낸드플래시 구현 프로그램 계획서

과 목	소프트웨어공학
담 당 교 수	권 세 진
학 번	201720970
학 과	소프트웨어·미디어·산업공학부
이 름	권 대 한

목 차

1. 낸드플래시 구현 프로그램이란?

Flash Memory는 HDD와 물리적인 구조가 완전히 다른 만큼 Flash Memory를 HDD 기반 소프트웨어에서 주 저장장치로 사용하기 위해서 Flash Memory에 적합한 데이터 액세스 방법이 필요하게 되었다. 기존의 HDD와 완전히 다른 물리적 구조, 데이터 저장 구조를 가짐에도, 기존 프로그램, HDD의 파일 시스템에서 Flash Memory는 어떻게 최적의 성능을 보여주는 것일까? 이는 Flash Memory를 관리하는 FTL 알고리즘, 컨트롤러로 인해 가능했다. 또 현재 Flash Memory의 상용화 이후 오랜 시간이 지났지만, 소프트웨어에서의 대응은 아주 미흡하다. 예를 들어 최신 버전의 운영체제인 Windows 10에서는 Trim(유휴 상태에서 필요 없는 블록을 지워주는 기술) 정도만 지원하며, SSD라고 인식을 하고 있음에도 불구하고, 유휴 상태에서 SSD에 Trim과 자동으로 디스크 조각 모음을 하도록 명령을 내려 SSD의 수명을 줄이고 있다. 심지어 어떤 운영체제는 인식조차 불가한 일도 있다.

NAND Flash의 셀당 저장 비트가 늘어날수록 수명이 수십, 수백 배 줄어들면서 데이터 액세스 시간, 속도 모두 상당히 떨어지게 되는데, 생산단가의 절약과 Flash Memory 진보를 위해서 Flash의 관리를 담당하는 FTL 알고리즘의 개선은 꾸준히 진행되어야 한다.

- 전체적인 프로그램 구성 계획

사용자에게 Text 기반의 명령을 입력받아, 각 기능이 작동하는 것이 프로그램의 요구 점이므로, 콘솔의 입력 Stream에서 Token 단위로 명령 신호를 받아오면서 각 기능을 실행시키는 프로그램이다.

또 이번 프로그램은 NAND Flash를 FTL이라는 특별한 계층을 이용해서 수정하는 것이므로, 데이터를 읽고자 할 때, 실제 File System과 같이 논리 인덱스를 통해 물리 인덱스인 PSN를 알아낸 이후 각 작업을 실행하도록 하였다. Overwrite(Update)가 발생한다면, 본래 FTL을 이용해 NAND Flash Erase를 최대한 방지할 것이지만, 이는 결국 비휘발성 메모리이므로, 언젠가는 데이터를 제거해야 할 상황이 올 것이다. 최소 지우기 단위인 Block을 이용한 함수까지 구현하면 추후 완벽한 NAND Flash 시뮬레이션 프로그램을 만들 수 있다고 생각되었다.

- 기능 구현에 대한 계획

(1) 프로그램 초기 실행 시

이 프로그램은 Sector / Block Mapping 알고리즘의 사용을 지원하며, 결론적으로 두 가지 알고리즘의 실행 결과는 같겠지만, 실제 작동 방식이 다르며 구현되어야 하는 부분이 서로 다르다. 이 같은 이유로 프로그램 실행 시 시험해보고자 하는 매핑 방법을 입력받은 후에, 저장된 정보를 복원해서 Flash Memory를 사용하는 듯한 생각을 가지도록 하였으며, 해당 알고리즘에 해당하는 메뉴를 실행시킨다. 출력된 명령 중에 사용하고자 하는 명령이 있어, 그 명령을 입력받았다면 명령에 맞는 각 기능을 실행한다.

<Sector Mapping Menu를 선택했을 경우>

(2) NAND Flash FTL 맵핑 테이블과 데이터 공간 할당 기능 호출 시

기존의 PSN 기반 프로그램과 같이 할당하고자 하는 용량을 입력받았다면, 포인터 배열, 동적 배열로 할당하고자 하는 Flash Memory 공간을 할당한다. 할당이 끝났다면, 동시에 정적 형태의 Sector Mapping Table 생성을 위해, Sector Size를 구해 Sector와 같은 크기의 Table을 생성하며, 모든 Logical Sector Number(Table의 Index)와 Physical Sector Number(Table Index의 데이터)를 연결해줘서 초기 데이터 액세스에 문제가 없도록 하였다.

(3) NAND Flash FTL 맵핑 테이블 출력 호출 시

Table 출력 호출 시에 Mapping Table의 Index와 해당 Index가 가진 값을 출력하며, Flash Memory 할당 시, 할당된 공간의 각 Sector에 Logical Sector Number를 연결해두는 방식인 정적 할당을 이용해 Table을 생성했으므로, Flash Memory 할당 직후 Mapping Table을 출력하고자 한다면, 모든 섹터의 Mapping을 출력해줄 것이다.

(4) NAND Flash FTL 맵핑 테이블을 통한 데이터 read 호출 시

Flash Memory 공간 할당 시 만들어진 Sector Mapping Table의 Index를 Logical Sector Number, Index가 가진 값을 Physical Sector Number로 Mapping 되어있으므로, 사용자가 특정 값을 읽기 위해 LSN 전달해주었다면, 먼저 Mapping Table의 LSN Index에 접근하여 실질적인 데이터가 저장된 PSN을 읽기 함수에 전달한다면, 결과적으로 Table 내 맵핑 된 PSN의 데이터를 출력하게 하여, 사용자는 LSN의 데이터를 읽었다고 생각하게 되며, 읽기 함수를 종료하게 된다.

(5) NAND Flash FTL 맵핑 테이블을 통한 데이터 write 호출 시

기존의 읽기 함수와 마찬가지로 사용자는 LSN에 데이터를 쓰는 것으로 인지하고 있으므로, 데이터를 쓰기 위해 LSN과 쓰고자 하는 데이터가 입력되었다면, Mapping Table의 Index를 참조하여, 데이터 쓰기 함수에 PSN과 데이터를 리턴하여, 데이터 쓰기 작업을 시작한다. 만약 해당 PSN이 이미 사용 중이라면, 지우기 연산을 방지하기 위해 여유 블록의 섹터에 Overwrite 데이터를 기록하며, Mapping Table의 PSN 수정 후 마치게 된다. 이 과정을 통해 사용자는 Flash Memory에 대한 E-B-W 특성에 대해 간과하게 된다. 그리고 Overwrite가 발생하여, 여유 블록에 데이터를 쓰려고 하는데, 여유 블록이 가득 차 있는 경우가 있을 수 있다. 실제 호출된 PSN에 Overwrite 되지 않기 때문이다.

이를 해결하기 위해 든 생각이 실질적으로 매핑 테이블은 DRAM 내 저장되어있으므로, Access Time에 영향을 받지 않는다는 것을 이용하기로 하였다. 그리고 여유 블록이 가득 차 있는 것은 Flash Memory 내 여유 공간이 더 없다는 것을 뜻하기 때문에, 여유 블록의 데이터를 RAM에 저장해야 한다. 그리고 이 복사 과정에서 생각해야 하는 것이 결국 우리는 실질적으로 사용되는 (Overwrite 된) 데이터만 필요하므로, Mapping Table을 참조해서 실제로 사용되는 데이터만 RAM에 할당한 여유 블록에 복사해준다. 이때 Mapping이 새로 필요하다고 생각될 수도 있겠지만, 기존의 여유 블록 전체를 복사하여 사용되는 데이터가 있던 위치에 그대로 복원하기 때문에 Mapping은 필요 없다.

<Block Mapping Menu를 선택했을 경우>

(2) NAND Flash FTL 맵핑 테이블과 데이터 공간 할당 기능 호출 시

Sector Mapping 메뉴의 할당 기능과 같은 크기로 Flash Memory를 할당해야 하므로, 기존의 할당 함수를 그대로 사용하였으며, 대신 기존의 Sector Mapping Table과 완전 다른 Block 기반 Mapping Table을 생성해야 하므로, 호출된 메뉴가 Block 할당 함수라면, 할당된 Flash Memory의 Sector Size를 기반으로 할당된 Block Size를 구해 Block Mapping Table을 생성하고, 모든 Logical Block Number(Table의 Index)와 Physical Block Number(Table Index의 데이터)를 연결해주면서, 할당 직후에 Block 기반 데이터 Access를 하고자 할 때 오류를 방지하였다.

(3) NAND Flash FTL 맵핑 테이블 출력 호출 시

Block Mapping Table 출력이 호출되었을 때, 기존의 Sector Mapping Table과 같이 정적 형태로 모든 Logical Block Number에 Physical Block Number가 모두 매핑되어 있으므로, Flash Memory 할당 직후 Mapping Table 출력을 호출한다면, 단순히 호출된 Logical Block Number가 할당된 Physical Block Number를 출력해준다.

(4) NAND Flash FTL 맵핑 테이블을 통한 데이터 read 호출 시

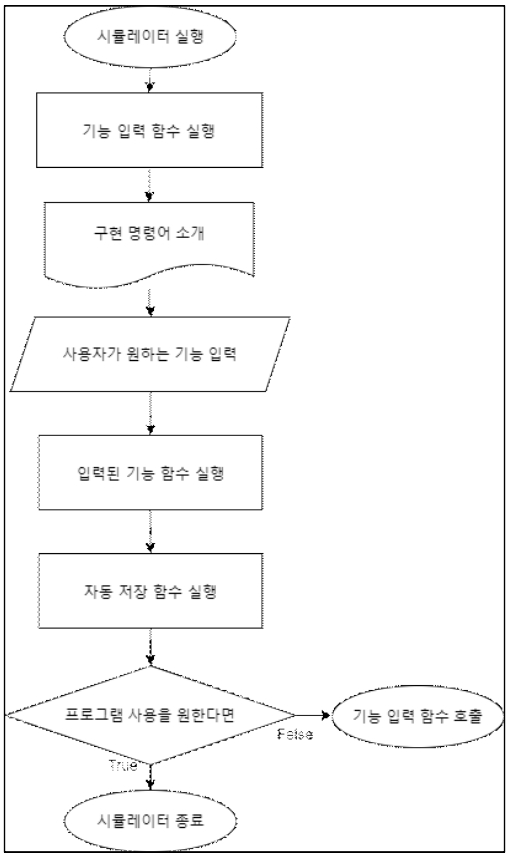
Sector Mapping 방식과는 다르게 Block 단위로 Flash Memory의 공간에 액세스할 수 있으며, 기존의 HDD 기반 File System이 Sector 단위로 R/W가 가능하므로, Sector 단위의 Flash Memory(LSN) 접근이 시도되었을 때, Block 단위 데이터 접근을 도와주는 Solver를 통해 LBN, PSN, offset을 구해 읽기 함수에 리턴해줘서, 데이터를 읽을 수 있게 된다. 이 과정으로 LSN의 위치에 데이터 저장이 되었다고 착각하게 만든다.

(5) NAND Flash FTL 맵핑 테이블을 통한 데이터 write 호출 시

블록 할당은 입력된 LSN을 기반으로 Solver를 통해, 단 하나의 논리 Block, offset을 구해주기 때문에, Sector Mapping과는 다르게, 여유 블록에 특정 섹터만 할당해주기가 어렵다. 데이터를 쓰기 위해 LSN과 쓰고자 하는 데이터가 입력되었다면, 먼저 LSN을 기반으로 Solver를 이용해 LBN과 offset을 구하며, Mapping Table을 참조하여 데이터 쓰기 작업을 시작한다. 만약 해당 PSN이 이미 사용 중이라면, PSN이 속한 블록에 다른 데이터가 존재하는지 확인한다. 이후 여유 블록에 백업 데이터와 Overwrite 하고자 하는 데이터를 기록하며, Mapping Table의 PBN 수정 후 마치게 된다. 이후 사용하지 않는 블록은 지운다.

2. 논리 구성도

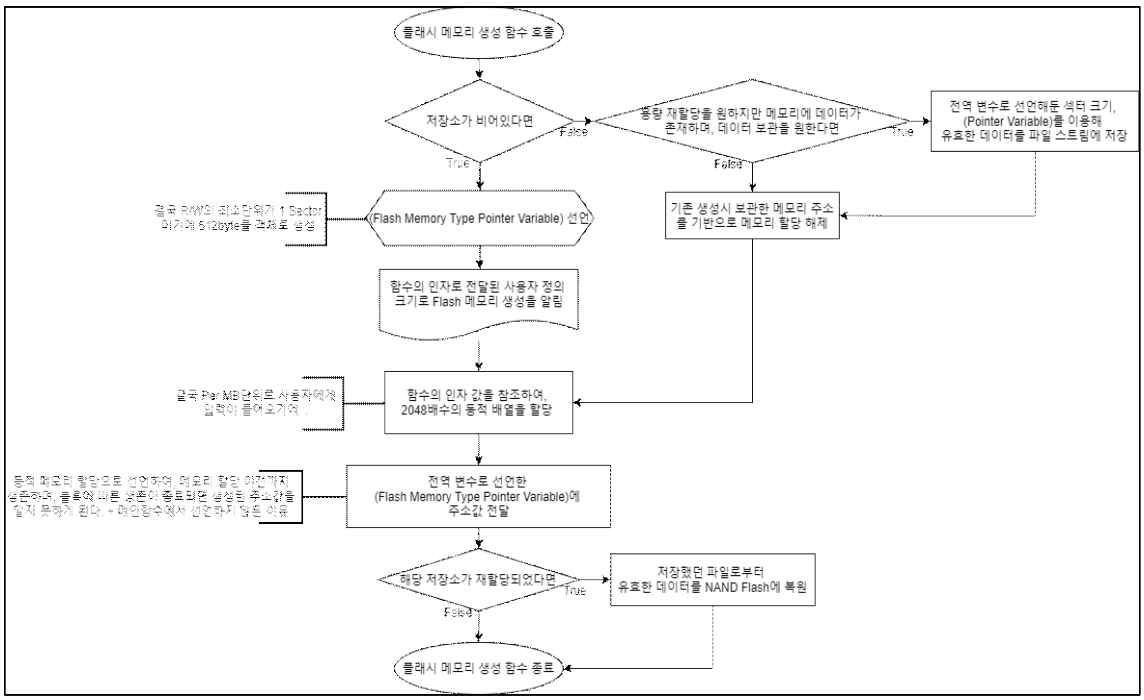
<시뮬레이터의 시작>



요구사항을 설계하면서, 들었던 생각은 이 프로그램의 요구 점인 Flash Memory의 특성에 맞게 설계해야 한다고 생각이 들었으며, 결과적으로 최대한 용량 낭비가 없게 저장하면서 추후 데이터를 읽고, 저장할 때도 효율적으로 사용할 수 있는 데이터 구조를 사용해야겠다고 생각했다. Class를 이용한 객체 배열 형태의 데이터 구조를 사용해 Flash Memory의 섹터와 비슷한 구조를 가지도록 만들었다. 그리고 기존에 작성한 PSN 기반 프로그램은 액세스하고자 하는 인덱스를 직접 접근할 수 있어 연속된 메모리 주소로 저장하는 배열의 특성상 STL보다 빠른 액세스가 가능하며, 비교적 적은 메모리를 사용해 프로그램을 구현 할 수 있었다.

그리고 이번 프로그램의 요구사항은 F2FS를 제외한 HDD 기반 File System에서의 성능과 수명을 최적화하기 위해 특별한 계층인 FTL을 만들어서 Flash Memory 친화적인 작동을 기대하는 프로그램이므로, 먼저 프로그램 실행 시, Token 단위로 입력된 명령의 실행을 FTL Table을 통해 실행시키도록 하여 수명, 성능을 최적화하였다. 그리고 입력된 텍스트를 구분해서, 실행 명령(데이터)으로 가져옴으로, 미리 구현된 각 기능 함수에 전달해줌으로 실질적인 기능을 실행한다.

<Flash Memory 생성 함수가 호출된다면>

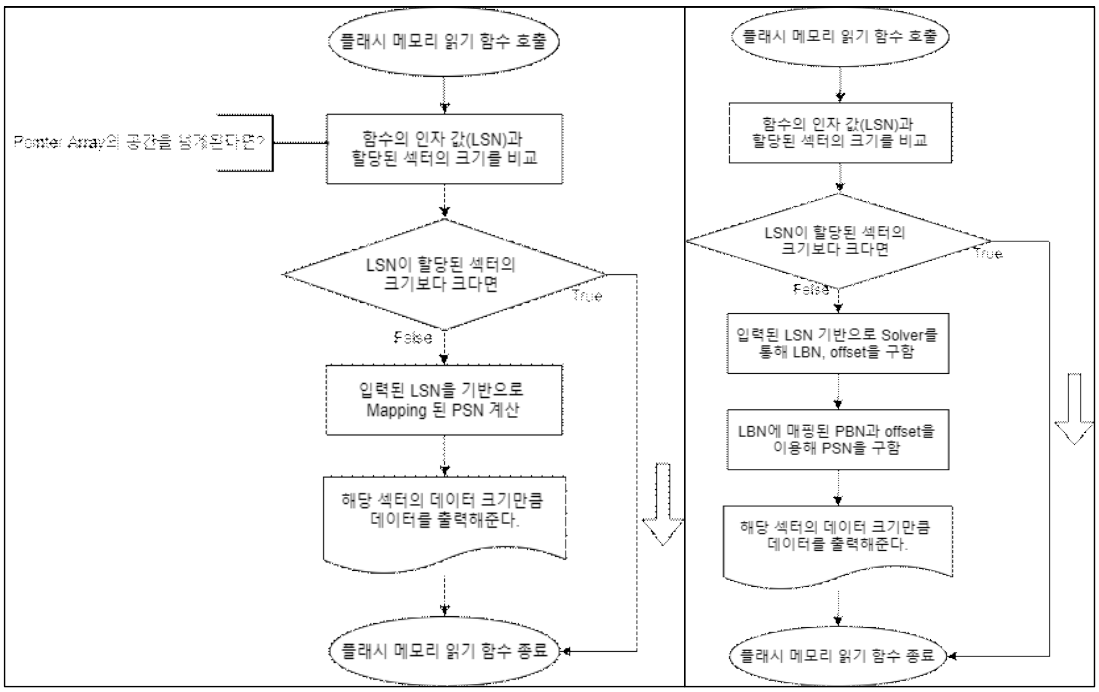


생성(할당) 함수가 실행되었을 때, 객체 배열을 할당하기 위해서, 포인터 배열을 선언하고, 사용자가 입력한 용량의 공간을 할당해줌으로 함수가 종료된다. 그러나 Flash Memory가 이미 할당이 되어있는 상태에서 할당이 두 번 실행된다면, 여러 문제가 발생하게 된다. 프로그램적으로 생각해보면, 동적 메모리 할당이 되어있는 상태에서 다시 할당하게 되었을 때는, 추가로 동적 메모리 할당이 되는 것이므로, 메모리 누수 문제가 발생하며, 만약 기존에 할당한 Flash Memory 내 데이터가 들어있었다면, 알 수 없는 메모리 공간, 더는 접근할 수 없는 메모리 공간으로 데이터가 유실되는 문제가 발생할 것이다.

그리고 이 프로그램은 엔지니어가 Flash Memory 시뮬레이션하면서 원하는 수치(성능)가 나올 때까지 디버깅하는 것이 프로그램의 존재 목적이라 생각이 들었다. 그래서 프로그램 사용자가 재할당하고자 한다면, 먼저 Flash Memory 내에 데이터 존재 여부를 확인하며, 데이터가 존재한다면, 손실이 있을 수 있음을 사용자에게 통보해준다. 그리고 Sector Write 시 미리 기록한 데이터 길이를 참조해서 텍스트 파일의 형태로 데이터를 백업한다. 그리고 백업이 완료되었다면 기존의 동적 할당된 Flash Memory의 메모리 해제와 사용자가 재할당하고자 하는 저장소의 크기로 할당하면서 재할당 함수가 종료된다.

종료 후 초기에 호출된 할당 함수에서 Flash Memory의 재할당되었음을 감지해, 파일 스트림으로부터 재할당 전 데이터 복원을 마치고 Flash Memory 할당 함수가 종료된다. 이 과정에서는 할당된 섹터를 넘는 데이터가 입력되었을 때, 오류를 방지하기 위해 데이터를 복원하지 않는다. 마지막으로 Sector / Block Mapping Menu에 따라 할당 함수가 종료된 이후에 Sector / Block Mapping Table을 선언하고 사용자에게서 명령 입력을 기다린다.

<Flash Memory 읽기 함수가 호출된다면>



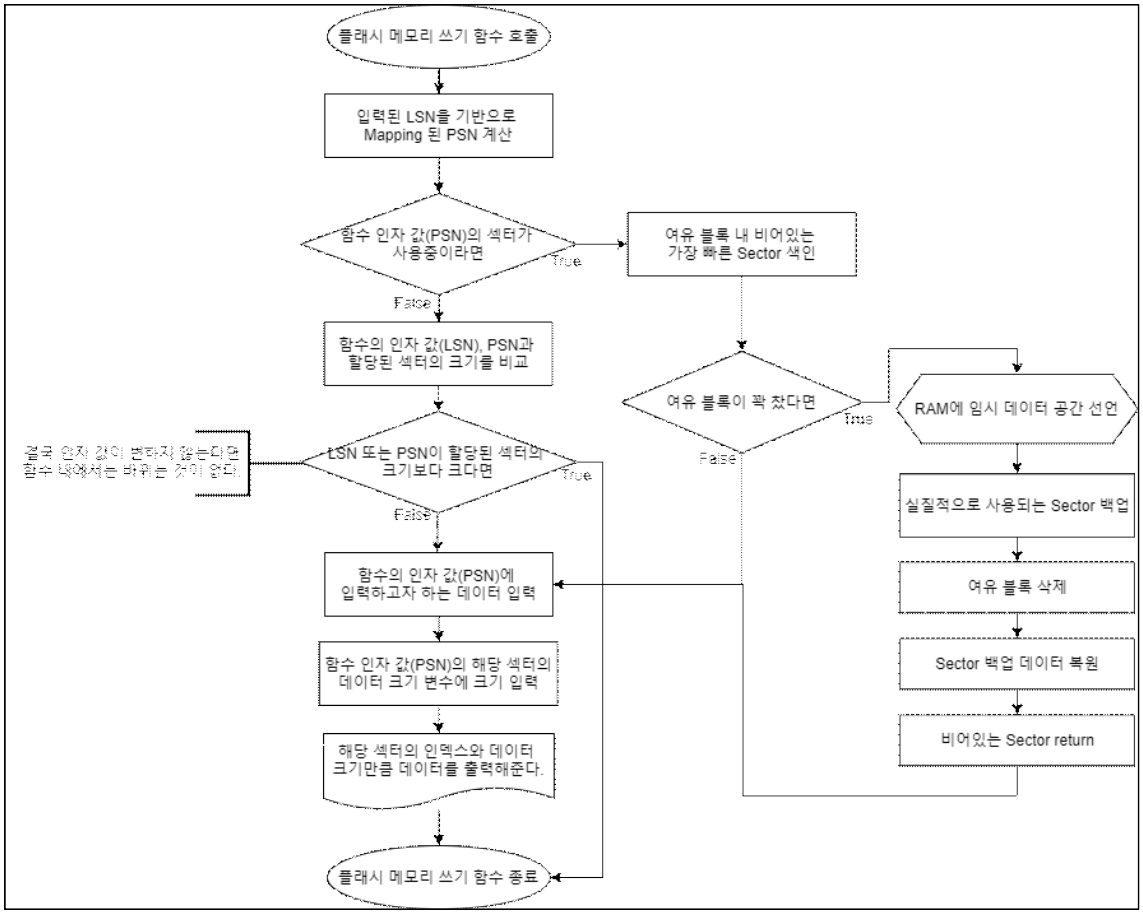
좌 : Sector Mapping 방식에서의 읽기 함수, 우 : Block Mapping 방식에서의 쓰기 함수

데이터를 읽고자 하는 LSN의 입력을 받았다면, 먼저 프로그램에서는 Sector / Block Mapping인지 확인한다. Sector Mapping으로 구성되어 있다면, Sector Mapping Table에 저장된 PSN 값을 기반으로 읽기 함수를 작동한다.

Block Mapping 기반으로 구성되어 있을 때 File System은 이를 인지하지 못하고, LSN을 기반으로 데이터 입출력을 요구하기 때문에, Block Mapping 기반으로 구성된 Flash에 접근하기 위해서 입력받은 LSN을 Table에 접근하기 위한 LBN, 섹터의 위치를 파악할 척도인 offset으로 계산해주는 Solver가 필요하게 된다. 이후 계산된 LBN, offset을 기반으로 PBN을 구하고, PSN을 구하게 된다면, 기존의 PSN 기반으로 직접 읽고 쓰는 함수들에 계산된 PSN을 전달한다면 각 기능을 바로 사용할 수 있게 된다.

그리고 입력된 LSN이 할당된 Sector / Block의 크기보다 크다면, 잘못된 메모리 주소의 접근이 예상되므로 읽기 함수의 작동을 중단할 것이다. 그러나 Mapping 되어있는 PSN은 할당된 여유 블록에 의해 할당된 섹터보다 큰 범위에 데이터를 접근할 경우가 생기므로, 실질적으로 데이터를 읽는 함수에서 PSN의 입력을 받을 때 여유 블록의 크기만큼의 데이터 입력을 허용한다.

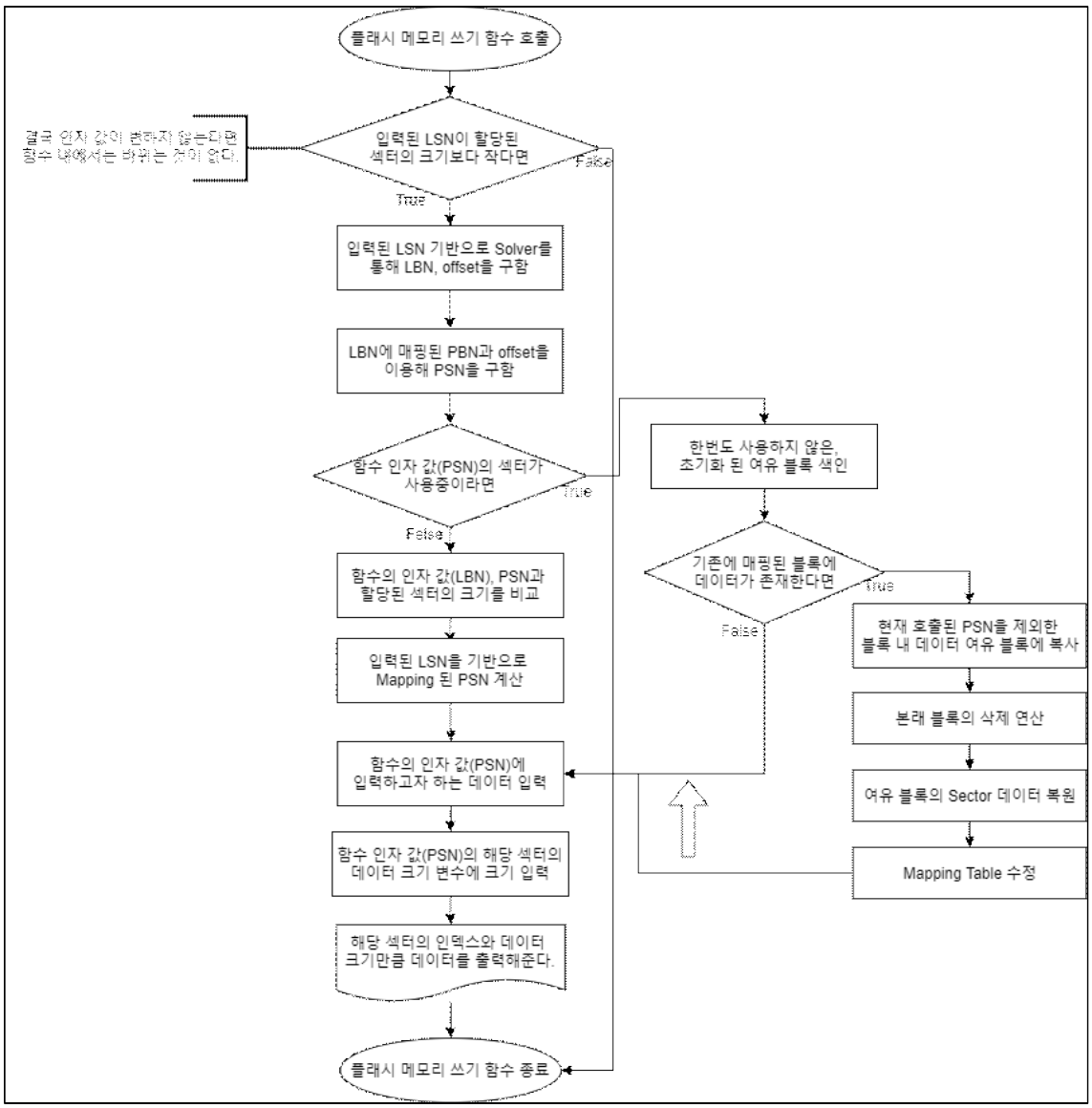
<Flash Memory 쓰기 함수가 호출된다면>
<Sector Mapping Write>



Flash Memory가 할당되고, Sector에 데이터를 쓰고자 할 때, 현재 0-Fill에 의해 어느 Sector에도 데이터가 없는 상태이므로, Mapping Table에 할당된 Sector에 데이터를 쓴다면 아무 문제가 없다. 그러나, 이미 데이터가 들어 있는 LSN(PSN)에 데이터를 쓰고자 한다면 어떻게 처리를 해야 할까? E-B-W 특성에 의해서, Overwrite는 절대 불가능하다. 이러한 이유로 Overwrite 처리를 꼭 해줘야 하는 이유가 있다. 우리가 컴퓨터를 사용하면서, 뭔가 처리한 데이터를 저장해야 할 일이 꼭 생기게 되며, 결국 저장을 위한 마지막 Layer가 비휘발성 메모리 Flash Memory이므로 Overwrite에 대한 예외처리는 필요하다. 그리고 1셀당 3bit를 저장하는 TLC, 4bit를 저장하는 QLC가 지우기 연산을 할 수 있는 횟수는 고작 몇백 회밖에 되지 않으므로, 지우기 연산을 최소화하는 Overwrite 처리를 해야 한다. 이러한 이유로 인해 Overwrite 연산이 호출되었을 때, 여유 블록 내 가장 빠른 Sector를 찾아 해당 블록에 데이터를 저장시키며, Mapping Table을 수정해주면 File System에서 Overwrite 된 것 같이 보이게 되며, 쓰기 함수가 종료된다.

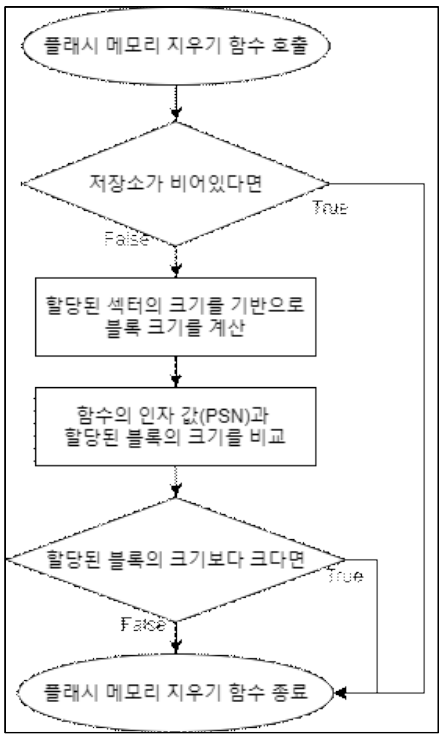
그러나 여유 블록이 꽉 찼을 때 Overwrite가 호출될 때, 실질적으로 사용하는 데이터를 Mapping Table을 이용해서 구분하며, RAM에 백업해둔다. 그리고 실제 속하는 여유 블록을 비워주며, RAM에 백업해둔 Sector 데이터를 복원하면, 지우기 연산이 실행되었지만, 실질적으로 사용하는 데이터만 남겨두고, 여유 블록의 공간을 확보할 수 있겠다.

<Block Mapping Write>



Block Mapping 기반 쓰기 함수가 호출되었을 때, File System에 의해 LSN이 입력되기에, 입력된 LSN이 할당된 Sector의 크기보다 작을 때를 기준으로, 올바른 데이터 범위에 저장하는지 확인한다. PBN, PSN을 구하기 위해서 Block 단위로 저장된 테이블을 이용해 LBN에 해당하는 PBN을 먼저 구하며, 이후 LBN을 이용해 offset을 구해서 PBN을 구할 수 있다. 그리고 해당 PSN이 비어있어, 처음 저장된다면, 데이터를 해당 PSN에 기록하고 쓰기 함수를 마치면 되지만, 이미 데이터가 들어 있는 PSN에 쓰기 명령이 내려졌다면, 지우기 연산을 최대한 방지하기 위해서, 블록맵핑의 특성에 따라 Overwrite 연산이 입력되었을 때, PSN이 속한 블록의 데이터를 여유 블록에 저장하며, 동시에 기존의 블록을 삭제한다. 그리고 Mapping Table을 여유 블록을 칭하도록 수정한다. 수정되었다면, Sector가 비어있을 때와 같이 여유 블록 데이터 쓰기 작업을 시행하고 쓰기 함수를 종료한다.

<Flash Memory 블록 지우기 함수가 호출된다면>



조금의 데이터 블록들을 관리하는 것이기에 데이터가 들어있는지, 정상적인 데이터가 들어있는 것인지, 직접 확인이 가능하지만, 만약 현재 상용화되어 있는 Flash Memory SSD처럼 1TB와 같은 용량을 가진 저장소를 일일이 확인하고 지우는 것을 불가능하다고 생각이 들었다. 그리고 추후 FTL 알고리즘을 이용해 Flash Memory를 관리하고자 할 때, 엔지니어가 정한 특정 조건에 따라, 필요 없는 데이터를 Block 내에서 색인하고, 지워야 할 일이 생길 것이기에, 블록 단위로 제거하는 함수를 따로 만들어 입력된 블록 번호 기반으로 실질적인 블록 제거를 할 수 있어야 한다고 생각이 들었다. 블록 제거 함수에서는 기본적으로 할당된 공간 내에 입력된 블록이 속하는지 확인하며, 속한다면 해당 블록을 지우면서 함수가 종료된다.

3. 프로그램 실제 작성

Flash Memory를 구현하는 프로그램의 목적과 요구 점을 생각해봤을 때 엔지니어가 FTL 알고리즘을 설계하거나, 리버스 엔지니어링 하기 위해 사용하는 프로그램이라 생각이 들어, 저장되는 데이터 무결성을 중점으로 프로그램을 작성하였다.

<프로그램 헤더 부분>

```
#include <iostream>
#include <string>

//for debugging, reallocation
#include <fstream>
#include <vector>
```

기본적인 C++의 헤더 iostream을 사용해서 기본적인 기능을 구현하였으며, 입력받은 데이터, 명령을 관리하기 위한 string 헤더, 파일 입출력을 위한 fstream, 그리고 Mapping Table을 위한 vector 헤더를 사용하였다.

<Flash Memory 구조>

```
9   class Flash
10  {
11  public:
12      char Sector[512]; // 512 + 1 byte, Pointer == (Variable *= 4)
13      int Valid_Check = 0; // length of data?
14      Flash()
15      {
16          for (int i = 0; i < 512; i++)
17          {
18              Sector[i] = 0; //init data allocation, 0 Fill
19          }
20      }
```

프로그램에서 구성되는 1개의 Sector는 (512 + 1) + 4byte(Sector Length)의 용량으로 512byte의 Sector를 구현한다. 그리고 우리가 Flash Memory를 처음 구매했을 때와 같은 상태를 만들어주기 위해서, 초기 선언 시 생성자를 통해 Sector 내 모든 데이터를 0으로 채운다(0-Fill). 이 과정은 컴퓨터에서 포맷할 때의 과정과 같다. (흔히 말하는 Low-level 포맷을 말함)

<프로그램 실행 부분, Pointer Variable>

```
int main()
{
    Flash* data;
    vector<int> table;
    Select_Func(data, table);
}
```

프로그램 실행 시, Flash Memory가 할당되고, 각 기능에서 같은 메모리 공간을 액세스하게 하려면, 메인 함수에 Flash 객체의 Pointer, Mapping Table 객체를 선언하였다. 이후 미리 만들어둔 Sector / Block 선택 함수를 실행한다.

<Sector / Block 선택, 기능 실행 함수>

```
void Select_Func(Flash*& o1, vector<int>& o2) //동시에 1개의 에뮬레이션만 가능하기에
{
    string Input1 = "";
    while (true)
    {
        cout << "FTL Mapping 방식을 선택하세요. (sector, block, exit)" << endl;
        Input_Func(Input1, 6);
        Exit_Func(Input1);
        if (Input1 == "sector" || Input1 == "Sector" || Input1 == "SECTOR")
```

```
void Sec_Func(Flash*& o1, vector<int>& o2)
{
    string Input1 = "", Input2 = "", Input3 = ""; //write data!!
    while (Input1 != "exit" && Input1 != "Exit" && Input1 != "EXIT")
    {
        cout << "시뮬레이션 하고 싶은 명령을 내려주세요. (init, read, write, table, exit)" << endl;
        Input_Func(Input1, 6); //maximum length set
        if (Input1 == "init" || Input1 == "Init" || Input1 == "INIT") //if init + Eng Input?? => Error
```

Sector / Block Mapping마다 데이터의 저장 방식이 달라지므로 이번 프로그램에서는 Sector / Block Mapping 알고리즘을 바꾸려고 할 때마다 각 메뉴에서 할당된 Flash Memory 공간, 저장된 데이터, 그리고 Mapping Table을 초기화시킨다. 그리고 각 기능이 전환되었다면, 처음 실행시킨 것처럼 기능 작동을 하게 된다.

여기서 든 생각은 이 프로그램은 비휘발성 메모리인 Flash Memory를 구현해보는 것이 목적이므로, 각 기능 전환이 되기 전에, 현재 가지고 있는 데이터, Table, 섹터 크기를 파일로 저장해둔다. 그리고 각 기능이 전환되었을 때, 마지막 실행 시 가지고 있던 데이터들을 불러와서, 기존에 작업하던 데이터를 그대로 사용할 수 있게 하였다.

데이터, 섹터 크기, Mapping Table 저장 정보를 모두 불러왔다면, 텍스트 형태의 명령을 받을 준비를 한다. 명령이 입력되었을 때, 입력 함수에서 제대로 된 입력인지 확인하고 각 기능 함수에 전달해준다.

<공간 할당 함수, 동적 배열 할당>

```
void init(Flash*& o1, vector<int>& o2, const string& o3, int& v1)
{
    system("cls");
    bool temp = false;
    if (pos) //if allocated
    {
        cout << "이미 Flash Memory가 할당되어 있습니다. 재할당하시겠습니까? (Y / N)"
```

```
    //real init, dynamic allocation
    o1 = new Flash[2048 * v1 + 32]; //dynamic array allocation, spare block
    secsize = v1 * 2048; //secsize / 32 == blksize
    Mapping_init(o2, o3);
```

그리고 위의 기능 실행 함수에서 명령 “init”과 함께 할당하고자 하는 Flash Memory 크기를 입력받았다면, init 함수에 전달되게 되며, 초기에는 Flash Memory가 할당되어있는지 확인한다. 만약 이 부분을 확인하지 않고, 입력되는 명령만큼 계속 할당을 하게 된다면, 동적 할당으로 인해 메모리 누수가 발생하게 될 것이며, 기존 메모리 공간에서 가지고 있는 데이터는 더는 접근할 수 없는 메모리 공간에 들어있게 되어, 데이터 유실이 발생하게 된다. 여기서 재할당을 하고자 한다면, 기존의 Sector에서 가지고 있는 데이터만 임시로 파일에 복사해두며, 메모리 해제, 메모리 할당, Sector 데이터를 파일로부터 복원하고 할당을 마치게 된다.

이후에 해당 프로그램이 FTL이라는 계층을 만들어서 블록 내 I/O를 결정하도록 설계하였기에, Flash Memory 할당이 끝났다면, Mapping Table을 호출된 메뉴에 따라 Sector / Block에 맞는 Mapping Table을 구성해주면서 정말 할당 함수가 종료된다.

<Sector 읽기 함수, 올바른 데이터 접근, 데이터 길이(무결성) 체크>

```
void FTL_read(Flash*& o1, vector<int>& o2, int& v1, const string& o3)
{
    if (o3 == "sector" && pos && v1 < secsize) //input limit
    {
        Flash_read(o1, o2.at(v1));
    }
    else if (o3 == "block" && pos && v1 < secsize) //need Trusted Solver
    {
        int lbn = 0, offset = 0, psn = 0;
        offset_calc(v1, lbn, offset); //lsn, lbn, offset
        psn = o2.at(lbn) * 32 + offset;
        Flash_read(o1, psn); //Flash, psn
```

Flash Memory에서의 데이터 읽기 단위는 Sector이므로, Sector 단위로 데이터 읽기를 구분한다는 것을 유추할 수 있다. 그리고 이전에 만든 프로그램이 PSN 기반으로 데이터를 읽도록 작성하였기 때문에, Sector 메뉴에서 “read” 명령이 들어왔다면, 입력된 LSN을 이용해 PSN을 바로 구하고, Block 메뉴에서 명령이 들어왔다면, 저장된 Mapping 방식이 다르기에, LBN, offset를 구해서 PBN, PSN을 구하게 된다면, Block Mapping 기반 Flash Memory에서 데이터를 읽을 수 있다.

<Sector 쓰기 함수, 올바른 데이터 접근, 데이터 길이 기록>

```
void FTL_write(Flash*& o1, vector<int>& o2, int& v1, const string& o3, const string& o4)
{
    if (o4 == "sector" && pos && v1 < secsize)
    {
        Flash_write(o1, o2, o2.at(v1), o3, o4); //Flash, Table, psn, data, sector
    }
    else if (o4 == "block" && pos && v1 < secsize) //need Trusted Solver
    {
        int lbn = 0, offset = 0, psn = 0;
        offset_calc(v1, lbn, offset); //lsn, lbn, offset
        psn = o2.at(lbn) * 32 + offset;
        Flash_write(o1, o2, psn, o3, o4); //Flash, psn, data, block
```

그리고 명령 함수로부터 "write" 명령을 받았다면, Sector / Block의 입력을 구분해서, 만약 Sector라면, Mapping Table을 통해 바로 PSN을 받아올 수 있으므로, 바로 write 함수를 작동하였지만, Block 메뉴에서 작동하였다면, LSN을 이용해서 LBN, offset, PBN, PSN을 계산해서 write 함수에 전달한다. 그리고 Block 할당임을 알려주며 실질적인 쓰기 함수의 작동이 시작된다.

<Flash_write Sector / Block 작동부>

```
void Flash_write(Flash*& o1, vector<int>& o2, int& v1, const string& o3, const string& o4)
{
else if (o4 == "sector" && pos && v1 < secsize + 32)
{
    int spareblk = secsize / 32;
    if (o1[v1].Valid_Check != 0) //sector is Full
    {
        cout << "비어있는 가장 빠른 Sector에 데이터를 저장합니다. " << endl;
        //Indexing Func..
        SecIndexing_Func(o1, v1); //입력된 index 기반으로 처리하면 빠른 처리가 가능할 것!
        //spare block isFull
        if (Block_isFull(o1, spareblk)) //마지막 인덱스가 사용중일 것이므로, 마지막 인덱스의 초기화가 안된다...!
```

일반적인 데이터 쓰기 작업은 전달된 PSN과 데이터를 바로 기록하고 끝나게 되지만, Overwrite 연산이 발생하게 된다면, E-B-W 구조에 의해, 실질적으로는 Overwrite가 불가능하다. 그러나 여기서 Overwrite를 위해 매번 블록을 지우는 것은 수명에 상당히 치명적이므로, 여유 블록에 빈 곳이 있는지 확인하며, 빈 곳에 우선 데이터를 저장시킨다. 이후 Mapping Table의 값을 방금 저장한 섹터를 칭해주면, 사용자는 Overwrite 되었다고 인지하게 된다. 이렇게 지우기 연산을 줄였다. 그러나 여유 블록이 꽉 찼다면 어떻게 해야 할까? 우선 여유 블록 내 실질적으로 사용 중인 데이터만 RAM에 복사하고, 여유 블록을 지운다. 그리고 저장해뒀던 데이터를 복원하고 여유 블록의 첫 섹터로 Mapping 후 데이터를 쓰도록 한다면, Sector에서의 쓰기 함수는 종료된다.

```
else if (o4 == "block" && pos && v1 < secsize + 32) //at spare block
{
    int pbn = 0, lbn = 0, offset = 0, spareblk = secsize / 32;
    reverse_calc(o2, v1, pbn, lbn, offset);
    if (o1[v1].Valid_Check != 0) //sector is Full
    {
        cout << "여분의 블록에 데이터를 저장합니다. " << endl;
        //spare block isFull
        if (Block_isEmpty(o1, spareblk))
        {
            BlkExist_check(o1, pbn, lbn, offset, spareblk);
            Block_Erase(o1, pbn); //erase_index = spareblk;
            o2.at(lbn) = spareblk;
```

해당 함수에서 Block Mapping 쓰기 명령을 받았을 때 문제가 있다. 먼저 PSN을 기반으로 입력되므로, 처음 쓰기 작업에서는 문제가 발생하지 않는다. 두 번째 쓰기 작업이 발생하였을 때, Overwrite 연산이 발생하였다는 것을 감지할 수는 있지만, Block Mapping Table의 수정을 하지 못하기 때문에, 실질적으로 할 수 있는 것이 없다. 그래서 reverse_calc이라는 함수를 만들었으며, 이 함수에 PSN을 입력해서, 먼저 PBN, LBN, offset을 구한다. 그리고 여유 블록이 비어있는지 확인하며, 만약 비어있다면, 기존의 PBN에 속한 데이터를 모두 여유 블록에 복사하고, 기존의 블록을 삭제한다. 그리고 Mapping Table을 여유 블록으로 수정해주면, Overwrite와 같은 결과를 보여준다.

그리고 다시 같은 위치의 LSN, LBN에 Overwrite 연산이 발생한다면, 본래 가지고 있던 블록이 비어있고. Mapping 되어있지 않으므로, 비어있는 본래의 블록에 실질적으로 사용하는 데이터를 복사하고, 그리고 Mapping Table의 재수정을 해주게 된다면, 본래의 블록에 Overwrite 데이터를 복사해줄 수 있게 된다.

처음에는 여유 블록뿐만 아니라 비어있는 블록들에 순차적으로 Overwrite를 발생시켜서 웨어 레벨링을 조금 구현해보려고 했는데, 실질적으로 이 블록의 수명의 척도를 기록하고, 구분하며 다시 그것을 기반으로 Mapping Table을 구성하기가 까다로워 구현해보지 못했다. 쓰기 함수에서 이것을 충족하지 못한 것이 아쉽다는 생각이 든다.

<쓰기 함수로부터 호출된 PBN 삭제>

```
void Flash_erase(Flash*& o1, int& v1) //v1 is blk index
else if (pos && v1 < (secsize / 32))
{
    Block_Erase(o1, v1);
    cout << "Block " << v1 << "의 삭제가 완료되었습니다." << endl;
    erase_index = v1;
}
void Block_Erase(Flash*& o1, int& v1) //v1 is blk index 0-31, 32-63, 64-95
{
    ++erase;
    erase_index = v1;
    int start = v1 * 32;
    int end = (v1 + 1) * 32; // end 전까지
    while (start < end)
    {
        memset(o1[start].Sector, '\0', 512); //0-511 Data Clear
```

명령을 받는 함수에서 직접적으로 Block을 지우는 명령을 받지는 않지만, 기존 저장된 데이터에 Overwrite가 계속 발생하게 된다면, FTL 계층을 통해 어느 정도 Overwrite 연산을 대체할 수는 있지만, 물리적으로, 그리고 근본적으로 Flash Memory는 Overwrite가 불가능해서, 언젠가는 Block을 지워줘야 하는 일이 발생하게 된다. 그 경우 쓰기 함수에서 호출되는 블록 값(PBN)이 할당된 블록 내에 속하는지 확인하며, 이후 실질적인 블록 삭제 함수가 실행된다. 결국, 프로그램으로 Flash Memory의 특성을 에뮬레이트하는 것이므로, Flash Memory와 아주 같은 방식으로 데이터를 지우는 것은 아니지만, 해당 Block 내 속한 32개의 Sector를 비우고 데이터 길이를 기록한 변수까지 지워주면, 특정 블록이 삭제된 것처럼 보이게 된다.

<비휘발성 메모리와 같이 보이도록 자동으로 섹터 데이터를 저장하는 함수 >

```
void TableSave_func(vector<int>& o1, const string& o2, const string& o3)
{
    ofstream ofs, ofs_size;
    ofs.open(o2);
    ofs_size.open(o3);
    vector<int>::reverse_iterator iter;
    for (iter = o1.rbegin(); iter != o1.rend(); iter++)
    {
        ofs << o1.at(iter - o1.rbegin()) << " " << "\n";
    }

    ofs_size << secsize << " " << erase_index << " " << "\n";
```

```
void SecSave_Func(Flash*& o1, const string& o2)
{
    if (!pos)
    {
        cout << "Flash Memory 할당이 되어있지 않습니다. 확인 후 재실행해주세요. " << endl;
        return;
    }
    else
    {
        ofstream s_ofs;
        s_ofs.open(o2);
        for (int i = 0; i < secsize; i++)  //size != 0 saving!!, secsize exist
        {
            if (o1[i].Valid_Check) //data is Full
```

명령 입력 함수에서 다른 Mapping 알고리즘으로 바꾸고자 하거나, 프로그램을 종료하고자 하면 Sector 데이터, 크기, Table의 내용을 자동으로 저장해서, 추후 디버깅을 유용하게 하였으며, 프로그램 내에서 작업하던 내용을 다시 입력해서 시간을 빼앗길 필요 없이 엔지니어링에만 집중하도록 하였다. 그리고 파일에 출력할 때, Sector 내에 실제로 존재하는 데이터만 백업해서 할당된 메모리 용량보다 작은 크기의 데이터 저장이 가능하다.

<프로그램 종료 후 재실행 시에 자동으로 섹터 데이터를 불러오는 함수>

```
void TableLoad_func(vector<int>& o1, const string& o2, const string& o3)
{
    ifstream ifs, ifs_size;
    ifs.open(o2);
    ifs_size.open(o3);
    if (ifs.is_open())
    {
        int Temp = 0;
        while (ifs >> Temp)
```

```
void SecLoad_Func(Flash*& o1, const string& o2)
{
    if (!pos)
    {
        cout << "Flash Memory 할당이 되어있지 않습니다. 확인 후 재실행해주세요. " << endl;
        return;
    }
    else
    {
        ifstream s_ifs;
        s_ifs.open(o2);
        if (s_ifs.is_open())
```

프로그램 실행 시, 사용하고자 하는 알고리즘을 선택할 때, 백업 데이터가 존재하는 경우, 백업 데이터에 저장된 Sector 크기에 맞춰 자동으로 Flash Memory를 할당한다.

이를 통해 데이터를 받을 준비를 하며, Table에 Mapping 된 Sector / Block을 모두 복원해준다. 마지막으로 파일로부터 Sector 데이터를 가져오게 되며, 사용자는 실제 Flash Memory를 사용하는 것과 같은 느낌을 받을 수 있게 된다.

<전체 코드 캡처>

```
#include <iostream>
#include <string>

//for debugging, reallocation
#include <fstream>
#include <vector>

using namespace std;

class Flash { ... };

1  void Sector_Frame() { ... }
   void SectorData_Frame() { ... }
   void SectorMap_Frame() { ... }
   void BlockMap_Frame() { ... }

2  Flash* pos, * Spare; // 8 byte + 8 byte
   int secsize = 0, read = 0, write = 0, erase = 0, erase_index = 0, gavage = 0; // 4 + 4 + 4 + 4 + 4 + 4 byte

   void IO_Check() { ... }
3
   void Saved_Clear() { ... }


   void Input_Func(string& o1, const unsigned int& v1) { ... }

4  int Conv_Func(string& o1) { ... }

   void Exit_Func(const string& o1) { ... }

   void SecSave_Func(Flash*& o1, const string& o2) { ... }
5  void SecLoad_Func(Flash*& o1, const string& o2) { ... }

   void TableSave_func(vector<int>& o1, const string& o2, const string& o3) { ... }
   void TableLoad_func(vector<int>& o1, const string& o2, const string& o3) { ... }

6  void offset_calc(int& v1, int& v2, int& v3) { ... }
   void reverse_calc(vector<int>& o1, int& v1, int& v2, int& v3, int& v4) { ... }

7  void Sector_read(Flash*& o1, int& v1) { ... }
   void Sector_write(Flash*& o1, int& v1, const string& o2) { ... }

8  bool Block_isEmpty(Flash*& o1, int& v1) { ... }
   bool Block_isFull(Flash*& o1, int& v1) { ... }
   void SecIndexing_Func(Flash*& o1, int& v1) { ... }
```

9
```
void SecExist_check(Flash*& o1, vector<int>& o2, int& v1){ ... }
void BlkExist_check(Flash*& o1, int& v1, int& v2, int& v3, int& v4){ ... }
```

10
```
void Block_Erase(Flash*& o1, int& v1){ ... }
void Restore_Block(Flash*& o1, int& v1){ ... }
```

11
```
void Gavage_erase(Flash*& o1, vector<int>& o2){ ... }
```

12
```
void Mapping_init(vector<int>& o1, const string& o2){ ... }
void init(Flash*& o1, vector<int>& o2, const string& o3, int& v1){ ... }
```

```
void Flash_read(Flash*& o1, int& v1){ ... }

void Flash_write(Flash*& o1, vector<int>& o2, int& v1, const string& o3, const string& o4){ ... }

void Flash_erase(Flash*& o1, int& v1){ ... }
```

13
```
void Print_Table(vector<int>& o1, const string& o2){ ... }

void FTL_read(Flash*& o1, vector<int>& o2, int& v1, const string& o3){ ... }

void FTL_write(Flash*& o1, vector<int>& o2, int& v1, const string& o3, const string& o4){ ... }
```

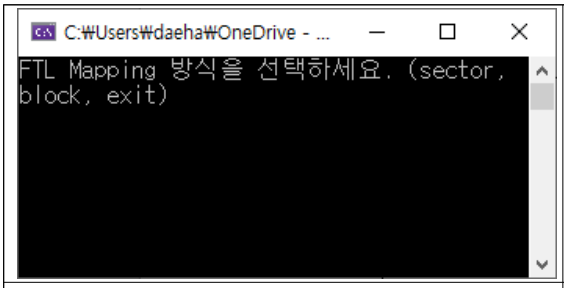
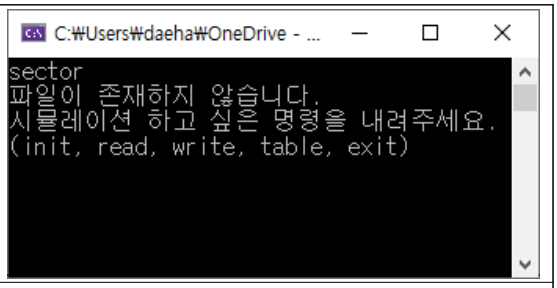
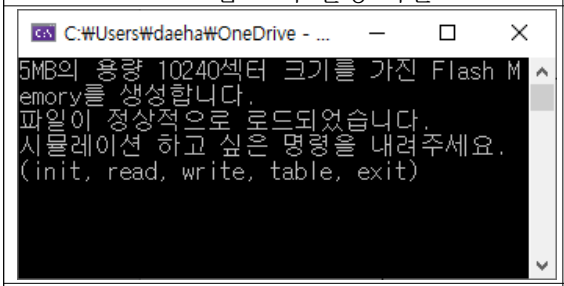
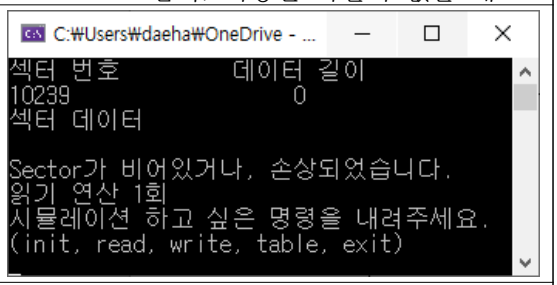
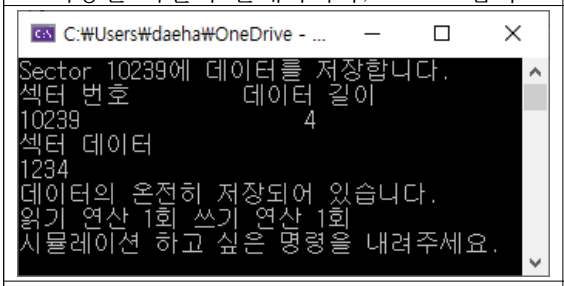
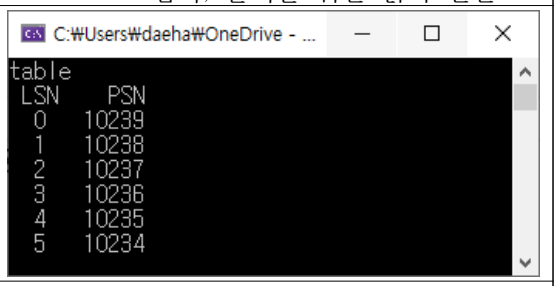
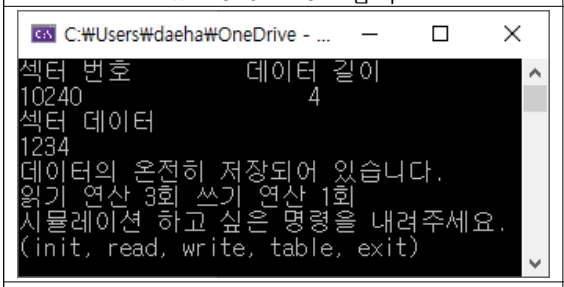
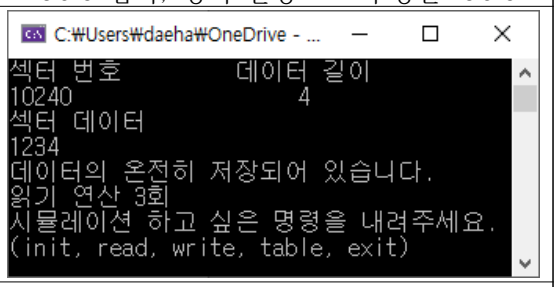
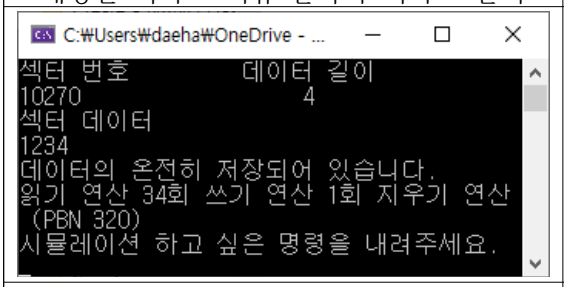
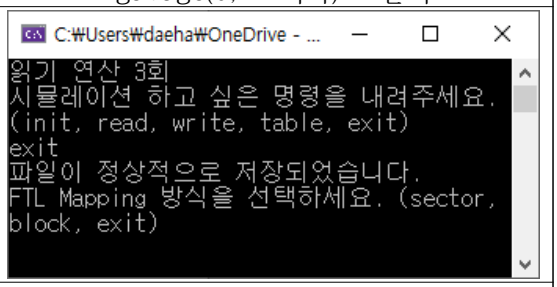
14
```
void Sec_Func(Flash*& o1, vector<int>& o2){ ... }
void Blk_Func(Flash*& o1, vector<int>& o2){ ... }

void Select_Func(Flash*& o1, vector<int>& o2){ ... }

int main(){ ... }
```

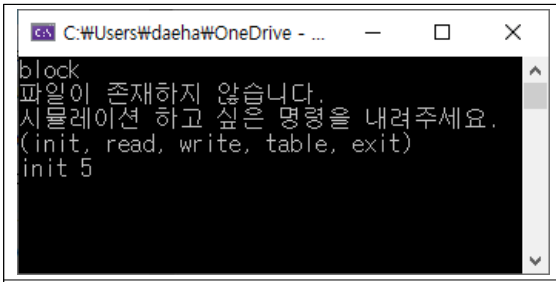
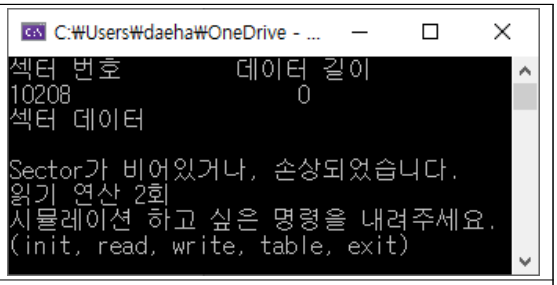
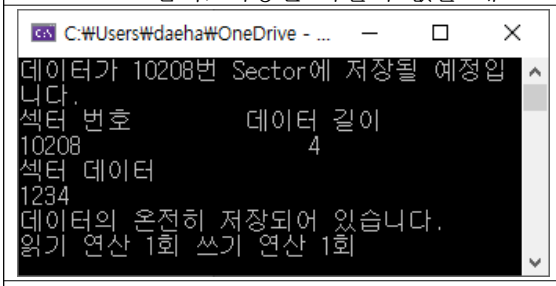
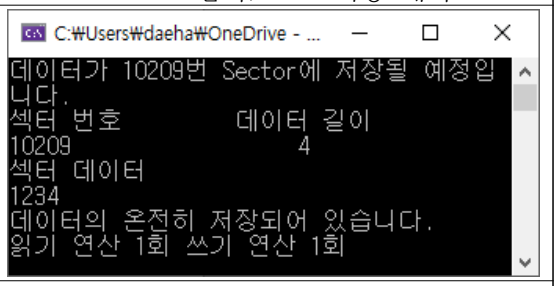
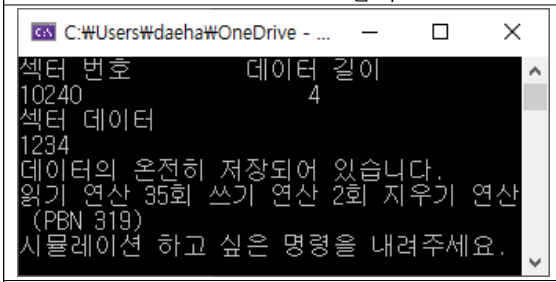
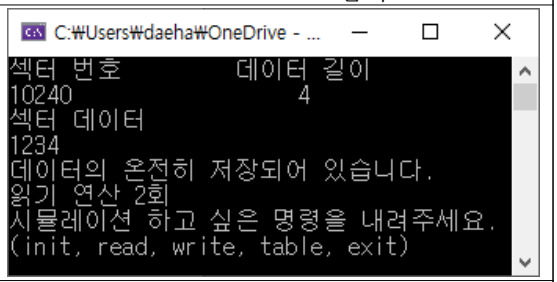
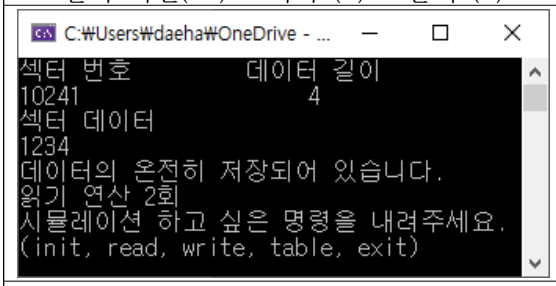
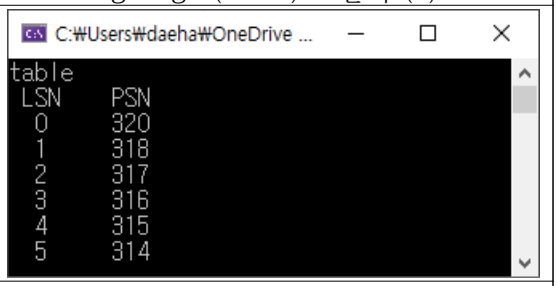
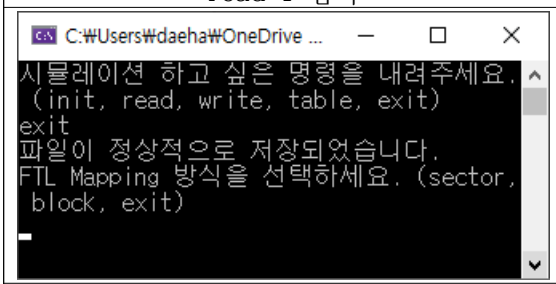
1. 데이터 출력 전 사용자에게 데이터 구분을 위한 출력함수
   (1) Sector Index, 데이터 길이임을 알려주는 부분
   (2) Sector Data임을 출력해주는 부분
   (3) Sector Mapping Table임을 출력
   (4) Block Mapping Table임을 출력
2. 각 함수가 종료되어도 데이터를 저장할 곳이 필요해서 전역변수 선언
   (1) 할당된 Flash 영역, 추후 섹터 맵핑에서 새로 할당할 Spare 영역
   (2) STL로 작성하지 않아 Sector size 보관, read, write, erase의 count를 위한 변수
3. 전역변수 관리 함수
   (1) 저장된 read, write, erase를 이용해 데이터가 존재한다면, 횟수를 출력해주는 함수
   (2) 모든 기능이 종료될 때 전역변수 초기화, Spare 공간 메모리 해제
4. 데이터 입력 관리 함수
   (1) 입력된 숫자만큼 데이터 입력받는 함수, 입력의 오류를 찾아 계속 재입력받음
   (2) 문자열 String을 Integer 형으로 Casting 해주는 함수
   (3) 입력된 문자열이 exit이라면, 메모리 할당 해제 후 프로그램 종료
5. Sector, Table을 텍스트 파일 형태로 저장하고, 로드하는 함수
6. Trusted Solver
   (1) LSN을 LBN, PBN, offset으로 계산해주는 함수
   (2) PSN을 PBN, LBN, offset으로 계산해주는 함수

7. 입력받은 PSN을 기반으로 R/W하는 함수
8. 비 STL 기반 객체를 위한 함수
   (1) 입력된 PBN을 기반으로 블록이 비었는지 확인
   (2) 입력된 PBN을 기반으로 블록이 꽉 찼는지 확인
   (3) 섹터 맵핑에서 여유 블록 내 비어있는 섹터 매칭시키는 함수
9. 블록을 비우거나 옮겨야 할 때 사용하는 데이터가 있는지 색인하는 함수
   (1) 섹터 기반에서 실제 사용하는 데이터를 Spare 공간을 새로 선언하여, 백업해둔다.
   (2) 블록 기반에서 pbn, lbn, offset, dest blk 모두 입력받아, 사용하는 데이터만 목적지 블록에 복사해준다.
10. 블록을 삭제하는 함수, 섹터 기반에서 Spare 공간에 데이터가 있을 때 배정받은 블록에 데이터 복원하는 함수
11. 쓰기 함수를 제외한 모든 함수가 실행될 때 같이 실행되는 함수로 어느 기능이 1회 실행되었을 때 0번 블록을 색인해서, 블록이 차 있는지, 필요 없는 데이터가 있는지 확인해서 쓰레기 데이터를 비워준다. 이후 다음 실행에서는 1번 블록을 색인하게 된다.
12. 모든 기능이 실행되기 전에 제일 먼저 실행하는 함수
   (1) init 함수가 실행되고, 테이블 구성을 위해 호출되는 함수
   (2) Flash Memory 할당을 위해 사용되는 함수
13. 표면적으로 Flash에 액세스하는 함수
   (1) FTL_read에 의해 전달되는 PSN에 의해 작동되는 함수
   (2) FTL_write에 의해 전달되는 PSN에 의해 작동되는 함수
   (3) Flash_write에 의해 호출되는 블록 지우기 함수
   (4) Mapping Table을 출력해주는 함수
   (5) 명령 함수로부터 호출되는 read 함수, LSN을 해석해서 실질적 읽기 함수에 전달해 줌
   (6) 명령 함수로부터 호출되는 write 함수, LSN을 해석해서 쓰기 함수에 전달해 줌
14. Sector / Block에 맞게 각 기능 함수를 호출해주는 함수
   (1) Select_Func로부터 Sector 명령받았을 때 실행하는 함수, 기능 입력이 목적
   (2) Select_Func로부터 Block 명령받았을 때 실행하는 함수, 또한 기능 입력이 목적
   (3) 메인 함수에서 Flash, Table이 선언되고 제일 먼저 실행되는 함수, 기능 입력이 목적

4. 프로그램 실행 화면

<테스트 결과>

<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>프로그램 초기 실행 화면</td><td>Sector 입력, 저장된 파일이 없을 때</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>저장된 파일이 존재하거나, init 5 입력</td><td>read 0 입력, 출력을 위한 읽기 연산</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>write 0 1234 입력</td><td>table 입력, 정적 할당으로 구성된 table</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>write 0 1234 재입력,
매핑된 섹터 + 여유 블록의 섹터 + 출력</td><td>read 0 입력
gavage(0, 1 섹터) + 출력</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>블록이 꽉 찬 상태에서, read, table 호출</td><td>exit 입력 후 초기 화면으로 복귀</td></tr>
</table>

<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>block 입력, 저장된 파일이 없을 때</td><td>read 0 입력, LSN 자동 해석</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>write 0 1234 입력</td><td>write 1 1234 입력</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>write 0 1234 재입력,
블록 확인(32) + 복사 (2) + 출력 (1)</td><td>read 0 입력,
gavage (blk 0) + 출력 (1)</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>read 1 입력</td><td>table 입력</td></tr>
<tr><td> 
</td><td></td></tr>
<tr><td>exit 입력, 파일 저장</td><td></td></tr>
</table>

<프로그램 작성 후 결과 분석>

Flash Memory를 구현하는 프로그램에서 특별한 계층(FTL)을 이용해 데이터 읽기, 쓰기를 하는 프로그램을 작성하였다. FTL의 구조를 가지는 Table과 이를 이어주는 FTL_read, write를 기존의 read, write 대신 사용하도록 해서, File System 내에서 명령을 받은 것처럼 구현한 프로그램이다.

먼저 내가 작성한 프로그램의 문제점부터 설명하겠다.

문제 1. 블록을 골고루 마모시키지 못하는 문제

Sector 매핑, 그리고 Block 매핑에서 처음 데이터를 쓰고자 하면 문제가 없으므로, Sector 매핑에서 Overwrite 하고자 한다면, 여유 블록에 데이터를 쓸 준비를 한다. 나는 바로 이것이 문제라고 생각한다. SLC, MLC, TLC, QLC 등 각 NAND가 가지는 수명을 떠나서 여유 블록도 Flash Memory의 일종이므로 계속 Overwrite의 발생으로 데이터를 쓰고, 비운다면, 결국 여유 블록의 수명이 모두 닳아서 제 기능을 하지 못하게 된다. 그래서 사용자가 사용 가능한 공간이 줄어들게 될 것이다.

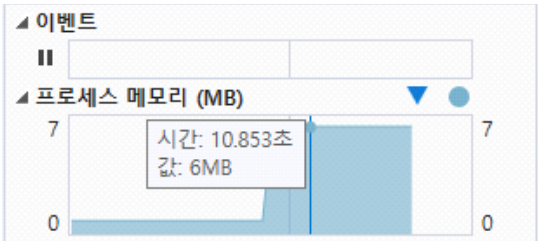
일반적인 Block 매핑에서 더 크게 이 문제가 발생하게 되는데, Sector 매핑처럼 각 섹터의 매핑 값을 바꾸지 못하므로, 블록 내 존재하는 실질적인 데이터만 여유 블록에 복사하고, 매핑 값을 변경시켜, Overwrite 데이터를 받아야 한다. 그리고 같은 블록 내에서 Overwrite가 또 발생한다면, 매핑 된 여유 블록을 비우고, 매핑 되어있지 않은 본래의 블록에 데이터가 저장되게 되어, 블록의 수명이 상당히 떨어지게 된다. 이를 해결하기 위해, 블록의 수명을 기록하는 척도가 필요하다고 생각이 들었으며, Overwrite라고 감지되었다면, Block Mapping을 위한 또 다른 Solver 또는 다른 Table을 더 만들어서, Overwrite 데이터만 여유 블록에 저장되게 할 수 있게 한다면, Flash의 수명은 훨씬 늘어날 것으로 생각이 든다. 그리고 Overwrite가 계속 발생했을 때를 고려해서 특정 수치만큼 Overwrite가 발생한다면, RAM에 저장되게 하고, 특정 시간 후 Overwrite 데이터를 실제 블록에 저장시킨다면, 수명을 늘릴 수 있다는 생각이 들었다.

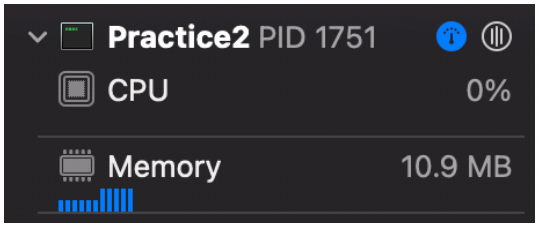
문제 2. Flash가 사용 중일 때 쓰레기 데이터를 지우는 문제

프로그램의 특성상 계속 명령을 기다리고 있는데, 이로 인해 다른 명령을 실행할 수가 없다. 그래서 write 연산을 제외한 모든 기능을 사용할 때, 쓰레기 데이터를 지우게 되어, Access Time이 늘어지게 되며, 사용자는 사용 중인 데이터의 용량이 높을수록 본래 Flash의 성능이 떨어진다고 생각하게 된다. 그리고 내가 작성한 Block 매핑의 경우, Overwrite 시 실질적으로 사용하는 데이터만 복사하고, 이후 블록을 바꿔주게 되어, 사용하지 않는 데이터가 Block 내에 저장될 일이 없지만, Sector 매핑의 경우 Overwrite 발생 시, 여유 블록에 데이터를 적고, 매핑시키므로, 본래의 블록에 사용하지 않는 데이터가 계속 쌓이게 된다. 이는 추후 사용할 수 있는 공간에 대한 낭비라고 생각이 들어, 가득 차 있는 블록을 사용하는 데이터만 RAM에 백업 후 해당 블록을 비워주는데, 동작 전 가득 차 있는 블록 내에 실질적인 데이터의 정도를 알지 못해서, 최악의 경우 모두 사용인 가득 찬 블록의 경우 쓸모없는 지우기 연산을 하게 되어, 수명을 단축하게 된다.

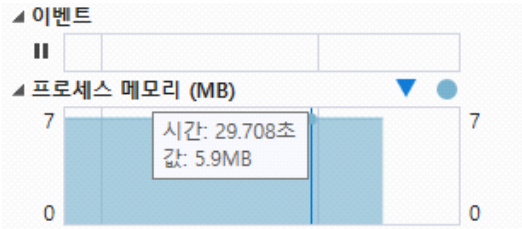
<DRAM>

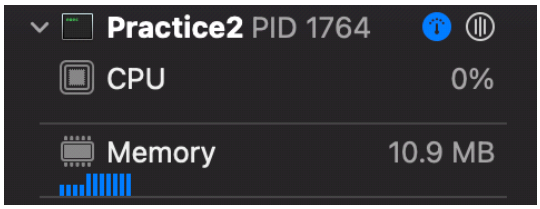
<Sector Mapping에서 init 5 호출 시>





<Block Mapping에서 init 5 호출 시>



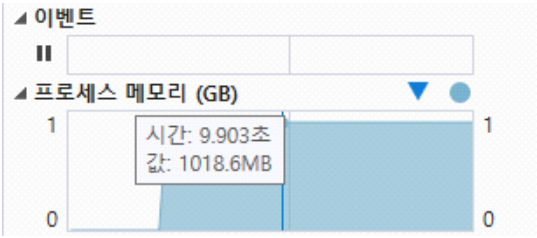


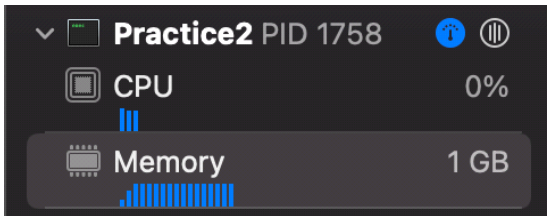
왼쪽의 사진은 Visual Studio에서 측정한 메모리 사용량이며, 오른쪽은 Xcode에서의 메모리 사용량 측정 사진이다. Visual Studio는 실행 초기에 840KB의 메모리를 가졌으며, Xcode는 5.8MB의 메모리를 예약했다. 이는 전역변수로 선언한 6개의 integer 변수가 24Byte, (x86 기준 4Byte) 그리고 메모리 지점을 기록하기 위한 pointer 변수 2개를 사용해서 16Byte의 메모리 차지를 하게 될 것이다. 프로그램의 구동을 위한 메모리 용량을 이론적으로 계산해보면 5MB의 용량 할당과 섹터 매핑 테이블을 할당함으로, Flash Memory에서 5MB를 차지하며, 데이터 길이 보관을 위한 10,240개의 int 변수 선언을 했으므로 40,960Byte를 추가로 사용하게 될 것이며, Table에서는 10,240개의 int 변수를 사용해서 각 섹터를 매핑함으로, 40,960Byte의 메모리 공간이 필요하게 된다. 그리고 만약 데이터가 꽉 차서 램에 여유 블록을 선언하였을 때는 512*32, 16,384Byte, 16KB가 더 사용될 것이다.

5,242,880 + 81,920 + 16,384 + 24 + 16 + 8 = 5,341,232 Byte가 된다. 크게 의미는 없지만, 실제 실행을 위해서 약 927KB의 용량이 더 필요함을 알 수 있다.

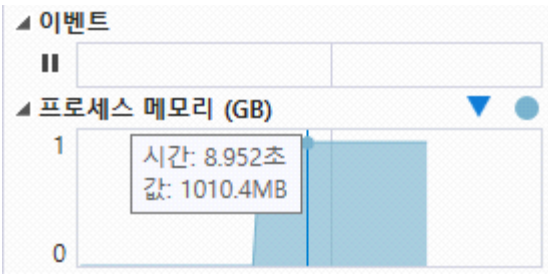
그리고 Block Mapping에서 init 5를 선언하였을 때, 이론적으로 필요한 메모리 용량은 320개의 블록 매핑, 데이터 길이 보관, 여유 블록, 전역변수 선언공간만큼 필요할 것이며, 5,242,880 + 40,960 + 16,384 + 1,280 + 24 + 16 + 8 = 5,301,552 Byte가 될 것이며, Visual Studio에서는 100KB가 줄었다고 나왔지만, 이론적으로 계산했을 때는 Xcode와 같이 Sector Mapping과 Block Mapping과의 사용용량 차이는 약 38KB 정도 발생했다.

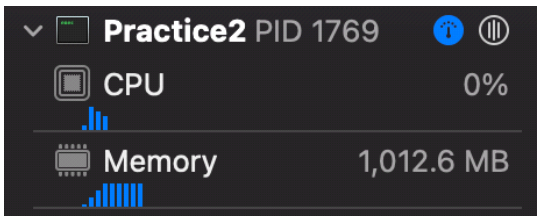
<Sector Mapping에서 init 999 호출 시>





<Block Mapping에서 init 999 호출 시>





앞에서 할당한 5MB의 용량으로는 컴파일러에서 표기해주는 용량 차이가 너무 적어서 999MB를 Sector, Block Mapping으로 할당해보았다.

먼저 Sector Mapping에서는 1,047,527,424(999MB) + 8,183,808(Sec Table) + 8,183,808(데이터 길이 기록) + 16,384(여유 블록) + 24(전역변수) + 16(포인터 변수) + 8(메인 포인터 변수) = 1,063,911,472 Byte가 된다. 약 1,014MB를 뜻하며, 4MB 정도 오버헤드가 발생했음을 알 수 있다.

Block Mapping에서는 1,047,527,424(999MB) + 255,744(Blk Table) + 8,183,808(데이터 길이 기록) + 16,384(여유 블록) + 24(전역변수) + 16(전역 포인터 변수) + 8(메인 포인터 변수) = 1,055,983,408 Byte가 되며 약 1,007MB가 된다. 컴파일러를 통해 약 3MB 정도 오버헤드가 발생했음을 알 수 있다. 실제로 두 값을 비교해보았을 때, 7.7MB의 용량 차이가 발생했음을 알 수 있다. 이 실험 결과를 통해서, Sector Mapping Table과 Block Mapping Table은 구조상 32배의 용량 차이를 가지고 있으므로 큰 용량의 Flash Memory일수록 Block Mapping 방식이 용량 부분에서 훨씬 유리함을 알 수 있었다.

<성능 비교>

공간 할당 직후 Mapping Table의 모든 섹터/블록을 매핑해두는 것을 가정하고 설명하겠다. 데이터의 입력이 순차적으로 들어올 것이라는 보장이 없으며, 또 동시에 들어오게 되어 서로 다른 논리 섹터/블록에 같은 물리 섹터/블록의 매핑을 방지할 것이라는 자신이 없었으며, 만약 발생한다면 이로 인해 많은 연산이 필요하게 되기 때문이다. 만약 특정 논리 섹터의 데이터를 읽고자 할 때 Sector Mapping Table에서 물리 섹터를 찾고자 할 때 5MB라면 최대 10,240개의 섹터를 찾아야 하므로, 블록 기반으로 물리 섹터를 찾는 방법보다 시간이 오래 걸릴 수밖에 없다. 그리고 실제 프로그램 구동 시, 큰 시간 차이는 안 났지만, Block 매핑에서 읽기 함수가 더 빨리 출력되었다. 이로 인해 읽기 시에는 확실히 블록 매핑이 빠르다는 것을 알 수 있다. 그렇다면 쓰기, Overwrite에서는 어떨까? 섹터 매핑에서는 여유 블록이 비어있는지 확인한 다음, 해당 섹터에 데이터를 복사하고, LSN이 매핑하는 값만 바꿔주면 Overwrite가 끝난다. 그러나 일반적인 블록 매핑에서는 Overwrite 발생 시, 사용하지 않는 여유 블록에 매번 기존의 블록이 가지고 있던 데이터를 복사해주고, 또 기존의 블록을 비워줘야 하므로, 쓰기 부분 특히 Overwrite에서는 Sector 매핑이 훨씬 빠르다고 볼 수 있다.

그래서 결론적으로는 읽기에서는 Block, 쓰기에서는 Sector가 빠르다고 볼 수 있다.

<성능 향상하는 방법>

결론적으로 두 알고리즘을 비교했을 때, 저장공간이 32배 더 필요하지만, 섹터 매핑을 이용하면, 수명 중심적인 Flash Memory 설계가 가능하며, 블록 매핑을 사용하면 저장공간을 32배 적게 사용해서 데이터 접근이 가능하다는 장점이 있지만, LSN에 의해 액세스 가능한 섹터가 정해져 있으므로, FTL의 장점을 극대화하기가 어렵다는 단점이 있다.

그래서 기본적으로 Block Mapping이 가진 저용량, 빠른 액세스의 특징을 사용하기 위해서 Block Mapping 형태를 가지면서, Overwrite에 비교적 강한 Hybrid 매핑이 등장하였다고 생각이 든다. Hybrid Mapping에서 Overwrite가 발생하였다면, 본래의 섹터는 사용되지 않는다고 기록하면서 동시에 여유 블록에 데이터와 LSN를 기록하도록 한다. 이후 Overwrite된 LSN을 읽고자 할 때 본래 칭하는 섹터의 데이터가 유효하지 않음을 인식하며, 여유 블록의 데이터 내에서 같은 LSN을 가지며 가장 늦게 수정된 데이터를 읽도록 한다면 최적의 매핑 방법이 될 것으로 생각이 들었다. 또 입력된 Overwrite 데이터의 단위가 Sector일 테니까 Overwrite가 발생하였을 때, Overwrite에 대한 Table을 따로 만들어줘서 이 Table을 통해 여유 블록이나 사용도가 낮은 블록의 섹터를 매핑해준다면, 성능 특히 Access Time이 많이 개선될 것으로 생각이 든다.