第2章 进程管理

进程是操作系统中最重要的概念之一。进程不仅是最基本的并发执行单位，而且也是分配资源的基本单位。

从进程观点出发，对计算机系统进行结构设计，也是软件开发的一种新技术。

主要内容

- 2.1 程序的顺序执行和并发执行
- 2.2 进程的概念
- 2.3 进程控制
- 2.4 进程互斥
- 2.5 进程同步
- 2.6 经典进程问题
- 2.7 管程*
- 2.8 进程通信
- 2.9 Linux进程管理机制
- 2.10 本章小结

本章要点

- 现代操作系统在管理处理器资源时，是以进程为执行层面的基本单位。
- 本章首先从程序的执行过程引入进程概念；然后围绕进程生命周期内状态的转换过程，逐步说明进程的控制、组织与通信，并简单介绍线程的基本知识；接着重点阐述临界区、PV操作和管程机制，以此来保证并发进程之间正确的同步与互斥关系。
- 本章重点掌握以下要点：
  - 理解程序的顺序执行和并发执行，进程状态转换，线程状态转换，临界区概念，两个经典进程问题：生产者-消费者问题和读者写者问题。
  - 掌握进程概念，进程的基本状态，进程的控制，线程概念，进程互斥和同步。
  - 了解进程通信。

2.1 程序的顺序执行和并发执行

- 程序是一组有序指令集合
  - 顺序执行
  - 并发执行

2.1.1 程序的顺序执行

- 程序
  - 是指令的有序集合，是一个在时间上按严格次序前后相继的操作序列，仅当前一操作执行完后，才能执行后继操作，它是一个静态的概念。
  - 一个程序只有经过执行才能得到最终结果。
  - 一般用户在编写程序时不考虑在自己的程序执行过程中还有其它用户程序存在这一事实。

2.1.1 程序的顺序执行

- 顺序程序
  - 一个程序由若干个程序段组成，而这些程序段的执行必须是顺序的 。

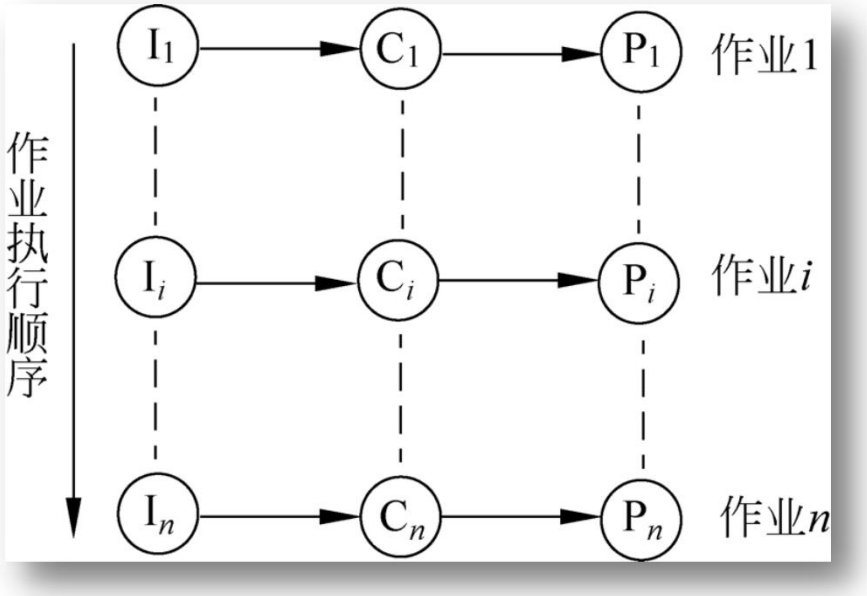
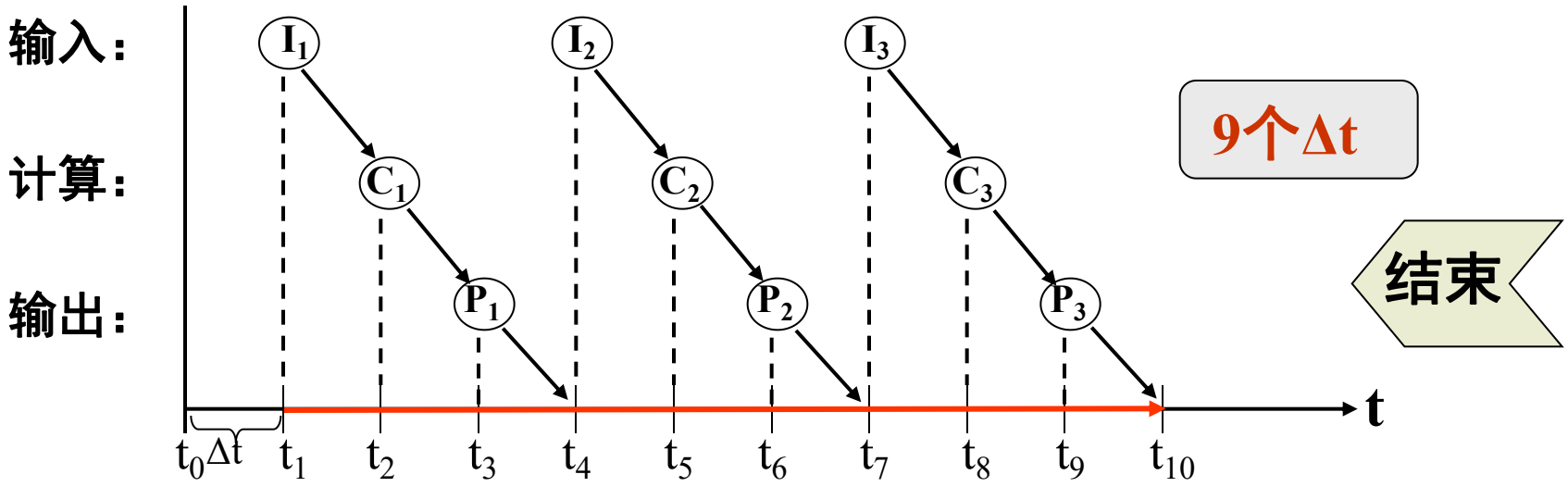


图2- 1 程序的顺序执行

2.1.1 程序的顺序执行

三个程序间顺序执行

程序1：$I_1 \rightarrow C_1 \rightarrow P_1$　程序2：$I_2 \rightarrow C_2 \rightarrow P_2$　程序3：$I_3 \rightarrow C_3 \rightarrow P_3$



2.1.1 程序的顺序执行

- 程序的顺序执行特点
  - 顺序性：严格按程序规定的状态转移过程，每个动作都在上个动作结束后才开始。
  - 封闭性：只有程序本身的动作才能改变程序的运行环境，不受外界因素的影响。
  - 可再现性：只要输入的初始条件相同，则无论何时重复执行该程序都会得到相同的结果，与程序运行的速度无关。

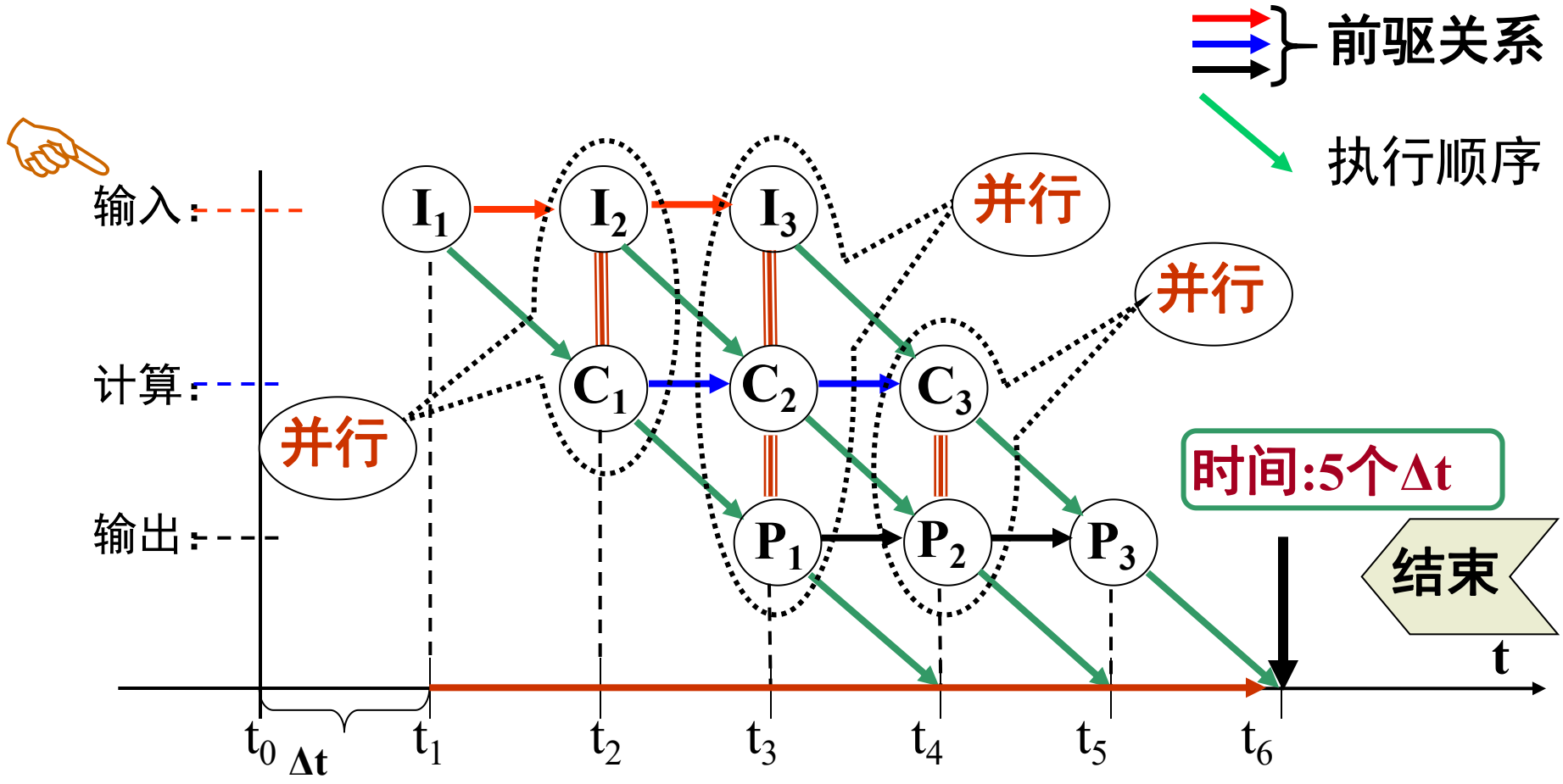
  - 极大的方便程序员检测和校正程序的错误。

2.1.2 程序的并发执行

- 并发执行是为了增强计算机系统的处理能力和提高资源利用率所采取的一种同时操作技术。
  - 第一种是多道程序系统的程序执行环境的变化所引起的多道程序的并发执行。
  - 第二种并发执行是在某道程序的几个程序段中，包含着一部分可以同时执行或顺序颠倒执行的代码。
- 程序的并发执行可总结为：
  - 一组在逻辑上互相独立的程序或程序段在执行过程中其执行时间在客观上互相重叠。

2.1.2 程序的并发执行

下一步



三个程序并发执行示例

2.1.2 程序的并发执行

- 程序并发执行的特征
  - 间断性：程序在并发执行时，由于它们共享资源或为完成同一项任务而相互合作，致使在并发程序间形成了相互制约的关系，导致并发程序具有“执行-暂停-执行”这种间断性的活动规律。
  - 失去封闭性：并发执行的多个程序共享系统中的资源，执行的相对速度是不确定的，控制转换并非由程序本身决定，使程序的运行失去了封闭性。
  - 不可再现性：程序在并发执行时，由于失去了封闭性，也将导致失去其可再现性。

2.2 进程的概念

- 计算机源代码是静态的文本
  - 编译后，得到由指令组成的有序集合的可执行程序。
- 程序的每一次执行的动态过程都不一样

- 在计算机系统中是如何描述和管理程序的每一次执行的呢？

2.2.1 进程的定义

- 使用**程序**这个概念，只能对并发程序进行静止的、孤立的研究，不能深刻地反映它们活动的规律和状态变化。

- 引入新的概念——**进程**
  - 从变化的角度，动态地分析研究并发程序的活动 。
  - 60年代初期，首先在MIT的Multics系统和IBM的TSS/360系统中引用的。

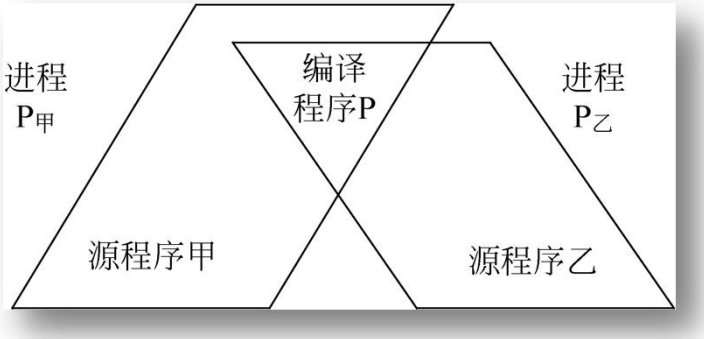


图2- 3 进程的构成

2.2.1 进程的定义

- 人们对进程下过各式各样的定义：
  - ① 进程是可以并发执行的计算部分（S. E. Madnick, J. T. Donovan）
  - ② 进程是一个独立的、可以调度的活动（E. Cohen, D. Jofferson）
  - ③ 进程是一抽象实体，当它执行某个任务时，将要分配和释放各种资源（P. Denning）
  - ④ 行为的规则叫程序，程序在处理器上执行时的活动称为进程（E. W. Dijkstra）
  - ⑤ 一个进程是一系列逐一执行的操作，而操作的确切含义则有赖于我们以何种详尽程度来描述进程（Brinch Hansen）
  - …
  - 以上定义都注意到进程是一个动态的执行过程这一本质概念

- 进程定义
  - 可并发执行的程序在一个数据集上的一次执行过程，它是系统进行资源分配的基本单位 。

2.2.1 进程的定义

- 进程和程序是两个截然不同的概念。

- 进程基本特征

（1）动态性。最基本的特性：“它由创建而产生，由调度而执行，因得不到资源而暂停执行，以及由撤销而消亡”。

（2）并发性。重要特征：能在一段时间内同时运行。

（3）独立性。能独立运行的基本单位，独立获得资源和独立调度的基本单位；进程与程序并非是一一对应的，一个程序运行在不同的数据集上就构成不同的进程。

（4）异步性。进程按各自独立的、不可预知的速度向前推进。

（5）结构特征。进程实体是由程序段、数据段及进程控制块三部分组成，统称为“进程映像”。

2.2.2 进程状态与转换

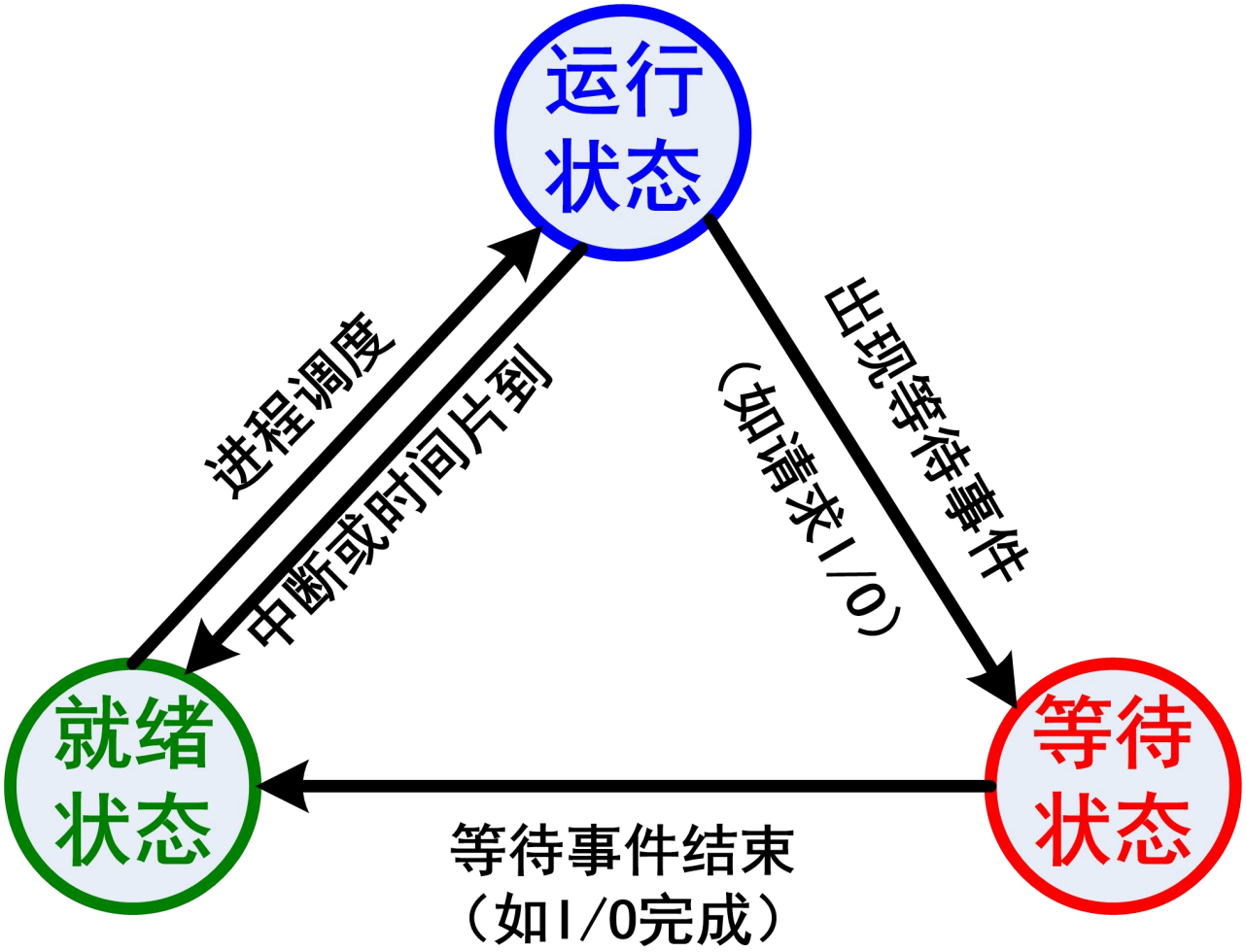
- 在进程的生命周期内，由于并发执行及相互间的制约，进程的状态会不断变化。

- 一般而言，进程至少具备3种基本状态：运行状态、就绪状态和等待状态。

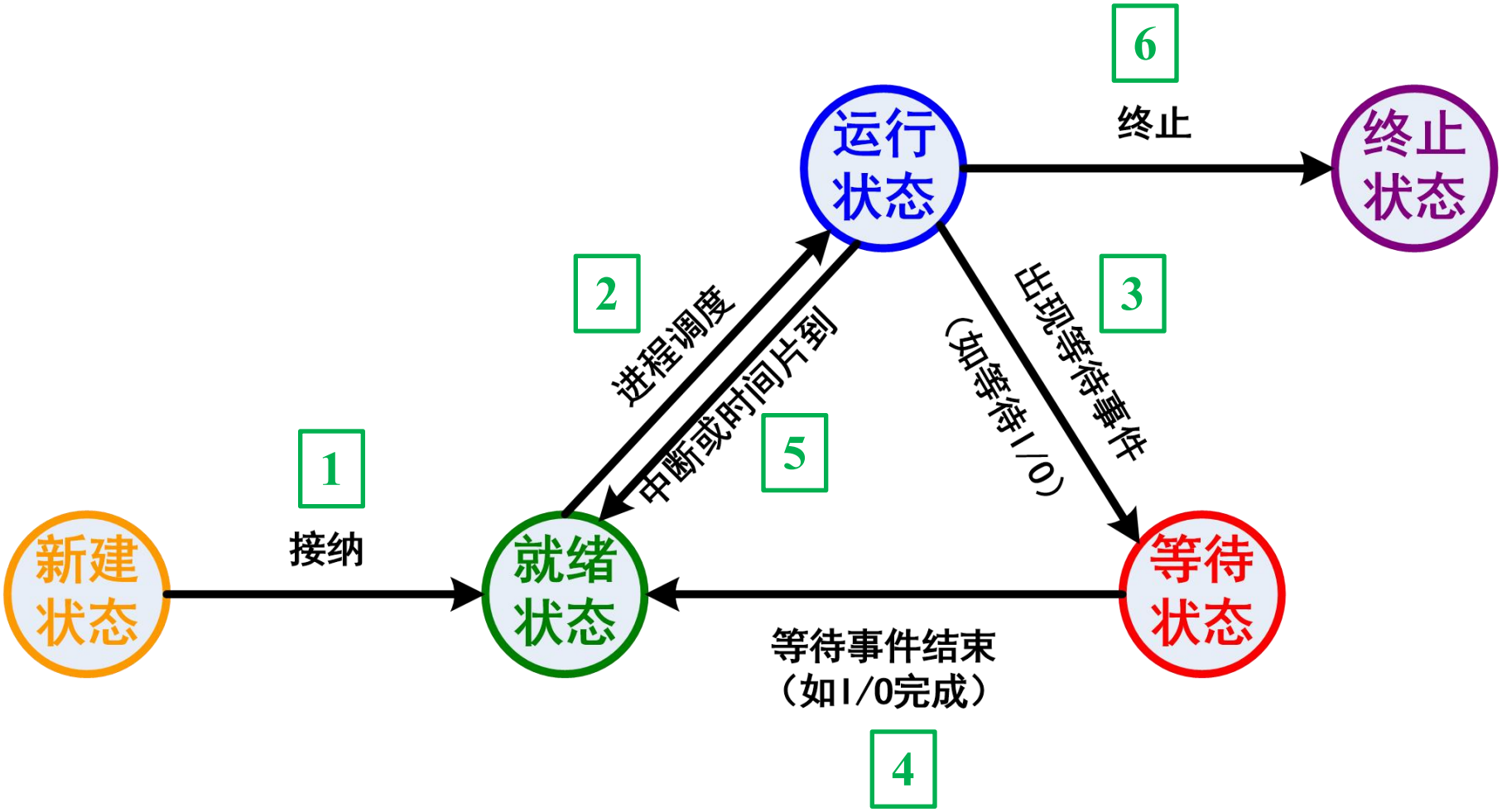
2.2.2 进程的状态

- 进程的状态
  - 就绪状态：就绪状态是指进程已经具备运行条件，但因其他进程正占用CPU，使其不能运行而只能处于等待CPU的状态。处于此状态的进程数目可以有多个。
  - 运行状态：运行状态是指当前进程已经分配到CPU，正在处理机上执行时的状态。
  - 等待状态：又称为阻塞状态或封锁状态，一个进程正在等待某一事件（如等待某资源成为可用，等待输入输出完成或等待与其他进程的通信等）而暂时不能运行的状态。

2.2.2 进程的状态



2.2.2 进程状态的转换


新建状态
就绪状态
运行状态
终止状态
等待状态
1
接纳
2
进程调度
3
出现等待事件
(如等待I/O)
4
等待事件结束
(如I/O完成)
5
中断或时间片到
6
终止

2.2.3 进程控制块

- 每一个进程都有一个也只有一个进程控制块（Process Control Block，简称 PCB）
  - 进程控制块是操作系统用于记录和刻画进程状态及有关信息的数据结构，也是操作系统控制和管理进程的主要依据，它包括了进程执行时的情况，以及进程让出处理器后所处的状态、断点等信息。
- 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。
  - “操作系统是根据PCB来对并发执行的进程进行控制和管理的”

2.2.3 进程控制块

进程控制块包含四类信息：

（1）**标识信息**：唯一的标识符

（2）**说明信息**：当前状态、等待原因、“进程程序存放位置”、“进程数据存放位置” 等

（3）**现场信息**：处理器中通用寄存器内容、控制寄存器内容以及程序状态字寄存器内容等

（4）**管理信息**：对进程进行管理和调度的信息

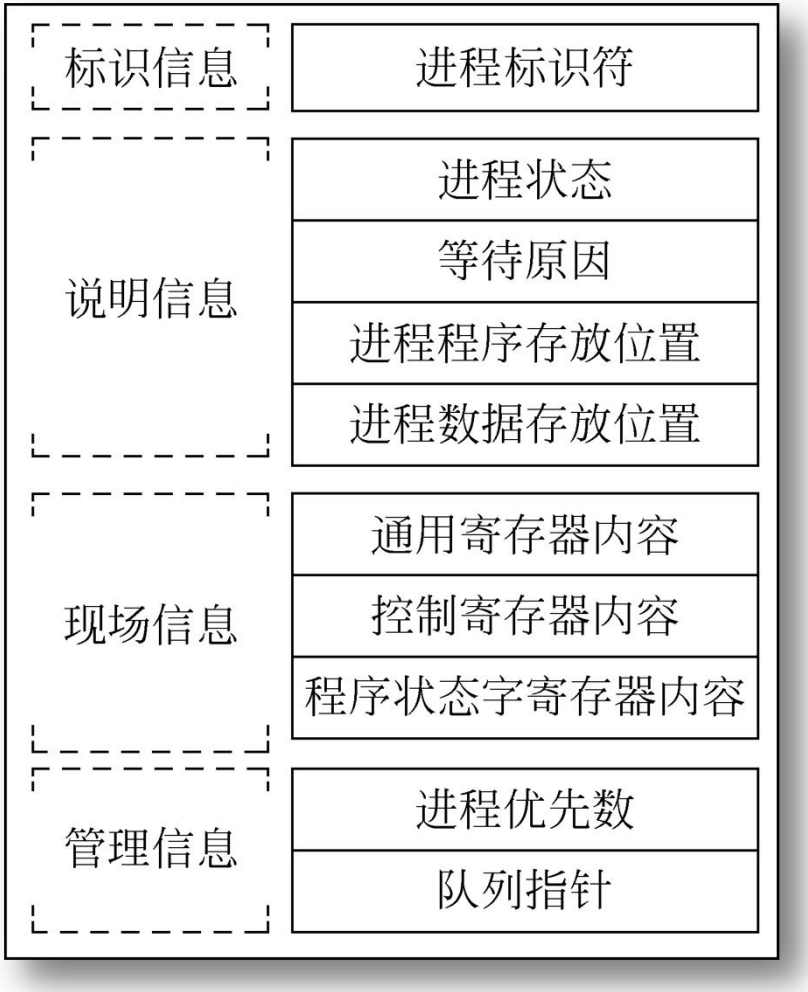


图2- 6 进程控制块

2.2.4 进程队列

- 为了便于管理，通常把处于相同状态的进程链接在一起，称为“进程队列”
  - “就绪队列”
  - “等待队列”

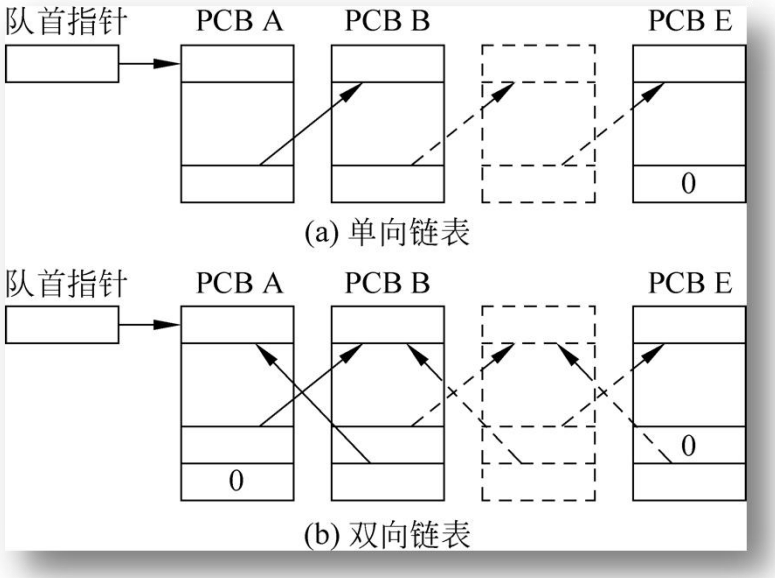


图2- 7 进程队列

- 进程队列可以用进程控制块的
  - 链接来形成
    - 单向链接
    - 双向链接
  - 索引方式来形成

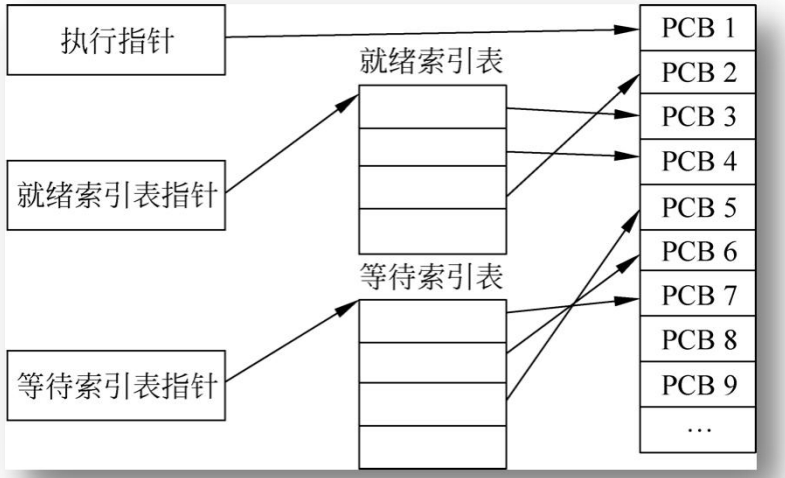


图2- 8 按索引方式组织PCB

2.2.5 线程

- 自从60年代进程概念提出后，一直作为现代操作系统设计的一个核心，操作系统运行的基本单位就是进程。

- 80年代中期，人们又提出了比进程更小的能独立运行的基本单位——线程，以进一步提高程序并发执行的程度，降低并发执行的时空开销。

1. 线程的引入

- 进程是实现系统并发运行的一种实体
  - 创建进程时需要申请必要的系统资源
  - 运行过程中，根据需要还将申请更多资源
- 进程既是资源申请及拥有的实体，同时也是调度的实体
  - 系统因为创建进程、调度进程、管理进程等将付出很大的额外开销
- 保持系统的并发性，降低系统为此付出的额外开销
  - 将传统意义的进程进行分离：将资源申请与调度执行分开
  - 进程作为资源的申请与拥有单位，线程作为调度的基本单位
    - 线程是进程中的一个实体
    - 线程自己基本上不拥有系统资源
    - 一个线程可以创建另一个线程，同一进程中的多个线程可以并发执行

2. 线程的定义

- 线程（Thread）
  - 是进程中的一个实体，是可独立参与调度的基本单位
  - 一个进程可以有一个或多个线程，它们共享所属进程所拥有的资源
- 线程属性：
  - ① 多个线程可以并发执行
  - ② 一个线程可以创建另一个线程
  - ③ 线程具有动态性
  - ④ 每个线程同样有自己的数据结构即线程控制块（Thread Controlling Block, TCB）
  - ⑤ 在同一进程内，各线程共享同一地址空间（即所属进程的存储空间）
  - ⑥ 一个进程中的线程在另一进程中是不可见的
  - ⑦ 同一进程内的线程间的通信主要是基于全局变量进行的

3. 线程的状态

- 与进程类似，线程也有生命周期

- 线程的状态有运行、就绪和等待，线程的状态转换也与进程类似
  - 对于多线程进程的进程状态，由于进程不是调度单位，不必将其划分成过细的状态
    - Windows操作系统中仅把进程分成可运行态和不可运行态

4. 线程的特征

- 有时称线程为轻量级进程

- 线程的特征

① 拥有资源方面
  - 线程只拥有一点在运行中必要的资源，如程序计数器、寄存器和栈

② 调度方面
  - 而线程是独立参与调度的基本单位
  - 同一个进程内的线程切换不会引起进程切换

③ 并发性方面
  - 同一个进程的多个线程间亦可并发执行

④ 系统开销方面
  - 系统开销将显著降低
  - 进程通信更加容易（共享所属进程的存储空间）

5. 线程的分类

- 多线程的实现分为三类：
  - 内核级线程(Kernel Level Thread, KLT)
  - 用户级线程（User Level Thread, ULT）
  - 混合式线程方式

5. 线程的分类

- 1）内核级线程
  - 内核级线程是指线程的管理工作由内核完成，由内核所提供的线程API来使用线程
    - 内核为其创建进程和一个基线程
    - 线程在执行过程中通过内核的创建线程原语来创建其他线程
    - 应用程序的所有线程均在一个进程中获得支持
  - 内核级线程的优点
    - 在多处理器上，内核能够同时调度同一进程中的多个线程并行执行
    - 若进程中的一个线程被阻塞，内核能够调度同一进程的其他线程占有处理器运行，也可以运行其他进程中的线程
    - 切换速度比较快，内核自身也可用多线程技术实现，从而提高系统的执行效率
  - 内核级线程的缺点
    - 线程在用户态运行，而线程的调度和管理在内核实现
    - 线程需要用户态　核心态　用户态的模式切换，系统开销较大（线程模式切换）

5. 线程的分类

- 2）用户级线程
  - 用户级线程是指线程的管理由应用程序完成，在用户空间中实现，内核无须感知线程的存在
    - 由用户空间中的线程库来完成
  - 用户级线程优点
    - 线程切换无须使用内核特权方式
    - 允许进程按照应用的特定需要选择调度算法，且线程库的线程调度算法与操作系统的低级调度算法无关
    - 能够运行在任何操作系统上，内核无须做任何改变
  - 用户级线程的缺点
    - 一个用户级线程的阻塞将引起整个进程的阻塞
    - 进程执行不可能得益于多线程的并发执行

5. 线程的分类

- 3) 混合式线程
  - 操作系统既支持用户级线程，又支持内核级线程
    - Solaris
  - 线程的实现分为两个层次
    - 用户层
      - 用户层线程在用户线程库中实现 (用户级线程)
    - 内核层
      - 内核层线程在操作系统内核中实现 (内核级线程)
  - 宏观上和微观上都具有很好的并行性
    - 在混合式线程中，一个应用程序中的多个用户级线程能分配和对应于一个或多个内核级线程，内核级线程可同时在多处理器上并行执行，且在阻塞一个用户级线程时，内核可以调度另一个线程执行

6. 线程与进程结构

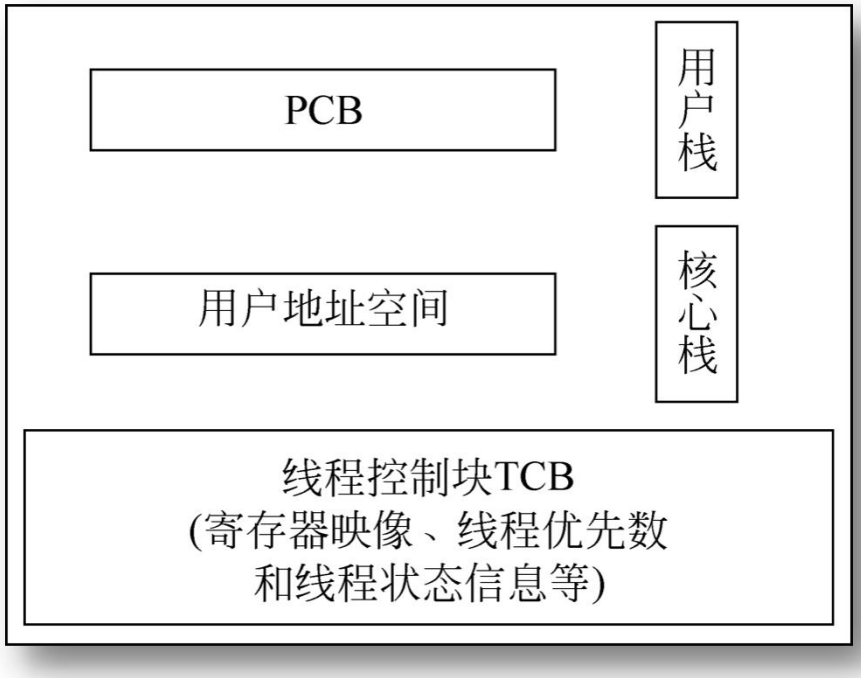


图2- 9 单线程进程结构

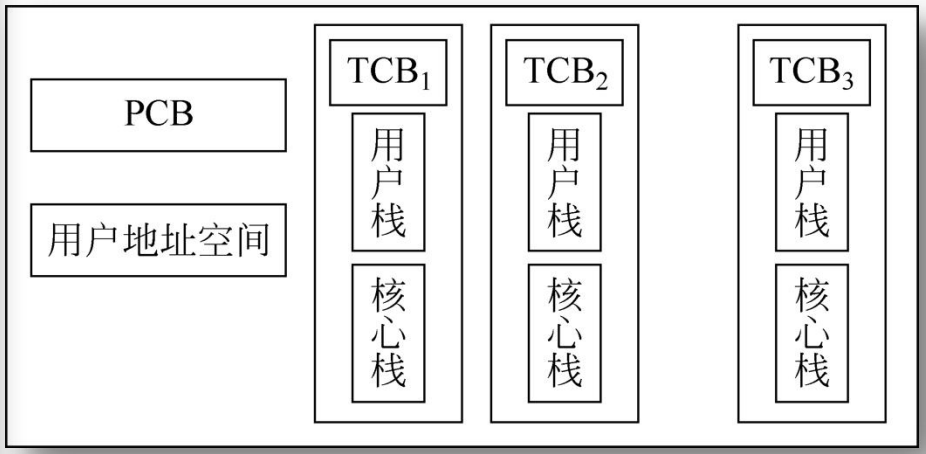


图2- 10 多线程进程结构

2.3 进程控制

- 为了防止操作系统及关键数据受到用户程序有意或无意的破坏，通常将处理器的运行状态分成核心态和用户态两种。

（1）核心态，又称为管态

- 它具有较高的特权，能执行一切指令，访问所有寄存器和存储区

（2）用户态，又称为目标态、目态

- 具有较低特权的运行状态，只能执行规定的指令，访问指定的寄存器和存储区

2.3 进程控制

- 处理器管理的一个重要任务是进程控制
  - 进程控制是系统使用一些具有特定功能的程序段来创建、撤销进程以及完成进程各状态之间的转换，从而达到多进程、高效率的并发执行和协调，实现资源共享的目的。
  - 把核心态下执行的某些具有特定功能的程序段称为原语。
  - 特点是在执行期间不允许中断，是一个不可分割的基本单位。
  - 原语的执行是顺序的而不是并发的，系统对进程的控制使用原语来实现。
    - 进程创建、进程撤销、进程阻塞、进程唤醒等原语

2.3.1 进程创建

- 在系统生成时，要创建一些必需的、承担系统资源分配和管理工作的系统进程
  - 创建者称为父进程，被创建者称为子进程，创建父进程的进程称为祖父进程。
  - 构成了一个进程家族

- 创建原语
  - 无论是系统或是用户创建进程都必须调用创建原语来实现
  - 创建原语的主要功能是创建一个指定标识符的进程，主要任务是形成该进程的PCB

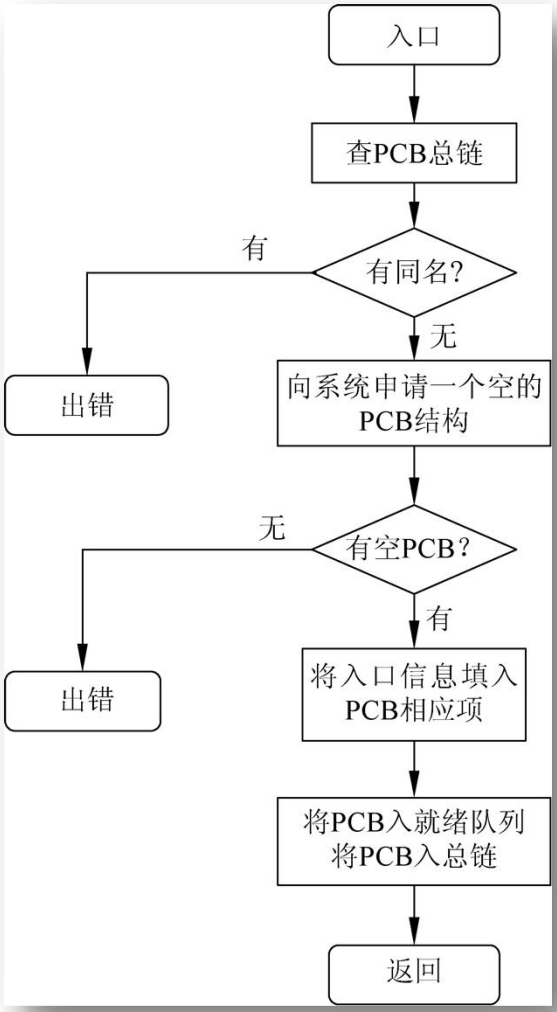


图2- 11 创建原语实现过程

2.3.2 进程撤销

- 操作系统通常提供各种撤销（或称终止）进程的方法
  - 正常终止
  - 由于一个错误而非正常终止
  - 依据祖先进程的要求被终止

- 进程被撤销时
  - 从系统队列中移出
  - 释放并归还所有系统资源
  - 审查并撤销子孙进程

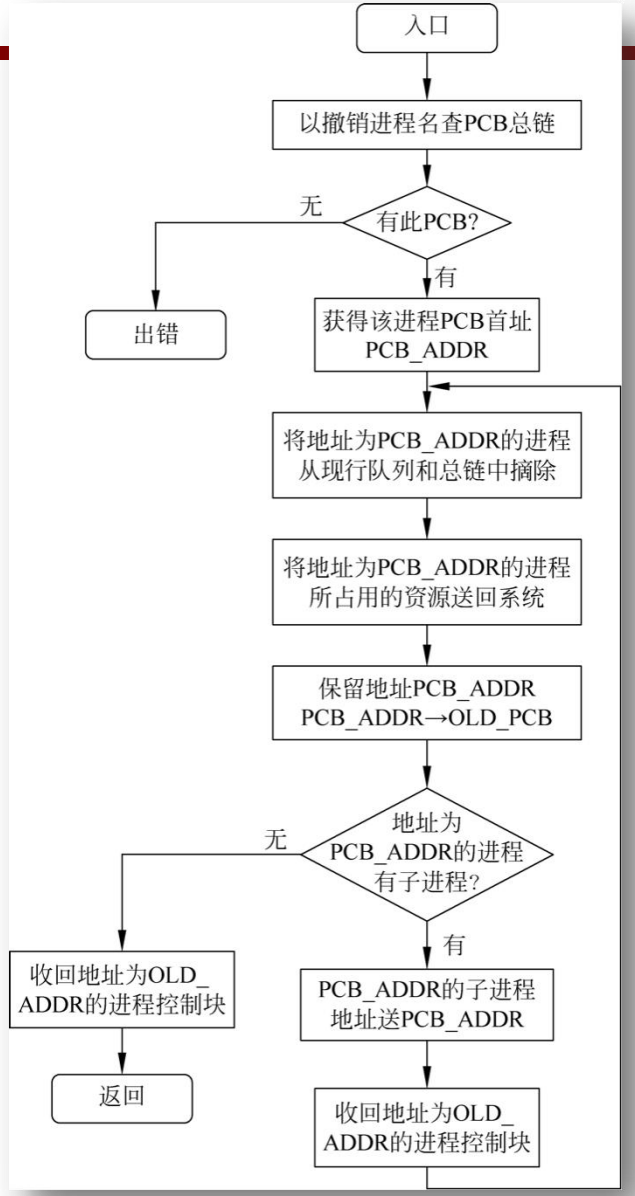


图2- 12 撤销原语的执行过程

2.3.3 进程阻塞与唤醒

- 进程各种状态之间的转换
  - 例子：
  - “执行”转换为“等待”
  - “等待”转换为“就绪”

- 方式：
  - 通过进程之间的同步或通信机构来实现
  - 直接使用“阻塞原语”和“唤醒原语”来实现

2.3.3 进程阻塞与唤醒

- 进程阻塞
  - 当一个进程出现等待事件时，该进程调用**阻塞原语**将自己阻塞

- 功能
  - 保护进程现场数据
  - 置进程的状态为“等待”
  - 插入到相应的等待队列
  - 转进程调度程序

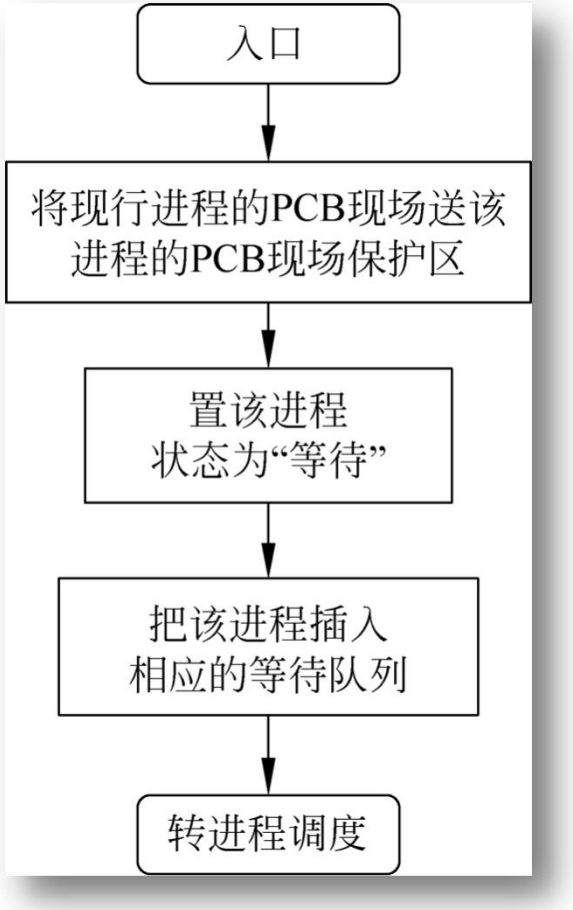


图2- 13 阻塞原语执行过程

2.3.3 进程阻塞与唤醒

- **进程唤醒**
  - 进程所期待的事件出现时，由“发现者”进程用**唤醒原语**叫醒它
- **功能**
  - 唤醒处于某一等待队列中的进程，入口信息为唤醒进程名

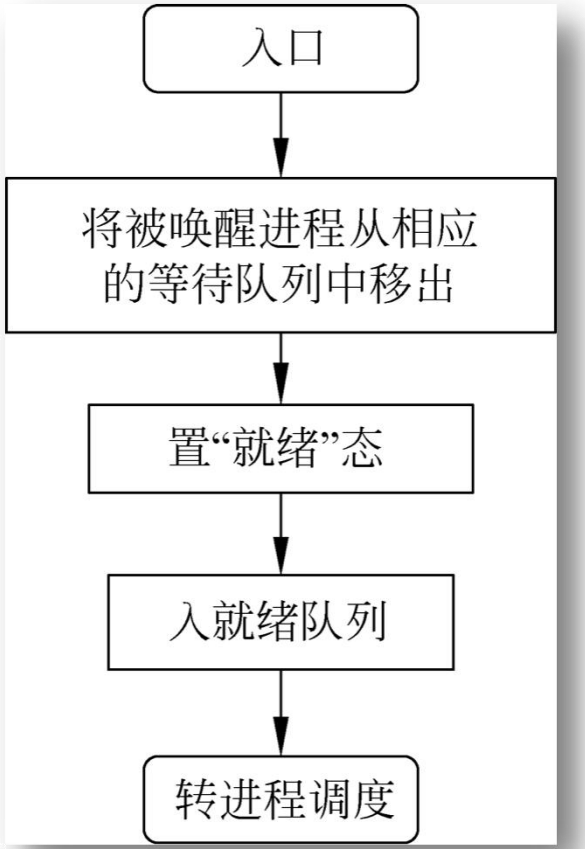


图2- 14 唤醒原语执行过程

2.3.4 进程挂起与激活

- 挂起（suspend）状态
  - 刻画主存中的进程由于输入输出的速度慢于处理机的运算速度等待时，因被操作系统对换至外存或称辅存（如磁盘交换区等）的进程状态

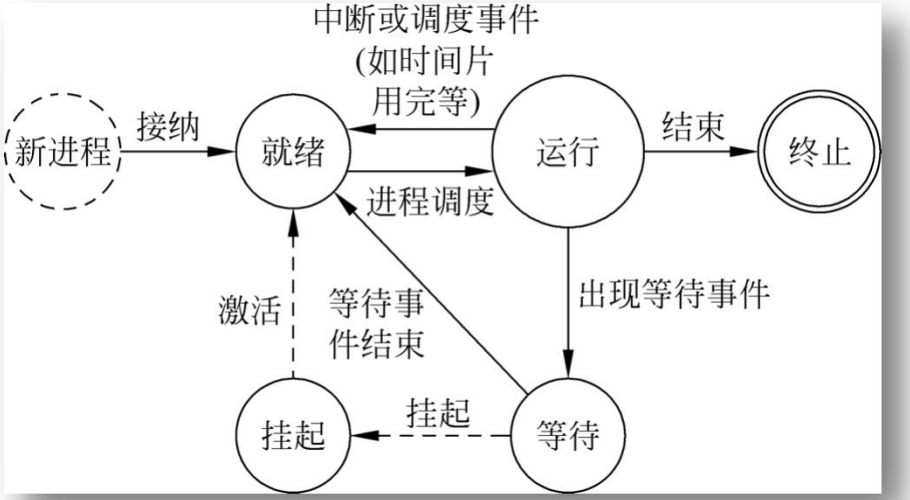


图2- 15 具有挂起状态的进程转换图

- 激活（active）原语
  - 主存中空出了进程执行所需的资源，操作系统调用激活原语激活进程，将进程从外存调入主存中使用空出的资源，修改这个进程的状态为就绪态

2.4 进程互斥

- 并发进程相互之间可能是无关的，也可能是有交往的
  - 一个进程的执行不影响其他进程的执行，且与其他进程的进展情况无关，则这些并发进程相互之间是无关的。
  - 一个进程的执行可能影响其他进程的执行结果，则说这些并发进程相互之间是有交往的，是有关的。

- 有哪些原因使得一个进程的执行可能影响其他进程的执行结果？
  - 有交往的并发进程一定共享某些资源
  - 进程执行与时间有关的错误

2.4.1 与时间有关的错误

- 进程执行的速度不能由自己来控制，若干并发进程同时使用共享资源，一个进程一次使用未结束，另一进程已开始使用，**形成交替使用共享资源的现象**。
  - 这种情况不加控制的话，就可能出现与时间有关的错误。

2.4.1 与时间有关的错误

- 【例1】 火车票售票问题
  - 假设一个火车订票系统有两个终端，分别运行进程T1和T2。该系统的公共数据区中的一些单元Aj（j=1, 2, …）分别存放某月某日某次航班的余票数，而x1和x2表示进程T1和T2执行时所用的工作单元。

可能出现如下所示的运行情况（设Aj =m）：

T1: x1=Aj;	即	x1=m(m>0)
T2: x2=Aj;	即	x2=m
T2: x2=x2-1; Aj=x2; [输出一张票]；	即	Aj=m-1
T1: x1=x1-1; Aj=x1; [输出一张票]；	即	Aj=m-1

把同一张票卖给了两个旅客的情况

```
void      Ti(int  i)  {  //i=1,2,…
int  xi;
[按旅客订票要求找到Aj];
xi=Aj;
if (xi>=1){
  xi=xi-1;
  Aj = xi;
 [输出一张票];
  }
else
  [输出信息“票已售完”];
}
```

2.4.1 与时间有关的错误

- 【例2】 主存管理问题：假定两个并发进程borrow和return分别负责申请和归还主存资源，算法中x为现有空闲主存总量，B表示申请或归还的主存量。
  - 由于borrow和return共享了表示主存物理资源的临界变量x，对并发执行不加限制会导致错误。

```
int x=1000;
cobegin
void borrow(int B){
if (B>x)
        [申请进程进入等待队列等主存资源];
else{
        x=x-B;
        [修改主存分配表，申请进程获得主存资源];}
}
void return(int B)
{ x=x+B;
[修改主存分配表];
[释放等待主存资源的进程];
}
coend;
```

2.4.1 与时间有关的错误

- 【例3】 自动计算问题：某交通路口设置了一个自动计数系统，该系统由“观察者”和“报告者”进程组成。观察者进程能识别卡车，并对通过的卡车计数；报告者进程定时（每小时）将观察者的计数值打印输出，每次打印后把计数值清0。
  - 并发执行可能出现的两种情况：

（1）报告者进程执行时无卡车通过

（2）报告者进程执行时有卡车通过

```
int count=0;
cobegin
void observer(){
while(1){
        [oberver a lorry];
        count=count+1;
}
}
void reporter(){
print(“%d”,count);
count=0;
}
coend;
```

2.4.2 临界区

- 出现与时间有关的错误
  - 根本原因是对共享资源（变量）的使用不加限制

```
进程 T1 的临界区为：
x1=Aj;
if (x1>=1) {
   x1=x1-1;
   Aj= x1;
   [输出一张票]；
   } else
      [输出信息“票已售完”];
```

```
进程 T2 的临界区为：
x2=Aj;
if (x2>=1) {
   x2=x2-1;
   Aj=x2；
   [输出一张票]；
   } else
      [输出信息“票已售完”];
```

两个临界区都要使用共享变量Aj，故属于相关临界区

- “临界区”
  - 指并发进程中与共享变量有关的程序段
- “临界资源”
  - 指共享变量所代表的资源
- “相关临界区”
  - 指多个并发进程中涉及相同共享变量的那些程序段

- 不出现与时间有关的错误的**要求**：

① 一次最多让一个进程在临界区执行，当有进程在临界区执行时，其他想进入临界区执行的进程必须等待。

② 任何一个进入临界区执行的进程必须在有限的时间内退出临界区，即任何一个进程都不应该无限地逗留在自己的临界区中。

③ 不能强迫一个进程无限地等待进入它的临界区，即有进程退出临界区时应让一个等待进入临界区的进程进入它的临界区。

2.4.3 进程的互斥

- 进程的互斥
  - 指当有若干进程都要使用某一共享资源时，任何时刻最多只允许一个进程去使用，其他要使用该资源的进程必须等待，直到占用资源者释放该资源
  - 共享资源的互斥使用就是限定并发进程互斥地进入相关临界区
  - 对相关临界区管理
    - 可实现进程的互斥主要方法有：
    - 标志方式、上锁开锁方式、PV操作方式和管程方式等

1. 信号量与PV操作

- 信号量的概念和PV操作是荷兰科学家E. W. Dijkstra提出来的
  - 在操作系统中，信号量S是一整数
    - S大于等于零时，代表可供并发进程使用的资源实体数
    - S小于零时则用|S|表示正在等待使用资源实体的进程数
  - 建立一个信号量必须说明此信号量所代表的意义并且赋初值
  - 信号量仅能通过PV操作来访问
- 信号量按用途分为两种

（1）公用信号量

- 相关的进程均可在此信号量上执行P操作和V操作，初值常常为1
- 用于实现进程互斥

（2）私有信号量

- 仅允许拥有此信号量的进程执行P操作，而其他相关进程可在其上施行V操作，初值常常为0或正整数
- 多用于实现进程同步

1. 信号量与PV操作

- S表示信号量
  - P操作和V操作是两个在信号量上进行操作的过程
  - 把这两个过程记作P(S)和V(S)，定义如下：

```
void P(Semaphore S) {
    S=S-1;
    if (S<0)    W(S); }
```

```
void V(Semaphore S){
    S=S+1;
    if (S<=0)    R(S); }
```

  - W(S)表示将调用P(S)过程的进程置成“等待信号量S”的状态，且将其排入等待队列。
  - R(S)表示释放一个“等待信号量S”的进程，使该进程从等待队列退出并加入到就绪队列中。

1. 信号量与PV操作

- 用PV操作来管理共享资源，首先要确保PV操作自身执行的正确性。

- 由于P(S)和V(S)都是在同一个信号量S上操作，为了使得它们在执行时不发生交叉访问信号量S而可能出现的错误，约定P(S)和V(S)必须是两个不可被中断的过程。
  - 把不可被中断的过程称为“原语”

P操作原语

- **P操作的主要动作是：**
  - ① S减1；
  - ② 若S减1后仍大于或等于零，则进程继续执行；
  - ③ 若S减1后小于零，则该进程被阻塞后放入等待该信号量的等待队列中，然后执行权转交系统进程调度。

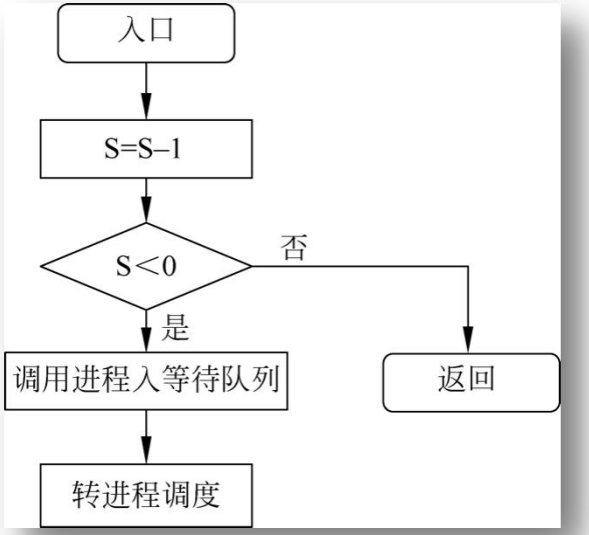


图2- 16 P操作功能

V操作原语

- V操作主要动作是：
  - ① S加1；
  - ② 若相加结果大于零，进程继续执行；
  - ③ 若相加结果小于或等于零，则从该信号的等待队列中释放一个等待进程，然后再返回原进程继续执行或执行权转交系统进程调度。

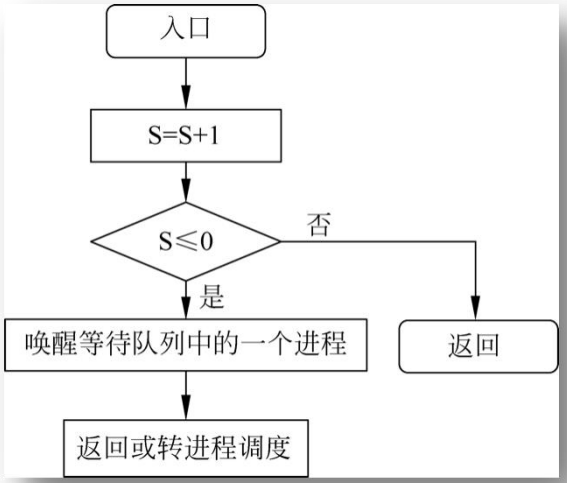


图2- 17 V操作功能

- S的初值可定义为0、1或其它整数，在系统初始化时确定

- 推论：
  - 推论1：若信号量S为正值，则该值等于在阻塞进程之前对信号量S可施行的P操作数，亦即等于S所代表的实际还可以使用的物理资源数。
  - 推论2：若信号量S为负值，则其绝对值等于登记排列在该信号量S队列之中等待的进程个数，亦即恰好等于对信号量S实施P操作而被阻塞并进入信号量S等待队列的进程数。
  - 推论3：P操作意味着请求一个资源，V操作意味着释放一个资源。在一定条件下，P操作代表阻塞进程操作，而V操作代表唤醒被阻塞进程的操作。

2. 用PV操作实现进程互斥

- **用PV操作实现并发进程的互斥步骤：**

  （1）设立一个互斥信号量S表示临界区，其取值范围1，0，-1，…

  - **S=1表示无并发进程进入S临界区**
  - **S=0表示已有一个并发进程进入了S临界区**
  - **S等于负数表示已有一个并发进程进入了S临界区，且有|S|个进程等待进入S临界区（注，S的初值为1）**

  （2）用PV操作表示对S临界区的申请和释放

  - **在进入临界区之前，通过P操作进行申请**
  - **在退出临界区之后，通过V操作释放**

P 进程	Q 进程
……	……
P(S);	P(S);
临界区；	临界区；
V(S);	V(S);
……	……

- 【例4】 用PV操作管理火车票售票问题

```
Semaphore S =1;
cobegin
void Ti () // i=1, 2  即 T1 和 T2 进程
{ int xi;
   [按旅客订票要求找到 Aj];
   p(S);
   xi=Aj;
   if (xi>= 1)
        { xi=xi-1;
          Aj=xi;
          V(S);
          [输出一张票];
          } else
             { V(S);
               [输出信息“票已售完”]; }
      }
coend;
```

◆ 【例5】 用PV操作管理主存问题

```
int x=1000;
Semaphore S=1;

cobegin
void borrow (int B) {
p(S);
if (B>x)
   [申请进程进入等待队列等主存资源];
x=x-B;
[修改主存分配表，申请进程获得主存
资源];
V(S);
}

void return (int B){
p(S);
x=x+B;
[修改主存分配表];
V(S);
[释放等主存资源的进程];
}
coend;
```

◆ 【例6】 用PV操作管理自动计数问题

```
int count=0;
Semaphore S=1;

cobegin
void observer(){
while(1) {
   [observer a lorry];
   p(S);
   count=count+1;
   V(S);}
}
```

```
void reporter(){
p(S);
printf("%d", count);
count=0;
V(S);
}
coend;
```

- 【例7】 用PV操作解决五个哲学家吃通心面问题
  - 有五个哲学家围坐在一圆桌旁，桌子中央有一盘通心面，每人面前有一只空盘子，每两人之间放一把叉子。每个哲学家思考、饥饿，然后，欲吃通心面。为了吃面，每个哲学家必须获得两把叉子，且每人只能直接从自己左边或右边取得叉子（见图2- 18所示）。

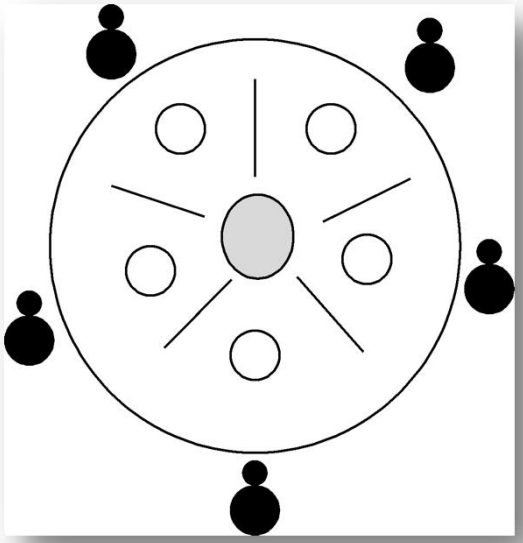


图2- 18 五个哲学家吃通心面问题

- 每一把叉子都是必须互斥使用的
  - 每把叉子设置一个互斥信号量S_i(i=0, 1, 2, 3, 4)，初值均为1
- 一个哲学家吃通心面之前必须获得自己左边和右边的两把叉子
  - 执行两个P操作
- 一个哲学家吃完通心面后必须放下叉子
  - 执行两个V操作

```
Semaphore S0, S1, S2, S3, S4;
S0=1;
S1=1; S2=1; S3=1; S4=1;

cobegin
void Ki() // (i=0, 1, 2, 3, 4)
{
while(1) {
      [思考];
      P(Si);
      P((Si+1) % 5);
      [吃通心面];
      V(Si);
      V((Si+1) % 5);
  }
}
coend;
```

2.5 进程同步

- 同步关系
  - 进程A需要等到进程B的执行结果作为输入才能执行，它们之间协作完成一件事情
  - （如同人们走路时左脚和右脚之间的协作一样）

2.5.1 进程的同步

- 互斥问题
  - 利用信号量解决
  - 互斥主要是解决并发进程对临界区的使用问题
  - 比较简单的问题

- 异步环境下的进程同步问题
  - 相互合作的一组并发进程，其中每一个进程都以各自独立的、不可预知的速度向前推进，但它们又需要密切合作，以实现一个共同的任务
  - 彼此“知道”相互的存在和作用

- 例如：进程A启动输入设备不断地读记录，每读出一个记录就交给进程B去加工，直至所有记录都处理结束
  - 如果两个进程不相互制约的话就会造成错误
  - 当进程A的执行速度超过进程B的执行速度时，可能造成记录的丢失
  - 当进程B的执行速度超过进程A的执行速度时，可能造成重复地取同一个记录加工

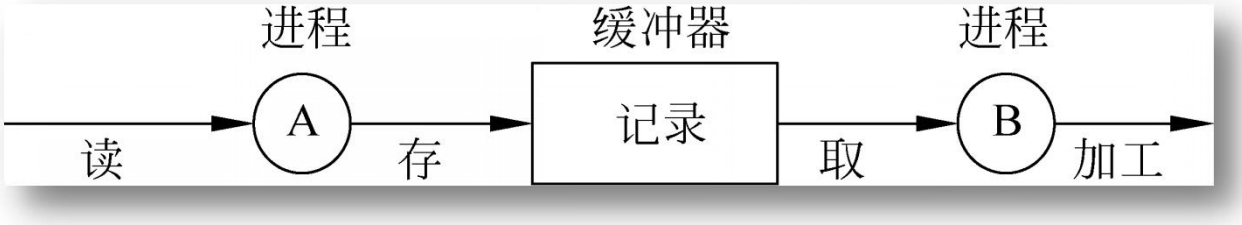


图2- 19 进程协作

- 用进程互斥的办法不能克服上述两种错误
  - 进程A和进程B虽然共享缓冲器，但它们都是在无进程使用缓冲器时才向缓冲器存记录或从缓冲器取记录的
  - 引起错误的根本原因是它们之间的相对执行速度

- 采用互通消息的办法来控制执行速度，使相互协作的进程正确工作：
  ① 进程A把一个记录存入缓冲区后，应向进程B发送“缓冲器中有等待处理的记录”的消息；
  ② 进程B从缓冲器中取出记录后，应向进程A发送“缓冲器中的记录已取走”的消息；
  ③ 进程A只有在得到进程B发送来的“缓冲器中的记录已取走”的消息后，才能把下一个记录再存入缓冲器。否则进程A等待，直到消息到达；
  ④ 进程B只有在得到进程A发送来的“缓冲器中有等待处理的记录”的消息后，才能从缓冲器中取出记录并加工。否则进程B等待，直到消息到达。

2.5.2 用PV操作实现进程的同步

- 同步机制
  - 同步机制是指把其它进程需要的消息发送出去，也能测试自己需要的消息是否到达的一种机制
  - 典型的同步机制
    - PV操作
    - 管程

- PV操作
  - 是实现进程互斥的有效工具
  - 是一个简单而方便的同步工具

- **PV操作实现同步**
  - 用信号量S表示某个期望的消息，消息尚未产生 （值为“0” ）或已经存在 （值为非“0” ）
  - 步骤：
    （1）调用P操作测试消息是否到达
    （2）调用V操作发送消息
  - 注意：
    - 一定要根据具体的问题来定义信号量和调用P操作或V操作
    - 一个信号量与一个消息联系在一起，当有多个消息时必须定义多个信号量
    - 对不同的信号量调用对应信号量的P操作或V操作

2.5.3 时间同步问题

- 进程同步
  - 空间上的同步问题
    - 一组有关的并发进程在执行时访问共享变量
  - 时间上的同步问题
    - 一组有关的并发进程在执行时间上有严格的先后顺序

- 例如，有七个进程，它们的执行顺序如图2- 20所示
- 定义六个信号量
  - S2：表示进程P2能否执行
  - S3：表示进程P3能否执行
  - S4：表示进程P4能否执行
  - S5：表示进程P5能否执行
  - S6：表示进程P6能否执行
  - S7：表示进程P7能否执行
  - 进程P1不需定义信号量，可随时执行
  - 这些信号量的初值都为0，表示不可执行，而当大于等于1时，表示可执行

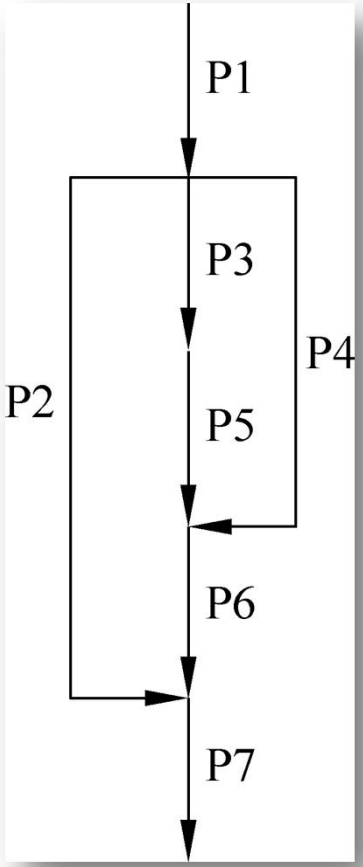
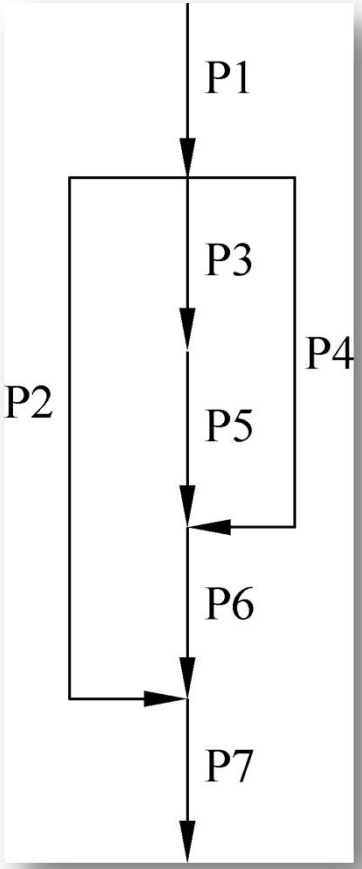


图2- 20 七个进程的执行顺序



同步工作描述如下：

```
Semaphore S2, S3, S4, S5,
S6, S7;
S2=0; S3=0; S4=0; S5=0;
S6=0; S7=0;
cobegin
    void P1(){
    ……
    V(S2);
    V(S3);
    V(S4);
    }

    void P2(){
    P(S2);
    ……
    V(S7);
    }

    void P3(){
    P(S3);
    ……
    V(S5);
    }

    void P4(){
    P(S4);
    ……
    V(S6);
    }

    void P5(){
    P(S5);
    ……
    V(S6);
    }

    void P6(){
    P(S6);
    P(S6);
    ……
    V(S7);
    }

    void P7(){
    P(S7);
    P(S7);
    ……
    }

coend;
```

- 【习题】用信号量与P、V操作实现司机与售票员之间的同步问题。设公共汽车上有一名司机和一名售票员，为了安全起见，显然要求：
  ① 关车门后方能启动车辆；
  ② 到站停车后方能开车门。亦即“启动车辆”这一活动应当在“关车门”这一活动之后，“开车门”这一活动应当在“到站停车”这一活动之后。

司机的活动（P1）：

```
do {

    启动车辆;
    正常行车;
    到站停车;

}while(1);
```

售票员的活动（P2）：

```
do {
    关车门;

    售票;

    开车门;
}while(1);
```

现设置信号量start，初值为0，用于查看是否可以启动车辆；
设置信号量open，初值为0，用于查看是否可以打开车门。
通过PV操作来实现同步。

司机的活动（P1）：

```
do {
    P(start);
    启动车辆;
    正常行车;
    到站停车;
    V(open);
}while(1);
```

售票员的活动（P2）：

```
do {
    关车门;
    V(start);
    售票;
    P(open);
    开车门;
}while(1);
```

2.6 经典进程问题

- 生产者—消费者问题
- 读者—写者问题
- 理发师问题
- 独木桥问题

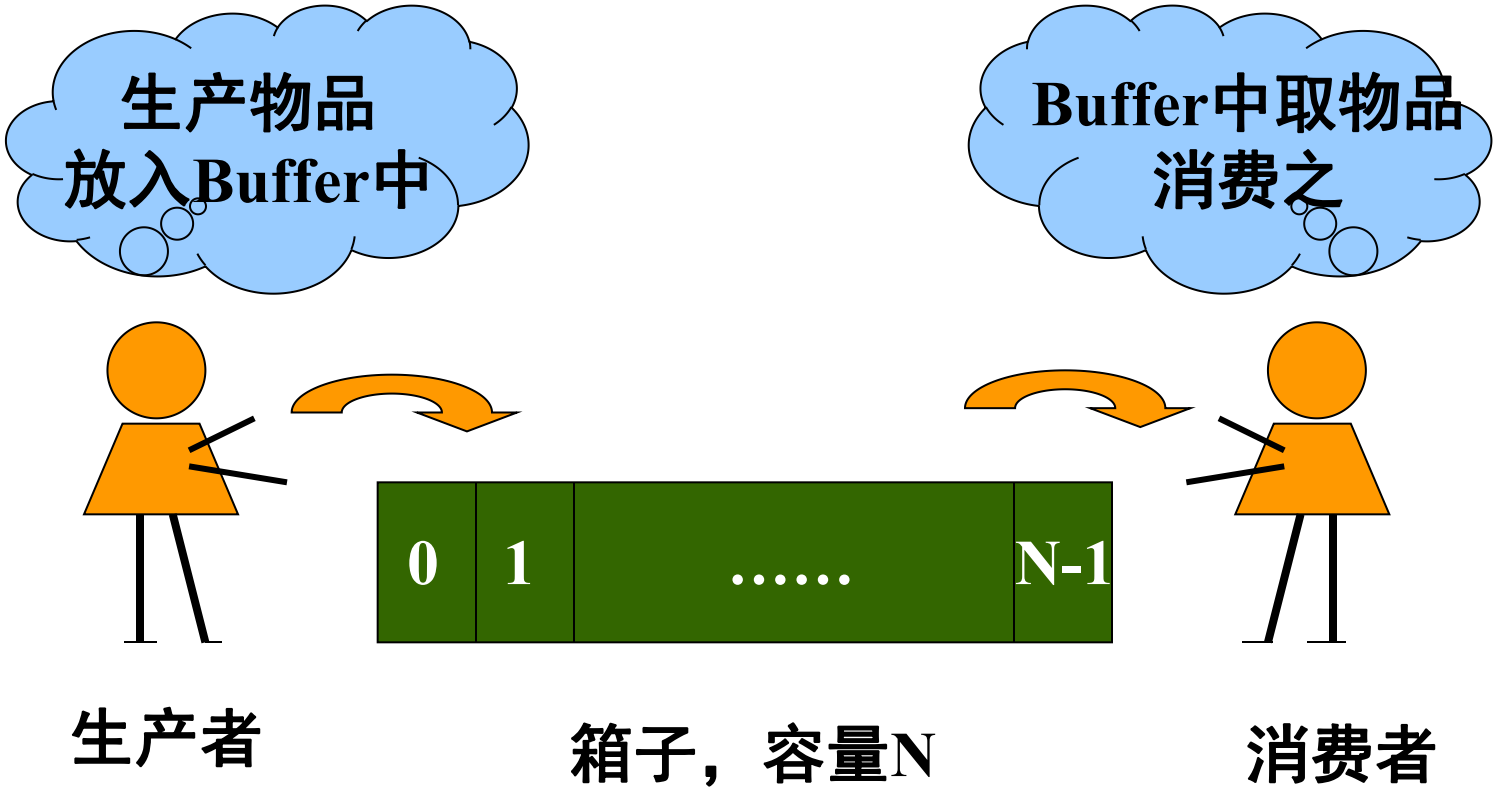
2.6.1 生产者-消费者问题

- 生产者: 能产生并释放资源的进程
- 消费者: 单纯使用（消耗）资源的进程

- 问题表述
  - 一组生产者进程和一组消费者进程（设每组有多个进程）通过缓冲区发生联系。生产者进程将生产的产品（数据、消息等统称为产品）送入缓冲区，消费者进程从中取出产品。
  - 假定缓冲区共有N个，不妨把它们设想成一个环形缓冲池。

2.6.1 生产者-消费者问题

信号量与P、V操作



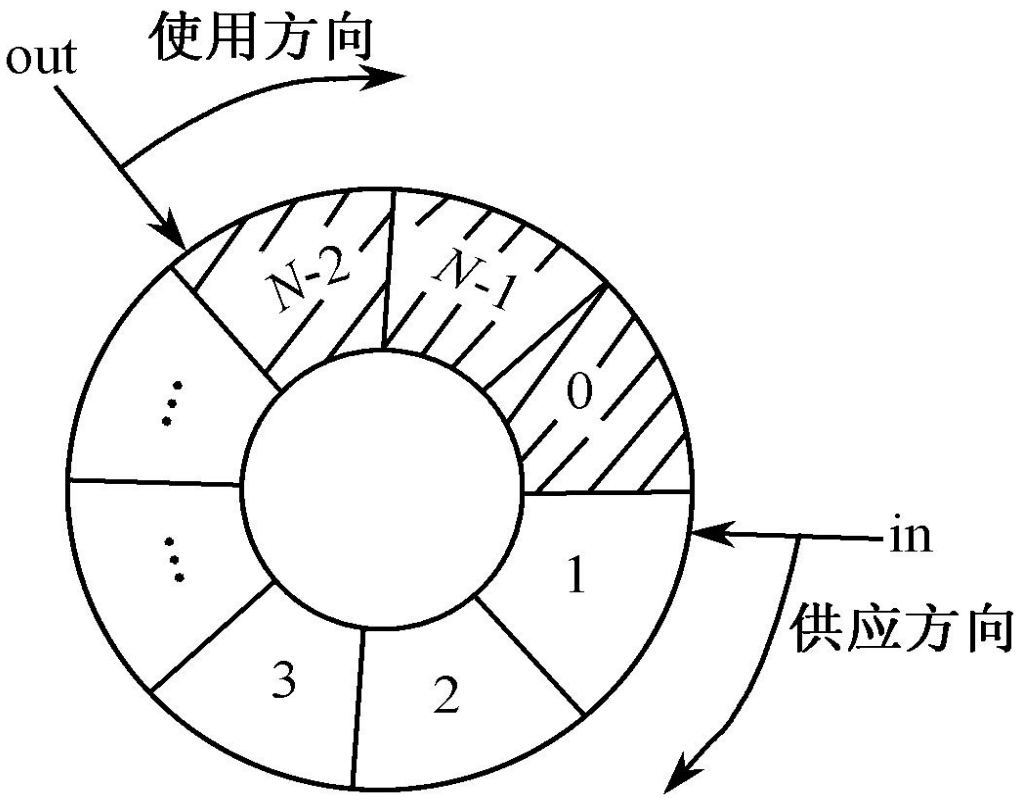
Buffer：Array[0..N-1] Of item

2.6.1 生产者-消费者问题

它们应满足同步条件：

① 任一时刻所有生产者存放产品的单元数不能超过缓冲区的总容量（N）。

② 所有消费者取出产品的总量不能超过所有生产者当前生产产品的总量。



生产者-消费者问题环形缓冲池

2.6.1 生产者-消费者问题

- 设缓冲区的编号为0～N-1，in和out分别是生产者进程和消费者进程使用的指针，指向下面可用的缓冲区，初值都是0。
- 设置三个信号量
  - full：表示放有产品的缓冲区数，其初值为0。
  - empty：表示可供使用的缓冲区数，其初值为N。
  - mutex：互斥信号量，初值为1，表示各进程互斥进入临界区，保证任何时候只有一个进程使用缓冲区。

2.6.1 生产者-消费者问题

生产者进程Producer

```
while(TRUE) {
    P(empty);
    P(mutex);
  产品送往buffer(in);
    in=(in+1)mod N;
     /*以N为模*/
    V(mutex);
    V(full);
}
```

消费者进程Consumer

```
while(TRUE){
    P(full);
    P(mutex);
  从buffer(out)中取出产品;
    out=(out+1)mod N;
       /*以N为模*/
    V(mutex);
    V(empty);
}
```

2.6.2 读者-写者问题

- 读者-写者问题也是一个著名的进程互斥访问有限资源的同步问题。

- 例如，一个航班预订系统有一个大型数据库，很多竞争进程要对它进行读、写。允许多个进程同时读该数据库，但是在任何时候如果有一个进程写（即修改）数据库，那么就不允许其他进程访问它—— 既不允许写，也不允许读。

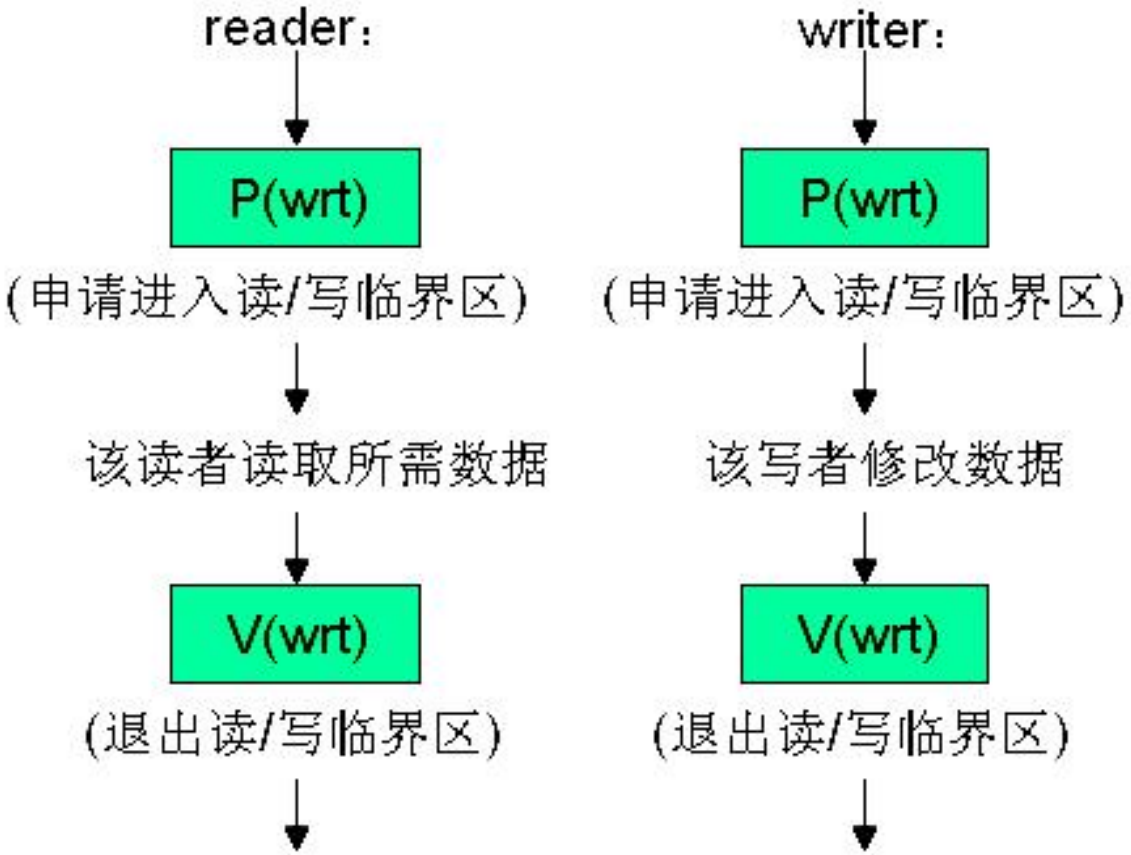
2.6.2 读者-写者问题

- 问题表述： 一些进程只读数据，称“读者”；另一些会对数据进行修改，称“写者”。允许多个读者同时访问数据；若有写者访问数据，那么后来的写者和读者只能等待；若在读者访问数据时来了写者，那么写者只能等待，而后续来到的读者仍允许访问数据。
- 试用P、V操作协调各读者和写者间的工作。

(1) 构筑读者进程和写者进程间的临界区

对这批数据，要保证读者和写者互斥使用，也要保证写者和写者互斥使用。设置信号量 wrt ，用来在读者和写者程序中构成如图所示的临界区。

这种安排在某读者进入临界区使用数据时，其他读者都不能使用该数据，这不符合题目要求！



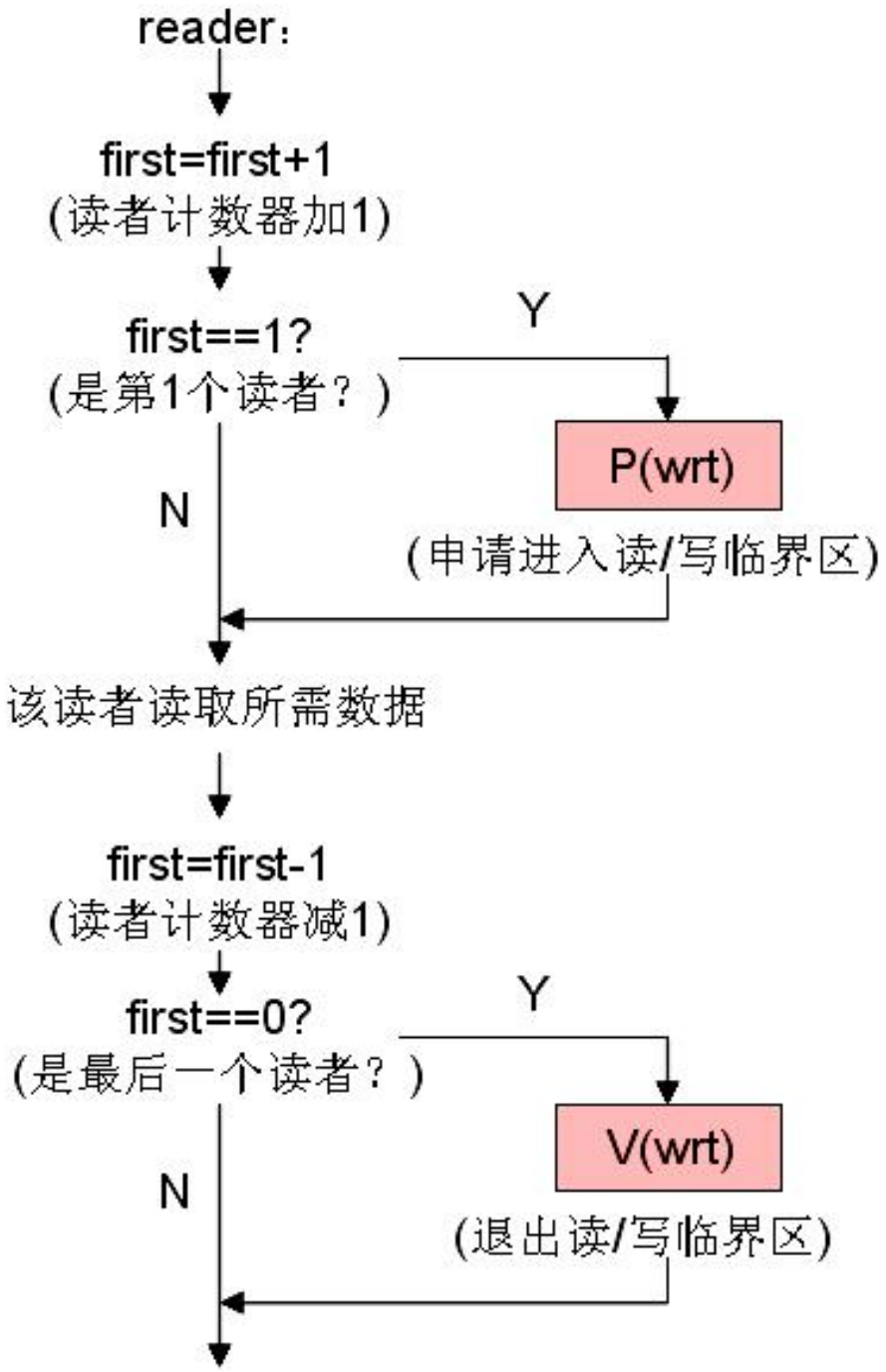
(2) 判定是否是第1个读者

希望在读者进程里，有一个办法能判定请求进入临界区的是否是第1个读者。如果是第1个读者，就对信号量 wrt做P操作，以取得和写者的互斥；如果是后继读者，就不再对wrt 做P操作。

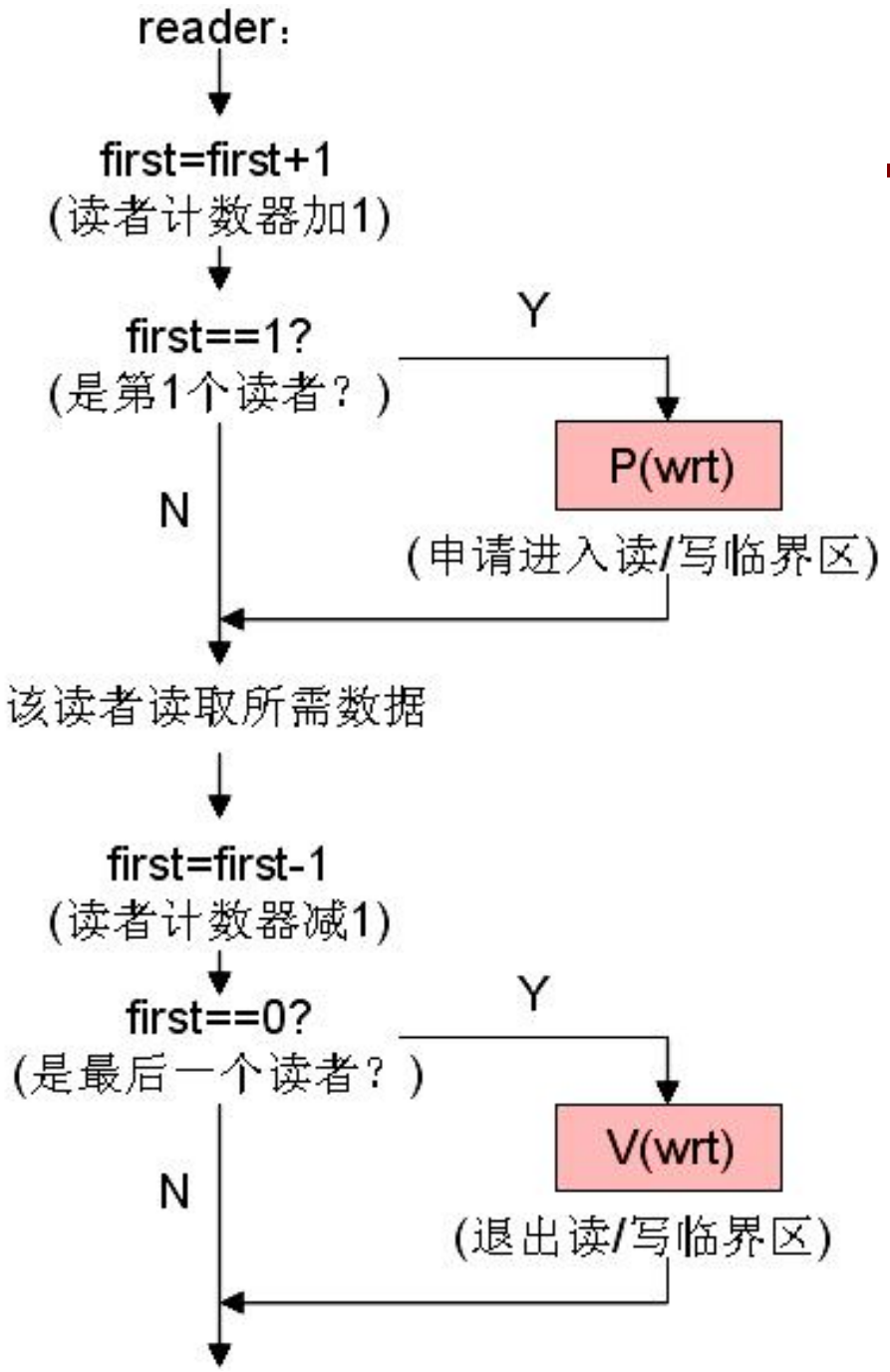
为此，设变量 first(它不是信号量)初值为0。任何一个读者运行时，都先在first上加1，然后判定它是否取值为1，若是1，则做P(wrt) ，否则不做。如图所示。

为此，设变量 first(它不是信号量)初值为0。任何一个读者运行时，都先在 first上加1，然后判定它是否取值为1。若是1，则做P(wrt)，否则不做。

与此同时，不能让每个读者退出临界区时都对信号量 wrt 做V操作，而是要判断它是否是最后一个读者。只有最后一个读者，才会对信号量wrt做V操作，以便能够让写者有机会进入临界区。



这时，**若写者先进入临界区，由于在信号量wrt上做了P操作，就可把随后要求进入临界区的写者和读者都阻挡在临界区外**；若读者先进入临界区，且是第1个读者，那么就会在信号量wrt上做P操作，随后到来的写者就被阻挡在临界区之外，而后继到来的读者由于不是第1个，因此不再对信号量wrt做P操作，也就不会被阻挡在临界区之外了。



(3) first是所有读者共享的公共变量

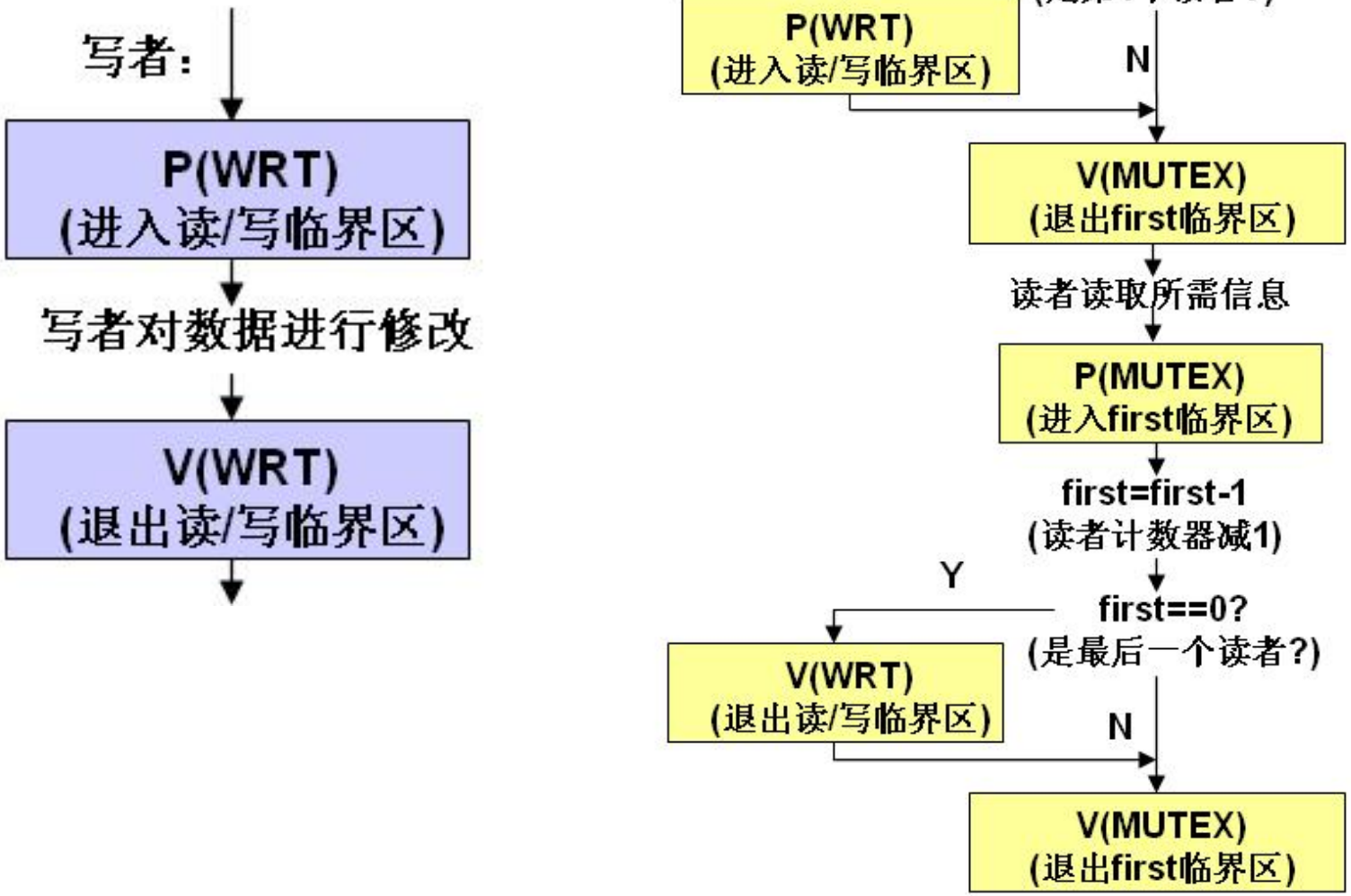
但解法仍然有问题，因它没有考虑到变量first是所有读者都要访问的变量，每个读者在使用它时，应该互斥才对。

所以，必须再设置个初值为1的互斥信号量mutex，由它保证读者不会同时去测试和操作变量first。

2.6.2 读者-写者问题

- 信号量设置：
  - 读互斥信号量mutex，初值为1，读者互斥地访问first
  - 写-写或写-读互斥信号量wrt，初值为1，保证一个写者与其他读者/写者互斥地访问共享资源

★first 是整型读者计数器变量，不是信号量，其初值为0。


写者：
P(WRT)
(进入读/写临界区)
写者对数据进行修改
V(WRT)
(退出读/写临界区)
读者：
P(MUTEX)
(进入first临界区)
first=first+1
(读者计数器加1)
Y
first==1?
(是第1个读者?)
P(WRT)
(进入读/写临界区)
N
V(MUTEX)
(退出first临界区)
读者读取所需信息
P(MUTEX)
(进入first临界区)
first=first-1
(读者计数器减1)
Y
first==0?
(是最后一个读者?)
V(WRT)
(退出读/写临界区)
N
V(MUTEX)
(退出first临界区)

读者Readers

```
while(TRUE){
    P(mutex);
    first=first+1;
    if(first==1)
        P(wrt);
    V(mutex);
    执行读操作
    P(mutex);
    first=first-1;
    if(first==0)
        V(wrt);
    V(mutex);
    使用读取的数据
}
```

写者Writers

```
while(TRUE) {
        P(wrt);
        执行写操作
        V(wrt);
}
```

这个算法隐含读者的优先级高于写者（读者优先）

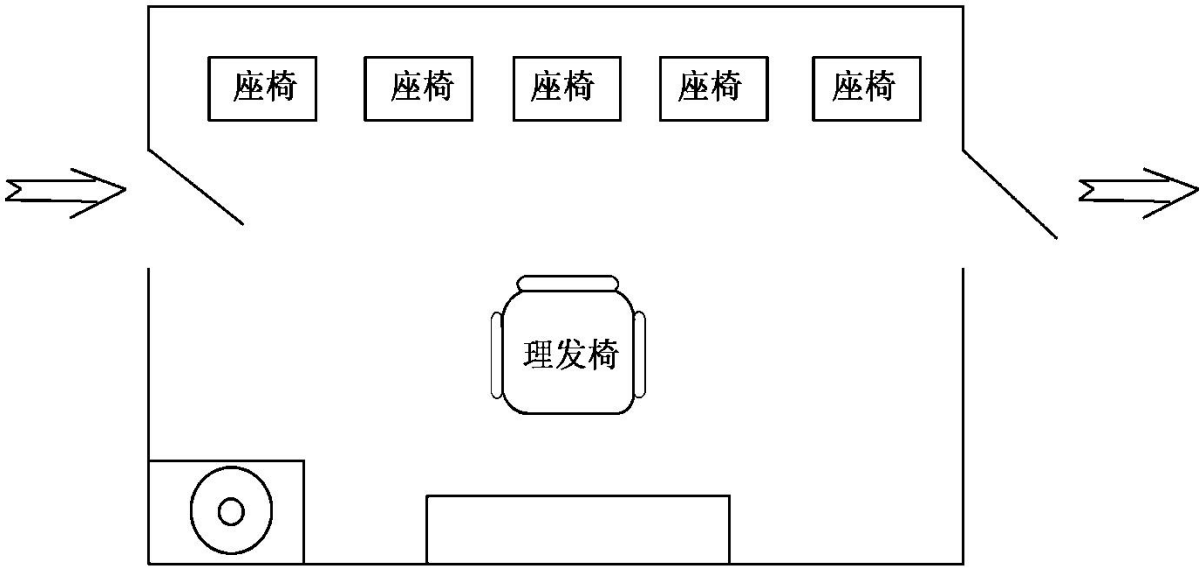
写者优先

```
//读者进程
while(1) {
    P(wblock);
    P(rblock);
    P(rmutex);
    readcount=readcount+1;
    if(readcount==1)
        P(mutex);
    V(rmutex);
    V(rblock);
    V(wblock);
    执行读操作;
    P(rmutex);
    readcount=readcount-1;
    if(readcount==0)
        V(mutex);
    V(rmutex);
}
```

```
//写者进程
while(1) {
    P(wmutex);
    writecount=writecount+1;
    if(writecount==1)
        P(rblock);
    V(wmutex);
    P(mutex)
    执行写操作;
    V(mutex)
    P(wmutex);
    writecount=writecount-1;
    if(writecount==0)
        V(rblock);
    V(wmutex);
}
```

2.6.3 理发师问题

- 设置变量waiting表示等待理发的顾客的数量，初值为0。定义三个信号量：
  - customers：正在等待的顾客的数量，数值上与waiting相同，初值为0。
  - barbers：理发师的状态，初值为0。
  - mutex：用于互斥访问变量waiting，初值为1。



2.6.3 理发师问题

```
int waiting=0;
Semaphore mutex=1,barbers=0,customers=0;
cobegin
void barber( ) {
    while(1) {
        P(mutex);
        if(waiting==0) [睡觉]; //如果没有顾客，理发师打瞌睡
        V(mutex);
        P(customers);
        P(mutex);           //互斥进入临界区
        waiting--;
        v(barbers);         //理发师准备理发
        v(mutex);           //退出临界区
        [给顾客理发];
    }
}
```

2.6.3 理发师问题

```
void customer( )  {
    P(mutex);                  //互斥进入临界区
    if(waiting < n) {
        waiting++;
        V(customers);          //若有必要，唤醒理发师
        V(mutex);              //退出临界区
        P(barbers);            //如果理发师忙着，则顾客打瞌睡
        [坐下等待];
    }else
        V(mutex);              //店里人满了，不等了，离开
}
coend;
```

2.6.4 独木桥问题

- 有一座比较窄的、东西朝向的独木桥。桥两头的汽车可以通过该桥，但为了保证安全，需要执行如下规则：桥上无车的，允许一侧汽车过桥，待一侧的汽车全部通过之后，才允许另一侧汽车通过。

- 3个信号量
  - mutexBrigde：初值为1，用于独木桥的互斥访问。
  - mutexEast：初值为1，表示东侧汽车上桥后的东侧汽车之间互斥。
  - mutexWest：初值为1，表示西侧汽车上桥后的西侧汽车之间互斥。

- 2个计数器变量
  - eastCount：初值为0，东侧汽车在桥上的数量。
  - westCount：初值为0，西侧汽车在桥上的数量。

2.6.4 独木桥问题

```
int eastCount=0,westCount=0;
Semaphore mutexBrigde=1,
mutexEast=1, mutexWest=1;
cobegin
void eastPart {
  while(1) {
  P(mutexEast);
  eastCount++;
  if(eastCount==1)
     P(mutexBrigde);
  V(mutexEast);
  [上桥];
  汽车行进;
  [下桥];
  P(mutexEast);
  eastCount--;
  if(eastCount==0)
     V(mutexBrigde);
  V(mutexEast);
  }
}
```

```
void westPart {
  while(1) {
  P(mutexWest);
  westCount++;
  if(westCount==1)
     P(mutexBrigde);
  V(mutexWest);
  [上桥];
  汽车行进;
  [下桥];
  P(mutexWest);
  westCount--;
  if(westCount==0)
     V(mutexBrigde);
  V(mutexWest);
  }
}
coend;
```

2.8 进程的通信

- 并发进程之间交换信息
  - PV操作交换信息实现进程的同步与互斥
  - 只交换了少量的信号
    - 低级通信方式
  - 进程间要交换大量的信息？-- “进程通信”
    - 设计专门的通信机制
    - 高级的通信方式
- “进程通信”
  - 指通过专门的通信机制实现进程间交换大量信息的通信方式

2.8.1 进程通信的类型

- 低级通信方式
  - 信号量
- 高级通信方式
  - 共享存储器系统
  - 消息传递系统
  - 管道通信

1. 共享存储器系统

- 为了传输大量数据，在存储器中划出一块共享存储区，相互通信的进程共享某些数据结构或共享存储区。
  - 基于共享数据结构的通信方式：进程间通过某种类型的数据结构交换信息，设置共享数据结构和进程间同步的处理由程序员完成，操作系统仅提供共享存储器。效率低、适于传递少量信息。
  - 基于共享存储区的通信方式：在存储中划出一块共享存储区，进程间对共享存储区读写实现通信。

2. 消息传递系统

- 消息系统中，进程间的信息交换以消息或报文为单位，程序员直接利用系统提供的一组通信命令(原语)来实现通信。
- 消息系统的通信方式可分成以下两种：
  - 直接通信方式
    - Send（receiver，　message);
    - Receive（sender，　message);
  - 间接通信方式
    - 在间接通信方式中，进程之间需要通过某种中间实体（信箱），来暂存发送进程发送给某个或某些目标进程的消息。接收进程则从中取出发送给自己的消息。

3 管道通信

- 管道通信是指用于连接一个写进程和一个读进程的一个共享文件。这是基于原有的文件系统形成的一种通信方式，即它利用共享文件实现进程间的通信。
- 为了协调双方的通信，管道通信机制必须提供三方面的协调能力：
  - 互斥
    - 当一个进程正在对pipe进行读／写操作时，另一进程必须等待。
  - 同步
    - 当写(输入)进程把一定数据(如4kB)写入pipe后便去睡眠等待，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，直到写进程将消息写入管道后，才将它唤醒。
  - 判断对方是否存在
    - 只有已确定对方存在时，方能进行通信。

2.8.2 直接通信

- 直接通信
  - 指发送进程利用操作系统所提供的发送命令直接把消息发送给接收进程，而接收进程则利用接收命令直接从发送进程接收消息

- 每个进程必须指出信件发给谁或从谁那里接收消息
  - 调用send原语和receive原语来实现进程之间的通信
  - send(P，消息)：把一个消息发送给进程P
  - receive(Q，消息)：从进程Q接收一个消息

  - 这一种联结仅仅发生在这一对进程之间
  - 消息可以有固定长度或可变长度两种

2.8.3 间接通信

- 间接通信方式
  - 进程间发送或接收消息通过一个共享的数据结构（信箱）来进行
    - 消息→信件
    - 每个信箱有一个唯一的标识符
  - “发送”和“接收”原语的形式
    - send(A，信件)：把一个信件（消息）发送给信箱A
    - receive(A，信件)：从信箱A接收一封信件（消息）
- 信箱
  - 存放信件的存储区域
  - 每个信箱可以分成信箱头和信箱体两部分
    - 信箱头指出信箱容量、信件格式、存放信件位置的指针等
    - 信箱体用来存放信件，信箱体分成若干个区，每个区可容纳一封信

2.8.3 间接通信

- “发送”和“接收”两条原语的功能为：
  - 发送信件。如果指定的信箱未满，则将信件送入信箱中由指针所指示的位置，并释放等待该信箱中信件的等待者；否则，发送信件者被置成等待信件状态。
  - 接收信件。如果指定信箱中有信，则取出一封信件，并释放等待信箱的等待者，否则，接收信件者被置成等待信箱中信件的状态。

```
Typedef box=record
int size;     //信箱大小
int count;   //现有信件数
message letter[n]; //信箱
Semaphore s1,s2;
//等信箱和等信件信号量
cobegin
void send(box B,message M) {
   int i;
   if(B.count==B.size) W(B.s1);
   i=B.count+1;
   B.letter[i]=M;
   B.count=i;
   R(B.s2);
}
```

```
void receive(box B) {
   int i;
   if(B.count==0) W(B.s2);
   B.count=B.count-1;
   x=B.letter[1];
   if(B.count!=0)
     for(i=1;i<B.count;i++)
       B.letter[i]=B.letter[i+1];
   R(B.s1);
   return x;
}
coend;
```

2.8.3 间接通信

- 信箱可由操作系统创建，也可由用户进程创建
- 信箱分为三类：

（1）私用信箱

- 用户进程自己建立，作为进程的一部分
- 信箱的拥有者有权从信箱中读取信息，其他用户只能将消息发送到该信箱
- 采用单向通信链路实现
- 当拥有该信箱的进程结束时，信箱也随之消失

（2）公用信箱

- 操作系统创建，提供给系统中的所有核准进程使用
- 核准进程发送消息到信箱中，也可从信箱中取出发送给自己的消息
- 公用信箱采用双向通信链路的信箱来实现
- 在系统运行期间始终存在

（3）共享信箱

- 由某进程创建，指明它是可共享的，同时指出共享进程（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

- 信箱通信时，发送进程和接收进程之间存在的四种关系：

① 一对一关系：为发送进程和接收进程建立一条专用的通信链路。使它们之间的交互不受其他进程的影响。

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。

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形式，向接收者发送消息。

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，也可以从信箱中取走属于自己的消息。

2.10 本章小结

- 顺序执行是严格按照程序规定的指令顺序去执行，其结果与它的执行速度无关（即与时间无关）。
- 并发执行是一组在逻辑上互相独立的程序，在执行过程中的执行时间在客观上互相重叠。
- 并发执行具有间断性、失去封闭性、不可再现性的特征。
- 人们引入进程从变化的角度，动态地分析研究程序的并发执行过程。
- 进程是可并发执行的程序在一个数据集上的一次执行过程，它是系统进行资源分配的基本单位。
- 进程具有动态性、并发性、独立性、异步性、结构性等五个基本特征。
- 进程有就绪、执行和等待三个基本状态，并可以进行转换。

2.10 本章小结

- 人们提出了比进程更小的、能独立运行的基本单位——线程，以进一步提高程序并发执行的程度，降低并发执行的时空开销。
- 我们把并发进程中与共享资源有关的程序段称为“临界区”。
- 采用PV操作及管程的方法来解决临界区的互斥问题。
- 采用PV操作及管程的方法来解决进程同步问题。
- 通过专门的通信机制实现进程间交换大量信息。
- 共享存储器系统、消息传递系统以及管道通信系统三大类机制。
- 直接通信和间接通信两种通信方式。