운영체제 프로그래밍 과제

<SFS 파일 시스템>

과제 개요

본 과제에서는 현존하는 파일 시스템을 간략화 시킨 Simple File System(SFS)의 구현을 통한 파일시스템의 이해를 높이는 것을 목적으로 합니다. SFS는 실제 디스크를 사용 하는 것이 아닌, 파일을 통해 디스크를 시뮬레이션 하는 형태로 동작하는데, 과제에서는 주어진 디스크 이미지 파일을 사용하여 SFS와 관련된 명령어들을 구현하게 됩니다.

Simple File System

본 과제에서는 실제 디스크를 사용 하는 것이 아닌, 디스크와 동등한 형태의 디스크 이미지 파일을 사용 하여 과제를 진행합니다.(파일 DISK1.img, DISK2.img, DISKFull.img) 이들 이미지들은 SFS에서 사용하는 레이아웃 형태로 포맷되어 있는데, disk_read(), disk_write() 함수를 통하여 실제 디스크와 같이 블록단위로 엑세스하게 됩니다. 파일시스템은 디스크의 가용한 블록(free blocks)들을 관리 하거나, 파일시스템 내 실제 파일을 관리 하기 위해 많은 방법들을 사용하는데, FAT이나 i-node와 같은 방법이 이에 해당합니다. SFS에서는 디렉토리 구조나 파일을 관리하기 위해 i-node를 사용하고, 가용한 블록(free blocks)들을 관리하기 위하여 비트맵 형태의 사용합니다.

SFS Layout

SFS에서 사용 하는 디스크 이미지는 다음과 같은 형태로 이루어져 있습니다. 아래 그림의 각 칸은 디스크의 한 블록을 나타내게 되는데 디스크의 첫 번째 블록에는 슈퍼블록이, 두 번째 블록에는 루트 디렉토리의 i-node가 위치하게 됩니다. 슈퍼블록이 포함하고 있는 정보, i-node가 유지하고 있는 정보들은 모두 sfs.h에 정의 되어 있습니다. 항상 첫 번째 두 번째 블록은 고정이며 세 번째 블록부터는 비트맵이 오게 되는데 비트맵의 블록 수는 디스크 전체 블록 수에 따라 유동적이게 됩니다. 비트맵은 특별한 구조를 가지지 않고 순수하게 비트맵 데이터만 유지하게 되는데, 즉 한 비트맵 블록은 512*8개의 비트를 가질 수 있으므로 512*8개 블록의 할당/해제 정보를 나타 낼 수 있게 됩니다.(512*8인 이유는 한 블록의 크기가 512byte이고, 1byte는 8bits 이기 때문입니다) 하나의 비트맵 블록만 사용하는 디스크의 가장 초기 상태에서는 1, 2, 3, 4번째 블록(Super block, Root Directory i-node block, bitmap block, Root directory data block)만 사용 하고 있으므로 비트맵 블록의 1, 2, 3, 4번째 비트만 1로 표시되고

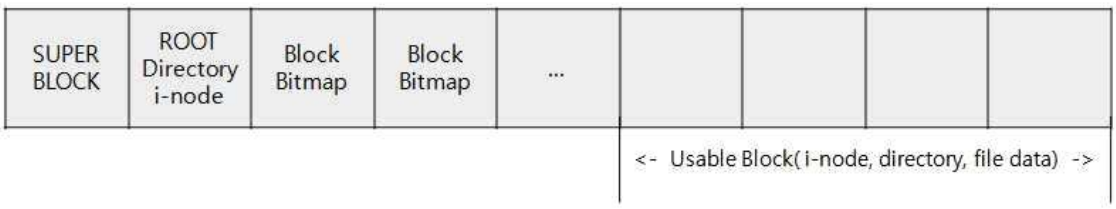

그림 1 . SFS layout

나머지는 0으로 표시되게 됩니다(실제 비트맵 블록은 SFS에서 사용되는 전체 블록의 수의 따라 여러 개 사용 될 수 있습니다). SFS에서는 i-node에 관한 리스트 (i-list)가 존재 하지 않습니다. 수업시간에 다룬 파일 시스템 구조에서는 i-list가 존재해서 i-node와 데이터 블록(데이터,

디렉토리 블록)이 다른 곳에 구별되어 저장 됩니다. 반면에 SFS에서는 이러한 구분 없이 저장되고, 디스크에 존재하는 어떠한 블록도 i-node가 될 수 있습니다. 이러한 특징으로 인해 단순히 블록번호를 i-node 번호로 사용할 수 있고, 따라서 새로운 i-node 할당은 디스크에서 사용되지 않는 한개의 블록을 할당하는 것과 일치 하게 됩니다.(i-node를 할당하기 위해 디스크의 한개의 블록을 할당했다면, 해당 블록의 번호가 i-node 번호 입니다)

과제 파일 설명

- DISK1.img, DISK2.img, DISKFULL.img

  과제에서 사용할 디스크 이미지 파일입니다. 세 파일 모두 위와 같이 SFS에서 사용하는 형태로 이루어져 있고 DISK1.img는 몇 개의 디렉토리 및 파일이 포함된 이미지, DISK2.img는 초기 상태의 이미지입니다. DISKFULL.img 는 꽉찬 디스크의 이미지 파일입니다.

- sfs_disk.h, sfs_disk.c

  DISK1.img, DISK2.img를 사용하기 위한 인터페이스입니다. 디스크 형태로 사용하기 위하여 블록단위로 엑세스할 수 있는 함수들이 정의되어 있습니다. 해당 함수들은 모두 구현이 완료되어 있는 상태로 수정할 수 없고, DISK1.img, DISK2.img에 대한 읽기와 쓰기는 반드시 이 파일에 정의된 인터페이스를 사용해야 합니다.

  ※ sfs_disk.h에 정의된 함수 중 과제에서 사용 할 인터페이스(함수)는 disk_read()와 disk_write()뿐입니다. disk_blocksize도 있긴 하지만 블록사이즈는 sfs.h에 정의된 것을 사용 하는 것이 좋습니다. disk_read와 disk_write에 대해 간략히 설명을 하면 두 함수 모두 첫 번째 인자로 버퍼의 주소를 받습니다.(블록에 대한 연산이므로 버퍼의 크기는 반드시 블록사이즈와 같거나 커야 합니다) 두 번째 인자는 실제로 쓰거나 읽을 블록의 번호를 받게 되는데 예를 들어 disk_read(buf, 1)와 같이 호출한다면, SFS에서 블록 번호가 1인 곳에 위치하는 데이터는 루트 디렉토리의 i-node 이므로 buf에는 루트 디렉토리의 i-node가 읽어지게 됩니다.

- sfs.h

  SFS에서 사용할 구조체 및 매크로가 정의되어 있습니다. 매크로로는 SFS의 블록사이즈나 매직넘버 등의 정보와, 블록 당 비트 수, 블록 당 디렉토리 엔트리 수와 같이 자주 쓰이는 연산 등이 정의되어 있습니다. 또한 본 과제에 있어 가장 많이 사용될 i-node 구조와 디렉토리 엔트리 구조가 정의되어 있습니다. 과제를 진행 할 때 i-node와 디렉토리 엔트리의 대한 이해도가 낮다면 과제를 진행하는데 어려움이 있을 수 있습니다.

  ※ sfs_inode의 sfi_linkcount는 하드 링크를 위한 변수로 본 과제에서는 사용하지 않습니다. sfs.h에 정의되어 있는 구조체를 변경하지 마세요. 정상적으로 동작하지 않을 수 있습니다

- sfs_main.c

  과제에서 사용 할 메인 함수 부분으로 과제에 필요한 쉘이 구현되어 있습니다. 과제에서 구현할 명령어에 대한 함수를 호출 합니다.

- sfs_func.h sfs_func_hw.c

  sfs_func.h 파일에는 본 과제에서 구현해야 할 명령어들의 프로토타입이 정의되어 있습 니다. 본 과제에서는 모든 구현을 sfs_func_hw.c에만 하게 됩니다. sfs_func_hw.c 파일에는 현재 디렉토리를 나타내는 sd_cwd와 슈퍼블록의 정보를 가지고 있는 spb를 전역변수로 가지고 있습니다. 각각의 명령어에서 필요시 이 두 변수의 값을 조정 하거나 참조함으로써 명령어를 구현 할 수있습니다. 숙제 제출시 sfs_func_hw.c 함수만 제출 하면 되고 다른 *.c *.h 파일은 숙제 검사 시 다른 버전이 사용될 수 있으므로 변경하면 안 됩니다.

  ※ sfs_main.c에 있는 명령어 함수 중 구현이 안 된 함수는 해당 함수를 sfs_func_hw.c 에 printf("Not Implemented₩n"); 문장을 넣어 임시로 완성하여 컴파일이 가능하도록 합니다.

과제 제한 사항

1. 모든 명령어의 구현에는 에러 상황에 대한 처리를 포함해야 합니다. 실제 쉘에서 명령어를 수행했을 때 처리되는 에러들을 포함 하시면 되는데 error_message() 함수를 사용하여 에러 메시지를 출력하기 바랍니다. (error_message() 함수에 없는 메시지는 별도로 출력하여도 됩니다) 명령어 수행 시 발생할 수 있는 에러 중 error_message() 함수의 메시지에 해당하는 에러는 반드시 처리하기 바랍니다. 각 명령어 설명에 발생 가능한 에러 상황에 대한 설명이 있습니다.
2. 과제에서 파일 수정은 sfs_func_hw.c만 가능합니다. 내부적으로 함수를 만들어서 사용하는 것은 무방하고, 명령어의 기능을 적절히 제공 할 수 있으면 됩니다.
3. 한 명령어가 끝날 때에는 디스크 이미지의 모든 변경 사항이 디스크 이미지로 쓰여 져야 합니다. 기존의 파일시스템의 경우 버퍼캐시 등을 이용하여 디스크의 변경 사항이 바로 쓰여지지 않고 주기적으로 쓰여 지지만, SFS에서는 버퍼캐시가 없으므로 명령어가 실행 될 동안 변경된 디스크 이미지는 명령어가 끝나기 전에 반드시 쓰여져야 합니다. 즉, 비트맵 등을 메모리상에서 관리하면 안 됩니다. 필요시 디스크에서 읽은 후 다시 디스크에 쓰는 형태로 관리 되어야 합니다.
4. 디렉토리 엔트리의 정보는 i-node내의 direct blocks에만 저장하고 indirect block(single indirect pointer)에 저장 하는 것은 사용하지 않습니다. 따라서 한 디렉토리가 가질 수 있는 최대 엔트리 개수는 SFS_DENTRYPERBLOCK*SFS_NDIRECT가 됩니다.
5. 모든 경로(path)를 입력 받는 명령어들에 대해서(ls, cd, mkdir, 등) 실제 쉘에서 사용하는 경로가 아닌 디렉토리 내의 실제 파일이나 디렉토리 이름을 입력 받는다고 가정합니다. 즉 “/os/homework” 또는 “./func.c” 같이 경로가 아닌 “a_file”, “b_directory” 같은 파일 또는 디렉토리 이름이 해당 명령어의 인자(path)로 사용된다고 가정합니다.

명령어 설명

※ 명령어 신택스 (syntax) 는 “명령어 [인자]” 또는 “명령어 인자1 인자2” 이며 [인자]는 “인자”가 있거나 아무것도 없는 것을 나타냅니다. 인자가 path 인 경우 파일 또는 디렉토리 이름만 사용되고 따라서 path에 “/” 들어가는 경우는 지원하지 않아도 됩니다.

- mount *disk_image_file_name*

  mount 명령어는 디스크 파일 이미지를 사용하고자 할 때 사용하는 명령어입니다. 다른 명령어를 사용하기 전에 mount가 선행되어야 합니다. mount 명령어에서는 슈퍼블록을 읽고 매직넘버 등을 확인하는 작업을 수행합니다. (mount 명령어는 예제로 제공이 되므로 다른 명령어 구현 시 참고 바랍니다)
- umount

  umount 명령어는 사용하던 디스크 파일 close하고 슈퍼블록과 current working directory 구조체 (spb, sd_cwd)를 초기화하여 다음 mount 명령어를 사용 할 수 있도록 하여 줍니다. (umount 명령어는 제공 되므로 사용만 하면 됩니다.)
- dump

  dump 명령어는 현재 디렉토리(current working directory)의 정보 - 디렉토리 i-node 정보, 디렉토리 엔트리 정보, 디렉토리 엔트리의 대상이 파일인 경우 파일 i-node의 정보를 출력합니다. (dump 명령어는 예제로 제공되므로 다른 명령어 구현 및 디버깅 시 사용하기 바랍니다.)
- ls *[path]*

  path로 명시된 디렉토리 내의 파일과 디렉토리의 이름을 출력하는 명령어입니다. 이 때 파일과 디렉토리의 구분을 위하여 디렉토리 이름 뒤에는 / 를 함께 붙어서 출력해야 합니다. path가 없으면 현재 디렉토리를 의미 하며 path가 파일이면 해당 파일 이름을 출력합니다. 에러 상황으로는 path에 해당하는 디렉토리나 파일이 없으면 경우로 에러 메시지를 출력해줍니다.
- cd *[path]*

  현재 디렉토리(current working directory)를 변경 하는 명령어입니다. 전역변수인 sd_cwd 는 current working directory 나타내는 구조체입니다. sd_cwd를 변경함 으로써 구현 할 수 있습니다. path 가

없으면 root 디렉토리가 current working directory가 됩니다. (mount 수행후 current working directory). 에러 상황으로는 변경할 디렉토리가 존재 하지 않는 경우, 해당 경로가 디렉토리인가 아닌 경우 등이 있습니다.

- touch *path*

  새로운 빈(크기가 0) 파일을 만드는 명령어입니다.(빈 파일을 만드는 명령어 이므로 저장 할 데이터를 위한 블록은 할당 받을 필요가 없고, 새로운 파일의 i-node를 위한 블록만 할당 받으면 됩니다) touch 명령어는 예제로 일부분이 구현되어 있는데 제한된 상황에서만 동작되는 형태로 구현되어 있습니다.(루트 디렉토리가 빈 상태에서만 동작 합니다. 이는 사용하지 않은 DISK2.img를 mount한 상태와 일치합니다) 이러한 이유는 디렉토리 엔트리를 위한 공간이나, i-node를 할당 받기 위해 블록을 할당 받을 번호를 실행 시간에 동적으로 탐색 하는 것이 아닌, 초기 루트 디렉토리 폴더만 존재 한다고 가정했을 때의 디렉토리 엔트리 위치와 블록 번호를 예측해서 구현했기 때문입니다. 비록 제한된 상황 에서만 동작되는 형태이지만 구현된 코드를 분석한다면 과제를 수행하는데 많은 도움이 될 것입니다. 실제로 디렉토리 엔트리의 위치, 할당, 비트맵을 이용한 블록 할당 등이 추가 된다면 완전한 구현이 될 수 있습니다. 에러 상황으로는 path가 이미 존재하는 경우, 디렉토리가 꽉차 더 이상 엔트리를 허용 못하는 경우, I-node를 할당 못한는 경우등이 있습니다. 참고로 할당된 I-node의 모든 필드 (특히 sfi_direct[], sfi_indirect)는 0으로 초기화 한 후 필요한 필드를 설정합니다.

- mkdir *path*

  새로운 디렉토리를 생성 하는 명령어입니다. i-node 구조체를 보면 현재 디렉토리에 대해 몇 개의 디렉토리 엔트리를 유지 하고 있는지에 대한 필드(변수)가 존재하지 않습니다. 이는 데이터 파일과 같은 형태의 i-node를 사용하기 때문인데, 데이터 파일에서 실제 용량을 나타내는 sfi_size 필드를 활용 할 수 있습니다. 본 과제에서는 디렉토리 엔트리가 추가 될 때마다 sfi_size의 값을 sizeof(struct sfs_dir) 만큼 증가시킴으로써 실제 몇 개의 엔트리가 사용 중인지 판단하도록 하겠습니다. mkdir을 통해 새로운 디렉토리를 생성 한 경우 새 디렉토리에 대해 .(자신)과 ..(부모) 디렉토리 엔트리를 추가해 주어야 합니다. 또한 할당 받은 디렉토리 블락의 모든 엔트리는 SFS_NOINO로 초기화 시키줍니다. 에러 상황으로 path가 이미 있는 경우(같은 이름의 파일이나 디렉토리), 할당 받을 디렉토리 엔트리가 없는 경우, 할당한 블락이 없는 경우 등입니다.

- rmdir *path*

  기존에 존재하는 디렉토리를 삭제 하는 명령어입니다. 해당 디렉토리의 이름(*path*)과 I-node번호를 저장하고 있던 부모 디렉토리의 디렉토리 엔트리의 내용은 지워지지만 차후의 재활용을 위하여 남겨 둡니다. 이 경우 해당 엔트리에 대해 i-node 번호로 SFS_NOINO를 사용합니다. 또 부모 디렉토리의 I-node의 sfi_size도 struct sfs_dir 사이즈만큼 줄어 들어야 합니다. 실제 할당 받은 디렉토리 블록에서 사용하는 디렉토리 엔트리의 수는 sfi_size를 struct sfs_dir 사이즈로 나누어 알아낼 수 있습니다. 내용이 지워진 디렉토리의 엔트리도 나중에 재사용 될 때까지 그대로 유지하기 때문에 할당되어 있는 디렉토리의 엔트리 수는 항상 sfi_size/sizeof(struct sfs_dir) 보다 같거나 크게 됩니다. 하나의 데이터 블락에 있던 모든 디렉토리 엔트리가 제거되어도 해당 데이터 블락을 free 할 필요는 없습니다. 에러 상황으로는 path가 없는경우, 해당 경로가 디렉토리가 아닌 경우, 디렉토리가 안 비어 있는 경우 등이 있습니다.

- mv *path1 path2*

  파일 또는 디렉토리 path1의 이름을 path2의 이름으로 바꾸어 주는 명령어입니다. 실제 쉐에서 사용하는 mv와 달리 이름만 바꾸어 주는 기능만 구현 합니다. 결국 path1의 디렉토리 엔트리에서 이름을 바꾸어 주는 명령어입니다. 에러 상황으로 path1 이 존재하지 않는 경우와 path2 가 이미 존재하는 경우를 처리 해주어야 합니다.

- rm *path*

  기존에 존재하는 데이터 파일을 삭제 하는 명령어입니다. 이 명령어는 rmdir과 비슷하게 구현 될 수 있습니다(해당 파일에 대한 디렉토리 엔트리의 재활용등 포함). sfi_direct[]와 sfi_indirect를 확인하여 할당된 모든 데이터 블락을 릴리즈해야 합니다. 그리고 할당된 I-node 블락도 릴리즈해야 합니다. 에러 상황으로는 path가 실제로 존재하는지, 해당 경로가 실제 데이터 파일인지에 대해서 검사해야 합니다.

- cpin *local_path path*

cpin은 호스트 PC(실제 머신)의 파일 시스템에 존재하는 파일(*path*)을 SFS의 새로운 파일(*local_path*)로 복사하는 명령어입니다. 파일을 복사하는 경우 direct blocks 외에 indirect block(single indirect pointer) 또한 고려하여 구현하셔야 합니다.(쓰려는 파일 용량이 커 direct blocks에 모두 저장하지 못한다면 indirect block에 저장하는 형태로 구현 해야 합니다). 에러 상황으로는 path 파일이 없는 경우, local_path 파일이 이미 존재하는 경우, local_path 파일을 만들 수 없는 경우 (디렉토리가 꽉 찬 경우, i-node를 할당 못 받는 경우, 데이터 블락을 할당 못 받는 경우), path 파일의 사이즈가 SFS 가 허용하는 파일의 사이즈를 넘어가는 경우 등입니다. 참고로 디스크가 꽉 차서 파일을 다 복사 못해도 가능한 부분까지 복사합니다.

- cpout *local_path path*

  cpout은 SFS에 존재하는 파일(l*ocal_path*)을 호스트 PC 파일 시스템의 path 파일로 복사 하는 명령어입니다. local_path 파일의 크기에 따라 sfi_direct[] 와 sfi_indirect에 설정된 모든 데이터 블락의 데이터를 path 파일로 복사 하여야 합니다. 에러 상황으로는 local_path 파일이 없는 경우, path 파일이 이미 존재하는 경우등 path 파일을 생성 못하는 경우입니다.

- 그 밖의 명령어 (디버깅용으로 이미 구현 되어 있음 sfs_func_ext.o)

  fsck 와 bitmap 명령어를 수행할 수 있다. fsck 는 파일 시스템의 트리를 차례로 따라가며 bitmap의 bit와 block의 사용이 일치하는지 확인한다. 문제가 있을시 "error"가 들어 있는 문장을 출력한다. bitmap 은 bitmap 데이터를 출력한다. 이들 명령어는 과제의 명령어 수행 후 명령어가 잘 수행되었는지 디버깅 할 때 참고로 사용하면 됩니다.

과제 구현

본 과제에서는 난이도에 따라 1, 2, 3단계로 나누어서 과제를 진행합니다. 각 단계는 구현 하는 명령어에 따라서 나뉘게 되는데, 1단계를 가장 처음 구현 하셔야 하고, 2단계, 3단계는 순서를 바꾸어 구현해도 무방합니다. 즉, 2단계가 어렵다면 1, 3단계를 구현하셔서 제출해도 됩니다.

1단계 - ls, cd의 구현

1단계는 i-node의 할당(블록 할당)이 필요하지 않은 명령어들의 구현입니다.

1단계를 구현 하실 때 아직 구현되지 않은 명령어에 대해서는 sfs_main.c 파일에서 해당 함수 호출 부분을 임시로 주석처리 해서 테스트 하시면 됩니다. 하지만 제출 시에는 printf("Not Implemented₩n"); 문장을 넣어 임시로 완성하여 컴파일이 가능하도록 합니다.

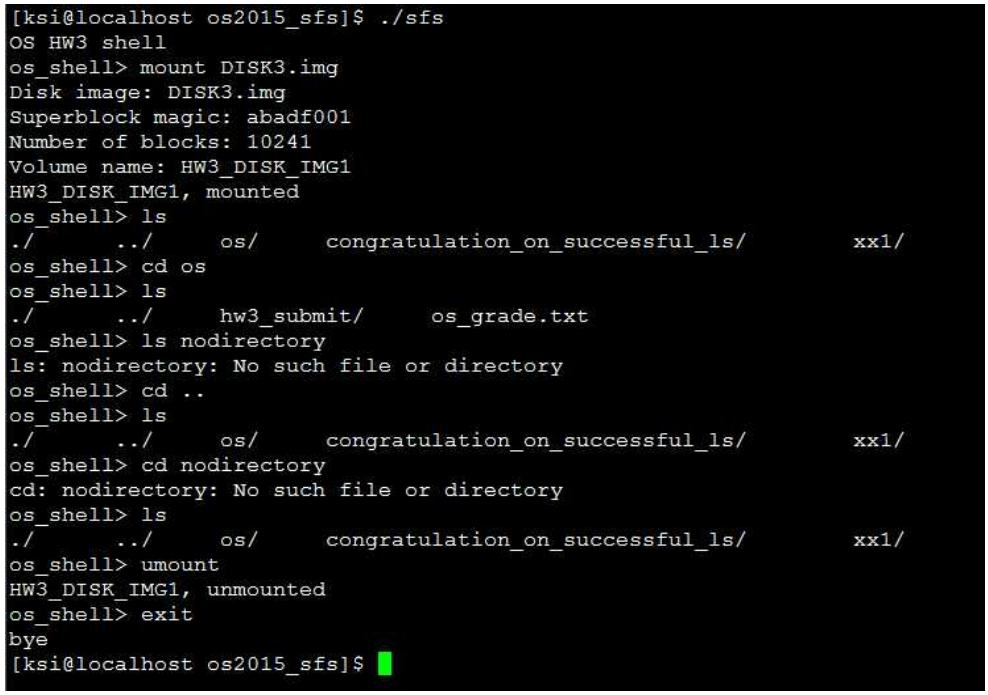

그림 2. 1단계 실행 화면

2단계 - mkdir, rmdir, rm, touch, mv의 구현

2단계와 3단계에서 구현할 명령어들은 일반적으로 i-node(블록)를 할당/할당해제 하는 메커니즘이 필요하고 디스크에 변경사항을 기록하는 명령어들 입니다. mkdir의 경우 direct blocks에 대한 블록 할당이 필요할 수 있고 rmdir 이나 rm은 더 이상 필요하지 않은 direct blocks 및 indirect block에 대해 할당 해제가 필요 할 수 있습니다. 명령어들은 해당 디렉토리 i-node의 sfi_size를 sizeof(struct sfs_dir) 만큼씩 조정함으로써 디렉토리가 가지는 실제 디렉토리 엔트리가 몇 개 인지를 파악할 수 있고, sfd_ino의 값을 보고 엔트리가 빈 상태 인지 구분 하는데 사용 할 수 있습니다. mkdir은 . 디렉토리(자신)와 .. 디렉토리(부모)의 초기화 과정도 포함해야 합니다.

```
[ksi@localhost os2015_sfs]$ ./sfs
OS HW3 shell
os_shell> mount DISK3.img
Disk image: DISK3.img
Superblock magic: abadf001
Number of blocks: 10241
Volume name: HW3_DISK_IMG1
HW3_DISK_IMG1, mounted
os_shell> ls
./      ../     os/     congratulation_on_successful_ls/        xx1/
os_shell> mkdir doc
os_shell> ls
./      ../     os/     congratulation_on_successful_ls/        xx1/    doc/
os_shell> touch afile
os_shell> ls
./      ../     os/     congratulation_on_successful_ls/        xx1/    doc/    afile
os_shell> rmdir doc
os_shell> rm afile
os_shell> mv xx1 xx2
os_shell> ls
./      ../     os/     congratulation_on_successful_ls/        xx2/
os_shell> cd os
os_shell> ls
./      ../     hw3_submit/     os_grade.txt
os_shell> cd ..
os_shell> rmdir os
rmdir: os: Directory not empty
os_shell>
```

그림 3. 2단계 실행 화면

3단계 - cpin, cpout의 구현

3단계에서는 2단계에서 설명한 것과 같이 i-node(블록)를 할당/할당 해제하는 메커니즘이 필요합니다. 추가적으로 indirect block을 사용 하는 부분도 추가 되어야 합니다. 현재 sfs.h에 정의되어 있는 값으로는 15개의 direct block을 사용 합니다. 이는 최대 7680byte의 파일만 수용 할 수 있게 되는데 이를 확장하기 위하여 indirect block을 사용 합니다 indirect block을 사용 할 경우 (512/4)*512byte, 즉 64kbyte의 데이터를 추가적으로 수용 할 수 있습니다. cpin과 cpout의 동작을 확인해보고자 할 경우 같은 파일에 대하여 cpin과 cpout을 실행하여 호스트 PC의 diff 명령어 등을 사용하여 확인하면 됩니다.

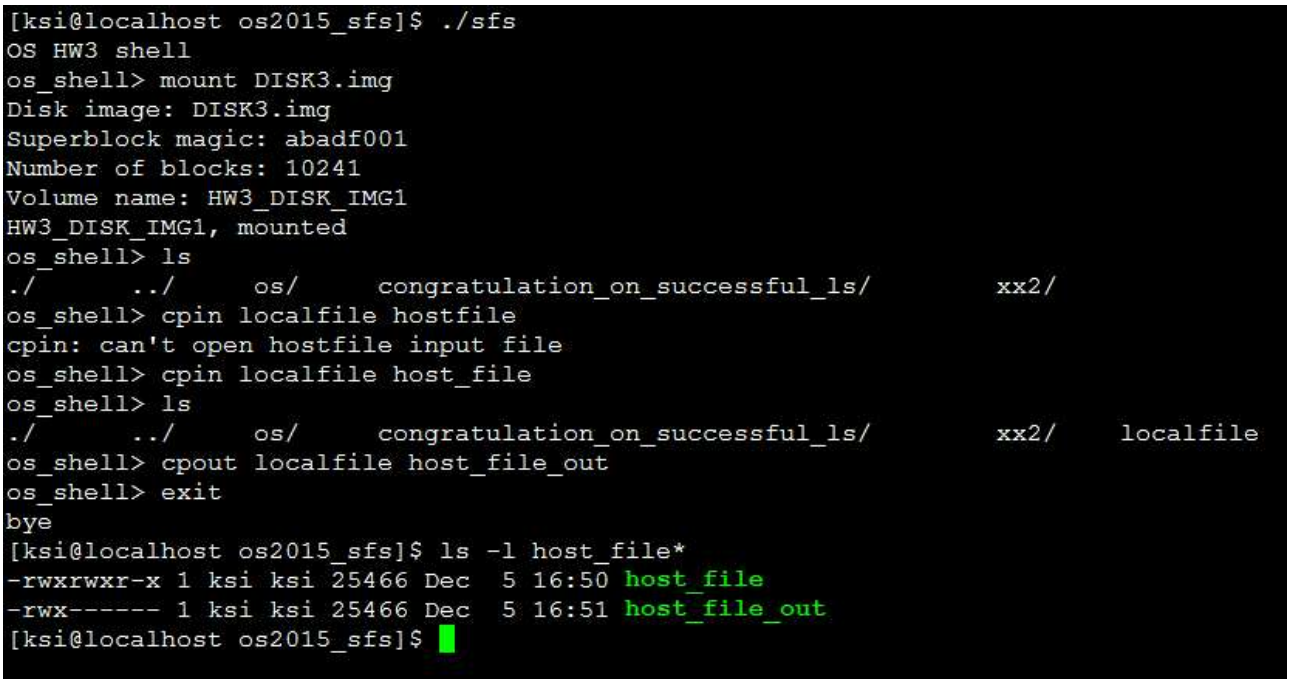

그림 4. 3단계 실행 화면

주의 사항

1.어떠한 이유, 어떠한 형태로든 치팅은 금지됩니다. 적발될 경우 이유 불문 F입니다. 필요시 추가 면담이 있습니다.

2. late는 받지 않고, 소스코드는 오로지 submit을 통해서만 제출 받습니다.

3.다른 사람의 코드와 유사도가 높은 경우 이유여하를 불문하고 감점됩니다.

4.submit 에러 (submit –l 로 제출후 반드시 확인), 소스코드 실수, 컴파일 프래그 실수 등 모든 실수는 고려 안되면 제출상태 그대로 채점합니다.

주의사항 : 제출 파일의 처음에 제출년도/과목명/과제명/학번/이름을 명시하시오.