Programming Assignment #4: Simple File System

What is required as part of this assignment?

As part of this assignment, you are expected to design and implement a *simple file system* (SFS). The simple file system handles a single application at any given time, it implements no user concept, does not support protection among files. Although these assumptions are quite dramatic, it leaves the file system still usable in single-tasking environments such as digital cameras and other "embedded" environments. Also, you will implement a simplified interface to the file system with notable restrictions such as: limited length filenames, limited length file extensions, no subdirectories (i.e., only a single "root" directory), few file attributes such as size, creation or last modified date, and no file permissions. A library of C functions will emulate the disk system. This library is provided to you.

Objectives in detail

Your SFS should implement the following *application programming interface* (API). The C language based API should look like the following. If you modify this API, your API should be a superset of the given API (i.e., should provide more functionality).

```
void mksfs(int fresh);          // creates the file system
void sfs_ls();                  // lists files in the root directory
int sfs_fopen(char *name);      // opens the given file
void sfs_fclose(int fileID);    // closes the given file
void sfs_fwrite(int fileID,
  char *buf, int length);       // write buf characters into disk
void sfs_fread(int fileID,
  char *buf, int length);       // read characters from disk into buf
void sfs_fseek(int fileID,
  int loc);                     // seek to the location from beginning
int sfs_remove(char *file);     // removes a file from the filesystem
```

Try to make your file system as simple as possible. You need not implement support for hierarchy of directories. All the files will be in a single root directory. The `mksfs()` - will format the virtual disk for your own file system, i.e., create necessary disk resident data structures and initialize them. The `mksfs()` 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 filesystem. This persistence is IMPORTANT so you reuse existing data or create a new file sytem. The `sfs_ls()` - will list the contents of the directory in details, i.e., including the information stored in the file control blocks. The `sfs_fopen()` - opens a file and returns the index on the file descriptor table. If file does not

exist, create the new file and set size to 0. If file exists, open the file in append mode. The `sfs_fclose()` - closes a file, i.e., removes the entry from the open file descriptor table. The `sfs_fwrite()` - writes length bytes of buffered data in `buf` onto the open file, starting from the current file pointer. This in effect increases the size of the file by "length" bytes. The `sfs_remove()` - will remove the file from the directory entry as well as release the file allocation table entries and data blocks used by the file, so that they can be used by new files in future.
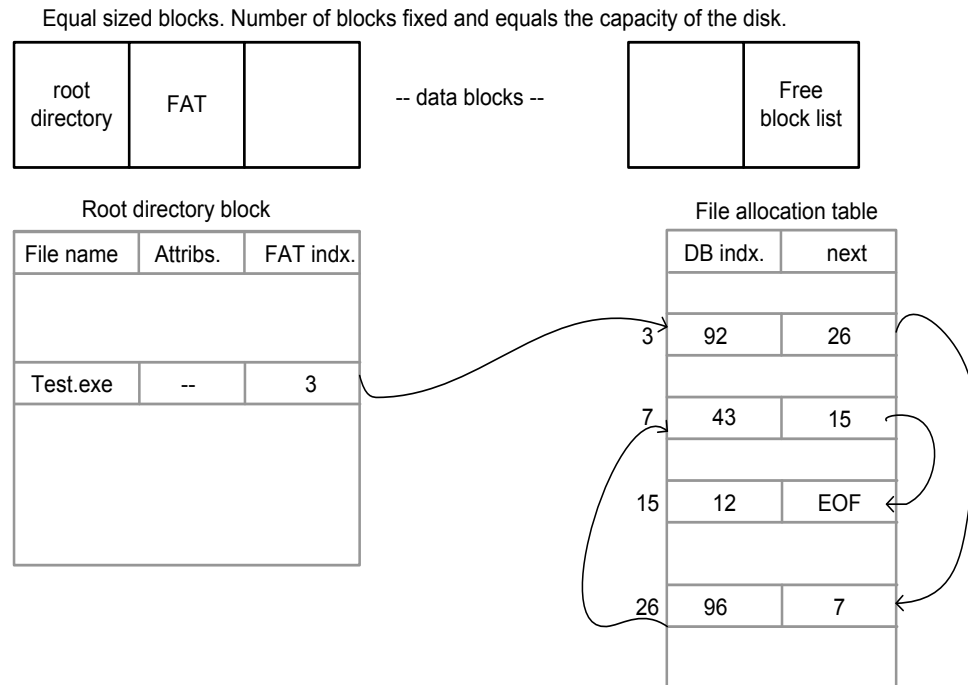
A file system is somewhat different from other components because it maintains data structures in **memory as well as disk**! The disk data structures are important to manage the space in disk and allocate and de-allocate the disk space in an intelligent manner. Also, the disk data structures indicate where a file is allocated. This information is necessary to access the file.

Implementation strategy

The disk emulator given to you provides a constant-cost disk (CCdisk). This CCdisk can be considered as an array of sectors (blocks of fixed size). You can randomly access any given sector for reading or writing. The CCdisk is implemented as a file on the actual file system. Therefore, the data you store in the CCdisk is persistent across program invocations. Let your CCdisk have *N* disk sectors with each sector having a size of *M* bytes. The disk space should be used to allocate disk data structures of the file system as well the files. On disk data structures of the file system include the root directory, free sector list, file allocation table. You can make some simplifying assumptions here and place these data structures in fixed sectors of the disk. For example, sector 0 can be the root directory, sector 1 can be the file allocation table, and sector *N-1* (last sector) can be the free sector list. Within a sector you can use fixed length data structures to store control data.

Files are identified by human readable "file names." These are strings formed by the user that conform to the file system conventions. You can make up reasonable conventions for the SFS regarding names. A directory is a table that maps these names to data block locations. The data block locations are defined by the *file allocation table* (FAT). Therefore, the directory entry need not specify all the data block locations. Instead, it just points to the FAT table entry that defines the data block mappings for the file. Each FAT entry specifies the location of a single data block. This means a file needs multiple FAT entries to completely specify its mapping on the disk. To implement this requirement, the FAT entries can be organized in a linked list. It is important to realize that the FAT table is implemented in disk NOT memory. Therefore, ordinary C pointers cannot be used to implement the list in FAT. Instead you should use FAT indexes.

Figure 1 shows an example set of on disk data structures for implementing the SFS. The figure shows the allocation for an example file `Test.exe`. In this case, the first FAT entry for `Test.exe` is 3. It points to data block 92, which holds the first portion of `Test.exe`. Suppose a data block is 1000 bytes. Bytes 0 to 999 of `Test.exe` will be found in data block 92. The last data block (data block 12) may not be fully populated with `Test.exe`'s data. This can be determined using the size attribute. The contents of `Test.exe` are held in blocks 92, 96, 43, and 12 (in that order).

Equal sized blocks. Number of blocks fixed and equals the capacity of the disk.

root directory	FAT		-- data blocks --		Free block list

Root directory block

File name	Attribs.	FAT indx.
Test.exe	--	3

File allocation table

|	DB indx.	next
3	92	26
7	43	15
15	12	EOF
26	96	7

Figure 1: On-disk data structures of the file system.

In addition to the **on-disk** data structures, we need a set of **in-memor**y data structures to implement the file system. The in-memory data structures improve the performance of the file system by caching the on disk information in memory. Two data structures should be used in this assignment: directory table and file descriptor table(s). The directory table keeps a copy of the directory block in memory. When you want to create, delete, read, or write a file, first operation is to find the appropriate directory entry. Therefore, directory table is a highly accessed data structure and is a good candidate to keep in memory. Another data structure to cache in the memory is the free block list. See the class notes for different implementation strategies for the free block list.

Figure 2 shows an example set of in-memory data structures. The open file descriptor table(s) can be implemented in two different ways. You can have a process specific one and a system-wide one. This is more general and closely follows the UNIX implementation. You can simplify the situation and have only the table – this is reasonable because we assume that only process is accessing a file at any given time (i.e., no simultaneous access to a single file by multiple processes).

As shown in Figure 2, the entry in the file descriptor table can be used to provide some information regarding the reading and writing locations. The mandatory information is the FAT root for the file. For example in the previous example the FAT root is 3 for **Test.exe**. When a file is written to, the write pointer moves by the amount of bytes that is written onto the file. Normally, the write pointer would always point towards the end of the file unless it is explicitly manipulated to point elsewhere (for example, using a **seek()** routine in C/UNIX). The **sfs_fseek()** function modifies the read and write pointers. In SFS, both pointers are set my a single invocation of the **sfs_fseek()** function. Subsequent **sfs_fread()** calls and **sfs_fwrite()** calls change the read and write pointers independently. You should think of how to implement the read and write pointers. Please note you are required to write data in arbitrary

length chunks onto the file. In addition to these data structures, we can have caching structures for FAT and directory blocks.

Possible structure for a file descriptor table entry

| Root FAT | Write ptr. | Read ptr. |
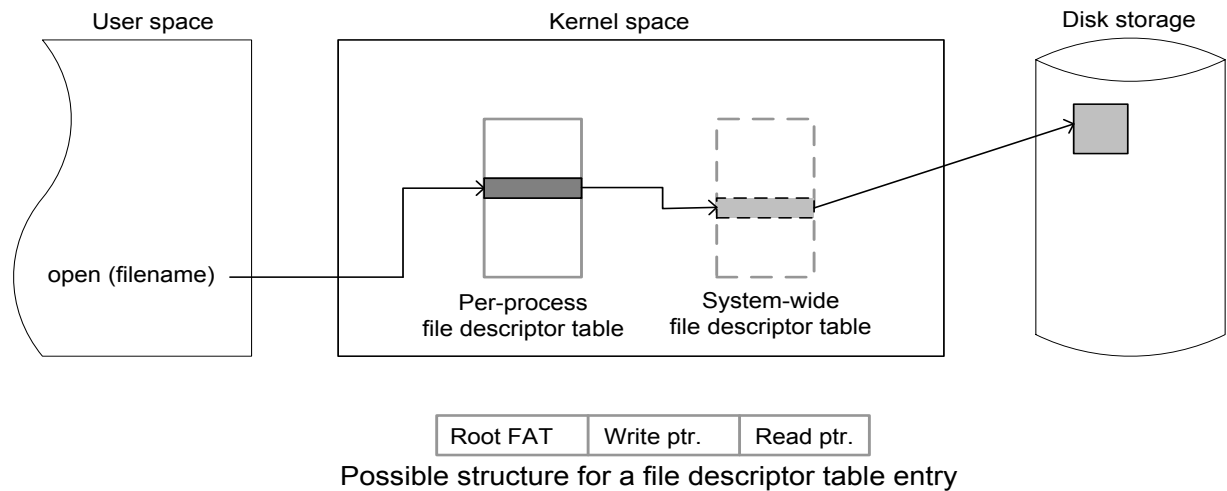
Figure 2: In-memory data structures for the file system.

Following are some of the main operations supported by the filesystem: creating a file, growing a file, shrinking a file, removing a file, and directory modifications.

To create file:
1. Allocate and initialize an FAT node.
2. Write the mapping between the FAT node and file name in the root directory.
3. Write this information to disk.
4. No disk data block allocated. File size is set to 0.
5. This can also "open" the file for transactions (read and write). Note that the SFS API does not have a separate create() call. So you can do this activity as part of the open() call.

To grow a file:
1. Allocate disk blocks (mark them as allocated in your free block list).
2. Modify the file's FAT node to point to these blocks.
3. Write the data the user gives to these blocks.
4. Flush all modifications to disk.
5. Note that all writes to disk are at block sizes. If you are writing few bytes into a file, this might actually end up writing a block to next. So if you are writing to an existing file, it is important you read the last block and set the write pointer to the end of file. The bytes you want to write goes to the end of the previous bytes that are already part of the file. After you have written the bytes, you flush the block to the disk.

To shrink a file:
1. Remove pointers to the disk blocks from the FAT node of the file.
2. Mark the disk blocks as free.

To seek on a file:

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

To read from a file:
2. Remove pointers to the disk blocks from the FAT node of the file.
3. Mark the disk blocks as free.

What to Hand In

Submit the following files separately:

1. A README file with:
   1. Your name and McGill ID number
   2. If you have not fully implemented all, then list the parts that work so that you can be sure to receive credit for the parts you do have working. Indicate any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit.
2. All source files needed to compile, run and test your code. If multiple source files are present, provide a Makefile for compiling them. Do not submit object or executable files.
3. Output from your testing of your program.

Suggestions

1. Make sure your source code could compile in Linux. Test them on the Trottier lab machines.

Part of the grading scheme you might want to know (to keep the TA happy):

1. 0 points if it does not compile.
2. A README text file must come with the assignment, especially if it does not implement all the features or has some known errors. (In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit. On the other hand, it also may lead the TA to notice something that otherwise he or she would not.). Otherwise, the highest mark you can get is half of the total.
3. Name your parameters and functions properly by following one of the naming rules, such as Camel, Pascal, and even Hungarian notation (not recommended), to make your program clear enough to readers. (It could be you in 6 months later.). Otherwise, the highest mark you can get is half of the total.
4. Proper comments should be written in your program, unless your program is highly well-organized. Otherwise, the highest mark you can get is half of the total.
5. Use indentation correctly in your program, otherwise you will lose 20 percents out of 100.