Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

Unterschrift des Betreuers

M A S T E R A R B E I T

Java-based Filesystem for a Commercial Database System

ausgeführt am Institut für

Informationssysteme
der Technischen Universität Wien

unter der Anleitung von
Dipl.-Ing. Dr.techn. Roman Kurmanowytsch
und
Doz. Dipl.Ing. Dr.techn. Engin Kirda
als verantwortlich mitwirkendem Universitätsassistenten

durch

Marian Daubner

Kammelweg 1/4/15, 1210 Wien

Datum

Unterschrift (Student)

Abstract

Database systems have in recent years emerged as a crucial part in the storage process of immense data. Data is stored in diverse sophisticated relation models to maximize the performance and efficiency of data mining. On the other side, these features require explicitly built applications to manage the data, causing a lack of interoperability of the system. We consider and solve this problem with a user-space filesystem based on Java and describe the design and implementation of this Java-based filesystem for a particular database system – Braintribe Content Service Platform (BTCSP).

Zusammenfassung

Datenbanksysteme bekamen in den letzten Jahren ein entscheidendes Teil der Datenverwaltung. Daten sind bewahrt in raffinierten organisierten Strukturen mit komplexen Beziehungen um die Performanz und Effektivität von Datamining zu steigern. Anderseits erfordert diese Einrichtung explizit gebaute Applikationen um die Daten zu verwalten, und damit verursacht ein Mangel an Interoperabilität mit anderen Systemen bzw. Applikationen. Wir betrachten und lösen dieses Problem mit einem Java-basierendem user-space Filesystem. Diese Arbeit schildert den Entwurf und die Implementierung eines Filesystem entwickelt für ein bestimmtes Datenbanksystem – Braintribe Content Service Platform (BTCSP).

Content

ABSTRACT **2**

ZUSAMMENFASSUNG **2**

CONTENT **2**

1. PROBLEM DESCRIPTION **4**

1.1 INTRODUCTION 4
1.2 PROBLEM DEFINITION 5
1.3 ORGANIZATION OF THIS THESIS 6

2. REVIEW OF THE STATE OF THE ART **7**

3. DESCRIPTION OF THE APPROACH **9**

4. FILESYSTEMS **11**

4.1 MOST COMMON FILESYSTEM OBJECTS 13
4.1.1 Inode object 13
4.1.2 Dentry object 13
4.1.3 File object 13
4.2 BASIC FILE PROPERTIES 14
4.3 RELATION TO DATABASE SYSTEMS 16
4.4 VIRTUAL FILESYSTEM (VFS) 17
4.4.1 Registering and Mounting a Filesystem 18
4.4.2 Insight into a simple procedure 21
4.4.3 VFS Interface 22

5. FUSE **23**

5.1 USER-SPACE FILESYSTEM 23
5.1.1 Kernel-User-Space Communication 24
5.2 DESIGN 26
5.2.1 FUSE kernel module 26
5.2.2 fusermount 30
5.2.3 FUSE library 31
5.2.4 Call-Flow 32
5.3 BASIC FUSE STRUCTURES 35
5.3.1 FUSE kernel structures 35
5.3.2 FUSE user-space objects 38
5.3.3 FUSE exchange structures 39
5.4 FUSE EXPORTED INTERFACE 41
5.5 USAGE 42

6. BRAINTRIBE CSP **44**

7. JAVAFS **46**

7.1 FUSE MODIFICATIONS 47
7.2 JAVAFS CLIENT IMPLEMENTATION 48
7.3 COMMUNICATION PROTOCOL 51
7.4 JAVAFS SERVER 52

7.5 JAVAFS API 54
7.5.1 IFSAccessDir *54*
7.5.2 IFSFileAccess *56*
7.5.3 IFSAccessFileDir *57*
7.5.4 IFSAccessInode *58*
7.5.5 IFSAccessFS *60*
7.6 USING JAVAFS FRAMEWORK 60
8. BTCSP INTERNALS 62
8.1 API 64
8.2 BTCSP FS 70
8.2.1 Usage *74*
9. RELATED WORK 78
10. EVALUATION AND FURTHER WORK 80
10.1 EVALUATION 80
10.2 FURTHER WORK 82
11. SUMMARY AND CONCLUSION 83
REFERENCES 84
APPENDIX 86
A. VFS INTERFACE 86
B. FUSE OPERATION CODES AND PARAMETERS 96
C. LIST OF FUSE MODIFICATIONS RELATED TO 256BIT INODE SUPPORT 98

1. Problem Description

In the course of this thesis a filesystem was designed and developed for a database system called Braintribe Content Service Platform. This database system is used to store clients and their data (e.g. contracts, photos and other documents). The stored data is composed of fixed and variable part, which will be described later. To gain the idea behind this work filesystems are first briefly introduced.

1.1 Introduction

Since the development of the very first computer, filesystems are an inevitable part of all computer operating systems. About twenty years ago, only a handful number of filesystems existed (one of the most famous being FAT16 and FAT32). However, because of their widespread usage, some of them are still supported.

A filesystem may be divided into two main operational blocks. One must conform to a standard set of operations, which are identical in all filesystems. It is mostly called virtual filesystem (VFS) interface. It is developed in a way, that no matter of the filesystems storage medium and implementation, it gives all filesystems the standard look and usage. This part is accessible from applications over the system calls.

The latter part is specific for a particular filesystem. The methods here are private, usually called from the implementation of the VFS interface. This part contains calls to the underlying device, implements cache and organizes the data on the specific medium.

Nowadays, there is countless number of filesystems, e.g. device specific filesystems, RAM filesystems, network filesystems and even filesystems based on another filesystems. Operating systems usually support only the most common ones.

1.2 Problem Definition

The subject of the thesis originates from an actual software business problem. An Austrian company **Braintribe IT-Technologies GmbH**[1] was looking for an alternative way to present data managed by its application Braintribe Content Service Platform (BTCSP). The data managed by BTCSP stores information about clients and their documents with attached properties and notes. The data has a hierarchical structure similar to filesystems but with built-in properties and versioning.

The company wants to make the data accessible from multiple platforms independently of the operating system, preferably in an explorer-like environment. In cooperation, we considered three solutions as follows:

1. to develop a windows shell application (something like windows ZIP file explorer);
2. to use an open source web-tool called WebDAV that would make the data available in a browser; or
3. to build a filesystem.

All the pros and cons were accounted and we came to the conclusion that a filesystem would be the most flexible and robust solution. The pros and cons considerations may be found in chapter 3.

The filesystem feasibility analysis and development includes an open source framework called FUSE that was chosen to be used in the filesystem creation. FUSE became a part of the thesis focus, as the FUSE framework was necessary to modify to server purposes of this thesis. FUSE does not have yet any documentation therefore part of the work was dedicated to analyzing FUSEs implementation and design. The resulting knowledge is summed in chapter 5.

The next section gives a better insight into the chapters of this thesis.

[1] Braintribe IT-Technologies GmbH is a software development company based in Vienna. More about the Braintribe product can be found in chapter 6.

1.3 Organization of This Thesis

In this section the reader becomes an idea of the structure of this thesis. Single chapters are briefly described what they are about.

Data presentation is a common problem in database systems. Therefore next two chapters are dedicated to some other techniques used to solve this problem.

Chapter 3 describes the decision making process that steered us to the final solution presented in the thesis – the use of filesystems.

Chapter 4 gives an insight into the world of filesystems. It gives answers to the questions what a filesystem is, how it works and how it is used. It also illustrates the complex processes in the operating system and the design of the virtual filesystem.

In chapter 5, the practical part of the assignment begins with a description of the FUSE framework. Chapter 6 introduces the Braintribe product – Braintribe CSP – for which the filesystem was developed. In chapter 7 is explained JavaFS framework, which was included into the thesis to enhance the BTCSP maintenance and to create another abstraction layer for development of another filesystem in Java. Chapter 8 describes the internals of the BTCSP that this filesystem was made for, its API used to implement the BTCSP filesystem and the actual developed BTCSP filesystem.

Related work is mentioned in chapter 9 and the BTCSP filesystem evaluation is in chapter 10. Finally in chapter 11 can be found the summary and conclusion of the thesis.

2. Review of the state of the art

Graphical interface and presentation of data is a very important issue in database systems. This is the part of the software that the user comes in contact with. This may give a software product some advantage in the highly competitive market. Therefore software producers must be innovative and try to create software that would have all features that could be of some use to the end-user (client) that could give it the crucial advantage in the market. The most common presentation techniques are depicted in Figure 1.

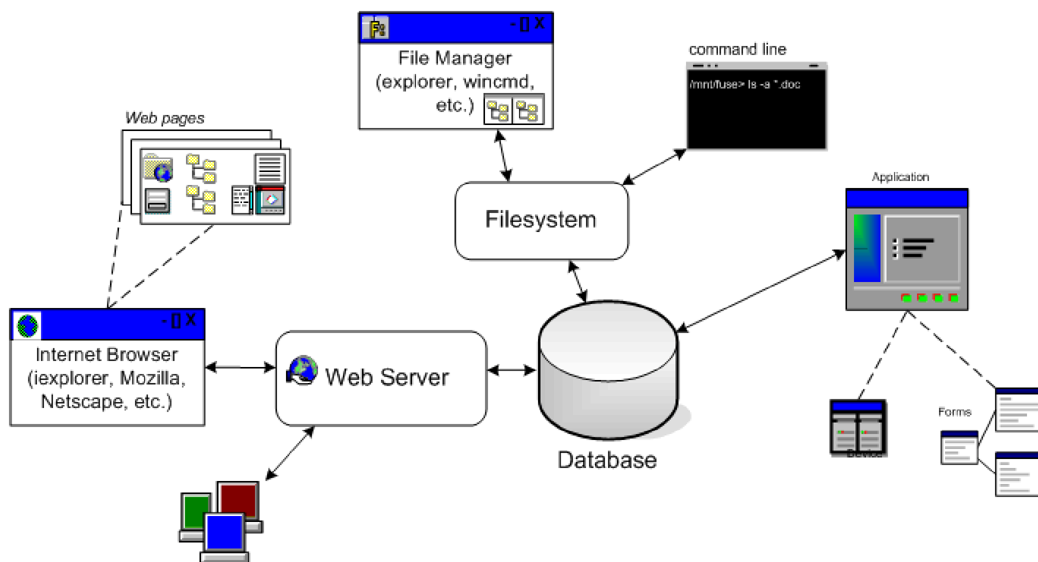


Figure 1: Common presentation models

Database systems are used to store huge piles of information and therefore without any arrangement of the data, data mining would be impossible. Therefore data must be often organized into sophisticated structures. The database architect must find the best solution for the design of the database to fit the requirements based on its future use. They face many different trade-offs, the most common being the trade-off between performance and storage space. The result is then a complex customized database architecture, which is often composed of many recurrent relations with various types of data stored. Such a database can be hardly represented in form of a filesystem.

Therefore the most common way to present the data in the database is also a customized application. In a multi-user environment web applications are more popular because of their effortless support and configuration.

In some cases the database is composed of hierarchical parts that have a more filesystem-like design that makes it possible to partially display the data in a filesystem. Filesystem then allows some simple but frequently used operations on these data.

It is now apparent that there is not only one approach to present data. It decidedly depends on its future use and on its hierarchical and structural characteristics. Multi-user database systems have mostly a web application access on contrary to single-user systems that will do with a normal application. Database systems composed of hierarchical architecture and storing files may enable also a filesystem access.

3. Description of the approach

We decided to make a filesystem to access the database system data in the BTCSP. We considered it as the most flexible and robust choice from the other possibilities we had. To implement the filesystem kernel module we used open-source user-space filesystem FUSE, which enormously simplified the development process. The implementation of the actual filesystem logic was written in Java.

BTCSP provides a database system that has a hierarchical structure composed of files. There were three options to access these data:

- to use a windows shell program similar to the ZIP explorer,
- to use an open-source Java web server (we considered WebDAV) that would allow an internet browser access, or
- to make a filesystem.

The first option allows windows explorer access. It would look good and it would make a good filesystem illusion, but it would be almost as complex as a filesystem and restrained for windows use only. Java web server choice would be a more platform independent solution, giving an advantage of developing in a object-oriented programming language, but loosing of some filesystem usage features. Making a BTCSP filesystem is the most difficult option, but it has all the pros of the other options, except the development in Java, and it has the advantage of being the most robust solution. For example scripts are easily written for a filesystem, a partial backup is easily possible and another filesystem may be used on it; I have in mind SAMBA filesystem, which allows network access.

Based on its characteristics, the filesystem choice was straightforward. Now we need to handle the complexity of this option. Filesystems are drivers or kernel-space programs implemented in the programming language of the operational systems, which is usually C. Kernel-space is a very restrictive programming environment, allowing only a set of pre-implemented functions. Besides it requires great knowledge of the operational system kernel that it is implemented for, which led us to the

conclusion to develop BTCSP filesystem for Linux, which is open-source and has lots of information free available on internet. Linux kernel-space programs are implemented in form of kernel modules; therefore from now on I will refer to them as kernel module.

But developing in kernel module has as well other difficulties. Debugging is almost impossible and bugs in the implementation lead mostly to a system crash; prepare to spend some time in front of rebooting machine. Therefore it is necessary to move most of the implementation into user-space.

But such problems arising with kernel module programming led us to search a way to avoid or facilitate it. We looked for a filesystem kernel module that would allow further development in user-space. And we found some user-space filesystems. We considered FUSE and LUFS[2]. We chose FUSE as it appeared to be the most documented, bug-free and most used one. A comparison of both filesystem APIs may be found in chapter FUSE. This chapter also explains some techniques used to exchange communication data between kernel and user space.

FUSE was ideal to build on. FUSE supports all necessary filesystem operations. These operations are then delegated to user-space. As will be seen later, FUSE does not fully fit to our requirements; therefore it was necessary to make some modifications to it. The section 7.1 is dedicated to show the problems with FUSE and the modifications of it.

Although implementing in user-space was relative simple, it was yet necessary to pass on all calls to Java so that we could use the Braintribe CSP client API. We decided to use TCP/IP communication to pass the commands to Java server, which would allow the Java module to be run on a different machine.

For the Java module we chose to use API architecture to allow an effortless implementation of different kinds of Java-based filesystems. The exact architecture and implementation is described in the following chapters.

[2] LUFS – Linux User-space Filesystem [lufs]

4. Filesystems

To grasp the idea behind filesystems, one must understand what the user sees and why things are presented the way they are, what concept lies behind it all.

The basic and most common elements of a filesystem are regular files and directories, with regular files being the most spread type of file in operating systems. Sources, programs, scripts, pictures and documents are all regular files. One might argue that executable files are a special type of regular files but they are handled by the filesystem in the same way – as a stream of bytes.

Although directories also contain a stream of bytes, these are interpreted by the filesystem in a different way. They contain the information about files and directories that are present in it and how they are linked together, giving the user a hierarchical perspective over the files and directories. Therefore files and directories in a filesystem may be also presented in tree-like form; where everything other then a leave is a directory, see figure 2.

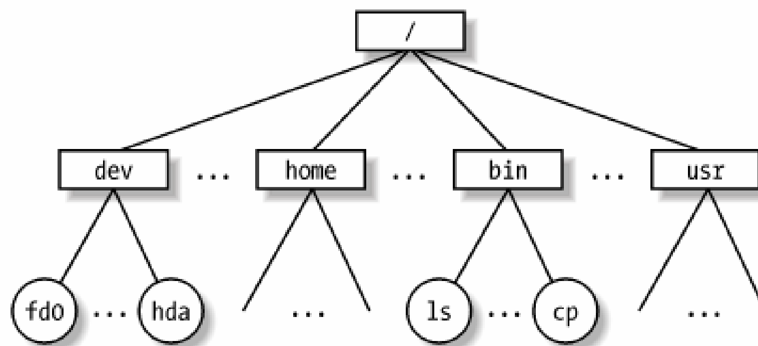


Figure 2: An example of the directory tree.[3]

There are also other types of files and directories that need to be considered when building a filesystem, but not all of them have need to be supported. They are outlined here.

[3] Figure from [utlk02]

- **Regular files.** Regular files hold any data that is not interpreted by the filesystem. They include for example script, documents, videos, images and binaries.
- **Directories.** Directories provide structure in the filesystem. They catalog entries of files or other directories. Data they contain is being interpreted and managed by the filesystem. The data conforms to the virtual filesystem standard.
- **Symbolic links.** A symbolic link, also called symlink, hold a reference to another file or directory. They don't have to have the same name as the referee and they may cross through different filesystems. Removing a symbolic link does not have any impact on the referenced file. Removing or renaming the indexed file causes the symbolic to be invalid or broken, although the symbolic link is not removed. A broken symbolic link may become again valid, when the referenced file»reappears«. Symlink is also translated by the filesystem. Usually it just holds the path to the referee as a null terminated string, which the filesystem tries to find.
- **Hard links.** In contrast to symlink, hard link isn't just a simple mapping with any impact on the referenced file. By creating a hard link the file's link count is incremented, which on the other hand deleting decrements. When the file's link count hits zero, the file is removed from the filesystem. Hard links cannot cross filesystem boundaries.
- **Named pipes.** Named pipes are part of *Inter Process Communication* (IPC) mechanism, which allows unrelated processes to bi-directionally communicate. This differs from traditional UNIX pipes that can only by accessed by related processes and which operational unit is built-in in the operational system.
- **Special files.** Special file is a file that refers to a character or block device such as a disk or a tape. This allows the user of the file to access the device with the file's read write methods. These files can be found in devfs or procfs.

4.1 Most Common Filesystem Objects

This chapter explains some of the filesystem's object that will be used later in this thesis to explain some of the processes in the VFS.

4.1.1 Inode object

Inode is an object that »lives« on the disk and represents each file on the filesystem. Every file accessed, independently of its type, is held in the virtual filesystem as an `inode` structure. The `inode` structure has a number of members used by the VFS to manage access to the file and to store some of the file's properties. Among others it contains the i_ino field (inode id or inode number) that is a unique 64bit value for a file and identifies the file in the specific filesystem. The virtual filesystem holds exactly one such structure for a file in the cache, and discards it from the cache after some time if there is no outstanding `open` method and it is not needed any more.

4.1.2 Dentry object

Dentries are not disk objects; they are dynamically allocated in RAM to enhance performance and therefore are never saved to disk. VFS stores filesystem's structure in memory as a linked list of dentries. The `dentry` structure (defined in *linux/dcache.h*) is used as the dentry's representation and contains managing mechanisms and a reference to the inode object. In order to resolve a pathname into a dentry VFS may resort to allocating dentries along the way. For relevant dentries the inodes are loaded and associated with the corresponding dentry.

4.1.3 File object

File object is represented by the `file` structure. It holds the information about the file interaction in scope of a process. This object is created when a file is opened and freed when the last reference is released. It doesn't have any corresponding disk image, although it includes a reference to the dentry object that further references the inode object.

The main information stored in a file object is the offset position in the file data from which the next operation will take place. It also stores the interface with file operations implemented by the corresponding filesystem.

How all these objects interact with each other shows the figure 3.

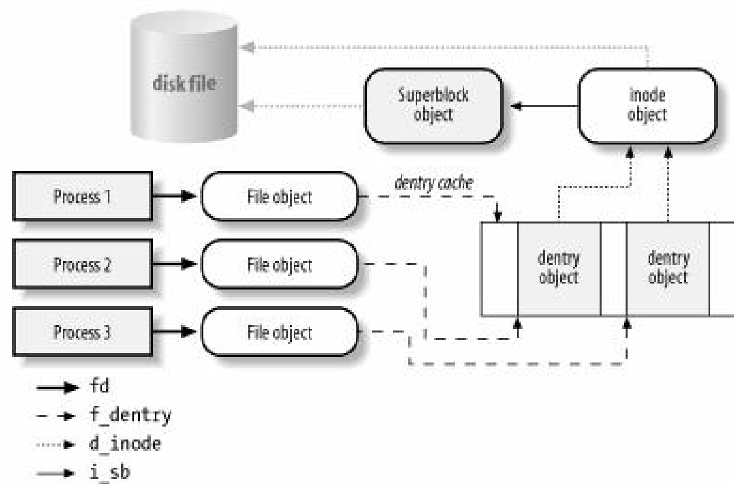


Figure 3: Interaction between processes and VFS objects.[4]

4.2 Basic File Properties

To obtain the properties of any file the `stat ()` system call may be invoked. A user may use the *stat* or *ls* commands, which also invoke the `stat ()` system call. One line of the result of an *ls* command is showed and described in figure 4.

The main properties showed are:

- The File type and access permissions
- The link count
- The owner and group of the file
- The size of the file
- The last modification date of the file
- The name of the file

[4] Figure from [utlk02]

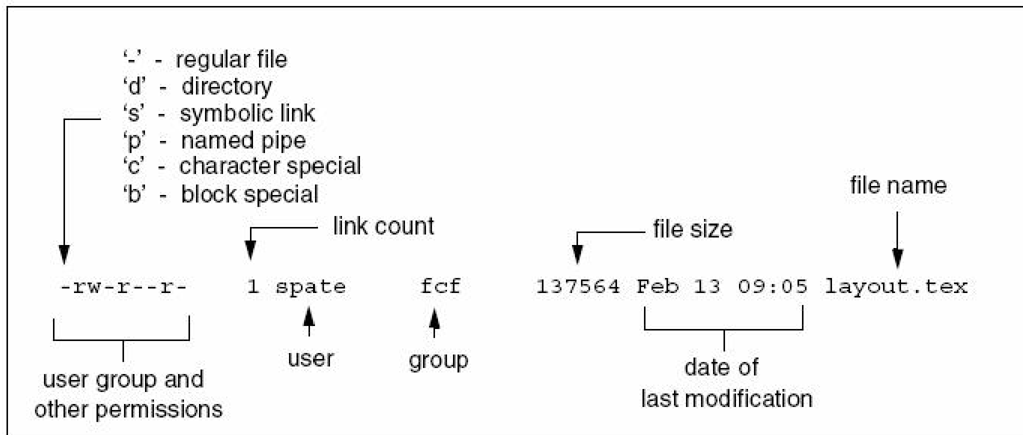


Figure 4: File properties shown by typing *ls –l*[5]

First the *ls* command obtains the list of all files in the current directory using the `getdent ()` system call and then it calls `stat ()` for each file to get the file's properties. The interface of these two system calls is defined in *sys/stat.h* and looks like this:

```
int stat(const char *path, struct stat *buf);
int getdents(unsigned int fd, struct dirent *dirp,
             unsigned int count);
```

The `getdents` system call uses a file descriptor[6] to obtain the entries in the directory. The caller provides the buffer for a list of `dirent` structures that is filled in the `getdents` call. The `dirent` structure contains the inode id and file name that is subsequently used to make a pathname and passed to the `stat ()` call that returns the file's properties in a `stat` structure (defined in *sys/types.h*), which has the following form:

```
struct stat {
      dev_t st_dev; /* ID of device containing file */
      ino_t st_ino; /* Inode number / file serial number */
      mode_t st_mode; /* File mode */
      nlink_t st_nlink; /* Number of links to file */
      uid_t st_uid; /* User ID of file */
      gid_t st_gid; /* Group ID of file */
      dev_t st_rdev; /* Device ID for char/blk special file */
```

[5] Figure and description from [ufedi03].
[6] To access the file's data, the file must be first opened using the *open system* call. The result of a successful open is the file descriptor that is further used to access the file.

```
        off_t st_size; /* File size in bytes (regular file) */
        time_t st_atime; /* Time of last access */
        time_t st_mtime; /* Time of last data modification */
        time_t st_ctime; /* Time of last status change */
        long st_blksize; /* Preferred I/O block size */
        blkcnt_t st_blocks; /* Number of 512 byte blocks
                               allocated */
};
```

Given this information it is relatively simple to map the structure fields to the information displayed by *ls*. This structure is not easily altered. Usually the filesystem updates this information. If you want to change the fields manually, than you have to use system calls like chmod, chown, utime or truncate, which are mentioned in VFS. Additional information can be found in Linux manual pages.

Besides the standard file properties Linux VFS supports extended file attributes. These may be customized by the filesystem. The *getfattr*[7] command may be used to obtain the attribute names and values of a specific file. Typically only attributes with names starting with *"user."* prefix are displayed. To get a value of an attribute without the user prefix, one must know the attribute's name. These attributes aren't yet widely used, although some security systems like SElinux do use them to store some information about the files (e.g. *security.selinux* attributes).

System calls responsible for handling extended attributes are `fgetxattr`, `lgetxattr` and `getxattr`. The VFS functions are described later in this chapter.

4.3 Relation to Database Systems

Filesystems and database systems have a totally entangled relation. A filesystem is not a database system, although it may use one to manage the filesystem data. A database system is not a filesystem, although at some point it certainly uses a filesystem (if it doesn't use direct device calls) to maintain its data. The main difference is the data organization where filesystem obey a standard structure and

[7] *getfattr* is not a standard Linux command. It must be installed with the attr package, although it is mostly included in the basic installation.

database systems enjoy the freedom of the database architect. Another foremost discrepancy is the data access. Both have standard interfaces to implement, but they are obviously different (Select, insert, update vs. open, read, write).

Of course it is possible to design a database system with a structure adhere to the filesystem data organization, see figure 5. Such a database could easily be used by a filesystem. It would just need to bind to the VFS by implementing its interface.

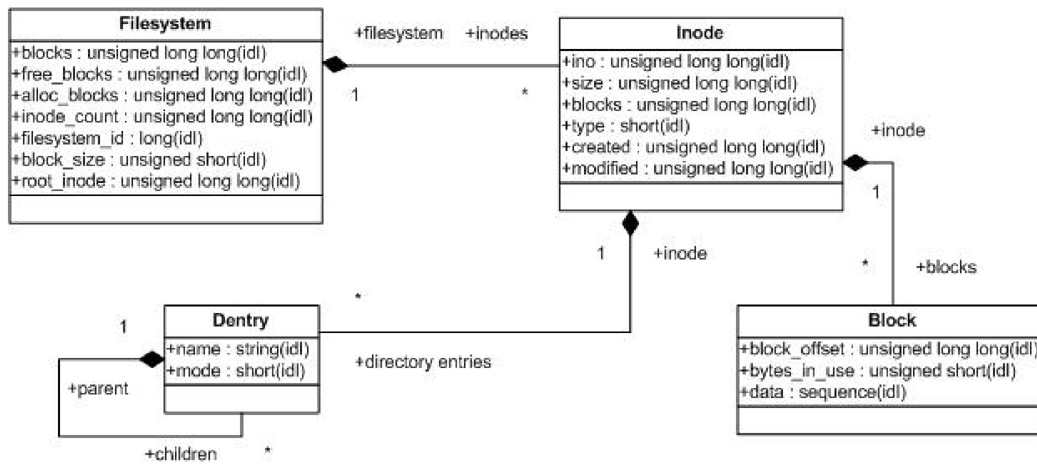


Figure 5: An example of a database model conforming basic filesystem data organization

4.4 Virtual Filesystem (VFS)

Virtual Filesystem (also known as Virtual Filesystem Switch or VFS) is an abstract kernel software layer that handles all filesystem related calls and delegates them to the underlying filesystem implementation. As mentioned earlier, it provides a standard interface for all filesystems. A *cp /mnt/floppy/foo /home/user* command would then look as in figure 6.

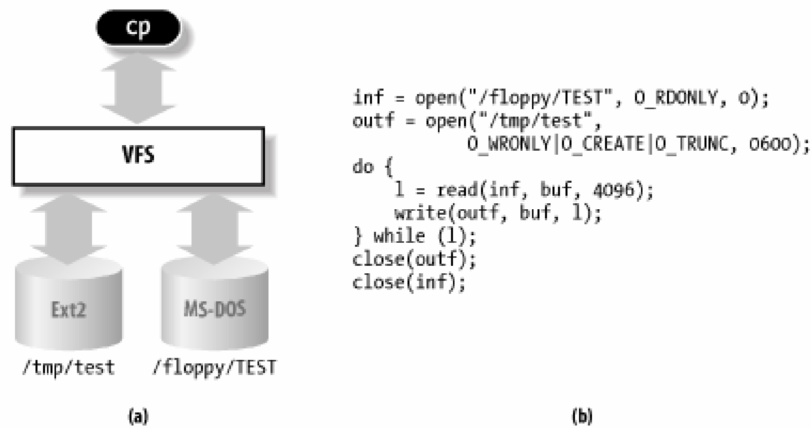


Figure 6: VFS role in a *cp* operation[8]

There is no `cp ()` or `copy ()` system call, the *cp* command is a program that reads (`read ()` system call) data from one file and writes (`write ()` system call) them into another file.[9] In our example the command crosses between two filesystems, the ext2 and msdos filesystem. The VFS acts as an abstraction layer between the application (the *cp* program) and the filesystem implementations, therefore the application doesn't care about the filesystems the files are stored in and interacts with the VFS only by means of generic system calls.

4.4.1 Registering and Mounting a Filesystem

In order to access a filesystem, it has to be mounted. Each filesystem has its own root directory where its filesystem's space begins. If filesystem's root directory is also the root of the system's directory tree, then the filesystem is called a root filesystem. Other filesystems may be mounted at any point on the system's directory tree; and other on the directory tree of the mounted filesystems. The directory on which the filesystem is inserted is called mounting point, and the filesystem that the directory belongs to, is called the parent filesystem.

Mounting a filesystem is done by the mount command.
Example:

[8] Figure from [utlk02]
[9] Of course the cp program invokes other system calls as well (e.g. open (), stat(), etc.)

mount -t ext2 /dev/fd0 /flp

where first argument is the device associated with the filesystem, second argument is the mounting point and the -t flag argument specifies the filesystem. To view all the currently available filesystems, execute the command *cat /proc/filesystems.*

It is possible to mount multiple filesystems on a single mount point, although it is very rare. The last mounted filesystem always hides the previous, although applications working with files from the previous filesystem may continue to do so. When the topmost filesystem is removed (unmounted), then the previous filesystem is made visible again.

Registering a filesystem

To support a new filesystem in the kernel, the filesystem needs to be registered. This is not done by the mount command. The filesystem must be already registered at the time the mount command is executed. Typically the filesystem is registered in the init function of its kernel module when you run the *insmod* command.

Registering is provided by the `register_filesystem ()` kernel function. You pass it a `file_system_type` structure that describes your filesystem and your filesystem is available for mount.

```
struct file_system_type {
    const char *name;
    int fs_flags;
    struct super_block *(*get_sb) (struct file_system_type *,
                                  int, const char *, void *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;
};
```

name – is the name of the filesystem, such as »ext2« or »fat32«.

fs_flags – various flags (i.e. `FS_REQUIRES_DEV`, `FS_NO_DCACHE`, etc.)

get_sb – reads the super-block of the filesystem on mount

kill_sb – frees all necessary structures of the filesystem on unmount

for internal VFS use

owner – module responsible for the filesystem, instantiated with `THIS_MODULE`

next – next filesystem in the list, this should be initialized to NULL

fs_supers – list of superblocks, initialized to NULL

`get_sb` and `kill_sb` are part of filesystem's implementation.

Internal mount

Internally the mount command calls the `mount ()` system call (with `sys_mount ()` service routine). The `sys_mount` temporarily copies the parameter into a kernel buffer, locks the global kernel lock and invokes the `do_mount ()` kernel function that does all the work. The next pseudo code shows some of the important steps the `do_mount` function does.

1. If any mount flags (e.g. `MS_NODEV`, `MS_NOEXEC`) are passed then set the corresponding flags (e.g. `MNT_NODEV`, `MNT_NOEXEC`) in the mounted filesystem object.
2. Lookup the pathname of the mount point
3. Check the mount flags to determine what has to be done
4. Invoke `path_release ()` to terminate pathname lookup of the mount point

The core of the mount operation is done by the `do_kern_mount ()` function that does most of the above-mentioned steps. Among these tasks, it allocates a `super_block` structure partially initializes its members. The rest is then set by the filesystem's `get_sb ()` function (specified in the `file_system_type` structure used in the registration process). This object provides an interface, `super_operations` stored in field `s_op`, which is then further used to access the filesystem. The interface is described in section VFS interface.

4.4.2 Insight into a simple procedure

This subchapter describes the call flow of an `open`, `read` and `close` system call. This is one of the most common scenarios carried out by an application.

Opening a file

Any access to a file must be preceded by the open method. The `open ()` system call is handled by the `open ()` kernel function. It receives the file's pathname as argument; therefore its first task is to call the `namei ()` that looks up and parses the dentry cache (filesystem's tree) and on success returns the inode number of the inode object referenced by the last dentry.

Next the kernel `open` function calls an `open1 ()` function, passing it its opening mode argument. The split between the `open ()` and `open1 ()` allows `open ()` and `creat ()` system calls to share much of their code. To ensure that the process has permission to access the file with the passed mode, `open1` in first place invokes the `access ()` method. If all is fine, a call to `falloc ()` is made that allocates a file table entry (file structure) and internally calls the `ufalloc ()` to allocate a file descriptor from `u_ofile[]` array. The allocated file descriptor will be set to point to the allocated file table entry and a reference to the inode object is established.

Reading the File

The system call `read ()` is invoked with a file descriptor parameter and a buffer that will be filled with the data from the file. At first it obtains the file table entry and verifies the permission to read. The next steps involve repeated calls to `bmap ()` function to obtain the disk block address from the file offset. `bmap` maps the logical block number within the file into a physical block number on the disk. This is passed to the `bread ()` function that reads in the blocks on the disk. To fill the passed buffer with read data calls to the `copy_to_user ()` function are made. This buffer may not be directly accessed, because the buffer is not in the kernel address space. After a successful read, the file's offset in the file structure is modified and the number of bytes that have been read is returned. The `read` system call always reads the requested number of bytes, if not, the specific error number is returned.

Closing the File

The `close ()` system call is handled by the `close ()` kernel function. It does do little work other than the opposite of open. It obtains the file table entry calling the `getf ()` function, zeros the appropriate entry in `u_ofile[]` and invokes the `closef ()` function. The file table entry cannot be simply freed because it may be referred by another file descriptor. This may happen, for example, when a `dup ()` call was made on the file descriptor. For this purposes the reference count of the file is stored in the file table entry in the field `f_count`. If `f_count` is 1, then there is no other reference and the file table entry may be freed and a call to the `closei ()` is made to free the inode.

`closei` invokes `iput ()` function that checks inode's reference count `i_count` and releases it if it reached 0. Additional work is performed in `closei` to allow a device call in the case the closed file was a device. One additional check that has to be made is to see if the hard link count of the inode is not 0. If it is 0, what means that an `unlink ()` call was made on the file while it was opened, then the inode may be freed on the disk as well.

4.4.3 VFS Interface

Linux kernel is written in C, which isn't an object-oriented programming language; therefore all the objects and interfaces are represented in form of the structure type. The VFS interface includes longer lists of supported operations and interface structures, which is together with brief descriptions situated in appendix A.

It is useful to state that some of the operations where added to the VFS interface to support specific filesystem processes (e.g. `dir_notify` for the Common Internet Filesystem (CIFS)). Not all interface members are therefore used. Unused or not implemented members of the interface should be set to NULL and will be ignored by the VFS.

5. FUSE

FUSE (Filesystem for User-Space) is a simple API for developers that exports virtual filesystem interface to user-space. It is an open-source project created by Miklos Szeredi. It can be downloaded from: *[prjfuse]* or from FUSE's homepage *[wwwfuse]*.

In this chapter is explained its implementation, design and how it is used. To get the overall picture about FUSE it is helpful to understand what a user-space filesystem is and why are they useful, therefore the next subsection is dedicated to user-space filesystems.

5.1 User-space Filesystem

Filesystem that provides data and metadata from user-space is called user-space filesystem. As stated in chapter about filesystems, Linux filesystems must be implemented and linked into the kernel. Such an implementation restricts the developer to work in kernel's address space. Because of the different address space a kernel module can't call user-space standard functions and a user-space application can't call kernel functions. This makes complex kernel modules even more strenuous.

To facilitate the implementation such demanding kernel modules, developers move parts of the implementation into user-space. There are only few techniques (described in next subsection) for the kernel-user-space communication, which are described in next subsection, and programmers are forced to use them. To prevent repeated implementations of the same code, open-source programmers make a framework that can be then used by everyone.

This was also the motivation behind user-space filesystems, to facilitate the development process of a complex filesystems. With same scenario as any other kernel module, it became obvious that a framework for user-space filesystems would be useful, and among other frameworks, FUSE was born and became popular in user-space filesystem development under Linux.

5.1.1 Kernel-User-Space Communication

As described above simple function calls don't work between kernel and user-space, therefore some special techniques are used instead. This problem is different for the directions of communication therefore they will be treated here separately.

User ➔ Kernel Calls

To call a kernel function from user-space one may call a system call, which typically invokes some kernel functions. In general developers don't have their own implementations of system calls. Adding a new system call into the system involves couple of steps, with the last one being the rebuilding of the kernel. As a rule programmers try to avoid rebuilding of kernel as it opposes the philosophy of its kernel design.

Instead of making an own system call, it is better to get use of an existing one. The question is how to make a system call execute own implementation and this is where special filesystems like proc or dev step in. They provide interface that kernel modules can use to register their own directories and files with own implementation of the file operations (i.e. read, write, ioctl, etc.).[10]

Using proc filesystem for user-kernel communication is very common, although dev filesystem provides similar functionality and becoming also very popular. Proc entries are registered via `create_proc_entry` function. The caller sets then the `read_proc` and `write_proc` pointers to its implemented functions. Dev users register their devices with `misc_register` function. They pass it a structure with information about the device name, device number and file operations.

Kernel ➔ User Calls

Calls from kernel to user-space have to be handled differently. They use the same manner (i.e. proc, dev, etc.) for the data exchange, but the procedure is other. There

[10] There are many kernel modules that use the proc filesystem (typically mounted in /proc). Kernel modules use it, to allow user-space applications access to some of the internal information. For example the file cpuinfo in proc filesystem is handled by kernel module that returns CPU information (i.e. CPU frequency, flags, cache size, etc.)

must be a user-space application running in background as a deamon that will handle the commands from kernel-space. These calls are done by combination of read and write operations (figure 7).

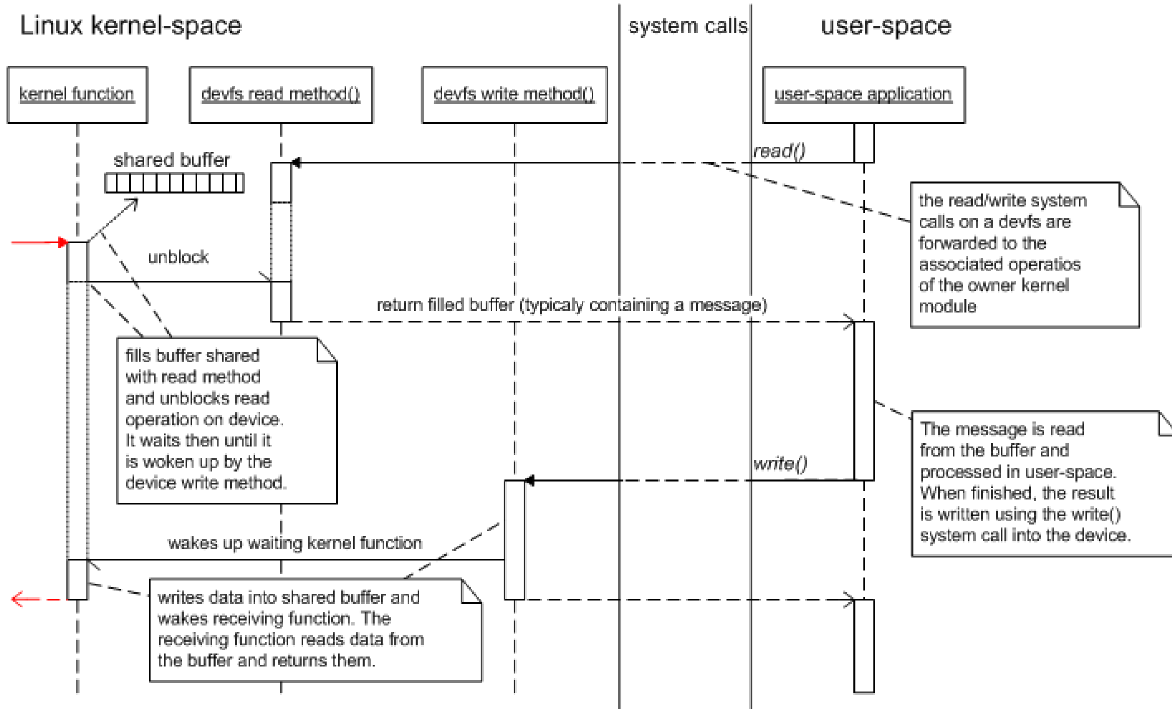


Figure 7: Sequence diagram of the kernel to user-space call. The invoking and returning of the special call are represented by the red arrows left.

First the deamon process calls the `read` system call for the desired file in proc filesystem. The `read` system call invokes the read operation of the file, which was set by the registration of the file. In the body of the read method the process blocks and waits until the kernel module will have an outstanding request that should be processed in user-space. When the kernel module will want to make use of the user-space functionality, it fills the buffer in read method, unblocks it and blocks it self. The deamon interprets the returned data as a command with arguments, calls the necessary functions and returns the result by calling the `write` system call on the file owned by the kernel module. The write operation will be delegated to the kernel module implementation that sets the result buffer and unblocks the kernel module block.

Of course the data exchanged is not just an array of bytes, it has to be correctly interpreted by the deamon process. Therefore the communication or request protocol needs to be implemented in both, kernel and user-space.

The procedure used for kernel to user-space exchange is very much the same as two producer-consumer problems. First the deamon plays the role of the consumer and kernel module is the producer, and the product is the request. In the returning process are the roles switched and the product is the data to be returned.

Besides proc, dev or other special filesystems sockets may be used for the kernel-user-space communication, though the implementation is more complicated. Because the kernel module can't make use of the `socket`, `open`, `read` and `write` system calls, the corresponding low-level functions have to be used. The call-flow would be same as in the special filesystem case.[11]

5.2 Design

FUSE is a medium size open-source project. It can be divided into three main modules. One is the kernel module responsible for the implementation of VFS and communication with user-space. The other two are user-space implementations; one is the FUSE API library and the other is a small utility called *fusermount*. These modules and their cooperation will be described in next subsections.

5.2.1 FUSE kernel module

As every filesystem, FUSE must implement a kernel module. This module implements the callbacks of the VFS interface. Calls from VFS are translated into command and sent in request structures forward to the user-space. In the initial phase of the module it registers the filesystem and the device for the user-space communication.

[11] You can find more about socket programming in kernel-space here *[ksockprog]*.

The implementation of the kernel module lies in the kernel directory (in FUSE source code). To prevent name conflicts inside kernel, all exported FUSE functions have the prefix »`fuse_`«.

fuse device

FUSE implemented kernel to user-space calls using the dev filesystem. Its source code is in the file *kernel/dev.c*. The methodology of the kernel to user-space communication in FUSE is very much the same as suggested in previous chapter. Read method blocks and waits until a request is pending. When a request is created, its structure representation is copied into read buffer and the request owner function blocks. In write operation is the result written into the requests results buffer and the request owner is woken up. The implementation details are showed in next paragraphs.

It isn't bound to any real device; it is just a software imitation. The device is registered by calling `misc_register` function, passing it the FUSE `miscdevice` structure. The minor device number is set to FUSE_MINOR (defined in fuse_kernel.h), device name is »fuse« (with this name will be created a character device in */dev* directory) and file operations are set with the `fuse_dev_operations` structure. Another task the initialization function of the device has to do is to establish the request cache (with the name »`fuse_request`«) using the `kmem_cache_create` function. This is then used to allocate new requests, which are represented by the `fuse_req` structure defined in fuse_i.h.

Besides the request data, which can have the maximum size of `FUSE_MAX_PAGES_PER_REQ` (defined in fuse_i.h) pages[12], it includes control members such as the current state of the request, request id, reference count, wait-queue, etc. As you may see blocking of request owners is implemented using wait queue[13] (`waitq` member); and as you will further see, all blocking mechanisms are done with wait

[12] A page has typically 4096 bytes and `FUSE_MAX_PAGES_PER_REQ` is defined as 32, therefore the maximum size of request data is 32 * 4096 = 131 072 bytes or 128kb.
[13] Further information about wait queues may be found in Linux Kernel Internals [lki00].

queues and spinlocks[14]. A sequence diagram depicting this process is shown in figure 8.

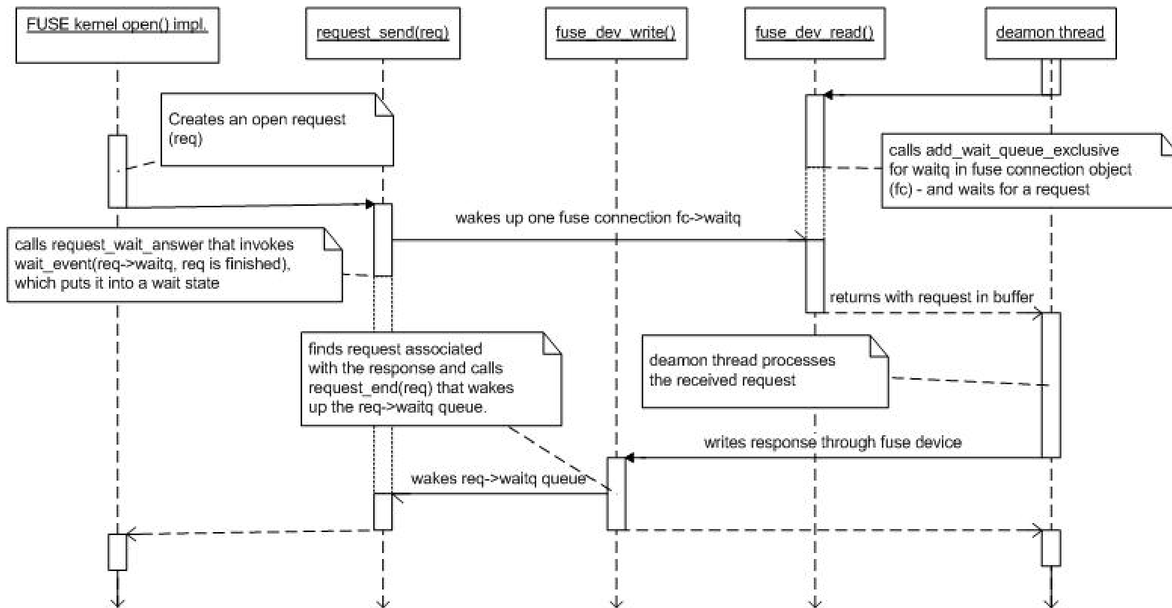


Figure 8: Sequence diagram of the FUSE filesystem request processing.

Sequence diagram in figure 8 shows 5 functions of two threads; some control mechanisms and special handling is left out for simplicity. One thread belongs to the FUSE deamon process that is started by mounting the filesystem. It opens the FUSE device file and calls the `read` system call, which in a trace of calls invokes the `fuse_dev_read` function shown in the figure; the read system call and other functions in the stack trace are not depicted as they are of no relevance here. Then the read method calls the `request_wait` function, which uses `add_wait_queue_exclusive` fuse connection object (`fuse_conn`, described in next subsection) member `waitq` to add a task entry into the queue.

In this case exclusive means that `wake_up` call wakes up only one exclusive task. On wake up is the owner process state set to `TASK_RUNNING` and therefore on next schedule it will have assigned CPU time.

[14] Spinlocks should be used only for short time, for example testing/setting couple of variable, because it uses CPU time. Wait queues have a greater overhead than spinlocks; therefore they should not be used for short time blocking. Wait queues are instead used when the caller waits until some longer process finishes.

The request_wait function implements the block mechanism in a special way. Instead of a wait call it runs a loop where it enables interrupting, sets the owner process state to TASK_INTERRUPTIBLE (this means that this process won't be rescheduled until it is woken up) checks whether the process has a pending signal, if there is no signal then it calls the schedule kernel function that reschedules another process. Now it waits until the wake_up call wakes up this process by marking the process as running, otherwise it won't be rescheduled. When it is woken up, it checks whether the fuse connection isn't disconnected and whether there is a request waiting to be processed, if not the whole thing repeats itself. This might look complicated therefore here are the steps again from the point of view of the blocking process:

1. Add ourselves to the reader's wait queue (fuse connection field waitq)
2. Mark ourselves as sleeping (TASK_INTERRUPTIBLE)
3. Ask the kernel to schedule tasks again (schedule function)
4. The kernel sees we are asleep and schedules some other process
5. The next request interrupt sets our state to TASK_RUNNING and notifies the kernel to reschedule tasks
6. The kernel sees we are running again and continues our execution

The other thread belongs to a process that called the getdents system call on a FUSE filesystem directory. Again this figure doesn't show the caller stack trace functions and illustrates only the FUSE function that handles the getdents system call, the fuse_readdir function. The function fuse_readdir checks some parameters, prepares the request structure and invokes the request_send method with the request as parameter. This method wakes up the reader's queue (fuse connection member waitq) and executes the request_wait_answer that using the wait_event_interruptible kernel method (with the request waitq member as parameter) blocks until the request is answered (i.e. when the queue is woken up with wake_up call).

Now the deamon thread in read method is unblocked and returns the request structure that is to be interpreted and processed. When the deamon is finished with the request,

it calls the `write` system call that further calls the `fuse_dev_write` function as illustrated in the figure. The write method finds the corresponding request structure by its request id in the list of all requests being processed (stored in fuse connection), attaches the answer to the request and wakes all processes waiting on this request by calling `wake_up` call on the `waitq` member of the request structure. The write method is hereby done and returns to the deamon.

The woken up processes waiting for the request continue in the `request_send` function that before returning unlocks the fuse connection spinlock. The execution is continued in the specific FUSE function, in our example `fuse_readdir`, which reads and deals with the results of the request, and returns (through the `getdents` system call to the user-space program).

FUSE VFS interface implementation

The VFS interface is in FUSE implemented in three files: file.c, dir.c and inode.c. The names of the files give already an insight into which operations are where implemented.

Operations are straightforward implemented without any sophisticated logic. The body of a typical VFS interface function does the following tasks:

- Allocate and prepare a request object
- Send request object via `request_send` function to user-space
- Use the request result to finish the work of the specific operation

This implementation also includes special FUSE mount options. In function `parse_fuse_opt` are the options interpreted and corresponding members of the `fuse_mount_data` object are set.

5.2.2 fusermount

The utility *fusermount* is a user-space program that is used to initialize the mount process – loads fuse kernel module if necessary. Besides this job it calls the actual `mount` system call. It is implemented as a different program in the directory util in the

FUSE source code directory. Typically it is executed from the `fuse_mount` function implemented in the FUSE library. It may be also called from command line with different options. The most used one from command line is the unmount (-u) option. In case the user-space filesystem has problems with unmounting, *fusermount* may be called with this option to force the unmount. While the filesystem is used, this module is not used.

5.2.3 FUSE library

FUSE library is the user-space part of FUSE. It provides functions for the developers that would like to use FUSE API. It is designed like the virtual filesystem. The user may choose only to implement a predefined set of functions, which are then attached to the FUSE deamon implementation, or uses some of the library functions to create a new deamon. The list of all functions in the library can be found in the file *lib/fuse_versionscript*.

A programmer using FUSE library may also choose to between two abstraction levels; FUSE interface and FUSE low-level. Low-level interface is defined by *fuse_lowlevel_ops* structure and FUSE interface is represented by *fuse_operations*. The basic difference is that the FUSE interface makes the use of inodes transparent. Files and directories are instead identified with the full path, although the FUSE interface uses the low-level functions. Its most overhead lies in the translation of the inodes into path; otherwise both abstraction levels share the code.

The implementation itself contains also some levels of abstraction. One was already mentioned in previous paragraph, the inode abstraction level. The fuse_kern_chan.c file implements another abstraction level, the communication channel level. The communication channel provides the `fuse_chan_ops` interface, this includes the implementation of receive, send and destroy functions.

To sum it up, the FUSE library offers an implementation of the communication with the device, the deamon that repeatedly interprets the commands from device and writes the answers back, parses the mount options and allows a debugging useful verbose mode.

5.2.4 Call-Flow

As we already mentioned, FUSE can be divided into three modules, two of them running in user-space. This section offers an insight into the interaction between the modules and depicts the call-flow of a three scenarios. First scenario being the initialization and mount process of FUSE, latter scenario is the typical handling of a command in the running state, and last is when the FUSE is being closed – unmounted.

Initialization

FUSE initialization starts with the *insmod* command that links FUSE kernel module into Linux kernel and starts its init function. The FUSE kernel module init function is `fuse_init` and its source can be found in file kernel/inode.c. It allocates and initializes some objects, registers the FUSE filesystem and the FUSE device, which is then accessed over the /dev/fuse file.

When the FUSE kernel module is loaded in Linux kernel the filesystem can be mounted. The mounting of FUSE is different from other filesystems in that it isn't mounted with the *mount* command, although it calls the `mount ()` system call in the process. Filesystems based on FUSE are built as a program with the main method and are started as commands. The program implements the FUSE interface and in the main function it may initialize its own objects. In the initialization it may use any of the FUSE help functions, like `fuse_parse_cmdline` that parses the arguments and returns the mount point. At the end of the initialization the program must call the fuse_main (or in the case of a low-level filesystem the `fuse_mount`) method to start the filesystem – that is, mount the filesystem. To use these functions the program must include the header file include/fuse.h and link with the FUSE library.

Here starts the initialization of the FUSE `fuse_main` parses the passed arguments and calls the `fuse_mount` function. `fuse_mount` creates a UNIX domain socket pair and executes the *fusermount* command, which gets one end of the socket as a file descriptor in the FUSE_COMMFD_ENV environment variable and in command line arguments the mount arguments passed to `fuse_mount`.

The command *fusermount* checks whether the fuse module is loaded, and loads it if not. Then it opens the FUSE device – /dev/fuse – and sends the obtained file descriptor through the UNIX socket. The `fuse_mount` reads the /dev/fuse file descriptor from the socket and closes it as it won't be of any need anymore. Then it initializes a fuse channel object and returns the pointer to it.

Now either the `fuse_main` or the program's main function runs deamon function that uses the obtained fuse channel object to capture filesystem system calls and to send the results back to the kernel-mode process. The `fuse_main` prepares its fuse object, which is used to store and maintain configurations, and uses the `fuse_loop` (or `fuse_loop_mt`) functions to start the main deamon loop.

Running

Now when FUSE is loaded and the user-space filesystem is mounted, the deamon sleeps in the `read` system call of the FUSE device that was invoked from the receive operation of the fuse channel object, and waits for a command or an interrupt. The figure 9 shows the path of a call.

As soon as a filesystem system call invokes one of the FUSE kernel functions, the blocked read method is woken up and a request structure is created and sent over the `read` syscall buffer to user-space. The FUSE library deamon loop reads the command from the device and calls the specific function from the operation structure filled by the filesystem program.

When the filesystem user-space function is done with the command, it calls a specific FUSE library function to send the command results such as the `fuse_reply_err` function that is used to reply an error. The FUSE library function then uses the fuse channel object to write the answer to FUSE device. The FUSE device write method finds the corresponding request, hangs the answer to it and wakes up all the tasks waiting for it.

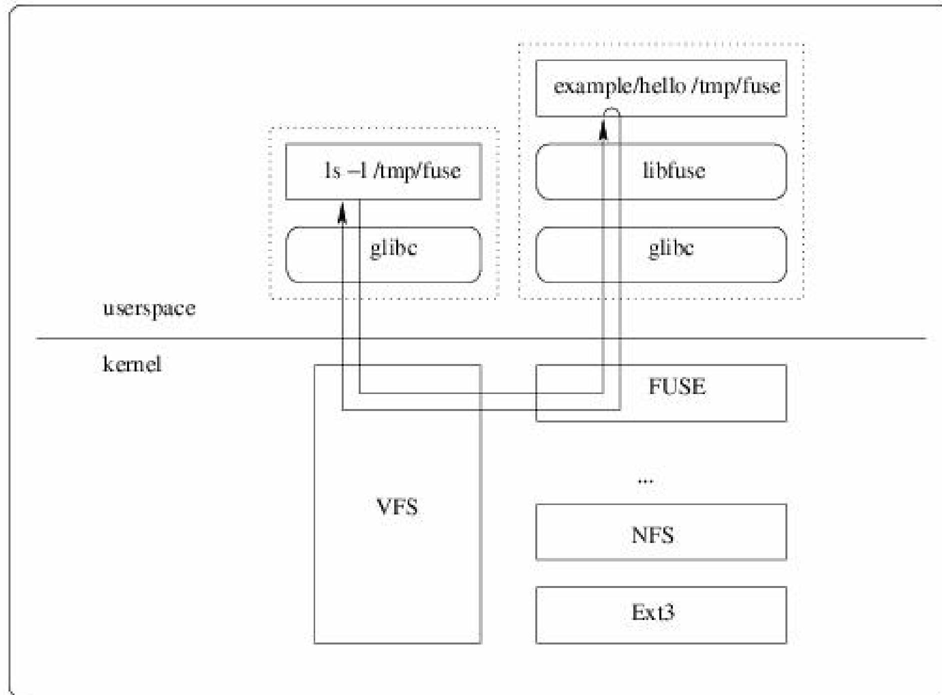


Figure 9: This figure shows the path of a filesystem system call. It can be found on FUSE homepage [wwwfuse].

What is not obvious on the picture is that this call-flow involves two processes. One process is the FUSE deamon that runs all the time and doesn't stop after this communication exchange. It handles all the requests of the filesystem while the filesystem is mounted. The other process initiates the call-flow, in our example it is the *ls* program. Life of this process spreads usually only over a couple of requests.

Unmounting

When a user is done with the user-space filesystem, he may unmount it. He can either use the *umount* command, the *fusermount -u* command provided from FUSE or kill the FUE deamon process. The last option differs from the other, and is possible only if you capture the SIGINT signal, what the FUSE library deamon does.

In case the filesystem was unmounted either with *umount* or *fusermount -u* command, the `umount ()` system call is invoked. This command furthermore invokes the FUSE `put_suser` function that creates a destroy request and sends it to user-space. When the request is done, `put_super` closes fuse connection object, releases some objects and wakes up all waiting processes that return with an error.

Next FUSE device read operations don't block anymore and just return with the `ENODEV` error code that breaks the deamon loop. A poll () system call on the device returns `POLLERR` error code what means that the filesystem is already unmounted.

If the deamon process is being killed, the unmounting looks slightly different. The deamon signal handler calls the `fuse_session_exit` function that sets the session as exited and closes the FUSE device file. This breaks the FUSE deamon loop and exits the process.

5.3 Basic FUSE Structures

Just like the implementation of virtual filesystem, FUSE objects are represented in structures. Besides objects, structures are used to implement the kernel-user-space communication protocol. There are structures that are used specifically in FUSE kernel module and structures used only in FUSE user-space part – library.

5.3.1 FUSE kernel structures

The reason that some objects are needed only in kernel module is because the kernel and user-space are so different and because they have different roles. They are declared in kernel directory in fuse_i.h header files. There is also another header file in this directory, fuse_kernel.h, but this file declares the FUSE communication protocol structures and is therefore used also in user-space module.

fuse_i.h is included in every FUSE kernel module C file.

The first couple pages of fuse_i.h set version macros and default value macros and include some other necessary header files. The first object that you encounter is fuse inode object, `fuse_inode`. This object is stored instead of the Linux kernel inode object; therefore the first member in the structure is reserved for the Linux kernel object, although it doesn't have to be the first member.

The allocation of the Linux inode object is done entirely in FUSE implementation, although it will be used by the virtual filesystem. Therefore any reference to Linux inode object returned from a VFS function is in memory a fuse inode object and can be translated to it using the `get_fuse_inode` function (defined also in fuse_i.h).

Next member is the fuse inode identification object, `node_id`, used to identify the inode between fuse kernel and user mode. The other fuse inode object members are used in maintenance. `nlookup` holds the count of lookups performed on this inode. On each forget operation is this count decremented by one. The fuse inode object stores the reference to forget operation request object in the `forget_req` field. Last field is `i_time` and it defines the life-time of the inode attributes – it holds the time in jiffies until the attributes are valid.

After inode object definition comes fuse file object representation, `fuse_file`. It is a small object and holds only a reference to a request object that is reserved for the flush and release commands (when a file is released), and a file handle used in user-space.

FUSE request object is on the other hand one of the largest objects defined in FUSE. It is defined on couple pages, including some other objects. This objects stores all information relevant for the execution of the command and many control fields used to manage the process of execution. First fields store lists of pending processes and tasks. The `count` field saves the reference count to this object to ensure that the object is not released when some process still uses it. The `intr_unique` is used if this request is an interrupt request that has higher priority than other requests and is used to identify this interrupt request.

The next dozen fields are dedicated to request flags (e.g. `isreply`, `force`, etc.) and to store the state of the request.

The following `fuse_in` and `fuse_out` objects in the request store the request input arguments and output – answer. They define the number, size, buffer and some flags

of the arguments/answer. The request `pages` and `num_pages` members are therefore used only if the fuse io objects need them.

The `waitq` member was already mentioned in Call-flow subchapter, and is used to wake up processes waiting for this request to finish. The last request fields are used to make sure that some objects (e.g. `vfsmount`) aren't released before some task is done.

The fuse connection object is another large object. It is implemented in the `fuse_conn` structure and is created when the filesystem is mounted. It has the same life-time as the filesystem and is therefore destroyed when the FUSE device is closed and filesystem unmounted. The first three members implement some locking mechanism. `count` is used the same way as in request object, and *lock* protects from simultaneous access and inconsistencies. Some of the mount information is stored in `user_id`, `group_id` and `flags`. The `waitq` field is used for waking up the reader processes as described in Call-flow subchapter and the next two lists store the requests that are pending and request that are being processed.

If there are too many requests the connection is blocked by setting the `blocked` field and all next request are added to the `blocked_waitq` wait queue. The `reqctr` saves the last request id and is incremented for each new request. The field `connected` is 1 over the life-time of fuse connection object and is cleared only at the end when the filesystem is unmounted, the connection is aborted or the FUSE device is released.

Next come a dozen of flags describing either the state of the connection or configuration. A fuse connection object is identified by the `id` field and stores also the list entry definition in `entry` member. Each fuse connection object stores also a reserved request for the destroy command.

The rest pages of the fuse_i.h file define functions used for obtaining pointers from different structures and declare some global functions that are implemented in C files.

5.3.2 FUSE user-space objects

User-space part of FUSE is implemented in FUSE library. The definitions are stored in include directory. fuse_opt.h file defines two objects used in mount process to parse and store the arguments, I won't go deeper into this one. Neither will I describe the FUSE interface objects in this section, because it will be examined in next subchapter.

The only objects of some relevance are declared in fuse_common.h. The `fuse_file_info` object is used only on create/open operation. This object is stored only in user-space and is only partly used. The most important member is the `fh` and stores a file handle passed from user-space filesystem. This member is also stored in kernel module. The other object defined in this file is `fuse_conn_info` and is used only on init to set some parameters.

As in kernel module, the user-space implementation has also a representation for the request, although it is much smaller as it doesn't have to implement so many control mechanisms. The name of the structure is the same as in kernel, `fuse_req`, but it isn't defined in a header file but in the fuse_lowlevel.c file.

The user-space request implementation stores a reference to a fuse low-level object in the `f` field to maintain a pointer to its parent. It provides also a mutex for locking the request in the `lock` field, although it is used only for the request interrupt function. For security reasons, the request holds context information about the calling process, which is passed from the kernel space. The `ch` member contains the reference to the fuse channel object and `interrupted` is a flag that stores if there is an interrupt command to handle before this request. The next union then stores the interrupt information. Besides these members, the request object stores the request id in `unique` and the reference count in `ctr`.

The fuse_lowlevel.c includes one other private object, the fuse low-level object – `fuse_ll`. It stores some general configuration information (e.g. `debug`, `allow_root`, etc.), the list of requests and interrupts and the FUSE low-level interface. For the user-

space filesystem it stores a reference to the user defined data – `userdata`. To prevent inconsistencies, it provides a mutex named `lock`.

The file fuse_session.c implements two other important objects: the fuse channel object – `fuse_chan`, which is referenced from user-space request and a fuse session object – `fuse_session`. These objects are double linked together.

The fuse session object provides a list of session operations (i.e. `process`, `exit`, `destroy`, etc.), the information about the sessions state (`exit` field) and a pointer to fuse channel object.

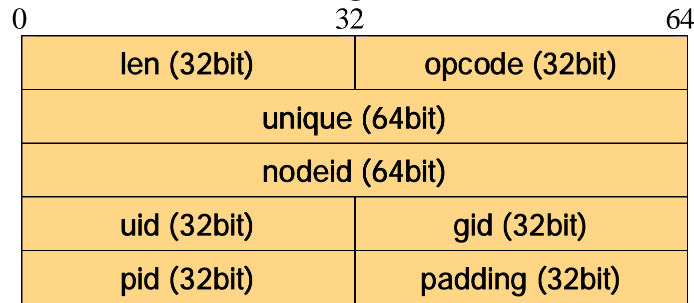
The fuse connection object exports an interface for the communication with kernel-space (`op`) and stores the FUSE device file descriptor in `fd`. It stores also a reference to the session object (`se`) to allow the double link.

5.3.3 FUSE exchange structures

As previously stated there is no simple or direct way to call a user-space function from the kernel-space. The only way is to use some kind of pipe (in form of a filesystem or socket, see subchapter User-space Filesystem) that allows data exchange between these two spaces. A filesystem implementation needs a broad number of functions that have different parameters and return values; therefore it is impossible to implement this with simple data exchange. This requires some kind of interpretation of the data. For performance reasons it is necessary to make multiple »calls« at the same time over the same pipe. The best way to support these requirements is to use a protocol request system.

FUSE has also a communication protocol for its request system. All requests and their arguments have standard structures that are defined in fuse_kernel.h. This file is in the FUSE directory twice. One is in the /include and the other in /kernel directory. Both are and have to be identical because FUSE library (user-space) has to implement the same protocol as the kernel module (kernel-space), otherwise the two processes would not comprehend each other. The request and request answer headers are in figure 10.

a) request header – all fields are unsigned



b) answer header – all fields are unsigned except error

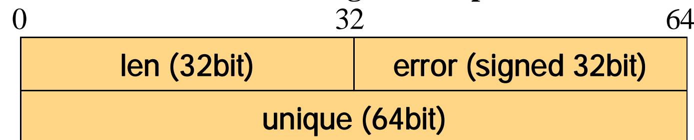


Figure 10: This figure shows an illustration of, a) the request header that is used to send commands and, b) the request answer header that is used to send answer of a request.

The message headers are defined by `fuse_in_header` and `fuse_out_header` structure. All fields of the request header are unsigned values.

- **len** – the size of the whole request
- **opcode** – the code of the command, each method in the FUSE low-level interface has a unique `opcode`
- **unique** – identifier of the request, each request has a unique number
- **nodeid** – id of the inode handling this request
- **uid** – user id of the process executing this request
- **gid** – group id of the process executing this request
- **pid** – process id of the process executing this request
- **padding** – message padding

The answer has smaller header, because the unnecessary fields were removed. All fields, except the `error` field, have unsigned values.

- **len** – size of the whole answer
- **error** – in case some error occurred during the processing of this request, this field contains the error code

- **unique** – identifier of the request that this answer is for.

Requests and answers have various arguments. The size, number and structure of the arguments depend on the `opcode` – the requested command. The specific arguments for all commands are listed in appendix B.

5.4 FUSE Exported Interface

As mentioned in previous subchapters, FUSE implements two abstraction levels and exports therefore two types of interfaces. The FUSE basic interface is defined in the include directory in file fuse.h under `fuse_operations`. All of these functions are optional but some functions have to be implemented for a use-full filesystem. `Open`, `flush`, `release`, `fsync`, `opendir`, `releasedir`, `fsyncdir`, `access`, `create`, `ftruncate`, `fgetattr`, `lock`, `init` and `destroy` are special purpose methods, which aren't essential for a full featured filesystem. Most of this functions work like UNIX filesystem operations, but instead returning the error codes in `errno`, they are returned directly as a negated value (e.g. `-ENOTSUP`).

Low-level interface is very similar to the basic interface. It is defined by the `fuse_lowlevel_ops` structure in the file fuse_lowlevel.h and the main difference is that the functions have an inode (`fuse_ino_t`) instead of the path (`char *`) parameter. Most of the methods receive also a fuse request argument, which must be passed to the reply function.

There is no need to explain the functions because the FUSE interface has a very good in-code documentation. The documentation can be found in the file *fuse_lowlevel.h* (in ./include directory) in the definition of `fuse_lowlevel_ops` structure. Unfortunately in meanwhile such an in-code documentation is the only reference point for a developer using FUSE.

5.5 Usage[15]

Let us now examine the usage of FUSE from user point of view. FUSE may be used on any Linux kernel version 2.6.X where X is greater equal 9. FUSE doesn't typically have precompiled binaries; therefore it is necessary to compile it for the given Linux kernel, although Linux kernel 2.6.14 and later may contain FUSE support. If the Linux kernel contains FUSE support the FUSE kernel module won't be compiled. To override this, use the '--enable-kernel-module' configure option.

Before compiling it may be necessary to add '/usr/local/lib' to '/etc/ld.so.conf' and/or run *ldconfig*. Then run the *configure* command in the FUSE directory. If '*./configure'* cannot find the kernel source or it says the kernel source should be prepared, you may either try

./configure --disable-kernel-module

or if the Linux kernel does not already contain FUSE support:

1. Extract the kernel source to some directory
2. Copy the running kernel's config (usually found in /boot/config-X.Y.Z) to .config at the top of the source tree
3. and run '*make prepare*'

When the *configure* command successfully finishes, run *make*. This compiles the FUSE library, FUSE kernel module, fusermount utility and examples. If you want to install FUSE, run '*make install*'. This copies the binaries into system directories (e.g. *fusermount* into /bin). This installation sets the fusermount user id to root, to allow normal users to mount their filesystems. There are some limitations to prevent security risk.

- The user can only mount on a mount-point, for which it has write permission.
- The mount-point is not a append-only directory which isn't owned by the user (like /tmp usually is).
- No other user (including root) can access the contents of the mounted filesystem.

[15] Installation and usage description can also be found in FUSE directory in README file. For more information on installation see also the file INSTALL

It is possible to set some options regarding mount policy in the file /etc/fuse.conf. Currently these options are:
mount_max = NNN - Set the maximum number of FUSE mounts allowed to non-root users. The default is 1000.
user_allow_other - Allow non-root users to specify the **allow_other** or **allow_root** mount options.

Mounting is done by running the user-space filesystem implementation. For example the example/hello file. The first argument is the mount-point (e.g. *example/hello /mnt/fuse*). Most of the generic mount options are supported (i.e. ro, rw, suid, nosuid, dev, nodev, exec, noexec, atime, noatime, sync async, dirsync); they are described in *mount* manual pages. Filesystems are mounted with the options '-onodev,nosuid' by default, which can only be overridden by a privileged user. The FUSE specific mount options are listed in README file in the FUSE directory.

Recenly the author of FUSE opened an online wikipedia for FUSE that should work as a tutorial for beginners and shows how to build a filesystem in FUSE. This wikipedia can be found here *[fusewiki]*.

6. Braintribe CSP

Content Integration becomes more and more a major part of Enterprise Content Management (ECM). The introduction of Enterprise-Content-Integration (ECI) solutions facilitates the access to content in a heterogenic environment and decreases the integration costs of businesses. **B**rain**t**ribe **C**ontent **S**ervice **P**latform (Braintribe CSP or BTCSP) combines the ECM and ECI technologies to become a unique service oriented ECM platform solution.

The concept of the Braintribe CSP is based on years of experience in document management that Braintribe gathered from the requirements and knowledge of its customers in the field of ECM and ECI technology. The content service platform unites the main aspects of Java and XML (i.e. platform independence, scalability and robustness) with the demands of content integration and the enterprise service bus technology approach. Besides, the service oriented approach, used in Braintribe CSP, conforms to the Service Oriented Architecture (SOA) design. SOA describes a software infrastructure that splits an application into a couple of standalone service modules that are dynamically interconnected to work together in accordance to the business processes.

The key services provided by Braintribe CSP may be divided into four categories: system, content, integration and business services. The system services include essential functions of Braintribe content service platform, which supports the proper working of provided operations. Its functionality includes security, transaction management, logging, caching, etc. Content services focus on the content management - standard ECM functions, such as content analysis, saving, searching or lifecycle and records management. Standard interfaces and connectors to other applications and data repositories offer the integration services. The last service category, business services, embraces all other operations supporting business processes and business requirements of a given industry sector.

Under the right orchestration these services create a new functionality to better fit business processes of customers and industry sectors. More about Braintribe and its products can be found on their homepage *[braintribe]*.

Before describing the Braintribe CSP internals and API, the framework JavaFS, which stands between FUSE and the new BTCP filesystem, is introduced in chapter 7.

7. JavaFS

In order to develop a filesystem for the BTCSP as described in the problem description we decided to use FUSE as it showed to be a very powerful and helpful framework. Instead of implementing the BTCSP filesystem directly to FUSE we chose to create a layer of abstraction. The layer should export the filesystem interface to Java, that's where the name JavaFS comes from. Having this abstraction layer we could then build upon it the BTCSP FS. The figure .. shows where the JavaFS framework is placed in the component model.

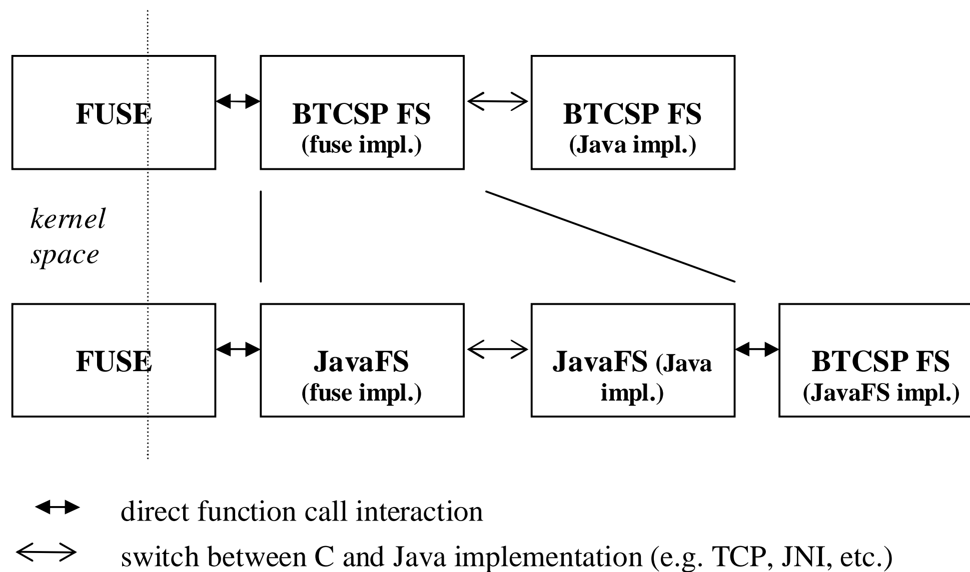


Figure 11: Component model showing how JavaFS was integrated instead of a BTCSP component, which would be directly accessed from FUSE.

JavaFS is an abstraction layer that stands between FUSE and the BTCSP FS, although due to the requirements of the BTCSP FS it was necessary to modify FUSE a little so that it supports larger inode ids (256bit), this improves the performance of the BTCSP FS. As mentioned in section 4.1.1, inodes are typically identified by a 64 bit integer value. From now on, the modified version of FUSE will be referred as FUSEbtcsp.

JavaFS is composed of two parts. One part is the C implementation of the FUSEbtcsp interface and the other is a Java implementation of our API. The two modules are

connected through a TCP connection using own communication protocol, very similar to the FUSE one.

7.1 FUSE Modifications

The FUSEbtcsp added support for 256bit inode ids in FUSE. This could also be done in a different way, but to build-in the support was the most effective solution for the BTCSP FS requirements. The modification involved changes in all objects storing inode id, in function declarations having an inode id parameter and functions handling with inode id. The list of all changes related to the 256bit support in FUSE shows a table in appendix C.

The FUSEbtcsp includes also other adjustments. The JavaFS filesystem implementation has its own directory in FUSE source, which induced subsequent changes in *configure* and *Makefile* files. To support new options (i.e. port and ip address) for JavaFS filesystem, some adjustments were made in the mount option parsing process, see following table.

	Line #	Change	description
Changes in file include/fuse_opt.h:			
+	9:	`#include <arpa/inet.h>`	Includes support for ip address structure *in_addr_t*

+	117:	`int port;`
		`in_addr_t ipaddr;`
		`int debug;`
			New option attributes added into `fuse_args` structure. The fields `port` and `ipaddr` store the JavaFS server connection information and `debug` is flag setting the C part of JavaFS into debug mode.

±	125:	Macro `FUSE_ARG_INIT`	Needed small modification for the support of new options.
±	184:	Changes in the declaration of function that parses options, `fuse_opt_proc_t`	A new parameter was added, passing the context of the parse process in object `fuse_opt_context_t`.
+	188:	Declaration of structure `fuse_opt_context` moved from file	

		fuse_opt.c	
In file include/fuse_opt.c:			
-	16:	Structure `fuse_opt_context` was declared in fuse_opt.h.	
±	129:	The parse context parameter (`ctx`) passed to the parse function.	
±	318:	Couple lines added to support skipping of option arguments and adding them to the argument list.	The context object was used to skip option arguments to avoid handling them as options.
In file lib/fuse_lowlevel.c:			
+	1184:	`KEY_DEBUG_JavaFS,…`	The three new mount options added.
+	1202:	`FUSE_OPT_KEY("-a", KEY_IPADDR),…`	The three new mount options added in the *fuse_opt* structure.
+	1225:	`… fuse_ll_help_javafs(void) {…}`	New static function added for printing out JavaFS mount options.
±	1235:	`… fuse_ll_opt_proc(…) {…}`	Function modified to support new options.

Table 1: Changes made in option parsing part of FUSE implementation. See table 2 for explanation of the symbols used.

The new javafs directory in FUSE source contains the implementation of JavaFS. The C implementation is in topmost files and Java implementation lies in the java subdirectory.

The C implementation has two parts. One is the implementation of the FUSE interface with FUSE deamon. It is situated in file javafs.c. The other part is the set of functions used in communication with JavaFS Java server. They can be found in comm.h, comm.c, conn_list.c and conn_list.h. How FUSE deamon and an implementation of FUSE interface works has been examined in chapter FUSE, therefore we will go into more detail only for the communication part of JavaFS.

7.2 JavaFS client implementation

The JavaFS C and Java parts work together in client-server architecture. The data exchange is achieved over multiple TCP connections, although this data transmission

is not as straightforward as in the kernel-user-space data exchange. C has types that are dependent on the underlying platform; Java and its types are on the other hand platform independent. This demands standardization of some types on the C side. The most involved C types are the numerical types and JavaFS uses only integers what greatly facilitated the problem.

The solution is to translate all integers to network byte order. We chose network byte order because Java types use it and C standard interface already provides the conversion functions (i.e. `htonl`,`ntohs`,etc.). Any integer type exchanged over the TCP connection must be therefore either translated to or from the network byte order; depends on the direction the data is sent.

Another translation takes place in error codes. Naturally Java doesn't know the C error codes and they cannot be hard coded because they depend on the underlying system. The error codes have to be translated in C code and that's what the function

```
int convert_javafs_errno(int javafs_err)
```

is for (conn_list.c); converts for example the `JavaFS_CODE_ENOSYS_ERR` error code into the system `errno ENOSYS`. The `JavaFS_CODE_...._ERR` error codes are then hard coded on both sides of the connection (in MessageHeaderBuf.java and conn_list.h). The reverse conversion is not necessary because error codes are sent only one way.

As mentioned earlier, multiple connections are used to send the requests. The ideal number of connection is when it is same as the number of deamon threads. The connections are managed as resources with exclusive access. The sequence diagram in figure 11 shows, how a connection is allocated if all are currently in use.

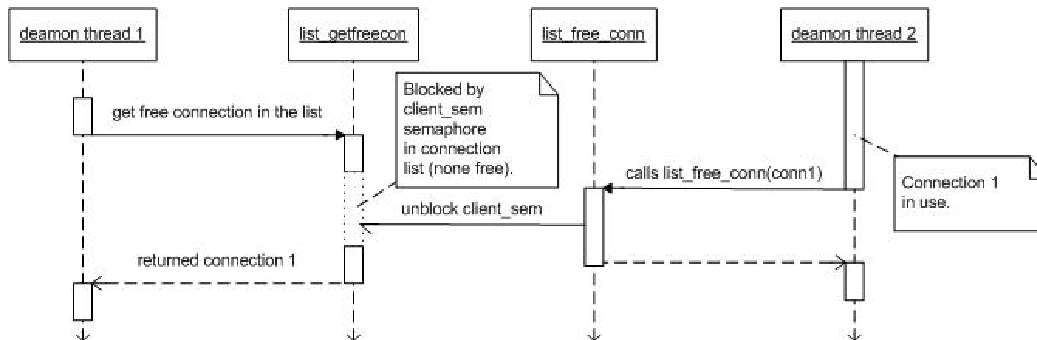


Figure 12: Allocation of a connection if all are currently in use.

Problem illustrated in figure 11 occurs only if there are fewer connections established than the number of running threads in deamon. Then another deamon must wait until a connection is released. In our example the deamon thread 2 uses a connection and deamon thread 1 called the `list_getfreeconn` function to get hold of a free connection. The `list_getfreeconn` locks the list `client_sem` semaphore, which is already locked, and blocks until the thread 2 invokes the `list_freeconn` method that sets the processing bit (`STATUS_PROCESSING_BIT`) of its connection to 0 and unlocks the list `client_sem` semaphore. In that moment thread 1 is woken up, and the `client_sem` semaphore is decreased. It than locks the lists semaphore (`list_sem`) to get exclusive access to the list, and searches the list with the `test_and_set` atomic operation for an unused connection (with processing bit 0). When it finds such connection it unlocks the list semaphore and returns the connection. The next paragraph briefly explains how semaphores work therefore those who already know the semaphores may want to skip it.

Semaphores are IPC instruments used to manage access to exclusive resources. A semaphore contains an integer value that is changed with atomic operations. The basic operations are the *up* and *down*. The *up* operation increments the semaphores value by 1 and *down* decrements it by 1. If the semaphore value is 0 and the *down* operation is invoked, the operation blocks until the semaphores value is more than 0. For example if we want to create a semaphore that will allow limited access to a resource, we may initialize the semaphore to N, and call the *down* operation on each access. This would allow no more than N instances to gain access to the resource.

Semaphores used in JavaFS are accessed with the `locksem` (down) and `unlocksem` (up) method. The `client_sem` semaphore is stored in the list of connections. `client_sem` is incremented when a new connection is added or when a process releases a connection. It is decreased when process gets hold of a connection or when a connection is removed from the list.

Besides `client_sem` a list of connection contains one more semaphore, `list_sem`. This semaphore has never a value greater than 1 because it is used to gain exclusive

access to the list to prevent inconsistencies when a connection is either added or removed. It is also used to prevent another process from removing or adding a connection when the list is parsed.

7.3 Communication Protocol

The protocol used between JavaFS client and server is similar to the FUSE protocol used between kernel-user-space, although it uses own header structures. The request header is defined in file comm.h and illustrated on figure 12. Request answer uses the same message header.

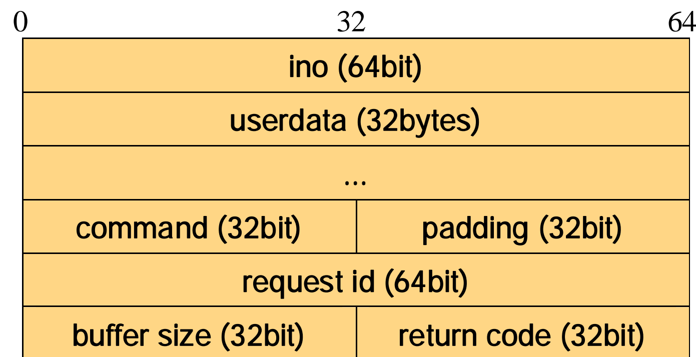


Figure 13: Request header of JavaFS communication protocol.

- **ino –** same as in FUSE protocol, id of the involved inode
- **userdata –** 32 bytes of additional inode information stored from userspace
- **command –** equivalent to `opcode` field in FUSE
- **padding** – 32bit padding
- **request id** – unique request id, equivalent to `unique` field in FUSE
- **buffer size** – number of bytes in buffer
- **return code** – return code of command, 0 on success

After the request header comes typically the command header that is specific for each command. The declarations of command headers are stated in file conn_list.h. For each command header there is also a conversion function that shifts the integers into network byte order as stated earlier.

The headers are of course implemented in Java as well. Their implementation can be found in package `javafsapi.request.structs`. In contrast to C, in Java are no structures and pointers therefore the reading and writing is much more difficult. All of them implement the abstract class `javafsapi.request.IFSBuffer`. The classes are representations of header structures having same members. The members are set by invoking the `Read()` method passing it the byte array of the buffer and the index where the header starts. The `Write()` methods writes the object members into a buffer using the same parameter types as in the read process. Each single member of the header object is read and written using the static class javafsapi.utils.ConvertCTypes. This class provides a list of functions for 16bit, 32bit and 64bit integers, each for signed and unsigned. Java doesn't however support unsigned integers, therefore an unsigned 16bit integer is stored in `int` type (that has 32bit) and a 32bit is stored in `long` type (64bit). An unsigned 64bit integer is also stored in `long`, but the sign bit is chopped off. For example the read of a 16bit unsigned integer is done by the following statement:

```
int u16 = getUnsigned16bitValueAt(index, buff);
```

where `index` is the starting position in byte array `buff`. If there is not enough space to hold the 16bit value on the given position in the buffer, then -1 is returned.

To write the 16bit value back into some byte array buffer, the next statement is used:

```
setUnsigned16bitValueAt(u16, index2, buff2);
```

In case the 16bit value doesn't fit into the buffer at the given position, nothing is done.

7.4 JavaFS server

This section covers the implementation of JavaFS server. It explains the handling of requests and a description of the packages follows. Let's start with an overview of, what the JavaFS is and what it does, to get a good starting point.

The JavaFS sever is a Java application that handles requests from JavaFS client. Just like a web server it has a specific port opened and listens for the incoming TCP connections; therefore the JavaFS is not restricted to run on the same machine as the JavaFS client. If the number of established connections isn't greater than a given value, the incoming connection is accepted and a new worker thread is started that

takes care of the requests that were received over this connection. The used application protocol was already revealed in previous sections. The next figure illustrates the route a request takes to be processed with establishing of the TCP connection.

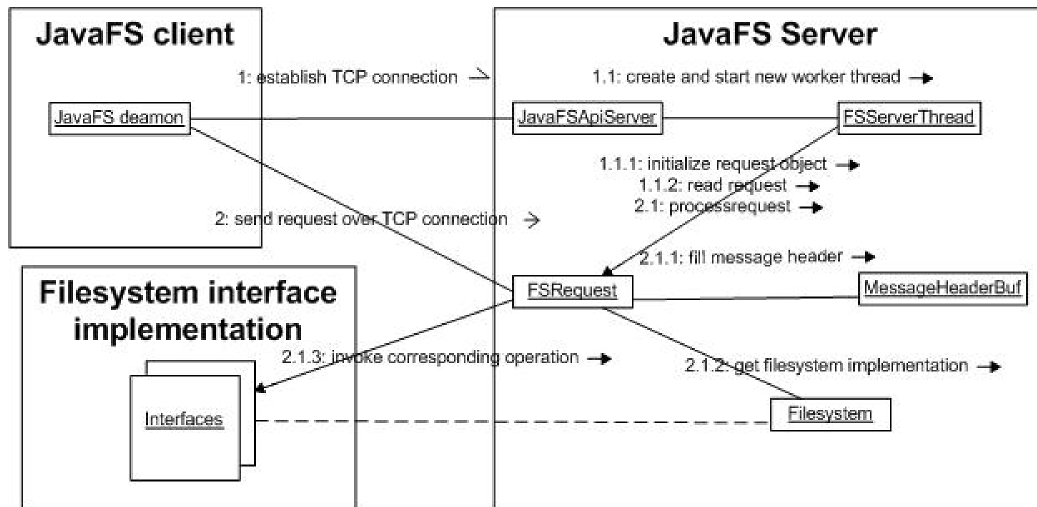


Figure 14: JavaFS server request processing

Requests are sent by JavaFS client and executed by JavaFS server. The `javafs.api.server.JavaFSServerApi` class is just a listener class that waits for connections. Every connection has an own worker object handling the incoming messages - communicating over the connection. The worker class is implemented by `javafs.api.server.FSServerThread` class, which extends the `java.util.Thread` class to run in its own thread and contains the process loop (i.e. read, process and send over and over again).

The read, process and send methods used on a request are included in the `javafs.request.FSRequest` class. The read methods receives a message through a specific TCP socket, fills its internal buffer and sets member values, then reads more data from the socket if the request contains any buffer data. The request header fields are stored in the member `messHdrBuf` (class `MessageHeaderBuf`). If the read function successfully load the received request into the buffer, the worker threads calls the process method in the `FSRequest` object.

At the beginning of the processing the received message is checked whether it conforms the JavaFS protocol and a response object is initialized (initially it is set with an error code). The body of the processing is implemented in a large switch statement with the all possible `command` field as cases. It jumps to the case equal to the command received in request header. Each case prepares the command objects, gets the specific interface from the `javafs.fs.FileSystem` and invokes the command-specific method from the interface. On success, at the end of the case, the response header fields and its buffer are filled. The process function returns after the response is sent.

7.5 JavaFS API

JavaFS is not a filesystem by its own. It just provides framework for a filesystem in Java. The framework is supposed to be a transparent layer between the developed Java filesystem and FUSEbtcsp. To create a filesystem using the JavaFS, the JavaFS API must be implemented, which is pretty much the FUSE low-level interface translated to Java. The JavaFS API is composed of 5 interfaces grouping the filesystem operations by their scope. The corresponding interfaces are listed in package `javafsapi.interfaces`. The function parameters have various types that the JavaFS framework exports to be used as their C language equivalents. The examples package contains a simple »Hello world« example that shows how the API is used in practice. The description of the API interfaces can be found in the following subsections.

Error handling is in all functions same. In case the filesystem wants to generate an error, it should throw an `FSErrorException` exception that contains the corresponding error code and message that will be printed in the console. Errors are also produced outside of the filesystem methods, in case the returned object isn't what the JavaFS framework expected.

7.5.1 IFSAccessDir

As the name of the interface already shows, this interface exposes directory specific functions. However the mkdir command is not present here, it is included in an

interface that contains file and directory operations because it is combined with the create file command.

```
FSInodeObj findEntry(FSInodeObj p_inode, String dentry_name)
      throws FSErrorException;
```

The `findEntry` method is used to look up an entry with the `dentry_name` in the directory given by the `p_inode` argument. In case such entry was found, its inode is returned. If the entry does not exist the method may either throw an exception with error code `ERR_FLAG_ENOENT_ERR` or return null. This method corresponds to the `COMMAND_LOOKUP` command and must be implemented in functioning filesystem.

```
int readDir(FSInodeObj inode, long pos, FSDirEntries dentries)
      throws FSErrorException;
```

This method reads the entries from the directory specified in `inode` parameter. The *pos* parameter gives the position of the entry in the directory where the read should start. The passed `FSDirEntries` object is used to fill a buffer with the entries. It provides the `add` method for this purpose. The `add` method returns 1 if the buffer is full and the `readDir` function may quit, and 0 on success. The `readDir` return value is used when the filesystem uses the JavaFS framework temporary (while the filesystem is running) symbolic links. It specifies the positive difference between the given position and the number of all entries in the directory (e.g. if `pos` is 4 and the directory has 3 entries than the return value is 1). <u>If the filesystem doesn't want to use the framework temporary links, it must always return 0.</u> This method is invoked in the `COMMAND_READDIR` command and should not be missing in a filesystem.

```
boolean openDir(FSInodeObj inode) throws FSErrorException;
boolean releaseDir(FSInodeObj inode) throws FSErrorException;
```

Before a directory is opened or closed these methods are called. The directory is represented by the passed inode object and if there is problem, these methods should return false (or throw an exception). If the filesystem does not want to implement these operations, I suggest to coding it to always return true. Commands `COMMAND_OPENDIR` and `COMMAND_RELEASEDIR` are responsible for invoking these methods.

7.5.2 IFSFileAccess

This interface exports file-specific operations that are listed here.

```
int writeFile(FSInodeObj inode, long pos, long size, byte[] data,
      int data_offset) throws FSErrorException;
```

Writing into file is done here the same way as in FUSE interface. The file is represented by the given `inode`, writing position is passed in the `pos` argument, and the count of bytes to write together with the buffer are sent in the next arguments. This method includes one more parameter that indicates where in the buffer the data starts. This has a simple reason, to avoid multiple copying from byte array to byte array; therefore one byte array is used for sending and receiving data from the TCP socket, and for processing in the filesystem operations. The drawback is that the filesystem implementation may mess-up the response header by writing in the buffer on wrong position. The `writeFile` method is coupled with the `COMMAND_WRITE` command and does not have to be implemented (e.g. in read-only filesystem). In that case it is necessary to have the function to throw the `ERR_FLAG_ENOSYS_ERR` or in a read-only filesystem the `ERR_FLAG_EROFS_ERR` exception. On success the method should return the number of bytes written. The next code snippet shows the read-only implementation.

```
public int writeFile(FSInodeObj inode, long pos, long size,
      byte[] data, int data_offset) throws FSErrorException
{
      throw new FSErrorException("read-only filesystem",
            MessageHeaderBuf.ERR_FLAG_EROFS_ERR);
}
```

```
int readFile(FSInodeObj inode, long pos, long size, byte[] data,
      int data_offset) throws FSErrorException;
```

The `readFile` method does the opposite of the write operation described above. It has the same parameters, whereby the `pos` parameter specifies the position in the files where the read operation should start and `data` is the buffer where the file data should be written. This function must always read the number of bytes denoted by `size`, the exception is only end of file or an error. The number of read bytes is returned.

```
boolean flushFile(FSInodeObj inode) throws FSErrorException;
```

This method is called on each `close` of an open file (even if it is opened with a read-only flag); however there may be more `close` calls than there are `open` calls. This happens when a file descriptor is copied with `dup()` or some other system call. The `COMMAND_FLUSH` command leads to this method.

```
boolean openFile(FSInodeObj inode) throws FSErrorException;
boolean releaseFile(FSInodeObj inode) throws FSErrorException;
```

These are same case as the `openDir/releaseDir` operations for directories. Open is called when a file is opened, and release is invoked when the last `close` is called on the opened file. The file is identified by the `inode` argument. `COMMAND_OPEN` is associated with the `openFile` and `COMMAND_RELEASE` with the `releasFile` method.

7.5.3 IFSAccessFileDir

Some filesystem operations are shared by files and directories and they are grouped in this interface. The method descriptions follow.

```
FSInodeObj Symlink(FSInodeObj parent_inode, String name, String Path)
      throws FSErrorException;
```

This method supports the symbolic link in a filesystem. It used to create a symbolic link inode that will store the path to the linked inode. It receives the directory inode where the new link will be placed in the `parent_directory` argument. The name of the link entry is given by the `name` parameter. On success the created symbolic link inode is returned. This method is connected to the `COMMAND_SYMLINK` command.

```
FSInodeObj Create(FSInodeObj parent_inode, String name, int type,
      long mode) throws FSErrorException;
```

The `Create` operation is shared by the commands `COMMAND_MKDIR` and `COMMAND_MKNOD`. It is used to create new file and directory in the directory given by `parent_inode`. The `type` argument says which inode should be created (e.g. `EntryVarBuf.TYPE_DIR`) and `mode` specifies the initial permissions using the Linux standard (e.g. 0744). On success the newly created inode is returned.

```
boolean Link(FSInodeObj parent_inode, FSInodeObj inode, String name,
      long mode) throws FSErrorException;
```

This method creates a hard link with the given name to the `inode` in the directory denoted by `parent_directory` argument. The `mode` parameter specifies the permissions of the link. It does not create a new inode though it increases the link count of the given inode in the inode attributes object (`FSInodeAttr`). Inode should not be deleted while the link count is greater than zero (see description of the `link` filesystem operation). It returns true on success. The command of this operation is denoted `COMMAND_LINK`.

```
String readLink(FSInodeObj inode) throws FSErrorException;
```

This method is called on a symbolic link inode to retrieve the path stored in it. It is interpreted from the `COMMAND_READLINK` command.

```
boolean Rename(FSInodeObj parent_inode, String name,
      FSInodeObj new_parent_inode, String new_name)
      throws FSErrorException;
```

This operation moves a file or directory with given name in the directory `parent_inode` to `new_name` in the directory `new_parent_inode`. This function does not have to be implemented, however the generic rename function used is very inefficient therefore it is recommended to include it in the filesystem. It returns true on success. Used by the `COMMAND_RENAME` command.

```
boolean Remove(FSInodeObj parent_inode, String name, int type)
throws FSErrorException;
```

This method bonds the `COMMAND_UNLINK` and `COMMAND_RMDIR` commands together. It removes a file or directory with given name from the `parent_inode` directory. The argument `type` specifies the inode type. In case the filesystem supports hard-links, the number of links of the associated inode should be decreased. The inode should be than removed only if the link count hit 0. Return true on success.

7.5.4 IFSAccessInode

This interface implements inode specific operations; for reading and altering of inode attributes.

```
FSInodeAttr GetAttr(FSInodeObj inode) throws FSErrorException;
```

This method implements one of the most important task of a functioning filesystem; in this interface it is the only method that must be implemented. It gets the attributes of an inode in an `FSInodeAttr` object that holds the information about the inode size, creation time, type, number of hard-links, etc. This function is called for each inode that the filesystem is going to work with; it may be sometimes even called a couple of times, if the inode attributes expire. The command `COMMAND_GETATTR` is in charge of this method.

```
FSInodeAttr SetAttr(FSInodeObj inode, FSInodeAttr attr)
      throws FSErrorException;
```

To save modified inode attributes virtual filesystem calls the `SetAttr` method. After the change is applied the new inode attributes are returned. The handler command is `COMMAND_SETATTR`.

```
byte[] GetXattr(FSInodeObj inode, String name)
      throws FSErrorException;
```

This method retrieves the given inode extended attribute for the specified inode. The value of the extended attribute is returned as a byte array. This method is typically called twice. First it finds out the size of the extended attribute value and than it gets the value. The command `COMMAND_GETXATTR` is interpreted as this operation.

```
boolean SetXAttr(FSInodeObj inode, String name, byte buffer[],
      int index, int size) throws FSErrorException;
```

The `SetXAttr` method is used to set the value of the given extended attribute for an inode. The new value is stored in the provided buffer starting at the given offset. The parameter `size` holds the number of bytes that the value is composed of. On success this method returns true. This method is invoked when the `COMMAND_GETXATTR` command is received.

```
String[] ListXattrs(FSInodeObj inode) throws FSErrorException;
```

Extended attributes are user customized properties of the inode therefore the virtual filesystem doesn’t know which extended attributes the inode supports. To get a list of all attributes for a given inode it calls this method that returns the list of extended

attribute names. If there are no extended attributes for the given inode, it should simply return null or an empty string array. The appropriate command is `COMMAND_LISTXATTR`.

```
StatfsBuf GetStatfs(FSInodeObj inode) throws FSErrorException;
```

This operation is used to get inode statistics. The inode statistics are stored in the `StatfsBuf` class which is on success returned. The underlying command is `COMMAND_STATFS`.

7.5.5 IFSAccessFS

The last interface that a filesystem may implement is `IFSAccessFS`. It doesn't include any essential filesystem operations. The three operations are as follows.

```
StatfsBuf GetStatfs() throws FSErrorException;
```

Just like the inode `GetStatfs` this method returns the statistics object and is also invoked from the `COMMAND_STATFS` command. But the returned `StatfsBuf` object contains here the filesystem statistics (i.e. number of blocks used, free and all).

```
void Init() throws FSErrorException;
void Destroy() throws FSErrorException;
```

These methods are called at the beginning (`Init`) and at the end (`Destroy`) of the filesystem usage. Neither of them returns a value. The used commands are `COMMAND_INIT` and `COMMAND_DESTROY`.

7.6 Using JavaFS Framework

JavaFS framework is compiled in a jar file - *JavaFSApi.jar*. Using it is therefore a simple matter; just include the jar file in the classpath. JavaFS framework also allows some configurations, which specify some behavior of the JavaFS framework. The configurations are read from a file named JavaFSAPI.conf, which should be in the same directory as the jar file. If the file is missing, JavaFS API will run with default settings. The configuration file has the following syntax:

```
property = value # comment
```

The next table shows all properties with their default values, which can be set in the configuration file.

Name	default	description
`JavaFS_SERVER_PORT`	4567	TCP port number which will be used by JavaFS server.
`JavaFS_SERVER_MAX_CLIENTS`	100	Maximum number of clients that can be simultaneously connected to the JavaFS server.
`JavaFS_FILESYSTEM_READONLY`	NO	Specifies whether the filesystem should be handled as read-only – all writable operations are ignored.
`JavaFS_MAX_NAME_LENGTH`	252	The maximum length of entry name.
`JavaFS_BLOCK_SIZE`	2048	Files block size. Used in read/write operations.
`JavaFS_ROOT_INODE`	2	Inode id of the root inode (better left unchanged).
`JavaFS_DEBUG`	NO	Run in verbose mode (show debug info).
`JavaFS_SHOW_TIMING`	DEBUG option	Show timing information about execution of commands.

Table 2: JavaFS API configurations. The **`JavaFS_DEBUG`** and **`JavaFS_SHOW_TIMING`** are static options of the Configuration class.

8. BTCSP Internals

As depicted in chapter 6 the BTCSP is a commercial document management system developed by Braintribe for archiving and document management with support for versioning. It has a server and a client component. The BTCSP client provides a user-interface into the database management system. It allows the user to search in the archive database and create, remove and alter the documents and their attributes.

Server runs in background and may serve multiple clients. Its data presentation may be divided into two parts. The variable part is composed of the document data that the user may alter in run-time. This is stored in the underlying database, although the archived files are stored directly on the systems filesystem. The other part is fixed, but may be configured before its first time use. The server configuration is stored in the directory \i2z\server\cfg (in the server application directory). For us are only the content configurations important.

By *content* is to understand a cabinet like structure that may hold multiple documents that share some basic properties. For example a company using BTCSP to manage their documents may want to assign each customer a content that would hold all the documents regarding this customer (e.g. contract, scanned mail received, emails, etc.). Besides the stored documents, each *content* may hold multiple sets of properties which may be later also used in searching. The collection of properties is called content definition and a single property is referred to as index. Our example could have two content definitions:

1. personal_data – containing indexes: title, name, surname, customer id, date of creation, etc.,
2. location_info – would contain: address, city, country, building type, etc...

The basic structure (composed of nodes) that groups the *contents* together and the content definitions with the user customized indexes are configured in two xml files: content_definition.xml and content_hierarchy.xml. Their names already indicate

where the hierarchy is and where the content definitions are stored. Figure 14 shows the overall hierarchy of the BTCSP system.

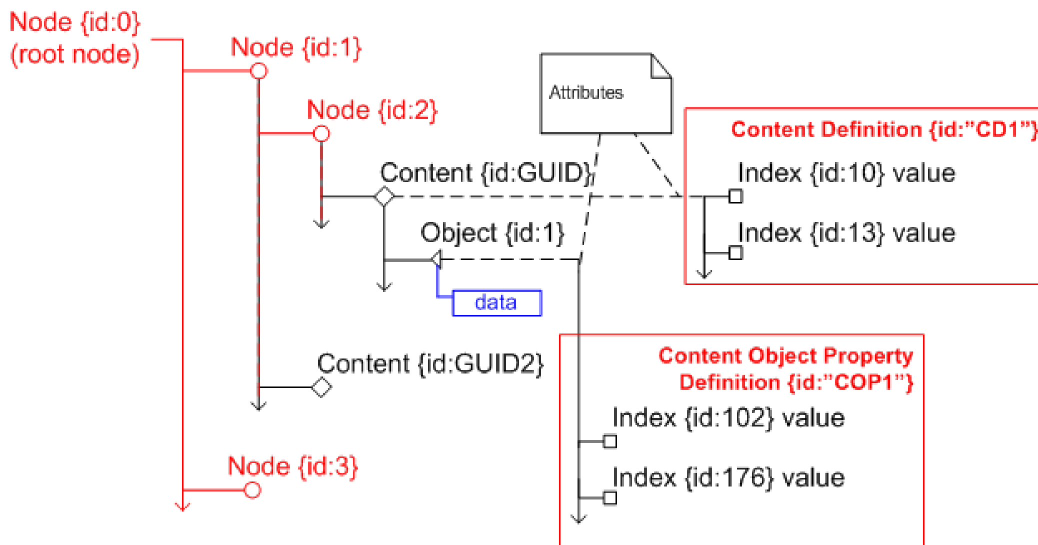


Figure 15: BTCSP data hierarchy.

The figure shows filesystem-like hierarchy. The red part is preconfigured and can't be changed at run-time, black are stored in the database and blue part is accessed on the system filesystem. The four types of components are clearly visible each being denoted by a special symbol. The types are listed here:

- **Node** – represented by a small circle is part of the preconfigured hierarchy. It defines the content definitions that will be used with the direct *content* children. It may have sub-nodes which define new content definitions. Each node is identified by a positive integer; the root node having typically id 0.
- **Content –** is a virtual directory in the figure symbolized with a diamond. It must always have a node as a parent that defines the content definitions bound to this *content* (and all *content* siblings). Content then declares the values for the *indexes* in those content definitions. Content may be added or removed and have its indexes modified at run-time, and each *content* is identified by its GUID.
- **Content object (or object) –** having in figure the triangle symbol, is the database system representation of the archived document. It is always a child of a *content* identified by an object id, which is unique only in the scope of the

content. It is partially stored in the database and partially on the system filesystem. The database holds the path to the file and the values of some basic attributes that may be supplemented with additional information as defined by its grandparent node.

- **Index –** is symbolized by a small square. Each *index* expresses a single attribute for either a *content* or an *object*. It defines the name and the type of an attribute value (i.e. String, Date, etc.). The possible indexes of *contents* and *objects* are defined in nodes (in preconfigured part), but the values are stored in database with the specific *content* or *object*. Indexes are identified and accessed by unsigned integers. They are grouped in content definitions.

BTCSP server supports multiple databases and operating systems therefore it provides an API to access the data. The client API is implemented in Java (like the server) and some essential parts are addressed in the following subchapter.

8.1 API

The BTCSP client API is very useful in that it allows programmable access to the data. Instead of accessing the database and the documents on the filesystem directly, this API allows to use the BTCSP server functionality. The Java API has two abstraction levels, but we will cover here only the low-level API, which we used in the BTCSP filesystem implementation. However both levels initialize the session in same way:

```
theFactory = I2zFactory.getI2zFactory(clientCommFile, Log4jFile);
i2zSession = theFactory.getSession(serverName,userName,password);
```

where clientCommFile indicates the path to the file that configures the communication with the BTCSP server. The Log4jFile parameter holds the path to the logging configuration file. When the session is created, the user may choose the preferred API level.

The basic low-level API class used to communicate with BTCSP server is `COMM_Module` in package `biz.i2z.comm`. This class manages the communication channel between the client and BTCSP server. In order to run procedures on the

server the `COMM_CallInterface` class is used. To get a reference to this class the `get_COMM_Call()` method of the communication module must be invoked.

The call interface class allows calling procedures remotely on the server. The authentication of the client using the low-level API would then look like this:

```
COMM_Parameter[] authParms = new COMM_Parameter[2];
authParms[0] = new COMM_Parameter(COMM_Protocol.STRING, "i2z");
authParms[1] =
        new COMM_Parameter(COMM_Protocol.STRING, "operating");
COMM_Parameter retSec = call.callMethod("authenticateUser",
        authParms, null,  "SECURITY");
```

A remote call is made using the `callMethod()` method, but before the call the procedure parameters must be prepared. The `COMM_Parameter` class is a variant type (something like the Variant type in Visual Basic or VARIANT in C++). It stores the type (i.e. String, integer, etc.) and the specific value; only simple types are supported. To pass the arguments to the method an array of parameter classes is made and the reference is passed to the `callMethod` method, which first argument is the name of the procedure that should be invoked. The last argument specifies the module that the client wishes to access. In our example it is the *SECURITY* module, though the *RETRIEVAL* module is also very common.

A parameter class is also returned, typically having a string type. The communication exchange is based on XML therefore most of the string values, especially retrieval results, have an XML form which depends on the procedure invoked. Here is a list of the most common procedures:

- **getContentHierarchy, getContentDefinitions** – these procedures are used to get the data structure configuration. Basically they return the contents of the content definition and content hierarchy files, but with a slightly different syntax. The result of these procedures is always static (does not change in run-time) therefore it is useful to save it in a cache. These procedures don't have any parameters – the parameter array is null. Part of *RETRIEVAL* module.
- **authenticateUser –** This procedure was already shown in the previous example. It authenticates a user for this session and must be therefore invoked

before any other remote call. It takes two string parameters: user name and password. To access this procedure one must use the *SECURITY* module.

- **fastSearchISAndGetResults** – Looks up *contents* with specified attributes and returns the list in an XML string. Accessed through *RETRIEVAL* module taking 6 parameters:
    1. **Boolean** saying whether this search should include subnodes.
    2. **int[]** an array of integers specifying the nodes that will be searched.
    3. **String** defining the search attributes. Has the following XML form: _"<i2z><Index id=\"4001\" op=\"=\" expr=\"*\"/>…</i2z>"_ where the various attributes may be grouped with <*and*> or <*or*> tags.
    4. **int[]** integer array holding the index ids whose values will be included in the result.
    5. **Integer** constraining the results to maximum count of *contents*.
    6. **Integer** specifying sts.
- **getSearchId, openISSearch, isSearchResultsAvailable, getResults, closeSearch** – does all together the same as the procedure above, however allowing to split the results in chunks when the result string includes too many *contents* that would exceed the maximum size. The `getSearchId` procedure doesn't have any parameters and returns the search id as string used in the other four search procedures; it opens a kind of search session. Actual search constrains are specified calling the `openISSearch` procedure. It takes four arguments, the search id as first argument, and the second, third and fourth parameters are the same as in `fastSearchISAndGetResults` method. The `isSearchResultsAvailable` function may be then invoked, with the search id as the single argument, to check the status of the search, to see whether there are any results available. `getResults` is then used to get the results of the given search. The number of contents to get in the result string is defined by the second parameter. This call may be called several times, returning always results that haven't yet been returned. The final `closeSearch` call closes the search session. These procedures are invoked in the *RETRIEVAL* module.

- **getContentObjectsProperties** – Retrieves the *objects* of a given *content.* Here is the declaration of this function in pseudo code:

```
string getContentObjectsProperties(string content_definition,
      string content_GUID, int version);
```

First parameter specifies the content definition id of the content given by *content_GUID*. The *version* argument doesn't have to be used. It constrains the results only to the given object versions (use -1 to get all versions).

- **getContentData** – Obtains index values for a given *content*.

We didn't show yet how a result XML looks like. Here is a list of typical XML formatted results that are returned by BTCSP server. The XML results depicted in this table are only simple examples highlighting the essentials of the syntax.

Retrieved content definition

```
<?xml version="1.0" encoding="UTF-8"?>
<i2z>
 <ContentDefinitions>
  <ContentDefinition id="CD3" name="Exekutionen" description="">
   <IndexStore store="IND_STORE">
    <Index id="3">
     <Parameters>
      <Parameter name="length" value="100" />
      <Parameter name="dbLength" value="100" />
     </Parameters>
    </Index>
    <Index id="11" />
    <Index id="12" />
    </Index>
   </IndexStore>
   <ObjectStore name="OBJ_STORE">
    <ContentObjectsPropertyDefinition name="COP">
     <ContentObjects>
      <IndexDefinition />
     </ContentObjects>
     <ContentObject>
      <IndexDefinition>
       <Index id="2002" name="originalFileExtension" type="string">
        <Parameters>
         <Parameter name="length" value="50" />
         <Parameter name="dbLength" value="50" />
        </Parameters>
        <TypeInfo>
         <Length from="0" to="2147483647" />
        </TypeInfo>
       </Index>
       <Index id="2001" name="originalFileName" type="string">
        <Parameters>
         <Parameter name="length" value="255" />
         <Parameter name="dbLength" value="255" />
        </Parameters>
        <TypeInfo>
```

	<**Length** from="0" to="2147483647" /> </**TypeInfo**> </**Index**> </**IndexDefinition** </**ContentObject**> <**ObjectPart**> <**IndexDefinition** /> </**ObjectPart**> </**ContentObjectsPropertyDefinition**> </**ObjectStore**> </**ContentDefinition**> <**ContentDefinition** id="CD2" name="Pensionsakt" description=""> … </**ContentDefinition**> </**ContentDefinitions**> <**IndexDefinition**> <**Index** id="12" name="SAP-Schluessel" type="integer"> <**TypeInfo**> <**Length** from="0" to="2147483647" /> </**TypeInfo**> </**Index**> <**Index** id="11" name="Fallnummer" type="integer"> <**TypeInfo**> <**Length** from="0" to="2147483647" /> </**TypeInfo**> </**Index**> … </**IndexDefinition**> </**i2z**>
Retrieved content hierarchy	<?xml version="1.0" encoding="UTF-8"?> <**ContentHierarchy**> <**node** id="0" name="DMS" access="AddTextComments, AddTwain, DeactivateContent, EditAnnotation, … " description="Knoten Beschreibung" guiVisible="Yes"> <**node** id="1" name="Personendaten" access="AddTextComments, AddTwain, DeactivateContent, EditAnnotation, …" description="" guiVisible="Yes"> <**ContentDefinitions**> <**ContentDefinition** name="CD1" /> </**ContentDefinitions**> </**node**> <**node** id="2" name="Pensionsakt" access="AddTextComments, AddTwain, DeactivateContent, EditAnnotation, …" description="" guiVisible="Yes"> <**ContentDefinitions**> <**ContentDefinition** name="CD2" /> </**ContentDefinitions**> </**node**> <**node** id="3" name="Exekutionen" access="AddTextComments, AddTwain, DeactivateContent, …" description="" guiVisible="Yes"> <**ContentDefinitions**> <**ContentDefinition** name="CD3" /> </**ContentDefinitions**> </**node**> </**node**> </**ContentHierarchy**>
Search results	<?xml version="1.0" encoding="UTF-8"?> <**i2z** searchId="070623151032019e6a6c314e333d1545" displayName="BEDNR = *"> <**HitList** objectstore="OBJ_STORE" indexstore="IND_STORE"

	contentdefinition="CD2"> <**Nodes**> <**Node** id="2" /> </**Nodes**> <**Entry** contentId="070623135505831a1a618e7bd054c6aa" status="0" score="1" dts="23.06.2007 13:55:38" objectstore="OBJ_STORE" miid="0"> <**Value** id="1" value="" /> <**Value** id="2" value="test content 2" /> <**Value** id="3" value="" /> </**Entry**> </**HitList**> <**HitList** objectstore="OBJ_STORE" indexstore="IND_STORE" contentdefinition="CD2"> <**Nodes**> <**Node** id="2" /> </**Nodes**> <**Entry** contentId="070619114224327aa39c61701fd42c98" status="0" score="1" dts="19.06.2007 11:42:55" objectstore="OBJ_STORE" miid="0"> <**Value** id="1" value="" /> <**Value** id="2" value="test_nofile" /> <**Value** id="3" value="" /> </**Entry**> </**HitList**> … </**i2z**>
Content object list	<?xml version="1.0" encoding="UTF-8"?> <**ContentProperties**> <**Parameters**> <**Parameter** name="contentPath" value="0706/2313/51/" /> <**Parameter** name="filename" value="070623135158552b00c86d6fd8d4420a.1" /> <**Parameter** name="port" value="9011" /> <**Parameter** name="host" value="192.168.190.1" /> <**Parameter** name="prefix" value="smi://i2z_master//C//braintribe/Server-1.2.15/i2z/server/JBX/" /> <**Parameter** name="path" value="0706/2313/51/" /> <**Parameter** name="server" value="i2z_master" /> <**Parameter** name="contentId" value="070623135158552b00c86d6fd8d4420a" /> </**Parameters**> <**ContentObjects**> <**ContentObject** id="1"> <**Parameters**> <**Parameter** name="filename" value="070623135158552b00c86d6fd8d4420a.1.1" /> <**Parameter** name="objectPath" value="0706/2313/51//" /> <**Parameter** name="originalFileName" value="desktop.ini" /> <**Parameter** name="prefix" value="smi://i2z_master//C//braintribe/Server-1.2.15/i2z/server/JBX/" /> <**Parameter** name="sts" value="0" /> <**Parameter** name="path" value="0706/2313/51//" /> <**Parameter** name="originalFileExtension" value=".ini" /> </**Parameters**> </**ContentObject**> … </**ContentObjects**> </**ContentProperties**>
Content data	<?xml version="1.0" encoding="UTF-8"?> <**IndexValues** contentDefinitionName="CD2" indexStoreName="IND_STORE" dts="19.06.2007 11:42:55" version="1"

	miid="0"> <**Index** id="1"> <**Values** nodeId="2" /> </**Index**> <**Index** id="2"> <**Values** nodeId="2"> <**Value** val="test_nofile" /> </**Values**> </**Index**> <**Index** id="3"> <**Values** nodeId="2"> <**Value** val="" /> </**Values**> </**Index**> … </**IndexValues**>

Table 3: Result examples of five most common operations.

Archived document data are accessed differently. Instead of using a call interface the class `FileAccess` in package `biz.i2z.system.utils` is used. The file is then accessed as it would be a local file using I/O streams. A code snippet is showed here:

```
fa = new FileAccess(filepath);
InputStream in = fa.getFileInput();
in.read(buf, off, len);
fa.close();
```

Of course the BTCSP client API supports dozen other procedures and modules. This subchapter handled only a fraction of the available methods. To get a bigger picture see the client API introduction [capii].

8.2 BTCSP FS

BTCSP filesystem (BTCSP FS) provides another way to access BTCSP archived documents. It binds the FUSEbtcsp and BTCSP API together. The figure 15 illustrates how the different APIs are connected.

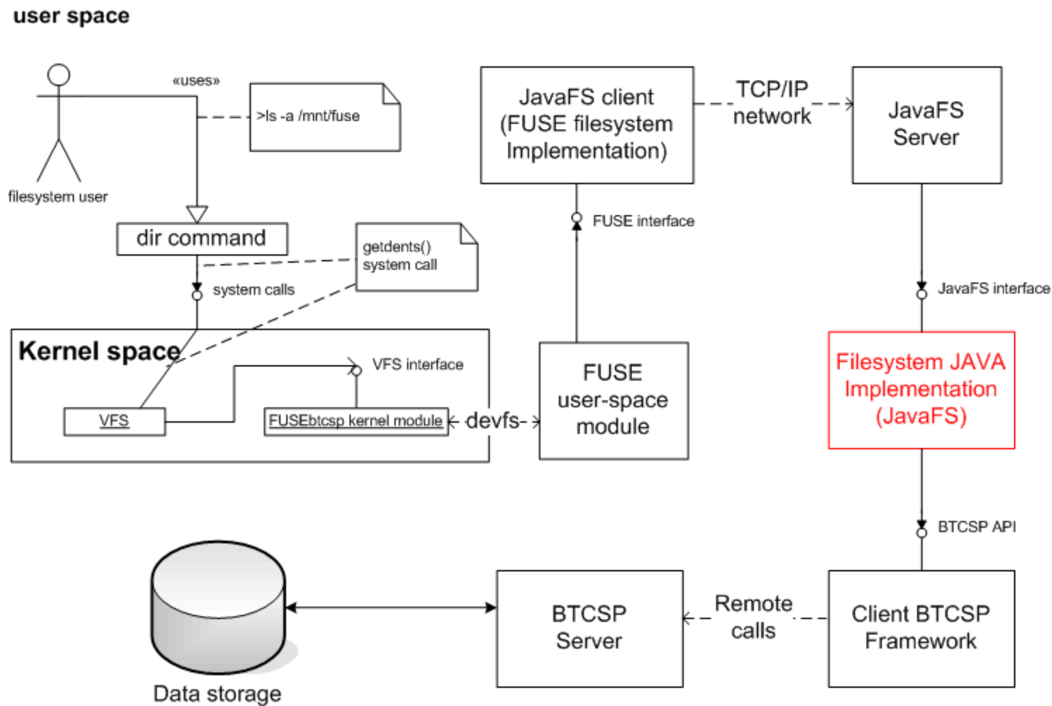


Figure 16: Integration of the BTCSP filesystem (in red).

Commands originate on the Linux machine in user-mode. System calls interact with the virtual filesystem that delegates the commands to the FUSEbtcsp filesystem implementation. FUSEbtcsp translates the commands to requests and tunnels them back to user-mode JavaFS implementation. The requests are transcribed into JavaFS protocol that sends them over TCP to the Java implementation of JavaFS server. Here joins BTCSP FS the command stack by implementing the JavaFS API. It interprets the commands and makes necessary remote calls over the BTCSP client API. The result traverses then again back to the command origin through all the API layers. Although some commands may reach only some of the layers if an error interrupts the command flow.

The BTCSP FS has a relatively straightforward implementation (most of the work is done by the under- and above-lying frameworks) therefore we will describe only the functionality – how the filesystem is generated from the BTCSP data. The basic hierarchy is already defined (as illustrated in figure 14), but the naming of the files and a deeper structure are still open; besides, inode identification can't be directly carried over (i.e. content GUID).

The naming is not simple if the BTCSP server is not configured for this use (i.e. no unique names for filesystem use defined). Names of all item types of the BTCSP structure are identified by integers or GUID not by a string therefore it is overall possible to use same names (name attribute) for multiple entries with same parent (i.e. siblings having same name), although this would be rather accidental than intentional from the user. We solved this by allowing different possible naming conventions (e.g. use name and id, only name, or use chosen attributes).

The tree structure of the BTCSP FS isn't also restricted to be used in the filesystem. The BTCSP FS may organize the data into any hierarchical structure, although it should have some system to keep track of the entries. To help keep track and organize the data the FUSEbtcsp stores 32 bytes user-data with each inode. The BTCSP FS uses this buffer as illustrated in next figure.

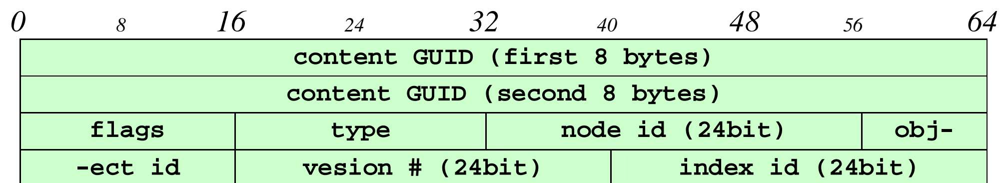


Figure 17: BTCSP filesystem 32 byte user-data usage.

- **content GUID** – identifies the parent content in the BTCSP hierarchy. If the inode is a node, this is filled with zeros. (This doesn't hold the string representation of GUID! – this would be a waste of space – instead it is the 128bit representation of a GUID)
- **flags** – some flags that may be stored with an inode (this is the only field that is excluded from hash generation, later in this section)
- **type** – another 16bit used to store some info about the inode. It differs from the *flags* field in that this field is used in the hash algorithm.
- **node id** – Id of the parent (sub)node in the BTCSP hierarchy. Should never have the value 0. It doesn't need more than 24 bit because as we already mentioned the node structure is preconfigured in a file and contains rarely more than hundred nodes.

- **object id –** id of the object in a content. With the content GUID this makes a unique identification of each object (in case versioning is turned off). One content doesn't typically hold many objects.
- **version number –** holds the version number of the object.
- **index id –** identifies attribute in a content or object to support file representation of attributes.

As described above, this 32 byte user-data is more than enough to identify all BTCSP items; besides identification it holds also the BTCSP hierarchical structure of each inode. But as you might already suggest, an inode in a filesystem is identified by a 64bit integer and we have here 256bit. The easiest way would be to assign each inode an id in order they have been accessed and hold this in an array until the inode is removed from the virtual filesystem inode cache. The inode cache would then hold several thousand inodes. The inode user-data mapping would then be used to translate all inodes in a readdir or lookup operation to the inode id - integer value. This would have a negative affect on the lookup and readdir performance, but every inode would have definitely a unique integer id, although FUSEbtcsp and JavaFS would then need to include another command (`PUT_INODE`) into their protocol to capture the event when an inode is removed from the VFS inode cache. This would also cause problems when restarting server while a filesystem is in use. The server side would loose the inode cache what could create inconsistencies when new inode id would be assigned.

To avoid the countless lookups in the association array and to avoid the modification of the protocol another possibility was examined and implemented. Instead of assigning each inode a unique integer on access the inode ids are generated from the user-data. The process of generating inode ids from user-data must:

- assign each inode a unique id,
- an inode should always have the same integer assigned.

The second premise doesn't imply any problems, but the first premise constraint is impossible to hold (i.e. mapping all 256bit integers to unique 64bit integers). But we don't need all the inodes to have unique id, it suffices an inode has a pseudo unique id; by pseudo unique is meant that the id is unique in a group of ids. The VFS works

always only with a dozen inodes and its inode cache holds only a small group of all inodes in the filesystem. The mapping algorithm should therefore assign each inode in this group a unique id.

The best solution is to randomly spread the 64bit unique ids in the 256bit space. Then we will get a probability P_c that in a group of C elements at least two will have same id.

$$P_c = \frac{\prod_{i=0}^{c-1} (2^{63} - i)}{(2^{63})^c}$$

That is, if a group (cache) has $C = 1.000.000$ inodes, then probability $P_c \approx 7{,}14 \times 10^{-8}$; this is less than being struck by a lightning or being involved in an airplane accident. If two inodes have same id it is still possible to find out that they have different user-data and this rare occasion can be handled.

The BTCSP filesystem uses message digest (MD5) to hash the inode user-data; except the flags field. The hashing is implemented in BTCSP inode implementation and is called only when necessary, i.e. when the inode id is requested, to spare CPU time. How is the BTCSP filesystem overall performance is described in chapter 10.

8.2.1 Usage

Just like JavaFS API, BTCSP filesystem is compiled into a jar file - *BTCSP_FS.jar*, however BTCSP filesystem has additional external dependencies related to the BTCSP client API. These dependencies are not included in the BTCSP_FS package; therefore these need to be added into the class path ass well. This involves the: *clientApi.jar*, *jdom.jar*, *log4j-1.2.8.jar* and *xalan.jar* packages.

BTCSP filesystem implements JavaFS API interfaces and is responsible for implementing the main class. The main class is implemented in the *BTCSP.BTCSP_Main.class.* The resulting BTCSP_FS.jar file may be then executed in

a simple *java -jar* command, although it needs all the necessary packages to be in the .\lib directory (relative to the *BTCSP_FS.jar* file). This isn't very reasonable, but there is no other way to do it (besides including all dependencies in the BTCSP_FS jar file).[16] The source of the manifest file is in the META-INF directory.

The main class doesn't take any parameters. Similar to JavaFS API, all configurations are read from a configuration file; from the *BTCSP_FSS.conf*. If the file is not found, the application takes hard-coded default values implemented in the class *BTCSP.config.BTCSP_Configuration*. The syntax of the BTCSP filesystem configuration file is the same as before. The following table shows the possible parameters with their default values and description.

Name	**default**	**description**
`BTCSP_ClientCommPath`	*client_comm.xml* in local directory	Path to the BTCSP client comm xml file.
`BTCSP_Log4jPath`	*log4j.xml* in local directory	Path to the BTCSP log4 xml file.
`BTCSP_USE_INDEX_NAME_IN_CONTENT_NAMELIST`	NO	Specifies whether the name indices are defined using index names. (e.g. if set to "YES", instead writing CD1 = 2, one would write CD1 = Name)
`BTCSP_CONTENT_NAMES`	Null	Content definitions, for which one wants to define naming indexes. (e.g. … = CD1, CD2)
`BTCSP_USE_INDEX_NAME_IN_CONTENT_AUTO`	NO	Specifies whether the name indices should be generated automatically – from all obligatory indices of a content definition.
`BTCSP_NAME_CONTENT_SHOW_ID`	NO	Show content GUID in the name of a content directory.
`BTCSP_NAME_NODE_SHOW_ID`	NO	Show node id in the name of a node directory.
`BTCSP_NAME_OBJECT_SHOW_ID`	NO	Show object id in the name of an object file.
`BTCSP_NAME_OBJECT_HIDE_VERSION`	NO	Hide version in the name of object file.
`BTCSP_NAME_INDEX_SHOW_ID`	NO	Show index id in the name of an index file.
`BTCSP_DEFAULT_NAME_INDEX_CONTENT`	2	Index to use for generating a content directory name in case not defined and auto naming is

[16] When running jar files, the classpath option is ignored. Instead the Class-path parameter is taken from the jar file. Therefore all external dependencies have to be in set in the manifest file of the jar file. See [runjar].

		disabled.
BTCSP_CONTENT_SHOW_INDEXFILE	NO	Show file with values of all content indices in the content directory. Also called index-file.
BTCSP_CONTENT_INDEXFILE_NAME	index	Name of the indexfile.
BTCSP_CONTENT_MODIFICATION_ENABLE BTCSP_CONTENT_DELETE_ENABLE BTCSP_CONTENT_CREATE_ENABLE	YES	Permissions for content changing (index values), deleting and creating.
BTCSP_OBJECT_WRITE_ENABLE BTCSP_OBJECT_DELETE_ENABLE BTCSP_OBJECT_CREATE_ENABLE	YES	Permissions for object writing, deleting and creating.

Table 4: BTCSP filesystem configuration parameters.

Before closing this section it would be handy to explain the naming of the files and directories in the BTCSP filesystem. Naming of objects (files) and nodes (directories) wasn't as much effort. Objects have their original filename that is used to generate their names. Depending on the configurations object name may also contain object id or version number. Nodes have a name property that is used in their naming. If necessary, node naming may be configured to include node id in their names.

The content naming wasn't that straight forward. Contents don't have to have any name property. The content properties depend on the parent node content definition. For this reason it was practical to give the user options how to generate the content names. One option is, as in object and node naming, to include content id (GUID) in names, which would certainly generate unique names, but would not be very user-friendly. To include some useful information about the content in the content name without user guidance, an option enables automatically generated names. These names are generated from the most important indices in the content definitions, which are typically the ones that have to be set in the content creation process – the obligatory indices. This generates user-friendly names, which however might become unnecessarily long. Another configuration gives therefore the user the freedom to select the indices he or she values the most.

Besides generating content names is the naming configuration also used in initialization of content index values when a content directory is created (using the *mkdir* command). The user enters a dot separated content name that is parsed according to the name indices to set the content index values. In case an index value

should include a dot, this should be preceded by the '%' escape character. Ids (version number) should not be included in the creation commands, because at this time they are typically unknown. The entry is internally renamed to include the ids (version number) if configured to show the ids or other automatic numbers.

The next pictures show how BTCSP filesystem is used in an explorer and terminal.

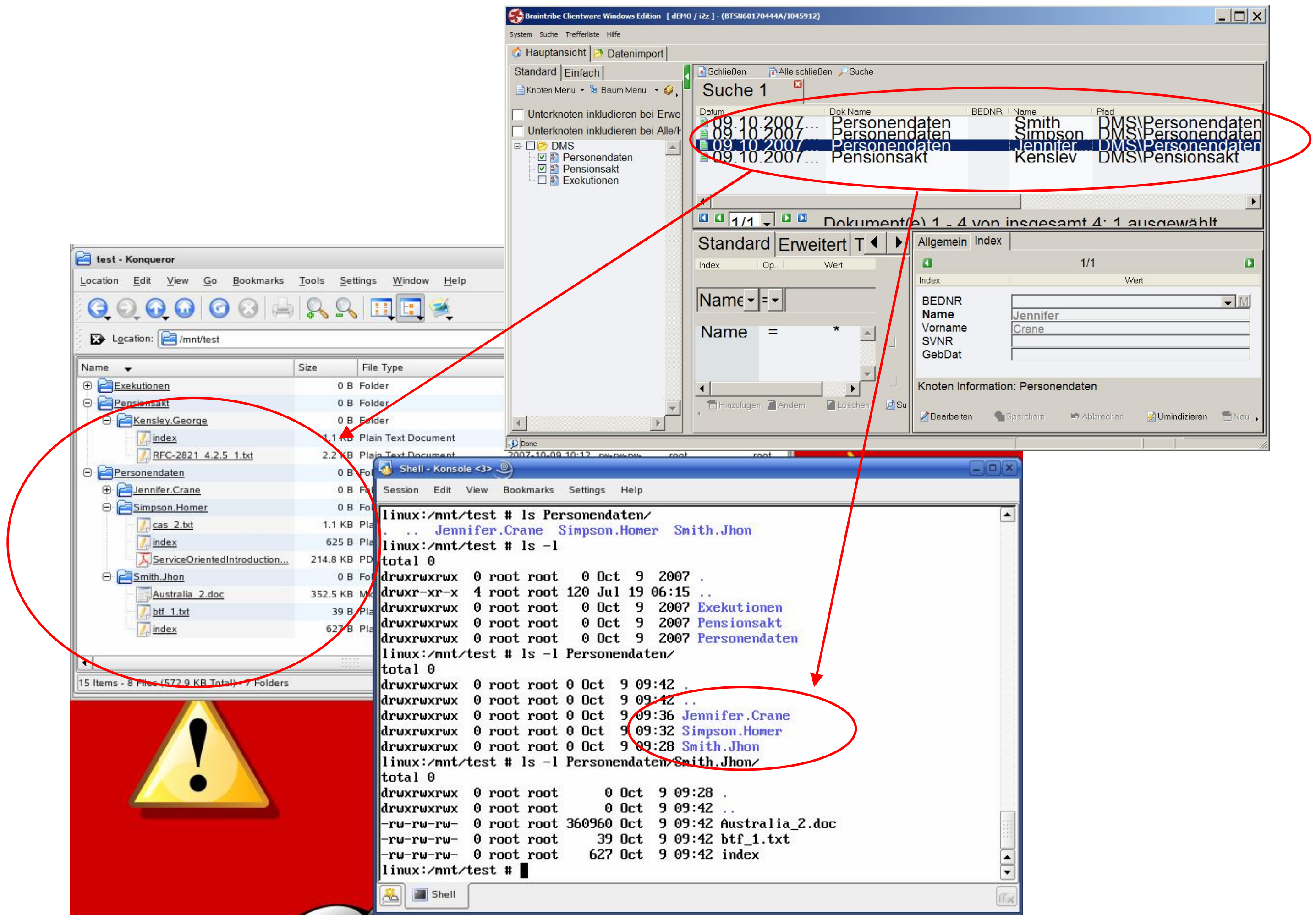


Figure 18: Screenshots from the BTCSP client application (top right), and from the Linux Konqueror (down left) and terminal (down right) using the mounted BTCSP filesystem (mount point is */mnt/test* directory). The BTCSP application shows the same contents as the Linux applications. The red ellipses highlight the contents.

9. Related Work

As already mentioned, there are lots of user-space filesystems. Many of them running on FUSE but also many developed from scratch. Some of the FUSE based filesystem are listed here:

- **SSHFS** – a filesystem that provides access to files over an SSH connection.
- **EncFS** – an encrypted filesystem
- **RelFS** – filesystem storing files and directories in a relational database

The list goes on as FUSE is an open-source application under GNU General Public License (GPL) license – allowing the use of the code for free (with some conditions stated in GNU GPL). Of course not all the user-space filesystems are based on FUSE. For example:

- **V9FS** – a user-space filesystem implementation of the 9P distributed file protocol (also called styx) [plan9]
- **DAVFS2** – user-space filesystem providing access to WebDAV server. It is based on Coda [coda], a networked filesystem, and neon, a WebDAV interface.

These are also GPL filesystems. Examples of commercial (non GPL) filesystems are harder to find because it is harder to judge if a commercial filesystem is user-space or not, but an example would be the **Callback File System™** [cfm] for Microsoft Windows. It is also a user-space framework like FUSE, but for Microsoft Windows OS.

As in our case, many times filesystems are not only implemented in user-space but also in a different programming language. Therefore there are some open-source projects based on FUSE allowing implementation in various languages. Some of the language bindings for FUSE:

- **C** – this is the native FUSE API
- **C++** - FUSECPP
- **Java** – FUSE-j
- **Haskell** – hfuse. Haskell is a purely functional programming language

- **Python, Perl, C# -** some other languages[17]

For us is FUSE-j of relevance [fusej]. It is a Java API that uses JNI bindings to FUSE library. The last version released was in 6/2007 for the FUSE 2.4 version, what was also one of the reasons to make JavaFS API. There is also a project called FuseWrapper [fw] that allows Java and other high-level languages for FUSE. It is a SWIG[18] based wrapper that binds the FUSE low-level library with Java. This would not work with FUSEbtcsp because of the modified low-level API.

[17] The whole list can be found on website: http://fuse.sourceforge.net/wiki/index.php/LanguageBindings
[18] Simplified Wrapper and Interface Generator (SWIG) is an open-source tool used to bind C/C++ program libraries with other script or high-level languages [swig].

10. Evaluation and Further Work

At the end of the work some test were run to show some missing issues and hidden problems. The obvious and important problems were improved in the test process. All outstanding problems and issues are described in the two following sections.

10.1 Evaluation

The operational test of the BTCSP filesystem didn't show any functional problems. The next table shows the average times in milliseconds a JavaFSApi command is executed in the Java environment. JavaFSApi commands that are not listed have negligible times or are called only relatively few times to be of any importance.

Command	time (ms)	command	time (ms)	command	time (ms)
LOOKUP	37	READDIR	46	READ	1,48
GETATTR	18	WRITE	0,16	GETXATTR	12

Table 5: Average times a JavaFS command needs to execute.

The writing and reading of a file doesn't make many overhead operations, therefore the *cp* command shows a good estimate of the framework speed (including network transmission time). In test we used the *cp* command to write from and into the filesystem. In this I/O test one large file (~10Mb) was copied into the BTCSP filesystem and then copied from the BTCSP filesystem to the local disk. The resulting speed in this test was 1272 kbps for writing (*cp /local/tesfile /mnt/fuse/Node/Content*) and 2089 kbps for reading (*cp /mnt/fuse/Node/Content/tesfile /local/.*)[19].

Besides IO operations, readdir (*ls -l* command) operations have a great significance in a filesystem. The tests were therefore run also on the *ls* command. To the readdir operations belong the LOOKUP, GETATTR and READDIR. However the Linux

[19] Speed is in kilobyte per second units. Converted to kilobits per second the speeds are 9,94 Mb/s and 16,32 Mb/s respectively. Here it is obvious that the read operation is a couple times faster than writing, however the Table 5 shows the opposite. The difference is that the times in the table show only the execution time in JavaFS, but the speed is measured for the *cp* command which includes the whole execution and other factors as well (e.g. writing to local filesystem may be faster than reading from it, the overhead for writing is greater than reading in FUSE framework, etc.).

system security module (selinux) typically invokes in the *ls -l* command the GETXATTR command for each entry listed. As stated in Table 5 these operations don't have negligible times.

According to the times from the table, listing a directory with 100 entries would in optimal case (each operation is executed only minimum number of times) take approximately 8 seconds. However the tests showed much worse performance. This performance issue has two reasons:

1. The implementation of the filesystem does not include any caching mechanisms.
2. The JAVA implementation of the BTCSP filesystem's readdir method is not optimized.

The next table shows the commands called in *ls –l* command on the root (containing 3 nodes).

#	command	inode	params	result
1	GETATTR	root		root inode attributes
2	OPENDIR	root		
3	GETATTR	root		root inode attributes
4	READDIR	root	pos = 0	3 node names
5	READDIR	root	pos = 3	empty
6	GETATTR	root		root inode attributes
7	GETXATTR	root	"system.posix_acl_access"	empty
8	LOOKUP	root	"Node1"	Node1 inode attributes
9	GETATTR	Node1		Node1 inode attributes
10	LOOKUP	root	"Node1"	Node1 inode attributes
11	GETXATTR	Node1	"system.posix_acl_access"	empty
12	LOOKUP	root	"Node2"	Node2 inode attributes
13	GETATTR	Node2		Node2 inode attributes
14	LOOKUP	root	"Node2"	Node2 inode attributes
15	GETXATTR	Node2	"system.posix_acl_access"	empty
16	LOOKUP	root	"Node3"	Node3 inode attributes
17	GETATTR	Node3		Node3 inode attributes
18	LOOKUP	root	"Node3"	Node3 inode attributes
19	GETXATTR	Node3	"system.posix_acl_access"	empty
20	READDIR	root	pos = 3	empty
21	RELEASEDIR	root		

Table 6: JavaFS commands called in *ls –l* command. The red rows show commands that could accessed items in cache. Blue rows denote commands that could access items in cache if readdir method is optimized for caching.

The table gives an idea about the performance improvement if caching is correctly implemented. The execution time of *ls –l* command would be halved.

In general the BTCSP filesystem did well in the evaluation tests. The BTCSP filesystem provides a useful access into the BTCSP content system. The only drawback of the filesystem is its performance in certain operations, but the optimization of these and other processes belong to the future work.

10.2 Further work

The combination of FUSEbtcsp, JavaFS and BTCSP FS leaves a large space for further work. FUSE is being continually maintained and evolves over time therefore to keep pace with it FUSEbtcsp will always have some rests to do. It would be useful to have some kind of patch that would modify FUSE automatically if possible; supposedly there would not be any crucial API changes.

JavaFS may also be updated with FUSEbtcsp to support new commands or even a new protocol (e.g. encrypted, compressed, negotiation, etc.). Another wished feature would be another communication type between the FUSEbtcsp and JavaFS; e.g. over a pipe, a UNIX socket or using Java RMI.

On the other hand BTCSP FS could support different types of view of the BTCSP data. It could create a deeper hierarchical structure that would group *contents* according to their names in an A-Z directory structure. The reading/writing file operations are not yet optimal and XML parsing of BTCSP responses could also bear some tuning. Also caching is missing in all layers, which would certainly bring some performance advance. Caching would be especially useful in directory listing operation as mentioned in section Evaluation.

As one might see, there is enough work left for the future and if all mentioned tasks are done, the porting of the filesystem driver to Microsoft Windows would be definitely a great mission to undertake.

11. Summary and Conclusion

Many applications store their own data in files or databases organizing them also in a directory tree, but usually keeping them manageable only from the application. But often clients (users) demand out of application access to the data. Software development companies trying to satisfy all their clients are then forced to allow such an access either by internet browsers, filesystems or other kind of software or device. The more options they provide to the clients, the more diverse client groups will be pleased. This thesis gives a glance into the filesystem alternative.

Filesystems are a very useful and necessary part of operating systems and applications. They organize data into a logical hierarchical structure and provide standardize operations to work with them. In course of this thesis we designed and implemented a user-space filesystem BTCSP FS in Java for a commercial database system BTCSP developed by Braintribe. The filesystem involves multiple layers and protocols. An operation on a file is not processed in one layer of the BTCSP FS but is forwarded through all the layers (if necessary over multiple platforms) to the Java implementation of the BTCSP filesystem.

The core of the filesystem that runs in Linux kernel is built by a modified version of FUSE – FUSEbtcsp. It is directly connected to the virtual filesystem (VFS) that handles filesystem calls from user-space and passes them to the underlying filesystem – in our case FUSEbtcsp. FUSEbtcsp exports the VFS interface back to user-space where filesystem operations reach JavaFS module. In this module the operations are translated into requests that are sent over TCP connection to the JavaFS server; implemented in Java. JavaFS server interprets received request as filesystem commands and calls the implementation of the underlying Java filesystem. Here the Java filesystem processes the command by accessing data sources – the BTCSP server – and creates a response that traverses the layers back to the VFS and to the process where the filesystem call originated. As we see, a filesystem call takes a long journey to be taken care of, but at the end it is successfully carried out in the Java application.

References

[capii] **Roman Kurmanowytsch**. *Introduction to the Client API of the Multilingual Content Integration Framework i2z*. Technical report. Braintribe Group, 2005.

[utlk02] **Daniel P. Bovet, Marco Cesati**. *Understanding the Linux Kernel, 2nd Edition*. O'Reilly & Associates, Inc., 2002.

[ufedi03] **Steve D. Pate**. *UNIX Filesystems: Evolution, Design and Implementation, VERITAS Series*. Wiley Publishing, Inc., Indiana USA 2003.

[lki00] **M. Beck, H Böhme, et al**. *Linux Kernel Internals, 2nd Edition. Addison-Wesley*, 2000.

[ldd05] **Jonathan Corbert, et al**. *Linux Device Drivers 3rd Edition*. O'Reilly & Associates, Inc., 2005.

[lkd05] **Robert Love**. *Linux Kernel Development 2nd Edition*. Sams Publishing, 2005.

[dque83] **Robert Elz**. *Disc Quotas in a UNIX environment*. Technical report. Department of Computer Science, University of Melbourne, Australia 1983.

[braintribe] **Braintribe homepage.** *http://www.braintribe.com/*

[wwwfuse] **FUSE homepage.** *http://fuse.sourceforge.net/*

[fusewiki] **FUSE wiki.** *http://fuse.sourceforge.net/wiki/index.php/FUSE%20tutorial*

[prjfuse] **FUSE project homepage.** *http://sourceforge.net/projects/fuse*

[lufs] **Linux Userland Filesystem.** *http://sourceforge.net/projects/lufs/*

[plan9] **Plan 9 Filesystem Protocol.** *http://9p.cat-v.org/*

[coda] **Coda File System.** *http://coda.cs.cmu.edu/*

[cfm] **Callback File System™**. *http://www.eldos.com/cbfs/*

[fusej] **FUSE-j.** http://sourceforge.net/projects/fuse-j

[fw] **FuseWrapper.** *http://arg0.net/wiki/fusewrapper*

[swig] **Simplified Wrapper and Interface Generator.** *http://www.swig.org/*

[ksockprog] **Socket programming in kernel-space.**
http://www.linuxforums.org/forum/linux-kernel/55923-kernel-sockets.html

[runjar] **Classpath in executable jar files.** http://www.timfanelli.com/item/22

Appendix

A. VFS Interface

This appendix describes the most common operations used by filesystems. Detailed information can be also found in *[lki00]* and *[utlk02]*.

super_operations

This is the first interface that the VFS will use. It is saved in the `super_block` strucutre and set by the filesystem's `get_sb` method referenced by the `file_system_type`.

```
struct super_operations {
      struct inode *(*alloc_inode) (struct super_block *sb);
      void (*destroy_inode) (struct inode *);
      void (*read_inode) (struct inode *);
      void (*dirty_inode) (struct inode *);
      void (*write_inode) (struct inode *, int);
      void (*put_inode) (struct inode *);
      void (*drop_inode) (struct inode *);
      void (*delete_inode) (struct inode *);
      void (*put_super) (struct super_block *);
      void (*write_super) (struct super_block *);
      int (*sync_fs) (struct super_block *, int);
      void (*write_super_lockfs) (struct super_block *);
      void (*unlockfs) (struct super_block *);
      int (*statfs) (struct super_block *, struct statfs *);
      int (*remount_fs) (struct super_block *, int *, char *);
      void (*clear_inode) (struct inode *);
      void (*umount_begin) (struct super_block *);
      int (*show_options) (struct seq_file *, struct vfsmount *);
      int (*show_stats)( struct seq_file *, struct vfsmount *);
      ssize_t (*quota_read)( struct super_block *, int, char *,
                             size_t, loff_t);
      ssize_t (*quota_write)( struct super_block *, int,
                              const char *, size_t , loff_t);
};
```

- **alloc_inode**

  This method creates and initializes a new inode under the specific super block
- **destroy_inode**

  This function does the opposite of the `alloc_inode`.
- **read_inode**

  Reads the inode from the disk and fills the data of the provided inode object. The `i_ino` field of the inode object identifies the specific filesystem inode.
- **dirty_inode**

  Invoked by the VFS when the inode is marked as modified (dirty). Journal filesystems (such as ReiserFS) use this to update their journal.
- **write_inode**

  Writes the passed inode object to the disk (`i_ino` identifies the physical inode). The second parameter indicates whether the operation should be synchronous.
- **put_inode**

  Releases the inode object passed as parameter. This doesn't mean that it will be freed from memory, because other processes may still use it, see `drop_inode`.
- **drop_inode**

  This method is called by VFS when the last reference to the inode object is dropped. The caller must hold the inode's lock. Now it is possible to free the inode object from memory, although this method is rarely implemented in which case the VFS simply frees the inode.
- **delete_inode**

  Deletes the inode on the disk represented by the inode object parameter.
- **put_super**

  Just like `put_inode`, releases the reference to the super block object. Called by VFS on unmount.
- **write_super**

  Updates the super block on the disk with the data in the super block object.
- **sync_fs**

  Synchronizes filesystem metadata with the on-disk filesystem. Called when VFS writes out all the data associated with the super block. Second parameter

indicates whether the method should be synchronous; whether it should wait until the process finishes.

- **write_super_lockfs**
  Prevents changes in the filesystem and updates the on-disk data with the super block object argument.
- **unlockfs**
  Unlocks the filesystem locked in the `write_super_lockfs` function.
- **statfs**
  Fills the provided buffer with filesystem statistic information.
- **remount_fs**
  Remounts the filesystem with new mount flags.
- **clear_inode**
  Like `put_inode`, but clears any pages associated with this inode.
- **umount_begin**
  Used only by network filesystems to interrupt a mount operation, because the unmount was already started.
- **show_options**
  Used to display filesystem specific options.
- **show_stats**
  Display all filesystem related information including mount options.
- **quota_read** and **quota_write**
  Functions used to read and write quota files that support resources limitations for different users. The quota files hold information about the usage on the filesystem. Typically these files have name like »quotas« or »quotas.user« and access is allowed only through these methods, which are called by VFS.[20]

inode_operations

This interface contains the operations on an inode. VFS comes into contact with it just after the super operations. This interface doesn't have to be same for each inode. It is stored in the inode object and initialized by the `read_inode` function mentioned above.

[20] Some information about the disk quotas system may be found in [dque83]

```
struct inode_operations {
      int (*create) (struct inode *,struct dentry *,int,
                  struct nameidata *);
      struct dentry * (*lookup) (struct inode *,struct dentry *,
                              struct nameidata *);
      int (*link) (struct dentry *,struct inode *,struct dentry *);
      int (*unlink) (struct inode *,struct dentry *);
      int (*symlink) (struct inode *,struct dentry *,const char *);
      int (*mkdir) (struct inode *,struct dentry *,int);
      int (*rmdir) (struct inode *,struct dentry *);
      int (*mknod) (struct inode *,struct dentry *,int,dev_t);
      int (*rename) (struct inode *, struct dentry *, struct inode *,
                  struct dentry *);
      int (*readlink) (struct dentry *, char __user *,int);
      void * (*follow_link) (struct dentry *, struct nameidata *);
      void (*put_link) (struct dentry *, struct nameidata *, void *);
      void (*truncate) (struct inode *);
      int (*permission) (struct inode *, int, struct nameidata *);
      int (*setattr) (struct dentry *, struct iattr *);
      int (*getattr) (struct vfsmount *mnt, struct dentry *,
                  struct kstat *);
      int (*setxattr) (struct dentry *, const char *,
                  const void *,size_t,int);
      ssize_t (*getxattr) (struct dentry *, const char *, void *,
                     size_t);
      ssize_t (*listxattr) (struct dentry *, char *, size_t);
      int (*removexattr) (struct dentry *, const char *);
      void (*truncate_range)(struct inode *, loff_t, loff_t);
};
```

- **create**

  Run from the `open` and `creat` system call to create a new inode with the given mode and associate it with the provided dentry object.

- **lookup**

  Searches a directory for an inode specified in the dentry object parameter.

- **link**

  Creates a hard link to the file specified by the dentry object in the first argument.

- **unlink**

Removes the given dentry object from the specified directory and decreases the hard link reference count of the inode of the removed dentry.

- **symlink**

  Creates a new inode for a symbolic link to the file represented by the dentry in the given directory.

- **mkdir**

  Function called from `mkdir ()` system call to create a new inode for a directory associated with the given dentry in some directory.

- **rmdir**

  Removes a subdirectory represented by the second argument from the directory in the second argument.

- **mknod**

  Creates a special file just like `create` and associates it with the device specified by the major device number in last argument.

- **rename**

  This function is called by the VFS to move a file identified by the second argument from one directory to another. The new filename is included in the dentry object in the last parameter. This function is called only for a move in the scope of one filesystem. Move crossing two filesystems is solved differently; typically the file is copied to the new filesystem and the old file is deleted.

- **readlink**

  This function is used to get the full path associated with the given symbolic link dentry.

- **follow_link**

  Translates a symbolic link path into an inode and the result is stored in the nameidata pointer.

- **put_link**

  Called by the VFS to clean up after the `follow_link` function.

- **truncate**

  Modifies the size of the file associated with the inode. Before calling thes method the new size should be set in the inode object member `i_size`;

- **permission**
  Checks whether an access mode is possible on the file represented by the given inode. It returns 0 on success and negative error code otherwise. This method is typically set to NULL to let the VFS check the permissions based on the properties of the file (i.e. owner, group, etc.).
- **setattr**
  This function is called whenever the inode is being notified to be dirty (changed).
- **getattr**
  This function is called by the VFS whenever it is notified that the inode should be refreshed from the disk.
- **setxattr**
  Sets extended attribute of the given dentry to the specified value.
- **getxattr**
  Copies the value of an extended attribute into the buffer provided.
- **listxattr**
  Is called by the VFS to get a list of the names of all extended attributes of a dentry. The list is an array of zero terminated strings.
- **removexattr**
  This function removes the given extended attribute from the file.
- **truncate_range**
  Removes a number of blocks in the file starting at the given block position.

file_operations

This section covers the operations of files and directories. Files and directories share the same interface, though directory inodes have typically other implementation of the interface functions than the files. `readdir` function is for example unnecessary for files. To keep things simple I split this interface into groups of functions specific for files, directories and general group that is used in both types. I will start with directory group, because the root is always a directory.

```
struct file_operations {
      struct module *owner;
      loff_t (*llseek) (struct file *, loff_t, int);
```

```
ssize_t (*read) (struct file *, char __user *, size_t,
                  loff_t *);
ssize_t (*aio_read) (struct kiocb *, char __user *, size_t,
                        loff_t);
ssize_t (*write) (struct file *, const char __user *, size_t,
                  loff_t *);
ssize_t (*aio_write) (struct kiocb *, const char __user *,
                        size_t, loff_t);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *,
                        struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int,
                  unsigned long);
long (*unlocked_ioctl) (struct file *, unsigned int,
                        unsigned long);
long (*compat_ioctl) (struct file *, unsigned int,
                        unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*readv) (struct file *, const struct iovec *,
                  unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *,
                  unsigned long, loff_t *);
ssize_t (*sendfile) (struct file *, loff_t *, size_t,
                        read_actor_t, void *);
ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                        loff_t *, int);
unsigned long (*get_unmapped_area)(struct file *,unsigned long,
            unsigned long,   unsigned long, unsigned long);
int (*check_flags)(int);
int (*dir_notify)(struct file *filp, unsigned long arg);
int (*flock) (struct file *, int, struct file_lock *);
ssize_t (*splice_write)(struct pipe_inode_info *,
            struct file *, loff_t *, size_t, unsigned int);
ssize_t (*splice_read)(struct file *, loff_t *,
```

```
                struct pipe_inode_info *, size_t, unsigned int);
};
```

directory functions

There are only one directory specific operations, although being one of the most used one.

- **readdir**

  Reads next directories in a directory listing. A callback function is passed in one argument to be used to fill the given buffer with dentries.

  ```
  typedef int (*filldir_t)(void *, const char *, int, loff_t,
                        ino_t, unsigned);
  ```

  Allows the kernel to read directories into the kernel-space or to have different dirent layouts depending on the binary type.

file functions

File provides a wider interface than the directory. The file specific functions are listed here.

- **llseek**

  Updates the working offset of a file object. Used by the `llseek ()` system call.

- **read, aio_read**

  Called by the `read ()` (`aio_read ()` system call) to read a given number of bytes from a specific offset and file into a buffer. The `aio_read` function starts an asynchronous read.

- **write, aio_write**

  Called by the `write ()` (`aio_write ()` system call) to write a given number of bytes from buffer in the file. Write begins at the given offset that is at the end updated. The `aio_write` function starts an asynchronous write.

- **poll**

  This function waits for an activity on the given file. It is called from the `poll ()` system call.

- **mmap**

  This function performs a memory mapping of the file into a process address space.

- **flush**

  Called whenever the `f_count` field in the file object is decremented (i.e. whenever the last close on a file is called).

- **fsync, aio_fsync**

  Writes all cached data of a file to the disk; writes in asynchronous mode in case of `aio_fsync`.

- **lock**

  This function handles a lock on the given file. Called by the `fcntl ()` system call for the commands: `F_GETLK`, `F_SETLK` and `F_SETLKW`.

- **readv, writev**

  Similar to *`read/write`* functions used to access data of a file, but these methods process work with multiple buffers defined by the given vector.

- **sendfile**

  Called by the `sendfile ()` system call, to copy data of one file into another without switching to user address space.

- **sendpage**

  This function is used to send data from one file to another; this method is used by sockets.

- **get_unmapped_area**

  Gets unused address space to map the given file. Used for mapping frame memory.

- **flock**

  Applies or removes advisory lock on the given file. Advisory locks allow cooperating processes to perform consistent operations on files. For more, see man pages for the `flock ()` system call.

- **splice_write, splice_read**

  Called by the `splice ()` system call to splice data from a pipe to a file (i.e. from a file to a pipe in the `splice_read` method). It is used to move data between to files without copying between kernel and user -space where one side refers to a pipe.

shared functions

As mentioned in the name, these methods are available to all file types, though only some of them apply to a specific type.

- **ioctl, unlocked_ioctl, compat_ioctl**

  Called by the `ioctl ()` system call to send a command with arguments to a device. `ioctl` is one of the remaining kernel parts running under the Big Kernel Lock (BKL). Filesystems that do not require the BKL should use `unlocked_ioctl` instead. The `compat_ioctl` is called by the `ioctl` system call when 32bit system calls are used on 64bit kernels.
- **open**

  Called by the `open ()` system call to create a new file object and associate it with the corresponding inode object.
- **release**

  Releases the file object created in `open` function when the last reference to an open file is closed; i.e. called when the `f_count` reaches 0.
- **fasync**

  Enables or disables asynchronous I/O signal notifications.
- **check_flags**

  This function is used to check the validity of flags passed to the `fcntl ()` system call for the `SETFL` command. This method is recently used only by the NFS filesystem to disable the combination of the `O_APPEND` and `O_DIRECT` flags.
- **dir_notify**

  This function is a new method introduced in the 2.6.8 kernel. It partly does the work of the obsolete `fcntl ()` function (2.6.6 kernel). It is used by CIFS to handle `F_NOTIFY` requests. Called by the `fcntl ()` system call.

B. FUSE Operation Codes and Parameters

The following table lists all FUSE commands together with the input/output parameter.

Opcode	Request arguments	Answer arguments
FUSE_LOOKUP	string – name of the entry to lookup (i.c.z.).	*fuse_entry_out* – information about the inode associated with found dentry. This includes: • inode id • inode generation • cache timeout for the name • cache timeout for attributes • inode attributes (i.e. ino, size, atime, blocks, etc.)
FUSE_FORGET	*fuse_forget_in* – contains the inode *nlookup* field	None
FUSE_GETATTR	None	*fuse_attr_out* – stores inode attributes and cache timeout for the attributes
FUSE_SETATTR	*fuse_setattr_in* – inode attributes except *ino*, *blocks*, *ctime*, *ctimensec*, *nlink* and *rdev*. Besides this it has *valid* and *fd* (file handle) fields.	Same as *FUSE_GETATTR*
FUSE_READLINK	None	string – full path to the entry that the link points to (i.c.z.)
FUSE_SYMLINK	2 arguments string – name of the new entry (i.c.z.) string – full path of the link (i.c.z.)	Same as *FUSE_LOOKUP*
FUSE_MKNOD	2 arguments *fuse_mknod_in* – mode and rdev string – name of the new entry (i.c.z.)	Same as *FUSE_LOOKUP*
FUSE_MKDIR	2 arguments *fuse_mkdir_in* – mode string – name of the new entry (i.c.z.)	Same as *FUSE_LOOKUP*
FUSE_UNLINK, *FUSE_RMDIR*	string – name of the entry to delete (i.c.z.)	none
FUSE_RENAME	3 arguments *fuse_rename_in* – inode id of the parent directory where the new entry will be added string – name of the old entry (i.c.z.) string – name of the new entry (i.c.z.)	none
FUSE_LINK	2 arguments *fuse_link_in* – inode id of the referred inode string – name of the link entry (i.c.z.)	Same as *FUSE_LOOKUP*
FUSE_OPEN, *FUSE_OPENDIR*	*fuse_open_in* – flags and mode	*fuse_open_out* – user-space file handle, open flags and padding
FUSE_READ,	*fuse_read_in* – user-space file	Variable size (maximum *count* bytes

FUSE_READDIR	handle, file offset and count of bytes	in 1st argument), store in pages
FUSE_WRITE	2 arguments *fuse_write_in* – user-space file handle, file offset, size and write flags *count* bytes stored in pages	*fuse_write_out* – size and padding
FUSE_STATFS	None	*fuse_statfs_out* – filesystem information stored in *fuse_kstatfs* (i.e. *blocks*, *bfree*, *bavail*, *files*, etc.)
FUSE_RELEASE, FUSE_RELEASEDIR	*fuse_release_in* – user-space file handle, flags, release flags and lock owner	none
FUSE_FSYNC, FUSE_FSYNCDIR	*fuse_fsync_in* – user-space file handle, fsync flags and padding	none
FUSE_SETXATTR	3 arguments *fuse_setxattr_in* – size and flags string – attribute name (i.c.z.) byte array – attribute value	none
FUSE_GETXATTR	2 arguments *fuse_getxattr_in* – size and padding string – attribute name (i.c.z.)	Conditional If *size* in first input argument is 0 then *fuse_getxattr_out* – necessary size and padding otherwise byte array – attribute value
FUSE_LISTXATTR	*fuse_getxattr_in* – size and padding	Same as *FUSE_GETXATTR*, but byte array contains a list of attribute names (i.c.z.)
FUSE_REMOVEXATTR	string – name of the attribute to remove (i.c.z.)	none
FUSE_FLUSH	*fuse_flush_in* - user-space file handle, unused 32bit, padding and lock owner	none
FUSE_INIT	*fuse_init_in* – major and minor FUSE kernel module version, max read-ahead and flags	*fuse_init_out* – (variable length for backward compatibility) FUSE library major and minor version, max read-ahead, flags, unused 32bit and max write
FUSE_GETLK	*fuse_lk_in* - user-space file handle, owner and *fuse_file_lock* (i.e. *start*, *end*, *type* and *pid*)	*fuse_lk_out* - *fuse_file_lock*
FUSE_SETLK, FUSE_SETLKW	Same as *FUSE_GETLK*	none
FUSE_ACCESS	*fuse_access_in* – mask and padding	none
FUSE_CREATE	2 arguments *fuse_open_in* – flags and mode string – name of the new entry (i.c.z.)	2 arguments *fuse_entry_out* – see *FUSE_LOOKUP* *fuse_open_out* – see *FUSE_OPEN*
FUSE_INTERRUPT (has a special handling)	*fuse_interrupt_in* – interrupt unique id	none
FUSE_BMAP (used only by device backed filesystems)	*fuse_bmap_in* – block within file, block size and padding	*fuse_bmap_out* – block within device
FUSE_DESTROY	None	None

Table 7: List of commands with input and output parameters. i.c.z – including closing zero

C. List of FUSE Modifications related to 256bit inode support

This appendix lists all the modifications in FUSE that are related to the 256bit support.

Line #	Change	description
Changes in file include/fuse_common.h:		
+ 18:	`#include <sys/stat.h>`	Includes support for *stat* structure
+ 123:	`struct fuse_stat {…};`	New inode attributes used only in kernel-user-space data exchange.
+ 130:	`struct fuse_inode_id {…} fuse_ino_t;`	New inode identification structure, includes a byte array of 32 bytes (256bit) named *i_userdata*.
+ 136:	`FUSE_FIRST5_USERDATA(usrdt)`	Macro used to represent the first 5 bytes of the passed *usrdt* array.
+ 137:	`FUSE_COPY_IUSERDATA(dst,src)`	Macro copying the *i_userdata* field from structure *dst* to *i_userdata* field in *src*. The number of copied bytes depends on the size of the field in *dst*.
In file include/fuse_kernel.h (and in kernel/):		
+ 82:	`char i_userdata[32];`	The 256bit inode identification in fuse inode attributes - *fuse_attr*.
+ 178:	`fuse_inode_kernel_id {…} fuse_ino_kernel_t;`	Inode id structure defined for kernel (in principal same as `fuse_ino_t`)
± 186:	`- __u64 nodeid;` `+ fuse_ino_kernel_t noedid;`	Change of types for inode id in object *fuse_entry_out*.
± 218:	`- __u64 newdir;` `+ fuse_ino_kernel_t newdir;`	Same as above, but for object *fuse_rename_in*.

± 224:	`- __u64 oldnodeid;` `+ fuse_ino_kernel_t oldnodeid;`	Same as above, but for object *fuse_link_in*.
± 367:	`- __u64 nodeid;` `+ fuse_ino_kernel_t nodeid;`	Same as above, but for object *fuse_in_header*. (request header)

In file include/fuse_lowlevel.h:

- 30:	`#include <sys/stat.h>`	Has been moved to fuse_common.h.
- 45:	`typedef unsigned long fuse_ino_t;`	Has been changed and moved to fuse_common.h.
± 82:	`- struct stat attr;` `+ struct fuse_stat attr;`	In object *fuse_entry_param*. The new inode attributes with 256bit inode id.
± 236:	In function `setattr` change of *attr* parameter from *stat* to *fuse_stat* type.	New parameter type with 256bit support.

In file kernel/dir.c:

±112,113:	`req->in.h.nodeid.ino = get_node_id(dir);` `get_iuserdata(dir, req->in.h.nodeid.i_userdata);`	Changes the way inode id is copied to request header. In first line *nodeid* in request header is now a structure and *ino* is its field. Second line copies the 256bit inode id.
± 167:	`if (!err && !outarg.nodeid.ino)`	*nodeid* is a structure with *ino* field that hold the older *nodeid* value.
±171,256, 257,259, 261,429, 430,441, 447,566, 567,598, 599,626, 627,629, 630,675, 676,705, 706,803,	The changes because of the new nodeid structure are always the same therefore the lines left show all the places where this changes were made, but it isn't necessary to show the code. Typically `nodeid` is changed into `nodeid.ino`. The function `get_iuserdata` is used in case the 256bit inode id also needs to be copied.	

Line	Change	Comment
804,963, 964		
+ 419:	`struct fuse_inode *fi;`	In function *create_new_entry* it was necessary to add additional code to copy the 256bit userdata.
+ 454:	`fi = get_fuse_inode(inode);` `memcpy(&(fi->nodeid.i_userdata), &(outarg.nodeid.i_userdata), sizeof(outarg.nodeid.i_userdata));`	

<u>*In file kernel/file.c*</u>:

Line	Change	Comment
± 38,39, 220,221, 260,261, 293,294, 455,456, 699,700, 833,834	Same as in the file dir.c, the lines left show the changes required because of the change in `nodeid` structure.	
± 122:	Change in the function `fuse_release_fill` of the `nodeid` parameter type from `u64` to `fuse_in_kernel_t`.	
± 147:	The function `fuse_release_fill` takes as 2nd argument `get_kernel_ino(inode)` instead `get_node_id(inode)`.	This is modification is a result of the above change.

<u>*In file kernel/fuse_i.h*</u>:

Line	Change	Comment
± 118:	`- u64 nodeid;` `+ fuse_ino_kernel_t nodeid;`	In object *fuse_inode*. This is the object stored together with kernel *inode* object.
+ 463:	`… get_kernel_ino(…`	Implementation of the static inline `get_kernel_ino` function that is used to get the `fuse_ino_kernel_t` from a kernel inode.
± 470:	`return get_fuse_inode(inode)->nodeid.ino;`	Sticks with the structural change of nodeid.
+ 474:	`… get_iuserdata(…`	Implementation of the static inline `get_iuserdata` function that copies the 256bit userdata from a kernel

		inode object into a 256bit buffer.
± 486:	Change of the `nodeid` parameter type from unsigned long to u64 in function `fuse_iget`.	
± 493:	Change of the `nodeid` parameter type from unsigned long to `fuse_ino_kernel_t` in function `fuse_send_forget`.	
± 513:	Change of the `nodeid` parameter type from u64 to `fuse_ino_kernel_t` in function `fuse_release_fill`.	Same as in file file.c on the line 122. Here is the declaration of the function.

In file kernel/fuse_i.h:

± 61,99, 100,186, 538,545	Changes resulting from structural changes of `nodeid`. Same as in dir.c (lines 171, 256…) and file.c (lines 38, 39…).	
± 513:	Change of `nodeid` parameter in function `fuse_send_forget`.	See change in function declaration in file fuse_i.h on line 493.
+ 133:	One line added to the implementation of `fuse_change_attributes` function.	Copies 256bit userdata from inode attributes into inode.
± 185:	Change of type of the `nodeid` variable from unsigne long to u64.	In implementation of `fuse_inode_set()`.

In file lib/fuse_lowlevel.c:

± 65:	The function `convert_stat` has been modified to handle the new `fuse_attr` attributes.	
+ 86:	New function `convert_kernel_inode_id` added to convert type `fuse_ino_kernel_t` to `fuse_ino_t`.	
±290,291, 316:	Changes resulting from structural changes of `nodeid`.	
± 338:	Parameter `attr` changed to type `fuse_stat` in function `fuse_reply_attr`.	
± 463:	`struct fuse_stat stbuf;`	Variable `stbuf` changed to type `fuse_stat`.
± 463:	`convert_attr(arg, &stbuf.stat_attr);`	

±554,566, 1166:	Conversion of `fuse_ino_kernel_t` inode id to `fuse_ino_t`.	
+ 1039:	`uint64_t fuse_req_unique(…) {…}`	New function added that gets request id from a request object.

Table 8: List of changes related to the 256bit inode support. First row shows the line number. '+' means one or more lines were added. The character '-'means a line was deleted and the symbol '± ' stands for one or more modified lines. In modified code sections red ink denotes added text.