

Implementing Fallthru links in Overlay Mounts

Alireza Saberi and DilipSimha

December 10, 2009

Thanks to Professor Erez Zadok for his great support and guidance all through the project. Special thanks to Valerie Aurora for giving us this project and for her invaluable documentation. Thanks to all our friends who joined with us in lengthy discussions.

Appendix A

Abstract

In the present design of Overlay mounts[2], fallthrus are created during directory read operation. Fallthrus are like regular files except that they have a special dirent file type.

When users want to change any information about a file, the entire file is copied up to the overlay layer and the desired changes are applied to the newly created file on the upper layer. The newly created file on upper layer will now have it's own inode, data, but will carry the same file name as found in lower read only layer.(Lets exclude rename operation for sometime)

This copyup operation is done irrespective of the change requested by user. If the change is a metadata change like permission bits on a file, then also we have to copyup the entire file.

Hence we propose a new design to differentiate between data and metadata changes and do a copyup only in case of data changes. For metadata changes, we propose to make changes on the fallthru-link file in overlay FS.

Appendix B

A brief overview of how Overlay mount works

A unioning file system combines the namespaces of two file systems to make a single uniform view. Different unioning file systems have been developed since 1988 for different operating systems. These unioning file systems have had the same idea while they use different approaches in design and implementation. Linux Union Mount, aka Overlay mounts is a unioning file system which is implemented at VFS layer. Overlay mounts (OM) is designed to work with multiple read write layers on top of a read only underlying FS.

However, as of today[2], it only handles one read write layer as overlay and one read only layer as underlay layer. The Union Mount file system needs some minor support from overlay file system for special kind of objects which are called whiteouts, fallthrus and opaques. These objects are implemented by using regular files with special dirent file types. The underlying file system needs no change. Lets focus on one of the design decisions in OM: copyup mechanism. Whenever a user wants to change some information of a file, then UM cannot directly apply the change, because currently the file resides on a read only FS. So OM's design is to copyup the entire file to overlay FS which must be writable(this is forced at the time of mount) and then apply the changes on the newly created file.

Any further lookups or changes will result in pointer to this new file and hence the read only file is no longer accessed by anyone. When directory read (`read_dir()`) is called, there are additional challenges because of the strict posix semantics to maintain the current offset position to the child dirents. To handle this, fallthrus are created for every child in the directory we read. Of course for files that are already copied up or for files which are created fresh on overlay, this copyup procedure is skipped. This Fallthru file is represented as a special file. It's type is set to UNKNOWN and it's file type in the parent's dirent structure is set to FALLTHRU. There is also a special dentry flag to mark a dentry as FALLTHRU.

This is how lookup procedure can identify that a file is fallthru and hence it proceeds lookup to the underlying FS. Note however that this FALLTHRU file on overlay FS does not have any inode. So the dentry with FALLTHRU flag set will either have it's inode NULL or will point to the underlying FS.

Appendix C

Current implementation drawbacks

1. Copyup procedure is invoked irrespective of the type of change to the file. So if there is a metadata change like permission change or owner change, then still entire file is copied up. This is a drawback if the file size happens to be very huge.
2. If there are any hard links on the underlying FS, then if one of the hard links is moved to a different location, none of those hard link files can be accessed. It's a flaw in the implementation.[7]
If one of the hard links are moved, then a whiteout is created for it on the overlay FS, so that any further access to the same file should now report as being deleted.
A new file should be created on overlay FS which is a copy of the file from underlying FS. Now if somebody accesses any of the hard links to this moved file, then they should automatically access this new file because hard links fundamentally have the same inode number.
But in the current implementation, the hard link files will point to the older inode number on the underlying FS and hence they report incorrect data.
3. If a metadata change is done to a file which is not already copied up to the overlay FS, then it should be copied to the overlay FS and then the desired changes should be applied there.
But in the current implementation, they fail to copyup and hence they try to apply changes to the underlying file This will obviously not succeed because you cannot change anything on a read only FS.
Hence this is a flaw.
4. If a file is renamed, then the following steps are performed:
A whiteout is created for the file on overlay FS.
File is copied up and then renamed.
This approach is not efficient because rename is just like a metadata change and hence the costly copyup operation should be avoided.
5. If a directory is renamed, then the following steps are performed:

A whiteout is created for the file on overlay FS.
File corresponding to the directory is copied up and then renamed.
Now what will happen to the lookups on the files inside this directory?
It will always result in NOT FOUND, because at the time of lookup, if a directory finds that it doesn't have the requested file, it will try to search with the same path name in underlying layer. We know that this pathname doesn't exist on the underlying layer.
In the current implementation this is not handled at all:
Rename operation simply returns EXDEV. It's just not supported!
But if you do a rename or move operation using some mv utility on linux, all of the child files are copied up too. This process is continued recursively, until all child files are copied up.
This is done intelligently by user tools and is not done by Overlay mount code.

Appendix D

Possible solutions to solve these drawbacks

1. To avoid unnecessary copyups, we should store the metadata changes somewhere and then apply it on every lookup/stat.
2. To handle hard links properly, we should map inode information for every rename/move operation. So that whenever somebody accesses an inode which is present in our mapping, we should return the new mapped inode.
3. To handle rename operation efficiently, the new name should be stored persistently somewhere and avoid the costly lookup.
Or else only required metadata changes should be copied up and they should point to the underlying file for any data access.
4. To handle directory rename operation efficiently: We should copy the given directory completely to overlay FS.
Then make all the child files as some sort of a symbolic link which will point down to the correct file on underlying FS.

Appendix E

Goals

We split the project into following stages:

1. **Provide a readonly Fallthru link**
2. **Make chmod work and hence show that metadata changes can be applied without doing a copyup**
3. Make a full working copy of fallthru link.
4. Handle rename operation of files.
5. Handle rename operation on directories.
6. Handle hard count issue.

Appendix F

Our new design

Based on the ideas proposed by the maintainers and well wishers of Overlay mount code, we propose to implement fallthru as some sort of a symbolic link called fallthru-link. As a proof of concept, we wish to implement the changes in ext2 FS. In this project, we are aiming at solving the first drawback mentioned above.

This new fallthru-link is like a symbolic link, except that it's dirent type will be FALLTHRU and file type a regular file. So this new file type will have it's own inode and inode number. But it's not visible to users, because, whenever the lookup reaches to a fallthru file, it will continue lookup on the underlying FS and will report back the corresponding file on underlay FS.

For a fallthru file, at the end of lookup, we will have nameidata and path filled with inode details[1] from corresponding file on underlay FS. The inode information, which is the metadata of the file, will now have information merged from both underlay and overlay FS. Inode number, length of file and other file data specific information should be reported from the inode of the file in underlay FS. Other inode details should be used from file in overlay FS.

figure: Path Lookup, describes the lookup operation. The edges are labelled with the following meaning:

1. Cache lookup
2. If cache fails, come to overlay FS
3. Do a real lookup on overlay FS.
4. If file is a fallthru-link, then go to underlay FS.
5. Do a real lookup on underlay FS
6. Use metadata from files in overlay(fallthru link) and underlay FS. Make a consistent view in dentry(dcache)

Note that this merged inode information is only in memory and not committed to disk. Whenever the in-memory copy is lost, lookup builds it again.

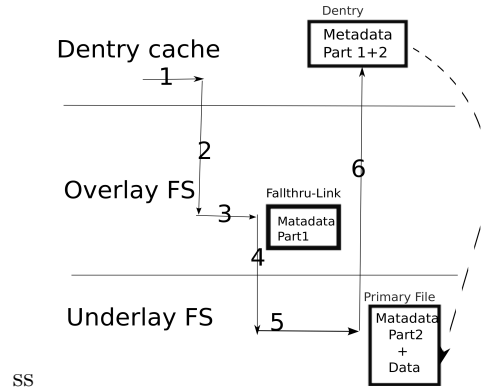


Figure F.1: Path Lookup

There are two possible options to handle this merged view:

1. Default lookup, to point at inode of upper file.
This approach needs to fetch details from underlay file and then merge the inode information and place it in inode information of the overlay file.
2. Default lookup to point at inode of lower file.
Here we need to remember the inode details of overlay file inode details before continuing lookup on file in underlay FS.
This way we can merge inode information from both views and update the merged view in lower FS inode.
We chose this approach because the current code flow is on this line. We just need to make minor modifications to existing code.

To handle metadata changes, we need to identify a common entry point for all metadata changes and then provide the inode of the file on overlay FS. If not, metadata changes cannot be updated on the underlay FS which is readonly. Hence all metadata changes are updated to the file in overlay FS.

Figure “Metadata Update: chmod” gives a nice flow of control in case of chmod command.

Step 0: Lookup is done and hence dentry is pointing to underlay FS.

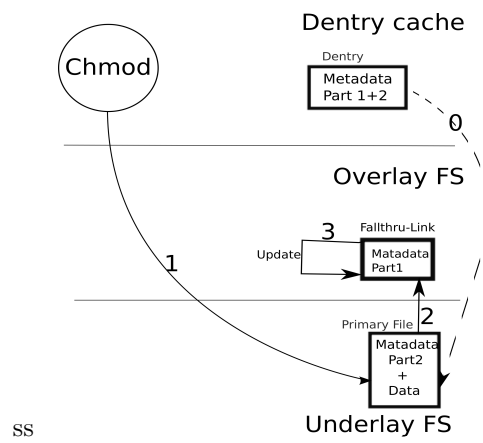
Step 1: chmod uses dentry in cache and comes to file in underlay FS.

Step 2: Follow union mount to get the corresponding fallthru link on overlay FS.

Step 3: If the fallthru link doesn't exist, then fallthru link is created. Then chmod permissions are updated to the fallthru link file.

Because metadata changes are done to file on overlay FS, we need to have an updated inode on the overlay at all times.

Hence we copyup some of the necessary metadata information from file on underlay FS to newly created file(Fallthru-link) on overlay FS. This is done at the creation time of fallthru-link.



ss

Figure F.2: Metadata Update: chmod

Appendix G

Challenges in our approach and how we handle them

- Because of very less documentation it's very difficult to understand the code flow. It took a long time to understand the code flow.
- Though it was suggested by a few people that fallthru can be implemented as a symbolic link, it was not clear as to what exactly is the change needed to fallthru.
- Provide a consistent view of dcache. In the original Overlay Mount implementation, fallthrus didn't have any corresponding inode on overlay FS. But in our approach of fallthru-link, we maintain a valid inode for every fallthru-link.
So files on both overlay and underlay FS have valid inode and hence it's very important to provide a consistent view of cache. Dentry should always point to file on underlay FS with merged inode information.
- It's not an easy task to separate data and metadata information of a file. To add to complexity, we also have part of metadata on underlay FS and part of metadata on overlay FS. So giving a merged view of these 2 inodes and maintaining them consistently is a major challenge.
- Updating metadata of a file and storing it on a different file system somewhere else is a tough task. Lookup will always give merged view, but metadata operations need inode of the overlay FS only. Hence we need to follow union mount to overlay FS and then update changes to that inode. Again cache should be maintained consistently to point to file in underlay FS.

Appendix H

Future work

As described in section Goals, we split this project into several stages. We have so far completed first 2 stages which form the backbone of the rest of the stages. Due to lack of time, we are unable to complete rest of the stages in this assignment. We would love to continue working on other stages. We will be more than happy if somebody wants to continue our work from here. To handle renaming directories: There are some pros and cons:

pros:

- Recursive file copyup is deferred. It's deferred until there is some data change to those files. Until then, they are marked as fallthru.
- This approach gives a method of making rename operation work on Overlay mounts.

cons:

- Recursion in kernel is a BIG NO! This new idea can run into recursion of any depth depending on how many directories are already present on the overlay FS before issuing a rename operation on the directory. We need to recursively continue creating fallthrus for all child files in a directory, because it's not possible to store the original name of the file anywhere in the directory file on overlay FS. An alternate design consideration could be to store all these rename operations in a separate file(per mount or per directory) and then apply it to future lookups. Anyway, that's loads of work.

Appendix I

Performance results

We have given a prototype of the final working fallthru-link implementation. This prototype works only for chmod metadata update. Any other metadata updates, will return with an error saying update not possible on a read-only FS. Hence this project in this current state cannot be tested with standard test suites like LTP. We will show here some of the results that we can see on a typical data/metadata change and hence we show that this design works!

I.1 Test Scenarios

To test the functionality we have to mount the required device first and then use them as Overlay Fs.

1. Mount the devices.

```
cd /root/union_mount
mount -o loop,ro ./img/ro mnt/union/
cp img/rw_srcimg/rw
mount--o loop, noatime, union/root/union_mount/img/rw/root/union_mount/mnt/union
```

2. Do a ls to create the fallthru-link

```
ls mnt/union/test_dir
Now we have the fallthru-link for following file and folder:
mnt/union/test_dir/test_file
mnt/union/test_dir/test_subdir
```

3. We can use chmod on *test_file*

```
chmod 777 mnt/union/test_dir/test_file
```

After this chmod command, permission metadata of *test_file* is stored in overlay FS (Ext2) and the remaining metadata inode information is on the underlay FS.

4. Use cat to make sure dentry is in a consistent position and points to the corresponding underlay file.

```
cat mnt/union/test_dir/test_file
```

5. Use `ls -il` to make sure the permission is changes successfully.
`ls -il mnt/union/test_dir/test_file`

Appendix J

Conclusion

This is a very challenging project and we were able to learn many new things like:
Good understanding of file system internals, how to use User Mode Linux(UML),
util-linux.

Bibliography

- [1] Wolfgang Mauerer, *Professional Linux kernel Architecture*, Wiley publishing Inc, 2008.
- [2] Valerie Aurora, State of writable overlays (formerly union mounts), <http://valerieaurora.org/union/design.txt>
- [3] Overlay writable filesystem over r/o partition <http://lkml.indiana.edu/hypermail/linux/kernel/9801.0/0132.html>
- [4] Read/Write Compression: Combining UnionFS and SquashFS <http://www.linux-mag.com/cache/7409/1.html>
- [5] Kernel Korner - Unionfs: Bringing Filesystems Together <http://www.linuxjournal.com/article/7714>
- [6] [RFC] Union mount readdir support in glibc <http://sources.redhat.com/ml/libc-alpha/2008-03/msg00032.html>
- [7] Union file systems: Implementations, part I <http://lwn.net/Articles/325369/>
- [8] Unioning file systems: Implementations, part 2 <http://lwn.net/Articles/327738/>
- [9] Unioning file systems: Architecture, features, and design choices <http://lwn.net/Articles/324291/>