

Union Mounts for Linux

Jan Blunck

4. Juni 2005

Lizenzbestimmungen

Dieser Beitrag ist unter der Creative Commons Licence <<http://creativecommons.org/licenses/nd-nc/1.0/>> lizenziert.

Zusammenfassung

Unlike a traditional mount that hides the contents of the mountpoint, a union mount presents a view as if the filesystems are merged together. Although only the topmost filesystem of the union stack can be altered, it appears as if it is possible to delete or modify anything. Files in the lower layers may be deleted with whiteouts in the topmost layer. Modified files are automatically copied into the topmost layer first. Union mounts make the implementation of some applications easier, e.g. live cds or source tree management. In combination with an execute-in-place filesystem, union mounts can be used for efficient software management on read-only filesystems shared between Linux z/VM guests. For a VFS based implementation, heavy changes to the VFS and some of the low-level filesystems are necessary. This includes modification of lookup and directory reading operations as well as the introduction of a persistent whiteout file type.

After giving an overview about union mounts and some demonstrations, the speech will focus on the kernel implementation.

The target audience are people interested in kernel development and filesystems.

Union Mounts for Linux

Jan Blunck (j.blunck@tu-harburg.de)

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

1 Introduction

The mount mechanism in UNIX-like operating systems provides a way to attach complete file systems to directories in the *UNIX file system namespace*¹. Traditionally, the mount operation is opaque, which means that the contents of the mount point, the directory where the file system is mounted on, is hidden afterwards until the file system is unmounted. The mounted file systems are organized in a *file system hierarchy*, which keeps track about the stacking of file systems upon each other. The per directory view on the file system hierarchy is called *mount stack* and reflects the order of file systems which are mounted on a specific directory.

Union mounts provide a way to transparently attach file systems to their mount point. They present the users a single unified view of the contents of two or more file systems as if they are merged together. The file system objects which are part of a unified view are ordered in a so-called *union stack*.

Linux is an UNIX-like operating system so it also follows the “*everything is a file*” paradigm. From a programmers point of view that means every object in the file system namespace can be dealt with like a file. The differences between the file system objects are handled as *file types*². Due to the fact that some file type’s data isn’t interpreted by the operating system (e.g. regular files) or a definition of merging the data in a meaningful way isn’t naturally given (e.g. device files, sockets), a unified view for all kinds of file types doesn’t make sense.

For directories the definition of merging them is naturally clear, as the data of a directory consists of a list of file or directory names. This leads to the problem of merging two or more lists.

On union mounted file systems, a directory listing consists of the concatenation of the directory listings of the directories on the union stack in which file name duplicates are removed.

The requirement to remove file name duplicates from the directory listing follows from the fact that the Linux file system semantics doesn’t provide a way to let the users access the different file system layers in the mount stack.

There are some possible solutions to address this problem:

- The topmost layers file name is unchanged. The lower layer files are renamed to “filename#layernu
- The different layers are accessible through special directories named “... for the first layer” for the second and so on.

Both proposals lead to new restrictions with respect to filenames on union mounted file systems. They introduce additional reserved filenames (besides “. and “..”) which users are not allowed to use anymore. This breaks user-space compatibility and therefore is considered as a bad design.

¹The UNIX file system namespace basically consists of the file system hierarchy and the namespaces of the specific file systems. Some definitions also refer to it as the *UNIX namespace* which also includes special files like devices, fifos and sockets.

²Well known file types are regular files, directories, device files, symbolic links, named pipes and sockets.

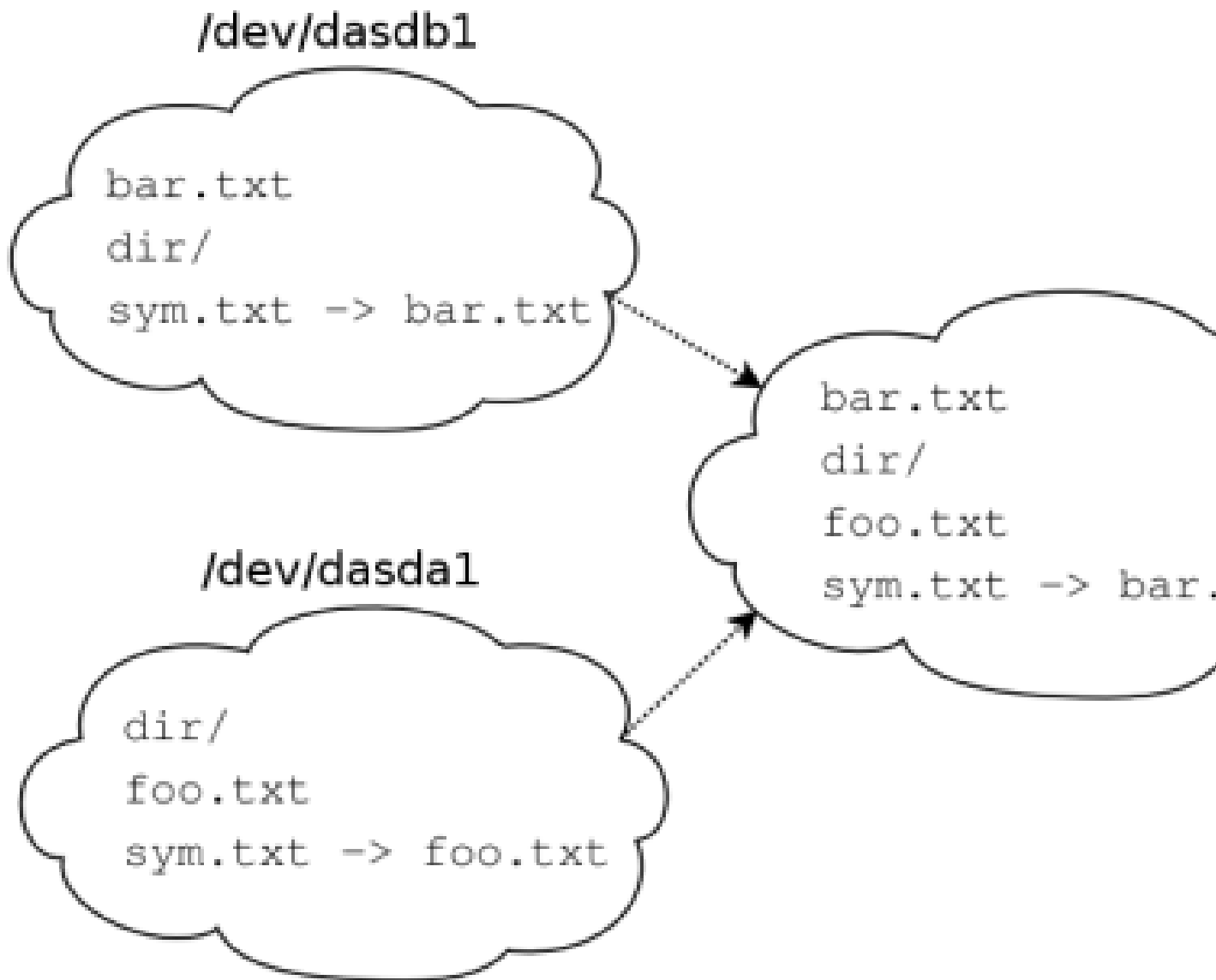


Figure 1: Example union mount directory listing

Therefore the implementation sticks to the behavior of traditional mounts. With traditional mounts only files from the topmost file system layer of the mount stack are accessible by users.

On union mounted file systems, if homonymous files exist on different layers of the mount stack only the topmost file is accessible.

On traditional mounted file systems only the files from the topmost layer of the mount stack are visible by users. Hence, they can write to the topmost file system only.

By definition, on union mounted file systems, writes go to the topmost file system only.

Otherwise, when mounting a mixed set of read-only and writable file systems users might become confused on which file system the write occurs.

As the topmost file on union mounted file systems doesn't need to be on the topmost file system, a copy-up operation between the lower layers and the topmost layer is necessary. When a file is modified, it is copied to the topmost layer first. When deleting a file from a lower layer of the mount stack, an identically named whiteout file is placed in the topmost file system.

Like with traditional mounts, with union mounts it is only possible to unmount file systems in the reverse order than they have been mounted. The last file system mounted in a mount stack must be the first to be unmounted.

2 Changes to VFS objects

One of the authors main goals was to keep the implementation as simple as possible. Therefore the author focused on the extension of existing VFS objects instead of introducing new ones.

In the Linux kernels VFS the file system hierarchy is represented by the `vfsmount` structure. This structure is being used during lookup to find the next file system on the mount stack if a mount point is passed. Since this structure reflects the stacking of whole file systems, it is not reusable as a union stack. The union stack should reflect the stacking of existing files on the unified file systems. Thus the union stack is a per-file structure.

The information of the union stack is only useful if the corresponding file is a directory that exists on different layers. Only in then there should be a relationship between those files. Therefore the union stack should be included in an object which is associated with the filename.

Union stacks have following properties:

- They include homonymous directories from different file system layers of the mount stack.
- They never include any non-existent files.
- They only exist as temporary objects, otherwise problems arise when mounting parts of the union in another context.

In the virtual file system of the Linux kernel, a file is represented by different structures:

- The `inode` structure, which represents an existing file in the low-level file system.
- The `dentry` structure, which represents a file name and the directory tree in which it resides. The `dentry` maybe has a reference to an `inode`³.
- The `file` structure, which represents the usage of a file by a specific user and assigns an `inode` and a `dentry` to it.

³A *negative dentry* is a `dentry` which corresponding file in the file system is not existing. Therefore it doesn't has a reference to an `inode`. They are used to speed up the lookup on non-existent files.

Unlike the dentry, the inode is partially stored on the low-level file system's media. Union mounts shouldn't have any impact on persistent objects. Therefore the inodes shouldn't be altered to implement a union stack. The file structure is dynamically created for every opened file instance. Implementing the union stack inside this structure would waste a lot of memory. Hence, the dentry structure is the only file system object which needs to be changed to implement the union stack.

```
struct dentry {
    atomic_t d_count;
    unsigned int d_flags;
    spinlock_t d_lock;
    struct inode *d_inode;
    struct dentry *d_parent;
    struct qstr d_name;
#ifdef CONFIG_UNION_MOUNT
    struct dentry * d_overlayed; /* overlayed directory */
    struct dentry * d_toplevel; /* topmost directory */
    struct union_stack * d_union; /* union stack data */
#endif
    struct list_head d_lru;
    struct list_head d_child;
    struct list_head d_subdirs;
    struct list_head d_alias;
    unsigned long d_time;
    struct dentry_operations *d_op;
    struct super_block *d_sb;
    void *d_fsdata;
    struct rcu_head d_rcu;
    struct dcookie_struct *d_cookie;
    struct hlist_node d_hash;
    int d_mounted;
    unsigned char d_iname[DNAME_INLINE_LEN_MIN];
};
```

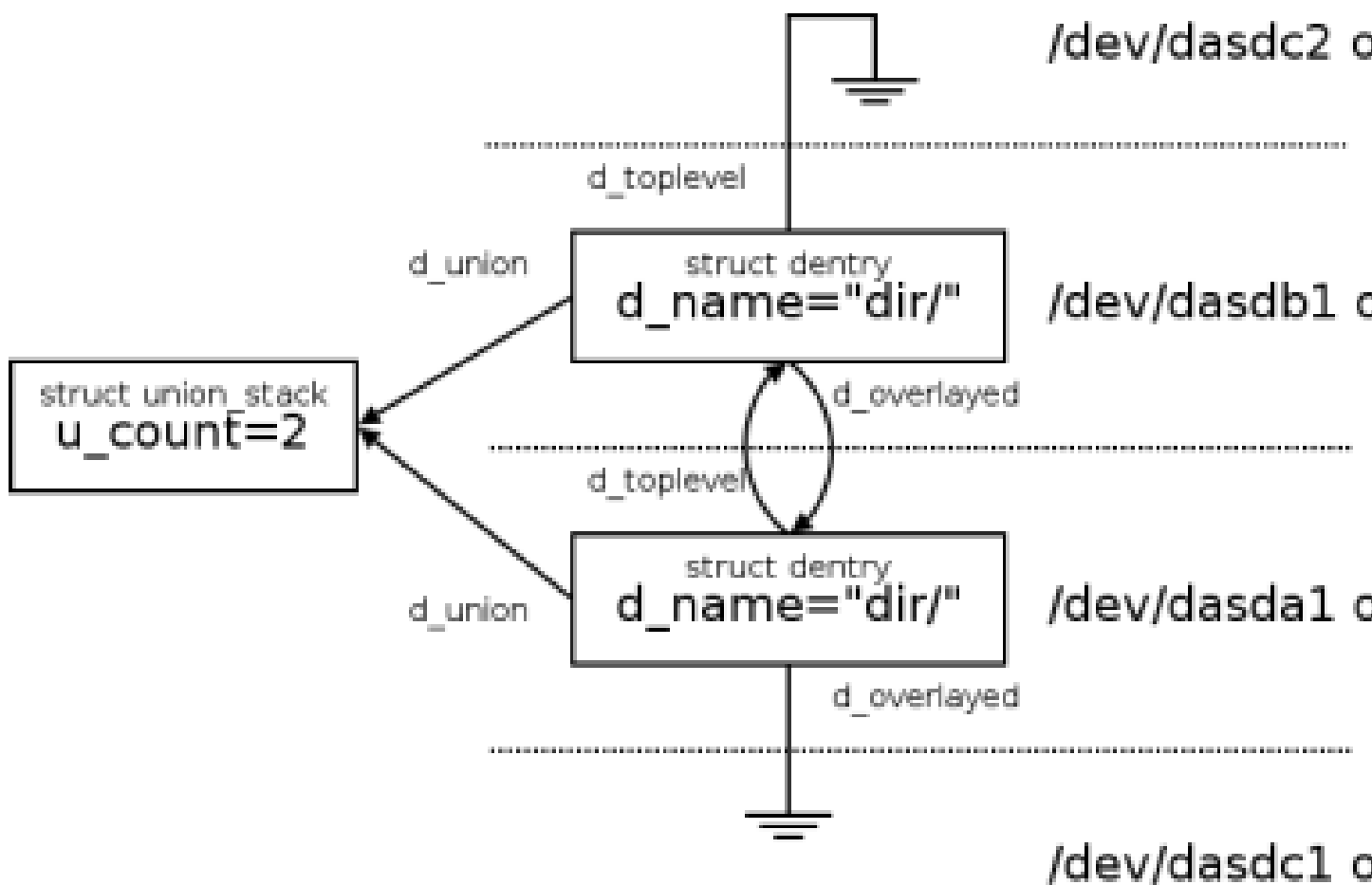


Figure 2: Example union stack

The union stack is implemented as a list of dentries. While every dentry must include the pointers necessary to attach the dentry to the union stack, there is one single object that describes the union stack. That is the union_stack structure. The union_stack structure keeps track about the locking and the reference counting of the dentries in a union stack.

```
struct union_stack {
    struct semaphore u_mutex;
    atomic_t u_count;
};
```

3 Lookup and the dentry cache

Although, the modification of the dentry structure is focused on small changes in the handling of union stacks, it has a huge impact on some parts of the VFS.

The VFS is, through its system call interface, the interface between the users and the low-level file systems. Often, the system calls take a pathname as an argument which the VFS has to look up before proceeding with its further operations. The process of looking up a file

is a central part of the VFS functionality. The result of a look up are two objects: a dentry with its corresponding vfsmount object.

With union mounts it is possible that the VFS lookup returns not only a single dentry but the topmost dentry of a union stack.

Most of the VFS lookup mechanism is build around the following two functions:

- `link_path_walk()` is the full featured way to lookup, including mount stack walking, following of symbolic links and resolving of a complete pathname
- `lookup_hash()` doesn't follow links or mount points and only looks up a single pathna-me component in a directory

These two function calls utilize some internal functions for looking up the components of a pathname ⁴ either from the *dentry cache* (also referred to as the *dcache*), via `d_lookup()`, or directly from the media, via `real_lookup()`.

When working with union mounts the lookup has to assure that the union stack in the dcache is valid. Since the lookup starts at the topmost directory of the union stack, the first dentry which is found is also the topmost one. This dentry is locked through the union's mutex lock (`dentry->d_union->u_mutex`) against concurrent traversal until the complete lookup has finished. Because the lookup might need to do I/O, e.g. to read an inode from a hard disk, the code might sleep while holding the union lock.

If the dentries of the union are still available from the dcache, it is important to rebuild the union stack as long as one of the union's dentries is unused ⁵. If one of the dentries is not found in the dcache, a real lookup from the hard disk must be started.

When following a path to the parent directory (through following `“..”`) it is important that the union stack is followed to the topmost directory afterwards. Otherwise the next lookup doesn't start on the topmost directory and might miss some files.

Most users of the lookup mechanism don't need to know about the possibility that they are using union stacks instead of single dentries. To keep source code changes at a minimum level, the normal reference counting on the dentry structure is changed to support union stacks:

- `dget()` must reference every dentry in the union stack to assure that none is discarded from memory while one is in use. Therefore it is walking down the union stack and references every dentry on its way. Due to that fact it has to hold the union mutex lock. **With union mounts `dget()` might sleep.**
- `dput()` must dereference the dentries of the union stack. Like `dget()` it has to hold the union mutex lock too, but since the original `dput()` might already sleep that doesn't

⁴When looking up the pathname `“/a/b/c/d.txt”` the components are `“/”`, `“a”`, `“b”`, `“c”` and `“d.txt”`. The first component `“/”` of the pathname is the root directory of the root file system while later occurrences of `“/”` describe the UNIX path separator.

⁵A dentry can have three different states: used, unused and negative. The dentry is unused, if it points to a valid inode (`d_inode != NULL`) but the reference count (`d_count`) is 0. Unused dentries can be discarded if the kernel needs to reclaim memory.

make any difference. When one of the union stack dentries reference count reaches zero, `dput()` needs to destroy the union stack because otherwise the traversing of the union stack might lead into already freed memory.

4 The `readdir()` union cache

As already mentioned above, union mounts make the contents of many directories visible as if they are merged together. One issue to consider is that duplicate names in the directory listing must be suppressed. Duplicate file names occur whenever a name is present in more than one file system layer, in other words in more than one directory of the union stack.

When the directory listing is read via the `getdents()` or `readdir()` system call, the union stack is traversed from the topmost dentry. The Linux kernel keeps track about the names already delivered to the userspace with the help of the union cache.

```
struct union_cache_entry {
    struct list_head list;
    struct qstr name;
};
```

The union cache is a list of *quick strings* (`struct qstr`) which is allocated while traversing the union stack and freed before returning from the system call. The union cache is used to suppress the duplicate names before they are copied to the userspace buffer.

The suppression of whiteouts, which are explained in detail below, is handled in a similar way. A whiteout is added to the union cache but it is never copied to the userspace buffer because `filldir()` and `filldir64()` ignore them.

5 File Copy-up

One restriction arises from using union mounts: writes should only occur on one file system of the union stack. Otherwise users might become confused about on which device the writes occur. Per definition, the topmost file system is the only writable file system in the union stack, due to the fact that this is similar to the traditional mounts behavior.

Because the lower layers of the union stack are not writable, files which should be modified but reside on a lower layer file system must be copied to the topmost layer first. Since the userspace doesn't have access to the layers of the union stack, the file copy must be done in the kernel. This is done by utilizing Joern Engels MADCOW patches (<<http://wohnheim.fh-wedel.de/~joern/cowlink/>>).

At the moment only a *copy-on-open* (COO) copy-up is implemented. Although this might not be optimal, it is reducing the complexity of the implementation especially when it comes to memory mapped files.

The copy-up is triggered from `open_namei()`, when the user opens a file either with `O_CREATE` or with a writable file mode. This leads to a call to `copy_data()` which is basically using `vfs_sendfile()` to do a local in-kernel file copy.

6 Whiteout, the ENOENT filetype

With union mounts only the topmost file system in the union stack is writable. This leads to a problem when removing files from the lower layers of the union stack.

The solution to this problem is a new file type, called *whiteout*. A whiteout stops the VFS from further lookups of the whiteouts name. A whiteout is a file which doesn't exist from a logical point of view. Whiteouts may be created when calling following VFS functions:

- `sys_unlink()`: if the file to unlink resides in the topmost file system it is unlinked first
- `sys_rmdir()`: if the directory is present in the topmost file system it is removed first including its possibly contained whiteouts

The renaming of files is handled differently but creates also a whiteout file for the old filename.

Whiteouts also have an impact on other VFS calls. Whenever a file is created, a corresponding whiteout may be deleted first. These checks must be made in the following calls:

- `vfs_create()`, `vfs_mknod()`, `vfs_symlink()`, `vfs_link()`: only need to unlink the corresponding whiteout
- `vfs_mkdir()`: if there is a homonymous directory in a lower layer file system of the union stack the new directory which is created must have the `S_OPAQUE` inode flag set

The `S_OPAQUE` flag on newly-created directories is needed because otherwise the directory listings of overlayed file systems might become visible. This might confuse the users as they expect to have an empty directory after a call to `vfs_mkdir()`. The `S_OPAQUE` flag on a directory tells the directory reading functions to stop walking the union stack further down. Earlier versions of this union mounts implementation used to whiteout every file in the lower layer directories. This was dropped because the `S_OPAQUE` flag is more flexible, since it can be turned off again, and more efficient, since a lesser number of whiteouts is created.

```
/* from <linux/stat.h> */
#define S_IFWHT 0160000 /* whiteout */
#define S_ISWHT(m) (((m) & S_IFMT) == S_IFWHT)
```

Whiteouts are implemented as a real file type. File systems which support the new file type must set `FS_WHT` on `.fs_type` in their `file_system_type` structure and implement the whiteout operation in their `inode_operations` structure:

```
int (*whiteout) (struct inode *, struct dentry *);
```

At the time of writing this, the only two file systems with support for whiteouts are the *second extended (EXT2)* and the *RamFS* file system.

7 Directory renaming

As mentioned above renaming of directories is problematic on union mounts. When the user is renaming a directory, he expects the contents of the directory to be unchanged afterwards. Since union mounts provide a single view on more than two directories with the same name, only renaming the topmost directory isn't sufficient. This may result in totally different directory contents after renaming it because with the renamed directory the union stack might have changed on the next lookup.

There are two possible solutions:

- The lazy solution, which is implemented right now and works for all file systems, is to return `-EXDEV` for renamed directories and let the userspace recursively copy the directory contents. Afterwards the renamed directory is removed. The downside is, that a lot of space on the topmost file system might be wasted.
- The complex solution works with extended attributes (XATTR) and thus isn't available on all file systems, as extended attributes are only implemented on some. When renaming a directory, the filename which should be used to lookup the directory in the next lower layer is stored as an extended attribute. The renamed directory works like a directory and a symbolic link at the same time.

At the moment, only the lazy solution is implemented because it is available for all supported file systems.

8 Optimization

The implementation of union mounts is far from being optimal. There are some things which the author thinks must be addressed:

- Directory reading is really inefficient, since for every attempt to read a directory's content, it is necessary to read everything from the start of the first directory on the union stack unto the file position which might be in another directory. Additionally, a lot of memory allocations are necessary to maintain the union cache.
- In some situations, especially when users try to modify files on union mounted file systems and have insufficient permissions, unnecessary copy-up operations occur. That wastes file system space and inodes.
- Directory renaming is implemented in a very wasteful way. Due to the fact that extended attributes are not available for every file system and to keep the implementation as simple as possible, this was the right choice. Since the only file systems which support whiteouts, and therefore are supported as writable topmost file systems, are EXT2

(XATTR available) and RamFS (XATTR not available) it might be necessary to implement XATTR for RamFS and switch to the XATTR renaming method.

- Instead of allocating an inode for every whiteout in the file system, a singleton inode which is stored in the superblock object should be used. Whiteouts doesn't store any information and therefore are an ideal candidate to become a singleton object.

Bibliography

[1] Jan-Simon Pendry, Marshall Kirk McKusick: "Union Mounts in 4.4BSD-Lite, USENIX Winter 1995 Technical Conference, <<http://www.usenix.org/publications/library/proceedings/neworl/mckusick.html>>

[2] Bernhard Wiedemann: "The concept of translucency, <<http://translucency.sourceforge.net/doc/conceptual/idea.ps>>

[3] Remy Card, Theodore Ts'o, Stephen Tweedie: "Design and Implementation of the Second Extended Filesystem, <<http://web.mit.edu/tytso/www/linux/ext2intro.html>>

[4] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, Erez Zadok, and Mohammad Nayyer Zubair: "Versatility and Unix Semantics in a Fan-Out Unification File System, <<http://www.fsl.cs.sunysb.edu/docs/unionfs-tr/index.html>>