高级进程间通信实验报告

——快速排序问题

无64 2016011088 徐泽来

2018 年 11 月 15 日

1 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Linux中定义的与高级进程间通信有关的函数。

2 设计思路

本次实验编写了rand.c和quick_sort.c两个程序，其中rand.c生成包含1,000,000个随机整数的序列，并将其写入test.in文件；quick_sort.c读取test.in文件中的随机整数序列，完成多线程快速排序，并将排序后序列写入test.out文件。rand.c的实现过程相对容易，下主要说明quick_sort.c的设计思路。

2.1 主函数

主函数的设计思路如下：

1. 创建共享内存，为其分配内存空间，并将待排序数组指针映射至该共享内存；
2. 从文件中读取随机整数序列，创建主线程对该序列执行多线程快速排序，并在结束后将排序后序列写入文件；
3. 解除待排序数组指针与共享内存的映射，并删除共享内存。

主函数的程序结构如下所示，具体实现可参考quick_sort.c中的main()函数。

```
int main()
{
    int fd = shm_create();                          /*创建共享内存*/
    ftruncate(fd,sizeof(int)*1000000)               /*分配内存空间*/
    int *arr = mmap(fd);                            /*映射至共享内存*/

    read_file(&arr);                                /*读取文件*/
    create(&main_thread,quick_sort_multhread);      /*执行多线程快排*/
    join(main_thread);
    write_file(&arr);                               /*写入文件*/

    munmap(arr);                                    /*解映射共享内存*/
    shm_unlink();                                   /*删除共享内存*/
}
```

2.2 单线程快排

若待排序序列arr的左端元素序号为left，右端元素序号为right，实现序列划分操作的函数为partition()，则单线程快排的设计思路如下：

1. 检查待排序序列的元素个数，若大于1则继续执行，否则直接返回；
2. 调用partition()函数实现序列划分，并分别对左右序列递归调用单线程快排。

单线程快排的程序结构如下所示，具体实现可参考quick_sort.c中的quick_sort()函数。

```
void quick_sort(int *arr,int left,int right)
{
    if(left < right)                            /*元素个数大于1*/
    {
        index = partition(arr,left,right);      /*序列划分*/
        quick_sort(arr,left,index-1);           /*对左序列执行快排*/
        quick_sort(arr,index+1,right);          /*对右序列执行快排*/
    }
    return;
}
```

2.3 多线程快排

多线程快排的设计思路如下：

1. 打开共享内存，并将待排序数组指针映射至该共享内存；

2. 检查待排序序列的元素个数，若大于1000则继续执行，否则直接调用单线程快排；

3. 调用partition()函数实现序列划分，并分别对左右序列递归调用多线程快排；

4. 解除待排序数组指针与共享内存的映射。

多线程快排的程序结构如下所示，具体实现可参考quick_sort.c中的quick_sort_multhread()函数。

```
void *quick_sort_multhread(void *arg)
{
    int left = ((int *)arg)[0];                    /*读取左端元素序号*/
    int right = ((int *)arg)[1];                   /*读取右端元素序号*/

    int fd = shm_open();                           /*打开共享内存*/
    int *arr = mmap(fd);                           /*映射至共享内存*/

    if(right-left < 1000)                          /*元素个数小于1000*/
    {
        quick_sort(arr,left,right);                /*执行单线程快排*/
    }
    else
    {
        index = partition(arr,left,right);         /*序列划分*/

        /*对左右序列分别执行多线程快排*/
        create(&left_thread,quick_sort_multhread);
        create(&right_thread,quick_sort_multhread);
        join(left_thread);
        join(right_thread);
    }

    munmap(arr);                                   /*解映射共享内存*/
}
```

3 运行结果

使用说明

1. 实验平台为Linux系统；

2. 在命令行输入 ./rand 生成随机数文件test.in；

3. 在命令行输入 ./quick_sort 执行快速排序，并将排序结果写入文件test.out；

4. 由于使用共享内存实现进程间通信，因此在程序运行过程中可能出现打开文件数目超过系统限制的情况。解决方案为在在命令行输入 ulimit -n 4096 将最大打开文件数目临时修改为4096。

测试文件

为初步验证实验结果，在测试中先进行了10元素序列的多线程快速排序测试，具体的测试文件如下

test.in

```
43  5  47  71  77  10  22  71  21  29
```

运行结果

排序后输出文件如下

test.out

```
5  10  21  22  29  43  47  71  71  77
```

在10元素序列的小规模测试后，我又在quick_sort中加入了检查排序后数组是否满足升序要求的测试代码，在1,000,000元素序列的测试中检查结果为排序成功。因测试文件过大，在报告中不予显示。

4 思考题解答

思考题1 你采用了共享内存的机制而不是管道或消息队列的机制解决该问题，请解释你做出这种选择的理由。

解答： 选择共享内存机制的原因是其方便性和高效性。方便性体现在只需要在不同的进程（或线程，下同）中将相关变量映射至同一共享内存的地址空间，即可实现不同进程间的通信，且实现共享内存的函数接口也较为简单；高效性体现在进程间通信的数据不需要进行传输，而只需要直接访问内存，因此共享内存是速度最快的进程间通信方式。

而另一方面，共享内存也存在着数据同步困难的缺点，但这一缺点在本次实现多线程快速排序的实验中并没有体现，这是因为不同线程读写的是共享内存中序列的不同片段，故不会出现数据读写的竞争条件，因此也无需考虑线程间的互斥和同步，从而使得实现过程大大简化。

思考题2 你认为另外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；如果不行请解释理由。

解答： 采用管道或消息队列的机制均可以解决该问题，通过这两种机制实现的思路如下

1. 管道：程序的整体结构与采用共享内存类似，区别在于父子进程之间的双向通信需要建立两个管道：一个管道以父进程为写端，子进程为读端，实现待排序数组的传递；另一个管道以子进程为写端，父进程为读端，实现排序后数组的传递。虽然通过管道也可以实现进程间通信，但该机制存在只能承载无格式字节流且缓冲区大小受限的缺点。

2. 消息队列：消息队列与管道类似，区别在于父子进程之间的双向通信只需要建立一个消息队列：子进程读取父进程向消息队列中写入的待排序数组，完成快速排序后再向消息队列中写入排序后数组供父进程读取。虽然消息队列克服了管道只能承载无格式字节流的缺点，但其仍然存在缓冲区大小受限，以及信息传递效率较低的不足。

5 实验总结

5.1 调试问题

问题1 当待排序序列元素个数较多时，出现打开共享内存失败的情况。

解决方案 通过调试发现打开共享内存失败时返回的错误代码errno为24，即错误为 too many open files。分析可知，这是因为在每个线程中都打开了共享内存，相当于每个线程都打开了一个文件，因线程数目过多导致打开文件数目超过系统限制，从而出现共享内存打开失败的情况。

该问题有两种可能的解决方案。一种为减少并发执行的线程数目，如在创建左线程后不马上创建右线程，而是等待左线程执行结束后再创建右线程。但是，这样的解决方式降低了程序的并发执行程度，实际执行过程与单线程快排没有任何区别，不符合我们通过多线程提高程序效率的目标。

另一种为临时修改系统的最大打开文件数目。分析可知，停止多线程快排调用时的序列规模大约为1,000个元素，因此这样的线程大约有1,000,000/1,000=1,000个，注意到线程的递归调用，所有线程大约有2,000个，再考虑划分操作时的非理想因素，实际线程数目大约为3,000个，因此最大文件打开数目约为3,000个。通过在命令行输入 ulimit -n 4096 命令，可以临时将系统的最大打开文件数目修改为4096个，从而保证程序可以正常执行。

问题2 在单次快排中将序列的尾元素作为划分元素，可能出现序列划分不平均的情况，从而导致快排的性能下降。

解决方案 对划分元素的选择进行简单的改进：考虑待排序序列的首元素、尾元素、和中间元素，选择三者的中间值作为划分元素，并与尾元素交换。这样至少可以保证划分元素不是待排序序列中的最大值或最小值，且较均匀划分待排序序列的概率也得到提升，使快排的对数时间复杂度性能得到保证。

5.2 收获体会

本次实验让我进一步熟悉和掌握了进程间通信的共享内存机制，同时锻炼了我的分析调试能力，较大的收获是通过错误代码errno对程序出错的原因进行分析和调试。

在进程间通信方面，我通过本次实验熟悉了Linux下共享内存的创建、打开、映射和解映射，也通过实际的编程和调试对共享内存的实现和访问机制有了更加深入的理解。

在程序调试方面，我通过本次实验掌握了通过错误代码errno对程序出错原因进行分析并调试的方法。这种调试方法在解决系统调用出错的情况时大大提高了我对程序错误的定位和分析效率，避免了无意义的修改和调试。