Lecture Notes on Operating Systems

Lab: A Simple Filesystem

In this lab, we will implement some basic file operations in a simple filesystem that runs over an emulated disk. You are provided files `simplefs-disk.h` and `simplefs-disk.c` that emulate the disk, and provide functions to access the disk. Using these emulation functions, you will implement some basic filesystem operations like opening, reading, and writing files, by modifying the files `simplefs-ops.h` and `simplefs-ops.c`.

Disk Emulation

In this section, we will first describe how the disk is emulated, and the functions available to you to implement filesystem operations. You may refer to the files `simplefs-disk.h` and `simplefs-disk.c` when reading this description.

The disk of the simple filesystem is emulated over a simple text file on disk, at a location specified in the code (say,the file "simplefs"). The emulated disk consists of 35 blocks of size 64 bytes each. The first block of the filesystem is the superblock, which contains a summary of the status of all other blocks in the system. The next 4 blocks of the disk store inodes, with two inodes per block. That is, the filesystem can store a total of 8 files on disk. The remaining 30 blocks are available to store file data. There is no directory hierarchy in the filesystem; all files are addressed simply by their filenames.

The inode structure stores direct pointers for up to 4 file data blocks. That is, the size of a file in our filesystem is restricted to 4 blocks or 256 bytes. In addition to the direct pointers to data blocks, the inode also stores the size of the file (in bytes) and the filename. The inode structure is 32 bytes in size, enabling the filesystem to pack two inodes into a disk block.

The following functions are already implemented by our disk emulation code.

- `void simplefs_formatDisk()` formats the simple filesystem and initializes the superblock.
- `int simplefs_allocInode()` allocates a free inode on disk, updates the freelist of inodes in the superblock, and returns the number of the inode allocated. This function returns -1 if no free inode can be found.
- `void simplefs_freeInode(int inodenum)` frees up the specified inode, by marking it as free in the superblock.
- `void simplefs_readInode(int inodenum, struct inode_t *inodeptr)` reads the specified inode from disk and copies it into the inode pointer.
- `void simplefs_writeInode(int inodenum, struct inode_t *inodeptr)` copies the inode from the given pointer onto the disk.

- `int simplefs_allocDataBlock()` allocates a free data block on disk, updates the freelist of blocks in the superblock, and returns the number of the data block allocated. This function returns -1 if no free block can be found.
- `void simplefs_freeDataBlock(int blocknum)` frees up the specified data block, by marking it as free in the superblock.
- `void simplefs_readDataBlock(int blocknum, char *buf)` reads the contents of the specified data block on disk, and copies it into the provided buffer.
- `void simplefs_writeDataBlock(int blocknum, char *buf)` writes the contents of the specified block from the buffer onto the emulated disk.
- `void simplefs_dump()` prints the disk state in a readable format

All operations that write to the disk will make changes to the emulated disk (text file) before returning. We are currently assuming only one process modifying the text file at any time, so our emulation code does not handle any concurrency when editing the emulated disk file.

When you use the above functions in your filesystem code, ensure that you take into account the return values from the functions to catch errors. All disk operations that do not return an error code above are assumed to always succeed. For example, the operation of allocating a free data block may fail (if no blocks are free), but the operation of reading a valid data block is always expected to succeed. You are expected to catch and handle the errors for only those functions that return an error code.

Filesystem Operations

Using the emulated disk access functions described above, you will now implement the following simple operations on files in our simple filesystem. These operations are defined in `simplefs-ops.h` and must be implemented in `simplefs-ops.c`.

Unlike Linux, our simple filesystem has separate functions to create and open a file. Creating a file only creates it on disk, while opening a file opens an already created file for reading and writing. `simplefs-disk.h` defines an additional in-memory datastructure related to the filesystem, called the file handle array. The state of open files in the system (the inode number and the offset of reading/writing) is captured in a file handle of an open file. The filesystem maintains an array of all such file handles of open files, and the index of a file handle in this array is used to uniquely identify an open file in all subsequent read/write operations.

- `int simplefs_create(char *filename)` creates a file with a specified name in the filesystem. A file creation should succeed only if a file with the same name does not already exist in the filesystem, and if the system has space for an additional file to be created. During file creation, this function must allocate a free inode for the file and initialize it suitably on disk. This function returns the inode number of the newly created file on success, and -1 on a failure.
- `void simplefs_delete(char *filename)` deletes a file with the specified name from the filesystem (if it exists), and frees up the resources of the file such as its data blocks and its inode. You may assume that the file is closed before deleting it.

- `int simplefs_open(char *filename)` opens an existing file for reading and writing. This operation succeeds only if the file with the same name was previously created. This operation allocates an unused file handle from the global file handle array, initializes it suitably, and returns the index of the newly allocated file handle. This function returns -1 if the file open failed for any reason.

  Please note that an array of file handles has already been defined for you; you must populate an entry in this array when opening a file. All subsequent read and write operations on the file will have to obtain the inode number of the file using this returned file handle array index.

- `void simplefs_close(int file_handle)` closes an open file and frees up its file handle. This operation will not delete the file from disk.

- `int simplefs_read(int file_handle, char *buf, int nbytes)` reads the specified number of bytes from the current offset in an open file into the given buffer. The requested number of bytes can span multiple blocks on disk. This function returns 0 on success and -1 on failure. Note that we do not wish to support partial reads, i.e., your implementation must read all the requested `nbytes` number of bytes into the buffer, or none at all. If reading the requested number of bytes causes you to go beyond the end of the file, then this operation should simply not read anything and must return a failure. Note that you must handle the cases where reads are not aligned at the block boundary, and can start or end in the middle of a block. Note that reading a file will NOT result in updating the offset value in the file handle.

- `int simplefs_write(int file_handle, char *buf, int nbytes)` writes the specified number of bytes from the current offset in an open file to disk, from the given buffer. The requested number of bytes can span multiple blocks on disk. This function returns 0 on success and -1 on failure. Note that we do not wish to support partial writes, i.e., your implementation must write all the requested `nbytes` number of bytes into the file, or none at all. If the requested number of bytes cannot be written (e.g., writing the requested number of bytes causes you to go beyond the maximum file size limit, or there are no empty disk blocks available to complete the write) then this operation should simply not write anything and must return a failure. Further, on such failed writes, any data blocks that have been allocated before encountering the failure must be freed up and returned back to the file system. That is, a failed write must leave the filesystem in the same state that it started with. Note that you must handle the cases where writes are not aligned at the block boundary, and can start or end in the middle of a block. Note that writing a file will NOT result in updating the offset value in the file handle.

- `int simplefs_seek(int file_handle, int nseek)` increments the file offset in the file handle by `nseek` bytes. As with Linux filesystems, the offset indicates the next byte that can be read/written from the file. Note that the offset should decrease if `nseek` is negative. The seek operation should not move the offset to beyond the current file size boundaries. This function returns 0 on a success and -1 on a failure. Note that this is the only function that can change the current offset value from which the next read or write operation can happen.

All the above operations should implement changes to the on-disk datablocks of the emulated disk suitably, by only using the functions available to you in `simplefs-disk.h`.

Note that our semantics of the offset of an open file are different from what you would have seen in Linux filesystems. In Linux, reading or writing a file automatically updates the offset of the open

file, so that future read/write operations proceed in sequence. However, our simple filesystem does not automatically update offsets. If you wish to read or write sequentially in a file, you must manually reposition the file offset using the seek function after every read/write operation.

Testing your code

The folder `testcases` contains several sample test programs to test your filesystem implementation. Each test formats the file system, and performs several operations like creating, reading and writing files. Each program prints the success/failure status of each operation performed, and the final state of the disk after all operations have completed. You may compile and link any of these test programs with your simplefs C files to generate an executable to run. The folder `expected_output` contains the output we expect your completed code will generate; this output was generated using our solution code. After you complete your implementation, you should ensure that the output of your program matches the expected output exactly.

We have provided the following shell scripts to help you in testing your code.

- `runTestcase.sh` takes one argument: a testcase file. It compiles and links this file with `simplefs-disk.c` and `simplefs-ops.c`, creates an executable, and executes it. You may use this script to test your code with one test case at a time.
- `generateOutput.sh` takes 2 arguments: a folder with multiple testcases, and a folder in which to store the outputs. This scripts runs multiple testcases and stores the outputs, and can be used to test your code with multiple testcases in one shot.
- `autograder.sh` takes two arguments: a folder with expected testcase outputs, and a folder with the actual outputs produced by `generateOutput.sh`. It compares outputs of the same testcase in both these folders and prints that the testcase has succeeded if your actual output matches the expected output.

Submission instructions

- You must submit the files `simplefs-ops.h` and `simplefs-ops.c`.
- Place these files and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Grading

We will use the autograding script (with possibly new testcases than those provided to you) to test the correctness of your code. We will also read your code to ensure that you have adhered to the problem specification in your implementation.