# 操作系统课程试验
# 进程间同步/互斥问题——银行柜员服务问题

BobAnkh

June 2021

编程平台: Ubuntu 18.04.5 LTS (GNU/Linux 4.15.0-142-generic x86_64)
编程语言: C/C++, 采用了部分 C++11 特性

## 一. 实验题目

### 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. 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。

### 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。

## 二. 设计思路

考虑到需要存储的数据，将顾客和柜员分别采用结构体的形式进行存储，并且为了很好地模拟顾客与柜员的行为，所以为每一个顾客和每一个柜员分别安排一个线程，其中顾客线程再被服务完之后就结束，而柜员线程则会反复执行，直到全部顾客都被服务完毕后才会被取消。

因为顾客和柜员分别都需要等待对方，为了同步，所以分别定义了顾客信号量和柜员信号量。同时，因为顾客需要先后被叫号，所以需要有一个等待队列用于存放在排队的顾客。在本程序中，所有数据结构（包括等待队列以及输出流等等）都使用 pthread 的互斥锁来保证互斥访问。最为核心主要的顾客和柜员的函数伪码算法如下 1所示，整体结构图如下图 1所示。

**Algorithm 1:** 银行柜员问题伪码算法 (mutex 是互斥锁, customers 和 counters 是信号量, queue 是顾客等待队列)

```
1  Function Customer(customer c):
2      lock(&mutex)
3      queue.push(c)
4      up(&customers)
5      unlock(&mutex)
6      down(&counters)
7      served()
8      return 0
9
10 Function Counter():
11     while TRUE do
12         down(&customers)
13         lock(&mutex)
14         c = queue.front()
15         queue.pop()
16         unlock(&mutex)
17         up(&counters)
18         serve(c)
```

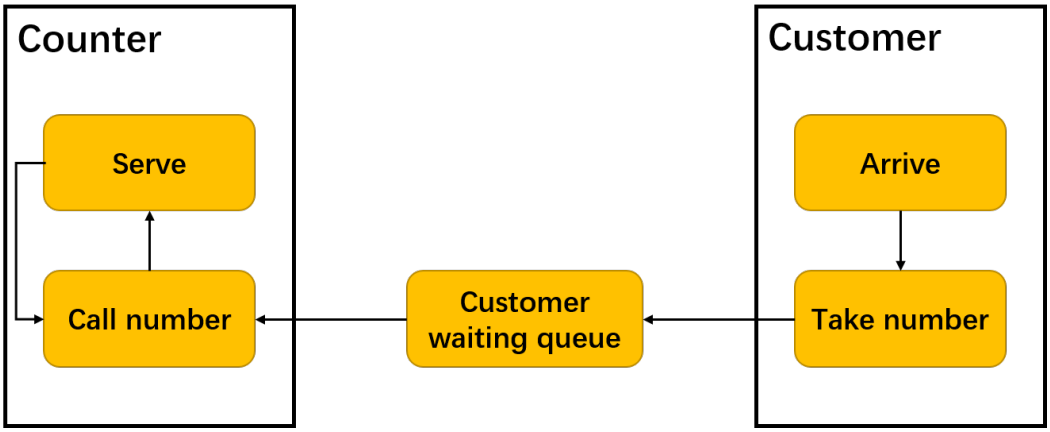


图 1: 整体结构图

# 三. 程序结构

本程序在具体实现上，主要采用了 pthread 以及 semaphore 两个库进行实现，仅在队列 queue 和向量 vector 上采用了 C++11 的相关特性，从而方便编程和数据结构的实现，同时这样也利于跨平台。

本程序对于每一个柜员和每一个顾客都建立一个线程加以模拟，对于每一个时刻采用 1s 进行模拟，用 sleep 函数模拟每一个顾客包括到达、接受服务等事件。柜员数量以 int 型常量 COUNTER_NUM 的方式给定。定义了两个结构体 Customer 和 Counter 分别用于存放顾客和柜员相关的数据，具体如下：

```
struct Customer
{
    pthread_t pID;          //线程 id
    int id;                 //客户序号
    int arrive_time;        //客户到达的时刻
    int service_duration;   //客户需要服务的时长
    int service_start;      //服务开始的时刻
    int counter_id;         //服务柜员号
    Customer(int cus_id, int a_time, int s_duration)
    {
        pID = 0;
        id = cus_id;
        arrive_time = a_time;
        service_duration = s_duration;
    }
};

struct Counter
{
    pthread_t pID; //线程 id
    int id;         //柜员序号
    Counter(int count_id)
    {
        pID = 0;
```

```
        id = count_id;
    }
};
```

此外，本程序中定义了一个顾客向量 `vector<Customer> customer_list` 和一个柜员向量 `vector<Counter> counter_list` 用于存放所有的顾客和柜员，在初始时读取输入文件并完成所有顾客结构体和柜员结构体的创建；同时还定义了一个队列 `queue<Customer *> customer_waiting` 用于存放等待中的顾客的指针。定义了互斥访问锁 `pthread_mutex_t mutex` 用于互斥访问所有的数据结构，包括顾客等待队列以及输出流等。定义了顾客信号量和柜员信号量 `sem_t customer_sem` 和 `sem_t counter_sem` 用以实现顾客和柜员的同步。

主程序主要是先读取输入文件并完成一些变量的初始化工作，然后产生顾客线程和柜员线程，随后等待所有顾客被服务完毕（即所有顾客线程运行结束），接着向所有柜员线程发出取消信号，结束所有的柜员线程。随后销毁互斥锁和信号量并将结果写入到输出文件中。

下面主要分析顾客线程和柜员线程各自所运行的函数从而说明和设计的一致性。其余完整源代码参见文件附件 `bankcounter.cpp`。这两个函数具体分析如下：

### 1. 顾客线程函数

这一部分代码具体如下：

```
// 顾客线程
void *customer_thread(void *arg)
{
    Customer &customer = *(Customer *)arg;
    // 模拟顾客到来
    sleep(customer.arrive_time);

    // 顾客取号(加入等待队列)
    pthread_mutex_lock(&mutex);
    cout << "[CUSTOMER] id " << customer.id << " arrive at " << time(NULL) -
        program_start_time << endl;
    customer_waiting.push(&customer);
    sem_post(&customer_sem);
    pthread_mutex_unlock(&mutex);

    // 顾客等待柜员空闲后接受服务
    sem_wait(&counter_sem);
    pthread_mutex_lock(&mutex);
    cout << "[CUSTOMER] id " << customer.id << " served by " << customer.
        counter_id << " at " << customer.service_start << ". Now time: " <<
        time(NULL) - program_start_time << endl;
    pthread_mutex_unlock(&mutex);
    // 模拟顾客服务过程
    sleep(customer.service_duration);
    return NULL;
}
```

该函数中，用 sleep 模拟顾客到来的过程，在取号加入等待队列前，先对互斥访问锁进行加锁，然后将指向该用户的指针放入顾客等待队列中，然后提升 `customer_sem` 这个信号量以使柜员知晓等待队列中出现了顾客，随后解锁互斥访问锁。然后获取 `counter_sem` 这个信号量

以实现等柜员空闲后接受服务，同样用 sleep 来模拟顾客接受服务的过程。在服务完成后该函数就运行完毕了。

## 2. 柜员线程函数

这一部分代码具体如下：

```
// 柜员线程
void *counter_thread(void *arg)
{
    Counter &counter = *(Counter *)arg;
    while (true)
    {
        // 柜员等待顾客
        sem_wait(&customer_sem);
        // 在柜员线程工作期间不能取消线程
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);

        // 柜员叫号(从等待队列中取出顾客)
        pthread_mutex_lock(&mutex);
        Customer &customer = *customer_waiting.front();
        customer.counter_id = counter.id;
        customer.service_start = time(NULL) - program_start_time;
        cout << "[COUNTER] id " << counter.id << " serve customer " <<
            customer.id << " at " << customer.service_start << ". Now time: "
             << time(NULL) - program_start_time << endl;
        customer_waiting.pop();
        sem_post(&counter_sem);
        pthread_mutex_unlock(&mutex);

        // 模拟柜员服务顾客
        sleep(customer.service_duration);
        pthread_mutex_lock(&mutex);
        cout << "[COUNTER] id " << counter.id << " finish serving customer "
            << customer.id << " at " << time(NULL) - program_start_time <<
            endl;
        pthread_mutex_unlock(&mutex);

        // 解除线程取消的限制
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
        pthread_testcancel();
    }
}
```

该函数中，先获取 `customer_sem` 这个信号量以实现柜员对于顾客的等待，获取之后，则进入工作状态，即令该线程不能够在此时被取消。然后对互斥访问锁进行加锁，从而访问顾客等待队列，取出第一个顾客并修改其对应信息，然后提升 `counter_sem` 这个信号量一实现通知顾客该柜员空闲，随后解锁互斥访问锁并用 sleep 来模拟柜员对顾客的服务过程。服务完毕后解除线程不能取消的限制，使得脱离工作状态的柜员线程是可以被取消的。该函数将在一个线程中反复运行直到被取消。

# 四. 运行结果

使用了 2 个柜员线程 (柜员号从 1 开始计数)，随机生成了 7 个顾客的输入文件测试样例在输入文件 `input.txt` 中，每行三个字段，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。本次测试运行使用的样例如下：

1 1 11
2 1 4
3 1 7
4 5 2
5 6 3
6 7 5
7 7 2

运行 `make` 或 `g++ bankcounter.cpp -o bankcounter.exe -std=c++11 -lpthread` 指令完成编译，运行 `./bankcounter.exe` 进行测试，运行时输出如下所示：

```
❯ ./bankcounter.exe
Read input file...
[CUSTOMER] id 1 arrive at 1
[CUSTOMER] id 2 arrive at 1
[COUNTER] id 1 serve customer 1 at 1. Now time: 1
[CUSTOMER] id 3 arrive at 1
[COUNTER] id 2 serve customer 2 at 1. Now time: 1
[CUSTOMER] id 1 served by 1 at 1. Now time: 1
[CUSTOMER] id 2 served by 2 at 1. Now time: 1
[CUSTOMER] id 4 arrive at 5
[COUNTER] id 2 finish serving customer 2 at 5
[COUNTER] id 2 serve customer 3 at 5. Now time: 5
[CUSTOMER] id 3 served by 2 at 5. Now time: 5
[CUSTOMER] id 5 arrive at 6
[CUSTOMER] id 6 arrive at 7
[CUSTOMER] id 7 arrive at 7
[COUNTER] id 1 finish serving customer 1 at 12
[COUNTER] id 1 serve customer 4 at 12. Now time: 12
[CUSTOMER] id 4 served by 1 at 12. Now time: 12
[COUNTER] id 2 finish serving customer 3 at 12
[COUNTER] id 2 serve customer 5 at 12. Now time: 12
[CUSTOMER] id 5 served by 2 at 12. Now time: 12
[COUNTER] id 1 finish serving customer 4 at 14
[COUNTER] id 1 serve customer 6 at 14. Now time: 14
[CUSTOMER] id 6 served by 1 at 14. Now time: 14
[COUNTER] id 2 finish serving customer 5 at 15
[COUNTER] id 2 serve customer 7 at 15. Now time: 15
[CUSTOMER] id 7 served by 2 at 15. Now time: 15
[COUNTER] id 2 finish serving customer 7 at 17
[COUNTER] id 1 finish serving customer 6 at 19
All thread finish!
Write output file...
Program finished!
```

图 2: 运行时输出情况

结果被输出到 `output.txt` 中，如下图所示，每行 5 个字段分别为每个顾客进入银行的时间、开始服务的时间、离开银行的时间和服务柜员号 (柜员号从 1 开始计数)：

```
❯ cat output.txt
1 1 1 12 1
2 1 1 5 2
3 1 5 12 2
4 5 12 14 1
5 6 12 15 2
6 7 14 19 1
7 7 15 17 2
```

图 3: output.txt 中结果输出情况

从结果可以看到，时序运行准确且事件发生（到达、叫号、完成服务）与理论分析相吻合，而且屏幕输出也与输出到文件中的结果一致。当然由于同一时间有多于柜员数的顾客同时到达，所以分配的柜员可能在不同此运行时有不同，这也体现了多线程运行情况的不可预期。具体分析可以看到，在 1 时刻，1,2,3 这三位顾客同时到来了，由于只有两个柜员，所以 1 号和 2 号柜员分别先叫了 1 和 2 这两位顾客，2 号柜员在服务完 2 这位顾客后叫了先等在队列中的 3 这位顾客。等到 12 时刻，1 和 3 这两位顾客同时被服务完毕，两个柜员均空闲，他们分别叫了队列中依次等待着的 4 和 5 这两位顾客，并且在各自顾客服务完毕后又分别叫了依次的 6 和 7 这两位顾客。由此可以验证程序可以正确运行，非常好地实现了设计和要求。

# 五. 思考题

## 1. 柜员人数和顾客人数对结果分别有什么影响？

顾客人数不变的情况下，柜员人数的增加，初始时会使得总需要时间下降，因为同时到来的顾客可以有更多的得到服务而不必等待，但是在柜员人数增加到一定程度之后，总时间的下降便不再明显甚至基本没有，因为这个时候柜员大部分时间是在闲置等待顾客。

柜员人数不变且饱和（饱和指柜员单位时间总服务速度小于顾客总到达速度）的情况下，顾客人数的增加会使得总需要时间上升，而且时间增长基本上与顾客人数增加成正比，这一点在假设顾客到来服从泊松分布的条件下也可以通过数学推导加以证明，但直观上也可以理解。

## 2. 实现互斥的方法有哪些? 各自有什么特点? 效率如何?

- 禁止中断：进入临界区前执行关中断指令，离开临界区后执行开中断指令。该方法实现起来简单但是将禁止中断的权利交给用户进程而导致系统可靠性差，且不适合多处理器。
- 轮转法：进程严格轮流进入临界区。但是该方法程序代码不对称，且可能忙等待，效率较低，且可能在临界区外阻塞别的进程 (可能违反空闲让进的原则)。
- Petersen 算法：可以正常工作的解决互斥问题的算法，仍使用锁。解决了互斥访问的问题，克服了轮转法的确定，但仍然会忙等待，效率较低。
- 硬件指令方法：利用处理器提供的专门的硬件指令，提供不被打断的以单条指令对共享变量进行的读写。适用于任意数目进程，且较简单，但仍有忙等待，耗费 CPU 时间。
- 信号量：使用初始化和两个标准的原语访问信号量管理资源。不必忙等待，效率较高，但信号量的控制分布在整个程序中，易读性差且正确性难以保证，同时也难于修改和维护。
- 管程：对于信号量及其操作原语的高级语言封装。效率同信号量一样，易于管理开发，不过作为编程语言概念需要编译器（编程语言）支持。

- 消息传递：用以实现分布式系统（不同机器间）的同步、互斥。支持分布式系统，但消息传递本身效率较低且不完全可靠。

# 六. 实验感受和总结收获

本次实验涉及 Linux 环境下多线程的编写，也是我第一次接触用 C++ 来编写多线程，而且感觉 C++ 的多线程也相对比较底层，对于这方面学到了不少实际使用方面的经验。同时，我处理了多线程之间的同步和互斥问题，对于相应的函数和操作有了更加清楚的认知。通过该次实验，我对于同步和互斥的设计和实现有了更加深入清晰的认知和理解，对于课程中教授的理论知识有了更加深刻的巩固和印证。本次实验对我来说收获颇丰，收获了 C++ 多线程程序编写方面相关细节的经验和同步互斥设计方面的经验。

# 七. 附录

随报告附主程序 `bankcounter.cpp`，本测试所用的输入文件 `input.txt` 和对应输出文件 `output.txt`，以及可以用来进行方便编译的 Makefile。