2

进 程 管 理

目录

2.1 概述

- 所谓进程（process），就是正在运行的程序及其占用的系统资源，如CPU（寄存器），内存，I/O设备等资源
- 同一程序同时被两次运行，就是两个独立的进程——虽代码相同，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
- 进程管理是操作系统的核心功能之一

2.2 程序的执行方式

中国矿业大学

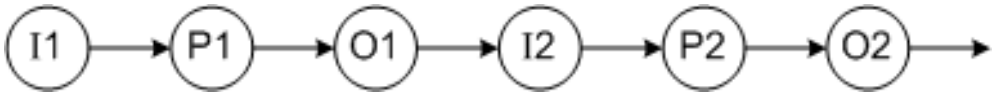
- 程序的**执行**是指将二进制代码文件（如*.exe等）装入内存，由CPU按程序逻辑运行指令的过程，包括两种执行方式：
  - ✓顺序
  - ✓并发
- **单道程序设计技术**：内存一次只允许装载一个程序运行，在这次程序运行结束前，其它程序不允许使用内存
- **多道程序设计技术**：允许多个程序进驻内存，系统通过某种调度策略交替执行程序

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**

  ✓**程序的顺序执行**意味着程序间的执行序列是顺序的，一个程序执行完，才能执行另一个程序

  ✓**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



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：程序执行的最终结果由给定的初始条件决定，不受外界因素的影响
  - ③ 可再现性：只要输入的初始条件相同，无论何时重复执行该程序都会得到相同的结果
- 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

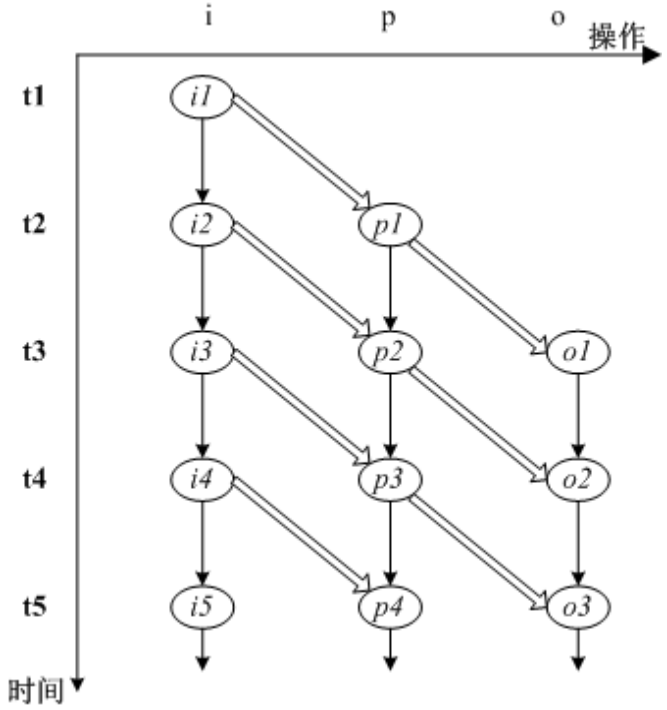
- 为提高系统的效率，允许“同时”执行多个程序，程序的执行不再是顺序的，一个程序未执行完另一个程序便开始执行，内存中同时载入多个相对独立的程序代码，它们争用CPU、内存、外设等软硬件资源，这就引出了并发的概念。
  - ✓并发——多个事件在同一时期内发生
  - ✓并行——多个事件在同一时刻发生

2.2.2 程序的并发执行与并行执行

- 并发指在一段时间内系统中宏观上有多个程序在同时运行，但在单CPU系统中，每一时刻却仅能有一道程序执行，微观上这些程序是分时地交替执行
- 若计算机系统中有多个CPU，则这些并发执行的程序便被分配到多个CPU上，多个程序便可以真正地同时执行，实现并行
- 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

2.2.2 程序的并发执行与并行执行

现代计算机的硬件:CPU、内存、外设等是能够同时进行工作的，因此，如果进行适当的程序设计，在计算机系统中也可以充分发挥不同硬件设备间并行工作的能力,实现程序的并发执行。



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：系统中载入了多个程序，各程序的运行流程可能是“运行→暂停→继续→…”这样的模式

  2、开放/交互性：由于系统中载入多个程序，存在资源共享/争用的情况，因此多个程序运行时可能会相互交互、互相影响

2.2.2 程序的并发执行与并行执行

- 并发执行的特性：

  3、不可再现性：上述原因会造成程序在不同的情况下运行可能会出现不同的结果，甚至会造成错误

2.2.3 进程概念的引入

- 相关术语
  - ✓程序（program）——特指代码文件，强调其静态性，其代码可以是二进制机器指令，也可以是高级语言
  - ✓进程（process）——进程目前还没有统一的定义，约定俗成的说法是可并发执行的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

2.2.3 进程概念的引入

- 相关术语
  - ✓作业（job）——批处理系统要装入系统运行处理的一系列程序和数据，一般由相应的作业控制语言来描述作业步骤、参数等执行细节

2.3 进程的特征与控制

中国矿业大学

进程的特征：

1. 结构性：进程包含程序及其相关数据结构。进程的实体包含进程控制块（PCB）、程序块、数据块和堆栈
2. 动态性：进程是程序在数据集合上的一次执行过程，具有生命周期，由创建而产生，由调度而运行，由结束而消亡，是一个动态推进，不断变化的过程

2.3 进程的特征与控制

中国矿业大学

进程的特征

3. 独立性：进程是操作系统资源分配、保护和调度的基本单位，每个进程都有其自己的运行数据集，以各自独立的、不可预知的进度异步运行
4. 并发性：进程在单CPU系统中并发执行，在多CPU系统中并行执行，并发性能够提高资源利用率，但并发执行意味着进程的执行可以被打断，可能会带来意想不到的问题，因此必须对并发执行的进程进行协调

2.3 进程的特征与控制

进程分为系统进程和用户进程，区别如下：

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由用户负责，操作系统提供一套任务调用命令作为协调手段

2.3 进程的特征与控制

进程分为系统进程和用户进程，区别如下：

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，由系统调度和分配

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

中国矿业大学

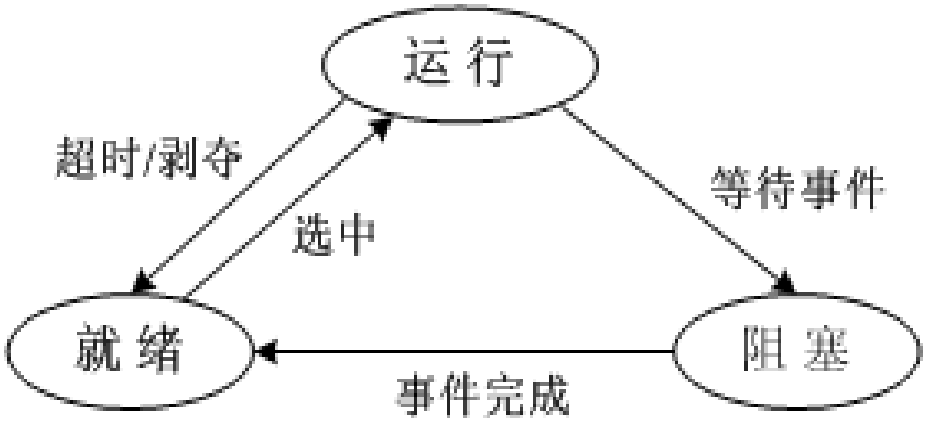
进程上下文：进程的生命周期中，进程实体和支持进程运行的环境，包括：

1. CPU中的多个程序寄存器、各类控制寄存器、地址寄存器等
2. 进程控制块PCB、内存管理信息、系统栈
3. 进程的代码区、数据区、用户栈区和共享存储区

- 一个进程被系统调度而占有CPU运行时，会发生CPU在新老进程之间切换，切换的内容就是进程上下文

2.3.1 进程状态及转换

- 进程在生命周期中会处于不同的状态，进程的三种基本状态为：**就绪状态、运行状态、阻塞状态**。



2.3.1 进程状态及转换

进程的基本状态

- **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，一旦被分配CPU，进程立刻执行。一个进程在创建后处于就绪状态。
  - 如果一个计算机系统中有多个进程都处于就绪状态，这些处于就绪状态的进程会被以队列方式组织在一起，称为就绪队列

2.3.1 进程状态及转换

中国矿业大学

进程的基本状态

- 运行状态（running）——进程占用CPU并正在执行，在单CPU系统中，任一个时刻只有一个进程处于运行状态。

2.3.1 进程状态及转换

进程的基本状态

- 阻塞状态（blocked）——也称为等待状态，当正在运行的进程由于发生某事件，如：请求并等待输入/输出过程的完成、等待进程通信之间的通信信号到来或进程同步之间的同步信号的到来等，而受到阻塞不能继续执行时，就需要放弃CPU，从运行状态转换到阻塞状态
  - 一个系统中有多个进程都处于阻塞状态，这些进程被组织成队列形式，称为阻塞队列。

2.3.1 进程状态及转换

中国矿业大学

进程状态之间的转换有如下几种情况：

- 就绪状态→运行状态：当CPU空闲时，操作系统从就绪队列中选中一个就绪进程并分配CPU，此时，该进程的状态便从就绪状态转换到运行状态
- 运行状态→阻塞状态：当正在运行的进程需要等待某些事件（如I/O）的发生时，其状态将从运行状态转换为阻塞状态

2.3.1 进程状态及转换

中国矿业大学

进程状态之间的转换有如下几种情况：

- 阻塞状态→就绪状态：处于阻塞状态的进程，由于等待的事件到来而不需要再等待时，进程状态便从阻塞状态转换到就绪状态。
  - 需要注意的是，即便阻塞的进程所等待的事件完成，运行条件满足，该进程也不会立即获得CPU运行，仍需按部就班地先转换为就绪状态，进入就绪队列，再等待被操作系统调度。

2.3.1 进程状态及转换

中国矿业大学

进程状态之间的转换有如下几种情况：

- 运行状态→就绪状态：正在运行的进程被操作系统中断执行（如分配的时间片用完，或优先级更高的进程进入就绪队列），该进程状态将从运行状态转换到就绪状态，等待被再次调度。

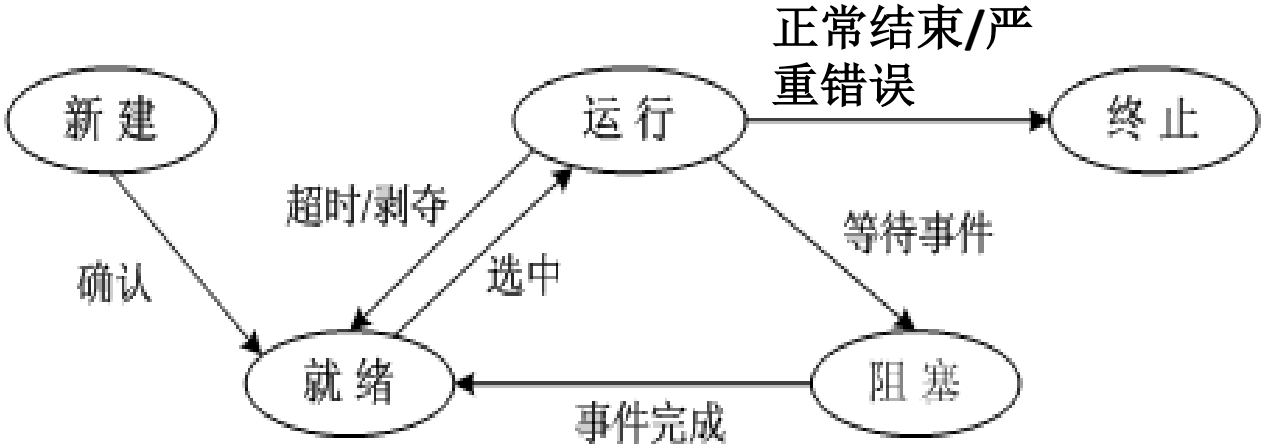
2.3.1 进程状态及转换

进程五态模型

- 新建状态（new）——进程被创建时所处的状态。进程新建成功后即转入就绪状态，进入就绪队列
- 终止状态（terminated）——进程正常结束或出现了严重错误时，会被操作系统终止或被其它有终止权的进程终止，进入终止状态的进程不再被执行，等待操作系统完成进程终止处理。

2.3.1 进程状态及转换

- **进程五态模型**



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

Q：什么时候需要挂起进程？

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

- 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

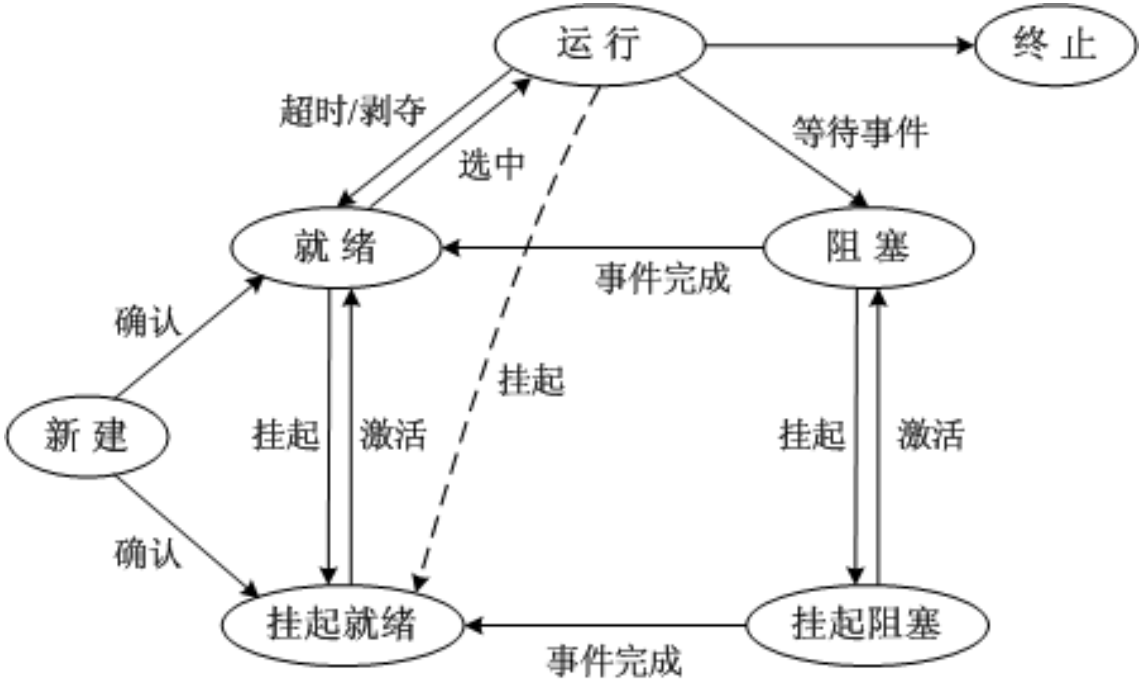


图2.5 具有挂起功能的进程状态及转换

2.3.1 进程状态及转换

有挂起功能的进程状态

- ✓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）：进程在等待某一事件或条件，且该进程目前不在内存中

2.3.1 进程状态及转换

中国矿业大学

有挂起功能的进程状态转换

- 进程在运行态也可以被挂起，转换为挂起就绪状态，原因是具有更高优先级的进程要抢占CPU，而此时内存空间不够
- 如果内存紧张，则就绪状态的进程可以被挂起，进程状态转换为挂起就绪状态；当内存空间富裕时，处于挂起就绪状态的进程可以被恢复/激活，从挂起就绪状态转换为就绪状态

2.3.1 进程状态及转换

有挂起功能的进程状态转换

- 如果内存紧张，阻塞状态的进程可以被挂起，进程状态就转换为挂起阻塞状态，如果内存空间富裕，则挂起阻塞状态的进程被恢复/激活，进程状态会转换为阻塞态
- 如果挂起阻塞状态的进程的阻塞事件或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

2.3.1 进程状态及转换

中国矿业大学

有挂起功能的进程状态转换

- 如果创建进程时，系统内存空间足够，则新建进程进入就绪状态；若没有足够的内存空间，则转入到挂起就绪状态
- 所有被挂起的进程只有被恢复/激活后才能由外存调入内存，只有处于就绪状态的进程才有可能被调度分配CPU运行

2.3.2 进程控制块--PCB

- 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（Process Control Block，它是进程实体的一部分，是操作系统中最重要的数据结构之一，PCB中记录了描述进程的当前状态以及控制进程运行的信息，主要包括：
  1）进程标识信息
  2）现场信息
  3）控制信息

2.3.2 进程控制块--PCB

1. 进程标识信息

- 内部标识符是操作系统为进程设置的一个唯一整数，操作系统管理进程时使用进程的内部标识符
- 外部标识符由字母和数字组成，是进程创建者提供的进程名，用户访问进程时使用外部标识符
- 进程创建时，用户给出进程的外部标识，操作系统给出进程的内部标识

2.3.2 进程控制块--PCB

2. 现场信息

指进程运行时CPU的即时状态，即各寄存器中的数值，包括各通用寄存器（EAX，EBX，ECX，……），控制寄存器（MSW/CR0、CR2和CR3），栈指针等

2.3.2 进程控制块--PCB

3. 控制信息

指操作系统控制进程需要的信息，包括：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、进程的等待时间、进程的执行时间、与进程状态变化相关的事件等内容

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：

- 操作系统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
- 在调度运行该进程后，要根据其PCB中所保存的CPU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，找到指令和数据

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：

- 进程在执行过程中，当需要其它相关进程实现同步、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，必须将其断点的CPU上下文保存在PCB中

在进程的整个生命期中，操作系统总是通过PCB控制进程，所以**PCB是进程存在的惟一标志**

2.3.2 进程控制块--PCB

- PCB经常被系统访问，故PCB常驻内存
- 将所有进程的PCB集中在系统的特定位置，构成PCB表
- 再进一步将处于相同状态的进程的PCB组织进一个队列，形成运行队列、就绪队列、阻塞队列
- 既可将所有阻塞进程放入一个阻塞队列，也可根据引起阻塞的不同原因构成多个阻塞队列
- 单CPU系统中，运行队列长度为1

2.3.2 进程控制块--PCB

进程队列的组织方式通常采用：

- 线性方式
- 链接方式
- 索引方式

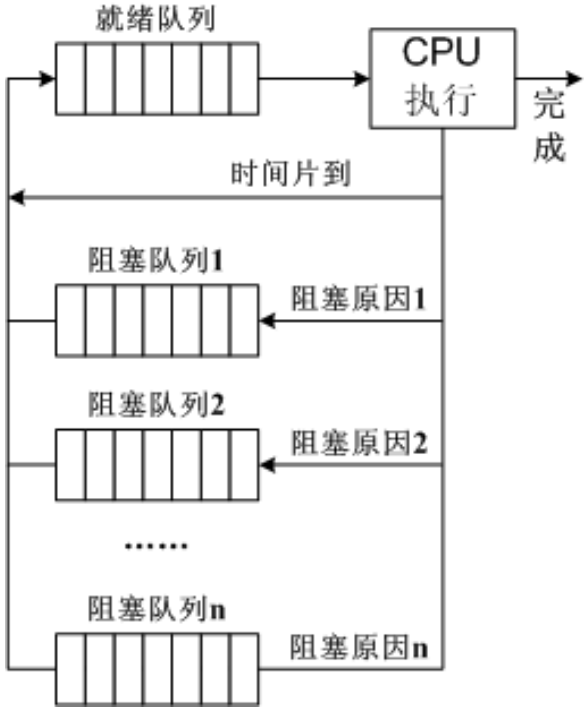


图2.6 多进程队列组织方式

2.3.2 进程控制块--PCB

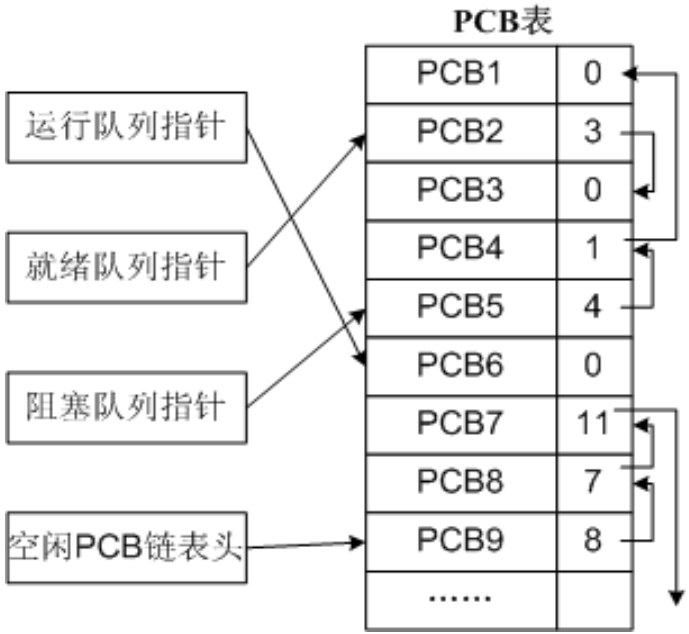


图2.7 PCB链接方式

2.3.2 进程控制块--PCB

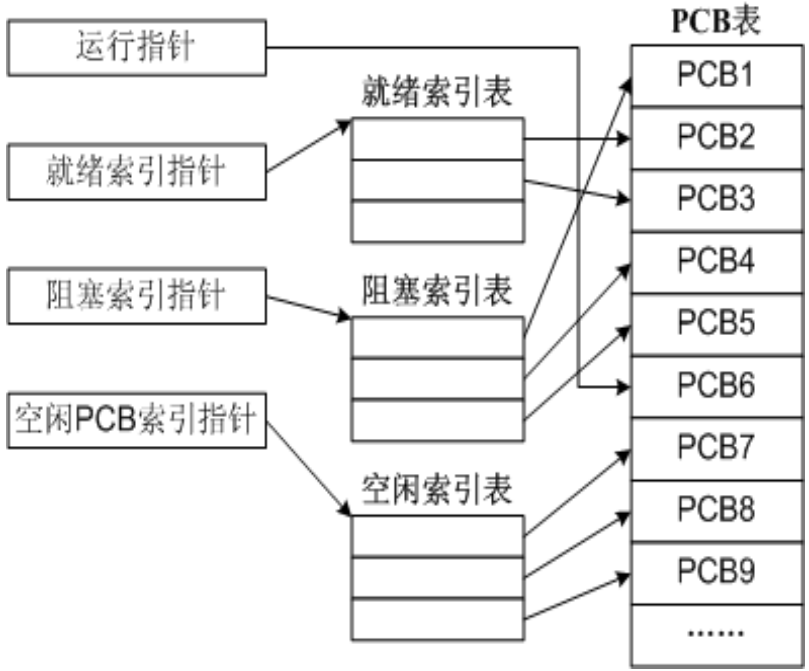


图2.8 PCB索引方式

2.3.3 进程控制

CPU的两种运行模式

- **核心态**（内核态）：操作系统内核代码、设备驱动等需要获得无限制的指令执行特权，访问内存空间任何有效地址，直接进行设备端口操作，所以需要运行在核心态
- **用户态**：一般用户进程代码要受到CPU保护模式的诸多安全限制，运行于用户态，只能执行部分普通指令，只能访问自己的虚拟地址空间，只能访问系统许可的端口

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

- 进程控制通过--原语
- 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
  - ✓原语是不可分割的执行单位，不能并发执行
  - ✓原语是操作系统核心的组成部分，且常驻内存
  - ✓通常在核心态下执行

2.3.3 进程控制

1. 创建进程：

进程的存在以PCB为标志，所以创建进程的主要任务就是建立PCB、填入相关信息、把该PCB插入就绪队列

✓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
- 应用程序请求创建——由用户调用操作系统提供的系统调用完成

Windows的API函数：CreateProcess( )

Linux的系统调用：fork( )

2.3.3 进程控制

1. 创建进程

✓创建进程主要步骤(8)：

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

2.3.3 进程控制

1. 创建进程

✓创建进程主要步骤(8)：

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. 通知操作系统的日志、性能监控程序等

2.3.3 进程控制

2. 撤销与终止进程

- 进程完成之后，应退出系统而消亡，系统及时收回它占有的全部资源以便其它进程使用
- 撤销原语撤销的是标志进程存在的进程控制块PCB，而不是进程的程序段。这是因为一个程序段可能是几个进程的一部分，即可能有多个进程共享该程序段

2.3.3 进程控制

2. 撤销与终止进程

撤销原语实现过程：

1. 根据提供的要被撤销进程的标识，在PCB链中查找对应的PCB，若找不到或该进程尚未停止，则转入异常终止处理程序
2. 否则，从PCB链中撤销该进程及其所有子孙进程，检查此进程是否有等待读取的消息，有则释放消息缓冲区，最后释放该进程的工作空间和PCB空间，以及其它资源

2.3.3 进程控制

中国矿业大学

3. 阻塞与唤醒进程

- 进程阻塞是进程的一种自主行为，是进程为了等待某事件的发生，或等待I/O操作的完成，而自己调用系统阻塞原语使得自己放弃CPU，进入阻塞队列中等待
- 当该进程所需要的资源可用或事件发生时，由释放资源或触发事件的进程调用唤醒原语，唤醒阻塞在该资源/事件上等待的进程

2.3.3 进程控制

3. 阻塞与唤醒进程

进程阻塞的实现过程：

- 进程停止运行，将CPU的现行状态存放到PCB的CPU状态保护区中
- 将该进程置阻塞状态，并把它插入到阻塞队列中
- 最后系统执行调度程序，将CPU分配给另一个就绪的进程。

2.3.3 进程控制

3. 阻塞与唤醒进程

进程唤醒的实现过程：

- 首先找到被唤醒进程的标识，让该进程脱离阻塞队列，转换为就绪状态，然后插入就绪队列，等待调度运行即可

2.3.3 进程控制

3. 阻塞与唤醒进程

- 阻塞和唤醒操作的主体和客体有区别
- **进程阻塞是进程的自我行为**
- **进程唤醒是被动行为，**当前处于阻塞状态的进程**必须被其相关进程唤醒**

2.3.3 进程控制

中国矿业大学

4. 挂起与激活进程

- 处于阻塞状态、就绪状态的进程均可被挂起，系统使用挂起原语将其挂起，其挂起状态取决于之前进程的状态，被挂起进程的非常驻内存部分将被交换到磁盘上
- 如果进程挂起的时间到或内存资源充足时，系统或相关进程的原语操作会激活被挂起的进程
- 挂起原语既可以由该进程自己调用，也可由其它进程或操作系统调用
- 但激活原语只能由其它进程或操作系统调用

2.4 进程的互斥与同步

中国矿业大学

- 并发运行的多个进程之间存在两种基本关系：互斥和同步
  - ✓互斥——在资源有限的系统上并发运行的多个进程之间由于争用资源而引发的竞争制约关系，互斥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

2.4 进程的互斥与同步

中国矿业大学

- ✓同步——为完成共同任务的并发进程基于某个条件来协调其运行进度、执行次序而产生的协作制约关系

- ✓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

- 并发进程可以是无关的，也可以是交互的
- 无关是指并发进程分别运行在不同的数据集上，一个进程的执行与其它进程无关
- 若两个进程共享了某数据集，则一个进程的执行可能影响其它进程的结果，即两进程间存在制约关系，形成交互的并发进程
- 交互的并发进程，如果其执行的相对速度控制不好，会出现所谓与时间有关的错误

2.4.1 与时间有关的错误

中国矿业大学

多终端系统：银行ATM、售票系统……

A → D → B → C → E → F

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

```
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

- 多个进程共享各种资源，某些资源在某一时刻只能允许一个进程使用，例如打印机、磁带机等硬件设备和变量、队列等数据结构，如果有多个进程同时使用这类资源就会造成混乱
- **临界资源**：在某段时间内只能允许一个进程使用的资源
- **临界区**：进程中访问临界资源的代码段

2.4.2 临界资源与临界区

- 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

- 由于对临界资源的使用必须互斥进行，所以进程在进入临界区时，首先判断是否有其它进程在使用该临界资源，
  - 如果有，则该进程必须等待
  - 如果没有，该进程才能进入临界区，使用临界资源，同时，关闭临界区，以防其它进程进入
  - 当进程用完临界资源时，要开放临界区，以便其它进程进入

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进入临界区内执行；
  2）如果已有进程在临界区，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

2.4.2 临界资源与临界区

1. 用软件方法管理临界区
   – 不需要硬件、操作系统或程序设计语言的任何支持

2.4.2 临界资源与临界区

先判断对方进程是否在临界区的方法：

- 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，若发现其他进程正在访问临界资源，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;
```

存在问题：

- 当进程i和进程j都没有进入临界区时，各自的访问标志flag都为false，进程while语句通过，进程各自进入下一个语句时，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

2. 用硬件方式管理临界区

1）禁止中断法

- 将中断关上，计算机系统在进程测试并进入临界区期间不响应中断，只有临界区访问完后系统才打开中断，从而保证了对临界资源的互斥访问

2.4.2 临界资源与临界区

- 缺点：（简单粗暴）

1. 影响计算机效率——如果关中断的时间太长，则会限制CPU交叉执行程序的能力，影响计算机系统的效率。
2. 不能及时处理重要程序——滥用关中断会引起计算机响应不及时，重要的中断程序不能及时处理，导致出现严重的后果
3. 在多CPU下失效——访问相同临界资源的临界区代码可能会被另一个进程在另一个CPU上执行

2.4.2 临界资源与临界区

2. 用硬件方式管理临界区

2）特殊指令法

- 特殊的硬件指令可以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：将TS指令看作一个函数，该函数有一个布尔参数x和一个返回值，当x为true时则置x为false，并返回true，否则返回false

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

2.4.2 临界资源与临界区

- TS指令
  - 将布尔变量x与临界区关联起来——如果x为true，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，如TS指令就是锁机制
  - 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

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
  - 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

中国矿业大学

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

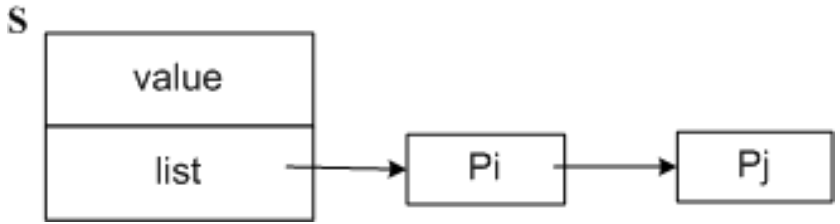
2.4.3 进程同步机制

中国矿业大学

- 信号量表示系统资源的实体
- 具体实现：信号量是一种记录型数据结构，含两个分量：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、实现进程同步和维护资源计数

2.4.3 进程同步机制

- 设信号量s为一个记录型数据结构，一个分量为整型量value，系统初始化时可对其赋值，另一个为信号量队列list，初始化状态下为空。



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唤醒一个处于阻塞状态的进程，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（伪代码）：

```
typedef struct semaphore {
    int value;              //信号量值
    struct pcb *list;       //阻塞进程队列指针
};
```

2.4.3 进程同步机制

- P、V操作原语的定义（伪代码）：

```
void P(semaphore &s) {
    s.value - -;
    if (s.value<0)  block(s.list);    //阻塞本进程并进入s信号量后的阻塞队列
}
void V(semaphore &s) {
    s.value++;
    if (s.value<=0)  wakeup(s.list);//唤醒s 信号量后的阻塞队列中的一个进程
}
```

2.4.3 进程同步机制

- 由信号量和P、V原语定义可得到如下结论：

1. 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
2. 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

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：

3. 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

2.4.3 进程同步机制

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
```

mutex的取值范围为[-(n-1), 1]

2.4.3 进程同步机制

- 互斥示例：
- 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
```

2.4.3 进程同步机制

- 同步示例：公共汽车上，司机和售票员的工作分别是：

  司机：启动车辆
  正常行驶
  到站停车

  售票员：关车门
  售票
  开车门

在公共汽车行驶过程中，如何用P,V操作实现司机和售票员的同步？

2.4.3 进程同步机制

中国矿业大学

分析：在公共汽车行驶过程中，司机和售票员的之间同步关系为：售票员关车门后，向司机发开车信号，司机收到开车信号后，启动车辆驾驶，在汽车正常行驶过程中售票员售票，到站时司机停车，售票员在车停后打开车门让乘客上下车。

司机启动车辆驾驶必须与售票员关车门的动作取得同步；售票员开车门的动作也必须与司机停车取得同步

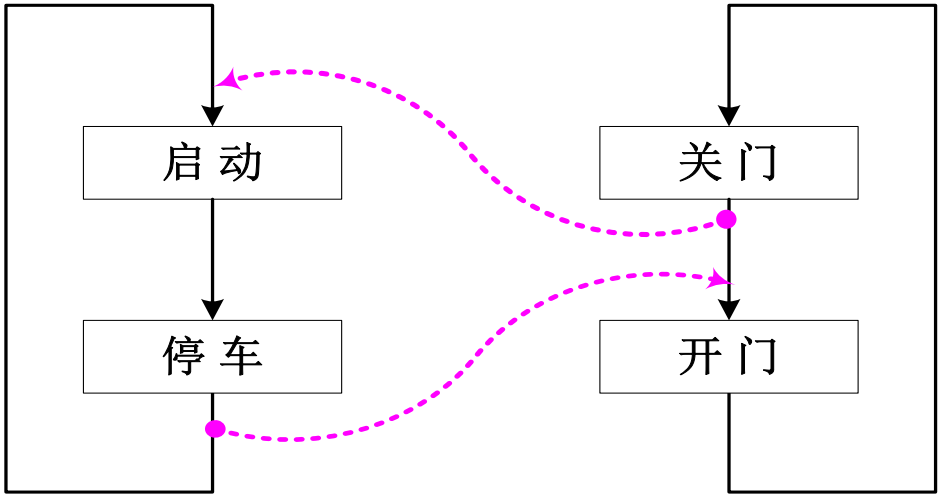
2.4.3 进程同步机制

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： 表示是否允许司机启动汽车，初值为0；
- 设信号量S： 表示是否允许售票员开门，初值为0；
- D.value=0，S.value=0；

2.4.3 进程同步机制

- 同步制约：

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

```
process conductor{
     while(true){
          关车门;
          V( D );
          卖票;
          P(S);
          开车门;
          上下乘客;
     }
}
```

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”模型
- ✓计算机系统中许多问题都可归结为生产者-消费者问题
  - 生产者进程可以是计算、发送进程
  - 消费者进程可以是打印、接收进程

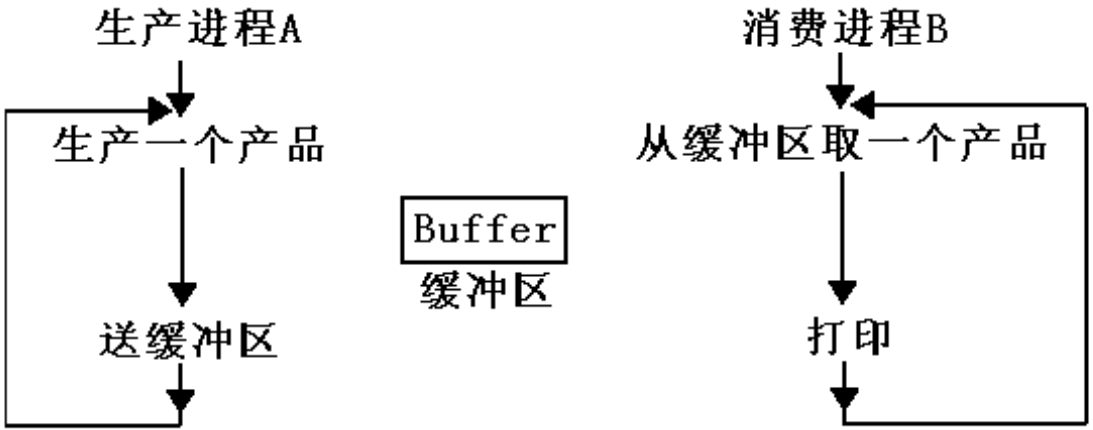
2.4.4 进程同步经典问题

1. 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

2.4.4 进程同步经典问题

1. 单缓冲区生产者—消费者问题



2.4.4 进程同步经典问题

1. 单缓冲区生产者—消费者问题

引入两个信号量：empty, full

- 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;
```

2.4.4 进程同步经典问题

用P、V操作解决同步与互斥步骤：

(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

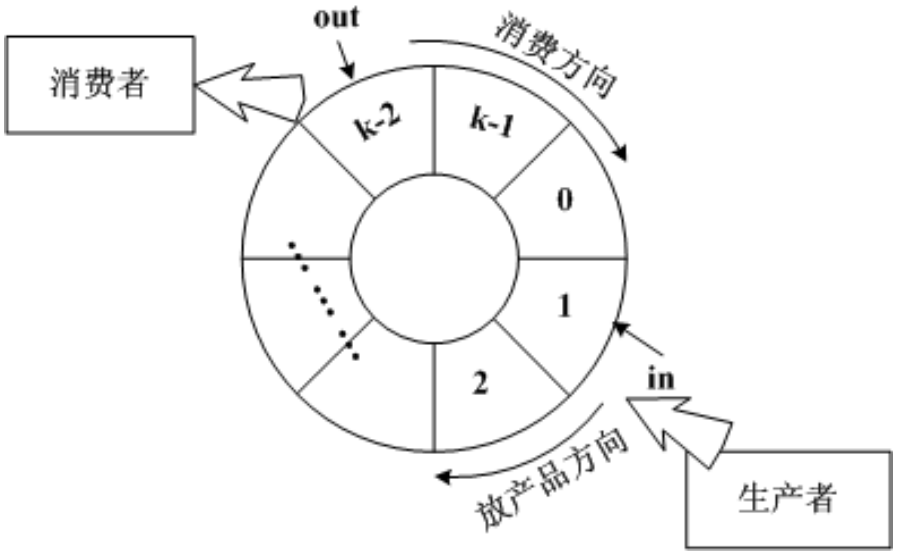
2.4.4 进程同步经典问题

2. 多生产者-消费者-多缓冲区问题

- 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）
- 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
- 一次只能有一个进程进入缓冲区取或者放产品

2.4.4 进程同步经典问题

2. 多生产者-消费者-多缓冲区问题



2.4.4 进程同步经典问题

2. 多生产者-消费者-多缓冲区问题

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

2. 多生产者-消费者-多缓冲区问题

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

(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

(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

2.4.4 进程同步经典问题

2．多生产者-消费者-多缓冲区问题

```
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
```

2.4.4 进程同步经典问题

Cobegin

```
process producer_i ( ) {
   while(true) {
      produce( );     //生产一个产品
      P(empty);      //申请空缓冲区
      P(mutex);      //申请互斥使用缓
   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);
   }
}
Coend
```

Empty(-n,k), full(-m,k)

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

2.4.4 进程同步经典问题

```
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);
   }
}
Coend
```

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

2.4.4 进程同步经典问题

- **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**

2.4.4 进程同步经典问题

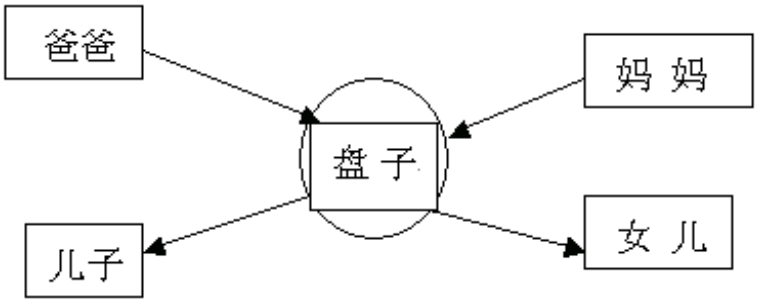
- 同样P(full)和P(mutex)的先后顺序不能弄错，否则当缓冲区全部为空时会引起消费者进程进入缓冲区后不能消费
- 生产进程也不能进入缓冲区放入产品，生产者进程与消费者进程同样进入死锁

- 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

3. 同步经典问题

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

- 本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。
- 分析其同步互斥关系如下：
- 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；

2.4.4 进程同步经典问题

- 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中的水果，则应发送“盘子中又可存放水果”的消息，但此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。

2.4.4 进程同步经典问题

中国矿业大学

```
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;
```

中国矿业大学

2.4.4 进程同步经典问题

4. 读者、写者问题

- 一个数据对象（比如一个文件或记录）被多个并发进程共享，其中一些进程只需要读该数据对象的内容，另一些进程则要求修改其内容。
- 只想读的进程称之为“读者”
- 要求修改的进程称为“写者”

2.4.4 进程同步经典问题

- 如果有两个甚至更多的读者同时访问这个可共享数据对象，那么任何一个读者的访问结果都是正确的。
- 但是，如果是一个写者和任何一个其它的读者或写者同时访问这个数据对象，就有可能导致不确定的访问结果。
- 所有类似的这类问题都可归结为“读者—写者问题”

2.4.4 进程同步经典问题

4. 读者-写者问题

——两组并发进程，读者和写者，共享一个文件，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

2.4.4 进程同步经典问题

- readcount记录当前正在访问该对象的读者个数
- 信号量mutex 用来互斥对readcount的修改，初值为1
- 信号量ws用于写者互斥，初值为1，它可由第一个要求访问该对象的读者和最后一个退出访问的读者使用，但它不会被中间的那些读者使用

```
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);
 }
```

2.4.4 进程同步经典问题

5. 阅览室读者

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。

坐位信号量 S=100，表示初始有100个空座位;
登记信号量 MUTEX=1，登记簿为临界资源，互斥使用;

```
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最大值：100（最多有100个空座位）
S最小值：-100（最多有100个人在等座位）
MUTEX最大值：1
MUTEX最小值：-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

课后习题10：

三个饮料厂P1、P2、P3都要生产橙汁，他们各自已购得三种必需原料（水、糖、浓缩汁）中的两种，待购得第三种原料后即可配制出售。有一供应商能够不断供应这些原料，但每次只能拿出一种原料放入容器出售，当容器中有原料时需要该原料的饮料厂可取走，容器空时，供应商又可放入一种原料，假定：P1已有糖和水，P2已有水和浓缩汁，P3已有糖和浓缩汁，试用PV操作写出供应商和三个饮料厂之间的同步进程。

2.4.4 进程同步经典问题

semaphore S=1, SO=0, SS=0, SW=0;

//容器是否可用, 容器中是浓缩汁/糖/水

```
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;
  }
}
```

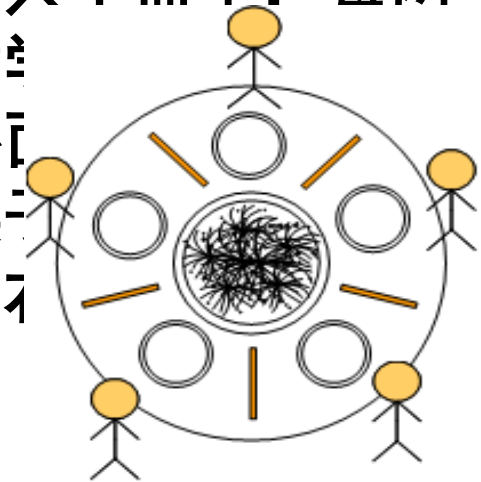
```
process P1 {
  while(true){
            P(SO);
            从容器中取浓缩汁;
            V(S);
            生产橙汁;
  } }
coend
```

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

2.4.4 进程同步经典问题

6. 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。每个哲学家交替地思考，感到饥饿，然后吃通心面。每个哲学家必须拿到两只筷子才能吃通心面，只能直接从自己的左手边和右手边拿筷子。



2.4.4 进程同步经典问题

- 哲学家就餐问题最简单解法：

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

2.4.4 进程同步经典问题

6. 哲学家就餐问题

- 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
```

2.4.4 进程同步经典问题

6. 哲学家就餐问题

- 信号量解决哲学家吃通心面问题的算法：

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);
}
```

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，属低级通信机制
- 进程通信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
- 主要有消息传递通信（Message Passing）、共享内存通信（Shared Memory）和管道通信（Pipe）

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

- 每个进程都有一个消息队列，其队列头由该进程的PCB中的消息队列指针指向
- 当来自其它进程的消息传递给它时，就将这些消息链入消息队列，进程按先来先服务的原则处理这一队列
- 消息传递通信机制包括：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）

2.5.1 消息传递通信

- send( )：
  - 查找接收进程的PCB，接收进程若存在，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，使用接收原语复制/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

- 发送进程与接收进程之间直接传递消息，需要发送进程与接收进程之间协调工作，才能做到可靠地发送和接收消息，否则，消息容易丢失
- 在直接通信中发送进程与接收进程之间的同步问题有两种同步方式

2.5.1 消息传递通信

- 当发送进程调用send原语发送消息后，有两种选择：

1）发送进程阻塞等待接收进程发回的确认信息，接收进程收到消息后会向发送进程回送确认信息，当发送进程接收到确认信息后，发送进程发送完成，被唤醒，继续执行自己的后继程序，通常情况下会删除已经发送的消息内容，释放消息缓冲区，

注意：发送进程阻塞等待时间的最长值要设置

2.5.1 消息传递通信

- 当发送进程调用send原语发送消息后，有两种选择：

2）发送进程发送完消息后，不阻塞等待接收进程的回送信息，而是继续执行自己的程序。期间，会接收到接收进程的回送信息。收到回送消息，则删除原来的发送信息。如果限定时间到还没有收到确认消息，或重发或放弃。

2.5.1 消息传递通信

- 接收进程调用receive原语后也有两种选择：

1）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。此种方式常与发送进程的第二种情况匹配。如果发送进程一直阻塞等待接收进程，则接收进程就不必等待发送进程，这样效率更高。

2.5.1 消息传递通信

- 接收进程调用receive原语后也有两种选择：

2）调用receive原语后，不阻塞等待发送来的消息，继续执行本身的程序。当需要接收消息时，再去查看消息，回送确认信息给发送进程。此种情况常与发送进程的第一种情况匹配。

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
  - 基于共享内存的通信方式
- 所谓共享存储区，是指为了传送大量数据，在内存中划出一块共享区域，多个进程可通过对共享区进行读写数据实现通信

2.5.2 共享内存通信

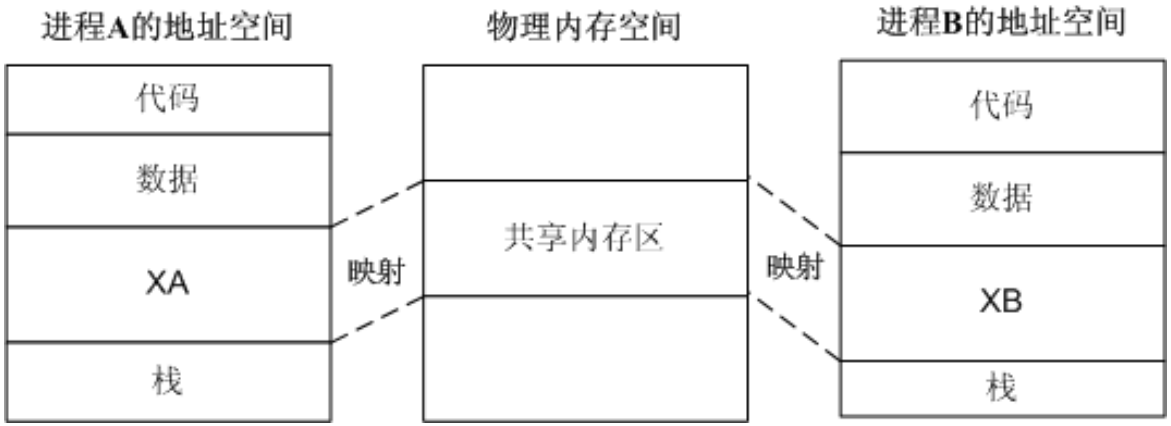


图2.13 共享内存通信机制

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

先利用系统调用建立一块共享内存区，并提供该共享内存区的标识符和共享内存区的长度等参数，若系统中已经建立了指定的共享内存区，则该系统调用将返回该共享内存区的描述符

2.5.2 共享内存通信

- 共享内存通信的实现过程如下：

2）共享内存区的管理

可通过系统调用对共享内存区的状态信息进行查询，如其长度、所连接的进程数、创建者标识符等；也可设置或修改其属性，如共享内存区的许可权、当前连接的进程计数等；还可用来对共享内存区加锁或解锁，以及修改共享内存区标识符等

2.5.2 共享内存通信

- 共享内存通信的实现过程如下：

3）共享内存区的映射与断开

建立共享内存区后，利用系统调用将该区映射到某进程的虚地址上，并指定该共享内存区是只读，还是可读可写，此后，该共享内存区便成为该进程虚地址空间的一部分，进程对其与其它虚地址空间一样存取和访问。

当进程不再需要该共享内存区时，再利用系统调用把该区与进程断开。

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（先进先出）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（存在于文件系统的高速缓冲区中）。

2.5.3 管道通信

- 创建管道的进程称为管道服务器，连接到一个管道的进程为管道客户。
- 进程对管道应该互斥使用，写进程把一定数量的数据写入管道，就阻塞等待，直到读进程取走数据后，将其唤醒。

2.5.3 管道通信

- 管道的两种实现机制：

1. 匿名管道：每次打开管道的进程，只能是与该进程相关的进程（子进程），他们共享该匿名管道，完成相互之间的通信，不提供全局服务

2.5.3 管道通信

- 管道的两种实现机制：

2. 命名管道：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。特别是为服务器通过网络与客户端交互而设计的命名管道，是一种永久通信机制，具有文件名、目录项、访问权限，能象一般文件一样操作，读写性能与匿名管道相同。

2.5.3 管道通信

- 每一个命名管道都有一个唯一的名字，以区分于存在于系统中的其它命名管道
- Windows系统中，管道服务器调用CreateNamedPipe( )函数创建命名管道；
- 管道客户通过调用CreateFile( )或CallNamedPipe( )函数以连接一个命名管道。

2.5.3 管道通信

- 命名管道通信模式：字节模式和消息模式
  - 在CreateNamePipe( )创建命名管道时分别用PIPE_TYPE_BYTE 和PIPE_TYPE_MESSAGE标志进行设定
- 字节模式：信息以连续字节流的形式在客户与服务器之间流动
- 消息模式：客户机和服务器通过一系列不连续的数据包进行数据的收发，从管道发出的每一条消息都必须作为一条完整的消息读入