# 快速排序问题

无 76　RainEggplant　2017******

## 1 问题

对于有 1,000,000 个乱序数据的数据文件执行快速排序。

### 1.1 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）。

### 1.2 实验平台和编程语言

自由选择 Windows 或 Linux。
编程语言不限。

## 2 设计思路

出于简便以及效率的考虑，我们采用多线程来实现该题目，因为这样可以共享内存。

题目要求采用快速排序。快速排序的核心思想是分治法，即将原始问题一步一步拆解为多个子问题进行求解。其实现步骤为：

1. **挑选基准值**：从数列中挑出一个元素，称为 “基准”（pivot）
2. **分割**：重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（与基准值相等的数可以到任何一边）。在这个分割结束之后，对基准值的排序就已经完成。
3. **递归排序子序列**：递归地将小于基准值元素的子序列和大于基准值元素的子序列排序。

可以看到，每次分割后的两部分间是相互独立，不存在共享数据的。因此，我们自然可以采用独立的线程分别对两部分进行排序，而不用考虑这些线程间的同步问题。

题目要求每个线程处理的数据小于 1000 以后不再分割，这个可以直接通过分割区间的长度进行判断。若大于等于 1000 则在分割时创建新线程执行，若小于 1000 则在本线程内分割执行。题目同时又要求产生的线程控制在 20 个左右，该要求可以通过线程池实现。我们只需要向线程池提交任务，由线程池进行调度。

最后，我们还需要阻塞主线程，直到排序完成（即所有排序线程退出）。由于每个排序线程是平等的，其无法直接知道自己是否是最后一个排序线程，因此我们设置了一个共享变量 `_n_thread` 来记录当前的排序线程数，并用 `_n_thread_lock` 实现互斥访问。当创建新线程时 `_n_thread` 增加 1，当线程退出时 `_n_thread` 减去 1。同时，每个线程结束时会产生 `_thread_done` 事件。这样，我们就可以在主线程里监听 `_thread_done` 事件，当且仅当事件发生且 `_n_thread` 为 0 时，判定排序完成。

## 3 代码实现

本次实验使用 Python 3.7，利用 `threading` 模块中的 `Lock` 和 `Event` 实现锁和事件，利用 `concurrent.futures` 中的 `ThreadPoolExecutor` 实现线程池。

我们将多线程快速排序封装为 `QuicksortService` 类，其代码如下：

```
from threading import Lock, Event
from concurrent.futures import ThreadPoolExecutor


class QuicksortService:
    def __init__(self, data, n_threads=20):
        self._data = data
        self._executor = ThreadPoolExecutor(max_workers=n_threads)
        self._n_thread = 0
        self._n_thread_lock = Lock()
        self._thread_done = Event()

    @property
    def data(self):
        return self._data

    def sort(self):
        self._thread_done.clear()
        if len(self._data) > 1000:
            with self._n_thread_lock:
                self._n_thread += 1
            self._executor.submit(self._quicksort, 0, len(self._data) - 1, True)
```

```
            while True:
                self._thread_done.wait()
                with self._n_thread_lock:
                    if self._n_thread == 0:
                        return
                    self._thread_done.clear()
        else:
            self._quicksort(0, len(self._data) - 1)

    def _quicksort(self, left, right, new_thread=False):
        if left >= right:
            return

        # choose median as pivot
        candidate_index = [left, (left + right) // 2, right]
        pivot_index = sorted(candidate_index, key=lambda val: self._data[val])[1]
        self._data[right], self._data[pivot_index] =\
            self._data[pivot_index], self._data[right]

        i = left
        j = right - 1
        pivot = self._data[right]
        while True:
            # from L to R, find the first element greater than or equal to the pivot
            while self._data[i] < pivot:
                i += 1
            # from R to L, find the first element less than the pivot
            while i < j and self._data[j] >= pivot:
                j -= 1
            # if markers do not intersect, then swap the elements
            if i < j:
                self._data[i], self._data[j] = self._data[j], self._data[i]
            else:
                break

        # swap pivot with the element where marker stops
        self._data[right] = self._data[i]
        self._data[i] = pivot
```

```
        if i - left > 1000:
            with self._n_thread_lock:
                self._n_thread += 1
            self._executor.submit(self._quicksort, left, i - 1, True)
        else:
            self._quicksort(left, i - 1)

        if right - i > 1000:
            with self._n_thread_lock:
                self._n_thread += 1
            self._executor.submit(self._quicksort, i + 1, right, True)
        else:
            self._quicksort(i + 1, right)

        if new_thread:
            with self._n_thread_lock:
                self._n_thread -= 1
            self._thread_done.set()
```

要调用它进行快速排序非常简单，只需要：

```
from quicksort_service import QuicksortService
sorter = QuicksortService(data, args.n_threads)
sorter.sort()
```

然后就可以通过 `sorter.data` 获取排序好的数据了。

# 4 实验结果

调用 `gen_rndnum.py` 可以生成指定数量的随机整数到随机文件。其用法为：

```
usage: gen_rndnum.py [-h] [-n NUMBER] [-o OUTPUT]

Generate a text file with specified number of random integers

optional arguments:
  -h, --help            show this help message and exit
  -n NUMBER, --number NUMBER
                        the number of random integers
```

```
  -o OUTPUT, --output OUTPUT
                        output filename
```

不提供参数直接执行，即可在同目录的 `random.txt` 中生成 1000000 个随机整数。

调用 `main.py` 即可读取指定文件中的随机数，使用多线程进行快速排序后，再输出到指定文件。其用法为：

```
usage: main.py [-h] [-i INPUT] [-o OUTPUT] [-n N_THREADS]

Perform quicksort on given data with given number of threads, then output the
result

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        input filename
  -o OUTPUT, --output OUTPUT
                        output filename
  -n N_THREADS, --n_threads N_THREADS
                        the number of threads used for quicksort
```

不提供参数直接执行，即读取在同目录的 `random.txt`，采用 20 个线程进行快速排序后，再输出到同目录的 `result.txt` 中。

以下为一组实验数据：

**random.txt**

```
455818699
12518897
1633332098
1679979080
...
```

**result.txt**

```
5615
8414
13136
13341
...
```

经检查，我们的程序正确地完成了快速排序。

这里插一句，由于 Python 解释器全局解析锁的存在，我们的多线程快速排序并不能利用多核 CPU，因此其速度可能还会比单线程慢。

# 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。

我们采用的是共享内存的方式，这在多线程程序里是自然的。如果采用管道或者消息队列，则还存在数据传递的开销，这会降低程序性能。因此，我们选用共享内存的方式。

## 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。

可以。

若采用管道，则不同之处在于当新建子进程执行分割后的数据的排序时，需要通过管道把待排序的数据传送给子进程，排序完成后又需要将结果通过管道传回。最后当所有子进程均退出时，排序结束。

若采用消息队列，其实现思路类似线程池，可以将排序的任务表示为消息，放入消息队列，然后由排序的线程接受消息，完成排序任务。