第 8 章：死锁

第 8 章：死锁

- 死锁示例 死锁特征 资源分配图
  处理死锁的方法
-
-
-
  - 死锁预防 死锁避免 死锁检测和从死锁中恢复
  -
  -

本章目标

- 说明使用互斥锁时如何发生死锁。定义描述死锁的四个必要条件。在资源分配图中识别死锁情况。评估防止死锁的四种不同方法。应用庄家算法来避免死锁。
- 应用死锁检测算法。评估从死锁中恢复的方法。

多线程应用程序中的死锁

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

- 线程的运行顺序取决于 CPU 计划程序如何调度它们
- 此示例说明了这样一个事实，即很难识别和测试仅在某些调度情况下可能发生的死锁。

具有锁排序的死锁示例

```
void transaction（Account from，  Account to，  double amount）
{

    互斥锁 lock1，  lock2;锁
    1 = get_lock（从）;锁 2
    = get_lock（to）;获取
     （锁定 1）；


        获取（锁定 2）；

            withdraw（from，  金
            额）;deposit（to，  amount）；

    释放 （lock2）;释放
     （lock1）；

}
```

事务 1 和 2 同时执行。交易 1 将 25 美元从账户 A 转移到账户 B，交易 2 将 50 美元从账户 B 转移到账户 A

系统模型

- 系统由资源组成
- 资源类型 R1、R2、. . .、Rm

  CPU 周期、内存空间、文件、I/O 设备、信号量
- 每个资源类型 Ri 都有 Wi 实例。
- 每个进程 Pi 使用的资源如下：
  - 请求使
  - 用释放
  -

死锁特征描述

如果同时满足以下四个条件，则可能会出现涉及多个进程的死锁 – 它们是必要条件，但不是充分条件

- 互斥：一次只有一个进程可以使用一个资源
- Hold and wait：持有至少一个资源的进程正在等待获取其他进程持有的其他资源
- 无抢占：资源只能在该进程完成其任务后由持有该资源的进程自愿释放
- 循环等待：存在一组 {P0， P1， …， Pn} 等待进程，使得 P0 正在等待 P1 持有的资源，P1 正在等待 P2 持有的资源，…，Pn–1 正在等待 Pn 持有的资源，Pn 等待 P0 持有的资源。

资源分配图

一组顶点 V 和一组边 E。

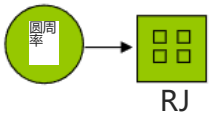
- V 分为两种类型：
  - P = {P1， P2， …， Pn}，由系统中的所有进程组成的集合
  - R = {R1， R2， …， Rm}，由系统中的所有资源类型组成的集合
- 请求边 – 有向边 Pi → Rj 赋值边 边 – 有向边 Rj → Pi
-

资源分配图 （续）

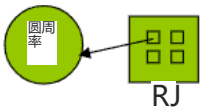
- 过程

- 具有 4 个实例的资源类型

- Pi 请求 Rj 的实例

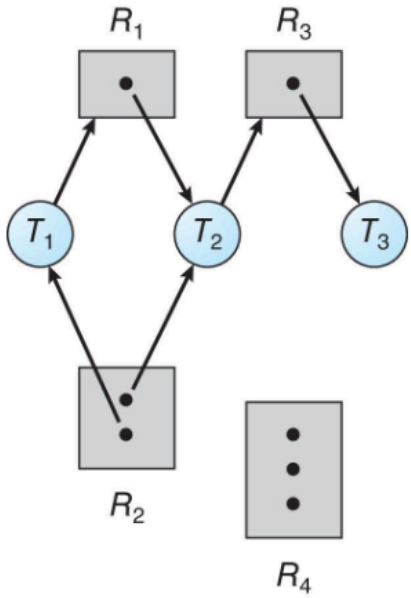


- Pi 持有 Rj 的实例

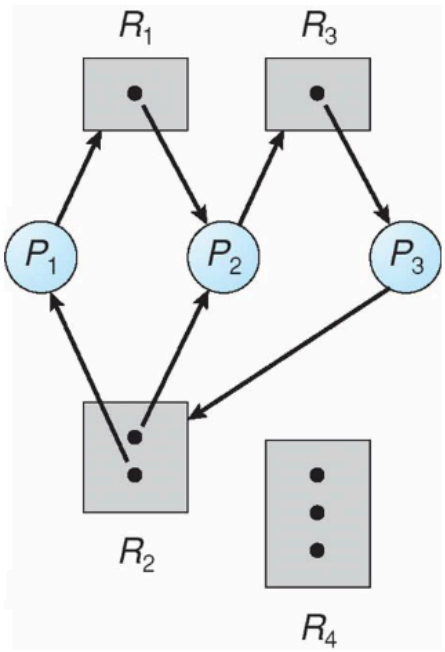


资源分配图示例

- R1 的一个实例 R2 的两个实例
- R3 的一个实例 R4 的三个实例
- T1 包含一个 R2 实例，正在等待 R1 的一个实例 T2 包含一个 R1 实例，一个 R2 实例，并且正在等待 R3 的实例
- T3 包含 R3 的一个实例



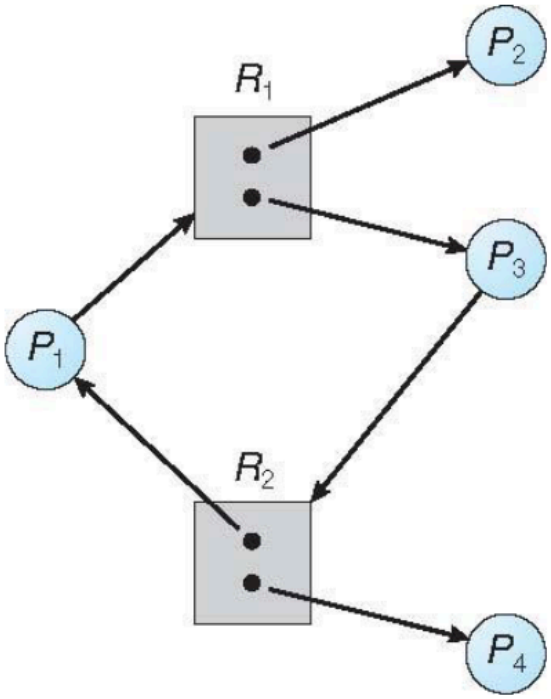
具有死锁的 Resource Allocation 图形



存在循环

- P1 → R1 → P2 → R3 → P3 → R2 → P1
- P2 → R3 → P3 → R2 → P2
- 进程 P1、P2 和 P3 死锁

具有循环但没有死锁的图形



存在一个循环

- P1 → R1 → P3 → R2 → P1

- 但是，没有死锁。请注意，线程 P4 可能会释放其资源类型 R2 的实例。然后，可以将该资源分配给 P3，从而打破循环。

基本事实

- 如果图形不包含任何周期 □ 无死锁
- 如果图形包含循环 □ 则系统可能处于死锁状态，也可能不处于死锁状态
  - 如果每个资源类型只有一个实例，则 deadlock如果每个资源类型有多个实例，则可能出现死锁
  -

处理死锁的方法

- 确保系统永远不会进入死锁状态：
  - 死锁预防：它提供了一组方法来确保至少一个必要条件不能成立
  - Deadlock avoidance：这需要提前提供有关进程在其生命周期内将请求和使用哪些资源的其他信息。在此类知识范围内，操作系统可以决定每个资源请求是否应等待进程
- 死锁检测 - 允许系统进入死锁状态，定期检测是否存在死锁，然后从中恢复许多商业操作系统，特别是台式机、笔记本电脑和智能手机的操作系统，由于开销而忽略了死锁问题，并假装系统中从未发生过死锁
-
  - 这将导致系统的性能下降，因为资源被无法运行的进程占用，并且越来越多的进程在请求资源时将进入死锁状态 – 手动重新启动系统

死锁预防

限制可以发出请求的方式

- 互斥 – 可共享资源（例如，只读文件）不需要;但它必须适用于不可共享的资源
- Hold and Wait – 必须保证每当进程请求资源时，它不持有任何其他资源
  - 要求每个进程在开始执行之前请求并被分配其所有资源，或者仅在进程没有资源时才请求资源缺点 - 资源利用率低，并且可能匮乏
  -
- 无抢占 –
  - 如果一个持有某些资源的进程请求另一个无法立即分配给它的进程，则当前持有的所有资源都将被释放 被抢占的资源添加到该进程正在等待的资源列表中 只有当它可以重新获得所有旧资源以及当前请求的新资源时，进程才会重新启动
  -
  -
  - 这只能应用于状态可以轻松保存和恢复的资源，例如寄存器、内存空间和数据库事务。它通常不能应用于锁和信号量等资源

死锁预防（续）

- 循环等待 – 对所有资源类型施加总排序，并要求每个进程按枚举的递增顺序请求资源 – R = {<R1， R2， …， Rm>}

  这就要求一个进程不能在请求资源 Ri 之前请求资源 Rj if j > i 这可以通过矛盾来证明

-
  - 设循环等待中涉及的进程集为 P = {<P0， P1， …， Pn>}，其中 Pi 正在等待由进程 Pi+1 持有的资源 Ri，因此 Pn 正在等待由 P0 持有的资源 Rn 。
  - 由于进程 Pi+1 在请求资源 Ri+1 时持有资源 Ri，因此所有 i 都必须有 Ri < Ri+1。
  - 这意味着 R0 < R1 < R2 …< Rn < R0 R0 < R0，这是不可能的，因此不能有循环等待
  -

循环等待

- 使循环等待条件无效是最常见的。只需为每个资源（即互斥锁）分配一个唯一的编号。必须按顺序获取资源。
-
-
- 如果:

```
first_mutex = 1
second_mutex = 5
```

thread_two 的代码不能编写如下：

```
                                                    nction */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

死锁避免

要求系统具有一些额外的先验信息可用

例如，在了解了每个进程的 request 和 release 的完整序列后，系统可以决定每个请求是否应该等待以避免将来可能出现的死锁。

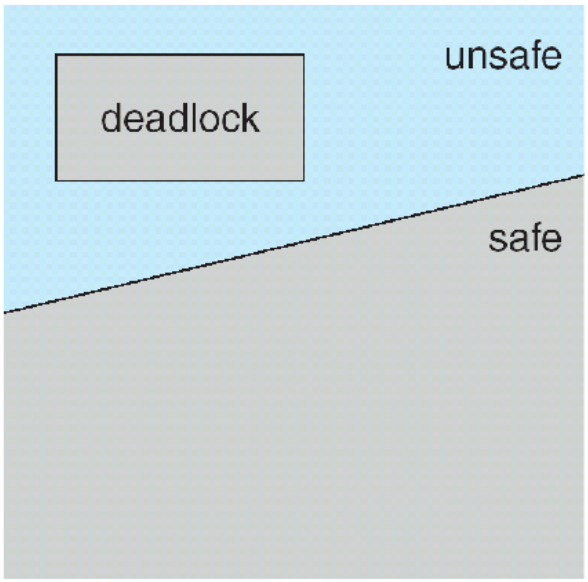
- 最简单、最有用的模型要求每个进程声明它可能需要的每种类型的最大资源数
- 死锁避免算法动态检查资源分配状态，以确保循环等待条件永远不会存在
- 资源分配状态由 （1） 个可用资源和 （2） 个已分配资源的数量以及 （3） 进程的最大需求量定义

安全状态

- 当进程请求可用资源时，系统必须决定此类分配是否会使系统处于安全状态
- 如果存在由系统中的所有进程组成的安全序列 <P1， P2， ...， Pn> 则系统处于安全状态，以便对于每个 Pi，Pi 仍然可以请求的资源（基于先前的声明）可以通过当前可用资源加上所有 Pj 持有的资源来满足，其中 j < i。那是：
  - 如果 Pi 资源需求不能立即可用，则 Pi 可以等待所有 Pj 都完成
  - 当 Pj 完成后，Pi 可以获取所需的资源，执行，返回分配的资源，然后终止
  - 当 Pi 终止时，Pi +1 可以获取其需要的资源，依此类推
- 如果不存在这样的序列，则称系统状态为不安全。

基本事实

- 如果系统处于安全状态 □ 没有死锁
- 如果系统处于不安全状态， 死锁的可能性
- 避免 □ 确保系统永远不会进入不安全状态
  - 在此方案中，如果进程请求当前可用的资源，它可能仍必须等待 （如果分配导致不安全状态）。资源利用率可能低于其他情况
  -



避障算法

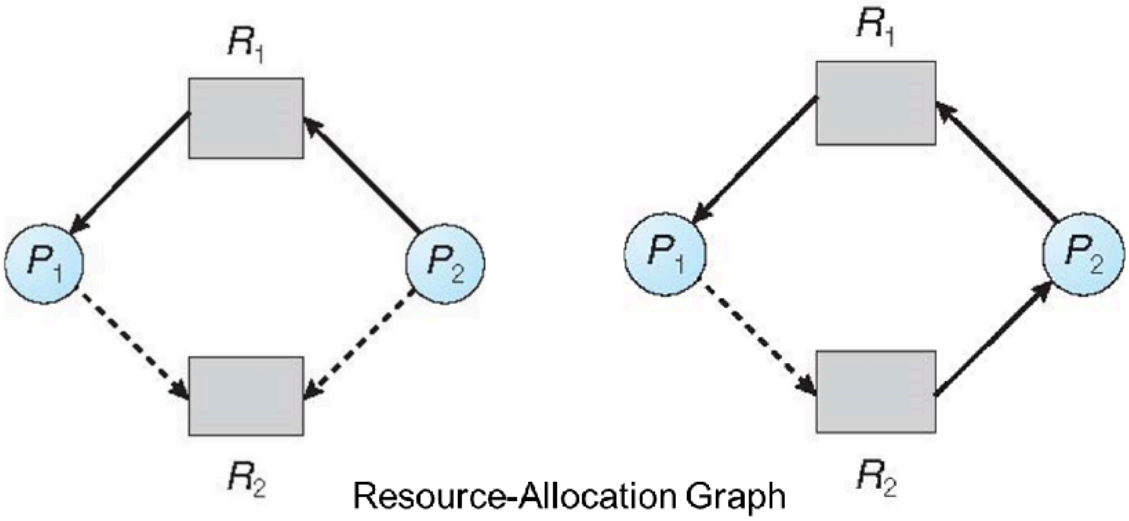
- 资源类型的单个实例
  - 使用资源分配图
- 资源类型的多个实例
  - 使用庄家算法

资源分配图 Scheme

- Claim edge Pi → Rj 表示进程 Pi 可以请求资源 Rj;由虚线表示
- 当进程请求资源时，声明边缘转换为请求边缘
- 将资源分配给流程时，请求边缘将转换为分配边缘
- 当进程释放资源时，分配边缘将重新转换为声明边缘
- 必须在系统中先验地声明资源

资源分配图算法

- 假设进程 Pi 请求资源 Rj
- 仅当将请求边缘转换为分配边缘不会导致在资源分配图中形成循环时，才能授予该请求



Resource-Allocation Graph

Banker’ s Algorithm

- 多个实例
- 每个进程都必须声明一个先验的最大使用量
- 当进程请求资源时，它可能必须等待 – 检查此分配是否导致安全状态
- 当进程获取其所有资源时，它必须在使用后的有限时间内返回它们
- 这类似于银行贷款系统，它有一个最高金额，即总额，可以一次借给一组企业，每个企业都有信用额度。

Banker's Algorithm 的数据结构

设 n = 进程数，m = 资源类型数。

- 可用：长度为 m 的向量。如果可用 [j] = k，则有 k 个资源类型 Rj 的实例可用
- 最大值：n x m 矩阵。如果 Max [i，j] = k，则进程 Pi 最多可以请求 k 个资源类型 Rj 的实例
- 分配：n x m 矩阵。如果 Allocation[i，j] = k，则当前为 Pi 分配了 Rj 的 k 个实例
- 需要：n x m 矩阵。如果 Need[i，j] = k，则 Pi 可能需要 k 个 Rj 实例才能完成其任务

需求 [i，j] = 最大值 [i，j] – 分配 [i，j]

安全算法

1. 设 Work 和 Finish 分别为长度为 m 和 n 的向量。初始化：

   工时 = 可用完成时间 [i] = false，
   对于 i = 0， 1， ...， n- 1

2. 找到一个 i，使两者都：

   （a） 完成 [i] = false （b）
   需要 □ 工作 如果不存在这样
   的 i，请转到第 4 步

3. 工作 = 工作 + 分配i 完成 [i] =
   真

   转到步骤 2

4. 如果所有 i 的 Finish [i] == true，则系统处于安全状态，否则不安全

进程 Pi 的资源请求算法

请求 = 进程 Pi 的请求向量。如果 Requesti [j] = k，则进程 Pi 需要 k 个资源类型 Rj 的实例

1. 如果 Requesti □ Needi 进入步骤 2。否则，引发错误条件，因为进程已超出其最大声明

2. 如果 Requesti □ 可用，请转到步骤 3。否则，Pi 必须等待，因为资源不可用

3. 通过修改状态，假装已将请求的资源分配给 Pi，如下所示：

   可用 = 可用 – 请求;分配i = 分配i + 请求i;needi = needi – 请求;

- 运行安全算法： 如果安全 □ 资源可以分配给 Pi 如果不安全 □ Pi 必须等待，并恢复旧的资源分配状态
-

Banker's Algorithm 示例

- 5 个进程 P0 到 P4;3 种资源类型：

  A （10 个实例）、B （5 个实例） 和 C （7 个实例）

T0 时间的快照：

	分配	麦克斯	可用
	A B C	A B C	A B C
P0 系列	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2 系列	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4 系列	0 0 2	4 3 3	

示例（续）

- 矩阵 Need 的内容定义为 Max – Allocation

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

- 由于 P1、P3、P4、P2、P0 <序列满足安全标准，因此系统处于安全状态>

示例（续）

- 5 个进程 P0 到 P4;3 种资源类型：A（10 个实例）、B（5 个实例）和 C（7 个实例）

T0 时间的快照：

	分配	麦克斯	可用	需要
	A B C	A B C	A B C	A B C
P0 系列	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2 系列	3 0 2	9 0 2		6 0 0
P3	2 1 1	2 2 2		0 1 1
P4 系列	0 0 2	4 3 3		4 3 1

由于 P1、P3、P4、P2、P0 <序列满足安全标准，因此系统处于安全状态>

示例：P1 请求 （1,0,2）

- 检查请求 □ Available，即 （1,0,2） □ （3,3,2） □ true

	分配	需要	可用
	A B C	A B C	A B C
P0 系列	0 1 0	7 4 3	2 3 0
P1	3 0 2	0 2 0	
P2 系列	3 0 2	6 0 0	
P3	2 1 1	0 1 1	
P4 系列	0 0 2	4 3 1	

- 执行安全算法表明，< P1、P3、P4、P0、P2 序列>满足安全要求

- P4 可以批准 （3,3,0） 的请求吗？ – 资源不可用

- 可以批准 P0 的 （0,2,0） 请求吗？ – 状态不安全

死锁检测

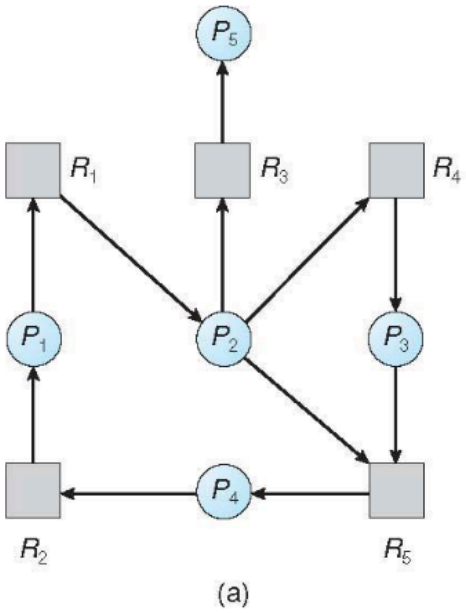
如果系统不使用死锁预防或死锁避免算法，则可能会发生死锁情况。在此环境中，系统可以提供

- 一种检查系统状态以确定是否可能发生死锁的算法
- 从死锁中恢复的算法

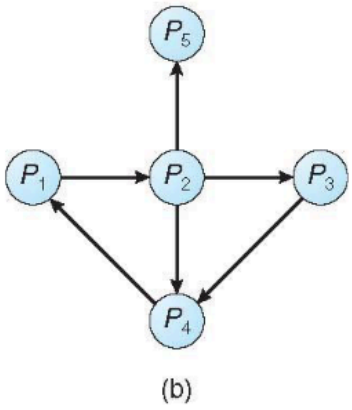
每种资源类型的单个实例

- 维护等待图节点是进程
  - 如果 Pi 正在等待 Pj，则 Pi → Pj
- 定期调用在图中搜索循环的算法。如果存在循环，则存在死锁
- 检测图中循环的算法需要 n2 次运算的顺序，其中 n 是图中的顶点数
- 等待图方案不适用于每种资源类型具有多个实例的资源分配系统

Resource-Allocation 图形和 Wait-for 图形



资源分配图



相应的等待图

一个资源类型的多个实例

- Available：长度为 m 的向量表示每种类型的可用资源数量
- 分配：n x m 矩阵定义当前分配给每个进程的每种类型的资源数量
- 请求：n x m 矩阵表示每个进程的当前请求。如果请求 [i][j] = k，则进程 Pi 正在请求 k 个资源类型 Rj 的实例。

检测算法

1. 设 Work 和 Finish 为长度为 m 和 n 的向量，分别初始化：

   （a） 工作 = 可用 （b） 对于 i = 1,2， ...， n，如果分配i □ 0，则
   完成 [i] = false;否则，Finish[i] = true

2. 找到一个索引 i，使得两者都：

   （a） 完成[i] == false （b） 请求 □ 工作 如果不存在这样的 i，请转到步骤 4

3. 工作 = 工作 + 分配i 完成 [i] = 真

   转到步骤 2

4. 如果 Finish[i] == false，对于某些 i，1 □ i □ n，则系统处于死锁状态。此外，如果 Finish[i] == false，则 Pi 死锁

此算法需要 O（m x n2） 次操作来检测系统是否处于死锁状态

检测算法示例

- 五个过程 P0 到 P4;三种资源类型 A（7 个实例）、B （2 个实例）和 C （6 个实例）

- T0 时间的快照：

	分配	请求	可用
	A B C	A B C	A B C
P0 系列	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2 系列	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4 系列	0 0 2	0 0 2	

- 序列 <P0， P2， P3， P1， P4> 将导致所有 i 的 Finish[i] = true

示例（续）

- P2 请求类型 C 的额外实例

	请求 A B C
P0 系列	0 0 0
P1	2 0 2
P2 系列	0 0 1
P3	1 0 0
P4 系列	0 0 2

- 系统状态?
  - 可以回收进程 P0 持有的资源，但资源不足，无法完成其他进程;请求
  - 存在死锁，由进程 P1、P2、P3 和 P4 组成

检测算法使用

- 调用的时间和频率取决于：
  - 死锁可能发生的频率如何？当死锁发生时，有多少个进程将受到影响 □ 每个不相交周期一个
  -
- 如果任意调用检测算法，则资源图中可能会有很多循环;我们无法判断许多死锁进程中的哪一个 “导致” 了死锁。
- 为每个资源请求调用死锁检测算法将产生相当大的计算开销。
  - 一种更便宜的替代方法是按定义的时间间隔调用算法 - 例如，每小时一次，或者在 CPU 利用率低于 40% 时调用

从死锁中恢复：进程终止

- 中止所有死锁进程：这显然打破了死锁循环，但代价很高
- 一次中止一个进程，直到消除死锁循环：这也会产生相当大的开销，因为在每个进程中止后，需要运行死锁检测算法
- 我们应该选择以什么顺序中止？– 许多因素：
  1. 流程的优先级
  2. 计算了多长时间的过程，需要多长时间才能完成？
  3. 进程已使用的资源
  4. 流程需要完成的资源
  5. 需要终止多少个进程？
  6. 流程是交互式的还是批处理的？

从死锁中恢复：资源抢占

连续从进程中抢占某些资源，并将这些资源提供给其他进程，直到打破死锁循环

- 选择受害者 – 最小化成本（哪些资源和哪些进程将被抢占）
- Rollback – 返回到某个安全状态，从该状态重新启动进程
- 饥饿 – 可能始终选择相同的进程作为受害者，包括成本因子中的回滚次数可能有助于减少饥饿

第 8 章结束