第3章 死　　锁

死锁是操作系统中的一个理论研究问题，它是由于进程之间对资源的竞争访问所引发的相互等待、相互妨碍的现象。本章主要讨论四个方面的内容。首先介绍死锁的基本概念，什么是死锁，引发死锁的原因是什么。然后讨论当死锁发生以后，如何检测死锁的存在并且解除死锁。接下来讨论死锁的避免，即能否设计一个好的资源分配算法，从而从源头上避免死锁的发生。最后是死锁的预防，即通过破坏死锁产生的必要条件来防止死锁的出现。

在具体讨论之前，先来看一个关于死锁的小笑话。

一个人去一家公司面试，以下是他与面试官之间的对话。

面试官：请给我们解释一下什么叫死锁，如果解释清楚了我们就会录用你。

应聘者：如果你们录用了我，我就会给你们解释什么叫死锁。

3.1 死锁概述


扫码观看

3.1.1 什么是死锁

如图3.1所示为死锁的一个例子。在一个城市中，有一座桥梁，桥很窄，每次只能容许一辆汽车通过。现在有两辆汽车在桥上迎面碰上了，谁也无法再往前开。

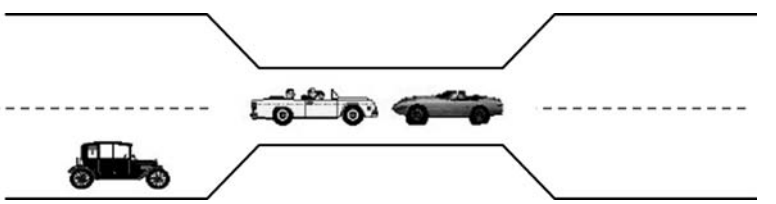

图3.1　过桥问题

另一个死锁的例子是前面介绍过的生产者和消费者问题。如果把该问题的代码稍微修改一下，如下所示，即把生产者函数中的两条语句调换一下顺序，那么其结果可能就是死锁。

```
void producer(void)
{
    int item;
    while(TRUE)
    {
        item = produce_item( );
        P(Mutex);
        P(BufferNum);
        insert_item(item);
        V(Mutex);
        V(ProductNum);
    }
}
```

生产者进程

```
void consumer(void)
{
    int item;
    while(TRUE)
    {
        P(ProductNum);
        P(Mutex);
        item = remove_item( );
        V(Mutex);
        V(BufferNum);
        consume_item(item);
    }
}
```

消费者进程

例如，假设当进程运行到某个时刻，缓冲区中已经装满了产品。这时，生产者进程运行到第一个 P 原语的地方，并且顺利地进入了临界区，此时 Mutex 信号量的值变成 0。然后当它再去调用第二个 P 原语时，由于此时缓冲区已经满了，因此信号量 BufferNum 的值等于 0，因此它就在这里被阻塞了。另一方面，当消费者进程在运行时，首先碰到第一个 P 原语，即 P(ProductNum)，由于当前缓冲区是满的，即 ProductNum 信号量的值大于 0，因此它就顺利地通过了这个 P 原语。但是当它执行第二个 P 原语即 P(Mutex)时，由于信号量 Mutex 的值已经等于 0，因此，消费者进程就在这里被阻塞了。也就是说，现在两个进程都被阻塞住了，都没有办法运行，这就形成了一种死锁的状态。由此可见，在基于信号量的进程间同步互斥问题中，稍微一个小的错误，都有可能导致严重的后果。

在上面两个例子中，都出现了事情无法进展下去的情形，这种情形称为“死锁”。死锁现象既可以出现在现实生活中，也可以出现在计算机科学的不同领域，例如，在操作系统中，多个进程对各种输入/输出设备的争夺所引起的死锁；在一个数据库系统中，多个进程对不同数据记录的互斥访问所引起的死锁；等等。也就是说，死锁既可能发生在硬件资源上，也可能发生在软件资源上。因此，为了对死锁问题进行更抽象、更具有普遍性的讨论，使之适用于各式各样不同的领域背景，我们把引发死锁的各种 I/O 设备、数据记录和共享文件等对象统称为资源(Resource)。

在一组进程当中，每一个进程都占用着若干个资源，同时它又在等待另外一个进程所占用的其他资源，从而造成的所有进程都无法进展下去的现象，这种现象称为死锁，这一组相关的进程就称为死锁进程。在死锁状态下，每一个进程都动弹不得，既无法运行，也无法释放所占用的资源，它们互为因果、互相等待，无穷无尽。

例如，在如图 3.1 所示的过桥问题中，可以把桥梁一分为二，即桥梁的左侧和右侧，每一侧都可以看成是一个资源。如果一辆汽车想要过桥，那么它必须同时拥有这两个资源。而现在的情形是有两辆汽车，其中，每一辆汽车各自占用了一个资源，同时又在等待对方释放另一个资源。

3.1.2 资源

如前所述,对资源的竞争访问是产生死锁的根本原因。在计算机系统中,有各种不同类型的资源,如CPU、时钟、各种输入/输出设备、内存空间、数据库当中的记录等。对于某些类型的资源来说,它们可能会有多个相同的实例。例如,在系统中有三个磁带驱动器,那么当用户在申请该资源时,这三个磁带驱动器中的任何一个都能满足要求,没有区别。当然,对于任何一个具体的资源来说,在任何时刻只能被一个进程所使用。

资源可以分为两类:可抢占的和不可抢占的。

- 可抢占的资源:当一个进程正在使用这种类型的资源时,可以把它拿走而不会对该进程造成任何不良的影响,例如,内存和CPU。在大多数计算机上,CPU的个数是有限的,如只有一个。因此从微观上来说,在任何时候,最多只能有一个进程在CPU上运行。但是系统可以利用进程切换机制,使得在宏观上可以有多个进程同时在内存中运行。当一个进程在CPU上运行一段时间后,系统会把它的硬件上下文保存到内存中该进程的PCB中,然后再切换到另一个进程去运行。因此CPU这个资源是可以抢占的。
- 不可抢占的资源:当一个进程正在使用这种类型的资源时,如果强行把它拿走,将会导致该进程运行失败,例如,光盘刻录机。

死锁主要是由不可抢占资源所引起的,对于可抢占的资源,可以通过重新分配资源的方法来避免死锁。因此,在讨论死锁问题时,主要考察的是不可抢占的资源。另外,进程在使用一个资源的时候,一般要经过三个步骤,即申请资源、使用资源和释放资源。如果申请不成功,则该进程会被阻塞,进入相应的等待队列。

3.1.3 死锁的模型

什么时候才会出现死锁现象呢?1971年,Coffman提出了死锁发生的4个条件,只有当这4个条件同时成立时,才会出现死锁。

- 互斥条件:在任何时刻,每一个资源最多只能被一个进程所使用,即对任何一个资源的访问都必须互斥地进行。
- 请求和保持条件:进程在占用若干个资源的同时又可以去请求新的资源。
- 不可抢占条件:进程已经占用的资源,不会被强制性拿走,而必须由这个进程主动释放。事实上,如果资源是可抢占的,那么可以通过重新分配资源的方法来避免死锁。
- 环路等待条件:存在一条由两个或多个进程所组成的环路链,其中每一个进程都在等待环路链中下一个进程所占用的资源。显然,单个进程不可能产生死锁,因为最少要有两个进程才能形成这样的一个环路。

以上4个条件缺一不可。如果有一个条件不满足,就不会出现死锁现象。但是,这四个条件是必要而不充分条件。换句话说,如果死锁,必定满足这4个条件。反之,即使满足这4个条件,也不一定死锁。

1972年,Holt提出用资源分配图的方法来描述死锁发生的4个条件。如图3.2所示,在这种资源分配图中,他用两种类型的结点来表示进程和资源,然后用有向边来表示进程与

资源之间的请求和分配关系。具体来说，每个进程用一个圆圈表示，每个资源用一个方框表示。然后，如果一个进程 P 正占用一个资源 R，那么就从资源 R 引一条有向边指向进程 P。反之，如果一个进程 P 在请求资源 R 时没有成功，被阻塞了，那么就用一条有向边从进程 P 指向资源 R。

例如，对于如图 3.1 所示的过桥问题，如果把桥上的那两辆汽车分别称为 P1 和 P2，把桥梁的左侧和右侧分别看成是资源 R1 和 R2，那么可以画出相应的资源分配图，如图 3.3 所示。在该图中，进程 P1 占用了资源 R2，然后在等待资源 R1，进程 P2 占用了资源 R1，然后在等待资源 R2，这两个进程当前都处于阻塞状态，没法运行，也就没法释放自己所占用的资源。另外，图中出现一条环路，即 P1→R1→P2→R2→P1，这就意味着出现了死锁，这次死锁所涉及的进程是 P1 和 P2，所涉及的资源是 R1 和 R2。

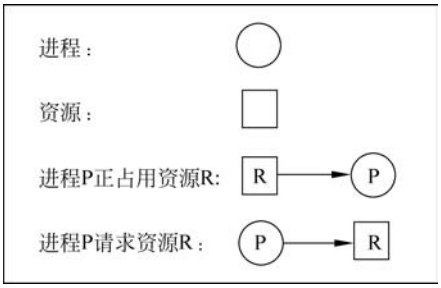

图 3.2　资源分配图描述死锁

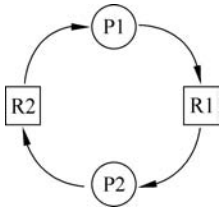

图 3.3　过桥问题的资源分配图

资源分配图的表示方法有一个问题。在图中，每一种资源类型都是用一个方框来表示，也就是说，它默认每一种类型的资源在系统中的个数只有一个。但实际上，对于有的资源类型来说，它在系统中的个数可能不止一个。例如，系统中可能有两个磁盘驱动器。因此，这种数量关系就没法体现出来。

为了体现这种数量关系，人们又对资源分配图进行了改进。如图 3.4 所示，进程的表示方法不变，还是一个圆圈。对于每一种资源类型，也还是用一个方框来表示，不过在方框里面会标上一些小圆点，表示这种类型的资源个数。如果一个进程 P 正占用 R 类型的某一个资源，就从资源 R 当中的这个小圆点出发，引一条有向边指向进程 P。反之，如果进程 P 想要申请 R 类型的资源，但没有成功，那么就用一条有向边，从进程 P 指向资源 R（即指向整个方框，而不是方框中的某个圆点），表示它已经被加入到资源 R 的等待队列中。

资源分配图描述的是在某个特定的时刻，系统的当前状态，它是在不断变化的。每当有进程去申请和释放资源的时候，资源分配图就可能会发生变化。

下面来看一个资源分配图的例子。假设有三个进程 P1、P2 和 P3，以及三种不同类型的资源 R1、R2 和 R3，其中，R1 有两个资源实例，R2 和 R3 都只有一个资源实例。已知这三个进程对资源的请求和释放顺序如下。

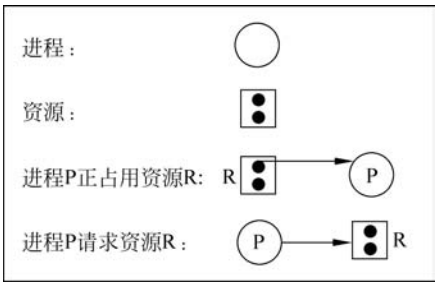

图 3.4　改进的资源分配图

```
进程 P1
请求资源 R1
请求资源 R2
释放资源 R1
释放资源 R2
```

```
进程 P2
请求资源 R1
请求资源 R2
请求资源 R3
释放资源 R1
释放资源 R2
释放资源 R3
```

```
进程 P3
请求资源 R3
释放资源 R3
```

要求分析在各种情形下进程与资源之间的分配关系。

首先需要指出,死锁是由于资源竞争而导致的,如果没有资源竞争,那么就不会出现死锁的问题。具体来说,假设系统采用的调度算法为先来先服务算法。那么在这种情形下,结论是不会发生死锁。因为先来先服务算法是不可抢占的,只有当一个进程运行结束或被阻塞时,才会去运行另一个进程。因此,它可以先运行 P1,然后再运行 P2,最后运行 P3,即按照顺序逐一去运行各个进程。这样,在各个进程之间,就没有对资源的竞争访问。例如,当 P1 运行时,申请了两个资源 R1 和 R2。当它运行完后,就释放了这两个资源。然后当进程 P2 和 P3 运行时,也是类似的。这样一来,相当于是在每个进程运行的时候,所有的资源都是可以使用的,因此就不会出现对资源的竞争访问。

如果系统采用的调度算法为时间片轮转法,那结果就不一定了。由于进程调度和时钟中断等原因,各个进程的执行顺序可能是不一样的。例如,假设在某一次执行过程中,各进程对资源请求的发生顺序如下:

(1) P1 请求资源 R1;

(2) P2 请求资源 R1;

(3) P3 请求资源 R3;

(4) P2 请求资源 R2;

(5) P1 请求资源 R2;

(6) P2 请求资源 R3。

在这种情形下,在这些进程的执行过程中,将会得到如图 3.5(a)所示的资源分配图。具体来说,当 P1 请求资源 R1 时,由于 R1 此时有两个实例,因此申请成功,P1 将会得到 R1。当 P2 请求 R1 时,也会申请成功,P2 也得到一个 R1,此时 R1 就没有空闲的资源实例了。当 P3 请求 R3 时,申请成功,P3 将得到 R3,然后 R3 就没有空闲的资源实例了。当 P2 申请 R2 时,也会成功。接下来,当 P1 申请 R2 时,由于 R2 已经没有空闲的资源实例了,因此申请失败,P1 将进入 R2 的等待队列。然后当 P2 申请 R3 时,也申请失败,进入 R3 的等待队列。总之,在当前时刻,从进程的角度,P1 占用了一个 R1 资源,并在等待 R2; P2 占用了一个 R1 资源和一个 R2 资源,并在等待 R3; P3 占用了一个 R3 资源。从资源的角度,R1、R2 和 R3 的所有资源都已经被占用。显然,在图 3.5(a)中,并没有出现环路,因此它不是一种死锁状态。事实上,虽然进程 P1 和 P2 都处于阻塞状态,但进程 P3 并没有被阻塞,它可以运行。当它运行结束后,就会释放 R3 资源,从而把进程 P2 唤醒。而当 P2 运行结束后,又会释放它所占用的所有资源,包括 R1、R2 和 R3。然后,随着 R2 被释放,进程 P1 又会被唤醒。这样,到最后时,所有进程都能顺利地运行完毕,不会发生死锁的现象。

假设进程 P3 在运行时,又增加了一个资源请求,去申请资源 R1。但由于此时 R1 的两

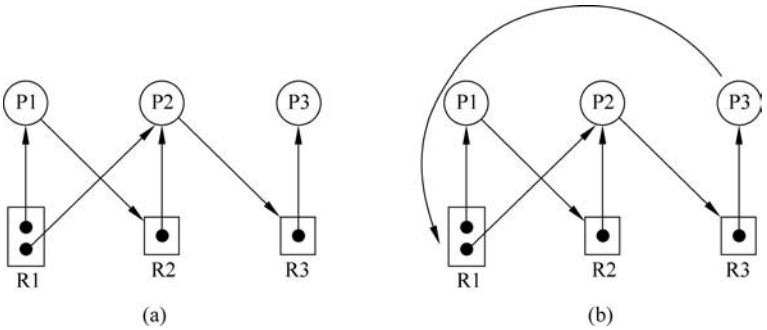

图 3.5　时间片轮转法的资源分配图

个资源实例已经分配出去了，因此申请失败，P3 将进入 R1 的等待队列，如图 3.5(b)所示，此时，在资源分配图中将存在两条环路：

P1→R2→P2→R3→P3→R1→P1

P2→R3→P3→R1→P2

另外，P1、P2 和 P3 这三个进程当前都处于阻塞状态，都无法运行，因此，这时就会出现死锁的现象。

既然在系统运行过程中，有可能会出现死锁的现象，那么如何来应对呢？一般来说，在应对死锁问题时，人们有以下四种策略可供选择。

(1)“无为而治”：故意忽略这个问题，假装在系统中不会出现死锁现象。这种方法虽然简单，但却是一种明智的解决方案。原因在于：虽然在死锁问题上，科学家们提出了各种理论上的研究成果。但如果要把这些理论成果应用到实际的系统中，将会大大增加系统的额外开销。另一方面，死锁现象也不会经常出现。因此，如果采用“鸵鸟”策略，不去管它，那么问题也不是很大。事实上，当前主流的操作系统，如 Windows 和 Linux，都采用了这种应对策略。

(2) 死锁的检测和解除：在系统中允许死锁发生，但是会通过不断地检测及时发现，然后采取相应的措施来解除死锁。

(3) 动态避免：通过精心设计的资源分配方案来避免死锁的发生。

(4) 死锁预防：破坏死锁产生的 4 个必要条件之一，使得死锁无法发生。

下面分别介绍这几种应对策略。

3.2　死锁的检测和解除

3.2.1　死锁检测算法

死锁的检测指的是判断系统中是否存在死锁。这里只考察单资源的情形，即每一种类型的资源只有一个。例如，在系统中只有一台扫描仪、一台光盘刻录机、一个绘图仪以及一个磁带驱动器等。

最简单的死锁检测方法是人工观察法，即对一个系统而言，首先构造它的资源分配图，然后人工去观察，看看在这个图中是否存在封闭的环路。如果有，说明系统中存在死锁，而

且这条环路上的所有进程都是死锁进程；如果没有，则说明系统中不存在死锁。当然，这种策略只适合于单资源的情形。在多资源的情形下，即便在资源分配图中存在环路，也不意味着一定会死锁。

人工观测法虽然简单，但如果要在一个实际的系统中检测，还是需要一个自动化的检测工具，即需要提出一个形式化的算法。这个算法的目标其实就是判断在一个有向图中，是否存在环路。学过数据结构课程的读者，对这种算法应该不会陌生。

以下是一个检测算法的具体实现，它的基本思路是依次把图中的每一个结点作为起始结点(树根结点)，然后利用“试探与回溯”策略进行深度优先搜索，并判断在这一轮搜索中是否存在环路。

数据结构：Nodes 表示该有向图中所有结点的集合，L 表示一个结点列表，flag 表示状态标记(初始化为 0，表示没有环路)。

```
for(N : Nodes) {                          //N 为 Nodes 中的某一个结点，以它作为起始结点
    把 L 初始化为空表，把图中所有的边置为未标记;
    t = N;                                //t 表示当前结点
    while(true) {
        把 t 加入列表 L 的末尾;
        if (t 在 L 中出现了两次) {
            flag = 1;                     //图中包含环路(并已列在 L 中)
            break;
        }
        else {
            从当前的结点出发，看是否还有未被标记的输出边;
            if (有未被标记的输出边) {
                按照顺序选择下一条未被标记的输出边，标记它;
                t = 下一个结点;           //顺着该输出边走到下一个结点
            }
            else if( t == N)              //如果走到死胡同，并且当前结点正是起始结点
                break;                    //没有找到任何环路，退出
            else {
                把 t 从 L 中删去;
                t = 上一个结点(父结点);          //把当前结点回溯到它的上一个结点
            }
        }
    }
    if(flag == 1) break;                  //找到环路，退出算法
}
```

3.2.2 死锁的解除

如果在一个操作系统中出现了死锁，那么将会有大量的系统资源被占用。例如，所有的死锁进程都待在内存中，又没有办法继续运行下去，这样就会造成系统资源的浪费。因此，一旦通过自动检测算法发现了死锁的存在，就应该尽快采取相应的措施来解除死锁，以最小的代价恢复系统的运行。具体来说，主要有三种方法：剥夺资源、进程回退和撤销进程。

第一种解除死锁的方法是剥夺资源，它的基本思路是：把一个资源从一个进程手里强行抢过来，然后交给另一个进程去使用。等它用完之后，再交还给原来的进程。这样，这两

个进程对该资源的请求就都能得到满足，只不过一个在先，一个在后。

不过，这种强行剥夺资源的做法需要付出代价，而这种代价的大小，或者说，对被剥夺资源的进程所造成的不良影响的大小，完全取决于这种资源自身的性质。实际上，如前所述，死锁问题主要是由不可抢占的资源所引起，而对于可抢占的资源，可以通过重新分配资源的方法来避免死锁。因此，与死锁有关的资源都属于不可抢占资源，如果硬要强行剥夺，必然会影响该进程的正常运行。当然，这种影响的程度和严重性可能各不相同。例如，如果这个资源是一台光盘刻录机，那么强行剥夺资源的结果可能是把一张光盘给刻坏了。但如果这个资源是一台打印机，那么结果可能只是重复地打印了几页纸的内容。总之，这种剥夺资源的方法，实现起来有点困难。如果非要这么做，只能选择那些相对而言比较容易剥夺的资源。

第二种解除死锁的方法是进程回退，它的基本思路是：定期地把每个进程的状态信息保存在文件中，这样就得到了一个文件序列，其中每个文件分别记载了这个进程在不同时刻的状态信息，包括它的内存映像以及它所占用的资源的状态。当系统检测到死锁的存在时，首先查明在这一次死锁中，有哪一些资源被涉及，然后把其中一个资源的拥有者，即某一个进程，回退到以前的某个时刻，即它还没有拥有该资源的时刻。这样，这个资源现在就可以被其他的进程使用了，从而打破了死锁的僵局。但是对于那个回退的进程而言，它需要做出一些牺牲，也就是说，从原来那个时刻到现在，这一段时间内的所有工作都丢失了。

这种进程回退的方法给进程造成的伤害比较小，但缺点是代价非常高。因为对于每一个进程，系统都要定期地对它的状态信息进行备份，这就增加了系统的时间和空间开销。当然，除去这个缺点以外，该方法不失为一种比较现实的方法。在一些实际的系统中，如数据库系统，为了保证安全，例如，为了防止在突然断电时所产生的数据异常、数据不一致等现象，在它的运行过程中，会不断地把当前的状态信息保存在一个日志文件中。这样，即使出现异常，也能很容易地恢复。

第三种解除死锁的方法是撤销进程，它的基本思路是：撤销一个或多个处于死锁状态的进程，把它们从系统中踢出去。具体来说，先撤销一个死锁进程，或者是一个虽然没有死锁但占用了资源的进程。这么做的目的是抢夺它手中的资源，然后把这些资源再分配出去。如果此时其他的死锁进程能够运行起来，就说明有效；否则，只好继续撤销其他的进程，并释放它们所占用的资源，直到死锁得以解除为止。

为了减少被撤销进程的伤害程度，应该尽可能选择那些能够安全地重新运行的进程，如编译进程。因为编译进程的任务很简单，就是读入一个源文件，然后生成相应的目标文件。如果它在运行过程中被撤销，那么没有太大的伤害，只需重新运行一遍即可。

总之，以上这三种解除死锁的方法，虽然能够打破死锁，但都存在着这样或那样的问题，所以对于实际的系统来说，它们都不很有吸引力。

扫码观看

3.3 死锁的避免

对于死锁的检测与解除，它的基本思路是允许死锁发生，然后一旦检测到死锁，就想办法去解除。我们知道有一句话叫“扬汤止沸，不如釜底抽薪”。那么对于死锁这个问题来说，与其让它发生以后再去想办法补救，还不如一开始就不让它发生。这就是一种更高明的策

略。但这种策略是否可行呢?

如前所述,死锁问题的本质在于系统资源的数量有限,由此所造成的各个进程之间的资源分配矛盾。如果系统中的资源个数非常多,想要多少就有多少,那么就不会出现死锁问题了。前面在讨论死锁的检测时,曾经默认进程在申请资源时,是一次性地全部申请。但在一个实际的系统中,并不是这样。一个进程每次只能申请一个资源,而且系统只有在确认了安全性之后,才会把这个资源分配给它。因此,一个想法就是,能否设计出一个好的资源分配算法,在分配资源时经过精心的安排,从而从源头上避免死锁的发生。具体来说,系统提前知道各个进程对所有资源的使用情况(即何时申请哪一个资源,何时释放哪一个资源),然后每当一个进程来申请一个资源时,系统先看一下能不能给它,如果给了它,会不会有死锁的危险,如果没有危险才分配给它。如果这个想法是可行的,那么对于死锁问题,就能做到釜底抽薪。

3.3.1 死锁避免举例

下面通过一个例子来阐述死锁避免问题。假设系统中有两个进程A和B,另外还有两种不同类型的资源R1和R2,每种类型的资源都只有一个。已知这两个进程对资源的请求和释放顺序如下:

进程A
A1: 请求资源R1
A2: 请求资源R2
A3: 释放资源R1
A4: 释放资源R2

进程B
B1: 请求资源R2
B2: 请求资源R1
B3: 释放资源R2
B4: 释放资源R1

假设系统采用的调度算法为时间片轮转法,那么由于进程调度和时钟中断等原因,各个进程的执行顺序可能是不一样的。下面分不同的情形来讨论。

首先,如果进程A先执行,并且运行到A3以后的位置,那么在这种情形下,肯定不会发生死锁的现象,也就不需要系统做出什么决策。因为此时进程A已经占用了全部的资源,而且将来也不再需要任何资源,因此它只要得到CPU,就能很顺畅地执行。而对于进程B来说,它在申请资源R2的时候肯定会失败,会被阻塞起来,从而让出CPU。后来,当进程A释放了资源R1和R2以后,进程B会被唤醒,并重新开始执行。此时进程B所需要的两个资源都是空闲的,因此,它也能很顺畅地执行完毕。

其次,如果进程B先运行,并且执行到B3以后的位置,那么在这种情形下,也不会发生死锁的现象,这与上一种情形是类似的。

再次,如果进程A先执行,在执行完A1即请求资源R1以后,被时钟中断所打断,然后进程B去运行,它要执行B1,即请求资源R2。那么在这种情形下,这就是一个关键时刻,系统的资源分配算法就必须做出一个决策,是否批准进程B的这次请求。如果系统答应了进程B的请求,把R2分配给它,那么其结果就是必然会死锁。因为在这种情形下,后面无非是两种情形。一是进程B继续执行,然后去执行B2,即请求资源R1。由于R1已经被进程A所占用,因此进程B将被阻塞起来,后来当进程A去申请资源R2时也会被阻塞起来;二是进程A又抢到CPU去执行,但是它在请求资源R2时,由于R2已经被进程B所占用,因

此进程 A 将被阻塞起来，后来当进程 B 去申请资源 R1 时也会被阻塞起来。总之，无论是哪一种情形，其结果都是一样的，两个进程都将被阻塞起来，都在等待对方释放资源，从而进入了死锁的状态。因此，如果是一个好的资源分配算法，那么当进程 B 在请求资源 R2 的时候，就会意识到这将导致不安全的状态，因此就不会批准这次请求。这样，进程 B 运行受阻，而进程 A 就可以顺畅地运行下去，先请求资源 R2，然后在用完之后释放 R1 和 R2。最后进程 B 被唤醒，也能顺畅地运行完成。因此，由于资源分配算法的这次正确决策，避免了一次死锁的发生。

最后，如果进程 B 先运行，在执行完 B1 即请求资源 R2 以后，被时钟中断所打断，然后进程 A 去运行，它在请求资源 R1 时，这也是一个关键时刻，资源分配算法也必须否决这次申请，否则就会导致死锁。

通过上述例子，我们了解了死锁避免问题，并且引入了两个概念，即安全的状态和不安全的状态。例如，在上述第三种情形下，如果系统同意了进程 B 的请求，把资源 R2 分配给它，那么系统将会进入一个不安全的状态，就可能会发生死锁。反之，如果系统没有把资源 R2 分配给它，而是让进程 A 继续往下执行，那么系统将会进入一个安全的状态，在这种情形下就不会发生死锁。因此，对于一个资源分配算法而言，如果它能判断出系统当前是处于安全状态还是不安全状态，它就能做出正确的选择，从而避免死锁。当然，这里的前提条件是系统必须提前知道每一个进程将会在何时去申请和释放哪一个资源，而在一个实际的系统中，这些信息可能无法提前获得。

另外，对于上述这个例子，这种理论阐述可能有点费解。实际上，它跟图 3.1 中的过桥问题是完全一样的。如果是过桥问题，那么就非常直观了。在过桥时，要么让左边的车先过，要么让右边的车先过，这样都没有问题。但如果两辆车同时上桥，即各自占用一半的资源，那么其结果必然是死锁。事实上，在马路交通方面，十字路口的红绿灯就是起到资源分配算法的作用，它先让东西方向的车流占用所有的路口资源，顺利通过，如果此时有南北方向的车来了，则必须等待，禁止占用路口资源。过了一会儿，再让南北方向的车流占用所有的路口资源，顺利通过，如果此时有东西方向的车来了，也必须等待，这样就不会出现大家各占一半资源从而死锁的情形。事实上，十字路口的堵车往往就是因为红绿灯失灵或者某些汽车驾驶员不遵守红绿灯的指示。

3.3.2 安全状态与不安全状态

如何判断系统当前是处于安全状态还是不安全状态呢？这需要解决两个问题，一个是数据结构，即如何表示系统的当前状态，或者确切地说，如何描述在当前这个时刻，系统当中的进程与资源之间的请求和分配关系。另外，还要了解安全状态的定义是什么。第二个问题是算法，即如何判断系统的当前状态是否安全。

先来看数据结构，即系统状态的表示方法。假设在系统中有 n 个进程（$P_1 \sim P_n$），资源类型的个数为 m。那么可以定义如下 4 个数据结构：

- 总的资源向量 $\boldsymbol{E}=(E_1,E_2,E_3,\cdots,E_m)$，其中，$E_i$ 表示系统中第 i 种类型的资源个数。例如，若第一种类型的资源为打印机，则 $E_1=2$ 表示系统中共有 2 台打印机。
- 空闲资源向量 $\boldsymbol{A}=(A_1,A_2,A_3,\cdots,A_m)$，其中，$A_i$ 表示第 i 种类型的资源中，尚未被占用的个数，即空闲的资源个数。例如，仍以刚才的打印机为例，若 $A_1=0$，表示

系统中的两台打印机,已经全部被占用了。

- 当前分配矩阵 $\boldsymbol{C}=(C_{ij})_{n\times m}$,该矩阵的第 i 行表示进程 P_i,第 j 列表示第 j 种类型的资源,C_{ij} 表示进程 P_i 所占用的类型为 j 的资源个数。
- 请求矩阵 $\boldsymbol{R}=(R_{ij})_{n\times m}$,其中,$R_{ij}$ 表示进程 P_i 还需要的类型为 j 的资源个数。

图 3.6 是这 4 个数据结构的图形表示。

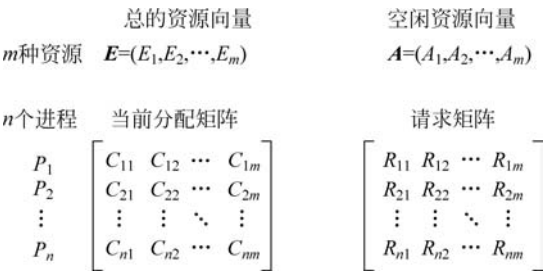

图 3.6　描述系统状态的 4 个数据结构

对于以上这 4 个数据结构,恒有下列等式成立:

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

它表示对于第 j 种类型的资源,E_j 为系统所拥有的总的资源个数,它可以分为两部分,一部分是已经分配给各个进程的资源总和,另一部分是系统当中还剩余的空闲资源个数。

图 3.7 是描述系统状态的 4 个数据结构的一个例子。从图 3.7 中可以看出,有 4 个进程,共享系统提供的 5 种类型的资源。

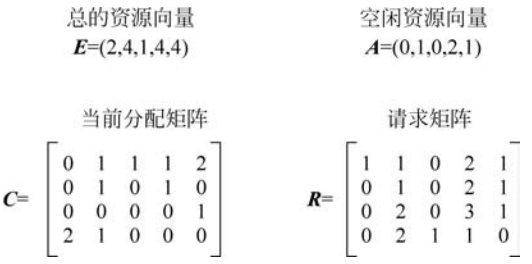

图 3.7　描述系统状态的数据结构的例子

有了数据结构以后,就可以来定义安全状态的概念。一个状态如果满足以下两个条件,被称为是“安全的”。

- 它自身不存在死锁问题。
- 存在某种调度顺序,使得即使在最坏的情况下(所有的进程突然间同时向操作系统请求它们最大数目的资源,即矩阵 $\boldsymbol{R}$ 中的数值),每一个进程都能顺利地运行结束。

下面通过一个简单的例子来说明这个概念的含义。假设在系统中只有一种类型的资源,其个数为 10。进程有 3 个,即进程 P_1、P_2、P_3。在这种情形下,总的资源向量 $\boldsymbol{E}$ 和空闲资源向量 $\boldsymbol{A}$ 就只有一个元素,而当前分配矩阵 $\boldsymbol{C}$ 和请求矩阵 $\boldsymbol{R}$ 则退化为 3 行 1 列。对于图 3.8(a),我们来判断一下,该状态是否安全。在图 3.8(a)中,在当前分配矩阵 $\boldsymbol{C}$ 中,三个进程各占用了 4、3 和 2 个资源,共 9 个资源,因此空闲资源向量 $\boldsymbol{A}$ 的个数为 1。在请求矩阵

$\boldsymbol{R}$ 中，三个进程将来还需要申请 5、1 和 4 个资源。

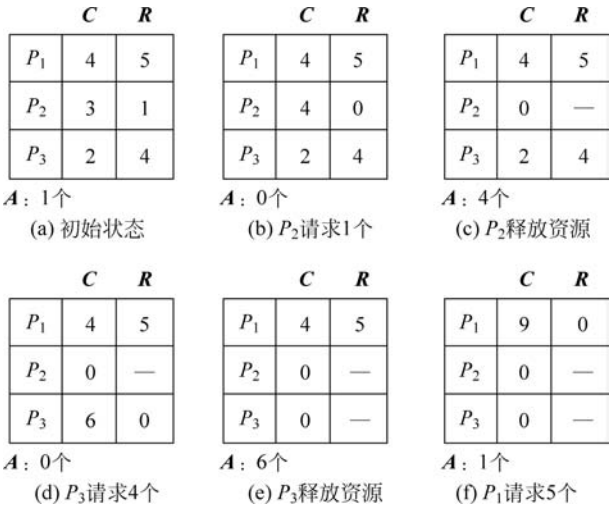

图 3.8　安全状态的一个例子

显然，图 3.8(a)的状态是安全的，因为即使在最坏情形下，也存在一个调度顺序，使所有的进程都能顺利地完成。具体来说，可以先调度进程 P_2 去运行，这样 P_2 会再去请求剩余的 1 个资源，而系统中正好还有 1 个空闲资源，所以分配成功。如图 3.8(b)所示，进程 P_2 将占用 4 个资源，而且以后不再需要新的资源，就能够顺利地运行下去，直到运行结束，然后把它所占用的 4 个资源全部释放，如图 3.8(c)所示，这样系统中就会有 4 个空闲资源。接下来，再去调度进程 P_3 运行，它会再去申请剩余的 4 个资源，而系统中正好有 4 个空闲资源，所以分配成功。如图 3.8(d)所示，进程 P_3 将占用 6 个资源，而且以后不再需要新的资源，就能顺利地运行下去。当它运行结束后，把它所占用的 6 个资源全部释放，如图 3.8(e)所示，这样系统中就会有 6 个空闲资源。最后，调度进程 P_1 去运行，它会再去申请剩余的 5 个资源，而系统能够满足它的要求，所以分配成功，如图 3.8(f)所示，因此 P_1 也能正常运行结束。通过以上过程可以看出，只要按照 P_2、P_3、P_1 的调度顺序去执行，那么所有的进程都能顺利地运行结束。这也就说明，最初的状态是安全的。

3.3.3　银行家算法

银行家算法是 1965 年由 Dijkstra 提出的一种避免死锁的调度算法，它模拟了一个银行家在发放信用贷款时的处理方式。具体来说，在一个小镇上，有一位银行家和一些需要贷款服务的客户。银行家会根据每一位客户的背景情况，为他设定相应的最高贷款限额。例如，对于那些信誉较好、还贷能力较强而且比较熟悉的老顾客，这个最高限额可能就比较大；而对于那些信誉不太好，或者说不太熟悉的新顾客，这个最高限额可能就比较小。现在的问题是：银行家必须设计出一种算法，以保证借贷过程的顺利进行。也就是说，当某个客户提出了一个贷款申请时，该算法必须去判断，如果批准了这个申请，是否会导致一种不安全的状态。如果是，那就拒绝这个申请；如果不是，那就批准这个申请。显然，在这个问题中，贷款就是前面所说的资源，而且只有这一种类型的资源，所以把它称为单一资源类型的死锁避免问题。

如图3.9所示为一个例子,假设这个银行家有4位客户,即A、B、C、D。每位客户都有一个最高贷款限额,分别为3000美元、5000美元、7000美元和9000美元。在最开始时,银行家手里有1万美元,如图3.9(a)所示。

	已贷	仍需	限额
A	0	3	3
B	0	5	5
C	0	7	7
D	0	9	9

银行家:10k

(a) 安全状态

	已贷	仍需	限额
A	1	2	3
B	2	3	5
C	2	5	7
D	3	6	9

银行家:2k

(b) 安全状态

	已贷	仍需	限额
A	1	2	3
B	2	3	5
C	2	5	7
D	4	5	9

银行家:1k

(c) 不安全状态

图3.9 单一资源类型的例子

在图3.9中,A、B、C、D这4个客户相当于4个进程,银行家的1万美元相当于总的资源向量$\boldsymbol{E}$,银行家手里剩余的金额相当于空闲资源向量$\boldsymbol{A}$,由于在系统中只有一种类型的资源,因此$\boldsymbol{E}$和$\boldsymbol{A}$就退化为一个变量。4位客户的已贷金额相当于当前分配矩阵$\boldsymbol{C}$,而仍需贷款的金额相当于请求矩阵$\boldsymbol{R}$,这两个矩阵退化为4行1列。对于每一位客户,他的最高贷款限额是固定不变的,并且等于已贷金额与仍需贷款金额之和。

根据安全状态的定义,可以证明如图3.9(a)所示的初始状态是安全的。因为此时银行家手里有1万美元,而对于4位客户,仍需贷款金额最大的是D,他还需要9000美元,因此,任意的调度顺序都是可行的,都能保证所有客户顺利完成任务。

假设这个借贷过程进行到某个时刻,出现了图3.9(b)的状态。即A已经贷了1000美元,B贷了2000美元,C贷了2000美元,D贷了3000美元,而银行家手里只剩下2000美元。同样可以证明,这个状态也是安全的。例如,只要采用A、B、C、D的调度顺序,就能保证所有客户都能顺利完成任务。

假设在图3.9(b)这个状态下,客户D又要申请1000美元的贷款,那么能不能批准呢?如果批准,那么系统的状态就会变成图3.9(c),此时银行家手里只剩下1000美元。那么这就是一个不安全的状态,因为假设这4个客户同时来申请它们剩余的最大贷款数额,那么银行家就没有办法满足他们当中的任何一个人。这就意味着,在刚才客户D来申请1000美元时,银行家就不能批准该请求。当然,所谓的不安全状态,并不是说一定会发生死锁,而只是说存在死锁的风险。因为在实际的系统运行中,在某一时刻,所有进程都同时请求最大限额的资源,这种情形也不太多见。

在单一资源类型的情形下,银行家算法可以归纳为如下的形式。

```
S1  某个客户提出贷款请求;
S2  假设批准该请求,将得到系统状态 T;
S3  判断状态 T 是否安全,
    如果安全,则批准该请求,转 S1;
    如果不安全,则不批准该请求,延期到以后处理,转 S1。
```

银行家算法

S1　银行家检查一下，看手里的资源能否满足某个客户的请求(剩余的最大限额)；
S2　如果可以，则该客户的贷款请求已经满足，因此他将偿还所有贷款，转 S1；
S3　如果到最后，所有贷款都能偿还，则状态 T 就是安全的，否则就是不安全的。

判断一个状态 T 是否安全

上面这个算法考虑的是单一资源类型的情形，还可以对它进行推广，用来处理多种资源类型的情形。在这种情形下，对于银行家算法本身来说，它和单一资源类型是完全相同的，无须修改。需要修改的是第二个算法，即在多种资源类型的情形下，如何判断一个状态 T 是否安全。该算法需要用到前面讲的 4 个数据结构，即总的资源向量 $\boldsymbol{E}$、空闲资源向量 $\boldsymbol{A}$、当前分配矩阵 $\boldsymbol{C}$，以及请求矩阵 $\boldsymbol{R}$。

S1　在请求矩阵 $\boldsymbol{R}$ 当中，寻找某一行 R_i，它的每一个分量均小于或等于 $\boldsymbol{A}$ 中的相应元素；如果不存在这样的行，则表示找不到一个进程可以运行结束，系统将可能陷入死锁；
S2　如果 R_i 存在，则假设进程 P_i 将请求它需要的所有资源，并得到了满足.然后运行结束，并释放它的所有资源(加入到 $\boldsymbol{A}$ 中)；
S3　重复上述两个步骤，直到所有的进程都能运行结束，这就说明最初的状态 T 是安全的；或者是死锁发生，这就说明 T 是不安全的。

在多种资源类型的情形下判断状态 T 是否安全

图 3.10 是一个具体的例子。在系统中总共有 5 个进程，即 P_1、P_2、P_3、P_4 和 P_5。另外，还有 4 种类型的资源，即 R1、R2、R3 和 R4。从向量 $\boldsymbol{E}$ 可以看出，在系统当中，总共有 3 个 R1、12 个 R2、14 个 R3 和 14 个 R4。在当前的状态下，R1 已经用了 2 个，R2 已经用了 6 个，R3 已经用了 12 个，R4 已经用了 12 个，所以空闲资源向量的值为(1,6,2,2)。

总的资源向量$\boldsymbol{E}$=(3,12,14,14)
空闲资源向量$\boldsymbol{A}$=(1,6,2,2)

	R1	R2	R3	R4
P_1	0	0	3	2
P_2	1	0	0	0
P_3	1	3	5	4
P_4	0	3	3	2
P_5	0	0	1	4

当前分配矩阵$\boldsymbol{C}$

	R1	R2	R3	R4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

请求矩阵$\boldsymbol{R}$

图 3.10　多种资源类型的例子

对于当前这个状态，它是一个安全状态。因为可以找到一个调度顺序，使得每一个进程都能顺利地运行结束。例如，可以先调度进程 P_1 去运行，因为在请求矩阵 $\boldsymbol{R}$ 中，它的每一个分量的值，都是小于或等于向量 $\boldsymbol{A}$ 中的相应元素，即向量(0,0,1,2)是小于或等于向量(1,6,2,2)的。当 P_1 运行结束后，它会释放所占用的所有资源，这些资源被加入空闲资源向量中。因此，向量 $\boldsymbol{A}$ 的值就变成了(1,6,5,4)。接下来，可以选择进程 P_4 去运行，因为它的请求矩阵中的向量为(0,6,5,2)，这是小于或等于 $\boldsymbol{A}$ 的当前值。当 P_4 运行结束后，它也会释放所占用的所有资源，因此空闲资源向量 $\boldsymbol{A}$ 的值将更新为(1,9,8,6)。此时对于 P_2 和 P_5 这两

个进程,它们都能满足条件,随便选择哪一个去运行都可以。如果选择的是 P_5,则当它运行结束并释放了所有资源后,$\boldsymbol{A}$ 的值将更新为(1,9,9,10),然后再依次选择 P_2 和 P_3 即可;如果刚才选择的是 P_2,则当它运行结束并释放了所有资源后,$\boldsymbol{A}$ 的值将更新为(2,9,8,6),此时,系统中的空闲资源的数量已经足够多了。因此对于剩下的 P_3 和 P_5 这两个进程,随便怎么安排都可以。总之,图 3.10 中的状态是安全的,它可以按照 P_1、P_4、P_5、P_2、P_3 的调度顺序去执行,也可以按照 P_1、P_4、P_2、P_5、P_3 或者 P_1、P_4、P_2、P_3、P_5 的顺序去执行,最后所有的进程都能顺利地运行完毕。

假设在当前状态下,进程 P_3 又要请求 1 个 R1 资源和 2 个 R4 资源,那么系统到底给不给它呢?这就要判断如果给了它以后,系统的状态是否还是安全的。

图 3.11 是把 1 个 R1 和 2 个 R4 资源分配给进程 P_3 以后的系统状态,显然,在这种情形下,根据银行家算法,找不到一个调度顺序,使得每一个进程都能顺利地运行结束。事实上,对于请求矩阵 $\boldsymbol{R}$ 中的每一行来说,它们都有 1 个或多个分量的值是大于 $\boldsymbol{A}$ 中的相应分量,这样就找不到任何一个进程去运行。因此,这就说明,图 3.11 中的状态是不安全的,也就是说,在图 3.10 的状态下,不能再把 1 个 R1 资源和 2 个 R4 资源分配给进程 P_3。

总的资源向量$\boldsymbol{E}$=(3,12,14,14)
空闲资源向量$\boldsymbol{A}$=(0,6,2,0)

	R1	R2	R3	R4
P_1	0	0	3	2
P_2	1	0	0	0
P_3	2	3	5	6
P_4	0	3	3	2
P_5	0	0	1	4

当前分配矩阵$\boldsymbol{C}$

	R1	R2	R3	R4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	1	3	5	4
P_4	0	6	5	2
P_5	0	6	5	6

请求矩阵$\boldsymbol{R}$

图 3.11　把 1 个 R1 和 2 个 R4 分配给 P_3 后的状态

通过上面的讨论可以看到,银行家算法能够有效地避免死锁的发生,这是不是就意味着,死锁问题得到了彻底的解决,我们能够从源头上来避免死锁的发生呢?回答是“不”。这就是理论与实际的差别。从理论上来说,银行家算法是精彩的、有效的,但是从实际上来说,它没有太大的用处。

首先,在一个实际的系统中,请求矩阵 $\boldsymbol{R}$ 如何得到?因为请求矩阵描述的是将来的信息,是在将来的一段时间内,进程将会使用哪些资源。但是一个进程在运行之前,并不知道自己将来需要用到哪些类型的资源,以及每一种类型的资源需要多少个,这些信息都是无法事先知道的。

其次,银行家算法需要知道在系统中有多少个进程,而在一个实际的系统中,进程的个数不是固定的,而是动态变化的。

最后,系统中的资源的个数也不是固定的。例如,假设在某个时刻,有一台磁带机突然坏了,这种事情是无法提前预测的。

总之,虽然在理论上银行家算法是非常精彩的。但是在实际的操作系统中,由于信息的缺乏以及系统的动态特征,使得银行家算法难以应用在一个真实的系统中,难以用来实现死锁的避免。

3.4 死锁的预防

既然死锁的避免无法实现，那么在实际的系统中，如何来防止死锁的出现呢？如前所述，死锁的产生有 4 个必要条件，即互斥条件、请求和保持条件、不可抢占条件以及环路等待条件，只有当这 4 个条件同时成立时，才有可能会出现死锁。因此，应对死锁的另一个思路就是破坏死锁产生的 4 个必要条件之一，而这就是死锁的预防所要讨论的内容。

1. 破坏互斥条件

死锁产生的第一个必要条件是互斥条件，即在任何时刻，每一个资源最多只能被一个进程所使用。而破坏这个条件，就意味着允许多个进程同时使用一个资源。

例如，打印机是一种常用的资源，在几乎所有的应用程序中，都有打印的功能。在 Word 文字编辑器中，可以打印当前的文档；在 IE 浏览器中，可以打印当前的网页；在一个图像编辑器中，可以打印一幅图像。但是通常来说，打印机只有一台，而这么多个应用程序在同时运行的时候，如何解决它们对打印机资源的竞争使用呢？如果采用互斥访问的方法，那肯定不行。因为如果一个进程占用了打印机，其他的进程就没法打印了。因此，在当前的系统中，一般采用的是假脱机的打印方式。也就是说，每个应用程序并不是直接与打印机打交道，它们的工作仅仅是生成打印数据，然后提交给一个后台打印进程，然后由这个后台打印进程去真正地使用打印机。由于这个打印进程不会占用任何其他的资源，因此，就可以消除由于争夺打印机资源而引发的死锁问题。

但遗憾的是，破坏互斥条件这种方法并不具有普遍性，并不是所有的资源都可以采用这种方法。

2. 破坏请求和保持条件

请求和保持条件指的是进程在占用若干个资源的同时，又可以去请求新的资源。而破坏这个条件，就是说，不允许进程在占用资源的同时又去申请新的资源。在具体实现上，主要有以下两种做法。

- 要求各个进程在开始运行之前，先一次性请求所有的资源。只有当这些资源都空闲时，系统才会分配给它，然后它可以开始运行。如果在这些资源中，有一个或多个正忙，那么系统就不会分配任何资源给它。换言之，要么全部都给，要么一个也不给。但这种方法有很大的局限性，因为很多进程在运行之前并不知道自己将来需要用到哪些资源，否则，银行家算法就能派上用场了。另外，这种方法难以保证资源的使用效率。例如，进程在一开始申请的资源，可能要到最后才能用上，那么在中间这一段时间内，该资源就一直空闲在那里，而且别的进程也无法使用。
- 要求进程在请求一个新资源时，先暂时释放它已经占用的各个资源，然后再重新申请所有的资源，包括它原来占用的资源和这个新的资源。

3. 破坏环路等待条件

环路等待条件指的是在系统的资源分配图中，存在一条由两个或多个进程所组成的环路链，其中每一个进程都在等待环路链中下一个进程所占用的资源。

为了破坏这种环路链的产生，可以把系统中的所有资源进行编号。然后，进程在申请资源时，必须严格地按照资源编号的递增次序进行，否则操作系统不予分配。

例如,假设在系统中有10个资源,可以把它们编号为R1、R2、…、R10。假设进程P_1当前已经申请了资源R1、R3和R6。此后,它就不能再申请R6之前的资源了,只能申请R6以后的资源。

可以证明:如果每一个进程都按照以上规则进行资源的申请和分配,那么在资源分配图中就不可能出现环路。也就是说,不可能出现死锁。

习　题

一、单项选择题

1. 银行家算法是一种(　　)算法。

A. 死锁解除　　B. 死锁检测　　C. 死锁预防　　D. 死锁避免

2. 某系统中有3个并发进程,都需要同类资源4个,请问该系统中不会发生死锁的最少资源数是(　　)。

A. 9　　B. 10　　C. 11　　D. 12

3. 某计算机系统中有8台打印机,有K个进程竞争使用,每个进程最多需要3台打印机。该系统可能会发生死锁的K的最小值是(　　)。

A. 2　　B. 3　　C. 4　　D. 5

4. 3个进程共享4个同类资源,这些资源的分配与释放只能一次一个。已知每一个进程最多需要两个该类资源,则该系统(　　)。

A. 有某进程可能永远得不到该类资源

B. 必然有死锁

C. 当进程请求该类资源时立刻就能得到

D. 必然无死锁

5. 破坏死锁的4个必要条件之一就可以预防死锁。若规定一个进程在请求新资源之前,首先释放已占有的资源,这是破坏了哪一个条件?(　　)

A. 不可抢占条件　　B. 互斥条件

C. 请求和保持条件　　D. 环路等待条件

二、填空题

1. 死锁产生的根本原因是_____。

2. 在计算机系统中,资源可以分为两种类型:可抢占的资源和不可抢占的资源。对于可抢占的资源,可以通过重新分配资源的方法来避免死锁。那么在计算机系统中,哪一些资源是可抢占的资源?请给出两个具体的例子:_____和_____。

3. 对内存资源的竞争访问_____(可能/不可能)引起死锁。

4. 死锁产生的4个必要条件是:_____、_____、不可强占条件和环路等待条件。

5. 在一个系统中,如果出现了死锁,那么在它的资源分配图中肯定存在有_____。

6. 在一个系统中,要想形成死锁,至少要有_____个进程。

7. 在应对死锁的 4 种策略中，银行家算法属于_____。

8. 死锁的解除主要有 3 种方法，即_____、_____和撤销进程。

三、应用题

1. 在一个系统中，总共有 n 个进程(从 P_1 一直到 P_n)，它们共享使用某一种类型的资源(如绘图仪)，这种资源的个数为 m 个。如果：

   (1) 对于每一个进程 $P_i(i=1,2,\cdots,n)$，它所需要的资源的总个数最少为 1，最多为 m。

   (2) 所有进程对资源的需求量总和小于 $m+n$。

   试证明：该系统没有死锁的危险。

2. 一台计算机有 10 台磁带机。它们由 N 个进程竞争使用，每个进程最多需要 4 台磁带机。请问 N 为多少时，系统没有死锁的危险？并说明原因。

3. 假设系统中有 3 种类型的资源{A,B,C }和四个进程{P_1,P_2,P_3,P_4}，A 资源的数量为 12，B 资源的数量为 9，C 资源的数量为 12。已知在某个时刻系统状态如下：

当前分配矩阵

	A	B	C
P_1	2	1	3
P_2	1	2	3
P_3	5	4	3
P_4	2	1	2

请求矩阵

	A	B	C
P_1	2	2	1
P_2	4	1	0
P_3	1	0	0
P_4	2	0	0

   (1) 请问，系统是否处于安全状态？如果是，请给出一个安全序列；如果不是，为什么？

   (2) 假设进程 P_1 再申请两个 A 类型的资源，系统能否给它？为什么？

4. 设系统中有 4 种类型的资源{A,B,C,D}和 5 个进程{P_1,P_2,P_3,P_4,P_5}，A 资源的数量为 3，B 资源的数量为 12，C 资源的数量为 14，D 资源的数量为 14。假设在某时刻，系统的状态如下：

进　　程	已分配的资源数量 (A　B　C　D)	仍需要的资源数量 (A　B　C　D)
P_1	0　0　3　2	0　0　1　2
P_2	1　0　0　0	1　7　5　0
P_3	1　3　5　4	2　3　5　6
P_4	0　3　3　2	0　6　5　2
P_5	0　0　1　4	0　6　5　6

   请问：

   (1) 该状态是否安全？若是请给出一个安全序列。

   (2) 如果此时进程 P_3 请求资源(1,2,2,2)，系统能否将这些资源分配给它？

5. 设系统中有 3 种类型的资源{A,B,C}和 5 个进程{P_1,P_2,P_3,P_4,P_5}，A 资源的

数量为17,B资源的数量为5,C资源的数量为20。在 T_0 时刻系统状态如下:

	最大资源需求量	已分配资源数量
	(A B C)	(A B C)
P_1	5 5 9	2 1 2
P_2	5 3 6	4 0 2
P_3	4 0 11	4 0 5
P_4	4 2 5	2 0 4
P_5	4 2 4	3 1 4

(1) T_0 时刻是否为安全状态?若是,请给出一个安全序列。

(2) 在 T_0 时刻,若进程 P_2 请求资源(0,3,4),是否能实施资源分配?为什么?

(3) 在(2)基础上,若进程 P_4 请求资源(2,0,1),是否能实施资源分配?

(4) 在(3)基础上,若进程 P_1 请求资源(0,2,0),是否能实施资源分配?

6. 在操作系统中,可以通过破坏环路等待条件来预防死锁。具体来说,可以把系统中的所有资源进行编号,进程在申请资源时必须严格按资源编号的递增次序进行,否则操作系统不予分配。例如,假设系统有10个资源R1、R2、…、R10,如果一个进程已经申请了R1、R3和R6,那么它就不能再申请R6之前的资源。请证明:如果按照以上规则进行资源的申请和分配,系统就不会发生死锁。