Operating System (Fall 2014) Project 4
File System

Jongwook Choi

Introduction

The fourth pintos project aims to expand the existing basic file system and implement a file system that supports extensible file and subdirectory functions. There are three major requirements, as follows.

- Buffer Cache
- Indexed and Extensible File System •
  Subdirectory

Buffer Cache is not a required requirement, but is included in the project because it is a recommended implementation in PintOS implementation. Let's summarize the design, implementation, and various issues for each item.

1. Buffer Cache

The buffer cache is built on top of read/write operations on file blocks . In other words, you just need to implement file system functions to use buffer_cache_write(...) instead of block_write (fs_device, ...) and buffer_cache_read(...) instead of block_read(fs_device, ...).

Design

The size of the buffer cache was set to 64 according to the manual . Cache entries are expressed as the structure below, and the cache is maintained as an array of entries . An occupied bit of 0 becomes a free slot and stores what contents should be written to which sector of the disk block . When deleted or flushed from the cache , it must be written back to the disk, and for this purpose, the dirty bit is maintained.

```
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];
```

In this state, the implementation of the buffer cache is straightforward .

- read() : Retrieves a cache entry from the cache through sector number . If there is no corresponding entry and there are no empty slots, one cache entry is evict through the clock algorithm . Fills empty slots with new cache entries (loads the contents of the disk into a buffer) and copies the contents of the buffer cache to the target memory address . • write() :

  Likewise, performs eviction and entry load depending on whether there is a cache hit/miss . After that, the contents of the memory Write to buffer cache .

When the buffer cache is closed by filesys_done() , and the cache entry is evict and the dirty bit is set, it is flushed (recording the buffer contents to disk) . You can create a separate kernel thread and flush it periodically (via a timer) , but since it is not a required function in the test case, it is only implemented to this extent.

For detailed implementation see 86a41246 and 272d1c1e Please refer to the commit message and diff .

2. Indexed and Extensible File System

Requirements

In the existing file system, one file consisted only of a single extent (contiguous block sectors), so external fragmentation could occur and it was difficult to increase the file size. To achieve this, we implement an indexed inode structure to solve these two problems.

In other words, a file consists of several block sectors (the physical locations do not need to be contiguous), and their sector numbers are stored in the inode block . According to the requirements, a file size of 8MB or more must be supported, so 8×2 20/512 = 16384 or more blocks must be supported. The size of the inode block is 512B , and the sector number is 4 bytes , so at least the doubly-indirect block scheme must be used.

Design

The inode block of one file consists of 123 direct blocks , 1 indirect block, and 1 doubly indirect block . One indirect block consists of 128 blocks (pointers to) , so it can contain a total of 128 sectors, and a doubly indirect block can contain 1282 = 16384 sectors . Therefore, the maximum size that one file can have is 123 + 128 + 1282 = 16635 sectors , which is equivalent to approximately 8.12MB .

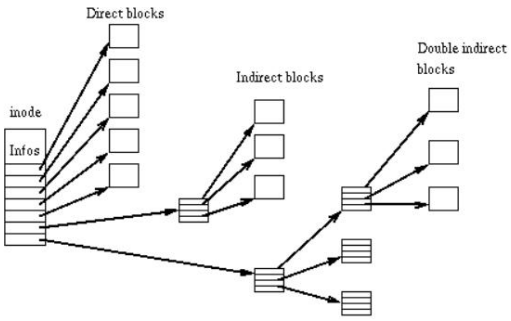

Figure 1: The inode structure.

```
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

In the internal implementation, it is key to know in which block sector a specific offset is stored. Therefore, mapping is needed to convert the logical sector index (values between 0 and 16635 are valid) to the physical block sector number by referring to the inode block . This operation is implemented in index_to_sector() . Of course, the blocks are allocated in the order of direct block, indirect block, and doubly indirect block , so you can convert them accordingly. In the case of indirect blocks , you need to perform a disk read operation to retrieve the indirect block table in the middle (contents of inode_indirect_block_sector). It may be possible.

Also, the allocation and deallocation parts must be changed correctly according to the changed scheme above . During inode allocation , the number of block sectors can be known from the file size , so direct blocks, indirect blocks, and doubly indirect blocks are allocated in advance in a similar manner. (All disk blocks are allocated from the first to the last block , and the remaining blocks are left as unassigned (0)) Similarly, during inode deallocation , all appropriately allocated blocks are released because the file size is recorded in the inode . The implementation of this is also very straightforward , so commit c5b441dd Please refer to .

Extensible File

Because there is an indexed block structure, external fragmentation no longer occurs. Also, when writing occurs after the EOF of a file, the size of the file grows can be handled very simply.

First, we need to detect writing after the EOF of the file. This can be found out in inode_write_at() by checking (ÿ1) whether there is a sector corresponding to the last byte to be written. In this case, you just need to expand the inode blocks according to the new file size (last byte) . The operations that occur at this time are almost identical to inode block allocation , and can be implemented by leaving already allocated blocks as is and newly allocating all unallocated blocks at the end. Likewise, all newly allocated disk blocks are zero-filled . For detailed implementation, see commit 08fce8fe . Please refer to .

3. Subdirectories

The existing file system had a single directory structure with all files in the root directory , but in this project, it must be implemented to support subdirectories like ordinary UNIX . First, let's briefly summarize the requirements.

- Design a structure that can store the parent-child relationship of the directory. • All existing file system operations (open/close/read/write of files, not directories) work well. At this time , it must work well even if a path (absolute path or relative path) including a directory is provided , and special directory names such as '.' and '..', as in UNIX , must also be supported.

- The open() and close() system calls must be able to open and close the directory . remove() also removes an empty directory It must be modified so that it can be erased.
- Implements directory-related system calls, mkdir(), chdir(), readdir(), isdir(), and inumber() .

changes in design

- Parent directory: Stores parent directory information in the first directory entry .
  - Looking at the existing PintOS directory structure, multiple (entry_cnt) dir_entries are stored in a disk block that stores one directory (dir_create()). These refer to files or directories under that directory. (referred to as inode sector number)
  - To save parent directory information, assign the first directory entry as the parent directory . The sector number in the first entry becomes the sector number of the parent directory . Therefore, all parts that traverse the directory entry were modified accordingly, and when looking up the parent directory path segment ('..') in dir_lookup() , this entry was read so that it could be followed to the parent directory .

- In the file descriptor (by process) , dir , a handle pointing to an open directory , was added.
  - This is information needed to open the directory with a system call . Previously, there was only file, which was the handle of an open file , and it was set only when a directory was opened. NULL if a plain file is opened .
- To record the CWD of a process (or thread) , record cwd in struct thread .
  - This is also an open directory (struct dir*) , and is opened when a process is created, stored in the process, and closed when the process exits . – Kernel threads other than processes (eg main thread) can be set to NULL , but where cwd is needed (relative Set the root directory to be referenced in (path resolving , etc.) .
- In the inode disk block, a bit (is_dir) indicating whether the file is a directory is stored.
  - In isdir() system calls, etc., it is necessary to determine whether the file pointed to by a specific inode is a directory .

Implementation: Utilities

- split_path_filename() : A utility function that separates directory and filename in the path string. (eg path/to/file.txt ÿ path/to, file.txt) Used in various directory-related functions. For details, see commit 78f5400d. Please note.

- struct dir* dir_open_path() : Opens the directory pointed to by the given path string and returns its dir* handle. The implementation of this function uses strtok_r to divide directories using '/' as a delimiter and follows the directory structure down through dir_lookup() . As a minor detail, there are cases where an attempt is made to open a directory that has already been deleted, and this has also been handled so that opening a directory that has already been deleted fails.

Implementation: System calls

First, let's summarize the changes made to the existing system call . See 3d251e89 for detailed implementation. Please note.

- open() : Must be modified to open the directory .
  - This system call is delegated to filesys_open() . It is split into path and filename , and looks up the base directory . Since the inode opened here does not have a clear file/directory distinction, the inode of the directory also remains open .
  - It was said earlier that dir is maintained in the file descriptor , but if the previously opened inode is a directory, the corresponding dir is stored in the file. Save it in descriptor .
- close() : Must be modified to close the directory .
  - It is symmetrical to the case of open() . When closing the corresponding file descriptor , if the open file is a directory, dir is also added . Close it.
- remove() : If it is an empty directory, deletion must be possible.
  - It is also delegated to filesys_remove() and dir_remove() . There is no special implementation that needs to be added, and as long as path splitting is done correctly, the file or directory looked up in the directory entry is deleted. However, if the file to be deleted is a directory, the directory empty check was performed.
- Other existing file processing system calls: For example , reading and writing should not be possible in an open directory. When searching for a file descriptor through fd , operations applicable only to files, such as read(), write(), etc., are processed so that they are searched only when the file object pointed to by the file descriptor is a plain file , not a directory.

The newly added system calls are as follows. See 804cfd78 for detailed implementation. Please note.

- mkdir(name)
  - Delegated to filesys_create() . What happens here is (i) creating an inode , and (ii) adding it under the parent dir (dir_add()). When creating an inode, the directory must be recorded in the inode (is_dir), and when added under the parent directory, the parent directory information is recorded in the new file (inode) . As mentioned in the previous design, the block sector number of the parent directory is recorded in the first directory entry within the inode block .

- chdir(name)
  - Delegated to filesys_chdir() . Open the directory corresponding to the given path and use it as the cwd of the current thread. Change it (the existing dir* is closed.)
- readdir(fd, name)
  - Find the inode corresponding to the file descriptor . Check if it is a valid directory , and dir , the directory handle, is So, you can delegate to dir_readdir() .
- isdir(fd)
  - When an inode corresponding to a file descriptor is found, a bit indicating whether it is a directory is stored in it. So solved.
- inumber(fd)
  - This is a system call that returns a unique ID for each directory , so you can naturally use the block sector number (of the inode) . It can be implemented straight-forward by using a function called inode_get_inumber(), which returns the sector number of the inode .

various issues

Most of the implementation is pretty self-explanatory, but there were a few exception cases and parts that needed to be handled carefully. Just a few things To mention, it is as follows:

- When the CWD of another process is deleted: The manual indicates that two operations are possible. One is to allow deletion (at this time, file creation and r/w under this directory are not possible), and the other is to not allow deletion. I chose the former here because it is a more natural implementation. As mentioned before, since directory lookup handles the case where the directory has already been deleted, other operations that occur under this directory will fail.
- Processing of the parent directory of the top-level (root) directory : When creating the top-level directory, the parent directory is It was set to self. Therefore, paths such as /a/../../b can also be recognized correctly.
- Synchronization: Even if different processes access files at the same time, data race does not occur. All file system related system calls (including newly added directory related system calls) operate as critical sections by a lock called filesys_lock .
- Although this is not an action in the test case, when creating a child process (exec()), CWD is inherited. In other words, the dir of the parent process is re-opened and set as the CWD of the child process . When the child process performs start_process() , it retrieves the parent thread and duplicates the cwd directory (handle) . If appropriate information is not available, CWD becomes the root directory .

Test Results

All tests were passed by correctly implementing the given requirements. Prevent unexpected regression when testing code

For this purpose , test cases of Project 3 (VM) were also included.

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

Among these , there is a test case called filesys/extended/dir-vine , which uses resource allocation like Musk Tree.

Create subdirectories repeatedly until failure . If you repeat more than 200 depth , it is successful, but the execution result is up to 962 depth.

It was possible. Since the disk is 2MB (4096 blocks), you can create about 1000 directories, which is not as expected.

Since similar numerical results were obtained, it can be seen that most of the disk sectors were filled without any particular resource leak .

Conclusion

As a result, all PintOS projects were completed during one semester . Threads, User process, Virtual Memory, Filesystem

I was able to understand and implement the core functions of the OS kernel, and was able to completely pass all tests with appropriate quality code.

I was very proud of this, but even more so was what I learned through detailed implementation and troubleshooting rather than understanding the operation and concepts of the kernel.

I think it was very beneficial. PintOS, Quod Erat Demonstrandum!