操作系统

实验二 快速排序问题
实验报告

姓名：张凯

班级：无 97

学号：2019011159

项目代码链接：*https://github.com/zhangkai0425/OS*

一、实验目的

1.通过对进程间高级通信问题的编程实现，加深理解进程间高级通信的原理

2.对 Windows 或 Linux 涉及的几种高级进程间通信机制有更进一步的了解

3.熟悉 Windows 或 Linux 中定义的与高级进程间通信有关的函数

二、实验内容

1.问题描述：

对于有 1,000,000 个乱序数据的数据文件执行快速排序

2.实验步骤：

（1）首先产生包含 1,000,000 个随机数（数据类型可选整型或者浮点型）的数据文件；

（2）每次数据分割后产生两个新的进程（或线程）处理分割后的数据，每个进程（线程）处理的数据小于 1000 以后不再分割（控制产生的进程在 20 个左右）；

（3）线程（或进程）之间的通信可以选择下述机制之一进行：

- 管道（无名管道或命名管道）
- 消息队列
- 共享内存

（4）通过适当的函数调用创建上述 IPC 对象，通过调用适当的函数调用实现数据的读出与写入；

（5）需要考虑线程（或进程）间的同步；

（6）线程（或进程）运行结束，通过适当的系统调用结束线程（或进程）。

三、实验设计

本题的目的在于采用多线程运行快速排序算法，从而提高算法效率。

快速排序算法本身是分治的思想，分治之后的任务可以独立完成。学习了操作系统多线程编程之后，一个很自然的想法就是将分治之后的任务分别交给不同的进程运行，这样，当每个进程运行结束之后，总的任务也就完成了。多线程的引入，使得分治算法能够真正发挥其优越的性能，有效地提高了算法的效率。

但是，引入多线程的同时，也不可避免地引入了互斥和同步的问题。这是因为系统的资源是有限的，不可能同时分配给每个进程，因此，合理地处理互斥的冲突和同步各进程占用资源的信息，是能够成功运行多线程程序和避免死锁、忙等待等问题的关键。

那么，具体来说如何实现多线程快速排序呢？按照实验方案要求，设计如下：

首先，产生随机数文件，比较容易。设计程序产生 1000,000 个随机数，存放到“data.dat”二进制文件中，并同时存放一个相同的 txt 格式文件便于阅读。进行排序时，为了能和原结果相比较，所以不在原文件上进行，复制一份原文件作为输出文件“out.dat”，之后的排序在**输出文件**上进行。

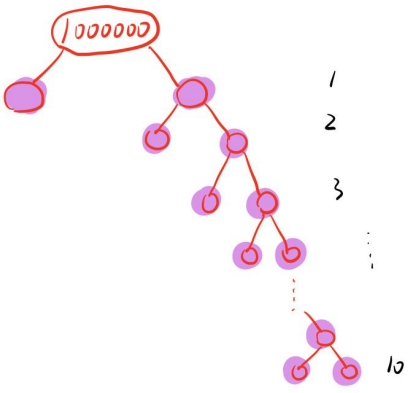
程序中利用共享内存进行多线程之间的通信，这是因为共享内存无疑是最简单的方式，只要使用文件映射函数，将文件内容映射到内存的地址当中，则只需要首地址的指针，即可读取和修改文件的内容，此时，文件本质上相当于一个“大数组”，而数组首地址的指针，可以作为 C++函数参数在所有的线程之间共享，从而直接实现了通信的作用。具体程序中，**采用 Linux 的 mmap()函数实现对文件“out.dat”的内存映射，映射返回的首地址作为函数形参传递到各个进程之中**。

之后，最根本的问题在于如何用不超过 20 个进程处理和分割 1000,000 个整数。

考虑分治算法，即相当于将 1000,000 个数用二叉树进行展开。考虑最坏的情况，展开到有<1000 的节点时，需要层数为：

$$log_2(1000000/1000) = 9.96 \approx 10$$

因此，如果每层只展开 2 个节点，则同一时刻内最多只需要 20 个进程即可，如下图所示：



实际中，这样展开不好操作，不如采取 DFS 的方式，即深度优先搜索展开，当线程数目小于等于 20 时，每分割一次，都开辟两个新的线程(但实际上，每分割只能开辟一个线程，否则 10 层后线程数达到 2^10，20 个线程不能确保到达<1000 的分割)。当线程数>20 时，新分割的信息加入队列中，队列信息也在全局共享，同样是多线程通信的方式。这样，能够确保 20 个线程分割之后，一定能够有可以工作的线程，不会造成死锁。

由此，每个线程兼顾了分割和排序的任务，即如果序列长度>1000 时，线程执行的是分割的任务；如果序列长度<1000 时，线程执行的是排序的任务。因为队列的存在，当线程执行完当前的排序任务后，可以从队列中重新取出未排序的区间，进行排序，达到了 20 个线程的复用。

排序结束后，虽然是对内存中的数据进行操作，但由于文件映射的存在，排序后的文件内容实际上已经写好了，只需要再专程 txt 文件供阅读即可。

四、代码实现

实验代码全部采用 **C++11** 编写，在 **Linux** 上运行，由 **g++**编译器进行编译。供进程通信的全局互斥、同步信号量及队列如下表所示：

类型	变量	描述
std::mutex	print_mutex	多线程打印互斥锁
	thread_open_mutex	创建新线程锁
	sorted_num_mutex	已排序个数修改锁
	request_queue	请求队列操作锁
queue<pair<int,int>>	unsorted	二叉树节点队列
Semaphore	queue_length	队列长度同步信号量
vector<std::thread>	Thread	总线程数组
int	Amount (**const int**)	排序总数 1000,000
	sorted_num	已排序数目
	thread_num	已开辟线程数

在实现多线程时，直接使用了 C++的 std::thread 类创建线程；在设置互斥锁变量时，也直接调用了 C++的 std::mutex；在实现同步信号量时，我直接重新写

了一个 Semaphore 类，从而避免了调用系统的 Semaphore。这样，可以有效避免了 Linux 系统和 Windows 系统平台不同的问题，但实际上我只有 Mac 电脑，和 Linux 比较类似，代码是在 Linux 服务器上运行的，没有在 Windows 上尝试过。(但是 Linux 上是一定能够正常运行的)

具体实现代码见附录 main_A.cpp。

五、拓展思路

实际上，上文所述是一般的思路，对原实验要求进行了稍微的改动，其特点是高效，因为 20 个线程同时进行了分割和排序的任务，大部分同学都是这样实现的。但是，这种实现也有一种缺点，即代码不够简洁和直观，因为如果要线程同时承担分割和排序的任务，另一种更加简洁的思路是直接开启一个单独的进程做分割任务，再同时开启 20 个进程，做对 1000 以内的数组做排序任务。分割进程相当于生产者，负责生产<1000 的数组分割下标，存到队列当中；20 个排序进程相当于消费者，负责消费队列中的未排序数组。从而，可以将本问题转化为经典的生产者消费者问题，这种方法代码实现极为简洁易懂，我在实现上文思路之余，也实现了此思路，当然，这种思路可能和原实验要求有所区别，但仍然是多线程快速排序很好的实现，代码见 main_B.cpp。

六、实验结果

1.main_A.cpp 结果

代码运行方法见 Github 代码说明。

结果如下：

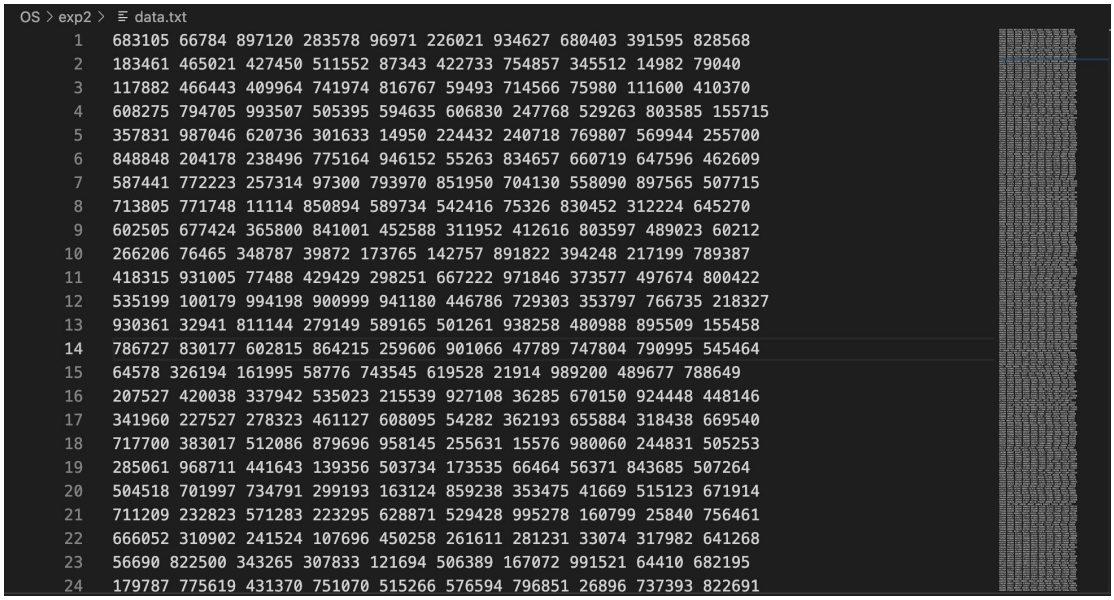
```
(base) zhangkai@server2:/hdd1/zhangkai/OS/exp2$ g++ -std=c++11 main_A.cpp -o mainn_A -lpthread
(base) zhangkai@server2:/hdd1/zhangkai/OS/exp2$ ./mainn_A
随机数生成保存完毕,开始创建文件映射...
退出线程:
一切顺利!
排序完数目:1000000
总用时: 0.392845 seconds
terminate called without an active exception
Aborted (core dumped)
(base) zhangkai@server2:/hdd1/zhangkai/OS/exp2$
```

排序数目达到 1000,000 后，结束线程，程序终止。此处我的实现有不太好的地方，即最后强行终止了线程，在所有的排序结束后，调用 **std::thread** 的析构函数终止了线程。其实这是**不完美**的，因为最佳的方式是等待线程结束，自行析

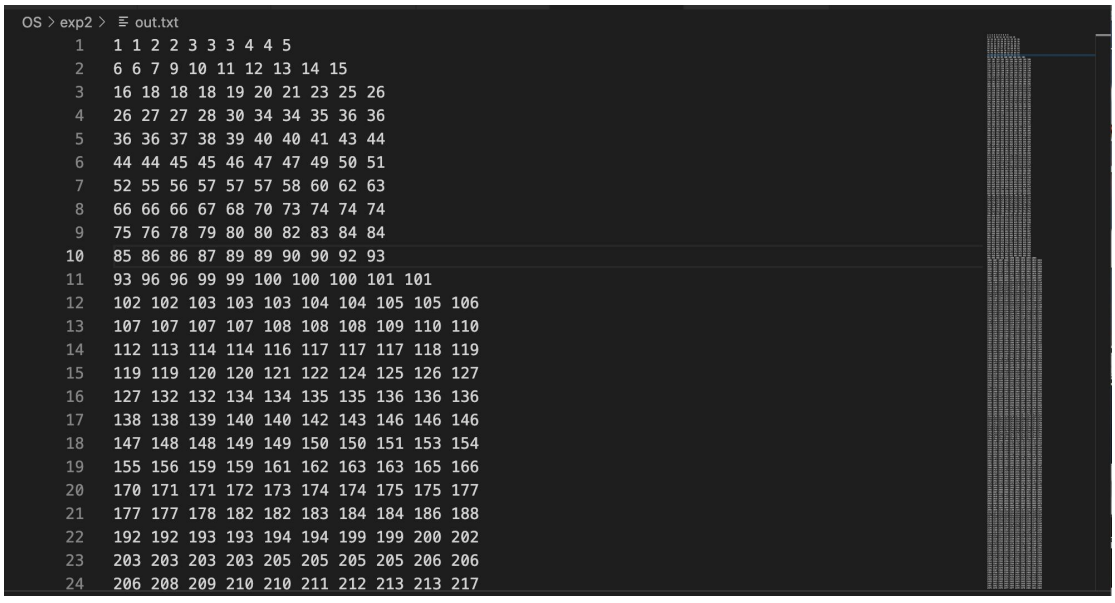
构。但是由于我的信号量是自己实现的，没有像 Windows 系统可以直接调用 Closehandle()关闭句柄，所以没法直接关闭信号量，会导致最后线程都在 P()操作当中阻塞等待了，为了方便，直接在满足条件后析构掉线程，销毁线程。

查看文件结果如下：

data.txt



out.txt



由此可见，确实实现了排序的结果，检查无误。

2.main_B.cpp 结果

结果如下：

```
(base) zhangkai@server2:/hdd1/zhangkai/OS/exp2$ g++ -std=c++11 main_B.cpp -o mainn_B -lpthread
(base) zhangkai@server2:/hdd1/zhangkai/OS/exp2$ ./mainn_B
随机数生成保存完毕,开始创建文件映射...
一切顺利！
 排序完数目:1000000
总用时: 0.41001 seconds
terminate called without an active exception
Aborted (core dumped)
(base) zhangkai@server2:/hdd1/zhangkai/OS/exp2$
```

查看文件结果如下：

data.txt

OS > exp2 > data.txt

```
610385 870464 63086 881777 970101 68388 299642 540153 105067 145243
388228 980671 924804 961211 227228 601683 711092 840213 32541 802158
911076 821744 164576 829635 192426 729020 723664 281890 683085 334262
299515 809822 204726 362602 691600 174827 947342 991242 231333 52409
652838 619561 549432 577642 97124 776661 179325 808217 133226 728218
610375 44302 66315 291303 873937 258741 20324 113953 56983 219761
448215 356499 29583 652942 235453 237535 344121 182795 745130 575454
751557 397968 711368 300989 975610 808492 594002 154936 133061 727228
399506 259788 287882 465821 551092 161819 724563 87768 792125 781546
307529 240340 654397 853464 409634 889850 91000 753756 588998 836130
845562 856907 750450 73282 157896 726060 398127 268251 397348 531188
995479 796855 790977 283362 779028 858421 961533 503591 946189 753658
801490 770070 510351 972239 139886 919985 378442 747238 190093 483792
583368 552008 340699 850170 625290 14947 576231 23417 283198 973579
70958 278678 286786 861935 78392 65815 236708 39925 85758 182897
309936 403600 469319 820287 375840 609205 740272 754282 356444 930366
238074 456164 482374 95125 822687 624016 110072 398918 163786 393271
888849 234744 188301 691988 613031 266693 757803 849739 822970 359913
548988 132906 763514 534659 469545 655706 660216 209818 926340 533012
656536 680766 505529 655262 775891 328216 795630 885963 727134 959416
795586 132335 710512 500239 824323 323543 283284 98478 689634 106255
458392 754974 755513 738258 289633 741411 393964 949850 951229 320304
482862 124117 517422 988391 295731 293313 832959 91361 695628 560093
50778 7567 692429 761290 507806 516752 601186 791091 131583 807172
```

out.txt

OS > exp2 > out.txt

```
0 3 4 4 5 6 8 8 8 10
11 12 12 12 13 13 14 16 16 17
18 18 19 19 20 21 22 22 23 23
24 26 29 29 32 34 35 36 36 36
39 39 41 41 42 43 44 46 46 47
47 48 49 51 51 51 52 52 56 57
57 57 57 58 58 58 60 60 61 61
62 62 63 64 65 71 71 73 73 75
76 78 78 78 80 83 83 85 87 87
88 88 89 91 92 92 93 94 95 95
97 97 98 99 99 100 104 104 104 104
106 106 106 107 108 109 110 110 114 116
117 118 121 122 122 124 125 127 127 128
131 135 135 136 137 139 139 141 141 143
147 147 148 148 149 149 149 150 151 153
154 155 155 156 158 158 160 161 161 164
164 164 166 167 167 167 168 169 170 172
173 173 175 175 176 177 178 178 179 180
180 182 183 184 185 185 189 189 190 190
190 190 190 190 192 192 192 194 197 198
201 201 201 202 202 204 204 205 205 205
205 207 208 210 210 211 211 212 212 214
214 214 216 217 218 220 222 222 224 225
227 227 229 230 230 230 231 231 232 234
```

由此可见，确实实现了排序的结果，检查无误。

多次运行平均时间上来看，这种我自己改进的“生产者-消费者”快速排序思路要比原始的要求的思路慢一些，这是由于仅仅有一个线程进行数据划分，而对于原始思路代码来说，有 20 个线程可以同时进行数据划分，显然，原始思路要更高效一些。但是就逻辑思路和代码的简洁性来说，main_B.cpp 是优于 main_A.cpp 的。

七、实验小结

通过本次实验，我也再一次熟悉了多线程编程的基本流程。C++ std::thread、std::mutex，自己创建的信号量的使用，在上一次的银行柜台问题中已经写过了，此次实验又一次熟悉了相关的代码编写。实验全部代码都是在 Linux 上运行的，在编程和 debug 的过程中，我也真正去熟悉了 Linux 操作系统的基本系统调用，也体会到了 Linux 的简洁。大部分同学都是用 Windows 平台进行编写，因此，Linux 相关的代码和介绍都比较少，在查阅相关 API 和 C++特性的过程中，我也进一步熟悉了 Linux 多线程编程的相关知识和技巧。

八、代码附录

main_A.cpp

```
//  main.cpp
//  操作系统大作业实验2-A
//  Created by 张凯 on 2022/5/24.
//  Copyright © 2022 张凯. All rights reserved.

#include <iostream>
#include <cstring>
#include <vector>
#include <mutex>
#include <thread>
#include <fstream>
#include <algorithm>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <queue>
#include <time.h>
#include <condition_variable>

using namespace std;

const int Amount = 1000000;
const int Buffer_size = 4000000;
const int Max_thread_num = 20;

//信号量的实现
class Semaphore
{
public:
    Semaphore(int count = 0) : count(count) {}
    // V操作，唤醒
    void V()
    {
        std::unique_lock<std::mutex> unique(mt);
        ++count;
        if (count <= 0)
            cond.notify_one();
    }
    // P操作，阻塞
    void P()
    {
        std::unique_lock<std::mutex> unique(mt);
        --count;
        if (count < 0)
            cond.wait(unique);
    }
```

```
    void getcount()
    {
        cout << this->count << endl;
    }

private:
    int count;
    mutex mt;
    condition_variable cond;
};

int data[Amount];
int thread_num = 0;
int sorted_num = 0;

//互斥锁
std::mutex print_mutex;        //多线程打印互斥信号量
std::mutex thread_open_mutex; //创建新进程锁
std::mutex sorted_num_mutex;  //已排序个数修改锁
std::mutex request_queue;     //申请进入队列锁

//二叉树节点队列
queue<pair<int, int>> unsorted;

//队列长度信号量
Semaphore queue_length(0); //队列长度信号量,如果小于等于0,当前排序进程应该阻塞

//总线程
vector<std::thread> Thread;

// quicksort algorithm
void quicksort(int *nums, int L, int R)
{
    auto ref = nums[L], i = L, j = R;
    if (L < R)
    {
        while (i != j)
        {
            while (nums[j] >= ref && j > i)
                j--;
            while (nums[i] <= ref && j > i)
                i++;
            if (i < j)
                swap(nums[i], nums[j]);
        }
        swap(nums[L], nums[i]);
        quicksort(nums, L, i - 1);
        quicksort(nums, i + 1, R);
    }
```

```
}
int getmid(int *nums, int L, int R)
{
    int ref = nums[L], i = L, j = R;
    if (L < R)
    {
        while (i != j)
        {
            while (nums[j] >= ref && j > i)
                j--;
            while (nums[i] <= ref && j > i)
                i++;
            if (i < j)
                swap(nums[i], nums[j]);
        }
        swap(nums[L], nums[i]);
    }
    auto mid = i;
    //得到分割的中点
    return mid;
}
void Sort(int *nums, pair<int, int> LR)
{
    int L = LR.first;
    int R = LR.second;
    while (sorted_num < Amount)
    {
        if (L == R)
        {
            //此进程运行结束,可以从队列中取出未排序的序列->复用当前进程,不必重新关闭和开启
            sorted_num_mutex.lock();
            sorted_num++;
            sorted_num_mutex.unlock();
            if (sorted_num == Amount)
                break;
            else
            {
                queue_length.P();
                request_queue.lock();
                pair<int, int> T = unsorted.front();
                unsorted.pop();
                print_mutex.lock();
                //                cout<<"out node:"<<"(L,R) = ("<<T.first<<","
<<T.second<<")"<<endl;
                print_mutex.unlock();
                request_queue.unlock();
                L = T.first;
                R = T.second;
            }
```

```
            continue;
        }
        else if (L < R)
        {
            if (R - L <= 1000)
            {
                quicksort(nums, L, R);
                sorted_num_mutex.lock();
                sorted_num = sorted_num + R - L + 1;
                // print_mutex.lock();
                // cout << "排序完数目:" << sorted_num << endl;
                // print_mutex.unlock();
                sorted_num_mutex.unlock();
                //此进程运行结束,可以从队列中取出未排序的序列->复用当前进程,不必重新关闭和开启
                if (sorted_num == Amount)
                    break;
                else
                {
                    //占用元素,判断队列是否有元素 P操作
                    queue_length.P();
                    request_queue.lock();
                    pair<int, int> T = unsorted.front();
                    unsorted.pop();
                    request_queue.unlock();
                    L = T.first;
                    R = T.second;
                }
            }
            // R-L>1000的情况,进行分割以及进入队列操作
            else
            {

                auto mid = getmid(nums, L, R);
                if (mid == L)
                {
                    sorted_num_mutex.lock();
                    sorted_num++;
                    sorted_num_mutex.unlock();
                    L++;
                }
                else if (mid == R)
                {
                    sorted_num_mutex.lock();
                    sorted_num++;
                    sorted_num_mutex.unlock();
                    R--;
                }
                else
                {
```

```
                    // 进行分治操作->二叉树分割
                    sorted_num_mutex.lock();
                    sorted_num++; //考虑到中间点无需排序了
                    sorted_num_mutex.unlock();
                    //开辟新进程
                    thread_open_mutex.lock();
                    if (Thread.size() == 20)
                    {
                        //进程数已经达到20
                        //入队操作锁
                        request_queue.lock();
                        unsorted.push(make_pair(mid + 1, R));
                        queue_length.V();
                        request_queue.unlock();
                        R = mid - 1;
                    }
                    else
                    {
                        //仅右子树进入线程,反之如果会造成死锁
                        pair<int, int> T = make_pair(mid + 1, R);
                        Thread.push_back(std::thread(Sort, nums, T));
                        print_mutex.lock();
                        // cout << "线程总数:" << Thread.size() << endl;
                        print_mutex.unlock();
                        R = mid - 1;
                    }
                    thread_open_mutex.unlock();
                }
            }
        }
    }
    print_mutex.lock();
    cout << "退出线程:" << endl;
    print_mutex.unlock();
    return;
}

int main()
{
    //二进制文件
    fstream data_in, data_out;
    data_in.open("data.dat", ios_base::out | ios_base::binary);
    data_out.open("out.dat", ios_base::out | ios_base::binary);
    if (!data_in.is_open())
    {
        std::cerr << "Opening file error!" << endl;
        exit(0);
    }
    std::srand(unsigned(time(nullptr)));
```

```
    for (int i = 0; i < Amount; i++)
        data[i] = rand() % 1000000;
    data_in.write((char *)data, Amount * sizeof(int));
    data_in.close();
    data_out.close();

    //保存TXT文件
    ofstream data_txt("data.txt");
    for (int i = 0; i < Amount; i++)
    {
        data_txt << data[i] << " ";
        if (i % 10 == 9)
            data_txt << endl;
    }
    data_txt.close();
    cout << "随机数生成保存完毕,开始创建文件映射..." << endl;
    //创建文件映射
    auto fd = open("data.dat", O_RDONLY);

    auto len = lseek(fd, 0, SEEK_END);
    //建立内存映射
    char *buffer = (char *)mmap(NULL, len, PROT_READ, MAP_PRIVATE, fd, 0);
    close(fd);

    //创建输出文件映射,复制原文件内容
    auto fd_out = open("out.dat", O_RDWR);

    lseek(fd_out, len - 1, SEEK_END);
    write(fd_out, "", 1);

    int *nums = (int *)mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd_out, 0);
    close(fd_out);

    //复制
    memcpy(nums, buffer, len);
    //解除映射
    munmap(buffer, len);
    //计时变量
    clock_t start, finish;
    //计时开始
    start = clock();
    //进入快速排序进程
    pair<int, int> T = make_pair(0, Amount - 1);
    thread_open_mutex.lock();
    Thread.push_back(std::thread(Sort, nums, T));
    thread_open_mutex.unlock();
    // print_mutex.lock();
    // cout << "Thread num:" << Thread.size() << endl;
    // print_mutex.unlock();
```

```
    while (sorted_num < Amount);
    //计时结束
    finish = clock();
    //解除映射
    munmap(nums, len);

    //保存TXT文件
    ifstream F("out.dat", ios::binary | ios::in);
    F.read((char *)data, Amount * sizeof(int));

    ofstream out_txt("out.txt");
    for (int i = 0; i < Amount; i++)
    {
        out_txt << data[i] << " ";
        if (i % 10 == 9)
            out_txt << endl;
    }
    out_txt.close();
    cout << "一切顺利!" << endl;
    print_mutex.lock();
    cout << "排序完数目:" << sorted_num << endl;
    print_mutex.unlock();
    cout << "总用时: "<<double(finish - start)/CLOCKS_PER_SEC<<" seconds"<<endl;
    //销毁线程
    for (int i = 0; i < Thread.size(); i++)
        Thread[i].~thread();
    return 0;
}
```

main_B.cpp

```
//  main.cpp
//  操作系统大作业实验2-B
//  Created by 张凯 on 2022/5/24.
//  Copyright © 2022 张凯. All rights reserved.

#include <iostream>
#include <cstring>
#include <vector>
#include <mutex>
#include <thread>
#include <fstream>
#include <algorithm>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <queue>
#include <condition_variable>
```

```
using namespace std;

const int Amount = 1000000;
const int Max_thread_num = 20;

//信号量的实现
class Semaphore
{
public:
    bool flag = true;
    Semaphore(int count = 0) : count(count) {}
    // V操作，唤醒
    void V()
    {
        std::unique_lock<std::mutex> unique(mt);
        ++count;
        if (count <= 0)
            cond.notify_one();
    }
    // P操作，阻塞
    void P()
    {
        std::unique_lock<std::mutex> unique(mt);
        --count;
        while (count < 0 && flag)
            cond.wait(unique);
    }
    void getcount()
    {
        cout << this->count << endl;
    }

private:
    int count;
    mutex mt;
    condition_variable cond;
};

int data[Amount];
int thread_num = 0;
int sorted_num = 0;

//互斥锁
std::mutex print_mutex;        //多线程打印互斥信号量
std::mutex thread_open_mutex; //创建新进程锁
std::mutex sorted_num_mutex;  //已排序个数修改锁
std::mutex request_queue;      //申请进入队列锁

//二叉树节点队列
```

```
queue<pair<int, int>> unsorted;

//队列长度信号量
Semaphore queue_length(0); //队列长度信号量,如果小于等于0,当前排序进程应该阻塞

//总线程
vector<std::thread> Thread;

void Producer(int *nums, int L, int R)
{
    auto ref = nums[L], i = L, j = R;
    if (L == R)
    {
        sorted_num_mutex.lock();
        sorted_num++;
        sorted_num_mutex.unlock();
    }
    else if (R - L <= 1000)
    {
        request_queue.lock();
        unsorted.push(make_pair(L, R));
        //释放队列资源 v操作
        queue_length.V();
        request_queue.unlock();
    }
    else
    {
        while (i != j)
        {
            while (nums[j] >= ref && j > i)
                j--;
            while (nums[i] <= ref && j > i)
                i++;
            if (i < j)
                swap(nums[i], nums[j]);
        }
        swap(nums[L], nums[i]);
        sorted_num_mutex.lock();
        sorted_num++;
        sorted_num_mutex.unlock();
        Producer(nums, L, i - 1);
        Producer(nums, i + 1, R);
    }
}

// quicksort algorithm
void quicksort(int *nums, int L, int R)
{
    auto ref = nums[L], i = L, j = R;
```

```
    if (L < R)
    {
        while (i != j)
        {
            while (nums[j] >= ref && j > i)
                j--;
            while (nums[i] <= ref && j > i)
                i++;
            if (i < j)
                swap(nums[i], nums[j]);
        }
        swap(nums[L], nums[i]);
        quicksort(nums, L, i - 1);
        quicksort(nums, i + 1, R);
    }
}

void Sort(int *nums, int pid)
{
    while (sorted_num < Amount)
    {
        if (sorted_num >= Amount)
            break;
        //申请队列元素 P操作
        // cout << "Pid:" << pid << " 现在的数量" << sorted_num << " 要开始申请P操作了" <<
endl;
        queue_length.P();
        if (sorted_num >= Amount)
            break;
        request_queue.lock();
        pair<int, int> T = unsorted.front();
        unsorted.pop();

        sorted_num_mutex.lock();
        sorted_num += T.second - T.first + 1;
        sorted_num_mutex.unlock();
        quicksort(nums, T.first, T.second);
        request_queue.unlock();
        // print_mutex.lock();
        // cout << " 排序完数目:" << sorted_num << endl;
        // print_mutex.unlock();
    }
}

int main()
{
    //二进制文件
    fstream data_in, data_out;
    data_in.open("data.dat", ios_base::out | ios_base::binary);
```

```
    data_out.open("out.dat", ios_base::out | ios_base::binary);
    if (!data_in.is_open())
    {
        std::cerr << "Opening file error!" << endl;
        exit(0);
    }
    std::srand(unsigned(time(nullptr)));
    for (int i = 0; i < Amount; i++)
        data[i] = rand() % 1000000;
    data_in.write((char *)data, Amount * sizeof(int));
    data_in.close();
    data_out.close();

    //保存TXT文件
    ofstream data_txt("data.txt");
    for (int i = 0; i < Amount; i++)
    {
        data_txt << data[i] << " ";
        if (i % 10 == 9)
            data_txt << endl;
    }
    data_txt.close();
    cout << "随机数生成保存完毕,开始创建文件映射..." << endl;
    //创建文件映射
    auto fd = open("data.dat", O_RDONLY);
    auto len = lseek(fd, 0, SEEK_END);
    //建立内存映射
    char *buffer = (char *)mmap(NULL, len, PROT_READ, MAP_PRIVATE, fd, 0);
    close(fd);

    //创建输出文件映射,复制原文件内容
    auto fd_out = open("out.dat", O_RDWR);

    lseek(fd_out, len - 1, SEEK_END);
    write(fd_out, "", 1);

    int *nums = (int *)mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd_out, 0);
    close(fd_out);

    //复制
    memcpy(nums, buffer, len);
    //解除映射
    munmap(buffer, len);
    //计时变量
    clock_t start, finish;
    //计时开始
    start = clock();
    //创建Producer 线程
    thread_open_mutex.lock();
```

```
    Thread.push_back(std::thread(Producer, nums, 0, Amount - 1));
    thread_open_mutex.unlock();
    Thread[0].join();
    //创建20个快速排序进程
    for (int i = 0; i < Max_thread_num; i++)
    {
        thread_open_mutex.lock();
        Thread.push_back(std::thread(Sort, nums, i + 1));
        // print_mutex.lock();
        // cout << "Thread num:" << Thread.size() << endl;
        // print_mutex.unlock();
        thread_open_mutex.unlock();
    }

    while (sorted_num < Amount);
    //计时结束
    finish = clock();
    //保存TXT文件
    ifstream F("out.dat", ios::binary | ios::in);
    F.read((char *)data, Amount * sizeof(int));

    ofstream out_txt("out.txt");
    for (int i = 0; i < Amount; i++)
    {
        out_txt << data[i] << " ";
        if (i % 10 == 9)
            out_txt << endl;
    }
    out_txt.close();
    munmap(nums, len);
    cout << "一切顺利!" << endl;
    print_mutex.lock();
    cout << " 排序完数目:" << sorted_num << endl;
    print_mutex.unlock();
    cout << "总用时: " << double(finish - start) / CLOCKS_PER_SEC << " seconds" << endl;
    queue_length.flag = false;
    //销毁线程
    for (int i = 1; i < Thread.size(); i++)
        Thread[i].~thread();
    //解除映射
    return 0;
}
```