作业 6

毕定钧 2021K8009906014

本次作业包含:

6.1

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*，代码逻辑如右图所示。写完后打印出数组相邻两个数的最大绝对差值。

```
int MAX = 10000000;
int 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*()(*__sync_fetch_and_add*() *for gcc* 4.1+) 函数，并实现上述功能。

提交内容:

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. 请找一个双核系统测试三种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。

6.2

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:

1) 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。

提交内容: 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。

6.1

Peterson 算法

Peterson 算法适用于只有两个进程（或线程）之间的互斥问题，这两个进程共享一个临界资源。假设存在两个进程 *P*0 和 *P*1，它们都希望访问临界区。*Peterson* 算法需要定义两个标志位 *flag*[2] 与 *turn*，其中 *flag* 为 *bool* 类型，用来表示两个进程有意图进入临界区，*turn* 为整型，用来指示哪个进程有权限进入临界区。初始时，两个 *flag* 都是 *false*，*turn* 的初始值为 0，即表示 *P*0 有权限进入临

界区。进程在进入临界区之前，会先设置自己的 $flag$ 为 $true$，并将 $turn$ 设置为另一个进程的标识（0 或 1）。进程会检查另一个进程的 $flag$ 和 $turn$，以确定是否可以进入临界区。如果另一个进程也想进入临界区，并且它的 $turn$ 标识比当前进程小，那么当前进程会等待，直到轮到自己进入临界区。当一个进程离开临界区时，它会将自己的 $flag$ 设置为 $false$，表示不再有意进入临界区，这样另一个进程就有机会进入。

(1)

```c
void PetersonLock(int thread) {
    flag[thread] = true;
    turn = 1 - thread;
    while (flag[1 - thread] && turn == 1 - thread);
}

void PetersonUnLock(int thread) {
    flag[thread] = false;
}

void *add_even(void) {
    for (int i = 0; i < MAXNUM; i += 2) {
        if (i % 200 == 0) PetersonLock(0);
        data[index++] = i;
        if (i % 200 == 198) PetersonUnLock(0);
    }
    return NULL;
}

void *add_odd(void) {
    for (int i = 0; i < MAXNUM; i += 2) {
        if (i % 200 == 0) PetersonLock(1);
        data[index++] = i + 1;
        if (i % 200 == 198) PetersonUnLock(1);
    }
    return NULL;
}

int main() {
    pthread_t thread_0, thread_1;
    pthread_create(&thread_0, NULL, add_even, NULL);
    pthread_create(&thread_1, NULL, add_odd, NULL);
    pthread_join(thread_0, NULL);
    pthread_join(thread_1, NULL);

    int max = 0;
    for (int i = 1; i < MAXNUM; i++) {
        int now = abs(data[i] - data[i - 1]);
        if (now > max) max = now;
    }

    printf("Max delta = %d\n", max);
    return 0;
}
```

```
stu@stu:~/OS/task6$ ./a.out
Max delta = 999999
stu@stu:~/OS/task6$ ./a.out
Max delta = 541399
stu@stu:~/OS/task6$ ./a.out
Max delta = 603197
stu@stu:~/OS/task6$ ./a.out
Max delta = 247399
```

(2)

```c
pthread_mutex_t index_mutex = PTHREAD_MUTEX_INITIALIZER;

void *add_even(void) {
    for (int i = 0; i < MAXNUM; i += 2) {
        if (i % 200 == 0) pthread_mutex_lock(&index_mutex);
        data[index++] = i;
        if (i % 200 == 198) pthread_mutex_unlock(&index_mutex);
    }
    return NULL;
}

void *add_odd(void) {
    for (int i = 0; i < MAXNUM; i += 2) {
        if (i % 200 == 0) pthread_mutex_lock(&index_mutex);
        data[index++] = i + 1;
        if (i % 200 == 198) pthread_mutex_unlock(&index_mutex);
    }
    return NULL;
}

int main() {
    pthread_t thread_0, thread_1;
    pthread_create(&thread_0, NULL, add_even, NULL);
    pthread_create(&thread_1, NULL, add_odd, NULL);
    pthread_join(thread_0, NULL);
    pthread_join(thread_1, NULL);

    int max = 0;
    for (int i = 1; i < MAXNUM; i++) {
        int now = abs(data[i] - data[i - 1]);
        if (now > max) max = now;
    }

    printf("Max delta = %d\n", max);
    pthread_mutex_destroy(&index_mutex);
    return 0;
}
```

```
stu@stu:~/OS/task6$ ./a.out
Max delta = 396199
stu@stu:~/OS/task6$ ./a.out
Max delta = 510599
stu@stu:~/OS/task6$ ./a.out
Max delta = 188599
stu@stu:~/OS/task6$ ./a.out
Max delta = 355799
```

(3)

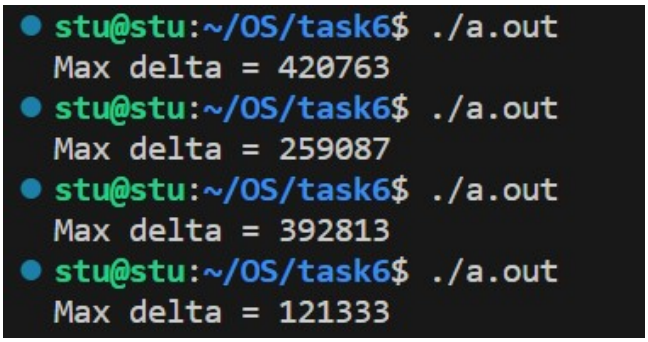
```c
void *add_even(void) {
    for (int i = 0; i < MAXNUM; i += 2) {
        int temp = __sync_fetch_and_add(&index, 1);
        data[temp] = i;
    }
    return NULL;
}

void *add_odd(void) {
    for (int i = 0; i < MAXNUM; i += 2) {
        int temp = __sync_fetch_and_add(&index, 1);
        data[temp] = i + 1;
    }
    return NULL;
}

int main() {
    pthread_t thread_0, thread_1;
    pthread_create(&thread_0, NULL, add_even, NULL);
    pthread_create(&thread_1, NULL, add_odd, NULL);
    pthread_join(thread_0, NULL);
    pthread_join(thread_1, NULL);

    int max = 0;
    for (int i = 1; i < MAXNUM; i++) {
        int now = abs(data[i] - data[i - 1]);
        if (now > max) max = now;
    }

    printf("Max delta = %d\n", max);
    return 0;
}
```



在程序中，通过循环来执行赋值操作，那么就可以使用循环变量 i 进行判断。每次 i 为 200 的倍数即代表进入进程，此时解锁，执行 100 次后，i 为 198，此时加锁，即实现了进入临界区后执行 100 次操作离开临界区，如果要实现 200 次，可以将判断改为 398 即可。根据测试结果，Peterson 耗时最短，Mutex 其次，使用原子操作耗时较长。

6.2

```c
void* thread1(void* arg) {
    for (int i = 0; i < NUM_LOOPS; i++) {
        for (int j = 0; j < ARRAY_SIZE; j++) {
            data[j] = rand() % 20 + 1; // 写入随机数
        }
        sem_post(&sem1); // 通知线程2可以开始计算
        sem_wait(&sem2); // 等待线程2完成计算
    }
    pthread_exit(NULL);
}

void* thread2(void* arg) {
    for (int i = 0; i < NUM_LOOPS; i++) {
        int sum = 0;
        sem_wait(&sem1); // 等待线程1写入数据
        for (int j = 0; j < ARRAY_SIZE; j++) {
            sum += data[j]; // 求和
            printf("THREAD2: Add %d to sum.\n", data[j]);
        }
        printf("Sum for iteration %d: %d\n", i + 1, sum);
        sem_post(&sem2); // 通知线程1可以继续
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;
    sem_init(&sem1, 0, 0);
    sem_init(&sem2, 0, 0);
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    sem_destroy(&sem1);
    sem_destroy(&sem2);
    return 0;
}
```

```
THREAD2: Add 4 to sum.
THREAD2: Add 7 to sum.
THREAD2: Add 18 to sum.
THREAD2: Add 16 to sum.
THREAD2: Add 14 to sum.
Sum for iteration 1: 59
THREAD2: Add 16 to sum.
THREAD2: Add 7 to sum.
THREAD2: Add 13 to sum.
THREAD2: Add 10 to sum.
THREAD2: Add 2 to sum.
Sum for iteration 2: 48
THREAD2: Add 3 to sum.
THREAD2: Add 8 to sum.
THREAD2: Add 11 to sum.
THREAD2: Add 20 to sum.
THREAD2: Add 4 to sum.
Sum for iteration 3: 46
THREAD2: Add 7 to sum.
THREAD2: Add 1 to sum.
THREAD2: Add 7 to sum.
THREAD2: Add 13 to sum.
THREAD2: Add 17 to sum.
Sum for iteration 4: 45
THREAD2: Add 12 to sum.
THREAD2: Add 9 to sum.
THREAD2: Add 8 to sum.
THREAD2: Add 10 to sum.
THREAD2: Add 3 to sum.
Sum for iteration 5: 42
```