Universidade de Aveiro

Departamento de Electrónica, Telecomunicações e Informática

Sistemas de Operação / Operating Systems

sofs17 (Academic year of 2017/18) September, 2017

Previous note

The *sofs17* 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 course in academic year of 2017/2018. 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 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 the 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 — an identifier used 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 — an attribute that in conjunction with the ownership allows to grant or deny access to the file;

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

localization of the data — means to identify the clusters 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 define 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 the entry points into the operating system. If follows a list, not complete, of the system calls provided by 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.

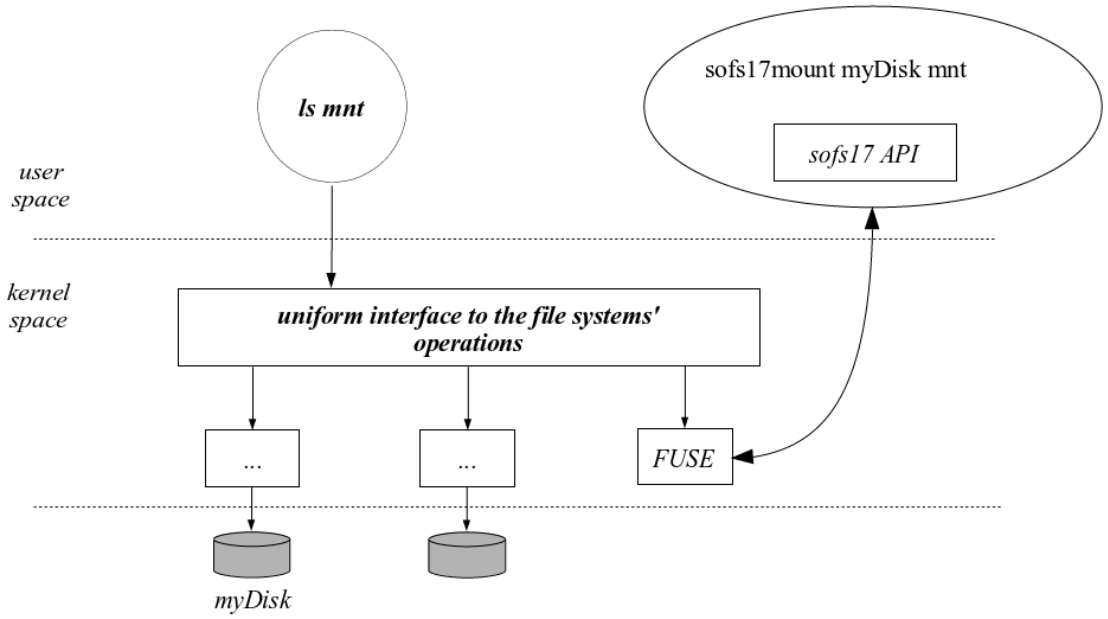
FUSE (File system in User SpacE) is a canny solution that allows for the implementation of file systems in user space (memory where normal user programs run). Thus, any effect of flaws of the suporting software are restricted to the user space, keeping the kernel imune to them.

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 their 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 *sofs17* file system is integrated into the operating system using FUSE.

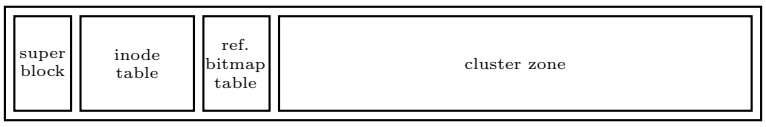


2 The *sofs17* architecture

As mentioned above a disk is seen as a set of numbered blocks. For *sofs17* it was decided that each block is 512 bytes long. Beyond the block size, there are more 5 key elements in defining the *sofs17* file system: superblock, inode, directory, cluster and reference map.

- 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 data structures.
- The **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 identification of an inode can be given by the index representing its relative position in the inode table.
- A **directory**, as mentioned above, is a special type of file that allows to implement the typical hierarchical access to files. A directory is composed of a set of **directory entries**, each one associating a name to an inode. It is assumed that the root directory is associated to inode number 0.
- The disk blocks used for storing file information are organized into groups of 4 contiguous blocks, called **clusters**. The identification of a cluster is given by the index that represents its relative position in the cluster zone.
- For each cluster in the disk there is a corresponding bit that represents its state. These bits are stored in a region of the disk called **reference bitmap table**.

In a general view, the N blocks of a *sofs17* disk are divided into 4 areas, as shown in the following figure.



2.1 List of free inodes

The number of inodes in an *sofs17* disk is fixed after formatting. At any given time, there are inodes in use, while others will be 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 *sofs17* a FIFO policy is used, meaning that the first free inode to be used is the oldest one. The implementation is based in a double linked list of free inodes, built using the inodes themselves. Two fields in the inode data structure hold, when the inode is free, the indexes of the next and previous free inodes. These lists are kept in a circular way, meaning that the previous of the first free inode is the last free inode, while the next of last free inode is the first one. Two fields in the superblock hold the numbers of free inodes and the number of first free inode.

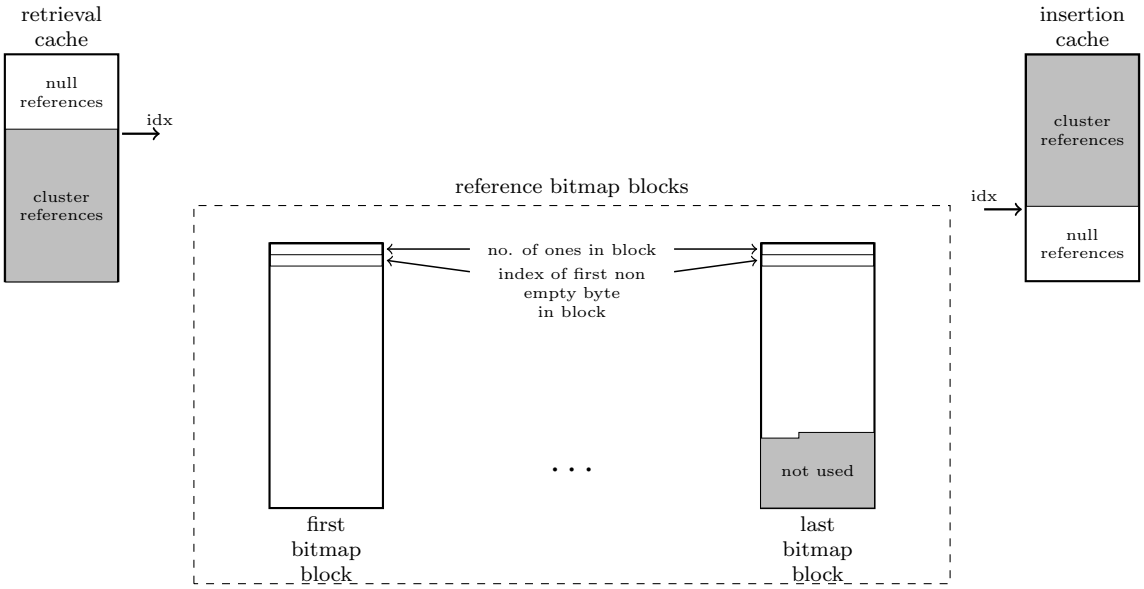
2.2 List of free clusters

The number of clusters in a disk is fixed by formatting. Similar to what was stated for inodes, in a given moment, part of the clusters will be in use (by the files stored in the disk), while the others will be available (free). Thus, it will be necessary:

- to define a way o representing the state (either free or in use) of every cluster in the disk;
- to define a policy to decide which free cluster should be used when one is required;
- to define and store in the disk a data structure suitable to represent the clusters' states and implement such policy;

In *sofs17* it was decided to use one bit map, 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 bit map there is a bit for every cluster, that acts as a boolean variable representing its state. The two caches are used to store directly references to clusters.

A cluster is free if either its reference is in one of the caches, or its corresponding bit in the bit map it at one. The two caches are used to improve the operations of allocation (assignment of a free cluster 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 cluster is got from the retrieval cache. If this cache is empty, references are transfered from the bit map to the cache, before proceeding as before. References (bits at one) on the bit map are transfered sequentially. A global byte position in the bit map, 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 clusters. However, note that this does not actually represent a FIFO policy. (Can you figure out why?) If both retrieval cache and bit map are empty, references are transfered from the insertion cache. If the release operation, the reference to the new free cluster is inserted in the insertion cache. If the cache is full, the references in the cache are transfered to the bit map, before proceeding as before.

2.3 List of clusters used by a file (inode)

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

$$N_c = \texttt{roundup}(\frac{\texttt{size}}{\texttt{ClusterSize}})$$

where `size` and `ClusterSize` represent, respectively, the size in bytes of the file and the size in bytes of a cluster.

N_c can be very high. Assuming, for instance, that the block size in bytes is 512 (the usual) and that the cluster size in blocks is 4, a 2 GByte file needs 1 million clusters to store its data. But, N_c can also be very small. In fact, for a 0 bytes file, N_c is equal to zero. Thus, it is impractical that all the clusters used by a file are contiguous in disk. The data structure used to represent the sequence of clusters 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 cluster that holds such byte? Dividing j by the cluster size in bytes, one get the index of the cluster, in the file point of view, that contains the byte. But, what is the number of the cluster, in the disk point of view? The data structure should allow for a efficient way of finding that number.

In *sofs17*, the defined data structure is dynamic and allows for a quick identification of any data cluster. Each inode allows to access a dynamic array, denoted d, that identified the sequence of clusters used to store the data of the associated file. Being `ClusterSize` the size in bytes of a cluster, $d[0]$ stores the number of the cluster that contains the first `ClusterSize` bytes, $d[1]$ the next `ClusterSize` bytes, and so forth.

Array d is not stores in a single place. The first 6 elements are directly stored in the inode, in the field named `d`. The next elements, when they exist, are stored in an indirect and double indirect way. Inode field `i1` represents the first element of an array i_1 ($i_1[0]$), used to indirectly extend array d. It is used to store the number of a cluster used to store the extension of array d from $d[6]$ to $d[\texttt{RPC} + 5]$, where `RPC` is the number of references to clusters that can be stored in a cluster. For bigger files, the double indirect approach is used. Inode field `i2` is used to stored the number of a cluster used to extend array i_1 from $i_1[1]$ to $i_1[\texttt{RPC}]$. For example, the first reference in this cluster, representing $i_1[1]$, is the number of the cluster containing the segment of array d from $d[\texttt{RPC} + 6]$ to $d[2 * \texttt{RPC} + 5]$.

Pattern `NullReference` is used to represent a non-existent reference. For example: if `d[1]` is equal to `NullReference`, the file does not contains cluster index 1; if `i1`, representing $i_1[0]$, is equal to `NullReference`, it means $d[6]$ to $d[\texttt{RPC} + 5]$ are equal to `NullReference`; if `i2` is equal to `NullReference`, it means $i_1[1]$ to $i_1[\texttt{RPC}]$ are equal to `NullReference`, and thus $d[\texttt{RPC} + 6]$ to $d[\texttt{RPC}^2 + \texttt{RPC} + 5]$ are equal to `NullReference`.

2.4 Directories

As stated before, a directory can be seen as an array of directory entries. In *sofs17*, 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. The data structure was defined such that a cluster holds an integer number of directory entries. 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.

Every directory entry can be in one of three states: in-use, deleted or clean. A directory entry is in-use if it contains the name and inode number of an existing file (regular file, directory or shortcut). When a directory entry is deleted, it keeps the slot in the directory. Only the name is changed, swapping the first and last bytes of the array. Thus, it is a null string, but with the original name recoverable. A directory entry in the clean state has the name field totally filled with the null character (`'\0'`) and the reference field equal to `NullReference`.

When a cluster is added to a directory, it is first totally formatted as a sequence of clean directory entries. Thus, the size of a directory is always a multiple of the size of a cluster. Also, and 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 *sofs17* device. It is the operation to be performed before the disk can be used.

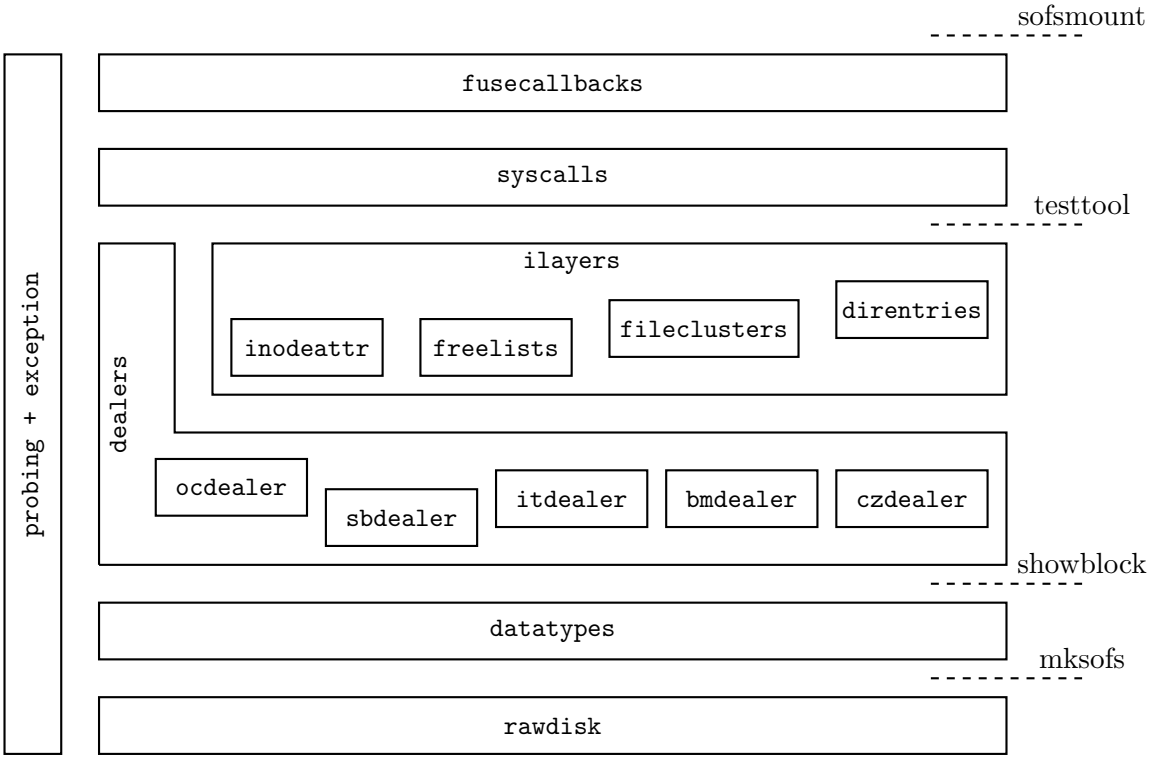
An empty disk (just formated) has root directory, which in turn has two entries filled, the `"."` e `".."` entries. By convention, the parent of the root is the root itself, thus both entries `"."` and `".."` points to inode number 0.

The formating operation must:

- Choose the appropriated values for the number of inodes, the number of clusters and the number of blocks used by the bit map, taking into consideration the number of inodes requested by the user and the total number of blocks of the disk. Note that all blocks of the disk must be used, so those that can not be used for other purposes should be added to the inode table.
- Fill in all fields of the superblock, taking into consideration the state of the disk after formating. The two caches should be put empty.

- Fill in the table of inodes, knowing inode number 0 is used by the root and that all other inodes are free. The list of free inodes must start in inode number 1 and go sequentially to the last.
- Fill in the bit map, knowing that cluster 0 is in use and that the bits not used should be put as in-use.
- Fill in the root directory, knowing that it should occupy cluster number 0.
- fill in with zeros all free clusters, if such is required by the formatting command.

4 Code structure



rawdisk – physical access to the disk

dealers – access to superblock, inodes, bit map and clusters

- **sbdealer** – access to the superblock
- **itdealer** – access to the inode table and to the inodes
- **bmdealer** – access to the reference bit map table
- **czdealer** – access to the cluster zone, using cluster references
- **ocdealer** – open/close the dealers

ilayers – intermmediate functions

- **inodeattr** – manage fields of the inodes
- **freelists** – manage list of free inodes and list of free clusters
- **filecluster** – handle clusters of an inode
- **direntries** – handle directory entries

syscalls – *sofs17* version of file system calls

fusecallbacks – interface with FUSE

probing – a debugging library

exception – the type of exception thrown in case of error