操作系统 作业 7

宋婉婷 2022K8009929009

7.1

设有两个优先级相同的进程 T1,T2 如下。令信号量 S1,S2 的初值为 0，已知 z=2，试求 T1,T2 并发运行结束后 x,y,z 的值。请分析所有可能的情况，并给出结果与相应执行顺序。

```
线程 T1       线程 T2
y:=1;          x:=1;
y:=y+2;        x:=x+4;
V(S1);         P(S1);
z:=y+3;<a>     x:=x+y;<b>
P(S2);         V(S2);
y:=z+y;<c>     z:=x+z;<d>
```

答： 前两行顺序不影响，执行后为 x=5,y=3,z=2。对剩下赋值语句编号，因为 P(S1) 晚于 V(S1)，P(S2) 晚于 V(S2)，且 a>c，b>d，则剩下语句顺序可能情况有：

1. a-b-c-d 得 x=8 y=9 z=14
2. a-b-d-c 得 x=8 y=17 z=14
3. b-d-a-c 得 x=8 y=9 z=6
4. b-a-c-d 得 x=8 y=9 z=14
5. b-a-d-c 得 x=8 y=17 z=14

综上，总共有三种可能结果：$x=8,y=9,z=14$ 或 $x=8,y=17,z=14$ 或 $x=8,y=9,z=6$

7.2

在生产者 - 消费者问题中，假设缓冲区大小为 5，生产者和消费者在写入和读取数据时都会更新写入 / 读取的位置 offset。现有以下两种基于信号量的实现方法：

方法一：

```
Class BoundedBuffer {
    mutex = new Semaphore(1);
    fullBuffers = new Semaphore(0);
    emptyBuffers = new Semaphore(n);
}
BoundedBuffer::Deposit(c) {
    emptyBuffers->P();
    mutex->P();
    Add c to the buffer;
    offset++;
    mutex->V();
    fullBuffers->V();
}
BoundedBuffer::Remove(c) {
    fullBuffers->P();
    mutex->P();
```

```
    Remove c from buffer;
    offset--;
    mutex->V();
    emptyBuffers->V();
}
```

方法二：

```
Class BoundedBuffer {
    mutex = new Semaphore(1);
    fullBuffers = new Semaphore(0);
    emptyBuffers = new Semaphore(n);
}
BoundedBuffer::Deposit(c) {
    mutex->P();
    emptyBuffers->P();
    Add c to the buffer;
    offset++;
    fullBuffers->V();
    mutex->V();
}
BoundedBuffer::Remove(c) {
    mutex->P();
    fullBuffers->P();
    Remove c from buffer;
    offset--;
    emptyBuffers->V();
    mutex->V();
}
```

请对比分析上述方法一和方法二，哪种方法能让生产者、消费者进程正常运行，并说明分析原因。

答： 两个方法的区别是对 mutex 的操作位置不同，方法一将缓冲区加锁放在获取 emptyBuffers 和 fullBuffers 之后，而方法二放在之前。方法一能正常运行，而方法二会造成死锁，因为若 emptyBuffers 为空，Deposit() 会在持有锁的情况下被阻塞，而 Remove() 也无法获取锁来修改 emptyBuffers。

7.3

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

答： 伪代码实现如下，使用 mutex 保护共享队列的读写。

```
mutex = new Semaphore(1); // 互斥锁，保护共享队列访问
availTeller = new Semaphore(n); // 空闲柜员
waitingCustomer = new Semaphore(0); // 等待服务顾客
queue = new Queue(MAX_NUM);
void customerService(){
    int my_number = getNumber();
    mutex->P();
    queue.push(my_number);
```

```
    mutex->V();
    waitingCustomer->V();
}
void tellerService(){
    while(ture){
        availTeller->P();
        waitingCustomer->P();
        mutex->P();
        int cus_number = queue.pop();
        mutex->V();
        do_service(cus_number);
        availTeller->V();
    }
}
```

7.4

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 5~10ms。

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。

答： 程序如下：

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

#define NUM_THREADS 8
#define NUM_ELEMENTS 16
#define MIN_DELAY 5000  // 5ms
#define MAX_DELAY 10000 // 10ms

typedef struct {
    int *data;
    int *results;
    int data_index
    int result_count;
    int is_done;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} ReductionMonitor;

void monitor_init(ReductionMonitor *monitor, int *data) {
    monitor->data = data;
    monitor->results = (int *)malloc(NUM_THREADS * sizeof(int));
    monitor->data_index = 0;
```

```
    monitor->result_count = 0;
    monitor->is_done = 0;
    pthread_mutex_init(&monitor->mutex, NULL);
    pthread_cond_init(&monitor->cond, NULL);
}

void monitor_destroy(ReductionMonitor *monitor) {
    free(monitor->results);
    pthread_mutex_destroy(&monitor->mutex);
    pthread_cond_destroy(&monitor->cond);
}

// 获取原数据
void monitor_get_task(ReductionMonitor *monitor, int *a, int *b) {
    pthread_mutex_lock(&monitor->mutex);
    while (monitor->data_index >= NUM_ELEMENTS && monitor->result_count < NUM_THREADS) {
        pthread_cond_wait(&monitor->cond, &monitor->mutex);
    }
    if (monitor->data_index < NUM_ELEMENTS) {
        *a = monitor->data[monitor->data_index++];
        *b = monitor->data[monitor->data_index++];
    } else {
        *a = -1;
        *b = -1;
    }
    pthread_mutex_unlock(&monitor->mutex);
}

// 提交结果
void monitor_put_result(ReductionMonitor *monitor, int result) {
    pthread_mutex_lock(&monitor->mutex);
    monitor->results[monitor->result_count++] = result;
    if (monitor->result_count == NUM_THREADS || monitor->data_index >= NUM_ELEMENTS) {
        monitor->is_done = 1;
        pthread_cond_broadcast(&monitor->cond);
    }
    pthread_mutex_unlock(&monitor->mutex);
}

// 检查是否所有任务完成
int monitor_is_done(ReductionMonitor *monitor) {
    pthread_mutex_lock(&monitor->mutex);
    int done = monitor->is_done;
    pthread_mutex_unlock(&monitor->mutex);
    return done;
}

// 线程函数
void *thread_reduce(void *arg) {
    ReductionMonitor *monitor = (ReductionMonitor *)arg;
    int a, b;
    while (1) {
        monitor_get_task(monitor, &a, &b);
        if (a == -1 && b == -1) {
```

```
            break;
        }
        // 模拟不均衡的加法操作
        int delay = MIN_DELAY + rand() % (MAX_DELAY - MIN_DELAY + 1);
        usleep(delay);
        int sum = a + b;
        printf("Thread cal: %d + %d = %d (delay: %d us)\n",
               a, b, sum, delay);
        monitor_put_result(monitor, sum);
    }
    return NULL;
}

int main() {
    srand(time(NULL));
    int data[NUM_ELEMENTS];
    printf("data:");
    for (int i = 0; i < NUM_ELEMENTS; i++) {
        data[i] = rand() % 100;
        printf("%d", data[i]);
    }
    printf("\n");

    ReductionMonitor monitor;
    monitor_init(&monitor, data);
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_reduce, &monitor);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    int final_result = 0;
    for (int i = 0; i < monitor.result_count; i++) {
        final_result += monitor.results[i];
    }
    printf("Final reduction result: %d\n", final_result);
    monitor_destroy(&monitor);
    return 0;
}
```

运行结果如下：

```
# ./monitor
data: 46 45 57 71 85 68 90 10 44 29 34 44 45 18 59 15
Thread : 57 + 71 = 128 (delay: 5158 us)
Thread : 44 + 29 = 73 (delay: 5371 us)
Thread : 90 + 10 = 100 (delay: 6431 us)
Thread : 46 + 45 = 91 (delay: 6892 us)
Thread : 34 + 44 = 78 (delay: 7175 us)
Thread : 59 + 15 = 74 (delay: 8597 us)
Thread : 45 + 18 = 63 (delay: 8835 us)
Thread : 85 + 68 = 153 (delay: 9894 us)
Final reduction result: 760
```