Project 2_2
快速排序问题

班级：无 53

姓名：陈相宁

学号：2015011033

Email：c.xiangning1997@gmail.com

② 快速排序问题

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。

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）。

实验平台和编程语言：

自由选择 Windows 或 Linux。

编程语言不限。

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。
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。

一、 实验思路：

采用多线程处理本实验。

在快速排序问题中，采用分治思想：首先找到其中一个基数的正确位置，从而将序列分割为比基数小的一段以及比基数大的一段，而这两段序列的排序相互独立，因此可以交给两个线程分别完成。

许多多线程编程方案中，线程会即时创建，即时销毁，尽管与创建进程相比，创建线程的时间已经大大的缩短，但是如果提交给线程的任务是执行时间较短，而且执行次数极其频繁，那么服务器将处于不停的创建线程，销毁线程的状态。本实验中，限定线程数量为 20 个左右，并且很多线程完成的任务只是将序列一份为 2，执行时间较短。因此，我就自然地想到了利用线程池来解决本问题。

线程池采取这样的思路：确定数量的线程被预先创立并置于线程池中，另外有一任务队列处理任务的生成及结束，线程处理完任务后并不被销毁而是继续从任务队列中领任务，这样便能够避免多次创立线程，从而节省线程创建和销毁的开销，能带来更好的性能和稳定性。

排序问题本身没有特别大的难度，不过由于涉及到共享空间，还是需要利用互斥控制同一时刻只有一个线程读写序列。

二、 代码实现：

编程语言为 python，采取 queue 库中的 Queue 类实现线程池中的任务队列，采取 threading 中的 Thread 类实现线程，Lock 类实现互斥访问，Semaphore 类实现信号量。

1. Generate_Randoms.py 生成随机数：

比较 trivial。

```
1  from random import random
2
3  with open('Randoms.txt', 'w') as f:
4      for k in range(int(1e6)):
5          print('{:6f}'.format(random()), file = f)
```

2. Thread_Job 类：

自建 Thread_Job 类抽象任务。初始化时输入该任务针对的子序列的起止点。do_job 方法中，首先，利用 with lock 语句实现互斥读取子序列。之后，如果子序列长度小于 1000，则

不再分割，直接排序并互斥写回原序列。否则，将子序列中最后 1 个数作为基数，找出小于基数的所有数作为 less，大于基数的所有数作为 greater，等于基数的所有数作为 equal。将原序列对应位置更改为 less + equal + greater，同时在任务队列中添加两个新的排序任务并对表示任务量的信号量执行 2 次 V 操作。

```
class Thread_Job(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def do_job(self, work_queue, work_num):
        global Numbers, lock
        with lock:
            numbers = Numbers[self.start:self.end]
        if len(numbers) < 1000:
            with lock:
                Numbers[self.start:self.end] = sorted(numbers)
        else:
            base = numbers[-1]
            less, equal, greater = [], [], []
            for i in numbers:
                if i < base:
                    less.append(i)
                elif i == base:
                    equal.append(i)
                else:
                    greater.append(i)
            numbers = less + equal + greater
            with lock:
                Numbers[self.start:self.end] = numbers
            work_queue.put(Thread_Job(self.start, self.start + len(less)))
            work_queue.put(Thread_Job(self.end - len(greater), self.end))
            work_num.release()
            work_num.release()
        return work_queue, work_num
```

3. Thread_Pool_Manager 类：

自建 Thread_Pool_Manager 类抽象线程池。初始化时生成空任务队列，初始值为 0 的表示 job 数量的信号量 work_num 用于防止从空队列中提取任务，以及名为 mutex 的锁保证同一时间只有一个线程从 work_queue 中提取任务。其中有 4 个方法：

① start_threads 用于启动线程，启动既定数量的线程，执行函数为 do_job。其中关键的一步是将线程池中的线程设置为守护线程 daemon，这样所有任务都完成后，所有线程自动退出。

② do_job 中设置一个无限循环，首先对 work_num 执行 P 操作，解除阻塞后在锁变量 mutex 的保护下从任务 work_queue 中提取一个任务，完成任务同时更新 work_queue 以及 work_num。之后关键的一步是调用 work_queue 的 task_done 函数告诉队列这一任务已经完

成，这一步是之后的 join 函数能够正常退出的保证。

③ join 函数即执行 work_queue 的 join 操作，必须等到所有任务都完成后程序才继续执行。

④ add_job 函数用于将第一个任务加入 work_queue 并对 work_num 执行 V 操作。

```
44  class Thread_Pool_Manager(object):
45      def __init__(self, thread_num):
46          self.thread_num = thread_num
47          self.work_queue = Queue()
48          self.work_num = Semaphore(0)
49          self.mutex = Lock()
50
51      def start_threads(self):
52          for i in range(self.thread_num):
53              thread = Thread(target = self.do_job)
54              thread.daemon = True    # set thread as daemon
55              thread.start()
56
57      def do_job(self):
58          while True:
59              self.work_num.acquire()
60              with self.mutex:
61                  thread_job = self.work_queue.get()
62              self.work_queue, self.work_num = thread_job.do_job(self.work_queue,
63                  self.work_num)
64              self.work_queue.task_done()
65
66      def join(self):
67          self.work_queue.join()
68
69      def add_job(self, job):
70          self.work_queue.put(job)
71          self.work_num.release()
72
```

全部代码：

Thread_Sort.py：

```python
from queue import Queue
from threading import Thread, Lock, Semaphore
import threading
from time import time, sleep

Numbers = []
thread_num = 20
lock = Lock()
with open('Randoms.txt', 'r') as f:
    for line in f:
        Numbers.append(float(line))

class Thread_Job(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def do_job(self, work_queue, work_num):
        global Numbers, lock
        with lock:
            numbers = Numbers[self.start:self.end]
        if len(numbers) < 1000:
            with lock:
                Numbers[self.start:self.end] = sorted(numbers)
        else:
            base = numbers[-1]
            less, equal, greater = [], [], []
            for i in numbers:
                if i < base:
                    less.append(i)
                elif i == base:
                    equal.append(i)
                else:
                    greater.append(i)
            numbers = less + equal + greater
            with lock:
                Numbers[self.start:self.end] = numbers
            work_queue.put(Thread_Job(self.start, self.start + len(less)))
            work_queue.put(Thread_Job(self.end - len(greater), self.end))
            work_num.release()
            work_num.release()
        return work_queue, work_num

```

```python
class Thread_Pool_Manager(object):
    def __init__(self, thread_num):
        self.thread_num = thread_num
        self.work_queue = Queue()
        self.work_num = Semaphore(0)
        self.mutex = Lock()

    def start_threads(self):
        for i in range(self.thread_num):
            thread = Thread(target = self.do_job)
            thread.daemon = True    # set thread as daemon
            thread.start()

    def do_job(self):
        while True:
            self.work_num.acquire()
            with self.mutex:
                thread_job = self.work_queue.get()
            self.work_queue, self.work_num = thread_job.do_job(self.work_queue,
                self.work_num)
            self.work_queue.task_done()

    def join(self):
        self.work_queue.join()

    def add_job(self, job):
        self.work_queue.put(job)
        self.work_num.release()

thread_pool_manager = Thread_Pool_Manager(thread_num)
thread_pool_manager.add_job(Thread_Job(0, len(Numbers)))
thread_pool_manager.start_threads()
start_time = time()
thread_pool_manager.join()
print('sort time:', time() - start_time)

with open('Sorted_Numbers.txt', 'w') as f:
    for number in Numbers:
        print('{:6f}'.format(number), file = f)
print('file saved!')
```

实验结果：

```
peter_cxn@Peter-cxndeMacBook-Pro ~/desktop/Project2_2 python3 Thread_Sort.py
sort time: 1.8543570041656494
file saved!
```

可见，线程池中放置 20 个线程，则 1e6 个 float 型数据排序需要 1.85 秒，排序好的数据输入到 Sorted_Numbers.txt 中。从速度来讲，线程并行排序的速度高于 c 语言标准库中的函数。

前 30 个数：

```
0.000000
0.000001
0.000003
0.000005
0.000005
0.000006
0.000006
0.000008
0.000009
0.000011
0.000013
0.000015
0.000016
0.000016
0.000016
0.000017
0.000017
0.000018
0.000022
0.000022
0.000027
0.000028
0.000029
0.000031
0.000031
0.000035
0.000035
0.000037
0.000037
0.000039
```

后 30 个数：

```
0.999976
0.999976
0.999977
0.999977
0.999977
0.999978
0.999978
0.999978
0.999979
0.999982
0.999982
0.999982
0.999983
0.999984
0.999984
0.999985
0.999985
0.999986
0.999986
0.999987
0.999988
0.999994
0.999994
0.999995
0.999996
0.999997
0.999997
0.999999
0.999999
1.000000
```

通过循环逐一检测以及抽样的方法确认所有数据均正确排序。

三、 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问题，请解释你做出这种选择的理由。

我采取的是“共享内存”的方式。因为采用多线程，内存本身就是共享的，最为直观，且不会带来额外的开销。我在本实验中设置了 2 个全局变量，就是共享内存最为直接的体现。如果使用管道或者消息队列的方式都涉及消息的复制以及传播，都会带来时间及空间上的损耗。

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。

① 管道：采用多进程，当父进程开启两个子进程时，通过管道将数据传播至子进程即可，

当所有进程完成时，排序任务也就自然完成了。

② 消息队列：将需要完成的任务塞入消息队列中，由线程或者进程接收消息完成排序任务即可。