第三章 进程同步与通信

授课教师：邢欣来

目录

CONTENTS

01. 进程的同步与互斥

进程的同步与互斥：同步与互斥的引入

- 并发带来的问题
  - 程序执行出现不可再现性
    - 全局变量的共享充满了危险
  - 操作系统很难最佳地管理资源的分配
    - 资源的竞争可能会导致死锁问题
  - 存在不可再现性和不确定性
    - 定位程序的错误是很困难的

进程的同步与互斥：同步与互斥的引入

- 由于进程的异步性，可能会导致程序执行结果的不确定性，使程序执行时出现***不可再现性***、同时在资源使用上***存在竞争***

- 进程互斥与同步的主要任务是使并发执行的诸进程之间能有效地***共享资源***和***相互合作***，从而使程序的执行***具有可再现性***

进程的同步与互斥

- 进程同步与互斥的基本概念
  - 同步：指多个进程中发生的事件存在着某种时序关系，它们必须按规定时序执行，以共同完成一项任务 。
  - 互斥：多个进程不能同时使用同一资源。
  - 临界资源：某段时间内仅允许一个进程使用的资源。
  - 临界区：每个进程中访问临界资源的那段代码。

进程的同步与互斥：临界资源实例

例：P1，P2两进程共享变量
COUNT（COUNT的初值为5）

```
P1:{
  R1=COUNT;
  R1=R1+1;
  COUNT=R1; }

P2:{
  R2=COUNT;
  R2=R2+1;
  COUNT=R2; }
```

用Bernstein条件考察

R(P1) = {R1，COUNT}

W(P1) = {R1，COUNT}

R(P2) = {R2，COUNT}

W(P2) = {R2，COUNT}

$R(P1) \cap W(P2)=\{COUNT\}$

$W(P1) \cap R(P2)=\{COUNT\}$

$W(P1) \cap W(P2)=\{COUNT\}$

- P1、P2不符合Bernstein条件
- 必须对程序的执行顺序施加某种限制

进程的同步与互斥：临界资源实例

例：P1，P2两进程共享变量
COUNT（COUNT的初值为5）

```
P1:{
  R1=COUNT;
  R1=R1+1;
  COUNT=R1; }

P2:{
  R2=COUNT;
  R2=R2+1;
  COUNT=R2; }
```

分析：

1》执行顺序 P2→P1，执行结果：

P1：COUNT为7

P2：COUNT为6

2》执行顺序

1. P1：{R1=COUNT}
2. P2：{R2=COUNT}
3. P1：{R1=R1+1；COUNT=R1}
4. P2：{R2=R2=1；COUNT=R2}

执行结果：

P1：COUNT为6，

P2：COUNT为6。

进程的同步与互斥：临界资源实例

P1:{

进入区

R1=COUNT;

R1=R1+1;

COUNT=R1;

退出区

}

P2:{

进入区

R2=COUNT;

R2=R2+1;

COUNT=R2;

退出区

}

进程的同步与互斥：同步机制应遵循的准则

访问临界资源的进程描述为

```
While(1){
    进入区
    临界区
    退出区
    剩余区
}
```

- 空闲让进
  - 当无进程处于临界区时，临界资源处于空闲状态。此时允许进程进入临界区。
- 忙则等待
  - 当已有进程进入临界区时，临界资源正在被访问，其他想进入临界区的进程必须等待。
- 有限等待
  - 对于要求访问临界资源的进程，应保证在有效的时间内进入，以免进入“死等”状态。
- 让权等待
  - 当进程不能进入临界区时，应立即释放处理机，以免进程进入“忙等”。

进程的同步与互斥：互斥实现的硬件方法

- 禁止中断
  - 通过系统内核开启、禁止中断来实现
- 专用机器指令
  - TS命令
  - Swap指令
- 硬件方法的优点、缺点 (p-75)

进程的同步与互斥：互斥实现的软件方法

——单标志算法

//进程0

```
while (turn!=0)
        //什么都不做;
临界区;
turn = 1;
剩余区;
```

//进程1

```
while (turn!=1)
        //什么都不做 ‘;
临界区;
turn = 0;
剩余区;
```

- 设置公共整型变量turn，用于指示进入临界区的进程编号i(i=0,1)。使P0、P1轮流访问临界资源。
- 缺点：强制性轮流进入临界区，不能保证“**空闲让进**”。

进程的同步与互斥：互斥实现的软件方法

//进程0

```
while (flag[1])
        //什么都不做；
flag[0]=true;
临界区;
flag[0] =false;
剩余区;
```

//进程1

```
while ( flag[0])
        //什么都不做；
flag[1]=true;
临界区;
flag[1] =false;
剩余区;
```

——双标志、先检查算法

- 设置数组flag，初始时设每个元素为false，表示所有进程都未进入临界区。若flag[i]=true，表示进程进入临界区执行。
- 在每个进程进入临界区时，先查看临界资源是否被使用，若正在使用，该进程等待，否则才可进入。解决了“空闲让进”问题。
- 缺点：可能同时进入临界区，不能保证“**忙则等待**”。

进程的同步与互斥：互斥实现的软件方法

——双标志、先修改后检查算法

```
//进程0
    flag[0]=true;
    while (flag[1])
        //什么也不做;
    临界区;
    flag[0] =false;
    剩余区;
```

```
//进程1
    flag[1]=true;
    while (flag[0])
        //什么也不做;
    临界区;
    flag[1] =false ;
    剩余区;
```

- 两进程先后同时作　flag[i]=true;
- 缺点：保证了不同时进入临界区，但又可能都进不去。不能保证“有空让进”。

进程的同步与互斥：互斥实现的软件方法

——先修改、后检查、后修改算法

//进程0

```
flag[0]=true;
turn=1;
while (flag[1]) &&
(turn==1)
        //什么也不做;
临界区;
flag[0] =false ;
剩余区;
```

//进程1

```
flag[1]=true;
turn=0;
while (flag[0]) &&
(turn==0)
        //什么也不做;
临界区;
flag[1] =false ;
剩余区;
```

保证了“**空闲让进**”和“**忙则等待**”，但对于多个进程并发的情况实现就很困难了

进程的同步与互斥：互斥实现的软件方法

- 小结：
  - 以进程为参考对象进行标识，不是一个靠谱的思路
  - 进程数量多、具有动态性
  - 换个思路，针对资源进行标识是否可行呢？

进程的同步与互斥：信号量和PV操作

- 1965年，荷兰学者Dijkstra提出了信号灯机制，卓有成效地解决了进程同步问题。
- 记录型信号灯的数据结构定义

```
struct  semaphore{
   int  value;          //初值代表的是系统中的资源数量
   struct  PCB  *queue;    //进程等待队列
}
```

进程的同步与互斥：信号量和PV操作

- 信号量 semaphore s 的物理含义
  - s.value的初值表示系统中某种资源数目
  - s.value <0时，|s.value|表示等待队列的进程数

进程的同步与互斥：信号量和PV操作

- P原语

```
void  wait(semaphore  s)
{   s.value = s.value - 1;
    if  (s.value < 0)
        block(s.queue);        /* 将进程阻塞，并将其投入等待队列s.queue */
}
```

  - 用wait(s)函数来表示
  - 表示进程试图进占资源
    - 成功则继续执行
    - 失败则进入阻塞

进程的同步与互斥：信号量和PV操作

- V原语

```
void  signal(semaphore  s)
{   s.value = s.value + 1;
    if  (s.value <= 0)
        wakeup(s.queue);   /* 唤醒阻塞进程，将其从等待队列s.queue 取出，投入就绪队列*/
}
```

  - 用 signal(s)函数来表示
  - 表示进程对资源释放，若存在阻塞进程，还需要将其唤醒，进入就绪

进程的同步与互斥：信号量解决互斥操作

semaphore mutex=1;

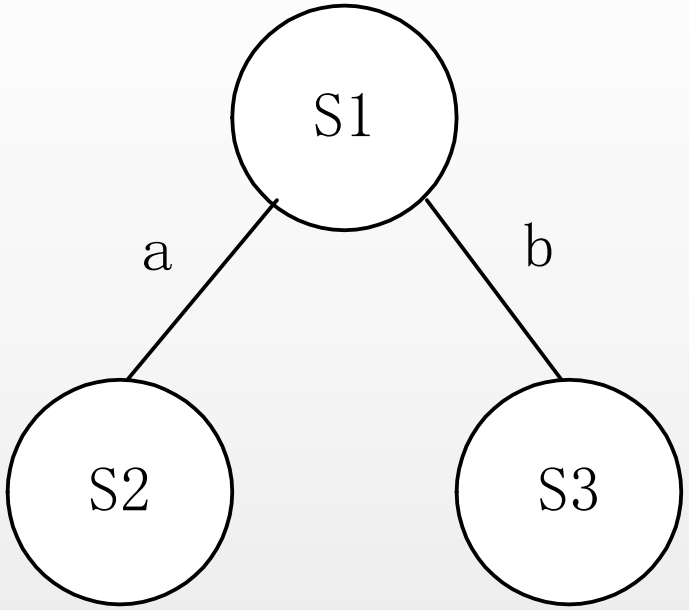
P1:

```
while (1){
    P(mutex);
    临界区;
    V(mutex);
    剩余区;
};
```

P2:

```
while (1){
    P(mutex);
    临界区
    V(mutex);
    剩余区;
};
```

进程的同步与互斥：信号量解决同步操作



semaphore a,b=0,0;

{s1; V(a); V(b)}

{P(a); s2}

{P(b); s3}

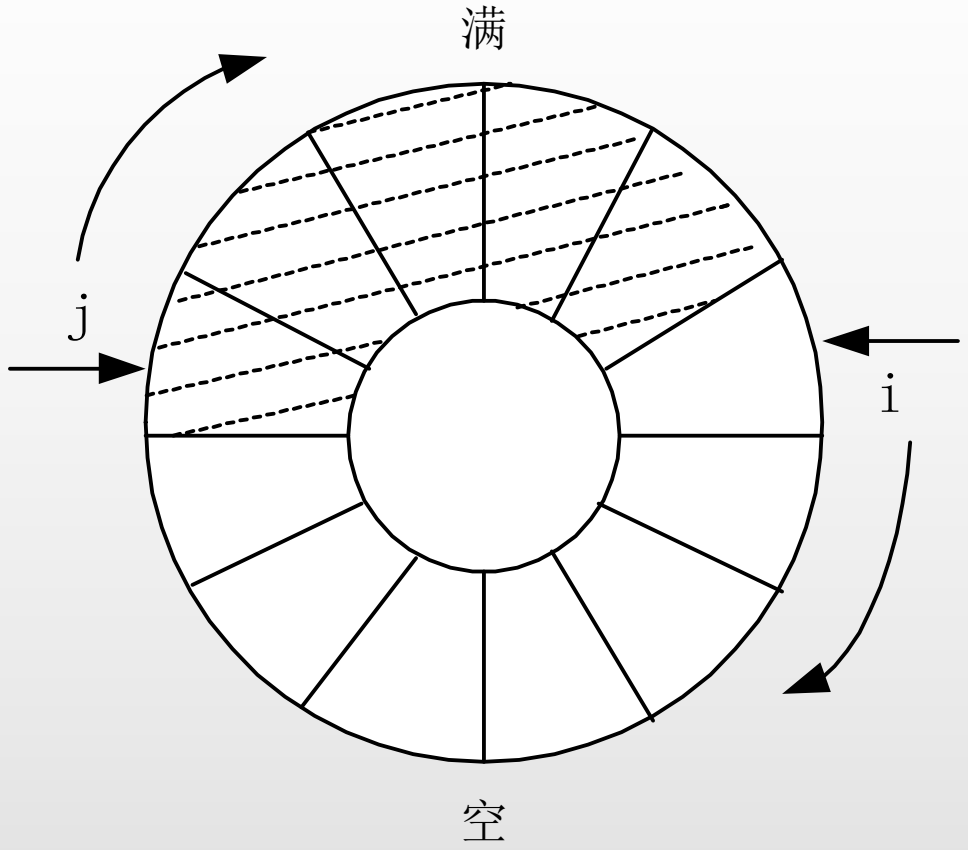
02. 经典进程同步问题

经典进程同步问题

- 生产者—消费者问题
- 读者—写者问题
- 哲学家进餐问题
- 打磕睡的理发师问题

经典进程同步问题:生产者-消费者问题

- 指有两组进程共享一个环形的缓冲池。一组进程被称为生产者，另一组进程被称为消费者。
- 缓冲池是由若干个大小相等的缓冲区组成的，每个缓冲区可以容纳一个产品。
- 生产者进程不断地将生产的产品放入缓冲池，消费者进程不断地将产品从缓冲池中取出。



经典进程同步问题：
用信号量解决“生产者-消费者”问题

```
semaphore  mutex =1;
semaphore  empty = n;
semaphore  full = 0;

int  i,j;
ITEM  buffer[n];
ITEM  data_p, data_c;
```

```
void  producer() //生产者进程
 {while  (true)
    {produce an item in data_p;
     P(empty);
     P(mutex);
     buffer[i] = data_p;
     i = (i + 1) % n;
     V(mutex);
     V(full);}
  }
```

```
void consumer()//消费者进程
{while  (true)
   { P(full);
     P(mutex);
     data_c =  buffer[j];
     j = (j + 1) % n;
     V(mutex);
     V(empty);
     consume the item in data_c;}
  }
```

经典进程同步问题：读者-写者问题

- 一个数据对象若被多个并发进程所共享，且其中一些进程只要求读该数据对象的内容，而另一些进程则要求写操作，对此，我们把只想读的进程称为“读者”，而把要求写的进程称为“写者”。
- 问题描述：
  - 读者可同时读；
  - 读者读时，写者不可写；
  - 写者写时，其他的读者、写者均不可进入。

经典进程同步问题：用信号量解决读者-写者问题

Semaphore Wmutex=1;

```
void  reader()    /*读者进程*/
{
    while  (true) {
        P(Wmutex);
        read;      /* 执行读操作 */
        V(Wmutex);
    }
}
```

```
void  writer()  /*写者进程*/
{
    while  (true){
        P(Wmutex);
        write; /* 执行写操作 */
        V(Wmutex);
    }
}
```

经典进程同步问题：用信号量解决读者-写者问题

Semaphore Wmutex=1;

Semaphore Rmutex=1;

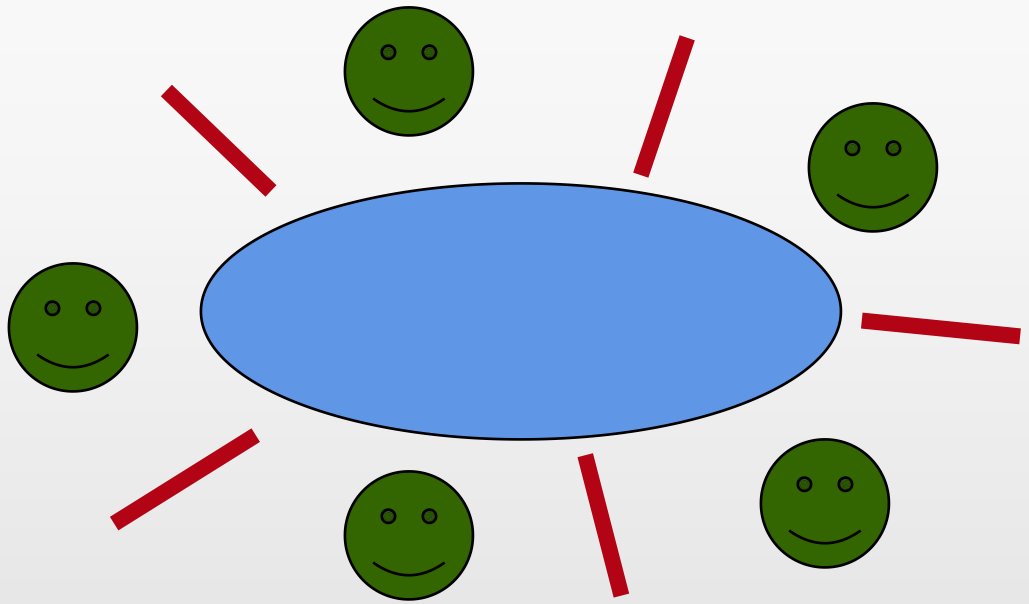
int Rcount;//记录读者的在线数量

```
void  reader()    /*读者进程*/
{while  (true) {
    P(Rmutex);
    if  (Rcount == 0)    P(Wmutex);
    Rcount = Rcount + 1;
    V(Rmutex);
    read;      /* 执行读操作 */
    P(Rmutex);
    Rcount = Rcount - 1;
    if  (Rcount == 0)    V(Wmutex);
    V(Rmutex);
}}
```

```
void  writer()  /*写者进程*/
{
   while  (true){
      P(Wmutex);
      write; /* 执行写操作 */
      V(Wmutex);
   }
}
```

经典进程同步问题：哲学家进餐问题

- 五个哲学家，他们的生活方式是交替地思考和进餐。
- 哲学家们共用一张圆桌，围绕着圆桌而坐，在圆桌上有五个碗和五支筷子，平时哲学家进行思考，饥饿时拿起其左、右的两支筷子，试图进餐，进餐完毕又进行思考。
- 这里的问题是哲学家只有拿到靠近他的两支筷子才能进餐，而拿到两支筷子的条件是他的左、右邻居此时都没有进餐。



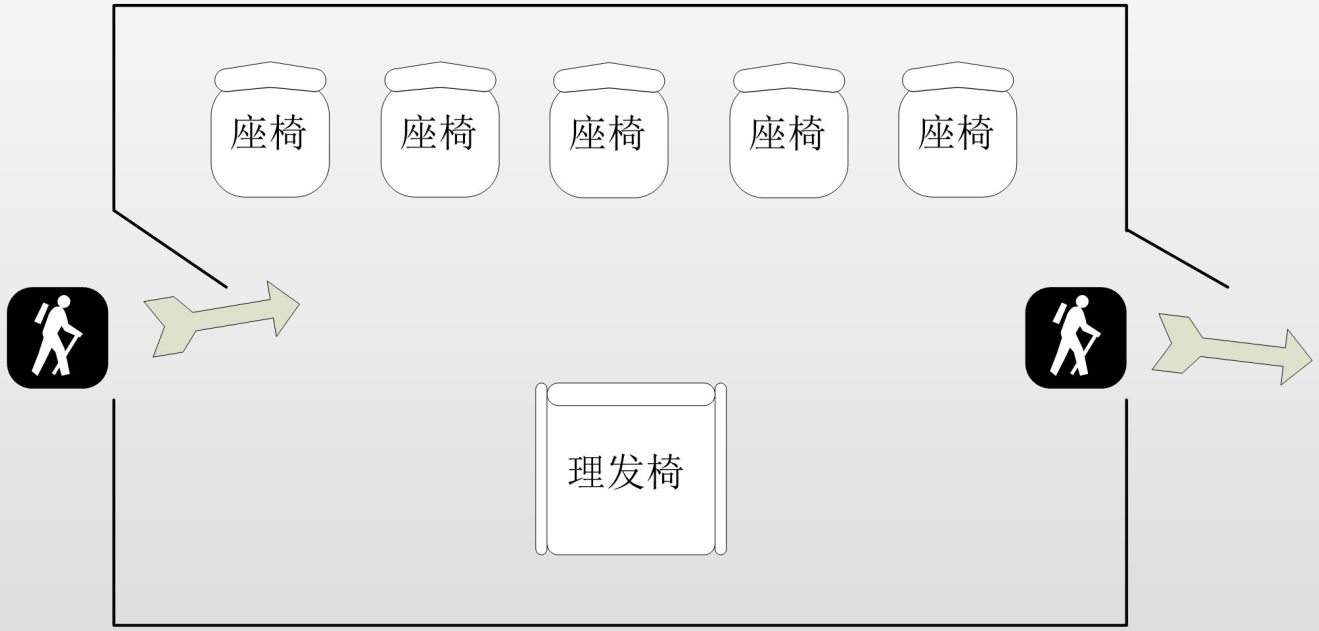
经典进程同步问题：用信号量解决哲学家进餐问题

```
semaphore  chopstick[5] ={1,1,1,1,1};
void  philosopher (int i )    /*哲学家进程*/
{while  (true)
    { P(chopstick[i]);
      P(chopstick[(i + 1) % 5]);
      eating;      /* 进餐 */
     V(chopstick[i]);
     V(chopstick[(i + 1) % 5]);
      thinking;      /* 思考 */}
}
```

- 潜在问题
  - 当五名哲学家同时拿起左边的筷子时，却都没办法拿到右边的筷子，会导致死锁

经典进程同步问题：打瞌睡的理发师问题

- 理发店有一名理发师，一把理发椅，还有N把供等候理发的顾客坐的普通椅子。如果没有顾客到来，理发师就坐在理发椅上打磕睡。当顾客到来时，就唤醒理发师。如果顾客到来时理发师正在理发，顾客就坐下来等待。如果N把椅子都坐满了，顾客就离开该理发店去别处理发。



经典进程同步问题：用信号量解决打磕睡的理发师问题

```
#define CHAIRS 5
//为等候的顾客准备的座椅数

semaphore customers = 0;

semaphore barners = 0;

semaphore mutex = 1;

int waiting;
```

```
void barber()        //理发师进程
{
   while  (true)
  {P(customers);
//如果没有顾客，理发师就打磕睡
    P(mutex);//互斥进入临界区
    waiting--;
    V(barners);//理发师准备理发了
    V(mutex);
    cut_hair();//理发
  }
}
```

```
void customer ()            //顾客进程
{P(mutex);//每次只能有一位顾客进门
if (waiting < CHAIRS)//如果有空位，顾客等待
    {waiting++;
    //如果有必要，唤醒理发师
    V(customers);
    V(mutex);
    P(barners);  //如果理发师正在理发，则顾客等待
    get_haircut();            }
else      //如果没有空位，则顾客离开
 V(mutex);
}
```

03. AND信号量

AND信号量

- 用信号量解决了很多同步和互斥问题，但在解决问题的过程中，我们也发现还存在一些问题，如：
  - 在生产者和消费者问题中两个P操作的位置不能颠倒
  - 哲学家进餐问题中的死锁现象等
  - p-85

这些问题的出现促使AND信号量的产生

AND信号量：基本思想

- 将进程在整个运行期间所需要的所有临界资源一次性全部分配给进程，待该进程使用完成后再一起释放。
- 只要尚有一个资源不能满足进程要求，其他所有能分配给该进程的资源也都不予分配
- P操作的原语为Swait，V操作的原语为Ssignal

AND信号量：解决哲学家进餐问题

```
semaphore  chopstick[5] ={1,1,1,1,1};
void  philosopher (int i )    /*哲学家进程*/
{  while  (true)
    {
        Swait（chopstick[i]，chopstick[(i + 1) % 5]);
        eating;      /* 进餐 */
        Ssignal(chopstick[i]，chopstick[(i + 1) % 5]);
        thinking;      /* 思考 */
    }
}
```

AND信号量：解决生产者-消费者问题

```
semaphore  mutex =1;
semaphore  empty = n;
semaphore  full = 0;

int  i,j;
ITEM  buffer[n];
ITEM  data_p, data_c;
```

```
void  producer() //生产者进程
 {while  (true)
    {produce an item in data_p;
     Swait(empty,mutex);
     buffer[i] = data_p;
     i = (i + 1) % n;
     Ssignal(mutex,full);}
 }
```

```
void consumer()//消费者进程
{while  (true)
  { Swait(full,mutex);
    data_c =  buffer[j];
    j = (j + 1) % n;
    Ssignal(mutex,empty);
    consume the item in data_c;}
  }
```

04. 管程

管程：引入的原因

- 信号灯机制虽然既方便又有效地解决了进程同步问题，但要求访问临界资源的进程自备同步操作wait(s)、signal(s)，使得大量的同步操作分散在各个进程中，给进程的管理带来不便，并会因同步操作使用不当导致死锁。
- 管程的基本思想是把信号量及其操作原语封装在一个对象内部，即将共享资源以及针对共享资源的所有操作集中在一个模块中。

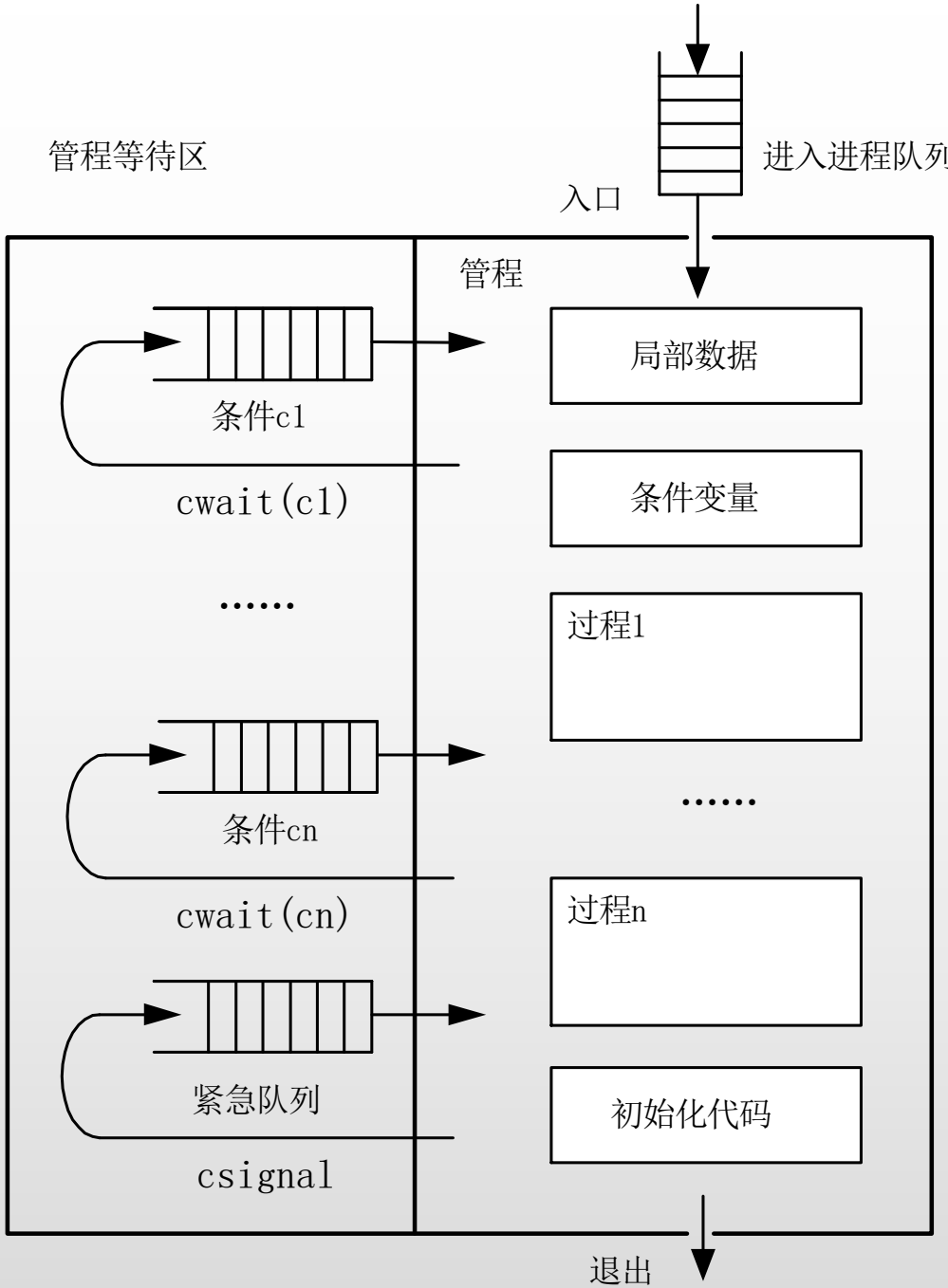
管程：管程的定义

- 一个共享资源的数据结构
- 以及一组能为并发进程在其上执行的针对该资源的一组操作
- 这组操作能同步进程和改变进程中的数据
- 管程 = 数据结构+操作+对数据结构中变量的初始化

管程：管程模型

- 如何解决互斥
  - 进程在操作共享变量之前，增加一个等待队列，每一个进程想要操作共享变量的话，都需要在等待队列中等待，直到管程选出一个进程操作共享变量。每次只能有一个进程进入管程。
- 如何解决同步
  - 在操作共享变量时候，它不一定是直接执行，可能有一些自己的执行条件限制（比如消费者一定要有产品才能消费，生产者一定要的空余的空间才能继续生产），我们将这些限制称之为条件变量
    - cwait（C）：将进程挂起再条件C上。
    - csignal（C）：恢复再cwait上因为某些条件而挂起的进程。
  - 每一个条件变量也有自己对应的等待队列，当线程发现自己的条件变量不满足时，就进入相应的等待队列中排队。即便条件变量满足，其等待队列中的线程也不会是立马执行，而是到最开始共享变量对应的等待队列中再次排队，重复之前的过程。

管程：管程模型


管程等待区
进入进程队列
入口
管程
局部数据
条件c1
cwait(c1)
条件变量
.....
过程1
条件cn
.....
cwait(cn)
过程n
紧急队列
初始化代码
csignal
退出

管程：管程的特征

- 资源保护：局限于管程的共享变量（数据结构）只能被管程的过程访问，任何外部过程都不能访问
- 访问方式：一个进程通过调用管程的一个过程进入管程
- 解决互斥：任何时候只能有一个进程在管程中执行，调用管程的任何其他进程都被挂起，以等待管程变为可用，即管程有效的实现互斥

管程：生产者-消费者问题

```
monitor  monitor_PC;//管程对象
char  buffer[n];
int  nextin, nextout; //下一个存取位置
int  count; //记录产品的数量
condition  notfull, notempty;   //条件变量

void  put(char  x);     /*过程*/
{
   if (count == n) cwait(notfull);
   buffer[nextin] = x;
   nextin = (nextin + 1) % n;
   count = count + 1;
   csignal(notempty);
}
```

```
void  get(char  x);     /*过程*/
{
  if (count == 0)  cwait(notempty);
      x = buffer[nextout];
      nextout = (nextout + 1) % n;
      count = count - 1;
      csignal(notfull);
}
{             /*管程体*/
       nextin = 0;
       nextout = 0;
       count = 0;   /*变量初始化*/
}
```

管程：生产者-消费者问题

```
void producer()      /* 生产者进程 */
{
    char  x;
    while (true)
    {
        produce an char in x;
        monitor_PC.put(x);
    }
}
```

```
void consumer() /* 消费者进程 */
{
    char  x;
    while (true)
    {
        monitor_PC.get(x);
        consume an x;
    }
}
```

05. 进程通信

进程通信:引言

- 目的：解决进程间的数据交流
- 低级通信：信号量作为进程同步和互斥工具是卓有成效的。但作为通信工具就不够理想。其原因为:
  - 效率低——一次只传一条消息
  - 通信对用户不透明
- 高级通信:
  - 传送任意数量的数据
  - 操作系统隐藏了进程通信的实现细节，用户透明

进程通信：高级通信机制分类

- 共享存储系统：进程之间通过共享存储区域进行通信
  - 通信前向系统申请共享存储区域
  - 进程将申请获得的共享区域链接在本进程上
  - 通过对存储区域的读写操作，实现大量信息的传递
- 进程对于存储空间的共享是互斥的，操作系统提供互斥工具

进程通信：高级通信机制分类

- 消息传递系统：通过操作系统提供的一组消息通信原语来实现信息的传递
  - 直接通信方式
    - 发送方--->接收方
  - 间接通信方式
    - 发送方--->消息队列--->接收方
    - 一对一、多对一、一对多、多对多

进程通信：高级通信机制分类

- 管道通信
  - 定义：管道指的是用于连接读、写进程的一个共享文件（内存中开辟的一个大小固定的缓冲区）
  - 互斥：当一个进程对管道进行读或写操作时，另一个进程必须等待
  - 同步：数据以字符流的形式写入管道，管道写满时，write()阻塞，管道为空时，read()阻塞
  - 只有确认对方存在，方能采用管道通信

进程通信：进程通信中的问题

- 通信链路的建立方式
  - 显式建立链路：常见于网络间进程通信
    - “建立连接”的显式命令建立通信链路
    - “拆除链接”的显示命令拆除通信链路
  - 隐式建立链路：常见于本机内进程通信
    - 发送进程不必明确提出建立链路的请求
    - 操作系统自动为之建立一条通信链路

进程通信：进程通信中的问题

- 通信方向
  - 单向通信方式：发送进程--->接收进程
  - 双向通信方式：发送进程<--->接受进程
- 通信链接方式
  - 点对点连接方式：一条通信链路连接两个进程进行通信
  - 广播方式：一条链路上接入多个（>2）进程，一个进程向其他多个进程同时发送消息

进程通信：进程通信中的问题

- 通信链路的容量
  - 通信链路上是否包含用于暂存数据的缓冲区
    - 无：不能暂存消息
    - 有：可以暂存消息
    - 缓冲区数目越大，通信链路的容量越大

进程通信：进程通信中的问题

- 数据格式
  - 字节流
    - 无具体的格式
    - 接受方不需要保留歌词发送之间的分界
  - 报文
    - 报头：发送进程名、报文长度、发送日期等
    - 正文：具体发送的信息
    - 分定长报文和不定长报文

进程通信：进程通信中的问题

- 同步方式
  - 阻塞方式：发送进程发送消息后阻塞，等待接收进程收到消息后被唤醒
  - 非阻塞方式：发送进程发送消息后，无需等待，继续执行后续任务

进程通信：消息传递系统的实现

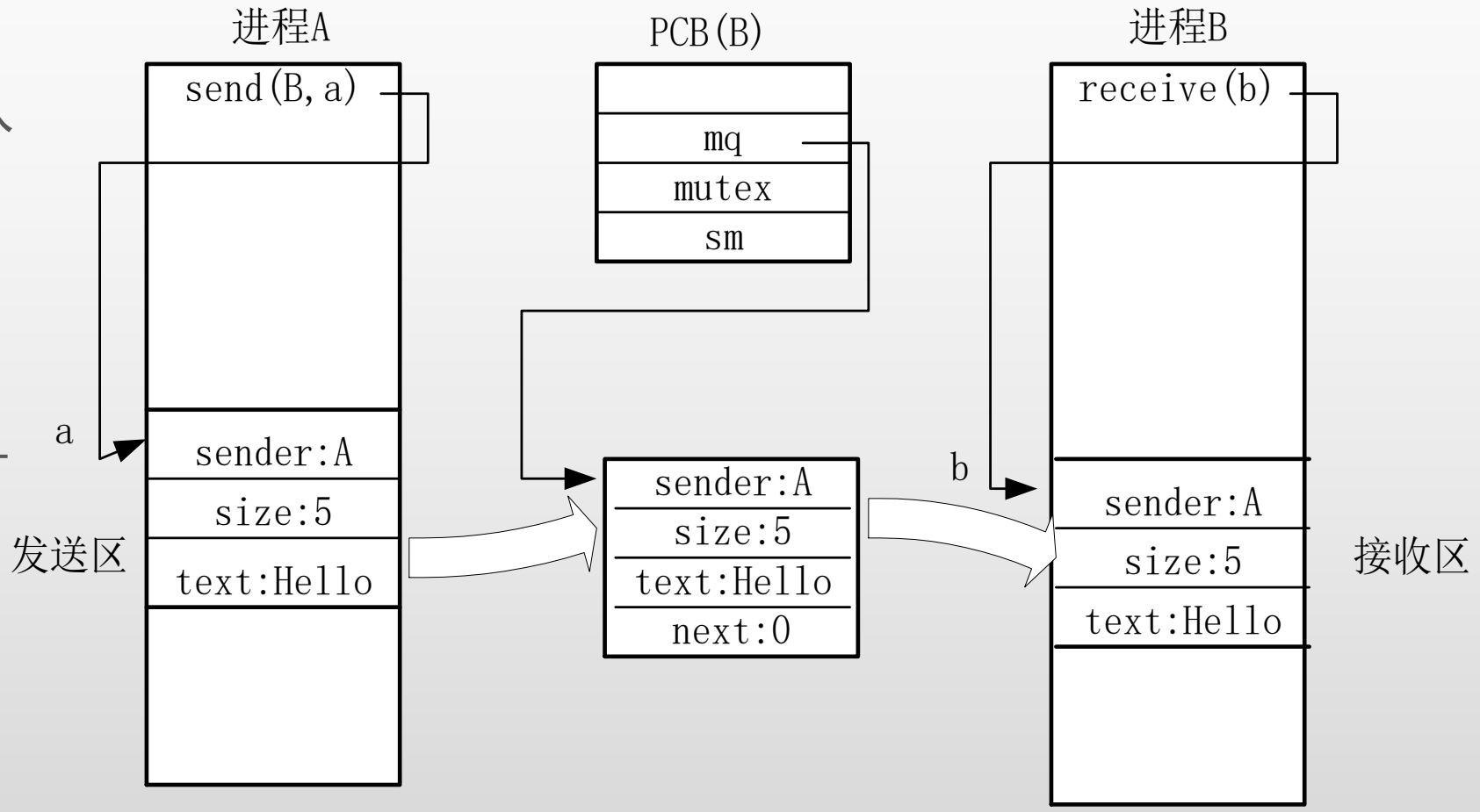
1.在该进程内存开辟一个发送区i，将发送进程标识符，报文长度，正文等内容填入其中

2.获得接收进程的PCB j

3.将i挂到消息队列j.mq上

4.唤醒接收进程，通知其可以接收消息了

消息缓冲队列示意图



进程通信：消息缓冲队列-数据结构定义

```
//消息缓冲区定义
struct   message_buffer
{ char  sender[30];                /*发送进程标识符*/
   int  size;                      /*消息长度*/
   char  text[200];                /*消息正文*/
   struct  message_buffer  *next;          //指向下一个消息缓冲区的指针
}

//PCB中有关通信的数据项
struct   process_control
{  struct  message_buffer  *mq;    /*消息队列队首指针*/
    semaphore  mutex=1;            /*消息队列互斥信号量，初值为1*/
    semaphore  sm=0;    /*消息队列同步信号量，记录消息的个数.初值为0*/
}
```

进程通信：消息缓冲队列-发送原语

```
//发送原语
char receiver[30]; struct message_buffer a;
void send(receiver, a)
{   struct message_buffer i;
    struct process_control j;
    getbuf(a.size, i);          /*发送区a消息的长度申请一缓冲区i*/
    i.sender = a.sender;      i.size = a.size;
    i.text = a.text;       i.next = NULL;
    getid(PCB_set, receiver, j);  /*获得接收进程的进程标识符j*/
    P(j.mutex);
    Insert(j.mq, i);       /*将消息缓冲区i挂到的消息队列j.mq上*/
    V(j.mutex);
    V(j.sm);
}
```

进程通信：消息缓冲队列-接收原语

```
//接收原语
struct  message_buffer  b;
void  receive(b)
{  struct  message_buffer  i;
   struct  process_control  j;
   j = internal_name();        /*接收进程的内部标识符*/
   P(j.sm);
   P(j.mutex);
   remove(j.mq, i); /*从消息队列中摘下第一个消息缓冲区*/
   V(j.mutex);
   b.sender = i.sender;
   b.size = i.size;
   b.text = i.text;
}
```

进程通信：客户端-服务器系统通信

- 常用的通信方式：
  - 命名管道
  - 套接字
  - 远程过程调用