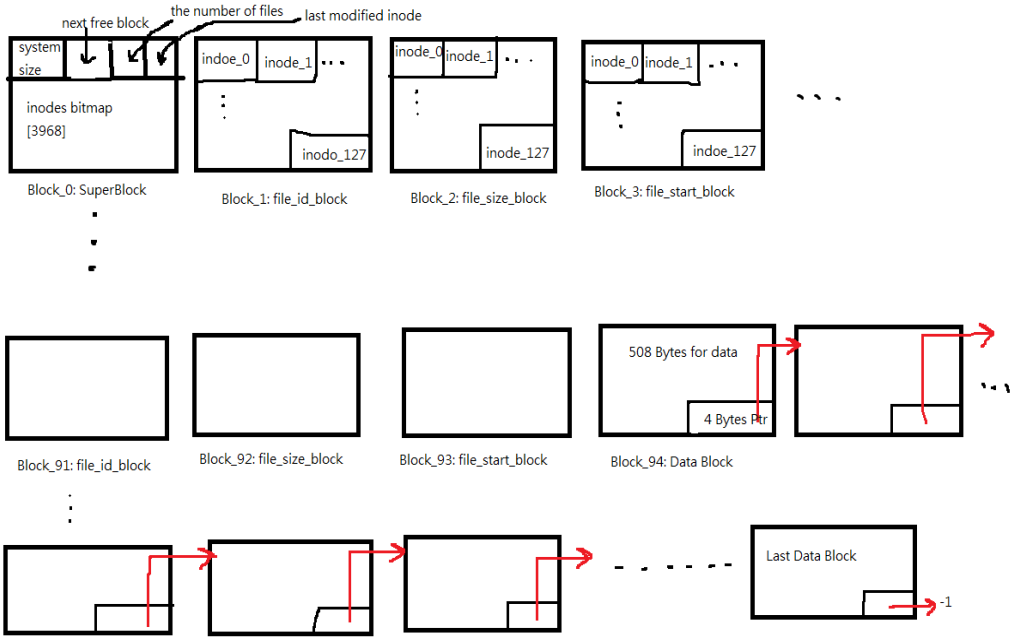
OS611_MP7_Report

Name: Wei-Che Hsu
UIN: 826001725

In this MP, I finished one file system for the **basic requirement** and proposed idea for **bonus option 1**. When grader tests my MP7 submission, please replace these 4 files: **file.C, file.H, file_system.C, file_system.H** in the test folder. For the bonus option1, I design the file system for thread-safe consideration. However, I haven't implemented it yet, so it is just draft and idea, which I would describe in page 5~6 in this report.

First, I would like to describe **my file system design**. The below picture is my block layout for my file system after initialization.



The head of block layout is **"Superblock"**, it includes one 4-Bytes variable to record the size of mounted disk (in original example in kernel.C is 1MB), one 4-Bytes variable to record the next available data block, one 4-Bytes variable to record the number of total files, one 4-Bytes to record the last modified i-node (for file-system update inode information), and the one 496-Bytes bitmap for 3968 indoes. If the bit of one inode in bitmap is 1, it means that this inode has been occupied by one file.

The index of Superblock is Block_0, and the following blocks from Block_1 to Block_93 are "inodeblocks". One "**inode**" maps one file and includes the information of file's id, file's size, and file's beginning block_no. For easier coding, I separate one inode into 3 blocks. For example, Block_1, Block_4, ……, Block_91 records file_id information for each file. Block_2, Block_5, ……, Block_92 records file_size information for each file. Block_3, Block_6, ……, Block_93 records

start_block_no information for each file. But this design will sacrifice performance because reading one inode should take 3 disk block IO operation.

In the previous introduction, the management of inode blocks depends on bitmap, and the management of data blocks depends on free-link-list. From block_94 to the end block of this file system, all blocks are data blocks. One data block includes 508-Bytes space for accessing data and 4-Bytes pointer to record the next free data blocks. Then, I will explain how this link-list works.

For data block allocation: When the file system is initialized, all data blocks are free, so the pointer in each data_block points to the nearby next data_block. Assume the value of next available data block in superblock is block "X". When one file creates, file system does not allocate data block for this file. Until user assigns writing task to this file, file system allocates the block "X" which is recorded in next available data block in superblock. After this allocation, the value of next available data block becomes the block "Y" which was pointed by the allocation block "X". Now, for this file, the block X with the pointer to Y is meaningless, it can't use this pointer at this time. After 508 Bytes ran off by writing operation, filesystem will allocate one block to this file. At this time, the value of next available data block is the block "A" because some other file has already used Y. Thus, file system assigns block "A" to this file, the value of the pointer in the block "X" changes to "A", and the value of next available data block in superblock becomes the block "B" which was pointed by the allocation block "A". When the value of next available data block is -1, it means that there is no available data block.

For data block recycling: When user would like to rewrite one file or delete one file, file system has to recycle the used data block of this file. For example, one file with 3 data blocks : block_A →block_B→block_C. The recycling method can recycle the block from beginning to end because rewrite operation and delete operation recycles all block instead of partial blocks. Assume the value of next available data block in superblock is block_Y. First, file system recycles block_A, and set the value of next available data block as block_A , and set the pointer in block_A points to block _Y. Then, file system recycles block_B, and set the value of next available data block as block_B, and set the pointer in block_B points to block _A. Finally, file system recycles block_C, and set the value of next available data block as block_C, and set the pointer in block_C points to block _B. After recycling, the file_size of this file is 0, and the value of next available data block is block_C. In the free blocks linking is block_C→ block_B→ block_A→ block_Y→…

Next, I will briefly describe the purpose of each **function/variable** in **file** class and **file_system** class:

file_system:

SimpleDisk * disk; // I record the pointer to the mounted disk.
size; // the available size of mounted disk
prep_freeblock_index; // the next available free block
file_count; // the number of files in this filesystem
last_modified_inode; //to know which inode should update its information
inodeFlags_array[INODE_FLAGS_COUNT]; //bitmap for inodes
inodeFileID_array[128], inodeFileSize_array[128],
inodeFileStartBlockIndex_array[128] //record 128 inodes information from 3 blocks
The constructor of FileSystem initial above variables except for disk and 3 inodes buffer.

The following 5 functions has already described in MP7 handout:
bool Mount(SimpleDisk * _disk);
static bool Format(SimpleDisk * _disk, unsigned int _size);
File * LookupFile(int _file_id);
bool CreateFile(int _file_id);
bool DeleteFile(int _file_id);

Additional functions:
void UpdateInode();
void LoadInode(int _inode_index);
//when doing file **writing operation, rewrite operation and delete operation** , inode information will change. But, the three inode buffer[128] is public used for different File objects because these objects maps to one file system. Therefore, it should use the correct inode information when accessing these operations.
int FindFreeInode(); //find the index of free inodes.

file:

file_id;
file_inode_index;
int file_size;
file_start_block;
//above 4 variables are inode information which records in this file object, if they needs to update to inode, called filesystem:: UpdateInode()

current_location; //unit: bytes; range: 0~file_size(EOF)
current_block; //where is the current block?
FileSystem* file_system; //pointer to the file system in the disk

File(int _file_id, unsigned int _file_size, unsigned long _file_start_block, unsigned int _file_inode_index, FileSystem* _file_system);
This constructor calls in File * LookupFile(int _file_id); This constructor will initial above variables.

The following 5 functions has already described in MP7 handout:
int Read(unsigned int _n, char * _buf);
void Write(unsigned int _n, const char * _buf);
void Reset();
void Rewrite();
bool EoF();

For my basic implementation for this file system is not safe for multiple File* to one file. After some operations from multiple File* would make filesystem work incorrectly. For example, user creates two File* to one file at some moment. One File* called File_pa, and the other File* called File_pb. They set commands to this file in interleaved way. File_pa writes 600 bytes to this file. Next, File_pb writes 200 Bytes in this file sequentially, but it will fail because there is no way for File_pb to update its current_location and current_block from the previous operation. How to synchronize information between different File* is the key factor for multiple thread-safe.
Now, my implementation accesses **one file** with **one File* object**, which is not support multiple File* which are initialized with the same condition to access one file. For above example, for correctly accessing file, just make File_pa write 600 bytes, and next make File_pa write 200 bytes.

[Thread-Safety consideration]

For thread-safe consideration, I should firstly solve synchronization problem between multiple File*. Because I only have upload information from File object to inode in my submission. Now, in **file_system**, it should be add one function which download information from inode to File object. Also, inode information should include current location and current block for each file. Before File object takes read, write, rewrite, reset, Eof function, it should call this function to make sure that the variables in this File object is the latest.

Second, some commands should take care, like “write and write (W/W)” as well as “read and write(R/W)”. Because doing these commands would bring risk to obtain in-concurrent data in file system. Therefore, In **File** class, I will add new variable for each File*, for indicating this File* is only for read or not? If it is only for read, this File* is reader which can only call read and reset functions; otherwise, it is writer which can do all operations in File class.

File * LookupFile(int _file_id, bool read_only);

File(int _file_id, unsigned int _file_size, unsigned long _file_start_block, unsigned int _file_inode_index, FileSystem* _file_system, bool read_only);

Also, design the lock system in **file_system**. Here, I refer the text-book 5.7.2 Readers–Writers Problem [1]. Apply the solution of type-1 problem which is beneficial for readers. **For every File operation,** When File* object would like to call any functions in File class, it should pass one scheduler of file_system. This scheduler includes the following 2 functions to handle thread/process of reader and writer.

Initially, the following value

semaphore rw mutex = 1;

semaphore mutex = 1;

int read count = 0;

rw mutex can avoid both W/W and R/W cases.

mutex is used for avoiding read count race condition.

The priority of any reader is higher than any writer which not writes yet.

wait(semaphore X); it will make X reduce its value 1. If it is negative value, the process/thread is blocked here. Otherwise, it can access critical section.

signal(semaphore X) ; it will make X increase its value 1.

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

Line2: wait(rw_mutex) to confirm W/W and R/W exclusive property.

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
     wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
     signal(rw_mutex);
   signal(mutex);
} while (true);
```

Figure 5.12 The structure of a reader process.

Line4: if there is one reader, it can do reader task.
Line5: wait(rw_mutex) to confirm W/W and R/W exclusive property.
Line10: This line shows that the priority of reader is higher than that of writer.

Last, **file_system operation** should be also considered. Here, I would take one filter-lock to protect all functions, like “create file” or “delete file”, in file_system except the above scheduler. If user deletes one file, program should clear all operations regarding this file in scheduler.

Reference:

[1] Operating System Concepts 9th edition, by Silberschatz, Galvin, Gagne, John Wiley & Sons, Inc., New York, 2012.