快速排序问题

无 36　　李思涵　　2013011187　　lisihan969@gmail.com

2016 年 1 月 20 日

目录

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 实验报告内容要求

- 写出设计思路和程序结构，并对主要代码进行分析；
- 实际程序运行情况；
- 对提出的问题进行解答；
- 体会或者是遇到的问题。

2 模块设计

由于多线程可以共享内存，使用多线程来实现我们的代码是很自然的选择。

在快排中，问题的规模是被逐步减小为一系列相互独立的问题。例如，若要用快速排序对一段序列进行排序，首先需要选取一个枢纽元，然后将这段数分为比枢纽元大的一段和比枢纽元小的一段。这个时候，问题就被减小为了对两段短序列进行排序，且二者的排序完全独立，故可以交给两个线程去完成。

为此，我们很自然的使用 Worker 线程的概念。我们将对某一段数据排序的任务抽象成一个任务对象，放入一个任务队列中。多个独立的 Worker 线程从任务队列中抽取队列，若需要排序的段小于 1000 则直接排序，若大于 1000 则先选取枢纽元并分为两段，然后将两个新的排序任务加入任务队列。这样，我们便可以实现对原序列的多线程排序。同时，我们可以通过控制线程池中 Worker 的数目方便的控制使用的线程数目。

然而，需要注意的是，由于不同的 Worker 在不同的线程中完成各自的排序任务，我们还需要在主线程中判断排序的完成。或者说，各个 Worker 线程应能在排序完成之后自动退出。然而，若按照一般设计，线程在应该退出时正处于等待抽取任务的状态下，无法主动判断是否应该退出。

为了解决这个问题，我们在 Worker 每次完成一个排序任务之后，检查自己刚刚完成的是否是最后一个任务。若是，此时剩余所有 Worker 一定都已经完成了他们的任务，并阻塞在抽取任务的语句上。于是，该 Worker 向任务队列中插入足够多个空任务并退出。剩余的线程在抽取到了新任务之后，发现其是空任务，便也退出。这样，这些 Worker 线程便可以自动退出，我们在主线程中只需要将其全部 join 便可保证排序任务的完成。

3 代码实现

我们使用 Python 3.5.0 来实现我们的代码。然而需要注意的是，由于全局语句锁的存在，python 中并不存在真正的多线程。也就是说，python 中的多线程只有在 I/O 等操作时可能能有帮助，在 CPU 密集的程序中实际上并不能提升效率。然而，个个线程之间的行为仍然和真正多线程时的一致，也就是说，多个线程的执行顺序不确定，当前活跃的线程也随时可能改变。故其仍然能用来验证我们的多线程实现是否正确。

具体来说，我们使用 queue.Queue 来实现任务队列，使用 threading 中的 Thread 和 Lock 来实现线程和锁。具体代码如下：

```
from queue import Queue
from sys import argv, exit
```

```
from threading import Thread, Lock
from time import sleep, time

class SortJob(object):
    """docstring for SortJob"""
    def __init__(self, l, begin, end, lock):
        super().__init__()

        self.l = l
        self.begin = begin
        self.end = end
        self.lock = lock

    def run(self, pool):
        if self.l is None:  # A padding.
            return True

        with self.lock:
            part = self.l[self.begin:self.end]

        if len(part) < 1000:
            part.sort()
            with self.lock:
                self.l[self.begin:self.end] = part
        else:
            mid = len(part) // 2
            pivot = sorted([part[0], part[mid], part[-1]])[1]

            left = []
            equal = []
            right = []
            for x in part:
                if x < pivot:
                    left.append(x)
                elif x == pivot:
                    equal.append(x)
                else:
                    right.append(x)

            part = left + equal + right

            with self.lock:
```

```
                self.l[self.begin:self.end] = part

            separate1 = self.begin + len(left)
            separate2 = self.end - len(right)
            pool.add_job(SortJob(self.l, self.begin, separate1, self.lock))
            pool.add_job(SortJob(self.l, separate2, self.end, self.lock))

        return False

class ThreadPool(object):
    def __init__(self, size):
        super().__init__()

        self.jobs = Queue()
        self.monitors = [Thread(target=self.monitor)
                         for i in range(size)]
        for monitor in self.monitors:
            monitor.start()

        self.jobs_left = 0
        self.jobs_left_lock = Lock()

    def monitor(self):
        while True:
            job = self.jobs.get()
            if job.run(self):  # run return true, so we should quit.
                return

            with self.jobs_left_lock:
                self.jobs_left -= 1
                if self.jobs_left == 0:
                    # No more jobs.
                    # Other monitors are waiting.
                    # Add padding jobs.
                    self.__add_padding_jobs()

    def add_job(self, job):
        with self.jobs_left_lock:
            self.jobs_left += 1
        self.jobs.put(job)

    def join(self):
```

```
        for monitor in self.monitors:
            monitor.join()

    def __add_padding_jobs(self):
        for i in range(len(self.monitors)):
            self.jobs.put(SortJob(None, None, None, None))  # Do NOT use add_job


if __name__ == '__main__':
    if len(argv) != 3:
        print('Usage:', argv[0], '<thread number> <iters>')
        exit()

    thread_num = int(argv[1])
    iters = int(argv[2])

    numbers = []

    with open('random.txt') as f:
        for line in f:
            numbers.append(float(line))

    start_time = time()
    for i in range(iters):
        numbers_copy = numbers[:]
        lock = Lock()
        pool = ThreadPool(thread_num)

        pool.add_job(SortJob(numbers_copy, 0, len(numbers_copy), lock))
        pool.join()
    print((time() - start_time) / iters)

    # Test
    last = -1
    for x in numbers:
        if last > x:
            print('Not sorted.')
            break
        last = x
    else:
        print('sorted.')
```

```
    with open('sorted.txt', 'w') as f:
        for number in numbers:
            print('{:6f}'.format(number), file=f)
```

4 实验结果

我们使用大小为 20 的线程池进行实验，结果如下：

```
$ python3 main.py 20 1
2.3255510330200195
sorted.
```

我们也可以通过查看 `random.txt` 来确认排序确实已经完成了，在这里我们取它的前 20 行：

```
$ head -20 sorted.txt
0.000001
0.000003
0.000003
0.000004
0.000005
0.000006
0.000008
0.000008
0.000008
0.000009
0.000012
0.000012
0.000013
0.000013
0.000013
0.000014
0.000014
0.000017
0.000017
0.000022
```

可以看到，我们确实实现了数据的并行排序。

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。

在本次实验中，我们使用的是共享内存的方式。因为我们使用的是多线程而不是多进程，内存直接就是共享的，使用共享内存并不会带来额外的麻烦或者开销。相比而言，管

道/消息队列都涉及到数据的传递，故会有复制发生，会有额外的开销。

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。

同样可以解决。若使用管道的话，一个进程通过开启两个子进程，并将其需要排序的数据通过管道传递出去，便可以完成这一段的排序任务。最后只要保证所有进程都已经完成任务退出，便可以保证排序完成。对于消息队列，我们可以使用和线程池一样的思路，将排序任务放入消息队列，由 Worker 线程/进程接受消息并完成排序任务。

6 实验感想

开始进行本次实验时，我使用的是最朴素的方式：在对一段进行排序时，创建两个子线程，等待他们运行完成。同时，若线程数达到了 20 就不再创建新线程，而是等待有线程退出。然而，这样设计有一个问题：很快所有进程就会进入等待状态，程序没有办法正常运行。

为此，我修改了程序，在线程开始之后再判断是否达到了 20 个线程。然而，虽然线程不会卡死了，但又出现了新的问题：同时创建的线程过多，python 的线程库没有办法创建更多的线程了 QAQ。

最后，我采用了进程池的实现方法，并用上面提到的方法解决了任务的退出判定，才真正解决了并行排序问题。可以说，这次实验是一个摸索和试验的过程，我对线程池也有了一个直观的认识。