CSCE 611

Machine Problem 7: Vanilla File System

Submitted By – Vaibhav Rawat
UIN: 626008171

I completed this assignment and attended the bonus options of 1 and 2 for thread-safe file system. There is a separate folder for options code.

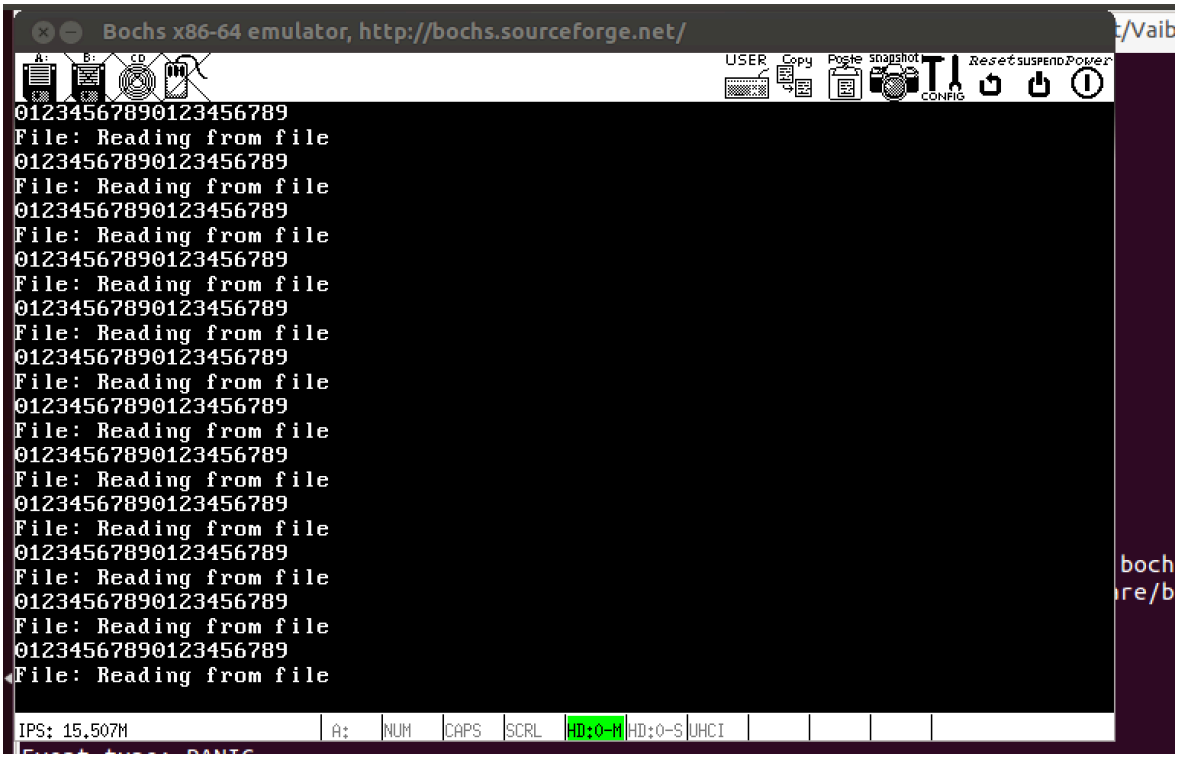
Main task in this assignment is to work on top of existing source code provided in file.C and file_system.C to build a file system. To begin with, I will direct attention to implementation of file class. For this MP, since files support sequential access only, we don't need a mechanism to support random access. For the Files class, I have a file id member variable which represents the file name (multi-level directories are not supported). All Inodes are stored in initial blocks assigned only for inodes, and data blocks follow them. File class also has a reference to store the position of inode in disk, that is the block number to read the file's inode. Inode maintains information about the file size, data blocks on the file points to and the file id. So, using disk read corresponding inode can be fetched for a file id and later used to fetch the corresponding data blocks for the file. File also include other member variables such as blocks list on which file is stored, current block index, current position in the current block and file size. So initially when file constructor is called it creates a new file object with default values for member variables and associated with a file id. When Read requests are made for that file, it loops through all the blocks in the blocks list starting from current block and current position of file object. It verifies for every block in the list if end of file is reached - by checking if it reached the last position for file in the last block. If not it proceeds to read from the blocks till requested characters are read. Otherwise if end of file condition is reached, it breaks the loop and does not read beyond end of file.
When write requests are made for the file, data is written to blocks starting from current block and position, in a loop. In loop it first checks if all blocks in file are

full and current position is has already traversed the last block completely. If all blocks are full it raises a request to file system to allocate a new block, updates the maintained blocks list and increases the block count. Corresponding inode on the disk is updated to reflect the same. It then copies all the information from buffer to the current block starting from current position. Till all n characters are written from the buffer, same steps are repeated in a loop.
Implementation of Reset is easy and both current block index and current position are reset to zero. Rewrite function of file goes through all the data blocks of file and make calls to file system to free the data blocks. It then resets the member variables – current block index, current position and blocks count to 0 and empties list of blocks. It updates inode on disk to reflect the same.

Second part of the design focus is on file system class. File system has to deal with disk read and write calls. A disk is registered to the file system using Mount function with the provided disk object, which also updates file system with existing files on disk. Format function is a static function which takes a disk object as parameter, and using a disk buffer writes 0 to all the blocks of the disk. It also sets the first block to used and writes 0 for specifying the number of files on disk. Lookup File function takes in file id as parameter and check for matching file id. If such a file id exists, it reads the inode of the file and traverses through the data blocks to populate the file object with relevant data blocks number for the file. If a file does not exists, it simply return NULL. CreateFile function first checks if a file id is present with file system, if it exists it simply returns false. Otherwise for the file id provided to it as parameter, it first allocates a block by checking disk for a free block. Then it uses the free block for maintaining inode information for the file. It sets this block's file id and changes its status to used. Then it writes back this inode info block to disk. It also adds the newly created file to the list of files maintained by file system.
Remove file function traverses through the list of files to find a matching file object with file_id same as the one to be removed. If such an object is found it clears the contents of the file from disk by making a call to Rewrite function of file and updates corresponding file inode. Then finally it deallocates disk memory assigned for even the inode to delete the file completely from file system. It also updates the files list maintained by file system to remove the file id from the list.
I initially tested my implementation with 20 bytes read/write. Following are snapshot of the read/write results for two strings "0123…" and "abcd…":

```
File: Writing to file
FileSystem: Assign Block From Disk
File: Rewrite/erase content of file
File: Writing to file
FileSystem: Assign Block From Disk
exer_file_sys : Write done
FileSystem: Looking up file
file found
FileSystem: Looking up file
file found
File: Reset current position in file
File: Reading from file
01234567890123456789
File: Reset current position in file
File: Reading from file
abcdefghijabcdefghij
```

I have also tested my implementation for 2000 bytes read/write operations spanning through multiple blocks. Following snaphot describes the same-
Write of string “01234567890123456789” is performed 100 times
Read operation – 20 bytes at a time is performed 100 times. I have commented out these test changes for 100 iterations in kernel.C included with my submission.



```
File: Writing to file
File: Writing to file
FileSystem: Assign Block From Disk
File: Writing to file
File: Writing to file
File: Reading from file
01234567890123456789
File: Reading from file
01234567890123456789
File: Reset current position in file
File: Reading from file
abcdefghijabcdefghij
File: Reading from file
abcdefghijabcdefghij
File: Reading from file
```

For option 1 and 2, I implemented a thread safe design for file system from perspective of file system access and file access –

a) File system has operations of create file and delete file, and format which may suffer from race conditions in multi-threaded scenarios. To handle the race conditions, I have enabled locking of create file, delete file, format operations so that they can be performed atomically. Moreover, I tried my implementation with blocking disk implementation which involves blocking disk with a scheduler queue to provide additional synchronization. Mutual exclusion is enabled by a simple implementation of Lock class in file system having member functions - lock and unlock. So, while creating a file this mechanism is used to lock, then disk is accessed to create a new file inode and write it on the disk. With locks enabled, all other threads wait till the current thread is done with creation of file. Same lock is used for deletion of files and mount, so that scenarios such as creation and deletion of same file should be mutually exclusive.

b) Locking is not only limited to file system. File has functions of read, write and rewrite which may suffer from race conditions in multi-threaded scenarios. To handle them, I have enabled locking of write, read and rewrite operations. So, for a read operation lock is done before accessing current block from disk. This ensure that no other operations such as write, rewrite etc access the disk and modify/remove the contents while read is still active. Same lock is used for write, rewrite, create delete file operations to ensure mutual exclusion.

I tested my implementation by performing read/write operations by calling exercise file system in kernel.C from multiple threads(function2 and function 3).

// Snapshots of function2 in order – it becomes active, writes to the two files, reads from two files

```
FUN 2 IN BURST[0]
FileSystem: Creating file
FileSystem: Assign Block From Disk

File: Writing to file
File: Writing to file

File: Reset current position in file
File: Reading from file
01234567890123456789
File: Reading from file
01234567890123456789

File: Reading from file
abcdefghijabcdefghij
File: Reading from file
abcdefghijabcdefghij
```

// Snapshots of function 3 in order.

```
FUN 3 IN BURST[0]
FileSystem: Creating file
FileSystem: Assign Block From Disk

File: Writing to file
File: Writing to file
File: Writing to file

File: Reset current position in file
File: Reading from file
01234567890123456789
File: Reading from file
01234567890123456789
```

```
File: Reading from file
abcdefghijabcdefghij
File: Reading from file
abcdefghijabcdefghij
```

I have included output files for all the above tests in respective mp7 folders.