华东师范大学数据学院上机实践报告

Computer Network & Coding Lab 1

课程名称： 计算机网络	**年级：2023**	**上机实践成绩：**
指导老师：张召	**姓名**：陈子谦	
上机实践名称：Java多线程编程	**学号**：10235501454	**上机实践日期**：

一、题目要求

任务一：字母数字交替打印
编写一个多线程程序，采用创建线程的两种方式，创建两个线程，一个线程负责打印字母，一个线程负责打印数字，要求最终打印出来的结果的为 **a1b23c456d7891e23456**，将关键代码和实现思路写入报告中。

任务二：多窗口售票系统模拟

编写一个多线程程序，模拟车站三个售票窗口同时进行售票、退票和新进票的操作，将关键代码和实现思路写入报告中。具体要求如下：

1. **售票**：
    - 每个窗口可以同时出售车票。
    - 购票时可能存在购买多张车票的情况。
    - 如果余票充足，则必须出售车票。
    - 如果余票不足，购票者可以选择：
        - 继续等待，直到有足够的车票。
        - 直接离开。
2. **退票**：
    - 每个窗口可以同时处理退票。
    - 退票时可能存在退多张车票的情况。
    - 退票成功后，余票数量增加。
3. **新进票**：
    - 车站会定期新进一定数量的车票。
    - 新进票后，如果有购票者在等待，应通知他们。
4. **同步要求**：
    - 多个窗口同时操作车票时，需要保证线程安全。
    - 购票时若无足量余票，购票者应进入等待状态，直到有新票进入或被其他购票者释放资源。
    - 退票和新进票操作应通知等待的购票者。
5. **输出要求**：
    - 每次售票、退票或新进票时，输出操作详情（如：窗口编号、操作类型、票数、余票数量等）。
    - 输出参考格式如下，不限定。

二、项目实现思路

项目要解决的核心问题是如何在并发环境下协调多个线程的执行顺序，以及如何安全地访问共享资源。

开发环境：

- **编程语言**：Java
- **JDK版本**：JDK 11
- **IDE工具**：IntelliJ IDEA

1. ThreadPrintDemo - 线程交替打印

经过分析发现输出格式**a1b23c456d7891e23456**有如下规律：

- 字母按顺序出现：a, b, c, d, e
- 每个字母后面跟着的数字个数依次增加：1个、2个、3个、4个、5个
- 数字按照1到9循环排列

实现思路与尝试过程：

最初我尝试用简单的共享变量和synchronized块来控制，但很快发现这种方式难以精确控制打印顺序。经过反复思考后，我决定采用信号量（Semaphore）这种更精细的同步机制：

1. 创建两个信号量：letterSemaphore初始值为1，numberSemaphore初始值为0
2. 字母线程先获取letterSemaphore的许可，打印一个字母，然后释放numberSemaphore
3. 数字线程获取numberSemaphore的许可，根据当前轮次打印指定数量的数字，然后释放letterSemaphore
4. 重复上述过程，直到所有字母和数字都打印完毕

2. TicketSystem - 多窗口售票系统

这个任务模拟了现实世界中的售票系统，涉及到更复杂的线程协作和资源管理。在设计时我考虑了以下几个关键点：

1. **线程安全访问票库**：多个窗口同时售票，必须确保不会出现超卖或数据不一致
2. **等待-通知机制**：当票不足时，购票者可能需要等待新票或退票
3. **模拟真实场景**：包括随机的售票、退票操作和用户行为模式

最终决定采用以下架构：

- **核心组件**：
  - TicketManager：中央票务管理器，负责所有与票务相关的同步操作
  - TicketWindow：售票窗口，作为独立线程运行
  - TicketSupplier：票源供应商，定期向系统添加新票
- **同步机制**：
  - 使用ReentrantLock而不是synchronized，因为它提供了更灵活的锁操作
  - 使用Condition实现等待-通知模式，比wait/notify更加精确
  - 通过lock/unlock确保原子操作，避免资源争用问题

三、功能实现情况

1. ThreadPrintDemo - 线程交替打印

关键代码实现与难点突破

```
// 使用Semaphore控制线程交替执行
private static Semaphore letterSemaphore = new Semaphore(1);
private static Semaphore numberSemaphore = new Semaphore(0);

// 方式一：继承Thread类创建线程
static class LetterThread extends Thread {
    private char[] letters = {'a', 'b', 'c', 'd', 'e'};

    @Override
    public void run() {
        try {
            for (int i = 0; i < letters.length; i++) {
                letterSemaphore.acquire(); // 获取字母打印许可
                System.out.print(letters[i]);
                numberSemaphore.release(); // 释放数字打印许可
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// 方式二：实现Runnable接口创建线程
static class NumberRunnable implements Runnable {
    private int[] numberCounts = {1, 2, 3, 4, 5}; // 每轮打印的数字个数
    private int currentNumber = 1;

    @Override
    public void run() {
        try {
            for (int count : numberCounts) {
                numberSemaphore.acquire(); // 获取数字打印许可

                // 打印指定数量的数字
                for (int i = 0; i < count; i++) {
                    System.out.print(currentNumber);
                    currentNumber++;
                    if (currentNumber > 9) {
                        currentNumber = 1; // 数字超过9重置为1
                    }
                }

                letterSemaphore.release(); // 释放字母打印许可
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

编写这段代码的过程中，我遇到了几个有趣的问题：

1. **初始值设置**：一开始我把两个信号量的初始值都设为1，结果导致输出顺序混乱。后来才意识到必须设置letterSemaphore=1和numberSemaphore=0，才能确保先打印字母。
2. **数字循环逻辑**：一开始没注意到数字需要循环到9后重置，导致输出结果不符合预期。调试了好几遍才发现这个小细节。
3. **异常处理**：信号量操作可能抛出InterruptedException，我本想简单地忽略，但后来意识到在实际应用中应该正确处理线程中断，这是一个良好的编程习惯。

执行流程与结果验证

整个程序的执行流程如下：

1. 初始时，letterSemaphore=1表示字母线程可以执行，numberSemaphore=0表示数字线程必须等待
2. 字母线程获取许可打印'a'，然后释放数字线程的许可
3. 数字线程获取许可，打印1个数字'1'（因为这是第一轮），然后释放字母线程的许可
4. 字母线程打印'b'，再次释放数字线程许可
5. 数字线程打印2个数字'23'（第二轮），然后释放字母线程许可
6. 以此类推，直到完成所有预定的打印任务

运行程序，输出结果符合预期：

```
a1b23c456d7891e23456
```

2. TicketSystem - 多窗口售票系统

关键数据结构

- **TicketManager**: 管理票务和同步操作的核心类
  - availableTickets: 当前可用票数
  - ReentrantLock: 保证线程安全的锁
  - Condition: 实现票务不足时的等待和通知

关键代码实现

```
// 票务管理类，负责管理票务和同步操作
class TicketManager {
    private int availableTickets; // 可用票数
    private final ReentrantLock lock = new ReentrantLock(); // 线程锁
    private final Condition ticketsAvailable = lock.newCondition(); // 条件变量

    // 售票操作
    public boolean sellTickets(int windowId, int count) {
        lock.lock();
        try {
            if (count <= availableTickets) {
                // 余票充足，直接售出
                availableTickets -= count;
```

```
                System.out.println("窗口 " + windowId + " 售出 " + count + " 张票，
余票: " + availableTickets);
                return true;
            } else {
                // 余票不足，根据随机概率决定是等待还是离开
                if (random.nextBoolean()) { // 50%的概率等待
                    System.out.println("窗口 " + windowId + " 余票不足，购票者等
待...");
                    try {
                        // 等待新票或退票
                        ticketsAvailable.await();

                        // 被唤醒后，再次尝试购票
                        if (count <= availableTickets) {
                            availableTickets -= count;
                            System.out.println("窗口 " + windowId + " 售出 " +
count + " 张票，余票: " + availableTickets);
                            return true;
                        } else {
                            System.out.println("窗口 " + windowId + " 余票仍不足，购
票者离开");
                            return false;
                        }
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        return false;
                    }
                } else { // 50%的概率直接离开
                    System.out.println("窗口 " + windowId + " 余票不足，购票者直接离
开");
                    return false;
                }
            }
        } finally {
            lock.unlock();
        }
    }

    // 退票操作
    public void refundTickets(int windowId, int count) {
        lock.lock();
        try {
            availableTickets += count;
            System.out.println("窗口 " + windowId + " 退票 " + count + " 张，余票: "
+ availableTickets);

            // 通知等待的购票者
            ticketsAvailable.signalAll();
        } finally {
            lock.unlock();
        }
    }

    // 新进票操作
    public void addNewTickets(int count) {
        lock.lock();
```

```
        try {
            availableTickets += count;
            System.out.println("新进票 " + count + " 张，余票: " +
availableTickets);

            // 通知等待的购票者
            ticketsAvailable.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

线程模型设计

1. **售票窗口线程**:
   - 随机选择售票或退票操作
   - 随机确定票数
   - 调用TicketManager相应方法执行操作
2. **票供应线程**:
   - 定期向系统添加新票
   - 通过TicketManager的addNewTickets方法执行

同步机制

1. **ReentrantLock**:
   - 确保多线程环境下票务数据的一致性和完整性
   - 保护availableTickets变量的安全访问
2. **Condition**:
   - 实现等待-通知模式
   - 当余票不足时，购票线程可以等待
   - 当有新票或退票时，通知等待的线程

运行结果

```
初始所有窗口票数为: 20
窗口 1 售出 3 张票，余票: 17
窗口 3 售出 2 张票，余票: 15
窗口 2 售出 4 张票，余票: 11
窗口 1 售出 5 张票，余票: 6
窗口 3 退票 2 张，余票: 8
窗口 2 余票不足，购票者等待...
新进票 12 张，余票: 20
窗口 2 售出 9 张票，余票: 11
窗口 3 售出 5 张票，余票: 6
窗口 1 退票 3 张，余票: 9
...
```

四、总结

这次实验主要围绕Java多线程编程展开，目的是让我们熟悉多线程的核心概念和实际应用场景。通过两个不同难度的任务，我将逐步掌握从基础到进阶的多线程编程技术。

1. **多线程并发控制**：
    - 通过本实验，我深入理解了多线程编程中的同步与互斥问题
    - 掌握了使用Semaphore控制线程按特定顺序执行的技巧
    - 学会了使用ReentrantLock和Condition实现复杂的线程协作
2. **线程安全设计**：
    - 理解了在共享资源访问时保证数据一致性的重要性
    - 掌握了通过锁机制保证关键操作的原子性
    - 学会了在设计多线程应用时如何避免死锁和饥饿现象
3. **条件变量的应用**：
    - 学习了条件变量在生产者-消费者模式中的应用
    - 理解了await()和signalAll()方法的机制和使用场景