进程间同步/互斥实验报告

——银行柜员问题

无64 2016011088 徐泽来

2018 年 11 月 5 日

1 实验目的

1. 通过对进程间通信同步/互斥问题的编程实现，加深理解信号量和P, V操作的原理；
2. 对Linux涉及的几种互斥、同步机制有更进一步的了解；
3. 熟悉POSIX中定义的与互斥、同步有关的函数。

2 设计思路

2.1 整体框架

互斥信号量

在银行柜员问题的具体实现中，存在两个可能被多个线程同时访问的共享变量num_to_serve和num_to_call，因此在实现中使用了两个互斥信号量mutex_nts和mutex_ntc来对共享变量分别进行保护。下面对各共享变量及其互斥信号量进行简要说明。

- num_to_serve及mutex_nts
  num_to_serve为尚未安排服务的顾客个数，其初始值为client_num。它作为共享变量在柜员线程完成一轮服务后被读取，以判断是否需要开始下一轮的等待和服务：若num_to_serve不为0，则将num_to_serve减1，并使该柜员线程开始下一轮的等待和服务；否则结束该柜员线程。mutex_nts用于实现不同柜员线程访问num_to_serve时的互斥。

- num_to_call及mutex_ntc
  num_to_call为下一个将被叫到的顾客号，其初始值为0。它作为共享变量在柜员线程进行叫号前被读取，以决定当前应该被叫到的顾客号，并在读取后将num_to_call加1。mutex_nts用于实现不同柜员线程访问num_to_call时的互斥。

需要说明的是，num_to_serve和num_to_call的变化并不存在时间上的同时性：num_to_serve是在柜员线程完成一轮服务后进行访问和更新，而num_to_call是在柜员线程进行叫号前进行访问和更新。考虑如下情况：

柜员0在t=1时结束一轮服务，柜员1在t=2时结束一轮服务，最后一位顾客2在t=3时进入银行。

则该情况下num_to_serve和num_to_call的变化情况如下表所示

t	num_to_serve	num_to_call	说明
0.5	1	2	无
1.5	0	2	柜员0开始下一轮服务
2.5	0	2	柜员1结束线程
3.5	0	3	柜员0叫号顾客2

表 1: 某情况下num_to_serve和num_to_call的变化情况

可见num_to_serve和num_to_call的变化并不存在时间上的同时性，因此使用两个共享变量和两个互斥信号量是必要的。

同步信号量

在银行柜员问题中，我们需要实现两个同步：一个是顾客排队和柜员等待过程中的同步；另一个是顾客等待叫号和柜员叫号过程中的同步。在具体实现中使用了两个(组)同步信号量queue和sem_client[ ]分别实现上述两个同步，下面对各同步过程及其同步信号量进行简要说明。

- queue
  queue为当前等待叫号的顾客个数，其初始值为0。该信号量用于实现顾客排队和柜员等待过程中的同步。在柜员线程中，开始一轮服务时先对queue进行down()操作：若queue>0，即存在等待的顾客，则进行后续操作准备叫号；若queue=0，即没有顾客，则该柜员线程被阻塞，即进入等待。在顾客线程中，进入银行后先对queue进行up()操作：若queue=0，即存在等待的柜员，则唤醒一个等待柜员，并进行后续操作等待叫号；若queue>0，即已存在等待的顾客，则该顾客线程被阻塞，即进入等待。

- sem_client[ ]
  sem_client[ ]是一个信号量数组，给每个顾客均分配一个信号量，初始值均为0。该信号量用于实现顾客等待叫号和柜员叫号过程中的同步。在顾客线程中，当其结束排队后对自己的 sem_client进行down()操作：若sem_client=1，即已有柜员叫号，则开始服务；否则等待柜员叫号；在柜员线程中，当其结束等待并读取当前应该被叫到的顾客号后，对该顾客的sem_client进行up()操作，即进行叫号。

2.2 顾客线程

设顾客的编号为num，进入银行的时间为enter_time，需要服务的时间为serve_time；开始服务的时间为 begin_time，离开银行的时间为end_time，则顾客线程的设计思路如下：

1. 睡眠enter_time后进入银行，并读取实际进入银行的时间重新写入enter_time；

2. 对queue信号量进行up()操作：若存在等待的柜员则将其唤醒；否则进行排队；

3. 对自己的sem_client[num]信号量进行down()操作：若已有柜员叫号则记录开始服务的时间，否则等待柜员叫号；

4. 开始服务后睡眠serve_time时长，结束后记录离开银行的时间。

顾客线程的程序结构如下所示，具体实现可参考bank.c中的client()函数。

```
void *client(void *arg)
{
    sleep(enter_time);          /*睡眠enter_time后进入银行*/
    enter_time = gettime();     /*记录进入银行的时间*/

    up(&queue);                 /*顾客等待队列加1*/

    down(&sem_client[num]);     /*等待柜员叫号*/
    begin_time = gettime();     /*记录开始服务的时间*/

    sleep(serve_time);          /*接受serve_time时长的服务*/
    end_time = gettime();       /*记录离开银行的时间*/
}
```

2.3 柜员线程

设柜员的编号为num；用数组serving_teller[ ]记录服务各顾客的柜员编号，数组serve_time[ ]记录各顾客需要的服务时间，则柜员线程的设计思路如下：

1. 在mutex_ntc的保护下，将num_to_serve的值读入nts，并将num_to_serve的值减1；

2. 根据nts的值判断是否需要进入下一轮的等待与服务循环

   (a) 对queue信号量进行down()操作：若存在等待的顾客则进行后续操作准备叫号；否则进入等待；

   (b) 在mutex_nts的保护下，将num_to_call的值读入ntc，并将num_to_call的值加1；

   (c) 对sem_client[ntc]信号量进行up()操作：若对应顾客已在等待叫号，则将服务该顾客的柜员编号设为自己的编号，并开始服务；否则等待顾客回应叫号；

   (d) 在mutex_ntc的保护下，将num_to_serve的值读入nts，并将num_to_serve的值减1，返回循环条件的判断。

柜员线程的程序结构如下所示，具体实现可参考bank.c中的teller()函数。

```
void *teller(void *arg)
{
    lock(&mutex_nts);                    /*在mutex_nts的保护下*/
    nts = num_to_serve;                  /*读取num_to_serve的值*/
    num_to_serve--;
    unlock(&mutex_nts);

    while(nts>0)                         /*判断是否开始下一轮服务循环*/
    {
        down(&queue);                    /*查看是否存在等待服务的顾客*/

        lock(&mutex_ntc);                /*在mutex_ntc的保护下*/
        ntc = num_to_call;               /*读取num_to_call的值*/
        num_to_call++;
        unlock(&mutex_ntc);

        up(&sem_client[ntc]);            /*叫对应顾客的号*/
        seving_teller[ntc] = num;        /*记录服务该顾客的柜员编号*/
        sleep(serve_time[ntc]);          /*服务serve_time时长*/

        lock(&mutex_nts);                /*在mutex_nts的保护下*/
        nts = num_to_serve;              /*读取num_to_serve的值*/
        num_to_serve--;
        unlock(&mutex_nts);
    }
}
```

3 运行结果

使用说明

1. 实验平台为Linux系统；
2. 输入文件名为test.in；
3. 使用方式为在命令行输入./bank -n [N]，其中[N]为正整数。通过该参数可以设置柜员的个数，例如想要设置柜员个数为5，则应输入./bank -n 5。

测试文件

在测试中随机生成了10个顾客进入银行的时间和需要服务的时间，具体的测试文件内容如下

test.in

```
1  1  8
2  6  17
3  15  12
4  26  11
5  26  19
6  27  6
7  29  16
8  30  16
9  33  8
10  37  12
```

运行结果

在测试中分别设置柜员个数为1, 3, 5进行了测试，运行结果均符合实现要求，具体如下

./bank -n 1

No	enter	start	end	teller
1	1	1	9	0
2	6	9	26	0
3	15	26	38	0
4	26	38	49	0
5	26	49	68	0
6	27	68	74	0
7	29	74	90	0
8	30	90	106	0
9	33	106	114	0
10	37	114	126	0

./bank -n 3

No	enter	start	end	teller
1	1	1	9	1
2	6	6	23	0
3	15	15	27	2
4	26	26	37	1
5	26	26	45	0
6	27	27	33	2
7	29	33	49	2
8	30	37	53	1
9	33	45	53	0
10	37	49	61	2

./bank -n 5

No	enter	start	end	teller
1	1	1	9	1
2	6	6	23	4
3	15	15	27	0
4	26	26	37	2
5	26	26	45	3
6	27	27	33	1
7	29	29	45	4
8	30	30	46	0
9	33	33	41	1
10	37	37	49	2

4 思考题解答

思考题1 柜员人数和顾客人数对结果分别有什么影响？

解答： 设柜员人数为n，顾客人数为m，顾客的进入时间为t_{enter}，服务时间为t_{serve}，开始时间为t_{start}，离开时间为t_{end}，则有

$$t_{enter} \leq t_{start} < t_{end}，且 t_{end} - t_{begin} = t_{serve}$$

一般而言，n越大，$t_{start} - t_{enter}$越小，即顾客等待的时间越短，t_{end}也越小；m越大，序号靠后顾客的$t_{start} - t_{enter}$越大，即等待时间越长，t_{end}也越大。下面将分不同情况具体讨论。

- 若$n \geq m$，即柜员多于顾客。在这种情况下，每个顾客进入银行就可以得到服务，服务结束即可离开，故对每一个顾客均有

  $$t_{start} = t_{enter}, t_{end} = t_{start} + t_{serve}$$

  服务各顾客的柜员编号是不确定的，由操作系统的调度机制所决定。

- 若$n < m$，即顾客多于柜员。在这种情况下，前n个顾客进入银行就可以得到服务，服务结束即可离开，即有

  $$t_{start} = t_{enter}, t_{end} = t_{start} + t_{serve}$$

  而对于后m-n个顾客，他们进入银行后通常需要进行等待。一般而言，顾客的t_{enter}越大，其t_{start}也就越大，但这并不意味着$t_{enter} - t_{start}$越大。服务各顾客的柜员编号是不确定的，由操作系统的调度机制所决定。

思考题2 实现互斥的方法有哪些？各自有什么特点？效率如何？

解答： 实现互斥的各方法，特点，和效率总结如下

方法	优点	缺点	效率
禁止中断	简单	- 系统可靠性较差 - 不适用于多处理器	低（忙等待）
严格轮转法	简单	临界区外的进程可能阻塞其他进程	低（忙等待）
Peterson算法	正确	效率较低	低（忙等待）
硬件指令方法	- 简单，正确 - 适用于任意数目的进程 - 进程可存在多个临界区	效率较低	低（忙等待）
信号量	正确，效率较高	- 同步操作分散 - 不利于修改和维护	高（阻塞态）
管程	- 模块化 - 抽象数据类型 - 信息封装	- 需编译器支持 - C等多数程序设计语言不支持管程	高（阻塞态）

表 2: 实现互斥方法的特点与效率总结

5 实验总结

5.1 调试问题

问题1 在最初设计算法时没有考虑到顾客被叫号的先后顺序，因此直接将问题类比为了生产者-消费者问题，仅用一个信号量对顾客和柜员进行同步，导致出现后进入的顾客先被叫号的问题。

解决方案 对每一个顾客使用一个信号量实现顾客和柜员之间的同步，同时使用一个共享变量num_to_call记录当前叫到的顾客号。每个柜员在叫号前先互斥地读取num_to_call的值，并将其值加1，然后将对应顾客的信号量进行post()操作，从而实现了有序的叫号。

问题2 在调用pthread_create()创建线程时第4个参数传递出错，即传递给线程运行函数的参数列表指针出错。

解决方案 经分析后发现，这是由于在循环创建线程的过程中每次都使用了同一个变量的地址作为第4个参数进行传递，而在循环过程中该变量的值在不断发生变化，导致线程开始运行时该地址的值已经不再是原来想要传递的值了。通过使用多个变量的不同地址进行参数传递能够解决上述问题。

问题3 在程序中使用了sleep()函数模拟顾客进入银行前的状态，以及柜员对顾客进行服务时的状态。在将1个时间单位设为1s时可以获得正确的运行结果，但将1个时间单位设为1us（即使用

usleep()函数）时就会出现错误的运行结果。

解决方案 注意到线程在调用sleep()或usleep()函数睡眠指定时间后，并不一定能立即被执行，而是需要等待操作系统进行调度，因此线程实际被阻塞的时间一般会略大于设定的时间。在1s的量级上，操作系统调度线程的时间开销几乎可以忽略不计；而在1us的量级上，操作系统调度线程的时间开销就对程序的定时产生了较大影响，因此会出现错误的运行结果。通过将1个时间单位设为10ms可以在保证程序效率的情况下获得正确的运行结果。

5.2 收获体会

本次实验是我第一次尝试多线程编程，也是我第一次在Linux下进行C程序的编写和调试，虽然实验的难度不大，但给我的收获还是比较丰富。

在多线程编程方面，我通过这次实验熟悉了POSIX中线程的创建、等待、结束等操作，同时也掌握了使用mutex和semaphore的相关函数实现进程间的互斥和同步，并通过实际的编程和调试对程序的并发执行有了更加深入的理解。

在Linux编程方面，我通过这次实验熟悉了Linux终端的基本命令，学会了在Linux下进行精确计时的方法，同时也掌握了在C程序中调用shell命令并获得返回参数值的实现方法。