B.Tech, 5th Semester

UE16CS305: Introduction to Operating System Laboratory

Lab Project - Simple File System

Goals: The primary goal for this lab project is for you to have practical knowledge of building a functional file system for Linux. Doing so will familiarize you with the concepts of Linux file systems, file I/O, device interactions and system calls.

Also, this would help you to work in a team setting, understand the requirements and design a system to meet these requirements.

Problem Statement: To build a simple file system. The details are as follows:

The project can be decomposed into three phases in terms of development:

PHASE I: System call implementation

This requires you to implement system calls for file operations that can be issued by Linux. For the system call components, you can use a software facility call FUSE. FUSE (File System in User Space) https://en.wikipedia.org/wiki/Filesystem_in_Userspace is available for most Linux systems. It provides a way to intercept file system calls issued by Linux programs and to redirect the program flow into a handler (daemon running as a user-level process) function. It also helps to mount the file system. Now if a programmer (via any Linux program) makes a file system call, FUSE will invoke a routine in a daemon process (http://man7.org/linux/man-pages/man3/daemon.3.html) or a handler that you have written instead of sending that system call to the Linux file system implementation in the kernel. And in return it will show you the response by the handler function rather than the original file system.

So, FUSE helps you in writing a virtual file system. Thus, you can test your file system as if it were any other file system. More importantly, you can compare the results that your file system produces to the results produced by the Linux file system. This comparison between the results of your file system vs linux file system is how we will evaluate your work for phase 1 at the time of submission.

Task:

Implement the basic File I/O operations i.e. open/close/read/write/seek and directory functions such as mkdir, readdir etc. To perform this task, you are required to build an emulator for secondary storage that can be loaded in memory with your file system. This emulator need not persist beyond the lifetime of a mount i.e. it does not implement persistence.

- Implement mkfs (https://linux.die.net/man/8/mkfs) to make a file system in the emulator
- Implement the file abstractions (block management, block maps, directories, etc.) using the in-memory emulator
- Integrate the file system implementation with FUSE

At the end of phase 1, you should have a basic file system that works in memory only. When you unmount the file system all files are lost. Similarly, when you mount it, you must make a new file system before any operations start.

Make sure to finish this task by first week of October 2018. Your lab mentors will be checking on your progress. 2nd Week of October will be your Phase 1 demo along with Viva 2 on File System.

PHASE II: File System Abstractions

This requires developing the internal data structures and procedures necessary to implement files. You have the freedom to build this in your own way but make sure that the semantics of the file systems are correct. Any internal organization is acceptable however it must be in memory only (i.e. you can't just stick things in a database). A possible implementation is given below.

Task

Port your file system from memory to use secondary memory. To accomplish Phase 2, you will need to swap out the in-memory emulator to the raw disk device itself. It's important for a file system to store its data in a persistent manner. The persistent medium in this phase can be a disk. Your task is to design a simple disk library that reads and writes 4-Kbyte disk blocks. Your file system will be supported on top of this interface.

However, it is difficult to provide a raw disk for everyone, an alternative is to implement your disk library using a single large Unix file to emulate a disk. Your disk library will divide the file into 4-Kbyte blocks. Blocks will be written using a ``block number." You will use this number to compute the offset into the Unix file. For instance, disk block 10 will be at byte offset 10 * 4096 = 40960.

The disk interface is as follows:

*int openDisk(char *filename, int nbytes);*

openDisk() opens a file (filename) for reading and writing and returns a descriptor used to access the disk. The file size is fixed at nbytes. If filename does not already exist, openDisk creates it. The calls return an error if the underlying Unix system calls fail.

*int readBlock(int disk, int blocknr, void *block);*

readBlock reads disk block blocknr from the disk pointed to by disk into a buffer pointed to by block. If you read a block beyond the end of the disk, it should return an error. Therefore, there must be a way of specifying how big the disk is.

*int writeBlock(int disk, int blocknr, void *block);*

writeBlock writes the data in block to disk block blocknr pointed to by disk. Overwrite on a block will be treated as overwrite and no error will be produced. Error going beyond end of disk must be captured.

Your task is to implement this interface. After this phase, your file system will be able to survive across mounts (i.e. implement persistence).

PHASE III: Secondary Storage

The third part of this lab project is to write the secondary storage management such that it implements persistence and preserves the file system state across machine reboots. The goal is to be able to shut down your file system (either through and unmount or a machine reboot) and have all the files remain intact and in the same state when the file system is remounted.

Task

Complete as many of the FUSE file operations as you can and optimize the performance. At the end of Phase 2, you may have a working file system but it is likely to be quite slow if you are careful about persistence. At the end of Phase 3, you should be able to run regular Linux commands (e.g. tar, gcc, grep, vi, etc.) in your file system just as on the Linux file system itself. The expectation for Phase 3 is that it will be faster than Phase 2, but no less reliable. That is, the Phase 3 version is a more complete version of the necessary Linux functionality, that may also improve performance.

What to hand in?

Submit the source code (including Makefile) and documentation. You are supposed to make a report (documentation) which documents the file system and its features, as you have implemented them. You must specify amongst other features, how you handled meta data and persistence in your design. There are separate marks for documentation and based on how well you document your design, marks will be allotted. Also, marks will be allotted based on individual contributions.