INF 2201
May 2022
Exam, File System

Name: Magnus Dahl-Hansen
Github Username: MagnusDH
Email: mda105@uit.no

Introduction:

This report describes the implementation of a simple file-system. The system has functions that supports creation&deletion of files and directories, writing to files, navigating between directories, viewing file&directory status and moving the file pointer within a file.
The current implementation of the file system can be run using the *shell_sim* simulator and the *Bochs* emulator.

Technical background:

Before reading this report the reader should be familiar with general basics of what a file system is: A file system defines how files are stored, named and retrieved from a storage device. The file system in this implementation consists of a super block, inode table, bitmap and root directory that are stored into different data blocks each being able to store 512 bytes of data. The first part of the file system is the super block.

The super block contains important information about a file system and is loaded into memory when a computer is booted up or when a file system is initialized. The super block in this implementation stores among other things a magic number that identifies it as a super block, the number of data blocks and inodes the system consists of, a pointer to the inode table and the max size a file can be.

After the super block follows the inode table. The inode table is a listing to all inodes that the file system consists of. Inodes are short for index nodes and is used to store information about a particular file such as its size, type, reading&writing position and pointers to its data blocks.

The third part of the file system is the bitmap for the inodes and data blocks. The bitmap is a data structure that keeps track of which inodes and data blocks in the file system are free and which are in use. It is used to provide free inodes and data blocks when new files are created. Following the bitmap in the file system are data blocks that store the actual data a file consists of.

Design&Implementation:

The implementation of the file system consists of 15 different functions which can create a new file system, create and delete files, directories and links, open and close existing files, write to files,

help navigating between directories, print status of a file and its contents and reading and writing contents of a data block to a storage device.

When the file system simulator is run for the first time, a check is be made to see if there already exists a file system from before. This is done by looking for a data block with a magic signature that identifies an existing super block and file system. If the super block is not found the file system will be created.
The file system is created by first fetching a data block from the bitmap and fill it with variables that defines a super block. The most important variables for the super block are initialized when the block is created such as a pointer to the inode table and the magic signature. Other variables such as the number of inodes and data blocks in the system are set to zero and will be incremented when new data blocks are needed. The inode table is initialized by setting all its entries to “available” before creating the first root inode. The root inode is of type “directory” and contains a parent directory and a current directory pointer which both points to the root inode itself. All other entries in the root inodes data block are initialized as “available”. Finally the file descriptor table is initialized by setting all its entries to “available” and the proceeding to write the super block, inode table and bitmap to disk. Writing&reading from/to the disk is performed by using the library functions taken from “usb/scsi.h”.

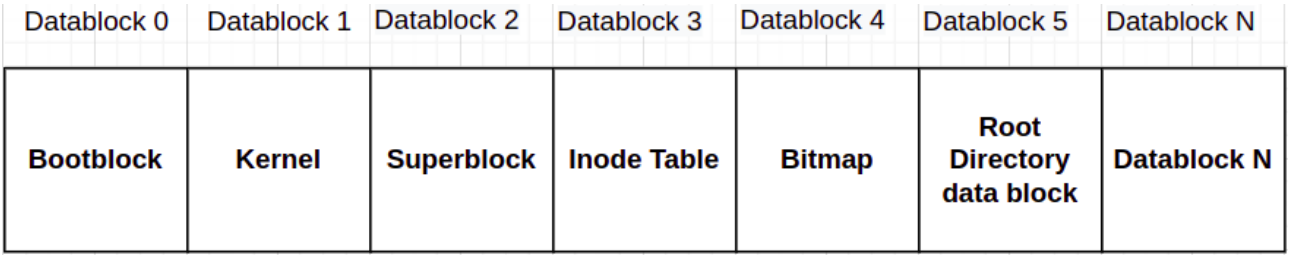

Figure 1: Layout of the file-systems data blocks

Some of the functions that are implemented takes a file name or a path string as argument. It is therefore necessary to be able to parse this string of filenames and find the inode that is needed. Name2inode is a function that takes care of this and does so by fetching a file name from the given string, searching through the data blocks of the inodes in the inode table and checking if any of the entries in the blocks match the file name. If a matching inode is found then the inode number is returned. If there are several file names in the given path then this process is repeated until the "\0" terminator of the given string is reached.

When a file is opened the inode number of the given file name is retrieved. This inode number is then placed on the first available spot in the file descriptor table. Afterwards the inode’s open_count variable is incremented and the index where the inode was placed in the descriptor table is returned. Should it be the case that the file name given to the function does not exist a new file will be created. This file will then be of the type "file" and will have its own inode and data blocks depending on how much is written to the file. The new inode is then added to the Inode table&file descriptor table and a new entry is added to the current directory’s data block. When a file is closed a index in the file descriptor table is given. The information on this index is then reset to the original status and the corresponding inode's open count is decremented.

When reading from a file, the content being read will be differ from whether the file is of type "directory" or "file". If the file is of type "directory" only one directory entry is read at a time. In addition, the file's position pointer will be incremented so that the same directory entry will not be read more than once. If the file is of type "file" then the content that will be read are characters. A given number of bytes from the file will therefore be written to a buffer and returned. The position pointer in the file will also be incremented by the same number of bytes that were read.

If content is to be written to a file it is only allowed for files of type "file". When writing to a file, it will first be checked if there is enough space in the file. The files have a space limit of 4096bytes, so if the content that is supposed to be written to the file along with any existing file contents exceeds this limit an error will be returned to the user. If there is enough space however the buffer is written to the file from the position of its position pointer.

fs_mkdir is the function that takes care of creating a new file of type "directory". This function will proceed to check if there already exists a file with an identical name, if there is room for a new inode in the inode table and that the data block of the current directory has room for a new entry. If all these checks have been passed a new inode of the "directory" type is created, a data block with directory entries is assigned to the inode and current and parent directories are set as directory entries. Lastly the the new data block, the current working directory data block and the inode table are updated to disk.
fs_rmdir will, unlike fs_mkdir, remove a directory. Should the case be that the current working directory is the root directory and the user tries to remove the parent or current directory, the user will not be allowed to do so. If the file name that will be removed exists, the entry that this file takes up in the parent directory and the inode table will be removed. The function proceeds to use the helper function "free_bitmap_entry" to free an entry in the both the inode bitmap and data block bitmap. The inode table, bitmap, and data block that contained the entry for the file are then written back to the disk.

When the user of the program wants to move from, for example, the root directory to another directory, the fs_chdir function is used. This function will go through all the directory entries that the current directory has and check if there is a directory with the same file name that the user inputs. If there is a match, the process control block's current_working_directory variable is set to be the inode of the directory that the user wants to switch to.

The user has the option to create a hard link between two files. If the user creates a link, a new file will be created so it is necessary to check if only the link file exists. The new file that is created will share the same inode as the link file but will get a different file name. If only the link name file exists it will be checked whether there is room for a new entry in the current directory's data blocks. Should there be space, a new entry will be added with a new file name that will share the same inode as the link file.
The user can also remove a link. If the file that the user wants to remove exists, its entry will be removed from the parent directory's data block. Should the file not have any other files linked to it, it will be deleted. The file's inode will be removed from the inode table and its data blocks and inode entries in the bitmap will be freed as well.

The last two functions allow the user to view details of a file and also move its position pointer. Each inode contains information about a file's type, size, file name and number of links. These variables are written to a buffer which the simulator will print to the screen and show the user the status of the given file.
The user has three options of how to move the files position pointer. The pointer can be moved from a given starting point with a given offset. A switch case function is used to move the pointer from either from the beginning of a file, from the current position of the position pointer or from the end of the file depending on the user input.

Results&Discussion:

The implementation has been run and tested on a HP laptop with the following specs: AMD Ryzen 5 2500U processor, Linux operating system, 8GB RAM, using the shell_sim simulator.

The implemented code is able to execute all the required system calls when using the *Bochs* emulator or the *shell_sim* simulator. A user of the program will be able to use "ls" to view the contents of the current directory, create and remove directories by using the "mkdir" and "rmdir" commands. The user can also create and write to new files using the "cat" command and will be able to change current directories by using the "cd" command. Linking files can be done by typing "ln" with a name of an existing file followed by the name of a new file. Removing a link or deleting a file can be done by typing "rm" followed by a filename. Finally the "state" and "more" commands will print out information about a file and its contents.

The program can be compiled and run in the *Bochs* emulator but a known bug will occur when printing the contents of a file. When reading from a file the contents will be printed out in a correct way, but in different cases some of the characters will be of a different format or extra letters of this unknown format will be added to the existing contents. This problem has not been resolved when using *Bochs* but will not be a problem when using *shell_sim.*

Another known bug occurs when writing to a file from another directory that is not root. If the user creates a new directory, changes directory to the new one and tries to write to a file within this directory then stack smashing will occur. Why this is the case is uncertain, however, this will not be a problem when using the *Bochs* emulator. In this emulator the creation of a new file within another directory and printing its contents is not a problem.

Lastly, trying to run the given python tests will result in only some of them finishing correctly. Why some tests will fail is not certain, but taken in consideration that this implementation does not support mounting of the current file system may have an impact on how some of the tests performs.

Conclusion:

This report described how 15 different functions were implemented to be able to create a new file system and to use it in a very simple way. The implementation of the file system is tested, debugged and run using the *shell_sim* simulator. The code can also be compiled and run using the *Bochs* emulator, but will have troubles when printing the contents of a file.