银行柜员服务问题

林仲航 无63 2016011051

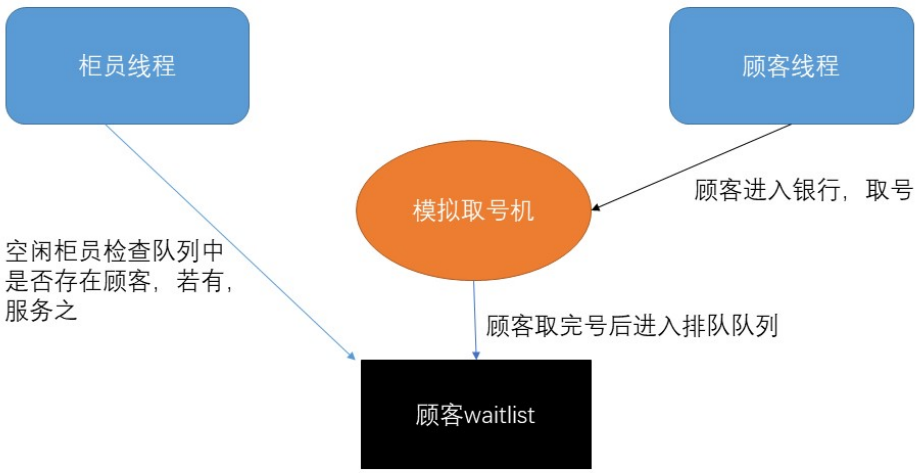
1、问题描述：

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。利用PV操作实现柜员与顾客的同步。

2、总体解决思路：

顾客与柜员个体都可以作为一个线程存在。叫号的过程可以形象为一个机器，对每个刚进银行的顾客实行赋予新号码的任务（故而，该过程需要实现互斥）。顾客进入银行后，相当于进入一个等待队列中。柜员空闲时，对队列进行检查，如果有顾客则进行服务，同时队列人数减一（显然，此过程必须保证队列的线程安全）。如此循环往复。

3、总体构架框图：



4、程序设计细节：

a、信号量实现机制：

关于信号量的实现，由于C++新标准中去除了信号量，故我们需要利用互斥锁以及条件变量来构造一个信号量类Semaphora。具体构造采用了https://segmentfault.com/a/1190000006818772中的定义。代码如下：

```
class Semaphora {
    //define a semaphora to use PV operations
private:
    std::mutex mtx;
    std::condition_variable cv;
    int signal;

public:
    Semaphora(int _count = 1) { signal = _count; }
    void up() {
        std::unique_lock<std::mutex> lock(mtx);
        signal++;
        cv.notify_one();
    }
    void down() {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [=] {return signal>0; });
        signal--;
    }
    int get_signal() { return signal; }
};
```

b、顾客线程实现机制：

(1) 顾客线程的基本信息：

定义了Consumer结构体来记录顾客的信息。

```
struct Consumer {
    int id;
    int serve_time;
    int enter_time;
    Consumer(int _id = 0, int _t = 0, int _e = 0) :
        id(_id), serve_time(_t), enter_time(_e) {}
};
```

(2) 进入银行时间：

利用C++的sleep_for函数，控制其进入银行的时间。在此之前首先完成所有顾客线程的初始化并将其阻塞，直到所有顾客初始化完毕后进行唤醒。

```
{
    std::unique_lock<std::mutex> lk(begin_m);
    begin_cv.wait(lk, [] {return ready; });
}//等待直到所有顾客均被构造完毕

std::this_thread::sleep_for(std::chrono::milliseconds(10*c.enter_time));
```

(3) 顾客取号以及进入队列：

定义全局变量number，每次顾客进入银行时拿到number的号码，随后number进行加1操作。由于每个顾客的号码必须不同，必须实现拿号的互斥操作。随后顾客进入队列waitlist。对于waitlist，由于柜员也会对waitlist进行操作，故而需要保证waitlist的线程安全，故而也需要利用PV操作实现互斥。

在使用PV操作中定义了几个信号量：

```
Semaphora call_signal(1);  //叫号的信号量
Semaphora take_signal(1);  //取号的信号量
Semaphora list_signal(1);  //对waitlist操作的信号量
```

随后有关PV操作以及临界区的代码如下：

```
take_signal.down(); //enter region

msg_vec[c.id-1].enter_t = get_millisec(sys_clk::now(),begin_time);
number_id.insert(std::pair<int, int>(number, c.id));//确定进入的时间
Consumer cos(number++, c.serve_time, c.enter_time);

//将顾客放入waitlist
list_signal.down();
waitlist.push(cos);
list_cv.notify_all();   //通知等待的柜员线程
list_signal.up();

take_signal.up();   //out region
```

c、柜员线程：

柜员线程初始化后一直进行循环，若队列中存在顾客，则取出并服务（睡眠对应的服务时间）；若不存在，则被阻塞直到有顾客进入。随后继续循环。当服务顾客总数达到上限后（即顾客均已被服务），柜员线程返回。

```
void server(int _id) {
    //柜员线程，保证取号互斥且访问waitlist也互斥

    bool get_work = false;
    Consumer cos;
    int this_id = 0;

    while (1) {
        {
        std::unique_lock<std::mutex> lk(list_m);
        list_cv.wait(lk, [] {return (!waitlist.empty()) | done; });}
```

```
        //当waitlist有顾客进入时，唤醒所有在等待waitlist的线程
        call_signal.down();
        list_signal.down();
        //enter region

        if (!waitlist.empty()) {
            get_work = true;
            cos = waitlist.front();
            waitlist.pop();
            this_id = number_id.at(cos.id);
            msg_vec[this_id - 1].begin_t = get_millisec(sys_clk::now(), begin_time);
            msg_vec[this_id - 1].server_id = _id;
        }

        //out region
        list_signal.up();
        //std::cout << "server " << _id << "out region" << std::endl;
        call_signal.up();

        if (get_work) {

            std::this_thread::sleep_for(std::chrono::milliseconds(10*cos.serve_time));
            //使线程休眠对应时间
            {
                done_signal.down();
                done_num++;             //保证done_num 线程安全
                done_signal.up();
            }
            msg_vec[this_id - 1].leave_t = get_millisec(sys_clk::now(), begin_time);

        }
        if (done_num == consumer_num) {
            done = true;
            list_cv.notify_all();
            //std::cout << "server "<<_id <<  "return " << std::endl;
            return;
        }
        get_work = false;
    }
}
```

d、计时：

利用C++的chrono库中的system_clock::time_point进行时间的记录，主要记录以下几个时间点：

- 顾客线程均被初始化后并被唤醒的时间点
- 顾客进入银行的时间点
- 顾客开始被服务的时间点
- 顾客完成服务的时间点

并将上述数据存到结构体msg中，每个msg对应一个顾客。最终结果即可从msg中得到。

相关代码如下：

```
//consumer初始化，记录时间点，唤醒所有顾客线程
ready = true;
begin_time = sys_clk::now();
begin_cv.notify_all();


void consumer(Consumer c){
    //...
    msg_vec[c.id-1].enter_t = get_millisec(sys_clk::now(), begin_time);
    //记录进入银行时间点

    //...
}

void server(int _id){
    //...
    msg_vec[this_id-1].begin_t = get_millisec(sys_clk::now(), begin_time);
    //记录开始服务时间
    //...
    msg_vec[this_id-1].leave_t = get_millisec(sys_clk::now(), begin_time);
    //记录顾客离开的时间

}
```

5、实验环境及操作：

Win10操作系统+VisualStudio2017，正确启动该程序需要在**项目属性**中加入如下命令行参数:

```
consumer.txt  server_num(柜员数量)
```

若在Linux下运行该程序，采用g++编译时需要在后面加上 **-pthread**，否则编译可能无法通过。

此外搭配一个随机生成顾客txt的python文件，使用命令如下：

```
python generate_consumer.py  consumer_num(顾客数量)
```

需要注明的是，程序中的时间为1单位对应10ms。打印结果时显示的时间单位为10ms。

主程序会在命令行输出如下信息：

1. 每个顾客被服务的信息（顾客id，柜员id，进入时间，开始服务时间，离开时间）
2. 总共耗时

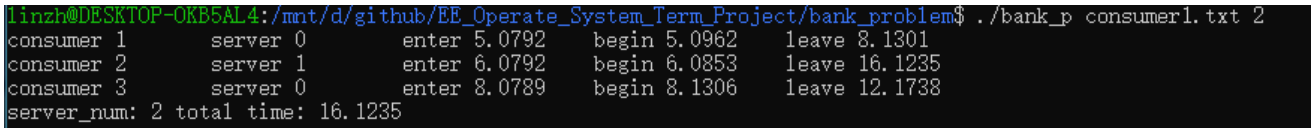
6、程序测试结果：

我们先使用一个简单的例子来查看程序的正确性：

输入样例（在consumer1.txt中）为：

```
1 5 3
2 6 10
3 8 4
```

输出结果为：



由结果可知，顾客1在时间5时进入银行，被柜员0服务3单位时间。随后顾客2在6时进入银行，被柜员1服务。当顾客1服务完后，柜员0转而服务顾客3。程序运行符合逻辑。

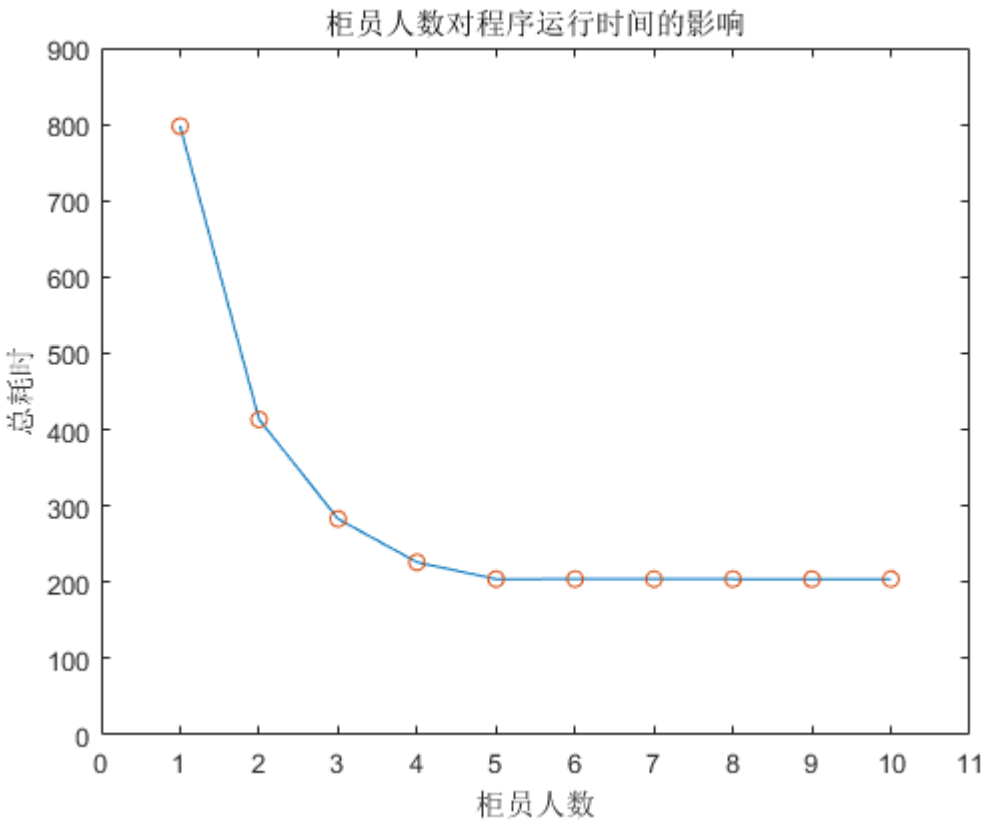
由于实际程序的调度等操作需要耗费一些时间，实际的时间并不为整数x10ms，但是当一个时间单位较大（如10ms）时，小数部分的误差可以很小到可以忽略的地步。

7、思考题：

(1)

首先我们探讨柜员人数对程序运行的影响。在此我们选择生成一个包含40个顾客信息的输入样例（consumer.txt)，随后依次测试不同柜员人数下的时间，结果绘制成曲线如下：

人数	1	2	3	4	5	6	7	8	9	10
时间	797.69	413.28	282.98	226.14	204.13	204.20	204.17	204.17	204.08	204.14



可以看出总时间随着柜员人数的增多而下降，下降到一定程度后趋于稳定（此时认为达到饱和），每增加一个柜员所带来的边际效益在减少。

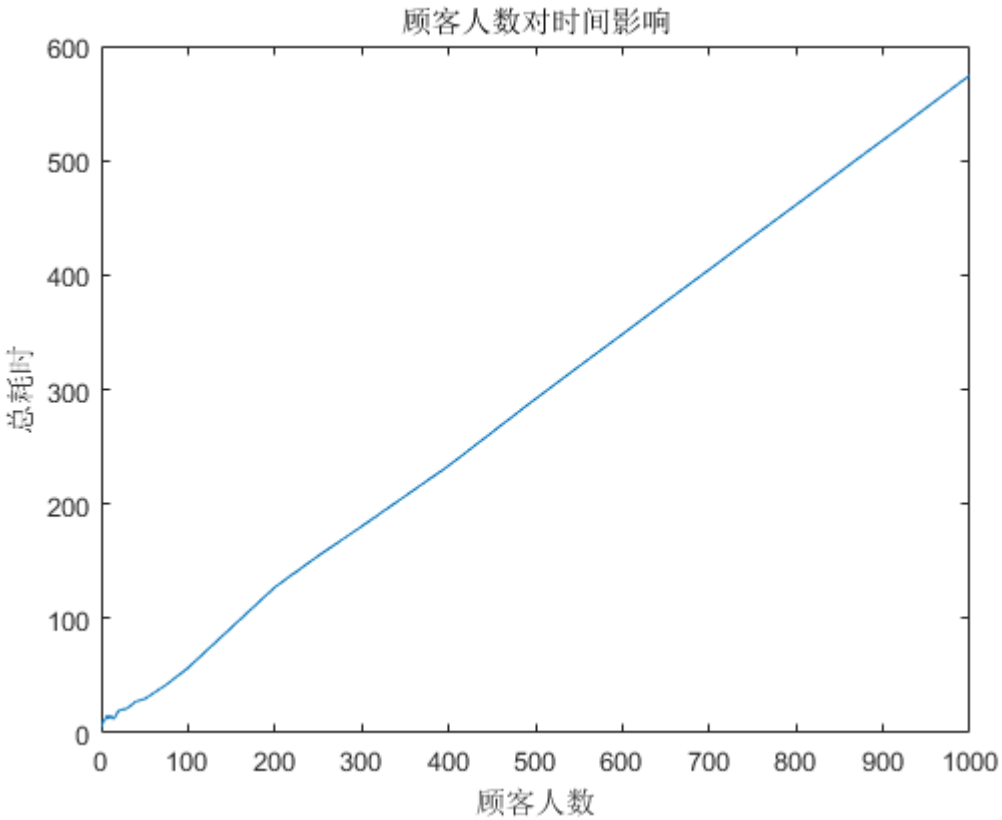
随后我们探讨顾客人数对总运行时间的影响，此时我们固定柜员人数为10人，逐渐增加顾客人数，由于较少的顾客人数下运行时间有着很大的随机性（即与顾客进入时间、需要服务时间有关），故我们需要对顾客输入序列做出相应的修改，方案如下：

- 顾客进入的时间限定在一个较小的区间内（1,5）
- 顾客的服务时间在一个（1,10）内进行取值

此时可以近似认为顾客在很接近的时间内进入银行，此时的结果方具有讨论意义

结果如下：

顾客	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
时间	5.15	9.20	11.16	10.14	12.71	14.14	15.22	13.25	12.15
顾客	10	15	20	30	40	50	75	100	200
时间	15.11	12.28	19.34	21.41	27.48	29.48	42.04	56.76	127.29
顾客	250	300	400	500	1000				
时间	154.69	180.44	233.34	291.83	574.46				



如图，顾客人数对于总时间的影响接近于线性（在人数少时，具有一定的随机性）。直观上也容易理解，当顾客人数大于柜员人数时，对于总耗时的影响必然是正相关。

(2)

实现互斥的方法有：

(a) 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。

- 优点：简单
- 缺点：可靠性差，不适用于多处理器。

(b) 严格轮换法

- 缺点：忙等待，可能在临界区外阻塞别的进程

(c) 锁变量

连续测试直到锁变量出现某个值为止

- 缺点：忙等待
- 优点：等待时间非常短时可以用

(d) Peterson算法

- 优点：解决了互斥访问的问题，可以正常工作
- 缺点：忙等待

(e) 硬件指令方法

- 优点：适用于任意数目进程；简单且容易验证正确性；可以支持进程中存在多个临界区。
- 缺点：忙等待

(f) 信号量

- 优点：实现了进程互斥，原子操作可以避免忙等待；可以实现进程间的同步。
- 缺点：当程序中存在多个临界区，分析正确性较难

(g) 管程

- 优点：可以提高代码可读性，便于修改维护，正确性易于保证；
- 缺点：编译器必须要识别管程并对互斥做出安排。

(h) 消息传递

- 优点：可以完成不同机器间的进程通信
- 缺点：消息传递可能失效，性能可能下降

8、感想：

通过本次实验我深入理解了信号量概念及其操作。同时对于C++的线程互斥等操作有了了解。