Design document (Project 4: File System)

Team 3

- ChaYoung You chacha@kaist.ac.kr
- HyunJin Jung jhjnav7@kaist.ac.kr

Use 3 tokens.

Preliminaries

> If you have any preliminary comments on your submission, notes for the TAs, or extra credit, please give them here.
>
> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

Indexed and extensible files

Data structures

> A1: 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.

- `filesys/inode.h`
  - `struct inode`
    - `struct list_elem elem` : Element in inode list.
    - `disk_sector_t sector` : Sector number of disk location.
    - `int open_cnt` : Number of openers.
    - `bool removed` : True if deleted, false otherwise.
    - `int deny_write_cnt` : 0: writes ok, >0: deny writes.
    - `struct lock lock` : Lock for writing data.
- `filesys/inode.c`

    - `struct inode_disk`
        - `off_t length` : File size in bytes.
        - `size_t sector_count` : Number of used disk sectors.
        - `bool is_dir` : This is directory or not.
        - `disk_sector_t parent` : Sector number of parent directory.
        - `disk_sector_t directs[INODE_DIRECT_BLOCKS]` : Direct blocks.
        - `disk_sector_t indirect` : Single indirect block.
        - `disk_sector_t double_indirect` : Double indirect block.
        - `unsigned magic` : Magic number.
        - `uint32_t unused[109]` : Not used.
    - `struct indirect_block`
        - `disk_sector_t blocks[INODE_INDIRECT_BLOCKS]` : Blocks.
- `threads/thread.h`
    - `struct thread`
        - `struct dir *dir` : Current directory.

> A2: What is the maximum size of a file supported by your inode structure? Show your work.

It has 12 direct blocks, one indirect block, and one double indirect block. One direct block can have 512 bytes, one indirect block can have 128 * 512 bytes, one double indirect block can have 128 * 128 * 512 bytes. So we can have file of 512 + 128 * 512 + 128 * 128 * 512 = 512 + 65536 + 8388608 = 8454656 bytes = 8256 KB = 8 MB size.

Synchronization

> A3: Explain how your code avoids a race if two processes attempt to extend a file at the same time.

We use `inode->lock` for synchronization. `inode_extend()` also uses that lock.

> A4: 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.

Read from or write to a file involves buffer cache. Also these operations on buffer cache acquires lock.

> A5: 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.

We use buffer cache and it uses sector as a unit. So the cache lock is acquired only for the operation of sector size.

Rationale

> A6: 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?

Since files of small size are many, statistically, when we use direct blocks, average access time will increase.

Subdirectories

Data structures

> B1: 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.

- `filesys/directory.h`
  - `struct dir`
    - `struct inode *inode`: Backing store.
    - `off_t pos`: Current position.
- `threads/thread.h`
  - `struct thread`
    - `void *esp`: ESP register.
    - `struct dir *dir`: Current directory.

Algorithms

> B2: Describe your code for traversing a user-specified path. How do traversals of

> absolute and relative paths differ?

At first, we parse the user-specified path and split to dirname and filename. Open the root directory if the dirname starts with '/', current directory otherwise. Then traverse directories one by one from that directory.

Synchronization

> B4: 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.

We use filesys lock.

> B5: 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 prevent removing an open directory by checking `open_cnt` of `inode`.

Rationale

> B6: Explain why you chose to represent the current directory of a process the way you did.

We store the current directory in each process, which is a thread. It holds open `struct dir *` for the easy usage.

Buffer cache

Data structures

> C1: 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.

- `filesys/cache.h`
  - `struct cache`
    - `uint8_t buffer[DISK_SECTOR_SIZE]`: Buffer.

        - `disk_sector_t sec_no` : Sector number of disk.
        - `bool loaded` : Cache is loaded.
        - `bool dirty` : Dirty bit.
        - `struct lock lock` : Lock for writing.
        - `struct list_elem elem` : List element.
- `filesys/cache.c`
    - `struct read_ahead_entry`
        - `disk_sector_t sec_no` : Sector number of disk.
        - `struct list_elem elem` : List element.
    - `static struct list cache_list`
    - `static struct list cache_free_list`
    - `static struct lock cache_lock`
    - `static struct list read_ahead_list`
    - `static struct lock read_ahead_lock`
    - `static struct condition read_ahead_cond`

Algorithms

> C2: Describe how your cache replacement algorithm chooses a cache block to evict.

Whenever a cache is accessed, it goes to the front of `cache_list`. When `cache_list` is full, we evict the last element of it.

> C3: Describe your implementation of write-behind.

We hold the content of cache until it is evicted. Also we flush cache for each `CACHE_WRITE_BEHIND_INTERVAL` time ticks, which is 50.

> C4: Describe your implementation of read-ahead.

We create a new thread listening `read_ahead_cond` for any insertion to `read_ahead_list`, and whenever we read a sector, add next sector to `read_ahead_list` if it is not in `cache_list`.

Synchronization

> C5: When one process is actively reading or writing data in a buffer cache block, how are other processes prevented from evicting that block?

`cache_read()`, `cache_write()`, and `cache_read_ahead()` acquires `cache_lock`,

and eviction is occurred only on `cache_insert()`, which is only called by that three functions.

> C6: During the eviction of a block from the cache, how are other processes prevented from attempting to access the block?

As we mentioned above, eviction is occurred only on `cache_insert()`, which is handled under `cache_lock`.

Rationale

> C7: Describe a file workload likely to benefit from buffer caching, and workloads likely to benefit from read-ahead and write-behind.

When we access a small set of disk sector frequently, buffer cache will provide benefits. Instead of reading from hardware disk, accessing memory will be much faster. When we does very frequent small I/O operation, instead of writing to disk every time, doing write-behind will be cheap. Also when we access continuous sectors, read-ahead will provide benefits.

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 semesters 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?