Project 3 Report
Name: Zijie Yu Student id: 522031910031

1 Step 1: Disk-storage system

Implement a simulation of a physical disk, which is organized by cylinder and sector. Assume the sector size is fixed at 256 bytes, and store the actual data in a real disk file. Additionally, use something to account for track-to-track time. The disk-storage system is composed of three main parts: the disk file (`BDS.c`), a command-line driven client (`BDC.c`) and a random data client (`BDC_random.c`).

1.1 BDS.c

As the tcp-related codes have been given ,I only need to implement two parts: attach to a file and handle some command.

1.1.1 attach to a file

```
    // open file
    int fd = open (diskfname, O_RDWR | O_CREAT, 0);
    if (fd < 0) {
        printf("Error: Could not open file '%s'.\n", diskfname);
        exit(-1);
    }

    // stretch the file
    long FILESIZE = BLOCKSIZE * ncyl * nsec;
    int result = lseek (fd, FILESIZE-1, SEEK_SET);
    if (result == -1) {
        perror ("Error calling lseek() to 'stretch' the file");
        close (fd);
        exit(-1);
    }
    result = write (fd, "", 1);
    if (result != 1) {
        perror("Error writing last byte of the file");
        close(fd);
        exit(-1);
    }
    // mmap
    diskfile= (char *)mmap(NULL, FILESIZE,PROT_READ | PROT_WRITE,MAP_SHARED, fd, 0)
    ;
    if (diskfile== MAP_FAILED)
    {
        close(fd);
        printf("Error: Could not map file.\n");
        exit(-1);
    }
```

This code snippet demonstrates the process of creating and mapping a file to memory in a C program. It begins by opening a file specified by the `diskfname` variable with read-write permissions (`O_RDWR`) and creating it if it does not exist (`O_CREAT`). If the file fails to open, an error message is printed and the program exits. Next, the code calculates the desired file size (`FILESIZE`) based on

predefined constants `BLOCKSIZE`, `ncyl`, and `nsec`. It then uses the `lseek` function to set the file pointer to the last byte of the intended file size. If `lseek` fails, it prints an error message, closes the file descriptor, and exits. After positioning the file pointer, the code writes a single byte at the end to ensure the file is physically extended to the desired size. Again, if this write operation fails, an error message is printed, the file descriptor is closed, and the program exits. Finally, the code maps the entire file into the process's address space using `mmap`, allowing direct memory access to the file's contents. If `mmap` fails, the file descriptor is closed, an error message is printed, and the program exits. This mapped file can now be accessed via the `diskfile` pointer as if it were a regular memory array.

1.1.2 handle command

```
int cmd_i(tcp_buffer *write_buf, char *args, int len) {
    static char buf[64];
    sprintf(buf, "%d %d", ncyl, nsec);

    // send to buffer, including the null terminator
    send_to_buffer(write_buf, buf, strlen(buf) + 1);
    return 0;
}
```

The `cmd_i` function is responsible for handling a specific command (information request) and sending relevant data to a TCP buffer. The function first creates a static buffer to hold a formatted string containing the values of `ncyl` and `nsec`, which represent the number of cylinders and sectors, respectively. This formatted string is then sent to the `write_buf` buffer using the `send_to_buffer` function, including the null terminator. The function returns 0, indicating successful execution.

```
int cmd_r(tcp_buffer *write_buf, char *args, int len) {
    int cyl, sec;
    if (sscanf(args, "%d %d", &cyl, &sec) != 2 || cyl >= ncyl || sec >= nsec) {
        send_to_buffer(write_buf, "Error: Invalid read parameters.", 32);
        return 0;
    }
    int tsleep = abs(cur_cyl - cyl) * ttd;
    usleep(tsleep * 1000);
    cur_cyl = cyl;
    char* block = diskfile + (cyl * nsec + sec) * BLOCKSIZE;
    send_to_buffer(write_buf, block, BLOCKSIZE);
    return 0;

    // send_to_buffer(write_buf, args, len);
    // return 0;
}
```

The `cmd_r` function handles the read command, extracting the cylinder (`cyl`) and sector (`sec`) values from the provided arguments. It first verifies that the extracted values are valid by checking their ranges against `ncyl` and `nsec`. If the values are invalid, it sends an error message to the `write_buf` buffer. If the values are valid, it calculates the time to move the read/write head (`tsleep`) based on the distance from the current cylinder (`cur_cyl`) to the target cylinder, and simulates this delay with `usleep`. After updating the current cylinder position, it calculates the starting address of the target block in the `diskfile` and sends the block's content to the `write_buf` buffer. The function returns 0, indicating successful execution.

```

int cmd_w(tcp_buffer *write_buf, char *args, int len) {
```

```
    int cyl, sec, data_len;

    if (sscanf(args, "%d %d %d", &cyl, &sec, &data_len) != 3 || cyl >= ncyl || sec
    >= nsec || data_len > BLOCKSIZE) {
        send_to_buffer(write_buf, "Error: Invalid write parameters.", 33);
        return 0;
    }


    if (data_len < 0 || data_len > BLOCKSIZE) {
        send_to_buffer(write_buf, "Error: Invalid data length.", 28);
        return 0;
    }

    int tsleep = abs(cur_cyl - cyl) * ttd;
    usleep(tsleep * 1000);
    cur_cyl = cyl;


    char *block = diskfile + (cyl * nsec + sec) * BLOCKSIZE;

    memset(block, 0, BLOCKSIZE);

    char *data_start = strchr(args, ' ') + 1;
    data_start = strchr(data_start, ' ') + 1;
    data_start = strchr(data_start, ' ') + 1;


    if (data_start - args > len) {
        send_to_buffer(write_buf, "Error: Malformed command.", 26);
        return 0;
    }


    memcpy(block, data_start, data_len);
    //send_to_buffer(write_buf, "Write successful", 17);
    send_to_buffer(write_buf, data_start, data_len);

    return 0;
}
```

The `cmd_w` function manages the write command, parsing the cylinder (`cyl`), sector (`sec`), and data length (`data_len`) from the provided arguments. It first checks the validity of these parameters. If they are invalid, it sends an error message to the `write_buf` buffer. If valid, it calculates the head movement time (`tsleep`) and simulates the delay with `usleep`. After updating the current cylinder position, it calculates the starting address of the target block in the `diskfile` and clears the block. The function then locates the starting position of the data within the arguments by skipping past the first three spaces. If this position is invalid, it sends an error message. Otherwise, it writes the specified data length from the arguments into the block and sends the written data back to the `write_buf` buffer. The function returns 0, indicating successful execution.

1.2 BDC.c

The codes are given by the teachers.

1.3 BDC_random.c

```c
#define BLOCKSIZE 256
#define BUFFER_SIZE 4096
void generate_random_data(char *data, int length) {
    static const char charset[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    for (int i = 0; i < length - 1; i++) {
        int key = rand() % (int)(sizeof(charset) - 1);
        data[i] = charset[key];
    }
    data[length - 1] = '\0';
}


int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <Port>", argv[0]);
        exit(EXIT_FAILURE);
    }
    int port = atoi(argv[1]);
    int cylinders, sectors;

    tcp_client client = client_init("localhost", port);
    static char buf[BUFFER_SIZE];
    printf("Request 0: I\n");
    strcpy(buf, "I");
    client_send(client, buf, strlen(buf) + 1);
    int n = client_recv(client, buf, sizeof(buf));
    buf[n] = 0;
    printf("%s\n", buf);
    sscanf(buf, "%d %d", &cylinders, &sectors);
    printf("Totally %d cylinders, %d sectors per cylinder.\n", cylinders, sectors);
    printf("Please input the number of R/W requests: ");

    int nn;
    scanf("%d", &nn);

    srand(time(NULL)); // Seed the random number generator
    for (int i = 0; i < nn; i++)
    {
        int is_write = rand() % 2;
        int cyl = rand() % cylinders;
        int sec = rand() % sectors;
        if (is_write) {
            char data[BLOCKSIZE];
            generate_random_data(data, BLOCKSIZE);
            printf("Request %d: W %d %d %lu %s\n", i + 1, cyl, sec, strlen(data) + 1, data);
            snprintf(buf, sizeof(buf), "W %d %d %lu %s", cyl, sec, strlen(data) + 1, data);
            client_send(client, buf, strlen(buf) + 1);
            int n = client_recv(client, buf, sizeof(buf));
            buf[n] = 0;
            //printf("%s\n", buf);
            if (strcmp(buf, "Bye!") == 0) break;
        } else {
            printf("Request %d: R %d %d\n", i + 1, cyl, sec);
            snprintf(buf, sizeof(buf), "R %d %d", cyl, sec);
            client_send(client, buf, strlen(buf) + 1);
            int n = client_recv(client, buf, sizeof(buf));
```

```
            buf[n] = 0;
            printf("%s\n", buf);
            if (strcmp(buf, "Bye!") == 0) break;
        }
    }
    printf("Request %d: E\n",nn+1);
    strcpy(buf, "E");
    client_send(client, buf, strlen(buf) + 1);
    n = client_recv(client, buf, sizeof(buf));
    buf[n] = 0;
    printf("%s\n", buf);


    client_destroy(client);
}
```

- The program begins by including necessary headers and defining constants for block size and buffer size.
- A helper function `generate_random_data` is defined to generate random data for write operations. This function fills the given buffer with random characters from a predefined set.
- The `main` function checks for the required command-line argument (port number) and initializes the TCP client.
- A buffer is allocated for communication, and the program sends an initial request to the server with the command `"I"` to get the disk's configuration (number of cylinders and sectors).
- The response from the server is parsed to retrieve the number of cylinders and sectors, which are then printed.
- The user is prompted to input the number of read/write requests they want to perform.
- A loop iterates for the number of requests specified by the user. For each iteration:
  - A random choice is made between a write operation and a read operation.
  - Random cylinder and sector values are generated.
  - For a write operation, a 256-byte block of random data is generated using `generate_random_data` and sent to the server along with the command `"W"`.
  - For a read operation, the command `"R"` along with the cylinder and sector values is sent to the server.
  - The response from the server is received. For write operations, the response is not printed to minimize output clutter. For read operations, the response is printed.
  - If the server responds with `"Bye!"`, the loop breaks.
- After completing the read/write requests, an end command `"E"` is sent to the server to terminate the session.
- Finally, the TCP client is destroyed and the program exits.

2 Step 2: Design a basic File System

2.1 Inode Design and Initialization

```
typedef struct {
    int is_used;
    int type;
    int parent_inode;          // inode of parent directory
    int direct_blocks[NUM_DIRECT_POINTERS];
    time_t last_modified;       // Last modification time
    int size;             // Name of the file or directory
} inode;
```

The inode structure stores metadata about files and directories. It includes information such as the file type, parent inode, direct block pointers, last modification time, and file size.

```
void load_inodes_from_disk() {
    char data[BUFFER_SIZE];
    for (int i = 0; i < MAX_INODES; i++) {
        int block_number = ncyl * nsec - 1 - i;
        read_block(block_number, data, sizeof(data));
        if (strlen(data) > 0) {
            string_to_inode(data, &inodes[i]);
        }
    }
}
```

I store the i-th inode in the (ncyl * nsec - 1 - i)th block. Everytime I begin the file system, I will load the inodes from disk.

```
void inode_to_string(inode* node, char* buffer, int buffer_size) {
    int offset = snprintf(buffer, buffer_size, "%d,%d,%d", node->is_used, node->
    type, node->parent_inode);
    for (int i = 0; i < NUM_DIRECT_POINTERS; i++) {
        offset += snprintf(buffer + offset, buffer_size - offset, ",%d", node->
    direct_blocks[i]);
    }
    offset += snprintf(buffer + offset, buffer_size - offset, ",%ld,%d", node->
    last_modified, node->size);
}


void string_to_inode(const char* str, inode* node) {
    sscanf(str, "%d,%d,%d", &node->is_used, &node->type, &node->parent_inode);
    const char* ptr = str;
    for (int i = 0; i < 3; i++) {
        while (*ptr != ',' && *ptr != '\0') {
            ptr++;
        }
        ptr++;
    }
    for (int i = 0; i < NUM_DIRECT_POINTERS; i++) {
        sscanf(ptr, "%d", &node->direct_blocks[i]);
        while (*ptr != ',' && *ptr != '\0') {
            ptr++;
        }
        if (*ptr == ',') {
            ptr++;
        }
    }
```

```
    sscanf(ptr, "%ld,%d", &node->last_modified, &node->size);
}
inode inodes[MAX_INODES];
inode *get_inode(int inode_index) {
    if (inode_index >= 0 && inode_index < MAX_INODES) {
        return &inodes[inode_index];
    }
    return NULL;
}
```

To successfully store the relavant information , I also implement an encode and decode function.

2.2 Free_block Design and Initialization

```
void read_free_blocks_from_disk() {
    char buffer[BUFFER_SIZE];
    int total_blocks = FREE_BLOCKS_SIZE;
    int remaining_bytes = sizeof(int) * ncyl*nsec;
    for (int i = 0; i < total_blocks; i++) {
        printf("1\n");
        read_block(FREE_BLOCKS_START + i, buffer, sizeof(buffer));
        printf("2\n");
        int bytes_to_copy = (remaining_bytes < SECTOR_SIZE) ? remaining_bytes :
   SECTOR_SIZE;
        printf("3\n");
        memcpy(free_blocks + i * SECTOR_SIZE / sizeof(int), buffer, bytes_to_copy);
        printf("4\n");
        remaining_bytes -= SECTOR_SIZE;
    }
}
```

I use an array named free_block to store which block is available and which is not. Everytime I begin the file system, I will load the free_block array from disk.

2.3 format command

```
// Initialize the file system
void initialize_inodes() {
    for (int i = 0; i < MAX_INODES; i++) {
        inodes[i].is_used = 0;
        for(int j=0;j<NUM_DIRECT_POINTERS;j++){
            inodes[i].direct_blocks[j] = -1;
            //inodes[i].use_block[j] = -1;
        }
    }
}

void initialize_free_blocks() {
    for (int i = 0; i < size; i++) {
        free_blocks[i] = 1;  // All blocks are initially free
    }
}
void format_file_system(tcp_buffer *write_buf, char *args, int len) {
    initialize_inodes();
    initialize_free_blocks();
    //num_inodes = 0;
    int root_inode_index = allocate_inode();
    inodes[root_inode_index].type = DIR_TYPE;
```

```
    //inodes[root_inode_index].size = 0;
    inodes[root_inode_index].parent_inode = root_inode_index; // Root's parent is itself
    inodes[root_inode_index].last_modified = time(NULL);
    inodes[root_inode_index].size=0;
    current_directory_inode = root_inode_index;
    write_inode_to_disk(&inodes[root_inode_index], root_inode_index);
    strcpy(current_path, "/");
    send_to_buffer(write_buf, "formatted", 10);
    return;
}
```

In response to the "f" command , the format_file_system function will initialize the inodes and the free_blocks and set the root node.

2.4 mk and mkdir command

```
void mk(tcp_buffer *write_buf, char *args, int len) {
    char filename[MAX_FILENAME_LEN + 1];  // Allocate buffer for filename
    if (sscanf(args, "%200s", filename) != 1 || strlen(filename) > MAX_FILENAME_LEN) {
        send_to_buffer(write_buf, "Error: Invalid ", 32);
        return;
    }

    int inode_index = allocate_inode();
    if (inode_index == -1) {
        send_to_buffer(write_buf, "No free inodes available", 25);
        return;
    }

    inode *node = get_inode(inode_index);
    if (!node) {
        //printf("Error retrieving inode.\n");
        send_to_buffer(write_buf, "Error retrieving inode", 23);
        return;
    }
    node->type = FILE_TYPE;
    node->size = 0;
    node->last_modified = time(NULL);
    node->parent_inode = current_directory_inode;
    //write_inode_to_disk(node, inode_index);
    add_directory_entry(filename, current_directory_inode, inode_index);
    update_file_size(current_directory_inode, sizeof(directory_entry)+sizeof(inode));
    send_to_buffer(write_buf, "file created!", 14);
    return;
}

void mkdir(tcp_buffer *write_buf, char *args, int len) {
    char filename[MAX_FILENAME_LEN + 1];  // Allocate buffer for filename
    if (sscanf(args, "%200s", filename) != 1 || strlen(filename) > MAX_FILENAME_LEN) {

        send_to_buffer(write_buf, "Error: Invalid ", 32);
        return;
    }
    // if (get_directory_entry(filename) != NULL) {
```

```
    //     //printf("File %s already exists.\n", filename);
    //     send_to_buffer(write_buf, "already exists", 15);
    //     return 0;
    // }

    int inode_index = allocate_inode();
    if (inode_index == -1) {
        send_to_buffer(write_buf, "No free inodes available", 25);
        return;
    }

    inode *node = get_inode(inode_index);
    if (!node) {
        //printf("Error retrieving inode.\n");
        send_to_buffer(write_buf, "Error retrieving inode", 23);
        return;
    }
    node->type = DIR_TYPE;
    node->size = 0;
    node->last_modified = time(NULL);
    node->parent_inode = current_directory_inode;
    add_directory_entry(filename, current_directory_inode, inode_index);
    update_file_size(current_directory_inode, sizeof(directory_entry)+sizeof(inode)
    );
    send_to_buffer(write_buf, "file created!", 14);
    return;
}
```

These two commands are very similar. We allocate an inode, add the corresponding entry and update the file size(the two functions will be given below).

```
int add_directory_entry(const char *name, int cur_inode_index, int tar_inode_index)
{
    inode *cnode = get_inode(cur_inode_index);
    int block_number;
    for (int i = 0; i < NUM_DIRECT_POINTERS; i++) {
        if (cnode->direct_blocks[i] == -1) {
            block_number = allocate_block();
            cnode->direct_blocks[i] = block_number;
            break;
        }
    }
    directory_entry my_entry;
    my_entry.inode_index=tar_inode_index;
    strcpy(my_entry.name,name);

    char data[SECTOR_SIZE];


    directory_entry_to_string(&my_entry, data, sizeof(data));
    printf("data: %s\n",data);
    write_data_to_disk(data, block_number, nsec);

    return 0;
}
void update_file_size(int inode_index, int size_change) {
    inode *node = get_inode(inode_index);
    if (!node) {
        printf("Error: Inode %d not found\n", inode_index);
        return;
```

```
    }

    // Update the size of the current inode
    node->size += size_change;
    node->last_modified = time(NULL);

    // Write the updated inode to disk
    write_inode_to_disk(node, inode_index);

    // If it's not the root directory, recursively update the parent inode
    if (inode_index != 0 && inode_index != node->parent_inode) {
        update_file_size(node->parent_inode, size_change);
    }
}
```

2.5 ls command

```
void ls(tcp_buffer *write_buf, char *args, int len)
{
    inode *cnode = get_inode(current_directory_inode);
    if (!cnode) {
        send_to_buffer(write_buf, "Error retrieving current directory inode", 41);
        return ;
    }
    char buffer[BUFFER_SIZE];
    int offset = snprintf(buffer, sizeof(buffer), "Current directory size: %d bytes
    , last modified: %s\n",cnode->size, ctime(&cnode->last_modified));
    printf("Current directory size: %d bytes, last modified: %s\n",cnode->size,
    ctime(&cnode->last_modified));
    for (int i = 0; i < NUM_DIRECT_POINTERS; i++) {
        if (cnode->direct_blocks[i] != -1) {
            char data[BUFFER_SIZE];
            int block_number = cnode->direct_blocks[i];
            read_block(block_number, data, sizeof(data));
            printf("data: %s\n",data);
            directory_entry entry;
            string_to_directory_entry(data, &entry);
            offset += snprintf(buffer + offset, sizeof(buffer) - offset, "%s\t",
    entry.name);
        }
    }
    send_to_buffer(write_buf, buffer, strlen(buffer) + 1);
    return;
}
```

The `ls` function traverses each block of the current directory, copies the contents into a single string, and finally sends this string.

1. The function starts by retrieving the inode of the current directory using `get_inode(current_directory_inod`
2. If the inode cannot be retrieved, an error message is sent to the write buffer, and the function returns.
3. A buffer is initialized to store the directory's information. The size of the directory and the last modified timestamp are added to the buffer.
4. The function iterates through each direct block pointer in the inode:

- If a direct block pointer is valid (not equal to -1), the block is read using `read_block()`, and the data is stored in a temporary buffer.
- The data is then converted into a directory entry structure using `string_to_directory_entry()`.
- The name of the directory entry is appended to the main buffer.

5. Finally, the complete buffer containing all the directory entries' names is sent to the write buffer using `send_to_buffer()`.

This approach ensures that all the directory contents are collected and sent in one unified message.

2.6 rm command

```
void rm(tcp_buffer *write_buf, char *args, int len) {
    char filename[MAX_FILENAME_LEN + 1];  // Allocate buffer for filename
    if (sscanf(args, "%200s", filename) != 1 || strlen(filename) > MAX_FILENAME_LEN
    ) {
        send_to_buffer(write_buf, "Error: Invalid ", 32);
        return;
    }
    directory_entry *entry = get_directory_entry(filename);
    if (entry == NULL) {
        send_to_buffer(write_buf, "File not found", 15);
        return;
    }
    inode *node = get_inode(entry->inode_index);
    if (!node) {
        send_to_buffer(write_buf, "Error retrieving inode", 23);
        free(entry);
        return;
    }
    // Free the blocks used by the file
    for (int i = 0; i < NUM_DIRECT_POINTERS; i++) {
        if (node->direct_blocks[i] != -1) {
            free_blocks[node->direct_blocks[i]] = 1;
            node->direct_blocks[i] = -1;
        }
    }
    // Free the inode
    node->is_used = 0;
    inodes[current_directory_inode].last_modified = time(NULL);
    // Remove the directory entry
    inode *cnode = get_inode(current_directory_inode);
    for (int i = 0; i < NUM_DIRECT_POINTERS; i++) {
        if (cnode->direct_blocks[i] != -1) {
            char data[BUFFER_SIZE];
            int block_number = cnode->direct_blocks[i];
            read_block(block_number, data, sizeof(data));
            directory_entry dir_entry;
            string_to_directory_entry(data, &dir_entry);
            if (dir_entry.inode_index == entry->inode_index) {
                cnode->direct_blocks[i] = -1;
                free_blocks[block_number] = 1;
                break;
            }
        }
    }
```

```
44      update_file_size(current_directory_inode, -(sizeof(directory_entry) +sizeof(
        inode) +node->size));
45      send_to_buffer(write_buf, "file deleted", 13);
46      free(entry);
47      return;
48  }
```

The `rm` function is responsible for deleting a file from the filesystem.

1. The function starts by extracting the filename from the arguments. It verifies the filename length and validity.
2. It retrieves the directory entry corresponding to the filename using `get_directory_entry()`.
3. If the file is not found or there is an error retrieving the inode, an error message is sent to the write buffer.
4. The function then frees the blocks used by the file by iterating through the direct block pointers of the inode and marking them as free.
5. It marks the inode as unused and updates the last modified time of the current directory's inode.
6. The function proceeds to remove the directory entry from the current directory by finding and marking its block as free.
7. The file size is updated, and a success message is sent to the write buffer.
8. Finally, the directory entry is freed.

This ensures the file is completely removed from the filesystem and the corresponding resources are freed.

2.7 rmdir command

The `rmdir_r` and `cmdrmdir` functions are responsible for recursively deleting a directory and its contents from the filesystem.

`cmdrmdir` Function

1. Extracts the directory name from the arguments and validates its length.
2. Retrieves the directory entry corresponding to the directory name using `get_directory_entry(dirname)`.
3. Checks if the directory entry exists and is of directory type. If not, sends an error message and returns.
4. Calls `rmdir_r` to recursively remove the directory and its contents.
5. Removes the directory entry from the current directory and frees the corresponding block.
6. Updates the last modified time and file size of the current directory's inode.
7. Sends a success message to the write buffer.

```c
void cmdrmdir(tcp_buffer *write_buf, char *args, int len) {
    char dirname[MAX_FILENAME_LEN + 1];
    if (sscanf(args, "%200s", dirname) != 1 || strlen(dirname) > MAX_FILENAME_LEN)
    {
        send_to_buffer(write_buf, "Error: Invalid ", 32);
        return;
    }
    directory_entry *entry = get_directory_entry(dirname);
    if (entry == NULL) {
        send_to_buffer(write_buf, "Directory not found", 20);
        return;
    }
    inode *node = get_inode(entry->inode_index);
    if (!node || node->type != DIR_TYPE) {
        send_to_buffer(write_buf, "Not a directory", 17);
        free(entry);
        return;
    }
    int tmpsize=node->size;
    rmdir_r(entry->inode_index,write_buf,args,len);
    inode *cnode = get_inode(current_directory_inode);
    for (int i = 0; i < NUM_DIRECT_POINTERS; i++) {
        if (cnode->direct_blocks[i] != -1) {
            char data[BUFFER_SIZE];
            int block_number = cnode->direct_blocks[i];
            read_block(block_number, data, sizeof(data));
            directory_entry dir_entry;
            string_to_directory_entry(data, &dir_entry);
            if (dir_entry.inode_index == entry->inode_index) {
                cnode->direct_blocks[i] = -1;
                free_blocks[block_number] = 1;
                break;
            }
        }
    }
    inodes[current_directory_inode].last_modified = time(NULL);
    update_file_size(current_directory_inode, -(sizeof(directory_entry) + sizeof(
    inode) + node->size));
    send_to_buffer(write_buf, "directory deleted", 18);
    free(entry);
    return;
}
```

`rmdir_r` Function

1. Retrieves the inode of the directory using `get_inode(inode_index)`.
2. Checks if the inode is valid and is of directory type. If not, sends an error message and returns.
3. Iterates through each direct block pointer of the inode:
   - Reads the block data and converts it to a directory entry.
   - If the directory entry is another directory, calls `rmdir_r` recursively.
   - If the directory entry is a file, calls `rm` to remove the file.
4. Frees the blocks used by the directory and marks them as free.

5. Marks the inode as unused and updates the free blocks list.

```
void rmdir_r(int inode_index,tcp_buffer *write_buf, char *args, int len)
{

    inode *node = get_inode(inode_index);
    if (!node || node->type != DIR_TYPE) {
        send_to_buffer(write_buf, "Not a directory", 17);

        return;
    }
    for (int i = 0; i < NUM_DIRECT_POINTERS; i++) {
        if (node->direct_blocks[i] != -1) {
            char data[BUFFER_SIZE];
            int block_number = node->direct_blocks[i];
            read_block(block_number, data, sizeof(data));
            directory_entry temp_entry;
            string_to_directory_entry(data, &temp_entry);

            if(inodes[temp_entry.inode_index].type==DIR_TYPE){
                rmdir_r(temp_entry.inode_index,write_buf,args,len);
            }
            else
            {
                if(inodes[temp_entry.inode_index].type==FILE_TYPE){
                    rm(write_buf,temp_entry.name,strlen(temp_entry.name));
                }
            }

        }
    }
    // Free the blocks used by the directory
    for (int i = 0; i < NUM_DIRECT_POINTERS; i++) {
        if (node->direct_blocks[i] != -1) {
            free_blocks[node->direct_blocks[i]] = 1;
            node->direct_blocks[i] = -1;
        }
    }
    // Free the inode

    node->is_used = 0;
    free_blocks[ncyl*nsec-1-inode_index] = 1;
    // Remove the directory entry

    return;
}
```

These functions ensure that a directory and all its contents are properly deleted and the filesystem is updated accordingly.
The `cd` function changes the current working directory in the filesystem.

2.8 cd command

1. Extracts the path from the arguments and validates its length.

2. Checks if the path is the root directory ("/"). If so, sets the current directory to root and updates the path, then returns.

3. Tokenizes the path and initializes `inode_index` to the current directory inode.
4. Iterates through each token in the path:
   - If the token is ".", it continues to the next token.
   - If the token is "..", it retrieves the parent inode of the current directory and updates `inode_index`.
   - For other tokens, it retrieves the directory entry using `get_directory_entry(token)`.
   - Checks if the directory entry exists and is of directory type. If not, sends an error message and returns.
   - Updates `inode_index` to the inode index of the directory entry.
5. Sets the current directory inode to `inode_index` and updates the current path.
6. Sends a success message to the write buffer indicating the directory has been changed.

```
void cd(tcp_buffer *write_buf, char *args, int len) {
    char path[MAX_FILENAME_LEN + 1];
    if (sscanf(args, "%200s", path) != 1 || strlen(path) > MAX_FILENAME_LEN) {
        send_to_buffer(write_buf, "Error: Invalid ", 32);
        return;
    }
    if (strcmp(path, "/") == 0) {
        current_directory_inode = 0;
        strcpy(current_path, "/");
        send_to_buffer(write_buf, "Changed to root directory", 27);
        return;
    }
    char *token;
    char temp_path[MAX_FILENAME_LEN + 1];
    strcpy(temp_path, path);
    token = strtok(temp_path, "/");
    int inode_index = current_directory_inode;
    while (token != NULL) {
        if (strcmp(token, ".") == 0) {
            token = strtok(NULL, "/");
            continue;
        }
        if (strcmp(token, "..") == 0) {
            inode *current_inode = get_inode(inode_index);
            if (current_inode == NULL) {
                send_to_buffer(write_buf, "Error retrieving current directory inode
    ", 41);
                return;
            }
            inode_index = current_inode->parent_inode;
            token = strtok(NULL, "/");
            continue;
        }
        printf("1\n");
        directory_entry *entry = get_directory_entry(token);
        if (entry == NULL) {
            send_to_buffer(write_buf, "Directory not found", 20);
            return;
        }
        printf("2\n");
```

```
        printf("entry->inode_index: %d\n",entry->inode_index);
        inode *node = get_inode(entry->inode_index);
        if (node == NULL || node->type != DIR_TYPE) {
            send_to_buffer(write_buf, "Not a directory", 17);
            return;
        }
        printf("3\n");
        inode_index = entry->inode_index;
        token = strtok(NULL, "/");
    }
    current_directory_inode = inode_index;
    update_current_path(path);
    send_to_buffer(write_buf, "Directory changed", 18);
    return;
}
```

2.9 w command

The `write_to_file` function overwrites the contents of a specified file with a given amount of data. If the new data is longer than the existing data, the file is extended. If the new data is shorter, the file is truncated.

1. Parses the input arguments to extract the filename, data length, and data.
2. Validates the filename and data length. If invalid, sends an error message and returns.
3. Retrieves the directory entry for the filename. If the file is not found or is not a regular file, sends an error message and returns.
4. Calculates the number of blocks needed to store the new data.
5. Allocates additional blocks if the new data is longer than the existing data.
6. Writes the data to the allocated blocks. For each block:
   - Determines the number of bytes to write to the block.
   - Copies the data to a temporary buffer.
   - Writes the buffer to the disk.
7. Frees any unneeded blocks if the new data is shorter than the existing data.
8. Updates the inode's size and modification time.
9. Writes the updated inode back to the disk and updates the directory's file size.
10. Sends a success message to the write buffer.

This function ensures that the file's contents are accurately updated to reflect the new data, adjusting the file's length as necessary.

```
void write_to_file(tcp_buffer *write_buf, char *args, int len) {
    char filename[MAX_FILENAME_LEN + 1];
    int data_len;
    char data[BUFFER_SIZE];

```

```
    // Parse the input arguments
    if (sscanf(args, "%200s %d %[^\n]", filename, &data_len, data) != 3 || strlen(filename) > MAX_FILENAME_LEN || data_len > BUFFER_SIZE) {
        send_to_buffer(write_buf, "Error: Invalid arguments", 25);
        return;
    }

    directory_entry *entry = get_directory_entry(filename);
    if (entry == NULL) {
        send_to_buffer(write_buf, "File not found", 15);
        return;
    }

    inode *node = get_inode(entry->inode_index);
    if (!node || node->type != FILE_TYPE) {
        send_to_buffer(write_buf, "Not a file", 10);
        free(entry);
        return;
    }

    // Calculate the number of blocks needed
    int num_blocks_needed = (data_len + SECTOR_SIZE - 1) / SECTOR_SIZE;

    // If the new data is longer than the old data, allocate additional blocks if needed
    for (int i = 0; i < num_blocks_needed; i++) {
        if (node->direct_blocks[i] == -1) {
            int block_number = allocate_block();
            if (block_number == -1) {
                send_to_buffer(write_buf, "No free blocks available", 25);
                free(entry);
                return;
            }
            node->direct_blocks[i] = block_number;
        }
    }

    data_len=data_len+1;

    for (int i = 0; i < num_blocks_needed; i++) {
        int block_number = node->direct_blocks[i];

        int bytes_to_write = (data_len > SECTOR_SIZE) ? SECTOR_SIZE : data_len;
        char block_data[SECTOR_SIZE+1] = {'\0'};
        strncpy(block_data, data + (i * SECTOR_SIZE), bytes_to_write);
        write_data_to_disk(block_data, block_number, nsec);
        data_len -= bytes_to_write;
    }

    // If the new data is shorter than the old data, free the unneeded blocks
    for (int i = num_blocks_needed; i < NUM_DIRECT_POINTERS; i++) {
        if (node->direct_blocks[i] != -1) {
            free_blocks[node->direct_blocks[i]] = 1;
            node->direct_blocks[i] = -1;
        }
    }

    // Update the inode size and modification time
    int pre_size = node->size;
```

```
    node->size = num_blocks_needed * SECTOR_SIZE;
    node->last_modified = time(NULL);
    write_inode_to_disk(node, entry->inode_index);
    update_file_size(current_directory_inode, node->size - pre_size);
    send_to_buffer(write_buf, "file written", 13);
    free(entry);
    return;
}
```

2.10 cat command

The `cat` function reads the contents of a specified file and returns the data.

1. Parses the input arguments to extract the filename and validates its length. If invalid, sends an error message and returns.
2. Retrieves the directory entry for the specified filename. If the file is not found, sends an error message and returns.
3. Retrieves the inode associated with the directory entry. If the inode is not found or is not a regular file, sends an error message and returns.
4. Initializes buffers to store the file data and the final result.
5. Iterates through each direct block pointer in the inode:
   - Reads the data block by block.
   - Appends each block's data to the result buffer.
   - Keeps track of the total bytes read.
6. If no data is read, sends a message indicating the file is empty.
7. Otherwise, sends the accumulated data from the result buffer to the write buffer.
8. Frees the directory entry memory.

This function ensures that the contents of the specified file are accurately read and returned, allowing the user to view the file's data.

```
void cat(tcp_buffer *write_buf, char *args, int len) {
    char filename[MAX_FILENAME_LEN + 1];
    if (sscanf(args, "%200s", filename) != 1 || strlen(filename) > MAX_FILENAME_LEN
    ) {
        send_to_buffer(write_buf, "Error: Invalid arguments", 25);
        return;
    }

    directory_entry *entry = get_directory_entry(filename);
    if (entry == NULL) {
        send_to_buffer(write_buf, "File not found", 15);
        return;
    }

    inode *node = get_inode(entry->inode_index);
    if (!node || node->type != FILE_TYPE) {
        send_to_buffer(write_buf, "Not a file", 10);
```

```
        free(entry);
        return;
    }

    char buffer[BUFFER_SIZE];
    char result[BUFFER_SIZE]="";  //
    int total_bytes_read = 0;

    for (int i = 0; i < NUM_DIRECT_POINTERS && node->direct_blocks[i] != -1; i++) {
        read_block(node->direct_blocks[i], buffer, sizeof(buffer));
        printf("buffer: %s\n",buffer);
        strncat(result, buffer, SECTOR_SIZE);
        printf("result: %s\n",result);
        total_bytes_read += strlen(buffer);
    }

    if (total_bytes_read == 0) {
        send_to_buffer(write_buf, "File is empty", 14);
    } else {
        send_to_buffer(write_buf, result, strlen(result) + 1);
    }

    free(entry);
}
```

2.11 i command

The `insert_into_file` function inserts data into a specified file at a given position. If the position exceeds the file's size, the data is appended to the end of the file.

1. Parses the input arguments to extract the filename, position, data length, and data.
2. Validates the filename and data length. If invalid, sends an error message and returns.
3. Retrieves the directory entry for the specified filename. If the file is not found, sends an error message and returns.
4. Retrieves the inode associated with the directory entry. If the inode is not found or is not a regular file, sends an error message and returns.
5. Reads the existing file content into a buffer.
6. Ensures the specified position is within the current content length. If the position exceeds the length, sets the position to the end of the content.
7. Inserts the new data at the specified position within the existing content, creating a new content buffer.
8. Calculates the number of blocks needed to store the new content.
9. Writes the new content to the file's blocks. Allocates additional blocks if needed.
10. Frees any unused blocks if the new content is shorter than the old content.
11. Updates the inode's size and modification time.

12. Writes the updated inode back to the disk and updates the directory's file size.
13. Sends a success message to the write buffer.
14. Frees the directory entry memory.

This function ensures that the specified data is accurately inserted into the file at the given position, adjusting the file's length as necessary.

2.12 d command

The `delete_from_file` function deletes a specified length of data from a given position in a file. If the specified position exceeds the file's size, deletion occurs from the end of the file.

1. Parses the input arguments to extract the filename, position, and length of data to be deleted.
2. Validates the filename and arguments. If invalid, sends an error message and returns.
3. Retrieves the directory entry for the specified filename. If the file is not found, sends an error message and returns.
4. Retrieves the inode associated with the directory entry. If the inode is not found or is not a regular file, sends an error message and returns.
5. Reads the existing file content into a buffer.
6. Ensures the specified position is within the current content length. If the position exceeds the length, sets the position to the end of the content.
7. Calculates the actual length of data to delete, ensuring it does not exceed the file size.
8. Creates the new content by deleting the specified data from the existing content.
9. Calculates the number of blocks needed for the new content.
10. Writes the new content to the file's blocks. Allocates additional blocks if needed.
11. Frees any unused blocks if the new content is shorter than the old content.
12. Updates the inode's size and modification time.
13. Writes the updated inode back to the disk and updates the directory's file size.
14. Sends a success message to the write buffer.
15. Frees the directory entry memory.

This function ensures that the specified data is accurately deleted from the file, adjusting the file's length as necessary.

```
void delete_from_file(tcp_buffer *write_buf, char *args, int len) {
    char filename[MAX_FILENAME_LEN + 1];
    int position, delete_len;

    // Parse the input arguments
```

```
    if (sscanf(args, "%200s %d %d", filename, &position, &delete_len) != 3 ||
strlen(filename) > MAX_FILENAME_LEN) {
        send_to_buffer(write_buf, "Error: Invalid arguments", 25);
        return;
    }

    directory_entry *entry = get_directory_entry(filename);
    if (entry == NULL) {
        send_to_buffer(write_buf, "File not found", 15);
        return;
    }

    inode *node = get_inode(entry->inode_index);
    if (!node || node->type != FILE_TYPE) {
        send_to_buffer(write_buf, "Not a file", 10);
        free(entry);
        return;
    }

    // Read the existing file content
    char current_content[BUFFER_SIZE] = "";
    char buffer[BUFFER_SIZE];
    int total_bytes_read = 0;
    for (int i = 0; i < NUM_DIRECT_POINTERS && node->direct_blocks[i] != -1; i++) {
        read_block(node->direct_blocks[i], buffer, sizeof(buffer));
        strncat(current_content, buffer, SECTOR_SIZE);
        total_bytes_read += strlen(buffer);
    }

    // Ensure position is within the current content length
    if (position > total_bytes_read) {
        position = total_bytes_read;
    }

    // Calculate the actual length to delete
    int actual_delete_len = (position + delete_len > total_bytes_read) ?
total_bytes_read - position : delete_len;

    // Create the new content by deleting the specified data
    char new_content[BUFFER_SIZE];
    snprintf(new_content, sizeof(new_content), "%.*s%s", position, current_content,
     current_content + position + actual_delete_len);

    // Calculate the number of blocks needed for the new content
    int new_content_len = strlen(new_content);
    int num_blocks_needed = (new_content_len + SECTOR_SIZE - 1) / SECTOR_SIZE;

    // Write the new content to the file's blocks
    for (int i = 0; i < num_blocks_needed; i++) {
        if (node->direct_blocks[i] == -1) {
            int block_number = allocate_block();
            if (block_number == -1) {
                send_to_buffer(write_buf, "No free blocks available", 25);
                free(entry);
                return;
            }
            node->direct_blocks[i] = block_number;
        }
        int block_number = node->direct_blocks[i];
```

```
        int bytes_to_write = (new_content_len > SECTOR_SIZE) ? SECTOR_SIZE :
    new_content_len;
        char block_data[SECTOR_SIZE+1] = {'\0'};
        strncpy(block_data, new_content + (i * SECTOR_SIZE), bytes_to_write);
        write_data_to_disk(block_data, block_number, nsec);
        new_content_len -= bytes_to_write;
    }

    // Free any unused blocks if the new content is shorter than the old content
    for (int i = num_blocks_needed; i < NUM_DIRECT_POINTERS; i++) {
        if (node->direct_blocks[i] != -1) {
            free_blocks[node->direct_blocks[i]] = 1;
            node->direct_blocks[i] = -1;
        }
    }

    // Update the inode size and modification time
    int pre_size = node->size;
    node->size = strlen(new_content);
    node->last_modified = time(NULL);
    write_inode_to_disk(node, entry->inode_index);
    update_file_size(current_directory_inode, node->size - pre_size);
    send_to_buffer(write_buf, "file updated", 13);
    free(entry);
}
```

3 Step 3: Support Multi-users

3.1 Some Modifications to Global Settings

To support multi-user functionality in our file system, several global settings and structures were modified and introduced:

- **User Structure:** A structure to store user information, including username, password, and home directory inode.
- **Global Arrays:** Arrays to store users and free blocks.
- **Current User:** A global variable to keep track of the currently logged-in user.
- **User Functions:** Functions to encode/decode user data and manage user operations.

```
#define MAX_USERS 100
#define MAX_PASSWORD_LEN 50

typedef struct {
    int is_used;  // 1 indicates in use, 0 indicates not in use
    char username[MAX_FILENAME_LEN];
    char password[MAX_PASSWORD_LEN];
    int home_inode;
} user;

user users[MAX_USERS];
int current_user_id = -1;  // -1 means no user is logged in

// Encode function
```

```
void user_to_string(user* usr, char* buffer, int buffer_size) {
    snprintf(buffer, buffer_size, "%d,%s,%s,%d", usr->is_used, usr->username, usr->password, usr->home_inode);
}

// Decode function
void string_to_user(const char* str, user* usr) {
    sscanf(str, "%d,%[^,],%[^,],%d", &usr->is_used, usr->username, usr->password, &usr->home_inode);
}
```

Explanation:
The above code defines a user structure to store user information, including a flag indicating whether the user slot is used, the username, password, and the home directory's inode index. Additionally, it provides functions to encode and decode user data to and from strings for easy storage and retrieval from disk.

3.2 User Management Functions

```
void initialize_users() {
    for (int i = 0; i < MAX_USERS; i++) {
        users[i].is_used = 0;
    }
}

int allocate_user() {
    for (int i = 0; i < MAX_USERS; i++) {
        if (!users[i].is_used) {
            users[i].is_used = 1;
            return i;
        }
    }
    return -1;
}

void write_users_to_disk() {
    char buffer[BUFFER_SIZE];
    for (int i = 0; i < MAX_USERS; i++) {
        if (users[i].is_used) {
            user_to_string(&users[i], buffer, sizeof(buffer));
            int block_number = size - 1 - MAX_INODES - i; // Correct block number for user data
            write_data_to_disk(buffer, block_number, nsec);
        }
    }
}

void read_users_from_disk() {
    char buffer[BUFFER_SIZE];
    for (int i = 0; i < MAX_USERS; i++) {
        int block_number = size - 1 - MAX_INODES - i; // Correct block number for user data
        read_block(block_number, buffer, sizeof(buffer));
        if (strlen(buffer) > 0) {
            string_to_user(buffer, &users[i]);
        }
    }
}
```

Explanation:
The user management functions include:

- `initialize_users()`: Initializes the user array by setting all slots to unused.
- `allocate_user()`: Finds a free slot in the user array, marks it as used, and returns the index.
- `write_users_to_disk()`: Encodes and writes all user data to the disk.
- `read_users_from_disk()`: Reads and decodes all user data from the disk.

3.3 User Commands: adduser, deluser, login, logout

```
void add_user(tcp_buffer *write_buf, char *args, int len) {
    char username[MAX_FILENAME_LEN + 1];
    char password[MAX_PASSWORD_LEN + 1];
    if (sscanf(args, "%200s %50s", username, password) != 2) {
        send_to_buffer(write_buf, "Error: Invalid arguments", 25);
        return;
    }
    int user_id = allocate_user();
    if (user_id == -1) {
        send_to_buffer(write_buf, "Error: No free user slots", 25);
        return;
    }
    strncpy(users[user_id].username, username, MAX_FILENAME_LEN);
    strncpy(users[user_id].password, password, MAX_PASSWORD_LEN);
    mkdir(write_buf, username, strlen(username));
}

void delete_user(tcp_buffer *write_buf, char *args, int len) {
    char username[MAX_FILENAME_LEN + 1];
    if (sscanf(args, "%200s", username) != 1) {
        send_to_buffer(write_buf, "Error: Invalid arguments", 25);
        return;
    }
    for (int i = 0; i < MAX_USERS; i++) {
        if (users[i].is_used && strcmp(users[i].username, username) == 0) {
            users[i].is_used = 0;
            cmdrmdir(write_buf, username, strlen(username));
            send_to_buffer(write_buf, "User deleted successfully", 26);
            return;
        }
    }
    send_to_buffer(write_buf, "Error: User not found", 21);
}

void login(tcp_buffer *write_buf, char *args, int len) {
    char username[MAX_FILENAME_LEN + 1];
    char password[MAX_PASSWORD_LEN + 1];
    if (sscanf(args, "%200s %50s", username, password) != 2) {
        send_to_buffer(write_buf, "Error: Invalid arguments", 25);
        return;
    }
    int user_id = authenticate_user(username, password);
    if (user_id == -1) {
        send_to_buffer(write_buf, "Error: Authentication failed", 29);
        return;
```

```
    }
    current_user_id = user_id;
    send_to_buffer(write_buf, "Login successful", 17);
}

void logout(tcp_buffer *write_buf, char *args, int len) {
    if (current_user_id == -1) {
        send_to_buffer(write_buf, "No user is logged in", 21);
        return;
    }
    current_user_id = -1;
    send_to_buffer(write_buf, "Logout successful", 18);
}
```

Explanation:

- `add_user()`: Adds a new user to the system by allocating a user slot, setting the username and password, and creating a home directory for the user.
- `delete_user()`: Deletes a user by marking the user slot as unused and removing the user's home directory.
- `login()`: Authenticates a user and sets the current user ID.
- `logout()`: Logs out the current user by resetting the current user ID.

3.4 File and Directory Permissions

```
void chmod(tcp_buffer *write_buf, char *args, int len) {
    char filename[MAX_FILENAME_LEN + 1];
    int new_permissions;

    if (sscanf(args, "%200s %d", filename, &new_permissions) != 2 || strlen(
    filename) > MAX_FILENAME_LEN || (new_permissions != 0 && new_permissions != 1))
     {
        send_to_buffer(write_buf, "Error: Invalid arguments", 25);
        return;
    }

    directory_entry *entry = get_directory_entry(filename);
    if (entry == NULL) {
        send_to_buffer(write_buf, "File or directory not found", 28);
        return;
    }

    inode *node = get_inode(entry->inode_index);
    if (!node) {
        send_to_buffer(write_buf, "Error retrieving inode", 23);
        free(entry);
        return;
    }

    if (node->owner_id != current_user_id) {
        send_to_buffer(write_buf, "Permission denied", 18);
        free(entry);
        return;
    }

```

```
    node->permissions = new_permissions;
    node->last_modified = time(NULL);
    write_inode_to_disk(node, entry->inode_index);

    send_to_buffer(write_buf, "Permissions updated", 20);
    free(entry);
}
```

Explanation:
The ‘chmod‘ command allows users to change the permissions of files or directories. It validates the arguments, retrieves the directory entry and inode, checks ownership, updates permissions, and writes the changes to disk. It ensures that only the owner can change the permissions, maintaining the security of the file system.

4 Summary

In this project, we designed and implemented a basic file system with multi-user support. Our file system simulates a physical disk with sectors and cylinders, and provides essential file and directory operations such as creation, deletion, reading, and writing. We utilized inodes to manage metadata and an array to track free blocks, ensuring efficient storage management. Additionally, we introduced user management functionalities, including adding, deleting, logging in, and logging out users. Permissions were implemented to control access to files and directories, enhancing the security of the file system. The combination of these features provides a comprehensive, multi-user file system suitable for various use cases.