作业 6

贾城昊
2021K8009929010

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()(_sync_fetch_and_add()for gcc 4.1+)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。

调研：

pthread_mutex_lock/unlock()

互斥锁（Mutex，全名为 Mutual Exclusion）是一种多线程同步机制，用于控制多个线程对共享资源的访问，以避免竞态条件（Race Condition）和数据不一致性问题。互斥锁确保在任何给定时间只有一个线程可以访问共享资源，其他线程必须等待互斥锁被释放才能继续执行。在多线程编程中，互斥锁是非常重要的工具，用于确保线程安全性。

以下是一般的互斥锁的使用步骤：

1. **定义互斥锁变量：**

   首先需要创建一个互斥锁变量，以便线程能够使用它来访问共享资源。在C语言中，可以使用 pthread_mutex_t 类型定义互斥锁变量。可以选择静态初始化或动态初始化互斥锁。

   - 静态初始化：使用 PTHREAD_MUTEX_INITIALIZER 宏来进行初始化，通常在定义变量的同时进行初始化。
   - 动态初始化：使用 pthread_mutex_init 函数进行初始化，通常在运行时进行。

2. **获取互斥锁：**

   在访问共享资源之前，线程需要请求获取互斥锁。使用 pthread_mutex_lock 函数来获取互斥锁。如果互斥锁已经被其他线程占用，当前线程将被阻塞，直到互斥锁可用。

3. **访问共享资源：**

   一旦线程成功获取了互斥锁，它就可以安全地访问共享资源，进行读取或写入操作。

4. **释放互斥锁：**

   当线程完成对共享资源的操作后，应使用 pthread_mutex_unlock 函数来释放互斥锁，以允许其他线程获取它。

5. **销毁互斥锁（可选）：**

   如果使用了动态初始化方式，在不再需要互斥锁时，应使用 pthread_mutex_destroy 函数来销毁互斥锁，以释放相关资源

atomic_add()

C语言中的atomic_add()函数不是标准C语言库函数，而是C11和C++11标准引入的 <stdatomic.h> 头文件提供的一组原子操作函数之一。这些函数用于执行原子操作，以确保在多线程环境中对共享变量的操作是原子的，不会被中断或交错执行，从而确保数据的一致性

和可靠性。atomic_add()函数的主要作用是将指定的值原子地添加到一个 _Atomic 类型的变量中。

以下是atomic_add()函数的一般形式：

```
T atomic_add(volatile _Atomic T *obj, T operand);
```

其中，T 是变量的类型，obj 是指向 _Atomic 类型变量的指针，operand 是要添加的值。这个函数将 operand 的值原子地添加到 obj 指向的变量，并返回添加前的值。这个操作是原子的，确保在多线程环境中不会发生竞争条件。

主要要点和用法：

1. _Atomic 类型：要使用atomic_add()函数，变量必须声明为 _Atomic 类型。这是C11和C++11引入的原子类型，用于表示能够进行原子操作的变量。例如，可以声明 _Atomic int 变量。
2. volatile 限定符：volatile 限定符通常与 _Atomic 类型变量一起使用，以确保编译器不会对这些变量的读取和写入进行优化，从而保持原子性。
3. 返回值：atomic_add()函数返回执行加法操作前的值。这对于获取先前的值以执行其他操作非常有用。
4. 原子性：atomic_add()函数的操作是原子的，不会被其他线程的操作中断。这确保了多线程环境下的数据安全性。

(1) 使用 Peterson 算法

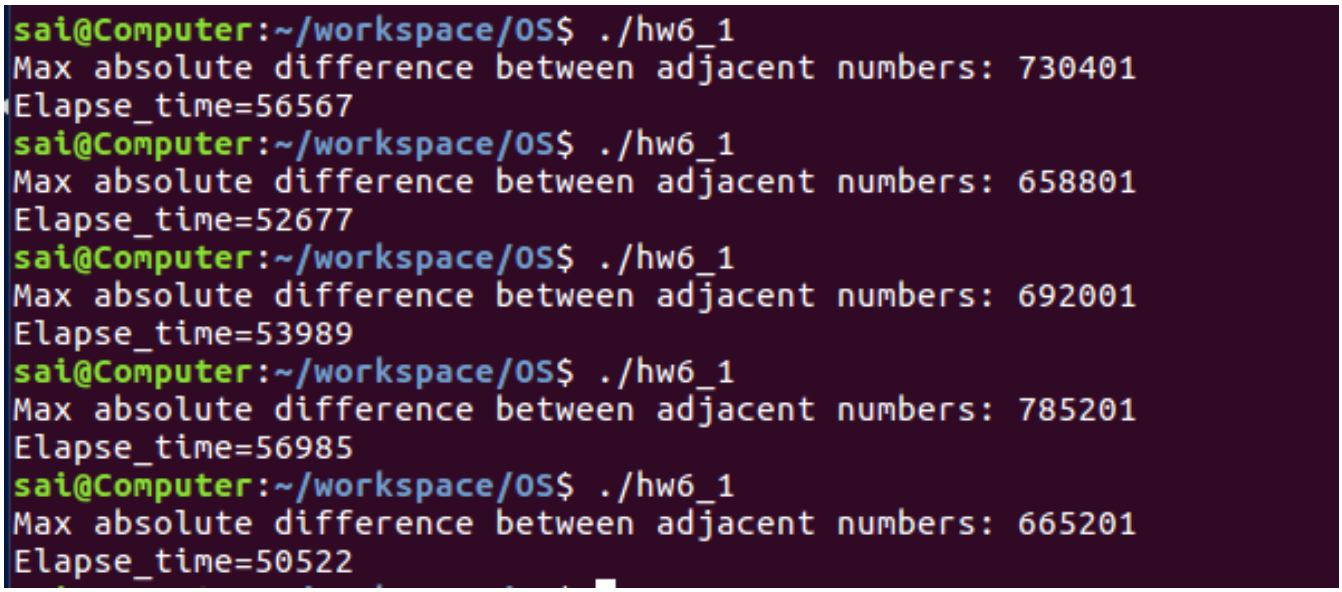
1. C 程序如下：

```
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>

#define MAX 10000000

int data[MAX];
int index = 0;
int turn = 0;
int flag[2] = {0, 0};

void* thread1(void* arg) {
	for (int i = 0; i < MAX; i += 2) {
		if (i % 400 == 0) {		// Lock when 100 operation starts
			flag[0] = 1;
			turn = 1;
			while (flag[1] && turn == 1)
				;
		}
		data[index] = i;		// even
```

```
            index ++;
            if (i % 400 == 398) {           // Unlock when 100 operation ends
                flag[0] = 0;
            }
        }
        pthread_exit(NULL);
}

void* thread2(void* arg) {
        for (int i = 0; i < MAX; i += 2) {
            if (i % 400 == 0) {             // Lock when 100 operation starts
                flag[1] = 1;
                turn = 0;
                while (flag[0] && turn == 0)
                    ;
            }
            data[index] = i + 1;      // odd
            index ++;
            if (i % 400 == 398) {           // Unlock when 100 operation ends
                flag[1] = 0;
            }
        }
        pthread_exit(NULL);
}

int main() {
        pthread_t tid1, tid2;

        struct timeval start_time, end_time;
        gettimeofday(&start_time, NULL);

        pthread_create(&tid1, NULL, thread1, NULL);
        pthread_create(&tid2, NULL, thread2, NULL);

        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);

        gettimeofday(&end_time, NULL);
        long elapse_time = (end_time.tv_sec - start_time.tv_sec) * 1000000 + (end_time.tv_usec -
start_time.tv_usec);

        int max_diff = 0;
        for (int i = 1; i < MAX; i++) {
            int diff = data[i] - data[i - 1];
            if (diff > max_diff) {
                max_diff = diff;
            }
        }

        printf("Max absolute difference between adjacent numbers: %d\n", max_diff);
        printf("Elapse_time=%ld\n", elapse_time);
```

```
    return 0;
}
```

2. 运行结果：



3. 临界区：

这个程序中的共享资源是 index 和 data 数组，因此程序的临界区分别是：

```
data[index] = I;
index++;
```

和

```
data[index++] = i + 1;
index++;
```

(2) 使用 pthread_mutex_lock/unlock()函数:

1. C 程序如下：

```
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>

#define MAX 10000000

int data[MAX];
int index = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* thread1(void* arg) {
    for (int i = 0; i < MAX; i += 2) {
```

```
            if (i % 400 == 0) {             // Lock when 100 operation starts
                pthread_mutex_lock(&mutex);
            }
            data[index] = i;
            index++;
            if (i % 400 == 398) {           // Unlock when 100 operation ends
                pthread_mutex_unlock(&mutex);
            }
        }
        pthread_exit(NULL);
}

void* thread2(void* arg) {
        for (int i = 0; i < MAX; i += 2) {
            if (i % 400 == 0) {             // Lock when 100 operation starts
                pthread_mutex_lock(&mutex);
            }
            data[index] = i + 1;
            index++;
            if (i % 400 == 398) {           // Unlock when 100 operation ends
                pthread_mutex_unlock(&mutex);
            }
        }
        pthread_exit(NULL);
}

int main() {
        pthread_t tid1, tid2;

        struct timeval start_time, end_time;
        gettimeofday(&start_time, NULL);

        pthread_create(&tid1, NULL, thread1, NULL);
        pthread_create(&tid2, NULL, thread2, NULL);

        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);

        gettimeofday(&end_time, NULL);
        long elapse_time = (end_time.tv_sec - start_time.tv_sec) * 1000000 + (end_time.tv_usec -
start_time.tv_usec);

        int max_diff = 0;
        for (int i = 1; i < MAX; i++) {
            int diff = data[i] - data[i - 1];
            if (diff > max_diff) {
                max_diff = diff;
            }
        }

        printf("Max absolute difference between adjacent numbers: %d\n", max_diff);
        printf("Elapsed time=%ld\n", elapse_time);
```

```
    return 0;
}
```

2. 运行结果：

```
sai@Computer:~/workspace/OS$ ./hw6_2
Max absolute difference between adjacent numbers: 4704403
Elapsed time=40762
sai@Computer:~/workspace/OS$ ./hw6_2
Max absolute difference between adjacent numbers: 4119201
Elapsed time=40626
sai@Computer:~/workspace/OS$ ./hw6_2
Max absolute difference between adjacent numbers: 7085603
Elapsed time=45918
sai@Computer:~/workspace/OS$ ./hw6_2
Max absolute difference between adjacent numbers: 3961603
Elapsed time=43038
sai@Computer:~/workspace/OS$ ./hw6_2
Max absolute difference between adjacent numbers: 2335601
Elapsed time=39843
```

3. 临界区：

这个程序中的共享资源是 index 和 data 数组，因此程序的临界区分别是：

```
data[index] = I;
index++;
```

和

```
data[index++] = i + 1;
index++;
```

(3)使用 atomic_add()

1. C 程序如下：

```
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>
#include <stdatomic.h>

#define MAX 10000000

int data[MAX];
_Atomic int index = 0;
```

```
void* thread1(void* arg) {
	for (int i = 0; i < MAX;) {
		int begin = atomic_fetch_add(&index, 200);
		for(int j = 0;j < 200; j++, i += 2){
			data[begin + j] = i;
		}
	}
	pthread_exit(NULL);
}

void* thread2(void* arg) {
	for (int i = 0; i < MAX;) {
		int begin = atomic_fetch_add(&index, 200);
		for(int j = 0;j < 200; j++, i += 2){
			data[begin + j] = i + 1;
		}
	}
	pthread_exit(NULL);
}

int main() {
	pthread_t tid1, tid2;

	struct timeval start_time, end_time;
	gettimeofday(&start_time, NULL);

	pthread_create(&tid1, NULL, thread1, NULL);
	pthread_create(&tid2, NULL, thread2, NULL);

	pthread_join(tid1, NULL);
	pthread_join(tid2, NULL);

	gettimeofday(&end_time, NULL);
	long elapse_time = (end_time.tv_sec - start_time.tv_sec) * 1000000 + (end_time.tv_usec -
start_time.tv_usec);

	int max_diff = 0;
	for (int i = 1; i < MAX; i++) {
		int diff = data[i] - data[i - 1];
		if (diff > max_diff) {
			max_diff = diff;
		}
	}

	printf("Max absolute difference between adjacent numbers: %d\n", max_diff);
	printf("Elapsed time=%ld\n", elapse_time);

	return 0;
}
```

2. 运行结果：

```
sai@Computer:~/workspace/OS$ ./hw6_3
Max absolute difference between adjacent numbers: 4873601
Elapsed time=21979
sai@Computer:~/workspace/OS$ ./hw6_3
Max absolute difference between adjacent numbers: 838001
Elapsed time=19298
sai@Computer:~/workspace/OS$ ./hw6_3
Max absolute difference between adjacent numbers: 664801
Elapsed time=18073
sai@Computer:~/workspace/OS$ ./hw6_3
Max absolute difference between adjacent numbers: 2183601
Elapsed time=19830
sai@Computer:~/workspace/OS$ ./hw6_3
Max absolute difference between adjacent numbers: 641201
Elapsed time=23311
```

3. 临界区：

这个程序中的共享资源仍然是 index 和 data 数组，但通过原子操作，两个线程不会访问到 data 数组的相同位置，因此程序的临界区：

```
int begin = atomic_fetch_add(&index, 200);;
```

(4)对时间结果的分析

可以看出第一种方法（Peterson 算法）的执行时间较长，第二种方法(pthread_mutex)次之，而第三种方法（原子操作）执行时间最短，下面是本人最初认为的一些原因

1. **Peterson 算法：**

   Peterson 算法是一种经典的软件级别的互斥锁算法，它依赖于共享内存中的标志和一个轮询等待机制。

   在多核系统上，Peterson 算法可能导致更多的上下文切换和竞争。当一个线程进入临界区时，另一个线程必须等待，这可能导致不必要的等待时间。

   原理上，Peterson 算法可能导致较长的执行时间，尤其是在高度竞争的情况下。

2. **pthread_mutex：**

pthread_mutex 是基于操作系统的互斥锁机制，它提供了相对高效的互斥保护。它利用底层操作系统内核支持，因此通常是可靠的。

pthread_mutex 可能涉及到一些额外的开销，如内核调用和上下文切换，这可能会导致相对较长的执行时间，但在多核系统上通常能够提供良好的性能。

3. **原子操作：**

原子操作通常是硬件级别的操作，可以保证多个线程对共享变量的操作是原子的，因此不需要额外的锁。

原子操作通常非常高效，因为它们避免了大部分锁竞争和上下文切换，以及不必要的等待。

在现代多核处理器上，原子操作通常能够以非常快的速度执行，因为它们利用硬件支持。

因此，第三种方法中的原子操作执行速度较快，因为它们利用了硬件级别的支持，减少了竞争和上下文切换，而第二种方法中的 pthread_mutex 相对较快，但可能会有一些额外的操作系统开销，最后，第一种方法中的 Peterson 算法在多核系统上可能会导致更多的上下文切换和竞争，因此执行时间较长。

但是，本人也注意到第三种方法与前两种方法的临界区并不相同，为了进一步控制变量，本人针对前两种方法重新写了与第三种方法类似的实现方式：

(5)对方法一与方法二重写以控制变量(更好比较时间)

方法一的新逻辑如下：

```
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>

#define MAX 10000000

int data[MAX];
int index = 0;
int turn = 0;
int flag[2] = {0, 0};

void *thread1(void* arg)
{
	for (int i = 0; i < MAX; )
	{
		int begin = 0;
		flag[0] = 1;
		turn = 1;
		while (flag[1] && turn == 1)
			;
```

```
            begin = index;
            index += 200;

            flag[0] = 0;
            for (int j = 0; j < 200; i += 2, j ++)
                data[begin + j] = i;
        }
        pthread_exit(NULL);
}

void *thread2(void* arg)
{
        for (int i = 0; i < MAX; )
        {
            int begin = 0;
            flag[1] = 1;
            turn = 0;
            while (flag[0] && turn == 0)
                ;

            begin = index;
            index += 200;

            flag[1] = 0;
            for (int j = 0; j < 200; i += 2, j++)
                data[begin + j] = i + 1;
        }
        pthread_exit(NULL);
}

int main() {
        pthread_t tid1, tid2;

        struct timeval start_time, end_time;
        gettimeofday(&start_time, NULL);

        pthread_create(&tid1, NULL, thread1, NULL);
        pthread_create(&tid2, NULL, thread2, NULL);

        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);

        gettimeofday(&end_time, NULL);
        long elapse_time = (end_time.tv_sec - start_time.tv_sec) * 1000000 + (end_time.tv_usec -
start_time.tv_usec);

        int max_diff = 0;
        for (int i = 1; i < MAX; i++) {
            int diff = data[i] - data[i - 1];
            if (diff > max_diff) {
                max_diff = diff;
            }
```

```
    }

    printf("Max absolute difference between adjacent numbers: %d\n", max_diff);
    printf("Elapse_time=%ld\n", elapse_time);


    return 0;
}
```

运行结果如下：

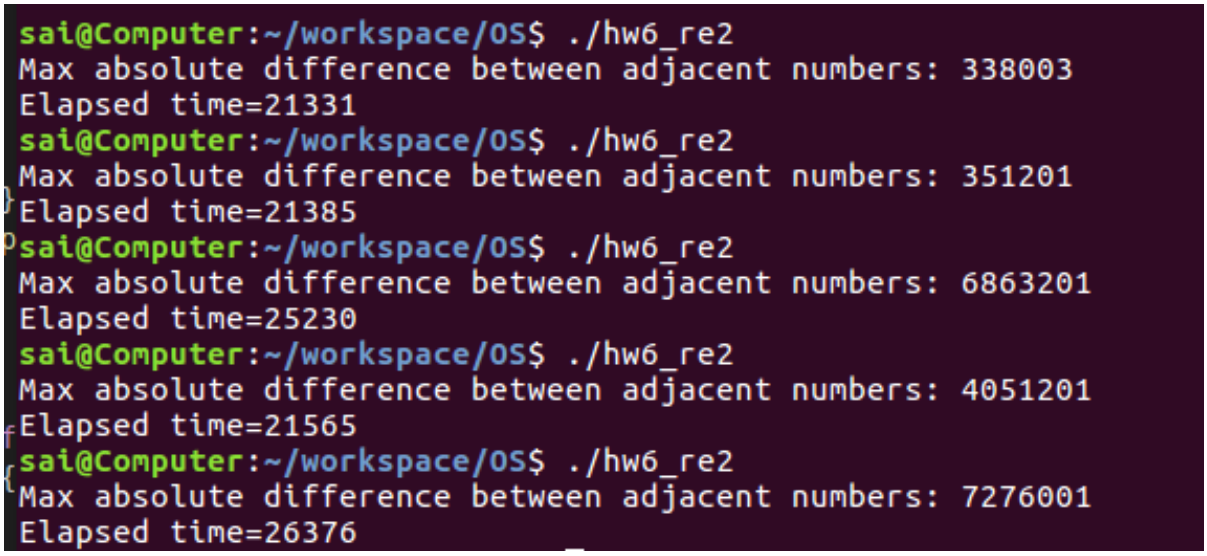
```
Max absolute difference between adjacent numbers: 4559201
Elapse_time=22527
sai@Computer:~/workspace/OS$ ./hw6_re1
Max absolute difference between adjacent numbers: 3589201
Elapse_time=21991
sai@Computer:~/workspace/OS$ ./hw6_re1
Max absolute difference between adjacent numbers: 4491601
Elapse_time=22902
sai@Computer:~/workspace/OS$ ./hw6_re1
Max absolute difference between adjacent numbers: 4846401
Elapse_time=19432
sai@Computer:~/workspace/OS$ ./hw6_re1
Max absolute difference between adjacent numbers: 9342801
Elapse_time=21248
```

方法二的新逻辑如下：

```
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>

#define MAX 10000000

int data[MAX];
int index = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *thread1(void* arg)
{
    for (int i = 0; i < MAX; )
    {
        int begin = 0;
        pthread_mutex_lock(&mutex);
        begin = index;
        index += 200;
        pthread_mutex_unlock(&mutex);
        for (int j = 0; j < 200; i += 2, j ++)
            data[begin + j] = i;
    }
    pthread_exit(NULL);
}
```

```
void *thread2(void* arg)
{
	for (int i = 0; i < MAX; )
	{
		int begin = 0;
		pthread_mutex_lock(&mutex);
		begin = index;
		index += 200;
		pthread_mutex_unlock(&mutex);
		for (int j = 0; j < 200; i += 2, j++)
			data[begin + j] = i + 1;
	}
	pthread_exit(NULL);
}

int main() {
	pthread_t tid1, tid2;

	struct timeval start_time, end_time;
	gettimeofday(&start_time, NULL);

	pthread_create(&tid1, NULL, thread1, NULL);
	pthread_create(&tid2, NULL, thread2, NULL);

	pthread_join(tid1, NULL);
	pthread_join(tid2, NULL);

	gettimeofday(&end_time, NULL);
	long elapse_time = (end_time.tv_sec - start_time.tv_sec) * 1000000 + (end_time.tv_usec -
start_time.tv_usec);

	int max_diff = 0;
	for (int i = 1; i < MAX; i++) {
		int diff = data[i] - data[i - 1];
		if (diff > max_diff) {
			max_diff = diff;
		}
	}

	printf("Max absolute difference between adjacent numbers: %d\n", max_diff);
	printf("Elapsed time=%ld\n", elapse_time);

	return 0;
}
```

运行结果如下：



可以看到这样就跟第三种方法的时间差别不大了，控制临界资源和进入临界区执行操作的次数这两个变量后，可以说明这两种方法时间开销差别不大。由此也可以认识到合理设置临界资源及每次进入临界区的次数，可以有效提高程序运行速度。

但是第三种方法仍然是平均运行时间最短的，可见由于原子操作通常是硬件级别的操作，不涉及系统调用或线程切换，它们允许非常高的并发性能，所以时间开销要更小。

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：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。

(1)

上述过程不能通过pthread_mutex_lock/unlock函数实现，因为不能保证线程1写完数组后，线程2马上读取这五个数。如果线程1写完后，线程1在线程2读取之前便对数组重新写入，那么就会对原来的5个数进行覆盖，从而导致线程2读取的数据错误。

对该题的一点补充：

本题可以通过增加一个指示应当由哪个线程执行的变量实现，实现程序如下：

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

#define CIRC_TIME 500

int turn;
int data[5];
int circ;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond;

void *thread0()
{
	pthread_cond_signal(&cond);
	int k;
	k = 0;
	srand((unsigned)time(NULL));
	while (circ < CIRC_TIME)
	{
		pthread_mutex_lock(&mutex);
		if (turn == 1)
			pthread_cond_wait(&cond,&mutex);
		for (int j = 0; j < 5; j++)
```

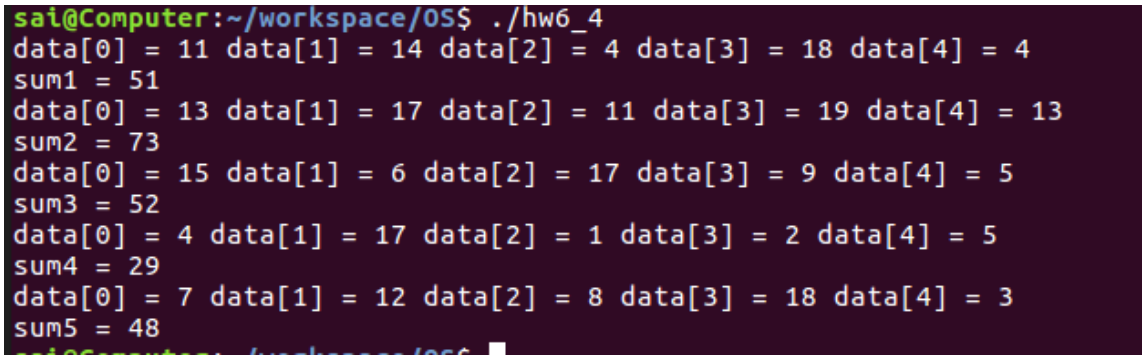
```
            data[j] = rand() % 20 + 1;
        turn = 1;
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

void *thread1()
{
    while (circ < CIRC_TIME)
    {
        int sum = 0;
        pthread_mutex_lock(&mutex);
        if (turn == 0)
            pthread_cond_wait(&cond,&mutex);
        for (int j = 0; j < 5; j++)
        {
            printf("data[%d] = %d ", j, data[j]);
            sum += data[j];
        }
        turn = 0;
        circ++;
        printf("\nsum%d = %d\n", circ, sum);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

int main()
{
    circ = turn = 0;
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, thread0, NULL);
    pthread_create(&thread[1], NULL, thread1, NULL);

    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
}
```

这个程序使用了 pthread_cond_wait() 函数，其函数原型为 pthread_cond_wait(&cond, &mutex)。其作用如下：

当线程执行到这个语句时，它会释放 &mutex 所指向的互斥锁，从而允许其他线程获得锁并访问共享资源的关键部分。同时，当前线程会进入等待状态，等待条件变量 &cond 的信号或广播。这意味着线程会暂停执行，直到其他线程发出信号或广播来唤醒它。一旦线程被唤醒，它会重新获取 &mutex 指向的互斥锁，然后继续执行后续的代码。

在这个程序中，还使用了 pthread_cond_signal(&cond) 用于唤醒一个等待的线程。

以下是程序的运行结果：


sai@Computer:~/workspace/OS$ ./hw6_4
data[0] = 11 data[1] = 14 data[2] = 4 data[3] = 18 data[4] = 4
sum1 = 51
data[0] = 13 data[1] = 17 data[2] = 11 data[3] = 19 data[4] = 13
sum2 = 73
data[0] = 15 data[1] = 6 data[2] = 17 data[3] = 9 data[4] = 5
sum3 = 52
data[0] = 4 data[1] = 17 data[2] = 1 data[3] = 2 data[4] = 5
sum4 = 29
data[0] = 7 data[1] = 12 data[2] = 8 data[3] = 18 data[4] = 3
sum5 = 48

可以看出，结果是正确的。