CS 130 Project 4: File Systems Design Document

Tianyi Zhang
2018533074
zhangty2@shanghaitech.edu.cn

Haoran Dang
2018533259
danghr@shanghaitech.edu.cn

Derun Li
2018533152
lidr@shanghaitech.edu.cn

0 PRELIMINARIES

0.1 Preliminary Comments

No preliminary comment for this project.

0.2 References

- Pintos Guide by Stephen Tsung-Han Sher: https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5e1bb4809e4b0a78012be132/1578874001361/Sher%282016%29_Pintos_Guide

1 INDEXED AND EXTENSIBLE FILES

1.1 Data Structures

1.1.1 Copy here the declaration of each new or changed `struct` *or* `struct` *member, global or static variable,* `typedef`*, or enumeration. Identify the purpose of each in 25 words or less.*

- In file `filesys/inode.c`

```
/* Number of direct blocks in an inode. */
#define DIRECT_BLOCK 12
/* Number of indirect blocks stored
   in a sector. */
#define INDIRECT_BLOCK \
    (BLOCK_SECTOR_SIZE / \
    (sizeof (block_sector_t)))
/* Maximum number of sectors in an inode */
#define MAXIMUM_SECTORS_IN_INODE \
    DIRECT_BLOCK + INDIRECT_BLOCK + \
    INDIRECT_BLOCK * INDIRECT_BLOCK

...

/* inode operation lock. */
struct lock inode_extension_lock;

...

struct inode_disk
{
    /* Direct and indirect blocks.
       - DIRECT_BLOCK blocks
       - 1 indirect block
       - 1 double indirect block */
    block_sector_t blocks[DIRECT_BLOCK + 2];

    ...

    /* To meet BLOCK_SECTOR_SIZE size
       requirement. */
    char unused[BLOCK_SECTOR_SIZE
                - sizeof (block_sector_t)
                  * DIRECT_BLOCK
                - sizeof (block_sector_t)
                - sizeof (block_sector_t)
                - sizeof (off_t)
                - sizeof (unsigned)
              ];
};


/* Indirect blocks stored in a sector. */
struct inode_indirect_block_sector
{
    /* Sectors */
    block_sector_t blocks[INDIRECT_BLOCK];
};

/* Double indirect blocks stored in a
   sector. */
struct inode_double_indirect_block_sector
{
    /* Sectors */
    block_sector_t
    indirect_blocks[INDIRECT_BLOCK];
};
```

1.1.2 What is the maximum size of a file supported by your inode structure? Show your work. In this file structure, in total $12 + 128 + 128^2 = 16524$ sectors can be stored in an inode. Since each sector contains 512B data, the maximum file length is $512\text{B} \times 16524 = 8.068\text{MiB}$.

1.2 Synchronization

1.2.1 Explain how your code avoids a race if two processes attempt to extend a file at the same time. The operation of extension of file must be done with lock `file_extension_lock` acquired, which avoids the race. After one have acquired a lock, it will double check whether file extension is needed, to avoid the situation where the file is extended when it is waiting for the lock.

1.2.2 Suppose processes A and B both have file F open, both positioned at end-of-file. If A reads and B writes F at the same time, A may read all, part, or none of what B writes. However, A may not read data other than what B writes, e.g. if B writes nonzero data, A is not allowed to see all zeros. Explain how your code avoids this race. When reading data, function `inode_read_at` will check each byte at the very beginning, and the loop will be terminated when the

offset does not exist in the sectors, so data cannot be read when not allocated in the sectors.

1.2.3 Explain how your synchronization design provides "fairness". File access is "fair" if readers cannot indefinitely block writers or vice versa. That is, many processes reading from a file cannot prevent forever another process from writing the file, and many processes writing to a file cannot prevent another process forever from reading the file. The lock only locks the file extension process. Thus, this is determined by the schedular, which currently uses round-robin method to schedule all threads, so the fairness is achieved in this method.

1.3 Rationale

1.3.1 Is your `inode` *structure a multilevel index? If so, why did you choose this particular combination of direct, indirect, and doubly indirect blocks? If not, why did you choose an alternative inode structure, and what advantages and disadvantages does your structure have, compared to a multilevel index?* This is a two-level index which is able to carry 8MiB file as stated in section 1.1.2.

In this design, each structure contains file metadata, 12 direct blocks, a 1-level indirect block and a double indirect block. We choose this because it is much the same as the Unix File System except for lack of triple indirect blocks, and it is adequate to handle the testcases.

2 SUBDIRECTORIES

2.1 Data Structures

2.1.1 Copy here the declaration of each new or changed '`struct`*' or '*`struct`*' member, global or static variable, '*`typedef`*', or enumeration. Identify the purpose of each in 25 words or less.*

- In file

2.2 Algorithms

2.2.1 Describe your code for traversing a user-specified path. How do traversals of absolute and relative paths differ?

2.3 Synchronization

2.3.1 How do you prevent races on directory entries? For example, only one of two simultaneous attempts to remove a single file should succeed, as should only one of two simultaneous attempts to create a file with the same name, and so on.

2.3.2 Does your implementation allow a directory to be removed if it is open by a process or if it is in use as a process's current working directory? If so, what happens to that process's future file system operations? If not, how do you prevent it?

2.4 Rationale

2.4.1 Explain why you chose to represent the current directory of a process the way you did.

3 BUFFER CACHE

3.1 Data Structures

3.1.1 Copy here the declaration of each new or changed '`struct`*' or '*`struct`*' member, global or static variable, '*`typedef`*', or enumeration. Identify the purpose of each in 25 words or less.*

- In file `filesys/cache.c`

```
/* Size of buffer cache */
#define BUFFER_CACHE_SIZE 64
/* Period to flush all the cache into the
   disk */
#define BUFFER_CACHE_FLUSH_INTERVAL 20

/* Entries of buffer cache */
struct buffer_cache_entry
{
    /* Whether this sector is being used */
    bool using;

    /* Information of the cache */
    /* Sector on the disk of the cached file */
    block_sector_t sector;
    /* Dirty bit */
    bool dirty;
    /* Last access time */
    int64_t access_time;

    /* Data storage for a block */
    uint8_t buffer[BLOCK_SECTOR_SIZE];
};

/* Buffer cache entries */
static struct
buffer_cache_entry
buffer_cache[BUFFER_CACHE_SIZE];
/* Lock for buffer cache operations */
static struct lock buffer_cache_lock;
/* Flag that the buffer cache is initialzed */
bool buffer_cache_initialized = false;

/* Last time buffer cache flushed */
int64_t buffer_cache_last_flush = 30;
/* Last sector read, 0 for no need to read
   ahead */
block_sector_t
buffer_cache_last_sector_loaded = 0;
```

3.2 Algorithms

3.2.1 Describe how your cache replacement algorithm chooses a cache block to evict. This implementation uses LRU to choose the cache to evict. Each time when accessing a cache block, including reading, writing and reading ahead, the current timer ticks will be stored in `access_time` of `buffer_cache_entry`.

When eviction is needed, we find the cache blcok with the smallest `access_time` and evict it.

3.2.2 Describe your implementation of write-behind. A boolean variable `dirty` is set to `false` when data is freshly loaded, and only set to `true` when this part of cache is written.

The written data in cache are not directly written to the disk. Instead, it will be written back only during the flushing process, which will be invoked during eviction, periodically flushing and halting.

3.2.3 Describe your implementation of read-ahead. When reading a data, the sector read will be stored in `buffer_cache_last_sector_loaded`. A thread repeatedly executing function `buffer_cache_period` will keep checking whether read-ahead is needed. If `buffer_cache_last_sector_loaded` is not zero, then the function will load the next sector into the cache if it is not loaded.

Since this function is executed in a separate thread, it will be done asynchronously only when this thread is scheduled.

3.3 Rationale

3.3.1 When one process is actively reading or writing data in a buffer cache block, how are other processes prevented from evicting that block? A global lock of the whole buffer cache is implemented, and any operation on the buffer cache need to be done with the lock acquired. This prevents conflict eviction and reading/writing processes.

4 SURVEY QUESTIONS

In your opinion, was this assignment, or any one of the three problems in it, too easy or too hard? Did it take too long or too little time?

Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

Any other comments?

CONTRIBUTORS

Task		Tianyi Zhang	Haoran Dang	Derun Li
Task 1 Indexed and Extensible Files	Concept	✓	✓	✓
	Implementation	✓	✓	✓
	Debugging	✓	✓	✓
	Design Document	✓	✓	✓
Task 2 Subdirectories	Concept	✓	✓	✓
	Implementation	✓	✓	✓
	Debugging	✓	✓	✓
	Design Document	✓	✓	✓
Task 3 Buffer Cache	Concept		✓	
	Implementation		✓	
	Debugging	✓	✓	✓
	Design Document		✓	