CSE 312 – OPERATING SYSTEMS MIDTERM REPORT

Main Plan:

The homework is about mimicking a logical file system which needs to use inodes. Therefore I studied that file system in book and come up with filesystem very similar to Linux Ext2 file system. The program seperates disk into abstract blocks those have same size.

After separating these blocks we need to design our reserved space structure and let others blocks to be user data blocks. Our file system will include superblock, inode blocks, inode bit map block, data bit map block and remaning data blocks.

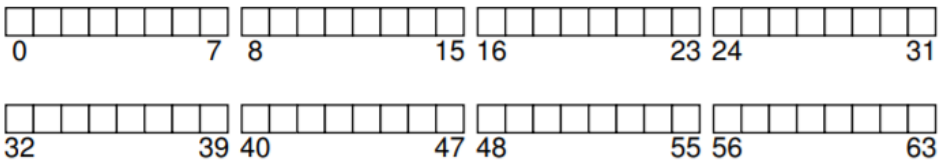

Figure 1: Abstract Disk Blocks

Superblock:

The reserved space generally occupies 10% of our all data blocks. In our homework we take inode number and data block size from user so it can be anything. Therefore we need to fill the disk sequentially with the help of file pointers.

After getting block size from user and having an abstract plan of disk now we need to decide how much our reserved space structures occupies bytes. Let's start with superblock. Superblock contains key informations about file like block size, total number of blocks etc. In my file system it contains structures like :

```
typedef struct superblock_t
{
    unsigned int i_node_count;
    unsigned int data_block_count;
    unsigned int i_node_starting_addr;
    unsigned int log_size_of_block;
    unsigned int total_no_blocks;
    unsigned int i_node_block_count;
    unsigned int i_node_bit_map_starting_addr;
    unsigned int data_bit_map_starting_addr;
    unsigned int data_blocks_starting_addr;
}superblock;
```

Figure 2: Superblock structure in my implementation

Let’s say integers are 4 bytes in our OS, there are 9 fields in my struct so it will be 36 bytes of space(ignoring structure padding). We’ll reserve one block for this special structure. In short 36 bytes will be hold in for example 4 KB structure. It may look like a bit waste of space but for the sake of simplicity we’ll do it like this.

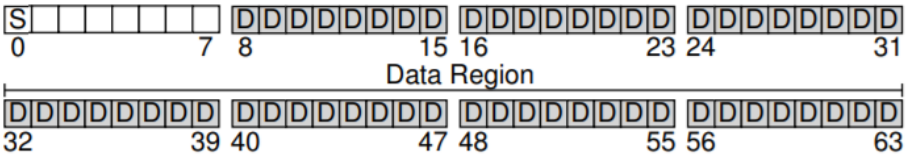

Figure 3:Reserved spaces and data blocks

I Nodes:

After superblock, we need to keep inodes, since there are likely around 200+ inodes we can’t keep all of inodes in seperate blocks. Therefore we need to keep inodes sequentially in blocks called with “inode blocks”. Note that the only fancy thing about it, it’s name. Still it’s an ordinary block.

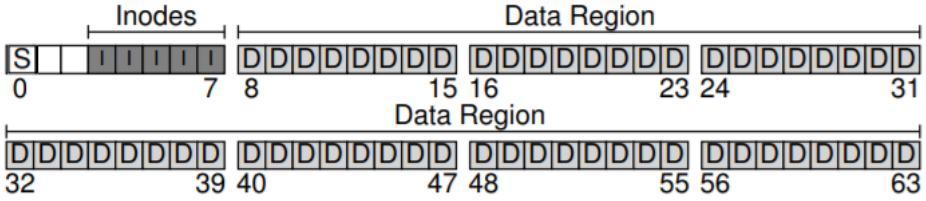

Figure 4:inodes layout in memory.

```
typedef struct inode_t
{
    unsigned int type;
    unsigned int file_size;
    unsigned int day;
    unsigned int month;
    unsigned int year;
    unsigned int hour;
    unsigned int minute;
    unsigned int second;
    unsigned int direct_ptrs[8];
    unsigned int indirect_ptr;
    unsigned int double_indirect_ptr;
    unsigned int triple_indirect_ptr;

}i_node;
```

Figure 4: inode implementation in my implementation

We can think I nodes as metadata about files. In keeps useful information about file(directory also). In my implementation I keeped file type which is regular file or directory, filesize, date, 8 direct pointers that can point up to 8 blocks and three raw integer values that keeps another pointers .

I node and Data Bit Maps:

After keeping inodes then we just have to keep 2 more blocks of reserved data's which are inode and data bitmaps. These are called with bitmaps because it's actually contains raw bit values to ensure which block is empty and which block is not.

After keeping those inode bitmap and data bitmap our structure will be like:

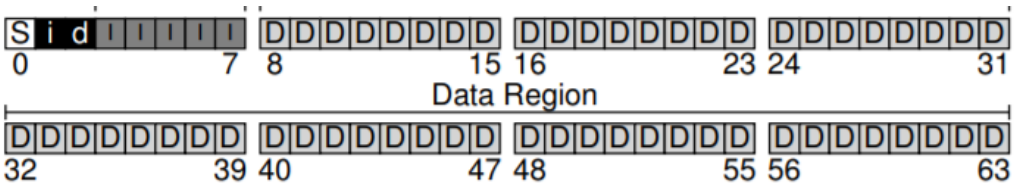

Figure 5: Final reserved space structure

Directories and File Entries:

Directories are same as regular files in my system. They just have special directory structure like this":

```
typedef struct directory_t
{
    file *file;
    unsigned int used;
    unsigned int size;
} directory;
```

Figure 6: Directory structure in my implementation

Each directory holds array of files, which can be another directory also. File structure contains filename.

I will be reaching file name using directories through inodes. Even directories are occupies small space in reality, I will use whole single block for single directory, which can be like 4KB's

File structure looks like:

```
typedef struct file_t
{
    unsigned int i_node;
    unsigned int type;
    char *file_name;
} file;
```

Figure 7:File structure in my implementation

File structure keeps single inode no associated with that file, file type and file name.

Implementation of Operations:

Mounting the Disk:

In part 2 I filled all the reserved structures like superblock inodes etc. While mounting the disk I created one inode structure as root directory and filled it values. I did it so when we start running part3 we won't able to explicitly create root directory and other stuff.

List Function:

In order to implement list we need to traverse into directory that will be listed. In order to do that we follow these steps:

1-)Traverse Into Directory:

- Always start at root directory. We know where is our root inode is stored. In my implementation I store root directory at I node 2 (3rd Inode)
- We need to load that inode into our programs memory. Therefore we need to calculate offset on disk. The calculation will be like: I_node_address _starting_location + 2nd I Node no * BLOCK_SIZE;
- Seek current file pointer to calculated value at number. Fetch the inode.
- That inode we fetched holds an address of data block where it points to.
- Take that address and fetch the block.

2-)List Function

- Now we have current directory in our program which include names and I nodes numbers.

- We can print the file name and type with directory info, in order to print attributes we need to fetch the current inode like I explained in “Traverse Directory”

Mkdir Function:

1-)Traverse Into Directory:

- Always start at root directory. We know where is our root inode is stored. In my implementation I store root directory at I node 2 (3rd Inode)
- We need to load that inode into our programs memory. Therefore we need to calculate offset on disk. The calculation will be like: I_node_address _starting_location + 2nd I Node no * BLOCK_SIZE;
- Seek current file pointer to calculated value at number. Fetch the inode.
- That inode we fetched holds an address of data block where it points to.
- Take that address and fetch the block.

1-)Mkdir:

- Load inode bitmap
- Find random free index on that inode bit map
- Set that index to “not free”
- Write that modified inode bit map to file again
- Load data bitmap
- Find free index for data bit map (randomly)
- Set that random index to “not free”
- Write back that inode bit map to file.
- Traverse current directory that you’ve reached via Traverse Into Directory function.
- Get parent inode number with file named with ..
- Read that parent inode from file
- Create new directory entry
- Fill it with inode no that you have choosen from bit map
- Copy the desired file name and file type
- Add it to current directory
- Write it to disk
- Create new inode for new file
- Create two more file entries named with dot and dotdot
- Add them to current directory and write it at datablocksstartingaddr+ blocksize * free inode bitmap index

Rmdir Function:

1-)Traverse Into Directory:

- Always start at root directory. We know where is our root inode is stored. In my implementation I store root directory at I node 2 (3rd Inode)
- We need to load that inode into our programs memory. Therefore we need to calculate offset on disk. The calculation will be like: I_node_address _starting_location + 2nd I Node no * BLOCK_SIZE;
- Seek current file pointer to calculated value at number. Fetch the inode.
- That inode we fetched holds an address of data block where it points to.
- Take that address and fetch the block.

2-)Rmdir:

- Traverse current directory and get the current directory i node no.
- If that file have specified entry then we can remove otherwise print an error
- Load inode from file and set the directory data location to single inode pointer
- İf that directory have 2 values which are . and .. then we're good to delete otherwise print an error.
- Load inode and data bit map and get random indexes.
- Create new directory that contain those indexes.
- Create new 2 inodes with those indexes.
- Read back the current directory to file

Dumpe2fs Function:

- Read superblock
- Load inode bitmap
- Load data bitmap
- Print inode bitmap
- Print data bitmap
- Print all superlock fields
- Print all free, not free inode and bit maps

Write Function:

1-)Traverse Into Directory:

- Always start at root directory. We know where is our root inode is stored. In my implementation I store root directory at I node 2 (3rd Inode)
- We need to load that inode into our programs memory. Therefore we need to calculate offset on disk. The calculation will be like: I_node_address _starting_location + 2nd I Node no * BLOCK_SIZE;
- Seek current file pointer to calculated value at number. Fetch the inode.
- That inode we fetched holds an address of data block where it points to.
- Take that address and fetch the block.

2-)Write:

- Load inode bitmap
- Get block size
- Traverse the current directory find the inode no with name .
- Read that inode
- Find free inode bit map
- Set it and write it back to disk
- Create a new file
- Set it attributes
- Add that file to directory
- Load data bit map
- Create random bit for first index
- Read from another file block by block
- Fill the first 8 pointers
- If file still not ended create new block that points to another block of pointers.
- Fill their addresses and continue like that.

Read Function:

1-)Traverse Into Directory:

- Always start at root directory. We know where is our root inode is stored. In my implementation I store root directory at I node 2 (3rd Inode)
- We need to load that inode into our programs memory. Therefore we need to calculate offset on disk. The calculation will be like: I_node_address _starting_location + 2nd I Node no * BLOCK_SIZE;
- Seek current file pointer to calculated value at number. Fetch the inode.
- That inode we fetched holds an address of data block where it points to.
- Take that address and fetch the block.

2-)Read:

- Load inode bitmap
- Get block size
- Traverse the current directory find the inode no with name .
- Read that inode
- Find free inode bit map
- Set it and write it back to disk
- Create a new file
- Set it attributes
- Add that file to directory
- Load data bit map
- Create random bit for first index
- Read from another file block by block
- Read the first 8 pointers
- If file still not ended read new block that points to another block of pointers.

read their addresses and continue like that.

Sources:

http://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf

understanding the linux kernel book