CS333: Operating Systems Lab
Autumn 2022, Lab 10
A Simple Filesystem (emufs)

In this lab, we will build a simple file system to handle files on an emulated disk. You are provided with files emufs-disk.h and emufs-disk.c that emulate the disk, and provide functions to access the disk. Using these emulation functions, you will implement basic filesystem operations like opening, reading, writing, and deleting files and directories, by modifying the files emufs.h and emufs-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 must refer to the files emufs-disk.h, emufs.h and emufs-disk.c when reading this description.

The disk of the emufs filesystem is emulated using text files on disk. We call these files ***mount points*** and store these (mounted) mount points in an array of **mount_t struct**.

The emulated disk consists of at most 64 blocks of size 256 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 as well as summary of all the inodes. The next 2 blocks of the disk store inodes, with 16 inodes per block. That is, the filesystem can store a total of 32 entities (files & directories) on disk. The remaining blocks are available to store file data.

The inode structure is 16 bytes in size, enabling the filesystem to pack 16 inodes into a disk block. There are only 4 mappings allowing only 1024 bytes per file and 4 entities per directory. Inode 0 is the inode for the root directory.

You can find the description of the following structs in emufs-disk.h:

- superblock_t
- inode_t
- metadata_t
- mount_t

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

- struct mount_t* opendevice(char* device_name, int size) opens and sets up a device for emulation via a file (called device_name). It also attaches the

opened device to one of the available mount points. A mount point emulates a pathname to disk mapping.
When a device/disk is opened for the first time the function creates a file to emulate the disk and initializes the super block on the disk with device_name, disk_size, and magic_number.

- int readblock(int dev_fd, int blocknum, char* buf) emulates reading a block from disk, identified by the device number (file descriptor) dev_fd and a memory region (buf). The size of the buffer is assumed to be one block.

- int writeblock(int dev_fd, int blocknum, char* buf) emulates writing a memory buffer to a block. The size of the buffer is assumed to be one block.

- void closedevice(struct mount_t* mount_point) closes the device (file used to emulate the device) and removes device from the corresponding mount point. When a device is closed, further accesses to the files/directories stored on the device are not allowed so the corresponding entries in directory and file handles are also closed.

- void mount_dump() prints information about the mounted devices.

- void read_superblock(int mount_point, struct superblock_t* superblock) reads the superblock of the device into the provided buffer.

- void write_superblock(int mount_point, struct superblock_t* superblock) updates the superblock of the device with the provided buffer.

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 modifies the text file (emulated disk) at any time. Our emulation code does not handle any concurrency when editing the emulated disk file.

You need to implement these functions in emufs-disk.c:

- int alloc_inode(int mount_point)
- void free_inode(int mount_point, int inodenum)
- void read_inode(int mount_point, int inodenum, struct inode_t* inodeptr)
- void write_inode(int mount_point, int inodenum, struct inode_t* inodeptr)
- int alloc_datablock(int mount_point)
- void free_datablock(int mount_point, int blocknum)
- void read_datablock(int mount_point, int blocknum, char* buf)
- void write_datablock(int mount_point, int blocknum, char* buf)

The description of these functions can be found in emufs-disk.c file

When you use the above functions in your filesystem code (emufs-ops.c), 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 emufs.h and must be implemented in emufs-ops.c.

Unlike Linux, our simple filesystem has separate functions to create and open a file/directory. Creating a file only creates it on disk, while opening a file opens an already created file for reading and writing. emufs-ops.c defines additional in-memory data structures related to the filesystem, called the file handle and directory handle arrays. 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 and of open directories in a directory handle. The filesystem maintains an array of all such file handles and directory handles of open files/directories, and the index of a file/directory handle in this array is used to uniquely identify an open file/directory in all subsequent operations.

The following functions are already implemented by us:

- int create_file_system(int mount_point, int fs_number) sets up the file system emufs on the opened disk attached to the mount point. Fs-number represents a file system, e.g., 0 is emufs with non-encrypted content and 1 is emufs with encrypted content.

- void fsdump(int mount_point) displays details from the metadata block regarding the directory structure on the disk attached to a mount point. For output format refer to sample outputs.

You need to implement these functions in emufs-ops.c:

- int open_root(int mount_point)
- int change_dir(int dir_handle, char* path)
- int open_file(int dir_handle, char* path)
- int emufs_create(int dir_handle, char* name, int type)

- int emufs_delete(int dir_handle, char* path);
- int emufs_close(int handle, int type)
- int emufs_read(int file_handle, char* buf, int size)
- int emufs_write(int file_handle, char* buf, int size)
- int emufs_seek(int file_handle, int nseek)

The description of these can be found in emufs-ops.c file.

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 emufs-disk.h.

Encrypted File System

In this part you'll implement the encrypted version of emufs. In this version we store the datablocks and metadata in encrypted form as defined in the encrypt function in emufs-disk.c. And we retrieve the data by decryption using the decrypt function. When you create or mount an encrypted emufs, you'll be prompted to provide a key that'll be used for decryption and encryption.
All the blocks except the superblock are encrypted. Only the magic number in the superblock is encrypted.

You need to update the following functions using the directions provided in emufs-disk.c:

- open_device(char* device_name, int size)
- int read_superblock(int mount_point, struct superblock_t* superblock)
- int write_superblock(int mount_point, struct superblock_t* superblock)
- int write_inode(int mount_point, int inodenum, struct inode_t* inodeptr)
- int read_inode(int mount_point, int inodenum, struct inode_t* inodeptr)
- int write_datablock(int mount_point, int blocknum, char* buf)
- int read_datablock(int mount_point, int blocknum, char* buf)

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 emufs 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.

You can compile and run your code with a single testcase in the following manner (where testcase.c is your testcase of interest):

```
gcc testcase.c emufs-ops.c emufs-disk.c
./a.out
```

Testcase1 and Testcase2 are for testing files only. You can just implement the file related functions along with encryption. If you implement the directory functions consider it a bonus. Testcase3 and Testcase4 are for this. Note that Testcase4 should be preceded by Testcase3.

Submission instructions

- You must submit the files emufs_disk.c, emufs_disk.h, emufs.h and emufs-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 lab10-<rollno>.tar.gz lab10` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.