# 第二章 进程与线程

## 进程

进程就是某个应用程序，是某种类型的活动。

四个主要事件会导致进程的创建

1 系统初始化

有的是前台进程，用于交互，其他的是后台进程，比如启动一个后台进程接收电子邮件，大部分时间都是在睡眠，当电子邮件到来时，被唤醒。

2 执行正在运行的进程所调用的进程创建系统调用

一个正在运行的进程经常会发出系统调用，以便创建一个或多个新进程来协助其工作。比如有大量的数据要通过网络调取并进程顺序处理，那么创建一个进程取数据，并把数据放入共享缓冲区，而让第二个进程取数据并处理。

3 用户请求创建一个新进程

4 一个批处理作业的初始化

用户在这种系统中提交批处理作业，在操作系统认为有资源可运行另一个作业时，它创建一个新的进程，并运行其输入队列中的下一个作业。

在linux中，只有一个系统调用可以用来创建新进程：fork。这个系统调用会创建一个与调用进程相同的副本，在调用fork之后，这两个进程（父进程和子）拥有相同的存储映像、同样的环境字符串和同样的打开文件。父进程和子进程有不同的地址空间，如果其中某个进程在其地址空间中修改了一个字，这个修改对于其他进程而言是不可见的。

对于一个新创建的进程而言，有可能共享其创建者的其他资源，比如打开的文件等。

进程的退出：

1 正常退出（自愿的）

当编译器完成了所给定程序的编译之后，编译器执行一个系统调用，通知操作系统它的工作已经完成。在UNIX中调用的是exit，在Windows中调用的是exitProcess。

2 出错退出（自愿的）

编译某个不存在的文件时，编译器就会退出。

3 严重错误（非自愿的）

进程引起的错误，通常是由程序中的错误导致。比如执行了一条非法的指令，引用不存在的内存，或者除数是0等。

4 被其他进程杀死

在UNIX中通过kill杀死。在Win32中对应的函数是TerminateProcess。

进程的层次结构

在UNIX中，进程和它的子女以及后裔组成了一个进程组。有父子进程关系，一个进程可以有多个子进程，但是只能有一个父进程。

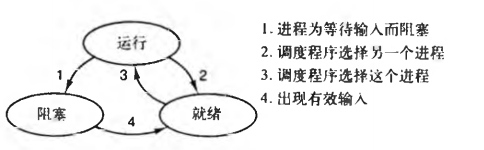
在Win中，没有进程层次的概念，所有进程地位相同的。只不过在创建进程时，父进程会得到一个令牌（称为句柄），该句柄可以用来控制子进程，但是它有权把这个令牌传给其他某个进程，这样就不存在进程层次了。在UNIX中，进程并不能剥夺其子女的继承权。

进程的状态

1 运行态（该时刻进程实际占用CPU）

2 就绪态（可运行，但因为其他进程正在运行而暂时终止）

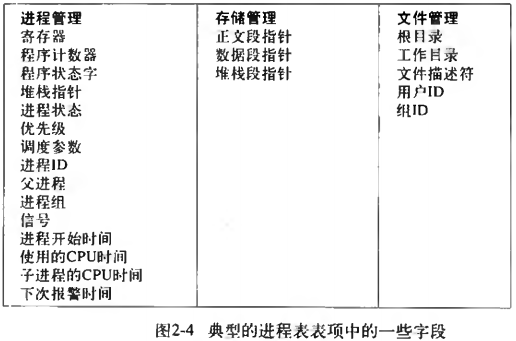
3 阻塞态（除非某种外部事件发生，否则进程不能运行）



进程的实现

为了实现进程模型，操作系统维护一张表格（一个结构数组），即进程表。

每个进程占用一个进程表项。表项中包含了程序计数器 堆栈指针 内存分配状况 所打开文件的状态 账号和调度信息以及其他在进程中由运行态转换到就绪态或者阻塞态时必须保存的信息，从而保证该进程随后能再次启动，就像没有被中断过一样。



当一个中断发生后，用户进程1正在运行，则中断硬件会将程序计数器 程序状态字，有时还将一个或多个寄存器压入堆栈，计算机随即跳转到中断向量所指示的地址。这些都是由硬件完成的操作。然后是软件，特别是中断服务例程就接管一切剩余的工作。

所有的中断都会先保存寄存器值，随后会在堆栈中删除由中断硬件机制存入堆栈的那部分信息，并将堆栈指针指向一个由进程处理程序所使用的临时堆栈。（这部分由汇编语言完成）

当该例程结束后，它调用一个C过程处理某个特定的中断类型剩下的工作，在完成有关工作之后，大概会使某些进程就绪，接着调用调度程序，决定随后该运行哪个进程。然后将控制权转给汇编，为当前进程装入寄存器值以及内存映射并启动该进程运行。

## 线程

为什么要多线程？

1 并行实体共享同一个地址空间和所有可用数据。

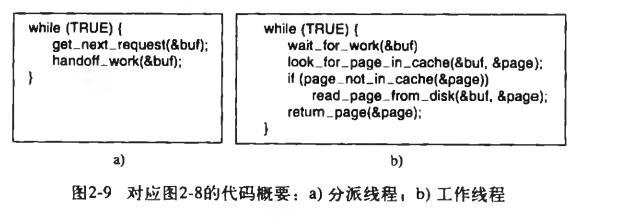
2 线程比进程更轻量级，所以线程比进程会更容快创建，也更容易撤销。

3 如果多个线程都是CPU密集型的，那么并不能获得性能上的增强，但是如果存在大量的计算和IO处理，拥有多个线程允许这些活动彼此重叠进行，从而会加快应用程序的执行速度。

例子：比如生产者和消费者需要共享数据

比如一个进程中需要多个线程，一个线程负责与用户交互，一个线程负责后台处理。提高响应性。

比如web服务。Servlet就是这样做的。有一个分派线程，负责接收请求和将请求转发给工作线程进行处理，有一个线程池，里面存放的是工作线程。正常时，工作线程处于睡眠状态，只有当有任务分派时，才会被分派线程唤醒进行工作。

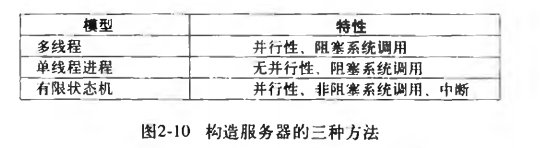


如果web服务器只有一个线程，那么web服务器的主循环获得请求 检查请求，并在取下一个请求之前完成整个工作。在等待磁盘操作时，服务器就空转，并且不处理任何到来的其他请求。

前面两种设计都是“顺序进程”的模型，也就是说每个线程按照通常方式顺序编程。

第三种是异步的，在请求到来时，这个唯一的线程对请求进行考察，如果该请求能够在高速缓存中得到满足，那么直接取出数据返回即可，如果不可以，则启动一个非阻塞的磁盘操作。服务器在表格中记录当前请求的状态，然后去处理下一个事件。下一个事件可能是一个新工作的请求，或者是磁盘对先前操作的回答，如果是新工作的请求，就开始工作。如果是磁盘的回答，就从表格中取出对应的信息，并处理该回答。对于非阻塞磁盘IO而言，这种回答多以信号或中断形式出现。这样的设计，每次服务器会从某个请求工作的状态切换到另外一个工作状态时，都必须显示保存或重新装入相应的计算状态。在这里每个计算都有一个被保存的状态，存在一个会发生且使得相关状态发生改变的事件集合，我们把这类设计称为有限状态机。

在多线程中，以顺序进程的思想保留下来，这种顺序进程阻塞了系统调用，但是仍旧实现并行性。



在处理大量数据时，多线程可以采用一个输入线程 一个处理线程和一个输出线程。输入线程将数据输入到输入缓冲区中，处理线程从输入缓冲区取出数据，处理数据，并把结果放到输出缓冲区，输出线程把这些结果写到磁盘上。这种模型只有当系统调用阻塞的是线程而不是整个进程时才能正常工作。

进程是将相关资源集中在一起，进程中存放程序正文和数据以及其他资源的地址空间，这些资源包括打开的文件 子进程 即将发生的报警 信息处理程序 账号信息等。把它们都放在进程中可以容易管理。

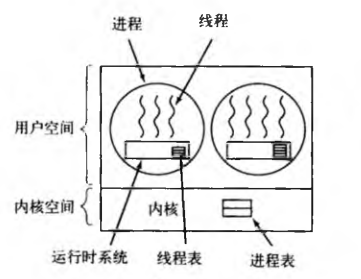
进程有一个执行的线程时，通过简写为线程。线程有一个程序计数器，用来记录接着要执行哪一条指令。线程拥有寄存器，用来保存线程当前的工作变量。线程有一个堆栈，用来记录执行历史，其中每一帧保存了一个已经调用的但是还没有从中返回的过程。

进程用于把资源集中在一起，而线程则是在CPU上被调度执行的实体。

多个线程共享同一个地址空间和其他资源，多个进程共享物理内存 磁盘 打印机和其他资源。

在用户空间实现线程

将整个线程包放在用户空间，内核对线程包一无所知。对于内核来说，就是单线程进程。



在用户空间管理线程时，每个进程需要有其专用的线程表，用来跟踪该进程中的线程。和进程表类似，不过它只记录各个线程的属性，如每个线程的程序计数器 堆栈指针 寄存器和状态等。该线程表由运行时系统管理，当一个线程转换到就绪状态或者阻塞状态时，在该线程表中存放重新启动该线程所需的信息，与内核在进程表中存放进程的信息完全一样。

当某个线程做了一些会引起本地阻塞的事情之后，例如，等待进程中另一个线程完成某项工作，它调用一个运行时系统的过程，这个过程检查该线程是否必须进入阻塞状态，如果是，则在线程表中保存该线程的寄存去，查看表中可运行的就绪线程，并把新线程的保存值重新装入机器的寄存器中。只要堆栈指针和程序计数器一被切换，新的线程就自动投入运行。如果机器有一条保存所有寄存器的指令和另一条装入全部寄存器的指令，那么整个线程的切换可以在几条指令内完成。

优点 1 用户线程包可以在不支持线程的操作系统上实现。

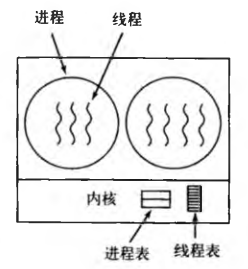
2 线程在完成运行时，比如调用yield方法，可以把该线程的信息保存在线程表中，进而可以调用线程调度程序来选择另一个要运行的线程。保存该线程状态的过程和调度程序都只是本地过程。所以启动它们比进行内核调用效率更高。另一方面，不需要陷阱，不需要上下文切换，也不需要对内存高速缓存进行刷新，这就使得线程调度非常快捷。

3 允许每个进程有自己定制的调度算法。

缺点：1 如何实现阻塞系统的调用。

2 线程之间的切换。除非一个线程自动放弃CPU，在单独的进程颞部，没有时钟中断，所以不可能轮流调度的方式调度进程。解决方案：让运行时系统请求每秒一次的时钟信号（中断），但是开销很大。

在内核中实现线程

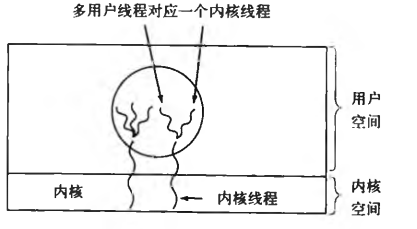


此时不需要运行时系统，每个进程中也没有线程表。在内核中有用来记录系统中所有线程的线程表。当某个线程希望创建一个新线程或者撤销一个线程时，就会进行一个系统调用，这个系统调用通过对线程表的更新完成线程创建或者撤销工作。

所有能够阻塞线程的调用都以系统调用的形式出现，代价比较大。当一个线程阻塞时，内核可以选择运行同一个进程的另一个线程，或者运行另一个进程中的线程。

混合实现

1 使用内核级线程，然后将用户级线程与某些内核线程多路复用起来。



内核只识别内核级线程，并对其进行调度。每个内核级线程有一个可以轮流使用的用户级线程集合。

弹出式线程

处理到来的消息，一种方式是将进程或者线程阻塞在一个receive系统调用上，等待消息到来，当消息到达时，该系统调用接收消息，进行处理。

另一方式是当有消息到达时，会导致系统创建一个处理该消息的线程，称为弹出式线程。

好处是没有历史，就不需要必须存储的寄存器 堆栈这些内容，每个线程从全新开始。每个线程都一样，可以快速创建这类线程。消息到达与处理开始之间的时间非常短。

在使用弹出式线程之前需要提前进行计划。比如哪个进程中的线程先运行。

## 进程之间通信

### 竞争条件

多个进程或者线程在读写数据时，其最终的结果依赖于多个进程的指令执行顺序。

比如两个进程共享了变量a，在某一时刻需要对a进程更新值。

### 临界区

对共享内存进行访问的程序片段称为临界区。

### 互斥的方案

1 屏蔽中断

进程在刚刚进入临界区后立即屏蔽所有中断，并在要离开之后打开中断。屏蔽中断后，时钟中断也被屏蔽。CPU只有发生时钟中断或其他中断时才会进行进程切换。

2 锁变量

设置一个锁变量，其初始值为0，当一个进程想进入临界区时，首先测试这把锁的值为0，则该进程将其设置为1并进入临界区。若这把锁的值已经是1，则该进程将等待直至将其值变为0.

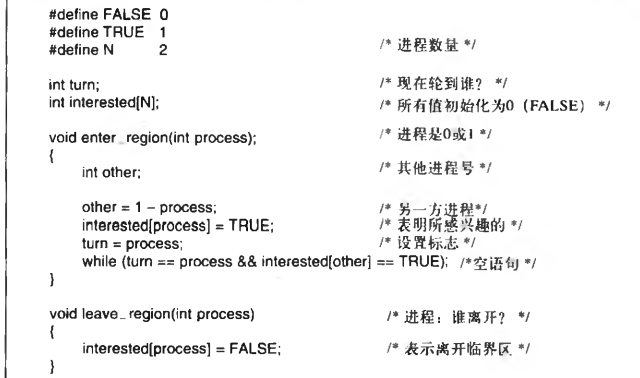
问题：如果一个进程读取锁变量值为0，但是此时另外一个进程将锁变量设为1，就冲突了。

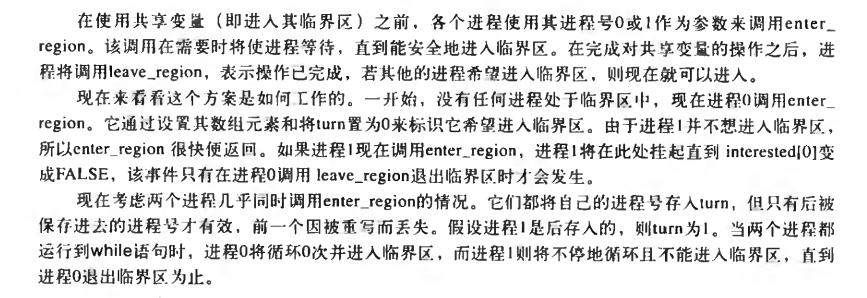
3 严格轮换法

整型变量turn，初始值为0，用于记录轮到哪个进程进入临界区，并检查或更新共享内存。开始时，进程0检查turn，发现其值为0，于是进入临界区，进程1发现其值为0，所以在一个忙等待中不停测试turn，看其值何时变为1.进程0离开临界区时，将turn的值设置为1，以便允许进程1进入临界区。假设进程1离开了临界区，turn值会被设为0.但是这样会导致一个进程被临界区之外的进程阻塞。

4 Peterson解法

将锁变量和警告变量的思想结合。





5 TSL指令

需要硬件支持。

TSL RX，LOCK 称为测试并加锁。将一个内存子lock读到寄存器RX中，然后在该内存地址上存一个非零值，读字和写字操作保证是不可分割的。即该指令结束之前，其他处理器均不允许访问该内存字。执行TSL指令的CPU将锁住内存总线，以禁止其他CPU在本指令结束之前访问内存。

lock是一个共享变量，当lock为0时，任何进程都可以使用TSL指令将其设置为1，并读写共享内存。当操作结束时，进程用一条remove指令将lock值设为0.

### 等待与唤醒

生产者消费者问题，也是有界缓冲区问题。

### 信号量

用于生产者消费者中。设置三个信号量，分别为full代表记录充满的缓冲槽数据 empty代表空的缓冲槽数目 mutex用来确保生产者和消费者不会同时访问缓冲区。初始时，full为0，empty为缓冲区槽的数目，mutex值为1.

用于实现同步。Full和empty保证了某种事件的顺序发生或者不发生。上例中，当缓冲区满的时候生产者停止生产，以及缓冲区空的时候消费者停止运行。

### 互斥量

### 条件变量

互斥量一般和条件变量一起使用。互斥量在允许或阻塞临界区的访问上起作用，条件变量则允许线程由于一些未达到的条件而阻塞。

### 管程

管程是由一个过程 变量及数据结构等组成的一个集合。进程可以在任何需要的时候调用管程中的过程，但它们不能在管程之外声明的过程中直接访问管程内的数据结构。

任意时刻，管程中只能有一个活跃线程。因此可以完成互斥。

当一个进程调用管程过程时，该过程中的前几条指令将检查在管程中是否有其他活跃进程，如果有，则调用进程将被挂起，直至另一个进程离开管程将其唤醒。如果没有活跃进程在使用管程，则该调用进程可以进入。

进入管程时的互斥是由编译器负责，但是通常是用一个互斥量或二元信号量。引入条件变量和wait和signal实现了管程中进程的阻塞。

如果向一个条件变量发送信号，但是在该条件变量上并没有等待进程，则该信号会丢失。因此wait操作必须在signal操作之前。

### 消息传递

前一个调用向一个目标发送消息，后一个调用从一个源中接收消息，如果没有消息可用，则接收者可能被阻塞，直至有一条消息到达。

解决生产者消费者问题。使用N条消息，相当于一块共享内存缓冲区中的N个槽。消费者首先将N条空消息发送给生产者，当生产者向消费者传递一个数据项时，它取走一条空消息并送回一条填充了内容的消息。消息的总数保持不变，所以消息都可以存在事先确定数量的内存中。如果生产者速度比较快，则所有的消息最终将被填满，等待消费者，生产者阻塞。。。

### 屏障

用于进程组。除非所有的进程都就绪准备进入下一个阶段，斗则任何一个进程都不能进入下一个阶段。

当一个线程到达屏障时，就会被屏障拦截，直至所有进程都到达该屏障为止。

## 调度

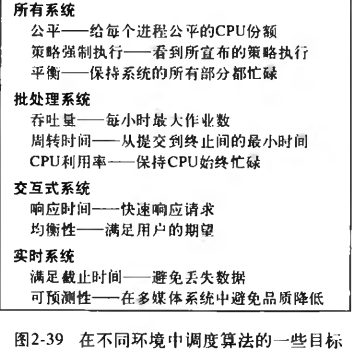
进程切换的代价比较高，首先用户态必须切换到内核态，然后保存当前进程的状态，包括在进程表中存储寄存器的值以便以后重新装载。在许多系统中，内存影像（如页表内的内存访问位）也必须保存，接着通过运行调度算法选定一个新进程，之后，应该将新进程的内存影像重新装入MMU，最后新进程开始运行。进程切换还会使整个内存高速缓存失效，强迫缓存从内存中动态装入两次（进入内核一次，离开内核一次）。如果每秒钟切换进程次数太多，会耗费大量CPU时间。

调度算法分类

批处理系统中，如处理存货清单 账目收入支出 利息计算等，可以采用非抢占式算法或对每个进程有长时间周期的抢占式算法，因为不会有用户在终端胖等待。这种处理方式减少了进程切换从而提高了性能。

在交互式环境中，为了避免一个进程霸占CPU而拒绝为其他进程服务，抢占是必须的。

在实时限制的系统中，抢占有时是不需要的，因为进程了解它们可能会长时间得不到运行，所以通常很快完成各自工作而阻塞。实时系统与交互式系统差别是，实时系统只运行那些用来现有应用程序，而交互式系统是通用的，可以运行任意的非协作甚至是有恶意的程序。



公平：相似的进程应该得到相似的服务。不同类型的进程可以采用不同的方式处理

与公平有关系的是系统策略的强制执行。

吞吐量是每小时完成的作业量。

周转时间：从一个批处理作业提交时刻开始直至该作业完成时刻为止的统计平均时间。

能够使吞吐量最大化的调度算法不一定有最小的周转时间。比如，对于确定的短作业和长作业的一个组合，总是运行短作业而不运行长作业的调度程序，可能获得出色的吞吐性能，但是对于长作业，周转时间很差。

CPU利用率常用于对批处理系统的度量。

最小响应时间：发出命令到得到响应之间的时间。

均衡性：用户对做一件事需要多长时间总是有一种固定的看法。如果点击一个图标花费60s显然不合理。

实时系统：或多或少必须满足截止时间。最主要的要求是满足所有的截止时间要求。

### 批处理系统中的调度

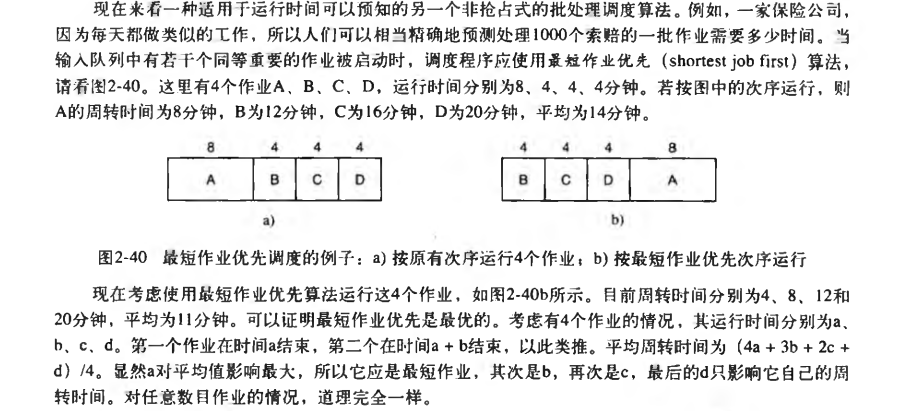
1 先来先服务

非抢占式。有一个就绪进程的单一队列。当正在运行的进程被阻塞时，队列中第一个进程就接着开始运行。在被阻塞的进程变为就绪时，就像一个新来到的作业一样，排到队列的末尾。

适合于CPU繁忙型作业，而不利于IO繁忙型的作业。

2 最短作业优先

适应于运行时间可以预知的。



3 最短剩余时间优先

调度程序总是选择剩余运行时间最短的那个进程运行。当一个新的作业到达时，如果新的进程比当前运行进程需要更少的时间，则当前进程就被挂起，而运行新的进程。

### 交互式系统的调度

1 轮转调度

每个进程 被分配一个时间段，称为时间片。允许该进程在该时间段上运行，如果时间段结束，进程还在运行，则剥夺CPU并分配给另一个进程。如果该进程在时间片结束之后阻塞或结束，则CPU立即切换。

维护一张可运行的进程列表。当一个进程用完他的时间片后，就会被移动到队列的末尾。

2 优先级调度

为了防止高优先级进程一直运行下去，调度程序可以在每个时钟中断降低当前进程的优先级。或者为每个进程赋予一个运行的最大时间片，当这个时间片用完时，下一个次高级的进程获得机会运行。

3 多级队列

4 最短进程优先

最短作业优先常常伴随着最短响应时间。

5 保证调度

6 彩票调度

7 公平分享调度

前面的调度关注的是进程自身，而不是关注所有者是谁。问题：用户1启动9个进程，用户2启动1进程，使用轮转或者相同优先级调度时，那么用户1有90%的CPU时间，而用户2只得到10%的CPU时间。

每个用户分配到CPU时间的一部分，比如两个用户都获得50%CPU时间的保证，那么无论一个用户有多少进程存在，每个用户都会得到应有的CPU份额。

### 实时系统中的调度

分为硬实时和软实时。前者是必须满足绝对的截止时间，后者是不希望偶尔错失截止时间，但是可以容忍。

## IPC问题（进程间通信）

### 哲学家就餐

1 每位哲学家都先拿起一只筷子，然后等待另外一只筷子。 会发生死锁。

2 每位哲学家都先拿起左边的叉子，然后检测右边叉子可用否，如果不可用，则放弃左边叉子，随机等待一段时间，再重新尝试上述过程。

3

# 第六章 死锁

资源死锁的条件

互斥条件 占有并等待条件（请求与保持条件） 不可剥夺条件 循环等待条件

### 鸵鸟算法

当对一件事没有一个很好的解决办法时，那就忽略它，就像鸵鸟面对危险时会把它的头埋在沙砾中，装作看不到。这样的算法称为鸵鸟算法。

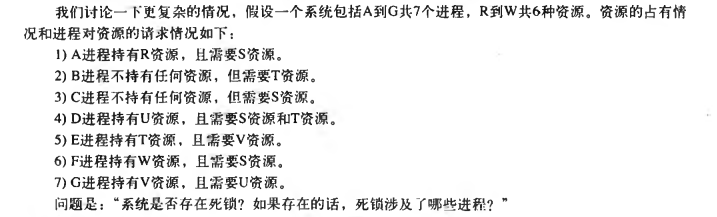
这是一种消极的策略，但是在实际系统中采用最多的一种策略。当死锁真正发生并且影响系统正常运行时，手动干预——重新启动。前提是假设这种问题出现的概率很低。当系统发生死锁时不会对用户造成多大影响。这样，开销可能比不允许发生死锁及检测和解除死锁的小。

### 死锁检测和死锁恢复

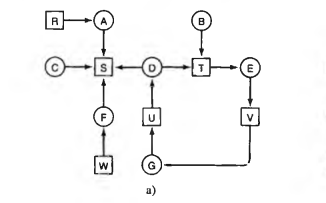
允许死锁的发生，当检测到死锁发生时，采取措施进程恢复。

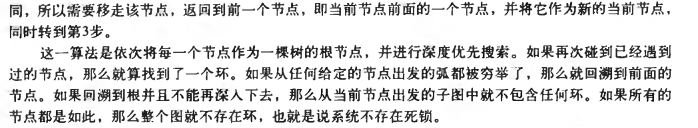
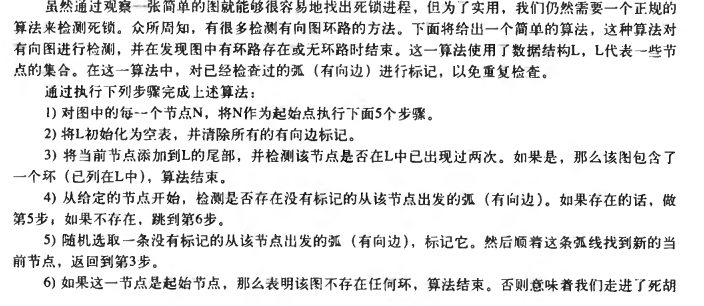
* 每种类型一个资源的检测

每种类型只有一个资源。



资源分配图如下：





重点：从每个节点开始进行深搜！

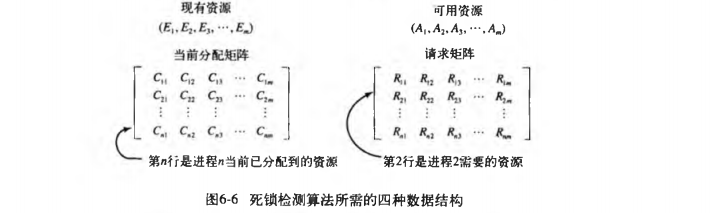
* 每种类型多个资源的死锁检测

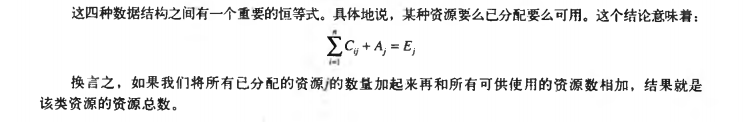
有多种相同的资源存在。

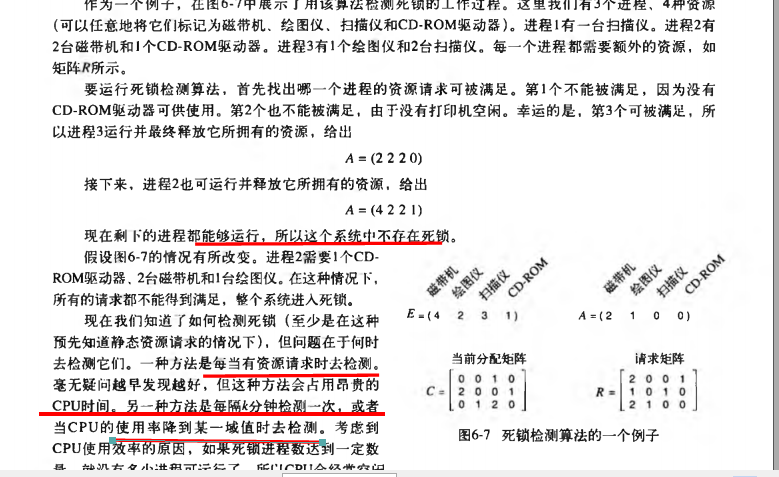
基于矩阵的算法来检测从P1到Pn这n个进程的死锁。假设资源的类型数是m，E1代表资源类型1，Ei代表资源类型i（1<=i<=m）。E是现有资源的向量，代表每种已存在的资源总数。如E1=2代表系统有2个E1类型的资源。

A表示可用资源向量，则Ai表示当前您可供使用的资源数。

C代表当前分配矩阵，R代表请求矩阵。C的第i行代表Pi当前所持有的每一种类型资源的资源数。Cij代表进程i所持有资源j的数量。Rij代表Pi所需要的资源j的数量。





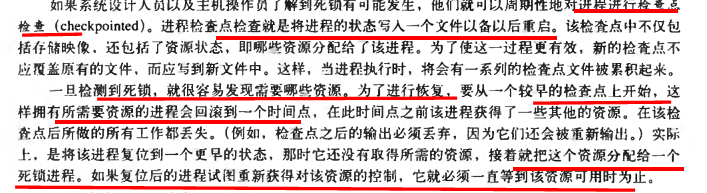


* 从死锁中恢复

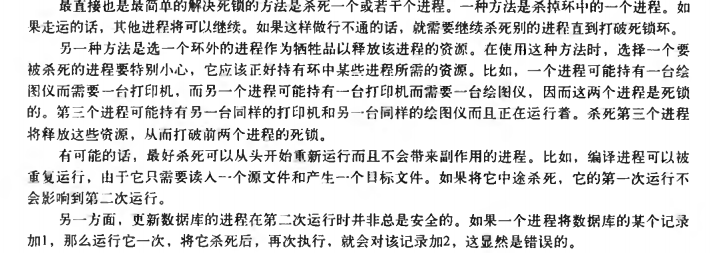
1 利用抢占恢复

这种做法是否可行取决于该资源本身的特性。用这种方法恢复通常比较困难或者不太可能，如果选择挂起某个进程，则取决于哪一个进程拥有比较容易收回的资源。

2 利用回滚恢复



3 通过杀死进程恢复



### 死锁避免

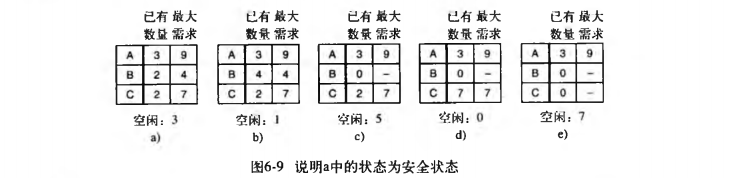
#### 资源轨迹图

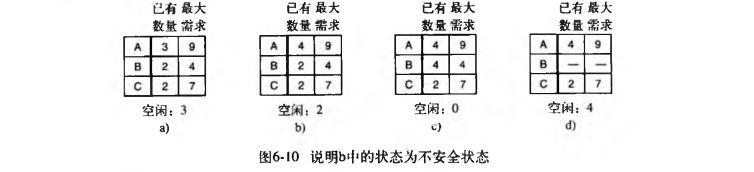
#### 安全状态和不安全状态

安全状态：存在一个分配序列使得所有的进程都能完成。

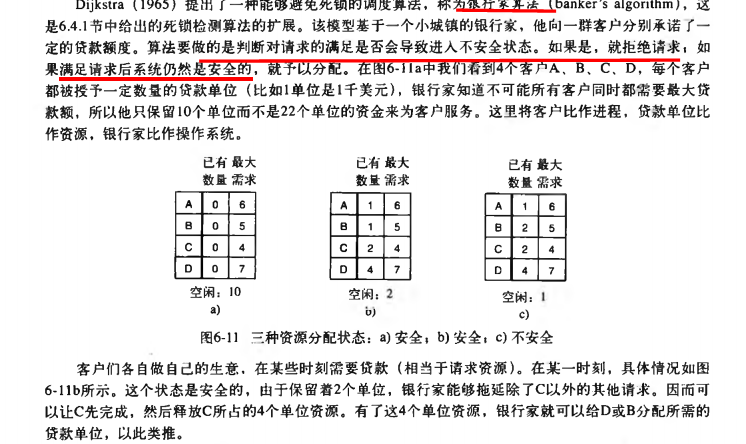
不安全状态并不是死锁。

从安全状态出发，系统能够保证所有进程都能完成，从不安全状态出发，是没有这种保证的。





#### 单个资源的银行家算法



#### 多个资源的银行家算法

P270

### 死锁预防

死锁避免从本质上来说是不可能的，因为它需要知道未来的请求，而这些请求是不可知的。

#### 破坏互斥条件

#### 破坏占有和等待条件

禁止已持有资源的进程再等待其他资源便可以消除死锁

方法一：规定所有进程在开始执行前请求所需的全部资源，如果所需的全部资源可用，那么将它们分配给这个进程，于是该进程肯定能够运行结束。如果一个或者多个资源正被使用，那么就不进行分配，进程等待。问题：一个进程在只有到运行时才知道它需要多少资源。而且资源利用率比较低。

方法二：当一个进程请求资源时，先暂时释放其当前占用的所有的资源，然后再尝试一次获得所需的全部资源。

#### 破坏不可抢占条件

#### 破坏循环等待条件

方法一：保证每个进程在任何时刻只能占用一个资源，如果要请求另外一个资源，就必须先释放另外一个资源。 不可取。

方法二：将所有资源统一编号，进程可以在任何时刻提出资源请求，但是所有请求必须按照资源编号的顺序（升序）提出。如果一个进程当前所拥有资源的编号是m，则不可以再申请小于m编号的资源。

### 其他问题

#### 两阶段加锁

在数据库系统中，当请求锁住一些记录，然后更新所有锁住的记录。当同时有多个进程在运行时，就有出现死锁的危险。

在第一阶段，进程试图对所有所需的记录进行加锁，一次锁一个记录。如果第一阶段加锁成功，就开始第二个阶段，完成更新然后释放锁。在第一阶段并没有做实际的工作。如果在第一阶段某个进程需要的记录已经被加锁，那么该进程释放它所有加锁的记录，然后重新开始第一阶段。

#### 通信死锁

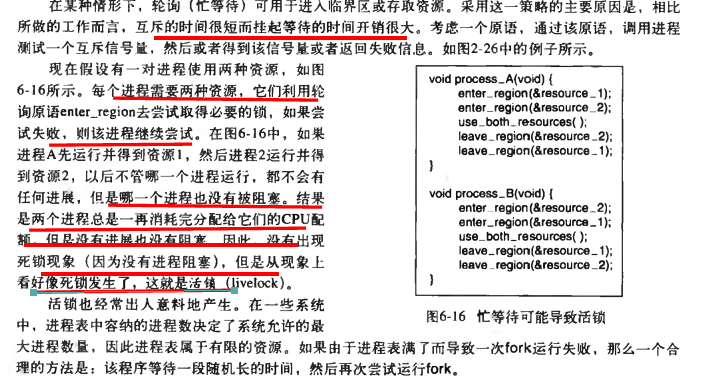
两个或两个以上进程利用发送信息来通信时，进程A向进程B发送请求信息，然后阻塞直至B回复。假设请求信息丢失，A将阻塞以等待回复，而B会阻塞等待一个向其发送命令的请求。因此发生了死锁。

这里并不是经典的资源死锁。

死锁标准的定义：在一系列进程中，每个进程因为等待另外一个进程引发的事件而产生阻塞。

可以通过超时来中断通信死锁。

#### 活锁



活锁是一系列进程在轮询等待某个不可能为真的条件为真。活锁的进程不会被阻塞，这可能导致耗尽cpu资源。

例子：

单一实体的活锁：线程从队列中拿出一个任务来执行，如果任务执行失败，那么将任务重新加入队列，继续执行。假设任务总是执行失败，或者某种依赖的条件总是不满足，那么线程一直在繁忙却没有任何结果。

协同导致的活锁：两个人在窄路中相遇，同时向一个方向避让，然后又让另一个方向避让，如此反复。在通信中，比如对讲机，同一时刻只能有一方发送信息，发送信号的用户会进行冲突检测，如果发生冲突，那么就选择避让，然后再发送。如果避让算法不合理，就会导致每次发送都冲突。

活锁解决：引入随机性。