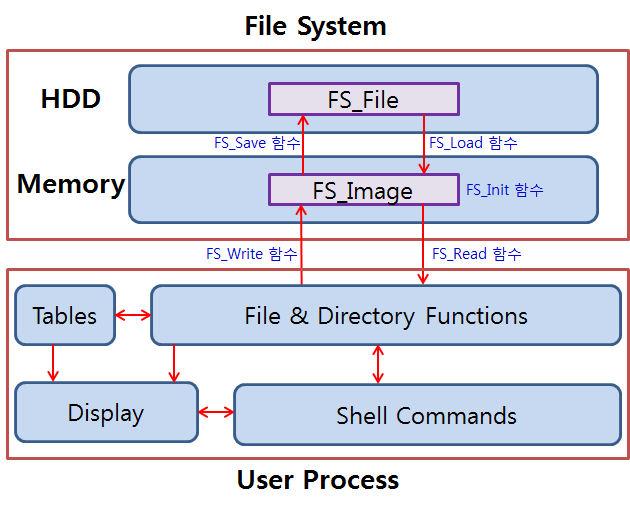
**리눅스 에뮬레이터**

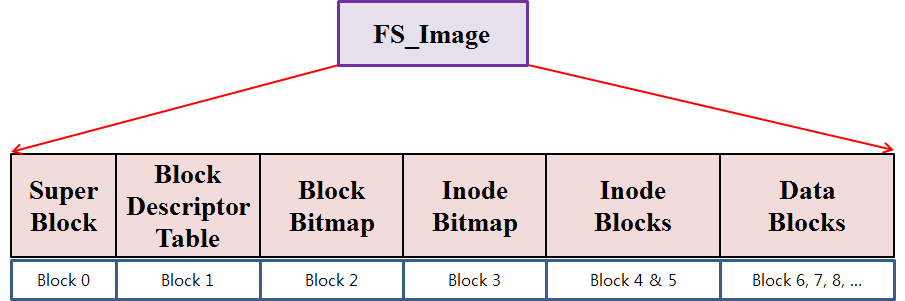
cd, ls, cat, rm 등과 같은 리눅스 기본 명령어를 실행하고 테스트할 수 있는 리눅스 에뮬레이터를 구현한다. 리눅스 기본 명령어를 다루기 위해서는 파일 시스템이 구축되어 있어야 한다. 즉, 구축된 파일 시스템 상에서 명령어를 수행한다. 다음 그림은 구현할 리눅스 에뮬레이터의 전체적인 구조를 나타낸다. 리눅스 에뮬레이터는 크게 파일 시스템과 사용자 프로세스로 나눠진다.

****

**[리눅스 에뮬레이터 구조]**

**<< 파일 시스템 >>**

일반적인 파일 시스템은 운영체제가 하드디스크에 구축하는 것이지만 본 과제에서는 하드디스크에 직접 구축하는 것이 불가능하므로 메모리를 이용하여 가상으로 간단한 파일 시스템을 구축한다. 그림에서 FS\_Image는 메모리에 구현할 파일 시스템을 위한 가상의 하드디스크를 의미하고 FS\_File은 구현한 FS\_Image 내용이 실제 하드디스크에 저장된 것을 의미한다. 파일 시스템은 리눅스에서 지원하는 Ext2를 기반으로 한다. 사이트(http://www.nongnu.org/ext2-doc/ext2.html)를 참고하여 작성한다.



**[FS\_Image 구조]**

FS\_Image 크기는 NUM\_BLOCKS \* BLOCK\_SIZE이다.

* NUM\_BLOCKS: 100 (FS\_Image가 갖는 총 Block 수)
* BLOCK\_SIZE: 4KB (4096 바이트) (한 Block의 크기)
* 블록이란 파일시스템에서 데이터를 저장하는 단위이며, 메모리에서 I/O 작업을 한 번에 읽거나 쓰는 단위가 된다.
* 4KB씩 100개 할당이 아닌 400KB를 문자형(char)으로 한 번에 할당한다. 할당된 FS\_Image의 각 Block에 접근하기 위해서는 FS\_Image가 일차원 배열이라 가정하고 4KB 단위로 읽고 쓴다.

FS\_Image는 6개의 블록으로 구성된다: Super Block(Block 0), Block Descriptor Table(Block 1), Block Bitmap(Block 2), Inode Bitmap(Block 3), Inode Blocks(Block 4 & 5), Data Blocks(Block 6~NUM\_BLOCKS). 실제 Ext2에는 아래에 설명된 정보보다 더 많지만 구현하는 시스템에서는 다음 정보만 포함한다.

* Super Block: Block 0을 사용하고 정보는 다음과 같다.
* Magic Number: 이 파일 시스템을 구현한 학생의 학번 입력
* Block Size: BLOCK\_SIZE (4KB)
* Blocks Per Group: NUM\_BLOCKS (100)
* Number of Inodes: Inode 수 (= Inode Blocks에 할당된 블록 수 \* Block Size / Inode Size)
* Inode Size: 128바이트 (하나의 Inode 정보를 저장하기 위한 크기)
* 날짜 및 시간: 파일 시스템을 구축 완료한 날짜 및 시간. 실제 리눅스 시스템 시간으로 함.
* 첫 번째 Data Block 번호: 6
* 첫 번째 Inode: Root ‘/’ 디렉토리 엔트리 (각 팀에서 결정할 것)
* Block Descriptor Table: Block 1 사용
* Block Bitmap의 블록 번호: 2
* Inode Bitmap의 블록 번호: 3
* 첫 번째 Inode Table의 블록 번호: 4
* 미 할당된 블록 수: 파일/디렉토리 생성 및 삭제, 크기 증감에 따라 계산
* 미 할당된 Inode 수: 파일/디렉토리 생성 및 삭제에 따라 계산
* Block Bitmap & Inode Bitmap: Block 2와 Block 3 사용
* Block Bitmap: 각 비트는 각 블록의 사용여부 나타냄 (1 - 사용 중, 0 - 미사용)
* 첫 번째 비트는 Block 0의 사용여부, 두 번째 비트는 Block 1의 사용여부, …
* Inode Bitmap: 각 비트는 각 Inode의 사용여부 나타냄 (1 - 사용 중, 0 - 미사용)
* 첫 번째 비트는 Inode 0의 사용여부, 두 번째 비트는 Inode 1의 사용여부, …
* Inode Blocks: Block 4 & 5를 사용하고 여러 개의 inode를 가지며 각 inode 정보는 다음과 같다.
* mode: 접근 권한 및 파일 종류
* size: 바이트 단위의 파일 크기
* time: 파일에 마지막으로 접근한 시간
* ctime: 파일 생성 시간
* mtime: 파일이 마지막으로 수정된 시간
* links\_count: 하드 링크 수
* blocks: 데이터를 저장하기 위해 할당된 block 수
* block[]: 데이터가 저장된 data block 번호 저장. 모두 Direct block으로 최대 파일 크기는 60KB이다. block[] 배열 순으로 실제 파일은 구성된다. 예로, blocks가 2이고, block[0]이 4, block[1]이 2이면 해당 파일은 Block 4가 파일의 앞부분이고 Block 2는 뒷부분이다.
* Data Blocks: Block 6번부터 사용하여 실제 데이터를 저장한다.
* 주의사항: 데이터 저장 시 데이터 정보 명은 저장되지 않고 순수 데이터만 문자형으로 저장된다.

**파일 시스템 관련 함수**

다음은 파일 시스템에서 사용할 기본적인 함수를 설명한다. 각 함수의 이름 및 파라미터는 참고용으로 프로그램을 구현할 때 추가/삭제/수정한다.

* FS\_Init() 함수: 리눅스 에뮬레이터 프로그램이 실행될 때 한번 호출된다. 메모리를 FS\_Image 크기만큼 동적 할당한다. 할당된 공간을 모두 0으로 초기화한다.
* FS\_Load(char\* FS\_File) 함수
* 하드디스크에 저장된 FS\_File 내용이 있으면 FS\_Init() 함수로 할당된 메모리 FS\_Image에 로드한다. 리눅스 시스템 콜 함수(open, read, close)를 이용하여 구현한다.
* FS\_File이 없으면 이 시스템이 처음으로 동작하는 것을 의미하므로 FS\_Init() 함수를 통해 할당된 FS\_Image에 파일 시스템을 구축한다.
* FS\_Image의 값을 위에서 설명한 것처럼 초기화하고 다음 그림과 같이 여러 개의 디렉토리와 inode를 생성한다.



* 구축한 후 FS\_Save() 함수를 호출하여 FS\_Image 데이터를 FS\_File에 저장한다.
* FS\_Save(char\* FS\_File) 함수: 메모리에 있는 FS\_Image 내용을 하드디스크의 FS\_File에 저장한다. 리눅스 시스템 콜 함수(open, write , close)를 이용하여 구현한다. 프로그램이 종료될 때 호출된다.
* FS\_Write(int block, char\* buffer) 함수: 사용자 프로세스의 buffer에 있는 데이터를 FS\_Image의 block에 저장한다.
* FS\_Read(int block, char\* buffer): FS\_Image의 block 위치에 있는 데이터를 buffer로 저장한다.
* FS\_Write와 FS\_Read의 buffer 파라미터는 동적으로 임시 할당된 메모리로 크기는 BLOCK\_SIZE이다. 즉, Block (4KB) 단위로 Read/Write를 수행한다.
* FS\_Image 값은 문자형이다. 따라서 FS\_Write()에서는 각 데이터를 문자형으로 변환한 후에 저장하고 FS\_Read()는 문자형을 읽어 각 데이터에 맞게 변환한다.

**<< 사용자 프로세스 >>**

사용자가 직접 파일 시스템에 접근하여 수정/삭제할 경우 치명적인 오류를 유발할 수 있다. 따라서 에뮬레이터 프로그램의 안정성을 높이기 위해 사용자가 파일 시스템에 직접 접근하는 것을 제한한다. 즉, 사용자는 파일 및 디렉토리 관련 함수를 이용하고 파일 및 디렉토리 관련 함수는 파일 시스템 관련 함수 중에서 FS\_Write()와 FS\_Read() 함수를 통해서만 파일 시스템에 접근 가능하다.

**파일 관련 함수**

다음은 파일 시스템에서 사용할 기본적인 파일 관련 함수를 설명한다. 각 함수의 이름 및 파라미터는 참고용으로 프로그램을 구현할 때 추가/삭제/수정한다. 관련 함수의 동작에서 필요한 file 명은 경로(path)를 포함할 수 있다. 경로는 절대경로 또는 상대경로로 표현된다. 절대경로는 루트를 나타내는 ‘/’로 시작한다. 상대경로는 . 또는 ..으로 시작한다. 경로 없이 파일명만 있으면 현재 디렉토리를 의미한다.

* File\_Create(char \*file): 파라미터 file이 가리키는 이름으로 크기 0바이트의 새로운 파일을 생성한다. 경로에 있는 파일이 이미 존재하면 에러를 리턴한다. 파일명의 최대 길이는 255바이트이다. 기본 접근 권한은 666이다.
* File\_Open(char \*file): file이 가리키는 이름의 파일을 열고 0 이상의 정수 파일 디스크립터(fd)를 리턴한다. 기존에 열려 있는 파일을 다시 열 수 없다. 즉, 닫힌 파일에 대해서만 열 수 있다.
* File\_Read(int fd, void \*buffer, int size): 파일 디스크립터 fd가 가리키는 파일로부터 size 바이트만큼 읽은 후 buffer에 저장한다. 파일 포인터(file\_pointer)가 가리키는 위치부터 읽어온다. 파일 포인터는 읽은 후에 새로운 위치로 갱신된다.
* File\_Write(int fd, void \*buffer, int size): buffer에 있는 데이터를 size 바이트만큼 파일 디스크립터 fd가 가리키는 파일에 파일 포인터(file\_pointer) 위치부터 저장한다. 저장한 후에 파일 포인터 값도 size만큼 증가한다.
* 위에 사용된 buffer 파라미터는 동적으로 임시 할당된 메모리로 크기로 사용자가 정의한 크기이다. 파일 시스템 관련 함수에서의 buffer 파라미터와 다르다. 만약 파일 크기가 6KB이면 4KB 단위로 하여 FS\_Write() 함수를 두 번 호출해야 한다. Read할 때도 이를 고려해야 한다.
* File\_Seek(int fd, int offset): 파일 디스크립터 fd가 가리키는 파일의 현재 파일 포인터의 값을 갱신한다. offset은 해당 파일의 첫 번째 위치(0번지)부터의 떨어진 거리이다. 즉, 파일 포인터 값을 offset로 설정한다 (file\_pointer = offset). 예로, offset이 150이면 해당 파일의 150번째 바이트를 가리키는 위치를 의미한다. offset이 음수이면 파일의 끝 위치로 설정한다.
* File\_Close(int fd): 파일 디스크립터 fd가 가리키는 파일을 닫는다.
* File\_Unlink(char \*file): file이 나타내는 파일을 삭제한다. 또한 해당 파일을 포함하는 디렉토리에서도 관련 내용을 삭제한다. 해당 파일이 사용하는 data block과 inode도 삭제해야하며 관련 bitmap도 0으로 설정한다. 단, File\_Open() 함수를 호출해 현재 사용 중인 파일은 삭제 못한다.

**디렉토리 관련 함수**

디렉토리는 다음과 같은 정보를 포함한다.

* Inode number (4바이트)
* Name length (1바이트)
* Name (0~255바이트): 파일 및 디렉토리 명

여러 함수의 동작에서 필요한 디렉토리 명(direc)은 경로(path)를 포함할 수 있다.

* Dir\_Create(char \*direc): direc에 있는 이름으로 새로운 디렉토리를 만든다. 새로운 디렉토리를 만드는 과정은 다음과 같다. 첫 째, 파일 종류가 디렉토리인 새로운 파일을 만든다. 둘 째, 새로 만들어진 디렉토리의 상위 디렉토리에 새로운 디렉토리 엔트리를 추가한다. 모든 디렉토리는 기본적으로 두 개의 엔트리를 갖는다. 하나는 현재 디렉토리를 나타내는 .과 상위 디렉토리를 나타내는 ..이다. 기본 접근 권한은 777이다.
* Dir\_Read(char \*direc, void \*buffer, int size): 디렉토리 direc에 있는 내용을 읽어 buffer에 저장한다. size는 buffer의 크기이다.
* Dir\_Unlink(char \*direc): direc가 나타내는 디렉토리를 삭제한다. 또한 해당 디렉토리를 포함하는 상위 디렉토리에서도 관련 내용을 삭제한다. 해당 디렉토리가 사용하는 data block과 inode도 삭제해야하며 관련 bitmap도 0으로 설정한다. 디렉토리 direc 내에 파일이 존재하면 디렉토리를 삭제하지 않고 에러 메시지를 출력한다. 즉, 빈 디렉토리만 삭제 가능하다. 루트(/) 디렉토리는 삭제 불가능이다.

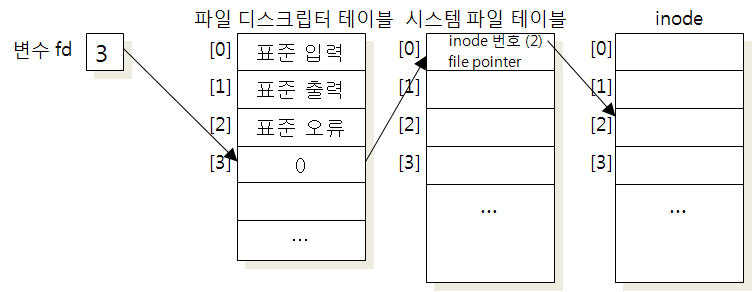
**파일 디스크립터 테이블(File Descriptor Table) & 시스템 파일 테이블(System File Table)**

리눅스 에뮬레이터 프로그램이 실행되면 메모리를 동적으로 할당하여 파일 디스크립터 테이블, 시스템 파일 테이블, Inode 테이블을 생성한다. 파일 디스크립터 테이블의 0, 1, 2는 자동으로 Std In/Out/Err로 할당된다. 시스템 파일 테이블과 Inode 테이블은 어떤 정보도 포함하지 않는다.

* 파일 디스크립터 테이블은 시스템 파일 테이블의 인덱스 번호를 저장한다. 0번은 표준입력, 1번은 표준출력, 2번은 표준오류로 자동 할당된다. 새로운 파일이 열릴 때마다 3번부터 부여한다. 중간에 빈 번호가 있을 경우 가장 작은 번호부터 부여한다 (0, 1, 2 제외).
* 시스템 파일 테이블은 2개의 정보로 구성된다.
* inode\_number: Inode 테이블에 해당 파일의 Inode 정보가 저장된 인덱스 번호 저장
* file\_pointer: 해당 파일에서 다음에 읽거나 쓸 위치를 바이트 단위로 나타낸다. file\_pointer는 파일이 처음 열리면 값 0으로 초기화된다. 해당 파일로부터 N바이트를 읽거나 쓰면 file\_pointer = file\_pointer + N으로 갱신된다.
* 시스템 파일 테이블도 0번부터 자동 부여하며 빈 공간이 있을 경우 가장 작은 번호부터 부여한다.
* Inode 테이블의 정보는 다음과 같다.
* Inode number: 해당 파일의 inode 번호
* Inode Blocks에 있는 해당 파일의 실제 inode 정보들

파일 I/O를 다루는 쉘 명령어에서 File\_Open() 함수를 호출하면 Inode Blocks에서 해당 파일의 Inode 정보를 읽어 Inode 테이블의 인덱스 n에 저장한다. 그러고 나서 시스템 파일 테이블의 인덱스 k에 관련 정보를 저장한다. 즉, inode 번호에 인덱스 n을 저장하고 파일 포인터 (file pointer)는 0으로 초기화된다. 파일 디스크립터 테이블의 인덱스 j에 시스템 파일 테이블의 인덱스 k를 저장한다. 마지막으로 File\_Open() 함수는 인덱스 j를 fd 값으로 리턴한다.

아래 그림은 이에 대한 예를 보여준다. File\_Open(test.txt) 함수를 이용하여 test.txt 파일을 열면 이 파일이 있는 디렉토리의 엔트리를 읽은 후 파일명으로 엔트리를 검색하여 inode 번호를 찾고 Inode Blocks에서 해당 Inode 정보를 Inode 테이블에 저장한다. 아래 그림에서는 Inode 테이블의 인덱스 2번에 저장하였다고 가정한다. 시스템 파일 테이블의 인덱스 0에 inode\_number = 2, file\_pointer = 0으로 설정한다. 그리고 파일 디스크립터 테이블 인덱스 3에 시스템 파일 테이블의 인덱스 0을 저장하고 사용자에게 파일 디스크립터 테이블 번호(fd) 3을 리턴한다. File\_Close(3) 함수를 호출하여 열린 파일을 닫으면 파일 디스크립터 테이블 인덱스 3에 저장된 시스템 파일 테이블 인덱스 0을 찾고 시스템 파일 테이블 인덱스 0에 저장된 Inode 테이블의 인덱스 2를 찾아 Inode 테이블의 인덱스 2에 있는 모든 정보를 삭제한다. 또한 시스템 파일 테이블의 인덱스 0에 있는 정보도 삭제하고 파일 디스크립터 테이블의 인덱스 3에 있는 정보도 삭제한다.



**디스플레이**

* 테이블 정보 출력: 파일 디스크립터 테이블과 시스템 파일 테이블, Inode 테이블 정보를 출력한다. 파일 디스크립터 테이블의 출력 정보 다음과 같이 현재 사용 중인 인덱스 번호(아래 예에서 Index로 표시)와 해당 인덱스에 저장된 시스템 파일 테이블의 인덱스 번호(아래 예에서 SF Index로 표시)이다. Index 0, 1, 2는 기본적으로 자동 할당되므로 “-“로 표시한다. 시스템 파일 테이블 정보의 출력 정보는 인덱스 번호, inode number (아래 예에서 Inode Index), file pointer이다. Inode 테이블은 현재 사용 중인 인덱스 번호와 Inode 번호만 출력한다.

|  |
| --- |
| << File Descriptor Table >>  Index SF Index  0 -  1 -  2 -  3 0  << System File Table >>  Index Inode Index File Pointer  0 11 0  << Inode Table >>  Index Inode Number  0 57 |

* 블록 정보 출력: 출력할 블록 번호를 입력받고 해당 블록의 정보를 출력한다. FS\_Read() 함수를 이용하여 구현한다.
* Block 0: Super Block에 해당하고 Super Block의 내용에 맞게 정보를 출력한다.

|  |
| --- |
| Magic Number: 20101234  Block Size: 4KB  … |

* Block 1: Block Descriptor Table에 해당하고 내용에 맞게 정보를 출력한다.

|  |
| --- |
| Block Bitmap: 2  Inode Bitmap: 3  … |

* Block 2: Block Bitmap에 해당하고 정보를 다음과 같이 출력한다. 한 줄에 10개의 정보를 출력한다. NUM\_BLOCKS까지만 출력한다.

|  |
| --- |
| 1111101111  0000000111  … |

* Block 3: Inode Bitmap에 해당하고 정보를 Block 2와 같은 형식으로 출력한다. (Inode 수만큼만 출력한다)
* Block 4: Inode Blocks에 해당하고 각 Inode 정보에 맞게 내용을 출력한다. 사용 중인 Inode에 대해서만 출력한다. (Block 5에 있는 정보도 같이 출력한다.) 아래 예와 같이 사용 중인 Inode 개수와 각 inode의 번호도 출력한다.

|  |
| --- |
| < 사용중인 Inode 수: 7 >  inode: 0  mode: (ls –ali 결과의 파일 타입 및 접근권한과 비슷하게 출력)  …  Blocks: 2  Block[] = 2, 5  …  inode: 1  mode: (ls –ali 결과와 비슷하게 출력)  … |

**쉘 명령어**

에뮬레이터에서 사용되는 쉘 명령어는 실제 리눅스 쉘 명령어와 비슷하게 동작한다. 그러나 옵션은 없다. 명령어에 사용되는 파일명이나 디렉토리 명에는 절대경로 또는 상대경로를 포함할 수 있다. 모든 쉘 명령어 입력은 정상으로 가정하고 예외처리 하지 않는다.

* ls - Dir\_Read() 함수를 이용하여 해당 디렉토리의 엔트리 정보를 출력한다.
* 디렉토리와 일반 파일의 정보를 리눅스 명령어 “ls –ali” 결과처럼 출력한다.
* 사용 예

ls - 현재 디렉토리 엔트리 출력

ls /a/b - 루트 밑에 디렉토리 a 밑에 디렉토리 b의 엔트리 출력 (절대경로)

ls ../a - 현 디렉토리의 상위 디렉토리에 있는 디렉토리 a의 엔트리 출력 (상대경로)

* mkdir - Dir\_Create() 함수를 이용하여 새로운 디렉토리를 생성한다.
* 사용 예

mkdir a - 현재 디렉토리에 새로운 디렉토리 a 생성

mkdir /a/b - 절대경로에 맞는 위치에 디렉토리 b 생성

mkdir ../a - 상대경로에 맞는 위치에 디렉토리 a 생성

* rmdir - Dir\_Unlink() 함수를 이용하여 해당 디렉토리를 삭제한다.
* 사용 예

rmdir a - 현재 디렉토리에 있는 디렉토리 a 삭제

rmdir /a/b - 절대경로에 맞는 위치에 디렉토리 b 삭제

rmdir ../a - 상대경로에 맞는 위치에 디렉토리 a 삭제

* cat > a - 키보드로 데이터를 입력받아 새로운 파일 a 생성, 기존 파일이 있는 경우 덮어쓰기
* File\_Create(), File\_Open(), File\_Write() 함수 등을 이용하여 구현
* "/quit"을 입력할 때까지 입력한 모든 내용을 저장한다.
* cat >> a - 키보드로 데이터를 입력받아 새로운 파일 a 생성, 기존 파일이 있는 경우 이어쓰기
* File\_Create(), File\_Open(), File\_Write(), File\_Seek() 함수 등을 이용하여 구현
* "/quit"을 입력할 때까지 입력한 모든 내용을 저장한다.
* cat a - 파일 a의 내용을 20줄씩 화면에 출력한다. 엔터를 치면 다음 20줄을 출력한다.
* "q"를 입력하면 종료한다.
* rm a - 파일 a 삭제
* a가 디렉토리인 경우에는 해당 디렉토리 및 하위 모든 내용을 같이 삭제한다.
* pwd - 현재 작업 디렉토리의 절대경로를 출력한다.
* chmod – 파일 및 디렉토리의 접근권한을 변경한다.
* chmod 222 a – 현재 디렉토리에 있는 파일 a의 접근권한을 -W--W--W-로 변경한다.
* 숫자를 이용하여 변경하는 것만 지원한다.
* mv a b - 파일 a를 파일 b로 이동한다. (a와 b가 같은 디렉토리이면 파일명 변경)
* cp a b - 파일 a를 파일 b로 복사한다.
* cd – 현재 작업 디렉토리를 바꾼다.
* cd a - 현재 디렉토리에 있는 디렉토리 a로 이동
* cd /a/b - 루트 밑에 a밑에 있는 디렉토리 b로 이동 (절대 경로)
* cd .. - 상위 디렉토리로 이동
* cd ../a – 상위 디렉토리에 있는 디렉토리 a로 이동 (상대 경로)
* split – 큰 파일을 절반 크기의 두 개 파일로 분할한다. 분할된 파일 이름은 xaa와 xab이다.
* split a - 현재 디렉토리에 있는 파일 a를 xaa와 xab로 분할하여 저장한다.
* paste – 입력된 두 개의 파일에 대해 첫 번째 파일의 끝에 두 번째 파일의 내용을 추가한다.
* paste a b - 현재 디렉토리에 있는 파일 a에 파일 b의 내용 추가하여 저장한다.
* filecopy a b – 이 명령어는 리눅스에는 없지만 테스트를 위해 만든 별도의 명령어이다. 실제 하드디스크에서 a라는 파일을 읽은 후에 구현한 파일 시스템에 새로운 파일 b를 만든다. a와 b의 파일 내용은 동일하다.
* close fd – 리눅스에는 없는 명령어지만 파일 디스크립터 번호가 fd인 파일을 닫기 위해 사용된다. File\_Close() 함수를 호출하여 동작한다.
* 주의: File\_Open() 함수로 호출된 모든 파일은 close 명령어가 실행되기 전에는 절대 File\_Close() 함수를 호출하지 않는다. 즉, 정상적인 절차에 따라 종료가 필요한 곳에서는 File\_Close() 함수를 호출하여 구현하지만 여기에서는 File\_Close() 함수를 호출하는 곳에 코멘트 처리한다.
* 예로 cat a하면 파일 a 내용을 확인하기 위해 File\_Open() 함수를 호출하고 모든 내용을 확인한 후에는 File\_Close() 함수를 호출하여 종료하는 것이 정상적인 절차이다. 그러나 여기에서는 디스플레이를 통해 테이블 정보를 확인하기 위해 종료하지 않는다.
* display – 리눅스에는 없는 명령어지만 디스플레이 기능을 위해 만든 별도의 명령어이다.
* display t: 디스플레이의 “테이블 정보 출력” 기능 수행
* display n: 디스플레이의 “블록 정보 출력” 기능 수행 – 블록 번호 n의 정보를 출력한다.

**<< 참고용 main() 함수 >>**

|  |
| --- |
| void usage(char \*prog)  {  printf("usage: %s <disk image file>\n", prog);  }  int main(int argc, char \*argv[]) // **프로그램 실행 시 FS\_File명 같이 입력**  {  if (argc != 2) {  usage(argv[0]);  exit(1);  }  char \*fs\_file = argv[1];  FS\_Init() 및 FS\_Load() 함수 호출  runShell();  FS\_Save() 함수 호출  return 0;  } |

* runShell() - 쉘 명령어를 수행한다. 시작 위치는 항상 “/home”이다. “quit”을 입력하면 runShell() 함수를 종료한다.

**<< 추가 사항 >>**

에뮬레이터 프로그램을 여러 개의 소스 파일로 구성하고 make 명령어로 컴파일한다.

* Makefile
* Makefile 작성하여 컴파일 수행
* Make clean도 동작해야 함

**[과제 검사 및 보고서 제출]**

1. 과제 검사는 **6월 9일 09:00**부터 **D440**에서 한다.
   1. 쉘 명령어 수행 여부로만 채점한다.
   2. 파일 시스템이 구현되지 않으면 0점 처리한다. 일부 미 구현된 내용에 대해서는 감점한다.
   3. 검사 방법은 추후 공지한다.
2. 결과보고서는 다음 페이지에 있는 구현 내용 점검 표, 프로그램 source code, 실행 결과를 포함하여 작성한다. (한글이나 MS Word로 작성)
3. 결과보고서는 과제 검사 시작 전(09:00)에 인쇄하여 제출한다. (이후에는 절대 받지 않음)
4. 또한, 관련 파일들을 1개의 파일로 압축하여 “강의지원시스템 – 과제”에 업로드 한다. 압축 파일에 포함될 파일은 다음과 같다.

* 결과보고서
* 소스코드

**\* 과제는 4인 이하 팀으로 한다.**

**\* 부정 행위 발견 시에는 의도와 상관없이 무조건 0점 처리한다.**

**\* 사용 언어: C, C++ (Java 등 다른 언어 제외)**

\* 결과보고서 두 번째 페이지에 각 기능의 구현 및 동작 여부를 나타낸다. (첫 페이지는 표지)

\* 정확한 동작: ○, 미 구현 또는 많은 오류: ×, 사소한 오류: △ (위 두 줄은 보고서에서 제외 바람)

**■ 구현 내용 점검 표**

|  |  |  |
| --- | --- | --- |
| **구분** | **기능 및 함수** | **동작** |
| 파일 시스템 동작 함수 | FS\_Init() | × |
| FS\_Load() | ○ |
| FS\_Save() | △ |
| FS\_Write() | × |
| FS\_Read() | ○ |
| 파일 관련 함수 | File\_Create() |  |
| File\_Open() |  |
| File\_Read() |  |
| File\_Write() |  |
| File\_Seek() |  |
| File\_Close() |  |
| File\_Unlink() |  |
| 디렉토리 관련 함수 | Dir\_Create() |  |
| Dir\_Read() |  |
| Dir\_Unlink() |  |
| 테이블 | 파일 디스크립터 테이블 |  |
| 시스템 파일 테이블 |  |
| Inode 테이블 |  |
| 쉘 명령어 | ls |  |
| mkdir |  |
| rmdir |  |
| cat > a |  |
| cat >> a |  |
| cat a |  |
| rm |  |
| pwd |  |
| mv |  |
| cp |  |
| cd |  |
| split |  |
| paste |  |
| filecopy |  |
| close |  |
| display 테이블 정보 |  |
| display 블록 정보 |  |
| 기타 | make |  |