Universidade de Aveiro

Departamento de Electrónica, Telecomunicações e Informática

Sistemas de Operação/ Fundamentos de Sistemas Operativos

sofs19 (Academic year of 2019/2020) September, 2019

Previous note

The *sofs19* 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 2019/2020. 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.

1.1 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;

localization of the data — means to identify the blocks where the file's data is stored;

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.

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.

1.2 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.

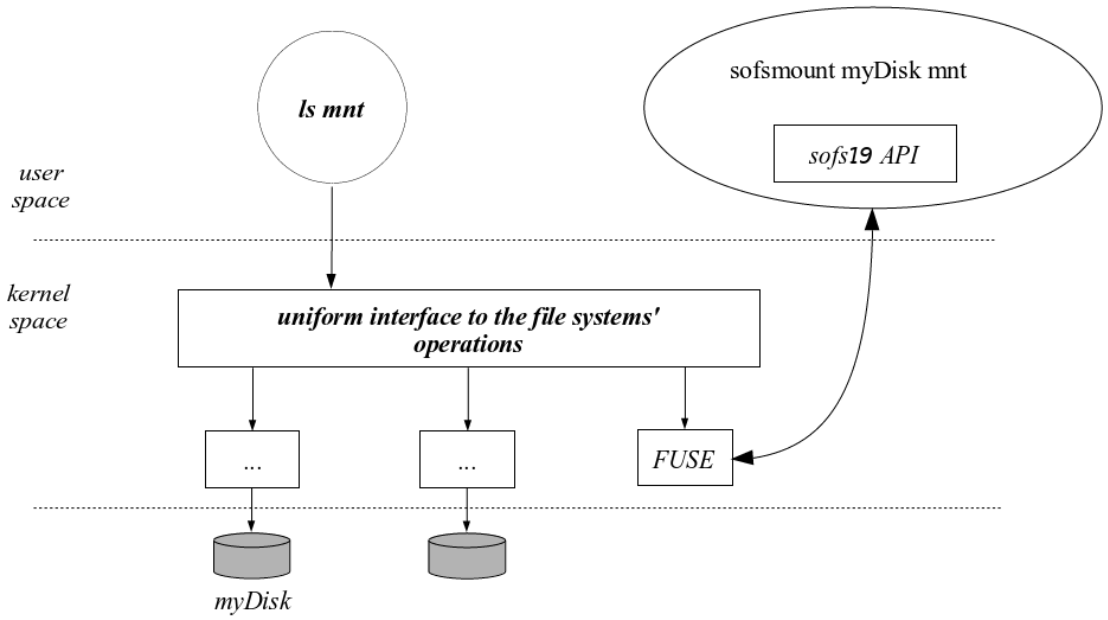
FUSE (File system in User SpacE) is a canny solution that allows for the implementation of file systems in user space (memory where normal user programs run). Thus, any effect of flaws of the suporting software are restricted to the user space, keeping the kernel imune to them.

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.

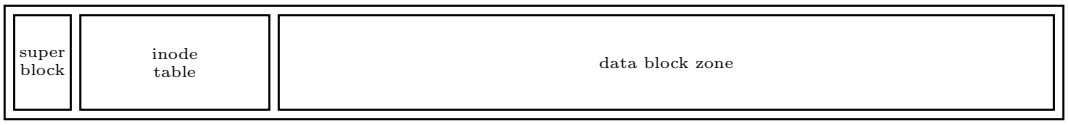
The following diagram illustrates how the *sofs19* file system is integrated into the operating system using FUSE. Assuming a `sofs19` file system is mounted in directory `mnt`, the execution of the command `ls mnt` proceeds as follows:

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/sofs19`, it is redirect through the `FUSE` kernel module to the `sofs19` API, again in user space.
4. Since, in the case of `sofs19`, 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.



2 The *sofs19* architecture

As mentioned above a disk is seen as a set of numbered blocks. For *sofs19* it was decided that each block is 1024 bytes long. In a general view, the N blocks of a *sofs19* disk are divided into 3 areas, as shown in the following figure.



Superblock

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

Inodes

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 the identification of an inode can be given by a number representing its relative position in the inode table. This also means that the number of inodes in a *sofs19* disk is fixed after formatting.

List of free inodes

At any given time, there are inodes in use, while others are available to be used (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.

In *sofs19* a FIFO policy is used, meaning that the first free inode to be used is the oldest one in the list. The implementation is based in a linked list of free inodes, built using the inodes themselves. A field in the inode data structure holds, when the inode is free, the number of the next free inode. Two fields in the superblock hold the numbers of the first and last free inodes.

Data blocks

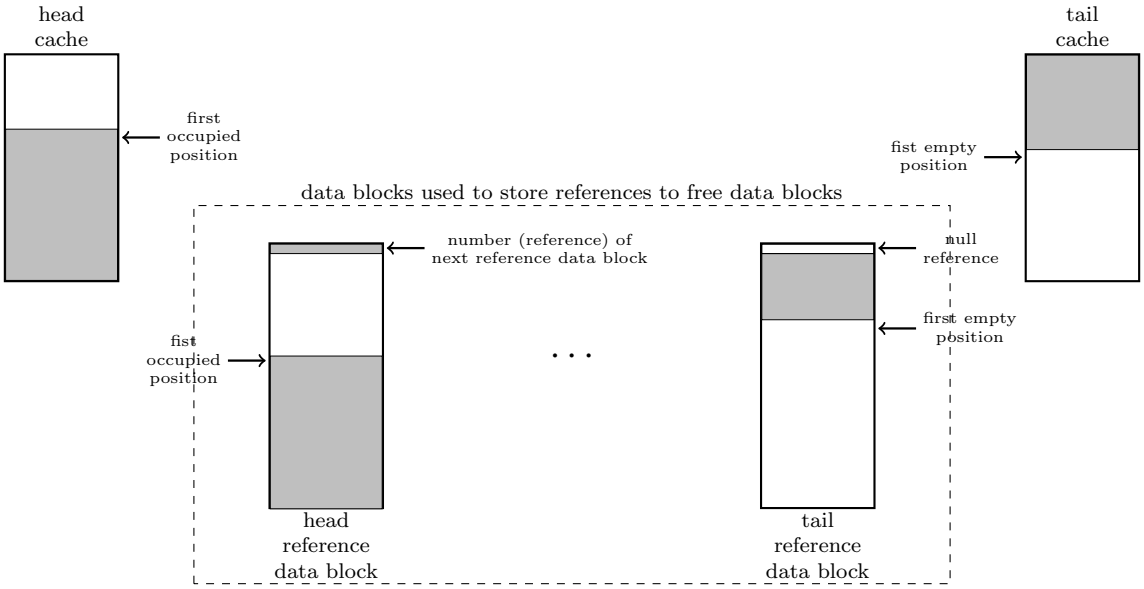
The actual data of any file is stored in blocks taken from the **data block zone**, which is a contiguous sequence of blocks at the end of the disk. This means the identification of a data block can be given by a number representing its relative position in the data block zone. This also means that the number of blocks in a *sofs19* disk is fixed after formatting.

List of free data blocks

Similar to what was stated for inodes, at any given moment, part of the data blocks will be in use (for example, by files stored in the disk), while the 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 *sofs19*, like for inodes, a FIFO policy is used, meaning that the first free data block to be used is the oldest one in the list. The implementation is based in a list of references to free data blocks, stored in the superblock and some data blocks, as illustrated by the next figure.



The list of free data blocks can be seen as a sequence of sub-sequences of references. The first sub-sequence is stored in the **head cache**. It represents the oldest references in the list. The last sub-sequence is stored in the **tail cache**. It represents the most recent references in the list. Intermediate sub-sequences are stored in data blocks. At a given moment, it can exist zero or more data blocks with references to free data blocks. The first of these data blocks, if exist, is called the **head reference data block**; The last of these data blocks, if exist, is called the **tail reference data block**. If only one data block contains references to free data blocks, it is both the head and the tail reference data block.

At a given moment, both the tail cache and the tail reference data block can be partially occupied. When a data block is freed and the tail cache is not full, it is stored in the first empty position of the tail cache. If the tail cache is full, a deplete operation must take place first, which means references are transferred to the tail reference data block, in order to create room. If there is enough room in the tail reference data block, all references in the tail cache are transferred. Otherwise, only those necessary to fill completely the tail references data block are transferred. In this case, in the next deplete operation, a new reference data block is added to the list, becoming the (new) tail reference data block.

At a given moment, both the head cache and the head reference data block can be partially empty. When a free data cluster is required and the head cache is not empty, a reference is retrieved from the first occupied position of the head cache. If the head cache is empty, a replenish operation must take place first, which means references from the head data block are transferred to the head cache. If there are enough references in the head reference data block, the head cache is fill completely. Otherwise, only those in the head reference data block are transferred. After the replenish operation, if the head reference data block get empty, the next reference data block, if any, must become the (new) head reference data block.

2.1 List of blocks used by a file (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 8 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, growing as necessary.

The access to a 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 holds 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 a efficient way of finding that number.

In *sofs19*, 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, $d[0]$ stores 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 8 elements are directly stored 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` represents the first 8 elements of an array i_1 ($i_1[0] \cdots i_1[7]$), each element being used to indirectly extend array d. Being `RPB` the number of references to blocks that can be stored in a block, $i_1[0]$ is the number of the data block that extends array d from $d[8]$ to $d[\texttt{RPB} + 7]$, $i_1[1]$ is the number of the data block that extends array d from $d[\texttt{RPB} + 8]$ to $d[2 * \texttt{RPB} + 7]$, and so forth.

For bigger files, the double indirect approach is used. Inode field `i2` represents an 8-elements array i_2 ($i_2[0] \cdots i_2[7]$), each element being used to indirectly extend array i_1. Thus, $i_2[0]$ is the number of the block that extends array i_1 from $i_1[8]$ to $i_1[\texttt{RPB} + 7]$, $i_2[1]$ is the number of the block that extends array i_1 from $i_1[\texttt{RPB} + 8]$ to $i_1[2 * \texttt{RPB} + 7]$, and so forth.

Pattern `NullReference` is used to represent a non-existent block. For example: if `d[1]` is equal to `NullReference`, the file does not contains block index 1; if `i1[0]` is equal to `NullReference`, it means $d[8]$ to $d[\texttt{RPB} + 7]$ are equal to `NullReference`; if `i2[0]` is equal to `NullReference`, it means $i_1[2]$ to $i_1[\texttt{RPB}+7]$ are equal to `NullReference`, and thus $d[8*\texttt{RPB}+8]$ to $d[\texttt{RPB}^2+8*\texttt{RPB}+7]$ are equal to `NullReference`.

2.2 Directories

As stated before, a directory is a set of directory entries. In *sofs19*, a directory entry is a data structure composed of a fixed-size array of bytes, used to store the name of a file, and a reference, used to indicate the inode associated to that file. Thus, a directory can be seen as an array of fixed-size slots, each one with capacity to store a directory entry. The data structure is defined such that a data block stores an integer number of directory entries.

The slots of a directory can be in one of three states: in-use, deleted or (clean) free. A slot is in-use if it contains a directory entry, the name and inode number of an existing file (regular file, directory or shortcut). A slot becomes deleted after a remove operation on a directory. In this state, the entry keeps the inode number and the first and last characters of the name are swapped. A slot in the (clean) free state has the name field totally filled with the null character (`'\0'`) and the reference field equal to `NullReference`. When a new directory is created, the first slot not in-use is used.

On creation, a directory has a 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. When all the slots of a directory are in-use and a new

entry is to be created, a new block is added to a directory, with the entry in the first slot and all the others put in the free state. Thus, the size of a directory is always a multiple of the size of a block. Because of the way deletion is done, a directory never shrinks.

3 Formatting

The formatting operation must fill the blocks of a disk in order to make it an empty *sofs19* device. It is the operation to be performed before the disk can be used.

In a newly formatted disk, there is one inode in-use, assigned to the root directory, while all the others are free. The root directory is assigned to inode number 0. The parent of the root directory is the root itself. The list of free inodes must start in inode number 1 and go sequentially to the last.

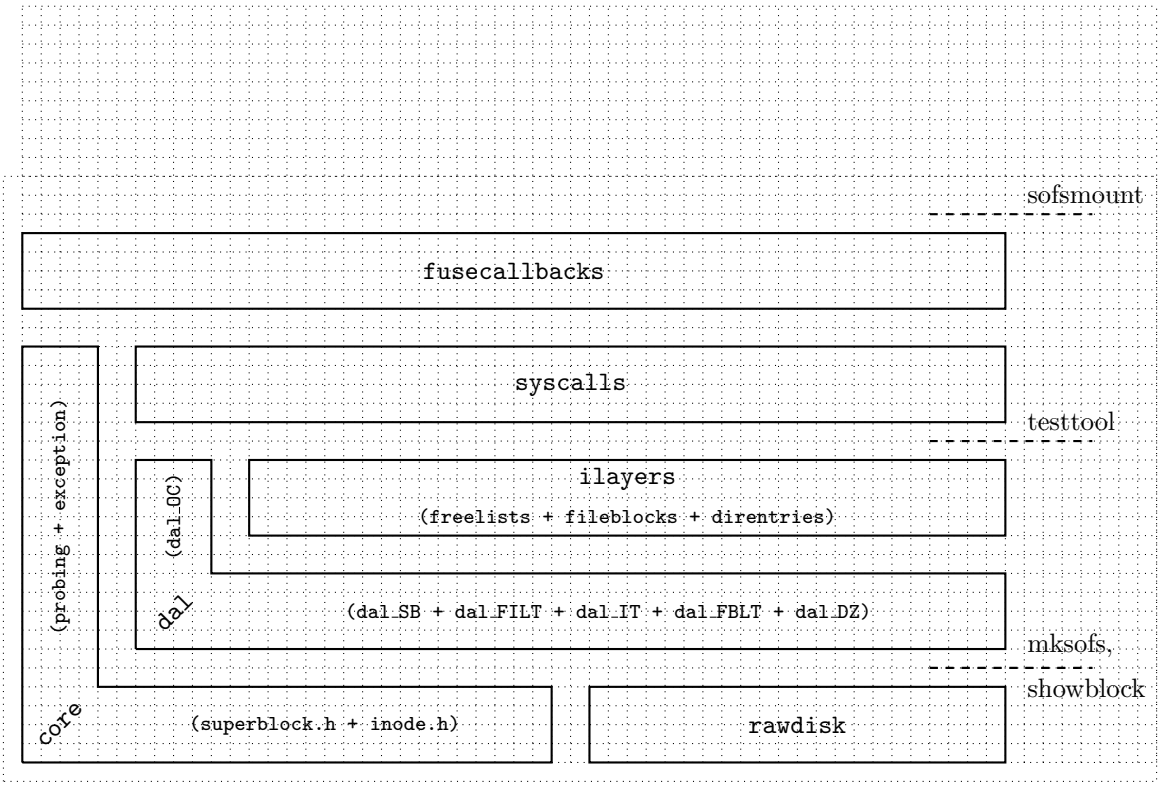
In a newly formatted disk, a number of the first blocks are in-use. Data block number 0 is used by the root directory, to store the entries `"."` and `".."`. If the list of free data blocks has more references than the size of the head cache, some data blocks must be used store the remaining references. Assuming storing the list requires n data blocks, the list of free data blocks must start in data block number $n+1$ and go sequentially to the last.

Thus 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 data blocks used by the list of free data blocks, 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. The tail cache should be empty and the head cache should be filled (totally if possible).
- Fill in the table of inodes, knowing inode number 0 is used by the root and that all other inodes are free.
- Fill in the data blocks used to support the list of free data blocks.
- Fill in the root directory.
- Fill in with zeros all free data blocks, if such is required by the formatting command.

4 Code structure

The following diagram represents the structure of the *sofs19* code.



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

core – This layer implemnets a debugging library (probing), the exception handling (exception) and defines the *sofs19* data types (superblock and inode).

dal – dal stands for disk abstraction layer and implements the access to the different regions of a *sofs19* disk (superblock, inode table, and data block zone); 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 – *sofs19* version of file system calls.

fusecallbacks – Interface with FUSE.