FINAL Project (JAVA)

Sunday, November 29, 2015 3:35 PM

File System

IMPORTANT: THIS IS A **DESIGN** PROJECT, meaning that you will have to design AND implement parts of the file system based on the skeleton files. Descriptions will not be provided.

You may work in teams of 2 or on your own.

0. Required Reading

Read Chapters 11 and 12 on the textbook "File system interface" and "File system implementation". The chapter numbers and titles may differ depending on your book's edition. However, you can use the slides to identify the topics that you need to read about in preparation for this project.

	Ch13 in 10th Edition 11.1 - 11.3, 11.6.1, 11.6.2
	Ch 14-15 in 10th Edition 12.1, 12.2.1, 12.2.2, 12.3, 12.4, 12.5, 12.6.2
Project "mini Lecture" from SP23	https://washington.zoom.us/rec/play/FLbUyJZiD-YhKffmzoBFKS0FzOd313MPT51isahCI0xmBDGEVHTXCEZI4ihAYwb4qeqtX1DWci38Co9T.Um3bYytPLzjHCWRF
FinalProject_v2	A set of slides dedicated to the project

1. Goal

In this project, you will build a Linux-like file system on the ThreadOS. Through the use of the file system, user programs will now be able to access persistent data on disk through stream-oriented files instead of [the more painful] direct access to disk blocks with `rawread()` and `rawrite()`.

2. Collaboration

Given the size of this project, you are encouraged to form a team of two students (no teams of 3 students allowed). The workload should divided *equally* across all members. This project requires much more time to be completed than the previous four assignments. Start early, manage your time and schedule design and coding accordingly. You may work independently if you feel comfortable to do so.

3. What Files to Use?

IMPORTANT: DO NOT USE THE CODE IN THE `cssmpilab` MACHINES!

I recommend that you use the files in the zip file below. Or, if you decide to download the code from the lab and have any issues with the .class files try using the zip files instead

Option 1 Download: I recommend you just get the code from this zip file

Zip file containing class files and kernel source from here ☆ This zip file contains both, the classes for ThreadOS AND free extra code I am giving you in Autumn 21 to help you out, so even if you get your code from the lab, you may want to get this zip to get that extra code ;-)	 FinalProject_21

Option 2 Linux Lab: Get a fresh copy of ThreadOS from the normal location except for `Kernel.java`. For `Kernel.java`, use the one you got in the assignment 4, (i.e., *`Kernel_org.java`*). For simplicity, you should just re-download all readable files from the *ThreadOS* directory to get started.

The complete ThreadOS source code and .class files can be found in the UW1-320 Linux machines
`/usr/apps/CSS430/ThreadOS/`

A note on `Kernel.java`: if for some reason you are using the version that has the following line of code in `RAWREAD`, `RAWWRITE` and `SYNC`: `Kernel.ioQueue.dequeueAndWakeup(Kernel.COND_DISK_REQ);`
You can remove these lines of code.

4. Statement of Work

Design and implement a **single level** file system in *ThreadOS* as specified below. Run Test5.class library provided in the *ThreadOS* folder.

4.1 Interface

The file system should provide user threads with the system calls that will allow them to format, to open, to read from, to write to, to update the seek pointer of, to close, to delete, and to get the size of their files.

For simplicity, the file system being created will consist of a single level. The "/" root directory is predefined by the file system and permanently available for user threads to store their files. No other directories are provided by the system or dynamically created by users.

Each user thread needs to keep track of all files it has opened. To support this, each thread should maintain a table of those open files in its TCB. Such a table is also known as a user file descriptor table and it has 32 entries. Whenever a thread opens a file, it must allocate a new table entry for the file; this is known as a *file descriptor*. Each file descriptor includes the file access mode and the reference to the corresponding file (structure) table entry.

The file access mode indicates "read only", "write only", "read/write", or "append".

The file (structure) table is a system-maintained table shared among all user threads, each entry of which maintains the seek pointer and the inode number of a file. Depending on the access mode, the seek pointer is set to the front or the end of the file, and keeps track of a next position to read from and to write to the file.

It is possible for a thread to open the same file many times, thus having several entries in the corresponding TCB's user file descriptor table. Each of user file descriptor table entries refer to a different file (structure) table entry with its own seek pointer, all of them ultimately point to the same inode.

The file system implementation must provide the eight system calls described in Table 1.

(Note: you will have to update both `Kernel.java` and `Syslib.java`).

Table 1. System calls for the file system implementation and descriptions. Unless specified otherwise, each of the following system calls return -1 when detecting an error.

Method	Description
`int SysLib.format( int files );`	1. Formats the disk (`Disk.java`'s data contents). • The files parameter specifies the maximum number of files to be created (The number of inodes to be allocated) in your file system. 2. The return value is 0 on success, and -1 otherwise.
`int fd = SysLib.open( String fileName, String mode );`	1. Opens the file specified by the `fileName` string in the given mode (where "r" = ready only, "w" = write only, "w+" = read/write, "a" = append). 2. The call allocates a new file descriptor, fd to this file. 3. The file is created if it does not exist in the mode "w", "w+" or "a". 4. `SysLib.open` must return a negative number as an error value if the file does not exist in the mode "r". See Figure 1 (below) for clarification.

`int read( int fd, byte buffer[] );`	1. Reads up to `buffer.length` bytes from the file indicated by the file descriptor `fd`, starting at the position currently pointed to by the seek pointer. 2. If bytes remaining between the current seek pointer and the end of file are less than `buffer.length`: a. `SysLib.read` reads as many bytes as possible and puts them into the beginning of buffer. b. It increments the seek pointer by the number of bytes to have been read. c. The return value is the number of bytes that have been read, or a negative value upon an error.
`int write( int fd, byte buffer[] );`	1. Writes the contents of buffer to the file indicated by `fd`, starting at the position indicated by the seek pointer. • The operation may overwrite existing data in the file and/or append to the end of the file. 2. `SysLib.write` increments the seek pointer by the number of bytes to have been written. 3. The return value is the number of bytes that have been written, or a negative value upon an error.
`int seek( int fd, int offset, int whence );`	Updates the seek pointer corresponding to fd as follows: • If whence is SEEK_SET (= 0), the file's seek pointer is set to offset bytes from the beginning of the file • If whence is SEEK_CUR (= 1), the file's seek pointer is set to its current value plus the offset. The offset can be positive or negative. • If whence is SEEK_END (= 2), the file's seek pointer is set to the size of the file plus the offset. The offset can be positive or negative. If the user attempts to set the seek pointer to a negative number you must clamp it to zero and the return must be **successful**. If the user attempts to set the pointer to beyond the file size, you must set the seek pointer to the end of the file and the return must be **successful**. The offset location of the seek pointer in the file is returned from the call to seek.
`int close( int fd );`	1. Closes the file corresponding to `fd`, commits all file transactions on this file 2. Unregisters `fd` from the user file descriptor table of the calling thread's TCB. 3. The return value is 0 in success, otherwise -1.
`int delete( String fileName );`	1. Destroys the file specified by `fileName`. • If the file is currently open, it is not destroyed until the last open on it is closed, but new attempts to open it will fail.
`int fsize( int fd );`	Returns the size in bytes of the file indicated by `fd`.

Operation	**Mode**	**File**	**What to do?**
Open	r	exists	Open the file for reading
		does not exist	Return error (negative number)
	w, w+, a	exists	Open the file for writing
		does not exist	Create the file

Figure 1. Detail on different modes of the Open system call

Note that the file descriptors 0, 1, and 2 are reserved as the standard input, output, and error, and therefore a newly opened file must receive a new descriptor numbered in the range between 3 and 31. If the calling thread's user file descriptor table is full, `SysLib.open` should return an error value. The seek pointer is initialized to zero in the mode "r", "w", and "w+", whereas initialized at the end of the file in the mode "a".

4.2. Implementation

The first disk block, block 0, is called the *superblock*. It is used to describe:

- The number of disk blocks.
- The number of inodes.

- The block number of the head block of the free list.

The superblock is a block managed by the OS. No other information must be recorded in and no user threads must be able to get access to the superblock.

```java
class Superblock {
   public int totalBlocks; // the number of disk blocks
   public int totalInodes; // the number of inodes
   public int freeList;    // the block number of the free list's head

   public SuperBlock( int diskSize ) {
       ...
   }
   ...
}
```

[view raw](#) [Superblock.java](#) hosted with ❤ by [GitHub](#)

Note: the data in the disk is in bytes, so you will need to translate to integers back and forth as needed. For this purpose, `SysLib` provides the functions **`bytes2int`** and **`int2bytes`**. For example, in SuperBlock above, totalBlocks is an integer which can be obtained as:

```
totalBlocks = SysLib.bytes2int(superBlock, offset);
```

here `superBlock` is a byte array of 512 (the maximum disk block size).

Similarly, `int2bytes`, is used to calculate offsets in bytes for Free List management.

Inodes

The inode blocks start after the superblock. Each inode describes one file. The *inode* in this project is a simplified version of the Linux *inode*. The inode includes 12 pointers of the index block. The first 11 of these pointers point to direct blocks. The last pointer points to an indirect block. In addition, each inode must include

(1) the length of the corresponding file

(2) the number of file (structure) table entries that point to this inode

(3) the flag to indicate if it is unused (= 0), used(= 1), or in some other status (= 2, 3, 4, ..., i.e., what/how the file is currently being used for).

Note that 16 *inodes* can be stored in one block.

```java
public class Inode {
   private final static int iNodeSize = 32;       // fix to 32 bytes
   private final static int directSize = 11;      // # direct pointers

   public int length;                             // file size in bytes
   public short count;                            // # file-table entries pointing to this
   public short flag;                             // 0 = unused, 1 = used, ...
   public short direct[] = new short[directSize]; // direct pointers
   public short indirect;                         // a indirect pointer

   Inode( ) {                                     // a default constructor
      length = 0;
      count = 0;
      flag = 1;
      for ( int i = 0; i < directSize; i++ )
         direct[i] = -1;
      indirect = -1;
   }

   Inode( short iNumber ) {                       // retrieving inode from disk
      // design it by yourself.
   }

   int toDisk( short iNumber ) {                  // save to disk as the i-th inode
      // design it by yourself.
   }
}
```

view raw Inode.java hosted with ❤ by GitHub

You will need a constructor that retrieves an existing *inode* from the disk into the memory. Given an *inode* number, termed *inumber*, this constructor reads the corresponding disk block, locates the corresponding *inode* information in that block, and initializes a new *inode* with this information.

The system must avoid any *inode* inconsistency among different user threads. There are two ways of maintaining the *inode* consistency:

1. Before an *inode* in memory is updated, check the corresponding *inode* on disk, read it from the disk if the disk has been updated by another thread. Then, write back its contents to disk immediately. Note that the *inode* data to be written back include `int length`, `short count`, `short flag`, `short direct[11]`, and `short indirect`, thus requiring a space of 32 bytes in total. For this write-back operation, you will need the `toDisk` method that saves this inode information to the `iNumber`-th *inode* in the disk, where `iNumber` is given as an argument.
2. Create a `Vector<Inode>` object that maintains all *inode* on memory, is shared among all threads, and is exclusively access by each thread.

Root Directory

For simplicity, the file system being created will consist of a **single level**. The "/" root directory is predefined by the file system and permanently available for user threads to store their files. No other directories are provided by the system or dynamically created by users.

The "/" root directory maintains each file in a different directory entry that contains its file name (maximum 30 characters; 60 bytes in Java) and the corresponding *inode* number. The directory receives the maximum number of *inodes* to be created, (i.e., thus the max. number of files to be created) and keeps track of which *inode* numbers are in use. Since the directory itself is considered a file, its contents are maintained by an *inode*, specifically *inode* 0. This is located in the first 32 bytes of the disk block 1.

Upon booting ThreadOS, the file system instantiates the `Directory` class as the root directory through its constructor, reads the file from the disk -- found through inode 0, at 32 bytes of the disk block 1 -- and initializes the Directory instance with file contents. Prior to shutdown, the file system must write back the `Directory` information onto the disk. The methods `bytes2directory( )` and `directory2bytes` are used to initialize the `Directory` instance with a byte array read from the disk, and convert the `Directory` instance into a byte array, which will later be written back to the disk.

```java
public class Directory {
   private static int maxChars = 30; // max characters of each file name

   // Directory entries
   private int fsize[];        // each element stores a different file size.
   private char fnames[][];    // each element stores a different file name.

   public Directory( int maxInumber ) { // directory constructor
      fsizes = new int[maxInumber];     // maxInumber = max files
      for ( int i = 0; i < maxInumber; i++ )
         fsize[i] = 0;                 // all file size initialized to 0
      fnames = new char[maxInumber][maxChars];
      String root = "/";                // entry(inode) 0 is "/"
      fsize[0] = root.length( );        // fsize[0] is the size of "/".
      root.getChars( 0, fsizes[0], fnames[0], 0 ); // fnames[0] includes "/"
   }

   public int bytes2directory( byte data[] ) {
      // assumes data[] received directory information from disk
      // initializes the Directory instance with this data[]
   }

   public byte[] directory2bytes( ) {
      // converts and return Directory information into a plain byte array
      // this byte array will be written back to disk
      // note: only meaningfull directory information should be converted
      // into bytes.
   }

   public short ialloc( String filename ) {
      // filename is the one of a file to be created.
      // allocates a new inode number for this filename
   }

   public boolean ifree( short iNumber ) {
      // deallocates this inumber (inode number)
```

```
        // the corresponding file will be deleted.
    }

    public short namei( String filename ) {
        // returns the inumber corresponding to this filename
    }
}
```

Directory.java hosted with ❤ by GitHub

Starter code

Directory

File (Structure) Table

The file (structure) table is a system-maintained table shared among all user threads, each entry of which maintains the seek pointer and the inode number of a file. Depending on the access mode, the seek pointer is set to the front or the end of the file, and keeps track of a next position to read from and to write to the file.

The file system maintains the file (structure) table shared among all user threads. When a user thread opens a file, it follows these steps:

1. The user thread allocates a new entry of the user file descriptor table in its TCB. This entry number itself becomes a file descriptor number. The entry maintains a reference to a file (structure) table entry.
2. The user thread then requests the file system to allocate a new entry of the system-maintained file (structure) table. This entry includes the seek pointer of this file, a reference to the inode corresponding to the file, the inode number, the count to maintain #threads sharing this file (structure) table, and the access mode. The seek pointer is set to the front or the tail of this file depending on the file access mode.
3. The file system locates the corresponding inode and records it in this file (structure) table entry.
4. The user thread registers a reference to this file (structure) table entry in its file descriptor table entry of the TCB.

The file (structure) table *entry* should be:

```
public class FileTableEntry {          // Each table entry should have
   public int seekPtr;                 //    a file seek pointer
   public final Inode inode;           //    a reference to its inode
   public final short iNumber;         //    this inode number
   public int count;                   //    # threads sharing this entry
   public final String mode;           //    "r", "w", "w+", or "a"
   public FileTableEntry ( Inode i, short inumber, String m ) {
      seekPtr = 0;             // the seek pointer is set to the file top
      inode = i;
      iNumber = inumber;
      count = 1;               // at least on thread is using this entry
      mode = m;                // once access mode is set, it never changes
      if ( mode.compareTo( "a" ) == 0 ) // if mode is append,
         seekPtr = inode.length;        // seekPtr points to the end of file
   }
}
```

FileTableEntry.java hosted with ❤ by GitHub

The file (structure) table is defined as follows:

```
public class FileTable {

   private Vector table;         // the actual entity of this file table
   private Directory dir;        // the root directory

   public FileTable( Directory directory ) { // constructor
      table = new Vector( );     // instantiate a file (structure) table
      dir = directory;           // receive a reference to the Director
   }                             // from the file system

   // major public methods
   public synchronized FileTableEntry falloc( String filename, String mode ) {
      // allocate a new file (structure) table entry for this file name
      // allocate/retrieve and register the corresponding inode using dir
      // increment this inode's count
      // immediately write back this inode to the disk
      // return a reference to this file (structure) table entry
   }
```

```
   }

   public synchronized boolean ffree( FileTableEntry e ) {
      // receive a file table entry reference
      // save the corresponding inode to the disk
      // free this file table entry.
      // return true if this file table entry found in my table
   }

   public synchronized boolean fempty( ) {
      return table.isEmpty( );  // return if table is empty
   }                            // should be called before starting a format
}
```

FileTable.java hosted with ❤ by GitHub

NOTE: `FileStructureTable` is the same thing as `FileTable`. Older documentation (and some files) mention FileStructureTable. For consistency, you can rename ***every*** `FileStructureTable` as `FileTable`, including the `.java` file from Canvas.

File Descriptor Table

Each user thread needs to keep track of all files it has opened. To support this, each thread should maintain a table of those open files in its TCB. Such a table is also known as a user file descriptor table and it has 32 entries. Whenever a thread opens a file, it must allocate a new table entry for the file; this is known as a *file descriptor*. Each file descriptor includes the file access mode and the reference to the corresponding file (structure) table entry.

The file access mode indicates "read only", "write only", "read/write", or "append".

It is possible for a thread to open the same file many times, thus having several entries in the corresponding TCB's user file descriptor table. Each of user file descriptor table entries refer to a different file (structure) table entry with its own seek pointer, all of them ultimately point to the same inode.

Each user thread maintains a user file descriptor table in its own TCB. Every time it opens a file, it allocates a new entry table including a reference to the corresponding file (structure) table entry. Whenever a thread spawns a new child thread, it passes a copy of its TCB to this child which thus has a copy of its parent's user file descriptor table. This in turn means the both the parent and the child refer to the same file (structure) table entries and eventually share the same files.

```
public class TCB {
   private Thread thread = null;
   private int tid = 0;
   private int pid = 0;
   private boolean terminate = false;

   // User file descriptor table:
   // each entry pointing to a file (structure) table entry
   public FileTableEntry[] ftEnt = null;

   public TCB( Thread newThread, int myTid, int parentTid ) {
      thread = newThread;
      tid = myTid;
      pid = parentTid;
      terminated = false;

      // The following code is added for the file system
      ftEnt = new FileTableEntry[32];
      for ( int i = 0; i < 32; i++ )
         ftEnt[i] = null;         // all entries initialized to null
         // fd[0], fd[1], and fd[2] are kept null.
   }
}
```

TCB.java hosted with ❤ by GitHub

Note that the file descriptors 0, 1, and 2 are reserved for the standard input, output, and error, and thus those entries must remain null.

You do not need to implement TCB.java, use the .class provided in the ThreadOS folder.

FileSystem

You must implement the file system as a separate class in its own file. Below is a skeleton showing relevant methods for functionality.

```java
public class FileSystem {
   private SuperBlock superblock;
   private Directory directory;
   private FileTable filetable;

   public FileSystem( int diskBlocks ) {
      // create superblock, and format disk with 64 inodes in default
      superblock = new SuperBlock( diskBlocks );

      // create directory, and register "/" in directory entry 0
      directory = new Directory( superblock.inodeBlocks );

      // file table is created, and store directory in the file table
      filetable = new FileTable( directory );

      // directory reconstruction
      FileTableEntry dirEnt = open( "/", "r" );
      int dirSize = fsize( dirEnt );
      if ( dirSize > 0 ) {
         byte[] dirData = new byte[dirSize];
         read( dirEnt, dirData );
         directory.bytes2directory( dirData );
      }
      close( dirEnt );
   }

   void sync( ) {
   }
   boolean format( int files ) {
   }
   FileTableEntry open( String filename, String mode ) {
   }
   boolean close( FileTableEntry ftEnt ) {
   }
   int fsize( FileTableEntry ftEnt ) {
   }
   int read( FileTableEntry ftEnt, byte[] buffer ) {
   }
   int write( FileTableEntry ftEnt, byte[] buffer ) {
   }
   private boolean deallocAllBlocks( FileTableEntry ftEnt ) {
   }
   boolean delete( String filename ) {
   }

   private final int SEEK_SET = 0;
   private final int SEEK_CUR = 1;
   private final int SEEK_END = 2;

   int seek( FileTableEntry ftEnt, int offset, int whence ) {
   }
}
```

Source with "hint" functions `format` and `close`

FileSystem

The FileSystem needs to be instantiated in `Kernel.java`, as a parameter give the number of files (1000).

```java
// instantiate a file system;
fs = new FileSystem( 1000 );
```

4.3 Testing

Run Test5.class library provided in the *ThreadOS* folder or from the zip in Canvas. If you want to see the source code for this test, you can download it here: ***Test5.java***. This program will test the functionality of every feature and system call listed above and check the data consistency of your file system. Some of the things that are tested include:

1. Does the file system allow one to read the same data from a file that was previously written?
2. Are malicious operations handled appropriately and proper error codes returned? Those operations include file accesses with a negative seek pointer, too many files to be opened at a time per a thread and over the system, etc.
3. Does the file system synchronize all data on memory with disk before a shutdown and reload those data from the disk upon the next boot?

5. What to Turn In

Upload to Canvas:

- **Per team**:
  1. All source code you have modified and created (refer to the Grading Guide for detail on source files).
  2. Report encompassing ***at least***:
     i. Results when you run your own test program, e.g., explain what additional testing you did, how and why
     ii. Your file system specification including your assumptions and limitations
     iii. Descriptions on internal design including inode, file table, etc.
     iv. Consideration on performance estimation, current functionality, and possible extended functionality

     The report should be submitted separately from the code's ZIP file! (see submission guidelines)

     These are the minimum documentation requirements. However, it is expected that you provide reasonable amount of detail as well as thorough explanations regarding your design and implementation. Focus on explaining the *why's*, not just stating the *what's*. Furthermore, your document should be properly formatted and reflect the professionalism expected from a student at your level.

- **Individually**: submit **separately** in Canvas a Word or PDF document describing what you contributed to in the project as well as a description and assessment of what your teammates contributed to. This is CONFIDENTIAL and will not be shared with any other team members. ***You must submit this document in order to receive a grade in the final project!***

Grading guide is the grade guide for the final project.

NOTE: *You have to implement your own FileSystem.java. As you may know, there is a FileSystem.class under the ThreadOS folder, the purpose of this file is to test. The recommended process is that you code everything and test it with the FileSystem.class. Then you implement your own version of FileSystem.*

You also have to implement the SuperBlock, you can use the slides as guidance.

6. FAQ

This website could answer your questions. Please click here before emailing the professor.

7. Additional Materials

Notes.

~~(Typo collection in this pdf: Note that bytes2directory should return **integer** rather than **void**.)~~