作业 6

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。

```
⊗ solomon@DESKTOP-23PER74:~/CODES/C/OS/homework6$ gcc peterson.c -o peterson -lpthread
peterson.c:12:5: warning: built-in function ‘index’ declared as non-function [-Wbuiltin-declaration-mismatch]
   12 | int index = 0;
      |     ^~~~~
```

1、源代码如下：

```
#define __USE_GNU
#include <sched.h>
#include <pthread.h>
```

```c
#define __USE_GNU
#include <sched.h>
#include <pthread.h>
#include <sys/time.h>
#include   <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MAXNUM 10000000
#define MAXTHREAD 2
#define TRUE   1
#define FALSE 0
int INDEX = 0;
int data[MAXNUM];
int flag[MAXTHREAD];
int turn;
int start[MAXTHREAD]={0,0};
int end[MAXTHREAD]={400,400};
//peterson pseudocode
/*
flag[i] = True;
turn = j;
while(flag[j] && turn == j);
critical section;
flag[i] = False;
remainder section;
*/
void *work(void *args){
      int thread_id = *(int *)args;
      int other_thread = 1 - thread_id;
      while (INDEX < MAXNUM) {
            flag[thread_id] = 1;
            turn = other_thread; // 转让控制权
            while (flag[other_thread] && turn == other_thread) {
                  //   忙等阶段
            }
  //------------------------------临界区--------------------------------
                  if (INDEX < MAXNUM) {
                        int i;
                        for (i = start[thread_id]; i < end[thread_id]; i+=2) {
                                    data[INDEX] = i + thread_id;
                                    INDEX ++;
                        }
                        start[thread_id] = end[thread_id];
```

```
                        end[thread_id] =   start[thread_id]+400;
                }
                else {
                        pthread_exit(NULL);
                }
  //--------------------------------临界区---------------------------------
            flag[thread_id] = 0;

      pthread_exit(NULL);
}

int main(){
      pthread_t t1,t2;
      struct timeval START,END;
      int max_diff = INT_MIN;
      long long int elapsed_time;
      int id[] = {0, 1};
      gettimeofday(&START, NULL);
      pthread_create(&t1, NULL, work, &id[0]);
      pthread_create(&t2, NULL, work, &id[1]);
      pthread_join(t1, NULL);
      pthread_join(t2, NULL);
      gettimeofday(&END, NULL);
      elapsed_time = (END.tv_sec - START.tv_sec) * 1000000 + (END.tv_usec -
START.tv_usec);
      printf("The elapsed time is %lld us in total\n", elapsed_time);
      for (int i = 0; i < MAXNUM-1; i++) {
        int diff1 = abs(data[i + 1] - data[i]);
        int diff2 = abs(data[i] - data[i + 1]);
        int diff = diff1 > diff2 ? diff1 : diff2;
        if (diff > max_diff) {
              max_diff = diff;
        }
      }
      printf("The max_diff is: %d\n", max_diff);
      printf("\n");
      return 0;
}
```

写一个 bash 脚本文件：

```
for ((i=1; i<=10; i++))
do
./peterson
done
```

运行结果如下：

```
The elapsed time is 28303 us in total
The max_diff is: 54397

The elapsed time is 26581 us in total
The max_diff is: 17997

The elapsed time is 26196 us in total
The max_diff is: 12397

The elapsed time is 26435 us in total
The max_diff is: 23597

The elapsed time is 25831 us in total
The max_diff is: 41197

The elapsed time is 26102 us in total
The max_diff is: 26797

The elapsed time is 26469 us in total
The max_diff is: 59197
```

分析：

设计上，这两个程序应该是严格的交替进行，但是根据其最大差值来看，显然并非如此，观察代码设计后，注意倘若线程 0 将 turn 置给 1 后，切换到线程 1，此时线程 1 又将 turn 置为 0，若恰好切换回线程 0，将导致线程 0 连续运行两次，所以导致结果不太符合预期中的固定值。

因此修改代码如下：

```
if((INDEX/200)%2==thread_id){
            if (INDEX < MAXNUM) {
                  int i;
                  for (i = start[thread_id]; i < end[thread_id]; i+=2) {
                              data[INDEX] = i + thread_id;
                              INDEX ++;
                  }
                  start[thread_id] = end[thread_id];
                  end[thread_id] =   start[thread_id]+400;
            }
            else {
                  pthread_exit(NULL);
            }
            }
```

加入(INDEX/200)%2==thread_id 可以判断是否存在错误抢锁的情况，这样的话就等待下一个线程来进行。

修改后的执行结果如下：

```
The elapsed time is 29176 us in total
The max_diff is: 397

The elapsed time is 26503 us in total
The max_diff is: 397

The elapsed time is 26050 us in total
The max_diff is: 397

The elapsed time is 26047 us in total
The max_diff is: 397

The elapsed time is 25269 us in total
The max_diff is: 397

The elapsed time is 26805 us in total
The max_diff is: 397

The elapsed time is 26979 us in total
The max_diff is: 397
```

结果符合预期，说明两个线程严格地交替切换并赋值正确。
通过 gdb 调试可以看到，data 数组里的赋值情况：

```
[192]: 384
[193]: 386
[194]: 388
[195]: 390
[196]: 392
[197]: 394
[198]: 396
[199]: 398
[200]: 1
[201]: 3
[202]: 5
[203]: 7
```

显然，最大差值的产生是来自于 200 次交替后，398 和 1 的差值，这从另一方面表明了，线程是严格交替进行的。

2、

```c
#define __USE_GNU
#include <sched.h>
#include <pthread.h>
#include <sys/time.h>
#include   <stdio.h>

#define MAXNUM 10000
#define MAXTHREAD 2
```

```
int data[MAXNUM];
int INDEX = 0;
pthread_mutex_t mutex ; //互斥锁
int start[MAXTHREAD]={0,0},end[MAXTHREAD]={400,400};
void* work(void* args){
	int thread_id = *(int*)args;
	while(INDEX<MAXNUM){
			pthread_mutex_lock(&mutex);   //加锁
			//--------------------------------临界区--------------------
--------------
			if(INDEX<MAXNUM){
			for(int i=start[thread_id];i<end[thread_id];i+=2) {
				data[INDEX] = i + thread_id; //even ( i+1 for thread 2)
				INDEX++;
			}
			start[thread_id] = end[thread_id];
			end[thread_id] =   start[thread_id]+400;
			}
			//--------------------------------临界区--------------------
--------------
			pthread_mutex_unlock(&mutex); //解锁
		// }
	}
	pthread_exit(NULL);
}

int main(){
	pthread_t t1,t2;
	struct timeval START, END;
	int delta=-1;
	int thread_id[] = {0, 1};
	long long int elapsed_time;
	pthread_mutex_init(&mutex, NULL);
	gettimeofday(&START, NULL);
	pthread_create(&t1, NULL, work, &thread_id[0]);
	pthread_create(&t2, NULL, work, &thread_id[1]);
	pthread_join(t1, NULL);
	pthread_join(t2, NULL);
	gettimeofday(&END, NULL);
	elapsed_time =(END.tv_usec - START.tv_usec);
	printf("The elapsed time is %lld us in total\n", elapsed_time);
	for(int i=0;i<MAXNUM-1;i++){
		if(data[i+1]-data[i]>delta)
			delta=data[i+1]-data[i];
```

```
            else if(data[i]-data[i+1]>delta)
                  delta=data[i]-data[i+1];
      }

      printf("The MAX of delta is: %d\n", delta);
      printf("the index is %d\n",INDEX);
      pthread_mutex_destroy(&mutex);
      return 0;
}
```

重复运行 mutex 程序，运行结果如下：

```
The elapsed time is 43102 us in total
The max_diff is: 3307999

The elapsed time is 47059 us in total
The max_diff is: 2616399

The elapsed time is 44976 us in total
The max_diff is: 769199

The elapsed time is 44365 us in total
The max_diff is: 3240399

The elapsed time is 32728 us in total
The max_diff is: 885997

The elapsed time is 43315 us in total
The max_diff is: 1479997
```

分析后发现，这个程序很容易出现锁住之后不会切到另一个线程，一直是同一个线程抢占锁，给数组赋值，最终导致给数组赋值的结果不正确，出现不稳定的差值。

对线程锁函数调研：

pthread_mutex_lock(pthread_mutex_t *mutex)：这个函数用于获取一个互斥锁。**如果该互斥锁已经被其他线程占用，则当前线程会被阻塞等待**。如果互斥锁已经被锁住，调用这个函数的线程阻塞直到互斥锁可用为止。

pthread_mutex_unlock(pthread_mutex_t *mutex)：这个函数用于释放一个互斥锁，这样其他线程就可以获取这个互斥锁 3。如果调用 pthread_mutex_unlock()时有多个线程被互斥锁对象阻塞，则互斥锁变为可用时调度策略可确定获取该互斥锁的线程。

注意到：thread_mutex_unlock()函数释放有参数 mutex 指定的 mutex 对象的锁。如果被释放取决于该 Mutex 对象的类型属性。如果有多个线程为了获得该 mutex 锁阻塞，调用 pthread_mutex_unlock()将是该 mutex 可用，一定的调度策略将被用来决定哪个线程可以获得该 mutex 锁。（在 mutex 类型为 PTHREAD_MUTEX_RECURSIVE 的情况下，只有当 lock

count 减为 0 并且调用线程在该 mutex 上已经没有锁的时候）如果一个线程在等待一个 mutex 锁得时候收到了一个 signal,那么在从 signal handler 返回的时候，该线程继续等待该 mutex 锁，就像这个线程没有被中断一样。

如何避免这样的情况呢？当一个线程执行到 pthread_mutex_lock 处时，如果该锁此时被另一个线程使用，那此线程被阻塞，即程序将等待到另一个线程释放此互斥锁。

- 我们可以使用 pthread_delay_np 函数，让线程睡眠一段时间，防止一个线程始终占据此函数。
- 也可以使用函数 pthread_mutex_trylock，它是函数 pthread_mutex_lock 的非阻塞版本，当一个线程尝试获取锁失败时并不会加入到阻塞队列中去。
- 也可以使用第一题的方法检测进入的线程是否为对应的线程。

此处用最简单的判定方法，也就是第三种方法，另外两种方法仍有可能会导致线程不会被调度。同样将暂存区内的代码修改为

```
                if((INDEX/200)%2==thread_id){
                        if(INDEX<MAXNUM){
                        for(int i=start[thread_id];i<end[thread_id];i+=2)
{
                                data[INDEX] = i + thread_id;
                                INDEX++;
                        }
                        start[thread_id] = end[thread_id];
                        end[thread_id] =   start[thread_id]+400;
                        }
                }
```

修改后，运行结果如下：

```
The elapsed time is 147825 us in total
The max_diff is: 397

The elapsed time is 152621 us in total
The max_diff is: 397

The elapsed time is 136585 us in total
The max_diff is: 397

The elapsed time is 137910 us in total
The max_diff is: 397

The elapsed time is 141153 us in total
The max_diff is: 397

The elapsed time is 169869 us in total
The max_diff is: 397

The elapsed time is 154784 us in total
```

（3）、

调研原子操作的相关函数：__sync_fetch_and_add 系列一共有十二个函数，

分别为：

type __sync_fetch_and_add (type *ptr, type value);
type __sync_fetch_and_sub (type *ptr, type value);
type __sync_fetch_and_or (type *ptr, type value);
type __sync_fetch_and_and (type *ptr, type value);
type __sync_fetch_and_xor (type *ptr, type value);
type __sync_fetch_and_nand (type *ptr, type value);
type __sync_add_and_fetch (type *ptr, type value);
type __sync_sub_and_fetch (type *ptr, type value);
type __sync_or_and_fetch (type *ptr, type value);
type __sync_and_and_fetch (type *ptr, type value);
type __sync_xor_and_fetch (type *ptr, type value);
type __sync_nand_and_fetch (type *ptr, type value);

gcc 4.1.2 版本之后，对 X86 或 X86_64 支持内置原子操作。这些函数提供了跨平台的原子操作，可以保证在多线程环境下对共享变量的操作是原子的，不会数据竞争。它们通常用于编写多线程程序，确保共享资源的一致性和完整性。

这些内置函数执行原子比较和交换，如果 *ptr 的当前值是 oldval，则将 newval 写入 *ptr。如果比较成功，type 的版本会返回*ptr 原先的值。

参考链接

https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html

可以直接利用原子操作来实现一个锁机制，但是也可以直接利用原子操作来进行加操作以避免被另外一个线程打断，这需要使用两个不同的函数来完成以确保它们的局部变量是不同的而不会因为打断而错误累加。以下实现的是一个简单的直接通过原子操作进行加法的程序，但是它并不能保证两个线程交替进行， data 中的值分布是随机的。

源代码如下：

```
#define __USE_GNU
```

```c
#include <sched.h>
#include <pthread.h>
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MAXNUM 10000000

int data[MAXNUM];
int INDEX = 0;

void* write_even(void *arg)
{
	for(int i=0; i<MAXNUM; i+=2) {
		int idx = __sync_fetch_and_add(&INDEX, 1);
		data[idx] = i;
	}
	return NULL;
}

void* write_odd(void *arg)
{
	for(int i=1; i<MAXNUM; i+=2) {
		int idx = __sync_fetch_and_add(&INDEX, 1);
		data[idx] = i;
	}
	return NULL;
}

int main(void)
{
	pthread_t thread1, thread2;
	int max_diff = INT_MIN;
	struct timeval START, END;
	long long int elapsed_time;
	gettimeofday(&START, NULL);
	pthread_create(&thread1, NULL, write_even, NULL);
	pthread_create(&thread2, NULL, write_odd, NULL);

	pthread_join(thread1, NULL);
	pthread_join(thread2, NULL);
	gettimeofday(&END, NULL);
```

```
        elapsed_time = (END.tv_sec - START.tv_sec) * 1000000 + (END.tv_usec -
START.tv_usec);
        printf("The elapsed time is %lld us in total\n", elapsed_time);
          // 计算数组相邻两个数的最大绝对差值
          for (int i = 0; i < MAXNUM-1; i++) {
          int diff1 = abs(data[i + 1] - data[i]);
          int diff2 = abs(data[i] - data[i + 1]);
          int diff = diff1 > diff2 ? diff1 : diff2;
          if (diff > max_diff) {
                max_diff = diff;
          }
        }
        printf("The max_diff is: %d\n", max_diff);
        printf("\n");

        return 0;
}
```

在这个程序中，临界区是__sync_fetch_and_add(&INDEX, 1)，因为这是一个原子操作，它同时读取和更新 INDEX 的值。由于这个操作是原子的，所以不需要使用互斥锁或其他同步机制来保护这个临界区。

运行结果如下：

```
The elapsed time is 215042 us in total
The max_diff is: 423985

The elapsed time is 321272 us in total
The max_diff is: 176133

The elapsed time is 106481 us in total
The max_diff is: 412605

The elapsed time is 230506 us in total
The max_diff is: 407209

The elapsed time is 330136 us in total
The max_diff is: 318415

The elapsed time is 193620 us in total
The max_diff is: 232285

The elapsed time is 146038 us in total
```

分析：

理论上来说速度应该如下： 原子操作 > Peterson 算法 > Mutex 锁。原因是因为互斥锁实现是基于系统调用，其属于例外，其间会经历内核态和用户态的切换，导致耗时大大增加。而原子操作和 Peterson 算法更简洁， 原子操作是基于特定硬件指令实现的操作，它们可以直接由硬件执行，并且不需要系统调用或者线程上下文切换。Peterson 算法是一种基于软件的解决方案，它不依赖特定的硬件指令，可能在编译时可以很好的优化。

但是现在测试的结果的耗时：原子操作 > Mutex 锁> Peterson 算法，猜测可能的原因是原子操作通常是由硬件直接支持的，因此它的性能会受到硬件平台和操作系统的影响。如果硬件平台可能没有提供原子操作的硬件支持，这就需要通过软件模拟来实现，这会增加额外的开销。即原子操作的内存屏障特性导致运行时的流水线未能很好地进行优化调度。在现代处理器中，指令可能会被重新排序以提高性能。这种重排可能发生在编译器层面（编译器的指令重排），也可能发生在硬件层面（CPU 的指令重排，也称为乱序执行）。指令重排需要阻止编译器和处理器的指令重排以保证正确性。这可能会限制编译器的优化能力，从而影响性能。同时原子操作需要确保在多线程环境下对共享数据的正确访问，这通常需要插入内存屏障（memory barrier）。内存屏障会阻止某些内存访问的重排，从而确保数据的一致性。然而，插入内存屏障会增加额外的开销，从而影响性能。

该部分参考连接：

https://zhuanlan.zhihu.com/p/611868395

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。

可以通过锁机制实现，用 pthread_mutex_lock/unlock 这两个函数可以用来创建一个互斥锁，以保证在任何时刻只有一个线程可以访问数组。但是由于线程调度的不确定性，可能会出现读线程在写线程写入数据之前就开始读取数据的情况。为了避免这种情况，可以使用条件变量来确保读线程在写线程写入数据之后才开始读取数据。

因此仅靠 pthread_mutex_lock/unlock 函数是无法保证结果的正确性的。这是因为互斥锁只能保证在任何时刻只有一个线程可以访问共享资源，但不能控制线程的执行顺序。线程的调度顺序是由操作系统决定的，我们无法控制。

以下是一个使用条件变量（pthread_cond_wait 和 pthread_cond_signal 函数）来实现该操作的程序：

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

#define ARRAY_SIZE 5
#define MAX_RANDOM_VALUE 20

int array[ARRAY_SIZE];
pthread_mutex_t mutex;
pthread_cond_t cond;
int written = 0;
```

```c
void *write_to_array(void *arg) {
	for (int i = 0; i < 5; i++) {
		pthread_mutex_lock(&mutex);
		for (int j = 0; j < ARRAY_SIZE; j++) {
			array[j] = rand() % MAX_RANDOM_VALUE + 1;
		}
		written = 1;
		pthread_cond_signal(&cond);
		pthread_mutex_unlock(&mutex);
	}
	return NULL;
}

void *read_and_sum_array(void *arg) {
	for (int i = 0; i < 5; i++) {
		pthread_mutex_lock(&mutex);
		while (!written) {
			pthread_cond_wait(&cond, &mutex);
		}
		int sum = 0;
		for (int j = 0; j < ARRAY_SIZE; j++) {
			sum += array[j];
		}
		printf("Sum: %d\n", sum);
		written = 0;
		pthread_mutex_unlock(&mutex);
	}
	return NULL;
}

int main() {
	srand(time(NULL));
	pthread_t writer, reader;

	pthread_mutex_init(&mutex, NULL);
	pthread_cond_init(&cond, NULL);

	pthread_create(&writer, NULL, write_to_array, NULL);
	pthread_create(&reader, NULL, read_and_sum_array, NULL);

	pthread_join(writer, NULL);
	pthread_join(reader, NULL);
```

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

    return 0;
}
```

运行该程序，发现程序不会结束，只会输出一次结果，但是调试的时候却可以输出五次结果。

```
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework6$ ./random
Sum: 60
```

```
Sum: 69
Sum: 57
Sum: 49
Sum: 48
Sum: 47
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-ag5lyikn.shr" 1>"/tmp/Microsoft-MIEngine-Out-r0h5
```

分析原因可能是因为，写入线程可能在读取线程开始等待之前就已经完成了写入并发送了信号。那么读取线程就会错过这个信号，并且会一直等待下去，因为它没有收到信号。导致程序只输出一次结果然后就停止了。

调试时程序能够正常运行可能是因为调试器改变了线程的执行顺序。当你单步执行或设置断点时，调试器会暂停一个或多个线程，这可能会导致写入线程和读取线程按照你期望的顺序执行。

如何进行修改呢？可以使用两个条件变量：一个用于通知写入完成，另一个用于通知读取完成。这样，写入线程和读取线程就可以按照正确的顺序交替执行了。

(以下部分参考 Bing AI)

修改后的源代码如下：

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

#define ARRAY_SIZE 5
#define MAX_RANDOM_VALUE 20

int data[ARRAY_SIZE];
pthread_mutex_t mutex;
pthread_cond_t cond_writer, cond_reader;
int written = 0;

void *write_to_array(void *arg) {
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&mutex);
        while (written) {
            pthread_cond_wait(&cond_writer, &mutex);
```

```
            }
            for (int j = 0; j < ARRAY_SIZE; j++) {
                data[j] = rand() % MAX_RANDOM_VALUE + 1;
            }
            written = 1;
            pthread_cond_signal(&cond_reader);
            pthread_mutex_unlock(&mutex);
        }
        return NULL;
}

void *read_and_sum_array(void *arg) {
        for (int i = 0; i < 5; i++) {
            pthread_mutex_lock(&mutex);
            while (!written) {
                pthread_cond_wait(&cond_reader, &mutex);
            }
            int sum = 0;
            for (int j = 0; j < ARRAY_SIZE; j++) {
                sum += data[j];
            }
            printf("Sum: %d\n", sum);
            written = 0;
            pthread_cond_signal(&cond_writer);
            pthread_mutex_unlock(&mutex);
        }
        return NULL;
}

int main() {
        srand(time(NULL));
        pthread_t writer, reader;

        pthread_mutex_init(&mutex, NULL);
        pthread_cond_init(&cond_writer, NULL);
        pthread_cond_init(&cond_reader, NULL);

        pthread_create(&writer, NULL, write_to_array, NULL);
        pthread_create(&reader, NULL, read_and_sum_array, NULL);

        pthread_join(writer, NULL);
        pthread_join(reader, NULL);

        pthread_cond_destroy(&cond_writer);
```

```
    pthread_cond_destroy(&cond_reader);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

该程序运行结果如下：

```
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework6$ ./random
Sum: 43
Sum: 37
Sum: 51
Sum: 66
Sum: 60
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework6$ ./random
Sum: 65
Sum: 50
Sum: 44
Sum: 77
Sum: 55
```

发现其正确地输出了结果，表明读写线程正确地交替执行。