CS39002: Operating Systems Lab
Spring 2012

Assignment 8
Due: April 5, 2012, 1 pm

Part 1: Design and Implementation of a Filesystem (80% of the total marks)

In this assignment, you will design a filesystem of your own. Your filesystem will reside physically in a file on disk. The name of the file (for example, *$HOME/myFS*) will be the name of your filesystem. In the rest of this assignment, we will refer to the name of the filesystem as *FS_Name*. The raw size of your filesystem should be not less than 100 KB.

Your filesystem will store files and directories. The directories will also be treated as files. Each file will have the following attributes: *name, type (file or directory), size (in bytes), time of creation,* and *mode* (*read-only*, *read-write*). The *mode* attribute is valid only if the file is a regular file. You will have to maintain a tree-structured directory organization. You should be able to create and delete files and directories in your filesystem, and read and write to the files as well.

Each file and directory should have an i-node allocated to it on the filesystem. The i-node, along with file attributes, should store the blocks containing the file/directory data. The block addresses should be stored using 8 direct blocks and 2 single-indirect blocks.

Think of your filesystem as a sequence of bytes, divided into 64-byte long blocks. The first block should be the *boot* block, and should be filled with all 0's as your filesystem is not bootable. The next block (or blocks if you need more than one) will be the *superblock*(s). Your superblock(s) will contain at least the following information: the name of the filesystem, its size, the i-node number of the root directory, number of blocks used, number of free blocks, number of free i-nodes, and location of the i-node list. In addition, you are free to keep any other information that you may need to manage your filesystem.

The i-nodes must be stored immediately after the superblock(s) (design your own scheme for storing them), followed by any free space-management info that you may need (you can use any scheme). The data blocks will come after that.

The filesystem will support a *myformat*() function call to create the filesystem. This is the same as logical formatting of a disk. It should create the superblock(s) on your filesystem, create and initialize any persistent data structures you need to store on the filesystem, and set to 0 all other locations in the filesystem. The *myformat*() function can be called from within a program to "format" your filesystem at any time.

After the filesystem is created, any application program should be able to use it by making the following calls:

1. *mount(FS_Name)*: This function creates and initializes the in-memory data structures that you need for subsequent operations. This will also need to read the superblock of the filesystem and cache it in memory. This function should be the first function called, and returns 0 on success. Trying to mount an already mounted filesystem should return an error. Making any of the following operations on an unmounted filesystem should also return an error.

2. *unmount(FS_Name)*: This function frees the in-memory data structures created, making sure that any cached data is properly written back to disk if not already so. This should be the last function called.

3. *myopen(filename, mode)*: This opens a file by name. Mode can be read-only or read-write (use "r" and "w"). If the file does not exist, it is created with the mode specified, with 0 size. If an existing file is attempted to be opened in the wrong mode, an error occurs. The function returns an integer (> 0) file descriptor to the file on success, 0 otherwise.

4. *myread*: This has the same syntax, meaning, and return value as the *read* call you know.

5. *mywrite*: This has the same syntax, meaning, and return value as the *write* call you know.

6. *myclose*: This is same as the *close* call.

7. *mymkdir(dirname)*: This creates a new directory *dirname.*

8. *myrmdir (dirname)*: This deletes an existing directory *dirname*. If the directory is not empty, an error occurs.

9. *myrm (filename)*: This deletes the file *filename* from your filesystem.

10. *myls(dirname)*: This lists all files and directories under the directory *dirname* in your file system. If this command is called without the parameter *dirname*, contents of the root directory is listed.

Your filesystem should implement these calls efficiently. You should keep a file descriptor table in memory to keep track of open files and their information. In addition, you can cache any other info from disk that you want in order to efficiently implement your calls. However, if the cache copy is modified, you should have a scheme for updating the disk copy within reasonable time.

For Part 1, you can assume that only a single process will access the filesystem at one time, so there is only one file descriptor table.

If you decide to do only Part 1 and skip Part 2 below, you should package the filesystem as a static library named *myFS*, with appropriate *myFS.h* and *myFS.c* files. However, if you do Part 2 below also, you do not have to create the static library, please read on for what is required in that case.

Part 2: Integrating with FUSE **(20% of the total marks)**

If you are to use the functions you wrote from an application, your application will have to call the *my** functions, so an existing application that uses standard linux system calls like *open*, *read*, *write* will not work. Integrating your code with the FUSE software (*fuse.sourceforge.net*) allows you to make a standard application using standard linux calls to access your filesystem. In this part, you will have to study FUSE from the Internet, and see how you can integrate your code with it.

For this part, you need to change some of the function prototypes above so as to match function prototypes provided by FUSE, but ultimately your application should be able to do the same operations, namely, *open*, *read*, *write*, *close*, *mkdir*, *rmdir*, *rm*, and *ls* (some of them are functions that you will call from a program, some of them like *ls* are commands you give on the shell prompt).

For this part, you will also have to handle the case when multiple processes access the same file. Thus, you will have to maintain per-process file-descriptor tables.

Information on what to submit exactly will follow later.
