# 第三章 进程管理

系统中的一个工作单位，描述了程序的执行过程，系统在此基础上进行工作流程的控制和系统资源的分配。

在现代计算机系统中，以进程的观点来设计和研究操作系统的。因此，只有深刻理解进程的概念，才能够很好地理解OS各部分功能和工作。

本章，首先引入“进程”的概念，指出其特点，然后，逐步介绍进程的管理，包括进程的建立、调度、同步控制等。

# 本章主要内容

# 3.1 进程的概念

## 3.1.1 进程的引入

1．单道程序的特点 ：

顺序性

封闭性(独占性)

不可再现性

## 2．多道程序系统中程序的特点：

间断性（间接制约和直接制约）

失去封装性

不可再现性

例3-1：设有两道程序CP和PP，它们共享一个变量n，其初值为0。CP程序循环10000次，做n=n+1；PP程序将n的值打印。CP和PP可分别描述如下：

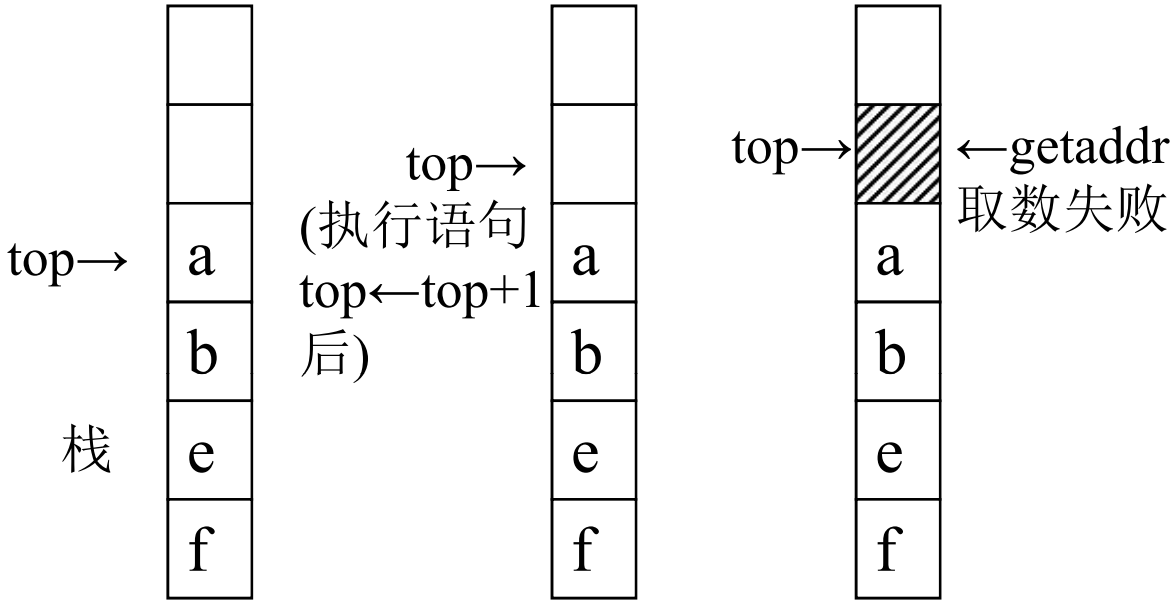
```
CP（）                          PP（）
 {                               {
    while(n<10000)                 printf(“n=%d\n”,n);
    n=n+1;                       }
}
```

结果：可能，n=10000，0，2000…

## 3．进程概念的引入

附加例：共享堆栈，取数和存数。

```
Cobegin      //并发
 getaddr(top)
      {
           var  r
           r=top;
           top=top-1;
           return(r);
       }
 reladdr(var blk)
      {
          top=top+1;
          (top)=blk;
      }
Coend   //结束
```



考虑一下：

为什么程序的执行具有不可再现性？

**共享软硬件资源**

为了控制和协调各程序执行过程中对软硬件资源的共享和竞争，在操作系统中引入了进程的概念来反映和刻画系统和用户程序的活动。

## 3.1.2 进程的定义

1. **进程的定义：**程序在某个数据集上的执行过程和分配资源的基本单位。

1. 其中较典型的进程定义有：

(1) 进程是程序的一次执行。

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

2. 进程和程序的区别： 例：科学家

静态与动态；

长久与暂时；

组成不同。

## 3. 进程的特征：

并发性

动态性

独立性

异步性

结构性

## 3.1.3 引入进程的利弊

利：

1. 改善资源利用率
2. 提高系统吞吐量

弊：

1. 空间开销
2. 时间开销

# 3.2 进程控制块和进程的状态

## 3.2.1进程的状态及其变化

### 1. 三种基本状态

进程执行时的间断性决定了进程可能具有多种状态。事实上，运行中的进程可能具有以下三种基本状态。

1） 就绪(Ready)状态

当进程已分配到除CPU以外的所有必要资源后，只要再获得CPU，便可立即执行，进程这时的状态称为就绪状态。所有处于就绪状态的进程排成一个就绪队列。

2）执行状态

进程已获得CPU，其程序正在执行。

（单处理机和多处理机）

3）阻塞状态

正在执行的进程由于发生某事件而暂时无法继续执行时，便放弃处理机而处于暂停状态，把这种暂停状态称为阻塞状态，有时也称为等待状态。致使进程阻塞的典型事件有：请求I/O，申请缓冲空间等。

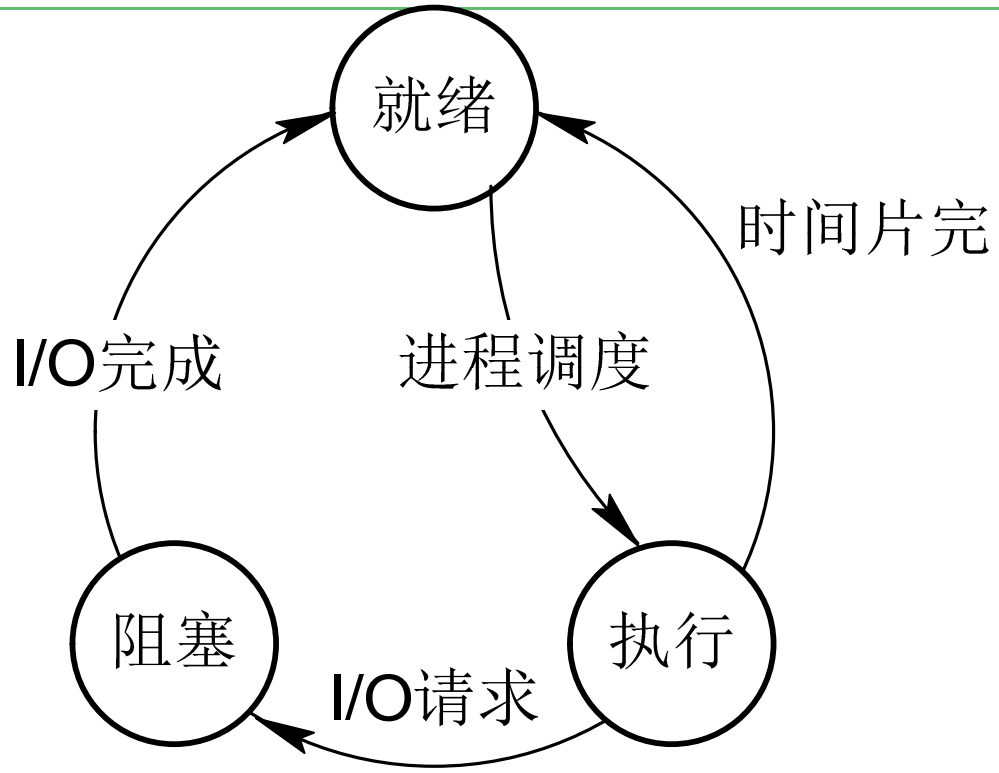



图2-5 进程的三种基本状态及其转换

## 2. 挂起状态

1）引入挂起状态的原因

在不少系统中进程只有上述三种状态，但在另一些系统中，又增加了一些新状态，最重要的是挂起状态。引入挂起状态的原因有：

(1) 终端用户的请求。当终端用户在自己的程序运行期间发现有可疑问题时，希望暂时使自己的程序静止下来。亦即，使正在执行的进程暂停执行；若此时用户进程正处于就绪状态而未执行，则该进程暂不接受调度，以便用户研究其执行情况或对程序进行修改。我们把这种静止状态称为挂起状态。

(2)父进程请求。有时父进程希望挂起自己的某个子进程，以便考查和修改该子进程，或者协调各子进程间的活动。

(3)负荷调节的需要。当实时系统中的工作负荷较重，已可能影响到对实时任务的控制时，可由系统把一些不重要的进程挂起，以保证系统能正常运行。

(4)操作系统的需要。操作系统有时希望挂起某些进程，以便检查运行中的资源使用情况或进行记账。

2）进程状态的转换

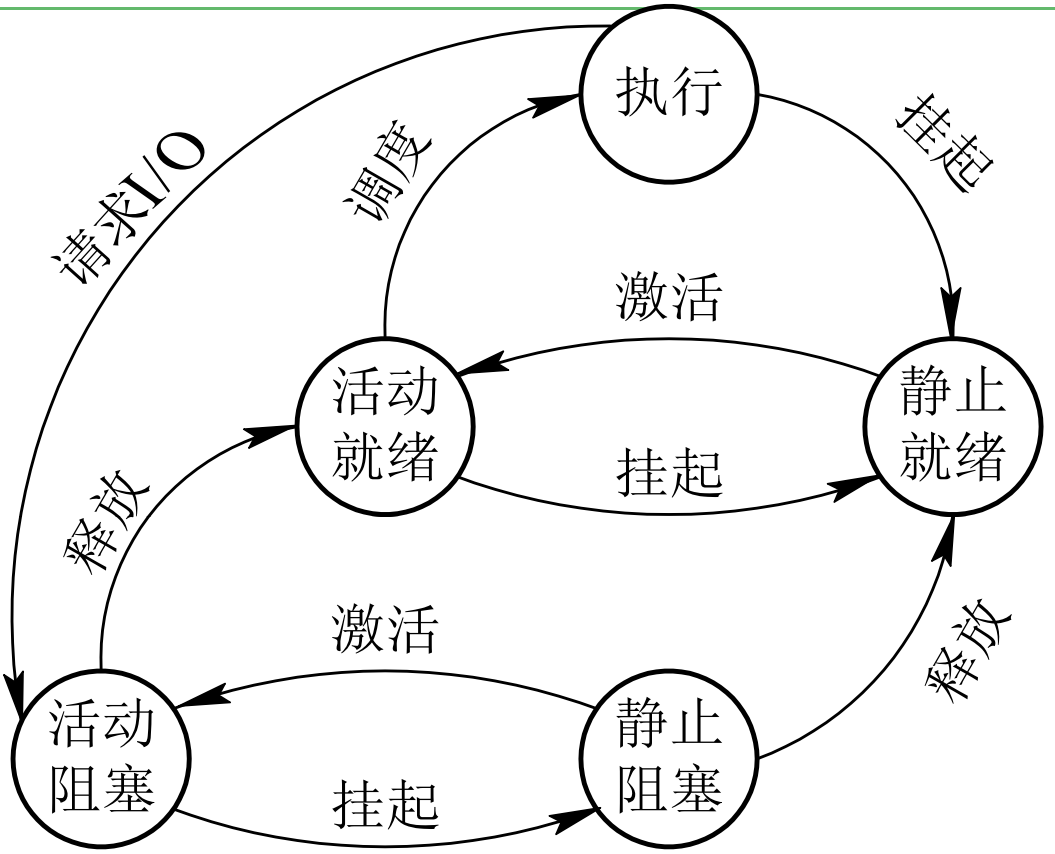
在引入挂起状态后，又将增加从挂起状态(又称为静止状态)到非挂起状态(又称为活动状态)的转换；或者相反。可有以下几种情况：

(1) 活动就绪→静止就绪。当进程处于未被挂起的就绪状态时，称此为活动就绪状态，表示为Readya。当用挂起原语Suspend将该进程挂起后，该进程便转变为静止就绪状态，表示为Readys，处于Readys状态的进程不再被调度执行。

(2) 活动阻塞→静止阻塞。当进程处于未被挂起的阻塞状态时，称它是处于活动阻塞状态，表示为Blockeda。当用Suspend原语将它挂起后，进程便转变为静止阻塞状态，表示为Blockeds。处于该状态的进程在其所期待的事件出现后，将从静止阻塞变为静止就绪。

(3) 静止就绪→活动就绪。处于Readys状态的进程，若用激活原语Active激活后，该进程将转变为Readya状态。

(4) 静止阻塞→活动阻塞。处于Blockeds状态的进程，若用激活原语Active激活后，该进程将转变为Blockeda状态。图2-6示出了具有挂起状态的进程状态图。




具有挂起状态的进程状态图

## 3．创建状态和终止状态

1）创建状态

创建一个进程一般要通过两个步骤：首先，为一个新进程创建PCB，并填写必要的管理信息；其次，把该进程转入就绪状态并插入就绪队列之中。当一个新进程被创建时，系统已为其分配了PCB，填写了进程标识等信息，但由于该进程所必需的资源或其它信息，如主存资源尚未分配等，一般而言，此时的进程已拥有了自己的PCB，但进程自身还未进入主存，即创建工作尚未完成，进程还不能被调度运行，其所处的状态就是创建状态。

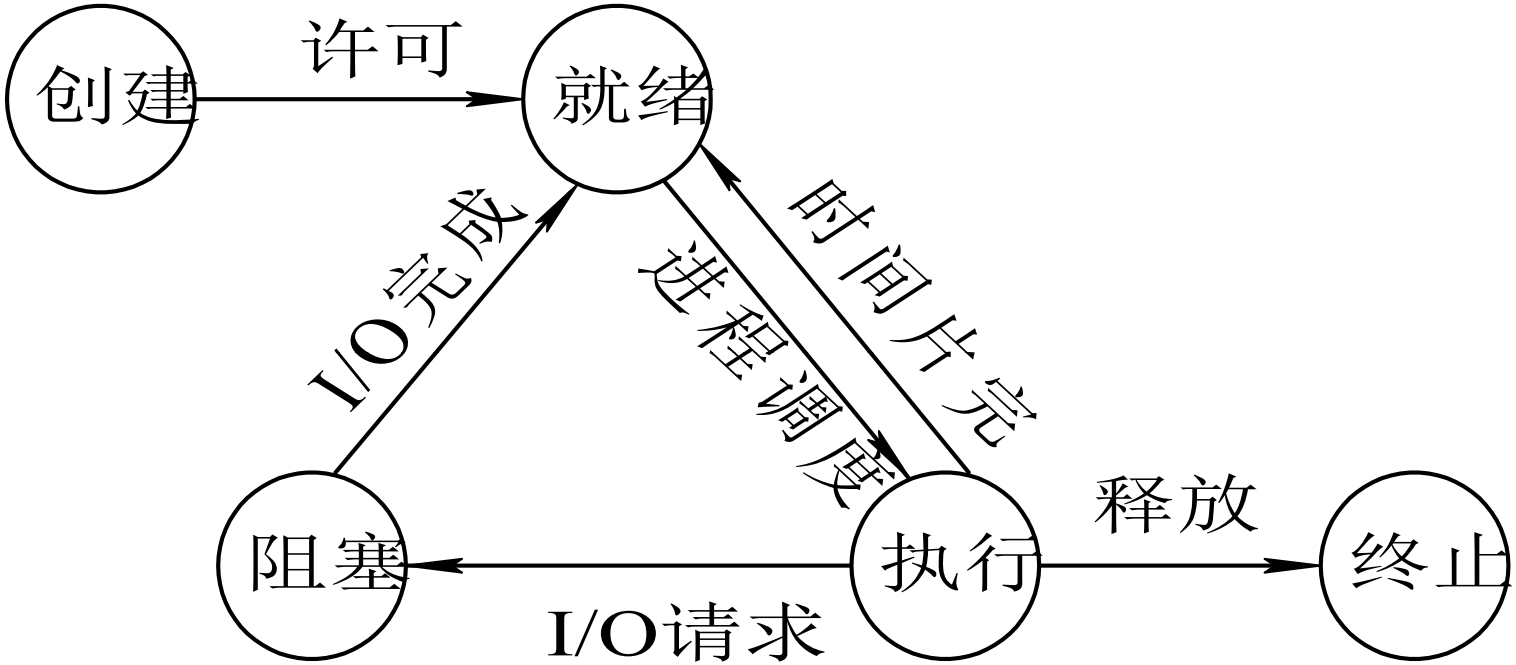
引入创建状态，是为了保证进程的调度必须在创建工作完成后进行，以确保对进程控制块操作的完整性。同时，创建状态的引入，也增加了管理的灵活性，操作系统可以根据系统性能或主存容量的限制，推迟创建状态进程的提交。

对于处于创建状态的进程，获得了其所必需的资源，以及对其PCB初始化工作完成后，进程状态便可由创建状态转入就绪状态。

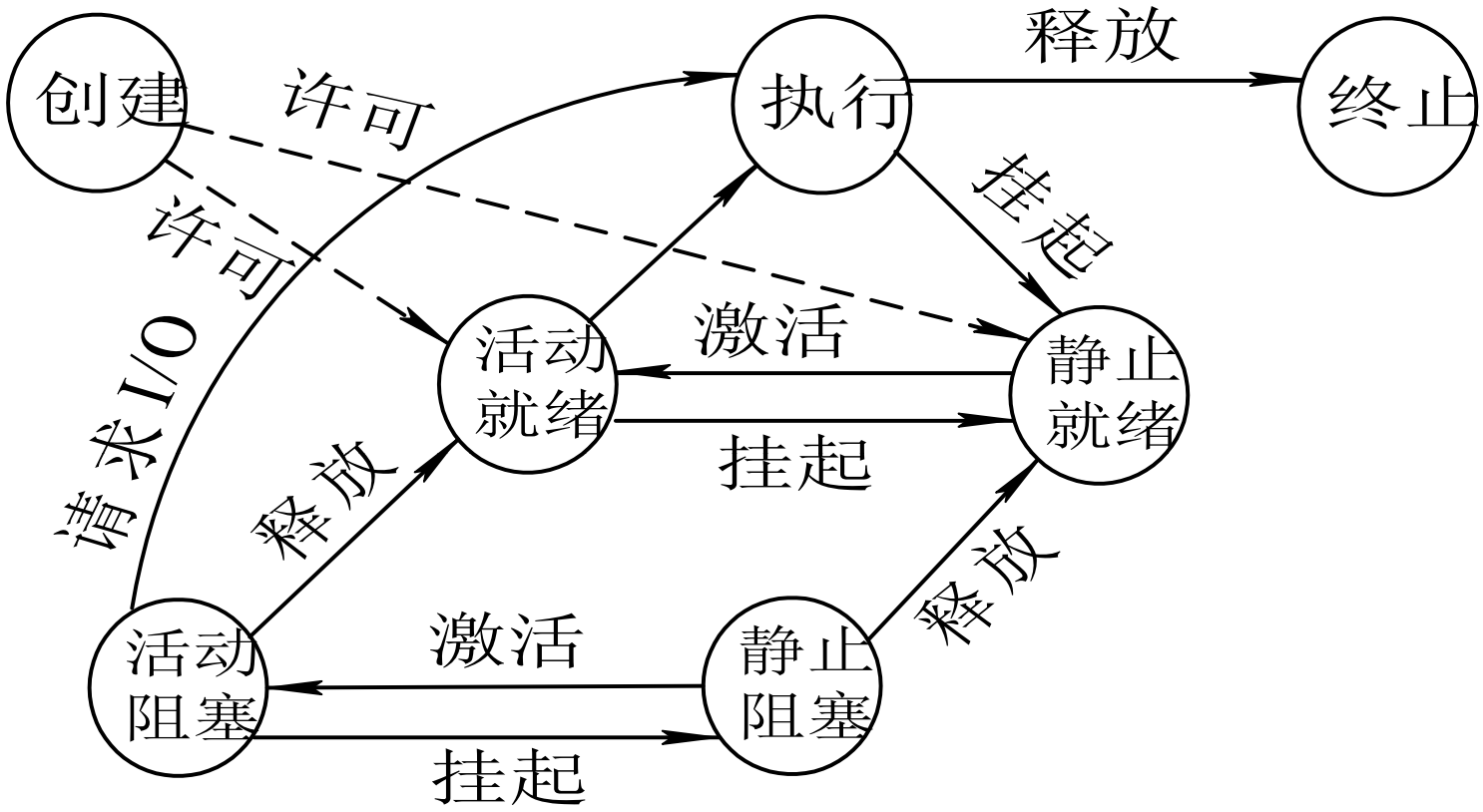
2）终止状态

进程的终止也要通过两个步骤：首先等待操作系统进行善后处理，然后将其PCB清零，并将PCB空间返还系统。当一个进程到达了自然结束点，或是出现了无法克服的错误，或是被操作系统所终结，或是被其他有终止权的进程所终结，它将进入终止状态。进入终止态的进程以后不能再执行，但在操作系统中依然保留一个记录，其中保存状态码和一些计时统计数据，供其它进程收集。一旦其它进程完成了对终止状态进程的信息提取之后，操作系统将删除该进程。

下图示出了增加了创建状态和终止状态后，进程的三种基本状态及转换图衍变为五种状态及转换关系图。




进程的五种基本状态及转换




具有创建、终止和挂起状态的进程状态图

引进创建和终止状态后，在进程状态转换时，相比较所示的进程五状态转换而言，需要增加考虑下面的几种情况。

（1） NULL→创建：一个新进程产生时，该进程处于创建状态。

(2) 创建→活动就绪：在当前系统的性能和内存的容量均允许的情况下，完成对进程创建的必要操作后，相应的系统进程将进程的状态转换为活动就绪状态。

(3) 创建→静止就绪：考虑到系统当前资源状况和性能要求，并不分配给新建进程所需资源，主要是主存资源，相应的系统进程将进程状态转为静止就绪状态，对换到外存，不再参与调度，此时进程创建工作尚未完成。

(4) 执行→终止：当一个进程到达了自然结束点，或是出现了无法克服的错误，或是被操作系统所终结，或是被其他有终止权的进程所终结，进程即进终止状态。

# 3.2.2　进程控制块

1．进程的组成：

## 2．进程控制块作用

进程控制块PCB(Process Control Block)，是操作系统中最重要的记录型数据结构。PCB中记录了操作系统所需的、用于描述进程的当前情况以及控制进程运行的全部信息。

OS是根据PCB来对并发执行的进程进行控制和管理的。

当OS要调度某进程执行时，要从该进程的PCB中查出其现行状态及优先级；

在调度到某进程后，要根据其PCB中所保存的处理机状态信息，设置该进程恢复运行的现场，并根据其PCB中的程序和数据的内存始址，找到其程序和数据；

进程在执行过程中，当需要和与之合作的进程实现同步、通信或访问文件时，也都需要访问PCB；

当进程由于某种原因而暂停执行时，又须将其断点的处理机环境保存在PCB中。

## 3．进程控制块中的信息

1）进程标识符

进程标识符用于惟一地标识一个进程。一个进程通常有两种标识符：

(1) 内部标识符。

(2) 外部标识符。

## 2）处理机状态

处理机状态信息主要是由处理机的各种寄存器中的内容组成的。

① 通用寄存器，又称为用户可视寄存器，它们是用户程序可以访问的，用于暂存信息；

② 指令计数器，其中存放了要访问的下一条指令的地址；

③ 程序状态字PSW，含有状态信息，如条件码、执行方式、中断屏蔽标志等；

④ 用户栈指针，指每个用户进程都有一个或若干个与之相关的系统栈，用于存放过程和系统调用参数及调用地址。

3）进程调度信息

① 进程状态，指明进程的当前状态，作为进程调度和对换时的依据；

② 进程优先级，用于描述进程使用处理机的优先级别的一个整数，优先级高的进程应优先获得处理机；

③ 进程调度所需的其它信息，它们与所采用的进程调度算法有关，比如，进程已等待CPU的时间总和、进程已执行的时间总和等；

④ 事件，指进程由执行状态转变为阻塞状态所等待发生的事件，即阻塞原因。

4）进程控制信息

① 程序和数据的地址，指进程的程序和数据所在的内存或外存地(首)址；

② 进程同步和通信机制，如消息队列指针、信号量等，它们可能全部或部分地放在PCB中；

③ 资源清单，除CPU以外的、进程所需的全部资源及已经分配到该进程的资源的清单；

④ 链接指针，它给出了本进程(PCB)所在队列中的下一个进程的PCB的首地址。

进程控制块集中反映了进程的动态特征。

例3-2：PCB的C语言描述

```
struct  pentry{
        int  pid;
        int  pprio;
        char  pstate;
        int pname;
        int  msg;
        int  paddr;
        int  pregs[SIZE];     /现场保护区大小
        …..
}pcb[];
```

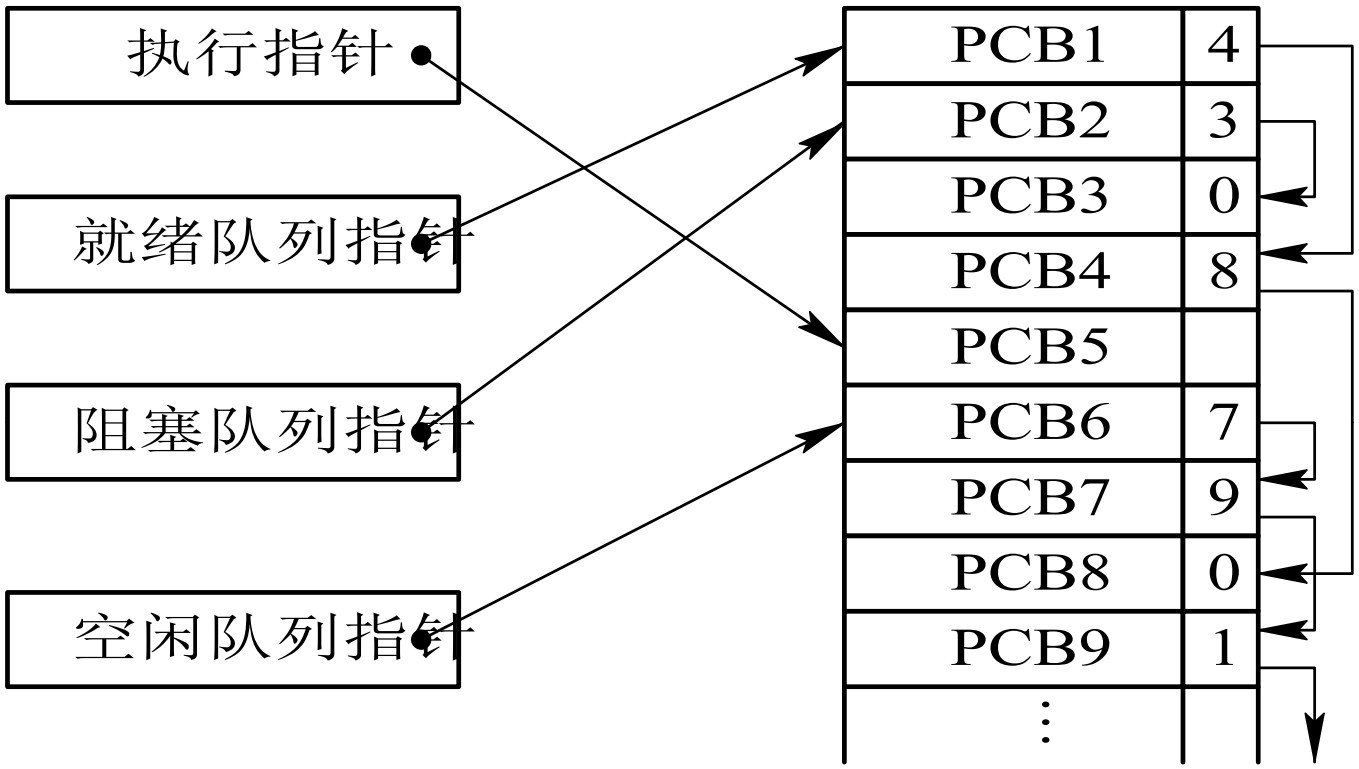
## 4. 进程控制块的组织方式

1）链接方式

这是把具有同一状态的PCB链接成一个队列。这样，可以形成就绪队列、若干个阻塞队列和空白队列等。

**就绪队列：**按进程优先级的高低排列，

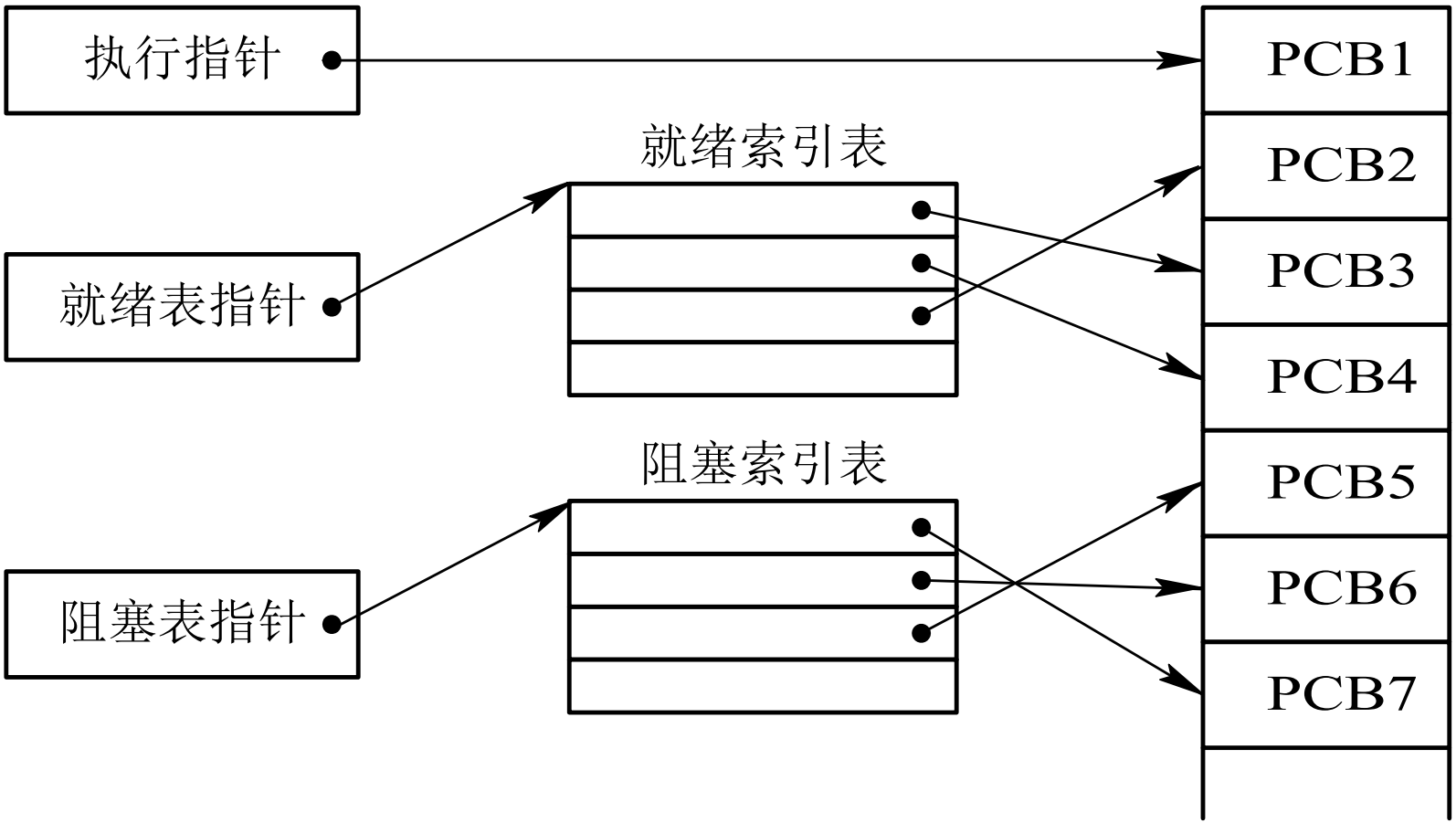
**阻塞队列：**也可根据阻塞原因的不同分别排队。如等待I/O操作完成的队列、等待分配内存的队列等。


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

PCB链接队列示意图

2）索引方式

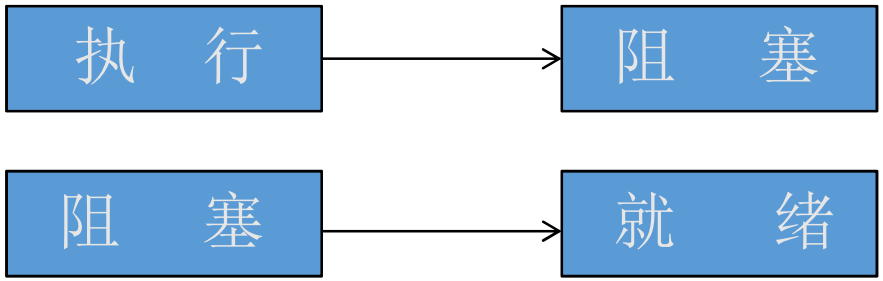
系统根据所有进程的状态建立几张索引表。例如，就绪索引表、阻塞索引表等，并把各索引表在内存的首地址记录在内存的一些专用单元中。在每个索引表的表目中，记录具有相应状态的某个PCB在PCB表中的地址。


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

按索引方式组织PCB

# 3.3 进程的控制

进程控制是进程管理中最基本的功能。它用于创建一个新进程，终止一个已完成的进程，或终止一个因出现某事件而使其无法运行下去的进程，还可负责进程运行中的状态转换。




进程控制一般是由OS的内核中的**原语**来实现的。

原语(Primitive)是由若干条机器指令组成的，用于完成一定功能的程序段。

它与一般过程的区别在于：它们是“原子操作(Action Operation)”。

一个操作中的所有动作要么全做，要么全不做。它是一个不可分割的基本单位。

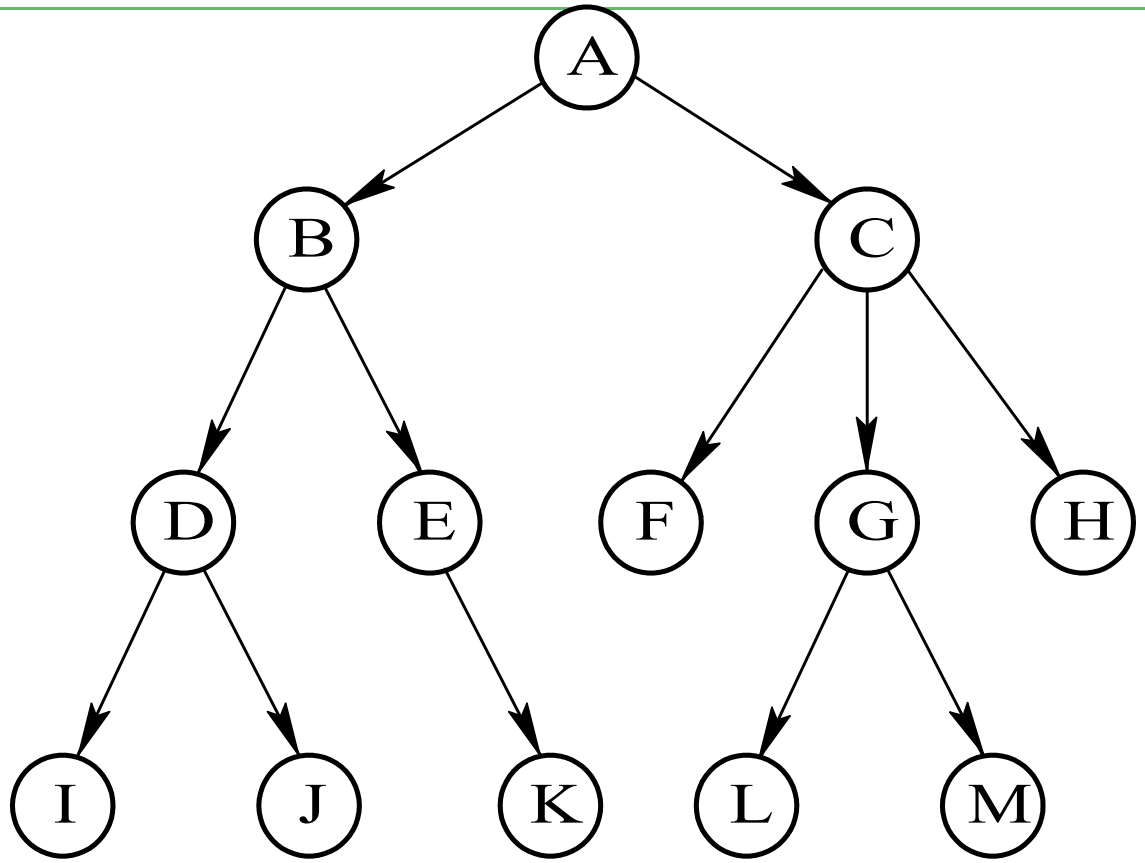
在执行过程中不允许被中断。

原子操作在管态下执行，常驻内存。

## 2.2.1 进程的创建

### 1. 进程图(Process Graph)

进程图是用于描述一个进程的家族关系的有向树，图中的结点(圆圈)代表进程。在进程D创建了进程I之后，称D是I的父进程(Parent Process)，I是D的子进程(Progeny Process)。




进程树

## 2. 引起创建进程的事件

在多道程序环境中，只有(作为)进程(时)才能在系统中运行。因此，为使程序能运行，就必须为它创建进程。导致一个进程去创建另一个进程的典型事件，可有以下四类：

(1)用户登录。在分时系统中，用户在终端键入登录命令后，如果是合法用户，系统将为该终端建立一个进程，并把它插入就绪队列中。

(2) 作业调度。在批处理系统中，当作业调度程序按一定的算法调度到某作业时，便将该作业装入内存，为它分配必要的资源，并立即为它创建进程，再插入就绪队列中。

（3）提供服务。当运行中的用户程序提出某种请求后，系统将专门创建一个进程来提供用户所需要的服务，例如，用户进程打印进程可并发执行

（4）应用请求。在上述三种情况下，都是由系统内核为它创建一个新进程；而第4类事件则由应用进程自己创建一个新进程，以便使新进程以并发运行方式完成特定任务。

例如，用于键盘终端输入数据，处理数据，表格形式在屏幕上显示的应用用进程为使这几个操作能并发执行，可以分别建立键盘输入进程、数据处理进程表格输出进程。

### 3．进程的创建(Creation of Process)

操作系统调用进程创建原语Creat( )按下述步骤创建一个新进程。

(1) 申请空白PCB。为新进程申请获得惟一的数字标识符，并从PCB集合中索取一个空白PCB。

(2) 为新进程分配资源。为新进程的程序和数据以及用户栈分配必要的内存空间。（批处理和交互型）

（3）初始化进程控制块。PCB的初始化包括：

① 初始化标识信息，将系统分配的标识符和父进程标识符填入新PCB中；

② 初始化处理机状态信息，使程序计数器指向程序的入口地址，使栈指针指向栈顶；

③ 初始化处理机控制信息，将进程的状态设置为就绪状态或静止就绪状态，对于优先级，通常是将它设置为最低优先级，除非用户以显式方式提出高优先级要求。

（4）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

在任何计算机系统中，都应有一个用于表示进程已经运行完成的指示。例如，在批处理系统中，通常在程序的最后安排一条Holt指令或终止的系统调用。在分时系统中，用户可利用Logs off去表示进程运行完毕，同样可产生一个中断，去通知OS进程已运行完毕。

## 2）异常结束

在进程运行期间，由于出现某些错误和故障而迫使进程终止(Termination of Process)。

(1) 越界错误。这是指程序所访问的存储区已越出该进程的区域。

(2) 保护错。这是指进程试图去访问一个不允许访问的资源或文件，或者以不适当的方式进行访问，例如，进程试图去写一个只读文件。

(3) 非法指令。这是指程序试图去执行一条不存在的指令。出现该错误的原因，可能是程序错误地转移到数据区，把数据当成了指令。

(4) 特权指令错。这是指用户进程试图去执行一条只允许OS执行的指令。

(5) 运行超时。这是指进程的执行时间超过了指定的最大值。

(6) 等待超时。这是指进程等待某事件的时间超过了规定的最大值。

(7) 算术运算错。这是指进程试图去执行一个被禁止的运算，例如被0除。

(8) I/O故障。这是指在I/O过程中发生了错误等。

3）外界干预

(1) 操作员或操作系统干预。由于某种原因，例如，发生了死锁，由操作员或操作系统终止该进程。

(2) 父进程请求。当父进程提出请求时，系统将终止该进程。

(3) 父进程终止。当父进程终止时，OS也将它的所有子孙进程终止。

## 2．进程的终止过程

OS调用进程终止原语去终止指定的进程。

（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，以防它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(PCB)从所在队列(或链表)中移出，等待其他程序来搜集信息。

## 2.2.3 进程的阻塞与唤醒

### 1. 引起进程阻塞和唤醒的事件

有下述几类事件会引起进程阻塞或被唤醒。

1）请求系统服务

如，一进程请求使用某资源，如打印机，由于系统已将打印机分配给其他进程而不能分配给请求进程，这时请求者进程只能被阻塞，仅在其他进程在释放出打印机的同时，才将请求进程唤醒。

2）启动某种操作

当进程启动某种操作后，如果该进程必须在该操作完成之后才能继续执行，则必须先使该进程阻塞，以等待该操作完成。例如，进程启动了某I/O设备，如果只有在I/O设备完成了指定的I/O操作任务后进程才能继续执行，则该进程在启动了I/O操作后，便自动进入阻塞状态去等待。在I/O操作完成后，再由中断处理程序或中断进程将该进程唤醒。

3）新数据尚未到达

例如，有两个进程，进程A用于输入数据，进程B对输入数据进行加工。假如A尚未将数据输入完毕，则进程B将因没有所需的处理数据而阻塞；一旦进程A把数据输入完毕，便可去唤醒进程B。

4）无新工作可做

系统往往设置一些具有某特定功能的系统进程，每当这种进程完成任务后，便把自己阻塞起来以等待新任务到来。例如，系统中的发送进程，其主要工作是发送数据，若已有的数据已全部发送完成而又无新的发送请求，这时(发送)进程将使自己进入阻塞状态； 仅当又有进程提出新的发送请求时，才将发送进程唤醒。

## 2．进程阻塞过程

正在执行的进程，当发现上述某事件时，由于无法继续执行，于是进程便通过调用阻塞原语block把自己阻塞。进程的阻塞是进程自身的一种主动行为。进入block过程后，应先立即停止执行，把进程控制块中的现行状态由“执行”改为“阻塞”，并将PCB插入阻塞队列。

转调度程序进行重新调度，将处理机分配给另一就绪进程并进行切换，亦即，保留被阻塞进程的处理机状态(在PCB中)，再按新进程的PCB中的处理机状态设置CPU的环境。

### 3．进程唤醒过程

当被阻塞进程所期待的事件出现时，如I/O完成或其所期待的数据已经到达，则由有关进程(比如用完并释放了该I/O设备的进程)调用唤醒原语wakeup( )，将等待该事件的进程唤醒。唤醒原语执行的过程是：首先把被阻塞的进程从等待该事件的阻塞队列中移出，将其PCB中的现行状态由阻塞改为就绪，然后再将该PCB插入到就绪队列中。

应当指出，block原语和wakeup原语是一对作用刚好相反的原语。因此，如果在某进程中调用了阻塞原语，则必须在与之相合作的另一进程中或其他相关的进程中安排唤醒原语，以能唤醒阻塞进程；否则，被阻塞进程将会因不能被唤醒而长久地处于阻塞状态，从而再无机会继续运行。

## 2.2.4 进程的挂起与激活

### 1．进程的挂起

当出现了引起进程挂起的事件时，比如，用户进程请求将自己挂起，或父进程请求将自己的某个子进程挂起，系统将利用挂起原语suspend( )将指定进程或处于阻塞状态的进程挂起。

挂起原语的执行过程是：

a）首先检查被挂起进程的状态，若处于活动就绪状态，便将其改为静止就绪； 对于活动阻塞状态的进程，则将之改为静止阻塞。

b）把该进程的PCB复制到某指定的内存区域。

c）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

当发生激活进程的事件时，例如，父进程或用户进程请求激活指定进程，若该进程驻留在外存而内存中已有足够的空间时，则可将在外存上处于静止就绪状态的该进程换入内存。这时，系统将利用激活原语active( )将指定进程激活。

激活原语先将进程从外存调入内存，检查该进程的现行状态，若是静止就绪，便将之改为活动就绪；若为静止阻塞，便将之改为活动阻塞。

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。

# 3.4 进程同步

## 3.4.1 互斥

### 1.临界资源

例3-3：假设在一飞机售票系统中，某一时刻数据库中关于某一航班的机票数量counter＝5。某一窗口的售票程序执行的一条操作语句是counter=counter-1；而另一窗口退票程序执行的一条操作语句是counter=counter+1。

用高级语言书写的语句counter=counter+1和counter=counter-1所对应的的汇编语言指令如下：

```
①LOAD   A, conuter ;          ①LOAD   B, conuter ;
②ADD    A, 1;                 ②SUB    B, 1;
③STORE   A,  counter ;        ③STORE   B,  counter ;
```

## 2．临界区

由前所述可知，不论是硬件临界资源，还是软件临界资源，多个进程必须互斥地对它进行访问。人们把在每个进程中访问临界资源的那段代码称为临界区(critical section)。

进程如何进入临界区？

# 互斥进入

在临界区前面增加一段用于进行检查的代码，把这段代码称为进入区(entry section)。相应地，在临界区后面也要加上一段称为退出区(exit section)的代码，用于将临界区正被访问的标志恢复为未被访问的标志。

进程中除上述进入区、临界区及退出区之外的其它部分的代码，在这里都称为剩余区。这样，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；

# 3.互斥

一组并发进程中的两个或多个程序段，因共享某一公有资源而使得这组并发进程不能同时进入临界区的关系称为进程的互斥。

## 4. 同步机制应遵循的规则

为实现进程互斥地进入自己的临界区，可用软件方法，更多的是在系统中设置专门的同步机构来协调各进程间的运行。所有同步机制都应遵循下述四条准则：

(1) 独立平等。各并发进程享有平等的、独立的竞争共享资源的权利。

(2) 空闲让进。当无进程处于临界区时，表明临界资源处于空闲状态，应允许一个请求进入临界区的进程立即进入自己的临界区，以有效地利用临界资源。

(3) 忙则等待。当已有进程进入临界区时，表明临界资源正在被访问，因而其它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。

(4) 有限等待。对要求访问临界资源的进程，应保证在有限时间内能进入自己的临界区，以免陷入“死等”状态。

(5) 让权等待。当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入“忙等”状态。

## 3.4.2 进程的同步

例3-4: 在控制测量系统中，数据采集任务反复把所采集的数据送入一个单缓冲区；计算任务不断从该单缓冲区中取出数据进行计算。

```
var  buf;
int  flag=0;
collection（    ）
{
  while (TRUE)
   {
    采集数据；
    while(flag＝＝1);
    将采集的数据放入buffer；
    flag=1
   }
}
```

```
calculate (  )
{
      while(TRUE)
       {
        while(flag＝＝0);
        从buf中取出数据；
        flag=0;
        计算处理；
       }
}
```

异步环境下的一组并发进程，在某些程序段上需互相合作、互相等待，使得各进程在某些程序段上必须按一定的顺序执行的制约关系称为进程的同步。

# 3.4.3 同步机构

系统中用来实现进程将同步与互斥的机构统称为同步机构。同步机构采用一个物理实体：锁、信号量等，并提供相应的原语。

1．加锁原语LOCK(ω) （ω=1表示没有别的进程进入临界区
ω=0表示资源正在被使用 ）

定义：(整型信号量，wait(s))
①测试ω是否为1；②若ω=1，则0→ω；
③若ω=0，则返回到①。
开锁原语UNLOCK(ω)
定义：(signal(s))
1→ω。

临界区的互斥控制：进入临界区的进程先要执行加锁原语，退出时要执行开锁原语。

描述：

```
pro{
        …
        LOCK(ω)
         <临界区>
        UNLOCK(key(ω))
        …
   }
```

## 2. 信号量和P、V原语

(1)信号量

①S是一个整型变量而且初值非负。

②对信号量仅能实施P(S)操作和V(S)操作,也只有这两种操作才能改变S的值。

③对每一个信号量，都对应有一个（空或空非）的等待队列，队列中的进程处于阻塞状态。

## （2）P(S)原语(wait(s))

①S减1；

②若S-1 ≥ 0，则进程继续执行；

③若S-1＜0，则该进程被阻塞后并进入该信号相对应的等待队列中，然后转进程调度。Block(s.1)

## （3）V原语(signal(s))

①S加1；

②若S+1 ＞0，进程继续执行；

③若S+1 ≤0，则从该信号的等待队列中唤醒一等待进程，然后再返回原进程继续执行或转进程调度。

Wakeup(s.1)

- P、V操作的执行决不允许中断。
- P、V操作必须成对出现。

P、V操作原语的实现（加锁法P62）

## (4)P、V操作的物理意义

信号量的初值用来表示系统中同类资源的可用数目。

当S＝0时，？

S<0时，？

每执行一次P操作意味着请求分配一个单位的某类资源，因此描述为S=S－1；

若S ＜ 0表示已无该类资源可供分配，因此把该进程排列到与该S相关的等待队列中。若进程使用完某类资源而必须执行一次V操作，意味者释放一个单位的该类资源，因此描述为S=S＋1；

若S ≤ 0表示已有进程在等待该类资源，因此唤醒等待队列中的第一个或优先数最高的进程，允许其使用该类资源。

## 3．AND型信号量

假定现有两个进程A和B，他们都要求访问共享数据D和E。当然，共享数据都应作为临界资源。为此，可为这两个数据分别设置用于互斥的信号量Dmutex和Emutex，并令它们的初值都是1。相应地，在两个进程中都要包含两个对Dmutex和Emutex的操作，即

process A:　　process B:

P(Dmutex)；　　V(Emutex)；

P(Emutex)；　　V(Dmutex)；

若进程A和B按下述次序交替执行wait操作：

process A: P(Dmutex)； 于是Dmutex=0

process B: P(Emutex)； 于是Emutex=0

process A: P(Emutex)； 于是Emutex=-1 A阻塞

process B: P(Dmutex)； 于是Dmutex=-1 B阻塞

最后，进程A和B处于僵持状态。在无外力作用下，两者都将无法从僵持状态中解脱出来。我们称此时的进程A和B已进入死锁状态。显然，当进程同时要求的共享资源愈多时，发生进程死锁的可能性也就愈大。

AND同步机制的基本思想是：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它。亦即，对若干个临界资源的分配，采取原子操作方式：要么把它所请求的资源全部分配到进程，要么一个也不分配。这样就可避免上述死锁情况的发生。为此，在wait操作中，增加了一个“AND”条件，故称为AND同步，或称为同时wait操作，即Swait(Simultaneous wait)定义如下:

Swait($S_1$，$S_2$，…，$S_n$)

if $S_i$>=1 and ... and $S_n$>=1 then

for i:=1 to n do

$S_i$:=$S_i$-1；

endfor

else

place the process in the waiting queue associated with the first $S_i$ found with $S_i$<1，and set the program count of this process to the beginning of Swait operation

endif

Ssignal($S_1$，$S_2$，…，$S_n$)

for i:=1 to n do

$S_i$:=Si+1；

Remove all the process waiting in the queue associated with Si into the ready queue.

endfor；

## 4．信号量集

在记录型信号量机制中，wait(S)或signal(S)操作仅能对信号量施以加1或减1操作，意味着每次只能获得或释放一个单位的临界资源。而当一次需要N个某类临界资源时，便要进行N次wait(S)操作，显然这是低效的。此外，在有些情况下，当资源数量低于某一下限值时，便不予以分配。因而，在每次分配之前，都必须测试该资源的数量，看其是否大于其下限值。基于上述两点，可以对AND信号量机制加以扩充，形成一般化的“信号量集”机制。

Swait操作可描述如下，其中S为信号量，d为需求值，而t为下限值。

Swait($S_1$，$t_1$，$d_1$，...，$S_n$，$t_n$，$d_n$)

if $S_i$>=$t_1$ and ... and $S_n$>=$t_n$ then

for i:=1 to n do

$S_i$:=$S_i$-$d_i$;

endfor

else

Place the executing process in the waiting queue of the first Si with $S_i$<$t_i$ and set its program counter to the beginning of the Swait Operation.

endif

Ssignal($S_1$，$d_1$，…，$S_n$，$d_n$)

for i:=1 to n do

$S_i$:=$S_i$+$d_i$;

Remove all the process waiting in the queue associated with Si into the ready queue

endfor;

下面我们讨论一般“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。

## 5. 管程机制

### 1）管程的定义

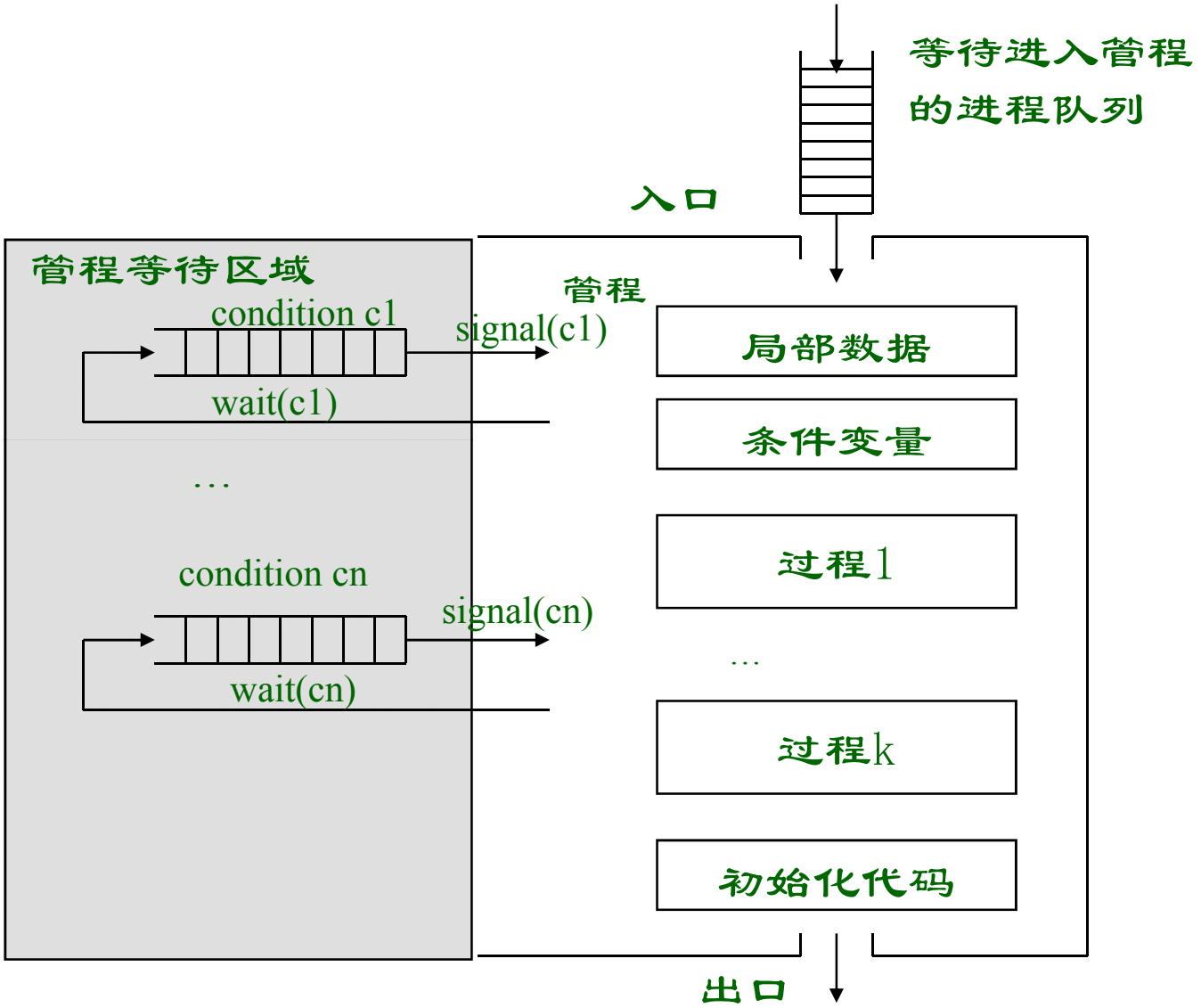
系统中的各种硬件资源和软件资源，均可用数据结构抽象地描述其资源特性，例如，对一台电传机，可用与分配该资源有关的状态信息(busy或free)和对它执行请求与释放的操作，以及等待该资源的进程队列来描述。又如，一个FIFO队列，可用其队长、队首和队尾以及在该队列上执行的一组操作来描述。

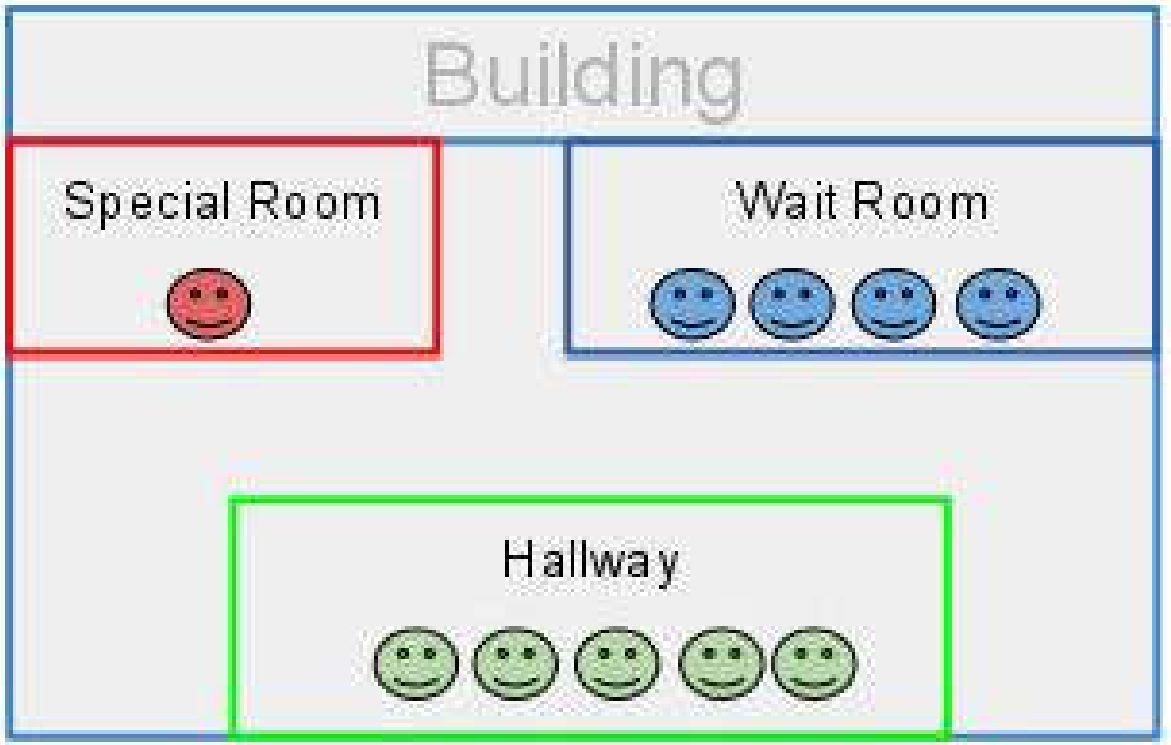
利用共享数据结构抽象地表示系统中的共享资源，而把对该共享数据结构实施的操作定义为一组过程，如资源的请求和释放过程request和release。

进程对共享资源的申请、释放和其它操作，都是通过这组过程对共享数据结构的操作来实现的，这组过程还可以根据资源的情况，或接受或阻塞进程的访问，确保每次仅有一个进程使用共享资源，这样就可以统一管理对共享资源的所有访问，实现进程互斥。

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，共同构成了一个操作系统的资源管理模块，我们称之为管程。**管程被请求和释放资源的进程所调用。Hansan为管程所下的定义是：“一个管程定义了一个数据结构和能为并发进程所执行(在该数据结构上)的一组操作，这组操作能同步进程和改变管程中的数据”。**

# 管程的结构


等待进入管程
的进程队列
入口
管程等待区域
condition c1
signal(c1)
wait(c1)
…
condition cn
signal(cn)
wait(cn)
管程
局部数据
条件变量
过程1
…
过程k
初始化代码
出口


Building
Special Room
Wait Room
Hallway

管程由四部分组成:

① 管程的名称;

② 局部于管程内部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 对局部于管程内部的共享数据设置初始值的语句。

管程的语法描述如下：

type monitor_name = MONITOR;

<共享变量说明>;

define <(能被其他模块引用的)过程名列表>;

use <(要调用的本模块外定义的)过程名列表>;

procedure <过程名>(<形式参数表>);

begin

end;

```
function <函数名>(<形式参数表>)：值类型；
    begin
      ⋮
    end；

begin
  <管程的局部数据初始化语句序列>；
end
```

局部于管程内部的数据结构，仅能被局部于管程内部的过程所访问，任何管程外的过程都不能访问它；反之，局部于管程内部的过程也仅能访问管程内的数据结构。

管程相当于围墙，它把共享变量和对它进行操作的若干过程围了起来，所有进程要访问临界资源时，都必须经过管程(相当于通过围墙的门)才能进入，而管程每次只准许一个进程进入管程，从而实现了进程互斥。

管程是一种程序设计语言结构成分，它和信号量有同等的表达能力，从语言的角度看，管程主要有以下特性：

(1) 模块化。管程是一个基本程序单位，可以单独编译。

(2) 抽象数据类型。管程中不仅有数据，而且有对数据的操作。

(3) 信息掩蔽。管程中的数据结构只能被管程中的过程访问，这些过程也是在管程内部定义的，供管程外的进程调用，而管程中的数据结构以及过程(函数)的具体实现外部不可见。

## 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。

但是仅仅有上述的同步工具是不够的。

如：当一个进程调用了管程，在管程中时被阻塞或挂起，直到阻塞或挂起的原因解除，而在此期间，如果该进程不释放管程，则其它进程无法进入管程，被迫长时间地等待。

为了解决这个问题，引入了条件变量condition。通常，一个进程被阻塞或挂起的条件(原因)可有多个，因此在管程中设置了多个条件变量，对这些条件变量的访问，只能在管程中进行。

管程中对每个条件变量都须予以说明，其形式为：Var x，y：condition。对条件变量的操作仅仅是wait和signal，条件变量也是一种抽象数据类型，每个条件变量保存了一个链表，用于记录因该条件变量而阻塞的所有进程，同时提供的两个操作即可表示为x.wait和x.signal，其含义为：

① 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。

② 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。如果存在多个这样的进程，则选择其中的一个，如果没有，则继续执行原进程，而不产生任何结果。

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。

# 3.4.4 同步机构应用

1．用信号量实现进程间互斥

例3-5：用信号量实现两个并发进程PA，PB互斥的描述如下：

```
semp  mutex=1;
PA ( )
{
    ⋮
P(mutex);
    临界区操作;
V(mutex);
    ⋮
}
```

```
PB ( )
{
    ⋮
P(mutex);
    临界区操作;
V(mutex);
    ⋮
}
```

## 2．用P，V操作实现同步

例3-6：用P、V原语实现例3-4中数据采集过程和计算过程的同步执行。

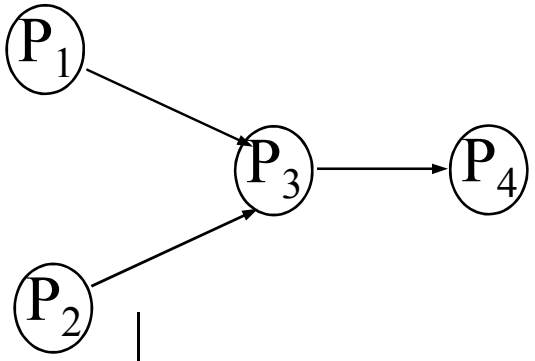
```
var  buf;
semp  Bufempty=1, //空的
buf
Buffull=0; //满的buf
 collection（  ）
 {
   while (TRUE)
   {
     采集数据;
     P(Bufempty);
    将采集的数据放入buf;
      V(Buffull);
   }
}
```

```
calculate（  ）
{
  while(TRUE)
   {
     P(Buffull);
     从buf中取出数据;
     V(Bufempty);
     计算处理;
   }
}
```

## 3. 利用信号量实现前趋关系

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。图中的每个结点可用于描述一个程序段或进程，乃至一条语句；结点间的有向边则用于表示两个结点之间存在的偏序(Partial Order，亦称偏序关系)或前趋关系(Precedence Relation)“→”。 →={($P_i$，$P_j$)|$P_i$ 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。

## 例3-7：前趋图




int a1,a2,a3=0;//分别表示pi进程是否完成

```
P1：a=x+2;
P2：b=y+4;
P3：c=a+b;
P4：d=c+b;
```

```
P1(  )
{
   a=x+2
;
   V(a1);
}
```

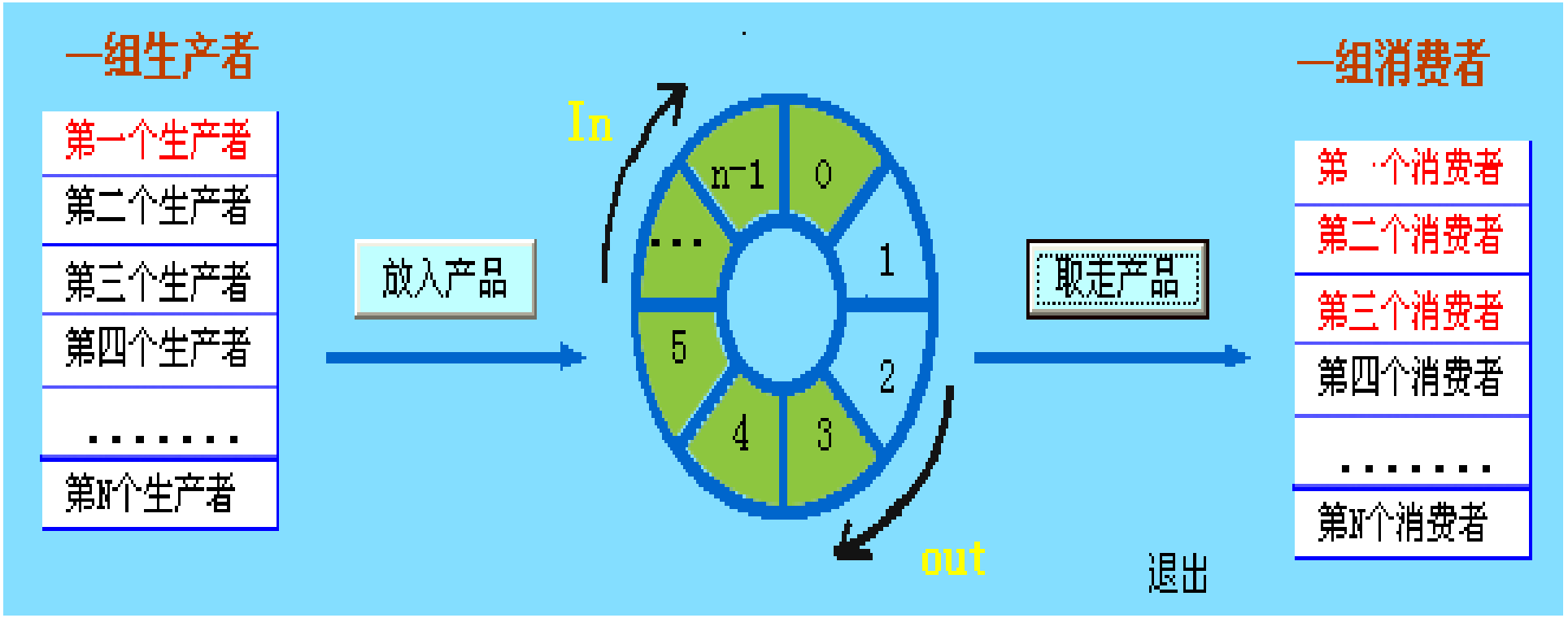
```
P2(  )
{
   b=y+4;
      V(a2);
}
```

```
P3(  )
{
   P(a1);
   P(a2);
   c=a+b;
   V(a3);
}
```

```
P4(  )
{
   P(a3);
   d=c+b;
}
```

# 3.5 经典进程的同步问题

## 3.5.1 生产者–消费者问题

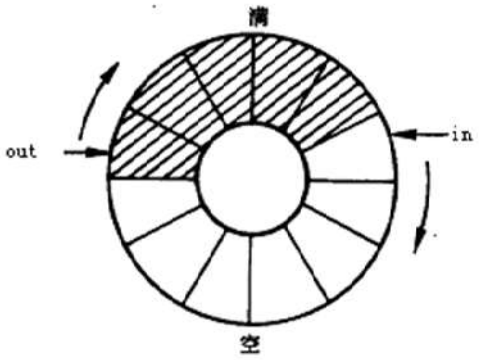


## 1．利用信号量解决生产者–消费者问题

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。又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。对生产者—消费者问题可描述如下:

```
int  in=0,  out=0; var  Buffer[n];
semp full=0,empty=n，mutex=1;
//空、满缓冲数 ，缓冲互斥

producer(i){
  var nextp;
  while (TRUE)
    {
    生成新的产品放入nextp;
    P (empty);
    P (mutex);
    Buffer [in]=nextp;
    in = (in+1) % n;
    V (mutex);
    V (full);
    }
}
```




```
consumer (i)
    var nextc;
    while (TRUE)
      {
       P (full);
       P (mutex);
       nextc=Buffer [out];
       out= (out+1) % n;
        V (mutex);
        V (empty);
        处理产品 nextc;
        }
 }
```

在生产者—消费者问题中应注意：

（1）无论是互斥信号量还是资源信号量都要成对出现；

（2）在每个程序中的多个P操作应先执行对资源信号量的P操作，然后再执行对互斥信号量的P操作，否则可能引起进程死锁。

## 2．利用管程解决生产者–消费者问题

在利用管程方法来解决生产者–消费者问题时，首先便是为它们建立一个管程，并命名为ProducerConsumer，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。

```
Monitor   ProducerConsumer   {
    int：count，n，out；// 数据结构定义
   buf：array [0..n-1] of item_type；
   condition：full，empty；
procedure put(item)   //过程
{      if  count>=n  then empty.wait;  //n个满缓冲，等待有空的缓冲
       buf[in]=item;
       in:=(in+1) mod n;
       count++;
       if full.queue then full.signal;  //有等待满缓冲的进程，则唤醒
}
procedure get(item)   //
 {      if  count<=0  then full.wait; //没有数据，等待有满的缓冲
       item=buf[out];
       out:=(out+1) mod n;
       count--;
      if empty.quence then  empty.signal; //有等待空缓冲的进程
  }
```

# 利用管程实现生产者-消费者问题

```
    { count=0 ; int=0;   out=0 } // 初始值
}
producer（）
   {
     while (true)
     {      produce(item);
            ProducerConsumer .put(item);
        }
   }
  consumer（）
    {
     while (true)
     {       ProducerConsumer .get(item);
             consume(item);
      }
    }
```

通过临界区互斥的自动化，管程比信号量更保证并发编程的正确性。但编译器必须识别管程并使用某种方法对其互斥。

## 3.5.2 读者–写者问题

读者—写者问题（**Readers-Writers problem**）也是一个经典的并发程序设计问题，是经常出现的一种同步问题。计算机系统中的数据（文件、记录）常被多个进程共享，但其中某些进程可能只要求读数据（称为读者Reader）；另一些进程则要求修改数据（称为写者Writer）。就共享数据而言，Reader和Writer是两组并发进程共享一组数据区，要求：

（1）允许多个读者同时执行读操作；

（2）不允许读者、写者同时操作；

（3）不允许多个写者同时操作。

## 1.利用信号量解决读者–写者问题

互斥信号量Wmutex：实现Reader与Writer进程间在读或写时的互斥。

整型变量Readcount：表示正在读的进程数目。由于只要有一个Reader进程在读，便不允许Writer进程去写。因此，仅当Readcount=0，Reader进程才需要执行Wait(Wmutex)操作。若Wait(Wmutex)操作成功，做Readcount+1操作。同理，仅当Reader进程在执行了Readcount减1操作后其值为0时，才须执行signal(Wmutex)操作，以便让Writer进程写。

互斥信号量rmutex：readcount为可以被多个读者访问的临界资源，读者需要互斥。

```
int  Readcount=0;
semp  Wmutex=1; //读者和写者互斥量
Rmutex=1;//Rreadcount互斥
 Reader(i){
  while (TRUE)
   {
    P (Rmutex);
    If (Readcount==0) P (Wmutex);
    Readcount++;}
    V (Rmutex);
    读数据库;
    P (Rmutex);
    Readcount--;
    If (Readcount==0) V (Wmutex);
    V (Rmutex);
     }
}
```

```
Writer (i){
   while (TRUE)
    {
     P (Wmutex);
     写数据库;
     V (Wmutex);
    }
}
```

## 2．利用信号量集机制解决读者–写者问题

这里的读者—写者问题与前面的略有不同，它增加了一个限制，即最多只允许RN个读者同时读。为此，又引入了一个信号量L，并赋予其初值为RN，通过执行wait(L，1，1)操作，来控制读者的数目。每当有一个读者进入时，就要先执行wait(L，1，1)操作，使L的值减1。当有RN个读者进入读后，L便减为0，第RN+1个读者要进入读时，必然会因wait(L，1，1)操作失败而阻塞。对利用信号量集来解决读者—写者问题的描述如下：

```
int  Readcount=0;
semp  Wmutex=1; //读者和写者互斥量
Rmutex=1;//Rreadcount互斥
 Reader(i){
  while (TRUE)
    {
    Swait(L,1,1); 控制读者数目
    Swait(mx,1,0); 无writer进程写
    perform read operation;
    Ssignal(L,1);
      }
}                          }
```

其中，Swait(mx，1，0)语句起着开关的作用。只要无writer进程进入写，mx=1，reader进程就都可以进入读。但只要一旦有writer进程进入写时，其mx=0，则任何reader进程就都无法进入读。

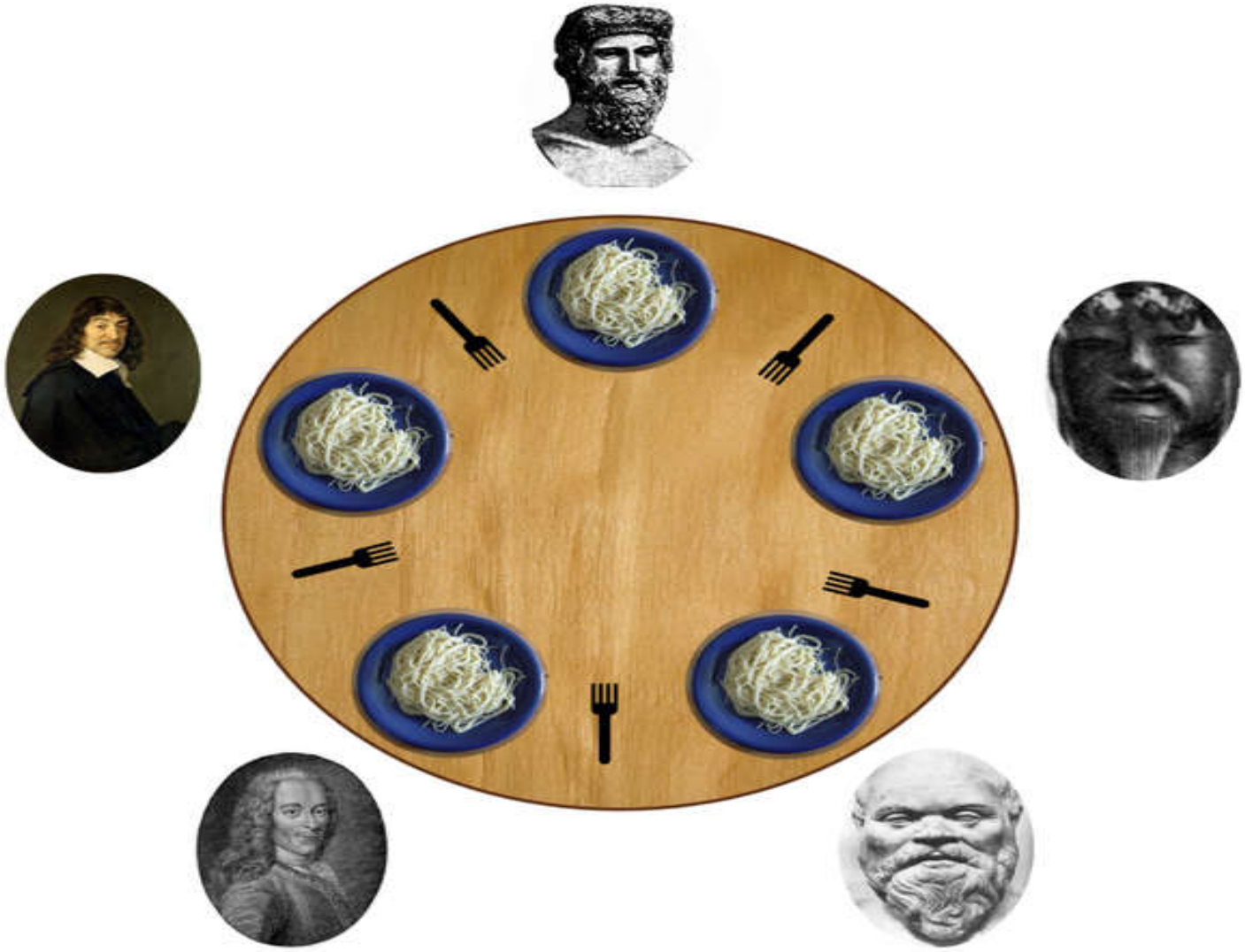
```
 Writer (i){
    while (TRUE)
      {
Swait(mx,1,1; L,RN,0);
perform write operation;
Ssignal(mx,1);
      }
 }
```

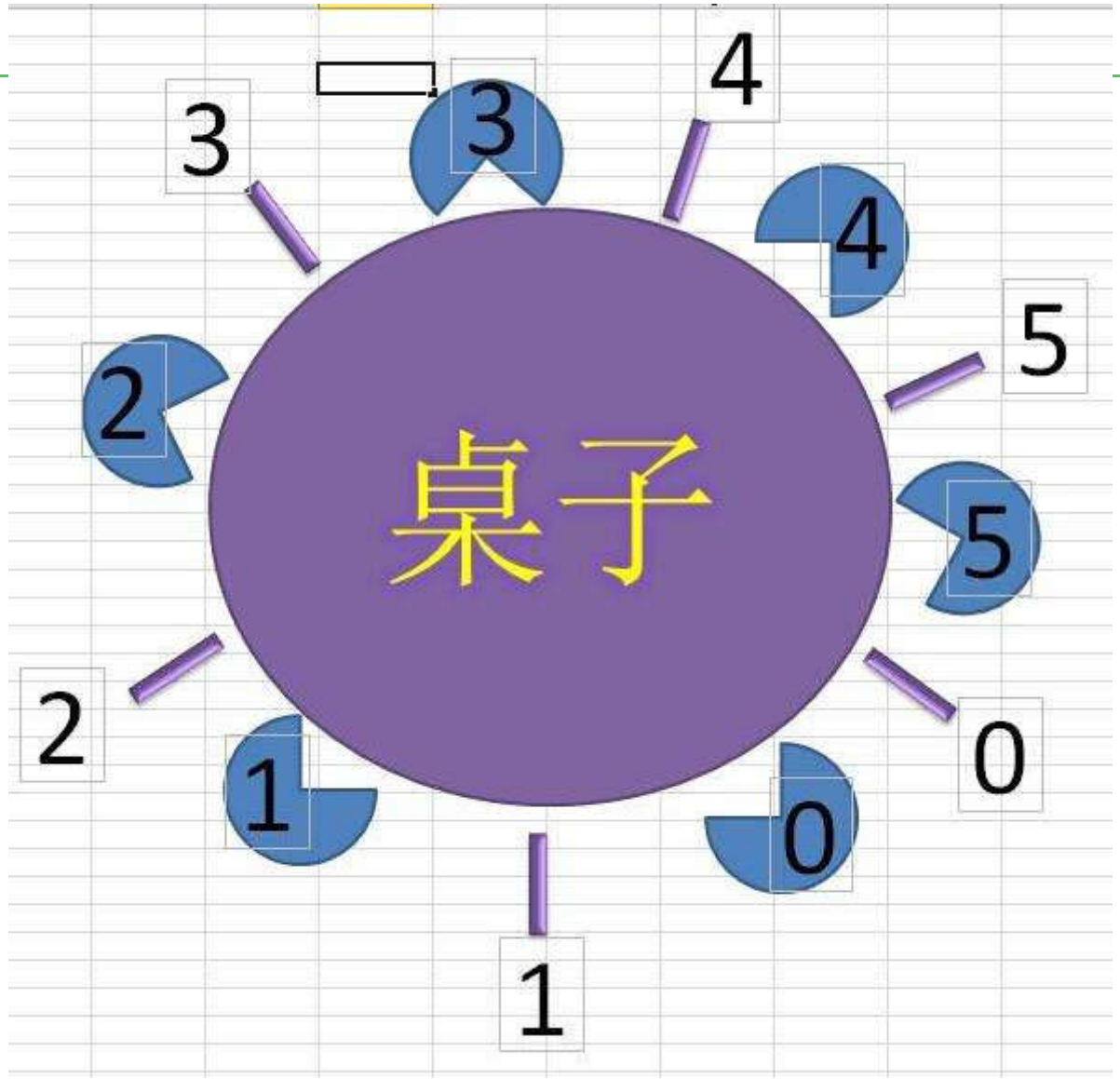
Swait(mx，1，1；L，RN，0)语句表示仅当既无writer进程在写(mx=1)，又无reader进程在读(L=RN)时，writer进程才能进入临界区写。

### 3.5.3 哲学家进餐问题

由荷兰学者Dijkstra提出的哲学家进餐问题(The Dinning Philosophers Problem)是经典的同步问题之一。哲学家进餐问题是一大类并发控制问题的典型例子，涉及信号量机制、管程机制以及死锁等操作系统中关键问题的应用，在操作系统文化史上具有非常重要的地位。

对该问题的剖析有助于深刻地理解计算机系统中的资源共享、进程同步机制、死锁等问题，并能熟练地将该问题的解决思想应用于生活中的控制流程。




桌子
3
4
3
4
5
2
5
2
0
1
0
1

```
    semp  chopstick [5]={1, 1, 1, 1, 1};
//筷子的互斥
    thinker (i){
        P (chopstick[i]);
        P (chopstick[(i+1)mod5]);
         进餐;
        V (chopstick[i]);
        V (chopstick[(i+1)mod5]);
         思考;
    }
```

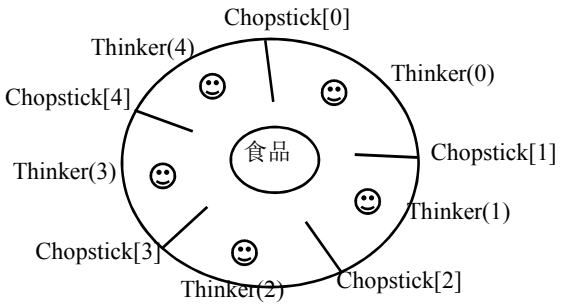



图3－8　哲学家进餐问题

在以上描述中，当哲学家饥饿时，总是先去拿他右边的筷子，即执行wait(chopstick[i])； 成功后，再去拿他左边的筷子，即执行wait(chopstick[(i+1)mod 5])；又成功后便可进餐。进餐完毕，又先放下他右边的筷子，然后再放左边的筷子。虽然，上述解法可保证不会有两个相邻的哲学家同时进餐，但有可能引起死锁。假如五位哲学家同时饥饿而各自拿起右边的筷子时，就会使五个信号量chopstick均为0； 当他们再试图去拿左边的筷子时，都将因无筷子可拿而无限期地等待。对于这样的死锁问题，可采取以下几种解决方法：

(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，并在用毕时能释放出他用过的两只筷子，从而使更多的哲学家能够进餐。

(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。按此规定，将是1、2号哲学家竞争1号筷子；3、4号哲学家竞争3号筷子。即五位哲学家都先竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一位哲学家能获得两只筷子而进餐。

# 生活概念

- 一个十字路口就是一个multi-task系统，下面这些概念都有了： 进程，临界区，信号量，并发，互斥，中断，事件，时间片，阻塞，优先级,等等。 这和一个操作系统的调度作业是惊人的相似。
- 发明红绿灯的人是天才，发明系统调度方法的人也是天才，只不过他们的思维方式、逻辑、处理事情的方法是一样的。
- 程序是无限的，新技术新概念是无限的，但算法是有限的，思维方式是有限的，所谓万变不离其宗。把握住有限的思维方法和逻辑，以应对千变万化的技术，在我看来，这才是上智 。

# 3.6 进程通信

## 3.6.1 进程通信的类型

进程通信，指进程之间的信息交换。分高级通信和低级通信。本节介绍高级通信，是指用户可以直接利用操作系统所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。

# 3.6.2 进程通信的方式

## 1. 共享存储器系统

(1) 基于共享数据结构的通信方式。在这种通信方式中，要求诸进程公用某些数据结构，借以实现诸进程间的信息交换。如生产者—消费者问题？

缺点：？ 程序员工作量太大

这种通信方式是低效的，只适于传递相对少量的数据。

(2) 基于共享存储区的通信方式。为了传输大量数据，在存储器中划出了一块共享存储区，诸进程可通过对共享存储区中数据的读或写来实现通信。这种通信方式属于高级通信。

## 2．管道通信

所谓“管道”，是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件。向管道(共享文件)提供输入的发送进程(即写进程)，以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于UNIX系统，由于它能有效地传送大量数据，因而又被引入到许多其它的操作系统中。

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：

(1) 互斥，即当一个进程正在对pipe执行读/写操作时，其它(另一)进程必须等待。

(2) 同步，指当写(输入)进程把一定数量(如4 KB)的数据写入pipe，便去睡眠等待，直到读(输出)进程取走数据后，再把它唤醒。当读进程读一空pipe时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。

(3) 确定对方是否存在，只有确定了对方已存在时，才能进行通信。

```
#include <stdio.h>
main()
{    int x,fd[2];   char buf[50],s[50];
     pipe(fd); /*创建管道*/
     while((x=fork())==－1) /*创建子进程失败，循环*/
  {
   if (x==0) /*执行子进程*/
     {
      sprintf(buf,“This is an example of pipe\n”);
      write(fd[1],buf,50); /*把buf中的字符写入管道*/
      exit(0);
     }
  else /*父进程返回*/
   {
        wait（0）;
        read（fd[0],s,50）; /*父进程读管道中的字符*/
        printf(“%s”,s);
   }
}
```

## 3．消息传递系统

消息传递系统(Message passing system)是当前应用最为广泛的一种进程间的通信机制。

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。

程序员直接利用操作系统提供的一组通信命令(原语)，不仅能实现大量数据的传递，而且还隐藏了通信的实现细节，使通信过程对用户是透明的，从而大大减化了通信程序编制的复杂性，因而获得了广泛的应用。

在当今最为流行的微内核操作系统中，微内核与服务器之间的通信，无一例外地都采用了消息传递机制。又由于它能很好地支持多处理机系统、分布式系统和计算机网络，因此它也成为这些领域最主要的通信工具。消息传递系统的通信方式属于高级通信方式。又因其实现方式的不同而进一步分成直接通信方式和间接通信方式两种。

## 3.6.3 消息缓冲队列通信机制

由美国的Hansan提出，并在RC4000系统上实现。操作系统将一组数据称为一个消息，并在系统中设立一个大的缓冲区，作为消息缓冲池，缓冲池分为一个一个的消息缓冲区，每个缓冲区中存放一个消息。

消息通信采用一对系统调用Send（发送）过程和Receive（接收）过程来实现。

发送进程发送时先在自己的内存空间设置一个发送区把欲发送的消息填入其中，然后用发送过程将其发送出去。接收进程在接收消息之前，在自己的内存空间设置相应的接收区，然后用接收过程接收消息。

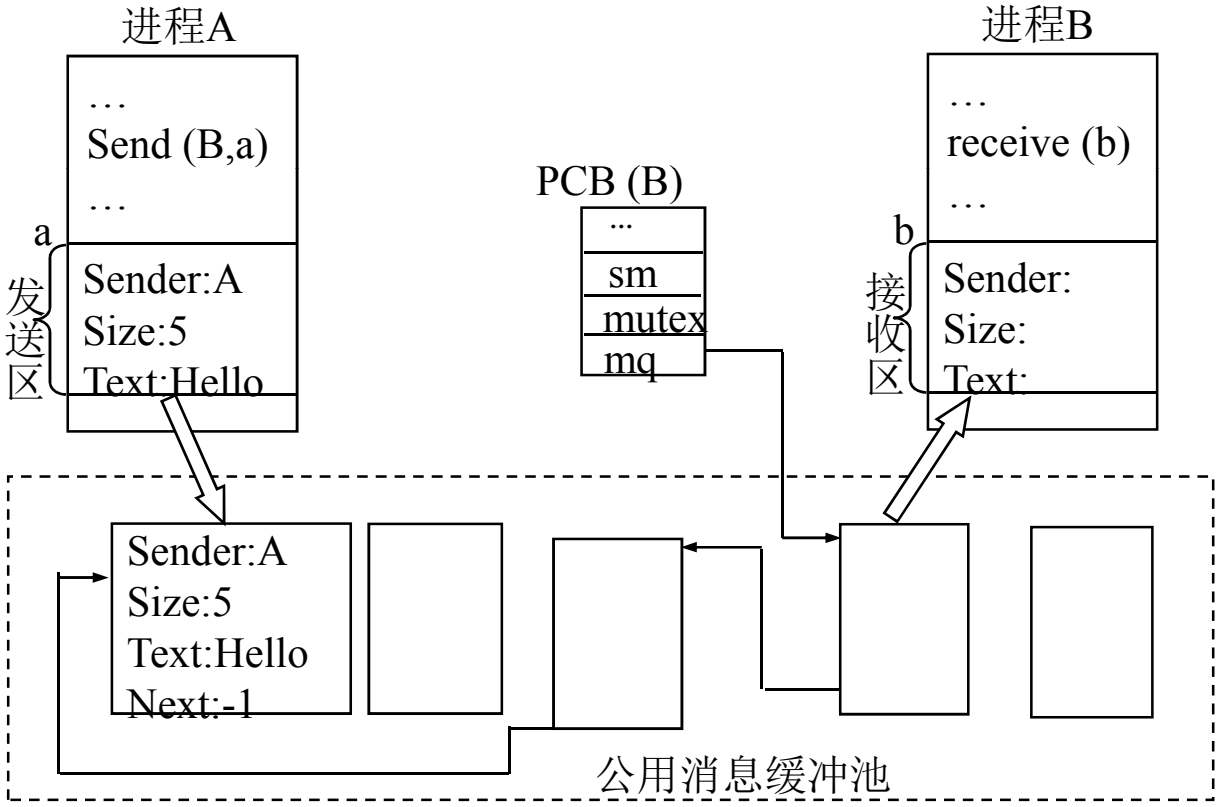
发送过程要向系统申请缓冲区放入自己的消息并通知接收过程，接收过程从缓冲区取走消息，同时释放缓冲区交回系统。

（1）消息缓冲区。在消息缓冲队列通信方式中，主要利用的数据结构是消息缓冲区，它是一个记录结构，主要包含下列内容：

```
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
```

## (2) PCB中有关通信的数据项:

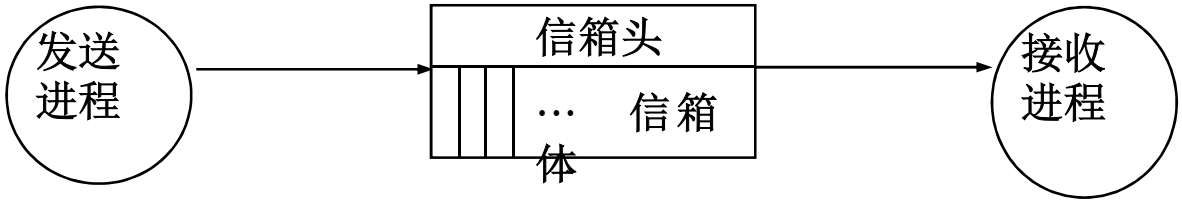
```
mq ;          //消息队列队首指针
mutex ;       //消息队列互斥信号量，初值为1
sm ;          //消息队列资源信号量，
```



```
send(receiver, a)
{getbuf(a.size,i); 根据a.size申请缓冲区;
i.sender:= a.sender;  将发送区a中的信息复制到消息缓冲区i中;
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;
P(j.mutex);
insert(j.mq, i); 将消息缓冲区插入消息队列;
signal(j.mutex);
signal(j.sm);
}
```

```
receive(b){
  j=getid( );
  P(j.sm) ;
  p(j.mutex);
  remove(j.mq, i);将消息队列第一个消息移出;
  V(j.mutex);
  b.sender=i.sender ;
  b.size=i.size;
  b.text =i.text;
  releasebuf(i);释放消息缓冲区
}
```

# 3.6.4 邮箱通信




信箱是大小固定的私有数据结构，它不像缓冲区那样被系统内所有进程共享。

**信箱头：名称，大小，方向，拥有该信箱的进程名**

**信箱体：消息**

A进程希望与B进程通信时创建一个连接两个进程的信箱。A调用发送过程deposit（m）发送消息进邮箱，B调用接受过程remove（m）将消息取出。

# 3.7 线程

## 3.7.1 线程的引入

引入进程的目的是什么？

为什么要引入线程？

为了说明这一点，我们首先来回顾进程的两个基本属性:① 进程是一个可拥有资源的独立单位；② 进程同时又是一个可独立调度和分派的基本单位。

正是由于进程这两个基本属性，构成了进程并发执行的基础。然而，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

系统在创建一个进程时，必须为它分配其所必需的、除处理机以外的所有资源，如内存空间、I/O设备，以及建立相应的PCB。

2) 撤消进程

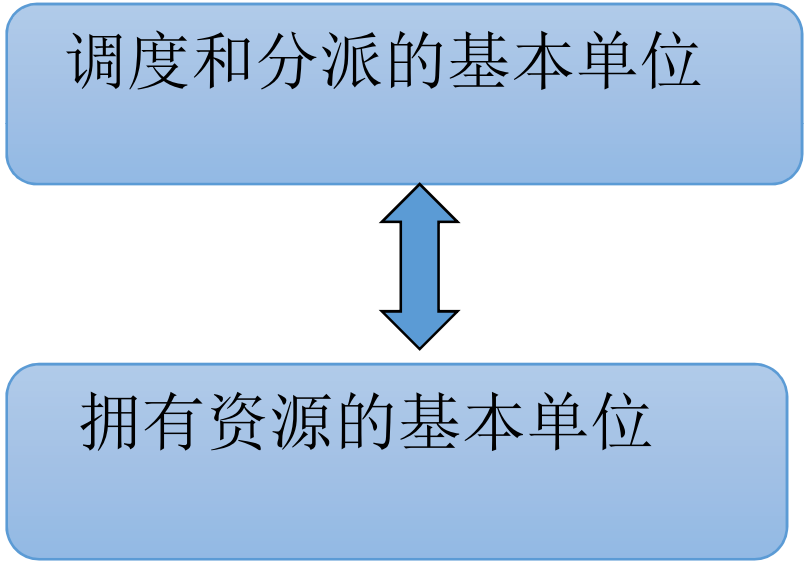
系统在撤消进程时，又必须先对其所占有的资源执行回收操作，然后再撤消PCB。

### 3) 进程切换

对进程进行切换时，由于要保留当前进程的**CPU**环境和设置新选中进程的**CPU**环境，因而须花费不少的处理机时间。

换言之，由于进程是一个资源的拥有者，因而在创建、撤消和切换中，系统必须为之付出较大的时空开销。正因如此，在系统中所设置的进程，其数目不宜过多，进程切换的频率也不宜过高，这也就限制了并发程度的进一步提高。

近年来设计操作系统时所追求的重要目标----如何能使多个程序更好地并发执行同时又尽量减少系统的开销。



随着对称多处理机（symmetric multiprocess，SMP）计算机系统的出现，利用传统的进程概念和设计方法已经难以设计出适合于SMP结构计算机系统的OS。

因为进程“太重”，需要花费较大的时空开销。如果在OS中引入线程，以线程作为调度和分派的基本单位，则可以有效地改善多处理机系统的性能。

引入线程，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。

# 3.7.2. 线程的概念

## 1.概念

一个进程内的基本调度单位称为线程或轻权进程。

## 2.线程与进程的比较

1) 调度

在传统的操作系统中，作为拥有资源的基本单位和独立调度、分派的基本单位都是进程。而在引入线程的操作系统中，则把线程作为调度和分派的基本单位，而进程作为资源拥有的基本单位。

## 2) 并发性

在引入线程的操作系统中，

进程之间可以并发执行。

一个进程中的多个线程之间可并发执行。

不同进程中的线程也能并发执行。

文字处理器（三个线程：一个显示文字和图形

一个从键盘读入数据

一个在后台进行拼写和语法检查

）

网页浏览器（两个线程：一个显示图像或文本

一个从网络中接收数据）

此外，有的应用程序需要执行多个相似的任务。例如，一个网络服务器经常会接到许多客户的请求，如果仍采用传统的单线程的进程来执行任务，则每次只能为一个客户服务。但如果在一个进程中可以设置多个线程，将其中的一个专用于监听客户的请求，则每当有一个客户请求时，便立即创建一个线程来处理该客户的请求。

3) 拥有资源

所有的操作系统中，进程都可以拥有资源，是系统中拥有资源的一个基本单位。

引入线程的操作系统中，线程自己不拥有系统资源(也有一点必不可少的资源，如TCB，程序计数器，保留局部变量，少数状态参数和返回地址等的一组寄存器和堆栈)，但它可以访问其隶属进程的资源，即一个进程的代码段、数据段及所拥有的系统资源（打开的文件、I/O设备）等，可以供该进程中的所有线程所共享。

4) 独立性

在同一进程中的不同线程之间的独立性要比不同进程之间的独立性低的多。

每个进程都拥有一个独立的地址空间和其他资源，除了共享全局变量外，不允许其他进程的访问。

而线程常常共享进程的内存地址空间和资源。如每个线程都可以访问它们所属进程地址空间中的所有地址，比如一个线程的堆栈可以被其它线程程读、写甚至完全清除。由一个线程打开的文件可供其它线程读、写。

## 5) 系统开销

在进程切换时，涉及到当前进程CPU环境的保存及新被调度运行进程的CPU环境的设置，

线程的切换时，则仅需保存和设置少量寄存器内容，不涉及存储器管理方面的操作，所以就切换代价而言，进程也是远高于线程的。

例如，在Solaris 2 OS中，线程的创建要比进程的创建快30倍，上下文切换要比进程快5倍。

此外，由于一个进程中的多个线程具有相同的地址空间，在同步和通信的实现方面线程也比进程容易。

**6) 支持多处理机系统**

在多处理系统中，对于传统的进程，不管有多少处理机，该进程只能运行在一个处理机上。

但对于多线程进程，就可以将一个进程中的多个线程分配到多个处理机上，使它们并行执行，可以加速进程的完成。

(7)状态

与传统的进程类似，线程可以创建子线程，在各线程之间也存在着共享资源和相互合作的制约关系。

线程运行时也具有三种基本的状态：运行、阻塞、就绪。

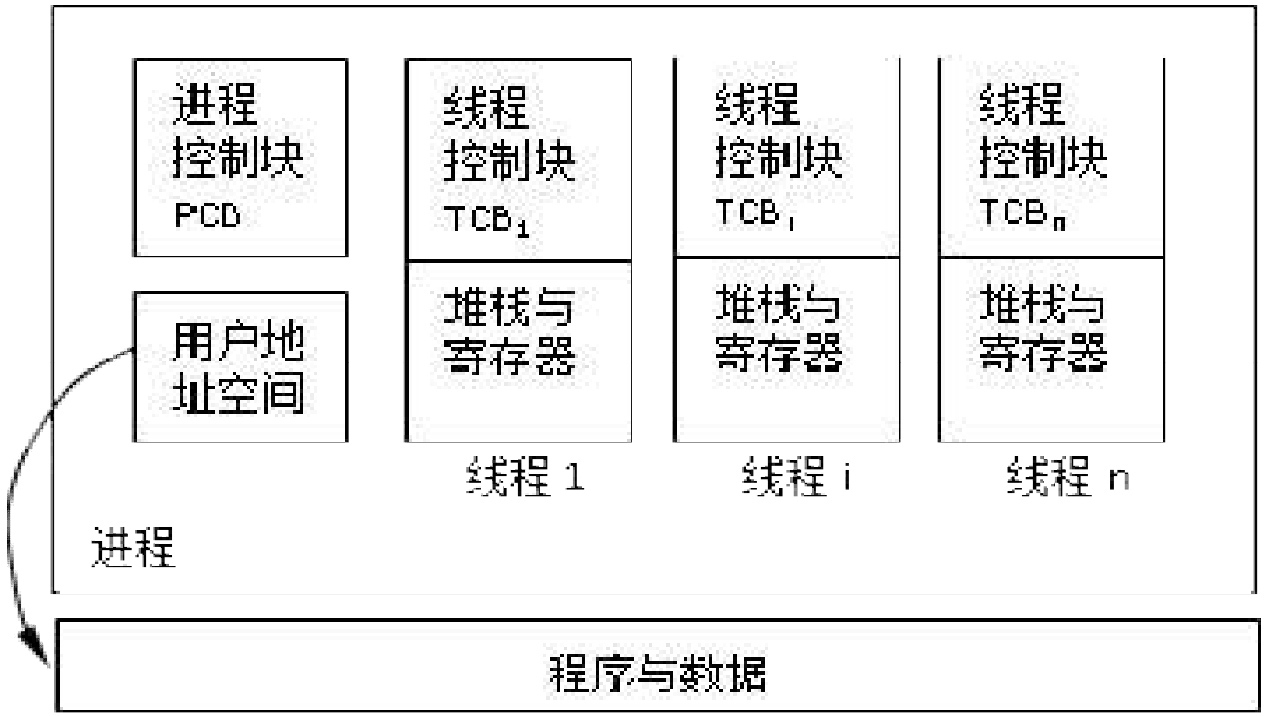



图 3-11　多线程与进程之间的关系

# 3.7.3 线程的控制

1.**TCB**(Thread Control Black)：用于指示被执行指令的程序计数器和寄存器保留局部变量、少数状态参数和返回地址的一组寄存器和堆栈、运行状态和调度参数、现场存储区。

2.建立与撤销：初始化线程去创建多个线程；自行终止或强行终止，有些系统一旦建立一直会运行。

3.调度：与进程调度类似

4.进程全局变量和线程(私有)全局变量：一个进程中的所有变量便分为三类：进程全局变量(对该进程中的所有线程中的所有过程可见)，线程(私有)全局变量(对该线程中的所有过程可见)，过程局部变量(只对该过程可见)。

3.互斥与同步：与进程类似。

# 3.7.4 线程的实现

## 1．用户态线程

将线程包完全放在用户空间内，而核心对此一无所知。就核心而言，它只是在管理常规的进程——即单线程进程。线程们在一个“线程运行管理系统’’上执行，而线程运行管理系统则是一组管理线程的过程。

## 2．核心态线程

核心为每个进程准备了一张表，每个线程占一项，填有关的寄存器、状态、优先级和其他信息。这些信息与用户级线程是一样的，只是现在放在核心空间。所有对线程的操作都以系统调用的形式实现。当线程被阻塞时，操作系统不仅可以运行同一进程中的另一线程，而且可以运行别的进程中的线程。

## 3．对用户级线程和核心级线程的评价

（1）用户级线程的优点

①用户级线程包最明显的优点，是它可以在一个不支持线程的操作系统上实现。例如，UNIX并不支持线程，但已有了多个基于UNIX的用户级线程包。

②开销和性能。用户级线程切换比陷入核心至少快一个数量级。而核心态线程实现中，由于所有线程操作都以系统调用的形式实现，从而开销比在用户级调用线程运行管理系统中的过程大得多。

③用户级线程允许每个进程有自己特设的调度算法。对有些应用，如那些配有一个空闲区回收线程的应用，有了自己的调度算法，就不必担忧一个线程会在一些不适当的地方停下来。

④用户级线程的可扩充性也很好。而核心线程需要不停地使用核心空间，这在线程数较多时是个问题。

## (2) 核心级线程的优点

用户级线程虽然有较好的性能，但用户级线程包中也有一些大的问题。

①第一个问题是阻塞型系统调用会阻塞所有的线程。例如，线程在读一条空的管道时，会导致所在进程阻塞，这意味着该进程的所有线程(包括本线程)都被阻塞。而在核心实现中，同样情况发生时线程陷入内核，内核将线程挂起，并开始运行另一个线程。

②用户级线程包的另一个问题是：一个线程开始运行以后，除非它自愿放弃CPU，否则没有其他线程能得到运行。而在核心级线程中，周期发生的时钟中断可以解决这个问题。在用户级线程实现中，单进程中没有时钟中断，从而轮转式的调度是行不通的。

(3) 用户态线程和核心态线程都存在的问题

一个进程内的所有线程共享该进程的所有数据区和信号等资源，很多库程序变成不可再入代码。

# 3.7.5 线程的适用范围

(1) 服务器中的文件管理或通信控制。局域网的文件服务器由于等待盘操作经常被阻塞，可以由服务器派生出多个线程第一个线程睡眠第二个线程可继续运行。

(2) 线程也经常用于客户进程。例如：在多个服务器上备份文件，或者处理信号如Del，break等键盘中断。

(3) 前后台处理。

(4) 数据的批处理以及网络系统中的信息发送与接收和其他相关处理等。..

# 3.8 Linux的进程管理

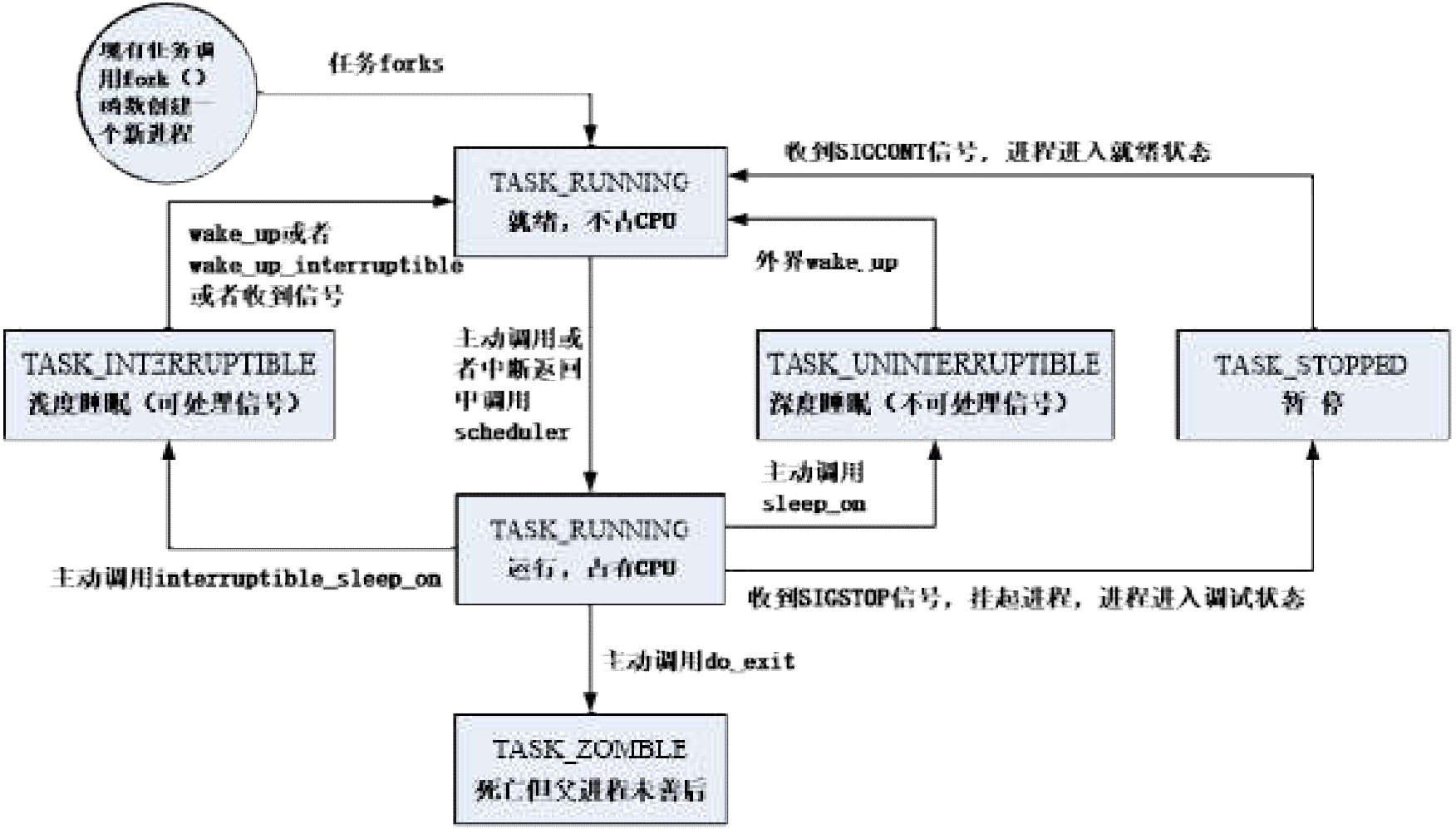
## 3.8.1 Linux进程概念与描述

Linux中的进程控制块是一个名叫task_struct的数据结构，记录了进程控制有关的信息， 所有的task_struct结构包含了在双向链表的task数组中。

```
struct task_struct {
        volatile long state; /* 进程的状态*/
        unsigned long flags; /* 进程标志 */
        mm_segment_t addr_limit; /* 线性地址空间*/
        …
       long counter; /* 进程的动态优先级*/
       long nice; /* 进程的静态优先级*/
       unsigned long policy; /* 进程采用的调度策略*/
      struct mm_struct *mm; /* 进程属性中指向内存管理的数据
                    结构mm_structd的 指针*/
       …
       struct task_struct *next_task , *prev_task; /*进程通
                              过这两个指针组成一个双向链表*/
```

```
 struct mm_struct *active_mm; /* 指向活动地址空间*/
 /* task state */
pid_t pid; /* 进程标志符, */
pid_t pgrp; /* 进程组标号*/
…
struct  task_struct  *p_opptr,  *p_pptr,  *p_cptr,  *p_ysptr,
        *p_osptr;
 /* 这五个标志表示一个进程的在计算机中的亲属关系,分别标志祖先
 进程,父进程,子进程,弟进程和兄进程,为了在两个进程之间共享方便
 而设立 */
struct list_head thread_group;
…
struct task_struct *pidhist_next;
 struct task_struct *pidhist_pprev; /* 上面两个指针是为了在计
                    算机中快速查一个进程而设立*/
…
struct fs_struct *fs; /* 指向和文件管理有关的数据结构*/
 …
};
```

# 3.8.2 Linux 中的进程状态及其转换



# 3.8.3 Linux的进程控制

## 1. 进程的建立

系统启动时，运行于核心模式，此时只有一个初始化进程即0#进程。当系统初始化完毕后，初始化进程启动init进程，然后进入空闲等待的循环中。

Init进程即1#进程，它完成一些系统初始化工作，如打开系统控制台，装根文件系统等。初始化工作结束后，init进程通过系统调用fork（）为每个终端承建一个终端子进程为用户服务，如等待用户登录、执行shell命令解释程序等。每个终端进程又可创建自己的子进程，从而形成一颗进程树。

在linux系统中，系统函数fork()，vfork()和clone都可以创建一个进程,但他们都是通过内核函数do_fork（）来实现。

do_fork（）函数主要工作如下：

（1）为新进程分配一个惟一的进程标识号PID和task_struct结构，然后把父进程中PCB的内容复制给新进程后检查用户具有执行一个新进程的必需的资源。

（2）设置task_struct中哪些与父进程值不同的数据成员。比如初始化自旋锁，初始化堆栈信息等等，同时会把新创建的子进程运行状态置为TASK_RUNNING（这里应该是就绪态）。

（3）设置进程管理信息，根据所提供的clone_flags参数值，决定是否对父进程task_struct结构中的文件系统、已打开的文件指针等所选择的部分进行拷贝，增加与父进程相关联的有关文件系统的进程引用数。

（4）初始化子进程的内核栈。通过拷贝父进程的上下文来初始化新进程的硬件下文。把新进程加入到pidhash[ ]散列表中，并增加任务计数值。

（5）启动调度程序使子进程获得运行机会。向父进程返回了子进程的PID。设置子进程在系统调用do_fork()返回0。

## 例3-11：调用fork()创建子进程的例子：

```
/* fork_test.c */
#include<sys/types.h>
#inlcude<unistd.h>
main()
{
     pid_t pid;
          /*此时仅有一个进程*/
     pid=fork();
     /*此时已经有两个进程在同时运行*/
     if(pid<0)
     printf("error in fork!");
     else if(pid==0)
     printf("I am the child process, my process ID is
%d\n",getpid());
     else
     printf("I am the parent process, my process ID is
%d\n",getpid());
}
```

结果：I am the parent process, my process ID is 1991

I am the child process, my process ID is 1992

## 2. 进程的撤销

当进程执行完毕，即正常结束时，调用exit（）自我终止。进程终止的系统调用sys_exit通过调用do_exit（）函数实现。do_exit系统调用主要完成下列工作：

（1）将进程的状态标志设为PF_EXITING，表示进程正在退出状态。

（2）释放分配给这个进程的大部分资源，包括内存，线性区描述符和页表、文件对象相关资源等。

（3）向父进程发送信号，给其子进程重新找父进程。

（4）将进程设置为TASK_ZOMBIE（僵死）状态，使进程不会再被调度。

（5）调用schedule（），重新调度其它进程执行。

处于“僵死状态”的进程运行已经结束，不会再被调度，内核释放了与其相关的所有资源，但其进程控制块还没有释放，由其父进程调用wait( )函数来查询子孙进程的退出状态，释放进程控制块。

## 3. 程序的装入和执行

当父进程使用fork（）系统调用创建了子进程之后，子进程继承了父进程的正文段和数据段，从而执行和父进程相同的程序段。为了使fork产生的子进程可以执行一个指定的可执行文件，系统内核中开发了一个系统函数调用exec（）。exec（）是一个调用族，每个调用函数参数稍有不同，但它们的目的都是把文件系统中的可执行文件调入并覆盖调用进程的正文段和数据段之后执行。

# 3.8.4 Linux进程通信

## 1. 管道通信

管道通信技术又分为无名管道和有名管道两种类型。

无名管道为建立管道的进程及其子孙进程提供一条以比特流方式传递消息的通信管道。该管道在逻辑上被看做管道文件，在物理上则由文件系统的高速缓存区构成。发送进程利用系统调用write（fd[1]，buf，size）把buf中的长度为size字符的消息送入管道口fd[1]，接收进程则使用系统调用read（fd[0]，buf，size）从管道出口 fd[0]读出size字符的消息送入buf中。此外，管道按先进先出（FIFO）方式传递消息，且只能单向传递。

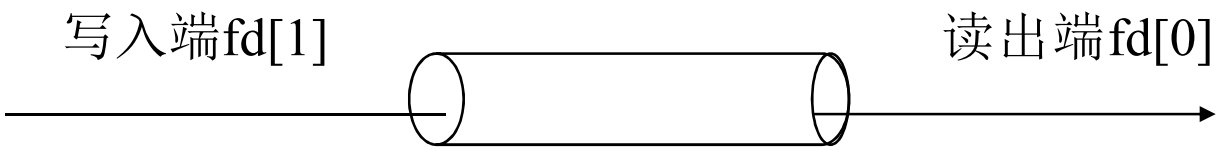



图3-14 管道通信示意图

## 例3-12：管道通信

```
#include <stdio.h>
main()
{    int x,fd[2];   char buf[50],s[50];
     pipe(fd); /*创建管道*/
     while((x=fork())==－1) /*创建子进程失败，循环*/
  {
   if (x==0) /*执行子进程*/
    {
     sprintf(buf,“This is an example of pipe\n”);
     write(fd[1],buf,50); /*把buf中的字符写入管道*/
     exit(0);
    }
  else /*父进程返回*/
   {
        wait（0）;
        read（fd[0],s,50）; /*父进程读管道中的字符*/
        printf(“%s”,s);
   }
}
```

## 2. 信号

如果想终止正在运行的进程，按Ctrl+C,进程将收到SIGINT信号，内核运行相应的中断处理程序，完成有关SIGINT的处理工作。

进程可以选择对某种信号所采取的特定操作，这些操作包括（1）忽略或阻塞信号。进程可忽略产生的信号，但SIGKILL和SIGSTOP信号不能被忽略；进程可选择阻塞某些信号;（2）由进程处理该信号。进程本身可在系统中注册处理信号的处理程序地址，当发出该信号时，由注册的处理程序处理信号；（3）由内核进行默认处理。信号由内核的默认处理程序处理。大多数情况下，信号由内核进行处理。

用signal(int sig，int(*sig-process))来处理其它进程发来的信号:捕获信号sig，当捕捉到该信号时后，以函数sig-process来处理该信号。

Kill(int pid,int sig):给进程pid发送sig信号。

Pause():使进程暂停执行直到收到信号为止。

## 3. IPC机制（Interprocess communication)

1）消息队列

Linux 为系统中所有的消息队列维护一个 msgque 链表，该链表中的每个指针指向一个 msgid_ds 结构，该结构完整描述一个消息队列。当建立一个消息队列时，系统从内存中分配一个 msgid_ds 结构并将指针添加到 msgque 链表。

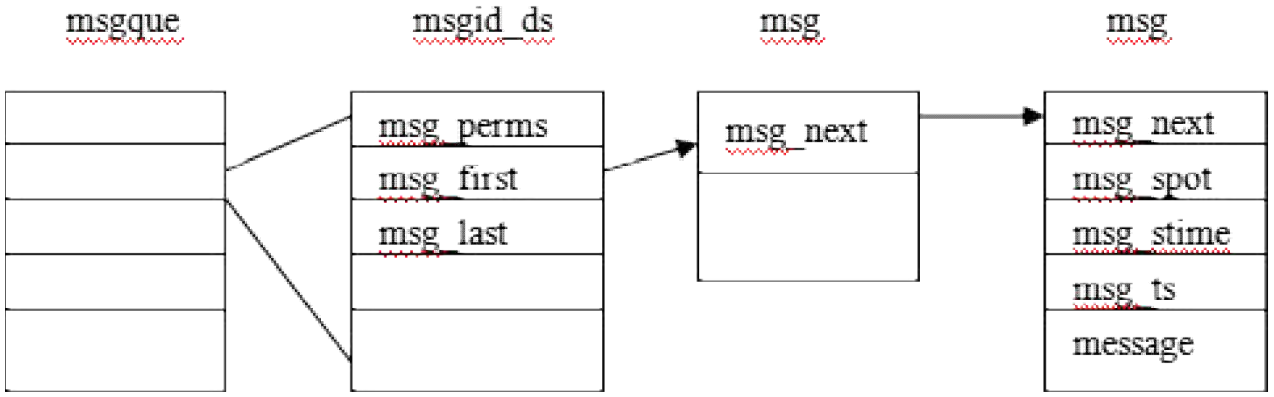



图 3－16　Linux 消息队列

# 2）信号量

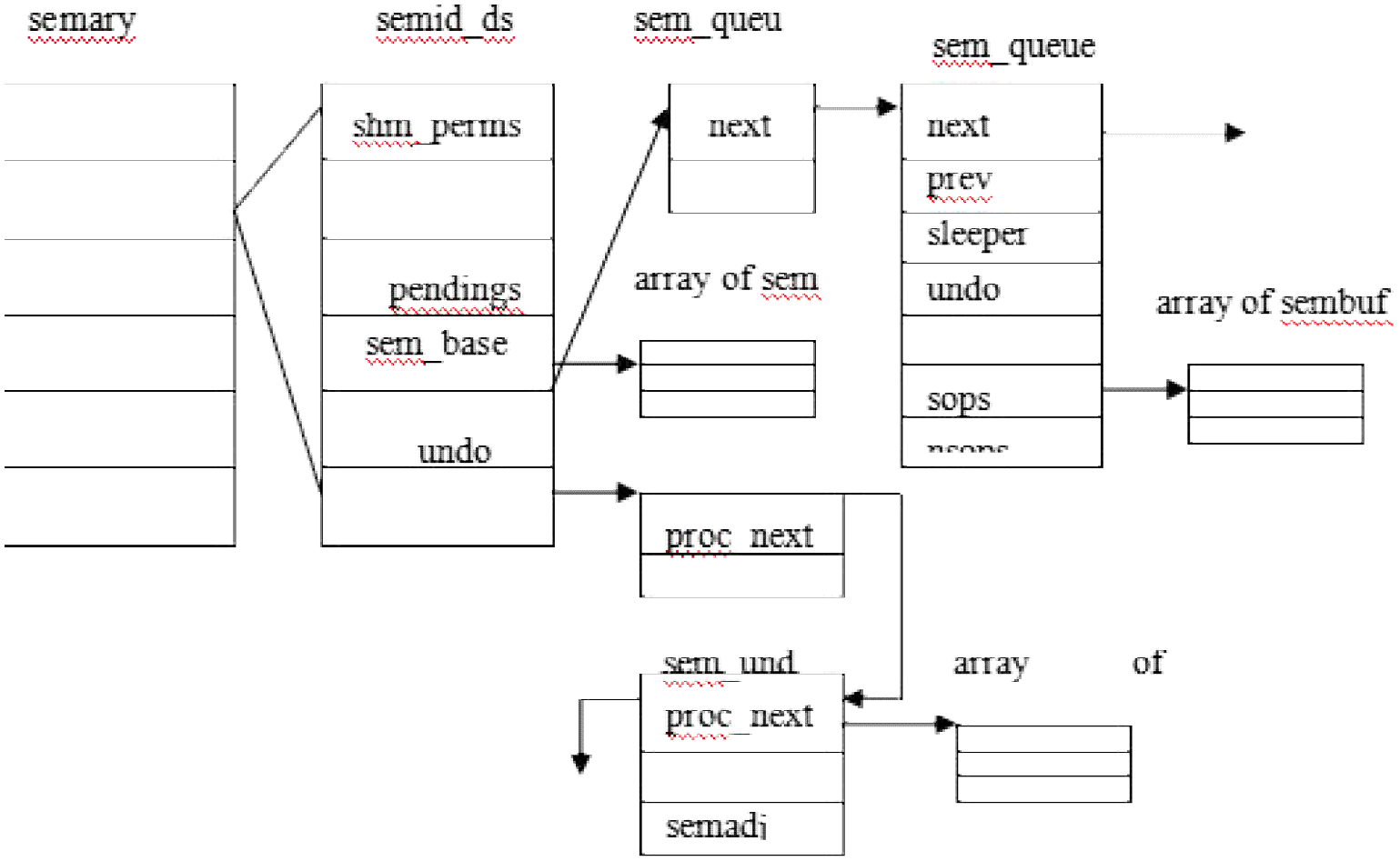



图 3－17　linux 信号量集合

```
struct sem {
        int  semval;              /* current value */
        int  sempid;              /* pid of last operation */
  };
```

int semid=semget(key-t key,int nsems,int semflg )创建一个信号量组，nsems信号量个数， semflg信号量存取权限。

int semop(int semid,struct sembuf *sops,unsignd nosps)完成对信号量的操作。nosps信号量操作个数,

操作数组：struct sembuf {

```
        unsigned short  sem_num;       /* semaphore index in array */
         short          sem_op;        /* semaphore operation */
        short  sem_flg;          /* operation flags */};(P、V)
```

如果sem_op<0, 信号量的值减去sem_op的绝对值，意味着要获取资源，这些资源由信号量控制来存取。如果没有指定IPC-NOWAIT，那么调用的进程睡眠等到资源得到满足。

如果sem_op>0，把它的值加到信号量，这意味着归还资源。

如果sem_op=0，本次操作要对信号量进行测试。若为零正常返回，不为零，进程睡眠直到信号量为0。

# 3）共享内存

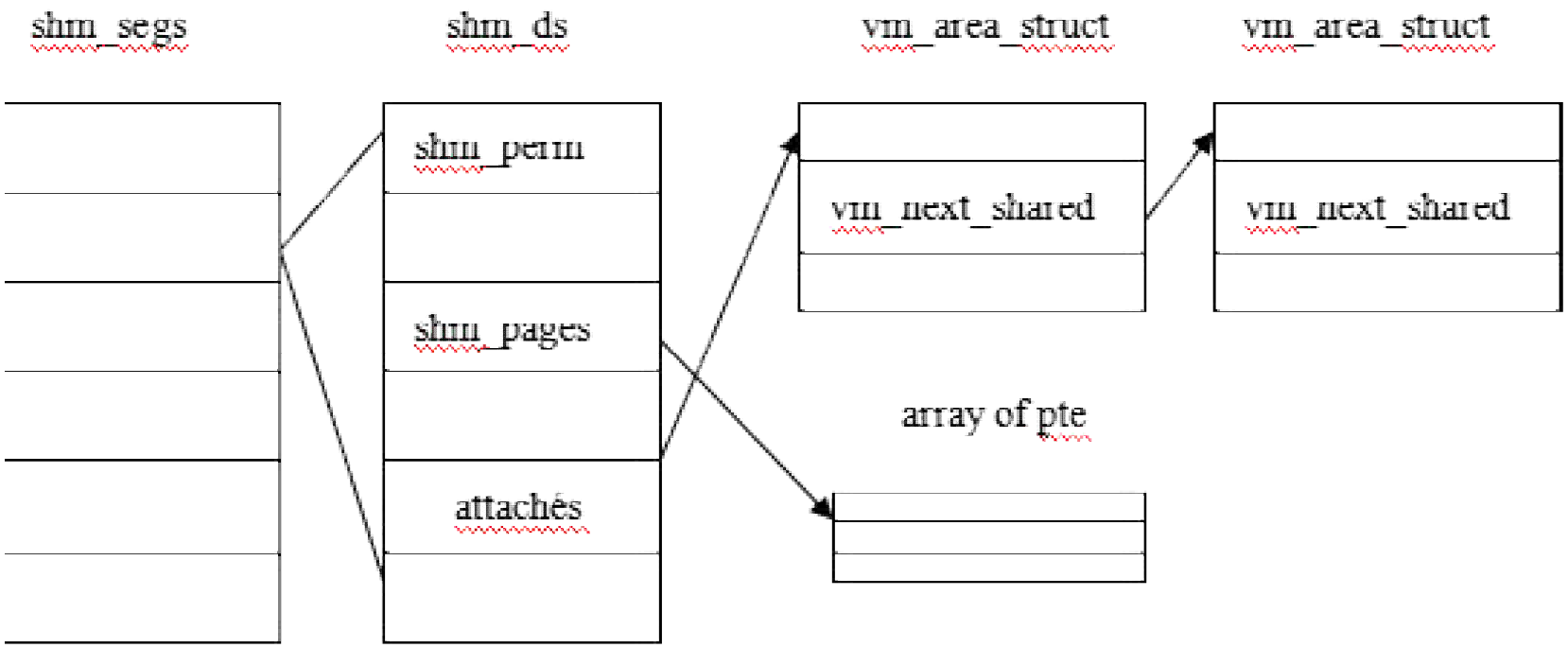



图 3　18　linux 共享内存段