[Operating System Assignment Document]

과제 2: FS쿠

201911291 컴퓨터공학부 허준호

운영체제 수업시간 중 CPU, Memory Virtualization, 그리고 Concurrency와 관련한 내용을 배운 후, File System에 관련한 내용들을 배웠다. 이번 과제의 목표는 특정한 Disk나 File System에 의존적이지 않게 공통적으로 구현되어 있는 File System(File Descriptor, File offset 등을 의미하는 것이다)인 Virtual File System보다 Low Level에 위치하는 File System을 간단히 Emulation하는 환경을 응용 프로그램으로서 만들어 보자는 것이었다.
우선, 과제를 진행하기 위하여 Low Level에서의 File System에 대한 이해와 이 File System과 Storage 사이의 관계를 이해하는 것이 우선시 된다고 생각했기에 해당 부분들을 짚어가며 구현에 집중했다. 무엇보다, 저번 과제와 다르게 따로 제공되는 소프트웨어가 없었기에, 과제에서 가정한 사항들을 바탕으로 구현의 방향성을 잡는 데에 많은 시간을 할애했다.

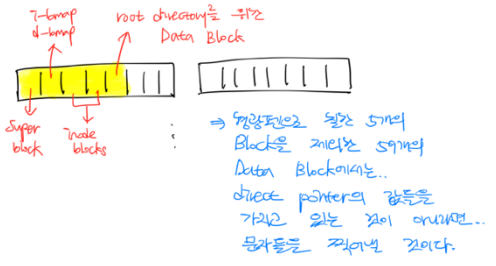
1. 문제 분석 및 구현

이번 과제에서 구현할 FS쿠는 한 Block의 크기가 512bytes이고, 총 64개의 Block으로 이루어진 Partition을 가정한다. 추가적으로, inode에 관련한 정보를 가지고 있는 구조체를 오른쪽 그림과 같이 정의한다. 이러한 Partition을 실제 Disk가 아닌, 메모리 영역을 따로 할당하여 그 영역이 Disk(Storage)라고 가정하고 과제를 진행한다. 이러한 가정을 구현하기 위해 malloc() 함수를 사용하여 동적 메모리 할당을 진행했다. 그런데, 여기서 드는 의문이 하나 있다. 64개의 block들에는 Partition의 MetaData들을 포함하고 있는 Superblock(이번 과제에서는 따로 구현하지 않고, 512bytes의 공간만을 차지하고 있다고 가정함)과, inode들이 현재 할당되어 있는 상태인지의 유무를 확인하기 위한 i-bmap, Data Block들이 현재 할당되어 있는 상태인지의 유무를 확인하기 위한 d-bmap, inode들을 저장하여 담고 있는 inode block, 그리고 실제 Data들을 담고 있는 Data Block들, 이렇게 data type이 다른 정보들이 들어가야 하는데, 하나의 Partition을 메모리 영역에 할당하고 배열처럼 사용하는 경우, 이러한 상황을 구현할 수 있느냐는 것이다. 이는 memcpy() 함수를 통해서 본래 선언되어 있던 배열의 data type과 관계없이 다양한 형태의 type들을 집어넣을 수 있다. 물론, memcpy()라는 함수 자체가 type casting을 해주는 함수는 아니기 때문에 추후에 내가 할당해 놓은 Partition에서 data에 접근하여 사용할 때에는, 적절한 type casting을 통해 데이터에 접근해야 할 것이다. 아래의 코드처럼 말이다.

```
//inode 관련한 정보를 가지고 있는 구조체
typedef struct {
    unsigned int fsize; //이 파일의 크기는 몇 바이트인가?
    unsigned int blocks; //몇 개의 블럭이 이 파일에 할당되어 있는가?
    unsigned int dptr; //Direct Pointer
    unsigned int iptr; //Indirect Pointer
} inode;
```

```
file_inode = ((inode*)overall_Partition[2].data)[file_inum];
```

위 코드는 이번 과제에서 Disk처럼 사용하기 위해 정의하고 메모리 영역을 할당해 놓은 overall_Partition에 접근하여 inode 값을 가져오기 위한 코드이다. overall_Partition이라는 배열이 처음에 정의 되었을 당시 inode* 형태로 정의된 것이 아니므로, inode* 형으로 type casting을 진행하여 file_inode에 값을 넘겨주고 있다. 이번 fsku.c를 구현하며 overall_Partition에 접근할 때마다, 이러한 방식으로 접근하는 형태를 취하고 있다.



이러한 Partition을 구성하는 Block의 기본 type을 그러면 어떻게 정의해야 하는 지가 중요한 문제일 것이다. 이번 과제에서 가장 사용 빈도가 가장 높다고 판단되는 data type으로 설정하는 것이 효율적일 것이라 판단되었다. 왼쪽의 그림을 보면, 공간만을 차지하고 있는 Superblock의 data type은 고려하지 않아도 될 것이고, 비트맵을 통해 inode의 사용 유무, data block의 사용 유무를 나타내는 i-bmap과 d-bmap은 사용 유무만을 비트 단위로 나타낼 것이다. 비트 단위로 표현하므로 Superblock과 마찬가지로 data type을 크게 고려하지 않아도 된다는 결론을 지을 수 있다. Inode를 저장하는 inode block은 inode들을 저장하는 배열 형태로 저장될 것이므로, 따로 정의해 놓은 구조체 inode형 배열이 들어갈 것이다. Data Block 0번은 root directory가 file들의 정보들(file의 inode number, file의 이름)이 table 형태로 들어갈 것이므로 따로 file들의 2가지

정보를 멤버로 가지는 구조체를 따로 정의하여, 그 구조체 형 배열이 들어갈 것이다(추후에도 언급하겠지만, file들의 2가지 정보를 멤버로 가지는 구조체 rootDirectoryInfo를 구현의 편의를 위해서 따로 정의함). 그 이후의 남은 59개의 Data block들에는 file의 inode에서 indirect pointer가 특정한 data block을 가리키며, 그 data block에 direct pointer 값들이 저장되는 경우(이 경우에는 direct pointer의 값들이 배열 형태로 저장되어야 하므로, int형(또는 unsigned int형) 배열이 저장되어야 할 것임)가 아니라면, write 연산의 결과인 char형 배열이 저장될 것이다(이번 과제에서 write 연산은 파일 이름의 첫 번째 알파벳을 명시된 size만큼 적어주는 것 이외에는 수행하지 않는다). 분석 결과, 문자를 Data block에 찍어내는 작업이 Partition 내에서 많은 비중을 차지할 것으로 판단된다. 따라서, Block의 기본 type을 char형 배열로 선언하는 것이 좋을 것으로 판단된다.

내가 구현한 fsku.c에는 기본적으로 선언해야 하는 구조체인 inode를 제외하고, 2가지의 구조체를 추가적으로 선언하였다. 구현의 편의를 위해서 추가적으로 선언한 것이지만, 이번 과제에 사용되는 개념들의 활용을 직관적으로 할 수 있는 부분에서 중요하다고 생각한다. 내가 추가적으로 선언한 구조체는 오른쪽 그림과 같다. 한 block의 형태를 정의하고 있는 Block(Partition에서 1개의 Block 단위를 나타내고 있다고 생각하면 편하다), root Directory 내부에서 내부의 파일들의 정보를 저장하는 역할을 하는 rootDirectoryInfo(Data Block 0번에 root Directory의 정보들이 table 형태로 들어가는데, 그 table의 한 row라고 생각하면 편하다), 이렇게 2가지이다. BLOCK_SIZE는 이번 과제에서 가정한 한 Block의 크기인 512를 저장하고 있는 상수이다.

```
//한 block의 형태를 정의하고자 한다 (이는 나중에 Partition에 사용될 것임)
typedef struct {
    char data[BLOCK_SIZE];
} Block;

//Data Block 0번에서 root Directory 내부의 파일들의 정보를 저장하는 역할을 할 것임
//이는 이후 배열로 선언되어 Data Block 0번에 집어넣을 것이다
typedef struct {
    unsigned char inum; //file의 inode number
    unsigned char Name[3]; //file의 이름
} rootDirectoryInfo;
```

이렇게 과제에서 주어진 가정들을 분석하고, 필요한 data type들을 정의하면서 본격적인 구현의 기틀을 다졌다. 이제 단계적으로 구현을 시작해보려 한다. 첫번째로 구현해야 하는 기능은 Block을 64개 갖는 Partition을 동적 할당하고 초기 상태로 초기화해주는 것이다. 초기화하는 inode number 0번과 1번은 각각 inode가 존재하지 않는 상태와 bad block(이번 과제에서는 발생하지 않음) 상태로 예약되어 있고, 2번은 root directory를 위한 inode로 할당시켜 주어야 한다. 또한, Data Block 0번은 앞서 언급했듯 root Directory 내부의 file 정보들을 저장하는 table이 들어가야 하는데, 초기 상태에서 table의 Name들은 (위 그림 내부의 rootDirectoryInfo 구조체의 멤버 Name을 의미하는 것)은 NULL 값, inum 값들은 0으로 초기화된 상태를 만들어야 한다고 과제의 요구사항에 존재한다. 즉, 앞서 설명한 이러한 초기화 기능만을 동작시키면 되기에 크게 어렵지 않은 작업이라 할 수 있다. 우선 최대 블럭 개수인 64개만큼 overall_Partition을 BLOCK의 배열 형태로 동적할당 시켜준다(main 함수에서 동적 할당된다. 내가 구현한 fsku.c에서 overall_Partition은 전역 변수로 선언되어 있기 때문에 구현의 편의를 위해 main에서 동적할당 및 free 실시). 그리고, i-bmap(inode 비트맵)의 1,2,3번째 비트, d-bmap(data block 비트맵)의 첫번째 비트를 바꿔주어야 한다. i-bmap의 1,2번째 비트는 어떠한 directory나 data와 관계 없이 예약된 bit이므로 초기화 시작 직후 바로 bit값을 바꿔 주었으며, i-bmap의 3번째 비트와 d-bmap의 첫번째 비트는 root directory 초기화와 관련된 비트이므로 inode 초기화까지 진행된 이후 따로 root Directory 관련 초기화가 진행될 때 바꿔주는 작업을 수행하였다. 추가적으로, i-bmap과 d-bmap은 절반씩 공간을 차지하며 한 block에 정의되어 있으므로, i-bmap에 뒤에 이어오는 d-bmap의 시작 지점을 알려주는 상수인 DATA_BITMAP_STARTPOINT(i-bmap과 d-bmap은 비트 단위로 처리되므로 256 * 8와 같이 비트 단위로 처리할 수 있도록 값을 정의했다)를 define 해주었다. 이러한 기능들을 수행하는 함수 initialize_Partition()을 따로 정의해 주었다. 추가적으로 root Directory의 초기화를 수행하는 set_RootDirectoryDataBlock(), inode 블럭들의 초기화 작업을 수행하는 initialize_inodeBlock(), i-bmap과 d-bmap에서의 비트 연산 수행 편의를 위한 setBit(), clearBit(), getBit()(앞에서부터 각각 특정 bit 정보를 1로 set, 특정 bit 정보를 0으로 set, 특정 bit 정보를 가져오는 기능을 수행하는 함수이다) 함수들을 정의해 주었다. 내가 구현한 초기화 과정에서 특징이라면, inode block을 초기화할 때 dptr(해당 file의 direct pointer), iptr(해당 file의 indirect pointer) 값은 data block 0번이 존재하기 때문에 fsize(해당 file의 file size), blocks(해당 file이 차지하고 있는 block 개수) 변수와 다르게 함부로 0이라는 값으로 할당하기 곤란하다고 판단되어 이번 과제에서 구현한 시스템에서 전혀 사용하지 않는 data block 번호인 100을 할당 해주었다는 것이다. 이는 추후 write operation에서 file의 inode의 iptr이 가리키는 data block 내부의 dptr들을 초기화할 때에도 100이라는 값으로 초기화하여, 유효하지 않은 값으로 판단하도록 설정하고, 여러 함수 내부의 조

건문에서 활용된다. 이러한 함수들을 통해서 이번 과제에서 Partition 역할을 수행하는 overall_Partition을 과제의 요구사항에 맞게 초기화 해줄 수 있었다.

```
AA w 30
AX w 28670
AZ r 300
ZX w 3000
```

두번째로 구현해야 하는 기능은 input file을 읽어내는 기능을 구현하는 것이다. 그러나, 이는 main() 함수에서 argv[] 인자를 통해서 input file 이름을 받아온 후, 그 input file을 분석하는 것으로 기능 구현이 마무리 되기에 크게 어렵지 않은 문제이다. 가장 중요한 문제는 문자열이 왼쪽 그림과 같은 형태이므로 한 줄씩 문자열을 읽어와서 각각 파일명, I/O command, 바이트 수를 분리해야 하는 것인데, 이는 strtok()를 이용하여 해결할 수 있었다. 공백과 개행문자('\n')를 기준으로 문자열을 분리하여 각각 따로 저장하였다. 특히, main 함수에서는 I/O command에 따라서 시행해야 하는 operation이 다르므로, 조건문을 사용하여 각 command마다 호출해야 하는 함수를 달리 해주었다. 이로서 두번째 기능은 간단히 구현이 마무리될 수 있다.

세번째로 구현해야 하는 기능은 'w' command가 들어왔을 때 시행되어야 하는 쓰기(write) 기능을 구현하는 것이다. write operation은 해당 file이 이미 있는 경우(input file에서 읽어 들인 file이 이미 존재하는 경우)와 해당 file이 없는 경우, 크게 2가지 경우로 나누어서 실시해야 한다. 또한, 해당 file이 이미 있는 경우에서도 현재 file이 가지고 있는 data block 만으로도 충분한 경우와, 현재 file이 가지고 있는 data block 만으로는 부족하여 data block을 새로 할당해 주어야 하는 경우로 나눠볼 수 있다. 현재 가지고 있는 data block 만으로 부족한 경우 안에서도 2가지 경우로 나누어 생각해야 한다. 해당 file의 inode에 indirect pointer가 할당되어 있어 새롭게 indirect pointer가 가리키는 data block을 새로 할당해 주지 않아도 되는 경우(이번 과제에서는 indirect pointer가 가리키는 data block 내부에 direct pointer들만 들어간다고 가정한다)와, indirect pointer가 할당되어 있지 않아 새롭게 indirect pointer가 가리키는 data block을 할당해 주어야 하는 경우로 말이다.

해당 file이 없는 경우 속에서도 2가지 경우로 나누어 생각해 보아야 한다. (현재 input file에서 읽어 들인 bytes값) + 1 (1을 더한 이유는 문자를 data block에 다 집어넣은 후, 해당 file의 끝이라는 것을 표시하는 '\0' 문자를 삽입하기 위함이다. 이는 내가 구현한 read operation에서 가득 차지 않은 data block을 읽어내는 기능을 구현할 때 %c로 출력하는 대신, %s로 출력 함으로서('\0'로 char형 배열의 끝부분이라는 것을 표시하고 있으므로 %s를 통해 출력할 수 있다 / 이는 실제 file들에서도 문자열의 끝에는 '\0'을 넣어주며 문자열의 끝을 표시하므로 그러한 방식을 따른 구현이라 볼 수 있다) 구현을 편하게 만들어준다)의 값이 512보다 작거나 같은 경우여서 data block을 하나만 할당하면 되는 경우와, 512보다 커서 data block을 최소 3개(direct pointer가 가리키는 data block, indirect pointer가 가리키는 data block, indirect pointer가 가리키는 data block 내부의 direct pointer가 가리키는 data block들로 구성되어야 함) 할당 해주어야 하는 경우로 말이다. write에서 나눠야 하는 경우가 상당히 많은데 위의 경우들을 정리하면 아래와 같다.

1. **<u>해당 file이 이미 존재 (root directory 정보에 해당 file의 이름이 이미 존재하는 경우)</u>**
   - A. 현재 file이 가지고 있는 data block 만으로도 충분히 write 가능
     - ➔ *(512 - ((해당 file의 inode의 file size(fsize) 값)%512) >= (쓰려는 bytes수 + 1)인 경우*
   - B. 현재 file이 가진 data block 만으로 부족하여 data block을 새로 할당해 주어야 함
     - i. 해당 file의 inode에 iptr(indirect pointer)가 할당되어 있어 iptr이 가리키는 data block을 새로 할당해 주지 않아도 되는 경우
     - ii. 해당 file의 inode에 iptr이 할당되어 있지 않아 iptr이 가리키는 data block을 새로 할당해 주어야 하는 경우
2. **<u>해당 file이 존재하지 않음 (root directory 정보에 해당 file의 이름이 존재하지 않는 경우)</u>**
   - A. (현재 input file에서 읽어 들인 bytes값) + 1 의 값이 512보다 작거나 같은 경우여서 새로 만들려는 file에 data block을 하나만 할당하면 되는 경우
   - B. (현재 input file에서 읽어 들인 bytes값) + 1 의 값이 512보다 커서 data block을 최소 3개 할당 해주어야 하는 경우

위 분석 결과를 바탕으로 fsku.c에서 file의 write operation을 시행하는 write_operation()을 구현했다. 이번 과제 중 가장 많은 비중을 차지하며, 가장 구현하기 까다로운 함수였다. 발생할 수 있는 상황들을 모두 고려하여 위처럼 분기해야 하는 경우의 수를 정리해야 했기 때문이다.

내가 구현한 write_operation()의 가장 큰 특징 중 하나는, file의 inode에 저장되어 있는 fsize 값과 input file로 들어온 input 하려는 bytes 값과의 직접적인 비교를 통해서 위의 1, 2번의 경우, 더 나아가 하위의 경우들도 조건문을 통하여 별다른 변수 선언 없이 분기를 나눌 수 있었다. 이러한 구현이 가능했던 이유는 과제에서 가정했던 write operation 연산 방식과 512bytes로 고정되어 있는 block size 덕분이었는데, 이번 과제에서는 input file로 들어온 파일명의 첫번째 알파벳을 input 하려는 bytes 값만큼 data block에 찍어내는 것이 유일한 write operation이었다. 마지막 block에 적혀 있는 bytes의 개수는 위 2가지의 가정으로 인해 file의 fsize 값과 512와의 나머지 연산을 통해 도출해낼 수 있으며, 따라서 마지막 data block의 남은 공간을 (512 - ((해당 file의 inode의 file size(fsize) 값)%512)라는 수식으로 구해낼 수 있다. 해당 함수를 더 자세히 살펴보면, data를 write하기 위해 필요한 block 개수 또한 file의 fsize를 이용하여 도출해 내고 있는 것을 확인할 수 있다. fsize의 값을 이용하는 곳은 write_operation() 이외에도 read, delete operation에서도 유용하게 활용하고 있다. i-bmap과 d-bmap을 바꾸기 위해 해당 분기마다 setBit(), getBit() 함수를 사용한 것, 해당 file의 마지막 write 연산에서 '\0' 문자를 삽입해 주는 것, 기존에 있던 file의 뒤에 이어서 써야 한다면 기존에 쓰여져 있는 '\0'을 지우고 write하기 시작해야 하므로 fsize를 -1 해주고 write operation을 해주는 것 등등 write_operation()에 관하여 설명해야 할 사항이 많다고 생각되지만, 소스 코드의 양 자체가 워낙 방대하여 해당 보고서에서는 전부 다루기 어렵다고 생각한다. 대신 구현한 fsku.c 소스 코드 내부에 설명이 필요하다고 생각이 들었던 구역마다 주석을 달아 놓았으므로, 이를 참고하면 방대한 양의 소스 코드 이해에 도움이 될 것이라 판단된다.

네번째로 구현해야 하는 기능은 'r' command가 들어왔을 때 시행되어야 하는 읽기(read) 기능을 구현하는 것이다. read operation은 write operation에 비하면 크게 복잡하지 않게 구현할 수 있다. 우선 read operation의 기본 구성은 우선 root directory가 가지고 있는 file 정보들을 찾아보며 읽으려 하는 file 이름이 있는지 체크한다. 해당하는 file 이름이 존재한다면 해당 file 이름에 해당하는 inode number를 이용하여 file의 inode를 inode block에서 가져온다. file 이름이 존재하지 않는다면, 비정상 종료임을 명시하는 flag 값(-1)을 넘겨주며 함수가 종료된다. -1을 넘겨주며 비정상 종료임을 알리는 방식은 read operation에서 뿐만 아니라 앞에서 설명했던 write, 그리고 뒤에서 설명할 delete operation 모두에서 사용하는 방식이다. main()에서 각 operation에 대한 함수로부터 -1을 return 받으면 과제에서 제시한 문장들을 출력하도록 구현되어 있다. 반대로, 이 3개의 함수에서 정상적으로 해당하는 과정을 모두 수행했다면 0을 return한다.

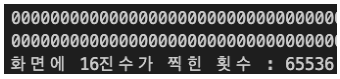
-1을 반환하지 않고 해당하는 file 이름을 찾았다면, 읽어낼 bytes 수와 해당 file의 fsize 값을 비교한다. fsize 값이 읽어낼 bytes 수보다 작거나 같은 경우엔, 해당 file의 모든 내용을 다 읽어낼 수 있는 경우이다. 이 경우에도 하위 2가지 경우로 나눌 수 있다. 해당 file에 iptr이 할당되어 있지 않은 경우(dptr로 가리켜져 있는 data block 하나만을 차지하고 있는 경우), iptr이 할당되어 있는 경우(dptr로 가리켜져 있는 data block 뿐만 아니라, iptr이 가리키는 곳의 dptr도 차근차근 읽어내도록 해야 한다)로 나뉜다. iptr이 가리키는 곳이 없으면, 읽어낼 bytes만큼 data block에서 읽어내면 되고, iptr이 가리키는 곳이 있다면 우선 dptr이 가리키는 data block부터 읽어내고, iptr이 가리키는 data block 내부의 dptr들이 data가 write 되는대로 순차적으로 저장되기에 읽을 때도 순차적으로 읽어내서 data block 내부의 dptr들이 가리키는 data block에 순차적으로 가서 문자들을 읽어내면 된다.

fsize값이 읽어낼 bytes 수보다 큰 경우에는, 해당 file의 모든 내용을 다 읽어낼 수 없는 경우이다. 이 상황에서도 위와 마찬가지로 해당 file에 iptr이 할당되어 있는 지의 유무에 따라 시행하는 과정이 다르다. iptr이 가리키는 곳이 없다면 block 하나만 읽고 끝내며, iptr이 가리키는 곳이 있다면 iptr이 가리키는 곳으로 따라가서 dptr값을 토대로 data block의 값들을 순차적으로 읽어내면 된다. 이러한 구성이 read operation을 수행하는 read_operation()의 전부이다. write_operation()에 비해 비교적 간단한 로직으로 구현할 수 있었다.

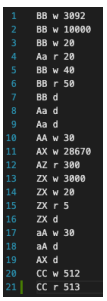
다섯 번째로 구현해야 하는 기능은 'd' command가 들어왔을 때 시행되어야 하는 삭제(delete) 기능을 구현하는 것이다. delete operation 또한 write operation에 비하면 크게 어렵지 않게 구현 가능하다. 우선 read_operation()과 마찬가지로 root directory에서 삭제할 file 이름을 찾는다. 존재하지 않으면 비정상 종료(-1) 값을 반환하고, 찾았다면 read에서와 마찬가지로 해당 file의 inode number를 가져와 inode block에서 해당 file의 inode를 찾는다. 이를 이용하여 삭제 작업을 실시하는데, 삭제 작업을 실시할 때 2가지 경우로 나누어 수행해야 한다. file이 가지고 있는 block 개수가 2개 이상(iptr이 할당되어 있는 경우)와 아닌 경우로 말이다. iptr이 할당되어 있다면, iptr이 가리키고 있는 data block 내부에 저장되어 있는 dptr 값들을 이용하여 그와 연관된 data들을 모두 지워줘야 한다. 그리고, 사용하고 있는 data block들을 비워준 후, d-bmap에서 해당 data block은 사용하지 않는 상태라고 bit 값을 0으로 수정해준다. bit 값 수정을 용이하게 하기 위해

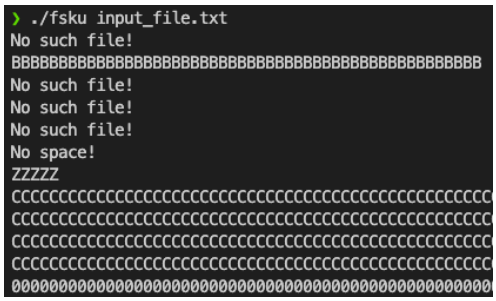
clearBit() 함수를 사용했다. iptr과 연관된 삭제 작업이 모두 진행되면, iptr이 가리키고 있는 data block 또한 비활성화 시키고, dptr의 데이터를 지우고, dptr이 가리키는 data block을 비활성화 시키고, 해당 file의 inode를 지우고, 마지막으로 i-bmap에서 해당 inode를 사용하지 않는 상태라고 bit 값을 0으로 수정해준다. iptr이 할당되어 있지 않은 경우는 dptr에 관한 정보들만 삭제 작업을 수행해주면 되므로 더욱 간단하다. 이러한 과정을 거치면 write operation을 수행하는 delete_operation() 함수 또한 구현이 완료된다.

마지막으로 구현해야 하는 기능은 input file의 끝에 도달했을 때, 해당 과제에서 구현했던 partition(내가 구현한 fsku.c에서의 overall_Partition을 의미)의 전체 내용을 16진수로 출력해주는 것이다. 이는 16진수가 4비트씩 묶어서 계산하면 편리하다는 사실을 이용하면 간단한 비트 연산을 통해 구현할 수 있다. 인자로 overall_Partition과 overall_Partition의 총 block 개수를 받아서 화면에 내용을 출력할 것이다. 32kb를 4bit씩 끊어서 출력할 것이므로 화면엔 총 65,536개((32kb/4bit = 32,768byte /4bit = 262,114bit/4bit)의 숫자가 찍혀야 한다. 내가 따로 정의한 Partition의 전체 내용을 16진수로 출력해주는 printBlockArrayTo_HexaDecimal() 내에서 int형 count 변수 하나를 0으로 초기화 시킨 후, print문 뒤에 count 변수를 계속 증가시키며 65,536개가 찍히는 지 세본 결과 위쪽의 결과처럼 잘 count된 것을 확인할 수 있다. 이로서, 과제에서 요구한 모든 기능들을 구현 완료했다.



지금까지 내가 구현한 fsku.c의 전반적인 형태를 살펴보았다. 옆의 2가지 그림 중 왼쪽은 임의의 input file을 적어 놓은 것이고, 오른쪽은 그에 대한 결과 화면의 일부를 나타낸 것이다. 0으로 쭉 찍혀 있는 부분부터는 Partition의 전체 내용을 65,536개의 숫자로 찍어내는 부분이므로 전체 스크린샷을 담지 못했지만, 그 이전까지는 시행한 read, write, delete 연산에 대해 적절하게 과정을 수행하는 것을 확인할 수 있다.





소스 코드가 길어짐에 따라 설명해야 할 내용들이 많아져서 보고서의 제한된 분량 상 분석한 모든 사항들을 담을 수는 없었다. 그렇지만, 앞서 언급했듯 fsku.c 소스 코드 내부에서 설명이 필요한 곳마다 주석을 달아 놓았기에, 이러한 주석들을 참고한다면 방대한 양의 소스 코드 이해에 도움이 될 것이다. 이에 더해 아래에 fsku.c에 내가 구현한 주요 함수들에 대한 간단한 표를 제시할 것이다. 이 또한 참고하면 프로그램 이해에 도움이 될 것이다.

2. Function Description

Function Name	Details	
initialize_Partition	Functionality	해당 과제에서 Storage 역할을 하는 Partition을 초기화하는 함수
	Parameters	Block* overall_Partition (해당 과제에서 Storage 역할을 하는 Parittion)
	Return Value	void **(없음)**
read_operation	Functionality	해당하는 file의 read operation을 수행하는 함수
	Parameters	char* file_Name(읽으려는 file 이름) / int bytes (읽어내려는 bytes수)
	Return Value	int (비정상 종료인 경우 -1, 정상 종료인 경우 0을 반환)
write_operation	Functionality	해당하는 file의 write operation을 수행하는 함수 (이어적기 혹은 신규 생성 후 write)
	Parameters	char* file_Name(write 하려는 file 이름) / int bytes (write 하려는 bytes수)
	Return Value	int (비정상 종료인 경우 -1, 정상 종료인 경우 0을 반환)
delete_operation	Functionality	해당하는 file의 delete operation을 수행하는 함수
	Parameters	char* file_Name(삭제 하려는 file 이름)
	Return Value	int (비정상 종료인 경우 -1, 정상 종료인 경우 0을 반환)
printBlockArrayTo_HexaDecimal	Functionality	해당 과제에서 구현한 Partition의 전체 결과를 16진수로 출력하는 함수
	Parameters	Block* blockArray (해당 과제에서 Storage 역할을 하는 Partition) / int size (Partition의 Block 개수(이번 과제에선 64의 값을 가짐))
	Return Value	void **(없음)**