上述的执行次序不会引起死锁（因为没有资源的竞争），但程序也没有任何并行性。进程在执行过程中，不仅要请求和释放资源，还要做计算或者输入/输出工作。如果进程是串行运行，不会出现当一个进程等待I/O时让另一个进程占用CPU进行计算的情形。因此，严格的串行操作有可能不是最优的。不过，如果所有的进程都不执行I/O操作，那么最短作业优先调度会比轮转调度优越，所以在这种情况下，串行运行有可能是最优的。

如果假设进程操作包含I/O和计算，那么轮转法是一种合适的调度算法。对资源请求的次序可能会如图6-4d所示。假如按这个次序执行，图6-4e～图6-4j是相应的资源分配图。在出现请求4后，如图6-4h所示，进程A被阻塞等待S，后续两步中的B和C也会被阻塞，结果如图6-4j所示，产生环路并导致死锁。

不过正如前面所讨论的，并没有规定操作系统要按照某一特定的次序来运行这些进程。特别地，对于一个有可能引起死锁的资源请求，操作系统可以干脆不批准请求，并把该进程挂起（即不参与调度）一直到处于安全状态为止。在图6-4中，假设操作系统知道有引起死锁的可能，那么它可以不把资源S分配给B，这样B被挂起。假如只运行进程A和C，那么资源请求和释放的过程会如图6-4k所示，而不是如图6-4d所示。这一过程的资源分配图在图6-4l～图6-4q中给出，其中没有死锁产生。

在第q步执行完后，就可以把资源S分配给B了，因为A已经完成，而且C获得了它所需要的所有资源。尽管B会因为请求T而等待，但是不会引起死锁，B只需要等待C结束。

在本章后面我们将考察一个具体的算法，用以做出不会引起死锁的资源分配决策。在这里需要说明的是，资源分配图可以用作一种分析工具，考察对一给定的请求/释放的序列是否会引起死锁。只需要按照请求和释放的次序一步步进行，每一步之后都检查图中是否包括了环路。如果有环路，那么就有死锁；反之，则没有死锁。在我们的例子中，虽然只和同一类资源有关，而且只包含一个实例，但是上面的原理完全可以推广到有多种资源并含有若干个实例的情况中去（Holt, 1972）。

总而言之，有四种处理死锁的策略：

1) 忽略该问题。也许如果你忽略它，它也会忽略你。

2) 检测死锁并恢复。让死锁发生，检测它们是否发生，一旦发生死锁，采取行动解决问题。

3) 仔细对资源进行分配，动态地避免死锁。

4) 通过破坏引起死锁的四个必要条件之一，防止死锁的产生。

下面四节将分别讨论这四种方法。

6.3 鸵鸟算法

最简单的解决方法是鸵鸟算法：把头埋到沙子里，假装根本没有问题发生[⊖]。每个人对该方法的看法都不相同。数学家认为这种方法根本不能接受，不论代价有多大，都要彻底防止死锁的产生；工程师们想要了解死锁发生的频度、系统因各种原因崩溃的发生次数以及死锁的严重性。如果死锁平均每5年发生一次，而每个月系统都会因硬件故障、编译器错误或者操作系统故障而崩溃一次，那么大多数的工程师不会以性能损失和可用性的代价去防止死锁。

为了能够让这一对比更具体，考虑如下情况的一个操作系统：当一个open系统调用因物理设备（例如CD-ROM驱动程序或者打印机）忙而不能得到响应的时候，操作系统会阻塞调用该系统调用的进程。通常是由设备驱动来决定在这种情况下应该采取何种措施。显然，阻塞或者返回一个错误代码是两种选择。如果一个进程成功地打开了CD-ROM驱动器，而另一个进程成功地打开了打印机，这时每个进程都会试图去打开另外一个设备，然后系统会阻塞这种尝试，从而发生死锁。现有系统很少能够检测到这种死锁。

6.4 死锁检测和死锁恢复

第二种技术是死锁检测和恢复。在使用这种技术时，系统并不试图阻止死锁的产生，而是允许死锁发生，当检测到死锁发生后，采取措施进行恢复。本节我们将考察检测死锁的几种方法以及恢复死锁的几种方法。

⊖ 这一民间传说毫无道理。鸵鸟每小时跑60公里，为了得到一顿丰盛的晚餐，它一脚的力量足以踢死一头狮子。