Project File System Design

Group 48

Name	Autograder Login	Email
Ben Plate	student219	bplate@berkeley.edu
Jeffrey Liang	student50	jeffreyliang@berkeley.edu
Teddy Cho	student139	teddycho@berkeley.edu
Theophilus Pedapolu	student270	theopedapolu@berkeley.edu

Buffer Cache

Data Structures and Functions

We add files `filesys/cache.h` and `filesys/cache.c` to contain the data structures and functions for the buffer cache.

We declare the following functions in `filesys/cache.h`.

```
void cache_init(void);

/* Reads the contents of SECTOR into BUFFER via the cache. */
void cache_read(block_sector_t sector, void* buffer);

/* Writes BUFFER into SECTOR via the cache. */
void cache_write(block_sector_t sector, void* buffer);

/* Returns the data buffer in the cache entry corresponding to SECTOR.
   Can be called only by one thread at a time. */
void* cache_get_buffer(block_sector_t sector);
```

```
/* Releases BUFFER returned by cache_get_buffer to be used by
another thread. */
void cache_release_buffer(void* buffer);
```

In `filesys/cache.c`, we add the following data structures.

```
#define CACHE_SIZE 64

struct cache_entry {
  block_sector_t sector;            /* Cache tag */
  bool valid;                       /* Data is valid */
  bool dirty;                       /* Data should be written b
ack on eviction */
  uint64_t last_accessed;           /* Time of last access */
  int num_accessing;                /* Number of threads trying
to access data */
  struct condition valid_wait;      /* Wait for valid data */
  struct lock data_lock;            /* Lock on data */
  uint8_t data[BLOCK_SECTOR_SIZE]; /* Data on disk */
};

struct cache_entry buffer_cache[CACHE_SIZE];
struct lock cache_lock;
```

We also delete the field `struct inode_disk data` from `struct inode` so that its data points to an entry in our buffer cache. Whenever we need to read or write this data, we call `cache_read` or `cache_write`.

Algorithms

`void cache_init(void)` should do the following.

- Initialize all entries of `buffer_cache` to have `last_accessed` set to zero, `num_accessing` set to zero, and `dirty` set to false. This allows any entry in the cache to be able to be “evicted”. Also, call `cond_init` on each `valid_wait` and `lock_init` on each `data_lock`.

- Call `lock_init` on `cache_lock`.

We introduce the helper functions `static struct cache_entry* cache_get_entry(block_sector_t sector, bool read_on_miss)`, which returns a pointer to an entry in `buffer_cache`matching `sector` or the entry to evict based on LRU.

- Acquire `cache_lock`.
- Scan through the entries of `buffer_cache`.
    - If a cache entry matching `sector` is found and `valid` is true, we have a cache hit. Increment `num_accessing`, release `cache_lock`, and return a pointer to this entry.
    - If a cache entry matching `sector` is found but `valid` is false, we have to wait for valid data to be loaded. Increment `num_accessing`, call `cond_wait` on `valid_wait` and `cache_lock`, and return a pointer to this entry.
    - If a cache entry matching `sector` is not found, we have a cache miss. Increment `num_accessing`, set `sector` to the new sector, set `valid` to false, and release `cache_lock`.
        - If `dirty` was true, while holding `data_lock`, write `data` back to the old `sector` using `block_write`.
        - If `read_on_miss` is true, while holding `data_lock`, read the contents of `sector` into `data` using `block_read`. Reacquire `cache_lock`, set `valid` to true, and call `cond_broadcast` on `entry->valid_wait` and `cache_lock`.
        - Finally, return a pointer to this entry.

We also introduce the helper method `static void cache_release_entry(struct cache_entry* entry, bool valid, bool dirty)`, which should be called after a thread is done reading or writing to `entry`.

- Acquire `cache_lock`.
- Set `entry->dirty` to `dirty`.
- Decrement `entry->num_accessing` and set `entry->last_accessed` to `timer_ticks()`.
- If `entry->valid` is false and `valid` is true, set `entry->valid` to true and call `cond_broadcast` on `entry->valid_wait` and `cache_lock`.
- Release `cache_lock`.

`void cache_read(block_sector_t sector, void* buffer)` should read the contents of `sector` into `buffer`.

- Call `cache_get_entry(sector, true)` to get a pointer to the cache entry we will use, `entry`.
- Acquire `entry->data_lock`.
- Copy the data from `entry->data` to `buffer`.
- Release `entry->data_lock`.
- Call `cache_release_entry(entry, true, false)`.

`void cache_write(block_sector_t sector, void* buffer)` should write the contents of `buffer` into `sector`.

- Call `cache_get_entry(sector, false)` to get a pointer to the cache entry we will use, `entry`.
- Acquire `entry->data_lock`.
- Copy the data from `buffer` to `entry->data`.
- Release `entry->data_lock`.
- Call `cache_release_entry(entry, true, true)`.

`void* cache_get_buffer(block_sector_t sector)` should return the buffer corresponding to `sector` if a thread wants to perform multiple synchronized reads or writes to `sector`.

- Call `cache_get_entry(sector, true)` to get a pointer to the cache entry we will use, `entry`.
- Acquire `entry->data_lock`.
- Return `entry->data`.

`void cache_release_buffer(void* buffer)` should be called after a thread is done using the buffer returned by `cache_get_buffer`.

- Transform `buffer` into a pointer to its cache entry, `entry`, using pointer arithmetic.
- Release `entry->data_lock`.
- Call `cache_release_entry(entry, true, true)`.

In addition, we have to modify all calls to `block_read` or `block_write` with calls to `cache_read` or `cache_write`. In `filesys/inode.c`, these occur in the functions `inode_open`, `inode_read_at`, and `inode_write_at`. We also have to call `cache_init()` in `filesys_init`.

Synchronization

We have two types of locks on the buffer cache: a global `cache_lock` and a `data_lock` on each entry's data. The global lock synchronizes access and modifications to the metadata in each entry, and the data lock synchronizes access and modifications to the data in each entry. This ensures that reads or writes that go to disk (on a cache miss or eviction) do not block accesses to other blocks.

To prevent a block that is currently being accessed from being evicted, we increment that entry's `num_accessing` field while it is being accessed, and ensure that a block can be evicted only if its `num_accessing` field is zero. Access to `num_accessing` is synchronized by `cache_lock`.

To prevent other threads from accessing a block during eviction, we set its `sector` to correspond to the block that will replace it and we set `valid` to false. We do the former to prevent another thread accessing the evicted block from reading the new data, and we do the latter to prevent another thread accessing replacing block from reading invalid data. If a thread gets a cache hit when that entry's `valid` field is false, it will wait using the `valid_wait` condition variable. This scheme also prevents a thread from accessing a block before it is fully loaded and from loading a block currently being loaded into the cache into a different entry.

Rationale

We implement a fully associative buffer cache with LRU as our replacement policy. We implement a fully associative cache because its size is relatively small, and because we have to scan the whole cache anyway, we can implement LRU by tracking the time of each entry's last access and evicting the one that was used least recently. The procedure and synchronization strategy for doing this is outlined in the previous two sections.

Extensible Files

Data Structures and Functions

We modify `struct inode_disk` to use an indexed inode structure with direct, indirect, and doubly-indirect pointers, similar to Unix FFS. The addition of `is_dir` is for our implementation of subdirectories (described in the Subdirectories section).

```
#define NUM_DIRECT_POINTERS 123

struct inode_disk {
  block_sector_t direct[NUM_DIRECT_POINTERS]; /* Direct pointers. */
  block_sector_t indirect;                     /* Indirect pointer. */
  block_sector_t doubly_indirect;              /* Doubly indirect pointer. */
  off_t length;                                /* File size in bytes. */
  bool is_dir;                                 /* True if represents a directory */
  unsigned magic;                              /* Magic number. */
  uint8_t unused[3];                           /* Not used. */
};
```

We also add `struct lock open_inodes_lock` to synchronize access to the `struct list open_inodes`.

We modify the functions `inode_create`, `inode_close`, `byte_to_sector`, `inode_read_at`, and `inode_write_at` in `filesys/inode.c` to support our new inode structure and extending files.

Algorithms

We modify the signature of `inode_create` to be `bool inode_create(block_sector_t sector, off_t length, bool is_dir)` and modify the function as follows.

- While initializing `disk_inode`, set `disk_inode->is_dir` to `is_dir`.

- Replace the `if (free_map_allocate(sectors, &disk_inode->start)) { ... }` block with `cache_write(sector, disk_inode)`. There is no need to allocate blocks for the inode's contents until data is written.

We `void inode_close(struct inode* inode)` to deallocate our new inode structure on removal. Specifically, we replace the logic in the `if (inode->removed) { ... }` with the following.

- Call `cache_get_buffer(inode->sector)` to get `struct inode_disk* disk_inode`. Call `cache_release_buffer(disk_inode)` before returning.
- Iterate through `disk_inode->direct` and release each block that is not null.
- If `disk_inode->indirect` is not null
    - Get `block_sector_t* indirect` using `cache_get_buffer(disk_inode->indirect)`. Call `cache_release_buffer(indirect)` before returning.
    - Iterate through `indirect` and release each block that is not null.
- If `disk_inode->doubly_indirect` is not null
    - Get `block_sector_t* doubly_indirect` using `cache_get_buffer(disk_inode->doubly_indirect)`. Call `cache_release_buffer(doubly_indirect)` before returning.
    - Iterate through each `block_sector_t indirect_sector` in `doubly_indirect`.
        - If `indirect_sector` is null, continue.
        - Get `block_sector_t* indirect` using `cache_get_buffer(indirect_sector)`. Call `cache_release_buffer(indirect)` before the next iteration.
        - Iterate through `indirect` and release each block that is not null.
- Release `inode->sector`.

We modify the signature of helper function `byte_to_sector` to be `static block_sector_t byte_to_sector(const struct inode* inode, off_t pos, bool allocate)` and modify its logic to use the new on-disk inode layout.

- Define
    - `DIRECT_MAX` to be `NUM_DIRECT_POINTERS * BLOCK_SECTOR_SIZE`
    - `INDIRECT_CAPACITY` to be `BLOCK_SECTOR_SIZE * BLOCK_SECTOR_SIZE / sizeof(block_sector_t)`
    - `INDIRECT_MAX` to be `DIRECT_MAX + INDIRECT_CAPACITY`

    - `DOUBLY_INDIRECT_CAPACITY` to be `BLOCK_SECTOR_SIZE * BLOCK_SECTOR_SIZE / sizeof(block_sector_t) * BLOCK_SECTOR_SIZE / sizeof(block_sector_t)`
    - `DOUBLY_INDIRECT_MAX` to be `INDIRECT_MAX + DOUBLY_INDIRECT_CAPACITY`
- Call `cache_get_buffer(inode->sector)` to get `struct inode_disk* disk_inode`. Always call `cache_release_buffer(disk_inode)` before returning.
- If `pos < DIRECT_MAX`
    - Let `sector` be `disk_inode->direct[pos / BLOCK_SECTOR_SIZE]`.
    - If `sector` is not null, return `sector`.
    - If `sector` is null and `allocate` is false, return -1.
    - If `sector` is null and `allocate` is true, assign `disk_inode->direct[pos / BLOCK_SECTOR_SIZE]` to a sector newly allocated using `free_map_allocate`, zero that block, and return that sector. If `free_map_allocate` fails, return -1.
- If `pos < INDIRECT_MAX`
    - If `disk_inode->indirect` is null
        - If `allocate` is false, return -1.
        - If `allocate` is true, assign `disk_inode->indirect` to a sector newly allocated using `free_map_allocate` and zero that block. If `free_map_allocate` fails, return -1.
    - Call `cache_get_buffer(disk_inode->indirect)` to get `block_sector_t* indirect`, a pointer to a buffer containing the inode's indirect pointers. Call `cache_get_buffer(indirect)` before returning.
    - Let `sector` be `indirect[(pos - DIRECT_MAX) / BLOCK_SECTOR_SIZE]`.
        - If `sector` is not null, return `sector`.
        - If `sector` is null and `allocate` is false, return -1.
        - If `sector` is null and `allocate` is true, assign `indirect[(pos - DIRECT_MAX) / BLOCK_SECTOR_SIZE]` to a sector newly allocated using `free_map_allocate` and zero that block. If `free_map_allocate` fails, return -1.
- If `pos < DOUBLY_INDIRECT_MAX`
    - If `disk_inode->doubly_indirect` is null
        - If `allocate` is false, return -1
        - If `allocate` is true, assign `disk_inode->indirect` to a sector newly allocated using `free_map_allocate` and zero that block. If `free_map_allocate` fails, return -1.

  - Call `cache_get_buffer(disk_inode->doubly_indirect)` to get `block_sector_t* doubly_indirect`, a pointer to a buffer containing the inode's doubly indirect pointers. Call `cache_release_buffer(doubly_indirect)` before returning.
  - Set `doubly_indirect_index` to `(pos - INDIRECT_MAX) / INDIRECT_CAPACITY`.
  - If `doubly_indirect[doubly_indirect_index]` is null
    - If `allocate` is false, return -1
    - If `allocate` is true, assign `doubly_indirect[doubly_indirect_index]` to a sector newly allocated using `free_map_allocate` and zero that block. If `free_map_allocate` fails, return -1.
  - Call `cache_get_buffer(doubly_indirect[doubly_indirect_index])` to get `block_sector_t indirect`. Call `cache_release_buffer(indirect)` before returning.
  - Set `indirect_index` to `(pos - INDIRECT_MAX - doubly_indirect_index * INDIRECT_CAPACITY) / BLOCK_SECTOR_SIZE`.
  - Let `sector` be `indirect[indirect_index]`.
    - If `sector` is not null, return `sector`.
    - If `sector` is null and `allocate` is false, return -1.
    - If `sector` is null and `allocate` is true, assign `indirect[indirect_index]` to a sector newly allocated using `free_map_allocate` and zero that block. If `free_map_allocate` fails, return -1.
- Otherwise, return -1.

We modify `inode_read_at` as follows.

- In the block `if (sector_ofs == 0 && chunk_size == BLOCK_SECTOR_SIZE) { ... }`, replace the call to `block_read` with a call to `cache_read`.
- In the corresponding `else` block, replace the bounce buffer with a call to `cache_get_buffer`, followed by `cache_release_buffer`.

We modify `inode_write_at` as follows.

- In the block `if (sector_ofs == 0 && chunk_size == BLOCK_SECTOR_SIZE) { ... }`, replace the call to `block_write` with a call to `cache_write`.
- In the corresponding `else` block, replace the bounce buffer with a call to `cache_get_buffer`, followed by a call to `cache_release_buffer`. Perform reads and writes on the buffer in the buffer cache.

Synchronization

We use `open_inodes_lock` to synchronize access to the list `open_inodes`. This essentially means that only one thread can call `inode_open` at a time.

Otherwise, we use the methods in `filesys/cache.h` to synchronize access to blocks, including the blocks containing inodes.

Rationale

As stated in the project spec, doubly indirect pointers are required to support files of size 8 MiB. Our `struct inode_disk` is constructed to maximize the number of direct pointers, which minimizes the number of disk accesses required for small files. This inode structure can support files up to size $123 \times 512\ \mathrm{B} + \frac{512}{4} \times 512\ \mathrm{B} + \frac{512}{4} \times \frac{512}{4} \times 512\ \mathrm{B} = 8\,517\,120\ \mathrm{B} > 8\ \mathrm{MiB}$.

Subdirectories

Data Structures and Functions

As described above, we add the field `is_dir` to `struct inode_disk`.

```
struct inode_disk {
  /* Existing fields ommitted */
  bool is_dir; /* True if this inode represents a directory */
}
```

We create a new struct, an directory analog to the user_file struct

```
struct user_dir {
  int fd;
  struct dir* directory;
  struct list_elem elem;
}
```

We modify the PCB

```
struct process {
  /* Existing fields ommitted */
  struct dir* working_dir;      /* Current working directory f
or process */
  struct list user_directories; /* List of user_dir's */
}
```

We also define a helper function

```
dir* dir_exists(char* dir_name) {
}
```

This function first breaks `dir` into tokens with slash as the delimiter. For each token, we traverse through the inode structure, starting from the current PCB's `working_dir` and check if each token exists and forms a directory (`is_directory` is true). We invoke the `dir_lookup` and `dir_open` functions to do this. If the directory name is an absolute path, we check if it exists starting from the root directory. Return the corresponding dir* if it exists and null otherwise.

Algorithms

chdir
Call `dir_exists` on the directory name. If null return false. Otherwise, set `working_dir` to the `dir*` returned by this function

mkdir
Call `dir_exists` on the directory name excluding the last token. Find a free sector using the free_space map and call `dir_create` to create a new inode for the directory. Call `dir_add` to add this new dir to the `dir*` returned by `dir_exists`

readdir
Call `dir_open` on the file inode corresponding to the `fd` to get the `dir*` for this directory. Then call `dir_readdir` to store the name of the next directory entry.

isdir
Call `dir_exists` on the directory name and return false if it returns null and true otherwise

inumber
Find the inode corresponding to this `fd` in the `user_files` list in the current PCB and return the sector number corresponding to this inode , i.e. inode→sector`

File syscalls

open
If the `char* name` passed in is a directory name, call `dir_exists` to get the `dir*` corresponding to this directory. Then `malloc` a new `user_dir` struct holding the `dir*` with the current max_fd (num_user_files) from the PCB, incrementing `num_user_files`. Then, add this struct to the `user_directories` list in the PCB.
close
If the given fd cannot be found in the `user_files` list, search for it in the `user_directories` list and get the corresponding struct. Then, remove the struct from the `user_directories` list, call `dir_close` on the `dir*`, and free the struct.
exec
Set the `working_dir` field of the child PCB to the `working_dir` of the current PCB (i.e. the parent process)
remove
Modify the `filesys_remove` function in `filesys.c` to use the current PCB's `working_dir` to call `dir_remove` instead of the the root directory.

For the remove syscall,
If the `char* name` passed in is a directory name, check if it is the current PCB's `working_dir` or a parent directory of the `working_dir`. If so, return false and exit the syscall. Otherwise, call `dir_exists` to get the `dir*` of the given directory and, using `dir_readdir` in `directory.c` check that the directory is empty and, if true, call `dir_remove` on this `name` with the parent directory. Return false if any of the above conditions are not met.

Synchronization

The synchronization for the `chdir`, `mkdir` `isdir`, `readdir`, and `inumber` syscalls is handled by the buffer cache since we hold locks on the inode as we are scanning through them with `dir_exists`. For the rest of the filesys syscall modifications, we already hold a global file system lock on them so no additional synchronization is required

Rationale

The main idea behind this task was to create a directory analog to files (the user_dir struct and the user_directories list in the PCB). Then, for the new directory syscalls we simply find the `dir*` and call the appropriate utility function(s) in `directory.c`.

For the modifications to the existing filesyscal syscalls, we check if the name/fd passed in corresponds to a directory and follow similar logic to the file operations.

Concept check

1. One implementation strategy for a write-behind cache is to create a new kernel thread that runs in an infinite loop that flushes the cache to disk and then sleeps for a specified period of time.

   One implementation strategy for a read-ahead cache is likewise to create a new kernel thread on each request to the cache that will request the following block from disk and place it in the cache. In our proposed implementation, this would occur in the function `cache_get_entry`.