

# 进程间同步/互斥问题 - 银行柜员服务问题

---

- 无42 林子恒 2014011054
- Brian Lin,Tzu-Heng's Work
  - Mailto: [lzhbrian@gmail.com](mailto:lzhbrian@gmail.com)
  - Github: [lzhbrian](#)
  - Linkedin: [lzhbrian](#)

## 进程间同步/互斥问题 - 银行柜员服务问题

### 一、问题描述

1. 实现要求
2. 实现提示
3. 测试文本格式
4. 输出要求

### 二、实现

1. 思路：
2. 主要函数实现：

### 三、实验结果

### 四、思考题

### 五、感想

## 一、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. 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。

### 2. 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。

### 3. 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
```

## 4. 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、实现

## 1. 思路：

- 因为我的电脑是 *\*nix* 系列的，所以我采用 **posix** 的函数来实现进程、线程的操作
- 本实验中，我采用 **信号量**、**互斥锁** 的方式来实现顾客与柜员的同步以及互斥，主要思路如下：
  - 信号量：sem\_customers 顾客同步信号量、sem\_servers 柜员同步信号量
  - 互斥锁：mutex\_customers 顾客资源访问互斥锁、mutex\_servers 柜员资源访问互斥锁
  - main()函数首先每个顾客、柜员生成一个线程，顾客睡到(sleep())它进入银行的时间为止
  - 在顾客线程Customer\_do()里：
    - 顾客首先拿号(sem\_post(sem\_customers))
      - 然后进入等待的list(queueing\_cus\_list)里面
    - 等待柜员叫号(sem\_wait(sem\_servers))
    - 柜员叫号后，记录其开始服务时间(start\_serve\_time)
  - 在柜员线程Serve\_do()里：
    - 柜员不断叫号(sem\_wait(sem\_customers))
      - 若叫号成功，获取第一位顾客信息(cus\_No = queueing\_cus\_list[0])，
      - 并将其从队列中erase，加入已开始(或完成)服务顾客队列start\_served\_cus\_list
      - 并记录这个柜员的编号
    - 服务这个顾客(sleep())
    - 服务完成后，检查已开始(或完成)服务顾客队列start\_served\_cus\_list是否已经包括所有顾客
      - 若是，则退出
      - 若否，则完成(sem\_post(sem\_servers))后，继续叫号(sem\_wait(sem\_customers))
  - 最后main函数输出顾客的信息
- 另外，为了程序的debug需要，我还设立了一个mutex\_cout 输出锁，保证不会有多个线程同时输出，导致输出乱序的情况。

## 2. 主要函数实现：

### 1. 顾客结构体：

```
1  /***** Customer 结构体 *****/
2  typedef struct Customer_struct
3  {
4      int id;
5      int enter_time;          // known
6      int serve_time;          // known
7
8      int start_serve_time;    // unknown
9      int leave_time;          // unknown
10     int server_No;           // unknown
11
12     Customer_struct(int id_s, int enter_time_s, int serve_time_s) {
13         id = id_s;
14         enter_time = enter_time_s;
15         serve_time = serve_time_s;
16         start_serve_time = 0;
17         leave_time = 0;
18         server_No = 0;
19     }
20     Customer_struct(){
21         id = 0;
22         enter_time = 0;          // known
23         serve_time = 0;          // known
24
25         start_serve_time = 0;    // unknown
26         leave_time = 0;          // unknown
27         server_No = 0;
28     }
29 } Customer;
```

## 2. 顾客线程：

```
1 // 顾客服务 func
2 void *Customer_do(void* cus_No_v) {
3
4     int cus_No = *(int*)cus_No_v;
5
6     // debug 用 输出当前顾客信息
7     // pthread_mutex_lock(&mutex_cout);
8     // cout << "call customer " << Customers[cus_No].id << endl;
9     // cout << "sleep time: " << Customers[cus_No].enter_time << endl;
10    // pthread_mutex_unlock(&mutex_cout);
11
12    sleep(Customers[cus_No].enter_time);    // 睡到进入的时间
13
14    pthread_mutex_lock(&mutex_cout);
15    now = getSystemTime();
16    cout << (now - open_time)/1000 << ", Customer " <<
Customers[cus_No].id << " comes to the bank!" << endl;
17    pthread_mutex_unlock(&mutex_cout);
```

```
// P: 顾客多一个资源，相当于顾客拿号
sem_post(sem_customers);
    pthread_mutex_lock(&mutex_customers);    // lock customer
        queueing_cus_list.push_back(cus_No); // 顾客入队
        now = getSystemTime();
    pthread_mutex_unlock(&mutex_customers); // free customer
// V: 柜员少一个资源，相当于等待柜员叫号
sem_wait(sem_servers);

// 记录开始服务的时间
Customers[cus_No].start_serve_time = (now - open_time)/1000;

return 0;
```

}

## 3. 柜员线程：

```
```c++
// 柜员服务 func
void *Serve_do(void* server_No_v) {
    int server_No = *(int*)server_No_v;
    while(1) {

        // V: 顾客少一个资源，相当于柜员叫号
        sem_wait(sem_customers);
```

```

// 有没有柜员叫同一个号
pthread_mutex_lock(&mutex_servers);           // lock server
    int cus_No = queueing_cus_list[0];         // 获取第一个
顾客信息
    queueing_cus_list.erase(queueing_cus_list.begin());

    start_served_cus_list.push_back(cus_No);    // 只要开始服
务就加入队列
    // cout << start_served_cus_list.size() << endl;

    Customers[cus_No].server_No = server_No;    // 记录服务这
个顾客的柜员
pthread_mutex_unlock(&mutex_servers);    // unlock server

pthread_mutex_lock(&mutex_cout);
    now = getSystemTime();
    cout << (now - open_time)/1000 << ", Customer " <<
Customers[cus_No].id << " starts being served by server " << server_No << endl;
    Customers[cus_No].start_serve_time = (now - open_time)/1000;
pthread_mutex_unlock(&mutex_cout);

sleep(Customers[cus_No].serve_time);    // 服务

pthread_mutex_lock(&mutex_servers);           // lock server
    if (start_served_cus_list.size() == n)
    {
        pthread_mutex_lock(&mutex_cout);
        now = getSystemTime();
        cout << (now - open_time)/1000 << ",
Customer " << Customers[cus_No].id << " suc. served by server " << server_No <<
endl;
        Customers[cus_No].leave_time = (now -
open_time)/1000;

        pthread_mutex_unlock(&mutex_cout);

        cout << server_No << " server end!" << endl;
        // 如果所有顾客都完成服务了, 就break
        pthread_mutex_unlock(&mutex_servers);    // unlock
server
        break;
    }
pthread_mutex_unlock(&mutex_servers);    // unlock server

```

```

        pthread_mutex_lock(&mutex_cout);
        now = getSystemTime();
        cout << (now - open_time)/1000 << ", Customer " <<
Customers[cus_No].id << " suc. served by server " << server_No << endl;
        Customers[cus_No].leave_time = (now - open_time)/1000;
        pthread_mutex_unlock(&mutex_cout);

        // 柜员多一个资源, 相当于柜员完成一个服务
        sem_post(sem_servers);

    }
    return 0;
}

```

#### 4. 主函数:

```

1  /***** Main func *****/
2  int main(int argc, char * argv[])
3  {
4
5      int id, enter_time, serve_time;
6
7      // Read File into struct
8      // n: 一共有多少 顾客;
9      ifstream file("./Customer_easy.txt");
10     while (!file.eof())
11     {
12         file >> id >> enter_time >> serve_time;
13         Customers[n].id = id;
14         Customers[n].enter_time = enter_time;
15         Customers[n].serve_time = serve_time;
16         n = n + 1;
17     }
18     // debug 用 cout 读入的数据
19     // for (int i = 0; i < n; ++i)
20     // {
21     //     cout << Customers[i].id << "," << Customers[i].enter_time << ","
22     //         << Customers[i].serve_time << endl;
23     // }
24
25     // Semaphore init
26     sem_unlink("sem_servers");
27     sem_unlink("sem_customers");
28     sem_servers = sem_open("sem_servers", O_CREAT, 0, server_num);
29     sem_customers = sem_open("sem_customers", O_CREAT, 0, 0);
30
31     // Start time

```

```

32     open_time = getSystemTime();
33
34     // Servers, Customers threads init
35     pthread_t server_threads[server_num];
36     pthread_t customer_threads[customer_num];
37     int tid_s[server_num];
38     for (int i = 0; i < server_num; ++i) {
39         tid_s[i] = i;
40         if(pthread_create(&server_threads[i], NULL, Serve_do, &tid_s[i]))
{
41             printf("\n ERROR creating server thread %d", tid_s[i]);
42             exit(1);
43         }
44     }
45     int tid_c[customer_num];
46     for (int j = 0; j < n; ++j) {
47         tid_c[j] = j;
48         if(pthread_create(&customer_threads[j], NULL, Customer_do,
&tid_c[j])) {
49             printf("\n ERROR creating customer thread %d", tid_c[j]);
50             exit(1);
51         }
52     }
53
54     // Wait for threads to complete
55     for (int i = 0; i < server_num; ++i) {
56         if(pthread_join(server_threads[i], NULL)) {
57             cout << "\n ERROR joining server thread: " << i << endl;
58             exit(1);
59         }
60     }
61     for (int j = 0; j < n; ++j) {
62         if(pthread_join(customer_threads[j], NULL)) {
63             cout << "\n ERROR joining customer thread: " << j << endl;
64             exit(1);
65         }
66     }
67
68     // Destroy Semaphore, Mutex
69     sem_close(sem_customers);
70     sem_close(sem_servers);
71
72     pthread_mutex_destroy(&mutex_customers);
73     pthread_mutex_destroy(&mutex_servers);
74     pthread_mutex_destroy(&mutex_cout);
75
76     cout << "All done!" << endl;

```

```
77
78 // output result
79 cout << "ID\tEnter\tStart\tLeave\tServerNo." << endl;
80 for (int i = 0; i < n; ++i)
81 {
82     cout << Customers[i].id << "\t"
83         << Customers[i].enter_time << "\t"
84         << Customers[i].start_serve_time << "\t"
85         << Customers[i].leave_time << "\t"
86         << Customers[i].server_No << endl;
87 }
88 return 0;
89 }
```



### 三、实验结果

1. 一个很简单的例子，3个顾客，2个柜员，如下：

顾客编号	进入时间	服务时间
1	1	10
2	5	2
3	6	3

输出结果为：

ID	Enter	Start	Leave	ServerNo.
1	1	1	11	1
2	5	5	7	0
3	6	7	10	0

2. 一个稍微复杂一些的例子，10个顾客，4个柜员，如下：

顾客编号	进入时间	服务时间
1	1	10
2	5	2
3	6	3
4	6	5
5	3	8
6	7	1
7	10	5
8	9	7
9	2	8
10	8	2

输出结果为：

ID	Enter	Start	Leave	ServerNo.
1	1	1	11	0
2	5	5	7	3
3	6	7	10	3
4	6	10	15	1
5	3	3	11	2
6	7	10	11	3
7	10	11	16	3
8	9	11	18	2
9	2	2	10	1
10	8	11	13	0

可以看到，我们成功地解决了银行柜员服务问题！

## 四、思考题

1. 柜员人数和顾客人数对结果分别有什么影响？
  - 柜员人数不变的时候，顾客越多，每个顾客平均等待的时间就越长，这很自然
  - 顾客数量不变的时候，柜员人数越多，每个顾客平均等待的时间就越长，这也很自然
    - 需要注意的是，本程序没有考虑到柜员人数 > 顾客人数的情况，如果是这种情况，本程序会无法结束，因为有一些柜员永远也无法分配到顾客而一直等待，停不下来
2. 实现互斥的方法有哪些？各自有什么特点？效率如何？
  - 主流、实用的实现互斥的方法有以下几种：
    - Peterson算法、TSL、XCHG：可能会有忙等待的情况
    - 信号量：包括Mutex、Semaphore：将互斥进入系统调用层面，效率相对于上述较高

## 五、感想

- 一开始我设置了一个list，在顾客完成服务之后才将其加入，柜员需要在判断这个list的个数等于全部顾客总数的时候才会退出线程，但实际操作的过程中发现这样是不可行的；因为如果有一个线程判断还有顾客为完成服务，进入等待状态，但此时只是有一些顾客没有完成服务，已经不会再有顾客进来，如此一来，这个柜员线程就永远也无法结束。
  - 解决方法就是，一旦柜员开始服务一个顾客，就将这个顾客放入list里面。
- 第一次用C++写多线程程序，阵痛期非常久，在网上查阅了很多资料，最终还是完成了，很开心～