Design Document
Vanilla File System

CSCE 611
Operating Systems
MP7 - Fall 2018
By
Himanshu Gupta

Design for Vanilla File System.
Bonus options: **Option 1- Design of thread safe file system**

In this MP we implement a file system. The design is divided in two different layers. The first is the file system layer and other as the file class. The file system provides the basic design and utility functions to be used by file.
We first describe the design of the file system. We first create a inode structure which are used to manage the files on this disk. I call them as m_node in the code. This node will hold the information regarding the particular file it points to. As we name file by its id number, we have this in the inode structure. It also maintains array of all the disk blocks that this file is going to use and the size of the current file. This will be used to determine the EoF location while reading the file. Finally we have the inode struct as following:

```
typedef struct node {
   unsigned long fd;
   unsigned long block[BLOCK_LIMIT];
   unsigned long size;
   unsigned long b_size;
}m_node;
```

The class for file system uses several private variables. First one is the pointer to a simple disk object. We do not include the blocking disk from the previous implementation as the simple disk is good enough for this file system. Then we also maintain bitmap for all the free blocks, therefore we can store the free block info for 8 blocks in each index. We also have the number of total blocks, management blocks and the number of inodes used.

The inode as used to store the file information and written on the disk. Whenever we need to update the information for any file, we search through the inodes to locate the correct inode by comparing the file id in the node. Then we update this information and write back the updated inode to the disk.

For the File system api, the Mount api will update the disk pointer to the private member. The constructor will just initialize the file system variables. Format api will first update the disk and the size of this file system that will be installed in the disk. Then it updates the local management information to be used for file management. It sets the bit map to be used for free file allocation where we set the reserved number of blocks to be used for inodes. We mark the block map as 1 for each allocated map in the bitmap to be used for inodes. Then we set all the blocks in the disk as 0 and write that iteratively on all blocks.

The lookup api is used to initialize the file object. We do not create the file object when the create file is called. The create file api first searches the already existing file id, if it doesn't exist then it creates an inode for that file and writes it to the disk. This way we can retrieve the file information later when we look up via the lookup api. It will return a new file pointer with all its information initialized. This includes the size and the block information of the file. We define in the design of this file system that a particular file can only hold 16 blocks of information. This can be changed depending on the requirement of the application we have. Therefore, for now we keep the size

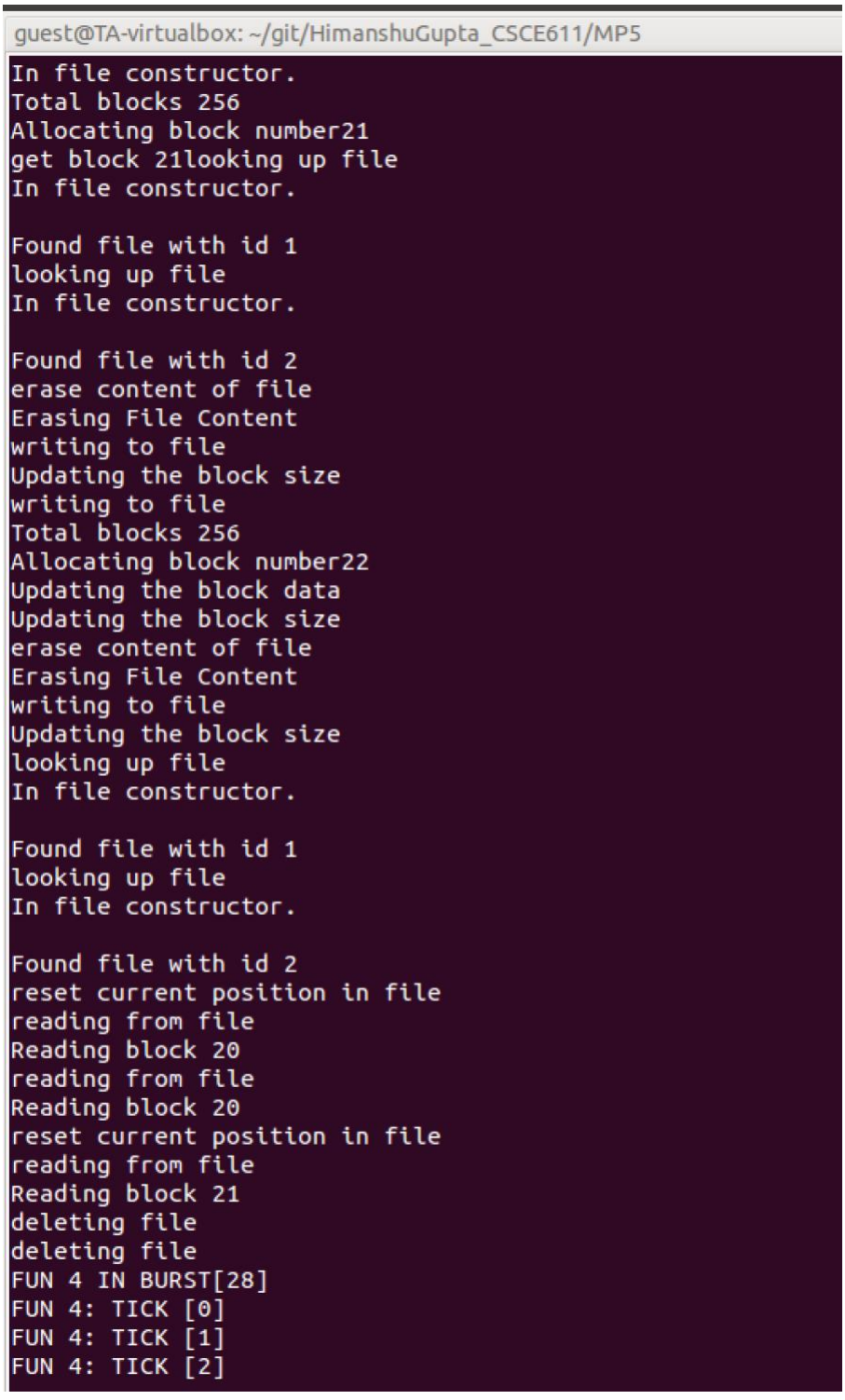
of the file as 16 blocks. Next, we implement the delete file api. This will free all the allocated api to this file and clear the inode data.
I also implemented some utility api's to be used in the file and file system class. Basically they allow me to allocate, delete blocks for use. Update size and data of the blocks and erase its content.
The file class declares FileSystem as a friend class which enables us to use the private members of the File System class. We also cache the blocks used in this file in the file class so that we don't have to read the inode again. They are only read if we need to add more blocks to the file. This way we save the read from the disk. Next, we use an index and a position variable to do read write operations. The index is used to indicate the block number and the position is used to indicated the byte index in a particular block. This way we can iterate over each byte if the 16 blocks used in the file system .

The File constructor is used to set the private variables in the file. They are initialized when the file lookup api is called from the application. In the read api, we create a 512 bytes buffer, and issue a read operation on the disk from the current position of the file. If the position pointer exceeds the 512 mark, we move to next block and reset the position to 0. We also check the EoF condition for each read. The write api is implemented similarly but need to update the block data and size to the inode when write is done. The reset api simply puts the pointer of the file to start.

The rewrite api is use to erase the content of the file. In this API we erase the content of all the blocks and free them but keep the first block, i.e. the first block is erased but not freed. This way we keep the first allocated block to file and the inode information is updated with the same data. Finally the EoF function wil determine if the current position of the file has not crossed the size of the file. This is a simple check of index and position with the size variable of the class.

TESTING

I tested the regular exercise_file_system call in the kernel file and also added a check to write a very long stream of data that would exceed 512 bytes. Therefore, it will exceed the 1 block mark for the file write and read. This test was successful and I was able to execute the read write calls. This demonstrated the correctness if this implementation. If you need any clarification please contact me. I am also attaching a screenshot of the successful test.



Design of thread safe file system.

To implement a thread safe file system, we need to allow simultaneous read and write operations. This will change the interface for file system. The file system class will have a lock

implemented on the disk resource. This will be done in a way that only one thread can perform the actual read or write operation. If we have multiprocessor system, then we need to make sure that the lock implemented does not prevents starving to other threads, i.e. if the thread dies it does not keeps the lock itself and is released upon termination. We will also need to implement this in a critical section so that the access is controlled. When multiple read write operations are requested, we can also implement a journal or a cache so that we can speed up the write process. This will allow us to perform the write in a very fast manner. The file system can also be improved by adding media awareness, this will be very tricky to implement but will significantly improve the multi-threaded performance.