2023학년도 데이터베이스시스템 텀프로젝트

맵리듀스 프레임워크
개발 및 결과 보고서

5조

2018136005 고재민
2018136008 김경건
2018136035 박동진
2018136045 박형진

차 례

Ⅰ. 서론

1. 팀 구성원 및 업무 분담

저희 2분반 5조는 4명의 팀원으로 이루어져 있으며 각 팀원은 아래와 같은 역할을 담당하였습니다.

이름	역할
김경건	프레임워크 설계, 발표자료 작성 및 발표
고재민	프레임워크 설계, 발표자료 작성
박동진	프레임워크 설계 및 구현, 보고서 작성
박형진	프레임워크 설계 및 구현, 보고서 작성

2. 구현물 동작 방법

ⅰ) 컴파일 방법

제출한 파일은 MapReduce.cpp 파일과 wordcount.cpp 파일 두 개의 파일입니다.

컴파일 순서
1. 두 개의 파일을 같은 디렉토리에 위치시키기
2. **g++ -pthread -g -o wordcount wordcount.cpp** 입력
3. **wordcount 테스트용텍스트파일** 입력 ex) ./wordcount part.tbl
4. result 파일 확인 ex) vi result

result 파일에 성공적으로 단어의 개수가 출력되는 것을 확인할 수 있습니다.
혹시나 너무 오래걸리신다면 MAP_BLOCK_SIZE를 1000000으로 0하나 더 붙여서 다시 컴파일 진행하시면 빠르게 진행되지만 메인메모리의 용량(리눅스에서의 용량)이 부족하다면 컴파일이 안될 수도 있습니다. MAP_BLOCK_SIZE는 MapReduce.cpp 파일의 21번째 줄에 #define으로 정의되어 있습니다.

ii) 컴파일 주의사항

```
157     static void map_function(struct RECORD records[], FILE* split_file, FILE* map_file, struct KeyValue maps[], int i) {
158         int m_size = 0;
159         size_t x = 0;
160
161         char delimiters[15] = " \t\n,.;:!?|"; // 여러 구분자 사용 가능
162
163         while (1) {
```

MapReduce.cpp 파일의 161번 째 줄의 delimiters를 통해 개행문자를 조절할 수 있습니다. 예를 들어 공백과 줄바꿈 문자로만 단어를 구분하고 싶다면 char delimiters[15] = “ \n”; 로 바꾸어 주면 됩니다. 개행문자에 따라 결과가 다르게 나올 수 있기 때문에 사용자가 바꾸어주어야 제대로 된 결과가 출력됩니다.

Ⅱ. 본론

1. 구현 상세

i) 프레임워크 설명

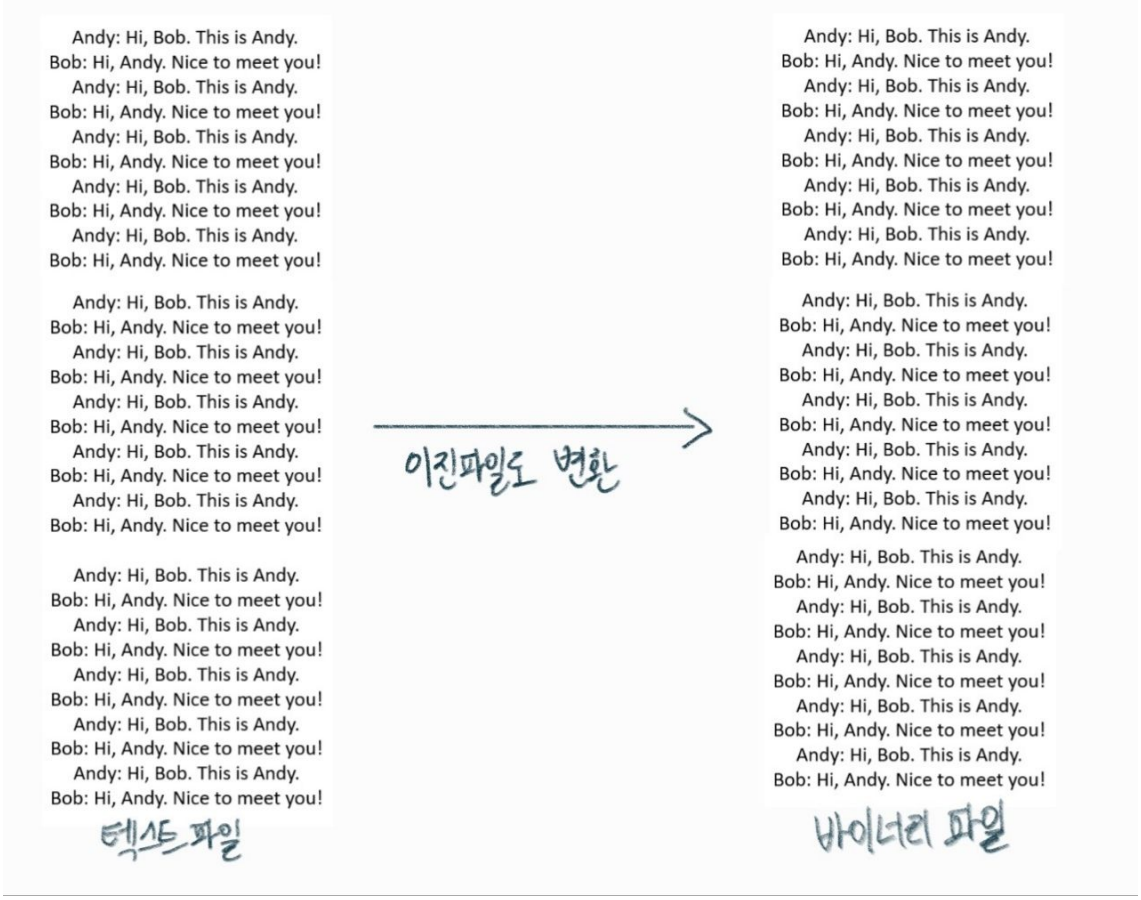
```
630     void run() {
631         clock_t time_1, time_2;         // 시간을 체크하기 위한 변수
632
633         time_1 = clock();
634
635         to_binary();        // 데이터를 이진파일로 변환
636
637         split();            // 문자열들 3개의 파일에 분배
638
639         Map();              // map() 호출
640
641         partition();        // partition() 호출
642
643         external_sort();    // sort() 호출
644
645         reduce();           // reduce() 호출
646
647         word_count_print();
648
649         time_2 = clock();
650
651         getrusage(RUSAGE_SELF, &r_usage);   // 메모리 출력하기 위해 구조체를 받아오는 함수
652
653         printf("실행 시간 : %f 초₩n", ((double)(time_2 - time_1)) / CLOCKS_PER_SEC);
654         printf("메모리 사용량: %ld KB₩n", r_usage.ru_maxrss);
655     }
```

run 함수를 살펴보면 프로그램의 시간을 체크하기 위해 time_1 변수에 clock을 걸어주며 시작합니다. 그리고 tbl 파일을 이진파일로 변환하는 함수인 to_binary()함수, 단어들이 모여있는 각각의 문자열들을 3개의 파일에 분배하는 split() 함수, 분리된 각각의 파일을 단어와 숫자의 개수로 나타내는 Map()함수, 각각의 맵파일들을 읽으며

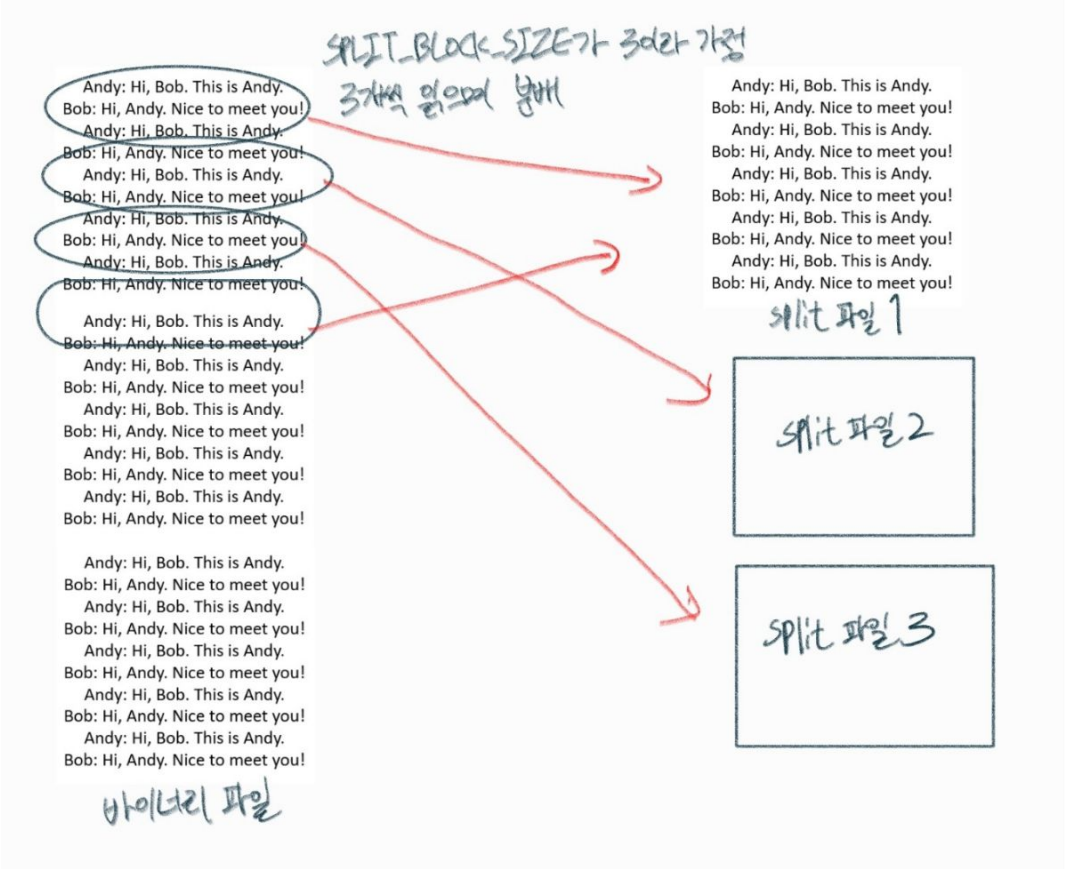
파티션을 진행하는 partition() 함수, 외부 정렬을 진행하는 external_sort() 함수, 정렬된 단어들을 읽으며 합쳐주는 reduce() 함수, 각 단어들과 숫자를 최종적으로 하나의 파일에 써주는 word_count_print() 함수로 이루어져 있습니다.

각 함수들의 코드를 모두 나타내기보다는 파일을 업로드했고, 소스코드 전문을 업로드 했기 때문에 각 함수의 해당 줄 번호를 보고서에 작성하고 알고리즘을 설명하도록 하겠습니다.

1. to_binary() : MapReduce.cpp 79번 줄

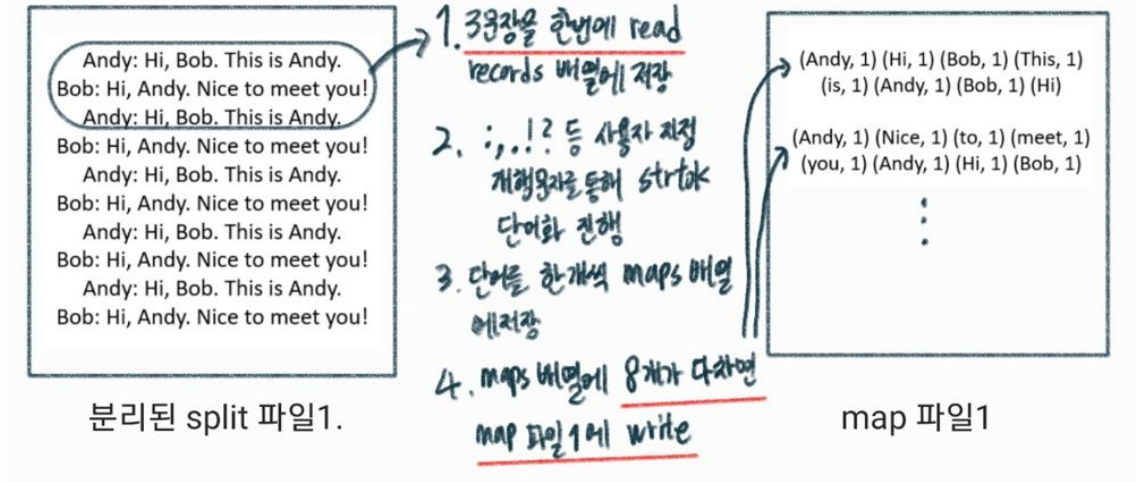


2. split() : MapReduce.cpp 99번 줄

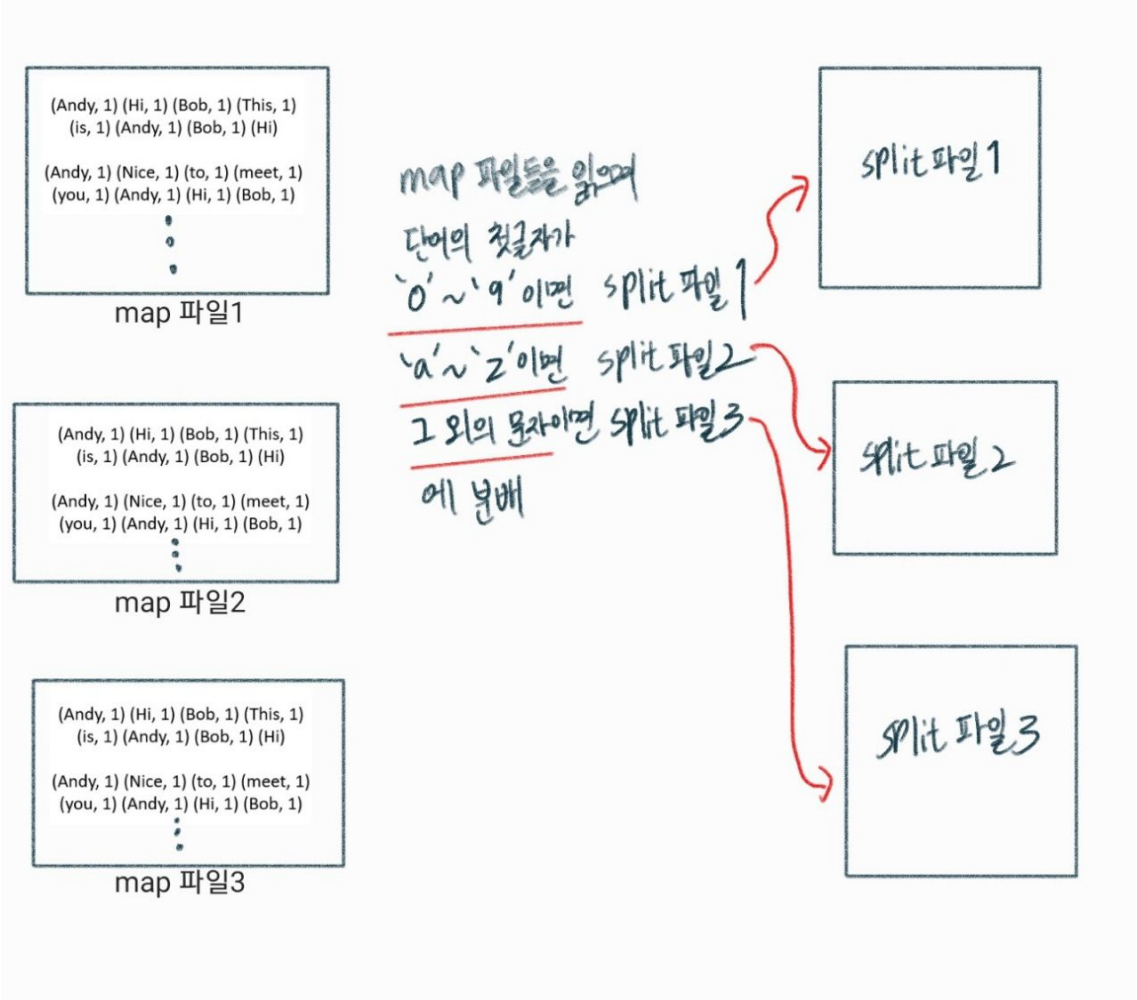


3. Map() : MapReduce.cpp 229번 줄 (세부 map_function() : 157번 줄)

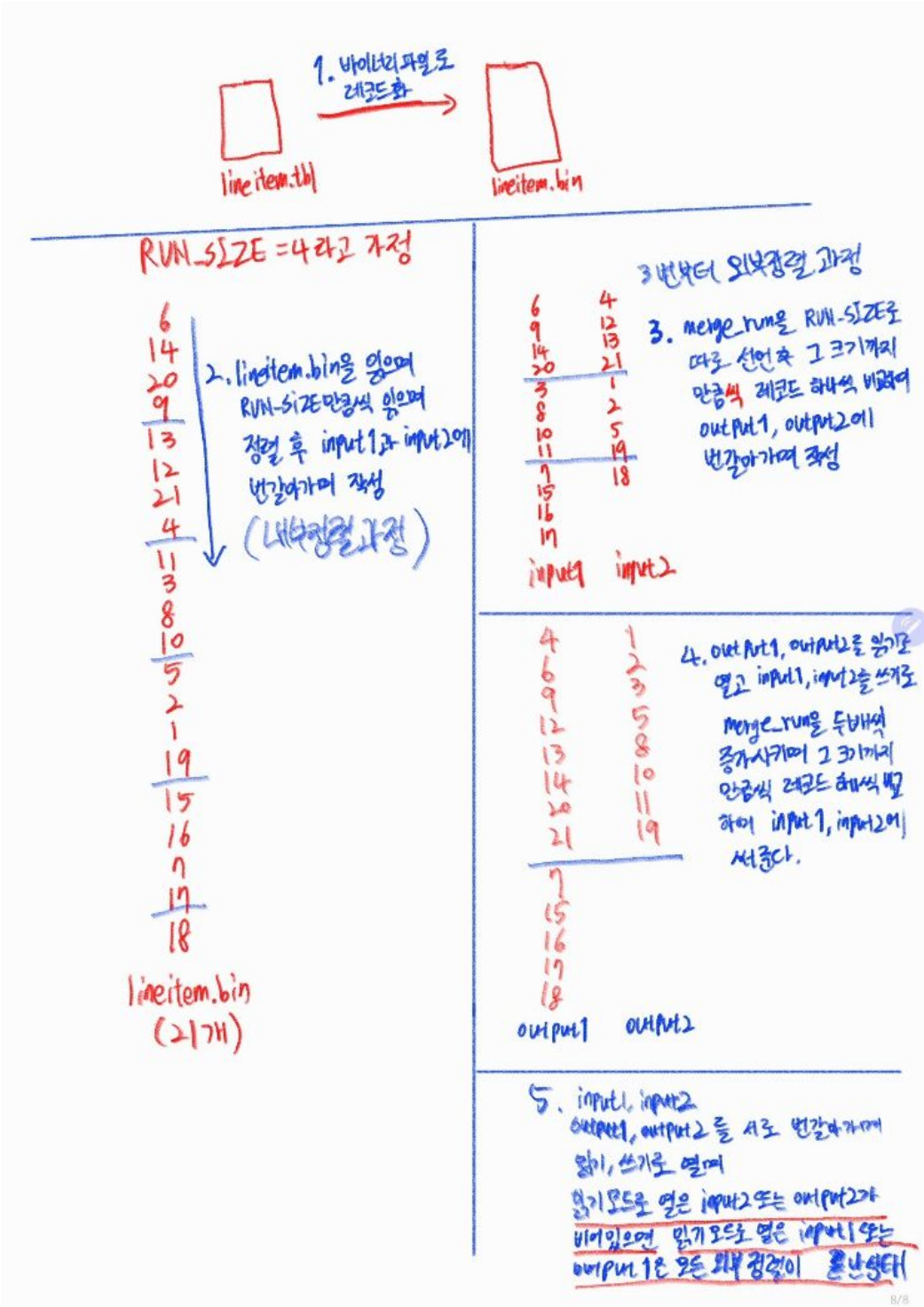
- SPLIT_BLOCK_SIZE가 3이라 가정
- MAP_BLOCK_SIZE가 8이라 가정



4. partition() : MapReduce.cpp 302번 줄 (세부 partition_function() : 272번 줄)

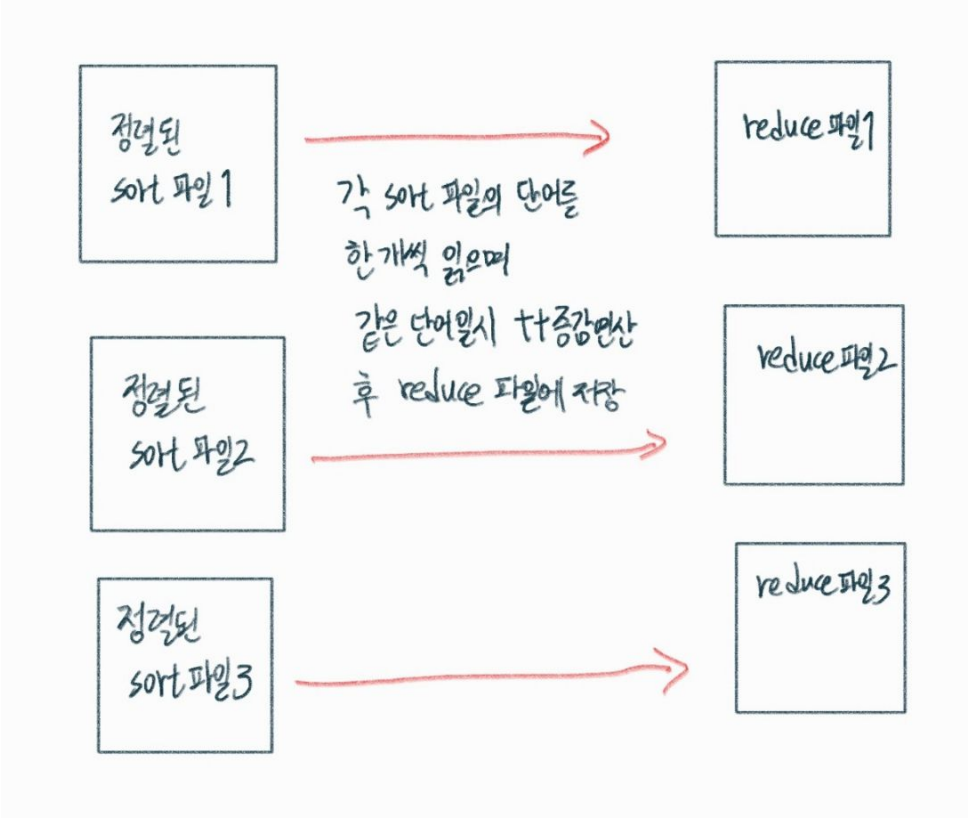


5. external_sort() : MapReduce.cpp 470번 줄 (세부 sort_function(): 334번 줄)

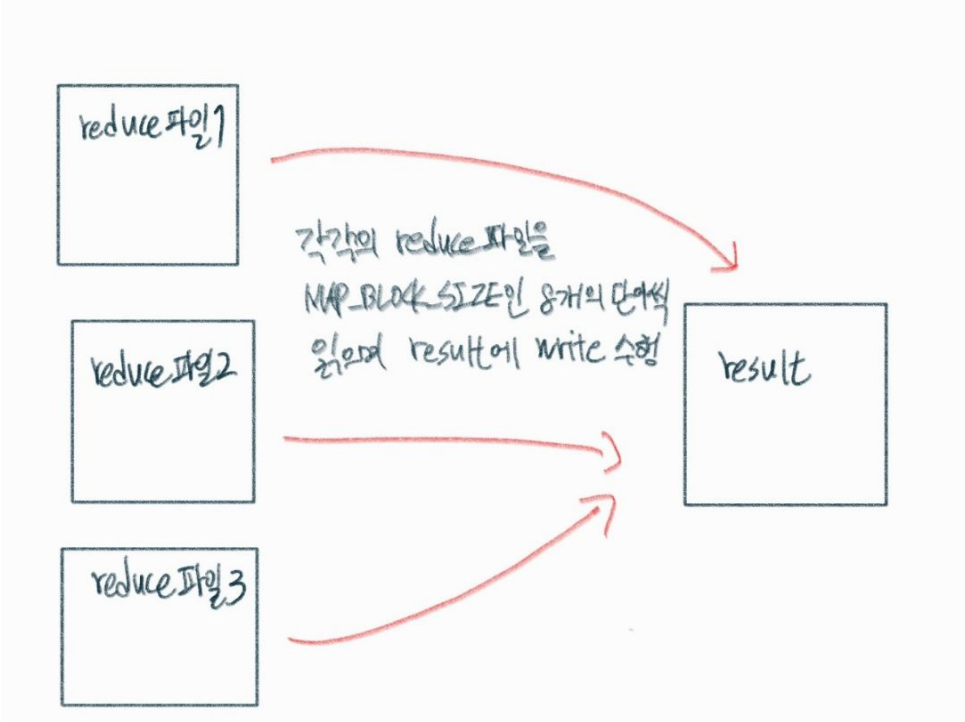


위의 RUN_SIZE가 MAP_BLOCK_SIZE라고 생각하시면 됩니다. 위에서는 4개라고 가정했지만 지금까지 설명드린 알고리즘으로 생각하면 8개라고 생각하시면 되고 따라서 3개의 split 파일들을 MAP_BLOCK_SIZE인 8개의 단어까지 내부정렬 후 외부정렬을 시켜주는 과정입니다.

6. reduce() : MapReduce.cpp 519번 줄 (세부 reduce_function(): 494번 줄)



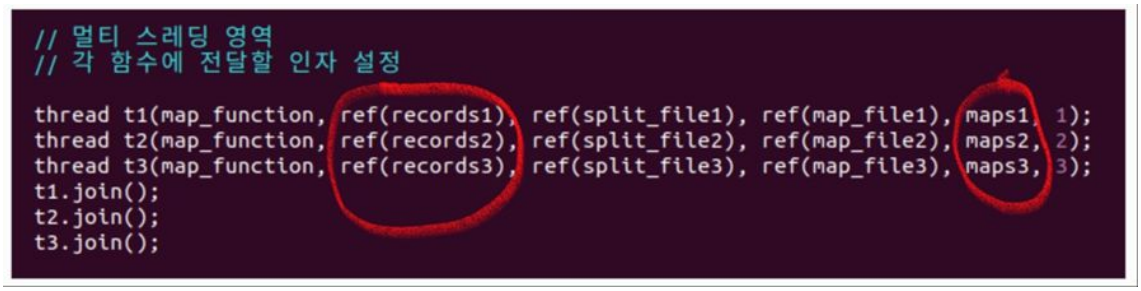
7. word_count_print() : MapReduce.cpp 562번 줄
(세부 word_count_print_function(): 494번 줄)



2. 최적화 상세

ⅰ) 병렬처리 구조

병렬처리 구조를 구현하며 겪었던 문제를 설명드리겠습니다. 반복문이 많은 맵과 리듀스 인터페이스에서 mutex를 통해 동기화 진행 시 많은 오버헤드가 발생하였습니다. 따라서 저희는 Reentrant 구조에 따라 애초에 인자로 넘겨줄 records배열과 maps배열을 스레드 개수만큼 생성하여 자원을 공유하지 못하도록 설계하였습니다.



```
// 멀티 스레딩 영역
// 각 함수에 전달할 인자 설정

thread t1(map_function, ref(records1), ref(split_file1), ref(map_file1), maps1, 1);
thread t2(map_function, ref(records2), ref(split_file2), ref(map_file2), maps2, 2);
thread t3(map_function, ref(records3), ref(split_file3), ref(map_file3), maps3, 3);
t1.join();
t2.join();
t3.join();
```

이렇게 진행해서 공유하는 자원이 없도록 설계하였는데도 불구하고 단어의 개수가 정확히 출력되지 않음을 확인했습니다. 문제점을 생각하며 해결하고자 노력하였고 Map()함수 내부의 strtok 함수에서 문제점을 발견하였습니다. 공유하는 자원이 없는 것이 아니었고 아래의 strtok내부함수에서 static으로 선언되어 있는 변수를 확인할 수 있었습니다.

```
char* strtok(char* src, const char* delim)
{
  // src, delim 이 NULL 인지, delim 이 "" 인지 체크하는 코드는 생략
  char* tok;
  static char* next;     // 분석을 시작할 위치
  if (src != NULL)
    next = src;
  tok = next;

  // boundary condition check
  if (*next == '\0')
    return NULL;
```

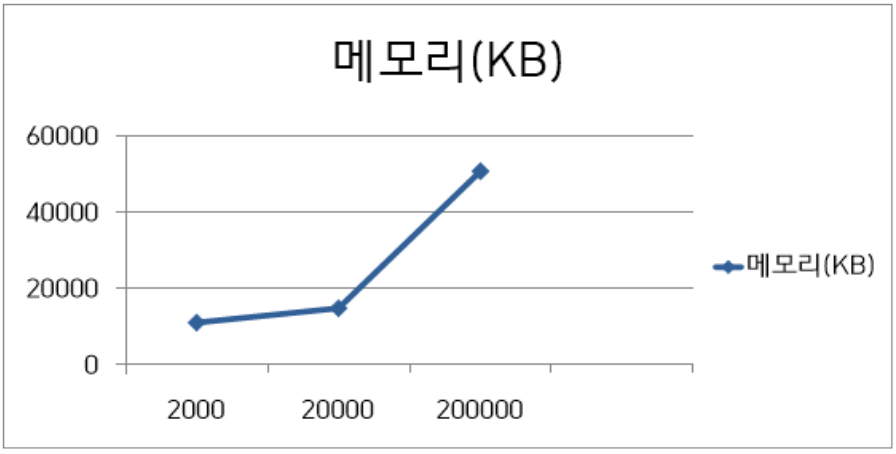
위 문제는 strtok_r 함수를 사용하여 해결하였습니다.
strtok_r 함수는 3번 째 인자로 분석을 시작할 주소를 받고 시작하는 함수이기 때문입니다.

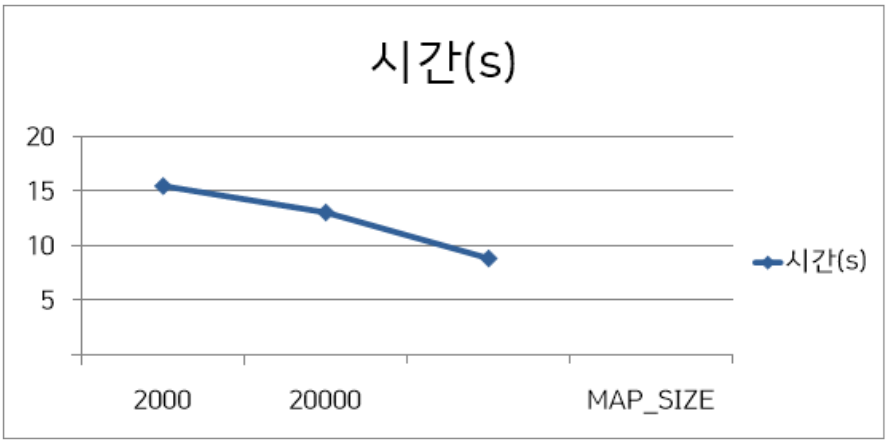
```
// 첫 번째 토큰 분리
token = strtok_r(records[i].str, delimiters, &context);
```

하지만 맵과 리듀스에서 IO 연산이 워낙 많고 컨텍스트 스위칭이 자주 일어나 병렬처리시 오히려 성능이 떨어지는 경우가 있음을 확인하여 결국 큰 성능 차이가 없음을 확인할 수 있었습니다.

ii) 메모리와 시간에 따른 상관관계

SPLIT_BLOCK_SIZE는 10000으로 고정 input_file은 대략 48MB 데이터 파일입니다.
MAP_BLOCK_SIZE의 크기에 따른 메모리 사용량과 시간 비교 측정





MAP_SIZE가 커질수록 구조체 배열의 크기가 증가하기 때문에 메모리의 크기는 증가하지만 외부정렬의 파일입출력은 줄어들기 때문에 시간은 감소하는 것을 확인할 수 있습니다. **차트에서 MAP_SIZE란 MAP_BLOCK_SIZE를 의미합니다.**

Ⅲ. 결론

1. 고찰

이번 프로젝트를 진행하면서 개발 실력 및 메모리 및 시간의 최적화에 대한 부분을 경험하며 실력향상에 많은 도움이 되었습니다. 하지만 아직은 효율적으로 코드를 설계하고 깔끔한 코드를 짜는 능력은 부족하다고 생각이 들었고 어떻게 하면 더욱 간결하고 알아보기 쉬운 코드로 짤 수 있을지에 대해 생각하며 이 부분에서의 실력을 더욱 쌓고 싶다는 생각이 들었습니다.

Ⅳ. 부록

1. 참고 문헌

"DatabaseSystem_Practice_04_프로젝트.pdf" (한국기술교육대학교 전강욱 교수 제작)

“https://blog.naver.com/lswsean54/222759252620” (네이버 블로그 맵리듀스 설명)

2. 소스코드 전문

wordcount.cpp

```
#include "MapReduce.cpp"

int main(int argc, char** argv) {

	if(argc != 2) {
		printf("follow the execution format [./wordcount file_name]");
		return 0;
	}

	MapReduce wordcount(argv[1]);
	wordcount.run();
```

```
        return 0;
}
```

MapReduce.cpp

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <unordered_map>
#include <cstring>
#include <algorithm>
#include <time.h>
#include <sys/resource.h>
#include <fcntl.h>
#include <unistd.h>
#include <vector>
#include <algorithm>
#include <map>
#include <thread>
#include <mutex>
#include <string>
using namespace std;
#define SPLIT_BLOCK_SIZE 10000
#define MAP_BLOCK_SIZE 100000
#define MEMORY_SIZE 1800000
    struct RECORD {
        char str[256];
    } record;
    struct KeyValue {
        char key[64];
        int value;
        bool operator<(const KeyValue& a) const {
            return strcmp(key, a.key) < 0;
        }
    } keyvalue;
        struct RECORD records1[SPLIT_BLOCK_SIZE];
        struct RECORD records2[SPLIT_BLOCK_SIZE];
        struct RECORD records3[SPLIT_BLOCK_SIZE];
        struct KeyValue maps1[MAP_BLOCK_SIZE];
        struct KeyValue maps2[MAP_BLOCK_SIZE];
        struct KeyValue maps3[MAP_BLOCK_SIZE];
class MapReduce{
private:
    FILE* lineitem_file;
    FILE* lineitem_bin_file;
```

```
    FILE* split_file1;
    FILE* split_file2;
    FILE* split_file3;
    FILE* map_file1;
    FILE* map_file2;
    FILE* map_file3;
    FILE* partition_file1;
    FILE* partition_file2;
    FILE* partition_file3;
    FILE* input1_file;
    FILE* input2_file;
    FILE* output1_file;
    FILE* output2_file;
    FILE* tmp_file;
    FILE* sort_file1;
    FILE* sort_file2;
    FILE* sort_file3;
    FILE* reduce_file1;
    FILE* reduce_file2;
    FILE* reduce_file3;
    FILE* result_file;
    double cpu_time_used;
    struct rusage r_usage;
    static mutex file_mutex;
    string input_file;
    void to_binary() {
        char buffer[256];
        lineitem_file = fopen(input_file.c_str(), "r");                 // open lineitem.tbl
        if (lineitem_file == NULL) {
            printf("your_input_file open error\n");
            exit(0);
        }
        lineitem_bin_file = fopen("work", "wb");          // open lineitem.bin
        while (fgets(buffer, sizeof(buffer), lineitem_file)) {                    //  interval  one
line
            strcpy(record.str, buffer);
            if (record.str[0] == '\n')                                  // 공백 줄 예외 처리
                continue;
            fwrite(&record, sizeof(struct RECORD), 1, lineitem_bin_file);
        }
        fclose(lineitem_file);                                                  // close file
        fclose(lineitem_bin_file);
    }
```

```
    void split() {
        lineitem_bin_file = fopen("work", "rb");        // 닫아주었던 변환 완료한 이진파일을 읽기모드로 연다.
        if (lineitem_bin_file == NULL) {
            printf("lineitem.bin open error\n");
            exit(0);
        }
        split_file1 = fopen("work1", "wb");                    // 문장 블록 크기 (SPLIT_BLOCK_SIZE) 만큼씩 분배하기 위한 파일
        split_file2 = fopen("work2", "wb");
        split_file3 = fopen("work3", "wb");
        /* split 과정 */
        while (!feof(lineitem_bin_file)) {
            size_t x = 0;
            if ((x = fread(&records1[0], sizeof(struct RECORD), SPLIT_BLOCK_SIZE, lineitem_bin_file)) < SPLIT_BLOCK_SIZE) {
                if (x == 0)
                    break;
                fwrite(records1, sizeof(struct RECORD), x, split_file1);
                break;
            }
            else {
                // 초반에 x는 계속 RUN_SIZE 보다 클 것
                fwrite(records1, sizeof(struct RECORD), SPLIT_BLOCK_SIZE, split_file1);
            }
            if ((x = fread(&records2[0], sizeof(struct RECORD), SPLIT_BLOCK_SIZE, lineitem_bin_file)) < SPLIT_BLOCK_SIZE) {
                if (x == 0)
                    break;
                fwrite(records2, sizeof(struct RECORD), x, split_file2);
                break;
            }
            else {
                // 초반에 x는 계속 RUN_SIZE 보다 클 것
                fwrite(records2, sizeof(struct RECORD), SPLIT_BLOCK_SIZE, split_file2);
            }
            if ((x = fread(&records3[0], sizeof(struct RECORD), SPLIT_BLOCK_SIZE, lineitem_bin_file)) < SPLIT_BLOCK_SIZE) {
                if (x == 0)
                    break;
                fwrite(records3, sizeof(struct RECORD), x, split_file3);
                break;
            }
```

```
            else {
                // 초반에 x는 계속 RUN_SIZE 보다 클 것
                fwrite(records3, sizeof(struct RECORD), SPLIT_BLOCK_SIZE, split_file3);
            }
        }
        fclose(lineitem_bin_file);
        unlink("work");
        fclose(split_file1);
        fclose(split_file2);
        fclose(split_file3);
    }
    static void map_function(struct RECORD records[], FILE* split_file, FILE* map_file,
struct KeyValue maps[], int i) {
        int m_size = 0;
        size_t x = 0;
        char delimiters[15] = " \t\n,.;:!?|"; // 여러 구분자 사용 가능
        while (1) {
            if ((x = fread(&records[0], sizeof(struct RECORD), SPLIT_BLOCK_SIZE,
split_file)) < SPLIT_BLOCK_SIZE) {
                if (x == 0)
                    break;
                for (int i = 0; i < x; i++) {
                    char* token;
                     char *context;
                    KeyValue key_value;
                    // 첫 번째 토큰 분리
                    token = strtok_r(records[i].str, delimiters, &context);
                    // 토큰이 NULL이 될 때까지 반복
                    while (token != NULL) {
                        m_size++;
                        strcpy(maps[m_size - 1].key, token);
                        maps[m_size - 1].value = 1;
                        // 다음 토큰 분리
                        token = strtok_r(NULL, delimiters, &context);
                        if (m_size == MAP_BLOCK_SIZE) {
                            fwrite(maps, sizeof(KeyValue), m_size, map_file);
                            m_size = 0;
                        }
                    }
                }
                break;
            }
            else {
```

```
                for (int i = 0; i < SPLIT_BLOCK_SIZE; i++) {
                    char* token;
                     char* context;
                    KeyValue key_value;
                    // 첫 번째 토큰 분리
                    token = strtok_r(records[i].str, delimiters, &context);
                    // 토큰이 NULL이 될 때까지 반복
                    while (token != NULL) {
                        m_size++;
                        strcpy(maps[m_size - 1].key, token);
                        maps[m_size - 1].value = 1;
                        // 다음 토큰 분리

                        token = strtok_r(NULL, delimiters, &context);
                        if (m_size == MAP_BLOCK_SIZE) {
                            fwrite(maps, sizeof(KeyValue), m_size, map_file);
                            m_size = 0;
                        }
                    }
                }
            }
            if (feof(split_file)) {
                break;
             }
        }
        if (m_size) {
            fwrite(maps, sizeof(KeyValue), m_size, map_file);
        }
    }
    void Map() {
        split_file1 = fopen("work1", "rb");
        if (split_file1 == NULL) {
            printf("split_file1 open error\n");
            exit(0);
        }
        split_file2 = fopen("work2", "rb");
        if (split_file2 == NULL) {
            printf("split_file2 open error\n");
            exit(0);
        }
        split_file3 = fopen("work3", "rb");
        if (split_file3 == NULL) {
            printf("split_file3 open error\n");
```

```
            exit(0);
        }
        map_file1 = fopen("work4", "wb");
        map_file2 = fopen("work5", "wb");
        map_file3 = fopen("work6", "wb");
        // 멀티 스레딩 영역
        // 각 함수에 전달할 인자 설정
/*
        thread t1(map_function, ref(records1), ref(split_file1), ref(map_file1), maps1, 1);
        thread t2(map_function, ref(records2), ref(split_file2), ref(map_file2), maps2, 2);
        thread t3(map_function, ref(records3), ref(split_file3), ref(map_file3), maps3, 3);
        t1.join();
        t2.join();
        t3.join();
*/
         map_function(records1, split_file1, map_file1, maps1, 1);
         map_function(records2, split_file2, map_file2, maps2, 2);
         map_function(records3, split_file3, map_file3, maps3, 3);
        fclose(split_file1);
        fclose(split_file2);
        fclose(split_file3);
        fclose(map_file1);
        fclose(map_file2);
        fclose(map_file3);
    }
    void partition_function(struct KeyValue maps[], FILE* map_file, FILE* partition_file1,
FILE* partition_file2, FILE* partition_file3) {
        size_t x = 0;
        while (!feof(map_file)) {
            if ((x = fread(&maps[0], sizeof(struct KeyValue), MAP_BLOCK_SIZE,
map_file)) < MAP_BLOCK_SIZE) {
                if (x == 0)
                    break;
                for (int i = 0; i < x; i++) {
                    if (maps[i].key[0] >= '0' && maps[i].key[0] <= '9')
                        fwrite(&maps[i], sizeof(KeyValue), 1, partition_file1);
                    else if (maps[i].key[0] >= 'a' && maps[i].key[0] <= 'z')
                        fwrite(&maps[i], sizeof(KeyValue), 1, partition_file2);
                    else
                        fwrite(&maps[i], sizeof(KeyValue), 1, partition_file3);
                }
                break;
            }
```

```
            else {
                for (int i = 0; i < MAP_BLOCK_SIZE; i++) {
                    if (maps[i].key[0] >= '0' && maps[i].key[0] <= '9')
                        fwrite(&maps[i], sizeof(KeyValue), 1, partition_file1);
                    else if (maps[i].key[0] >= 'a' && maps[i].key[0] <= 'z')
                        fwrite(&maps[i], sizeof(KeyValue), 1, partition_file2);
                    else
                        fwrite(&maps[i], sizeof(KeyValue), 1, partition_file3);
                }
            }
        }
    }
    void partition() {
        map_file1 = fopen("work4", "rb");
        if (map_file1 == NULL) {
            printf("map_file1 open error\n");
            exit(0);
        }
        map_file2 = fopen("work5", "rb");
        if (map_file2 == NULL) {
            printf("map_file2 open error\n");
            exit(0);
        }
        map_file3 = fopen("work6", "rb");
        if (map_file3 == NULL) {
            printf("map_file3 open error\n");
            exit(0);
        }
        partition_file1 = fopen("work1", "wb");
        partition_file2 = fopen("work2", "wb");
        partition_file3 = fopen("work3", "wb");
        partition_function(maps1, map_file1, partition_file1, partition_file2, partition_file3);
        partition_function(maps2, map_file2, partition_file1, partition_file2, partition_file3);
        partition_function(maps3, map_file3, partition_file1, partition_file2, partition_file3);
        fclose(map_file1);
        fclose(map_file2);
        fclose(map_file3);
        fclose(partition_file1);
        fclose(partition_file2);
        fclose(partition_file3);
    }
    void sort_function(FILE* partition_file, int check_sort_number) {
        input1_file = fopen("input1", "wb");
```

```
        input2_file = fopen("input2", "wb");
        /* ------ 내부 정렬 진행 과정 ----------*/
        while (!feof(partition_file)) {
            size_t x = 0;                                   // x : 구조체 크기를
RUN_SIZE 만큼 읽으라고 했는데 RUN_SIZE보다 작다면
                                                            // 딱 그만큼만 정렬하고 입력파일에 써주
기 위한 변수
            if ((x = fread(maps1, sizeof(struct KeyValue), MAP_BLOCK_SIZE,
partition_file)) < MAP_BLOCK_SIZE) {
                if (x == 0)
                    break;
                sort(begin(maps1), begin(maps1) + x);
                fwrite(maps1, sizeof(struct KeyValue), x, input1_file);
                break;
            }
            else {
        // 초반에 x는 계속 RUN_SIZE 보다 클 것
                sort(begin(maps1), begin(maps1) + MAP_BLOCK_SIZE);
                fwrite(maps1, sizeof(maps1), 1, input1_file);
            }
            if ((x = fread(maps1, sizeof(struct KeyValue), MAP_BLOCK_SIZE,
partition_file)) < MAP_BLOCK_SIZE) {
                if (x == 0)
                    break;
                sort(begin(maps1), begin(maps1) + x);
                fwrite(maps1, sizeof(struct KeyValue), x, input2_file);
                break;
            }
            else {                                                  // 초반
에 x는 계속 RUN_SIZE 보다 클 것
                sort(begin(maps1), begin(maps1) + MAP_BLOCK_SIZE);
                fwrite(maps1, sizeof(maps1), 1, input2_file);
            }
            getrusage(RUSAGE_SELF, &r_usage);                       // 메모
리 넘어가는지 체크
            if (r_usage.ru_maxrss > MEMORY_SIZE) {
                printf("메모리 사용량: %ld KB\n", r_usage.ru_maxrss);
                printf("MAP BLOCK SIZE를 줄이거나 메모리 크기를 늘려서 선언해주세
요.\n");
                exit(0);                                            // 정해둔 메모
리 넘어가면 탈출
            }
        }
```

```
        fclose(partition_file);
        fclose(input1_file);
        fclose(input2_file);
        input1_file = fopen("input1", "rb");  // 여기부터 2원균형합병을 위한 파일포인터
들
        input2_file = fopen("input2", "rb");  // 서로 읽기파일과 쓰기파일을 번갈아가면
서 맡게 될 파일포인터들
        output1_file = fopen("output1", "wb");
        output2_file = fopen("output2", "wb");
        /* ------------- 외부정렬 진행 과정 ---------- */
        int merge_run = MAP_BLOCK_SIZE;          // 병합하면서 run크기는 2배씩 증
가할 예정이기 때문에 따로 정의해둔 상수를 변수로 설정해준다
        int file_index = 0;                      // file_index : input파일들과 output파일들
을 서로 읽기 쓰기 모드를 번갈아가며 하기 위한 체크변수
        while (fread(&keyvalue, sizeof(struct KeyValue), 1, input2_file) > 0) {
        // 입력파일2의 내용이 비어있지 않은동안 반복하는 알고리즘
            fseek(input2_file, 0, SEEK_SET);                                    // 위에
서 읽어서 확인했기 때문에 다시 0번 위치로 돌아간다.
            int index = 1;                       // 쓸 파일을 정해주기 위한 체크변수
            int idx1 = 0, idx2 = 0;              // 외부 정렬시 필요한 변수
            KeyValue a, b;
            fread(&a, sizeof(struct KeyValue), 1, input1_file);                 // 외부
정렬
            fread(&b, sizeof(struct KeyValue), 1, input2_file);
            while (1) {
                if (index % 2 == 1)              // index가 홀수 일때는 output1_file에 써주
기 위해
                    tmp_file = output1_file;
                else
                    tmp_file = output2_file;
                // 이 부분부터는 병합정렬의 알고리즘과 유사합니다.
                while (idx1 < merge_run && idx2 < merge_run && !feof(input1_file)
&& !feof(input2_file)) {
                    if (strcmp(a.key, b.key) < 0) {          // PARTKEY 기준으로 내
부정렬 진행했기 때문에 외부정렬도 똑같이 PARTKEY 기준
                        fwrite(&a, sizeof(struct KeyValue), 1, tmp_file);
                        fread(&a, sizeof(struct KeyValue), 1, input1_file);
                        idx1++;
                    }
                    else {
                        fwrite(&b, sizeof(struct KeyValue), 1, tmp_file);
                        fread(&b, sizeof(struct KeyValue), 1, input2_file);
                        idx2++;
```

```
                    }
                }
                // 하나의 idx가 끝나거나 파일이 끝날 시 남아있는 부분들 처리
                while (!feof(input1_file) && idx1 < merge_run) {
                    fwrite(&a, sizeof(struct KeyValue), 1, tmp_file);
                    fread(&a, sizeof(struct KeyValue), 1, input1_file);
                    idx1++;
                }
                while (!feof(input2_file) && idx2 < merge_run) {
                    fwrite(&b, sizeof(struct KeyValue), 1, tmp_file);
                    fread(&b, sizeof(struct KeyValue), 1, input2_file);
                    idx2++;
                }
                index++;            // index : tmp_file을 뭐로 정할지 결정해주기 위한
체크카운트 변수(쓸 파일을 정해준다)
                idx1 = 0, idx2 = 0;         // 블록 크기만큼의 외부정렬을 끝내면 다시
0으로하고 아래쪽의 블록 크기만큼의 외부정렬을 진행
                if (feof(input1_file) && feof(input2_file))        // 블록크기만큼씩의 외부
정렬을 모두 끝내면 종료
                    break;                                          // 종료 후 새로
운 input2_file을 읽으며 반복
            }
            merge_run *= 2;                                                 //
murge_run 크기는 두배씩 증가
            file_index++;
            if (file_index % 2) {               // file_index가 홀수면 output1을 읽을 파일
로 열고
                freopen("output1", "rb", input1_file);
                freopen("output2", "rb", input2_file);
                freopen("input1", "wb", output1_file);
                freopen("input2", "wb", output2_file);
            }
            else {                                      // file_index가 짝수면 input1을 읽
을 파일로 연다. 왜냐하면 홀수일때 input1을 쓰기로 열었기 때문
                freopen("input1", "rb", input1_file);
                freopen("input2", "rb", input2_file);
                freopen("output1", "wb", output1_file);
                freopen("output2", "wb", output2_file);
            }
        }                                               // 모두 끝나면 파일들 닫아준다.
        fclose(input1_file);
        fclose(input2_file);
        fclose(output1_file);
```

```
        fclose(output2_file);
        if (file_index % 2) {                    // 홀수면 output2에다가 쓴 상태
이고 이것을 읽기로 열어서 빈파일이기 때문에 output1에는 다 쓰여져 있다
            if (check_sort_number == 1)
                rename("output1", "sort1");
            else if (check_sort_number == 2)
                rename("output1", "sort2");
            else if (check_sort_number == 3)
                rename("output1", "sort3");
        }
        else {                                   // 짝수면 input2에다가 쓴 상태이
고 이것을 읽기로 열어서 빈파일이기 때문에 input1에는 다 쓰여져 있다.
            if (check_sort_number == 1)
                rename("input1", "sort1");
            else if (check_sort_number == 2)
                rename("input1", "sort2");
            else if (check_sort_number == 3)
                rename("input1", "sort3");
        }
    }
    void external_sort() {
        partition_file1 = fopen("work1", "rb");      // 닫아주었던 변환 완료한 이진파
일을 읽기모드로 연다.
        if (partition_file1 == NULL) {
            printf("partition_file1 open error\n");
            exit(0);
        }
        partition_file2 = fopen("work2", "rb");      // 닫아주었던 변환 완료한 이진파
일을 읽기모드로 연다.
        if (partition_file2 == NULL) {
            printf("partition_file2 open error\n");
            exit(0);
        }
        partition_file3 = fopen("work3", "rb");      // 닫아주었던 변환 완료한 이진파
일을 읽기모드로 연다.
        if (partition_file3 == NULL) {
            printf("partition_file3 open error\n");
            exit(0);
        }
        /* partition1 파일 정렬 */
        sort_function(partition_file1, 1);
        /* partition2 파일 정렬 */
        sort_function(partition_file2, 2);
```

```
        /* partition3 파일 정렬 */
        sort_function(partition_file3, 3);
    }
    static void reduce_function(struct KeyValue maps[], FILE* sort_file, FILE* reduce_file,
int i) {
        int cnt = 1;
        KeyValue pre;
        if (fread(&pre, sizeof(struct KeyValue), 1, sort_file) != 1)
            return;
        KeyValue curr;
        while (!feof(sort_file)) {
            if (fread(&curr, sizeof(struct KeyValue), 1, sort_file) != 1)
                break;
            if (strcmp(pre.key, curr.key) == 0)
                cnt++;
            else {
                pre.value = cnt;
                fwrite(&pre, sizeof(struct KeyValue), 1, reduce_file);
                cnt = 1;
            }
            pre = curr;
        }
        curr.value = cnt;
        fwrite(&curr, sizeof(struct KeyValue), 1, reduce_file);
    }
    void reduce() {
        sort_file1 = fopen("sort1", "rb");
        if (sort_file1 == NULL) {
            printf("sort_file1 open error\n");
            exit(0);
        }
        sort_file2 = fopen("sort2", "rb");
        if (sort_file2 == NULL) {
            printf("sort_file2 open error\n");
            exit(0);
        }
        sort_file3 = fopen("sort3", "rb");
        if (sort_file3 == NULL) {
            printf("sort_file3 open error\n");
            exit(0);
        }
        reduce_file1 = fopen("work4", "wb");
        reduce_file2 = fopen("work5", "wb");
```

```
        reduce_file3 = fopen("work6", "wb");
        /* parallel processing */
        /*
        thread t1(reduce_function, ref(maps1), ref(sort_file1), ref(reduce_file1), 1);
        thread t2(reduce_function, ref(maps2), ref(sort_file2), ref(reduce_file2), 2);
        thread t3(reduce_function, ref(maps3), ref(sort_file3), ref(reduce_file3), 3);
        t1.join();
        t2.join();
        t3.join();
*/
                /* single processing */
        reduce_function(maps1, sort_file1, reduce_file1, 1);
        reduce_function(maps2, sort_file2, reduce_file2, 2);
        reduce_function(maps3, sort_file3, reduce_file3, 3);
        fclose(sort_file1);
        fclose(sort_file2);
        fclose(sort_file3);
        fclose(reduce_file1);
        fclose(reduce_file2);
        fclose(reduce_file3);
    }
    void word_count_print_function(FILE* reduce_file, FILE* result_file) {
        while (!feof(reduce_file)) {
            size_t x = 0;                                   // x : 구조체 크기를
RUN_SIZE 만큼 읽으라고 했는데 RUN_SIZE보다 작다면
                                                    // 딱 그만큼만 정렬하고 입력파일에 써주
기 위한 변수
            if ((x = fread(maps1, sizeof(struct KeyValue), MAP_BLOCK_SIZE, reduce_file))
< MAP_BLOCK_SIZE) {
                if (x == 0)
                    break;
                for (int i = 0; i < x; i++) {
                    fprintf(result_file, "%s, %d\n", maps1[i].key, maps1[i].value);
                }
                break;
            }
            else {
        // 초반에 x는 계속 RUN_SIZE 보다 클 것
                for (int i = 0; i < MAP_BLOCK_SIZE; i++) {
                    fprintf(result_file, "%s, %d\n", maps1[i].key, maps1[i].value);
                }
            }
        }
```

```
    }
    void word_count_print() {
        reduce_file1 = fopen("work4", "rb");
        if (reduce_file1 == NULL) {
            printf("reduce_file1 open error\n");
            exit(0);
        }
        reduce_file2 = fopen("work5", "rb");
        if (reduce_file2 == NULL) {
            printf("reduce_file2 open error\n");
            exit(0);
        }
        reduce_file3 = fopen("work6", "rb");
        if (reduce_file3 == NULL) {
            printf("reduce_file3 open error\n");
            exit(0);
        }
         result_file = fopen("result", "w");
        word_count_print_function(reduce_file1, result_file);
        word_count_print_function(reduce_file2, result_file);
        word_count_print_function(reduce_file3, result_file);
        fclose(reduce_file1);
        fclose(reduce_file2);
        fclose(reduce_file3);
        unlink("work1");
        unlink("work2");
        unlink("work3");
        unlink("work4");
        unlink("work5");
        unlink("work6");
        unlink("sort1");
        unlink("sort2");
        unlink("sort3");
        unlink("input1");
        unlink("input2");
        unlink("output1");
        unlink("output2");
    }
public:
    MapReduce(string input_file) {
        this->input_file = input_file;
    }
    void run() {
```

```
        clock_t time_1, time_2;                         // 시간을 체크하기 위한 변수
        time_1 = clock();
        to_binary();                    // 데이터를 이진파일로 변환
        split();                        // 문자열들 3개의 파일에 분배
        Map();                                  // map() 호출
        partition();                    // partition() 호출
        external_sort();                                // sort() 호출
        reduce();                       // reduce() 호출
        word_count_print();
        time_2 = clock();
        getrusage(RUSAGE_SELF, &r_usage); // 메모리 출력하기 위해 구조체를 받아오는
함수
        printf("실행 시간 : %f 초\n", ((double)(time_2 - time_1)) / CLOCKS_PER_SEC);
        printf("메모리 사용량: %ld KB\n", r_usage.ru_maxrss);
    }
};
mutex MapReduce::file_mutex;
```