

# 作业 8

贾城昊

2021K8009929010

**8.1 银行有  $n$  个柜员,每个顾客进入银行后先取一个号,并且等着叫号,当一个柜员空闲后,就叫下一个号.**

**请使用 PV 操作分别实现:**

**//顾客取号操作 Customer\_Service**

**//柜员服务操作 Teller\_Service**

调研:

本次作业代码种会使用 **sem\_wait** 和 **sem\_post** 函数, **sem\_wait** 和 **sem\_post** 是信号量的两个主要操作, 用于实现同步和互斥。它们通常用于多线程和多进程编程中来控制共享资源的访问。

**1. sem\_wait (等待) :**

- **sem\_wait** 函数用于请求获得一个信号量。如果信号量的计数值大于 0, 表示资源可用, **sem\_wait** 将减少信号量的计数, 并立即返回, 允许线程或进程访问共享资源。
- 如果信号量的计数值为 0, 表示资源不可用, **sem\_wait** 将阻塞线程或进程, 直到信号量的计数值变为大于 0。这使得线程或进程等待其他线程或进程释放资源。
- 一旦成功获得信号量, **sem\_wait** 将继续执行, 并允许访问共享资源。如果多个线程或进程同时尝试获得信号量, 只有一个会成功, 其余将被阻塞。

**2. sem\_post (释放) :**

- **sem\_post** 函数用于释放信号量, 增加其计数值。这表示共享资源现在可用于其他线程或进程。
- 当一个线程或进程完成了对共享资源的使用, 应该调用 **sem\_post** 来释放信号量, 以允许其他线程或进程继续访问资源。
- 如果有线程或进程在等待信号量, 它们中的一个将被唤醒, 以获得对资源的访问。

所以 **sem\_wait** 和 **sem\_post** 分别相当于 P 操作和 V 操作

### 1. C 程序如下:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define NUM_CUSTOMERS 10 // 顾客数量
#define NUM_TELLERS 3    // 柜员数量

sem_t customers; // 顾客信号量, 初始值为 0
sem_t tellers;   // 柜员信号量, 初始值为 NUM_TELLERS

void *Customer_Service(void *customer_number) {
    // 获取顾客号码
    int customer_id = *((int *)customer_number);

    // 顾客取号操作
    printf("Customer %d takes a number and waits.\n", customer_id);
    sem_post(&customers); // 通知柜员有新顾客

    // 等待柜员服务
    sem_wait(&tellers);
    printf("Customer %d is being served.\n", customer_id);
    // 顾客被柜员服务

    pthread_exit(NULL);
}

void *Teller_Service(void *teller_id) {
    // 获取柜员号码
    int teller_number = *((int *)teller_id);

    while (1) {
        // 等待顾客
        sem_wait(&customers);
```

```

        // 调用顾客的号码
        printf("Teller %d calls the next customer.\n", teller_number);

        // 为顾客服务
        //...

        // 通知柜员空闲
        sem_post(&tellers);
    }
}

int main() {
    pthread_t customers_threads[NUM_CUSTOMERS];
    pthread_t tellers_threads[NUM_TELLERS];

    sem_init(&customers, 0, 0);
    sem_init(&tellers, 0, NUM_TELLERS);

    int customer_numbers[NUM_CUSTOMERS];
    int teller_numbers[NUM_TELLERS];

    // 创建顾客线程
    for (int i = 0; i < NUM_CUSTOMERS; i++) {
        customer_numbers[i] = i;
        pthread_create(&customers_threads[i], NULL, Customer_Service, &customer_numbers[i]);
    }

    // 创建柜员线程
    for (int i = 0; i < NUM_TELLERS; i++) {
        teller_numbers[i] = i;
        pthread_create(&tellers_threads[i], NULL, Teller_Service, &teller_numbers[i]);
    }

    // 等待顾客线程和柜员线程完成
    for (int i = 0; i < NUM_CUSTOMERS; i++) {
        pthread_join(customers_threads[i], NULL);
    }
    for (int i = 0; i < NUM_TELLERS; i++) {
        pthread_join(tellers_threads[i], NULL);
    }
}

```

```
    }

    sem_destroy(&customers);
    sem_destroy(&tellers);

    return 0;
}
```

## 2. 运行结果:

```
sai@Computer:~/workspace/OS$ ./hw8_1
Customer 0 takes a number and waits.
Customer 0 is being served.
Customer 1 takes a number and waits.
Customer 1 is being served.
Customer 2 takes a number and waits.
Customer 2 is being served.
Customer 3 takes a number and waits.
Customer 4 takes a number and waits.
Customer 5 takes a number and waits.
Customer 6 takes a number and waits.
Customer 7 takes a number and waits.
Customer 8 takes a number and waits.
Customer 9 takes a number and waits.
Teller 0 calls the next customer.
Customer 3 is being served.
Teller 1 calls the next customer.
Customer 4 is being served.
Teller 2 calls the next customer.
Customer 5 is being served.
Teller 0 calls the next customer.
Customer 6 is being served.
Teller 1 calls the next customer.
Customer 7 is being served.
Teller 1 calls the next customer.
Customer 8 is being served.
Teller 0 calls the next customer.
Teller 0 calls the next customer.
Teller 0 calls the next customer.
Teller 0 calls the next customer.
Customer 9 is being served.
```

可以看出，结果是正确的，满足题目要求。

**8.2 多个线程的规约(Reduce)操作是把每个线程的结果按照某种运算(符合交换律和结合律) 两两合并直到得到最终结果的过程。**

**试设计管程 monitor 实现一个 8 线程规约的过程，随机初始化 16 个整数，每个线程通过调用 monitor.getTask 获得 2 个数，相加后，返回一个数 monitor.putResult ，然后再 getTask() 直到全部完成退出，最后打印归约过程和结果。**

要求: 为了模拟不均衡性，每个加法操作要加上随机的时间扰动，变动区间 1~10ms。

提示: 使用 pthread\_系列的 cond\_wait, cond\_signal, mutex 实现管程；使用 rand() 函数产生随机数，和随机执行时间。

1. C 程序如下:

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

#define THREADS 8
#define ARRAY_SIZE 16

// Monitor structure
typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t task_available;
    pthread_cond_t result_available;
    int task_index;
    int result_index;
    int array[ARRAY_SIZE];
    int result[ARRAY_SIZE];
    int sum;
} Monitor;
```

Monitor monitor;

```
void monitor_init(Monitor *m) {
    pthread_mutex_init(&m->mutex, NULL);
    pthread_cond_init(&m->task_available, NULL);
    pthread_cond_init(&m->result_available, NULL);
    m->task_index = ARRAY_SIZE;
    m->result_index = 0;
    m->sum = 0;
    for (int i = 0; i < ARRAY_SIZE; i++) {
        m->array[i] = rand() % 100; // Initialize the array with random integers
    }
}
```

```
int get_random_delay() {
    return (rand() % 10 + 1);
}
```

```
void monitor_getTask(int *task1, int *task2) {
    pthread_mutex_lock(&monitor.mutex);
    while (monitor.task_index + monitor.result_index < 2) {
        if(monitor.task_index == 0 && monitor.result_index == 1){
            pthread_mutex_unlock(&monitor.mutex);
            *task1 = *task2 = -1;
            return;
        }
        pthread_cond_wait(&monitor.task_available, &monitor.mutex);
    }
}
```

```
if(monitor.task_index >= 2){
    *task1 = monitor.array[monitor.task_index - 1];
    *task2 = monitor.array[monitor.task_index - 2];
    monitor.task_index -= 2;
}else if (monitor.result_index >= 2){
    *task1 = monitor.result[monitor.result_index - 1];
    *task2 = monitor.result[monitor.result_index - 2];
    monitor.result_index -= 2;
}else{
    *task1 = monitor.array[monitor.task_index - 1];
    *task2 = monitor.result[monitor.result_index - 1];
    monitor.task_index -= 1;
    monitor.result_index -= 1;
}
```

```

    }

    pthread_mutex_unlock(&monitor.mutex);
}

void monitor_putResult(int local_sum) {
    pthread_mutex_lock(&monitor.mutex);
    monitor.sum += local_sum;
    monitor.result[monitor.result_index++] = local_sum;
    pthread_cond_broadcast(&monitor.result_available);
    pthread_mutex_unlock(&monitor.mutex);
}

void *thread_routine(void *arg) {
    long thread_id = (long)arg;
    int local_sum = 0;
    int task1, task2;

    while (1) {
        monitor_getTask(&task1, &task2);
        if(task1 == -1 && task2 == -1){
            break;
        }
        usleep(get_random_delay() * 1000); // Simulate random delay

        local_sum = task1 + task2;
        printf("Thread %ld added %d and %d to get %d\n", thread_id, task1, task2, local_sum);

        monitor_putResult(local_sum);
    }

    pthread_exit(NULL);
}

void monitor_destroy(Monitor monitor) {
    pthread_mutex_destroy(&monitor.mutex);
    pthread_cond_destroy(&monitor.task_available);
    pthread_cond_destroy(&monitor.result_available);
}

int main() {
    srand(time(NULL));

```

```
pthread_t threads[THREADS];

monitor_init(&monitor);

for (long i = 0; i < THREADS; i++) {
    pthread_create(&threads[i], NULL, thread_routine, (void *)i);
}

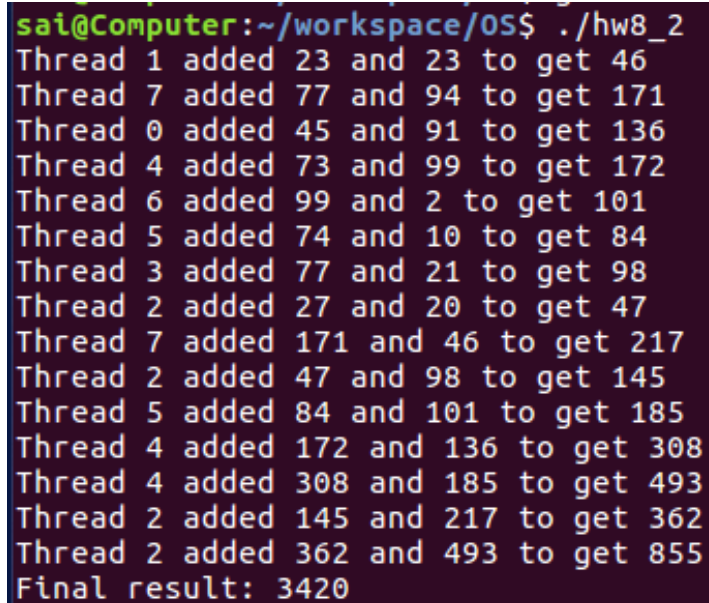
for (int i = 0; i < THREADS; i++) {
    pthread_join(threads[i], NULL);
}

monitor_destroy(monitor);

printf("Final result: %d\n", monitor.sum);

return 0;
}
```

## 2. 运行结果:



```
sai@Computer:~/workspace/OS$ ./hw8_2
Thread 1 added 23 and 23 to get 46
Thread 7 added 77 and 94 to get 171
Thread 0 added 45 and 91 to get 136
Thread 4 added 73 and 99 to get 172
Thread 6 added 99 and 2 to get 101
Thread 5 added 74 and 10 to get 84
Thread 3 added 77 and 21 to get 98
Thread 2 added 27 and 20 to get 47
Thread 7 added 171 and 46 to get 217
Thread 2 added 47 and 98 to get 145
Thread 5 added 84 and 101 to get 185
Thread 4 added 172 and 136 to get 308
Thread 4 added 308 and 185 to get 493
Thread 2 added 145 and 217 to get 362
Thread 2 added 362 and 493 to get 855
Final result: 3420
```

可以看到其产生了正确的结果