
P(rwmutex);  
P(mutex);  
if readcount=0 then P(wmutex);  
readcount := readcount + 1;  
V(mutex);  
V(rwmutex);  
read  
P(mutex);  
readcount := readcount - 1;  
if readcount=0 then V(wmutex);  
V(mutex);
```

注意这个位置
能不能放在read之后?

哲学家进餐问题 (the dining philosophers problem)

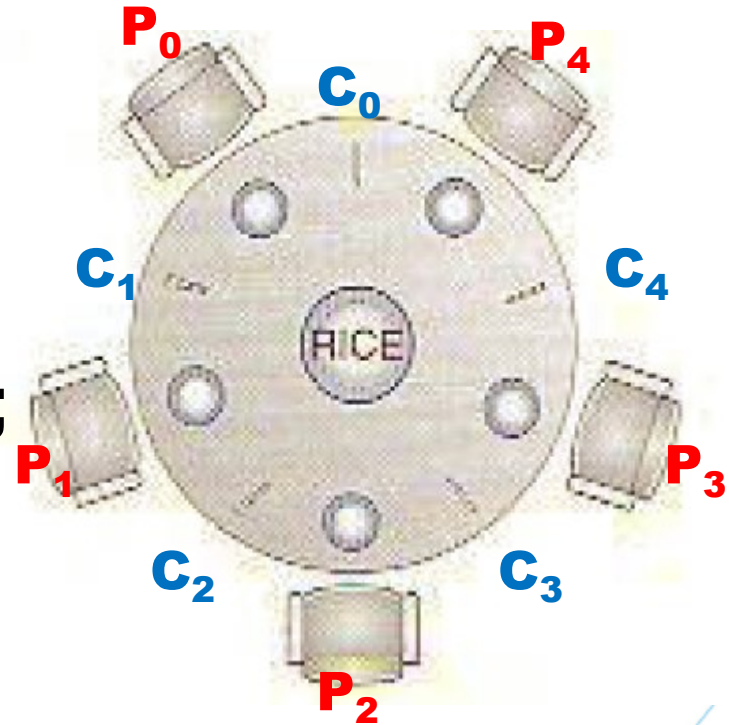
- 问题描述：（由Dijkstra首先提出并解决）
5个哲学家围绕一张圆桌而坐，桌子上放着5支筷子，每两个哲学家之间放一支；
- 哲学家的动作包括思考和进餐，进餐时需要同时拿起他左边和右边的两支筷子，思考时则同时将两支筷子放回原处。
- 如何保证哲学家们的动作有序进行？如：不出现相邻者同时进餐；不出现有人永远拿不到筷子。

基于P-V操作的方法

Var chopstick : array[0..4] of semaphore;

```

P(chopstick[i]);
P(chopstick[(i+1)mod 5]);
eat
V(chopstick[i]);
V(chopstick [(i+1)mod 5]);
think
    
```



有没有什么问题？

哲学家就餐问题的解题思路

- 至多只允许四个哲学家同时（尝试）进餐，以保证至少有一个哲学家能够进餐，最终总会释放出他所使用过的两支筷子，从而可使更多的哲学家进餐。设置信号量 $room=4$ 。（破除**资源互斥**）
- 对筷子进行编号，奇数号先拿左，再拿右；偶数号相反。（破除**循环等待**）
- 同时拿起两根筷子，否则不拿起。（破除**保持等待**）

基于AND信号量集的方法

```
Var chopstick : array[0..4] of semaphore;  
think  
SP(chopstick[(i+1)mod 5], chopstick[i]);  
eat  
SV(chopstick[(i+1)mod 5], chopstick[i]);
```



“生产者-消费者”扩展问题

- 设有一个可以装A、B两种物品的仓库,其容量无限大,但要求仓库中A、B两种物品的数量满足下述不等式:
 - $-M \leq A \text{物品数量} - B \text{物品数量} \leq N$
 - 其中M和N为正整数.
- 试用信号量和PV操作描述A、B两种物品的入库过程.



算法描述

Semaphore mutex=1, sa=N, sb=M;

cobegin

procedure A:

while(TURE)

begin

p(sa);

p(mutex);

A产品入库;

V(mutex);

V(sb);

end

procedure B:

while(TURE)

begin

p(sb);

p(mutex);

B产品入库;

V(mutex);

V(sa);

end

coend

理发师问题

- 理发店里有一位理发师、一把理发椅和 n 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，当一个顾客到来时，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，就坐下来等待，否则就离开。



互斥资源：理发师、顾客、椅子

同步约束：访问椅子、顾客唤醒理发师、理发师唤醒下一个位等待顾客

算法描述

信号量: customers=0;barbers=0;mutex=1

整型变量: waiting=0;

假设: CHIRS=10

理发师进程:

```
begin
  While(true)then
    begin
      P(customers);
      P(mutex);
      waiting=waiting-1;
      V(mutex);
      V(barbers);
      Cut hair();
    end
  end
end
```

顾客进程:

```
begin
  P(mutex);
  If (waiting<CHIRS)
  then
    begin
      waiting=waiting+1;
      V(mutex);
      V(customers);
      P(barbers);
      Get_haircut();
    end
  else
    begin
      V(mutex);
    end
  end
end
```

吸烟者问题

- 三个吸烟者在一间房间内，还有一个香烟供应者。为了制造并抽掉香烟，每个吸烟者需要三样东西：烟草、纸和火柴。供应者有丰富的货物提供。
- 三个吸烟者中，第一个有自己的烟草，第二个有自己的纸，第三个有自己的火柴。
- 供应者将两样东西放在桌子上，允许一个吸烟者吸烟。当吸烟者完成吸烟后唤醒供应者，供应者再放两样东西（随机地）在桌面上，然后唤醒另一个吸烟者。
- 试为吸烟者和供应者编写程序解决问题。

解题思路

- P: 供应者; S1,S2,S3:吸烟者;
a:烟; b:纸; c:火柴。
- 互斥资源: 桌子t
- 同步进程: P与S1, P与S2, P与S3

P

P(t);

放东西;

if(b & c) **V(S1);**

else if(a & c) **V(S2);**

else **V(S3);**

S1

P(S1);

取纸和火柴

V(t);

吸烟

小结

- 同步与互斥
- 基于忙等待的方法
- 基于信号量的方法
- 管程方法
- 经典同步与互斥问题
 - 生产者-消费者
 - 读者-写者
 - 哲学家就餐