

操作系统原理

PRINCIPLES OF OPERATING SYSTEM

北京大学计算机科学技术系 陈向群

Department of computer science and Technology

Peking University

2015 春季

第5讲

同步互斥机制1

同步互斥机制

- ◎ 进程的并发执行
- ◎ 进程互斥
- ◎ 进程同步
- ◎ 信号量及PV操作
- ◎ 经典的IPC问题

并发环境下进程的特征

进程并发执行

问题的提出

并发是所有问题产生的基础

并发是操作系统设计的基础

从进程的特征出发

并发

- 进程的执行是中断性的
- 进程的相对执行速度不可预测

共享

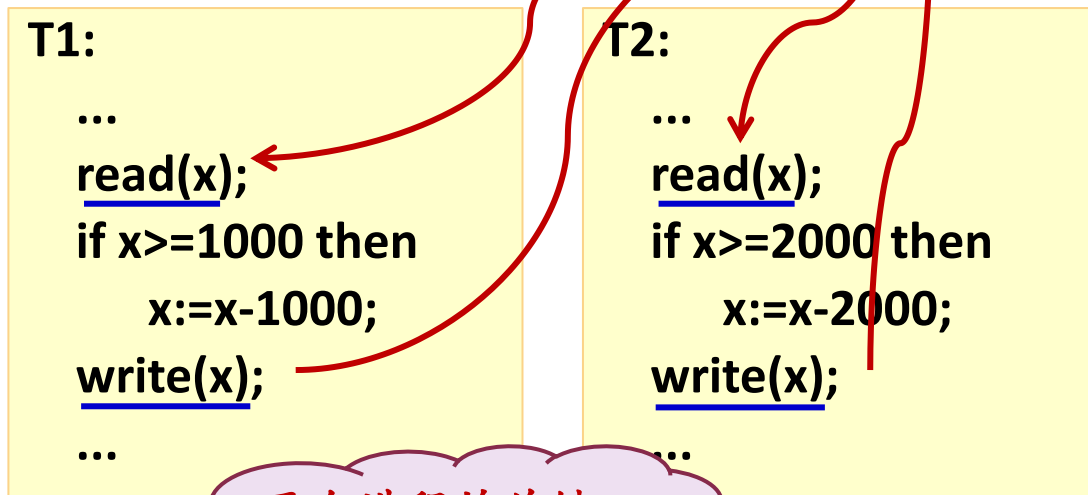
- 进程/线程之间的制约性

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

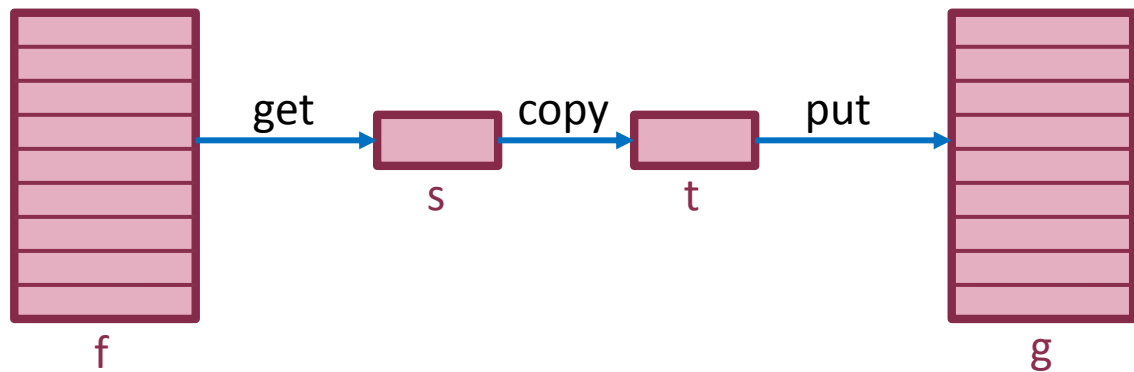
与时间有关的错误——例子1

某银行业务系统，某客户的账户有5000元，有两个ATM机T1和T2



两个进程的关键
活动出现交叉

与时间有关的错误——例子2



场景

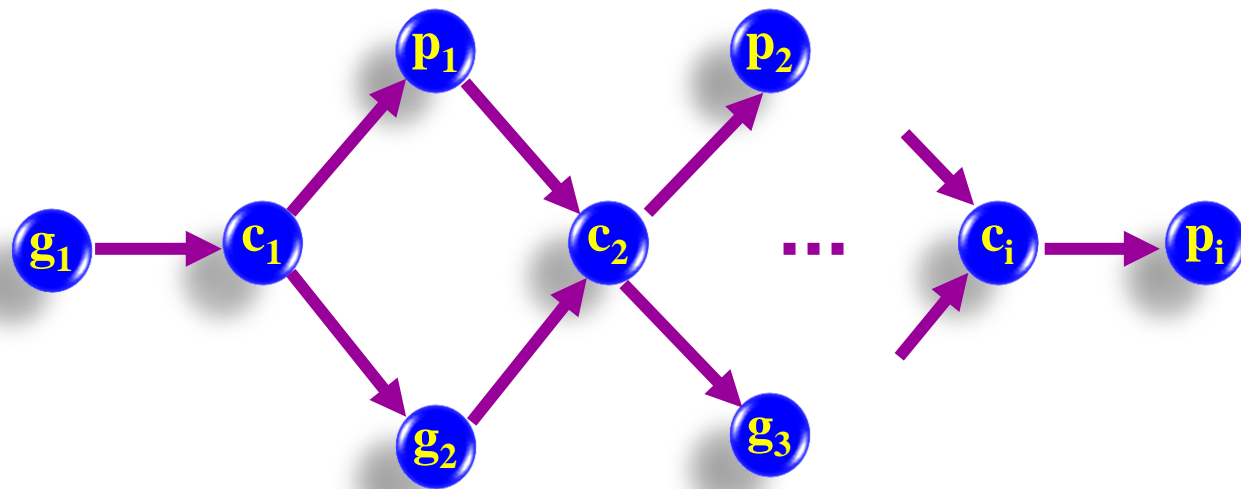
get、copy和put三个进程并发执行

并发执行过程分析

	f	s	t	g
当前状态	(3,4,...,m)	2	2	(1,2)
可能的执行 (假设g,c,p为get,copy,put的一次循环)				
g,c,p	(4,5,...,m)	3	3	(1,2,3) ✓
g,p,c	(4,5,...,m)	3	3	(1,2,2) ✗
c,g,p	(4,5,...,m)	3	2	(1,2,2) ✗
c,p,g	(4,5,...,m)	3	2	(1,2,2) ✗
p,c,g	(4,5,...,m)	3	2	(1,2,2) ✗
p,g,c	(4,5,...,m)	3	3	(1,2,2) ✗

设信息长度为m，有多少种可能性？

进程前趋图

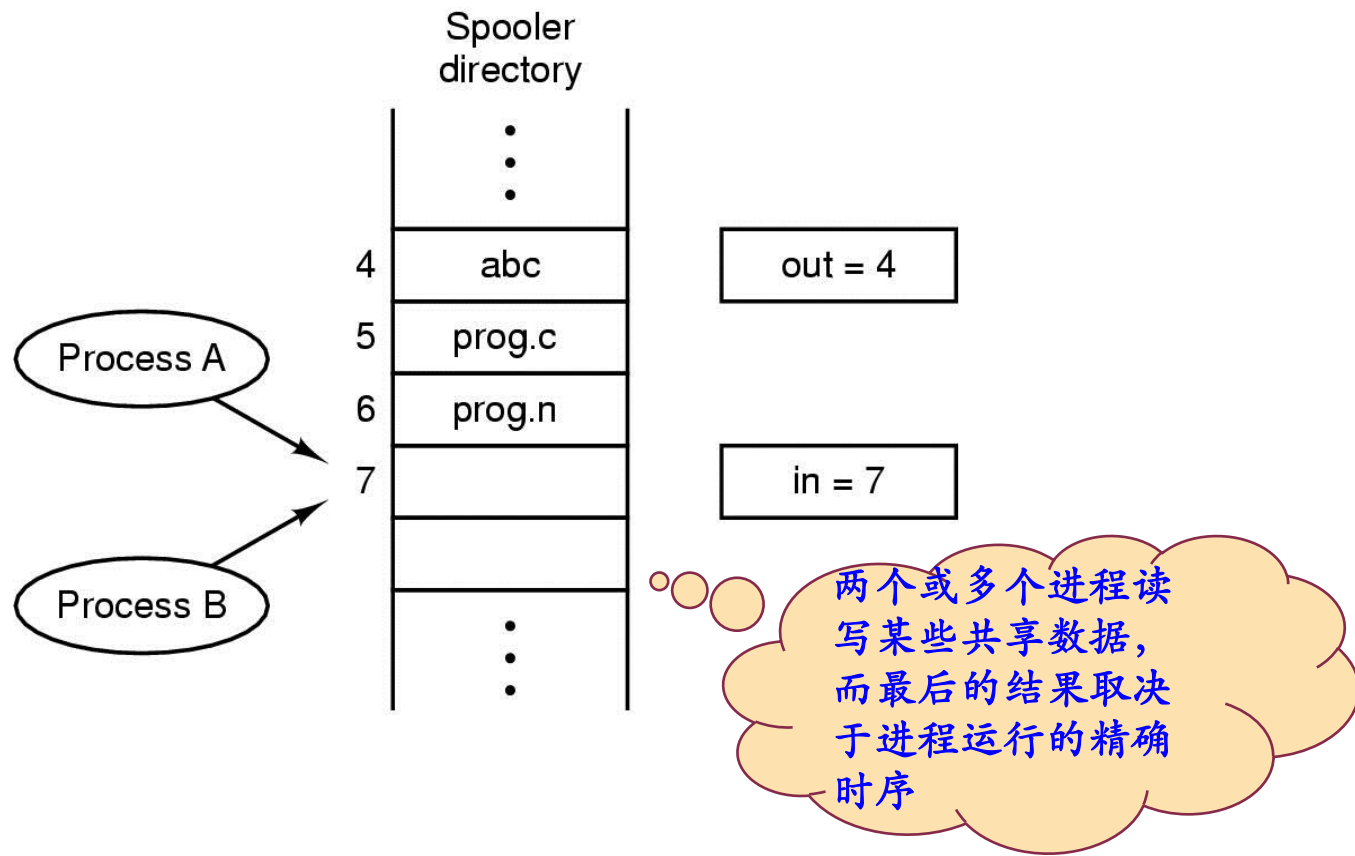


并发环境下进程间的制约关系

临界资源、临界区

进程互斥

竞争条件 (RACE CONDITION)

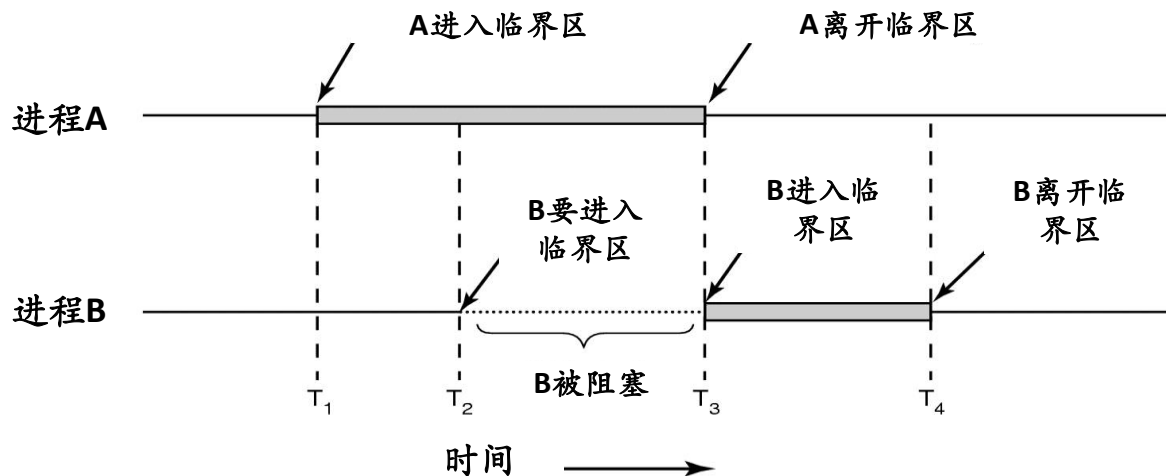


进程互斥(MUTUAL EXCLUSIVE)

- ◉ 由于各进程要求使用共享资源（变量、文件等），而这些资源需要排他性使用，各进程之间竞争使用这些资源——这一关系称为**进程互斥**
- ◉ **临界资源：critical resource**
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**
- ◉ **临界区(互斥区)：critical section(region)**
各个进程中对某个临界资源（共享变量）实施操作的程序片段

临界区(互斥区)的使用原则

- 没有进程在临界区时，想进入临界区的进程可进入
- 不允许两个进程同时处于其临界区中
- 临界区外运行的进程不得阻塞其他进程进入临界区
- 不得使进程无限期等待进入临界区



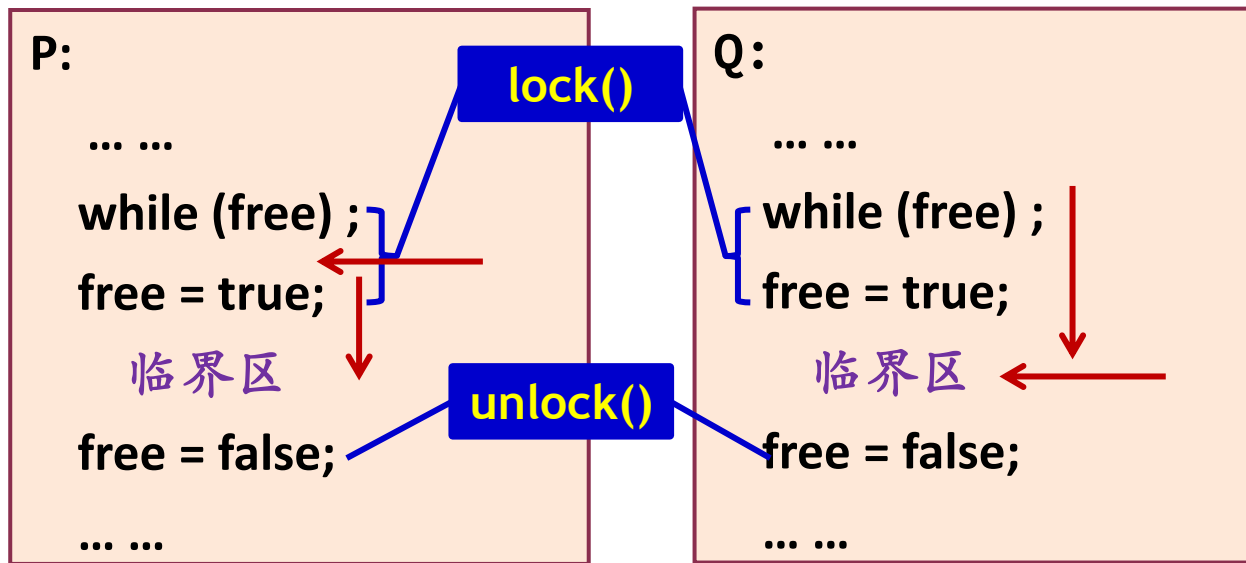
实现进程互斥的方案

- ◎ 软件方案
 - ◎ Dekker解法、Peterson解法
- ◎ 硬件方案
 - ◎ 屏蔽中断、TSL(XCHG)指令

Dekker解法、Peterson解法

进程互斥的软件解 决方案

软件解法1



free: 临界区空闲标志

true: 有进程在临界区

false: 无进程在临界区

初值: free为false

软件解法2

P:

... ..

while (not turn) ;

临界区

turn = false;

... ..

Q:

... ..

while (turn) ;

临界区

turn = true;

... ..

turn: 谁进临界区的标志

true: P进程进临界区

false: Q进程进临界区

初值任意

软件解法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: 初值为false

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

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

软件解法4——DEKKER算法

在解法3基础上
引入turn变量

P:

```
... ..  
pturn = true;  
while (qturn) {  
    if (turn == 2) {  
        pturn = false;  
        while (turn == 2);  
        pturn = true;
```

```
    }  
}
```

临界区

```
turn = 2;  
pturn = false;
```

```
... ..
```

循环

Q:

```
... ..  
qturn = true;  
while (pturn) {  
    if (turn == 1) {  
        qturn = false;  
        while (turn == 1);  
        qturn = true;
```

```
    }  
}
```

临界区

```
turn = 1;  
qturn = false;
```

```
... ..
```

第一个
1965

软件解法5——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)

中断屏蔽、TSL指令、XCHG指令

进程互斥的硬件解决 方案

硬件解法1—中断屏蔽方法

“开关中断”指令

执行“关中断”指令

临界区操作

执行“开中断”指令

- 简单，高效
- 代价高，限制CPU并发能力（临界区大小）
- 不适用于多处理器
- 适用于操作系统本身，不适用于用户进程

硬件解法2

——“测试并加锁” 指令

TSL指令：TEST AND SET LOCK

enter_region:

TSL REGISTER, LOCK

CMP REGISTER, #0

JNE enter_region

RET

| 复制锁到寄存器并将锁置1

| 判断寄存器内容是否为零?

| 若不是零, 跳转到enter_region

| 返回调用者, 进入了临界区

循环

leave_region:

MOVE LOCK, #0

RET

| 在锁中置0

| 返回调用者

提问：对多处理器系统
有效吗？为什么？

硬件解法3——“交换”指令

XCHG指令: EXCHANGE

enter_region:

```
MOVE REGISTER,#1  
XCHG REGISTER,LOCK  
CMP REGISTER,#0  
JNE enter_region  
RET
```

| 给寄存器中置1
| 交换寄存器与锁变量的内容
| 判断寄存器内容是否是零?
| 若不是零, 跳转到enter_region
| 返回调用者, 进入了临界区

循环

leave_region:

```
MOVE LOCK,#0  
RET
```

| 在锁中置0
| 返回调用者

小结

- ◎ 软件方法

 - 编程技巧

- ◎ 硬件方法

- ◎ 忙等待(**busy waiting**)

进程在得到临界区访问权之前，持续测试而不做其他事情

自旋锁 **Spin lock** (多处理器 v)

- ◎ 优先级反转（倒置）

协作关系

进程同步

进程的同步

进程同步: synchronization

指系统中多个进程中发生的事件存在某种时序关系，需要相互合作，共同完成一项任务

具体地说，一个进程运行到某一点时，要求另一伙伴进程为它提供消息，在未获得消息之前，该进程进入阻塞态，获得消息后被唤醒进入就绪态

生产者/消费者问题

又称为
有界缓冲区问题

问题描述:

- 一个或多个生产者生产某种类型的数据放置在缓冲区中
- 有消费者从缓冲区中取数据, 每次取一项
- 只能有一个生产者或消费者对缓冲区进行操作

生产者
进程



消费者
进程



缓冲区

要解决的问题:

- ⊙ 当缓冲区已满时, 生产者不会继续向其中添加数据;
- ⊙ 当缓冲区为空时, 消费者不会从中移走数据

⊙ 避免忙等待

睡眠与唤醒操作(原语)

生产者/消费者问题

```
#define N 100
```

```
int count=0;
```

```
void producer(void)
```

```
{ int item;
```

```
while(TRUE) {
```

```
    item=produce_item();
```

```
    if(count==N) sleep();
```

```
    insert_item(item);
```

```
    count=count+1;
```

```
    if(count==1)
```

```
        wakeup(consumer);
```

```
}
```

```
}
```

检查
count
的值

```
void consumer(void)
```

```
{
```

```
    int item;
```

```
    while(TRUE) {
```

```
        if(count==0) sleep();
```

```
        item=remove_item();
```

```
        count=count-1;
```

```
        if(count==N-1)
```

```
            wakeup(producer);
```

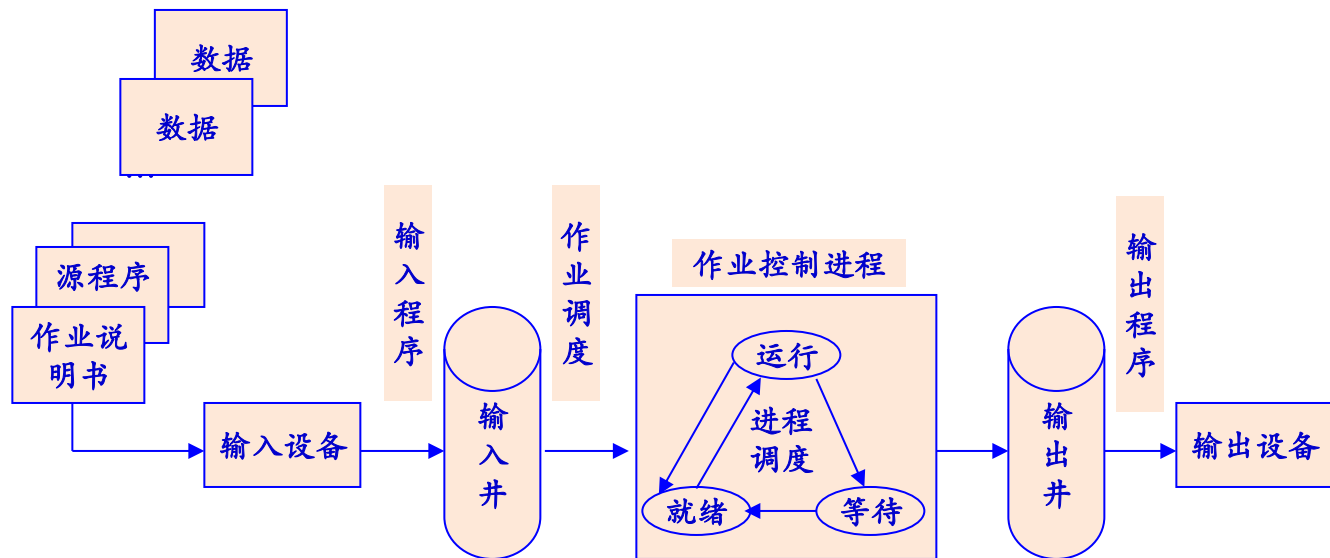
```
        consume_item(item);
```

```
    }
```

```
}
```

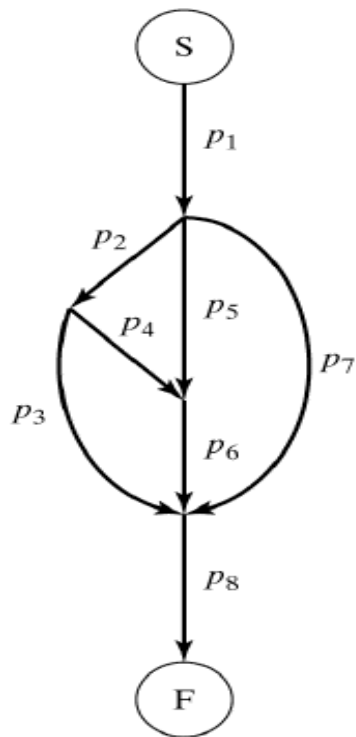
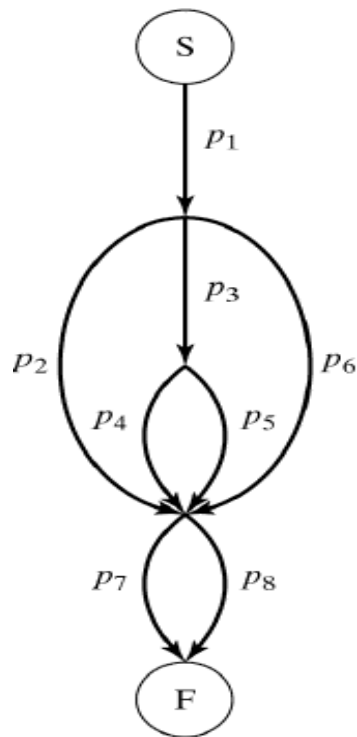
一种场景：消费者
判断count=0后进入
睡眠前被切换

其他同步例子1



SPOOLing 系统

其他同步例子2



一种经典的进程同步机制

信号量及P、V操作

信号量及PV操作

- ◎ 一个特殊变量
- ◎ 用于进程间传递信息的一个整数值
- ◎ 定义如下：

```
struc semaphore
{
    int count;
    queueType queue;
}
```

- ◎ 信号量说明：semaphore s;
- ◎ 对信号量可以实施的操作：初始化、P和V（P、V分别是荷兰语的test(proberen)和increment(verhogen)）

是一种卓有成效的进程同步机制

1965年，由荷兰学者Dijkstra提出

P、V操作定义

P(s)

```
{  
    s.count --;  
    if (s.count < 0)  
    {  
        该进程状态置为阻塞状态;  
        将该进程插入相应的等待队  
        列s.queue末尾;  
        重新调度;  
    }  
}
```

down, semWait

V(s)

```
{  
    s.count ++;  
    if (s.count <= 0)  
    {  
        唤醒相应等待队列s.queue中  
        等待的一个进程;  
        改变其状态为就绪态, 并将其  
        插入就绪队列;  
    }  
}
```

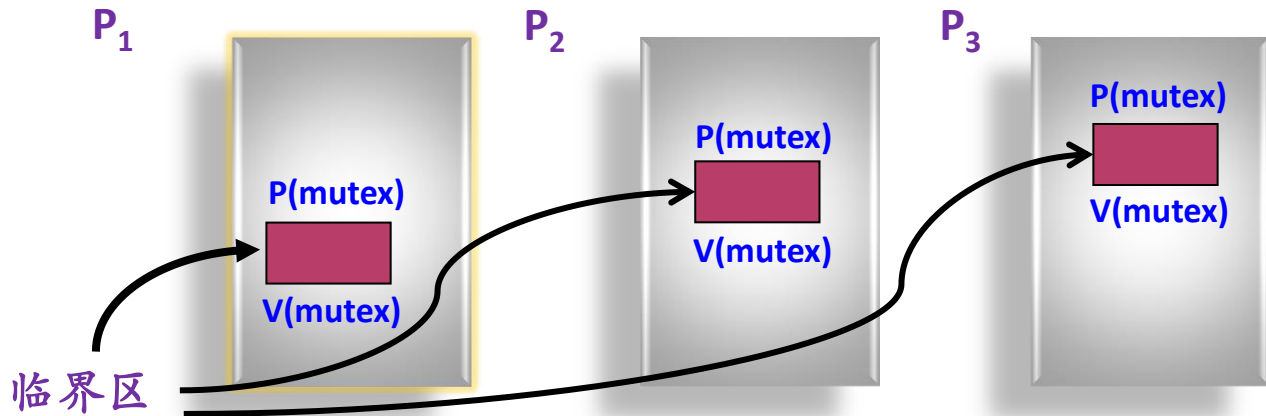
up, semSignal

有关说明

- P、V操作作为原语操作(primitive or atomic action)
- 在信号量上定义了三个操作
初始化(非负数)、P操作、V操作
- 最初提出的是二元信号量（解决互斥）
之后，推广到一般信号量（多值）或计数信号量
（解决同步）

用PV操作解决进程间互斥问题

- 分析并发进程的关键活动，划定临界区
- 设置信号量 **mutex**，初值为1
- 在临界区前实施 **P(mutex)**
- 在临界区之后实施 **V(mutex)**



生产者消费者问题

用信号量解决生产者/消费者问题

```
void producer(void)
{
    int item;
    while(TRUE) {
        item=produce_item();
        P(&empty);
        P(&mutex);
        insert_item(item);
        V(&mutex);
        V(&full);
    }
}
```

```
void consumer(void)
{
    int item;
    while(TRUE) {
        P(&full);
        P(&mutex);
        item=remove_item();
        V(&mutex);
        V(&empty);
        consume_item(item);
    }
}
```

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

```
/* 缓冲区个数 */
/* 信号量是一种特殊的整型数据 */
/* 互斥信号量：控制对临界区的访问 */
/* 空缓冲区个数 */
/* 满缓冲区个数 */
```

讨论一下

```
void producer(void)
```

```
{ int item;
```

```
while(TRUE) {
```

```
    item=produce_item();
```

```
    P(&empty);
```

```
    P(&mutex);
```

```
    insert_item(item);
```

```
    V(&mutex);
```

```
    V(&full);
```

```
}
```

```
}
```

```
void consumer(void)
```

```
{ int item;
```

```
while(TRUE) {
```

```
    P(&full);
```

```
    P(&mutex);
```

```
    item=remove_item();
```

```
    V(&mutex);
```

```
    V(&empty);
```

```
    consume_item(item);
```

```
}
```

```
}
```

思考：

若颠倒两个P
操作的顺序？

思考：

若颠倒两个V
操作的顺序？

顺序与
位置

读者写者问题

用信号量解决读者/写者问题

问题描述：

多个进程共享一个数据区，这些进程分为两组：

读者进程：只读数据区中的数据

写者进程：只往数据区写数据

要求满足条件：

- ✓ 允许多个读者同时执行读操作
- ✓ 不允许多个写者同时操作
- ✓ 不允许读者、写者同时操作

第一类读者写者问题：读者优先

如果读者执行：

- 无其他读者、写者，该读者可以读
- 若已有写者等，但有其他读者正在读，则该读者也可以读
- 若有写者正在写，该读者必须等

如果写者执行：

- 无其他读者、写者，该写者可以写
- 若有读者正在读，该写者等待
- 若有其他写者正在写，该写者等待

第一类读者-写者问题的解法

```
void reader(void)
```

```
{
```

```
    while (TRUE) {
```

```
        .....
```

```
        P(w);
```

```
        读操作
```

```
        V(w);
```

```
        .....
```

```
    }
```

```
}
```

不需要每个
读者都做

```
void writer(void)
```

```
{
```

```
    while (TRUE) {
```

```
        .....
```

```
        P(w);
```

```
        写操作
```

```
        V(w);
```

```
        .....
```

```
    }
```

```
}
```

第一类读者-写者问题的解法

```
void reader(void)
```

```
{
```

```
while (TRUE) {
```

```
P(mutex);
```

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

```
if (rc == 1) P(w);
```

```
V(mutex);
```

读操作

```
P(mutex);
```

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

```
if (rc == 0) V(w);
```

```
V(mutex);
```

其他操作

```
}
```

第一个读者

最后一个读者

临界区

```
void writer(void)
```

```
{
```

```
while (TRUE) {
```

```
.....
```

```
P(w);
```

写操作

```
V(w);
```

```
}
```

```
}
```

LINUX提供的读写锁

◉ 应用场景

如果每个执行实体对临界区的访问或者是读或者是写共享变量，但是它们都不会既读又写时，读写锁是最好的选择

◉ 实例：Linux的IPX路由代码中使用了读-写锁，用 `ipx_routes_lock` 的读-写锁保护IPX路由表的并发访问

要通过查找路由表实现包转发的程序需要请求读锁；
需要添加和删除路由表中入口的程序必须获取写锁
(由于通过读路由表的情况比更新路由表的情况多得多，使用读-写锁提高了性能)

本讲重点

- ◎ 基本概念
 - 竞争条件、临界区
 - 进程同步、进程互斥
 - 自旋锁（忙等待）
- ◎ 信号量、PV操作
- ◎ 经典问题模型及解决方案
 - 生产者-消费者问题、读者-写者问题

本周要求

- 重点阅读教材

第2章相关内容： 2.3.1~2.3.5， 2.5.2

- 重点概念

竞争条件 与时间有关的错误 忙等待
临界资源 临界区 进程互斥 进程同步
信号量 P、V操作 锁 自旋锁
生产者消费者问题 读者写者问题

THANKS

The End