第3章 死锁

死锁是多道程序并发执行带来的另一个严重问题，它是操作系统乃至并发程序设计中最难处理的问题之一。进程死锁产生的根本原因有两个：一是竞争资源；二是进程间推进的顺序不合理。死锁处理不好将导致整个系统运行效率下降，甚至不能正常运行。

第3章 死锁

- 3.1 死锁的定义和产生原因
- 3.2产生死锁的必要条件
- 3.3死锁的处理方法
- 3.4死锁的预防
- 3.5死锁的避免
- 3.6死锁的检测
- 3.7死锁的解除
- 3.8死锁的综合处理策略
- 3.9线程死锁
- 3.10本章小结

本章要点

- 本章主要讲解死锁的基本概念、死锁的处理策略、死锁的预防、死锁的避免以及死锁的检测和解除。本章重点掌握以下要点：
- 了解死锁的基本概念；了解死锁的检测、死锁解除的方法；
- 理解死锁的产生的原因；理解死锁预防与死锁避免的区别；
- 掌握死锁产生的四个必要条件；掌握系统安全状态及其判别方法；掌握处理死锁的方法；
- 学会银行家算法及其应用。

3.1 死锁的定义和产生原因

3.1.1 死锁的定义

- 在多道程序系统中，虽可通过多个进程的并发执行来改善系统的资源利用率和提高系统的处理能力，但可能发生一种危险——死锁。**死锁是多个进程因竞争资源而造成的一种僵局**，若无外力作用，这些进程都将永远不能再向前推进。
- 死锁的定义：**死锁是指一组并发执行的进程彼此等待对方释放资源，而在没有得到对方占有的资源之前不释放自己所占有的资源，导致彼此都不能向前推进，称该组进程发生了死锁**。

3.1 死锁的定义和产生原因

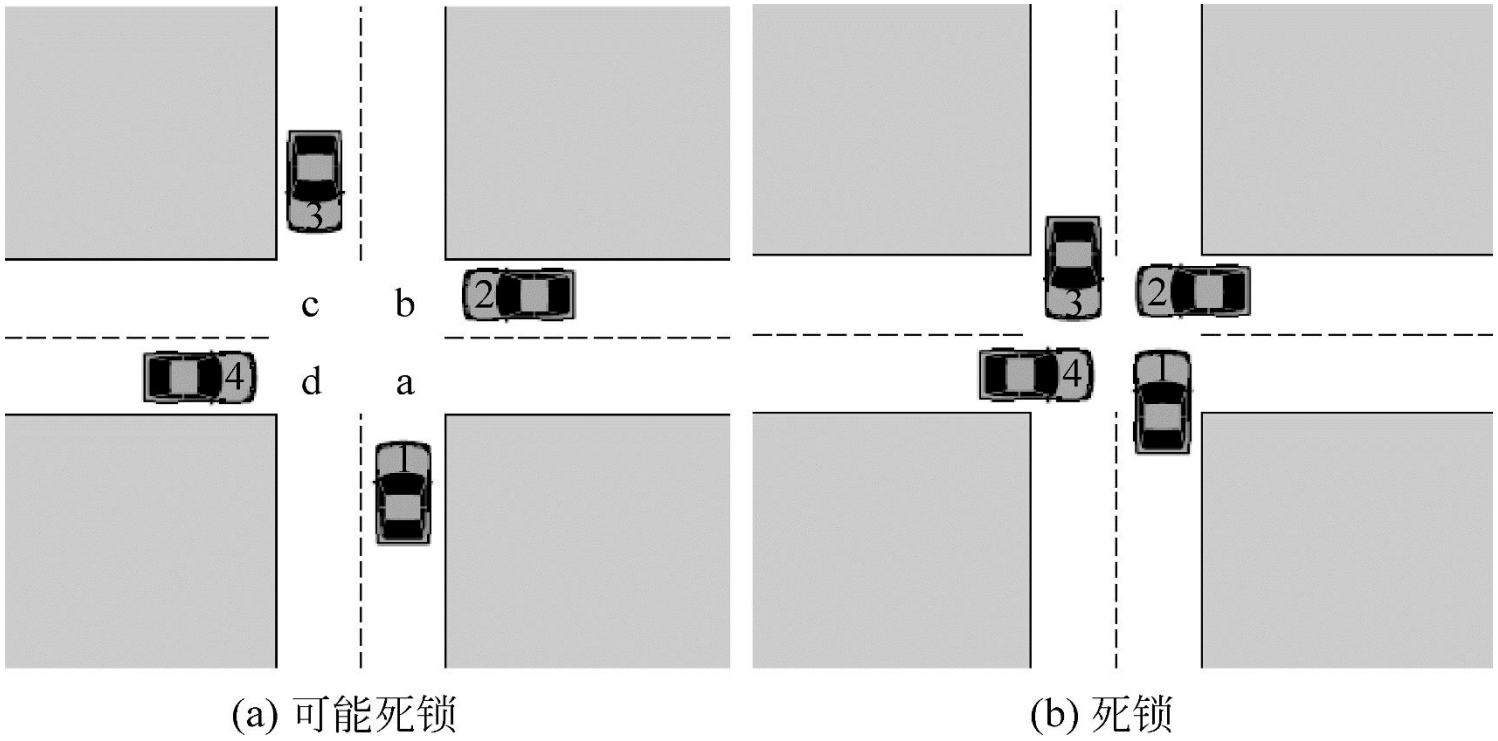


图3-1 交通阻塞导致死锁示意图

3.1 死锁的定义和产生原因

- 死锁具有以下特点：
  ① 陷入死锁的进程是系统并发进程中的一部分，且至少要有两个进程，单个进程不会形成死锁。
  ② 陷入死锁的进程彼此都在等待对方释放资源，形成一个循环等待链。
  ③ 死锁形成后，在没有外力干预下，陷入死锁的进程不能自己解除死锁，死锁进程无法正常结束。
  ④ 如不及时解除死锁，死锁进程占有的资源不能被其他进程所使用，导致系统中更多进程阻塞，造成资源利用率下降。

3.1.2 死锁产生的原因

- 产生死锁的原因可归结为两点：
  ① **竞争资源**。当系统中供多个进程所共享的资源，不足以同时满足它们的需要时，引起它们对资源的竞争而产生死锁。
  ② **进程推进顺序不当**。进程在运行过程中，请求和释放资源的顺序不当，导致了进程死锁。

3.1.2 死锁产生的原因

1. 竞争资源

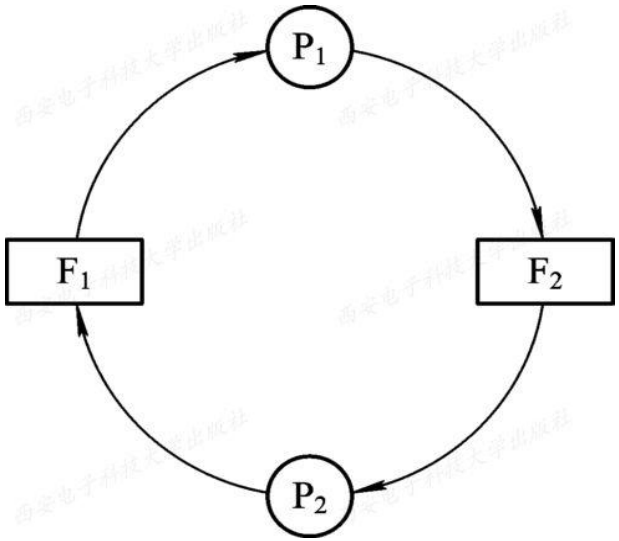
- 死锁产生的根本原因是资源竞争且分配不当。
- 按占用方式来分，可分为：

① **可剥夺资源**：某进程在获得这类资源后，即使该进程没有使用完，该类资源也可以被其他进程剥夺使用。

② **不可剥夺资源**：当系统把这类资源分配给某进程后，不能强行收回，只能在进程使用完后自行释放，其他进程才能使用。

进程P1占用资源F1

进程P1申请资源F2

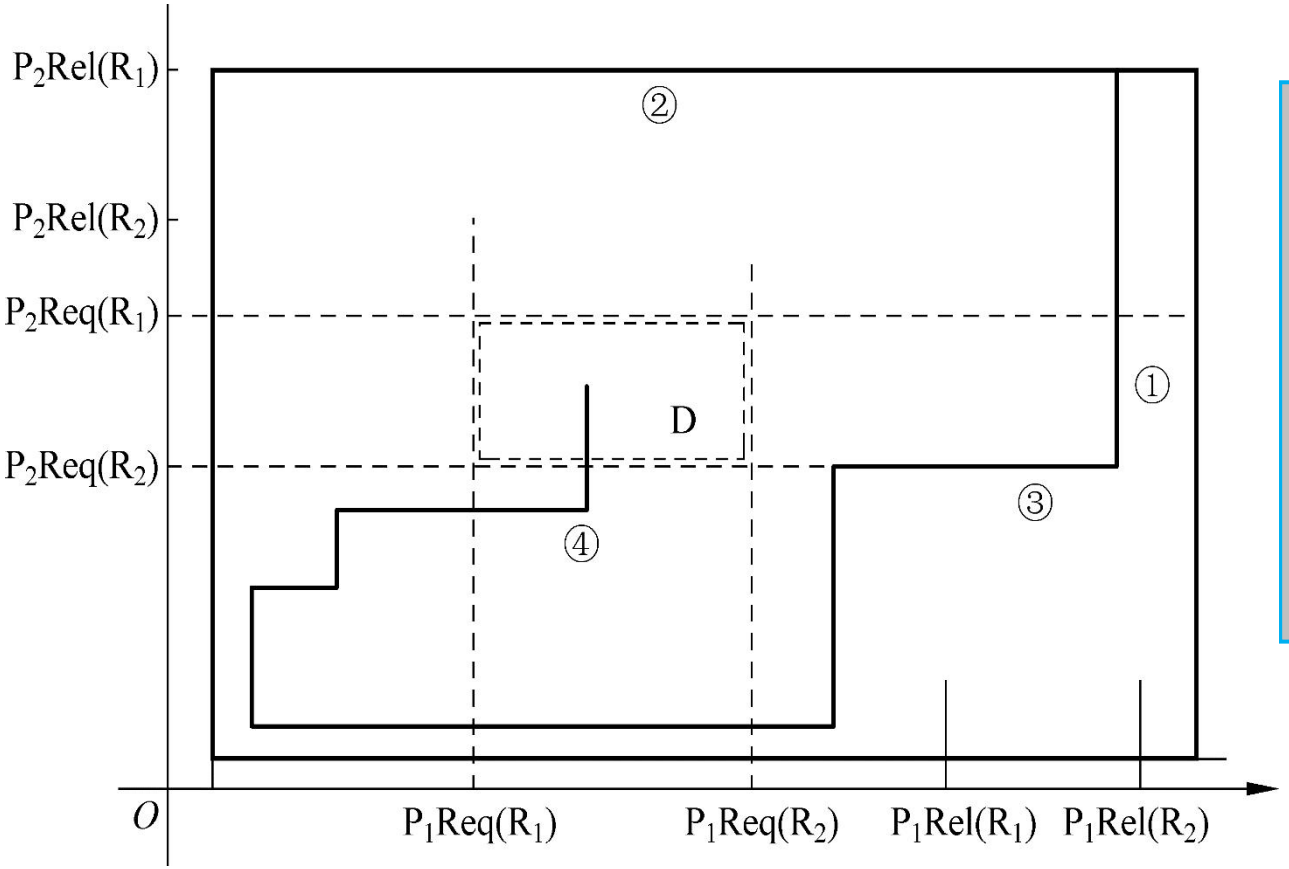


进程P2申请资源F1

进程P2占用资源F2

3.1.2 死锁产生的原因

2. 进程推进顺序不当



进程p_1和进程p_2交替占用CPU执行时，顺序为①②③曲线时，不会出错。

但为④ 曲线时，会死锁。

3.2 产生死锁的必要条件

- 系统产生死锁必定同时保持以下四个必要条件：
  ① **互斥条件**：进程应互斥使用资源，任一时刻一个资源仅为一个进程独占，若另一个进程请求一个已被占用的资源时，它被置成等待状态，直到占用者释放资源。
  ② **占有且等待条件**：一个进程请求资源得不到满足而等待时，不释放已占有的资源。
  ③ **不剥夺条件**：任一进程不能从另一进程那里抢夺资源，即已被占用的资源，只能由占用进程自己来释放。
  ④ **循环等待条件**：存在一个循环等待链，其中，每一个进程分别等待另一个进程所持有的资源，造成永远等待。

3.2 产生死锁的必要条件

- 这四个条件仅是**必要条件**而不是充分条件，即只要发生死锁则这四个条件一定会同时成立，但反之则不然。循环等待条件隐含着前三个条件，即只有前三个条件成立，第四个条件才会成立。特别注意：环路等待条件只是死锁产生的必要条件，而不是等价定义。**死锁一旦产生则死锁进程间必存在循环等待链，但存在循环等待链不一定产生死锁**。

3.2 死锁的处理方法

- 死锁普遍存在于并发执行进程间，处理死锁的方法很多种。其中最简单的处理方法就是**忽略死锁**。就像鸵鸟遇到无法避免的危险时就把头埋在沙子里一样，对出现的危险不管不顾。操作系统处理死锁的一种策略就是不预防、不避免，对可能出现的死锁采取放任的态度，称作**鸵鸟算法**。
- 鸵鸟算法的意义在于，当出现死锁的概率很小，并且出现之后处理死锁会花费很大的代价时，执行死锁避免的开销很大，还不如不做处理。因此，**鸵鸟算法是平衡性能和复杂性的一种方法，是目前通用操作系统中采用最多的方法**。

3.2 死锁的处理方法

- 按照死锁处理的时机划分，可把死锁处理的方法分成四种
  - ① 预防死锁（静态策略）
  - ② 避免死锁（动态策略）
  - ③ 检测死锁
  - ④ 解除死锁

3.2 死锁的处理方法-预防死锁

- 预防死锁是在系统运行之前就采取相应措施，消除发生死锁的任何可能性。
- 消除死锁发生的必要条件可预防死锁。破坏产生死锁的四个必要条件中一个或几个来预防产生死锁。
- 预防死锁是处理死锁的静态策略，它虽比较保守、资源利用率低，但因简单明了较易实现，现仍被广泛使用。

3.2 死锁的处理方法-避免死锁

- 避免死锁是为了克服预防死锁的不足而提出的动态策略。
- 避免死锁与预防死锁的策略不同，它并不是事先采取各种限制措施，去破坏产生死锁的四个必要条件，而是在资源动态分配过程中，用某种方法防止系统进入不安全状态，从而可以避免发生死锁。
- 避免死锁方法虽好，但也存在两个缺点：
  - 一是对每个进程申请资源分析计算较为复杂且系统开销较大；
  - 二是在进程执行前，很难精确掌握每个进程所需的最大资源数。

3.2 死锁的处理方法-检测死锁

- 这种方法无须事先采取任何限制性措施，允许进程在运行过程中发生死锁。但可通过检测机构及时地检测出死锁的发生，然后采取适当的措施，把进程从死锁中解脱出来。
- 死锁检测不延长进程初始化时间，允许对死锁进行现场处理，其缺点是通过剥夺解除死锁，给系统或用户造成一定的损失。

3.2 死锁的处理方法-解除死锁

- 当检测到系统中已发生死锁时，就采取相应措施，将进程从死锁状态中解脱出来。
- 常用的方法是撤销一些进程，回收它们的资源，将它们分配给已处于阻塞状态的进程，使其能继续运行。
- 在实际执行中，由于并发进程推进顺序的多样性，系统很难做到有效地解除死锁。

上述的四种方法，从1）到4）对死锁的防范程度逐渐减弱，但对应的是资源利用率的提高，以及进程因资源因素而阻塞的频度下降（即并发程度提高）。具体处理死锁的基本方法比较如表3-1所示。

表3-1 处理死锁的基本方法比较

方法	资源分配策略	各种可能模式	主要优点	主要缺点
死锁的预防	保守的； 宁可资源闲置	一次性请求所有资源	适用于作突发式处理的进程，不用被剥夺	效率低；进程初始化时间延长
		资源被剥夺	适用于状态可以保存和恢复的资源	剥夺次数过多；多次对资源重新启动
		资源按序申请	可以在编译时（而不必在运行时）就进行检查	不便灵活申请新资源，申请序号很难确定
死锁的避免	在运行时动态的分配资源，判断系统是否是安全状态	寻找安全序列	不会进行剥夺	必须知道将来的资源需求；进程可能会长时间阻塞
死锁的检测和恢复	宽松的；只要允许，就分配资源	定期检查系统是否发生死锁	不延迟进程初始化时间；允许对死锁进程进行现场处理	通过剥夺解除死锁，造成损失

3.4 死锁的预防

- 预防死锁的方法是通过破坏产生死锁的四个必要条件中的一个或几个，以避免发生死锁。由于互斥条件是非共享设备所必须的，不仅不能改变，还应加以保证，因此主要是破坏产生死锁的后三个条件。

3.4.1 破坏“请求”和“保持”条件

- 破坏这个条件很简单，可采用预分配资源方法。所有进程在开始运行前，系统一次性地分配其运行所需要的全部资源。进程在运行期间，不会再提出资源要求。
- 系统在分配资源时，只要有一种资源不能满足进程的要求，即使其他所需要的资源空闲也不能分配给该进程，而让该进程等待。

- 优点：简单、易于实现。
- 缺点：资源利用率低；进程产生饥饿现象。

3.4.2 破坏“不剥夺”条件

- 破坏“不剥夺”条件就是采用可剥夺的资源分配方式，即允许对系统的资源进行抢占。当一个已经保持了某些不可被抢占资源的进程，提出新的资源请求而不能得到满足时，它必须释放已经保持的所有资源，待以后需要时再重新申请。这就意味着进程已占有的资源会被暂时地释放，或者被抢占了，从而破坏了“不剥夺”条件。

- 缺点：实现复杂，代价大。
- 适用范围：资源的状态能够很方便的保存和恢复，例如CPU寄存器和存储空间。

3.4.3 破坏“循环等待”条件

- 一个能保证“循环等待”条件不成立的方法是，对系统所有资源类型进行线性排序，并赋予不同的序号。
  - 资源编号：设$R=\{r_1, r_2, \ldots, r_m\}$，表示一组资源，定义一对一的函数F：R→N，N是一组自然数。如：F（磁带机）= 1，F（磁盘机）= 5，F（打印机）= 12
- ①依序分配
  - 约定所有进程对资源的申请严格按照序号递增的次序进行。
- ②先弃大，再取小
  - 一个进程申请资源r_j，它应释放所有满足$F(r_i) \geq F(r_j)$ 关系的资源r_i
- 这两种办法都可行，都可排除循环等待条件。
  - 优点：资源利用率和系统吞吐量有所提高。
  - 缺点：找不出一种人人满意的编号顺序；仍存在资源浪费；用户编程受到限制。

3.5 死锁的避免

- 死锁的预防是静态策略，对进程申请资源的活动进行严格限制，以保证死锁不会发生。
- 死锁的避免和死锁的预防不同，系统允许进程动态地申请资源，系统在进行资源分配之前，对进程发出的资源申请进行严格检查，如满足该申请后系统仍处于安全状态，则分配资源给该进程，否则拒绝此申请。

3.5.1 安全状态

- 所谓安全状态是指系统能够按照某种进程执行序列，如<P_1,P_2,P_3,...,P_n>为每个进程分配所需资源，直至满足每个进程对资源的最大需求，使得每个进程都可顺利地完成。此时，称系统处于系统安全状态，进程执行序列<P_1,P_2,P_3,...,P_n>为当前系统的一个安全序列。

- 如果系统无法找到这样一个安全序列，则称当前系统处于不安全状态。

- **【例1】现有12个同类资源供3个进程共享，进程P_1总共需要9个资源，但第一次先申请2个资源，进程P_2总共需要10个资源，第一次要求分配5个资源，进程P_3总共需要4个资源，第一次请求2个资源。经第一轮的分配后，系统中还有3个资源未被分配，现在的分配情况如表3-2所示。**

表3-2　资源分配状态

进程	已占资源数	最大需求数
P_1	2	9
P_2	5	10
P_3	2	4

问系统此时是否处于安全状态？

- 在这种情况下，如果按P_3、P_2、P_1的顺序分配资源的话，可顺利执行完而无死锁发生。
- 在这种情况下，如果按P_1、P_2、P_3的顺序分配资源的话，则可发生死锁。
- 因为系统只要能保持处于安全状态就可避免死锁的发生，故每当有进程提出分配资源的请求时，系统应分析各进程已占资源数、尚需资源数和系统中可以分配的剩余资源数，然后决定是否为当前的申请分配资源。如果能维持系统的安全状态则可为进程分配资源，否则暂不为申请者分配资源，直到有其他进程归还资源后再分配给它。

3.5.2 银行家算法

- 最有代表性的避免死锁的算法就是Dijkstra的银行家算法。银行家算法的基本思想是：在资源分配前，判断系统是否处于安全状态，如处于安全状态则把资源分配给申请进程，如处于不安全状态则令申请资源的进程阻塞，不响应其资源申请。

3.5.2 银行家算法

- 用现实生活中的银行贷款实例来类比银行家算法的执行过程。例如银行家有一笔资金M万元，N个客户需要贷款，他们都和银行签订了贷款协议，每个客户所需的资金不同且都不超过M万元，但客户们的贷款总和远远超过M万元。协议中规定银行根据自身的情况向各个客户发放贷款。客户只有在获得全部贷款后，才能在一定的时间内将全部资金归还给银行家。银行家并不一定批准客户每次的贷款请求，在每次发放贷款时，银行家都要考虑发放该笔贷款是否会造成银行无法正常运转。只有在批准贷款请求不会导致银行银根不足时，该贷款请求才被批准。

银行家算法的数据结构

① **可利用资源向量Available**。这是一个含有m个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果Available［j］=K，则表示系统中现有K个R_j类资源。

② **最大需求矩阵Max**。这是一个n×m的矩阵，它定义了系统中n个进程中的每一个进程对m类资源的最大需求。如果Max［i,j］=K，则表示进程i需要R_j类资源的最大数目为K。

银行家算法的数据结构

③ **分配矩阵Allocation**。这也是一个n×m的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果Allocation［i,j］=K，则表示进程i当前已分得R_j类资源的数目为K。

④ **需求矩阵Need**。这也是一个n×m的矩阵，用以表示每一个进程尚需的各类资源数。如果Need［i,j］=K，则表示进程i还需要R_j类资源K个，方能完成其任务。

$$Need[i,j]=Max[i,j]-Allocation[i,j]$$

3.5.2 银行家算法

- 设$Request_i$是进程P_i的请求向量，如果$Request_i[j]=K$，表示进程P_i需要K个R_j类型的资源。当P_i发出资源请求后，系统按下述步骤进行检查：
  - ① 如果$Request_i[j] \le Need[i,j]$，便转向步骤②；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。
  - ② 如果$Request_i[j] \le Available[j]$，便转向步骤③；否则， 表示尚无足够资源，$P_i$须等待。
  - ③ 系统试探着把资源分配给进程P_i，并修改下面数据结构中的数值：
    - a) $Available[j] := Available[j] - Request_i[j]$;
    - b) $Allocation[i,j] := Allocation[i,j] + Request_i[j]$;
    - c) $Need[i,j] := Need[i,j] - Request_i[j]$;
  - ④ 系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式将资源分配给进程P_i，以完成本次分配；否则， 将本次的试探分配作废，恢复原来的资源分配状态，让进程P_i等待。

3.5.2 银行家算法

- 安全性算法：
  - ① 设置两个向量： (a)工作向量Work: 它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时，Work：=Available; (b)Finish: 它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做Finish［i］：=false; 当有足够资源分配给进程时， 再令Finish［i］：=true。
  - ② 从进程集合中找到一个能满足下述条件的进程：a) Finish［i］=false; b)Need［i,j］≤Work［j］； 若找到， 执行步骤③， 否则，执行步骤④。
  - ③ 当进程P_i获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：
    a) Work［j］：=Work［j］+Allocation［i,j］;
    b) Finish［i］：=true;
    c) 执行步骤②;
  - ④ 如果所有进程的Finish［i］=true都满足， 则表示系统处于安全状态；否则，系统处于不安全状态。

3.5.2 银行家算法

- 【例2】假设系统中有五个进程{P_0、P_1、P_2、P_3、P_4}和三类资源{A，B，C}，各种资源的数量分别为10、5、7，在T_0时刻的资源分配情况如下表所示。

资源情况 / 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3	3	3	2
P_1	3	2	2	2	0	0	1	2	2			
P_2	9	0	2	3	0	2	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	4	3	3	0	0	2	4	3	1			

3.5.2 银行家算法

① T_0时刻的安全性：存在着安全序列$\{P_1,P_3,P_4,P_2,P_0\}$，如下表所示。

资源 情况 进程	Max			Allocation			Need			Work(Available)			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	
P_0	7	5	3	0	1	0	7	4	3	10	4	7	10	5	7	5
P_1	3	2	2	2	0	0	1	2	2	3	3	2	5	3	2	1
P_2	9	0	2	3	0	2	6	0	0	7	4	5	10	4	7	4
P_3	2	2	2	2	1	1	0	1	1	5	3	2	7	4	3	2
P_4	4	3	3	0	0	2	4	3	1	7	4	3	7	4	5	3

3.5.2 银行家算法

② P_1请求资源：P_1发出请求向量$Request_1(1，0，2)$，系统按银行家算法进行检查：

a) $Request_1(1, 0, 2) \leq Need_1(1, 2, 2)$

b) $Request_1(1, 0, 2) \leq Available_1(3, 3, 2)$

c) 系统先假定可为P_1分配资源，并修改Available, $Allocation_1$和$Need_1$向量，由此形成的资源变化情况如下表所示。

d) 再利用安全性算法检查此时系统是否安全。{P_1,P_3,P_4,P_0,P_2}

资源情况 / 进程	Max			Allocation			Need			Work(Available)			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	
P_0	7	5	3	0	1	0	7	4	3	7	4	5	7	5	5	4
P_1	3	2	2	3	0	2	0	2	0	2	3	0	5	3	2	1
P_2	9	0	2	3	0	2	6	0	0	7	5	5	10	5	7	5
P_3	2	2	2	2	1	1	0	1	1	5	3	2	7	4	3	2
P_4	4	3	3	0	0	2	4	3	1	7	4	3	7	4	5	3

3.5.2 银行家算法

③ P_4请求资源：P_4发出请求向量$Request_4(3，3，0)$，系统按银行家算法进行检查：

a) $Request_4(3, 3, 0) \leq Need_4(4, 3, 1)$;

b) $Request_4(3, 3, 0) > Available(2, 3, 0)$，让$P_4$等待。

④ P_0请求资源：P_0发出请求向量$Requst_0(0，2，0)$，系统按银行家算法进行检查：

a) $Request_0(0, 2, 0) \leq Need_0(7, 4, 3)$;

b) $Request_0(0, 2, 0) \leq Available(2, 3, 0)$;

c) 系统暂时先假定可为P_0分配资源，并修改有关数据，如下图所示。

资源情况 / 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	3	0	7	2	3	2	1	0
P_1	3	2	2	3	0	2	0	2	0			
P_2	9	0	2	3	0	2	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	4	3	3	0	0	2	4	3	1			

3.5.2 银行家算法

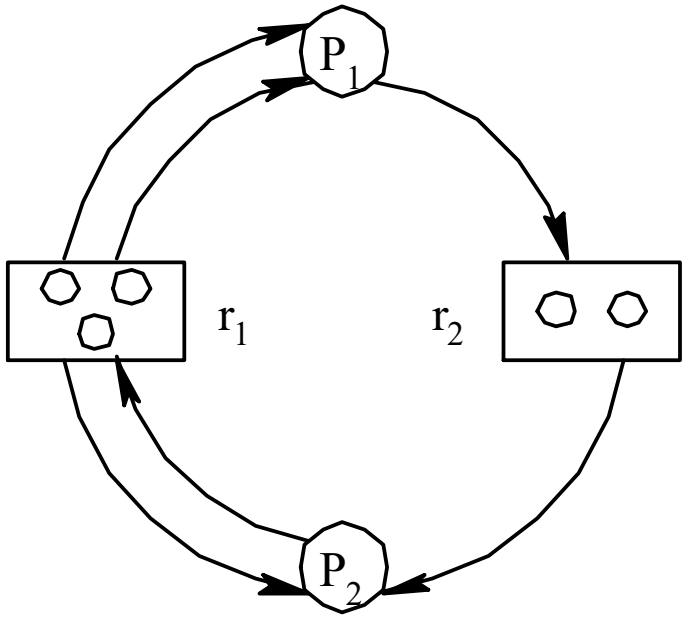
⑤ 进行安全性检查：可用资源Available(2，1，0)已不能满足任何进程的需要，故系统进入不安全状态，此时系统不分配资源。

- 通过这个例子，我们看到银行家算法确实能保证系统时时刻刻都处于安全状态，但它要不断检测每个进程对各类资源的占用和申请情况，需花费较多的时间。

3.6 死锁的检测

- 对资源的分配加以限制可以防止和避免死锁的发生，但这不利于各进程对系统资源的充分共享。解决死锁问题的另一条途径是死锁检测和解除，这种方法对资源的分配不加任何限制，也不采取死锁避免措施，但系统定时地运行一个“死锁检测”程序，判断系统内是否已出现死锁，如果检测到系统已发生了死锁，再采取措施解除它。
- 操作系统中的每一时刻的系统状态都可以用进程资源分配图来表示，进程资源分配图是描述进程和资源间申请及分配关系的一种有向图，可用以检测系统是否处于死锁状态。

3.6.1 资源分配图



圆圈○代表一个进程；
方框□代表一类资源；
请求边：○→□；
分配边：□→○。

3.6.1 资源分配图

- 设一个计算机系统中有许多类资源和许多个进程。每一个资源类用一个方框表示，方框中的黑圆点表示该资源类中的各个资源，每个进程用一个圆圈表示，用有向边来表示进程申请资源和资源被分配的情况。约定$P_i \rightarrow R_j$为请求边，表示进程P_i申请资源类R_j中的一个资源得不到满足而处于等待R_j类资源的状态，该有向边从进程开始指到方框的边缘，表示进程P_i申请R_j类中的一个资源。反之$R_j \rightarrow P_i$为分配边，表示R_j类中的一个资源已被进程P_i占用，由于已把一个具体的资源分给了进程P_i，故该有向边从方框内的某个黑圆点出发指向进程。
- 图3-10是进程—资源分配图的一个例子，这个例子中，由于存在占有和等待资源的环路，导致一组进程永远处于等待资源状态，发生了死锁。
- 进程—资源分配图中存在环路并不一定发生死锁。因为循环等待资源仅是死锁发生的必要条件，而不是充分条件，图3-11便是一个有环路而无死锁的样子。

3.6.1 资源分配图

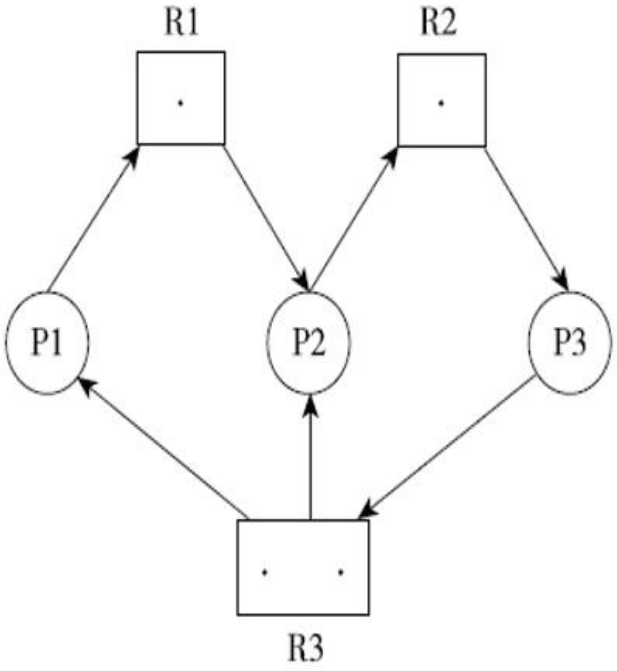


图3-10 进程资源分配图的一个例子

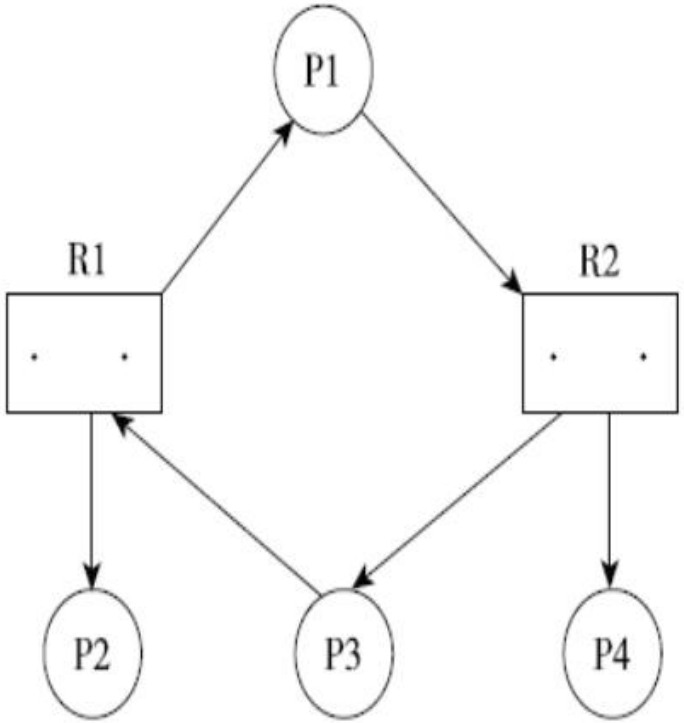


图3-11 有环路而无死锁的一个例子

3.6.2 死锁定理

- 可以利用下列步骤运行一个“死锁检测”程序，对进程资源分配图进行分析和简化，以此方法来检测系统是否处于死锁状态：
  ① 如果进程资源分配图中无环路，则此时系统没有发生死锁；
  ② 如果进程资源分配图中有环路，且每个资源类中仅有一个资源，则系统中发生了死锁，此时，环路是系统发生死锁的充分条件，环路中的进程便为死锁进程；
  ③ 如果进程资源分配图中有环路，且涉及的资源类中有多个资源，则环路的存在只是产生死锁的必要条件而不是充分条件，系统未必一定就会发生死锁。系统为死锁状态的充分条件是：当且仅当该状态的进程资源分配图是不可完全简化的。该充分条件称为死锁定理。

3.6.2 死锁定理

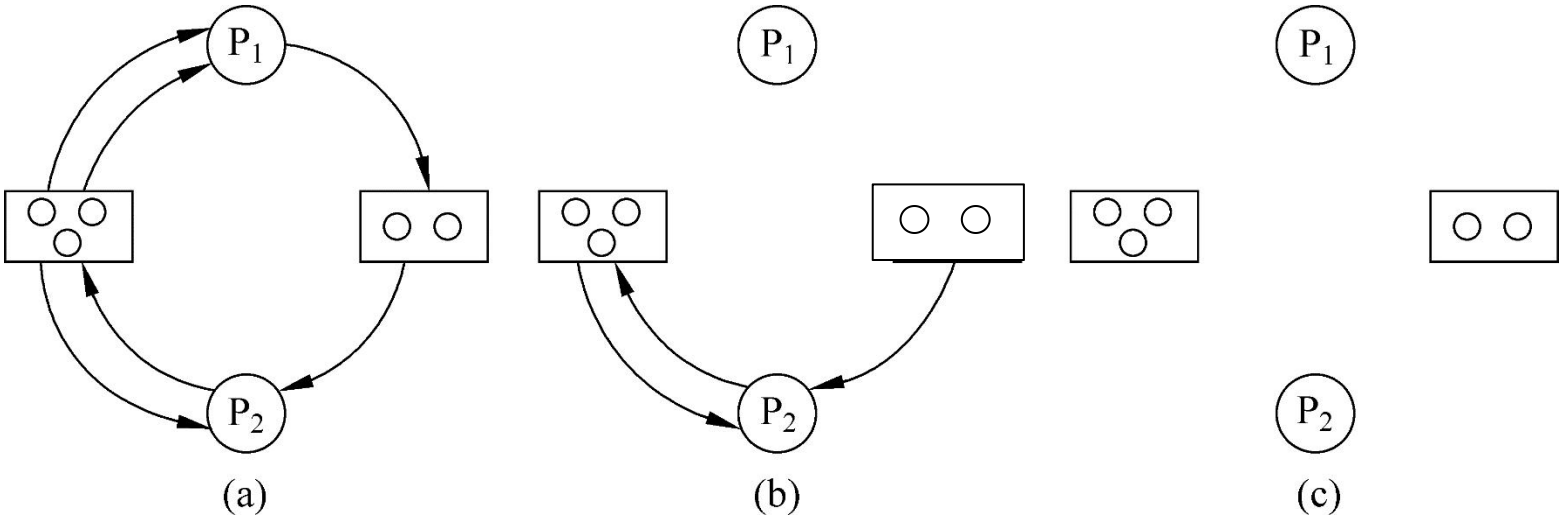


图3-12 资源分配图的简化

3.6.3 死锁检测算法

- 系统中每类资源的资源实例是多个时，可采用下面介绍的死锁检测算法进行检测。该算法由Shoshani和Coffman提出，采用了和银行家算法类似的数据结构。

3.6.3 死锁检测算法-数据结构

① 当前可分配的空闲资源向量Available(1:m)。m是系统中的资源类型数。

Available[i]表示系统中现有的i类资源数量。

② 资源分配矩阵Allocation(1:n，1:m)。

Allocation[i,j]表示进程i已占有的j类资源的数量。

③ 需求矩阵Request(1:n，1:m）。

Request[i,j]表示进程i还需申请j类资源的数量。

3.6.3 死锁检测算法-检测步骤

① 令Work和Finish分别表示长度为m和n的向量，初始化Work=Available；对于所有i=1，…，n，如果Allocation[i]≠0，则Finish[i]=false，否则Finish[i]=true。

② 寻找一个下标i，它满足条件：Finish[i]=false且Request[i]<=Work，如果找不到这样的i，则转向步骤④。

③ Work=Work+Allocation[i]；Finish[i]=true；转向步骤②。

④ 如果存在i，1<=i<=n，Finish[i]=false，则系统处于死锁状态。若Finish[i]=false，则进程处于死锁环中。

3.6.3 死锁检测算法

- 【例3】设系统中有3个资源类{r_1，r_2，r_3}和5个并发进程{P_1，P_2，P_3，P_4，P_5}，其中r_1有7个，r_2有3个，r_3有6个。在T_0时刻各进程分配资源和申请情况表如表3-8所示。

表3-8 T_0时刻各进程分配资源和申请情况表

资源 / 进程	Allocation			Request			Available		
	r1	r2	r3	r1	r2	r3	r1	r2	r3
P_1	0	1	0	0	0	0	0	1	0
P_2	2	0	0	2	0	2			
P_3	3	0	3	0	0	0			
P_4	2	1	1	1	0	0			
P_5	0	0	2	0	0	2			

3.6.3 死锁检测算法

- 根据上面的死锁检测算法，我们可以得到一个进程的安全序列< P_1，P_2，P_3，P_4，P_5>对于所有的Finish[i]=true，所以，此时系统T_0时刻不处于死锁状态。假定，进程P_3现在申请一个单位为r_3的资源，则系统资源分配情况表如表3-9所示。

表3-9 满足进程P_3申请后的系统资源分配情况

资源 / 进程	Allocation			Request			Available		
	r1	r2	r3	r1	r2	r3	r1	r2	r3
P_1	0	1	0	0	0	0	0	1	0
P_2	2	0	0	2	0	2			
P_3	3	0	3	0	0	1			
P_4	2	1	1	1	0	0			
P_5	0	0	2	0	0	2			

参与死锁的进程集合为{ P_2，P_3，P_4，P_5}

3.7 死锁的解除

- 当死锁检测程序检测到死锁存在时，应设法将其解除，让系统从死锁状态中恢复过来，常用的解除死锁的办法有以下几种：
  ① 立即结束所有进程的执行，并重新启动操作系统。这种方法简单，但以前所做的工作全部作废，损失很大。
  ② 撤销涉及死锁的所有进程，解除死锁后继续运行。这种方法能彻底破坏死锁的循环等待条件，但将付出很大代价。
  ③ 逐个撤销涉及死锁的进程，回收其资源，直至死锁解除。但是先撤销哪个死锁进程呢？可选择符合下面一种条件的进程先撤销：消耗的CPU时间最少者、产生的输出最少者、预计剩余执行时间最长者、占有资源数最少者或优先级最低者。
  ④ 抢夺资源，从涉及死锁的一个或几个进程中抢夺资源，把夺得的资源再分配给涉及死锁的其他进程直到死锁解除。

3.8 死锁的综合处理策略

- 在一个资源类中，使用该类资源最适合的算法。作为该技术的一个例子，可以考虑下列资源类：
  - 可交换空间：在进程交换中所使用的辅存中的存储块。
  - 进程资源：可分配的设备，如磁带设备和文件。
  - 主存：可以按页或按段分配给进程。
  - 内部资源：例如I/O通道。

- 以上列出的次序表示了资源分配的次序。

3.8 死锁的综合处理策略

- 考虑到一个进程在其生命周期中的步骤顺序，这次序是最合理的。在每一类资源中，可以采用以下策略：
- 对于可交换空间，通过要求一次性分配所有请求的资源来预防死锁，就像占有且等待预防办法一样。如果知道最大存储需求（一般通常情况下都知道），则这个策略是合理的。死锁避免也是可能的。
- 对进程资源，死锁避免的的方法常常是很有效的，这是因为进程可以事先声明它们将需要的这类资源。采用资源排序的预防策略也是可能的。
- 对于主存，基于抢占的预防是最适合的策略。当一个进程被抢占后，它仅仅被换到辅存，释放空间以解决死锁。
- 而对于内部资源可以使用基于资源按序排列的预防策略。

3.9 线程死锁

- 在支持多线程的操作系统中，除了会发生进程之间的死锁外，还会发生线程之间的死锁。由于不同的线程可以属于同一个进程，也可以属于不同的进程。因此，与进程死锁比较，线程死锁分为属于同一进程的线程死锁和属于不同进程的线程死锁。
- 1）同一进程的线程死锁。线程的同步工具有互斥锁。由于同一进程的线程共享该进程资源，为了实现线程对进程内变量的同步访问，可以采用互斥锁。假如，L_1和L_2为两个互斥锁，进程内的一个线程先获得L_1，然后申请获得L_2，同一进程内的另一个线程先获得L_2，再申请获得L_1。这样一来，同一进程内的两个线程陷入死锁。

3.9 线程死锁

- 2）不同进程的线程死锁。如果在进程P_1中存在一组线程{P_{11}，P_{12}，…，P_{1m}}，在进程P_2中存在一组线程{P_{21}，P_{22}，…，P_{2m}}。同一时间段内，进程P_1内的线程获得资源R_1，进程P_2内的线程获得资源R_2。如果进程P_1内的某个线程P_{1i}请求资源R_2，由于不能满足而进入阻塞状态；进程P_2内某个线程P_{2j}请求资源R_1，由于不能满足而进入阻塞状态。线程P_{1i}和线程P_{2j}相互等待对方释放资源，这是出现了不同进程线程间的死锁。

- 当将进程看作为单线程进程时，死锁进程的解决方法同样适用于同一进程的线程死锁和不同进程的线程死锁。

3.10 本章小结

- 死锁是多个并发进程因竞争资源及进程执行顺序非法而造成的一种状态。系统产生死锁的四个必要条件是互斥条件、占有且等待条件、不剥夺条件和循环等待条件。解决死锁的方法一般可分为预防、避免、检测和解除等四种。
- 预防是采用某种策略，限制并发进程对资源的请求，从而使得死锁的必要条件在系统执行的任何时间都不满足，例如，可以采用静态分配策略、抢占资源和层次分配策略来预防死锁。
- 避免则是指系统在分配资源时，根据资源的使用情况提前做出预测，从而避免死锁的发生。例如，可以采用银行家算法来避免死锁。
- 检测是指系统设有专门的机构，当死锁发生时，该机构能够检测到死锁发生，并精确地确定与死锁有关的进程和资源，通常可以用进程资源分配图来检测死锁。

3.10 本章小结

- 解除是与检测相配套的一种措施，用于将进程从死锁状态下解脱出来。可以采取重启系统、撤销所有涉及死锁进程、逐个撤销涉及死锁的进程、抢夺资源等方法来解除死锁。
- 所有解决死锁的方法都各有其优缺点。与其将操作系统机制设计为只采用其中策略，还不如在不同情况下使用不同的策略更有效。
- 最后介绍了线程死锁，与进程死锁比较，线程死锁分为属于同一进程的线程死锁和属于不同进程的线程死锁。当将进程看作为单线程进程时，死锁进程的解决方法同样适用于同一进程的线程死锁和不同进程的线程死锁。