Operating System: Semaphore & Monitor **Fall 2024**

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。

```
semaphore S1 = 0, S2 = n;
Customer_Service(){
  P(S2);
  P(mutex);
  // 顾客取号
  V(mutex);
  V(S1);
```

```
}
Teller_Service(){
  P(S1);
  P(mutex);
  // 柜员服务
  V(mutex);
  V(S2);
}
```

7.4

所写的代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

#define NUM_THREADS 8
#define NUM_INTS 16

int final_sum = 0;

typedef struct {
   int data[NUM_INTS];
   int index;
   int count;
   pthread_mutex_t mutex;
   pthread_cond_t cond;
} Monitor;

Monitor monitor;

void* thread_func(void* arg) {
   int id = *(int*)arg;

   // 初始化数据
   int a = rand() % 100;
   int b = rand() % 100;
   monitor.data[monitor.index++] = a;
   monitor.data[monitor.index++] = b;

   // 模拟不均衡性
   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：

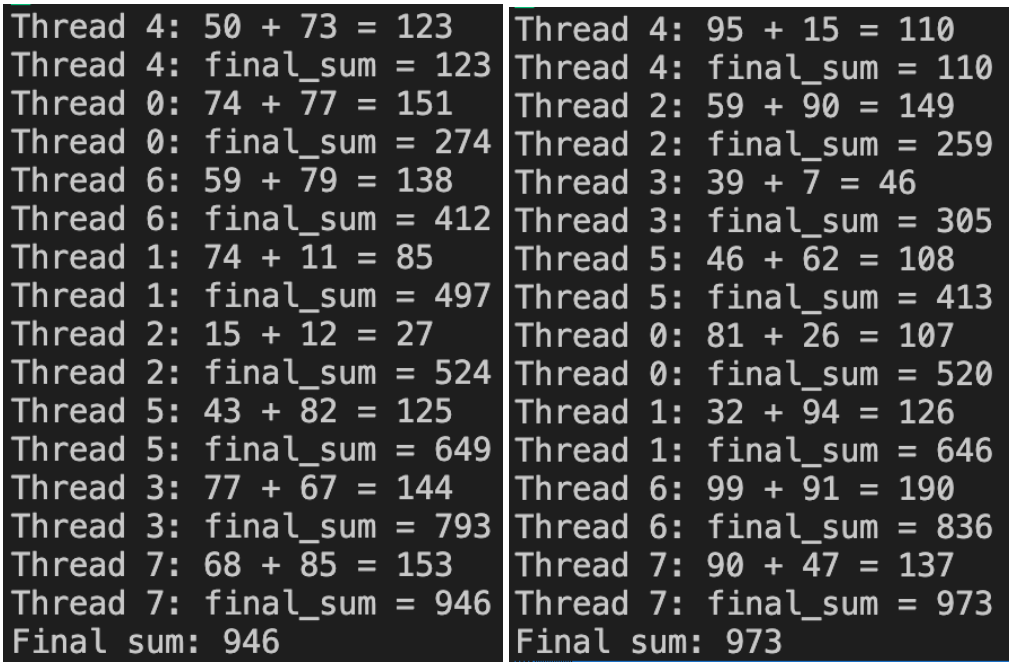

图 7.1. 运行结果