

课程编号: A0801040060

操作系统-大作业 一



姓名	柳成林	班级 学号	软件 2001 20206722	成绩	
报告名称	大作业一				
开设学期	2021-2022 春季学期				
开设时间	第 16 周 — 第 17 周				
使用主要技术	<p>主要用到的技术是:</p> <p>注: 以上内容, 请同学们按照自己作业的实际情况填写</p>				
是否使用框架、中间件、数据库及编写好的程序	<p><input type="checkbox"/> 是 <input checked="" type="checkbox"/> 否</p> <p>注: 以上内容, 请同学们按照自己作业的实际情况填写</p>				
可选难度	<p><input type="checkbox"/> 基础 <input checked="" type="checkbox"/> 进阶 <input type="checkbox"/> 高阶</p> <p>注: 以上内容, 请同学们按照自己作业的实际情况填写</p>				
加分项	<p><input checked="" type="checkbox"/> 测试设计和测试过程: 测试设计巧妙</p> <p><input checked="" type="checkbox"/> 为测试开发了测试工具或自动化测试工具</p> <p><input checked="" type="checkbox"/> 图形用户界面</p> <p>注: 以上内容, 请同学们按照自己作业的实际情况填写</p>				

东北大学软件学院

一、实验目的

实验内容：设计并实现一个银行记账系统，该系统需要支持以下功能：

1. 通过一个用户的 ID，为其存款或取款。该操作是实时操作，优先级高，必须要立刻执行。
2. 通过付款账户 ID 和收款账户 ID，进行转账。该操作是实时操作，优先级高，必须要立刻执行。
3. 批量为多个账户发工资。该操作属于批处理操作，优先级低。由于用户数量多，该操作会持续一段时间。
4. 批量为多个账户计算利息，并在账户余额中存入利息。该操作属于批处理操作，优先级低。由于用户数量多，该操作会持续一段时间。

设计知识点：

进程的同步与死锁；实时操作与批处理操作；原子量；

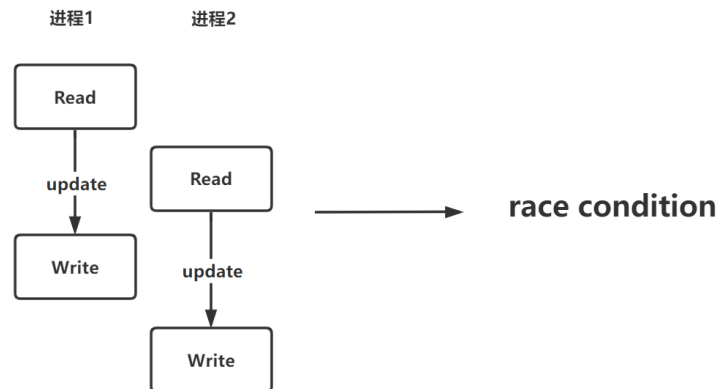
二、问题分析与程序设计

要求 1：

所有操作都是在并发状态下执行。即，同一账号可能同时被多个进程（或线程）操作，要注意避免 **race condition** 及死锁的出现。要保证数据在并发操作时的正确性。

分析：

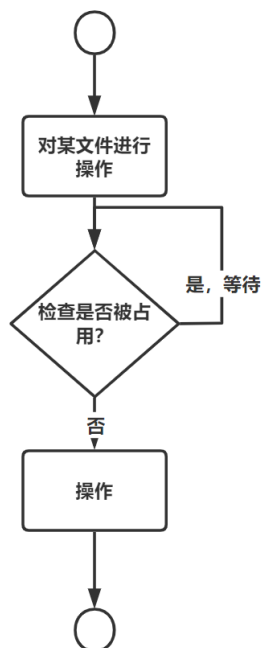
1. 可能出现死锁的场景：文件在多线程或多进程并发进行读写操作，可能出现资源竞争的情况出现，当有一条或多条线程一直占用其中的文件资源且想修改另一个文件资源的时候，就会出现死锁。
2. 可能会出现 **race condition** 的场景：文件操作分为读取和写入操作，假如读和写是分开进行的，当前连续进行了两次存款操作，那么这两次读取到的初始金额可能就是一样的（第二次读取在第一次写入操作之前），虽然这样不会出现死锁，但这样会造成实际结果与预期结果不符。



解决方案:

1. 针对可能出现死锁的情况:

当对某个文件进行读写操作时,先检查有没有其他线程或者进程正在对这个文件进行操作,如果没有,则进行操作;如果有,则暂停操作,等待正在占用该文件的进程结束后再进行操作。



具体的实施方案:

通过 Java 的 FileChannel 类操作文件:

当有多个进程或者多个应用同时操作文件时,会并行往文件中写入字节,如何保证多个进程中文件写入或者操作当原子性就很重要。此时,在 Java 层可以使用 FileChannel.lock 来

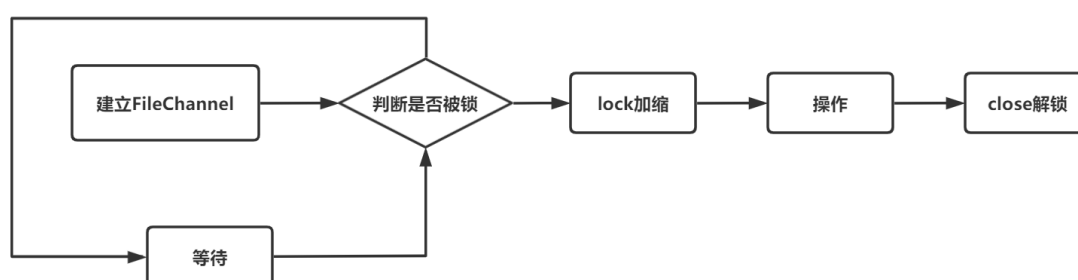
完成多进程之间对文件操作的原子性，而该 lock 会调用 Linux 的 `fcntl` 来从内核对文件进行加锁。

文件锁属于信号量的一种，可以控制多线程和不同进程之间的资源所有权。

当建立某一个文件的 `FileChannel` 时，获取该文件的加锁状态，如果没有被加锁，先对该 `FileChannel` 进行 lock 加锁，再对文件进行操作，当操作结束后，再将锁释放；如果该文件锁获取不到（即改文件正在被其他进程或者线程操作），那么就让系统等待一定时间，直到锁释放后，再进行操作。

具体的实现流程：

方法体使用 `synchronized`，方法内部逻辑：



2. 针对可能出现 race condition 的情况：

这里我搜索了很多种方法，比如说进程同步，或者避免使用共享变量，但都有一些弊端，而且实现起来也很麻烦。所以我使用了简单粗暴的方法，就是将‘读-修改-写’三个方法作为一个基本的文件操作 `update`，而不是将他们分开。

将这个方法整体加锁，这样当其他进程读取这个文件时，就会检测到该文件被占用。如果是分开的情况，虽然读、写时都可以检测到占用，但是如果在读写之间进行操作，则不会冲突。

程就像自旋锁一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择一个另一个线程进入。

具体实现方案如上图所示，由于我在系统中定义的批处理方法为单次批处理，所以要外部轮询判断实时操作是否结束。

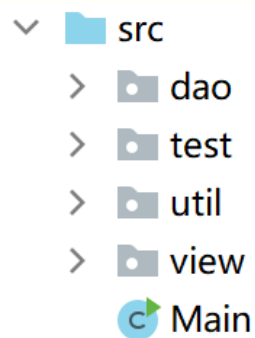
要求 3:

实现一个原型系统。编程语言不限。但不允许使用任何框架（framework）、中间件、数据库，及编写好的程序（如各种 web server）。

本系统均未使用。

三、实现过程与测试结果分析

项目结构:



项目采用 Swing 构建 GUI 界面，采用 Junit 进行测试。

程序设计:

界面设计




左边是侧边按钮，点击会弹出对应的界面，右边显示当前完成的所有操作。



页面输入要求：输入的 id 会自动判断是否有效，同时会判断输入是否为空

页面显示：会当前页面正在进行的操作，比如说正在转账，会显示“转账中..”，转账结束，会显示“转账结束，余额：XX”，同时所有活动会显示在主界面上，方便确定：



```
转账活动 Thread-0 已完成 Sat Jun 18 19:43:37 CST 2022
存钱活动 Thread-1 已完成 Sat Jun 18 19:43:37 CST 2022
取钱活动 Thread-2 已完成 Sat Jun 18 19:43:37 CST 2022
```

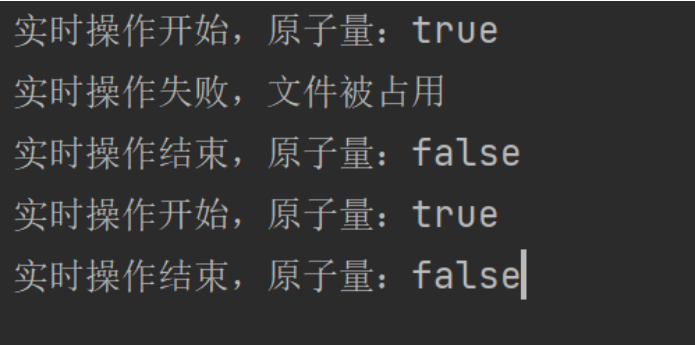
由于每一次读写文件都非常快，而为了体现进程是否被占用，我将实时操作时间定为 3 秒，批处理操作为 3n 秒，通过以下方法进行倒计时：

```
if (lock != null) { //lock不为空，说明当前文件没有被占，可以正常写入
    //延迟时间 3秒
    long start = System.currentTimeMillis();
    while (System.currentTimeMillis()-start<=(3000)){}
```

倒计时结束后，才进行文件读写操作，这样可以保证当前文件的锁被占用 3 秒以上，当其他线程访问此文件时，有足够的时间检测到该文件被占用，也有足够的时间将原子量置为 true，以检测当前是否有实时操作。

系统实现展示：

现在对用户 1 连续进行两次存钱操作，间隔时间小于 3 秒：



```
实时操作开始，原子量: true
实时操作失败，文件被占用
实时操作结束，原子量: false
实时操作开始，原子量: true
实时操作结束，原子量: false|
```

第一次存钱操作正常执行，第二次失败，因为第一次操作的 3 秒内，该文件还被上锁，进行等待，第 3 行表明第一次存钱已经结束，该文件已经不再被占用，第 4 行和第 5 行表明第二次操作正常进行了。

现在先对用户 1 进行存钱操作，再对用户 1 和用户 2 进行发工资操作，将每一步进行输

出：

```
实时操作开始，原子量: true
批处理操作开始，原子量: true
    发生实时操作，暂停批处理
实时操作结束，原子量: false
批处理操作开始，原子量: false
批处理操作结束
批处理操作开始，原子量: false
批处理操作结束
|
```

对 1 进行实时操作，此时原子量设为 `true`，紧接着进行发工资操作，由于实时操作需要进行 3 秒，此时批处理操作检测到原子量还不是 `false`，说明当前有实时操作，所以暂停批处理，等待实时操作完成。当实时操作完成时。原子量为 `false`，此时批处理可以正常执行，由于要对 1 和 2 进行发工资，所以批处理执行了两次。

现在先对用户 1 和用户 2 进行发工资操作，再对用户 3 进行存钱操作，将每一步进行输出：

```
批处理操作开始，原子量: false
实时操作开始，原子量: true
批处理操作结束
批处理操作开始，原子量: true
    发生实时操作，暂停批处理
实时操作结束，原子量: false
批处理操作开始，原子量: false
批处理操作结束
```

先开始批处理操作，此时检测到原子量为 `false`，所以可以进行；再开始实时操作，在第一个批处理操作结束后，第二个批处理操作检测到原子量为 `true`，所以暂停等待实时操作结束，重新开始。

代码实现：

```

/**
 * 存钱
 * @param countName
 * @param m 钱
 * @return
 * @throws IOException
 */
public String addMoney(String countName, long m) throws IOException, InterruptedException {
    int i = 0;
    while (i == 0) {
        boolean flag = FileUtil.updateRealTime(countName, m);
        if(flag){ //当前文件空闲
            i++;
        }
        else{ //当前文件被占
            Thread.sleep( millis: 3000); //如果文件被占用，等待3秒
        }
    }
    long money = FileUtil.read(countName);
    return String.valueOf(money);
}

```

存钱方法：通过判断文件是否被占用，如果被占用，则等待 3 秒，继续执行。

其他方法类似，调用的是 FileUtil 中的文件读写基本方法。

文件操作：update 方法：整合了读-修改-写，分为两种，一种是优先级高的实时方法，一种是优先级低的批处理方法：

```

/**
 * 文件更新实时
 * @param filename
 * @param updateMoney
 * @throws IOException
 */
public static boolean updateRealTime(String filename, long updateMoney) throws
IOException {
    long money = 0;
    FileLock lock = null; //文件锁 初始状态 lock 为空
    FileChannel fileChannel = FileChannel.open(Paths.get(".\\data\\" + filename +
".txt"), StandardOpenOption.READ,
        StandardOpenOption.WRITE, StandardOpenOption.CREATE);
    if (lock == null) {
        try {
            lock = fileChannel.tryLock();
        }
        catch (OverlappingFileLockException e){ //如果当前文件被占，报异常
            fileChannel.close();
        }
    }
}

```

```
        System.out.println("实时操作失败，文件被占用");
        return false;
    }
}
if (lock != null) { //lock 不为空，说明当前文件没有被占，可以正常写入
    realTime.set(true);
    System.out.println("实时操作开始，原子量: "+realTime.get());
    //延迟时间 3 秒
    long start = System.currentTimeMillis();
    while (System.currentTimeMillis()-start<=(3000)){

        ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
        fileChannel.read(byteBuffer);
        byteBuffer.flip();
        String moneyStr = StandardCharsets.UTF_8.decode(byteBuffer).toString();
        if (moneyStr.length() > 0) {
            //读出 money
            money = Long.parseLong(moneyStr);
        }
        money = money + updateMoney;
        //写入
        moneyStr = String.valueOf(money);
        byteBuffer = ByteBuffer.allocate(1024);
        byteBuffer.put(moneyStr.getBytes(StandardCharsets.UTF_8));
        byteBuffer.flip();
        fileChannel.truncate(0);
        fileChannel.write(byteBuffer);

        byteBuffer.clear();
        lock.close();
        realTime.set(false);
        System.out.println("实时操作结束，原子量: "+realTime.get());
        fileChannel.close();
        return true;
    }
}
else{
    fileChannel.close();
    System.out.println("实时操作失败，文件被占用");
    return false;
}
}
```

```

/**
 * 批处理文件更新
 * @param filename
 * @param updateMoney
 * @return
 * @throws IOException
 */
public static boolean upDate(String filename, long updateMoney) throws
IOException {
    long money = 0;
    if (updateMoney == 0) {
        return false;
    }
    boolean flag = realTime.get();//如果实时操作进行中，暂停批量操作
    System.out.println("批处理操作开始，原子量: "+flag);
    if (flag) {
        System.out.println("  发生实时操作，暂停批处理");
        return true;
    }
    boolean isWait = false;
    FileLock lock = null;
    FileChannel fileChannel = FileChannel.open(Paths.get(".\\data\\" + filename +
".txt"), StandardOpenOption.READ,
        StandardOpenOption.WRITE, StandardOpenOption.CREATE);
    if (lock == null) {
        try {
            lock = fileChannel.tryLock();
        }
        catch (OverlappingFileLockException e){ //如果当前文件被占，报异常
            fileChannel.close();
            System.out.println("批处理操作失败，文件被占用");
            return false;
        }
    }
    if (lock != null) { //lock 不为空，说明当前文件没有被占，可以正常写入
        //延迟时间 3 秒
        long start = System.currentTimeMillis();
        while (System.currentTimeMillis()-start<=(3000)){}

        ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
        fileChannel.read(byteBuffer);
        byteBuffer.flip();
        String moneyStr = StandardCharsets.UTF_8.decode(byteBuffer).toString();
    }
}

```

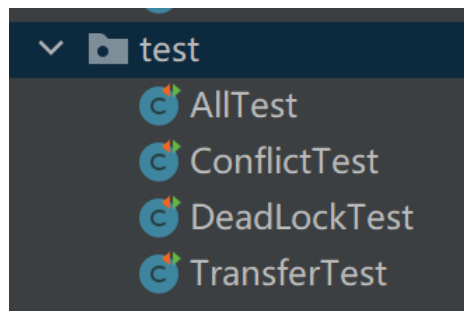
```

        if (moneyStr.length() > 0) {
            //读出 money
            money = Long.parseLong(moneyStr);
        }
        money = money + updateMoney;
        byteBuffer = ByteBuffer.allocate(1024);
        moneyStr = String.valueOf(money);
        byteBuffer.put(moneyStr.getBytes(StandardCharsets.UTF_8));
        byteBuffer.flip();
        fileChannel.truncate(0);
        fileChannel.write(byteBuffer);
        byteBuffer.clear();
        lock.close();
        System.out.println("批处理操作结束");
    } else {
        isWait = true;
    }
    fileChannel.close();
    return isWait;
}

```

测试设计：

使用 Junit 进行测试



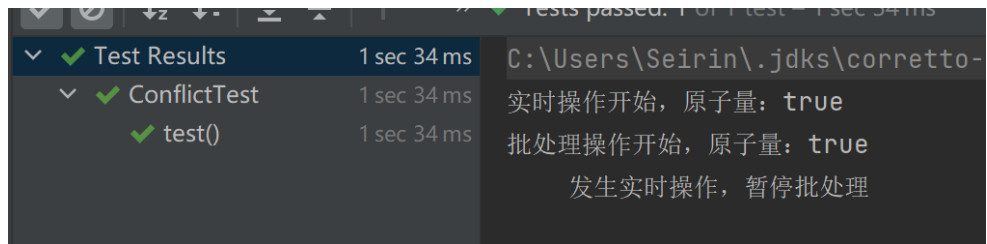
共设计 4 个测试类，分别为，所有功能测试（AllTest），抢占测试（ConflictTest），死锁测试（DeadLockTest），转账测试（TransferTest）。

所有功能测试：

通过 UserDao 运行基本方法，均可以实现。

抢占测试：

设计了先进行实时操作，再进行批处理。



转账测试:

设计了 A 对 B 转账, 同时或者间隔几秒 B 对 A 转账。

```
public class TransferTest {  
  
    @Test  
    void transfer1() throws IOException {  
        Thread thread1 = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                try {  
                    new UserDao().transferMoney( countName1: "1", countName2: "2", m: 3);  
                } catch (IOException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
        thread1.start();  
    }  
  
    @Test  
    void transfer2() throws IOException {  
        Thread thread1 = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                try {  
                    new UserDao().transferMoney( countName1: "2", countName2: "1", m: 3);  
                } catch (IOException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
        thread1.start();  
    }  
}
```

发生实时操作, 批处理暂停。

完成难度: 进阶

实现多用户同时操作的用户接口。不同用户可以使用各自的操作界面同时使用该系统。

程序可以多进程运行。

四、总结

这次作业使我巩固了操作系统的知识, 同时学习了线程和测试相关的知识。

操作系统的知识虽然学起来容易理解, 但实际操作便会无从下手, 因为不知道 Java 中对应的方法, 我也花费了很多时间去搜索学习。