# 《操作系统》实验 2：高级进程间通信问题——快速排序

班级:无 22　姓名:**王炜致**　学号:2022010542

## 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　设计思路

| 操作系统平台 | 编程语言 | 头文件 |
|---|---|---|
| Windows 11 | C++ | <iostream>,<vector>,<queue>,<fstream>,<ctime> 及 mingw.thread.h,mingw.mutex.h |

表 1: 实验背景

首先生成 1,000,000 个随机数构成数组，不赘述.

考虑引用实验 1 中的思想. 预定义一个线程池，其中有一定量的空线程可用；预定义一个消息队列. 将快速排序函数分支提出的新线程需求视为任务**“生产”**，将线程视为任务**“消费者”**. 生产者在快速排序函数运行的过程中将“有新任务”的消息挂载到消息队列，而消费者监视消息队列，并从队头接收消息订单并进行处理. 对队列的修改需要利用 mutex 互斥量互斥地进行，而修改队列（长度）即是实现线程同步.

具体而言：首先，主线程运行快速排序函数. 根据快速排序算法的分治思想，它将待排序数组拆分成两段并递归地排序这两段子数组. 正常的快速排序函数使用串行操作，如先递归排序“左”段，后递归排序“右”段；事实上，(在数组长度大于题给下阈值的情况下）我们完全可以引入多线程处理，并行地处理两段子数组（进一步递归也一样），起到加速效果. 由于每次递归都会多产生一个分支（二叉），我们可以让原线程继续操作“左”段；对于多出来的“右”段，并在消息队列中通知线程池派出一个空闲线程来接管“右”段任务.

为了方便地实现互斥量 **(P(mutex),V(mutex) 函数)** 及线程功能，考虑引用头文件 mingw.thread.h,mingw.mutex.h，注：引用 <thread><mutex> 时发现无法正常读取，参考教程 https://blog.csdn.net/Flag_ing/article/details/126967720 下载并改用.

## 3　关键代码解析

### 3.1　生成随机数，并存放在 data.txt 中

```
#define N 1000000
#define MIN 0
#define MAX 100000000
int main() {
    ofstream out("data.txt");
    if (out.is_open())
        for (int i=0; i<N; ++i)
            out << ((rand()<<10 + rand()) % (MAX - MIN + 1)) + MIN << endl;
    out.close();
}
```

### 3.2 定义超参数、信息结构体

定义一则消息包含待排序数组 arr 的左端下标和右端下标.

```
#define IN_FILENAME "data.txt" // 随机数文件名
#define OT_FILENAME "processed.txt" // 排序后文件名
#define THRES 1000 // 下限
#define THPOOL 20  // 线程池

struct message { // 消息，描述了任务信息：
    int left;          // 1) 待排序数组左头
    int right;         // 2) 待排序数组右头
};
typedef struct message message;
```

### 3.3 定义消息队列类

基于消息结构体定义消息队列，内置互斥锁、队列以及 send()（上载消息）方法、receive()（下载消息）方法、isEmpty()（判断队列是否为空，用于线程退出）. 函数方法内置了维护队列时的互斥访问.

```
class messageQueue {
private:
    mutex mtx_queue; // 互斥访问
    queue<message> messages; // 队列长度messages.size()为同步量，运行quickSort分支时++，线程池接单--
public:
    void send(int l, int r) {
        mtx_queue.lock();
        messages.push({l, r});
        // cout << "需要新线程处理" << l << "," << r << "段" << endl;
        mtx_queue.unlock();
    }
    message receive() {
        mtx_queue.lock();
        message msg = messages.front();
        messages.pop();
        // cout << "新线程开始处理" << msg.left << "," << msg.right << "段" << endl;
        mtx_queue.unlock();
        return msg;
    }
    bool isEmpty() {
        return messages.empty();
    }
};
messageQueue m_queue;
vector<int> arr; // 待排序数组
```

### 3.4 融合多线程化的快速排序函数

首先判断是否结束递归；其次判断数组长度是否满足阈值要求，如果过短则不引入新线程，直接用传统串行方法完成；否则对于二叉分支多出的那个分支配置新线程，另一个分支由当前分支继续执行. 注意在处理的时候，我们提前判断分支是否合法 (left<right) 而非直接分配线程、运行函数再退出，这样可以起到**剪枝**效果，节约线程调用及其时间开销.

```
void quickSort_withThread(vector<int>& arr, int left, int right){ // 融合线程调用操作的快排；【生产者】
    if (left < right) { // 递归出口
        int mid = partition(arr, left, right);
        if (right - left < THRES) { // 数组长度低于阈值，无需交付新线程，自行串行解决
            quickSort_withThread(arr, left, mid - 1);
            quickSort_withThread(arr, mid + 1, right);
        } else { // 数组长度高于阈值，留待线程处理，加入队列，相当于顾客到来
            if (left < mid-1 && mid+1 < right) { // 存在双分支，于消息队列保存其一供新线程用，原线程继续执行另一半，节约线程
                m_queue.send(left, mid-1);
                quickSort_withThread(arr, mid+1, right);
            }
            else if (left < mid-1) { // 事实上只有单分支，原线程继续执行
                quickSort_withThread(arr, left, mid-1);
            }
            else if (mid+1 < right) { // 事实上只有单分支，原线程继续执行
                quickSort_withThread(arr, mid+1, right);
            }
        }
    }
}
```

### 3.5 线程行为函数

由于分支任务大量产生，消息队列中总是存在消息，除非所有任务执行完毕，考虑以**消息队列空**为判断完成排序并退出机制.

```
void Thread() { //  【消费者】
    while (true) {
        if (m_queue.isEmpty()) break; // 队列已空，退出机制
        message msg = m_queue.receive();
        quickSort_withThread(arr, msg.left, msg.right);
    }
}
```

### 3.6 主函数

```
int main() {
    // 读文件
    ifstream inflow(IN_FILENAME);
    int temp;
    while (inflow >> temp) arr.push_back(temp);
    inflow.close();

    // 建立线程、运行并计时
    vector<thread> threads(THPOOL);
    clock_t T = clock();
    quickSort_withThread(arr, 0, arr.size()); // 初始化任务
    for (int i=0; i<threads.size(); ++i) threads[i] = thread(Thread);
    for (int i=0; i<threads.size(); ++i) threads[i].join(); // 等待全部线程结束
```

```
    double dt = double(clock() - T);
    cout << "排序耗时" << dt << "ms." << endl;

    // 写文件（排序后）
    ofstream outflow(OT_FILENAME);
    for (int d : arr) outflow << d << endl;
    outflow.close();
}
```

# 4 结果

## 4.1 正确完成排序

根据输出 processed.txt 显示，无序随机数的（升序）排序正常完成，可参见附件；进一步编程验证，输出确实为“排序正确！”

```
#define FILENAME "processed.txt"
int main() {
    ifstream inflow(FILENAME);
    vector<int> arr;
    int temp;
    while (inflow >> temp) arr.push_back(temp);
    for (int i=1; i<arr.size(); ++i) if (arr[i]<arr[i-1]) {
        cout << "排序有误！" << endl;
        return 0;
    }
    cout << "排序正确！" << endl;
}
```

## 4.2 时间性能优化

测量多组算法执行时间，与普通的快速排序算法比较如下，发现多线程方法存在明显的性能优势：

| 数据 | 多线程（并行）快排 (ms) | 普通（串行）快排 (ms) |
|---|---|---|
| 1 | 1236 | 3249 |
| 2 | 1341 | 3034 |
| 3 | 1307 | 3074 |
| 均值 | 1295 | 3119 |

表 2: 性能比较

# 5 思考题

## 5.1 你采用了你选择的机制而不是另外的两种机制解决该问题，请解释你做出这种选择的理由.

我采用了**消息队列机制**，因为其思想与我实现实验 1 的思想非常相似，依靠经验与直觉可以直接使用. 将银行柜员替换为线程池中的可用线程，将银行顾客替换为 quickSort_withThread() 抛出的分支线程请求，维护一个“顾客”等待队列作为消息队列即可.

### 5.2 你认为另外的两种机制是否同样可以解决该问题？如果可以请给出你的思路；如果不能，请解释理由.

**管道**可以解决该问题，大致思路为：对于快速排序算法产生的 2 分支，父进程创建 2 个管道连接 2 个子进程，并写管道，向子进程发送数组起止索引信息，子进程读取管道并用以执行分支任务，完成排序后将消息通过管道回传父进程. 递归地进行如上操作即可.

事实上，**共享内存**可以更直接方便地处理快速排序问题. 因为快速排序的子问题互相独立，且线程池对于任务（队列中的消息）并没有**选择性**，只要存在任务且空闲即可执行该任务，所以可以取消线程间的同步设计，即取消消息队列和消息收发操作，在需要调用新线程时直接定义 1 个新线程以控制分支函数. 但可能需要注意可用线程数相关的控制操作（如果存在线程**总数约束，如 20**），如等待线程空闲或放弃并行方法.