COSC 4740
Final Project

Due date: Nov 30

I hear and I forget, I see and I remember, I do and I understand. – Chinese proverb

Goal: To design and implement a subset of the Unix file system.
Grade: 30% of your total grade is allocated for this project. All the files you will need are in the repo, so your group must create the repo first. READ EVER FILE, there is lots of information in the code and other files included. Failure to read everything is your failure and likely lot of time and points.

The filesystem consists of five classes:

1. *Disk*: Simulates the behavior of a disk that is used to store files and other filesystem information.
2. *DiskManager*: Partitions and manages access to a disk.
3. *PartitionManager*: Manages partitions in a disk.
4. *FileSystem*: Implements various filesystem operations.
5. *Client*: Uses a filesystem.

Disk

Simulates the behavior of a disk using a Unix file. It exports the following functions:
int **initDisk**()
int **readDiskBlock**(int *blknum*, char **blkdata*)
int **writeDiskBlock**(int *blknum*, char **blkdata*)
int **getBlockSize**()
int **getBlockCount**()

The first function creates a Unix file to simulate a disk. If such a file already exists, it returns 0, otherwise it creates the file of specified size and returns 1. The second function reads the disk block numbered *blknum* into the buffer pointed to by *blkdata*. The third function copies the buffer pointed to by *blkdata* to the disk block numbered *blknum*. Both of these functions return 0 if successful and a negative number otherwise. Every Disk object is managed by a DiskManager object.

DiskManager

Partitions and manages access to a disk. There is one DiskManager object associated with each Disk object. A DiskManager object partitions a disk into multiple partitions, and provides some of the basic functions needed by a filesystem to store its information. These functions are:

int **readDiskBlock**(char *partitionname*, int *blknum*, char **blkdata*)
int **writeDiskBlock**(char *partitionname*, int *blknum*, char **blkdata*)
int **getBlockSize**()
int **getPartitionSize**(char *partitionname*)

readDiskBlock reads the partition block numbered *blknum* into the buffer pointed to by *blkdata*. writeDiskBlock copies the buffer pointed to by *blkdata* to the partition block numbered *blknum*. getBlockSize return the size of a disk block in bytes. getPartitionSize returns the total number of blocks available in this partition.

PartitionManager

A PartitionManager object manages partitions in a disk. A PartitionManager object is created by a FileSystem object (described next) and has a partition name associated with it. Each PartitionManager object is associated with a DiskManager object. Each PartitionManager has a bitmap, that keeps track of free and allocated blocks. The bitmap is written out block 0 of the partition. It exports the following operations:

int **readDiskBlock**(int *blknum*, char **blkdata*)
readDiskBlock reads the block numbered *blknum* in a disk partition into the buffer pointed to by *blkdata*.
int **writeDiskBlock**(int *blknum*, char **blkdata*)
This operation copies the buffer pointed to by *blkdata* to the block numbered *blknum* in a disk partition.
int **getBlockSize**()
This operation returns the size of a disk block in bytes.
int **getFreeDiskBlock**()
This operation allocates a free disk block in a partition. It returns -1 if there is no block available (ie the partition is full) or the block number allocated.
int **returnDiskBlock**(int *blknum*)
This operation deallocates a disk block in a partition. It will also write a blank (all c's) to the block that is returned. It returns 0 if successful or -1 for any other reason.

FileSystem

A FileSystem object implements a Unix file system. Each FileSystem object is associated with a DiskManager object, which it uses for all disk I/O. Multiple FileSystem objects may be associated with a single DiskManager object, which implies that multiple file systems can share a disk. This is equivalent to partitioning of a single disk drive into multiple directories in modern systems. Each partition is managed by a PartitionManager object, which is created by a FileSystem object.

Files and directories are implemented using a structure similar to the Unix i-nodes. There are three i-nodes: File i-node, directory i-node, and indirect inode. Every file in the file system has an file i-node associated with it. An file i-node will contain the following information: file name (1 byte), a 1 byte variable to indicate whether it is a file or a directory, file size (4 bytes), three disk block addresses (4 bytes each), and one single indirect block address (4 bytes). Use the remaining bytes in the block to store some other file attributes of your choice. An indirect block is 64 bytes (same as block size), with 16 spots for disk block addresses (4 bytes each). A directory i-node has the following structure, 64 bytes, with 10 slots containing: name (1 byte), block pointer (4 bytes), type file/directory (1 byte) and the last 4 bytes a pointer to the next block with the directory information will continue if needed. A FileSystem object exports the following operations:

int **createFile**(char **filename*, int *fname_len*)

This operation creates a new file whose name is pointed to by filename of size fname_len characters. File names and directory names start with `/' character and consist of a sequence of alternating `/' and alphabet (`A' to `Z' and `a' to `z') characters ending with an alphabet character. The CreateFile function returns -1 if the file already exists, -2 if there is not enough disk space, -3 if invalid filename, -4 if the file cannot be created for some other reason, and 0 if the file is created successfully.

int **createDirectory**(char **dirname*, int *dnameLen*)

This operation creates a new directory whose name is pointed to by *dirname*. This function returns -1 if the directory already exists, -2 if there is not enough disk space, -3 if invalid directory name, -4 if the directory cannot be created for some other reason, and 0 if the directory is created successfully.

int **lockFile**(char **filename*, int *fname_len*)

This operation locks a file. A file cannot be locked if (1) it doesn't exist, or (2) it is already locked, or (3) it is currently opened. It returns a number greater than 0 (lock id), if the file is successfully locked, -1 if the file is already locked, -2 if the file does not exist, -3 if it is currently opened, and -4 if the file cannot be locked for any other reason. A note, once a file is locked, it maybe only be opened with the lock id and the file cannot be deleted or renamed until the file is unlocked.

int **unlockFile**(char **filename,* int *fname_len*, int *lock_id*)

This operation unlocks a file. The *lock_id* is the lock id returned by the LockFile function when the file was locked. The UnlockFile function returns 0 if successful, -1 if lock id is invalid, -2 for any other reason.

int **deleteFile**(char **filename*, int *fname_len*)

This operation deletes the file whose name is pointed to by *filename*. A file that is currently in use (opened or locked by a client) cannot be deleted. It returns -1 if the file does not exist, -2 if the file is in use or locked, -3 if the file cannot be deleted for any other reason, and 0 if the file is deleted successfully.

int **deleteDirectory**(char **dirname*, int *dnameLen*)

This operation deletes the directory whose name is pointed to by *dirname*. Only an empty directory can be deleted. This function returns -1 if the directory does not exist, -2 if the directory is not empty, -3 if the directory cannot be deleted for any other reason, and 0 if the directory is deleted successfully.

int **openFile**(char **filename*, int *fname_len*, char *mode*, int *lock_id*)

This operation opens a file whose name is pointed to by *filename*. If mode = 'r', the file is opened for read only, If mode = 'w', the file is opened for write only, and if mode = 'm', the file is opened for read and write. An existing file cannot be opened if (1) the file is locked and lock_id doesn't match with lock_id returned by the **lockFile** function when the file was locked, or (2) the file is not locked and lock id ≠ -1. This operation returns -1 if the file does not exist, -2 if *mode* is invalid, -3 if the file cannot be opened because of locking restrictions, -4 for any other reason, and a unique positive integer (file descriptor) if the file is opened successfully. If the file is opened successfully, an rw pointer (read-write pointer) is associated with this file descriptor. This rw pointer is used by some of the operations described later for determining the access point in a file. The initial value of an rw pointer is 0 (beginning of the file).

int **closeFile**(int *filedesc*)

This operation closes the file with file descriptor *filedesc*. It returns -1 if the file descriptor is invalid, -2 for any other reason, and 0 if successful.

int **readFile**(int *filedesc*, char **data*, int *length*)
int **writeFile**(int *filedesc*, char **data*, int *length*)
int **appendFile**(int *filedesc*, char **data*, int *length*)

These operations perform read/write/append operations on a file whose file descriptor is *filedesc*. *length* is the number of bytes to be read from / written into / appended into the buffer pointed to by *data*. These operations return -1 if the file descriptor is invalid, -2 if length is negative, -3 if the operation is not permitted, and number of bytes read/written/appended if successful. The read and write operations operate from the byte pointed to by the *rw* pointer. The write operation overwrites the existing data in the file and may increase the size of the file. The append operation appends the data at the end of the file. The read operation may read less number of bytes than length if end of file is reached earlier. After the read/write/append is done, the *rw* pointer is updated to point to the byte following the last byte read/written/appended.

int **seekFile**(int *filedesc*, int *offset*, int *flag*)

This operation modifies the *rw* pointer of the file whose file descriptor is *filedesc*. The *rw* pointer is moved *offset* bytes forward if flag = 0. Otherwise, it is set to byte number offset in the file. This operation returns -1 if the file descriptor, *offset* or *flag* is invalid, -2 if an attempt to go outside the file bounds is made (end of file or beginning of file), and 0 if successful. A negative offset is valid only when flag is zero.

int **renameFile**(char **fname1*, int *fname1_len*, char **fname2*, int *fname2_len*)

This operation changes the name of the file whose name is pointed to by *fname1* to the name pointed to by *fname2*. It returns -1 invalid filename, -2 if the file does not exist, -3 if there already exists a file whose name is the same as the name pointed to by fname2, -4 if file is opened or locked, -5 for any other reason, and 0 if successful. A note, you can rename a directory with this operation.

int **getAttributes**(char **filename*, ...)
int **setAttributes**(char **filename*, ...)

These operations get/set the attributes of a file whose name is pointed to by filename. Work out the details of these operations based on the file attributes you choose. You must choose a minimum of **two** attributes for your filesystem.
Client

This class is used to create client objects that invoke file system operations. Each Client object is associated with a FileSystem object.

Implementation Guidelines

1. Implement your file system using C++.
2. Unix file system commands are used to implement the Disk. No other functions should use any Unix file system functions.
3. Use bitmaps to keep track of free disk blocks. The bitvector code is provided and is to be used to implement the bitmaps.
4. Some support files are provided to you to get started. These include bitvector.h and bitvector.cc that implement a BitVector class, disk.h and disk.cc that implement a Disk class, diskmanager.h and diskmanager.cpp that partially implement a DiskManager class, and filesystem.h and filesystem.cc class that partially implement a FileSystem class. In addition, an outline of a driver program to test the file system is provided in file driver.cpp. Driver1.cpp through driver6.cpp are used when turning in the project. All these files are available on the class web page.
5. Remember that your file system should survive the termination of clients that are using it.
6. Don't code until you understand what you are doing. Design, design, design first. While weeks of programming can say you hours of planning, I don’t advise it.
7. Start working on this project now. First complete the implementation of DiskManager class.
8. Make sure you read everything, first! There is a lot information in the cpp and header files and in other files provided.

Project Submission notes:

1. You will do this project in teams of up to three students. Groups will be choosing at random.

- You will turn in a separate evaluation of each member of your group, which is worth 5 points (of 100) for the project. See hard copy section.
- As note, you are responsible for submitting the final project irrespective of how your team partners work.

2. You are expected to make at least one update a week before the project is due. Even if it only to update the readme, to add WE DID NOTHING THIS WEEK.

3. You are expected to use the issue system in github. Each group member must make at least one commit.

4. Update the README.md file in the root directory.

    - Include all the names of the group at the top of the readme.

- Include what is working and not working. This is detail how much of the project is completed and working. Say for example you only get to driver4 completed and working, then you should state that.
- Including any information about the file attributes you choose for driver 6 as well.

5. You are expected to use the makefile to compile your code in project directory. If you change/add something, then it MUST be reflected in the makefile as well. If the makefile doesn't work/compile the project, then your project doesn't compile and you will lose half the points (which is 15% of your final grade too). Only the master branch will be used. All warnings are expected to be cleaned up as well.

6. Lastly, The project will be graded by looking DISK1 after each driverX run. And then at the code itself. It is highly suggested you spend a lot of time with a hex editor looking at the disk to ensure it is correct.

Hard (email) copy:

There will be two separate emails.
First email
This is the "cover page". Designate one person to send the email, if you group can't agree, then everyone send it.
1. Subject line OS Project submission
2. The email includes List of all group members, repo name,
3. Attach the output of the drivers (well only the working drivers)

Second email
1. Subject line OS Project evaluation
2. Attached a filled out evaluation document (provided in the repo)
It will list what you did and what the rest of the group members did as well. This doesn't have to very long. Describe how the project was broken up and who did what.
DO NOT COMMIT OR PUSH THIS DOCUMENT TO GITHUB.

Soft copy:
1. Use this link to create your (group) repo
https://classroom.github.com/g/LbGnMvZa
2. Upload the project to your repo (and to correct directories).
3. See issue #3 and submission #4 to fill out the readme file.
4. Lastly ensure everything has uploaded to the github website and not just the local repo. Remember only the Master branch will be pulled.

Code will be graded on correctness, comments, and coding style.