操作系统实验 1-5
银行柜员服务问题

马栩杰
2014011085
无 43 班

2016 年 12 月 3 日

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 实验要求

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 程序设计

3.1 整体设计

为了实现柜员与顾客之间的同步，使用一个全局的信号量来记录正在等待的顾客 (waiting customers)。每个顾客进入银行拿到号之后，对 waiting customers 执行 V 操作，表示需要服务的顾客数量增加 1。与此同时，柜员在状态为空闲时对 waiting customers 执行阻塞的 P 操作，直到需要服务的顾客数量大于 0 时才开始叫号。具体实现介绍如下。

3.2 顾客线程

顾客线程的主要流程：

1. sleep 一段时间后进入银行
2. 拿号（需要互斥）

3. 对 waiting customers 执行 V 操作
4. 阻塞在条件变量，等待被叫号
5. 被叫号，将柜员的状态改为正在服务
6. 记录开始服务的时间
7. sleep 一段时间，表示正在接受服务
8. 记录结束服务的时间
9. 通过条件变量通知柜员结束服务
10. 离开银行

其代码实现如下：

```
void CustomerThread(Customer *customer) {
  Sleep(customer->enter_time);  // 在 enter time 时进入银行
  printf("customer %d enter bank at %d\n", customer->id, Time());

  /* enter bank */
  pthread_mutex_lock(&customer->mutex);
  customer->ticket = GetTicket();  // 进入银行后先拿号
  pthread_mutex_unlock(&customer->mutex);

  printf("customer %d get ticket %d\n", customer->id, customer->ticket);

  sem_post(&waiting_customers);  // V 操作，等待的顾客数量增加 1

  /* start waiting */
  printf("customer %d is waiting\n", customer->id);
  while (customer->clerk_id == -1) {
    pthread_cond_wait(&customer->cond, &customer->mutex);  // 等待柜员叫自己的号
  }
  pthread_mutex_unlock(&customer->mutex);

  /* being served */
  Clerk *clerk = &Clerks[customer->clerk_id];  // 被 clerk 叫到
  printf("customer %d is called by clerk %d\n", customer->id, clerk->id);

  pthread_mutex_lock(&clerk->mutex);
  clerk->status = SERVING;
  pthread_cond_broadcast(&clerk->cond);  // 通知 clerk 开始给自己服务
  pthread_mutex_unlock(&clerk->mutex);

```

```
  printf("customer %d being served by clerk %d\n", customer->id, clerk->id);

  customer->service_start_time = Time();  // 记录服务开始的时间
  Sleep(customer->service_time);  // 然后顾客进程阻塞一段时间
  customer->service_end_time = Time();  // 记录服务结束的时间

  pthread_mutex_lock(&clerk->mutex);
  clerk->status = CLERK_WAITING;  // 结束服务后释放柜员
  pthread_mutex_unlock(&clerk->mutex);
  pthread_cond_broadcast(&clerk->cond);  // 并告知柜员结束服务

  printf("customer %d finish and leave bank at %d\n", customer->id, Time());
  pthread_exit(0);  // 最后结束顾客线程
}
```

代码与设计一致，并且在访问和修改会在多个线程中用到的变量时加了锁。

3.3 柜员线程

柜员的主循环过程如下：

1. 对 waiting customer 执行阻塞 P 操作，等待顾客进入银行并拿号
2. 结束信号量的阻塞。由信号量的性质可知，进入这一过程的柜员数一定不大于正在等待的顾客数量
3. 通过条件变量通知此时正在等待的顾客中拿到号最小的一个
4. 然后在条件变量上阻塞，等待顾客表示结束服务
5. 重新进入等待状态

代码实现如下：

```
void ClerkThread(Clerk *clerk) {
  printf("clerk %d start\n", clerk->id);
  while (true) {  // 主循环
    sem_wait(&waiting_customers);  // P 操作，阻塞到 waiting customer > 0
    pthread_mutex_lock(&waiting_customers_mutex);  // 开始叫号前先拿锁
    int min_uncalled_ticket = INF;
    int call_customer = -1;
    for (int i = 0; i != customer_num; ++i) {  // 检查列表中的所有顾客
      Customer *customer = &Customers[i];
      pthread_mutex_lock(&customer->mutex);
      if (customer->ticket >= 0   // 如果顾客拿到了号
        && customer->clerk_id == -1  // 并且还没有被任何柜员叫过
        && customer->ticket < min_uncalled_ticket) {  // 并且其拿到的号码尽可能小
```

```
14          min_uncalled_ticket = customer->ticket;
15          call_customer = i;
16        }
17        pthread_mutex_unlock(&customer->mutex);
18      }
19      Customer *customer = &Customers[call_customer];  // 决定叫这个顾客
20      printf("clerk %d is calling ticket %d\n", clerk->id, customer->ticket);
21
22      pthread_mutex_lock(&customer->mutex);
23      customer->clerk_id = clerk->id;  // 将该顾客的服务柜员设置为自己
24      pthread_cond_broadcast(&customer->cond);  // 并且通过该顾客的条件变量唤醒她
25      pthread_mutex_unlock(&customer->mutex);
26
27      pthread_mutex_unlock(&waiting_customers_mutex);  // 结束一次叫号过程后释放锁
28
29      do {
30        pthread_cond_wait(&clerk->cond, &clerk->mutex);  // 然后等待该顾客表示服务结束
31      } while (clerk->status != CLERK_WAITING);
32      pthread_mutex_unlock(&clerk->mutex);
33    }
34  }
```

同样实现了设计，并且对跨线程访问的变量加了锁。

4 实验结果与分析

4.1 编译运行

运行环境：Ubuntu 14.04.5
在代码目录 src 下执行 make（已经写好了 Makefile），然后执行

```
./main 3 20 in.txt out.txt
```

其中，3 表示银行柜员数量（当然其实多少都可以），20 表示等待运行结束的时间（一般要大于预计的总服务时长），in.txt 表示输入文件，out.txt 表示输出文件。

4.2 结果分析

使用题目中的样例进行测试，
输入 (in.txt)：

```
1 1 10
2 5 2
3 6 3
```

当柜员数量为 2 时得到正确的结果 (out.txt)：

```
1 1 1 11 0
2 5 5 7 1
3 6 7 10 1
```

从程序执行过程中打印的消息也可以看出，银行柜员系统正如我们所希望的那样工作。

```
maxujie@ubuntu:~/operating-system-project/project1$ ./main 2 15 in.txt out.txt
clerk num = 2, customer num = 3
clerk 0 start
clerk 1 start
customer 1 enter bank at 1
customer 1 get ticket 0
customer 1 is waiting
clerk 0 is calling ticket 0 at 1
customer 1 is called by clerk 0
customer 1 being served by clerk 0
customer 2 enter bank at 5
customer 2 get ticket 1
customer 2 is waiting
clerk 1 is calling ticket 1 at 5
customer 2 is called by clerk 1
customer 2 being served by clerk 1
customer 3 enter bank at 6
customer 3 get ticket 2
customer 3 is waiting
customer 2 finish and leave bank at 7
clerk 1 is calling ticket 2 at 7
customer 3 is called by clerk 1
customer 3 being served by clerk 1
customer 3 finish and leave bank at 10
customer 1 finish and leave bank at 11
```

当然实际上处理更加复杂的输入也是可以正确得到结果的。以下是 20 个顾客 4 个柜员的例子:
输入:

```
0 38 3
1 20 3
2 31 8
3 9 5
4 32 5
5 19 7
6 29 10
7 49 10
8 44 10
9 45 7
```

```
10 2 7
11 17 6
12 29 6
13 32 2
14 3 6
15 21 6
16 29 1
17 19 1
18 9 7
19 37 5
```

输出:

```
0 38 38 41 1
1 20 20 23 2
2 31 31 39 0
3 9 9 14 0
4 32 32 37 3
5 19 19 26 3
6 29 29 39 2
7 49 49 59 1
8 44 44 54 0
9 45 45 52 2
10 2 2 9 1
11 17 17 23 1
12 29 29 35 1
13 32 35 37 1
14 3 3 9 3
15 21 21 27 0
16 29 29 30 3
17 19 19 20 0
18 9 9 16 2
19 37 37 42 3
```

5 思考题

1 柜员人数与顾客人数对结果的影响?

为分析简单起见，先做这样的两个假定:

- 顾客进入银行的时间为相互独立的均匀分布
- 每个顾客需要的服务时长服从独立的指数分布

柜员数量不变、顾客数量增加：每个顾客的平均等待时间会随顾客数量的增加而线性增加。

顾客数量不变、柜员数量增加：如果顾客数量远大于柜员数量，则平均等待时间与柜员的数量成反比；当顾客与柜员数量相当时，由于绝大多数顾客都无需等待，所以柜员数量对顾客数量不再有影响。

2 实现互斥的方法、特点和效率?

方法	特点	效率
禁止中断	简单，将禁止中断的权力交给用户不安全	高
自旋锁	消耗 CPU 时间	浪费 CPU 资源
严格轮转	进程轮流进入临界区	进程速率不同时效率低
Peterson 算法	忙等待	高
TSL 指令	硬件实现、简单、优先级反转问题	高
信号量	正确性分析很困难	高
管程	一种抽象的编程语言概念，由编译器实现	?
消息传递	适用于不同机器通信	低

6 实验总结

这次实验全程用 C 来实现，使用封装层次比较低（相对于我此前写过的 C++11 thread 库和 Python multithreading 库而言）的 pthread 库，这加深了我对互斥锁、信号量、条件变量等概念的理解，也使我了解了（或者说是踩到了）多线程同步问题的一些坑。在以后用更高级的库编写多线程程序时，相信这次实验的经历会让我对代码背后解决的问题有更深刻的理解。