Project File System Design (Group 6)

Name	Autograder Login	Email
Noah Ku	student89	noahku@berkeley.edu
Seong Hyun Park	student366	spark6015@berkeley.edu
Meng Lu	student107	m_lu@berkeley.edu
Kevin Yee	student162	kevinsyee@berkeley.edu

Buffer Cache

Data Structures and Functions

Data structures/variables added:

inode.h

```
#define MAX_CACHE_SIZE 64
#define MAX_CHANCES 2 // Can be easily modified later


struct cache_node {
  bool valid;    // Boolean to see if it's empty (true if taken, false if empty)
  bool dirty;    // Boolean to see if cache node has been modified (through write)
  void* data;    // A pointer to whatever cached data we have
  block_sector_t sector_id; // sector id when added to the cache
  int chances;   // Init as N chances, and once it hits 0 we can evict
};
```

```
struct cache_node cache[MAX_CACHE_SIZE]; // Array contains 64 cache_node
struct lock global_cache_lock; // For synchronization between threads


int saved_clock_index; // When we resume the clock, we want to start here again
```

Data structures/variables modified:

```
NONE
```

Functions added:

inode.h

```
void cache_init(); // Sets up a new array creating MAX_CACHE_SIZE cache_nodes
void cache_read(block_sector_t sector_id, off_t size, off_t offset); // Reads the cache
void cache_write(block_sector_t sector_id, off_t size, off_t offset); // Writes to cache
struct cache_node* find_cache(block_sector_t id);
void cache_evict(bool is_read, block_sector_t sector_id); // Clock eviction policy
void cache_flush(struct cache_node *n); // Flushes cache
```

Functions modified:

inode.h

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

Algorithms

Algorithms — Modified Functions

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

- **Implementation:**
  - Replace `block_read` with `cache_read`.
- **Rationale:**
  - We do this because `inode_read_at` currently directly calls `block_read`, which is also known as directly going to disk. If we call `cache_read`, which is a new function we implemented, then we can have extra checks and we will be able to look for our file in the cache before actually fetching the file in disk.
- **Complexity:**
  - **Time:** Stays the same, more details found in `cache_read`
  - **Memory:** Stays the same, more details found in `cache_read`
- **Synchronization:**
  - Handled in `cache_read`.

```
off_t inode_write_at(struct inode* inode, const void* buffer_,
off_t size, off_t offset);
```

- **Implementation:**
  - Replace `block_write` with `cache_write`
- **Rationale:**
  - *(Similar to Inode Read At)*
  - We do this because `inode_write_at` currently directly calls `block_write`, which is also known as directly going to disk. If we call `cache_write`, which is a new function we implemented, then we can have extra checks and we will be able to look for our file in the cache before actually fetching and writing the file in disk.
- **Complexity:**
  - **Time:** Stays the same, more details found in `cache_write`
  - **Memory:** Stays the same, more details found in `cache_write`
- **Synchronization:**
  - Handled in `cache_write`

Algorithms — Added Functions

```
void cache_init();
```

- **Implementation:**

    - We should initialize each index 0 - 63
        - Set each `valid` to false (for whether it's taken or not)
        - Set each `dirty` to false (for whether we used write)
        - Set each `chances` to `MAX_CHANCES`
    - Initialize a global lock
- **Rationale:**
    - This is for initializing cache_blocks in the `cache` array. We have to initialize everything before we start using the cache.
- **Complexity:**
    - **Time:** O(N) where N is the max number of cache blocks
        - Loops through 0-63, since `MAX_CACHE_SIZE` is 64
    - **Memory:** O(N) where N is the max number of cache blocks
        - Loops through 0-63, since `MAX_CACHE_SIZE` is 64
- **Synchronization:**
    - **Needed:** Synchronization is not needed for initialization.
    - **Reasoning:** We do not need it because there is no possibility that cache_init is called multiple times at once from different threads.

```
struct cache_node* find_cache(block_sector_t id);
```

- **Implementation:**
    - We start from index 0 and loop through one cycle until it finds valid == false
        - if it found return to that `cache_node`
    - return `NULL`
- **Rationale:**
    - This is a helper function that will be useful when checking to see if we can get a cache hit. Since we are going to need to know whether the file we're looking for is in the current cache in both read and write, we decided to put this in its own separate function.
- **Complexity:**
    - **Time:** O(N) where n is max number of cache block
        - Loops through 0-63
    - **Memory:** O(N) where n is max_number of cache block
        - Loops through 0-63
- **Synchronization:**
    - Not needed in this part, since we are not working with shared data

```
void cache_evict(bool is_read, block_sector_t sector_id);
```

- **Implementation:**
  - We are using the n-th chance clock policy for the buffer cache. Our `chances` is an integer because we want it to be easier to decide how many chances we will be giving for our cache replacement policy later on. We have metadata (`valid`, `dirty`, `chances`), the id of the sector, and a pointer to data.
  - Here's our clock pseudocode algorithm:
    - While loop true, starting from index `saved_clock_index`.
      - If `chances` > 0:
        - `chances` -= 1
      - Else: (when `chances` is 0)
        - break and we know what cache_node we need to modify
      - Increment `saved_clock_index`
    - Evict current `cache_node` holder
    - Call `block_read`
      - The `sector_id` argument is from `block_read`
    - Replace its values with a new `cache_node`, resetting its values to be from the arguments passed in
    - If the current `cache_node` is dirty, we call `cache_flush`
- **Rationale:**
  - We decided to go with a clock policy since it is more ideal and is not extremely complicated. Additionally, if we want to ever change the amount of chances to give for our eviction policy, it can easily be changed by setting the initial `ref_bit` integer to be any starting N.
  - Going with LRU is also a possibility, but LRU is expensive and a lot more complicated. For example, if we wanted to keep track of the timestamps or access times, we would need to use complicated data structures such as priority queues. It is an option to consider, but we feel that it's better to go with a clock policy as it will be easier to debug and it achieves the same goal while allowing simpler modifications, like with the number of chances before eviction.
- **Complexity:**
  - **Time:** O(N) - max_cache_size
  - **Memory:** O(N) - max_cache_size
- **Synchronization:**
  - Handled in `cache_read` or `cache_write` with the global lock.

```
void cache_read(block_sector_t sector_id, off_t size, off_t of
fset);
```

- **Implementation:**
  - First, acquire `global_cache_lock`.
  - Loop through cache blocks
  - If file is in cache (by calling `find_cache`):
    - Reset the that corresponding `cache_node`'s chances to `MAX_CHANCES`
  - Else (file could not be found in cache):
    - Call `block_read`, which is reading into disk
    - Now that we want to store into cache...
    - If cache is full: (looping through to see if it's full)
      - Evict with **clock policy**, as stated
    - Else (cache has space)
      - Fill that extra cache space with the newly found file from disk
  - Call memcpy
  - Lastly, release `global_cache_lock`.

- **Rationale:**
  - The reason why we have these if statements and checks is because we want to make sure we don't immediately go into disk. With these checks, we can first:
    1. See if the file's already in the cache
    2. If it's not in the cache, we have to read into disk and fill it into our cache
  - The clock policy eviction algorithm has been handled in `cache_evict`, and we need to search in disk in this function **only** when the file is not in our current cache.
  - An alternative is to call `block_read` (or reading into disk) in `cache_evict`, but we decided to put `block_read` in this function since we have to read into disk even if our cache is not full yet and does not need eviction.
- **Complexity:**
  - **Time:** O(N) - Since we are iterating through an array of 64, it will search in a linear fashion.
  - **Memory:** O(N) where N is the cache block count. The pointer sizes and other variables in the node struct should remain mostly constant, so we only care about how many cache blocks there are.
- **Synchronization:**

- **Needed:** Yes. We will be using a global lock for the cache.
- **Reasoning:** There is a possibility where the cache is accessed by multiple threads at once and we need to make sure that we have a critical section set up for this.

```
void cache_write(off_t size, off_t offset);
```

- **Implementation:**
  - First, acquire `global_cache_lock`.
  - Loop through cache blocks
  - If file is in cache (by calling `find_cache`):
    - Reset the that corresponding `cache_node`'s chances
  - Else (file could not be found in cache):
    - Call `block_read`, which is reading into disk
    - Now that we want to store into cache...
    - If cache is full: (looping through to see if it's full)
      - Evict with **clock policy**, as stated
    - Else (cache has space)
      - Fill that extra cache space with the newly found file
  - call memcpy
  - Lastly, release `global_cache_lock`.
- **Rationale:**
  - *(Similar to Read)*
  - The reason why we have these if statements and checks is because we want to make sure we don't immediately go into disk. With these checks, we can first:
    1. See if the file's already in the cache
    2. If it's not in the cache, we have to read into disk and fill it into our cache
  - The clock policy eviction algorithm has been handled in `cache_evict`, and we need to search in disk in this function **only** when the file is not in our current cache.
  - An alternative is to call `block_read` (or reading into disk) in `cache_evict`, but we decided to put `block_read` in this function since we have to read into disk even if our cache is not full yet and does not need eviction.
- **Complexity:**
  - **Time:** O(N) - Since we are iterating through an array of 64, it will search in a linear fashion.

    - **Memory:** O(N) where N is the cache block count. The pointer sizes and other variables in the node struct should remain mostly constant, so we only care about how many cache blocks there are.
- **Synchronization:**
    - **Needed:** when the function starts and ends

```
void cache_flush(struct cache_node *n); // Empty the cache and reset everything
```

- **Implementation:**
    - We check if the current cache node is dirty
        - If it is, we call `block_write`
- **Rationale:**
    - We check to see if the cache is dirty and write if it is `block_write`. An alternative is to make this function loop through the entire cache without inputting a specific cache node as an argument, but it's easier for us to call this inside loops of other functions.
- **Complexity:**
    - **Time:** O(1) - constant time
    - **Memory:** O(1) - constant time
- **Synchronization:**
    - **Needed:** Not needed; handled in `cache_read` or `cache_write`.

Extensible Files

Data Structures and Functions

Somewhere we gotta keep track of all of the file opens/closes and make sure to close/free a file when there are no more references to it or something like that, ask around/add something to the file descriptor struct every time a file is opened or something like that?

Data structures/variables added:

```
#define DIRECT_BLOCK_POINTERS 10 //Direct pointer for smol files
#define INDIRECT_BLOCK_POINTERS 2 //Indirect pointer for slightly larger files
```

```
struct inode_disk {
 /* New block_sector_t arrays for direct, indirect, and doubly indirect pointers */
  block_sector_t direct[DIRECT_BLOCK_POINTERS]; //Direct pointers
  block_sector_t indirect[INDIRECT_BLOCK_POINTERS]; //2 indirect pointers -> 128 ->
                                                    //pointer -> 128 -> data
  block_sector_t doubly_indirect; //One doubly indirect pointer -> 128^2
  //indr_ptr -> 128 pointers -> 128 ptrs -> block of actual data
  off_t length; // File size in bytes
  unsigned magic; //Magic number



  /* Adjusted size of the unused array to still have the struct as the block size */
  uint32_t unused[113];
};
```

```
/* In-memory inode. */
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. */
```

```
  struct inode_disk data; /* Inode content. */
  //Add locks for each inode
  struct lock inode_lock;
};
```

Data structures/variables modified:

```
//Modified struct inode_disk to accommedate a direct, indirec
t, and doubly indirect ointer to allow for file growth.
```

Functions added:

```
/* Function to check space allocation failures */
bool block_allocate_check(block_sector_t sector){
  if (sector == 0) {
   inode_resize(id, id->length);
   return false;
  }
}
```

```
block_sector_t block_allocate(void){
  block_sector_t new_sector;
  if(free_map_allocate(1, &new_sector)) {
    return new_sector;
  }else{
    return -1; //Or some other error indication
  }
}
```

```
void block_free(block_sector_t n) {
  free_map_release(n, 1);
}
```

```
void block_read(block_sector_t n, uint8_t buffer[512]) {
  block_read(fs_device, n, buffer);
}
```

```
void block_write(block_sector_t n, uint8_t buffer[512]) {
  block_write(fs_device, n, buffer);
}
```

```
bool inode_resize(struct inode_disk* id, off_t size) {
  //Lock resize function
  block_sector_t sector;
  /* Handle direct pointers. */
  for (int i = 0; i < DIRECT_BLOCK_POINTERS; i++){
    if(size <= BLOCK_SECTOR_SIZE * i && id->direct[i] != 0){
      /* Shrink. */
      block_free(id->direct[i]);
      id->direct[i] = 0;
    }else if(size > BLOCK_SECTOR_SIZE * i && id->direct[i] ==
0){
      /* Grow. */
      id->direct[i] = block_allocate();
    }
  }
  if (id->indirect == 0 && size <= DIRECT_BLOCK_POINTERS * BLO
CK_SECTOR_SIZE) {
    id->length = size;
    return true;
  }
  block_sector_t buffer[128];
  memset(buffer, 0, 512);
  /* Check if indirect pointers are needed. */
```

```
  if (id->indirect == 0 && size <= 12 * 512) {
    id->length = size;
    return true;
  }
  if (id->indirect == 0) {
    /* Allocate indirect block. */
    id->indirect = block_allocate();
  } else {
    /* Read in indirect block. */
    block_read(id->indirect, buffer);
  }
  /* Handle indirect pointers. */
  /* Need to modify this to handle N indirect pointers, likley
with another loop */
  /* I'm not certian how the logic works out for checking the
size, would it simply be (j * 128) + i + DIRECT_BLOCK_POINTERS
or something like that */
  for (int i = 0; i < 128; i++){
    if (size <= (DIRECT_BLOCK_POINTERS + i) * BLOCK_SECTOR_SIZ
E && buffer[i] != 0){
      /* Shrink. */
      block_free(buffer[i]);
      buffer[i] = 0;
    }else if(size > (DIRECT_BLOCK_POINTERS + i) * BLOCK_SECTOR
_SIZE) && buffer[i] == 0){
      /* Grow. */
      buffer[i] = block_allocate();
    }
  }
  if(size <= DIRECT_BLOCK_POINTERS * BLOCK_SECTOR_SIZE){
```

```
    /* We shrank the inode such that indirect pointers are no
longer required. */
     block_free(id->indirect);
     id->indirect = 0;
   }else{
     /* Write updates to the indirect block back to disk.*/
     block_write(id->indirect, buffer);
   }
   id->length = size;
   return true;
}
```

```
/* Draft for how to handle doubly indirect pointers */

//Handle doubly indirect pointer
//Check if the size is greater than all of the direct block po
inters * the block sector size, plus the size of all of the in
direct pointers
if(size > (DIRECT_BLOCK_POINTERS * BLOCK_SECTOR_SIZE) + (128 *
BLOCK_SECTOR_SIZE * INDIRECT_BLOCK_POINTERS)) {
  if(id->doubly_indirect == 0){
    //Allocate doubly indirect block if not already allocated
    id->doubly_indirect = block_allocate();
  }

  //Pattern match the buffers and the zeroing out of all of th
e buffers
  block_sector_t indirect_block_buffer[128];
  block_sector_t direct_block_buffer[128];
  memset(indirect_block_buffer, 0, 512);
  memset(direct_block_buffer, 0, 512);
```

```
  //First loop to loop over all of the first layer of the doubly-indirect pointer
  for(int i = 0; i < 128; i++){
    off_t indirect_block_max_size = (DIRECT_BLOCK_POINTERS * BLOCK_SECTOR_SIZE) + (128 * INDIRECT_BLOCK_POINTERS * BLOCK_SECTOR_SIZE) + ((i + 1) * 128 * BLOCK_SECTOR_SIZE);

    if(size <= indirect_block_max_size){
      //Shrink: Free any blocks beyond the current size
      if(indirect_block_buffer[i] != 0){
        //Read in the pointers stored in the doubly indirect pointer
        block_read(id->doubly_indirect, indirect_block_buffer);
        block_read(indirect_block_buffer[i], direct_block_buffer);
        //Loop through the entirety of the second layer of the first pointer of the doubly indirect pointer
        for(int j = 0; j < 128; j++){
          if(direct_block_buffer[j] != 0){
            block_free(direct_block_buffer[j]);
            direct_block_buffer[j] = 0;
          }
        }
        block_write(indirect_block_buffer[i], direct_block_buffer);
        block_free(indirect_block_buffer[i]);
        indirect_block_buffer[i] = 0;
      }
    }else{
```

```
      //Grow: Allocate any blocks needed up to the current siz
e
      if(indirect_block_buffer[i] == 0){
        indirect_block_buffer[i] = block_allocate();
      }
      block_read(indirect_block_buffer[i], direct_block_buffe
r);
      for(int j = 0; j < 128; j++){
        if(size > ((DIRECT_BLOCK_POINTERS + i * 128 + j) * BLO
CK_SECTOR_SIZE) && direct_block_buffer[j] == 0){
          direct_block_buffer[j] = block_allocate();
        }
      }
      block_write(indirect_block_buffer[i], direct_block_buffe
r);
    }
  }

  //Write back the changes to the doubly indirect block
  block_write(id->doubly_indirect, indirect_block_buffer);
}else if(id->doubly_indirect != 0) {
  //Free the doubly indirect block if it's no longer needed
  block_free(id->doubly_indirect);
  id->doubly_indirect = 0;
}

id->length = size;
return true;
```

```
syscall inumber(){
```

```
  //index into the file descriptor table, get the file object, and return the associated inode sector, unique id to the inode
}
```

Functions modified:

```
bool inode_create(block_sector_t sector, off_t length){
  //Modify this function to match the direct/indirect/doubly indirect pointers as well as the size of the garbage/padding bytes
}
```

```
void inode_close(struct inode* inode){
  //Lock the the buffer cache
  //Evict this node on the buffer cache if it's there
  //Modify this function to actually close all of the things using block_free, iterate through the direct, indirect, and doubly indirect pointers
}
```

```
off_t inode_write_at(struct inode* inode, const void* buffer_, off_t size, off_t offset){
  //Lock the inode itself
  //Update buffer cache
  //Modify function to actually write to the direct/indirect/doubly indirect pointers instead of just writing sequantially to all of the things
  //Unlock the inode
}
```

```
off_t inode_read_at(struct inode* inode, void* buffer_, off_t size, off_t offset){
  //Lock?
```

```
  //Modify function to actually read from the direct/indirect/
doubly indirect pointers instead of just the sequantial memory
as of current


}
```

```
static block_sector_t byte_to_sector(const struct inode* inod
e, off_t pos){
  //Lock around this function probably since the inode's size
can be modified and such
  //Modify this function to accomedate actually going through
the pointers rather than just going one & done through the mem
ory
}
```

Algorithms

```
/* Function to check space allocation failures */
bool block_allocate_check(block_sector_t sector){
  if (sector == 0) {
   inode_resize(id, id->length);
   return false;
  }
}
```

- **Implementation:**
  - Basically checks if the sector is zero, and then resizes if it's zero and then calls resize if it's still zero/failed
- **Rationale:**
  - It was from the discussion and I straight copped it from there, so I assume it's right
- **Complexity:**
  - **Time:** O(n) - O(n) calling inode resize will basically always be dependent on the size of the input, upper bounding/worst case scenario putting it at O(N)
  - **Memory:** O(n) - O(n), Again, the resizing could take at most O(N) memory
- **Synchronization:**

    - **Needed:** `Yes`
    - **Reasoning:** Calling inode_resize in multiple different threads with different sizes would be weird if not done atomically

```
block_sector_t block_allocate(void){
  block_sector_t new_sector;
  if(free_map_allocate(1, &new_sector)) {
    return new_sector;
  }else{
    return -1; //Or some other error indication
  }
}
```

- **Implementation:**
    - Call free_map allocate on the sector and return the new sector, otherwise return -1 if it fails
- **Rationale:**
    - Kinda boilerplate-ish, I think it works? Allocate a block at a time, how it works in inode create wrapper function kinda sorta
- **Complexity:**
    - **Time:** O(1) - O(1) I believe the allocation of a new sector is O(1) since it allocates a single block
    - **Memory:** O(1) - O(1) This allocates a single new block, so it would take O(1) memory no matter what
- **Synchronization:**
    - **Needed:** `No`
    - **Reasoning:** It's only gonna be called in a locked context, so it should be fine.

```
void block_free(block_sector_t n) {
  free_map_release(n, 1);
}
```

- **Implementation:**
    - Just a wrapper function to free blocks, call map_free_release
- **Rationale:**
    - It was in the discussion and it looks nicer than free_map_release
- **Complexity:**
    - **Time:** O(1) Again, calling this on the specific sector should be constant time since I'm always freeing a constant amount of memory

    - **Memory:** O(1), since it's freeing memory/releasing memory, it's technically -n, but it's freeing the particular sector of memory
- **Synchronization:**
    - **Needed:** No
    - **Reasoning:** Will be called in locked contexts

```
bool inode_resize(struct inode_disk* id, off_t size) {}
```

- **Implementation:**
    - See the nitty gritty code above, basically the same as discussion, where we first iterate over all of the direct pointers, and then iterate over all of the indirect pointers, and then perhaps the doubly indirect pointer to see if the space is needed for it, and then properly grow/shrink the size of the file accordingly
- **Rationale:**
    - This was what was done in discussion, so plug chug pattern match all the way
- **Complexity:**
    - **Time:** O(N), this function is wholeheartedly dependent on the input and how much it's being resized to
    - **Memory:** O(N), it's basically literally by the function definition that (if successful) it's going to consume N sectors of memory to do this
- **Synchronization:**
    - **Needed:** `Yes`
    - **Reasoning:** Yeah we need to lock the inode if we're modifying the size of it since other modifications to the size being made concurrently would result in weirdness happening with the scheduler and such.

```
inumber(int fd){
  //index into the file descriptor table, get the file object, and return the associated inode sector, unique id to the inode
}
```

- **Implementation:**

```
index into the file descriptor table, get the file object, and return the associated inode sector, unique id to the inode
```

- **Rationale:**
    - That just seems like a logical way to do things.
- **Complexity:**
    - **Time:** O(N), dependent on the size of the file descriptor table

    - **Memory:** O(1), returns a single number, so constant size for memory since it doesn't need any more or less memory
- **Synchronization:**
    - **Needed:** `No`
    - **Reasoning:** Happens in interrupt context, so no explicit synchronization required/needed

Functions modified:

```
bool inode_create(block_sector_t sector, off_t length){
  //Modify this function to match the direct/indirect/doubly indirect pointers as well as the size of the garbage/padding bytes
}
```

- **Implementation:**
    - Basically call inode resize to resize the calloc'd inode_disk to be the proper size.
- **Rationale:**
    - We want to match the allocation of the new filesystem/allocation of memory that we're implementing rather than the current one of just writing however many bytes in a row.
- **Complexity:**
    - **Time:** O(N), this is basically a wrapper function for inode_resize, so again, it depends on the input and therefor is O(N)
    - **Memory:** O(N), same thing here once more, since it depends on the input value, then we're going to be O(N)
- **Synchronization:**
    - **Needed:** `Yes`
    - **Reasoning:** Creating things and immediate writing to them without ensuring that it's actually properly initialized would be bad.

```
void inode_close(struct inode* inode){
  //Lock the the buffer cache
  //Evict this node on the buffer cache if it's there
  //Modify this function to actually close all of the things using block_free, iterate through the direct, indirect, and doubly indirect pointers
```

```
}
```

- **Implementation:**
  - Lock the buffer cache, evict node from cache if it's there, and the basically begin iterating through all of the direct, indirect, and doubly indirect pointers, and free all of them
- **Rationale:**
  - It seems like a reasonable thing to do ngl
- **Complexity:**
  - **Time:** O(N), depends on the size of the actual inode_disk itself, where N is the size of the disk
  - **Memory:** O(-N), I suppose this frees up N sectors, so it would be the inverse of taking N sectors, but still takes N sectors
- **Synchronization:**
  - **Needed:** `Yes`
  - **Reasoning:** Buffer cache is gonna have to be locked

```
off_t inode_write_at(struct inode* inode, const void* buffer_, off_t size, off_t offset){
  //Update buffer cache
  //Modify function to actually write to the direct/indirect/doubly indirect pointers instead of just writing sequantially to all of the things
}
```

- **Implementation:**
  - if the offset + size is bigger than inode_length of current node, we need to resize by calling resize funciton we implemented,
  - other implementations stays the same
  - this function will call `cache_write`
- **Rationale:**
  - This will resize if need to and instead of writing to disk directly calling block_write, we wil call cache_write
- **Complexity:**
  - **Time:** O(N) - O(N) depending on the size
  - **Memory:** O(N) - O(N) depending on the size
- **Synchronization:**
  - **Needed:** No synchronization will needed at least for in inode_write at

```
off_t inode_read_at(struct inode* inode, void* buffer_, off_t size, off_t offset){
  //Lock?
  //Modify function to actually read from the direct/indirect/doubly indirect pointers instead of just the sequantial memory as of current


}
```

- **Implementation:**
  - other implementations stays the same
  - this function will call `cache_read` instead of block_read
- **Rationale:**
  - Before it was calling `block_read`.
  - This function is intended to call buffer cache `cache_read`
- **Complexity:**
  - **Time:** O(N) - O(N) depending on the size
  - **Memory:** O(N) - O(N) depending on the size
- **Synchronization:**
  - **Needed:** No

```
static block_sector_t byte_to_sector(const struct inode* inode, off_t pos){
  //Lock around this function probably since the inode's size can be modified and such
  //Modify this function to accomedate actually going through the pointers rather than just going one & done through the memory
}
TODO
```

- **Implementation:**
  - Acquire the inode lock
  - Iterate through the pointers rather than just going to a specific contiguous offset within memory
- **Rationale:**
  - Since memory is now allocated in blocks, this requires a modification of the function to instead iterate through the pointers rather than just going to a

specific constant offset

- **Complexity:**
  - **Time:** O(N) It needs to iterate through the positions to find the actual place where the sector
  - **Memory:** O(1), it's returning a specific block sector so it's going to be constant space.
- **Synchronization:**
  - **Needed:** `Yes`
  - **Reasoning:** The inode can't be resized or modified while the block device sector that contains that byte is being searched for

---

Subdirectories

Data Structures and Functions

Data structures/variables added:

```
struct dir // existing directory stream.
// Extended to support directory inodes.
struct inode  {
    // new addition
    bool is_directory;  // New field to distinguish between file and directory
}
```

Data structures/variables modified:

```
// Modified to support directory operations.
struct file  {
    // new addition
    bool is_directory;   /* True if this file is a directory. */
    struct dir* dir;     /* Directory structure, if this file is a directory. */
```

```
}


struct inode_disk {
    // new addition
    bool is_directory; // True if this inode is a directory.
}
```

Functions added:

```
// Main functions:
bool chdir(const char* dir) // Changes the current working dir
ectory.
bool mkdir(const char* dir) // Creates a new directory.
bool readdir(int fd, char* name) // Reads a directory entry.
bool isdir(int fd) // Checks if a file descriptor refers to a
directory.
int inumber(int fd) // Returns the inode number for a file/dir
ectory.


// Helper functions:
/* Extracts the next part of the path from 'path' into 'part'.
   Returns true if a part is extracted, false if 'path' is emp
ty or NULL.
   'path' is updated to point after the extracted part. */
bool get_next_path_part(char part[NAME_MAX + 1], const char **
path);
```

Functions modified:

```
bool open(const char* file) // Updated to support opening dire
ctories.
void close(int fd) // Updated to handle directories.
pid_t exec(const char* file) // Updated to support relative pa
ths.
```

```
bool remove(const char* file) // Updated to support removing directories.
```

Algorithms

Algorithms — Added Functions:

```
bool get_next_path_part(char part[NAME_MAX + 1], const char **path);
```

- **Implementation:**
  - Check if the path is empty or null. If it is, return false.
  - Skip all leading slash ('/') characters.
    - This positions the path pointer at the start of the next path part.
  - Initialize a counter to keep track of how many characters are copied to 'part'.

  - Copy characters from 'path' to 'part':
    - While the current character at 'path' is not a slash ('/') AND not the end of the string ('\0') AND the counter is less than NAME_MAX:
      - Copy the current character from 'path' to 'part'.
      - Increment both the path pointer and the counter.

  - After the loop, add a null terminator ('\0') to 'part' at the current counter position so that it marks the end of the current path part.
  - Return true if at least one character has been copied to 'part'
  - (counter > 0), otherwise return false. */

  - The function iteratively processes a string representing a file system path and extracts each directory or file name component. It ignores leading slashes and copies characters until it encounters a slash or the end of the string, up to a maximum of NAME_MAX characters.
- **Rationale:**
  - `W`e want to extract the next part of a file path so that we can break down paths into individual components, which can then be used to traverse the directory tree and for path resolution.
- **Complexity:**
  - **Time: O(N)** - the length of the path segment being processed
  - **Memory: O(1)** - It only requires space for the part array (up to NAME_MAX + 1 characters) and a few local variables
- **Synchronization:**

    - **Needed:** `No.` The function operates on local variables and parameters passed by value
    - **Reasoning:** Since `get_next_path_part` is designed to work on a copy of the path pointer and does not interact with shared data, it is inherently thread-safe and does not require synchronization.

```
bool chdir(const char* dir);
```

- Implementation:
    - Check if the 'dir' pointer is null or points to an empty string. If it does, return false.
    - Resolve the path 'dir':
    - Check if 'dir' is an absolute path (starts with '/')
    - If it is, start from the root directory.
    - Otherwise, start from the current directory of the process.
    - Use a loop to process each part of 'dir' using get_next_path_part.
        - For each part of the path:
            - Find the directory entry with the given name.
            - Check if the entry is a directory.
            - If it's a directory, move to that directory.
            - If any part is not found or is not a directory, return false.
    - After successfully processing all parts of 'dir', update the process's current directory to the final directory reached in the loop.
    - Return true after successfully changing the current directory.

    - The function changes the process's current working directory to the one specified by `dir`. It supports both relative and absolute paths and navigates through the file system hierarchy to reach the desired directory.
- Rationale:
    - We need a way to change the current working directory so that we can navigate the directories and subdirectories and set their working context for file operations.
- Complexity:
    - Time: O(N) - where N is the number of elements in the path and that each element's existence and type (file or directory) need to be checked
    - Memory: O(1) - Memory usage will remain constant because the function primarily relies on path string manipulation and directory lookups.
- Synchronization:

    - Needed: `Yes` Because the function modifies the process's state (current working directory), which may be shared across different threads within the same process.
    - Reasoning: Accessing and modifying inodes requires synchronization to prevent data races and ensure filesystem integrity. This means when chdir resolves the path to change the current directory, it may need to read from or write to inodes representing directories. This operation should be synchronized to ensure that the inode data is consistent and valid.

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

- Implementation:
    - Check if 'dir' is `NULL` or an empty string. If so, return false as the operation cannot proceed.
    - Resolve the path 'dir':
        - If 'dir' is an absolute path, start from the root directory.
        - If 'dir' is a relative path, start from the current working directory.
        - Use `get_next_path_part` to navigate through the path.
    - Check if the final directory in 'dir' already exists. If it does, return false (as mkdir should not overwrite existing directories).
    - Create a new directory at the resolved path. This involves allocating a new inode and initializing it as a directory.
    - Add an entry for the new directory in its parent directory's contents.
    - Return true if the directory was successfully created, false otherwise (due to reasons like insufficient disk space, invalid path, etc.).

    - The mkdir function takes a string representing the path to a new directory and attempts to create it. Same as our other dir related functions, it utilizes **`get_next_path_part`** to determine the exact location where the new directory should be created.
- Rationale:
    - By implementing mkdir, it allows users and applications to create new directories, thereby organizing files and other directories into a hierarchical structure, hence subdirectories.
- Complexity:
    - Time: O(N) - where N is the depth of the directory tree (number of directories in the path).

    - Memory: O(1) - Memory usage will remain constant because the function involves modifying file system metadata.
- Synchronization:
    - Needed: `Yes` Because the function modifies the information in struct inode, or file system structure, which is a shared resource.
    - Reasoning: Similar to chdir, concurrent access to the filesystem structure (like directory entries) by multiple processes could lead to inconsistencies.

```
bool readdir(int fd, char* name);
```

- Implementation:
    - Validate 'fd' to ensure it refers to an open directory.
    - Acquire a lock on the inode for process-safe access.
    - Check if the directory's current position is at the end of its entries. If so, release the lock and return false.
    - Sequentially read entries from the directory associated with 'fd'.
    - Skip over any entries that are '.' or '..'.
    - Ensure that the 'name' buffer can accommodate the entry's name, respecting the `READDIR_MAX_LEN` limit.
    - Copy the name of the next valid entry into the 'name' buffer and null-terminate it.
    - Update the directory's position marker to the next entry.
    - Release the lock on the inode.
    - Return true if a valid entry is read, false if there are no more entries.

    - The `readdir` function accesses the contents of a directory, identified by 'fd', and retrieves the next entry while skipping over '.' and '..'. It respects the maximum length limit for directory names, as defined by `READDIR_MAX_LEN`.
- Rationale:
    - We want to read directory contents in our file system, allowing programs to list files and navigate file systems with the addition of exclusion of '.' and '..' .
- Complexity:
    - Time: O(1) - Given that the directories are in ready to read mode.
    - Memory: O(1) - Memory usage is constant based on the `READDIR_MAX_LEN` limit.
- Synchronization:
    - Needed: Yes. This is because the operation involves accessing and modifying shared file system resources aka the struct inode, or file system structure,

which is a shared resource.
    - Reasoning: Similar to the other functions, concurrent access to a directory's contents could lead to inconsistencies or race conditions, especially in a multi-process environment. A lock on its inode ensures that only one thread can read or modify the directory's contents at a time, maintaining filesystem integrity.

```
bool isdir(int fd);
```

- Implementation:
    - Validate the file descriptor 'fd' to ensure it's valid and open.
    - Acquire a lock on the file or inode associated with 'fd' for process-safe access.
    - Retrieve the inode associated with 'fd'.
    - Check the type of the inode (file or directory).
    - This might involve checking a specific field or flag in the inode structure that indicates its type.
    - Release the lock on the file or inode.
    - Return true if the inode type indicates a directory, false otherwise.
    - The `isdir` function simply determines the type of the file system object referred to by 'fd'. It accesses the inode associated with 'fd' and checks whether it represents a directory.
- Rationale:
    - This function provides a mechanism to distinguish between files and directories, which is necessary for making decisions based on the type of a file system object, especially when dealing with file system operations.
- Complexity:
    - Time: O(1) - The function performs a constant-time operation to check the type of an inode.
    - Memory: O(1) - Memory usage will remain constant because the function primarily deals with already allocated structures.
- Synchronization:
    - Needed: `Yes` Because to ensure process-safe access to the inode and its metadata, locking is required.
    - Reasoning: Locking the inode or its associated file structure prevents concurrent access issues and ensures that the type of the inode is consistently determined.

```
int inumber(int fd);
```

- Implementation:
  - Validate the file descriptor 'fd' to ensure it's valid and open.
  - Acquire a lock on the file or inode associated with 'fd' for process-safe access.

  - Retrieve the inode associated with 'fd'.
  - Get the inode number from the inode structure.
  - This typically involves accessing a specific field in the inode structure that stores the inode number.
  - Release the lock on the file or inode.
  - Return the inode number.

  - The `inumber` function retrieves the inode number associated with a file descriptor 'fd'. It involves accessing the inode structure corresponding to 'fd' and returning the value of the inode number field.
- Rationale:
  - This function provides a mechanism to distinguish between files and directories, which is necessary for making decisions based on the type of a file system object, especially when dealing with file system operations.
- Complexity:
  - Time: O(1) - Retrieving the inode number should be a direct access operation so it’s constant runtime.
  - Memory: O(1) - The function works with already allocated structures so it doesn’t need additional memory.
- Synchronization:
  - Needed: `Yes` Same justification as isdir
  - Reasoning: Same reason as isdir

Algorithms — Modified Functions:

```
bool open(const char* file) {
    // Check if 'file' is NULL or an empty string. If so, retu
rn -1 (indicating failure) as the operation cannot proceed.


    /* Resolve the path 'file':
```

```
       If 'file' is an absolute path, start from the root dire
ctory.
       If 'file' is a relative path, start from the current wo
rking directory.
       Use path parsing functions (like get_next_path_part) to
navigate through the path.
    */

    // Attempt to find the file or directory at the resolved p
ath.
    // Use a structure that can represent both files and direc
tories (e.g., struct file_descriptor).

    struct file_descriptor* fd;
    fd = malloc(sizeof *fd);
    if (fd == NULL) {
        return -1; // Failed to allocate memory.
    }

    lock_acquire(&fs_lock);
    // Open the file or directory. Use a unified interface tha
t can handle both files and directories.
    fd->file_or_dir = unified_open(file);
    if (fd->file_or_dir != NULL) {
        struct thread* cur = thread_current();
        int handle = fd->handle = cur->pcb->next_handle++;
        list_push_front(&cur->pcb->fds, &fd->elem);
    } else {
        free(fd);
        handle = -1; // Indicate failure to open.
    }
```

```
    lock_release(&fs_lock);


    return handle; // Return the file descriptor handle or -1
if failed.
}
```

- Implementation:
  - The function starts by resolving the path to the file or directory specified by file. This process handles both absolute and relative paths.
  - The function then tries to open the file or directory. For this, it uses a unified open function (unified_open) capable of handling both files and directories.
  - If successful, the function creates a new file descriptor, assigns it a unique handle, and adds it to the process's list of file descriptors.
- Rationale:
  - With this modified open system call, we can to handle files and directories uniformly, enabling operations on directories and supporting hierarchical file systems with subdirectories.
- Complexity:
  - Time: O(N) - where N is the depth of the directory tree.
  - Memory: O(1) - Memory usage remains constant as the function primarily modifies file system metadata.
- Synchronization:
  - Needed: Yes, because the function accesses and modifies shared file system structures.
  - Reasoning: Since the open system call can alter the state of the file system, it's crucial to use synchronization mechanisms like locks to ensure consistent and safe access to shared resources.

```
void close(int fd) {
    // Check if the file descriptor 'fd' is valid. If not, sim
ply return without doing anything.


    /* Locate the file descriptor in the current process's lis
t of file descriptors.
       If it's not found, return immediately as there's nothin
g to close. */
```

```
    struct file_descriptor* fd_struct;
    fd_struct = find_file_descriptor(fd);
    if (fd_struct == NULL) {
        return; // File descriptor not found.
    }

    lock_acquire(&fs_lock);
    // Check if the file descriptor refers to a file or a dire
ctory.
    if (is_file_descriptor_a_file(fd_struct)) {
        file_close(fd_struct->file); // Close the file.
    } else {
        dir_close(fd_struct->dir); // Close the directory.
    }
    lock_release(&fs_lock);

    // Remove the file descriptor from the process's list and
free its memory.
    list_remove(&fd_struct->elem);
    free(fd_struct);
}
```

- Implementation:
  - The function begins by validating the provided file descriptor (fd). It then searches for the corresponding file_descriptor structure in the current process's list of file descriptors.
  - We call file_close or dir_close based on the input type (file or directory).
  - The file descriptor is then removed from the process's list and its memory is freed.
- Rationale:
  - This implementation ensures that resources associated with both files and directories are properly released when they are no longer needed. This is

crucial for maintaining the integrity and efficiency of the file system.

- Complexity:
  - Time: O(N) - The complexity comes from searching the list for the specified file descriptor.
  - Memory: O(1) - The function uses a constant amount of memory, as it only deallocates resources associated with the specific file descriptor.
- Synchronization:
  - Needed: Yes, same justification as open
  - Reasoning: Same as open

```
pid_t exec(const char* file) {
    // Verify that 'file' is a valid pointer to a user-space a
ddress.
    if (!is_user_vaddr(file)) {
        return -1; // Return an error code for invalid addres
s.
    }
    char* kfile = copy_in_string(file); // Copy the file path
from user space to kernel space.
    // Resolve the file path to its absolute form.
    char* resolved_path = resolve_path(kfile);
    pid_t pid = -1; // Initialize the process ID to an invalid
value.
    if (resolved_path != NULL) {
        lock_acquire(&fs_lock);
        pid = process_execute(resolved_path); // Execute the f
ile.
        lock_release(&fs_lock);
        free(resolved_path); // Free the resolved path memory.
    }
    free(kfile); // Free the kernel copy of the file path.
    return pid; // Return the process ID (or error code).
}
```

- Implementation:
  - The function begins by validating the file pointer to ensure it's a valid user-space address.
  - It copies the file path from user space to kernel space for safe access.
  - The function then resolves the path to its absolute form. This involves checking if the path is relative and, if so, appending it to the current working directory.
  - After path resolution, process_execute is called with the absolute path to execute the file.
- Rationale:
  - Supporting relative paths in exec is vital for a flexible and user-friendly file system. It allows programs to be executed relative to the current working directory, which is a common requirement in many operating systems.
- Complexity:
  - Time: O(N) - For the length of the path, and process execution complexity depends on the underlying process management.
  - Memory: O(N) - Mainly for storing the resolved absolute path and the kernel-space copy of the file path.
- Synchronization:
  - Needed: Yes, same justification as open
  - Reasoning: Same as open

```
bool remove(const char* file) {
    // Validate 'file' pointer for safety.
    if (!is_user_vaddr(file)) {
        return false; // Return error for invalid address.
    }
    char* kfile = copy_in_string(file); // Copy the file path
from user space to kernel space.
    // Resolve the file path to its absolute form.
    char* resolved_path = resolve_path(kfile);
    bool success = false; // Initialize success flag.
    if (resolved_path != NULL) {
        lock_acquire(&fs_lock);
        struct inode* inode = NULL;
```

```
        if (filesys_lookup(resolved_path, &inode)) { // Look up the file or directory.
            if (is_inode_directory(inode) && !is_directory_empty(inode)) {
                // If it's a non-empty directory, fail the removal.
                success = false;
            } else {
                // If it's a file or an empty directory, remove it.
                success = filesys_remove(resolved_path);
            }
            inode_close(inode); // Close the inode.
        }
        lock_release(&fs_lock);
        free(resolved_path); // Free the resolved path memory.
    }
    free(kfile); // Free the kernel copy of the file path.
    return success; // Return the status of removal.
}
```

- Implementation:
  - The function checks the validity of the file pointer to ensure it's in user-space.
  - The file path is copied into kernel space.
  - The path is resolved to its absolute form, considering the current working directory.
  - A lookup is performed to find the inode associated with the path.
  - If the inode represents a directory, the function checks whether the directory is empty. Non-empty directories cannot be removed.
  - If it's a file or an empty directory, the function proceeds to remove it using filesys_remove.
- Rationale:

    - This modification allows the remove function to handle directories and files uniformly, adhering to common file system operations in many operating systems.
- Complexity:
    - Time: O(N) - For the path resolution and directory checking.
    - Memory: O(N) - Memory usage is mainly for the resolved path and the kernel-space copy of the file path.
- Synchronization:
    - Needed: Yes, same justification as open
    - Reasoning: Same reasoning as open

Concept Check

1. There are two types of cache features that we could implement and these are the ways we would do it:

- **Cache Write-Behind**
    - The way write-behind works is that we would have to write periodically in case a random crash occurs. To do this, we can use the timer we implemented in Project 2. We can initialize a thread in `cache_init` that has a main job of just looping through the cache array and flushing all structs with valid bits `true` and dirty bits `true` at a specific timer tick. We would call `timer_sleep()` on this thread for a certain period (this period could be adjusted before the program is run), and whenever it is awoken, we would call a function that loops and runs `cache_flush` the corresponding structs. This will repeat until the program runs to completion and will account for the case where a crash occurs.
- **Cache Read-Ahead**
    - For read-ahead, we need to read more blocks than requested so that we can increase the chances of a cache hit. To do this, we have to read an extra block whenever we go to disk. We can create a new function called `cache_read_ahead` that will `block_read` first, and then spawn a new thread using `thread_create` that will read the block that is next to the already-read block (by calling `block_read` with a certain offset). This will be running in the background, and can acquire a separate lock that's not the global lock to ensure synchronization. If needed, we can always adjust the amount of blocks

we want to read ahead by spawning more threads and calling `block_read` more times with a larger offset. However, the main logic needed would be to spawn at least one separate thread running read in the background.