McGill

Programming Assignment #3: Simple Shadow File System

Due date: Check My Courses`

1. New simplifications to the assignment

The following simplifications should override the original assignment.

- Maximum of ten characters for the filename
- Directory entry limited to 16 bytes in length (or fit within 16 bytes)
- Limit the number of data blocks in the file system to 1024. Note: superblock, FBM, and WM are not included in this. So your actual disk file will include them. Alternatively, you could use 1021 for the data blocks.
- The reason I want to limit to 1024 is to compensate for an oversight in my part. It seems the FBM and WM have to be versioned as well! You can follow a much simpler approach now. Only use the least significant bit to indicate whether a block is used or unused. Since you have 8 bits you can keep an history of 8 versions in FBM and WM.

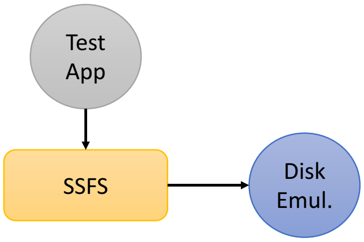
Here is the drastic simplification:
I am ***removing*** the commit/restore part from the assignment. I am putting it in the extra credit part. You may complete that for additional 5 points. This could be getting you a letter grade bump in the course.

2. What is required as part of this assignment?

In this assignment, you are expected to design and implement a **Simple Shadow File System** (SSFS). **You need to demonstrate the file system working in Linux**. The SSFS introduces many limitations such as restricted filename lengths, no user concept, no protection among files, no support for concurrent access, etc. You could introduce additional restrictions in your design. However, such restrictions **should be reasonable to not oversimplify the implementation and should be documented in your submission**. Here is a list of restrictions of the SSFS:

- Limited length filenames (select an upper limit such as 16)
- No multi-user access or file protection support
- No subdirectories (only a single root directory – this is a severe restriction – relaxing this would enable your file system to run many applications)
- Your file system is implemented over an emulated disk system, which is provided to you.

Here is a schematic that illustrates the overall concept of the SSFS. It is completely in the user space. mountable simple file system.



You are expected to develop the yellow colored module. Other modules are provided to you. You may write your own simpler test applications while developing the SSFS module.

3. Objectives in detail

The file system you implement will be tested by compiling and linking with a test suite consisting of one or more testing programs. The testing programs will invoke the APIs provided by your file system to create, read, and write file data to the emulated disk. The testing programs will perform various operations and count the number of violations of the file integrity constraints. Following are some example integrity constraints: able to read all written data to a file, all space held by a deleted file added to the free list, etc. Ideally, you should debug your implementation to reach zero errors as reported by the test suite. Even in that state your implementation can have many more errors that are not detected by our test suite. We will **not evaluate your implementation using a private test suite** that is not released to you.

You are expected to support the following API in your SSFS implementation. In SSFS, a file has two independent pointers: one for reading and another for writing. When a file is opened (we have only one mode for opening a file: read and write), the **read pointer** is at the beginning of the file and **write pointer** is at the end of the file. That is, the file is setup for appending data to it. If the file is new (i.e., zero data in it), both pointers are at the beginning of the file. As you write data to the file, only the write pointer moves. As you read, only the read pointer moves. So we can read a file sequentially, without manipulating (or repositioning) the read pointer. Similarly, we can just append to a file without manipulating the write pointer. By manipulating the write pointer, we could write data in other places like overwriting some existing data.

The C based API for the SSFS is given below. Your implementation should adhere to this one to compile with the testing program.

```
void mkssfs(int fresh);                   // creates the file system

int ssfs_fopen(char *name);               // opens the given file
int ssfs_fclose(int fileID);              // closes the given file
int ssfs_frseek(int fileID,
        int loc);                         // seek (Read) to the location from beginning
int ssfs_fwseek(int fileID,
        int loc);                         // seek (Write) to the location from beginning
int ssfs_fwrite(int fileID,
        char *buf, int length);           // write buf characters into disk
int ssfs_fread(int fileID,
        char *buf, int length);           // read characters from disk into buf
int ssfs_remove(char *file);              // removes a file from the filesystem

int ssfs_commit();                        // create a shadow of the file system
int ssfs_restore(int cnum);               // restore the file system to a previous shadow
```

The `mkssfs()` formats the virtual disk implemented by the disk emulator and creates an instance of SSFS file system on top of it. The `mkssfs()` has a fresh flag to signal that the file system should be created from scratch. If flag is false, the file system is opened from the disk (i.e., we assume that a valid file system is already there in the file system. The support for persistence is important so you can reuse existing data or create a new file system.

The `ssfs_fopen()` opens a file and returns an integer that corresponds to the index of the entry for the newly opened file in the file descriptor table. If the file does not exist, it creates a new file and sets its size to 0. If the file exists, the file is opened in append mode (i.e., set the write file pointer to the end of the file and read at the beginning of the file). The `ssfs_fclose()` closes a file, i.e., removes the entry from the open file descriptor table. On success, `ssfs_fclose()` should return 0 and a negative value otherwise. The `ssfs_fwrite()` writes the given number of bytes of buffered data in `buf` into the open file, starting from the current write file pointer. This in effect could increase the size of the file by the given number of bytes (it may not increase the file size by the number of bytes written if the write pointer is located at a location other than the end of the file). The `ssfs_fwrite()` should return the number of bytes written. The `ssfs_fread()` follows a similar behavior. The `ssfs_rfseek()` moves the read pointer and `ssfs_wfseek()` moves the write pointer to the given location. It returns 0 on success and a negative integer value otherwise. The `ssfs_remove()` removes the file from the directory entry, releases the *i*-node entry and releases the data blocks used by the file, so that they can be used by new files in the future.

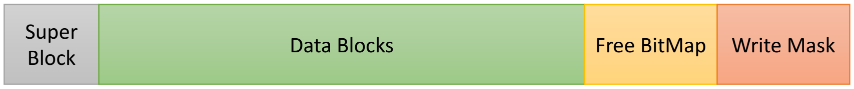
A file system is somewhat different from other components because it maintains data structures in memory as well as on disk! The disk data structures are important to manage the space on disk and allocate and de-allocate the disk space in an intelligent manner. Also, the disk data structures indicate where a file is allocated. This information is necessary to access the file.

One of the unique features of SSFS is the shadowing feature, which can be used to create snapshots of the file system so that users can undo the changes made to the file system. For example, files can be undeleted with shadowing. Also, shadowing can help in the crash recovery process.

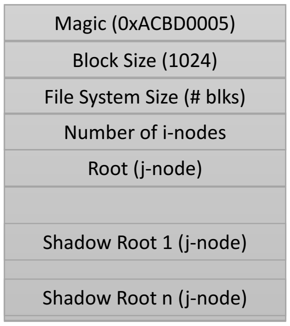
4. Implementation strategy

The disk emulator given to you provides a constant-cost disk (CCdisk). This CCdisk can be considered as an array of blocks (sectors of fixed size). You can randomly access any given block for reading or writing. The CCdisk is implemented as a file on your computer's file system. Therefore, the data stored in the CCdisk is persistent across program invocations. To mimic the real disk, the CCdisk is divided into blocks of fixed size. For example, we can split the space into 1024 byte blocks. The number of blocks times the size of a block gives the total size of the disk. In addition to holding the actual file and directory data, we need to store auxiliary data (meta data) that describes the files and directories in the disk. The structure and number of bytes spent on meta data storage depends on the file system design, which is the concern in this assignment.

On-disk data structures of the file system include a super block, free bitmap block, write mask block, and data blocks. The figure below shows a schematic of the on-disk organization of the SSFS.



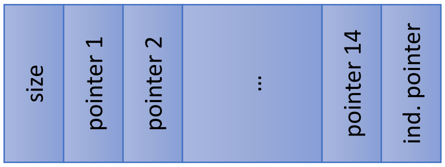
The super block, which is the first block in the file system, defines the file system geometry. So the super block needs to have some form of identification to inform the program what type of file system format is followed for storing the data. The magic value in the super block achieves this purpose. The figure below shows the proposed structure for the super block. Each field in the figure below is 4 bytes long. For instance, the magic number field is 4 bytes long. Following the magic number is the block size of the file system. It is recommended that you use 1024 bytes. With 1024-byte sized blocks, the super block can have plenty of unused space. Other pieces of information held in the super block include: file system size (in blocks), total number of *i*-nodes, and the roots of the file systems. The roots themselves are *j*-nodes (they have the exact same structure as the simplified *i*-node shown below).



The root (j-node) points to several data blocks that constitute a file that holds all the i-nodes. This is major difference between SSFS and the UNIX file system you saw in the class. Unlike a UNIX file system, where the i-nodes are written directly onto the disk after the super block, SSFS puts the i-nodes in a file and points to the blocks holding the file using a j-node. The j-node is held in the super block. The super block has provision to hold one "root" j-node and several shadow roots (themselves j-nodes). We will see why the shadow root nodes are required very soon.

In the suggested SSFS design, we have a single root directory. So only i-node will be taken up by the directory. Suppose we have 200 i-nodes in the file system, we can have a maximum of 199 files in the file system. To hold the 200 i-nodes, we need 200*64 bytes = 12800 bytes = 12800/1024 blocks = 13 blocks. So we can just use the direct pointers in the i-nodes (i.e., no need to use the indirect blocks).

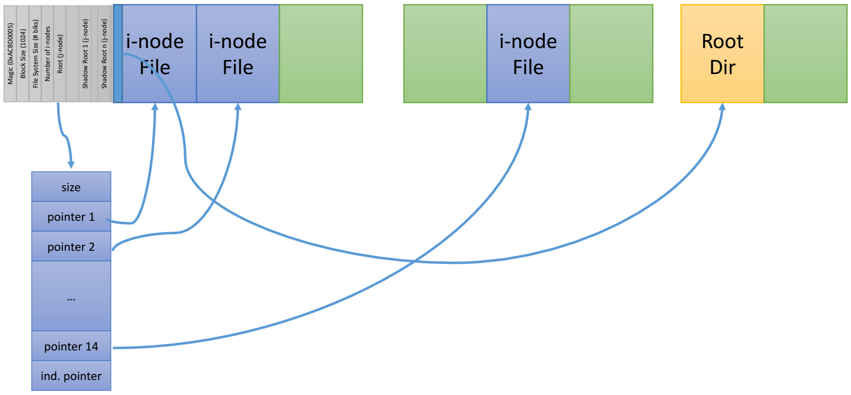
The figure below shows a simplified i-node structure. You can notice that it does not have double and triple indirect pointers. Nor does it have meta information such as user ID (uid). Assuming each field is 4 bytes long, the i-node is 64-bytes long. The j-nodes have the same structure.



The Free Bit Map (FBM) can be used to determine the locations of the data blocks that are available for allocation. That is unused blocks. As discussed in the class, one bit is used to denote the used/unused status of a data block (bit **value 1 indicates the data block is unused** and 0 indicated used). The FBM only tracks the data blocks. That is the data blocks allocated for the super block, FBM, and Write Mask (WM) are not tracked by the FBM. With 1024 byte blocks, by allocating a single data block for FBM, we can track the status of the 8 * 1024 blocks = 8192 blocks. Therefore, for SSFS, FBM can be restricted to one block. The WM is made equal to 1 block as well. So the SSFS file system will have one data block each for super block, FBM, and WM. Other data blocks will hold the i-nodes (in files), actual file data, and directory data.

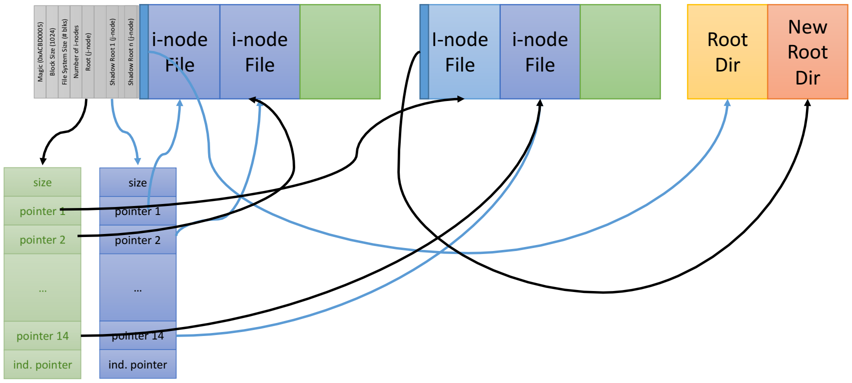
The Write Mask (WM) is structured exactly like the FBM. Instead of tracking whether a block is used/unused as the FBM does, the WM is responsible for tracking whether a block is writeable or not. If the **write mask bit is 1 the corresponding data block is writeable**. Otherwise, the data block tracked by the bit is read only.

The SSFS file system is initialized by the `mkssfs()` function. It does the following: setup the super block, setup the FBM, setup the WM, and initialize the file containing all the i-nodes, setup the root directory. A part of the file system state after initialization is shown in the figure below. The figure does not show the FBM and WM. The FBM will have the bits corresponding to the blue blocks as used and the rest as unused. Similarly, the WM will have the bits corresponding to the blue blocks as read-only.



Also, at startup, the root directory will be created. The one of the i-nodes inside the blue blocks (all the blue blocks constitute a single file) points to the root directory. The block allocated to the root directory is shown in orange in the above diagram. Of course, the FBM and WM are modified such that the orange block is used and read-only, respectively.

Suppose we want to create a file and write some data into the file. First, we need to create a directory entry by modifying the root directory. The problem is that the data block holding the root directory is marked as read-only. So we need to copy the block and modify the new copy. However, the i-node that corresponds to the root directory is pointing towards the orange block. Therefore, we need modify the i-node, which is in a read-only block as well. Again, we copy that block and the i-node inside the new block can be modified to point towards the new root directory. Now, we need to modify the root so that the data block containing the modified i-node is pointed to by the root. Luckily, the super block is not read-only, so we can modify it. However, we want to keep the original root so we can revert back to the original tree. We copy the root to one of the available shadow roots. The configuration of the file system after the new root directory creation is shown below.



Suppose we want to add another file to the file system, it is quite easy. The directory block is writeable now. We assume that the root directory will be held within a single data block so the newly copied block is sufficient and will take the modifications necessary for the addition of the new file. The new file needs an i-node and its number needs to inserted into the root directory. You need to scan the i-node file to detect the first empty i-node. You will notice that the i-node as shown above does not have a free/busy flag. You can use the size field for that purpose. A size of -1 for example could be used to denote an unused i-node. Otherwise, you need to make room for a full/empty flag in the i-node. In this assignment, you can easily reduce the size of a data block pointer from 4 bytes to 2 bytes and make room within the i-node.

You will notice that the **`mkssfs()`** left the file system in a read-only state and all modifications to the file system needed new data block copies. The newly created block copies remain writeable until we call

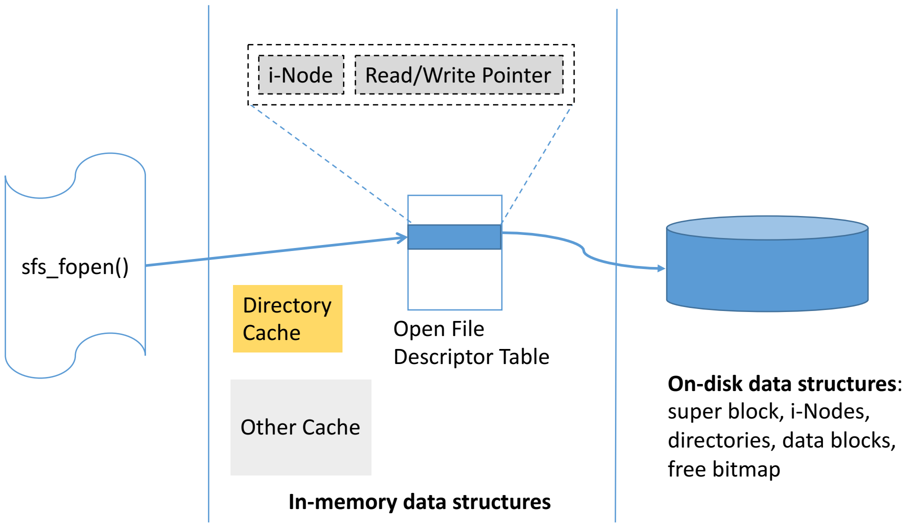
ssfs_commit(). At that point, the newly added blocks become read-only. The **ssfs_commit()** returns index of the shadow root that holds previous commit. Making the newly copied blocks read-only is simple; you need to merely copy the FBM into WM. To revert the file system to the previous state, we run **ssfs_restore()** with the appropriate commit number. This command restores the file system state to the previous state (before any modifications) by simply copying the shadow root to root.

In SSFS, super block, FBM, WM, and i-nodes are on-disk data structures created to store data in the disk. That is, the disk becomes usable only creating and initializing these data structures in the proper way. In addition to maintaining these data structures properly, we need to maintain some in-memory data structures when the file system is used by applications. Note that all transactions with the disk has to be at the block level: that is reads and writes are at the block level. Therefore, we need a cache to hold at least the current block so that it can be modified and written back.

Because the root directory is a highly accessed disk block you may want to keep it in memory to make disk transactions easy. Similarly, the FBM and WM are good candidates to cache in memory. The figure below shows the different in-memory data structures that you may want to have and how they connect to the other components. We need at least a table that combines the open file descriptor tables (the per-process one and system-wide one) in a UNIX-like operating system. We simplify the situation because we assume that only one process is accessing a file at any given time.

When a file is opened we create an entry in this table. The index of the newly created entry is the "file descriptor" that is return by the file opening activity. That is the return value of the **ssfs_fopen()** is this index. The entry created in the file descriptor table has at least three pieces of important information: i-node number, read pointer, and write pointer. The i-node number is the one that corresponds to the file. Remember just like there is an i-node for the root directory, there is one i-node associated with each file. When a file is opened that i-node is copied into this table entry. The read and write pointers are also set as specified earlier.

In this assignment, the file system performance is not a concern – correct operation is what we need. You could tailor the in-memory data structures to suit the reasonable simplifications you make.


i-Node
Read/Write Pointer
sfs_fopen()
Directory Cache
Open File Descriptor Table
Other Cache
In-memory data structures
On-disk data structures:
super block, i-Nodes, directories, data blocks, free bitmap

<u>Suggested procedure (may be incomplete) for creating a file:</u>

1. **Allocate and initialize an i-node**. This will involve scanning through the i-node file that is pointed to by the root. Finding an empty i-node (an i-node with size -1) and setting its size to 0. Setting the size to 0 would indicate that it is now used. You store this i-node number in a newly created root directory entry that associates the i-node with the new file. As explained before, modifying the i-node and root directory would lead to data block copying because the blocks may be read-only.
2. **Setup the open file descriptor table**. An open file descriptor (OFD) table entry needs to be created for the new file. The OFD table is in memory and can be modified at will. You need copy the i-node and set the read and write pointers in this table. The index of the newly created entry in the OFD table is the return value of the `ssfs_fopen()` function.

Suggested procedure (may be incomplete) for writing a file:

1. **Get block to write**. There could be different writing scenarios: writing to a newly created file, writing to an old file where there is room in the last block, and writing to an old where there is no room in the last block. Read the last block of the file into a block cache in memory. You can find the last block using the i-node and the write pointer. If the write pointer is not at the end of the file, we are not writing to the last block, instead we are writing to an arbitrary location pointed to by the write pointer.
2. **Write the block**. Writing the block can involve copying existing data blocks. For example, when we write data into a newly created file, the data would be written into a new data block so there is no copying of an existing block. Once the data is written, we need to update the i-node. If we have already made a writeable copy of the i-node, then we can just write that block without making another copy. *Note that the data block copy is only done when the block to be written is tagged as read-only.*

Suggested procedure for seeking on a file:

1. Modify the read and write pointers in memory. There is nothing to be done on disk!

5. Testing and Evaluation

We will provide you with a tester for the file system. You could write your own testers and submit with the assignment. If our testers are not working with your assignment and you have substituted with your own testers, they should be equivalent in functionality to the supplied testers.

The programming assignment should be submitted via My Courses. Other submissions (including email) are not acceptable. **Your programming assignment should compile and run in Linux. Otherwise, the TAs can refuse to grade them**. Here is the mark distribution for the different components of the assignment.

Revised evaluation grid

Illustrated design (pseudo code, UMLs, your favorite strategy) This step may be skipped if you have the full implementation. For partial grade this is essential!	30%

Function skeletons Connect the function skeletons to the design	15%
File system compiling and partially working File creation, read/write working, able to retrieve previously written data	20%
Random reads and writes working with seeking, deletes working, file persistence	20%
Reads and writes working with seeking	10%
Memory leak problems	5%
Code quality and general documentation (make grading a pleasant exercise)	10%
Commit and restore working (extra credit)	**30%**