CSCE 611 Operating Systems Summer 2017

825008857 Chaiwei Chang

MP7

Design for file and file system

We follow the implementation of a file system including file and file system provide directory service. And how file were placed on the disk is that I will record the block index number for that specific file to know where to read from by using SYSTEM_DISK read along with the current location pointer. So let’s first start with the file. A file have the information of which filesystem it’s belonged to. And which physical block it use to record the inode information. Also, the info of file itself including file_id, file_size, file location in terms of block, cur_block and cur_position to determine where to start writing and reading. The files are maintained by linked list of blocks. Each block is 512 bytes and with 4 bytes pointer information. So the actual size of each block is 508 bytes. In File::File(): Every file has a unique id, the constructor will first check if the file is already exist. If not, the file system will call CreateFile() to create a new file. Here I will mention how we do createfile in file system. I use File object link list to list to the newly created file. Read and write is simple since we have all the block info and size info, all we need to do is to check the edges case where file doesn’t has that much stuff in it. In other words, file has reached the EOF(), if the current position is same to the pre-defined size, will return fail to read or write.

```
FileSystem*file_system;
```

```
unsigned int   inode_block_num
unsigned int   file_id;
unsigned int   file_size; // in blocks
unsigned int*  block_nums;//pointers of designated file blocks
unsigned int   cur_block; //current block position, always starts from 0
unsigned int   cur_position; //current position in currrent block
```

Reset is even easier, I will just set the current position and current block to the beginning so that the next write will overwrite the garbage data. Additionally, for rewrite I will link the block to the free block list, which file_system class maintains. At last, the file_system will update the file data and store the size.

For file_system:

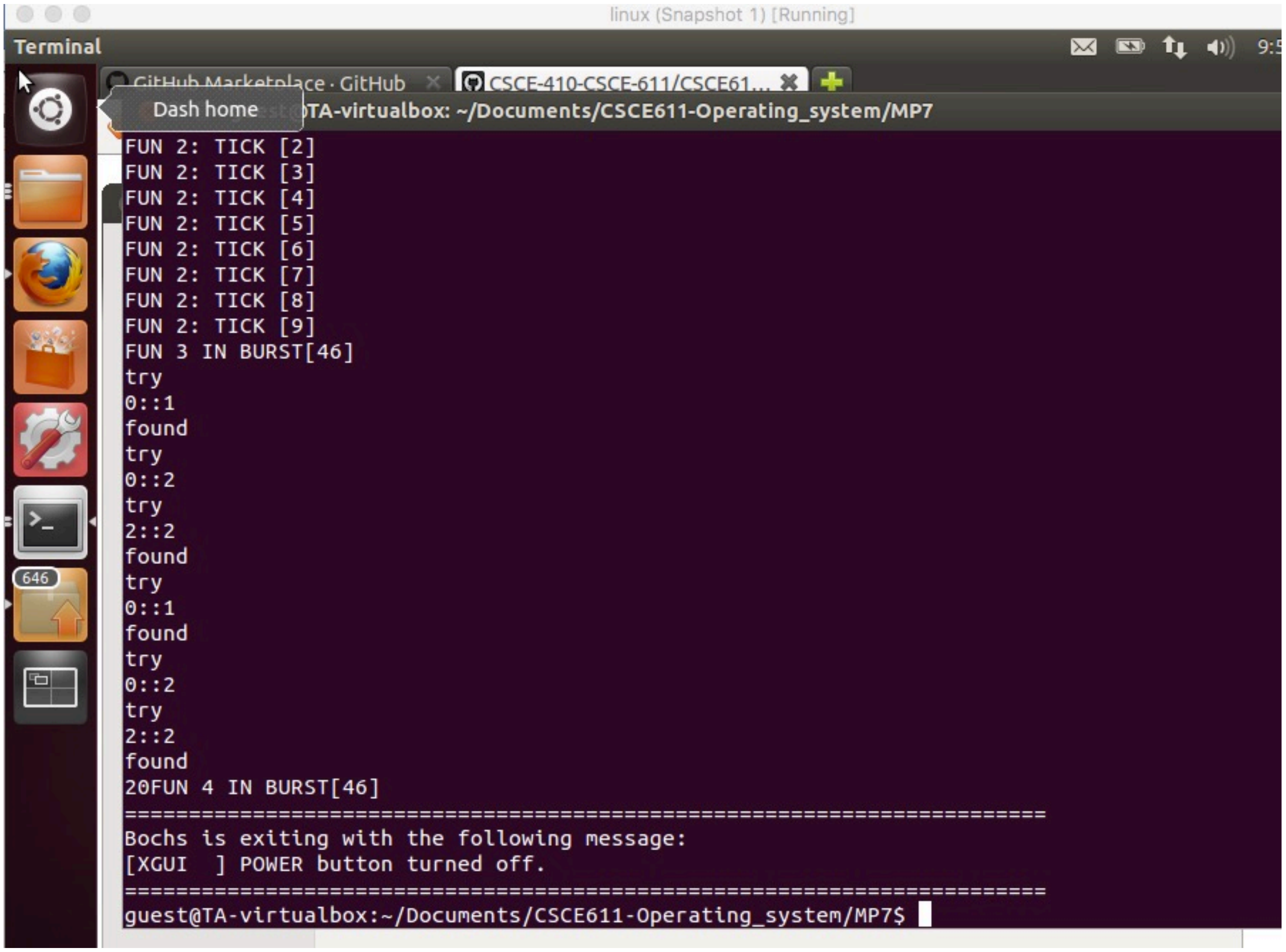
```
private:
            /* -- DEFINE YOUR FILE SYSTEM DATA STRUCTURES HERE. */

           SimpleDisk * disk;
            unsigned int size; //size of disk
            File*  files;
            unsigned int num_files;//number of files
            unsigned int block_num;//current block to look for free blocks
```

The system has the most important list which is the file that allocated inside this system in the linked list format. The file system maintains the first block to be reserved for these information. Format(): function will go through each block and link all blocks. Mount() function will first point to the mounted disk and then copy all the information from block 0 into this new block. LooupFile() function will get the file from the list, which has the same file id. And initialize the file object and return TRUE. Otherwise return FALSE. DeleteFile() function will first find the file being deleted by loopup the given file id. And then move the block to the free block list. Then refresh the file found with the give file id and set to 0.

For the bonus option 1, I tried to design a thread safe system. However due the upcoming final exam, I don't have time to implement it. For the File System access point of view, we may encounter race condition when create and delete is not an atomic action. So we need to add the scheduler to do the synchronizing job. A thread to execute a task in such a thread safe manner that other threads have apparently no side-effect over the state variables of that task when it was being executed by this thread. In other words, for this create method the state is existence or non-existence of the file, when the thread was about to execute this method. Let's say that when this method was being executed by the thread the file did not exist. So according to the concept of Atomicity no other thread should be able to create the same file(which was not existing before) during these execution time otherwise the very first assumption of this thread about the state of the file will change and hence the output. So in this case the executing thread does-this by obtaining a write lock over the directory where file is to be created. Inside the scheduler, we will do the directory lock or maybe a commit log for each thread to prevent interrupt and do recovery. Delete do the exact same thing.

For the file access perspective, I will try to implement some more sophisticated read / write lock to protect the critical section inside a file.C. A thread that wants write access to the resource can be granted so when no threads are reading nor writing to the resource. It doesn't matter how many threads have requested write access or in what sequence, unless you want to guarantee fairness between threads requesting write access. I will not take reentrance into condition, so this will only prevent race condition but not efficient enough. Therefore, all the mechanisms that I have mentioned is my design for thread save system.

The following figure is my final result for this MP. Finally, I wrote some command on my code, but if grader of TA have any question about my logic feel free to contact me at hunkywei@tamu.edu


linux (Snapshot 1) [Running]
Terminal
GitHub Marketplace · GitHub
CSCE-410-CSCE-611/CSCE61...
Dash home
TA-virtualbox: ~/Documents/CSCE611-Operating_system/MP7
646

```
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
FUN 3 IN BURST[46]
try
0::1
found
try
0::2
try
2::2
found
try
0::1
found
try
0::2
try
2::2
found
20FUN 4 IN BURST[46]
========================================================================
Bochs is exiting with the following message:
[XGUI  ] POWER button turned off.
========================================================================
guest@TA-virtualbox:~/Documents/CSCE611-Operating_system/MP7$
```