Design notes

Mian Qin UIN:725006574

1 Overview

1.1 File System Layout

In this machine problem, our goal is to design and implement a simple file system. I choose to design and implement a **simplified Unix file system**.

Superblock block 0	File table block 1	Inode bitmap block 2	Inode table block 3-4	Data block bitmap Many blks	Data block Many blks

Here is the overview of this file system The very first block is used as superblock, in my design, it only contains a 4byte size of the whole filesystem (1MB in test case). Second block (block 1) is a file table block, since in our case we use a flat namespace, so I didn't implement the hierarchy directory in the standard unix file system. In the file table, it maintains a table of the filename (4 bytes) and inode num (4 bytes). Since the block size is 512 bytes, each file table entry is 8 bytes, my file system can support at most 64 files. Since the **maximum number of files is fixed as 64**, the inode number is also fixed as 64. Here is the definition of the inode:

```
class Inode {
public:
    uint32_t size;
    uint16_t direct_index[5];
    uint16_t indirect_index;
};
```

As you can see, the Inode has a 4 bytes size indicating the file size, 5 2bytes direct index and 1 2byte indirect index for the data block. Here I assume the **block address is 16 bits.** The maximum file size support is 5*512 + 512/2*512 = 130KB. Back to the filesystem layout. The inode bitmap only takes 1 block (block 2) (Note: here I use 1 byte rather than 1 bit to keep track of the inode usage since we have plenty of space.) For the inode table, since each inode takes 16 bytes, we need 2 blocks to hold 64 inodes.

The next part is the data block bitmap, this is a real bitmap and it keeps record of all the blocks used in the filesystem which means it starts from block 0 and marks all the metadata block as invalid as format. The data block bitmap block number is calculated by the total filesystem size when we do format.

1.2 Metadata Management

Another issue need to be address for my simplified Unix file system is the metadata management. This include the file system matadata (file table, inode bitmap, data block bitmap) and file metadata basically inode.

The key part of the file system is persistency, which means all the metadata must be persistent in the disk. In my implementation, I made in-memory caches for all the metadata and do block flush when the in-memory metadata cache got updated. That's the fundamental approach to keep the file system persistent. In case of the file system metadata, the inode

bitmap and file table are kept use static memory allocation basically stay in the stack. However, the data block bitmap is variable length since it depends on how many blocks we totally have, so, it it's dynamic allocated during **Mount** time. The file metadata inode is kept statically in the File object.

For the detailed flush operations see the next section for detailed implementation for each part.

2 Implementation

2.1 FileSystem

The FileSystem class maintains the metadata of the file system (file table, inode/data block bitmap) and also implements private member functions for inode/data block allocation/deallocation for the File class.

2.1.1 FileSystem:: FileSystem ()

In FileSystem constructor, it will initialize the corresponded disk, bitmap pointer and file table current pointer, etc.

2.1.2 FileSystem:: Mount ()

In the Mount function, there are basically 4 steps to. 1, it will read the super block and get the size of the file system. 2, it will read the file table in the second block and also determine the file table current pointer. **Here noted that I assume that the file name is non-zero** 3, read the inode bitmap from the third block. 4, read the data block bitmap from the following few blocks, the actual size of data block bitmap is determined by the file system size read from the super block in step1.

After all 4 steps, the file system is successfully mounted into memory and the file operations can be performed normally.

2.1.3 FileSystem:: Format ()

For Format function, it also needs 5 steps, which is related to the Mount() operation. 1, write the super block about the file system size. 2, wipe out the file table block, basically write all zeros into it. 3, Initialize the inode bitmap as all valid. 4, Also need to wipe out all the inodes in the inode table (2 blocks writing all zeros) 5, initialize the data block bitmap according to the filesystem size and marks all the metadata block as invalid.

2.1.4 FileSystem:: LookupFile ()

For LookupFile, it will first go through the in-memory file table and if the filename is found, it will create a File object. Then, it will read the inode from the inode_id in the file table and initialize to the File object. Finally it will return the File pointer, otherwise, if the filename is not in the file table return a NULL pointer.

2.1.5 FileSystem:: CreateFile ()

For CreateFile, it will first look through the file table and see if the file is already existed. Then, it will allocate an inode using the api defined by myself (show later in detail). If the no available inode, it will also return false.

If inode return successfully, it will update in the file table and also flush the file table to disk and return true.

2.1.6 FileSystem:: DeleteFile ()

When deleting a file, it first performs the LookupFile to get the file object pointer. Then, it will

call Rewrite() file function to release all the data blocks in the file and also release the corresponded inode using my self-defined api release_inode() and then update the file table by swapping the last entry (wipe out the deleted entry). Finally, it will flush the file table to disk.

Follwing are the private member functions that defined by myself to help implement the above operations and also the File operations.

2.1.7 FileSystem:: alloc_inode ()

Alloc_inode() basically allocate inode to the file system during File creation. Important node is that it needs to flush the inode bitmap when allocate a inode successfully.

2.1.8 FileSystem:: release_inode ()

Release_inode() will reclaim an inode and update the inode bitmap and also wipeout the corresponded inode. So, it need to flush two blocks to disk.

2.1.9 FileSystem:: alloc_data_block ()

For alloc_data_block() it works for allocating data block, this is used for the File write operations. This also need to update the data block bitmap in the file system which means it also need to flush to disk.

2.1.10 FileSystem:: release_data_block ()

For release_data_block() it will look through the data block bit map to see whether the release block is actually allocated. If so, it will update the data block bitmap and also flush the data block bitmap to disk.

2.1.11 FileSystem:: flush_inode ()

The flush_inode() is used by the File write() operation, since every time we write the file we may change the file size (if the write beyond the original file size). This will lead to the inode change (size information and the block index information). The flush_inode() will help the File write operation flush the inode to disk.

2.2 File

For the File class, it does the actual file operations like read, write, reset(rewind), rewrite. Every File object will associate with the Filesystem object during construction to do the actual inode/data block allocation as well as disk operation.

As you can see in the section 1, the inode definition. There are 5 direct index block and 1 indirect index block for each file. The maximum size of a file is around 130KB. There is a in-memory copy of the inode in each File object.

2.2.1 FileSystem:: File ()

In the File constructor it will set the current pointer of the file as 0 and take a FileSystem class pointer as an input to associate with FileSystem object.

2.2.2 FileSystem:: Read ()

For all the read/write operations of File, it actually start from the current pointer of the file. For Read(), it will first determine the start block and block offset to read, and then conduct the disk read for that block. Then it read from the memory buffer of that block and advance the current pointer of the file until EoF or finish read all the bytes.

Noted, the copy of disk buffer and user space buffer (input to Read) is byte by byte, and the block read is also done block by block.

2.2.3 FileSystem:: Write ()

For Write() operation, it's actually more complex. First, the first byte to write is always start from If the current pointer is before the actual file size, it will overwrite the data blocks for the file, if the current pointer run beyond the actual file size, it will start append which means we need to allocate new data block to the file and also need to update the inode (both size and data block index).

The work flow is similar to Read() function, it will first use the direct block index, if all 5 direct index is used, it will start use the indirect index, this will need to allocate an extra block for the indirect index and also need to update that block every time a new actual data block is allocated.

2.2.4 FileSystem:: Reset ()

For reset() operation, it will reset the current pointer of file as 0.

2.2.5 FileSystem:: Rwrite ()

For Rewrite() operation, it will traverse all the direct index and indirect index if existed and release all the data blocks for the file.

2.2.6 FileSystem:: EoF ()

For the EoF() operation, just check whether the current pointer of the file is the same as the file size (in inode).

3 Extra File changes

There only I extra file change that is the **kernel.C**. Two parts are changed:

1, **line 303** in my uploaded file. Switch from thread1 to thread3 rather than thread2.

This remove a weird bug. If the thread1 switch to thread2 (as original) the system will hang. However, after this modification, it works fine. I have no idea what's going on here.

2, **start from line 149,** I added another test case for large file (20KB for each file). This can test the read write logical for indirect index block.

4 Debug and Test

During debugging I found a strange bug. When enabling the gdb, the system will hang in at the is_ready() function for the simple disk. I guess this related to the bochs implementation of emulating the device which doesn't support the debugger.

So, in order to debug my code, I mostly use the printf (Console::puts.puti) to perform the debugging. I also added another test case (and got passed) named exercise_file_system2() as below which test large files (20KB each file). This will go through the logic of indirect index for both file read and write.

```
void exercise_file_system2(FileSystem * _file_system)
{
    const char * STRING1 = "01234567890123456789";
    const char * STRING2 = "abcdefghijabcdefghij";
```

```
int loop_size = 512;
/* -- Create two files -- */

assert(_file_system->CreateFile(1));
assert(_file_system->CreateFile(2));

/* -- "Open" the two files -- */

File * file1 = _file_system->LookupFile(1);
assert(file1 != NULL);

File * file2 = _file_system->LookupFile(2);
assert(file2 != NULL);

/* -- Write into File 1 -- */
file1->Rewrite();
for (int i = 0; i < loop_size; i++)
	file1->Write(20, STRING1);

/* -- Write into File 2 -- */

file2->Rewrite();
for (int i = 0; i < loop_size; i++)
	file2->Write(20, STRING2);

/* -- "Close" files -- */
delete file1;
delete file2;

/* -- "Open files again -- */
file1 = _file_system->LookupFile(1);
file2 = _file_system->LookupFile(2);

/* -- Read from File 1 and check result -- */
file1->Reset();
char result1[30];
for (int i=0;i<loop_size;i++)
{
	assert(file1->Read(20, result1) == 20);
	for(int i = 0; i < 20; i++) {
		assert(result1[i] == STRING1[i]);
	}
}
```

```
	/* -- Read from File 2 and check result -- */
	file2->Reset();
	char result2[30];
	for (int i=0;i<loop_size;i++)
	{
		assert(file2->Read(20, result2) == 20);
		for(int i = 0; i < 20; i++) {
			assert(result2[i] == STRING2[i]);
		}
	}

	/* -- "Close" files again -- */
	delete file1;
	delete file2;

	/* -- Delete both files -- */
	assert(_file_system->DeleteFile(1));
	assert(_file_system->DeleteFile(2));
}
```