
 This repository

[Pull requests](#) [Issues](#) [Gist](#)

+ ▾

 ▾

 sysuzyc / ES2016_14353404

 Watch ▾

0

 Star

1

 Fork

0

 Code

 Issues 0

 Pull requests 0

 Projects 0

 Wiki

 Pulse

 Graphs

 Settings

Branch: master ▾

ES2016_14353404 / assignment / deadlock.md

Find file

Copy path

 sysuzyc Update deadlock.md

89f1819 8 minutes ago

1 contributor

113 lines (69 sloc)5.88 KB

Raw

Blame

History







Lab4 死锁

14353404 张亚琛

1、基础知识：

死锁产生的条件

死锁是进程通信间常见的问题，我们在操作系统中也学习过了，所以，我们可以直接在下面进行总结：

死锁就是两个或者多个进程，互相请求对方占有的资源。并且死锁需要满足下面四个条件：

互斥：一个资源每次只能被一个进程使用

占有并等待：一个进程因请求资源而阻塞时，对已获得的资源保持不放

非抢占：进程已获得的资源，在未使用完之前，不能强行剥夺

循环等待：若干进程之间形成一种头尾相接的循环等待资源关系

上面的四个条件，如果我们有一个不满足的话，我们就不会产生死锁。

所以，避免死锁也是四个方面下手的：

互斥：对于非共享资源，一定要采用互斥机制

占有并等待：只允许进程在没有资源的时候申请资源

非抢占：如果申请的资源不能立即分配，则释放掉自己占有的资源

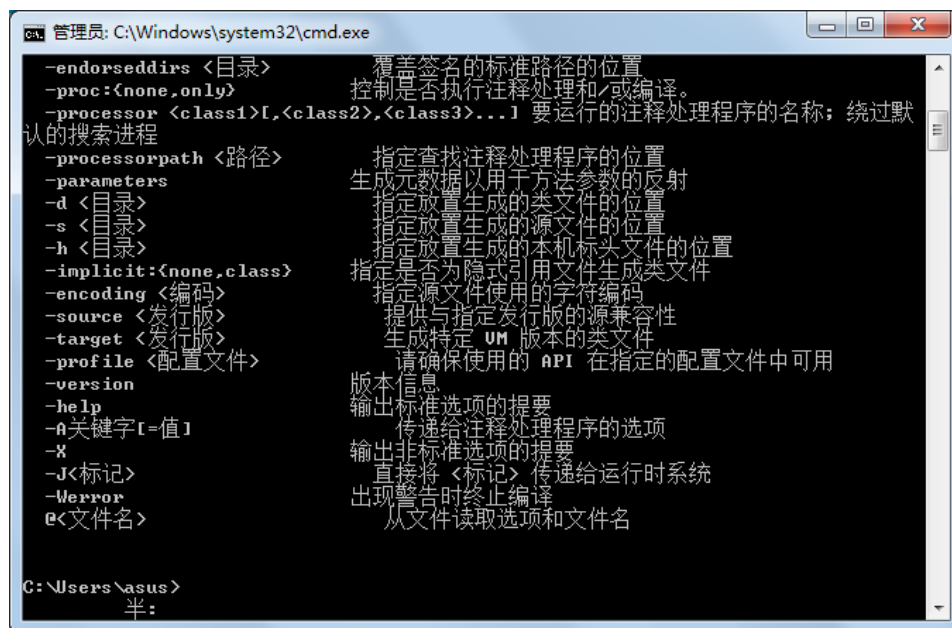
如果申请的资源在等待进程中，申请的进程必须等待

只有在得到之前占有的资源和分配后的资源之后，才可以重新执行进程

循环等待：按照顺序申请资源

2、实验过程

1、打开cmd，看javac是否可以正常运行

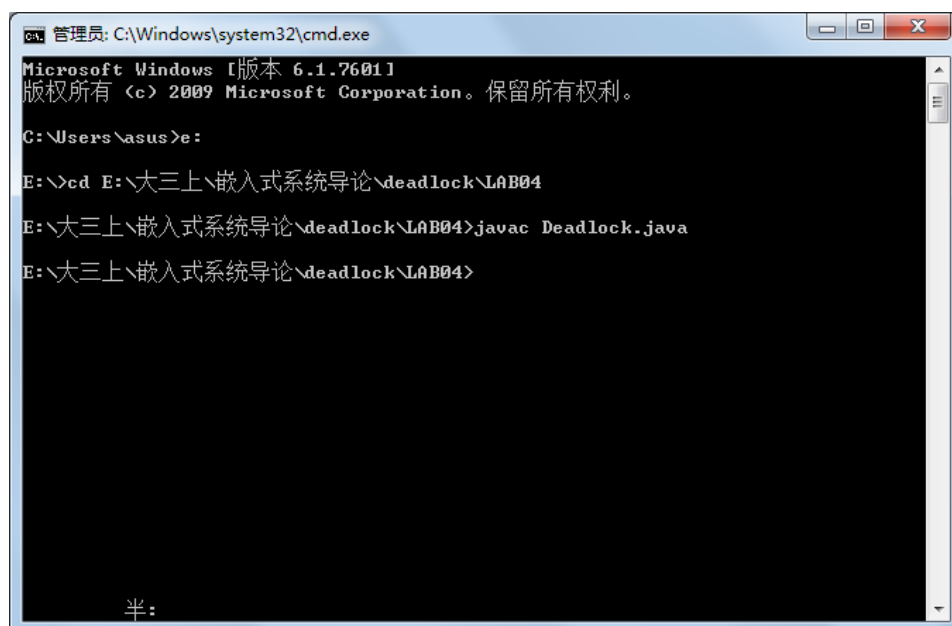


这种代表可以正常的运行

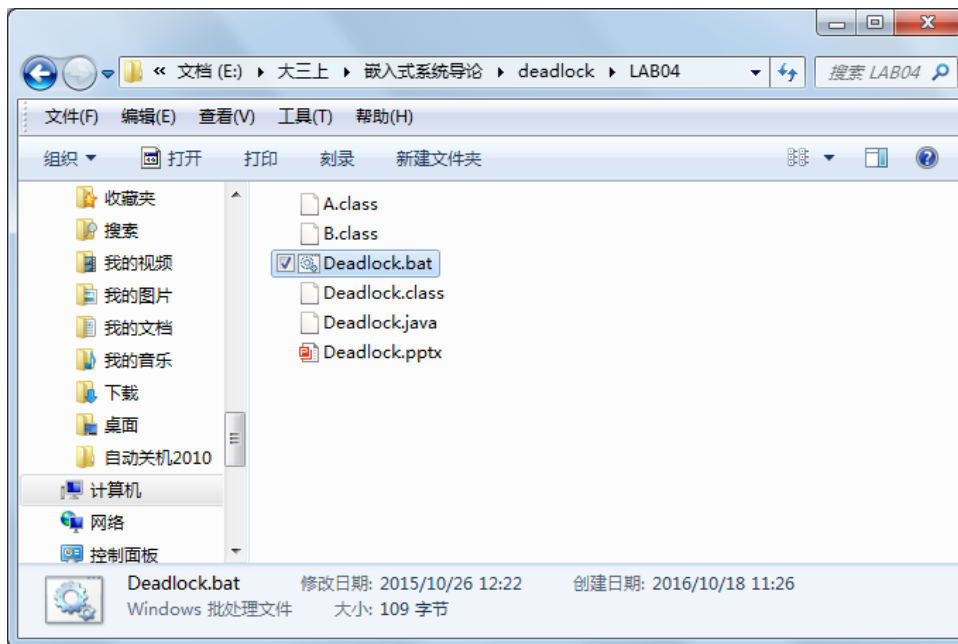
2、编译Deadlock.java

首先进入自己的文件夹中，可以通过下面命令进入

```
e: //进入放置deadlock.java的盘中
cd xxxxx/xxxxx是我们的放置java代码的绝对路径
javac Deadlock.java//编译java代码
```



这样，我们就编译成功了，点击bat文件，就可以运行我们的结果了。



3、实验结果：

死锁的截图

```
C:\Windows\system32\cmd.exe
The time of this thread is 6.0
21
Inside B.last()
Inside A.last()
this is in the run 1.0
this is deadlock 1.0
The time of this thread is 6.0
22
Inside B.last()
Inside A.last()
this is deadlock 1.0
this is in the run 2.0
The time of this thread is 5.0
23
Inside B.last()
Inside A.last()
this is in the run 2.0
this is deadlock 1.0
The time of this thread is 8.0
24
-
```

我们看到的是在第24的时候产生了死锁。由于这次的实验是有很大的随机性的，所以，这里是无法确定到底什么时候会发生死锁的。

死锁产生的原因

要理解死锁的情况，我们首先需要知道的是，这个程序到底是怎么回事。

```
class A{
    synchronized void methodA(B b){
        ...
        b.last();
    }

    synchronized void last(){
        ...
        System.out.println("Inside A.last()");
    }
}
```

这个是我们的class A，其中，第一个同步是我们可以去执行b.last(),而A的last函数是输出一句话“Inside A.last()”。

```
class B{
    synchronized void methodB(A a){
        a.last();
    }

    synchronized void last(){
        System.out.println("Inside B.last()");
    }
}
```

这个则是我们的B class的内容，第一个就是进行a.last的输出，后面则是输出“Inside B.last()”。

所以，我们可以看到的是，Deadlock的函数的实现为：

```
Deadlock(){
    Thread t = new Thread(this);
    int count = 7500;
    double m_msc = System.currentTimeMillis();
    t.start();
    while(count-->0);
    a.methodA(b);
    double mm_msc = System.currentTimeMillis() - m_msc;
    System.out.println("this is deadlock " + mm_msc);
}
//System.out.println("this is between");
public void run(){
    double m_msc = System.currentTimeMillis();
    b.methodB(a);
    double mm_msc = System.currentTimeMillis() - m_msc;
    System.out.println("this is in the run " + mm_msc);
}
```

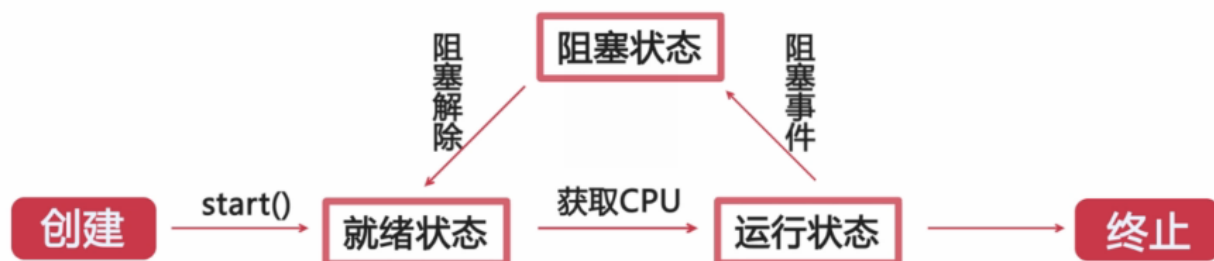
中间的一些输出是为了可以更清晰的看清楚这次的代码而写入其中的。

```
public static void main(String args[]){
    double m_msc = System.currentTimeMillis();
    new Deadlock();
    double mm_msc = System.currentTimeMillis() - m_msc;
    System.out.println("The time of this thread is " + mm_msc);
}
```

主函数如上所示。我们可以进行分析：

- 1、进入Deadlock这个函数中
- 2、在Deadlock中，我们新建了一个线程t，然后开始这个线程。
- 3、经过一段时间的延迟，调用A类的methodA函数
- 4、通过methodA的函数，我们调用B的last，所以，输出“Inside B.last()”
- 5、在上面的过程中，run也在同步的进行的，执行B的methodB，所以，输出“Inside A.last()”
- 6、结束run的线程，我们输出一些数据后，整个函数结束

我们看到的是，上面的过程中，run函数是一直存在的，而A和B由于有synchronized，是存在着竞争的关系的，因此，如果A和B同时竞争者cpu资源的话，那么我们会陷入死锁。



上图是我们的一个线程的从建立到终止的过程。如果这个中间有runnable，那么我们的run线程就是在运行状态的情况下执行run里面的内容。所以，run是和我们的thread是同步进行的，在同一个时间点开始执行，如果两者的时间相撞的话，那么我们会发现资源分配存在竞争，因此满足了死锁的第一点。而由于synchronized的存在且代码中并没有释放资源的说明，我们是不放弃自己的资源的，所以，第二点也是成立的。关于第三点，由于没有释放资源，所以，也是满足。第四点，执行是按照自己书写的命令进行的A中有B，B中有A，所以，是相连的。因此，我们可以断定，死锁是成立的。总结如下：

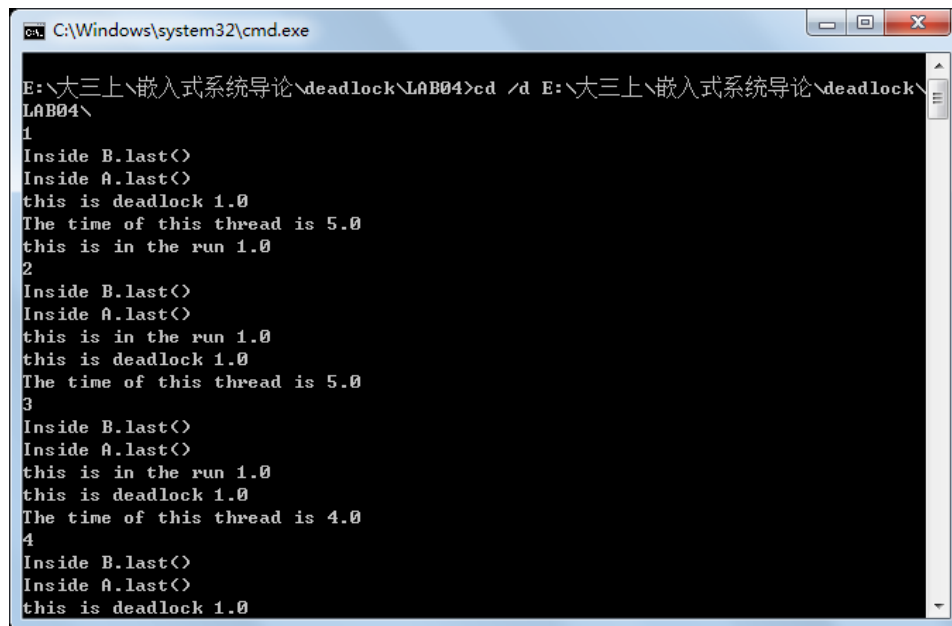
- 1、互斥：run和thread是同步进行的，所以，A.methodA以及B.methodB是会发生cpu资源的抢占的。
- 2、占有并等待：由于有synchronized关键字的存在，所以，是只有在执行完之后，才会释放资源，因此在执行的过程中，占有资源，其他进程不得使用资源
- 3、非抢占：和上面的原因类似，执行完之后自动释放资源，所以，不是抢占的状态
- 4、循环等待：thread中的a.methodA(b)是占用了b的last，而run中的b.methodB(a)是占用了a的last，所以，我们是首尾相连的，a等待b释放资源，然后进入其中。

综上，我们的死锁是成立的。

4、实验讨论

在这次的实验中，我们还发现了一些比较好玩的事情：

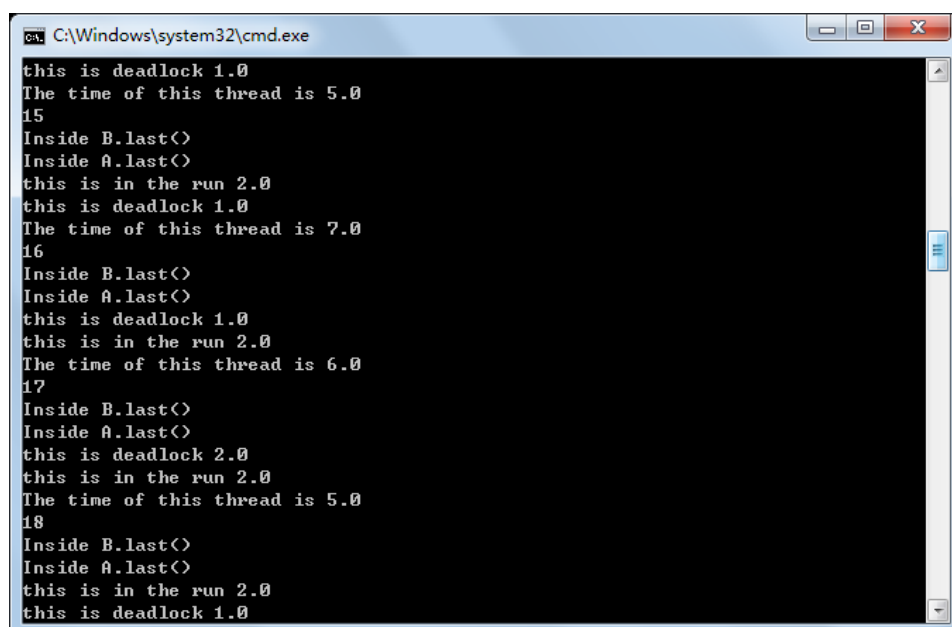
1、run和thread的时间问题



```
C:\Windows\system32\cmd.exe
E:\大三上\嵌入式系统导论\deadlock\LAB04>cd /d E:\大三上\嵌入式系统导论\deadlock\LAB04\
1
Inside B.last()
Inside A.last()
this is deadlock 1.0
The time of this thread is 5.0
this is in the run 1.0
2
Inside B.last()
Inside A.last()
this is in the run 1.0
this is deadlock 1.0
The time of this thread is 5.0
3
Inside B.last()
Inside A.last()
this is in the run 1.0
this is deadlock 1.0
The time of this thread is 4.0
4
Inside B.last()
Inside A.last()
this is deadlock 1.0
```

我们看到，在前面的时候，是thread先输出的，而run是后输出的，所以，其实最后main的结束是在run结束之后才结束的，只有run结束了，我们才会终止main

2、时间突变



```
C:\Windows\system32\cmd.exe
this is deadlock 1.0
The time of this thread is 5.0
15
Inside B.last()
Inside A.last()
this is in the run 2.0
this is deadlock 1.0
The time of this thread is 7.0
16
Inside B.last()
Inside A.last()
this is deadlock 1.0
this is in the run 2.0
The time of this thread is 6.0
17
Inside B.last()
Inside A.last()
this is deadlock 2.0
this is in the run 2.0
The time of this thread is 5.0
18
Inside B.last()
Inside A.last()
this is in the run 2.0
this is deadlock 1.0
```

我们看到的是自己的thread是不确定的，有时候很大，有时候很小，这个和cpu的处理是有关系的了，由于是随机的过程，这里不再赘述，大家明白这种问题，并不是自己跑的有问题，而是这个确实存在就好了。

