CS 570 - Advanced Operating Systems

Assignment 3

Submission Deadline: May 03, 2020 11:55pm

This assignment constitutes **10%** of your overall grade. The purpose of this assignment is to help students develop a better understanding of:

- Implementation of file systems
- Managing multiple users of a filesystem

Code must be written in C and should compile on a Linux machine with gcc version 7.4.X or above. You may use a machine with a Linux installation or run it inside VMware Workstation or VirtualBox. Your program should be able to handle all possible errors (unless specified otherwise) and should not crash at any point.

Late Submission Policy

For unavoidable circumstances, you will have 2 late days for which no penalty will be imposed. But these 2 days are for all the assignments i.e. if you have already used both days in previous assignments, you don't have any for this one.

Viva Policy

There will be a remote viva (via Skype/Phone call). Make sure you know each and every part of your assignment. There will be a deduction in marks if you are unable to answer any question related to the assignment.

Plagiarism Policy

Familiarize yourself with LUMS policy on plagiarism and penalties associated with it. We will use a tool to check whether your assignment is plagiarized or not. If found, the case will be forwarded to the disciplinary committee.

BACKGROUND

File system is one of the most important utilities provided by the operating system. Almost all real-world applications require some form of persistent storage. The main challenge with storage devices is that the access time is very slow as compared to the computing power of the processor. File system not only hides the underlying complexity and heterogeneous nature of different devices, but also tries to fill in the gap between processing speed and access time of storage devices. One of the most widely used techniques is buffering. Data is read/written from/to the storage device in large chunks. This is what you will also be doing in this assignment.

Files are organized in directories are subdirectories. Each file/directory in a file system has an associated inode. For this assignment, your inode need to store the following information:

1. Flag to distinguish between file and directory
2. Filesize
3. Address of 4 data blocks
4. Address of 2 single indirect blocks
5. Address of 1 double indirect block

Assume 4K as the default block size for your file system.

<u>TO-DO LIST</u>

You are provided with a generator program. Use it to create a file of 1 MB. This file will serve as a storage device for your file system. Note that we are not implementing an actual file system but mimicking it to understand how the actual file system works under the hood.

You are provided with a header file "myfs.h" containing the functions' declarations. Implement the following functions to provide persistent storage capabilities to external programs.

- **int my_open(const char *pathname, int mode)**
  <u>pathname</u> is the full path of the file to be opened. <u>mode</u> specifies the opening mode of the file which can be:
  - Read(r) - If the file doesn't exist, my_open should return -1. On success, zero is returned.
  - Write(w) - If the file doesn't exist, an empty new file should be created and it's file descriptor should be returned. If there is not enough space in the file system to create a new file, -1 is returned. If the file already exists, it should be replaced with an empty file and it's file descriptor should be returned.
  - Append(w+) - If the file doesn't exist, an empty new file should be created. If it already exists, the current write-pointer should be pointing to the end of the file. On failure, -1 is returned.
  - Read/Write(rw) - If the file doesn't exist, an empty new file should be created. If it already exists, read-pointer should point to the start of the file and write-pointer should point to the end of the file. On success, the file descriptor is returned. On failure, -1 is returned.

  For the simplicity, we will assume that the file can only be opened by a single process using one and only one file descriptor. If the file is already opened, return -1.

- **int my_close(int fd)**
  <u>fd</u> is the file descriptor returned by one of the my_open calls. It should close the corresponding file. Any buffered writes should be immediately written back to the file. If no such file is opened, my_close should return -1. On success, zero is returned.

- **int my_read(int fd, void *buffer, int count)**
  If the file with file descriptor fd is opened in read-supported mode, count number of bytes should be read from the current read-pointer and written into the buffer. Pointer is updated. *my_read* returns the number of bytes read in case of success. On failure, -1 is returned.
- **int my_write(int fd, void *buffer, int count)**
  If the file with file descriptor fd is opened in write-supported mode, count number of bytes should be written at the current write-pointer from the buffer. Pointer is updated. Note that if write-pointer refers to somewhere middle of the file, my_write will result in overwriting the earlier bytes. It is also possible that some bytes are overwritten and others get appended to the end of the file resulting in the increase in filesize. my_write returns the number of bytes written in case of success. On failure, -1 is returned.
- **int my_mkdir(const char *pathname)**
  Create a directory with the specified pathname. On success, return 0. On failure, return -1.
- **int my_format(int blocksize)**
  This function is similar to what happens when you format your USB drive or a Hard Disk drive. You should format your file system into blocks of blocksize. Remember that your super block also resides on the same disk. Therefore, you will need to determine that given your file system size, how many blocks will be possible. For efficiency, use bits to keep track of free blocks.
- **int my_unlink(const char *pathname)**
  If a file with pathname exists, delete that file. The space occupied by the file should now be available to the file system. On success, zero is returned. If the file doesn't exist, -1 is returned.
- **int my_rmdir(const char *pathname)**
  If a directory with pathname exists, delete all files, directories and subdirectories inside that directory and lastly the directory itself. On success, zero is returned. If the directory doesn't exist, -1 is returned.

When you have 2 or more options to accomplish the same operation, always choose the optimized one. 10 percent of the marks will depend on your implementation and design choices. Note that at any point, the underlying file used by your file system should not shrink or grow beyond 1MB. Think of it as your hard disk which of course has a fixed size.

TEST IF YOUR FILE SYSTEM REALLY WORKS

Suppose that you want to offer storage service (e.g. Google Drive) using your own file system implementation. For simplicity, you are not required to use socket programming. Just write functions which will be called by your application. Consider three users (e.g. Nauman, Fuad and Shahbaz) who have purchased your service. Again for simplicity, you are only required to provide upload, download and delete functionalities. Each function will have a username (which is the name of the user) as an argument.

You can assume that all file contents (no matter what the file size is) can be passed and retrieved in a single function call. Different users may choose the same name for their file, therefore, they should be saved independently and each user should be able to access only their files. Note that although we are considering only three users, this information should also be stored on your file system and your code should be flexible enough to incorporate any number of users.

Create a main function inside your test file, call upload, download and delete functions multiple times for different users to test your implementation. Format your file system with 4K block size.

Deliverables

- myfs.h
- myfs.c
- test.c

Marks Division

- Format function [10]
- Open function [8]
- Close function [2]
- Read function [4]
- Write function [8]
- Mkdir function [7]
- Unlink function [6]
- Rmdir function [10]
- Test code [25]
- Code readability [5]
- Optimization [10]

Please stick to the following guidelines to get full credit:

- There should be no compilation errors.
- Write readable and well-commented code.
- Zip all the source files and name it as **YourRollNumber_assignment3.zip** and upload it on the LMS within the given deadline. *No email submissions will be accepted*.