

Homework 7 — October 14

Lecturer: Jiang Dejun

Completed by: Zhang Jiawei

7.1

两个线程在进入对信号量 S1 的操作之前,各个变量的值分别为: $y=3, x=2, z=2, S1=S2=0$ 。

由于 S1 的值为 0,故 P(S1) 操作会被阻塞,而 V(S1) 操作会使得 S1 的值加 1,此时具有多种可能的执行顺序:

- 线程 T2 直接执行完毕,更新 $x=5, S2=1, z=7$,然后线程 T1 执行,更新 $z=4, y=7$,最终 $x=5, y=7, z=4$ 。
- 线程 T2 更新 $x=5, S2=1$,然后线程 T1 执行,更新 $z=4, y=7$,然后线程 T2 执行,更新 $z=9$,最终 $x=5, y=7, z=9$ 。
- 线程 T2 更新 $x=5$,然后线程 T1 执行,更新 $z=4$,然后线程 T2 执行完毕,更新 $S2=1, z=9$,然后线程 T1 执行,更新 $y=12$,最终 $x=5, y=12, z=9$ 。
- 线程 T2 更新 $x=5$,然后线程 T1 执行,更新 $z=4$,然后线程 T2 执行,更新 $S2=1$,然后线程 T1 执行,更新 $y=7$,然后线程 T2 执行,更新 $z=9$,最终 $x=5, y=7, z=9$ 。
- 线程 T1 更新 $z=4$,然后线程 T2 执行完毕,更新 $x=5, S2=1, z=9$,然后线程 T1 执行,更新 $y=12$,最终 $x=5, y=12, z=9$ 。
- 线程 T1 更新 $z=4$,然后线程 T2 执行,更新 $x=5, S2=1$,然后线程 T1 执行,更新 $y=7$,然后线程 T2 执行,更新 $z=9$,最终 $x=5, y=7, z=9$ 。

7.2

两种方法的区别在于,信号量的 P 操作和 V 操作是不是在互斥锁内。显然,刚开始缓冲区为空时,方法二中倘若 Remove 函数拿到互斥锁,在 `fullBuffers->P()` 会陷入忙等,无法释放互斥锁,导致 Deposit 函数拿不到互斥锁,无法向缓冲区中添加数据,最终程序无法执行下去。所以方法二是错误的,应该使用方法一。

7.3

此题仍是生产者-消费者问题, n 个柜员可以看做大小为 n 的缓冲区,缓冲区满即为 n 个柜员都在工作,缓冲区空即为 n 个柜员都在空闲。顾客取号操作 `Customer_Service` 可以看做生产者,柜员服务操作 `Teller_Service` 可以看做消费者。信号量的初始值为 $S1=0, S2=n$,表示初始时没有顾客取号, n 个柜员都在空闲。

```
1 semaphore S1 = 0, S2 = n;
2 Customer_Service(){
3     P(S2);
4     P(mutex);
5     // 顾客取号
6     V(mutex);
7     V(S1);
```

```
8  }
9  Teller_Service(){
10     P(S1);
11     P(mutex);
12     // 柜员服务
13     V(mutex);
14     V(S2);
15 }
```

7.4

所写的代码如下：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5  #include <time.h>
6
7  #define NUM_THREADS 8
8  #define NUM_INTS 16
9
10 int final_sum = 0;
11
12 typedef struct {
13     int data[NUM_INTS];
14     int index;
15     int count;
16     pthread_mutex_t mutex;
17     pthread_cond_t cond;
18 } Monitor;
19
20 Monitor monitor;
21
22 void* thread_func(void* arg) {
23     int id = *(int*)arg;
24
25     // 初始化数据
26     int a = rand() % 100;
27     int b = rand() % 100;
28     monitor.data[monitor.index++] = a;
29     monitor.data[monitor.index++] = b;
30
31     // 模拟不均衡性
32     usleep((rand() % 10 + 1) * 1000);
```

```
33
34     int sum = a + b;
35     printf("Thread %d: %d + %d = %d\n", id, a, b, sum);
36
37     pthread_mutex_lock(&monitor.mutex);
38     if (monitor.count > NUM_INTS)
39         pthread_cond_wait(&monitor.cond, &monitor.mutex);
40     final_sum += sum;
41     monitor.count++;
42     pthread_mutex_unlock(&monitor.mutex);
43     return NULL;
44 }
45
46 int main() {
47     srand(time(NULL));
48
49     monitor.index = 0;
50     monitor.count = 0;
51     pthread_mutex_init(&monitor.mutex, NULL);
52     pthread_cond_init(&monitor.cond, NULL);
53
54     pthread_t threads[NUM_THREADS];
55     int thread_ids[NUM_THREADS];
56
57     // 创建线程
58     for (int i = 0; i < NUM_THREADS; i++) {
59         thread_ids[i] = i;
60         pthread_create(&threads[i], NULL, thread_func, &thread_ids[i]);
61     }
62
63     // 等待所有线程完成
64     for (int i = 0; i < NUM_THREADS; i++) {
65         pthread_join(threads[i], NULL);
66     }
67
68     // 打印结果
69     printf("Final sum: %d\n", final_sum);
70
71     // 销毁互斥锁和条件变量
72     pthread_mutex_destroy(&monitor.mutex);
73     pthread_cond_destroy(&monitor.cond);
74     return 0;
75 }
```

输出结果如下：

```
Thread 4: 50 + 73 = 123
Thread 4: final_sum = 123
Thread 0: 74 + 77 = 151
Thread 0: final_sum = 274
Thread 6: 59 + 79 = 138
Thread 6: final_sum = 412
Thread 1: 74 + 11 = 85
Thread 1: final_sum = 497
Thread 2: 15 + 12 = 27
Thread 2: final_sum = 524
Thread 5: 43 + 82 = 125
Thread 5: final_sum = 649
Thread 3: 77 + 67 = 144
Thread 3: final_sum = 793
Thread 7: 68 + 85 = 153
Thread 7: final_sum = 946
Final sum: 946

Thread 4: 95 + 15 = 110
Thread 4: final_sum = 110
Thread 2: 59 + 90 = 149
Thread 2: final_sum = 259
Thread 3: 39 + 7 = 46
Thread 3: final_sum = 305
Thread 5: 46 + 62 = 108
Thread 5: final_sum = 413
Thread 0: 81 + 26 = 107
Thread 0: final_sum = 520
Thread 1: 32 + 94 = 126
Thread 1: final_sum = 646
Thread 6: 99 + 91 = 190
Thread 6: final_sum = 836
Thread 7: 90 + 47 = 137
Thread 7: final_sum = 973
Final sum: 973
```

图 7.1. 运行结果