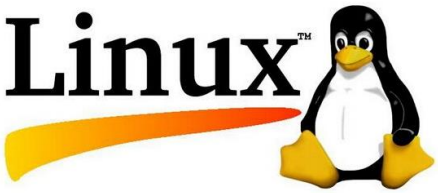
操作系统



2

进程管理

目录

2.1 概述

- **进程：**一个运行的程序（或程序的运行）
  - 同一程序同时被运行两次，就是两个独立的进程——代码相同，但所占用的系统资源、所处理的数据以及运行状态不同
- 在有限的软硬件资源上允许“同时”运行多个程序时，就需要**进程管理**
- 现代操作系统中，**线程**代替进程成为CPU调度的基本单位

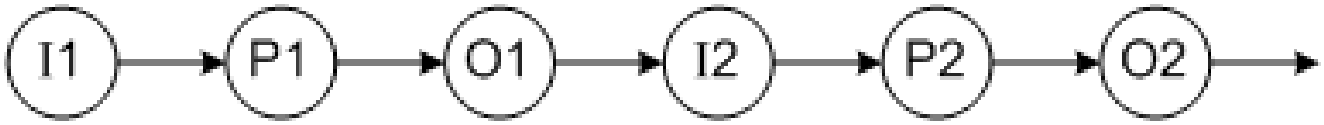
2 2.2 程序的执行方式

- 程序的执行是指将二进制代码文件（如*.exe等）装入内存，由CPU按程序逻辑运行指令的过程
  - ✓ **顺序**
  - ✓ **并发**

- 单道程序设计技术是指内存一次只能允许装载一个程序运行，在这次程序运行结束前，其它程序不允许使用内存
- 多道程序设计技术允许多个程序进驻内存，系统通过某种调度策略交替执行程序

2.2.1 程序的顺序执行

- 一个具有独立功能的程序独占处理器直至最终结束的过程称为程序的顺序执行
  - 程序设计中的顺序控制结构仅能控制程序内部指令的执行序列
  - 程序的顺序执行意味着程序间的执行序列也是顺序的——一个程序执行完，才能执行另一个程序



2.2.1 程序的顺序执行

- 顺序执行的特性：
  1. 顺序性
  2. 封闭性
  3. 可再现性

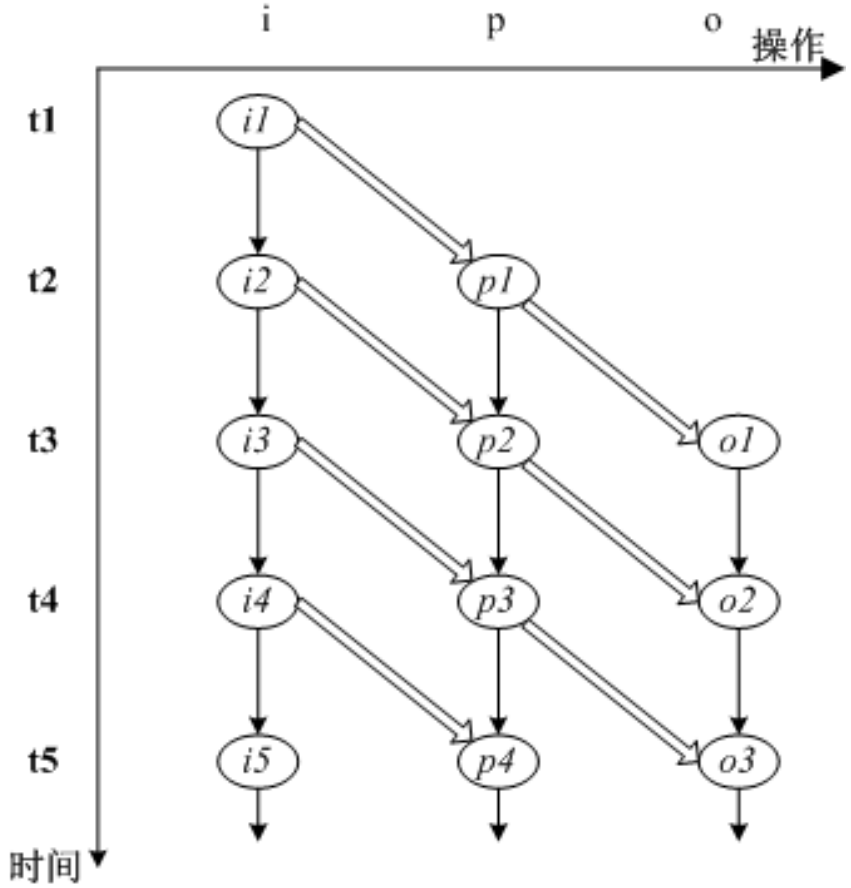
- 顺序执行方式便于程序的编制与调试，但不利于充分利用计算机系统资源，运行效率低下

2.2.2 程序的并发执行与并行执行

- 为了提高系统的运行效率，允许“同时”执行多个程序
  - 并行（parallel）——多个事件在同一时刻发生
  - 并发（concurrent）——多个事件在同一时期内发生
- 并行是并发的特例，程序并行执行的硬件前提是系统中有多个CPU
- 并发的本质是一个CPU在多个程序运行过程中的时分复用

2.2.2 程序的并发执行与并行执行

- 适当的程序设计可以充分发挥不同硬件设备间并行工作的能力



2.2.2 程序的并发执行与并行执行

- 并发执行的特性：
  1. **间断性**
  2. **开放/交互性**
  3. **不可再现性**

- 并发程序设计应当避免由于程序间开放交互引起的不可再现性而产生运行时错误

2.2.3 进程概念的引入

- 相关术语
  - **程序（program）**——静态的代码文件（*.exe）
  - **进程（process）**——可并发程序在某个数据集合上的一次执行过程，是操作系统资源分配、保护和调度的基本单位
  - **作业（job）**——批处理系统要装入系统运行处理的一系列程序步骤和数据

2.3 进程的特征与控制

进程的特征

结构性

PCB（进程控制块） + Code（程序块） + Data（数据块） + Stack（堆栈） == Process Image（进程影像）

动态性

生命周期（创建→调度→结束）

独立性

资源分配、调度单位，进程间独立也可通信

并发性

在一段时间内，若干进程共享一个CPU，并发执行

2.3 进程的特征与控制

- 进程分为**系统进程**和**用户进程**，区别：
  1. 系统进程是操作系统管理系统资源并行活动的并发软件；用户进程是可以独立执行的用户程序段，是操作系统提供服务的对象，是系统资源的实际使用者
  2. 系统进程之间的关系由操作系统负责，有利于增加并行性，提高资源利用率；用户进程之间的关系主要由用户负责，操作系统提供一套任务调用命令作为协调
  3. 系统进程直接管理软、硬设备的活动；用户进程只能间接地使用系统资源，必须向系统提出请求
  4. 进程调度中，系统进程的优先级高于用户进程

2.3 进程的特征与控制

- **进程上下文（process context）：**进程的生命周期中，进程实体和支持进程运行的环境
  1. **用户级上下文（user-level context）**——进程的代码区、数据区、用户栈区和共享存储区；编译成目标文件时生成，占据进程的虚拟地址空间
  2. **系统级上下文（system-level context）**——PCB、内存管理信息、进程环境块、系统栈
  3. **寄存器上下文（register context）**——程序状态寄存器、各类控制寄存器、地址寄存器和用户栈指针
- 一个进程被系统调度而占有CPU时，会发生CPU在新老进程之间切换，切换的内容是进程上下文，进程运行是在进程的上下文中执行的

一个典型的上下文切换过程

```
context_switch( ){
    Push registers onto stack
    Save ptrs to code and data
    Save stack pointer     //以上语句保护当前进程上下文
    Pick next process to execute//选中/调度新进程
    //以下语句恢复所选中/调度的进程的上下文
    Restore stack ptr of that process
    Restore ptrs to code and data
    Pop registers
    Return
}
```

2.3.1进程状态及转换

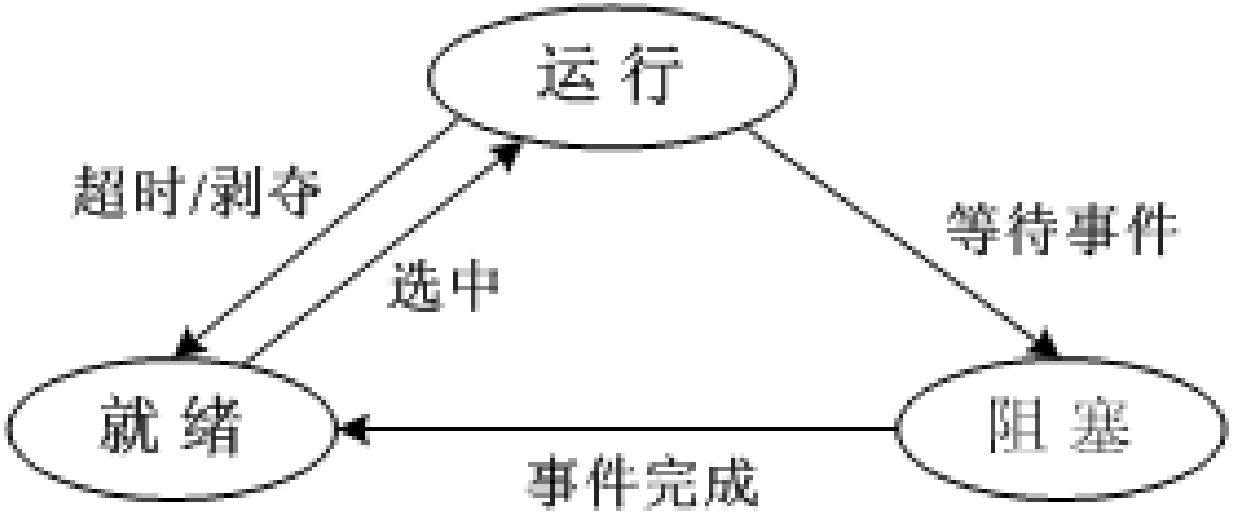
进程三态模型

- 就绪状态（ready）——进程在内存中已经具备执行的条件，等待分配CPU
- 运行状态（running）——进程占用CPU并正在执行
- 阻塞状态（blocked，也称等待（waiting）状态）——运行的进程由于发生某事件而放弃CPU

- **就绪队列 & 阻塞队列：多个进程处于就绪/阻塞状态**

2.3.1进程状态及转换

- **三态模型：**



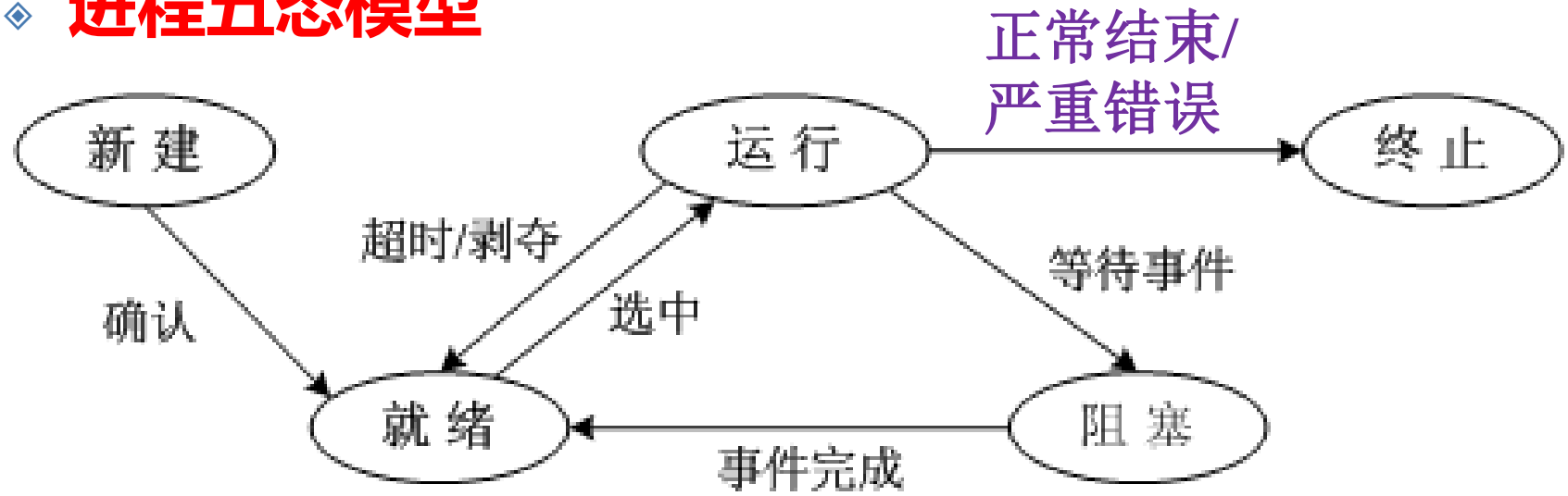
2.3.1进程状态及转换

状态转换

- 就绪状态→运行状态
- 运行状态→阻塞状态
- 阻塞状态→就绪状态
- 运行状态→就绪状态

2.3.1 进程状态及转换

- **进程五态模型**

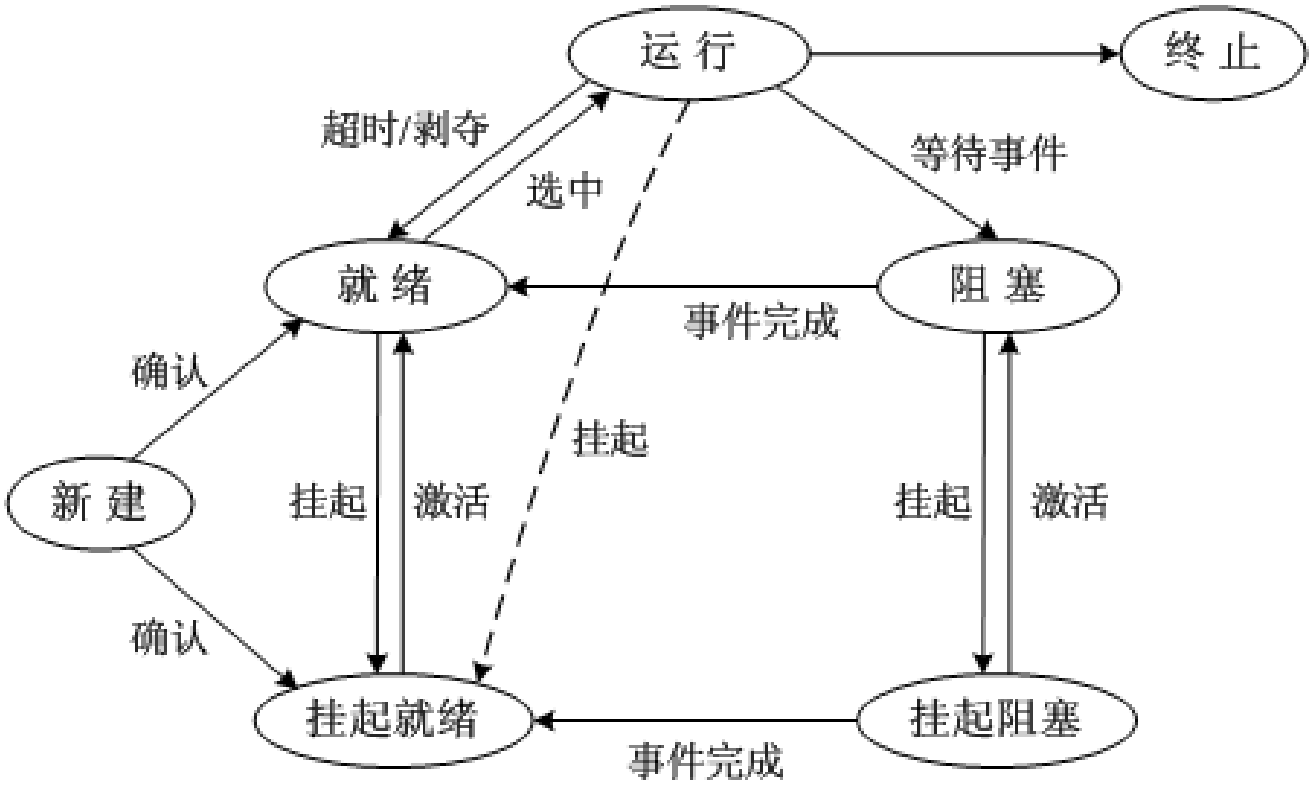


  - 新建状态（new）
  - 终止状态（terminated）

- **Q: 操作系统进程过多，内存中不能满足所有进程的运行要求，怎么办?**

2.3.1 进程状态及转换

A: 挂起——将进程从内存交换到磁盘上，暂时释放所占用的部分资源。挂起条件独立于等待事件，只能由操作系统或其父进程解除



2.3.1 进程状态及转换

Q: 什么时候需要挂起进程?

A: 系统故障，用户调试程序，父进程对子进程试试控制、修改和检查，某些定期执行的进程在时间未到而等待

Remark：挂起条件独立于等待事件，只能由操作系统或其父进程解除

2.3.1进程状态及转换

- 有挂起功能的进程状态转换
  - 挂起就绪（ready suspended）：具备运行条件，但不在内存中，需系统调入内存才能运行
  - 挂起阻塞（blocked suspended）：等待某一事件，切不在内存中
  - 进程在运行态也可以被挂起，转换为挂起就绪状态
  - 阻塞状态的进程被挂起后，若阻塞事件或I/O请求完成，则进程状态转换为挂起就绪状态——仍然是挂起状态
  - 创建进程时若没有足够的内存空间，则转入挂起就绪状态
  - 只有处于就绪态的进程才有可能被调度分配CPU运行

关键转换

- 阻塞－挂起阻塞：内存紧张，且当前不存在就绪进程，系统根据资源分配状况和性能要求选择一个阻塞进程交换出去，让它挂起以便接收新的进程
- 挂起阻塞－挂起就绪：等待的事件（I/O）完成
- 挂起就绪－就绪：内存中不存在就绪态进程，或者挂起就绪态进程具有比当前就绪态进程更高的优先级
- 就绪－挂起就绪：内存紧张，系统把就绪态进程交换出去
- 挂起阻塞－阻塞：主存拥有足够空间，而某个挂起阻塞态进程具有更高优先级别，且系统得知导致该进程阻塞的事件也即将结束
- 运行态－挂起就绪态：具有更高优先级的挂起阻塞态进程所等待的事件完成后，要抢占CPU，而此时主存空间不够
- 新建态－挂起就绪态：内存有限，将新建进程挂起

2.3.2 进程控制块PCB

- **进程控制块（Process Control Block，进程描述符，Process Descriptor）——进程存在的惟一标志，描述和控制进程运行的数据结构**

1. 进程标识信息——内部标识符（PID）和外部标识符（进程名），**ps命令**
2. 现场信息——进程运行时CPU的即时状态（各寄存器的值）
3. 控制信息——程序和数据地址、进程同步和通信机制信息、进程的资源清单和链接指针，进程状态、进程优先级……

2.3.2 进程控制块PCB

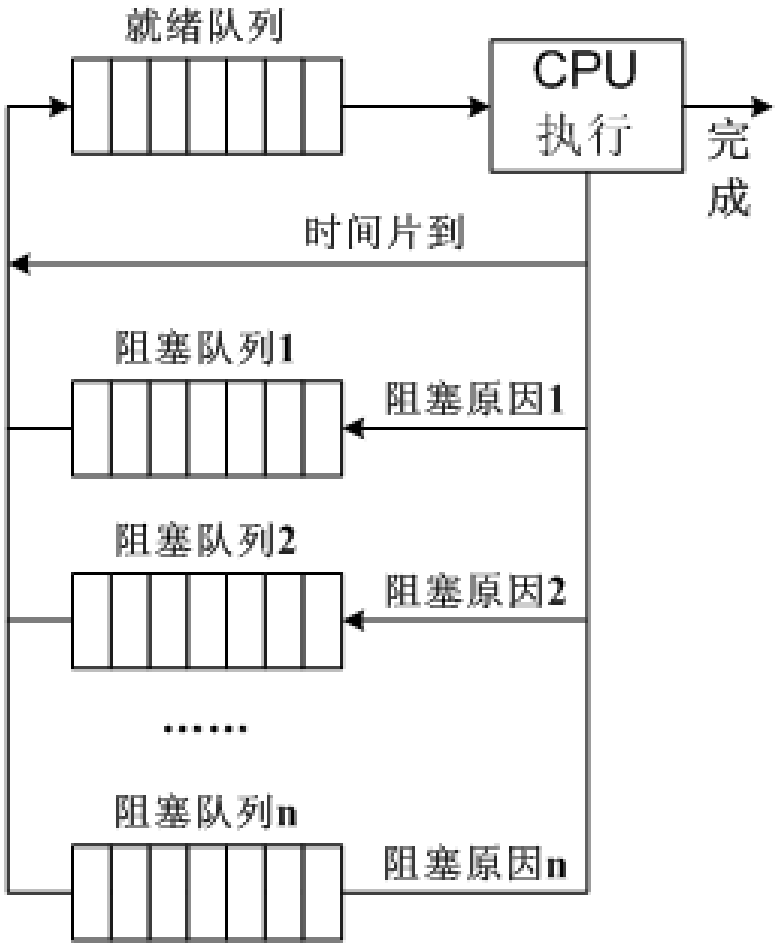
- 操作系统根据PCB对进程进行控制和管理
  1. 查询进程现行状态及优先级
  2. 恢复现场
  3. 相关进程同步、通信

- 创建新进程时，建立PCB，进程结束时回收PCB

- PCB由系统多个功能模块读写，须常驻内存，并进一步组织形成队列（运行队列、就绪队列、阻塞/等待队列）

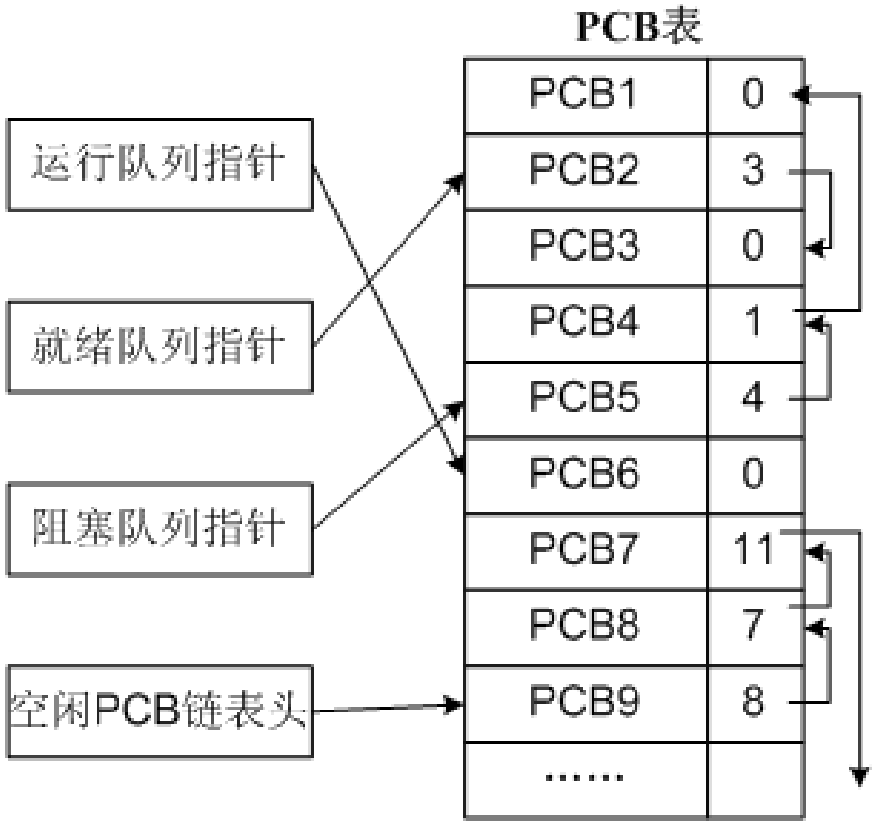
2.3.2 进程控制块PCB

PCB组织方式——多进程队列



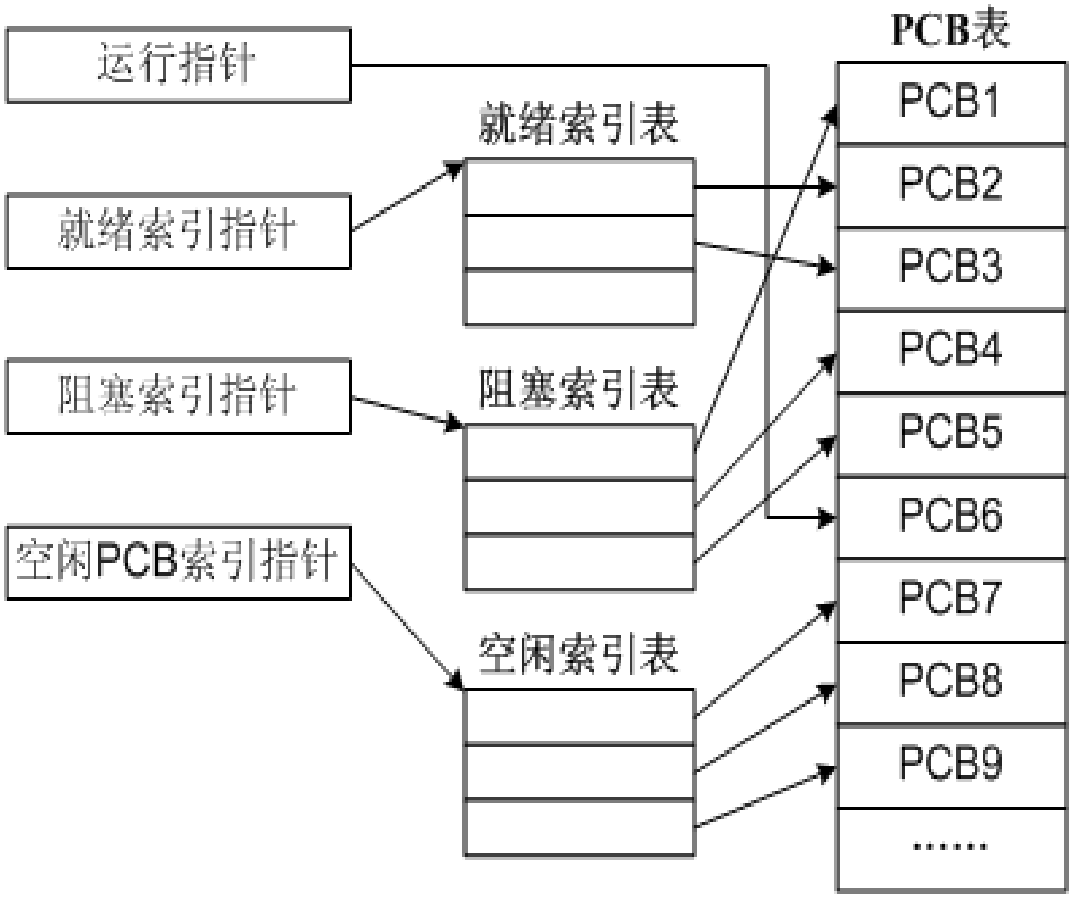
2.3.2 进程控制块PCB

PCB组织方式——PCB链接方式



2.3.2 进程控制块PCB

PCB组织方式——PCB索引方式



2.3.3 进程控制

- CPU的运行模式
  - 核心态（内核态），Ring0
    - 内核代码、设备驱动
    - 特权指令，直接访问物理内存空间、设备端口
  - 用户态，Ring3
    - 保护模式安全限制
    - 普通指令，访问映射的虚拟地址空间、系统许可的映射端口

- CPU模式切换不同于进程切换，也不一定引起进程切换或状态转换
  - 当发生中断或系统调用时，用户进程暂停，CPU模式从用户态切换到核心态，执行系统服务例程，此时进程仍在原上下文中运行，仅模式变化

2.3.3 进程控制

- **进程控制**——系统对进程生命周期的各个环节进行控制

- **进程控制的职能：**对系统中的所有进程实行有效的管理——对进程进行创建，撤销或终止，以及在某些进程状态间的转换控制

2.3.3 进程控制

- 进程控制通常由原语（primitive）完成
- 原语：
  - 由若干条指令组成，实现某个特定功能，在执行过程中不可被中断的程序段
  - 不可分割的执行单位，不能并发执行
  - 操作系统核心（不是由进程而是由一组程序模块组成）的组成部分，且常驻内存
  - 通常在核心态/管态下执行

2.3.3 进程控制

- 进程控制原语——创建进程
  - 建立PCB、填入信息、插入就绪队列
  - 创建进程主要步骤：
    1. 命名进程：为新进程设置进程标志符
    2. 从PCB集合中为新进程申请一个空PCB
    3. 确定新进程的优先级
    4. 为新进程的程序段、数据段和用户栈分配内存空间；如果进程中需要共享某个已在内存的程序段，则必须建立共享程序段的链接指针
    5. 为新进程分配除内存外的其它各种资源
    6. 初始化PCB，将新进程的初始化信息写入进程控制块
    7. 如果就绪队列能够接纳新创建的进程，则将新进程插入到就绪队列
    8. 通知操作系统的记账、性能监控等管理模块

2.3.3 进程控制

- **导致创建进程的事件**
  - 用户登录、作业调度、提供服务——系统内核直接调用创建原语创建新进程
  - 应用请求——由用户调用操作系统提供的系统调用完成
    - Windows的API——**CreateProcess**
    - Linux ——**fork()**

2.3.3 进程控制

- 进程控制原语——撤消与终止进程
  - 进程完成后，应退出系统，系统及时收回占有的全部资源以便其它进程使用
  - 撤销原语撤销的是PCB，而非进程的程序段
- 进程控制原语——阻塞与唤醒进程
  - 阻塞是进程的自主行为
  - 唤醒是被动的
- 进程控制原语——挂起与激活进程
  - 可以由进程自己调用，也可由其它进程或系统调用，但激活原语只能由其它进程或系统调用

2 2.4 进程的互斥与同步

- 并发运行的多个进程之间存在两种基本关系
  - 竞争——会引发以下两种极端情况：
    - 死锁（deadlock）——一组进程都陷入永远等待的状态
    - 饥饿（starvation）——被调度程序长期忽视
  - 协作——同步（synchronization）
    - 一个进程的执行依赖于其协作进程的消息或信号
- 互斥（mutual exclusion）
  - 若干进程因互相竞争独占性资源而产生的竞争制约关系
  - 互斥也是一种特殊的同步——以一定次序协调地使用共享资源

2.4.1与时间有关的错误

- 若两个进程共享了数据集，则可能存在制约关系，形成交互的并发进程
- 执行的相对速度无法相互控制，就会出现与时间有关的错误

多终端系统：银行ATM、售票系统……

```
void T1( ) {
    int x=Q;      //A
    if(x>0){
        x--;      //B
        Q=x;      //C
        打印机票;
    }
    else 打印“售罄”；
}
```

A ↓ D ↓ B ↓ C ↓ E ↓ F

```
void T2( ) {
    int x=Q;      //D
    if(x>0){
        x--;      //E
        Q=x;      //F
        打印机票;
    }
    else 打印“售罄”；
}
```

2.4.2临界资源与临界区

- 临界资源：在某段时间内只能允许一个进程使用的资源
  - 打印机、磁带机等硬件设备和变量、队列等数据结构
- 临界区（critical section）：进程中访问临界资源的代码段
- 几个进程若共享同一临界资源，它们必须以互斥的方式使用临界资源
  - 上述售票系统中公用变量Q为临界资源，T1中的A~C语句和T2中的D~F语句为临界区

2.4.2临界资源与临界区

- 临界区调度原则：

  1）一次至多一个进程能够执行其临界区代码；

  2）如果已有进程在临界区运行，其它试图进入的进程应等待；

  3）进入临界区内的进程应在有限时间内退出，以便让等待进程中的一个进入

- 选择临界区调度策略时，不能因为该原则而造成进程饥饿或死锁
- 实现临界区管理有软件和硬件两种方式

2.4.2临界资源与临界区

- 软件方法管理临界区
  - 实施依据是内存访问的基本互斥性——对内存同一地址的并发访问将被存储管理器序列化，而访问的顺序并无需事先指定
  - 不需要硬件、操作系统或程序设计语言的任何支持

早期的临界区管理方法

先判断对方进程是否在临界区的进程标志法

- 进程i访问临界资源前，先查看其他进程访问临界资源的标志，若发现其他进程正访问临街资源，则进程i等待。
- 当其他进程标志处于没有访问临界资源时，进程i才能进入临界区访问。
- 设置数组flag[]为每个进程是否访问临界资源的标志。
- 对进程i，flag[i]为true，标志进程i正在临界区访问；flag[i]为false表示进程i没有访问临界区。

```
turn: integer
   var flag: array[1,2,...,n] of Boolean;
   flag[i] = false;      /* flag[]初始化为false */
   flag[j] = false;
   f_Pi;                 /* 创建进程Pi */
   f_Pj;                 /* 创建进程Pj */
 cobegin
```

```
Pi:          /* 对进程i */
begin
 while flag[j] do nothing;
 flag[i] = true;
 critical section;
/* 访问临界区 */
 flag[i] = false;
 ……
  end;
```

```
 Pj          /* 对进程j */
begin
……
while flag[i] do nothing;
flag[j] = true;
critical section;
 /* 访问临界区 */
flag[j] = false;
……
end;
```

```
coend;
```

早期的临界区管理方法

存在问题：

- 当进程i和进程j都没有进入临界区时，即各自的访问标志flag都为false时，进程while语句通过，进程进入下一个语句时设置进程自己的访问标志，双方都将自己的访问标志设置成true，都要访问临界区。此时，进程i和进程j都进入临界区访问，发生冲突，违背了“忙则等待”的同步准则。
- 当某个进程要访问临界区已经将自己的标志置为true之后，进程又放弃了临界区的访问，从而会引起其它进程一直等待。

2.4.2临界资源与临界区

- Peterson算法（1981年）
  - 为每个进程设置标志flag用于表示该进程是否有意访问临界资源（进入临界区），又设置标志turn用于表示临界资源此时是否有其它进程在访问
  - 只有在对方进程的访问标志flag为true并且turn也为该进程标识时，才表明对方进程在访问临界资源，需要等待对方进程访问完并释放资源后才能访问；否则本进程不需要等待对方进程即可访问临界资源

```
//Peterson算法:
boolean flag [2]={false, false};
int turn;
cobegin
void P0( ){
    while (true) {
        flag [0] = true;
        turn = 1;
        while( flag[1]&& turn==1] ;
        临界区
        flag [0] = false;
        ……
    }
}
coend
```

```
void P1( ){
    while (true) {
        flag [1] = true;
        turn = 0;
        while( flag[0]&&
    turn==0] ;
        临界区
        flag [1] = false;
        ……
    }
}
```

- 虽然现在很少用软件算法实现临界区管理问题，但这些算法对理解同步问题还是很有指导意义的

2.4.2临界资源与临界区

- 软件方法管理临界区的标志算法比较容易出现标志逻辑混乱的情况，其根本原因在于管理临界区标志要用两条指令：
  - 一条指令是看对方的标志
  - 一条指令是设置自己的标志
- 进程并发可能导致进程在执行这两条指令时被另一个进程中断
- 保证进程在执行这两条指令时不被中断，即可很容易地进行临界区管理

2.4.2临界资源与临界区

- 硬件方式管理临界区
  1）禁止中断法
  - 在检查临界区标志的两条指令之前将中断关上，临界区访问完后系统才打开中断
  - 缺点：影响计算机效率、不能及时处理重要程序、对多CPU系统无效

  2）特殊指令法
  - 特殊的硬件指令保证几个动作的原子性——不会被中断，不受到其它指令的干扰
  - “测试并设置（Test and Set）”指令TS，或者交换（exchange）指令SWAP

2.4.2临界资源与临界区

- TS指令功能：

```
bool TS (bool & x){
    if (x == true){
        x = false;
        return true;
    }
    else return false;
}
```

- 将布尔变量x与临界区关联起来——如果x为真，表示没有进程在临界区内，临界资源可用，并立即将x置为false，阻止其它进程进入临界区；若x为假，则表示有其它进程进入临界区，本进程需要等待。

2.4.2临界资源与临界区

- TS指令实现互斥：

```
bool x = true ;
cobegin                    // 并发段开始
process Pi ( ){            // i = 1，2，3……
   while ( !TS(x) );
   临界区
   x = true;
}
coend                      // 并发段结束
```

2.4.2临界资源与临界区

- **用交换指令实现互斥：**

```
bool lock=false;                    //表示无进程进入临界区
cobegin
process Pi( ) {                     //i =1，2，3……
    bool ki = true;
    do {
      SWAP( ki, lock );
    }while(ki);                     //上锁
    临界区
    SWAP( ki, lock );               //开锁
}
coend
```

2.4.3进程同步机制

- 常见的同步机制有锁、信号量、管程和消息传递
  - 锁机制的开锁和关锁原语，主要用于解决互斥问题，但效率低、浪费CPU，加重编程负担
  - 1965年*E.W.Dijkstra*引进了比开锁和关锁原语的更一般的形式——信号量与P/V操作来克服忙碌等待，极大地简化了进程的同步与互斥
  - 管程（monitor），把分散在各进程中的临界区集中起来进行管理，用数据结构抽象表示共享资源，便于用高级语言写程序，也便于程序正确性验证
  - 用信号量能够解决的同步问题，同样也可用管程解决

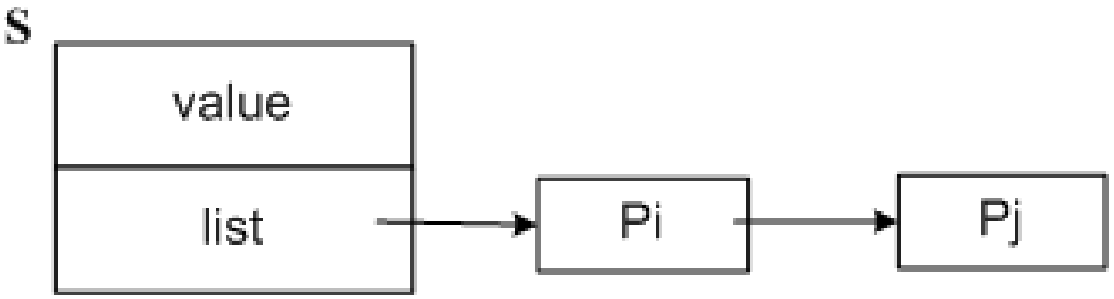
2.4.3进程同步机制

1．信号量机制

- 在这一体制下，进程在某一特殊点上被迫停止执行（阻塞）直到接收到一个对应的特殊变量值，这种特殊变量就是信号量(semaphore)，除了赋初值外，信号量的值只能由P操作和V操作进行修改，进程通过P、V这两个特殊操作在信号量所关联的系统资源上实现同步与互斥。

2.4.3进程同步机制

- 信号量表示系统资源的实体
- 具体实现时，信号量是一种记录型数据结构，有两个分量：一个是信号量的值，另一个是在信号量关联资源上阻塞的进程队列的队头指针



- 信号量在操作系统中的主要作用是封锁临界区、进程同步和维护资源计数。

2.4.3进程同步机制

- P操作和V操作原语的功能：
  - P(s)：将信号量s的值减1，若结果小于0，则调用P(s)的进程被阻塞，并进入信号量s的阻塞队列中；若结果不小于0，则调用P(s)的进程继续运行
  - V(s)：将信号量s的值加1，若结果不大于0，则调用V(s)的进程从该信号量阻塞队列中释放、唤醒一个处于等待状态的进程，将其转换为就绪状态，调用V(s)的进程继续运行；若结果大于0，则调用V(s)的进程继续运行

2.4.3进程同步机制

- 信号量的数据类型以及P、V操作原语的定义：

```
typedef struct semaphore {
    int value;              //信号量值
    struct pcb *list;       //阻塞进程队列指针
};
void P(semaphore &s) {
    s.value - -;
    if (s.value<0)  block(s.list);          //阻塞本进程并进入s信号量队列
}
void V(semaphore &s) {
    s.value++;
    if (s.value<=0)  wakeup(s.list);//唤醒s 队列中的一个进程进入就绪队列
}
```

2.4.3进程同步机制

- 由上述信号量和P、V原语的定义可以得到如下结论：
  - P操作意味进程申请一个资源，求而不得则阻塞进程，V操作意味着释放一个资源，若此时还有进程在等待获取该资源，则被唤醒
  - 若信号量的值为正数，该正数表示可对信号量可进行的P操作的次数，即可用的资源数。信号量的初值一般设为系统中相关资源的总数，对于互斥信号量，初值一般设为1
  - 若信号量的值为负，其绝对值表示有多个进程申请该资源而又不能得到，在阻塞队列等待，即在信号量阻塞队列中等待该资源的进程个数

使用P、V操作的方法

- P、V操作在同一个系统中总是成对出现，不可分离。
- P操作用于进程申请资源，V操作表示进程使用完资源，将资源归还给系统。

2.4.3进程同步机制

- 信号量机制实现进程互斥进入临界区

```
semaphore mutex = 1;
cobegin
process Pi( ) {   // i=1,2,…,N
    P(mutex);
    临界区
    V(mutex);
}
coend
```

2.4.3进程同步机制

- 小河上有一座独木桥，规定每次只允许一人过桥。如果把每个过桥者看作一个进程，为保证安全，请用信号量操作实现正确管理。

```
semaphore s;
s=1;
cobegin
process  pi( ){
      P(s);
      过桥;
      V(s);
}
coend
```

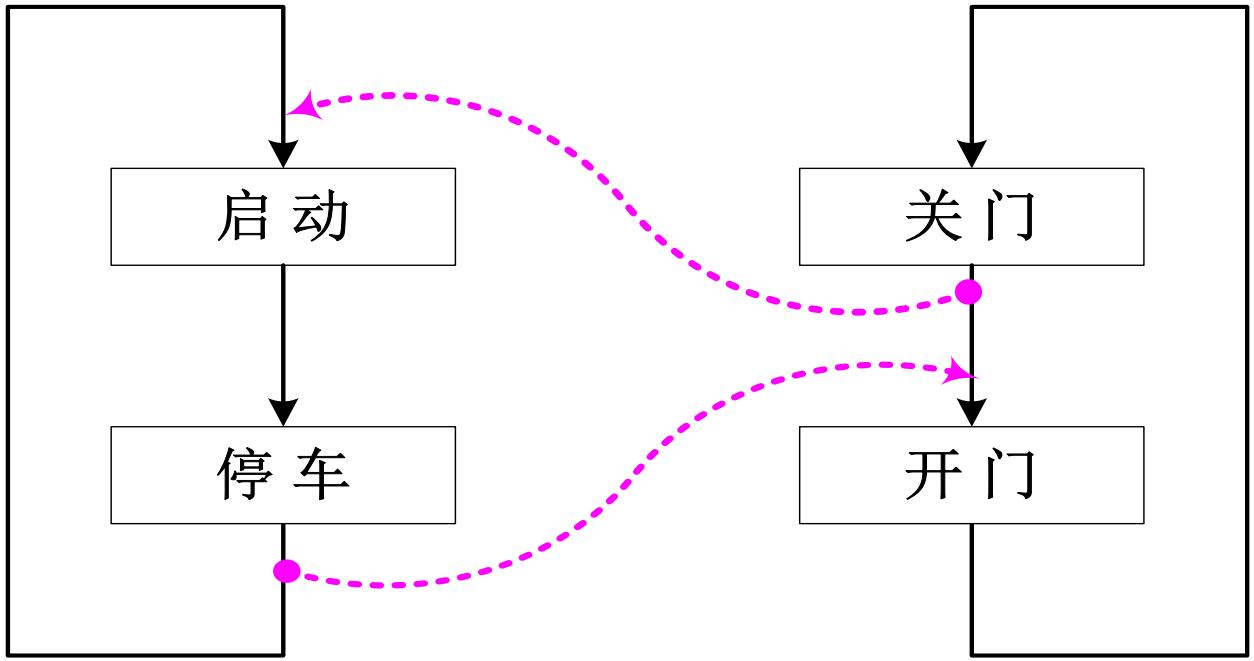
公交司机-售票员

司机：

启动、驾驶、到站停车

售票员：

开关门、卖票



2.4.3进程同步机制

- 设信号量D、S： D.value=0，S.value=0；
- 同步制约算法：

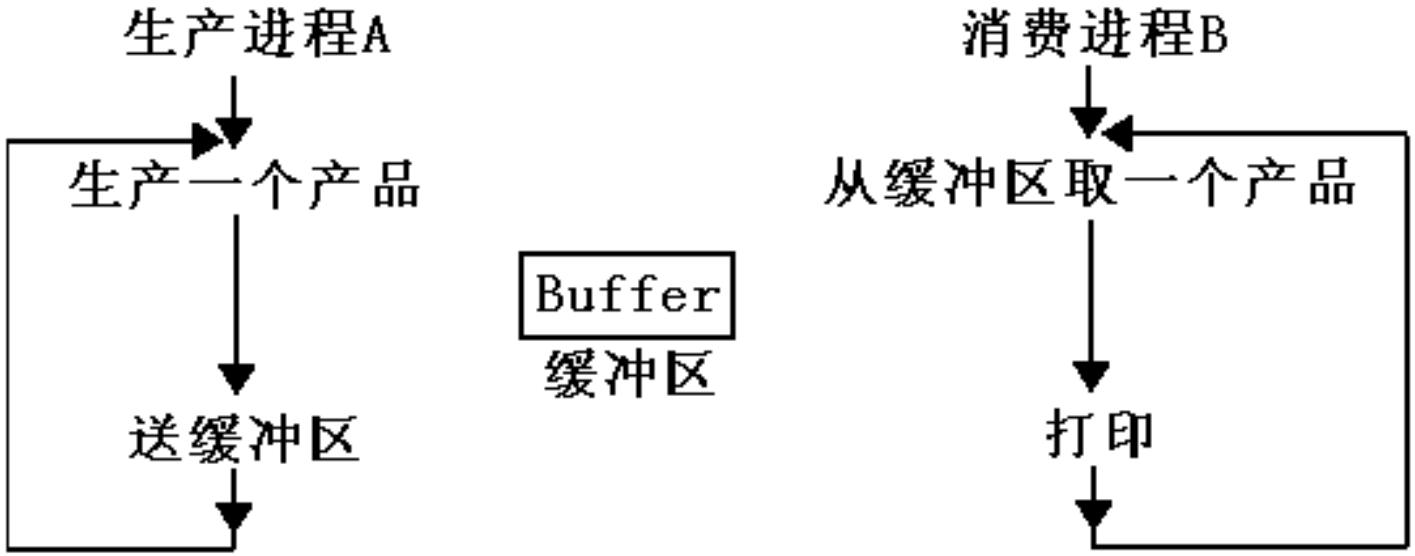
```
cobegin
  process driver {
      while(true){
          P(D);
          启动;
          驾驶;
          到站停车;
          V(S);
      }
  }
coend
```

```
process conductor{
      while(true){
          关门;
          V( D );
          卖票;
          P(S);
          开门;
      }
}
```

2.4.4 进程同步经典问题

- 1．生产者-消费者问题
  - E.W.Dijkstra把广义同步问题抽象成“生产者-消费者问题”（producer-consumer problem）模型
  - 生产者-消费者问题：n个生产者进程和m个消费者进程，连接在一块长度为k个单位的有界缓冲区上（故此问题又称有界缓冲问题）。其中，P*i*和C*j*都是并发进程，只要缓冲区未满，生产者P*i*生产的产品就可送入缓冲区；只要缓冲区不空，消费者进程C*j*就可从缓冲区取走并消耗产品

单缓冲区生产者—消费者问题



例：有打印进程及计算进程，且只有一个缓冲区。计算进程的功能是进行计算并将结果送入缓冲区，打印进程则从缓冲区中将结果取出并交给打印机进行打印。

•计算进程相当于生产进程，打印进程相当于消费进程，缓冲区个数为1。

单缓冲区生产者—消费者问题

- 引入两个信号量，empty, full解决进程间同步问题
- empty:可用的空缓冲区数,指示生产者是否可以向缓冲区放入产品,初值为1
- full:可用的产品数,指示消费者是否可从缓冲区获得产品,初值为0
- 生产者: { P(empty) ; 生产; V(full); }
- 消费者: { P(full); 消费; V(empty); }

```
int B;
semaphore empty;   //可用的空缓冲区数
semaphore full;    //缓冲区内可用的产品数
empty=1;           // 缓冲区内允许放入一件产品
full=0;            //缓冲区内没有产品
Cobegin
process producer(){
    while(true){
        produce();
        P(empty);
        append() to B;
        V(full);
    }
}
process consumer(){
    while(true){
        P(full);
        take() from B;
        V(empty);
        consume();
    }
}
coend;
```

用P、V操作同步与互斥进程的步骤

(1) 分析同步关系：上例中共有两项同步制约条件

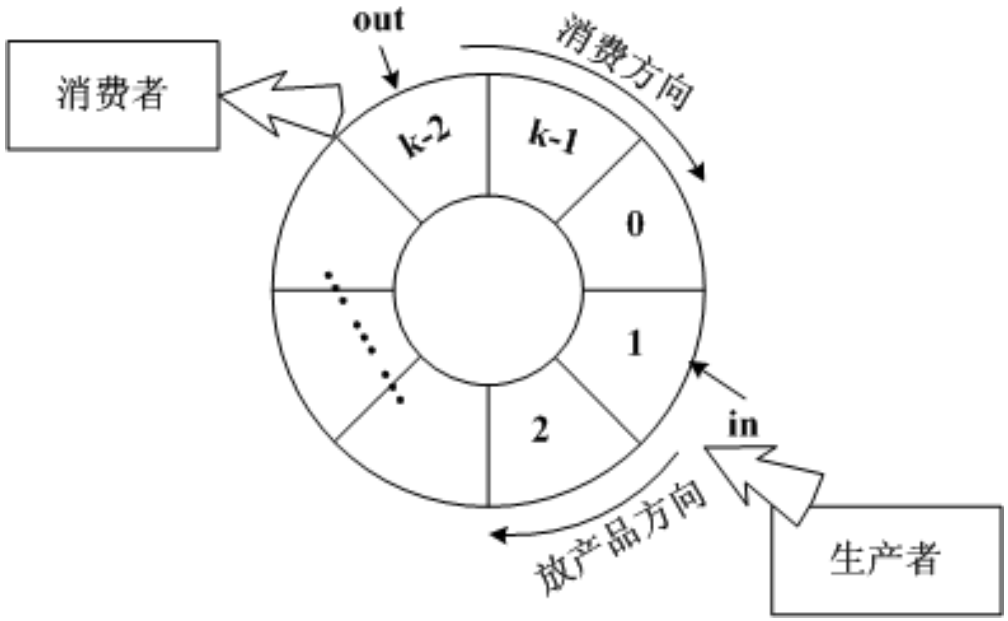
(2) 设置信号量：一般情况下，有几项制约条件就应设置几个信号量

(3) 选择并确定信号量的初值

(4) 利用P、V操作写出进程同步关系

2.4.4 进程同步经典问题

- 多生产者-多消费者-多缓冲区问题示意图



- 在并发环境下生产者、消费者进程访问缓冲区的速度不协调、不匹配——不同步，或者没有做到互不影响地使用、更新缓冲区——互斥，所以会出现运行错误甚至是死锁

同步制约关系：

(1) 一次只能有一个进程进入Buffer区活动(输入或输出产品)；

(2) 缓冲区产品放满后生产者不能再向缓冲区中放入产品；

(3) 消费者不能从空缓冲区中取产品。

2.4.4 进程同步经典问题

信号量设置及初值：

(1) 根据制约关系(1)设置互斥信号量mutex，表示进程互斥进入缓冲区，初值为1，即mutex=1

(2)根据制约关系(2)设置同步信号量empty表示生产者目前可用的缓冲区数，初值为k，即empty=k

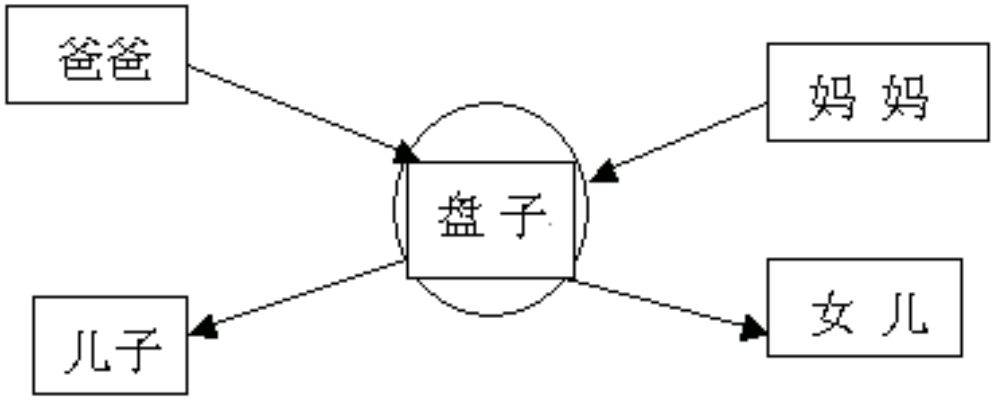
(3)根据制约关系(3)设置同步信号量full表示消费者可消费的产品数(缓冲区数)，初值为0，即full=0

```
item B[k];                  //缓冲区，长度k
semaphore empty = k;        //可用的空缓冲区数
semaphore full = 0;         //缓冲区内可用的产品数
semaphore mutex = 1;        //互斥信号量
int in=0;                   //缓冲区放入位置
int out=0;                  //缓冲区取出位置
Cobegin

process producer_i ( ) {
   while(true) {
      produce( ); //生产一个产品
      P(mutex);   //申请互斥使用缓冲区
      P(empty);   //申请空缓冲区
      append to B[in]; //产品放入缓冲
      in=(in+1)%k;          //更新缓冲区指
   针
      V(mutex);
      V(full);
   }
}

process consumer_j ( ) {
    while(true) {
        P(mutex);
        P(full);
        take( ) from B[out];
        out=(out+1)%k;
        V(mutex);
        V(empty);
        consume( );
    }
}

Coend
```

- 注意：进程同步中P操作的顺序是十分重要的。如果P操作位置不当，将会产生许多错误。
- P(empty)和P(mutex)调用的先后顺序不能颠倒，否则在缓冲区全部为满时会引起生产者进程已进入缓冲区，却不能放入产品的问题。而此时消费者进程也不能进入缓冲区取走产品消费。生产者进程与消费者进程发生死锁。
- 同样P(full)和P(mutex)的先后顺序不能弄错，否则当缓冲区全部为空时会引起消费者进程进入缓冲区后不能消费，生产进程也不能进入缓冲区放入产品。生产者进程与消费者进程同样进入死锁。
- 通常将用于互斥的信号量的P操作放置在用于同步的私有信号量的P操作之后，便可避免这些错误的发生，对于V操作，则没有顺序的问题。

2.4.4 进程同步经典问题

例：桌上有一只盘子，每次只能放入/取出一只水果。爸爸专向盘子中放苹果（apple），妈妈专向盘子中放桔子（orange），一个儿子专等吃盘子中的桔子，一个女儿专等吃盘子中的苹果。请写出爸爸、妈妈、儿子和女儿正确同步工作的程序。



2.4.4 进程同步经典问题

这个问题实际上可看作是两个生产者和两个消费者共享了一个仅能存放一件产品的缓冲区，生产者各自生产不同的产品，消费者各自取走自己需要的产品。

由于盘子中每次只能存放一个水果，因此爸爸和妈妈在存放水果时必须互斥。儿子和女儿分别要吃桔子和苹果，因而，当爸爸向盘子中放入一只苹果后应把“盘中有苹果”的消息发送给女儿；同样，当妈妈向盘子中放入一只桔子后应把“盘中有桔子”的消息发送给儿子。如果儿子或女儿取走盘子中的水果，则应发送“盘子中又可存放水果”的消息。但这个消息不应特定地发送给爸爸或妈妈，至于谁能再向盘中放水果则要通过竞争资源(盘子)的使用权来决定。

2.4.4 进程同步经典问题

- 定义一个是否允许向盘子中存放水果的信号量S，其初值为“1”，表示允许存放一只水果；
- 定义两个信号量SP和SO分别表示盘子中是否有苹果或桔子的消息，初值应该均为“0”；
- 儿子或女儿取走水果后要发送“盘中又可存放水果”的消息，只要调用V(S)就可达到目的。

```
semaphore   S, SP, SO;
    S:=1;   SP:=0;   SO:=0;
```

```
Cobegin
   Process 爸爸
   {
     while (true) {
        准备一个苹果；
        P ( S ) ;
        把苹果放入盘子中；
        V ( SP ) ;
        }
     }
   Process 妈妈
   {
     while (true){
       准备一个桔子；
        P ( S ) ;
        把桔子放入盘子中；
        V ( SO) ;
        }
    }
```

```
Process 儿子
      {
      while (true) {
         P ( SO ) ;
         从盘子中取一只桔子；
         V ( S ) ;
         吃桔子；
         }
      }
Process 女儿
{
    while (true) {
        P ( SP ) ;
         从盘子中取一只苹果;
         V ( S );
         吃苹果；
         }
      }
Coend;
```

2.4.4 进程同步经典问题

- 2．读者-写者问题

——两组并发进程，读者和写者，共享一个文件*F*，要求：

- 允许多个读者进程同时读文件
- 只允许一个写者进程写文件
- 任何一个写者进程在完成写操作之前不允许其它读者或写者工作
- 写者执行写操作前，应让已有的写者和读者全部退出

```
int readcount=0;  //读进程计数器
semaphore ws = 1, mutex = 1;
cobegin
process reader_i( ) {
  P(mutex);
  readcount++;
  if(readcount==1) P(ws);
  V(mutex);
  读文件;
  P(mutex);
  readcount--;
  if(readcount==0) V(ws);
  V(mutex);
}
 coend
```

```
process writer_j( )
 {
   P(ws);
   写文件;
   V(ws);
 }
```

```
int readcount=0, writecount=0;
semaphore mrc=l, mwc= 1, wr=1, wsem=1, rsem=l;
cobegin
```

```
void reader_i( ){
P(wr);
P(rsem);
P(mrc);
readcount++;
if (readcount ==1) P(wsem);
V(mrc);
V(rsem);
V(wr);
READ;
P(mrc);
readcount--;
if (readcount ==0) V(wsem);
V(mrc);
}
```

```
void writer_j( ){
P(mwc);
writecount++;
if (writecount ==1) P(rsem);
V(mwc);
P(wsem);
WRITE;
V(wsem);.
P(mwc):
writecount--;
if(writecount==0) V(rsem);
V(mwc);
}
```

coend

读者-写者问题——写者优先算法-2

```
semaphore  rmutex,wmutex, S; // S在写者到达后封锁读者
rmutex = 1;  wmutex = 1;  S= 1;
int  count = 0;
cobegin
```

```
process reader {
  while(true){
     P(S);
     P( rmutex)
     if (count==0) P( wmutex);
     count++;
     V( rmutex);
     V(S);
     READING the FILE;
     P( rmutex);
     count--;
     if (count==0) V( wmutex);
     V( rmutex);
  }
}
```

```
process writer{
    while(true){
      P( S );
      P( wmutex);
      WRITING the FILE;
      V(wmutex);
      V(S);
    }
}
```

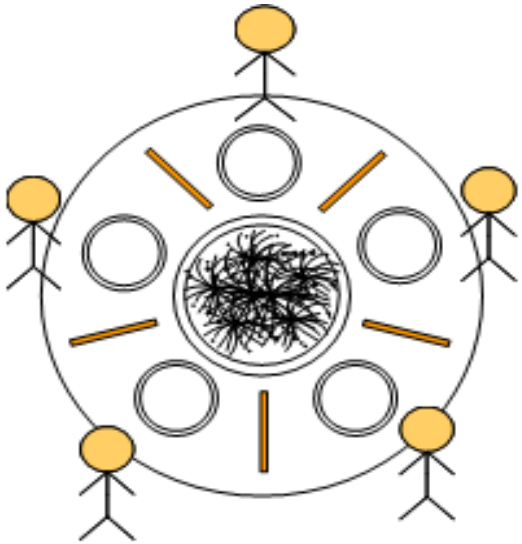
```
coend
```

- 例：假定一个阅览室有100个坐位，读者进入阅览室必须有空闲坐位,进入和离开阅览室时都在阅览室门口的一个登记表上进行登记和去掉登记，而且每次只允许一人登记或去掉登记，请用P、V操作写出读者同步制约关系。设读者最多有200个，计算信号量的最大和最小值。

2.4.4 进程同步经典问题

3．哲学家就餐问题

- 五个哲学家围坐在一圆桌旁，桌中央有一盘通心面，每人面前有一只空盘子，每两人之间放一只筷子。每个哲学家的行为是思考，感到饥饿，然后吃通心面。为了吃面，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手边和右手边去取筷子。



2.4.4 进程同步经典问题

- 哲学家就餐问题最简单的解法：

```
void philmac (int i) {
    思考;
    取chopsticks [i];
    取chopsticks [(i+1) % 5];
    吃面;
    放chopsticks [i];
    放chopsticks [(i+1) % 5];
}
```

  - 可能发生死锁

- 信号量解决哲学家吃通心面问题的算法：
- 算法1：给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之。

```
semaphore chopsticks [5];
for (int i=0; i<5; i++) chopsticks [i] = 1;
cobegin
process philmac_i( ) {  //i=0,1,2,3,4
    think( );
    if(i%2 ==0) {
      P(chopsticks [i]);
      P(chopsticks [(i+1)%5] );
    }
    else{
      P(chopsticks [(i+l)% 5]);
      P(chopsticks [i]);
    }
    eat( );
    V(chopsticks [i]);
    V(chopsticks ([i+ 1] % 5);
}
coend
```

- 算法2：通过发放令牌最多允许4个哲学家同时吃面

```
semaphore chopsticks[5] = {1,1,1,1,1};
semaphore token = 4;                //4个令牌
int i;
process philmac_i( ) {    //i=0,1,2,3,4
    think( );
    P(token);
    P(chopsticks[i]);
    P(chopsticks [(i+l) %5]);
    eat( );
    V(chopsticks [(i+l) % 5]);
    V(chopsticks[i]);
    V(token);
}
```

```
semaphore  S=1, SO=0, SS=0, SW=0; //容器是否可用, 容器中是浓缩汁/糖/水
enum { sugar, water, orange } container;
```

```
cobegin
process Provider {
  while(true){
    P(S);
    将原料装入容器内;
    if (cantainer==orange) V(SO);
   else if (cantainer==sugar) V(SS);
    else V(SW);
  }
}
```

```
process P2 {
  while(true){
            P(SS);
            从容器中取糖;
            V(S);
            生产橙汁;
  }
}
```

```
process P1 {
  while(true){
            P(SO);
            从容器中取浓缩汁;
            V(S);
            生产橙汁;
  }
}
coend
```

```
process P3 {
  while(true){
            P(SW);
            从容器中取水;
            V(S);
            生产橙汁;
  }
}
```

例：假定一个阅览室有100个坐位，读者进入阅览室必须有空闲坐位,进入和离开阅览室时都在阅览室门口的一个登记表上进行登记和去掉登记，而且每次只允许一人登记或去掉登记，请用P、V操作写出读者同步制约关系。设读者最多有200个，计算信号量的最大和最小值。

解：设置坐位信号量S，S=100，表示初始有100个空座位;

登记信号量MUTEX，MUTEX=1，登记簿为临界资源;

```
While（true）
{  P(S)
   P(MUTEX)
     登记
   V(MUTEX)
     进入阅览室阅读
   P(MUTEX)
     去掉登记
   V(MUTEX)
   V(S)  }
```

S最大可能值为：100（最多有100个空座位）；
S最小可能值为：-100（最多有100个人在等座位）
MUTEX最大可能值为：1；
MUTEX最小可能值为：-99（最多有99个人在等待使用登记簿）

2.4.4 进程同步经典问题

- **管程解决哲学家就餐问题的算法：**
- **采用Hoare管程实现，算法思想是将哲学家的状态分为思考、饥饿、吃面，并且仅当哲学家左右两边的筷子都可用才允许他拿筷子，否则一只筷子也不拿。**

2.4.4 进程同步经典问题

- **4．睡眠理发师问题（sleepy barber problem）**
  - **理发店里有一个理发师，一把理发椅，N个供等候顾客休息的椅子。若无顾客，理发师躺在理发椅上睡觉。顾客到来时唤醒理发师，若理发师正在理发，新来的顾客坐在空闲的休息椅上等候，如果没有空椅子，顾客离开。**
  - **进程Barber ()和Customer()分别描述理发师和顾客，理发师和顾客之间是同步的关系，顾客之间是互斥关系，竞争理发师和休息椅。**
  - **算法采用信号量机制，引入一个控制变量和3个信号量：**
    - **控制变量waiting用来记录等候理发的顾客数，初值为0；**
    - **信号量customers用来关联表达等候理发的顾客数，并用作阻塞理发师进程，初值为0；**
    - **信号量barbers用来记录正在等候顾客的理发师数，并用作阻塞顾客进程，初值为0；**
    - **信号量mutex用于互斥，初值为1。**

2.4.4 进程同步经典问题

- **进程同步(互斥)类题目**
  - **解读分析题设，找出进程、资源等实体**
  - **分析设计进程的流程(单任务/封闭环境)**
  - **围绕资源的使用分析进程间协作同步或互斥制约的关系**
  - **依据上述分析结果设计信号量**
  - **将信号量及其PV操作融入进程的流程，在开放的并发环境中检验进程间并发运行时的合理性和正确性**

2.5 进程通信

- 进程同步与互斥，实现了进程间的信息交换。但由于交换的信息量少，可以看作是低级通信
- 交换的数据量较大时，为了提高效率，系统需要采用一些相应的通信机制来完成进程间通信
- 进程通信（IPC：InterProcess Communication）是进程之间数据的相互交换和信息的相互传递，是一种高级通信机制
- 主要有消息传递（message passing）通信、共享内存（shared memory）通信和管道（pipe）通信

2.5.1 消息传递通信

- 进程将通信数据封装在消息中，消息通过消息缓冲区在进程之间互相传递
- 消息是指进程之间以不连续的成组方式发送的信息
- 消息缓冲区应包含消息发送进程标识、消息接收进程标识、指向下一个消息缓冲区的指针、消息长度、消息正文等。

2.5.1 消息传递通信

- Windows系统中，消息结构的定义

```
typedef struct tagMSG {
    HWND hWnd;          // 指定消息发向的目标窗口句柄
    UINT message;               // 消息标识
    WPARAM wParam;      // 消息参数，32bit
    LPARAM lParam;      // 消息参数，32bit，含义和值依
                           不同消息而不同
    DWORD time;         // 消息进入队列的时间
    POINT pt;           // 消息进入队列时鼠标指针的屏
    幕坐标
}MSG;
```

2.5.1 消息传递通信

- 每个进程都有一个消息队列，其队列头由接收进程的PCB中的消息队列指针指向
- 当来自其它一些进程的消息传递给它时，就将这些消息链入消息队列。通常，进程按先来先服务的原则处理这一队列
- 消息通信中存在直接通信与间接通信两种类型

2.5.1 消息传递通信

- 在直接通信方式下，发送进程将发送的数据封装到消息正文后，发送进程必须给出接收进程的标识，然后用发送原语将消息发送给接收进程
- 收发消息的原语：
  - send(接收进程标识，消息队列首指针)
  - receive(发送进程标识，接收区首地址指针)
  - 在直接通信中隐含着发送进程与接收进程之间的同步问题

2.5.1 消息传递通信

- send( ):
  - 查找接收进程的PCB，存在，则申请一个存放消息的缓冲区，若消息缓冲区已满，则返回到非同步错误处理程序入口，进行特殊处理
  - 若接收进程因等待此消息的到来而处于阻塞状态，则唤醒此进程
  - 将存放消息的缓冲区连接到接收进程的消息队列上

2.5.1 消息传递通信

- receive ( ) :
  - 接收进程在其进程的存储空间中设置一个接收区，复制/读取消息缓冲区中的内容，释放消息缓冲区
  - 若无消息可读，则阻塞接收进程至有消息发送来为止

2.5.1 消息传递通信

- 两种同步方式
  - 发送进程阻塞等待接收进程发回的确认信息
  - 发送进程发送完消息后，不阻塞等待接收进程的回送信息，而是继续执行；限定时间到仍未收到确认消息，重发或放弃

2.5.1 消息传递通信

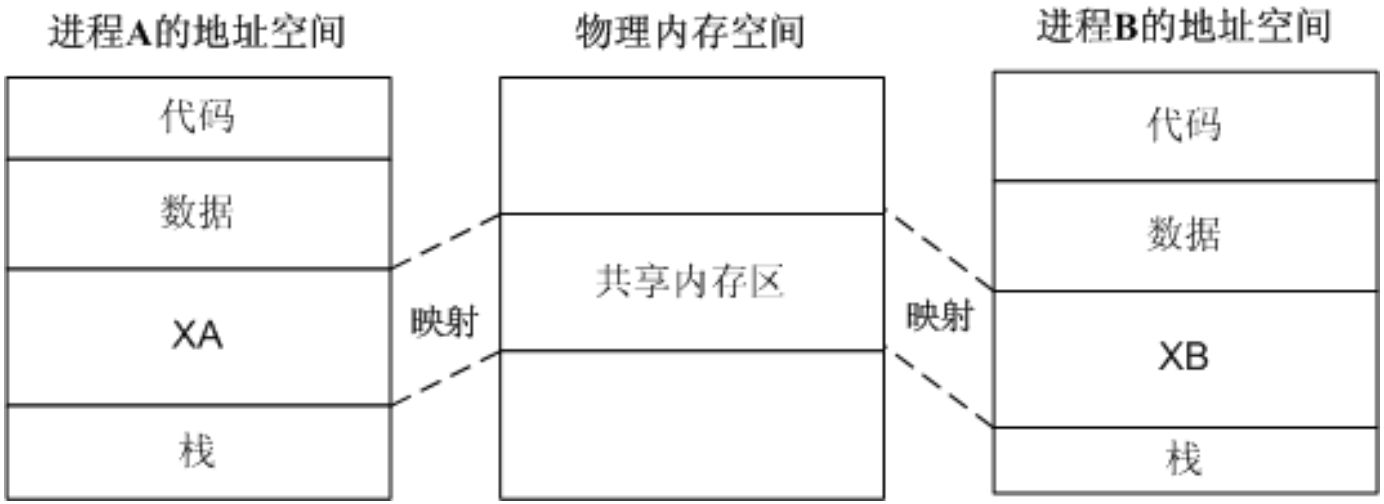
- 两种同步方式
  - 接收进程调用receive原语并一直阻塞等待发送来的消息，直到接收到消息——与发送进程的第二种同步方式匹配
  - 接收进程调用receive原语后，不阻塞等待发送来的消息，而是继续执行——与发送进程的第一种同步方式匹配

2.5.1 消息传递通信

- 消息传递的间接通信方式是指发送进程与接收进程之间通过邮箱来进行通信，发送进程将消息发送到邮箱，接收进程从邮箱接收消息
  - 发送原语：send(mailboxname, message);
  - 接收原语：receive(mailboxname, message);
  - 与直接通信比较，间接通信灵活性更大，不需要发送进程与接收进程同步，是一种方便、可靠的进程通信方式

2.5.2 共享内存通信

- 共享内存通信可进一步分为
  - 基于共享数据结构的通信方式——比较低效，只适于传递少量数据
  - 基于共享存储区的通信方式



2.5.2共享内存通信

- 共享内存通信的实现过程如下：
  - 1）建立共享内存区——标识和长度等参数
  - 2）共享内存区的管理
  - 3）共享内存区的映射与断开
  - 允许多个进程将共享内存映射到自己的地址空间，进程对各自所映射的地址段的读写操作代码应纳入临界区管理

2.5.3管道通信

- 管道是连接读、写进程的一个特殊文件，允许进程按FIFO方式传送数据，也能使进程同步执行操作。发送进程以字符流形式把数据送入管道，接收进程从管道中接收数据
- 管道的实质是一个共享文件（文件系统的高速缓冲区中），进程对管道应该互斥使用
- 写进程把一定数量的数据写入pipe，就去睡眠等待，直到读进程取走数据后，将其唤醒

2.5.3管道通信

- 命名管道（named pipe）用来在不同的地址空间之间进行通信，不仅可以在本机上实现两个进程间的通信，还可以跨网络实现两个进程间的通信，特别为服务器通过网络与客户端交互而设计，是一种永久通信机制
- 每一个命名管道都有一个唯一的名字
- Windows系统中，管道服务器在调用*CreateNamedPipe( )*函数创建命名管道的一个或多个实例时为其指定了名称。对于管道客户，则是在调用*CreateFile( )*或*CallNamedPipe( )*函数以连接一个命名管道实例时对管道名进行指定。命名管道提供了两种基本的通信模式：字节模式和消息模式