Operating Systems Laboratory
Spring Semester 2017-18

Assignment 4: *Implement a Memory Resident Unix-like File System*

Assignment given on:	March 05, 2018
Interim evaluation:	March 12, 2018
Assignment deadline:	March 19, 2018

Implement a memory resident file system (MRFS) that can be accessed using a library of Application Programming Interfaces (APIs). The API will contain the following functions:

a) Create the file system, with function prototype:

```
int create_myfs (int size);
```

where **`size`** denotes the total size of the file system in Mbytes. A memory block of the specified size will be dynamically allocated, and the data structure for the file system will be initialized in that space. If memory allocation fails, the function will return -1.

b) Copy a file from the PC file system to the memory resident file system, with function prototype:

```
int copy_pc2myfs (char *source, char *dest);
```

where **`source`** denotes a file in the PC file system, while **`dest`** is the new file in the MRFS to be created (in the current working directory). If the operation is unsuccessful, the function will return -1.

c) Copy a file from the memory resident file system to the PC file system, with function prototype:

```
int copy_myfs2pc (char *source, char *dest);
```

where **`source`** denotes a file in the MRFS in the current working directory, while **`dest`** is the new file in the PC file system. If the operation is unsuccessful, the function will return -1.

d) Remove a specified file from the current working directory, with function prototype:

```
int rm_myfs (char *filename);
```

where **`filename`** is the name of the file under the current working directory that we want to remove. If the file does not exist, the function will return -1.

e) Display the contents of a (text) file, with the function prototype:

```
int showfile_myfs (char *filename);
```

where filename denotes the name of the file in the current working directory whose contents are to be displayed on the screen. If the file does not exist, the function will return -1.

f) Display the list of files in the current working directory, with the function prototype:

```
int ls_myfs ();
```

The list will be similar to the output of the "**`ls -l`**" command. The function returns -1 if the list cannot be displayed due to some error.

g) Create a new directory in the current working directory, with function prototype:

```
int mkdir_myfs (char *dirname);
```

where **`dirname`** is the name of the directory to be created. If the function fails, it returns -1.

h) Change working directory, with function prototype:

```
int chdir_myfs (char *dirname);
```

where **dirname** is the name of the new working directory under the current working directory. If the directory **dirname** does not exist, the function will return -1.

i) Remove a specified directory, with function prototype:

```
int rmdir_myfs (char *dirname);
```

where **dirname** is the name of the directory under the current working directory that we want to remove. If the directory **dirname** does not exist, the function will return -1.

j) Open a file, with the function prototype:

```
int open_myfs (char *filename, char mode);
```

where **filename** denotes the file in the current working directory to be opened, and **mode** denotes the access mode ('**r**' for read, '**w**' for write). The function returns a file descriptor, which is the index in a file table where the corresponding file information is stored. If the file does not exist, the function will return -1. When a file is opened for reading or writing, an entry in the table contains the byte offset in the file where the next read or write operation is to be carried out. The byte offset will get updated for the file read and write operations.

k) Close a file, with the function prototype:

```
int close_myfs (int fd);
```

where **fd** denotes the file descriptor of the file that is to be closed. If **fd** is not valid, the function will return -1. When a file is closed, the corresponding entry in the file table is deleted.

l) Read a block of bytes from a file, with the function prototype:

```
int read_myfs (int fd, int nbytes, char *buff);
```

where **fd** denotes the file descriptor of the (already open) file from where we want to read, **nbytes** denotes the number of bytes to read, and **buff** points to a buffer where the bytes read will be stored. The function returns the number of bytes that has been actually read, and returns -1 in case of some error.

m) Write a block of bytes into a file, with the function prototype:

```
int write_myfs (int fd, int nbytes, char *buff);
```

where **fd** denotes the file descriptor of the (already open) file into which we want to write, **nbytes** denotes the number of bytes to write, and **buff** points to a buffer from where the bytes will be written. The function returns the number of bytes that has been actually written, and returns -1 in case of some error.

n) Check for end of file, with the function prototype:

```
int eof_myfs (int fd);
```

where **fd** denotes the file descriptor of a file, already opened. The function will return 0 if the file pointer has not reached the end of file; it will return 1 if the pointer has reached the end of file; and will return -1 if **fd** is not valid.

o) Dump the memory resident file system into a file on the secondary memory, with the function prototype:

```
int dump_myfs (char *dumpfile);
```

where **dumpfile** is the name of the file in secondary memory where the MRFS will be dumped (backed up). In case of some error, the function will return -1.

p) Load the memory resident file system from a previously dumped version from secondary memory, with the function prototype:

```
int restore_myfs (char *dumpfile);
```

where **dumpfile** denotes the name of the file in secondary memory (disk) where the MRFS was dumped earlier. In case of some error, the function will return -1.

q) Display the status of the file system, with the function prototype:

```
int status_myfs ();
```

Relevant information about the file system will be displayed, like the total size of the file system, how much is occupied, how much is free, how many files are there in total, etc. The function returns -1 if the status cannot be displayed due to some error.

r) Change the access mode of a file or directory, with the function prototype:

```
int chmod_myfs (char *name, int mode);
```

where **name** denotes the file or directory, while **mode** denotes the 9-bit access permissions as followed in Unix. The function returns -1 if the file/directory does not exist.

Design Details:

Some of the design details for the file system are as follows:

- The file system will consist of a sequence of blocks, numbered as 0, 1, 2, etc., with block size of 256 bytes. The block number is considered as a 32-bit integer.
- The file system will consist of three regions: (a) super block, (b) inode list, (c) data blocks.
- The super block will contain information about the file system, like total size, maximum number of inodes, actual number of inodes being used, maximum number of disk blocks, actual number of disk blocks being used, and a data structure to keep track of the free disk blocks and inodes. Use the bitmap data structure for the purpose.
- The index node (inode) list will consist of a set of blocks (you decide how many) that will store the inodes in a sequential manner; inodes will be numbered as 0, 1, 2, etc. Each file inode will consist of the file type (0 for regular, or 1 for directory), file size in bytes (32 bits), time last modified, time last read, access permissions, and pointers to data blocks. Each pointer will be 32 bits in size, and will contain a block number. Assume there are 8 direct blocks, 1 indirect block, and 1 doubly indirect block.
- The data blocks will contain the actual data being stored in the files and directories.
- Each directory is stored as a file, where each directory entry is stored as 32 bytes, 30 bytes for the file name, and 2 bytes for the corresponding inode number. Therefore, 8 directory entries can be stored in every block.
- Use semaphores to avoid race conditions while updating the data structures used in the MRFS.

Test Cases:

The following test cases have to be created (as C/C++ programs) to test the various functionalities of the MRFS. Assume that the header "myfs.h" has to be included at the beginning of the test programs.

1. Create a MRFS of size 10Mbytes. Copy 12 files from the PC file system to the MRFS. List the files in the current working directory. Interactively ask the user to enter the name of a file to be deleted, delete it, and list the files after deletion.
2. Open a file **`mytest.txt`** in write mode. Generate 100 pseudo-random integers, write them into the file, and close the file. Ask the user to enter an integer **`N`**, and create **`N`** copies of the file with names **`mytest-1.txt`**, **`mytest-2.txt`**, etc. Use the **`open()`**, **`read()`**, **`write()`** and **`close()`** functions to create the copies. Dump the MRFS into a file **`mydump-<group_no>.backup`** on the PC file system.
3. Load the file **`mydump-<group_no>.backup`** from the PC file system. List the files in the root directory, and print the numbers stored in the file mytest.txt. Now sort the numbers, store them into another file **`sorted.txt`**, and then display the contents of the sorted file.
4. Create the following hierarchy of directories:

```
myroot ------ mydocs -------- mytext
                      -------- mypapers
       ------ mycode
```

   Create two processes P1 and P2 using **`fork()`**. P1 creates a file in the directory **`myroot/mydocs/mytext`**, and writes the alphabets A, B, …, Z there. P2 copies a file from the PC file system and stores it in the directory **`myroot/mycode`**. Verify whether the files have been created correctly.

Submission Guideline:

- Submit all the programs as a single file as **`Ass4_<groupno>.zip`**, and upload it.
- You must upload a version by March 12, 2018 (2:30 PM), where you are expected to demonstrate ***Test Case 1***, and all the APIs necessary for the same.
- The complete version must be uploaded by March 18, 2018 (11:55 PM), along with a PDF documentation explaining the details of your design.