# Programming Assignment #2: Simple File System

COMP310/ECSE427 Winter 2008

Due: See WebCT

## 1 Goals of this assignment

The file system is one of the most important portions of an operating system, and it is used by all applications. There is a wide variety of file system implementations in use, from relatively simple devices such as digital cameras, cell phones, and MP3 players, to complex file systems that run on networked servers and highly-redundant storage devices (e.g. RAID devices and tape robots). This project will introduce you to a simple file system, which will introduce you to the design issues faced by any file system implementation.

## 2 Requirements of this assignment

You are expected to design and implement a "Simple File System" suitable for use on a stand-alone machine in a single-process, single-user environment. This simplifies the implementation, but is still useful on single-tasking "embedded" devices such as phones or cameras. We will also allow restrictions such as

- Limiting the length of file names and extensions
- Single-level hierarchy (no subdirectories required)
- A limited set of attributes (file size, date, etc.)
- No security or user permissions

## 3 Detailed requirements and specifications

This section describes the requirements of the assignment in detail. The disk system will be emulated using the library of C functions you developed for programming assignment 1.

A file system is somewhat different from other operating system components in that it maintains data structures both in memory and on disk. The disk data structures are important, in that they manage the space on the disk and help allocate the space in an intelligent manner. These disk data structures also indicate which blocks on the disk are assigned to a particular file.

## 3.1 Programming interface

Your implementation should include the following functions, expressed here in the C language:

```
int mksfs(int fresh);      // Create and initialize the file system
void sfs_ls();              // List the files in the root directory
int sfs_open(char *name); // Open the given file name
int sfs_close(int fd);   // Close the given file handle
int sfs_write(int fd, char *buf, int length); // Write bytes to file
int sfs_read(int fd, char *buf, int length); // Read bytes from file
int sfs_remove(char *name); // Delete a file name from the system
```

You may implement additional functions, or enhanced versions of these functions, as you see fit. However, try to make the system as simple as possible. As we mentioned, you need not implement a hierarchy of directories, instead maintaining a single root directory.

### 3.1.1 `mksfs()`

This function will format the emulated disk for your own file system creating and initializing whatever necessary data structures are required on the disk. If the `fresh` parameter is non-zero, the system should be created from scratch. If `fresh` is zero, the existing file system is just opened and initialized based on the current virtual disk contents (this assumes there is already a valid filesystem on the disk). This persistence is *important*.

The return value should be zero if the function succeeds.

### 3.1.2 `sfs_ls()`

This function will simply list the contents of the directory as well as whatever details and attributes are appropriate for each file name.

There is no return value.

### 3.1.3 `sfs_open()`

This function opens a file and returns an index into the file descriptor table. If the file does not exist, it creates a new file and sets the initial size to zero. If the file does exist, the file will be opened in "append" mode.

If an error occurs, the return value should be less than zero.

### 3.1.4 `sfs_close()`

This function closes an existing file handle and removes the entry from the file descriptor table. The return value should be zero if the function succeeds.

### 3.1.5 `sfs_write()`

This function writes `length` bytes from buffer `buf` into the open file specified, starting from the current write offset. This will increase the size of the file by `length` bytes.

This function should return the number of bytes actually written to the file, or a negative value if an error occurs.

### 3.1.6 `sfs_read()`

This function reads at most `length` bytes from the file, copying the data to the memory pointed to by the `buf` argument, starting at the current read offset.

The function should return the number of bytes actually read, which may be less than the number requested, or a negative value if an error occurs.

### 3.1.7 `sfs_remove()`

This function removes the named file from the directory, and marks any data blocks used by the file as available for use by other files in the future.

The return value should be zero if the operation succeeds.

## 4 Implementation strategy

The disk emulator from programming assignment one provides an interface to an abstract disk. This can be treated as an array of sectors or blocks of fixed size. You can randomly access any sector for reading or writing. This abstract disk is implemented as a file on the real disk (and real file system). Therefore, the data you store in the abstract disk is persistent across program invocations. Let your abstract disk have $N$ disk sectors with each sector having a size of $M$ bytes. The disk space must be used to store file system data structures as well as the data in the files themselves. On-disk data structures of the file system will include the root directory, free sector list, and file allocation table. One simplifying assumption you can make is to create these structures with a fixed size and position on the disk. For example, the first sector might contain the root directory, the second sector might contain the file allocation table, and the last sector might 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's conventions. You can make up reasonable conventions for your names, but please allow at least 12 alphanumeric characters per file name.

A directory is a table that maps file names to data block locations. However, the full mapping from the file to data block locations is defined in the *file allocation table* (FAT). Each FAT entry specifies the location of a single data block, so a file needs multiple FAT entries to specify its mapping on the disk. You can implement this by making the FAT entries specify a linked list (a "FAT chain"). The directory entry therefore will not specify all of the data blocks used by the file - it can merely point to the FAT table entry that defines the head of the chain of FAT entries mapping the blocks to the file. It is important to realize that the FAT table is implemented on disk, *not* memory. Therefore you will want to use sort sort of index into the FAT, rather than ordinary C pointers, to define the list.

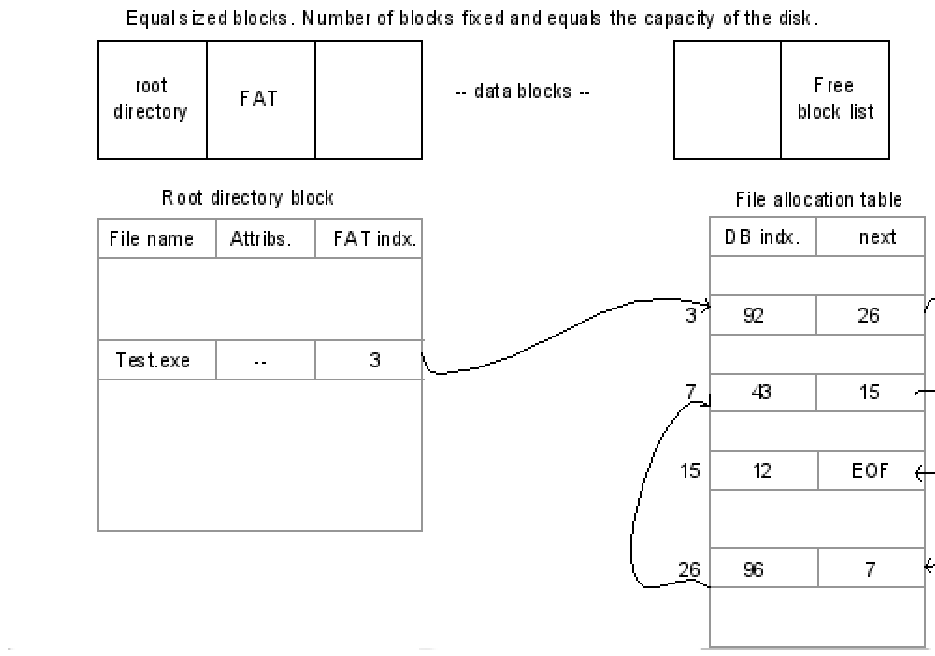

Figure 1: On-disk data structures of the file system

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. This points to data block 92 which holds the first portion of `test.exe`. Suppose a data block contains 1000 bytes. Bytes 0-999 of `test.exe` will be found in data block 92. The last data block, 12, may not be fully populated with data - this can be determined by maintaining a separate size attribute.

In addition to the on-disk data structures, we use a set of in-memory data structures to implement the file system. The in-memory data structures improve the performance of the file system by caching important information. Two data structures should be used in this assignment: a directory table and a file descriptor table. The directory table maintains a copy of the directory block(s) in memory. Whenever you create, delete, read, or write a file, you will probably need to locate the file's directory entry. Since it is accessed so often, you will need to keep this structure in memory.

The file descriptor table will keep track of dynamic information about open files, including the current read and write offsets.

Another good candidate for caching in 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 table and a system-wide table. This is more general and closely follows the UNIX implementation. You can simplify this and have only a single table, which is sufficient because we assume that only one process is accessing a file at any given time.

As shown in Figure 2, the entry in the file descriptor table is used to store the read and write offsets for an open file, as well as the first FAT entry for the file. When one writes to a file, the write offset advances by the number of bytes written to the file. Normally the write offset would point to the end of the file unless it is explicitly changed (e.g. by the fseek() function in C/UNIX). Because

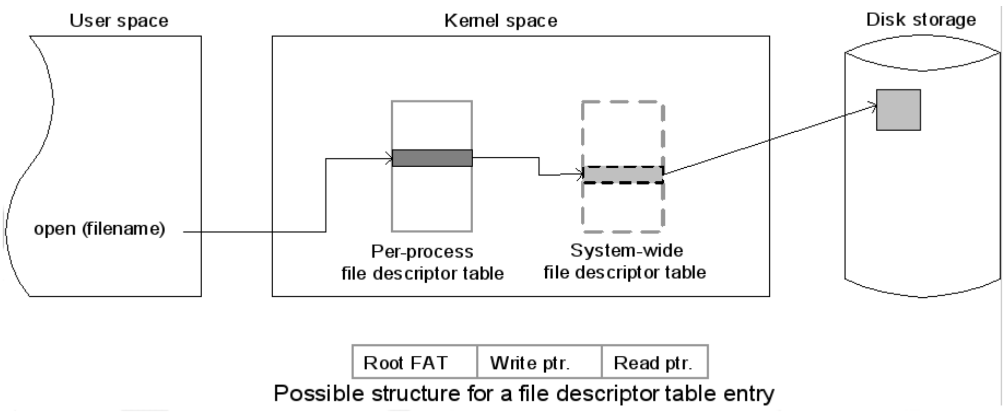
User space
Kernel space
Disk storage
open (filename)
Per-process file descriptor table
System-wide file descriptor table
Root FAT
Write ptr.
Read ptr.
Possible structure for a file descriptor table entry

Figure 2: In-memory data structures of the file system

SFS does not require a seek operation, the write offset is only advanced by write operations, and SFS allows for sequential access only. Please note that you are required to support reading and writing arbitrary length chunks of data.

In addition to these data structures, you may also wish to cache FAT and/or directory blocks.

The following are descriptions of some of subtasks required for the main operations supported by the file system: creating a file, growing a file, and removing a file.

To create a file:

1. Allocate and initialize a FAT node which will be the head of the file's FAT chain.
2. Add the mapping between the FAT node and file name in the root directory.
3. Write this information to disk.

To grow a file:

1. Allocate a disk block (remove it from free list).
2. Modify the file's FAT chain to include the new block.
3. Write the new data to the new block.
4. Write all modifications to disk.

To remove a file:

1. Remove all pointers to disk blocks from the file's FAT chain.
2. Mark all of the file's blocks as free.
3. Mark the directory entry as free.
4. Write all modifications to disk.

## 5 What to hand in

Submit your files on WebCT. It is preferred that you create a “tar” file consisting of a README describing the specifics of your implementation, the source files (*.c; *.h; or *.java), and any Makefiles. Your README should include the following information:

1. Your full name and McGill student ID number
2. If you have not implemented the full functionality, clearly indicate the parts that work and how to test those features, so that you can receive partial credit.
3. A list of all source files required to compile, run and test the code. You can supply any test scripts you create.
4. Output from your testing sessions.

You can use the included program `sfs_test.c` to test your implementation, but you may add tests or modify this code as needed for your implementation.