Operating System (Fall 2014) Project 4
File System

Jongwook Choi

Introduction

네 번째 pintos 프로젝트는 기존에 주어진 basic file system을 확장하여, extensible file 및 subdirectory 기능을 지원하는 file system을 구현하는 것을 목표로 한다. 크게 세 가지의 요구사항이 있는데 아래와 같다.

- Buffer Cache
- Indexed and Extensible File System
- Subdirectory

Buffer Cache는 필수 요구사항은 아니지만, PintOS 구현에 있어서 권장되는 구현 사항이므로 프로젝트에 포함시켰다. 각 항목별로 설계 및 구현, 각종 이슈사항에 대해 정리해보기로 한다.

1. Buffer Cache

Buffer cache는 file block의 read/write 연산 위에 얹어진다. 즉, `block_write(fs_device, ...)` 대신 `buffer_cache_write(...)`, `block_read(fs_device, ...)` 대신 `buffer_cache_read(...)` 를 파일시스템의 기능들이 사용하도록 구현하기만 하면 된다.

Design

Buffer cache의 크기는 매뉴얼에 따라 64 로 정하였다. Cache entry는 아래의 구조체로 표현되며, entry들의 배열로 캐시를 유지한다. `occupied` bit가 0인 것은 free slot이 되며, disk block의 어떤 sector에 어떤 내용들이 쓰여져야 하는지를 저장한다. 캐시에서 삭제되거나 flush될 때에는 disk에 write-back 되어야 하는데 이를 위하여 `dirty` bit를 유지한다.

```
struct buffer_cache_entry_t {
  bool occupied;  // true only if this entry is valid cache entry

  block_sector_t disk_sector;
  uint8_t buffer[BLOCK_SECTOR_SIZE];

  bool dirty;     // dirty bit
  bool access;    // reference bit, for clock algorithm
};

/* Buffer cache entries. */
static struct buffer_cache_entry_t cache[BUFFER_CACHE_SIZE];
```

이 상태에서, buffer cache의 구현은 straightforward 하다.

- `read()` : sector number를 통해 캐시에서 cache entry를 찾아온다. 만약 해당 entry가 없는데, 빈 슬롯이 없다면 clock algorithm을 통해서 하나의 cache entry를 evict한다. 빈 슬롯을 새로운 cache entry로 채우고 (디스크의 내용을 버퍼에 올림), buffer cache의 내용을 target memory address로 복사한다.
- `write()` : 마찬가지로 cache hit/miss 여부에 따라 eviction, entry load 를 수행한다. 그 이후 메모리의 내용을 buffer cache에 쓴다.

`filesys_done()`에 의해 buffer cache 가 닫힐 때 및, cache entry가 evict 되는데 dirty bit가 세팅되어 있다면 flush (disk에 버퍼 내용을 기록해줌) 하도록 처리하였다. 커널 쓰레드를 별도로 만들어서 (timer를 통해) 주기적으로 flush를 해줄 수도 있지만, 일단 테스트 케이스에서 필수로 요구하는 기능이 아니기 때문에 이 정도로만 구현되어 있다.

자세한 구현은 86a41246 및 272d1c1e 의 커밋 메세지와 diff 를 참고하라.

2. Indexed and Extensible File System

요구사항

기존의 파일시스템에서는, 하나의 file은 single extent (연속된 block sectors) 로만 이루어져 있었기 때문에 external fragmentation이 발생할 수 있고 파일 크기를 늘리는 것이 어려웠다. 이를 위해 indexed inode structure를 구현하여 이 두 가지 문제를 해결하도록 한다.

즉, 파일은 (물리적 위치가 연속적일 필요는 없는) 여러개의 block sector 들로 구성되고 이들의 sector number가 inode block에 저장되는 것이다. 요구사항에 따르면, 파일 크기가 8MB 이상은 지원되어야 하므로, $8 \times 2^{20}/512 = 16384$ 개 이상의 block을 지원해야 한다. inode block의 크기도 512B 이고, sector number는 4byte 이므로 최소한 doubly-indirect block 스킴을 사용해야 한다.

Design

한 파일의 inode 블록은, 123 개의 direct block 과, 1개의 indirect block, 그리고 1개의 doubly indirect block 으로 구성되어 있다. 하나의 indirect block 은, 128개의 block (을 가리키는 포인터) 으로 구성되어 있으므로 총 128 개의 sector 를 담을 수 있고, doubly indirect block은 $128^2 = 16384$ 개의 sector 를 담을 수 있다. 따라서 하나의 file이 가질 수 있는 최대 크기는, $123 + 128 + 128^2 = 16635$ sectors 이고, 약 8.12MB 에 해당한다.

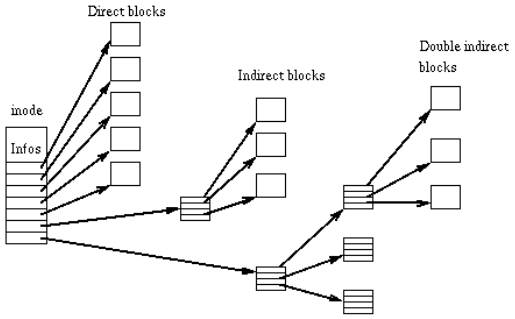

Figure 1: The inode structure.

```
struct inode_disk
  {
    /** Data sectors */
    block_sector_t direct_blocks[DIRECT_BLOCKS_COUNT];
    block_sector_t indirect_block;
    block_sector_t doubly_indirect_block;

    bool is_dir;
    off_t length;                       /* File size in bytes. */
```

```
    unsigned magic;                     /* Magic number. */
  };

struct inode_indirect_block_sector {
  block_sector_t blocks[INDIRECT_BLOCKS_PER_SECTOR];
};
```

Implementation

내부 구현에서는, 특정 offset 이 어떤 block sector 에 저장되어 있는지를 아는 것이 가장 핵심이 된다. 따라서 논리적 sector index (0 이상 16635 미만의 값이 유효) 를 inode block을 참고하여 물리적 block sector number로 변환해주는 매핑이 필요한데, 이 동작은 `index_to_sector()` 에 구현되어 있다. 당연히 direct block, indirect block, doubly indirect block 의 순서대로 block 들이 할당되어 있으므로 이에 맞추어 변환을 하면 되고, indirect block 들의 경우는 중간에 indirect block table을 갖고오기 위하여 (`inode_indirect_block_sector` 의 내용), disk read operation 을 해야할 수도 있다.

그리고, 변경된 위의 scheme에 따라 allocation 및 deallocation 하는 부분도 올바르게 변경되어야 한다. inode allocation 시에는, file size로부터 block sector 의 갯수를 알 수 있고, 따라서 비슷한 방법으로 미리 direct block, indirect block, doubly indirect block 들을 할당해 둔다. (처음부터 마지막 block 까지 모두 디스크 block을 allocate하고, 남은 block 들은 unassigned (0) 으로 남겨둔다) 비슷하게 inode deallocation 시에도, inode에 file size가 기록되어 있으므로 적절하게 할당된 모든 block들을 해제해 준다. 이것의 구현 역시 굉장히 straightforward 하므로 commit `c5b441dd` 를 참고하자.

Extensible File

Indexed block 구조가 있으므로 더이상 external fragmentation 도 발생하지 않는다. 또한 파일의 EOF 뒤에 write 가 일어날 때, 파일의 크기가 grow 하는 것도 매우 간단하게 처리가 가능하다.

먼저, 파일의 EOF 뒤에 쓰는 것을 감지해야 한다. 이것은 `inode_write_at()` 에서, 쓰려고 하는 마지막 byte에 해당하는 sector가 없는지를 (-1) 검사함으로써 알 수 있다. 이 경우에는, 새로운 파일 크기 (마지막 byte) 에 따라 inode block 들을 확장하면 된다. 이 때 일어나는 동작들은 inode block allocation 과 거의 동일하며, 이미 할당된 block들은 그대로 두고 맨 뒤의 할당되지 않은 block 들을 모두 새롭게 할당하는 방식으로 구현할 수 있다. 마찬가지로 새롭게 할당된 disk block들은 모두 zero-fill 된다. 자세한 구현은 역시 commit `08fce8fe` 을 참고하자.

3. Subdirectories

기존의 filesystem은 root directory에 모든 파일이 존재하는 단일 디렉토리 구조였지만, 이번 프로젝트에서는 평범한 UNIX 처럼 subdirectory가 지원되도록 구현되어야 한다. 먼저 요구사항을 간단하게 요약해보자.

- directory 의 parent-child relationship을 저장할 수 있는 구조를 설계한다.
- 기존의 파일시스템 동작 (디렉토리가 아닌 파일의 open/close/read/write)이 모두 잘 동작한다. 이 때 디렉토리를 포함한 path(절대경로 및 상대경로)가 오더라도 잘 동작해야 하며, UNIX에서처럼 '.', '..' 와 같은 특수한 디렉토리 이름도 지원해야 한다.
- `open()`, `close()` 시스템 콜이 directory 를 열고 닫을 수 있도록 해야 한다. `remove()` 또한 empty directory를 지울 수 있도록 수정해야 한다.
- 디렉토리 관련 system call인, `mkdir()`, `chdir()`, `readdir()`, `isdir()`, `inumber()` 를 구현한다.

설계상의 변화

- Parent directory: 첫 번째 directory entry에 parent directory 정보를 저장한다.
  - 기존의 PintOS의 디렉토리 구조를 살펴보면, 하나의 directory 를 저장하는 disk block 에는 복수 (`entry_cnt`) 개의 `dir_entry` 들이 저장되어 있다(`dir_create()`). 이것들이 해당 디렉토리 밑에 있는 파일 혹은 디렉토리를 의미한다. (inode sector number로 참조됨)
  - Parent directory 정보를 저장하기 위해서, 이 중 첫 번째 directory entry를 parent directory로 할당한다. 첫번째 entry에 들어있는 sector number가 바로 parent directory의 sector number가 된다. 따라서 directory entry를 순회하는 부분들은 모두 이에 맞게 수정되었고, `dir_lookup()` 에서 parent directory path segment ('`..`')를 lookup 할 때에는, 이 entry를 읽어들임으로써 parent directory로 따라갈 수 있도록 처리해 주었다.
- (process 별) file descriptor 에, open directory 를 가리키는 핸들인 `dir` 이 추가되었다.
  - 시스템 콜로 directory를 open하기 위해 필요한 정보이다. 기존에는 open file의 핸들인 `file` 밖에 없었고, 디렉토리를 연 경우에만 셋팅된다. 만약 plain file을 연 경우라면 `NULL`.
- Process (또는 thread)의 CWD를 기록하기 위해, `struct thread`에 cwd를 기록한다.
  - 이것 또한 open directory (`struct dir*`) 이며, 프로세스가 생성될 때 open되어 process에 저장되고 process가 exit할 때 close 된다.
  - 프로세스가 아닌 커널 쓰레드 (e.g. main thread) 는 `NULL`로 셋팅될 수 있지만, `cwd`가 필요한 곳(relative path를 resolving 하는 등)에서 root directory를 참조하도록 설정한다.
- `inode` 디스크 block에, 해당 파일이 directory인지 아닌지를 나타내는 bit (`is_dir`)를 저장한다.
  - `isdir()` 시스템 콜 등에서, 특정 inode가 가리키는 파일이 directory인지 판단하는 것이 필요하다.

구현: Utilities

- `split_path_filename()` : 유틸리티 함수로서, `path` 문자열에서 directory 와 filename 을 분리해준다. (e.g. `path/to/file.txt` → `path/to`, `file.txt`) 각종 디렉토리 관련 함수들에서 사용된다. 자세한 것은 commit `78f5400d`를 참고하라.
- `struct dir* dir_open_path()` : 주어진 `path` 문자열이 가리키는 directory를 열고, 그것의 `dir*` 핸들을 반환한다. 이 함수의 구현은, `strtok_r`를 이용하여 '/'를 구분자로 디렉토리들을 나누고, `dir_lookup()`을 통해 디렉토리 structure를 따라내려가는 것으로 되어있다. 자잘한 부분으로, 이미 삭제된 디렉토리를 open하려 하는 경우가 있는데 이에 대한 처리도 해 주어서 이미 삭제된 디렉토리는 open이 실패하도록 하였다.

구현: System calls

먼저 기존의 system call 에 수정된 내용들을 정리해보자. 자세한 구현은 `3d251e89`를 참고하라.

- `open()` : 디렉토리를 open할 수 있도록 수정되어야 한다.
  - 이 시스템 콜은 `filesys_open()` 으로 위임 (delegate) 된다. `path`와 `filename`으로 split 되고, base directory를 lookup한다. 여기서 open하는 inode는 파일/디렉토리 구분이 딱히 없기 때문에 directory의 inode 또한 open 상태로 유지된다.
  - 앞서 file descriptor에 `dir`를 유지한다고 했는데, 앞서 open된 inode가 directory이면 대응되는 `dir` 을 file descriptor에 저장해준다.
- `close()` : 디렉토리를 close할 수 있도록 수정되어야 한다.
  - `open()`의 경우와 대칭이다. 대응되는 file descriptor를 닫을 때, 열린 파일이 디렉토리인 경우는 `dir`를 함께 닫아준다.
- `remove()` : 빈 디렉토리인 경우에 삭제가 가능해야 한다.
  - 역시 `filesys_remove()` 및 `dir_remove()` 로 delegate 된다. 특별히 추가되어야하는 구현은 없고, path splitting만 올바르게 해주면 directory entry에서 lookup된 파일 또는 디렉토리가 삭제된다. 단, 지우려고 하는 파일이 디렉토리인 경우에는 directory empty 체크를 해 주었다.
- 그 외 기존 파일 처리 system calls: 가령 open된 디렉토리에는 read, write를 할 수 없어야 한다. `fd`를 통해 file descriptor를 찾을 때, `read()`, `write()` 등과 같이 파일에만 적용가능한 연산들은 file descriptor가 가리키는 파일 객체가 디렉토리가 아닌 plain file인 경우만 검색되도록 처리를 해 주었다.

새롭게 추가된 system call 들은 아래와 같다. 자세한 구현은 `804cfd78`를 참고하라.

- `mkdir(name)`
  - `filesys_create()`로 위임된다. 여기서 일어나는 작업은 (i) inode를 생성하고, (ii) parent dir 밑에 추가한다 (`dir_add()`). inode 생성시에는 디렉토리임을 inode에 기록해야 하고 (`is_dir`), parent directory 밑에 추가될 때 신규 파일 (inode) 에 parent directory 정보를 기록해준다. 앞의 설계에서 언급했듯이, inode block 내의 첫 번째 directory entry에 parent directory의 block sector number를 기록하는 방식이다.

- `chdir(name)`
  - `filesys_chdir()`로 위임된다. 주어진 path에 해당하는 directory를 열고, 이것을 현재 `thread`의 cwd 로 바꾸어 준다 (기존의 `dir*` 는 closed.)
- `readdir(fd, name)`
  - file descriptor 와 대응되는 inode 를 찾는다. valid directory 인지 검사하고, 디렉토리 핸들인 `dir`가 있으므로 `dir_readdir()`에 위임하면 된다.
- `isdir(fd)`
  - file descriptor 와 대응되는 inode 를 찾으면, 그 안에는 directory인지 여부를 나타내는 bit가 저장되어 있으므로 해결.
- `inumber(fd)`
  - 디렉토리 별로 unique한 id를 반환하는 시스템콜인데, 자연스럽게 (inode의) block sector number를 사용하면 된다. inode의 sector number를 반환하는 `inode_get_inumber()` 라는 함수를 사용하면 straightforward 하게 구현이 가능하다.

각종 이슈들

대부분의 구현은 매우 자명하지만, 몇 가지 예외적인 케이스나 신중하게 처리해야 하는 부분들이 있었다. 몇 가지만 언급하자면 다음과 같다.

- 다른 프로세스의 CWD가 삭제되는 경우 : 매뉴얼에는 두 가지 동작이 가능하다고 나와있다. 하나는 삭제를 허용하는 것이고 (이 때에는 이 디렉토리 밑에서 file 생성 및 r/w가 불가능), 다른 하나는 삭제를 허용하지 않는 것이다. 여기서는 전자를 택했는데, 이것이 구현이 자연스럽기 때문이다. 앞서 말했듯이, directory lookup 시 이미 삭제된 디렉토리인 경우에 대한 처리를 해 주었기 때문에 이 디렉토리 밑에서 일어나는 다른 operation은 실패한다.
- 최상위(root) directory의 parent directory 처리 : 최상위 디렉토리를 생성할 때에는 parent directory를 자기 자신으로 설정하였다. 따라서 `/a/../../b` 등의 path도 올바르게 인식할 수 있다.
- 동기화: 서로 다른 프로세스가 동시에 파일을 접근하더라도 data race는 발생하지 않는다. 모든 file system (새로 추가된 디렉토리 관련 system call 포함) 관련 system call은 `filesys_lock`이라는 lock에 의해 critical section 으로 동작한다.
- 테스트 케이스에는 없는 동작이지만, child process 를 생성할 때 (`exec()`), CWD 를 상속한다. 즉, parent process 의 `dir`를 re-open 하여 child process의 CWD로 셋팅하도록 처리한다. Child process가 `start_process()`를 수행할 때, parent thread를 가져와 cwd 디렉토리(핸들)를 복제한다. 적절한 정보가 없을 때에는, CWD는 root directory가 된다.

Test Results

주어진 요구사항을 올바르게 구현하여, 모든 테스트를 통과하였다. 코드를 테스트할 때, 혹시 모를 regression을 방지하기 위해 Project 3 (VM) 의 테스트케이스 또한 포함시켰다.

```
TOTAL TESTING SCORE: 110.0%
ALL TESTED PASSED -- PERFECT SCORE

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                        Pts Max  % Ttl  % Max
----------------------------------------------- --- --- ------ ------
tests/filesys/extended/Rubric.functionality      34/ 34  30.0%/ 30.0%
tests/filesys/extended/Rubric.robustness         10/ 10  15.0%/ 15.0%
tests/filesys/extended/Rubric.persistence        23/ 23  20.0%/ 20.0%
tests/filesys/base/Rubric                        30/ 30  20.0%/ 20.0%
tests/userprog/Rubric.functionality             108/108  10.0%/ 10.0%
tests/userprog/Rubric.robustness                 88/ 88   5.0%/  5.0%
tests/vm/Rubric.functionality                    55/ 55   8.0%/  8.0%
tests/vm/Rubric.robustness                       28/ 28   2.0%/  2.0%
----------------------------------------------- --- --- ------ ------
Total                                                   110.0%/110.0%
```

이 중 `filesys/extended/dir-vine` 라는 테스트 케이스가 있는데, 이 테스트 케이스는 사향 트리처럼 리소스 할당이 실패하기 전까지 subdirectory를 반복적으로 만든다. 200 depth 이상 반복하면 성공인데, 실행 결과 962 depth 까지 가능했다. 디스크가 2MB (4096 blocks) 이므로 계산해보면 1000개 근처의 디렉토리를 만들 수 있는데, 예상한 것과 비슷한 수치의 결과가 나왔으므로 특별히 resource leak 없이 디스크의 sector들을 대부분 채웠음을 알 수 있다.

Conclusion

이로써 한 학기동안 PintOS 의 모든 프로젝트가 완성되었다. Threads, User process, Virtual Memory, Filesystem 등 OS 커널의 핵심적인 기능들을 이해하고 구현할 수 있었는데, 적절한 품질의 코드로 모든 테스트를 완벽히 통과할 수 있어서 매우 뿌듯했지만 커널의 동작과 개념에 대한 이해보다도 디테일한 구현과 트러블슈팅을 통해 깨닫게 된 점이 더욱 많아 유익했던 것 같다. PintOS, Quod Erat Demonstrandum!