Document

Zhiren Zhou

December 2022

1 Options Implemented

Beyond the a simple file system that supports the sequential access only, I implemented the **OPTION1:DESIGN of an extension to the basic file system to allow for files that are up to 64kB long**.

2 Design Concept

1. Modified the **Inode** class:

   Add the private data member **block_no** and **size** and public function **WriteToDisk()**, **ReadFromDisk()**.

2. Implemented the **Inode::WriteToDisk()** :

   This function is used to write the inode list, which is stored in the file system object, into the disk. Even though the **FileSystem** class also has the similar function, this function is mainly implemented for **File** class to use. This is because the **File** class is not the friend class of **FileSystem** class. It has no right to visit the inode list in **FileSystem** object, so it can not use the API in **FileSystem** class.

3. Implemented the **Inode::ReadFromDisk()** :

   This function is used to read the inode list to the memory, which is stored in the file system object, from the disk. Even though the **FileSystem** class also has the similar function, this function is mainly implemented for **File** class to use. This is because the **File** class is not the friend class of **FileSystem** class. It has no right to visit the inode list in **FileSystem** object, so it can not use the API in **FileSystem** class.

4. Modified the **FileSystem** class:

   Add the pubclic API **WriteBlockToDisk()** and **ReadBlockFromDisk()**. Uncomment the two funcion **GetFreeBlock()** and **GetFreeInode()**.

5. Implemented the **FileSystem::FileSystem()** function:

   This function is the constructor of **FileSystem** class. It mainly allocates memory for inode list and free block list.

6. Implemented the **FileSystem::∼FileSystem()**function:

   This function is the destructor function of **FileSystem** class. It mainly writes the inode list and free block list to the disk and deallocates memory for inode list and free block list.

7. Implemented the **FileSystem::GetFreeBlock()** function:

   This function is used to search the free block list and look for a free block. If it successfully finds the free block, it returns the block number.

8. Implemented the **FileSystem::GetFreeInode()** function:

   This function is used to search the inode list and look for an unused inode. If it successfully finds the unused inode, it returns the index of that inode in inode list.

9. Implemented the **FileSystem::Mount()** function:

   This function mounts the disk to the computer, which means it reads the information of inode list and free list, which stores in the first the second block in the disk, and stores them into the memory. It also check whether the first two blocks in disk are used to store the information.

10. Implemented the **FileSystem::Format()** function:

    This function is used to format the disk. It first creates an inode list,in which none inode is used, and a free block list, in which only the first two blocks are used to store the inode list and free block list. Then it writes theses information back to the first and second block of the given disk.

11. Implemented the **FileSystem::LookupFile()** function:

    This function is used to look for a file in inode list given the file id. If it successfully finds the existing file, it will return the pointer to that inode. If not, it will return the NULL pointer.

12. Implemented the **FileSystem::CreateFile()** function:

    This function first looks up the file with given file id, if the file already exists, it will return false. Then it tries to get a free block, if there is no free block, it return false. Then it tries to get a unused inode, if all the inode are used, it again return false. When everything is prepared, it

marks the free block as used and fill the inode with relative information. Finally, it writes the inode list and free block list back to the disk.

13. Implemented the **FileSystem::DeleteFile()** function:

    This function first looks up the file with given file id, if the file does not exists, it will return false. Then it finds the block number of the given file id through its inode. It sets the block as free in free block list. It also erases the information stored in the inode list. Finally, it writes the inode list and free block list back to the disk.

14. Implemented the **FileSystem::WriteBlockToDisk()** function:

    This function mainly writes the data,which stores in the given buffer, to the given block of disk.

15. Implemented the **FileSystem::ReadBlockFromDisk()** function:

    This function mainly reads the data,which stores in the given block number in disk, to the given buffer.

16. Modified the **File** class:

    Add the private date member **fs**,**inode** and **current_pos**.

17. Implemented the **File::File()** function:

    This function is the constructor the **File** class. It sets the filesystem object and inode and current position of the new file object. Once belonging to one filesystem, this file knows which disk it belongs to. Once this file object knows its inode, it knows which block in that disk should it read/write from/to.

18. Implemented the **File::~File()** function:

    This function is the destructor the **File** class. It writes the Cache of file object back to the disk through the API of **FileSystem** object. It also updates the inode list in the disk with the help of inode object.

19. Implemented the **File::Read()** function:

    This function reads the data from its Cache to the given buffer and returns the number of characters it reads. If it reads the given number of characters or reaches the end of the file, which is limited by the file size stored in inode, It will stop.

20. Implemented the **File::Write()** function:

    This function writes the data given buffer to its Cache and returns the

number of characters it writes. If it writes the given number of characters, it will return. It the file size extends the original file size but less than one block, It will updates the file size in inode. If the file size is going to greater than one block size, it will stop and limit the file size no more than one block size.

21. Implemented the **File::Reset()** function:

    This function resets the position pointer of current file to the beginning of the file.

22. Implemented the **File::EoF()** function:

    This function checks whether the position pointer reaches the end of the file, which equals the size of file.

23. **OPTION1:DESIGN of an extension to the basic file system to allow for files that are up to 64kB long.**

    (1)Modified **Inode** class: The currenct file size is at most 1 block size, so we do not need to consider about the external fragmentation. If we want a file that is up to 64KB long. We need to eliminate this problem by modifying the inode structure. In practice, we can use **an index structure with direct, indirect, and doubly indirect blocks**. The **size** should also be modified to record the **size_of_final_block**.

    (2)Add **Inode::RequireNewBlock()** function: This function is mainly called by the file object. When the file wants to extend its size by one block, it has to call this function to require for a new block. This function will call the **ExtendFileSize()** function in filesystem object and the filesystem will look for a free block and return its block number. This function can add that new block to the inode and update the inode list in the disk.

    (3)Add **Inode::GetNextBlock()** function: This function is mainly called by the file object. The file will only caches one block in its block_cache, so when the file wants to go from one block to next block, it will call this function to get the next block number and cache the data in that block again. This function will return the next block number which is stored in inode structure. If the current block is the final block of that file, it will return -1.

    (4)**DO NOT** modified **FileSystem::CreateFile()** function: This function still creates a file with 0 size and allocates one inode for it. It allocates one free block for this file when creating it.

    (5)Add **FileSystem::ExtendFileSize()** function: When a file needs to extend its size, e.g when the system writes data in to that file and it size

extends one block after writting, the file object will call **RequireNewBlock()** in its inode object, and that inode object will call this function. This function will look for a free block and return the block number.

(6)Modified **FileSystem::DeleteFile()** function: This function needs to inactivate every used blocks for the given file id.

(7)**DO NOT** modified **File::block_cache** function: Since the file size could be up to 64kB long, if we cache the whole file in the file object, it is possible to use up the memory quickly. So the we **still cache one block size file at one time**.

(8)Add **File::current_block**: The file now could be composed of more than one block now, and the block_cache can only cache the data of one certain block. So we need this private data member to record which block data is cached now.

(9)Add **File::Recache()** function: This function will first call the **Inode::GetNextBlock()** function to get the next block number of this file. Then it reads the data in next block to the **block_cache** and updates the value of **current_block**

(9)Modified **File::Read()**: This function is used to read data from the cache. If the current position reaches the end of file, it will stop automatically and return the number of characters it reads. If it needs to jump to next block, it will call **File::Recache()** and keep reading.

(10)Modified **File::Write()**: This function is used to write data to the cache. If the file needs to write to the next existing block, it will call **FileSystem::WriteBlockToDisk()** to write the current block to the disk and then call **File::Recache()** to get the data of next block and keep writing. If the current position reaches the end of file, it will extends the file size. If it does not need a new block, it just extends the **size_of_final_block**. If the file needs to increase one block and its size is less than 64kB after increasing one block size, it will call **Inode::RequireNewBlock()** to get one new block. Then it will call **FileSystem::WriteBlockToDisk()** to write the current cache to the disk by call. Finally, it will update the **current_block** to the return value of **Inode::RequireNewBlock()** and set the **current_possition** to 0 and keep writing. If the file size is going to greater than 64kB, this function will stop.

(11)Modified **File::Reset()**: This function needs to finds the first block of current file and update the **current_block** to that value and set the **current_possition** to 0.

(12)Modified **File::Eof()**: This function needs to check whether**Inode::GetNextBlock()** equals -1 and **size_of_final_block** equals **current_block**.

3 Files Modified

1. **file_system.H**
2. **file_system.C**
3. **file.H**
4. **file.C**

4 Testing Results

1. Testing the simple file system that supports the sequential access