《并行程序设计》
实验报告

学号：201936380086

姓名：陈涵

班级：2019 级软件工程 1 班

任课教师：汤德佑

2022/5/11

一. 实验目标

1. 熟悉掌握 pthread 库的用法
2. 理解线程池的概念，以及常用的线程调度方法

二. 实验内容

1. 编写一个 pthreads 程序实现一个“任务队列”，主线程启动用户指定数量的线程，这些线程会在因为等待某个条件而立即睡眠。主线程还生成由其他线程执行的任务块；每次它生成一个新的任务块，就会用一个条件信号唤醒一个线程，当一个线程完成任务块的执行时，它又会回到条件等待。当主线程完成了所有的生成任务后，它会设置某个全局变量，指示再也没有更多的任务生成了，并用一个条件广播唤醒所有线程。
2. 使用 pthreads 库函数实现分组算法的数据并行处理

三. 实验结果分析

1. 程序关键代码

```
//fast read and process
char* buffer;
int64_t buffer_size;
int64_t buffer_pos;
vector<int64_t> split_point;
vector<pthread_mutex_t> split_point_mutex;
int thread_num;
vector<vector<string> > split_result;

void init(int init_thread_num)
{
	thread_num = init_thread_num;
	buffer_pos = 0;
	split_point.resize(thread_num+1);
	split_point[0] = 0;
	split_point_mutex.resize(thread_num+1);
	for(int i=0;i<thread_num+1;i++)
	{
		pthread_mutex_init(&split_point_mutex[i], NULL);
```

```
        }
        split_result.resize(thread_num);
}

void* one_thread_preprocess(void* arg)
{
        int thread_id = *(int*)arg;
        cout<<"thread "+to_string(thread_id)+" start\n";
        pthread_mutex_lock(&split_point_mutex[thread_id]);
        uint64_t start = getTime();
        int64_t start_pos = split_point[thread_id];
        int64_t end_pos = split_point[thread_id+1];
        string tmp;
        for(int64_t i=start_pos;i<end_pos;i++)
        {
                if(buffer[i]=='\n')
                {
                        split_result[thread_id].push_back(tmp);
                        tmp.clear();
                }
                else
                {
                        tmp += buffer[i];
                }
        }
        pthread_mutex_unlock(&split_point_mutex[thread_id]);
        uint64_t end = getTime();
        cout<<"thread "+to_string(thread_id)+"\t preprocess time:
                "+to_string(end-start)+" ms\n";
        return NULL;
}

void read(const char* filename)
{
        uint64_t start = getTime();
        std::ifstream in(filename, std::ios::binary);
        if (!in)
        {
                std::cout << "open file failed" << std::endl;
                return;
        }
        buffer_size = in.seekg(0, std::ios::end).tellg();
        buffer = (char*)malloc(buffer_size+1);
        buffer[buffer_size] = '\n';
```

```
    int64_t each = buffer_size/thread_num;
    for (int i = 1; i < thread_num; i++)
    {
        split_point[i] = split_point[i-1] + each;
    }
    split_point[thread_num] = buffer_size;
    for(int i = 0; i <= thread_num; i++)
    {
        pthread_mutex_lock(&split_point_mutex[i]);
    }
    pthread_t thread_id[thread_num];
    int thread_id_arg[thread_num];
    for(int i=0;i<thread_num;i++)
    {
        thread_id_arg[i] = i;
        pthread_create(&thread_id[i], NULL, one_thread_preprocess,
                &thread_id_arg[i]);
    }
    for(int i = 0; i < thread_num; i++)
    {
        in.seekg(split_point[i], std::ios::beg).read(buffer+split_point[i],
                split_point[i+1]-split_point[i]);
        while(buffer[split_point[i+1]-1] != '\n')
        {
            split_point[i+1]--;
        }
        pthread_mutex_unlock(&split_point_mutex[i]);
    }
    in.close();
    uint64_t end = getTime();
    std::cout << "read file cost: " << end-start << "ms" << std::endl;
    for(int i = 0; i < thread_num; i++)
    {
        pthread_join(thread_id[i], NULL);
    }
    //delete[] buffer;
}
```

（1）线程池中共定义了多少个线程，每个线程的任务是什么，分别是如何执行的？

答：定义了 257 个线程，一个读线程作为生产者，把数据从磁盘中分块读入内存，256 个预处理线程作为消费者，每当读线程读完一块文件块，预处理线程就将该块的字符整理为 string 类存入相应 vector 中。

（2）本次实验中是如何定义池的（采用了什么样的数据结构类），池的作用是什么，对池的操作有哪些，分别如何实现的？

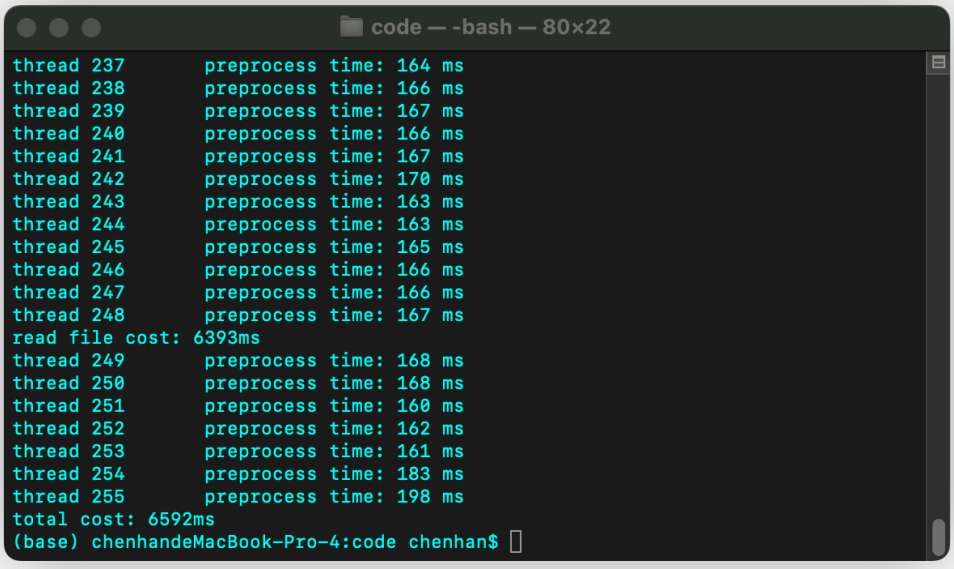
答：使用数组定义的线程池，对线程池有 create 和 join 两种操作。

2. 性能分析

测试环境	
CPU	Intel 9880H 8-core 2.3GHz
Memory	DDR4-2400MHz Dual-Channel 32GB
compiler	Apple Clang-1300.0.29.30
disk	APPLE SSD AP2048M-2TB@PCIe 3.0 x4

测试数据		
文件名	文件大小	字符串行数
80M_low.txt	7.32G	80000000

于使用了线程池操作，文件读 IO 和数据处理的计算负载并行，在读入的同时也在进行处理操作，所以总耗时约为读入 IO 时长+单个线程处理数据时长。



3. 总结

在利用 ptrhead 创建线程时，注意到传参为指针，所以要预先开好参数数组，而

不是使用循环变量进行传参，否则会发生临界资源冲突。
由于边读边处理的操作，靠后线程休眠时间较长，所以可以开远大于设备物理线程数的线程池来保证低空载。