

**HUBEI UNIVERSITY OF AUTOMOTIVE TECHNOLOGY**



**操作系统**

**实验二**

计算机222 章崇文202202296

电气与信息工程学院计算机工程系

2024年4月7日

**实验二 多线程编程实验**

**一、实验目的**

1. 掌握线程的概念，明确线程和进程的区别。

2. 学习Linux下线程创建方法及编程。

3. 了解线程的应用特点。

4． 掌握用锁机制访问临界区。

**二、实验内容**

1. 通过多线程模拟多窗口售票,在主线程下创建4个子线程，模拟4个售票窗口，假设有20张票待售，运行该程序看会有什么样的结果，分析原因。**（2分）**

参考程序

#include <iostream>

#include <thread>

#include <mutex>

#include <vector>

#include <chrono>

std::mutex ticket\_mutex; // 用于保护票数的互斥锁

int ticket\_sum = 20; // 假设有20张票待售

// 售票函数

void sell\_ticket(int window\_id) {

for (int i = 0; i < 20; ++i) {

std::unique\_lock<std::mutex> guard(ticket\_mutex); // 锁定互斥锁

if (ticket\_sum > 0) {

std::cout << "Window " << window\_id << " sells the " << (20 - ticket\_sum + 1) << "th ticket." << std::endl;

-- ticket\_sum; // 卖出一张票

}

guard.unlock(); // 解锁互斥锁

std::this\_thread::sleep\_for(std::chrono::seconds(1)); // 模拟售票操作耗时

}

}

int main() {

const int num\_windows = 4; // 售票窗口数量

std::vector<std::thread> threads;

// 创建并启动4个售票线程

for (int i = 0; i < num\_windows; ++i) {

threads.emplace\_back(sell\_ticket, i + 1);

}

// 等待所有售票线程完成

for (auto &thread : threads) {

thread.join();

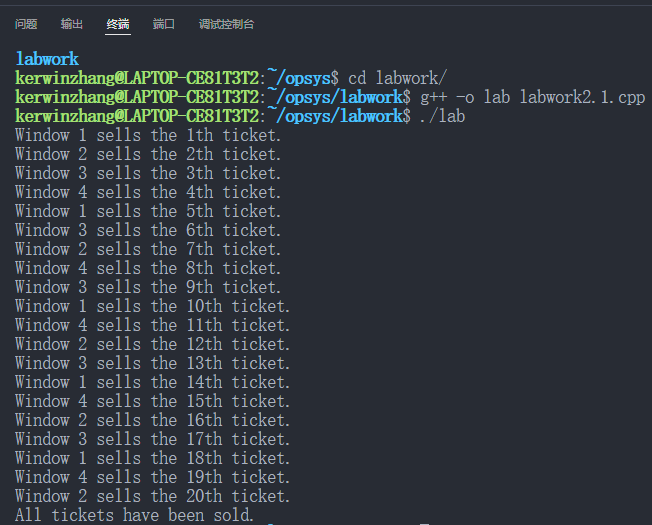
}

std::cout << "All tickets have been sold." << std::endl;

return 0;

}

运行结果：（部分截屏）



分析原因：

在这个程序中，我们首先定义了一个全局的互斥锁 ticket\_mutex 来保护全局变量 ticket\_sum，确保在多线程环境下对票数的访问是安全的。sell\_ticket 函数模拟了一个售票窗口的行为，每次尝试售票前都会先锁定互斥锁，检查票数是否足够，然后卖出票并更新票数，最后释放互斥锁。

在 main 函数中，我们创建了4个线程，每个线程都调用 sell\_ticket 函数，并传递了一个唯一的窗口ID。每个线程都会尝试卖出20张票。我们使用 std::vector<std::thread> 来存储所有的线程对象，并在创建线程后调用它们的 join 方法，等待所有线程完成售票操作。

使用 std::this\_thread::sleep\_for 函数来模拟售票操作需要花费一定的时间。这个函数会让当前线程暂停执行指定的时间，这里我们设置为1秒。

1. 修改第1题，显示哪个窗口卖出的票（显示线程ID）。**（2分）**

**#include <iostream>**

**#include <thread>**

**#include <mutex>**

**#include <vector>**

**#include <chrono>**

**std::mutex ticket\_mutex; // 用于保护票数的互斥锁**

**int ticket\_sum = 20; // 假设有20张票待售**

**// 售票函数**

**void sell\_ticket() {**

**for (int i = 0; i < 20; ++i) {**

**std::unique\_lock<std::mutex> guard(ticket\_mutex); // 锁定互斥锁**

**if (ticket\_sum > 0) {**

**std::cout << "Window " << std::this\_thread::get\_id() << " sells the " << (20 - ticket\_sum + 1) << "th ticket." << std::endl;**

**-- ticket\_sum; // 卖出一张票**

**}**

**guard.unlock(); // 解锁互斥锁**

**std::this\_thread::sleep\_for(std::chrono::seconds(1)); // 模拟售票操作耗时**

**}**

**}**

**int main() {**

**const int num\_windows = 4; // 售票窗口数量**

**std::vector<std::thread> threads;**

**// 创建并启动4个售票线程**

**for (int i = 0; i < num\_windows; ++i) {**

**threads.emplace\_back(sell\_ticket);**

**}**

**// 等待所有售票线程完成**

**for (auto &thread : threads) {**

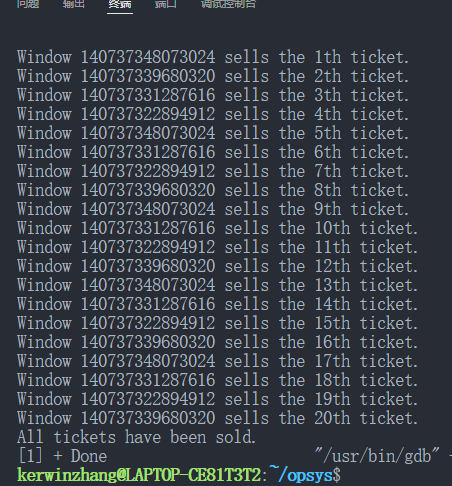
**thread.join();**

**}**

**std::cout << "All tickets have been sold." << std::endl;**

**return 0;**

**}**



1. 修改上题，用锁机制实现线程互斥进入临界区。 **（3分）**

#include <iostream>

#include <thread>

#include <mutex>

#include <vector>

std::mutex mtx; // 定义一个互斥锁

int ticket\_sum = 20; // 假设有20张票待售

// 售票函数

void sell\_ticket(int window\_id) {

for (int i = 0; i < 20; ++i) {

std::unique\_lock<std::mutex> lock(mtx); // 尝试获取互斥锁

if (ticket\_sum > 0) {

std::cout << "Window " << window\_id << " sells the " << (20 - ticket\_sum + 1) << "th ticket." << std::endl;

--ticket\_sum; // 卖出一张票

}

lock.unlock(); // 释放互斥锁

std::this\_thread::sleep\_for(std::chrono::seconds(1)); // 模拟售票操作耗时

}

}

int main() {

const int num\_windows = 4; // 售票窗口数量

std::vector<std::thread> threads;

// 创建并启动4个售票线程

for (int i = 0; i < num\_windows; ++i) {

threads.emplace\_back(sell\_ticket, i + 1);

}

// 等待所有售票线程完成

for (auto &thread : threads) {

thread.join();

}

std::cout << "All tickets have been sold." << std::endl;

return 0;

}

