第 6 章 死锁

多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止，这种情况叫死锁。本章将对 Java 多线程编程中可能出现死锁的情况进行详细的讲解，以及如何采用 MTRAT 来检查死锁。

6.1 死锁概述

线程又称为轻量级进程，它和进程一样拥有独立的执行控制，由操作系统负责调度，区别在于线程没有独立的存储空间，而是和所属进程中的其它线程共享一个存储空间，这使得线程间的通信较进程简单。编写多线程序时，必须注意每个线程是否干扰了其他线程的工作。每个进程开始生命周期时都是单一线程，称为“主线程”，在某一时刻主线程会创建一个对等线程。如果主线程停滞则系统就会切换到其对等线程。和一个进程相关的线程此时会组成一个对等线程池，一个线程可以杀死其任意对等线程。

因为每个线程都能读写相同的共享数据。这样就带来了新的麻烦：由于数据共享会带来同步问题，进而会导致死锁的产生。

由多线程带来的性能改善是以可靠性为代价的，主要是因为有可能产生线程死锁。死锁是这样一种情形：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不能正常运行。简单的说就是:线程死锁时，第一个线程等待第二个线程释放资源，而同时第二个线程又在直接或间接等待第一个线程释放资源。这里举一个通俗的例子：如在人行道上两个人迎面相遇，为了给对方让道，两人同时向一侧迈出一步，双方无法通过，又同时向另一侧迈出一步，这样还是无法通过。假设这种情况一直持续下去，这样就会发生死锁现象。

更形象的例子如下： 五个哲学家围坐在一圆桌旁，每人的两边放着一支筷子，共五支筷子。大家边讨论问题边用餐。并规定如下的条件是：

1）每个人只有拿起位于自己两边的筷子，合成一双才可以用餐。

2）用餐后每人必须将两只筷子放回原处。

我们可以想象，如果每个哲学家都彬彬有礼，并且高谈阔论，轮流吃饭，则这种融洽的气氛可以长久地保持下去。但是可能出现这样一种情景：当每个人都拿起自己左手边的筷子，并同时去拿自己右手边的筷子时，会发生什么情况：五个人每人拿着一支筷子，盯着自己右手边的那位哲学手里的一支筷子，处于僵持状态。这就是发生了“线程死锁”。

多个线程竞争共享资源时可能出现的一种系统状态：线程 1 拥有资源 1，并等待资源 2，

而线程 2 拥有资源 2，并等待资源 3,…,以此类推，线程 n 拥有资源 n-1,并等待资源 1。在这种状态下，各个线程互不相让，永远进入一种等待状态。于是出现了死锁的现象。

虽然线程死锁只是系统的一种状态，该状态出现的机会可能会非常小，但简单的测试往往无法发现。遗憾的是 Java 语言也没有有效的方法可以避免或检测死锁。

6.2 死锁示例

下面给出出现死锁的一些案例。

【6-1】由 Mtrat 提供的线程死锁的案例

```
 class T3 extends Thread{
StringBuffer L1;
StringBuffer L2;
public T3(StringBuffer L1, StringBuffer L2) {
   this.L1 = L1;
   this.L2 = L2;
}
public void run() {
   synchronized (L1) {
      synchronized (L2) {
      }
   }
}
 }
public class Deadlock{
void harness2() throws InterruptedException
{
   StringBuffer L1 = new StringBuffer("L1");
   StringBuffer L2 = new StringBuffer("L2");

   Thread t1 = new T3(L1, L2);
   Thread t2 = new T3(L2, L1);
   t1.start();
   t2.start();
   t1.join();
   t2.join();
}
public static void main(String args) throws InterruptedException
{
   Deadlock dlt = new Deadlock();
   dlt.harness2();
}
}
```

在类 Deadlock 的 harness2 方法中，类 Deadlock 的两个实例被创建，作为参数传递到类 T3 的构造函数中。在类 T3 的 run 方法中，线程会依次获得这个两个对象的锁，然后以相反的顺序释放这两个锁。由于两个 StringBuffer 实例以不同的顺序传递给类 T3，两个线程会以不同的顺序获得这两个锁。这样，死锁就出现了。

【6-2】哲学家吃饭的案例

```
// ChopStick.java
public class ChopStick
{
    private String name;
    public ChopStick(String name)
    {
        this.name = name;
    }
    public String getNumber()
    {
        return name;
    }
}

// Philosopher.java
import java.util.*;
public class Philosopher extends Thread
{
    private ChopStick leftChopStick;
    private ChopStick rightChopStick;
    private String name;
    private static Random random = new Random();
    public Philosopher(String name, ChopStick leftChopStick,
            ChopStick rightChopStick)
    {
        this.name = name;
        this.leftChopStick = leftChopStick;
        this.rightChopStick = rightChopStick;
    }
    public String getNumber()
    {
        return name;
    }
    public void run()
    {
        try {
```

```
                sleep(random.nextInt(10));
            } catch (InterruptedException e) {
            }
            synchronized (leftChopStick) {
                System.out.println(this.getNumber() + " has "
                        + leftChopStick.getNumber() + " and wait for "
                        + rightChopStick.getNumber());
                synchronized (rightChopStick) {
                    System.out.println(this.getNumber() + " eating");
                }
            }
        }
        public static void main(String args[])
        {
            // 建立三个筷子对象
            ChopStick chopStick1 = new ChopStick("ChopStick1");
            ChopStick chopStick2 = new ChopStick("ChopStick2");
            ChopStick chopStick3 = new ChopStick("ChopStick3");
    ChopStick chopStick4 = new ChopStick("ChopStick4");
    ChopStick chopStick5 = new ChopStick("ChopStick5");

            // 建立哲学家对象，并在其两边摆放筷子。
            Philosopher philosopher1 = new Philosopher("philosopher1", chopStick1,
                    chopStick2);
            Philosopher philosopher2 = new Philosopher("philosopher2", chopStick2,
                    chopStick3);
            Philosopher philosopher3 = new Philosopher("philosopher3", chopStick3,
                    chopStick4);
            Philosopher philosopher4= new Philosopher("philosopher4", chopStick4,
                    chopStick5);
            Philosopher philosopher5 = new Philosopher("philosopher5", chopStick5,
                    chopStick1);

            // 启动五个线程
            philosopher1.start();
            philosopher2.start();
            philosopher3.start();
```

```
        philosopher4.start();
        philosopher5.start();
            }
        }
```

运行结果如下：

```
philosopher1 has ChopStick1 and wait for ChopStick2
philosopher1 eating
philosopher2 has ChopStick2 and wait for ChopStick3
philosopher2 eating
philosopher5 has ChopStick5 and wait for ChopStick1
philosopher5 eating
philosopher3 has ChopStick3 and wait for ChopStick4
philosopher3 eating
philosopher4 has ChopStick4 and wait for ChopStick5
philosopher4 eating
```

本例中由于采用了干预，避免死锁。如在哲学家问题中，如果规定每个哲学家必须在拿到自己左边的筷子后，才能去拿自己右边的筷子，那么讲很容易形成一个请求环，因此也就可能形成死锁。但如果我们规定其中的某一个哲学家只能在拿到自己右边筷子的前提下，才能去拿左边的筷子，那么就不会形成请求环，从而也不会出现死锁。

【6-3】另一个例子

```
public class AnotherDeadLock {
    public static void main(String[] args) {
        final Object resource1 = "resource1";
        final Object resource2 = "resource2";
        // t1 tries to lock resource1 then resource2
        Thread t1 = new Thread() {
            public void run() {
                // Lock resource 1
                synchronized (resource1) {
                    System.out.println("Thread 1: locked resource 1");

                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {
                    }

                    synchronized (resource2) {
                        System.out.println("Thread 1: locked resource 2");
                    }
                }
            }
        };
```

```
        // t2 tries to lock resource2 then resource1
        Thread t2 = new Thread() {
          public void run() {
            synchronized (resource2) {
              System.out.println("Thread 2: locked resource 2");

              try {
                Thread.sleep(50);
              } catch (InterruptedException e) {
              }
              synchronized (resource1) {
                System.out.println("Thread 2: locked resource 1");
              }
            }
          }
        };

        // If all goes as planned, deadlock will occur,
        // and the program will never exit.
        t1.start();
        t2.start();
      }
    }
```

该例中，对锁作了一个调整，变得更普遍了。本质上和例 6-1 是一致的。程序中存在死锁的问题。

6.3 避免死锁和死锁诊断

一般来说，要出现死锁必须同时具备四个条件。因此，如果能够尽可能地破坏这四个条件中的任意一个，就可以避免死锁的出现。

1） 互斥条件。即至少存在一个资源，不能被多个线程同时共享。如在哲学家问题中，一支筷子一次只能被一个哲学家使用。

2） 至少存在一个线程，它拥有一个资源，并等待获得另一个线程当前所拥有的资源。如在哲学家聚餐问题中，当发生死锁时，至少有一个哲学家拿着一支筷子，并等待取得另一个哲学家拿着的筷子。

3） 线程拥有的资源不能被强行剥夺，只能有线程资源释放。如在哲学家问题中，如果允许一个哲学家之间可以抢夺筷子，则就不会发生死锁问题。

4） 线程对资源的请求形成一个圆环。即：线程 1 拥有资源 1，并等待资源 2，而线程 2 拥有资源 2，并等待资源 3,…,以此类推，最后线程 n 拥有资源 n-1,并等待资源 1，从而构成了一个环。这是构成死锁的一个重要条件。如在哲学家问题中，如果规定每个哲学家必须在拿到自己左边的筷子后，才能去拿自己右边的筷子，那么讲很容易形成一个请求环，因此也就可能形成死锁。但如果我们规定其中的某一个哲学家只能在拿到自己右边筷子的前提下，才能去拿左边的筷子，那么就不会形成请求环，从而也不会出现死锁。

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止线程在处于等待状态的情况下占用资源,在系统运行过程中，对线程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配 。因此，对资源的分配要给予合理的规划。下面有两种方法可以有效避免死锁。

1）有序资源分配法

这种算法资源按某种规则系统中的所有资源统一编号（例如打印机为 1、磁带机为 2、磁盘为 3、等等），申请时必须以上升的次序。系统要求申请线程：

1、对它所必须使用的而且属于同一类的所有资源，必须一次申请完；

2、在申请不同类资源时，必须按各类设备的编号依次申请。例如：进程 PA，使用资源的顺序是 R1，R2； 进程 PB，使用资源的顺序是 R2，R1；若采用动态分配有可能形成环路条件，造成死锁。

采用有序资源分配法：R1 的编号为 1，R2 的编号为 2；

PA：申请次序应是：R1，R2。

PB：申请次序应是：R1，R2。

这样就破坏了环路条件，避免了死锁的发生。

2）银行算法

避免死锁算法中最有代表性的算法是 Dijkstra E.W 于 1968 年提出的银行家算法：

在系统运行过程中，对线程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配。

系统安全序列的概念：对于线程序列{P1，…，Pn}是安全的话，如果对于每一个线程

Pi(1≤i≤n)，它以后尚需要的资源量不超过系统当前剩余资源量与所有线程 Pj(j < i)当前占有资源量之和，系统则处于安全状态(安全状态一定是没有死锁发生的)。

银行家算法的中心思想是在安全状态下系统不会进入死锁，不安全状态可能进入死锁。在进行资源分配之前，先计算分配的安全性，判断是否为安全状态。

该算法需要检查申请者对资源的最大需求量，如果系统现存的各类资源可以满足申请者的请求，就满足申请者的请求。

这样申请者就可很快完成其计算，然后释放它占用的资源，从而保证了系统中的所有进程都能完成，所以可避免死锁的发生。

死锁排除的方法

1、撤消陷于死锁的全部线程；

2、逐个撤消陷于死锁的线程，直到死锁不存在；

3、从陷于死锁的线程中逐个强迫放弃所占用的资源，直至死锁消失。

4、从另外一些线程那里强行剥夺足够数量的资源分配给死锁线程，以解除死锁状态

6.4 减小锁的竞争和粒度

竞争性的锁将会导致两种损失：可伸缩性和性能，所以减少锁的竞争能够改进性能和可伸缩性。

访问独占锁守护的资源是串行的——一次只能有一个线程访问它。当然，我们有很好的理由使用锁，比如避免数据过期，但是这样的安全性是用很大的代价换来的。对锁长期的竞争会限制可伸缩性。并发程序中，对可伸缩性首要的威胁是独占的资源锁。

有两个原因影响着锁的竞争性：锁被请求的频率，以及每次持有该锁的时间。如果这两者的乘积足够小，那么大多数请求锁的尝试都是非竞争的，这样竞争性的锁将不会成为可伸缩性巨大的阻碍。但是，如果这个锁的请求量很大，线程将会阻塞以等待锁；在极端的情况下，处理器将会闲置，即使仍有大量工作等着完成。

有 3 种方式来减少锁的竞争：

- 减少持有锁的时间；
- 减少请求锁的频率；
- 用协调机制取代独占锁，从而允许更强的并发性。

6.4.1 缩小锁的范围

减小竞争发生可能性的有效方式是尽可能缩短把持锁的时间。这可以通过把与锁无关的代码移出 synchronized 块来实现，尤其是那些花费“昂贵”的操作，以及那些潜在的阻塞操作，比如 I/O 操作。

我们很容易观察到长时间持有“热门”锁究竟是如何限制可伸缩性的;对于例 6-4 的例子，无论你拥有多少个空闲处理器,如果一个操作持有锁超过 2 毫秒并且每一个操作都需要那个锁，吞吐量不会超过每秒 500 个操作。但是如果减少持有这个锁的时间到 1 毫秒，那将能够把这个与锁相关的吞吐量提高到每秒 1000 个操作。

【6-4】AttributeStore

```
public class AttributeStore {
@GuardedBy("this") private final Map<String, String>
attributes = new HashMap<String, String>();
public synchronized boolean userLocationMatches(String name,
            String regexp) {
  String key = "users." + name + ".location";
  String location = attributes.get(key);
    if (location == null)
    return false;
     else
    return Pattern.matches(regexp, location);
  }
}
```

将例 6-4 的代码改称下面 6-5 的形式

【6-5】BetterAttributeStore

```
public class BetterAttributeStore {
    @GuardedBy("this") private final Map<String, String>
      attributes = new HashMap<String, String>();
     public boolean userLocationMatches(String name, String regexp) {
     String key = "users." + name + ".location";
     String location;
     synchronized (this) {
        location = attributes.get(key);
     }
     if (location == null)
        return false;
      else
     return Pattern.matches(regexp, location);
    }
}
```

缩小 userLocationMatches 方法中锁守护的范围，这大大减少了调用中遇到锁住情况的次数。串行化的代码少了，减少了占有锁的时间。

尽管缩小 synchronized 块能够提高可伸缩性，synchronized 块可以变得极小——需要原子化的操作（比如在限定约束的情况下更新多个变量）必须包含在一个 synchronized 块中。并且因为同步的开销非零，保证正确的情况下，如果把一个 synchronized 块分拆成多个 synchronized 块，在某些时刻反而会对性能产生反作用。

6.4.2 减小锁的粒度

减小持有锁的总体时间比例的另一种方式是让线程减少调用它的频率。可以通过分拆锁（lock splitting）和分离锁（lock striping）来实现，也就是采用相互独立的锁，守卫多个独立的状态变量，在改变之前，它们都是由一个锁守护的。这些技术减小了锁发生时的粒度，潜在实现了更好的可伸缩性。但是使用更多的锁同样会增加死锁的风险。

例如：待分拆锁的候选程序

【6-6】// ServerStatus.java

```
public class ServerStatus {
   @GuardedBy("this") public final Set<String> users;
   @GuardedBy("this") public final Set<String> queries;
   public synchronized void addUser(String u) { users.add(u); }
   public synchronized void addQuery(String q) { queries.add(q); }
   public synchronized void removeUser(String u) {
   users.remove(u);
   }
   public synchronized void removeQuery(String q) {
   queries.remove(q);
   }
}
```

【6-7】拆分后的锁

```
public class ServerStatus {
   @GuardedBy("users") public final Set<String> users;
   @GuardedBy("queries") public final Set<String> queries;
   public void addUser(String u) {
   synchronized (users) {
    users.add(u);
   }
   }
   public void addQuery(String q) {
   synchronized (queries) {
```

```
            queries.add(q);
          }
          }
        }
```

中等竞争强度的锁，能够切实地把它们大部分转化成非竞争的锁，这个结果是性能和可伸缩性都期望得到的。

把一个竞争激烈的锁分拆成两个，很可能形成两个竞争激烈的锁。尽管这可以通过两个线程并发执行，取代一个线程，从而对可伸缩性有一些小的改进，但这仍然不能大幅地提高多个处理器在同一系统中并发性的前景。作为分拆锁的例子，ServerStatus 类并没有提供明显的机会来进行进一步分拆。

分拆锁有时候可以被扩展，分成可大可小加锁块的集合，并且它们归属于相互独立的对象，这样的情况就是分离锁

用于减轻竞争锁带来的影响的第三种技术是提前使用独占锁，这有助于使用更友好的并发方式进行共享状态的管理。这包括使用并发容器、读-写锁、不可变对象，以及原子变量。

读-写锁（ReadWriteLock）实行了一个多读者-单写者（multiple-reader, single-write）加锁规则：只要没有更改，那么多个读者可以并发访问共享资源，但是写者必须独占获得锁。对于多数操作都为读操作的数据结构，ReadWriteLock 与独占的锁相比，可以提供更好的并发性；对于只读的数据结构，不变性可以完全消除加锁的必要。

原子变量（参见第 3 章）提供了能够减少更新“热点域”的方式，如静态计数器、序列发生器、或者对链表数据结构头节点的引用。原子变量类提供了针对整数或对象引用的非常精妙的原子操作（因此更具可伸缩性），并且使用现代处理器提供的低层并发原语，比如比较并交换（compare-and-swap）实现。如果你的类只有少量热点域，并且该类不参与其他变量的不变约束，那么使用原子变量替代它可能会提高可伸缩性。

6.5 使用 MTRAT 诊断死锁

下面为 Mtrat 提供的关于死锁的案例

【6-7】Deadlock.java

```
package mtrat.test;
import java.util.concurrent.atomic.AtomicBoolean;
class T1 extends Thread{
      StringBuffer G;
      StringBuffer L1;
```

```
        StringBuffer L2;
        public T1(StringBuffer G, StringBuffer L1, StringBuffer L2)       {
                this.G = G;
                this.L1 = L1;
                this.L2 = L2;
        }
        public void run()     {
                synchronized (G) {
                        synchronized (L1)   {
                                synchronized (L2)   {
                                        System.out.println("Thread " + Thread.currentThread().getId()
                                                        + " : acquire " + G + ", " + L1 + ", " + L2);
                                }
                        }
                }

                Thread t3 = new T3(L1, L2);
                t3.start();
                System.out.println("Thread " + getId() + " start Thread " + t3.getId());
                try   {
                        t3.join();
                } catch (InterruptedException e){
                        e.printStackTrace();
                }
                System.out.println("Thread " + getId() + " join Thread " + t3.getId());
                synchronized (L2)   {
                        synchronized (L1)   {
                                System.out.println("Thread " + getId() + ": acquire " + L2 + ", " + L1);
                        }
                }
        }
}

class T2 extends Thread{
        StringBuffer G;
        StringBuffer L1;
        StringBuffer L2;
        public T2(StringBuffer G, StringBuffer L1, StringBuffer L2)       {
                this.G = G;
                this.L1 = L1;
                this.L2 = L2;
        }
        public void run()     {
                synchronized (G){
```

```
                synchronized (L2)  {
                    synchronized (L1)  {
                        System.out.println("Thread " + getId() + " : acquire " + G + ", "+
L2 + ", "+ L1);
                    }
                }
            }
        }
    }

    class T3 extends Thread{
        StringBuffer L1;
        StringBuffer L2;
        static AtomicBoolean locked = new AtomicBoolean(false);
        public T3(StringBuffer L1, StringBuffer L2){
            this.L1 = L1;
            this.L2 = L2;
        }

        public void run()    {
            synchronized (L1)  {
                while (!locked.compareAndSet(false, true))
                    ;
                synchronized (L2)  {
                    System.out.println("Thread  "  +  Thread.currentThread().getId()  +  "  :
Acquire " + L1 + " " + L2);
                }
                locked.compareAndSet(true, false);
            }
        }
    }

    public class Deadlock{
        // no deadlock here
        void harness1(){
            StringBuffer G = new StringBuffer("G");
            StringBuffer L1 = new StringBuffer("L1");
            StringBuffer L2 = new StringBuffer("L2");
            Thread t2=new T2(G, L1, L2);
            System.out.println ("T2 starts");
            t2.start();
            try   {
                t2.join();
            } catch (InterruptedException e){
```

```
                    e.printStackTrace();
                }
                System.out.println ("T2 join");
                System.out.println ("T1 starts");
                new T1(G, L1, L2).start();
        }

        void harness2() throws InterruptedException{
                StringBuffer L1 = new StringBuffer("L1");
                StringBuffer L2 = new StringBuffer("L2");
                Thread t1 = new T3(L1, L2);
                Thread t2 = new T3(L2, L1);

                t1.start();
                t2.start();

                t1.join();
                t2.join();
        }
        public static void main(String[] args) throws InterruptedException
        {
                Deadlock dlt = new Deadlock();
                dlt.harness1();
                dlt.harness2();
        }
    }
```

程序运行结果如下：

```
T2 starts
Thread 11 : acquire G, L2, L1
T2 join
T1 starts
Thread 12 : acquire G, L1, L2
Thread 13 : Acquire L1 L2
Thread 12 start Thread 15
Thread 15 : Acquire L1 L2
Thread 12 join Thread 15
Thread 12: acquire L2, L1
Thread 14 : Acquire L2 L1
```

采用 Mtrat 进行分析，在 Eclipse 平台下进行分析，发现了两个潜在的死锁问题。

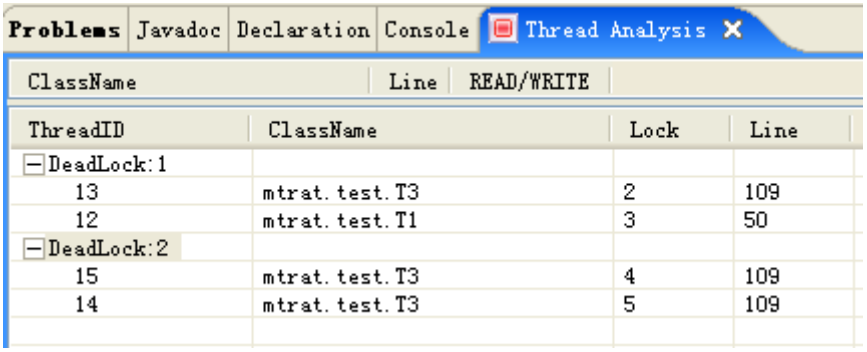

图 6-1 Mtrat 分析死锁的结果

6.6 饿死和活锁

在多线程编程过程，除了可能遇到死锁的情况之外，我们还可能遇到活锁和饿死的情况。

对于死锁来说，由于系统中两个或多个部件的集合发生阻塞，并且每个部件都等待集合中其他部件，从而使计算无法进行；典型的情况下，每个部件是一个被阻塞的线程，它等待集合中其他线程释放所掌握的资源。

什么活锁呢？活锁的产生于循环依赖，当一个线程忙于接受新任务以致它永远没有机会完成任何任务时，就会发生活锁。这个线程最终将超出缓冲区并导致程序崩溃。试想一个秘书需要录入一封信，但她一直在忙于接电话，所以这封信永远不会被录入。如果明智地使用 synchronized 关键字，则完全可以避免内存错误这种气死人的问题。

什么是饿死的概念？饿死的概念和死锁不一样，一些线程不能获得服务，而其他客户端却可以；违反了公平原则。这些不能获得服务的线程即成为饿死的线程。

产生饿死的主要原因是：在一个动态系统中，对于每类系统资源，操作系统需要确定一个分配策略，当多个线程同时申请某类资源时，由分配策略确定资源分配给线程的次序。有时资源分配策略可能是不公平的，即不能保证等待时间上界的存在。在这种情况下，即使系统没有发生死锁，某些线程也可能会长时间等待. 当等待时间给线程推进和响应带来明显影响时，称发生了线程饿死，当饿死到一定程度的线程所赋予的任务即使完成也不再具有实际意义时称该线程被饿死。举个例子，当有多个线程需要打印文件时，如果系统分配打印机的策略是最短文件优先，那么长文件的打印任务将由于短文件的源源不断到来而被无限期推迟，导致最终的饿死甚至饿死。

饿死没有其产生的必要条件，随机性很强。并且饿死可以被消除，因此也将忙式等待时发生的饿死称为活锁。

由于饿死和活锁与资源分配策略有关，因而解决饿死与活锁问题可从资源分配策略的公平性考虑，确保所有线程不被忽视。如时间片轮转算法(RR)。它将 CPU 的处理时间分成一个个时间片，就绪队列中的诸线程轮流运行一个时间片，当时间片结束时，就强迫运行程序让出 CPU，该线程进入就绪队列，等待下一次调度。同时，线程调度又去选择就绪队列中的一个线程，分配给它一个时间片，以投入运行。如此方式轮流调度。这样就可以在不考虑其他系统开销的情况下解决饿死的问题。

最后，我们来比较的看一下死锁与饿死。

死锁与饿死有一定相同点：二者都是由于竞争资源而引起的。但又有明显差别：

(1) 从线程状态考虑，死锁线程都处于等待状态，忙式等待(处于运行或就绪状态)的线程并非处于等待状态，但却可能被饿死；

(2) 死锁线程等待永远不会被释放的资源，饿死线程等待会被释放但却不会分配给自己的资源，表现为等待时限没有上界(排队等待或忙式等待)；

(3) 死锁一定发生了循环等待，而饿死则不然。这也表明通过资源分配图可以检测死锁存在与否，但却不能检测是否有线程饿死；

(4) 死锁一定涉及多个线程，而饿死或被饿死的线程可能只有一个。

(5)在饿死的情形下，系统中有至少一个线程能正常运行，只是饿死线程得不到执行机会。而死锁则可能会最终使整个系统陷入死锁并崩溃。

参考资料：

1) http://www.cqzol.com/programming/Java/200801/83962.html
2) http://book.csdn.net/bookfiles/398/index.html