《操作系统》课程实验一

实验报告

进程间同步/互斥问题

<银行柜员服务问题>

班级：无 47

姓名：刘 前[1]

学号：2014011216

日期：2016 年 10 月 30 日

操作平台：Windows 8.1

编程语言：C++

[1] 清华大学电子工程系(E-mail: liuqian14@mails.tsinghua.edu.cn)

进程间同步/互斥问题

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。

实验题目：

本实验共有 5 个实验题目，任选其中之一。不同实验的难度不同，基准分也不同，请同学根据自己的情况加以选择。

实验题目	基准分
读者—写者问题	85
生产者—消费者问题	85
哲学家进餐问题	90
睡眠理发师问题	95
银行柜员服务问题	100

操作系统平台可选 Windows 或 Linux，编程语言不限。

实验报告内容要求：

1. 写出设计思路和程序结构，并对主要代码进行分析；
2. 实际程序运行情况；
3. 对提出的问题进行解答；
4. 体会或者是遇到的问题。

注：本人在实验一中选择的是**银行柜员服务问题**。

银行柜员服务问题

一、 问题描述及实验要求

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

a) 某个号码只能由一名顾客取得；不能有多于一个柜员叫同一个号；
b) 有顾客的时候，柜员才叫号；
c) 无柜员空闲的时候，顾客需要等待；
d) 无顾客的时候，柜员需要等待。

3. 实现提示

a) 互斥对象：顾客拿号，柜员叫号；
b) 同步对象：顾客和柜员；
c) 等待同步对象的队列：等待的顾客，等待的柜员；
d) 所有数据结构在访问时也需要互斥。

4. 测试文本格式

测试文件由若干记录组成，记录的字段用空格分开。记录第一个字段是顾客序号，第二字段为顾客进入银行的时间，第三字段是顾客需要服务的时间。

下面是一个测试数据文件的例子：

1 1 10
2 5 2
3 6 3

5. 输出要求

对于每个顾客需输出进入银行的时间、开始服务的时间、离开银行的时间和服务柜员号。

6. 思考题

a) 柜员人数和顾客人数对结果分别有什么影响？
b) 实现互斥的方法有哪些？各自有什么特点？效率如何？

二、 设计思路

根据问题要求，可以从顾客和银行柜员两个角度分析这一问题。

a）从顾客的角度：

顾客到银行接受服务的过程可以分为以下三个主要阶段：

1. 到达银行后取号；
2. 等待银行柜员叫号；
3. 到柜员接受服务；
4. 服务结束后离开银行。

以下对每一阶段作简要分析：

1. 取号相当于让顾客进入一个队列，队列顺序按照取号的先后(即取号从小到大) 排列；
2. 顾客等待银行柜员叫号时，每当某一柜员空闲时，根据队列的 FIFO 原则，当前队列队首的顾客会被叫到；
3. 到柜员接收服务，只要保证顾客所需服务的时间即可；
4. 顾客在服务结束后认为立即离开银行。

需要注意的是：顾客取号是互斥的，一个顾客只能获得一个号且一个号只能被一个顾客获得；顾客等待柜员叫号的过程是需要同步的。因而需要使用进程的同步与互斥来实现。

b）从柜员的角度

柜员的工作分为叫号和为顾客服务。柜员之间彼此互斥，即不能有多于同一柜员叫两个号，没有顾客时柜员也需要等待，因而柜员也需要同步和互斥实现等待和叫号的过程。

根据实验要求，使用 P，V 操作实现顾客和柜员的同步。

三、 程序结构

1. 基本数据结构及函数：

Customer	结构体数组
WaitingQueue	等待队列
Mutex_Counter	柜员叫号互斥信号量
Mutex_Customer	顾客拿号互斥信号量
Sema_Customer	等待队列中顾客信号量
CustomerIntoQueue	顾客到达银行后拿号入队列的函数
CounterCallCustomer	柜员叫号及顾客接受服务

2. 实现方法：

将上述分析用 C++程序加以实现，以下是准备工作：

1. 每个顾客和柜员都视作一个线程，创建这些线程；
2. 等待队列 WaitingQueue 使用 C++中 STL 的单向队列 queue 来实现
3. 等待队列中的顾客，使用一个信号量来表征，信号量的整数大小表示等待服务的顾客数；
4. 互斥信号量 Mutex_Customer 实现在取号时顾客之间的互斥；
5. 互斥信号量 Mutex_Counter 实现在叫号时柜员之间的互斥；
6. 每个顾客接受完服务建立一个事件信号 Service_Over，用于提示顾客服务完毕。

3. 对程序结构的解释：

对于顾客，创建线程后，先使用 C++库函数中 windows.h 的 Sleep 函数让顾客到相应的时间“进入银行”，取号时，为了避免两个顾客取到同一个号，使用互斥信号量 Mutex_Customer 实现互斥。取号之后进入等待队列 WaitingQueue，并且对顾客的信号量进行 V 操作，表示队列中增加了一位等待的顾客。

对于柜员，创建线程之后，判断等待队列不空时，并且根据互斥信号量 Mutex_Counter 判断此时没有其他柜员线程在叫号，就可以叫号，这一过程是对顾客进行 P 操作，即申请一个顾客资源，将顾客的信号量减一。之后顾客开始接受服务，同样使用 Sleep 函数实现。

当柜员服务结束时，为了提示顾客服务完毕，即线程结束，需另外设置一个事件对象，当顾客离开银行时发出信号。一直按上述过程操作，直到等待队列中等待的顾客全部服务完毕，可以使用 WaitForMultipleObjects 来实现。最后将顾客的相关信息输出到文件。

四、 代码分析

1. 顾客信息的结构体 Customer

为了便于存储顾客的所有相关信息，建立了结构体类型变量 Customer，数据成员包括：

顾客编号；
进入银行的时间；
开始服务的时间；
离开银行（服务结束）的时间；
服务的柜员号。

使用 Customer 结构体数组存储所有顾客信息，同时在顾客取号入队时，是将顾客的所有信息入队，因而等待队列 WaitingQueue 是队列的成员都是 Customer 结构体变量。

```
struct Customer                 //Customer 结构体，存储顾客的一系列信息
{
    int ID;                     //顾客序号
    int time_arrive;            //顾客进入银行的时间
    int time_begin;             //顾客开始服务的时间
    int time_need;              //顾客服务所需的时间
    int time_over;              //顾客服务结束离开银行的时间
    int server_counter;         //顾客接受服务的柜台号
};
```

2. 文件输入

文件的输入输出使用 C++库函数 fstream 实现。

输出文件：input.txt

输出文件：output.txt

因为需要建立顾客的结构体数组，所以需要从文件中获取顾客的总数。先编写 Get_Cust_Num( )函数，其中核心思路是使用 input.getline( )函数读取文件的每一行，从而得到行数，即为顾客数目。

```
int Get_Cust_Num()      //从文件中获取顾客的数目
{
    fstream input;                      //新建文件流
    input.open("input.txt",ios::in);    //打开"input.txt"文件，准备读入数据
    if(input == 0)          //input == 0 时，文件打开失败，提示并退出
    {
        cout<<"Sorry, cannot open the file!"<<endl;
        exit(0);
    }
    int cust_num = 0;                   //顾客数目初始化为 0
    char str[100];          //设置每行的字符数最大值，以此判断文件的行数
    while(!input.eof())  //文件读完之前
    {
        input.getline(str,sizeof(str));  //每读入一行,将顾客数目增加 1
        cust_num++;
    }
    input.close();          //关闭 input 文件流
    return cust_num;    //返回值为顾客的数量
}
```

获取顾客数目之后，即可创建 Customer 结构体数组：

Customer *cust = new Customer[cust_num];

但是此时结构体数组中尚未存储任何信息，因而还需要从 input.txt 文件中读入顾客信息，存入结构体数组。编写函数 Input( )进行实现。

```
void Input(Customer *cust)          //将文件中的信息存入 Customer 结构体数组
{
    fstream input;                  //新建文件流
    input.open("input.txt",ios::in);      //打开"input.txt"文件，准备读入数据
    if(input == 0)                  //input == 0 时，文件打开失败，提示并退出
    {
        cout<<"Sorry,cannot open the file!"<<endl;
        exit(0);
    }
    int i = 0;                      //结构体 index 初始化
    while(!input.eof())             //逐行读入顾客到银行服务的相关信息
    {
        //依次为顾客序号，到达银行的时间及需要服务的时间
        input>>cust[i].ID>>cust[i].time_arrive>>cust[i].time_need;
        i++;
    }
    input.close();                  //关闭文件流
}
```

3. 创建各类内核对象

HANDLE Sema_Customer;　//创建同步信号量的句柄对象
HANDLE Mutex_Customer = CreateMutex(NULL,FALSE,NULL);　//互斥信号量
HANDLE Mutex_Counter = CreateMutex(NULL,FALSE,NULL);
HANDLE *Service_Over;
//创建句柄对象，用于标志顾客接受服务是否完毕，之前已对各内核对象作解释。

4. 建立并运行各线程

```
HANDLE *customer_thread = new HANDLE[cust_num];    //每个顾客也作为一个线程
HANDLE *counter_thread = new HANDLE[counter_num]; //每个柜台作为一个线程
```

基本思路是：为每个顾客创建一个线程，其中 CreateThread 第三个参数是第 i 个顾客拿号后进入等待队列的函数 CustomerIntoQueue( )；同时为每个柜台创建一个线程，CreateThread 第三个参数是柜台叫号的函数 CounterCallCustomer( )。

每个顾客的线程运行结束之后，会发送出一个事件信号，从而帮助程序判断所有顾客全部服务完毕。

```
for (int i = 0;i<cust_num;i++)
{
    customer_thread[i]=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)
(CustomerIntoQueue),&(cust[i]),0,NULL);
    Service_Over[i]=CreateEvent(NULL,FALSE,FALSE,NULL);
}
for(int i = 0;i < counter_num;i++)
{
    counter_thread[i] = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)
(CounterCallCustomer),&(counter_ID[i]),0,NULL);
}
```

5. CustomerIntoQueue( )函数

该函数用于描述顾客拿号并进入等待队列的过程。主要思路是：任何任何一个顾客到达银行之后，都需要先拿号，拿号也可以视作是进入等待队列。顾客拿号彼此之间是互斥的，因而需要设置互斥信号量 Mutex_Customer，当一个顾客在拿号时，互斥信号量被占用，表明有顾客在拿号；当一个顾客拿号完毕，并将信息加入等待队列后，释放互斥信号量，解除占用，另一顾客可以开始拿号。

```
void CustomerIntoQueue(Customer *cust)      //顾客进入等待队列
{
    Customer* cust_in = cust;               //来到银行的某一顾客
    Sleep(time_cycle*(cust_in->time_arrive));
    //调整时间到该顾客应该到达的时刻
    //若在指定的时间内 mutex 互斥信号量被触发，函数返回 WAIT_OBJECT_0，表
    明此时新的顾客进入等待队列
    if(WaitForSingleObject(mutex,INFINITE) == WAIT_OBJECT_0)
    {
        WaitingQueue.push(cust_in);    //顾客的数据信息进入等待队列
        ReleaseSemaphore(Sema,1,NULL);   //V 操作释放信号量，等待顾客数增加 1
        ReleaseMutex(mutex); //V 操作，互斥信号量被释放，表明该顾客进入队列
    }
}
```

6. CounterCallCustomer( )函数

该函数用于描述柜员叫号的过程。主要思路是：与顾客拿号一样，柜员叫号也是一个互斥的过程，使用互斥信号量 Mutex_Counter，当一个柜员叫号时（从队列中取出队首的顾客）时，互斥信号量保证另一个柜员线程不会叫到相同的顾客号。叫完号后，即可释放互斥信号量。每次被叫到的顾客是最先进入队列，即最先拿号的顾客。

除此之外，函数中还设置了一个事件信号，当顾客完成服务之后触发产生信号。与 WaitForMultipleObjects 函数相对应。

```
void CounterCallCustomer(int *counter_ID)     //表示柜台叫号
{
    time_t time_base,time_begin;
    //获取时间，依次为初始（基准）时间和开始接受服务的时间
    time(&time_base);                          //获取程序初始的时间
    int* counter_id = counter_ID;              //获取柜台编号
    while(1)
    {
        //如果 Sema 信号量和 mutex 互斥信号量同时被触发，即既有等待的顾客，
          同时柜台又有空闲，即可使用 P 操作，进行同步
        if(WaitForSingleObject(Sema,INFINITE)==WAIT_OBJECT_0 &&
WaitForSingleObject(mutex,INFINITE) == WAIT_OBJECT_0)
        {
            Customer* cust_out=WaitingQueue.front();
            //被叫到的顾客是最先进入队列，即最先拿号的顾客
            WaitingQueue.pop();        //将被叫号的顾客从队列中 pop 出
            ReleaseMutex(mutex);       //V 操作，等待队列需要释放互斥信号量
            cust_out->server_counter=*counter_id;     //为该顾客服务的柜台号
            time(&time_begin);         //获取被叫号（开始服务）的时间
            cust_out->time_begin =int(time_begin) - time_base;
           //开始服务的时间（相对于基准时间）
            Sleep(time_cycle*cust_out->time_need);   //Sleep 直到该顾客服务完成
            cust_out->time_over = cust_out->time_begin + cust_out->time_need;
           //顾客服务结束（离开银行）的时间
            SetEvent(Service_Over[cust_out->ID-1]);
           //此处设置了一个事件信号，当顾客完成服务之后触发产生信号
        }
    }
}
```

7. 等待各顾客线程运行完毕

使用函数 WaitForMultipleObjects( )，通过接收事件信号 Service_Over，判断各线程是否结束。

WaitForMultipleObjects(cust_num,Service_Over,TRUE,INFINITE);

8. 文件输出

在所有的操作完成之后，还需要将程序的运行的结构输出到 output.txt 文件。编写 Output 函数实现。只是基本的写文件操作，详见注释。

```
void Output(Customer *cust,int cust_num)  //将顾客在银行服务涉及的信息输出
{
    ofstream output;                      //输出文件流
    output.open("output.txt",ios::out);   //打开文件，准备写入数据
    for(int i=0;i<cust_num;i++)           //逐个写入各顾客的信息
    {
        //依次为：顾客序号，到达银行时间，开始接受服务时间，服务结束离开银行时间，以及接受服务的柜台号
        output<<cust[i].ID<<" "<<cust[i].time_arrive<<" "<<cust[i].time_begin<<" "
            <<cust[i].time_over<<" "<<cust[i].server_counter<<endl;
    }
    output.close();                       //关闭文件流
}
```

五、 程序运行结果[2]：

1. 运行结果 1：

测试文件 1：

input.txt	output.txt
1 1 10 2 5 2 3 6 3	1 1 1 11 2 2 5 5 7 1 3 6 7 10 1

[2] 程序中，柜员数目的设定需要#define counter_num N，N 的具体数值可视情况确定。

输出文件如表中左侧所示，设置柜员数目 counter_num 为 2 时，输出文件如表中右侧所示。

结果分析：

由于没有对柜员设置优先顺序，所以，当同时存在多个空闲柜员时，选某一个柜员不影响结果正确性。

顾客 1 时刻 1 进入银行，立即取号，柜员 2 空闲叫号，顾客 1 即刻开始服务；
时刻 5 顾客 2 进入银行并拿号，此时柜员 1 空闲，到柜员 1 处接受服务；
时刻 6 顾客 3 进入银行并拿号，但是所有柜员都在忙碌，因而需要等待；
时刻 7 顾客 2 在柜员 1 处接受服务完毕，此时顾客 3 开始接收服务；
时刻 10 顾客 3 在柜员 1 处接受服务完毕，此时柜员 1 空闲；
时刻 11 顾客 1 在柜员 2 出接受服务完毕，此时柜员 2 空闲。
因而，根据上述分析，输出文件所示的结果是正确的。

2. **测试文件 2：**

input.txt	output.txt
1 1 5	1 1 1 6 3
2 3 6	2 3 3 9 2
3 5 8	3 5 5 13 1
4 6 3	4 6 6 9 3
5 9 2	5 9 9 11 2
6 9 7	6 9 9 16 3
7 10 10	7 10 11 21 2
8 17 6	8 17 17 23 3
9 20 8	9 20 20 28 1
10 22 1	10 22 22 23 2

设置柜员数目 counter_num 为 3，容易验证输出文件的结果同样是正确的。

六、 思考题

1. 柜员人数和顾客人数对结果分别有什么影响？

解：

a）柜员人数对结果的影响：

不难猜测，当顾客数量及每个顾客的服务时间确定时，柜员数目越多，顾客等待的平均时间越短，在程序上体现为程序运行的时间越短，最终输出的结果中，每个顾客离开银行的时间也越早[3]。

不妨考虑极端情况，当柜员数等于甚至大于顾客数时，每个顾客到了银行立刻就能接受服务，每个顾客等待的时间为 0；另一个极端情况是，很多个顾客，只有一个柜员时，除了第一个顾客，每个顾客都有可能[4]等待，这就导致整个时间较长，运行时间也很长。

通过实验来验证：

假定有 10 个顾客，现将柜员数分别设为 1，2，3，比较三种情况下最后一个顾客离开银行的时间。

input.txt
1 1 5
2 3 6
3 5 8
4 6 3
5 9 2
6 9 7
7 10 10
8 17 6
9 20 8
10 22 1

记柜员数为 N，最后一个顾客离开银行时间为 T，

N = 1 时，T = 57；
N = 2 时，T = 27；
N = 3 时，T = 23；
N = 4 时，T = 23。

根据实验，随着柜员数目的增加，最后一个顾客离开银行的时间越短，最后等于直接接受服务情况下的离开时间。

b）顾客人数对结果的影响：

当柜员人数确定时，某一时段内顾客人数越多，每个顾客等待的平均时间越长，最后一个顾客离开银行的时间也应当越晚[5]。

[3] 这一假设基于最后一个顾客来得足够早，即前一个顾客的服务时间会影响最后一个顾客的等待时间。
[4] 对“可能”的解释：如果顾客比较稀少，一个柜员的情况，顾客也有可能不需要等待。
[5] 这一假设同样基于最后一个顾客来得足够早，即前一个顾客的服务时间会影响最后一个顾客的等待时间。

综合以上分析，可以大致归纳出下面的结论：

假设每位顾客所需的时间基本相同（不会有服务时间过长的异常情况），那么顾客数与柜员数的比值越大，顾客等待的时间可能越长，离开银行的时间越晚，这一点比较容易理解。

事实上，除了顾客的数目之外，顾客的疏密（客流量）程度也会对结果造成较大的影响。对于同样多数目的顾客，如果顾客到达的时间间隔越大，即使只有一个柜员，可能也不需要等待，立即就能接受服务。

2. 实现互斥的方法有哪些？各自有什么特点？效率如何？

解：

实现互斥的方法主要有以下几种：

a) 禁止中断；
b) 严格轮转法；
c) Peterso 算法；
d) 共享锁变量；
e) 互斥信号量；等。

a）禁止中断法：

基本原理：

由于 CPU 只有在发生时钟中断或者其他中断时才会进行，所以通过中断的控制实现互斥。进程在进入临界区之前执行“关中断”指令，中断被关闭之后，不会发生进程的切换，因而其他进程无法进入临界区，进程离开临界区之后执行开中断指令。

特点及效率：

禁止中断法简单易操作，但是禁止中断本应受到内核的控制，将禁止中断的权力交给用户进程将导致系统的可靠性变差。这一方法也不使用于多处理器。

b）严格轮转法：

基本原理：

设置一整型变量 turn，用于记录轮到哪个进程进入临界区。每个进程只有 turn 的值与进程序号一致时，才能进入临界区。

特点及效率：

严格轮转法要求两个进程严格轮流进入临界区，而且也会出现忙等待的问题，因而效率也比较低。

c) Peterson 算法：

基本原理：

当一个进程想要进入临界区时，需要先调用 enter_region 函数，判断是否能够安全进入，不能则等待。当进程从临界区退出后，调用 leave_rgion 函数，允许其他进程进入临界区。

特点及效率：

Peterson 算法解决了进程互斥访问临界区的问题，而且避免了严格轮转法“进程严格轮流进入临界区”的缺点，能够正常地工作，但是仍然克服不了忙等待的问题。

d）共享锁变量：

基本原理：

设置一个共享锁变量 lock，当有进程在临界区内时，lock＝1；无进程在临界区内时，lock＝0。对于一个进程，在进入临界区时，需要先判断 lock 是否为 0。如果为 1，则一直被阻塞，直到临界区内的进程出了临界区，lock 变为 0 后，该进程才能进入临界区，并同时将 lock 置为 1。

特点及效率：

该方法的突出特点是忙等待。进程在进入临界区之前，需要连续测试一个变量，直到该变量的值满足能进入临界区的条件。因而，非常浪费 CPU 时间，效率很低。

e）互斥信号量：

基本原理：

给临界资源设置一个互斥信号量 Mutex，互斥信号量为 0 时，表明临界资源被某一进程占用，其他进程无法获取临界资源；但是，当互斥信号量被释放后（ReleaseMutex），互斥信号量为 1，其他进程如果检测到这一变化，就能占用临界资源。

特点及效率：

这一方法实现进程的间的互斥是较为常用，而且效率最高的。本次实验所解决的银行柜员服务问题就是使用互斥信号量的方法实现了顾客拿号之间的互斥以及柜员叫号时的互斥。

七、 实验总结及体会

本次实验刚开始虽然明白问题的要求，但是完全不知道如何使用 C++来实现，主要是因为一开始对 Windows 的互斥和同步只有概念上的理解，但是对 Windows 中的互斥对象、信号量对象、事件对象以及各对象相关的 API 不了解，根本不知道如何使用。所以先通过查阅书籍和网上的相关技术博客，对 Windows 中的有关多线程的各对象和 API 进行了学习和了解。

学习完相关函数，准备开始解决这一问题。其实解决问题的思路比较简单，但是实现的方式有些复杂，通过 P、V 操作实现同步与互斥。另外，困扰我很长时间的问题是程序中时间的设置，因为程序运行时涉及到时间的相关操作，比如某一顾客接受服务时，需要模拟让该进程忙碌（挂起）一段时间，然后才能体现出真实的情况。本来想通过设置一个全局的计时变量，当作程序的时钟信号，但是后来发现 Windows 自带了与时间的相关函数，比如 Sleep 函数就能实现进程的挂起，这给编程提供了很大的便利。

当时还有一个问题是，没有设置事件对象来提示各顾客是否服务完毕，而是创建完进程，调用了两个关键的 CustomerIntoQueue 和 CounterCallCustomer 函数之后，直接往文件里输出，结果发现输出结果基本是错的。后来了解到 Windows 某些操作完成后，需要使用事件对象提示进程的完成，之前错误的结果是因为输出到文件时进程还尚未结束，因而输出的信息是错误的。使用了 CreateEvent 和 SetEvent 之后，问题得到了解决。

总之，这次实验让我加深了对进程间互斥及同步的理解，同时基本掌握了 Windows 中一些关键的函数，使用这些函数实现了 P、V 操作，对解决多线程问题积累了一定的经验!