操作系统原理

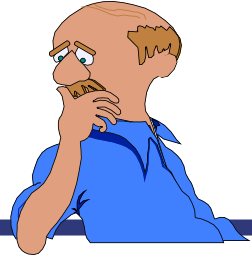
黄俊杰

junjiehuang@swu.edu.cn

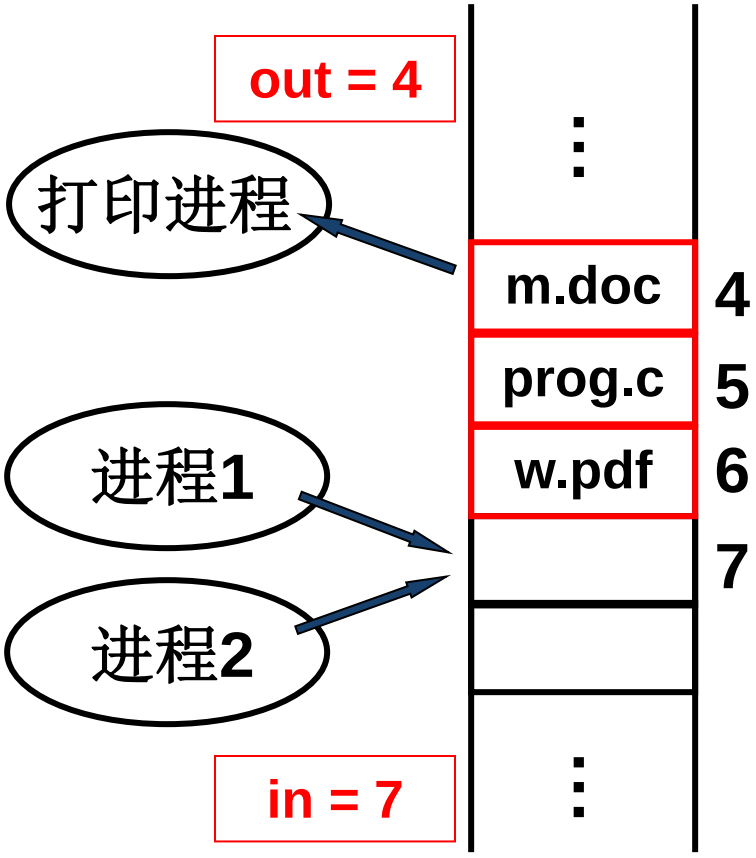
2024年9月

第3章 进程同步与通信

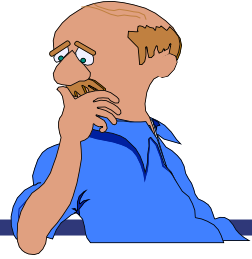
多个进程共同完成一个任务



- 想一想打印工作过程
  - 应用程序提交打印任务
  - 应用程序去做其它工作
  - 打印任务被放进打印队列
  - 打印进程从队列中取出任务
  - 打印进程控制打印机打印



生产者–消费者



共享数据

```
#define BUFFER_SIZE 10
typedef struct { . . . } item;
item buffer[BUFFER_SIZE];
int in = out = counter = 0;
```

BUFFER_SIZE:
缓冲区的大小

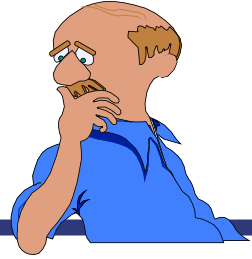
生产者进程（打印任务请求者）

```
while (true) {
    while(counter== BUFFER_SIZE)
        ;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
  }
```

消费者进程（打印进程）

```
while (true) {
    while(counter== 0)
        ;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
  }
```

生产者–消费者引出的一个问题



- 共享变量counter会出现语义错误
- 如生产者进程和消费者进程各执行一次

初始情况

```
counter = 5;
```

生产者P

```
register = counter;
register = register + 1;
counter = register;
```

消费者C

```
register = counter;
register = register - 1;
counter = register;
```

一个可能的执行序列

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

术语1: 竞争条件(Race Condition)

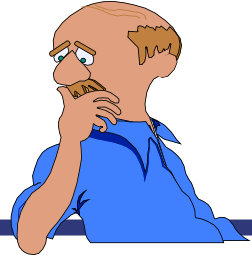
- **竞争条件:** 和时间有关的共享数据语义错误

第i次执行	第j次执行

P.register = counter;
P.register = P.register + 1;
C.register = counter;
C.register = C.register - 1;
counter = P.register;
counter = C.register;
	P.register = counter;
	P.register = P.register + 1;
	counter = P.register;
	C.register = counter;
	C.register = C.register - 1;
	counter = C.register;

  - 含义1: 多个进程并发访问和操作共享数据
  - 含义2: 和时间有关就是和调度顺序有关
  - 含义3: 这样的错误非常难于调试

避免竞争条件的一个尝试



- 在写counter时阻断其他进程访问counter

一个可能的但不期望执行序列

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

生产者P 给counter上锁

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

消费者C 检查counter锁

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

生产者P 给counter开锁

消费者C 给counter上锁

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

给counter开锁

一段代码一次只允许一个进程进入

如何实现：一段代码一次只允许一个进程进入？

术语2: 临界区(Critical Section)

- 临界区: 一次只允许一个进程进入的一段代码

生产者

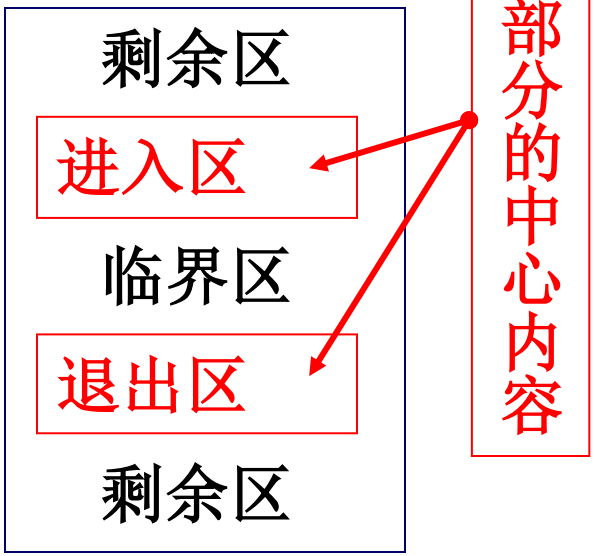
```
counter++;
```

消费者

```
counter--;
```

这就是临界区

- 临界区定义了进程间的协作协议，因此一个非常重要的工作:
  找出进程中的临界区代码



进程代码结构

如何进入临界区? – 进入原则

- **1. 互斥进入:** 如果一个进程在临界区中执行，则其他进程不允许进入
  - 这些进程间的约束关系称为互斥(mutual exclusion)
  - 这是临界区进入的基本原则(正确性保证)
- 一个好的临界区进入方法还需考虑…
  - **2. 空闲让进:** 若干进程要求进入空闲临界区时，应尽快使一进程进入临界区
  - **3. 有限等待:** 从进程发出进入请求到允许进入，不能无限等待
  - **4. 让权等待:** 当进程不能进入临界区时，应该释放处理机

进入临界区的一个尝试 – 轮换法

```
while (turn !=0)* ;
    临界区
turn = 1;
    剩余区
```

进程P_0

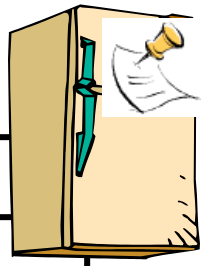
```
while (turn !=1) *;
    临界区
turn = 0;
    剩余区
```

进程P_1

- 满足互斥进入要求
- 问题: P_0完成后不能接着再次进入，尽管进程P_1不在临界区…(不满足有空让进)

进入临界区的又一个尝试

- 似乎没有任何头绪… 可借鉴生活中的道理



时间	丈夫	妻子
3:00	打开冰箱，没有牛奶了	
3:05	离开家去商店	
3:10	到达商店	打开冰箱，没有牛奶了
3:15	买牛奶	离开家去商店
3:20	回到家里，牛奶放进冰箱	到达商店
3:25	看电视 or 洗碗	买牛奶
3:30		回到家里，牛奶放进冰箱

- 上面的轮换法类似于值日
- 更好的方法应该是立即去买，留一个便条

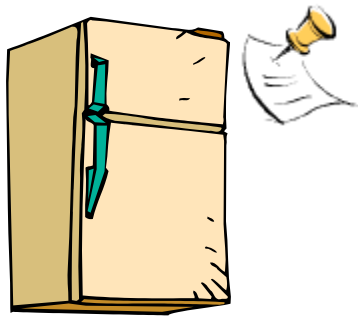
许多复杂的道理往往就埋藏在日常生活中！

进入临界区的又一个尝试 – 标记法

```
if(noMilk){
   if(noNote){
       leave Note;
       buy milk;
       remove note;
    }
}
```

计算机考虑问题的方式

```
leave Note;
if(noMilk){
   if(noNote){
       buy milk;
    }
}
remove note;
```



进程P_0

```
flag[0] = true;
while (flag[1]) ;
```

临界区

```
flag[0] = false;
```

剩余区

进程P_1

```
flag[1] = true;
while (flag[0]) ;
```

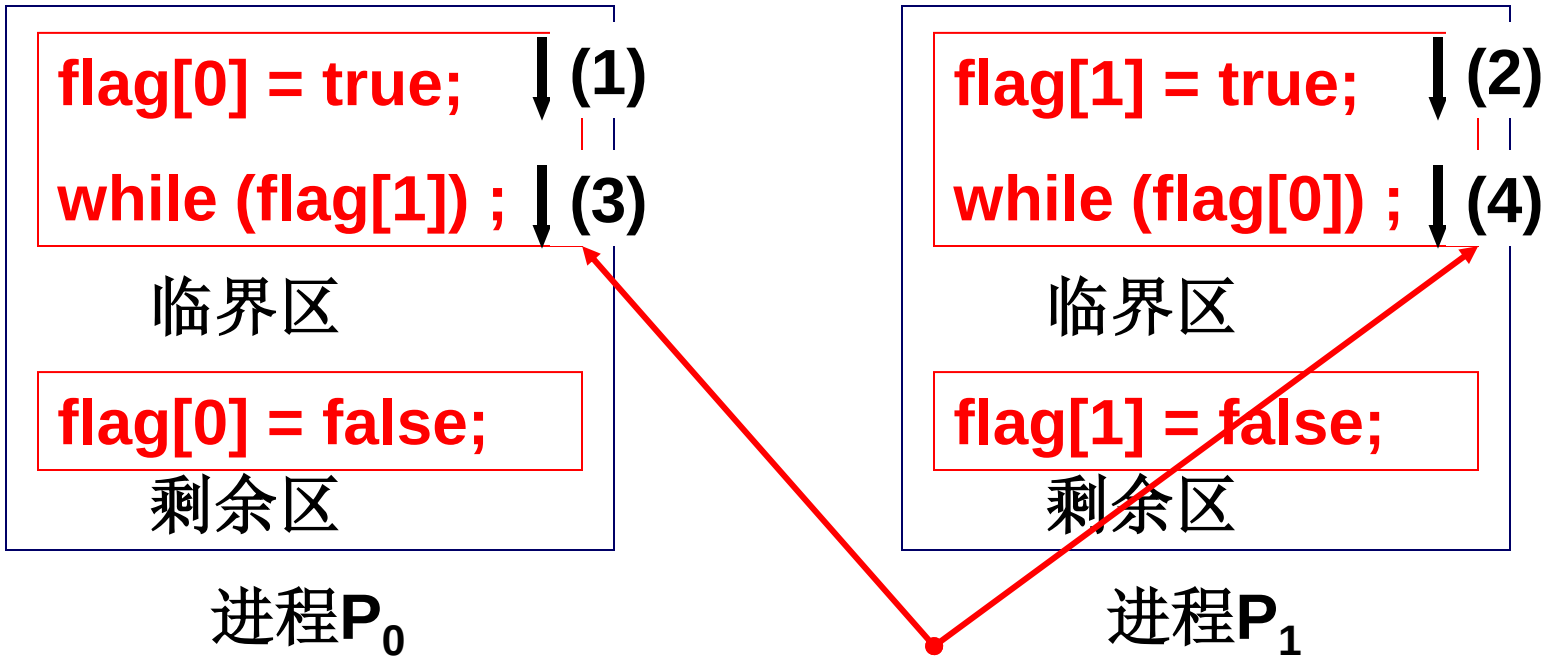
临界区

```
flag[1] = false;
```

剩余区

标记法能否解决问题?

- 考虑下面的执行顺序



- 此时P_0和P_1的进入请求会无限等待

进入临界区Peterson算法

- 结合了标记和轮转两种思想

```
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1) ;
```

临界区

```
flag[0] = false;
```

剩余区

进程P_0

```
flag[1] = true;
turn = 0;
while (flag[0] && turn == 0) ;
```

临界区

```
flag[1] = false;
```

剩余区

进程P_1

Peterson算法的正确性

- 满足互斥进入:

  如果两个进程都进入，则flag[0]=flag[1]=true，turn==0==1，矛盾!

- 满足有空让进:

  如果进程P_1不在临界区，则flag[1]=false，或者turn=0，都P_0能进入!

- 满足有限等待:

  P_0要求进入，flag[0]=true；P_1在临界区有turn=0，P_1不再改变turn值，所有后面的P_1会循环，P_0进入。

```
flag[i] = true;
turn = j;
while (flag[j] && turn == j) ;
        临界区
flag[i] = false;
        剩余区
```

进程P_i

一个显然的感觉是: 太复杂了!

进入临界区的硬件原子指令解法

- 提供硬件原子指令**TestAndSet**

```
boolean TestAndSet(boolean
   &target)
{
    boolean rv = target;
    target = true;
    return rv;
}
```

一次执行完毕

```
while(TestAndSet(&
lock)) ;
```

临界区

```
lock = false;
```

剩余区

进程P_i

- 用户需要查看硬件手册… 显然也不方便

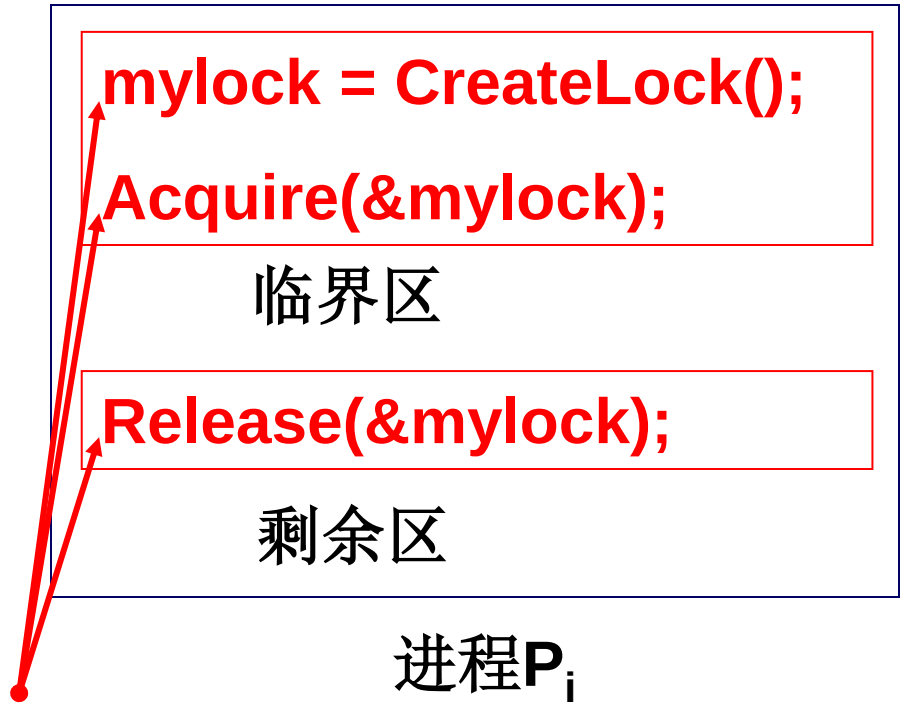
将这些进入临界区(互斥)方法整理一下…

- 给出了三种临界区进入方法(即互斥方法)
  - 软件方法：**Peterson**算法
  - 硬件原子指令**(TestAndSet)**方法
  - 其它（比如，关中断）

都不适合由普通用户实现

- 当然由操作系统承担这个复杂而易错的工作…

引入“锁”



都是系统调用

```
CreateLock(){
   int lock = 0;
   return lock;
}

Acquire(&lock){
while(
   TestAndSet(&lock));
}

Release(&lock){
   lock = 0;
}
```

- 交给系统许多事情就不一样了！

可惜的是，”锁”方案仍然存在问题…

生产者

```
while (true) {
  // Acquire(&lock);
   if(counter== BUFFER_SIZE)
         sleep();
   …
   counter = counter + 1;
   if(counter ==1) wakeup(消费者);
  // Release(&lock);  }
```

消费者

```
while (true) {
   // Acquire(&lock);
   if(counter== 0) sleep();
   …
   counter = counter – 1;
   if(counter == BUFFER_SIZE-1)
        wakeup(生产者);
   // Release(&lock); }
```

- 场景: **(1)** 在缓冲区装满后生产者又生产了两个**item**，此时有两个生产者进程在等待队列中。**(2)** 消费者进程执行**1**次，**counter==BUFFER_SIZE-1**，一个生产者进程被唤醒。**(3)** 消费者再执行**1**次，**counter=BUFFER_SIZE- 2**，另一个生产者进程没有被唤醒。

需要记录“积累下来”的唤醒操作个数

- 怎么记录? 当然是一个整型变量!
- 对这个整型变量的分析
  - 需互斥修改: 生产者要减少它，消费者要增加它
  - 会引起睡眠: 生产者将它减到0以后，生产者要睡眠
  - 会引起唤醒: 消费者要增加它，如果有睡眠的生产者，需要唤醒。
- 这是一个什么样的整型变量?

信号量 – 由伟大人物提出的伟大概念

- 信号量: 1965年，由荷兰学者Dijkstra提出的一种特殊整型变量。

- 信号量定义：1个数据结构+2个基本操作

```
struct semaphore
{
   int value; //记录唤醒操作个数
   PCB *queue;//等待在该信号量上的进程
}
P(semaphore s);     //消费唤醒操作
V(semaphore s);     //产生唤醒操作
```

信号量的P操作

```
P(semaphore s){
   s.value--;
   if(s.value < 0) {
      放入等待队列s.queue;}
}
```

等待某事件，
阻塞自己

- P的名称来源于荷兰语的proberen，即test

信号量的V操作

```
V(semaphore s){
   s.value++;
   if(s.value <= 0) {
      从s.queue上取出一个进程;
      将其放入就绪队列;}
   }
}
```

等待事件已发生，唤醒之

- V的名称也来源于荷兰语verhogen(increment)

术语3: 同步(Synchronization)

- **同步:** 多个进程按确定的协作顺序执行
- 同步的例子随处可见…

实例1:

司机	售票员
while (true){ 启动车辆； 正常运行； 到站停车；}	while (true){ 关门； 售票； 开门；}

实例2:

读者写者问题
两组并发进程：读者和写者，共享一个数据区。
(1)允许多个读者同时执行读操作；
(2)不允许读者、写者同时操作；
(3)不允许多个写者同时操作。

互斥与同步的区别

- **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：**合作进程之间协调彼此的工作，控制自己的执行速度，由此产生的相互合作，相互等待的制约关系，即操作是有序的。

信号量 vs. 锁

- 信号量是一个更一般意义的锁…
  - 锁: {0,1}　信号量: {-∞, … , -2, -1, 0, 1, 2, … , ∞}
  - 信号量即信号的数量，唤醒实际上就是发一个信号
  - 开锁动作也是发一个信号

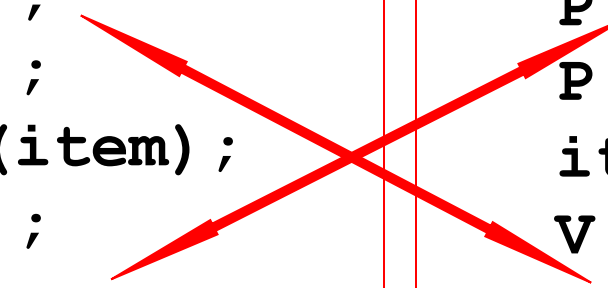
- 信号量：一个伟大的概念
  - 可以用来实现互斥
  - 可以记录资源个数(s.value >0)
  - 可以记录等待进程个数(s.value < 0)
  - 可以用来实现复杂的进程间同步关系

生产者 – 消费者的信号量解法

```
semaphore 有蛋糕= 0;
semaphore 有盘子= BUFFER_SIZE;
semaphore mutex = 1;
```

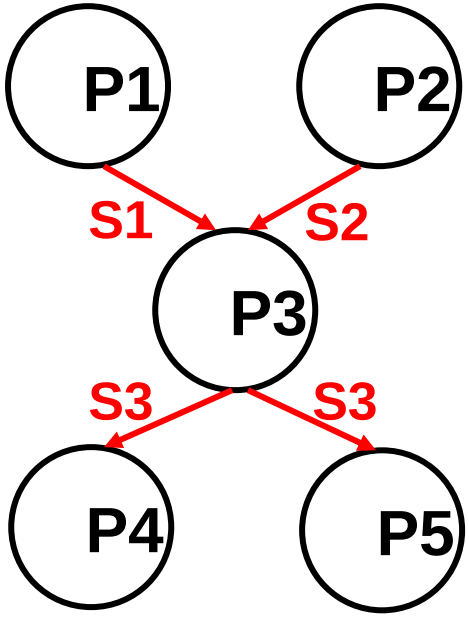
```
Producer(item) {
    P(有盘子);
    P(mutex);
    Enqueue(item);
    V(mutex);
    V(有蛋糕);
}
```

```
Consumer() {
    P(有蛋糕);
    P(mutex);
    item = Dequeue();
    V(mutex);
    V(有盘子);
}
```



- 终于给出了一个正确的解
- 那信号量 **mutex** 是干嘛的？

信号量可以控制进程的执行顺序




```
semaphore S1,S2,S3;
S1=0; S2=0; S3=0;
```

```
P1:...
   V(S1);
```

```
P2:...
   V(S2);
```

```
P3: P(S1); P(S2);
      …V(S3);
```

```
P4:
   P(S3);...
```

```
P5:
   P(S3); ...
```

信号量解法的分析

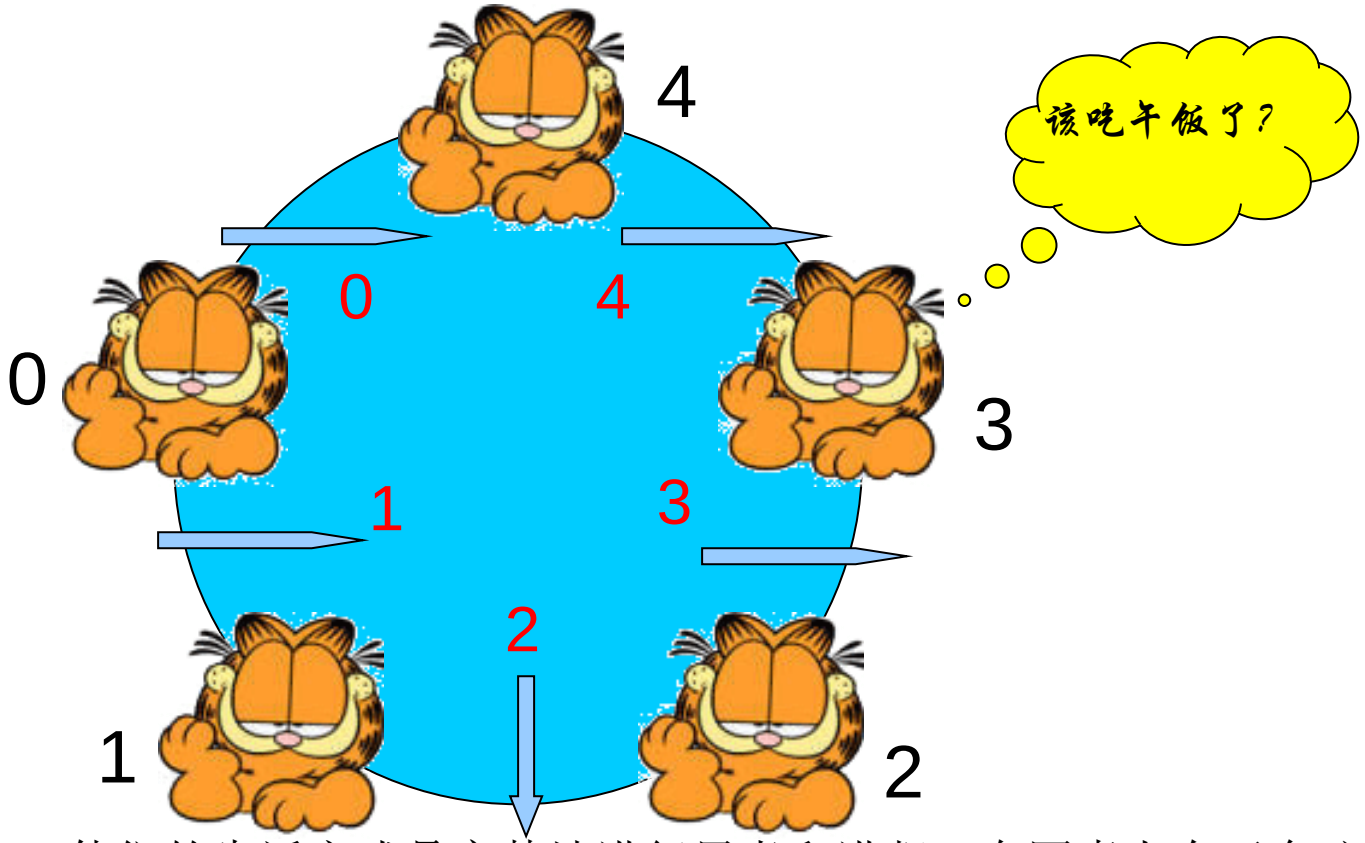
- 用户写起来不算难，但需要经过一定的训练。
- 仍然可能存在一些难于发现的错误...

```
Producer(item) {
    P(mutex);
    P(有盘子);
    Enqueue(item);
    V(mutex);
    V(有蛋糕);
}
```

```
Consumer() {
    P(mutex);
    P(有蛋糕);
    item = Dequeue();
    V(mutex);
    V(有盘子);
}
```

- 这样的顺序会出现什么问题？

哲学家进餐问题



有五个哲学家，他们的生活方式是交替地进行思考和进餐。在圆桌上有五个碗和五支筷子，平时一个哲学家进行思考，饥饿时便试图取用其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐。进餐毕，放下筷子又继续思考。

哲学家进餐解决方案的分析

```
semaphore  stick[5]={1,1,1,1,1};
 /*分别表示5支筷子*/
main (void){
     cobegin
          philosopher(0);
          philosopher(1);
          philosopher(2);
          philosopher(3);
          philosopher(4);
      coend
      }
```

```
procedure philopher(i){
    while(true)
    {
          思考;
          P(stick[i]);
          P(stick[(i+1)%5]);
          进餐;
          V(stick[i]);
          V(stick[(i+1)%5]);
     }
}
```

哲学家进餐问题：存在的问题

说明：

- **此算法可以保证不会有相邻的两位哲学家同时进餐，因为筷子只会被一个人拿起。**
- **若五位哲学家同时饥饿而各自拿起了左边的筷子？**
  - **这使五个信号量stick均为0，当他们试图去拿起右边的筷子时，都将因无筷子而无限期地等待下去，即可能会引起死锁。**

我们需要更高级更有效的方式！

3.3 AND信号量

- **用信号量解决了很多同步和互斥问题，但在解决问题的过程中，我们也发现还存在一些问题，如:**
  - 在生产者和消费者问题中两个P操作的位置不能颠倒;
  - 哲学家进餐问题中的死锁现象等。
- **这些问题的出现促使AND信号量的产生**

AND信号量定义

```
Swait(s1,s2,…,sn){
   if (s1 >= 1 && s2 >= 1 && … && sn>= 1)
    { /* 满足资源要求时*/
          for (i = 1; i <= n; i = i + 1)
                si = si -1;
   }
   else
    { /* 某些资源不能满足要求时*/
          block(si.queue )
          /*将进程投入第一个小于1的信号量的等待队列
          si.queue */ ;}
}
```

AND信号量定义

```
Ssignal(s1,s2,…,sn){
    for (i = 1; i <= n; i = i + 1){
      si = si +1;
      for ( 等待队列si.queue中的每个进程P)
          if ( 进程P通过Swait中的测试)
         {/* 通过检查，即资源够用 */
            wackup(p);  //唤醒进程P；}
         else
           {/* 未通过检查，即资源不够用 */
            进程P继续等待；}
    }
}
```

用AND信号量解决哲学家进餐问题

```
semaphore  chopstick[5] ={1,1,1,1,1};
void  philosopher ( int i )    /*哲学家进程*{
    while  (true){
        Swait(chopstick[i], chopstick[ (i + 1) % 5]);
        eat();      /* 进餐 */
        Ssignal(chopstick[i], chopstick[(i + 1) % 5]);
        think();      /* 思考 */
    }
}
```

引入更高级的工具 – 管程

- 问题关键: 在程序的许多地方都有信号量的操作
- 想法: 将共享数据、互斥(同步)操作集中隐藏起来，由辅助工具负责展开

```
monitor example
{ int counter;
  void producer(x);
  void consumer(x);}
```

```
while(true){
example.producer(x);}
```

```
while(true){
example.consumer(x);}
```

编译器将其展开

```
while(true)
{
   P(mutex);
    producer函数体;
    if(??)
       V(xx);
   V(mutex);
 }
```

管程描述

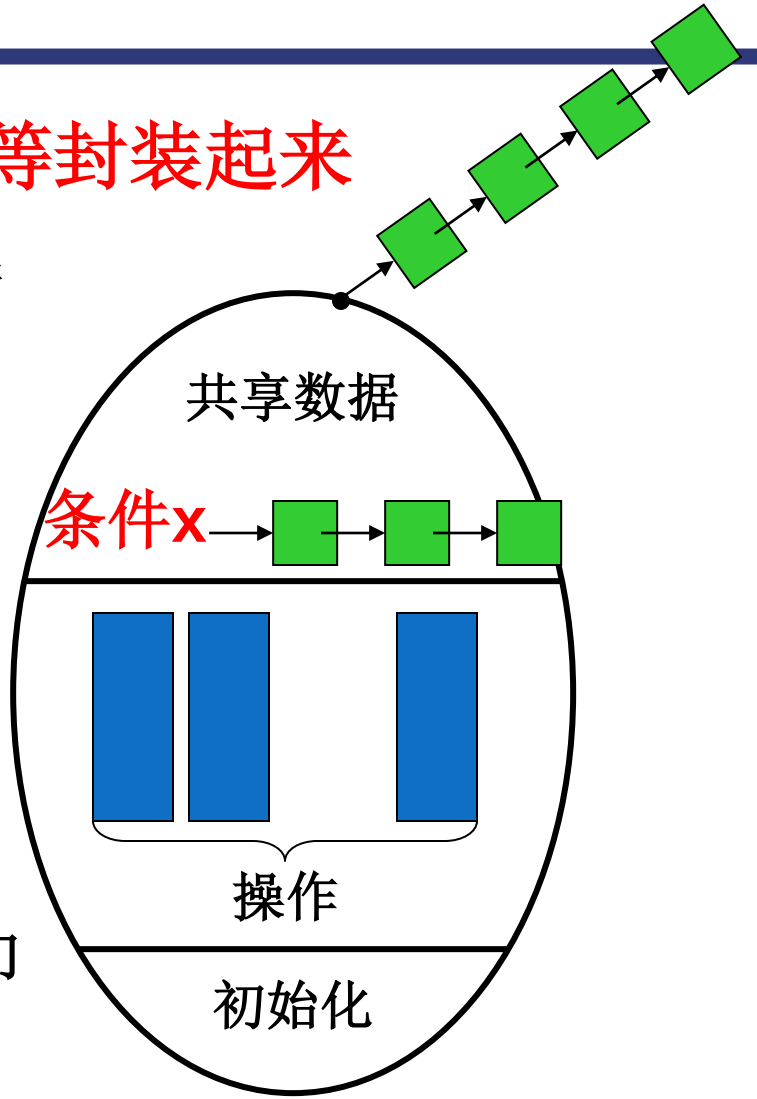
- 管程将共享数据、同步互斥等封装起来

隐式实现互斥

- 管程里的数据结构只能通过管程中的操作访问
- 一次只允许一个进程调用这些操作(互斥进入)

隐式实现同步

- 条件变量(condition)关联一个队列，用来实现同步
- 条件变量上的操作只有wait(即P)和signal(即V)

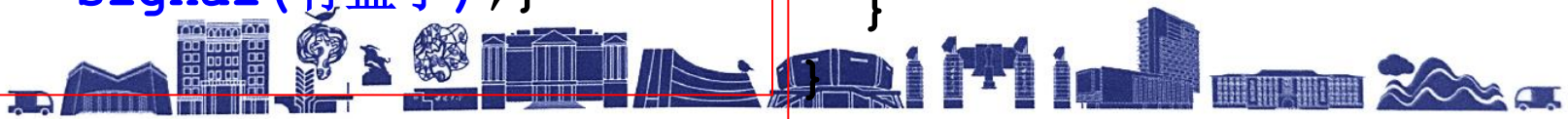


用管程解决生产者 – 消费者问题

```
monitor ProducerConsumer
{ int counter;
  condition 有盘子,有蛋糕;
  void put(x)
  { if(counter == N)
       wait(有盘子);
    将x放入缓冲区;counter++;
    if(counter == 1)
       signal(有蛋糕); }
  item get()
  { if(counter == 0)
       wait(有蛋糕);
    从缓存区取出x;counter--;
    if(counter == N-1)
       signal(有盘子);}
}
```

```
void producer()
{
  while(true){
    生产x;
    ProducerConsumer.put(x);
  }
}
```

```
void consumer()
{
  while(true){
       x =
       ProducerConsumer.get();
       Consume x
  }
}
```



上述概念的总结

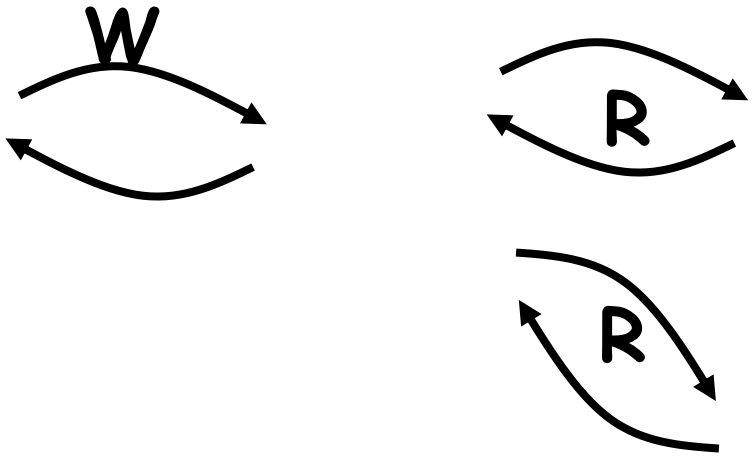
- 概念那么多，应该适时整理一下。

用户	应用程序
高层 工具	管程(monitors) 信号量(semaphores) 锁(locks)
硬件	Load/Store　Disable Ints　TestAndSet

- 锁、信号量、管程是系统提供给普通用户开发并发程序的高级工具…

做一个练习!

读者–写者问题



- 如共享一个数据库
  - 如果整个数据库设一个锁，则效率太低
  - 显然应该: 多个读者可以同时读，写者互斥、写者和读者互斥

读者–写者问题解的基本结构

```
Reader()
    如果有writer，则等待；
    访问数据库；
    离开，唤醒等待的writer；
```

```
Writer()
    如果有reader或writer，
则等待；
    访问数据库；
    离开，唤醒等待的reader
和writer；
```

- 需要定义读者和写者计数、互斥(同步)信号量
  - int AR/AW：活动的读者/写者个数
  - 互斥信号量mutex
  - 同步信号量有数据库

读者–写者问题的信号量解法(1)

```
Reader()
{
    P(Mutex);
    AR++;
    if(AR == 1)
        P(有数据库);
    V(Mutex);
    开始读数据库;
    P(Mutex);
    AR--;
    if(AR == 0)
        V(有数据库);
    V(Mutex);
}
```

```
Writer()
{
    P(有数据库);
    开始写数据库;
    V(有数据库);
}
```

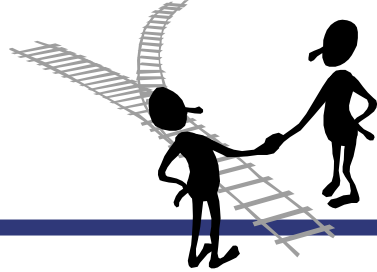
- 第一个读者阻塞在有数据库上
- 写者阻塞在有数据库上
- 读者优先，会造成写者等待很长时间

进程总结

- 并发 ⇒ 多个进程同时存在 ⇒ 相互影响
- 非原子操作共享变量 ⇒ 出现语义错误 ⇒ 竞争条件
- 竞争条件 ⇒ 临界区 ⇒ 互斥 ⇒ 临界区进入方法
- 复杂**Peterson**算法⇒强硬的关中断⇒硬件支持的**TestAndSet**
- 都不适合用户实现 ⇒ 封装成锁
- 一般的锁会盲等 ⇒ 引入睡眠 ⇒ 将锁一般化为信号量
- 信号量也容易出错 ⇒ 管程，将复杂性交给编译器

- 所有一切都是为了使用户更容易、使系统更好用(不出错)

3.5 进程通信

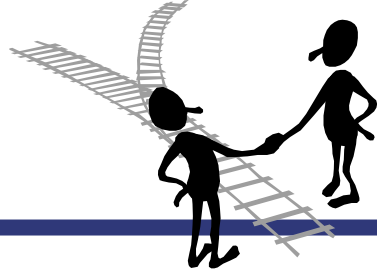


- 信号量作为进程同步和互斥工具是卓有成效的。但作为通信工具就不够理想。其原因为：
  - 效率低：一次只传一条消息。
  - 通信对用户不透明。
- 因此必须引入高级通信工具，解决进程之间大量的信息传递问题
- 进程之间互相交换信息的工作称为进程通信 **IPC（Inter Process Communication）**

进程需要通信的情况

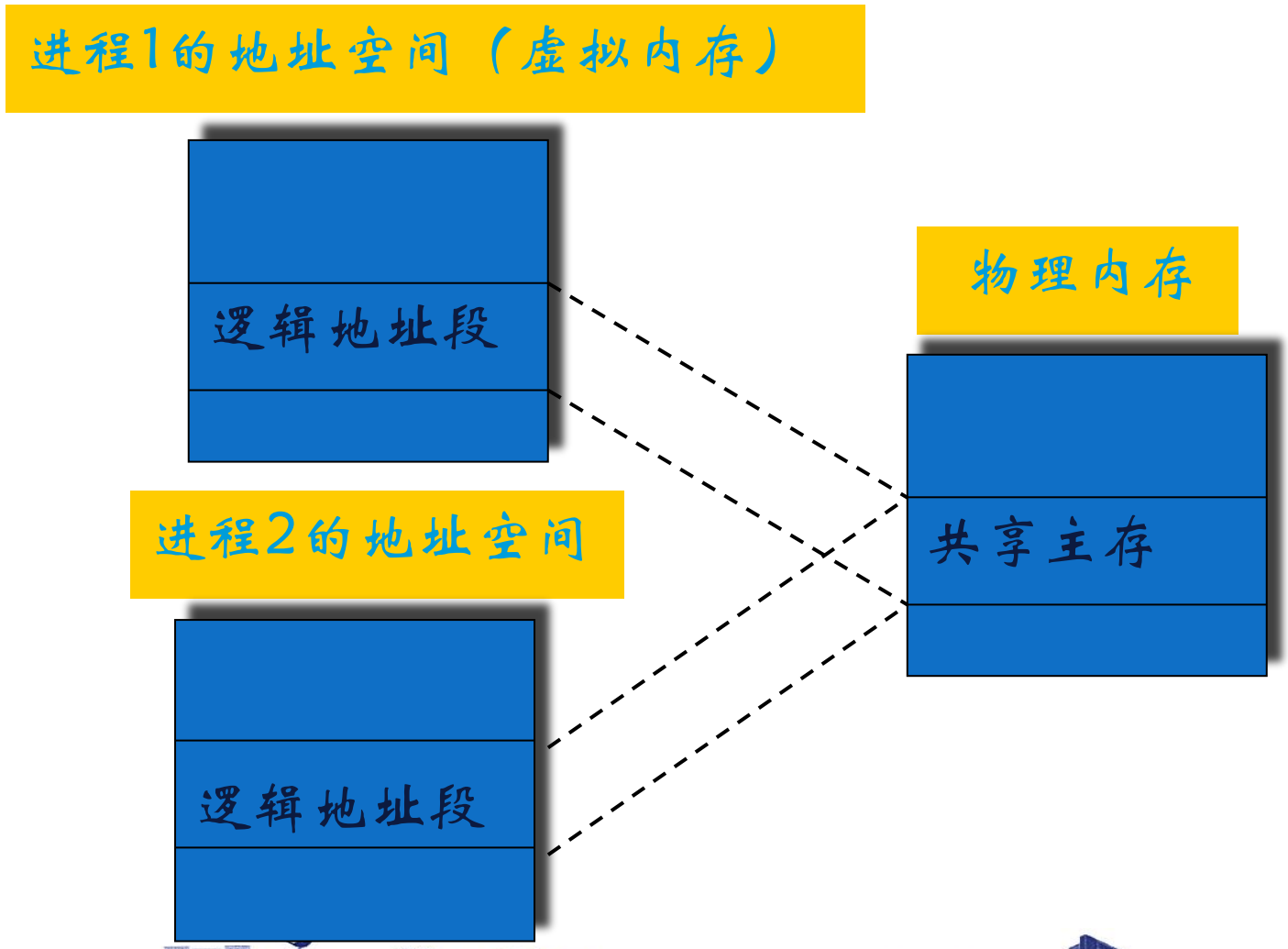
- 共享资源
- 协同工作
- 并发控制
- 通知进程
- 传递数据

进程通信的类型



- 共享存储器系统
- 管道通信
- 消息传递系统
  - 直接通信方式
  - 间接通信方式

共享主存通信机制



管道通信机制

- 管道(pipeline)是连接读写进程的一个特殊文件，允许进程按先进先出方式传送数据,也能使进程同步执行操作。
- 发送进程以字符流形式把大量数据送入管道，接收进程从管道中接收数据，所以叫管道通信。
- 管道的实质是一个共享文件，基本上可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。

消息传递

- 什么是消息传递（message passing）？
- 消息和消息传递机制
- 基本的消息传递原语send， receive

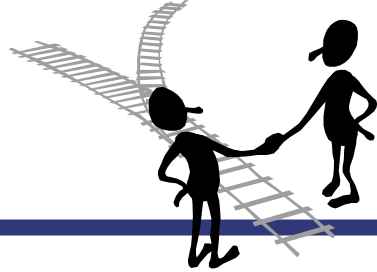
消息传递

- 采用消息传递机制后，一个正在执行的进程可在任何时刻向另一个正在执行的进程发送消息；一个正在执行的进程也可在任何时刻向正在执行的另一个进程请求消息。
- 一个进程在某一时刻的执行依赖于另一进程的消息或等待其他进程对发出消息的回答，那么，消息传递机制紧密地与进程的阻塞和释放相联系。消息传递就进一步扩充了并发进程间对数据的共享，提供了进程同步的能力。

直接通信

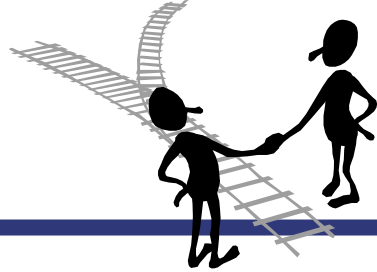
- 发送或接收消息的进程必须指出信件发给谁或从谁那里接收消息
- 原语send（P，消息）：把一个消息发送给进程P
- 原语receive（Q，消息）：从进程Q接收一个消息

消息缓冲队列-数据结构定义



```
//消息缓冲区定义
struct   message_buffer
{        char sender[30];          /*发送进程标识符*/
      int  size;                    /*消息长度*/
      char  text[200];                  /*消息正文*/
      struct  message_buffer  *next;    //指向下一个消息缓冲区的指针
}
//PCB中有关通信的数据项
struct   process_control
{     struct  message_buffer  *mq;    /*消息队列队首指针*/
   semaphore  mutex=1;          /*消息队列互斥信号量，初值为1*/
   semaphore  sm=0;    /*消息队列同步信号量，记录消息的个数.初值为0*/
}
```

消息缓冲队列-发送原语

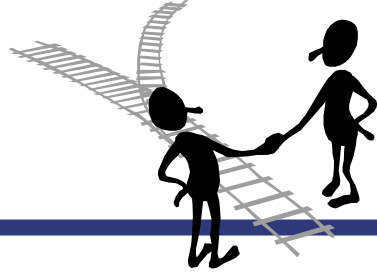


```
//发送原语
char receiver[30]; struct message_buffer a;
void send(receiver, a)
{ struct message_buffer i; struct process_control j;
    getbuf(a.size, i);          /*发送区a消息的长度申请一缓冲区i*/
    i.sender = a.sender;      i.size = a.size;
    i.text = a.text;       i.next = NULL;
    getid(PCB_set, receiver, j);  /*获得接收进程的进程标识符j*/
    P(j.mutex);
    Insert(j.mq, i);       /*将消息缓冲区i挂到的消息队列j.mq上*/
  V(j.mutex);
    V(j.sm);
}
```

干嘛用的？

干嘛用的？

消息缓冲队列-接收原语



```
struct  message_buffer  b;
void  receive(sender, b)
{  struct  message_buffer  i;
    struct  process_control  j;
    j = internal_name();          /*接收进程的内部标识符*/
    P(j.sm);
    P(j.mutex);
    remove(j.mq, i); /*从消息队列中摘下第一个消息缓冲区*/
  V(j.mutex);
    b.sender = i.sender;
    b.size = i.size;
    b.text = i.text;
}
```

这里有P操作

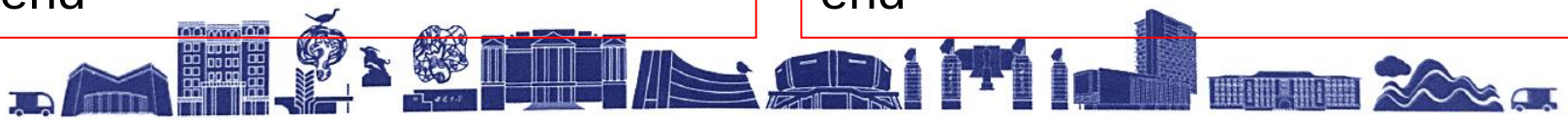
- 设公用信号量**mutex**为互斥信号量，初值为**1**；
- **SM**为接收进程的私用信号量，表示等待接收的消息个数，初值为**0**。

发送进程Send(m)

```
begin
    申请一个新消息缓冲区
    P(mutex)
    消息m→新消息缓冲区
    新消息缓冲区→队列
    V(mutex)
    V(SM)
end
```

接收进程Receive(n)

```
begin
    P(SM)
    P(mutex)
    从队列中摘下消息n
    将消息n拷贝到接收区
    释放内容已取走的缓冲区
    V (mutex)
end
```



间接通信

- 信箱是存放信件的存储区域，每个信箱可分成信箱特征和信箱体两部分。
- 信箱特征指出信箱容量、信件格式、指针等；信箱体用来存放信件
  - 原语send（A，信件）：把一封信件（消息）传送到信箱A
  - 原语receive（A，信件）：从信箱A接收一封信件（消息）

间接通信的实现

- 发送信件
  - 如果指定信箱未满，则将信件送入信箱中由指针所指示的位置，并释放等待该信箱中信件的等待者；否则发送信件者被置成等待信箱状态
- 接收信件：
  - 如果指定信箱中有信，则取出一封信件，并释放等待信箱的等待者，否则接收信件者被置成等待信箱中信件的状态

进程通信小结 (1/2)

- **管道（Pipe）及有名管道（named pipe）**：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，允许无亲缘关系进程间的通信；

- **消息队列（Message）**：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

进程通信小结 (2/2)

- **共享内存：**使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
- **信号量（semaphore）：**主要作为进程间以及同一进程不同线程之间的同步手段。
- **套接口（Socket）：**更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

附：读者–写者问题的信号量解法(2)

- 写者优先: 没有等待的写者时读者才进入

```
Reader()
{
    mutex.P();
    while((AW+WW)>0){WR++; mutex.V(); ToRead.P();
  mutex.P(); WR--}
    AR++; mutex.V();
    开始读数据库;
    mutex.P(); AR--;
    if(AR == 0 && WW > 0) ToWrite.V();
    mutex.V();
}
```

要保持警惕!

- mutex用来互斥访问AR、AW、WR、WW等
- 所有读者等在ToRead上，所有写者等在ToWrite上

附：读者–写者问题的信号量解法(2)

```
Writer()
{
    mutex.P();
    while((AW+AR)>0)
       {WW++; mutex.V(); ToWrite.P(); mutex.P(); WW--}
    AW++; mutex.V();
    开始写数据库;
    mutex.P();
    AW--;
    if(WW > 0) ToWrite.V();
    else if(WR > 0) ToRead.V();
    mutex.V();
}
```

写者优先

是否合适?
ToRead.broadcast

- 一个显然的感觉: 太多的信号量、太多的操作…

附：读者–写者问题的管程解法

```
monitor ReaderWriter
{ int AR, WR, AW, WW;
  condition ToRead,ToWrite;
  item read()
  { while((AW+WW)>0){ WR++; ToRead.wait(); WR--}
    AR++; 读出数据库元素; AR--;
    if(AR == 0 && WW > 0) ToWrite.signal();
  }
  write(x)
  { while((AW+AR)>0){WW++; ToWrite.wait(); WW--}
    AW++; 将x写入数据库; AW--;
    if(WW > 0) ToWrite.signal();
    else if(WR > 0) ToWrite.signal();
  }                          (或broadcast())
}
```

参考资料

- 廖剑伟，操作系统，西南大学，重庆，2023