

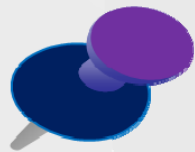
《操作系统》

进程与线程

首都师范大学
信息工程学院
霍其润

Processes and Threads

- ❖ Processes
- ❖ Threads
- ❖ Interprocess communication
- ❖ Classical IPC problems
- ❖ Scheduling



三 进程间通信 (Interprocess Communication)

1、基本问题

Interprocess Communication

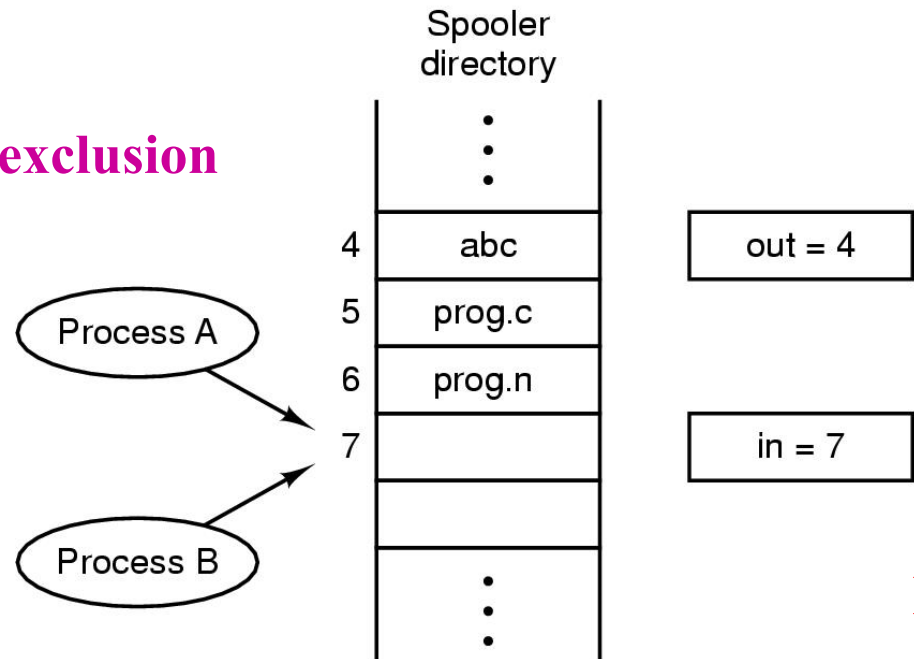
Three Issues

Communication

Mutual exclusion

Synchronous

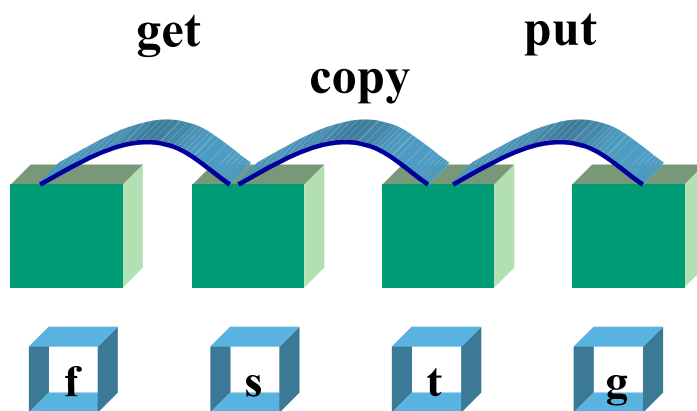
Mutual exclusion



Race Conditions

- Two processes want to access shared memory at same time

Synchronization

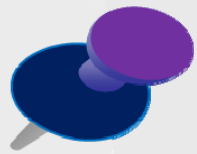


	f	s	t	g	结果
初始状态	3,4,...,m	2	2	(1,2)	
gcp	4,5,...,m	3	3	(1,2,3)	正确
gpc	4,5,...,m	3	3	(1,2,2)	错误
cgp	4,5,...,m	3	2	(1,2,2)	错误
cp,g	4,5,...,m	3	2	(1,2,2)	错误
p,c,g	4,5,...,m	3	2	(1,2,2)	错误
p,g,c	4,5,...,m	3	3	(1,2,2)	错误

Coming data blocks

- 进程间的制约关系

- 间接制约：进行竞争——独占分配到的部分或全部共享资源，“互斥”
- 直接制约：进行协作——等待来自其他进程的信息，“同步”



三 进程间通信 (Interprocess Communication)

2、临界区

Critical Regions (0)

临界资源 (Critical Resources) : 一种一次只能为一个进程服务的资源。

临界区 (Critical Section) : 进程中访问临界资源的程序。每个使用该资源的进程都要包含一个临界区。

...

critical section

... ; remainder section

两个进程不能同时进入访问同一临界资源的临界区，这称为**进程互斥**。

一个飞机订票系统，两个终端，运行T1、T2进程

T1 :

...

Read(x) ;

i f x \geq 1 t hen

x:=x- 1;

w r i t e(x) ;

...

T2:

...

Read(x) ;

i f x \geq 1 t hen

x:=x- 1;

w r i t e(x) ;

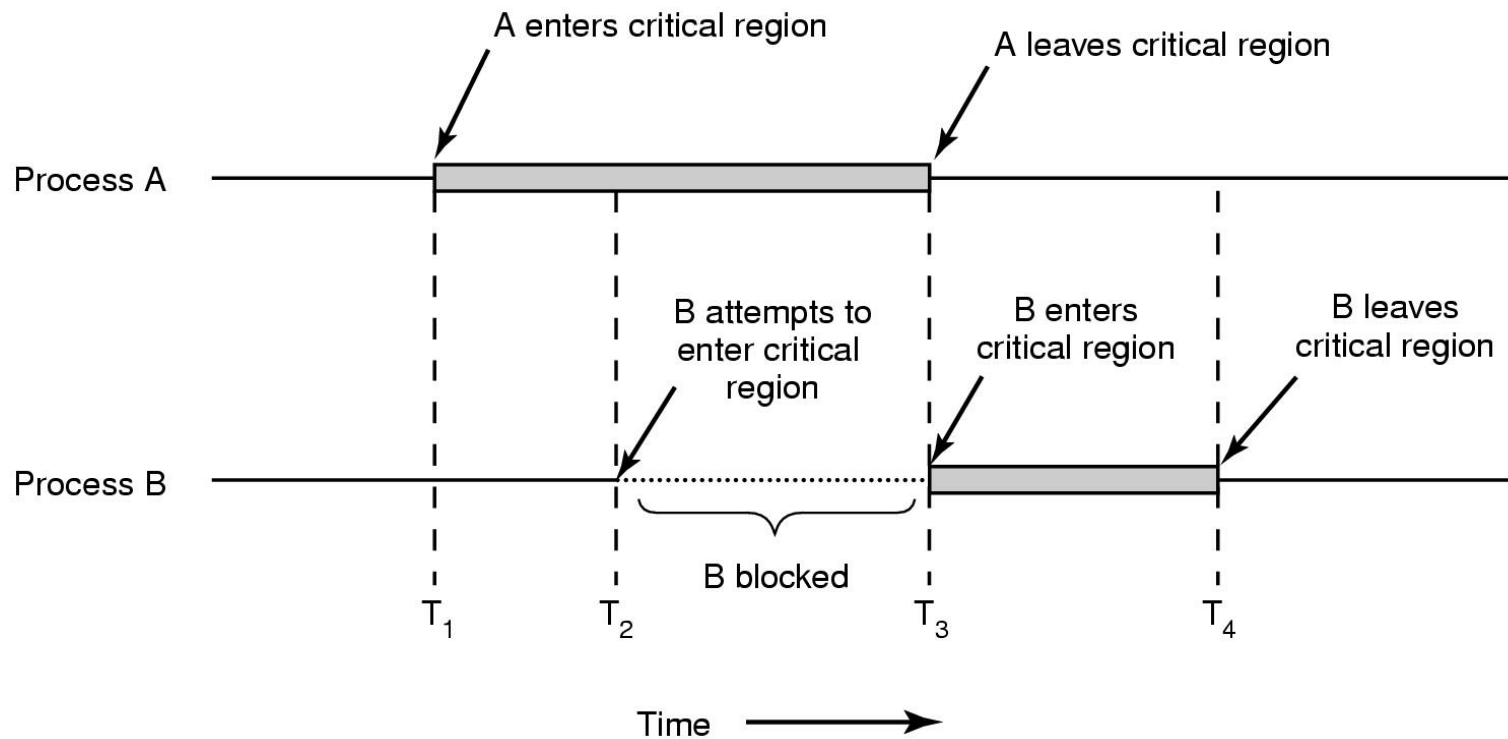
...

Critical Regions (1)

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region (空闲则入)
2. No assumptions made about speeds or numbers of CPUs (忙则等待)
3. No process running outside its critical region may block another process (让权等待)
4. No process must wait forever to enter its critical region (有限等待)

Critical Regions (2)



- Mutual exclusion using critical regions




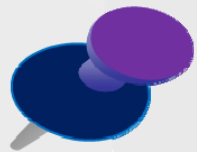
entry section

critical section

exit section

remainder section





三 进程间通信 (Interprocess Communication)

3、互斥的几种实现

Mutual Exclusion with Busy Waiting

Proposals for achieving mutual exclusion:

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

Mutual Exclusion with Busy Waiting (1)

while (lock!=0);	<a>
lock= 1;	

critical section

lock= 0;

remainder section

Lock Variables

Mutual Exclusion with Busy Waiting (2)

```
while (TRUE) {  
    while (turn != 0)    /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1)    /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(b)

Strict Alternation

(a) Process 0. (b) Process 1.

Mutual Exclusion with Busy Waiting (3)

```
#define FALSE 0
#define TRUE 1
#define N      2                /* number of processes */

int turn;                        /* whose turn is it? */
int interested[N];              /* all values initially 0 (FALSE) */

void enter_region(int process);  /* process is 0 or 1 */
{
    int other;                   /* number of the other process */

    other = 1 - process;         /* the opposite of process */
    interested[process] = TRUE;  /* show that you are interested */
    turn = process;              /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)   /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Peterson's solution for achieving mutual exclusion

Mutual Exclusion with Busy Waiting (4)

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

Entering and leaving a critical region using the
TSL instruction

enter_region:

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

```
| put a 1 in the register
| swap the contents of the register and lock variable
| was lock zero?
| if it was non zero, lock was set, so loop
| return to caller; critical region entered
```

leave_region:

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

```
| store a 0 in lock
| return to caller
```

Entering and leaving a critical region
using the XCHG instruction.

• 优点

- 简单，容易验证其正确性
- 可以支持进程内存在多个资源对应的临界区，只需为每个临界资源对应的临界区设立一个布尔变量

• 缺点

- 等待要耗费CPU时间，即“忙等”，不能实现“让权等待”
- 可能产生“饿死”现象：有的进程可能永远执行不了

试考虑以下进程 P_A 和 P_B 反复使用临界区的情况：

P_A

A: lock(key [S])

〈 S 〉

unlock(key [S])

Goto A

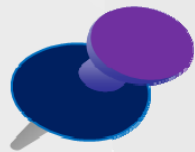
P_B

B: lock(key [S])

〈 S 〉

unlock(key [S])

Goto B



三 进程间通信 (Interprocess Communication)

4、信号量的定义

试考虑以下进程 P_A 和 P_B 反复使用临界区的情况：

P_A

A: lock(key [S])

〈 S 〉

unlock(key [S])

Goto A

P_B

B: lock(key [S])

〈 S 〉

unlock(key [S])

Goto B

Semaphores

前面的互斥算法缺乏公平性，只有获得处理机的进程才可进行进入临界区的测试，且测试结果具有偶然性。

需要一个地位高于进程的管理者来解决公共资源的使用问题。OS可从进程管理者的角度来处理互斥的问题，信号量就是OS提供的管理公共资源的有效手段。

信号量代表可用资源实体的数量。

- 信号量具有以下特性：
 - 1) 信号量是一个整形变量。
 - 2) 每一个信号量表示一种系统资源的状况，其值表示该资源当前可用的数量，初值为非零。
 - 3) 每一个信号量都对应一个空或非空的等待队列。该队列就是信号量所代表的资源的等待队列。
 - 4) 对信号量只能实施down、up（P、V）操作，只有P、V操作原语才能改变其值。

原语 $down(s); P(s)$

procedure down(var s:semaphore)

begin

s:=s-1;

if s<0 then W(s);

end

//将调用P操作原语的进程置
成等待信号量s的状态。

原语 $up(s); V(s)$

Procedure up(var s:semaphore)

begin

s:=s+1;

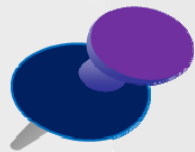
if $s \leq 0$ then R(s);

end

//从信号量s的等待队列中释放一个进程。

➤ 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

➤ **初始化**资源信号量为指定一个非负整数值，表示**空闲资源总数**——若为非负值表示**当前的空闲资源数**，若为负值其绝对值表示**当前等待临界区的进程数**



三 进程间通信 (Interprocess Communication)

5、信号量实现互斥

Mutexes

- ✓ 在实现进程互斥时，信号量的初值设为1，表示中只允许一个进程进入临界区。
- ✓ 在进程执行过程中，当进入临界区时执行down操作，在离开临界区时执行up操作，使临界区位于对同一个信号量的down操作和up操作之间。

```
down(mutex);
```

```
critical section
```

```
up(mutex);
```

```
remainder section
```

- **mutex**为互斥信号量，其初值为1；在每个进程中将临界区代码置于down(mutex)和up(mutex)原语之间
- 必须成对使用down和up原语：遗漏down原语则不能保证互斥访问，遗漏up原语则不能在使用临界资源之后将其释放（给其他等待的进程）；down、up原语不能次序错误、重复或遗漏

- 在进程互斥中使用的信号量，每个进程都可以对它实施down操作，这样的信号量又称为公用信号量。

用信号量实现两并发进程的互斥

s:semaphore;

s:=1;

进程A

· · · · ·

down(s);

临界区CRA;

up(s);

· · · · ·

进程B

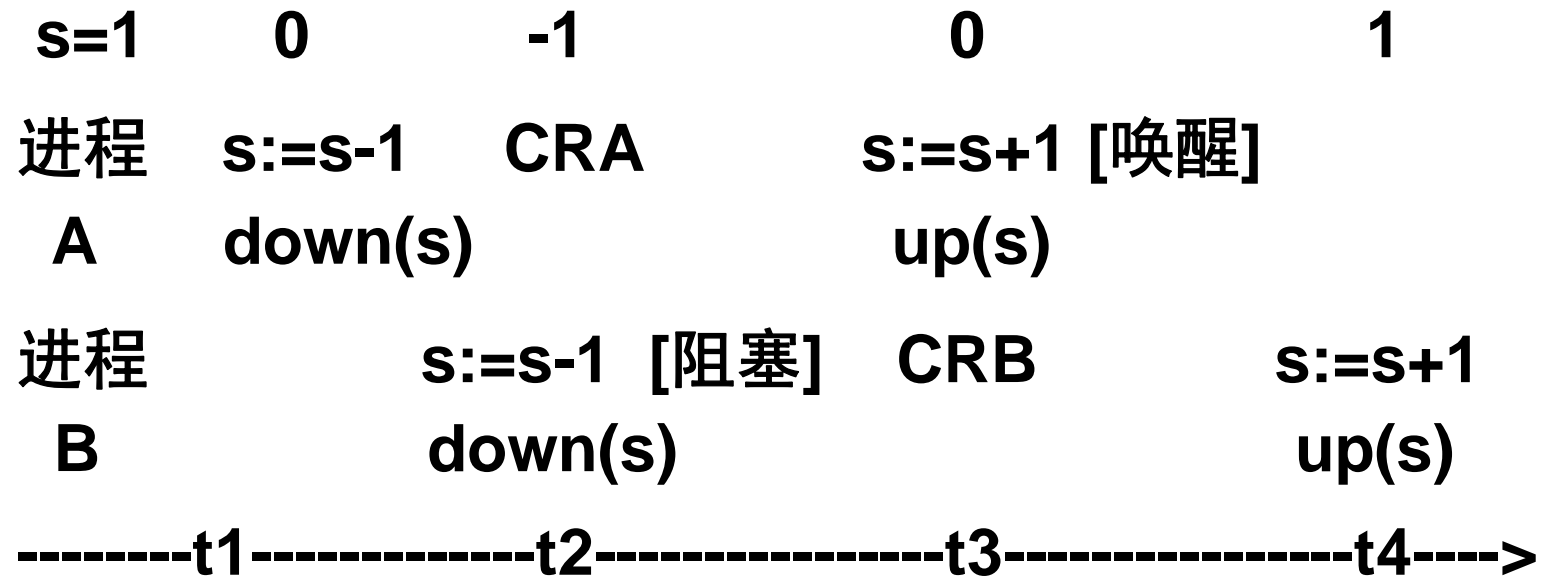
· · · · ·

down(s);

临界区CRB;

up(s);

· · · · ·



Mutexes

mutex_lock:

TSL REGISTER,MUTEX

| copy mutex to register and set mutex to 1

CMP REGISTER,#0

| was mutex zero?

JZE ok

| if it was zero, mutex was unlocked, so return

CALL thread_yield

| mutex is busy; schedule another thread

JMP mutex_lock

| try again later

ok: RET | return to caller; critical region entered

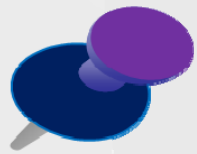
mutex_unlock:

MOVE MUTEX,#0

| store a 0 in mutex

RET | return to caller

- Implementation of mutex_lock and mutex_unlock



三 进程间通信 (Interprocess Communication)

6、同步问题

Synchronous

P_C （计算进程）

:

A: 计算
 得到计算结果
 Buf ← 计算结果
 Goto A

P_P （打印进程）

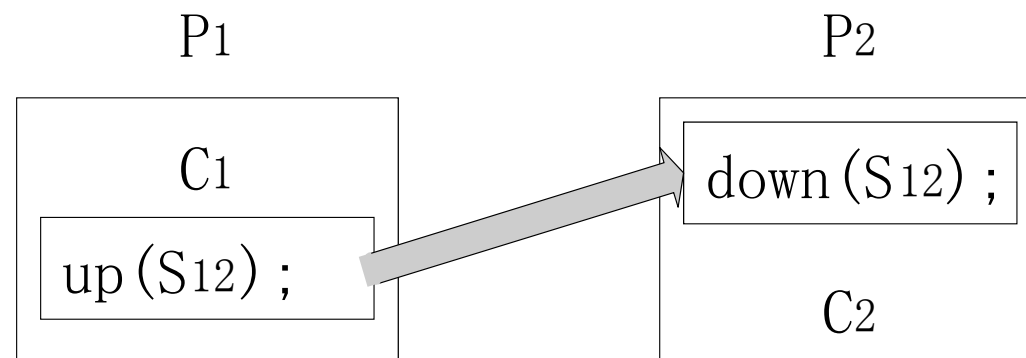
:

B: 打印 Buf中的数据
 清除 Buf中的数据
 Goto B

- 消息(事件) 进程互相给对方进程发送执行条件已经具备的信号。
- 同步 一组并发进程，因直接制约而互相发送消息而进行互相合作、互相等待，使得各进程按一定的速度执行的过程称为进程间的同步。

一般来说，可以把各进程之间发送的消息作为信号量看待。与进程互斥时不同的是，这里的信号量只与制约进程及被制约进程有关而不是与整组并发进程有关。因此，称该信号量为**私用信号量**（**Private Semaphvre**）。

利用信号量来描述前趋关系



- 前趋关系：并发执行的进程P1和P2中，分别有代码C1和C2，要求C1在C2开始前完成；
- 为该前趋关系设置一个信号量 S12，初值为0

如运算和打印进程分别为进程T1和T2，要求T1和T2同步运行，则

设定两个信号量s1和s2，s1的初值为1，s2的初值为0。

```
s1,s2:semaphore;
```

```
s1=1,s2=0;
```

```
cobegin
```

```
    repeat T1;
```

```
    repeat T2;
```

```
coend;
```

procedure T1:

begin

. . .

down(s1);

m1;

up(s2);

. . .

end;

procedure T2:

begin

. . .

down(s2);

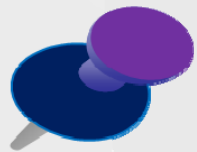
m2;

up(s1);

. . .

end;

- 只能由一个进程对其实施down操作的信号量称为该进程的私有信号量。
- s1是T1的私有信号量，s2是T2的私有信号量。
- 在两个进程相互推进的运行过程中，哪个进程的私有信号量为1，就表示它可以向前推进。

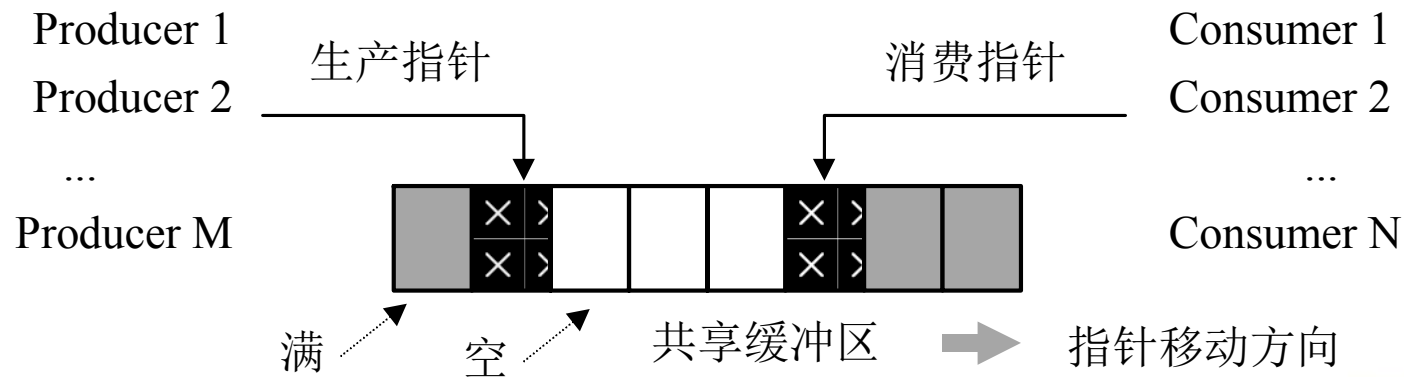


三 进程间通信 (Interprocess Communication)

7、生产者-消费者问题

生产者—消费者问题 (the producer-consumer problem)

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。



Sleep and Wakeup

```
#define N 100                                     /* number of slots in the buffer */
int count = 0;                                    /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                                /* repeat forever */
        item = produce_item();                    /* generate next item */
        if (count == N) sleep();                  /* if buffer is full, go to sleep */
        insert_item(item);                        /* put item in buffer */
        count = count + 1;                        /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);        /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                                /* repeat forever */
        if (count == 0) sleep();                  /* if buffer is empty, got to sleep */
        item = remove_item();                    /* take item out of buffer */
        count = count - 1;                        /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer);    /* was buffer full? */
        consume_item(item);                      /* print item */
    }
}
```

Producer-consumer problem with fatal race condition

引入：信号量——用来累计唤醒次数
(对方的执行许可)

当产生一个唤醒信号执行**up**操作;
用到一个唤醒信号执行**down**操作

Semaphores

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

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

/ number of slots in the buffer */*
/ semaphores are a special kind of int */*
/ controls access to critical region */*
/ counts empty buffer slots */*
/ counts full buffer slots */*

/ TRUE is the constant 1 */*
/ generate something to put in buffer */*
/ decrement empty count */*
/ enter critical region */*
/ put new item in buffer */*
/ leave critical region */*
/ increment count of full slots */*

/ infinite loop */*
/ decrement full count */*
/ enter critical region */*
/ take item from buffer */*
/ leave critical region */*
/ increment count of empty slots */*
/ do something with the item */*

The producer-consumer problem using semaphores

- 采用信号量机制:

- full是“满”数目, 初值为0, empty是“空”数目, 初值为N。实际上, full和empty是同一个作用, 始终有 $full + empty = N$
- mutex用于访问缓冲区时的互斥, 初值是1

- 每个进程中各个down/P操作、up/V的次序是否可变?

Producer

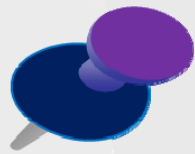
P(empty);
P(mutex); //进入区
one unit --> buffer;
V(mutex);
V(full); //退出区

Consumer

P(full);
P(mutex); //进入区
one unit <-- buffer;
V(mutex);
V(empty); //退出区

- 无论是生产者进程还是消费者进程，其中up/V操作的次序是无关紧要的，但down/P操作的次序不能颠倒：先检查资源数目，再检查是否互斥。

——否则进程可能阻塞在临界区里面，致使其他进程也进不了临界区，从而均无法执行下去。



三 进程间通信 (Interprocess Communication)

8、管程

信号量同步的缺点

- **同步操作分散：**信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁（如P、V操作的次序错误、重复或遗漏）；
- **易读性差：**要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发程序；
- **不利于修改和维护：**各模块的独立性差，任一组变量或一段代码的修改都可能影响全局；
- **正确性难以保证：**操作系统或并发程序通常很大，很难保证这样一个复杂的系统没有逻辑错误。

管程 (Monitor) 的引入

管程是一种具有面向对象程序设计思想的同步机制。它提供了与信号量机制相同的功能。

管程 (Monitor) 概念是著名学者Hore于1974年提出的，并在很多系统中得到实现，诸如Pascal、Java、Modula-3等。

管程是由局部数据结构、多个处理过程和一套初始化代码组成的模块。

monitor *example*

integer *i*;
condition *c*;

procedure *producer*();

·
·
·

end;

procedure *consumer*();

·
·
·

end;

end monitor;

Example of a monitor

Monitors (1)

管程的特征：

- (1) 管程内的数据结构只能被管程内的过程访问，任何外部访问都是不允许的；
- (2) 进程可通过调用管程的一个过程进入管程；
- (3) 任何时间只允许一个进程进入管程，其他要求进入管程的进程统统被阻塞到等待管程的队列上。

实现方法：

- 将所有的临界区转换成管程中的过程——互斥；
- 引入条件变量**full**、**empty**及相关的操作**wait**、**signal**——同步。

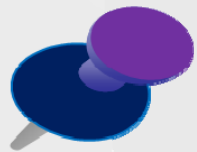
Monitors (2)

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    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;
end;
```

- Outline of producer-consumer problem with monitors

- 管程实现了临界区互斥的自动化，使并行编程更容易保证正确性。
- 管程是一个编程语言中用到的概念，必须用支持管程的语言才能实现。



三 进程间通信 (Interprocess Communication)

9、其他通信方式

Message Passing

之前介绍的方法多用于有公共内存的进程间通信，当涉及局域网相连的机器之间的通信——**消息传递**。

消息传递中的管理机制由操作系统提供，主要体现为以下两个原语。

发送原语： `send(destination, message)`，其中，`destination`为消息的目的地（接收进程名：`process@machine.domain`）。该原语表示发送消息到进程`destination`。

接收原语： `receive(source, message)`，其中，`source`是消息发出地（发送进程名）。该原语表示从进程`source`接收消息。

Producer-Consumer Problem with Message Passing (1)

```
#define N 100                                /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                                /* message buffer */

    while (TRUE) {
        item = produce_item();                /* generate something to put in buffer */
        receive(consumer, &m);                /* wait for an empty to arrive */
        build_message(&m, item);              /* construct a message to send */
        send(consumer, &m);                   /* send item to consumer */
    }
}

. . .
```

The producer-consumer problem with N messages.

Producer-Consumer Problem with Message Passing (2)

• • •

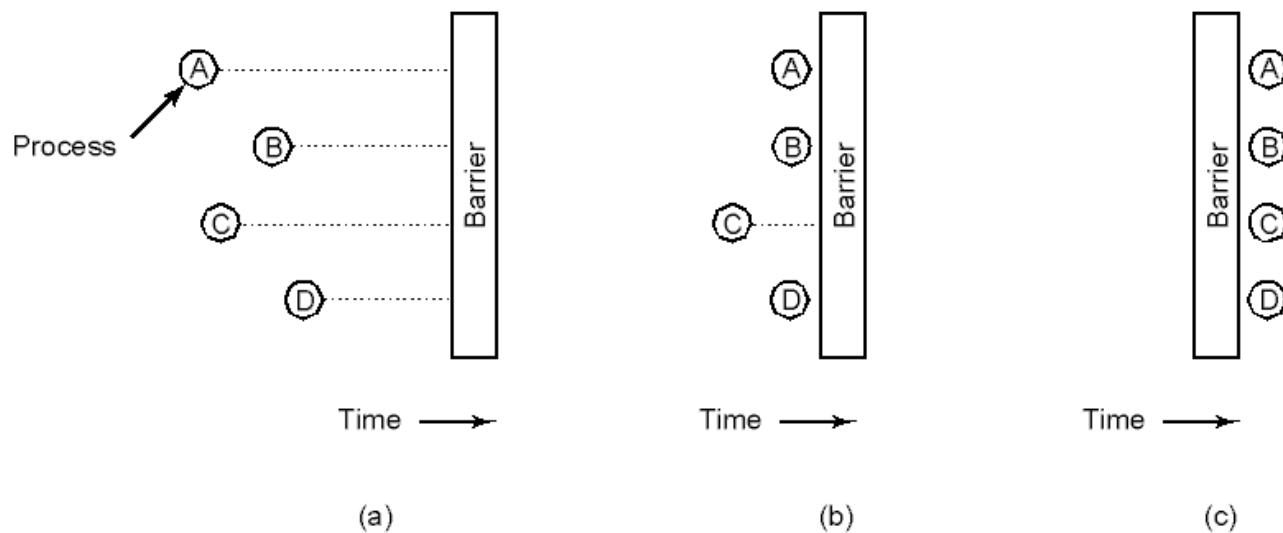
```
void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);                /* get message containing item */
        item = extract_item(&m);                /* extract item from message */
        send(producer, &m);                    /* send back empty reply */
        consume_item(item);                    /* do something with the item */
    }
}
```

The producer-consumer problem with N messages.

Barriers

针对
一组
进程的
同步机
制。



- Use of a barrier
 - processes approaching a barrier
 - all processes but one blocked at barrier
 - last process arrives, all are let through

补充：进程间通信——进程之间交换信息

- **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。
 - 主要用于控制进程执行速度的作用。
 - 优点：速度快。
 - 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多次通信。
- **高级通信：**能够传送任意数量的数据，目的主要是用于交换信息。

高级通信常用方式

- 共享存储区方式
 - 不要求数据移动，可以任意读写和使用任意数据结构，需要进程互斥和同步的辅助来确保数据一致性
- 消息或邮箱机制
- 管道、文件、文件映射

共享存储区

这种通信需要在内存中开辟一个共享的存储空间，供进程之间进行数据传递。比如计算进程将所得的结果送入内存共享区的缓冲区环中，打印进程从中将结果取出来，就是一个利用共享存储器进行通信的例子。这种通信方式在UNIX、Linux、Windows及OS/2等系统中都有具体的实现。

共享存储器系统中，共享的空间一般应当是需要互斥访问的临界资源。诸多进程为了避免丢失数据或重复取数，需要执行特定的同步协议。

在利用共享存储器进行通信之前，信息的发送者和接收者都要将共享空间纳入到自己的虚地址空间中，让它们都能访问该区域。存储器管理模块将共享空间映射成实际的内存空间。

- (1) 创建或删除共享存储区
- (2) 共享存储区的附接与断接
- (3) 共享存储区状态查询
- (4) 共享存储区管理

共享存储器系统的特点：

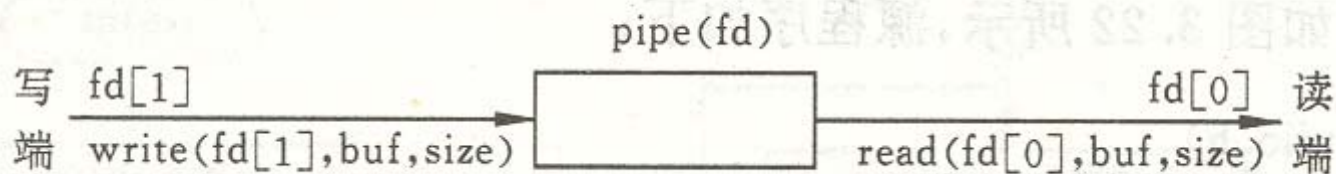
- 利用共享存储器系统进行通信的效率特别高，适用于通信速度要求特别高的场合。
- 这种同步与互斥机制的实现一般要由程序员来承担，系统仅仅提供一个共享内存空间的管理机制。

Linux的共享存储区

- 创建或打开共享存储区(shmget): 依据用户给出的整数值key, 创建新区或打开现有区, 返回一个共享存储区ID。
- 连接共享存储区(shmat): 连接共享存储区到本进程的地址空间, 可以指定虚拟地址或由系统分配, 返回共享存储区首地址。父进程已连接的共享存储区可被fork创建的子进程继承。
- 拆除共享存储区连接(shmdt): 拆除共享存储区与本进程地址空间的连接。
- 共享存储区控制(shmctl): 对共享存储区进行控制。如: 共享存储区的删除需要显式调用shmctl(shmid, IPC_RMID, 0)。

管道(pipe)

管道是可供进程共享的一种特殊文件，是单向的，主要用于连接一个写入进程和一个读出进程，以实现它们之间的数据通信。在使用管道前要建立相应的管道，然后才可使用。



利用UNIX提供的系统调用pipe，可建立一条同步通信管道。

其格式为：
`pipe(fd)`
`int fd [2] ;`

这里，fd [1] 为写入端，fd [0] 为读出端。

示例:

用C语言编写一个程序，建立一个pipe，同时父进程生成一个子进程，子进程向pipe中写入一字符串，父进程从pipe中读出该字符串。

解： 程序如下：

```
#include <stdio.h>
main()
{
    int x, fd [2] ;
    char buf [30] , s [30] ;
    pipe(fd); /*创建管道*/
    while((x=fork())!=-1); /*创建子进程失败时，循环*/
    if(x==0)
```

```
{
    sprintf(buf, "This is an example \n");
    write(fd [1] , buf, 30); /*把buf中字符写入管道*/
    exit(0);
}
else/*父进程返回*/
{
    wait(0);
    read(fd [0] , s, 30); /*父进程读管道中字符*/
    printf("%s", s);
}
}
```

Linux 管道

- 通过pipe系统调用创建无名管道，得到两个文件描述符，分别用于写和读。
 - `int pipe(int fildes[2]);`
 - 文件描述符fildes[0]为读端，fildes[1]为写端；
 - 通过系统调用write和read进行管道的写和读；
 - 进程间双向通信，通常需要两个管道；
 - 只适用于父子进程之间或父进程安排的各个子进程之间；
- Linux中的命名管道，可通过mknod系统调用建立：指定mode为S_IFIFO
 - `int mknod(const char *path, mode_t mode, dev_t dev);`

Windows NT 管道

无名管道：类似于Linux管道，CreatePipe可创建无名管道，得到两个读写句柄；利用ReadFile和WriteFile可进行无名管道的读写；

```
BOOL CreatePipe( PHANDLE hReadPipe,  
                // address of variable for read handle  
                PHANDLE hWritePipe,  
                // address of variable for write handle  
                LPSECURITY_ATTRIBUTES lpPipeAttributes,  
                // pointer to security attributes  
                DWORD nSize          // number of bytes reserved for pipe  
                );
```

小 结

- 进程间通信的关系
 - 竞争：互斥（临界资源；临界区）
 - 协作：同步
- 信号量的含义、操作
- 信号量实现互斥同步
- 管程等其他方式