Operating System Project #4

File System - Extensible Files / Subdirectories / Persistence

Team 8
2009-11841 최영진
2011-11749 윤태훈
2011-11642 권현석
2013-13436 김시준

목 차

1. Abstract

Pintos Project #4에서는 Pintos에 파일 시스템을 구축한다. 지난 프로젝트 User Program에서 핀토스 자체의 파일 시스템을 사용하였으나, 이는 유저 프로그램 관련 구현을 돕기 위한 것으로 기능이 매우 제한적이다. 디스크의 Block을 연속해서만 할당할 수 있기 때문에 빈 Block이 있어도 요구하는 크기보다 작으면 쓸 수 없어서 Fragmentation이 발생한다. 또한 Subdirectory 개념이 없어 Root에서만 파일을 읽고 쓸 수 있다.

본 프로젝트에서는 크게 이 두 가지 문제를 수정하여 일반적인 파일 시스템을 구현할 것이다. 또한 파일 시스템이라는 것은 메모리와 달리 시스템이 꺼져도 내용이 유지되어야 한다. 따라서 각 기능을 구현하면서 "Persistence"도 신경써야 한다. 다음 순서로 구현하였다.

1. Extensible File
2. Subdirectory
3. System Call
4. Other Implementation

2. Extensible File Design

inode의 크기가 무한하지 않기 때문에, 보통 OS에서는 여러 가지 방법으로 Disk Block에 접근하게 된다. 작은 파일의 경우 Direct Access만으로 충분하지만, 크기가 커질 경우 Pointer Block을 한 번 거치는 Single Indirect Access, 두 번 거치는 Double Indirect Access 등을 사용하게 된다.

본 프로젝트에서 파일의 최대 크기는 8MB인데, Single Indirect Access만 사용하여도 거의 모든 사이즈를 커버할 수 있기 때문에 다른 방식은 사용하지 않았다. 한 Block에 들어갈 수 있는 Block Sector 수는 128개(512 / 4)이나, inode에는 index, is_dir 등 다른 정보가 들어가야 하기 때문에 124개만 사용할 수 있었다. 그러나 124 * 124 * 512 의 Maximum Size로도 모든 테스트를 통과할 수 있었다.

```
// Only Single Indirect
// 124 * 124(126) * 512 is similar to 8MB, enough to pass tests
#define INDIRECT_SIZE 124
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    int iindex;                         /* Indirect Block Index */
    int is_dir;                         /* Is directory, 바이트 수 맞추기 위해 bool 대신 int 사용 */
    block_sector_t iblocks[INDIRECT_SIZE];  /* Indirect Blocks */
  };

struct indirect
{
    int index;                          /* Block Index */
    block_sector_t blocks[INDIRECT_SIZE];   /* Blocks */
    uint32_t unused[2];                 /* Not used */
};
```

위와 같이 struct inode_disk를 수정하였다. length, magic은 inode 처리에 필수적이고, Block index 및 디렉토리인지 여부를 나타내는 플래그를 추가하였다.

변경된 inode 타입에 맞게 기존 함수들을 변경하였다. 우선 주어진 offset에 따르는 block을 찾는 byte_to_sector 함수를 Indirect Access에 맞게 고쳤다. 예를 들면 100000 위치는 올림(100000 / 512) = 196 번째 sector이고, 올림(196 / 124) = 2번째 Indirect Block에서 72번째 Block 을 검색하면 된다.

기존에 연속된 Block만을 가져오던 로직도 효율적으로 바뀌었다. allocate_indirect 함수에서는 필요한 수의 sector를 가져오고 이를 Indirect Pointer들에 저장하는데, 이 때 free_map_allocate를 반드시 1로 호출함으로써 연속되지 않은 Block도 한 파일 저장에 쓸 수 있도록 하였다.

```
54
55 void
56 allocate_indirect(int sectors, struct inode_disk *disk_inode)
57 {
58     int i, j;
59     static char zeros[BLOCK_SECTOR_SIZE] = {0,};
60
61     int indirect_num = DIV_ROUND_UP(sectors, INDIRECT_SIZE);
62     for(i=0; i<indirect_num; i++)
63     {
64         struct indirect *indi;
65         indi = calloc(1, sizeof *indi);
66         indi->index = 0;
67
68         int isize = sectors >= INDIRECT_SIZE? INDIRECT_SIZE : (sectors % INDIRECT_SIZE);
69         for(j=0; j<isize; j++)
70         {
71             free_map_allocate(1, &indi->blocks[j]);
72             block_write(fs_device, indi->blocks[j], zeros);
73             indi->index++;
74         }
```

또한 이번 프로젝트부터는 사용자가 파일 크기 이상의 위치에 seek를 할 수 있고, 파일의 뒤에서 쓸 수도 있다. 이럴 때는 파일 크기가 확장되어야 하는데, 해당 기능을 하는 함수를 allocate_more라는 이름으로 구현하였다. 이 함수는 현재 indirect index를 참조하여 다음 index에 새로운 sector를 할당한다.

3. Subdirectory Design

제공되는 코드에서 directory.c 등 디렉토리 관련 처리를 일부 하고 있어 각 기능들을 추가하면 되나, 기본 코드는 루트 파일 사용 정도만 동작한다. Subdirectory 처리를 위해 우선 struct inode_disk에 is_dir flag를 추가하였다. 이 때 512바이트의 크기를 맞추기 위해 bool 대신 int 타입을 사용하였다. 디렉토리를 저장하는 타입은 struct dir을 그대로 사용할

수 있다. directory.c의 관련 소스들을 조금씩 고쳐서 루트 외의 디렉토리에서도 사용할 수 있도록 구현하였다.

또한 a/b/c 형태의 Path를 파싱하는 로직이 필요하다. 이는 기존에 사용해봤던 strtok 함수를 썼다. dir_from_path는 struct dir *를 리턴하며, filename_from_path는 마지막의 파일명을 리턴한다. dir_from_path의 주요 부분은 다음과 같다.

```
198     tok = strtok_r(NULL, "/", &ptr);
199     while(tok != NULL)
200     {
201         //printf("dir prev : %s // tok : %s\n", prev, tok);
202         if(strcmp(prev, ".") == 0)
203         {
204         }
205         else // ".."은 dir_entry에 삽입하여 처리
206         {
207             if( ! dir_lookup(dir, prev, &inode))
208                 return NULL;
209
210             if(inode->data.is_dir == 1)
211             {
212                 if(inode->removed)
213                     return NULL;
214                 dir_close(dir);
215                 dir = dir_open(inode);
216             }
217             else
218             {
219                 inode_close(inode);
220             }
221         }
```

상대 경로를 지원해야 하기 때문에, “.” 및 “..” 의 처리가 필요하다. 현재 디렉토리의 경우 파싱 과정에서 아무것도 처리하지 않도록 하였다. (그 전 경로를 이용한다. 처음이라면 이미 루트 혹은 Current Working Directory로 설정된다.) “..”를 구현하는 방법은 여러 가지가 있겠으나, 본 조는 루트를 제외한 디렉토리들의 맨 처음 엔트리로 ..를 넣어 구현하였다. 부모 dir_entry는 이름으로 ..를 가지고, 부모 디렉토리의 sector 정보를 저장한다. 물론 readdir 등 다른 순회 로직에서는 이 부분을 무시하도록 하였다.

```
20
21  if(parent != NULL)
22  {
23      char p[NAME_MAX+1] = "..";
24      strlcpy (p_entry.name, p, sizeof p_entry.name);
25      p_entry.inode_sector = inode_get_inumber(parent->inode);
26      p_entry.in_use = true;
27      ret = inode_write_at(c->inode, &p_entry, sizeof p_entry, 0) == sizeof p_entry;
28  }
29
```

마지막으로 Current Working Directory를 thread 구조체에 “directory”라는 이름으로 추가하였다. 이는 chdir 명령 시 바뀔 수 있으며, 상대 경로 파싱 시에 사용된다.

4. System Call Implementation

기존의 다음 시스템 콜을 변경하였다.

1) int open(const char *file)

파일 뿐 아니라 디렉토리도 open할 수 있다. Directory를 열 경우 기존의 파일 정보에 더하여 디렉토리 정보까지 저장한다. 이 역시 fd로 관리된다.

2) void close(int fd)

사용이 끝난 디렉토리는 일반 파일과 마찬가지로 닫아줘야 한다. 변경된 close는 파일과 디렉토리 모두에 대해 동작하며, 디렉토리일 경우 디렉토리 정보를 저장하기 위해 할당된 자원들도 해제하게 된다.

3) bool remove(const char *file)

디렉토리일 경우도 각 정보들을 모두 지운 뒤 삭제할 수 있도록 구현하였다. 이 때 empty가 아닌 디렉토리는 삭제가 실패한다.

새로 다음 시스템 콜이 추가되었다.

1) bool chdir(const char *path)

현재의 Working Directory를 바꾼다. path를 파싱하여 해당 디렉토리를 thread_current()의 directory로 설정한 뒤, 기존의 directory는 닫아주는 것까지 실행한다. 다음 파일 작업을 할 때는 thread_current()->directory를 통해 현재 속한 디렉토리부터 출발할 수 있다.

```
64
65 bool
66 filesys_mkdir(const char *path)
67 {
68     block_sector_t inode_sector = 0;
69     struct dir *dir = dir_from_path(path);
70     char *dir_name = filename_from_path(path);
71     struct inode *dummy;          // 같은 이름 파일 존재하는지 확인용
72
73     bool success = (dir != NULL
74                     && ! dir_lookup(dir, dir_name, &dummy)
75                     && free_map_allocate(1, &inode_sector)
76                     && dir_create(inode_sector, 16, dir)
77                     && dir_add(dir, dir_name, inode_sector));
78
79     if( ! success && inode_sector != 0)
80         free_map_release(inode_sector, 1);
81     dir_close(dir);
82     free(dir_name);
83
84     return success;
85 }
```

2) bool mkdir(const char *dir)

새로운 디렉토리를 생성한다. 내부적으로는 일반 파일을 생성하는 create -> filesys_create와 흡사한 구조로 구현되었다. 디렉토리도 일종의 파일이라고 할 수 있기 때문에 상위 디렉토리를 체크하는 등의 검증을 같이 거치며, 디렉토리 정보만 더 추가된다.

3) bool readdir(int fd, char *name)

디렉토리 안의 파일들을 하나씩 읽는다. (이를 통해 ls와 같은 기능들이 구현될 수 있다.) 디렉토리 정보가 있는 sector들을 순회하며, 이 때 ".." 는 skip한다.

4) bool isdir(int fd)

fd가 가리키는 파일이 디렉토리인지 확인한다. 본 조는 inode_disk에 디렉토리인지 여부를 저장하는 플래그를 추가하였으며, 이 값을 바탕으로 함수를 수행한다.

5) int inumber(int fd)

inode 번호를 리턴한다. 핀토스에서는 sector index를 사용한다.

4. Other Implementation

1. "a/b/c" 라는 path를 파싱할 때, a/b/까지 디렉토리이고 c는 파일로 처리하였다. 그런데 chdir 명령은 마지막 인자까지 디렉토리로 받아와서 처리해야 한다. 이를 다른 파싱 로직으로 처리할 수도 있겠지만, 여기서는 chdir일 경우 마지막에 /. 를 추가하는 방식으로 같은 파싱 함수를 사용할 수 있도록 구현하였다.
2. 삭제 관련 테스트를 검토해보면, 디렉토리 삭제에 대한 정책이 두 가지인 것 같다. 자신의 parent일 때만 삭제가 불가능하게 하는 경우가 있고, empty가 아니면 무조건 실패하게 하는 경우가 있다. 이는 실제 xNIX 시스템에서도 rm / rm -f 등의 명령으로 구분되는 부분이다. 본 프로젝트에서는 empty가 아니면 삭제가 실패하도록 구현하였다.

5. Testing Result

```
pass tests/filesys/extended/dir-mkdir-persistence
pass tests/filesys/extended/dir-open-persistence
pass tests/filesys/extended/dir-over-file-persistence
pass tests/filesys/extended/dir-rm-cwd-persistence
pass tests/filesys/extended/dir-rm-parent-persistence
pass tests/filesys/extended/dir-rm-root-persistence
pass tests/filesys/extended/dir-rm-tree-persistence
pass tests/filesys/extended/dir-rmdir-persistence
pass tests/filesys/extended/dir-under-file-persistence
pass tests/filesys/extended/dir-vine-persistence
pass tests/filesys/extended/grow-create-persistence
pass tests/filesys/extended/grow-dir-lg-persistence
pass tests/filesys/extended/grow-file-size-persistence
pass tests/filesys/extended/grow-root-lg-persistence
pass tests/filesys/extended/grow-root-sm-persistence
pass tests/filesys/extended/grow-seq-lg-persistence
pass tests/filesys/extended/grow-seq-sm-persistence
pass tests/filesys/extended/grow-sparse-persistence
pass tests/filesys/extended/grow-tell-persistence
pass tests/filesys/extended/grow-two-files-persistence
pass tests/filesys/extended/syn-rw-persistence
All 121 tests passed.
% [16:32:45] on git:project4_submit x ~/OS/prj0/src/filesys/build $
```

Pintos Project 4의 121개 테스트를 모두 통과함을 확인할 수 있다. 기존의 파일 시스템에서 구현되었던 유저 프로그램 실행 등의 기능이 잘 동작하고, 새롭게 추가된 파일 확장이나 삭제/수정 등의 파일 Operation 역시 문제 없음을 뜻한다. 또한 Subdirectory를 만들고 Directory를 이동하면서 파일 작업을 할 수 있으며, 이는 시스템 재부팅 후에도 보존됨을 알 수 있다.