MARÍA DEL CARMEN DÍAZ DE MERA GÓMEZ-LIMÓN - 100383384
MARINA TORELLI POSTIGO - 100383479

FILE SYSTEM

GROUP 89

PROGRAMMING ASSIGNMENT 2

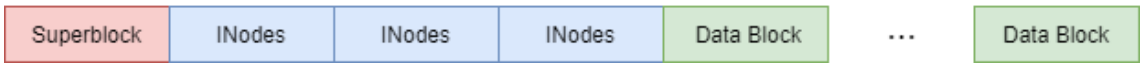
OPERATING SYSTEMS DESIGN

COMPUTER SCIENCE AND ENGINEERING
UNIVERSIDAD CARLOS III DE MADRID

REPORT

Table of Contents

1. Design



In the above image we show the structure of our file system. Firstly, we will have the superblock, followed by three blocks with the iNodes. After the iNodes the rest of the file system will be occupied by data blocks.

The superblock will contain the following information:

- The magic number used to differentiate the file system
- The number of iNodes on the device
- The number of blocks occupied by the iNodes
- The number of data blocks in the device
- The address of the first data block
- The size of the device
- The bitmap for the iNodes representing which ones are in use
- The bitmap for the data blocks
- Padding to fill a block

A requirement of the file system given by the statement is that the number of files does not exceed 48, which will be then the number of iNodes. Each iNode will contain the following information:

- The type of the iNode (which can be a regular file or a symbolic link)
- Name of the associated file or symbolic link
- In the case it is a symbolic link, the id of the iNode of the file it references
- The size of the file in bytes
- The direct block associated to the file
- The indirect blocks which can be allocated if necessary (as a file can have a maximum of 10240 bytes it will take 5 blocks maximum, so we will have maximum 4 indirect blocks)
- Whether the file includes integrity or not
- The CRC checksum calculated for the integrity

As we have a fixed number of iNodes we decided to fit the maximum number of iNodes in a block to minimize the space used, as if we used a block per iNode we would waste a lot of space for padding.

We decided to choose a fixed number of iNodes per block which would be a power of 2 but also a divisor of 48, so that all the iNodes would fit in exactly N blocks. We decided then to fit 16 iNodes per block, so we would use 3 blocks for all iNodes. With that in mind we calculated that each iNode would have a size of 2048/16=128 bytes, the block size divided by the number of iNodes per block. The iNodes still need padding to fill this size, but it will be minimized.

The bitmap of the iNodes will need 48 bits, one for each iNode, so we will use an array of length 48/8. To compute the size for the bitmap of the blocks we used the maximum size that our device could have (600KB) and divided it by the block size to compute the total number of blocks. To that number we subtracted 1 block representing the superblock and the number of blocks taken up by the iNodes (3). The result is the maximum possible number of data blocks, which will be the number of bits in the map. So, we will divide that number by 8 to get the length of the array.

We also declared some constants to be used like the magic number, the maximum length of a file name (32 given by the statement), the number of iNodes per block (16 as we explained previously) and the minimum and maximum sizes for the device (460KB and 600KB respectively as given in the statement). We also defined the types REGULAR and SYM_LINK for the iNodes (whether they are a regular file or a symbolic link).

In memory we will keep a structure for the superblock, as well as an array of all the information of the iNodes. These structures will serve to save the metadata of the file system in the main memory. We will also have a variable to indicate whether the file system is mounted (metadata is in memory) or not. We will also have another array for the iNodes, however this structure will store the following fields:

- Whether the file was opened in this session or not.
- Whether the file was opened with integrity.
- The pointer to the last byte read or written f_seek.

This structure is used only in memory in the current session and will not perdure after an unmount operation.

2. Functionality

For all the following functions, we will first check for errors in the arguments and the preconditions. For example, it will always be checked at the beginning whether the file system is mounted or not (except for the mkFS() and unmountFS() functions).

2.1. Basic Functionality

2.1.1. Device Management

- read_metadata():

  This function will be used to read the metadata from the file disk.dat and store it in the structures of the main memory. We will use the function bread to access the simulated file system block by block, checking for any error in this function.

First it will read the superblock, which can be stored directly as it fits exactly one block. Then it will read block by block the three blocks storing the iNodes, storing the full block first in a buffer. This buffer will be read using the size of each iNode (2048/16 bytes) to store each iNode in its corresponding position of the array.

- write_metadata():

  This function works exactly like read_metadata() but using the function bwrite to write the metadata from memory to the file disk.dat. It will first write the superblock and then the iNodes using an intermediate buffer to fill a block.

- mkFS(long deviceSize):

  To build the File System we will first need to check whether the argument passed by the system is within the limits of the size of the device (which we have defined as constants). This function will initialize all the values of the superblock and i_nodes as well as initialize the contents of all the data blocks.

  It will first compute how many blocks the device would have dividing its size by the size of a block. Most of the fields of the superblock are constants (for example the number of iNodes and the blocks occupied by them, which we have defined in our design), except the number of data blocks, which depend on the size of the device input by the user (We would have to subtract to the number of blocks in the disk the blocks corresponding to the metadata, which are 4).

  The value of all the iNodes and data blocks will be initially set to 0. We will also execute the command *./create_disk* with the number of blocks of the device computed before to create the file "disk.dat" that will emulate our file system. After setting all the values of the structures it will call write_metadata() to write them to the disk.

- mountFS():

  This function will first check that the file system is not already mounted. If it is not already mounted, it will do it using the function read_metadata() to load the metadata from the disk to memory.

- unmountFS():

  Similarly, to the previous function it will first check that the system is mounted, as you cannot unmount something that is not mounted. If it is, it will use the function write_metadata() to save the metadata from memory to the disk.

2.1.2. File Management

- ialloc():

  This function is used to allocate a new iNode. To do this it searches within the iNode map until it finds a node marked as free (0). It will set it as used (1) and return the index of this iNode. If there are no more free iNodes, the function returns -1.

- balloc():

  This method has the same functionality as ialloc but it searches within the data block map to allocate a new block. If there are no free blocks it returns -1.

- ifree():

  This function first check that the id of the iNode is valid. If it is, it sets its corresponding bit in the iNode bitmap to 0 and deletes the contents of the metadata and the information about the current session (inode_x), setting them to 0. Even if the iNode wasn't in use it can be freed, the contents of the memory wouldn't change as they were already set to 0.

- bfree():

  This method has the same functionality as ifree but with data blocks. It sets the corresponding bit to 0 in the block map and deletes the contents of the block in the file disk.dat, setting all its contents to 0 by means of a buffer the size of a block.

- remove_links(int inode_id):

  This function is used to remove all the symbolic links pointing to the file represented by inode_id. It will look for all the iNodes of type SYM_LINK and check if they point to inode_id. If they do, it will use the function removeLn described in section 3 to remove the link.

- createFile(char *fileName):

  The objective of this functionality is to create a new empty file in the file system. Firstly, we check for preconditions and errors:

  - There must not be another file already in the system with the same name as the one we want to create.

  Now we proceed to allocate an iNode and a direct data block for the new file. In this process we must check whether there are indeed free iNodes and blocks (using the auxiliary functions *ialloc()* and *balloc()*), otherwise the function will return an error.

Then, we can initialize all iNode values:

  - The type must be regular since it is a file and not a symbolic link.
  - We initialize the indirect blocks of the iNode to -1 since they are not being used yet.

- removeFile(char *fileName):

  The objective of this functionality is to delete a file of the system. As before, we check some preconditions for the function to run properly:

  - The file that is meant to be deleted must already exist in the system and it must be of type regular.

  Now we free the blocks that are being occupied (the direct block and the indirect ones, if that was the case) using the auxiliary function bfree().

  After that, the function remove_links() is used to remove all the symbolic links that point to the file that we are removing.

  Finally, we free the iNode the file was allocated to using the auxiliary function ifree().

- openFile(char *fileName):

  The objective of this functionality is to open an existing file of the system. First, we check the preconditions for this function to execute correctly, such as:

  - The file that is meant to be opened must already exist in the system.
  - The file must not be opened already with integrity.

  Then all there is left to do is to move the pointer f_seek of the iNode to the beginning of the file and set the iNode open field to 1.

- closeFile(int fileDescriptor):

  The objective of this functionality is to close a file in the system. As usual, we check that all the preconditions for the function are met:

  - The file descriptor, meaning the iNode ID, is valid.
  - The file descriptor must correspond to an existing iNode.
  - If the file was opened with integrity, it cannot be closed with this function, as it must be closed with integrity too.

  Finally, all there is left to do is to set the open field of the file iNode to 0, this meaning that the file is now closed.

2.1.3. Interacting with Files

- add_data_block(int inode_id):

  This function will be used to allocate a new indirect block when necessary (it will be used in the function writeFile() described in the following points). It will first check if the inode_id is valid and corresponds to a created file.

  Before adding the new block, it will first check to which indirect block it corresponds, obtaining its index by dividing the size of the file by the block size. Then it will use the function alloc() to allocate the new block and update the corresponding indirect block.

- readFile(int fileDescriptor, void *buffer, int numBytes):

  The objective of this functionality is to read a number of bytes from a file and store them in a buffer. Firstly, it checks the preconditions in the arguments:

  - The file descriptor must be valid.
  - The file descriptor must correspond to an existing iNode.
  - The size of the numByte parameter cannot be negative.

  Now for the main part of the function, we use a loop to go through every block of the file until the number of bytes we are yet to read reaches 0. To perform the read we will need the current block id (obtained with the function bmap) and the offset inside that block (the remainder of the position of f_seek by the size of a block). There could be two possible situations depending on the number of bytes to read:

  - They are smaller than the capacity of the current block (the size of the block minus the offset). In this case the function must read the number of bytes which is indicated.
  - They exceed the capacity of the current block, in which case the number of bytes to read in that block must be limited to the size of the block minus its offset.

  In both cases, the function bread() is used to read the block from the disk and then the function memmove() is used to store in the buffer the number of bytes indicated.

  After this, in every iteration of the loop, the function must update its pointers and variables, such as moving the pointers up the number of bytes read.

- writeFile(int fileDescriptor, void *buffer, int numBytes):

  This function writes the contents of the argument buffer into the file represented by fileDescriptor and returns the number of bytes

written. First it will check for errors in the preconditions and the arguments:

- The file descriptor must be valid.
- The file descriptor must correspond to an existing iNode.
- The number of bytes to be written cannot be negative.

First, we will check that the number of bytes to be written does not exceed the capacity of a file (10240) bytes. If it does, the number will be limited to the capacity of the file (the maximum size of a file minus the current position of the pointer f_seek). Before writing we will also check if we need to allocate a new indirect block. This will happen when the pointer f_seek is at the end of the file in a position bigger than 0 that is also a multiple of the block size (this would mean that the current block is full).

The main part works as the function readFile, with a loop that writes blocks to the file until the number of bytes to write reaches 0. To perform the write we will need the current block id (obtained with the function bmap) and the offset inside that block (the remainder of the position of f_seek by the size of a block). There could be two situations with the number of bytes to write:

- They are smaller than the capacity of the current block (the size of a block minus the offset).
- They exceed the capacity, in which case we have to limit it to the size of a block minus its offset.

The function will read the block to be written from the disk into a buffer with bread and then write into that buffer the contents of the buffer passed by the user (it will write the bytes limited depending on the situations above from the position stated by the offset). Then it will use bwrite to write back into the disk the updated buffer. Finally, it will update the values of f_seek and the size of the file, as well as the number of bytes left to write. It will also check again if we need to allocate any new blocks (when we are in the same situation described above and there are still bytes to be written).

- lseekFile(int fileDescriptor, long offset, int whence):

The objective of this functionality is to modify the position of the seek pointer of a file. Firstly, we check some preconditions for the function to run correctly:

- The file descriptor, that is the iNode ID, given as a parameter must be valid.
- The file whose pointer is meant to be modified must exist.

Once this is done, there are three possible situations, depending on the value of the parameter whence:

- The pointer is set to the end of the file: the pointer is placed using the iNode size.
- The pointer is set at the beginning of the file: the pointer is set at 0.
- The pointer is set to the current position: here the offset is added to the initial current position, provided that the pointer will not surpass the limits of the file.

2.2. Integrity Control

- checkFile(char *fileName):

  The objective of this functionality is to check the integrity of a file. Firstly, we check some preconditions for the function to run correctly:

  - The file to be checked must exist in the system.
  - This function can only be performed if the file is closed and it includes integrity.

  Now for the main part of the function a buffer with the size of the file is created. The functions lseekFile() and readFile() are used to read the whole contents of the file to then compute the CRC32 (cyclic redundancy check). This variable is later compared to the file iNode integrity field to check it is exactly the same. If they are not equal that means the file was corrupted.

- includeIntegrity(char *fileName):

  The objective of this function is to include the integrity of a file. As always, we first check for error in the parameters of the function and check all the preconditions are met:

  - The file must exist in the system.
  - The file must not already include integrity.
  - The file must be closed.

  Once this is done, the iNode field ‘includes_integrity’ must be changed to value 1. In addition, in the iNode field ‘integrity’ the function must store the value of the file data in a unit32_t variable through the CRC32 sumcheck calculated from the contents of the file (as we did in the checkFile() function).

- openFileIntegrity():

  The objective of this functionality is to open an existing file and check its integrity. Firstly, we check some preconditions for the function to run correctly:

  - The file must exist in the system.
  - The file must be closed.
  - The file must already include identity.

  Once all of that has been correctly checked, the function checkFile() is used to verify that the file is not corrupted and does not carry any other errors. If it was not corrupted, it sets the open_integrity field to 1.

- closeFileIntegrity():

  The objective of this functionality is to close a file and update its integrity. Firstly, we check some preconditions for the function to run correctly:

  - The file descriptor must be valid.
  - The file must have been opened with integrity.
  - The file must have not been opened without integrity.
  - The file must already include identity.

  Once all that has been correctly checked, the function computes the integrity of the file suing the CRC32 checksum as we did in the previous functions. Finally, the file is closed, changing the open_integrity field of the iNode to 0.

2.3. Symbolic Links

Once we added the functionality of the symbolic links to our file system, we added to all the previous functions used to modify the files (open, close, read, write, lseek, etc.) that if we try to use those functions with a symbolic link, those functions will automatically take the file the link points to.

- createLn():

  This function is used to create a symbolic link with the name "linkName" to the file with the name("fileName"). First it checks for errors in the arguments:

  - The name linkName is not used by any other iNode.
  - The name fileName corresponds to an existing iNode.
  - The name fileName does not belong to a symbolic link (we will only allow symbolic links to regular files to avoid cyclic links).

The functionality is similar to createFile(), as it will use ialloc() to allocate an iNode for the link, setting its type to SYM_LINK and name to linkName. The field "inode" will correspond to the iNode that corresponds to fileName, which the link points to. The rest of the fields of the iNode will be left empty as they are only used for regular files.

- removeLn():

  This method has a similar functionality to removeFile. It first checks for errors:

  - The argument linkName must correspond to an existing iNode
  - The iNode representing linkName must be of type SYM_LINK

  If the argument is valid it will use the function ifree to free the corresponding iNode and its metadata

3. Test Plan

In this table, only those tests that are include in the attached file 'test.c' are shown. To avoid redundancy, we have specifically tested some functionalities that are later being used inside many other functions. For example, we check that the system is mounted in a lot of functions, but its specific test is TP-05, which is the only one that actually explains it. Another example is the check of whether a file exists or not when calling a function that takes it as a parameter. The test TP-10 specifically checks it, even though many other functions and tests need to check it too as a precondition.

We also checked that the contents of the simulated file system in the file disk.dat were updated correctly with the command ***hexdump -C disk.dat > output.txt***

Test ID	Objective	Procedure	Return Value	Printed Output
TP-01	Check that the function mkFS() does not work with devices smaller than the minimum size	The device size used is 5*1024, which is considerably smaller than the minimum device size allowed	-1	Error mkFS: The size of the disk is smaller than the minimum size
TP-02	Check that the function mkFS() does not work with devices larger than the maximum size	The device size used is 1000*1024, which is considerably larger than the maximum device size allowed	-1	Error mkFS: The size of the disk is larger than the maximum size
TP-03	Check correct functionality of the function mkFS()	The device size used is a constant stated in the c file, which corresponds to the maximum size allowed, that is 600*1024	0	TEST mkFS SUCCESS

TP-04	Check correct functionality of the function mountFS()	Call for the function mount after the mkFS() function has been successfully run	0	TEST mountFS SUCCESS
TP-05	Check that a system cannot be mounted if it has already been mounted before	After having called mountFS() for the previous test, we call the function again	-1	Error mountFS: The file system is already mounted
TP-06	Check correct functionality of the function createFile()	Create a file with the name "./test.txt"	0	TEST createFile SUCCESS
TP-07	Check that you cannot create a file with a name that corresponds to another existing file	After the previous test, we call he function createFile() again with the same file name "./test.txt"	-1	Error createFile: File name already exists
TP-08	You cannot create more than 48 files, corresponding to the 48 iNodes	As we have already created one file, we perform a loop to create as many files as the maximum number of files iNodes (which is 48). When creating the last file of this for loop, an error should be caught.	-2	Error ialloc: There are no free iNodes
TP-09	Check correct functionality of the function openFile()	Called the function openFile() with the file name "./test.txt"	0 (iNode ID of the file)	TEST openFile SUCCESS
TP-10	Check that you cannot open a file that does not exists	Called the function openFile() with a file that has not been created, such as "/notafile.txt"	-1	Error openFile: File does not exist
TP-11	Check correct functionality of the function writeFile()	Create a buffer with the size of a block and fill it with the number "1" with memset(). Use the function writeFile() to write that buffer into the file that corresponds to iNode 0, which is "./test.txt"	2048 (the block size)	TEST writeFile SUCCESS
TP-12	Check correct functionality of the function lseekFile()	Since on the previous test we have written on the file, the pointer should be at the end of the file. Call lseekFile() function to move the f_seek pointer of the "./test.txt" file to the beginning of the file (FS_SEEK_BEGIN)	0	TEST lseekFile SUCCESS
TP-13	Check correct functionality of the function readFile()	We declare a buffer to store the contents read, which will have the size of a block. Call the function readFile() for the file "./test.txt".	2048, which is the number of bytes read.	TEST readFile SUCCESS
TP-14	Reading an empty file returns 0 bytes	Call the function readFile() for the iNode 1, which was created previously but it is empty.	0	

TP-15	Check correct functionality of the function lseekFile() when moving the f_seek pointer for a value given by an offset	Call lseekFile() to move the f_seek pointer of the file "./test.txt" with an offset of -1024 from the current position of the pointer (FS_SEEK_CUR), which should be at the end of the file after the read operation from TP-13. After that we perform another read operation.	lseekFile returns 0 readFile returns 1024	TEST lseekFile SUCCESS
TP-16	Check that if we try to write in a file a number of bytes that exceed the maximum number of bytes of a file (10240) then only the allowed size is written.	We will call the function writeFile() for the file with iNode 1 to write the maximum number of bytes plus one (10240+1). A buffer filled with "2s" with that size will be used.	10240.	TEST writeFile SUCCESS
TP-17	Check correct functionality of the function removeFIle()	We will use this function to remove the file "./test.txt"	0	TEST removeFile SUCCESS
TP-18	Check that we cannot call the functions writeFile(), readFile() and lseekFile() with a file that does not exist	We will call these functions with the iNode 0, which correspond to the file we just removed in the previous test	All the functions return -1.	Error readFile: file descriptor does not correspond to an existing iNode
TP-19	Check that we cannot open with integrity a file that does not include integrity	Call the function openFileIntegrity() with any of the files we have previously created.	-3.	Error openFileIntegrity: File does not include integrity
TP-20	Cannot close with integrity a file that was opened without it	We will open any of the files previously created with the function openFile(). Then we will try to close it with the function closeFileIntegrity().	-1	Error closeFIleIntegrity: File was opened without integrity
TP-21	Check correct functionality of the function closeFile()	We will close the file that we opened in the previous test.	0	TEST closeFIle SUCCESS
TP-22	Check correct functionality of the function includeIntegrity()	Call the function includeIntegrity() for the same file we used in the test TP-16, which we know is not empty.	0	TEST includeIntegrity SUCCESS
TP-23	Check correct functionality of the function openFileIntegrity() and closeFileIntegrity()	Call these two functions for the file we used in the previous test.	openFileIntegrity() returns fileDescriptor closeFileIntegrity() returns 0	TEST openFileIntegrity SUCCESS TEST closeFileIntegrity SUCCESS

TP-24	Check the correct functionality of the function openFileIntegrity() when the file is corrupted	We will use one of the files previously created and include integrity. Then we will open it without integrity with the function openFile() and write something in it. Then we close it with closeFile() and try to open it again with the openFileIntegrity() function.	-2	Error openFileIntegrity: File is corrupted
TP-25	Check correct functionality of the function createLn()	We call the function to create a link named "/link0" to one of the files we previously created	0	TEST createLn SUCCESS
TP-26	Check correct functionality of removeLn(), when we remove a file all the links to that file are removed.	We will remove the file to which the link we previously created was pointing to. Then we will call the function removeLn() with the name "/link0", which should have been removed already	removeFile() returns 0 removeLn() returns -1	Error removeLn: Link does not exist
TP-27	Check correct functionality of the function unmountFS()	Call the function unmountFS()	0	TEST unmountFS SUCCESS
TP-28	Check that you cannot unmount a system that is already unmounted	Call the function unmountFS() again	-1	Error unmountFs: The file system is already unmounted
TP-29	Check that if we try to write more blocks that the available data blocks, the function writeFile() stops writing after it has reached the maximum number allowed	As the maximum space all 48 files can take is 240 blocks and we have 4 blocks of metadata, we will create a new file system with 243 blocks. We will create all the files possible and fill them up to their maximum size.	writeFile() returns 8192 (=10240-2048) which is the number of bytes written for the last file we create (4 blocks)	TEST writeFile SUCCESS

4. Conclusions

This project was challenging to start, since we had to design the file system from scratch, and we lacked guidance on how to do it. However, once we figured out the structures that we had to use and got the first function mkFS() to work, the rest of the assignment was pretty straight forward.

Another difficulty of the project was how much attention we needed to pay to the little details, such as when we needed to allocate a new block for a file that is being written.

By the end of the project we gained a lot of knowledge on how file systems work, and we are happy with the work we have developed.