操作系统实验一

银行柜员服务问题

董凯杰

无 34

2013010572

2015 年 12 月 4 日

目录

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、 实验题目

银行柜员服务问题

三、 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。

四、 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。

五、 实验设计

主要数据结构

```
const int maxCustomerNo = 20;//假设所有顾客数的上限
const int ServerNo = 2;//柜员数量
const int Interval = 500;//为写程序方便，对顾客到来时间统一乘以一个系数

vector<int> LeaveList;//顾客离开的列表，当所有顾客离开时，程序终止
vector<int> WaitingList;//顾客等待的列表，便于柜员依次叫号
int come_time[maxCustomerNo];//顾客到来的时间
int wait_time[maxCustomerNo];//顾客需要被服务的时间
int serve_time[maxCustomerNo];//顾客开始被服务的时间
int leave_time[maxCustomerNo];//顾客离开的时间
int server_id[maxCustomerNo];//顾客被服务的柜员编号

HANDLE Server_mutex, Customer_mutex;//柜员互斥锁，顾客互斥锁
HANDLE Server_Semaphore, Customer_Semaphore;//柜员信号量，顾客信号量
HANDLE disp;//用于控制屏幕打印的互斥锁

//Server线程的参数
struct ServerParameter
{
        int ID;
};

//Customer线程的参数
struct CustomerParameter{
        int ID;
        int come_time;
        int wait_time;
};
```

程序框架

```
DWORD WINAPI Server(LPVOID lpPara)//柜员线程
{
        ServerParameter *SP = (ServerParameter *)lpPara;
```

```
        //柜员时刻准备着服务顾客
        while (true)
        {
            WaitForSingleObject(Customer_Semaphore, INFINITE); //申请一个
                顾客资源，即等待顾客到来
            WaitForSingleObject(Server_mutex, INFINITE);
            int ID = WaitingList[0];          //获取第一个等待的顾客编号
            serve_time[ID] = int(clock());   //当前时间即为开始服务时间
            server_id[ID] = SP->ID; //记录服务的柜员编号
            WaitingList.erase(WaitingList.begin()); //从等待列表中删除该顾客

            WaitForSingleObject(disp, INFINITE);
            out <<"Customer "<< ID << " is being servered by Server " << SP
                ->ID<<'.'<< endl;
            cout <<"Customer "<< ID << " is being servered by Server " << SP
                ->ID<<'.'<< endl;
            ReleaseMutex(disp);
            ReleaseMutex(Server_mutex);

            Sleep(wait_time[ID]);   //模拟顾客正在被服务

            WaitForSingleObject(Server_mutex, INFINITE);
            WaitForSingleObject(disp, INFINITE);
            out <<"Customer "<< ID << " is serverd." << endl;
            cout <<"Customer "<< ID << " is serverd." << endl;
            ReleaseMutex(disp);

            leave_time[ID] = int(clock());   //当前时间即为顾客离开时间
            LeaveList.erase(LeaveList.begin());      //从离开列表中删除该顾客
            ReleaseMutex(Server_mutex);
            ReleaseSemaphore(Server_Semaphore, 1, NULL); //释放一个可用服务
                资源
        }
        return 0;
    }
    DWORD WINAPI Customer(LPVOID lpPara)//顾客线程
```

```
{
        CustomerParameter *CP = (CustomerParameter *)lpPara;
        //顾客到来
        WaitForSingleObject(disp, INFINITE);
        out << "Customer " << CP->ID << " is coming." << endl;
        cout << "Customer " << CP->ID << " is coming." << endl;
        ReleaseMutex(disp);

        WaitingList.push_back(CP->ID); //在等待列表末尾添加该顾客编号
        LeaveList.push_back(CP->ID); //在离开列表末尾添加该顾客编号
        int come_clock = int(clock()); //记录顾客到来时间

        WaitForSingleObject(Server_Semaphore, INFINITE);    //申请一个服务资
            源，即等待有柜员空闲
        WaitForSingleObject(Customer_mutex, INFINITE);
        int wait_clock = floor((int(clock()) - come_clock) / Interval); //记录顾客等
            待时间
        WaitForSingleObject(disp, INFINITE);
        out << "After " << wait_clock << " seconds, " << "Customer " << CP->
            ID << " is called." << endl;
        cout << "After " << wait_clock << " seconds, " << "Customer " << CP->
            ID << " is called." << endl;
        ReleaseMutex(disp);
        ReleaseMutex(Customer_mutex);
        ReleaseSemaphore(Customer_Semaphore, 1, NULL); //唤醒一个空闲的柜员线
            程
        return 0;
}
int main()//主线程
{
        int init_time = int(clock());
        //柜员线程与顾客线程的线程ID
        DWORD ServerID[ServerNo];
        DWORD CustomerID[maxCustomerNo];
        //线程句柄
        HANDLE ServerHandle[ServerNo];
```

```
        HANDLE CustomerHandle[maxCustomerNo];
        //创建打印互斥量
        disp = CreateMutex(NULL, FALSE, NULL);
        //创建信号量及互斥量
        Server_mutex = CreateMutex(NULL, FALSE, NULL);
        Customer_mutex = CreateMutex(NULL, FALSE, NULL);
        Server_Semaphore = CreateSemaphore (NULL, ServerNo, ServerNo, NULL);
    Customer_Semaphore = CreateSemaphore (NULL, 0, ServerNo, NULL);
        //创建柜员线程
        ServerParameter SP[ServerNo];
        for (int i = 0; i < ServerNo; i++)
        {
                SP[i].ID = i;
                ServerHandle[i] = CreateThread(NULL, 0, (
                    LPTHREAD_START_ROUTINE)Server, &SP[i], 0, &ServerID[i])
                    ;
        if (ServerHandle[i]==NULL) return -1;
        }

        //读入顾客到来信息，并创建顾客线程
        int n;
        ifstream file ("Customer.txt");
        double last_time = 0;
        CustomerParameter CP[maxCustomerNo];
        while (!file.eof())
        {
                file >> n;
                file >> come_time[n] >> wait_time[n];
                come_time[n] = come_time[n] * Interval;
                wait_time[n] = wait_time[n] * Interval;
                CP[n].ID = n;
                CP[n].come_time = come_time[n];
                CP[n].wait_time = wait_time[n];
                int t = (come_time[n] - last_time);
                if (t < 0) t = 0;       //如果顾客同时到来，t可能为负值
                Sleep(t);       //模拟下一个顾客到来的时间
```

```
            //创建顾客线程
            CustomerHandle[n] = CreateThread(NULL, 0, (
                LPTHREAD_START_ROUTINE)Customer, &CP[n], 0, &
                CustomerID[n]);
            last_time = int(clock()) - init_time;
        }
        //等待直到顾客线程执行完成
        for (int i = 1; i <= n; i++)
            if (CustomerHandle[i] != NULL)
            {
                WaitForSingleObject(CustomerHandle[i], INFINITE);
                CloseHandle(CustomerHandle[i]);
            }

        //等待直到柜员线程执行完成
        while (LeaveList.size() != 0)
        {
            if (LeaveList.size() == 0)
                for (int i = 0; i < ServerNo; i++)
                    CloseHandle(ServerHandle);
        }
        //输出结果
        for (int i = 1; i <= n; i++)
        {
            come_time[i] = floor(come_time[i] / Interval);
            serve_time[i] = floor(serve_time[i] / Interval);
            wait_time[i] = floor(wait_time[i] / Interval);
            leave_time[i] = floor(leave_time[i] / Interval);
            out<< i <<" & "<< come_time[i]<<" & "<<serve_time[i]<<" & "
                <<wait_time[i]<<" & "<<leave_time[i]<<" & "<<server_id[i]
                << endl;
            cout<< i <<' '<< come_time[i]<<' '<<serve_time[i]<<' '<<
                wait_time[i]<<' '<<leave_time[i]<<' '<<server_id[i] << endl;
        }
    }
```

说明

Server_Semaphore（柜员信号量）表示当前有几个柜员是空闲的，每当一个顾客到来，就申请一个柜员信号量（信号量 −1）；每当一个顾客被服务完，就释放一个柜员信号量（信号量 +1）；当柜员信号量为 0 时，顾客需等待下一个空闲的柜员。

Customer_Semaphore（顾客信号量）表示当前有几个等待的顾客，每当一个顾客到来，就释放一个顾客信号量（信号量 +1）；每当一个顾客被服务完，就申请一个顾客信号量（信号量 −1）；当顾客信号量为 0 时，柜员需等待下一个到来的顾客；

Server_mutex（柜员互斥锁）保证了一个顾客只会被一个柜员所服务。

Customer_mutex（顾客互斥锁）保证了一个顾客只会拿一个号。

六、 实验结果

10 个顾客，2 个柜员：

顾客编号	到来时间	开始服务时间	服务时间	离开时间	服务者编号
1	1	1	10	11	0
2	3	3	9	12	1
3	5	11	2	13	0
4	8	12	10	22	1
5	9	13	1	14	0
6	13	14	5	19	0
7	18	19	7	26	0
8	20	22	2	24	1
9	20	24	3	27	1
10	20	26	4	30	0

10 个顾客，5 个柜员：

顾客编号	到来时间	开始服务时间	服务时间	离开时间	服务者编号
1	1	1	10	11	2
2	3	3	9	12	3
3	5	5	2	7	0
4	8	8	10	18	1
5	9	9	1	10	4
6	13	13	5	18	0
7	18	18	7	25	4
8	20	20	2	22	2
9	20	20	3	23	3
10	20	20	4	24	0

由以上结果可以看出，确实实现了顾客和服务员的等待关系。当顾客多服务员少的时候，顾客等待了服务员；当服务员多顾客少的时候，顾客到来即开始被服务。

七、 思考题

1. 柜员人数和顾客人数对结果分别有什么影响?
   答：当顾客数目比较多，顾客会等待；柜员人数比较多，会有服务员空闲，顾客到来立即就会被服务；具体可见上述结果。

2. 实现互斥的方法有哪些? 各自有什么特点? 效率如何?
   答：实现互斥的方法有临界区、互斥量、信号量。

   临界区：保证在某一时刻只有一个线程能访问数据的简便办法。在任意时刻只允许一个线程对共享资源进行访问。如果有多个线程试图同时访问临界区，那么在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占。只能用来同步本进程内的线程，而不可用来同步多个进程中的线程。临界区的实现最简单，效率最高，但是可能发生死锁。

   互斥量：互斥量跟临界区很相似，只有拥有互斥对象的线程才具有访问资源的权限，由于互斥对象只有一个，因此就决定了任何情况下此共享资源都不会同时被多个线程所访问。当前占据资源的线程在任务处理完后应将拥有的互斥对象交出，以便其

他线程在获得后得以访问资源。互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。因为互斥量的实现比较复杂，需要系统调用，所以效率相对比较低，也有可能发生死锁。

信号量：与临界区和互斥量不同，可以实现多个线程同时访问公共区域数据，原理与操作系统中 PV 操作类似，先设置一个访问公共区域的线程最大连接数，每有一个线程访问共享区资源数就减 1，线程在处理完共享资源后，应在离开的同时将当前可用资源计数加 1。但是当前可用计数减小到 0 时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入。信号量的效率往往不如互斥量高。

八、 实验总结

银行柜员服务问题本质上和生产者-消费者问题类似，顾客就相当于生产者，柜员就相当于消费者。每到来一个顾客，就相当于生产了一个产品，需要被消费掉。如果此时有柜员空闲，那就立即“消费”——服务，否则顾客就在等待列表中等待。唯一的区别就在于，生产者-消费者问题中，我们以固定的时间生产物品、消费物品，而在银行柜员服务问题中，顾客的到来时间由输入给定，同时“消费”掉这个“顾客”也需要一定的时间（程序中就用 sleep 实现）。

程序实现方面，一开始也遇到了比较大的困难，不太能理解那些跟进程线程相关的 api 以及它们的执行方式、顺序等。后来上网查阅了一些资料，也和马老师讨论过一次，对于多线程之间同步、互斥的问题有了更深入的了解和体会。