**电 子 科 技 大 学**

作 业 报 告

# 学生姓名：王冉恒 学 号：202422900232 指导教师：李林学生E-mail：[1976569841@qq.com](mailto:djhuang_1@qq.com)

**一、作业名称**

编写一个线程池，实现对海量数据排序

# 二、作业要求

输入：给定一个目录，其中包含大量的普通文件，每个文件中又包含了大量的数据每个数据都是 64 位的有符号数，文件/数据格式可自由设定，数据随机生成。

典型设置：所有文件包含的总的数据量达到 128GB，每个文件包含的数据个数并不平均，其空间占用量少的可能仅几十 KB，多的可能上 GB 文件个数可以达到 10 万个。

输出：按照从小到大的顺序，将排好序的数据写入到指定文件中，文件格式自定义。限制：用于缓存文件数据、中间排序结果的内存远小于 128GB，比如只能使用 64MB

缓存。

要求：建立一个包含若干线程的线程池，可以按照归并排序思路，在多线程间完成排序，线程间交换数据按照粗粒度锁、免锁、无锁实现方式，从低到高综合评分，要尽可能减少数据复制操作。

# 三、设计与实现

# 线程池实现

本作业在Pool.hpp 中实现了一个基本的线程池结构，允许用户提交任务以在池中异步执行。当创建一个 ThreadPool 对象时，需要为其指定了线程的数量。该线程池会创建指定数量的线程，用来执行用户交给线程池的任务，当任务提交过来，任务队列没有空闲空间，就会使得提交任务的线程阻塞，直到有空间，才会继续提交。

## 主要组件

## 线程池结构体成员

pthread\_t \*threads： 工作线程数组

int thread\_count： 线程数量

TaskQueue task\_queue： 任务队列

int stop： 停止标志

Task tasks[MAX\_TASKS]; // 任务数组

int head; // 队列头

int tail; // 队列尾

int count; // 当前任务数量

pthread\_mutex\_t lock; // 互斥锁

pthread\_cond\_t not\_empty; // 任务到达的条件变量

pthread\_cond\_t not\_full; // 任务队列不满的条件变量

**主要方法**

1. ThreadPool \*create\_thread\_pool(int thread\_count)：创建线程池，根据传入的参数，创建相等数量的工作线程，使用pthread\_create函数创建，指定其创建之后运行的函数为work函数，使得该线程在停止之前一直工作运行。
2. void \*worker(void \*pool)：线程池内的线程的工作函数，该函数内是一个循环，一直在不停的执行取任务，执行任务这两个操作，当没有任务可以取的时候，就会阻塞等待。
3. Task get\_task(TaskQueue \*queue)：从线程池的任务队列中取任务函数，当没有任务可取时就会阻塞等待，当取到任务后，就可以释放一个not\_full的信号量，以通知add\_task函数可以继续往线程池中添加任务。
4. void add\_task(TaskQueue \*queue, void (\*function)(void \*), TaskArgs \*arg)： 往线程池中添加任务函数，如过任务队列满了，就阻塞等待not\_full信号量，当添加成功后，就释放not\_empty信号量，使得get\_task函数可以取到任务。
5. void destroy\_thread\_pool(ThreadPool \*pool)：销毁线程池，将stop标记为1，添加结束任务 到线程池，使用pthread\_join函数进行线程分离，最后销毁线程池中的信号量，释放掉线程池所申请的空间。

**工作流程**

1. 首先使用create\_thread\_pool方法创建一个线程池。
2. 用户通过 add\_task()方法向线程池提交任务，其中任务要执行的函数，所传入的参数，被封装成一个Task结构体传入。
3. 空闲的工作线程使用get\_task从队列中获取任务，执行任务的包装函数。
4. 线程池关闭时，所有线程被通知停止，并等待所有线程完成当前任务。

# 外部排序

由于要排序的文件可能高达1G，而要求使用不超过64MB的内存来实现，数据不可能一次全部载入内存，所以本作业使用外部排序来实现这一要求。

外部排序具体实现如下：

1. 先使用一个list\_files函数来对要求排序的目录下的所有文件进行排序，list\_files函数能够遍历得到要求排序的目录下的所有文件的文件名和路径，每当得到一个路径后，就将这个文件作为排序任务添加进排序线程池。
2. external\_sort函数是排序的主函数，工作线程就会每次读取10000个数据，将其加载到内存中，使用快速排序将这10000个数据进行排序后，创建一个临时文件，将排序结果写入这个临时文件内。
3. 使用一个merge\_files函数，将所有的临时文件合并起来，合并时采用多路合并思路，同时打开所有的临时文件，一次读取文件内一个数据，每次选择最小的数据写入到一个大的临时文件中，直到所有的文件被读取完，这个大的临时文件就是源文件已排好顺序的文件，这个大文件的名字以0开头，以方便后续的归并操作。再将原来小临时文件删除掉。

# 归并所有已经排序好的文件

当外部排序的线程池结束后，所有文件的数据均已排序好，并存放在临时目录中，接下来，就再创建一个线程池，来将所有的已经排序好的临时文件，合并为一个新文件。每次从所有中间文件中选两个合并成下一级中间文件，等到这一级中间文件排序完成后再排序下一级，直到所有数据都被排序到一个文件中，用到的函数有：

1. int count\_prefix(int prefix, const char \*dir\_path)

用于统计在给定目录下以特定前缀开头的文件数量，可以得到某一级中间文件的个数，如果个数为 1，证明得到最终结果。

1. int merge\_two\_file(char \*file1, char \*file2, char \*output)用来将两个已排序的文件按照顺序合并成一个新文件。
2. char merge\_orderd\_files(int prefix, const char \*dir\_path)归并的主函数，来合并以特定前缀开头的已排序文件到以下一个前缀开头的多个文件中，将合并任务添加到线程池中。

具体步骤如下：

1. 首先创建一个合并文件的线程池，用于合并众多文件，并生产一个新文件
2. 循环遍历临时目录下的所有文件，使用一个遍历i来判断当前得到的是基数个文件，还是偶数个文件。
3. 当是奇数个文件时，就记录这个文件的文件名和文件路径，当是偶数个文件时，记录下这个文件的文件名后，将记录的两个文件封装好，做为一个任务添加到线程池中，线程池中每个任务执行merge\_two\_file函数
4. 当一个任务执行完毕后，生产一个新文件，这个新文件的文件名，就是两个合并文件的文件头数字加一，以此来标记是第几轮合并的文件。
5. 在merge\_orderd\_files递归调用merge\_orderd\_files函数，直到count\_prefix返回值为1，说明所有的文件均已合并完成。

## **4. 复制最终结果到文件里**

这一步要在临时目录下找到最终文件，并把它的内容复制到用于指定的输出文件中。用到的函数有：

void copy\_prefix\_file(const char \*path, int prefix, const char \*dst\_name)

作用是从指定目录中复制以特定前缀开头的文件的内容到另一个文件中。这里的特定开头由归并文件操作中的merge\_two\_file函数得到。

**主要步骤：**

1. 通过传入的特定前缀开头计算得到最后这个文件的文件名
2. 打开这个文件，并打开最终要写入的文件
3. 使用while循环，不停的使用fgets读取最终临时文件的内容，然后将读到的东西写入最终文件。
4. 关闭目录和目标文件。

执行完后，完成所有数据排序。

# 四、测试

## 生成测试数据

使用generate.cpp 在 data文件夹下生成一定规模的测试数据。

生成测试数据的程序 generate.cpp 如下：

#include <iostream>

#include <fstream>

#include <string>

#include <cstdlib>

#include <ctime>

#include <filesystem>

namespace fs = std::filesystem;

*// 生成随机数据文件（文本格式）*void generate\_random\_files(const std::string &dir, size\_t file\_count, size\_t max\_file\_size) {

*// 创建目标目录*

fs::create\_directory(dir);

*// 设置随机数种子*

std::srand(static\_cast<unsigned>(std::time(nullptr)));

for (size\_t i = 0; i < file\_count; ++i) {

*// 生成文件名*

std::string filename = dir + "/file" + std::to\_string(i) + ".txt";

*// 打开文件输出流*

std::ofstream outfile(filename);

if (!outfile) {

std::cerr << "Failed to create file: " << filename << std::endl;

continue;

}

*// 随机生成文件大小（每个文件包含的数据量）*

size\_t file\_size = rand() % max\_file\_size + 1; *// 随机文件大小 1 ~ max\_file\_size*

size\_t num\_elements = file\_size / sizeof(int64\_t);

*// 随机生成数据并写入文件（以空格分隔）*

for (size\_t j = 0; j < num\_elements; ++j) {

int64\_t num = rand() % 1000000 - 500000; *// 随机生成 -500,000 ~ 499,999 的数*

outfile << num;

if (j != num\_elements - 1) {

outfile << " "; *// 数据之间用空格分隔*

}

}

outfile.close();

std::cout << "Generated file: " << filename << " (" << num\_elements << " numbers)" << std::endl;

}

}

int main() {

*// 目标目录*

std::string target\_dir = "data";

*// 配置文件数量和最大文件大小*

size\_t file\_count = 1000; *// 文件数量*

size\_t max\_file\_size = 1024 \* 1024\*1024; *// 每个文件最大 1MB*

*// 调用函数生成随机文件*

try {

generate\_random\_files(target\_dir, file\_count, max\_file\_size);

std::cout << "All files generated in directory: " << target\_dir << std::endl;

} catch (const std::exception &e) {

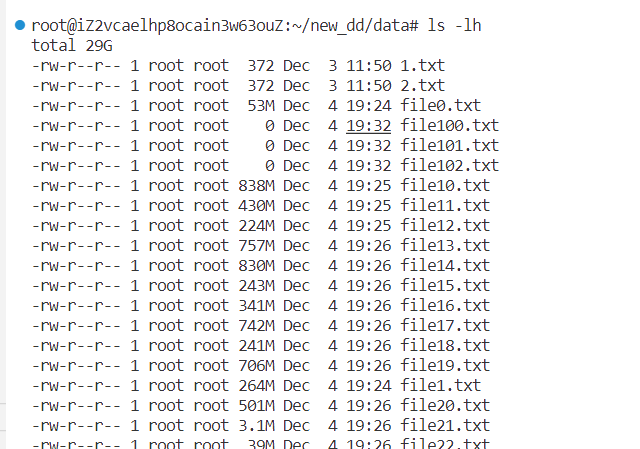
std::cerr << "Error: " << e.what() << std::endl;

}

return 0;

}

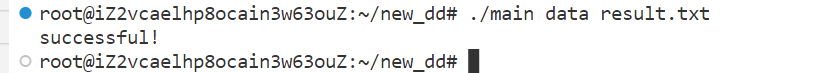
使用./generate在data目录生成数据，如下图所示，生产了30个G的数据：



## 编译源码

g++ -g -pthread \*.cpp -o main

## 执行：./main data result.txt

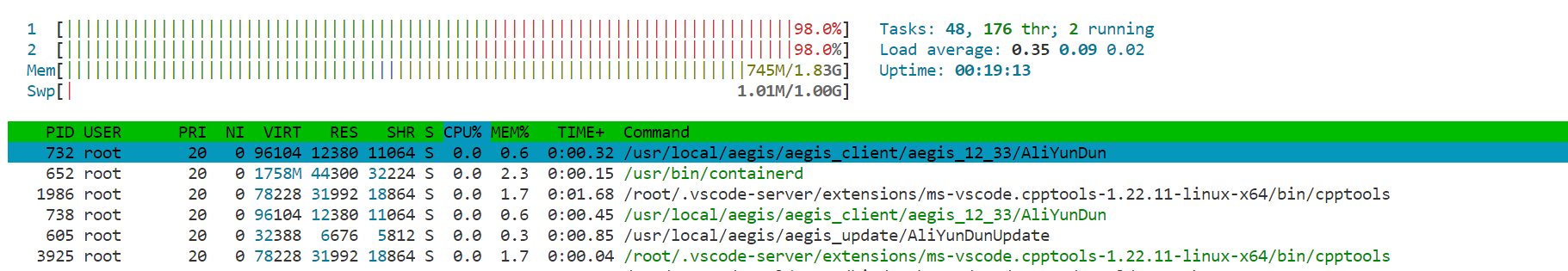


最终结果

result.txt 文件内容：



如图所示，所有要排序的文件均已成功排序，且因为使用了多线程的原因，速度很快，在短时间就能实现。



如图所示，cpu均已跑满，系统态与内核态各占一半，程序运行时，不会出现死锁、忙等等情况。说明成功实现实验要求。