Project 4: File Systems

Preliminaries

> Fill in your name and email address.

Hao Wang tony.wanghao@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, please give them here.

On my PC my code achieved 94.7%, failing syn-rw and syn-rw-persistence.

> 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.

In `inode.c/struct disk_inode`:

```
    bool dir;                                   /**< True if is directory */
    block_sector_t doubly_indirect_block;       /**< Doubly indirect block. */
    block_sector_t
    direct_blocks[DIRECT_BLOCK_NUM];            /**< Direct blocks. */
```

In `inode.c/struct inode`:

```
    int read_length;                    /**< Current length visible to read */
    struct lock lock;                   /**< Lock for extension of file */
```

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

The maximum size of a file supported is (124 (num of direct blocks) + 128 * 128 (one doubly indirect block)) * 512 (block size) = 8.06MB.

SYNCHRONIZATION

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

The `lock` in `struct inode` is acquired in `inode_write_at` each time when a extension is needed. Thus, if two processes attempt to extend the file at the same time, the lock would keep the extension atomic.

Additionally, since the `inode_extend` does not extend inode if the size to be extended to is smaller than the current length, it does not matter in what order the two processes extend the file.

> 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.

When B extends the file, it changes `disk_inode->length` to the new length. However, when A tries to read from the file, it could only get the length from `inode->read_length`. This length "visible only to read" is not updated until the `inode_write_at` for B is finished. Thus, not until B finished writing all the nonzero data into the extended part of the file could A realize that the file has been extended. In this way, we ensure that in this situation A does not see all zeros.

> 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.

First, readers don't need extra synchronization, so there could only be the problem where the readers are blocked by the writers. However, the writers "block" the readers mainly by modifying the `read_length`, which is not a lock. Additionally, `read_length` gets updated once a write that extends a file is finished, so the file could still be normally read. In this way, there would not be a problem regarding fairness.

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?

My inode structure has a multilevel index. I put a doubly indirect block and many direct blocks in my inode design. The reason I do this is because that I want my inode to support 8MB files, so there must be a doubly indirect block. However, I don't want all doubly indirect blocks because the greater the level is, the more the overhead would be (more space is needed to store the index). Thus, I chose the most space-efficient design: one doubly directed block with direct blocks.

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.

In `inode.c/struct disk_inode`:

```
    bool dir;               /**< True if is directory */
```

In `syscall.h/struct opened_file`:

```
    struct dir *dir;      /**< Directory wrapper of this file. */
```

In `process.h/struct process`:

```
    struct dir *cwd;                    /**< CWD of this process. */
```

ALGORITHMS

> B2: Describe your code for traversing a user-specified path. How do traversals of absolute and relative paths differ?

I use `strtok_r` to divide the path into tokens using delimiter `"/"`. I then iterate through the tokens one by one and use `dir_lookup` each time to find the next directory. Traversals of absolute and relative paths differ only in that for absolute paths the starting point of the traversal is the root directory, but that of relative paths is the CWD of current process.

SYNCHRONIZATION

> B3: 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.

In `directory.c`, four functions, i.e., `dir_readdir`, `dir_remove`, `dir_add`, and `dir_lookup`, have intrinsic synchronization by acquiring the lock included in `inode`. This way, we keep the operations on the same directory (on disk) atomic.

> B4: 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?

Yes, with an assumption that this directory is empty. That process cannot create new files or use relative paths to get out of it. It also cannot use `..` or `.`. It could only use `chmod ($ABS_PATH)` to get back to the main directory tree.

RATIONALE

> B5: Explain why you chose to represent the current directory of a process the way you did.

I think it is the most natural to include a `struct dir*` in the `struct process` in order to record the CWD of a process. There are no other specific reasons.

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.

In `cache.c`:

```
struct FCE {
    block_sector_t sector_id;           /**< Sector number of this FCE*/
    uint8_t cache[BLOCK_SECTOR_SIZE];   /**< Cache slot */

    bool available;                     /**< True if this slot is empty */
    bool dirty;                         /**< True if dirty */
    bool accessed;                      /**< True if accessed recently */
    struct lock lock;                   /**< Lock for synchronization */
};
static struct FCE fct[CACHE_SIZE];      /**< Global cache table */
```

ALGORITHMS

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

I use the CLOCK algorithm. Each time, I traverse (repeatedly) the list of cache blocks and see if there is a block not accessed while setting the blocks' access bit to false after each failed inspection. Once there is a block whose access bit is false, that block is chosen for eviction, and next time, it would be the start of the next traversal.

> C3: Describe your implementation of write-behind.

Each time we write to a sector `filesys_cache_write` first get that sector into the cache. Then it directly modify the cache. It does not write back to the disk until an eviction happens and `filesys_cache_flush` flushes the cache slot back to the disk.

> C4: Describe your implementation of read-ahead.

Each time we read from a sector `filesys_cache_read` automatically brings the next sector into memory using `filesys_load_cache`. This process is done asynchronously with small locks. If the sector read is already the last sector in the block device, we don't do read-ahead.

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?

There is a lock in each cache entry that is acquired when reading or writing data to the corresponding cache slot. Each time the CLOCK algorithm would first check on the lock to see if there is any process accessing the slot. If so, CLOCK would not consider the slot as a candidate for eviction until the access ends.

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

As is mentioned in C6, when evicting the block the lock in the cache entry is also held, which would prevent other processes from accessing the block until the eviction ends.

RATIONALE

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

For a workload benefiting from buffer caching, consider such a workload where all the processes in the OS keeps reading from the same file. Without buffer caching, the throughput could not be higher since every process needs to take a long time to read from disk. Buffer caching reduces the latency for every repeated operation writing to and reading from disk.

For a workload benefiting from read-ahead, consider two processes reading linearly from the beginning to the end of a large file. With read-ahead, each block is already in memory when needed to be read. Additionally, the time required for reading each block in advance could overlap with the time the other process needs to read and process data from the previous sector.

For a workload benefiting from write-behind, a process could do write and read many times such that without write-behind, the disk needs to be written many times, which is very costly.