Universidade de Aveiro

Departamento de Electrónica, Telecomunicações e Informática

Sistemas de Operação/ Fundamentos de Sistemas Operativos

sofs21 (Academic year of 2021/2022) October, 2021

Previous note

The *sofs21* is a simple and limited file system, based on the ext2 file system, which was designed for purely educational purposes and is intended to be developed in the practical classes of the Operating Systems and Fundamentals of Operating System courses during academic year of 2021/2022. The physical support is a regular file from any other file system.

1 Introduction

Almost all programs, during their execution, produce, access and/or change information that is stored in external storage devices, which are generally called mass storage. Fit in this category magnetic disks, optical disks, SSD, among others.

Independently of the physical support, structurally one can verify that:

- mass storage devices are usually seen as an array of blocks, each block being 256 to 8 Kbytes long;
- blocks are sequentially numbered (LBA model), and access to a block, for reading or writting, is done given its identification number.

Direct access to the contents of the device should not be allowed to the application programmer. The complexity of the internal structure and the necessity to enforce quality criteria, related to efficiency, integrity and sharing of access, demands the existence of a uniform interaction model.

The concept of **file** appears, then, as the logic unit of mass memory storagement. This means reading and writing in a mass storage device is always done in the context of files.

From the application programmer point of view, a file is an abstract data type, composed of a set of attributes and operations. Is the operating system's responsability to provide a set of system calls that implement such abstract data type. These system calls should be a simple and safe interface with the mass storage device. The component of the operating system dedicated to this task is the **file system**. Different approaches conduct to different types of file systems, such as `NTFS`, `ext3`, `FAT*`, `UDF`, `APFS`, among others.

2 File as an abstract data type

The actual attributes of a file depend on the implementation. The following represents a set of the most common ones:

name — a user-convenient identifier to access the file upon creation;

internal identifier — a unique (numerical) internal identifier, more suitable to access the file from the file system point of view;

size — the size in bytes of the file's data;

ownership — an indication of who the file belongs to, suitable for access control;

permissions — a set of bit-attributes that in conjunction with the ownership allows to grant or deny access to the file;

monitoring times — typically, time of creation, time of last modification and time last access;

type — the type of files that can be hold by the file system; here, 3 types are considered:

regular file — what a user usually defines as a file;

directory — an internal file type, with a predefined format, that allows to view the file system as a hierarchical structure of folders and files;

shortcut (symbolic link) — an internal file type, with a predefined format, that points to the absolute or relative path of another file;

localization of the data — internal means to identify the blocks/clusters where the file's data is stored.

The operations that can be applied to a file depend on the operating system. However, there is a set of basic ones that are always present. These operations are available through system calls, i.e., functions that are entry points into the operating system. If follows a list, not complete, of system calls provided by Unix/Linux to manipulate the 3 considered file types:

- common to the 3 file types: `open`, `close`, `chmod`, `chown`, `utime`, `stat`, `rename`;
- common to regular files and shortcuts: `link`, `unlink`;
- only for regular files: `mknod`, `read`, `write`, `truncate`, `lseek`;
- only for directories: `mkdir`, `rmdir`, `getdents`;
- only for shortcuts: `symlink`, `readlink`;

You can get a description of any one of these system calls executing, in a terminal, the command

```
man 2 <syscall>
```

where `<syscall>` is one of the aforementioned system calls.

3 FUSE

In general terms, the introduction of a new file system in an operating system requires the accomplishement of two different tasks. One, is the integration of the software that implements the new file system into the operating system's kernel; the other, is its instantiation on one or more disk devices using the new file system format.

In monolitic kernels, the integration task involves the recompilation of the kernel, including the sofware that implements the new file system. In modular kernels, the new software should be compiled and linked separalely and attached to the kernel at run time. Any way, it is a demanding task, that requires a deep knowledge of the hosting system.

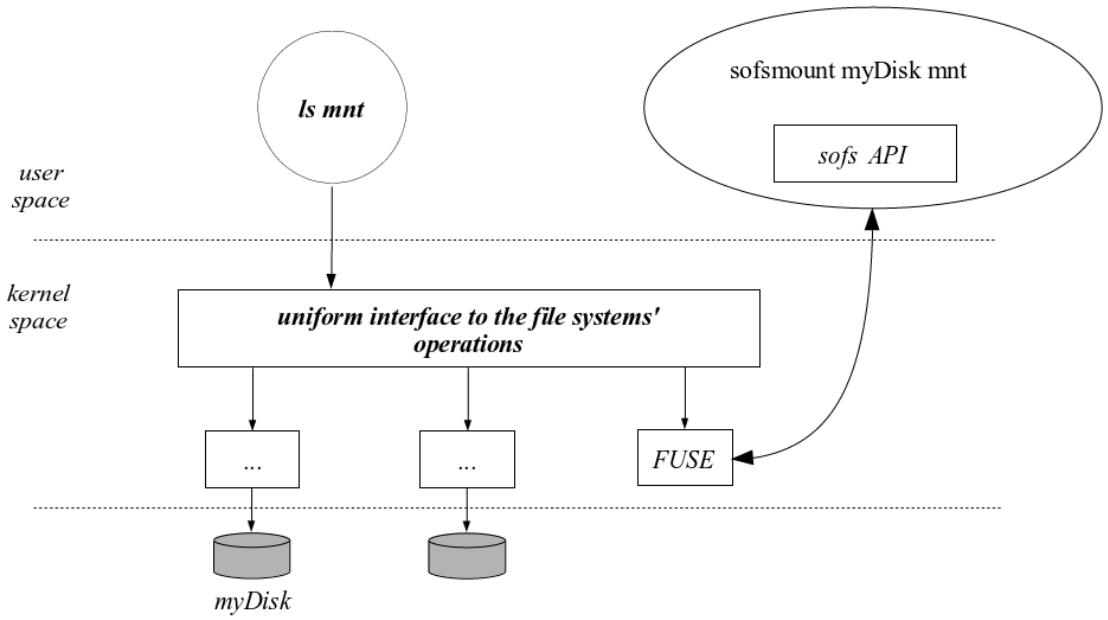
FUSE (File system in User SpacE) is a canny solution that allows for the implementation of file systems in user space (memory where normal user programs run). Thus, any effect of flaws of the suporting software are restricted to the user space, keeping the kernel imune to them.

The infrastructure provided by FUSE is composed of two parts:

- Interface with the file system — works as a mediator between the kernel system calls and the file system implementation in used space.
- Implementation library — provides the data structures and the prototypes of the functions that must be developed; also provides means to instantiate and integrate the new file system.

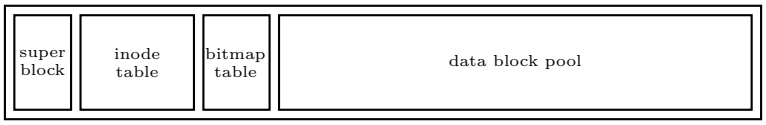
The following diagram illustrates how the *sofs21* file system is integrated into: the operating system using FUSE. Assuming a `sofs21` file system is mounted in directory `mnt`, the execution of the command `ls mnt` proceeds as follows:

1. The execution is decomposed in a sequence of calls to file system system calls in user space.
2. Each of these system calls enters the kernel space at the uniform interface.
3. Identified as `FUSE/sofs21`, it is redirect through the `FUSE` kernel module to the `sofs21` API, again in user space.
4. Since, in the case of `sofs21`, the disk is a file in another file system, new system calls are called to access that file.
5. The response is delivered to the `FUSE` kernel module, that, next, constructs the response to the original system call.



4 The *sofs21* architecture

As mentioned above a disk is seen as a set of numbered blocks. For *sofs21* it was decided that each block is 1024 bytes long. In a general view, the N blocks of a *sofs21* disk are divided into 4 areas, as shown in the following figure.



The **superblock** is a data structure stored in block number 0, containing global attributes for the disk as a whole and for the other main file system structures.

An **inode** is a data structure that contains all the attributes of a file, except the name. There is a contiguous region of the disk, called **inode table**, reserved for storing all inodes. This means that the number of inodes in a *sofs21* disk is fixed after formatting. This also means the identification of an inode can be given by a number representing its relative position in the inode table, thus an integer number in the range $[0, N-1]$, being N the total number of inodes.

The actual data of any file is stored in blocks taken from the **data block pool**, being a contiguous sequence of blocks. The number of blocks in a *sofs21* disk is fixed after formatting. The identification of a data block can be given by a number representing its relative position in the data block pool, thus an integer number in the range $[0, M-1]$, being M the total number of data blocks.

Managing the file system requires to know which inodes and data blocks are free and to keep this information stored in the disk. The list of free inodes is totally stored in the superblock. The list of free data blocks is partially stored in the superblock and partially in a dedicated sequence of blocks, the **reference bitmap table**.

4.1 Management of inodes

In *sofs21*, an inode can be in one of three possible states: **in-use**, **deleted**, and **free**. When a new file is to be created, a free inode must be assigned to it. It is therefore necessary to:

- define a policy to decide which free inode should be used when one is required;
- define and store in the disk a data structure suitable to implement such policy.

For each inode in the inode table there is a bit that represents its free state, either free (value 1) or in-use or deleted (value 0). These bits are stored in the superblock, in a field called **free inode bitmap**.

When a file is deleted, its inode transits from the in-use to the deleted state. The references of deleted inodes are kept in a queue in the superblock. This queue is managed as a circular FIFO. In some file operations, the deleted inode at the tail of this queue is cleaned and becomes free.

The operation of looking for a free inode (corresponding bit in the bitmap at one) is called *inode allocation*. There is not specific order by which inodes are allocated. However, in order to introduce a kind of rotativity in the allocation process, the bits in the bitmap are considered as forming a circular bit chain and the search for a 'free' bit starts in the position circularly next to the last one allocated. A field in the superblock holds this bit position, being updated by the inode allocation operation.

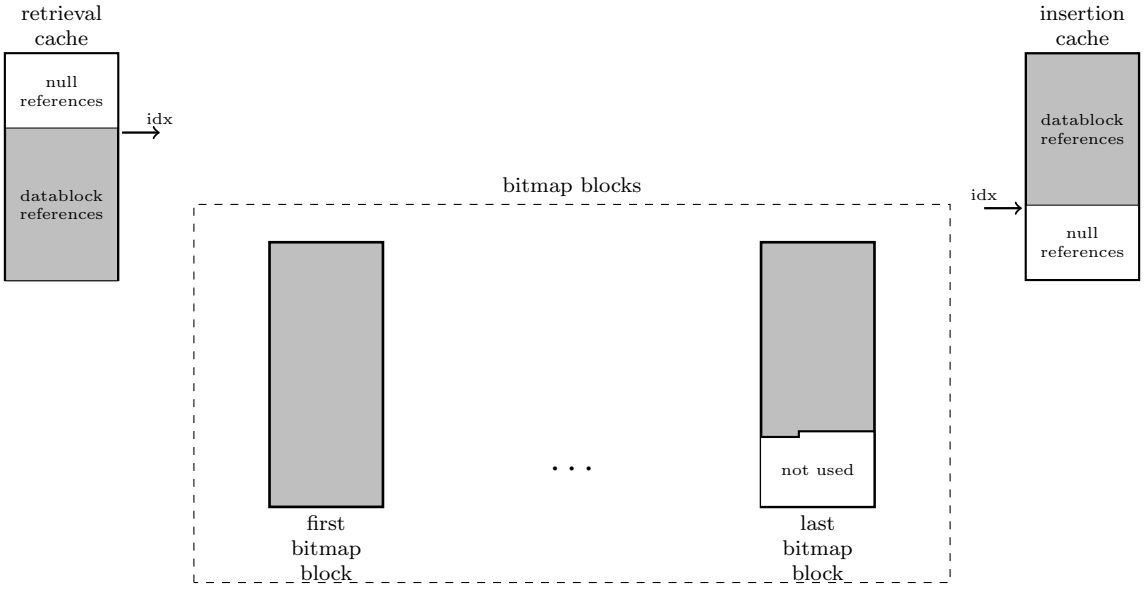
4.2 Management of data blocks

Similar to what was stated for inodes, at any given moment, some data blocks will be assigned to in-use or deleted inodes, while others will be available (free). Thus, again, it is necessary:

- to define a policy to decide which free data block should be used when one is required;
- to define and store in the disk a data structure suitable to implement such policy;

In *sofs21* it was decided to use one bitmap, stored in a set of contiguous blocks following the inode table, and two caches, named **retrieval cache** and **insertion cache**, stored in the superblock. In the bitmap there is a bit for every datablock, that acts as a boolean variable representing its state. The two caches are used to store directly references to datablocks.

A datablock is free if either its reference is in one of the caches, or its corresponding bit in the bitmap is at one. The two caches are used to improve the operations of allocation (assignment of a free datablock to a file) and releasement. This way, most of the times, the two operations only have to access the superblock. Next figure illustrates the supporting data structure.



In the allocate operation, a reference to a free datablock is got from the retrieval cache. If this cache is empty, references are transfered from the bitmap to the cache, before proceeding as before. References (bits at one) on the bitmap are transfered sequentially. A global 32-bit word index, stored in the superblock, indicates where in the bit map the transference should start from. This creates a kind of rotativity in the use of the datablocks. However, note that this does not actually represent a FIFO policy. (Can you figure out why?) If both retrieval cache and bitmap are empty, references are transfered from the insertion cache. If the release operation, the reference to the new free datablock is inserted in the insertion cache. If the cache is full, the references in the cache are transfered to the bitmap before proceeding as before.

4.3 Management blocks assigned to an inode

Blocks are not shared among files, thus, an in-use block belongs to a single file. The number of blocks required by a file to store its information is given by

$$N_b = \texttt{roundup}(\frac{\texttt{size}}{\texttt{BlockSize}})$$

where `size` and `BlockSize` represent, respectively, the size in bytes of the file and the size in bytes of a block. N_b can be very high. Assuming, for instance, that the block size in bytes is 1024, a 2 GByte file needs two million blocks to store its data. But, N_b can also be very small. In fact, for a 0 bytes file, N_b is equal to zero. Thus, it is impractical that all the blocks used by a file are contiguous in disk. The data structure used to represent the sequence of blocks used by a file must be flexible, both in size and location, growing as necessary.

The access to the file's data is in general not sequencial, but instead random. Consider for instance that, in a given moment, one needs to access byte index j of a given file. What is the block that stores such byte? Dividing j by the block size in bytes, one get the index of the block, in the file point of view, that contains the byte. But, what is the number of the block in the disk point of view? The data structure should allow for an efficient way of finding that number.

In *sofs21*, the defined data structure is dynamic and allows for a quick identification of any data block. Each inode allows to access a dynamic array, denoted d, that represents the sequence of blocks used to store the data of the associated file. Being `BlockSize` the size in bytes of a block, $d[0]$ represents the number of the block that contains the first `BlockSize` bytes, $d[1]$ the next `BlockSize` bytes, and so forth.

Array d is not stored in a single place. The first 6 elements are stored directly in the inode, in a field named `d`. The next elements, when they exist, are stored in an indirect and double indirect way. Inode field `i1` may contain the number of a datablock used to extend array d. Being `RPB` the number of references to blocks that can be stored in a block, `i1` represents the number of the data block that extends array d from $d[6]$ to $d[\texttt{RPB} + 6 - 1]$.

For bigger files, a double indirect approach is used. Inode field `i1` can be seen as the first element of an array i_1, each element being used to extend array d. In this understanding, inode field `i2` represents the number of a data block used to extend array i_1, from $i_1[1]$ to $i_1[\texttt{RPB}]$. Note that $i_1[i]$ contains the number of the data block that extends array d from $d[i * \texttt{RPB} + 6]$ to $d[(i + 1) * \texttt{RPB} + 6 - 1]$.

Pattern `NullBlockReference` is used to represent a non-existent block. For example: if `d[1]` is equal to `NullBlockReference`, the file does not contains block index 1; if `i1` is equal to `NullBlockReference`, it means $d[6]$ to $d[\texttt{RPB} + 6 - 1]$ are equal to `NullBlockReference`; if `i2` is equal to `NullBlockReference`, it means $i_1[1]$ to $i_1[\texttt{RPB}]$ are equal to `NullBlockReference`, and thus $d[\texttt{RPB} + 6]$ to $d[\texttt{RPB}^2 + \texttt{RPB} + 6 - 1]$ are equal to `NullBlockReference`.

A file can contain **holes**, meaning that a reference to a data block may be equal to `NullBlockReference` while another one afterwards does not. A block with reference `NullBlockReference` that is covered by the size of a file represents a stream of `BlockSize` zeros.

4.4 Directories

A **directory** is a special type of file that allows to implement the typical hierarchical access to files, the so-called *paths*. In *sofs21*, functionally, a directory is composed of a set of **directory entries**, each one associating a name to an inode. Structurally, a directory can be considered as an array of fixed-size **directory slots**. Each directory slot is composed of two fields, one to store a reference to an inode and another to store a string, named `nameBuffer`. A directory entry can occupy a single directory slot or be distributed among a sequence of contiguous slots, depending on the size of the name. The last character of `nameBuffer` indicates if the entry extends to the next slot or not. As names are defined to be null terminated, if the last character is not `'\0'`, the directory entry extends to the following slot. When two or more slots are associated to the same directory entry, they hold the same inode reference.

The slots of a directory can be in one of two states: in-use or clean (free). A slot is in-use if it is associated to a directory entry. When a directory entry is deleted, all of its slots are cleared, becoming free. A slot in the (clean) free state has the name field totally filled with the null character (`'\0'`) and the reference field equal to `NullInodeReference`. Only free slots can be used to store a new directory entry.

Upon creation, a directory has a complete data block assigned to it, with all the slots put in the free state, except the first two. The first two entries have special meaning and are presented in every directory. They are named, `"."` e `".."`', the former representing the directory itself and the latter representing the parent directory. At that moment, the size of the directory will be `BlockSize` (1024).

As a consequence of a file operation, a directory entry can be created or deleted. On creation, a sequence of contiguous free slots, with enough room to store the filename, must be used. This can cause the assignment of a new data block to the directory, in which case the data block must be properly formatted and the size of the directory increased in `BlockSize` bytes. Thus, the

size of a directory is always a multiple of the size of a block. On deletion, all slots are cleaned and become free. The directory in such cases is not shrinked, this meaning that after a free slots there can exist in-used slots.

5 Formatting

The formatting operation must fill the blocks of a disk in order to make it an empty, well-formatted *sofs21* device. It is the operation that have to be performed before the disk can be used.

In a newly formatted disk, there is one inode in-use, inode number 0, associated to the root directory, while all the others are free. The parent of the root directory is the root itself. The free inode bitmap in the superblock must be filled to reflect this state.

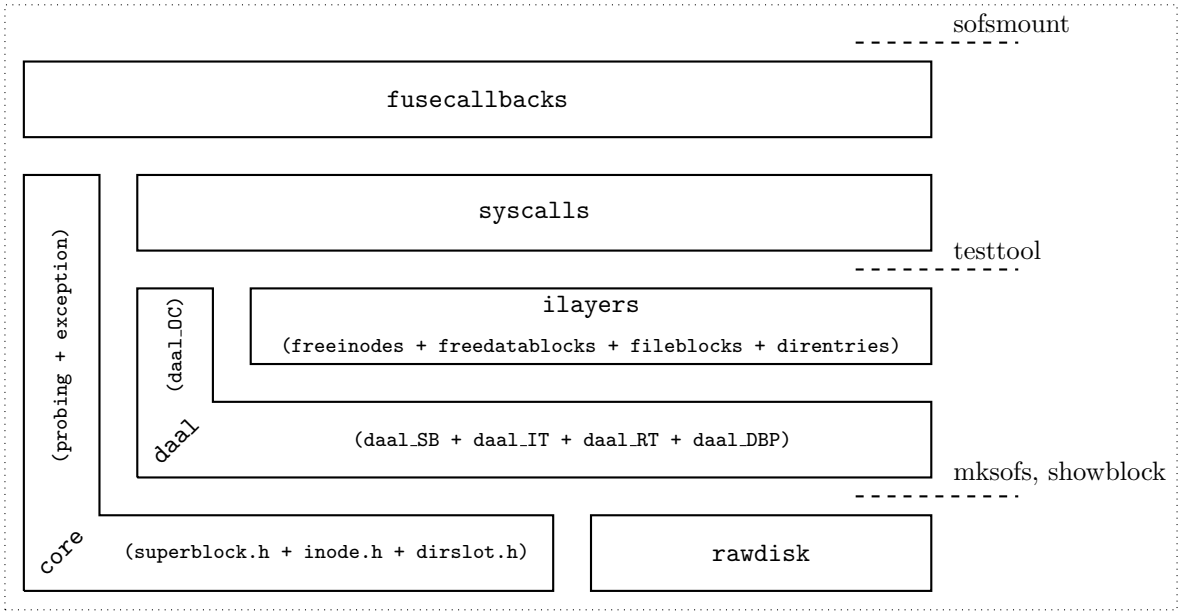
In a newly formatted disk, there is one data block in-use, data block number 0, while all the others are free. This data block is used by the root directory, to store the entries `"."` and `".."`.

The formating operation must:

- Choose the appropriated values for the number of inodes, the number of data blocks, the number of free data blocks, and the number of blocks for the reference bitmap table, taking into consideration the number of inodes requested by the user and the total number of blocks of the disk.
- Fill in all fields of the superblock, taking into consideration the state of a newly formatted disk. Both the insertion cache and the retrieval cache must be put empty.
- Fill in the table of inodes, knowing that inode number 0 is used by the root and that all other inodes are free.
- Fill in the bitmap table, knowing that data block 0 is in use and that the bits not used should be put as in-use.
- Fill in the root directory.
- Fill in with zeros all free data blocks, if such is required by the formatting command.

6 Code structure

Supporting code is structured in layers, as depicted in the following figure.



rawdisk – This layer implements the physical access to the disk.

core – This layer implements a debugging library (`probing`), the exception handling (`exception`) and defines the *sofs21* data types (`superblock`, `inode` and `direntry`).

daal – daal stands for disk access abstraction layer and implements the access to the different regions of a *sofs21* disk (superblock, inode table, data block pool and reference table); it also includes functions to open and close the disk.

ilayers – This layer implements a set of intermmediate functions that facilitataes the implementation of the system calls. It is composed of 3 modules: management of the free lists (freelists); access to the blocks of a file (fileblocks); and manipulation of directories (direntries).

syscalls – System calls are the entry points into the operating system. These are the *sofs21* version of the file system calls.

fusecallbacks – FUSE imposes a format to register file system calls to the operating system. This layer implements the interface with the syscalls layer.

mksofs – This is the (executable) formatting tool. It is the first program that have to be developed. Only upon formatting a disk it can be seen as a *sofs21* one.

showblock – This is a (executable) tool to visualize blocks. An option may be used to define how to interpret the blocks (superblock, sequence of inodes, sequence of directory entries, ...).

testtool – This is a (executable) tool to test the intermmediate layer functions during development.

sofsmount – This is the main tool to register the *sofs21* file system into the operating system (Linux).