**Design**

In this MP, I designed and realized a file system for the basic assignment and proposed an idea for the first bonus option. When grader tests my MP7 submission, please replace these 8 files: *kernel.C, makefile, scheduler.H, scheduler.C, file.C, file.H, file\_system.C,* and *file\_system.H* in the test folder. In order to realize FIFO, I added *scheduler.H, scheduler.C*, and modified *makefile*. And I also modified *file.C, file.H, file\_system.C,* and *file\_system.H* in order to realize a simple file system. Also, in *kernel.C*, I enabled *\_USES\_SCHEDULER\_* and changed the stack size from 1024 to 4096 in thread 3.

First, I would like to describe my file system design. Figure 1 shows my block layout for my file system after initialization. Now, let’s begin.

***FileSystem***

In the class of FileSystem, I maintained 5 unsigned int variables and an inode list to record the information of superblock, which is always maintained in memory. The most important thing I learned in MP7 is a trade-off between the memory (cache) size and the cost of read/write. The more information we can cache in our memory, the less write/read we need to do.

The first and second variables are the size of this file system and the number of files, and the third one is the next free block, which is very important for us to create a new file or allocate a new block to an existed file. The second function, allocating a new block to an existed file, is also provided in my submitted code, but in kernel.C we do not need this function. The fourth variable is the maximum value of inode number. In my previous consideration, this is used to give an upper limit while searching files. However, I optimized my design by introducing a directory and this variable is not useful in my submitted version. In my design, the director is constructed and allocated to the first free block (always block 16) when we create the first file in this file system. It occupies one block and records the file id and the inode number of a file, each of this pair using 8 characters. And we should pay attention to the fifth variable, which records the directory block number. Using this variable, we can enter into the directory block any time to look up a file. And at the end, we use a 480 bytes array to record 480 characters, which can be set as 0 or 1 to represent the state of its corresponding inode.

Since the total size of these variables is almost as large as one block, the original version of kernel.C in which the stack size of thread 3 is only two block (1024 Bytes) is not enough anymore. We have to increase the stack size and I choose to increase to 4096 Bytes, as is shown in Figure 2. After designing the file system, every function will follow smoothly. And in the following content, we will show some key points in these functions.

The first function is the static function “Format” which is used to initialize the system disk. We set all 32\*15 inodes in block 1 to block 15 as unoccupied state and the inode struct is shown in Figure 1. Then we string all blocks from block 16 to the end of the disk by the 4 bytes pointer at the end of each block. This will cost a little long time since we need to write more than 2000 blocks, as is shown in Figure 3. And also, we need to set the first block, block 0, as superblock, and write the information of file system into the disk. After that, we associate the disk with our new constructed file system, caching all information in the superblock into the thread stack.

We also add four functions to get free inodes and blocks, and also to free them. These functions are very useful when we create a new file or delete an existed file. The realization of “LoopupFile”, “CreateFile” and “DeleteFile” is not very difficult based on these four functions. However, we should pay attention to the consistency between the memory and the disk, which means we need to update the superblock and inode list on the disk in time. The read/write operations here are necessary that we cannot omit.

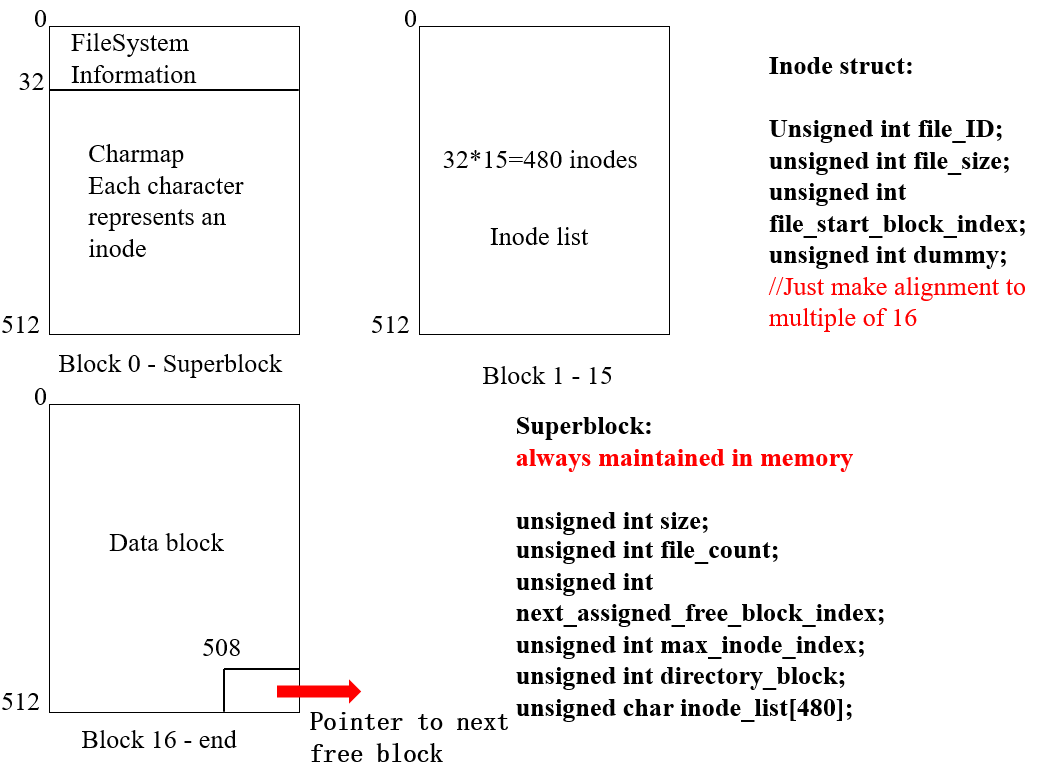


Figure 1

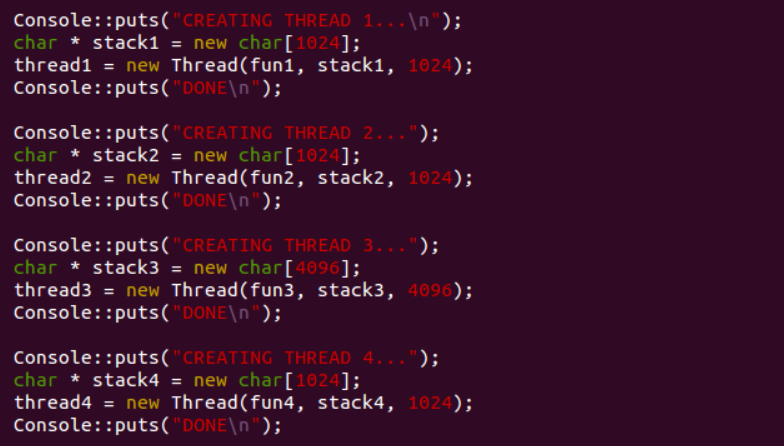


Figure 2

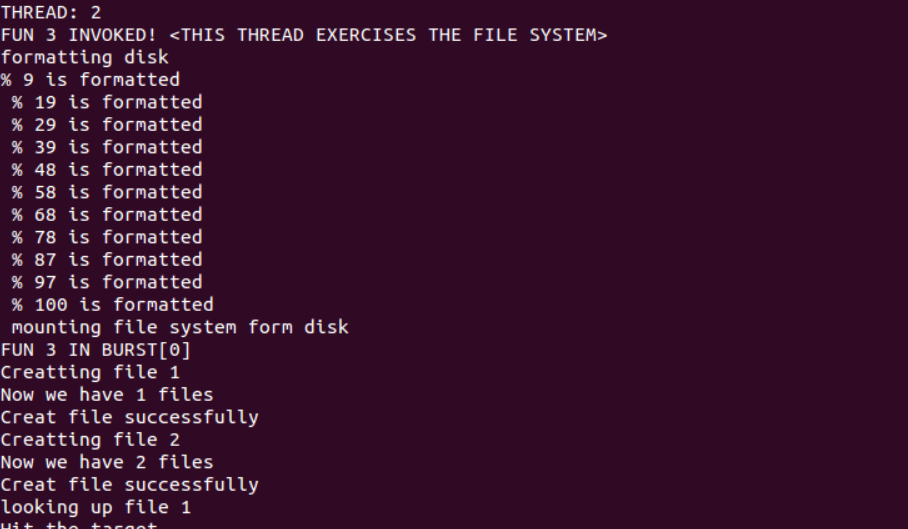


Figure 3

***File***

In the class of File, I maintained 6 variables to record file id, file size, the first block, the inode number, current location, and current block. For “Read” and “Write” operation, in this MP7 we do not need to enter into the second block of the file because our testing code only checks a small file, less than 30 characters. However, my submitted code can support operations on a file with more than one block. The most important thing when reading or writing a data block of a file is that the last 4 bytes are reserved as a pointer, pointing to the next data block of this file (set as -1 in the last block of the file). Thus, we can only read or write 508 bytes in each block.

And for “Rewrite” function, I reset the current location and the current block as the beginning of the file and also set the file size as 0. However, the file still maintains all block(s) allocated previously, which means next time when we write new information to this file, we do not need to allocate a new block.

***Bonus 1***

For thread-safe design, let’s have a brief discussion. Ee should carefully solve the synchronization problem between multiple threads, the current location and current block between which may be not the same. So every time when we read or write a block, we should download the recently updated information from the inode list in the block and when we finish the r/w operation, we have to update the inode information as soon as possible. Thus, the inode list should include current location and current block for each file. This seems really a heavy-hand method, but it works for a FIFO scheduling.

And if we face a more complicated scheduling, for example, Round Robin, we need to add a lock. This means when a thread is writing to a file, other threads cannot write or read, and when a thread is reading a file, other threads cannot write but maybe can read which makes our design a little more complicated.