进程和线程会考一个
资源分配的最小单位：进程
处理机调度的最小单位：线程

第二章 进程的描述与控制

2.1 前趋图和程序执行

2.1.1 前驱图

- 有向无循环图
- 直接前趋，直接后继
- **禁止循环**

2.1.2 程序顺序执行

1.程序的顺序执行

2. 程序顺序执行时的特征

- **顺序性**
- **封闭性**：计算结果不受外界影响
- **可再现性**：执行结构与执行速度无关，仅与初值有关

2.1.3 程序的并分执行

程序的并发执行

- **不存在前驱关系的程序之间才能并发执行。**

程序并发执行时的特征

- 间断性
- 失去封闭性
- 不可再现性

举例：

```
程序A ： n = n+1
程序B ： print(n); n = 0;
```

可能结果：

```
A快B慢： n+1 n+1 0
B快A慢： n 0 1
AB间隔： n n+1 0
```

2.2 进程的描述

2.2.1 进程的定义和描述

- 进程：程序关于某个数据集合的一次执行过程

进程控制块 (PCB)

- 创建进程的时候，就会创建PCB。放在内存PCB区。
- 内容
  - 记录PID (进程ID), UID (用户ID)
  - 记录进程分配了哪些资源 -> 用于对资源的管理
  - 记录进程的运行情况 -> 对进程的控制、调度
- 作用：PCB是进程存在的唯一标志

进程的实体

- **进程的实体 = PCB + 程序段 + 数据段**
- 进程是进程实体的运行过程。
- 进程是系统进行资源分配和调度的一个独立单位。

进程的特征

- 动态性（基本特性）：动态产生、变化和消亡。
- 并发性
- 独立性：独立运行、获得资源、调度
- 异步性

进程与程序的区别(重要)

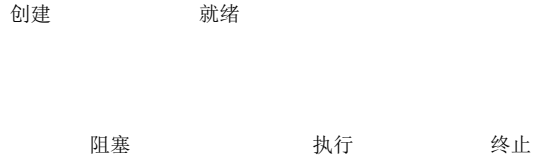
进程与线程的区别(重要)

2.2.2 进程的基本状态及转换

1. 三种基本状态

- **就绪状态**:
  - 除CPU，所有资源已分配。
  - 就绪队列
- **执行状态**
- **阻塞状态**
  - 执行进程被某事件打断，暂时无法继续执行。
  - 阻塞队列

2. 5种状态转换（重要）



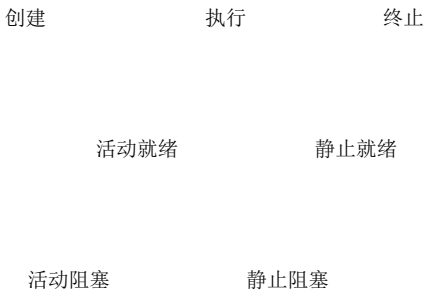
2.2.3 进程挂起和进程状态的转换

1.原因 (**了解)

- 终端用户的需要
- 父进程请求
- 负荷调节的需要
- 操作系统的需要

不能接受CPU调度，要激活之后才可以
Linux中子进程是父进程调度产生的

2. 引入挂起后，进程状态的转换（重要）



2.2.4 进程管理中的数据结构

1. PCB的作用

- 1. 作为独立运行的**基本单位的标志**
- 2. 能实现**间断性运行**方式
- 3. 提供**进程管理**所需要的信息
- 4. 提供**进程调度**所需要的信息
- 5. 实现与其他**进程的同步与通信**

2. PCB中的信息（要知道，但不会填空）

- **进程标识符**：唯一的标识一个进程
  - 内/外部标识符
- **处理机状态**
  - 通用寄存器、指令计数器、PSW、用户栈指针
- **进程调度信息**
  - 进程状态、进程优先级、其他信息、事件（阻塞原因）
- **进程控制信息**
  - 程序和数据的地址、进程同步和通信机制、资源清单、链接指针

3. PCB的组织方式

- 线性方式
- 链接方式
- 索引方式

2.3 进程控制

2.3.1 概念

原语

- 若干条指令组成的，用于完成一定功能的一个过程。

如何实现进程控制

- 利用PCB中的state表示：1 = 就绪态， 2 = 阻塞态
- 操作：
  - PCB2中state 设为1
  - 将PCB2从阻塞队列放到就绪队列

如何实现原语的“原子性”

- 利用开关中断

2.3.2 进程的创建

```
第一章的一些指令
ls
cat
ps
pstree 查看进程树
ln
chmod  rwx = 7
d
-
```

1. 进程的层次结构

- 父进程：创建进程的进程
- 子进程：被进程创建的进程

- 特点
  - 子进程可以**继承**父进程所拥有的资源
  - **子进程**被撤销，**归还资源**给父进程
  - **父进程**被撤销，必须**撤销**所有子进程
  - 进程不能拒绝子进程的继承权

2. 进程图（有向图）

3. 引起进程创建的事件

- 用户登录
- 作业调度
- 提供服务
- 应用请求

4. 进程的创建

- 进程创建原语Create()
- 创建过程：

- -> 申请空白PCB
- -> 分配资源
- -> 初始化进程控制块
- -> 将新进程插入就绪队列

2.3.3 进程的终止

1. 引起进程终止的事件

- **正常结束**
- **异常结束**
  - 越界错误
  - 保护错误
  - 非法指令
  - 特权指令
  - 运行超时
  - 等待超时
  - 算术运算错
  - I/O故障
- **外界干预**
  - 操作员/操作系统干预：Ctrl + Alt + delete
  - 父进程请求
  - 父进程终止

2. 终止的过程

- -> 找出被终止进程的PCB
- -> 若进程状态为运行态，置CPU调度标志为真，表示进程终止
- -> 若有子孙进程，终止其子孙进程并回收其资源
- -> 回收被终止进程的资源
- -> 回收被终止进程的PCB

2.3.4 进程的阻塞与唤醒

1. 引起进程阻塞和唤醒的事件

- 请求共享资源失败
- 等待某种操作完成
- 新数据尚未到达
- 新任务尚未到达

2. 进程阻塞过程

调用阻塞原语block()

- 调用阻塞原语
- 处于执行态，则**停止执行**，**修改状态**为阻塞态。
- PCB**插入阻塞队列**

3. 进程唤醒过程

调用唤醒原语wakeup()

- 把阻塞进程从等待该事件的**阻塞队列中移出**
- 设置进程状态为**就绪态**
- 将PCB**插入到就绪队列**中

2.3.5 进程的挂起和激活

1. 挂起的过程

调用挂起原语suspend()

- **检查进程状态**
  - 活动就绪 -> 静止就绪
  - 活动阻塞 -> 静止阻塞
- PCB**复制到指定的内存区域**
- 若挂起的进程正在执行，则转向调度程序**重新调度**。

2. 激活的过程

调用激活原语active

- 进程**从外存调入内存**
- **检查进程状态**
  - 静止就绪 -> 活动就绪
  - 静止阻塞 -> 活动阻塞
- **根据算法进行调度**

小结

- 更新PCB中的信息
  - a. **修改进程状态标志**
  - b. 剥夺当前运行进程的CPU使用权必然需要**保存期运行环境**
  - c. 某进程开始运行前必然要**恢复其运行环境**
- 将PCB插入合适的队列
- 分配/回收资源

2.4 进程同步（最重要！！！）

2.4.1 基本概念

1. 两种形式的制约关系

进程同步

- 异步性：进程不是一下做完，一步一步进行。
- **同步/直接制约关系**：完成某任务后，才能执行。

进程互斥

- **互斥/间接制约关系**：多个进程访问某个临界资源需要等待，一个一个访问。（例如：打印机）

2.临界资源

- 理解资源：一段时间内只允许一个进程访问的资源

3. 临界区

- 临界区：每个进程中访问临界资源的那段代码

访问临界区的程序设计：

- 对于访问的临界资源进行检查
- **进入区**：若此刻未被访问，**设正在访问的标志**
- **临界区**：访问临界资源
- **退出区**：将正在访问的标志恢复为未被访问的标志
- **剩余区**：其余部分

4. 同步机制应遵循的规则(要记)

- **空闲让进**
- **忙则等待**
- **有限等待**：等待进程，在有限时间内可以访问。
- **让权等待**：进程不能进入临界区时，应立即释放处理机，放置进程忙等待

2.4.2 硬件同步机制

1. 关中断

- 原理
  - 锁测试前关中断，完成所测试并上锁后，开中断
- 缺点
  - 滥用关中断权利，会导致严重后果
  - 时间过长，系统效率低
  - 不适用于多CPU系统

2. Test-and-Set指令实现互斥

```
// TS的机制
boolean TS(boolean *lock)
{
  boolean old;
  old = *lock;
  *lock = TRUE;   // 上锁
  return old;
}

// 利用TS指令实现的循环进程互斥结构
do{
  ...
  while TS(&lock); //循环，直到上锁，才能执行后面的程序
  critical section;
  lock = FALSE;
  remainder section;
}while(TRUE)
```

3. Swap指令实现进程互斥

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

// Swap实现的循环进程互斥结构
lock = FALSE;       // 全局变量lock
do{
  key = TRUE;       // 局部变量key
  do{
    swap(&lock, &key)
  }while(key != FALSE )  // key = TRUE，即lock = TRUE的时候持续交换，直到lock = FALSE进入进程
  临界区操作；
  lock = FALSE;
  ...
}while(TRUE)
```

硬件同步机制，处于忙等，不符合让权等待的原则

2.4.3 信号量机制

- wait(S) = P(S) 申请资源
- signal(S) = V(S) 释放资源

整型信号量

- P操作：<= 0 什么都不做；否则-1
- V操作：+1

```
// P操作： 申请资源
wait(S){
  while(S <= 0);
  S--;
}

// V操作：释放资源
signal(S){
  S++;
}
```

2. 记录型信号量（重要！！！！）

- P操作：
  - 先-1; 再判断 <0 :用block阻塞，使进程从运行态->阻塞态。
  - 因何事阻塞，则因何事唤醒
- V操作：
  - 先+1；判断：<=0 :则唤醒
- **S->value = 1**,转化为**互斥信号量**

```
typedef struct{
  int value;                                // 资源数目
  struct process_control_block *list;       // 进程链表指针
}semaphore

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

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

AND型信号量

- 一次申请所有的信号量，如果满足则申请，否则不予申请

```
Swait(S1,S2,S3...,Sn)
{
  while(TRUE)
  {
    if(S1 >=1 &&, ... && Sn >=1)
    {
      for(i = 1; i < n; i++)Si--;
      break;
    }
    else
    {
      等待队列，直到都满足才分配
    }
  }
}
Ssignal(S1,S2,S3...,Sn)
{
  while(TRUE)
  {
    for(i = 1; i < n; i++)Si++;
  }
}
```

4. 信号量集

- S：信号量
- t: 下限值,低于下限不予分配
- d: 需求量
- 特殊情况
  - **Swait(S,d,d)**
  - **Swait(S,1,1)**: 互斥信号量/可控
  - **Swait(S,1,0)**: 可控开关，S>=1时，允许多个进程进入某区域；S<0时，阻止任何进程进入某区域

2.4.4 信号量的应用

1.利用信号量实现进程互斥

```
semaphore mutex = 1
Pa(){
  while(1){
    wait(mutex);    // 进入区
    临界区;
    signal(mutex);  // 退出区
    剩余区;
  }
}
Pb(){
  while(1){
    wait(mutex);    // 进入区
    临界区;
    signal(mutex);  // 退出区
    剩余区;
  }
}
```

2.实现前驱关系

- 为每一对前驱关系各设置一个同步信号量：0.
- **在"前操作”之后执行V(S)**
- **在"后操作”之前执行P(S)**

```
P1(){
  code 1;
  code 2;
  V(S)    // S++,有资源，给P2放行
  ...
}

P2(){
  P(S);     //先执行到这里，S=0 无可用资源，会被阻塞。
  code 3;
  ...
}
```

2.4.5 管程机制

1. 管程的定义

- 一种特殊的资源管理模块

组成

- 管程的名字
- 局部于管程的共享数据结构说明
- 对该数据结构进行操作的一组过程。（**函数**）
- 对局部于管程的共享数据设置初始值的语句

特性

- 模块化
- 抽象数据类型
- 信息掩蔽

- 局部于管程的数据只能被局部于管程的过程所访问;

- 一个进程只有通过调用管程内的过程才能进入管程访问共享数据；
- **每次仅允许一个进程在管程内执行某个内部过程**；
- 管程通常是用来管理资源的，因而在管程中应当设有进程等待队列 以及相应的等待和唤醒操作。

2. 条件变量 condition c

- 条件变量：在管程内部可以说明和使用的一种特殊类型变量。（放置管程被占用，导致死等）
- 形式：condition c
- 操作：仅可执行，wait 和 signal
- **c.wait**
  - 正在**调用管程**的进程因c条件**阻塞或挂起**（生产者进程 发现缓冲区满），则调用c.wait
  - 将自己**插入**到c条件的**等待队列**， 并**释放管程**
- **c.signal**
  - **唤醒**正在等待的伙伴**进程**，对其伙伴正在等待的一个条件变量执行c.signal完成

多个进程处于管程中 (P唤醒Q)

- P等待、Q继续，直到Q退出或等待
- Q等待、P继续，直到P退出或等待
- 规定**唤醒操作为管程中最后一个可执行的操作**，所以P立即退出管程，Q马上被回复执行

2.5 经典进程的同步问题

2.5.1 生产者-消费者问题

1. 记录型信号量

- **wait和signal**必须成对出现。
- **对full和empty操作**必须成对出现。
- **wait**操作顺序**不可以颠倒**, signal可以颠倒。

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

void producer(){
  do{
    produce an item in nextp;
    ...
    wait(empty);      // 先资源，再操作。颠倒会导致死锁
    wait(mutex);
    buffer[in] = nextp;
    in = (in + 1) % n;
    signal(mutex);
    signal(full);
  }while(TRUE)
}

void consumer(){
  do{
    wait(full);
    wait(mutex);
    nextc = buffer[out];
    out = (out + 1) % n;
    signal(mutex);
    signal(empty);
    consumer the item in nextc;
    ...
  }while(TRUE)
}

void main(){
  cobegin
    producer();   consumer();
  coend
}
```

2. AND信号量

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

void producer(){
  do{
    produce an item in 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, empty);
    consumer the item in nextc;
    ...
  }while(TRUE)
}

void main(){
  cobegin
    producer();   consumer();
  coend
}
```

总结

- 同步的P操作在前，互斥的P操作在后

2.5.2 哲学家进餐问题

```
//定义每根筷子一个信号量
semaphore chopsticks[5] = {1, 1, 1, 1, 1}

// 第i位哲学家的活动
do{
  wait(chopsticks[i]);
  wait(chopsticks[(i + 1) % 5]);
  ...
  //eat
  ...
  signal(chopsticks[(i + 1) % 5]);
  signal(chopsticks[i]);
  ...
  //think
  ...
}while(TRUE);
```

- 问题
  - 死锁情况：每个哲学家均拿起左/右的筷子
  - 解决方法
    - 设置至多4人去拿筷子

- 仅左右手均有才可拿
- 奇数号哲学家先左后右；偶数相反。

2. AND型信号量

```
//定义每根筷子一个信号量
semaphore chopsticks[5] = {1, 1, 1, 1, 1}

// 第i位哲学家的活动
do{
  ...
  //think
  ...
  Swait(chopsticks[i], chopsticks[(i + 1) % 5]);
  ...
  //eat
  ...
  Ssignal(chopsticks[(i + 1) % 5], chopsticks[i]);
}while(TRUE);
```

2.5.3 读写者问题

> 读读共享；写写互斥；读写互斥

1. 记录型信号量

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

void reader(){
  do{
    wait(rmutex);
    if (readcount == 0) wait(wmutex);
    readcount++;
    signal(rmutex);
    ...
    //read
    ...
    wait(rmutex);
    readcount--;
    if (readcount == 0) signal(wmutex);
    signal(rmutex);
  }while(TRUE);
}

void writer(){
  do{
    wait(wmutex);
    ...
    //write
    ...
    signal(wmutex);
  }while(TRUE);
}

void main(){
  cobegin
    reader();     writer();
  coend
}
```

例题

例1. B0,B1,B3分别可放3，2，2个消息。初始，B0有3个消息。
每次Pi给Bi传消息（i=1,2,3)
用wait,signal写出P0,P1,P2的同步互斥流程

```
semaphore mutex0 = mutex1 = mutex2 = 1
semaphore full0 = 3 semaphore empty0 = 0
semaphore full1 = 0 semaphore empty1 = 2
semaphore full2 = 0 semaphore empty2 = 2

P0(){
  while(1){
    P(full0);
    P(mutex0);
    取数据;
    V(mutex0);
    V(empty0);
    加工模型;
    P(empty1);
    P(mutex1);
    放入商品;
    V(mutex1);
    V(full1);
  }
}
P1(){
  while(1){
    P(full1);
    P(mutex1);
    取数据;
    V(mutex1);
    V(empty1);
    加工模型;
    P(empty2);
    P(mutex2);
    放入商品;
    V(mutex2);
    V(full2);
  }
}

P2(){
  while(1){
    P(full2);
    P(mutex2);
    取数据;
    V(mutex2);
    V(empty2);
    加工数据;
    P(empty0);
    P(mutex0);
    放数据;
    V(mutex0);
    V(full0);
  }
}
```

例2. 有桥如图,车流如箭头所示,桥上不允许两车交会,但允许同方向车辆通行(即桥上可以有多个同方向的车)用 wait , signal 操作实现交通管理,以防桥上堵塞。

```
int countSN = countNS = 0        // 南向北，北向南的车计数
semaphore mutexSN = mutexNS = 1 //
semaphore bridge = 1             // 桥上的互斥信号量
StoN(){
  while(1){
    P(mutexSN);
    if(countSN == 0) // 判断是否是第一辆车
      P(bridge);
    countSN++;
    V(mutexSN);

    过桥...;

    P(mutexSN);       // 要先获取权限
    countSN--;
    if(countSN == 0)  //判断是否是最后一辆
      V(bridge);
    V(mutexSN)
  }
}

NtoS(){
  while(1){
    P(mutexNS);
    if(countNS == 0) // 判断是否是第一辆车
      P(bridge);
    countNS++;
    V(mutexNS);

    过桥...;

    P(mutexNS);       // 要先获取权限
    countNS--;
    if(countNS == 0)  //判断是否是最后一辆
      V(bridge);
    V(mutexNS)
  }
}
```

2.6 进程通信

了解四大通信类型和原理

- **低级通信**
  - 效率低
  - 对用户不透明
- **高级通信**
  - 使用方便
  - 高效地传送大量数据
  - **共享存储器系统、管道通信系统、消息传递系统、客户机-服务器系统**

2.6.1 进程通信的类型

1. 共享存储器系统

基于共享数据结构的通信方式（低级通信）

- 原理
  - **OS**仅提供**共享存储器**
  - **程序员**负责**对公用数据结构的设置**以及**对进程间同步的处理**
- 特点：
  - 低效
  - 只能传输少量数据

基于共享存储区的通信方式（高级通信）

- 原理
  - 在存储器中划出一块**共享存储区**，诸进程可通过对共享存储区中数据的读或写来实现通信
  - **数据的形式、存放位置**都由**进程控制**， 而不是操作系统。
- 特点
  - 高效
  - 数据传输量大

2. 管道通信 （高级通信）

- 原理
  - 基于共享文件（pipe文件）
  - 只能实现**单向的传输。**（某段时间内）互斥访问
  - 管道写满时，写进程被阻塞。读完时，读进程被阻塞；没写完，就不允许写，没读空，就不允许写。
  - **读进程最多只有一个**，数据读完，就从管道中被抛弃。

3. 消息传递系统 （高级通信）

- 原理
  - 利用原语进行数据发送
- 分类
  - 直接通信方式：OS原语直接
  - 间接通信方式：通过共享实体（邮箱）

2.6.2 消息传递通信的实现方式

1. 直接消息传递系统（直接通信）

直接通信原语

- Send(Receiver,message)
- Receive(Sender,message)

消息的格式

- 定长
- 变长：方便用户

同步方式

- 发送进程阻塞、接收进程阻塞
- 发送进程不阻塞、接收进程阻塞
- 发送进程和接收进程均不阻塞

通信链路

- 根据通信链路的建立方式
  - **显示连接**
    - 先用 “**建立连接**”命令(原语) **建立一条通信链路**，使用完**后拆除** 链路——用于**计算机网络**
  - **隐式连接**
    - 发送进程无须明确提出建立链路的要求，**直接利用**系统提供的发送命令(**原语**)，系统会**自动**地为之**建立一条链路。**——用于**单机系统**
- 根据通信方式
  - 单向通信链路
  - 双向链路

2. 信箱通信（间接通信）

- 信箱用来暂存发送进程发送给目标进程的消息，接收进程则从信箱中取出发送给自己的消息。
- 消息在信箱中可**安全保存**，只允许核准的目标用户随时读取
- 利用信箱通信方式，既**可实时通信**，又**可非实时通信**

2.6.3 直接消息传递系统实例

2.7 线程

2.7.1 线程引入

- 传统：
  - 资源分配、调度的基本单位：进程
- **引入线程**：
  - 资源分配的基本单位：进程
  - **调度和分派的基本单位**：**线程**

2.7.2 线程与进程的比较

1. 调度的基本单位

- 进程：
  - 每次调度，都需要切换上下文，开销大。
- 线程：
  - **同进程，切换线程，不引起进程切换。**
  - 不同进程，切换线程，引起进程切换。

2. 并发性

- 同/不同进程中的线程，支持多线程并发

3. 拥有资源

- 仅有必不可少的一点资源
- 可以

4. 独立性

- 独立性比进程弱

- 线程A可以被其他线程读，写。

5. 系统开销小

6. 支持多处理机系统

2.7.3 线程的状态和线程控制块

1. 线程运行的三个状态

- 执行状态
- 就绪状态
- 阻塞状态

2. 线程控制块TCB

- 线程标识符
- 一组寄存器
- 线程运行状态
- 优先级
- 线程专有存储区
- 信号屏蔽
- 堆栈指针

2.8 线程的实现

2.8.1 线程的实现方式

1. 内核级线程

- 线程管理工作：CPU**内核**实现
- 线程切换：**核心态**
- 调度：以**线程**为单位
- 优点：
  - 同进程中多线程，并行执行
  - **一个线程被阻塞，其他线程可运行**
  - 数据结构和堆栈小，**切换快，开销小**
  - 内核支持多线程技术，提高OS执行速度和效率
- 缺点：
  - 用户的线程切换，开销大：用户态->核心态

2. 用户级线程

- 线程管理工作：**用户空间**实现
- 线程切换：**用户态**
- 调度：以**进程**为单位
- 优点：
  - 切换无需变态， 开销小
- 缺点：

- **一个用户级线程被阻塞，整个进程都被阻塞。**

3. 多线程模型

一对一模型

- 1个用户级 <-> 1个内核级线程
- 缺点：需要频繁切换

多对一模型

- n个用户级 <-> 1个内核级线程
- 优点：
  - **开销小，效率高。**不需要频繁切换
- 缺点：
  - **一个线程阻塞，整个进程阻塞**
  - **一次只有一个线程访问内核**，多个线程不能同时在多个处理机上运行。
- 内核级线程才是处理机分配的单位。

多对多模型

- n个用户级 <-> m个内核级线程 (n>=m)
- 合成两个

注意

- 用户级线程是“代码逻辑”的载体
- 内核级线程是“运行机会”的载体

一个代码逻辑需要有运行机会才能运行。

第二章作业

```
get:
while(1):
{
P(f_out);
P(s_in);
将数从f取出放入s;
V(f_in);
V(s_out);
}
copy:
while(1):
{
P(s_out);
P(t_in);
将数从s取出放入t;
V(s_in);
V(t_out);
}
put:
while(1):
{
P(t_out);
P(g_in);
将数从s取出放入t;
V(t_in);
V(g_out);
}
```

```
/**
 * P、Q、R共享一个缓冲区，P、Q构成一对生产者-消费者，R即为生产者又为消费者，
 * 使用P、V操作实现其同步。
 */

typedef int semaphore;
semaphore mutex=1,empty=n,full=0;
//设置信号量mutex控制仓库进出，empty表示空仓库的个数，full表示满仓库的个数

void P()
{
    while(true)
    {
        wait(empty);//如果缓冲区已满，则阻塞
        wait(mutex);
        生产一个产品；
        signal(mutex);
        signal(full);//如果消费者被阻塞，则唤醒消费者
    }
}

void Q()
{
    while(true)
    {
        wait(full);//如果缓冲区为空，则阻塞
        wait(mutex);
        消费者取出一个产品
        signal(mutex);
        signal(empty);//如果生产者已经阻塞，则唤醒生产者
    }
}

void R()
{
    if(empty==n)//执行生产者的功能
    {
        wait(empty);
        wait(mutex);
        生产一个产品；
        signal(mutex);
        signal(full);
    }
    if(full==n)//执行消费者的功能
    {
        wait(full);
        wait(mutex);
        消费者取出一个产品
        signal(mutex);
        signal(empty);
    }
}
```