**CS350 Program 5 Spring 2017** **Mini File System**

This is a group assignment. You must work in groups of 2, 3, or 4 (preferably 3 or 4). Expectations for all groups (no matter the size) will be the same.

For this assignment, you will implement a flat super simple file system (SSFS) as a multi-threaded program. Some implementation decisions will be made for you and will serve as requirements. Others will be up to you.

**Structure of a File**

The structure of a file will closely resemble that of a unix file. You must support inodes, indirect blocks, and data blocks. Each inode should contain meta information about the file, including (but not necessarily limited to):

- **File Name**: a single string consisting of no more than 32 characters (including extension).
- **File Size**: an integer that holds the number of bytes in the file
- **Direct Block Pointers**: A set of exactly 12 block numbers, which identify the first 12 data blocks of the file.
- **Indirect Block Pointer**: One block number that identifies an indirect block, which contains block numbers of the next *d* data blocks, where *d* is determined by the number of block pointers that fit in the indirect block, as implied by the parameters of the file system (see below).
- **Double Indirect Block Pointer**: One block number of an indirect block, which contains block numbers of *i* indirect blocks, each of which contain block numbers for *d* data blocks. The values of *d* and *i* are determined by the parameters of the file system. (SSFS will not contain a triple indirect block pointer.)

**The "Disk"**

A single real linux file, the SSFS *disk file*, should store all SSFS files and all other information that you need to maintain the file system. The size of the disk file should be configured once at the beginning, in response to a single very simple utility called `ssfs_mkdsk`, and should never grow or shrink throughout the lifetime of your file system. (Disks don't generally grow or shrink!) The usage of ssfs_mkdsk is as follows:

```
ssfs_mkdsk <num blocks> <block size> <disk file name>
```

The `<disk file name>` specifies the name of the disk file that stores all SSFS file system bytes. If this parameter is not present (and for the rest of this description we will assume it is not), then the disk file should be named, simply, "DISK". The `<num blocks>` parameter indicates the number of disk blocks that the DISK comprises. Each disk block should contain `<block size>` bytes. Therefore, the result of running `ssfs_mkdsk` should be a linux file whose size is exactly `<num blocks> * <block size>` bytes. This is easily verified with "`ls –l`". Your file system will use most disk blocks to store inodes, indirect blocks, and data blocks for your users' files. However, you will need a portion of the DISK file to store

- an **inode map**, which will help you locate where all of your inodes are stored, and
- a **free block list**, to indicate which disk blocks are free and which are currently in use. (Note that this does not need to be implemented literally as a "free block list"… you can use any method you wish to distinguish free blocks from in-use blocks… other options include a "used block list", a bitmap, a "byte map", or anything else that you want to use. We will call it a "free block list" within this document.)

Your *inode map* and *free block list* must be stored on the disk within the `<num blocks>` blocks of the disk file, not as extra bytes. Therefore, not all `<num blocks>` blocks are available for use by user files. SSFS will contains a maximum number of files, so the inode map can be a fixed size. The amount of space needed for the free block list is directly determined by the disk size. These two data structures should reside at the top of your DISK file; if they do not fit neatly in an integral number of DISK blocks, that's OK, just leave the last one partially unused.

These two data structures must be stored on the disk because your file system must be able to shut down and restart (potentially multiple times), and correctly retain all of the files that the user has created. You may store a small amount of other information at the top of your DISK file as well (for example, file system parameters). Speaking of which…

**File System Parameters**

All block numbers will be the size of an int (four bytes on the CS lab and remote machines). The file system will hold at most 256 files. If a user tries to create an additional file, your file system should refuse to do so, but should not exit. Instead, the call to create the file should simply return an error code and continue. If files are subsequently deleted, thereby freeing inode numbers, your file system should then be able to create more files, up to the 256 files limit.

- The minimum block size we will use is 128 bytes. This will allow all of the information required of an inode (as specified above) to fit within a single block. The maximum block size will be 512.
- The DISK will consist of at least 1024 blocks and at most 128K blocks.
- The block size and the number of blocks in the DISK will both be powers of 2.

If the user specifies an illegal block size when running `ssfs_mkdsk`, SSFS should print an error message and exit. Otherwise, your file system program should configure the DISK file as specified, and store the number of blocks and block size at the top of the DISK. When the system is shut down and restarted, these parameters can be read from the disk to help reestablish in-memory data structures for SSFS.

**Multi-Threading**

Your file system must be implemented as a multi-threaded program (using pthreads).

You should start exactly one *Disk Controller* thread, and up to 4 *Disk Op* threads. The Disk Controller is the only thread that is allowed to read and write the DISK file. That thread receives requests (to read or write disk blocks) from the SSFS *Disk Op* threads within a buffer of block requests. The Disk Controller thread therefore acts like a consumer in the Producer Consumer bounded buffer problem that we studied in the Concurrency portion of the course. The Disk Controller thread receives requests to read or write to the DISK file by reading an item/request out of a shared buffer, and acting on that request. Once it has done so, it makes the result (perhaps the bytes of a disk block, in response to a read request) available to the requesting thread (you decide how), causes the appropriate Disk Op thread to be woken up (you decide how), and moves on to process the next request. Process disk requests in FCFS order; do not bother with SCAN or other disk scheduling algorithms.

*Disk Op* threads respond to sequences of file operations, contained in files (one file per *Disk Op* thread). These files contain a list of "commands" to SSFS, as follows:

```
CREATE <SSFS file name>
```

This command causes SSFS to create a new file named <SSFS file name>. If <SSFS file name> exists, the command should fail. SSFS does not support directories/folders, so all file names should be unique.

IMPORT <SSFS file name> <unix file name>

This command causes the contents of <SSFS file name>, if any, to be overwritten by the contents of the linux file (in the CS file system) named <unix file name>. If a file named <SSFS file name> does not yet exist in SSFS, then a new one should be created to contain the contents of <unix file name>.

CAT <SSFS file name>

This command displays the contents of <SSFS file name> on the screen, just like the unix cat command.

DELETE <SSFS file name>

Remove the file named <SSFS file name> from SSFS, and free all of its blocks, including the inode, all data blocks, and all indirect blocks associated with the file.

WRITE <SSFS file name> <char> <start byte> <num bytes>

Write <num bytes> copies of character <char> into file <SSFS file name>, beginning at byte <start byte>. If there are fewer than <start byte> bytes in the file, the command should report an error. If there are fewer than <start byte> + <num bytes> bytes in the file before the command runs, the file should be appended to make room for the extra characters. If not enough free file blocks exist to complete the WRITE command, you may either write as many bytes as you can and then abort the command, or just not carry out the command, instead returning an error message. (You may choose whichever of those two options you prefer.)

READ <SSFS file name> <start byte> <num bytes>

This command should display <num bytes> of file <SSFS file name>, starting at byte <start byte>.

LIST

List all the names and sizes of the files in SSFS.

SHUTDOWN

Closes the DISK file after the *Disk Controller* thread has finished servicing all pending requests, and exits the `ssfs` process.

---

SSFS should be invoked with a single command named `ssfs`, which will take the disk file name and one, two, three, or four other file names as command line arguments. For example:

```
ssfs <disk file name> thread1ops.txt thread2ops.txt thread3ops.txt
```

Upon startup, your file system program should look for a file named `<disk file name>` in the current directory and read the status of the current file system from that file. For full credit, you should allow multiple threads to execute multiple disk operation sequences concurrently, in different threads. The last three arguments imply three different *Disk Op* threads, which read their commands from the three named files (one per thread).

You should implement a library of functions to carry out the operations described above. The interface to that library is up to you.