Pinguesco: A 32-bit File System

Diogo Sousa
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Caparica, Portugal
dm.sousa@fct.unl.pt

ABSTRACT
Permanent storage of structured data is on of the most basic abstractions expected from a general purpose operating system. File system stored on a disk solve this requirement. This paper discusses the design, implementation and evaluation of Pinguesco[1], a simple 32-bit file system build on top of *FUSE*.

Categories and Subject Descriptors
D.4.3 [**File Systems Management**]

Keywords
file system, file allocation table, indirect block, fuse

1. INTRODUCTION
This paper presents Pinguesco, a 32-bit disk file system. This file system was implemented in *C* with *FUSE*, thus running in *userland*. The file system partially implements the *POSIX standard*.

PinguescoFS support every common operation except permission handling, file timestamps and soft- and hard-links, and the data is permanently stored on a block device.

A file allocation table allows fast block allocation and multiple indirection blocks are used to support large files. Directories are supported and can be nested to an arbitrary depth.

2. PINGUESCO FILE SYSTEM DESIGN
The PinguescoFS has a simple structure. The *superblock* is the first block (block 0) and contains a magic number identifying the device as a Pinguesco file system, the number of blocks used and the number of inodes. The superblock also contains a pointer to the first free block.

[1]Latin for *fat*.

The next $2^{14} = 16384$ blocks contain the file allocation table [1]. This table keep a list of free blocks, with the first free block specified in the superblock. The last free block points to 0. This FAT size allows the file system to handle $4GB$ of data. The FAT allows constant time block allocation with no hard drive accesses, since it is loaded to main memory when the file system is mounted.

Each file in the file system is represented by an inode. An inode can represent a directory or a regular file. The inode also keeps the number of blocks allocated to represent that file, being that data blocks, indirect blocks or, in case of directories dentry tables (explained bellow). If the inode represent a regular file then the number of bytes the file has is also present in the inode, as well as the pointers to the indirect blocks. There are three pointers to indirect blocks, with single indirection, double indirection and triple indirection to allow for large files. This is done in the same way as the UNIX V7 file system [1]. If the inode represent a directory a pointer to the first dentry table block is also kept in the inode.

A dentry table represent the content of a directory. This table stores the filenames and correspondent inode of the directory. Since the number of files in the directory may be arbitrarily large, at the end of each dentry table exists a pointer to the next dentry table.

The indirect blocks are simply a sequence of pointers. The final indirection level point to the block containing the correspondent data. If this points to 0 then that block represents a block in which every byte is 0x00. This makes it easy for the expansion of a file with a `truncate` operation.

The first directory inode is always in the block 16385, the first block after the file allocation table.

Figure 1 exemplifies a Pinguesco File System with a file called “foo” containing the line “Hello World”.

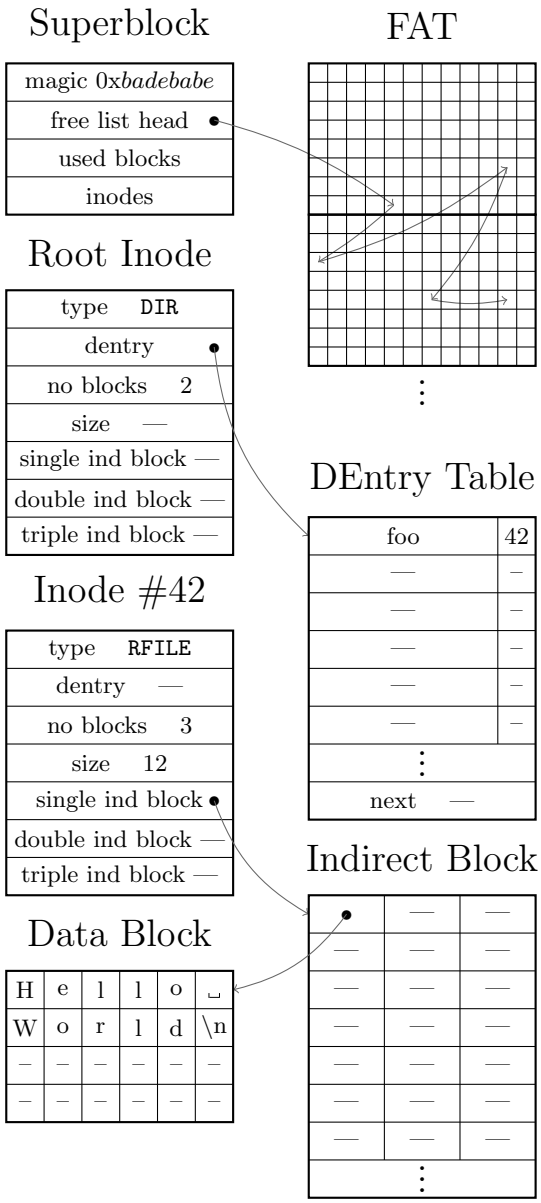

Figure 1: Pinguesco Example of PinguescoFS Structure.

3. IMPLEMENTATION

The implementation of PinguescoFS tries to be as minimalist as possible. There were no special concerns with seek times and the number of disk accesses, except in critical operations such as `read()` and `write()`. There are lots of room left for optimizations, such as caching.

The Pinguesco file system has a block size of 1024 bytes. This value was used to balance a tradeoff between internal fragmentation, number of access to read/write data and the size of the file allocation table (which end up being $\sim$ 16MB). This block size can easily be changed but this was the only size which was tested.

To simplify the manipulation of dentry table a directory always have at least one dentry table. When a file is created in a directory the first empty slot in the list of dentry table is used to store that file. When a file is removed the correspondent dentry table may get empty. In such cases that dentry table is freed (unless it is the first one) and the previous table will point to the next. A slot is empty when the filename has length zero. The filename is stored in a nul terminated string, making the maximum file name 63 bytes long. Each dentry table has 15 entries.

Every regular file always have a single indirect block. Each indirection has 256 pointers. This means that with only one indirection a file have a maximum of $256 \cdot 1024$ bytes = 256KB. The usage of double indirect block allows this limit to be $256 \cdot 1024 + 256^2 \cdot 1024$ bytes $=\sim$ 64MB and a triple indirect block expand the limit far beyond the maximum file system size: $256 \cdot 1024 + 256^2 \cdot 1024 + 256^3 \cdot 1024$ bytes $=\sim$ 16GB.

To create a valid PinguescoFS in a device a formatting tool was developed. This tool writes the initial state expected in a empty file system. Since no attention was made to normalize the endianness of integer values the portability of the PinguescoFS only hold for system using the same endianness. (An attempt to do otherwise will result in the magic number mismatch and the file system will refuse to mount, causing to harm to the file system.)

The operations implemented were `mkdir()`, `opendir()`, `readdir()`, `closedir()`, `rmdir()/unlink()`, `rename()`, `open()/creat()`, `read()`, `write()`, `close()`, `truncate()`, `access()`, `getattr()` and `statfs()`.

Since not all operation can be done in open file a red-black tree maintains a counter for each opened file. The only operation that cannot be done on an open regular file is `unlink()`, which will return `EBUSY` if done on a file which is opened. Other operations such as `rename()` can be done on open files since the inode remains unchanged. There is no problem in removing directories since the file system will be able to detect that and return a proper error to a process reading the directory (because the `readdir()` is atomic).

All operations are atomic using reentrant mutexes to enforce mutual exclusion.

4. EVALUATION

To evaluate PinguescoFS a series of stress test were performed.

To test the overall file system usage under lots of files of relatively small size the `linux-3.0.8.tar.bz2` was copied into the file system and extracted. The tarball itself have 74 MB, thus forcing the triple indirect block to be used. The *md5sum* of the tarball matched the expected result. The tarball was then extracted creating 39055 files. The directory structure was preserved and the *md5sum* of every file was ok[2]. The exact same procedure was done with `gcc-4.6.2.tar.bz2` with successful result except for 60 java compiled files with filenames larger than supported.

[2]There was only one missing file which was a soft-link, which are not supported in PinguescoFS

To test large files a few TV series were copied to the disk. The *md5sum* was also correct. The whole file system were filled with 4GB of TV series, my personal git repository ($\sim$ 630MB) and some photos. As expected the trying to copy more files gave a "no space left on device" error. After the removal of all these files the file system reported a usage of $\sim 1\%$, the minimum taking into account the file allocation table size. To test even larger files the Star Wars Episode I — The Phantom Menace[3], with $\sim$ 1.5GB was copied to the file system and the *md5sum* gave the correct result. MPlayer was also able to play it correctly and with no noticeable delay.

The `truncate()` operations was also tested by expanding and shrinking files, with the result compared to the same operations done on a *ext4* file system giving the same result.

5. CONCLUSION

Building a file system from scratch was very interesting and allowed me to better understand some of the fundamental data structures used in file system design. Since the Pinguesco was designed to have acceptable "real world" restrictions it was also very rewarding to see some normal common tasks being done in the file system.

6. REFERENCES

[1] A. S. Tanenbaum. *Modern Operating Systems*, chapter 6, pages 379–452. Prentice Hall, second edition.

[3] I apologize for not using the Episode IV.