操作系统 第七次作业

姓名：朱首赫

学号：2023K8009906029

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，试问 T1，T2 并发运行结束后 x=?y=?z=?

线程 T1	**线程 T2**
A1: $y := 1$	B1: $x := 1$
A2: $y := y + 2$	B2: $x := x + 4$
A3: V(S1)	B3: P(S1)
A4: $z := y + 3$	B4: $x := x + y$
A5: P(S2)	B5: V(S2)
A6: $y := z + y$	B6: $z := x + z$

注: 请分析所有可能的情况，并给出结果与相应执行顺序。

解：可以分析得到：

1. A1, A2 和 B1, B2 可以任意并发执行，此时状态为 y=3, x=5, z=2。

2. B3 后的操作必须在 A3 之后执行，也即 B4 一定是在 A1-A2 后执行；A5 后的操作必须在 B5 之后执行，也即 B4 一定在 A6 之前执行，所以 B4: x=x+y=5+3=8。

3. 最终，**可以确定运行结果的指令链为 A1-A3 和 B1-B5**，可以确定运行状态为 $x = 8, y = 3, z = 2, S_1 = 0, S_2 = 1$. 只需要分析 B6 插入 A4-A6 不同位置的运行结果。

情况一 B6 先于 A4：B6: z=x+z=8+2=10; A4: z=y+3=3+3=6; A6: y=z+y=6+3=9。此情况下最终 x=8, y=9, z=6

情况二 B6 在 A4 与 A5 之间：A4: z=y+3=3+3=6; B6: z=x+z=8+6=14; A6: y=z+y=14+3=17。此情况下最终 x=8, y=17, z=14

情况三 B6 在 A5 与 A6 之间：结果同上 x=8, y=17, z=14

情况四 B6 在 A6 之后：A4: z=y+3=3+3=6; A6: y=z+y=6+3=9; B6: z=x+z=8+6=14。此情况下最终 x=8, y=9, z=14

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，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。

Listing 1: 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();
}
```

Listing 2: 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();
}
```

解: 方法一可以正常运行, 因为等待信号量操作在获取锁操作之前, 所以线程在等待资源时不会持有互斥锁。方法二会造成死锁: 假设缓冲区满 (fullBuffers=5, emptyBuffers=0), 一个生产者调用 Deposit, 执行 mutex->P() 获取了锁, 然后执行 emptyBuffers->P() 时由于 emptyBuffers 为 0, 生产者阻塞, 导致生产者等待消费者发出 emptyBuffers 信号, 而消费者又等待生产者释放锁, 造成了死锁。

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

解：

Listing 3: 银行问题实现伪代码

```
Class Bank {
    mutex = new Semaphore(1);
    Customers = new Semaphore(0); // 正在排队的顾客数量
    Tellers = new Semaphore(n); // 空闲柜员数量
    int next_ticket = 0; //下一个取号号码
    int now_serving = 0; //下一个呼叫号码
}
Bank::Customer_Service(c) {
//  顾客取号
    mutex->P();
    my_ticket = get_a_ticket(next_ticket); // 取得号码为next_ticket
    next_ticket++;
    mutex->V();

    Customers->V() // 通知柜员有一个顾客在等
// 等待服务
    while(my_ticket != now_serving); //等叫号叫到自己（这里的忙等其实可以通过为每个号码创
        建一个信号量来避免）
    Tellers->P(); // 等待一个空闲柜员
    mutex->P();
    get_service(); // 接受服务
    now_serving++; // 结束服务
    mutex->V();

}
Bank::Teller_Service(c) {
    Customers->P() // 等待一个取号后的顾客
    mutex->P();
    call_a_customer(now_serving); // 呼叫号码为now_serving的顾客
    mutex->V();
    Tellers->V(); // 释放空闲柜员
}
```

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。

解： 程序源码和运行结果如下。

Listing 4: 8 线程规约的管程实现

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h> // usleep
#include <time.h>   // time() and srand()

#define NUM_THREADS 8
#define INITIAL_SIZE 16

// 用于记录归约过程
typedef struct {
    int value;
    int source_a;
    int source_b;
    int level;
} Result_t;

// 归约管程结构体
typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t task_ready; // 任务就绪条件变量

    // 任务队列（使用动态数组模拟堆栈式队列，方便取末尾两个数）
    Result_t task_queue[INITIAL_SIZE];
    int queue_size; // 当前队列元素数量
    int total_tasks_processed; // 已处理的归约次数
    int final_result;

} ReductionMonitor;

// --- 全局管程实例 ---
ReductionMonitor mon;

void init_monitor(int initial_data[]);
void monitor_putResult(int result, int src_a, int src_b, int task_count);
int monitor_getTask(int* num1, int* num2, int* source_a_out, int* source_b_out, int*
    task_count_out);
void* thread_reduce(void* arg);


void init_monitor(int initial_data[]) {
    pthread_mutex_init(&mon.mutex, NULL);
    pthread_cond_init(&mon.task_ready, NULL);

    mon.queue_size = INITIAL_SIZE;
    mon.total_tasks_processed = 0;

    // 初始化任务队列
```

```
48      for (int i = 0; i < INITIAL_SIZE; i++) {
49          mon.task_queue[i].value = initial_data[i];
50          mon.task_queue[i].source_a = 0;
51          mon.task_queue[i].source_b = 0;
52          mon.task_queue[i].level = 0;
53      }
54      mon.final_result = 0;
55  }
56
57  // 提交结果 (V 操作)
58  void monitor_putResult(int result, int src_a, int src_b, int task_count) {
59      pthread_mutex_lock(&mon.mutex);
60
61      // 将结果放入队列
62      if (mon.queue_size < INITIAL_SIZE) {
63          mon.task_queue[mon.queue_size].value = result;
64          mon.task_queue[mon.queue_size].source_a = src_a;
65          mon.task_queue[mon.queue_size].source_b = src_b;
66          mon.task_queue[mon.queue_size].level = (mon.task_queue[mon.queue_size-1].level) +
                1; // 简单增加层级标识
67          mon.queue_size++;
68      }
69
70      // 打印归约过程
71      printf("[规约%2d] 线程 %lu: %d = %d + %d \n",
72             task_count,
73             (unsigned long)pthread_self(),
74             result, src_a, src_b);
75
76      // 通知等待的线程
77      pthread_cond_broadcast(&mon.task_ready);
78
79      pthread_mutex_unlock(&mon.mutex);
80  }
81
82  // 获取任务 (P 操作)
83  // 返回 1 表示成功获取任务, 返回 0 表示规约已完成
84  int monitor_getTask(int* num1, int* num2, int* source_a_out, int* source_b_out, int*
        task_count_out) {
85      pthread_mutex_lock(&mon.mutex);
86
87      // 循环检查: 只有当任务数大于等于 2, 或者归约已经完成时才退出等待。
88      while (mon.queue_size < 2) {
89          if (mon.total_tasks_processed == INITIAL_SIZE-1) {
90              // 规约完成: 16 个初始数据, 需要 16-1=15 次归约
91              // 当只剩下一个数时, 且所有任务对都已被处理
92              *num1 = mon.task_queue[0].value;
93              mon.final_result = *num1;
94              pthread_mutex_unlock(&mon.mutex);
```

```
95              return 0; // 通知线程退出
96          }
97
98          // 等待新的结果产生 (释放锁并阻塞)
99          pthread_cond_wait(&mon.task_ready, &mon.mutex);
100     }
101
102     // 提取队列末尾的两个数 (模拟从任务堆栈中取出)
103     mon.queue_size -= 2;
104     *num1 = mon.task_queue[mon.queue_size].value;
105     *num2 = mon.task_queue[mon.queue_size + 1].value;
106
107     *source_a_out = *num1;
108     *source_b_out = *num2;
109
110     mon.total_tasks_processed++;
111     *task_count_out = mon.total_tasks_processed; // 记录本次归约编号
112
113     pthread_mutex_unlock(&mon.mutex);
114     return 1; // 成功获取任务
115 }
116
117 void* thread_reduce(void* arg) {
118     int num1, num2, result;
119     int src_a, src_b;
120     int task_count;
121
122     // 不断获取任务, 直到归约完成
123     while (1) {
124         // 获取任务
125         int success = monitor_getTask(&num1, &num2, &src_a, &src_b, &task_count);
126         if (!success) {
127             // 归约完成, 线程退出
128             break;
129         }
130
131         // 随机时间扰动 (5ms 到 10ms)
132         int delay_ms = (rand() % 6) + 5;
133         usleep(delay_ms * 1000); // usleep 以微秒为单位
134
135         // 执行归约操作
136         result = num1 + num2;
137
138         // 提交结果
139         monitor_putResult(result, src_a, src_b, task_count);
140     }
141     return NULL;
142 }
143 int main() {
```

```
    int initial_data[INITIAL_SIZE];
    pthread_t threads[NUM_THREADS];

    // 随机初始化 16 个数据 (1 到 10 之间)
    printf("--- 初始数据 ---\n[");
    for (int i = 0; i < INITIAL_SIZE; i++) {
        initial_data[i] = (rand() % 10) + 1;
        printf("%d%s", initial_data[i], (i == INITIAL_SIZE - 1) ? "" : ", ");
    }
    printf("]\n\n");
    init_monitor(initial_data);
    printf("--- 归约过程 ---\n");
    for (long i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_reduce, NULL);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("--- 所有线程退出 ---\n");
    printf("\n==========================================\n");
    printf("最终归约结果: %d\n", mon.final_result);
    printf("==========================================\n");
    pthread_mutex_destroy(&mon.mutex);
    pthread_cond_destroy(&mon.task_ready);

    return 0;
}
```

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw7$ ./7.4_monitor
--- 初始数据 ---
[4, 7, 8, 6, 4, 6, 7, 3, 10, 2, 3, 8, 1, 10, 4, 7]

--- 归约过程 ---
[规约 1] 线程 133372613064448: 11 = 4 + 7
[规约 2] 线程 133372604671744: 11 = 1 + 10
[规约 6] 线程 133372571100928: 10 = 4 + 6
[规约 8] 线程 133372554315520: 11 = 4 + 7
[规约 7] 线程 133372562708224: 14 = 8 + 6
[规约 3] 线程 133372596279040: 11 = 3 + 8
[规约 4] 线程 133372587886336: 12 = 10 + 2
[规约 5] 线程 133372579493632: 10 = 7 + 3
[规约 9] 线程 133372604671744: 22 = 11 + 11
[规约10] 线程 133372554315520: 21 = 10 + 11
[规约11] 线程 133372596279040: 25 = 14 + 11
[规约12] 线程 133372579493632: 22 = 12 + 10
[规约13] 线程 133372571100928: 43 = 22 + 21
[规约14] 线程 133372579493632: 47 = 25 + 22
[规约15] 线程 133372613064448: 90 = 43 + 47
--- 所有线程退出 ---

==========================================
最终归约结果 : 90
==========================================
```

图 1: 运行结果