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6讲

同步互斥机制2
进程通信机制

同步互斥机制与通信机制

- 管程monitor
- 进程间通信

  Inter-Process Communication
- 典型操作系统的IPC机制

语言机制、条件变量/wait/signal

管程 MONITOR

为什么会出现管程？

问题

- 信号量机制的不足：程序编写困难、易出错

解决

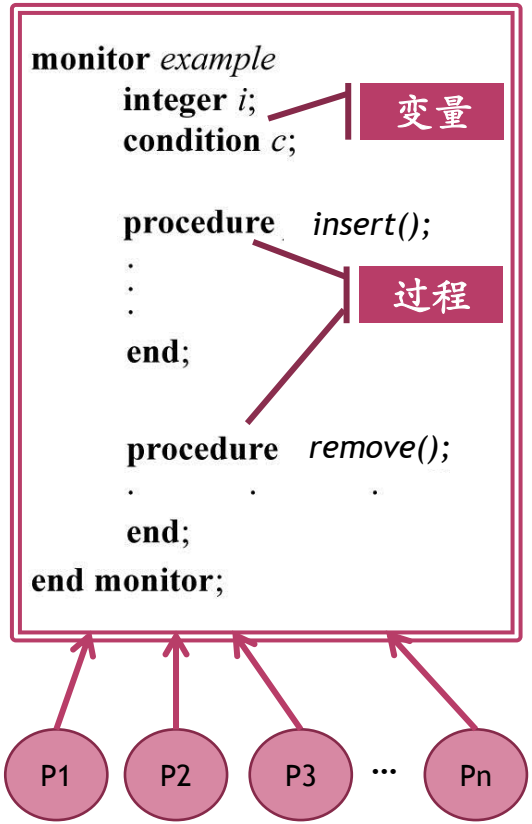
- Brinch Hansen(1973)
- Hoare（1974）

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地访问管程中的数据结构

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

  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

应用管程时遇到的问题

设问：是否会出现这样一种场景，有多个进程同时在管程中出现？

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

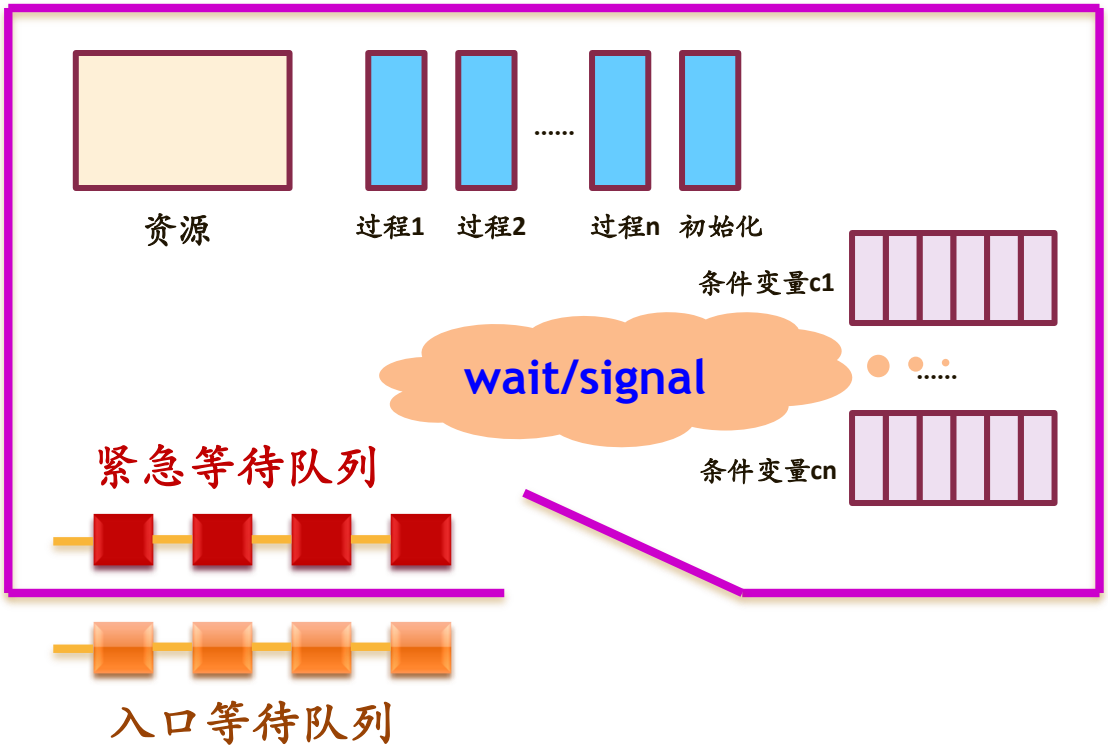
如何解决？

三种处理方法：

- P等待Q执行 Hoare √
- Q等待P继续执行 MESA
- 规定唤醒操作为管程中最后一个可执行的操作 Hansen，并发pascal

HOARE管程

HOARE管程示意图



HOARE管程说明

- 因为管程是互斥进入的，所以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
  - 在管程内需要设置一个进程等待队列，称为紧急等待队列，紧急等待队列的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链末尾

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末尾

用管程实现生产者消费者、JAVA 中的类似机制

管程的应用

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

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;
```

JAVA中的类似机制(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中的类似机制(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
   }
```

MESA管程与Hoare管程比较

MESA管程

MESA管程

- Lampson和Redell，Mesa语言（1980）
- Hoare管程的一个缺点
  - ✓ 两次额外的进程切换
- 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，不论是否被通知，一个等待时间超时的进程将被设为就绪态
  - 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，等待该条件的进程就会被无限制地推迟执行而处于饥饿状态

引入BROADCAST

broadcast：使所有在该条件上等待的进程都被释放并进入就绪队列

- 当一个进程不知道有多少进程将被激活时，这种方式是非常方便的
  - 例子：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，而仅仅执行一个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Mesa管程错误比较少
- 在Mesa管程中，由于每个过程 在收到信号后都重新检查管程变量，并且由于使用了while结构，一个进程不正确的broadcast广播或发信号notify，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

管程小结

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下面几点：
（1）只能通过管程的某个过程才能访问资源；
（2）管程是互斥的，某个时刻只能有一个进程或线程调用管程中的过程

条件变量：为提供进程与其他进程通信或同步而引入

- wait/signal 或 wait/notify 或 wait/broadcast

锁、条件变量

PTHREAD中的同步机制

PTHREAD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

保护临界区

条件变量

解决同步

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PTHREAD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);
}
```

生产者-消费者问题

```
#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);
}
```

讨论：PTHREAD_COND_WAIT

- pthread_cond_wait的执行分解为三个主要动作：

  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_broad_cast）之后，pthread_cond_wait马上需要做的动作是：

  3、上锁

生产者-消费者问题

```
#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);
}
```

基本概念、主要方式

进程间通信IPC

为什么需要通信机制？

- 信号量及管程的不足
- 不适用多处理器情况

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

进程通信机制

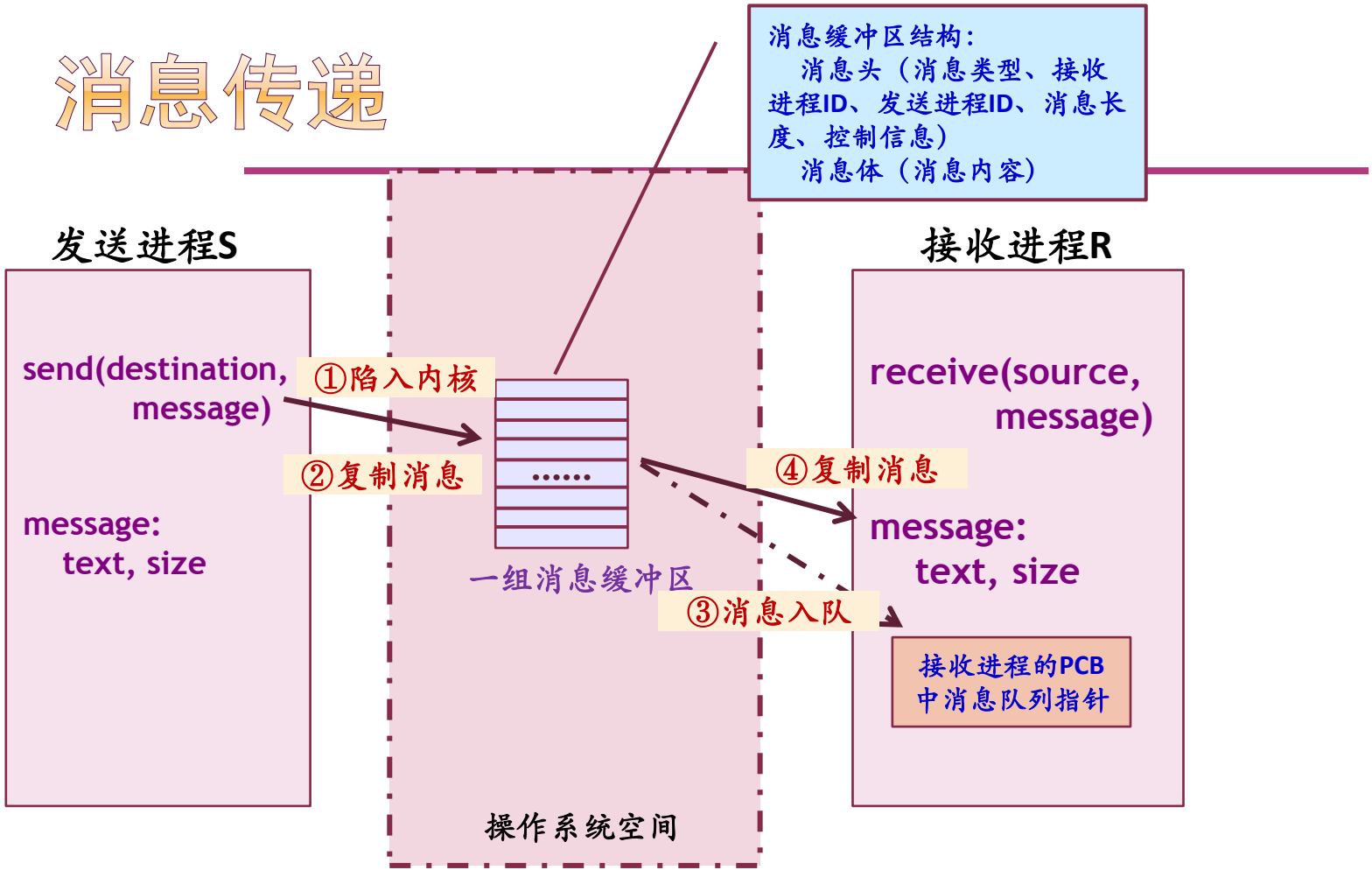
- 消息传递

  send & receive原语

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

消息传递


消息缓冲区结构：
消息头（消息类型、接收进程ID、发送进程ID、消息长度、控制信息）
消息体（消息内容）
发送进程S
接收进程R
send(destination, message)
①陷入内核
②复制消息
message:
text, size
.....
一组消息缓冲区
④复制消息
receive(source, message)
message:
text, size
③消息入队
接收进程的PCB中消息队列指针
操作系统空间

用P、V操作实现SEND原语

```
Send（destination, message）
{
 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);
}
```

receive原语的实现?

信号量:
buf-empty初值为N
buf-full初值为0
mutex1初值为1
mutex2初值为1

用消息传递实现生产者消费者问题

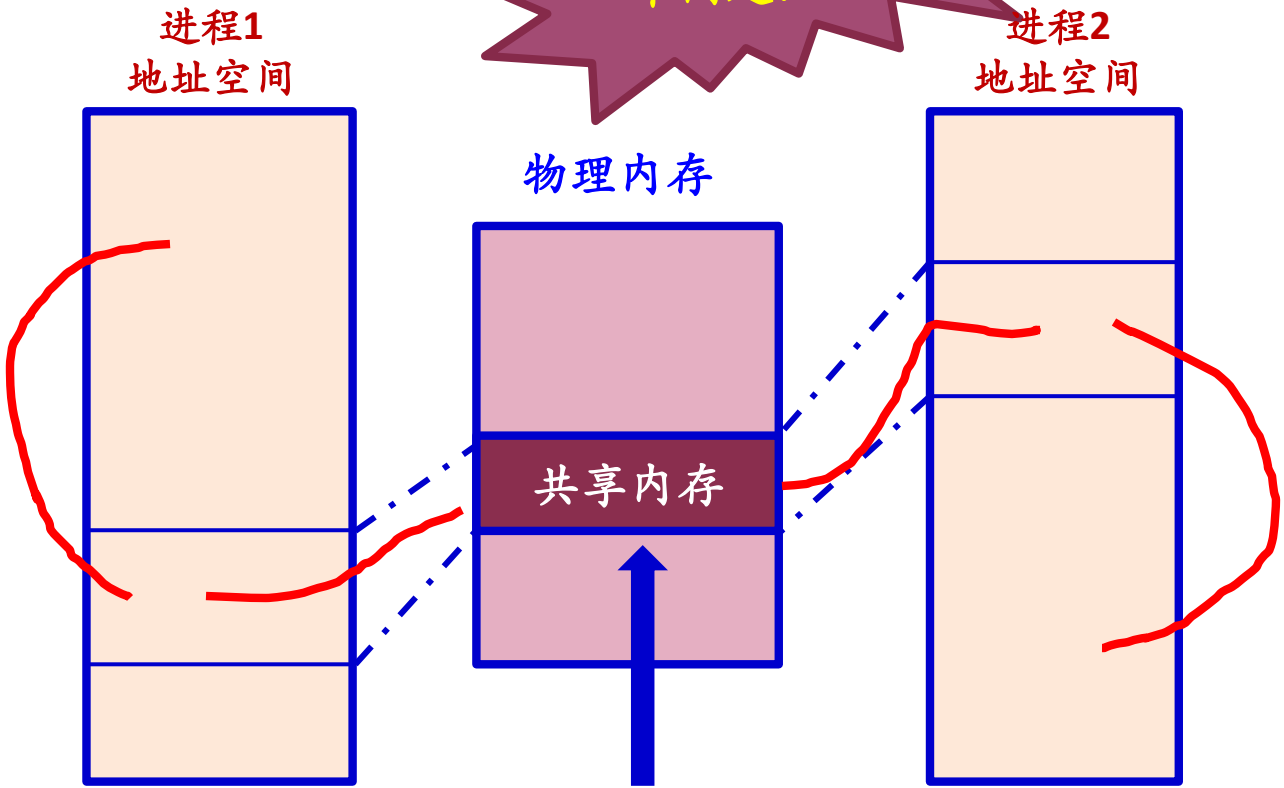
```
#define N 100
void producer(void)
{    int item;
     message m;
     while(TRUE) {
          item=produce_item();
          receive(consumer, &m);
          build_messasge(&m, item);
          send(consumer, &m);
     }
}
```

用消息解决互斥问题

```
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);
     }
}
```

共享内存

该通信模式需要解决两个问题?



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

Unix、Windows、Linux

典型操作系统中的IPC机制

进程同步/通信实例

UNIX

管道、消息队列、共享内存、信号量、信号、套接字

Linux

管道、消息队列、共享内存、信号量、信号、套接字

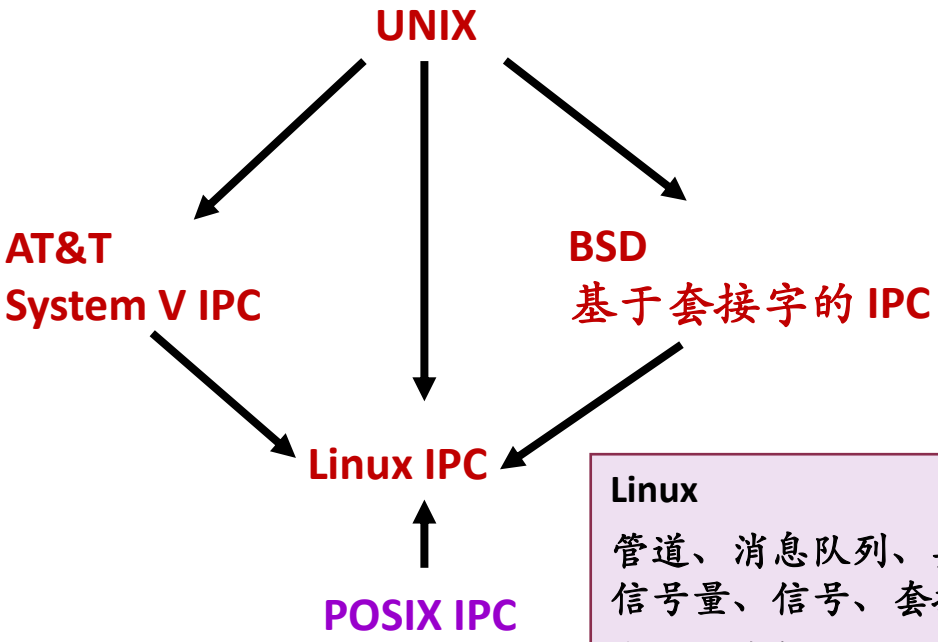
内核同步机制：原子操作、自旋锁、读写锁、信号量、屏障、BKL

Windows

同步对象：互斥对象、事件对象、信号量对象
临界区对象
互锁变量

套接字、文件映射、管道、命名管道、邮件槽、剪贴板、动态数据交换、对象连接与嵌入、动态链接库、远程过程调用

LINUX的进程通信机制



Linux

管道、消息队列、共享内存、信号量、信号、套接字

内核同步机制：原子操作、自旋锁、读写锁、信号量、屏障 … …

LINUX内核同步机制

- 原子操作
- 自旋锁
- 读写自旋锁
- 信号量
- 读写信号量
- 互斥体
- 完成变量
- 顺序锁
- 屏障
- …

原子操作(1)

- 不可分割，在执行完之前不会被其他任务或事件中断
- 常用于实现资源的引用计数
- atomic_t

```
typedef struct { volatile int counter; } atomic_t;

原子操作API包括:
atomic_read(atomic_t * v);
atomic_set(atomic_t * v, int i);
void atomic_add(int i, atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
```

原子操作(2)

- 例子

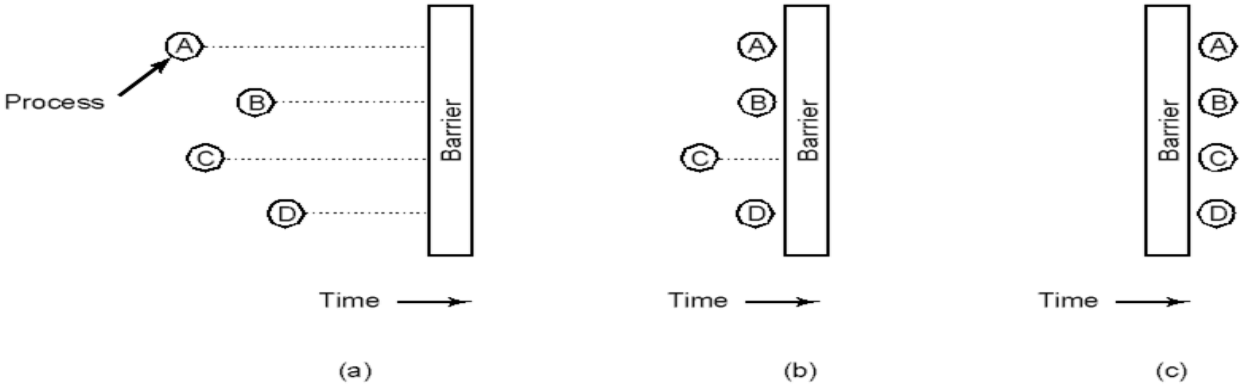
```
atomic_t   v = atomic_init(0);

atomic_set(&v, 4);
atomic_add(2, &v);
atomic_inc(&v);
printk("%d\n", atomic_read(&v);


int atomic_dec_and_test(atomic_t  *v)
```

屏障(BARRIER)

矩阵运算

- 一种同步机制(又称栅栏、关卡)
- 用于对一组线程进行协调
- 应用场景

一组线程协同完成一项任务，需要所有线程都到达一个汇合点后再一起向前推进



本讲重点

- 管程
  - 如何保证互斥
  - 如何保证同步——条件变量及wait/signal
  - Hoare管程
  - MESA管程
- 进程间通信
  - 消息传递、共享内存、管道
- Pthread中的同步机制
- Linux的IPC机制

本周要求

- 重点阅读教材

  第2章相关内容：2.3.6~2.3.9

- 重点概念

管程　Hoare管程　Mesa管程
条件变量　wait/signal
Pthread中的互斥锁与条件变量
共享内存　消息传递　管道
Linux：原子操作　屏障　读写锁

THANKS

The End