操作系统

实验一 银行柜员服务问题
实验报告

姓名：张凯

班级：无 97

学号：2019011159

项目代码链接：*https://github.com/zhangkai0425/OS*

一、实验目的

1.通过对进程间通信同步/互斥问题的编程实现，加深理解信号量和 P、V 操作的原理。

2.对 Windows 或 Linux 涉及的几种互斥、同步机制有更进一步的了解。

3.熟悉 Windows 或 Linux 中定义的与互斥、同步有关的函数。

二、实验内容

问题描述

银行有 n 个柜员负责为顾客服务，顾客进入银行先取一个号码，然后等着叫号。当某个柜员空闲下来，就叫下一个号。

编程实现该问题，用 P、V 操作实现柜员和顾客的同步。

实现要求

1.某个号码只能由一名顾客取得.

2.不能有多于一个柜员叫同一个号；

3.有顾客的时候，柜员才叫号；

4.无柜员空闲的时候，顾客需要等待；

5.无顾客的时候，柜员需要等待。

实现提示

1.互斥对象：顾客拿号，柜员叫号；

2.同步对象：顾客和柜员；

3.等待同步对象的队列：等待的顾客，等待的柜员；

4.所有数据结构在访问时也需要互斥。

三、实验设计

本题的目的是利用操作系统的基本互斥和同步机制，实现并解决银行柜员服务问题。

首先，要满足记录顾客的进入时间、开始服务时间、离开时间以及服务柜员号，最佳的方式是封装顾客为一个类，或者结构体。考虑到 C++的结构体已经可以首先相应的要求，就不再写一个顾客类了。程序中使用两个结构体，利用结构

体 1，记录进入顾客的信息；利用结构体 2，记录离开顾客的信息。

之后，创建顾客和柜员的函数，并分别交给不同的线程运行。为保证多线程程序的正确性，引入互斥和同步信号量，包括柜台互斥信号量、顾客取号信号量和保证柜台和顾客能够等待的同步信号量，记录了现有待服务顾客的数量。

接下来的工作即对此思路进行实现，由于此实验是第一个实验，内容也比较简单，具体的分析见代码实现部分，不再赘述。

四、代码实现

实验代码全部采用 **C++11** 编写，在 **Linux** 上运行，由 **g++**编译器进行编译。

首先实现的是进入顾客和离开顾客的结构体，供之后的信息记录和打印输出，结构体如下：

```
//进入顾客信息结构体
struct cus_in{
    int cus_number;//顾客编号
    int time_in;//进入时间
    int time_serve;//服务时间
};
//离开顾客信息结构体
struct cus_out{
    int cus_number;//顾客编号
    int time_in;//进入时间
    int time_serve;//服务时间
    double time_beginserve; //开始服务的时间
    int counter_no; //柜台号
    double time_served; //结束服务时间
};
```

接着，实现了柜员服务和顾客的函数，柜员服务操作基本就是在互斥锁的条件下(确保同一个柜台只能服务一个顾客)，然后进行待服务顾客数量的 P 操作，即服务一个顾客，待服务顾客减 1。顾客函数基本就是在互斥锁的条件下(确保同一时刻不会多个顾客同时取号)，进行待服务顾客数量的 V 操作，即一个顾客取完号，释放一个待服务顾客资源。两个函数如下所示：

```
void PVcounter(int id){
    while (true) {
        //等待顾客出现 P 操作
        sema_customer.P();
        //占用柜台资源
        counter_mutex[id].lock();
        time_t time_start = time(NULL);
        // cout<<time_start<<endl;
        //模拟服务时间
        int now_serve = customer_serve;
        customer_serve ++;
        cout<<"服务时长:"<<cus_outs[now_serve].time_serve<<" counter id:"<<id<<endl;
        std::this_thread::sleep_for(std::chrono::seconds(cus_outs[now_serve].time_serve));
        time_t time_end = time(NULL);
        cus_outs[now_serve].time_beginserve = time_start - time_begin;
        cus_outs[now_serve].time_served = time_end - time_start;
        cus_outs[now_serve].counter_no = id;
        //此顾客服务结束——>顾客号+1
        //释放柜台资源
        counter_mutex[id].unlock();
        //总服务过的人数+1
        customer_served ++;
        cout<<"服务结束:总服务过的人数:"<<customer_served<<endl;
    }
}
void PVcustomer(int id){
    //模拟睡眠至进入线程的时间
    std::this_thread::sleep_for(std::chrono::seconds(cus_ins[id].time_in));
    cout<<"顾客进入银行 id = " <<cus_ins[id].cus_number<<" 进入时
间:"<<cus_ins[id].time_in<<" 需要服务时长:"<<cus_ins[id].time_serve<<endl;
    //占用银行取号机
    customer_mutex.lock();
    cus_out tmp_cus;
    tmp_cus.cus_number = cus_ins[id].cus_number;
    tmp_cus.time_in = cus_ins[id].time_in;
    tmp_cus.time_serve = cus_ins[id].time_serve;
    cus_outs.push_back(tmp_cus);
    //取完号之后释放互斥量
    customer_mutex.unlock();
    //顾客开始等待柜台服务 V 操作
    sema_customer.V();
}
```

最后就是供进程的全局互斥、同步信号量的实现。实验中用到的全局互斥、同步信号量如下所示：

类型	变量	描述
std::mutex	counter_mutex[N_Counter]	柜台互斥信号量，防止一个柜台服务多个顾客
	customer_mutex	顾客互斥量，防止一个顾客同时取号
Semaphore	sema_customer	队列长度同步信号量

互斥锁变量直接调用 C++ std::mutex 类进行实现，同步信号量自己用 C++实现，如下所示：

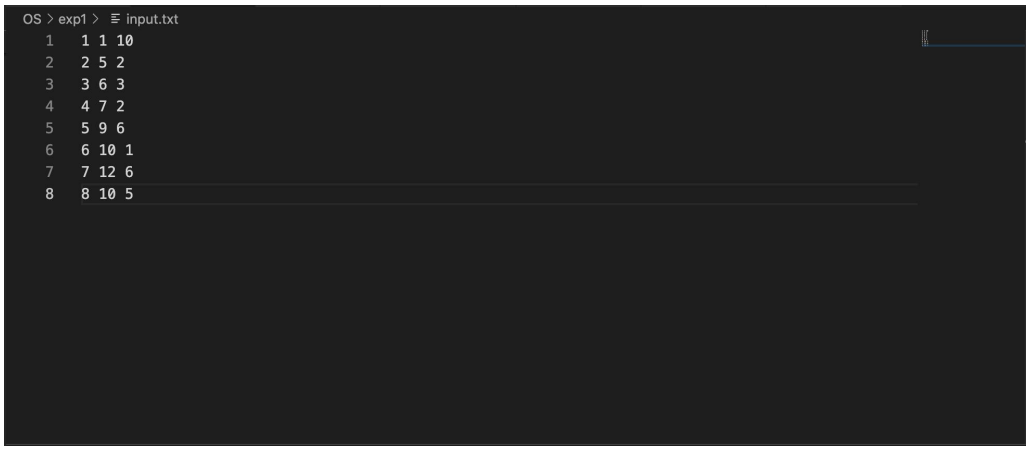
```
//信号量的实现
class Semaphore
{
public:
    Semaphore(int count=0) : count(count) {}
    //V 操作，唤醒
    void V()
    {
        std::unique_lock<std::mutex> unique(mt);
        ++count;
        if (count <= 0)
            cond.notify_one();
    }
    //P 操作，阻塞
    void P()
    {
        std::unique_lock<std::mutex> unique(mt);
        --count;
        if (count < 0)
            cond.wait(unique);
    }
    void getcount(){
        cout<<this->count<<endl;
    }
private:
    int count;
    mutex mt;
    condition_variable cond;
};
```

唯一的不足是这样实现的信号量无法像 Windows 那样 CloseHandle 直接销毁，但我没有 Windows，只有 Mac 和 Linux，所以我最后结束进程都是 std ::thread 析构函数强行结束的，这是自己实现信号量的一个缺点所在。

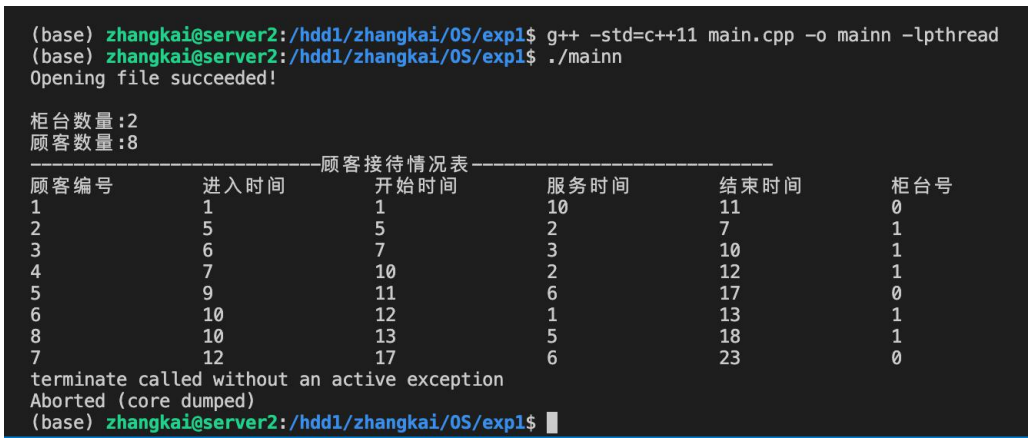
在实现多线程时，直接使用了 C++的 std::thread 类创建线程；在设置互斥锁变量时，也直接调用了 C++的 std::mutex；在实现同步信号量时，我直接重新写了一个 Semaphore 类，从而避免了调用系统的 Semaphore。这样，可以有效避免了 Linux 系统和 Windows 系统平台不同的问题，但实际上我只有 Mac 电脑，和 Linux 比较类似，代码也是在 Linux 服务器上运行的，没有在 Windows 上尝试过。全部代码见附录 main.cpp。

五、实验结果

输入数据如下：



输出结果如下：



可见，在 8 个顾客，2 个柜台的情况下，程序运行无误。

其实两个柜台不容易直接看出程序是否有问题，文件顾客内容信息不变，柜

台数量改成 1 个柜台，输出结果如下：

```
(base) zhangkai@server2:/hdd1/zhangkai/OS/exp1$ g++ -std=c++11 main.cpp -o mainn -lpthread
(base) zhangkai@server2:/hdd1/zhangkai/OS/exp1$ ./mainn
Opening file succeeded!

柜台数量:1
顾客数量:8
------------------------顾客接待情况表------------------------
顾客编号        进入时间        开始时间        服务时间        结束时间        柜台号
1               1               1               10              11              0
2               5               11              2               13              0
3               6               13              3               16              0
4               7               16              2               18              0
5               9               18              6               24              0
6               10              24              1               25              0
8               10              25              5               30              0
7               12              30              6               36              0
terminate called without an active exception
Aborted (core dumped)
(base) zhangkai@server2:/hdd1/zhangkai/OS/exp1$
```

从结果来看，程序运行无误，能够合理等待、服务，没有冲突发生。

六、实验思考题

1.柜员人数和顾客人数对结果分别有什么影响?

答:

柜员人数会影响总的时间长度和时间利用率。具体来说，柜员人数越少，总的等待时间越长，最极端情况是 1 个柜员，则总的等待时间最长，从第一个顾客进入到最后一个顾客离开的时间最长；柜员人数越多，总的等待时间越短，最极端情况是无穷多个柜员，则所有顾客都不需要等待和排队，总等待时间最短。时间利用率上来说，柜员人数越少，时间利用率越高，因为极端下 1 个柜员，所有的时间柜员都在工作，时间利用率为 100%，柜员数越多，柜员空闲的时间空隙只会多不会少，时间利用率越低。

2.实现互斥的方法有哪些？各自有什么特点？效率如何？

答:

结合课堂内容，主要的实现互斥的方法及特点主要如下：

实现互斥的方法	特点	效率
信号量	P、V 原语实现，简洁易懂，可以解决忙等待问题	效率较高，但 PV 必须成对使用，操作分散，大型程序容易出现编程问题

锁变量	对临界资源加锁，在同一时刻只能一个进程访问，用法简单	效率较高，但会出现忙等待现象
管程	高级进程通信的方式之一，代码可读性好，正确性高	效率较高，但应用不广泛，不是所有平台编译器都支持管程
消息传递	适用于分布式系统	对于大型的分布式系统来说可以实现高效率，对于 PC 来说过于复杂
硬件指令方法——禁止中断法	简单粗暴	效率低，系统可靠性差，代价较高，不适用于多处理器
PETERSON 算法	不会像普通转轮算法出现互斥访问的现象，能够正常解决互斥问题，	效率较高，但也会出现忙等待的错误

七、实验小结

通过本次实验，我熟悉了多线程编程的基本流程，也熟悉了 C++多线程编程的基本语法。由于没有 Windows 电脑，实验全部代码都是在 Mac 和 Linux 上运行的，在编程和 debug 的过程中，我也真正去熟悉了 Linux 操作系统的基本系统调用，也体会到了 Linux 的简洁。大部分同学都是用 Windows 平台进行编写，因此，Linux 相关的代码和介绍都比较少，在查阅相关 API 和 C++特性的过程中，我也进一步熟悉了 Linux 多线程编程的相关知识和技巧。

实验中也有一些不足，比如最后强行析构函数关闭线程，其实并不是一种很好的方式。如果调用 Linux 的 pthread 和信号量，而不是自己实现信号量，或许可以解决这个问题。(对于 Windows 来说，调用 CloseHandle()能够很方便地关闭信号量，进而线程也可以很好地关闭，因此不需要强行终止)限于时间，就不再重新编写了。

八、代码附录

```
//  main.cpp
//  操作系统大作业实验1
//  Created by 张凯 on 2022/5/9.
//  Copyright © 2022 张凯. All rights reserved.
#include <iostream>
#include <vector>
#include <mutex>
#include <thread>
#include <fstream>
#include <unistd.h>
#include <condition_variable>

using namespace std;

//柜台数量和最大顾客数
const int N_Counter = 2;
const int N_Customer = 20;

int customer_number = 0;//顾客总数
int customer_serve = 0; //正在服务顾客序号
int customer_served = 0;//已经服务过的顾客数
int counter_number = 0; //柜台序号

//读取顾客信息数据
struct cus_in{
    int cus_number;//顾客编号
    int time_in;//进入时间
    int time_serve;//服务时间
};

struct cus_out{
    int cus_number;//顾客编号
    int time_in;//进入时间
    int time_serve;//服务时间
    double time_beginserve; //开始服务的时间
    int counter_no; //柜台号
    double time_served; //结束服务时间
};
//信号量的实现
class Semaphore
{
public:
    Semaphore(int count=0) : count(count) {}
    //V操作，唤醒
    void V()
    {
        std::unique_lock<std::mutex> unique(mt);
        ++count;
```

```
        if (count <= 0)
            cond.notify_one();
    }
    //P操作，阻塞
    void P()
    {
        std::unique_lock<std::mutex> unique(mt);
        --count;
        if (count < 0)
            cond.wait(unique);
    }
    void getcount(){
        cout<<this->count<<endl;
    }
private:
    int count;
    mutex mt;
    condition_variable cond;
};

vector<cus_in>cus_ins;
vector<cus_out>cus_outs;

Semaphore sema_customer(0);                                    //同步信号量,确保柜台处于等待状态情况
std::mutex counter_mutex[N_Counter];                           //柜台互斥量,防止一个柜台服务多个顾客
std::mutex customer_mutex;                                     //顾客互斥量,防止多个顾客同时取相同的号

time_t time_begin = time(NULL);
void PVcounter(int id){
    while (true) {
        //等待顾客出现 P操作
        sema_customer.P();
        //占用柜台资源
        counter_mutex[id].lock();
        time_t time_start = time(NULL);
        // cout<<time_start<<endl;
        //模拟服务时间
        int now_serve = customer_serve;
        customer_serve ++;
        // cout<<"服务时长:"<<cus_outs[now_serve].time_serve<<" counter id:"<<id<<endl;

std::this_thread::sleep_for(std::chrono::seconds(cus_outs[now_serve].time_serve));
        time_t time_end = time(NULL);
        cus_outs[now_serve].time_beginserve = time_start - time_begin;
        cus_outs[now_serve].time_served = time_end - time_start;
        cus_outs[now_serve].counter_no = id;
```

```
        //此顾客服务结束——>顾客号+1
        //释放柜台资源
        counter_mutex[id].unlock();
        //总服务过的人数+1
        customer_served ++;
        // cout<<"服务结束:总服务过的人数:"<<customer_served<<endl;
    }
}
void PVcustomer(int id){
    //模拟睡眠至进入线程的时间
    std::this_thread::sleep_for(std::chrono::seconds(cus_ins[id].time_in));
    // cout<<"顾客进入银行 id = " <<cus_ins[id].cus_number<<" 进入时间:"
<<cus_ins[id].time_in<<" 需要服务时长:"<<cus_ins[id].time_serve<<endl;
    //占用银行取号机
    customer_mutex.lock();
    cus_out tmp_cus;
    tmp_cus.cus_number = cus_ins[id].cus_number;
    tmp_cus.time_in = cus_ins[id].time_in;
    tmp_cus.time_serve = cus_ins[id].time_serve;
    cus_outs.push_back(tmp_cus);
    //取完号之后释放互斥量
    customer_mutex.unlock();
    //顾客开始等待柜台服务 V操作
    sema_customer.V();
}

int main(){
    ifstream file;
    file.open("input.txt",ios::in);
    if(!file.good()){
        cout<<"Opening file failed:EXIT(0)"<<endl;
        exit(0);
    }
    else
        cout<<"Opening file succeeded!"<<endl;
    cus_in tmp_cus;
    while (!file.eof()) {
        file>>tmp_cus.cus_number>>tmp_cus.time_in>>tmp_cus.time_serve;
        cus_ins.push_back(tmp_cus);
    }
    customer_number = cus_ins.size();
    vector<std::thread> Thread;

    //创建顾客线程
    for(int i=0;i<cus_ins.size();i++)
        Thread.push_back(std::thread(PVcustomer,i));
    //创建柜台线程
    for(int j=0; j<N_Counter;j++)
        Thread.push_back(std::thread(PVcounter,j));
```

```
    //等待线程结束,关闭线程
    while(customer_served<customer_number);
    cout<<endl;
    cout << "柜台数量:" << N_Counter << endl;
    cout << "顾客数量:" << customer_number << endl;
    cout<<"--------------------------顾客接待情况表-----------------------------"<<endl;
    cout << "顾客编号" << '\t'
         << "进入时间" << '\t'
         << "开始时间" << '\t'
         << "服务时间" << '\t'
         << "结束时间" << '\t'
         << "柜台号   " << endl;

    for(auto x:cus_outs)
        cout << x.cus_number << '\t' << '\t' << x.time_in << '\t' << '\t' <<
x.time_beginserve << '\t' << '\t' << x.time_serve << '\t' << '\t' << x.time_beginserve
+ x.time_serve << '\t' << '\t' << x.counter_no << endl;

    for (int i = 0; i < Thread.size(); i++)
        Thread[i].~thread();
}
```