

实验二：高级进程间通信

wangqr

0. 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。

1. 实现方式

快速排序的算法分为两步：拆分和对拆分结果递归排序。后一步由于拆分的两部分数据不重叠，因此可以不加互斥地在线程中对它们进行排序。这里，我们选用原地，基准元选为前、中、后三个元素的中位数版本的快速排序作为基础进行并行化修改。

此题的实验步骤中提到“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”，这是不合理的：因为新的进程将以 2 的幂的速度产生，而且由于新的线程如果不产生，则原来的线程也无法退出，在对总线程数加限制后将陷入死锁。本题的实现方式与实验一类似：定义一个只能单线程访问的任务队列，并运行预先指定数目的 Worker 线程，每个 Worker 线程从任务队列中领取任务，完成分割操作后将新产生的两个任务送入任务队列。主线程需在最初向任务队列插入初始的 1 个任务。

本题另一个实现难点是确定线程退出条件。这里采用任务计数的方式。我们使用一个变量存储尚未完成任务数（它与任务队列中的任务数，也就是尚未被领取的任务数并不相同），当一个线程发现其当前任务完成后尚未完成任务数为 0 时，它将向任务队列中插入 Worker 数目减一（因为除去自身）个退出任务，各线程收到退出任务后退出。

2. 代码实现

排序的数据和 Worker 线程存放在全局数组中。任务队列直接用数组实现。由于总任务数不会超过 $2\max\{N-N_t, 0\}+1$ （ N 是待排序数数量， N_t 是任务分割阈值；利用归纳法可证，初始条件是待排序数数量小于等于 1000 时总任务数等于 1）。实现时需要在此值的基础上加上退出任务的数量。读、写任务队列各有一个互斥锁加以控制。此外还有一个信号量（sem_queue）用于记录任务队列中的任务数、一个整数（task_remain）配合互斥锁记录尚未完成任务数。

```
uint32_t data[N];

pthread_t worker[WORKER_NUM];

struct queue_item {
    uint32_t *start;
    size_t length;
} queue[WORKER_NUM + 2*(N-MAX_THRESH)],
    *queue_head = queue, *queue_end = queue + 1;
size_t task_remain = 1;
pthread_mutex_t mutex_remain, mutex_queue_read, mutex_queue_write;
sem_t sem_queue;
```

Worker 线程中，在循环中从任务队列领取任务，若任务为退出任务则退出，否则根据任务中待排序数数量决定执行的操作。当待排序数数量小于任务分割阈值时，不再分割任务，直接在该线程内对任务范围内的数进行排序；排序完成后，将尚未完成任务计数减一，并判断是否需要插入退出任务。当待排序数数量大于任务分割阈值时，对待排序数据进行分割；分割完成后，将尚未完成任务计数加一，并插入分割后的两个任务。

```
void *qsort_worker(void *param __attribute__((unused))) {
    struct queue_item *current;
    uint32_t temp;
    while (1) {
        sem_wait(&sem_queue);
        pthread_mutex_lock(&mutex_queue_read);
        current = queue_head++;
        pthread_mutex_unlock(&mutex_queue_read);
        if (current->length > N) {
            // This is a quit signal
            return NULL;
        }
        else if (current->length < MAX_THRESH) {
            qsort(current->start, current->length, 4, uint_comp);
            pthread_mutex_lock(&mutex_remain);
            --task_remain;
            if (task_remain == 0) {
                pthread_mutex_lock(&mutex_queue_write);
                for (unsigned i = 1; i < WORKER_NUM; ++i) {
                    (queue_end++)->length = N + 1;
                    sem_post(&sem_queue);
                }
                pthread_mutex_unlock(&mutex_queue_write);
                return NULL;
            }
            pthread_mutex_unlock(&mutex_remain);
        }
        else {
            uint32_t *lo = current->start;
            uint32_t *mid = current->start + (current->length >> 1);
            uint32_t *hi = current->start + (current->length - 1);

            // sort LO, MID, HI
            ...

            // partition
            ...

            // assign work [lo, right] & [left, hi]
        }
    }
}
```

```

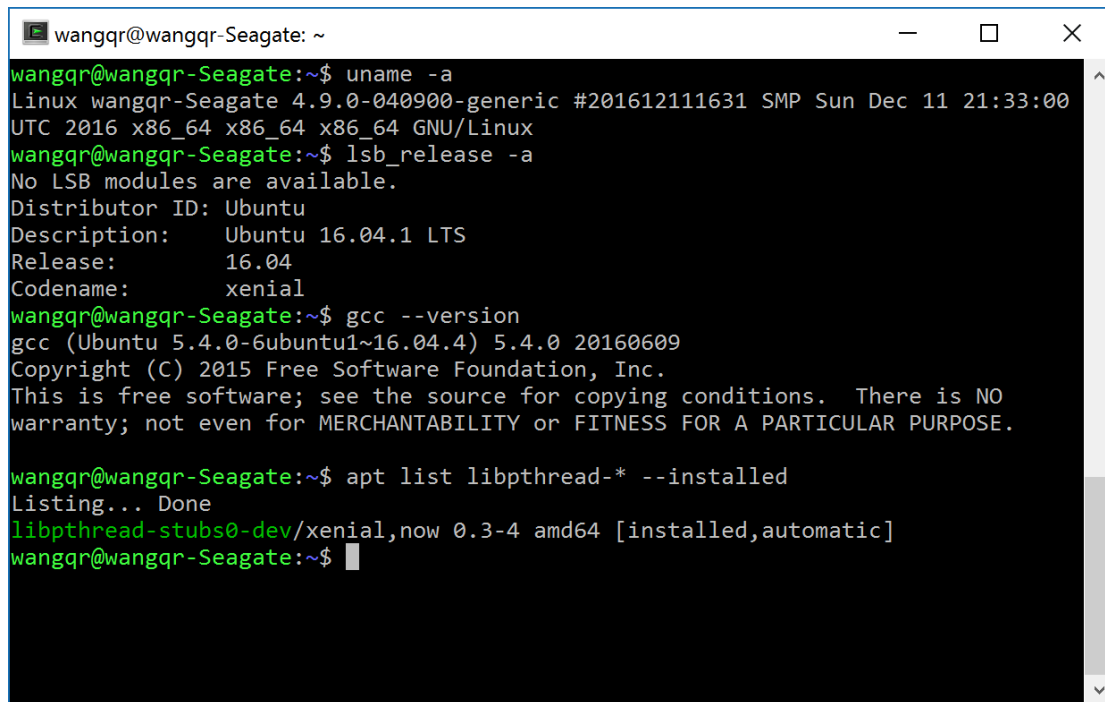
        // push larger partition first
        pthread_mutex_lock(&mutex_remain);
        ++task_remain;
        pthread_mutex_unlock(&mutex_remain);
        if (right - lo < hi - left) {
            pthread_mutex_lock(&mutex_queue_write);
            queue_end->start = left;
            (queue_end++)->length = hi - left + 1;
            sem_post(&sem_queue);
            queue_end->start = lo;
            (queue_end++)->length = right - lo + 1;
            sem_post(&sem_queue);
            pthread_mutex_unlock(&mutex_queue_write);
        } else {
            pthread_mutex_lock(&mutex_queue_write);
            queue_end->start = lo;
            (queue_end++)->length = right - lo + 1;
            sem_post(&sem_queue);
            queue_end->start = left;
            (queue_end++)->length = hi - left + 1;
            sem_post(&sem_queue);
            pthread_mutex_unlock(&mutex_queue_write);
        }
    }
}
}

```

完整的代码在 `main.c` 中，同一文件夹下的 `demo.sh` 提供了从编译至程序运行的完整演示。同时提供的 `std.c` 是单线程、直接调用 `qsort` 的参照版本，用作对拍的参考程序，以检验 `main.c` 的输出正确性。

3. 运行结果

在如下的系统环境中对所编写的程序进行了测试：

A terminal window titled 'wangqr@wangqr-Seagate: ~' with standard window controls. The terminal output shows the results of several system commands: 'uname -a' displays kernel and hardware details; 'lsb_release -a' shows Ubuntu 16.04.1 LTS information; 'gcc --version' shows GCC 5.4.0 details; and 'apt list libpthread-* --installed' shows 'libpthread-stubs0-dev' is installed. The prompt is green, and the output is white on a black background.

```
wangqr@wangqr-Seagate: ~$ uname -a
Linux wangqr-Seagate 4.9.0-040900-generic #201612111631 SMP Sun Dec 11 21:33:00
UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
wangqr@wangqr-Seagate: ~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.1 LTS
Release:        16.04
Codename:       xenial
wangqr@wangqr-Seagate: ~$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

wangqr@wangqr-Seagate: ~$ apt list libpthread-* --installed
Listing... Done
libpthread-stubs0-dev/xenial,now 0.3-4 amd64 [installed,automatic]
wangqr@wangqr-Seagate: ~$
```

图 1 测试机器系统环境

测试结果如图 2。

这里我们输入、输出文件均采用二进制文件的方式存储，每 4 个字节作为一个无符号整形（uint32_t）数据。随机数据的生成采用直接从字符设备/dev/urandom 复制的方式。该设备是*nix 系统中的非阻塞随机数生成器。结果的比较可以直接采用 diff 程序实现，但这里为了直观，计算了两个程序的输出文件的 SHA256 散列值。

从输出结果看，数据确实已经从小到大排序好了，且多线程版本与非多线程版本结果一致。

从运行时间看，多线程版本的用户态 CPU 占用时间(user)大于程序运行的总时间(real)，说明程序确实被分配到了 CPU 的多个核心上运行。同时对比多线程程序和参考程序的运行时间（real）可以看出，并行化快速排序确实对性能有提升。

```
wangqr@wangqr-Seagate: ~/src/os/hw2
wangqr@wangqr-Seagate:~/src/os/hw2$ ./demo.sh

* Compiling

* Generating unsorted data
1+0 records in
1+0 records out
4000000 bytes (4.0 MB, 3.8 MiB) copied, 0.0336947 s, 119 MB/s

* Printing unsorted data
000000 402599179 1797478615 2219082153 336871786
000010 920707835 2927988596 337627216 3962646967
000020 1208672939 3277190248 4225222825 2689660428
000030 1657917924 140693362 2395600356 2858067337
000040 2537583202 1406174477 3403092226 3240859545
000050 1233791142 2764343022 1765202704 2757591467
000060 2083439381 3358456090 4248998619 2767326092
000070 2781605436 2923014948 2677145285 3955406892
000080 4233415891 3230276985 2510102650 2942570715
000090 2579043104 1728451544 1202459320 3302970287
.....
3d0870 190126880 2251042807 3421838325 919076977
3d0880 3707427090 3986138468 1649153156 3062075922
3d0890 1611352830 698294155 987638718 3585972565
3d08a0 151458375 397275315 1619482413 147526708
3d08b0 2345921477 1372425924 3226065953 3397529077
3d08c0 4033962610 1634021188 1469190267 3086164890
3d08d0 979044036 1477665416 542840310 3038377054
3d08e0 1495360763 1686407875 2964616763 1189602800
3d08f0 470493696 593522600 1965127471 2742296527
3d0900

* Sorting data

real    0m0.123s
user    0m0.148s
sys     0m0.008s

* Sorting data (reference program)

real    0m0.206s
user    0m0.184s
sys     0m0.016s

* Printing sorted data
000000      5512      18983      19667      21110
000010     29636     30772     31980     32378
000020     34323     38628     39817     50131
000030     62958     63115     70972     73469
000040     74520     82220     83895     86514
000050     90168     90574     90778     91286
000060     92287     93436     94834     98955
000070    101540    113194    119136    126295
000080    126878    130436    130712    133013
000090    134301    144134    148863    155176
.....
3d0870 4294800220 4294803200 4294804058 4294805934
3d0880 4294806680 4294819347 4294820171 4294822697
3d0890 4294826704 4294829736 4294846562 4294864228
3d08a0 4294873241 4294879876 4294880019 4294883811
3d08b0 4294883978 4294884156 4294884470 4294890838
3d08c0 4294900138 4294906561 4294908182 4294912919
3d08d0 4294919486 4294919945 4294925618 4294927848
3d08e0 4294932287 4294936300 4294938940 4294939319
3d08f0 4294946966 4294947612 4294953897 4294957741
3d0900

* Comparing result (SHA256 should match)
7ea3db1ab6a326d5d47eec386b398268bb3380248e0f5dfa2e59be6d845e4dca  sorted
7ea3db1ab6a326d5d47eec386b398268bb3380248e0f5dfa2e59be6d845e4dca  sorted_std
wangqr@wangqr-Seagate:~/src/os/hw2$
```

图 2 运行结果

4. 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。

此题采用的是多线程并行的方式，因此不存在进程间通信的措施。同一进程的内存空间原本就可以被其所有线程直接访问。因此它与共享内存类似，但不涉及进程间共享内存时的措施。

因为快速排序算法是原地算法（排序过程直接在原数据数组上进行），如果采用管道将待排序数据和排序完数据在进程间来回发送，则来回发送过程将消耗大量时间；如果采用消息队列配合共享内存，则与多线程的实现方式类似，但由于启动新进程、进程间通信等的开销比线程间加锁同步更大，因此效率更低。从上述因素考虑，快速排序并行化采用线程更加合理。

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。

上面已经提到，另外的机制同样可以解决该问题。

采用管道时，主进程读入待排序数据至内存，启动预设个 Worker 进程，然后向空闲的进程经由管道传输分配的任务（将待排序数据传过去）。进程排序或分割完成后，将处理后的数据和可能产生的新任务经由管道发送给主进程，主进程更新内存中的数据，并将收到的新任务加入任务队列。当队列非空时，持续向 Worker 进程分发任务，直至所有任务完成，即可杀掉所有 Worker 进程并退出。

采用消息队列与上述类似，只是任务由消息队列发送给 Worker 进程。此时不再需要任务队列，主进程也不再需要指定任务由哪个 Worker 执行，只需将任务加入消息队列即可。Worker 进程可通过共享内存的方式访问待排序数据。