Project 4

CS6456 Operating Systems F16
15 November 2016
Anudeep Konda & William Young

1 Introduction

The outcome of this project is a mini file system implemented atop a virtual disk. To that end, we have developed a library offering a set of basic file system calls that allows the user to manipulate the file system similar to how one would interact with conventional library methods (open, read, etc). In addition to file data, the file metadata is stored on disk. Section 2 describes our design and methodology behind the virtual disk. Section 3 commences our discussion of the actually library system calls, beginning with `fs_make`, `fs_mount`, and `fs_dismount`; sections 4-6 continue the discussion, delving into open/close, read/write, and truncate/seek functionality.

2 Disk Design and Methodology

The virtual file system implemented in this report is based on a virtual disk consisting of 128 16-byte blocks (2 megabytes total). The first 64 blocks are reserved solely for metadata, inode tables, and various bitmaps and offset pointers, while the latter 64 blocks store actual file data. We have implemented a number of data structures on disk to represent various features of a real disk.

2.1 Metadata

A superblock, stored in block 0 of the file system, contains pointers to each element of the file system metadata, including the inode bitmap, data bitmap, directory entries, inode-data block mapping, and data region. The superblock is initialized and written to disk upon disk creation.

Each directory entry spans a single block of the metadata region, consuming 8 blocks total (1-8). A directory entry includes the filename, inode pointer, file length, and status (busy or free) of its associated data file. For simplicity, the file system stores all files in a single root directory on the virtual disk, supporting up to 8 individual files (though this number may decrease based on disk usage). There is no explicit limit on filesize, thus a single file of up to 1024 bytes may be accommodated, saturating the entire data block region (64 x 16 bytes = 1024 bytes). All metadata associated with a file is purged upon deletion.

An inode bitmap field, stored in block 9 of the file system, represents the availability of each inode pointer as a single bit (0 for available, 1 for busy) in an 8-bit integer. When a file is created, the first available free inode pointer is allocated to the file (assuming 8 files do not already exist in the directory). Each inode maps to an inode region in the inode-data block map. inodes are freed upon file deletion.

A data bitmap field, stored in block 10 of the file system, represents the availability of each data block as a single bit (0 for available, 1 for busy) in a 64-bit integer. Each time a block is allocated to a file, the first available free data block is consumed. When the disk no longer contains available blocks, each of the 64 bits in the data bitmap is set to 1.

The inode-data block map ("the mapping") provides the key functionality linking the inode and data bitmaps to the actual allocation of data blocks. The inode number of each file points to a 64 byte region in the mapping capable of storing the block addresses of up to 64 blocks, each addressed via a single byte. When a file is in need of an additional data block, it checks the data bitmap for the first available block. If one is found, the address of that block, represented by its bit position within the bitmap, is stored as an integer within the file's inode mapping. The mapping makes it quite simple to access any number of a file's discontiguous blocks in the correct order; the implementation simply consists of an iteration over each data block, at which point a read/write operation may commence.

File descriptors are stored in a local open file table (not written/stored on disk). When a file is opened via `fs_open`, its file offset (initially 0), directory index, and file descriptor (first available) are stored in a 4-entry table. All methods dependent upon a valid file descriptor reference this table before performing requested operations. When a file is closed, its OFT entry is purged.

3 Make, Mount, & Dismount

3.1 **`make_fs()`** – *W. Young*

This function creates the file system. Our implementation of this routine checks if a proper disk name is given to it as a parameter and then creates a new virtual disk by using the provided make_disk() routine. On successful creation, the disk is opened (using open_disk()) and a file system is created on the disk, i.e. the superblock mentioned in the above disk design is initialized and is written back to the disk (using block_write()) after which the disk is closed (using close_disk()).

3.2 `mount_fs()` – *W. Young & A. Konda*

A call to this function should be made before being able to use the file system. It first checks if the disk exists and if it does, the disk is opened. Then it 'mounts' the file system, in our implementation this means that a copy of contents pertaining to metadata of files on disk is created in local memory. To be specific, the super block, directory, inode bitmap, data bitmap and the inode regions are loaded into the local memory. Also, an open file table is created. All manipulations done to the files and the file system are first recorded in local memory and are written back to disk at appropriate time (unmount, file deletion, etc). Data is read from disk only using the provided block_read() routine.

3.3 `dismount_fs()` – *W. Young & A. Konda*

This function is called when the file system is no longer needed. However, this routine doesn't delete the file system, it is just dismounted and can be mounted later. In our implementation, this means that the copy of metadata created in mount_fs() should no longer exist in local memory. Before freeing the allocated data for the metadata structures in local memory, all of it is written to the disk using the block_write(). If this filesystem is mounted back at some point in future, it will still reflect all changes that were made to the filesystem till the moment in was 'dismounted'.

4 Create, Open, Close, & Delete

4.1 `fs_create()` – *W. Young*

This function creates a new empty file in the root directory (the only directory) of the file system that is currently mounted. Our implementation of this routine first checks if the name of the file is valid. Then the directory is checked in order to make sure that no other file that was already created has the same name. A check is also made to see if there are already eight files created, i.e if the directory is full. If the name is invalid or a file with same name already exists or the directory is full, then the function returns -1, indicating failure in file creation. If not, the file is created and an available inode is assigned to it. Changes are made in the inode bitmap to indicate this and the status of flag in the directory entry for this file is set to 1. The function returns zero on success.

4.2 `fs_open()` – *A. Konda*

A call to this function opens the file. First, the open file table is checked to know if there are four files already opened, in that case the opening of file fails returning -1. If there is an empty slot in the open file table and the filename is valid, then the empty slot's status is set to 1, indicating that it is no more empty and the directory index of the file and offset are loaded into the open file table.

4.3 `fs_close()` – *A. Konda*

This function is called when a file is no longer needs to be accessed. However, it doesn't delete the file, it just closes it. In our implementation this means that the entry in open file table corresponding to this file is deleted. Also, the directory is written back to disk once the file table entry is cleared. It returns -1 on failure and 0 on success.

4.4 `fs_delete()` – *W. Young*

This function actually deletes the file and its contents are gone forever. After validating the file name, this function frees the inode that was assigned to the file and make changes in inode map to indicate the same. The data blocks allocated to this file are cleaned and are added to the empty block list and the same reflects in the data bit map. Then, the data bitmap, inode map are written back to disk. Also, the directory entry of the file is deleted. It returns -1 on failure in file deletion and 0 on success.

5 Read & Write

5.1 `fs_read()` – *A. Konda*

This function adds the functionality of reading a file to the filesystem that was developed. It first checks if the filename is valid and that the file exists. It also checks if the file is opened, because a file needs to be opened before it can be read or written. Then, the appropriate data block are found and read using block_read(). The data that was read is stored in a temporary buffer. This is done because the disk can be read only in multiples of 16 bytes. If the read function is to read some other number of bytes, the trimming is done accordingly on the temporary buffer before loading the data into the buffer provided to the routine. The temporary buffer is then freed. The file offset is set to the last byte read as further read calls are to continued from that location. On failure it returns -1 and on success 0.

5.2 `fs_write()` – *W. Young*

This function can be used to write data into the file created on our file system. After making sure that the filename is valid and that the file exists, it finds the corresponding inode block and then the data blocks. The data is written into the data blocks until the write buffer is empty. This writing starts at the offset location, i.e the file pointer's current location. All of the writing is done using the block_write() function provided. Also, file size is updated after the writes. On failure it returns -1 and on success 0.

6 Truncate, Lseek, Getfilesize

6.1 `fs_truncate()` – *A. Konda*

This function reduces the file size to the required amount given to it as a parameter. The data that fits in the final size is kept and all the rest in thrown away, i.e. cleared. After making sure that the filename is valid and that the file exists, it calculates the number of blocks of data that is to be kept and number of partial blocks (less than 16 bytes) to be kept. The data block of the file are found using the information in the inode and all the blocks exceeding the number calculated in previous step are erased. The file offset is set to the last byte. So, if someone tries to read the file just after truncating it, nothing will be returned. The partial block truncation is handled loading the whole block into a temporary buffer, truncating it to required size and then writing it back. This can't be done directly on a data block because writing into data blocks can only be done 16 bytes at a time. Also, the filesize is updated. On failure this function returns -1 and on success 0.

6.2 `fs_lseek()` – *W. Young*

This function sets file pointer to the offset to the parameter passed to it. It first checks if adding this offset will put the pointer somewhere beyond the filesize or somewhere behind 0 bytes. After making sure that this doesn't happen, it updates the file offset in the open file table to be the current offset added to the offset sent to it as a parameter. On failure this function returns -1 and on success 0.

6.3 `fs_get_filesize()` – *W. Young*

This function retrieves the size of the file whose descriptor is passed as a parameter to it. Doing this is easy as we explicitly store the file size in the directory. The appropriate index of the file in directory can be found by checking the open file table. Once we have that index, we just return the value of file size. On failure this function returns -1 and on success 0.