

操作系统大作业实验 2 报告

高艺轩 2022010639

1 实验目的

1. 通过对进程间高级通信问题的片成实现，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。

2 实验题目

本次实验选择的题目是“快速排序问题”。

3 问题描述

对有1,000,000 个乱序数据的数据文件执行快速排序。

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函数 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）。

4 思路与程序结构

快速排序作为一种经典的分治算法，天然地适合多线程并行执行。在快速排序的各个分支调用中，调用的列表部分完全不重合，线程之间在逻辑上完全不可能出现数据竞争的情况，因此在实现算法时自然地想到可以直接使用共享内存实现，它们甚至不需要 Mutex 就可以保证进程间同步。

具体细节上，采用多线程来实现并行，因为相比起使用多进程，多线程中共享内存只需要使用全局变量即可，极大地减少了编程的复杂度。为了防止分治导致的产生的线程数过多，当递归调用的范围小于1000 时，改为调用单线程版本的qsort，不再产生更多子线程。

实验中，直接用全局变量int numbers[]实现待排序数组的存储。快速排序算法主要分为两个步骤：从待排区间中选取一个作为基准，将区间中所有小于基准的都放在基准左边，所有大于基准的都放在基准右边；在基准左与基准右侧都递归调用快速排序。具体的实现如下：

```
14 // LST_QSORT_IMPL
15 struct ThreadArgs {
16     int low;
17     int high;
18 };
19
20 void swap(int *a, int* b) {
21     int temp = *b;
22     *b = *a;
23     *a = temp;
24 }
25
26 // Returns the index of pivot in arr[].
27 int partition(int low, int high) {
28     int pivot = numbers[high]; // Pivot element
29     int i = low - 1;
30
31     for (int j = low; j < high; j++) {
32         if (numbers[j] < pivot) {
33             i++;
34             swap(&numbers[i], &numbers[j]);
35         }
36     }
37
38     swap(&numbers[i + 1], &numbers[high]);
39     return i + 1;
40 }
41
```

```

42 // Used for list less than 1000 in length.
43 void qsort_nothread(int low, int high) {
44     if (low ≥ high) {
45         return;
46     }
47
48     int pivot_idx = partition(low, high);
49     qsort_nothread(low, pivot_idx - 1);
50     qsort_nothread(pivot_idx + 1, high);
51 }
52
53 void *qsort_thread(void *arg) {
54     struct ThreadArgs* args = (struct ThreadArgs*) arg;
55     // printf("Low: %d, high: %d\n", args->low, args->high);
56     if (args->high - args->low ≤ SINGLE_THREAD_SRTN) {
57         qsort_nothread(args->low, args->high);
58         return NULL;
59     }
60
61     int pivot_idx = partition(args->low, args->high);
62     struct ThreadArgs left_thd_arg = {
63         .low = args->low,
64         .high = pivot_idx - 1
65     };
66     struct ThreadArgs right_thd_arg = {
67         .low = pivot_idx + 1,
68         .high = args->high
69     };
70     pthread_t left_handle, right_handle;
71     pthread_create(&left_handle, NULL, qsort_thread, (void *)&left_thd_arg);
72     pthread_create(&right_handle, NULL, qsort_thread, (void *)&right_thd_arg);
73     pthread_join(left_handle, NULL);
74     pthread_join(right_handle, NULL);
75 }
76 // LST_QSORT_IMPL

```

最后，在main中，只需要从文件中读入数据、存入numbers[]，调用排序，并将排序好的数据重新写入文件中即可：

```

78 // LST_MAIN_FUNC
79 void do_qsort() {
80     struct ThreadArgs arg = {
81         .low = 0,
82         .high = NUM_LENGTH - 1

```

```

83     };
84     qsort_thread(&arg);
85 }
86
87
88 int main() {
89     printf("Start reading file ... \n");
90     numbers = malloc(sizeof(int) * NUM_LENGTH);
91     if (!numbers) {
92         printf("Failed to allocate memory");
93         return 1;
94     }
95
96     FILE *file = fopen("numbers.txt", "r");
97     if (!file) {
98         printf("Failed to open numbers file");
99         return 1;
100    }
101
102    int count = 0;
103    char line[MAX_LINE_LENGTH];
104    while (count < NUM_LENGTH && fgets(line, sizeof(line), file)) {
105        // printf("Count: %d\n", count);
106        sscanf(line, "%d", &numbers[count]);
107        count++;
108    }
109
110    fclose(file);
111    printf("Start sorting ... \n");
112    do_qsort();
113    printf("Done sorting, writing to file ... \n");
114
115    file = fopen("sorted.txt", "w");
116    if (!file) {
117        printf("Filed to open sorted file");
118        return 1;
119    }
120
121    count = 0;
122    while (count < NUM_LENGTH) {
123        fprintf(file, "%d\n", numbers[count]);
124        count++;
125    }

```

```

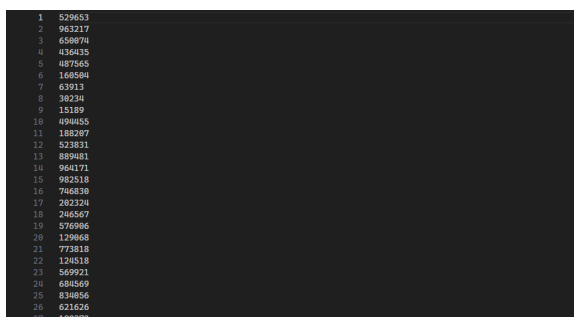
126 |     fclose(file);
127 |     return 0;
128 | }
129 | // LST_MAIN_FUNC

```

5 程序运行情况

利用gen_num.sh在numbers.txt生成1,000,000 个随机数（将 1-1,000,000）随机排列；之后调用make run运行程序，运行的结果存储在sorted.txt中。

运行效果：



```

1 529653
2 963217
3 658974
4 436435
5 487565
6 168594
7 63913
8 38234
9 15189
10 490455
11 188287
12 523831
13 889481
14 964171
15 982318
16 746838
17 282324
18 246587
19 576986
20 129668
21 773818
22 124518
23 569921
24 684569
25 834856
26 621626
27 199372

```

(a) 生成的随机数



```

1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
17 17
18 18
19 19
20 20
21 21
22 22
23 23
24 24
25 25
26 26
27 27
28 28
29 29
30 30
31 31

```

(b) 排序后的随机数

图 1: 程序测试结果

6 思考题解答

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。

答. 主要是考虑到本次要解决的的问题中，父线程要交给子进程大量的数据，而子线程排序完毕后还需要返回同样数量的数据，使用消息队列较为低效，而管道通常是单向传输的，因此不利于在本次问题中的双向信息传输。同时，由于快排算法本身逻辑上就不会造成数据竞争，因此也不需要信号量或者其它方法来对内存施加更多保护。另外，使用其它消息传递机制通常会导致不必要的内存拷贝，效率比共享内存更低。

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。

答. 我认为也可以解决。在本问题中, 需要解决的核心问题是要使用单向传输的消息队列和管道机制解决双向通信。基本的思路应当是, 在创建子线程前, 父线程先创建两个消息队列/管道, 其中一个是父进程写入的, 一个是父进程读取的, 之后再将它们的一端交给子线程, 再进行通信。

7 实验总结

本次实验本身并不复杂, 在重新确认了快速排序的算法内容之后我很快就调试出了正确的程序。这是我第一次用 C 语言编写多线程函数, 对多线程也有了更多的认识。