

**HUBEI UNIVERSITY OF AUTOMOTIVE TECHNOLOGY**



**操作系统原理**

**实 验 报 告**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验项目 | 进程控制实验 | | | |
| 学生姓名 | 章崇文 | 学生学号 | 202202296 | |
| 学生班级 | 计算机222 | 完成日期 | 2024.4.25 | |
| 实验成绩 |  | 评阅日期 | |  |
| 评阅教师 |  | | | |

**湖北汽车工业学院实验报告**

班 号 学 号   姓 名

选课班中的序号  完成日期 年 月 日 至 节

**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实现互斥的方法。

5、掌握用信号量访问临界区实现互斥的方法。

6、掌握线程下用信号量实现同步操作的方法。

二、实验内容

1、运行下列程序，给出执行结果，并分析运行结果。（3分）

1） #include <stdio.h>

#include <pthread.h>

#include <unistd.h>

// 打印机

void printer(char \*str){

while(\*str!='\0')

{ putchar(\*str);

fflush(stdout);

str++;

sleep(1);

}

printf("\n");

}

// 线程一

void \*thread\_fun\_1(void \*arg)

{

char \*str = "hello";

printer(str); //打印

}

// 线程二

void \*thread\_fun\_2(void \*arg)

{

char \*str = "world";

printer(str); //打印

}

int main(void)

{

pthread\_t tid1, tid2;

// 创建 2 个线程

pthread\_create(&tid1, NULL, thread\_fun\_1, NULL);

pthread\_create(&tid2, NULL, thread\_fun\_2, NULL);

// 等待线程结束，回收其资源

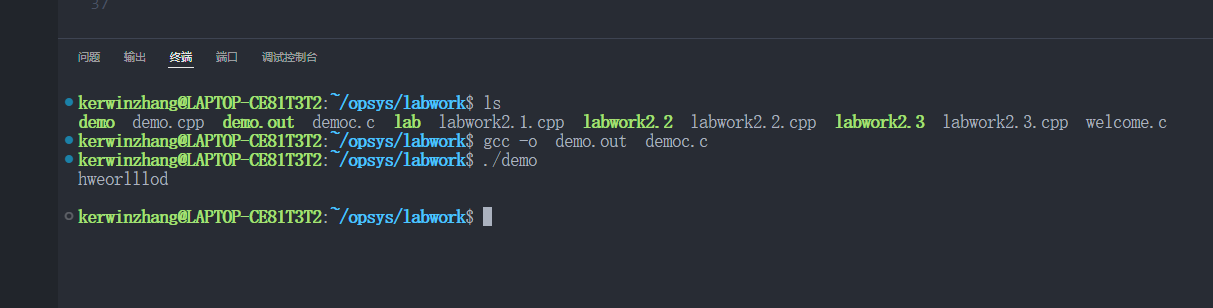
pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

return 0;

}

编译及执行过程和运行结果截屏：



结果分析：

线程调度：

线程调度是由操作系统的调度器进行管理的，具体哪个线程在何时运行取决于调度器的策略和当前系统的状态。在这段代码中，两个线程都尝试打印一个字符串，但是由于它们是并发运行的，它们的执行可能会交错进行。

输出交错：

当主线程创建了两个子线程后，它会等待这两个子线程完成。在子线程执行printer函数时，每个线程都会尝试逐个字符地打印它们各自的字符串。由于线程调度的不确定性，一个线程可能先打印了它字符串中的某些字符，然后另一个线程获得了CPU时间，开始打印它的字符串中的字符。

输出缓冲区：

标准输出stdout通常有一个缓冲区，用于优化输出操作。在某些系统上，当使用printf或putchar等函数时，输出可能会被存储在缓冲区中，直到缓冲区满了或者调用了fflush函数。然而，在这个例子中，由于每次打印字符后都调用了fflush，所以每次打印的字符都会立即显示，而不会被存储在缓冲区中。

1. #include <stdio.h>

#include <pthread.h>

#include <unistd.h>

pthread\_mutex\_t mutex\_x=PTHREAD\_MUTEX\_INITIALIZER;//定义并初始化锁

// 打印机

void printer(char \*str)

{

pthread\_mutex\_lock(&mutex\_x);//上锁

while(\*str!='\0')

{

putchar(\*str);

fflush(stdout);

str++;

sleep(1);

}

printf("\n");

pthread\_mutex\_unlock(&mutex\_x);//解锁

}

// 线程一

void \*thread\_fun\_1(void \*arg)

{

char \*str = "hello";

printer(str); //打印

}

// 线程二

void \*thread\_fun\_2(void \*arg)

{

char \*str = "world";

printer(str); //打印

}

int main(void)

{

pthread\_t tid1, tid2;

// 创建 2 个线程

pthread\_create(&tid1, NULL, thread\_fun\_1, NULL);

pthread\_create(&tid2, NULL, thread\_fun\_2, NULL);

// 等待线程结束，回收其资源

pthread\_join(tid1, NULL);

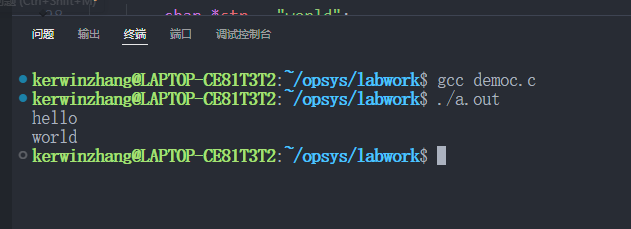
pthread\_join(tid2, NULL);

pthread\_mutex\_destroy(&mutex\_x); //销毁互斥锁

return 0;

}

编译及执行过程和运行结果截屏：



结果分析：

互斥锁mutex\_x在printer函数的开始处通过pthread\_mutex\_lock(&mutex\_x)被加锁，在函数的末尾通过pthread\_mutex\_unlock(&mutex\_x)被解锁。这意味着在任何时刻，只有一个线程可以进入printer函数进行打印操作。当一个线程尝试进入printer函数时，如果互斥锁已经被另一个线程持有，它将被阻塞，直到互斥锁被释放。

在主函数main中，两个线程被创建，但没有指定它们的执行顺序。线程调度器会根据系统的当前状态和调度策略来决定线程的执行顺序。在这个特定的例子中，可能的执行顺序有两种：

* **线程一先执行**：线程一首先获得CPU时间并进入printer函数，打印出"hello"字符串。在它打印完毕后，线程一释放互斥锁。此时，线程二获得CPU时间和互斥锁，打印出"world"字符串。
* **线程二先执行**：线程二首先获得CPU时间并进入printer函数，打印出"world"字符串。在它打印完毕后，线程二释放互斥锁。然后，线程一获得CPU时间和互斥锁，打印出"hello"字符串。

1. #include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#include <semaphore.h>

sem\_t semA; //声明一个名为semA的信号量变量

// 打印机

void printer(char \*str)

{

sem\_wait(&semA);//申请信号量（P操作）

while(\*str!='\0')

{

putchar(\*str);

fflush(stdout);

str++;

sleep(1);

}

printf("\n");

sem\_post(&semA);//释放信号量（V操作）

}

// 线程一

void \*thread\_fun\_1(void \*arg)

{

char \*str = "hello";

printer(str); //打印

}

// 线程二

void \*thread\_fun\_2(void \*arg)

{

char \*str = "world";

printer(str); //打印

}

int main(void)

{

pthread\_t tid1, tid2;

if(sem\_init(&semA, 0, 1)) //初始化信号量的值为1（二元信号量）

printf("error sem\_init!\n");

// 创建 2 个线程

pthread\_create(&tid1, NULL, thread\_fun\_1, NULL);

pthread\_create(&tid2, NULL, thread\_fun\_2, NULL);

// 等待线程结束，回收其资源

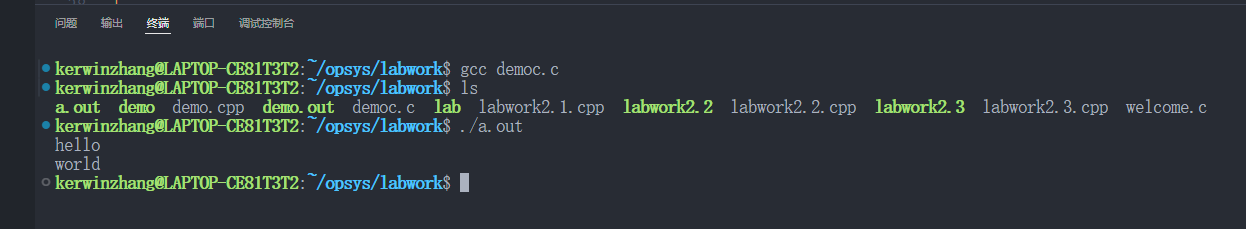
pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

return 0;

}

编译及执行过程和运行结果截屏：



结果分析：

线程一启动并进入printer函数，打印 "h"。

线程一打印完 "h" 后，可能因为某些原因（如操作系统调度）被挂起，线程二获得CPU时间并进入printer函数。

线程二打印 "w"。

线程一再次获得CPU时间，继续打印 "e"。

这个过程一直持续，直到 "hello" 完全打印出来，线程一释放互斥锁。

线程二接着打印完 "world"。

线程二打印完毕后，互斥锁被释放。

2、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，分析程序和执行结果。（2分）

<参考程序>

#include<pthread.h>

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

int ticket\_sum=20;

void \*sell\_ticket(void \*arg)

{

int i;

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

{

if(ticket\_sum>0)

{

sleep(1);

printf("sell the %dth\n",20-ticket\_sum+1);

ticket\_sum--;

}

}

return 0;

}

int main()

{

int flag,i;

pthread\_t tids[4];

for(i=0;i<4;i++)

{

flag=pthread\_create(&tids[i],NULL,&sell\_ticket,NULL);//创建线程

if(flag)

{

printf("pthread create error ,flag=%d",flag);

return flag;

}

}

sleep(20);

void \*ans;

for(i=0;i<4;i++)

{

flag=pthread\_join(tids[i],&ans);//等待线程结束

if(flag)

{

printf("tid=%lu,join erro flag=%d",tids[i],flag);

return flag;

}

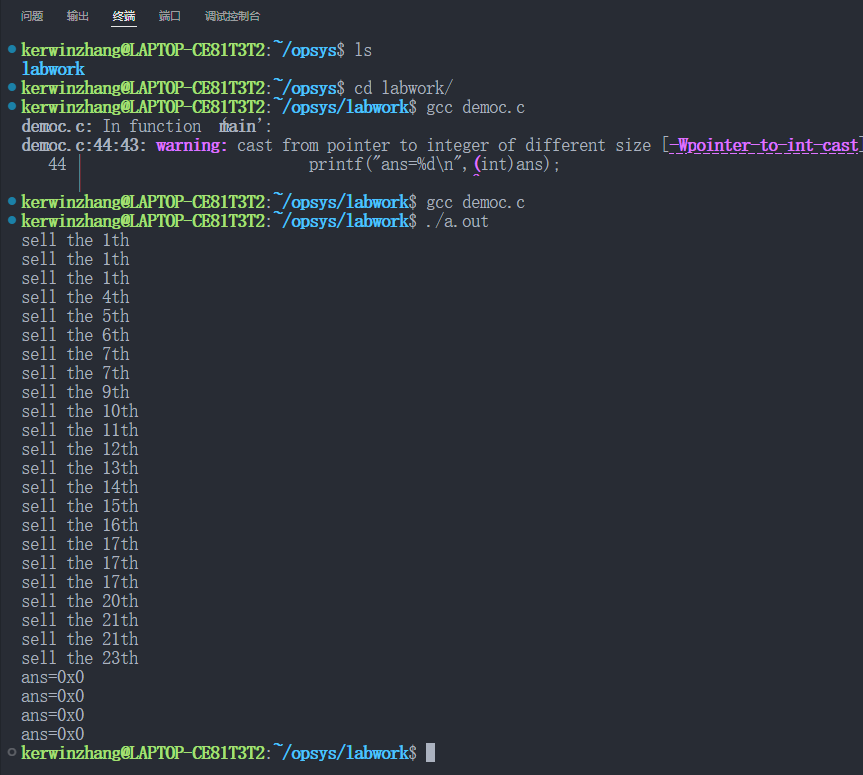
printf("ans=0x%llx\n",(unsigned long long)ans);

}

return 0;

}

给出编译及执行过程和运行结果：（部分截屏）



分析程序：

声明了一个全局变量 ticket\_sum，表示剩余的票数，初始值为 20。

定义了一个名为 sell\_ticket 的线程函数，它接受一个 void 指针作为参数，并返回一个 void 指针。该函数用于售票操作。

在 sell\_ticket 函数中，使用一个 for 循环来模拟售票的过程。在每次循环中，首先检查剩余票数是否大于 0，如果是，则继续执行售票操作。否则，线程将退出循环。

在售票操作中，通过调用 sleep(1) 函数来模拟售票的耗时。然后，打印出卖出的票的序号，即 20 - ticket\_sum + 1，并将剩余票数减一。

main 函数是程序的入口点。它声明了一些变量，包括 flag 和 i，以及一个用于存储线程 ID 的数组 tids。

在 main 函数中，使用一个 for 循环创建了四个线程。每次循环中，调用 pthread\_create 函数创建一个新线程，并将线程 ID 存储在 tids 数组中。同时，将线程函数指定为 sell\_ticket，并传递 NULL 作为参数。

在每次创建线程之后，检查返回值 flag 是否为 0，如果不是，则打印出错误信息并返回相应的错误码。

在创建完所有线程后，主线程通过调用 sleep(20) 函数等待一段时间，以确保所有的售票操作完成。

接下来，使用一个 for 循环调用 pthread\_join 函数，等待每个线程的结束。如果返回值 flag 不为 0，则打印出错误信息并返回相应的错误码。

在每次调用 pthread\_join 后，打印出线程的返回值 ans，并将其强制转换为 int 类型。

最后，主线程返回 0，程序结束。

结果分析：

售票的顺序并不是按照预期的顺序进行的。在输出中可以看到，有些票的序号会重复出现，例如 "sell the 1th" 和 "sell the 17th"。这是因为多个线程同时执行售票操作，它们之间的执行顺序是不确定的。

在售票过程中，存在一些线程间的竞争条件。由于没有对共享变量 ticket\_sum 进行同步操作，多个线程可能同时读取和修改该变量，导致出现问题。例如，可能会出现多个线程判断票数大于0的条件都为真，然后同时执行减票操作，导致票数减多了。

由于没有对这些操作进行同步，多个线程可以同时通过第一步的判断条件，进入售票操作。这就导致了重复售卖的情况。

在主线程中，通过调用 pthread\_join 函数等待线程结束，并打印出线程的返回值。然而，输出的结果为 0x0，表示线程的返回值为 NULL。这是因为在线程函数 sell\_ticket 中并没有使用 pthread\_exit 函数显式地返回一个值，因此默认返回值为 NULL。

假设有两个线程 A 和 B 同时通过了第一步的判断条件，它们都认为还有票可以售卖，然后同时执行第三步的打印操作。由于打印操作是原子的，不会被中断，所以线程 A 和 B 同时打印出售出票的序号。

接着，线程 A 和 B 分别执行第四步的减票操作，它们会分别将 ticket\_sum 减一。但是由于是并发执行，它们在减票之前并没有对 ticket\_sum 进行互斥访问的控制，所以可能会出现竞争条件。

例如，假设 ticket\_sum 的初始值为 2，线程 A 和 B 同时执行到第四步减票操作，它们都会将 ticket\_sum 减一，此时 ticket\_sum 的值变为 1。然后线程 A 和 B 分别继续执行后续的循环，但由于 ticket\_sum 的值已经被减少，它们可能会再次通过第一步的判断条件，导致重复售卖的情况。

3、修改上题，用锁机制实现线程互斥进入临界区，解决售票窗口超卖问题。要求给出编译及运行过程和结果截图。 （1分）

**#include <pthread.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <unistd.h>**

**int ticket\_sum = 20;**

**pthread\_mutex\_t mutex; // 声明互斥锁变量**

**void \*sell\_ticket(void \*arg)**

**{**

**int i;**

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

**{**

**pthread\_mutex\_lock(&mutex); // 锁定互斥锁，进入临界区**

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

**{**

**sleep(1);**

**printf("sell the %dth\n", 20 - ticket\_sum + 1);**

**ticket\_sum--;**

**}**

**pthread\_mutex\_unlock(&mutex); // 解锁互斥锁，离开临界区**

**}**

**return NULL;**

**}**

**int main()**

**{**

**int flag, i;**

**pthread\_t tids[4];**

**pthread\_mutex\_init(&mutex, NULL); // 初始化互斥锁**

**for (i = 0; i < 4; i++)**

**{**

**flag = pthread\_create(&tids[i], NULL, &sell\_ticket, NULL); // 创建线程**

**if (flag)**

**{**

**printf("pthread create error, flag=%d", flag);**

**return flag;**

**}**

**}**

**sleep(20);**

**void \*ans;**

**for (i = 0; i < 4; i++)**

**{**

**flag = pthread\_join(tids[i], &ans); // 等待线程结束**

**if (flag)**

**{**

**printf("tid=%lu, join error, flag=%d", tids[i], flag);**

**return flag;**

**}**

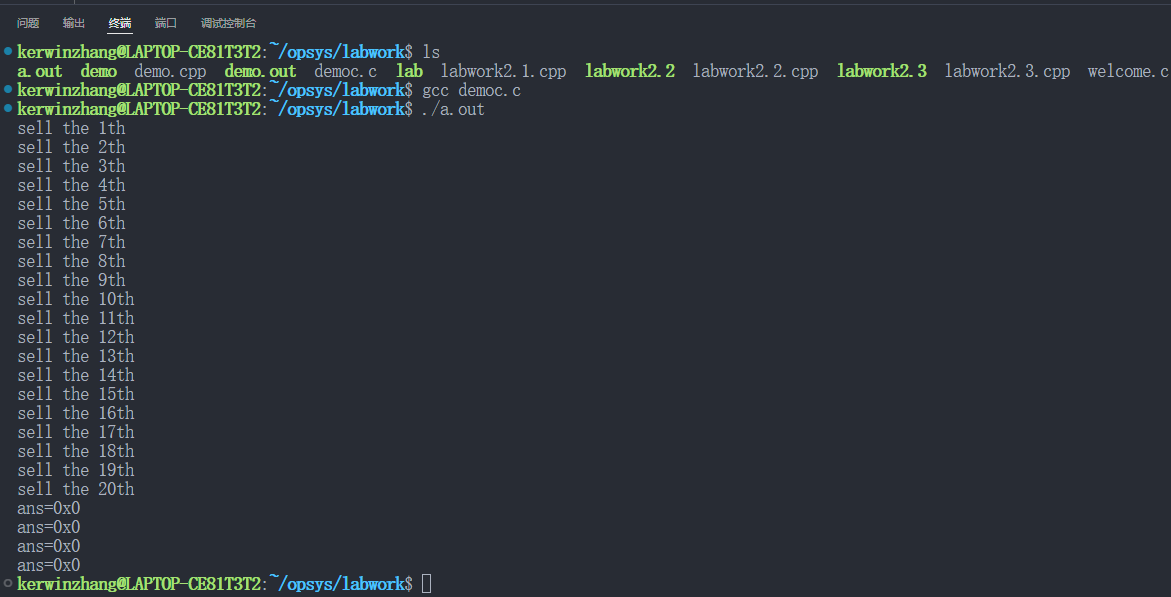
**printf("ans=0x%llx\n", (unsigned long long)ans);**

**}**

**pthread\_mutex\_destroy(&mutex); // 销毁互斥锁**

**return 0;**

**}**



4、修改实验内容2，用信号量实现线程互斥进入临界区，解决售票窗口超卖问题。要求给出编译及运行过程和结果截图。（1分）

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#include <unistd.h>

int ticket\_sum = 20;

sem\_t mutex; // 声明信号量变量

void \*sell\_ticket(void \*arg)

{

int i;

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

{

sem\_wait(&mutex); // 等待信号量，进入临界区

if (ticket\_sum > 0)

{

sleep(1);

printf("sell the %dth\n", 20 - ticket\_sum + 1);

ticket\_sum--;

}

sem\_post(&mutex); // 释放信号量，离开临界区

}

return NULL;

}

int main()

{

int flag, i;

pthread\_t tids[4];

sem\_init(&mutex, 0, 1); // 初始化信号量，初始值为1

for (i = 0; i < 4; i++)

{

flag = pthread\_create(&tids[i], NULL, &sell\_ticket, NULL); // 创建线程

if (flag)

{

printf("pthread create error, flag=%d", flag);

return flag;

}

}

sleep(20);

void \*ans;

for (i = 0; i < 4; i++)

{

flag = pthread\_join(tids[i], &ans); // 等待线程结束

if (flag)

{

printf("tid=%lu, join error, flag=%d", tids[i], flag);

return flag;

}

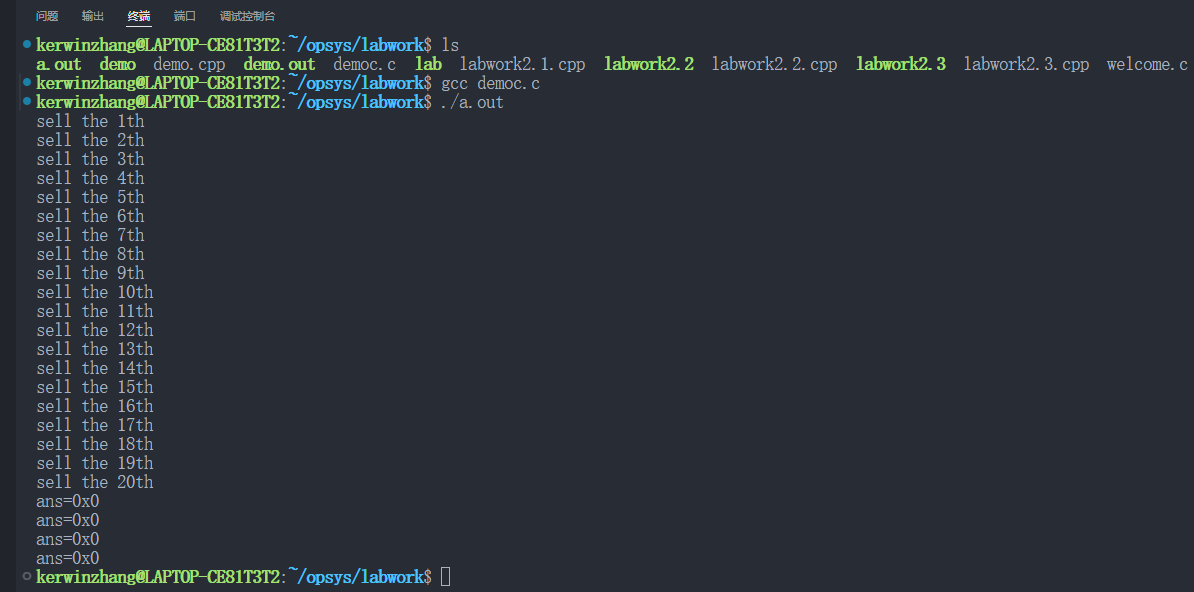
printf("ans=0x%llx\n", (unsigned long long)ans);

}

sem\_destroy(&mutex); // 销毁信号量

return 0;

}



5、利用线程和信号量机制实现司机售票员同步操作问题。要求给出编译及运行过程和结果截图。（2分）

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t driver\_sem, conductor\_sem;

void \*driver(void \*arg)

{

while (1)

{

sem\_wait(&driver\_sem); // 等待司机信号量

printf("Driver: Start driving.\n");

sleep(2); // 模拟驾驶过程

printf("Driver: Reached the destination.\n");

sem\_post(&conductor\_sem); // 唤醒售票员

}

return NULL;

}

void \*conductor(void \*arg)

{

while (1)

{

sem\_wait(&conductor\_sem); // 等待售票员信号量

printf("Conductor: Start selling tickets.\n");

sleep(1); // 模拟售票过程

printf("Conductor: Tickets sold.\n");

sem\_post(&driver\_sem); // 唤醒司机

}

return NULL;

}

int main()

{

pthread\_t driver\_thread, conductor\_thread;

sem\_init(&driver\_sem, 0, 1); // 初始化司机信号量，初始值为1

sem\_init(&conductor\_sem, 0, 0); // 初始化售票员信号量，初始值为0

pthread\_create(&driver\_thread, NULL, driver, NULL); // 创建司机线程

pthread\_create(&conductor\_thread, NULL, conductor, NULL); // 创建售票员线程

pthread\_join(driver\_thread, NULL); // 等待司机线程结束

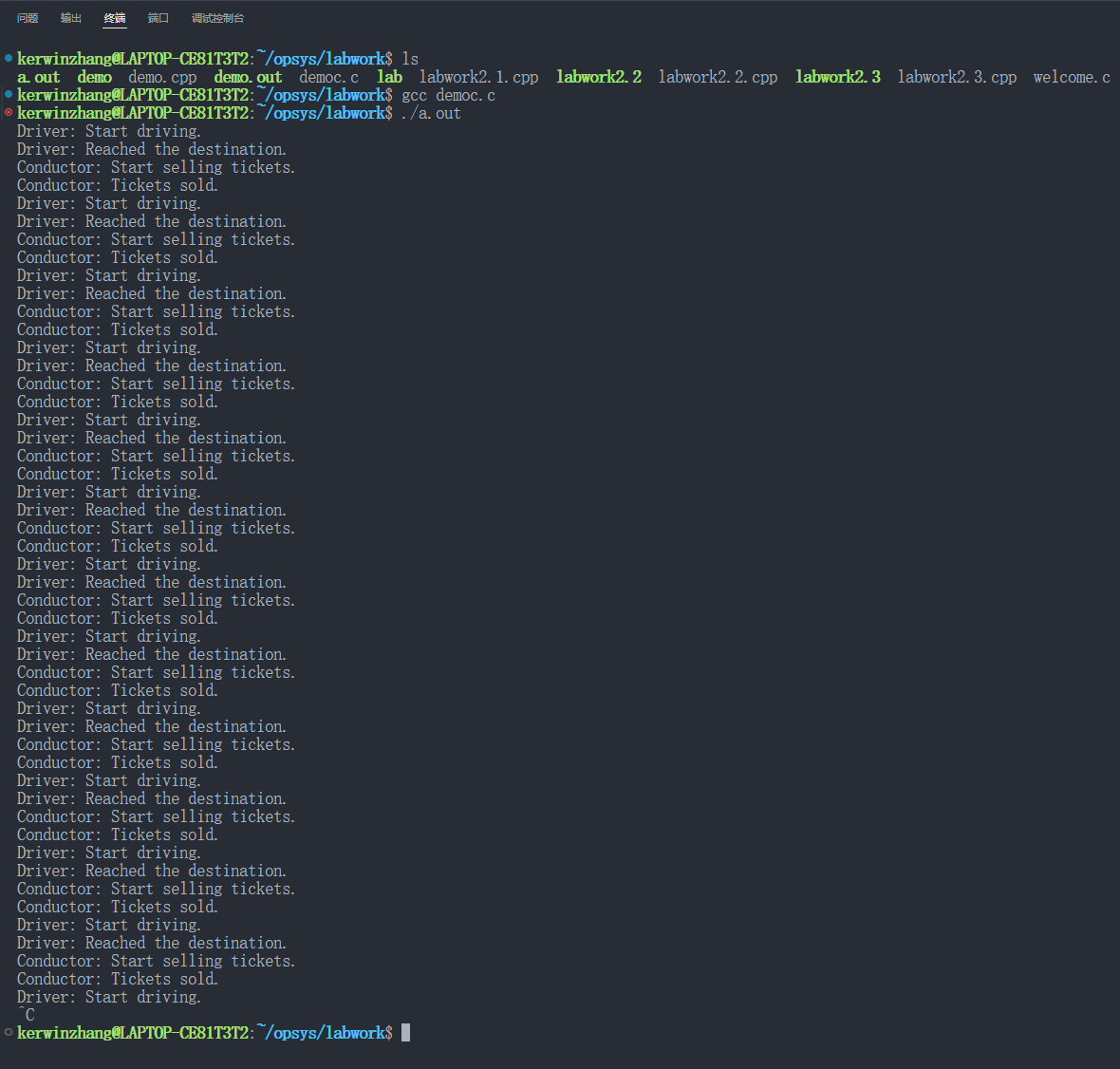
pthread\_join(conductor\_thread, NULL); // 等待售票员线程结束

sem\_destroy(&driver\_sem); // 销毁司机信号量

sem\_destroy(&conductor\_sem); // 销毁售票员信号量

return 0;

}



6.利用线程和信号量实现生产者消费者问题（涉及线程同步和互斥问题）。（附加题）

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int count = 0; // 缓冲区中的数据数量

sem\_t empty\_sem; // 空缓冲区的信号量

sem\_t full\_sem; // 满缓冲区的信号量

pthread\_mutex\_t mutex; // 互斥锁

void \*producer(void \*arg)

{

int item = 1;

while (1)

{

// 等待空缓冲区

sem\_wait(&empty\_sem);

// 获取互斥锁

pthread\_mutex\_lock(&mutex);

// 生产数据

buffer[count] = item;

printf("Produced: %d\n", item);

count++;

// 释放互斥锁

pthread\_mutex\_unlock(&mutex);

// 发送满缓冲区信号

sem\_post(&full\_sem);

item++;

// 模拟生产时间

sleep(1);

}

return NULL;

}

void \*consumer(void \*arg)

{

while (1)

{

// 等待满缓冲区

sem\_wait(&full\_sem);

// 获取互斥锁

pthread\_mutex\_lock(&mutex);

// 消费数据

int item = buffer[count - 1];

printf("Consumed: %d\n", item);

count--;

// 释放互斥锁

pthread\_mutex\_unlock(&mutex);

// 发送空缓冲区信号

sem\_post(&empty\_sem);

// 模拟消费时间

sleep(2);

}

return NULL;

}

int main()

{

pthread\_t producer\_thread, consumer\_thread;

// 初始化信号量和互斥锁

sem\_init(&empty\_sem, 0, BUFFER\_SIZE);

sem\_init(&full\_sem, 0, 0);

pthread\_mutex\_init(&mutex, NULL);

// 创建生产者线程和消费者线程

pthread\_create(&producer\_thread, NULL, producer, NULL);

pthread\_create(&consumer\_thread, NULL, consumer, NULL);

// 等待线程结束

pthread\_join(producer\_thread, NULL);

pthread\_join(consumer\_thread, NULL);

// 销毁信号量和互斥锁

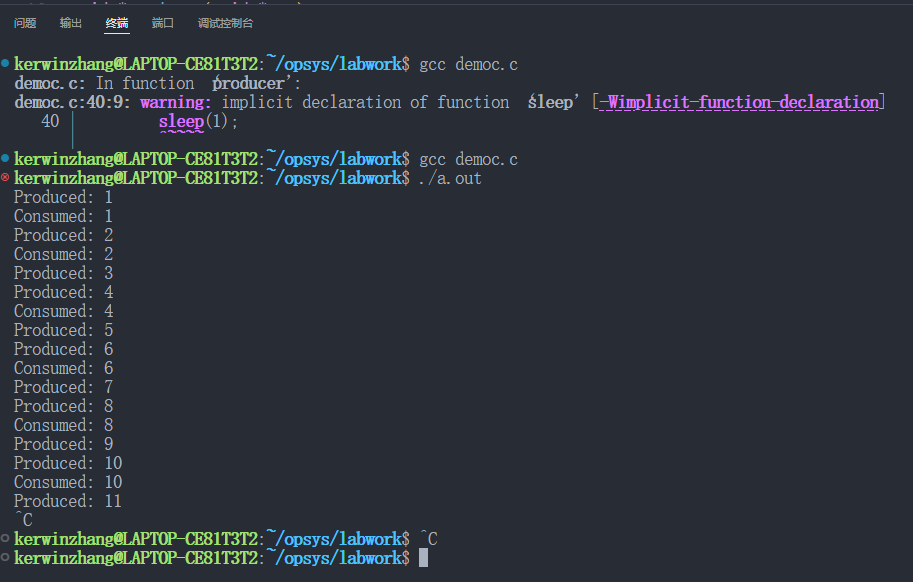
sem\_destroy(&empty\_sem);

sem\_destroy(&full\_sem);

pthread\_mutex\_destroy(&mutex);

return 0;

}



三、实验总结和体会

**次实验的主要目的是深入了解和掌握线程的概念，明确线程与进程的区别，并学习在Linux环境下线程的创建方法及相关编程技巧。同时，通过实践来探究线程的应用特点，并熟悉线程间互斥与同步的实现方法。**

**在实验过程中，我首先回顾了线程与进程的基本概念，理解了线程是进程内的一条执行路径，多个线程共享进程的资源，而进程则是系统资源分配的基本单位。这一理解对我后续的实验操作至关重要，它帮助我明确了线程与进程在资源占用、执行方式等方面的区别。**

**在Linux环境下，我学习了使用pthread库来创建和管理线程。通过编写简单的多线程程序，我掌握了线程创建、线程间通信以及线程同步的基本方法。在这个过程中，我深刻体会到了多线程编程的复杂性和挑战性，尤其是在处理线程间数据共享和同步问题时。**

**实验中，我还重点学习了使用锁机制和信号量来实现线程间的互斥和同步。通过实践，我了解到锁机制可以有效地保护临界区，防止多个线程同时访问共享资源，从而避免数据竞争和不一致的问题。而信号量则是一种更灵活的同步机制，它不仅可以实现互斥，还可以用于控制多个线程的执行顺序。**

**通过这次实验，我深刻感受到了线程编程的魅力和挑战。线程编程可以大大提高程序的并发性和执行效率，但同时也带来了线程间通信和同步的复杂问题。因此，在实际编程中，我们需要根据具体的应用场景和需求来选择合适的线程编程模型和方法。**

**此外，我也认识到了学习与实践相结合的重要性。理论知识是基础，但只有通过实践才能真正掌握和理解这些知识。在未来的学习和工作中，我将更加注重理论与实践的结合，不断提升自己的编程能力和问题解决能力。**