Report of Project3

Yiheng Bing

May 31, 2024

Contents

1 Basic Disk Server and two kinds of Clients

1.1 Description

A simple disk server (BDS) that can analyse the order of disk client (BDC_command) and execute the order. The server can read and write files in the disk-like file and simulate the disk-like file system. The server can also execute the random orders from the random order client (BDC_random).

In this step we should not only implement the following orders but also virtual track-to-track delay by using usleep() function.

The orders are as follows:

1. I: information of the disk-like file(num of cylinders and sectors per cylinder)
2. R <cylinder> <sector>: read the data in the specific cylinder and sector
3. W <cylinder> <sector> <length> <data>: write a fixed length of data in the specific cylinder and sector
4. E: exit the server

The disk-like file is a binary file that can be read and written by the server. The file is divided into several cylinders, and each cylinder is divided into several sectors. The server can read and write the data in the file by the order of the client.

1.2 usage

```
./BDS <disk-like file name> <num of cylinders>
    <num of sectors> <track to track delay> <port>
./BDC_command <port>
./BDC_random <num of random orders> <port>
```

1.3 Key functions

For example, the function to get data from client's order is not so easy. Because it's natural to contain space in the data. However, commom order analyser will divide the order by space.such as *strtok* function. So I have to write my own function to analyse which part is a different parameter while other spaces are just part of the data.

```
char **strspace(
    char *token, const char *delim,
    int count) {  // To ensure the space
    // in data can be read correctly
    // length is the length of the last token
    char **tokens = (char **)malloc
        (count * sizeof(char *));
    for (int i = 0; i < count; i++) {
        tokens[i] = (char *)malloc(
```

```
            300 * sizeof(char));
            // Assume max token length is 256
    }
    int i = 0;
    int c = 0;
    while (c < count && token[i] != '\0') {
        if (!notin(token[i], delim)) {
            while (!notin(token[i], delim)
                && token[i] != '\0') {
                i++;
            }
        } else {
            int j = 0;
            if (c + 1 < count) {
                while (notin(token[i], delim)
                    && token[i] != '\0') {
                    tokens[c][j] = token[i];
                    i++;
                    j++;
                }
                tokens[c][j] = '\0';
                c++;
            } else
                break;
        }
        int length = atoi(tokens[c - 1]);
        memcpy(tokens[c], token + i, length);
        tokens[c][length] = '\0';
    }
    return tokens;
}
```

In this function, I use a pointer to the token string and a delim string to divide the token. The count parameter is the number of tokens I want to get. The function will return a pointer to a pointer to the tokens.

1.4 Execution loop

In the two following figures, we can see the execution loop of the BDS with two different kinds of clients, BDC_command and BDC_random.

The BDC_command client sends the orders one by one, and the BDS executes the orders in the order of the client. The BDC_random client sends "I" instruction first to get the cylinders and sectors number of the BDS, and then send random orders to the BDS.

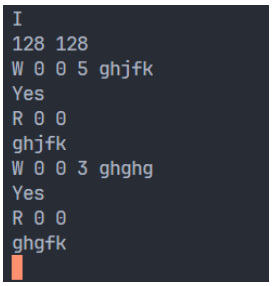

Figure 1: BDC_command

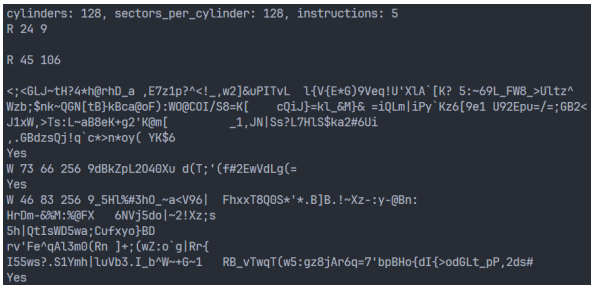

Figure 2: BDC_random

2 File Server and Client

2.1 Description

In the second step, I implemented a toy file system server and client. The server can read and write data in the disk-like file system and the client can send orders to the server to read and write files. Just as the slides show, the file server plays a role of the basic disk client at the same time.

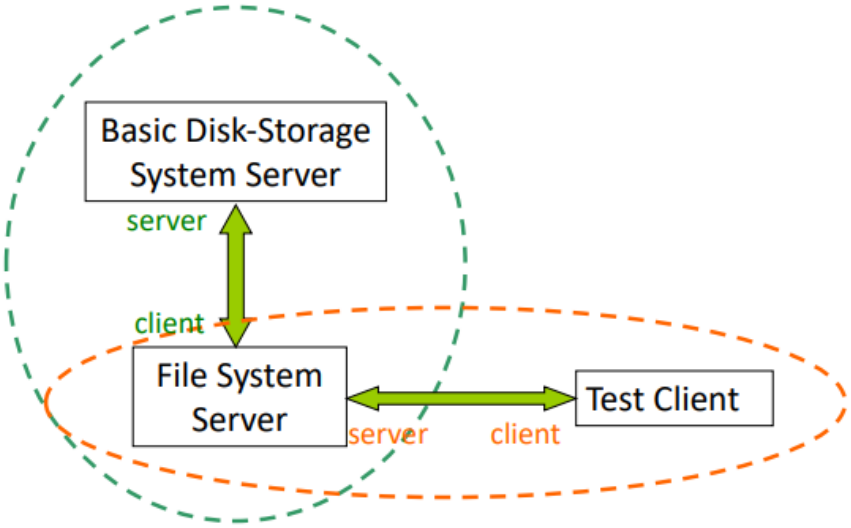

Figure 3: File System Architecture

And then for the underlying structure of the file system, the file server should analyse each order from the client and execute the order, transform them into simple orders mentioned and implemented in step1. In order to achieve this, I have to implement a file system structure as the following figure shows:

2.2 usage

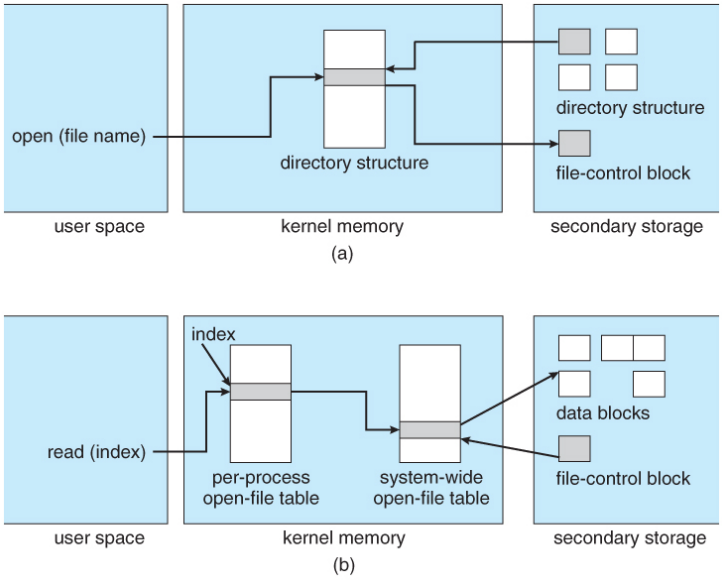

Figure 4: Underlying Structure

```
./FS <diskserver address> <diskserver port>
 <fileserver port>
./FC <fileserver address> <fileserver port>
```

2.3 Basic logic

To implement the directory and file hierarchy, I use a tree structure to store the information of the files and directories. Each file or directory would be a node in the tree. When the file system is initialized, the root node is created which is also the root directory.

When a new file or directory is created, a new node is created and added to the tree. When a file is deleted, the node is removed from the tree. When a directory is deleted, all the nodes in the directory are removed from the tree.

However, when I need to delete some node, I don't need to delete the node in the tree really. I have created a bit-map to indicate all blocks in the system. It's called bit-map just because each bit of the map can only indicated to one block(0 for free or 1 for occupied). And also each node has it' links to others.In my toy file system, each node is designed to have 48 pointers to other structures. Of course, they are not real pointer, just the index of the block in the blocks. One more good thng about the bit-map is that I can easily find a free node or block in the system to store a new structure created no long ago.

In the step1, our write or read opreation is based on the blocks. So we can store one node in such one block. And the bit-map can help us to find a free block to store the node. If one node is a file, it,s parent pointer will point to a directory the node in, and other pointers to the blocks storing the data of the file. If one node is a directory, it's parent pointer will point to the parent directory, and other pointers to other nodes, files or directories are all possible

in this directory.

To allocate all block information, we need not only the bit-maps, but also the super block. The super block is the first block in the disk-like file system. It contains the information of the file system, such as the number of blocks, the number of free blocks, the number of used blocks, the root directory the time information and so on.

Before we use such a file system, we need to format the file system first. The format operation will create the super block, the bit-maps and the root directory. The super block will be written to the first block of the disk-like file system, the bit-maps will be written to the following 16 blocks, and the root directory will be written to the 17th block.

Some important codes are shown as below:

```
int init_spb() {
    super_block *spb = (super_block *)malloc
        (sizeof(super_block));
    spb->s_root_bp[0] = 0;
    spb->s_root_bp[1] = 17;
    spb->s_free_blocks = 15488;
    spb->s_free_inodes = 768;
    spb->s_nums_blocks = 16384;
    spb->s_magic = 0xE986;
    spb->s_size = 18;  // blocks
    spb->s_user_num = 2;
    spb->s_timeinfo = time(NULL);
    for (int i = 0; i < spb->s_user_num; i++) {
        memcpy(spb->s_user_list[i], userslist[i],
             MAX_USER_LENGTH);
    }
    write_spb_to_block(0, 0, *spb, client, 256);
    free(spb);
    return 0;
}

int init_inodes() {
    // initialize the root inode(0,17)
    inode *root = (inode *)malloc(sizeof(inode));
    root->i_type = 0;
    root->i_link_count = 1;
    root->i_size = 0xffff;
    root->i_block_point[0] = 0;
    root->i_block_point[1] = 17;
    root->i_timeinfo = time(NULL);
    strcpy((char *)root->i_filename, "root");
    root->i_dir_inode[0][0] = 0;
    root->i_dir_inode[0][1] = 17;  // parent is itself
    for (int i = 1; i < 48; i++) {
        root->i_dir_inode[i][0] = 0xff;
        root->i_dir_inode[i][1] = 0xff;
    }  // fill the rest with 0xff
```

```
38      write_inode_to_block(0, 17, *root, client, 256);
39      current_path = *root;
40      free(root);
41      return 0;
42  }
43
44  int init_bitmap() {
45      bitmap_block bitmap[NUM_BITMAPS];
46      for (int i = 0; i < NUM_BITMAPS; i++) {
47          for (int j = 0; j < 8; j++) {
48              for (int k = 0; k < 16; k++) {
49                  bitmap[i].bitmap[j][k] = 0;
50              }
51          }
52          if (i == 0) {
53              bitmap[i].bitmap[0][0] = 0xff;
54              bitmap[i].bitmap[0][1] = 0xff;
55              bitmap[i].bitmap[0][2] = 0xc0;
56          }
57          write_bitmap_to_block(0, i + 1,
58              bitmap[i], client, 256);
59      }
60      return 0;
61  }
62
63  int init_datablocks(int cylinder_id,
64      int sector_id, int par_cy_id,
65                      int par_se_id, char *data) {
66      datablock db;
67      db.d_length = strlen(data);
68      db.d_block_point[0] = cylinder_id;
69      db.d_block_point[1] = sector_id;
70      db.d_parent_point[0] = par_cy_id;
71      db.d_parent_point[1] = par_se_id;
72      strcpy((char *)db.d_data, data);
73      write_db_to_block(cylinder_id, sector_id,
74          db, client, 256);
75      return 0;
76  }
```

2.4 Supported orders

As for the file system client, it can send the following orders to the server:

1. userls: list the users in the system
2. adduser: add a new user to the system
3. f: format the disk-like file system
4. ls: list the files and directories in the current directory

5. cd: change the current directory
6. mk: create a new file
7. mkdir: create a new directory
8. rm: remove a file
9. rmdir: remove a directory
10. stat: show the status of the file or directory
11. shocc: show the occupied blocks in bit-maps
12. r: read the data in a file
13. w: write data in a file
14. i: insert data in a file
15. d: delete data in a file
16. exit: exit the client
17. help: show the help information
18. pwd: show the current directory
19. cat: show the content of a file

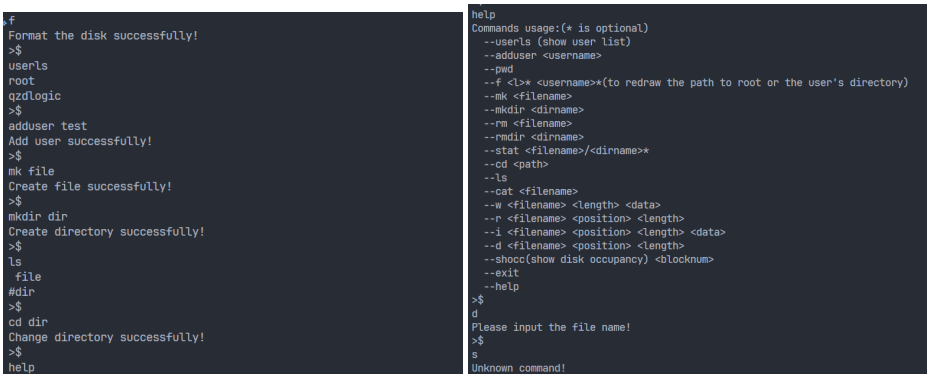

Figure 5: Some order examples

2.5 Data transmission

Each order will be transformed into simple W and R orders to BDS. The Basic Disk Server can only store characters, so the file system server will transform the data into characters and store them in the disk-like file system.

Here the problem occurs to me. For some data are stored by binary form and there will be continuous 0s in the data. When I transform them into characters, the continuous 0s will be seen as the end of the string and would stop the data writing. So I have to store the characters of the hexademical form of the binary

data. In this way, the data can be stored correctly but each 4-bits needs a byte to store, the content of the file will be 2 times larger than the original data. There might be better methods to solve this problem than mine.

3 Multi-users

3.1 Description

In the third step, I implemented a multi-user file system server and client. In the server side, each client can have their own "roots" named the same as the username.This design can be easily achieved by change the root node logic in step2. So I just modify a bit codes in the step 2 , and I have not to create new directory or new files for step3. In the format operation, users can use an extra parameter to specify the username. The server will create a new root directory for the user and set the user as the owner of the root directory. The user can only access the files and directories in their own root directory. The user can do any permitted operation in their own root directory.

Part of changed code ais as follows:

```
int cmd_format(tcp_buffer *write_buf,
    char *args, int len) {
    args = strtok(args, " \n\t\r");
    if (!args) {
        init_disk();
        send_to_buffer(write_buf,
            "Format the disk successfully! \n>$", 34);
        return 0;
    }
    if (strcmp(args, "l") == 0) {
        current_path = read_inode_from_block(0, 17);
        send_to_buffer(write_buf,
            "Format the disk successfully"
            " in root! \n>$",42);
        return 0;
    } else {
        for (int i = 0; i < MAX_USERS; i++) {
            if (strcmp(args, userslist[i]) == 0) {
                init_user(args, i);
                send_to_buffer(write_buf,
                    "Format the disk successfully"
                    " in user's directory! \n>$",
                    54);
                return 0;
            }
        }
        send_to_buffer(write_buf,
            "The user is not exist! \n>$", 27);
        return 0;
    }
    send_to_buffer(write_buf,
```

```
32          "Format the disk successfully! \n>$", 34);
33      return 0;
34  }
```