第二章　进程的描述与控制

BACK

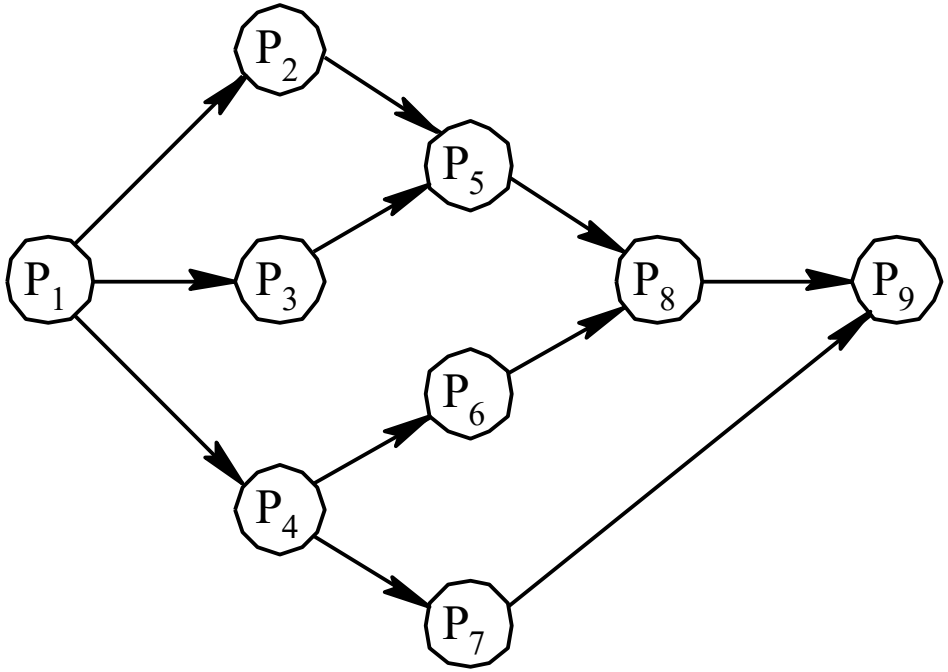
2.1 前驱图和程序执行

2.1.1 前趋图

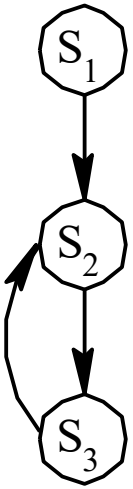
前趋图(Precedence Graph)是一个有向无循环图，记为DAG(Directed Acyclic Graph)，用于描述进程之间执行的前后关系。图中的每个结点可用于描述一个程序段或进程，乃至一条语句；结点间的有向边则用于表示两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)"→"。 ■

→={(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。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)。

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结点的执行时间。



(*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

对于图 2-2(*a*)所示的前趋图， 存在下述前趋关系：

$P_1 \rightarrow P_2$, $P_1 \rightarrow P_3$, $P_1 \rightarrow P_4$, $P_2 \rightarrow P_5$, $P_3 \rightarrow P_5$, $P_4 \rightarrow P_6$, $P_4 \rightarrow P_7$, $P_5 \rightarrow P_8$, $P_6 \rightarrow P_8$, $P_7 \rightarrow P_9$, $P_8 \rightarrow P_9$□

或表示为： ■

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

$\rightarrow=\{ (P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_5), (P_3, P_5), (P_4, P_6), (P_4, P_7), (P_5, P_8), (P_6, P_8), (P_7, P_9), (P_8, P_9)\}$

应当注意，前趋图中必须不存在循环，但在图2-2(*b*)中却有着下述的前趋关系： ■

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

2.1.2 程序的顺序执行

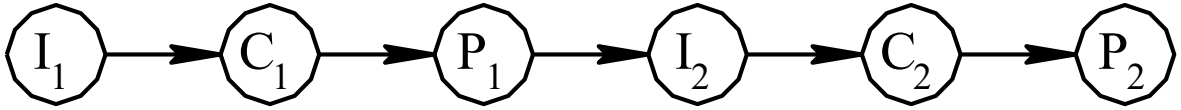
1. 程序的顺序执行

仅当前一操作(程序段)执行完后，才能执行后继操作。例如，在进行计算时，总须先输入用户的程序和数据，然后进行计算，最后才能打印计算结果。

S_1: a ：=x+y; ■

S_2: b ：=a-5; ■

S_3: c ：=b+1;



(*a*) 程序的顺序执行

(*b*) 三条语句的顺序执行

图 2-1 程序的顺序执行

2. 程序顺序执行时的特征

(1) 顺序性：一个程序开始执行必须要等到前一个程序已执行完成。

(2) 封闭性：程序一旦开始执行，其计算结果不受外界因素影响。

(3) 可再现性：程序的结果与它的执行速度无关（即与时间无关），只要给定相同的输入，一定会得到相同的结果。

2.1.3 程序的并发执行及其特征

1. 程序的并发执行

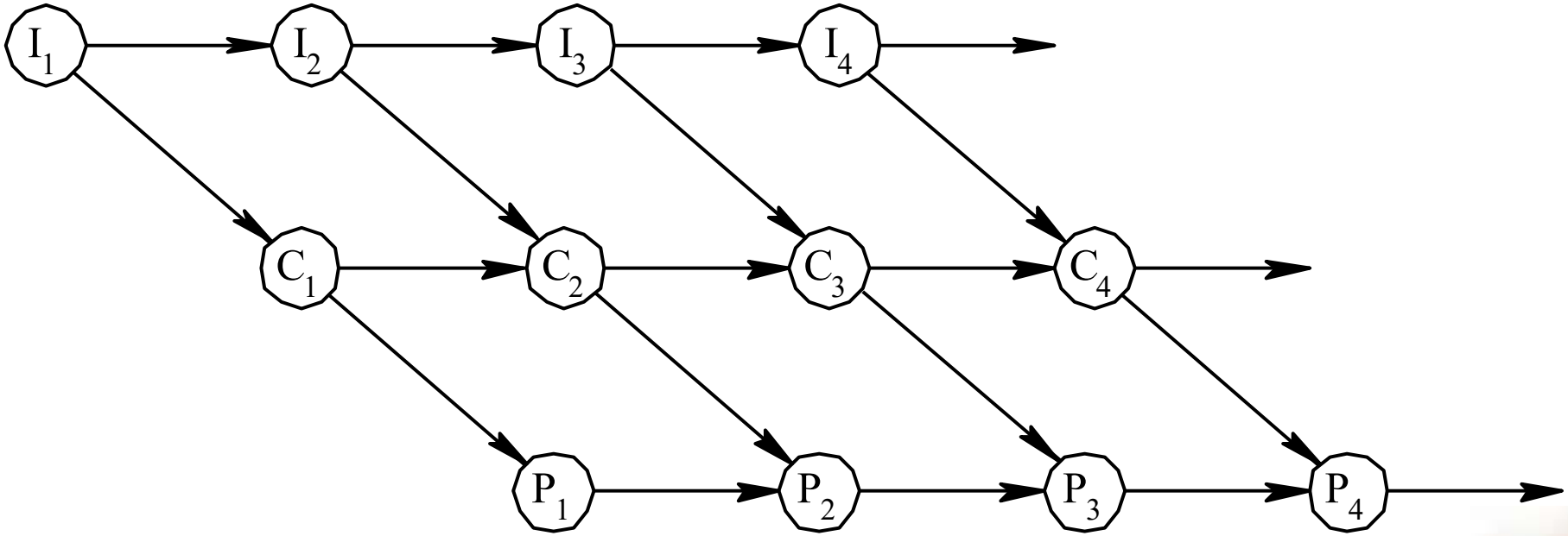


图 2-3 并发执行时的前趋图

在该例中存在下述前趋关系：■

$I_i \rightarrow C_i$，$I_i \rightarrow I_{i+1}$, $C_i \rightarrow P_i$, $C_i \rightarrow C_{i+1}$，$P_i \rightarrow P_{i+1}$

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

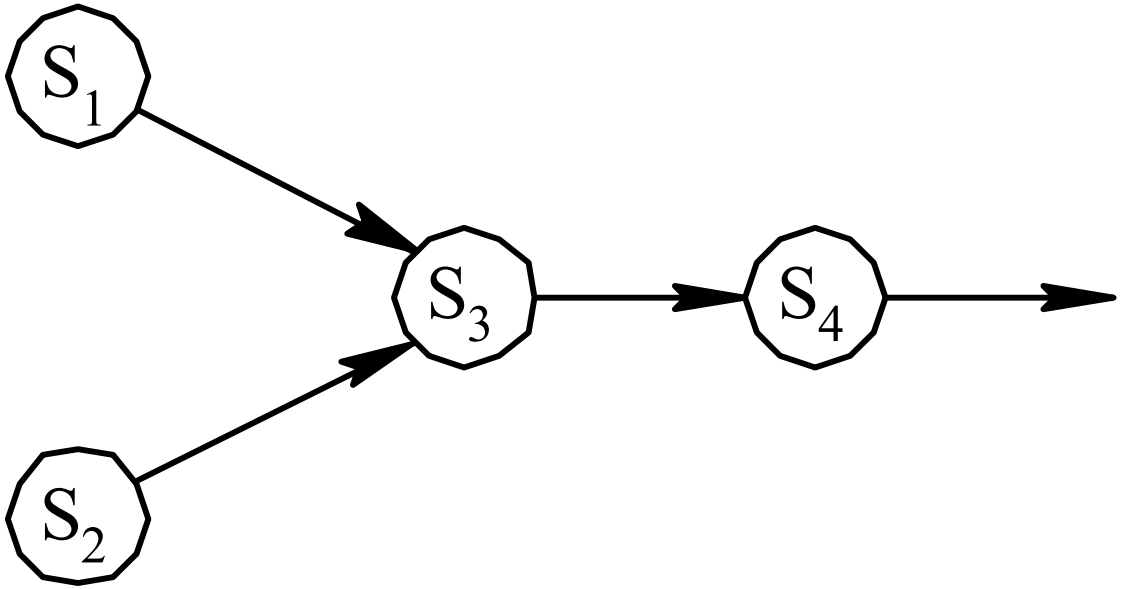


图 2-4 四条语句的前趋关系

2. 程序并发执行时的特征

程序并行执行时的特征

- **间断(异步)性**：“走走停停”，一个程序可能走到中途停下来，失去原有的时序关系；
- **失去封闭性**：共享资源，受其他程序的控制逻辑的影响。如：一个程序写到存储器中的数据可能被另一个程序修改，失去原有的不变特征。
- **失去可再现性**：失去封闭性 —>失去可再现性；外界环境在程序的两次执行期间发生变化，失去原有的可重复特征。并发程序执行的结果与其执行的相对速度有关，是不确定的。

Pa: a) N := N + 1;

Pb: b) Print(N);

c) N := 0;

由于并发程序A和B的运行速度不同，若N的初值为n，则可能会出现以下情况：

1）执行顺序: a->b->c, N值分别为：n+1, n+1,0。

2）执行顺序: b->c->a, N值分别为： n, 0,1。

3）执行顺序: b->a->c, N值分别为： n, n+1,0。

思考：任何程序并发执行都是不可再现的吗？

程序并发执行的条件

$R(p_i)=\{a_1,a_2,\ldots,a_n\}$：表示程序$p_i$在执行期间所需参考的所有变量的集合。称为$p_i$的“读集”。

$W(p_i)=\{b_1,b_2,\ldots,b_m\}$：表示程序$p_i$在执行期间所需改变的所有变量的集合。称为$p_i$的“写集”。

若两个程序p_1和p_2能满足下述Bernstein条件，它们可并发执行，且具有可再现性。

$R(p_1)\cap W(p_2)\cup R(p_2)\cap W(p_1)\cup W(p_1)\cap W(p_2)=\varphi$

2.2 进程的描述

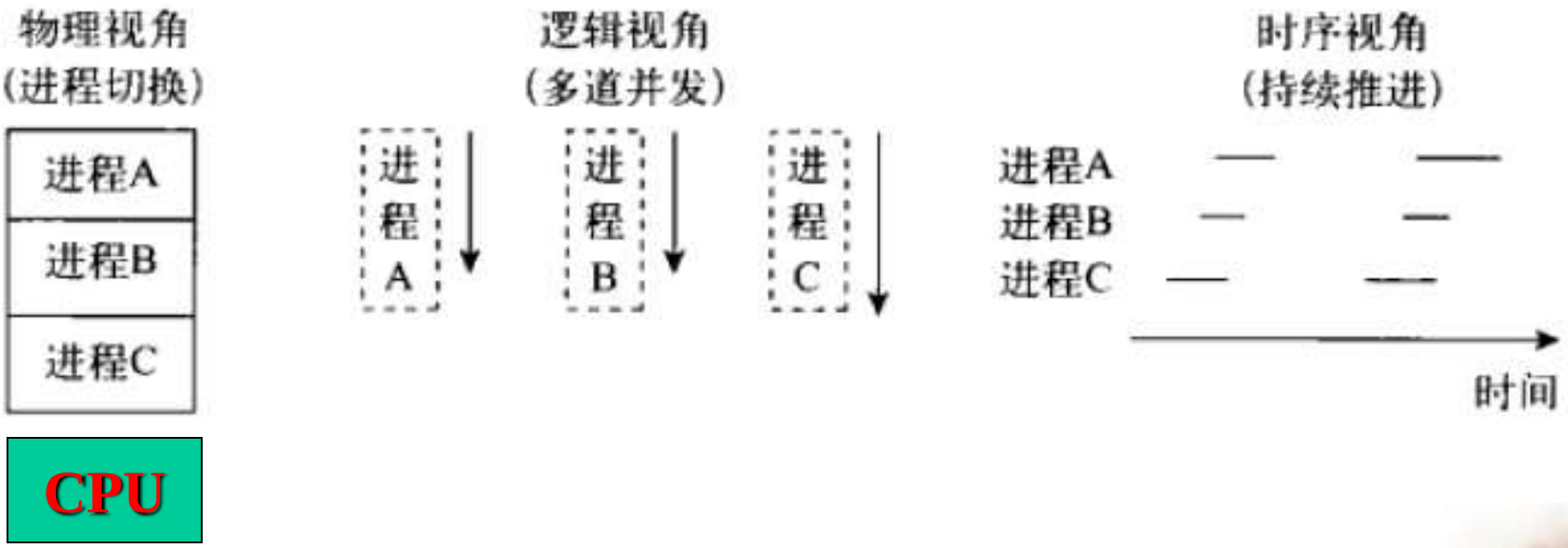
2.2.1进程的定义和特征

- 进程的引入

  为了使多道程序能够并发执行，以提高资源利用率和系统效率。

- 进程定义
  - **进程是具有独立功能的程序关于某个数据集合在计算机系统中的一次执行过程。**
  - 系统进行**资源分配和调度的、一个可并发执行的独立单位。**

- 进程模型


物理视角
(进程切换)
进程A
进程B
进程C
CPU
逻辑视角
(多道并发)
进程A
进程B
进程C
时序视角
(持续推进)
进程A
进程B
进程C
时间

进程同程序的比较：

- 程序是指令的有序集合，其本身没有任何运行的含义，是一个静态的概念。而进程是程序在处理机上的一次执行过程，它是一个动态的概念。
- 程序可以作为一种软件资料长期存在，而进程是有一定生命期的。程序是永久的，进程是暂时的。
- 进程更能真实地描述并发，而程序不能。
- 进程是由程序和数据两部分组成的。
- 进程具有创建其他进程的功能，而程序没有。
- 同一程序同时运行于若干个数据集合上，它将属于若干个不同的进程。也就是说同一程序可以对应多个进程 。

进程的特征：

•动态性：进程是程序的执行。

•并发性：多个进程可同存于内存中，能在一段时间内同时运行。

•独立性：独立运行的基本单位，独立获得资源和调度的基本单位。

•异步性：各进程按各自独立的不可预知的速度向前推进。

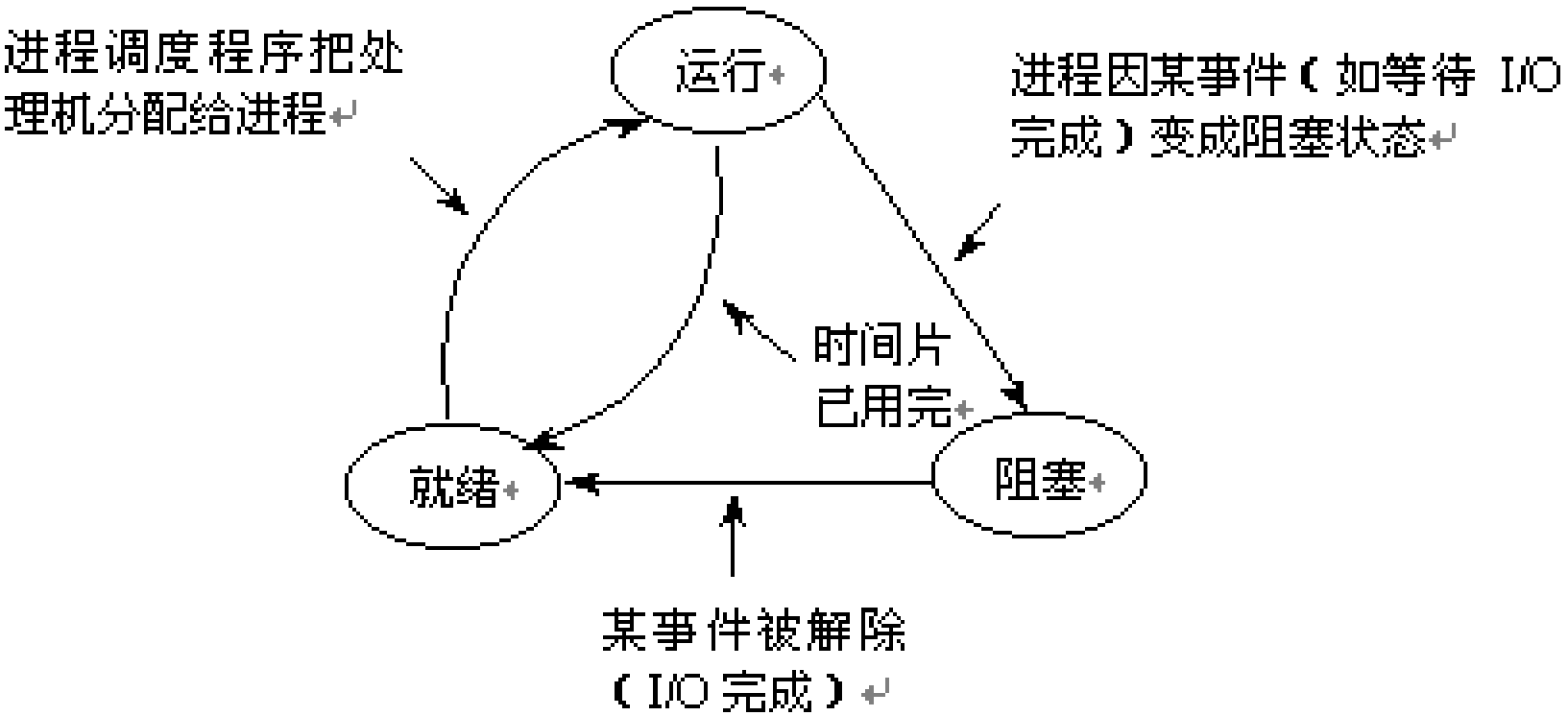
•结构特征：由程序段 、数据段、进程控制块三部分组成。

2.2.2 进程基本状态及转换

1. 进程的三种基本状态

- 就绪状态（Ready）：存在于处理机调度队列中的那些进程，它们已经准备就绪，一旦得到CPU，就立即可以运行。这些进程所处的状态为就绪状态。
- 执行状态（Running）：正在运行的进程所处的状态为执行状态(或运行状态)。
- 阻塞状态（Wait / Blocked ）：若一进程正在等待某一事件发生（如等待输入输出工作完成），这时，即使给它CPU，它也无法运行，称该进程处于阻塞状态（或等待、 睡眠、封锁状态）。

2. 三种基本状态的转换

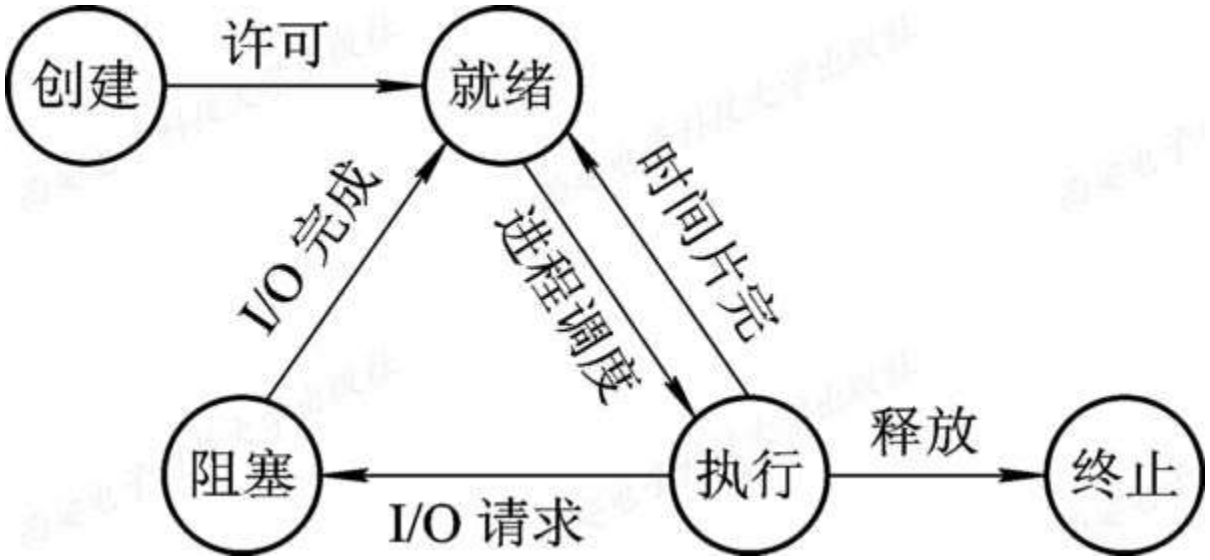

进程调度程序把处理机分配给进程
运行
进程因某事件（如等待 I/O 完成）变成阻塞状态
时间片已用完
就绪
阻塞
某事件被解除（I/O 完成）

3. 创建和终止状态

1）创建状态：进程创建过程中因资源得不到满足而不能被调度运行的状态。

2）终止状态：进程运行结束、或出现错误、或被其他进程终结，但还没有被撤销的状态。

五种状态及转换关系图



2.2.3 挂起操作和进程状态的转换

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。

引入挂起原语操作后三个进程状态的转换

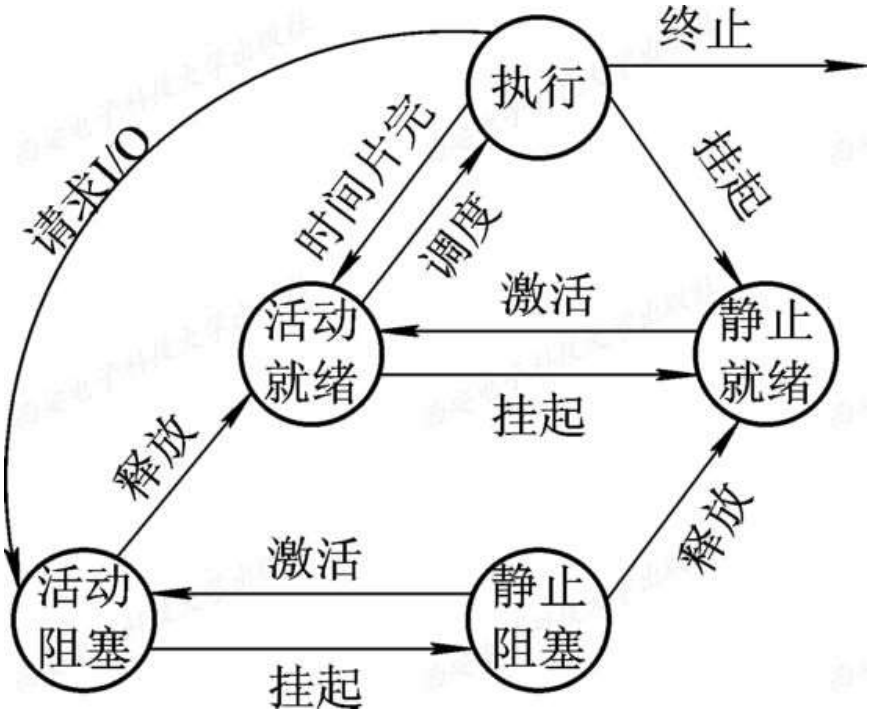
在引入挂起原语Suspend和激活原语Active后，在它们的作用下，进程将可能发生以下几种状态的转换：

(1) 活动就绪→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。



具有挂起状态的进程状态图

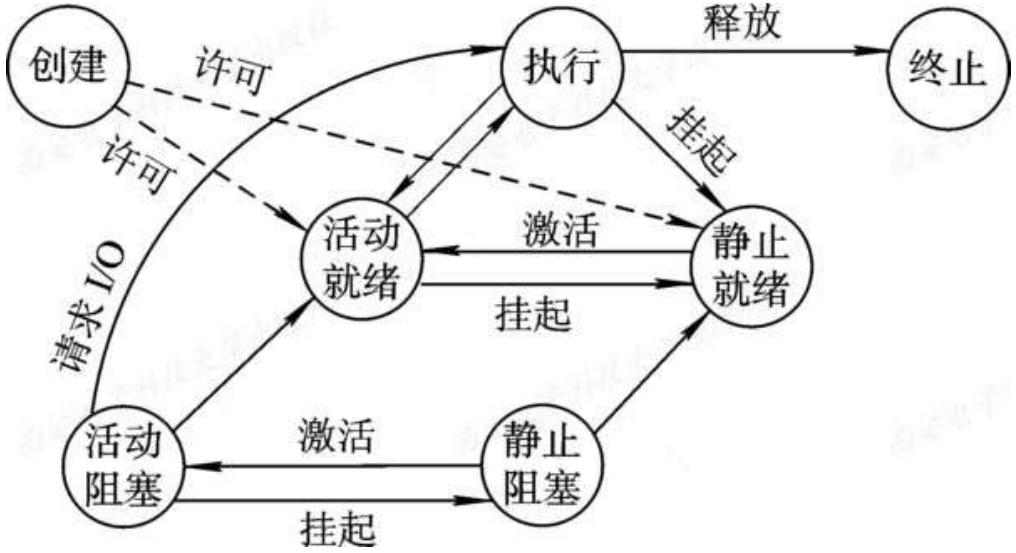
增加考虑下面的几种情况：

(1) NULL→创建：

(2) 创建→活动就绪：

(3) 创建→静止就绪：

(4) 执行→终止：



具有创建、终止和挂起状态的进程状态图

【思考题】

1. 如果系统中有N个进程，获处理机运行的进程最多几个，最少几个；就绪进程最多几个最少几个；阻塞进程最多几个，最少几个？

2. 有没有这样的状态转换，为什么？

阻塞—运行；　　就绪—阻塞

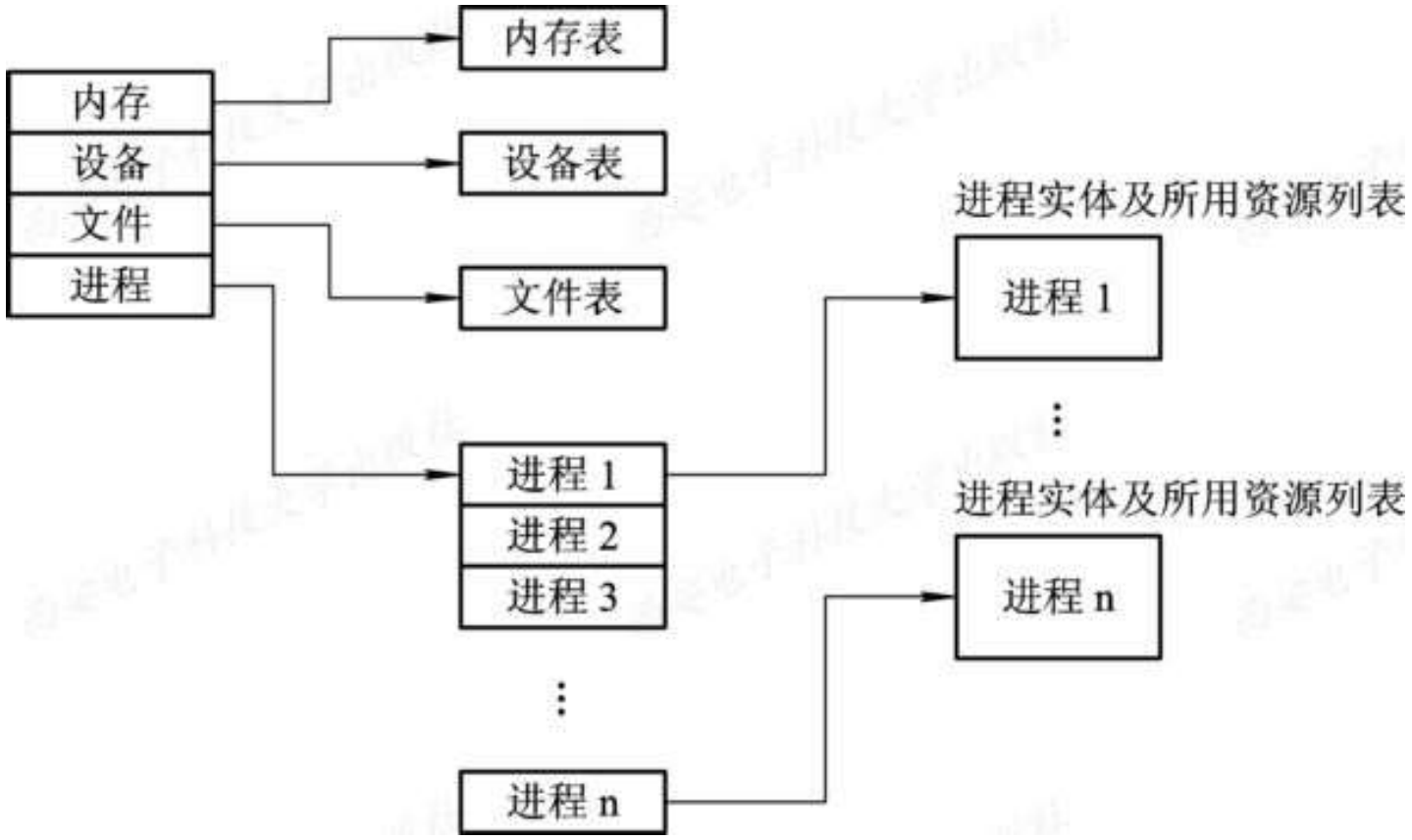
2.2.4 进程管理中的数据结构

1.操作系统中用于管理的数据结构

- 计算机中对于每个资源和每个进程都设置了一个数据结构。
- 为了描述一个进程和其它进程以及系统资源的关系，为了刻画一个进程在各个不同时期所处的状态，人们采用了一个与进程相联系的数据块，称为进程控制块（PCB）。
- 系统利用PCB来控制和管理进程，所以PCB是系统感知进程存在的唯一标志
- 进程与PCB是一一对应的。

- OS管理的这些数据结构一般分为以下四类

内存表	设备表	文件表	进程表
			• 用于进程管理，通常又被称为进程控制块PCB



操作系统控制表的一般结构

2. 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，一个能与其它进程并发执行的进程。或者说，OS是根据PCB来对并发执行的进程进行控制和管理的。

- 作为独立运行的基本单位的标志
- 能实现间断性运行方式
- 提供进程管理所需的信息
- 提供进程调度所需的信息
- 实现与其他进程的同步与同信

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。在所有的操作系统中，都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) 外部标识符。它由创建者提供，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，还可设置用户标识，以指示拥有该进程的用户。

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， 在大多数处理机中，有 8~32 个通用寄存器，在RISC结构的计算机中可超过 100 个；

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：其中含有状态信息，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PCB中还存放一些与进程调度和进程对换有关的信息，包括：

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：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 ■

进程控制信息包括：

① 程序和数据的地址： 是指进程的程序和数据所在的内存或外存地(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：指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：是一张列出了除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。

3. 进程控制块的组织方式

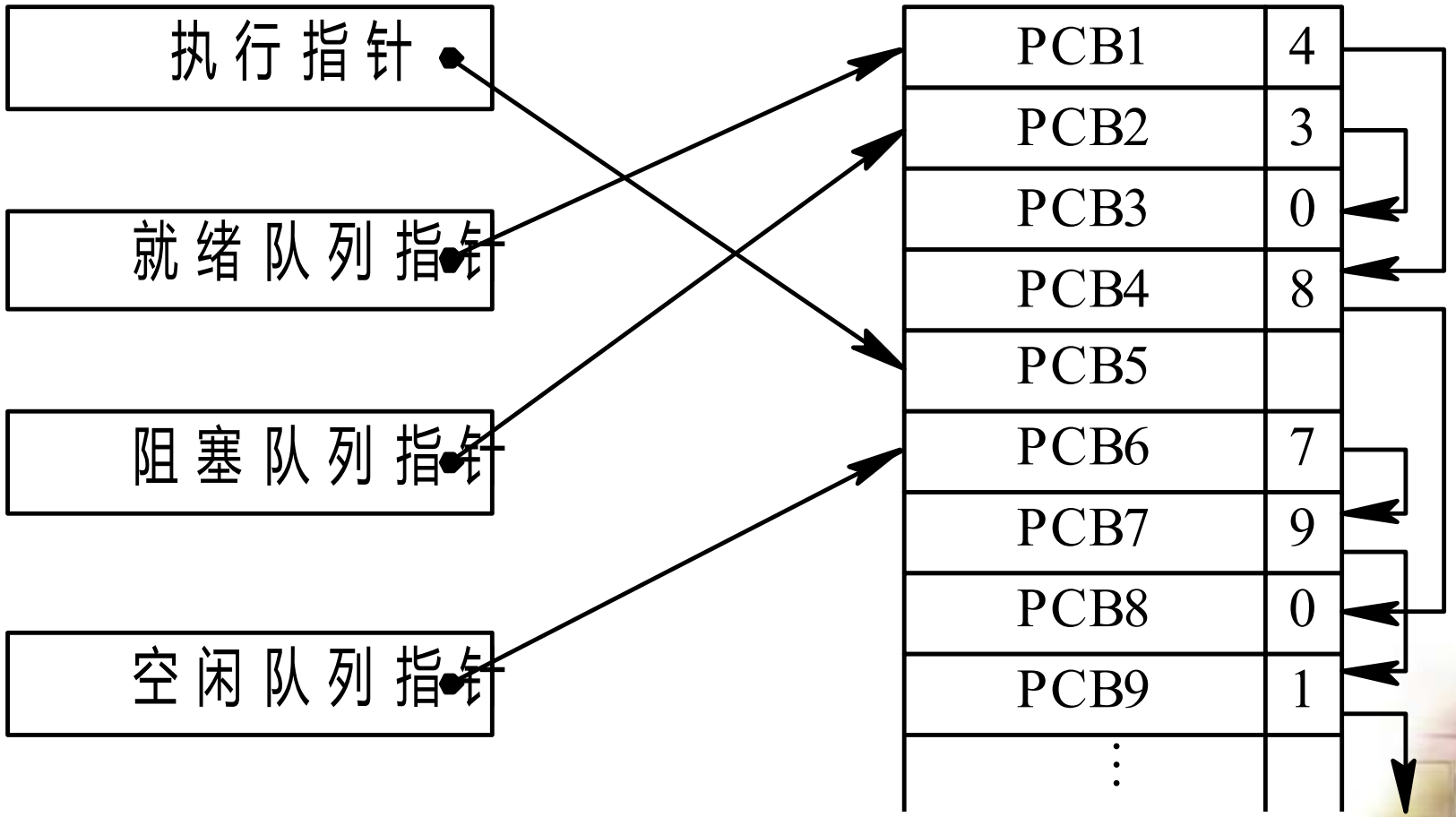
PCB表：

系统把所有PCB组织在一起，并把它们放在内存的固定区域，就构成了PCB表。

PCB表的大小决定了系统中最多可同时存在的进程个数，称为系统的并发度。

1) 链接方式

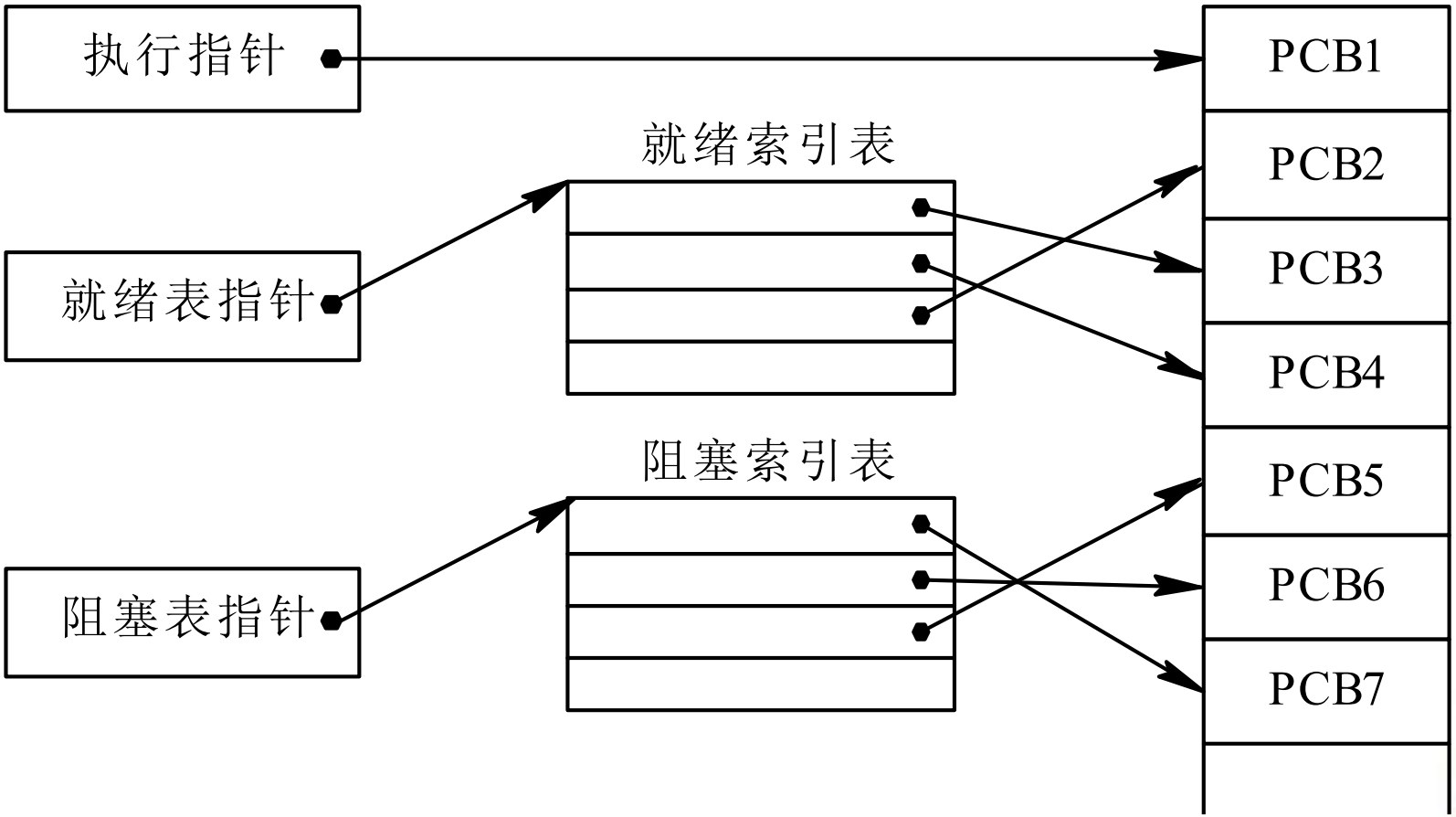
相同状态的进程PCB组成一个链表，不同状态对应多个不同的链表。



PCB链接队列示意图

2) 索引方式

对具有相同状态的进程，分别设置各自的PCB索引表，表明PCB在PCB表中的地址。



按索引方式组织PCB

2.3 进 程 控 制

2.3.1 操作系统内核

1.支撑功能

- 中断处理
- 时钟管理
- 原语操作

2. 资源管理功能

- 进程管理
- 存储器管理
- 设备管理
- 文件管理

进程控制就是对系统中的所有进程实施管理，进程控制一般由**原语**来实现。

所谓原语是一种特殊的系统功能调用，它可以完成一个特定的功能，其特点是原语执行时不可被中断。

思考：如何实现原语?

实现：开关中断

常用的进程控制原语

- 创建原语
- 终止原语
- 阻塞原语、唤醒原语

思考：为什么创建进程要用原语来实现？

2.3.2 进程的创建

图 2-9 进程树

2.3.2 进程的创建

- 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。

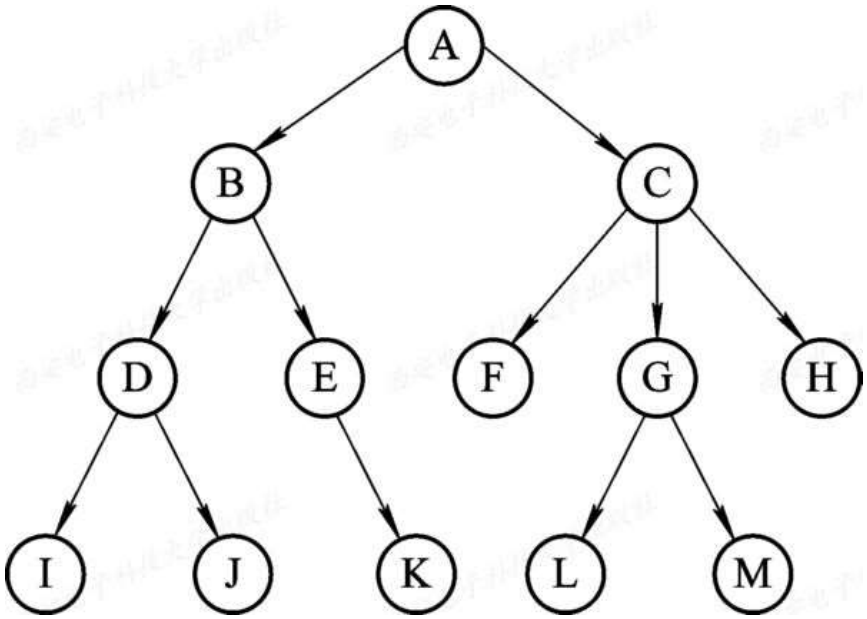
✓ 子进程可继续创建更多的孙进程，由此便形成了一个进程的层次结构。

✓ 如在UNIX中，进程与其子孙进程共同组成一个进程家族(组)。

- 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。



- 引起创建进程的事件

✓ 为使程序之间能并发运行，应先为它们分别创建进程。

✓ 导致一个进程去创建另一个进程的典型事件有四类：

(1) 用户登录。
(2) 作业调度。
(3) 提供服务。
(4) 应用请求。

- 进程的创建(Creation of Progress)

(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。

2.3.3 进程的终止

1. 引起进程终止(Termination of Process)的事件 ■

1) 正常结束 ■

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。当程序运行到Holt指令时，将产生一个中断，去通知OS本进程已经完成。在分时系统中，用户可利用Logs off去表示进程运行完毕，此时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。这类异常事件很多，常见的有：

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：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：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：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：进程的执行时间超过了指定的最大值；

⑥ 等待超时：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：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 ■

外界干预并非指在本进程运行中出现了异常事件，而是指进程应外界的请求而终止运行。这些干预有：

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：由于父进程具有终止自己的任何子孙进程的权利，因而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也将他的所有子孙进程终止。

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。

2.3.4 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、进程的阻塞过程

停止运行

修改PCB中的状态
（运行→阻塞）

插入到相应的阻塞队列

另调度一就绪进程，切换CPU
（保留阻塞进程的CPU状态信息至PCB，再按新进程PCB中的状态设置CPU环境）

3、进程的唤醒过程

将阻塞进程移出阻塞队列

修改PCB中的状态
（阻塞→就绪）

插入到就绪队列

如需要，则另调度
一就绪进程，切换CPU

注意：

进程的阻塞是进程自身的一种主动行为。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。

进程的唤醒是被动行为。处于阻塞状态的进程是绝不可能叫醒它自己的，它必须由它的合作进程用唤醒原语唤醒它。

2.3.5 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：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。 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。最后，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，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。

2.4 进 程 同 步

在OS中引入进程后，一方面可以使系统中的多道程序并发执行，这不仅能有效地改善资源利用率，还可显著地提高系统的吞吐量，但另一方面却使系统变得更加复杂。如果不能采取有效的措施，对多个进程的运行进行妥善的管理，必然会因为这些进程对系统资源的无序争夺给系统造成混乱。致使每次处理的结果存在着不确定性，即显现出其不可再现性。

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：多个进程彼此无关，同处于一个系统中，共享系统资源。

进程同步的主要任务：保证诸进程能互斥访问临界资源。

例：有两进程A 和B，如果A 提出打印请求，系统已把唯一的一台打印机分配给了进程B，则进程A 只能阻塞；一旦B 释放打印机，A 才由阻塞改为就绪。

(2) 直接相互制约关系：进程之间存在一种相互合作的关系。

进程同步的主要任务：保证相互合作的进程在执行次序上的协调，不会出现与时间有关的错误。

例：有输入进程A 通过单缓冲向进程B 提供数据。当缓冲空时，计算进程因不能获得所需数据而阻塞，当进程A 把数据输入缓冲区后，便唤醒进程B；反之，当缓冲区已满时，进程A 因没有缓冲区放数据而阻塞，进程B 将缓冲区数据取走后便唤醒A。

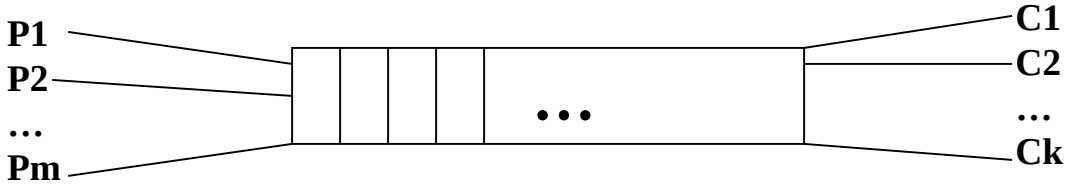
2. 临界资源(Critical Resouce)

临界资源：在一段时间内只允许一个进程访问的资源。当它用完释放后，其他进程才可使用。

如:打印机，绘图机，变量，数据等，各进程间采取互斥方式实现对这种临界资源的共享，从而实现并发进程的封闭性。

例：生产者-消费者(producer-consumer)问题

一个著名的进程同步问题。它描述的是：有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有n个缓冲区的缓冲池，生产者进程将它所生产的产品放入一个缓冲区中； 消费者进程可从一个缓冲区中取走产品去消费。尽管所有的生产者进程和消费者进程都是以异步方式运行的，但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。



我们可利用一个数组来表示上述的具有n个(0，1，...，n-1)缓冲区的缓冲池。用输入指针in来指示下一个可投放产品的缓冲区，每当生产者进程生产并投放一个产品后，输入指针加1；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。由于这里的缓冲池是组织成循环缓冲的，故应把输入指针加1表示成 in=(in+1)%n；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。此外，还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。生产者和消费者两进程共享下面的变量：

int n; ■

type item=...; ■

item buffer[n］; ■

int in, out,counter; ■

指针in和out初始化为0。在生产者和消费者进程的描述中，no-op是一条空操作指令，while (condition) no-op语句表示重复的测试条件(condication)，重复测试应进行到该条件变为false(假)，即到该条件不成立时为止。在生产者进程中使用一局部变量nextp，用于暂时存放每次刚生产出来的产品；而在消费者进程中，则使用一个局部变量nextc,用于存放每次要消费的产品。

```
void producer()
{while(1) ■
 {  produce an item in nextp; ■
    ✔ ... ■
    while (counter==n) no-op; ■
    buffer［in］=nextp; ■
    in =(in+1 )% n; ■
    counter++;} ■
   }
void consumer()
{while(1)
   { while (counter==0) no-op; ■
    nextc=buffer[out]; ■
    out = (out+1) % n; ■
    counter --; ■
    consumer the item in nextc;} ■
   }
```

虽然上面的生产者程序和消费者程序，在分别看时都是正确的，而且两者在顺序执行时其结果也会是正确的，但若并发执行时，就会出现差错，问题就在于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：

■ register 1 =counter; register 2 =counter; ■

register1 =register 1+1; register 2 =register 2-1;

counter =register 1; counter =register 2;

假设：counter的当前值是5。如果生产者进程先执行左列的三条机器语言语句，然后消费者进程再执行右列的三条语句， 则最后共享变量counter的值仍为5；反之，如果让消费者进程先执行右列的三条语句，然后再让生产者进程执行左列的三条语句，counter值也还是5，但是，如果按下述顺序执行：

register 1 =counter; (register 1=5) ■

register 1 =register 1+1; (register 1=6) ■

register 2 =counter; (register 2=5) ■

register 2 =register 2-1; (register 2=4) ■

counter =register 1; (counter=6) ■

counter =register 2; (counter=4)□

3. 临界区(critical section)

临界区：进程中访问临界资源的那段代码称为临界区。

可把一个访问临界资源的循环进程描述如下：

```
while(1)
{
        entry section
    critical section;
        exit section
    remainder section;
}
```

- 进入区(Entry Section)

  增加在临界区前面的一段代码，用于检查所要访问的临界资源此刻是否被访问。

- 退出区(Exit Section)

  增加在临界区后面的一段代码，用于将临界资源的访问标志恢复为未被访问标志。

- 剩余区(Remainder Section)

  进程中除了进入区、临界区及退出区之外的其余代码。

要进入临界区的若干进程必须满足：

（1）一次只允许一个进程进入临界区

（2）任何时候，处于临界区的进程不得多于一个

（3）进入临界区的进程要在有限的时间内退出

（4）如果不能进入自己的临界区，则应让出处理机资源

解决临界区（互斥）问题的几类方法：

（1）软件方法和硬件方法

（2） P-V操作

（3）管程机制

} 同步机制

4. 同步机制应遵循的规则

- 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。
- 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，因而其他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。
- 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。
- 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”。

补充1： 软件方法解决进程互斥

- 现在很少用软件方法解决互斥，但通过学习软件解法能使读者了解到，在早期进程互斥问题的解决并不是一件很简单的事。

假如有两个进程Pi和Pj，它们共享一个临界资源R。如何用软件方法使进程Pi和Pj能互斥地访问R。 算法如下：

- 为每一进程设标志位flag[i]，当flag[i]=true时，表示进程pi要求进入，或正在临界区中执行。此外再设一个变量turn，用于指示允许进入的进程编号。
- 进程pi为了进入先置flag[i]=true，并置turn为j，表示应轮到pj进入。接着再判断flag[j] and turn=j的条件是否满足。若不满足则可进入。或者说，当flag[j]=false或者turn=i时，进程pi可以进入。前者表示pj未要求进入，后者表示仅允许pi进入。

```
Pi：  while(1)
      {  flag[i]=true;
         turn=j;
         While (flag[ j] && turn==j) no_op;

         Critical section
         flag[i]=false;

         Other code
      }
```

```
Pj：  while(1)
      { flag[j]:=true;
        turn:=i;
        While (flag[ i] && turn==i) no_op;

        Critical section
        flag[j]:=false;

        Other code
    }
```

软件解法的缺点

1. 忙等待
2. 实现过于复杂，需要较高的编程技巧

2.4.2 硬件同步机制

用软件解决很难，现代计算机大多提供一些硬件指令。

- 利用Test-and-Set指令实现互斥
- 利用swap指令实现进程互斥

Test-and-Set指令实现互斥

1、Test-and-Set指令

```
boolean TS(boolean *lock)
{boolean old;
   old=*lock;
   *lock:=true;
   return old;}
```

其中，有lock有两种状态：当lock=false时，表示该资源空闲；当lock=true时，表示该资源正在被使用。

2、利用TS指令实现进程互斥

为每个临界资源设置一个全局布尔变量lock，并赋初值false，表示资源空闲。

```
do{
   while (TS(&lock)) skip;
   critical section
   lock=false;
   Other code
}while(true);
```

swap指令实现进程互斥

1、swap指令

又称交换指令，X86中称为XCHG指令。

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

2、利用swap实现进程互斥

为每一临界资源设置一个全局布尔变量lock，其初值为false，在每个进程中有局部布尔变量key。

```
do{
   key=true;
   do{Swap(&lock,&key);     while( key==true);
   Critical section;
   lock=false;
   Other code
}while(true);
```

2.4.3 信号量机制

1. 整型信号量 ■

1965年，由荷兰学者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。 wait和signal操作可描述为： ■

wait(S){ while (S<=0) no-op; ■

S=S-1;} ■

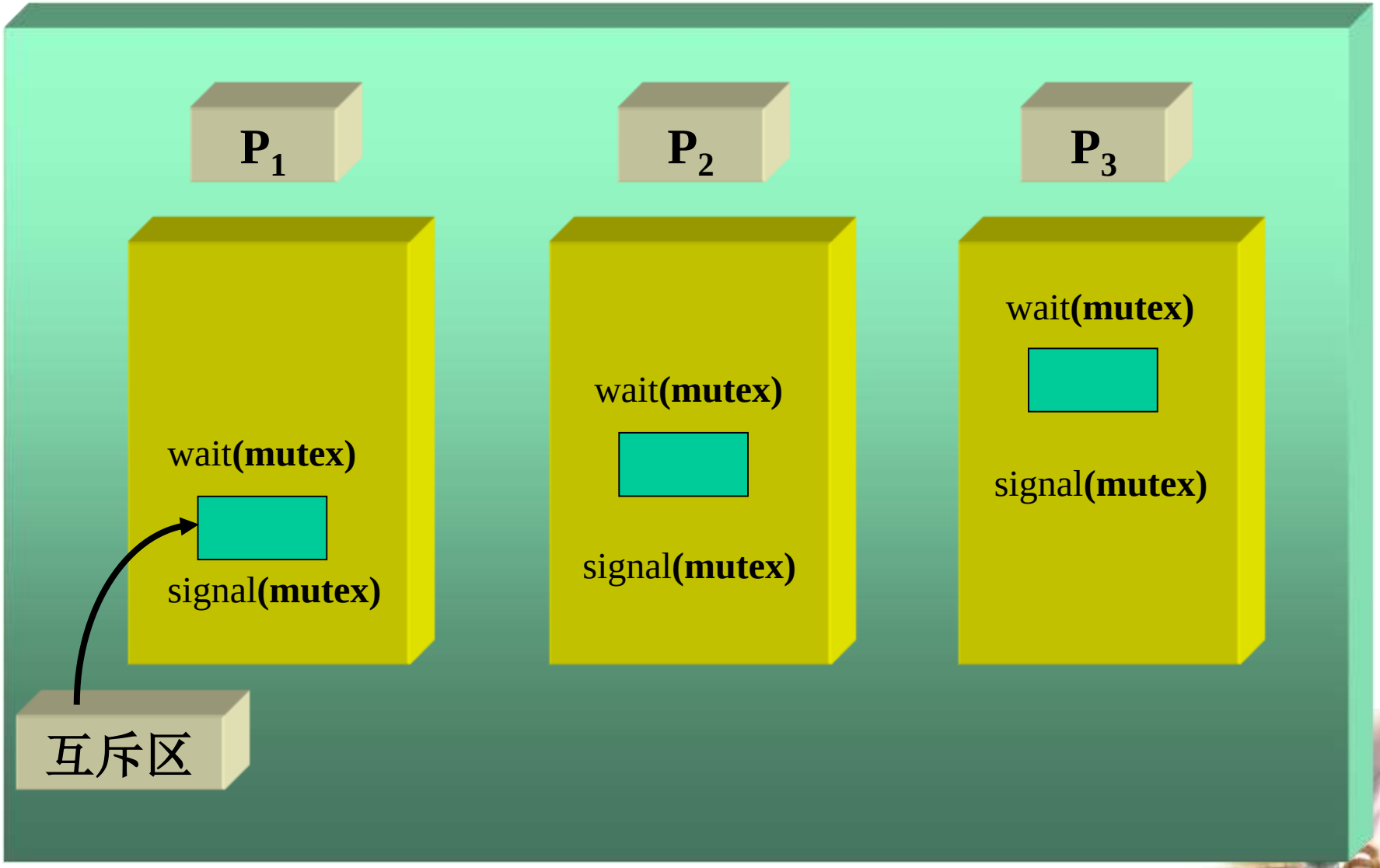
signal(S){ S=S+1;} ■

信号量的使用

必须置一次且只能置一次初值
初值不能为负数

只能执行wait()、signal()操作

用wait()、signal()操作解决进程间互斥问题



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。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L，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。它所包含的上述两个数据项可描述为：

```
typedef struct { ■
        int value; ■
        struct process_control_block *list; ■
        }semaphore;
```

相应地，wait(S)和signal(S)操作可描述为： ■

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

在记录型信号量机制中，S->value的初值表示系统中某类资源的数目，因而又称为资源信号量，对它的每次wait操作，意味着进程请求一个单位的该类资源，因此描述为S->value--；当S->value＜0时，表示该类资源已分配完毕，因此进程应调用block原语，进行自我阻塞，放弃处理机，并插入到信号量链表S->list中。可见，该机制遵循了“让权等待”准则。 此时S->value的绝对值表示在该信号量链表中已阻塞进程的数目。对信号量的每次signal操作，表示执行进程释放一个单位资源，故S->value++操作表示资源数目加1。 若加1后仍是S->value<=0，则表示在该信号量链表中，仍有等待该资源的进程被阻塞，故还应调用wakeup原语，将S.L链表中的第一个等待进程唤醒。如果S->value的初值为1，表示只允许一个进程访问临界资源，此时的信号量转化为互斥信号量。

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， 即

process A: process B: ■

wait(Dmutex); wait(Emutex); ■

wait(Emutex); wait(Dmutex); ■

若进程A和B按下述次序交替执行wait操作： ■

process A: wait(Dmutex); 于是Dmutex=0 ■

process B: wait(Emutex); 于是Emutex=0 ■

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

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

• AND型信号量集是指同时需要多种资源且每种占用一个时的信号量操作。

• AND型信号量集的基本思想：在一个原语中申请整段代码需要的多个临界资源，要么全部分配给它，要么一个都不分配，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。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，或称为同时wait操作。

AND型信号量集wait原语记为Swait
AND型信号量集signal原语记为Ssignal

定义如下：

Swait(S_1, S_2, ..., S_n) ■

{while(1)

{ if(S_i>=1 && ... && S_n>=1){ ■

for (i =1;i<=n;i++) S_i--; ■

berak;} ■

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.}

}

Ssignal(S_1, S_2, ..., S_n) ■

{while(1){

for (i =1;i<=n;i++)

{ Si++; ■

Remove all the process waiting in the queue associated with S_i into the ready queue. ■ }

}}

4. 信号量集

• 信号量集是指同时需要多种资源、每种占用的数目不同、且可分配的资源还存在一个临界值时的信号量处理。

（一次需要N个某类临界资源时，就要进行N次wait操作——低效又可能死锁）

• 信号量集的基本思路就是在AND型信号量集的基础上进行扩充，在一次原语操作中完成所有的资源申请。

进程对信号量Si的

测试值为ti（表示信号量的判断条件，要求Si >= ti；即当资源数量低于ti时，便不予分配）。

占用值为di（表示资源的申请量，即Si = Si - di）

对应的原语格式为：

Swait(S1, t1, d1; ...; Sn, tn, dn);

Ssignal(S1, d1; ...; Sn, dn);

Swait(S1, t1, d1, …, Sn, tn, dn) ■
{while(1)
{if(S1>=t1&& … && Sn>=tn){ ■
for (i=1;i<=n;i++) Si=Si-di; ■
break;}
else{ ■
Place the executing process in the waiting queue of the first Si found with Si＜ti and set its program counter to the beginning of the Swait Operation. } ■
}} ■
Ssignal(S1, d1, …, Sn, dn)
{while(1)
{ for (i=1;i<=n;i++) Si =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) 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)一般“信号量集”未必成对使用Swait和Ssignal：如：一起申请，但不一起释放。

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，并置初值为1，然后将各进程的临界区CS置于wait(mutex)和signal(mutex)操作之间即可。

```
Semaphore mutex=1;
   Pa(){                              Pb(){
   while(1) {                         while(1){
         wait(mutex);                       wait(mutex);
         critical section;                  critical section;
         signal(mutex);                     signal(mutex);
         remainder section;                 remainder section;
        }                                   }
  }                                    }
```

2. 利用信号量实现前趋关系

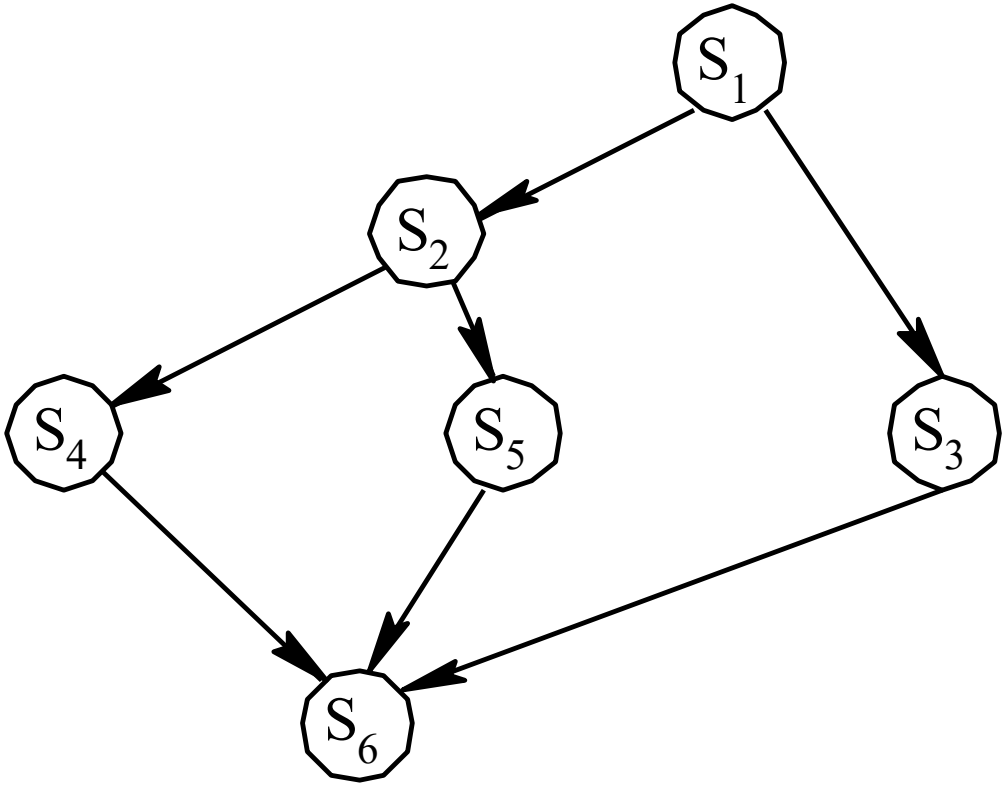


图 2-10 前趋图举例

P1(){ S_1; signal(a); signal(b); } ■

P2(){wait(a); S_2; signal(c); signal(d); } ■

P3(){wait(b); S_3; signal(e); }

P4(){wait(c); S_4; signal(f); } ■

P5(){wait(d); S_5; signal(g);] ■

P6(){wait(e); wait(f); wait(g); S_6; } ■

main(){

semaphore a,b,c,d,e,f,g;

a.value= b.value = c.value = d.value = e.value = f.value = g.value=0;

cobegin

p1();p2(); p3(); p4(); p5(); p6();

Coend}

信号量及wait、signal操作讨论(1)

1) 信号量的物理含义：

S>0表示有S个资源可用

S=0表示无资源可用

S<0则| S |表示S等待队列中的进程个数

wait(S)：表示申请一个资源

signal(S)：表示释放一个资源。

互斥信号量的初值为1；资源信号量初值为该资源的可用数量。

信号量及wait、signal操作讨论（2）

2) wait 、signal 操作必须成对出现，有一个wait操作就一定有一个signal操作。

当为互斥操作时，它们同处于同一进程。

当为同步操作时，则不在同一进程中出现。

如果wait(S1)和wait(S2)两个操作在一起，那么wait操作的顺序至关重要。

一个同步wait操作与一个互斥wait 操作在一起时同步wait操作在互斥wait操作前。

而两个signal操作无关紧要。

信号量及wait、signal操作讨论（3）

3）wait、signal操作的优缺点

优点：

简单，而且表达能力强（*用wait、signal操作可解决任何同步互斥问题*）

缺点：

不够安全，wait、signal操作使用不当会出现死锁；

遇到复杂同步互斥问题时实现复杂

2.4.5 管 程 机 制

建立管程的基本理由是：由于对临界区的执行分散在各进程中，这样不便于系统对临界资源的控制和管理，也很难发现和纠正分散在用户程序中的对同步原语的错误使用等问题。为此，应把分散的各同类临界区集中起来。并为每个可共享资源设立一个专门的管程来统一管理各进程对该资源的访问。这样既便于系统管理共享资源，又能保证互斥访问。

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。

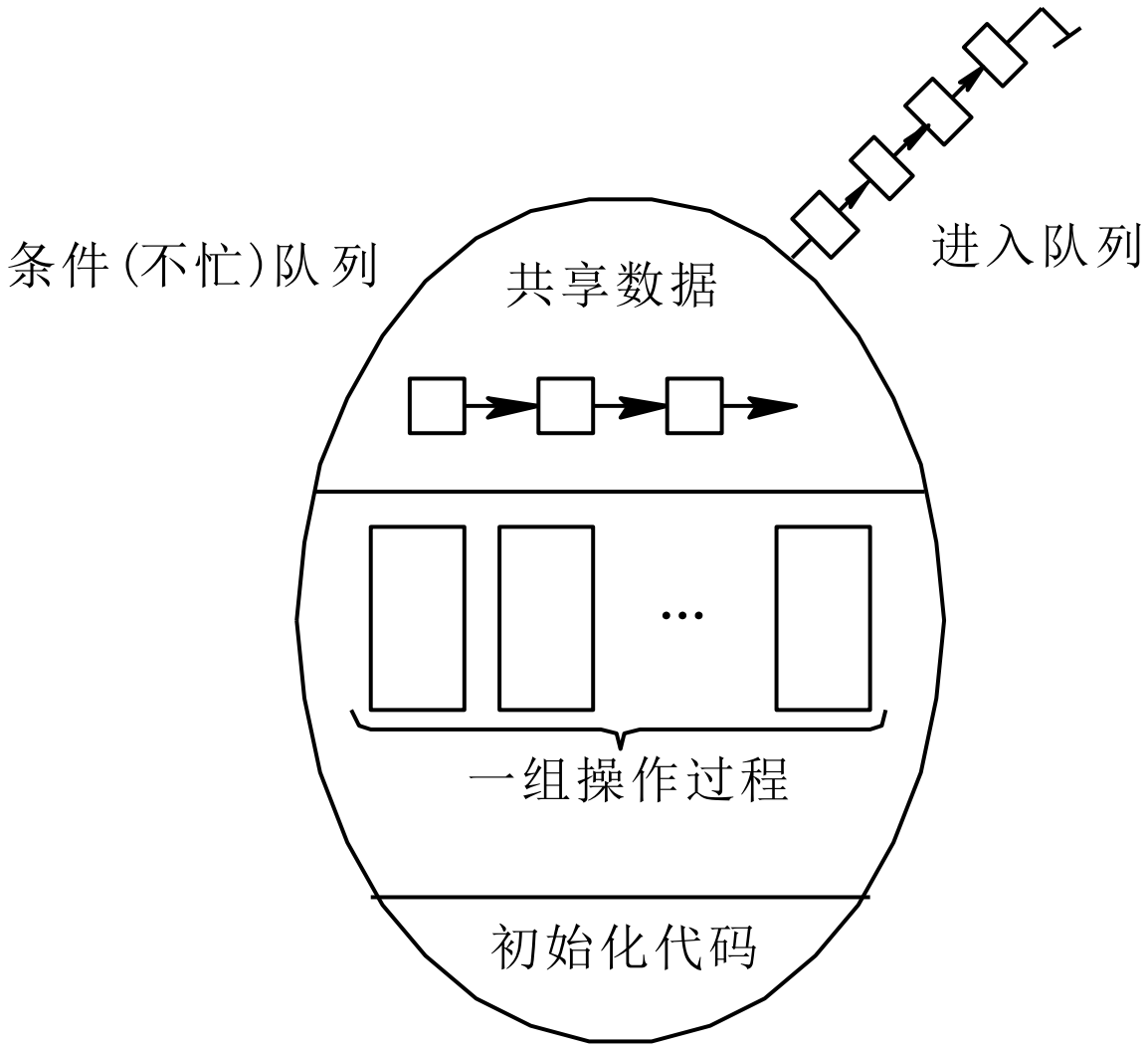


图 2-11 管程的示意图

管程的语法如下：

```
monitor monitor_name{
  share variable declarations;
  cond declarations;
public:
  void P1(...);
       { ... };
  void P2(...);
       { ... };
       ✔ ...
  void Pn(...);
       { ... };
initialization code;}
```

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。例如，由于共享数据被占用而使调用进程等待，该条件变量的形式为：nonbusy:condition。此时，wait原语应改为nonbusy.wait，相应地，signal应改为nonbusy.signal。■

应当指出，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。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。

如果有进程Q处于阻塞状态， 当进程P执行了X.signal操作后，怎样决定由哪个进行执行，哪个等待，可采用下述两种方式之一进行处理：■

(1) P等待，直至Q离开管程或等待另一条件。■

(2) Q等待，直至P离开管程或等待另一条件。■

采用哪种处理方式， 当然是各执一词。 但是Hansan却采用了第一种处理方式。

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。 ■

(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  producer-consumer ■
    {int in,out,count; ■
     item buffer[n]; ■
    condition notfull, notempty;
    public; ■
     void put(item x){ ■
       if (count>=n) notfull.wait; ■
          buffer[in]  =x; ■
          in =(in+1)% n; ■ count++; ■
          if notempty.queue then notempty.signal; ■
     } ■
```

```
void  get(item x) ■ {
    if (count<=0) notempty.wait; ■
       x=buffer[out]; ■
       out= (out+1) % n; ■
       count--; ■
       if notfull.quene then notfull.signal; ■
} ■
{ in =out=0; count=0 ;}
}PC;
```

在利用管程解决生产者-消费者问题时， 其中的生产者和消费者可描述为：

```
void producer(){ ■
        item x;
       while(1){ ■
         produce an item in nextp; ■
         PC.put(item); ■
        }} ■
   void consumer(){ ■
        item x;
       while(1){
         PC.get(item); ■
         consume the item in nextc; ■
        }}
```

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

2.5 经典进程的同步问题

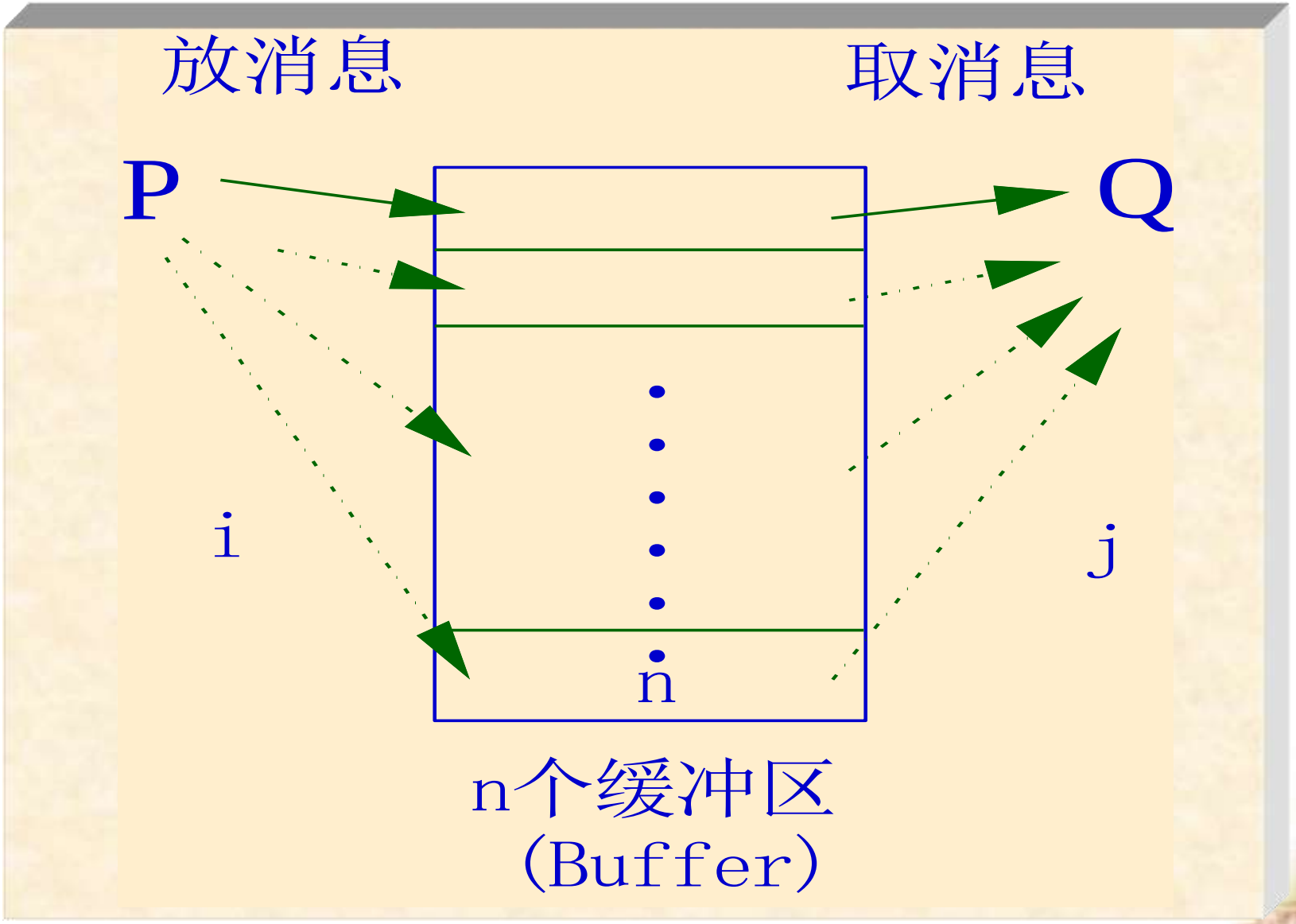
2.5.1 生产者–消费者问题 ■

前面我们已经对生产者–消费者问题(The proceducer-consumer problem)做了一些描述，但未考虑进程的互斥与同步问题，因而造成了数据Counter的不定性。由于生产者–消费者问题是相互合作的进程关系的一种抽象，例如，在输入时，输入进程是生产者，计算进程是消费者；而在输出时，则计算进程是生产者，而打印进程是消费者，因此，该问题有很大的代表性及实用价值。

1. 利用记录型信号量解决生产者–消费者问题

问题描述：

- 通过一个有n个缓冲区的公用缓冲池把一群生产者p1,p2…,pm，和一群消费者q1,q2,…,qn联系起来。
- 只要缓冲池未满，生产者就可以把消息送入缓冲池；
- 只要缓冲池未空，消费者就可以从缓冲池中取走消息。


放消息
取消息
P
Q
i
j
n
n个缓冲区
(Buffer)

问题分析：

- 为解决生产者消费者问题，应该设两个同步信号量，一个说明空缓冲区的数量，用empty表示，初值为缓冲池中空缓冲区的个数n，另一个说明已存放消息缓冲区的数量，用full表示，初值为 0 。
- 由于在此问题中有M个生产者和N个消费者，它们在执行生产活动和消费活动中要对公用缓冲池进行操作。由于公用缓冲池是一个临界资源，必须互斥使用，所以，另外还需要设置一个互斥信号量mutex，其初值为 1 。

算法如下：

```
semaphore mutex=1, empty=n, full=0; ■
item  buffer[n]; ■
int in=0, out=0; ■
void proceducer(){ ■
do ■ {
        ✔ ... ■
          producer an item nextp; ■
           ✔ ... ■
          wait(empty); ■
          wait(mutex); ■
          buffer[in]=nextp; ■
          in=(in+1) % n; ■
          signal(mutex); ■
          signal(full); ■
          }while(1); ■
} ■
```

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

在生产者–消费者问题中应注意：首先，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； 其次，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；最后，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. 利用AND信号量解决生产者—消费者问题

```
semaphore mutex=1, empty=n, full=0; ■
item buffer[n] ; ■
int   in=0 out =0; ■
void producer() { ■
  do { ■ ... ■
          produce an item in nextp; ■
           ✔ ... ■
          Swait(empty, mutex); ■
          buffer[in]  =nextp; ■
          in=(in+1)% n; ■
          Ssignal(mutex, full); ■
         }while(1); ■
} ■
```

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

2.6.2 哲学家进餐问题

- 有五个哲学家围坐在一圆桌旁，桌中央有一盘通心粉，每人面前有一只空盘子，每两人之间放一只筷子。
- 每个哲学家的行为是思考，感到饥饿，然后吃通心粉。
- 为了吃通心粉，每个哲学家必须拿到两只筷子，并且每个人只能直接从自己的左边或右边去取筷子。

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： ■

semaphore chopstick[5]={1,1,1,1,1};□

所有信号量均被初始化为1， 第i位哲学家的活动可描述为：

```
do { ■
    wait(chopstick[i]); ■
    wait(chopstick[(i+1) % 5[); ■
        ✔ ... ■
    eat; ■
        ✔ ... ■
    signal(chopstick[i]); ■
    signal(chopstick[(i+1) % 5]); ■
    ✔        ... ■
    think; ■
} while(1);
```

分析：以上解法会出现死锁。

为防止死锁发生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。

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};□

do  {
    think; ■
    Sswait(chopstick[(i+1) % 5], chopstick[i]); ■
    eat; ■
    Ssignat(chopstick[(i+1) % 5], chopstick [i]); ■
  }while(1); ■
```

2.5.3 读者-写者问题

有两组并发进程:

读者和写者,共享一组数据区。

要求:

允许多个读者同时执行读操作。

不允许读者、写者同时操作。

不允许多个写者同时操作。

第一类：读者优先

如果读者来：

1）无读者、写者，新读者可以读

2）有写者等，但有其它读者正在读，则新读者也可以读

3）有写者写，新读者等

如果写者来：

1）无读者，新写者可以写

2）有读者，新写者等待

3）有其它写者，新写者等待

1. 利用记录型信号量解决读者-写者问题 ■

分析：

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。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。又因为Readcount是一个可被多个Reader进程访问的临界资源，因此，应该为它设置一个互斥信号量rmutex。

- 设有两个互斥信号量Wmutex =1，**rmutex=1**
- 另设一个全局变量**readcount =0**
- Wmutex用于读者和写者、写者和写者之间的互斥
- **Readcount**表示正在读的读者数目
- **rmutex**用于对**readcount** 这个临界资源的互斥访问

算法如下：

读者-写者问题可描述如下： ■

```
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(1);   } ■
 void writer() { ■
  do  { ■
        wait(wmutex); ■
        perform write operation; ■
        signal(wmutex); ■
       }while(1); ■ } ■
void main()  { ■
    cobegin
       reader();  writer();
   coend   }
```

【思考题】写者优先

- 上述算法是正确的，但是对写者不公平，存在进程饥饿现象。
- 修改以上读者写者问题的算法，使之对写者优先，即一旦有写者到达，后续的读者必须必须等待，无论是否有读者在读。

- 提示，增加一个信号量，用于在写者到达后封锁后续的读者

2. 利用信号量集机制解决读者-写者问题

增加一个限制条件：同时读的“读者”最多RN个。

L：表示还允许进入临界区的读者数，初值为**RN**。

mx：开关，用于控制读者与写者、写者与写者互斥访问临界区，　　初值为**1**。

```
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(1)  ;       }
```

```
 void writer()  { ■
  do  { ■
        Swait(mx,1,1; L,RN,0); ■
        perform write operation; ■
        Ssignal(mx,1); ■
      }while(1); ■
}
void main()  { ■
   cobegin
     reader();  writer();
  coend   }
```

2.6 进 程 通 信

所谓进程通信是指进程之间可直接以较高的效率传递较多数据的信息交换方式。

信号量机制是有效的同步工具，但进程间的同步原语只能传递简单的信号量信息，属于低级通讯，存在效率低、对用户不透明等缺点。如果要在进程间传递大量信息则要用Send / Receive原语（高级通讯原语）。

2.6.1 进程通信的类型

1. 共享存储器系统(Shared-Memory System)

(1) 基于共享数据结构的通信方式。

诸进程公用某些数据结构，进程通过它们交换信息。如生产者-消费者问题中的有界缓冲区。

(2) 基于共享存储区的通信方式。

高级通信，在存储器中划出一块共享存储区，进程在通信前，向系统申请共享存储区中的一个分区，并指定该分区的关键字，若系统已经给其它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。接着，申请者把获得的共享存储分区连接到本进程上，此后可读写该分区。

以上两种方式的同步互斥都要由进程自己负责。

2. 消息传递系统(Message passing system)

不论是单机系统、多机系统，还是计算机网络，消息传递机制都是用得最广泛的一种进程间通信的机制。在消息传递系统中，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。操作系统隐藏了通信的实现细节，大大减化了通信程序编制的复杂性，而获得广泛的应用。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. 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。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操作系统中。

发送进程　　接收进程

字符流方式写入读出
先进先出顺序

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，再把他唤醒。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。

4. 客户-服务器系统

客户-服务器系统的通信机制主要用于不同计算机间进程的双向通信，是网络环境的主流通信实现机制。分为：套接字、远程过程调用和远程方法调用。

1）套接字

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等，是进程通信和网络通信的基本构件。套接字包括两类：

- 基于文件型
- 基于网络型

2) 远程过程调用和远程方法调用

远程过程调用是一个通信协议，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。

2.6.2 消息传递通信的实现方法

1. 直接通信方式 ■

这是指发送进程利用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。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，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。

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，接收原语可表示为：

Receive (id, message);

我们还可以利用直接通信原语，来解决生产者-消费者问题。当生产者生产出一个产品(消息)后，便用Send原语将消息发送给消费者进程；而消费者进程则利用Receive原语来得到一个消息。如果消息尚未生产出来，消费者必须等待，直至生产者进程将消息发送过来。生产者-消费者的通信过程可分别描述如下：

```
repeat ■
    ✔ ... ■
  produce an item in nextp; ■
    ✔ ... ■
  send(consumer, nextp); ■
 until false; ■
 repeat ■
  receive(producer, nextc); ■
    ✔ ... ■
  consume the item in nextc; ■
until false;
```

2. 间接通信方式

(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)；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；

信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。据此，可把信箱分为以下三类。■

1) 私用信箱 ■

用户进程可为自己建立一个新信箱，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。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。

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。这时可为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 消息传递系统实现中的若干问题

1. 通信链路(communication link) ■

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，也用显式方式拆除链路。

这种方式主要用于计算机网络中。第二种方式是发送进程无须明确提出建立链路的请求，只须利用系统提供的发送命令(原语)，系统会自动地为之建立一条链路。这种方式主要用于单机系统中。

根据通信链路的连接方法，又可把通信链路分为两类： ① 点–点连接通信链路，这时的一条链路只连接两个结点(进程)；② 多点连接链路，指用一条链路连接多个($n>2$)结点(进程)。而根据通信方式的不同，则又可把链路分成两种： ① 单向通信链路，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

在某些OS中，消息是采用比较短的定长消息格式，这减少了对消息的处理和存储开销。这种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；但这对要发送较长消息的用户是不方便的。在有的OS中，采用另一种变长的消息格式，即进程所发送消息的长度是可变的。系统在处理和存储变长消息时，须付出更多的开销，但方便了用户。 这两种消息格式各有其优缺点，故在很多系统(包括计算机网络)中，是同时都用的。

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。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。它可描述如下： ■

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

(2) PCB中有关通信的数据项。在利用消息缓冲队列通信机制时，在设置消息缓冲队列的同时，还应增加用于对消息队列进行操作和实现同步的信号量，并将它们置入进程的PCB中。在PCB中应增加的数据项可描述如下：

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 ■

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，见图 2 - 12 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内部标识符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。 ■

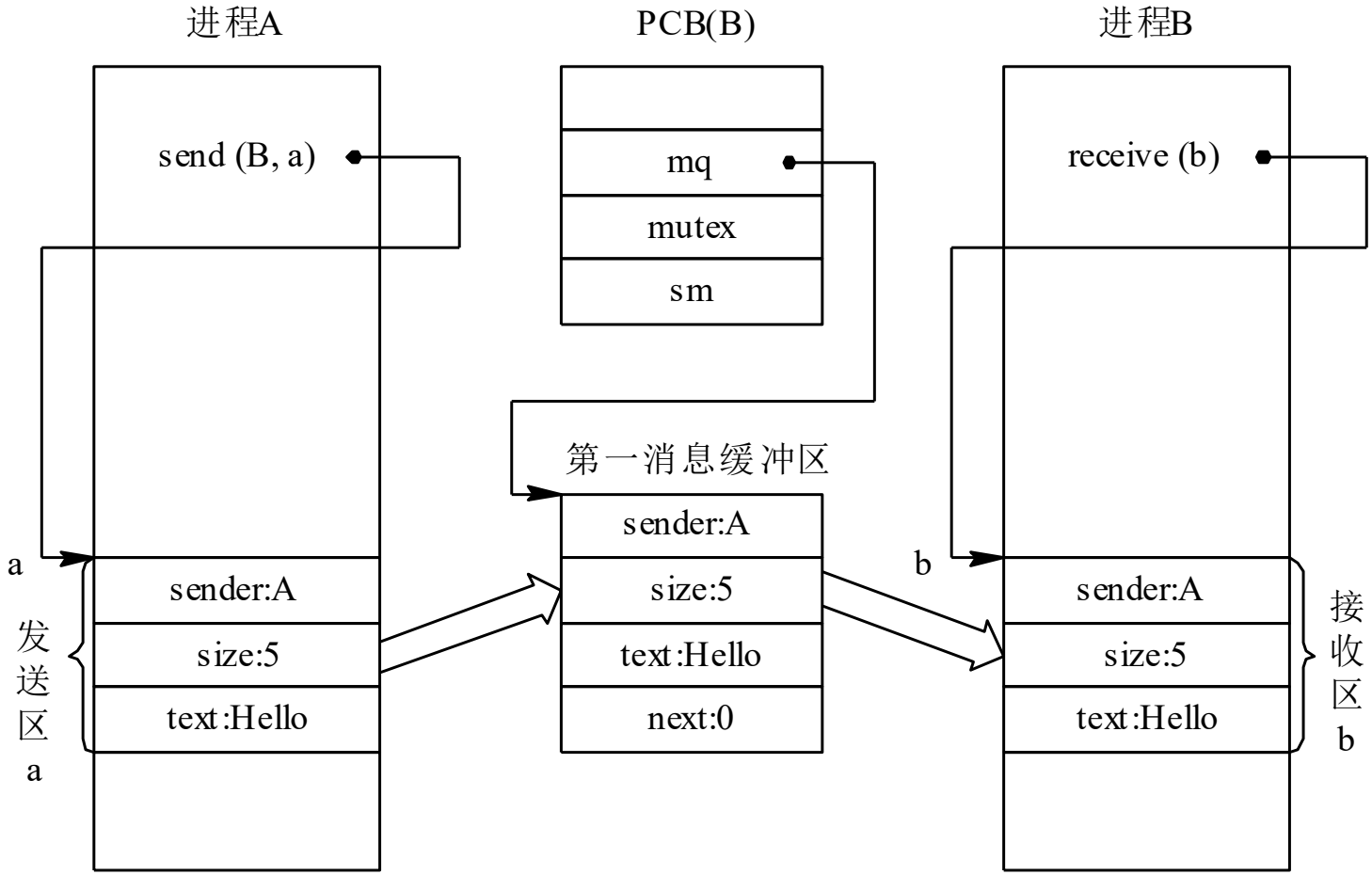

进程A
PCB(B)
进程B
send (B, a)
mq
mutex
sm
receive (b)
第一消息缓冲区
sender:A
size:5
text:Hello
next:0
a
发送区a
sender:A
size:5
text:Hello
b
sender:A
size:5
text:Hello
接收区b

图 2 - 12 消息缓冲通信

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

3. 接收原语

接收原语描述如下： ■

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

2.7 线 程的基本概念

2.7.1 线程的引入

引入进程的目的是为了使多个程序并发执行，以改善资源利用率、提高系统吞吐量。

引入线程则是为了减少程序并发执行时的所付出的时空开销。

进程的两个基本属性：

1. 进程是一个可拥有资源的基本单位。
2. 进程同时又是一个可独立调度和分派的基本单位。

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 ■

进程作为一个资源拥有者，在创建、撤消、切换中，系统必须为之付出较大时空开销。所以系统中进程的数量不宜过多，进程切换的频率不宜过高，但这也就限制了并发程度的进一步提高。

为解决此问题，人们想到将进程的上述两个属性分开，即对作为调度和分派的基本单位，不同时作为独立分配资源的单位；对拥有资源的单位，不对之进行频繁切换。

线程因而产生。

- 在引入线程的OS中，线程是进程中的一个实体，是被系统独立调度和分派的基本单位。
- 线程自己基本不拥有系统资源，只拥有少量必不可少的资源：程序计数器、一组寄存器、栈。
- 它可与同属一个进程的其它线程共享进程所拥有的全部资源。
- 一个线程可以创建和撤消另一个线程；同一进程中的多个线程之间可以并发执行。

引入线程的好处：

- 创建一个新线程花费时间少（结束亦如此）。
- 两个线程的切换花费时间少。
- 因为同一进程内的线程共享内存和文件，因此它们之间相互通信无须调用内核。
- 适合多处理机系统。

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。

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， 它包括程序计数器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。

(2) 线程运行状态。

如同传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。 相应地，线程在运行时，也具有下述三种基本状态：① 执行状态，表示线程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 ■

在多线程OS环境下，应用程序在启动时，通常仅有一个线程在执行，该线程被人们称为“初始化线程”。它可根据需要再去创建若干个线程。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。在线程创建函数执行完后，将返回一个线程标识符供以后使用。 ■

终止线程的方式有两种：一种是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，至少一个。

(3) 进程不是一个可执行的实体。

2.7.2 线程与进程的比较：

1、调度

在传统OS中，拥有资源、独立调度和分派的基本单位都是进程，在引入线程的系统中，线程是调度和分派的基本单位，而进程是拥有资源的基本单位。

在同一个进程内线程切换不会产生进程切换，由一个进程内的线程切换到另一个进程内的线程时，将会引起进程切换。

2、并发性

在引入线程的系统中，进程之间可并发，同一进程内的各线程之间也能并发执行。因而系统具有更好的并发性。

3、拥有资源

无论是传统OS，还是引入线程的OS，进程都是拥有资源的独立单位，线程一般不拥有系统资源，但它可以访问隶属进程的资源。即一个进程的所有资源可供进程内的所有线程共享。

4、系统开销

进程的创建和撤消的开销要远大于线程创建和撤消的开销，进程切换时，当前进程的CPU环境要保存，新进程的CPU环境要设置，线程切换时只须保存和设置少量寄存器，并不涉及存储管理方面的操作，可见，进程切换的开销远大于线程切换的开销。

同时，同一进程内的各线程由于它们拥有相同的地址空间，它们之间的同步和通信的实现也变得比较容易。

2.7.3 线程间的同步和通信

1. 互斥锁(mutex) ■

互斥锁是一种比较简单的、用于实现进程间对资源互斥访问的机制。由于操作互斥锁的时间和空间开锁都较低，因而较适合于高频度使用的关键共享数据和程序段。互斥锁可以有两种状态， 即开锁(unlock)和关锁(lock)状态。相应地，可用两条命令(函数)对互斥锁进行操作。其中的关锁lock操作用于将mutex关上，开锁操作unlock则用于打开mutex。

2. 条件变量

每一个条件变量通常都与一个互斥锁一起使用，亦即，在创建一个互斥锁时便联系着一个条件变量。单纯的互斥锁用于短期锁定，主要是用来保证对临界区的互斥进入。而条件变量则用于线程的长期等待， 直至所等待的资源成为可用的。

线程首先对mutex执行关锁操作，若成功便进入临界区，然后查找用于描述资源状态的数据结构，以了解资源的情况。只要发现所需资源R正处于忙碌状态，线程便转为等待状态，并对mutex执行开锁操作后，等待该资源被释放； 若资源处于空闲状态，表明线程可以使用该资源，于是将该资源设置为忙碌状态，再对mutex执行开锁操作。

下面给出了对上述资源的申请(左半部分)和释放(右半部分)操作的描述。

```
Lock mutex                          Lock mutex ■
check data structures;              mark resource as free; ■
while(resource busy);               unlock mutex; ■
wait(condition variable);           wakeup(condition variable); ■
mark resource as busy; ■
unlock mutex;
```

3. 信号量机制

(1) 私用信号量(private samephore)。

当某线程需利用信号量来实现同一进程中各线程之间的同步时，可调用创建信号量的命令来创建一私用信号量，其数据结构是存放在应用程序的地址空间中。私用信号量属于特定的进程所有，OS并不知道私用信号量的存在，因此，一旦发生私用信号量的占用者异常结束或正常结束，但并未释放该信号量所占有空间的情况时，系统将无法使它恢复为0(空)， 也不能将它传送给下一个请求它的线程。 ■

(2) 公用信号量(public semaphort)。

公用信号量是为实现不同进程间或不同进程中各线程之间的同步而设置的。由于它有着一个公开的名字供所有的进程使用，故而把它称为公用信号量。其数据结构是存放在受保护的系统存储区中，由OS为它分配空间并进行管理，故也称为系统信号量。如果信号量的占有者在结束时未释放该公用信号量，则OS会自动将该信号量空间回收，并通知下一进程。可见，公用信号量是一种比较安全的同步机制。

2.8 线程的实现

2.8.1 线程的实现方式

1. 内核支持线程

这里所谓的内核支持线程，也都同样是在内核的支持下运行的，即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，也是依靠内核实现的。此外，在内核空间还为每一个内核支持线程设置了一个线程控制块， 内核是根据该控制块而感知某线程的存在的，并对其加以控制。

2. 用户级线程 ■

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。对于用户级线程的切换，通常是发生在一个应用进程的诸多线程之间，这时，也同样无须内核的支持。由于切换的规则远比进程调度和切换的规则简单，因而使线程的切换速度特别快。可见，这种线程是与内核无关的。

2.8.2 线程的实现

1. 内核支持线程的实现

PTDA　进程资源

TCB # 1

TCB # 2

TCB # 3

图 2 - 13 任务数据区空间

2. 用户级线程的实现

1) 运行时系统(Runtime System) ■

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

- 核心不知道线程的存在。
- 线程切换不需要核心态特权。
- 调度是应用特定的 。

用户级线程的优点和缺点

优点：

- 线程切换不调用核心
- 调度是应用程序特定的：可以选择最好的算法
- ULT可运行在任何操作系统上（只需要线程库），可以在一个不支持线程的OS上实现

缺点：

- 大多数系统调用是阻塞的，因此核心阻塞进程，故进程中所有线程将被阻塞
- 核心只将处理器分配给进程，同一进程中的两个线程不能同时运行于两个处理器上。

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。

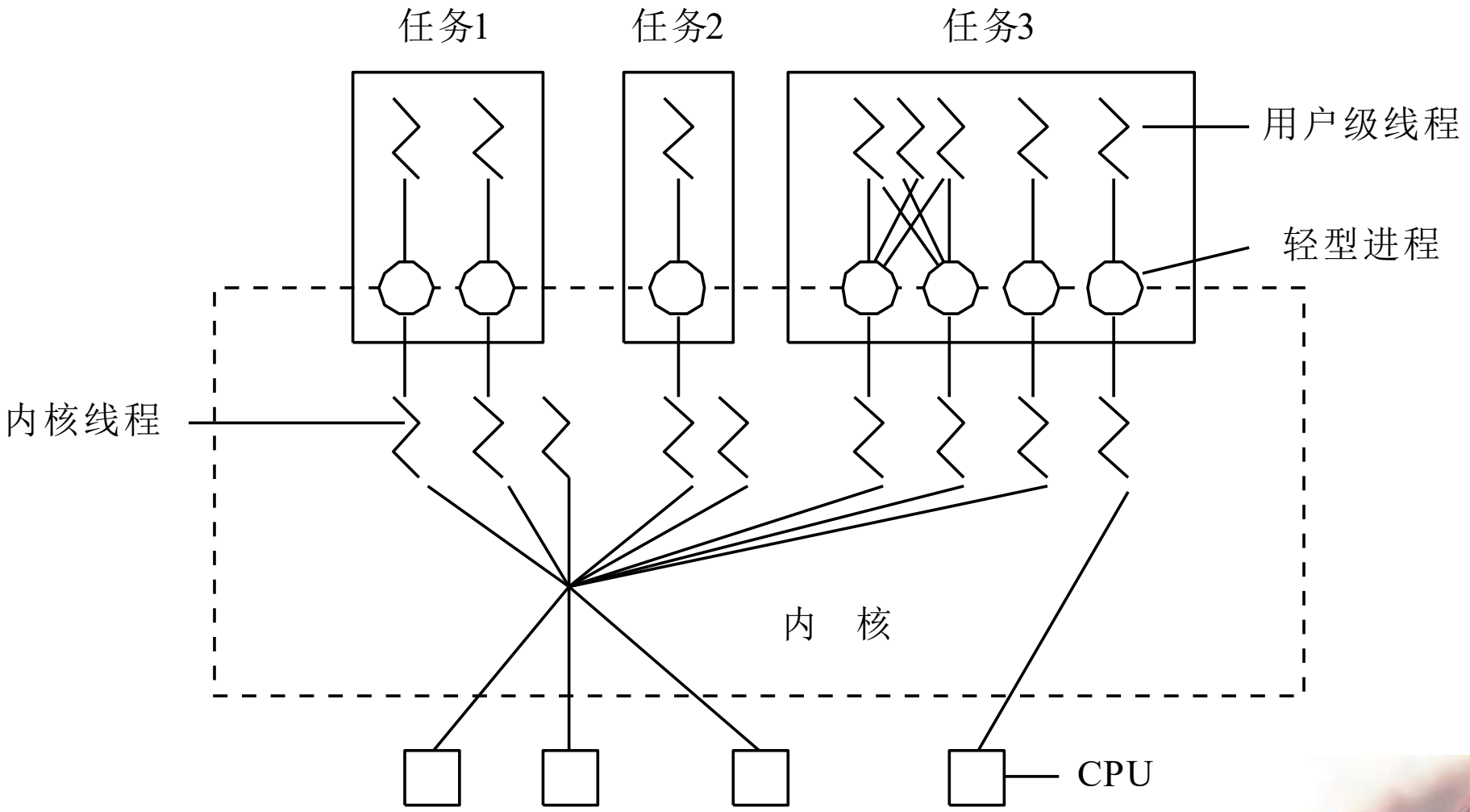


图 2 - 14 利用轻型进程作为中间系统

内核支持线程的优点和缺点：

优点：

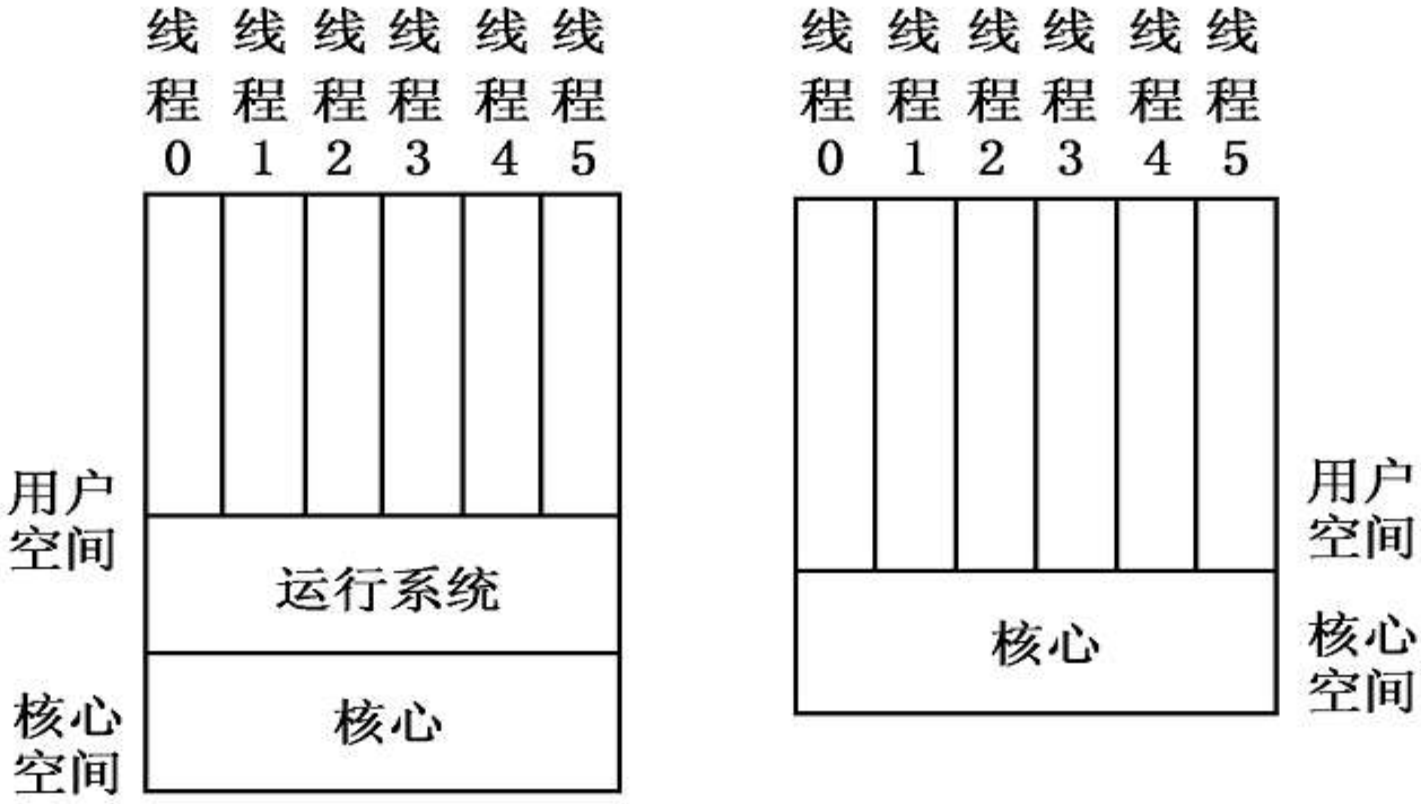
对多处理器，核心可以同时调度同一进程的多个线程

阻塞是在线程一级完成。

缺点：

在同一进程内的线程切换调用内核，导致速度下降。

图：用户级和核心级线程


线程0
线程1
线程2
线程3
线程4
线程5
用户空间
运行系统
核心空间
核心
(a) 用户级线程
线程0
线程1
线程2
线程3
线程4
线程5
用户空间
核心
核心空间
(b) 核心级线程

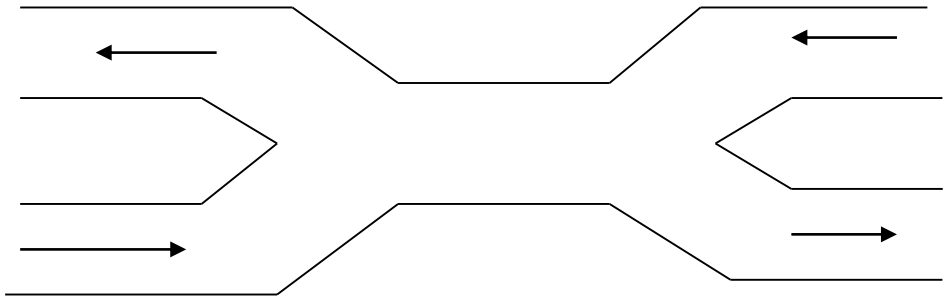
补充作业:

1.设有两个优先级相同的进程P1与P2，令信号量S1、S2的初值为0，已知Z=2，试问P1、P2并发运行后x=？ y=？ z=？

进程P1：
```
y=1;
y=y+2;
signal(s1);
z=y+1;
wait(s2);
y=y+z;
```

进程P2：
```
x=1;
x=x+1;
wait(s1);
x=x+y;
signal(s2);
z=z+x;
```

2.有桥如下图所示，车流方向如箭头所示，假设桥上不允许两车交会，但允许同方向多辆车依次通过（即桥上可有多个相同方向行驶的车辆），试用wait和signal操作实现桥上的交通管理。



3.某银行人民币储蓄业务由 n个柜员负责。每个顾客进入银行后先取一个号，并等着叫号。当一个柜员空闲时，就叫下一个号，试用wait和signal操作正确编写柜员和顾客进程。