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	1501	专业（方向）	移动信息工程
学号	15352006	姓名	蔡丽芝
电话	13538489980	Email	314749816@qq.com
开始日期	2017/5/31	完成日期	2017/6/1

1. 实验目的

(1) 编写代码，4 个子线程，一共输出 100 个数，第一个子线程输出 1~5，第二个子线程输出 6~10，第三个子线程输出 11~15，第四个子线程输出 16~20，然后又是第一个子线程输出 21~25，再到第二个子线程…，直到 100 个数 按顺序 输出完毕。

(2) 编写代码，实现 100000 个随机数的双线程归并排序。（提供模板）

2. 实验过程

（一）实验思路简述

(1) 实验一

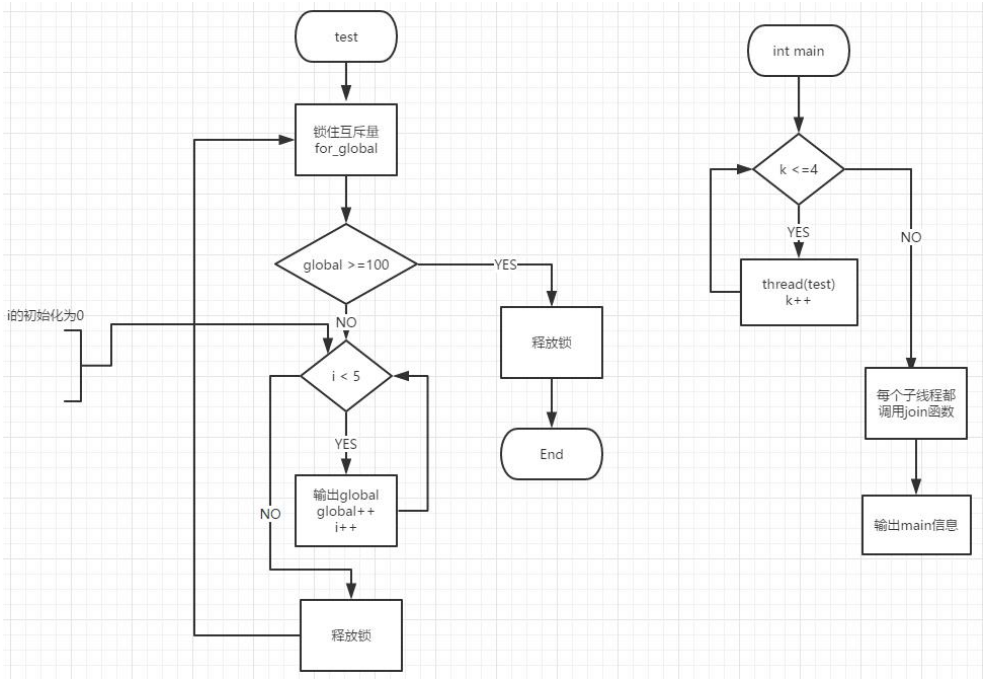
实验思路：声明一个全局互斥量 for_global，全局变量 global 用于记录线程要输出的数字。在 int main 函数里创建 4 个线程，每个线程调用执行函数 test，在这个过程中，第一个线程获得锁，从输出数据到释放锁之前，其他线程会被阻塞，直到前面的线程释放锁后，其他相应的线程才能获得锁，从而输出数据。余下的子线程执行情况和上述一致，因此会按照相应一定的顺序输出数据

(2) 实验二

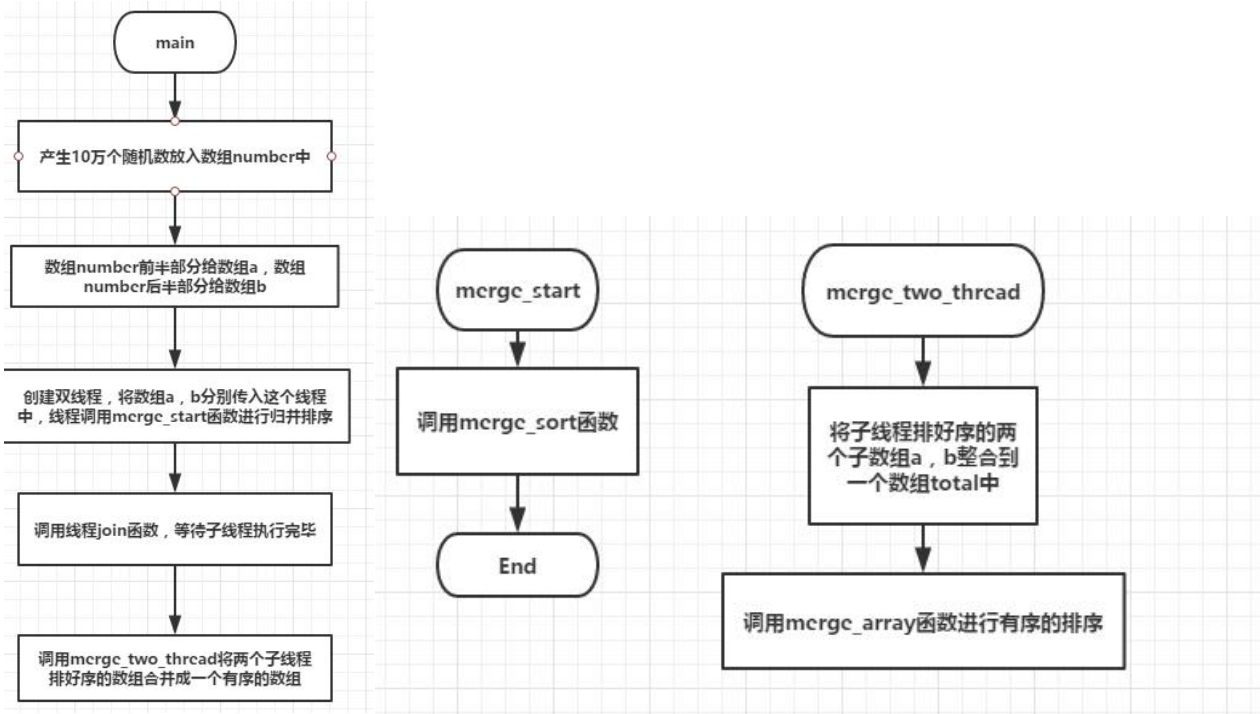
实验思路：int main 函数产生 10 万个随机的数字放入数组 number 中，创建两个子线程，将数组 number 分成两部分，分别给两个子线程进行归并排序，两个子线程排完序后，再将这两个局部有序的部分，调用 merge_array 函数归并即可将这 10 万个数字有序排序。

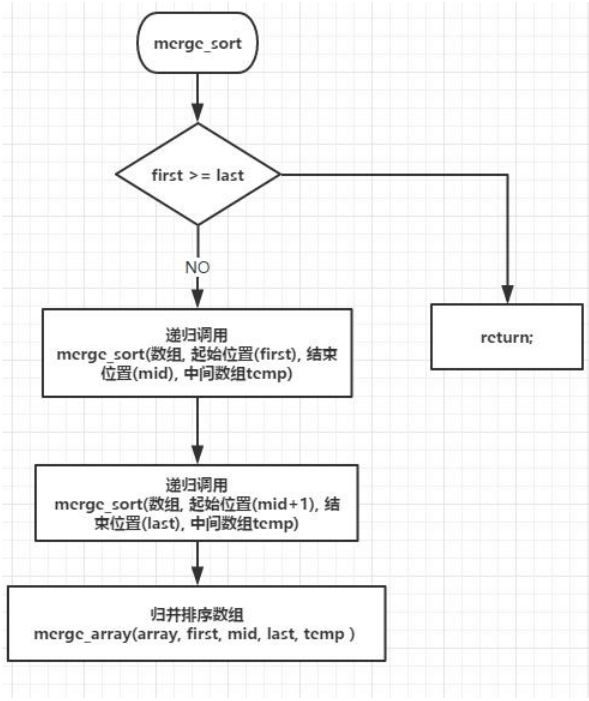
（二）伪代码

(1) 实验一流程图为：


test
锁住互斥量
for_global
global >=100
YES
NO
i的初始化为0
i < 5
YES
NO
输出global
global++
i++
释放锁
释放锁
End
int main
k <=4
YES
NO
thread(test)
k++
每个子线程都
调用join函数
输出main信息

(2)实验二流程图:


main
产生10万个随机数放入数组number中
数组number前半部分给数组a，数组
number后半部分给数组b
创建双线程，将数组a，b分别传入这个线程
中，线程调用mergc_start函数进行归并排序
调用线程join函数，等待子线程执行完毕
调用mergc_two_thread将两个子线程
排好序的数组合并成一个有序的数组
mergc_start
调用mergc_sort函数
End
mergc_two_thread
将子线程排好序的两
个子数组a，b整合到
一个数组total中
调用mergc_array函数进行有序的排序



Merge_array 的伪代码：

```
Merge_array(数组 a[], 起始位置，中间位置，结束位置，中间数组 temp [])
{
      i = 起始位置
      j = 中间位置的后一个位置
      k = 0
      while( i <= 中间位置并且 j <= 结束位置)
      {
          If(a[i] 小于 a[j])
          {
             temp[k] <= a[i];
             i++;       //i 后移一位
             k++;       //k 后移一位
          }
          else
          {
             temp[k] <= a[j];
             j++;   //j 后移一位
             K++;
          }
      }
     while(i <= mid)                                  // 前半部分数组剩余
         Temp[k++] <= a[i++]
     while(j<=mid)                                    // 后半部分数组剩余
         Temp[k++] <= a[j++]
     for t =0 到 t = k                                // 将排好序的 temp 数组拷贝给数组 a
      a[起始位置 + t] = temp[t];
}
```

（三）**具体实现**

(1) 实验一

```
int main() {
    global = 1;
    cout << "Main thread begin " << endl;
    vector<thread> threadset;                    //子线程容器
    for(int k = 1; k <= 4; k++)
    {
        threadset.push_back(thread(test));

    }

    for(auto& subthread : threadset)             //子线程执行完后，
       subthread.join();

    cout << "Main thread finish" << endl;
    return 0;
}
```

创建 4 个子线程，每个子线程都执行 test 函数，每个线程创建即被执行

```
void test()
{

    while(1)
    {
        for_global.lock();
        if(global >= 100)
        {
            for_global.unlock();
             break;
        }

        cout << "thread " <<this_thread::get_id() <<" ";
        for(int j = 1; j <= 5; j++)
        cout << global++ << " " ;
        cout << endl;

        //每输出4行，空一行
        cnt++;
        if(cnt % 4 == 0)
          cout << endl;

        for_global.unlock();
    }
}
```

test 函数会进入一个 while 循环，然后锁住互斥量，注意一点：这里首先要进行判断，如果全局变量 global 的值大于 100，就得释放锁。否则，直到该线程输出完 5 个依次递增的数据后，才能释放锁。

(2) 实验二

```
//将数组number分为两部分，前部分给数组a，后部分给数组b
for(int i = 0; i < n/2; i++)
{
    a[k] = number[i];
    b[t] = number[n/2 + i];
    k++;
    t++;
}
```

解释如注释

```
//创建双线程
thread t1(merge_start, a, n/2);
thread t2(merge_start, b, n/2);

t1.join();
t2.join();

//归并俩个数组
merge_two_thread(a, n/2, b, n/2, number) ;
```

创建两个子线程，调用 merge_start 函数，线程创建就会运行，调用线程的 join 函数，使得子线程执行完之后，主线程才会执行。

merge_start 函数

```
void merge_start(int array[],int size)
{
   int* temp = new int[size+1];
   int first = 0;
   int last = size-1;
   merge_sort(array, first, size, temp);
   delete []temp;
}
```

将要归并的数组等参数传入，调用 merge_sort 函数.

```
void merge_sort(int array[], int first, int last, int temp[])
{
    if(first >= last) return;
    int mid = (first + last) / 2;
    merge_sort(array, first, mid, temp);              //拆分
    merge_sort(array, mid + 1, last, temp);           //拆分
    merge_array(array, first, mid, last, temp);       //合并

}
```

对要排序的数组通过递归的方式，分割成足够小的子序列，再比较合并。

```
void merge_array(int a[], int first, int mid, int last, int temp[])
{
    int  i = first;                       // 将i的初始值设为起始位置
    int  j = mid + 1;                     // 将j的初始值设为中间位置的后一个
    int  k = 0;
    while(i <= mid && j <= last)          // 比较两部分对应的大小
    {                                     // 有序地放入tmep数组中
        if(a[i] < a[j])
            temp[k++] = a[i++];           // i要后移一位
        else
            temp[k++] = a[j++];           // j要后移一位
    }

    //将前半部分剩下的元素直接放入temp数组中
    while(i <= mid)
        temp[k++] = a[i++];

    //将后半部分剩下的元素直接放入temp数组中
    while(j <= last)
        temp[k++] = a[j++];

    //将已经排好序temp数组赋值给数组a
    for(int t = 0; t < k; t++)
        a[first + t] = temp[t];
}
```

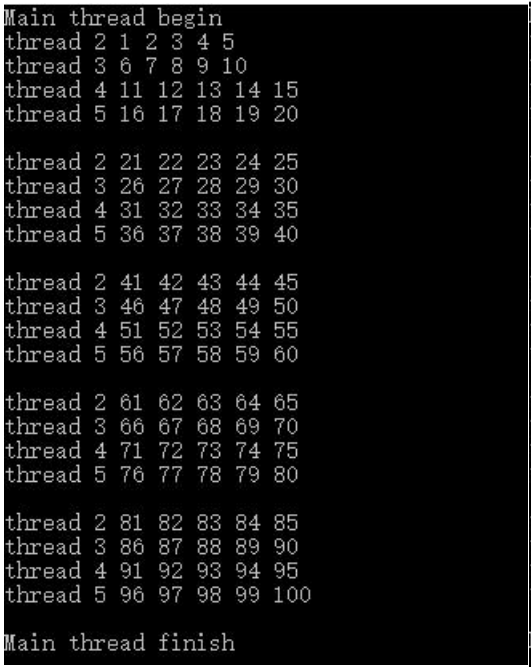
代码解释如注释

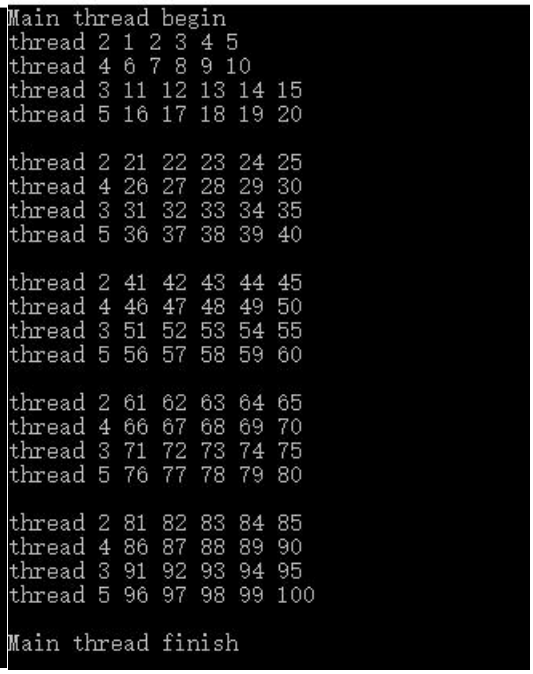
```
bool isSorted(int array[], int size)
{
    for(int i = 0; i < size-1; i++)
    {
        if(array[i] > array[i+1])
         return false;
    }
    return true;
}
```

判断数组是否有序，依次遍历，如果出现前一个数大于后一个数，就输出 false；

3. 实验结果

(1) 实验一





多次运行程序，子线程的执行顺序大多是有序的，即按照 id 为 2，3，4，5 的顺序，但是偶尔会出现乱序的现象如第二张图，我认为原因是：4 个线程几乎是同时创建完毕的，线程创建即被执行，但是线程从创建到申请锁的时间是无法确定的，所以可能会出现无序的状态。

解决尝试：首先我思考过用休眠的方式，让每个线程能在不同的时间唤醒，从而保证获取线程申请锁的顺序是固定的，但是失败告终。

(2) 实验二

双线程排序花费时间

明显双线程花费时间少

由于数据较多，因此写了一个判断函数判断排序是否正确，排好序的数组元素输出在文件 result.txt 中，TA 可打开查看。

4. 回答问题

- 1. 用自己的话简述多线程编程编程时需要注意的问题(参考百度)
- 答：①要协调好每个线程的优先级，一些重要的线程比如说控制类的线程优先级要高于一些基础的 Worker 类线程。比如一个网站，用户点击一个按钮，页面线程的优先级如果低于后台运行的 worker 线程，由于 worker 线程复杂所以需运行较长时间，此时页面就可能较长时间没有反应，这样会使得用户体验效果很差，以为自己没有操作而进行多次重复操作。因此页面线程的优先级需高于后台的 worker 线程。

  ② 如果涉及到公共资源的使用需要用到锁的时候，要注意避免出现死锁

  ③ 要合理分配每个线程需要执行的任务量，一些对响应速度要求高的线程执行的任务量应该比较少，而一些对响应速度要求不高的线程即可相应分配多些任务。

④ 注意信号量的使用，更高地解决同步问题。

- 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？
- 答：因为首先线程与其他共同属于同一个进程的线程共享资源，所以在创建和切换的时候开销会远远小于创建和切换一个线程，并且共享资源意味着可以让更多的线程拥有同一份资源，使得通信速度快，这样的话，可以更大程度地重复利用计算机资源，从而提高 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

答：一个生产者对应一个生产者线程，会存在多个生产者，也就是说会存在多个生产者线程，同理，也会有多个消费者线程，这时候就涉及到公共资源的访问问题，所以需要使用互斥锁。假设缓冲区最大为 n，对于生产者来说：当生产的数量使得缓冲区为满的时候，这时生产者不允许继续生产，必须被阻塞直到缓冲区不为满。对于消费者来所，当消费的数量使得缓冲区为空的时候，这时候消费者不允许继续消费，必须被阻塞直到缓冲区不为空。

可以使用信号量的方法：设置两个信号量：x1 表示缓冲区空余个数， x2 表示缓冲区中已有实体的个数。初始化 x1 = n，x2 = 0, 锁 lock

生产者

```
{
    .....
    P(x1);      //生产者每生产一个，空余个数减少一个，x1-1；
    Lock_acquire(lock);          //加锁
    //缓冲区实体数加 1
    Lock_release(lock);
    V(x2)；    //生产者生产一个实体数后，x2+1
    .....
}
```

消费者

```
{
   .....
    P(x2);      //消费者每消费一个，缓冲区中实体个数减少一个，x2 -1；
    Lock_acquire(lock);          //加锁
    //缓冲区实体数减 1
    Lock_release(lock);
    V(x1)；    //消费者消费数后，缓冲区空余个数就增加一个，所以 x1+1
}
```

5. 实验感想

这次试验的第一道题坑了很久，而且花费了很多时间去想为什么会乱序，但是最后也没有把原因找出来，挺失望的，第二个排序题，思路比较简单，只要把归并排序理解好了就可以写出代码来，其实之前对归并不怎么理解，这一次，终于理解了，收获满满。