

# 华东师范大学数据科学与工程学院实验报告

课程名称:计算机网络与编程	年级:22级	上机实践成绩:
指导教师:张召	姓名:郭夏辉	学号:10211900416
上机实践名称:Java多线程编程2	上机实践日期:2023年3月31日	上机实践编号:No.05
组号:1-416	上机实践时间:2023年3月31日	

## 一、实验目的

- 熟悉Java多线程编程
- 熟悉并掌握线程创建和线程间交互

## 二、实验任务

- 学习使用 `synchronized` 关键字
- 学习使用 `wait()`、`notify()`、`notifyAll()` 方法进行线程交互

## 三、实验环境

- IntelliJ IDEA 2022.3.2
- JDK 19

## 四、实验过程

### task1

对Lab4的3.2中给出的 `PlusMinus`、`TestPlus`、`Plus` 代码，使用 `synchronized` 关键字进行修改，使用两种不同的修改方式，使得 `num` 值不出现线程处理不同步的问题，将实现代码段及运行结果附在实验报告中。

## PlusMinus类

这个类还是十分简单的。`plusOne`方法会将`num`加1，`minusOne`方法会将`num`减1，`printNum`方法会返回`num`的当前值。

```
public class PlusMinus {  
    public int num;  
    public void plusOne(){  
        num = num + 1;  
    }  
    public void minusOne(){  
        num = num - 1;  
    }  
    public int printNum(){  
        return num;  
    }  
}
```

## TestPlus类

这个是程序的主体，透过这个我可以看到这个程序具体要干什么。它创建了一个`PlusMinus`对象，然后创建了10个`Plus`线程，并启动这些线程进行计算。每个`Plus`线程会执行10000次`plusOne`方法，将`num`加上10000。最后，主线程等待所有`Plus`线程执行完毕，并输出`num`的最终值。

```
public class TestPlus {  
    public static void main(String[] args) throws InterruptedException {  
        PlusMinus plusMinus = new PlusMinus();  
        plusMinus.num = 0;  
        int threadNum = 10;  
        Thread[] plusThreads = new Thread[threadNum];  
        for(int i=0;i<threadNum;i++){  
            plusThreads[i] = new Plus(plusMinus);  
        }  
        for(int i=0;i<threadNum;i++){  
            plusThreads[i].start();  
        }  
        for(int i=0;i<threadNum;i++){  
            plusThreads[i].join();  
        }  
        System.out.println(plusMinus.printNum());  
    }  
}
```

## Plus类

这个类也是比较简单的。每次执行run方法时，会调用 `PlusMinus` 的 `plusOne()` 方法，将num加上10000。

```
class Plus extends Thread {
    Plus(PlusMinus pm) {
        this.plusMinus = pm;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            plusMinus.plusOne();
        }
    }

    PlusMinus plusMinus;
}
```

## 结果

第一次运行

21120

第二次运行

22865

第三次运行

20469

可以看到不仅每次都没有达到预期的结果(100000),而且每次竟然运行的结果还不一样。

## 分析与修改

`PlusMinus` 对象中的num变量是被多个线程共享的，会出现线程竞争问题。根据操作系统的所学知识，线程竞争有三种具体的形式：原子性问题(原子性:一个操作不可分割、不可中断)、可见性问题(可见性:一个线程对共享变量的修改能够被其他线程及时地观察)和有序性问题(有序性:程序执行的顺序和预期的顺序一致)。在这个问题中，碰到的是原子性问题。

无论是什么问题，解决线程竞争问题的一般途径还是采用同步机制来保证线程之间的同步和互斥。在Java中，**synchronized**是一种同步机制，专门用来解决多线程并发执行时的数据竞争问题。方法被**synchronized**修饰后，同一时刻只有一个线程能执行该方法，其他线程会被阻塞，直到当前线程执行完毕，释放锁之后才能继续访问。所以我主要是修改**PlusMinus**的**plusOne()**方法，因为这个问题中只用了**plusOne()**。

```
public synchronized void plusOne(){
    num = num + 1;
}
```

修改后的结果如预期一样。

100000

还有最后一个问题，这个能不能去只用**volatile**修饰**PlusMinus**中**num**变量的类型呢？经过分析和测试，是不行的。首先，自加操作并不是一个原子操作，仅仅用**volatile**修饰并未解决原子性问题。其次，**Lab4-task5**中之所以添加**volatile**修饰就能正常，是因为**Testvolatile**碰到的是可见性问题，而不是原子性问题。

## task2

给出以下**TestMax**、**MyThread**、**Res**代码，使用**synchronized**关键字在**TODO**处进行修改，实现最后打印出的**res.max\_idx**的值是所有**MyThread**对象的**list**中保存的数的最大值，将实现代码段及运行结果附在实验报告中。

### 设计思路

这个其实还是有一点迷惑性，因为**idx**我一般认为是索引的，但是这个题目中要求的却是**list**中保存的最大的数。以下是我修改的代码。

```
@Override
public void run() {
    int maxx=Integer.MIN_VALUE;
    synchronized(res) {
        for (int i=0;i<list.size();i++) {
            if (list.get(i)>maxx) {
                maxx=list.get(i);
            }
        }
        if (maxx>res.max_idx) {
            res.max_idx=maxx;
        }
    }
}
```

这里有几个小问题，首先就是对于ArrayList<>对象应该用get(i)方法来获取元素。其次就是为什么我不能在if内部synchronized而要把整个运行区给synchronized了且额外多加一个中间变量maxx?因为如果我不这么做，在一个if执行完成准备修改max\_idx时，切换到另外的线程会引发覆盖问题。

运行结果

9951

## task3

设计3个线程彼此死锁的场景并编写代码(可基于上述代码或自己编写)，将实现代码段及运行结果附在实验报告中。

设计思路

首先根据实验指导，我先要理解为什么TestDeadLock等类在运行时会发生死锁现象。在MyThread类中的run()方法，每个线程都试图获得两个PlusMinus对象的锁。线程1先获得pm1的锁，然后尝试获取pm2的锁；而线程2先获得pm2的锁，然后尝试获取pm1的锁。会存在这样一个情况，就是线程1已经获得了pm1的锁，但是此时线程2正在占用pm2的锁，然后线程1就会等待线程2释放pm2的锁，而线程2此时并没有要释放pm2的锁的计划，反而是在同时等待线程1释放pm1的锁。两个线程互相等待对方释放资源而无法继续执行下去了，程序会永久地停滞，死锁现象便发生了。

要设计三个线程彼此死锁的情况，我的想法是让三个线程的依赖关系构成一个环，即thread1最初占用pm1，之后尝试占用pm2;thread2最初占用pm2，之后尝试占用pm3;thread3最初占用pm3，之后尝试占用pm1。

利用题中给到的代码，我构建了自己的类完成此问题。

## PlusMinus类

引用题中，不再赘述。

```
class PlusMinus {
    public int num;
    public void plusOne() {
        num = num + 1;
    }
    public void minusOne() {
        num = num - 1;
    }
    public int printNum() {
        return num;
    }
}
```

## MyThread类

这里参考题目中所给代码略微修改了一下，代码如下所示。

```
class MyThread implements Runnable {
    @Override
    public void run() {
        if (tid == 1) {
            synchronized (pm1) {
                System.out.println("thread" + tid + "正在占用 plusMinus1");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("thread" + tid + "试图继续占用 plusMinus2");
                System.out.println("thread" + tid + "等待中...");
                synchronized (pm2) {
                    System.out.println("thread" + tid + "成功占用了 plusMinus2");
                }
            }
        } else if (tid == 2) {
            synchronized (pm2) {
                System.out.println("thread" + tid + "正在占用 plusMinus2");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("thread" + tid + "试图继续占用 plusMinus3");
                System.out.println("thread" + tid + "等待中...");
                synchronized (pm3) {
                    System.out.println("thread" + tid + "成功占用了 plusMinus3");
                }
            }
        } else if (tid == 3) {
            synchronized (pm3) {
                System.out.println("thread" + tid + "正在占用 plusMinus3");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("thread" + tid + "试图继续占用 plusMinus1");
                System.out.println("thread" + tid + "等待中...");
                synchronized (pm1) {
```

```

        System.out.println("thread" + tid + "成功占用了 plusMinus1");
    }
}
}
}
MyThread(PlusMinus _pm1,PlusMinus _pm2,PlusMinus _pm3,int _tid) {
    this.pm1 = _pm1;
    this.pm2 = _pm2;
    this.pm3 = _pm3;
    this.tid = _tid;
}
PlusMinus pm1;
PlusMinus pm2;
PlusMinus pm3;
int tid;
}

```

## Deadlock3类

结合所给代码，根据我的设计思路，我的代码是这样的。

```

public class Deadlock3 {
    public static void main(String[] args) throws InterruptedException {
        PlusMinus plusMinus1 = new PlusMinus();
        plusMinus1.num = 1000;
        PlusMinus plusMinus2 = new PlusMinus();
        plusMinus2.num = 1000;
        PlusMinus plusMinus3 = new PlusMinus();
        plusMinus3.num = 1000;
        MyThread thread1 = new MyThread(plusMinus1, plusMinus2,plusMinus3, 1);
        MyThread thread2 = new MyThread(plusMinus1, plusMinus2,plusMinus3, 2);
        MyThread thread3 = new MyThread(plusMinus1, plusMinus2,plusMinus3, 3);
        Thread t1 = new Thread(thread1);
        Thread t2 = new Thread(thread2);
        Thread t3 = new Thread(thread3);
        t1.start();
        t2.start();
        t3.start();
        t1.join();
        t2.join();
        t3.join();
    }
}

```

## 运行结果

```
thread2正在占用 plusMinus2
thread3正在占用 plusMinus3
thread1正在占用 plusMinus1
thread2试图继续占用 plusMinus3
thread1试图继续占用 plusMinus2
thread3试图继续占用 plusMinus1
thread2等待中...
thread1等待中...
thread3等待中...
```

通过输出结果，可以看到发生了死锁。并且可以观察到三个线程的运行顺序是无法预先假定的。

## task4

首先阐述 `synchronized` 在实例方法上的作用，然后运行本代码段，同时打开检测cpu的工具，观察cpu的使用情况，将实验结果和cpu使用情况截图附在实验报告中。

### 设计思路&注意事项

在这个问题中，`PlusMinusOne` 的 `plusOne()` 和 `minusOne()` 都使用 `synchronized` 修饰地来对 `num` 访问和修改。然后在 `TestInteract` 中创建了两个线程，一个在 `plusOne()` 而另外一个在 `minusOne()`。在任何一个时刻，只有一个线程可以执行 `plusOne()` 或 `minusOne()`，在执行时会获取 `this` 对象的锁。这便是此处 `synchronized` 的作用。

正常的运行结果(有省略)

```
.....
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
.....
```

删去 `synchronized` 后的运行结果(有省略)



```
.....
num = 1
num = 1
num = 1
num = 1
num = 1
num = 1
num = 1
num = 1
num = 1
.....
```

可以很明显地看到，删去之后出现了两个线程同时对num的操作，一个线程在加，另一个线程在减，num值竟然不变。那这个程序到底在干什么？根据代码，plusOne()的休眠时间是minusOne()的十倍，num从50开始一直减，减了十个单位时间，然后开始加，加一个单位时间后就继续减（比如说50一路减到了30，然后一路加到32后继续减）；依此类推，直到num减少到1。

运行结果

最后附一下程序运行时的截图

任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	状态	7% CPU	44% 内存	0% 磁盘	0% 网络	5% GPU	GPU 引擎	电源使用情况	电源使用情况...
应用 (6)									
> Google Chrome (5)		0.1%	186.6 MB	0 MB/秒	0 Mbps	0%	GPU 0 - 3D	非常低	非常低
IntelliJ IDEA (5)		6.3%	2,346.7 ...	0.1 MB/秒	0 Mbps	0%		非常高	非常低
OpenJDK Platform binary		4.5%	16.2 MB	0 MB/秒	0 Mbps	0%		高	非常低
OpenJDK Platform binary		0%	86.3 MB	0.1 MB/秒	0 Mbps	0%		非常低	非常低
project5 - TestInteract.java		1.8%	2,232.7 ...	0 MB/秒	0 Mbps	0%		低	非常低
控制台窗口主机		0%	5.7 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
控制台窗口主机		0%	5.7 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
> Microsoft Edge (6)		0.1%	297.3 MB	0.1 MB/秒	0 Mbps	0%	GPU 0 - 3D	非常低	非常低
> Typora (5)		0%	199.2 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
> Windows 资源管理器		0%	49.7 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
> 任务管理器		0.4%	23.5 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
后台进程 (80)									
AcroTray		0%	1.1 MB	0 MB/秒	0 Mbps	0%		非常低	非常低

简略信息(D)

结束

task5

在Task4基础上增加若干减一操作线程，运行久一点，观察有没有发生错误。若有，请分析错误原因，给出解决代码。

## 设计过程&结果

```
Thread t3 = new Thread() {
    public void run() {
        while (true) {
            while (pmo.num == 1) {
                continue;
            }
            pmo.minusOne();
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
};
t3.start();
```

会出现错误。我只在 `TestInteract` 中只加了一个执行减一运算的线程，然后就出问题了。

```
.....
num = 1
num = 2
num = 1
num = 0
num = -1
num = -2
num = -3
num = -4
.....
```

可以看到，`num`竟然为负了（`num=1`时减法进程理论上不应该运行，这样`num`也不应该从1到0再到负数）。这是为什么呢？首先，`num`在`while`判断时没有被`synchronized`修饰，可能出现多个线程同时访问并修改`num`的情况。会出现这样的一种情况，就是某减一线程A运行后、结束前另外一个减一线程B也在运行并访问`num`（假设此时`num=2`）然后线程A把`num`从2减到了1，但是线程B中的`while`语句已经判断完毕了，线程B也把`num`减了一下（`num`从1减到了0），错误发生。

我最开始的解决思路是这样的，就是在每个减法线程中使用`synchronized`来修饰`pmo` (诸如以下的代码),但是这样之后程序就会出现死锁问题，无法像最初那样长久地运行了。

```
Thread t1 = new Thread() {
    public void run() {
        while (true) {
```

```

        synchronized (pmo){
            while (pmo.num == 1) {
                continue;
            }
            pmo.minusOne();
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
};
t1.start();

```

```

Thread t3 = new Thread() {
    public void run() {
        while (true) {
            synchronized (pmo){
                while (pmo.num == 1) {
                    continue;
                }
                pmo.minusOne();
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
};
t3.start();

```

```
.....  
num = 8  
num = 7  
num = 6  
num = 5  
num = 4  
num = 3  
num = 2  
num = 1
```

上面这个思路应该差不多，但是面对死锁我陷入了沉思.....到底该怎么办呢？结合前面的所学知识，死锁往往发生于两个/多个线程竞争相同的资源时，线程t1和线程t3在竞争 `pmo` 的锁时产生了竞争，那我可不可以设置一个独立的锁呢？这样就可以避免死锁现象了。抱着试一试的心态，我把代码改成了如下的模式。

```
Object lock = new Object();  
  
Thread t1 = new Thread() {  
    public void run() {  
        while (true) {  
            synchronized (lock) {  
                while (pmo.num == 1) {  
                    continue;  
                }  
                pmo.minusOne();  
                try {  
                    Thread.sleep(10);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
};  
t1.start();  
Thread t3 = new Thread() {  
    public void run() {  
        while (true) {  
            synchronized (lock) {  
                while (pmo.num == 1) {  
                    continue;  
                }  
                pmo.minusOne();  
                try {  
                    Thread.sleep(10);  
                }  
            }  
        }  
    }  
};  
t3.start();
```

```

        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

}

};
t3.start();
Thread t2 = new Thread() {
    public void run() {
        while (true) {
            pmo.plusOne();
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
};
t2.start();

```

运行结果很惊喜，看来这样做确实死锁被消除了！

```

.....
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
.....

```

## task6

在以下代码中加入若干获取product的线程，并运行截图；之后将 `while (productQueue.isEmpty())` 修改为 `if (productQueue.isEmpty())`，并观察运行结果，如发生错误，试分析原因。

我又增加了两个获取product的线程。（t1,t3,t4都在获取product,t2在添加product）

```

Thread t3 = new Thread() {

```

```

        public void run() {
            while (true) {
                try {
                    String s = pf.getProduct();
                    System.out.println("t3 get product: " + s);
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
Thread t4 = new Thread() {
    public void run() {
        while (true) {
            try {
                String s = pf.getProduct();
                System.out.println("t4 get product: " + s);
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    };
};
t3.start();
t4.start();

```

## 运行结果&分析

运行结果如下图所示。

```
t2 add product: product
t4 get product: product
t2 add product: product
t1 get product: product
t2 add product: product
t4 get product: product
t2 add product: product
t1 get product: product
t2 add product: product
t4 get product: product
t2 add product: product
t1 get product: product
t2 add product: product
t4 get product: product
t2 add product: product
t1 get product: product
t2 add product: product
t4 get product: product
```

在将while改为if后，输出结果如下所示。

```
Exception in thread "Thread-0" java.util.NoSuchElementException Create breakpoint
    at java.base/java.util.LinkedList.removeFirst(LinkedList.java:274)
    at java.base/java.util.LinkedList.remove(LinkedList.java:689)
    at ProductFactory.getProduct(ProductFactory.java:18)
    at Test$1.run(Test.java:10)
t2 add product: product
t3 get product: product
t2 add product: product
t4 get product: product
Exception in thread "Thread-2" java.util.NoSuchElementException Create breakpoint
    at java.base/java.util.LinkedList.removeFirst(LinkedList.java:274)
    at java.base/java.util.LinkedList.remove(LinkedList.java:689)
    at ProductFactory.getProduct(ProductFactory.java:18)
    at Test$3.run(Test.java:37)
t4 get product: product
t2 add product: product
```

鲜红的报错告诉我们，这里出错了。如果从表面上来说，出错的原因在于队列还没有`product`呢`t1/t3/t4`这样的获得`product`线程就去请求`product`了，造成了越界而出错。如果从线程的运行原理来说，这里出问题是因为使用`if`条件时，线程只会检查一次条件。现在也模拟一个例子，如果队列不空（有`product`但只有一个），线程`t1`就会继续执行，但此时另外的线程`t3`可能已经获取了`product`了（这是队列就空了，本来这时`t1`就该`wait()`了，但现在`t1`的控制流已经到了`if`之后），然后`t1`再次执行队列清空操作（`productQueue.remove()`），将一个本已经为空的队列再次弹出，这显然是不合逻辑的，错误便发生了。根本原因还在于如果是`while`就会反复地判断但是`if`只判断了一次就走了。

## Bonus Task1

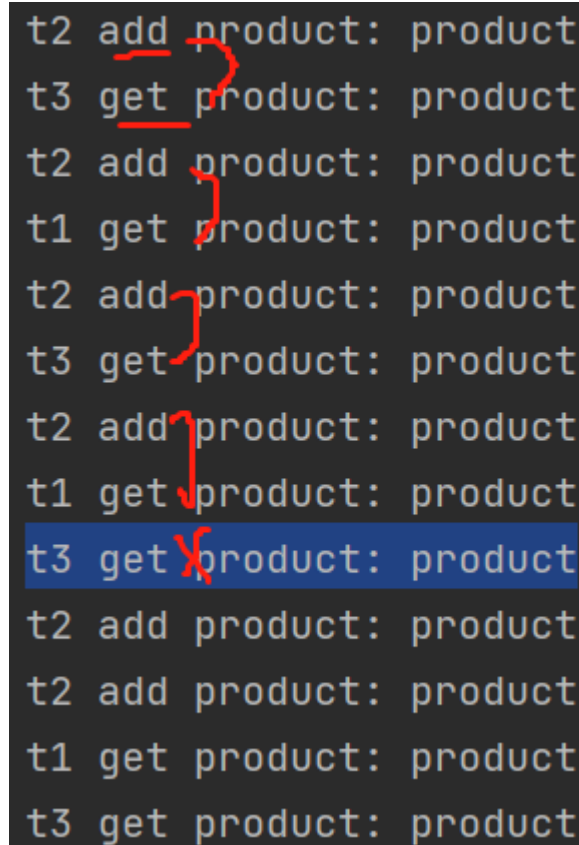
可以修改以下代码逻辑，试说明如果不使用 `notifyAll()` 而是使用 `notify()`，在哪些情况下可能出错？

这个题目让自己花了一些时间去理解，题目中其实有一些消歧义。因为如果不在 `Test` 类中添加获取`product`的线程，那么 `notifyAll()` 和 `notify()` 应该是一样的，因为只有一个线程需要等待（一般是获取`product`的线程需要等待）。

`notifyAll()` 能够唤醒所有的等待线程；但是 `notify()` 只能唤醒一个线程，并且还是随机唤醒的。如果运气不太妙的话，有可能出现有的线程一直都没被唤醒，一直都在等待队列中。如果多个线程都在等待同一个锁对象，采用 `notify()` 可能会唤醒与该锁对象关联的某一个线程，但是其他线程仍然在等待，造成无法预知的错误。

我在 `Test` 类添加了一个获取`product`的线程`t3`，然后在 `ProductFactory` 中修改了 `addProduct()` 方法的操作，将 `notifyAll()` 改成了 `notify()`，运行结果如下所示：





```
t2 add product: product
t3 get product: product
t2 add product: product
t1 get product: product
t2 add product: product
t3 get product: product
t2 add product: product
t1 get product: product
t3 get product: product
t2 add product: product
t2 add product: product
t1 get product: product
t3 get product: product
```

改成`notify()`之后就出现了逻辑错误。

## task7

在Task5的基础上，使用`wait`和`notify`修改代码，达到一致的代码逻辑，同时打开检测cpu的工具，观察cpu的使用情况，将实验结果和cpu使用情况截图附在实验报告中。

根据`wait`和`notify`方法的逻辑，我这样来再次实现Task5的效果。此时要注意`wait`和`notify`的先后顺序。

```
public class TestInteract {
    public static void main(String[] args) {
        PlusMinusOne pmo = new PlusMinusOne();
        pmo.num = 50;
        Object lock = new Object();

        Thread t1 = new Thread() {
            public void run() {
                while (true) {
                    synchronized (lock) {
                        while (pmo.num == 1) {
                            try {
                                lock.wait();
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }
                    }
                }
            }
        }
```

```

        continue;
    }
    pmo.minusOne();
    try {
        Thread.sleep(10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

}

};
t1.start();
Thread t3 = new Thread() {
    public void run() {
        while (true) {
            synchronized (lock) {
                while (pmo.num == 1) {
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    continue;
                }
                pmo.minusOne();
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
};
t3.start();
Thread t2 = new Thread() {
    public void run() {
        while (true) {
            synchronized (lock){
                pmo.plusOne();
                lock.notifyAll();
            }
        }
    }
};

```

```
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};
t2.start();
}
}
```

运行结果

```
.....
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
num = 1
num = 2
.....
```

最后附一下程序的CPU占用情况，可以看到在修改后确实比task4中要有明显地效率提升。

任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	状态	11% CPU	68% 内存	3% 磁盘	0% 网络	7% GPU	GPU 引擎	电源使用情况	电源使用情况...
应用 (7)									
> Google Chrome (5)		0%	198.1 MB	0.1 MB/秒	0 Mbps	0%		非常低	非常低
IntelliJ IDEA (3)		0.7%	2,481.5 ...	0 MB/秒	0 Mbps	0%		低	非常低
OpenJDK Platform binary		0%	86.0 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
OpenJDK Platform binary		0%	16.5 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
project5 - TestInteract.java		0.7%	2,379.0 ...	0 MB/秒	0 Mbps	0%		低	非常低
> Microsoft Edge (6)		0%	127.1 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
> Typora (5)		0%	185.2 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
> VirtualBox Virtual Machine		9.4%	98.5 MB	9.9 MB/秒	0.1 Mbps	0%		非常高	非常低
> Windows 资源管理器 (2)		0.1%	36.3 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
> 任务管理器		0.2%	24.3 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
后台进程 (83)									
AcroTray		0%	0.1 MB	0 MB/秒	0 Mbps	0%		非常低	非常低
> Activation Licensing Service (32 位)		0%	0.5 MB	0 MB/秒	0 Mbps	0%		非常低	非常低

简略信息(D)

结束任务(t)

## 五、总结

多线程的原理是复杂的，这次实验整体上来说还是有些难度的。对我个人而言，逐步摸索的过程中我对JAVA多线程有了更深刻的理解。在编程时，还是要采用更加稳妥的策略。比如尽量使用 `notifyAll()` 而不是 `notify()` 以防止假死现象；小心使用 `synchronized` 修饰资源防止死锁现象的发生。规范地使用相关的方法，才能更好地发挥多线程的力量，希望我在之后的网络程序编程时能娴熟地运用好多线程。