操作系统实验 2-2
并行快速排序

马栩杰
2014011085
无 43 班

2016 年 12 月 2 日

1 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。

2 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）。

3 程序设计

3.1 问题分析

我们需要解决的问题有以下特点：

- CPU 密集 -> 需要避免等待 IO 浪费时间
- 需要使用大量数据 -> 要避免在线程或进程间反复拷贝数据
- 需要创建大量新的线程/进程 -> 希望创建开销尽可能小

- 进行计算的线程/进程间相对比较独立，在计算过程中没有大量数据交换需求

从这几个特点来看，多线程 + 共享内存就是一个自然而然的选择了。

4 整体设计

整体上来说，程序可以分成两个主要的部分：一个控制线程，负责监视运行中的线程数量，并且在有需要的时候从任务队列中获取任务创建新线程；若干个计算线程，由控制线程创建，每个计算线程执行一次快排的划分过程，并且将划分后的区间作为新的任务添加到任务队列中，然后退出。

5 计算线程

在将计算与线程任务管理两部分功能独立以后，计算线程的工作就很单一了。计算线程只负责进行快排的计算，在创建线程时将该线程要处理的数据区间作为参数传入，然后该线程对区间进行一个判断，如果区间长度小于 1000 则直接执行插入排序，如果区间长度大于 1000 则按照通常快排的做法，随机选取一个元素作为中间值，然后对区间内的元素按照与中间值的相对大小进行划分，划分后，产生两个子任务，将这两个子任务加入任务队列，最后在退出线程时通知管理线程检查任务队列和工作线程数。

其中需要注意的是待排序的数据存储在一个全局的数组当中，由于计算线程之间在访问数组时访问的区间是相互没有重叠的，所以对该数组的访问并不需要加锁。

代码实现如下：

```
1  void CalcThread(void *argv) {  // 计算线程
2    SortParam *param = (SortParam *)argv;
3    int st = param->st;  // 快速排序起始位置
4    int ed = param->ed;  // 快速排序结束位置
5    if (ed - st < BLOCK_SIZE) {  // 如果区间很小的话，不采用快排，而是直接进行插入排序
6      InsertSort(st, ed);
7    } else {  // 否则需要快排
8      // 为避免快排效率恶化，随机取一个元素作为中值
9      int rand_ind = st + rand() % (ed - st);
10     int mid = numbers[rand_ind];
11     numbers[rand_ind] = numbers[ed - 1];
12
13     // 快排的主过程，将小于 mid 的元素移动到区间的前一半
14     int i = 0, j = 0;
15     while (i != ed - 1) {
16       if (numbers[i] < mid) {
17         int t = numbers[j];
18         numbers[j++] = numbers[i];
19         numbers[i] = t;
20       }
21       ++i;
22     }
```

```
23
24      // 划分结束
25      numbers[ed - 1] = numbers[j];
26      numbers[j] = mid;
27
28      // 然后将子任务加入任务队列
29      int st1 = st, ed1 = j;
30      int st2 = j + 1, ed2 = ed;
31      pthread_mutex_lock(&mutex);
32      Push(st1, ed1);  // 任务 1（左半边）：[起始点，中间点）
33      Push(st2, ed2);  // 任务 2（右半边）：[中间点 +1，终点）
34      pthread_mutex_unlock(&mutex);
35    }
36    pthread_mutex_lock(&mutex);
37    --working_threads;  // 在结束计算线程的时候将当前工作线程计数器减 1
38    pthread_mutex_unlock(&mutex);
39    pthread_cond_broadcast(&thread_manager_cond);  // 最后告知管理线程检查任务队列
40    // 完成一个子任务的计算后线程自然退出
41  }
```

6 管理线程

管理线程要做的工作是判断当前的任务队列中是否有未处理的任务，以及当前的工作线程数量是否还没有达到工作线程数的上限，如果条件符合的话就创建新的线程，否则阻塞在条件变量上等待被计算线程退出时唤醒。

代码实现如下：

```
1  void ThreadManager() {  // 管理线程主函数
2    while (true) {
3      pthread_mutex_lock(&mutex);  // 为了检查任务队列和工作线程数，先取得锁
4      if (queue_size > 0  // 如果任务队列非空
5        && working_threads < MAX_THREADS_NUM) {  // 并且正在工作的线程数未达到上限
6        ++working_threads;
7        pthread_t thread;
8        SortParam *task = Pop();  // 则从任务队列中取出一个任务
9        printf("new task (%d, %d), working_threads = %d, queue_size = %d\n",
10             task->st, task->ed, working_threads, queue_size);
11       // 然后创建一个新的线程处理这个任务
12       pthread_create(&thread, NULL, (void *(*)(void *))CalcThread, task);
13       pthread_mutex_unlock(&mutex);
14     } else if (queue_size == 0 && working_threads == 0) {
15       // 否则，如果（任务队列空 && 没有正在工作的线程）
16       // 这种情况说明计算已经结束，不可能新的子任务再产生了
```

```
      pthread_mutex_unlock(&mutex);
      return;  // 管理线程退出，向主函数表明计算已经结束
    } else {
      // 否则，(任务队列空 // 工作线程数达到上限 // 有正在工作的线程)
      pthread_mutex_unlock(&mutex);
      // 阻塞地等待一个工作线程退出后唤醒管理线程
      pthread_cond_wait(&thread_manager_cond, &mutex);
      pthread_mutex_unlock(&mutex);
    }
  }
}
```

7 实验结果与分析

7.1 编译运行

与前一次实验相同，在 Ubuntu 14.04.5 下用 make 编译。

运行

```
./main in.txt out.txt
```

7.2 运行结果

输入含有乱序 0 到 999999 整数的文件，执行程序。在等待十几秒钟之后，得到了输出文件。打开文件可以看到数据已经被正确地排序了。以最后 10 行为例：

```
999990
999991
999992
999993
999994
999995
999996
999997
999998
999999
```

显然排序结果是正确的。

另外，为方便验证管理线程的正确性，在排序过程中，创建新的计算线程时，也输出了几个重要的变量。取 log 的一部分：

```
new task (541679, 542098), working_threads = 20, queue_size = 82
new task (512206, 512718), working_threads = 20, queue_size = 81
new task (512719, 514226), working_threads = 20, queue_size = 80
new task (545702, 546268), working_threads = 20, queue_size = 79
```

```
new task (546269, 548785), working_threads = 20, queue_size = 78
new task (544368, 545403), working_threads = 16, queue_size = 81
new task (545404, 545701), working_threads = 17, queue_size = 80
new task (237172, 238122), working_threads = 18, queue_size = 79
new task (238123, 238207), working_threads = 19, queue_size = 78
new task (782084, 783066), working_threads = 20, queue_size = 77
new task (783067, 783147), working_threads = 20, queue_size = 76
```

可以看到在绝大多数时间工作线程数都达到上限 20，一个计算线程退出后创建一个新的计算线程时任务队列也确实减少了 1；当有几个计算线程几乎同时退出时管理线程也可以很快地补充若干个计算线程，让 CPU 资源得到更充分的利用。

8 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。

复制一下在程序设计一节中提到的内容：

- CPU 密集 -> 需要避免等待 IO 浪费时间
- 需要使用大量数据 -> 要避免在线程或进程间反复拷贝数据
- 需要创建大量新的线程/进程 -> 希望创建开销尽可能小
- 进行计算的线程/进程间相对比较独立，在计算过程中没有大量数据交换需求

由于在计算过程中需要创建大量的子任务，而创建线程的开销要比创建进程的开销更小，因此选择了多线程的实现。在多线程的情况下，共享内存是不需要额外操作的，使用共享内存也就避免了线程间数据的复制。并且由于线程间访问的数据区间相对独立，因此也避开了复杂的线程安全问题。综上，选择了多线程共享内存的实现方式。

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。

另外两种实现方式也可以解决问题。

管道：同样用一个控制进程和若干个子进程来实现，在创建子进程时通过管道将要排序的数据区间发送到子进程中，即可完成排序。

消息队列：可以通过将当前实现中的任务队列改成一个消息队列，计算线程一旦启动、在全部任务计算完成前就不再退出，用手动编写线程池的实现方式，每个线程循环地从消息队列中领取任务并且执行。

9 实验总结

由于有前一个实验的经验，现在使用 pthread 库编写多线程编码已经是轻车熟路，所以这次整体系统结构和编码风格都比前一次实验更好。