Final Project: File System

---

1. Purpose

In this project, you will build a Unix-like file system on our *ThreadOS*. User thread programs will be now relieved from painful direct access to disk blocks and given a vision of stream-oriented files.

2. Collaboration

You may form a team of **two** (or three students if you cannot find a single partner.) **If you work on this project in a team of three, you have to state in your report why you couldn't find a single partner.** If you work with your partner(s), your workload must be 50% or 33% depending on the number of partners, otherwise your report will be severely graded down. This project requires much more time to be completed than the previous four assignments. Schedule and pace your design and coding work accordingly. **If you are very confident of working independently, you may do so and will receive five extra points, however no excuse for a tight schedule for your exam preparation.**

3. What to Use

Use *ThreadOS*' original features, (i.e., the original classes) except *Kernel.java*. For *Kernel.java*, use the one you got in the assignment 4, (i.e., *Kernel_org.java*). To be simple, all you need is re-download all readable files from the *ThreadOS* directory again.

4. Interface

The "/" root directory is only the one predefined by this file system and permanently available for user threads to store their files. No other directories are provided by the system and created by users dynamically. Therefore, all files are maintained in the "/" root directory. The file system must provide user threads with the following system calls that 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.

Each user thread needs to keep track of all files it has opened. For this purpose, it should maintain a table of those open files in its TCB. This table is called a user file descriptor table. It has 32 entries. Whenever a thread opens a file, it must allocate to this file a new table entry, termed 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 first or the tail of the file, and keeps track of a next position to read from and to write to the file. It is entirely possible for one thread to open the same file many times, thus having several entries in the corresponding TCB's user file descriptor table. Although each of those user file descriptor table entries refer to a different file (structure) table entry with its own seek pointer, all of them eventually points to the same inode.

The file system you will implement must provide the following eight system calls.

1. **int SysLib.format(int files);**
   formats the disk, (i.e., *Disk.java*'s data contents). The parameter *files* specifies the maximum number of files to be created, (i.e., the number of inodes to be allocated) in your file system. The return value is 0 on success, otherwise -1.
2. **int fd = SysLib.open(String fileName, String mode);**
   opens the file specified by the *fileName* string in the given *mode* (where "r" = ready only, "w" = write only, "w+" = read/write, "a" = append), and allocates a new file descriptor, *fd* to this file. The file is created if it does not exist in the mode "w", "w+" or "a". *SysLib.open* must return a negative number as an error value if the file does not exist in the mode "r". 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".
3. **int read(int fd, byte buffer[]);**
   reads up to *buffer.length* bytes from the file indicated by *fd*, starting at the position currently pointed to by the seek pointer. If bytes remaining between the current seek pointer and the end of file are less than *buffer.length*, *SysLib.read* reads as many bytes as possible, putting them into the beginning of buffer. It increments the seek pointer by the number of bytes to have been read. The return value is the number of bytes that have been read, or a negative value upon an error.
4. **int write(int fd, byte buffer[]);**
   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. *SysLib.write* increments the seek pointer by the number of bytes to have been written. The return value is the number of bytes that have been written, or a negative value upon an error.
5. **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. 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. In both cases, you should return success.

6. **int close(int fd);**
   closes the file corresponding to *fd*, commits all file transactions on this file, and unregisters *fd* from the user file descriptor table of the calling thread's TCB. The return value is 0 in success, otherwise -1.
7. **int delete(String fileName);**
   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.
8. **int fsize(int fd);**
   returns the size in bytes of the file indicated by *fd*.

Unless specified otherwise, each of the above system calls returns -1 negative when detecting an error.

5. Implementation

Superblock

The disk block 0 is called a *superblock* and used to describe (1) the number of disk blocks, (2) the number of *inode*s, and (3) the block number of the head block of the free list. This is the OS-managed block. No other information must be recorded in and no user threads must be able to get access to the *superblock*.

```
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
}
```

Inodes

After the *superblock*, (i.e., from the second block) are *inode* blocks. Each *inode* describes one file. Our *inode* is a simplified version of the Unix *inode* (as explained in our textbook.) It 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, and (3) the flag to indicate if it is unused (= 0), used(= 1), or in some other status (= 2, 3, 4, ...). 16 *inodes* can be stored in one block.

```
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.
      }
   }
```

You will need another default constructor that retrieves an existing *inode* from the disk onto 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 solutions to maintain the *inode* consistency:

1. Before an *inode* on memory needs to be updated, check the corresponding *inode* in the disk, read it from the disk if the disk has been already updated by another thread. Thereafter, you should 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. You should 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

The "/" root directory maintains each file in a different directory entry that contains its file name (in maximum 30 characters = in max. 60 bytes in Java) and the corresponding *inode* number. The directory receives the maximum number of *inode*s 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 as a file, its contents are maintained by an *inode,* specifically saying *inode* 0. This can be located in the first 32 bytes of the disk block 1.

Upon a boot, the file system instantiates the following *Directory* class as the root directory through its constructor, reads the file from the disk that can be found through the *inode* 0 at 32 bytes of the disk block 1, and initializes the *Directory* instnace with the file contents. On the other hand, prior to a shutdown, the file system must write back the *Directory* information onto the disk. The methods *bytes2directory( )* and *directory2bytes* will initialize the *Directory* instance with a byte array read from the disk and converts the *Directory* instance into a byte array that will be thereafter written back to the disk.

```
   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
      }
   }
```

File (Structure) Table

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

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 an file (structure) table entry which will be obtained from the file system in the following sequence.
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 finally 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
      }
   }
```

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
   }
```

File Descriptor Table

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

6. Statement of Work

Design and implement the file system in *ThreadOS* as specified above. Run *Test5.java* that is found in the *ThreadOS* directory. This test program will test the functionality of every system listed above and check the data consistency and fault tolerantly of your file system:

1. Can you read the same data from a file as you wrote on it previously.
2. Can you avoid malicious operations and return an appropriate error code? 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. Can the file system synchronize all data on memory with disk before a shutdown and reload those data from the disk upon the next boot?

6. What to Turn In

- Hardcopy: all students must turn in their independent report:

1. All source code you have modified and created (may be the same among all team members.)
2. Results when you run your own test program (may be the same among all team members.)
3. Your file system specification including your assumptions and limitations (must be described independently by each student.)

4. Descriptions on internal design including inode, file tabl, etc. (must be described independently by each student.)
5. Consideration on performance estimation, current functionality, and possible extended functionality (must be described independently by each student.)
6. Confidential evaluation of your partner's (or partners') contribution to the project (must be reported independently by each student.)

- Softcopy:

    - Only a representative of each group or each independently-working student must turn in all source code. **Each file must have a heading comment mentioning who coded it.**