CSS 430
FAQ on Final Project: File System

Q1: If I work with my partner, how should we divide the project work into two sub tasks?

A: One person could implement all the parts used by the file system such as Inode.java, FileTable.java, Superblock.java, and Directory.java. The other person could implement FileSystem.java that uses those parts to implement the eight file-realted system calls.

Q2: How can we save various objects into the disk? How can we convert different types of data into bytes?

A:

You can't save an entire Superblock object, an entire Inode object, and an entire Directory object. You should identify which data members in Superblock, which data members in an Inode, and which data members in Directory should be saved. For instance, the meaningful Inode data to be save include:

```
        int length;
        short count;
        short flag;
        short direct[11];
        short intdirect;
```

So, rather than saving an entire Inode class, you should save only *length, count, flag, direct[11],* and indirect. Since all of those are not bytes, you must convert them into bytes. Then, you must use:

```
        SysLib.int2bytes( );
        SysLib.short2bytes( );
```

All data conversions from short to bytes or vise versa or from int to bytes or vise versa have been implemented in SysLib.java. You can use them.

```
void int2bytes( int i, byte[] b, int offset )
        converts the integer i into four bytes, and then
        copied those four bytes into
                b[offset], b[offset+1], b[offset+2], b[offset+3]

void short2bytes( short s, byte[] b, int offset )
        converts the short s into two bytes, and then
        copied those two bytes into
                b[offset], b[offset+1]

int bytes2int( byte[] b, int offset )
        converts the four bytes including
                b[offset], b[offset+1], b[offset+2], b[offset+3]
        into an integer and returns this integer.

short bytes2short( byte[] b, int offset )
        converts the two bytes including
                b[offset], b[offset+1]
        into a short and returns this short.
```

Q3: Should I have to use Cache.java?

A: If you didn't receive a perfect grade for your assignment 4, you should not use Cache.java. Just use SysLib.rawread( ) and SysLib.rawwrite( )

Q4: In SysLib.write(int fd, byte buffer[]), what's the max byte size, buffer[]?

A: There are no maximal limits. However, you can always know the buffer size with buffer.length.

Q5: On SysLib.delete( ), should I set fd to -1? fileName to null?

A: Yes, just nullify the file name registered in the directory "/". Also don't forget to zero-initialize the corresponding file size registered in the directory.

Q6: SysLib.fsize(int fd) returns the size in bytes of the file indicated by fd. What's the max size?

A: It is the size an inode can maintain.

There are 11 direct pointers and 1 indirect pointer. The indirect pointer points to an index block which includes 256 indexes, each represented in short. Thus the maximal size is 11 * 512 + 256 * 512 bytes.

I need more explanation about the disk structure and the superblock.

A: The disk consists of 1000 block and the block#0 is the superblock.

The disk consists of 1000 blocks and you can consider those blocks are numbered from 0 to 999. Use the block #0 as the superblock. For accessing this block, you should call SysLib.rawread(0, data) where data is a 512-byte array.

Q8: Where is the memory to maintain inodes?

A: If you instantiate a new object, it is considered to be on the memory.

Q9: When is iNode updated and what does that mean?

A:

The original copy of each inode must be placed somewhere in the disk. To avoid any consistency between what you modified on an inode instance and the original content of inode on the disk, I recommend that you should write back the modification to the disk. To improve the performance, you can skip writing back the modification and perform such write-back operations only when necessary, but this is very complicated. So, don't be concerned about the performance. First, try to complete your file system. After you made sure the validity, you can tune up the performance.

Q10: Where (what variable or structure and in what class) do we hold the running count of iNodes that have been opened with open( )? In the Entry class? If so, won't we need to do iNode++?

A:

Whenever inode is pointed by a new file structure table entry, its count is incremented. When a file structure table entry is released, this count should be decremented. A new file structure table entry is created when a thread opens a file and, and it is deleted when the thread closes this file.

Q11: When is iNode updated and what does that mean?

A: An inode is update when:

1. It is pointed by a file structure table entry (count++)
2. The file structure table entry pointing to it is deallocated. (count--)
3. No more file structure table entry points to this inode.(flag=0)
4. New blocks are assigned to this inode. (direct[] and indirect modified.)
5. Blocks are deallocated to this node (direct[] and indirect modified)

Q12: Do you want the filename to be in unicode or ascii?

A: Either way is okay as far as your file system works as specified.

Q13: Since we use only one indirect pointer, we can have at most 11 + 512/2 = 267 blocks in a file. That is also the maximum of block in one file?

A: Yes, you're right.

Q14: I don't understand why Inode needs two constructors.

Inode's second constructor retrieves and read the iNumber block, then locates inode info., and initializes the new inode with this info.. Why should it locates inode info. even though it is still there? And how does it initialize the Inode object? What is the new inode in this case? Could you explain more?

A: An existing file has already had an inode. It is in the disk. You have to retrieve it from the disk when opening such an existing file

If a new file is created, it is given a new inode. In this case, you'll just instantiate it from its default constructor where all direct pointers and the indirect pointer are null. The contents will be later updated as the file is written. An existing file has already had an inode. It is in the disk. When such an existing file is opened, you should find the corresponding inode from the disk. First, refer to the directory in order to find the inode number. From this inode number, you can calculate which disk block contains the inode. Read this disk block and get this inode information. Where should you store such inode information then? You should instantiate an inode object first, and then reinitialize it with the inode information retrieved from the disk.

Q15: Is there a limit to the number of FileStructureTable entries?

A: No.

Q16: Can we read on an append?

A: No

Q17: Do we let multiple threads access files on write, write/read, and append?

A: No.

Q18: Is our file system responsible for writing EOF?

A: No.

Q19: What calls shutdown? Do we have to handle a shutdown that happens while processes are running? Do we have to modify Loader.java?

A: Modify the EXIT case in Kernel.java, which is much easier to modify.

If processes are running, the Loader.java never gets control. So, you don't have to think about such a special case in that a shutdown is requested while some processes are running. Just simply modify the EXIT case statement in Kernel.java to reflect all Disk.java contents to the Disk file.

Q20: From what I saw the project description, the FileStructureTableEntry class includes an Inode in it. Why?

If anything in the inode changes (like the count), how will the FileStructureTableEntry be updated? For instance, Thread 1 opens a file - it gets the Inode from the disk, and sets count=1 in the Inode for FileTable Entry 1, then writes the Inode back to disk. Thread 2 opens the same file and creates FileTable Entry 2, goes to the disk and gets the Inode, then sets count=2 then saves the Inode back to disk. Now I have 2 threads opening the same file, but the Inode count is different in the two FileTable Entries.

A: The FileStructureTableEntry includes a reference, (i.e., a pointer to an Inode rather than the Inode itself.

Let's think about this story: Thread1 opens file 1. File table entry 1 is allocated to this thread. Its count becomes 1. The corresponding inode is pointed to by this file table entry. The inode's count is incremented to 1. Thread2 opens the same file. File table entry 2 is allocated to this thread, and its count becomes 1. The same inode is pointed to by the file table entry 2. The inode's count is incremented to 2. Thread1 spawns a child thread. It shows the file table entry 1. Thus, the file table entry 1's count becomes 2. In such a case, Thread 1 and its child thread shares the file table entry 1 which includes a seek pointer. Therefore, they can work together on the same file with the same seek pointer, which Thread 2 works on the same file independently with its own seek pointer.

Q21: Should the SuperBlock be located in the FileTable class or FileSystem class?

A: It's up to your implementation.

My implementation instantiates it in *FileSysytem.java*.

Q22: How many inodes are supposed to create?

A: The same as the argument passed to format( ).

Q23: If the total number of inodes is 64, from which block does the free list start?

A:

```
The superblock has:
  totalBlocks = 1000
  inodeBlocks = 64 or (4 if you interpret it is #blocks including
inodes)
  freeList = 5 (block#0 = super, blocks#1,2,3, and 4 = inodes.)
```

Q24: Could you explain what the format actually does?

A: format(int files) decides how many files you would like to create in the file system.

The *files* argument passed to *format* indicates the maximum number of inodes. If files is 48, #inodes is 48. This means, 48/16 = 3 blocks will be allocated to maintain those 48 inodes. Then, you can see:

```
        block #0: superblock
        block #1: inodes 0 - 15
        block #2: inodes 16 -31
        block #3: inodes 32 -47
        block #4: the first free block
```

Q25: When am I supposed to add FileTableEntry in FileTable?

A: Whenever a Thread opens a file, a new FileTableEntry must be allcoated.

This entry then must include a reference to the corresponding inode. If this thread spawns a child thread, this child thread should share the same File TableEntry so that they can work on the same file with the same "seek pointer".

Q26: If the open mode is "w", should writing be allowed beyond the file length? If that were to be true, then there would be no difference between "w" and "a".

A: Yes.

"w" should actually delete all blocks first and then start writing from scratch. Contrary to "w", the "a" mode should keep all blocks, set the seek pointer to the EOF, and thereafter appends new blocks.

"w+" should keep all blocks. If the seek pointer is within the EOF, the corresponding bytes should be updated. If the seek pointer is at the EOF, it behaves like "a".

Q27: I'm asking how we boot ThreadOS for the very first time before formatting?

At the very first time we boot, before we call format, there is no valid info on disk to fill Superblock and Directory. However, as I see it, ThreadOS can't know this and tries to read data from block 0 and inode 0 even so. However, if there is bad data in those blocks, such as negative values, then we could crash.

A: At the very beginning, the Disk.java thread zero-initialize all bocks.

When the block #0 is all zero, you recognize that the disk was just created. One implementation is just formatting this disk with your default value like format(48) or format(64). Then, once a user requests format(...), you should reformat your disk.

Q28: I saw the DISK file in the ThreadOS directory. What is it?

A: This is what ThreadOS' Disk.java has saved upon SysLib.sync( ).

If you delete DISK, ThreadOS recognizes that the current system boot is the very first one, and thus zero-initializes all disk contents. Whenever SysLib.sync( ) is called, ThreadOS saves all disk contents into DISK.