计算机操作系统

第二章　进程的描述与控制

第二章 进程的描述与控制

- 2.1 前趋图和程序执行
- 2.2 进程控制
- 2.3 进程同步
- 2.4 经典进程的同步问题
- 2.5 进程通信
- 2.6 线程

目的及要求

- 领会程序顺序执行和并发执行的特征
- 掌握进程的概念和特征、进程的基本状态及转换
- 理解进程控制块PCB的作用、包含信息和组织方式

一、程序的顺序执行及其特征

1.程序的顺序执行

一个程序由若干个程序段组成，而这些程序段的执行必须是顺序的，这种程序执行的方式就称为程序的顺序执行。

S1: a:=x+y；

S2: b:=a-5；

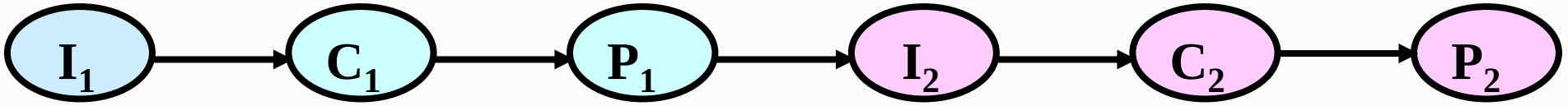
S3: c:=b+1；

例：在进行计算的时候：

首先输入用户的程序和数据 Input

然后进行计算　　Caculate

最后打印计算结果　Print



即有三个顺序执行的操作

I：输入操作

C：计算操作

P：输出操作

2. 程序顺序执行时的特征

（1）顺序性

处理机的操作严格按照程序所规定的顺序执行。

（2）封闭性

程序一旦开始执行，其计算结果不受外界因素的影响。

（3）可再现性

程序执行的结果与它的执行速度无关(即与时间无关)，而只与初始条件有关。

二、前趋图

前趋图是一个有向无循环图（DAG），用于描述进程之间执行的前后关系

结点：表示一个程序段或进程，或一条语句

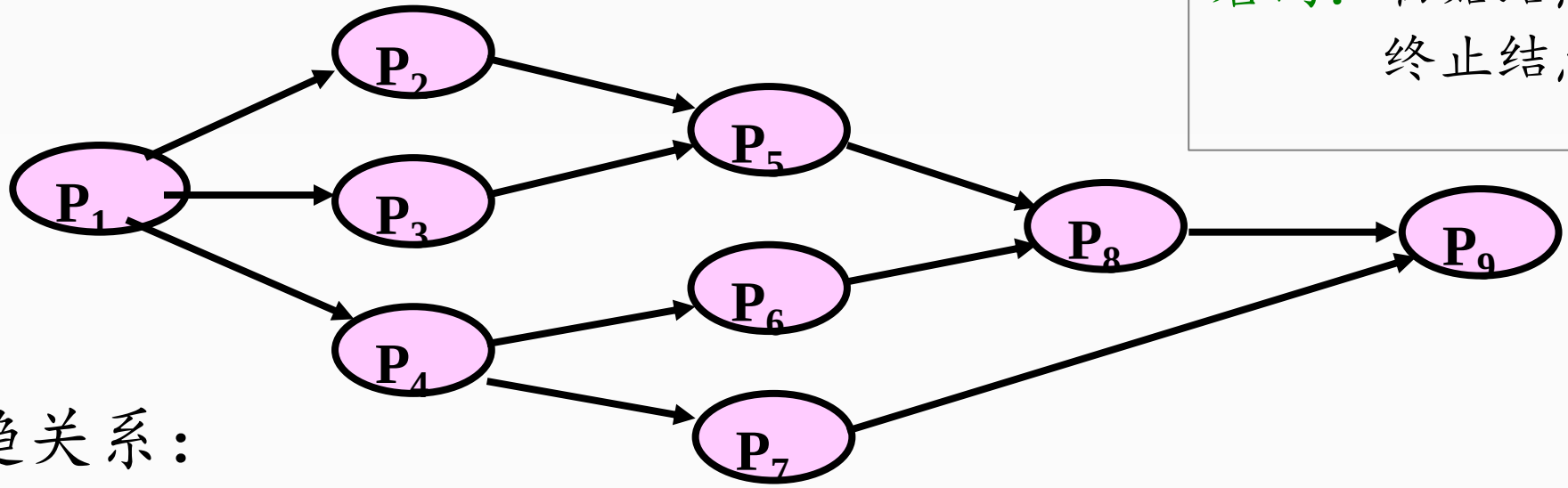
有向边：结点之间的偏序或前序关系“→”

$\rightarrow=\{(Pi,Pj)|$ Pi must complete before Pj may start$\}$

若$(Pi,Pj)\in\rightarrow$, 记为$Pi\rightarrow Pj$，则

Pi是Pj的直接前趋，Pj是Pi的直接后继

名词：初始结点；
终止结点；　重量



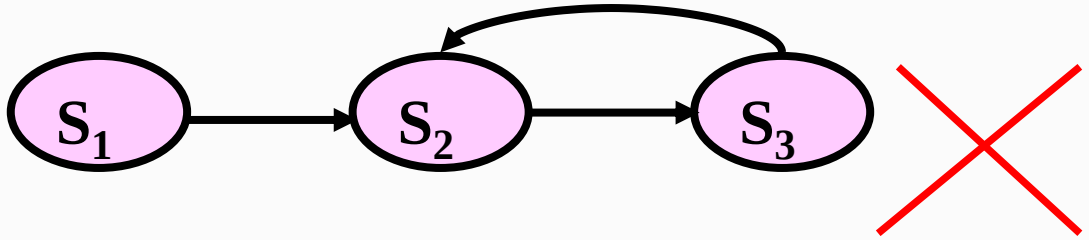
前趋关系：

P1→P2，P1→P3，P1→P4，P2→P5，P3→P5，P4→P6，P4→P7，P5→P8，P6→P8，P7→P9，P8→P9

或 P={P1，P2，P3，P4，P5，P6，P7，P8，P9}

→={(P1,P2)，(P1,P3)，(P1,P4)，(P2,P5)，(P3,P5)，(P4,P6)，(P4,P7)，(P5,P8)，(P6,P8)，(P7,P9)，(P8,P9)}

注意：前趋图中禁止存在循环



前趋关系：$S_2 \rightarrow S_3$，　$S_3 \rightarrow S_2$，不可能满足

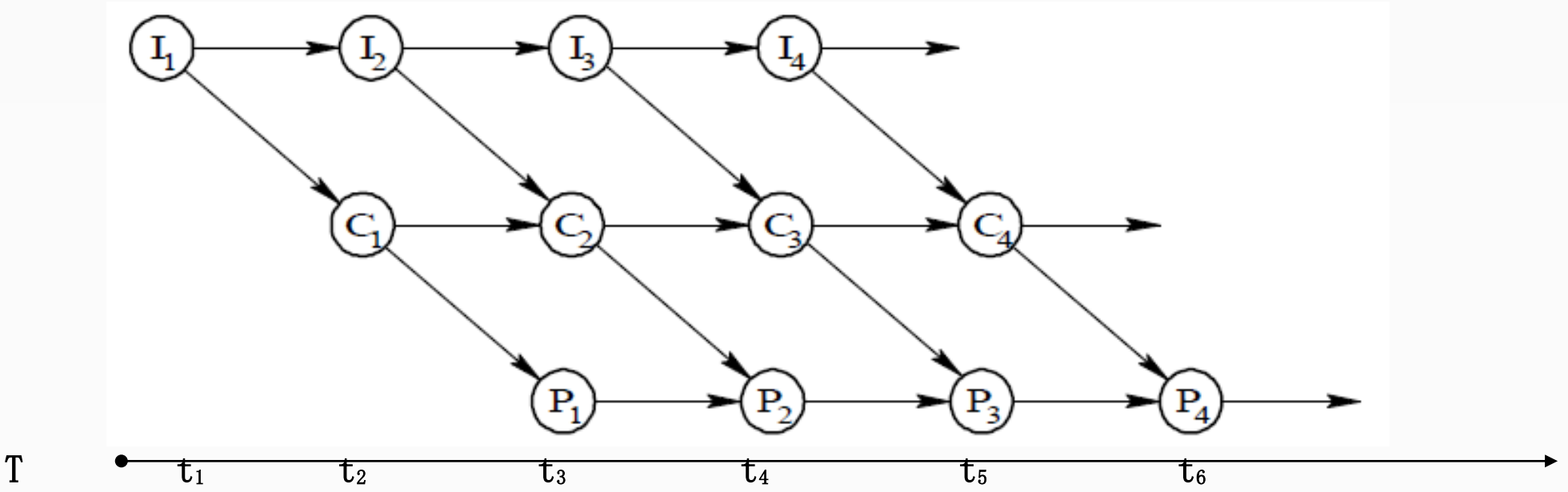
三、程序的并发执行及其特征

1. 程序的并发执行

例：

在系统中有n个作业，每个作业都有三个处理步骤：

输入数据、处理、输出结果，即I_i，C_i，P_i (i=1,2,3,...,n)



I_1、C_1、P_1的执行必须严格按照I_1，C_1，P_1的顺序；
而 C_1与 I_2，P_1与 C_2、I_3 是可以同时执行的。

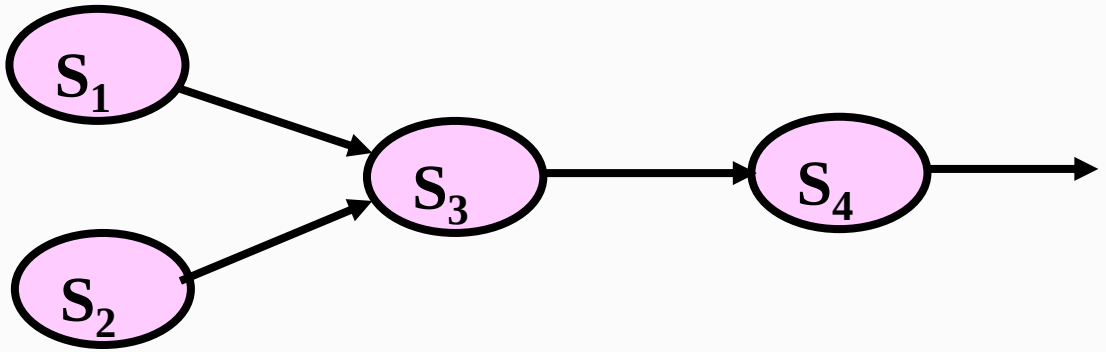
又如：四个程序段

S_1：　a:=x+2

S_2：　b:=y+4

S_3：　c:=a+b

S_4：　d:=c+b



2. 程序并发执行时的特征

（1） 间断性。

（2） 失去封闭性。

（3） 不可再现性。

(1) 间断性

在多道程序设计的环境下，程序的并发执行，以及为完成一项任务而相互合作，这些程序之间要共享系统的资源，形成了相互制约的关系。

相互制约导致并发程序具有“执行—暂停—执行”这种间断性的活动规律。

程序顺序执行时的特征

(1) 顺序性

(2) 封闭性

(3) 可再现性

(1) 间断性

(2) 失去封闭性

程序在并发执行时，系统的资源状态由多道程序来改变，程序运行失去封闭性。程序的运行受到其他程序的影响。

(3) 不可再现性

程序在并发执行时，多次运行初始条件相同的同一程序会得出不同的运行结果。

例：共享公共变量的两个程序，它们执行时可能产生不同结果。

四、进程的特征与状态

在多道程序设计的环境下，为了描述程序在计算机系统内的执行情况，必须引入新的概念--进程。

第二章 进程管理

- 2.1 前趋图和程序执行
- 2.2进程的描述
- 2.3 进程控制
- 2.4 进程同步
- 2.5 经典进程的同步问题
- 2.6 进程通信
- 2.7 线程

1、进程的定义

进程：程序关于某个数据集合的一次执行过程。

行为的一个规则叫做程序，程序在处理机上执行时所发生的活动称为进程（Dijkstra） 。

进程（有时称为任务）是一个程序与其数据一道通过处理机的执行所发生的活动。（Alan.C. Shaw）

进程是执行中的程序。（Ken Thompson and Dennis Ritchie）

2、进程的特征（与程序比较）

（1）结构特征

进程控制块(PCB） + 程序 + 数据 = 进程实体

（2）动态性--最基本特征

进程：进程实体的一次执行过程，有生命周期。

程序：程序是一组有序指令的集合，是静态的概念。

（3）并发性

（4）独立性

（5）异步性　　进程按各自独立的、不可预知的速度向前推进

3、进程的三种基本状态

(1)就绪状态(Ready)

进程已获得除CPU之外的所有必需的资源，一旦得到CPU控制权，立即可以运行。

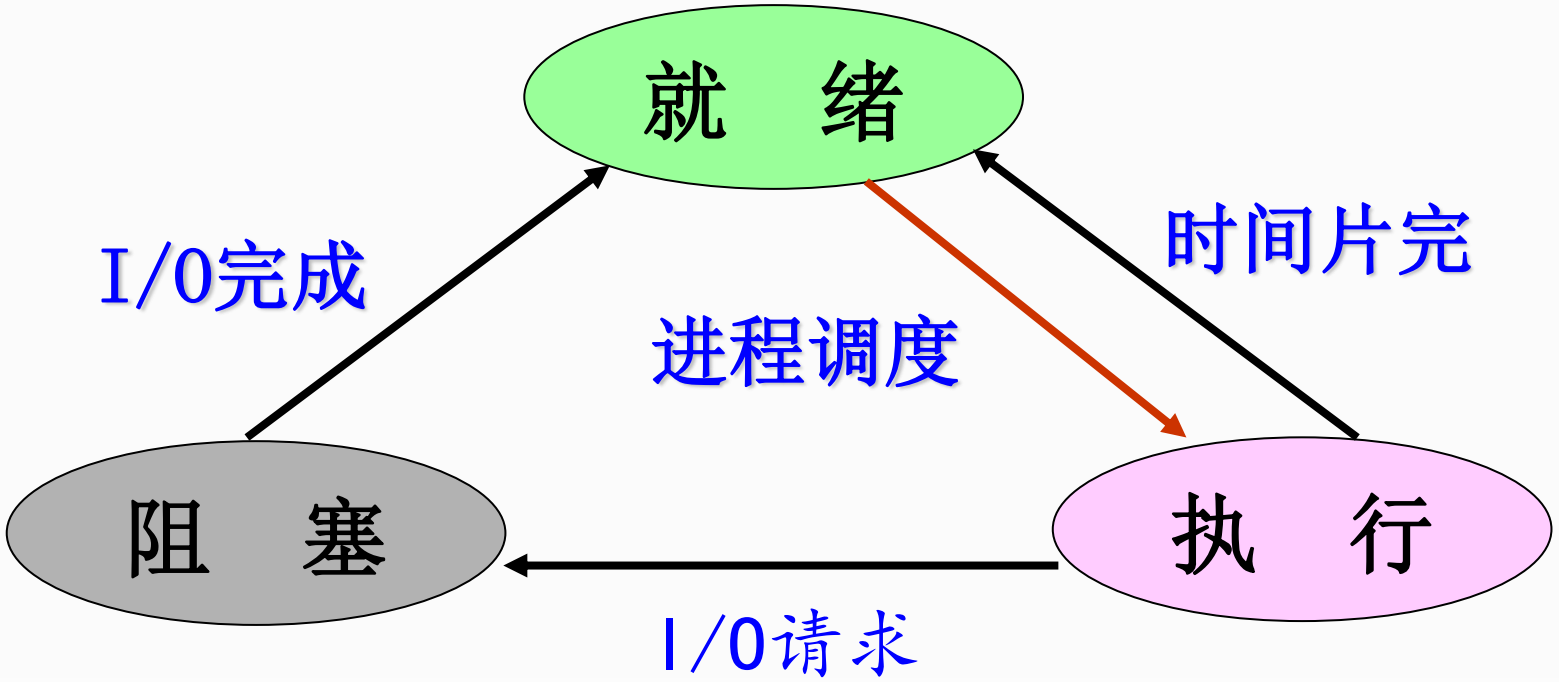
(2)运行状态(Running)

进程已获得运行所必需的资源，它正在处理机上执行。

(3)阻塞状态(Blocked)

正在执行的进程由于发生某事件而暂时无法执行时，便放弃处理机而处于暂停状态，称该进程处于阻塞状态或等待状态。

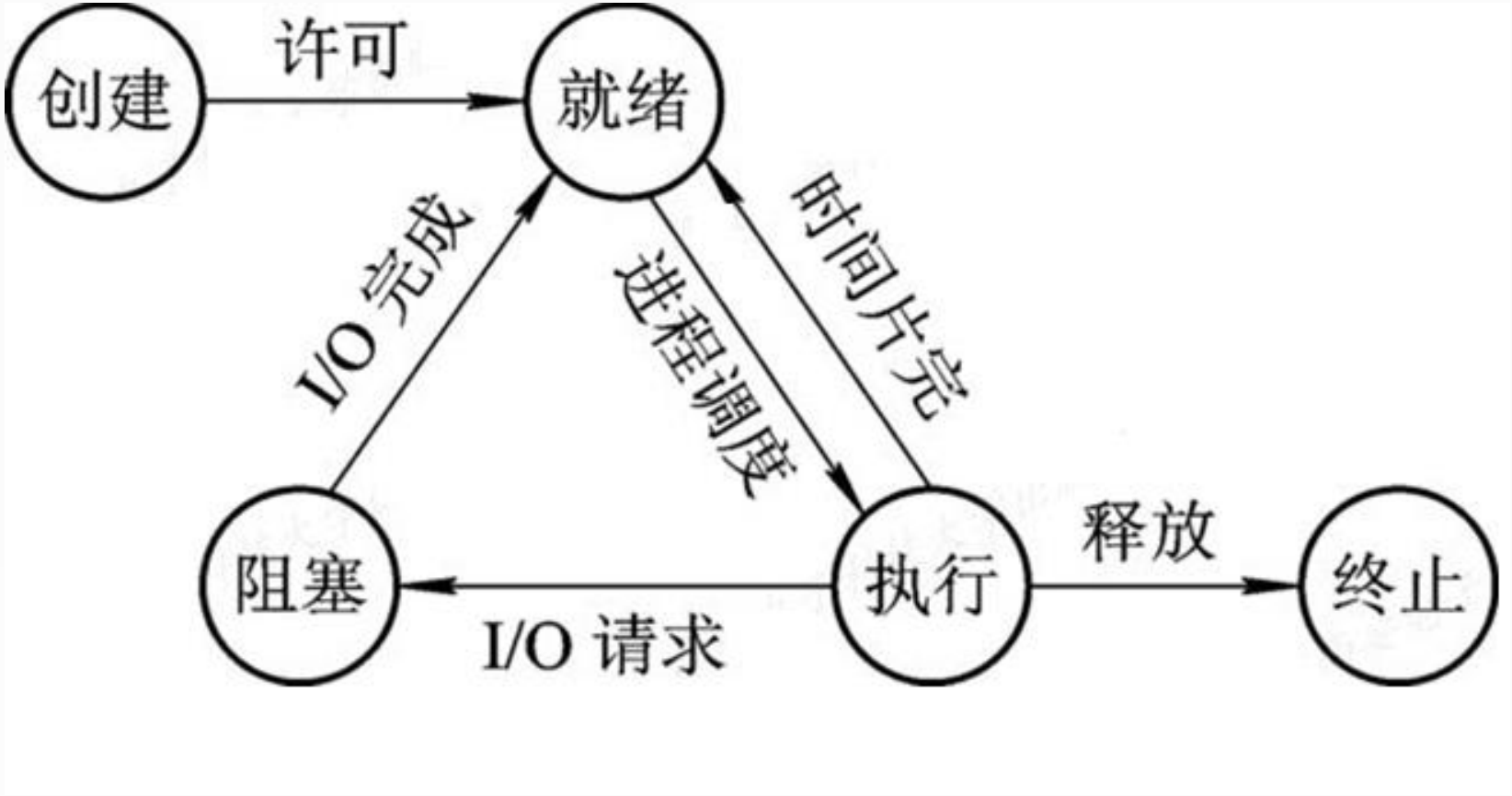
进程的三种基本状态以及各状态之间的转换



就绪队列、阻塞队列

3. 创建状态和终止状态

1）创建状态
2）终止状态



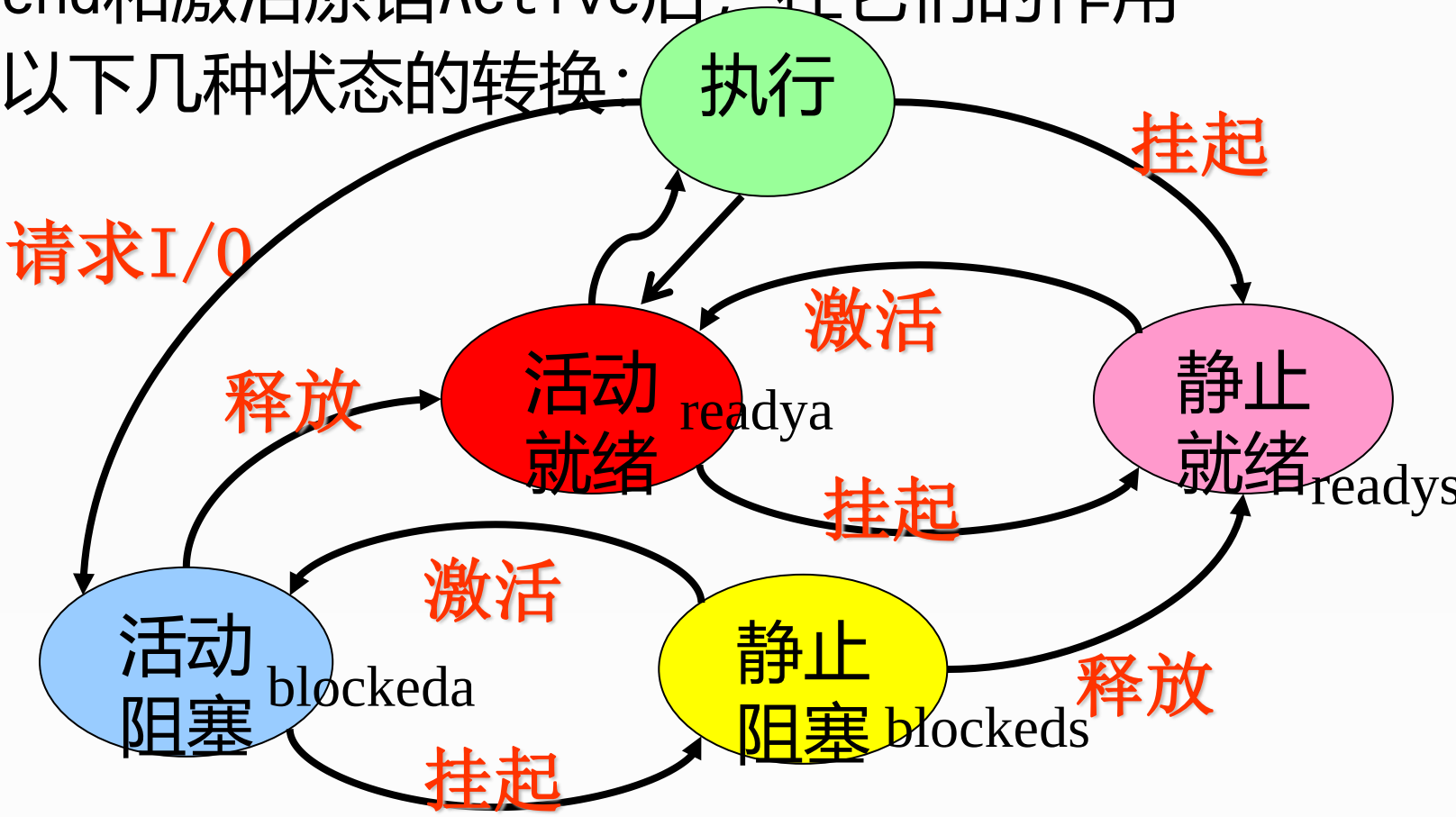
4、挂起状态

1） 引起挂起状态的原因：

- 终端用户的请求
- 父进程请求
- 负荷调节的需要
- 操作系统的需要

2） 进程状态的转换

在引入挂起原语Suspend和激活原语Active后，在它们的作用下，进程将可能发生以下几种状态的转换：

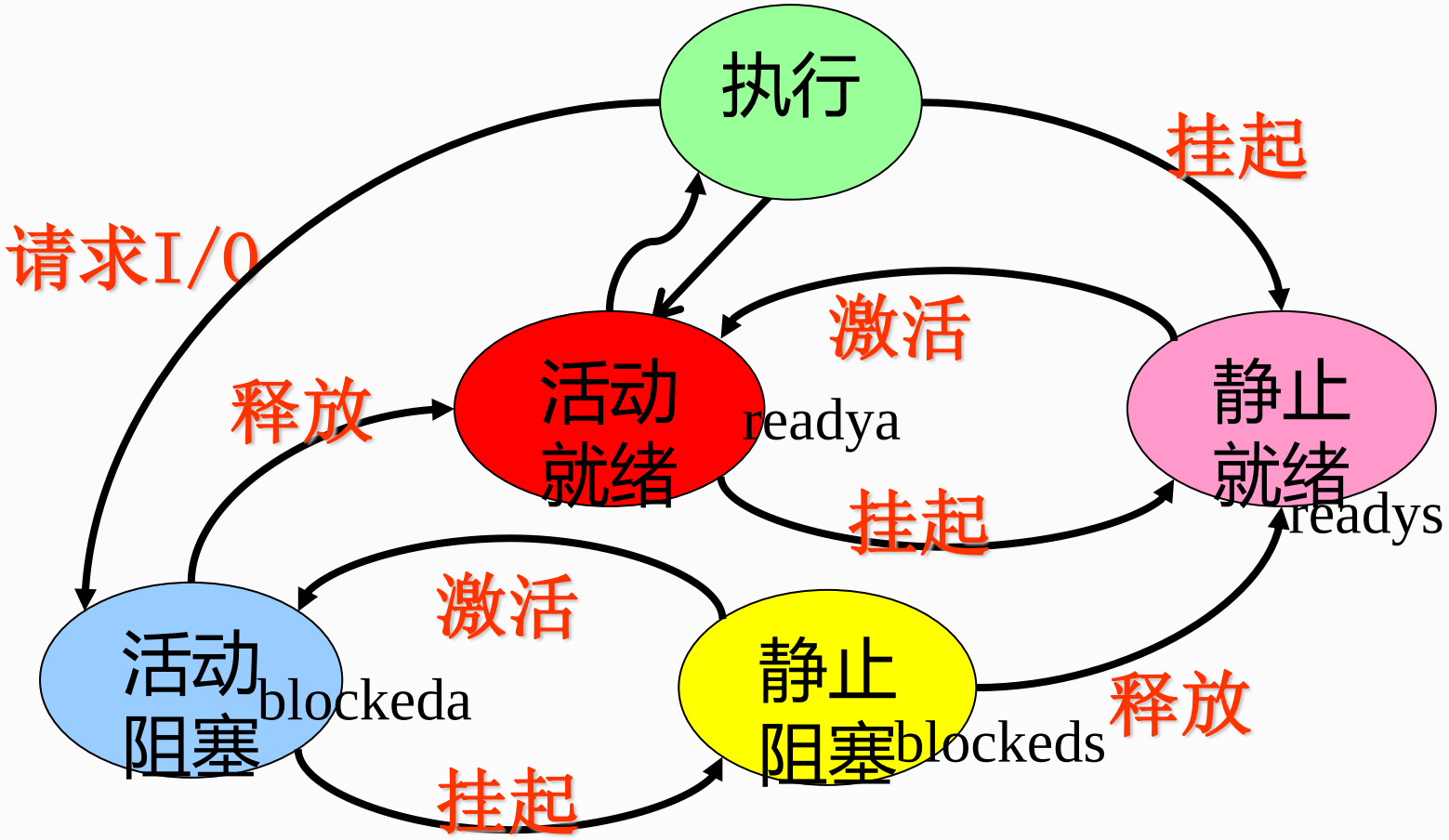


1.活动就绪→静止就绪

2.活动阻塞→静止阻塞

3.静止就绪→活动就绪

4.静止阻塞→活动阻塞



五、进程控制块（PCB）

1. 进程控制块的作用
2. 进程控制块中的信息
3. 进程控制块的组织方式

1. 进程控制块的作用

存放进程管理和控制信息的数据结构称为进程控制块。它是进程管理和控制的最重要的数据结构，在创建时，建立PCB，并伴随进程运行的全过程，直到进程撤消而撤消。

- PCB就像我们的户口。
- PCB是进程存在的唯一标志。

系统的所有PCB组织成链表或队列，常驻内存的PCB区。

2. 进程控制块中的信息

1） 进程标示符

2） 处理机状态

3） 进程调度信息

4） 进程控制信息

1） 进程标示符

每个进程都必须有一个唯一的标识符

- 内部标示符
- 外部标示符

2） 处理机状态

主要由处理机的各种寄存器中的内容组成。处理机运行时的信息存放在寄存器中，当被中断时这些信息要存放在PCB中。

3）进程调度信息

- 进程状态
- 进程优先级
- 进程调度所需的其他信息
- 事件

4）进程控制信息

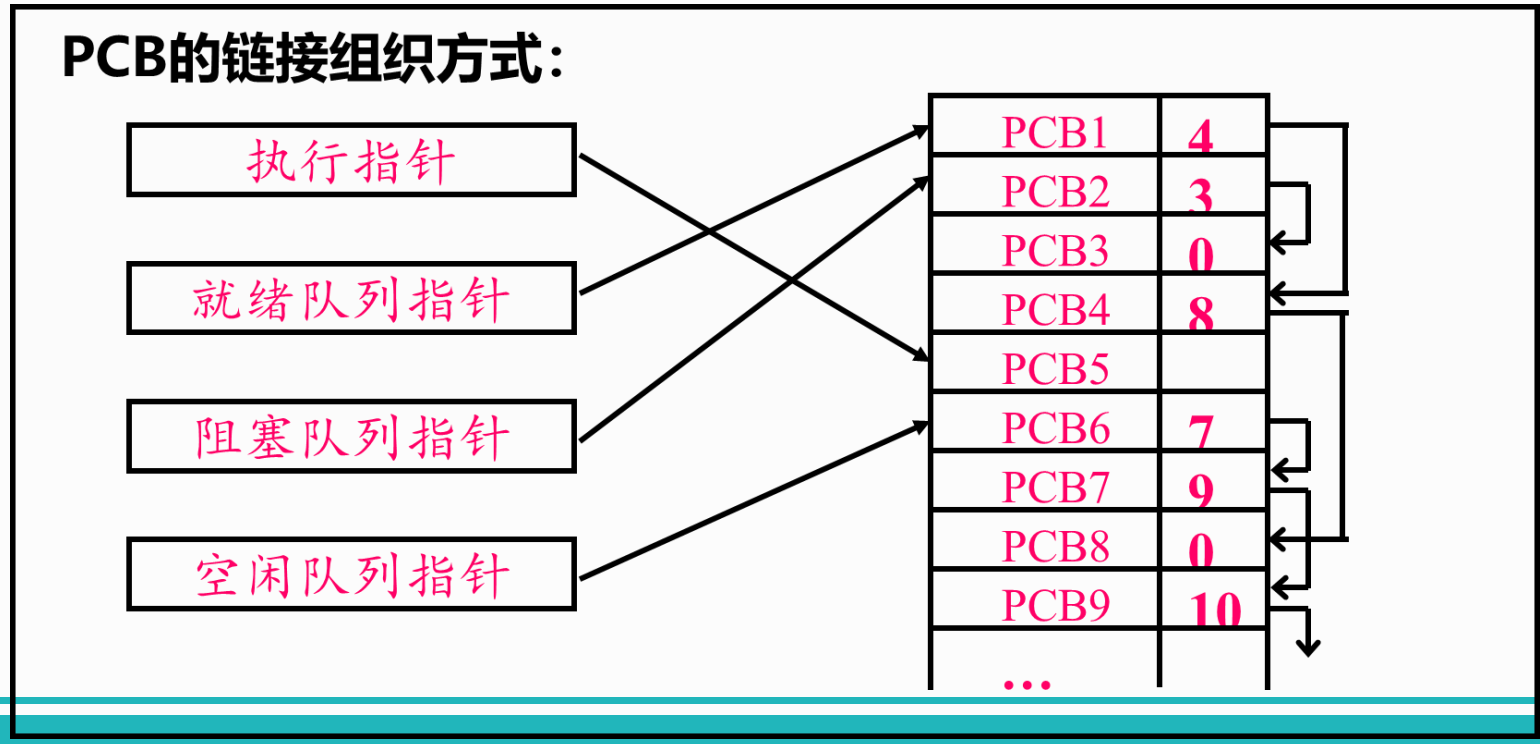
- 程序和数据的地址
- 进程通信和同步机制
- 资源清单
- 链接指针

3. 进程控制块的组织方式

1） 链接方式

把具有同一状态的PCB用其中的链接字链接成一个队列。

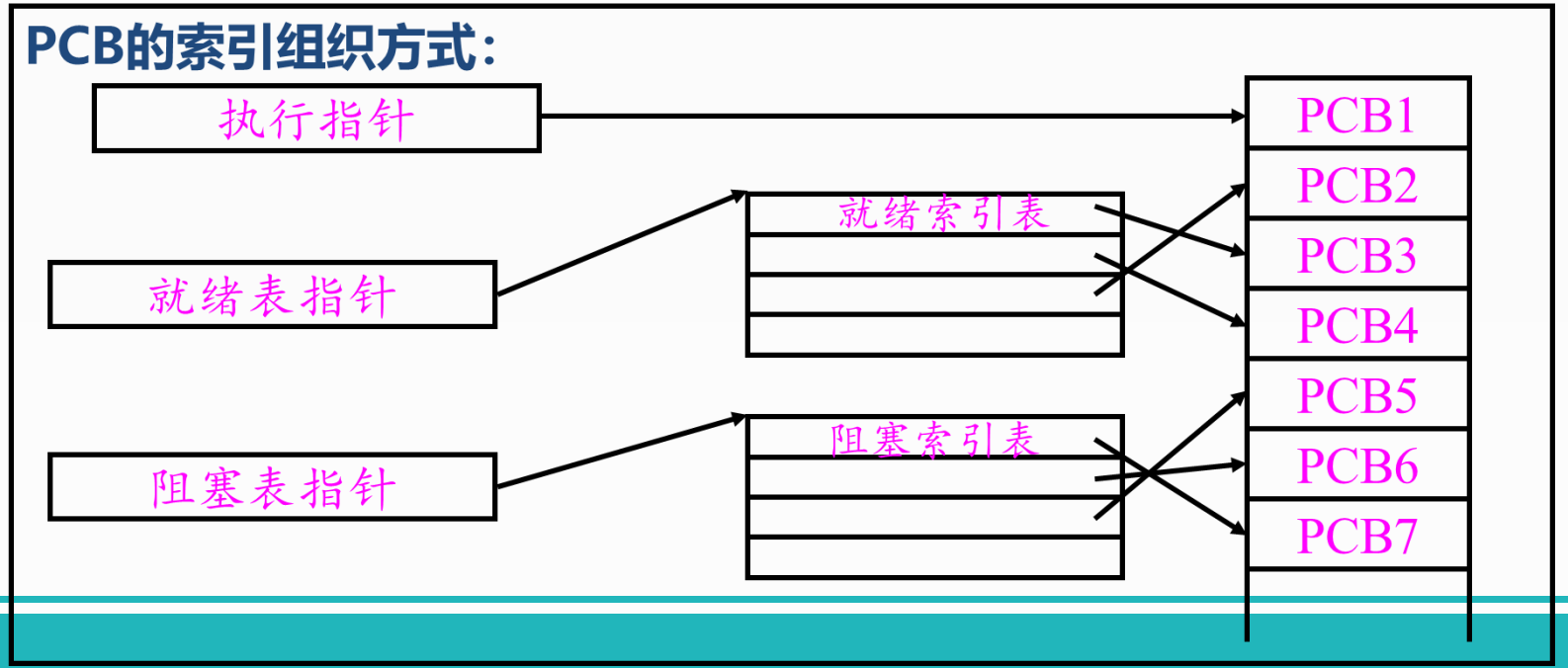
- 就绪队列；
- 若干个阻塞队列；



2） 索引方式

系统根据所有进程的状态建立几张索引表，把各表的内存首地址记录在内存的专用单元中。

索引表的表目中记录了相应状态的某个PCB在PCB表中的地址。



第二章 进程管理

- 2.1 前趋图和程序执行
- 2.2进程的描述
- 2.3 进程控制
- 2.4 进程同步
- 2.5 经典进程的同步问题
- 2.6 进程通信
- 2.7 线程

操作系统内核

1. 支撑功能

(1) 中断处理。

(2) 时钟管理。

(3) 原语操作。

2. 资源管理功能

(1) 进程管理。

(2) 存储器管理。

(3) 设备管理。

进程控制是对系统中的全部进程实施有效的管理，包括进程创建、终止、进程阻塞和唤醒。

一、进程的创建

二、进程的终止

三、进程的阻塞与唤醒

四、进程的挂起与激活

一、进程的创建

1. 进程的层次结构

在OS中，允许一个进程创建另一个进程，通常把创建进程的进程称为父进程，而把被创建的进程称为子进程。

子进程可继续创建更多的孙进程，由此便形成了一个进程的层次结构。如在UNIX中，进程与其子孙进程共同组成一个进程家族(组)。

一、进程的创建

2. 进程图

描述进程的家族关系的有向树

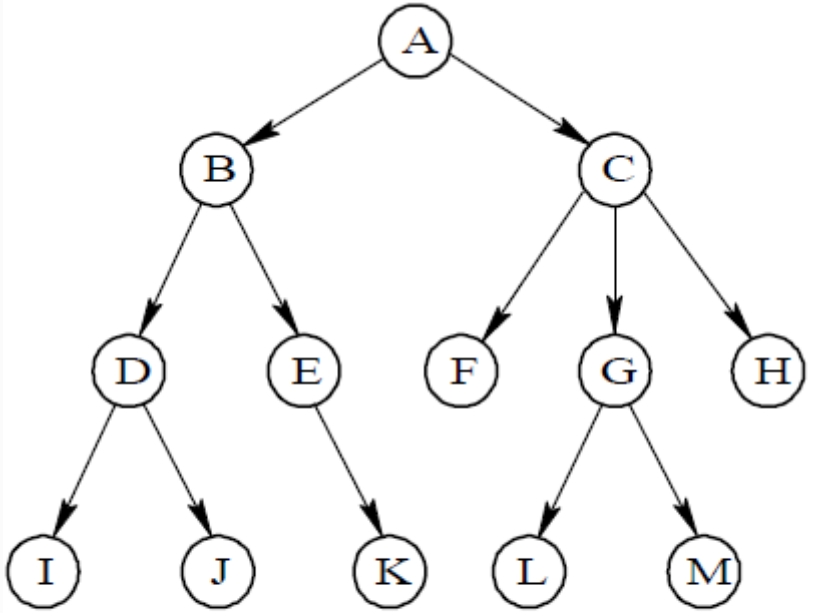


图 2-11　进程树

3. 引起创建进程的事件

为使程序之间能并发运行，应先为它们分别创建进程。导致一个进程去创建另一个进程的典型事件有四类：

(1) 用户登录。

(2) 作业调度。

(3) 提供服务。

(4) 应用请求。

2. 进程的创建

操作系统发现要求创建新进程的事件后，调用进程创建原语Creat()创建新进程。

进程的创建过程：

申请空白PCB → 为新进程分配资源 → 初始化进程控制块 → 将新进程插入就绪队列

2.3 进程控制

一、进程的创建

二、进程的终止

三、进程的阻塞与唤醒

四、进程的挂起与激活

二、进程的终止

1. 引起进程终止的事件

1) 正常结束

2) 异常结束

越界错误、非法指令等

3) 外界干预

操作员或操作系统干预;

父进程请求;

父进程终止

2.进程的终止过程

找出被终止进程的PCB → 若进程状态为运行态，置CPU调度标志为真 → 若其有子孙进程，终止其子孙进程并回收其资源 → 回收终止进程的资源 → 回收终止进程的PCB

2.3 进程控制

一、进程的创建

二、进程的终止

三、进程的阻塞与唤醒

四、进程的挂起与激活

三、进程的阻塞与唤醒

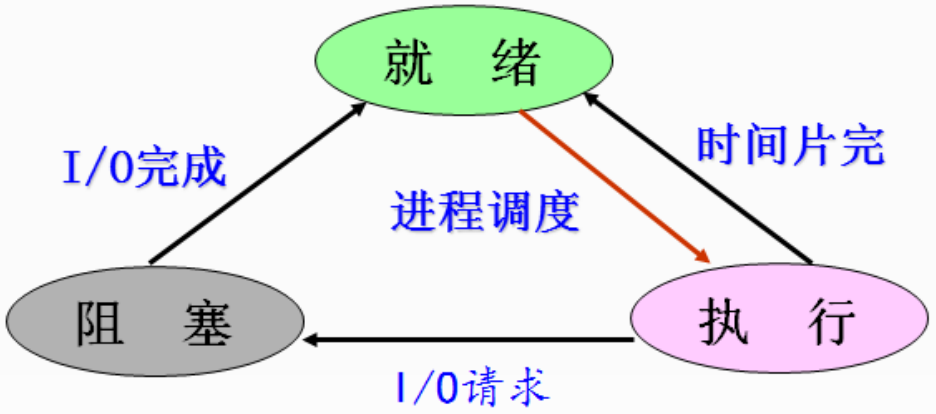
1. 引起进程阻塞和唤醒的事件

1）请求系统服务

2）启动某种操作

3）新数据尚未到达

4）无新工作可做



2. 进程阻塞过程

调用阻塞原语阻塞自己；→ 将PCB中的状态改为阻塞，并加入阻塞队列；→ 转进程调度。

3. 进程唤醒过程

阻塞进程等待的事件发生，有关进程调用唤醒原语 wakeup()
唤醒等待该事件的进程

把阻塞进程从等待该事件的阻塞队列中移出；

置进程状态为就绪态，将PCB插入到就绪队列中。

- 阻塞原语与唤醒原语作用相反，成对使用

2.3 进程控制

一、进程的创建

二、进程的终止

三、进程的阻塞与唤醒

四、进程的挂起与激活

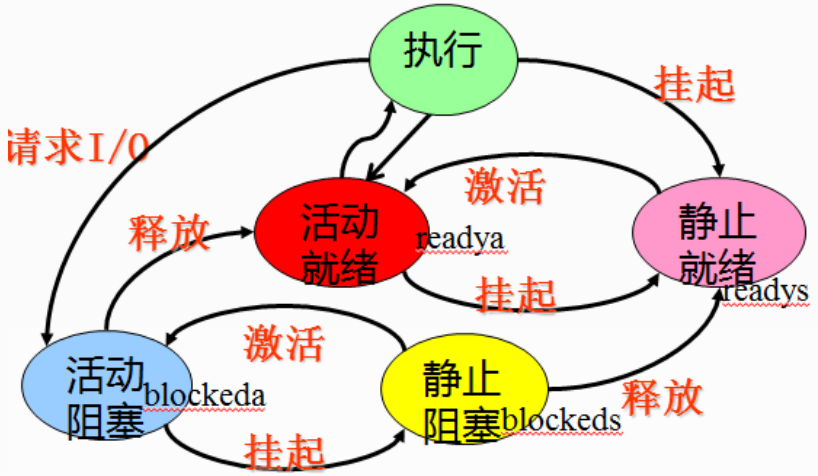
四、进程的挂起与激活

当出现引起进程挂起的事件时，系统利用挂起原语suspend()将指定进程或处于阻塞的进程挂起。

1. 进程的挂起过程

检查被挂起进程的状态：

- 若处于活动就绪，则改为静止就绪；
- 若处于活动阻塞，则改为静止阻塞；
- 若挂起的进程正在执行，则重新进行进程调度。



当发生激活进程的事件时，系统利用激活原语active()将指定进程激活。

2. 进程的激活过程

1） 激活原语先将进程从外存调入内存；

2） 检查该进程的状态：

- 若为静止就绪，则改为活动就绪；
- 若为静止阻塞，则改为活动阻塞。

第二章 进程管理

- 2.1 前趋图和程序执行
- 2.2进程的描述
- 2.3 进程控制
- 2.4 进程同步
- 2.5 经典进程的同步问题
- 2.6 进程通信
- 2.7 线程

目的及要求

- 理解临界资源和临界区的概念；
- 熟练掌握利用信号量机制解决进程同步问题；

2.4 进程同步

一、进程的同步基本概念
二、信号量机制
三、信号量的应用

2.4 进程同步

一、进程的同步基本概念

二、信号量机制

三、信号量的应用

四、管程机制

进程同步的主要任务：

对多个相关进程在执行次序上进行协调，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。

一、进程同步的基本概念

1. 进程间两种形式的制约关系

（1）间接相互制约关系 --- 源于资源共享

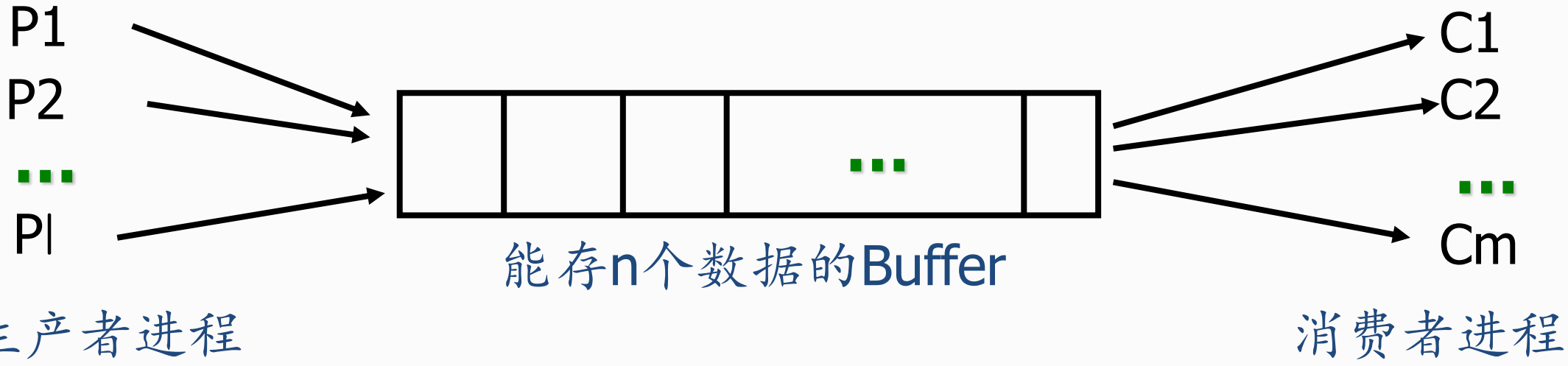
（2）直接相互制约关系 --- 源于进程合作

2. 临界资源

- 临界资源（Critical Resource）：把一段时间内只允许一个进程访问的资源称为临界资源或独占资源
- 临界区（Critical Section）：每个进程中访问临界资源的那段代码称为临界区

2. 临界资源

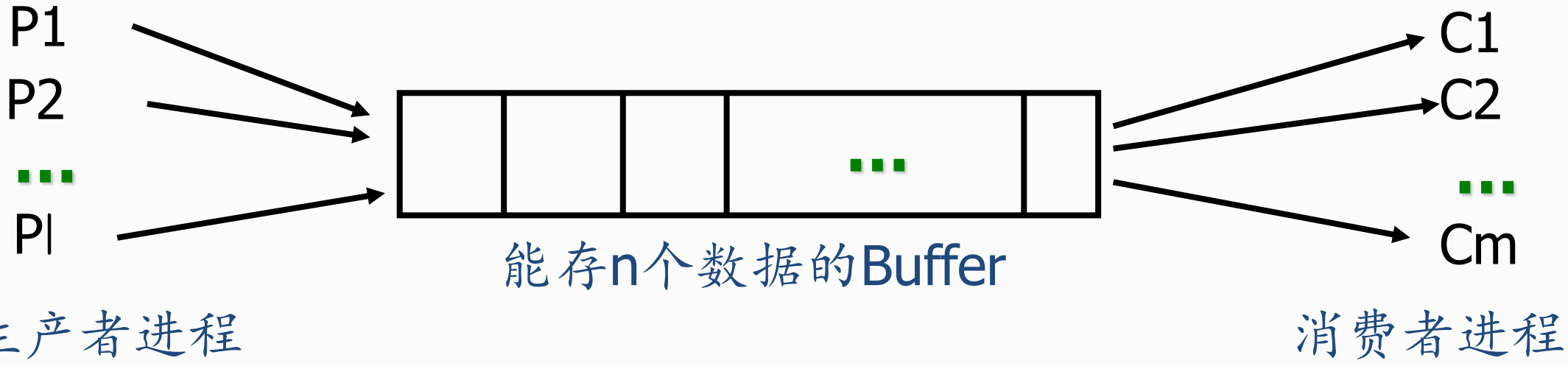
生产者—消费者问题：



生产者进程和消费者进程都以异步方式运行，但它们之间必须保持同步。

2. 临界资源

生产者—消费者问题：



生产者进程和消费者进程都以异步方式运行，但它们之间必须保持同步。

生产者进程和消费者进程共享的变量：

```
Var  n, integer;
type  item=……;
var  buffer: array[0, 1, …, n-1] of  item;
in, out: 0, 1, …, n-1;
counter:0, 1, …, n;
```

生产者进程和消费者进程共享的变量：

```
Var  n, integer;
type  item=……;
var  buffer: array[0, 1, …, n-1] of  item;
in, out: 0, 1, …, n-1;
counter:0, 1, …, n;
```

生产者—消费者问题

```
Producer:
  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:
  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;
```

生产者—消费者问题

```
Producer:
  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:
  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;
```

重复的测试条件

空操作指令

并发操作时会出错

并发执行出错:counter （初值为5）
生产者对它加1，消费者对它减1
机器语言实现：

register1:=counter;
register1:=register1+1;
counter:=register1;
	register2:=counter;
	register2:=register2-1;
	counter:=register2;

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)

counter应当为5
解决方法：把counter作为临界资源，对其互斥访问

```
Repeat
    entry    section
        critical    section
    exit    section
    remainder    section
until    false
```

3. 临界区

临界区：进程中访问临界资源的那段代码

访问临界区的程序设计为：

- 对欲访问的临界资源进行检查，
- 若此刻未被访问，设正在访问的标志 ……进入区
- 访问临界资源 ……临界区
- 将正在访问的标志恢复为未被访问的标志 ……退出区
- 其余部分 ……剩余区

进程互斥：两进程不能同时进入同一临界区

4. 同步机制应遵循的规则

- ✓ 空闲让进
- ✓ 忙则等待
- ✓ 有限等待
- ✓ 让权等待

2.4 进程同步

一、进程的同步基本概念

二、信号量机制

三、信号量的应用

四、管程机制

二、 信号量机制

1965年荷兰Dijkstra提出的信号量（Semaphores）是一种卓有成效的进程同步工具，在长期的应用中，得到了很大的发展，从整型信号量经过记录型信号量，进而发展为“信号量集”机制。

- 信号量就是OS提供的管理公有资源的有效手段。
- 信号量代表可用资源实体的数量。

1. 整型信号量

定义：把整型信号量定义为一个用于表示资源数目的整型量S，除初始化外，仅能通过两个原子操作wait(S),signal(S)来访问

- P操作　wait(S):

  While S<=0 do no-op;

  S:=S-1;

- V操作　signal(S):

  S:=S+1;

P(S)

未遵循“让权等待”原则，导致忙等

P、V操作是原子操作，不可中断。

整型信号量

结构： 一个整型数值。

优点：效率高。

缺点：不可被中断。由于PV操作是原子操作，因此若程序在PV操作间隙出现异常进而中断PV操作，那么会造成该程序无法恢复信号量，进而导致一直锁住某块临界资源，造成程序死锁，永远无法访问。

一个好的进程调度，需要满足这**四个条件**：闲则让进，忙则等待，有限等待，让权等待(见名词表释义)，然而整型信号量无法实现让权等待，导致处理器性能降低。

PV操作的意义：

我们用信号量及PV操作来实现进程的同步和互斥。PV操作属于进程的低级通信。

使用PV操作实现进程互斥时应该注意的是：

（1）每个程序中用户实现互斥的P、V操作必须成对出现，先做P操作，进临界区，后做V操作，出临界区。若有多个分支，要认真检查其成对性。

（2）P、V操作应分别紧靠临界区的头尾部，临界区的代码应尽可能短，不能有死循环。

（3）互斥信号量的初值一般为1。

记录型信号量

- 整型信号量有“忙等”问题，未遵循“让权等待”的准则。
- 记录型信号量是不存在“忙等”现象的进程同步机制。
- 除需要代表资源数目的整型变量value外，还增加一个进程链表指针list，用于链接上述的所有等待进程。

```
Typedef struct{
  int value;          //资源数量
  struct PCB  *list;  //等待队列
}semaphore;
```

记录型信号量续

```
wait(semaphore *S)   //P(S)
{
    S->value--;
    if(S->value<0)
        block(S->list);
}
```

S->value初值表示资源数目

<0时表示等待的进程数目

```
signal(semaphore *S)   //V(S)
{
    S->value++;
    if(S->value<=0)
        wakeup(S->list);
}
```

AND型信号量

- 整型信号量和记录型信号量适用于仅共享一个临界资源，多个临界资源时可能引起死锁

- A、B两个进程访问D、E两个共享数据
  两个信号量Dmutex,Emutex

Process A：

```
.....
wait(Dmutext);
wait(Emutext);
.....
```

Process B：

```
.....
wait(Emutext);
wait(Dmutext);
.....
```

AND型信号量

- AND型信号量基本思想：将进程运行所需资源一次性全部分配，使用完后一次性释放。若一个资源不能满足，则其它也不分配

- AND型信号量一次申请多个临界资源，若有一个不满足则一个也不分配

  Swait(S1,S2…Sn)　Ssignal(S1,S2….Sn)

- 进程一次申请所有需要的资源可避免死锁

信号量集

- 对AND信号量扩充，PV操作中指出每类资源申请数量(d)、分配下限值(t)，S>t才分配
- Swait(S1,t1,d1,S2,t2,d2....Sn,tn,dn)　Ssignal(S1,d1,....Sn,dn)

特殊情况

1）Swait(S,d,d)：只有一个信号量，一次申请d个

2）Swait(S,1,1)：相当于记录型信号量(S>1)或互斥信号量(S=1)

3）Swait(S,1,0)：可控开关，S>=1允许多个进程进入，S =0禁止进入

2.4 进程同步

一、进程的同步基本概念
二、信号量机制
三、信号量的应用
四、管程机制

三、信号量的应用

1. 利用信号量实现进程互斥（模式）
2. 利用信号量实现前驱关系（模式）

1.利用信号量实现进程互斥（模式）

为使多个进程互斥的访问某临界资源，须为该资源设置一互斥信号量mutex，并设其初始值为1，然后将各进程访问资源的临界区CS置于wait(mutex)和signal(mutex)之间即可。

设mutex为互斥信号量，其初值为1，取值范围为（-1,0,1）

当mutex = 1时，表示两个进程皆未进入需要互斥的临界区；

当mutex = 0时，表示有一个进程进入临界区运行，另外一个必须等待，挂入阻塞队列；

当mutex = -1时，表示有一个进程正在临界区运行，另外一个进程因等待而阻塞在信号量队列中，需要被当前已在临界区运行的进程退出时唤醒。

用信号量机制实现互斥的模式

```
Var  mutex: semaphore  :=1;
     begin
     parbegin
        process 1: begin
                        repeat
                           wait(mutex);
                           critical section
                           signal(mutex);
                           remainder section
                        until false;
                     end
     parend
```

wait(mutex)和signal(mutex)必须成对出现

```
process 2: begin
                repeat
                   wait(mutex);
                   critical section
                   signal(mutex);
                   remainder section
                until false;
             end
```

例：用记录型信号量实现两个进程互斥使用一台打印机

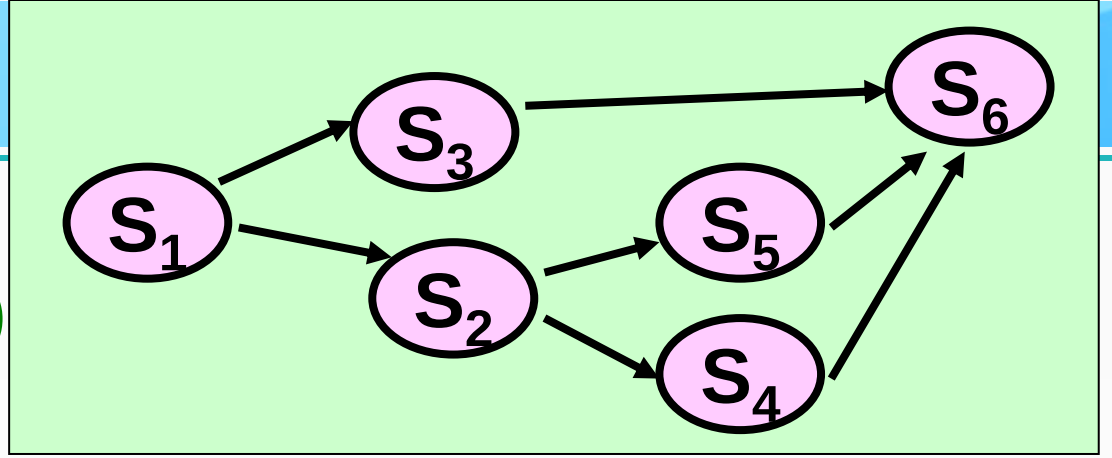
练习：用记录型信号量实现三个进程互斥使用一台打印机

```
Var  mutex: semaphore  :=1;   //表示打印机
     begin
     parbegin
        p1: begin
              repeat
                 … …
                 wait(mutex);
                 使用打印机
                 signal(mutex);
                 … …
              until false;
            end
     parend
```

```
p2: begin
      repeat
        … …
        wait(mutex);
        使用打印机
        signal(mutex);
        … …
      until false;
    end
```

```
p3: begin
      repeat
        … …
        wait(mutex);
        使用打印机
        signal(mutex);
        … …
      until false;
    end
```

2.利用信号量实现前驱关系（模式）




```
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); S5; signal(g); end;
            begin  wait(e); wait(f); wait(g); S6; end;
        parend
    end
```

2.4 进程同步

一、进程的同步基本概念
二、信号量机制
三、信号量的应用
四、管程机制

四、管程机制

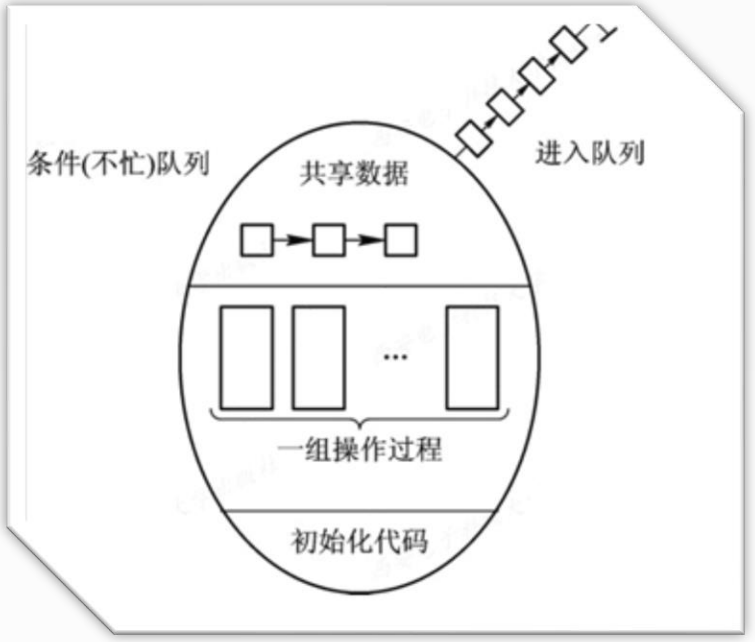
1. 管程的定义

系统中的各种硬件资源和软件资源均可用数据结构抽象地描述其资源特性，即用少量信息和对该资源所执行的操作来表征该资源，而忽略它们的内部结构和实现细节。

代表共享资源的数据结构以及由对该共享数据结构实施操作的一组过程所组成的资源管理程序共同构成一个操作系统的资源管理模块，我们称之为管程。

管程由四部分组成：

① 管程的名称；
② 局部于管程的共享数据结构说明；
③ 对该数据结构进行操作的一组过程；
④ 对局部于管程的共享数据设置初始值的语句。



2. 条件变量

在利用管程实现进程同步时，必须设置同步工具，如两个同步操作原语wait和signal。当某进程通过管程请求获得临界资源而未能满足时，管程便调用wait原语使该进程等待，并将其排在等待队列上。仅当另一进程访问完成并释放该资源之后，管程才又调用signal原语，唤醒等待队列中的队首进程。

管程中包含**条件变量**，用于管理进程的阻塞和唤醒。其形式为 condition x；对它的操作仅有wait和signal。

x.wait： 正在调用管程的进程因 x 条件需要被阻塞或挂起，则调用 x.wait 将自己插入到 x 条件的等待队列上，并释放管程，直到 x 条件变化。此时其它进程可以使用该管程。

x.signal： 正在调用管程的进程发现 x 条件发生了变化，则调用 x.signal，重新启动一个因 x 条件而阻塞或挂起的进程。（与信号量的signal不同，没有s:=s+1的操作）

第二章 进程管理

- 2.1 前趋图和程序执行
- 2.2进程的描述
- 2.3 进程控制
- 2.4 进程同步
- 2.5 经典进程的同步问题
- 2.6 进程通信
- 2.7 线程

2.5 经典进程的同步问题

一、生产者--消费者问题

二、哲学家进餐问题

三、读者--写者问题

2.5 经典进程的同步问题

一、生产者--消费者问题

二、哲学家进餐问题

三、读者--写者问题

生产者消费者问题描述

- 生产者消费者是相互合作进程关系的抽象：输入、计算、输出
- 一群生产者生产产品供消费者去消费
- 共用有n个缓冲区的缓冲池
- 生产者的产品放入池中，消费者从池中取产品消费
- 生产者消费者要同步，池满时停止生产，池空时停止消费
- 必须互斥访问临界资源（公共变量）

记录型信号量解决生产者-消费者

```
int in=0,out=0;
item buffer[n];    //缓冲区
semaphore mutex=1；//互斥信号量
semaphore empty=n,full=0;//资源信号量
void main(){
    cobegin
      producer();consumer();
    coend
}
```

记录型信号量解决生产者-消费者续

```
producer进程
  do{
    produce an item nextp;
     ....
    wait(empty);//不能颠倒
    wait(mutex);//不能颠倒
    buffer[in]=nextp;
    in=(in+1)%n;
    signal(mutex);
    signal(full);
  }while(TRUE);
```

```
consumer进程
  do{
    wait(full);//不能颠倒
    wait(mutex);//不能颠倒
    nextc=buffer[out];
    out=(out+1)%n;
    signal(mutex);
    signal(empty);
    comsume item in nextc;
     ....
  }while(TRUE);
```

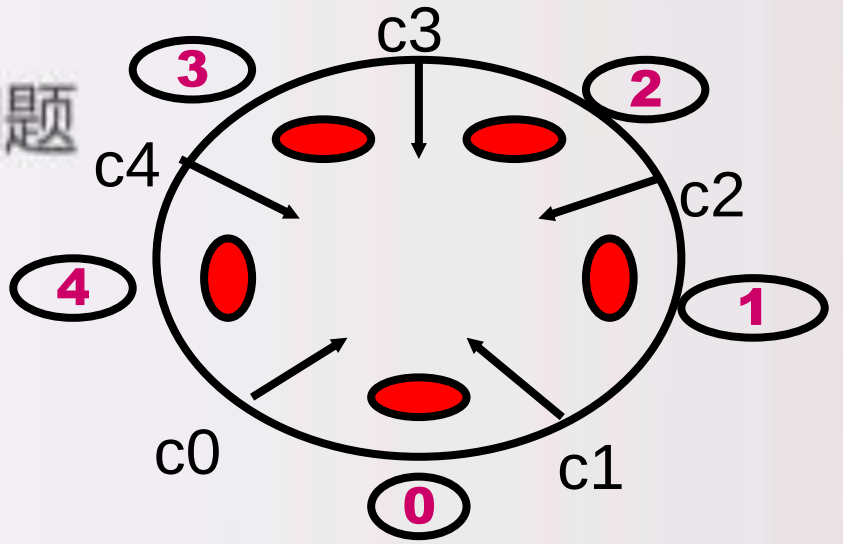
AND信号量解决生产者-消费者

```
producer进程
  do{
    produce an item nextp;
     ....
    Swait(empty,mutex);
    buffer[in]=nextp;
    in=(in+1)%n;
    Ssignal(mutex,full);
}while(TRUE);
```

```
consumer进程
  do{
    Swait(full,mutex);
    nextc=buffer[out];
    out=(out+1)%n;
    Ssignal(mutex,empty);
    comsume item in nextc;
     ....
  }while(TRUE);
```

哲学家就餐问题描述

- Dijksta首先提出并解决该问题，典型同步问题
- 五位哲学家围圆桌就餐，共5只筷子
- 只有拿到左右两只筷子才可就餐



记录型信号量解决哲学家就餐

```
semaphore ch[5]={1,1,1,1,1};
第i个哲学家进程
    do{
    wait(ch[i]);
    wait(ch[(i+1)%5]);
    ...eat...
    signal(ch[i]);
    signal(ch[(i+1)%5]);
    ...think...
    }while(TRUE);
```

- 筷子为临界资源，初值1
- 可能引起死锁
- 解决办法：
  1）至多允许4位同时去拿左筷子
  2）仅当左右筷子都可用才就餐
  3）奇数先拿左筷子，偶数先拿右筷子

AND信号量解决哲学家就餐

semaphore ch[5]={1,1,1,1,1};

第i个哲学家进程

```
do{
Swait(ch[(i+1)%5],ch[i]);  //同时获得两个临界资源才就餐，无死锁
...eat...
Ssignal(ch[(i+1)%5],ch[i]);
...think...
}while(TRUE);
```

读者写者问题描述

- 多个读者可以同时访问临界资源（读取）
- 一次最多一个写者访问临界资源（写入）
- 不允许读者和写者同时访问
- 与文件访问类似
- 实质是要保证写进程与其它进程互斥访问共享资源

用记录型信号量解决读者写者问题续

wmutex用于reader,writer互斥访问临界资源

rmutext用于互斥访问readcount （正在读的进程数目）

```
semaphore rmutex=1,wmutex=1;
int readcount=0;  // readcount表示正在读的进程数目
writer进程：    do{
                    wait(wmutex);
                    执行写操作
                    signal(wmutex);
                    } while(TRUE);
```

用记录型信号量解决读者写者问题

```
reader进程：do{
        wait(rmutex);
        if(readcount==0)wait(wmutex);//第一个读者
        readcount++;
        signal(rmutex);
        执行读操作
        wait(rmutex);
        readcount--;
        if(readcount==0)signal(wmutex); //最后一个读者
        signal(rmutex);
    } while(TRUE);
```

小结

- 精典进程同步问题：生产者-消费者问题、哲学家就餐问题、读者-写者问题
- 生产者-消费者问题是相互合作进程关系的抽象
- 读者-写者问题实现的是一个进程与其它进程互斥访问共享资源，例如文件访问

第二章 进程管理

- 2.1 前趋图和程序执行
- 2.2进程的描述
- 2.3 进程控制
- 2.4 进程同步
- 2.5 经典进程的同步问题
- 2.6 进程通信
- 2.7 线程

进程通信：指进程之间的信息交换。

●低级通信：进程间仅交换一些状态和少量数据。如：进程之间的互斥和同步

信号量机制作为通信工具的缺点：

(1)效率低　(2)通信对用户不透明

●高级通信：进程间可交换大量数据。

用户可直接利用操作系统提供的一组通信命令，高效地传送大量数据的一种通信方式。

操作系统隐藏了进程通信的细节，对用户透明，减少了通信程序编制上的复杂性。

2.6　进程通信（高级通信）

一、进程通信的类型
二、消息传递通信的实现方法
三、直接消息传递系统实例

2.6　进程通信（高级通信）

一、进程通信的类型

二、消息传递通信的实现方法

三、直接消息传递系统实例

一、进程通信的类型

高级通信机制可归结为三大类：

- 共享存储器系统
- 消息传递系统
- 管道通信系统

1、共享存储器系统

相互通信的进程间共享某些数据结构或共享存储区，通过这些空间进行通信。

1）基于共享数据结构的通信方式

进程公用某些数据结构，借以实现诸进程间的信息交换。

实现：公用数据结构的设置及对进程间同步的处理，都是程序员的职责。

操作系统--提供共享存储器

特点：低效。只适合传递相对少量的数据。

2）基于共享存储区的通信方式

在存储器中划出一块共享存储区，诸进程可通过对共享存储区中数据的读或写来实现通信。

2、消息传递系统

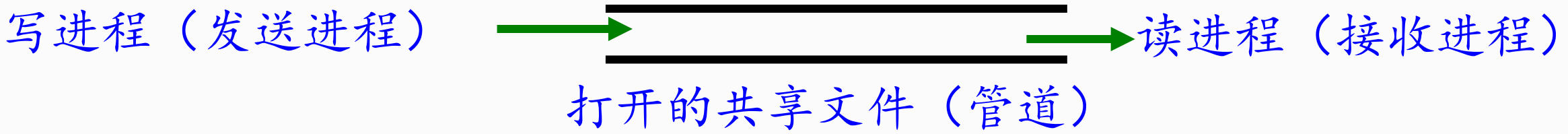
进程间的数据交换，以格式化的消息为单位。

程序员直接利用系统提供的一组通信命令(原语)进行通信。

例：计算机网络：网络报文

3、管道通信

管道：指用于连接一个读进程和一个写进程以实现他们之间通信的一个打开的共享文件，又名pipe文件。



管道机制提供的协调能力：互斥；同步；确定对方是否存在

4. 客户机-服务器系统(Client-Server system)

1) 套接字(Socket)

1.基于文件型

2.基于网络型

4. 客户机-服务器系统(Client-Server system)

2）远程过程调用

3）远程方法调用

2.6　进程通信（高级通信）

一、进程通信的类型

二、消息传递通信的实现方法

三、直接消息传递系统实例

二、消息传递通信的实现方法

进程间通信时，源进程可以直接或间接地将消息传送给目标进程，由此可将进程通信分为直接通信和间接通信。

1. 直接通信方式

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。发送进程和接收进程都以显式方式提供对方的标识符。

(1)对称寻址方式：该方式要求发送进程和接收进程利用OS所提供对方的标识符。

通常系统提供两条通信命令（原语）

通信原语：

Send(Receiver, message);　发送一个消息给接收进程

Receive(Sender, message);　接收Sender发来的消息

（2）非对称寻址方式

该方式的发送和接收原语可表示为：

Send(P,message);

Receive(id,message);

2) 消息的格式

在消息传递系统中所传递的消息，必须具有一定的消息格式。在单机系统环境中，由于发送进程和接收进程处于同一台机器中，有着相同的环境，所以消息的格式比较简单，可采用比较短的定长消息格式，以减少对消息的处理和存储开销。

3) 进程的同步方式

在完成消息的发送或接收后，存在三种情况：

1）发送进程阻塞，接收进程阻塞

2）发送进程不阻塞，接收进程

3）发送进程不阻塞，接收进程不阻塞

4) 通信链路

通信双方建立链路，两种方式：

1）由发送进程在通信之前用显式的“建立连接”命令（原语）请求系统为之建立一条通信链路，在链路使用完成后拆除链路 —— **主要用于计算机网络**

2）发送进程无须明确提出建立链路的请求，只须利用系统提供的发送命令（原语），系统会自动地为之建立一条链路 —— 主要用于单机系统中根据通信方式的不同，把链路分成两种：1）**单向通信链路**2）**双向通信链路**

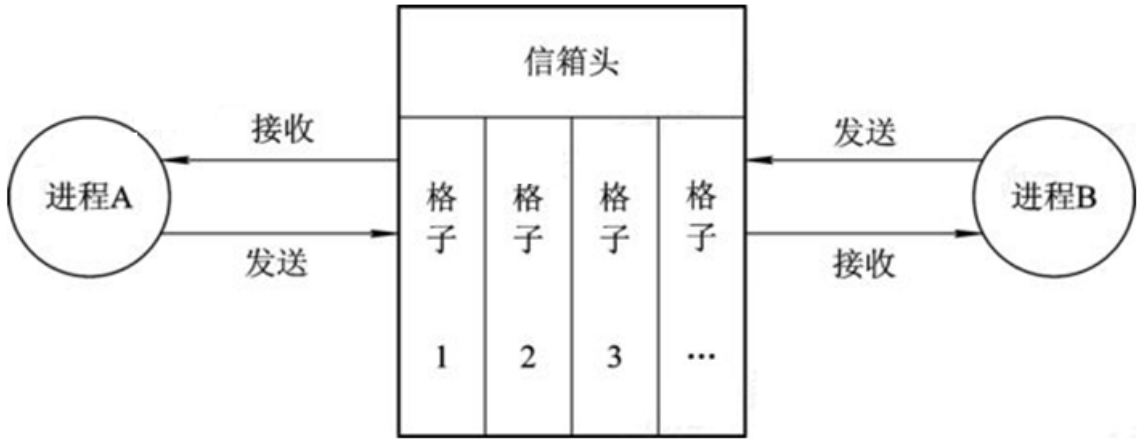
2. 信箱通信

1) 信箱的结构

信箱定义为一种数据结构。在逻辑上，可以将其分为两个部分：

(1) 信箱头

(2) 信箱体



2) 信箱通信原语

系统为邮箱通信提供了若干条原语，分别用于：

(1) 邮箱的创建和撤消。

(2) 消息的发送和接收。

Send（mailbox，message）；

Receive（mailbox，message）

3) 信箱的类型

邮箱可由操作系统创建，也可由用户进程创建，创建者是邮箱的拥有者。据此，可把邮箱分为以下三类：

(1) 私用邮箱。

(2) 公用邮箱。

(3) 共享邮箱。

发送进程和接收进程存在以下四种关系：

1）一对一

2）多对一

3）一对多

4）多对多

2.6　进程通信（高级通信）

一、进程通信的类型

二、消息传递通信的实现方法

三、直接消息传递系统实例

直接消息传递系统实例

消息缓冲队列通信机制首先由美国的Hansan提出，并在RC 4000系统上实现，后来被广泛应用于本地进程之间的通信中。在这种通信机制中，发送进程利用Send原语将消息直接发送给接收进程；接收进程则利用Receive原语接收消息。

1. 消息缓冲队列通信机制中的数据结构

(1) 消息缓冲区。

```
typedef struct message_buffer{
 int sender;    //发送者进程标识符
 int size;    //消息长度
 char *text; /消息正文
  struct message_buffer *next;   //指向下一个消息缓冲区的指针
}
```

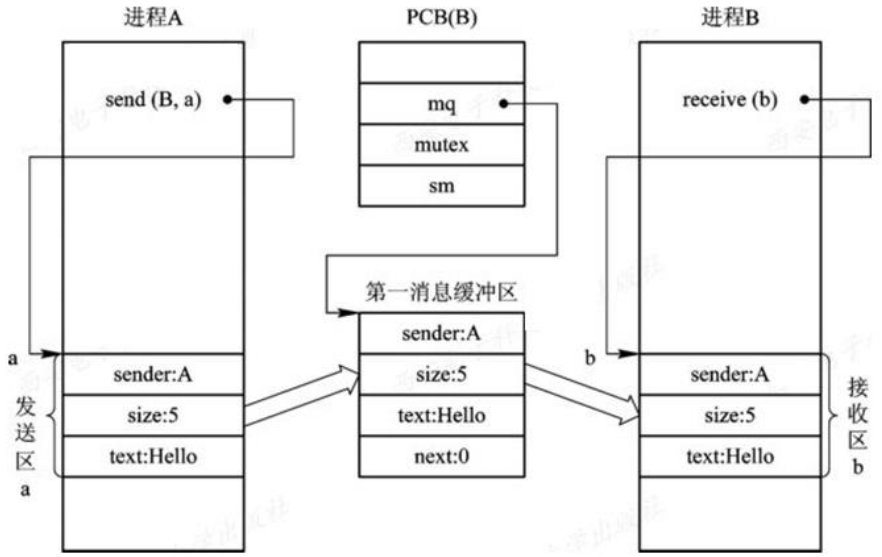
(2) PCB中有关通信的数据项。

```
typedef struct process control_block
{
        ...
        struct message_buffer *mq;   //消息队列队首指针
        semaphore mutex;             //消息队列互斥信号量
        semaphore sm;                //消息队列资源信号量
        ...
}PCB
```

2. 发送原语

发送进程在利用发送原语发送消息之前，应先在自己的内存空间设置一发送区a，把带发送的消息正文，发送进程标识符，消息长度等填入其中

```
void send(reciver,a)
{
   getbuf(a.size,i);        //根据a.size申请缓冲区
   i.sender = a.sender;
   i.size = a.size;
   copy(i.text,a.text);     //将发送区a中的消息复制到缓冲区i中
   i.next=0;
   getid(PCBset,reciver.j); //获得接受进程内部的标识符j
   wait(j.mutex);     //j.mq消息队列是临界资源，操作前要wait()
   insert(&j.mq,i);
   signal(j.mutex);
   signal(j.sm);
}
```



3. 接收原语

接收进程调用接收原语receive(b)，从自己的消息缓冲队列mq中摘下第一个消息缓冲区i，并将其中的数据复制到以b为首址的指定消息接收区内。

```
void receive(b)
{
        j = internal name;    //接收进程内部的标识符
        wait(j.sm);
        wait(j.mutex);
        remove(j.mq,i);       //将消息队列中的第一个消息移出
        signal(j.mutex);
        b.sender = i.sender;
        b.size = i.size;
        copy(b.text,i.text);  //将消息缓冲区i中的信息复制到接受区b中
        releasebuf(i);        //释放消息缓冲区
}
```

第二章 进程管理

- 2.1 前趋图和程序执行
- 2.2进程的描述
- 2.3 进程控制
- 2.4 进程同步
- 2.5 经典进程的同步问题
- 2.6 进程通信
- 2.7 线程

2.7　线程(Threads)

一、线程的引入

如果说，在OS中引入进程的目的是为了使多个程序能并发执行，以提高资源利用率和系统吞吐量，那么，在操作系统中再引入线程，则是为了**减少程序在并发执行时所付出的时空开销，使OS具有更好的并发性**。

- 1. 进程的两个基本属性

  ①拥有资源的独立单位

  ②独立调度和分派的基本单位

2. 程序并发执行所需付出的时空开销

1) 创建进程

2) 撤消进程

3) 进程切换

3. 线程——作为调度和分派的基本单位

线程设计的指导思想：

若进程的两个属性分开，即对于作为调度和分派的基本单位，不同时作为拥有资源的单位，以做到“轻装上阵”；而对于拥有资源的基本单位，又不对之进行频繁的切换。

(1)轻型实体。

(2)可并发执行。

(3)共享进程资源。

适宜多处理机系统

二、 线程与进程的比较

1. 调度的基本单位
2. 并发性
3. 拥有资源
4. 独立性
5. 系统开销
6. 支持多处理机系统

三、线程的状态和线程控制块

1. 线程运行的三个状态

与传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。相应地，线程在运行时也具有下述三种基本状态：

(1) 执行状态，表示线程已获得处理机而正在运行；

(2) 就绪状态，指线程已具备了各种执行条件，只须再获得CPU便可立即执行；

(3) 阻塞状态，指线程在执行中因某事件受阻而处于暂停状态，例如，当一个线程执行从键盘读入数据的系统调用时，该线程就被阻塞。

2. 线程控制块TCB

如同每个进程有一个进程控制块一样，系统也为每个线程配置了一个线程控制块TCB，将所有用于控制和管理线程的信息记录在线程控制块中。

①线程标识符;

②一组寄存器;

③线程运行状态;

④优先级;

⑤线程专有存储区;

⑥信号屏蔽;

⑦堆栈指针。

3. 多线程OS中的进程属性

通常在多线程OS中的进程都包含了多个线程，并为它们提供资源。OS支持在一个进程中的多个线程能并发执行，但此时的进程就不再作为一个执行的实体。多线程OS中的进程有以下属性：

(1) 进程是一个可拥有资源的基本单位。

(2) 多个线程可并发执行。

(3) 进程已不是可执行的实体。

四、线 程 的 实 现

1. 线程的实现方式

（1）内核支持线程

内核支持线程的优缺点

优点

(1) 多处理器系统中，同一进程中的线程可以并行；

(2) 内核知道进程中的线程运行情况；

(3) 线程切换开销小；

(4) 内核本身也可以采用多线程技术，以提高系统的执行速度和效率。

缺点

同一进程下的线程切换开销比较大。

2. 用户级线程ULT(User Level Threads)

用户级线程是在用户空间中实现的。对线程的创建、 撤消、同步与通信等功能，都无需内核的支持，即用户级线程是与内核无关的。在一个系统中的用户级线程的数目可以达到数百个至数千个。由于这些线程的任务控制块都是设置在用户空间，而线程所执行的操作也无需内核的帮助，因而内核完全不知道用户级线程的存在。

优点

(1)同一进程下的线程切换开销小，不用内核状态；

(2)调度算法可以是进程专用的；

(3)用户级线程的实现与OS平台无关；

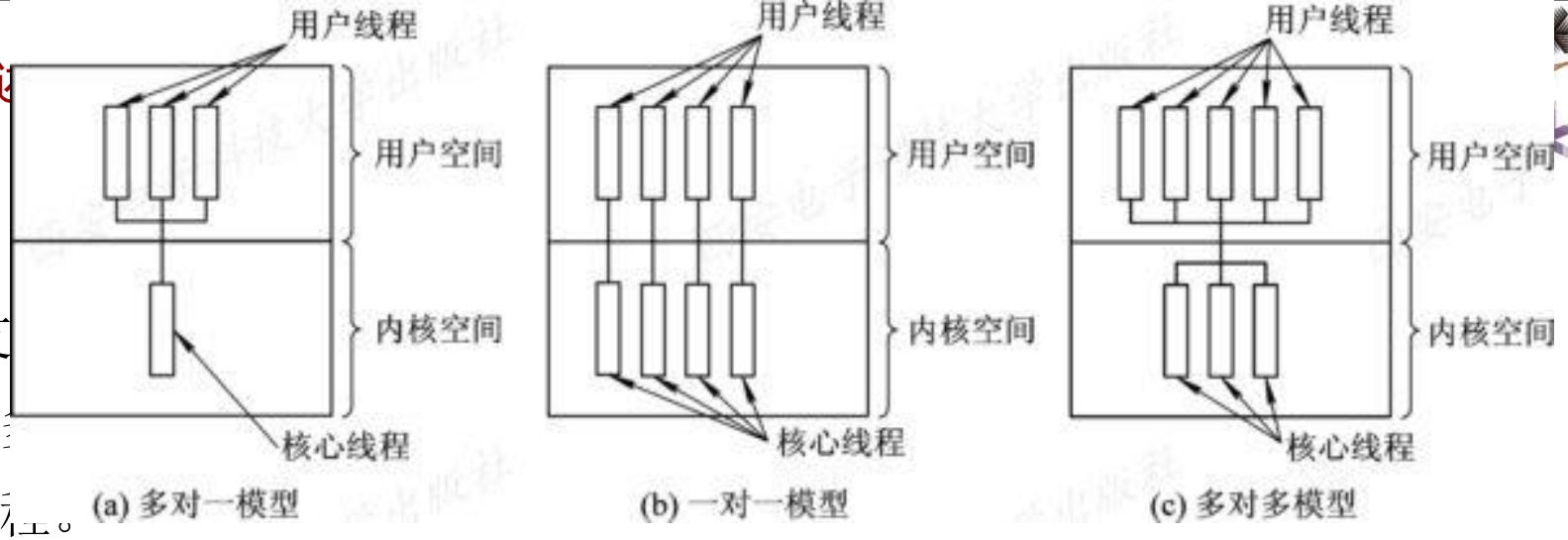
缺点

(1)系统调用导致同一进程下的所有线程阻塞；

(2)不能利用多处理机进行多重处理，进程中仅有一个线程能执行，其他线程只能等待。

3. 组合方式

有些OS把用户级线程和内核支



在组合方式线程系统中，内核支持

用程序建立、调度和管理用户级线程。

用户级线程与内核控制线程的连接（3种连接模型）

1)一对一模型 为每一个用户线程都设置一个内核控制线程与之连接。

优势：开销小，效率高

劣势：一个线程阻塞，整个进程阻塞；映射到一个内核的多线程不能使用多处理机

2)多对一模型 将多个用户线程映射到一个内核控制线程。

优势：更好的并发性，一个线程阻塞，允许调度另一个线程执行；

劣势：开销大；整个系统线程数有限；

3)多对多模型 将多个用户线程映射到多个内核控制线程。

2.8.2　线程的实现

1. 内核支持线程的实现

在仅设置了内核支持线程的OS中，一种可能的线程控制方法是，系统在创建一个新进程时，便为它分配一个任务数据区PTDA(Per Task Data Area)，其中包括若干个线程控制块TCB空间，如图所示。

任务数据区空间

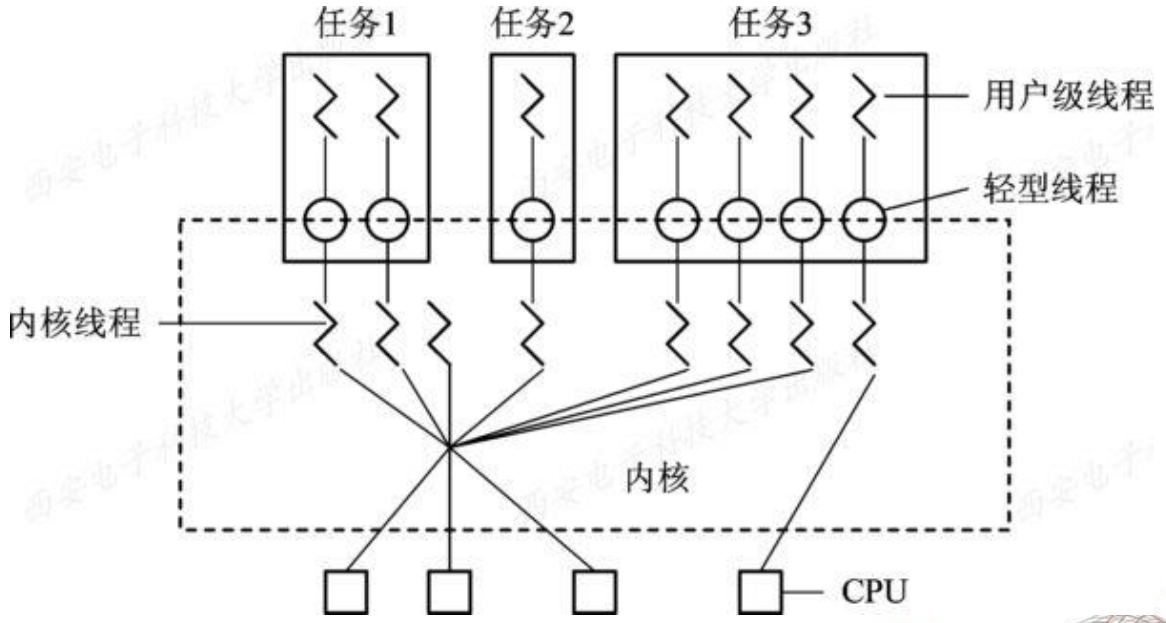
2. 用户级线程的实现

1) 运行时系统(Runtime System)

所谓“运行时系统”，实质上是用于管理和控制线程的函数(过程)的集合，其中包括用于创建和撤消线程的函数、线程同步和通信的函数，以及实现线程调度的函数等。正因为有这些函数，才能使用户级线程与内核无关。运行时系统中的所有函数都驻留在用户空间，并作为用户级线程与内核之间的接口。

2) 内核控制线程

这种线程又称为轻型进程LWP(Light Weight Process)。每一个进程都可拥有多个LWP，同用户级线程一样，每个LWP都有自己的数据结构(如TCB)，其中包括线程标识符、优先级、状态，另外还有栈和局部存储区等。LWP也可以共享进程所拥有的资源。LWP可通过系统调用来获得内核提供的服务，这样，当一个用户级线程运行时，只须将它连接到一个LWP上，此时它便具有了内核支持线程的所有属性。这种线程实现方式就是组合方式。



利用轻型进程作为中间系统

2.8.3　线程的创建和终止

1. 线程的创建

应用程序在启动时，通常仅有一个线程在执行，人们把线程称为“初始化线程”，它的主要功能是用于创建新线程。在创建新线程时，需要利用一个线程创建函数(或系统调用)，并提供相应的参数，如指向线程主程序的入口指针、堆栈的大小，以及用于调度的优先级等。在线程的创建函数执行完后，将返回一个线程标识符供以后使用。

2. 线程的终止

当一个线程完成了自己的任务(工作)后，或是线程在运行中出现异常情况而须被强行终止时，由终止线程通过调用相应的函数(或系统调用)对它执行终止操作。但有些线程(主要是系统线程)，它们一旦被建立起来之后，便一直运行下去而不被终止。在大多数的OS中，线程被中止后并不立即释放它所占有的资源，只有当进程中的其它线程执行了分离函数后，被终止的线程才与资源分离，此时的资源才能被其它线程利用。