操作系统作业 5

宋婉婷 2022K8009929009

5.1 奇偶线程

5.1.1 题目要求

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。

```
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。

5.1.2 Peterson 算法

代码如下，其中 turn 使用原子变量以防止程序编译时指令重排序破坏了锁结构（在测试中甚至会改变最终结果的正确性）。

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdatomic.h>

#define MAX 10000000
#define BATCH_SIZE 200

int data[MAX];
int data_index = 0;

// Peterson 算法变量
bool flag[2];
atomic_int turn;
```

```c
void* thread_even(void* arg){
    int local_index = 0;
    while(local_index < MAX){
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn == 1) ;

        // 临界区：批量处理 200 个数据
        for(int i = 0; i < BATCH_SIZE && local_index < MAX; i++){
            data[data_index] = local_index;
            data_index++;
            local_index += 2;
        }
        // 退出临界区
        flag[0] = false;
    }
    return NULL;
}

void* thread_odd(void* arg){
    int local_index = 1;
    while(local_index < MAX){
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn == 0) ;

        // 临界区：批量处理 200 个数据
        for(int i = 0; i < BATCH_SIZE && local_index < MAX; i++){
            data[data_index] = local_index;
            data_index++;
            local_index += 2;
        }
        // 退出临界区
        flag[1] = false;
    }
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_even, NULL);
    pthread_create(&t2, NULL, thread_odd, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("index = %d (should be %d)\n", data_index, MAX);

    int max_diff = 0;
    for(int i=1; i<MAX; i++){
        int diff = abs(data[i] - data[i-1]);
        if(diff> max_diff){
            max_diff = diff;
        }
    }
    printf("max diff is: %d\n", max_diff);
    return 0;
}
```

临界区在代码中标出，具体为对全局共享变量 data 和 data_index 的修改，局部变量 local_index 不算在内。

运行结果如下：

```
# ./phta1
index = 10000000 (should be 10000000)
max diff is: 20799
# ./phta1
index = 10000000 (should be 10000000)
max diff is: 19599
# ./phta1
index = 10000000 (should be 10000000)
max diff is: 11197
```

5.1.3 mutex_lock/unlock

代码如下：

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdatomic.h>

#define MAX 10000000
#define BATCH_SIZE 200

int data[MAX];
int data_index = 0;

//mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* thread_even(void* arg){
    int local_index = 0;
    while(local_index < MAX){
        pthread_mutex_lock(&mutex);
        // 临界区：批量处理 200 个数据
        for(int i = 0; i < BATCH_SIZE && local_index < MAX; i++){
            data[data_index] = local_index;
            data_index++;
            local_index += 2;
        }
        // 退出临界区
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

void* thread_odd(void* arg){
    int local_index = 1;
    while(local_index < MAX){
        pthread_mutex_lock(&mutex);
        // 临界区：批量处理 200 个数据
        for(int i = 0; i < BATCH_SIZE && local_index < MAX; i++){
            data[data_index] = local_index;
            data_index++;
            local_index += 2;
        }
        // 退出临界区
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_even, NULL);
    pthread_create(&t2, NULL, thread_odd, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("index = %d (should be %d)\n", data_index, MAX);

    int max_diff = 0;
    for(int i=1; i<MAX; i++){
        int diff = abs(data[i] - data[i-1]);
        if(diff> max_diff){
            max_diff = diff;
        }
    }
    printf("max diff is: %d\n", max_diff);
    return 0;
}
```

临界区同上。运行结果如下：

```
# ./phta2
index = 10000000 (should be 10000000)
max diff is: 3133599
# ./phta2
index = 10000000 (should be 10000000)
max diff is: 1110799
# ./phta2
index = 10000000 (should be 10000000)
max diff is: 2416397
```

5.1.4 __atomic_add_fetch

代码如下，因为 `atomic_add_fetch` 已经原子地执行 `data_index += BATCH_SIZE`，因此无需在退出临界区时再次同步。

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

```
#include <stdbool.h>
#include <stdatomic.h>

#define MAX 10000000
#define BATCH_SIZE 200

int data[MAX];
atomic_int data_index = 0; // 用于 atomic_add_fetch 第一个参数

void* thread_even(void* arg){
    int local_index = 0;
    while (local_index < MAX) {
        // 原子地获取当前 data_index，并增加 BATCH_SIZE
        int current_index = __atomic_add_fetch(&data_index, BATCH_SIZE,
__ATOMIC_RELAXED);
        int end_index = current_index;
        int start_index = end_index - BATCH_SIZE;

        // 检查越界
        if (start_index>= MAX) break;
        if (end_index> MAX) end_index = MAX;

        // 批量写入
        for (int i = start_index; i < end_index; i++) {
            data[i] = local_index;
            local_index += 2;
        }
    }
    return NULL;
}

void* thread_odd(void* arg){
    int local_index = 1;
    while (local_index < MAX) {
        // 原子地获取当前 data_index，并增加 BATCH_SIZE
        int current_index = __atomic_add_fetch(&data_index, BATCH_SIZE,
__ATOMIC_RELAXED);
        int end_index = current_index;
        int start_index = end_index - BATCH_SIZE;

        // 检查越界
        if (start_index>= MAX) break;
        if (end_index> MAX) end_index = MAX;

        // 批量写入
        for (int i = start_index; i < end_index; i++) {
            data[i] = local_index;
            local_index += 2;
        }
    }
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_even, NULL);
    pthread_create(&t2, NULL, thread_odd, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("index = %d (should be %d)\n", data_index, MAX);

    int max_diff = 0;
    for(int i=1; i<MAX; i++){
        int diff = abs(data[i] - data[i-1]);
        if(diff> max_diff){
            max_diff = diff;
        }
    }
    printf("max diff is: %d\n", max_diff);
    return 0;
}
```

临界区同上。运行结果如下：

```
# ./phta3
index = 10000000 (should be 10000000)
max diff is: 587597
# ./phta3
index = 10000000 (should be 10000000)
max diff is: 385997
# ./phta3
index = 10000000 (should be 10000000)
max diff is: 878399
```

5.1.5 双核性能测试

与作业 3 使用方法类似，对线程分配 cpu 核并绑定，添加以下代码：

```
#define _GNU_SOURCE
#include <sched.h>
...
void* thread_even(void* arg){
    cpu_set_t cpuset;   //CPU 核的位图
    CPU_ZERO(&cpuset);  // 将位图清零
    CPU_SET(0, &cpuset);  // 设置位图第 N 位为 1，表示与 core N 绑定。N 从 0 开始计数
    sched_setaffinity(0, sizeof(cpuset), &cpuset);  // 将当前线程和 cpuset 位图中指定
的核绑定运行
    ...
//thread_odd 同理，修改参数 0 为 1
...
int main(){
    struct timespec time1 = {0, 0};
    struct timespec time2 = {0, 0};
    pthread_t t1, t2;
    clock_gettime(CLOCK_MONOTONIC, &time1);
    pthread_create(&t1, NULL, thread_even, NULL);
    pthread_create(&t2, NULL, thread_odd, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    clock_gettime(CLOCK_MONOTONIC, &time2);
    long sec = time2.tv_sec - time1.tv_sec;
    long nsec = time2.tv_nsec - time1.tv_nsec;
    if(nsec < 0){
        sec--;
        nsec += 1000000000;
    }
    printf("%ld\n",nsec);
```

分别执行方法二和方法三 10 次，统计平均执行时间，比较如下：

Method	Average time /ns
pthread_mutex_lock/unock()	94020457
__atomic_add_fetch	39460142.2

可以看出原子操作的时间较少，执行效率较高。而 mutex 可能涉及较多同步操作，导致用时较多。

系统环境为在 Docker 上的 Ubuntu24.04 容器，操作系统配置如下：

```
# cat /proc/cpuinfo | grep "model name" | head -n 1
model name      : 12th Gen Intel(R) Core(TM) i5-1235U      -- CPU 架构型号
# nproc
2                                                          -- 可用 CPU 核
# cat /proc/meminfo | grep MemTotal
MemTotal:        7986356 kB                                -- 可用内存
# cat /etc/os-release | grep PRETTY_NAME
PRETTY_NAME="Ubuntu 24.04.3 LTS"                           -- 操作系统版本
```

注：以上四条指令由 文心一言 大模型提供

5.2 互斥同步

5.2.1 题目要求

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。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。

5.2.2 分析实现

不能只通过 pthread_mutex_lock/unlock 实现该功能，因为 mutex 只能提供线程的互斥访问，不能让他们之间交流状态，导致线程 2 还没读完线程 1 就写进新数据，那线程 2 就可能重复读取到同一行数据来错误计算。

为解决此问题可以引入一个信号，让线程 1 写完后发给线程 2，在线程 2 读取完成前使线程 1 等待，不要写入新数据。查阅资料发现线程条件变量 pthread_cond_t 可以满足需求，其中关键函数：

1. `pthread_cond_wait` 使自身线程阻塞并释放锁让另一个线程获取执行，在收到信号后重新加锁并唤醒自身进程
2. `pthread_cond_signal` 向另一个线程发送信号，唤醒其它线程，实现线程间沟通

据此编写程序如下：

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

#define ARRAY_SIZE 5
#define LOOP_COUNT 5

int array[ARRAY_SIZE];
int write_complete = 0;  // 标记线程 1 是否写完

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *thread1_write(void *arg) {
    for (int loop = 0; loop < LOOP_COUNT; loop++) {
        pthread_mutex_lock(&mutex);
        for (int i = 0; i < ARRAY_SIZE; i++) {
            array[i] = rand() % 20 + 1;
        }

        printf("Thread1 wrote:");
        for (int i = 0; i < ARRAY_SIZE; i++) {
            printf("%d", array[i]);
        }
        printf("\n");

        write_complete = 1;
        pthread_cond_signal(&cond);  // 通知线程 2
        pthread_mutex_unlock(&mutex);

        // 等待线程 2 读完
        pthread_mutex_lock(&mutex);
        while (write_complete) {
            pthread_cond_wait(&cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

void *thread2_read(void *arg) {
    for (int loop = 0; loop < LOOP_COUNT; loop++) {
        pthread_mutex_lock(&mutex);
        while (!write_complete) {
            pthread_cond_wait(&cond, &mutex);  // 等待线程 1 写完
        }
        // 读取数组并求和
        int sum = 0;
        printf("Thread2 read:");
        for (int i = 0; i < ARRAY_SIZE; i++) {
            printf("%d", array[i]);
            sum += array[i];
        }
        printf("| Sum result = %d\n", sum);

        write_complete = 0;  // 标记读完
        pthread_cond_signal(&cond);  // 通知线程 1 可以继续写入
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    srand(time(NULL));
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1_write, NULL);
    pthread_create(&t2, NULL, thread2_read, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);

    return 0;
}
```

测试结果如下，符合预期。

```
# ./phtb
Thread1 wrote: 6 4 1 17 18
Thread2 read: 6 4 1 17 18 | Sum result = 46
Thread1 wrote: 11 20 16 18 5
Thread2 read: 11 20 16 18 5 | Sum result = 70
Thread1 wrote: 12 20 3 14 18
Thread2 read: 12 20 3 14 18 | Sum result = 67
Thread1 wrote: 19 20 11 12 18
Thread2 read: 19 20 11 12 18 | Sum result = 80
Thread1 wrote: 5 10 10 15 16
Thread2 read: 5 10 10 15 16 | Sum result = 56
```