Project File System Design

Buffer Cache

Data Structures and Functions

```
struct buffer_cache_unit {
  block_sector_t sector_num; // Index of a block device sector.
  char data[BLOCK_SECTOR_SIZE]; // Actual data stored in cache unit, need to acquire lock to read
  bool dirty; // To mark whether the cache unit needs to be updated to disk, need to acquire lock for
  int use_bit_clock; // for the clock algorithm, it is either 1 or 0
  // Clock algorithm: https://www.youtube.com/watch?v=b-dRK8B8dQk
  struct lock cache_unit_lock; // lock to access fields in buffer_cache_unit
}
struct buffer_cache {
  struct buffer_cache_unit buffer_cache_units[64]; // array of buffer cache units
  int clock_index; // index of unit within the cache range from [0-63]
  struct lock cache_lock; // lock for accessing list of cache units
}
```

Algorithms

```
struct buffer_cache_unit* clock_algorithm(struct buffer_cache* bc) { //new function
  lock_acquire(&bc->buffer_lock);
  while (true) {
    buffer_cache_unit* cur_unit = bc->cache_units[bc->clock_hand];
    if (cur_unit->being_used == false && cur_unit->dirty == false){
      // update the bc->clock_hand to become (bc->clock_hand + 1) mod 64;
      lock_release(&bc->buffer_lock);
      return cur_unit;
    } else if(cur->use_bit == 1) { // this means that this has been used recently
      // acquire cache_unit_lock
      cur_unit->use_bit = 0;
      // release cache_unit_lock
      // update the bc->clock_hand to become (bc->clock_hand + 1) mod 64;
    } else { // means that it is not in use and not accessed recently so in_use
      // if the current is dirty, then we call the write to disk function because it means it is modified and we have to write back
      // before we evict it, we check if it is in use
      // set dirty to false
      // update the bc->clock_hand to become (bc->clock_hand + 1) mod 64;
      // release the buffer cache lock
      return current;
    }
  }
}
```

```
struct update_disk(struct buffer_cache_unit* unit){
```

- If `unit` is dirty, call `block_write()` and set dirty bit to false
- Acquire lock before setting dirty bit and release

```
struct buffer_cache_unit* get_buffer_cache_unit(block_sector_t
sector_num)
```

- Loops through all 64 buffer cache units in the buffer cache to find a buffer cache unit where its sector number is equal to `sector_num`
- If not found, return NULL
- Acquire lock before looping through the buffer cache array, release before returning

```
void cache_read(block_sector_t sector_num, void* buffer) {
  struct buffer_cache_unit* current = get_buffer_cache_unit(se
ctor_num);
  // Acquire current's cache unit lock
  if (current) { // Block sector exists in the cache.
    // Copy over data to buffer
    current->use_bit_clock = 1;
  } else { // Block sector does not exist in the cache.
    current = clock_algorithm(fs_device);
    // Replace current->data with disk data
    current->sector_num = sector_num;
    current->dirty = false;
    current->use_bit_clock = 1;
    // Copy over buffer_cache_unit data to buffer
  }
  // Release current's cache unit lock
}
```

```
void cache_write(block_sector_t sector_num, void* buffer) {
  struct buffer_cache_unit* current = get_buffer_cache_unit(se
ctor_num);
```

```
  // Acquire current's cache unit lock
  if (current) { // Block sector exists in the cache.
    // Copy data from buffer to cache unit's data
    current->use_bit_clock = 1;
  } else { // Block sector does not exist in the cache.
    current = clock_algorithm(fs_device);
    current->sector_num = sector_num;
    current->dirty = true;
    current->use_bit_clock = 1;
    // Replace current->data with data from buffer
  }
  // Release current's cache unit lock
}
```

Existing Functions:

```
bool inode_create(block_sector_t sector, off_t length)
```

- replace `block_write()` with `cache_write()`
- replace `block_read()` with `cache_read()`

```
struct inode* inode_open(block_sector_t sector)
```

- replace `block_read()` with `cache_read()`

```
off_t inode_read_at(struct inode* inode, void* buffer_, off_t
size, off_t offset)
```

- replace `block_read()` with `cache_read()`

```
off_t inode_write_at(struct inode* node, const void* buffer, o
ff_t size, off_t offset)
```

- replace `block_write()` with `cache_write()`
- replace `block_read()` with `cache_read()`

```
void filesys_done(void)
```

- Go through the buffer cache and call `update_disk()` on each buffer cache unit

```
void filesys_init(bool format)
```

- Allocate memory for the buffer cache
- Initialize buffer cache's lock and set the cache's `clock_index` to 0
- For each buffer cache unit in the cache: initialize the buffer cache unit's lock, set the dirty bit to false, and set the use_bit_clock to 0

Synchronization

To ensure that no data races exist and that the data is consistent within the cache, we use 2 locks. The buffer_cache lock is used for the overall cache. Whenever a function tries to modify or write to either of thiose the cache we use this lock such that only one thing at a time can access it.

Rationale

All algorithms that we add/modify will take at most O(n) time depending on the number of bytes that are read. It will also take O(1) space as we are iterating through a constant fixed size array of 64 slots and our buffer is constant size. We chose the clock algorithm over the Least Recently Used (LRU) algorithm because it is easier to implement. The clock algorithm uses a straightforward circular buffer mechanism, which avoids the complexity of constantly updating a linked list required by LRU, making it more scalable and easier to implement. For our cache design, we are building on top of the existing file system. In `inode.c`, we replace the `block_write()` and `block_read()` functions with their cache counterparts to implement our cache structure. Because our cache system builds on top of the existing file system, it is relatively easier to implement.

Anytime we need an eviction it is written to the entry, update metadeta first before bringing it it

---

Extensible Files

Data Structures and Functions

```
struct inode_disk {
  block_sector_t direct[12]; /* 12 direct pointer. */
  block_sector_t indirect; /* 1 indirect pointer. */
  block_sector_t doubly_indirect; /* 1 doubly indirect pointe
r. */
  off_t length;         /* File size in bytes. */
  unsigned magic;       /* Magic number. */
  uint32_t unused[112]; /* Not used. */
};
```

Added pointers to block sectors. The `unused` array's size is then changed to keep the struct's size at 512 bytes, the same size as a block sector.

```
struct inode {
  struct list_elem elem;  /* Element in inode list. */
  block_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. */
  lock_t file_lock;       /* Lock to ensure correctness with c
oncurrent access. */
};
```

Deleted the `data` field, we just read in the `inode_disk` from disk when needed. added a lock struct to provide protection during writes and reads from different threads.
Lock is for read, open, close, etc on the same inode struct. Add global lock to free map release and allocate.

Algorithms

File Growth

This is a crucial helper function for many subsequent inode operations. this allows the file size to change, and sectors will be released/allocated according to the target file size.

```
Given an inode_disk, and a target_size, we will try to shrink or grow the file


1. try grow the direct pointers
for direct_pointer_count, direct_pointer in enum(direct_pointers):
  if target_size is less than or equal to direct_pointer_count * sector_size
    and direct_pointer is pointing to a sector:
    free map release the direct_pointer;
  if target_size is more than direct_pointer_count * sector_size
    and direct_pointer is not pointing to a sector
    free map allocate a new sector, set direct_pointer to the newly allocated sector;


2. allocate indirect if needed
if target_size is less than the space allotted in the direct pointers
  and indirect pointer doesn't point to any page:
  update inode_disk length;
  done;
create a 512 bytes buffer filled with zeroes;
if indirect has not been allocated:
  free map allocate the 512 bytes sector for indirect;
else:
  read in the indirect to buffer;
for indirect_pointer_count, indirect_pointer in enum(buffer pointers):
  if target_size is less than or equal to (12 + indirect_pointer_count) * sector_size
    and indirect_pointer is pointing to a sector:
```

```
    free map release the indirect pointer;
    set indirect_pointer = 0;
  if target_size is more than (12 + indirect_pointer_count) * sector_size
    and indirect_pointer is not pointing to a sector:
    free map allocate, set indirect_pointer to the newly allocated sector;
if target_size is less than the space allotted in the direct pointers:
  free map release sector pointed to by indirect pointer;
  set indirect pointer to 0;
else:
  write the buffer back to the indirect sector;


3. allocate doubly indirect if needed
if target_size is less than space allotted in direct and indirect pointers
  and doubly indirect pointer doesn't exist:
  set doubly indirect pointer to 0;
  update inode_disk length;
  done;
if doubly indirect has not been allocated:
  free map allocate the 512 bytes sector for doubly indirect;
else:
  read in the doubly indirect to buffer;
for doubly_ind_ptr_cnt, doubly_ind_ptr in enum(buffer pointers):
  current_top_size = (12 + 128+ doubly_ind_ptr_cnt * 128) * sector_size
  if target_size is less than or equal to current_top_size
    and doubly_ind_ptr is pointing to a sector:
      free map release all sectors in indirect sector pointed
to by doubly_ind_ptr;
      free map release sector pointed to by doubly_ind_ptr;
      set doubly_ind_ptr to 0;
  if target_size is more than current_top_size:
    if doubly_ind_ptr is not pointing to a sector:
      free map allocate a sector, set it to doubly_ind_ptr_cn
t;
      create a new_buffer (block_sector_t[128]) for these page
s;
      for ind_pointer_cnt, ind_pointer in enum(new_buffer):
        if target_size is more than current_top_size + ind_poi
nter_cnt * sector_size:
          free map allocate new sector, write to the new_buffe
r.
      write the buffer to sector pointed to by doubly_ind_ptr_
cnt;
    else:
      read in the indirect sector pointed at doubly_ind_ptr to
a ind_buffer;
      for ind_pointer_cnt, ind_pointer in enum(ind_buffer):
        if target_size is less than or equal to current_top_si
ze +
          ind_pointer_cnt * sector_size and ind_pointer points
to a sector:
          free map release the sectored pointed to by ind_poin
ter;
        if target_size is more than current_top_size + ind_poi
nter_cnt * sector_size
          and ind_pointer is not pointing to a sector:
          free map allocate a new sector, set ind_pointer to t
he new sector
```

```
if target_size is less than space allotted in direct and indirect pointers:
  free map release sector pointed to by doubly indirect pointer;
  set doubly indirect pointer to 0;
update inode_disk length;
done;
```

If at any point, `free_map_release` fails to allocate the amount of memory needed, we will call `file_resize` and set the target size to be the original size of `inode_disk`, and return `false`.

Inode Creation
Instead of allocating sectors according to the length, first allocate only one sector for the `inode_disk`, and write the correctly set up struct to the allocated sector. Then, call the file resize function with the desired size on the `inode_disk` struct that was just created.

Inode Open
Remains mostly the same. `inode_disk` struct will not be read into memory directly to save space.

Inode Close
Update the deallocation process to deal with direct & indirect pointers. Simply call the file resize function with the target_size being 0. Free map release the `inode_disk` struct as well.

Inode Read
Since we no longer have the `inode_disk` as a part of the `inode` struct, first read in the `inode_disk` struct from disk. simply change `byte_to_sector` to update which sector to load in and read from.

Changes to `byte_to_sector`
Since the `inode_disk` has changed to adopt a sector pointer structure, `byte_to_sector` function also needs to be updated to reflect the change.

```
given the offset pos, and an inode
```

```
block_read to read the inode_disk struct from the disk
calculate the sector for the pos, pos / BLOCK_SECTOR_SIZE
go into the corresponding pointer
0 <= sector < 12: direct pointers
12 <= sector < 524: indirect pointer
524 <= sector: doubly indirect pointer
```

Inode Write
If the `offset + size` is greater than the length of `inode_disk`, we need to grow the file. call the file resize function from earlier, and simply grow the lenght of the file accordingly.

inumber(int fd)
First, get the file descriptor associated with the fd. Then, call `inode_get_inumber` on the inode associated to the file. We will modify `syscall.c` file to include this new syscall signature, and add its number to the table of syscall numbers.

Synchronization

We are adding a lock to the `inode` struct to ensure correctness on concurrent access. when the `read` or `write` syscalls are called by the user, they must acquire the lock on the `inode` associated with the file (so `fd->file->inode->lock`). All read and write operations as done with the lock acquired. If a thread cannot acquire the lock, it must block and wait. This ensures there's no read and write or read and read to the same sector. In the case where two thread tries to access different sectors, those sectors will have corresponding inodes and separate locks, so reading/writing one sector will not block operations on the other.

Rationale

Choice of # of direct pointers, indirect pointer, and doubly indirect pointer.

- We need to support files up to 8MiB, 2^23 bytes.
- Each sector is 512 bytes, 2^9 bytes.
- Total number of sectors needed for a 8MiB file: 2^23 / 2^9 = 2^14 block sector pointers.
- Usually there's 12 direct pointers for 12 block sectors.

- In an indirect pointer, you can have 2^7 pointers, one for each page.
- In an double indirect pointer, you can have 2^7 * 2^7 = 2^14 pointers. This is enough for our 8MiB files. In conclusion, 12 direct, 1 indirect, 1 double indirect pointers are needed.

Subdirectories

Data Structures and Functions

```
struct inode_disk {
  ...
  bool directory; /* Whether this is a directory or a file */
};

struct thread {
  struct dir *current_dir;/* current directory*/
}

struct file{
  int fd;
  struct thread* owned_thread;/* which thread having this hold
er*/
  struct dir* opened_dir;/* directory that this file open*/
  struct list_elem elem;/*list element*/
}
struct file_descriptor {
  struct list_elem elem; /* List element. */
  struct file* file;     /* File. */
  int handle;            /* File handle. */
};
```

Algorithms

```
open
set fd->file->owned_thread = thread_current()
check fd->file->inode->inode_disk->is_dir
call dir_open(inode_reopen(file_get_inode(file))) to open

close
check if isdir or not like open
change the fd-> opened_dir to close by calling dir_close

remove
first check if it is dir or not
by calling dir_remove to remove the dir
bool dir_remove(struct dir* dir, const char* name)
```

```
bool chdir(const char* dir)
1. first, check if the dir is root or not
2. if this is root, we close the current dir,to avoid resource
leakage and manage the system resources
3. then we change the current thread dir = dir_open_root(helpe
r function in the directory.c)
4. if this is not root directory, we first need separate the t
arget directory and directory name(create a helper function to
separate the target directory and directory name, we can name
it separate_dir this helper function should handle both relati
ve or absolute direcotry)
5. at the same time we need to make sure the subdir is a direc
tor not a file, then we call dir_close to close the current_di
r and also change the current_dir to the subdir we found
```

```
bool mkdir(const char* dir)
```

```
1. first check if dir is the root directory or not, if it is returns false

2. after calling helper function separate_dir, we will get the target directory and directory

3. also check if the target directory does it exist,return false it not exist

4. check if the name directory exists or not, return false if it is exist,

5. then we call dir_create(directory.c) and dir_add(directory.c) to create this directory
```

```
for the next three syscall, we can create another helper function to help us deal with the fd,this helper function will loop through the current thread to find  the struct holder base on the fd that is given, and return the holder

bool readdir(int fd, char* name)

1 check if name is valid

2 call helper fd_handler to get the holder

3 use file_get_inode to get the inode by this holder.opened_file if it is a dir,f->eax= dir_readdir
```

```
bool isdir(int fd)

1 check fd is valid

2 call helper fd_handler to get the holder

3 use file_get_inode to get the inode by this holder.opened_file, which return inode, and use this inode to check if dir is ture or false, then return this value by f-> eax
```

```
int inumber(int fd)

1 check fd is valid

2 call helper fd_handler to get the holder

3 use file_get_inode to get the inode by this holder.opened_file, which return inode, return the inode->sector
```

Synchronization

for all the dir and fd, we need to check if they are valid, size which is the length of the string, and memory page boundary check, which are two functions to handle this

Rationale

for the last three syscalls, we keep the fd inside each file, so the not waste to much resource during the looping, all these three follow the same logic by calling the same helper function which will be easy to understand. Same thing for mkdir and chdir.

Concept check

Write Behind

First, we can modify the `buffer_cache_unit` struct to include a field `ticks_since_last_write`, which stores the number of ticks since the last time the block was written to the cache. We can also implement a thread that wakes up periodically using `timer_sleep()`, and this thread will scan through the buffer cache and identify dirty blocks in the cache. If those blocks have been in the cache beyond an established threshold, we can write them to the disk. For synchronization purposes, we will also ensure that the thread acquires the appropriate locks when writing back to disk and modifying the `buffer_cache_unit`.

Read After

We can modify our `inode_read_at()` function to check if the following block is available, and if it is, schedule a thread for loading it into the cache. We schedule a **separate** thread because this operation should not block the current file operation of loading the first block. When handling the scheduled load, we must first check if the read-after block is already in the cache to avoid redundancy. For synchronization purposes, we will also ensure that the thread acquires the appropriate locks when writing back to disk and modifying the `buffer_cache_unit`.