운영체제 Pintos Project #4 리포트

Team05
2009-11604 정태호
2009-11779 이동우
2013-11399 박병준
2013-11431 정현진

1. Extensible File Design

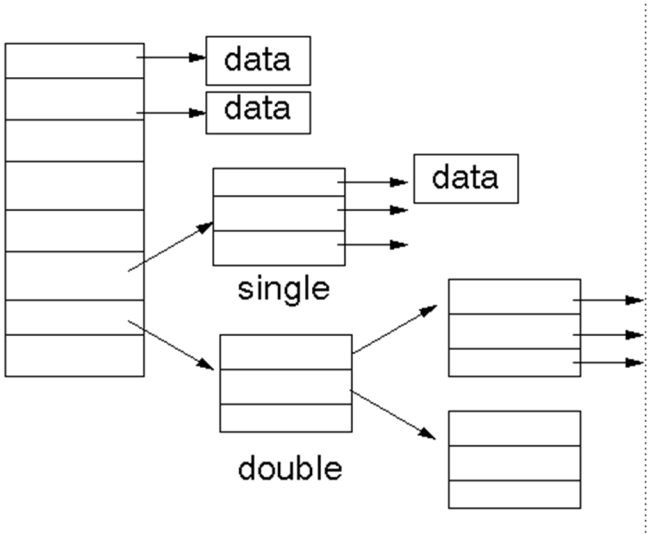
기존 pintos 파일 시스템은 파일을 하나의 extent로 취급했기 때문에 생기는 몇 가지 단점들이 있었다. Extensible File에서는 이러한 점들을 보완해 다음과 같은 기능들을 파일 시스템이 할 수 있도록 만드는 것이 목표였다.

1. 파일은 EOF 뒤에 write가 수행될 때 확장되어야 한다.
2. EOF 뒤의 position에 read가 일어날 때는 아무런 바이트도 반환하지 않는다.
3. 외부 단편화가 일어나지 않도록 해야 한다.
4. 파일은 크기 제한이 없다. (파일 파티션 사이즈인 8MB보다만 작으면 됨.)
5. EOF 뒤의 seek을 허용해야 한다.

우선 기존의 핀토스 inode에서는 시작 sector의 번호만 받은 후 그 sector의 뒤로 쭉 쓰는 방식을 사용했었는데, 이 때문에 외부 단편화에 취약했다. 이를 indexed inode를 사용하는 방식으로 바꾸기 위해 'inode_disk' 구조체에 indexed sector 번호들의 리스트와 현재 가지고 있는 sector의 개수를 저장하게 하여 관리를 할 수 있게 하였다.

```
struct inode_disk
  {
    block_sector_t sectors_of_data[14]; /* Sectors of Data blocks.
                                           First 12 sectors for direct data blocks,
                                           next 1 sector for single indirect block,
                                           last 1 sector for double indirect block */
    off_t length;                       /* File size in bytes. */
    size_t number_of_sectors;           /* The number of allocated sectors */
    bool isdir;                         /* directory or file */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[110];               /* Not used. */
  };
```

그리고 파일 크기 문제를 해결하기 위해 하나의 inode에 12개의 direct block, 1개의 single indirect block, 1개의 double indirect block을 가질 수 있게 하였다. 이 구조를 간단하게 그림으로 보면 다음과 같다.



이 때 최대 크기를 계산해 보면 direct block이 512Byte, single indirect block이 128 * 512 Byte, double indirect block이 128 * 128 * 512 Byte의 크기를 가지므로 파일이 총 12 * 512 + 128 * 512 + 128 * 128 * 512 = 8460288 Byte = 8.068 MB 만큼의 크기를 가질 수 있어 파일 크기 문제를 해결할 수 있었다.

그리고 EOF 뒤의 read 문제는 다음 그림과 같이 위치에 따른 sector 번호를 받아오는 함수에서 -1이 반환되면(즉 EOF의 뒷부분을 읽는다면) break를 하게 하여 해결하였다.

```
/* Disk sector to read, starting byte offset within sector. */
block_sector_t sector_idx = byte_to_sector (inode, offset);

if(sector_idx == -1)
  break;
```

EOF 뒤의 write 문제는 read와 같이 byte_to_sector() 함수에서 -1을 반환하면 파일 크기를 늘려야 할 때라고 판단하고 작업을 수행하였다. If문의 안 쪽에서는 필요한 크기만큼 sector를 할당 받은 뒤, 늘어난 size만큼 inode_disk(on-disk inode)의 length를 업데이트 한다. 이렇게 함으로써 EOF 뒤의 write가 수행될 때 파일 크기를 확장시켜 주었다. 이 때 파일 크기 확장은 최대 크기인 partition size 만큼만 가능하게 해 주었다.

```
if(byte_to_sector(inode, offset+size-1) == -1 && offset+size <= MAX_BYTE) {
  if(!inode_isdir(inode))
    lock_acquire(&inode->expand_lock);

  size_t total = bytes_to_sectors(size+offset);
  size_t allocated = inode->data.number_of_sectors;
  size_t need = total - allocated;

  if(!allocate_sectors(need, &inode->data))
    return 0;

  inode->data.length = size + offset;
  block_write(fs_device, inode->sector, &inode->data);

  if(!inode_isdir(inode))
    lock_release(&inode->expand_lock);
}
```

2. Subdirectory Design

기존 pintos file system에서는 파일이 모두 하나의 directory에 위치하고 있었으며 파일 이름에 14-character의 제한이 있었다. 이러한 단점들을 수정해 다음과 같은 기능을 할 수 있도록 하였다.

1. hierarchical name space를 지원해야 한다.
2. directory 또한 파일과 같이 크기를 확장시킬 수 있어야 한다.
3. 총 경로 길이가 14 character를 넘을 수 있도록 한다.
4. 각각 process 마다 current directory를 독립적으로 유지하여야 한다.
5. 상대 경로와 절대 경로를 모두 취급할 수 있어야 한다.

우선 struct thread에 current directory를 나타내는 struct dir dir* 구조체를 추가해 각 process마다 current directory를 독립적으로 유지할 수 있도록 하였다. 그리고 기존의 struct file_elem 구조체에 존재하던 struct file *file 외에도 struct dir *dir을 추가하여, file일 때와 directory일 때를 구분하여 각각 관리를 할 수 있도록 하였다.
Hierarchical name을 구현하기 위해 경로를 parsing해야 했는데 우리 조는 함수를 따로 만들어 반복적으로 call하기보다는 parsing이 필요한 함수마다 개별적으로 parsing을 수행하게 하였다. Parsing을 할 때는 strtok_r 함수를 사용하였다. 다음은 parsing 과정의 한 예로, mkdir의 parsing 과정이다.

```
// Absolute path
if(dir_name[0] == '/')
  open_dir = dir_open_root();
else // Relative path
  open_dir = dir_reopen(curr->dir);

token = strtok_r(dir_name, "/", &ptr);

// Parse the 'path'
for(token2 = strtok_r(NULL, "/", &ptr); token2 != NULL; token2 = strtok_r(NULL, "/", &ptr)) {
  struct inode *inode = NULL;
  success = dir_lookup(open_dir, token, &inode);
  if(!success)
    return false;
  dir_close(open_dir);
  open_dir = dir_open(inode);
  token = token2;
}

if(token == NULL)
  return false;
```

기존의 system call 중 수정한 것은 create, open, close, remove가 있다.
‘create’는 filesys_create() 함수를 call하는데, filesys_create()가 경로를 parsing하도록 고쳤다.
‘open’은 기존의 filesys_open 대신에 filesys_open_file()이라는 함수를 새로 만들어 call하게 하였다.
‘filesys_open_file()’은 경로를 parsing한 뒤 open하려고 하는 file이 directory일 때는 dir_open을 호출하고 file일 때는 file_open을 호출하도록 하였다.
‘close’는 file일 경우 fie_close를 호출하고 directory일 경우 dir_close를 호출하게 하였다.
‘remove’는 filesys_remove를 호출하는데, filesys_remove는 경로를 parsing할 수 있도록 수정하였다. 이 때 directory일 경우 dir_remove를 호출하는데, dir_remove에서는 여러 가지 조건들을 새로 추가하여 directory를 지울 때의 관리를 하게 했다.

새로 만든 system call로는 chdir, mkdir, readdir, isdir, inumber이 있다.
‘chdir’은 path를 인자로 받은 뒤 parsing하여 목적지 directory를 찾고, current directory의 dir을 dir_close를 호출해 닫은 다음 목적지 directory를 current thread의 dir로 바꾸어 주었다.
‘mkdir’은 경로를 parsing한 뒤, directory를 만들어 주었다. directory를 생성하는 과정은 filesys_create에서 파일을 생성하는 과정과 유사하다. 이 때 dir_create를 호출하는데, dir_create에서 각각 자신의 위치와 부모 directory의 위치를 가리키는 “.’와 “..”라는 directory들을 미리 추가하도록 수정함으로써 상대 경로일 때의 처리를 쉽게 할 수 있도록 하였다.

```
bool
dir_create (block_sector_t sector, block_sector_t parent, size_t entry_cnt)
{
  if(inode_create (sector, entry_cnt * sizeof (struct dir_entry), true)) {
    struct dir *dir = dir_open(inode_open(sector));

    // '.' is new directory itself, ".." is parent directory of new directory.
    dir_add(dir, ".", sector);
    dir_add(dir, "..", parent);
    dir_close(dir);
    return true;
  }
  else
    return false;
}
```

‘readdir’은 fd와 name을 받아 fd가 나타내는 파일이 directory일 경우 dir_entry를 하나 읽어와 name에 저장한다. 이 때 dir_create에서 만들어진 경로들인 “.”와 “..”는 무시한다.
‘isdir’, ‘inumber’는 작동 방식이 유사한데, fd를 인자로 받아 inode에 저장되어 있는 isdir 정보와 sector number 정보를 반환하도록 하였다.

또 다른 이슈로는 persistent test들과 관련된 synchronization 이슈들이 있다. 이를 구현하기 위해 inode 구조체에 dir_lock과 expand_lock을 추가했다. expand_lock은 inode_write_at() 함수에서 파일 크기를 확장시킬 때 acquire, release하여 상호 배제를 구현하였으며, dir_lock은 directory와 관련된 요소들을 변경하는 함수들에서 dir->inode의 dir_lock을 acquire, release하여 상호 배제를 구현하였다.

마지막으로 프로젝트 요구사항 1~5의 기능을 구현한 방법을 살펴 보면,
우선 1번은 여러 함수들에서 path를 parsing하면서 구현하도록 하였다. 이 때 mkdir이나 create를 할 때 각각 directory의 dir_entry에 dir_add를 통해 파일이나 directory들을 추가함으로써 hierarchical name space을 구현하였다.
2번은 directory에 dir_entry들이 추가됨으로써 크기가 늘어나게 된다.
3번은 총 경로 길이에 제한을 없앰으로써 요구 받은 경로의 길이가 14-character보다 크더라도 parsing하여 처리할 수 있도록 하였다.
4번은 struct thread 구조체의 구성 요소에 dir을 추가함으로써 구현하였다.
5번은 절대 경로는 root directory로부터 dir_entry를 반복적으로 참조함으로써 구현하였고, 상대 경로는 dir_create를 할 때 추가한 “.”와 “..” directory들을 이용해 구현하였다.