操作系统 第五次作业

姓名：朱首赫

学号：2023K8009906029

5.1 写一个两线程程序，两线程同时向一个数组分别写入 1000 万以内的奇数和偶数，写入过程中两个线程共用一个偏移量 index，代码逻辑如下所示。写完后打印出数组相邻两个数的最大绝对差值。

Listing 1: 代码示例

```
int MAX=10000000;
index = 0
//thread1
for(i=0;i<MAX;i+=2) {
data[index] = i; //even ( i+1 for thread 2)
index++;
}
//thread2
for(i=0;i<MAX;i+=2) {
data[index] = i+1; //odd
index++;
}
```

请分别按下列方法完成一个不会丢失数据的程序:

1) 请用 Peterson 算法实现上述功能;

2) 请学习了解 pthread_mutex_lock/unlock() 函数, 并实现上述功能;

3) 请学习了解 ___atomic_add_fetch (gcc > 4.7) 或者 C++ 中的 std::atomic 类型和 fetch_add 函数, 并实现上述功能。

提交:

1. 说明你所写程序中的临界区（注意：每次进入临界区之后，执行 200 次操作后离开临界区。）

2. 提供上述三种方法的源代码，运行结果截图 (即，数组相邻两个数的最大绝对差值)

3. 请找一个双核计算机系统测试前述 2）和 3）方法中完成数组写入时，各自所需的执行时间，不用提供计算绝对差值的时间。请注明使用的计算机的 CPU、内存和操作系统版本配置。

解答:

1) 表 2 即为 Peterson 算法实现上述功能的源码，其中临界区操作已在代码中标出，该区域代码原子性地分配该线程本次写入的数组范围（即限定 index 范围），实际的数据写入操作在临界区外，这样能保证临界区的操作最少且能解决竞态条件。运行结果如图 1 所示。

Listing 2: Peterson 算法实现

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <pthread.h>
#define MAX 10000000
#define CHUNK_SIZE 200
#define N 2
int flag[N] = {0};
int turn = 0;
int index = 0;
typedef struct {
    int* arr;     // 数组指针
    int even_odd; // 0:even 1:odd
} thread_arg_t;
void* write_num(void* arg){
    thread_arg_t *my_arg = (thread_arg_t *)arg;
    int *data = my_arg->arr;
    int cur = my_arg->even_odd;
    int opp = (cur + 1) % 2;
    int start, end;
    while (1)
    {
        flag[cur] = 1;  // 本进程想占用
        turn = opp;     // 允许另一进程占用
        while (flag[opp] && turn == opp);
        /*======原子性地分配index======*/
        if (index < MAX) {
            start = index;
            index += CHUNK_SIZE;
            end = (index > MAX) ? MAX : index;
        } else {
            flag[cur] = 0;
            break; // 工作完成，退出循环
        }
        /*============================*/
        flag[cur] = 0;
        for (int i = start; i < end; i++) {
            data[i] = i + cur;
        }
    }
}
int find_max_diff(int* a, int len){
    int max = 0;
    for (int i = 1; i < len; i++)
    {
        int diff = a[i] - a[i - 1];
        diff = diff < 0 ? -diff : diff;
        max = diff > max ? diff : max;
    }
    return max;
}
int main(int argc, char const *argv[])
```

```
{
    int *data = (int*)malloc(sizeof(int) * MAX);

    pthread_t threads[N];
    thread_arg_t args[N]={{.arr = data, .even_odd = 0}, {.arr = data, .even_odd = 1}};
    pthread_create(&threads[0], NULL, write_num, (void *)&args[0]);
    pthread_create(&threads[1], NULL, write_num, (void *)&args[1]);
    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);
    printf("The maximum absolute difference between two adjacent numbers: %d\n",
        find_max_diff(data, MAX));
    free(data);
    return 0;
}
```

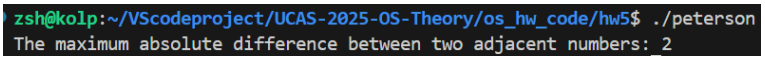

图 1: Peterson 算法运行结果

2) 表 3 即为利用 pthread_mutex_lock/unlock() 函数实现上述功能的源码，其中临界区操作同上，已在代码中标出。运行结果如图 2 所示，写入数组的操作用时 37881068 ns。(分析及运行环境在 **4)** 给出)

Listing 3: 互斥锁实现

```
// ...宏定义和库引入同上，略
int index = 0;
pthread_mutex_t mutex;
// ...thread_arg_t定义同上，略
void* write_num(void* arg){
    thread_arg_t *my_arg = (thread_arg_t *)arg;
    int *data = my_arg->arr;
    int cur = my_arg->even_odd;
    int opp = (cur + 1) % 2;
    int start, end;
    while (1)
    {
        // lock the mutex
        pthread_mutex_lock(&mutex);
        /*============临界区===============*/
        if (index >= MAX){
            pthread_mutex_unlock(&mutex);
            break;// 工作完成，退出循环
        }
        start = index;
        index += CHUNK_SIZE;
        end = (index > MAX) ? MAX : index;
        /*================================*/
        // Unlock the mutex
```

```
        pthread_mutex_unlock(&mutex);
        for (int i = start; i < end; i++) {
            data[i] = i + cur;
        }
    }
}
// ...find_max_diff函数定义，略
int main(int argc, char const *argv[])
{
    int *data = (int*)malloc(sizeof(int) * MAX);
    pthread_t threads[N];
    thread_arg_t args[N]={{.arr = data, .even_odd = 0}, {.arr = data, .even_odd = 1}};

    // Initialize the mutex
    if (pthread_mutex_init(&mutex, NULL) != 0) {
        perror("pthread_mutex_init error");
        return EXIT_FAILURE;
    }
    struct timespec start, end;
    long diff = 0;
    clock_gettime(CLOCK_MONOTONIC, &start);
    pthread_create(&threads[0], NULL, write_num, (void *)&args[0]);
    pthread_create(&threads[1], NULL, write_num, (void *)&args[1]);
    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);
    clock_gettime(CLOCK_MONOTONIC, &end);
    diff = (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
    printf("Writing array took %7ld ns\n", diff);
    // Destroy the mutex
    pthread_mutex_destroy(&mutex);
    printf("The maximum absolute difference between two adjacent numbers: %d\n",
        find_max_diff(data, MAX));
    free(data);
    return 0;
}
```

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw5$ ./mutex
Writing array took 37881068 ns
The maximum absolute difference between two adjacent numbers: 2
```

图 2: 互斥锁运行结果

3) 表 4 即为利用 ___atomic_add_fetch 原子指令实现上述功能的源码，其中临界区使用指令原子性地完成读取 index、计算 index+CHUNK_SIZE、将新值写入 index、返回旧值的操作。运行结果如图 3 所示，写入数组的操作用时 33531410 ns。

Listing 4: 原子指令实现

```
// ...略
void* write_num(void* arg){
```

```
    thread_arg_t *my_arg = (thread_arg_t *)arg;
    int *data = my_arg->arr;
    int cur = my_arg->even_odd;
    int opp = (cur + 1) % 2;
    int start, end;
    while (1)
    {
        /*============临界区===============*/
        start = __atomic_fetch_add(&index, CHUNK_SIZE, __ATOMIC_SEQ_CST);
        if (start >= MAX) break;
        end = (start + CHUNK_SIZE > MAX) ? MAX : start + CHUNK_SIZE;
        /*================================*/
        for (int i = start; i < end; i++) {
            data[i] = i + cur;
        }
    }
}
// ... main函数同上，略
```

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw5$ ./atomic
Writing array took 33531410 ns
The maximum absolute difference between two adjacent numbers: 2
```

图 3: 原子指令运行结果

4)cpu 配置： CPU 型号 13th Gen Intel(R) Core(TM) i9-13900HX；逻辑核心数 32；L1d cache 768 KiB；L1i cache 512 KiB；L2 cache 32 MiB；L3 cache 36 MiB；

操作系统版本： Linux 6.6.87.2-microsoft-standard-WSL2

总内存 (RAM)： 3.8 GiB

对于数组写入操作，使用互斥锁的用时为 37881068 ns，使用原子指令的用时为 33531410 ns。后者比前者快了约 13%。这是因为使用互斥锁实现，时每次加锁/解锁都需要在用户态和内核态之间进行切换，频繁切换时这是一笔不可忽视的开销；而原子指令有硬件直接实现，没有额外开销。

5.2 现有一个长度为 5 的整数数组，假设需要写一个两线程程序，其中，线程 1 负责往数组中写入 5 个随机数（1 到 20 范围内的随机整数），写完这 5 个数后，线程 2 负责从数组中读取这 5 个数，并求和。该过程循环执行 5 次。注意：每次循环开始时，线程 1 都重新写入 5 个数。

请思考： 上述过程能否通过 pthread_mutex_lock/unlock 函数实现？如果可以，请写出相应的源代码，并运行程序，打印出每次循环计算的求和值；如果无法实现，请分析并说明原因。

提交： 实现题述功能的源代码和打印结果，或者无法实现的原因分析说明。

解答： 不能仅仅依靠 pthread_mutex_lock/unlock 函数实现，因为该过程严格要求 T1 写完后 T2 才能读，而 mutex 仅能保证互斥而不能保证访问顺序。因此需要额外引入条件变量来进行线程同步，经调研发现可以使用 pthread_cond_wait/signal 函数来让线程在等待条件变量时

原子地释放锁并休眠；以及被条件变量唤醒并重获锁。由此实现的程序如表 5 所示，运行结果如图 4 所示，经检查确认求和结果正确。

Listing 5: 原子指令实现

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define ARR_LEN 5
#define TEST_TIMES 5
#define MAX_RAND 20
pthread_mutex_t mutex;
pthread_cond_t cv; // 条件变量
int is_written = 0; // 0: T1可写, 1: T2可读
int test_time = 0;
int data[ARR_LEN];
void* write_num(void* arg) {
    int *a = (int *)arg;
    while (1) {
        pthread_mutex_lock(&mutex);
        // 等待 T2 求和完成 (is_written == 0)
        while (is_written == 1 && test_time < TEST_TIMES) {
            // T1释放锁, 进入休眠状态, 等待 cv 信号后重新获取锁
            pthread_cond_wait(&cv, &mutex);
        }
        if (test_time >= TEST_TIMES) {
            pthread_mutex_unlock(&mutex);
            break;
        }
        printf("--- Test %d ---\n", test_time + 1);
        printf("Array: ");
        for (int i = 0; i < ARR_LEN; i++) {
            a[i] = rand() % MAX_RAND + 1;
            printf("%2d ", a[i]);
        }
        printf("\n");
        is_written = 1;
        pthread_cond_signal(&cv); // 唤醒 T2
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
void* add_num(void* arg) {
    int *a = (int *)arg;
    int sum;
    while (1) {
        pthread_mutex_lock(&mutex);
        // 等待 T1 写入完成 (is_written == 1)
        while (is_written == 0 && test_time < TEST_TIMES) {
            // T2释放锁, 进入休眠状态, 等待 cv 信号后重新获取锁
```

```
            pthread_cond_wait(&cv, &mutex);
        }
        if (test_time >= TEST_TIMES) {
            pthread_mutex_unlock(&mutex);
            break;
        }
        // 读取并求和
        sum = 0;
        for (int i = 0; i < ARR_LEN; i++) sum += a[i];
        test_time += 1; // 测试次数+1
        printf("Sum: %3d\n", sum);
        is_written = 0;
        pthread_cond_signal(&cv); // 唤醒 T1
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
int main(int argc, char const *argv[]) {
    if (pthread_mutex_init(&mutex, NULL) != 0 ||
        pthread_cond_init(&cv, NULL) != 0 ||
        pthread_cond_init(&cv, NULL) != 0){
        perror("Synchronization primitive initialization error");
        return EXIT_FAILURE;
    }
    pthread_t threads1, threads2;
    pthread_create(&threads1, NULL, write_num, (void *)data);
    pthread_create(&threads2, NULL, add_num, (void *)data);
    pthread_join(threads1, NULL);
    pthread_join(threads2, NULL);
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cv);
    pthread_cond_destroy(&cv);
    return 0;
}
```

```
--- Test 1 ---
Array:  4  7 18 16 14
Sum:  59
--- Test 2 ---
Array: 16  7 13 10  2
Sum:  48
--- Test 3 ---
Array:  3  8 11 20  4
Sum:  46
--- Test 4 ---
Array:  7  1  7 13 17
Sum:  45
--- Test 5 ---
Array: 12  9  8 10  3
Sum:  42
```

图 4: 运行结果