CS 333: Operating Systems Lab
Autumn 2018

Lab #11: who let the files out?

Goal

In this lab we will build a simple file system to handle files on an emulated disk.

faking the disk a.k.a disk emulation

The first task is to setup a disk interface backed up (emulated by) a file. Access to blocks on the disk is emulated via re-positioning the file offset and reading/writing regions of the file. The first block on the disk is the superblock which consists of the disk name, filesystem type, disk size and a bitmap of free blocks in the disk. The second block is the metadata block which stores of the `inodes`, which in turn contain details of eachi file—file name, size and blocks corresponding to the file. We will be dealing only with files (not directories) in this lab.
Further, assume the following:

- Block size is fixed at 512 bytes
- A maximum of 10 files can be present on the disk
- Each file can occupy at most 4 blocks i.e., file size cannot be more than 2048 bytes
- Size of meta data (inodes) block is one block
- The disk has at most 42 blocks (40 data blocks for files + 1 superblock + 1 metadata block)

The following functions to emulate access to the disk are provided in `disk.c`.

1. `struct mount_t* opendevice(char* device_name, int size)`
   This call *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`.

2. `int readblock(int dev_fd, int blocknumber, char* buffer)`
   The file system will use this call to emulate reading a block from disk, identified by the device number(file descriptor) `dev_fd` and a memory region (`buffer`). The size of the buffer is assumed to be one block.

3. `int writeblock(int dev_fd, int blocknumber, char* buffer)`
   The file system will use this call to emulate writing a memory `buffer` to a block. The size of the buffer is assumed to be one block.

4. `int closedevice(struct mount_t * mount_point)`
   This call 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 stored on the device are not allowed.

5. `void mount_dump(void)`
   Prints information about mounted devices.

The following data structures are provided in `emufs.h`

1. `superblock_t`
2. `inode_t`
3. `metadata_t`
4. `file_t`
5. `mount_t`

The emufs filesystem

Implement the file system functions and APIs declared in `emufs.h` in the file `emufs.c`. The header files also contain declarations of important files system objects.

Note:

1. Implementation of `emufs` file system functions and APIs should only use the above mentioned disk emulation functions to access the emulated disk.
2. Applications issue read/write requests to the file system using a file handle (`struct file_t`) which internally stores the `inode` number of the file, the offset in the file, and the `mount point` of device where the file is stored. Read/write operations refer to content using an implicit file offset, which has to be mapped to an appropriate block on the emulated disk.
3. the `emufs` file system does **not** store any in-memory file system objects, all the accesses to the super block and metadata block are from the emulated disk.
4. All the errors should be handled properly.
5. Make sure there are no memory leaks.

The `emufs` file system interface is follows:

1. `int create_file_system(struct mount_t* mount_point, int fs_number)`
   This call 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 (data and metadata) and 1 is emufs with encrypted content.
2. `struct file_t* eopen(struct mount_t* mount_point, char* filename)`
   This opens an existing file, `filename`, stored on the emulated disk (attached to a mount point) or creates the file if it does not exist. The function returns a pointer to a file handle (`struct file_t`), and is used to refer to the file.
3. `void eclose(struct file_t* file)`
   This closes an open file and frees any memory associated to it. This does **not** delete the file on disk.
4. `int eread(struct file_t* file, char* data, int size)`
   This call reads `size` bytes of data from the file represented by `file` into the `data` pointer. The bytes are read starting from the file offset stored in `struct file_t` pointer. The return value is number of bytes read.
5. `int ewrite(struct file_t* file, char* data, int size)`
   It writes up to `size` bytes from the buffer pointed `data` to the file referred by the file handle `file`. The bytes are written starting from the current offset stored in the file handle and the offset incremented by the number of bytes written to the file. This function should return the number bytes written to the file.

   If data is already present at the current offset, it will be overwritten.

   Partial writes are not allowed in case the file size exceeds more than 4 blocks or the number of free blocks is less than the number of additional blocks required.
6. `int eseek(struct file_t *file, int offset)`
   `eseek` function changes the offset in the file handle of an open file referred by `file` to `offset`.
7. `void fsdump(struct mount_t* mount_point)`
   Displays details read from the metadata block regarding files on the disk attached to a mount point. For output format refer to sample outputs.

Brief description on how to implement each of these is given in `emufs.c`

Implementation notes:

1. All the read and write operation on the file are in the multiple of blocks i.e., the the value of argument `size` passed to `eread()` and `ewrite()` functions is multiple of 512.
2. The read and write requests are issued at block boundaries i.e., the file offset is always a multiple of 512 when these requests are issued.
3. Blocks are assigned to files in an on-demand manner, and not statically allocated.

Hints: Some helper functions which you may find useful (declared in emufs.c):

1. `struct superblock_t* readSuperblock(...)` - read the superblock from disk and return it.
2. `int writeSuperblock(...)` - write the superblock to disk
3. `struct metadata_t* readMetadata(...)` - read the metadata block from disk and return it
4. `int writeMetadata(...)` - write the metadata block to disk

You can run the sample testcases provided for this part and check you implementation by comparing with sample outputs. However, we will test your implementation with more exhaustive testcases. So you should write your own testcases and run on them as well.

Submission Guidelines

- All submissions via moodle. Name your submission as: <**rollno-lab11**>**.tar.gz**
- The tar should contain the following files in the following directory structure: <`rollno-lab11`>/
  |__emufs.h
  |__emufs.c
  |__disk.c
  |__outputs/
  |____ testcase source files and outputs

- **Deadline: Monday, 29th October 2018 - 05:05 PM**.

extra cheese, anyone?

1. `int etruncate(struct file_t *file, int size)`
   This function is used to set the size of a file to `size`. If intended size is less than current size, the truncated data is lost. If size is greater, file size in increased and padded with zeros. The size and data block bitmap should be updated appropriately, and the new size of file is returned on success.
2. Update implementation to support read and write operations at any offsets and of any size (not only at block boundaries and of block-level granularity).