第二章　进程管理

重点

- 第一节　前趋图和程序执行
- 第二节　进程的描述
- 第三节　进程控制
- 第四节　进程同步
- 第五节　经典进程的同步问题
- 第六节　进程通信
- 第七节　线程

第一节 前趋图和程序执行

- 前趋图
- 程序顺序执行
- 程序并发执行

- **单道程序系统中，程序是顺序执行的：**

  必须在一个程序执行完后，才允许另一个程序执行

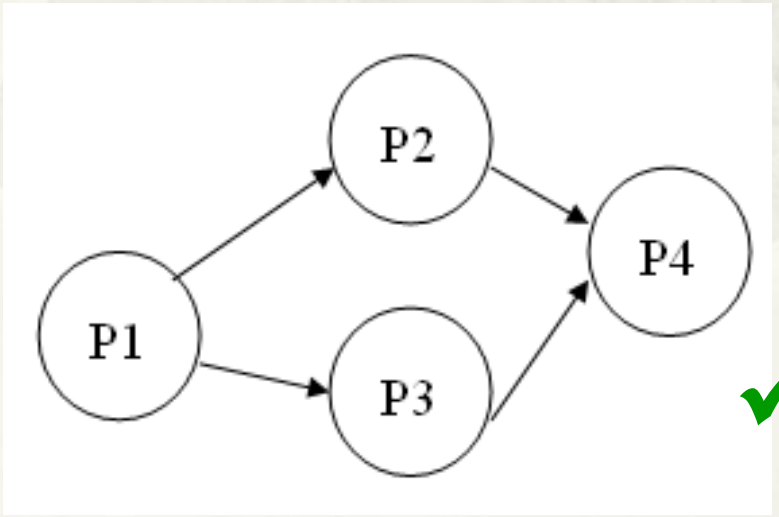
- **多道程序系统中，多个程序并发执行：**

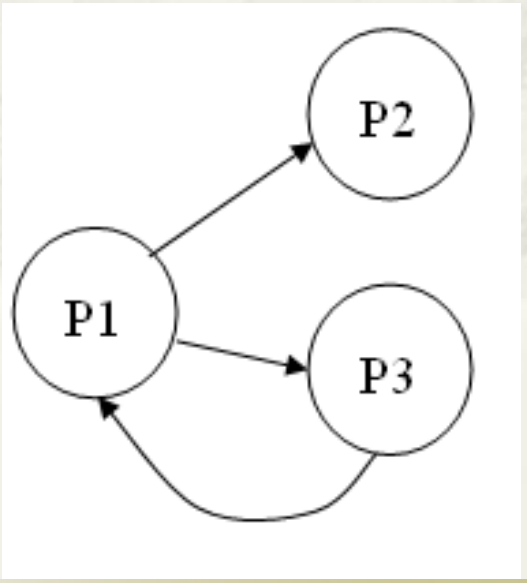
  允许暂停一个程序的执行，而转去执行另一个程序

- 程序的这两种执行方式间有着显著的不同

前趋图

- 前趋图：有向无环图(DAG)
  - 作用：描述进程间执行的先后顺序。
  - 结点：一个进程/程序段，或一条语句
  - 有向边：结点之间的前趋(偏序)关系





程序顺序执行

程序顺序执行的特征：

- **顺序性**

  每一个操作必须在下一个操作开始之前结束

- **封闭性**

  程序运行时独占全机资源，资源的状态只有本程序才能改变

- **可再现性**

  只要初始条件和运行环境相同，当程序重复执行时，都将获得相同的结果。

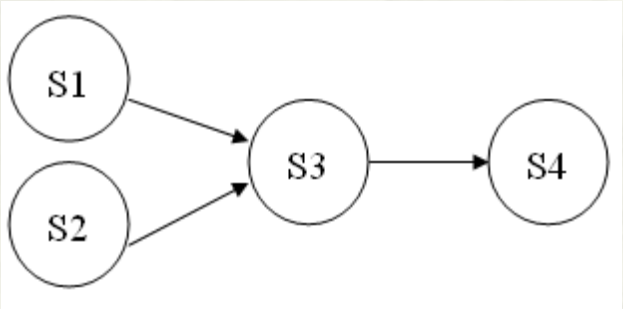
前趋图的例子：

S1：a = x + 2;

S2：b = y + 4;

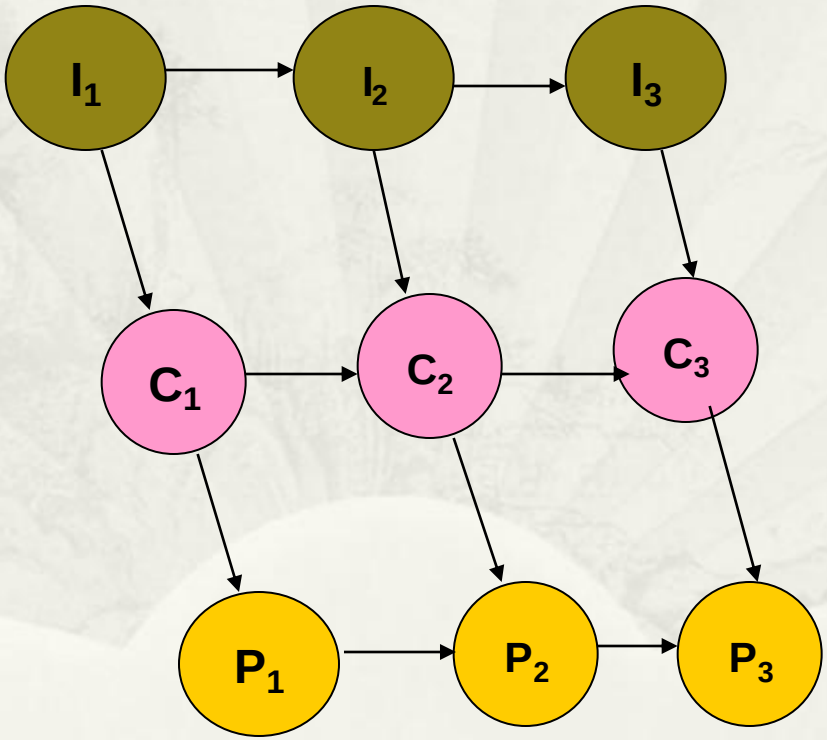
S3：c = a + b;

S4：d = c + b;



程序并发执行

- 并发执行时的前趋图
  - 有三个程序：I=输入，C=计算，P=打印



- 并发执行时的特征
  - 间断性：“停停走走”
  - 失去封闭性：因为多个程序共享资源
  - 不可再现性

例如:有两个循环程序A和B，共享一个变量n。程序A每执行一次时，都要做n=n+1;程序B每执行一次时，都要执行Print(n)；然后将n置成0。程序A和B并发执行时，可能出现的情况如下：

1、n=n+1在print(n)和n=0之前，得到的n值为n+1,n+1,0

2、n=n+1在print(n)和n=0之后，得到的n值为n,0,1

3、n=n+1在print(n)和n=0之间，得到的n值为n,n+1,0

程序顺序执行与并发执行的比较

	顺序执行	并发执行
执行过程	顺序执行	交替执行
封闭性	独占资源 有封闭性	共享资源 无封闭性
可再现性	√	x
程序间关系	无关系	间接制约或直接制约

- 多道程序系统中，程序并发执行，会失去封闭性，出现不可再现性，导致程序执行可能出现错误。
- 所以，通常意义下的程序不能参与并发执行。

第二节 进程的描述

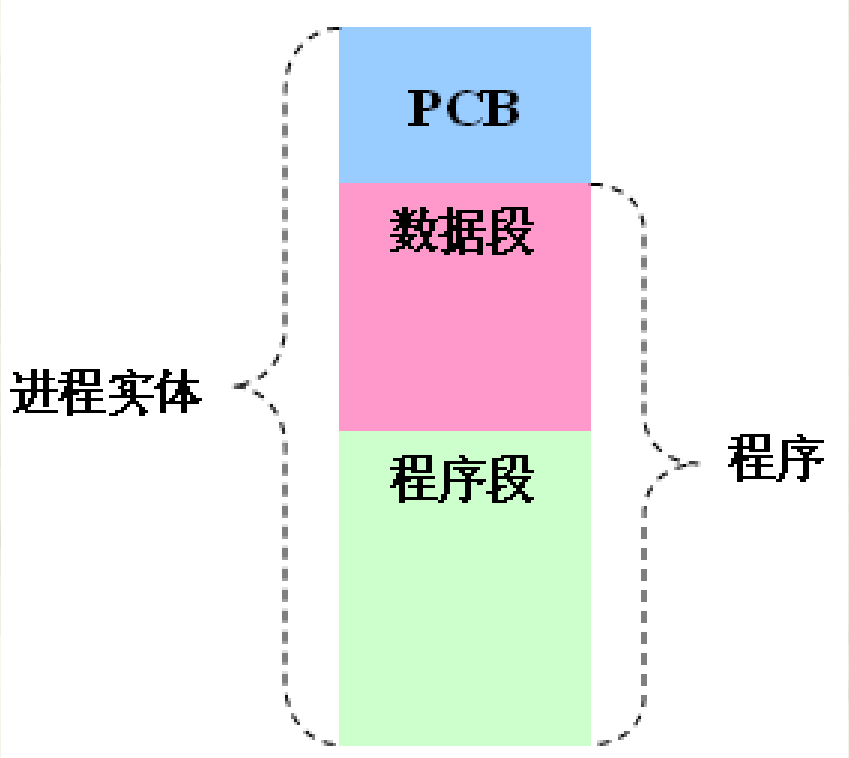
- 进程的定义和特征
- 进程的基本状态
- 进程的挂起状态
- 进程管理中的数据结构

进程的定义和特征

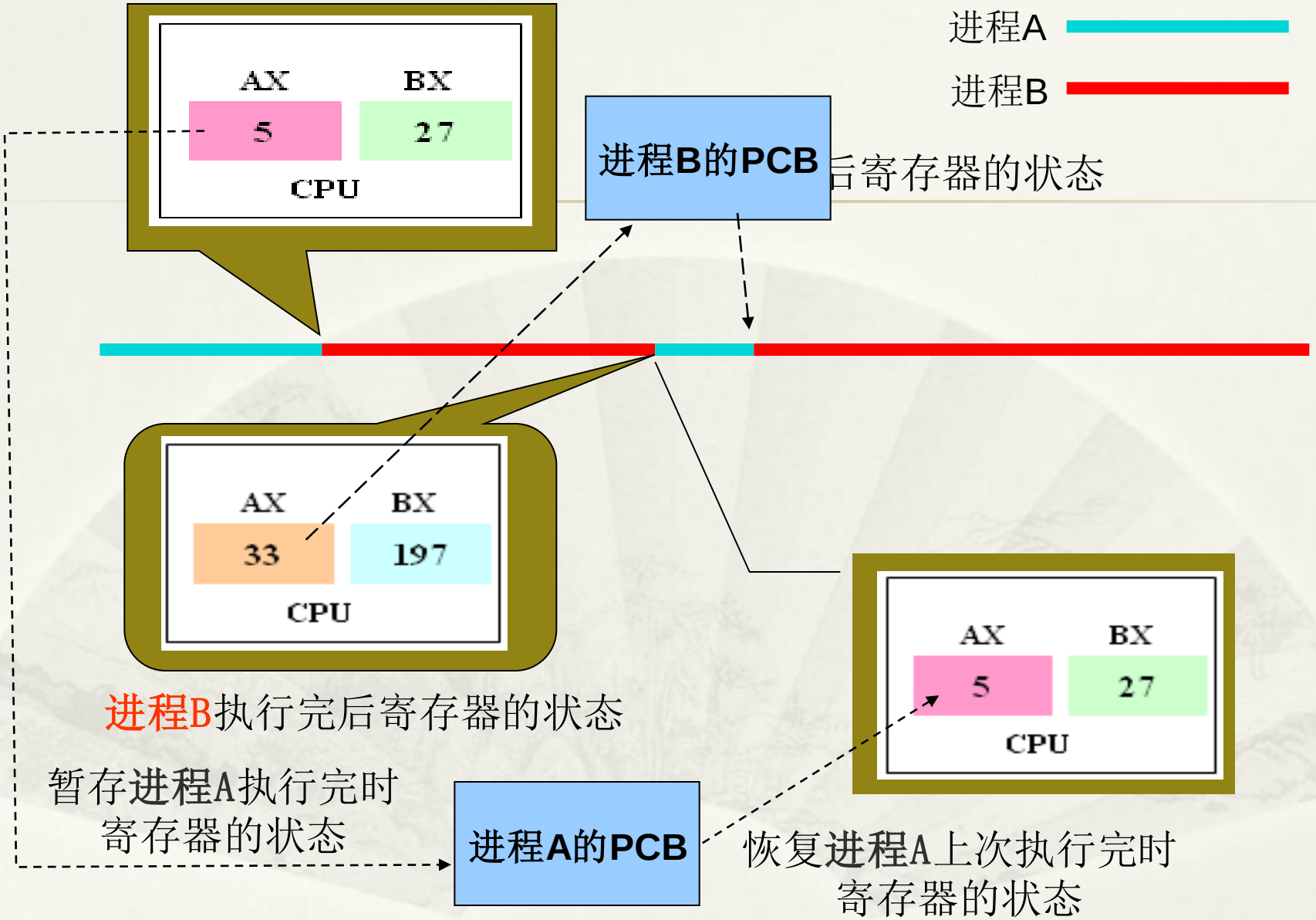
- 为使程序能正确地并发执行，且对并发执行进行描述和控制。
- 以“程序”为基础，又引入“进程”这一新的概念。

进程的结构特征

进程实体 = 程序 + 进程控制块(PCB)



Windows 中的进程


进程A
进程B
AX
BX
5
27
CPU
进程B的PCB
后寄存器的状态
AX
BX
33
197
CPU
进程B执行完后寄存器的状态
暂存进程A执行完时
寄存器的状态
进程A的PCB
AX
BX
5
27
CPU
恢复进程A上次执行完时
寄存器的状态

进程控制块

- PCB(Process Control Block)
  - 记录了OS用于描述进程的当前情况以及控制进程运行的全部信息。
  - 进程存在的唯一标志。
  - 作用：使一个在多道程序系统中不能独立运行的程序，成为一个能独立运行且能并发执行的进程。
  - 位置： OS中专门开辟的PCB区内

- **PCB中的信息**

① 进程标识符：唯一地标识一个进程

内部标识（OS）

外部标识（由创建者提供，由字母数字组成）

② 处理机状态

③ 进程调度信息

④ 进程控制信息

Unix/Linux下的进程标识符是什么样的？

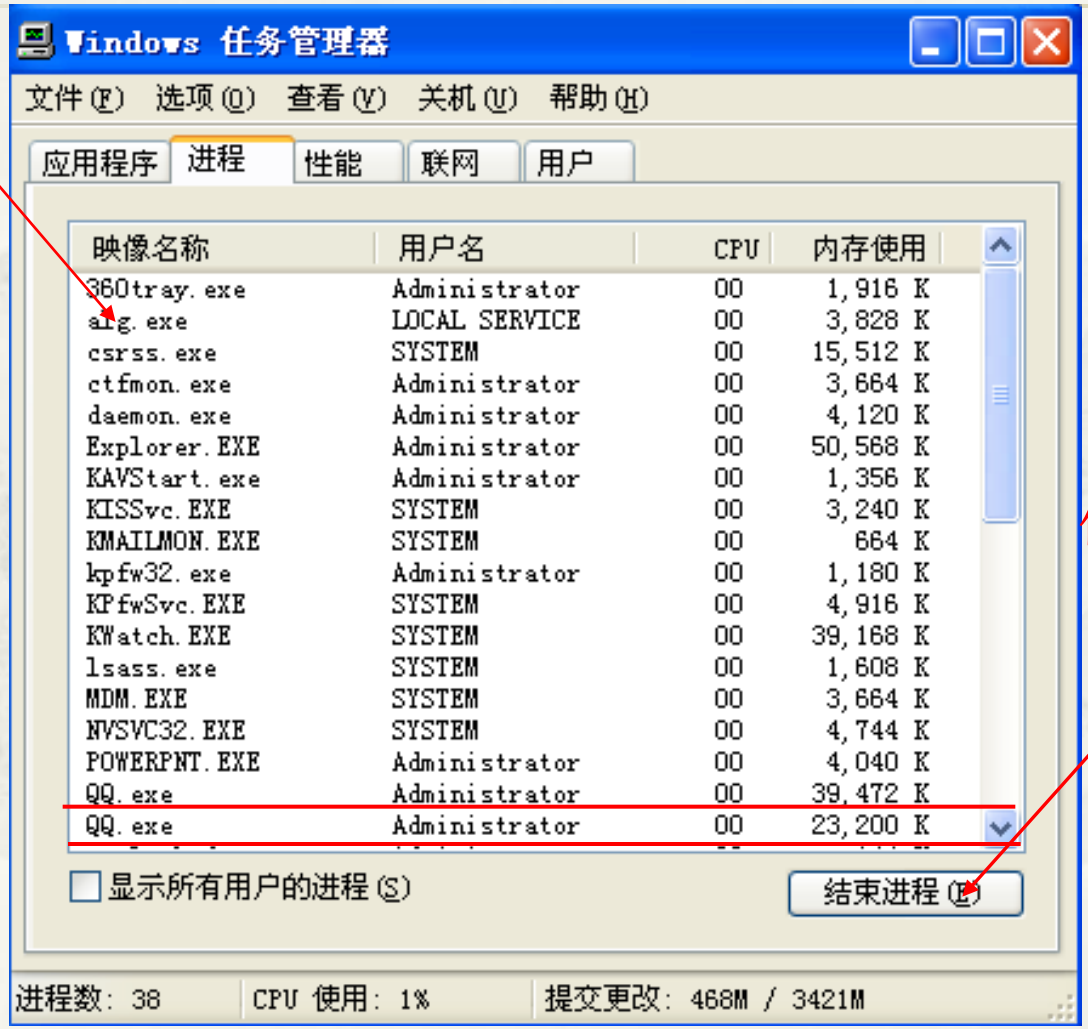
```
ps  -ef
```

kill -9 12189

PID	COMMAND	%CPU	TIME	#TH	#PRTS	#MREGS	RPRVT	RSHRD	RSIZE	VSIZE
12189	httpd	0.0%	0:00.00	1	11	84	92K	2.28M	492K	31.9M
12188	top	14.7%	0:03.71	1	19	22	592K	344K	1.01M	27.0M
12186	httpd	0.0%	0:00.11	1	12	85	1.12M	2.25M	2.55M	31.9M
12184	Vienna	1.7%	0:21.90	6	127	242	50.2M+	9.97M+	55.9M+	198M+
12182	mdimport	0.0%	0:00.62	4	62	54	956K	1.98M	3.55M	39.3M
12176	ImageWell	0.0%	0:22.27	4	105	222	24.6M	7.45M	28.1M	198M
12136	bash	0.0%	0:00.07	1	14	17	132K	728K	596K	27.2M
12135	login	0.0%	0:00.03	1	16	37	0K	260K	1.34M	26.9M
12133	Terminal	0.9%	0:12.74	4	95	148	1.41M	7.34M	19.0M	143M
12131	Microsoft	0.0%	0:04.91	2	66	115	520K	3.40M	1.77M	131M
12128	Microsoft	1.7%	6:39.41	7	117	473	15.1M	13.5M	15.3M	239M
9665	Adium	0.8%	4:23.24	9	347	450	13.0M	9.84M	17.6M	212M
272	InstantSho	3.2%	0:03.21	2	154	181	1.87M	17.3M	15.0M	159M
267	Safari	0.4%	12:24.84	10	203	1523	105M	15.6M	69.2M	425M
266	cupsd	0.0%	0:00.20	2	30	26	40K	540K	848K	27.8M
237	iStat	5.4%	50:08.89	6	989	188	4.23M	6.31M	8.33M	155M

* Windows下的进程标识符是什么样的？

外部标识符



使用“内部标识符”控制

进程的特征

- **动态性**
  - 进程实质是进程实体的一次执行过程

    体现在“由创建而生，由调度而执行，由撤销而亡”
- **并发性**
  - 多个进程实体同存于内存中，并发执行
  - 程序(没建立PCB)是无法并发执行的。

- **独立性**
  - 进程实体是一个独立运行、独立分配资源、独立接受调度的基本单位。
  - 凡未建立PCB的程序都不能作为一个独立单位参与运行。
- **异步性**
  - 进程按各自独立的、不可预知的速度执行(即按异步方式运行)。

- **进程是一个动态的概念**

① 程序的一次执行。

② 一个程序及其数据在处理机上顺序执行时所发生的活动。

③ 程序在一个数据集合上运行的过程，是系统进行资源分配和调度的一个独立单位。

- **进程的定义**

  进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位。

- 通常，也把“进程实体”简称为“进程”

- **小结：对“进程”这一概念如何理解**

① 动态的

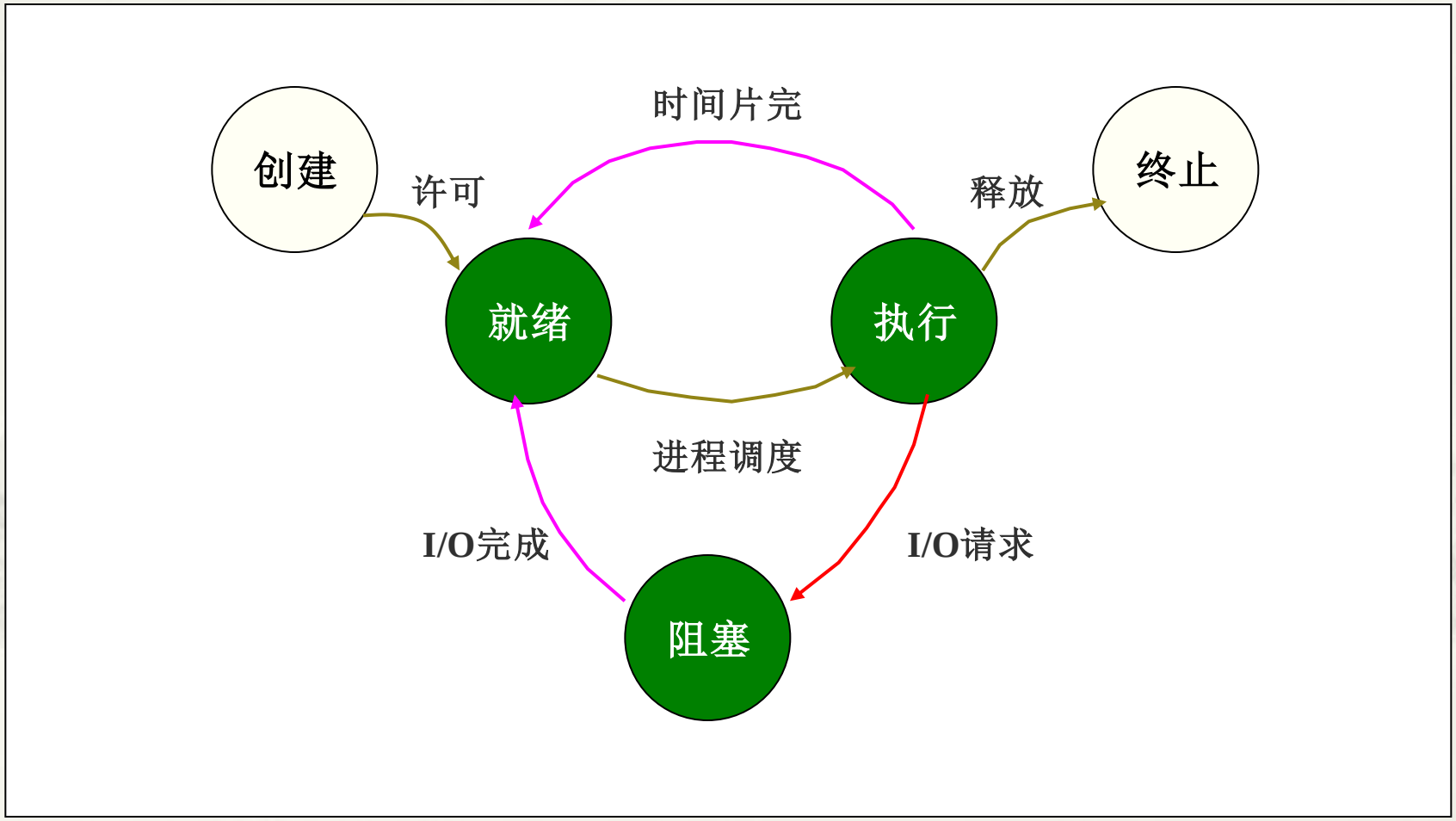
② 组成

- 给程序附加上PCB，形成了进程实体

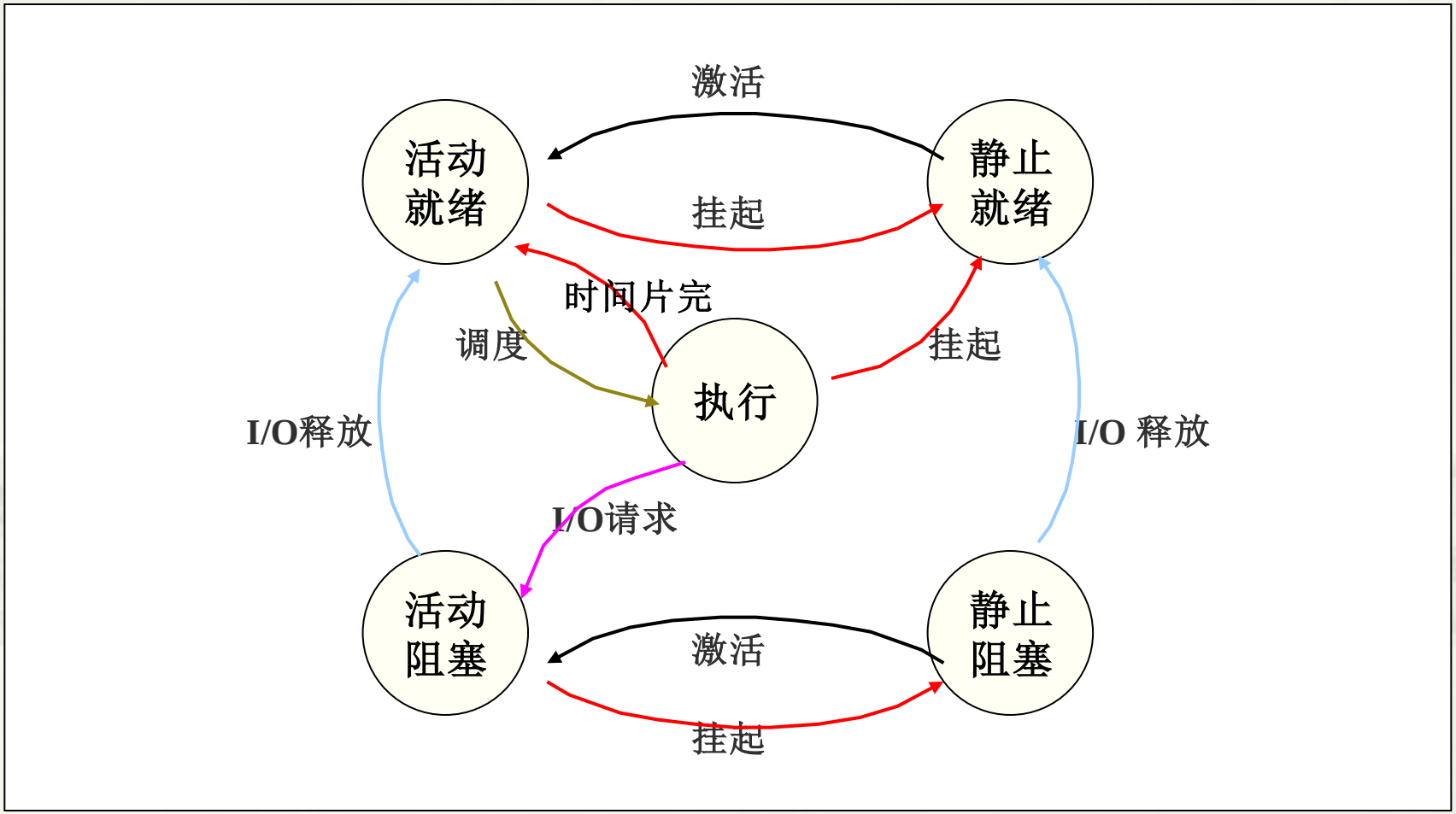
③ PCB的作用

- 保存执行过程中的一些信息，再次执行时恢复这些信息，以保证执行的正确性。
- 也可用于OS控制和描述进程的执行。

进程的基本状态



进程的挂起状态

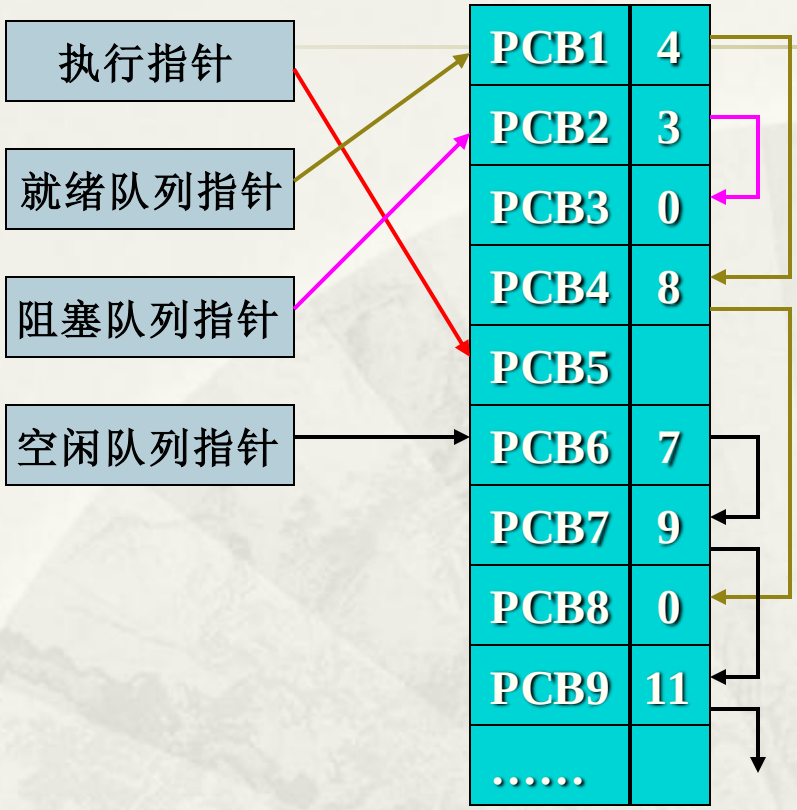


进程控制块的组织方式

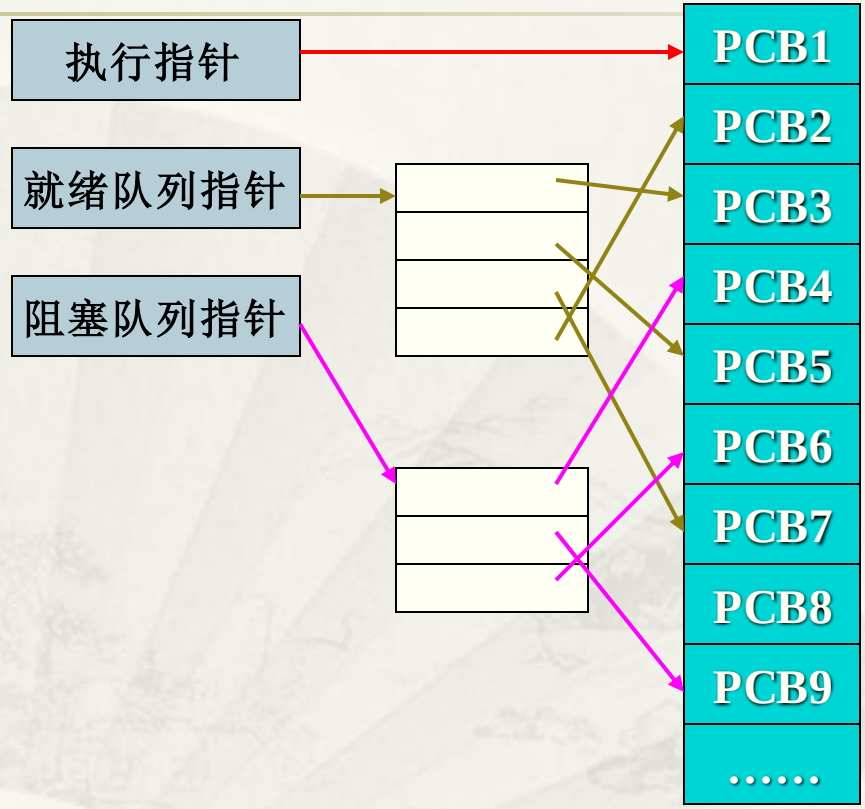
- PCB数目

  一个系统中的PCB数目可为数十个、数百个甚至数千个

- 线性方式
  - 所有PCB都组织在一张线性表里
  - 简单、开销小，但检索时浪费时间
- 链接方式
  - 把具有同一状态的PCB，链接成一个队列
  - 就绪队列、若干个阻塞队列、空队列
- 索引方式
  - 系统根据所有进程的状态建立相应的索引表



PCB链接队列示意图



按索引方式组织PCB

第三节 进程控制

- 操作系统内核
- 进程创建
- 进程终止
- 进程阻塞和唤醒
- 进程挂起与激活

由OS调用
原语完成

操作系统内核

- 操作系统内核
  - 与硬件有关的模块
  - 各种常用设备的驱动程序
  - 运行频率较高的模块（时钟管理、进程调度等）
  - 常驻内存
- 优点：
  ① 便于保护这些软件，防止遭其他程序破坏
  ② 提高OS的运行效率

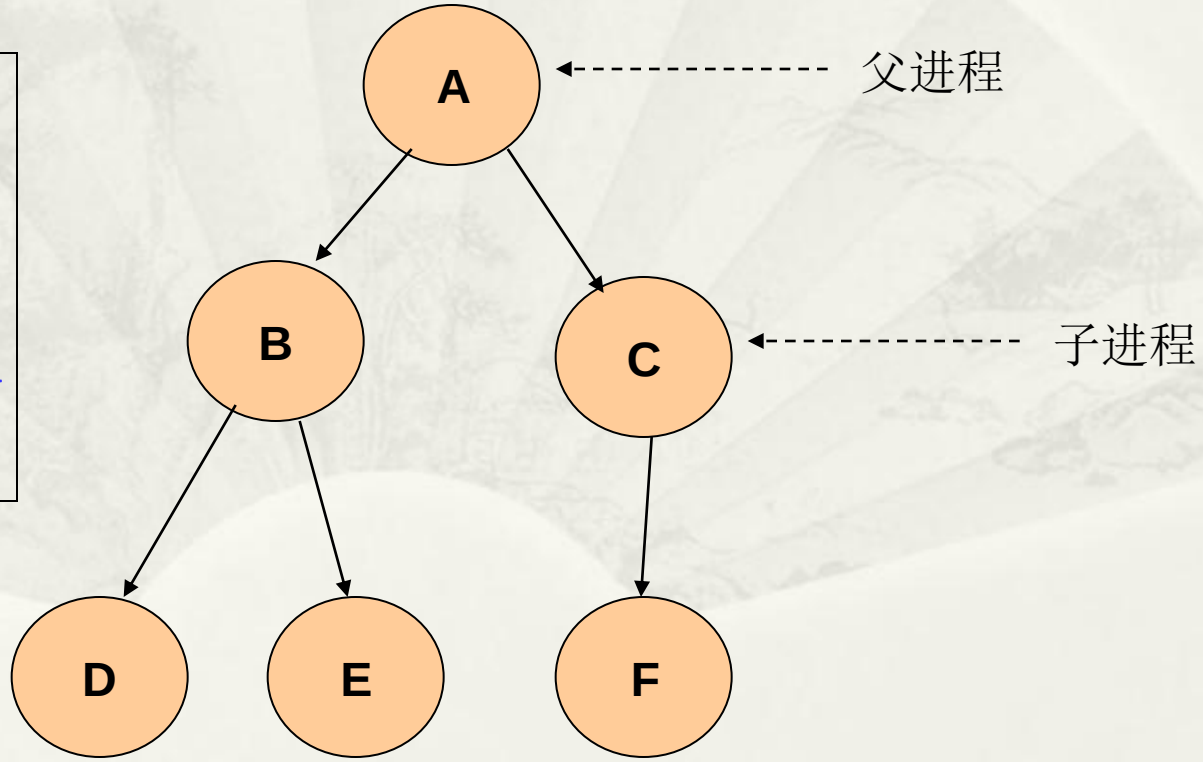
- 为防止OS遭破坏，通常将CPU的执行状态分为两种：
  - 系统态（管态、内核态）：能执行一切指令，访问所有寄存器和存储区
  - 用户态（目态）：具有较低权限，仅能执行规定的指令，访问指定的寄存器和存储区

- 原语
  - 由若干条指令组成，用于完成一定功能的一个过程。
  - 与一般过程/函数的区别：
    - “原子操作”：一个操作中的所有动作要么全执行，要么全不执行（它是一个不可分割的基本单位）
    - 执行过程中不允许被中断
    - 在系统态下执行，常驻内存

- **进程图——** 描述一个进程家族关系的**有向树**
  - 子进程可以**继承**父进程的**所有资源**(打开的文件、分配的缓冲区等)
  - 撤销子进程时，把它从父进程那里获得的资源还给父进程
  - 撤销父进程时，要同时撤销它所有的子进程

在PCB中设置家族关系表项：

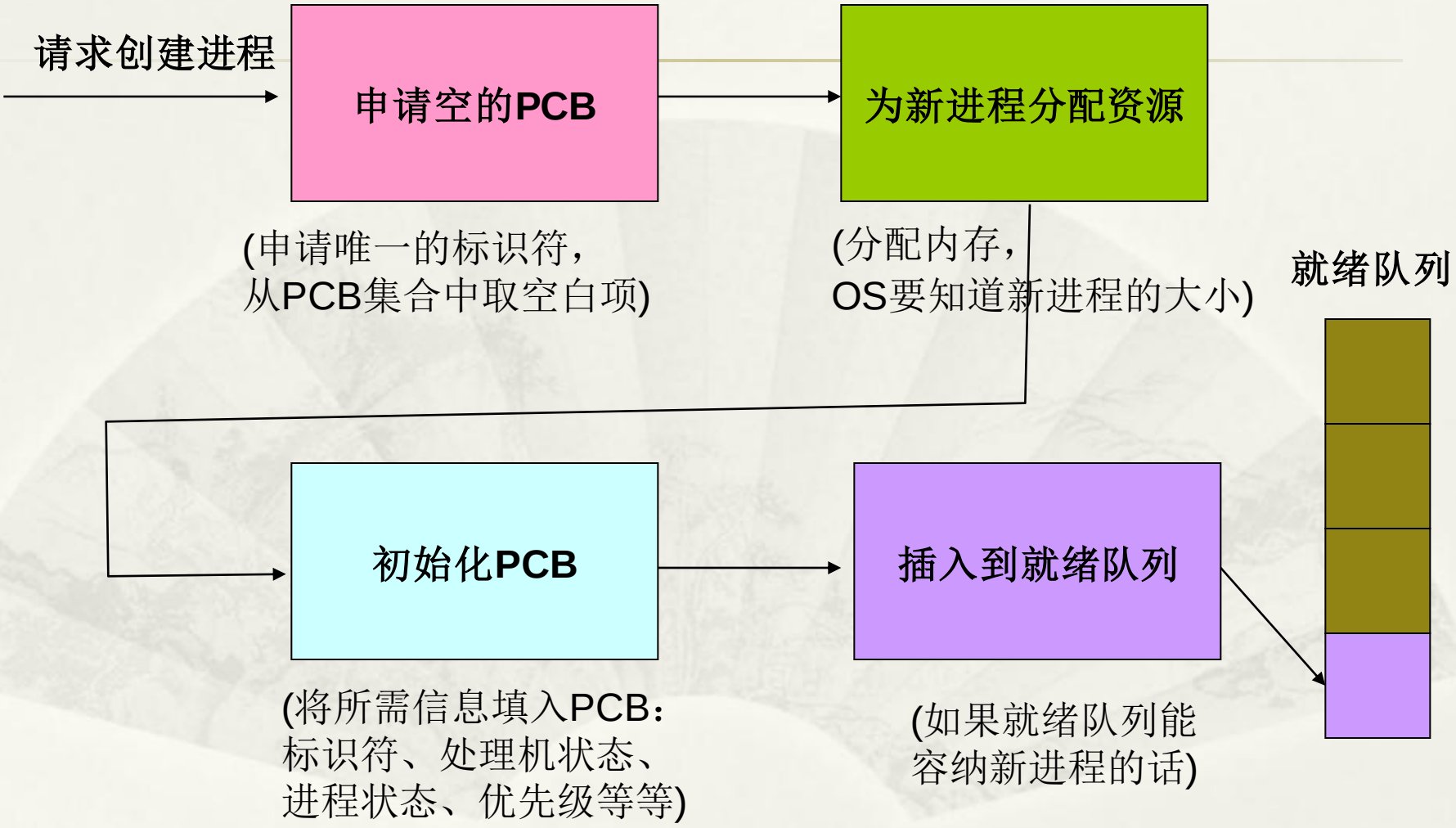
记录父进程，及所有的子进程



进程创建

- 引起创建进程的事件
  - 用户登录
    - 用户从终端登录，为其建立进程(分时系统)
  - 作业调度
    - 把作业装入内存 (批处理系统)
  - 提供服务
    - 为某个请求创建专门的进程。例如用户打印时，创建打印进程
  - 应用请求
    - 建子进程，以并发完成工作，加速任务 (由应用程序创建)

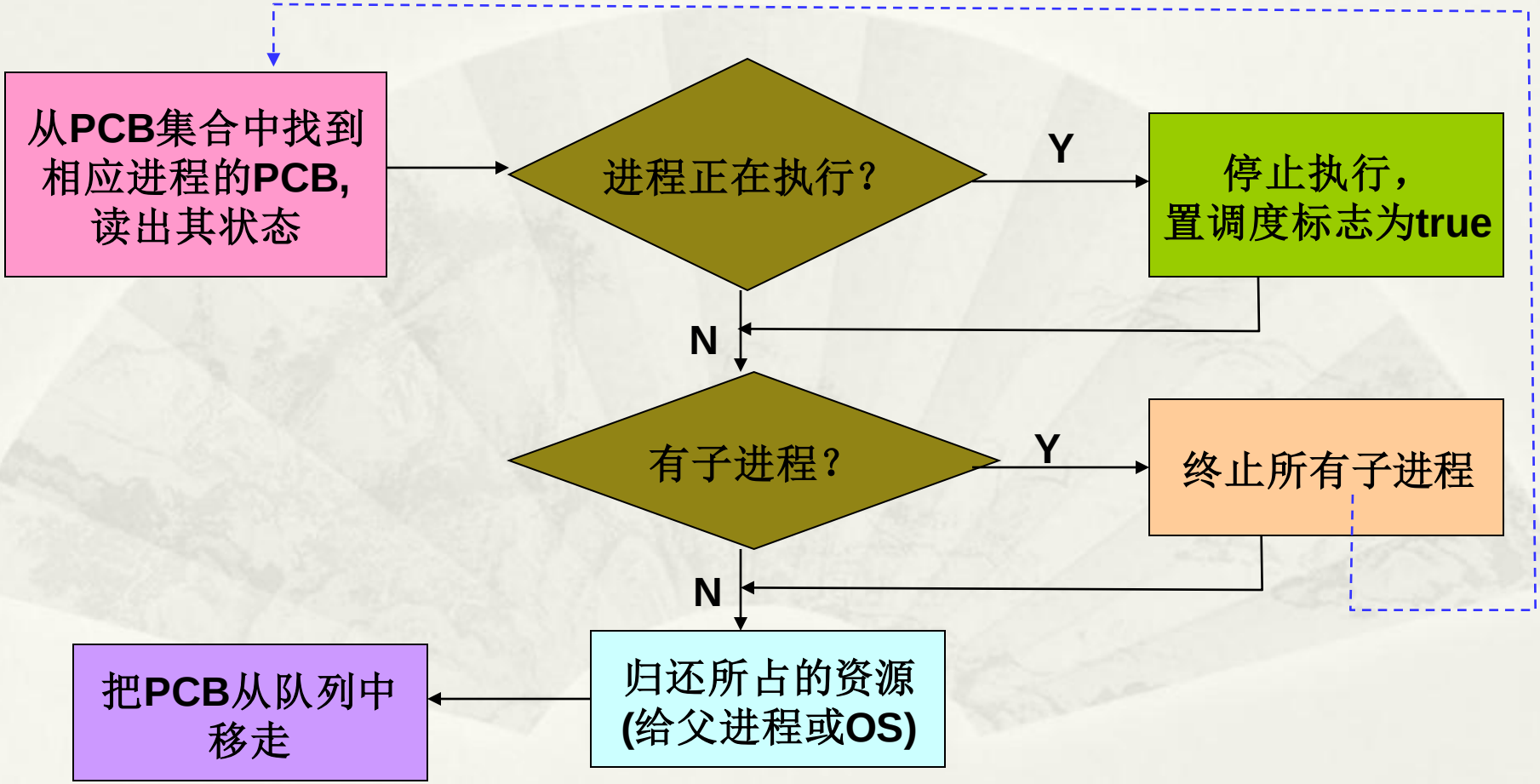
进程的创建流程


请求创建进程
申请空的PCB
为新进程分配资源
(申请唯一的标识符，
从PCB集合中取空白项)
(分配内存，
OS要知道新进程的大小)
就绪队列
初始化PCB
插入到就绪队列
(将所需信息填入PCB：
标识符、处理机状态、
进程状态、优先级等等)
(如果就绪队列能
容纳新进程的话)

进程终止

- 引起进程终止的事件
  - 正常结束：执行到最后的结束指令、中断
  - 异常结束：出现错误或因故障而被迫终止
  - 外界干扰：进程应外界的请求而终止运行
- 进程终止的过程
  - 一个进程可以向其父进程申请终止自己；
  - 也可以因父进程的被终止而被同时终止。

- 进程终止的流程


从PCB集合中找到相应进程的PCB,读出其状态
进程正在执行？
Y
停止执行，置调度标志为true
N
有子进程？
Y
终止所有子进程
N
归还所占的资源(给父进程或OS)
把PCB从队列中移走

进程阻塞与唤醒

- **引起阻塞的事件**
  - 请求系统服务失败
  - 等待某操作完成
  - 数据尚未到达
  - 等待新任务（针对系统进程）
- 进程调用block原语阻塞自己

- **引起唤醒的事件**
  - 与引起阻塞的事件相对应
- 进程调用wakeup原语唤醒另一个进程

- block原语与wakeup原语要匹配使用，以免造成“永久阻塞”

进程挂起与激活

- **进程挂起**
  - 检查被挂进程的状态，改为相应的挂起状态
  - 把进程的PCB复制到指定的区域
  - 最后，转向调度程序重新调度
- **进程激活**
  - 先将进程从外存调入内存
  - 检查该进程的现行状态，改为相应的活动状态
  - 根据优先级确定是否需要重新调度(抢占调度？)

小结

- **阻塞、唤醒一般由OS实现**
  - 阻塞：进程自己的行为，进程自己阻塞自己
  - 唤醒：进程不能唤醒自己，由其他进程唤醒
- 挂起、激活可由用户干预

第四节 进程同步

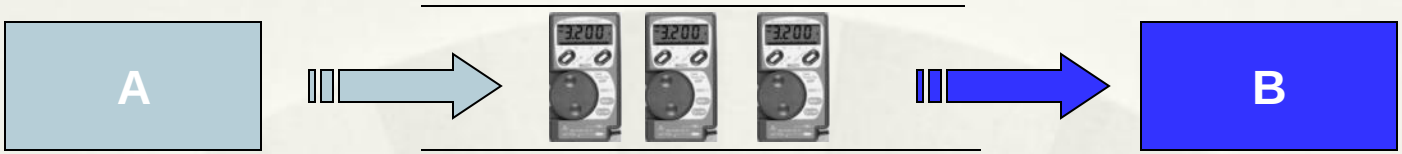
- 进程同步的主要任务
  - 协调多个相关进程的执行次序，使并发执行的进程之间能按一定的规则或时序共享资源和相互合作，从而使程序的执行具有可再现性。

1、基本概念

- 两种形式的制约关系
  - 直接相互制约：源于进程间的合作
  - 间接相互制约：源于进程对资源的共享

* **临界资源**
  * 具有“别人用时，你不能用”的特点
  * 访问时，必须采用“互斥”方式

* 进程同步著名例子：生产者–消费者问题



缓冲区

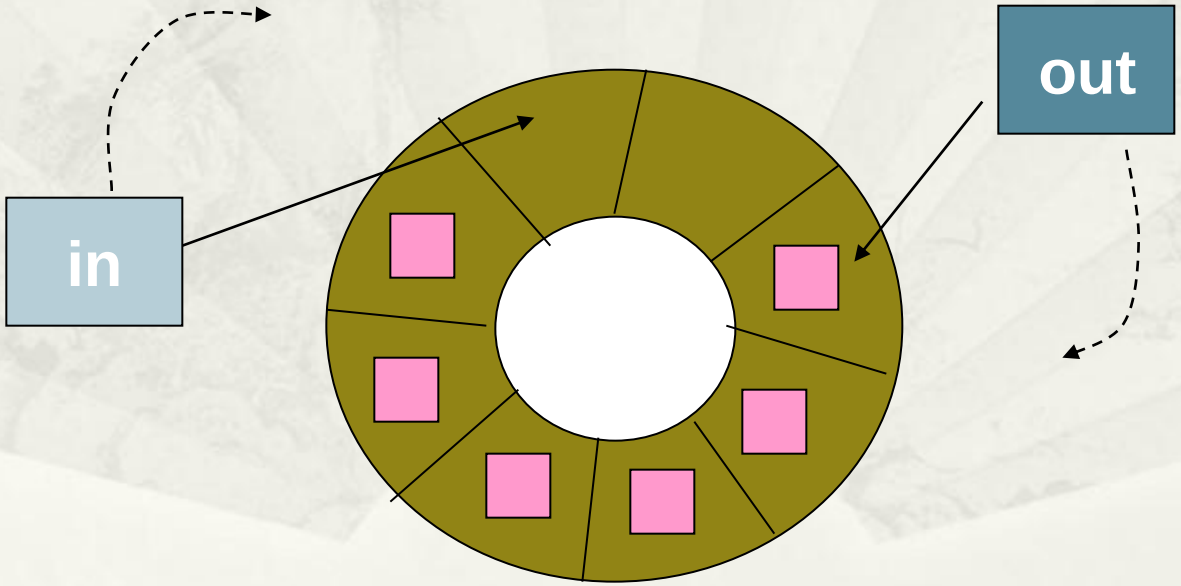
要求：

不允许生产者A向已经**满**的缓冲区**放产品**

不允许消费者B从**空**的缓冲区**拿产品**

* 模拟方法：
  * 缓冲区：数组[0,1,...,n-1]，循环队列
  * 下一个产品应该放在什么位置：in=0, ...,n-1
  * 下一个产品应该从什么位置拿：out=0, ...,n-1
  * 缓冲区里还有多少产品：counter=0, ...,n

* 放一个新产品：in = (in + 1) % n;
* 拿走一个产品：out = (out + 1) % n;
* 什么时候缓冲区满了：counter = n
* 什么时候缓冲区空了：counter = 0



```
void producer()
{
    while(1)
    {
        produce an item in nextp;
            ...
        while(counter == n);
        buffer[in] = nextp;
        in = (in + 1) % n;
        counter ++;
    }
}
```

```
void consumer()
{
  while(1)
  {
        while(counter == 0);
        nextc = buffer[out];
        out = (out + 1) % n;
        counter --;
        consumer the item in nextc;
        …
  }
}
```

* 上面的过程在并发执行时会出错
  * 原因：两者共享变量counter
  * 机器语言描述：

```
mov  ax, [counter]    mov  bx, [counter]
inc   ax              dec   bx
mov [counter], ax     mov [counter], bx
```

  * counter = 5　　生产者、消费者顺序执行一次，结果是5
  * 交替执行时的情况：

```
mov  ax, [counter]     (ax = 5)
inc   ax               (ax = 6)
mov  bx, [counter]     (bx = 5)
dec   bx               (bx = 4)
mov [counter], ax      (counter = 6)
mov [counter], bx      (counter = 4)
```

结果是4

- **刚才的例子说明什么问题？**
  - 程序执行失去了再现性：结果可能是4或5或6
- **解决思路**
  - 把counter当作临界资源处理
  - 令两个程序互斥访问它

- **临界区**
  - 每个进程中访问临界资源的代码段
- **如何实现互斥地访问临界资源：**
  - 保证各进程互斥地进入（执行）自己的临界区，便可实现对临界资源的互斥访问。

* **解决方法**
  * 各进程在进入自己的临界区之前，首先检查临界资源是否正在被别的进程使用/访问。
* **两个步骤**
  ① 没被使用，则设置“使用”标志，然后使用
  ② 使用完了，设置“未用”标志，让别人可以使用

```
repeat
    entry section    //进入区
    critical section;//临界区
    exit section     //退出区
    remainder section; //剩余区
until false;
```

* 设共享变量S=1，标志counter是否正在被使用。

- 由上可见，在程序中设置一个共享变量，用来标志另一个变量或资源是否被使用是行不通的。
- 所以，OS需要提供一个专门的机制，来解决这个问题。

* 同步机制应遵循的规则

① **空闲让进**

② **忙则等待**

③ **有限等待**

* 避免“死等”

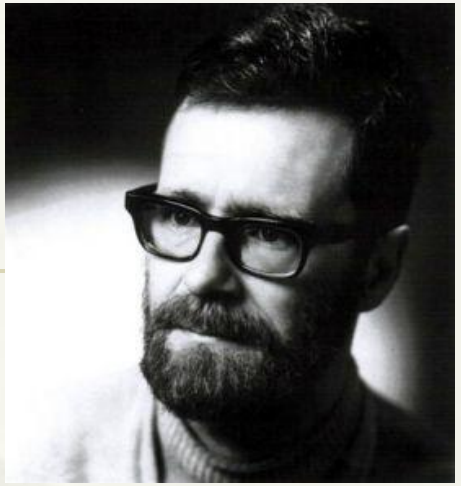
④ **让权等待**

* 不能进入临界区时，立即释放处理机，避免“忙等”。

2、信号量机制

- 卓有成效的进程同步工具
- **提出**
  - 1965. 荷兰 Dijkstra

Dijkstra



(1930—2002)

* 经典语录

① 编程的艺术就是处理复杂性的艺术。

② 简单是可靠的先决条件。

③ 1972年图灵奖演讲：优秀的程序员很清楚自己的能力是有限的，所以他对待编程任务的态度是完全谦卑的，特别是，他们会象逃避瘟疫那样逃避“聪明的技巧”。

④ 计算机科学是应用数学最难的一个分支，所以如果你是一个蹩脚的数学家，最好留在原地，继续当你的数学家。

⑤ 实际上如果一个程序员先学了BASIC，那就很难教会他好的编程技术了：作为一个可能的程序员，他们的神经已经错乱了，而且无法康复。

- **信号量的发展**
  - 整型信号量 → 记录型信号量 → AND型信号量 → 信号量集
- **基本思想**
  - 使用信号量标记临界资源是否被使用
- **注意**
  - 检查和释放信号量的任务由OS完成
  - 应用程序中可调用相应的原语（wait、signal），以完成对信号量的检查和释放

1、整型信号量

* **基本思想**
  * 改变整型变量S的值，以标记临界资源的使用情况
  * 信号量的值只能用wait(S)和signal(S)来访问
  * S的初始值为1或更大。

```
wait(S){
    while(S<=0); // do no-op
     S --;
}

signal(S) { S++;}
```

wait(S)、signal(S)是**原子操作，执行时不可中断。**

(又被称作P、V操作)

- **整型信号量机制的缺点**
  - 只要信号量S<=0，就会不停地测试。
  - 因此，未遵循**“让权等待”**准则—**“忙等”**。
- **故而，引入“记录型信号量机制”，避免“忙等”现象。**

2、记录型信号量

* **组成部分**
  * 整型变量 value：表示资源数目
  * 链表 list：等待访问该资源的进程形成的阻塞队列
* **数据结构定义**

```
typedef struct{
 int value; //资源数目
 struct process_control_block *list;//阻塞队列
} semaphore;
```

- wait(sempahore *S)：

```
S->value - -;
// 如果资源分配完毕，
// 自我阻塞,进入队列
// S.L进行等待
if (S->value<0)
    block(S->list);
```

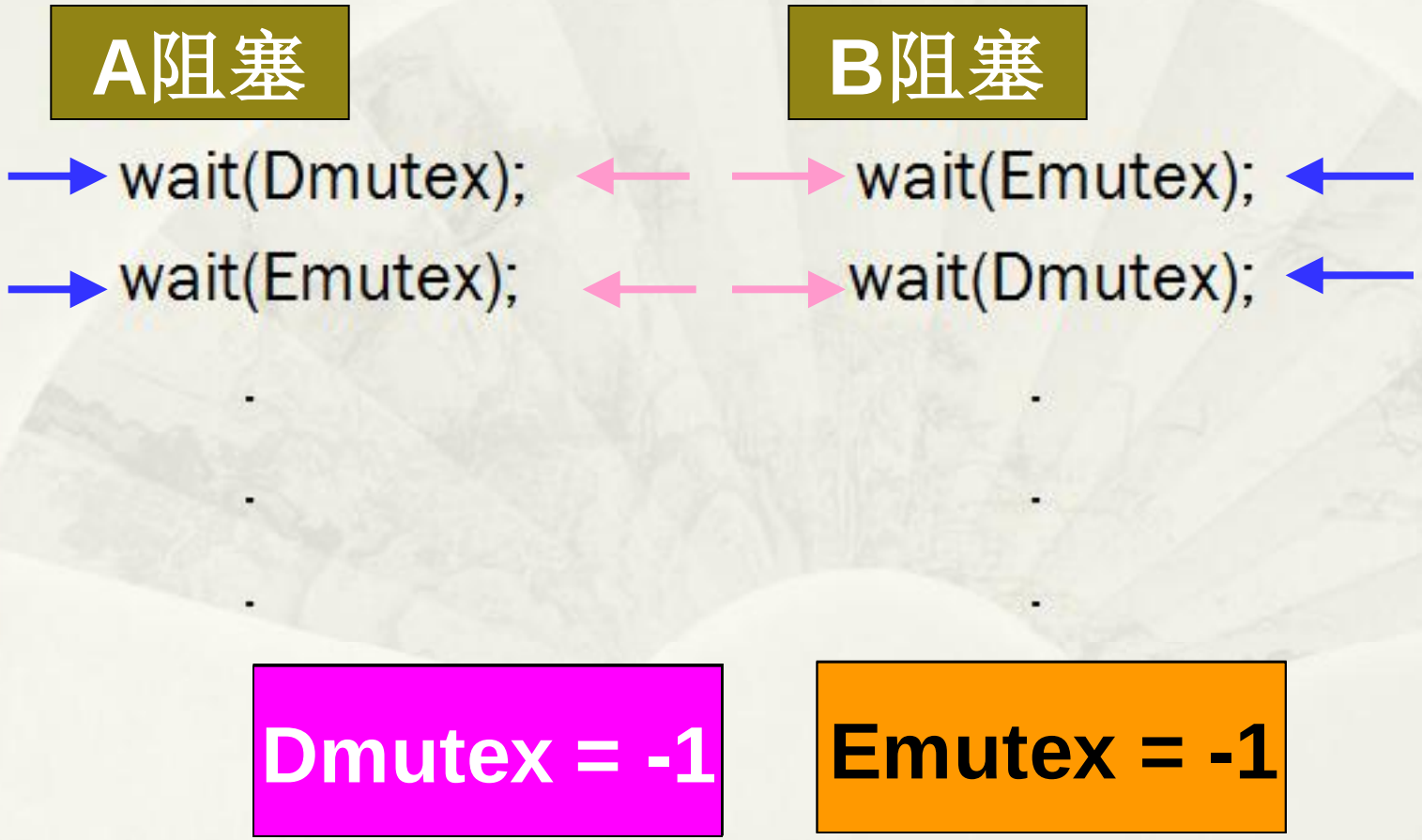
- signal(sempahore *S)：

```
S->value ++;
//如果S.value<=0,
//表示仍有等待的进程，
//则唤醒一个
if (S->value<=0)
    wakeup(S->list);
```

注意： **S.value < 0时， |S.value|表示被阻塞进程的数目**

- **一个现实的问题**
  - 如果进程需要获得多种资源后才能继续执行，如何实现同步？
- **一种解决方法**
  - 使用前面讲的信号量机制能否解决？
- **答案**
  - 不行！！！(why?)

例子：进程A、B访问共享变量D和E，为D和E各设一个互斥信号量Dmutex=1和Emutex=1。



执行顺序：

A: wait(Dmutex); → Dmutex = 0

B: wait(Emutex); → Emutex = 0

A: wait(Emutex); → ∵Emutex = -1 ∴A阻塞

B: wait(Dmutex); → ∵Dmutex = -1 ∴B阻塞

僵持，死锁

阻塞以后，A和B代码后面的Signal无法执行，也就无法改变Dmutex和Emutex的值，从而死锁

- 为解决以上问题，引入“AND型信号量”

3、AND型信号量

* 基本思想
  * **一次性**分配进程执行过程中**所需要的全部资源**，待使用完后再**一并释放**。
  * 即使只有一个资源无法分配，其他的资源也都不分配。
  * 也即，**要么全分配，要么一个也不分配**

- Swait(S1,…,Sn)

```
{
    if (S1>=1 && … &&  Sn>=1)
    {
       for(i= 1; i<=n; i++)
            Si --;
    }
    else{ 将进程阻塞，放入第一个发现si<1的等待队列中;
    将进程指针指向Swait开头; }
}
```

- Ssignal(S1,…,Sn)

```
{
    for(i=1;i<=n;i++)
      {  Si ++;
         将si等待队列中的进程转入就绪队列;}
}
```

- **一个现实的问题**
  - 如何一次申请N个同类资源？
- **一种解决方法**
  - 使用前面讲过的信号量机制？
- **回答**
  - 申请N个，要调用N次wait()，效率低
  - 另外，也无法满足“当资源数低于某个下限值时，便不分配”的需求。
- 所以，引入“信号量集”机制

4. 信号量集

- 对“AND信号量”机制进行改进，形成一般化的“信号量集”机制。

- **基本思路**
  - 每次分配之前，必须测试该资源的数量，看其是否大于其下限值。
  - 因此，除信号量S外，还设置资源的需求量d，和资源的下限值t

Swait(S1,t1,d1,...,Sn,tn,dn)

Ssignal(S1,d1,...,Sn,dn)

- Si>=ti才能分配资源
- 分配时:Si=Si-di

* “信号量集”的几种特殊使用

① Swait(S,d,d)

- 只有一个信号量，每次分d份资源，当资源数少于d时，不给分配。

② Swait(S,1,1)

- 蜕化为一般记录型信号量(S>1)或互斥信号量(S=1)

③ Swait(S,1,0)

- 特殊且有用的信号量——功能类似于“开关”。
- S>=1时，允许多个进程执行某特定的代码。
- 置S=0后，阻止任何程序执行该段代码。

4、信号量的应用

* **信号量机制的用途一般有三种**
  - ① **用于“限制”访问：**
    - * **限制进程互斥地访问共享资源**
    - * **限制进程访问有限数量的共享资源**
  - ② **用于“协调”执行：**
    - * **协调进程间的执行次序**

      （实现进程间的协调，即实现前趋关系）

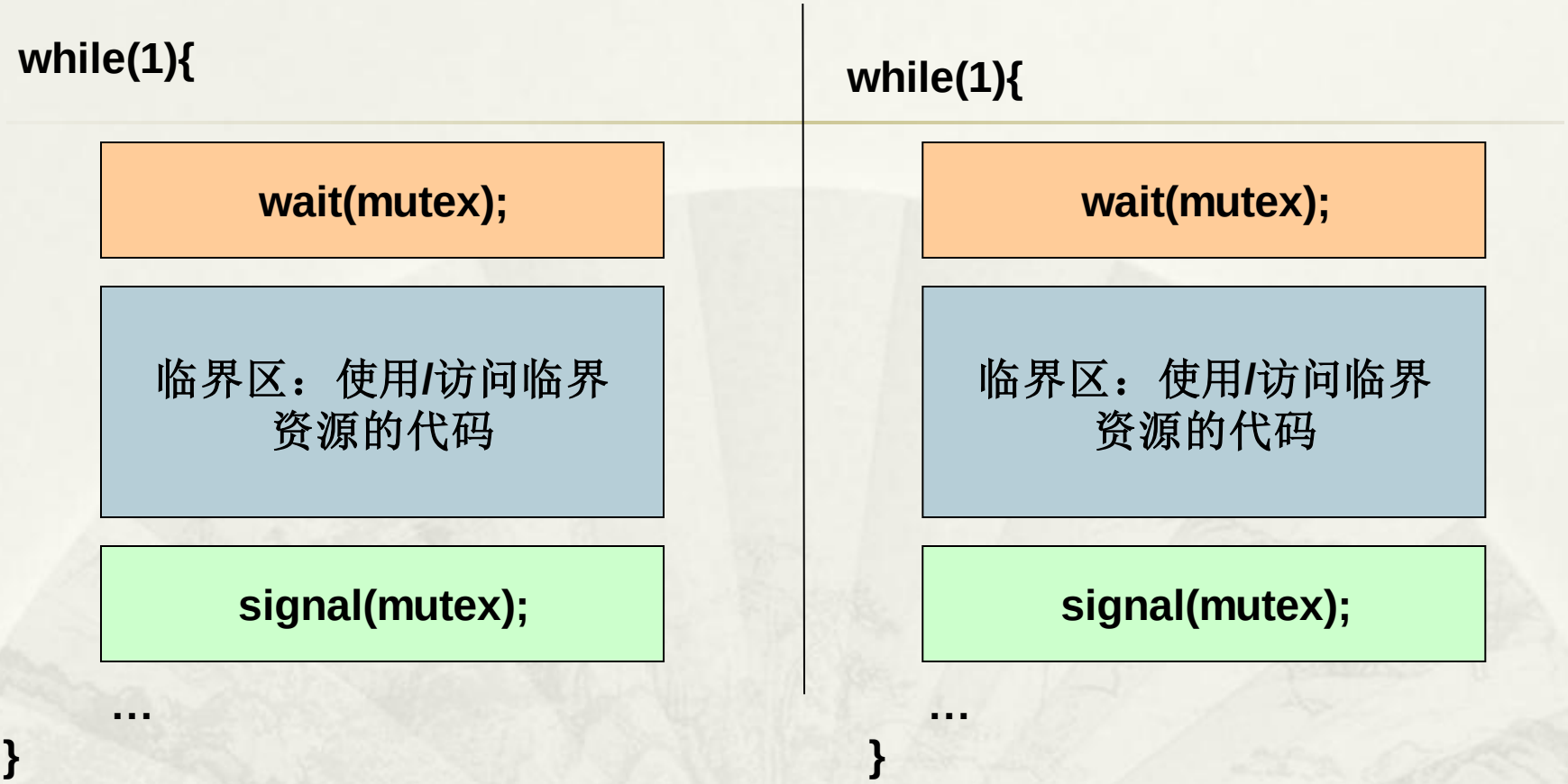
1、利用信号量实现进程互斥

* 如何让多个进程互斥访问一个临界资源？
* **方法**

① 设一个信号量mutex，令其初始值为1(互斥信号量)，用它控制进程的访问。

② 将各进程访问临界资源的代码放入wait(mutex)和signal(mutex)之间。

semaphore mutex= 1; //定义信号量



- **注意：** 利用信号量实现进程互斥时，wait、signal 一般要成对出现
- **缺少wait：** 导致系统混乱。
- **缺少signal：** 使阻塞进程无法被唤醒。

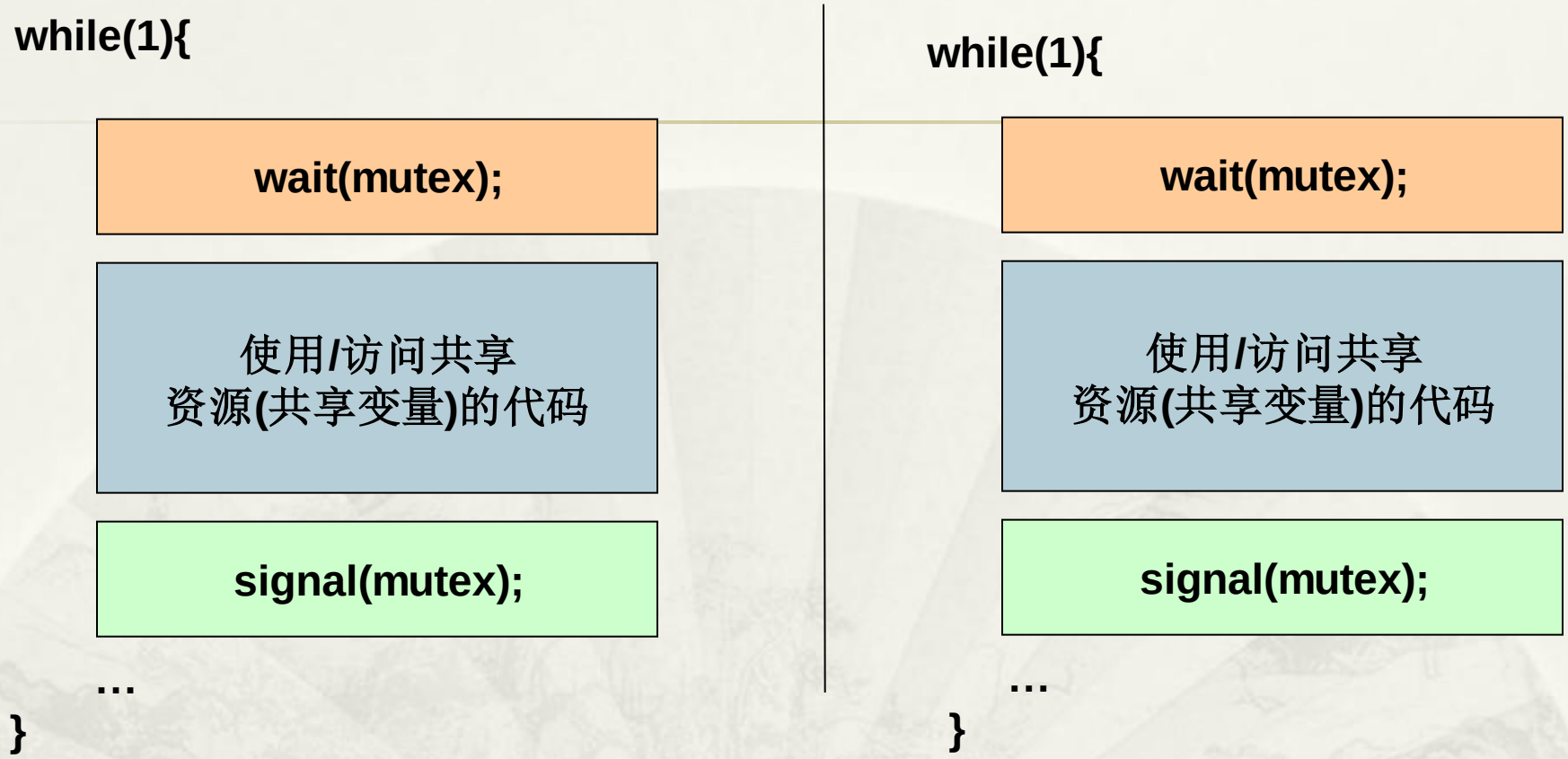
2、利用信号量实现进程访问有限数量的共享资源

* 如果共享资源的数量有限（设>1），如何控制多个进程对它们的访问？
* **方法**

① 设一个信号量mutex，令其初始值等于共享资源的数量，用它控制进程的访问。

② 将各进程访问共享资源的代码放入wait(mutex)和signal(mutex)之间。

semaphore mutex= 资源数量; //定义信号量



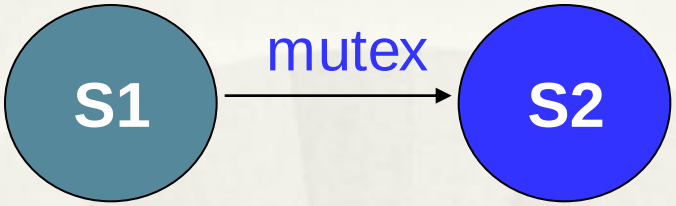
- 注意事项与使用互斥信号量时相同

3、利用信号量协调进程执行(实现前趋关系)

- 例1：设并发执行的两个进程P1、P2

要求：S1在S2之前执行

前驱图：



方法：设置信号量 mutex=0 (why?)

P1：S1;
signal(mutex);

P2：wait(mutex);
S2;

例2：利用信号量实现语句间的前趋关系

semaphore a=b=c=d=e=f=g=0;

P1(){S1; signal(a); signal(b);}

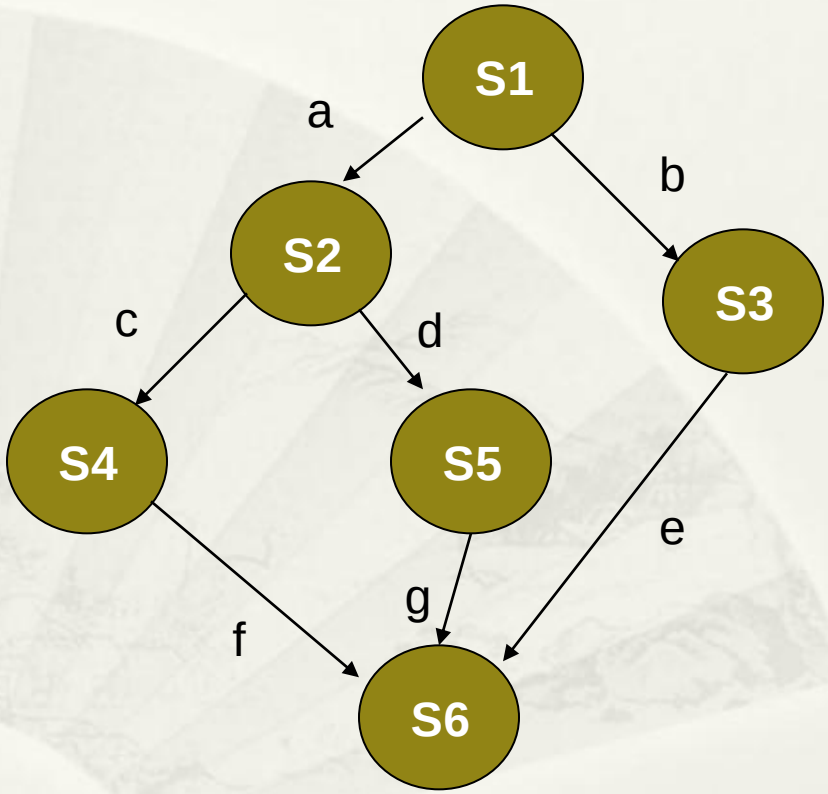
P2(){wait(a); S2; signal(c); signal(d);}

P3(){wait(b); S3; signal(e);}

P4(){wait(c);S4; signal(f);}

P5(){wait(d); S5; signal(g);}

P6(){wait(e); wait(f); wait(g); S6;}



信号量小结

- 信号量应用分类：

  “限制”：信号量的wait、signal在同一个进程中

  “协调”：信号量的wait、signal在不同的进程中

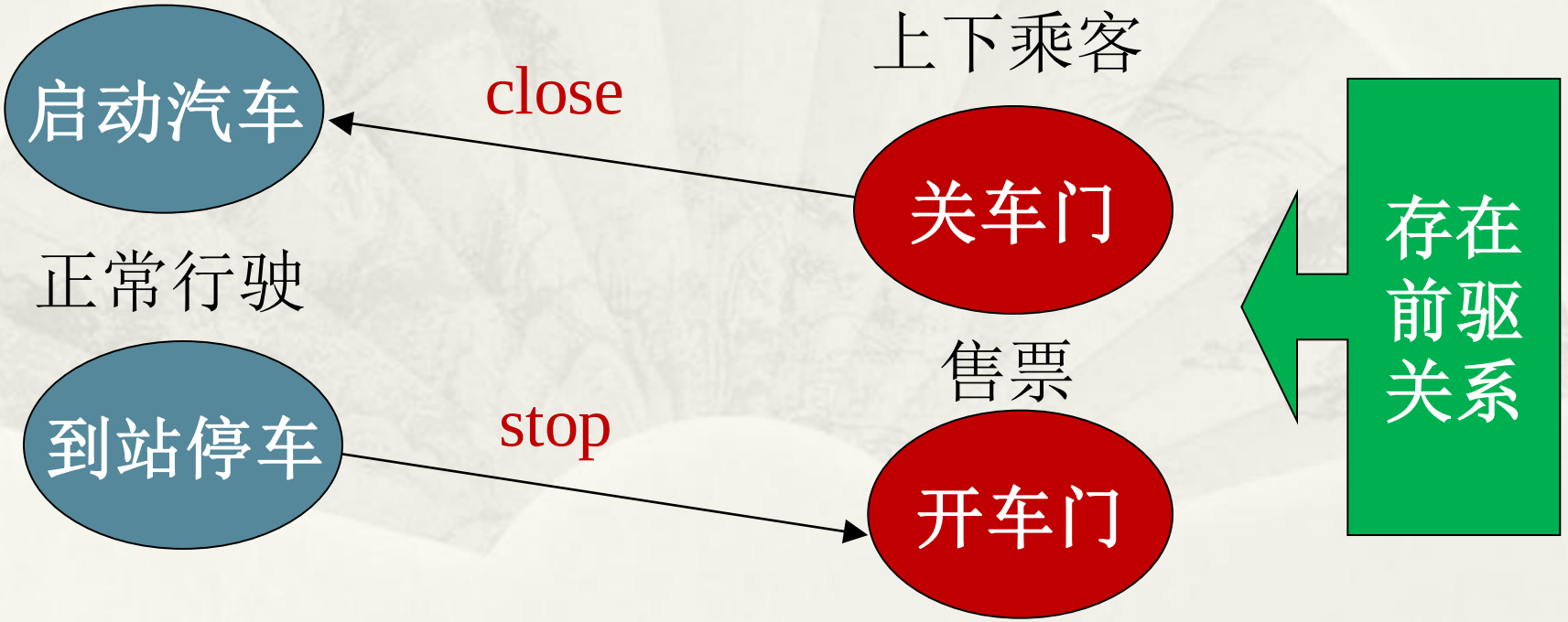
练习题

* 公交车上，司机和售票员的活动分别为：

司机：	售票员：
启动汽车	上下乘客
正常行驶	关车门
到站停车	售票
开车门	

问：二者活动之间有什么同步关系，请用信号量机制实现二者的活动。

* 解答
  * 二者的活动关系：
    ① 司机启动汽车前，售票员必须已关闭车门
    ② 售票员开车门前，司机必须已到站停车



semaphore stop=0, close=0;

司机：

```
while(1){
    wait(close);
    启动汽车;
    正常行驶;
    到站停车;
    signal(stop);
}
```

售票员：

```
while(1){
    上下乘客;
    关车门;
    signal(close);
    售票;
    wait(stop);
    开车门;
}
```

第四节 经典进程同步问题

* 经典进程同步问题：
  1. 生产者-消费者问题
  2. 哲学家进餐问题
  3. 读者-写者问题

1. 生产者-消费者问题

- 思路1
  - 利用“记录型信号量”解决
- 方法
  - 设三个信号量

① 互斥信号量 mutex
  - 用于各进程对缓冲区的互斥访问

② 信号量empty、full
  - 分别表示缓冲区中空缓冲单元和满缓冲单元的数量

```
semaphore  mutex=1, empty=n, full=0；// 定义信号量
item  buffer[n];   // 定义缓冲区
int   in=0, out=0; // 定义下标指示变量
void proceducer() //生产者如何执行(如何往缓冲区放东西)
{
  do{
      产生一个商品放入nextp中;
      wait(empty); //等待缓冲区有地方
      wait(mutex); //等待可以使用缓冲区
      buffer[in] = nextp;
      in = (in + 1) % n;
      signal(mutex); //告诉其他进程:缓冲区可以使用了
      signal(full); //告诉其他进程:缓冲区多了一个商品
   }while(1);
}
```

```
void consumer() //消费者如何执行(如何从缓冲区取东西)
 {
   do{
      wait(full); //等待缓冲区有商品
      wait(mutex); //等待可以使用缓冲区
      nextc = buffer[out];
      out = (out + 1) % n;
      signal(mutex); //告诉其他进程:缓冲区可以使用了
      signal(empty); //告诉其他进程:缓冲区空出一个地方
      消费商品 nextc;
   }while(1);
}
```

* 注意
  * wait(mutex)和signal(mutex)必须成对出现
  * empty、full的wait、signal也要成对出现
  * 注意多个wait的顺序不能颠倒。

- 颠倒wait的顺序以后可能会死锁，比如：
- 生产者p：　　消费者c：

wait(empty);	wait(mutex);
wait(mutex);	wait(full);

信号量初始值：mutex:=1, empty:=n, full:=0

- 执行顺序：

c的wait(mutex)→mutex=0

c的wait(full)→ ∵full=0 ∴ c阻塞

p的wait(empty)→ empty=n-1

p的wait(mutex)→ ∵ mutex=0 ∴ p阻塞

僵持状态，死锁

* 思路2
  * **利用AND信号量解决**
  * **可避免因为wait顺序书写错误，引发的死锁问题**
* 方法
  * 生产者:

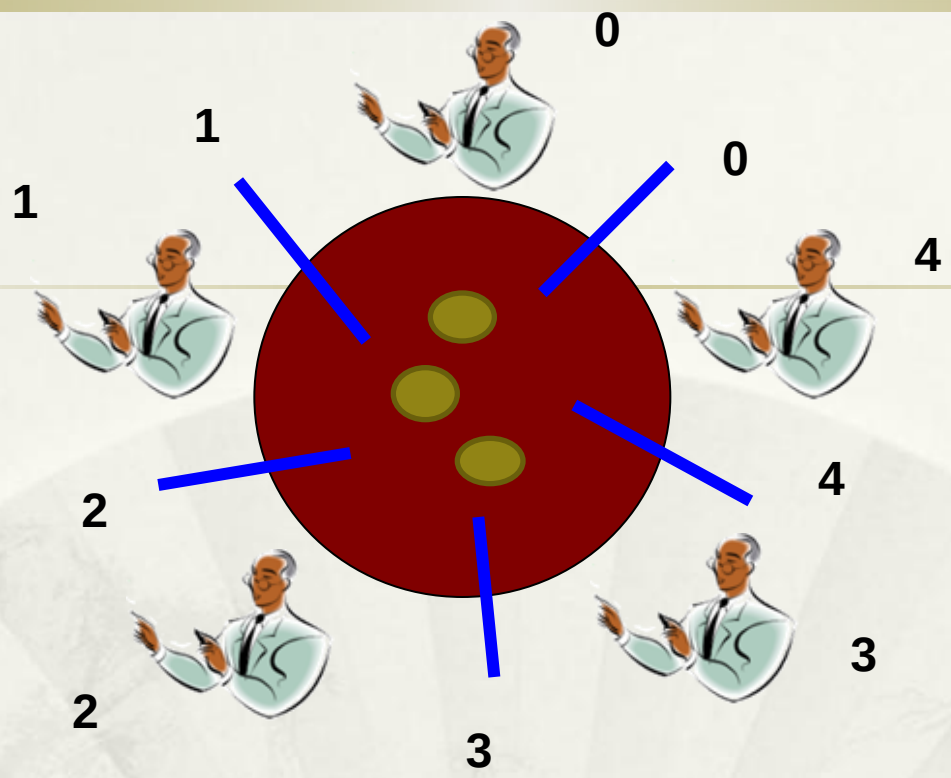
```
...
Swait(empty,mutex);
buffer[in] = nextp;
in := (in + 1) % n;
Ssignal(mutex,full);
```

  * 消费者:

```
Swait(full,mutex);
nextc = buffer[out];
out = (out + 1) % n;
Ssignal(mutex,empty);
...
```

2、舌尖上的哲学家

- 哲学家进餐问题
  - **提出者：**Dijkstra
  - **描述**
    - 5个哲学家共用一张圆形餐桌，
    - 桌子上放着5支筷子、5只碗。
    - 他们**吃饭**和**思考**交替进行。
    - 吃饭时只能拿自己左右两侧的筷子，
    - 不吃时放下筷子进行思考，周而复始。



- **筷子是共用的**。身边任何一个人吃饭,自己就没法吃(缺少筷子)
- **所以，每根筷子都是临界资源**（每次只允许一位哲学家使用）
- **可以为每根筷子设置一个信号量**（实现对筷子的互斥使用）

- **思路1**
  - **利用记录型信号量**
- **方法**

```
semaphore chopstick[5]={1,1,1,1,1};
void philosopher(int i) //第i个哲学家的活动
{  do{
      wait(chopstick[i]); //拿左筷子
      wait(chopstick[(i+1) % 5]); //拿右筷子
      吃饭;
      signal(chopstick[i]); //放左筷子
      signal(chopstick[(i+1) % 5]); //放右筷子
      思考;
    }while(1);
 }
```

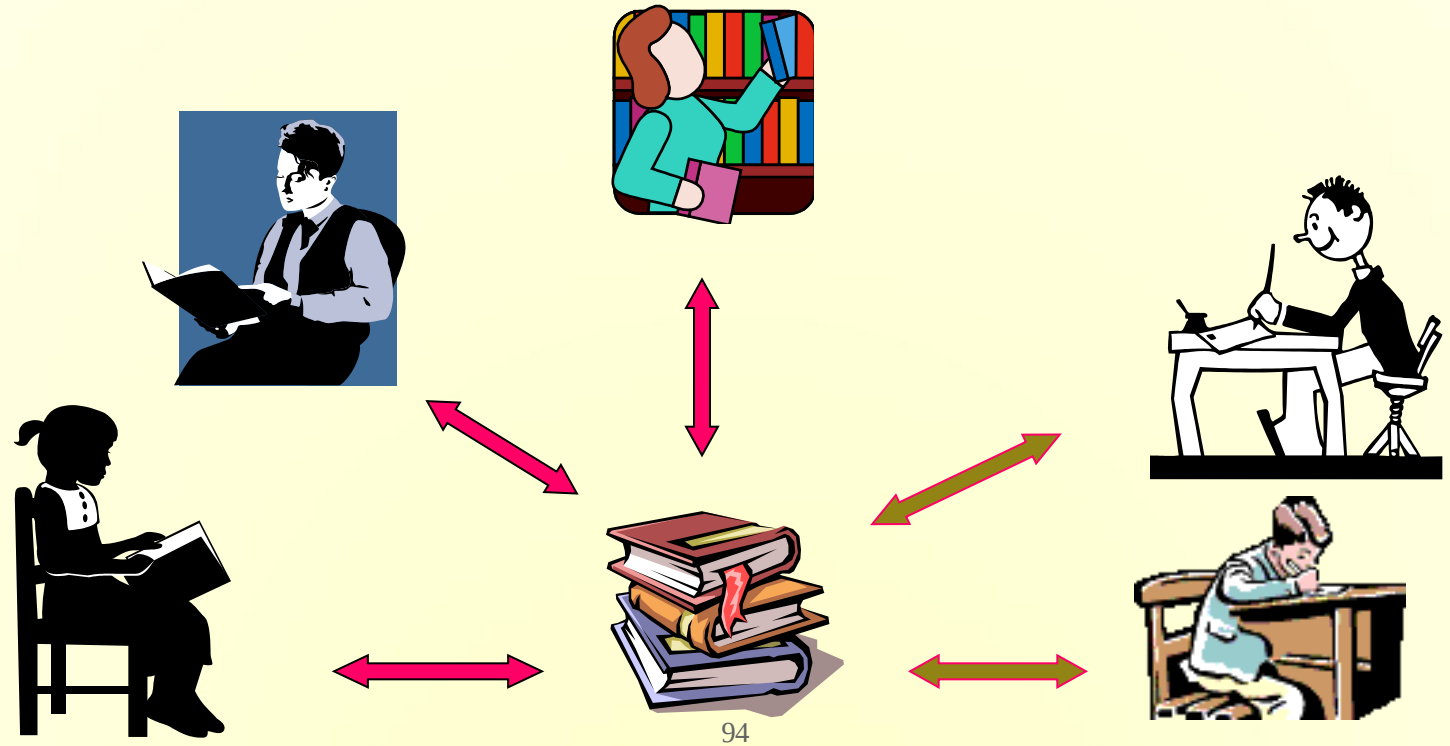
- **缺点**
  - 如果每个人开始都**先拿自己左边的筷子**
  - 可能导致人手一根筷子，而拿不到第二根筷子
  - 从而全部阻塞，引起死锁。
- **解决办法**
  ① 至多只允许四个人“同时”拿左边的筷子，最终保证至少一个人能吃上饭。
  ② 仅当左右筷子均可用时，才允许他拿筷子进餐。(AND信号量)
  ③ 规定**奇数号**哲学家先拿起他**左边**的筷子，再拿右边的筷子；而**偶数号**哲学家先拿起他**右边**的筷子，再拿左边的筷子。

* 思路2
  * 利用AND型信号量
* 方法

```
semaphore chopstick[5]={1,1,1,1,1};
void philosopher(int i)
{  do{
      Swait(chopstick[i], chopstick[(i+1)%5]);
      吃饭;
      Ssignal(chopstick[(i+1)%5], chopstick[i]);
      思考;
   }while(1);
}
```

3、读者—写者问题

- 有两组并发进程：读者、写者，共享一组数据区
- 要求：允许多个读者同时执行读操作
  不允许读者、写者同时操作（读写互斥）
  不允许多个写者同时操作（只能一人写）



利用记录型信号量

- 用信号量wmutex控制写者与其他进程的互斥
- 设整型变量readcount，记录正在进行读操作的读者数目

  ∵readcount=0时，写者才可以写

  ∴只有这时，读者才需要执行wait(wmutex)，看看是否有写者在写
- 因为readcount被所有读者共享，所以还要用信号量rmutex控制它们对readcount的操作。

```
semaphore  rmutex=1,wmutex=1;
int  readcount=0;
读者进程：
 do{
        wait(rmutex);
        if (readcount==0)
          wait(wmutex);
        readcount ++;
        signal(rmutex);
        读操作;
        wait(rmutex);
        readcount--;
        if (readcount == 0)
          signal(wmutex);
        signal(rmutex);
   }while(1);
```

```
写者进程：
   do{
        wait(wmutex);
        写操作;
        signal(wmutex);
```

•readcount是临界资源，需要用信号量控制

•如果没有读者在读，再看看是否有写者在写

•如果没有其他读者在读的话，就通知写者说现在可以写了

* 新要求
  * 如果最多只允许RN个读者读，该怎么解决？
* **利用信号量集解决**
  * 方法
  ① 设信号量L，初始值为RN，以控制读者的数目
    * 每当有一个读者要读时，执行swait(L,1,1),使L减1
    * 当有RN个读者正在读时，L=0
    * 再来一个读者要读的话，便会被阻塞。
  ② 设信号量 mx，初始值为1(互斥信号量)，以控制写者

•如果已有一个写者在写，则有mx=0
•如果已有至少一个读者在读，则有L<RN
•存在以上两种情况之一时，本写者阻塞并等待
•否则表示，没有其他写者在写，并且没有读者在读，则置mx=
并继续下一步
•注：

• 检查是否有写者在写 (只检查,不修改mx的值)

写者进程：

正在读操作的读者数目是否

Swait (mx,1,1;L,RN,0);

如果L>=1，说明还允许新读者，则将L减1，并继续执行下一步

否则，说明已有RN个读者在读，则阻塞并等待

Ssignal (mx,1);

信号量机制 练习题

1、医生诊病的过程如下：

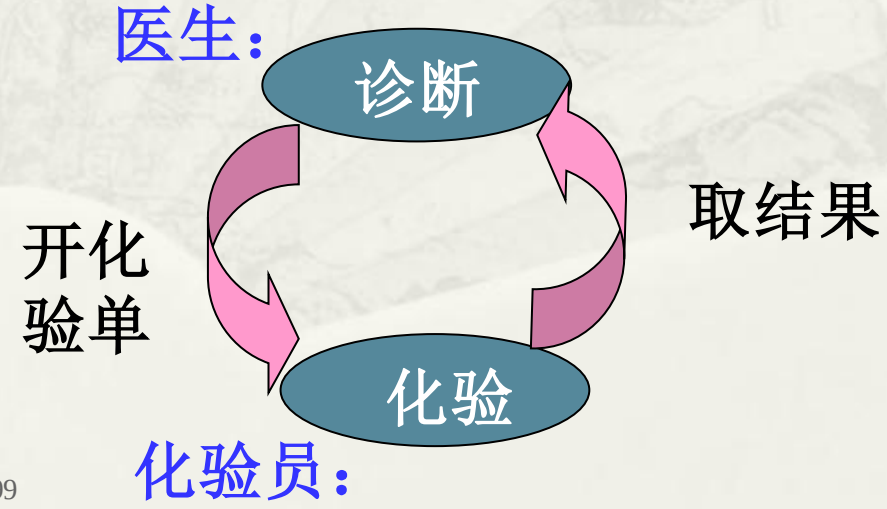
看病 → 开化验单 →

病人取样送化验室 → 等化验完毕

→ 交回化验结果给医生 →

医生继续诊病

- 请用两个进程模拟医生和化验员的活动，并实现二者之间的同步



* 解答

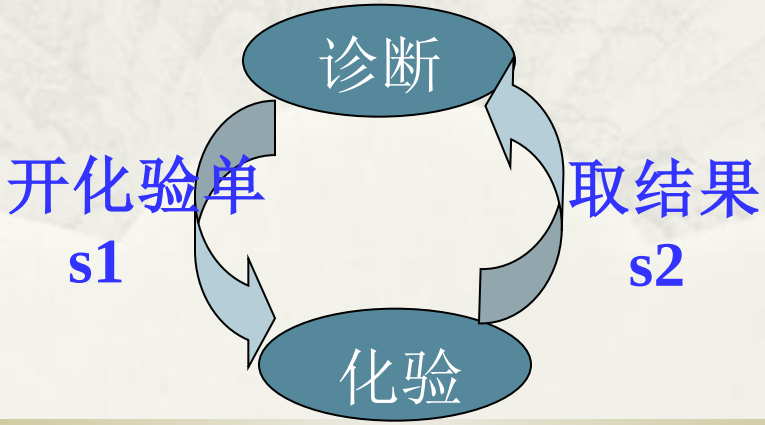
semaphore s1=0, s2=0;

化验员:

```
do{
    wait (s1);
    化验;
    signal (s2) ;
}while(1);
```

医生:

```
do{
    诊断;
    signal(s1);
    wait (s2);
}while(1);
```



2、描述一个没有死锁的解决哲学家进餐问题的算法

① 让奇数号的哲学家先右后左的筷子，让偶数号的哲学家先左后右的筷子

```
semaphore chopstick [5]={1,1,1,1,1};
void philosopher(int i)
 {
  while(1)
  { if ( i % 2==0)
     { wait (chopstick[i]);
       wait (chopstick[(i+1) % 5]);
     }
    else{ wait (chopstick[(i+1) % 5]);
          wait (chopstick[i]);
     }
                                   吃饭;
                                   signal (chopstick[i]);
                                   signal(chopstick[(i+1) % 5]);
                                   思考;
                                }
                              }
```

② 最多允许四位哲学家拿起筷子。

- semaphore chopstick[5]={1,1,1,1,1};

semaphore max = 4;

```
void philosopher(int i)
{  while(1)
   {  wait(max);
      wait (chopstick[i]);
      wait (chopstick[(i+1) %5]);
      吃饭;
      signal(chopstick[i]);
      signal (chopstick [(i+1) % 5]);
      signal (max);
      思考;
   }
}
```

3、售票问题

* 某车站售票厅，任何时刻最多可容纳20名购票者进入。
* 当售票厅中少于20名购票者时，厅外的购票者可立即进入，否则需在外面等待。
* 若把一个购票者看作一个进程，请回答下列问题：
  ① 用wait、signal操作管理这些并发进程时，应怎样定义信号量，写出信号量的初值以及信号量取值的含义。
  ② 写出购票者进程的算法描述。

- 解答

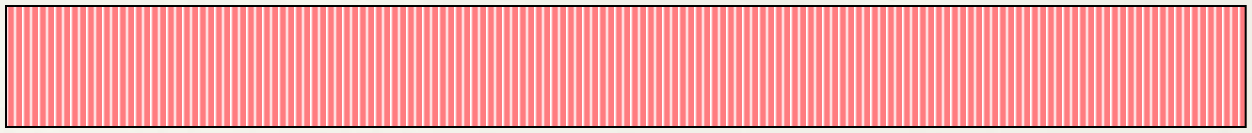
① 定义信号量empty=20，代表可以进入的购票者人数

② semaphore empty= 20;

```
void person()
{
    wait(empty);
     进入售票厅;
     购票;
     退出;
     signal(empty);
  }
```

4、单行车道问题

* a, b 两点间是一段东西向的单行车道
* 请设计一个自动管理系统，管理规则如下：
  ① ab段间有车辆行驶时，同方向的车可以同时进入，但另一方向的车必须在段外等待；
  ② ab之间无车时，到达a（或b）的车辆可以进入，但不能从a、b点同时进入；
  ③ 某方向在ab段行驶的车辆驶出了ab段且无车辆进入时，应让另一方向等待的车辆进入。
* 请用信号量机制对ab段实现正确管理



- 分析
  - 定义两个进程P_ab、P_ba，分别实现对汽车从a端或b端进入时的道路控制
    - 每当有汽车要进入道路时，便相应调用P_ab或P_ba
  - 定义两个变量countab、countba，分别记录从a端或b端进入的汽车数量
    - 每当有汽车进入时，都要检查或改变它们，因此它们属于临界资源，对它们的访问需要用信号量控制
    - 因此，再设两个互斥信号量mutexab、mutexba
  - 定义互斥信号量s，用于实现汽车“不能从a、b点同时进入”

```
semaphore  s=1, mutexab=1, mutexba =1;
int   countab=0, countba=0;
void P_ab()
{
    wait(mutexab);
    countab++;
    if  (countab==1) wait(s);
    signal(mutexab);
    进入;
    wait(mutexab);
    countab - -;
    if (countab==0) signal(s);
    signal(mutexab);
}
```

```
void P_ba()
{
    wait(mutexba);
    countba++;
    if (countba==1)  wait(s);
    signal(mutexba);
    进入;
    wait(mutexba);
    countba - -;
    if (countba==0)  signal(s);
    signal(mutexba);
}
```

5、阅览室问题

* 有一个阅览室，读者进入时必须在一张登记表上登记
* 该表中每一座位列为一表目，包括座号和读者姓名。
* 读者离开时要消掉登记信息，阅览室中共有100个座位
* 请问：
  ① 为描述读者动作，应设几个进程
  ② 用信号量机制实现进程间的同步

分析：登记表是临界资源 —— 设信号量 mutex = 1
用两个信号量分别表示座位的数量和在读读者的数量
—— seat = 100, reader = 0

解答:

（1）2个进程

（2）

```
semaphore seat=100, reader=0, mutex=1;
```

```
void P1()
{
  do{
    wait(seat);
    wait(mutex);
    登记信息;
    signal(mutex);
    signal(reader);
    就座，阅读;
  }while(1);
}
```

```
void P2()
{
   do{
      wait(reader);
      wait(mutex);
      消掉信息;
      signal(mutex);
      signal(seat);
      离开阅览室;
   }while(1);
}
```

第六节 进程通信

- 进程通信的概念
- 进程通信的类型
- 消息传递通信的实现方法
- 消息传递系统实现中的若干问题
- 消息缓冲队列通信机制

1、进程通信的概念

- 进程通信，是指进程之间的信息交换
- 低级通信（互斥、同步）
  - 利用信号量机制实现进程间的数据传递
  - 缺点：效率低；对用户不透明
- 高级通信（进程通信）
  - 进程之间利用OS提供的一组通信命令，高效地传送大量数据的信息交换方式
  - 优点：高效，方便，简化了通信程序的设计

2、进程通信的类型

- 高级通信机制的类型
  - ① 共享存储器系统
  - ② 管道通信系统
  - ③ 消息传递系统
  - ④ 客户机-服务器系统

① 共享存储器系统

* 基于共享数据结构的通信方式
  * 进程共享某些数据结构(如队列)
  * 程序员必须负责同步管理
  * 所以**低效**，只适合**传输少量数据**
* 基于共享存储区的通信方式
  * 属于高级通信方式
  * **划出一块共享存储区**，进程通过对存储区中数据的读写来通信

② 管道通信系统

* 首创于UNIX系统
* 管道
  * 是一个共享文件（pipe文件），用于连结一个发送进程和一个接收进程，以实现通信
  * 发送进程和接收进程利用管道进行通信
* 管道机制还提供三个方面的协调能力
  ① 互斥
    * 一个进程读/写管道，另一进程必须等待
  ② 同步
    * 发送进程写入一定数量的数据后便阻塞，接收进程取走后再唤醒
    * 接收进程读到空管道时，也会阻塞，有数据时才被唤醒
  ③ 确定对方都存在后，才进行通信

③ 消息传递系统

- 进程间交换的数据是有格式的，称为message（网络通信中，称为报文）
- OS提供了相应的通信命令

④ 客户机-服务器系统

- 主流通信机制
- 实现方法：
  1. 套接字
  2. 远程过程(方法)调用

3、消息传递通信的实现方式

① **直接消息传递系统**（直接通信方式）

- **发送进程利用OS提供的发送原语，直接把消息发送给目标进程**
  - **Send (Receiver, message)**
  - **Receive (Sender, message)**

② 信箱通信（间接通信方式）

- 进程之间通过共享数据结构（信箱），以消息暂存方式实现的通信
- 操作原语
  - 信箱的创建、撤消；消息的发送、接收
- 信箱的创建和拥有者
  - OS或用户（通信）进程
- 信箱的种类
  - 私用信箱、公用信箱、共享信箱
- 利用信箱通信时，发送和接收进程之间的关系
  - 一对一、多对一、一对多、多对多

4、消息缓冲队列通信机制

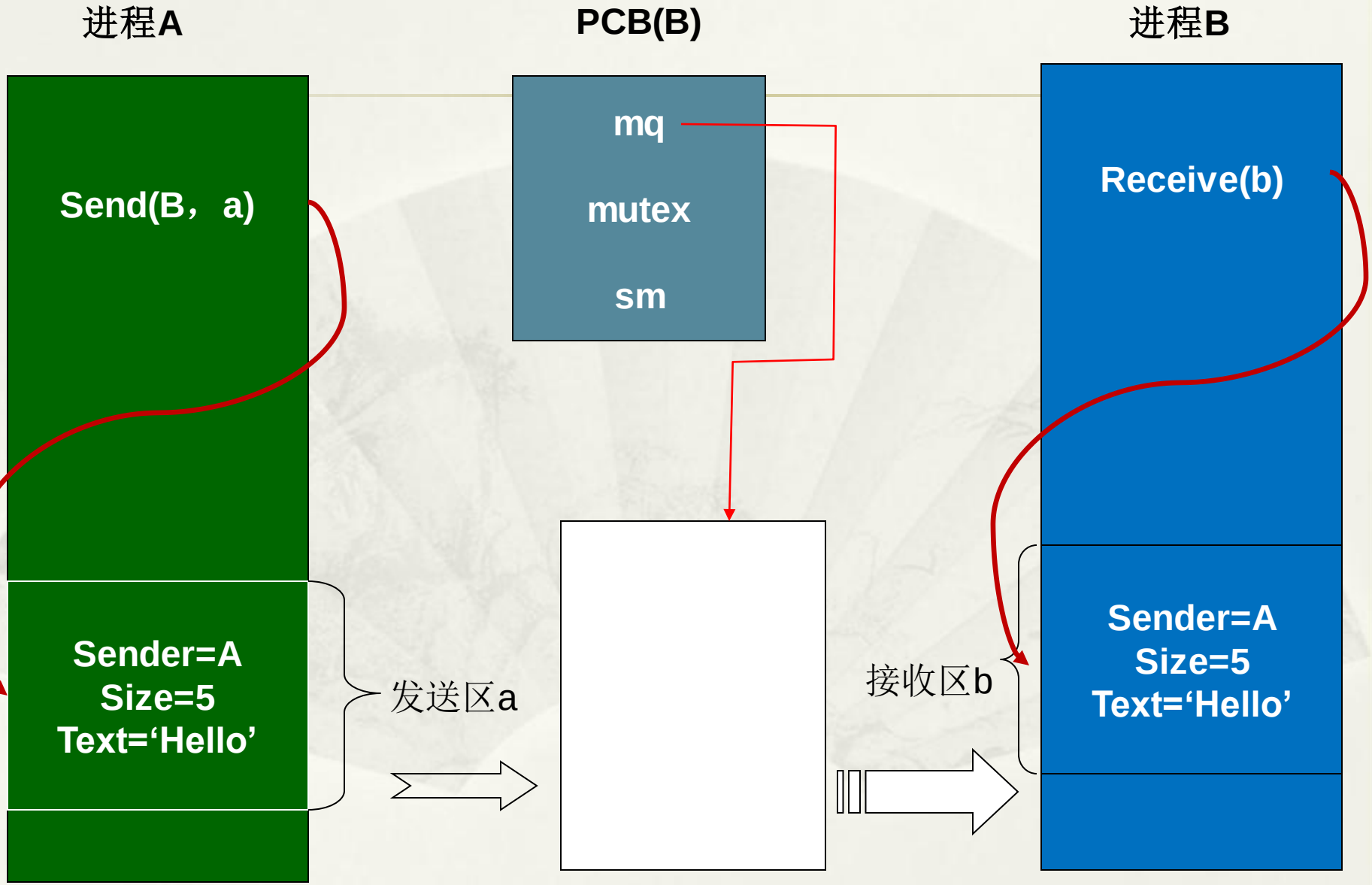
- 提出者：Hansan
  - 通过内存中的消息缓冲区进行进程通信
  - 广泛应用于本地进程间通信
- 发送原语 send(receiver , a)
  - a：发送区的地址
- 接收原语 receive(b)
  - b：接收区的地址

（1）消息缓冲区中的数据结构：

```
struct message_buffer {
    int sender ;  //发送进程标识符
    int size ;    //消息长度
    char *text;    //消息正文
    struct message_buffer *next; //下一消息缓冲区指针
};
```

（2）PCB中有关通信的数据项：

```
typedef struct processcontrol_block {
        …
        struct message_buffer *mq ;  //消息队列首指针
        semaphore mutex;  //消息队列互斥信号量
        semaphore sm;      //消息队列资源信号量
} PCB;
```


进程A
PCB(B)
进程B
Send(B，a)
mq
mutex
sm
Receive(b)
Sender=A
Size=5
Text='Hello'
发送区a
接收区b
Sender=A
Size=5
Text='Hello'

第七节 线程的基本概念

- 线程的基本概念
- 线程间的同步和通信
- 内核支持线程和用户级线程
- 线程控制

1、线程的基本概念

- 进程（60年代）
  - 目的
    - 使多个程序能够并发执行，提高资源利用率和系统吞吐量
  - 属性
    - OS中拥有资源和独立运行的基本单位
  - 局限性
    - 创建、撤消与切换时空开销大，限制了并发程度的提高
- 线程（80年代）
  - 目的
    - 减少并发时付出的时空开销，使系统具有更好的并发性
  - 基本思想
    - “轻装上阵”——将进程的两个属性分离
    - 线程—独立调度的基本单位，进程—拥有资源的单位

2、线程的属性

① 轻型实体

- 线程几乎不占资源(TCB等)

② 独立调度的基本单位

- 切换迅速且开销小

③ 可并发执行

④ 共享进程的资源

- 同一进程内的线程共享进程的资源

⑤ 可以创建、撤销另一个线程

3、线程的状态

* 每个线程都利用TCB和一组状态参数进行描述

① 状态参数

  * 寄存器状态、堆栈、运行状态、优先级…

② 运行状态

  * 就绪状态、执行状态、阻塞状态

线程和进程的区别与联系

- 调度
  - 线程是调度的基本单位
  - 进程是资源拥有的基本单位
- 拥有资源
  - 除必不可少的资源外，线程不拥有系统资源，但可以访问其隶属进程的系统资源，从而获得系统资源
- 并发性
  - 支持多线程的OS中，不仅不同进程的线程之间可以并发，同一进程内的线程之间也可并发
- 系统开销
  - 进程切换时的时空开销大
  - 线程切换时，只需保存和设置少量信息，因此开销很小
- 支持多处理机系统

小结(1)

- 进程的概念和定义
- **前趋图如何画**
- 进程的特征
- **进程的基本状态及转换**
- **PCB的作用**
- **进程、程序的联系与区别**

小结(2)

- 进程的创建、撤销、终止
- **进程同步、信号量、临界区、临界资源**
- 进程间高级通信方式
- **线程的概念**

练习题

1、临界区是指(　　)　D

A. 一块缓冲区　　B. 一段数据区

C. 同步机制　　D. 一段程序

2、一个进程实体是 (　　)　C

A. 由处理机执行的一个程序

B. 一个独立的程序 + 数据集

C. PCB + 程序 + 数据集

D. 一个独立的程序

3、当(B)时，进程由执行状态转为就绪状态

A. 进程被调度程序选中　B. 时间片到

C. 等待某一事件　D. 等待的事件发生

4、某进程在运行过程中需要等待从磁盘上读入数据，此时该进程的状态将(　　)

C

A. 从就绪变为执行

B. 从执行变为就绪

C. 从执行变为阻塞

D. 从阻塞变为就绪

5、分配到必要的资源并获得处理机时的进程状态是（　B　）

A. 就绪状态　　B. 执行状态

C. 阻塞状态　　D. 撤销状态

6、（　D　）是一种只能进行 wait 操作和 signal操作的特殊变量

A. 调度　　B. 进程　C. 同步　D. 信号量

7、进程控制块是描述进程状态和特性的数据结构，一个进程（D）

A. 可以有多个进程控制块

B. 可以和其他进程共用一个进程控制块

C. 可以没有进程控制块

D. 只能有惟一的进程控制块

8、当已有进程进入临界区时，其他试图进入临界区的进程必须等待，以保证对临界资源的互斥访问，这是下列（B）同步机制准则

A. 空闲让进　　B. 忙则等待

C. 有限等待　　D. 让权等待

9、下面对临界区的论述中，正确的论述是（ D）

A. 临界区是指进程中用于实现进程互斥的那段代码

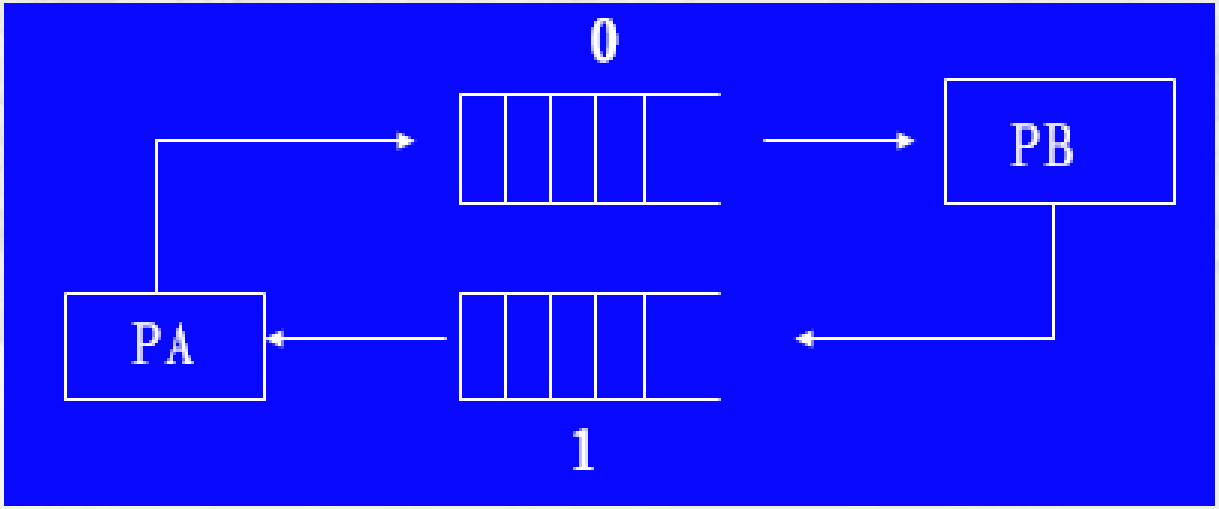
B. 临界区是指进程中用于实现进程同步的那段代码

C. 临界区是指进程中用于实现共享资源的那段代码

D. 临界区是指进程中访问临界资源的那段代码

6、缓冲队列问题

* 两个进程P_A、P_B通过FIFO缓冲队列连接，每个缓冲区长度等于传送消息的长度，P_A、P_B之间通信满足：
  ① 当空缓冲区存在时，相应的发送进程才能发送消息
  ② 当缓冲队列中至少存在一个非空缓冲区时，相应的进程才能接收一个消息
* 试描述发送函数send(I, m)和接收函数receive(I, m);并用它俩实现两个进程的通信



* 定义数组buf[0], buf[1]；bufempty[0],buffull[1]是PA的私有信息量； bufempty[1],buffull[0]是PB的私有信息量。

初始时：

bufempty[0]= bufempty[1]=n, (n为缓冲队列中的缓冲区个数)

buffull[0]=buffull[1]=0

```
void send (I, m)
{
    wait (bufempty [ I ] );
    按FIFO方式选择一个
    空缓冲区buf[ I ] (x);
    buf[ I ](x) = m;
    buf[ I ](x)置满标志;
   signal(buffull [ I ] );
}
```

```
void receive (I, m)
{
    wait (buffull [ I ] );
    按FIFO方式选择一个满
    数据的缓冲区buf[ I ] (x);
    m = buf[ I ](x) ;
    buf[ I ](x)置空标志;
   signal(bufempty [ I ] );
}
```

PA调用send (0, m)和receive (1, m)

PB调用send (1, m)和receive (0, m)