银行柜员服务问题

——《操作系统》课程实验一：

进程间同步/互斥问题

黄嘉浩　　无 27　　2022010666

huang-jh22@mails.tsinghua.edu.cn

2025 年 5 月 10 日

目录

1 问题描述及要求

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。

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。

编程语言 C++

编程环境 Windows 11

2 设计思路及程序结构

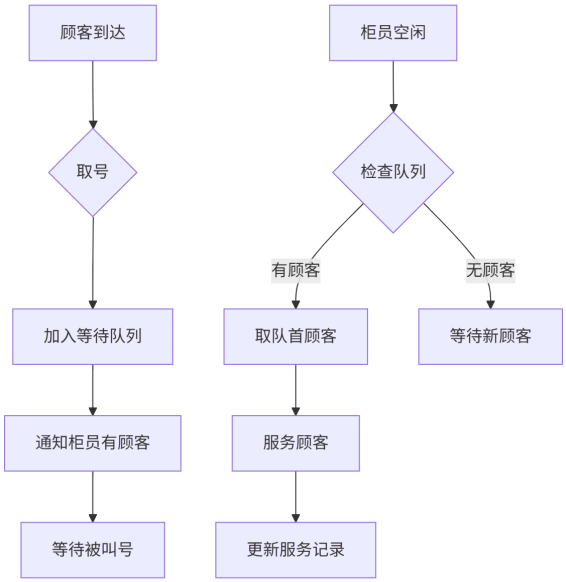
2.1 设计思路与核心流程

本实验利用同步/互斥原语（P、V 操作）实现银行柜员服务问题。

下面先来分析同步/互斥对象。首先，顾客取号与柜员叫号之间一定是互斥的。顾客当前取到的号（current_get_num）与柜台当前叫到的号（service_call_num）需要在互斥锁（mutex）的保护下进行访问。顾客取号、柜员叫号时需要执行 P 操作（WaitForSingleObject），顾客取号、柜员叫号后需要执行 V 操作（ReleaseMutex）。此外，顾客和柜员是两个进程。顾客进程在取号后进入等待队列（customer_wait_list）；柜员进程在叫号后进入工作状态，从等待队列选出一位顾客进行服务。我们使用信号量（service_semaphore）来实现互斥和同步。

为了监视进程的结束,我们还需要设置一个全局变量 current_served_num 用以表示已经被服务完毕的顾客数量。当已经被服务完毕的顾客数量与总顾客数量一致时，表示进程结束（注意当柜台数很多时，仍然有闲置柜台，但这不影响整体进程结束）。

核心流程



2.2 程序结构与关键代码解释

1. 全局变量

```
int current_get_num = 0;    //顾客取号
int service_call_num = 0;     //柜台叫号
int customer_num = 0;              //总顾客数量
int current_served_num = 0;    //已经被服务好的顾客数量

int customer_info[CUSTOMER_NUM_MAX][6] = {}; //顾客信息: No., arrival time, service time,
    Counter No., Start time, End time
int customer_wait_list[CUSTOMER_NUM_MAX][2] = {}; //等待队列: 顾客ID, 顾客No.
```

除了需要维护的等待队列外，顾客信息也需要维护。顾客信息包括顾客编号、到达时间、服务时间、服务柜台编号、开始时间、结束时间。其中顾客编号、到达时间、服务时间从测试文件（test.txt）读入，柜台编号、开始时间、结束时间由柜员进程生成。

2. 信号量

```
HANDLE customer_thread[CUSTOMER_NUM_MAX]; //顾客线程
HANDLE counter_thread[N];                 //柜台线程
HANDLE service_semaphore = CreateSemaphore(NULL, 0, CUSTOMER_NUM_MAX, NULL); // 柜台服务信号量
HANDLE counter_call_mutex = CreateMutex(NULL, FALSE, NULL); // 柜台叫号锁
HANDLE customer_get_mutex = CreateMutex(NULL, FALSE, NULL); // 顾客取号锁
```

3. 顾客线程

```
    long long customer_id = (long long)param;

    // Simulate customer arrival
    Sleep(1000 * customer_info[customer_id][1]); // Arrival time

    // Get a number
    WaitForSingleObject(customer_get_mutex, INFINITE);
    customer_wait_list[current_get_num][0] = customer_id; // Customer ID
    customer_wait_list[current_get_num][1] = customer_info[customer_id][0]; // Customer No.
    current_get_num++;
    cout << "Customer " << customer_id << " got number " << current_get_num << endl;
    ReleaseMutex(customer_get_mutex);

    // Signal that the customer is ready to be served
    ReleaseSemaphore(service_semaphore, 1, NULL);
    return 0;
```

顾客线程先模拟顾客到达银行（Sleep 实现）。由于每个顾客到达时间不同，所以 Sleep 时间与顾客信息（customer_info）中读入的到达时间成正比。顾客到达后获取一个号码（current_get_num），并将其放入等待队列（customer_wait_list）。获取号码后，顾客线程释放信号量（service_semaphore），表示顾客已准备好被服务。

4. 柜员线程

```
int service_time = 0;
int customer_index = 0;

while (true) {
    // Wait for a customer to be ready
    WaitForSingleObject(service_semaphore, INFINITE);

    // Call the next customer
    WaitForSingleObject(counter_call_mutex, INFINITE);
    customer_index = service_call_num++; // index of the customer being served
    cout << "Counter " << (long long)param << " is serving customer " <<
        customer_wait_list[customer_index][1] << endl;
    service_time = customer_info[customer_index][2]; // Service time
    customer_info[customer_index][3] = (long long)param; // Counter number
    customer_info[customer_index][4] = time(NULL) - init_time; // Start time being served
    customer_info[customer_index][5] = customer_info[customer_index][4] + service_time; // End
        time
    ReleaseMutex(counter_call_mutex);

    // Simulate service time
    Sleep(1000 * service_time);
    current_served_num++;
}
```

柜员线程在获取信号量后，调用 WaitForSingleObject 函数等待顾客取号锁（customer_get_mutex），然后从等待队列中获取一个顾客进行服务。柜员线程在服务顾客时，柜员线程会将柜员编号、开始时间、结束时间写入顾客信息（customer_info）中。

5. 主程序

核心代码

```
// Create customer threads
for (i = 0; i < customer_num; i++) {
    customer_thread[i] = CreateThread(NULL, 0, CustomerThread, (PVOID)i, 0, NULL);
    if (customer_thread[i] == NULL) {
        cout << "Error creating customer thread!" << endl;
        return 1;
```

```
7           }
8       }
9
10      // Create counter threads
11      for (i = 0; i < N; i++) {
12          counter_thread[i] = CreateThread(NULL, 0, CounterThread, (PVOID)i, 0, NULL);
13          if (counter_thread[i] == NULL) {
14              cout << "Error creating counter thread!" << endl;
15              return 1;
16          }
17      }
18
19      // Wait for all customer threads to finish
20      while (current_served_num < customer_num) {
21          Sleep(100);
22      }
23
24      for (i = 0; i < customer_num; i++) {
25          CloseHandle(customer_thread[i]);
26      }
27
28      for (i = 0; i < N; i++) {
29          CloseHandle(counter_thread[i]);
30      }
```

主程序的代码逻辑较直接。首先从 test.txt 文件中读取顾客信息（顾客编号、到达时间、服务时间），并将其存入顾客信息数组（customer_info）中。然后创建顾客线程（customer_thread）和柜员线程（counter_thread），并将顾客编号传入顾客线程，柜员编号（0∼N-1）传入柜员线程中。之后，主程序会等待所有顾客线程结束（current_served_num < customer_num），并关闭所有线程句柄。最后，主程序会输出所有顾客的服务信息。

为了方便测试,笔者编写了自动化生成测试文件的代码（create_exp.cpp），该代码会按照标准格式随机生成顾客编号、到达时间、服务时间，并将其写入 test.txt 文件中。

3 程序运行情况

测试文件格式

```
顾客序号 进入银行的时间 需要服务的时间
```

输出文件格式

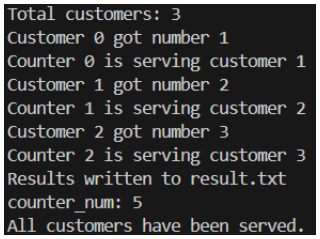
```
顾客序号 进入银行的时间 需要服务的时间 服务柜员号 开始服务的时间 离开银行的时间
```

3.1 作业测试样例

测试文件

```
// test.txt
1 1 10
2 5 2
3 6 3
```

进程监视



输出结果

```
// result.txt
Customer No. Arrival Time Service Time Counter No. Start Time End Time
1 1 10 0 2 12
2 5 2 1 6 8
3 6 3 2 7 10
```

注意到以上测试样例中，柜台数量为 5，顾客数量为 3。柜台数量大于顾客数量，输出结果是正确的。

3.2 其他测试样例

下面来测试柜台数量为 3，顾客数量为 10 的情况。

测试文件

```
// test.txt
1 8 6
2 10 7
3 8 8
4 18 4
5 1 1
6 14 8
7 19 9
8 15 2
9 10 6
10 14 8
```

进程监视

```
Total customers: 10
Customer 4 got number 1
Counter 0 is serving customer 5
Customer 2 got number 2
Customer 0 got number 3
Counter 1 is serving customer 3
Counter 2 is serving customer 1
Customer 1 got number 4
Counter 0 is serving customer 2
Customer 8 got number 5
Customer 9 got number 6
Customer 5 got number 7
Counter 0 is serving customer 9
Counter 1 is serving customer 10
Customer 7 got number 8
Counter 0 is serving customer 6
Counter 2 is serving customer 8
Customer 3 got number 9
Counter 2 is serving customer 4
Customer 6 got number 10
Counter 1 is serving customer 7
Results written to result.txt
counter_num: 3
All customers have been served.
```

输出结果

```
// result.txt
Customer No. Arrival Time  Service Time  Counter No.  Start Time  End Time
1            8             6             0            1           7
2            10            7             1            8           15
```

```
5  3   8   8   2   8   16
6  4   18  4   0   10  14
7  5   1   1   0   14  15
8  6   14  8   1   15  23
9  7   19  9   0   15  24
10 8   15  2   2   16  18
11 9   10  6   2   18  24
12 10  14  8   1   23  31
```

由输出结果可以很清晰的看到，在柜台数量小于顾客数量的情况下，输出结果依旧正确。这说明笔者同步/互斥的实现是正确且有效的。

4 思考题

1. 柜员人数和顾客人数对结果分别有什么影响？

为了更清晰的对比柜员人数和顾客人数对结果的影响，笔者考察了不同柜员人数和顾客人数的组合情况。总运行时间变化表现了柜员人数和顾客人数对结果的影响。

测试文件

```
1  // test.txt
2  1 8 6
3  2 10 7
4  3 8 8
5  4 18 4
6  5 1 1
7  6 14 8
8  7 19 9
9  8 15 2
10 9 10 6
11 10 14 8
```

表 1: 柜员人数和顾客人数对结果的影响

柜员人数 N	顾客人数 customer_num	运行时间 (s)
2	10	39.357
3	10	34.060
5	10	29.966
8	10	30.063
12	10	30.465

总体来说，当顾客人数一定时，柜员人数增加，运行时间会减少。这是符合常识的。但是，当柜员人数增长到一定程度后，随着柜员人数的继续增长，运行时间并没有显著下降。相反，运行时间会略有增加。运行时间达到瓶颈的原因是某些顾客需要服务的时间很长（一直占用着线程），那么柜员的多少并不会加快运行速度。运行时间会略有增加的原因是柜员线程在等待顾客线程时会消耗 CPU 资源，导致运行时间增加。

2. 实现互斥的方法有哪些? 各自有什么特点? 效率如何?

方法类型	实现方式	特点	效率
硬件方法	中断禁用	仅适用于单处理器，通过关闭中断 实现原子操作，可靠性高但扩展性差	高（无上下文切换）
	原子指令（TS/Swap）	利用 CPU 指令（如 TestAndSet、Exchange） 实现原子操作，适用于多核系统	极高（硬件级原子性）
软件方法	Peterson 算法	仅适用于两进程，无需硬件支持 但需忙等待，可能违反让权等待原则	低（忙等待消耗 CPU）
	单/双标志法	通过共享变量实现，易导致死锁或饥饿 （如双标志法后检查的活锁问题）	低（需多次状态检查）
OS 同步原语	互斥锁（Mutex）	内核态实现，支持阻塞等待 但存在上下文切换开销	中（依赖 OS 调度）
	自旋锁（Spinlock）	用户态忙等待，适合短临界区 避免上下文切换但浪费 CPU	高（无切换）/低（长时间等待）
	信号量（Semaphore）	支持计数与阻塞，灵活但需严格 的 P/V 操作配对以避免死锁	中（系统调用开销）
	读写锁（RW Lock）	区分读/写操作，允许多读单写 减少竞争，适合读多写少场景	高（减少无效竞争）
分布式方法	集中式算法	中央协调者管理请求队列 单点瓶颈，可靠性差	低（消息数 O(n)）
	令牌环算法	公平轮转，无单点故障 但存在无效令牌传递开销	中（延迟与节点数相关）
	分布式投票	需多数节点同意（如 Paxos） 容错性强但消息复杂度高	低（消息数 $O(n^2)$）

表 2: 实现互斥的方法及其特点和效率

5 实验体会

多线程编程通过精巧的并发设计扩展问题处理维度。在模拟银行柜员服务问题的 C++ 实践中，笔者体会到实际开发中需明确共享资源边界、合理使用互斥锁保护临界区，以及线程调度的时序敏感性。只要建立清晰的并发模型，准确定义受保护变量，辅以系统化的竞态分析，便能构建可靠高效的并行程序。这种化繁为简的过程令人感受到计算机科学的精妙张力。