TMFS Documentation

Project Overview:

The project's aim is to implement a minimal file system on the Linux environment using the C programming language, based on UFS(Unix file system) to transform the theoretical knowledge in low-level systems into practical experience. Although the scope that it covers is minimal, I was able to gain a vast amount of experience and knowledge regarding file systems.

Learning and Design Process:

My journey in low-level systems began over six months ago, when I read a low-level programming book written by Igor Zhirkov, where I learned the assembly language, and a lot of information regarding this area. After that, I ventured into the realm of operating systems where I read six chapters of the book " Modern Operating Systems". It taught me about processes, threads, file management, memory management, and so on. The knowledge about filesystems of which I implemented some of in my project was taken from this book.

Error Log and Debugging Journal:

Here I will present some of the many major errors I faced while implementing this project (The original photos of the papers of which I used to write the errors on is presented as a separate file) :

. Error #1: [undefined reference for strnum].

Description: While experimenting strnum's functionality I faced this issue.

Why: At first, I thought it had to do with header inclusion, but a after tracing the headers, I concluded that nothing is wrong with them. So, I immediately deduced that it was a make file error since no implementation or header issues were found. It turned out that I forgot to include Hfunc.c in the compilation of the .so file which had caused the issue.

Solution: included it in the compilation process of the .so file.

. Error #2: [undefined reference for dinit].

Description: While experimenting strnum's functionality I faced this issue.

Why: At first, I realised that the same problem faced with strnum is reappearing here, so the first place to check was the make file, and luckily it was the last. The problem was that I had forgotten to include the -fPIC flag for position independent code in the Hfunc.o compilation line.

Solution: I placed the flag and the problem was resolved.

. Error #3: [dinit not functioning correctly].

Description: While experimenting dinit's functionality I faced this issue, where it wasn't initializing the experimental file I had made to implement dinit on.

Why: At first, I had absolutely no clue on what may be the issue, but after resorting to using fstat I discovered that it was a "no such file or directory" error, of which I was able to deduce that it is a naming error. After searching a bit, I found out that the "BasePath" macro in DiskEm.h had "/drive." rather than "/disk." as the end of the path it was meant to be a name for.

TMFS Documentation

Solution: Used the correct name of the file which is disk.1.

. Error #4: [dshow printing 0 as the number of blocks instead of 65535].

Description: While experimenting dshow's functionality I faced this issue.

Why: At first, I thought it was related to the size of the format specified in the printf function, then I traced back the source of the information that I was printing, which showed me that the issue wasn't in the dshow function, but rather in the dmount function. It turns out that when assigning the number of blocks to dd→blocks variable was 65536 instead of 65535, larger than what the variable can hold. I checked the size of disk.1 and it was correct, so I deduced that st_blocks may be including an extra block for storing metadata.

Solution: Subtracted 1 from the the "blocks" variable during assignment .

. Error #5: [reading 512 bytes from a region of size 34].

Description: While experimenting dwrite's functionality I faced this issue. After hexdumping the file, one byte was written. And the error points out that one i-node is being written, rather than a whole block.

Why: At first, I thought it was a problem with the macro itself, so, I checked the address of the byte, converted it to decimal format using the "dec" command, and deduced that the byte was in the third block and was given 2.
After thinking a bit I discovered that the issue was that rather than initialising multiple bytes as intended, I had initialised only one byte in dinit.

Solution: Added a for loop that iterates through the bytes.

.Error #5: [i-node block printing a random number on the left of the validtype filed(03)].

Description: I hexdumped disk.1 to make sure that the number of (blocks, i-node blocks, inodes, the magic number and so on) are correct after formatting the disk using fsformat. And then this random number appeared on the left of the validtype field.

Why: I started by checking the size of each of (block, i-node, and so on) by calculating the sum of their fields, and soon discovered that the i-node is occupying 34 bytes instead of 32 although the sum of its fields were 32 bytes. While learning C I had picked the wrong idea that in a struct "packed" comes before the name, although it must come after the struct is initialise .

Solution: Put "packed" after initialising the structures rather than before.

. Error #6: [I-node blocks not printed when printing the bitmap].

Description: While experimenting the i-node functions functionality I faced this issue, where the bitmap was all zeros, although 10 percent must be i-node blocks.

Why: I started by checking the "setbit_" and "getbit_" i-node functions where I discovered that in the condition of setting and unsetting bits in the "setbit" function, I had called the macros to call the function without assigning them to the pointer "byte" which had caused their value to be lost, and therefore the bitmap was zeros only.

Solution: assign the “setbit_” and “unsetbit_” macro calls to “*byte”.

. Error #7: [At least 200 error messages regarding types and declarations].

Description: This was one of the scariest errors since I didn’t consider it at all. And since at the time I didn’t have any extensions for C on vscode (I was using code-OSS on arch linux), so the only way to debug was either searching through the whole code base, or following the error messages, and since the error messages didn’t provide any useful information except the error being related to types, I was left with manual debugging.

Why: After debugging the code base, it turned out that since I hadn’t test-run the last few functions I wrote, there were a dozen syntactic errors scattered in them. And the ones that caused the catastrophe was a semi-colon in a header file followed by definitions.

Solution: Fixed the syntactic errors in the header files.

Current Progress and Modifications:

Some of the main features that the system currently supports:

. Formatting the disk drive.
. Print information about the disk.
. Return information about a file.
. Mounting the filesystem.
. Unmounting the filesystem.
. Creating i-nodes.
. Finding i-nodes.
. Allocating i-nodes.
. Unallocating i-nodes.
. Saving an i-node.
. Making a bitmap

. Allocating a bitmap
. Unallocating a bitmap

In addition to that I built a disk emulator which uses a file in the filesystem to use as a disk. I built it so I can simulate all the real disk capabilities, whether creating, destroying, formatting, and test filesystems without affecting real hardware.

I am working on adding read, write, and many file and directory functions so the filesystem can be complete. After that I may consider introducing algorithms for security (such as AES encryption method and SHA-256 for data integrity verification) and system recovery such as journaling, or copy-on-write algorithms.

Reflection and Next Steps:

This project was a big leap from theoretical knowledge to practical experience. I learned about the data structures embedded in filesystems, the dependency between different functions of different uses, and how complicated the implementation is in comparison with learning theoretical knowledge by reading a book for example. Although it is neither perfect nor complete, I consider it as a stepping stone for future projects and work.

Function-Level Documentation:

Note:. If there were two conditions, one leading to an iteration, returning, etc…, and the other leading to continue execution for the following lines of code. We choose the one that leads to the latter. And continue the explanation from there.

. The Returns field in every function signals (most of the times) the return value when execution reaches the end of the function.

TMFS Documentation

. The functions that are initialised but not used throughout the code are helper functions for future modifications.
. The explanation of the disk emulator's functions are provided in its own repository at the following page: https://github.com/dedboi777/TMDE.git

The functions:

. fsformat: This function formats the disk drive by defining the fields of each of the superblock, and i-node block types, and assigning them to their default values.

Parameters:

. A pointer to a disk that we want to format.
. A pointer to the bootsector .
. A boolean to force the format if files were open.

Functionality:

. First, it checks if we have a disk mounted or not, if yes it will return an error, else we continue.
. After that, we check that if we have open files, if yes, we check if the force boolean's value, if it is false, we return an error. If true, we we close all files in the disk.
. Then we check how many blocks do we have by assigning it from the disk structure to the local blocks variable.
. And we calculate the number of i-node blocks by dividing the number of blocks by 10 and adding one to the number of i-nodes if it isn't a multiple of 10.
. After that we zero the fields of the inode(note: we calculate the size of the pointers so we could zero out the direct pointers).
. We then create the superblock.
. While creating the superblock we check if we have a bootsector, if yes, we copy it to the superblock's boot field, if not, we zero the boot field.

. Then, we check if it is allowed to write the disk fields or not.
. Starting with the superblock, we check the return value of the write operation on the super block field that is stored in the “allowed” variable. If it is true, we continue execution, if false, it returns an error.
. Moving on to the i-node blocks, we check the return value of the write operation on the i-node blocks stored in the “allowed” variable. If it is true, we continue execution. If false, it returns an error.
. And we zero out the fsblock variable “fsb” before writing it in a for loop that writes the “fsb” and checks its return value at each iteration. If 0, it returns an error. Else, it continues.
. After finishing writing the i-nodes, we need to return a filesystem. So, we calculate the size of the “s_filesystem” structure, then we assign the return value of the “alloc” operation on the previous size to the “fs” variable. After that, we zero out “fs” and assign the drive number field of “dd” to the drive number field of “fs”(dd is a pointer to a disk structure), and “dd” to the “dd” field of “fs”.
. Moving on, we copy the address of the superblock field “super” to the the “metadata” field in “fs” and assign the return value of the “mkbitmap” function to the variable “bm”. Then we assign the sum of the sizes of the superblock (1 byte) and the “inodeblocks” field of the “metadata” field of “fs”.After that, we set the bits in the bitmap one by one.
. Finally, we assign the “bm” variable to “bitmap” field of “fs”, call “fsshow”, and return “fs”.

Returns:

A pointer to a filesystem structure.

. mkbitmap: This function creates a bitmap and scans the disk if the user wants to.

Parameters:

. A pointer to the filesystem structure.

. A boolean that determines if it is going to scan the disk or not.

Functionality:

. First, it checks if we have a filesystem or not. If yes, we continue. If not we return an error.

. Then we calculate the size of the bitmap by dividing the number of blocks in the “dd” field in “fs” by 8. And if it isn’t a multiple of 8 we add 1 to the size.

. After that, we check the result of allocating “size” as the size of the bitmap by assigning its return value to the “bm” variable. If zero, we return. Else, we continue.

. We zero out the bitmap, then check if the user wants to the scan the disk or not by reviewing the value of the “variable”. If false, we return the bitmap. Else, we continue.

. To scan the disk, we need to loop through the “inodeblocks” field in the “metadatat” field stored in “fs”. Then, we zero each block on the disk. After that, we store the return value of the “dread” operation in the “ret” variable, and checking if it is zero or not. If not, we free the bitmap and return. Else, we continue.

. Then, we loop through the inodes per block, and check if they are valid or not by storing the result of ANDing the “validtype” field in the “inode” field stored in “block” with “1” in hexadecimal format (0x01) (to get the last bit of the i-node which is the valid field, so in other words we check if the i-node is valid or not) in the variable “valid”. If true, we set the current index of the bitmap as true by calling “setbit” with the true parameter. Else, we set it as false by calling the same function but with the false parameter. And at the end of each iteration, we increment “index”.

. Finally, we return the bitmap.

Returns:

A bitmap.

TMFS Documentation

. bitmapalloc: This function allocates a bit in a bitmap by looping through the bits and setting the first free one.

Parameters;

. A pointer to a filesystem structure.

. A pointer to a bitmap we want to allocate the bits from.

Functionality: (No need for explanation).

Returns:

If successful: “blockNum”.

If unsuccessful: 0.

. bitmapfree: This function frees a bitmap by setting the index to false.

Parameters:

. A pointer to a filesystem structure.

. A pointer to the bitmap we want to free the bits from.

Functionality: (No need for explanation).

Returns:

void

. fsshow: This function prints the following information about the disk:

. What disk is mounted where.

. The total number of blocks, 1 superblock, i-node blocks and how many i-nodes they contain.

. The following data about each i-inode:

. Valid or not.

. Its type.

. Name.

. Size in bytes.

. The bitmap

Parameters:

. A pointer to a filesystem structure.

. A boolean that determines if it is going to print the bitmap or not.

Functionality: (No need for explanation)

Returns:

void.

. fsstat: This function returns the following information about a file by returning a “s_fileinfo” struct pointer:

. i-node number
. The size

Parameters:
. A pointer to a filesystem structure.
. The i-node number.

Functionality:
. First, it checks if we have a filesystem or not. If we do, we continue. If not, we return an error.
. We then check if the i-node was found by assigning the result of doing a search for the i-node (What “findinode” returns) to the “ino” variable, and checking if it is zero or not. If it is, we return an error. If not, we continue.
. Then we assign the size of the “s_fileinfo” struct to the “size” variable, and the return value of calling “alloc” to the “info” variable.
. We then vheck the “info” variable if it is zero or not to know whether the allocation succeeded or not. If it did not, we return an error. If it did, we zero out “info” and continue.
. Then we assign the i-node number passed as the second parameter to the i-node number field of “info”. And the size of the local i-node to the “size” field of “info”.
. Finally, we free “ino” and return “info”.

Returns:
Pointer to a “fileinfo” structure.

. fsmount: This function mounts the filesystem.

Parameters:
. The drive number

Functionality:
. First, it checks if the drive number is larger than the max drive numbers. If true, it returns 0. Else, it continues.
. Then, we assign “driveNo – 1” to the index(“idx”) that will be the index in our global file descriptor structure.
. After that, we assign the global disk descriptor of “idx” to

the disk pointer(“dd”) and check if it is equal to zero or not. If it is, we return 0. If not, we continue.

. Moving on, we assign the sizeof the “s_filesystem” structure to the “size” variable, and the return value of the allocation of size “size” to “fs”. Which we then check if it is zero or not to make sure the allocation succeeded. If true, we return 0. If false, we continue and zero out “fs”.

. After that, we assign the local “driveNo”, and “dd” to the “driveNo” and “dd” fields of the “fs” respectively. And assign the return value of “mkbitmap” to the “bitmap” field of “fs”. Which we then check if it worked or not by checking if it is zero or not. If it is, we free “fs” and return 0. Else, we continue.

. Then, we “dread" 512 bytes from the “dd” field of “fs” into the address of the “metadata” field of “fs” and check if the return value is zero or not. If it is, we free “fs” and return 0. If not, we continue.

. Moving on, we call “kprintf”(A macro in Mystd.h) the drive that we mounted, ‘c’ for 1, ‘d’ for 2, and ‘?’ for anything else.

Returns:

Pointer to the filesystem structure.

. fsunmount: This function unmounts the filesystem.

Parameters:

. A pointer to a filesystem structure.

Functionality:

. First, it checks if we have a filesystem or not.

. Then, it assigns the drive number field of the field “dd” that belongs to “fs” to the local “driveNo”. After that, we assign “driveNo – 1” to the index(“idx”) that will be the index in our global file descriptor.

. Moving on, we assign 0 to the global disk descriptor of “idx”, and free “fs”.

. Then, we kprintf(A macro in Mystd.h) the drive that we unmounted. 'c' for 1, 'd' for 2, and '?' for everything else.

. Finally, we return.

Returns:

void

. findinode: Searches for an i-node.

Parameters:

. A pointer to a filesystem structure.

. A pointer.

Functionality:

. First, it checks if we have a filesystem or not.

. Then it assigns 0 to "ret".

. After that, we loop through the "inodeblocks" field of the "metadata" field of "fs"(we start from 2 because the first i-node block starts there). Where we start by zeroing out the block. Then, we "dread" the "dd" field of "fs" into the address of the "data" field of "bl", where we store the return value in the "res" variable that is later checked if it equals zero or not to make sure the operation succeeded. If it returns false, we return "ret". Else, we continue.

.Moving on, we enter another loop that goes through every i-node in a block. It starts by comparing the variable "n" to the index of the i-node that we are looking for. If it finds it, we enter the condition's body. Else, we iterate again.

. In the body, we begin by assigning the size off the "s_inode" struct to the "size" variable, and the return value of the "alloc" function(taking "size" as its argument) in the "ret" variable. Which it then checks if it is equal to zero or not to make sure that the allocation operation worked successfully. If true, we return 0. Else we continue by zeroing out "ret".

.After that, we copy the address of the inode at index "y" field that belongs to "bl", and we assign the "inodeblocks" field of the “metadata” field that belongs to “fs” to the variable x, then we break out of the loop.

. Finally, we return “ret”.

Returns:

Pointer to the i-node structure(“ret”).

. increate: This function creates an i-node by initialising its fields one by one.

Parameters:

. A pointer to a filesystem structure.

. A pointer to a filename structure.

. An enum.

Functionality:

. We start by assigning the “Clean” enumerator to the “errnumber” global variable.

. Then, we check if any of the “fs”, “name”, or “filetype” variables are 0. If true, we return an error. Else, we continue.

. After that, we assign the return value of calling “validfname” to the variable “valid”. Of which we then check if it is equal to zero or not. If true, we return an error. Else, we continue by assigning the return value of “inalloc” to the variable “idx”.

. Moving forward, we assign the size of the struct “s_filename” to the variable “size”, and the return value of calling “alloc” with “size” as its argument to “ino”.

. We then assign the “filetype” variable to the “validtype”, field of “ino”, and 0 to the “size” field of “ino”.

. After that, we (again) assign the size of the struct “s_filename” to the variable “size”. And we copy the “name” variable to the “name” field of “ino”.

. Moving on, we assign the return value of the function “fssaveinode” to the “valid” variable. Then we free “ino” and check if the “fssaveinode” call had succeeded or not by checking if the “valid” variable is 0 or not. If it is, we return an error. Else, we return “idx”.

Returns:

A pointer to the newly created i-node(“rdx”).

. inalloc: Allocates an i-node by searching until it finds a free i-node to allocate.

Parameters:

. A pointer to a filesystem structure.

Functionality:

. First, it checks if we have a filesystem or not.

. Then we assign the maximum number of i-nodes to the variable “max”.

. After that, we enter a loop where we start by assigning the return value of the “findinode” functoin call to “p”. Which we then check if it is zero or not. If it is, we break. Else, we continue.

. Moving on, we check if the i-node is valid or not by storing the result of ANDing the “validtype” field in the “inode” field stored in “block” with “1” in hexadecimal format (0x01) (to get the last bit of the inode which is the valid field, so in other words we check if the i-node is valid or not). If zero, we continue to enter the if’s body. Else, we iterate again.

. After entering its body, we first assign “n” to “idx”. Then we assign 1 to the “validtype” field of “p”.

. After that, we check the result of calling to save the i-node through the “fssaveinode”. If zero, we assign 0 to “idx” and break. Else, we continue.

. Moving on, we increment the number of i-nodes by incrementing the “inodes” field in the “metadata” field belonging to “fs”. Then, we perform a “dwrite” operation where we write the “dd” field of “fs” to the “metadata” field of “fs” and break.

. After that, we check if “p” is equal to zero or not. If it is, we free it. Else, we continue execution.

. Finally, we return “idx”.

Returns:

A pointer.

. inunalloc: Unallocates an i-node specified by the use after searching for it.

Parameters:

. A pointer to a filesystem structure.

. The i-node the user wants to unallocate.

Functionallity:

. First, it checks if we have a filesystem or not.

. Then it assigns the result of searching for the wanted i-node using “findinode” to “p”. Which is later checked if zero or not to make sure the searching was successful. If zero, we return false. Else, we continue execution.

. After that, we assign zero to the “validtype” field of “p”.

. Moving on, we store the result of calling “fssaveinode” in the “blockNo” variable to save the i-node, and then we free “p”.

. Finally, we return true if “blockNo” is other than zero. Else, we return false.

Returns:

A boolean that tells if the saving operation was successful or not.

. fssaveinode: Writes an i-node by updating the i-node and then writing it.

Parameters:

. A pointer to a filesystem structure.

. A pointer to the i-node we want to write.

. The i-node number.

Functionality:

. We start by checking that we have a filesystem and an i-node. If we do, we continue. Else, we return 0.

. Then we divide the i-node number by 16, add 2 to the result, that is stored in the “blockNo” variable.

. After that we assign “idx” MOD 16 to the variable “mod”.

. Moving on, we print “blockNo” and assign the result of dreading the “dd” field of “fs” into the “data” field of “bl”

to the variable “ret”. Of which we then check if it is equal to zero or not to make sure the “dread” operation was successful. If it equaled zero, we return 0. Else, we continue.
.Then we assign the size of the structure “s_inode” to the “size” variable. And then copy the pointer to the i-node that we want to write to the address of the i-node of “mod” that is a field of “bl”.
. After that, we assign the result of dwriting the “dd” field of “fs” into the address of the “data” field of “bl”. Of which we check if zero or not to make sure the “dwrite” operation was successful. If true, we return 0. Else, we continue.
. Finally, we return the “blockNo” variable.

Returns:

A pointer to the i-node.

. strnum: Concatenates a string and a single-digit integer, by storing the string in a buffer, adding 0x30 to the integer to convert it to ASCII format, and then add the number after the string in the buffer (Note: We converted it to ASCII so it can be concatenated with the string).

. isopen: Checks if a file is open or not by examining its posix file descriptor.

. load: Writes one character, by first checking if the file is open or not, then writing it to a buffer.

. store: Reads one character, by first checking if the file is open or not, then reading it to a buffer.

.zero: Zeroes out bytes of the memory region pointed to by the first parameter by iterating thorugh its bytes and zeroing them one by one.

. init: Handles initialisation by setting “errnumber” to zero, Calling the “setupfds” functions, and sets the “initialised” global variable to true.

TMFS Documentation

. setupfds: Sets up the file descriptors, by zeroing them, then mapping them.

. memorycopy: Copies information from source to destination. And can copy a string if user specifies.

. stringlen: Calculates the length of a string.

. getbit: Gets a certain bit from a byte (used in bitmaps).

. setbit: Sets a certain bit a specific value (used in bitmaps).

. tolowercase: Makes the string passed lower case.

. low: Makes a character passed lower cased by a condition that checks if the letter is valid by checking the ASCII representation of the letter, and an operation done on a new character representation.

. findchar: Searches for a character by taking the length of the the "haystack", then it searches for the the character specified. It can search both forwards and backwards for flexibility.

. cmd_format: A command-line wrapper for the "fsformat" function that handles user operations before calling the main formatting routine.

. usage: Gets called depending on a certain condition in "cmd_format" to print an error message and then terminate the program.

. usage_format: Gets called depending on a certain condition in "cmd_format" to print an error message and then terminate the program.

. file2str: Transforms the filename into a string by copying both of the filename's fields to the string with a dot between them.

TMFS Documentation

. str2file: Transforms a string into a filename by detecting both of its fields (name, and ext) in the string (by finding the “.” that separates them), and copies them to “name” and “ext” fields of “name”.

. validchar: Checks if the character passed is one of the characters specified in the “ValidChars” macro (used for file name character validation).

. validpchar: Checks if the character passed is one of the characters specified in the “ValidPathChars” macro (used for path name character validation).

. validfname: Checks if the filename is valid or not by checking the validation of its two parts (name, and extention).

. validpname: Checks if the pathname is valid or not by checking the validation of its name.
Functionality:

. parsepath: This function parses a path by recursively by :
. Checking a condition of “str” and “pptr” validity, and if “rdx” is larger or equal to “MaxDepth”, which if true, will result zeroing the “idx” of the “dirPath” field of “pptr” and returning true.
. Then we search for ‘/’ which is the symbol used to seperate directories in a path. If not found, It enters the if’s body. Else, it continues.
. After that it copies the first name from the left to the “idx” of the “dirPath” field of “pptr”, increments “idx” and assigns zero to the next “dirPath” field, and returns true.
. After exiting if’s body we assign zero to the dereferenced “p”, copy “str” to the “idx” of the “dirPath” field of “pptr”, and increment both “p” and “idx”.
. Finally, we do the recursive call.