

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(Threads)的基本概念

2.8 线程的实现

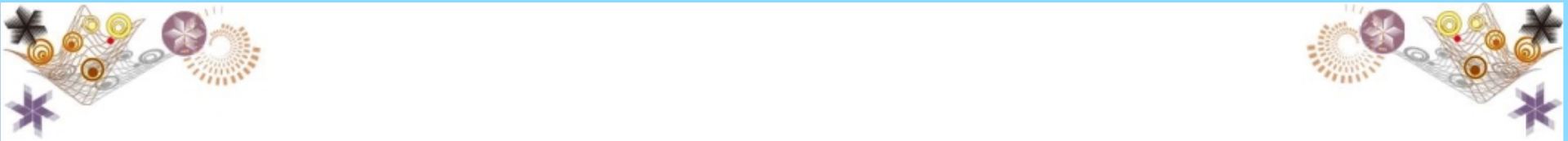
2.1 前趋图和程序执行

在早期未配置OS的系统和单道批处理系统中，程序的执行方式是顺序执行，即在内存中仅装入一道用户程序，由它独占系统中的所有资源，只有在一个用户程序执行完成后，才允许装入另一个程序并执行。可见，这种方式浪费资源、系统运行效率低等缺点。

2.1.1 前趋图

所谓前趋图(Precedence Graph), 是指一个有向无循环图, 可记为DAG(Directed Acyclic Graph), 它用于描述进程之间执行的先后顺序。

图中的每个结点可用来表示一个进程或程序段, 乃至一条语句, 结点间的有向边则表示两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)。



进程(或程序)之间的前趋关系用“ \rightarrow ”来表示，如果进程 P_i 和 P_j 存在着前趋关系，可表示为 $(P_i, P_j) \in \rightarrow$ ，也可写成 $P_i \rightarrow P_j$ ，表示在 P_j 开始执行之前 P_i 必须完成。此时称 P_i 是 P_j 的直接前趋，而称 P_j 是 P_i 的直接后继。

在前趋图中，把没有前趋的结点称为初始结点(Initial Node)，把没有后继的结点称为终止结点(Final Node)。此外，每个结点还具有一个重量(Weight)，用于表示该结点所含有的程序量或程序的执行时间。

在图2-1(a)所示的前趋图中，存在着如下前趋关系：

$$P_1 \rightarrow P_2, P_1 \rightarrow P_3, P_1 \rightarrow P_4, P_2 \rightarrow P_5,$$

$$P_3 \rightarrow P_5, P_4 \rightarrow P_6, P_4 \rightarrow P_7, P_5 \rightarrow P_8, P_6 \rightarrow P_8,$$

$$P_7 \rightarrow P_9, P_8 \rightarrow P_9$$

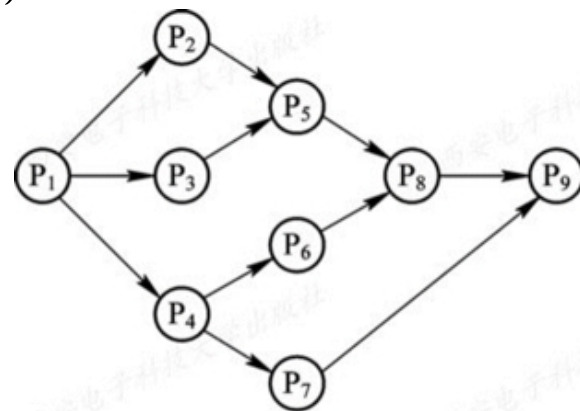
或表示为：

$$P = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$$

$$= \{(P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_5), (P_3,$$

$$P_5), (P_4, P_6), (P_4, P_7), (P_5, P_8), (P_6, P_8), (P_7, P_9),$$

$$(P_8, P_9)\}$$



(a) 具有九个结点的前趋图

前趋图中是**不允许有循环**的，否则必然会产生不可能实现的前趋关系。如图2-1(b)所示的前趋关系中就存在着循环。它一方面要求在S3开始执行之前，S2必须完成，另一方面又要求在S2开始执行之前，S3必须完成。显然，这种关系是不可能实现的。

$$S_2 \rightarrow S_3, S_3 \rightarrow S_2$$



(b) 具有循环的前趋图



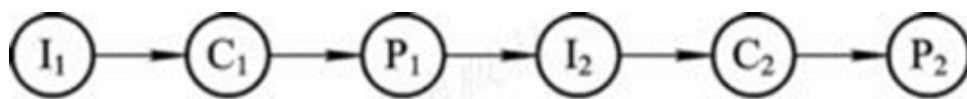
2.1.2 程序顺序执行

1. 程序的顺序执行

一个应用程序由若干个**程序段**组成，每一个程序段完成特定的功能，它们在执行时，都需要按照某种先后次序顺序执行，仅当前一程序段执行完后，才运行后一程序段。

例如，我们用结点(Node)代表各程序段的的操作(在图2-1中用圆圈表示)，其中I代表输入操作，C代表计算操作，P为打印操作，用箭头指示操作的先后次序。

上述的三个程序段间就存在着这样的前趋关系： $I_i \rightarrow C_i \rightarrow P_i$ ，其执行的顺序可用前趋图2-2(a)描述。



(a) 程序的顺序执行

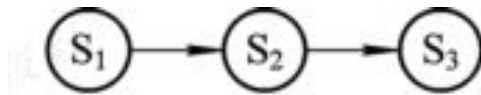
一个包含了三条语句的程序段：

$S_1: a := x + y;$

$S_2: b := a - 5;$

$S_3: c := b + 1;$

其中，语句 S_2 必须在语句 S_1 后(即 a 被赋值)才能执行，语句 S_3 也只能在 b 被赋值后才能执行，因此，三条语句存在着这样的前趋关系： $S_1 \rightarrow S_2 \rightarrow S_3$ ，应按前趋图2-2(b)所示的顺序执行。



(b) 三条语句的顺序执行



2. 程序顺序执行时的特征

① 顺序性：指处理机严格地按照程序所规定的**顺序执行**，即每一操作必须在下一个操作开始之前结束；

② 封闭性：指程序在封闭的环境下运行，即**程序运行时独占全机资源**，资源的状态(除初始状态外)只有本程序才能改变它，程序一旦开始执行，其执行结果不受外界因素影响；

③ 可再现性：指只要程序执行时的环境和初始条件相同，当程序重复执行时，不论它是从头到尾不停顿地执行，还是“停停走走”地执行，都可获得相同的结果。

2.1.3 程序并发执行

1. 程序的并发执行

设对一批作业进行处理时，每道作业的输入、计算和打印程序段的执行情况如图2-3所示。

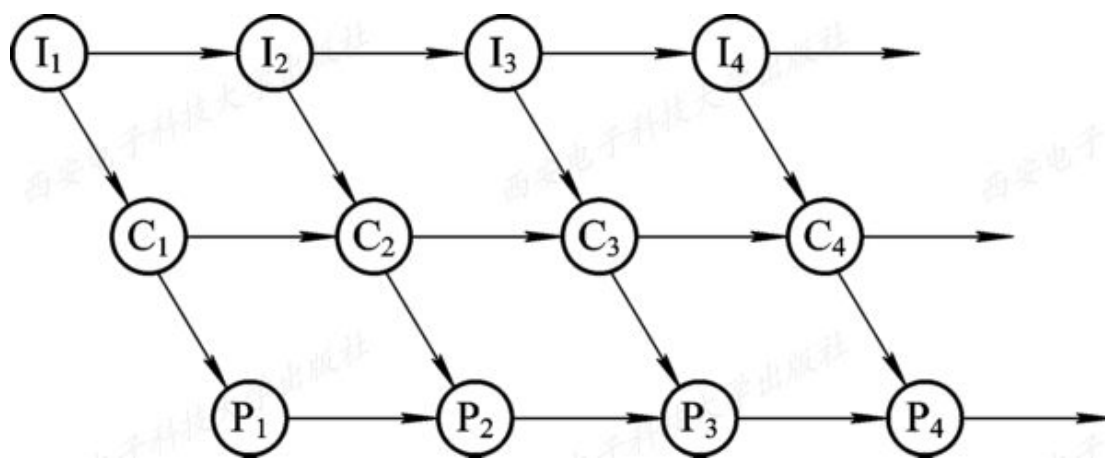


图2-3 程序并发执行时的前趋图

对于具有下述四条语句的程序段：

$S_1: a := x + 2$

$S_2: b := y + 4$

$S_3: c := a + b$

$S_4: d := c + b$

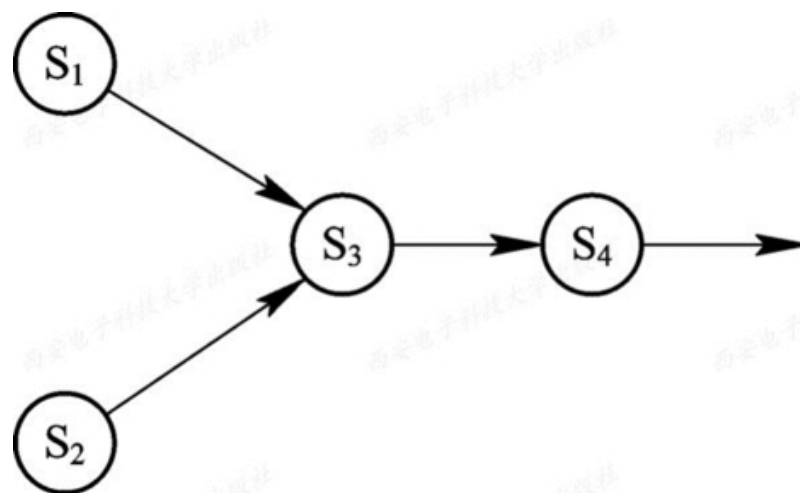




图2-4 四条语句的前趋关系



2. 程序并发执行时的特征

- (1) 间断性。
- (2) 失去封闭性。
- (3) 不可再现性。

(1) 间断性

为完成同一项任务而合作形成的程序间相互制约的关系：如果 I_2 已完成，但 C_1 未完成，计算程序不能继续执行完成 C_2 ，即计算只能暂停。

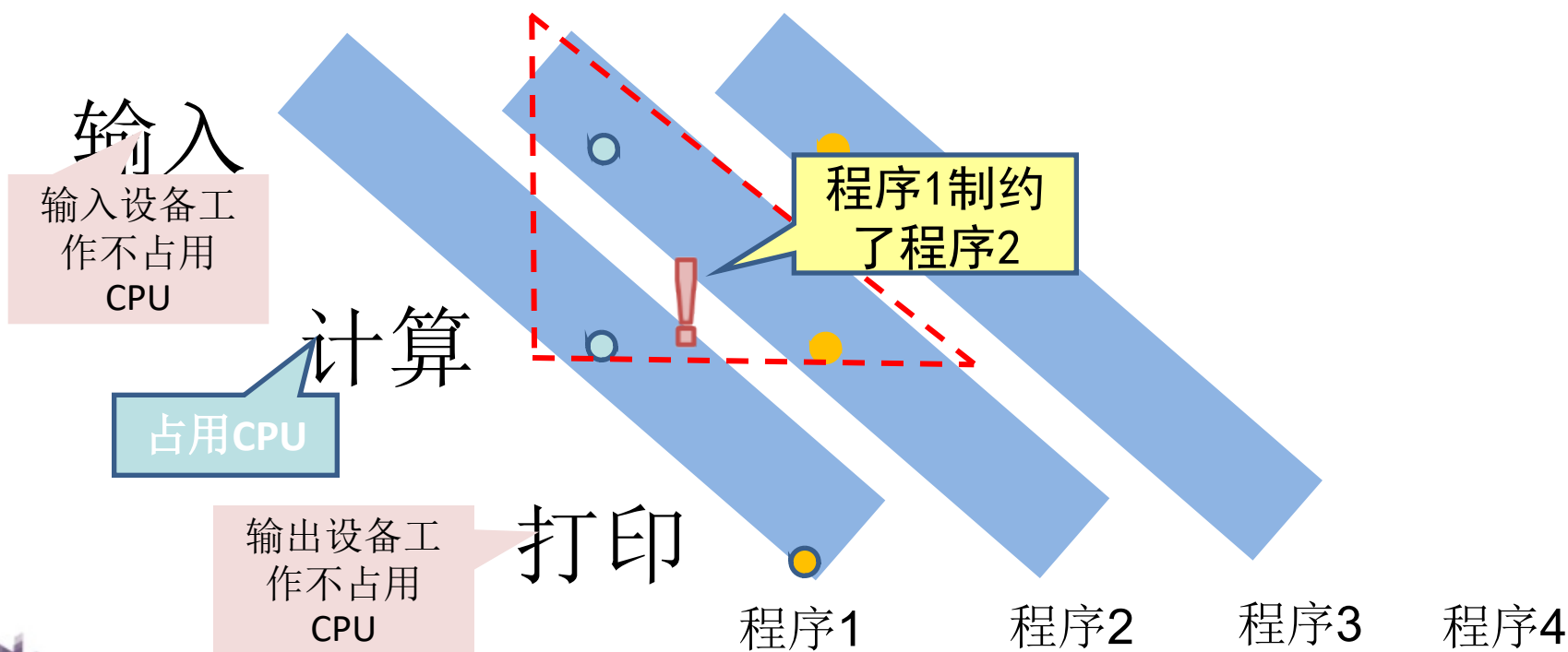


图 2-3 并发执行时的前趋图



(2)失去封闭性

系统中的**资源**由多个程序**共享**，资源的状态不再只由一个程序改变

(3)不可再现性

例：有两个循环程序A和B，它们共享一个变量N。

程序A：每执行一次时，都要做 $N := N+1$ 操作；

程序B：每执行一次时，先执行Print(N)操作，再将N置成“0”。

程序A和程序B对N的访问没有任何控制机制来协调。

程序A和B以不同的速度运行

(1)	N值
$N := N+1$	N+1
Print(N)	N+1
$N := 0$	0

(2)	N值
Print(N)	N
$N := 0$	0
$N := N+1$	1

程序A和程序B并发执行，不具有可再现性，执行环境和初始条件（N的值）相同，结果却不相同。

(3)	N值
Print(N)	N
$N := N+1$	N+1
$N := 0$	0



2.2 进程的描述

2.2.1 进程的定义和特征

1. 进程的定义

进程实体（又称进程映像）由程序段、相关的数据段和进程控制块（Process Control Block, PCB）三部分组成。一般情况下，把进程实体简称为进程。

创建进程：实质上是创建进程实体中的PCB。撤销进程，实质上是撤销进程的PCB。



2.2 进程的描述

2.2.1 进程的定义和特征

1. 进程的定义

从不同的角度，进程可以有不同的定义（都强调动态性）

:

(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。

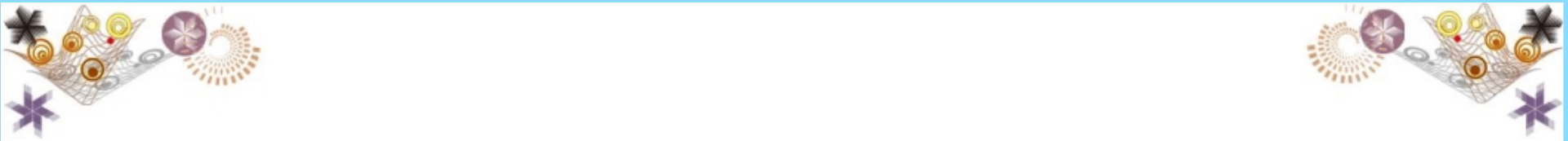
(3) 进程是具有独立功能的程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。



2. 进程的特征

进程和程序是两个截然不同的概念，除了进程具有程序所没有的PCB结构外，还具有下面一些特征：

- (1) 动态性：进程是动态地产生、变化和消亡的。
- (2) 并发性：内存中有多个进程实体，各进程并发执行。
- (3) 独立性：进程是能独立运行、独立获得资源、独立接受调度的基本单位。
- (4) 异步性：各进程按各自独立的、不可预知的速度向前推进。



2.2.2 进程的基本状态及转换

1. 进程的三种基本状态

一般而言，每一个进程至少应处于以下三种基本状态之一：

(1) **就绪(Ready)状态**：已经处于准备好运行的状态，但是没有空闲CPU，而暂时不能运行。

(2) **执行(Running)状态**：获得CPU，正在运行中的状态。在任一时刻，单处理机系统，只能有一个进程处于执行状态，在多处理机中，可以有多个进程处于执行状态。

(3) **阻塞(Block)状态**：正在执行的进程由于发生某事件暂时无法继续执行的状态。



	就绪状态	执行状态	阻塞状态
是否获得CPU	没有	有	没有
进程个数	可以有多个	单处理机：1个 多处理机：多个	可以有多个
进程能否被调度	能	*	不能



2. 三种基本状态的转换

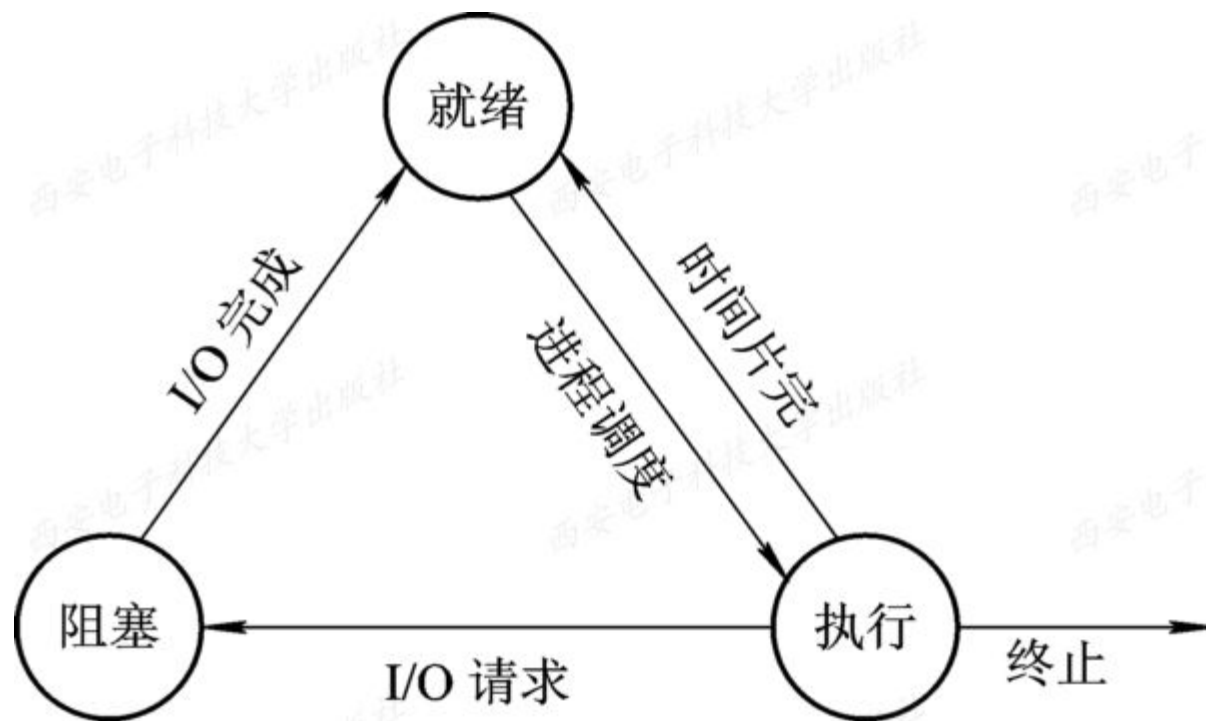
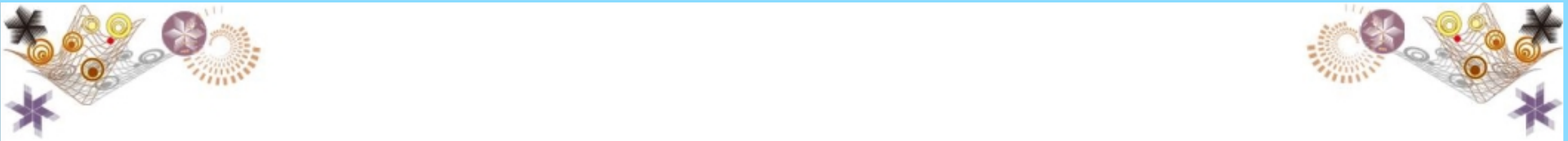


图2-5 进程的三种基本状态及其转换



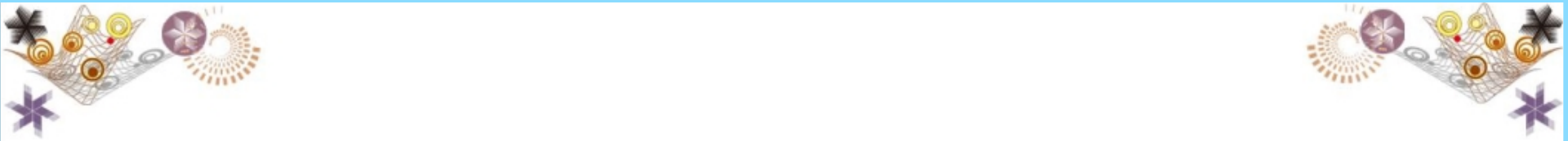
3. 创建状态和终止状态

1) 创建状态

进程是由创建而产生。创建一个进程是个很复杂的过程，一般要通过多个步骤才能完成：

- ① 首先由进程**申请一个空白PCB**，并向PCB中填写用于控制和管理进程的信息；
- ② 然后为该进程**分配**运行时所必须的**资源**；
- ③ 最后，把该进程转入就绪状态并**插入就绪队列**之中。

但如果进程所需的资源尚不能得到满足，比如系统尚无足够的内存使进程无法装入其中，此时创建工作尚未完成，进程不能被调度运行，于是把此时进程所处的状态称为**创建状态**。



2) 终止状态

进程的终止也要通过两个步骤：首先，是等待操作系统进行**善后处理**，最后将其**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，供其他进程收集。

一旦其他进程完成了对其信息的提取之后，操作系统将删除该进程，即将其**PCB清零**，并将该空白PCB返还系统。

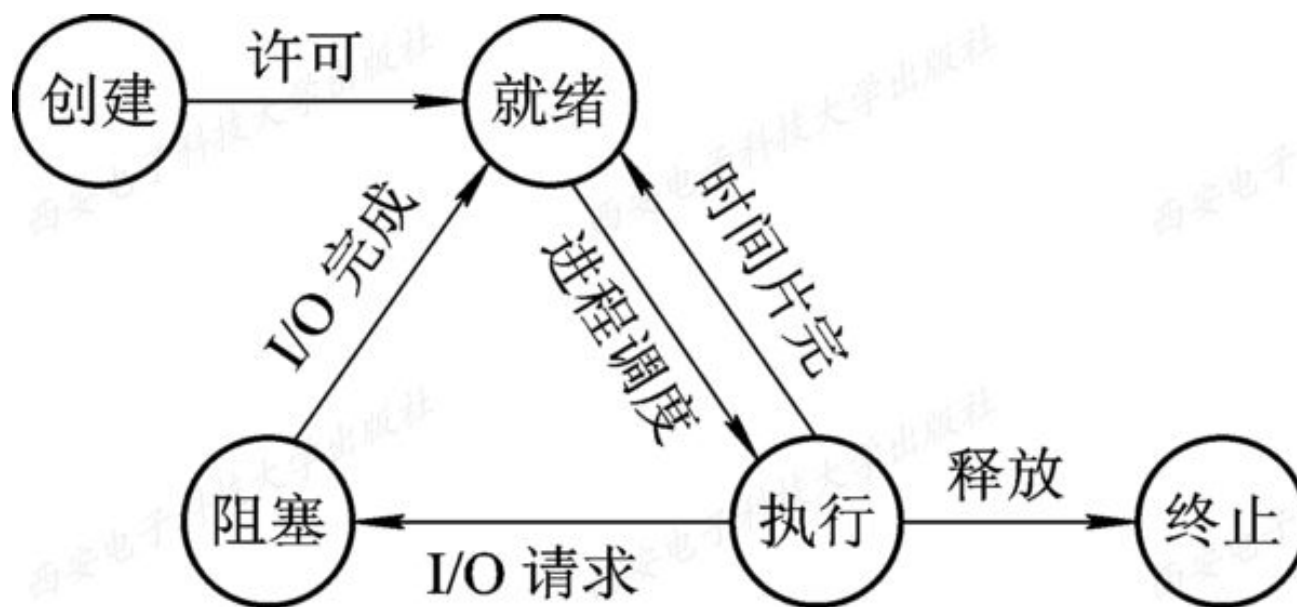
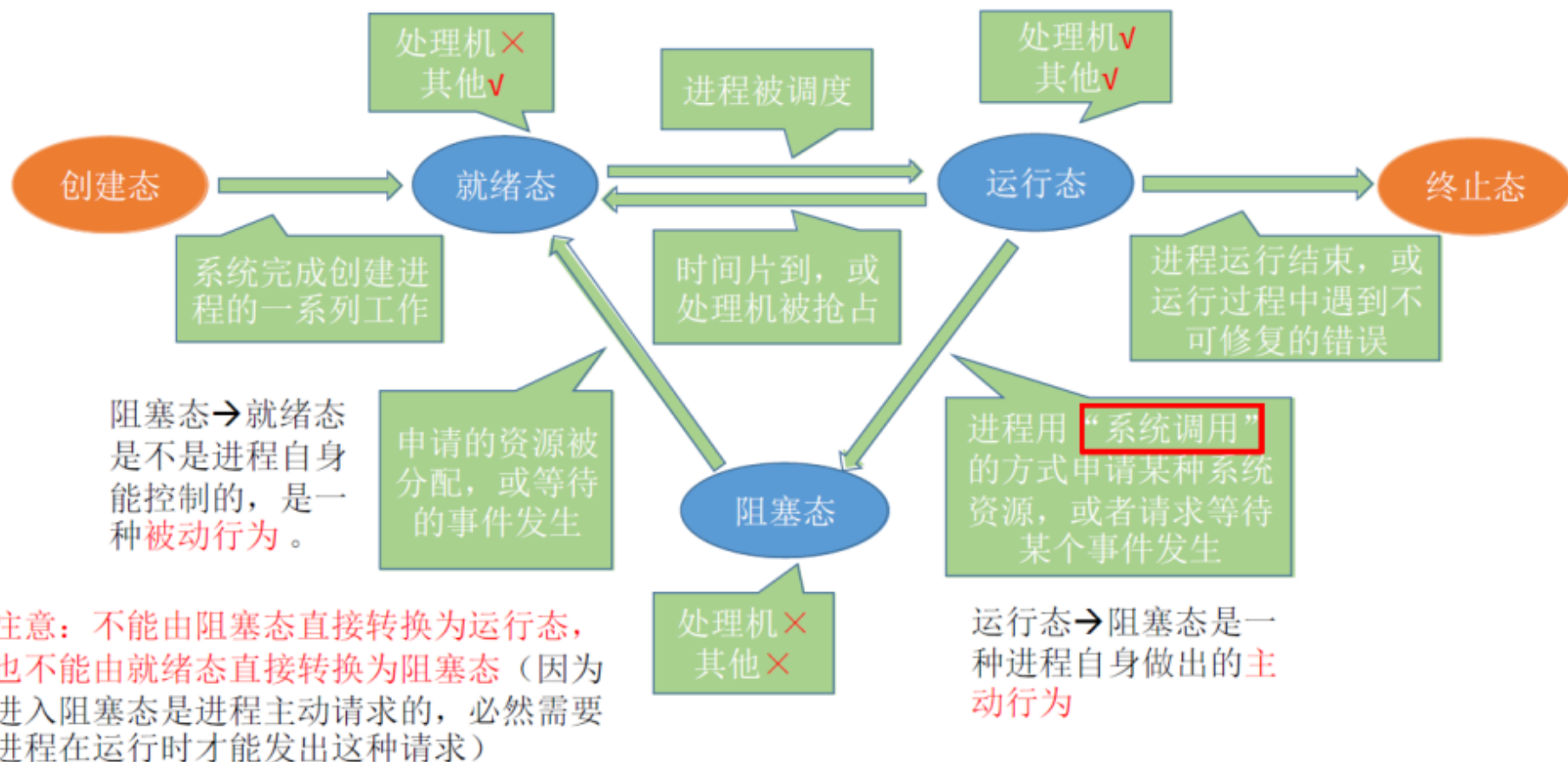


图2-6 进程的五种基本状态及转换 (1)



王道考研 / CSKAQYAN.CN

进程的五种基本状态及转换 (2)

2.2.3 挂起操作和进程状态的转换

挂起是指把一个进程从内存换到外存。

1. 挂起操作的引入

引入挂起操作的原因，是基于系统和用户的如下需要：

(1) 终端用户的需要：用户发现进程有问题，要求挂起自己的进程，以便进行检查和修改。

(2) 父进程请求：父进程希望挂起自己的某个子进程，以便进行检查和修改。

(3) 负荷调节的需要：系统中的进程过多，负荷过重，需要挂起一部分进程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者进行记账。

2. 引入挂起原语操作后三个进程状态的转换

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 (Readya) \leftrightarrow 静止就绪 (Readys)
- (2) 活动阻塞 (Blockeda) \leftrightarrow 静止阻塞 (Blockedts)

原语(Primitive)是由若干条指令组成的、用于完成一定功能的一个过程。原语是原子操作，即一个操作中的动作要么全做，要么全不做，是一个不可分割的单位。在执行过程中不允许被中断。

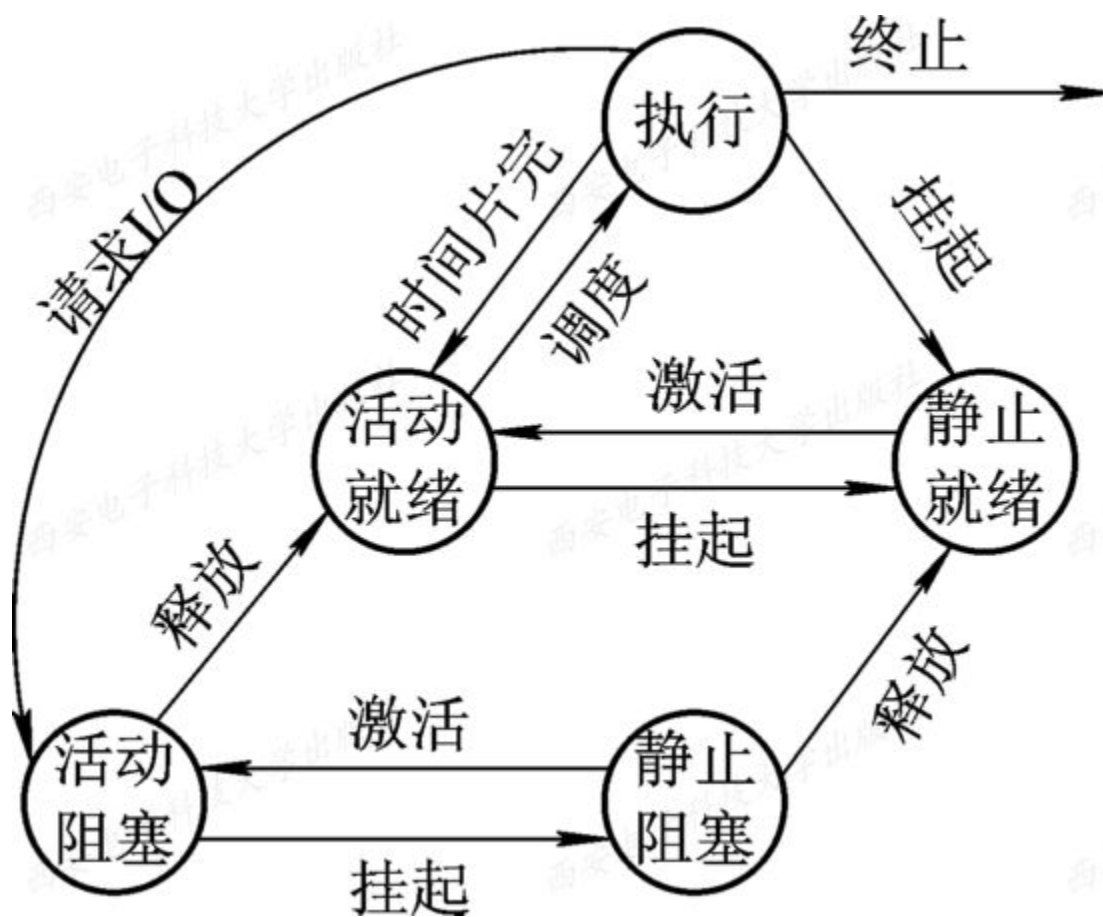


图2-7 具有挂起状态的进程状态图

3. 引入挂起操作后五个进程状态的转换

如图2-8示出了增加了创建状态和终止状态后具有挂起状态的进程状态及转换图。与图2-7所示的进程五状态转换相比较，要增加考虑下面的几种情况：

- (1) NULL→创建：
- (2) 创建→活动就绪：
- (3) 创建→静止就绪：
- (4) 执行→终止：

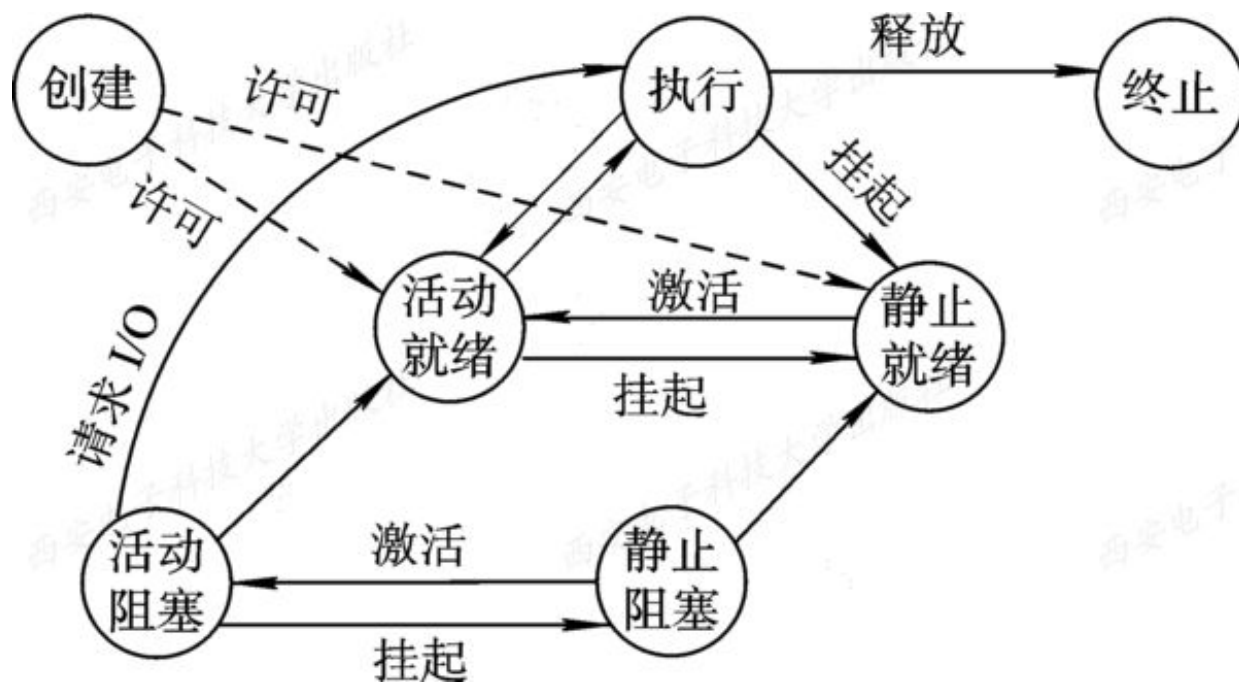


图2-8 具有创建、终止和挂起状态的进程状态图



2.2.4 进程管理中的数据结构

1. 操作系统中用于管理控制的数据结构

在计算机系统中，对于每个资源和每个进程都设置了一个数据结构，用于表征其实体，我们称之为**资源信息表**或**进程信息表**，其中包含了**资源**或**进程**的标识、描述、状态等信息以及一批指针。通过这些指针，可以将同类资源或进程的信息表，或者同一进程所占用的资源信息表分类链接成不同的队列，便于操作系统进行查找。

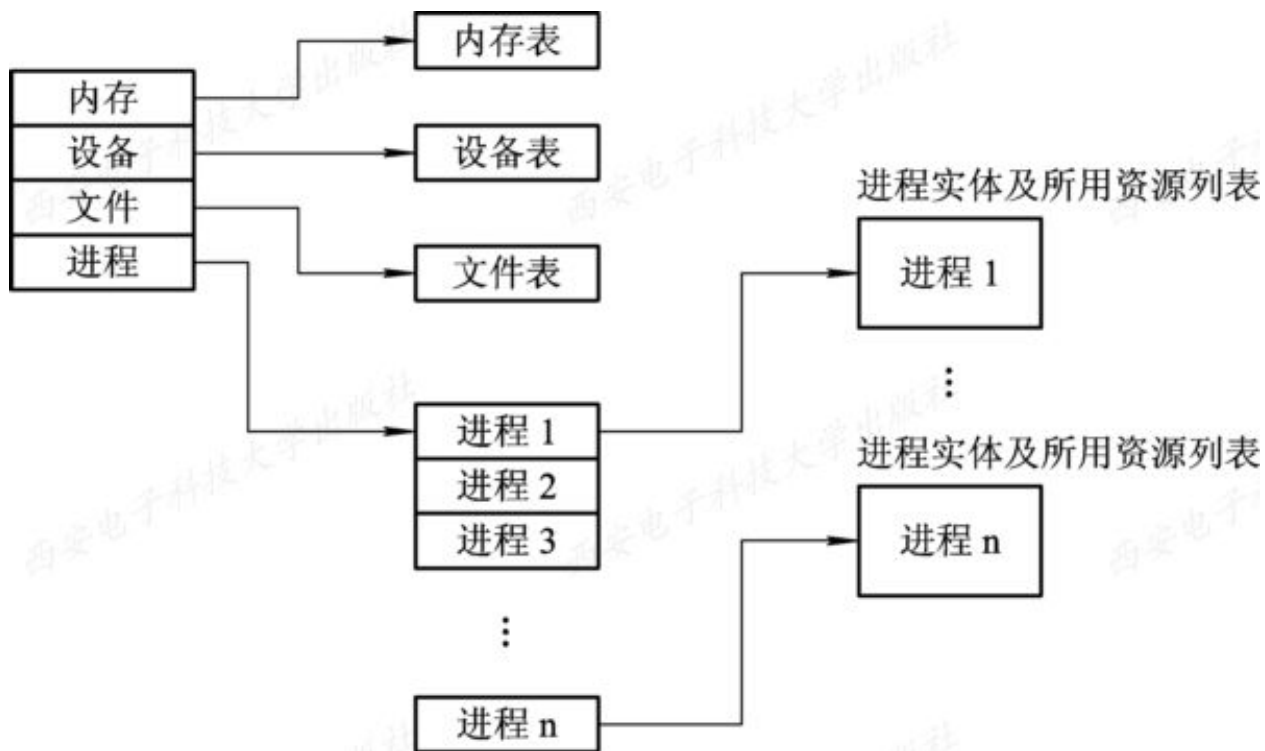


图2-9 操作系统控制表的一般结构



2. 进程控制块PCB的作用

进程表又被称为进程控制块PCB，其主要作用有：

- (1) 作为独立运行基本单位的标志；
- (2) 能实现间断性运行方式；
- (3) 提供进程管理所需要的信息；
- (4) 提供进程调度所需要的信息；
- (5) 实现与其它进程的同步与通信。



3. 进程控制块中的信息

在进程控制块中，主要包括下述四个方面的信息。

1) 进程标识符（PID）

进程标识符用于唯一地标识一个进程。一个进程通常有两种标识符：

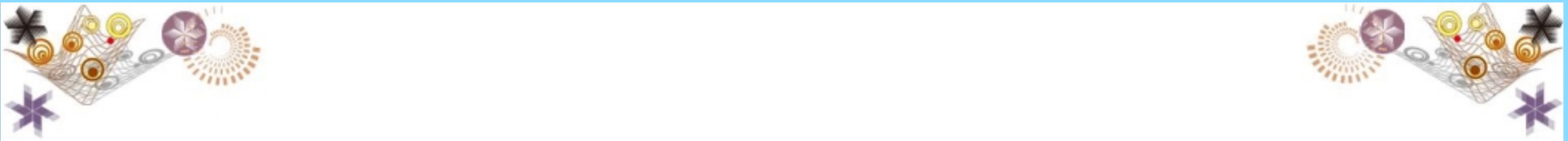
(1) 外部标识符：由创建者提供，通常是由字母、数字组成，往往是由用户(进程)在访问该进程时使用。为了描述进程的家族关系，还应设置父进程标识及子进程标识。此外，还可以设置用户标识，以指示拥有该进程的用户。

(2) 内部标识符：OS为每一个进程赋予一个惟一的数字标识符，它通常是一个进程的序号。设置内部标识符主要是为了方便系统使用。

2) 处理机状态

处理机状态信息也称为处理机的上下文，主要是由处理机的各种寄存器中的内容组成的。包括：

- ① **通用寄存器**（用户可视寄存器），是用户程序可以访问的，用于**暂存信息**；
- ② **指令计数器PC**，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。栈指针**指向该栈的栈顶**。



3) 进程调度信息

在OS进行调度时，必须了解进程的状态及有关进程调度的信息，这些信息包括：

- ① 进程状态，指明进程的当前状态，它是作为进程调度和对换时的依据；
- ② 进程优先级，是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**，是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

是指用于进程控制所必须的信息，它包括：

- ① **程序和数据地址**，进程实体中的程序和数据内存或外存地(首)址，以便再调度到该进程执行时，能从PCB中找到其程序和数据；
- ② **进程同步和通信机制**，这是实现进程同步和进程通信时必需的机制，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**，在该清单中列出了进程在运行期间所需的全部资源(除CPU以外)，另外还有一张已分配到该进程的资源清单；
- 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。



进程描述信息	进程控制和管理信息	资源分配清单	处理机相关信息
进程标识符(PID)	进程当前状态	代码段指针	通用寄存器值
用户标识符(UID)	进程优先级	数据段指针	地址寄存器值
	代码运行入口地址	堆栈段指针	控制寄存器值
	程序的外存地址	文件描述符	标志寄存器值
	进入内存时间	键盘	状态字
	处理机占用时间	鼠标	
	信号量使用		



总结：PCB包含的主要内容



4. 进程控制块的组织方式

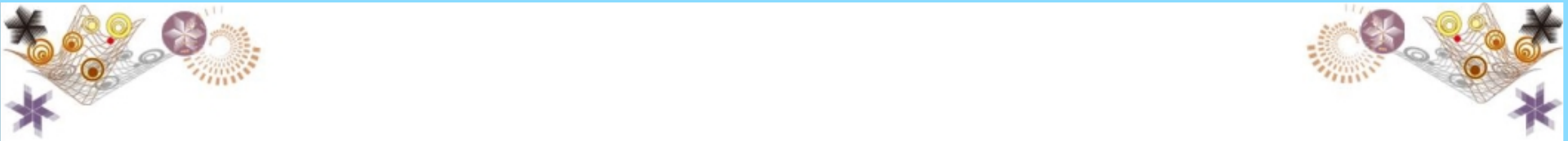
目前常用的组织方式有以下三种。

(1) 线性方式，即将系统中所有的PCB都组织在一张线性表中，将该表的首址存放在内存的一个专用区域中。该方式实现简单、开销小，但每次查找时都需要扫描整张表，因此适合进程数目不多的系统。图2-10示出了线性表的PCB组织方式。



PCB1
PCB2
PCB3
⋮
PCB _n

图2-10 PCB线性表示意图



(2) 链接方式，即把具有相同状态进程的PCB分别通过PCB中的链接字链接成一个队列，形成就绪队列、若干个阻塞队列和空闲队列等。

对就绪队列而言，往往按进程的优先级将PCB从高到低进行排列，将优先级高的进程PCB排在队列的前面。同样，也可把处于阻塞状态进程的PCB根据其阻塞原因的不同，排成多个阻塞队列，如等待I/O操作完成的队列和等待分配内存的队列等。图2-11示出了一种链接队列的组织方式。

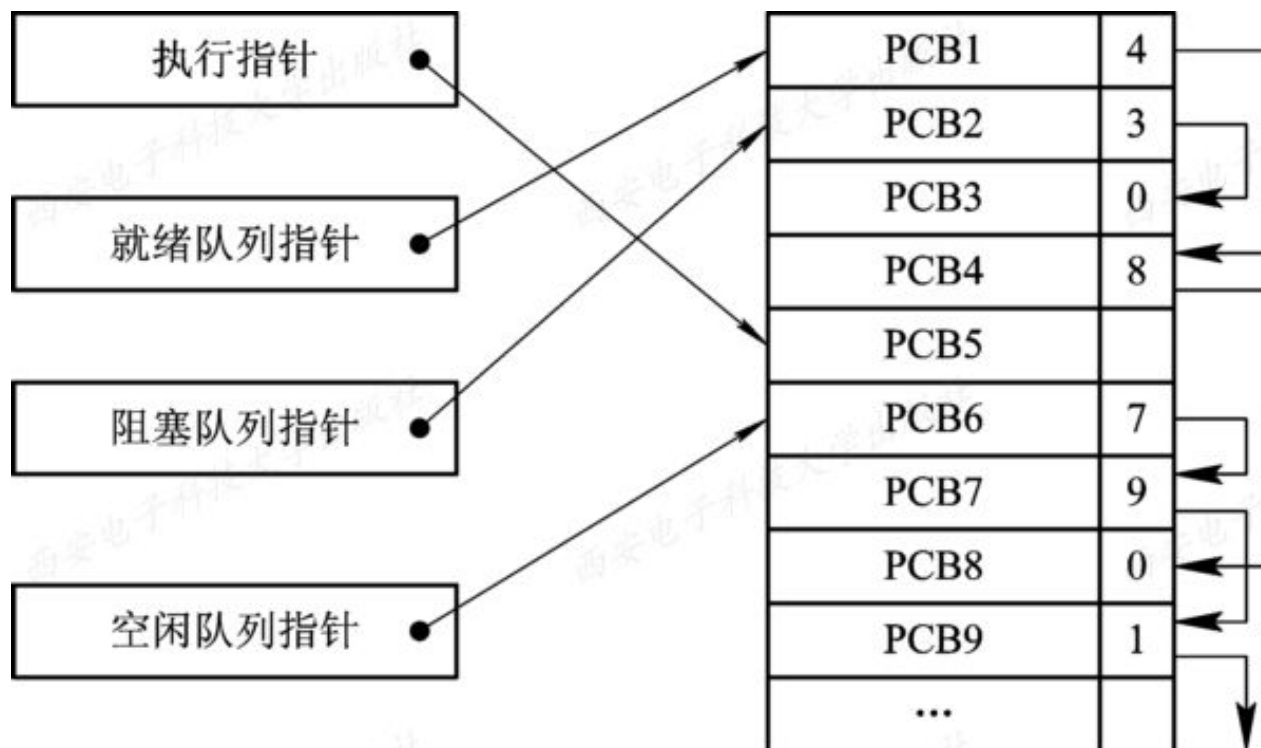




图2-11 PCB链接队列示意图



(3) 索引方式，即系统根据所有进程状态的不同，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。图2-12示出了索引方式的PCB组织。

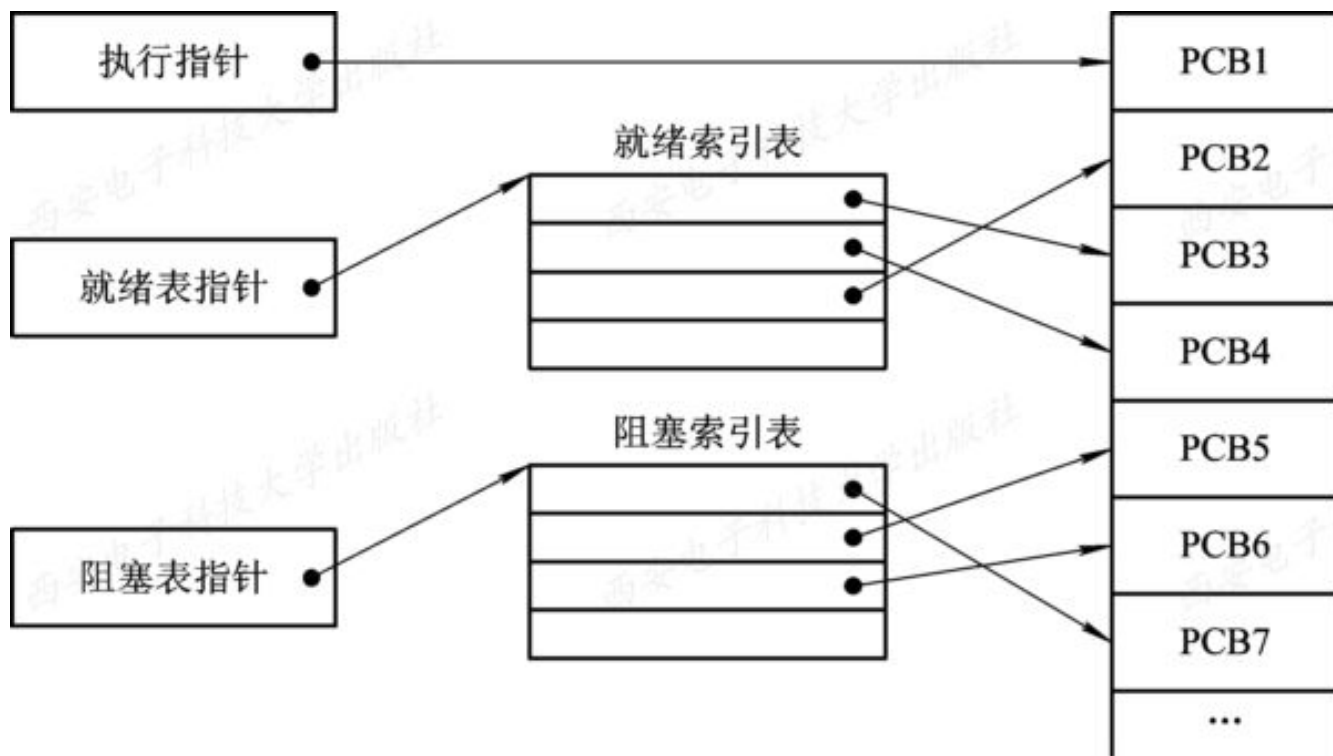


图2-12 按索引方式组织PCB

2.3 进程控制

进程控制的主要任务：

- (1) 创建进程
- (2) 撤销进程
- (3) 实现进程状态的转换

进程控制一般由OS内核来完成，OS的内核通过原语（Primitive）来实现进程控制。使用原语的目的是为了避免造成进程状态的不确定性。原语在执行时不允许被中断。



2.3.1 操作系统内核

现代操作系统一般将OS划分为若干层次，再将OS不同的功能分别设置在不同的层次中。

通常将一些与硬件紧密相关的模块、各种常用设备的驱动程序以及运行频率较高的模块（如时钟管理、进程调度和许多模块所公用的一些基本操作），都安排在紧靠硬件的软件层次中，将它们常驻内存，通常被称为**OS内核**。



通常将处理机的执行状态分成系统态和用户态两种：

① **系统态**(kernel mode, master mode, supervisor mode, privileged mode): 又称为**管态**，也称为**内核态**。它具有较高的特权，能执行一切指令（含特权指令和非特权指令），访问所有的寄存器和存储区，传统的OS都在系统态运行；

② **用户态**(user modes, slave mode, problem state): 又称为**目态**，它是具有较低特权的执行状态，仅能执行规定的指令（仅非特权指令），访问指定的寄存器和存储区。一般情况下，应用程序只能在用户态运行，不能去执行OS指令及访问OS区域，这样可以防止应用程序对OS的破坏。




2.3.1 操作系统内核

1. 支撑功能

(1) 中断处理：中断处理是内核最基本的功能，可以使CPU从用户态切换为系统态，使操作系统获得计算机的控制权，有了中断，才能实现多道程序并发执行。

(2) 时钟管理：内核的一项基本功能，在OS中的许多活动都需要它的支撑。

(3) 原语操作：所谓原语，就是由若干条指令组成的，用于完成一定功能的一个过程。原语在执行过程中不允许被中断，所有动作是一个不可分割的基本单位，要么全做，要么全不做。原语操作用关/开中断来实现。

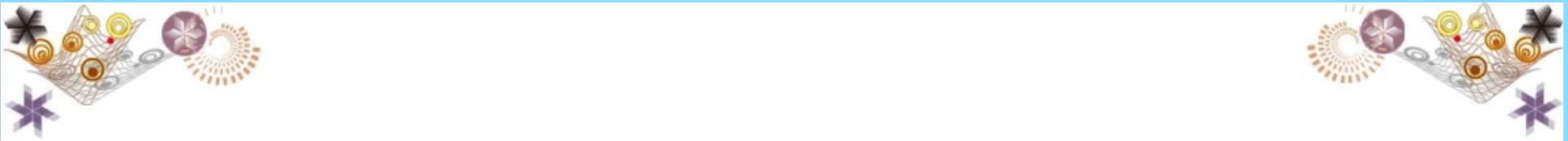


2. 资源管理功能

(1) 进程管理。

(2) 存储器管理。

(3) 设备管理。





2.3.2 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)。

为了标识进程之间的家族关系，在PCB中设置了家族关系表项，以标明自己的父进程及所有的子进程。进程不能拒绝其子进程的继承权。

在Windows中不存在任何进程层次结构的概念。



2. 进程图

为了形象地描述一个进程的家族关系而引入了进程图 (Process Graph)。所谓进程图就是用于描述进程间关系的一棵有向树，如图2-13所示。

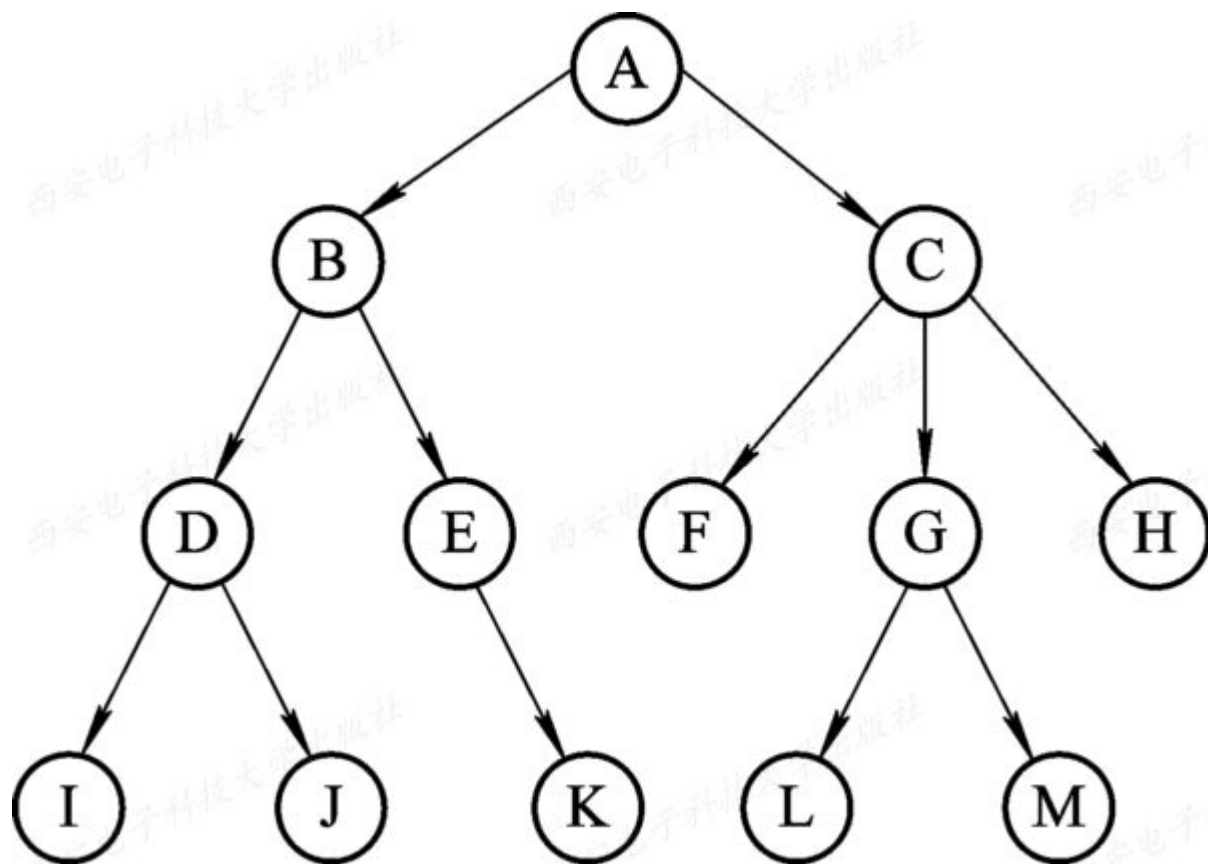


图2-13 进程树



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：分时系统中，用户登录成功，系统会为期建立一个新的进程。

(2) 作业调度：多道批处理系统中，当有新的作业放入内存时，会为其建立一个新的进程。

(3) 提供服务：当用户向操作系统提出某些请求时，会新建一个进程处理该请求。

(4) 应用请求：由用户进程请求创建一个子进程。

进程管理功能的微内核结构

进程管理服务器

如何确定优先级

如何修改优先级

微内核

进程控制（创建等）

进程调度：

设置优先级队列

取出一个进程投入运行

进程通信

进程切换



线程调度

多处理机之间的同步

4. 进程的创建 (Creation of Process)

在系统中每当出现了创建新进程的请求后，OS便调用进程**创建原语Create**按下述步骤创建一个新进程：

- (1) 申请空白PCB，为新进程申请获得唯一的数字标识符，并从PCB集合中索取一个空白PCB。
- (2) 为新进程分配其运行所需的资源，包括各种物理和逻辑资源，如内存、文件、I/O设备和CPU时间等。
- (3) 初始化进程控制块(PCB)。
- (4) 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。



为新进程分配内存资源时，OS必须知道它的大小。

- 对于批处理系统中的进程，由用户提供。
- 为应用程序的进程创建子进程，由父进程提供。
- 对于交互型的系统，用户可以不提供进程的大小而由系统分配一定的空间。
- 如果新进程要共享某个某个已在内存的共享段，必须建立相应的链接。



初始化进程控制块包括：

① 初始化标识信息：内部标识符

父进程标识符

② 初始化处理机状态信息：PC、栈指针；

③ 初始化处理机控制信息：

- 进程的状态：设置为就绪状态或静止就绪状态，
- 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。



2.3.3 进程的终止

1. 引起进程终止(Termination of Process)的事件



- (1) 正常结束
- (2) 异常结束
- (3) 外界干预



(1) 正常结束

用于表示进程已经运行完成的指示：

- 批处理系统中：Halt指令/系统调用，会产生一个中断，通知OS进程已经完成。
- 分时系统中：用户可利用Logs off表示进程运行完毕，会产生一个中断，通知OS进程已运行完成



(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)。



常见的异常事件	解释
越界	访问的存储区超出该进程的区域
保护错	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
非法指令	访问到数据区
特权指令错	用户进程试图去执行一条只允许OS执行的指令
运行超时	进程的执行时间超过了指定的最大值
等待超时	进程等待某事件的时间超过了规定的最大值
算术运算错	进程试图去执行一个被禁止的运算
I/O故障	在I/O过程中发生了错误等



(3) 外界干预

外界干预指进程应外界的请求而终止运行。

①操作员或操作系统干预。例如，发生了死锁。

②父进程请求

③父进程终止

2. 进程的终止过程

如果系统中发生了要求终止进程的某事件，OS便调用**进程终止原语**，按下述过程去终止指定的进程：

(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也都予以终止，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，等待其它程序来搜集信息。





2.3.4 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) 向系统请求共享资源失败。
- (2) 等待某种操作的完成。
- (3) 新数据尚未到达。
- (4) 等待新任务的到达。



2. 进程阻塞过程

正在执行的进程，如果发生了上述某事件，进程便通过调用**阻塞原语block**将自己阻塞。可见，阻塞是进程自身的一种主动行为。



- ①进入block过程后，由于该进程还处于执行状态，所以应先立即停止执行，把进程控制块中的现行状态由“执行”改为阻塞。
- ② 将PCB插入阻塞队列。如果系统中设置了因不同事件而阻塞的多个阻塞队列，则应将本进程插入到具有相同事件的阻塞队列。
- ③ 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，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发生时，比如它所启动的I/O操作已完成，或其所期待的数据已经到达，则由有关进程(比如提供数据的进程)调用**唤醒原语wakeup**，将等待该事件的进程唤醒。

wakeup执行的过程是：

- ① 把被阻塞的进程从等待该事件的阻塞队列中移出；
- 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。



block原语和wakeup原语是一对作用刚好相反的原语。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！！！！**

2.3.5 进程的挂起与激活

1. 进程的挂起

当系统中出现了引起进程挂起的事件时，OS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。

suspend的执行过程是：

- ① 检查状态：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。
- ② 复制PCB：为了方便用户或父进程考查该进程的运行情况，而把该进程的PCB复制到某指定的内存区域；
- ③ 重新调度：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

当系统中发生激活进程的事件时，OS将利用激活原语 active，将指定进程激活。

- ① 激活原语将进程从外存调入内存；
- ② 修改状态：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。
- ③ 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。



2.4 进 程 同 步

进程同步的主要任务是对并发的多个进程在**执行次序**上进行协调，以使进程之间能够有效地共享资源和相互合作，使**执行结果具有可再现性**。



2.4.1 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（进程互斥）：对于临界资源的访问，必须互斥地进行。**进程互斥**指当一个进程访问某临界资源时，另一个想要访问该临界资源的进程必须等待。当前访问临界资源的进程访问结束，释放该资源后，另一个进程才能去访问临界资源。

2) 直接相互制约关系（进程同步）：指为完成某任务而建立的两个或多个进程，这些进程因为需要在某些位置上协调它们的工作次序而产生的制约关系。进程间的直接制约关系就是源于它们之间的**相互合作**。

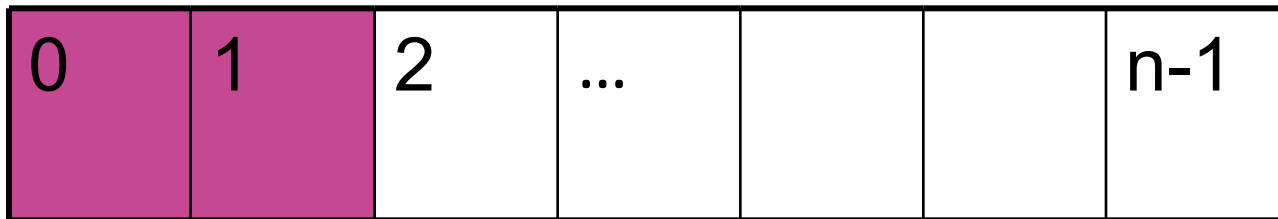


2. 临界资源(Critical Resouce)



一个时间段内只允许一个进程使用的资源称为临界资源。许多硬件资源如打印机、摄像头、磁带机等，都属于临界资源，诸进程间应采取互斥方式，实现对这种资源的共享。此外还有许多变量、数据、内存缓冲区等也都属于临界资源。

2. 临界资源(Critical Resouce)

生产者进程Producer



消费者进程Consumer



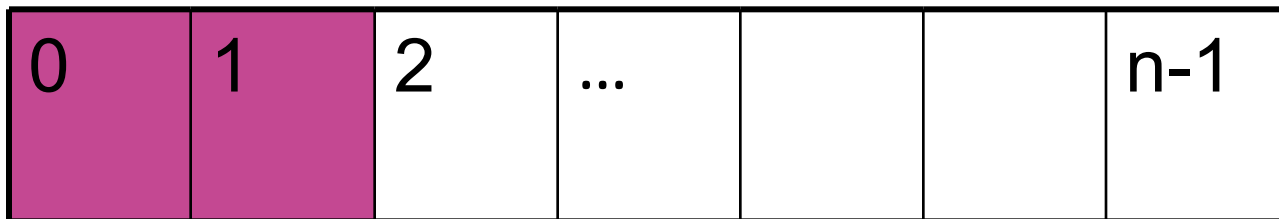
◆ 用**输入指针in**来指示**下一个可投放产品的缓冲区**（循环的），每当生产者进程生产并投放一个产品后，输入指针加1（表示成 $in = (in+1) \% n$ ）；

◆ 用一个**输出指针out**来指示**下一个可从中获取产品的缓冲区**，每当消费者进程取走一个产品后，输出指针加1（表示成 $out = (out+1) \% n$ ）。当 $(in+1) \% n = out$ 时表示缓冲池满；而 $in = out$ 则表示缓冲池空。

◆ 引入了一个**整型变量counter**，其初始值为0。每当生产者进程向缓冲池中投放一个产品后，使counter加1；反之，每当消费者进程从中取走一个产品时，使counter减1。

2. 临界资源(Critical Resouce)

生产者进程Producer

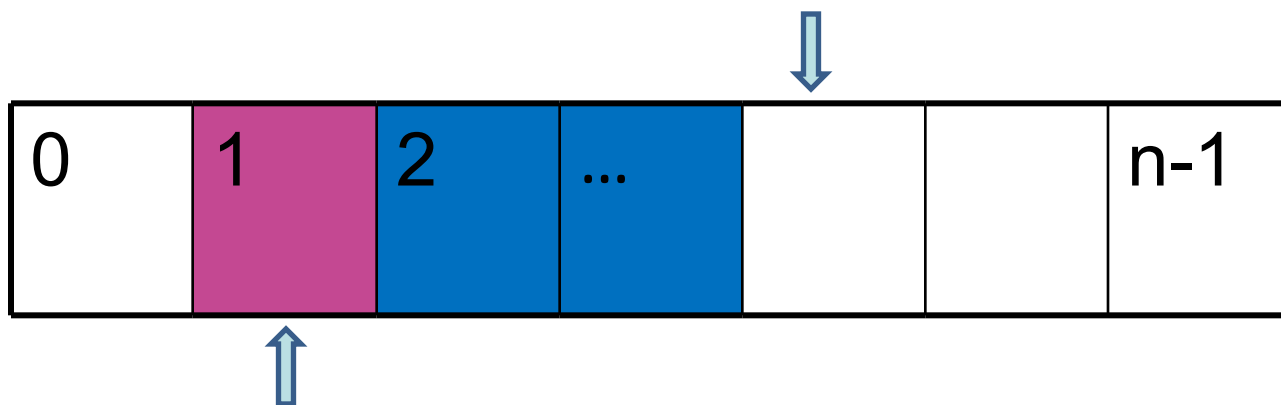


消费者进程Consumer

In: 2 out: 0 counter: 2

2. 临界资源(Critical Resouce)

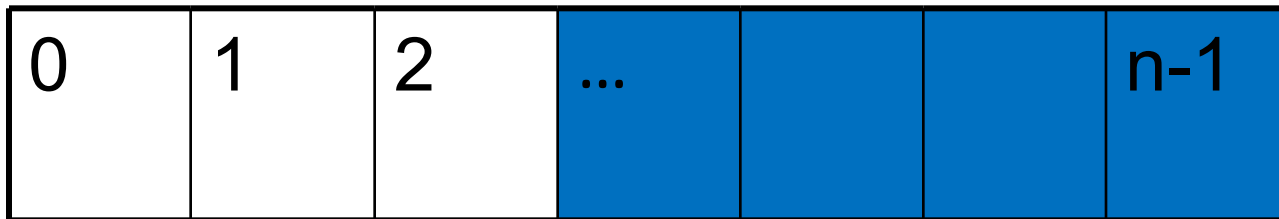
生产者进程Producer



消费者进程Consumer

In: 4 out: 1 counter: 3

生产者进程Producer



消费者进程Consumer

In: 0 out: 3 counter: 4



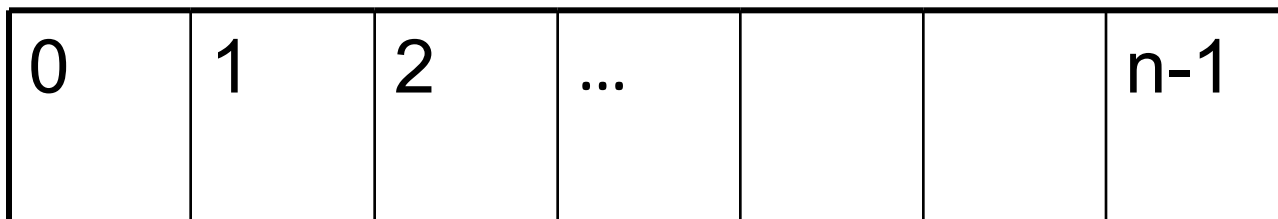
生产者和消费者两进程共享下面的变量：

```
int in = 0, out = 0, count = 0;  
item buffer[n];
```

指针in和out初始化为0，counter的值初始化为0
buffer中暂存产品

初始状态

生产者进程Producer



消费者进程Consumer

初始化: In: 0 out: 0 counter: 0

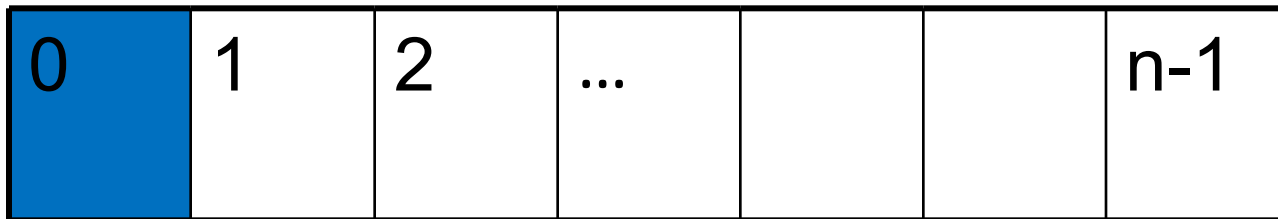
刚生产出来的产品

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

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

要消费的产品

生产者进程Producer



消费者进程Consumer

生产者生产一个产品后: In: 1 out: 0 counter: 0



这两个进程**共享变量counter**。生产者对它做加1操作，消费者对它做减1操作，这两个操作在用机器语言实现时，常可用下面的形式描述：

生产者Producer

```
register1 =counter;  
register1 =register1+1;  
counter =register1;
```

在register1中实现加1操作

消费者Customer

```
register2 =counter;  
register2 =register2-1;  
counter =register2;
```

在register2中实现减1操作



设counter当前值为5，如果按下述顺序执行：

register1:=counter; (register1=5)

register1:=register1+1; (register1=6)

register2:=counter; (register2=5)

register2:=register2-1; (register2=4)

counter:=register1; (counter=6)

counter:=register2; (counter=4)

3. 临界区(critical section)

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)。

在临界区前，负责检查是否可以进入临界区。若可进入，则应设置正在访问临界资源的标志，以阻止其他进程同时进入临界区，这段代码称为进入区(entry section)。

在临界区后，用于将临界区正被访问的标志恢复为未被访问的标志，这部分代码称为退出区(exit section)。

其他代码称为剩余区。



3. 临界区(critical section)

一个访问临界资源的循环进程可描述如下：

```
while(TRUE)
```

```
{
```

 进入区

 临界区

 退出区

 剩余区

```
}
```

4. 同步机制应遵循的规则

所有同步机制都应遵循下述四条准则：

- (1) 空闲让进：临界资源空闲时，应允许一个进程访问。
- (2) 忙则等待：临界资源正在被访问时，其他视图进入临界区的进程必须等待。
- (3) 有限等待：保证在有限时间内能进入自己的临界区，避免“死等”。
- (4) 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，避免“忙等”。



2.4.2 硬件同步机制

虽然可以利用软件方法解决诸进程互斥进入临界区的问题，但有一定难度，并且存在很大的局限性，因而现在已很少采用。相应地，目前许多计算机已提供了一些特殊的硬件指令，允许对一个字中的内容进行检测和修正，或者是对两个字的内容进行交换等。可利用这些特殊的指令来解决临界区问题。

1. 关中断

关中断是实现互斥的最简单的方法之一。在进入锁测试之前关闭中断，直到完成锁测试并上锁之后才能打开中断。

关中断的方法存在许多缺点：

- ① 滥用关中断权力可能导致严重后果；
- ② **关中断**时间过长，会影响系统效率，限制了处理器交叉执行程序的能力；
- ③ **关中断方法也不适用于多CPU 系统**，因为在一个处理器上关中断并不能防止进程在其它处理器上执行相同的临界段代码。

2. 利用Test-and-Set指令实现互斥

这是一种借助一条硬件指令——“测试并建立”指令 TS(Test-and-Set)以实现互斥的方法。在许多计算机中都提供了这种指令。

TS指令的一般性描述如下：

```
//布尔型共享变量lock表示当前临界区是否被加锁
//true表示已加锁， false表示未加锁
bool TS(bool* lock) {
    bool old;
    old = *lock; //old用来存放lock原有的值
    *lock = true; //无论之前是否已加锁，都将lock设为true
    return old;   //返回lock原来的值
}
```

使用TS指令实现互斥的算法逻辑

```
do {  
    ...  
    while TS(&lock);  
    critical section;  
    lock = false;  
    remainder section;  
} while (true);
```

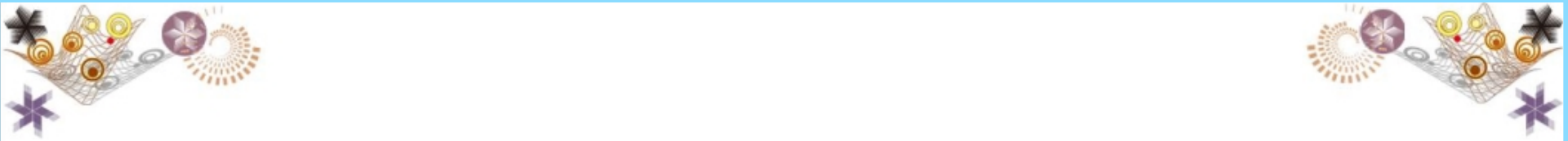
优点：实现简单，无需像软件实现方法那样严格检查是否有逻辑漏洞；适用于多处理机环境。

缺点：**不满足“让权等待”原则**，暂时无法进入临界区的进程会占用CPU并循环执行TS指令，从而导致“忙等”。

3. 利用Swap指令实现进程互斥

该指令称为对换指令，在Intel 80x86中又称为XCHG指令，用于交换两个字的内容。其处理过程描述如下：

```
void swap(bool *a, bool *b) {  
    bool temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```



利用swap指令实现进程互斥的循环进程可描述如下：

```
do {  
    key = true;  
    do {  
        swap(&lock, &key);  
        while (key != false);  
        临界区操作;  
        lock = false;  
        ...  
    } while (true);  
}
```

优点：实现简单，无需像软件实现方法那样严格检查是否有逻辑漏洞；适用于多处理机环境。

缺点：**不满足“让权等待”原则**，暂时无法进入临界区的进程会占用CPU并循环执行swap指令，从而导致“忙等”。

2.4.3 信号量机制

1. 整型信号量

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（来自荷兰语proberen和verhogen）。

//放在进入区

```
wait(S) {  
    while (S <= 0);  
    S--;  
}
```

//放在退出区

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

缺点：未遵循“让权等待”的准则

信号量机制——整型信号量

用一个**整数型的变量**作为信号量，用来**表示系统中某种资源的数量**。

Eg：某计算机系统中有一台打印机...

```
int S = 1; // 初始化整型信号量s，表示当前系统中可用的打印机资源数
```

```
void wait (int S) { //wait 原语，相当于“进入区”  
    while (S <= 0); //如果资源数不够，就一直循环等待  
    S=S-1;          //如果资源数够，则占用一个资源  
}
```

```
void signal (int S) { //signal 原语，相当于“退出区”  
    S=S+1;            //使用完资源后，在退出区释放资源  
}
```

进程P0:

```
...  
wait(S);          //进入区，申请资源  
使用打印机资源... //临界区，访问资源  
signal(S);        //退出区，释放资源  
...
```

进程P1:

```
...  
wait(S);  
使用打印机资源...  
signal(S);  
...
```

.....

进程Pn:

```
...  
wait(S);  
使用打印机资源...  
signal(S);  
...
```

与普通整数变量的区别：
对信号量的操作只有三种，
即 初始化、P操作、V操作

“检查”和“上锁”一气呵成，
避免了并发、异步导致的问题

存在的问题：不满足“让权等待”
原则，会发生“忙等”

2. 记录型信号量

记录型信号量机制是一种不存在“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。

为此，在信号量机制中，除了需要一个用于代表资源数目的整型变量value外，还应**增加一个进程链表指针list**，用于链接上述的所有等待进程。

记录型信号量是由于它采用了记录型的数据结构而得名的。其中的**value的初值表示某类资源的数目**，因而记录型信号量又称资源信号量。（初值为1时称为互斥信号量）。

记录型信号量所包含的两个数据项可描述为：

```
typedef struct {  
    int value; //剩余资源数  
    struct process_control_block *list; //等待队列  
} semaphore;
```

wait(S)和signal(S)操作描述:

//某进程需要资源时, 通过wait原语申请

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

//进程使用完资源后, 通过signal原语释放

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

如果剩余资源数不够, 使用block原语使进程从运行态进入阻塞态, 并挂到信号量S的等待队列(即阻塞队列)中

释放资源后, 若还有别的进程在等待这种资源, 则使用wakeup原语唤醒等待队列的一个进程, 该进程从阻塞态变为就绪态

3. AND型信号量

整型信号量和记录型信号量之间只共享一个临界资源而言的，而AND型信号量是**针对共享多个临界资源**而言的。

例：当共享多个临界资源时使用整型信号量机制会造成死锁。

临界资源D的互斥信号量Dmutex，初值为1；

临界资源E的互斥信号量Emutex，初值为1；

先请求D，
再请求E

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

process B:
wait(Emutex); □
wait(Dmutex); □

先请求E，
再请求D



若进程A和B按下述次序交替执行wait操作：



process A: wait(Dmutex); 于是Dmutex=0

process B: wait(Emutex); 于是Emutex=0

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



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

死锁



AND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，也不分配给他。亦即，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。**

由死锁理论可知，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（同时wait操作）



```
Swait (S1, S2, ..., Sn)
```

```
{
```

```
    while (true)
```

```
    {
```

```
        if (S1>=1 && ... && Sn>=1)
```

```
            for (i=1; i<=n; i++) Si--;
```

```
            break;
```

```
        }
```

```
    else {
```

```
        /*place the process in the waiting queue  
        associated with the first Si found with Si < 1,  
        and set the program count of this process  
        to the beginning of Swait operation*/
```

```
    }
```

```
}
```

```
}
```

以便唤醒后再继续请求这些资源


```
Ssignal(S1, S2, ..., Sn) {  
    while (true) {  
        for (i=1; i<=n; i++) {  
            Si++;  
            /*Remove all the process waiting in the queue  
              associated with Si into the ready queue.  */  
        }  
    }  
}
```

以便唤醒后再继续请求这些资源

4. 信号量集

记录型信号量机制中，wait(S)和signal(S)操作仅能对信号量进行加1和减1的操作，即每次只能获得或释放1个临界资源，这是低效的。

信号量集机制的作用：

- (1) 一次可以获得或释放多个临界资源；
- (2) 每次分配之前，先测试这种临界资源的数量，看是否大于某个下限值。当资源数量低于某一下限值时，不予分配。

在Swait和Ssignal的描述中:

S为信号量集 **d**为需求值 **t**为下限值

```
Swait (S1, t1, d1, ..., Sn, tn, dn)
{
    while (true)
    {
        if (S1>t1 && ... && Sn>tn)
            for (i=1; i<=n; i++) Si-=di;
            break;
        }
        else {
            /*place the process in the waiting queue
            associated with the first Si found with Si < ti,
            and set the program count of this process
            to the beginning of Swait operation*/
        }
    }
}
```

在Swait和Ssignal的描述中:

S为信号量集 **d**为需求值 **t**为下限值

```
Ssignal(S1, d1, ..., Sn, dn) {  
    while (true) {  
        for (i=1; i<=n; i++) {  
            Si+=di;  
            /*Remove all the process waiting in the queue  
            associated with Si into the ready queue. */  
        }  
    }  
}
```



一般“信号量集”的几种特殊情况：

(1) $\text{Swait}(S, d, d)$ 。在信号量集中只有一个信号量 S （仅有一种临界资源），允许每次申请 d 个资源，当现有资源数少于 d 时，不予分配。

(2) $\text{Swait}(S, 1, 1)$ 。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) $\text{Swait}(S, 1, 0)$ 。这是一种很特殊且很有用的信号量操作。当 $S \geq 1$ 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。



2.4.4 信号量的应用

1. 利用信号量实现进程互斥

为使多个进程能互斥地访问某临界资源，只需为该资源设置一互斥信号量mutex，并设其初始值为1，然后将各进程访问该资源的临界区CS置于wait(mutex)和signal(mutex)操作之间即可。

代码描述:

```
semaphore mutex = 1;  
PA() {  
    while (1) {  
        wait(mutex);  
        //临界区  
        signal(mutex);  
        //剩余区  
    }  
}
```

```
PB() {  
    while (1) {  
        wait(mutex);  
        //临界区  
        signal(mutex);  
        //剩余区  
    }  
}
```

理解：信号量 mutex 表示
“进入临界区的名额”

信号量机制实现进程互斥

1. 分析并发进程的关键活动，划定临界区（如：对临界资源打印机的访问就应放在临界区）
2. 设置互斥信号量 mutex，初值为 1
3. 在进入区 P(mutex)——申请资源
4. 在退出区 V(mutex)——释放资源

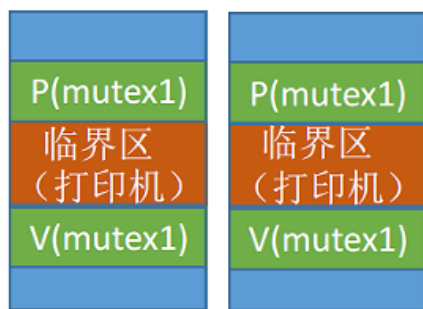
注意：对不同的临界资源需要设置不同的互斥信号量。

P、V操作必须成对出现。缺少 P(mutex) 就不能保证临界资源的互斥访问。缺少 V(mutex) 会导致资源永不被释放，等待进程永不被唤醒。

要会自己定义记录型信号量，但如果题目中没特别说明，可以把信号量的声明简写成这种形式

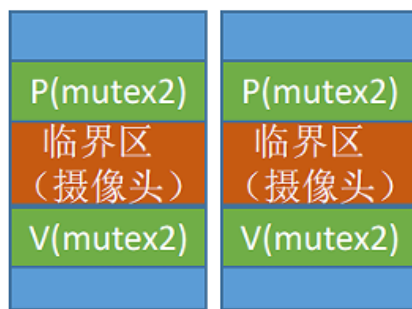
```
/*记录型信号量的定义*/  
typedef struct {  
    int value;           // 剩余资源数  
    struct process *L;   // 等待队列  
} semaphore;
```

```
/*信号量机制实现互斥*/  
semaphore mutex=1; // 初始化信号量  
  
P1(){  
    ...  
    P(mutex);        // 使用临界资源前需要加锁  
    临界区代码段...  
    V(mutex);        // 使用临界资源后需要解锁  
    ...  
}  
  
P2(){  
    ...  
    P(mutex);  
    临界区代码段...  
    V(mutex);  
    ...  
}
```



P1进程

P2进程



P3进程

P4进程

2. 利用信号量实现前趋关系(同步)

设有两个并发执行的进程 P_1 和 P_2 。 P_1 中有语句 S_1 ； P_2 中有语句 S_2 。我们希望在 S_1 执行后再执行 S_2 。为实现这种前趋关系，只需使进程 P_1 和 P_2 共享一个公用信号量 S ，并赋予其初值为0，将 $\text{signal}(S)$ 操作放在语句 S_1 后面，而在 S_2 语句前面插入 $\text{wait}(S)$ 操作，即

在进程 P_1 中，用

S_1 ;

$\text{signal}(S)$;

在进程 P_2 中，用

$\text{wait}(S)$;

S_2 ;

信号量机制实现进程同步

用信号量实现进程同步：

1. 分析什么地方需要实现“同步关系”，即必须保证“一前一后”执行的两个操作（或两句代码）
2. 设置同步信号量 S ，初始为 0
3. 在“前操作”之后执行 $V(S)$
4. 在“后操作”之前执行 $P(S)$

技巧口诀：前V后P

理解：信号量 S 代表“某种资源”，刚开始是没有这种资源的。P2 需要使用这种资源，而又只能由 P1 产生这种资源

/*信号量机制实现同步*/

semaphore $S=0$; // 初始化同步信号量，初始值为 0

```
P1(){  
    代码1;  
    代码2;  
    V(S);  
    代码3;  
}
```

释放资源

```
P2(){  
    P(S);  
    代码4;  
    代码5;  
    代码6;  
}
```

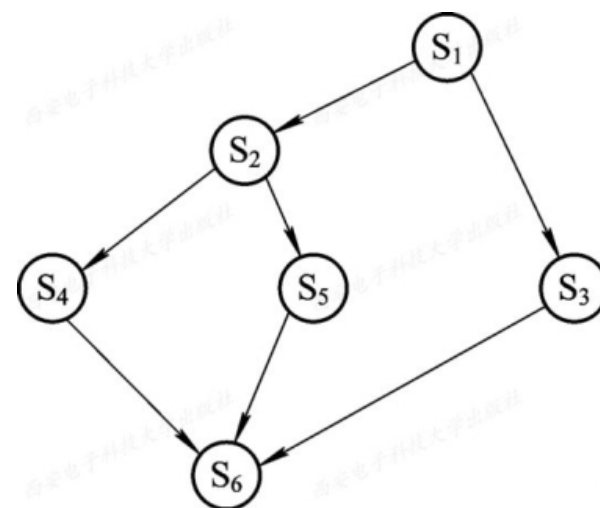
保证了 代码4 一定是在 代码2 之后执行

若先执行到 $V(S)$ 操作，则 $S++$ 后 $S=1$ 。之后当执行到 $P(S)$ 操作时，由于 $S=1$ ，表示有可用资源，会执行 $S--$ ， S 的值变回 0，P2 进程不会执行 block 原语，而是继续往下执行代码4。

若先执行到 $P(S)$ 操作，由于 $S=0$ ， $S--$ 后 $S=-1$ ，表示此时没有可用资源，因此P操作中会执行 block 原语，主动请求阻塞。之后当执行完代码2，继而执行 $V(S)$ 操作， $S++$ ，使 S 变回 0，由于此时有进程在该信号量对应的阻塞队列中，因此会在 V 操作中执行 wakeup 原语，唤醒 P2 进程。这样 P2 就可以继续执行代码4了

```
p1 () {S1; signal (a) ; signal (b) ;}  
p2 () {wait (a) ; S2; signal (c) ; signal (d) ;}  
p3 () {wait (b) ; S3; signal (e) ;}  
p4 () {wait (c) ; S4; signal (f) ;}  
p5 () {wait (d) ; S5; signal (g) ;}  
p6 () {wait (e) ; wait (f) ; wait (g) ; S6;}
```

```
main () {  
    semaphore a, b, c, d, e, f, g;  
    a.value = b.value = c.value = 0;  
    d.value = e.value = 0;  
    f.value = g.value = 0;  
    cobegin  
        p1 () ; p2 () ; p3 () ; p4 () ; p5 () ; p6 () ;  
    coend  
}
```



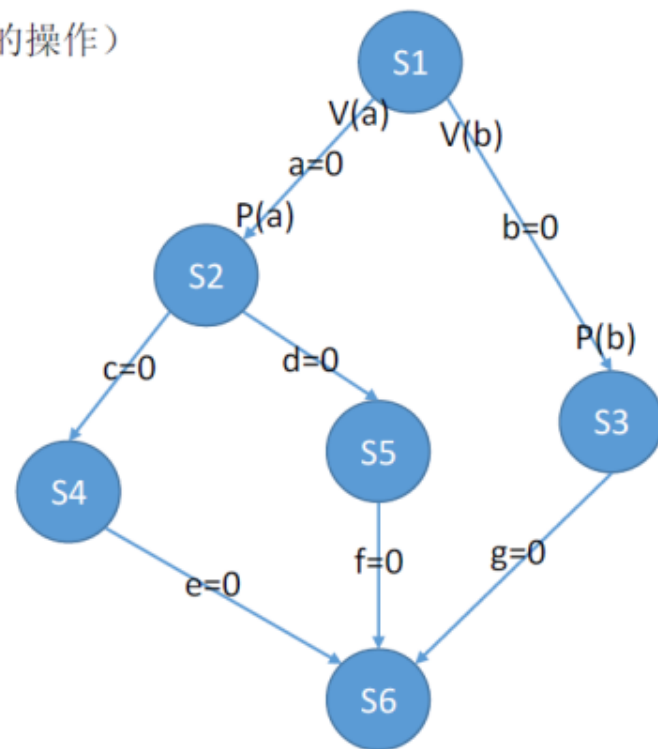
信号量机制实现前驱关系

进程 P1 中有句代码 S1，P2 中有句代码 S2，P3 中有句代码 S3 P6 中有句代码 S6。这些代码要求按如下前驱图所示的顺序来执行：

其实每一对前驱关系都是一个进程同步问题（需要保证一前一后的操作）
因此，

1. 要为每一对前驱关系各设置一个同步信号量
2. 在“前操作”之后对相应的同步信号量执行 V 操作
3. 在“后操作”之前对相应的同步信号量执行 P 操作

<pre>P1() { ... S1; V(a); V(b); ... }</pre>	<pre>P2() { ... P(a); S2; V(c); V(d); ... }</pre>	<pre>P3() { ... P(b); S3; V(g); ... }</pre>	<pre>P4() { ... P(c); S4; V(e); ... }</pre>	<pre>P5() { ... P(d); S5; V(f); ... }</pre>	<pre>P6() { ... P(e); P(f); P(g); S6; ... }</pre>
---	---	---	---	---	---





2.4.5 管程机制

1. 管程(Monitors)的定义

信号量机制存在的问题：**编写程序困难、易出错。**

1973年，Brinch Hansan首次在Pascal语言中引入了管程成分。

利用共享数据结构抽象地表示系统中的共享资源，并且将该共享数据结构实施的特定操作定义为一组过程。进程对共享资源的申请、释放和其它操作必须通过这组过程，间接地对共享数据结构实现操作。对于请求访问共享资源的诸多并发过程，可以根据资源的情况接受或阻塞，确保每次仅有一个进程进入管程，执行这组过程，使用共享资源，达到对共享资源所有访问的统一管理，有效地实现进程互斥。



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。

管程的特征：

- ① 局部于管程的数据只能被局部于管理的**过程**所访问；
- ② 一个进程只有通过调用管程内的过程才能进入管程访问共享数据；
- ③ **每次**仅允许一个进程在管程内执行某个内部过程。

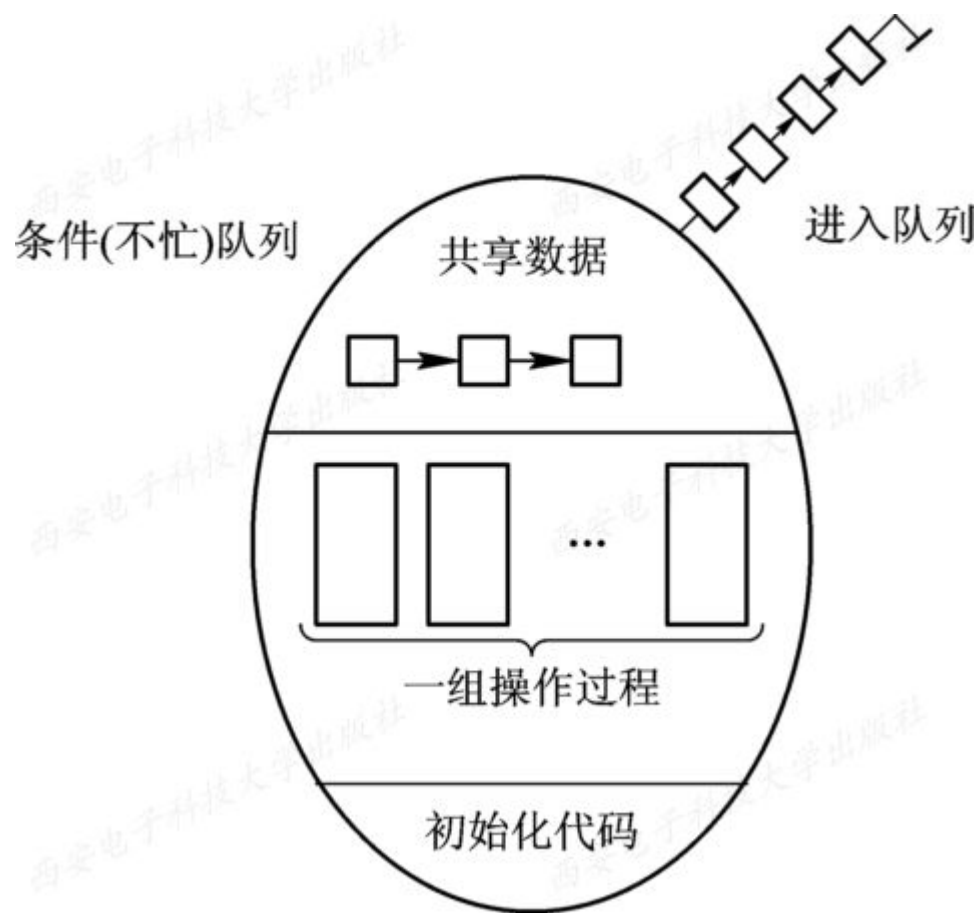


图2-15 管程的示意图

//管程的语法描述

```
Monitor monitor_name {                               /*管程名*/
    share variable declarations;                       /*共享变量说明*/
    cond declarations;                                /*条件变量说明*/
    public:                                           /*能被进程调用的过程*/
        void P1 (.....)                             /*对数据结构操作的过程*/
        {.....}
        void P2 (.....)
        {.....}
        .....
        void (.....)
        {.....}
        .....
        {                                             /*管程主体*/
            initialization code;                     /*初始化代码*/
            .....
        }
    }
```





从语言的角度看，管程的主要特性有：

- ① 模块化，即管程是一个基本程序单位，可以单独编译；
- ② 抽象数据类型，指管程中不仅有数据，而且有对数据的操作；
- ③ 信息掩蔽，指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。

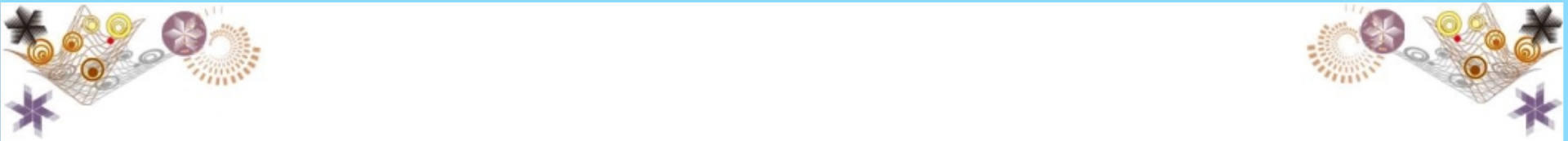


当一个进程进入管程后被阻塞，直到阻塞的原因消除时，在此期间，如果该进程不释放管程，那么其他进程无法进入管程。为此引入了条件变量condition。

通常，一个进程被阻塞或挂起的条件有多个，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。

条件变量的形式： **condition x, y;**

对条件变量只能进行两种操作： **wait**和**signal**。





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可以表示为**x.wait**和**x.signal**。

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x条件的等待队列上，并释放管程，直到x条件变化。此时其他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，继续执行原进程，而不产生任何结果。

//条件变量使用举例

```
monitor demo {  
    共享数据结构 S;  
    condition x;           //定义一个条件变量x  
    init_code() { ... }  
    take_away() {  
        if (S <= 0) x.wait(); //资源不够, 在条件变量x上阻塞等待  
        资源足够, 分配资源, 做一系列相应处理;  
    }  
    give_back() {  
        归还资源, 做一系列处理;  
        if (有进程在等待) x.signal(); //唤醒一个阻塞进程  
    }  
}
```



如果有进程Q因x条件处于阻塞状态，当正在调用管程的进程P执行了x.signal操作，进程Q被重新启动，此时两个进程P和Q，如何确定哪个执行哪个等待，可采用下属两种方式之一处理：

- ①P等待，直至Q离开管程或等待另一条件；
- ②Q等待，直至P离开管程或等待另一条件。

条件变量和信号量的比较

相似点：条件变量的wait/signal操作类似于信号量的P/V操作，可以实现进程的阻塞/唤醒。

不同点：条件变量是“没有值”的，仅实现了“排队等待”功能；而信号量是“有值”的，信号量的值反映了剩余资源数，在管程中，剩余资源数用共享数据结构记录。



2.5 经典进程的同步问题

2.5.1 生产者-消费者问题

2.5.2 哲学家进餐问题

2.5.3 读者-写者问题



2.5.1 生产者-消费者问题

问题描述：有一群生产者进程在不断生产产品，而另一群消费者进程在不断消费生产出来的产品。为了解决生产和消费速度不匹配的矛盾，在生产者和消费者进程之间设置了一个具有 n 个缓冲区的缓冲池；生产者进程不断将它生产出来的产品投放到缓冲区中，每次投放的产品占满一个缓冲区，消费者进程不断从缓冲区中取走产品去消费，一次取走一个缓冲区的产品。生产者和消费者进程必须保持同步，即不允许消费者进程到一个空缓冲池中取产品，也不允许生产者进程将产品投放到一个已装满产品的缓冲池。

1. 利用记录型信号量解决生产者-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，可利用：

(1) 互斥信号量`mutex`实现诸进程对缓冲池的互斥使用，其初值为1；

(2) 资源信号量`empty`表示缓冲池中空缓冲区的数量，其初值为 n ；

(3) 资源信号量`full`表示满缓冲区的数量，其初值为0。


假定这些生产者和消费者相互等效，只要缓冲池未滿，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;
item buffer[n];
semaphore mutex = 1, empty = n, full = 0;
```

```
void producer() {
    do {
        producer an item nextp;
        ...
        wait(empty);
        wait(mutex);
        buffer[in] = nextp;
        in = (in + 1) % n;
        signal(mutex);
        signal(full);
    } while(true)
}
```

```
void main() {
    cobegin
        producer(); consumer();
    coend
}
```



```
void consumer() {
    do {
        wait(full);
        wait(mutex);
        nextc = buffer[out];
        out = (out + 1) % n;
        signal(mutex);
        signal(empty);
        consumer the item in nextc;
        ...
    } while(true);
}
```



思考：能否改变相邻P、V操作的顺序？



```
void producer() {  
    do {  
        producer an item nextp;  
        ...  
        wait(mutex);  
        wait(empty);  
        buffer[in] = nextp;  
        in = (in + 1) % n;  
        signal(mutex);  
        signal(full);  
    } while(true)  
}
```

```
void consumer() {  
    do {  
        wait(mutex);  
        wait(full);  
        nextc = buffer[out];  
        out = (out + 1) % n;  
        signal(mutex);  
        signal(empty);  
        consumer the item in nextc;  
        ...  
    } while(true);  
}
```

结论：实现互斥的P操作一定要在实现同步的P操作之后，否则可能会产生死锁；V操作不会导致进程阻塞，因此两个V操作顺序可以交换。

2. 利用AND信号量解决生产者-消费者问题

对于生产者-消费者问题，也可利用AND信号量来解决，即用Swait(empty, mutex)来代替wait(empty)和wait(mutex)；用Ssignal(mutex, full)来代替signal(mutex)和signal(full)；用Swait(full, mutex)代替wait(full)和wait(mutex)，以及用Ssignal(mutex, empty)代替Signal(mutex)和Signal(empty)。



```
int in = 0; out = 0;
item buffer[n];
semaphore mutex = 1, empty = n, full = 0;
```

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

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

3. 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(x)过程。

(2) get(x)过程。





对于条件变量notfull和notempty，分别有两个过程cwait和csignal对它们进行操作：

(1) cwait(condition)过程：当管程被一个进程占用时，其他进程调用该过程时阻塞，并挂在条件condition的队列上。

(2) csignal(condition)过程：唤醒在cwait执行后阻塞在条件condition队列上的进程，如果这样的进程不止一个，则选择其中一个实施唤醒操作；如果队列为空，则无操作而返回。





Monitor producerconsumer { //管程定义

item buffer[N];

int in, out;

condition notfull, notempty;

int count;

public:

void put(item x) {

if (count >= N) cwait(notfull);

buffer[in] = x;

in = (in + 1) % N;

count++;

csignal(notempty);

}

void get(item x) {

if (count <= 0) cwait(notempty);

x = buffer[out];

out = (out + 1) % N;



count--;

csignal(notfull);

}

{ in = 0; out = 0; count = 0; }

} PC;





```
void producer() { //生产者
    item x;
    while(true) {
        ...
        producer an item nextp;
        PC.put(x);
    }
}
```

```
void main() {
    cobegin
        producer(); consumer();
    coend
}
```

```
void consumer() { //消费者
    item x;
    while (true) {
        PC.get(x);
        consumer the item in nextc;
        ...
    } while(true);
}
```



2.5.2 哲学家进餐问题

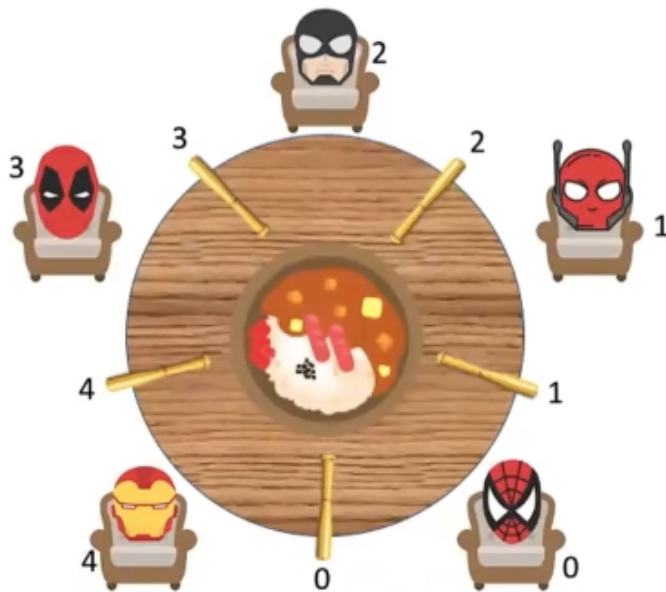
问题描述：有5个哲学家公用一张圆桌，分别坐在周围的5个椅子上，在圆桌上交替放有5个碗和5支筷子；哲学家的生活方式是交替地进行思考和进餐；平时，哲学家进行思考，饥饿时便试图取其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐；进餐完毕后放下筷子继续思考。



2.5.2 哲学家进餐问题

1. 利用记录型信号量解决哲学家进餐问题

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（进程）**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**。



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

```
//第i位哲学家的活动
```

```
do {
```

```
    wait(chopstick[i]);
```

```
    wait(chopstick[i + 1] % 5);
```

```
    ...
```

```
    //eat
```

```
    ...
```

```
    signal(chopstick[i]);
```

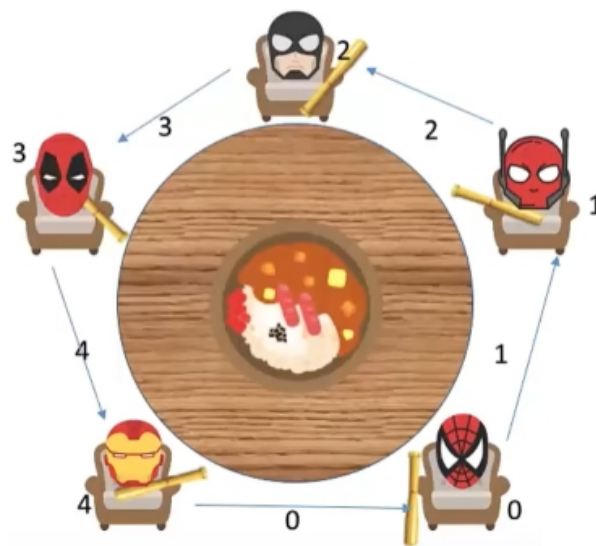
```
    signal(chopstick[i + 1] % 5);
```

```
    ...
```

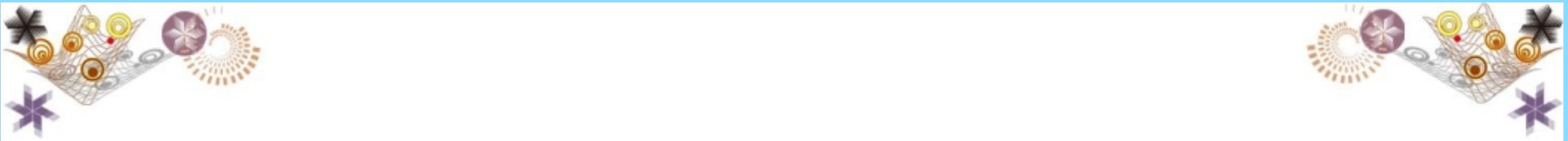
```
    //think
```

```
    ...
```

```
} while(true);
```



可能会引起死锁!



可采取以下三种解决方法：

(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0、1号哲学家竞争1号筷子；2、3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。



2. 利用AND信号量机制解决哲学家进餐问题

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源(筷子)后方能进餐，这在本质上就是前面所介绍的AND同步问题，故用AND信号量机制可获得最简洁的解法。



```
semaphore chopstick[5] = {1, 1, 1, 1, 1};  
//第i位哲学家的活动  
do {  
    ...  
    //think  
    ...  
    Swait(chopstick[i + 1] % 5, chopstick[i]);  
    ...  
    //eat  
    ...  
    Ssignal(chopstick[i+1] % 5, chopstick[i]);  
} while(true);
```







2.5.3 读者-写者问题

有一个数据文件，可以被多个进程共享，各进程共享数据文件的方式不同，有的进程只是从文件中读取信息，这类进程称为“**读者**进程”，有的文件或写信息到文件中，或又读又写，这些进程称为“**写者**进程”。为了不使文件内容混乱，要求各进程在使用文件时必须遵守以下规定：



- (1) 允许多个读者进程同时读取文件。
- (2) 不允许写者进程与其他进程同时访问文件。

1. 利用记录型信号量解决读者-写者问题

为实现Reader与Writer进程间在读或写时的互斥而设置了一个互斥信号量Wmutex。另外，再设置一个整型变量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，表示尚无Reader进程在读时，Reader进程才需要执行Wait(Wmutex)操作。若wait(Wmutex)操作成功，Reader进程便可去读，相应地，做Readcount+1操作。由于Readcount是一个可被多个Reader进程访问的临界资源，因此，也应该为它设置一个互斥信号量rmutex。





```
semaphore rmutex = 1, wmutex = 1;
int readcount = 0;
void reader() {
    do {
        wait(rmutex);
        if (readcount == 0) wait(wmutex);
        readcount++;
        signal(rmutex);
        ...
        perform read operation;
        ...
        wait(rmutex);
        readcount--;
        if (readcount == 0) signal(wmutex);
        signal(rmutex);
    } while(true);
}
```



```
void Writer() {  
    do {  
        wait(wmutex) ;  
        perform write operation ;  
        signal(wmutex) ;  
    } while (true) ;  
}  
void main() {  
    cobegin  
        Reader() ; Writer() ;  
    coend  
}
```

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 $\text{wait}(L, 1, 1)$ 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 $\text{wait}(L, 1, 1)$ 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 $\text{wait}(L, 1, 1)$ 操作失败而阻塞。



```
int RN;  
semaphore L = RN, mx = 1;  
  
void Reader() {  
    do {  
        Swait(L, 1, 1);  
        Swait(mx, 1, 0);  
        ...  
        perform read operation;  
        ...  
        Ssignal(L, 1);  
    } while(true);  
}
```



```
void main() {  
    cobegin  
        Reader(); Writer();  
    coend
```

```
void Writer() {  
    do {  
        Swait(mx, 1, 1; L, RN, 0);  
        perform write operation;  
        Ssignal(mx, 1);  
    } while (true);  
}
```

2.6 进程通信

进程通信是指进程之间的信息交换。由于进程的互斥与同步，需要在进程间交换一定的信息，故不少学者将它们也归为进程通信，但只能把它们称为**低级进程通信**。我们以信号量机制为例来说明，它们之所以低级的原因在于：

- ① **效率低**，生产者每次只能向缓冲池投放一个产品(消息)，消费者每次只能从缓冲区中取得一个消息；
- ② **通信**对用户不透明，OS只为进程之间的通信提供了共享存储器。



在进程之间要传送大量数据时，应当利用OS提供的高级通信工具，该工具最主要的特点是：

(1) 使用方便。OS隐藏了实现进程通信的具体细节，向用户提供了一组用于实现高级通信的命令(原语)，用户可方便地直接利用它实现进程之间的通信。或者说，通信过程对用户是透明的。这样就大大减少了通信程序编制上的复杂性。

(2) 高效地传送大量数据。用户可直接利用高级通信命令(原语)高效地传送大量的数据。



2.6.1 进程通信的类型

1. 共享存储器系统 (Shared-Memory System)

在共享存储器系统中，相互通信的进程共享某些数据结构或共享存储区，进程之间能够通过这些空间进行通信。据此，又可把它们分成以下两种类型：

- (1) 基于共享数据结构的通信方式。
- (2) 基于共享存储区的通信方式。



(1)基于共享数据结构的通信方式

例如，有界缓冲区。对公共数据结构（信号量、缓冲池等）的设置及对进程间同步的处理，都是程序员的职责，程序员负担较重。

特点：只适用于传输少量数据。



(2)基于共享存储区的通信方式

OS中划分出一块共享存储区，

- ① 进程在通信前，先向系统申请获得共享存储区中的一个分区，并指定该分区的关键字；
- ② 若系统已经给其他进程分配了这样的分区，则将该分区的描述符返回给申请者；
- ③ 由申请者把获得的共享存储分区连接到本进程上；
- ④ 此后，便可像读、写普通存储器一样地读、写该公用存储分区。

2. 管道(pipe)通信系统

“管道”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是指，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。

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，只有确定对方已存在时才能进行通信。



3. 消息传递系统(Message passing system)

在该机制中，进程不必借助任何共享存储区或数据结构，而是以**格式化的消息 (message)**为单位，将通信的数据封装在消息中，并利用操作系统提供的一组**通信命令(原语)**，在进程间进行消息传递，完成进程间的数据交换。

基于消息传递系统的通信方式属于高级通信方式，因其实现方式的不同，可进一步分成两类：

(1) 直接通信方式

(2) 间接通信方式



4. 客户机-服务器系统(Client-Server system)

网络环境的各种应用领域下的主流通信实现机制

1) 套接字(Socket)

主要是为了解决多对进程同时通信时端口和物理线路的多路复用问题。随着计算机网络技术的发展以及UNIX 操作系统的广泛使用，套接字已逐渐成为最流行的网络通信程序接口之一。

套接字就是一个通信标识类型的数据结构，包含了通信目的的地址、通信使用的端口号、通信网络的传输层协议、进程所在的网络地址，以及针对客户或服务器程序提供的不同系统调用（或API函数）等，是进程通信和网络通信的基本构件。



套接字包括两类：

1. 基于文件型：通信进程都运行在同一台机器的环境中，套接字是基于本地文件系统支持的，一个套接字关联到一个特殊的文件，通信双方通过对这个特殊文件的读写实现通信。
2. 基于网络型：该类型通常采用的是非对称方式通信，即发送者需要提供接受者命名。通信双方的进程运行在不同主机的网络环境下，被分配了一对套接字，一个属于接收进程（或服务器端），一个属于发送进程（或客户端）。

套接字的优势：不仅适用于同一台计算机内部的进程通信，也适用于网络环境中不同计算机间的进程通信。

2) 远程过程调用和远程方法调用

远程过程(函数)调用RPC(Remote Procedure Call)，是一个通信协议，用于通过网络连接的系统。该协议允许运行于一台主机(本地)系统上的进程调用另一台主机(远程)系统上的进程，而对程序员表现为常规的过程调用，无需额外地为此编程。如果涉及的软件采用面向对象编程，那么远程过程调用亦可称做远程方法调用。

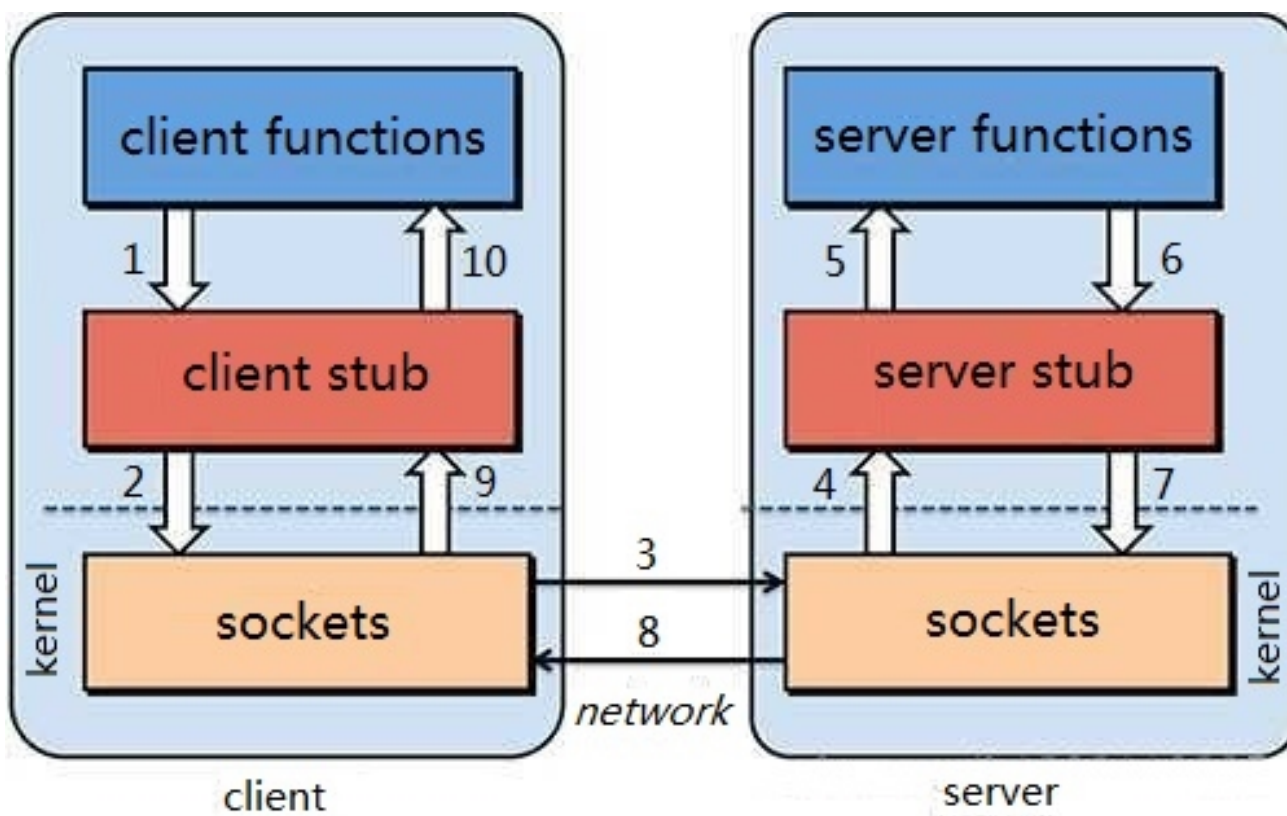


负责远程过程调用的进程有两个：一个本地客户进程，另一个是远程服务器进程，两个进程都被称为网络守护进程，主要负责网络间消息传递，一般情况下，两个进程都处于阻塞状态，等待消息。


为了使PRC看上去与本地过程调用一样（即实现RPC透明性），RPC引入一个存根(stub)的概念：

- (1) 在本地客户端，每个能够独立运行的远程过程都拥有一个客户存根（client stubborn），本地远程调用过程实际是调用该过程关联的存根；
- (2) 在每个远程进程所在的服务器端，其所对应的实际可执行进程也存在一个服务器存根（stub）与其关联。





过程调用的步骤



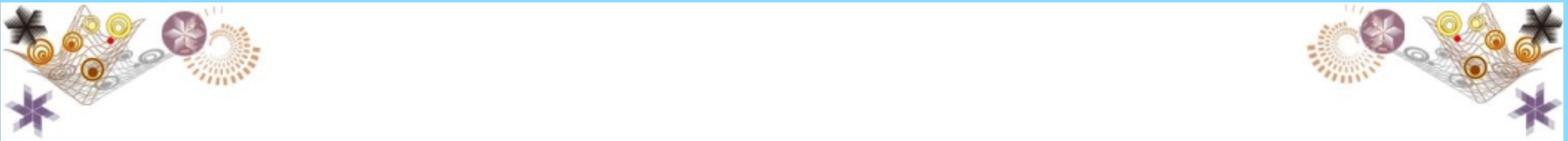
远程过程调用的主要步骤是：

(1) 本地过程调用者以一般方式调用远程过程在本地关联的客户存根，传递相应的参数，然后将控制权转移给**客户存根**；

(2) 客户存根执行，完成包括过程名和调用参数等信息的消息建立，将控制权转移给**本地客户进程**；

(3) 本地客户进程完成与服务器的消息传递，将消息发送到**远程服务器进程**；

(4) 远程服务器进程接收消息后转入执行，并根据其中的远程过程名找到对应的服务器存根，将**消息转给该存根**；



(5) 该服务器存根接到消息后，由阻塞状态转入执行状态，拆开消息从中取出过程调用的参数，然后以一般方式调用服务器上关联的过程；

(6) 在服务器端的远程过程运行完毕后，将结果返回给与之关联的服务器存根；

(7) 该服务器存根获得控制权运行，将结果打包为消息，并将控制权转移给远程服务器进程；

(8) 远程服务器进程将消息发送回客户端；

(9) 本地客户进程接收到消息后，根据其中的过程名将消息存入关联的客户存根，再将控制权转移给客户存根；

(10) 客户存根从消息中取出结果，返回给本地调用者进程，并完成控制权的转移。



2.6.2 消息传递通信的实现方式

1. 直接通信方式：直接消息传递系统
2. 间接通信方式：邮箱通信

1. 直接消息传递系统

1) 直接通信原语

(1) 对称寻址方式

在直接消息传递系统中采用直接通信方式，即发送进程利用OS所提供的发送命令(原语)，直接把消息发送给目标进程。

通常，系统提供下述两条通信命令(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；



例如：Send(P_2, m_1) Receive(P_1, m_1)

(2) 非对称寻址方式

在某些情况下，接收进程可与多个发送进程通信，因此，它不可能事先指定发送进程。例如，用于提供打印服务的进程，它可以接收来自任何一个进程的“打印请求”消息。对于这样的应用，在接收进程接收消息的原语中的源进程参数，是完成通信后的返回值，而发送进程仍需要命名接收进程。发送和接收原语可表示为：

send(P, message); 发送一个消息给接收进程P；

receive(id, message); 接收来自任何进程的消息，id变量可以设置为进行通信的发送方进程或名字



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。该方式可用于办公自动化系统中，为用户提供快速的便笺式通信。

对于需要发送较长消息的用户是不方便的。为此，**可采用变长的消息格式**，即进程所发送消息的长度是可变的。对于变长消息，系统无论在处理方面还是存储方面，都可能会付出更多的开销，但其优点在于方便了用户。

3) 进程的同步方式

在进程之间进行通信时，同样需要有进程同步机制，以使诸进程间能协调通信。不论是发送进程还是接收进程，在完成消息的发送或接收后，都存在两种可能性，即进程或者继续发送(或接收)或者阻塞。

- 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



4) 通信链路

为使在发送进程和接收进程之间能进行通信，必须在两者之间建立一条通信链路。有两种方式建立通信链路：

第一种方式是：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，在链路使用完后拆除链路。

第二种方式是：发送进程无须明确提出建立链路的请求，只需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。



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

在消息传递方式上，最简单的是单向传递。消息的传递也可以是双向的。

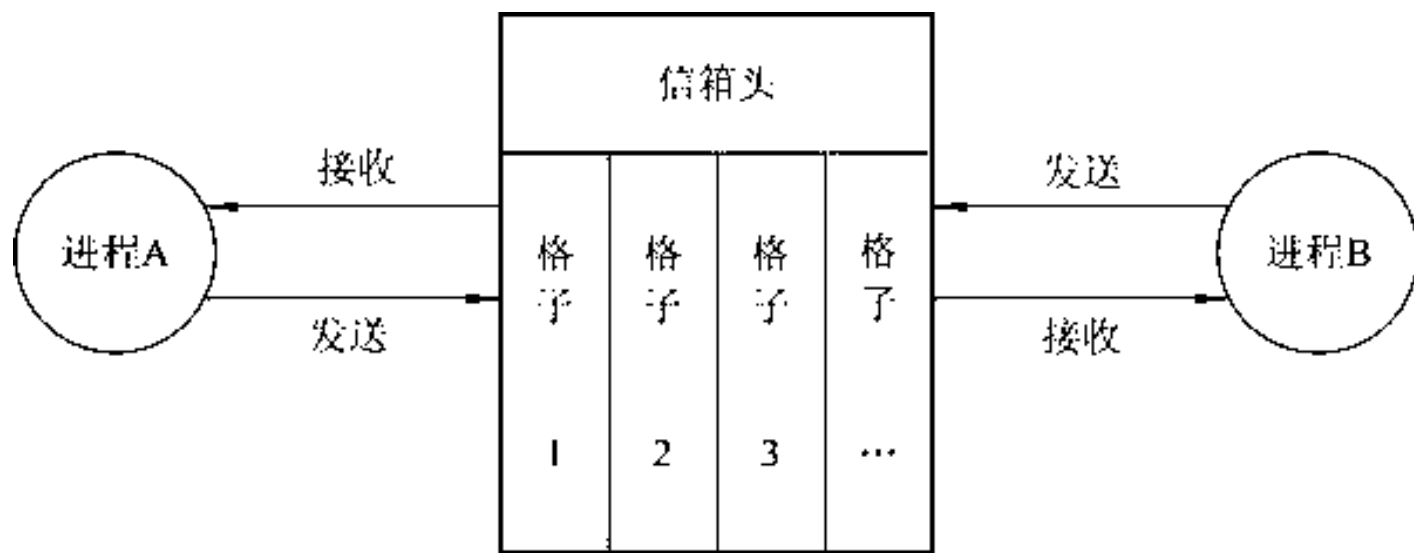


图2-16 双向信箱示意图



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。

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)；对于共享信箱，还应给出**共享者的名字**。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，并利用系统提供的下述通信原语进行通信。

Send(mailbox, message); 将一个消息发送到指定信箱

Receive(mailbox, message); 从指定信箱中接收一个消息



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

用户进程可为自己建立一个新信箱，并作为该进程的一部分。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。这种私用信箱可采用单向通信链路的信箱来实现。当拥有该信箱的进程结束时，信箱也随之消失。



(2) 公用邮箱

它由操作系统创建，并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。显然，公用信箱应采用双向通信链路的信箱来实现。通常，公用信箱在系统运行期间始终存在。

(3) 共享邮箱

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。



在利用信箱通信时，在发送进程和接收进程之间，存在以下四种关系：

- (1) 一对一关系。发送进程和接收进程建立一条两者专用的通信链路，使两者之间的交互不受其他进程的干扰。
- (2) 多对一关系。允许提供服务的进程与多个用户进程之间进行交互，也称为**客户/服务器交互**(client/server interaction)。
- (3) 一对多关系。允许一个发送进程与多个接收进程进行交互，使发送进程可用**广播**方式，向接收者(多个)发送消息。
- (4) 多对多关系。允许建立一个公用信箱，让多个进程都能向信箱中投递消息；也可从信箱中取走属于自己的消息。



2.6.3 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;  
    int size;  
    char *text;  
    struct message_buffer *next;  
}
```

(2) PCB中有关通信的数据项。

```
typedef struct processcontrol_block {  
    ...  
    struct message_buffer *mq;  
    semaphore mutex;  
    semaphore sm;  
    ...  
} 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，如图2-17所示，把待发送的消息正文、发送进程标识符、消息长度等信息填入其中，然后调用发送原语，把消息发送给目标(接收)进程。

发送原语首先根据发送区a中所设置的消息长度a.size来申请一缓冲区i，接着，把发送区a中的信息复制到缓冲区i中。为了能将i挂在接收进程的消息队列mq上，应先获得接收进程的标识符j，然后将i挂在j.mq上。由于该队列属于临界资源，故在执行insert操作的前后都要执行wait和signal操作。

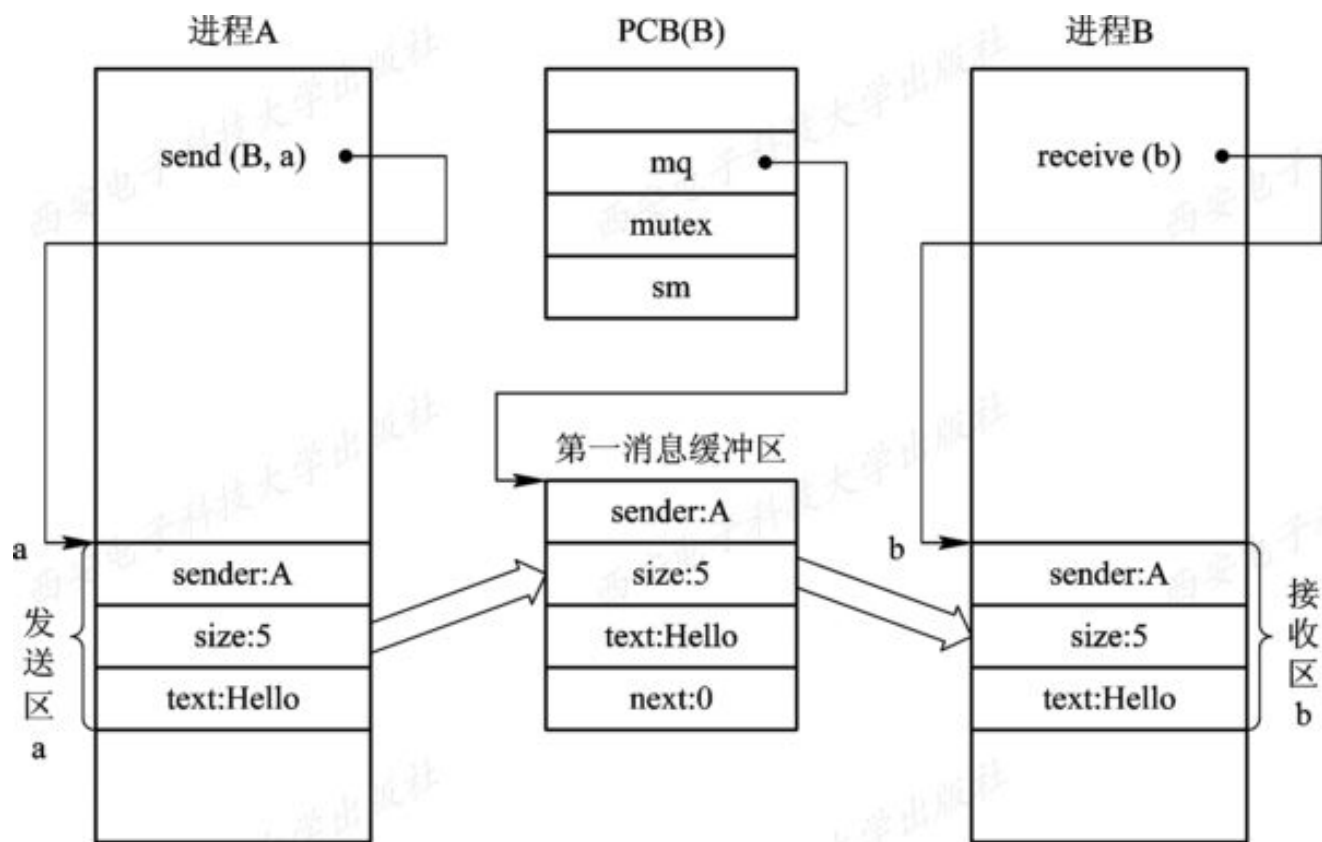


图2-17 消息缓冲通信

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。



2.7 线程(Threads)的基本概念

2.7.1 线程的引入

在OS中引入进程的目的是为了**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。**

在操作系统中再引入线程，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 进程是一个可拥有资源的独立单位。一个进程要能独立运行，它必须拥有一定的资源，包括用于存放程序正文、数据的磁盘和内存地址空间，以及它在运行时所需要的I/O设备、已打开的文件、信号量等；

② 进程同时又是一个可独立调度和分派的基本单位。每个进程在系统中有唯一的PCB，系统可根据其PCB感知进程的存在，也可以根据其PCB中的信息，对进程进行调度，还可将断点信息保存在其PCB中。反之，再利用进程PCB中的信息来恢复进程运行的现场。

2. 程序并发执行所需付出的时空开销

为使程序能并发执行，系统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

进程作为资源的拥有者，在创建、切换、撤销过程中，系统要付出**很大的时空开销**。系统中并发的进程数量不宜过多，进程切换的频率也不宜过高，也就是**进程并发执行程度不能太高**。



3. 线程——作为调度和分派的基本单位

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，已成为近年来设计操作系统时所追求的重要目标。

在OS中引入线程，以线程作为调度和分派的基本单位，可以有效地改善多处理机系统的性能。



2.7.2 线程与进程的比较

1. 调度的基本单位：进程作为独立调度和分发的基本单位。
2. 并发性：在引入线程的OS中，不仅进程之间可以并发执行，而且在同一个进程中的多个线程之间亦可并发执行。
3. 拥有资源：线程本身并不拥有系统资源，仅有一点必不可少的、能保证独立运行的资源，如：TCB、PC、少量寄存器和堆栈等。线程除了拥有自己的少量资源外，还允许多个线程共享该进程所拥有的资源。



2.7.2 线程与进程的比较

4. 独立性：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得多。

5. 系统开销：进程创建、撤销和上下文切换的开销明显大于线程创建、撤销、切换的开销。

6. 支持多处理机系统：对于传统的单线程进程，不管有多少处理机，该进程只能运行在一个处理机上。但是对于多线程进程，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。



2.7.3 线程的状态和线程控制块

1. 线程运行的三个状态

(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。

包括：① 线程标识符；② 一组寄存器的内容；③ 线程运行状态；④ 线程优先级；⑤ 线程专有存储器；⑥ 信号屏蔽；⑦ 堆栈指针。

3. 多线程OS中的进程属性

通常多线程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，但此处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。



2.8 线程的实现

2.8.1 线程的实现方式

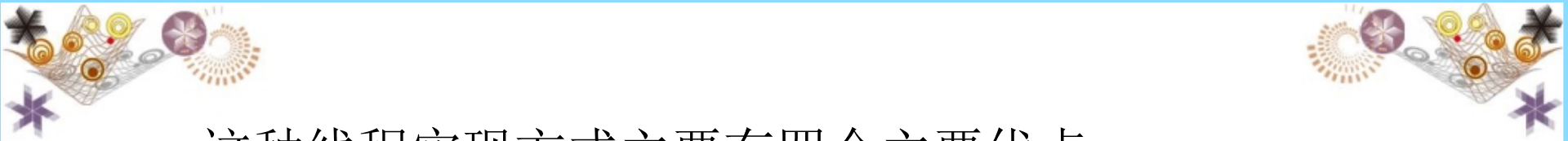
线程已在许多系统中实现，但各系统的实现方式并不完全相同。在有的系统中，特别是一些数据库管理系统，如infomix所实现的是用户级线程；而另一些系统(如Macintosh和OS/2操作系统)所实现的是内核支持线程；还有一些系统如Solaris操作系统，则同时实现了这两种类型的线程。

1. 内核支持线程KST (Kernel Supported Threads)

内核支持线程KST同样也是在内核的支持下运行的，它们的创建、阻塞、撤消和切换等，也都是在内核空间实现的。

为了对内核线程进行控制和管理，在内核空间也为每一个内核线程设置了一个线程控制块，内核根据该控制块而感知某线程的存在，并对其加以控制。

当前大多数OS都支持内核支持线程。



这种线程实现方式主要有四个主要优点：

(1) 在多处理器系统中，内核能够同时调度同一进程中的多个线程并行执行；

(2) 如果进程中的一个线程被阻塞了，内核可以调度该进程中的其它线程占有处理器运行，也可以运行其它进程中的线程；

(3) 内核支持线程具有很小的数据结构和堆栈，线程的切换比较快，切换开销小；

(4) 内核本身也可以采用多线程技术，可以提高系统的执行速度和效率。


内核支持线程的主要缺点：对于用户线程切换开销大，因为要从用户态进入内核态。



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。



使用用户级线程方式有许多优点：

(1) 线程切换不需要转换到内核空间。

(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，因为对于线程管理的代码是属于用户程序的一部分，所有的应用程序都可以对之进行共享。



而用户级线程方式的主要缺点则在于：

(1) **系统调用的阻塞问题**。在基于进程机制的OS中，大多数系统调用将使进程阻塞，因此，当线程执行一个系统调用时，不仅该线程被阻塞，而且，进程内的所有线程会被阻塞。而在内核支持线程方式中，则进程中的其它线程仍然可以运行。

(2) 在单纯的用户级线程实现方式中，多线程应用不能利用多处理机进行多重处理的优点，内核每次分配给一个进程的仅有一个CPU，因此，**进程中仅有一个线程能执行**，在该线程放弃CPU之前，其它线程只能等待。



3. 组合方式

有些OS把用户级线程和内核支持线程两种方式进行组合，提供了组合方式ULT/KST 线程。在组合方式线程系统中，内核支持多个内核支持线程的建立、调度和管理，同时，也允许用户应用程序建立、调度和管理用户级线程。

- (1) 多对一模型：将用户进程映射到一个内核控制线程。
- (2) 一对一模型：将每一个用户级线程映射到一个内核支持线程。
- (3) 多对多模型：将许多用户线程映射到同样数量或更少数量的内核线程上。

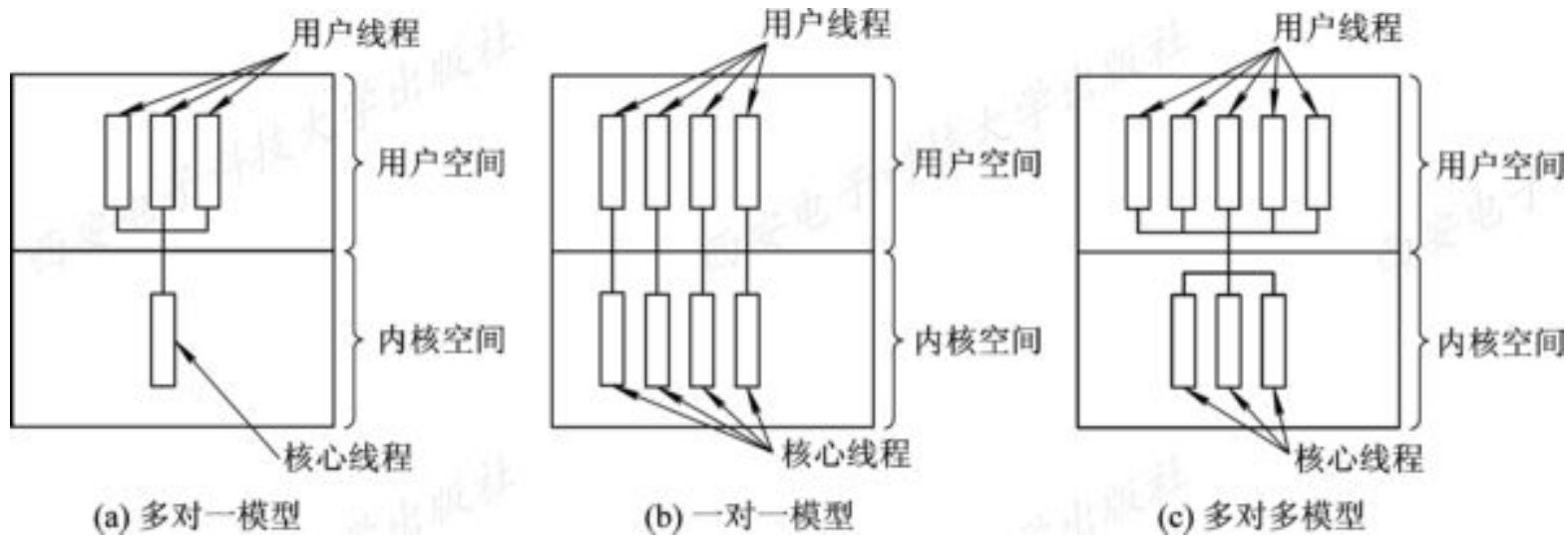


图2-18 多线程模型

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。

PTDA 进程资源

TCB #1

TCB #2

TCB #3



2. 用户级线程的实现

1) 运行时系统(Runtime System)

“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。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**，这样，当一个用户级线程运行时，只要将它连接到一个LWP上，此时它便具有了内核支持线程的所有属性。

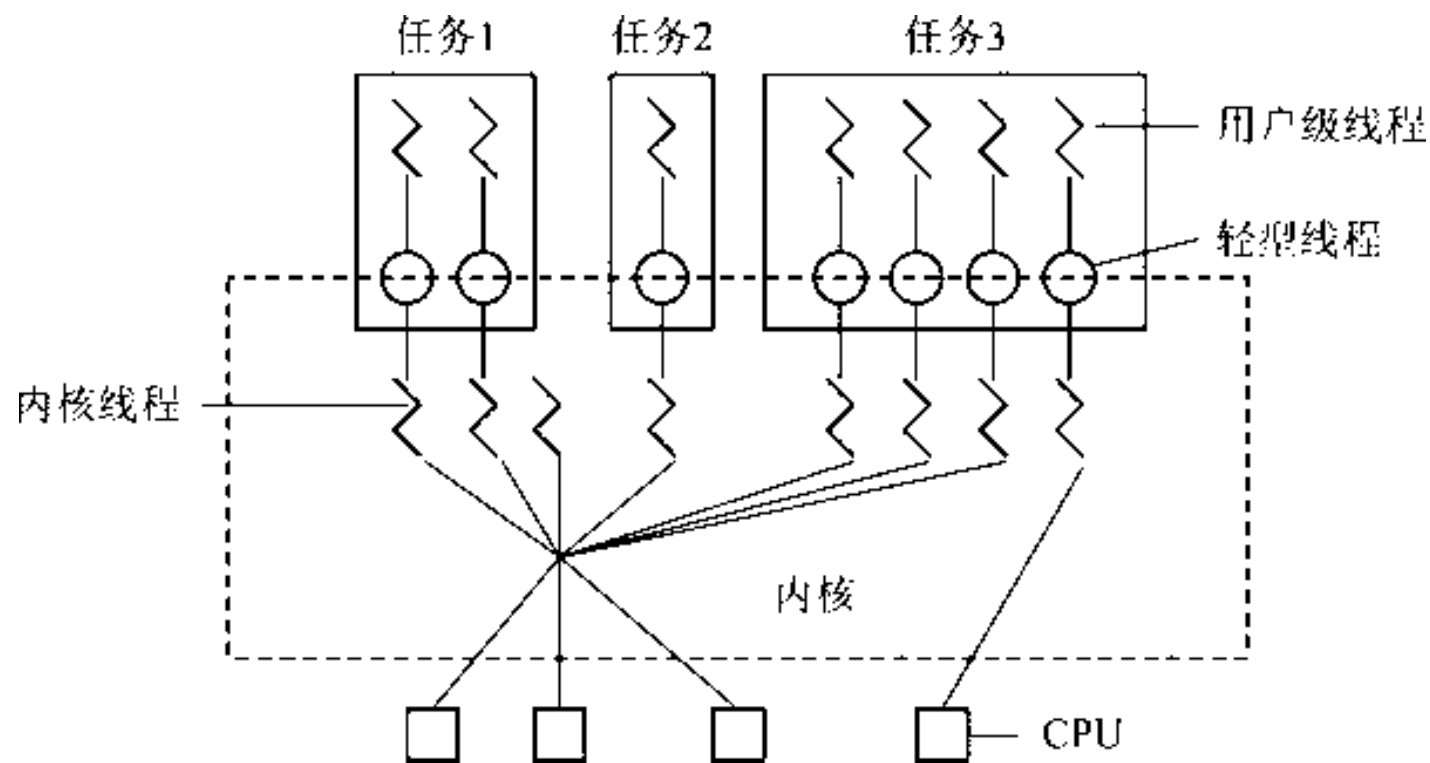


图2-20 利用轻型进程作为中间系统



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。