

# 高级进程间通信问题 - 快速排序问题

---

- 无42 林子恒 2014011054
- Brian Lin,Tzu-Heng's Work
  - Mailto: [lzhbrian@gmail.com](mailto:lzhbrian@gmail.com)
  - Github: [lzhbrian](#)
  - Linkedin: [lzhbrian](#)

## [高级进程间通信问题 - 快速排序问题](#)

### [一、问题描述](#)

#### [1. 实验步骤](#)

#### [2. 实验平台和编程语言](#)

### [二、实现](#)

#### [1. 思路：](#)

#### [2. 产生一个有\\$1,000,000\\$个数的文件：](#)

#### [3. 实现快速排序：](#)

### [三、实验结果](#)

### [四、思考题](#)

### [五、感想](#)

## 一、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。

### 1. 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）。

### 2. 实验平台和编程语言

- 自由选择Windows或Linux。
- 编程语言不限。

## 二、实现

### 1. 思路：

- 因为我的电脑是 *\*nix* 系列的，所以我采用 **posix** 的函数来实现进程、线程的操作
- 本实验中，我采用 **多线程**、**共享内存** 的方式来实现快速排序，主要思路如下：
  - `Quick_sort()` 函数：按照快速排序的思想，选定最后序列里最后一个数作为基准，将自己分成两半（大于基准，小于基准），再创建两个新线程递归地调用自己，分别处理这两半序列。
  - 如果新线程所需要处理的数据数量小于1000，则调用 `real_quick_sort()` 函数实现这些数的快速排序
  - 这里采用多线程的方式，所以我全局的申明了一个 `Datatype data[number_length]` 数组，即为这所有线程的共享内存；
    - 同时，他们访问 `data[]` 的时候也并不会发生冲突，因为每个线程处理的都是这个数组里面不同范围的数据。

### 2. 产生一个有1,000,000个数的文件：

- 我们首先编写 python 程序 `generate_1m_numbers.py` 来生成 1,000,000个随机小数，并将其写到一个文件 `./Numbers.txt` 里面，代码如下：

```
1 import random
2 fp = open('./Numbers.txt','w')
3 length = 1000000
4 min_x = 0
5 max_x = 60000
6 count = 0
7 for x in xrange(0,length):
8     print >> fp, (random.uniform(min_x, max_x))
9     count += 1
10 print "Suc. print ", count, " numbers"
```

### 3. 实现快速排序：

1. `Quick_sort()` 函数：

```
1 // quick sort unit
2 void *Quick_sort(void* para_s) {
3
4     // get received arg
5     struct para *recv_para;
6     recv_para = (struct para *)para_s;
7     int start_index = (*recv_para).start_index;
8     int end_index = (*recv_para).end_index;
```

```

9
10 // length of data this thread can control
11 int data_length = end_index - start_index + 1;
12 if (data_length <= 1000) {
13     real_quick_sort(data, start_index, end_index);
14     return 0;
15 }
16
17 int *pos = new int(1);
18 *pos = partition(data, start_index, end_index);
19
20 pthread_t front_thread;
21 pthread_t back_thread;
22
23 struct para para_front;
24 struct para para_back;
25
26 para_front.start_index = start_index;
27 para_front.end_index = *pos-1;
28
29 para_back.start_index = *pos+1;
30 para_back.end_index = end_index;
31
32 // Create front,back threads
33 if(pthread_create(&front_thread, NULL, Quick_sort, &(para_front))) {
34     cout << "\n ERROR creating front thread!" << endl;
35     exit(1);
36 }
37 if(pthread_create(&back_thread, NULL, Quick_sort, &(para_back))) {
38     cout << "\n ERROR creating back thread!" << endl;
39     exit(1);
40 }
41
42 // Join front,back threads
43 if(pthread_join(front_thread, NULL)) {
44     cout << "\n ERROR joining front thread!" << endl;
45     exit(1);
46 }
47 if(pthread_join(back_thread, NULL)) {
48     cout << "\n ERROR joining back thread!" << endl;
49     exit(1);
50 }
51
52 return 0;
53 }

```

2. 个数<1000时的快速排序 `real_quick_sort` :

```

1 // 小于 1000 时调用的 quicksort
2 void real_quick_sort(Datatype* a, int left, int right) {
3     if (right <= left) return;
4     int pos = partition(a, left, right);
5     real_quick_sort(a, left, pos-1);
6     real_quick_sort(a, pos+1, right);
7 }
8 // exchange
9 void exch(Datatype* a, Datatype* b) {
10     Datatype temp = *a;
11     *a = *b;
12     *b = temp;
13 }
14 int partition(Datatype* a, int low, int up) {
15     Datatype pivot = a[up];
16     int i = low-1;
17     for (int j = low; j < up; j++)
18     {
19         if(a[j] <= pivot)
20         {
21             i++;
22             exch(&a[i], &a[j]);
23         }
24     }
25     exch(&a[i+1], &a[up]);
26     return i+1;
27 }

```

### 3. 主函数:

```

1 /***** Main func *****/
2 int main(int argc, char * argv[])
3 {
4
5     // Start time
6     open_time = getSystemTime();
7
8     // Read 1m numbers
9     ifstream file("./Numbers.txt");
10    while (!file.eof()) {
11        file >> data[n];
12        n = n + 1;
13    }
14
15    // indicator of success read files
16    cout << getSystemTime() - open_time << " ms used to read " <<
    numbers_length << " numbers..." << endl;

```

```

17
18     // test for read numbers
19     // cout << data[100] << ", " << data[200] << ", " <<
data[numbers_length-1] << ", " << data[numbers_length] << endl;
20
21     pthread_t main_thread;
22
23     struct para pata_main;
24     pata_main.start_index = 0;
25     pata_main.end_index = numbers_length-1;
26
27     // Create main thread
28     if(pthread_create( &main_thread, NULL, Quick_sort, &(pata_main) )) {
29         cout << "\n ERROR creating first main thread!" << endl;
30         exit(1);
31     }
32
33     // Join main thread
34     if(pthread_join(main_thread, NULL)) {
35         cout << "\n ERROR joining first main thread!" << endl;
36         exit(1);
37     }
38
39
40     // Check result
41     cout << "First 10 numbers: \n";
42     for (int i = 0; i < 10; ++i) {
43         cout << data[i] << endl;
44     }
45     cout << endl;
46     cout << "Last 10 numbers: \n";
47     for (int i = numbers_length-10; i < numbers_length; ++i) {
48         cout << data[i] << endl;
49     }
50
51     // indicator of runtime
52     cout << getSystemTime() - open_time << " ms used to finish the whole
program! " << endl;
53
54     // write sorted result to file
55     ofstream fout("./sorted_Numbers.txt");
56     for (int i = 0; i < numbers_length; ++i) {
57         fout << data[i] << endl;
58     }
59     return 0;
60 }

```

### 三、实验结果

---

1. 通过 `generate_1m_numbers.py` 生成的文件为： `Numbers.txt`，我们设置的数范围为：0～60,000
2. 通过 `quick_sort_multithread.cpp` 的操作之后生成的排序后的文件为： `sorted_Numbers.txt`，其前十行、后十行代码如下：

```
0.0153327
0.13771
0.241184
0.24939
0.292859
0.359523
0.375892
0.419432
0.48103
0.507452
```

```
59999.3
59999.4
59999.5
59999.5
59999.6
59999.6
59999.7
59999.8
60000
60000
```

可以看到，我们的程序成功地对原数据排了序。

## 四、思考题

---

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。
  - 我们采用的是共享内存的方式。
  - 因为我们采用了多线程的方式，在同一个进程里，所有的资源都是共享的，因此我直接申明一个全局变量，那他就直接会变成所有线程的共享内存了。
  - 如果采用消息队列、管道的方式，就涉及到进程（or 线程）间传递数据的问题；不论是从**代码实现**还是**所需空间资源**来说都是比较不好的选择。
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。
  - 另外两种方式当然也可以解决这个问题
    - 如果使用管道，每个进程可以将partition过后的两半分别通过管道传递给两个子进程
    - 如果使用消息队列，也是与管道同样的思路，将两半数据分别传递给两个子进程

## 五、感想

---

- 有了第一题创建线程的经验基础，再做这道题的时候就比较熟练了
- 在这次实验中，我对线程的创建、使用又有了更加深刻的理解