

# Map Reduce 实验报告

## 一、问题描述

本次实验主要内容为实现Map功能、Reduce功能，以及Map、Reduce功能的衔接。

Map功能，要求将命令行传递给程序的若干个文件，分为指定组，每一组由一个线程负责读取和处理。每个线程需要对每个文件，执行用户编写的map函数。分组的时候应当尽可能提高处理性能。

用户编写map函数中，会调用MR\_Emit方法，来提供文件的处理结果。因此在MR\_Emit方法中，需要将用户传入的（key,value）键值对收集起来，备Reduce功能使用，从而实现Map和Reduce两部分的衔接。

Reduce功能，要求首先将所有的（key,value）按照用户指定的Partitioner函数分组，每一组由一个线程负责处理。注意，含有相同key的键值对，必须分在同一组，并且每一组在处理前要先按照key进行升序排序。然后对每一个不同的key，调用用户编写的reduce函数。

reduce函数中不会一次性接收当前key所有value的列表，而是在函数体中调用get\_func一次获取一个值。也就是说，get\_func函数便利当前key所有value的列表，并在到达列表尾部时返回NULL。

## 二、解决方案

### 1. 定义数据结构

由上述分析可知，本实验中需要处理key和value之间的映射，并且这些映射的数量不固定，而这种数据是数组难以实现的。因此需要事先定义一些数据结构，以便于存储数据。

贯穿整个实验的数据结构，主要有以下三个：

#### kv\_pair

该数据结构用于存储key和value，是三者中最简单的数据结构：

```
typedef struct kv_pair {
    char* key, * value;
} kv_pair;
```

#### array\_list

相较于C++中数组在定义时必须规定大小，该数据结构可以动态添加元素，为动态数组。

```
typedef struct array_list {
    void** array;
    size_t length, __volume;
    size_t iter_pos;
} array_list;
```

具有以下四个属性：

- array：负责数据的存储和随机读写。
- length：数组的元素个数。
- iter\_pos：用于get\_func记录当前迭代位置。
- \_\_volume：私有属性，记录当前数组容量。

以及以下三个操作：

```
array_list init_list();
void push_item(array_list* list, void* item);
void destroy_list(array_list* list);
```

- `init_list`: 初始化动态数组
- `push_item`: 在数组尾部添加元素
- `destroy_list`: 销毁数组，回收内存空间

## multi\_map

该数据结构为允许key重复的哈希表，可用于保存kv\_pair实体，并且可按照key快速找到相应的实体。

```
typedef unsigned long (*hash_func_t)(char* key, int hash_table_size);

typedef struct multi_map {
    array_list* lists;
    size_t table_size;
    hash_func_t hash_func;
} multi_map;
```

- `lists`: 哈希表。哈希值作为下标可查到该哈希值对应的实体列表，实体列表为动态数组。
- `table_size`: 哈希值上限，同时也是lists数组的容量。
- `hash_func`: 哈希函数。

具有以下操作：

```
multi_map init_map(hash_func_t hash_func, int hash_table_size);
void put_item(multi_map* map, char* key, char* value);
void destroy_map(multi_map* map);
```

- `init_map`: 初始化哈希表，需要指定哈希函数和哈希表容量。
- `put_item`: 向哈希表添加 (key,value) 实体。
- `destroy_map`: 销毁哈希表，回收内存空间。

## 2. 实现Map功能

Map功能由\_\_exec\_map函数负责执行。该函数首先需要对文件进行分组，考虑到一般来说，Map对每个数据单元进行操作基本相同，时间花费基本相似，因此这里尽可能使得在文件分好组后，每组文件总体积基本相近。由于Map功能的执行总时间取决于花费时间最多的文件组，因此这一策略有助于加快Map执行速度。

分组部分由以下几步构成：

- 获取命令行中给定文件的大小，计算出所有文件总体积：

```

file_stat* stats = malloc(sizeof(file_stat) * file_count);
size_t total_size = 0;

for (int i = 0; i < file_count; i++) {
    stats[i].path = files[i];
    if (stat(files[i], &stats[i].stat)) {
        fprintf(stderr, "cannot open file %s\n", files[i]);
        exit(1);
    }
    total_size += stats[i].stat.st_size;
}

qsort(stats, file_count, sizeof(file_stat), __compare_file_size);

```

其中，数据结构file\_stat为：

```

typedef struct stat stat_t;
typedef struct file_stat {
    char* path;
    stat_t stat;
} file_stat;

```

- 对每个组创建一个动态数组，用于存放该组含有的文件：

```

array_list* file_lists = malloc(num_mappers * sizeof(array_list));
for (int i = 0; i < num_mappers; i++)
    file_lists[i] = init_list();

```

- 升序遍历所有的文件，并将文件一个一个纳入到当前动态数组中，直到当前动态数组所盛放文件的大小已经超过应有的平均大小。

```

int allocated = 0;
for (int i = 0; i < num_mappers; i++) {
    size_t accum_size = 0;
    while (accum_size < total_size / num_mappers && allocated < file_count)
    {
        push_item(file_lists + i, stats[allocated].path);
        accum_size += stats[allocated].stat.st_size;
        allocated++;
    }
}

```

将文件分好组后，创建相应线程并执行：

```

pthread_t* tids = malloc(sizeof(pthread_t) * num_mappers);
for (int i = 0; i < num_mappers; i++)
    pthread_create(&tids[i], NULL, __map_thread_runner, file_lists + i);
for (int i = 0; i < num_mappers; i++)
    pthread_join(tids[i], NULL);

```

当线程全部执行完毕后，将无用的数据销毁：

```

for (int i = 0; i < num_mappers; i++)
    destroy_list(file_lists + i);
free(file_lists);
free(stats);

```

线程内的执行逻辑由\_\_map\_thread\_runner函数负责，该函数接收一个参数，即盛放文件的动态数组指针。逻辑比较简单，只需要对每个文件执行即可：

```

void* __map_thread_runner(void* param) {
    array_list* file_list = (array_list*) param;
    for (size_t i = 0; i < file_list->length; i++)
        (*mapper_func)(((char**)file_list->array)[i]);
}

```

### 3. 实现MR\_Emit功能

MR\_Emit函数比较简单，只需要将用户传递的 (key,value) 实体保存到哈希表中即可：

```

void MR_Emit(char *key, char *value) {
    sem_wait(&emit_sem);
    put_item(&pair_map, strdup(key), strdup(value));
    sem_post(&emit_sem);
}

```

但是注意，为了防止冲突，需要加锁，如上所示。

### 4. 实现Reduce功能

题目要求，在Reduce之前，键值对实体需要按照用户指定的Partitioner函数分组，每一组交给一个线程。在这里，直接让Partitioner作为哈希函数，用户指定的线程数作为哈希上限。如此，哈希表中的每一个动态数组，即自动对应一个线程。这一部分属于初始化，在MR\_Run里实现：

```

void MR_Run(int argc, char *argv[],
            Mapper map, int num_mappers,
            Reducer reduce, int num_reducers,
            Partitioner partition) {

    pair_map = init_map(partition, num_reducers);
    // ...
}

```

Reduce的主体功能由\_\_exec\_reduce负责，由于实际上事先已经分好组，因此这里创建线程并执行即可：

```

void __exec_reduce(Reducer reduce, int num_reducers) {
    reducer_func = reduce;
    pthread_t* tids = malloc(sizeof(pthread_t) * num_reducers);
    for (size_t i = 0; i < num_reducers; i++)
        pthread_create(tids + i, NULL, __reduce_thread_runner, (void *)i);
    for (size_t i = 0; i < num_reducers; i++)
        pthread_join(tids[i], NULL);
}

```

\_\_reduce\_thread\_runner负责线程内逻辑，它接收一个参数，为该线程的编号，线程通过该参数获得动态数组，并执行reduce操作，代码如下：

```

size_t index = (size_t) param;
array_list* list = &pair_map.lists[index];
qsort(list->array, list->length, sizeof(void*), __compare_pair);
for (size_t i = 0; i < list->length; i++) {
    if (!i || strcmp(((kv_pair*)list->array[i - 1])>key, ((kv_pair*)list->array[i])>key))
        reducer_func(((kv_pair*)list->array[i])>key, __get_next_value,
index);
}

```

这里，首先按照题目要求，对键值对实体按照key进行升序排序。升序排序后，相同key的实体便自动出现在一起，因此只需遍历一遍动态数组，若当前实体和上一次实体的key不同，则执行以下reducer\_func。

## 5. 实现get\_func

该函数为用户的reduce函数提供下一个键值对实体。在动态数组的iter\_pos属性中，已经记录了即将要提供的实体的位置。因此只需要将该位置对应的实体返回，然后将iter\_pos加一即可。注意这里需要判断一下，当前key已遍历完，当且仅当iter\_pos对应的实体的key与参数不符，或者当前已遍历完整个动态数组，此时返回NULL即可。

```

char* __get_next_value(char *key, int partition_number) {
    array_list* list = &pair_map.lists[partition_number];
    if (list->iter_pos ≥ list->length)
        return NULL;
    kv_pair* current = (kv_pair *)list->array[list->iter_pos];
    if (strcmp(current->key, key))
        return NULL;
    list->iter_pos++;
    return current->value;
}

```

## 6. 实现MR\_Run

MR\_Run负责初始化、调用Map和Reduce功能，以及最后的垃圾回收工作。具体代码如下：

```

void MR_Run(int argc, char *argv[],
    Mapper map, int num_mappers,
    Reducer reduce, int num_reducers,
    Partitioner partition) {

    pair_map = init_map(partition, num_reducers);
    sem_init(&emit_sem, 0, 1);

    __exec_map(map, num_mappers, argc - 1, argv + 1);
    __exec_reduce(reduce, num_reducers);

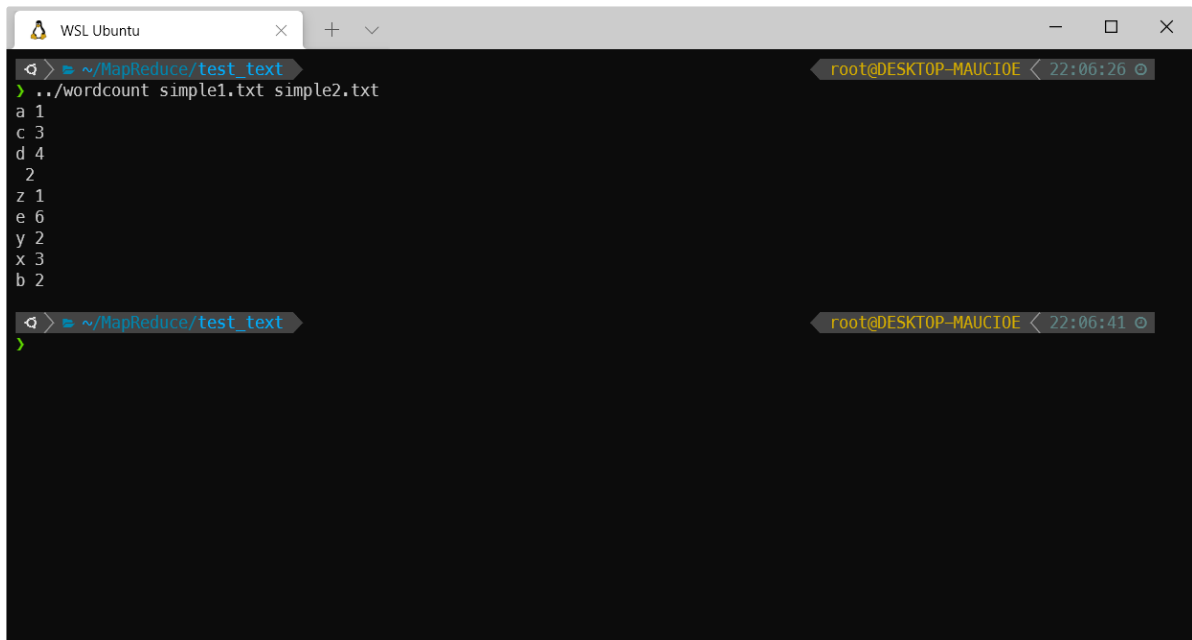
    sem_destroy(&emit_sem);
    destroy_map(&pair_map);
}

```

## 三、程序演示

在test\_text下有一定的测试数据。

其中simple1.txt和simple2.txt存储少量文本，将其输入到程序中：



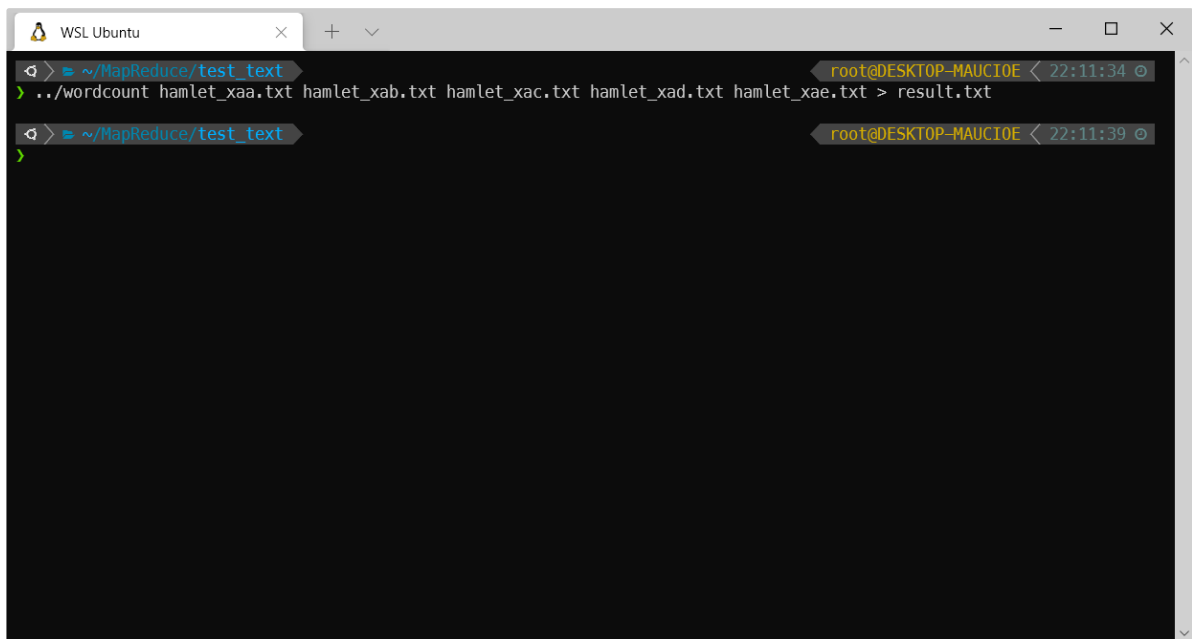
A terminal window titled 'WSL Ubuntu' showing the execution of the 'wordcount' program. The user runs the command `../wordcount simple1.txt simple2.txt` in the directory `~/MapReduce/test_text`. The output lists the frequency of words: 'a' (1), 'c' (3), 'd' (4), '2' (2), 'z' (1), 'e' (6), 'y' (2), 'x' (3), and 'b' (2). The terminal also shows the prompt `root@DESKTOP-MAUCIOE` and the time `22:06:26`.

```
WSL Ubuntu
~/MapReduce/test_text
> ../wordcount simple1.txt simple2.txt
a 1
c 3
d 4
2 2
z 1
e 6
y 2
x 3
b 2

~/MapReduce/test_text
>
```

手动统计simple1.txt和simple2.txt中的单词，结果与程序输出相同。（程序统计了两个空字符，这是由于wordcount.c没有处理分割后的空字符，因此属于正常现象）

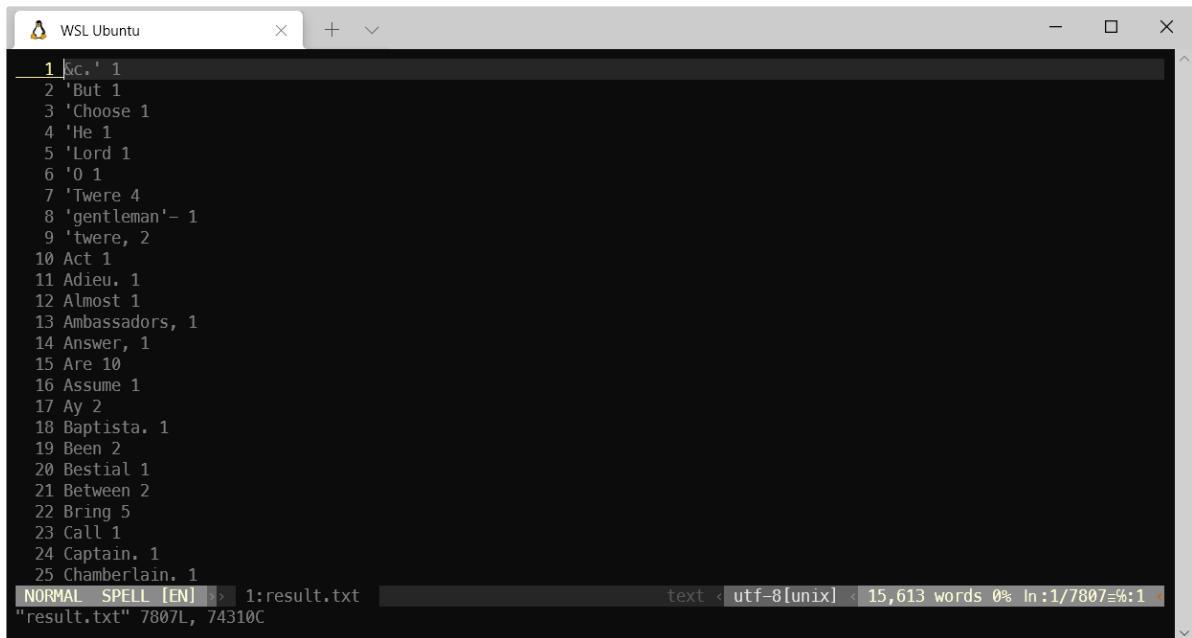
在hamlet\_xa\*.txt五个文件中存放了《哈姆雷特》英文全文，将其输入到程序中：



A terminal window titled 'WSL Ubuntu' showing the execution of the 'wordcount' program. The user runs the command `../wordcount hamlet_xaa.txt hamlet_xab.txt hamlet_xac.txt hamlet_xad.txt hamlet_xae.txt > result.txt` in the directory `~/MapReduce/test_text`. The terminal also shows the prompt `root@DESKTOP-MAUCIOE` and the time `22:11:34`.

```
WSL Ubuntu
~/MapReduce/test_text
> ../wordcount hamlet_xaa.txt hamlet_xab.txt hamlet_xac.txt hamlet_xad.txt hamlet_xae.txt > result.txt

~/MapReduce/test_text
>
```



```
1 'sc.' 1
2 'But 1
3 'Choose 1
4 'He 1
5 'Lord 1
6 'O 1
7 'Twere 4
8 'gentleman'- 1
9 'twere, 2
10 Act 1
11 Adieu. 1
12 Almost 1
13 Ambassadors, 1
14 Answer, 1
15 Are 10
16 Assume 1
17 Ay 2
18 Baptista. 1
19 Been 2
20 Bestial 1
21 Between 2
22 Bring 5
23 Call 1
24 Captain. 1
25 Chamberlain. 1
NORMAL SPELL [EN] 1:result.txt text < utf-8[unix] < 15,613 words 0% ln:1/7807=1
"result.txt" 7807L, 74310C
```

程序没有发生运行时错误，并且在result.txt中也得到了结果，说明程序能够应对较大的文本量。

## 四、实验总结

本次实验的主要收获是，练习了使用POSIX相关函数创建线程的方法，以及使用semaphore来创建互斥锁。同时相应地，在实验的编码过程中，使用了大量指针操作，充分练习了C语言编程能力。