Project 3: File Systems Design Doc

Group Members

Xinyu Fu (fuxy@berkeley.edu)
James Tang (jamestang@berkeley.edu)
Lanxiang Hu (hlxde1@berkeley.edu)

Task 1: Buffer Cache

Data Structures and Functions

Define buffer cache (an array of buffer cache blocks), and related functions in `inode.h`.

```
struct buffer_block {
void* data;
bool free;
bool dirty;
bool accessed;
block_sector_t sector;
struct lock lock;
}

static struct lock buffer_cache_lock;
static struct buffer_block buffer_cache[64];

void buffer_init();
void buffer_read(block_sector_t sector, void* buffer);
void buffer_write(block_sector_t sector, void* buffer);
void buffer_flush();
int buffer_evict();
void buffer_insert(block_sector_t sector);
```

Algorithms

When the file system boots, `buffer_init()` properly initializes the buffer cache (block by block). `buffer_read()` and `buffer_write()` when data is read or written using the file system. `buffer_flush()` flushes data to disk. `buffer_evict()` evicts a block to disk based on the clock algorithm, and if the block is dirty, flush it to disk. `buffer_insert()` insert the sector from disk into a free block in the buffer cache. If there is no free block, use `buffer_evict()` to evict and get a free block.

When `inode_read_at()` executes, a read is performed by `buffer_read()`, if the desired sector is in the buffer cache, it is read into the provided buffer, and `bool accessed` is set to true. Otherwise, before reading, we perform additional steps to read the sector from the disk and insert it into the buffer cache using `buffer_insert()`.

Similarly, for `inode_write_at()`, if the desired sector is in the buffer cache, we simply write to the buffer cache and set `bool dirty` and `bool accessed` to true. Otherwise, before we write to it, we need to retrieve the desired sector using `buffer_insert()`.

Additionally, in `inode_create()` and `inode_open()`, use `buffer_read()` and `buffer_write()` to perform reads and writes instead of directly using the block device. In `inode_close()`, we will also need to evict the corresponding block and mark it free.

Synchronization

Each block in the buffer cache has a lock to ensure data consistency and prevent concurrent access to the same sector. Whenever an operation is performed on a block (read, write, eviction), its lock is acquired, so that no other process can access it. In the case when a process is loading a block, another process that needs the same block would check the buffer cache and see the block is there, and would wait to acquire the lock after the block is fully loaded.

However, when we need to evict a block using the clock algorithm, we iterate through the block and potentially change `bool accessed` of the blocks. So we also have a global lock on the buffer cache for threads to hold when they are using the buffer cache. In the case when a thread is performing disk I/O, it would release the global lock on the buffer cache and reacquire it after the I/O completes, so that independent disk I/O operations can be issued concurrently.

Rationale

We decided to use an array to represent the buffer cache instead of the list data structure because the buffer cache has a fixed size. Additionally, accessing data in an array would be faster than in the list data structure because list requires converting struct to and from list elements.

For the replacement policy of the buffer cache, we decide to use the clock algorithm which is a good approximation of the LRU policy. The reason is that the clock algorithm simply uses an `accessed` variable to keep track of old blocks. It causes less computational overhead compared to LRU, which would require a list and list manipulation to keep track of all blocks.

Task 2: Extensible Files

Data Structures and Functions

`inode.c`

```
/* Global lock for free_map */
Struct lock free_map_lock;

/* On-disk inode.  Each block must be exactly 512 bytes long. */
struct inode_disk {
 off_t length;                                   /* File size in bytes. */
 unsigned magic;                           /* Magic number. */
 block_sector_t direct[124];             /*first fill all direct pointers*/
 block_sector_t indirect;            /*then fill indirect pointer*/
 block_sector_t double_indirect;
};


bool inode_create(block_sector_t sector, off_t length) {
    If (inode_grow_file(length)) {
    // implement normal initialization procedures as the provided in the staff code;
    // `free_map_allocate()` should be called iteratively with only one sector allocated each time.
} else {
    return FALSE;
}
}

/* Release memory from cur_block to starting_block. */
inode_release(struct inode_disk, int starting_block, int cur_block) {
    // This function examines whether the blocks to be released span the following three cases so that it ensures `free
    /* Direct pointers */
    /* Indirect pointer */
    /* Doubly indirect pointer */
}


bool inode_grow_file(struct inode *inode, off_t size) {
    struct inode_disk inode_disk = inode->data;
    int cur_block = bytes_to_sectors(inode_disk.length); // Current number of blocks in inode
    int end_block = num_blocks +bytes_to_sectors(size);
int starting_block = cur_block;

if (end_block > 123 + 128 + 128 * 128) { return FALSE; }
    /* Direct pointers */
// iterate through the first layer to `inode_release` each block
```

```
    /* Indirect pointer */
    if  (end_block > 123) {
        If (inode_disk.indirect == NULL) {
            // Initialize indirect pointer block and put it in cache.
}
        // malloc buffer
        // read indirect pointer block, if not in cache, read from disk and put in cache
        block_sector_t[128] indirect_pointers = // cast buffer into array of block_sector_t

// iterate through second layer to `inode_release` each block        free(buffer);
    }

    /* Doubly indirect pointer */
    if (end_block > 123 + 128) {
        If (inode_disk.double_indirect == NULL) {
                //initialize double_indirect pointer to be block_sector_t[128] and save it in cache.
}
        // read in inode_disk.double_indirect from cache/disk. Save to buffer and cast to block_sector_t array.
        // two while loops, Outer while loop reads from cache/disk the block_sector_t array of indirect pointers. Inner
    }
    return TRUE;
}

off_t inode_write_at(struct inode* inode, const void* buffer_, off_t size, off_t offset) {
    if(offset + size >= inode->length){
//grow the file by allocating more direct/indirect pointer block
    // update inode->length to length + size;
    If (!inode_grow_file()) {
        inode_release(inode_disk, starting_block, cur_block);
return 0;
}
}
}

/* Returns the block device sector that contains byte offset POS within INODE. */
static block_sector_t byte_to_sector(const struct inode* inode, off_t pos){
//calculate which data block contains the byte offset position using the new tree structure(since we always insert into
}
```

Algorithms

In order to make the files extensible, we add the inode data structure to our implementation of `struct inode`. Specifically, `struct inode` makes use of 124 direct blocks, one indirect pointer pointing to a block of pointers with 128 blocks and one doubly indirect pointer with another layer of indirection. Building upon this data structure, we modify `inode_create` to initialize `direct_block` as an array of `sector_index`, and initialize `indirect` and `double_indirect` to NULL.

In `inode_grow_file`, when it is about to extend a file and the file size is sufficiently large to use data blocks indirect pointers, the function checks whether `indirect` and `double_indirect` are NULL and calls malloc on them accordingly if necessary to grow the file.

With this data structure and layout, our design can therefore gradually increment the file size to 8MiB at max everytime an `inode_write_at()` is called with existing file size `length` plus new file size `length` greater than the current file capacity. Once file extension is needed, `direct`, `indirect` and `double_indirect` will be called consecutively to allocate necessary disk space. Notice that we also release all aborted work if an extension fails so that disk space leakage can be minimized.

Synchronization

We will use a global lock for accessing the `open_inodes` global list in `inode_open()` and `inode_close`, while removing the global `filesys_lock` from project 1.

In order to maintain data consistency, changes made to `inode_disk` need to be serialized. We need to make sure to call `lock_acquire` and `lock_release` around `free_map_allocate()` and `free_map_release()` since we don't want there to be any race condition while allocating space for free_map. This takes care of the case where two threads want to extend a file at the same time, and free_map's space will be allocated sequentially.

Rationale

The pointer type distribution has the following rationale in order to support the maximum file size (2^23 B) requirement. Since our maximum file size requires 2^23 B, and the size of each data block is 2^9 B, a direct pointer can hold 2^9 B and we have have 124 direct pointers in our `inode_disk`; an indirect pointer can hold 2^7 * 2^9 = 2^16 B; a doubly indirect pointer can hold 2^7 * 2^7 * 2^9 = 2^23 B. Therefore, using one douby indirect pointer, one indirect pointer, 124 direct pointers would suffice our need.

Task 3: Subdirectories

Data Structures and Functions

`inode.c`:

```
Struct  inode_disk {
...
 bool is_dir;   /* true if it’s a directory and false if it’s a file.  */
 Block_sector_t parent_dir;  /* to find the parent of an `inode`.*/
}
Struct fd_row {
...
 bool is_dir;  /* true if it’s a directory and false if it’s a file.  */
Void *dir_or_file /*struct file or directory, which points to a struct file or a struct dir. */
}
bool inode_isdir {struct inode* inode} {
    return (inode->data).is_dir;
}
```

`thread.h`:

```
struct thread {
...
struct dir* cwd;  /* add a field to store cwd. */
}
```

Modify `filesys_create` by adding a parameter bool `is_dir`.
`bool filesys_create(const char* name, off_t initial_size, bool is_dir);`

Add a function to find the inode based on the pathway.
`inode* get_inode(char* pathway)`
`static int get_next_part(char part[NAME_MAX + 1], const char** srcp)`

Algorithms

In `get_inode`, if the pathway starts with a `\`, this is a root directory. Otherwise, we start with the cwd. We call get_next_part to tokenize `pathway`. For each token, we can check `inode` is a file or directory. If it’s a directory, we use `dir_lookup` to check the existence of it. If it’s a file, it should be the end of the pathway. Otherwise, return NULL. We skip `.` pathway. For `..`, we can use parent_dir.

We will add a syscall `chdir`, which sets the current working directory to be the directory that passed in. Call `get_inode(dir)` and check whether the returned inode is a directory. If so, change the cwd to be `dir_open(returned_inode)`.

We also need to add a syscall `readdir`, which will call `dir_readdir` in `directory.c`.
For syscall `inumber`, we first find the `file_or_dir` by iterating the fd table entry. Then,

We return `file_or_dir->inode->sector`. For syscall `mkdir`, we call `filesys_create` function and pass true as the last argument. To implement `isdir`, we just iterate through the fd table entry and check `is_dir` based on that fd.

For close syscall, we are going to check whether it's a file or directory based on fd. If it's a directory, we will call `dir_close` on that directory. In remove syscall, if the file is an empty directory and it is not a cwd of any processes, call `dir_remove`.
For open syscall, call `get_inode_by_path(file_name)`. If the inode is a file, same as before. If it's a directory, we call `dir_open` and add it to our fd entry. In exec syscall, if the cwd of the current thread is NULL, we should set it to be the root directory.
For read, write, seek and tell syscall, if the entry of `fd` is a directory, return -1.

Synchronization

File synchronization will be handled in task 1 and task 2, so we don't consider synchronization issues in this part.

Rationale

We use a struct dir* instead of [ ]char to store cwd. Compared to [ ] char, dir has a tree structure.
In addition, `inode` is a common underlying structure for files and directories, so it's easier to use `inode` to deal with operations of a directory tree structure.

Additional Question

Write-Behind

We can use a thread to periodically wake itself up and flush the buffer cache. It would iterate through blocks of the buffer cache, flush the blocks with `bool dirty == true`, and set it back to false. After flushing the dirty blocks, this thread goes back to sleep using the `timer_sleep()` function and wakes up at the next time interval.

Read-Ahead

When fetching a block of a file into buffer cache, we can fetch the next block as well. We find the next block by adding 1 to the index of the current block in inode_disk. The index may exceed the number of direct blocks. In that case we need to find the next block in indirect blocks. We spawn a child process to fetch the additional block asynchronously in the background and will not block the current process after it loads the first block.