CS130 Project 4: File System Design Document

Group 13

Tianyao YAN
yanty@shanghaitech.edu.cn

Wenfei YU
yuwf@shanghaitech.edu.cn

I. Introduction

This document answers the questions in the template for Project 4.

II. Indexed and Extensible Files

A. Data Structures

- 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 inode.h:

```c
#define DIRECT_BLOCK_NUM 25
#define INDIRECT_BLOCK_NUM 100

struct inode_disk
{
    ...
    block_sector_t direct[
        DIRECT_BLOCK_NUM];
        /* Direct blocks. */
    block_sector_t indirect[
        INDIRECT_BLOCK_NUM];
        /* Indirect blocks. */
};
```

- What is the maximum size of a file supported by your inode structure? Show your work.

  With 25 direct blocks, 25 * 512 bytes can be stored. With 100 indirect blocks, 100 * 128 * 512 bytes can be stored. So the maximum size of a file is 6566400B.

B. Synchronization

- Explain how your code avoids a race if two processes attempt to extend a file at the same time.

  If a process attempts to extend a file, it needs SYSCALL_WRITE, and in that syscall, file_lock is acquired at first. If file_lock is not released, another process cannot do the following steps to extend the file.

- 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.

  In our code, both read and write need to acquire file_lock, so A's reading and B's writing cannot exactly happen at the same time.

- 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.

  In our code, both read and write need to acquire file_lock, and this lock will be released after the process finish its read or write. So other processes can acquire the file_lock no matter whether they are acquiring the lock for read or write.

C. Rationale

- 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?

  Yes, it's a multilevel index. But we didn't choose the particular combination of direct, indirect and doubly indirect blocks. We chose to use more indirect blocks instead of the doubly indirect block. In this way, it is easier to allocate space and it will be a little bit faster than doubly indirect block to access data. However, the maximum size is smaller than the particular combination.

III. Subdirectories

A. Data Structures

- 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 inode.h:

```
struct inode_disk
{
    ...
    bool is_dir;
        /* if this is a directory. */
    ...
}
```

In thread.h:

```
struct thread_file
{
    ...
    struct dir *dir; /* Directory. */
    ...
};
struct thread
{
    ...
    struct dir *cur_dir;
        /* Current working directory.
        */
    ...
};
```

B. Algorithms

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

  We split the path first so that we can get the name of the file or directory we need to operate and which directory(may be with many levels) it is in. Absolute and relative paths differ only when dir_lookup is called. This function will help to open the right directory.

C. Synchronization

- 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.

  Both remove and create need to acquire file_lock first.

- 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?

  We allow a directory to be removed only when it is empty, so there's no need to worry about the the files or child directories in that directory. And after it is removed, other process cannot do anything to this removed directory since they cannot find it. If the directory is the current working directory, it will be NULL and it means it will let root directory be the current working directory when open a directory by path.

D. Rationale

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

  We choose to store the current working directory in struct thread and it will be changed when chdir is called since it's very convenient.

IV. Buffer Cache

A. Data Structures

- 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 cache.h:

```
#define CACHE_SIZE 64
struct cache_entry
{
    block_sector_t sector;
        /* The sector of this cache
        entry. */
    uint8_t buffer[BLOCK_SECTOR_SIZE];
        /* Data in this cache block. */
    bool valid;
        /* Whether this cache block is
        valid. */
    bool dirty;
        /* Whether this cache block is
        modified. */
    bool second_chance;
        /* To implement clock algorithm
        . */
};
```

In cache.c:

```
static struct cache_entry cache[
    CACHE_SIZE];    /* Cache. */
static struct lock cache_lock;
    /* Lock of accessing and modifing
    cache. */
static int clock_ptr;
    /* To implement clock algorithm. */
```

B. Algorithms

- Describe how your cache replacement algorithm chooses a cache block to evict.

  We use clock algorithm. The cache entries is evicted with periodicity. If a cache entry is visited before the eviction, it will get a second chance. So it will be skipped when the eviction turns to itself and its second chance will be retracted at the same time.

- Describe your implementation of write-behind.

  We create a thread to periodically write back all the cache blocks.

- Describe your implementation of read-ahead.

  We create the function cache_read_ahead() to retrieve the subsequent block into the cache automatically when a block of a file is read.

C. Synchronization

- When one process is actively reading or writing data in a buffer cache block, how are other processes prevented from evicting that block?

  We use cache_lock. Both read and write need to acquire this lock first.
- During the eviction of a block from the cache, how are other processes prevented from attempting to access the block?

  The eviction needs to acquire cache_lock, too. So other processes cannot access the cache when the cache is evicting.

D. Rationale

- Describe a file workload likely to benefit from buffer caching, and workloads likely to benefit from read-ahead and write-behind.
  1) When we access a small file for a lot of times or we modify a small part in a large file, the buffer caching will improve the efficiency by avoiding repeatedly accessing disk or repeatedly reading and writing a large file.
  2) When the user's cat deliberately presses the power button of the computer, all the dirty blocks in the cache will be lost without implementing write-behind. However, with write-behind, only the changes in a small duration will be lost.
  3) When increasing the volume of an audio file, when thread 1 is processing the data, thread 2 can fetch the next block into the cache. Then reading next block will be faster.

V. Survey Questions

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want–these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

- 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?

  The tasks of indexed and extensible files took too long time. Other two tasks are moderately difficult.
- Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

  Maybe.
- 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?

  Students need to be told which functions need to be modified or added.
- Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

  For the physical and mental health of students, TA had better provide more tips.
- Any other comments?

  No.

References

https://github.com/youcunhan/cs130-OperatingSystem-pintos
https://github.com/Pst2000/CS130-Operating-System