Sample

# Detailed Class Specifications

## Directory

### Purpose

* Map file names to Inumber.
* Keeps track of file size and name.
  + Slot 0 -> inode 0 and so on.

### Specifications

* Constructor: gives a ‘blank’ directory object when called (**by thread**); provided by instructor
* bytes2directory: **thread calls this** to give the directory object some amount of bytes to be unpacked into a usable format (i.e., the format internal to directory object)
* directory2bytes: **thread calls this** if it needs to take the data internal to the directory object and pack it into a byte array to hand back as a return value
* Ialloc: ask the directory to **create an inode** for that file and return the # of the inode created
  + If no inodes are available, return an error
* Namei: open a file which already exists, so call namei which checks if there’s a file with that name which has an inode.
  + If the file doesn’t exist, return an error (-1)
* Ifree: delete a file when done. Call and give it the inode to get rid of and return success Boolean.

## FileTable

### Purpose

* The file system maintains the file (structure) table shared among all user threads.
* 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.
* 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.
* The file system locates the corresponding *inode* and records it in this file (structure) table entry.
* The user thread finally registers a reference to this file (structure) table entry in its file descriptor table entry of the TCB.

### Specifications

* **FileTable(Directory directory**);

Instantiate a file table, receive a reference to the directory from file system.

* **void 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.

* **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.

* **boolean fempty();**

Return true if table is empty, false otherwise. Should be called before starting format.

## FileTableEntry

### Purpose

* See FileTable Purpose, specifically the file table entry portions.

### Specifications

* **FileTableEntry(Inode I, short inumber, String m);**

Seek pointer is set to the file top. At least one thread is using this entry. Once access mode is set, it never changes. If mode is append, seekPtr points to the end of file.

## FileSystem

### Purpose

* 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 consiste 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 and created by users dynamically.
* 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 these 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.

### Specifications

* **int format( int files );**  
  Formats the disk (*Disk.java*'s data contents). The parameter *files* specifies the maximum number of files to be created (the number of inodes to be allocated) in your file system. The return value is 0 on success, otherwise -1.
* **int 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). The call 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".
* **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.
* **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.
* **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.

* **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.
* **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.
* **int fsize( int fd );**  
  Returns the size in bytes of the file indicated by *fd*.