

操作系统A

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

2020 Autumn

本章要求掌握的概念

临界区

进程互斥

进程同步

信号量

P、V操作

管程

条件变量

wait/signal

锁

Hoare管程

Pthreads

.....

生产者消费者
问题

读者写者问题

...2..

从进程的特征出发

并发是所有问题的基础
并发是操作系统设计的基础



并发

- 进程的执行过程不是连续的，是间断性的
- 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

并发环境下进程的特征

进程并发执行

顺序环境

► 顺序环境

在计算机系统中只有一个程序在运行，这个程序独占系统中所有资源，其执行不受外界影响

特征：程序执行的顺序性

► 程序执行的封闭性

独占资源，执行过程中不受外界影响

► 程序执行结果的确定性

即：程序结果的可再现性

程序运行结果与程序执行速度无关，只要初始状态相同，结果应相同

► 调度顺序不重要

并发环境

- 执行过程是不确定和不可重现的
- 程序错误可能是偶然发生的

特征:

- (1) 程序执行结果的不可再现性
并发程序执行结果与其执行的相对速度有关，是不确定的
- (2) 在并发环境下程序的执行是间断性的
执行——停——执行
- (3) 资源共享
系统中资源被多个进程使用
- (4) 独立性和制约性
独立的相对速度、起始时间
- (5) 程序和计算不再一一对应

与时间有关的错误——例子1

某银行业务系统，某客户的账户中有5000元，有两个ATM机T1和T2

T1:

...

read(x);

if $x \geq 1000$ then

$x := x - 1000$;

write(x);

...

T2:

...

read(x);

if $x \geq 2000$ then

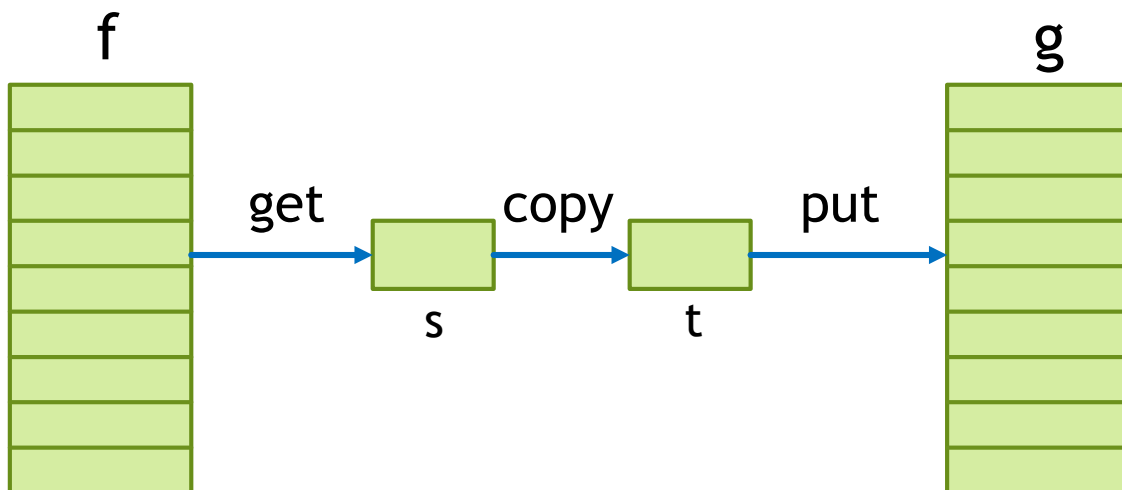
$x := x - 2000$;

write(x);

...

两个进程的关键
活动出现交叉

与时间有关的错误——例子2

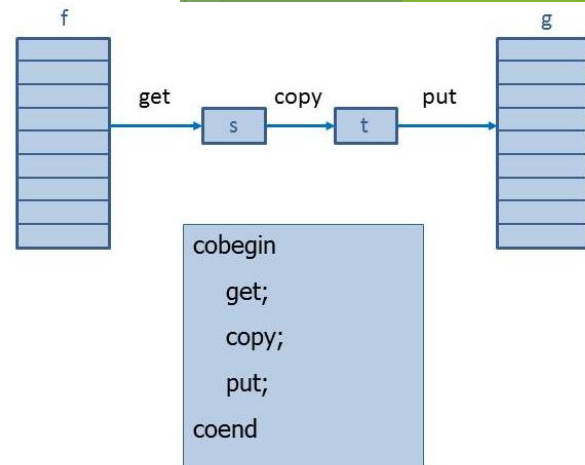


场景

```
cobegin  
  get;  
  copy;  
  put;  
coend
```

get、copy和put三个进程并发执行

并发执行过程分析



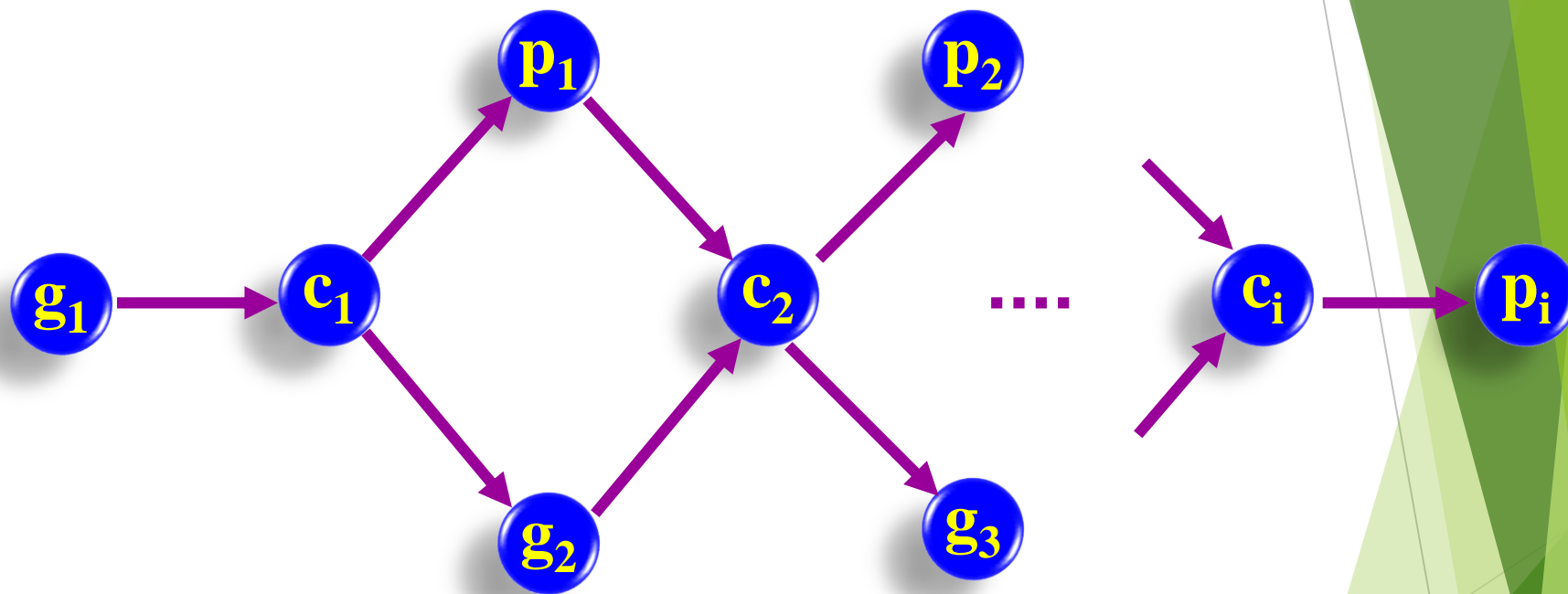
	f	s	t	g
当前状态	(3,4,...,m)	2	2	(1,2)

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？

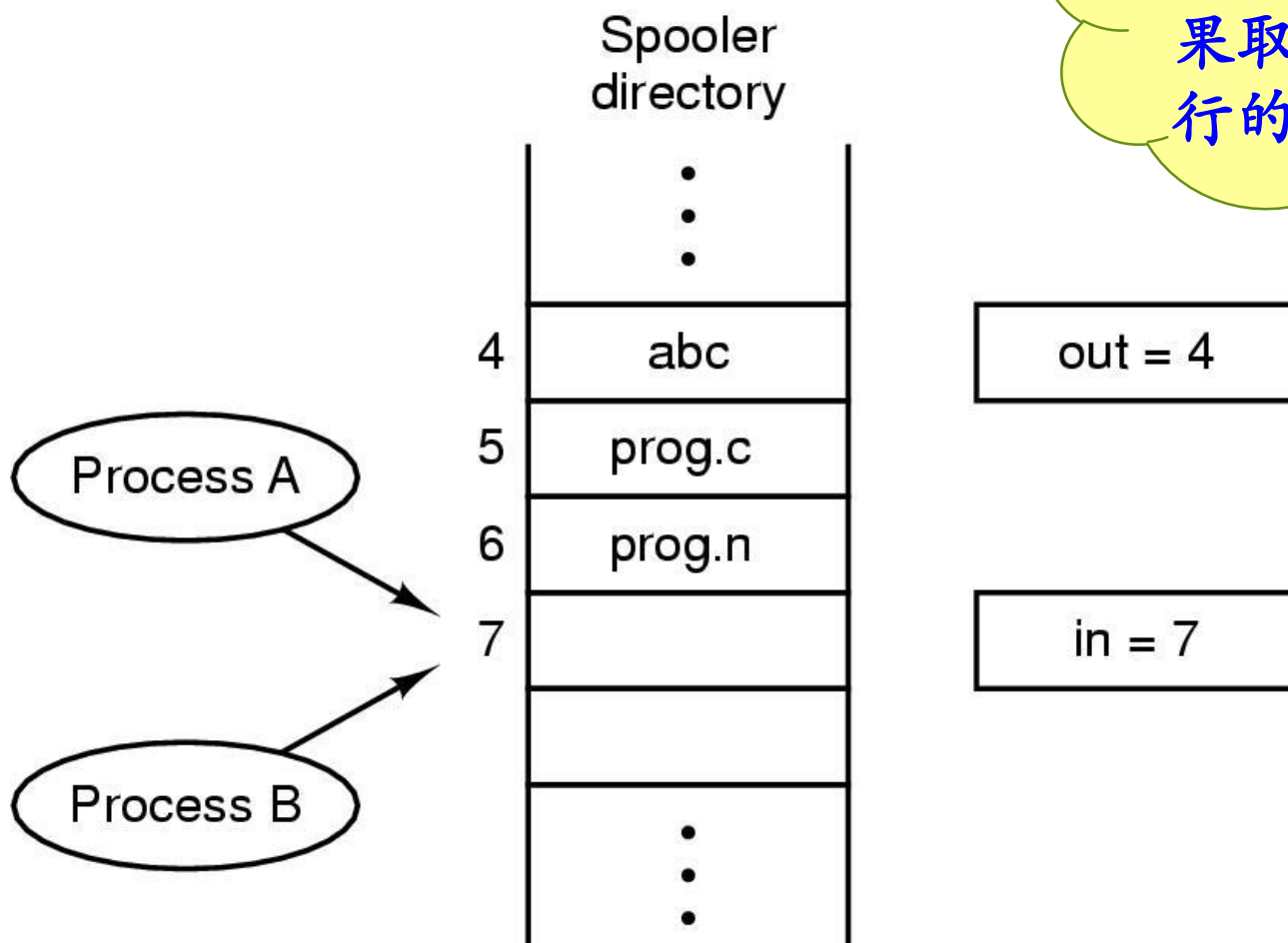
进程前趋图



并发环境下程序间的制约关系

竞争条件 (race condition)

两个或多个进程
读写某些共享数
据，而最后的结
果取决于进程运
行的精确时序

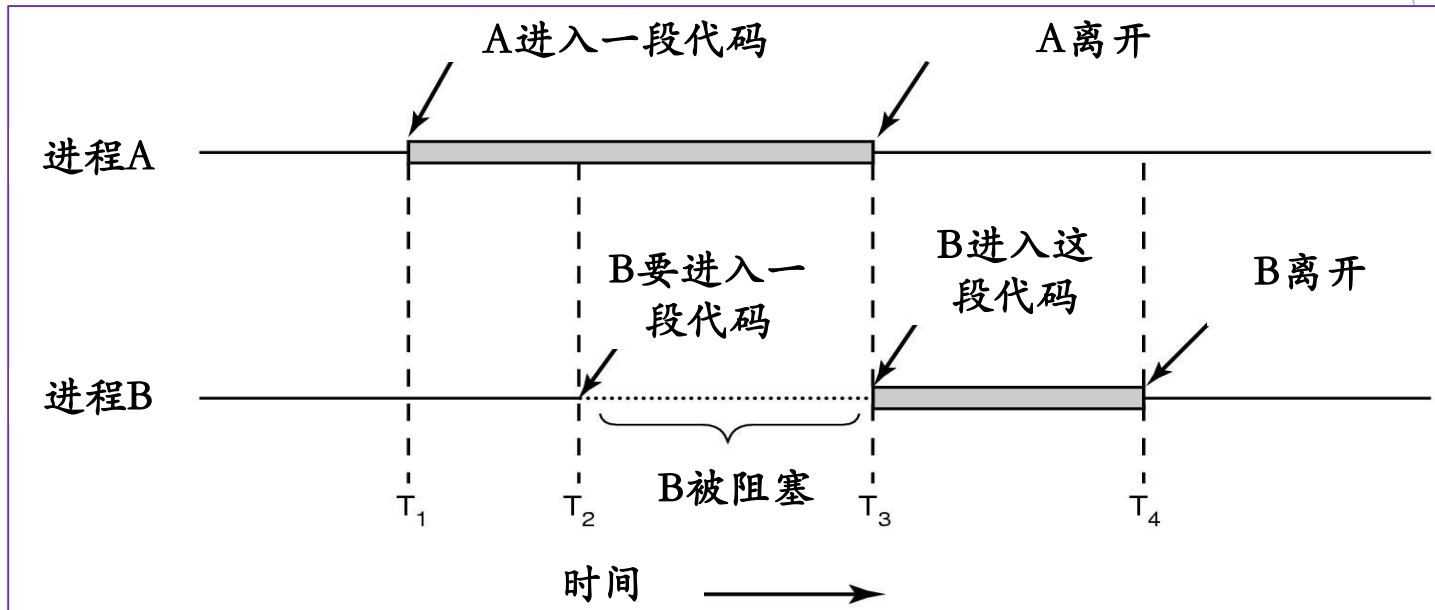


进程互斥1

- ▶ 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**



进程互斥2

► 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

- ▶ **有空让进**：当没有进程在临界区时，任何有权使用互斥区的进程可进入临界区
- ▶ **无空等待**：不允许两个以上的进程同时进入临界区
- ▶ **有限等待**：任何进入临界区的要求应在有限时间内得到满足
- ▶ **多中择一**：当没有进程在临界区，而同时有多个进程要求进入临界区时，只能让其中之一进入，其他进程必须等待
- ▶ **让权等待**：处于等待状态的进程应放弃占用CPU，以使其他进程有机会得到CPU的使用权

前提：任何进程无权停止其它进程的运行；进程之间相对运行速度无硬性规定

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

实现进程互斥的方案

- ▶ 软件方案

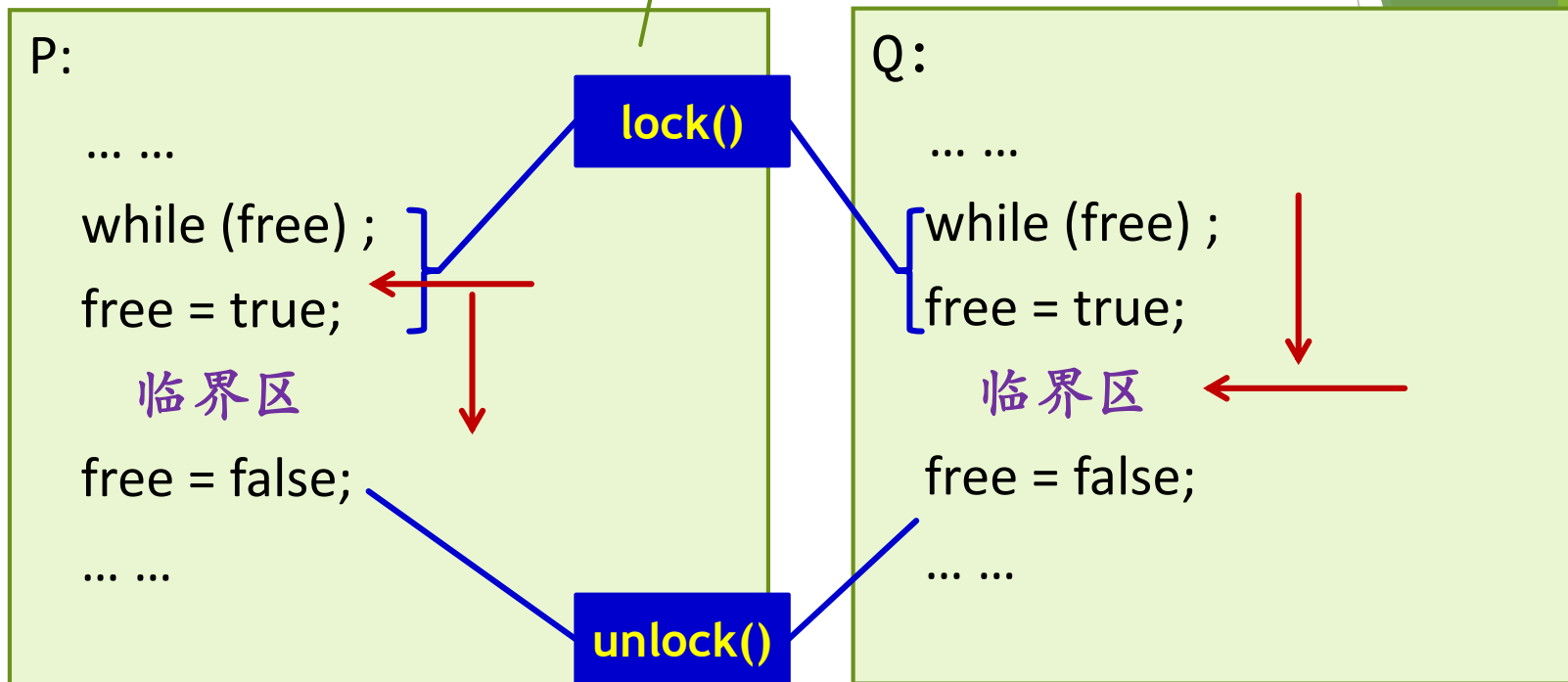
- ▶ Dekker解法、Peterson解法

- ▶ 硬件方案

- ▶ 屏蔽中断、TSL(XCHG)指令

软件解法1

lock()必须满足什么性质?



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: $pturn \wedge \text{not } qturn$

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

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;

... ..

Q:

... ..

qturn = true;

while (pturn) {

if (turn == 1) {

qturn = false;

while (turn == 1);

qturn = true;

}}

临界区

turn = 1;

qturn = false;

... ..

循环

1965年提出
第一个用软件正
确解决互斥问题

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)

硬件解法1——中断屏蔽方法

“开关中断”指令

```
while (true) {  
    /* disable interrupts */;  
    /* critical section */;  
    /* enable interrupts */;  
    /* remainder */;  
}
```

- 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
```

循环

| 复制锁到寄存器并将锁置1
| 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

leave_region:

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

| 在锁中置0
| 返回调用者

思考题：对多处理器
系统有效吗？为什么？

思考题：
XCHG指令

小结

- ▶ 软件方法

 - ▶ 需要编程技巧

- ▶ 硬件方法

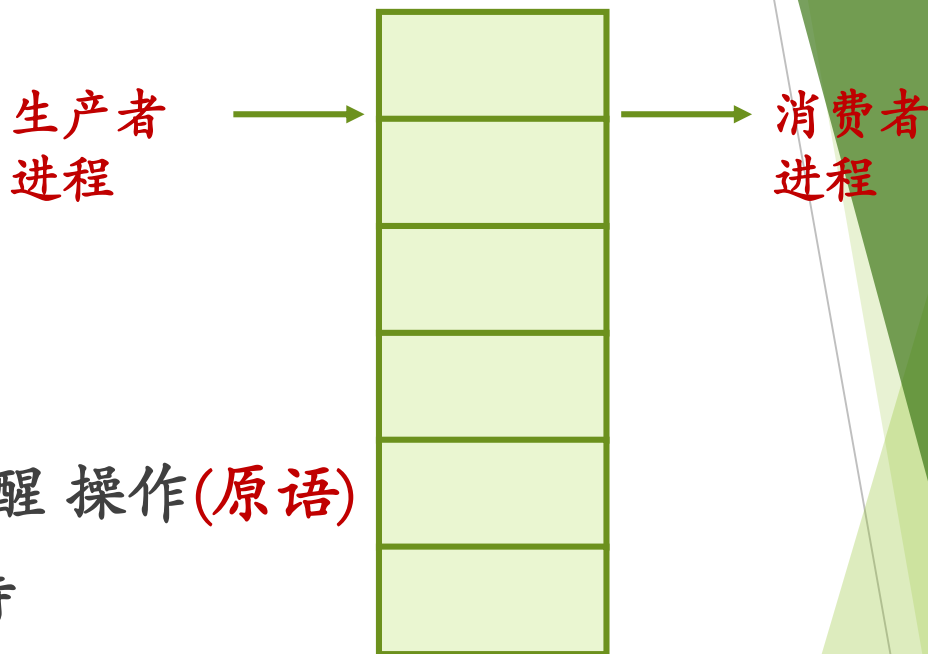
- ▶ 忙等待(busy waiting)

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）

生产者/消费者(有界缓冲区)问题



解决方案

- ▶ 提供：睡眠与唤醒操作(原语)
- ▶ 要求：避免忙等待

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

生产者/消费者问题

一种场景：消费者
判断count=0后进入
睡眠前被切换

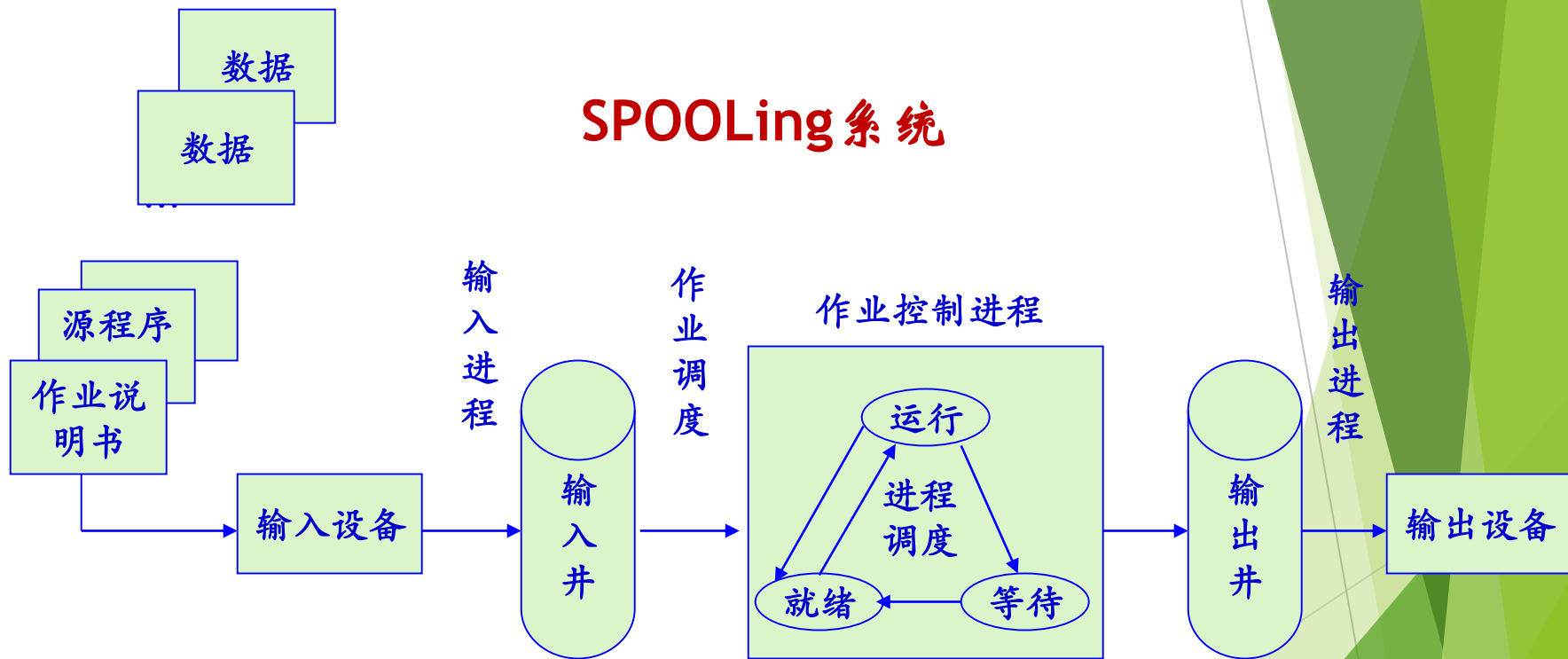
```
#define N 100
int count=0;

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

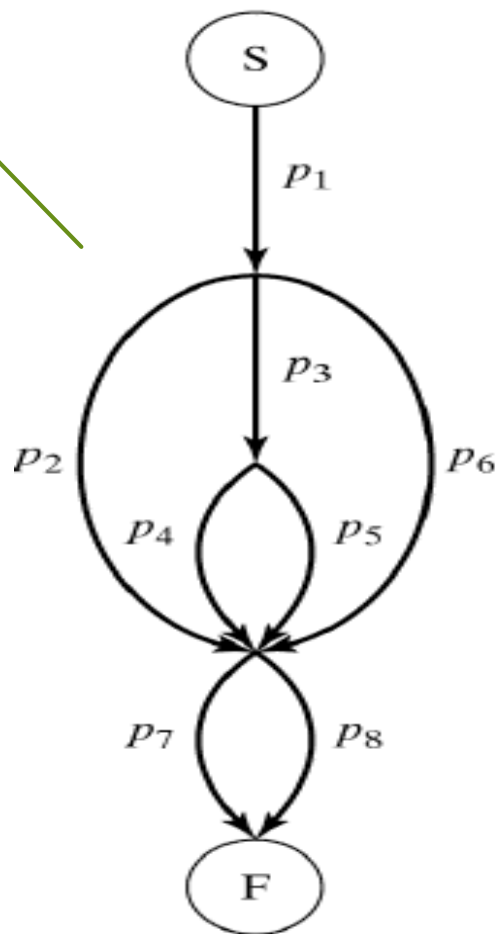
```
void consumer(void)
{
  int item;
  while(TRUE) {
    if(count==0) sleep();
    item=remove_item();
    count=count-1;
    if(count==N-1)
      wakeup(producer);
    consume_item(item);
  }
}
```

同步例子1

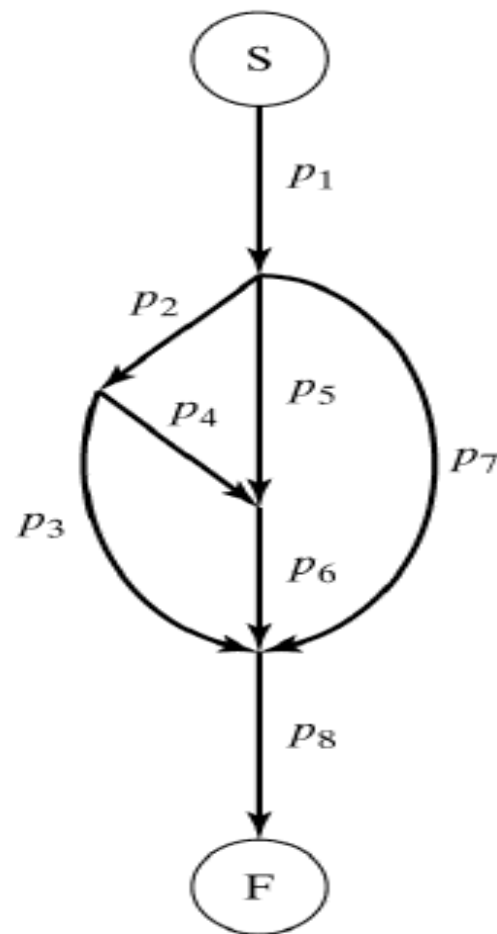


同步例子2

进程间只存在同步关系



Series/parallel



General precedence

典型的同步机制

- ▶ 信号量及P、V操作
- ▶ 管程
- ▶ 锁
- ▶ 条件变量

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

一种卓有成效的进程同步机制

由荷兰学者Dijkstra提出，1965年

- ▶ 一个特殊变量
- ▶ 用于进程间传递信号的一个整数值
- ▶ 定义如下：

```
struct 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)）

P、V操作定义

与一些参考书上或
ICS课上定义不同，
但效果是否一样？

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;  
    }  
}
```

³²up, semSignal

有关说明

➤ P、V操作为原语操作

原语(primitive or atomic action)

完成某种特定功能的一段程序，具有不可分割性或不可中断性

原语的执行必须是连续的，在执行过程中不允许被中断
可以通过屏蔽中断、测试与设置指令等来实现

➤ 在信号量上定义了三个操作

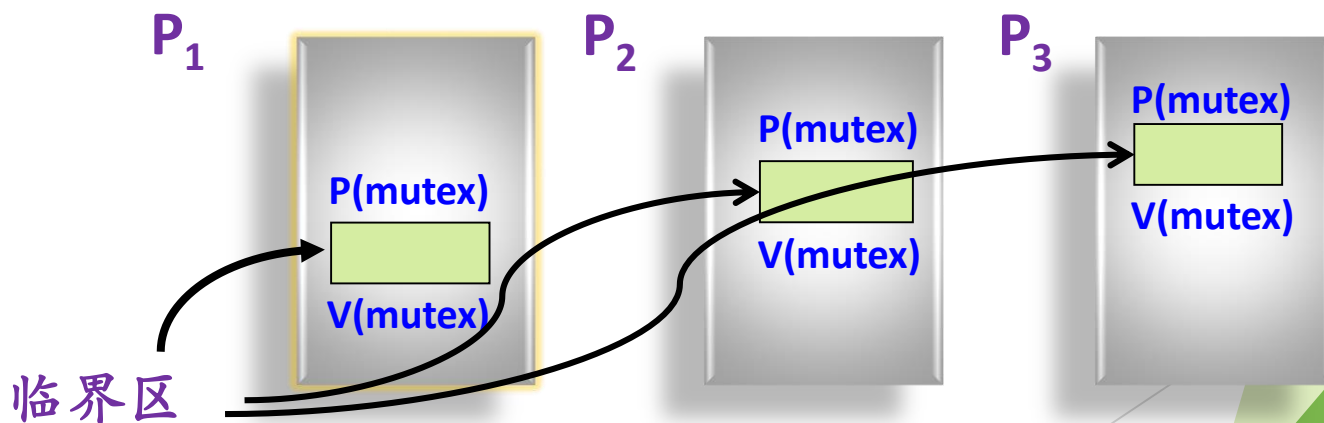
初始化(非负数)、P操作、V操作

➤ 最初提出的是二元信号量（互斥）

之后推广到一般信号量（多值）或计数信号量（同步）

用PV操作解决进程间互斥问题

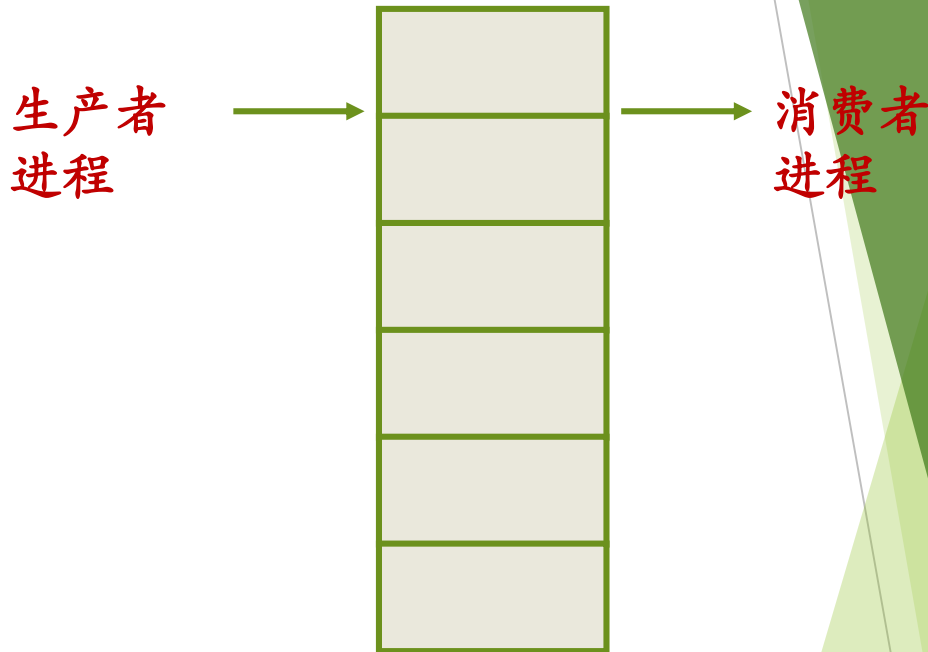
- ▶ 分析并发进程的关键活动，划定临界区
- ▶ 设置信号量 **mutex**，初值为1
- ▶ 在临界区前实施 **P(mutex)**
- ▶ 在临界区之后实施 **V(mutex)**



生产者/消费者(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



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

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 /* 缓冲区个数 */
typedef int semaphore; /* 信号量是一种特殊的整型数据 */
semaphore mutex = 1; /* 互斥信号量：控制对临界区的访问 */
semaphore empty = N; /* 空缓冲区个数 */
semaphore full = 0; /* 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);
    }
}
```

位置改变

若颠倒两个P操作的顺序?

若颠倒两个V操作的顺序?

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

位置改变

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

给出一种
应用场景

第一类：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);
```

```
    }
```

```
}
```

语言机制、条件变量/wait/signal

管程

为什么引入管程？

问题

- 信号量机制的不足：程序编写困难、效率低

解决

- Brinch Hansen(1973)
- Hoare (1974)

方案

- 在程序设计语言中引入管程成分
- 一种高级同步机制

管程的定义

- ▶ 是一个特殊的模块
- ▶ 有一个名字
- ▶ 由关于共享资源的数据结构及在其上操作的一组过程组成

□ 进程与管程

进程只能通过调用管程中的过程来间接访问管程中的数据结构

```
monitor example
integer i;
condition c;

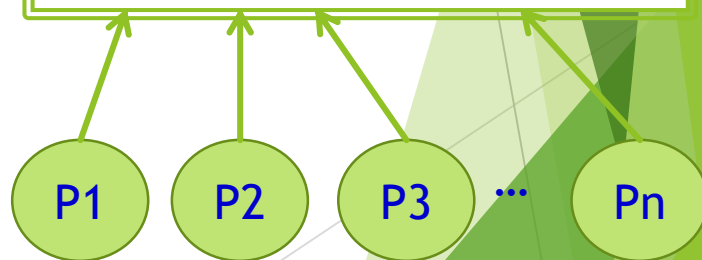
procedure producer();
.
.
end;

procedure consumer();
.
.
end;

end monitor;
```

变量

过程



管程要保证什么？

► 作为一种同步机制，管程要解决两个问题 

► 互斥

管程是互斥进入的

——为了保证管程中数据结构的数据完整性

注意： 管程的互斥性是由编译器负责、保证的

► 同步

管程中设置条件变量及等待/唤醒操作以解决同步问题

可以让一个进程或线程等待在条件变量上等待（此时，应先释放管程的使用权），也可以通过发送信号将等待在条件变量上的进程或线程唤醒

使用管程时遇到的问题

是否会出现这样一种情况，有多个进程同时在管程中？

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

如何解决？

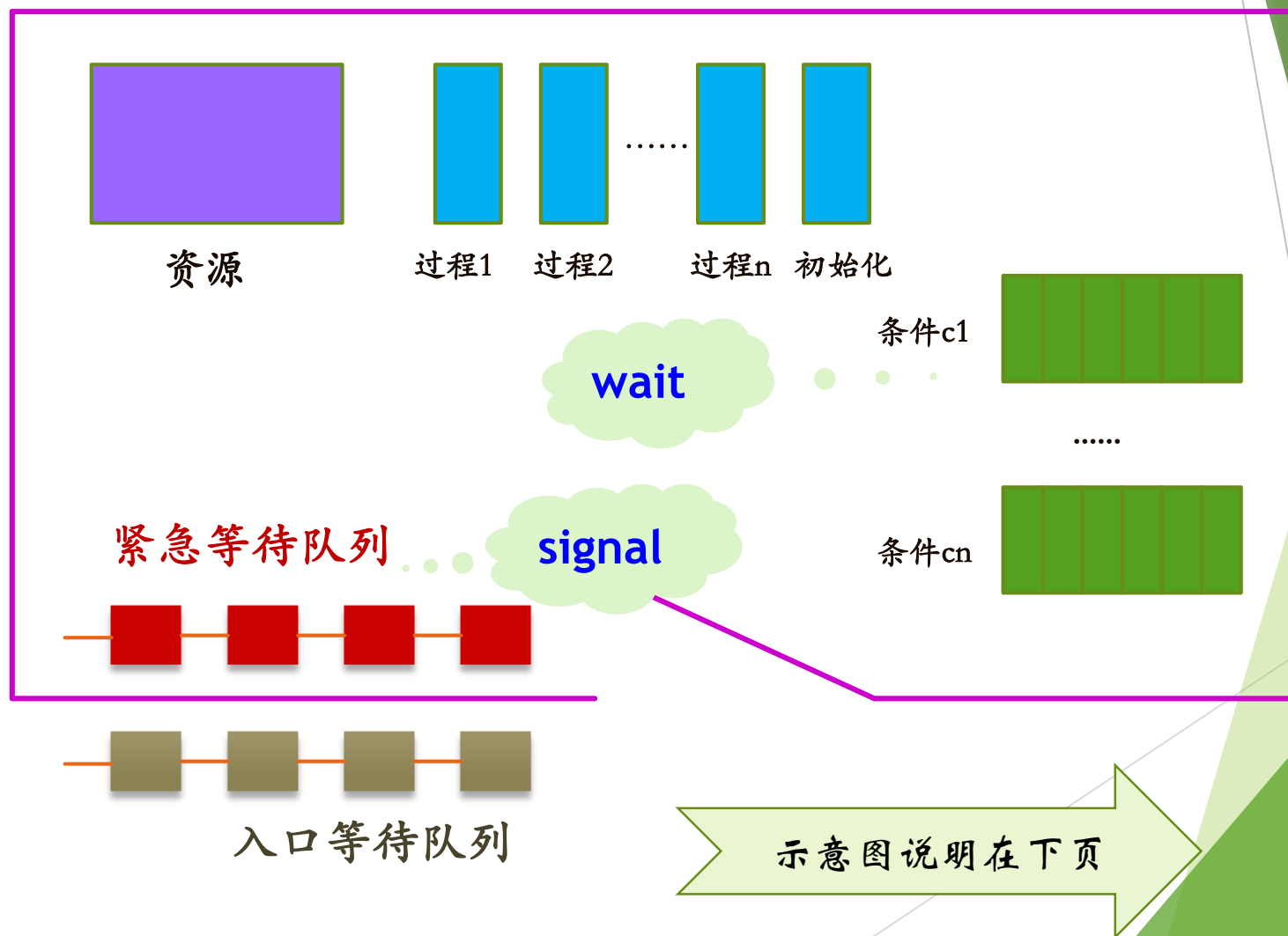
三种处理方法：

□ P等待Q执行 (Hoare) v

□ Q等待P继续执行 MESA

□ 规定唤醒为管程中最后一个可执行的操作 (Hansen, 并发pascal)

Hoare管程示意图



Hoare 管程说明

- 管程是互斥进入的，当一个进程试图进入一个已被占用的管程时，应当在管程的入口处等待
 - 为此，管程的入口处设置一个进程等待队列，称作入口等待队列
- 如果进程P唤醒进程Q，则P等待Q执行；如果进程Q执行中又唤醒进程R，则Q等待R执行；……，如此，在管程内部可能会出现多个等待进程
 - 管程内需要设置一个进程等待队列，称为紧急等待队列，其优先级高于入口等待队列的优先级

Hoare管程——条件变量的实现

- 条件变量——在管程内部说明和使用的一种特殊类型的变量
- `var c:condition;`
- 对于条件型变量，可以执行wait和signal操作

wait(c):

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；执行此操作的进程进入c链尾部

signal(c):

如果c链为空，则相当于空操作，执行此操作的进程继续执行；否则唤醒第一个等待者，执行此操作的进程进入紧急等待队列的尾部

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;
```

管程的实现

管程实现的两个主要途径：

- * 直接构造 → 效率高

- * 间接构造

→ 用某种已经实现的同步机制去构造

例如：用信号量
及P、V操作构造
管程

直接构造管程

The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.

管程：抽象数据类型

有一个明确定义的操作集合，通过它且只有通过它才能操纵该数据类型的实例

实现管程结构必须保证两点：

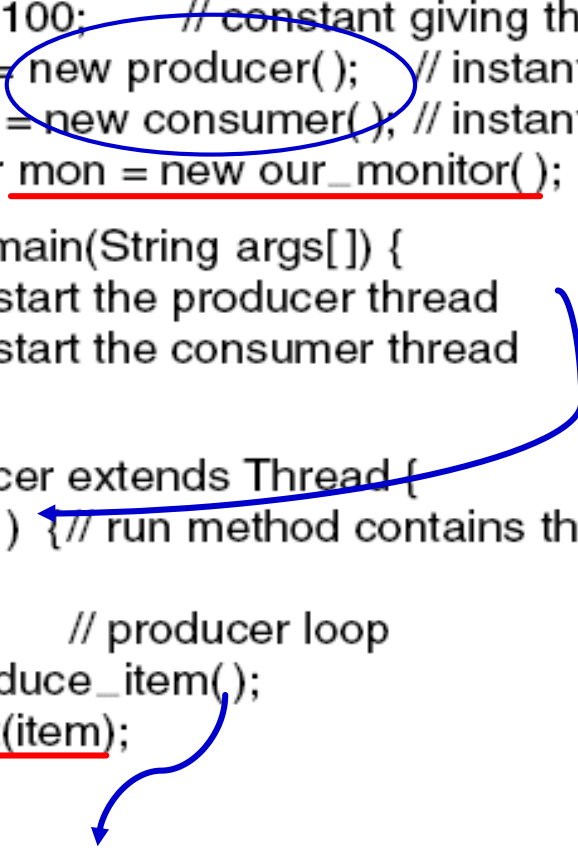
- (1) 只能通过管程的某个过程才能访问资源；
- (2) 过程是互斥的，某个时刻只能有一个进程或线程驻留在给定的管程中

条件变量：为提供进程与其他进程通信或同步而引入

► **wait/signal**

Java中的管程(1/2)

```
public class ProducerConsumer {  
    static final int N = 100; // constant giving the buffer size  
    static producer p = new producer(); // instantiate a new producer thread  
    static consumer c = new consumer(); // instantiate a new consumer thread  
    static our_monitor mon = new our_monitor(); // instantiate a new monitor  
  
    public static void main(String args[]) {  
        p.start(); // start the producer thread  
        c.start(); // start the consumer thread  
    }  
  
    static class producer extends Thread {  
        public void run() { // run method contains the thread code  
            int item;  
            while (true) { // producer loop  
                item = produce_item();  
                mon.insert(item);  
            }  
        }  
        private int produce_item() { ... } // actually produce  
    }  
}
```



Java中的管程(2/2)

```
static class consumer extends Thread {
    public void run() {run method contains the thread code
        int item;
        while (true) {    // consumer loop
            item = mon.remove();
            consume_item (item);
        }
    }
    private void consume_item(int item) { ... } // actually consume
}

static class our_monitor { // this is a monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep();    // if the buffer is full, go to sleep
        buffer [hi] = val; // insert an item into the buffer
        hi = (hi + 1) % N;    // slot to place next item in
        count = count + 1;    // one more item in the buffer now
        if (count == 1) notify();    // if consumer was sleeping, wake it up
    }
}
```

间接构造管程

```
TYPE one_instance=RECORD
    mutex: semaphore; (初值1)
    urgent: semaphore; (初值0)
    urgent_count: integer; (初值0)
END;
```

```
TYPE monitor_one =MODULE;
define enter,leave,wait,signal;
```

mutex (入口互斥队列)
urgent (紧急等待队列)
urgent_count (紧急等待队列计数)

首先，定义一个
一个基础管程

```
PROCEDURE enter(VAR
    instance:one_instance);
BEGIN
    P(instance.mutex)
END;
```

```
PROCEDURE leave(VAR
    instance:one_instance);
BEGIN
    IF instance.urgent_count >0
    THEN
        BEGIN instance.urgent_count--;
            V(instance.urgent)
        END
    ELSE
        V(instance.mutex)
    END;
```

基础管程中的 *wait* 和 *signal* 操作

PROCEDURE **wait**

```
(VAR instance:one_instance;VAR
s:semaphore;VAR count:integer);
BEGIN

    count++;
    IF instance.urgent_count>0 THEN
        BEGIN
            instance.urgent_count--;
            V(instance.urgent)
        END
    ELSE
        V(instance.mutex);
        P(s);
    END;
```

PROCEDURE **signal**

```
(VAR instance:one_instance;
VAR s:semaphore;VAR
count:integer);

BEGIN
    IF count>0 THEN
        BEGIN
            count--;
            instance.urgent_count++;
            V(s);
            P(instance.urgent)
        END
    END;
```


用构造的管程解决读者-写者问题

```
TYPE r_and_w=MODULE;  
VAR  
instance: one_instance;  
rq, wq: semaphore;  
r_count, w_count: integer;  
reading_count, write_count:  
integer;
```

define

```
    start_r, finish_r, start_w,  
    finish_w;
```

```
use monitor_one.enter,  
    monitor_one.leave,  
    monitor_one.wait,  
    monitor_one.signal;
```

BEGIN

```
    reading_count:=0;  
    write_count:=0;  
    r_count:=0;  
    w_count:=0;
```

END;

读者的活动:

```
    r_and_w.start_r;  
    读操作;  
    r_and_w.finish_r;
```

写者的活动:

```
    r_and_w.start_w;  
    写操作;  
    r_and_w.finish_w;
```

start_r的实现

```
PROCEDURE start_r;  
BEGIN  
    monitor_one.enter(instance);  
    IF write_count>0 THEN  
        monitor_one.wait  
            (instance,rq,r_count);  
    reading_count++;  
    monitor_one.signal  
        (instance,rq,r_count);  
    monitor_one.leave(instance);  
END;
```

finish_r的实现

```
PROCEDURE finish_r;  
BEGIN  
    monitor_one.enter(instance);  
    reading_count--;  
    IF reading_count=0 THEN  
        monitor_one.signal  
            (instance,wq,w_count);  
    monitor_one.leave(instance);  
END;
```

start_w的实现

```
PROCEDURE start_w;  
BEGIN  
    monitor_one.enter(instance);  
    write_count++;  
    IF (write_count>1)  
        OR(reading_count>0)  
    THEN monitor_one.wait  
        (instance,wq,w_count);  
    monitor_one.leave(instance);  
END;
```

finish_w的实现

```
PROCEDURE finish_w;  
BEGIN  
    monitor_one.enter(instance);  
    write_count --;  
    IF (r_count=0) THEN  
        monitor_one.signal(instance,wq,w_count);  
    ELSE  
        monitor_one.signal (instance,rq,r_count);  
    monitor_one.leave(instance);  
END
```

MESA 管程

管程讨论

一个问题

是否会出现这样一种情况，有多个进程同时在管程中



► 如何解决？

避免管程中同时有两个活跃进程

确定signal之后的规则

► 三种方案

► Hoare管程

► Concurrent Pascal (Hansen)

► Mesa管程

Mesa管程

- Hoare管程：signal的缺陷
 - ✓ 两次额外的进程切换
 - ✓ 是否会使条件队列中的进程永久挂起？
- Lampson和Redell，Mesa语言（1980）
- 解决：
 - ✓ **signal → notify**
 - ✓ **notify**：当一个正在管程中的进程执行**notify(x)**时，它使得x条件队列得到通知，发信号的进程继续执行

使用 *notify* 要注意的问题

- ▶ **notify的结果：位于条件队列头的进程在将来合适的时候且当处理器可用时恢复执行**
- ▶ 由于不能保证在它之前没有其他进程进入管程，因而这个进程必须重新检查条件
 - 用**while**循环取代**if**语句
- ▶ 导致对条件变量至少多一次额外的检测（但不再有额外的进程切换），并且对等待进程在**notify**之后何时运行没有任何限制

Mesa管程：生产者-消费者问题

```
void append (char x)
{
    while(count == N) cwait(notfull);    /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;                               /* one more item in buffer */
    cnotify(notempty);                    /* notify any waiting consumer */
}

void take (char x)
{
    while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                               /* one fewer item in buffer */
    cnotify(notfull);                     /* notify any waiting producer */
}
```

William Stallings

改进 notify

► 对notify的一个很有用的改进

- 给每个条件原语关联一个监视计时器，不论条件是否被通知，一个等待时间超时的进程将被设置为就绪状态
- 当该进程被调度执行时，会再次检查相关条件，如果条件满足则继续执行

► 超时可以防止如下情况的发生：

当某些进程在产生相关条件的信号之前失败时，等待该条件的进程被无限制地推迟执行而处于饥饿状态

broadcast原语的引入

broadcast:

可以使所有在该条件上等待的进程都进入就绪状态

- ▶ 当一个进程不知道有多少进程将被激活时，这种方式是很方便的
- ▶ 例如，在生产者/消费者问题中，假设insert和remove函数都适用于可变长度的字符块，此时，如果一个生产者往缓冲区中添加了一批字符，它不需要知道每个正在等待的消费者准备消耗多少字符，而仅仅产生一个broadcast，所有正在等待的进程都得到通知并再次尝试运行
- ▶ 当一个进程难以准确判定将激活哪个进程时，也可使用广播

Hoare管程与Mesa管程的比较

- ✓ Mesa管程优于Hoare管程之处在于应用Mesa管程时出错比较少

在Mesa管程中，由于每个过程在收到信号后都检查管程变量，并且由于使用了while结构，一个进程不正确的广播或发信号，不会导致收到信号的程序出错

收到信号的程序将检查相关的变量，如果期望的条件没有满足，它会重新继续等待

锁、条件变量

PTHREADS

锁 (互斥量 *mutex*)

- ▶ 一种容易实现且有效的机制
- ▶ 适用于管理共享资源或一小段代码
- ▶ 常用于实现 **用户空间线程包**
- ▶ 锁 (互斥量) 处于两种状态:

加锁 或 **解锁**

常为“锁”为一个整型量 → **0为解锁/1为加锁**

- ▶ 提供两个过程: `mutex_lock`和`mutex_unlock`
(类似地, `lock()`和`unlock()` / `acquire()`和`release()`)

```
acquire(lock)
...
critical section
...
release(lock)
```

Pthreads中的同步机制

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

互斥量

Pthreads中保护临界区

条件变量

Pthreads中的
同步机制

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

用 *Pthreads* 解决生产者-消费者问题

```
int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

```

#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                                /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                        /* buffer used between producer and consumer */
int buffer = 0;                                       /* produce data */

void *producer(void *ptr)
{
    int i;

    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr)                                /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

```

一个缓冲区

讨论: *pthread_cond_wait*

- ▶ 当发起一个pthread_cond_wait之后，分解为三个动作：
 1. 解锁
 2. 等待，当收到一个解除等待的信号
(pthread_cond_signal或者pthread_cond_broadcast) 之后，pthread_cond_wait 马上要做的动作是：
 3. 上锁

```

#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

```

一个缓冲
区

```

#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        if (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        if (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

```

一个缓冲区

MUTEX

锁的实现

各种机制的层次

应用

并发进程

系统

同步机制
(信号量、管程、锁、条件变量……)

硬件

原子操作
(load/store、开关中断、测试&设置、……)

锁（互斥量 *mutex*）的实现

► 通过开关中断指令

```
lock() {  
    disable interrupts  
    while (value != FREE) ;  
    value = BUSY  
    enable interrupts  
}
```

- 等待锁变为空闲
- 获得锁

思考：这样实现锁是否正确？

一种修订

```
lock() {  
    disable interrupts  
    while (value != FREE) {  
        enable interrupts  
        disable interrupts  
    }  
    value = BUSY  
    enable interrupts  
}
```



设计一个方案
编码验证一下

另一种方案

```
lock() {  
    disable interrupts  
    if (value == FREE) {  
        value = BUSY  
    } else {  
        add thread to queue of threads waiting for this lock  
        switch to next run-able thread  
    }  
    enable interrupts  
}
```

无忙等待
+
开关中断

思考一下

B是目前持有锁的线程

Thread A

```
enable interrupts
}  
<user code runs>  
lock() {  
    disable interrupts  
    ...  
    switch
```

```
    back from switch  
    enable interrupts  
}
```

Thread B

```
yield() {  
    disable interrupts  
    switch
```

```
    back from switch  
    enable interrupts  
}  
<user code runs>  
unlock() (move thread A to ready queue)  
yield() {  
    disable interrupts  
    switch
```

通过 *Test&Set* 指令实现锁

mutex_lock:

```
TSL REGISTER,MUTEX  
CMP REGISTER,#0  
JZE ok  
CALL thread_yield  
JMP mutex_lock
```

ok: RET

| copy mutex to register and set mutex to 1
| was mutex zero?
| if it was zero, mutex was unlocked, so return
| mutex is busy; schedule another thread
| try again
| return to caller; critical region entered

为什么?

mutex_unlock:

```
MOVE MUTEX,#0  
RET
```

| store a 0 in mutex
| return to caller

用户空间实现锁，如线程包

对比

23

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

思考题：对多处理器
系统有效吗？为什么？

思考题：
XCHG指令

基本概念、主要方式

进程通信

为什么需要通信机制?

- ▶ 信号量及管程的不足
- ▶ 多处理器情况下原语失效



进程通信机制

- 消息传递
- send & receive原语

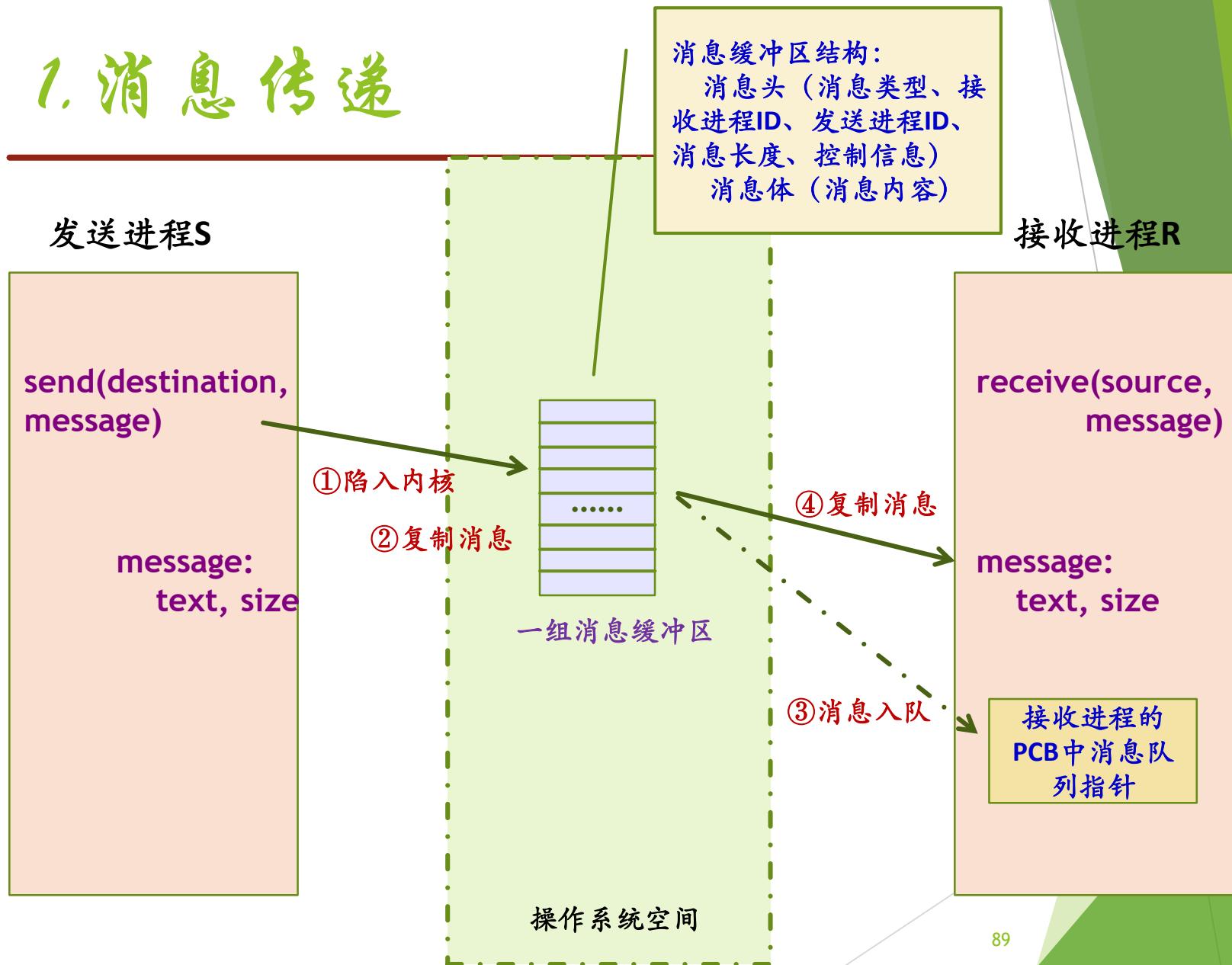
适用于
分布式系统、基
于共享内存的多处理
机系统、单处理机系
统
可以解决
进程间的同步问
题、通信问题



基本通信方式

- ▶ 消息传递
- ▶ 共享内存
- ▶ 管道
- ▶ 套接字
- ▶ 远程过程调用(RPC)

1. 消息传递



用P、V操作实现Send原语

Send (destination, message)

Begin

根据*destination*找接收进程;
如果未找到, 出错返回;

申请空缓冲区P(buf-empty);

P(mutex1);

取空缓冲区;

V(mutex1);

把消息从message处复制到
空缓冲区;

P(mutex2);

把消息缓冲区挂到接
收进程的消息队列;

V(mutex2);

V(buf-full);

End.

buf-empty初值为n

buf-full初值为0

mutex1初值为1

mutex2初值为1

思考题:
receive原
语的实现?

用消息传递实现生产者消费者问题

```
#define N 100

void producer(void)
{
    int item;
    message m;
    while(TRUE) {
        item=produce_item();
        receive(consumer, &m);
        build_messasge(&m, item);
        send(consumer, &m);
    }
}
```

```
void consumer(void)
{
    int item;
    message m;
    for(i=0;i<N;i++) send(producer, &m);
    while(TRUE) {
        receive(producer, &m);
        item=extract_item(&m);
        send(producer, &m);
        consume_item(item);
    }
}
```

用消息解
决互斥问
题

消息传递：其他考虑的问题

缓冲形式：

- ▶ 信箱
- ▶ 无缓冲
- ▶ 无限缓冲



消息丢失、身份
识别、效率

2. 共享内存

该通信模式需要解决两个问题？

进程1
地址空间

进程2
地址空间

物理内存

共享内存

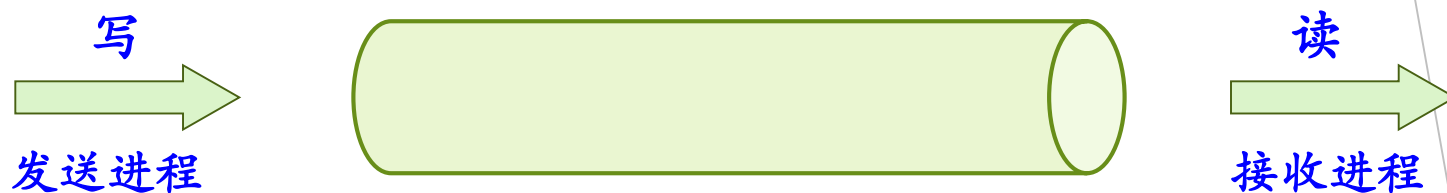
相互通信的进程间需要建立公共内存区域：

有向该共享区域写，有从该共享区域读

→ 实现进程间的信息交换

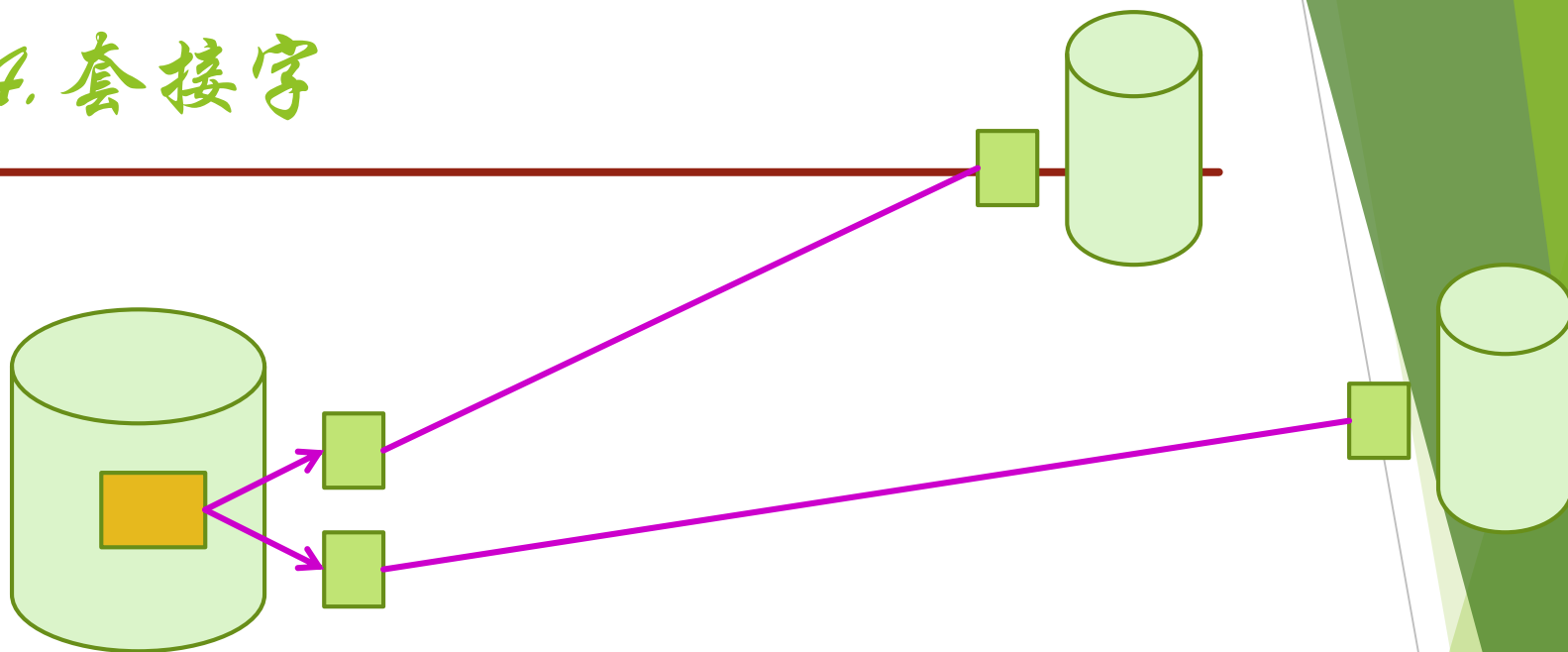
3. 管道通信方式 Pipe

- 利用一个缓冲传输介质——内存或文件连接两个相互通信的进程



- 字符流方式写入读出
- 先进先出顺序
- 管道通信机制必须提供的协调能力
 - 互斥、同步、判断对方进程是否存在

4. 套接字



服务器：创建一个套接字，并将其与本地地址/端口号绑定；监听；当捕获到一个连接请求后，接受并生成一个新的套接字，调用 `recv()` / `send()` 与客户端通信，最后关闭新建的套接字

客户端：为请求服务也创建一个套接字，并将其与本地地址/端口号绑定；指定服务器端的地址和端口号，并发出套接字连接请求；当请求被接受后，调用 `recv()` / `send()` 与服务器通信，最后关闭套接字连接

作业5

作业提交时间：

2020年11月8日晚23:30

重点阅读：

第2章相关内容：2.3.1~2.3.6、2.5.2，掌握相关概念

- 1、Tracy和Peter与金鱼的故事，理解题意并分析各种解法（见后续页）。
- 2、《ACM通讯》1966年一篇文章中提出的解决互斥问题的一种软件方法（见后续页）。
- 3、解决互斥的另一种软件方法是Lamport的面包店（bakery）算法，其思想来自于面包店或其他商店，每名顾客在到达时都得到一个票号，并按票号依次得到服务（见后续页）。

Tracy和Peter与金鱼的故事

问题描述

- ▶ Tracy和Peter共同饲养一条金鱼
- ▶ 为保持金鱼的生命，一天喂且只喂一次食
- ▶ 如果一天内喂两次鱼，则金鱼撑死
- ▶ 如果一天没有喂食，则金鱼饿死

Peter:

```
if (noFeed) {  
    feed fish  
}
```

Tracy:

```
if (noFeed) {  
    feed fish  
}
```


问题

Peter

- ◎ 3:00 观察鱼(没有喂)
- ◎ 3:05
- ◎ 3:10 feed fish
- ◎ 3:25

Tracy

- 观察鱼(没有喂)
- feed fish



金鱼
撑死了

解法1

Peter:

```
if (noNote) {  
    leave note  
if (noFeed) {  
    feed fish  
}  
    remove note  
}
```

Tracy:

```
if (noNote) {  
    leave note  
if (noFeed) {  
    feed fish  
}  
    remove note  
}
```

解法2

Peter:

leave notePeter

if (no noteTracy) {

 if (noFeed) {

 feed fish

 }

}

remove notePeter

Tracy:

leave noteTracy

if (no notePeter) {

 if (noFeed) {

 feed fish

 }

remove noteTracy

解法3

Peter:

```
    leave notePeter
    while (noteTracy)
        do nothing
```

```
}
```

```
if (noFeed) {
```

```
    feed fish
```

```
}
```

```
}
```

```
remove notePeter
```

Tracy:

```
    leave noteTracy
```

```
{
```

```
    if (no notePeter) {
```

```
        if (noFeed) {
```

```
            feed fish
```

```
        }
```

```
    remove noteTracy
```

ACM通讯

《ACM通讯》1966年一篇文章中提出了一种解决互斥问题的一种软件方法。请举出证明该方法不正确的一个反例。

```
boolean blocked[2];
int turn;
void P(int id)
{
while(true) {
blocked[id] = true;
while(turn != id) {
while(blocked[1-id])
/* 不做事*/;
turn = id;
}
/* 临界区*/
blocked[id] = false;
/* 其余部分 */
}
}
```

```
void main()
{
blocked[0] = false;
blocked[1] = false;
turn = 0;
parbegin(P(0), P(1));
}
```

Bakery算法

```
boolean choosing[n]; //初值 false
int number[n];        //初值 0
while(true) {
    choosing[i] = true;
    number[i] = 1 + getmax(number[], n);
    choosing[i] = false;
    for(int j = 0; j < n; j++){
        while(choosing[j]) { };
        while((number[j] != 0) &&(number[j],j) <
(number[i],i)) { };
    }
    /* 临界区*/;
    number [i] = 0;
    /* 其余部分*/;
}
```

每个数组的第*i*个元素可由进程*i*读或写，但其他进程只能读。

要求：

(1)用文字描述这个算法。

(2)说明它实施了互斥。

XV6源代码阅读要求

阅读XV6源代码之“锁”的部分，结合代码写出总结报告。

提交时间：2020年11月8日晚23:30

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern, layered effect on the right side of the slide.

Thanks

The End