《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

## 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

### 实验结果分析

#### 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。

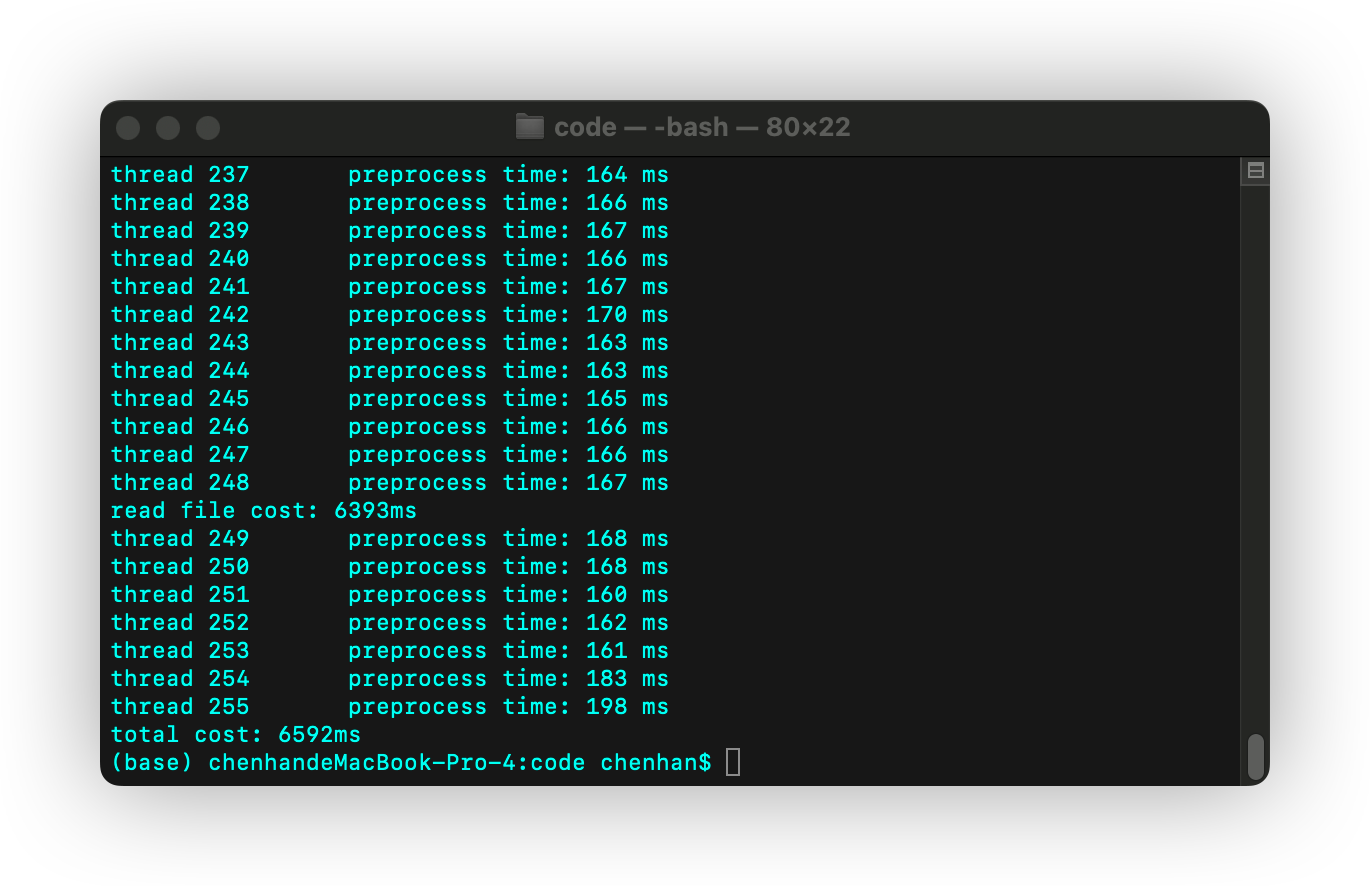
（2）本次实验中是如何定义池的（采用了什么样的数据结构类），池的作用是什么，对池的操作有哪些，分别如何实现的？

答：使用数组定义的线程池，对线程池有create和join两种操作。

#### 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。

#### 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。