中山大学移动信息工程学院本科生实验报告

（2017 学年春季学期）

课程名称：**Operating System**　　任课教师：**饶洋辉**　　**批改人(此处为 TA 填写)：**

年级+班级	**1506**	专业（方向）	**移动信息工程**
学号	**15352116**	姓名	**洪子淇**
电话	**13726205766**	Email	**1102229410@qq.com**
开始日期	**2017/5/31**	完成日期	**2017/6/2**

1. 实验目的

a) 理解进程和线程的含义。

b) 理解多线程的含义，以及使用多线程的好处。

c) 掌握创建一个线程的基本方法以及对于互斥量的使用。

2. 实验过程

（一）实验思路简述

实验一：

实验要求创建 4 个子线程，每个线程按顺序轮流输出 5 个数字，直到 100 个数字按顺序输出完毕。

首先声明四个线程，每个线程都调用了 join 函数，表示主线程等待子线程运行结束后再继续。每个线程调用了一个 test 的输出函数，用来对数字进行输出，为了保证输出 100 时结束，设置一个全局变量。又为了保证当前仅有一个线程运行，使用互斥量，当一个线程进入 test 函数时，获得互斥量，然后进入一个循环，

当输出数的个数达到 5 个的时候，然后再将互斥量进行释放。这样就可以实现每次只有一个线程在输出数据而且是按顺序执行的。

实验二：

实现 100000 个随机数的双线程归并排序并与单线程比较。归并排序就是指不断地把一个数组分成两组，当最后分到只有一个数地时候开始沿着之前的路径往上合并成一个有序的数组。即可以理解为先使每个子序列有序，再使自诩列段减有序的指令合并成一个有序表。而双线程和单线程的归并排序不同在于，只能对一个数组进行归并，而双线程可以同时对两个数组进行排序。所以双线程理论上应该比单线程的排序方法快。

（二）伪代码

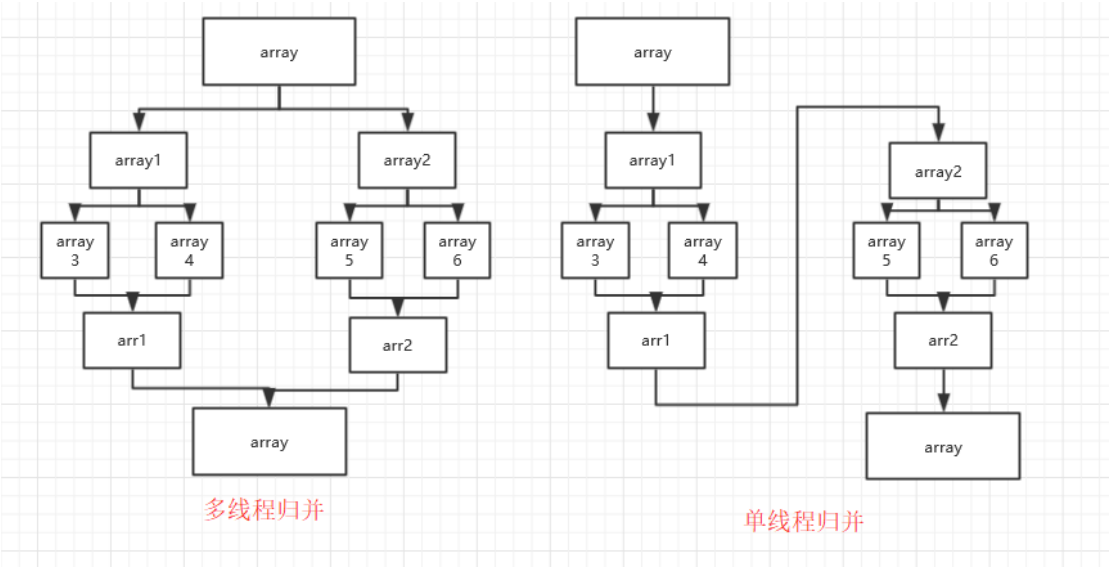
实验一（伪代码）：

```
Main（void）
{       count = 0；
            create 5 subthreads；
            subthread exeute the test function；
}
Test（void）
{
        ask lock；
            while   (true) {
                if (count >= 100)
                    break;
                for（int i=0; i < 5; i++)
                    printf count++；
             }
        release lock;
}
```

实验二(伪代码以及流程图)：

```
//一个数组合并
Merge_array（array[ ], first,mid, last, temp[ ]）
{
    i = first;   j= mid + 1;   k = 0;
    while (i <= first && j <= last ){
        If (array[i] < array[j])
            temp[k++] = array[i];
        else temp[k++] = array[j];
    while(i <= mid)   temp[k++] = array[i++];
    while(j <= last)   temp[k++] = array[j++];

    for(i = 0; i < last; i++){
        array[i] = temp[i];
    }
}
}
//归并排序
Merge_sort (array[ ], first, last, temp[ ])
{
     if(first < last)
    {
      mid = (first + last) / 2;
        merge_sort(array[ ], first, mid, temp);
        merge_sort(array[ ], mid + 1, last, temp);
        merge_array(array, first, mid, last, temp);
    }
}
//两个数组合并
merge_two_thread(arr1[ ], size1, arr2[ ], size2, result)
{
     i= j = k = 0;
     while (i < size1 && j < size2) {
        If (arr1 [i] < arr2 [j])
            result [k++] = arr1 [i];
        else result [k++] = arr2 [j];
     while(i <= mid)   result [k++] = arr1 [i++];
     while(j <= last)   result [k++] = arr2 [j++];
}
```



（三）具体实现

实验一：

主函数：

```
int main()                                                    主函数
{
    cout<<"Main thread begin"<<endl;
    vector<thread> threadset;
    count = 0;

    for (int i = 0; i < 4; ++i)                               创建4个测试线程
    {
        threadset.push_back(thread(test));
    }
    for (auto& subthread : threadset)                         保证执行完测试线程之后再执行主
        subthread.join();                                     函数，避免产生非法空间的访问

    cout<<"Main thread finished"<<endl;
}
```

线程函数：

```
void test()                                                   线程函数
{
    while(1)
    {
        loc.lock();          获得互斥量，保证当前只有一个线程运行
        if(count>=100){
            loc.unlock();避免最后没有释放锁，导致阻塞其他申请该锁的线程
            break;
        }
        for(int i=0;i<5;i++)
            cout << "thread " << this_thread::get_id() << " :    " << ++count << endl;
        loc.unlock();        输出5个数字之后就释放锁给其他线程
    }
}
```

根据注释可知，主线程主要创建 4 个测试线程，测试线程的功能是输出 1~100 个数字，其中为了避免主线程运行完释放空间后，测试线程还没有运行完造成非法空间的访问的结果，使用了 join 函数。又为了保证只有一个线程在运行，利用了互斥量，一个线程获得互斥量之后输出 5 个数，才会解锁，把锁让给其他线程。等到下一次释放其他线程释放锁之后他才能申请到锁继续输出。

实验二：

比模板多了一个 is_sort 函数，检查数组是否排好序。

merge_array 与 merge_two_thread 函数的实现方法一致，只不过前者是对一个数组的两段进行合并，而后者是直接对两个排好序的数组进行合并，只列出前者的代码：

```
void merge_array(int a[], int first, int mid, int last, int temp[])
{
    int i = first, j = mid + 1;
    int k = 0;
    while (i <= mid && j <= last) {
        if (a[i] < a[j])
        {
            temp[k++] = a[i++];
        }else
            temp[k++] = a[j++];
    }
    //前半段多余
    while(i <= mid)
        temp[k++] = a[i++];
    //后半段多余
    while(j <= last)
        temp[k++] = a[j++];

     for (i = 0; i < k; i++)
        a[first + i] = temp[i];

}
```

merge_sort 函数就是不断地划分数组，直到数组划分成只有一个数的地时候再往上进行数组合并。

这个实验中关键在于 merge_start 地实现，首先是新建两个临时的数组，用于存储双线程排序的数组。还新建一个用于合并之后的数组。在实现双线程中，首

先将要排序的数组分成两个子集，每个调用 merge_sort 函数进行排序，最后再将两个排好序的数组使用 merge_two_thread 进行合并。单线程则是直接调用 merge_sort 函数进行排序。（关键代码如下）

```
clock_t start_time = clock();

for(int i = 0; i < size / 2; i++)
    sa1[i] = array[i];
for(int i = size / 2; i < size; i++)
    sa2[i - (size / 2)] = array[i];


thread a(merge_sort, sa1, 0, size / 2 - 1, p1);
thread b(merge_sort, sa2, 0, size - size / 2 - 1, p2);
a.join();
b.join();

memset(sa, 0, size + 1);
//排完之后将两个有序的数组进行合并
merge_two_thread(sa1, size / 2, sa2, size - (size / 2), sa);

clock_t double_end_time = clock();
//***************double************//
```

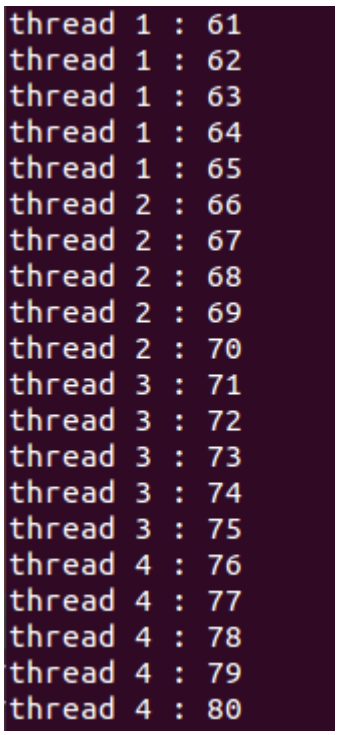
3. 实验结果

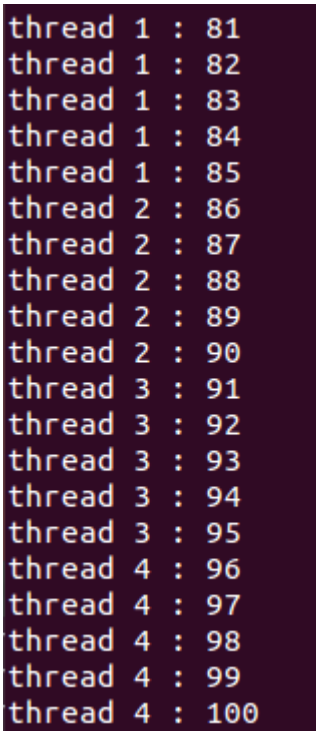
实验一：

```
thread 1 : 1
thread 1 : 2
thread 1 : 3
thread 1 : 4
thread 1 : 5
thread 2 : 6
thread 2 : 7
thread 2 : 8
thread 2 : 9
thread 2 : 10
thread 3 : 11
thread 3 : 12
thread 3 : 13
thread 3 : 14
thread 3 : 15
thread 4 : 16
thread 4 : 17
thread 4 : 18
thread 4 : 19
thread 4 : 20
```

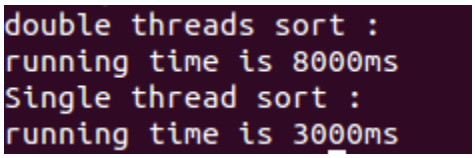
```
thread 1 : 21
thread 1 : 22
thread 1 : 23
thread 1 : 24
thread 1 : 25
thread 2 : 26
thread 2 : 27
thread 2 : 28
thread 2 : 29
thread 2 : 30
thread 3 : 31
thread 3 : 32
thread 3 : 33
thread 3 : 34
thread 3 : 35
thread 4 : 36
thread 4 : 37
thread 4 : 38
thread 4 : 39
thread 4 : 40
```

```
thread 1 : 41
thread 1 : 42
thread 1 : 43
thread 1 : 44
thread 1 : 45
thread 2 : 46
thread 2 : 47
thread 2 : 48
thread 2 : 49
thread 2 : 50
thread 3 : 51
thread 3 : 52
thread 3 : 53
thread 3 : 54
thread 3 : 55
thread 4 : 56
thread 4 : 57
thread 4 : 58
thread 4 : 59
thread 4 : 60
```





实验二：



理论上，多线程的时间应该比单线程的时间快一倍左右，可是由于不是多核的处理器，对于多线程的函数，其实采用的是单线程的执行方式，所以导致了多线程的时间比单线程大得多。

4. 回答问题

1. 用自己的话简述多线程编程时需要注意的问题。

   ①对锁的运用，需要考虑一个线程获得锁之后一定要释放锁。否则容易发生死锁现象。

   ②使用 join 函数。join 函数表示的是当子线程运行结束之后才运行主线程，这样就避免了主线程先运行完，释放空间之后，由于子线

程的空间被释放，导致了子线程空间的非法访问的后果。

③在多线程中，多个线程创建之后是同时运行线程的函数的，不是一个线程运行，另一个线程在等待 cpu 的情况，而是多个线程并行运行。

2. 用自己的话简述多线程编程为什么可以提高 CPU 的利用率?

在第二个实验中可以明显观测到多线程可以提高效率。这是由于多线程可以同时进行工作，来提高了 cpu 的利用率。举个例子就是体育考试，一个跑道上大概可以测试 8 个人的跑步成绩（这里的容量相当于 cpu 利用率高的多线程编程最大的线程数目),有 32 个人要测试，那么总共总共只需要测试 4 次，如果是单线程，则要测试 32 次，所以大大地提高了效率（cpu 的利用率)。

3. 简述如何在多线程编程的环境下解决生产者消费者问题。

生产者消费者主要是多线程中共享变量的问题。在多线程中生产者生产产品（变量增加)，消费者消费产品（变量减少)。而共享的变量一般是有额度的。当达到额度的时候，生产者将不能再生产产品，而当额度小于 0 时，消费者也不能消费产品。

所以我们需要保证生产者是一个一个进行生产，一旦生产者在生产的时候额度已经达到，其他生产者就不能进行生产（互斥)。同时也要保证生产者在生产的过程中，消费者不能进行消费，主要是在并发执行的时候信号量的计数可能会出错（互斥)。还要保证生产者和消费者对于产品个数的同步。

根据思路分析，我们需要声明一个产品的计数共享变量并且设置

下限和上限（EMPTY，FULL），还有一个用于锁操作的信号量 mutex，需要初始化为 1，表示这把锁没有被拿。

<table>
<tr><td colspan="2">Lock mutex = 1；//initial
Condition FULL;
Condition EMPTY;
resource = 0;</td></tr>
<tr><td>生产者：
while(true){
wait(EMPTY);
wait(mutex);

resource++;

signal(mutex);
signal(FULL);
}
}</td><td>消费者：
while(true){
wait(FULL);
wait(mutex);

resource--;

signal(mutex);
signal(EMPTY);
}
}</td></tr>
</table>

5. 实验感想

和之前的实验相比较，有很大的不同，首先就是之前的实验中，创建一个线程，首先要比较其优先级，看是否具备抢占能力才能运行他的函数。主要是每次只能运行一个线程。而在这次实验中，不再涉及到线程的优先级的问题。而是从执行的线程的数目的增加去提高 cpu 的利用率。一个时刻是可以并发运行多个线程的。我觉得主要是实验的侧重点转移了，在多线程中其实也会考虑到优先级的问题，实验从简单的开始入手。