第二章　进 程 管 理

- 2.1　进程的基本概念
- 2.2　进程控制
- 2.3　进程同步
- 2.4　经典进程的同步问题
- 2.5　进程通信
- 2.6　线程
- 2.7　openEular中的进程树以及线程的实现

2.1 进程的基本概念

- 2.1.1　程序的顺序执行及其特征
- 1. 程序的顺序执行

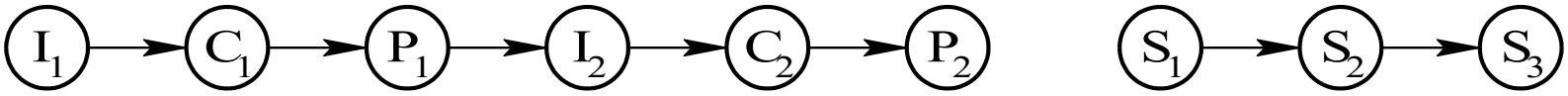
通常可以把一个应用程序分成若干个程序段，在未配置操作系统时，在各程序段之间，必须按照某种先后次序顺序执行，仅当前一操作(程序段)执行完后，才能执行后继操作。例如，在进行计算时，总须先输入用户的程序和数据，然后进行计算，最后才能打印计算结果。

2.1.1 程序的顺序执行及其特征

S1: a:=x+y；

S2: b:=a-5；

S3: 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

2.1.1 程序的顺序执行及其特征

- 顺序性：处理机的操作严格按照程序所规定的顺序执行，即每一操作必须在上一个操作结束之后开始。
- 封闭性：程序是在封闭的环境下执行的，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。
- 可再现性：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，都将获得相同的结果。

程序顺序执行时的特性，为程序员检测和校正程序的错误带来了很大的方便。

2.1.3 程序的并发执行及其特征

- 1．程序并发执行

- 并发程序:是指两道或两道以上程序同时装入内存中运行，这些程序的执行在时间上互相有重叠，即在一个程序执行结束之前，另一个程序已经开始执行。

- 特征:

  1) 间断性

  2) 失去封闭性

  3) 不可再现性

2.1.3 程序的并发执行及其特征

- 2．程序并发执行时的特征

1) 间断性

程序在并发执行时，由于它们共享系统资源，以及为完成同一项任务而相互合作，致使在这些并发执行的程序之间，形成了相互制约的关系。相互制约将导致并发程序具有“执行—暂停—执行”这种间断性的活动规律。

2.1.3 程序的并发执行及其特征

2) 失去封闭性

程序在并发执行时，是多个程序共享系统中的各种资源，因而这些资源的状态将由多个程序来改变，致使程序的运行失去了封闭性。这样，某程序在执行时，必然会受到其它程序的影响。例如，当处理机这一资源已被某个程序占有时，另一程序必须等待。

2.1.3 程序的并发执行及其特征

3) 不可再现性

程序在并发执行时，由于失去了封闭性，也将导致其再失去可再现性。例如，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，都要执行Print(N)操作，然后再将N置成“0”。程序A和B以不同的速度运行。这样，可能出现下述三种情况。

2.1.3 程序的并发执行及其特征

A	(假定某时刻变量N的值为n)	B
…		…
N=N+1;		Print(N);
…		N:=0;
…		…

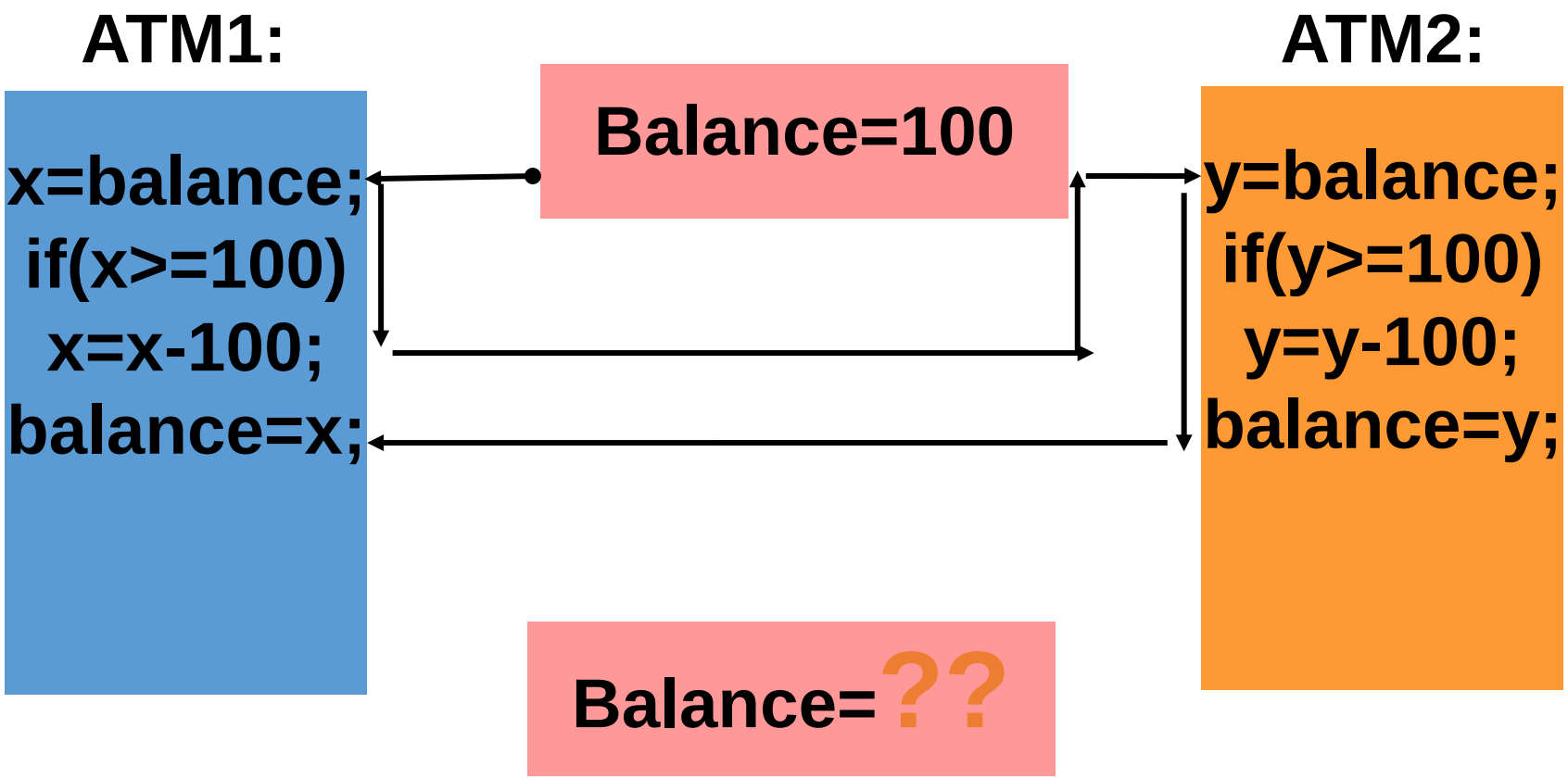
(1) N:=N+1在Print(N)和N:=0之前，此时得到的N值分别为n+1，n+1，0。

(2) N:=N+1在Print(N)和N:=0之后，此时得到的N值分别为n，0，1。

(3) N:=N+1在Print(N)和N:=0之间，此时得到的N值分别为n，n+1，0。

上述情况说明，程序在并发执行时，由于失去了封闭性，其计算结果已与并发程序的执行速度有关，从而使程序的执行失去了可再现性，亦即，程序经过多次执行后，虽然它们执行时的环境和初始条件相同，但得到的结果却各不相同。

2.1.3 程序的并发执行及其特征



ATM example

2.1.4 进程的特征与状态

- 1. 进程的特征和定义

在多道程序环境下，程序的执行属于并发执行，此时它们将失去其封闭性，并具有间断性及不可再现性的特征。为使程序能并发执行，且为了对并发执行的程序加以描述和控制，人们引入了“进程”的概念。

2.1.4 进程的特征与状态

1） 进程的定义

典型的进程定义有：

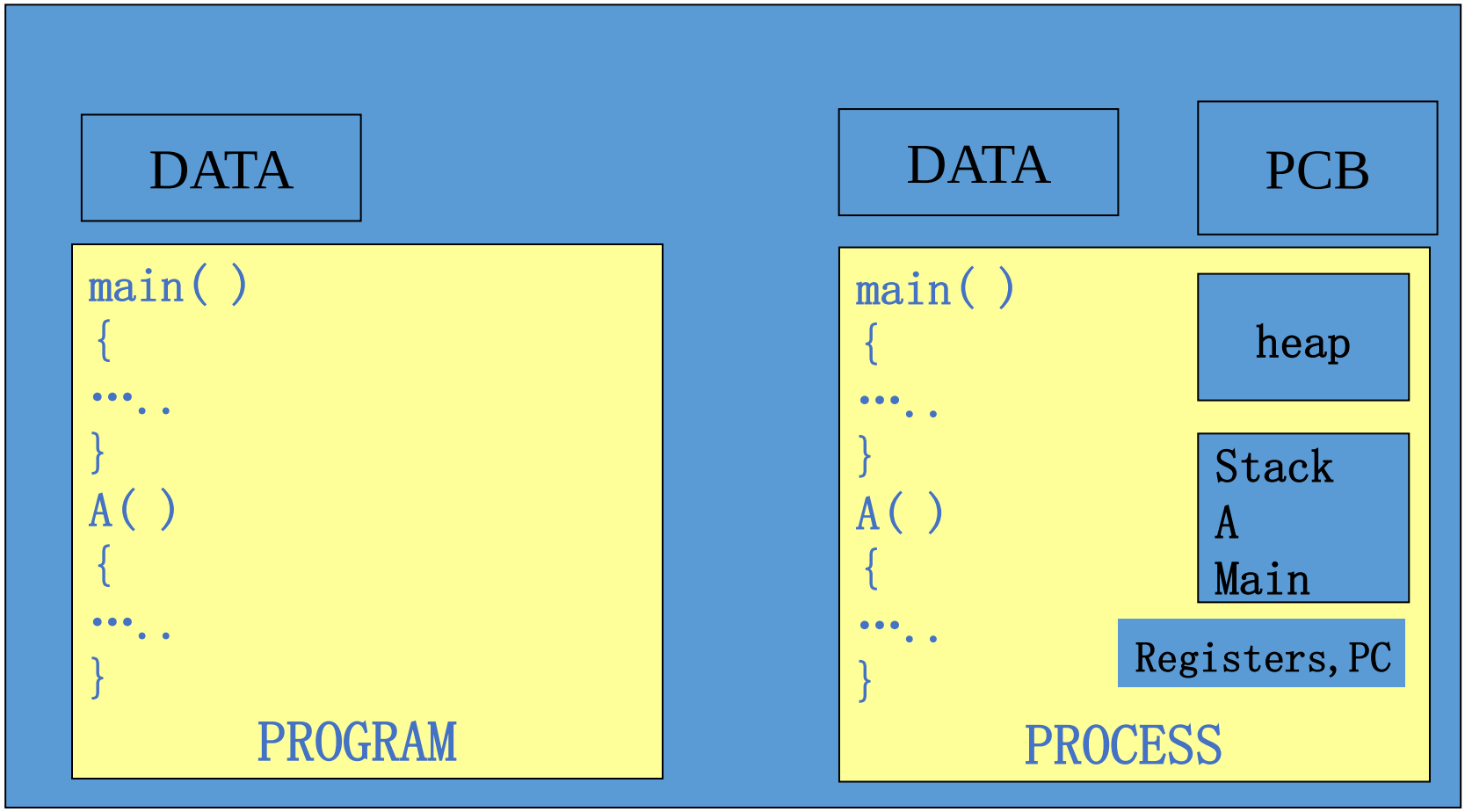
(1) 进程是程序的一次执行。

(2) 进程是一个程序及其数据在处理机上顺序执行时所发生的活动。

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

2.1.4 进程的特征与状态

进程与程序实体



2.1.4 进程的特征与状态

2） 进程特征

① 结构性：进程包括可执行的程序代码、程序的数据和堆栈、程序计数器、堆栈指针、有关寄存器、以及所有运行程序所必须的其它信息。

② 动态性：进程是程序执行的过程。

③ 并发性：进程使程序能并发执行。

④ 独立性：进程是独立运行的基本单位，也是系统资源分配与调度的独立单位。

⑤ 异步性：进程以各自独立的、不可预知的速度向前推进。

2.1.4 进程的特征与状态

① 结构性

通常的程序是不能并发执行的。为使程序(含数据)能独立运行，应为之配置一进程控制块，即PCB(Process Control Block)；而由程序段、相关的数据段和PCB三部分便构成了进程实体。所谓创建进程，实质上是创建进程实体中的PCB；而撤消进程，实质上是撤消进程的PCB。

2.1.4 进程的特征与状态

② 动态性

进程的实质是进程实体的一次执行过程，因此，动态性是进程的最基本的特征。动态性还表现在：“它由创建而产生，由调度而执行，由撤消而消亡”。可见，进程实体有一定的生命期，而程序则只是一组有序指令的集合，并存放于某种介质上，本身并不具有运动的含义，因而是静态的。

③ 并发性

这是指多个进程实体同存于内存中，且能在一段时间内同时运行。并发性是进程的重要特征，同时也成为OS的重要特征。引入进程的目的也正是为了使其进程实体能和其它进程实体并发执行；而程序(没有建立PCB)是不能并发执行的。

2.1.4 进程的特征与状态

④ 独立性

在传统的OS中，独立性是指进程实体是一个能独立运行、独立分配资源和独立接受调度的基本单位。凡未建立PCB的程序都不能作为一个独立的单位参与运行。

⑤ 异步性

这是指进程按各自独立的、 不可预知的速度向前推进，或说进程实体按异步方式运行。

2.1.4 进程的特征与状态

➢ 2. 进程的三种基本状态

- 运行中的进程可能具有以下三种基本状态：

  1) 就绪状态(Ready)

  2) 执行状态（Executing)

  3) 阻塞状态(Blocked)

- 当前还有许多系统中进程具有挂起状态（Suspend)

2.1.4 进程的特征与状态

1) 就绪(Ready)状态

当进程已分配到除CPU以外的所有必要资源后，只要再获得CPU，便可立即执行，进程这时的状态称为就绪状态。在一个系统中处于就绪状态的进程可能有多个，通常将它们排成一个队列，称为就绪队列。

2.1.4 进程的特征与状态

2) 执行状态

进程已获得CPU，其程序正在执行。在单处理机系统中，只有一个进程处于执行状态； 在多处理机系统中，则有多个进程处于执行状态。

3) 阻塞状态

正在执行的进程由于发生某事件而暂时无法继续执行时，便放弃处理机而处于暂停状态，把这种暂停状态称为阻塞状态，有时也称为等待状态或封锁状态。

致使进程阻塞的典型事件有：请求I/O，申请缓冲空间等。通常将这种处于阻塞状态的进程也排成一个队列。有的系统则根据阻塞原因的不同而把处于阻塞状态的进程排成多个队列。

2.1.4 进程的特征与状态

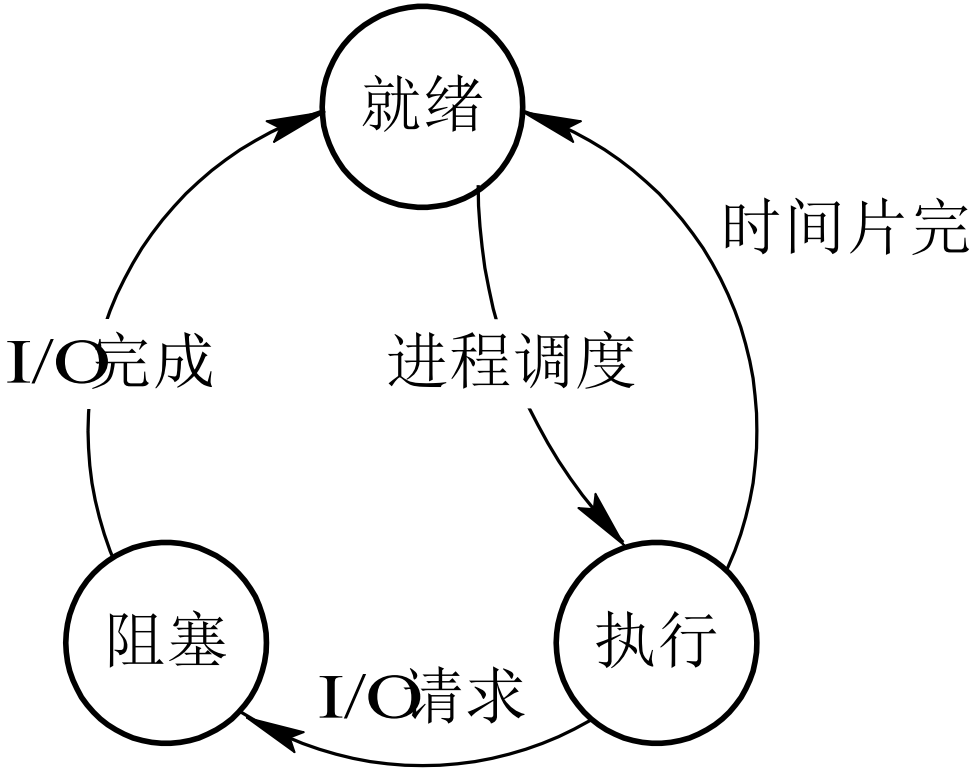


图2-5 进程的三种基本状态及其转换

2.1.4 进程的特征与状态

➢ 3. 挂起状态

挂起状态是指某进程在计算机操作系统中暂时被调离出内存的状况。

1) 引入挂起状态的原因

(1) 终端用户的请求。当终端用户在自己的程序运行期间希望暂时使自己的程序静止下来，以便用户研究其执行情况或对程序进行修改。

2.1.4 进程的特征与状态

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

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

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

2.1.4 进程的特征与状态

2) 具有挂起状态的进程状态转换

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

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

2.1.4 进程的特征与状态

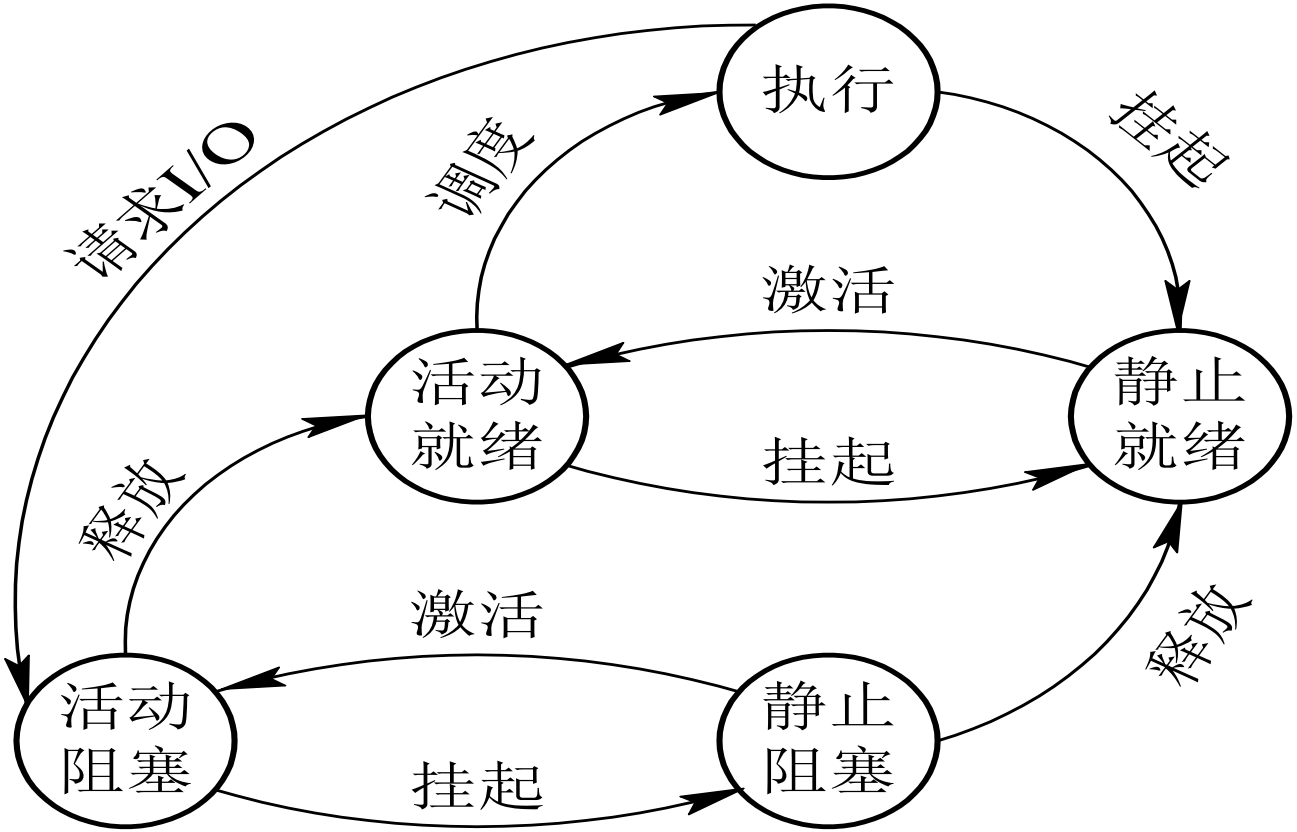


图 2-6　具有挂起状态的进程状态图

2.1.4 进程的特征与状态

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

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

(4) 静止阻塞→活动阻塞。处于Blockeds状态的进程，若用激活原语Active激活后，该进程将转变为Blockeda状态。

(5) 静止阻塞→静止就绪。进程在静止阻塞期间，等待事件发生，而被释放。

2.1.4 进程的特征与状态

- 4．创建状态和终止状态

1) 创建状态

创建一个进程一般要通过两个步骤：首先，为一个新进程创建PCB，并填写必要的管理信息；其次，把该进程转入就绪状态并插入就绪队列之中。

当一个新进程被创建时，系统已为其分配了PCB，填写了进程标识等信息，但由于该进程所必需的资源或其它信息，如主存资源尚未分配等，一般而言，此时的进程已拥有了自己的PCB，但进程自身还未进入主存，即创建工作尚未完成，进程还不能被调度运行，其所处的状态就是**创建状态**。

2.1.4 进程的特征与状态

2) 终止状态

1. 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。
2. 进程的终止也要通过两个步骤：首先等待操作系统进行善后处理，然后将其PCB清零，并将PCB空间返还系统。
3. 进入终止态的进程以后不能再执行，但在操作系统中依然保留一个记录，其中保存状态码和一些计时统计数据，供其它进程收集。
4. 一旦其它进程完成了对终止状态进程的信息提取之后，操作系统将删除该进程。

2.1.4 进程的特征与状态

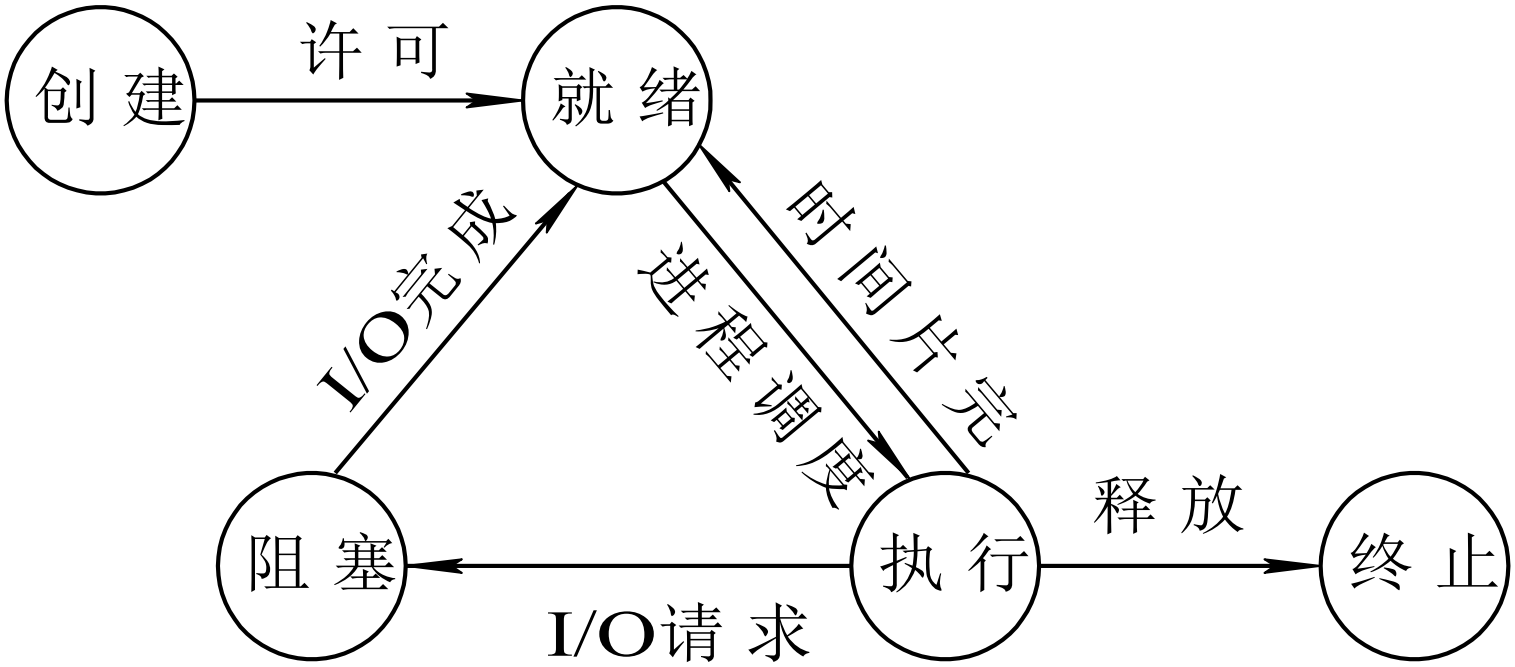


图2-7 进程的五种基本状态及转换

2.1.4 进程的特征与状态

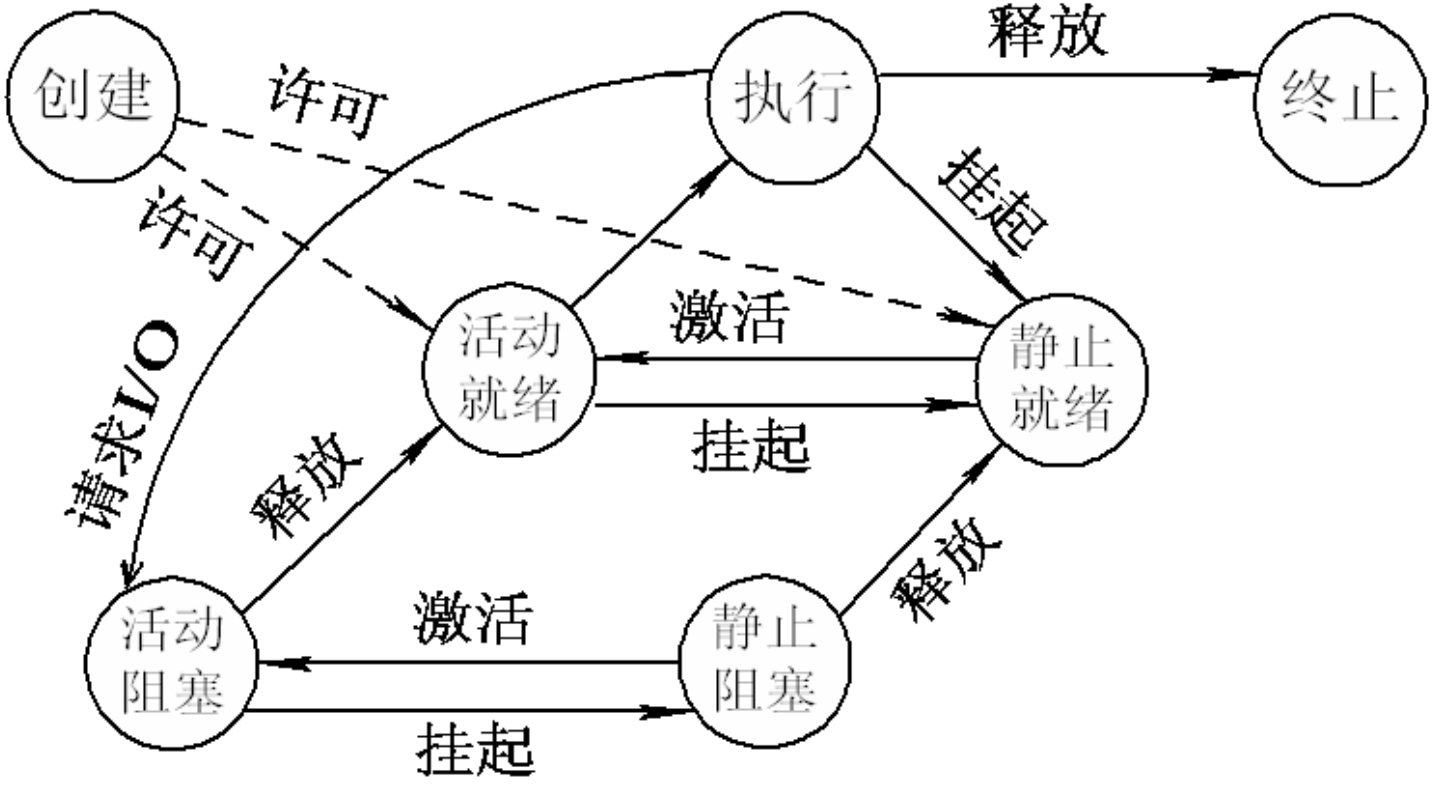


图2-8 具有创建、终止和挂起状态的进程状态图

2.1.4 进程的特征与状态

如图2-8所示，引进创建和终止状态后，在进程状态转换时，相比较图2-7所示的进程五状态转换而言，需要增加考虑下面的几种情况。

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

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

2.1.5 进程控制块 (PCB, Process Control Block)

1．进程控制块的作用

- 为了描述和控制进程的运行，系统为每个进程定义了一个数据结构——进程控制块PCB，它是进程实体的一部分，是操作系统中最重要的记录型数据结构。
- PCB中记录了操作系统所需的、用于描述进程的当前情况以及控制进程运行的全部信息。
- 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。

2.1.5 进程控制块 (PCB, Process Control Block)

1.系统是根据进程的PCB而感知到该进程的存在的，PCB是进程存在的唯一标志。

2.在进程的生命期中，系统总是通过PCB对进程进行控制的。

- 当OS要调度某进程执行时，要从该进程的PCB中查出其现行状态及优先级；
- 在调度到某进程后，要根据其PCB中所保存的处理机状态信息，设置该进程恢复运行的现场，并根据其PCB中的程序和数据的内存始址，找到其程序和数据；
- 进程在执行过程中，当需要和与之合作的进程实现同步、通信或访问文件时，也都需要访问PCB；
- 当进程由于某种原因暂停执行时，又须将其断点的处理机环境保存在PCB中。

2.1.5 进程控制块 (PCB, Process Control Block)

3. 当系统创建一个新进程时，就为它建立了一个PCB；进程结束时又回收其PCB，进程也随之消亡。
4. PCB可以并且只能被操作系统中的多个模块读或修改。如被调度程序、资源分配程序、中断处理程序以及监督和分析程序等读或修改。
5. PCB应常驻内存。因为PCB经常被系统访问，尤其是被运行频率很高的进程调度程序访问。
6. 系统将所有的PCB组织成若干个链表(或队列)，存放在操作系统中专门开辟的PCB区内。

2.1.5 进程控制块 (PCB, Process Control Block)

2．进程控制块中的信息

- 进程标识信息（进程标识符、用户名、程序名等）

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PID。在所有的操作系统中，都为每一个进程赋予了一个惟一的数字标识符。

(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。

2.1.5 进程控制块 (PCB, Process Control Block)

➢ 处理机状态信息

- 由处理机的各种寄存器中的内容组成的。处理机在运行时，许多信息都放在寄存器中。当处理机被中断时，所有这些信息都必须保存在PCB中，以便在该进程重新执行时，能从断点继续执行。包括：

  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；

  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：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。

2.1.5 进程控制块 (PCB, Process Control Block)

➢ 进程调度信息

- PCB中需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，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。

2.1.5 进程控制块 (PCB, Process Control Block)

➢ 进程控制信息

- 进程控制信息包括：

① 程序和数据的地址，指进程的程序和数据所在的内存或外存地(首)址，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，即一张列出了除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。

2.1.5 进程控制块 (PCB, Process Control Block)

进程号
进程状态
现场(每个寄存器1列)
就绪队列(即调度队列)指针
优先权
其他进程调度信息
进程等待原因
等待队列指针(注意一般与就绪队列指针为同一列)
记账信息(所用系统时间(即实际时间),CPU时间,时间限制,账号,作业号或进程号等)
页表或界址寄存器
I/O状态信息(未完成的I/O请求)
分配给该进程的I/O设备
打开文件表
...

图 2.24　PCB 表的内容

2.1.5 进程控制块 (PCB, Process Control Block)

3. 进程控制块的组织方式

1) 链接方式

把具有同一状态的PCB，用其中的链接字链接成一个队列。这样，可以形成就绪队列、若干个阻塞队列和空白队列等。对其中的就绪队列常按进程优先级的高低排列，把优先级高的进程的PCB排在队列前面。

此外，也可根据阻塞原因的不同而把处于阻塞状态的进程的PCB排成等待I/O操作完成的队列和等待分配内存的队列等。

2.1.5 进程控制块 (PCB, Process Control Block)

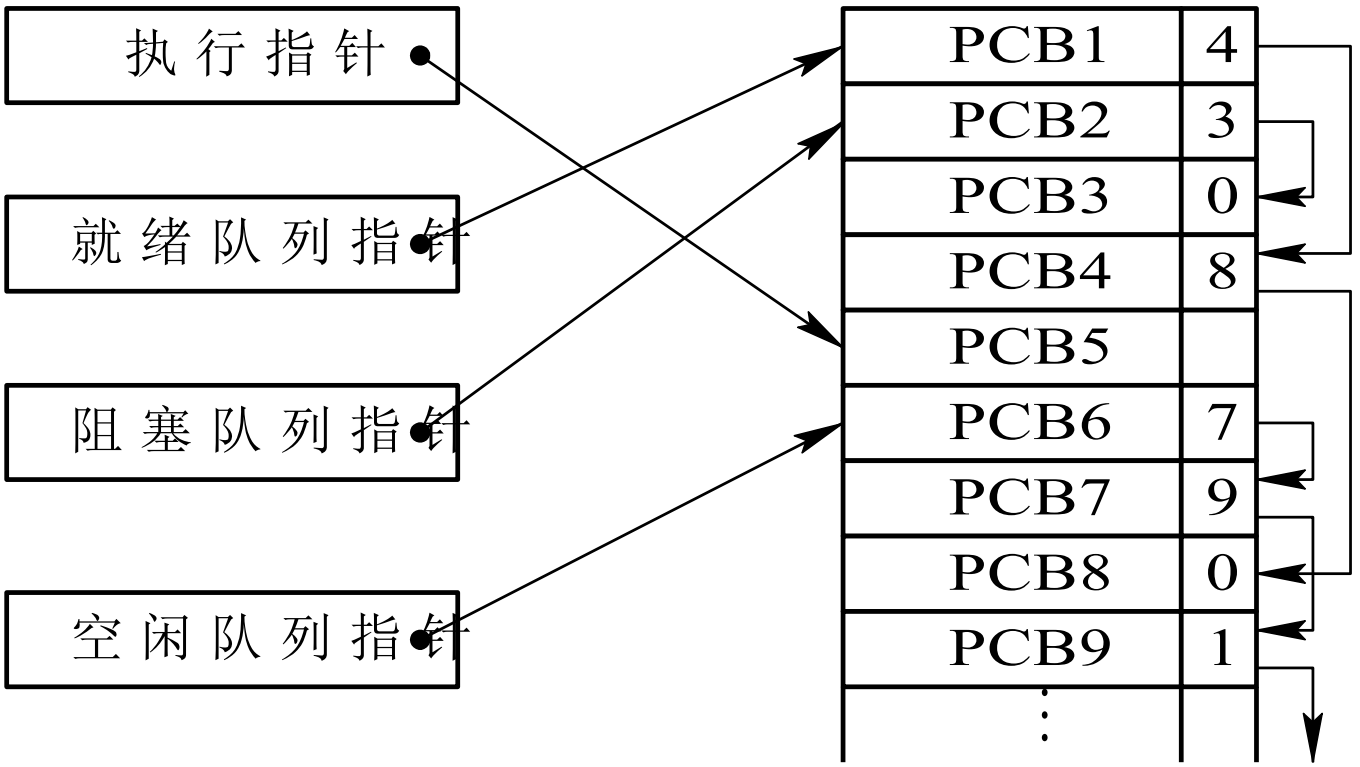


图 2-9　PCB链接队列示意图

2.1.5 进程控制块 (PCB, Process Control Block)

2) 索引方式

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

2.1.5 进程控制块 (PCB, Process Control Block)

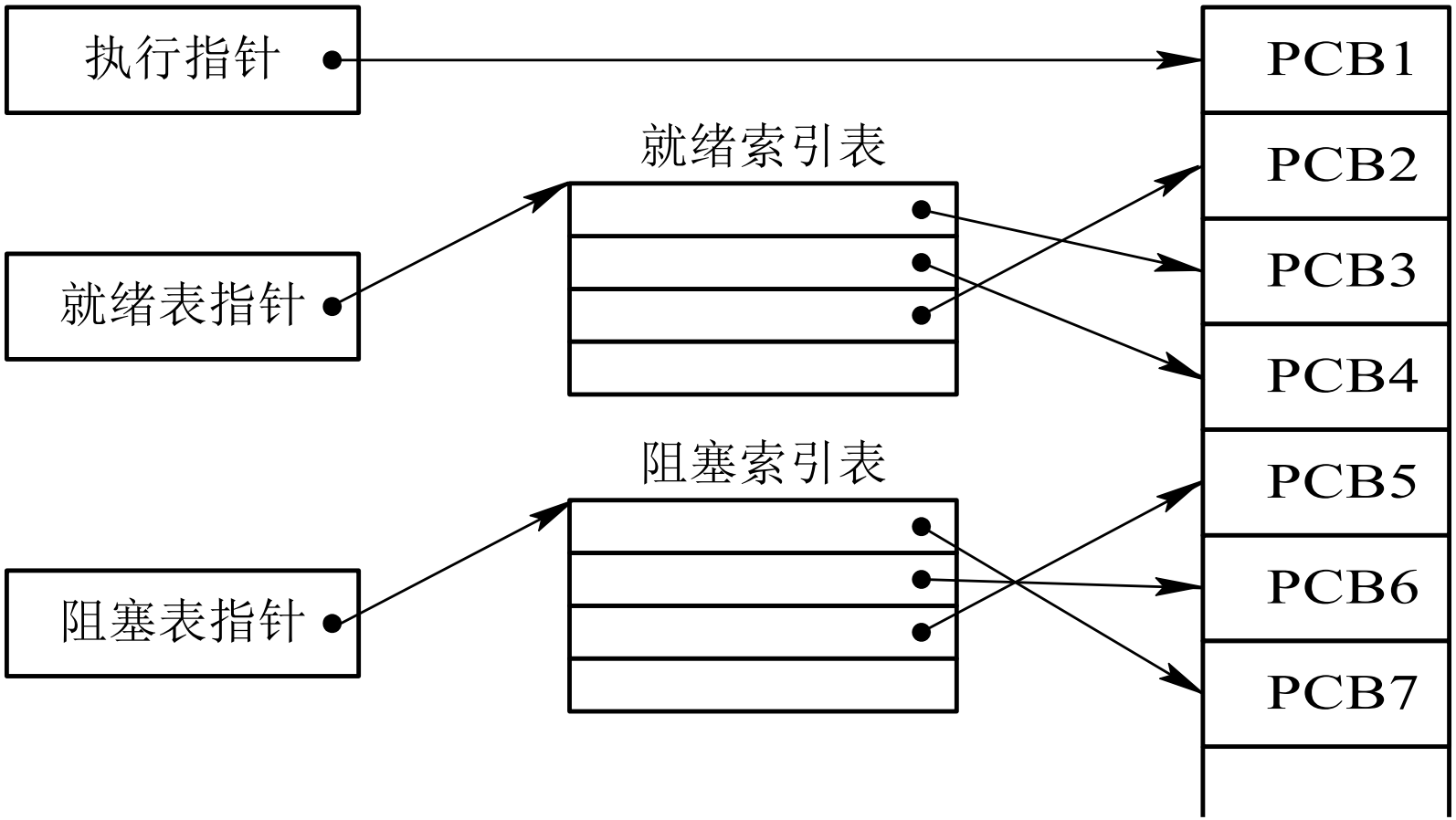


图 2-10　按索引方式组织PCB

2.2 进 程 控 制

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

进程控制一般是由OS的内核中的原语来实现的。

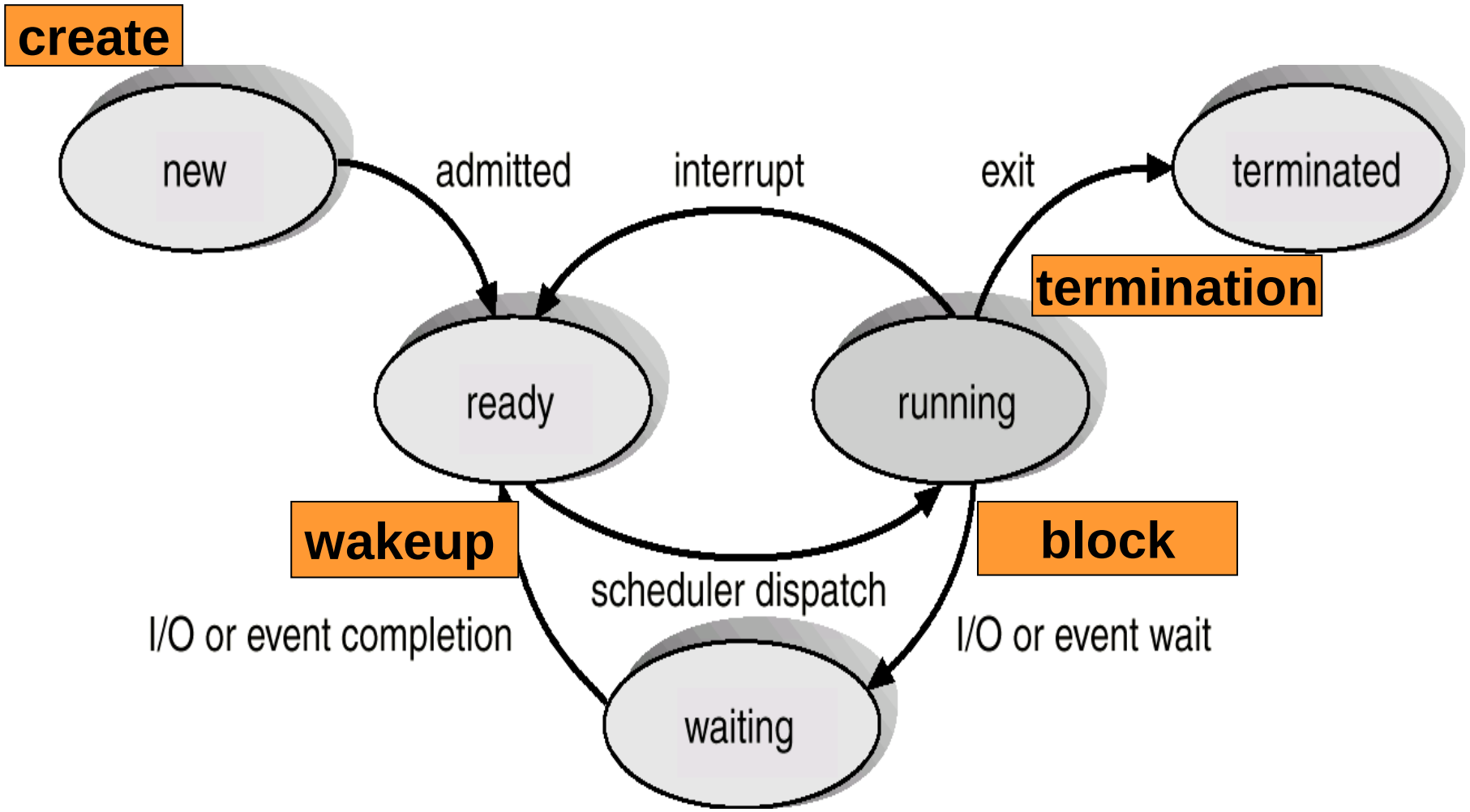
2.2 进 程 控 制

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。它与一般过程的区别在于：它们是“原子操作(Atomic Operation)”。所谓原子操作，是指一个操作中的所有动作要么全做，要么全不做。换言之，它是一个不可分割的基本单位，因此，在执行过程中不允许被中断。原子操作在管态下执行，常驻内存。

- **机器语言编写**
- **常驻内存**
- **不可中断**

Primitives (Operation on Processes)
四个进程控制原语



2.2 进 程 控 制

2.2.1 进程的创建

1．进程图(Process Graph)

进程图是用于描述一个进程的家族关系的有向树，如图2-11所示。图中的结点(圆圈)代表进程。在进程D创建了进程I之后，称D是I的父进程(Parent Process)，I是D的子进程(Progeny Process)。

2.2.1 进程的创建

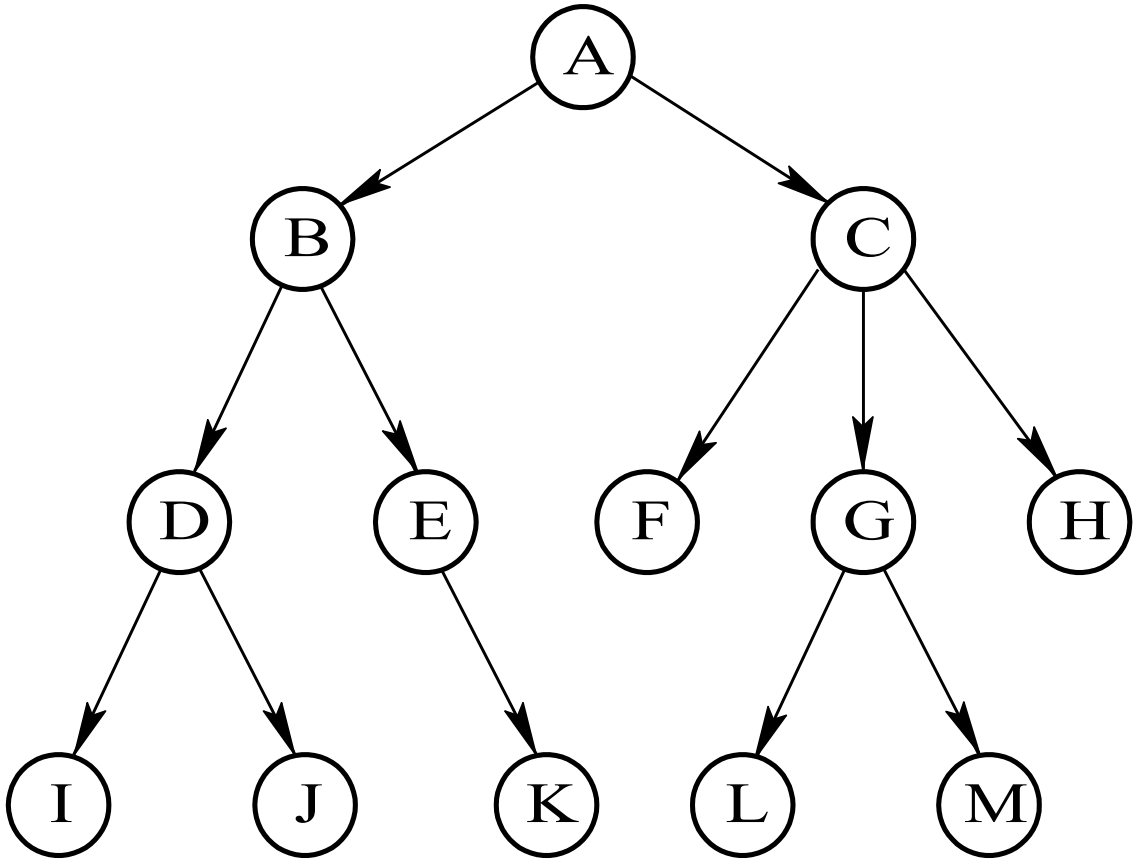
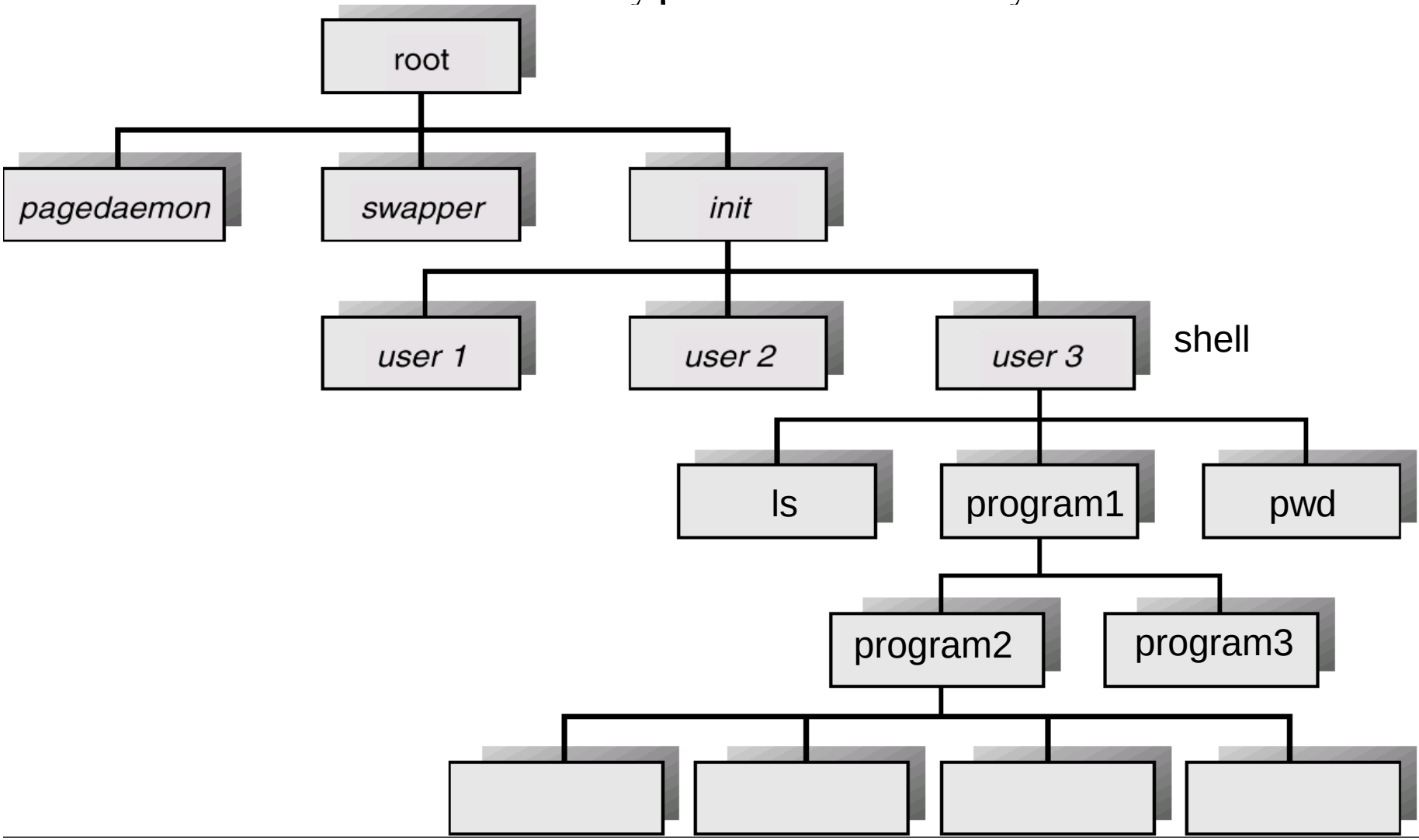


图2-11 进程树

2.2.1 进程的创建

a Tree of Processes on a Typical UNIX System



2.2.1 进程的创建

2．引起创建进程的事件

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

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

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

2.2.1 进程的创建

(3) 提供服务。当运行中的用户程序提出某种请求后，系统将专门创建一个进程来提供用户所需要的服务，例如，用户程序要求进行文件打印，操作系统将为它创建一个打印进程，这样，不仅可使打印进程与该用户进程并发执行，而且还便于计算出为完成打印任务所花费的时间。

(4) 应用请求。上述三种情况，都是由系统内核为它创建一个新进程；第4类事件则是基于应用进程的需求，由它自己创建一个新进程，以便使新进程以并发运行方式完成特定任务。

2.2.1 进程的创建

3．进程的创建原语（Create)

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

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

2.2.1 进程的创建

(3) 初始化进程控制块,包括:

① 初始化标识信息，将系统分配的标识符和父进程标识符填入新PCB中;

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

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

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

2.2.2 进程的终止

1．引起进程终止的事件

1) 正常结束

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。

2.2.2 进程的终止

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)。常见的有下述几种：

(1) 越界错误。这是指程序所访问的存储区已越出该进程的区域。

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

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

2.2.2 进程的终止

(4) 特权指令错。这是指用户进程试图去执行一条只允许OS执行的指令。

(5) 运行超时。这是指进程的执行时间超过了指定的最大值。

(6) 等待超时。这是指进程等待某事件的时间超过了规定的最大值。

(7) 算术运算错。这是指进程试图去执行一个被禁止的运算，例如被0除。

(8) I/O故障。这是指在I/O过程中发生了错误等。

2.2.2 进程的终止

3) 外界干预

外界干预指并非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力，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，有的OS也将它的所有子孙进程终止。

2.2.2 进程的终止

2．进程的终止原语（termination）

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

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

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

2.2.2 进程的终止

(4) 将被终止进程所拥有的全部资源，或者归还给其父进程，或者归还给系统。

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

子进程的处理方式: * 子进程也被终止。

* 重新设置子进程的父进程。

* UNIX :设置子进程的父进程为1#进程

2.2.3 进程的阻塞与唤醒

1. 引起进程阻塞和唤醒的事件

1) 请求系统服务

当正在执行的进程请求操作系统提供服务时，由于某种原因，操作系统并不立即满足该进程的要求时，该进程只能转变为阻塞状态来等待。

2) 启动某种操作

当进程启动某种操作后，如果该进程必须在该操作完成之后才能继续执行，则必须先使该进程阻塞，以等待该操作完成。

2.2.3 进程的阻塞与唤醒

3) 新数据尚未到达

对于相互合作的进程，如果其中一个进程需要先获得另一(合作)进程提供的数据后才能对数据进行处理，则只要其所需数据尚未到达，该进程只有(等待)阻塞。

4) 无新工作可做

系统往往设置一些具有某特定功能的系统进程，每当这种进程完成任务后，便把自己阻塞起来以等待新任务到来。

2.2.3 进程的阻塞与唤醒

2．进程阻塞原语（block)

- 处于运行状态的进程，如果要等待某事件的发生，自己调用阻塞原语，把自己变成阻塞状态。具体操作过程如下：

(1) 停止进程执行，把CPU现场信息保存到该进程的PCB的相应表目中；

(2) 修改PCB的有关表目内容，如把进程状态由运行状态改为阻塞状态；

(3) 根据阻塞原因，把修改后的PCB插入到相应的阻塞队列中；

(4) 转入进程调度程序，从就绪队列中重新调度其他进程运行。

2.2.3 进程的阻塞与唤醒

3．进程唤醒原语（wakeup）

- 当处于阻塞状态的进程等待的事件已经结束，这时就由完成等待事件的进程（如释放资源的进程或完成I/O的进程）调用唤醒原语，将该阻塞进程唤醒，转换成就绪状态。具体操作过程如下：

  (1)把PCB从相应的等待队列中移出。

  (2)修改PCB的有关表目内容，如把进程状态由阻塞状态改为就绪状态。

  (3)将修改后的PCB插入到就绪队列中，等待调度。

2.2.3 进程的阻塞与唤醒

- 运行状态的进程自己调用阻塞原语，把自己变成阻塞状态。
- 执行中的进程通过执行唤醒原语去唤醒一个被阻塞的进程。

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

2.2.4 进程的挂起与激活

1．进程的挂起

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

① 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 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 最后，若被挂起的进程正在执行，则转向调度程序重新调度。

2.2.4 进程的挂起与激活

2．进程的激活过程

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

① 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。

② 假如采用的是抢占调度策略，则每当有新进程进入就绪队列时，应检查是否要进行重新调度。

2.3 进程同步

2.3.1 进程同步的基本概念

生产者-消费者(producer-consumer)问题是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。

2.3.1 进程同步的基本概念

生产者-消费者示例

producer$_i$

consumer$_j$

producer: repeat

produce an item in nextp;

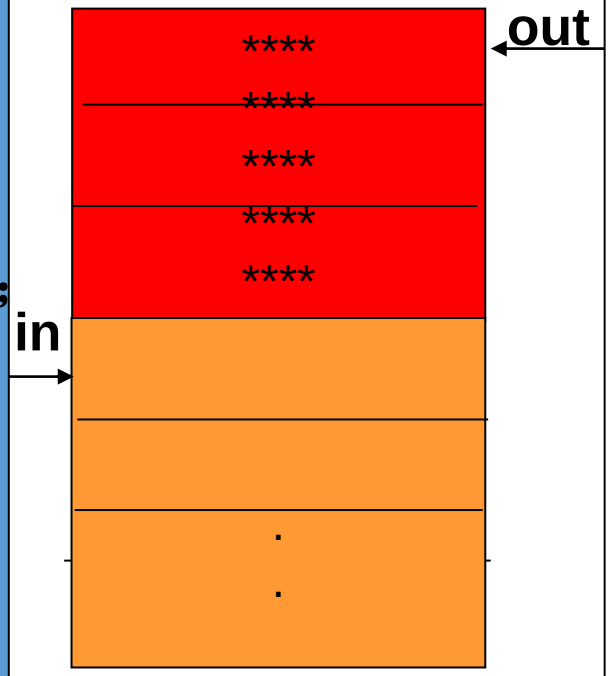
while counter=n do no-op;

buffer［in］:= nextp;

in:=(in+1) mod n;

counter:=counter+1;

until false;



consumer: repeat

while counter=0 do no-op;

nextc:=buffer［out］;

out:=(out+1) mod n;

counter:=counter-1;

consume the item in nextc;

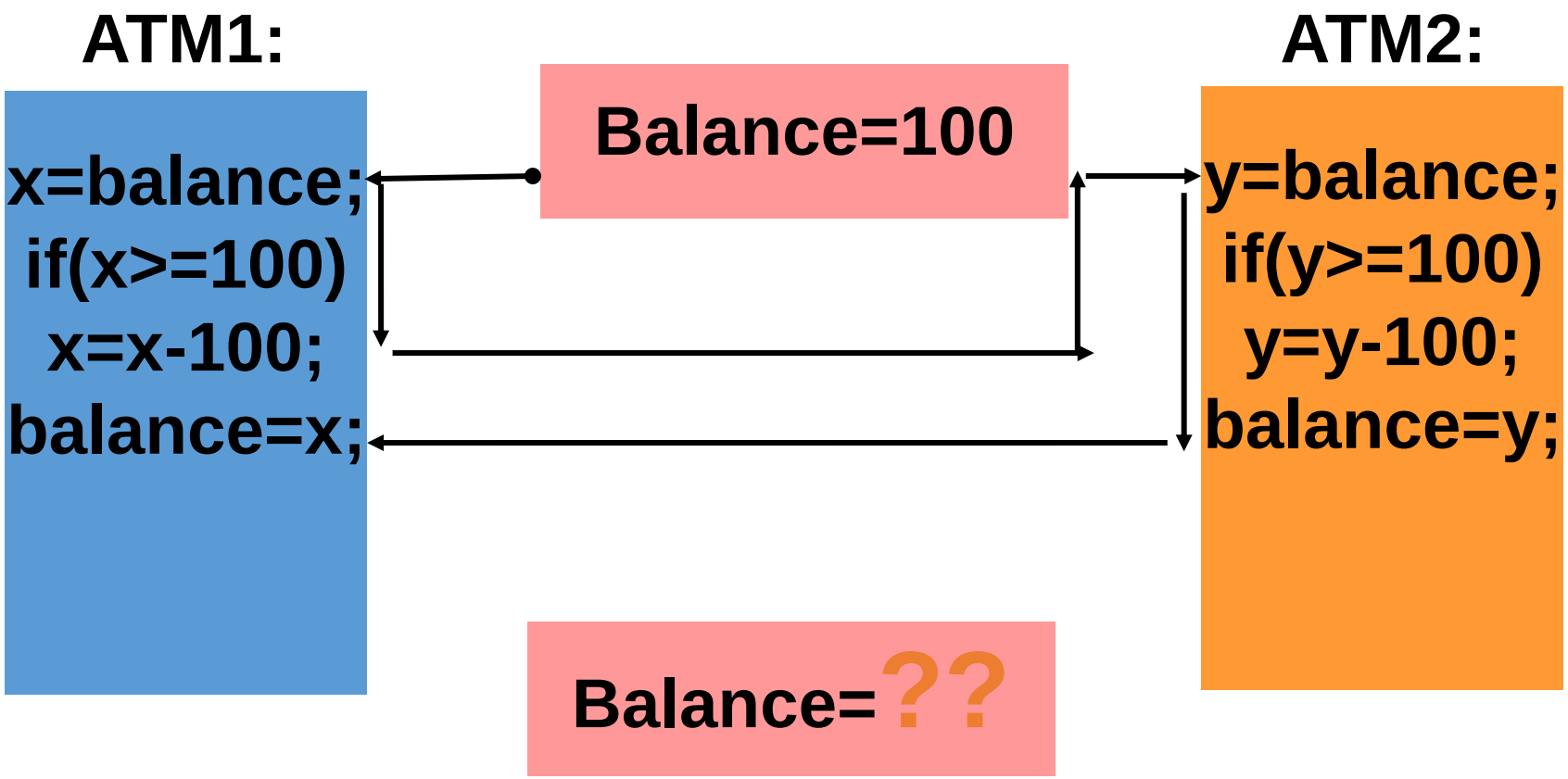
until false;

2.3.1 进程同步的基本概念

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：

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

2.3.1 进程同步的基本概念



ATM example

2.3.1 进程同步的基本概念

同步的例子：司机 –售票员

Driver：

loop1:

driving ;

stop;

go loop1;

Conductor:

Loop2:

close the door;

sale ticket ;

open the door;

go loop2;

如果不同步，会产生什么后果？

2.3.1 进程同步的基本概念

1．并发进程之间的制约关系

在多道程序环境下，当程序并发执行时，由于资源共享和进程合作，使同处于一个系统中的诸进程之间可能存在着以下两种形式的制约关系。

(1) 间接相互制约关系。同处于一个系统中的进程，通常都共享着某种系统资源，如共享CPU、共享I/O设备等。所谓间接相互制约即源于这种资源共享。

(2) 直接相互制约关系。这种制约主要源于进程间的合作。如生产者-消费者问题、司机售票员问题。

2.3.1 进程同步的基本概念

并发进程之间的关系

- 间接制约(互斥)：多个进程不能同时使用同一个资源。

进程 ⟷ 资源 ⟷ 进程

- 直接相互制约(同步)：逻辑上合作的进程，执行速度相互制约

进程 ⟷ 进程

- 操作系统最重要任务之一，就是采用一种高效，可靠的机制实现进程间的同步与互斥

2.3.1 进程同步的基本概念

2. 临界资源

定义：

临界资源：一次只允许一个进程使用的资源，如打印机、公共变量等。

许多硬件资源，如打印机、磁带机等，都属于临界资源(Critical Resource)，诸进程间应采取互斥方式，实现对这种资源的共享。

2.3.1 进程同步的基本概念

3．临界区(CS, Critical Section)

定义：临界区（段）：包含临界资源的程序段称为临界区。

1. 多个进程必须互斥地对临界资源进行访问。人们把在每个进程中访问临界资源的那段代码称为临界区。
2. 显然，若能保证诸进程互斥地进入自己的临界区，便可实现诸进程对临界资源的互斥访问。
3. 为此，每个进程在进入临界区之前，应先对欲访问的临界资源进行检查，看它是否正被访问。
4. 因此，必须在临界区前面增加一段用于进行上述检查的代码，把这段代码称为进入区(entry section)。相应地，在临界区后面也要加上一段称为退出区(exit section)的代码，用于将临界区正被访问的标志恢复为未被访问的标志。

2.3.1 进程同步的基本概念

进程中除上述进入区、临界区及退出区之外的其它部分的代码，在这里都称为剩余区。这样，可把一个访问临界资源的循环进程描述如下：

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

P1

……
Request printer
print..
Print..
Request plotter
print
plot
print
plot
free plotter
print
free printer
….

CS指那一段？

P1，P2若不制约，会出现什么问题？

临界区例

P2

……
Request plotter
plot..
Plot..
Request printer
print
plot
print
plot
free printer
plot
free plotter
….

2.3.1 进程同步的基本概念

4．同步机制应遵循的规则(临界区调度准则)

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

(1) 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。

(2) 忙则等待。当已有进程进入临界区时，表明临界资源正在被访问，因而其它试图进入临界区的进程必须等待，以保证对临界资源的互斥访问。

2.3.1 进程同步的基本概念

(3) 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。

(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，以免进程陷入“忙等”状态。

2.3.2 信号量机制

1．整型信号量

最初由Dijkstra把整型信号量定义为一个用于表示资源数目的整型量S，它与一般整型量不同，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。Wait(S)和signal(S)操作（P和V操作）可描述为：

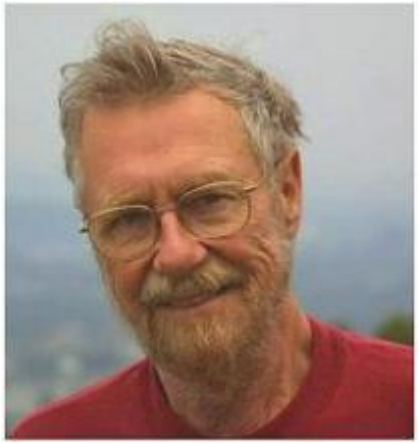
wait(S)： while S<=0 do no-op;
S:=S-1;

signal(S)： S:=S+1;

有忙等现象

wait(S)和signal(S)是两个原子操作，因此，它们在执行时是不可中断的。亦即，当一个进程在修改某信号量时，没有其他进程可同时对该信号量进行修改。

2.3.2 信号量机制



荷兰计算机科学家艾兹格·W·迪科斯彻（Dijkstra）

1 提出“goto有害论”；

2 提出信号量和PV原语;

3 解决了“哲学家聚餐”问题;

4 Dijkstra最短路径算法和银行家算法的创造者;

5 第一个Algol 60编译器的设计者和实现者;

6 THE操作系统的设计者和开发者;

被称为我们这个时代最伟大的计算机科学家的人之一。

1972年获得过素有计算机科学界的诺贝尔奖之称的图灵奖。

2.3.2 信号量机制

2．记录型信号量

① 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。

② 为此，在记录型信号量机制中，除了需要一个用于代表资源数目的整型变量value外，还应增加一个进程链表指针L，用于链接上述的所有等待进程。

2.3.2 信号量机制

type semaphore=record

value: integer；

L: list of process；

end

信号量:

有一个整型变量;

有一个相关的等待队列;

除了初始化外，只能对其做wait、signal 操作（P、V操作）

2.3.2 信号量机制

相应地，wait(S)和signal(S)操作可描述为：

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

2.3.2 信号量机制

1. 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，使系统中可供分配的该类资源数减少一个，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.L中。
2. 对信号量的每次signal操作，表示执行进程释放一个单位资源，使系统中可供分配的该类资源数增加一个，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。
3. 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，用于进程互斥。

2.3.2 信号量机制

(1) 利用记录式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，只须为该资源设置一互斥信号量mutex，并设其初始值为1，然后将各进程访问该资源的临界区CS置于wait(mutex)和signal(mutex)操作之间即可。

1. 这样，每个欲访问该临界资源的进程在进入临界区之前，都要先对mutex执行wait操作，若该资源此刻未被访问，本次wait操作必然成功，进程便可进入自己的临界区，
2. 这时若再有其他进程也欲进入自己的临界区，此时由于对mutex执行wait操作定会失败，因而该进程阻塞，从而保证了该临界资源能被互斥地访问。
3. 当访问临界资源的进程退出临界区后，又应对mutex执行signal操作，以便释放该临界资源。利用信号量实现进程互斥的进程可描述如下：

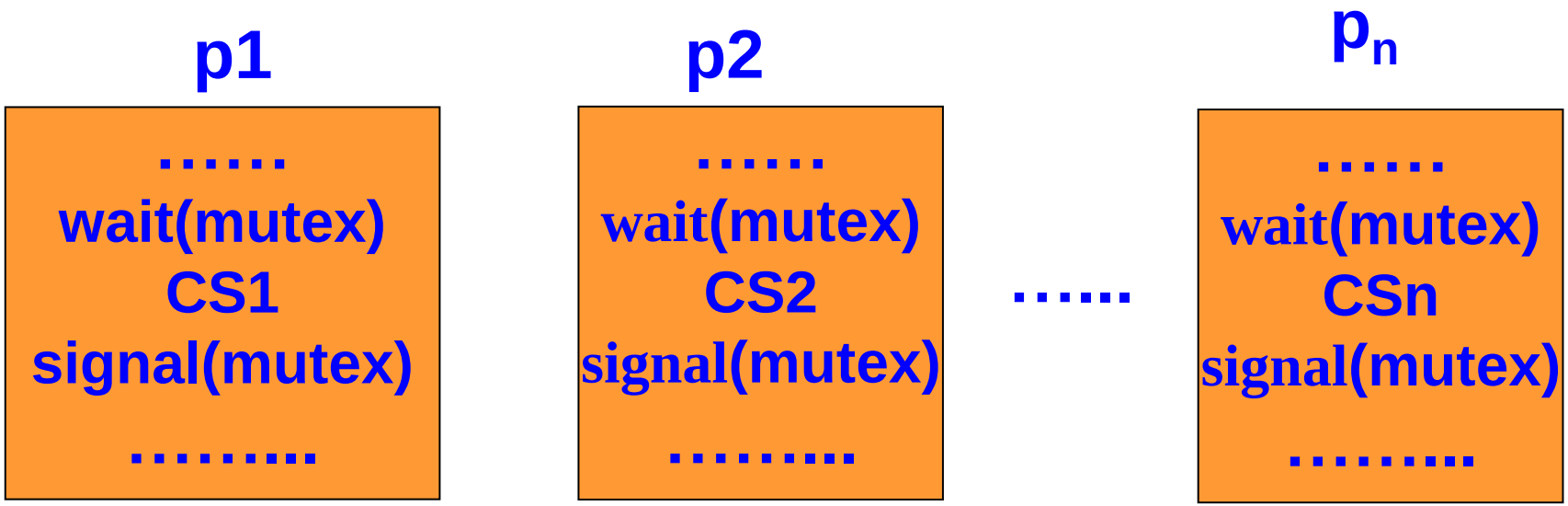
2.3.2 信号量机制

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

2.3.2 信号量机制

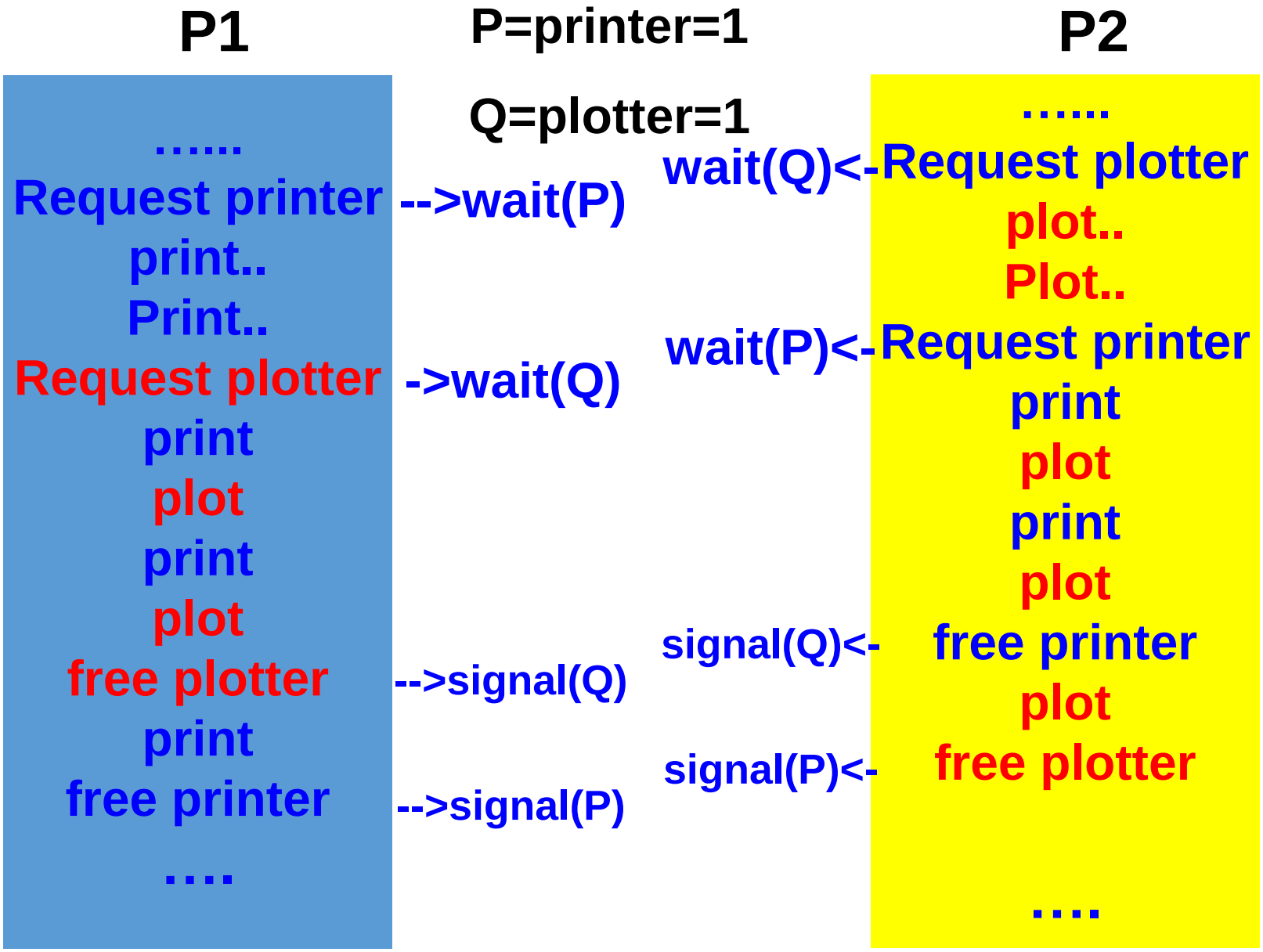
```
process 2: begin
                repeat
                    wait(mutex);
                    critical section
                    signal(mutex);
                    remainder section
                until false;
            end
```

利用信号量实现互斥示例



Mutex 初值=?

- **信号量的物理含义**：当mutex>0，表示可用资源个数；当mutex<0，|mutex|表示等待该信号量的进程个数。
- 须成对使用wait和signal原语：遗漏wait原语则不能保证互斥访问，遗漏signal原语则不能在使用临界资源之后将其释放（给其他等待的进程）；wait、signal原语不能次序错误、重复或遗漏。



信号量实现互斥例：请求资源执行wait操作，释放资源执行signal操作

2.3.2 信号量机制

(2) 用记录式信号量实现同步模型

为进程设置一个同步信号量S，其初值为0；在进程需要同步的地方分别插入wait(S)和signal(S)原语。一个进程使用P原语时，则另一进程往往使用V原语与之对应，具体怎么使用要看实际情况来定。

例如，有两个进程P1、P2，P1的功能是计算x=a+b的值，a和b是常量，在P1的前面代码中能得到；P2的功能是计算y=x+1的值。

```
P1              P2
...             ...
X=a+b;          wait(S)
signal(S)       Y=X+1;
...             ...
```

2.3.2 信号量机制

利用信号量实现同步

Code:

Pi	*Pj*
⋮	⋮
A	*wait(flag)*
signal(flag)	*B*

同步时，同一信号量，一个线程对其做wait操作，则另一个线程对其做signal操作

2.3.2 信号量机制

同步示例：司机 –售票员

Driver：

loop1:

wait(s1);

driving ;

stop;

signal(s2);

go loop1;

Conductor:

Loop2:

close the door;

signal(s1);

sale ticket ;

wait(s2);

open the door;

go loop2;

Semphore s1=0;s2=0;

2.3.2 信号量机制

(3)利用记录式信号量实现前趋关系

还可利用信号量来描述程序或语句之间的前趋关系。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，我们只须使进程P_1和P_2共享一个公用信号量S，并赋予其初值为0，将signal(S)操作放在语句S_1后面；而在S_2语句前面插入wait(S)操作，即

在进程P_1中，执行 S_1；signal(S)；

在进程P_2中，执行 wait(S)；S_2；

2.3.2 信号量机制

由于S被初始化为0，这样，若P_2先执行必定阻塞，只有在进程P_1执行完S_1；signal(S)；操作后使S增为1时，P_2进程方能执行语句S_2成功。同样，我们可以利用信号量，按照语句间的前趋关系(见图2-12)，写出一个更为复杂的可并发执行的程序。

图2-12示出了一个前趋图，其中S_1，S_2，S_3，…，S_6是最简单的程序段(只有一条语句)。为使各程序段能正确执行，应设置若干个初始值为“0”的信号量。如为保证$S_1 \rightarrow S_2$，$S_1 \rightarrow S_3$的前趋关系，应分别设置信号量a 和b，同样，为了保证$S_2 \rightarrow S_4$，$S_2 \rightarrow S_5$，$S_3 \rightarrow S_6$，$S_4 \rightarrow S_6$和$S_5 \rightarrow S_6$，应设置信号量c，d，e，f，g。

2.3.2 信号量机制

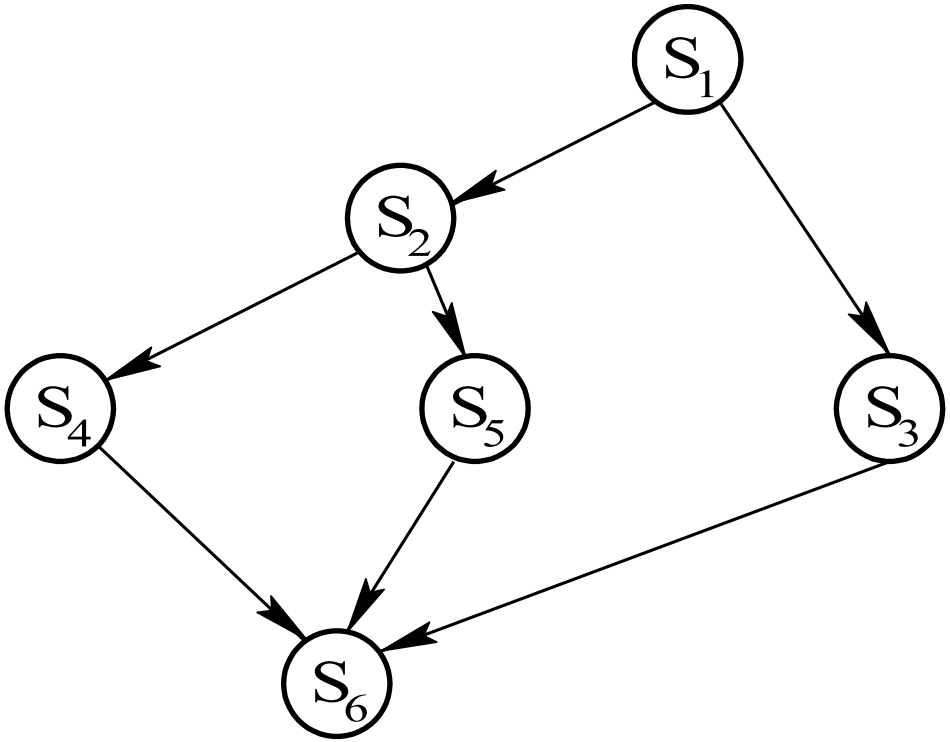


图2-12 前趋图示例

2.3.2 信号量机制

```
Var a,b,c,d,e,f,g: semaphore: =0,0,0,0,0,0,0;
    begin
      parbegin
        begin S1; signal(a); signal(b); end;
        begin wait(a); S2; signal(c); signal(d); end;
        begin wait(b); S3; signal(e); end;
        begin wait(c); S4; signal(f); end;
        begin wait(d); S5; signal(g); end;
        begin wait(e); wait(f); wait(g); S6; end;
      parend
    end
```

2.3.2 信号量机制

3．AND型信号量

上述的进程互斥问题，是针对各进程之间只共享一个临界资源而言的。在有些应用场合，是一个进程需要先获得两个或更多的共享资源后方能执行其任务。为此，可为这两个数据分别设置用于互斥的信号量Dmutex和Emutex，并令它们的初值都是1。

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

2.3.2 信号量机制

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

最后，进程A和B处于僵持状态。在无外力作用下，两者都将无法从僵持状态中解脱出来。我们称此时的进程A和B已进入死锁状态。

2.3.2 信号量机制

AND同步机制的基本思想是：将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程，待进程使用完后再一起释放。只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源也不分配给它。为此，在wait操作中，增加了一个“AND”条件，故称为AND同步，或称为同时wait操作，即Swait(Simultaneous wait)，使用方法如下:

Swait(Dmutex，Emutex)和Ssignal(Dmutex，Emutex)

和定义如下:

2.3.2 信号量机制

Swait(S_1，S_2，…，S_n)

if S_i>=1 and … and S_n>=1 then

for i:=1 to n do

S_i:=S_i-1;

endfor

else

place the process in the waiting queue associated with the first S_i found with S_i<1 and set its program counter to the beginning of the Swait Operation.

endif

Ssignal(S_1，S_2，…，S_n)

for i:=1 to n do

S_i:= S_i +1;

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

endfor;

2.3.2 信号量机制

4．信号量集

在记录型信号量机制中，wait(S)或signal(S)操作仅能对信号量施以加1或减1操作，意味着每次只能获得或释放一个单位的临界资源。而当一次需要N个某类临界资源时，便要进行N次wait(S)操作，显然这是低效的。

此外，在有些情况下，当资源数量低于某一下限值时，便不予以分配。因而，在每次分配之前，都必须测试该资源的数量，看其是否大于其下限值。

基于上述两点，可以对AND信号量机制加以扩充，形成一般化的“信号量集”机制。**Swait操作可描述如下，其中S为信号量，d为需求值，而t为下限值。**

2.3.2 信号量机制

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

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

for i:=1 to n do

S_i:=S_i-d_i;

endfor

else

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

endif

2.3.2 信号量机制

Ssignal(S_1，d_1，…，S_n，d_n)

for i:=1 to n do

S_i:=S_i+d_i;

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

endfor;

2.3.2 信号量机制

一般化“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。

2.3.4 管程机制

信号量机制的引入解决了进程同步和互斥问题，但信号量的大量同步操作分散在各个进程中，不便于管理，还有可能导致系统死锁，因此引入了管程机制。

1．管程的定义

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

① 例如，对一台电传机，可用与分配该资源有关的状态信息(busy或free)和对它执行请求与释放的操作，以及等待该资源的进程队列来描述。

② 又如，一个FIFO队列，可用其队长、队首和队尾以及在该队列上执行的一组操作来描述。

2.3.4 管程机制

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

b) 进程对共享资源的申请、释放和其它操作，都是通过这组过程对共享数据结构的操作来实现的，

c) 这组过程还可以根据资源的情况，或接受或阻塞进程的访问，确保每次仅有一个进程使用共享资源，这样就可以统一管理对共享资源的所有访问，实现进程互斥。

2.3.4 管程机制

管程定义：代表共享资源的数据结构，以及由对该共享数据结构实施操作的一组过程所组成的资源管理程序，共同构成了一个操作系统的资源管理模块。

① 管程由四部分组成：① 管程的名称；② 局部于管程内部的共享数据结构说明；③ 对该数据结构进行操作的一组过程；④ 对局部于管程内部的共享数据设置初始值的语句。

2.3.4 管程机制

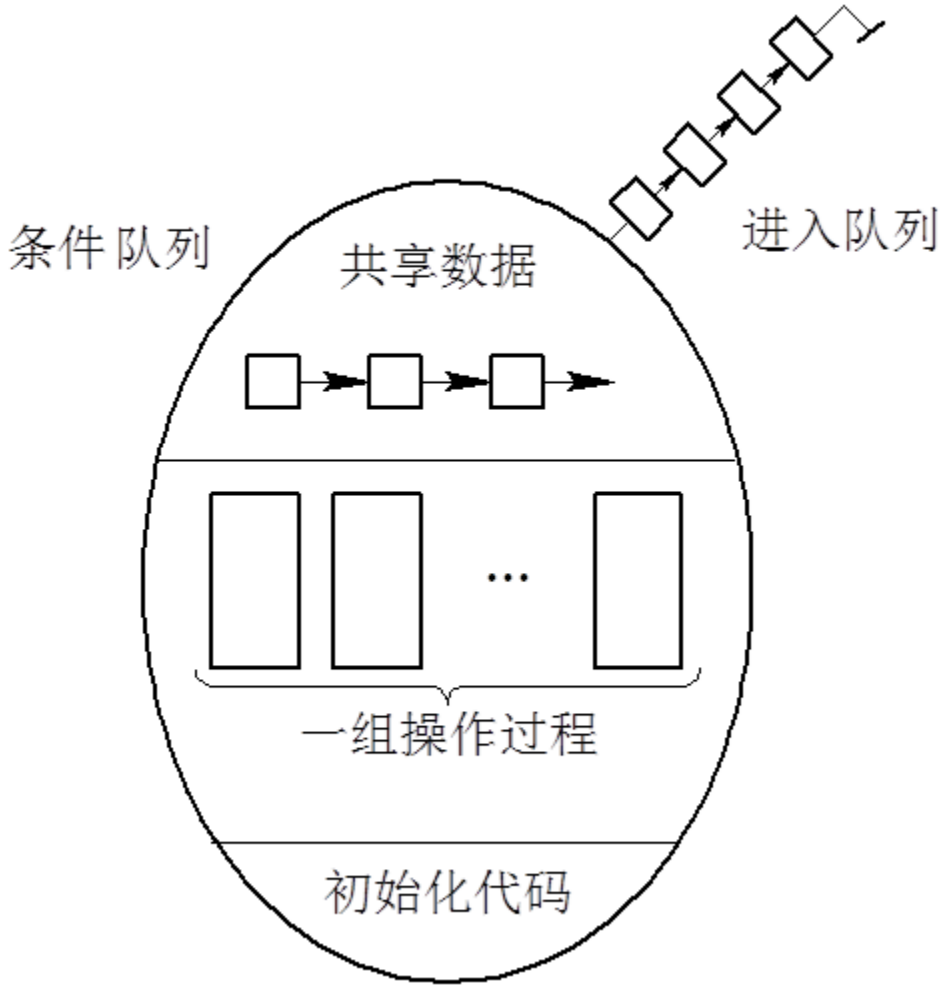


图2-13 管程的示意图

2.3.4 管程机制

管程的语法描述如下：

type monitor_name = MONITOR;

<共享变量说明>;

define <(能被其他模块引用的)过程名列表>;

use <(要调用的本模块外定义的)过程名列表>;

procedure <过程名>(<形式参数表>);

begin

⋮

end;

⋮

2.3.4 管程机制

```
function <函数名>(<形式参数表>)：值类型;
    begin
        ⋮
    end;
        ⋮
begin
  <管程的局部数据初始化语句序列>;
end
```

2.3.4 管程机制

A. 管程内部的数据结构，仅能被管程内部的过程所访问，任何管程外的过程都不能访问它；

B. 管程内部的过程也仅能访问管程内的数据结构。

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

2.3.4 管程机制

管程主要有以下特性：

(1) 模块化。管程是一个基本程序单位，可以单独编译。

(2) 抽象数据类型。管程中不仅有数据，而且有对数据的操作。

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

2.3.4 管程机制

管程和进程区别

(1) 虽然二者都定义了数据结构，但进程定义的是私有数据结构PCB，管程定义的是公共数据结构，如消息队列等；

(2) 二者都存在对各自数据结构上的操作，但进程是由顺序程序执行有关的操作，而管程主要是进行同步操作和初始化操作；

(3) 设置进程的目的在于实现系统的并发性，而管程的设置则是解决共享资源的互斥使用问题；

2.3.4 管程机制

(4) 进程通过调用管程中的过程对共享数据结构实行操作，该过程就如通常的子程序一样被调用，因而管程为被动工作方式，进程则为主动工作方式；

(5) 进程之间能并发执行，而管程则不能与其调用者并发；

(6) 进程具有动态性，由“创建”而诞生，由“撤销”而消亡，而管程则是操作系统中的一个资源管理模块，供进程调用。

2.3.4 管程机制

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，如图2-13 所示。

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。

2.3.4 管程机制

但是仅仅有上述的同步工具是不够的。考虑一种情况：当一个进程调用了管程，在管程中时被阻塞或挂起，直到阻塞或挂起的原因解除，而在此期间，如果该进程不释放管程，则其它进程无法进入管程，被迫长时间地等待。

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

2.3.4 管程机制

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

① x.wait：**正在调用管程的进程因x条件需要被阻塞或挂起，则调用x.wait将自己插入到x条件的等待队列上，并释放管程，直到x条件变化。此时其它进程可以使用该管程。**

2.3.4 管程机制

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，唤醒一个因x条件而阻塞或挂起的进程。如果存在多个这样的进程，则选择其中的一个，如果没有，则继续执行原进程，而不产生任何结果。（这与信号量机制中的signal操作不同，因为后者总是要执行s:=s+1操作，因而总会改变信号量的状态。）

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操作后，进程Q被唤醒，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。

习题

- 购物问题。某超级市场，可容纳100人同时购物。入口处备有篮子，每个购物者可持一个篮子入内购物。出口处结帐，并归还篮子（出、入口仅容纳一个人通过，篮子有无限多个）。（1）请用P、V操作解决购物问题。（2）若顾客最多为N个人，写出算法中**同步信号量**值的可能变化范围（*说明算法中所定义信号量的含义和初值）

习题

- 用信号量PV操作解决下述进程之间的同步与互斥问题：三个进程P1、P2、P3。P1进程通过计算将产生的整数（可能是奇数或偶数）送到容量为200的缓冲区buff1中，P2从buff1中取出偶数数据加1后送回到缓冲区buff1中，P3负责从buff1中取出奇数数据进行打印。要求：各进程不能同时进入buff1中。请写出每个进程的动作序列。
- 两个相对方向的车依次到达一座独木桥的两端。在任一时刻只允许一个方向的车在桥上通行。设计一个交通管制方案，使两个方向的车顺利通过该桥。