CIS 450 Operating Systems
Program 3
Winter 2020
Due Dates: April 10 and April 17

Objectives

There are two objectives to this assignment:

1. To understand how file systems work, specifically the directory hierarchy and storage management.
2. To understand some of the performance issues file systems deal with.

Overview

This project will build a user-level library, UMDlibFS, that implements a good portion of a file system. The file system will be built inside of a library that applications can link with to access files and directories. The library will simulate interaction with a "disk.”

There is a “working disk” and a “external disk” file represented by arrays in the program. The working disk represents actions in RAM and the external file disk represents the actual updated data on the external memory device.

UMDLibFS Specification

The UMDLibFS API to the file system has three parts to the API: three generic file system calls, a set of calls that deal with file access, and a set of calls that deal with directories.

Applications (e.g., the test cases) will link with UMDLibFS in order to test out the file system.

The library will be tested on how it functions and also particularly on how it handles errors. When an error occurs (each possible error is specified below in the API definition), the library should set the global string variables **osErrMsg** and **diskErrMsg** to the error described in the API definition below and return the proper error code. This way, applications that link with the library have a way to see what happened when the error occurred.

The two “disk” image will be made up of sectors of size [SECTOR_SIZE] 512 and have a total of [NUM_SECTORS] 1000 sectors.

Generic File System API

- **int FS_Boot()**
  FS_Boot() should be called exactly once before any other UMDLibFS functions are called. It will determine if the external disk image exists or if it doesn't exist then a new disk image needs to be created. The external disk is then copied into the working disk and verified it is a file system. Upon success, return 0. Upon failure, return -1 and set osErrMsg to E_FILE_BOOT.
- **int FS_Sync()**
  FS_Sync() makes sure the contents of the file system are stored persistently on disk. This copies the working disk image to the external disk image. Return 0, in real file systems there could be errors opening and writing the file.
- **int FS_Reset()**
  FS_Reset() makes FS_Sync() call and then makes file system unavailable for access until FS_Boot() is called. Upon success, return 0. Upon failure (e.g. FS_Reset() call fails), return -1 and set osErrMsg to E_FILE_RESET.
- **ANY ACCESS ATTEMPT WHILE FILE SYSTEM NOT AVAILABLE**
  Return a -1 and set osErrMsg to E_INVALID_ACCESS_ATTEMPT

File Access API

Note that a number of these operations deal with **pathnames.**

Assumptions about pathnames.

- All pathnames are **absolute.** That is, anytime a file is specified, the full path starting at the root is expected.
- The maximum name length of a single file name is 16 bytes (15 characters plus one for an end-of-string delimiter).
- The maximum length of a path is 256 total characters.

Methods – *file* is full path name

- **int File_Create(string file)**
  File_Create() creates a new file of the name pointed to by *file.* If the file already exists return -1 and set osErrMsg to E_FILE_CREATE. Note: the file will not be "open" after the create call. Rather, File_Create() will simply create a new file on disk of size 0. Upon success, return 0. Upon a failure, return -1 and return E_FILE_CREATE.
- **int File_Open(string file)**
  File_Open() opens up a file (whose name is *file*) and returns an integer file descriptor (a number greater than or equal to 0), which can be used to read from or write data to that file. If the file doesn't exist, return -1 and set osErrMsg to E_NO_SUCH_FILE. If there are already a maximum number of files open, return -1 and set osErrMsg to E_TOO_MANY_OPEN_FILES.

- **int File_Read(int fd, string buffer, int size)**
  File_Read() should read *size* bytes from the file referenced by the file descriptor *fd.* The data should be read into a buffer area *buffer.* All reads should begin at the current location of the file pointer, and file pointer should be updated after the read to the new location. If the file is not open, return -1, and set osErrMsg to E_READ_BAD_FD. If the file is open, the number of bytes actually read should be returned, which can be less than or equal to *size.* (The number could be less than the requested bytes because the end of the file could be reached.) If the file pointer is already at the end of the file, zero should be returned, even under repeated calls to File_Read(). *How many buffers will you need?*
- **int File_Write(int fd, string buffer, int size)**
  File_Write() should write *size* bytes from *buffer* and write them into the file referenced by *fd.* All writes should begin at the current location of the file pointer and the file pointer should be updated after the write to its current location plus *size.* Note that writes are the only way to extend the size of a file. If the file is not open, return -1 and set osErrMsg to E_BAD_FD. Upon success of the write, all of the data should be written out to disk and the value of *size* should be returned. If the write cannot complete (due to a lack of space), return -1 and set osErrMsg to E_NO_SPACE. Finally, if the file exceeds the maximum file size, you should return -1 and set osErrMsg to E_FILE_TOO_BIG.
- **int File_Seek(int fd, int offset)**
  File_Seek() should update the current location of the file pointer. The location is given as an offset from the beginning of the file. If *offset* is larger than the size of the file or negative, return -1 and set osErrMsg to E_SEEK_OUT_OF_BOUNDS. If the file is not currently open, return -1 and set osErrMsg to E_SEEK_BAD_FD. Upon success, return the new location of the file pointer.
- **int File_Close(int fd)**
  File_Close() closes the file referred to by file descriptor *fd.* If the file is not currently open, return -1 and set osErrMsg to E_CLOSE_ BAD_FD. Upon success, return 0.
- **int File_Unlink(string file)**
  This should delete the file referenced by *file,* including removing its name from the directory it is in, and freeing up any data blocks and inodes that the file was using. If the file does not currently exist, return -1 and set osErrMsg to E_NO_SUCH_FILE. If the file is currently open, return -1 and set osErrMsg to E_FILE_IN_USE (and do NOT delete the file). Upon success, return 0.

Directory API

- **int Dir_Create(string path)**
  Dir_Create() creates a new directory as named by *path.* Reminder, all paths are **absolute** paths, i.e., assume no tracking the current working. Creating a new directory takes a number of steps: first, you have to allocate a new file (of type directory), and then add a new directory entry in the current directory's parent. Upon failure of any kind, return -1 and set osErrMsg to E_DIR_CREATE. Upon success, return 0. Note that Dir_Create() is **not** recursive -- that is, if only "/" exists, and you want to create a directory "/a/b/", must first create "/a", and then create "/a/b".

- **int Dir_Size(string path)**
  Dir_Size() returns the number of bytes in the directory referred to by *path.* This routine should be used to find the size of the directory before calling Dir_Read() (described below) to find the contents of the directory.
- **int Dir_Read(string path, string buffer, int size)**
  Dir_Read() can be used to read the contents of a directory. It should return in the buffer a set of directory entries. Each entry is of size 20 bytes, and contains 16-byte names of the directories and files within the directory named by *path,* followed by the 4-byte integer inode number (an int data type in C++ is 4-bytes in size – amazing coincidence?). If *size* is not big enough to contain all of the entries, return -1 and set osErrMsg to E_BUFFER_TOO_SMALL. Otherwise, read the data into the buffer, and return the number of directory entries that are in the directory (e.g., 2 if there are two entries in the directory).
- **int Dir_Unlink(string path)**
  Dir_Unlink() removes a directory referred to by *path,* freeing up its inode and data blocks, and removing its entry from the parent directory. Upon success, return 0. Note: Dir_Unlink() should only be successful if there are no files within the directory. If there are still files within the directory, return -1 and set osErrMsg to E_DIR_NOT_EMPTY. If someone tries to remove the root directory ("/"), don't allow them to do it! Return -1 and set osErrMsg to E_DEL_ROOT_DIR.

File Pointer Notes

When reading or writing a file, implement a notion of a *current file pointer.* After opening a file, the current file pointer is set to the beginning of the file (byte 0). If the user then reads N bytes from the file, the current file pointer should be updated to N. Another read of M bytes will return the bytes starting at offset N in the file, and up to bytes N+M. Thus, by repeatedly calling read (or write), a program can read (or write) the entire file. Of course, File_Seek() exists to explicitly change the location of the file pointer.

Again, assume no implementing any functionality that has to do with relative pathnames. In other words, all pathnames will be absolute paths. Thus, all pathnames given to any of your file and directory APIs will be full ones starting at the root of the file system, *i.e.,* /root/a/b/CIS450.cpp. Thus, your file system does **not** need to track any notion of a "current working directory".

Implementation Hints

The Disk Abstraction

One of the first questions to ask is "Where is all of the file system data stored?" A real file system would store it all on disk, but since this is at user-level, it is all stored in a "fake" disk, in an array.

The "disk" has constants NUM_SECTORS sectors, each of size SECTOR_SIZE. Thus, use these values in creating the file system structures. The model of the disk is quite simple: in general, the file system will perform disk reads and disk writes to read or write a sector of the disk. In actuality, the disk reads and writes access an in-memory array for the data; other aspects of the disk API allow you to save the contents of your file system to a external disk, and later, restore the file system from that external disk file into the working disk.

Here is the basic disk API:

- **int Disk_Init()**
  Disk_Init() should be called **exactly once** by the OS before any other disk operations take place. Make the data in the array of sectors all zeroes. Return 0, in real file systems there could be a memory operation error in allocating and writing the data. Create superblock and empty root directory entries.
- **int Disk_Load()**
  Disk_Load() is called to load the contents of the external disk file system array into the into working disk. This routine (and Disk_Init() before it) should be executed once when it is "booting", i.e., during FS_Boot(). Return 0, in real file systems there could be an error opening the external file or memory operation error in reading the data.
- **int Disk_Save()**
  Disk_Save() saves the current in-memory working disk to the external disk. This routine will be used to save the contents of in-memory file system to the external file disk, so it can be later "booted" off of it again. This routine should be invoked by FS_Sync(). Return 0, in real file systems there could be an error opening the external file or memory operation error in writing the data.
- **int Disk_Write(int sector, string buffer)**
  Disk_Write() writes the data in *buffer* to the sector specified by *sector*. The buffer is assumed to be of size sector exactly. Success return 0; if sector less than 0 or sector greater than or equal to NUM_SECTORS or buffer is NULL return -1 and set diskErrMsg to E_WRITE_INVALID_PARAM. In real file systems there could also be a memory operation error in writing the data.
- **int Disk_Read(int sector, string buffer)**
  Disk_Read() reads a sector from *sector* into the buffer specified by buffer. As with Disk_Write(), the buffer is assumed to be of size sector exactly. Success return 0; if sector less than 0 or sector greater than or equal to NUM_SECTORS or buffer is NULL set diskErrMsg to E_READ_INVALID_PARAM. In real file systems there could also be a memory operation error in writing the data.

For all of the disk API: All of these operations return 0 upon success, and -1 upon failure. If there is a failure, diskErrMsg is set to some appropriate message.

On-Disk Data Structures

A big part of understanding a file system is understanding its data structures. Of course, there are many possibilities. Below is a simple approach, which may be a good starting point.

First, somewhere on disk you need to record some generic information about the file system, in a block called the **superblock.** This should be in a well-known position on disk -- in this case, make it the very first block. For this assignment, don't need to record much there – there is exactly one thing in the superblock -- a **magic number.** Pick any number, and when a new file system is initialized (as described in the booting up section below), write the magic number into the superblock. Then, when booting up with this same file system again, make sure that when that superblock is read, the magic number is there. If it's not there, assume this is a corrupted file system (and that it can't be used) – hence the error occurs in FS_BOOT.

To track directories and files, you need two types of blocks: **inode blocks** and **data blocks.**

inode

In each inode, track at least three things about each file.

- the size of the file
- the type of the file (normal or directory)
- which data blocks are allocated to the file

For this assignment, assume that the maximum file size is 10 blocks. Thus, each inode should contain at least 1 integer (size), 1 integer (type), and 10 pointers (to data blocks). Each inode is likely to be smaller than the size of a disk sector -- put multiple inodes within a disk sector to save space.

data blocks

Assume that each <u>data block is the exact same size as a disk sector</u>. Thus, part of disk must be dedicated to these blocks.

The file system will track which inodes have been allocated, and which data blocks have been allocated. One simple way to do this would use a bit map for each, i.e., the first block after the superblock should be the inode bitmap, and the second block after the superblock should be the data block bitmap.

One painful part about any file system is pathname lookup. Specifically, to open a file named /CIS450/prog3/UMDLibFS.cpp, first look in the root directory ("/"), and see if there is a directory in there called "CIS450". To do this, start with the root inode number (which should be a well-known number, like 0), and read the root inode in. This will tell how to find the data for the root directory, which should then be read in, and look for CIS450 in it. If CIS450 is in the root directory, find its inode number (which should also be recorded in the directory entry). From the inode number, figure out exactly which block to read from the inode portion of the disk to read CIS450's inode. Once the data within CIS450 is read, check to see if a directory "prog3" is in there, and repeat the process. Finally, get to "UMDLibFS.cpp", whose inode can be read in, and from there the system will get ready to do reads and writes.

Open File Table

When a process opens a file, first perform a path lookup. At the end of the lookup keep some data in order to be able to read and write the file efficiently (without repeatedly doing path lookups). This information should be kept in an **open file table.** When a process opens a file, allocate it the first open entry in this table -- thus, the first open file should get the first open slot, and return a file descriptor of 0. The second opened file (if the first is still open) would return a descriptor of 1, and so on. Each entry of the table should track what is needed to know about the file to efficiently read or write to it -- think about what this means and design your table accordingly. The limit the size of the table is a fixed size of 10. Once a file is closed that file description number can be reused.

Disk Persistence

The disk abstraction provided above keeps data in memory until Disk_Save() is called. Thus, call Disk_Save() to make the file system image persistent. A real OS commits data to disk quite frequently, in order to guarantee that data is not lost. However, in this assignment, this will only occur when FS_Sync() is called by the application which links with the UMDLibFS. Reminder - Disk_Read and Disk_Write are associated with the external disk and File_Read and File_Write are assciated with the working disk. Whatever is written to the working disk needs to eventually be written to the external disk.

Booting Up

When "booting" the OS (i.e., starting it up), if the external file disk exists, load it (via Disk_Load()), and then check and make sure that it is a valid disk - the superblock should have the magic number expected to be in there (as described above). If this piece of data is incorrect, report the error and exit. In real file systems another check occurs, the file size is checked to be equivalent to NUM_SECTORS times SECTOR_SIZE.

However, there is one other situation: if the external disk file does not exist (i.e. has not been initalized), this means a new external disk is created and its superblock magic number initialized, and an empty root directory in the file system is created. Thus, in this case, use Disk_Init().

Other Notes

Caching: Your file system should **not perform any caching.** That is, all operations should read and write the Disk API.

Directories: Treat a directory as a "special" type of file that happens to contain directory information. Thus, have to have a value in the inode that tells whether the file is a normal file or a directory. Keep your directory format simple: a fixed 16-byte field for the name, and a 4-byte entry as the inode number (remember an int data type in C++ is 4-bytes – how convenient!) .

Maximum file size: 10 sectors. If a program tries to grow a file (or a directory) beyond this size, it should fail. This can be used to keep your inode quite simple: keep 10 disk pointers in each inode. Don't worry about indirect pointers or anything like that (that a real file system would have to deal with). Maybe use array index as the data block pointer?

Maximum element length in pathname: 16 characters. Thus, keep it simple and reserve 16 bytes size for each name entry in a directory.

If File_Write() only partially succeeds (i.e. some of the file got written out, but then the disk ran out of space), return -1 and set osErrMsg to E_PARTIAL_FILE_WRITE.

You should **not** allow a directory and file name conflict in the same directory (i.e. a file and a directory of the same name in the same directory) – set osErrMsg to E_DUPLICATE_FILE_NAME.

Assuming that the maximum name length of a file is 16 bytes means 15 characters plus one for an end-of-string delimiter.

The maximum length of a path is 256 characters.

The maximum number of open files is 10.

Legal characters for a file name include letters (case insensitive), numbers, dots.

You should enforce a maximum limit of 100 files/directories. Define a constant internal to your OS called MAX_FILES and make sure you observe this limit when allocating new files or directories (100 total files and directories).

For ease of implementation –

Block 1 – superblock

Block 2 - inode bitmap – this can be an array of boolean values for inode location in use/available rather than a “true” bitmap that a real file system would use as the booleam array will fit into a single block based on program assumptions.

Block 3 – data block bitmap – a real file system would use this with a true bitmap as the block would hold a map to 4096 data blocks – this will be a “reserved for data block bitmap” block to indicate it would be used for the bitmap in a real file system. Since C++ does not have a bit variable data type simulate it by using a separate (i.e. not in disk) boolean array to represent the in use/available data blocks. This would need to be initalized, copied, etc. when disk is created and persisted .

Programming Stages

The team will build/write the program code in three stages - writing and testing groups of methods.

1. FS_Boot, FS_Sync, FS_Reset, Disk_Init, Disk_Load, Disk_Save, Disk_Write, Disk_Read, Dir_Create, Dir_Size, Dir_Read

2. Everything in group 1 plus File_Open, File_Read, File_Write, File_Close

3. Everything in group 2 plus Dir_Unlink, File_Unlink, File_Seek

You may “hard code” tests or have an input file into your test main() drivers.

Be sure you test for all the error conditions!

Turn-In

Part 1 –

Specifications section – since this is a IT system program not a true “business” application copy the UMDLibFS specification section above into the template business specification section. The assumptions should all be listed in the appropriate section.

Decomposition diagram – “not applicable” as already decomposed

Test Strategy – valid data and invalid data are only strategies

You do need to list test cases and algorithm breakdown for all the methods

Part 2 – program output

There are TWO output files PLUS File 1 data is displayed on screen

1. Directive log showing the directives, parameters of the directive (if any), result of success or error(s) in the order executed
2. Directive log showing the directives, parameters of the directive (if any), result success or error(s) in the order executed plus memory dump of the working disk and/or external disk as appropriate. Label data appropriately and remember to print out the data block “bitmap” when doing a memory dump.