4.2 死锁预防

如 4.2.1 小节所述，出现死锁有 4 个必要条件。在理论上，只要确保其中一个必要条件不成立，就能预防死锁的发生。下面通过详细讨论这 4 个必要条件来研究死锁预防的方法。

1. 互斥

对于非共享资源，必须要有互斥条件。例如，一台打印机不能同时为多个进程所共享。另一方面，共享资源不要求互斥访问，因此不会涉及死锁。共享资源的一个很好的例子是只读文件。如果多个进程试图同时打开只读文件，那么它们能同时获得对只读文件的访问。进程决不需要等待共享资源。然而，通常不能通过否定互斥条件来预防死锁：有的资源本身的特征就决定了它就是非共享的。

结论：通过破坏“互斥”条件来预防死锁，不可行。

2. 占有并等待

为了确保占有并等待条件不会在系统内出现，必须保证：当一个进程申请一个资源时，它不能占有其他资源。一种可以使用的方法是要求每个进程在执行前先申请并获得它所需要的所有资源。可以实现通过要求申请资源的系统调用在所有其他系统调用之前进行。

另外一种方法是允许进程在不占有任何资源时才可以申请资源。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，它必须先释放其现已分配的所有资源。

为了说明这两种方法之间的差别，考虑一个进程，它将数据从 DVD 驱动器复制到磁盘文件，并对磁盘文件进行排序，再将结果打印到打印机上。如果所有资源必须在进程开始之前申请，那么进程必须一开始就申请 DVD 驱动器、磁盘文件和打印机。在其整个执行过程中，它会一直占有打印机，尽管它只在结束时才需要打印机。

第二种方法允许进程在开始时只申请 DVD 驱动器和磁盘文件。它将数据从 DVD 复制到磁盘，再释放 DVD 驱动器和磁盘文件。然后，进程必须再申请磁盘文件和打印机。当数据从磁盘文件复制到打印机之后，它就释放这两个资源并终止。

虽然第二种方法对第一种方法进行了改进，但是这两种方法有两个主要缺点。第一，资源利用率（resource utilization）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。例如，在所给的例子中，只有确认数据始终在磁盘文件上的情况下，才可以释放 DVD 驱动器和磁盘文件，并再次申请磁盘文件和打印机资源。否则，不管采用哪种协议，必须在开始之前申请所有资源。第二，可能发生饥饿。当一个进程需要多个常用资源时，可能会永久等待，因为其所需要的资源中很可能至少有一个已分配给其他进程，总是无法凑完整。

结论：通过破坏“占有并等待”条件来预防死锁，会引起资源利用率大幅度下降，并可能诱发饥饿现象。

3. 非抢占

第三个必要条件是对已分配的资源不能抢占。为了确保这一条件不成立，可以使用如下方法：如果一个进程占有资源并申请另一个不能立即分配的资源，那么其现已分配的资源都可被抢占。换句话说，这些资源都被隐含地释放了。被抢占的这些资源会被操作系统放置到进程所等待的资源链表上。只有当进程获得其原有资源和所申请的新资源时，进程才可以重新执行。

换句话说，如果一个进程申请一些资源，那么首先检查它们是否可用。如果可用，那么就分配

它们。如果不可用，那么检查这些资源是否已分配给其他等待额外资源的进程。如果是，那么就从等待进程中抢占这些资源，并分配给申请进程。如果资源不可用且也不被其他等待进程占有，那么申请进程必须等待。当一个进程处于等待时，如果其他进程申请其拥有资源，那么该进程的部分资源可以被抢占。一个进程要重新执行，它必须分配到其所申请的资源，并恢复其在等待时被抢占的资源。

这个方法通常可用于那些状态可以保存和恢复的资源，如 CPU 寄存器和内存。但它一般不适用于其他资源，如打印机和磁带驱动器。

结论：通过破坏“非抢占”条件来预防死锁，不是一个通用的方法。

4. 循环等待

死锁的第 4 个也是最后一个条件是循环等待。一个确保此条件不成立的方法是对所有资源类型进行完全排序，且要求每个进程按递增顺序来申请资源。

设 R={R1, R2，…，Rm}为资源类型的集合。为每个资源类型分配一个唯一整数来允许比较两个资源以确定其先后顺序。可以定义一个函数 F：R→N，其中 N 是自然数的集合。例如，如果资源类型 R 的集合包括磁带驱动器、磁盘驱动器和打印机，那么函数 F 可以定义如下：

F(tape drive) = 1

F(disk drive) = 5

F(printer) = 12

有了这个函数，可以采用如下方法以预防死锁：每个进程只能按递增顺序申请资源，即一个进程开始可申请任何数量的资源类型 Ri 的实例。之后，当且仅当 F(Rj) > F(Ri) 时，该进程才可以申请资源类型 Rj 的实例。如果需要同一资源类型的多个实例，那么对它们必须一起申请。例如，对于以上给定函数，一个进程需要同时使用磁带驱动器和打印机，那么就必须先申请磁带驱动器，再申请打印机。换句话说，当一个进程申请资源类型 Rj 时，它必须先释放所有资源 Ri，其中 F(Ri)≥F(Rj)。

下面我们分析一下为什么这个方法可以预防死锁的发生。如果使用这个方法，那么循环等待就不可能成立。可以通过反证法来证明这一点。假定有一个循环等待存在。设涉及循环等待的进程集合为{P0，P1，…, Pn}，其中 Pi 等待一个资源 Ri，而 Ri 又为进程 Pi+l 所占有（最后 Pn 等待由 P0 所占有的资源 Rn）。因此，由于进程 Pi+l 占有资源 Ri 而同时申请资源 Ri+1，所以对所有 i，必须有 F(Ri) < F(Ri+1)。而这意味着 F(R0) < F(R1) < … < F(Rn) < F(R0)。根据传递规则，有 F(R0) < F(R0)，这显然是不可能的。因此，不可能有循环等待。

通过对系统内所有资源对象进行完全排序，可以在应用程序中实现这种方案。所有这些资源对象的请求必须按递增顺序进行。

请记住，设计一个资源完全排序并不能防止死锁，而是要靠应用程序员来按顺序编写程序。另外，函数 F 应该根据系统内资源使用的正常顺序来定义。例如，由于磁带通常在打印机之前使用，所以定义 F(tape drive) < F(printer)较为合理。

结论：这种方法需要设计一个良好的排序函数 F，还需要应用程序按照排序的顺序来申请资源。

4.3 死锁避免

在前一节讨论的死锁预防算法中，通过限制资源申请的方法来预防死锁。这种限制确保 4 个必要条件之一不会发生，因此死锁不成立。然而，通过这种方法预防死锁的代价是降低设备使用率和系统吞吐率。

避免死锁的另一种方法是事先获得进程以后将如何申请资源的附加信息。例如，对于有一台磁带

驱动器和一台打印机的系统，可能知道进程 P 会先申请磁带驱动器，再申请打印机，之后释放这些资源。另一方面，进程 Q 会先申请打印机，再申请磁带驱动器。有了关于每个进程的申请与释放的完全顺序，操作系统便可决定进程是否因申请而等待。每次申请要求系统考虑现有的可用资源、现已分配给每个进程的资源和每个进程将来申请与释放的资源，以决定当前的申请是否立即满足还是必须等待，从而避免死锁发生的可能性。

根据这种思路，研究者们提出了若干不同的算法，它们在所要求的信息量和信息的类型上有所不同。最为简单也最常用的模型要求每个进程事先说明可能需要的每种资源类型实例的最大需求。根据每个进程可能申请的每种资源类型实例的最大需求的先验信息，可以构造一个算法以确保系统决不会进入死锁状态。这种算法就是死锁避免(deadlock-avoidance)方法。死锁避免算法动态地检测资源分配状态以确保循环等待条件不可能成立。资源分配状态是由可用资源和已分配资源，及进程最大需求所决定的。下面来研究死锁避免算法。

1. 安全状态 (safe state)

如果系统能按某个顺序为每个进程分配资源（不超过其最大值）并能避免死锁，那么系统状态就是安全的。更为准确地说，如果存在一个安全序列，那么系统处于“安全状态”。

那么什么是“安全序列”呢？假设有进程序列<P1，P2，…，Pn>，如果对于每个 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，那么这一顺序称为安全序列。在这种情况下，进程 Pi 所需要的资源即使不能立即可用，那么只要等待直到所有 Pj 释放其资源。当它们完成时，Pi 可得到其所需要的所有资源，完成其给定任务，然后释放其所分配的资源并终止。当 Pi 终止时，Pi+l 可得到其所需要的资源，以此类推，直至整个序列中的所有进程都顺利完成。如果没有这样的安全序列存在，那么系统状态就处于“不安全状态”。

安全状态肯定不是死锁状态，而死锁状态肯定是不安全状态。然而，不是所有不安全状态都必然会导致死锁状态（见图 4.4）。不安全状态可能导致死锁，也可能没有发生死锁。只要状态为安全，操作系统就能避免死锁状态。在不安全状态下，操作系统不能阻止进程以会导致死锁的方式来申请资源，进程的行为控制了是否真的发生死锁。

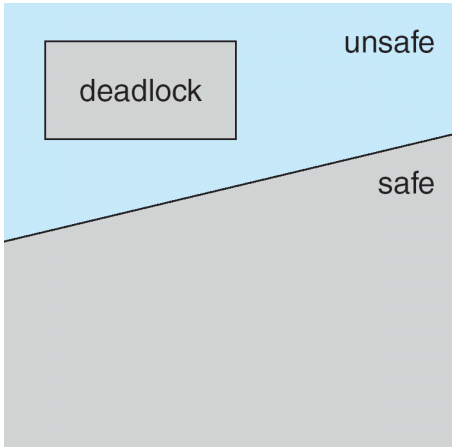


图 4.4 安全状态、不安全状态和死锁状态

例如，考虑一个系统，有 12 台磁带驱动器和三个进程 P0、Pl、P2。进程 P0 最多要求 10 台磁带驱动器，Pl 最多要求 4 台磁带驱动器，P2 最多要求 9 台磁带驱动器。假定，在时问 t0 时，进程 P0 占有 5 台磁带驱动器，进程 P1 占有 2 台磁帝驱动器，进程 P2 占有 2 台磁带驱动器。因此，系统中还有 3 台空闲磁带驱动器。

	最大需求	当前需求
P0	10	5
Pl	4	2
P2	9	2

在时刻 t0，系统处于安全状态。顺序<P1，P0，P2>满足安全条件，这是因为进程 Pl 可立即得到其所有磁带驱动器，完成工作并接着归还它们（系统会有 5 台磁带驱动器），接着进程 P0 可得到其所有磁带驱动器并归还它们（这时系统会有 10 台磁带驱动器），最后进程 P2 得到其所有磁带驱动器并归还它们（系统会有 12 台磁带驱动器）。

系统可能从安全状态转换为不安全状态。假定在时刻 t1，进程 P2 申请并又得到了 l 台磁带驱动器，系统就不再安全了。这时，只有进程 Pl 可得到其所有磁带驱动器。当其返回这些资源时，系统只有 4 台磁带驱动器可用。由于进程 P0 已分配了 5 台磁带驱动器而其最大需求为 10 台磁带驱动器，所以它还需要 5 台磁带驱动器。因为现在不够，所以进程 P0 必须等待。类似地，进程 P2 还需要 6 台磁带驱动器，也必须等待，这样就导致了死锁。这时的错误在于允许进程 P2 再获得 l 台磁带驱动器。如果让 P2 等待直到其他进程之一完成并释放其资源，那么就能避免死锁。

有了安全状态的概念，可定义死锁避免算法以确保系统不会死锁。其核心思想是简单地确保系统始终处于安全状态。开始，系统处于安全状态。当进程申请一个可用的资源时，系统必须确定这一资源申请是可以立即分配还是要等待。只有分配后使系统仍处于安全状态，才批准该申请。

采用这种方案，即使进程申请一个现已可用的资源，它仍然可能等待。因此，与没有采用死锁避免算法的系统相比，资源使用率可能变得更低。

2. 银行家算法 （banker’s algorithm）

对于每种资源类型都是单实例资源的情况，可以采用资源分配图算法来解决。但对于有多个实例的资源分配系统，资源分配图算法就不适用了。下面所讨论的死锁避免算法适用于这种多实例系统，但是其效率要比资源分配图方案低。这一算法通常被称为“银行家算法”。该算法如此命名是因为它常被用于银行系统。当银行家不能保证自己的资金安全时，它不会批准客户的贷款请求。

当一个进程进入系统时，它首先必须声明其可能需要的每种类型资源实例的最大数量，这一数量当然不能超过系统资源的总和。当用户申请一组资源时，系统必须确定这些资源的分配是否仍会使系统处于安全状态。如果是，就可分配资源；否则，进程必须等待直到某个其他进程释放足够资源（以便使得该进程的请求被批准之后系统仍然是安全的）为止。

为了实现银行家算法，必须先引入几个数据结构。这些数据结构对资源分配系统的状态进行了记录。设 n 为系统进程的个数，m 为资源类型的种类。我们定义如下数据结构：

·Availabe： 长度为 m 的向量，表示每种资源的现有实例的数量。如果 Availabe[j]=k, 那么资源类型 Rj 现有 k 个实例。

·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 k 个资源类型 Rj 的实例。

·Allocation： n×m 矩阵，定义每个进程现在所分配的各种资源类型的实例数量。如果 Allocation[i][j]=k，那么进程 Pi 已经分配了 k 个资源类型 Rj 的实例。

·Need： n×m 矩阵，表示每个进程还需要的剩余的资源。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，显然有 Need[i][j] = Max[i][j] – Allocation[i][j]。

这些数据结构的大小和值会随着时间而改变。

为了简化银行家算法的描述，先引入一些符号。设 X 和 Y 为长度为 n 的向量，则 X≤Y 当且仅当对所有 i=1，2，…，n，X[i]≤Y[i]。例如，如果 X=(l，7，3，2)而 Y=（0，3，2，1），那么 Y≤X。如果 Y≤X 且 Y ≠ X，那么 Y < X。

可以将矩阵 Allocation 和 Need 的每行看作向量，并分别用 Allocation[i]和 Need[i]来表示。向量 Allocation[i]表示分配给进程 Pi 的资源：向量 Need[i]表示进程 Pi 为完成其任务可能仍然需要申请的额外资源。

(1)．安全状态判定算法

判定计算机系统是否处于安全状态的算法分为如下几步：

①设 Work 和 Finish 分别为长度为 m 和 n 的向量。按如下方式进行初始化：Work=Available 且对于 i=0,l，…，n-l, 设置 Finish[i]=false。

②查找这样的 i 使其满足：Finish[i]=false 且 Need[i]≤Work

如果没有这样的 i 存在，那么就转到第④步。

③ Work = Work+Allocation[i]

Finish[i] = true

返回到第②步。

④如果对所有 i，Finish[i]=true，那么系统处于安全状态。

这个算法可能需要 $m \times n^2$ 数量级的操作以确定系统状态是否安全。

(2)．资源请求批准算法

有了上述安全状态判定算法，我们描述当一个进程提出资源请求时，系统如何判断是否可安全地批准该请求的算法。

设 Request 为进程 Pi 的请求向量。如果 Request[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 提出资源请求时，系统采取如下动作：

①如果 Request ≤Need[i]，那么转到第②步。否则，产生出错条件，因为进程 Pi 已超过了其最大请求。

②如果 Request ≤Available，那么转到第③步。否则，Pi 必须等待，因为没有足够的可用资源。

③假装系统批准了进程 Pi 所请求的资源，并按如下方式修改当前状态：

Available =Available – Request

Allocation[i] = Allocation[i] + Request

Need[i] = Need[i] – Request

使用上述安全状态判定算法，如果所产生的资源分配状态是安全的，那么就真正批准进程 Pi 提出的请求，使 Pi 分配到其所需要资源。否则（即如果新状态是不安全的），进程 Pi 必须等待，且系统保持原来资源分配状态不变。

(3)．实例

最后，为了说明银行家算法的使用，考虑这样一个系统，有 5 个进程 P0～P4，3 种资源类型 A、B、C。资源类型 A 有 10 个实例，资源类型 B 有 5 个实例，资源类型 C 有 7 个实例。假定在时刻 T0，系统状态如下：

	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	

P2	3 0 2	9 0 2
P3	2 1 1	2 2 2
P4	0 0 2	4 3 3

很容易算得矩阵 Need = Max – Allocation：

	Need
	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

可以判定系统现在处于安全状态。事实上，序列 < Pl，P3，P4，P2，P0> 满足安全条件。现在假定进程 P1 再请求 1 个资源类型 A 和 2 个资源类型 C，这样 Request = (1，0，2)。为了确定这个请求是否可以立即批准，首先检测 Request ≤ Need[1]，即 (1，0，2) ≤ (1，2，2)，通过；然后检测 Request ≤ Available，即 (1，0，2) ≤ (3，3，2) ，通过；最后，假定这个请求被满足，那么产生如下新状态：

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	**2 3 0**	7 4 3
P1	**3 0 2**	3 2 2		**0 2 0**
P2	3 0 2	9 0 2		6 0 0
P3	2 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

必须判定这个状态是否安全。为此，执行安全算法，并找到顺序 < Pl，P3，P4，P0，P2> 满足安全要求。因此，可以立即批准进程 Pl 的这个请求。

在此之后，P4 又提出了请求（3，3，0）。可以发现，我们是不能批准这个请求的，因为没有这么多资源可用。

接着，P0 又提出请求（0，2，0）。可以发现，我们也不能批准 P0 的这个请求。虽然有资源可用，但是这会导致系统进入不安全状态。

4.4 死锁检测和恢复

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，那么就有可能会出现死锁。在这种环境下，系统应提供：

. 一个用来检查系统状态从而确定是否出现了死锁的算法。

. 一个用来从死锁状态中恢复的算法。

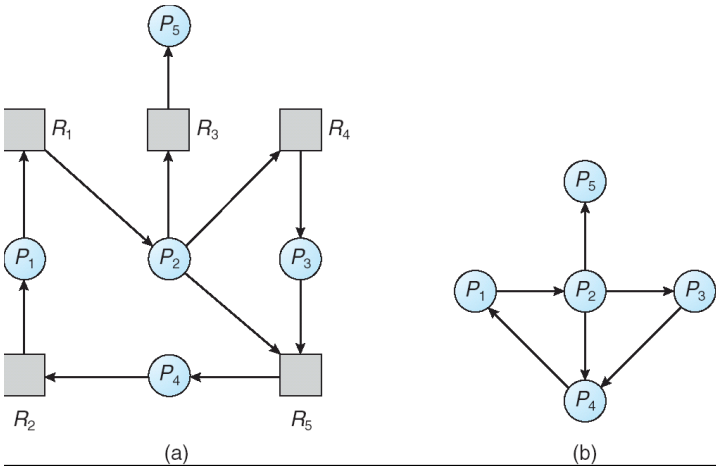
在以下讨论中，将针对每种资源类型只有单个实例和每种资源类型可有多个实例这两种情况，并分别讨论两个算法。不过，我们需要意识到检测并恢复方案会有额外开销，这些不仅包括维护所

需信息和执行检测算法的运行开销，而且也包括死锁恢复所引起的损失。

1. 每种资源类型只有单个实例

如果所有资源类型只有单个实例，那么可以定义这样一个死锁检测算法，该算法使用了资源分配图的一个变种，称为等待图(wait-for graph)。从资源分配图中，删除所有资源类型节点，合并适当的边，就可以得到等待图。

更确切地说，等待图中的由 Pi 到 Pj 的边意味着进程 Pi 等待进程 Pj 释放一个 Pi 所需的资源。等待图有一条 Pi→Pj 的边，当且仅当相应的资源分配图中包含两条边 Pi→Rq 和 Rq→Pj，其中 Rq 为资源。例如，图 4.7 显示了资源分配图和对应的等待图。



(a) 资源分配图　　(b) 对应的等待图

图 4.7　资源分配图和对应的等待图

与以前一样，当且仅当等待图中有一个环，系统中存在死锁。为了检测死锁，系统需要维护等待图，并周期性地调用在图中进行搜索的算法。从图中检测环的算法需要 n^2 级别的操作，其中 n 为图中的节点数。

2. 每种资源类型可有多个实例

等待图方案并不适用于每种资源类型可有多个实例的资源分配系统。下面所描述的死锁检测算法适用于这样的系统。该算法使用了一些随时间而变化的数据结构，与银行家算法相似。

·Awilable： 长度为 m 的向量，表示各种资源的可用实例。

·Allocation： n×m 矩阵，表示当前各进程的资源分配情况。

·Request： n×m 矩阵，表示当前各进程的资源请求情况。如果 Request[i][j]=k，那么 Pi 现在正在请求 k 个资源 Rj。

两个矢量之间的小于等于关系与第 4 节所定义的一样。为了简化起见，将 Allocation 和 Request 的行作为向量，且分别称为 Allocation[i]和 Request[i]。这里所描述的检测算法为需要完成的所有进程研究各种可能的分配序列。你可以将本算法与银行家算法做一比较。

① 设 Work 和 Finish 分别为长度为 m 和 n 的矢量。初始化 Work = Available。对于 i = 0，1，…，n-1，如果 Allocation[i] 不为 0，则 Finish[i]=false；否则，Finish[i]=true。

② 寻找这样的 i，以便同时使 Finish[i] = false 且 Request[i] ≤ Work 成立。如果没有这样的 i，则转到第④步。

③ 执行 Work = Work + Allocation[i]；Finish[i] = true；转到第②步。

④ 如果对某个 i（0≤i<n），Finish[i]=false，则系统处于死锁状态。而且，如果 Finish[i]=false，则进程 Pi 死锁。

该算法需要 $m \times n^2$ 级的操作来检测系统是否处于死锁状态。

你可能不明白为什么只要确定 Request[i] ≤Work（第②步中），就收回了进程 Pi 的资源（第③步）。已知 Pi 现在不参与死锁(因 Request[i]≤Work)，因此，可以乐观地认为 Pi 不再需要更多资源以完成其任务，它会返回其现已分配的所有资源。如果假定的不正确，那么稍后或许会发生死锁。下次调用死锁算法时，就会检测到死锁状态。

为了举例说明这一算法，考虑这样一个系统，它有 5 个进程 P0～P4 和 3 个资源类型 A、B、C。资源类型 A 有 7 个实例，资源类型 B 有 2 个实例，资源类型 C 有 6 个实倒。假定在时刻 To，有如下资源分配状态：

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

上图中的系统现在不处于死锁状态。事实上，如果执行检测算法，会找到这样一个序列 < P0，P2，P3，P1，P4> 会导致对所有 i，Finish[i] = true。

现在假定进程 P2 又请求了资源类型 C 的一个实例。这样，Request 矩阵修改成如下情形：

	Request
	A B C
*P*0	0 0 0
*P*1	2 0 1
*P*2	0 0 **1**
*P*3	1 0 0
*P*4	0 0 2

现在可以发现系统是死锁的。因为虽然可收回进程 P0 所占有的资源，但是接下去现有资源并不足以满足其他进程的请求。因此，会存在一个包含进程 Pl，P2，P3 和 P4 的死锁。

3. 死锁检测算法的应用

那么应该何时调用死锁检测算法？答案取决于两个因素。

·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?

如果经常发生死锁，那么就应经常调用检测算法。否则，直到死锁被打破前，分配给死锁进程的资源都会一直被占用。同时，进入死锁循环的进程数量也可能会不断增加。

我们注意到只有当某个进程提出请求且得不到满足时，才会出现死锁。这一请求可能是完成循环等待进程链的最后一个请求。在极端情况下．每当资源请求分配不能立即被批准时，就可以调用死锁检测算法。在这种情况下，不仅能确定是哪些进程进入死锁状态，而且可以确定是哪个特定的进程“造成”了死锁（尽管实际上，每个死锁进程都是资源图上的环的一个节点，因此，其实是所有进程一起造成了死锁）。如果有许多不同资源类型，那么一个请求可能造成资源图上的多个环，每个环由最近请求所完成，并且由提出这个请求的进程所“造成”。当然，对于每个请求都调用死锁检测算法会引入相当大的计算开销。

另一个代价较低的方法是以一个不太高的频率调用检测算法，如定时运行（如每小时一次），或者当 CPU 使用率低于一个特定的阈值时（如 40%）。这是因为死锁最终会使系统性能下降，并造成 CPU 使用率下降。如果在不确定的时间点调用检测算法，那么资源图上可能会有许多环。通常不能确定死锁进程中是哪个或哪些进程“造成”了死锁。

4. 死锁的恢复

当死锁检测算法确定死锁已存在，那么可以采取多种措施。一种措施是通知操作员死锁已发生，以便操作人员人工处理死锁。另一种措施是让系统从死锁状态中自动恢复过来。

打破死锁有两个方法。一个方法是简单地终止一个或多个进程以打破循环等待。另一个方法是从一个或多个死锁进程那里抢占一个或多个资源。

(1)．进程终止

通过终止进程以取消死锁有两种方法。不管采用哪种方法，系统都会收回分配给被终止进程的所有资源。

·终止所有死锁的进程。这种方法显然终止了死锁循环，但其代价很大。这些进程可能已计算了很长时间，这些计算结果都被废弃了，以后可能还要重新计算。

·一次只终止一个进程直到取消死锁循环为止。这种方法的开销也会相当大，这是因为每次终止一个进程，都必须调用死锁检测算法以确定系统是否仍处于死锁。

终止一个进程并不容易。如果进程正在更新文件，那么终止它会使文件处于不一致状态。类似地，如果进程正在打印文件，那么系统必须将打印机重新设置到正确状态，以便打印下一个文件。

如果采用后一种终止部分进程的方法，那么必须确定终止哪个进程或哪些进程可以打破死锁。这种决策类似于 CPU 调度问题，是个策略选择。该问题从根本上说是个经济问题，即应该终止代价最小的进程。然而，“代价最小”并不是精确的，有许多因素都影响着应选择哪个进程，例如：

① 进程的优先级是什么?

② 进程已计算了多久，进程在完成指定任务之前还需要多长时间?

③ 进程使用了多少什么类型的资源（例如，这些资源是否容易抢占）?

④ 进程还需要多少资源才能完成?

⑤ 多少进程需要被终止?

⑥ 进程是交互的还是批处理的？等等

(2)．资源抢占

通过抢占资源以取消死锁。逐步从死锁的进程那里抢占资源给其他进程使用，直到死锁环被打破为止。

如果要求使用资源抢占方法来处理死锁，那么有三个问题需要处理：

① 选择一个牺牲者：抢占哪些资源和哪个进程？与进程取消一样，必须确定抢占的顺序以使代价最小化。代价包括许多因素，如死锁进程所拥有的资源数量，死锁进程到现在为止在其执行过程

中所消耗的时间等。

② 回滚：如果从一个进程那里抢占一个资源，那么应对该进程做些什么安排？显然，该进程不能正常执行，它缺少所需要的资源。必须将进程回滚到先前的某个安全状态，以便从该状态重启进程。通常确定一个安全状态并不容易，所以最简单的方法是完全回滚：即终止进程并重新执行。更为有效的方法是将进程回滚到足够打破死锁的那个点。另一方面，这种方法要求系统维护有关运行进程状态的更多信息。

③ 饥饿问题：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？如果一个系统是基于代价来选择牺牲进程，那么同一进程可能总是被选为牺牲品。结果，这个进程永远不能完成其指定任务，任何实际系统都需要处理这种饥饿情况。显然，必须确保一个进程只能有限地被选择为牺牲品。最为常用的方法是在代价因素中加入回滚次数。

总之，这都是一些复杂的问题。