第二章

进 程 管 理

本章主要内容

- 进程的基本概念
- 进程控制
- 进程同步
- 经典进程的同步问题
- 管程机制
- 进程通信
- 线程

2.1 进程的基本概念

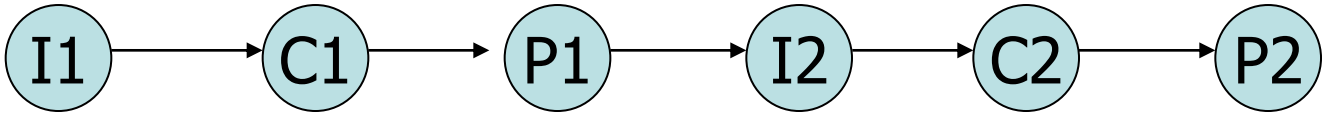
程序只能采用顺序的方式执行，为了使程序能并发执行，引入了进程概念，作为资源分配和独立运行的基本单位。操作系统所具有的四大特征都是基于进程建立起来。本节将介绍进程的基本概念

- 2.1.1 程序的顺序执行及特征
- 2.1.2 前趋图(DAG)
- 2.1.3 程序的并发执行及其特征
- 2.1.4 进程的特征与状态
- 2.1.5 进程控制块(PCB)

问题思考：为什么要引入进程？如何引入进程？

2.1.1 程序的顺序执行及其特征

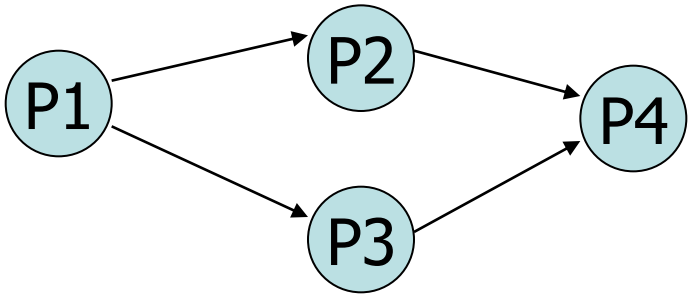
- **程序执行有固定的时序：**应用程序可分为若干程序段，各程序段必须按某种先后次序顺序执行。如图所示：I为输入，C为计算，P为打印



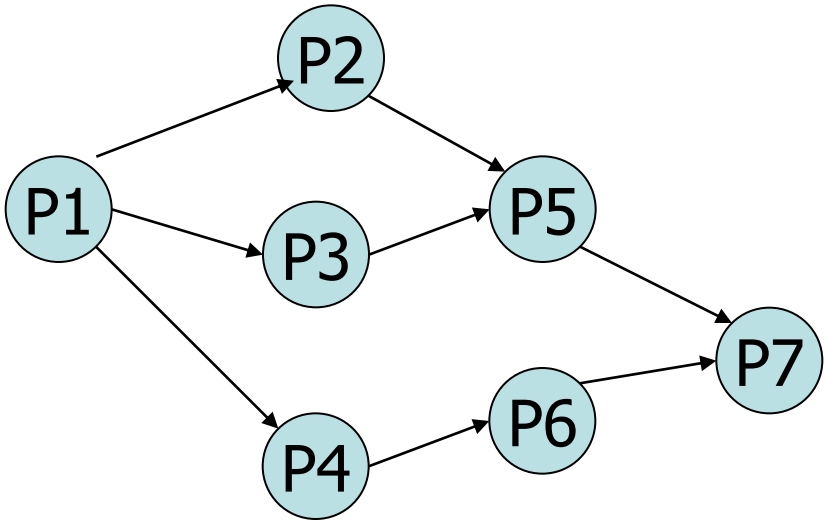
- **顺序执行的基本特征**
  - 顺序性
  - 封闭性：程序在封闭环境下执行
  - 可再现性：如环境和初始条件相同，程序执行结果相同。

2.1.2 前趋图PG定义

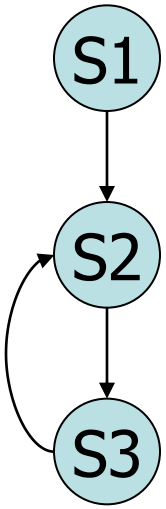
- 有向无循环图，用于描述进程之间执行的前后关系。
- 表示方式：
  - （1）p1→p2
  - （2）→={(p1,p2)| p1 必须在p2开始前完成}
- 基本概念：
  - 直接前驱，直接后继，初始节点、终止节点
  - 节点表示：一条语句，一个程序段，一进程。
  - 权重



前趋图示例



(a) 具有7个结点的前趋图



(a) 具有循环的前趋图

图(a)可表示为$P_1 \rightarrow P_2$, $P_1 \rightarrow P_3$, $P_1 \rightarrow P_4$, $P_2 \rightarrow P_5$, $P_3 \rightarrow P_5$, $P_4 \rightarrow P_6$, $P_5 \rightarrow P_7$, $P_6 \rightarrow P_7$

或表示为：

$P=\{P_1, P_2, P_3, P_4, P_5, P_6, P_7,\}$

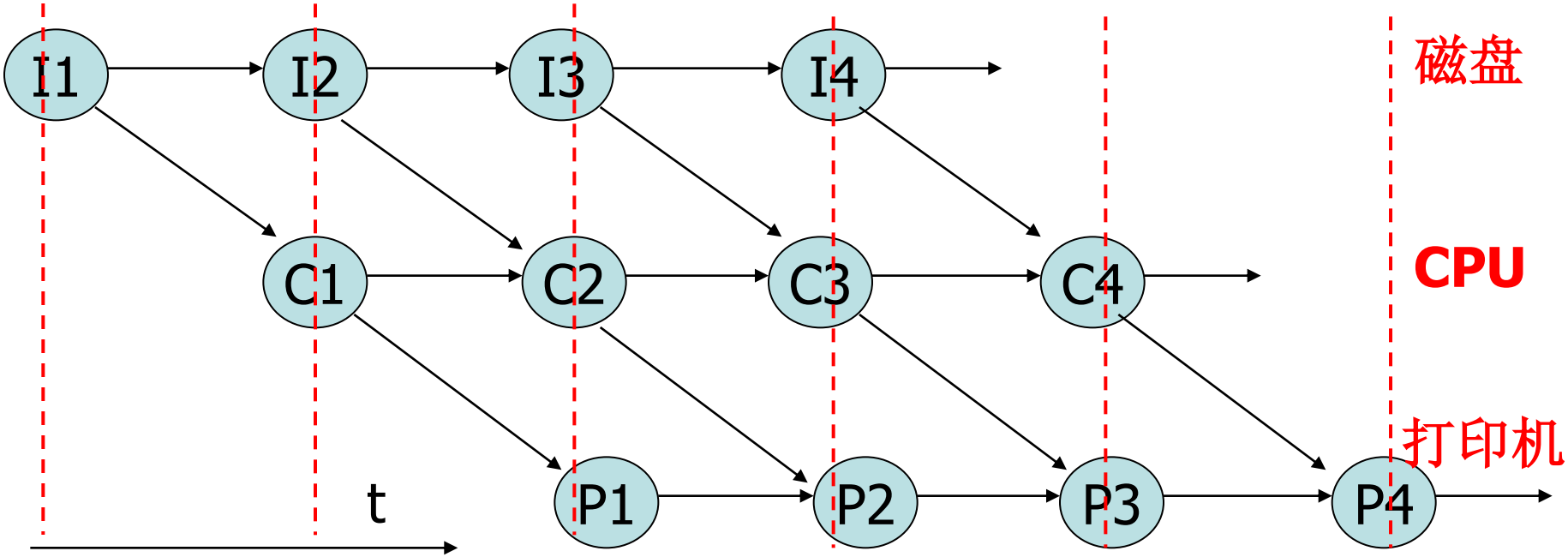
$\rightarrow=\{(P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_5), (P_3, P_5), (P_4, P_6), (P_5, P_7), (P_6, P_7)\}$

上周回顾

- 操作系统四大基本特征：并发、共享、虚拟、异步
- 操作系统主要功能：处理机管理、存储器管理、设备管理、文件管理、用户接口
- 操作系统的结构设计：微内核结构
- 前驱图及其表示

2.1.3 程序的并发执行

❖ 多个程序的并发执行（可能性分析）



前趋关系： $I_i \rightarrow C_i$，$I_i \rightarrow I_{i+1}$，$C_i \rightarrow P_i$，$C_i \rightarrow C_{i+1}$，$P_i \rightarrow P_{i+1}$

而I_{i+1}和C_i及P_{i-1}是重迭的，即P_{i-1}和C_i以及I_{i+1}之间，可以并发执行。

2.1.3 程序的并发执行

❖ 并发执行的特征

*间断性：“执行—暂停—执行”

*失去封闭性：主要由共享资源引起

*不可再现性：即运行速度不同，结果也不同

例：有2个循环程序A和B，它们共享一个变量N，设N的初值为n，A每次做N:=N+1；B则每次要执行Print(N)，然后再做N:=0. 若程序A,B以不同的速度运行有以下三种结果：

N:=N+1在print(N)和N:=0之前，则N值分别为n+1,n+1,0.

N:=N+1在print(N)和N:=0之后，则N值分别为n,0，1.

N:=N+1在print(N)和N:=0之间，则N值分别为n,n+1,0.

2.1.4 进程的特征和状态

❖ 进程的特征和定义

由于程序并发执行导致了不可再现性，为了对程序并发执行加以描述和控制，并保证正确性，引入了进程的概念。

- 进程的定义：程序的一次执行过程，是系统进行资源分配和调度的基本单位
- 结构特征
  - 进程＝程序段＋数据段＋进程控制块(PCB)，总称“进程映像”（进程实体）。
  - 创建和撤销进程，实质是创建和撤销PCB

2.1.4 进程的特征和状态

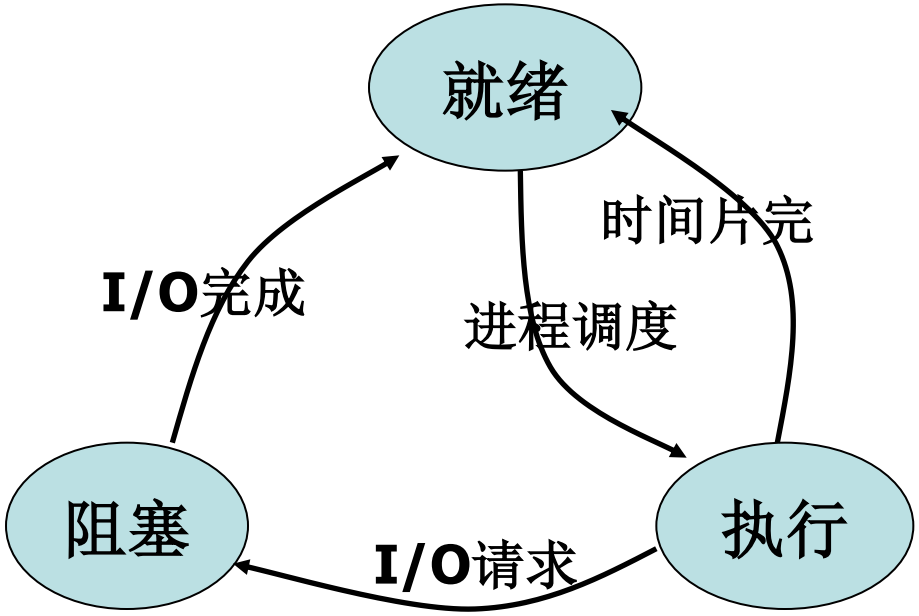
* 动态性（最基本特征）
  - 由“创建”而产生，由“调度”而执行，由得不到资源而阻塞，由撤消而消亡。存在生命周期（程序是静态的）。
* 并发性
  - 只有建立了进程，才能并发执行。
* 独立性
  - 进程是独立运行，独立获得资源，独立调度的基本单位。
* 异步性
  - 独立以不可预知的速度推进

2.1.4 进程的特征和状态

❖ 进程的三种基本状态

由间断性决定

- **就绪状态：** 获得除CUP外的所有资源。（就绪队列）
- **执行状态：** 获得CPU，正在执行。（当前进程）
- **阻塞状态：** 也称**等待状态**，指执行中的进程由某种原因不能继续执行，从而放弃CUP处于暂停状态。（阻塞队列）如：I/O请求等



进程的三种基本状态及其转换

2.1.4 进程的特征和状态

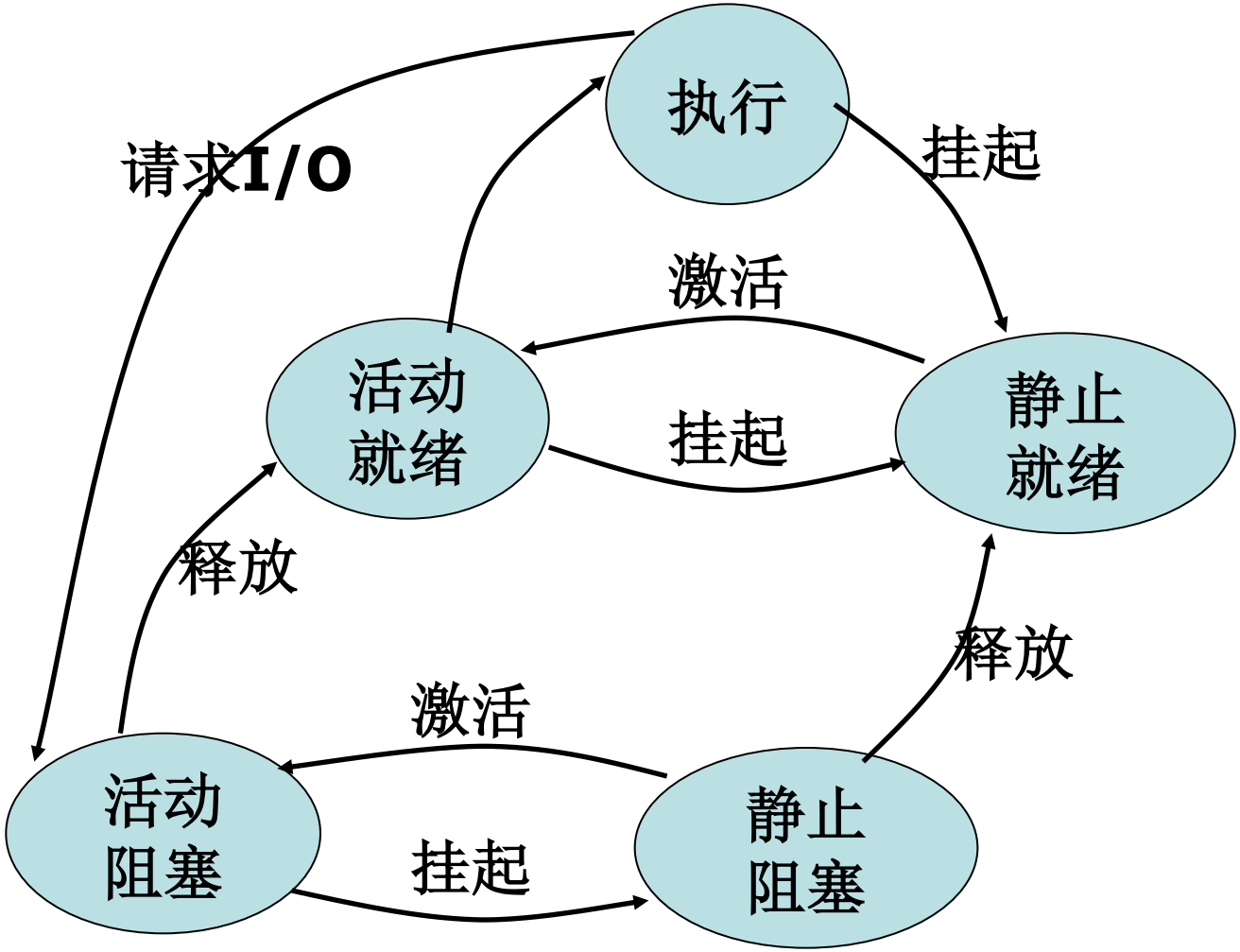
❖ 挂起状态（暂时被换出内存的状态）

* 引入原因
  - 终端用户请求
  - 父进程请求
  - 负荷调节需要
  - 操作系统需要

* 进程状态的转换（挂起←→非挂起）
  - 活动就绪→ 静止就绪
  - 活动阻塞→ 静止阻塞
  - 静止就绪→ 活动就绪
  - 静止阻塞→ 活动阻塞

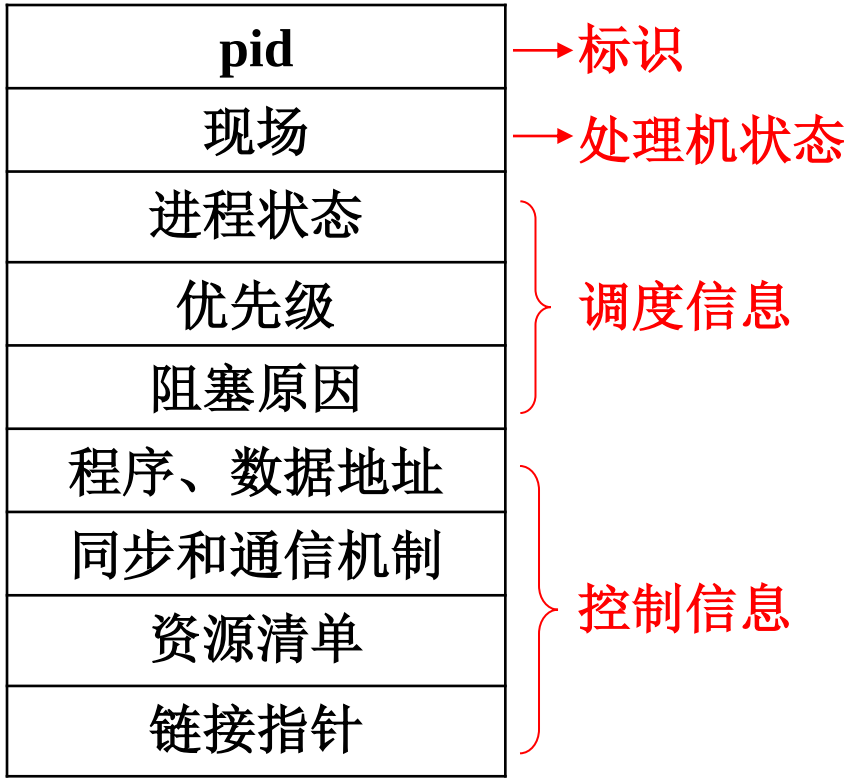
Suspend ←→Active

图2－6 具有挂起状态的进程状态图



2.1.5 进程控制块

- 进程控制块的作用
  - 描述和控制进程的运行，是进程存在的唯一标志，是操作系统中最重要的记录型数据结构
  - PCB(process control block)常驻内存
- 进程控制块中的信息
  - 标识、处理机状态，进程调度信息，进程控制信息



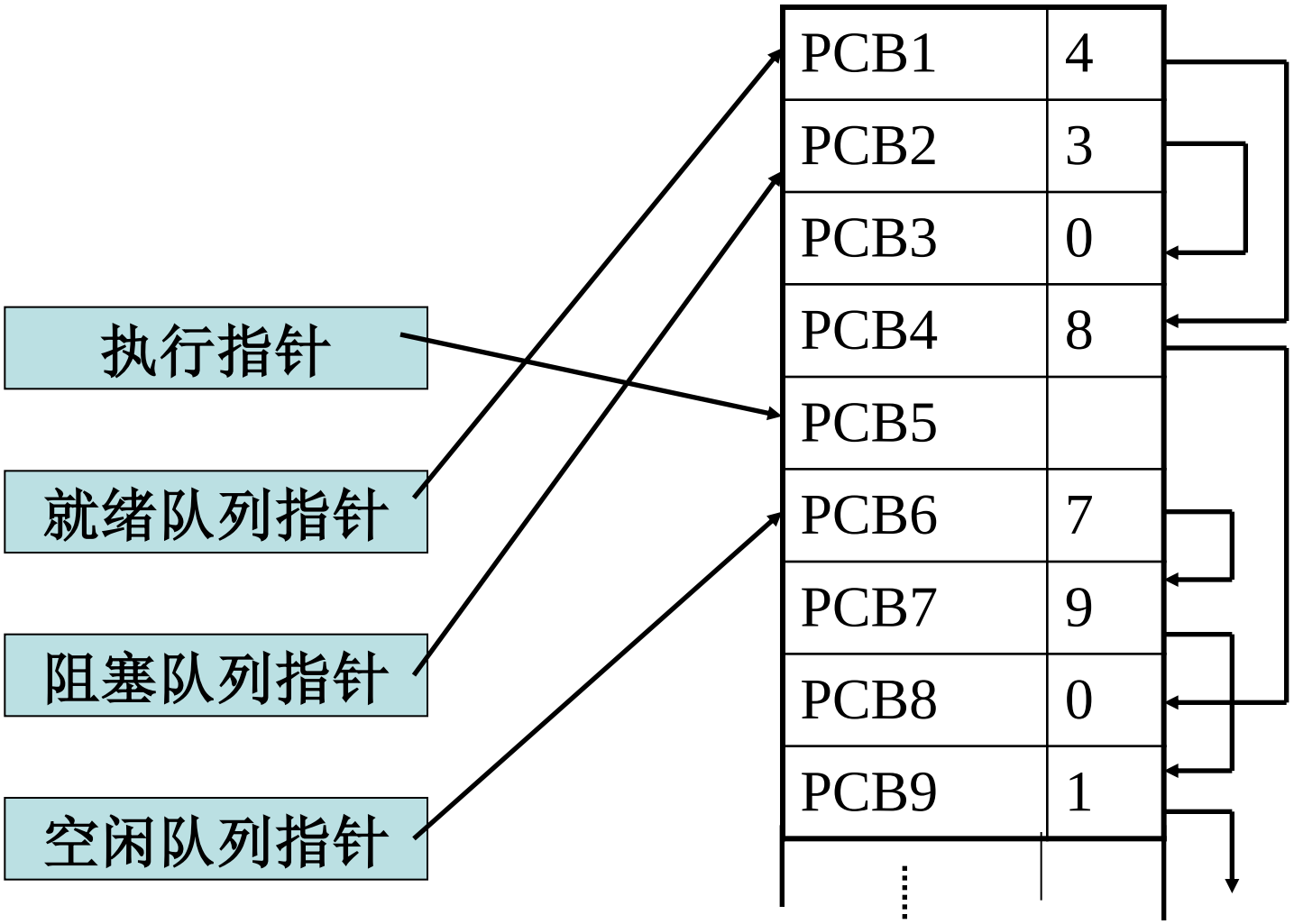
PCB示意图

❖ PCB的组织

系统中通常拥有大量的PCB，为了对它们加以有效的管理，必须以适当的方式将其组织起来，通常有以下组织方式：

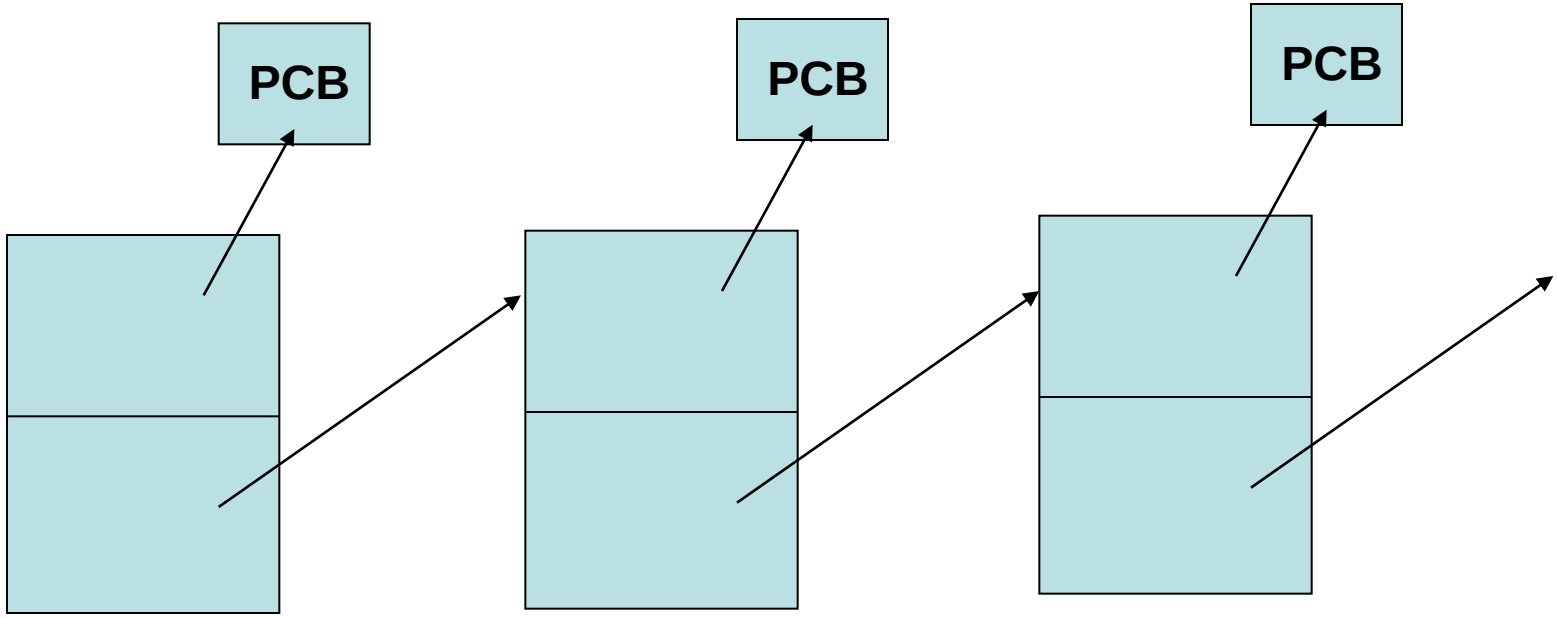
* 连接方式：把具有同一状态的PCB，用其连接指针连接成一个队列。
* 索引方式：根据进程状态建立索引表，所有表中存放某个PCB在总PCB表中的地址。

PCB链接队列示意图

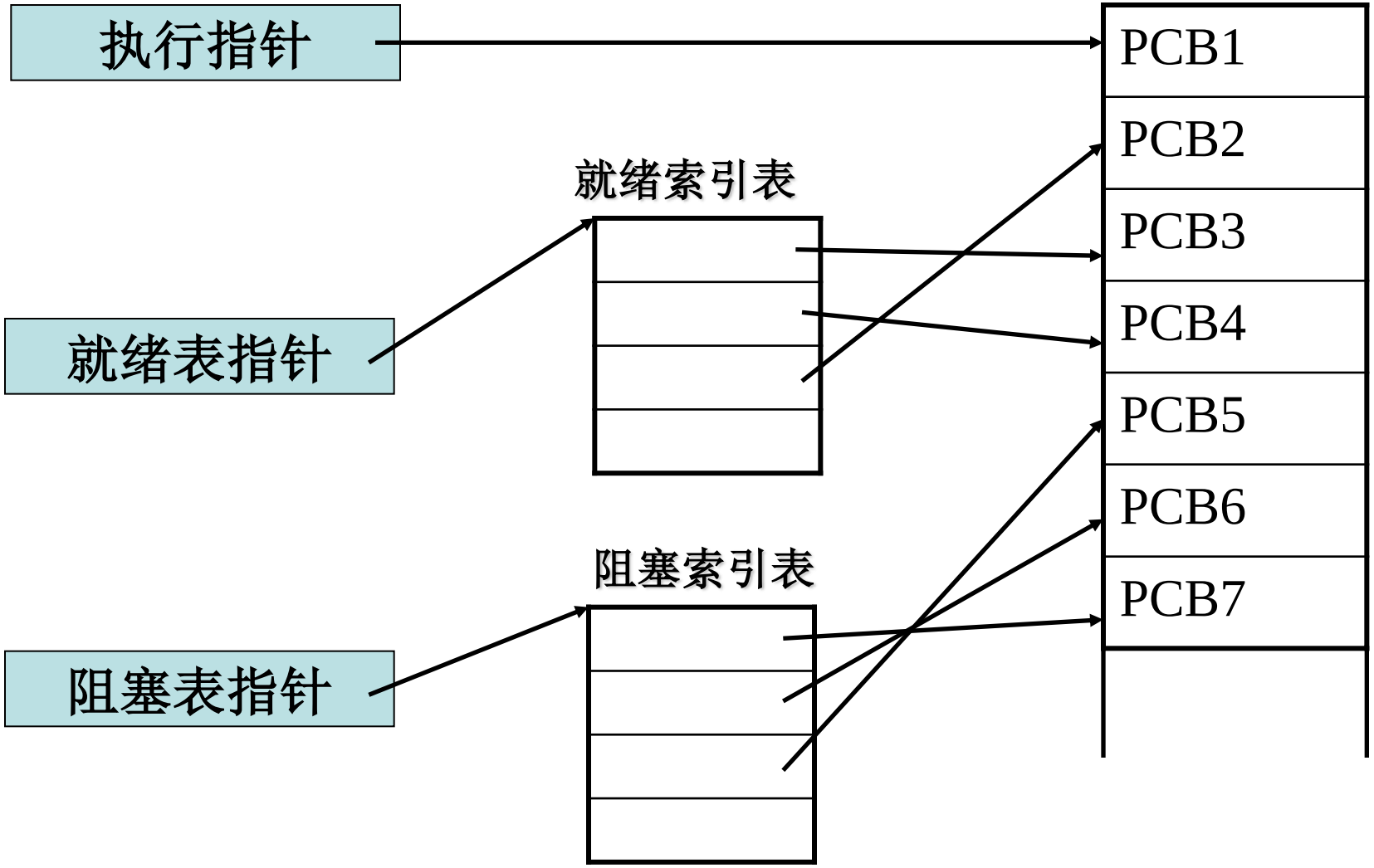

执行指针
就绪队列指针
阻塞队列指针
空闲队列指针
PCB1
4
PCB2
3
PCB3
0
PCB4
8
PCB5
PCB6
7
PCB7
9
PCB8
0
PCB9
1

等待队列示例

```
struct wait_queue {
  struct task_struct * task;
  struct wait_queue * next;
};
```



PCB索引方式示意图


执行指针
就绪表指针
阻塞表指针
就绪索引表
阻塞索引表
PCB1
PCB2
PCB3
PCB4
PCB5
PCB6
PCB7

补充

- PCB和进程的代码数据放在一起吗？
  - 系统态和用户态
  - 系统空间和用户空间
- 系统调用和普通调用的区别？
  - 系统调用会引起从用户态进入核心态

2.2 进程控制

进程控制是进程管理中最基本的功能，因此，进程控制一般由OS的内核来实现。

进程控制包括：

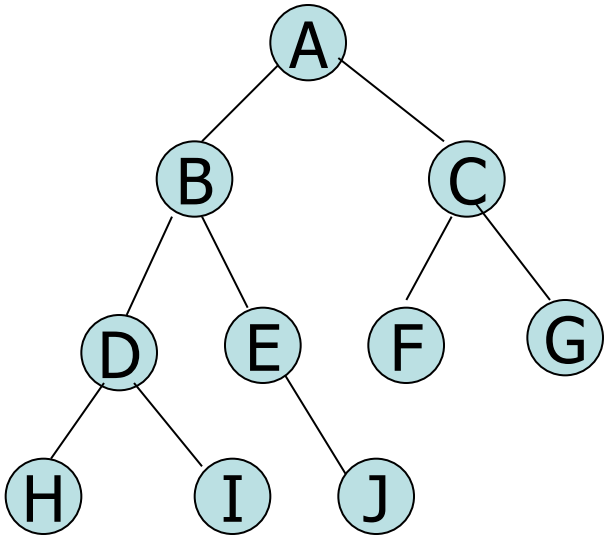
- 进程创建
- 进程终止
- 进程阻塞与唤醒
- 进程挂起与激活

- 系统采用的操作原语？
- 引起动作的事件？
- 动作的处理步骤？

2.2 进程控制—进程图(Process Graph)

- 定义：描述了进程的家族关系的有向树
- 基本概念：

  父进程、子进程、祖先进程、祖先
- 进程家族的作用：
  - 子进程可继承父的资源
  - 撤消子进程应将资源归还父进程
  - 撤消父进程会撤消全部子进程。



进程树（家族）

2.2.1 进程控制—进程创建

- 引起创建进程的四种事件
  - 用户登录：为终端用户建立一进程，放入就绪队列。
  - 作业调度：（不是进程调度）为被调度的作业建立进程，放入就绪队列。
  - 提供服务：系统创建进程为用户请求提供服务。
  - 应用请求：应用根据需要创建多个新进程以处理特定任务。

注意：前三事件进程由内核创建，后一事件进程由应用创建

2.2.1 进程控制—进程创建

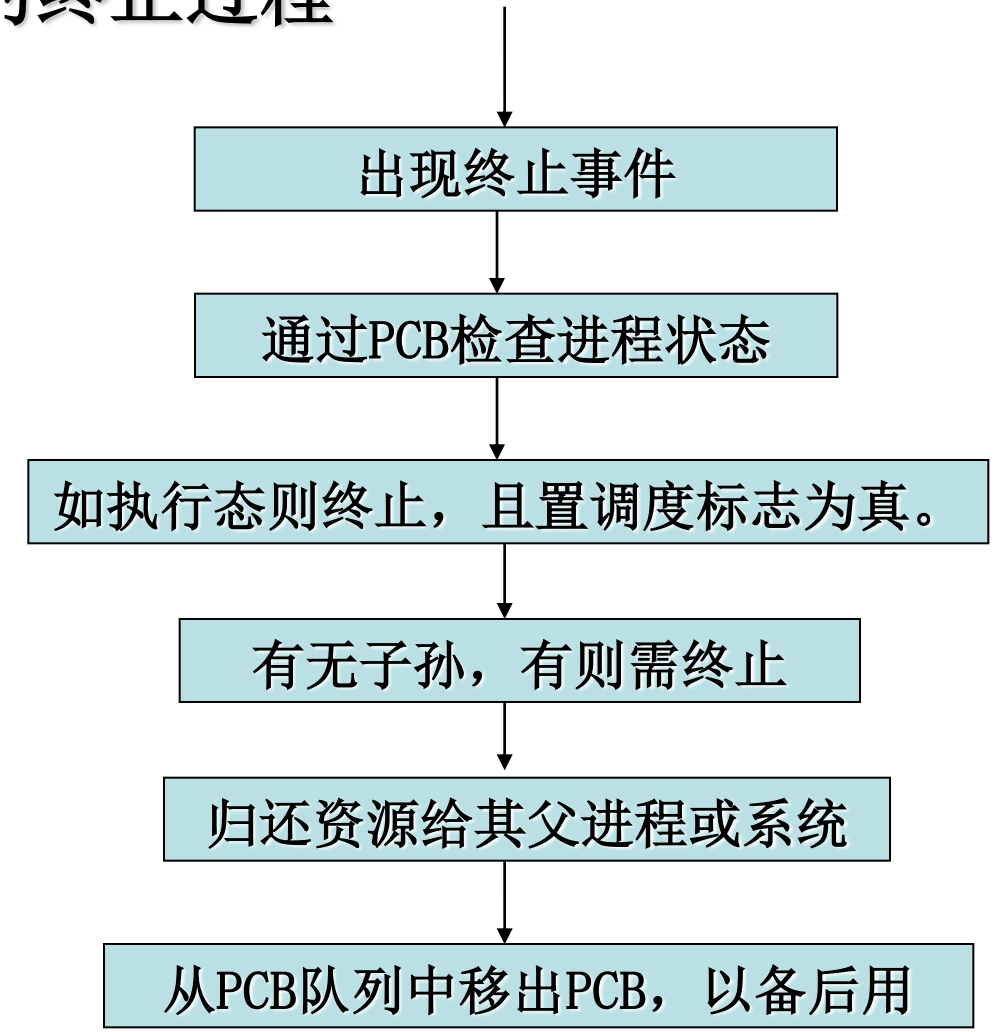
- 进程创建四步骤：（creat()原语）
  - 1. 申请空白PCB（一个系统的PCB是有限的），获取PCB数字标识符。
  - 2. 为新进程分配资源。即为进程的程序、数据、用户栈分配内存空间。
  - 3. 初始化PCB。
    - 标识信息：把自己和父进程的标识符写入PCB
    - 处理机状态：使程序计数器指向程序的入口地址
    - 处理机控制等信息：设置进程状态、优先级
  - 4. 将新进程插入就绪队列。

2.2.2 进程控制—进程的终止

- 引起进程终止的三种事件
  - 正常结束：如Halt、logoff
  - 异常结束：运行期间出现错误和故障而被迫终止。包括：①越界②保护错③非法指令④特权指令⑤运行超时⑥等待超时⑦算术运算错⑧ I/O故障
  - 外界干预：
    - a. 系统员kill进程；
    - b. 父进程请求；
    - c. 父进程终止。

2.2.2 进程控制—进程的终止

❖ 进程的终止过程



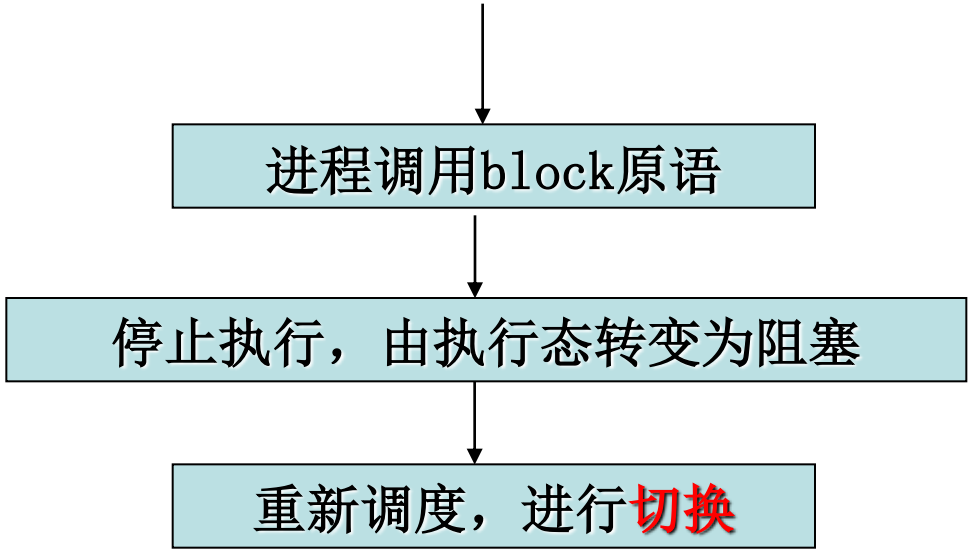
2.2.3 进程控制—进程的阻塞与唤醒

- 引起进程阻塞和唤醒的事件
  - 请求系统服务得不到满足时，如请求打印。
  - 启动某种操作而必须等待其完成时，如I/O。
  - 新数据尚未到达，如进程A写，进程B读，则A未写，完B不能读。
  - 无新工作可做：如数据传输进程

2.2.3 进程控制—进程的阻塞与唤醒

❖进程阻塞过程（block()原语）

当运行过程中发生上述事件，进程调用block原语来使自己阻塞，是进程自身的一种主动行为。



切换： 保留被阻塞进程的处理机状态(PCB)，并按新进程PCB中的处理机状态设置CPU环境。

2.2.3 进程控制—进程的阻塞与唤醒

- 唤醒过程（wakeup()原语）
  - 导致阻塞的事件完成后，有相关进程调用wakeup原语将等待该事件的进程唤醒。
  - 唤醒过程：修改PCB为就绪态，插入就绪队列

有block原语，其它进程中就必须有wakeup原语

2.2.4 进程控制—进程的挂起与激活

- 进程的挂起过程（suspend()原语）
  - 由进程自己或其父进程调suspend原语完成
  - 检查进程状态→更改状态→移动PCB
  - PCB移到指定内存区域,便于查询
  - 如被挂起进程正在执行，则重新调度。
- 进程的激活过程（active()原语）
  - 从外存调入内存→改变状态
  - 静止就绪→活动就绪 或 静止阻塞→活动阻塞

2.3进程同步

OS引入进程提高了资源利用率和系统吞吐量，但由于进程具有异步性，会造成混乱，尤其在争用临界资源时。进程同步就是为了解决上述问题，协调并发进程的执行次序，达到有效的资源共享和相互合作，使程序执行具有可再现性。

2.3.1 进程同步的基本概念

- 两种形式的制约关系
  - 资源共享关系：（进程间接制约）
    - 需互斥地访问临界资源。
  - 相互合作关系：（进程直接制约）
- 临界资源：（一次仅允许一个进程访问的资源）
  - 不可再现性源于临界资源没有互斥访问。

2.3.1 进程同步的基本概念

- 临界区：进程访问临界资源的那段代码。
- 访问临界资源的描述：

  进入区：检查有无进程进入

  退出区：将访问标志复位

  剩余区：除前三个区域外的其它代码

Repeat

Entry section

Critical section

Exit section

Remainder section

Until false

上周回顾

- 程序并发执行的三大特征，以及由此引出的问题。
- 进程的功能、组成、进程的状态图及进程的组织方式。
- 进程图（描述进程的家族关系）
- 进程控制：进程创建、终止、阻塞、唤醒、挂起与激活
  - 三要素：原语、事件、处理步骤！
- 进程同步的作用，程序的临界区

本节内容

- 同步机制的四规则
- 四种信号量机制
- 信号量机制的应用

2.3.1 进程同步的基本概念

❖同步机制的四规则

* 空闲让进（对临界区而言）
* 忙则等待（对临界区而言）
* 有限等待：应保证等待有限时间，不会产生死等。
* 让权等待：不能进入临界区的执行进程应放弃CPU，避免陷入忙等。

并发读写三规则

* 有写不能读
* 有读不能写

} 写是独占的

* 读可并发

2.3.2 信号量机制

1965年，荷兰学者Dijkstra提出了一种卓有成效的进程同步工具——信号量(Semaphores)机制。信号量机制被广泛地应用于单处理机、多处理机及计算机网络中。在不断地发展中，经历了整型信号量、记录型信号量、信号量集等机制。

- 每种机制如何操作？
- 解决了什么问题？
- 有什么不足？
- 信号量机制解决了所有同步准则吗？

2.3.2 信号量机制—整型信号量

利用整型变量来控制临界资源的访问，通过2个原子操作wait(s)—P和signal(s)—V来实现。

S的初值应为多少?

- 原子操作：执行时不可中断的操作
- 解决问题：实现了临界资源的互斥访问
- 操作过程：

```
Wait(s):   while s≤0 do no-op (s≤0，不断测试)
                s:=s-1;
Signal(s):  s:=s+1;
```

- 缺点：没有遵循“让权等待”准则，会使进程陷入“忙等”

2.3.2 信号量机制—记录型信号量

- 解决问题：采取了“让权等待”策略
- 缺点：只针对进程对一个(类)临界资源的共享
- 数据结构

```
type  semaphore=record
    value:integer;
    L: list of process;
End
```

Value：（资源信号量）表示资源数目

L:等待该类资源的进程链表

2.3.2 信号量机制—记录型信号量

❖ 操作过程

```
procedure wait(s)
var s: semaphore
begin
    s.value:=s.value -1;
    if s.value <0 them block (S.L)
End

procedure signal(S)
var s:semaphone
begin
    s.value:=s.vaule+1
    if s.value<=0 then wakeup(s.L)
end
```

s.value<0
绝对值表示
链表中阻塞
进程的数目。

2.3.2 信号量机制—AND型信号量

- 解决问题：在进程共享多个资源的情况下，避免系统发生死锁。
- 死锁：在无外力作用下的一种僵持状态。
- 基本思想：将进程在整个运行过程中需要的所有资源，一次全部分配给进程，使用完毕再一起释放。即对若干个临界资源的分配，采取原子操作方式。
  - 要么全分配，要么一个也不分配。

死锁示例

进程A，B中都包含两个对Dmutex和Emutex的操作， 即

```
process A:          process B:
wait(Dmutex);       wait(Emutex);
wait(Emutex);       wait(Dmutex);
```

若进程A和B按下述次序交替执行wait操作：

process A: wait(Dmutex); 于是Dmutex=0

process B: wait(Emutex); 于是Emutex=0

process A: wait(Emutex); 于是Emutex=-1 A阻塞

process B: wait(Dmutex); 于是Dmutex=-1 B阻塞

死锁

AND思想

2.3.2 信号量机制—AND型信号量

❖操作过程

```
Swait(s1,s2,...,sn)
  if s1≥1 and ...and sn ≥1 then
        for i:=1 to n do si:=si-1; endfor
  else
      如果si<1，则将进程放入等待队列
  end if

Ssignal(s1,s2,...,sn)
  for i:=1 to n do si:=si+1;
       移出所有等待Si的进程，并将其加入就绪队列
 endfor
```

4 信号量集

- 为提高效率而对AND信号进行扩充。采用AND型信号量机制一次只能对每种类型的资源申请一份。而不能解决一次要申请每种类型资源的多份的情况。
- 信号量集的优点：
  - 一次分配n个某类资源
  - 对资源设定下限值，当<下限值时，则不予分配。

4 信号量集

S为信号量，d为需求量，t为下限值。

```
Swait(S1, t1, d1, …, Sn, tn, dn)
    if  S1≥t1 and … and Sn≥tn then
        for i：=1 to n do
            Si：=Si-di;
        endfor
  else
    Place the executing process in the waiting queue of the first Si with Si＜ti
    and set its program counter to the beginning of the Swait Operation.
  endif

signal(S1, d1, …, Sn, dn)
  for i ：=1 to n do
    Si ：= Si+di;
    Remove all the process waiting in the queue associated with Si into  the
    ready queue
  endfor;
```

4 信号量集

❖ 信号量集的三种特例：

* Swait(S, d, d)：允许每次申请d个资源。
  当资源数少于d时，不予分配。
* Swait(s, 1, 1)：S>1，记录型信号量。
  S=1时，互斥型(整型)信号量。
* Swait(s, 1, 0)，可控开关，当S≥1时，允许进入，S<1时，不能进入。

2.3.3 信号量的应用—实现进程互斥

实现思想： 对某一临界资源设置一**互斥信号量mutex**，并设置其**初始值为1**，然后将各访问进程的临界区放在原子操作wait(mutex)和signal(mutex)之间。

示例：

```
process1:begin
        repeat
            wait(mutex);
            critical setion
            signal(mutex);
            remainder section
        until false;
        end
```

```
process2:begin
        repeat
            wait(mutex);
            critical setion
            signal(mutex);
            remainder section
        until false;
        end
```

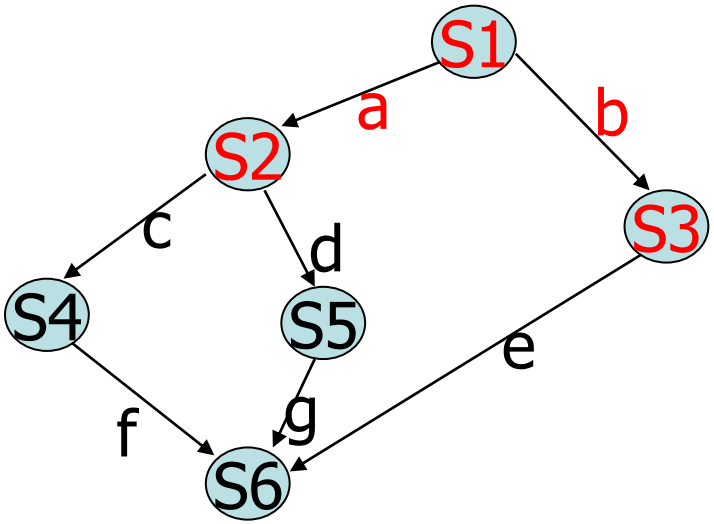
注意： 进程中wait(mutex)和signal(mutex)必须**成对出现**。**缺wait**则不能互斥，**缺signal**则资源不能释放

2.3.3 信号量的应用—实现前趋关系

实现方法： 描述程序或语句间前趋关系。使P1→P2关系共享一公用信号量S，初值为0；执行采用如下顺序。

P1 →signal(S) →wait(S) →P2

```
Var a,b,c,d,e,f,g:semaphore:=0,0,0,0,0,0,0;
Begin
    parbegin
      begin S1; signal(a); signal(b); end;
      begin wait(a);S2; signal(c); signal(d); end;
      begin wait(b);S3; signal(e); end;
      begin wait(c);S4; signal(f); end;
      begin wait(d);S1; signal(g); end;
      begin wait(e); wait(f);wait(g);S6; end;
    parend
end
```



语句前趋图举例

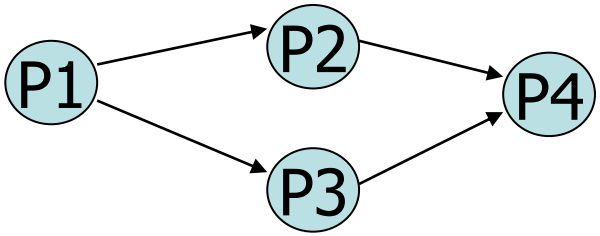
上节回顾

信号量进程同步机制

* 整型信号量机制、记录型信号量机制、AND型信号量机制、信号量集

信号量的应用

* 实现资源互斥访问
* 实现前趋关系



本节内容

➢经典进程同步问题

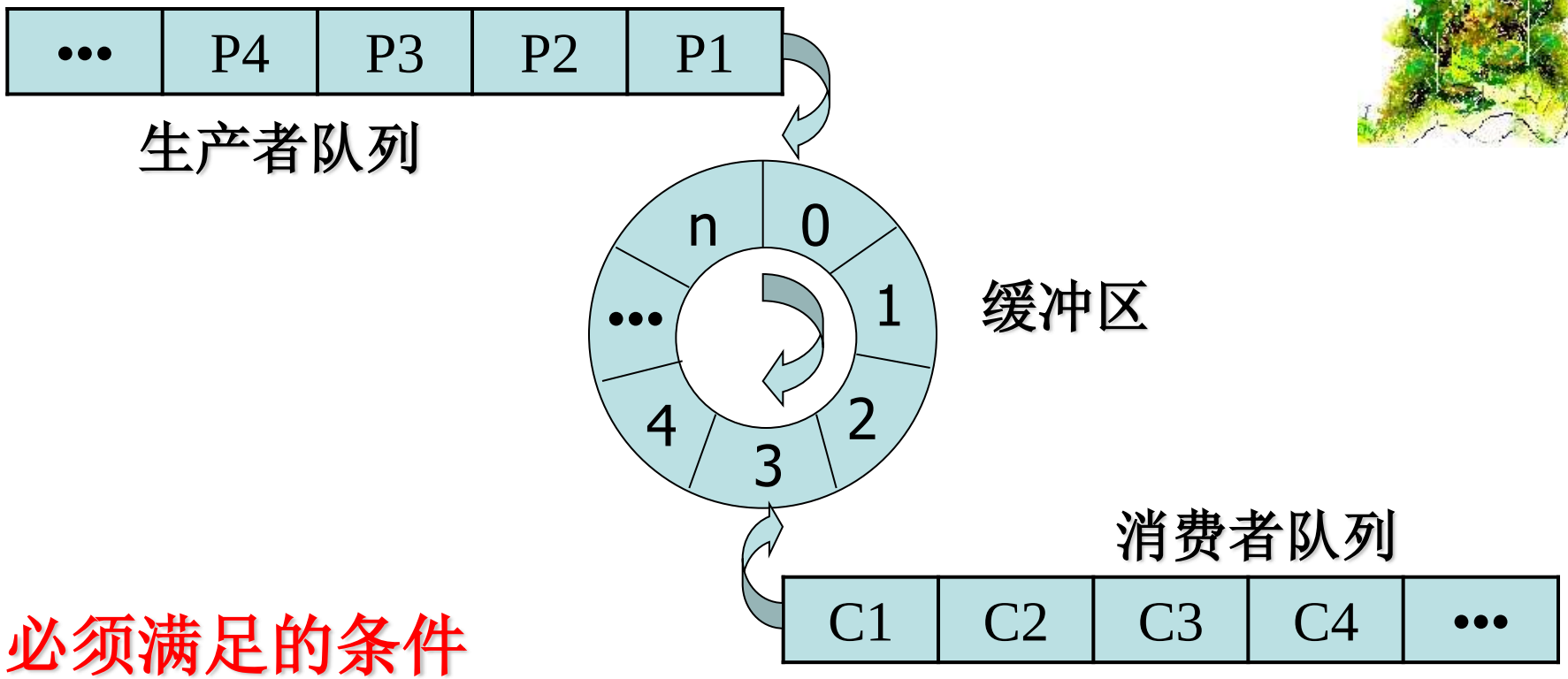
2.4 经典进程同步问题

进程同步协调并发进程的执行，达到有效的资源共享和相互合作，使程序执行具有可再现性；是保证多道程序环境下，程序正常运行的基本条件。在研究过程中，产生了一系列经典的进程同步问题，其中较有代表性的是以下三个问题：

- 生产者—消费者问题
- 哲学家进餐问题
- 读者—写者问题

2.4.1 生产者—消费者问题

问题的描述



必须满足的条件

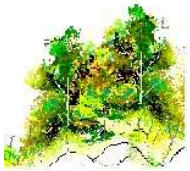
①所有进程采用异步方式运行；

②消费者不能去空缓冲区取商品；

③生产者不能向装满商品的缓冲区投放产品。

2.4.1 生产者—消费者问题

问题的产生与变量说明



Var n, integer;

Type item=…; 产品表示

var buffer:array[0,1,…,n-1] of item; n个缓冲池

in, out: 0,1, …, n-1; 输入\输出指针，初始化＝0

counter: 0,1,…,n; 当前产品数，初始化＝0

In 输入指针，指示下一个可投放产品的缓冲区，投放产品后in+1；in=(in+1) mod n

out输出指针，指示下一个可获取产品的缓冲区，消费产品后out+1；out=(out+1) mod n

基本进程实现

```
producer: (生产者进程)
repeat
    produce an item in nextp;
     while counter=n do no-op;
     buffer[in]:=nextp;
     in:=(in+1)mod n;
     counter:=counter+1;
until false;
```

```
consumer: (消费者进程)
repeat
     while counter=0 do no-op;
     nextc:=buffer[out];
     out:=(out+1) mod n;
     counter:=counter-1;
     consumer the item in nextc;
until false;
```

——共享变量为counter

(in+1)mod n=out 表示缓冲池已满

in=out 表示缓冲池已空

2.4.1 生产者—消费者问题

设counter的初值为5

生产者	消费者
register1:=counter;	register2:=counter;
register1 :=register1+1;	register2:=register2-1;
counter :=register1;	counter :=register2;

——————顺序执行

register1:=counter;	(register1:=5)
register1 :=register1+1;	(register1:=6)
register2:=counter;	(register2:=5)
register2 :=register2-1;	(register2:=4)
counter :=register1;	(counter:=6)
counter :=register2;	(counter:=4)

——————并行执行

显然counter为临界资源

一、利用记录型信号量解决生产者一消费者问题

mutex：使诸进程互斥地访问缓冲区（n个缓冲区）

empty、 full：空、满缓冲区数量。

```
Var mutex,empty,full:semaphore:=1,n,0;
    buffer:array[0,1,…,n-1] of item;
    in, out: integer: =0,0;
    begin
        parbegin
        producer:  begin
                        repeat
                            …
                        Produce an item in nextp;
                            …
```

一、利用记录型信号量解决生产者—消费者问题

```
wait(empty);
wait(mutex);
buffer(in):=nextp;
in:=(in+1) mod n;
signal(mutex);
signal(full);
Until false;
end
```

```
consumer:begin
    repeat
        wait(full);
        wait(mutex);
        nextc:=buffer(out);
        out:=(out+1) mod n;
        signal(mutex);
        signal(empty);
        Consumer the item in nextc;
    Until false;
    end
parend
end
```

注意互斥信号量mutex，资源信号量empty及full，其wait和signal的成对出现，及出现的顺序！

二、利用AND信号量解决生产者——消费者问题

实现技术

Swaitl(empty, mutex)→ wait(emptyl) 、 wait(mutex)

Ssignal(mutex, full)→signal(mutex)、signal(full)

Swaitl(full, mutex)→ wait(full) 、 wait(mutex)

Ssignal(mutex,empty)→signal(mutex)、signal(empty)

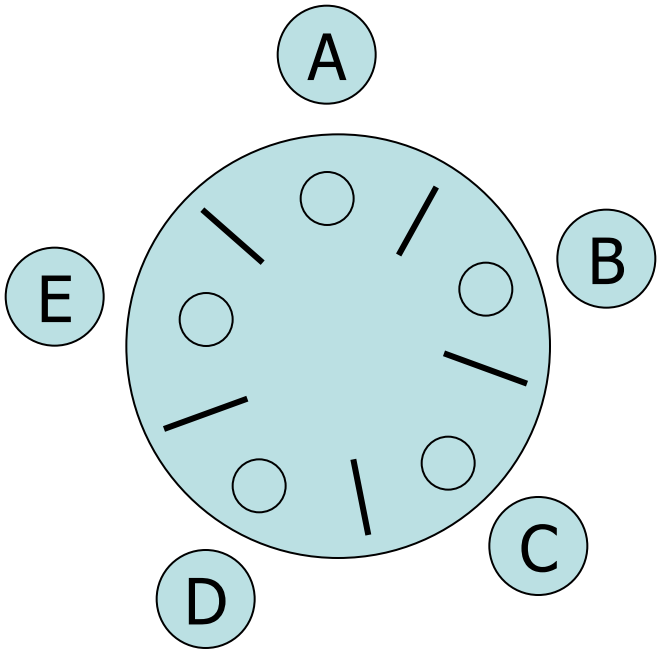
```
var  mutex, empty, full: semaphore:=1,n,0;
        buffer:array[0,…,n-1] of item;
        in out: integer :=0,0;
 begin
        parbegin
          producer: begin
                          repeat
```

```
produce an item in nextp;
        swait(empty, mutex);
         buffer(in):=nextp;
         in:=(in+1) mod n;
         ssingal(mutex, full);
   Until false;
   End
   Consumer: begin
     repeat
          swait(full, mutex);
          nextc:=buffer(out);
          out:=(out+1) mod n;
          ssignal(mutex, empty);
          consumer the item in nextc;
      until false;
   end
   parend
   end
```

2.4.2 哲学家进餐问题

问题的描述：

Dijkstra提出哲学家进餐问题。五位哲学家共用一张圆桌，圆桌上有5个碗和5只筷子。哲学家交替的思考和进餐，当进餐时，必须同时拿到左右两只筷子。进餐完毕，放下筷子继续思考。



2.4.2 哲学家进餐问题

❖利用记录型信号量解决哲学家进餐问题

显然，筷子是临界资源。为了实现对筷子的互斥使用，应为每一支筷子设置一个信号量，初始化为1。则第i位哲学家的活动可描述如下：

```
Var chopstick: array[0, …, 4] of semaphore :=(1,1,1,1,1);
Repeat
   wait(chopstick[i]);                    {拿左边的筷子}
   wait(chopstick[(i+1)mod 5]);           {拿右边的筷子}
   …
  eat;
   …
  signal(chopstick[i]);                   {释放左边筷子}
  signal(chopstick[(i+1)mod 5]);          {释放右边筷子}
   …
  think;
Until false
```

2.4.2 哲学家进餐问题

❖利用记录型信号量解决哲学家进餐问题的弊

会导致死锁。如五位哲学家同时饥饿时，将同时拿起左边的筷子，则都无法获得右边的筷子，而无限期的等待。

❖ 解决方法：

* 至多只允许有四位哲学家同时去拿左边的筷子，最终能保证至少有一位哲学家能够进餐。
* 仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐。
* 规定奇数号哲学家先拿他左边的筷子，然后再去拿右边的筷子；而偶数号哲学家则相反。

2.4.2哲学家进餐问题

❖ 利用AND信号量解决哲学家进餐问题

要求每个哲学家先获得两个临界资源(筷子)后方能进餐。

```
Var chopstick: array[0, …, 4] of
semaphore:=(1,1,1,1,1);
processi
Repeat
   think;
   Sswait(chopstick[(i+1)mod 5],chopstick[i]);
   eat;
   Ssignal(chopstick[(i+1)mod 5],chopstick[i]);
Until false
```

2.4.3 读者—写者问题

❖ 问题描述：

一个数据文件或记录，可被多个进程共享。读文件的进程称为“Reader进程”，写文件的进程称为“Writer进程”。

❖ 特点：

* 有写不能读
* 有读不能写
* 读可并发

❖ 问题核心：

保证一个写进程必须与其他进程互斥地访问共享对象

一、利用记录型信号量解决读者—写者问题

参数说明： wmutex为写互斥信号量； rmutex为多个读进程共享的readcount的信号量； readcount为正在读的进程数。

```
Var  rmutex, wmutex: semaphore: =1,1;
     readcount :integer: =0;
          begin
          parbegin
          reader: begin        {读进程开始}
                    repeat
                         wait(rmutex);
                         if readcount=0 then wait(wmutex); {互斥读写}
                          readcount:=readcount+1;
                         signal(rmutex);
                                   …
                         perform read operation
                                   …
                         wait(rmutex);
```

一、利用记录型信号量解决读者——写者问题

```
            readcount:=readcount-1;
            if readcount=0 then signal(wmutex); {读后释放写信号量}
            signal(rmutex);
        until false;
    end

    writer: begin  {写进程开始}
            repeat
              wait(wmutex);
              perform write operation;
              signal(wmutex);
            until false;
    end
  parend
end
```

二、信号量集解决读者——写者问题(略)

限制最多只允许RN个读进程同时读。

```
var RN integer;
    L, mx: semaphore: =RN, 1;
 begin
  parbegin
   reader: begin
             repeat
              swait(L,1,1);
              swait(mx,1,0);
              …
              perform read operation;
                    …
              ssignal(L,1);
            until false;
          end
```

有问题

```
writer: begin
          repeat
            swait(mx,1,1; L,RN,0);
            perform write operation;
            ssignal(mx, 1);
          until flase;
        end
  parend
end
```

Swait(mx,1,0)读开关，无写进程在写**mx=1**，则可执行读操作。

Swait(mx,1,1;L,RN,0)写开关，即无进程在写**mx=1**且无进程在读**L=RN**时，可执行写操作。

2.5 管程(Monitors)机制

- **引入原因：** 避免信号量机制下使用临界资源的进程都自备同步操作wait(s)和signal(s)的低效. **将同步操作的机制和临界资源结合到一起，形成管程。**
- **系统中各类资源均可用数据结构来抽象描述。**
  - 打印机（硬件资源）：busy,free等
  - FIFO（软件资源）：队长、队首、队尾以及一组操作

联系C++类
的定义过程

2.5 管程(Monitors)机制

- **管程定义：** 针对共享资源的一个数据结构和能为并发进程所执行的一组操作。

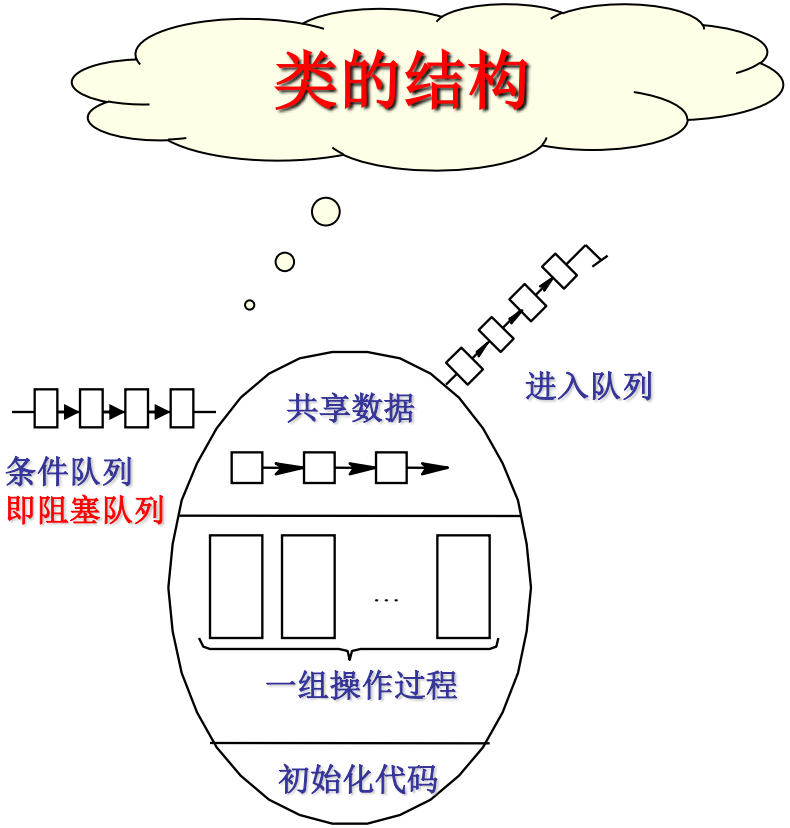
- **管程的构成**

  * 局部于管程的共享变量说明。（资源状态描述）
  * 对该数据结构进程操作的一组过程。（对资源的操作）
  * 对局部管程数据设置初值语句。（资源初始状态）

2.5 管程机制

❖ 管程示例

```
type monitor-name=monitor
variable declarations
procedure entry P1(…);
    begin … end;
procedure entry P2(…);
    begin … end;
    …
procedure entry Pn(…);
    begin … end;
begin
    initialization code;
end
```



管程示意图

进程对临界资源的访问必须通过管程，而管程每次只允许一个进程进入，从而实现进程互斥。

2.5 管程(Monitors)机制

- 利用管程实现同步机制，必须设置两个基本操作原语
  - *wait阻塞原语
  - *Signal释放原语

- 条件变量：用于根据不同阻塞条件调度阻塞队列。

  Var X:condition; （X可以根据具体情况进行设定）

  X.wait;

  X.Signal;

2.5.2 利用管程解决生产者—消费者问题

❖建立管程：PC

* 二过程

（1）put(item)：生产者将生产的产品放入缓冲池

（2）get(item)：消费者从缓冲池中取出一个产品

* 一变量：count≥n时满；≤0 时空

* 初始态：in=out=count=0

Type producer-consumer=monitor

var in,out,count：integer;

buffer: array [0,…,n-1] of item;

notfull, notempty: condition;

2.5.2 利用管程解决生产者—消费者问题

```
Procedure entry put(item)
    begin
      if count ≥n then notfull.wait;  {缓存池满则放入等待非满队列}
        buffer(in):=nextp;
        in:=(in+1)mod n
        count:=count+1;
        if notempty.queue then notempty.signal;
    end   {如等待产品不为空队列中有进程，则唤醒}
Procedure entry get(item)
    begin
      if count ≤ 0 then notempty.wait;
        nextc:=buffer(out);
        out:=(out+1)mod n
        count:=count-1;
        if notfull.queue then notfull.signal;
    end  {如等待仓库不满队列中有进程，则唤醒}
Begin in:=out:=0; count:=0 end
```

2.5.2 利用管程解决生产者—消费者问题

❖利用定义的管程解决生产者—消费者问题

```
 producer: begin
    repeat
      produce a item nextp
      PC. put (item);
    until false.
end

Consumer: begin
    repeat
      PC.get(item);
      Consume the item in nextc;
    Until false
end
```

可见，由管程来实现后，进程的同步更简单明了。

上周回顾

- 经典同步问题
  - 生产者—消费者问题
  - 哲学家进餐问题
  - 读—写者问题
- 进程同步的管程机制

本节内容

- 进程同步通信

2.6 进程通信

- 概念：进程间的信息交换。

- 实例：
  - 信号量机制（一种低级通信，信息量少）

    缺点：（1）效率低

    （2）通信对用户不透明
  - 高级通信（信息量大）

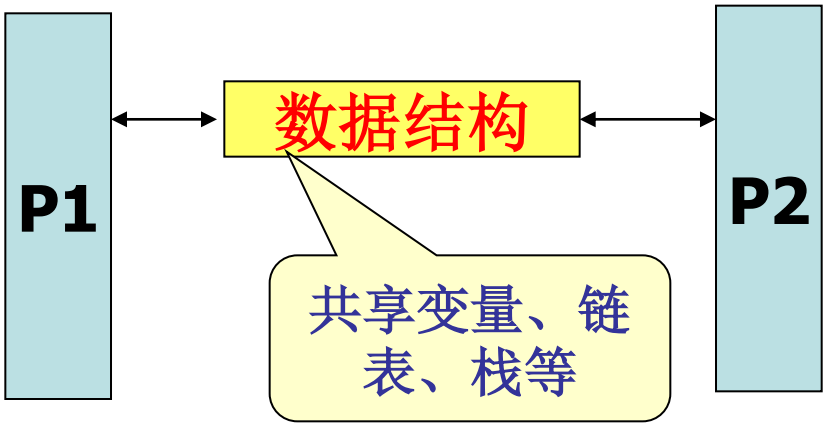
    特点：效率高，通信实现细节对用户透明
- 三种高级通信机制
  - 共享存储器系统
  - 消息传递系统
  - 管道通信系统

2.6.1 进程通信的类型

❖ 共享存贮器系统(Shared-Memory System)

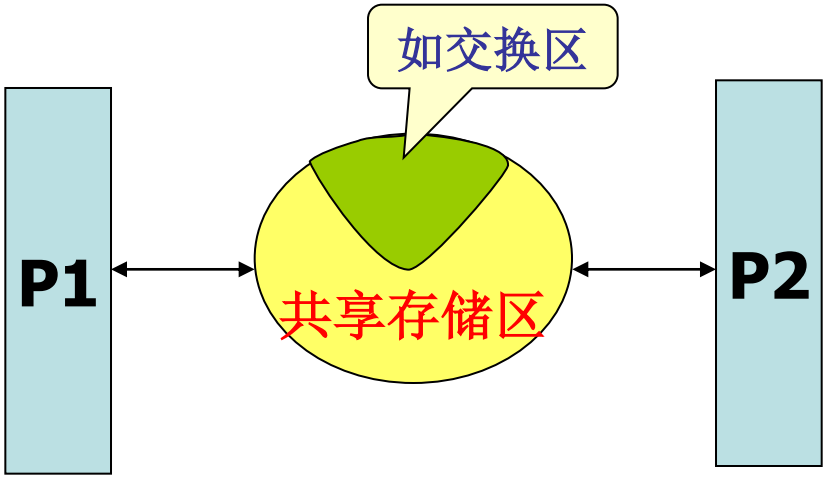
相互通信的进程共享某些数据结构或存储区，并通过这些空间实现通信。

* 基于共享数据结构的通信方式：
  - 如produce-consume中的缓冲区，低效，不透明。
  - 系统只提供了一共享存贮器，通信机制由程序原设计，适于少量通信。



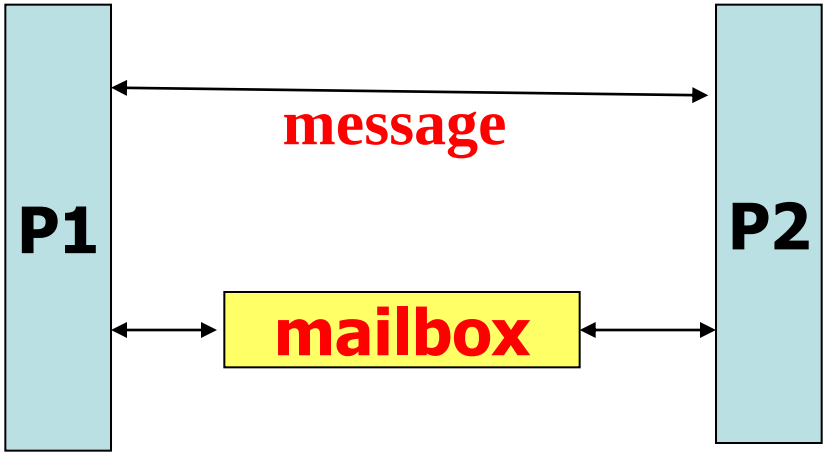
2.6.1 进程通信的类型

* 基于共享存储区的通信方式：
  - 系统提供：共享存储区。
  - 进程通信过程：
    - （1）向系统申请一个或多个分区
    - （2）获得分区获后即可读/写，实现通信.
  - 特点：高效，速度快。



2.6.1进程通信的类型

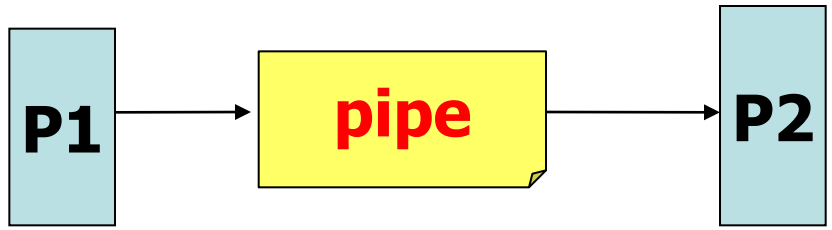
- 消息传递系统(Message Passing system)
  - 信息单位：格式化的消息（报文）
  - 可用于异种机，如计算机网络
  - 是目前的主要通信方式，分为直接通信方式、间接通信方式
  - 实现：一组通信命令(原语，如send,receive等)，具有透明性，内嵌了同步机制。



2.6.1进程通信的类型

❖ 管道通信(pipe)

- 管道：连接一个读进程和一个写进程之间通信的共享文件pipe。
- 功能：大量的数据传输。
- 管道必须的三方面协调能力：
  - 互斥：同一时间只能由一个进程对管道进行读写
  - 同步：规定数据大小，读写进程以该数据量交替访问管道
  - 对方是否存在

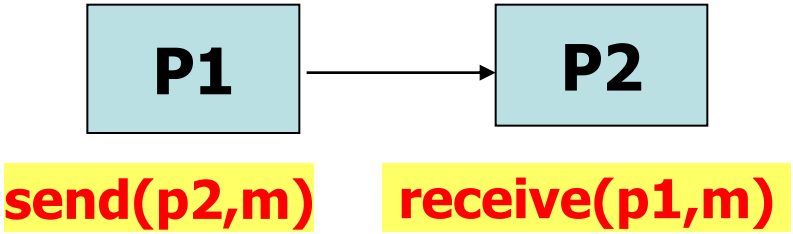


2.6.2 直接/间接通信方式（消息通信的2种方式）

❖ 直接通信方式（实时通信）

发送进程利用OS提供的发送命令，直接把消息发送给目标进程。发送和接收方以显式方式提供对方标识符。

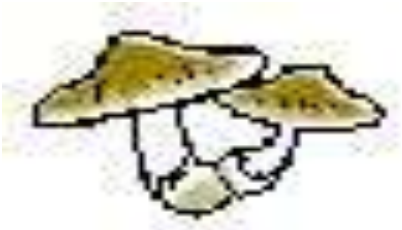
- send(Receiver, message)
- receive(Sender, message)



❖ 例：解决生产—消费问题。

```
repeat
        …
    produce an item in nextp;
        …
    send(consumer, nextp);
until false;
```

```
repeat
    receive( producer, nextc);
        …
    consumer the item in nextc;
until false;
```



2.6.2直接/间接通信方式（消息通信的2种方式）

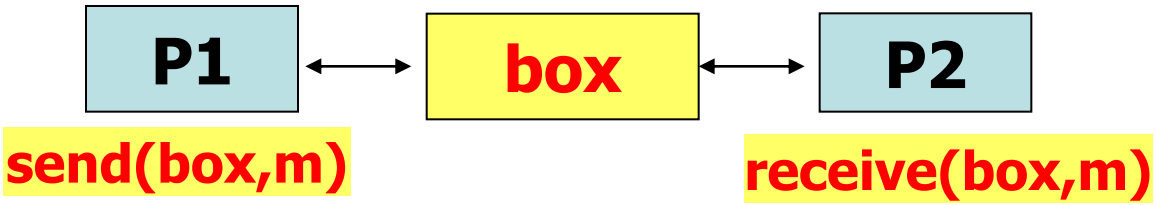
- 间接通信方式（可以实现非实时通信）

  通过共享的数据实体（信箱），来暂存通信的内容。

  - 优点：在读/写时间上的随机性

- 系统提供了用于控制信箱的各种原语

  - （1）信箱的创建与撤消：
    - 信箱名 属性（公用、私用、共享）（共享者名字）
  - （2）消息的发送和接收
    - Send (mailbox, message)
    - Receive (mailbox, message)



2.6.2直接/间接通信方式（消息通信的2种方式）

- 信箱类型（可由操作系统或用户创建）
  - 私用信箱：用户进程创建，拥有者有读/写权，其它进程只有写权，单向通信方式。存在期＝进程存在期。
  - 公用信箱：系统创建，所有核准进程可以使用，双向通信方式。存在期=系统存在期。
  - 共享信箱：一般进程创建，并指明其共享者，由创建者和共享者共同使用。双向通信。
  - 发送—接收进程之间的关系：
    - 一对一关系:专线通信方式
    - 多对一关系:客户/服务器方式
    - 一对多关系:广播方式
    - 多对多关系：公用信箱

2.6.3 消息传递系统中的几个问题

- 通信链路(communication link)
  - 显式建立：进程完成(计算机网络)
  - 隐式建立：系统完成(单机系统，利用原语完成)
  - 链路类型
    - 由连接方法分：点—点链路，多点链路。
    - 由通信方式分：单向、双向。
    - 由容量分：无容量（无缓冲区，不能暂缓信息）、有（有缓冲区）。

2.6.3消息传递系统中的几个问题

- 消息格式(消息头＋消息正文)
  - 消息头（控制信息）：收/发进程名，消息长度、类型、编号等
  - 定长消息：系统开销小，用户不便（分组）。
  - 变长消息：开销大，用户方便（报文）。
- 进程同步三种方式（协调进程通信）
  - 发送和接收进程阻塞（汇合）：用于紧密同步，无缓冲区时，阻塞直到有消息传递再唤醒。
  - 发送进程不阻塞，接收进程阻塞（多个）：如打印进程等待打印任务。
  - 发送/接收进程均不阻塞：一般在发、收进程间有多个缓冲区。

2.6.4 消息缓冲队列通信机制

由美国Hansan提出，发送进程利用Send将消息直接发送给接收进程，并挂在接收进程的缓冲队列上，接收进程通过Receive获取消息。

类似于生产者—消费者问题

❖数据结构

* 消息缓冲区

  type message buffer =record
  sender: 发送者进程标识符
  size: 消息长度
  text: 消息正文
  next: 指向下一消息缓冲区指针

* PCB中应增数据项：增加对于消息队列进行操作和实现同步的信号量。

  type pcb=record
  mq 消息队列（临界资源）首指针
  mutex 消息队列互斥信息量
  sm 消息队列资源信息量（表资源消息数）

2.6.4 消息缓冲队列通信机制

❖ 发送原语基本语法

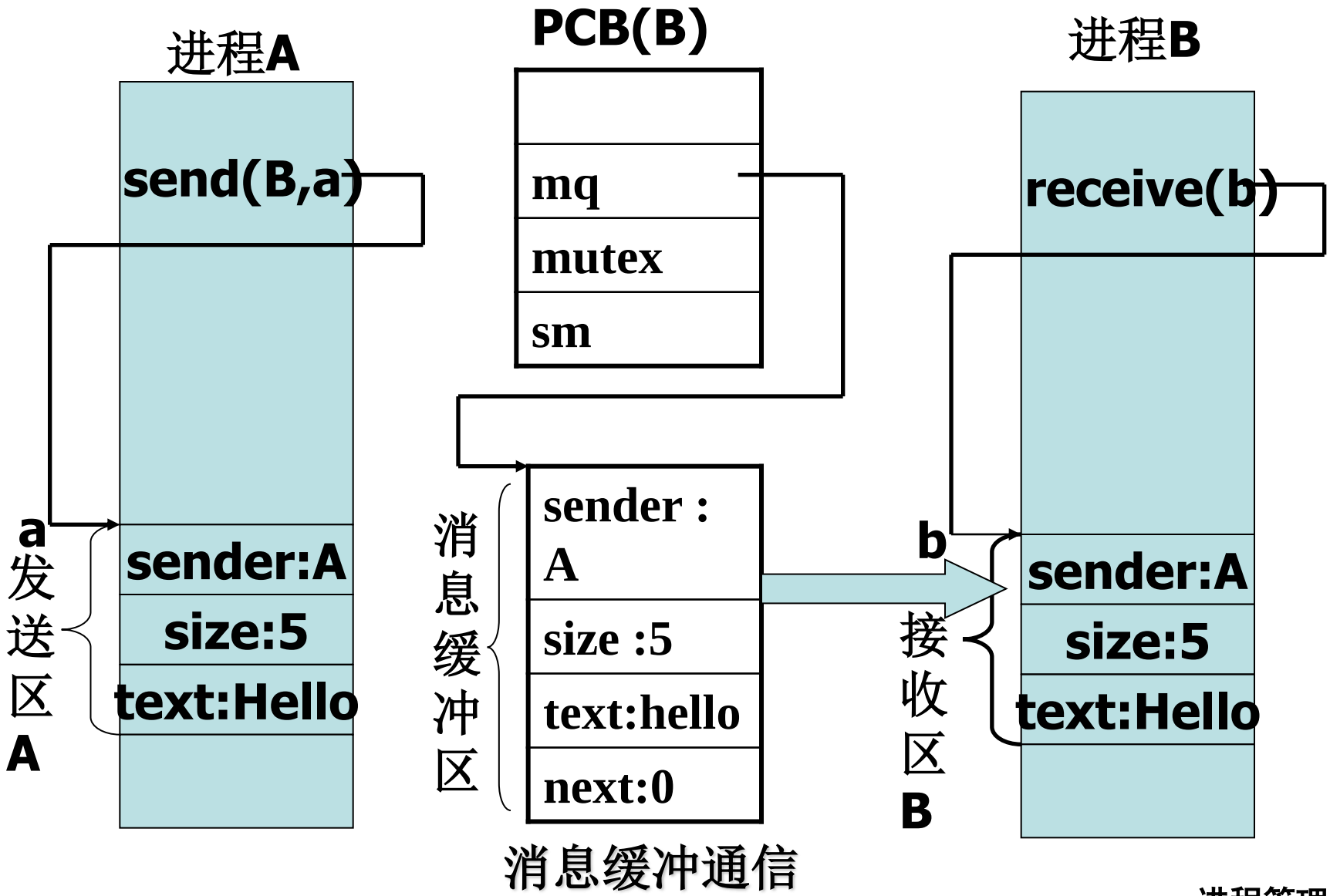
```
procedure send(receiver, a)
  begin
    getbuf(a.size, i);          根据a.size申请缓冲区
    i.sender:=a.sender;         将发送区a中信息复制到消息缓冲区
    i.size:=a.size;
    i.text:=a.text;
    i.next:=0;
    getid(PCB set, receiver.j);  获得接收进程内部标识符
    wait(j.mutex);
    insert(j.mq, i);            将消息缓冲区插入消息队列
    signal(j.mutex);
    signal(j.sm);               使资源信号量＋1
end
```

2.6.4 消息缓冲队列通信机制

❖接收原语的基本语法

```
procedure receive(b)
    begin
     j:=internal name;  j为接收进程内部标识符
     wait(j.sm);            使资源信号量－1
     wait(j.mutex);
     remove(j.mq, i);     将消息队列中第一个消息移出
     signal(j.mutex);
     b.sender:=i.sender;将消息缓冲区i中的信息复制到接收区b中
     b.size:=i.size;
     b.text:=i.text;
    end
```

2.6.4 消息缓冲队列通信机制



消息缓冲通信

上节回顾

- 进程通信
  - 共享存储器系统、消息传递系统、管道
  - 重点
    - 消息传递系统：直接通信方式、间接通信方式
    - 消息缓冲队列通信机制

本节内容

- 线　程
  - 基本概念
  - 线程的同步和通信
  - 内核支持线程和用户线程

2.7 线程

自从20世纪60年代提出进程概念后，OS中一直都是以进程作为资源分配和独立运行的基本单位。80年代，提出了比进程更小的能独立运行的基本单位—线程(Threads)。90年代，多处理机系统得到迅速发展，由于线程能比进程更好的提高程序的并行执行度，充分发挥多处理机的优越性，从而得到了广泛应用。

- 为什么引入线程，与进程作比较？
- 如何引入线程？
- 线程的基本控制技术？

2.7.1 线程的基本概念

❖线程引入的原因

对比进程

- 减少并发执行时的时空开销，进程的创建、撤消、切换较费时空，因它既是调度单位，又是资源拥有者。
- 线程实现了资源分配与调度的分离，以“轻装上阵”。线程是系统独立调度和分派的基本单位，其基本上不拥有系统资源，只有少量资源（IP，寄存器，栈），但共享其所属进程所拥有的全部资源。
- 线程满足了多处理机系统（SMP）的要求，能有效的改善多处理机系统的性能

2.7.1 线程的基本概念

❖2. 线程的属性

- 轻型实体
- 独立调度和分派的基本单位
- 可并发执行
- 共享进程资源

❖3. 线程的状态

- 状态参数：①寄存器状态②堆栈③运行状态④优先级⑤线程专有存储器⑥信号屏蔽
- 线程运行状态：①执行状态②就绪状态③阻塞状态

2.7.1 线程的基本概念

❖ 线程的创建和终止

* 初始化线程

程序启动 —创建→ 初始化线程 —创建函数→ 其它线程

* 生命期（正常终止、强行终止）
  - 线程终止后并不立即释放其资源。

线程终止 —分离函数→ 与其它线程分离 —释放资源→ 其它线程使用

* 已终止而未释放资源的线程，仍能被其他线程调用而重新执行。

调用者线程 —连接命令→ 终止线程重新执行

2.7.1 线程的基本概念

❖多线程OS中进程的属性

- 系统资源分配的基本单位
- 可包括多个线程（至少要有一个）
- 不再是一个开执行的实体
  - 就绪
  - 执行
  - 阻塞
  - 挂起

如何理解进程状态

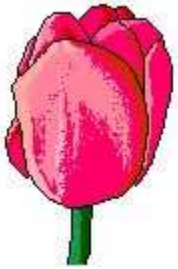
2.7.2 线程的同步和通信

为使系统中的多个线程能有条不紊地运行，必须提供用于线程间同步和通讯的机制。为支持不同频率的交互操作和不同程度的并行性，常提供了多种同步机制。本节主要介绍三种机制：互斥锁、条件变量、信号量机制。

❖互斥锁（适于高频度的资源互斥访问）

整形信号量

* 阻塞方式：lock(mutex)→访问→unlock(mutex)
* 非阻塞方式：开锁状态则成功，锁定则返回什么也不做
  - if（trylock）then
    operation
    else
    do nothing

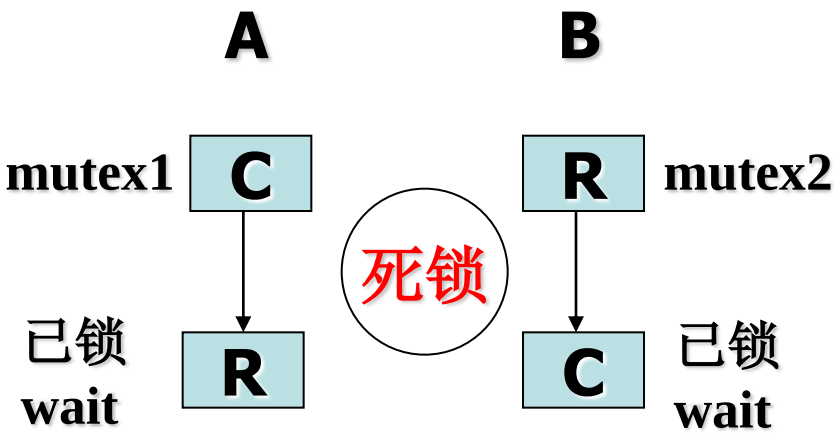


2.7.2 线程的同步和通信

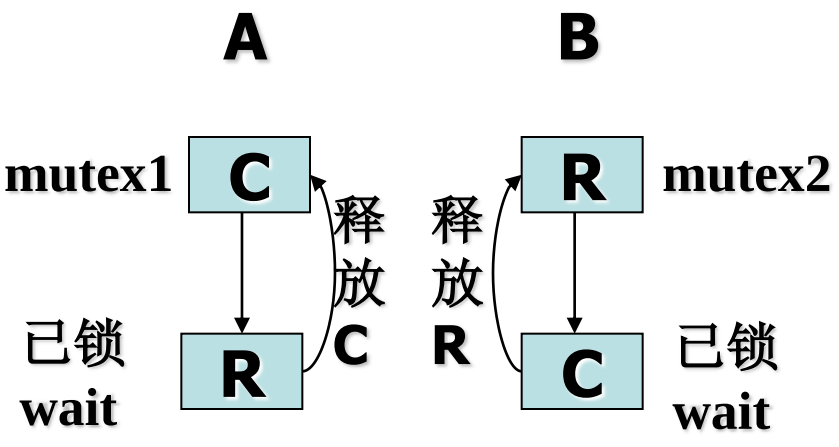
❖ 条件变量

仅利用互斥锁机制将可能引起死锁，为了克服该不足，引入了条件变量。每一个条件变量都与一个互斥锁一起使用。

❖主要思想：当请求资源被阻塞时，线程释放以前占有的资源。



互斥锁示例



条件变量示例

2.7.2线程的同步和通信

❖ 信号量机制

为提高效率，可分别为线程和进程设置相应的信号量。

* 私用信号量（private semaphore）
  - 作用域在一个进程中，由进程内的线程使用
  - 存放在程序的地址空间
  - OS不知道其存在，不便于回收
* 公用信号量（public semaphore）
  - 作用于多个进程间或不同进程的线程间使用
  - 存放在受保护的系统存储区
  - 有操作系统创建并管理回收

2.7.3 内核级和用户级线程

- 对于进程，无论系统进程还是用户进程，其操作都是通过内核来完成的：系统调用→内核→处理程序
- 两种线程实现方式：用户级线程(User-Level Threads)
  内核支持级线程(Kernel-Supported Threads)
- 用户级线程：不依赖于内核，TCB存在于用户空间，内核不知道其存在。
- 内核级线程：依赖内核，其创建、撤消和切换都由内核实现，存在于内核空间，通过线程控制块TCB控制。

2.7.3 内核级和用户级线程的比较

❖ 线程的调度和切换速度

- 内核级线程切换类似于进程切换，但速度快于进程切换。
- 用户级线程切换通常发生在同一用户进程的诸线程间，无需进入内核，速度更快。

❖ 系统调用与执行速度

- 用户级线程调度以进程为单位，以进程为单位平均分配时间，对线程间并发执行并不有利。拥有多线程的进程，在其一个线程发生系统调用时，将阻塞该进程。
- 内核支持线程调度以线程为单位，以线程为单位平均分配时间，对于拥有多个线程的进程而言非常有利。拥有多线程的进程，在其一个线程发生系统调用时，仅阻塞该线程，而进程继续执行。

2.7.4 线程控制

无论是进程还是线程，都必须直接或间接地取得内核地支持。内核级支持线程可直接调用，而用户级线程必须借助中间系统来取得内核的服务。

❖内核支持线程的实现

创建进程 →

PTDA	进程资源
TCB	#1
TCB	#2
TCB	#3
…	…

任务数据区

TCB保存在内核空间中。内核线程的创建、撤销、调度与切换均于进程相类似。

2.7.4 线程控制

❖用户级线程的实现

所有用户级线程都有相同的结构，都运行在一个中间系统之上。当前的中间系统有：运行时系统、内核控制线程。

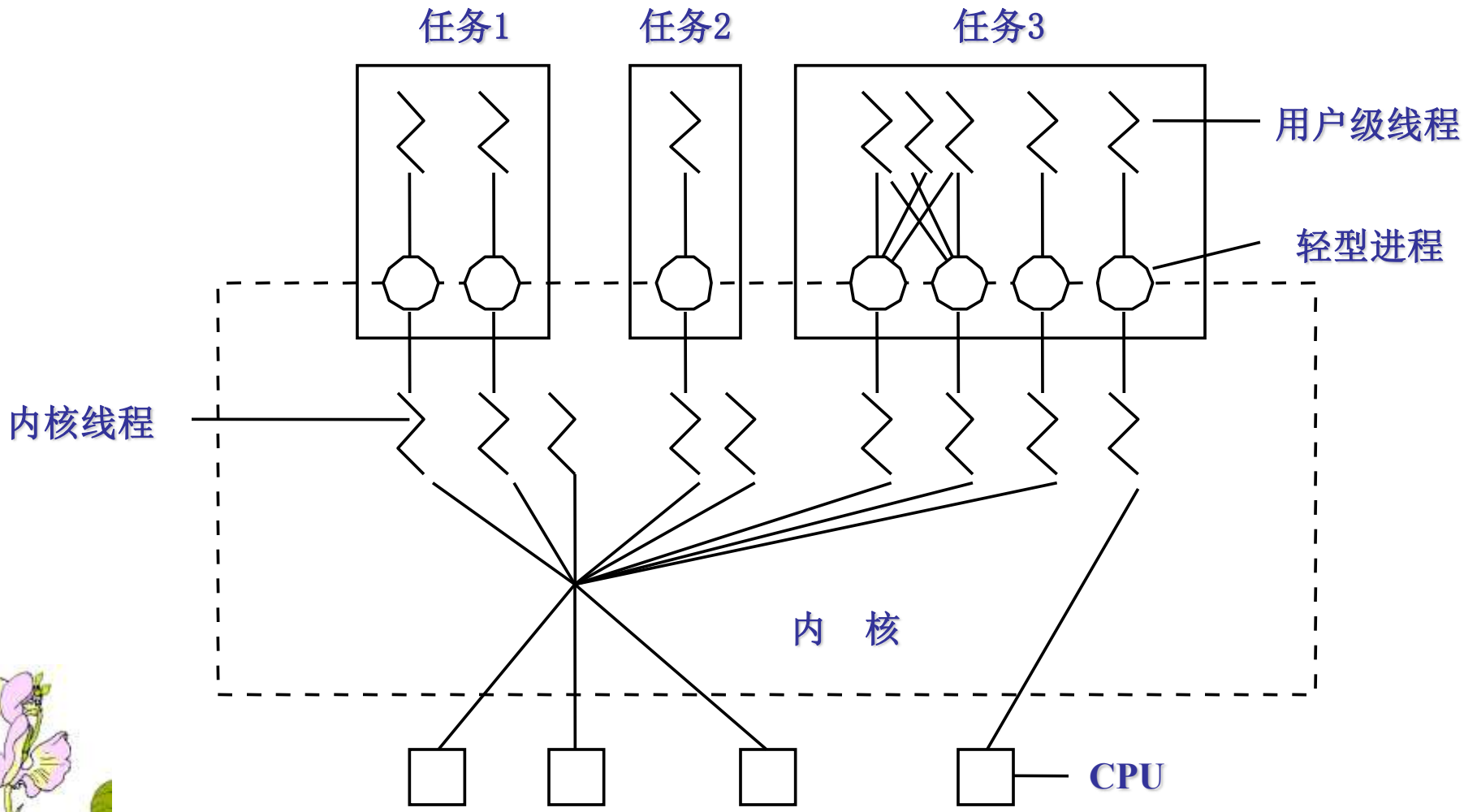
❖运行时系统(Runtime System)

* 概念：用于管理和控制线程的函数(过程)的集合。
* 运行时系统驻留在用户空间，作为用户级线程与内核的接口。
* 线程切换不用进入内核，非常迅速。

2.7.4 线程控制

- 内核控制线程（轻型进程LWP）
  - LWP通过系统调用来获得内核服务，一个用户级线程只要连接上一个LWP，就能调用内核服务，就能与内核进行通信；LWP称为用户线程与内核的桥梁。
  - 一个进程可拥有多个LWP
  - 多个用户级线程可以复用一个LWP
  - 每个LWP连接一个内核级线程

2.7.4 线程控制



利用轻型进程作为中间系统

2.7.4 线程控制

- 引入LWP后进程操作情况
  - 内核线程阻塞→LWP阻塞→用户线程阻塞
  - 如进程仅拥有一个LWP，则此时进程阻塞
  - 如进程拥有多个LWP
    - 当一个LWP阻塞，其他LWP继续执行；
    - 如所有LWP都阻塞，则进程中的线程仍能执行，但不再能访问内核。