第 4 章 死锁

在多道程序环境下，多个进程可能竞争一定数量的资源。某个进程申请资源，如果这时资源不可用，那么该进程进入等待状态。如果所申请的资源被其他等待进程占有，那么该等待进程有可能再也无法改变其状态。这种情况称为死锁(deadlock)。前面已经结合信号量讨论了这类情况。

生活中死锁例子可能是 Kansas 立法机构于 20 世纪初通过的一个法规，其中说到“当两列列车在十字路口逼近时，它们要完全停下来，且在一列列车开走之前，另一列列车不能启动。”

本章将介绍一些操作系统用于预防或处理死锁的方法。虽然现在绝大多数操作系统并不提供死锁预防功能，但是这些功能也可能很快会加入。随着更多数量的进程、多线程程序、更多的系统资源，死锁问题会变得更普遍。

4.1 死锁的概念

在计算机系统中存在一定数量的资源，分布在若干竞争进程之间共同使用。这些资源可分成多种类型，每种类型有一定数量的实例。资源类型的例子有内存空间、CPU 周期、文件、I/O 设备（打印机和 DVD 驱动器）等。如果系统有两个 CPU，那么资源类型 CPU 就有两个实例。类似地，资源类型打印机可能有 5 个实例。

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，且资源类型的分类也没有正确定义。例如，一个系统有两台打印机。如果没有人关心哪台打印机打印哪些输出，那么这两台打印机可定义为属于同一资源类型。然而，如果一台打印机是激光打印，而另一台是喷墨打印，那么对输出质量有要求的用户或许就不会认为这两台打印机是相同的，这样每个打印机就可能需要定义为属于不同类型。

进程在使用资源前必须申请资源，在使用资源之后必须释放资源。一个进程可能会申请许多资源以便完成其指定的任务。显然，所申请的资源数量不能超过系统所有资源的总量。换言之，如果系统只有两台打印机，那么进程就不能申请三台打印机。

在正常操作模式下，进程只能按如下顺序使用资源：

①申请：如果申请不能立即被允许（例如，所申请资源正在为其他进程所使用）．那么申请进程必须等待，直到它获得该资源为止。

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。

如先前章节所述，资源的申请与释放为系统调用，例如针对设备的有 request()/release()系统调用针对文件的有 open()/close()系统调用，针对内存的有 allocate()/free()系统调用。因此，对于进程或线程的每次使用，操作系统会检查以确保使用进程已经申请并获得了资源。操作系统使用一个专门的表来记录每个资源是否空闲或已被分配，分配给了哪个进程。如果进程所申请的资源正在为其他进程所使用，那么该进程会增加到该资源的等待队列。

这种对资源的申请和等待可能会诱发死锁。当一组进程中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进程引起，那么这组进程就处于死锁状态。这里所关心的主要事件是资源获取和释放。资源可能是物理资源（例如，打印机、磁带驱动器、内存空间和 CPU 周期），也可能是逻辑资源（例如，文件、信号量）。

为说明死锁状态，我们以一个具有三个 CD 刻录机的系统为例。假定有三个进程，每个进程当前

都占用了一个 CD 刻录机。如果每个进程现在需要另一个刻录机，那么这三个进程会处于死锁状态。每个进程都在等待 “CD 刻录机释放” 事件，而这恰恰又只可能由另外两个等待进程来完成。这个例子说明了涉及同一种资源类型的死锁。

死锁也可能涉及不同资源类型。例如，某一个系统有一台打印机和一台 DVD 驱动器。假如当前进程 Pi 占有 DVD 驱动器而 Pj 占有打印机，如果 Pi 申请打印机而 Pj 申请 DVD 驱动器，那么就会出现死锁。

开发多进程或者多线程应用程序的程序员必须特别关注死锁问题，因为多个线程可能因为竞争共享资源而容易产生死锁。

当出现死锁时，进程永远不能完成，并且系统资源被阻碍使用，阻止了其他作业开始执行。在讨论处理死锁问题的各种方法之前，先深入讨论一下死锁的特征。

1. 必要条件

如果在一个系统中下面 4 个条件同时满足，那么会引起死锁。

①互斥（mutual exclusion）：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使用。如果另一进程申请该资源，那么申请进程必须等到该资源被释放为止。

②占有并等待（hold and wait）：一个进程必须占有至少一个资源，并等待另一资源，而这另一个资源正好为其他进程所占有。

⑧非抢占（non-preemptive）：资源不能被抢占，即资源只能在进程完成任务后主动释放。

④循环等待（circular wait）：有一组等待进程{P0，P1，…，Pn），P0 等待的资源为 Pl 所占有，Pl 等待的资源为 P2 所占有，……，Pn-1 等待的资源为 Pn 所占有，Pn 等待的资源为 P0 所占有。

在此，强调所有 4 个条件必须同时满足才会出现死锁。循环等待条件隐含着占有并等待条件，所以这 4 个条件并不完全独立。然而，在后面我们将会看到分开考虑这些条件是有意义的。

2 资源分配图

死锁问题可用系统资源分配图进行更为形象化的描述。这种图由一个节点集合 V 和一个边集合 E 组成。节点集合 V 可分成两种类型的节点：系统活动进程的集合 P={Pl，P2，…，Pn}和系统所有资源类型的集合 R ={R1，R2，…，Rm}。

由进程 Pi 到资源类型 Rj 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，并正在等待该资源。由资源类型 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进程 Pi。有向边 Pi→Rj 称为申请边，有向边 Rj→Pi 称为分配边。

在图上，用圆形表示进程 Pi，用矩形表示资源类型 Rj。由于资源类型 Rj 可能有多个实例，所以在矩形中用圆点表示实例。注意申请边只指向矩形 Rj，而分配边必须指定矩形内的某个圆点。

当进程 Pi 申请资源类型 Rj 的一个实例时，就在资源分配图中加入一条申请边。当该申请可以得到满足时，那么申请边就立助转换成分配边。当进程不再需要访问资源时，它就释放资源，因此就删除分配边。

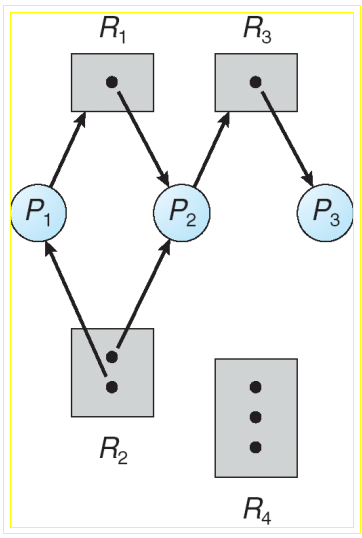

图 4.1 一个资源分配图

图 4.1 的资源分配图表示了如下情况：

集合 P、R 和 E：P={P1, P2，P3}；R={R1,R2，R3，R4}；E={Pl→R1, P2→R3, R1→P2, R2→P2, R2→P1, R3→P3}。

资源实例：资源类型 R1 有 1 个实例，资源类型 R2 有 2 个实例，资源类型 R3 有 1 个实例，资源类型 R4 有 3 个实例。

进程状态：进程 Pl 占有资源类型 R2 的 1 个实例，等待资源类型 R1 的 1 个实例；进程 P2 占有资源类型 R1 的 1 个实例和资源类型 R2 的 1 个实例，等待资源类型 R3 的 1 个实例；进程 P3 占有资源类型 R3 的 1 个实例。

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。

单实例资源情况

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。如果环涉及一组资源类型，而每个类型只有一个实例，那么就出现死锁。环所涉及的进程就死锁。在这种情况下，图中的环就是死锁存在的充分必要条件。

多实例资源情况

如果存在某些资源类型有多个实例，那么图中有环并不意味着肯定已经出现了死锁。在这种情况下，图中的环就是死锁存在的必要条件但还不是充分条件。

为了说明这种，我们回到图 4.1 所示的资源分配图。假设进程 P3 申请了资源类型 R2 的一个实例。由于现在没有资源实例可用，所以就增加了一条请求边 P3→R2，形成图 4.2。

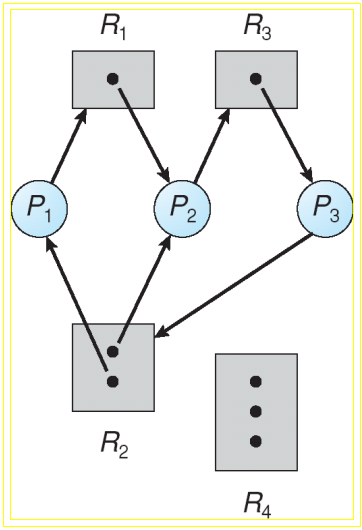

图 4.2 存在死锁的资源分配图

这时，在系统中有两个最小环：P1→R1→P2→R3→P3→R2→P1 和 P2→R3→P3→R2→P2。此时，进程 Pl、P2 和 P3 进入死锁状态。进程 P2 等待资源类型 R3，而它又被进程 P3 占有。另一方面，进程 P3 等待进程 P1 或进程 P2 以释放资源类型 R2。另外，进程 Pl 等待进程 P2 释放资源 R1。

现在考虑图 4.3 所示的资源分配图。在这个例子中，也有一个环：P1→R1→P3→R2→P1。然而，却没有死锁。注意进程 P4 可能释放资源类型 R2 的一个实例。这个资源以后便可分配给进程 P3，从而以打破了环。

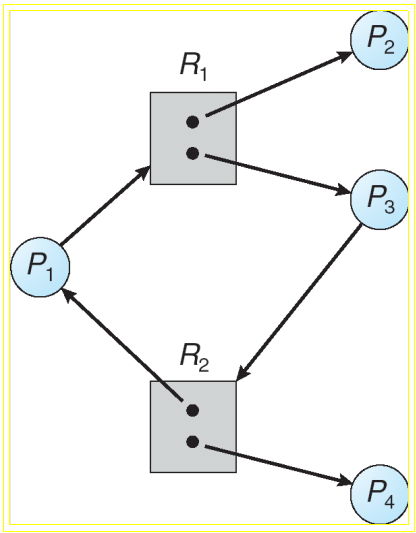

图 4.3 存在环但是没有死锁的资源分配图

总而言之，如果资源分配图没有环，那么系统就不处于死锁状态。另一方面，如果有环，那么又分两种情况：如果所有资源都是单实例的，那么系统就处于死锁状态；如果某些资源是多实例的，那么系统可能会处于死锁状态。

正是由于这种情况，我们在后面会看到，如果所有资源是单实例的，那么死锁检测可以采用基于资源分配图的回路检测算法；如果资源是多实例的，那么我们将采用另一种更通用的算法。

4.2.3 死锁的处理方法

从原理上来说．有三种方法来处理死锁问题：

·通过在操作系统中使用某种机制，确保系统不会进入死锁状态。

·允许系统进入死锁状态，然后检测它，并加以恢复。

·在操作系统的设计和实现中忽视这个问题，把责任留给应用程序设计者。

这里第三种方法看似驼鸟策略，对死锁问题视而不见，但恰恰为绝大多数操作系统所采用，包括著名的 UNIX 和 Windows 系统。至于为什么许多系统都这么做，等我们学习分析了前面这些方法之后，你会有所理解。因此在这些系统中，应用程序开发人员需要自己来处理死锁。

为了确保死锁不会发生，系统可以采用死锁预防或死锁避免方案。死锁预防(deadlock prevention)是一组方法，以确保至少一个必要条件不成立。这些方法通过限制如何申请资源的方法来预防死锁。在第 3 节中将讨论这些方法。

死锁避免（deadlock avoidance）要求操作系统事先得到有关进程申请资源和使用资源的额外信息。有了这些额外信息，系统可确定：对于一个申请，进程是否应等待。为了确定当前申请是被立即批准还是需要延迟，系统必须考虑现有可用资源、已分配给每个进程的资源、每个进程将来申请和释放的资源。在第 4 节中将讨论这些方法。

如果系统不使用死锁预防或死锁避免算法，那么就可能发生死锁情况。在这种情况下，系统提供一个死锁检测算法来检查系统状态以确定死锁是否发生，并提供另一个死锁恢复算法来从死锁中恢复。在第 5 节中将讨论这些方法。

如果系统不能确保不会发生死锁，且也不提供机制进行死锁检测和恢复，那么就真有可能出现这种情况：系统处于死锁而又没有办法检测到发生了什么。在这种情况下，未检测的死锁会导致系统性能下降，因为资源被不能运行的进程所占有，而越来越多进程会因申请资源而进入死锁。最后，整个系统会或许会停止工作，并且需要人工重新启动。

虽然看起来最后这种视而不见的态度似乎不是一个解决死锁问题的可行方法，但是它却为不少操作系统所使用。因为对于许多系统，死锁确实很少发生（如一年一次），因此，与其使用频繁的并且开销昂贵的死锁预防、死锁避免、死锁检测与恢复，这种方法更为便宜而且实际。而且，在有的情况下，系统实际上可能处于冻结状态而不是死锁状态。比如，当一个实时进程运行于最高优先级（或其他进程运行于非抢占调用程序之下）且它不将控制返回给操作系统。这时，系统必须人工地从非死锁状态中进行恢复。