《操作系统》实验 1：进程间同步/互斥问题——银行柜员服务问题

班级:无 22　姓名:**王炜致**　学号:2022010542

1　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.

2　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.

3　设计思路

操作系统平台	编程语言	头文件
Windows 11	C++	<iostream>,<vector>,<queue>,<fstream>,<ctime> 及 mingw.thread.h,mingw.mutex.h

表 1: 实验背景

通过超参数指定**银行柜员总数 ALL_SERVERS**；定义**顾客队列数据结构 cqueue**，并配置**互斥量 mtx_cqueue** 用于保证柜员线程对该队列的互斥访问；定义**信号量 cavail** 对应顾客队列长度（也可用 cqueue.size() 代替），用于同步顾客和柜员行为（队列中无顾客则柜员空闲时处于就绪等待状态）；定义**整型数 served**，用于统计已经服务的顾客数量，与顾客总数比较能够判断是否退出线程循环.

定义柜员线程函数 Server、顾客线程函数 Customer，并分别为所有柜员、顾客设置线程函数. 函数中规划了柜员、顾客两种角色的工作流程——顾客仅“负责”**到达取号、加入队列**；柜员“负责”**等待顾客、（竞争）叫号、服务顾客、结束服务（从队列中删除当前顾客等）**. 注意到不需要设置等待的柜员队列，如果同时有多个就绪柜员，可以随机地竞争顾客，而 cqueue 保证了互斥访问，这显然符合空闲让进原则、忙则等待原则及现实生活常识.

为了更好地实现题目中的输入输出任务，定义**结构体 customer** 用于存放顾客序号、进入银行时间、需要服务时间、开始服务时间、离开银行时间和服务柜员号并配置打印函数；引入**时钟变量 T_beg**，便于获取开始服务时间.

为了方便地实现互斥量 **（P(mutex),V(mutex) 函数）**及线程功能，考虑引用头文件 mingw.thread.h,mingw.mutex.h，注：引用 <thread><mutex> 时发现无法正常读取，参考教程 https://blog.csdn.net/Flag_ing/article/details/126967720 下载并改用.

4 代码

```
#include <iostream>
#include <queue>    //顾客队列
#include <vector>   //数组
#include <fstream>  //读文件
#include <ctime>    //计时
// thread, mutex heads ref: https://blog.csdn.net/Flag_ing/article/details/126967720
#include "mingw.thread.h"
#include "mingw.mutex.h"
using namespace std;

#define ALL_SERVERS 1
#define TIME_UNIT 1000
int ALL_CUSTOMERS;
typedef int semaphore;
typedef mutex mutex;

struct customer{
    public:
        int id;         // ifstream
        int t_entry;    // ifstream
        int t_need;     // ifstream
        int t_begin;    // set by server
        int t_leave;    // set by server
        int server;     // set by server
        void output(){
            cout << "顾客" << id << "\t进入时间" << t_entry << "\t服务时间" << t_need << "\t开始时间"
                << t_begin << "\t离开时间" << t_leave << "\t服务员" << server << endl;
        }
};

// 队列数据结构
queue<customer*> cqueue;

// 互斥
mutex mtx_cqueue; // 顾客队列结构互斥
// 同步，对应队列数据结构长度
semaphore cavail = 0; // 初始没有顾客（从0时刻开放到达）
// 时间
clock_t T_beg;
// 统计服务人数，用于终止线程
int served = 0;

void Server(int server_id) { // 柜员线程任务分配：等待顾客、叫号、服务顾客、结束顾客
    while (true){ // 柜员常驻
        if (served == ALL_CUSTOMERS) break; // 所有顾客服务完毕，退出
        mtx_cqueue.lock(); // P(mtx)
        if (cavail <= 0){ // 当<=0时，信号量无信号，表示没有顾客可用，柜员处于等待状态
            mtx_cqueue.unlock(); // V(mtx)
```

```
            continue;
        } else { // >0时进入服务状态
            customer* to_serve = cqueue.front();
            int duration = double(clock() - T_beg);
            to_serve->t_begin = duration / TIME_UNIT; // 记录时间
            // cout << "柜员" << server_id << "叫号顾客" << to_serve->id << endl;
            cqueue.pop();
            -- cavail;
            mtx_cqueue.unlock(); // V(mtx), 涉及改变cqueue的动作到此为止, 即可解锁

            Sleep(TIME_UNIT*to_serve->t_need); // 服务时间
            ++ served;
            // cout << "柜员" << server_id << "已服务顾客" << to_serve->id << endl;

            to_serve->t_leave = to_serve->t_begin + to_serve->t_need;
            to_serve->server = server_id;
        }
    }
}

void Customer(customer* customer) { // 顾客线程任务分配: 入场取号、入列等待
    // 顾客入场并取号
    Sleep(TIME_UNIT*customer->t_entry);
    // cout << "顾客(取号" << customer->id << ")到达" << endl;

    // 顾客入列等待
    mtx_cqueue.lock();
    cqueue.push(customer);
    ++ cavail; // 顾客队列+1, 对柜员而言可用顾客+1
    mtx_cqueue.unlock();
}

int count_lines(ifstream& file) { // 统计行数, 即顾客个数
    int cnt = 0;
    string line;
    while (getline(file, line)) {
        ++ cnt;
    }
    return cnt;
}


int main(){
    ifstream dataflow("src.txt");
    ALL_CUSTOMERS = count_lines(dataflow);
    dataflow.clear();
    dataflow.seekg(0, ios::beg); // 恢复指针到文件开头

    vector<thread> thread_servers(ALL_SERVERS);
    for (int i = 0; i < ALL_SERVERS; ++i) {
```

```
98          thread_servers[i] = thread(Server, i+1); // 创建柜员线程
99      }
100
101     vector<thread> thread_customers(ALL_CUSTOMERS);
102     vector<customer> customers(ALL_CUSTOMERS);
103
104     T_beg = clock();
105     for (int i = 0; i < ALL_CUSTOMERS; ++i) {
106         dataflow >> customers[i].id >> customers[i].t_entry >> customers[i].t_need;
107         // cout << customers[i].id << customers[i].t_entry << customers[i].t_need << endl;
108         thread_customers[i] = thread(Customer, &customers[i]); // 创建顾客线程
109     }
110     dataflow.close();
111
112     /* 在主线程环境下调用join()函数,
113     主线程要等待所有线程工作做完,
114     否则主线程将一直处于block状态 */
115     for (int i = 0; i < ALL_SERVERS; ++i) { // 收拾残局
116         thread_servers[i].join();
117     }
118     for (int i = 0; i < ALL_CUSTOMERS; ++i) { // 收拾残局
119         thread_customers[i].join();
120     }
121
122     // 输出
123     for (customer c : customers){
124         c.output();
125     }
126     return 0;
127 }
```

5 简单验证

根据指导书提供样例，分别令 ALL_SERVERS=1,2,4，分别得到

```
1  顾客1   进入时间1       服务时间10      开始时间1       离开时间11      服务员1
2  顾客2   进入时间5       服务时间2       开始时间11      离开时间13      服务员1
3  顾客3   进入时间6       服务时间3       开始时间13      离开时间16      服务员1
4
5  顾客1   进入时间1       服务时间10      开始时间1       离开时间11      服务员2
6  顾客2   进入时间5       服务时间2       开始时间5       离开时间7       服务员1
7  顾客3   进入时间6       服务时间3       开始时间7       离开时间10      服务员1
8
9  顾客1   进入时间1       服务时间10      开始时间1       离开时间11      服务员2
10 顾客2   进入时间5       服务时间2       开始时间5       离开时间7       服务员1
11 顾客3   进入时间6       服务时间3       开始时间6       离开时间9       服务员3
```

改变样例（参见输出前 3 列），令 ALL_SERVERS=4,8，分别得到

```
顾客1  进入时间1  服务时间10  开始时间1  离开时间11  服务员2
顾客2  进入时间5  服务时间2  开始时间5  离开时间7  服务员1
顾客3  进入时间6  服务时间10  开始时间6  离开时间16  服务员4
顾客4  进入时间7  服务时间10  开始时间7  离开时间17  服务员1
顾客5  进入时间7  服务时间9  开始时间7  离开时间16  服务员3
顾客6  进入时间8  服务时间8  开始时间11  离开时间19  服务员2
顾客7  进入时间9  服务时间4  开始时间16  离开时间20  服务员3

顾客1  进入时间1  服务时间10  开始时间1  离开时间11  服务员2
顾客2  进入时间5  服务时间2  开始时间5  离开时间7  服务员1
顾客3  进入时间6  服务时间10  开始时间6  离开时间16  服务员7
顾客4  进入时间7  服务时间10  开始时间7  离开时间17  服务员5
顾客5  进入时间7  服务时间9  开始时间7  离开时间16  服务员8
顾客6  进入时间8  服务时间8  开始时间8  离开时间16  服务员6
顾客7  进入时间9  服务时间4  开始时间9  离开时间13  服务员4
```

简单计算易知程序表现正确.

6 思考题

6.1 柜员人数和顾客人数对结果分别有什么影响？

根据“简单验证”一节可知，柜员人数越多，相对柜员人数较少的情况，完成全部服务的时间一般更短，这与常识相符，因为这样有利于减少顾客到达后因无人服务而需要等待的时间（或避免等待），但是柜员闲置率可能上升；顾客人数越多，则完成全部服务的时间一般更长，这主要是顾客到达时间多样、柜员忙而等待时间增加造成的，这也与常识相符.

6.2 实现互斥的方法有哪些？各自有什么特点？效率如何？

根据课程讲义，可以总结如下互斥实现方法：

互斥算法	特点	效率
锁变量	共享布尔锁变量 lock， 0：无进程在临界区 (初值) 1：有进程在临界区	忙等待浪费 CPU 时间， 等待时间非常短才使用
轮转法	整型变量 turn 用于记录轮到哪个进程进入临界区， 要求进程严格轮流进入临界区	忙等待浪费 CPU 时间， 等待时间非常短才使用
Peterson 算法	解决了 2 进程互斥访问的问题， 而且克服了轮转法的缺点，可以正常地工作	忙等待浪费 CPU 时间， 等待时间非常短才使用
硬件禁止中断	只有在发生时钟中断或其他中断时才会进行进程切换	把禁止中断的权利交给用户进程 ，系统可靠性较差；不适用于多处理器
硬件指令方法	利用处理机提供的专门的硬件指令， 对一个字的内容进行检测和修改	忙等待浪费 CPU 时间， 等待时间非常短才使用
信号量	引入高级管理者， 即使用资源/互斥信号量和 P,V 原语控制资源访问	避免忙等待，节约 CPU 资源 （但正确性分析很困难）
管程	把信号量及其操作原语封装在一个对象内部	避免忙等待，节约 CPU 资源