死锁

系统模型

- 资源与实例
  - 有一个系统拥有有限数量的资源， 需要分配到若干竞争进程。 这些资源可以分成多种类型， 每种类型有一定数量的实例。
  - 如果一个进程申请某个资源类型的一个实例， 那么分配这种类型的任何实例都可满足申请。 否则， 这些实例就不相同， 并且资源分类没有定义正确。
- 在正常操作模式下， 进程只能按如下顺序使用资源：
  - 1.申请： 进程请求资源。 如果申请不能立即被允许（ 例如， 申请的资源正在被其他进程使用)， 那么申请进程应等待， 直到它能获得该资源为止。
  - 2.使用： 进程对资源进行操作（ 例如， 如果资源是打印机， 那么进程就可以在打印机上打印了）。
  - 3.释放： 进程释放资源。
- 死锁定义
  - 当一组进程内的每个进程都在等待一个事件， 而这一事件只能由这一组进程的另一个进程引起， 那么这组进程就处于死锁状态。

死锁特征

- 必要条件
  - 如果在一个系统中以下四个条件同时成立， 那么就能引起死锁：
    - • 互斥: 至少有一个资源必须处于非共享模式， 即一次只有一个进程可使用。 如果另一进程申请该资源， 那么申请进程应等到该资源释放为止。
    - • 占有并等待: 一个进程应占有至少一个资源， 并等待另一个资源， 而该资源为其他进程所占有。
    - • 非抢占: 资源不能被抢占， 即资源只能被进程在完成任务后自愿释放。
    - • 循环等待:有一组等待进程{P0，P1,...,Pn}， Pi等待的资源为 P(i+1) 占有，
- 资源分配图
  - 通过称为系统资源分配图 的有向图可以更精确地描述死锁。
    - 从进程 pi 到资源类型尽的有向边记为 Pi-> Rj, 它表示进程pi已经申请了资源类型 Rj 的一个实例， 并且正在等待这个资源。称为申请边
    - 从资源类型Rj到进程Pi的有向边记为Rj->Pi， 它表示资源类型Rj的一个实例已经分配给了进程Pi，称为分配边
  - 当该申请可以得到满足时， 那么申请边就立即转换成分配边。 当进程不再需要访问资源时， 它就释放资源， 因此就删除了分配边
  - 环与死锁的关系
    - 根据资源分配图的定义， 可以证明： 如果分配图没有环， 那么系统就没有进程死锁。 如果分配图有环， 那么可能存在死锁。
    - 如果每个资源类型刚好有一个实例， 那么有环就意味着已经出现死锁。 如果环上的每个类型只有一个实例， 那么就出现了死锁。 环上的进程就死锁。 在这种情况下， 图中的环就是死锁存在的充分且必要条件
    - 如果每个资源类型有多个实例， 那么有环并不意味着已经出现了死锁。 在这种情况下，图中的环就是死锁存在的必要条件而不是充分条件

死锁处理方法

- 一般来说， 处理死锁问题有三种方法
  - • 通过协议来预防或避免死锁， 确保系统不会进人死锁状态。
  - • 可以允许系统进人死锁状态， 然后检测它， 并加以恢复。
  - • 可以忽视这个问题， 认为死锁不可能在系统内发生。（大多数操作系统）

死锁预防

发生死锁有 4 个必要条件。 只要确保至少一个必要条件不成立， 就能预防死锁发生。

- 互斥
  - 互斥条件必须成立。 也就是说， 至少有一个资源应是非共享的。 相反， 可共享资源不要求互斥访问， 因此不会参与死锁。
- 持有且等待
  - 为了确保持有并等待条件不会出现在系统中， 应保证： 当每个进程申请一个资源时， 它不能占有其他资源。
  - 一种可以采用的协议是， 每个进程在执行前申请并获得所有资源。 这可以这样实现： 要求进程申请资源的系统调用在所有其他系统调用之前进行。
  - 另外一种协议允许进程仅在没有资源时才可申请资源。 一个进程可申请一些资源并使用它们。 然而， 在它申请更多其他资源之前， 它应释放现已分配的所有资源。
  - 两种主要缺点
    - 第一， 资源利用率可能比较低， 因为许多资源可能已分配， 但是很长时间没有被使用。
    - 第二， 可能发生饥饿。 一个进程如需要多个常用资源， 可能必须永久等待， 因为在它所需要的资源中至少有一个已分配给其他进程。
- 无抢占
  - 为了确保这一条件不成立， 可以采用如下协议： 如果一个进程持有资源并申请另一个不能立即分配的资源（ 也就是说， 这个进程应等待)， 那么它现在分配的资源都可被抢占。
- 循环等待
  - 确保这个条件不成立的一个方法是： 对所有资源类型进行完全排序， 而且要求每个进程按递增顺序来申请资源。

死锁避免

通过限制如何申请资源来预防死锁这种方法预防死锁有副作用： 设备使用率低和系统吞吐率低。

避免死锁的另一种方法需要额外信息，即如何申请资源。

- 安全状态
  - 如果系统能按一定顺序为每个进程分配资源（ 不超过它的最大需求）， 仍然避免死锁， 那么系统的状态就是安全的
  - 进程序列〈 P1,P2,...,Pn〉 在当前分配状态下为安全序列是指： 对于每个Pi， Pi仍然可以申请的资源数小于当前可用资源加上所有进程Pj （其中J< i)所占有的资源。 在这种情况下， 进程Pi， 需要的资源即使不能立即可用， 那么可以等待直到所有Pj释放资源。 当它们完成时， Pi可得到需要的所有资源， 完成给定任务， 返回分配的资源， 最后终止。
  - 安全状态不是死锁状态。 相反， 死锁状态是非安全状态
- 资源分配图算法
  - 引人一新类型的边， 称为需求边。 需求边Pi— Rj表示， 进程Pi可能在将来某个时候申请资源Rj。
  - 当Pi想要申请Rj的时候，只有将申请边Pi->Rj变成分配边同时不导致他成环，才允许申请，并采用环检测算法来判断安全性
- 银行家算法
  - 几个变量
    - Available: 长 度 为 m 的向量， 表示每种资源的可用实例数量。 如果Available[j]= K,那么资源类型Rj尽有k个可用实例。
    - Max,n*m矩阵,定义每个进程的最大需求。 如果 Max[i][j] = k， 那么进程Pi最多可申请资源类型Rj的k 个实例。
    - Allocation: n*m 矩阵， 定义每个进程现在分配的每种资源类型的实例数量。 如果AHocation[i][j]= k, 那么进程Pi现在已分配了资源类型Rj的 k 个实例。
    - Need:n*m矩阵，表示每个进程还需要的剩余资源。 如果 Need[i][j] =k， 那么进程Pi, 还可能申请k 个资源类型Rj的实例。 注意 Need[i][j]= Max[i][j]-Allocation[i][j]。
  - 安全算法
    - 1 • 令 Work 和 Finish 分别为长度 m 和n 的向量。 对于 i = 0， 1， n-1, 初始化 Work =Available 和 Finish[j]= false。
    - 2查找这样的i 使其满足Finish[i]== false,Need[i]<work,如果没有这样的i存在就转到第四步
    - 3. work=work+allocation,finish[i]=true,转到第二步
    - 4.如果对所有 i， Finish[i]= true, 那么系统处于安全状态。
  - 资源请求算法
    - 设 Request,为进程 P,的请求向量。 如果 Requesti[j]= k, 那么进程Pi需要资源类型Rj的实例数量为k 当进程Pi作出这一资源请求时， 就采取如下动作：
    - 子主题
      - 1.如果 Requesti<= Needi, 转到第 2 步。 否则， 生成出错条件， 这是因为进程Pi已超过了其最大需求。
      - 2.如果 Requesti< Available, 转到第 3 步。 否则， Pi应等待， 这是因为没有资源可用。
      - 3.假定系统可以分配给进程 P,请求的资源， 并按如下方式修改状态：
        Available = Available-Request,
        Allocationi = Allocationi + Requesti,
        Needi = Needi -Requesti
        如果新的资源分配状态是安全的， 那么交易完成且进程Pi， 可分配到需要的资源。 然而，如果新状态不安全， 那么进程 P,应等待 Request* 并恢复到原来的资源分配状态。

死锁检测

如果一个系统既不采用死锁预防算法也不采用死锁避免算法， 那么死锁可能出现。 在这种环境下， 系统可以提供：

- • 一个用来检查系统状态从而确定是否出现死锁的算法；
- • 一个用来从死锁状态中恢复的算法。

- 每种资源类型只有单个实例
  - 使用了资源分配图的一个变形， 称为等待图。 从资源分配图中， 删除所有资源类型节点， 合并适当边， 就可以得到等待图。
  - 与以前一样， 当且仅当在等待图中有一个环， 系统死锁。 为了检测死锁， 系统需要维护等待图， 并周期调用用于搜索图中环的算法。 从图中检测环的算法需要n^2数量级的操作，其中 n 为图的节点数
- 每种资源类型可有多个实例
  - 检测算法，类似于银行家算法
    - 1、设 Work 和 Finish 分别为长度为m 和n 的向量。 初始化 Work = Available。对i = 0,1,2,...,n-1, 如果 Allocationi不为 0, 则 Finish[i]= false; 否则， Finish[i]= true。
    - 2、找这样的同时满足 a.Finish[i]= false .b.Requesti<=work,如果没有这样的i,转到第四步
    - 3. work=work+allocation,finish[i]=true,转到第二步
    - 4.如果对某个 i(0<i <n), Finish[i] == false, 则系统死锁。 而且, 如果Finish[i]=false, 则进程尸,死锁。
- 应用检测算法
  - 何时应该调用检测算法？
    - • 死锁可能发生的频率是多少？
    - • 当死锁发生时， 有多少进程会受影响？

死锁恢复

- 进程终止
  - 通过中止进程来消除死锁， 有两种方法。 这两种方法都允许系统收回终止进程的所有分配资源。
    - 中止所有死锁进程。 这种方法显然会打破死锁环， 但是代价也大。 这些死锁进程可能已计算了较长时间； 这些部分计算的结果也要放弃， 并且以后可能还要重新计算。
    - 一次中止一个进程， 直到消除死锁循环为止。 这种方法的开销会相当大， 这是因为每次中止一个进程， 都应调用死锁检测算法， 以确定是否仍有进程处于死锁。
- 资源抢占
  - 通过资源抢占来消除死锁， 我们不断地抢占一些进程的资源以便给其他进程使用，直到死锁循环被打破为止。
    - 选择牺牲进程： 抢占哪些资源和哪些进程？ 与进程终止一样， 应确定抢占的顺序， 使得代价最小。 代价因素可包括这样的参数， 如死锁进程拥有的资源数量 、死锁进程到现在为止所消耗的时间等。
    - 回滚： 如果从一个进程那里抢占了一个资源， 那么应对该进程做些什么安排？ 显然，该进程不能继续正常执行； 它缺少所需的某些资源。 我们应将该进程回滚到某个安全状态， 以便从该状态重启进程。
    - 饥饿： 如何确保不会发生饥饿？ 即如何保证资源不会总是从同一个进程中被抢占。