2024 NOVEMBER 30

3.2 차 과제- (구현) 섹터/블록 매핑 알고리즘

소프트웨어공학 (1)

심영준

202120204 심영준

강원대학교 삼척캠퍼스

3.2차 프로젝트 202120204 심영준

목차

3.2차 프로젝트 202120204 심영준

섹터/블록 매핑 알고리즘 비교분석 (총 2장이상)

1. 하드웨어적 스펙 분석

- 본인이 만든 섹터/블록 알고리즘의 여분 블록 수

- DRAM 요구량 (DRAM requirement) //본인의 DRAM용량 기입. 계산 공식 필요.

//본인의 섹터/블록 매핑 알고리즘의 비교를 위한 하드웨어적 스펙의 정당성 설명 필요

2. 성능 측면(storage performance) 분석

- 섹터/블록 알고리즘의 읽기/쓰기/지우기 연산의 횟수

//섹터/블록 매핑 알고리즘에 수행된 연산 read/write/erase 횟수 보여주기

- 섹터/블록 알고리즘의 전체 성능 시간

//전체 수행 시간 보여주기. Read=15μs, write= 200μs, erase = 2ms.

3. 성능 분석 및 결론

//성능 차이에 관한 원인분석

//알고리즘의 장단점 서술

섹터 매핑 알고리즘

1. 설계 (2장이상)

하드웨어의 기준은 3MB flash memory, 추가 블록 20개로 이루어짐

사용하는 DRAM은 논리주소 2B*6143 = 12286B, 섹터 정보 버퍼 529B 섹터 513B, 여분 섹터 17B 최소 사용 DRAM 13,345B(약 13.1KB) 이다

필수 구현 함수

Print_table ( ); FTL_read ( ); FTL_write ( );

섹터 매핑 알고리즘에서 읽기 기능의 구현은 LSN에 입력되어있는 PSN의 주소의 정보를 읽기만 하면 된다.

쓰기 기능은 기본적으로 LSN의 PSN 위치에 쓰기 연산을 실행한다.

섹터에는 512B만큼의 데이터를 저장하고, 여분 섹터에는 '1'이라는 글자를 입력한다.

하지만, PSN위치에 정보가 먼저 존재하는 경우에는 여분 블록에 순서대로 작성한다.

여분 블록은 10개의 블록으로 2개 구현하여 총 20개의 추가 블록으로 구성된다.

덮어쓰기 연산을 실행하면 추가블록의 첫번째부터 차례로 추가하여 LSN의 PSN주소를 추가된 섹터의 주소로 저장한다.

첫 번째 10개의 블록을 가득 채우면 두 번째 블록에 현재 할당 되어있는 주소(첫번째 블록)를 순서대로 작성한다. 후 첫번째 블록 10개를 지운 뒤 덮어쓰기 연산이 실행된 블록의 데이터를 첫번째 여분블록에 데이터에 옮겨 기존 정보를 저장한다.

이때 논리주소 전체를 순회하여 쓰기 연산이 여러 번 실행된 섹터가 존재하는 블록을 찾는 작업을 하여 여분블록에 저장할 블록을 선별한다.

기존 정보의 저장이 완료된다면 덮어쓸 블록을 삭제한 뒤 저장한 정보와 두번째 블록에 저장된 첫번째 블록의 정보를 논리주소와 물리주소를 1대1 대응하여 저장한다.

FTL_overwrite 에서

A	o
B	o
c	o

LSN 0 | PSN 0

LSN 1 | PSN 1

LSN 2 | PSN 2

LSN 3 | PSN 3

기존의 값이 존재하는 경우 여분 섹터(할당되어 있는)에 저장한다 (table의 PSN값도 해당 섹터로 할당한다.)

여기서 write 2 over 명령어를 입력하면

over	2

LSN 2 | PSN 6144 (할당 되어있는 여분섹터 주소)

여분 섹터에 할당되며 LSN값을 여분 섹터의 주소로 입력한다. (스페어 부분에는 기존 LSN의 주소를 입력한다)

이제 이 값을 저장하게 된다면(원본 테이블에 저장)

여분섹터 1

over	2

여분섹터 2

A	0
B	1

저장할 기존 섹터는 다른 여분 섹터에 저장한다. Table의 LSN값도 여분 섹터에 저장한다.

A	o
B	o
over	o

저장이 완료된다면 기존 섹터는 삭제하고 저장된 LSN의 값에 따라 LSN의 값이 변경되어 있다면 해당 값으로 다시 저장하는 연산을 한다.

저장이 완료 된다면 처음에 저장하던 여분 섹터를 초기화하고 원본 값을 저장하던 다른 여분 섹터에 값을 저장한다.

만약 write 0 over2 가 입력되면

여분 섹터 1

여분 섹터 2

A	0
B	1
over2	0

이런 식으로 저장기능을 수행할 때 마다 여분 섹터의 순서를 바꾸어 가며 연산을 한다.

저장기능을 위한 다른 여분 섹터가 가득 차는 것을 방지하기 위해 최대한 넣을 수 있는 블록을 저장한 뒤 다른 여분 섹터를 초기화한 후 그 위에 다시 저장기능을 실시한다.

이를 위해 저장의 단위는 1블록단위로 돌아가며 실시한다.

다음은 이를 구현하기 위한 DRAM의 테이블이다

LSN 0 | PSN 0
LSN 1 | PSN 1
LSN 2 | PSN 2
LSN 3 | PSN 3
LSN 4 | PSN 4
LSN 5 | PSN 5
LSN 6 | PSN 6
LSN 7 | PSN 7
LSN 8 | PSN 8

처음 프로그램을 실행하면 LSN와 PSN의 주소 값이 같도록 세팅된다.

각 LSN값은 unsigned short 값으로써 6143개의 배열을 가지고 있다.

2. 구현 (1장이상)

- 코드 설명

```
char buffer[MAX] = { 0, };  //한 섹터 정보를 저장
char sector[513] = { 0, };  //섹터의 저장 정보를 저장
char spare[17] = { 0, };   //섹터의 여분 섹터 정보를 저장
unsigned short index = 0;  //섹터주소
unsigned short table[SECTOR-1] = { 0, }; //TABLE의 섹터 개수 (2B * 6143)
unsigned short sparesector_index = 6144; //여분섹터주소

//지우기 쓰기 연산의 횟수를 적기위한 변수
counting* count = malloc(sizeof(count));
//0으로 초기화
count->read_count = 0;
count->write_count = 0;
count->erase_count = 0;
//시간을 측정하기 위한 변수
LARGE_INTEGER micro_start, micro_end, micro_timer;
double timer;
```

주로 사용되는 변수는 다음과 같다.

Buffer : 섹터값 하나를 저장한다.

Sector : 섹터 정보를 저장한다.

Spare : 섹터의 여분 정보를 저장한다.

Index : 주소값이다 주로 LSN 값을 입력받는다.

Table[SECTOR-1] : table 이다. LSN 의 PSN 값을 저장한다.

Sparesector_index : 덮어쓰기를 위한 여분 섹터의 주소이다.

Count: 연산할때 사용하는 FlashMemory 연산을 저장하는 구조체이다.

Micro_start, micro_end, micro_timer: 마이크로세컨을 측정하기 위한 변수

Timer: 연산 시간을 저장하는 변수

처음 프로그램을 시작하면 Flash Memory 의 init() 함수를 실행하여 파일의 무결성을 체크한다. 무결성이 확인된다면 마지막으로 실행했던 동작에서 만약 여분 섹터에 저장하고 있던 정보가 존재하는지 검사한다. 그 동작을 검사하는 함수는 void FTL_search_sapre_index() 함수이다.

```
Flash_read(SPARESECTORONE, str); //스페어 1 탐색
```

```
Flash_read(SPARESECTORTWO, str); //스페어 2 탐색
```

Flash_read 를 사용하여 각각의 여분섹터 첫번째가 비어 있는지 체크한다. 모든 저장 동작에서 한쪽은 비어있도록 구현했으므로 남아있는 정보가 있다면 바로 TABLE 의 PSN 값으로 할당해주면 된다. 여기서 동작을 'Y'를 입력 받아 복구작업을 실행한다면

```
for (unsigned short i = SPARESECTORTWO - 1; i >= SPARESECTORONE; i--) { //뒤에서 앞으로 이동
    Flash_read(i, str);
    count->read_count++;
    strncpy(spare, str + 512, 16);
    if (spare[0] == 'o');
    else if (spare[0] != ' ') {
        index = FTL_index_to_short(spare);
        if (table[index] < i) {
            table[index] = i; //테이블의 인덱스값을 스페어공간으로 이동
            if (*spare_index == 6144) {
                (*spare_index) = i + 1;
            }
        }
    }
```

다음과 같은 코드를 실행한다. 이 코드는 해당 스페어 섹터의 맨 뒤부터 불러오며 해당 섹터의 스페어 섹터 주소의 해당되는 값을 찾아 PSN 에 할당한다. 만약 table 의 PSN 값이 이미 할당되어 있다면 table 에 재할당 하지 않는다.

다른 섹터의 경우에도 똑같이 동작한다.

```
printf("TABLE           ...  매핑 테이블 출력(섹터)₩n");
printf("read LSN        ...  해당 LSN의 데이터 읽기₩n");
printf("write LSN DATA  ...  해당 LSN에 데이터 쓰기₩n");
printf("save            ...  여분 섹터 초기화₩n");
printf("exit            ...  프로그램 종료₩n₩n");
```

무결성 검사까지 끝났다면 프로그램의 첫 화면이 나온다 다음과 같은 선택지가 주어지며 해당 명령어를 입력하면 설명에 해당되는 동작을 실행한다.

TABLE 명령어를 입력한다면

`Print_table(table);` 이 FTL 함수를 실행하게 되는데 이 함수의 코드는 다음과 같다.

```
void Print_table(unsigned short* table) {
    for (int i = 0; i < SECTOR; i++) {
        printf("LSN %5hd  ||  PSN %5hd₩n", i, table[i]);
    }
}
```

TABLE 배열에 해당되는 값과 TABLE 배열의 주소 값을 출력한다.

Read LSN 을 입력하면

```
if (index >= SPARESECTORONE) {
    printf("초과된 주소입니다.₩n");
}
else {

    QueryPerformanceFrequency(&micro_timer);
    QueryPerformanceCounter(&micro_start);
    FTL_read(table[index], buffer, sector, spare, count);

    QueryPerformanceCounter(&micro_end);
    timer = (micro_end.QuadPart - micro_start.QuadPart) / (double)micro_timer.QuadPart;
    printf("FTL_read 걸린시간 : %lf ₩n", timer);


}
```

만약 index 값이 스페어 섹터 주소의 최소값(여기선 6144)보다 작다면 동작을 취소하고 아니라면 FTL_read 의 함수를 실행한다. 나머지 코드는 시간을 측정하는 코드이다.

FLT_read()함수

```
void FTL_read(unsigned short index, char* str, char* sector, char* spare, counting* count) {
    Flash_read(index, str);
    count->read_count++;

    strncpy(spare, str + 512, 16);
    if (spare[0] == ' ') {
        printf("해당 섹터의 값이 존재하지 않습니다.₩n");
    }
    else {
        strncpy(sector, str, 512);
        printf("sector : %s₩n", sector);
    }
}
```

다음은 FTL_read 의 함수이다. 받는 변수는 주소 값, main 의 buffer 포인터, main 의 sector 포인터, main 의 spare 포인터이다. Counting 은 FlashMemory 연산의 횟수를 측정하기 위한 구조체이다.

FTL_read 를 실행하면 받은 주소 값에 해당되는 table[LSN]값의 PSN 값을 읽어 (main 에서 Table[index]값으로 입력 받는다) 해당 섹터의 값을 출력한다. 만약 섹터의 정보가 없다면 해당 섹터의 값이 존재하지 않는다는 안내문을 출력한다.

Write LSN DAT 명령어를 입력하면

```
scanf("%hd", &index);
scanf("%s", sector);

QueryPerformanceFrequency(&micro_timer);
QueryPerformanceCounter(&micro_start);


FTL_wirte(table, index, buffer, sector, spare, &sparesector_index, count);
```

다음과 같은 동작으로 FTL_wirte 를 실행한다.

FTL_write() 함수

```
void FTL_wirte(unsigned short* table, unsigned short index, char* str, char* sector, char* spare, unsigned short* spare_index, counting* count)
```

void FTL_wirte(unsigned short* table, unsigned short index, char* str, char* sector, char* spare, unsigned short* spare_index, counting* count) 는 테이블 포인터, 주소값, buffer 포인터, sector 포인터, spare 포인터, 스페어섹터 주소의 포인터, FlashMemory 의 연산의 횟수를 측정하는 구조체를 받는다.

```
if (table[index] == index) {//LSN과 PSN이 같을때
    Flash_read(index, str);
    count->read_count++;
    strncpy(spare, str + 512, 16);
    if (spare[0] == 'o') { //값이 존재하는지
```

처음 값을 입력 받으면 일단 입력 받은 주소가 이미 추가 할당된(여러 번 연산되어 있는지)를 확인하여 만약 추가적인 연산이 되어있지 않은 섹터의 주소라면 해당 주소의 정보가 있는 지까지 확인한다.

```
printf("이미 존재하는 값 FTL_overwrite 실행₩n");
FTL_index_to_char(index, spare);
Flash_write(*spare_index, sector, spare);
count->write_count++;
table[index] = (*spare_index)++;
if (*spare_index == SPARESECTORTWO) {

    *spare_index = FTL_save_original_info(table, SPARESECTORTWO, str, sector, spare, count);
}
else if (*spare_index == ALLSECTOR) {

    *spare_index = FTL_save_original_info(table, SPARESECTORONE, str, sector, spare, count);

}
```

만약 값이 존재한다면 스페어 섹터의 주소 값인 spare_index 주소에 차례로 입력한다. 이때 섹터의 스페어 부분에는 원본 주소 값을 입력한다. 여기서 만약 spare_index 가 최대로 찬다면 스페어 섹터들을 초기화하는 FTL_save_original_info 함수를 실행하여 스페어 섹터와 table 을 초기화한다.

```
else {
    Flash_write(index, sector, "o");
    count->write_count++;
}
```

만약 추가 연산 되어 있지 않다면 스페어 부분을 'o'로 할당하여 정보를 해당 섹터에 저장한다.

```
else {
    printf("이미 존재하는 값 FTL_overwrite 실행₩n");
    FTL_index_to_char(index, spare);
    Flash_write(*spare_index, sector, spare);
    count->write_count++;
    table[index] = (*spare_index)++;
    if (*spare_index == SPARESECTORTWO) {

        *spare_index = FTL_save_original_info(table, SPARESECTORTWO, str, sector, spare, count);

    }
    else if (*spare_index == ALLSECTOR) {

        *spare_index = FTL_save_original_info(table, SPARESECTORONE, str, sector, spare, count);

    }
```

만약 원본 섹터가 이미 할당 되어있고 추가 연산까지 되어 있다면 바로 spare_index 에 해당되는 주소에 정보를 위와 같이 입력한다.

Save 를 입력하면

```
else if (strcmp(sector, "save") == 0) {
    count->read_count = 0;
    count->write_count = 0;
    count->erase_count = 0;
    if (sparesector_index < SPARESECTORTWO) {
        QueryPerformanceFrequency(&micro_timer);
        QueryPerformanceCounter(&micro_start);
        sparesector_index = FTL_save_original_info(table, SPARESECTORTWO, buffer, sector, spare, count);
        QueryPerformanceCounter(&micro_end);
        timer = (micro_end.QuadPart - micro_start.QuadPart) / (double)micro_timer.QuadPart;


        printf("save(overwrite) 걸린시간 : %lf ₩n", timer);
    }
    else {
        QueryPerformanceFrequency(&micro_timer);
        QueryPerformanceCounter(&micro_start);
        sparesector_index = FTL_save_original_info(table, SPARESECTORONE, buffer, sector, spare, count);
        QueryPerformanceCounter(&micro_end);
        timer = (micro_end.QuadPart - micro_start.QuadPart) / (double)micro_timer.QuadPart;


        printf("save(overwrite) 걸린시간 : %lf ₩n", timer);
    }
    printf("읽기 : %d, 쓰기 : %d, 지우기 :%d₩n", count->read_count, count->write_count, count->erase_count);
    printf("=============================================₩n");
```

다음과 같은 코드를 실행한다.

핵심은 아까 wirte 에서 사용한 FTL_save_original_info 를 사용하여 전체 정보를 저장하는 것이다. 이때 두가지 로 나누어지는 이유는 FTL 알고리즘이 TABLE 에 저장되는 스페어 섹터의 반대편 섹터에 원본정보를 저장하여 덮어쓰는 연산을 하기 때문에 현재 할당되고 있는 스페어 주소가 어디인지를 확인해야 하기 때문에 있는 함수이다.

unsigned short FTL_save_original_info(unsigned short* table, unsigned short start_index, char* str, char* sector, char* spare, counting* count)

```
unsigned short FTL_save_original_info(unsigned short* table, unsigned short start_index, char* str, char* sector, char* spare, counting* count)
```

함수를 실행하기 위한 변수는 테이블 포인터 시작주소, buffer 포인터, sector 포인터, spare 포인터, counting 구조체 포인터이다.

```
for (unsigned short i = 0; i < SECTOR; i++)
```

실행하면 모든 섹터의 주소를 순회한다.

```
if (table[i] != i) { //다르다면
    for (unsigned short j = i / 32 * 32; j < i / 32 * 32 + 32; j++) {//다른 섹터가 존재하면 해당섹터의 수정되지 않은 원본 값을 여분 섹터에 저장
        if (table[j] == j) {
            Flash_read(table[j], str);
            count->read_count++;
            strncpy(sector, str, 512);
            strncpy(spare, str + 512, 16);
            if (spare[0] == 'o') { //값이 존재하면
                Flash_write(another_sector, sector, "o");
                count->write_count++;
                table[j] = another_sector++; // 이 섹터가 끝까지 가면 오류남
            }
        }
    }
```

만약 LSN 값과 PSN 의 값이 다른 부분이 존재한다면 해당 섹터의 블록을 다른 쪽의 여분 섹터에 차례로 저장하며 table 의 PSN 값 또한 해당되는 섹터 주소로 할당한다.

```
Flash_erase(i / 32);
count->erase_count++;
```

저장한 블록(원본 블록)은 삭제 된다.

```
if (another_sector / 32 == (start_index) / 32 + 9) {
    for (unsigned short j = 0; j < SECTOR; j++) {
        if (table[j] != j) {
            Flash_read(table[j], str);
            count->read_count++;
            strncpy(sector, str, 512);
            strncpy(spare, str + 512, 16);
            Flash_write(j, sector, "o");
            count->write_count++;
            table[j] = j;
        }
    }
    for (unsigned short j = start_index / 32; j < start_index  / 32 +10; j++) {
        Flash_erase(j);
        count->erase_count++;
    }
    another_sector = start_index;
}
```

만약 저장하는 블록 10 개를 다 사용한다면(추가적인 연산이 불가능할 때) TABLE 을 초기화하는 연산을 한다. 할당 되어있는 PSN 의 값을 LSN 에 할당하고 해당 테이블의 PSN 값을 LSN 으로 할당한다. 초기화가 완료되었다면 다시 저장연산을 실시한다.

```
i = i / 32 * 32 + 32;
```

이 연산이 완료되면 저장한 블록 뒤부터 다시 저장을 실시한다.

```
for (unsigned short i = 0; i < SECTOR; i++) {
    if (table[i] != i) {
        Flash_read(table[i], str);
        count->read_count++;

        strncpy(sector, str, 512);

        Flash_write(i, sector, "o");
        count->write_count++;
        table[i] = i;
    }
}
```

모든 저장이 완료된다면 전체를 순회하여 table 의 LSN 값과 PSN 값이 다른 섹터를 원본 섹터에 저장하는 연산을 한다. 이때 섹터의 여분부분에 'o'를 할당한다.

```
if (start_index == SPARESECTORONE) {
    for (unsigned short j = SPAREBLOCKTWO; j < ALLSECTOR / 32; j++) {
        Flash_erase(j);
        count->erase_count++;
    }
}
else if (start_index == SPARESECTORTWO) {
    for (unsigned short j = SPAREBLOCKONE; j < SPAREBLOCKTWO; j++) {
        Flash_erase(j);
        count->erase_count++;
    }
}

if (another_sector == SPARESECTORTWO - 1) {
    for (unsigned short j = SPAREBLOCKONE; j < ALLSECTOR / 32; j++) {
        Flash_erase(j);
        count->erase_count++;
    }
    return SPARESECTORTWO;
}
else if (another_sector == ALLSECTOR) {
    for (unsigned short j = SPAREBLOCKTWO; j < ALLSECTOR / 32; j++) {
        Flash_erase(j);
        count->erase_count++;
    }
    return SPARESECTORONE;
}

return another_sector;
```

모든 저장이 완료된다면 원본을 저장하던 여분 섹터를 시작주소로써 리턴하게 되는데 만약 가득 차 있다면 해당 섹터를 삭제하고 다른 쪽 섹터를 시작 주소로 할당한다. 처음 연산하던 섹터는 초기화 한다.

```
unsigned short FTL_index_to_short(char* str) {
    unsigned short i = 0;
    unsigned short index = 0;
    if (str[0] >= '0' && str[0] <= '9') {
        while (str[i] != ' ') {

            index *= 10;
            index += str[i] - '0';
            i++;
        }
        return index;
    }
    else {
        return 9999;
    }

}

void FTL_index_to_char(unsigned short index, char* str) {
    if (index == 0) {
        str[0] = '0';
        str[1] = '₩0';
    }
    else {
        unsigned short i = 0;
        unsigned short count = 1;
        unsigned short temp = index;
        for (unsigned short i = 0; i < 4; i++) {
            if (temp / 10 != 0) {
                count++;
                temp /= 10;
            }
        }
        for (short i = count - 1; i >= 0; i--) {
            str[i] = '0' + (index % 10);
            index /= 10;
        }
        str[count] = '₩0';
    }

}
```

이 함수들은 FTL 연산을 할 때 사용되는 함수로써 FTL_index_to_short 는 입력 받은 str 의 값을 unsigned short 값으로 치환하여 리턴 해주는 함수이다.

FTL_index_to_char 함수는 입력 받은 주소 값을 문자열로 치환하여 입력 받은 포인터에 입력해주는 함수 이다.

3. 테스트 (1장이내)

```
명령어 입력 : write 0 a
```

```
a
o
```

0 번지에 입력된다.

```
명령어 입력 : write 0 b
```

```
LSN     0  ||  PSN  6144
```

PSN 값이 변경되며 b 의 값이 해당 PSN 에 저장되었음을 확인 할 수 있다.

```
a
o
```

```
b
0
줄 1, 열 3244033   3,581,952자
```

6144 에 b 가 저장되었다.

```
명령어 입력 : save
```
저장을 한다면

```
save(overwrite) 걸린시간 : 0.036794
읽기 : 32, 쓰기 : 1, 지우기 :11
```

0 번 PBN 을 삭제하고 예상대로인 여분 섹터 한 세트 10 블록을 삭제하였으므로 11 번의 지우기 연산이 발생하였음을 알 수 있다. 읽기 32 번은 원본 블록을 읽어 32 번의 연산이 발생하였고 (LSN 1 ~ 31 까지) 쓰기는 변경 값인 LSN 0 || PSN 6144 에서만 한번 연산 되었음을 알 수 있다.

```
b
o
```

b 가 입력되었음을 확인할 수 있다.

블록 맵핑 알고리즘

1. 설계 (2장이상)

하드웨어의 기준은 3MB flash memory, 추가 블록 20개로 이루어짐

사용하는 DRAM은 논리주소 1B*191 = 191B, 섹터 정보 버퍼 529B 섹터 513B, 여분 섹터 17B 최소 사용 DRAM 1,250B(약 1KB) 이다

필수 구현 함수

Print_table ( ); FTL_read ( ); FTL_write ( );

블록 매핑 알고리즘에서 읽기 기능의 구현은 LBN에 입력 되어있는 PBN의 주소의 정보를 읽기만 하면 된다.

읽기연산은 만약 LBN과 PBN이 같다면 입력 받은 섹터주소의 나머지 값에 존재하는 위치의 값을 읽어온다. 하지만 여분 블록으로 할당 되어있는 경우에는 여분 블록의 여분 섹터의 값을 읽어 맞는 섹터주소를 읽어 온다. 이때 선별하는 방법은 할당된 여분 블록의 맨 마지막부터 읽어 가장 첫번째로 읽어오는 값을 가장 최신화 된 정보라 생각한다.

쓰기 연산은 사용자에게 LSN값을 읽어오면 해당되는 PBN값을 할당되지 않는 여분 블록의 주소를 가져온다. 이때 하나의 블록에서 사용하는 블록의 개수는 2개다. 블록에 순서대로 기존에 작성되었던 LBN에 해당되는 PBN의 정보들을 할당된 여분블록에 저장한다.

총 10개의 여분 블록을 다 사용한다면 가장 첫번째의 블록을 초기화한다. 초기화하면 LBN에 해당되는 주소 값을 PBN에 재할당하며 여분 블록에 저장했던 값을 가장 최신순대로 정보를 저장한다.

하나의 여분 블록 세트를 다 사용한다면 해당 블록을 원본 블록에 저장하고 다시 여분 블록에 저장한다.

기존 섹터에 정보가 있다면 비어 있는 여분 블록에 기존 할당되어있던 정보를 전체 옮겨준다

A	o
B	o

Write 0 C를 입력한다면

LBN 0 | PBN 0 -> LBN 0 | PBN 192

Table 주소를 여분블록으로 바꾼다.

추가 할당된 PBN 192에 LBN 0에 해당되는 정보를 옮겨준다.

A	0
B	1
C	0

비어 있는 여분블록 192~193블록에 기존정보를 차례로 저장한다. 그 후에 저장되는 정보는 이 과정 없이 여분블록 마지막 섹터에 차례로 저장한다.

A	0
B	1
C	0

여분 섹터 부분에는 기존 섹터의 주소를 입력한다. 저장기능을 실행하면 여분 블록의 뒤부터 연산하여 가장 최신의 정보를 뒤의 기존 섹터의 주소와 비교하며 저장한다.

C	o
B	o

저장기능을 실행하면 LBN 0에 해당되는 PBN 0에 저장한다. 가장 아래에서부터 저장하였기 때문에 0번 섹터에 "C"가 저장되며 가장 처음 입력된 A는 저장되지 않았음을 알 수 있다.

이 과정을 실행하면 사용하던 여분 블록은 초기화하여 다른 블록에서 사용할 수 있도록 한다.

저장 후에는 TABLE의 LBN값도 기존 값으로 초기화 한다.

LBN 0 | PBN 0
LBN 1 | PBN 1
LBN 2 | PBN 2
LBN 3 | PBN 3
LBN 4 | PBN 4
LBN 5 | PBN 5
LBN 6 | PBN 6
LBN 7 | PBN 7
LBN 8 | PBN 8
LBN 9 | PBN 9
.....

프로그램을 첫 실행하면 다음과 같이 TABLE이 세팅 된다.

각 LSN값은 unsigned char 값으로써 191개의 배열을 가지고 있다.

2. 구현 (1 장이상)

- 코드 설명

앞서 설명한 섹터 맵핑 알고리즘과 매우 유사하다.

```
char buffer[MAX] = { 0, };
char sector[513] = { 0, };
char spare[17] = { 0, };
unsigned short index = 0;
unsigned char table[SPAREBLOCKONE] = { 0, };

counting* count = malloc(sizeof(counting));
count->read_count = 0;
count->write_count = 0;
count->erase_count = 0;

LARGE_INTEGER micro_start, micro_end, micro_timer;
double timer;
```

main 함수에서 사용한 변수들이다.

Buffer: 한 섹터를 불러온다.

Sector: 섹터정보를 저장한다.

Spare: 섹터의 여분정보를 저장한다.

Index: LSN 입력이다.

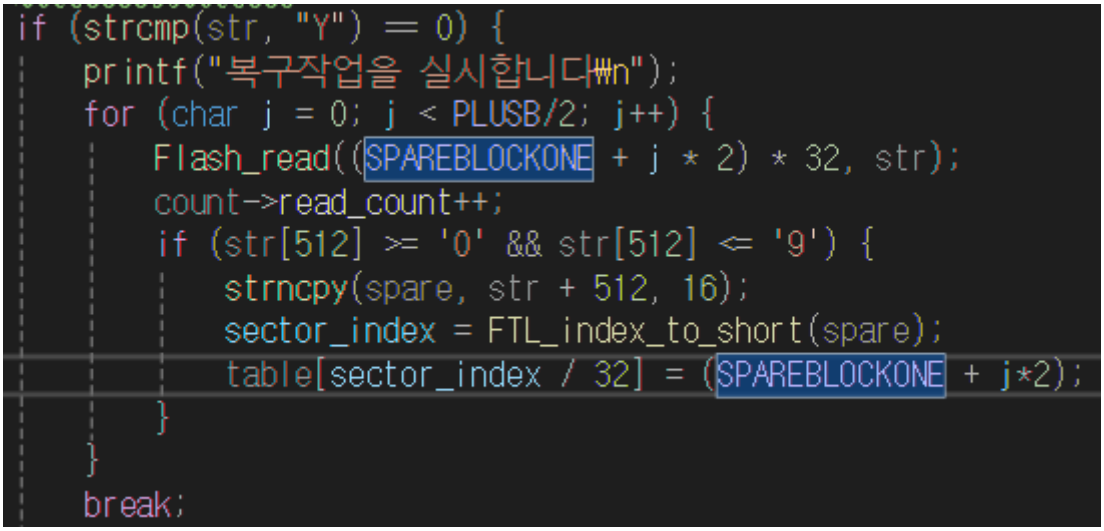
Table: 블록 맵핑 알고리즘의 LBN 주소다.

Count: 연산할때 사용하는 FlashMemory 연산을 저장하는 구조체이다.

Micro_start, micro_end, micro_timer: 마이크로세컨을 측정하기 위한 변수

Timer: 연산 시간을 저장하는 변수

처음 프로그램을 시작하면 Flash Memory 의 init() 함수를 실행하여 파일의 무결성을 체크한다. 무결성이 확인된다면 마지막으로 실행했던 동작에서 만약 여분 섹터에 저장하고 있던 정보가 존재하는지 검사한다. 그 동작을 검사하는 함수는 void FTL_is_empty_spare_block(unsigned char* table, char* str, char* spare, counting* count) 이다. 섹터 맵핑 알고리즘과 비슷하게 각 블럭의 첫번째 섹터를 읽고 만약 값이 있다면 복구 작업을 실시한다.



```
if (strcmp(str, "Y") == 0) {
    printf("복구작업을 실시합니다₩n");
    for (char j = 0; j < PLUSB/2; j++) {
        Flash_read((SPAREBLOCKONE + j * 2) * 32, str);
        count->read_count++;
        if (str[512] >= '0' && str[512] <= '9') {
            strncpy(spare, str + 512, 16);
            sector_index = FTL_index_to_short(spare);
            table[sector_index / 32] = (SPAREBLOCKONE + j*2);
        }
    }
    break;
```

복구 작업을 실시하면 스페어 블록을 순회하며 값이 존재하는지 확인한다. 만약 값이 존재한다면 해당 블록의 첫번째 섹터의 스페어 값의 주소를 바탕으로 원본 블록의 주소를 찾아 해당 블록의 주소를 복구 대상의 주소로 치환한다.

```
printf("TABLE           ...  매핑 테이블 출력(섹터)₩n");
printf("read LSN        ...  해당 LSN의 데이터 읽기₩n");
printf("write LSN DATA  ...  해당 LSN에 데이터 쓰기₩n");
printf("save            ...  여분 섹터 초기화₩n");
printf("exit            ...  프로그램 종료₩n₩n");
```

복구작업까지 끝냈다면 첫 실행화면이 나온다.

여기서 TABLE 을 입력하면 해당 블록 맵핑의 LSN 값과 PSN 값을 알 수 있다.

```
void Print_table(unsigned char* table) {
    for (unsigned char i = 0; i < SPAREBLOCKONE; i++) {
        printf("LBN %5hd  ||  PBN %5hd₩n", i, table[i]);
    }
}
```

Print_table()함수를 사용하여 해당 테이블을 보여준다.

다음은 read LSN 을 입력했을 경우이다

```
else if (strcmp(sector, "read") == 0) {
    count->read_count = 0;
    count->write_count = 0;
    count->erase_count = 0;
    scanf("%hd", &index);
    if (index >= SPARESECTORONE) {
        printf("초과된 주소입니다.₩n");
    }
    else {
        QueryPerformanceFrequency(&micro_timer);
        QueryPerformanceCounter(&micro_start);
        FTL_read(index, table, buffer, sector, spare, count);
```

마지막 블록 값의 마지막 섹터를 넘지 않으면 FTL_read 함수를 실행한다.

```
void FTL_read(unsigned short index, unsigned char* table, char* str, char* sector, char* spare, counting* count)
    if (index > 6143) {
        printf("너무 큰 값입니다.₩n");
    }
    else {
        if (table[index / 32] != index / 32) {
            for (char i = 63; i >= 0; i--) {
                Flash_read(table[index / 32]*32 + i, str);
                count->read_count++;
                strncpy(spare, str + 512, 16);
                printf("%s, %hd₩n",spare, FTL_index_to_short(spare));
                if (index == FTL_index_to_short(spare)) {
                    strncpy(sector, str, 512);
                    printf("sector : %s₩n", sector);
                    i = -1;
                }
                else if (i == 0 && index != FTL_index_to_short(spare) / 32) {
                    printf("해당 섹터에 값이 없습니다.₩n");
                }
            }

        }
        else {
            Flash_read(index, str);
            count->read_count++;
            strncpy(spare, str + 512, 16);
            if (spare[0] == ' ') {
                printf("해당 섹터의 값이 존재하지 않습니다.₩n");
            }
            else {
                strncpy(sector, str, 512);
                printf("sector : %s₩n", sector);
            }
        }
    }
```

다음은 FTL_read 의 코드이다.

할당하는 변수는 주소 값, 테이블 포인터, buffer 포인터, sector 포인터, spare 포인터, counting 구조체 포인터다.

유효 범위 내의 주소가 들어오면 입력된 주소의 값을 읽어온다.

만약 LBN 과 PBN 의 값이 다르면 추가 블록에 저장되어 있기 때문에 가장 최신정보를 읽기 위해 맨 뒤(64 번째 섹터)부터 앞으로 읽어 index 값과 같은 스페어 섹터 값일 때 정보를 읽어온다.

Write PSN DATA 를 입력하면

```
else if (strcmp(sector, "write") == 0) {
    count->read_count = 0;
    count->write_count = 0;
    count->erase_count = 0;
    scanf("%hd", &index);
    scanf("%s", sector);
    QueryPerformanceFrequency(&micro_timer);
    QueryPerformanceCounter(&micro_start);
    FTL_write(index, table, buffer, sector, spare, count);
```

다음과 같이 FTL_write 를 실행한다.

```
void FTL_write(unsigned short index, unsigned char* table, char* str, char* sector, char* spare, counting* count)
    if (index > 6143) {
        printf("너무 큰 값입니다.₩n");
    }
    else {
        if (index / 32 == table[index / 32]) {
            Flash_read(index, str);
            count->read_count++;
            if (str[512] == ' ') {
                Flash_write(index, sector, "o");
                count->write_count++;
            }
            else {
                printf("이미존재하는 섹터₩n");
                FTL_overwrite(index, table, str, sector, spare, count);
            }
        }
        else {
            printf("이미 추가블럭에 연산하는 블럭₩n");
            FTL_overwrite(index, table, str, sector, spare, count);
        }
    }
```

FTL_write 는 다음과 같다. 사용하는 변수로 주소 값, 테이블 포인터, buffer 포인터, sector 포인터, spare 포인터 , counting 구조체 포인터가 있다.

이 부분은 간단하게 해당 주소의 값이 존재한다면 덮어쓰기 연산을 실행한다. FTL_overwrite 함수를 사용하여 저장해야 할 주소를 자동으로 찾아 알아서 저장한다.

이미 추가블록에 연산중인 경우에도 FTL_overwrite 함수를 사용한다.

```
void FTL_overwrite(unsigned short index, unsigned char* table, char* str, char* sector, char* spare, counting* count) {
    char sector_temp[513] = { 0, };
    if (index / 32 == table[index / 32]) { //신규 스페어블럭 연산
        unsigned char i = 0;
        unsigned char j = 0;
        unsigned char k = 0;
```

다음은 FTL_overwrite 함수이다. 사용하는 변수는 주소 값, 테이블 포인터, buffer 포인터, sector 포인터, spare 포인터 counting 구조체 포인터가 있다. 추가로 기존 원본 sector 정보 값을 저장하기 위해 sector_temp 를 새로 선언해주고 while 문 사용을 위한 변수 설정을 해준다.

```
while (i < PLUSB / 2) {
    Flash_read((SPAREBLOCKONE + (i * 2)) * 32, str);
    count->read_count++;
    if (str[512] == ' ') {
        table[index / 32] = (SPAREBLOCKONE + (i * 2));// 192 194 196 ....
        while (k < 32) { //원본 블럭 전체 저장(해당부분)
            Flash_read((index / 32) * 32 + k, str); // 원본 읽기
            count->read_count++;

            if (str[512] == 'o') {
                FTL_index_to_str((index / 32) * 32 + k, spare);
                strncpy(sector_temp, str, 512);
                Flash_write((table[index / 32] * 32) + j++, sector_temp, spare); //바뀐 블럭에 저장
                count->write_count++;
            }
            k++;
        }//스페어 블럭에 저장


        FTL_index_to_str(index, spare);
        Flash_write(((table[index / 32] * 32) + j), sector, spare);
        count->write_count++;
        break;
    }
```

처음 실행하면 스페어 블록에 빈공간이 있는지 검사한다. 만약 빈 공간이 존재한다면 원본 블록의 데이터를 스페어 블록에 전체 저장한다. 이때 섹터 스페어 공간에 원본 주소 값을 넣어 저장한다. 빈 섹터는 무시한다.

블록의 주소는 2 블록당 1 블록의 저장을 하기 때문에 2 블록씩 검사한다.

```
else {
    i++;
    if ((i * 2) + 192 == ALLSECTOR / 32) {
        for (j = 0; j < PLUSB / 2; j++) {
            FTL_save(SPAREBLOCKONE + j * 2, table, str, sector_temp, spare, count); // 스페어 초기화 저장
        }
        j = 0;
        make_table(table);
        table[index / 32] = (SPAREBLOCKONE);
        while (k < 32) { //원본 블럭 전체 저장(해당부분)
            Flash_read((index / 32) * 32 + k, str);
            count->read_count++;

            if (str[512] == 'o') {
                FTL_index_to_str(((index / 32) * 32) + k, spare);
                strncpy(sector_temp, str, 512);
                Flash_write((table[index / 32] * 32) + j++, sector_temp, spare);
                count->write_count++;
            }
            k++;
        }//스페어 블럭에 저장
        FTL_index_to_str(index, spare);
        Flash_write((table[index / 32]*32 + j), sector, spare);
        count->write_count++;
        i = 0;
        break;
```

만약 빈 스페어 블록이 없다면 FTL_save 함수를 사용하여 정보를 저장한다. save 기능과 동일하다. 블록의 뒤부터 순차적으로 순회하여 가장 최신의 값의 정보를 불러온다. 저장 기능이 완료된다면 스페어 블록 1 번에 다시 저장한다.

```
else { //여기서부터 기존 스페어에 저장
    unsigned char i = 0;

    Flash_read(table[index / 32] * 32 + 63, str);
    count->read_count++;
    if (str[512] != ' ') {
        unsigned char temp_index = table[index / 32];
        FTL_save(table[index / 32], table, str, sector_temp, spare, count);
        table[index / 32] = temp_index;
    }
    else {
        while (i < 64) {
            //64칸 채우고 다음 칸으로 이동하게 만듬
            //>> 64칸까지는 유지 만약 64칸에 값이 있으면 연산 ㄱㄱ
            Flash_read(table[index / 32] * 32 + i, str);
            count->read_count++;

            if (str[512] == ' ') {
                FTL_index_to_str(index, spare);
                Flash_write(table[index / 32] * 32 + i, sector, spare);
                count->write_count++;
                break;
            }
            else {

                i++;
            }
```

기존에 스페어블록이 할당되어있다면 가장 최신순으로 할당한다. 만약 64 번째 섹터까지 다 사용하고 있다면 원본 데이터를 저장하고 해당 블록에 연속하여 저장한다.

Save 를 입력하면

```
else if (strcmp(sector, "save") == 0) {
    count->read_count = 0;
    count->write_count = 0;
    count->erase_count = 0;
    QueryPerformanceFrequency(&micro_timer);
    QueryPerformanceCounter(&micro_start);
    for (int i = 0; i < PLUSB/2; i++) {
        FTL_save(SPAREBLOCKONE + i * 2, table, buffer, sector, spare, count);
```

다음과 같은 동작을 한다.

```
void FTL_save(unsigned char index, unsigned char* table, char* str, char* sector, char* spare, counting* count) {
    unsigned short sector_index = 0;


    for (unsigned char i = 0; i < SPAREBLOCKONE; i++) {
        if (table[i] == index) {
            Flash_erase(i);
            count->erase_count++;

            for (char j = 63; j >= 0; j--) {
                Flash_read(table[i] * 32 + j, str);
                count->read_count++;
                if (str[512] != ' ') {
                    strncpy(sector, str, 512);
                    strncpy(spare, str + 512, 16);
                    sector_index = FTL_index_to_short(spare);
                    Flash_read(sector_index, str);
                    count->read_count++;
                    if (str[512] == ' ') {
                        Flash_write(sector_index, sector, "o");
                        count->write_count++;
                    }
                }
            }
            table[i] = i;
        }
    }
    Flash_erase(index);
    count->erase_count++;
    Flash_erase(index + 1);
    count->erase_count++;

}
```

FLT_save 는 다음과 같다. 사용하는 변수는 주소 값, 테이블 포인터, buffer 포인터, sector 포인터, spare 포인터, counting 구조체 포인터이다. 추가로 sector_index 변수를 사용하여 기존 주소를 저장한다. (섹터 스페어 부분의 값)

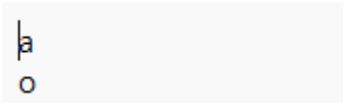
Table 을 전체 순회 하면서 LBN 과 PBN 이 다른 값을 찾아 다르다면 해당 PBN 의 정보를 LBN 에 저장한다. 이때 PBN 에 순서대로 저장했으므로 가장 나중의 정보가 최신의 정보이므로 뒤부터 앞으로 읽어 가장 최신의 값만 LBN 에 알맞게 저장한다. PSN 은 임시 저장된 값의 섹터 스페어 부분에 저장 되어있다. 모든 정보를 저장한다면 table 값을 초기화 하고 쓰였던 PBN 의 값을 초기화 한다.

```c
unsigned short FTL_index_to_short(char* str) {
    unsigned short i = 0;
    unsigned short index = 0;
    if (str[0] >= '0' && str[0] <= '9') {
        while (str[i] != ' ') {

            index *= 10;
            index += str[i] - '0';
            i++;
        }
        return index;
    }
    else {
        return 9999;
    }

}


void FTL_index_to_str(unsigned short index, char* str) {
    if (index == 0) {
        str[0] = '0';
        str[1] = '₩0';
    }
    else {
        unsigned short i = 0;
        unsigned short count = 1;
        unsigned short temp = index;
        for (unsigned short i = 0; i < 4; i++) {
            if (temp / 10 != 0) {
                count++;
                temp /= 10;
            }
        }
        for (short i = count - 1; i >= 0; i--) {
            str[i] = '0' + (index % 10);
            index /= 10;
        }
        str[count] = '₩0';
    }

}
```

다음은 문자열을 unsigned short 로 바꾸고 unsigned short 를 문자열로 바꾸도록 하는 함수이다.

3. 테스트 (1 장이내)

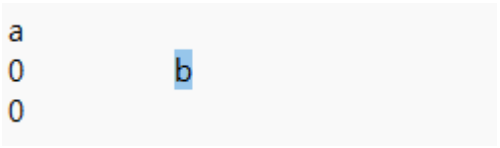
```
명령어 입력 : write 0 a
```
를 입력하면



FlashMemory.txt에 저장된다.

```
명령어 입력 : write 0 b
```
같은 위치에 다시 입력하면

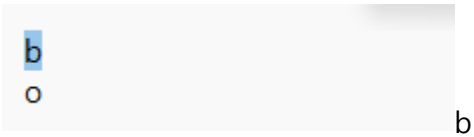
```
LBN     0  ||  PBN   192
```
PBN의 값이 변경되었다.



기존의 a값과 변경할 값 b가 192블록에 입력 되어있음을 알 수 잇다. 스페어 공간에 주소 값인 0 이 입력 되어있음을 알 수 있다.

```
명령어 입력 : save
```
이 값을 저장하면

```
LBN     0  ||  PBN     0
```
PBN값은 원래대로 돌아오고



b 로 저장되었음을 알 수 있다.

원본 값인 a는 사라졌음을 알 수 있다.

3.2차 프로젝트 202120204 심영준

섹터/블록 매핑 알고리즘 비교분석 (총 2장이상)

1. 하드웨어적 스펙 분석

- 본인이 만든 섹터/블록 알고리즘의 여분 블록 수

섹터 맵핑 알고리즘의 여분 블록의 수는 총 20개의 블록으로 이루어져 있다.

블록 맵핑 알고리즘의 여분 블록의 수는 총 20개의 블록으로 이루어져 있다.

- DRAM 요구량 (DRAM requirement) //본인의 DRAM용량 기입. 계산 공식 필요.

//본인의 섹터/블록 매핑 알고리즘의 비교를 위한 하드웨어적 스펙의 정당성 설명 필요

섹터 맵핑 알고리즘:

Unsigned short TABLE[6143] = 2*6143 = 12286B

char buffer[MAX] = 529B

char sector[513] = 514B

char spare[17] = 18B

unsigned short index = 2B

unsigned short sparesector_index = 2B

counting* count = malloc(sizeof(count)) = 16B

LARGE_INTEGER micro_start, micro_end, micro_timer = 24B

double timer = 8B

첫 실행 시의 크기

12286 + 529 + 514 + 18 + 2 + 2 + 16 + 24 + 8 = 13,399B =약 13.09KB

블록 맵핑 알고리즘:

Unsigned char TABLE[191] = 1*192 = 192B

char buffer[MAX] = 529B

char sector[513] = 514B

char spare[17] = 18B

unsigned short index = 2B

counting* count = malloc(sizeof(count)) = 16B

LARGE_INTEGER micro_start, micro_end, micro_timer = 24B

double timer = 8B

첫 실행 시의 크기

192+529+514+18+2+16+24+8 = 1303B = 약 1.3KB

2. 성능 측면(storage performance) 분석

- 섹터/블록 알고리즘의 읽기/쓰기/지우기 연산의 횟수

//섹터/블록 매핑 알고리즘에 수행된 연산 read/write/erase 횟수 보여주기

- 섹터/블록 알고리즘의 전체 성능 시간

//전체 수행 시간 보여주기. Read=15μs, write= 200μs, erase = 2ms.

측정을 위한 테스트 5가지 진행

test1 ... 모든 블록에 한번씩 저장 두번하기

test2 ... 모든 섹터에 한번씩 저장 두번하기

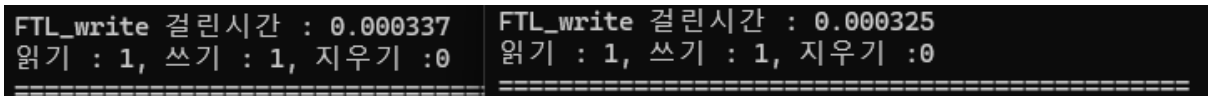
test3 ... 하나의 블록 전체에 저장 두번하기

test4 ... 랜덤주소에 5000번 저장하기

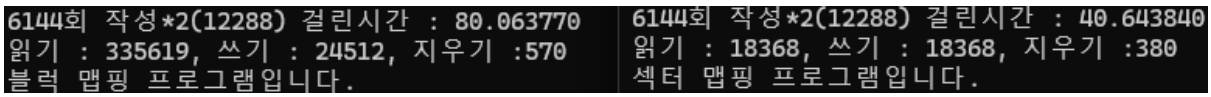
test5 ... 한 섹터에 1000번 저장 하기

read (read 0)

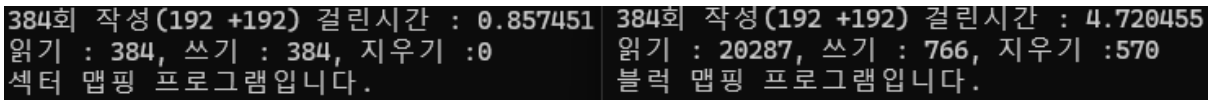
Write (write 50 a)



test 1

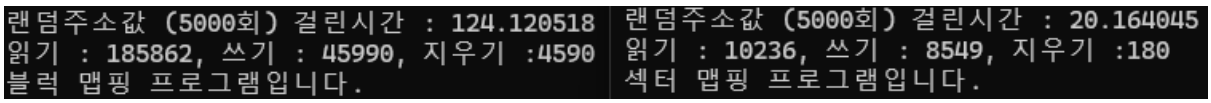


Test 2

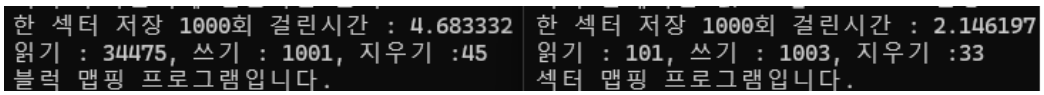


Test3

Test4



Test5



3. 성능 분석 및 결론

//성능 차이에 관한 원인분석

//알고리즘의 장단점 서술

5가지의 테스트의 결과로 보면 섹터 맵핑 알고리즘이 블록 맵핑 알고리즘보다 2상의 속도가 빠른 것을 알 수 있다.

이유로는 블록 맵핑은 원하는 섹터의 정보를 찾기 위해 해당 블록의 섹터와 추가적인 여분 블록의 섹터까지 전체 순회하여 검색을 하기 때문에 더 많은 읽기연산과 덮어쓰기

시에 한 블록이 정보를 저장하는 여분 섹터의 개수가 섹터 맵핑보다 최대 5배정도 적기 때문에 지우는 횟수가 많다는 것을 알 수 있다.

그에 비해 섹터 맵핑은 TABLE에 섹터 주소를 바로 저장하기 때문에 해당 주소의 값을 원본 섹터 주소에 옮겨주는 동작만 하면 되기 때문에 속도가 블록 맵핑보다 빠르다는 것을 알 수 있다.

각 알고리즘의 장단점은 다음과 같다

섹터 맵핑 알고리즘

장점 : 속도가 빠르다. 읽기 쓰기 덮어쓰기에서 연산 횟수가 적다.

단점 : DRAM의 사용량이 특히 TABLE에서 블록 맵핑 보다 크다.

블록 맵핑 알고리즘

장점 : DRAM의 사용량이 특히 TABLE에서 섹터 맵핑 보다 작다.

단점 : 속도가 느리다. 읽기 쓰기 덮어쓰기에서 연산 횟수가 많다.

최종적으로 장단점을 비교하자면 해당 섹터 맵핑 알고리즘은 빠른 연산이 필요하고 여유 있는 자원의 상황에서 사용하기 좋고 해당 블록 맵핑 알고리즘은 속도를 포기하고 적은 DRAM을 사용하는 장치에서 사용하기 좋다는 것이다.