银行柜员服务问题

无 81 马啸阳 2018011054

2020 年 4 月 15 日

1 实验题目

1.1 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。

1.2 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。

1.3 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。

1.4 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

1.5 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。

2 设计思路

程序中为模拟柜员和顾客的行为，每个柜员和顾客分别安排一个线程。为同步二者，采用信号量，由于柜员和顾客分别需要等待，因此分别设置柜员信号量与顾客信号量。顾客还需要按号服务，因此要设置一个队列用以存放等待的顾客。号码本身并不是重要的，仅标志着出队的次序，可以使用一个变量每次加锁计数，这里由于输出中并未要求输出号码，因此不存储号码。队列本身以及其它数据（例如输出流）加互斥锁保证互斥。伪代码如算法 1所示（仅顾客、柜员线程部分伪代码，初始化信号量、互斥锁等忽略），使用互斥量 mutex 实现数据互斥，信号量 counters 和 customers 结合 P、V 原语实现同步，queue 用以存储等待顾客队列。结构图如图 1所示。

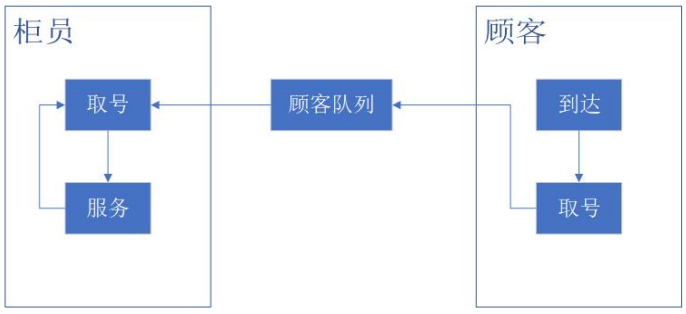


图 1: 银行柜员服务问题结构图

算法 1 银行柜员服务问题

```
1: function CUSTOMER(id)
2:     down(&mutex);
3:     queue.push(id);
4:     up(&customers);
5:     up(&mutex);
6:     down(&counters);
7:     get_served();
8: end function
9: function COUNTER(id)
10:    while TRUE do
11:        down(&customers);
12:        down(&mutex);
13:        customer_id = queue.top(), queue.pop();
14:        up(&mutex);
15:        up(&counters);
16:        serve_customer(customer_id);
17:    end while
18: end function
```

3 程序代码

本实验代码使用 C++ 结合 pthread 实现。实际上 C++11 及以上具有自身的标准线程库 std::thread 和互斥访问方法 std::mutex 等，C++20 中也有了 std::counting_semaphore 与 std::binary_semaphore 的信号量实现。但本实验代码中，本质采用 C 风格编程，使用 pthread 线程库，仅利用了 C++ 的极少特性（如 std::vector 和 std::queue）以简化数据结构的编写，本质可看作 C 风格程序，同时可跨平台。

程序中对每个柜员、顾客分别新建一个线程用以模拟，到达、服务事件使用 sleep 来模拟。柜员数量以常量在代码中给定（const int counter_num）。有一个队列 customer_queue 用以存储等待顾客队列。采用一个互斥量 lock 用以互斥所有数据结构，包括顾客队列以及对输出流 cout 的访问，因为对二者的访问通常是连续进行的，因而不区分使用多个互斥量。采用两个信号量 customer_sem 和 counter_sem 实现柜员和顾客的同步。最终主程序等待所有顾客被服务完毕后向柜台线程发送取消信号，终止所有柜台线程，并将结果输出文件（与程序运行中输出的一致，运行中输出的内容用以验证程序正确运行）。代码如下。

```cpp
#include <cstring>
#include <ctime>
#include <iostream>
#include <fstream>
#include <pthread.h>
#include <queue>
#include <semaphore.h>
#include <unistd.h>
#include <vector>
using namespace std;

#define _REENTRANT

const int counter_num = 2;

class Customer
{
public:
  pthread_t tid; // THREAD ID
  int id; // CUSTOMER ID
  int counter; // COUNTER ID
  int arrival; // ARRIVAL TIME
  int serve; // SERVE BEGIN TIME
  int wait; // WAIT TIME DURING SERVE
  Customer(int id, int arrival, int wait): tid(0), id(id), arrival(arrival), serve(0), wait(wait) {}
};

class Counter
{
public:
  pthread_t tid; // THREAD ID
  int id; // COUNTER ID
  Counter(): tid(0), id(0) {}
};

time_t begin_time;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int customer_num;
sem_t customer_sem;
sem_t counter_sem;
vector<Customer> customer_list;
vector<Counter> counter_list(counter_num);
queue<int> customer_queue; // CUSTOMERS WAITING

void* customer_thread(void* c)
{
```

```
47   Customer &customer = *(Customer*)c;
48   sleep(customer.arrival); // SIMULATE ARRIVAL TIME
49   pthread_mutex_lock(&lock);
50   cout << "customer arrive id " << customer.id << " time " << time(NULL) - begin_time << endl;
51   customer_queue.push(customer.id);
52   sem_post(&customer_sem);
53   pthread_mutex_unlock(&lock);
54
55   sem_wait(&counter_sem); // READY TO BE SERVED
56   pthread_mutex_lock(&lock);
57   cout << "customer serve (customer thread) id " << customer.id << " counter id " << customer.counter
         << " time " << time(NULL) - begin_time << endl;
58   pthread_mutex_unlock(&lock);
59   sleep(customer.wait); // SIMULATE SERVE
60   pthread_exit(NULL);
61   return NULL;
62 }
63
64 void* counter_thread(void* c)
65 {
66   Counter &counter = *(Counter*)c;
67   while (true)
68   {
69     sem_wait(&customer_sem);
70     pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL); // PREVENT CANCELLATION WHEN COUNTER WORKING
71     pthread_mutex_lock(&lock);
72     Customer& customer = customer_list[customer_queue.front() - 1];
73     customer_queue.pop();
74     customer.counter = counter.id;
75     customer.serve = time(NULL) - begin_time;
76     cout << "customer serve (counter thread) id " << customer.id << " counter id " << customer.counter
           << " time " << time(NULL) - begin_time << endl;
77     pthread_mutex_unlock(&lock);
78
79     sem_post(&counter_sem);
80     sleep(customer.wait); // SIMULATE SERVE
81     pthread_mutex_lock(&lock);
82     cout << "customer finish id " << customer.id << " time " << time(NULL) - begin_time << endl;
83     pthread_mutex_unlock(&lock);
84     pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
85     pthread_testcancel(); // MAYBE UNNECESSARY
86   }
87 }
88
89 void read_file()
90 {
91   ifstream input("input.txt", ios::in);
```

```
92    int id, arrival, wait;
93    while (true)
94    {
95      input >> id >> arrival >> wait;
96      if(input.fail())
97        break;
98      customer_list.push_back(Customer(id, arrival, wait));
99      ++customer_num;
100   }
101 }
102
103 int main()
104 {
105   pthread_mutex_init(&lock, NULL);
106   sem_init(&counter_sem, 0, 0);
107   sem_init(&customer_sem, 0, 0);
108   read_file();
109   begin_time = time(NULL);
110   for (int i = 0; i < counter_list.size(); ++i)
111   {
112     Counter& counter = counter_list[i];
113     counter.id = i + 1;
114     int error = pthread_create(&(counter.tid), NULL, counter_thread, (void*)(&counter));
115     if (error)
116     {
117       cerr << "Create counter thread " << i + 1 << " failed. " << strerror(error) << endl;
118       exit(-1);
119     }
120   }
121
122   for (int i = 0; i < customer_list.size(); ++i)
123   {
124     Customer& customer = customer_list[i];
125     int error = pthread_create(&(customer.tid), NULL, customer_thread, (void*)(&customer));
126     if (error)
127     {
128       cerr << "Create customer thread " << i + 1 << " failed. " << strerror(error) << endl;
129       exit(-1);
130     }
131   }
132
133   for (auto& customer : customer_list)
134     pthread_join(customer.tid, NULL); // WAIT FOR ALL CUSTOMERS TO FINISH
135
136   for (auto& counter : counter_list)
137     pthread_cancel(counter.tid);
138
```

```
139   for (auto& counter : counter_list)
140     pthread_join(counter.tid, NULL); // TEST THAT ALL COUNTER THREADS ARE FINISHED
141
142   cout << "Finish" << endl;
143
144   ofstream output("output.txt");
145   for (auto& customer : customer_list)
146     output << customer.id << ' ' << customer.arrival << ' ' << customer.serve << ' ' << customer.serve
          + customer.wait << ' ' << customer.counter << endl;
147
148   return 0;
149 }
```

输入文件测试样例随机生成 10 个顾客如下，每行三个字段分别为顾客序号、顾客进入银行的时间、顾客需要服务的时间。

1 13 6

2 14 10

3 11 10

4 4 5

5 16 5

6 10 8

7 15 6

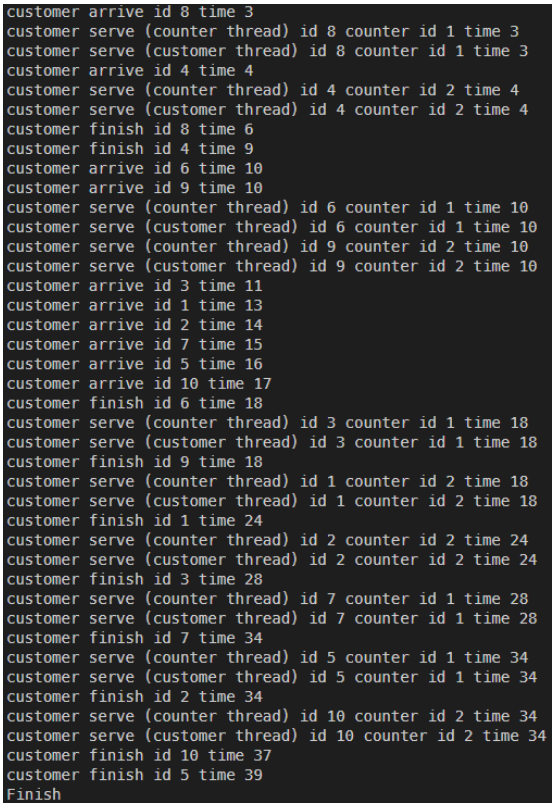
8 3 3

9 10 8

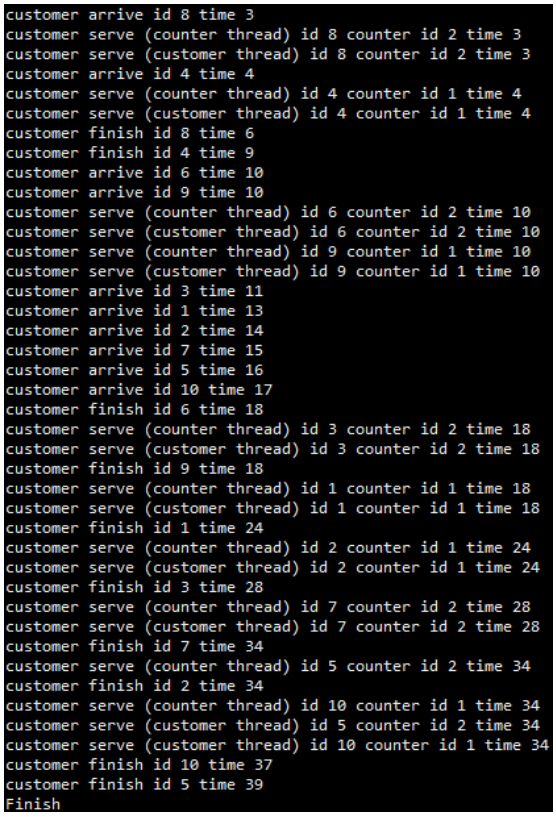
10 17 3

在 Windows 与 Linux 下分别用两个柜员线程测试，运行结果如图 2所示，输出如图 3所示，其中每行五个字段分别为：顾客序号、顾客需进入银行的时间、开始服务的时间、离开银行的时间以及服务柜员号（从 1 开始计数）。

从结果可见，时序运行准确且事件（到达、叫号、完成）发生时间一致，但顾客对应的柜员号未必一致。本样例中，第 9 个时间单位（程序中以秒模拟）后，无顾客，柜员等待；而第 17 个时间单位时，3、1、2、7、5、10 号顾客正确地处在等待队列中等待叫号。这两方面的同步都是运行正确的。注意到 18 个时间单位时两个柜员同时叫号，两次程序运行结果不一致，也表明多线程运行情况不可预期。程序运行中线程调度占用一定时间，但在秒级的时间模拟下影响可忽略。

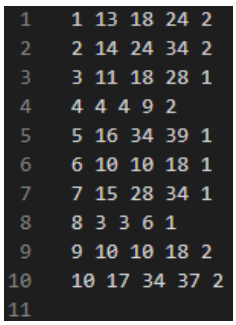


(a) Windows 运行结果

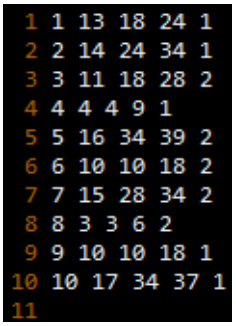


(b) Linux 运行结果

图 2: 程序运行结果



(a) Windows 输出文件



(b) Linux 输出文件

图 3: 程序输出文件

4 思考题

4.1 柜员人数和顾客人数对结果分别有什么影响？

以如上测试样例，改变柜员人数，柜员人数与总时间的关系如表 1所示，绘制曲线如图 4所示。可见增加柜员人数会使总时间减少，但柜员充分多时，柜员大部分时间闲置等待顾客，不再提升总时间。

柜员人数	1	2	3	4	5	6	7	8	9	10
总时间	67	39	30	28	24	24	24	24	24	24

表 1: 柜员人数与总时间关系表

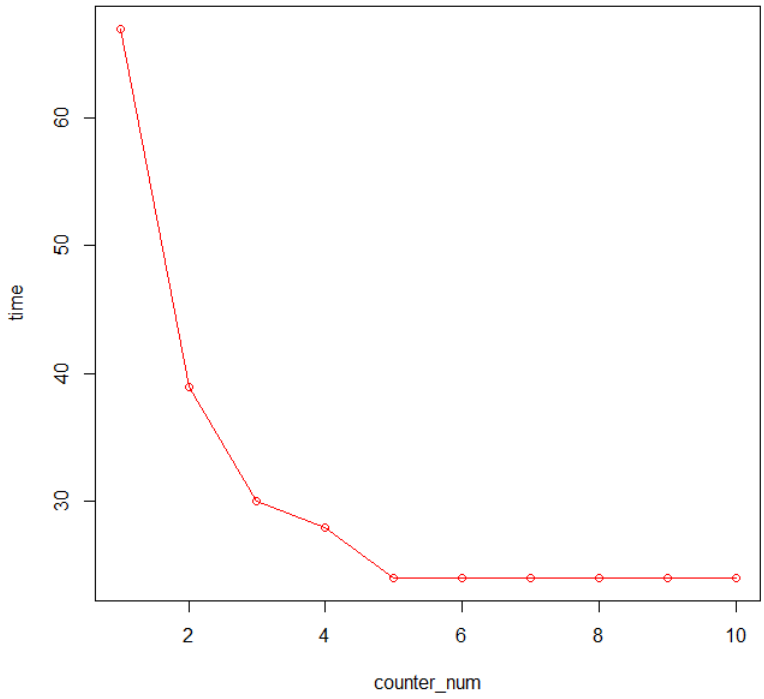


图 4: 柜员人数与总时间关系图

增加顾客人数时，固定 3 个柜员，随机生成顾客输入（随机生成程序代码略），到达时间均在 20 以内，服务时间在 10 以内。顾客人数与总时间的关系如表 2所示，绘制曲线如图 5所

示。在柜员饱和的情况下，顾客人数大致与总时间呈线性关系。在柜员单位时间服务平均人数小于单位时间抵达顾客数时，可视作饱和，总时间由柜员单位时间服务人数决定，因此呈线性。

顾客人数	5	10	15	20	25	30
总时间	28	30	38	44	51	56

表 2: 顾客人数与总时间关系表

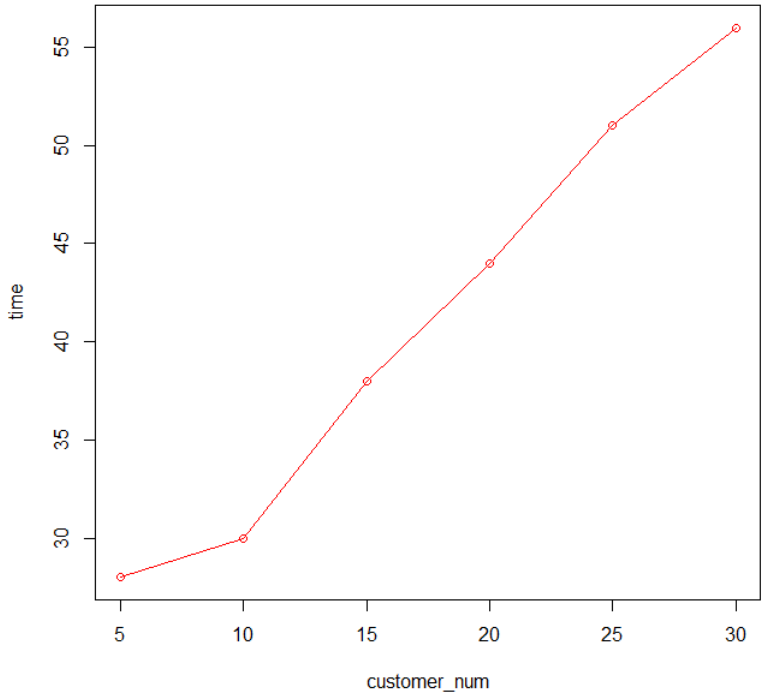


图 5: 顾客人数与总时间关系图

当线程数过大时，程序会出现一定问题，例如线程之间调度时间变得不可忽略，超过了模拟的时间间隔，这在程序中 1 秒的时间单位中。实际测试中大致在达到 1000 个线程（500 个柜员和 500 个顾客）时偶尔出现问题，但这个数目已经充分大。

4.2 实现互斥的方法有哪些？各自有什么特点？效率如何？

1. 禁止中断
   进入临界区前执行关闭中断指令，离开临界区后执行开中断。
   简单但可靠性差，不适合多处理器。

2. 严格轮转法
   进程严格轮流进入临界区。
   忙等待，效率较低，且可能在临界区外阻塞别的进程。

3. Petersen 算法
   可以正常工作的解决互斥问题的算法，仍使用锁。
   忙等待，效率较低。

4. 硬件指令方法
   使用硬件提供不被打断的单条指令读写共享变量。
   适用于任意数目进程，且较简单，但仍有忙等待。

5. 信号量
   使用原语访问信号量管理资源。
   不必忙等，效率较高，但信号量的控制分布在整个程序中，正确性难以保证。

6. 管程
   信号量及其操作的高级语言封装。
   效率同信号量一样，易于管理开发。

7. 消息传递
   用以实现分布式系统的同步、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。

5 实验感想

本次实验涉及多线程的编程，处理多线程的同步、互斥问题。我在实验中深入理解了信号量的基本原理与实际操作，更详细理解了多线程同步、互斥的程序设计方法。我采用了 C 风格的编写方式，使用 pthread 以完成跨平台，而没有采用 C++ 或其它高级语言中抽象程度更高的信号量与互斥量对象，使得实际跨平台时出现了较多问题和困难，例如 Linux 与 Windows 下 clock() 和 sleep() 表现不一致的问题（若使用 C++ 的 std::chrono 则不再成为问题）。下次

编写多线程程序时，可更多考虑使用 C++ 标准库中的互斥量与线程对象，以更简化代码的编写与调试。但总体而言，本次实验使我收获颇多，入门了多线程的程序编写。