# 操作系统

## 第二章 进程管理

# 第二章 进程管理

# 为什么要引入进程

现代操作系统的特征：

并发执行

资源被共享

虚拟性

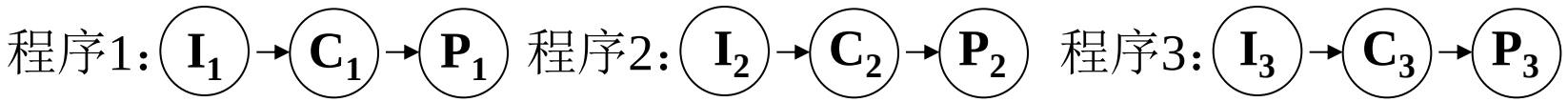
异步性

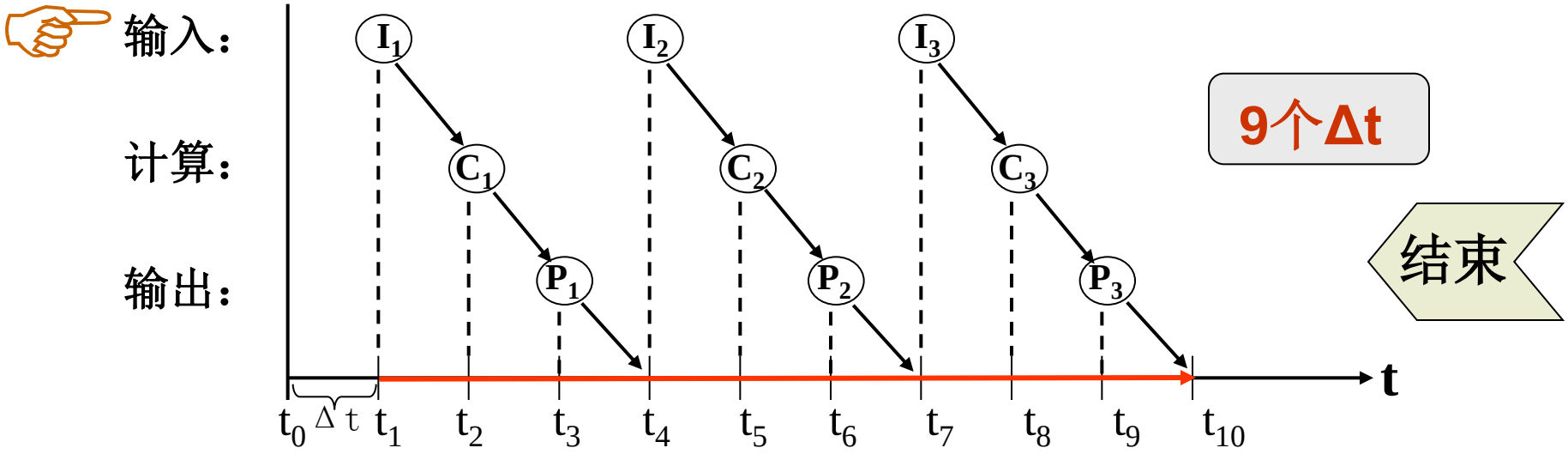
独立的用户如何使用有限的计算机系统资源？

为了使用户充分、有效地利用系统资源。采用一个什么样的概念，来描述程序的执行过程和作为资源分配的基本单位才能充分反映操作系统的并发执行、资源共享及用户随机的特点呢？这个概念就是进程。

# 2.1.1 程序的顺序执行及其特征

程序的顺序执行的有向图（前驱，后继）




我们把程序独占处理机执行，直至最终结束的过程称为程序的顺序执行。

程序的顺序执行具有如下特点：

（1）顺序性：上一条指令执行的结果是下一条指令开始执行的充分必要条件。

（2）封闭性：执行得到的最终结果由给定的初始条件决定，不受外界因素的影响。

（3）可再现性：只要输入的初始条件相同，则重复执行该程序都会得到相同的结果。与执行速度（时间）无关。

问题：执行过程能否再现？

## 2. 多道程序系统中程序执行环境的变化

该环境，能够同时处理多个具有独立功能的程序。

批处理系统、分时系统、实时系统以及网络与分布式系统等都是这样的系统。

多道程序执行环境具有下述三个特点：

（1）程序的并发执行：

多个作业或进程的执行在时间上是重叠的。

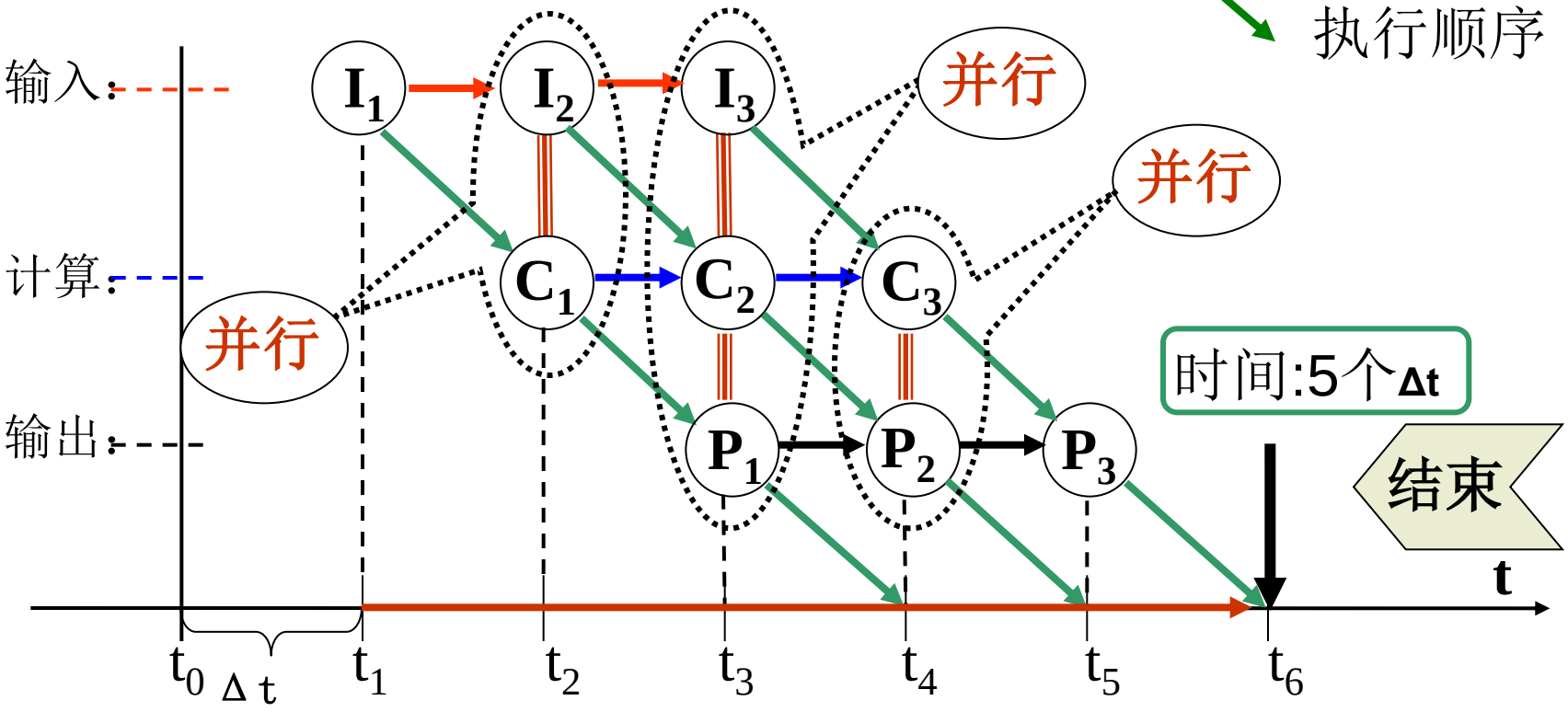
见下图：

没有前驱关系的过程可以**并发执行**（$I_3$，$C_2$，$P_1$），这是**系统的并发性**，提高 (9-5)/9*100% = 44%。

下一步

前驱关系

执行顺序




三个程序并发执行的前驱图

## (2) 随机性

在多道程序环境下，特别是在多用户环境下，程序和数据的输入与执行开始时间都是随机的。

## (3) 资源共享

资源共享将导致对进程执行速度的制约。

有两种方式：

A：由系统统一分配。先向系统提出申请，然后按一定的策略统一分配。如硬资源。

B：由程序自行分配。如数据表，变量等软资源。但系统应提供协调机构，避免因竞争资源而发生混乱。

# 多道程序的执行

吞吐率分别为：
**1/8 = 0.125**
**2/9 = 0.222**
**4/11 = 0.363**
**4**道程序情况比单道提高了近**3**倍。显然不仅使内存充分利用，再还带来处理机利用率的提高，使整个系统效率得以提高。

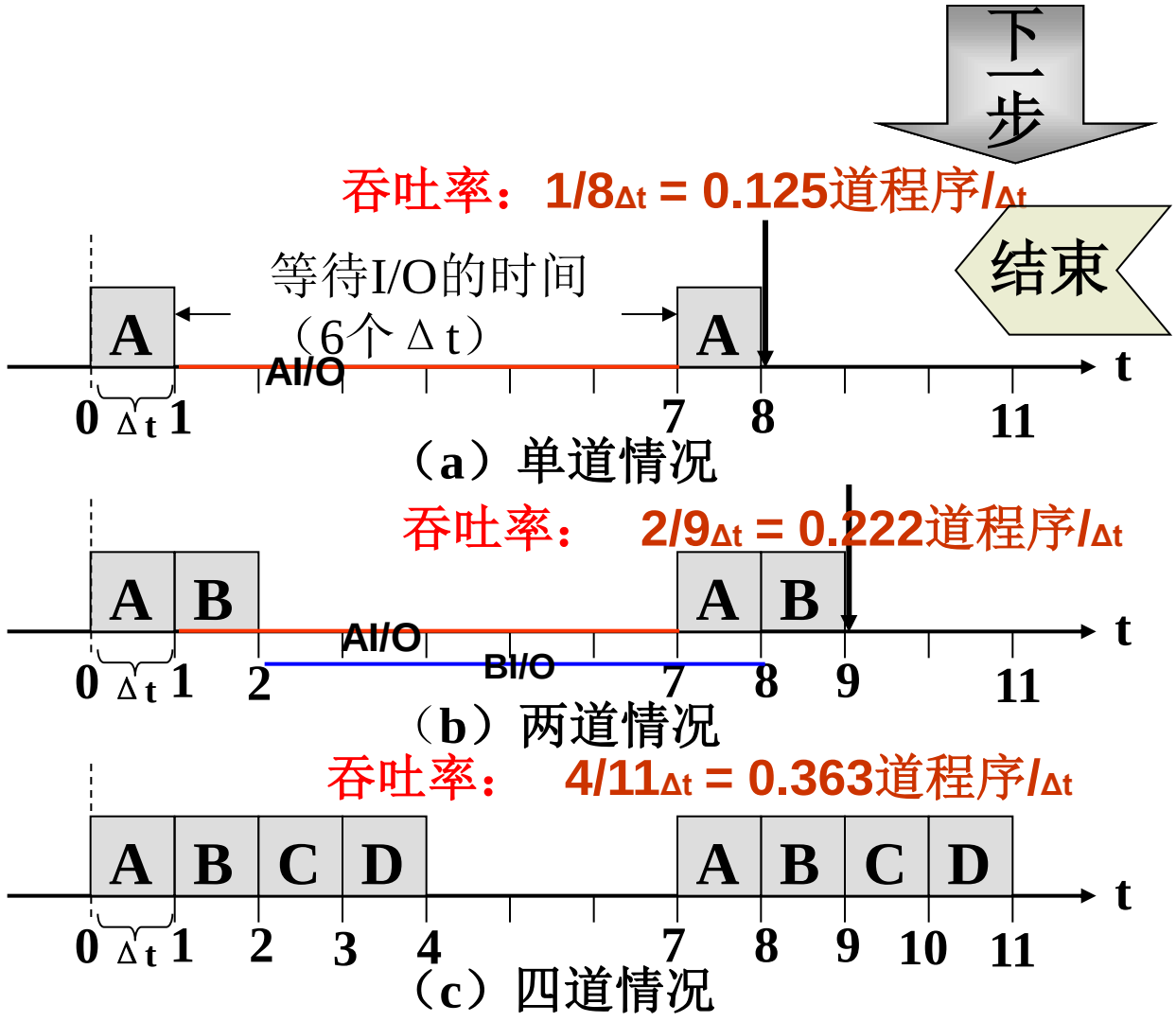



图3.1 单道、两道和四道情况

## 多道程序环境的特点：

### （1）独立性：

每道程序逻辑上是独立的，执行速度（占用**CPU**的时间）与其它程序无关，但程序的周转时间延长了。

### （2）随机性：

多个用户的程序和数据的输入，执行的时间都是随机的。

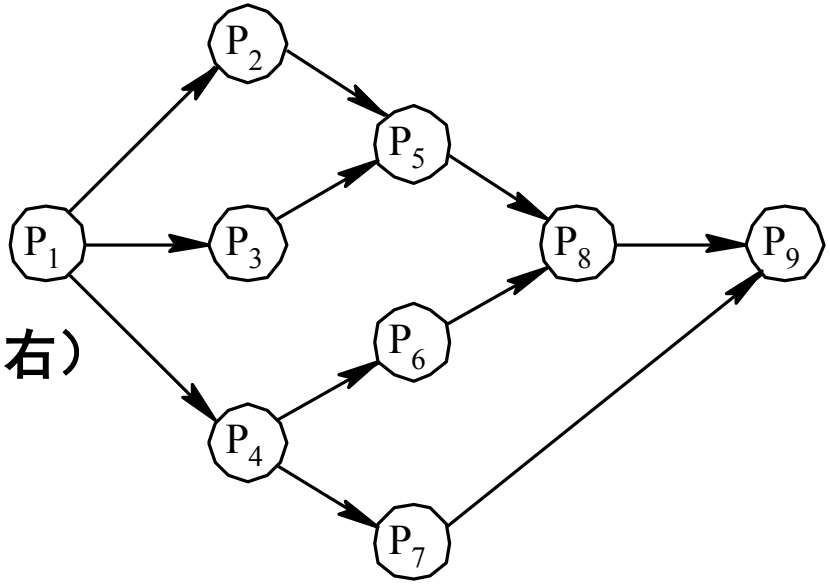
### （3）资源共享：

程序的道数 > CPU和外设的个数，导致了资源必须共享且竞争使用。

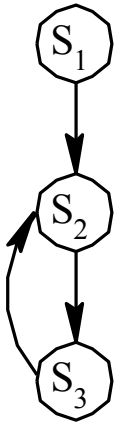
# 2.1.2 前趋图

**前趋图(Precedence Graph)是一个有向无循环图，记为DAG(Directed Acyclic Graph)，用于描述进程之间执行的前后关系。**

**→={($P_i$, $P_j$)|$P_i$ must complete before $P_j$ may start}, 如果($P_i$, $P_j$)∈→,可写成$P_i$→$P_j$，称$P_i$是$P_j$的直接前趋，而称$P_j$是$P_i$的直接后继。**




(*a*) 具有九个结点的前趋图




(*b*) 具有循环的前趋图

**图 2-2 前趋图（右）**

# 2.1.3 程序的并发执行及其特征

对于具有下述四条语句的程序段：

$S_1$: a：=x+2；　$S_2$: b：=y+4；

$S_3$: c：=a+b；　$S_4$: d：=c+b

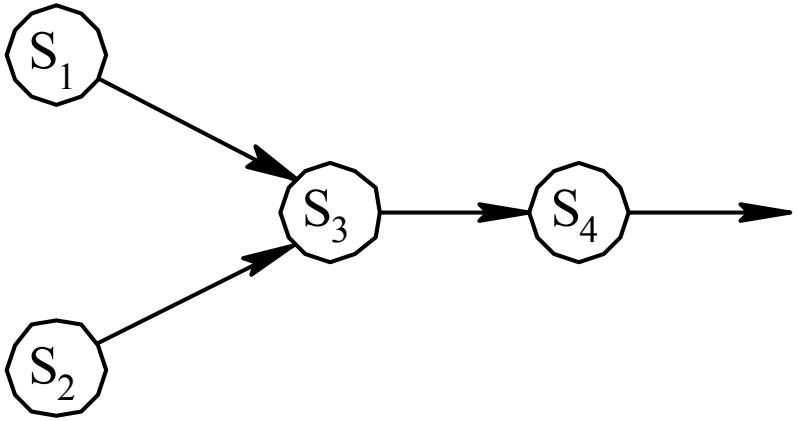



图 2-4 四条语句的前趋关系

# 2. 程序并发执行时的特征

**1)间断性**

**例如：进程1输出ABCD，进程2输出abcd，并发执行会使大小写字母交错在一起。（不是唯一的序列，具有不确定性）**

**2) 失去封闭性**

**并发执行的若干进程运行速度不同，会导致不同结果（如果共享变量的话）**

**3) 不可再现性**

**并发执行的若干进程，由于交错在一起，运行的序列具有不确定性，会导致重复多次运行产生不一样的结果。**

# 2.1.4 进程的特征与状态

1. 进程的基本概念

**进程定义：**

**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”。**

**进程存在的意义：**

**进程是多道程序和分时系统的需要，也是描述程序并发执行的需要。**

# 进程的特征

**1)结构特征**

**进程由程序、数据和进程控制块(PCB)三部分组成。**

**2)动态性**

**有创建，运行，消亡的过程（生命周期，不同的状态）。**

**3)并发性**

**任何进程都可以同其它进程一起并发执行。**

**4)独立性**

**进程是一个独立的运行，分配资源，管理，调度的基本单位。**

**5)异步性**

**由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进。**

# 补充：作业和进程之间的关系

作业：用户需要计算机完成任务时计算机所作工作的集合。

进程：程序的执行过程的描述，是资源分配的基本单位。

区别与关系：

(1)作业是用户向计算机提交任务的实体（单位）。系统将它放入外存中的作业等待队列中等待执行。而进程则是完成任务的实体（单位），是向系统申请分配资源的基本单位。

(2) 一个作业可由多个作业步组成。一个作业步由多个进程组成。一个作业必须至少由一个进程组成。

(3) 作业的概念主要用在批处理系统中。而进程的概念则用在几乎所有的多道系统中。

# 2. 进程的三种基本状态

**1) 就绪状态**

**2) 执行状态**

**3) 阻塞状态**

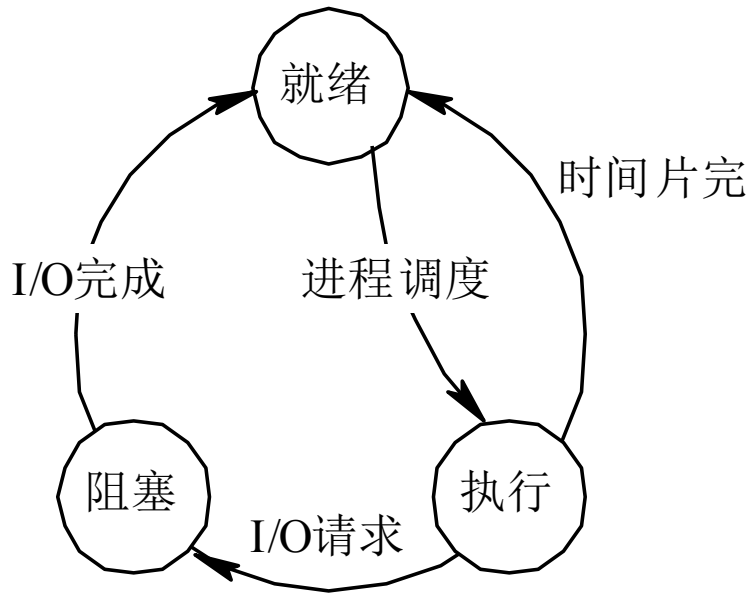



图 2-5 进程的三种基本状态及其转换

# 3. 挂起状态

**1)引入挂起状态的原因**

**(1)终端用户的请求。**

**(2) 父进程请求。**

**(3) 负荷调节的需要。**

**(4) 操作系统的需要。**

# 3. 挂起状态

**2) 进程状态的转换**

**(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。**

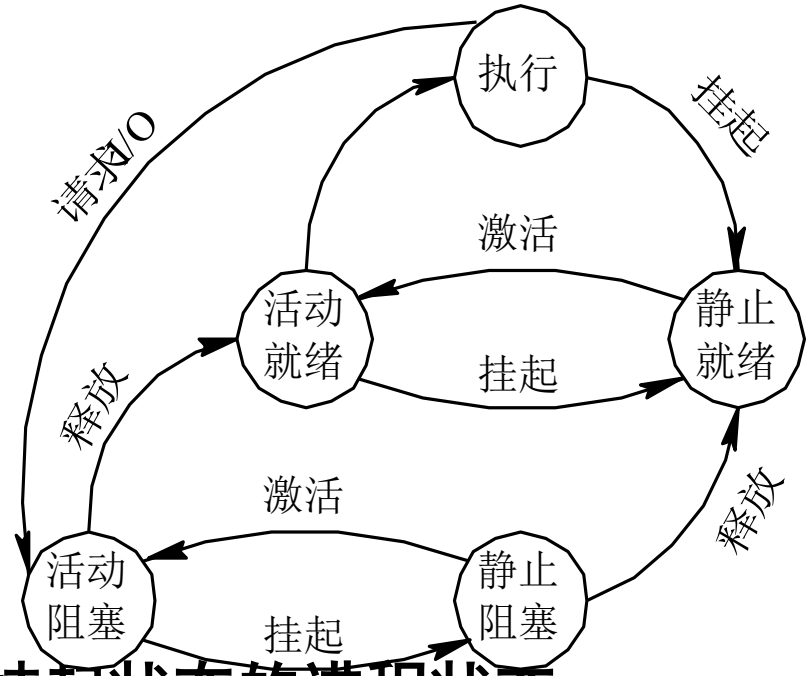



**图 2-6 具有挂起状态的进程状态**

# 进程的结构

进程由三部分组成：

**(1)进程控制块PCB:**

包含了有关进程的描述信息、控制信息、资源信息以及现场信息。系统根据PCB感知进程的存在和通过PCB中所包含的各项变量的变化，掌握进程所处的状态以达到控制进程活动的目的。每一个进程都有一个PCB。

一个进程的PCB结构全部或部分常驻内存的。

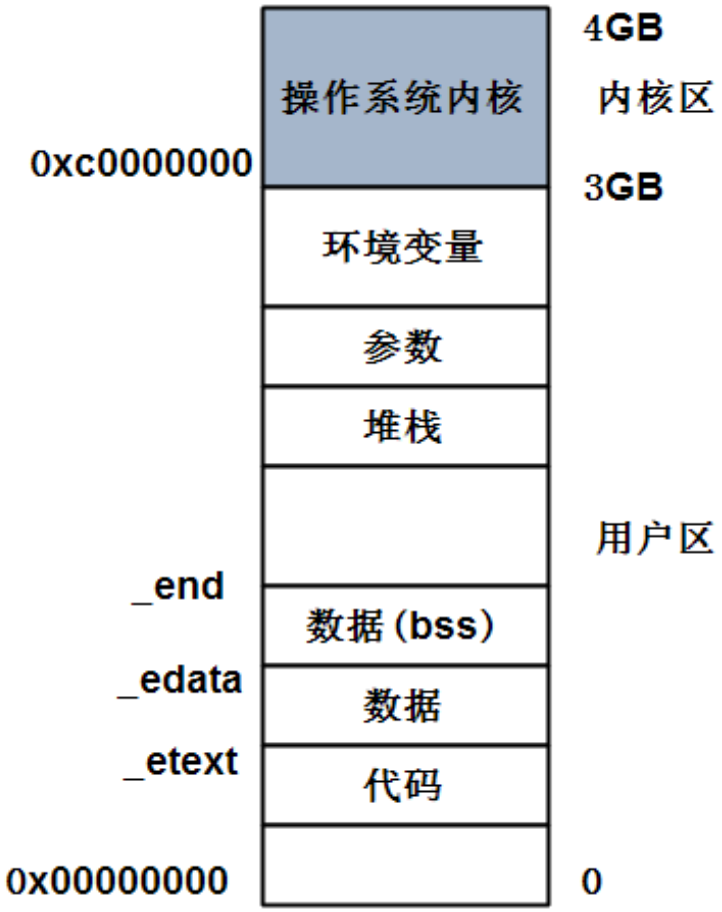
# 进程的结构

**(2)进程的程序:**描述进程所要完成的功能（部分程序代码）。

**(3)数据结构集:**是程序在执行时必不可少的工作区和操作对象。

为了节省内存，常将(2),(3) 两部分内容放在外存中，直到该进程执行时再调入内存。

# 进程的虚拟地址空间结构



# 2.1.5 进程控制块

◆ **进程控制块的作用**

进程控制块的作用是使一个在多道程序环境下不能独立运行的程序(含数据)，成为一个能独立运行的基本单位。

◆ **进程控制块中的信息**

进程标识符

处理机状态

进程调度信息

进程控制信息

# 进程控制块PCB存在的意义

由于资源共享，进程在并发执行时，带来各进程之间的**相互制约**。

为了反映这些制约关系和资源共享关系，在创建一个进程时，应首先创建其 PCB，然后才能根据PCB 中信息对进程实施有效的管理和控制。当一个进程完成其功能之后，系统则最后释放PCB，进程也随之消亡。

**总之， PCB 集中反映一个进程的动态特征。**

# 进程控制块的组织方式

**1) 链接方式**

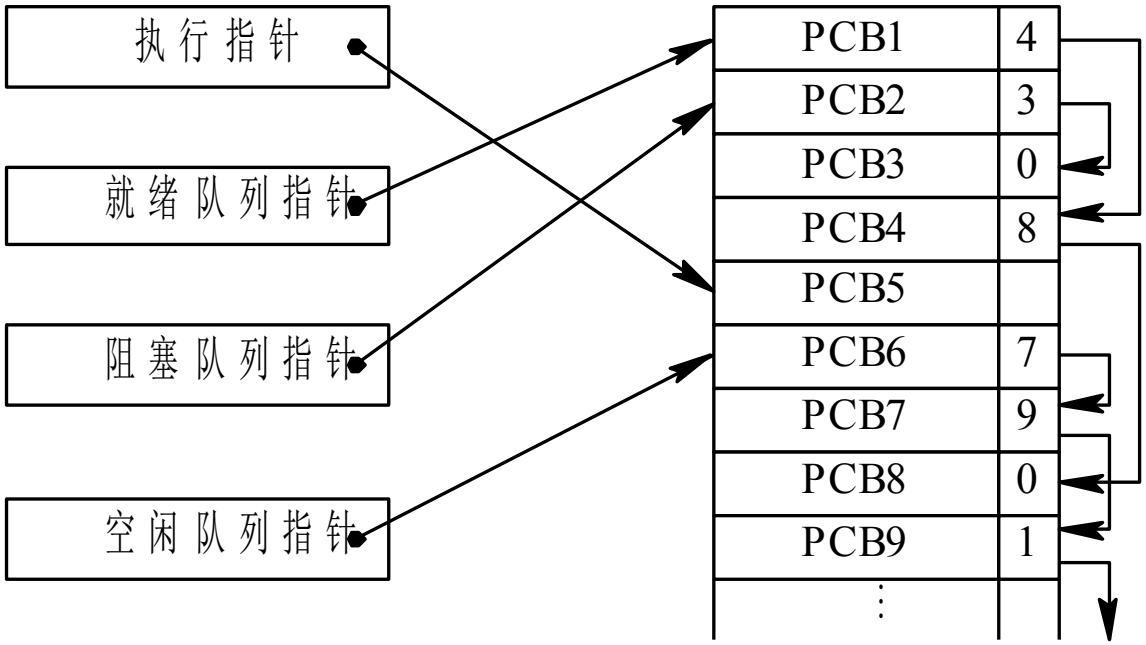



图 2-7 PCB链接队列示意图

## 2) 索引方式

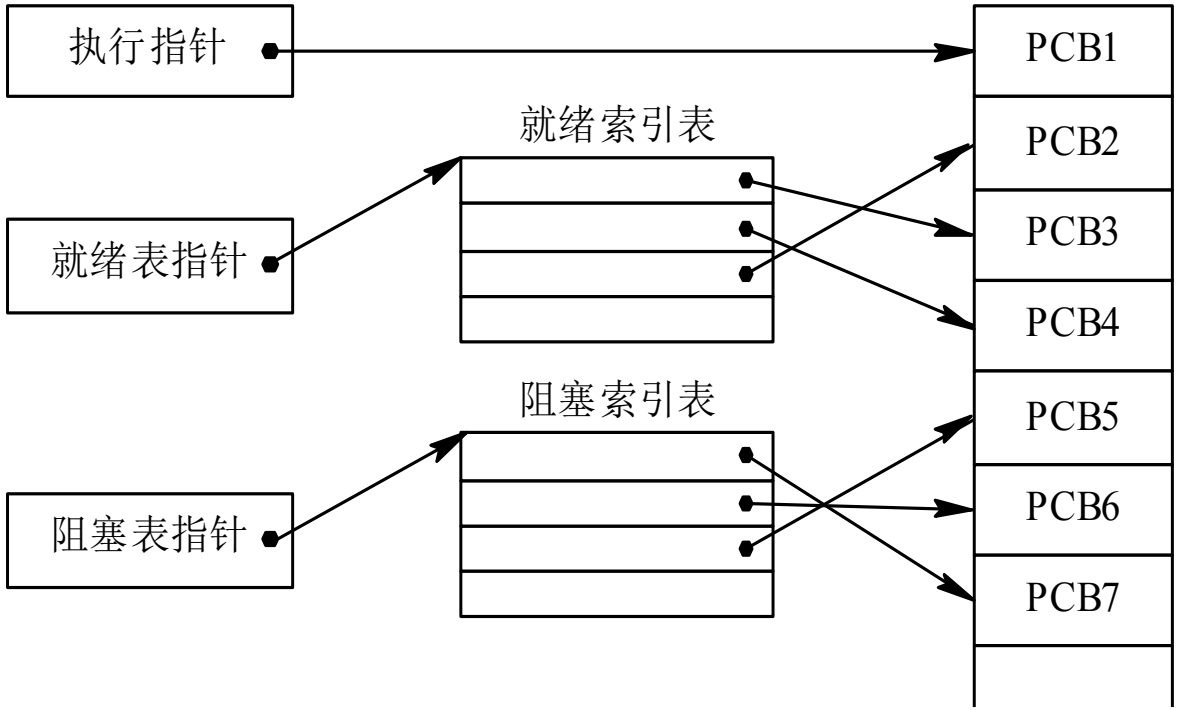



图 2-8 按索引方式组织PCB

# 第二章 进程管理

2.1 进程的基本概念

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 管程机制

2.6 进程通信

2.7 线程

# 2.2 进 程 控 制

## 2.2.1 进程的创建

**1.进程图**

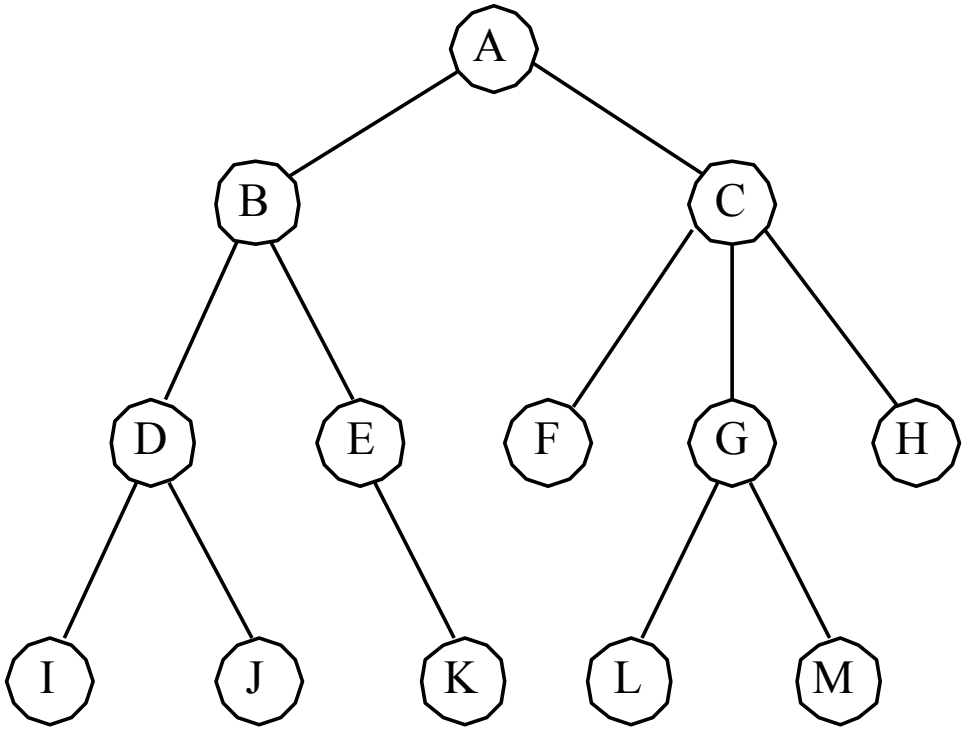



图 2-9 进程树(右)

# 进程的树状结构

```
linux@ubuntu:~$ pstree
init─┬─NetworkManager─┬─dhclient
     │                ├─dnsmasq
     │                └─2*[{NetworkManager}]
     ├─accounts-daemon───{accounts-daemon}
     ├─acpid
     ├─atd
     ├─avahi-daemon───avahi-daemon
     ├─bamfdaemon───2*[{bamfdaemon}]
     ├─bluetoothd
     ├─colord───2*[{colord}]
     ├─console-kit-dae───64*[{console-kit-dae}]
     ├─cron
     ├─cupsd
     ├─2*[dbus-daemon]
     ├─dbus-launch
     ├─dconf-service───2*[{dconf-service}]
     ├─gconfd-2
     ├─geoclue-master
     ├─6*[getty]
     ├─gnome-keyring-d───5*[{gnome-keyring-d}]
     ├─gnome-terminal─┬─bash───su───bash───exp_5_2_6
     │                ├─bash───pstree
     │                └─gnome-pty-helpe
```

# 2. 引起创建进程的事件

**(1) 用户登录。**

**(2) 作业调度。**

**(3) 提供服务。**

**(4) 应用请求。**

# 3. 进程的创建

1) 申请空白PCB。

2) 为新进程分配资源。

3) 初始化进程控制块。

4) 将新进程插入就绪队列，如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。

# 2.2.2 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

3) 外界干预

# 2. 进程的终止过程

1) 根据被终止进程的标识符，从PCB集合中检索出该进程的PCB，从中读出该进程的状态。

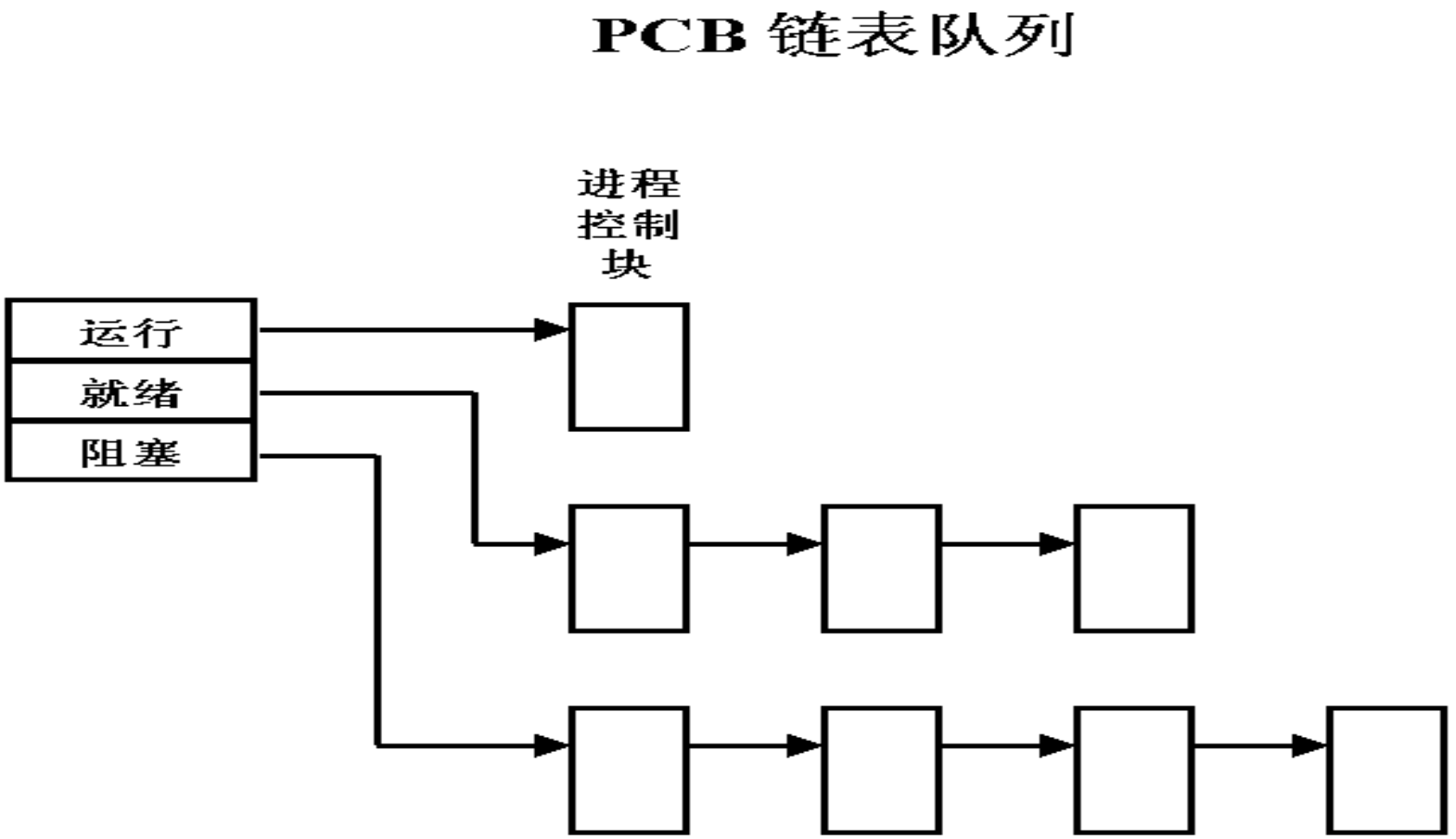
2) 若被终止进程正处于执行状态，应立即终止该进程的执行，并置调度标志为真，用于指示该进程被终止后应重新进行调度。

3) 若该进程还有子孙进程，还应将其所有子孙进程予以终止，以防他们成为不可控的进程。

4) 将被终止进程所拥有的全部资源，或者归还给其父进程， 或者归还给系统。

5) 将被终止进程(它的PCB)从所在队列(或链表)中移出， 等待其他程序来搜集信息。

进程控制块的组织：



# 2.2.3 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

进程调用阻塞原语block(),把自己阻塞。它是进程自身的一种主动行为。

然后，PCB中的状态由“执行”改为阻塞，并将PCB插入阻塞队列。

最后，转调度程序将处理机分配给另一就绪进程，并进行切换.

## 3. 进程唤醒过程

调用唤醒原语wakeup()，将等待该事件的进程唤醒。

执行的过程是：

1）首先把被阻塞的进程从阻塞队列中移出；

2）将其PCB中的现行状态由阻塞改为就绪；

3）将该PCB插入到就绪队列中。

# 2.2.4 进程的挂起与激活

## 1. 进程的挂起

系统用原语suspend()将指定进程或处于阻塞状态的进程挂起。

执行过程是：

1）检查被挂起进程的状态，活动就绪改为静止就绪；活动阻塞状态改为静止阻塞。

2）把该进程的PCB复制到某指定的内存区域。

3）若被挂起的进程正在执行，则转向调度程序重新调度。

# 2. 进程的激活过程

系统用原语active()将指定进程激活。

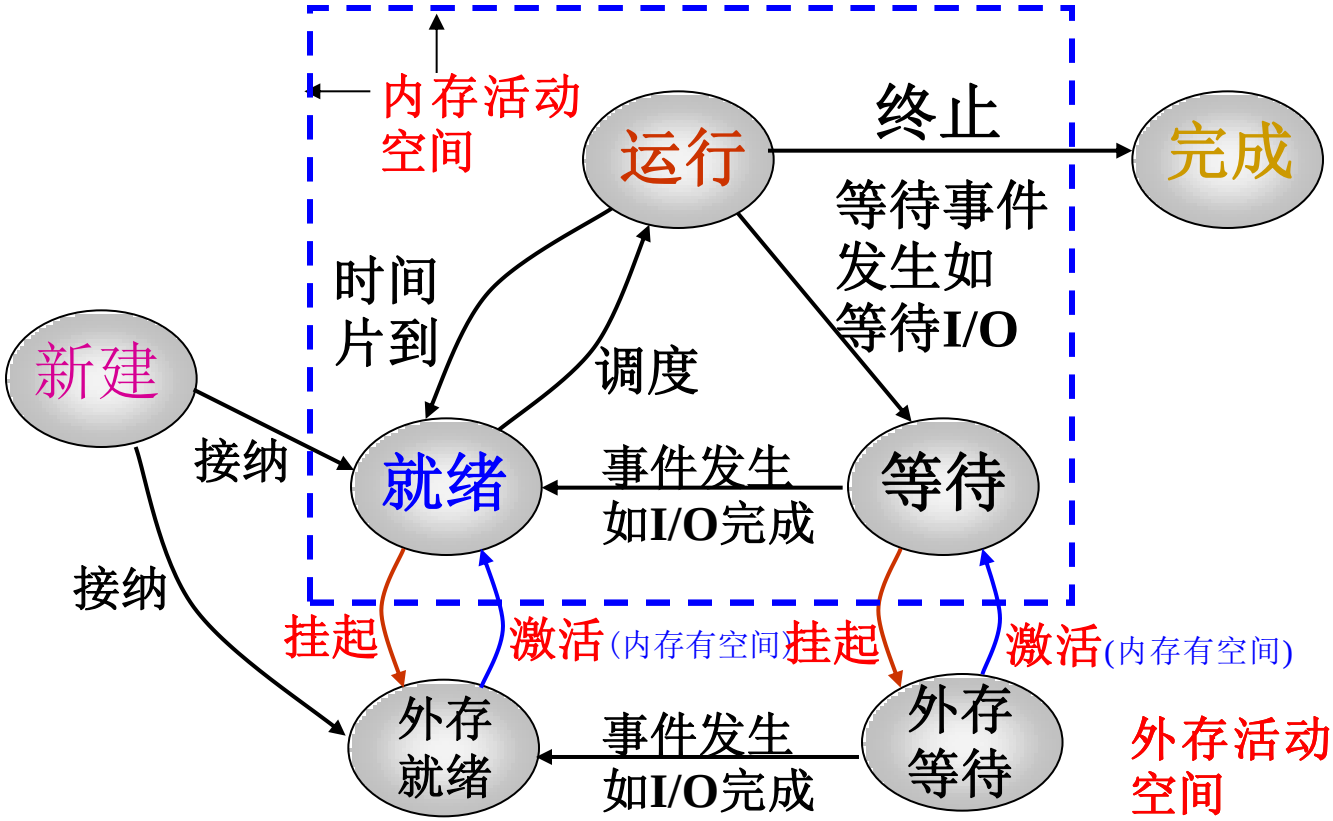
执行过程是:

1）先将进程从外存调入内存，静止就绪改为活动就绪；静止阻塞改为活动阻塞。

2）假如采用的是抢占调度策略，则每当有新进程进入就绪队列时，应检查是否要进行重新调度，即由调度程序将被激活进程与当前进程进行优先级的比较，如果被激活进程的优先级更低，就不必重新调度；否则，立即剥夺当前进程的运行，把处理机分配给刚被激活的进程。

进程的挂起状态（从内存交换到外存）

进程的激活状态（从外存交换到内存）




具有挂起状态的进程状态转换图

# 第二章 进程管理

2.1 进程的基本概念

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 管程机制

2.6 进程通信

2.7 线程

# 2.3 进 程 同 步

## 2.3.1 进程同步的基本概念

### 1. 两种形式的制约关系

**(1) 间接相互制约关系**

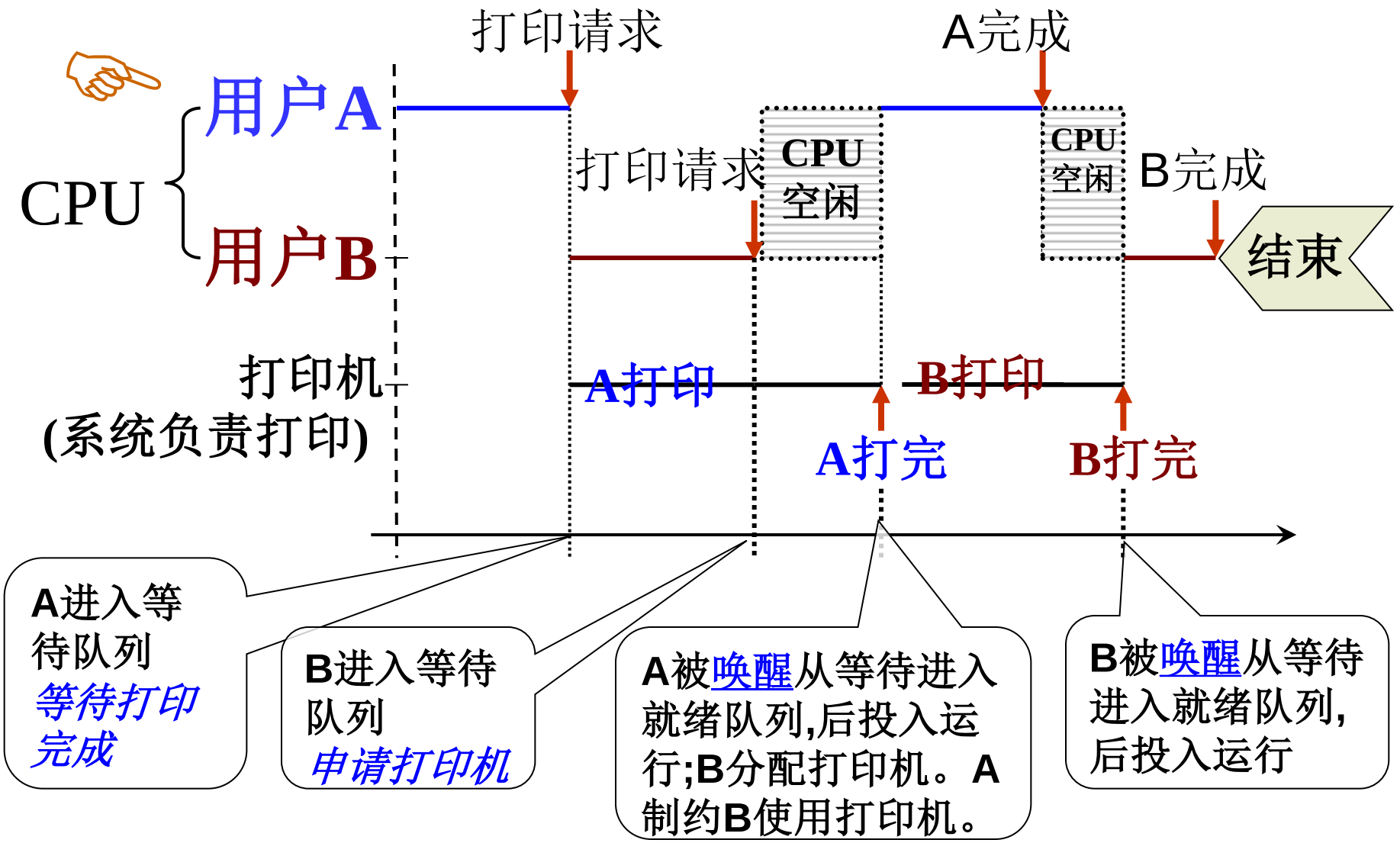
非相关进程对公有资源竞争造成的制约

**(2) 直接相互制约关系**

基于进程间的合作，比如：进程通信

# 间接制约关系示例：

下一步



# 2. 临界资源

**概念：并发进程间必须采用互斥方式共享的那类资源。**

例：生产者-消费者问题

**描述：**

一个具有n个缓冲区的缓冲池，生产者进程每次放入一个产品到缓冲区中； 消费者进程可从一个缓冲区中取走产品去消费。

**约束条件：**

它们之间必须保持同步，即不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。

# 临界资源（续）

**需要的数据结构：**

1、用一个数组来表示这n个(0，1，…，n-1)缓冲区的缓冲池。

2、输入指针in来指示下一个可投放产品的缓冲区；

3、输出指针out来指示下一个可从中获取产品的缓冲区，

输入指针加1表示成 in：=(in+1)mod n;

输出指针加1表示成out：=(out+1) mod n。

(in+1) mod n=out，表示缓冲池满；

in=out， 则表示缓冲池空。

4、一个整型变量counter, 其初始值为0。

放产品：counter加1；

取产品：counter减1。

# 临界资源

**对于共享变量counter，生产者做加1操作，消费者做减1操作，可用下面的形式描述：**

**register1：= counter;　　register2：= counter;**

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

**counter：= register1;　　counter：= register2;**

# 3. 临界区(CRITICAL SECTION)

**临界区：每个进程中，访问临界资源的那段代码**

可把一个访问临界资源的循环进程描述如下：

```
repeat
        entry section
            critical section;
        exit section
             remainder section;
until false;
```

# 4. 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。

4) 让权等待

# 2.3.2 信号量机制

## 1. 整型信号量

最初由Dijkstra把整型信号量定义为一个整型量，除初始化外，仅能通过两个标准的原子操作(Atomic Operation) wait(S)和signal(S)来访问。这两个操作一直被分别称为P、V操作。

**P操作和V操作可分别描述为：**

**wait(S): while S≤0 do no-op**

**S：=S-1;**

**signal(S): S：=S+1;**

# 2. 记录型信号量

在整型信号量机制中的wait操作，只要是信号量S≤0，就会不断地测试。因此，该机制并未遵循“让权等待”的准则，而是使进程处于“忙等”的状态。

记录型信号量机制，则是一种不存在“忙等”现象的进程同步机制。但在采取了“让权等待”的策略后，又会出现多个进程等待访问同一临界资源的情况。

**区别：**在记录型信号量机制中，除了需要一个用于代表资源数目的整型变量value外，还应增加一个进程链表L，用于链接上述的所有等待进程。

# 记录型信号量

**它所包含的上述两个数据项可描述为：**

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

**wait(S)和signal(S)操作可描述为：**

```
procedure wait(S)
var S: semaphore;
  begin
   S.value：   =S.value-1;
   if S.value＜0 then
        block(S,L)
  end
```

```
procedure signal(S)
var S: semaphore;
   begin
    S.value：   =S.value+1;
    if S.value≤0 then
         wakeup(S,L);
   end
```

# 3. AND型信号量

**基本思想是：**

将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程，待进程使用完后再一起释放。只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源，也不分配给他。

换言之，对若干个临界资源的分配，采取原子操作方式：要么全部分配到进程，要么一个也不分配。

# 3. AND型信号量

**在两个进程中都要包含两个对Dmutex和Emutex的操作：**

```
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

# 4. 信号量集

**Swait($S_1$, $t_1$, $d_1$, …, $S_n$, $t_n$, $d_n$)**

**if $S_i \geq t_1$ & … & $S_n \geq t_n$ then**

**for i：=1 to n**

**do $S_i$：=$S_i$-$d_i$;**

**end for**

**else**

**把那些满足$S_i < t_i$ 的进程$S_i$ 放在等待队列中，再把程序计数器设置为Swait操作的初始值。**

**end if**

# 4. 信号量集

**一般“信号量集”的几种特殊情况：**

**(1) Swait(S, d, d)。**

此时在信号量集中只有一个信号量S， 但允许它每次申请d个资源，当现有资源数少于d时，不予分配。

**(2) Swait(S, 1, 1)。**

此时的信号量集已蜕化为一般的记录型信号量(S＞1时)或互斥信号量(S=1时)。

**(3) Swait(S, 1, 0)。**

这是一种很特殊且很有用的信号量操作。当S≥1时，允许多个进程进入某特定区；当S变为0后，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。

# 4. 信号量集

**Ssignal($S_1$, $d_1$, …, $S_n$, $d_n$)**

**for i：=1 to n**

**do Si ：=Si+di;**

**end for**

**把等待队列中的进程Si (i=1,…,n) 移到就绪队列中。**

# 第二章 进程管理

# 用记录型信号量解决生产者–消费者问题

## 2.4.1 生产者–消费者问题

假定在生产者和消费者之间的公用缓冲池中，具有n个缓冲区，这时可利用互斥信号量mutex实现诸进程对缓冲池的互斥使用；

利用信号量empty和full分别表示缓冲池中空缓冲区和满缓冲区的数量。

假定这些生产者和消费者相互等效，只要缓冲池未满，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。

# 实现代码

```
Var mutex, empty, full:semaphore：=1,n,0;
    buffer:array［0, …, n-1］ of item;
    in, out: integer：=0, 0;
    begin
     parbegin
      proceducer:begin
            repeat  …   producer an item nextp;   …
              wait(empty);
             wait(mutex);
              buffer(in)：=nextp;
              in：=(in+1) mod n;
              signal(mutex);
               signal(full);
            until false;
     end
```

# 消费者进程实现代码

```
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：

- 在每个程序中用于实现互斥的wait(mutex)和signal(mutex)必须成对地出现；

- 对资源信号量empty和full的wait和signal操作，同样需要成对地出现，但它们可以分别处于不同的程序中。

例如，wait(empty)在计算进程中，而signal(empty)则在打印进程中，计算进程若因执行wait(empty)而阻塞，则以后将由打印进程将它唤醒；

- 在每个程序中的多个wait操作顺序不能颠倒。

应先执行对资源信号量的wait操作，然后再执行对互斥信号量的wait操作，否则可能引起进程死锁。

# 2.4.2 哲学家进餐问题

经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。其描述如下：

**Var chopstick: array［0, …, 4］ of semaphore;**

# 部分代码

**信号量均被初始化为1， 第i位哲学家的活动可描述为：**

```
repeat
        wait(chopstick［i］);
       wait(chopstick［(i+1) mod 5］);
              … eat;    …
        signal(chopstick［i］);
        signal(chopstick［(i+1) mod 5］);
                …
             think;
until false;
```

# 几种避免死锁的方案

(1) 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

(2) 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。

(3) 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。

# 2.4.3 读者-写者问题

**互斥信号量Wmutex：**

实现Reader与Writer进程间在读或写的互斥

**整型变量Readcount：**

表示正在读的进程数目。

**互斥信号量rmutex**

因为Readcount是一个可被多个Reader进程访问的临界资源

# 实现代码

```
Var rmutex, wmutex : semaphore：=1,1;
Readcount: integer：=0;
   begin   parbegin
      Reader: begin
      repeat     wait(rmutex);
       if Readcount=0 then
          wait(wmutex);
        Readcount：   =Readcount+1;
       signal(rmutex);
         …    perform read operation;   …
        wait(rmutex);
        Readcount：   =Readcount-1;
     if Readcount=0 then
     signal(wmutex);    signal(rmutex);
     until false;
    end
```

# 实现代码（续）

```
writer: begin
        repeat  wait(wmutex);
            …perform write operation;  …
        signal(wmutex);
        until false;
    end
parend
 end
```

# 第二章 进程管理

# 2.5 管 程 机 制

## 2.5.1 管程的基本概念

### 1. 管程的定义

管程由三部分组成：

① 局部于管程的共享变量说明；

② 对该数据结构进行操作的一组过程；

③ 对局部于管程的数据设置初始值的语句。

此外，还须为管程赋予一个名字。

# 管程的示意图

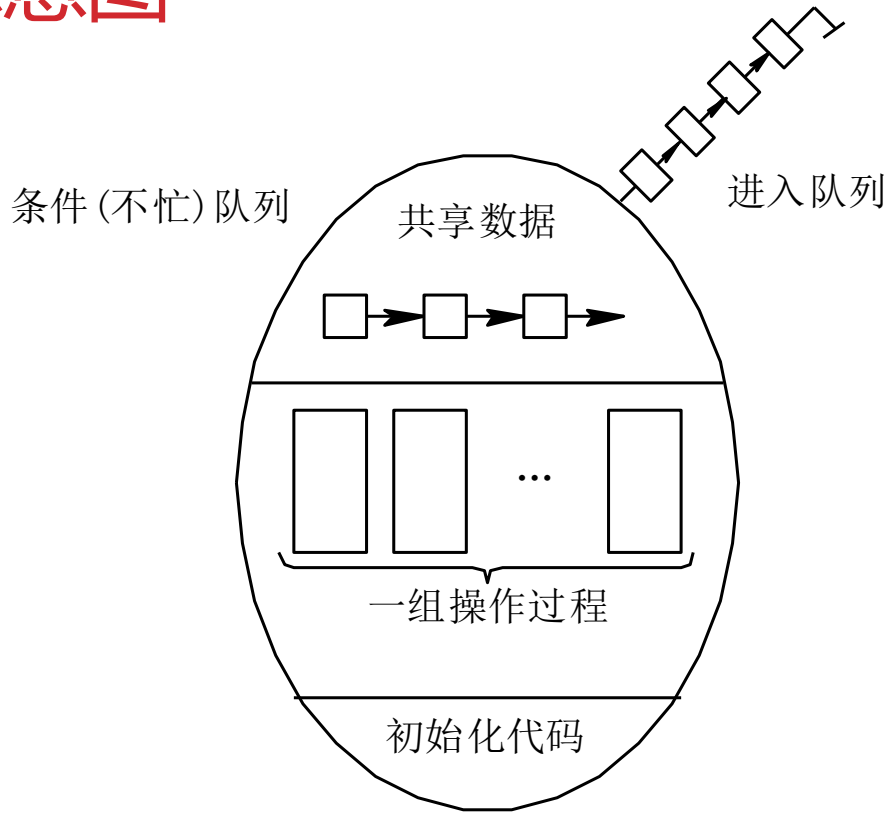



**图 2-11 管程的示意图**

# 管程的语法

```
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
```

# 2. 条件变量

管程中对每个条件变量，都须予以说明，其形式为：

Var x, y:condition。

该变量应置于wait和signal之前，即可表示为X.wait和X.signal。

X.signal操作的作用，是重新启动一个被阻塞的进程，但如果没有被阻塞的进程，则X.signal操作不产生任何后果。

注：这与信号量机制中的signal操作不同。因为，后者总是要执行s：=s+1操作，因而总会改变信号量的状态。

# 2.5.2 利用管程解决生产者-消费者问题

首先为它们建立一个管程，并命名为Proclucer-Consumer，或简称为PC。其中包括两个过程：

(1) put(item)过程

生产者利用该过程将自己生产的产品投放到缓冲池中，并用整型变量count来表示在缓冲池中已有的产品数目，当count≥n时， 表示缓冲池已满， 生产者须等待。

(2) get(item)过程

消费者利用该过程从缓冲池中取出一个产品，当count≤0时，表示缓冲池中已无可取用的产品， 消费者应等待。

# 建立一个管程PROCLUCER-CONSUMER

```
type producer-consumer=monitor
    Var in,out,count:integer;
     buffer:array［0,…,n-1］ of item;
     notfull, notempty:condition;
procedure entry put(item)
begin
     if count≥n then notfull.wait;
     buffer(in)：  =nextp;  in：  =(in+1) mod n;
     count：  =count+1;
     if notempty.queue  then  notempty.signal;
end
```

# 建立一个管程PROCLUCER-CONSUMER

```
procedure entry get(item)
begin
    if count≤0 then notempty.wait;
    nextc：  =buffer(out);
    out：  =(out+1) mod n;
    count：  =count-1;
   if notfull.quene then notfull.signal;
end
begin
    in：  =out：  =0;
    count：  =0
end
```

# 管程解决生产者-消费者问题

**producer:**

```
begin
    repeat
    produce an item in nextp;
    PC.put(item);
    until false;
end
```

**consumer:**

```
begin
    repeat
    PC.get(item);
    consume the item in nextc;
    until false;
 end
```

# 第二章 进程管理

# 2.6 进 程 通 信

**概念：**

并发执行的进程为了协调一致地完成指定的任务，进程间通过交换信息（如通过修改信号量）的方式进行联系。这种方式叫**进程通信。**

**分类：**

**低级通信：** 把进程间控制信息的交换。一般只传送一个或几个字节的信息，如P，V操作。低级通信以达到控制进程执行速度的作用。

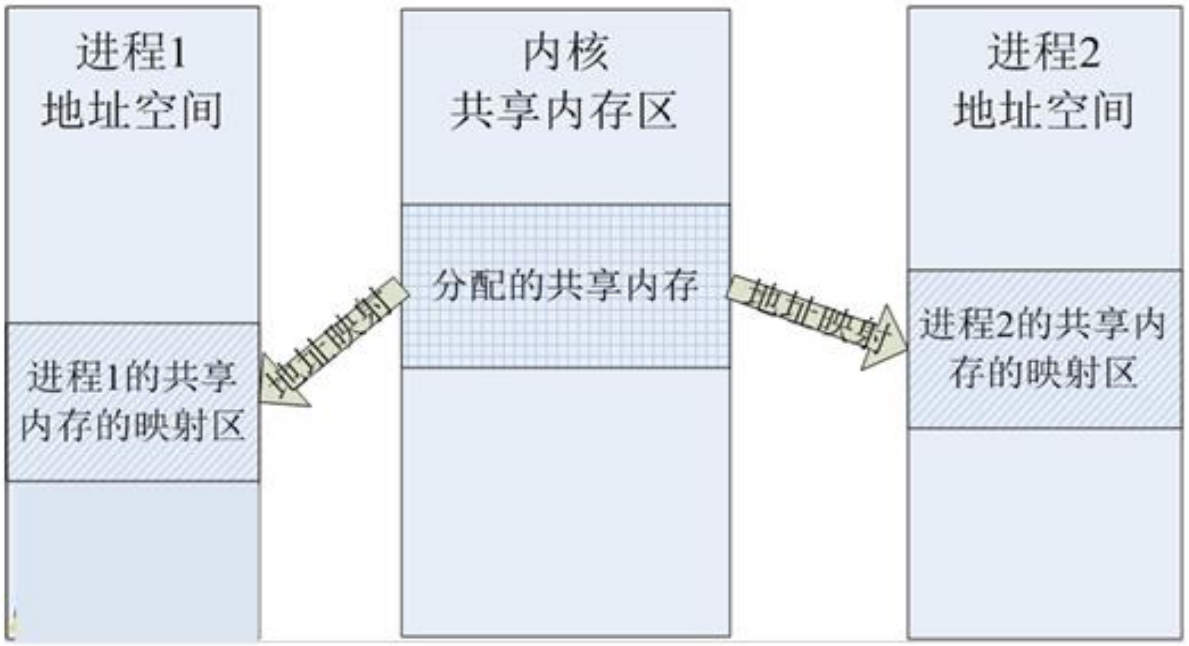
**高级通信：** 要传送大量数据。高级通信的目的不是为了控制进程的执行速度，而是为了交换信息。

# 2.6.1 进程通信的类型

**1. 共享存储器系统(Shared-Memory System)**

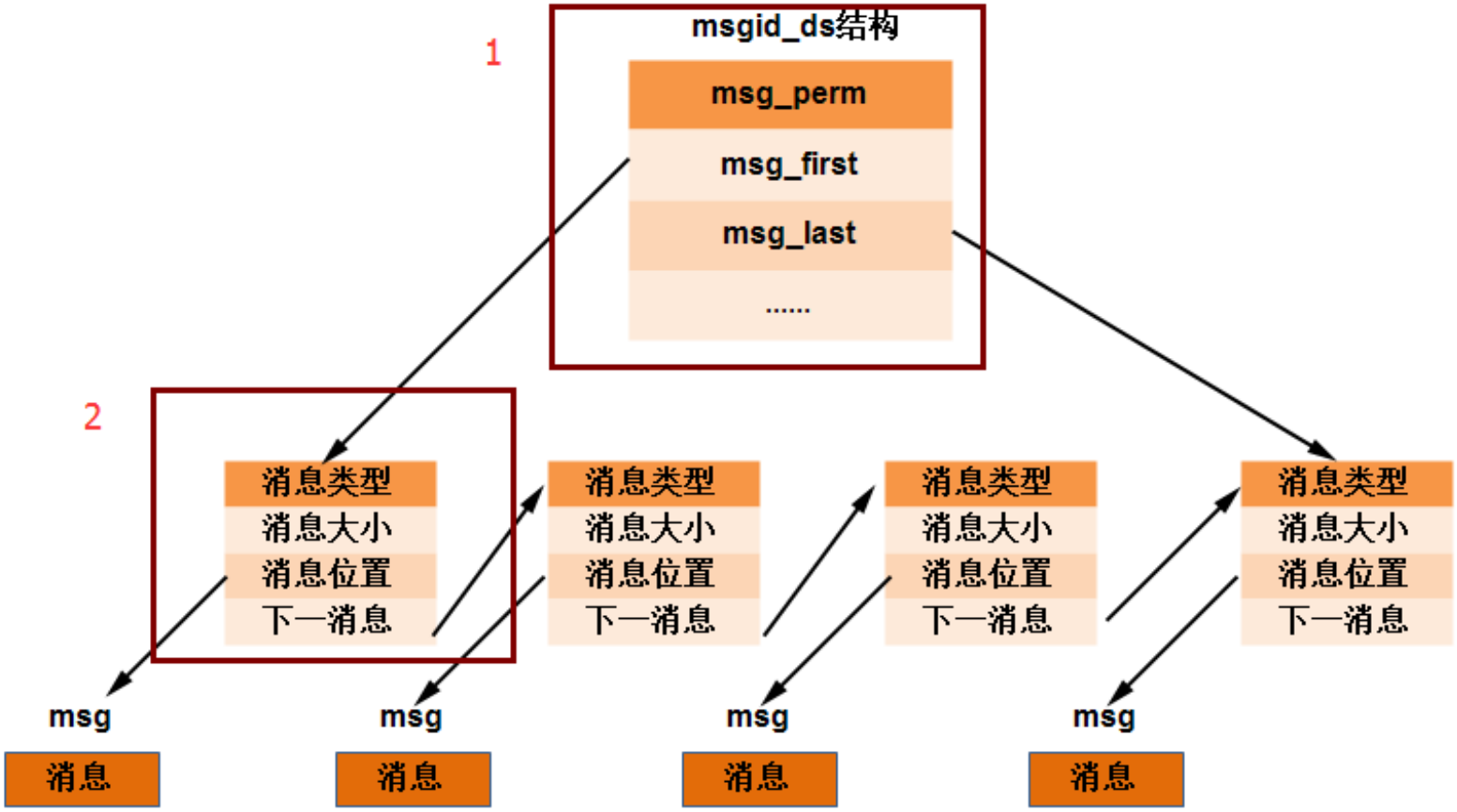
(1)基于共享数据结构

(2) 基于共享存储区



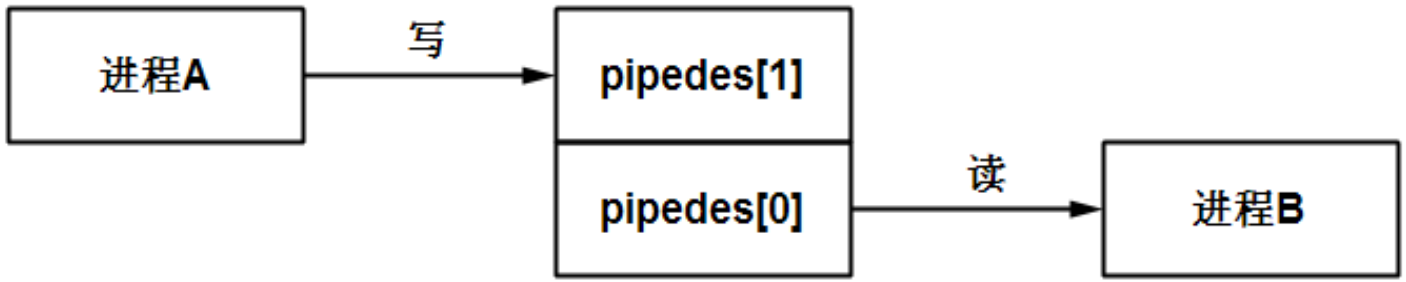
# 2. 消息传递系统(Message passing system)

进程间的数据交换，是以格式化的消息(message)为单位的；在计算机网络中，又把message称为报文。程序员直接利用系统提供的一组通信命令(原语)进行通信。



## 3. 管道(Pipe)通信

所谓“管道”，是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共享文件，又名pipe文件。



# 2.6.2 消息传递通信的实现方法

## 1.直接通信方式

发送进程利用OS所提供的发送命令，直接把消息发送给目标进程。此时，要求发送进程和接收进程都以显式方式提供对方的标识符:

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;

例如，原语Send($P_2$，$m_1$)表示将消息$m_1$发送给接收进程$P_2$；而原语Receive($P_1$，$m_1$)则表示接收由$P_1$发来的消息$m_1$。

# 2. 间接通信方式 –信箱

## (1) 信箱的创建和撤消。

进程可利用信箱创建原语来建立一个新信箱。创建者进程应给出信箱名字、信箱属性(公用、私用或共享)，当进程不再需要读信箱时，可用信箱撤消原语将之撤消。

## (2) 消息的发送和接收。

当进程之间要利用信箱进行通信时，必须使用共享信箱，并利用系统提供的通信原语Send、Receive进行通信。

# 信箱分类

## 信箱分为以下三类

### 1) 私用信箱

用户进程可为自己建立一个新信箱，当拥有该信箱的进程结束时，信箱也随之消失。

### 2) 公用信箱

它由操作系统创建，并提供给系统中的所有核准进程使用。通常，公用信箱在系统运行期间始终存在。

### 3) 共享信箱

它由某进程创建，在创建时或创建后，指明它是可共享的，同时须指出共享进程(用户)的名字。信箱的拥有者和共享者，都有权从信箱中取走发送给自己的消息。

# 2.6.3 消息传递系统实现的若干问题

## 1.通信链路

**根据连接方法，可分为两类：**

① 点—点连接通信链路，一条链路只连接两个结点

② 多点连接链路，指用一条链路连接多个(n＞2)结点。

**根据通信方式，可分成两种：**

① 单向链路，只允许发送进程向接收进程发送消息；

② 双向链路，既允许由进程A向进程B发送消息，也允许进程B同时向进程A发送消息。

## 2. 消息的格式

### 定长消息格式

这减少了对消息的处理和存储开销,但这对要发送较长消息的用户是不方便的。

### 变长的消息格式

即进程所发送消息的长度是可变的，须付出更多的开销，但方便了用户

## 3. 进程同步方式

(1) 发送进程阻塞、 接收进程阻塞。

(2) 发送进程不阻塞、 接收进程阻塞。

(3) 发送进程和接收进程均不阻塞。

# 2.6.4 消息缓冲队列通信机制

## 1. 消息缓冲队列通信机制中的数据结构

### (1) 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;       //指向下一个消息缓冲区的指针
end
```

# 1. 消息缓冲队列通信机制的数据结构

**(2) PCB中有关通信的数据项**

```
type processcontrol block=record
        …
        mq;      //消息队列队首指针
        mutex;   //消息队列互斥信号量
        sm;      //消息队列资源信号量
         …
end
```

# 2. 发送原语

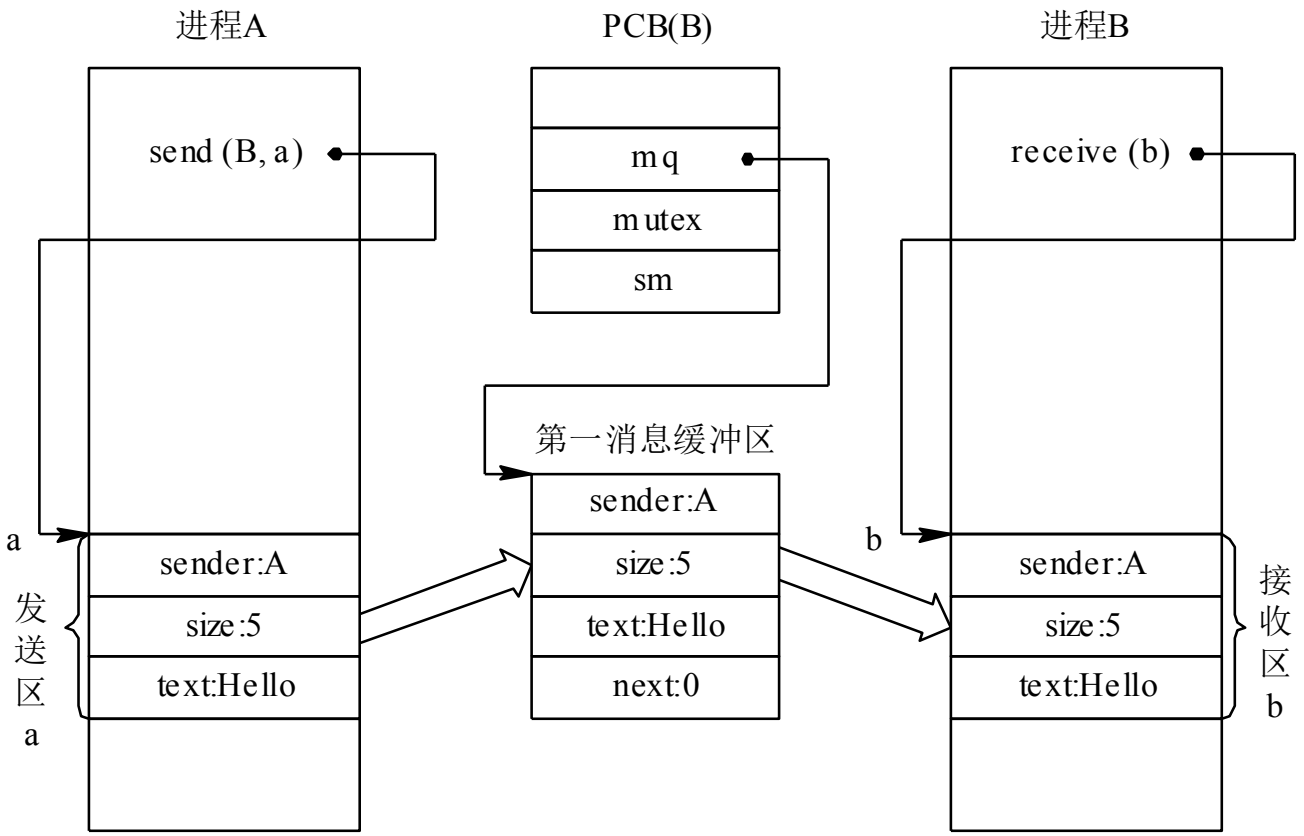



图 2 - 12 消息缓冲通信

# 第二章 进程管理

# 2.7 线 程

## 2.7.1 线程的基本概念

**为使程序能并发执行，系统还必须进行以下的一系列操作。**

1） 创建进程

2） 撤消进程

3） 进程切换

# 2. 线程的属性

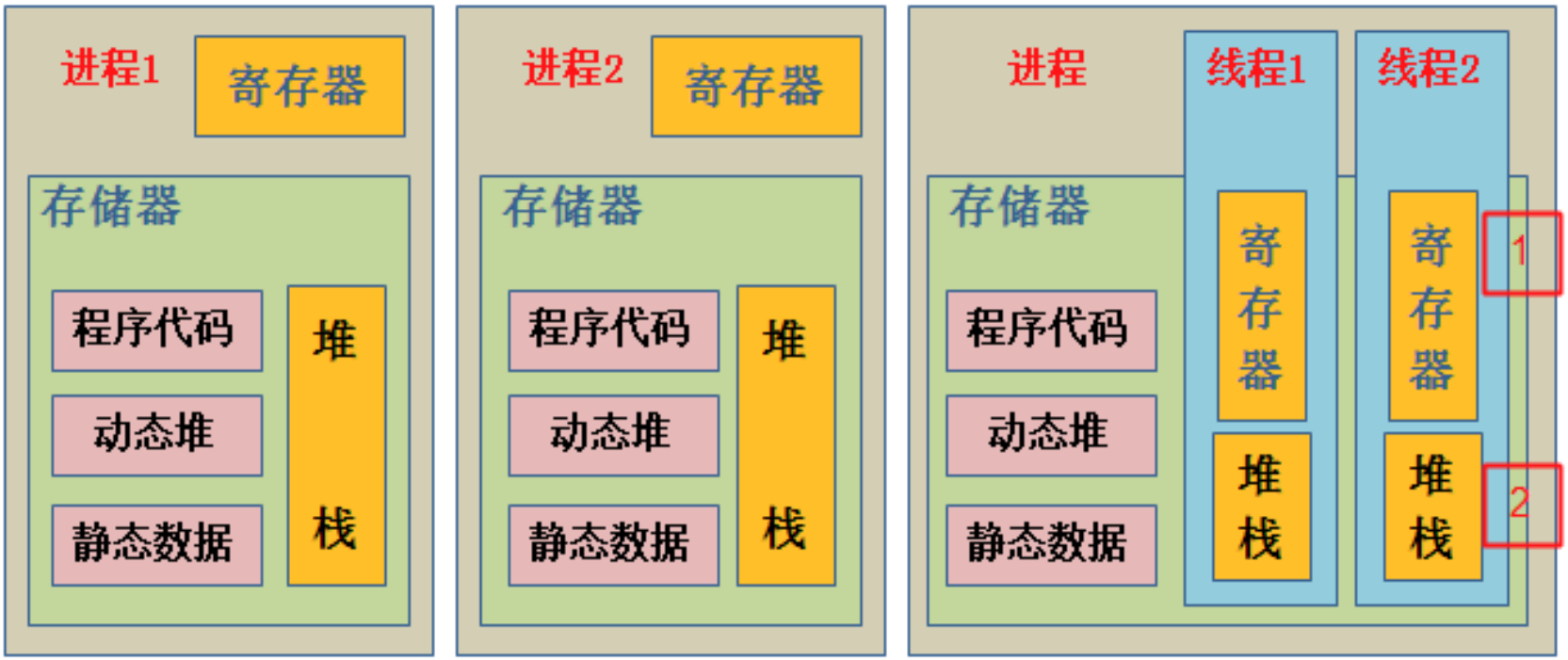
(1) 轻型实体。

(2) 独立调度和分派的基本单位。

(3) 可并发执行。

(4) 共享进程资源。

# 进程和线程的构造


进程1
寄存器
存储器
程序代码
动态堆
静态数据
堆
栈
进程2
寄存器
存储器
程序代码
动态堆
静态数据
堆
栈
进程
线程1
线程2
存储器
程序代码
动态堆
静态数据
寄存器
堆栈
寄存器
堆栈
1
2

# 3. 线程的状态

**(1) 状态参数。**

**在OS中的每一个线程都可以利用线程标识符和一组状态参数进行描述。状态参数通常有这样几项：**

① **寄存器状态，** 包括程序计数器PC和堆栈指针中的内容；

② **堆栈，** 在堆栈中通常保存有局部变量和返回地址；

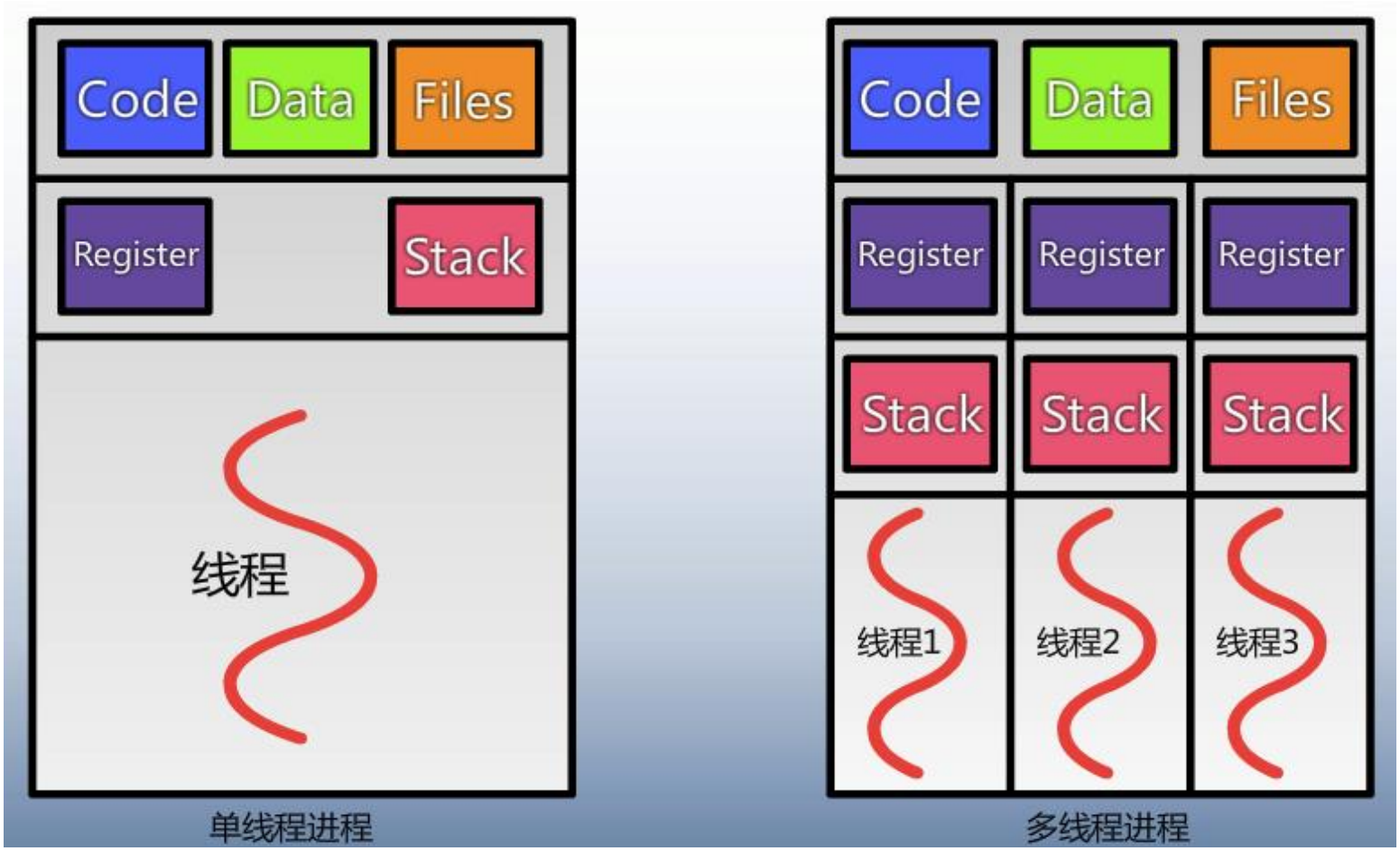
③ **线程运行状态，** 用于描述线程正处于何种运行状态；

④ **优先级，** 描述线程执行的优先程度；

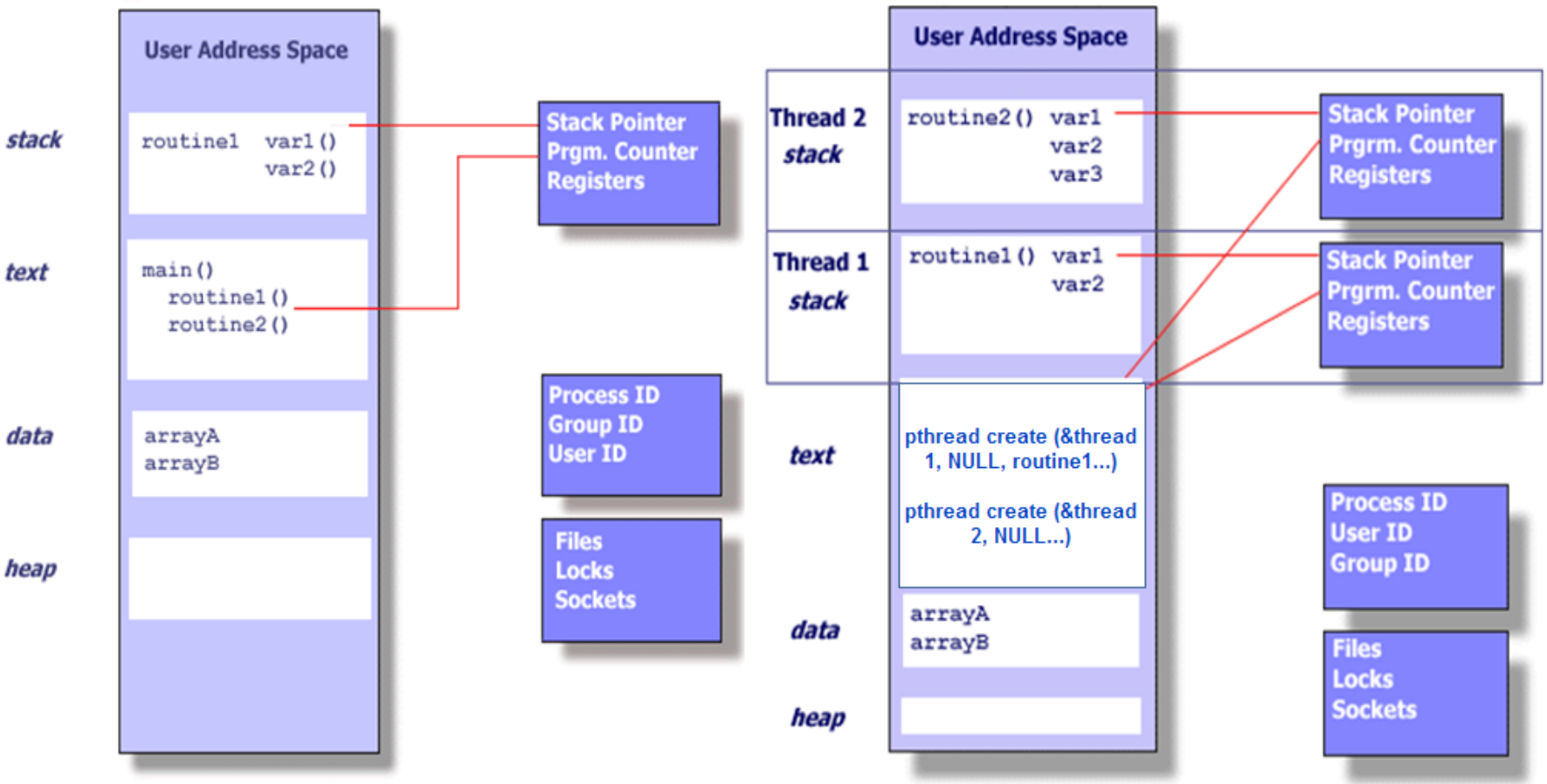
⑤ **线程专有存储器，** 用于保存线程自己的局部变量拷贝；

⑥ **信号屏蔽，** 即对某些信号加以屏蔽。

# 线程的内存布局


Code
Data
Files
Register
Stack
线程
单线程进程
Code
Data
Files
Register
Register
Register
Stack
Stack
Stack
线程1
线程2
线程3
多线程进程

# 进程和线程的比较




进　　程　　线　　程

# 3.线程的状态

## (2) 线程运行状态。

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，表示线程正获得处理机而运行；

② 就绪状态， 指线程已具备了各种执行条件，一旦获得CPU便可执行的状态；

③ 阻塞状态，指线程在执行中因某事件而受阻，处于暂停执行时的状态。

# 4. 线程的创建和终止

**终止线程的方式有两种：**

- 在线程完成了自己的工作后自愿退出；
- 另一种是线程在运行中出现错误或由于某种原因而被其它线程强行终止。

# 5. 多线程OS中的进程

在多线程OS中，进程是作为拥有系统资源的基本单位，通常的进程都包含多个线程并为它们提供资源，但此时的进程就不再作为一个执行的实体。

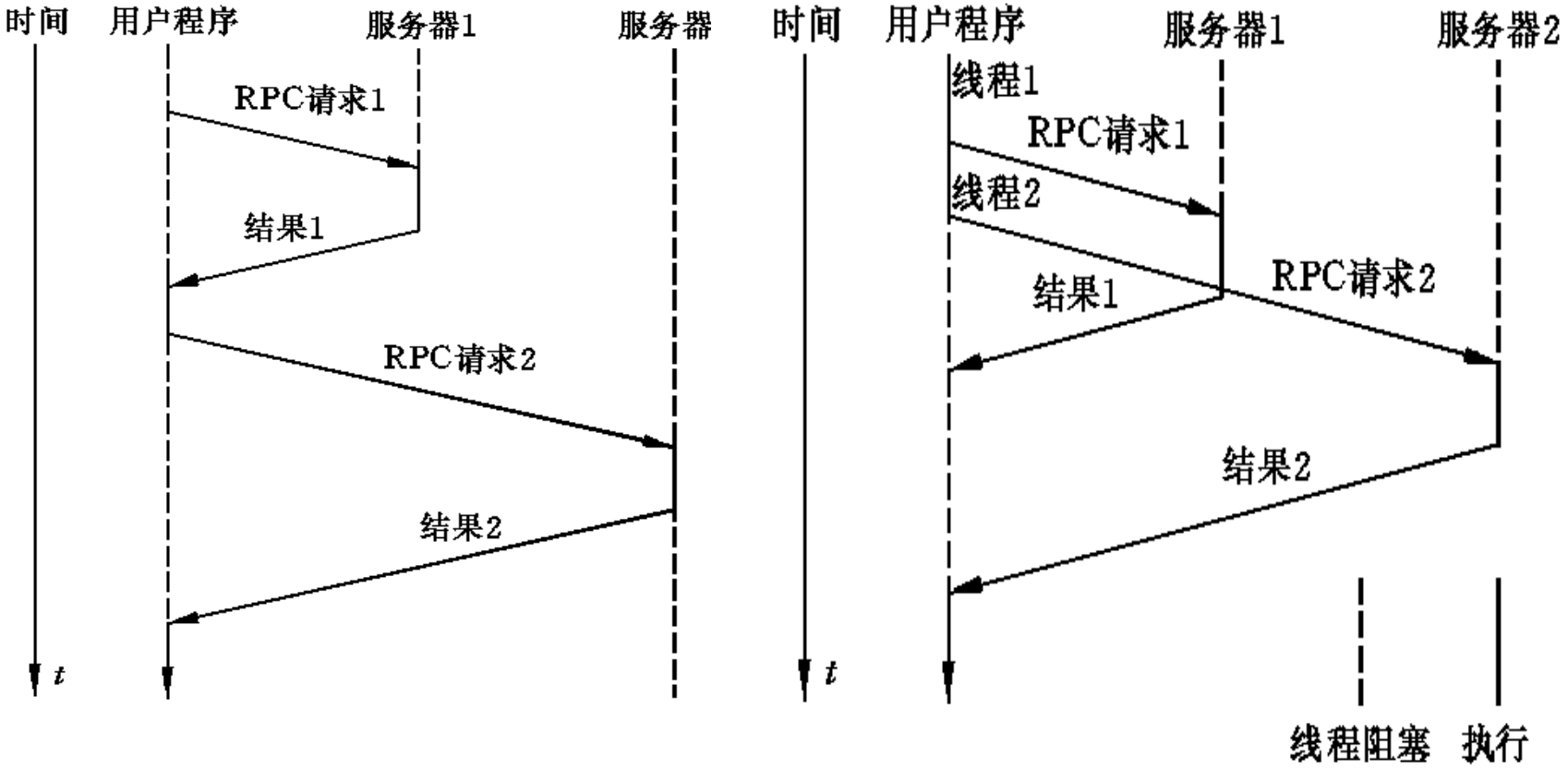
多线程OS中的进程有以下属性：

(1) 作为系统资源分配的单位。

(2) 可包括多个线程。

(3) 进程不是一个可执行的实体。

# 举例：一个用户主机向两台服务器进行远程调用（RPC）




(a) 单线程时的RPC请求处理　　(b) 多线程时的RPC请求处理

# 2.7.2 线程间的同步和通信

**1. 互斥锁**

互斥锁可以有两种状态， 即开锁(unlock)和关锁(lock)状态。

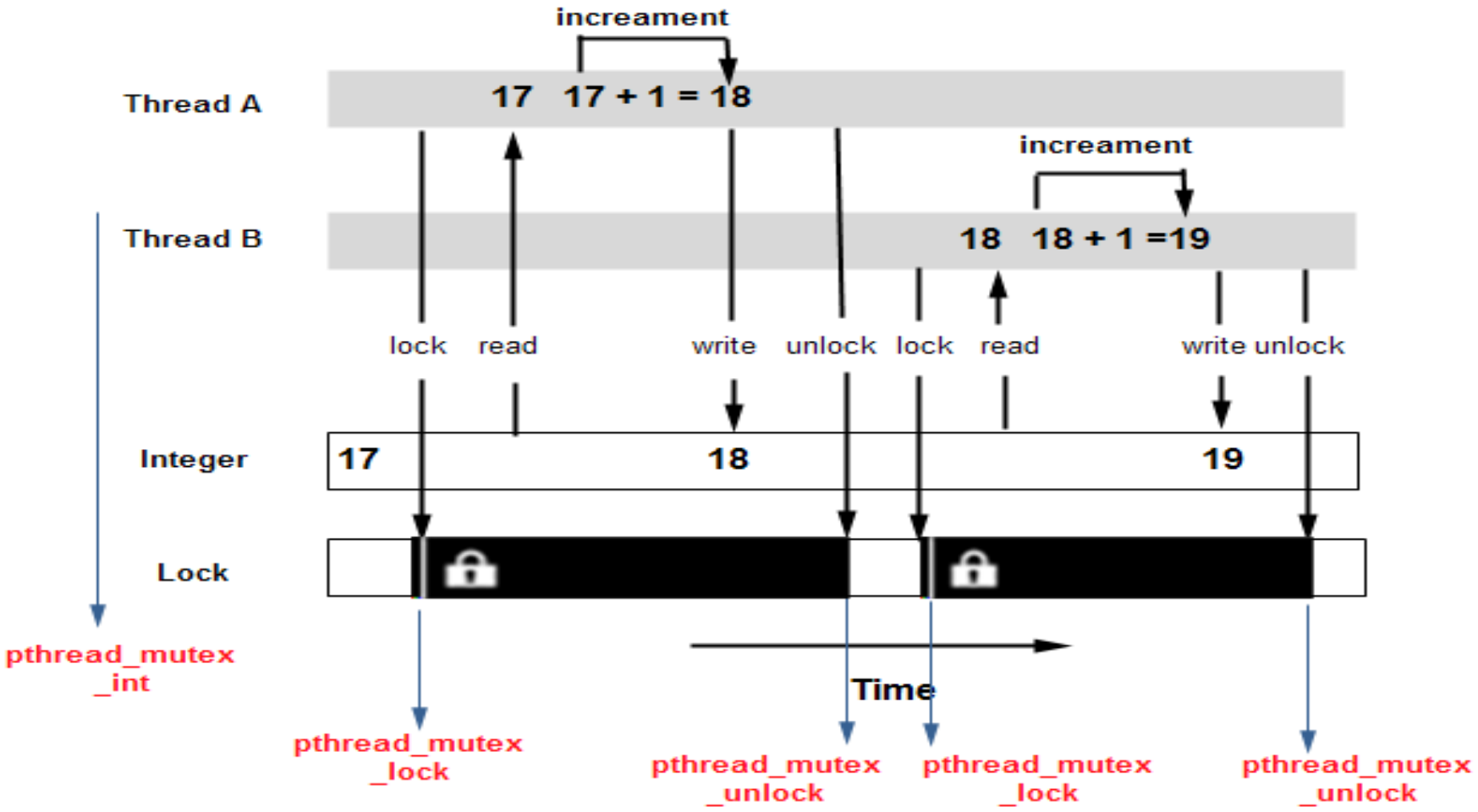
用于短期锁定，主要是用来保证对临界区的互斥进入。

**2. 条件变量**

在创建一个互斥锁时便联系着一个条件变量。

条件变量用于线程的长期等待， 直至所等待的资源成为可用的。

# 线程互斥锁示意图


increament
Thread A
17
17 + 1 = 18
increament
Thread B
18
18 + 1 =19
lock
read
write
unlock
lock
read
write
unlock
Integer
17
18
19
Lock
pthread_mutex
_int
Time
pthread_mutex
_lock
pthread_mutex
_unlock
pthread_mutex
_lock
pthread_mutex
_unlock

# 条件变量 V.S. 互斥锁

## 条件变量

生产者：

```
pthread_mutex_lock(lock_s);
sum++;
pthread_mutex_unlock(lock_s);
if(sum>=100)
pthread_cond_signal(&cond_sum_ready);
```

消费者：

```
pthread_mutex_lock(lock_s);
while(sum<100)
pthread_cond_wait(&cond_sum_ready, &lock_s);
printf("sum is over 100");
sum=0;
pthread_mutex_unlock(lock_s);
return OK;
```

## 互斥锁

生产者：

```
pthread_mutex_lock(lock_s);
sum++;
pthread_mutex_unlock(lock_s);
```

消费者：

```
pthread_mutex_lock(lock_s);
if(sum<100)
{
   printf("sum reaches 100!");
   pthread_mutex_unlock(lock_s);
}
else
{
   pthread_mutex_unlock(lock_s);
   my_thread_sleep(100);
   return OK;
}
```

## 3. 信号量机制

信号量只能进行两种操作来等待和发生信号：P操作和V操作。

P操作相当于获得了这块区域的使用权，其他的进程是进不来的

（如果信号量S>0就减1，如果它的值为0，就挂起）

V操作表示释放

（如果有其他进程因等待而被挂起，就恢复运行；　如果没有其他进程因等待而挂起，就给S加1）

# 2.7.3 内核支持线程和用户级线程

**1. 内核支持线程**

无论是用户进程中的线程，还是系统进程中的线程，他们的创建、撤消和切换等，都是依靠内核实现的。

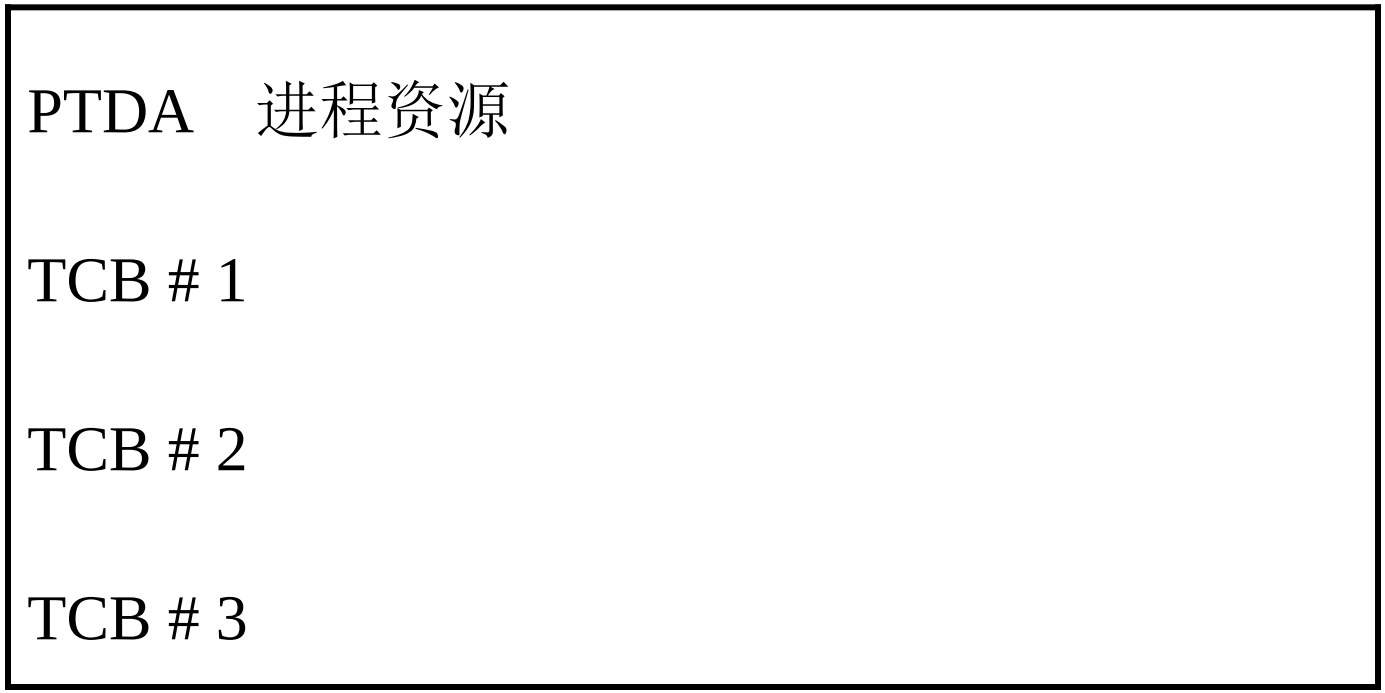
**2.用户级线程**

仅存在于用户空间中。对于这种线程的创建、 撤消、线程之间的同步与通信等功能，都无须利用系统调用来实现。

**用户级线程 的管理过程全部由用户程序完成，操作系统内核只对进程进行管理。**

# 2.7.4 线程控制

**1.内核支持线程的实现**



# 2. 用户级线程的实现

## 1) 运行时系统(Runtime System)

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。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. 用户级线程的实现

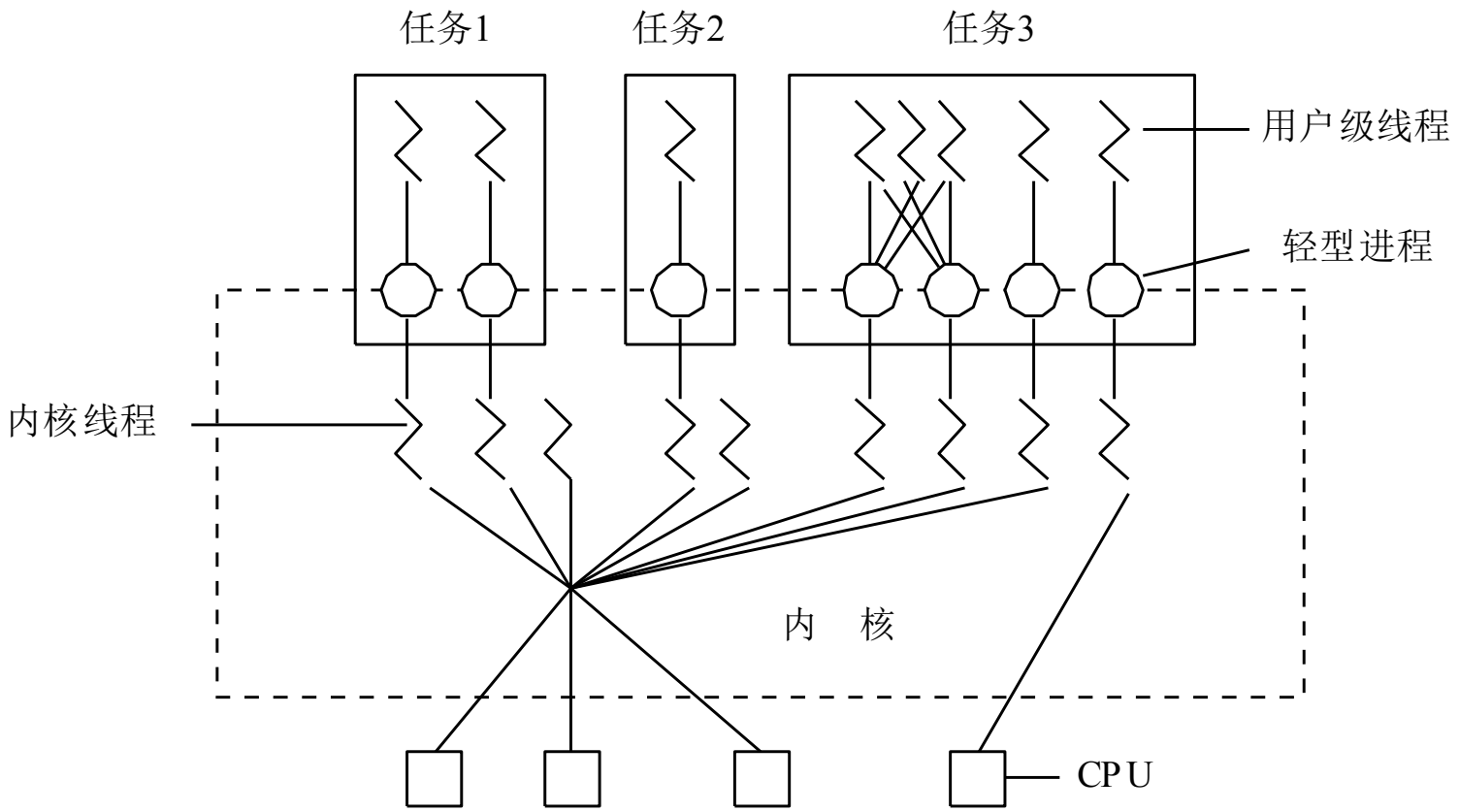
## 2) 内核控制线程

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，它们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，只要将用户级线程连接到一个LWP上，此时它便具有了内核支持线程的所有属性。

# 利用轻型进程作为中间系统



# 作业

1、在P、V操作中，若系统中总共有n个进程，当信号量S初始化m时（n>m），则处于等待状态的进程至少有_____个。（并分析原因）

2、对于进程的描述，下列哪个选项是错误的？（　）

A 进程可分为用户级进程和系统级进程；

B 每个进程都有自己的进程控制块（PCB）；

C 进程状态的切换需要操作系统内核来控制；

D 对于文件的读写操作，不需要创建进程来完成。

3、举一个例子说明进程如何在三种状态（执行、就绪、阻塞）之间转换。注意：要结合例子，说明这四种转换及其原因。

4、用一个整型信号量写出一个不会死锁的哲学家进餐问题的算法。