Written by: Neil Brown Please see MAINTAINERS file for where to send questions.

Overlay Filesystem

This document describes a prototype for a new approach to providing overlay-filesystem functionality in Linux (sometimes referred to as union-filesystems). An overlay-filesystem tries to present a filesystem which is the result over overlaying one filesystem on top of the other.

Overlay objects

The overlay filesystem approach is 'hybrid', because the objects that appear in the filesystem do not always appear to belong to that filesystem. In many cases, an object accessed in the union will be indistinguishable from accessing the corresponding object from the original filesystem. This is most obvious from the 'st_dev' field returned by stat(2).

While directories will report an st_dev from the overlay-filesystem, non-directory objects may report an st_dev from the lower filesystem or upper filesystem that is providing the object. Similarly st_ino will only be unique when combined with st_dev, and both of these can change over the lifetime of a non-directory object. Many applications and tools ignore these values and will not be affected.

In the special case of all overlay layers on the same underlying filesystem, all objects will report an st_dev from the overlay filesystem and st_ino from the underlying filesystem. This will make the overlay mount more compliant with filesystem scanners and overlay objects will be distinguishable from the corresponding objects in the original filesystem.

On 64bit systems, even if all overlay layers are not on the same underlying filesystem, the same compliant behavior could be achieved with the "xino" feature. The "xino" feature composes a unique object identifier from the real object st_ino and an underlying fsid index. The "xino" feature uses the high inode number bits for fsid, because the underlying filesystems rarely use the high inode number bits. In case the underlying inode number does overflow into the high xino bits, overlay filesystem will fall back to the non xino behavior for that inode.

The "xino" feature can be enabled with the "-o xino=on" overlay mount option. If all underlying filesystems support NFS file handles, the value of st_ino for overlay filesystem objects is not only unique, but also persistent over the lifetime of the filesystem. The "-o xino=auto" overlay mount option enables the "xino" feature only if the persistent st_ino requirement is met.

The following table summarizes what can be expected in different overlay configurations.

Inode properties

Configuration	**Persistent st_ino**		**Uniform st_dev**		**st_ino == d_ino**		**d_ino == i_ino [*]**	
	dir	!dir	dir	!dir	dir	!dir	dir	!dir
All layers on same fs	Y	Y	Y	Y	Y	Y	Y	Y
Layers not on same fs, xino=off	N	N	Y	N	N	Y	N	Y
xino=on/auto	Y	Y	Y	Y	Y	Y	Y	Y
xino=on/auto, ino overflow	N	N	Y	N	N	Y	N	Y

[*] nfsd v3 readdirplus verifies d_ino == i_ino. i_ino is exposed via several /proc files, such as /proc/locks and /proc/self/fdinfo/<fd> of an inotify file descriptor.

Upper and Lower

An overlay filesystem combines two filesystems - an 'upper' filesystem and a 'lower' filesystem. When a name exists in both filesystems, the object in the 'upper' filesystem is visible while the object in the 'lower' filesystem is either hidden or, in the case of directories, merged with the 'upper' object.

It would be more correct to refer to an upper and lower 'directory tree' rather than 'filesystem' as it is quite possible for both directory trees to be in the same filesystem and there is no requirement that the root of a filesystem be given for either upper or lower.

A wide range of filesystems supported by Linux can be the lower filesystem, but not all filesystems that are mountable by Linux have the features needed for OverlayFS to work. The lower filesystem does not need to be writable. The lower filesystem can even be another overlayfs. The upper filesystem will normally be writable and if it is it must support the creation of trusted.* and/or user.* extended attributes, and must provide valid d_type in readdir responses, so NFS is not suitable.

A read-only overlay of two read-only filesystems may use any filesystem type.

Directories

Overlaying mainly involves directories. If a given name appears in both upper and lower filesystems and refers to a non-directory in either, then the lower object is hidden - the name refers only to the upper object.

Where both upper and lower objects are directories, a merged directory is formed.

At mount time, the two directories given as mount options "lowerdir" and "upperdir" are combined into a merged directory:

```
mount -t overlay overlay -olowerdir=/lower,upperdir=/upper,workdir=/work /merged
```

The "workdir" needs to be an empty directory on the same filesystem as upperdir.

Then whenever a lookup is requested in such a merged directory, the lookup is performed in each actual directory and the combined result is cached in the dentry belonging to the overlay filesystem. If both actual lookups find directories, both are stored and a merged directory is created, otherwise only one is stored: the upper if it exists, else the lower.

Only the lists of names from directories are merged. Other content such as metadata and extended attributes are reported for the upper directory only. These attributes of the lower directory are hidden.

whiteouts and opaque directories

In order to support rm and rmdir without changing the lower filesystem, an overlay filesystem needs to record in the upper filesystem that files have been removed. This is done using whiteouts and opaque directories (non-directories are always opaque).

A whiteout is created as a character device with 0/0 device number. When a whiteout is found in the upper level of a merged directory, any matching name in the lower level is ignored, and the whiteout itself is also hidden.

A directory is made opaque by setting the xattr "trusted.overlay.opaque" to "y". Where the upper filesystem contains an opaque directory, any directory in the lower filesystem with the same name is ignored.

readdir

When a 'readdir' request is made on a merged directory, the upper and lower directories are each read and the name lists merged in the obvious way (upper is read first, then lower - entries that already exist are not re-added). This merged name list is cached in the 'struct file' and so remains as long as the file is kept open. If the directory is opened and read by two processes at the same time, they will each have separate caches. A seekdir to the start of the directory (offset 0) followed by a readdir will cause the cache to be discarded and rebuilt.

This means that changes to the merged directory do not appear while a directory is being read. This is unlikely to be noticed by many programs.

seek offsets are assigned sequentially when the directories are read. Thus if

- read part of a directory
- remember an offset, and close the directory
- re-open the directory some time later
- seek to the remembered offset

there may be little correlation between the old and new locations in the list of filenames, particularly if anything has changed in the directory.

Readdir on directories that are not merged is simply handled by the underlying directory (upper or lower).

renaming directories

When renaming a directory that is on the lower layer or merged (i.e. the directory was not created on the upper layer to start with) overlayfs can handle it in two different ways:

1. return EXDEV error: this error is returned by rename(2) when trying to move a file or directory across filesystem boundaries. Hence applications are usually prepared to hande this error (mv(1) for example recursively copies the directory tree). This is the default behavior.
2. If the "redirect_dir" feature is enabled, then the directory will be copied up (but not the contents). Then the "trusted.overlay.redirect" extended attribute is set to the path of the original location from the root of the overlay. Finally the directory is moved to the new location.

There are several ways to tune the "redirect_dir" feature.

Kernel config options:

- OVERLAY_FS_REDIRECT_DIR:
  If this is enabled, then redirect_dir is turned on by default.
- OVERLAY_FS_REDIRECT_ALWAYS_FOLLOW:
  If this is enabled, then redirects are always followed by default. Enabling this results in a less secure configuration. Enable this option only when worried about backward compatibility with kernels that have the redirect_dir feature and follow redirects even if turned off.

Module options (can also be changed through /sys/module/overlay/parameters/):

- "redirect_dir=BOOL":
  See OVERLAY_FS_REDIRECT_DIR kernel config option above.
- "redirect_always_follow=BOOL":
  See OVERLAY_FS_REDIRECT_ALWAYS_FOLLOW kernel config option above.

- "redirect_max=NUM":
  The maximum number of bytes in an absolute redirect (default is 256).

Mount options:

- "redirect_dir=on":
  Redirects are enabled.
- "redirect_dir=follow":
  Redirects are not created, but followed.
- "redirect_dir=off":
  Redirects are not created and only followed if "redirect_always_follow" feature is enabled in the kernel/module config.
- "redirect_dir=nofollow":
  Redirects are not created and not followed (equivalent to "redirect_dir=off" if "redirect_always_follow" feature is not enabled).

When the NFS export feature is enabled, every copied up directory is indexed by the file handle of the lower inode and a file handle of the upper directory is stored in a "trusted.overlay.upper" extended attribute on the index entry. On lookup of a merged directory, if the upper directory does not match the file handle stores in the index, that is an indication that multiple upper directories may be redirected to the same lower directory. In that case, lookup returns an error and warns about a possible inconsistency.

Because lower layer redirects cannot be verified with the index, enabling NFS export support on an overlay filesystem with no upper layer requires turning off redirect follow (e.g. "redirect_dir=nofollow").

Non-directories

Objects that are not directories (files, symlinks, device-special files etc.) are presented either from the upper or lower filesystem as appropriate. When a file in the lower filesystem is accessed in a way the requires write-access, such as opening for write access, changing some metadata etc., the file is first copied from the lower filesystem to the upper filesystem (copy_up). Note that creating a hard-link also requires copy_up, though of course creation of a symlink does not.

The copy_up may turn out to be unnecessary, for example if the file is opened for read-write but the data is not modified.

The copy_up process first makes sure that the containing directory exists in the upper filesystem - creating it and any parents as necessary. It then creates the object with the same metadata (owner, mode, mtime, symlink-target etc.) and then if the object is a file, the data is copied from the lower to the upper filesystem. Finally any extended attributes are copied up.

Once the copy_up is complete, the overlay filesystem simply provides direct access to the newly created file in the upper filesystem - future operations on the file are barely noticed by the overlay filesystem (though an operation on the name of the file such as rename or unlink will of course be noticed and handled).

Permission model

Permission checking in the overlay filesystem follows these principles:

1. permission check SHOULD return the same result before and after copy up
2. task creating the overlay mount MUST NOT gain additional privileges

3) non-mounting task MAY gain additional privileges through the overlay, compared to direct access on underlying lower or upper filesystems

This is achieved by performing two permission checks on each access

a. check if current task is allowed access based on local DAC (owner, group, mode and posix acl), as well as MAC checks
b. check if mounting task would be allowed real operation on lower or upper layer based on underlying filesystem permissions, again including MAC checks

Check (a) ensures consistency (1) since owner, group, mode and posix acls are copied up. On the other hand it can result in server enforced permissions (used by NFS, for example) being ignored (3).

Check (b) ensures that no task gains permissions to underlying layers that the mounting task does not have (2). This also means that it is possible to create setups where the consistency rule (1) does not hold; normally, however, the mounting task will have sufficient privileges to perform all operations.

Another way to demonstrate this model is drawing parallels between

```
mount -t overlay overlay -olowerdir=/lower,upperdir=/upper,... /merged
```

and

```
cp -a /lower /upper mount --bind /upper /merged
```

The resulting access permissions should be the same. The difference is in the time of copy (on-demand vs. up-front).

Multiple lower layers

Multiple lower layers can now be given using the colon (":") as a separator character between the directory names. For example:

```
mount -t overlay overlay -olowerdir=/lower1:/lower2:/lower3 /merged
```

As the example shows, "upperdir=" and "workdir=" may be omitted. In that case the overlay will be read-only.

The specified lower directories will be stacked beginning from the rightmost one and going left. In the above example lower1 will be the top, lower2 the middle and lower3 the bottom layer.

Metadata only copy up

When metadata only copy up feature is enabled, overlayfs will only copy up metadata (as opposed to whole file), when a metadata specific operation like chown/chmod is performed. Full file will be copied up later when file is opened for WRITE operation.

In other words, this is delayed data copy up operation and data is copied up when there is a need to actually modify data.

There are multiple ways to enable/disable this feature. A config option CONFIG_OVERLAY_FS_METACOPY can be set/unset to enable/disable this feature by default. Or one can enable/disable it at module load time with module parameter metacopy=on/off. Lastly, there is also a per mount option metacopy=on/off to enable/disable this feature per mount.

Do not use metacopy=on with untrusted upper/lower directories. Otherwise it is possible that an attacker can create a handcrafted file with appropriate REDIRECT and METACOPY xattrs, and gain access to file on lower pointed by REDIRECT. This should not be possible on local system as setting "trusted." xattrs will require CAP_SYS_ADMIN. But it should be possible for untrusted layers like from a pen drive.

Note: redirect_dir={off|nofollow|follow[*]} and nfs_export=on mount options conflict with metacopy=on, and will result in an error.

[*] redirect_dir=follow only conflicts with metacopy=on if upperdir=... is given.

Sharing and copying layers

Lower layers may be shared among several overlay mounts and that is indeed a very common practice. An overlay mount may use the same lower layer path as another overlay mount and it may use a lower layer path that is beneath or above the path of another overlay lower layer path.

Using an upper layer path and/or a workdir path that are already used by another overlay mount is not allowed and may fail with EBUSY. Using partially overlapping paths is not allowed and may fail with EBUSY. If files are accessed from two overlayfs mounts which share or overlap the upper layer and/or workdir path the behavior of the overlay is undefined, though it will not result in a crash or deadlock.

Mounting an overlay using an upper layer path, where the upper layer path was previously used by another mounted overlay in combination with a different lower layer path, is allowed, unless the "inodes index" feature or "metadata only copy up" feature is enabled.

With the "inodes index" feature, on the first time mount, an NFS file handle of the lower layer root directory, along with the UUID of the lower filesystem, are encoded and stored in the "trusted.overlay.origin" extended attribute on the upper layer root directory. On subsequent mount attempts, the lower root directory file handle and lower filesystem UUID are compared to the stored origin in upper root directory. On failure to verify the lower root origin, mount will fail with ESTALE. An overlayfs mount with "inodes index" enabled will fail with EOPNOTSUPP if the lower filesystem does not support NFS export, lower filesystem does not have a valid UUID or if the upper filesystem does not support extended attributes.

For "metadata only copy up" feature there is no verification mechanism at mount time. So if same upper is mounted with different set of lower, mount probably will succeed but expect the unexpected later on. So don't do it.

It is quite a common practice to copy overlay layers to a different directory tree on the same or different underlying filesystem, and even to a different machine. With the "inodes index" feature, trying to mount the copied layers will fail the verification of the lower root file handle.

Non-standard behavior

Current version of overlayfs can act as a mostly POSIX compliant filesystem.

This is the list of cases that overlayfs doesn't currently handle:

a) POSIX mandates updating st_atime for reads. This is currently not done in the case when the file resides on a lower layer.

b) If a file residing on a lower layer is opened for read-only and then memory mapped with MAP_SHARED, then subsequent changes to the file are not reflected in the memory mapping.

c) If a file residing on a lower layer is being executed, then opening that file for write or truncating the file will not be denied with ETXTBSY.

The following options allow overlayfs to act more like a standards compliant filesystem:

1. "redirect_dir"

Enabled with the mount option or module option: "redirect_dir=on" or with the kernel config option CONFIG_OVERLAY_FS_REDIRECT_DIR=y.

If this feature is disabled, then rename(2) on a lower or merged directory will fail with EXDEV ("Invalid cross-device link").

2. "inode index"

Enabled with the mount option or module option "index=on" or with the kernel config option CONFIG_OVERLAY_FS_INDEX=y.

If this feature is disabled and a file with multiple hard links is copied up, then this will "break" the link. Changes will not be propagated to other names referring to the same inode.

3. "xino"

Enabled with the mount option "xino=auto" or "xino=on", with the module option "xino_auto=on" or with the kernel config option CONFIG_OVERLAY_FS_XINO_AUTO=y. Also implicitly enabled by using the same underlying filesystem for all layers making up the overlay.

If this feature is disabled or the underlying filesystem doesn't have enough free bits in the inode number, then overlayfs will not be able to guarantee that the values of st_ino and st_dev returned by stat(2) and the value of d_ino returned by readdir(3) will act like on a normal filesystem. E.g. the value of st_dev may be different for two objects in the same overlay filesystem and the value of st_ino for filesystem objects may not be persistent and could change even while the overlay filesystem is mounted, as summarized in the Inode properties table above.

Changes to underlying filesystems

Changes to the underlying filesystems while part of a mounted overlay filesystem are not allowed. If the underlying filesystem is changed, the behavior of the overlay is undefined, though it will not result in a crash or deadlock.

Offline changes, when the overlay is not mounted, are allowed to the upper tree. Offline changes to the lower tree are only allowed if the "metadata only copy up", "inode index", "xino" and "redirect_dir" features have not been used. If the lower tree is modified and any of these features has been used, the behavior of the overlay is undefined, though it will not result in a crash or deadlock.

When the overlay NFS export feature is enabled, overlay filesystems behavior on offline changes of the underlying lower layer is different than the behavior when NFS export is disabled.

On every copy_up, an NFS file handle of the lower inode, along with the UUID of the lower filesystem, are encoded and stored in an extended attribute "trusted.overlay.origin" on the upper inode.

When the NFS export feature is enabled, a lookup of a merged directory, that found a lower directory at the lookup path or at the path pointed to by the "trusted.overlay.redirect" extended attribute, will verify that the found lower directory file handle and lower filesystem UUID match the origin file handle that was stored at copy_up time. If a found lower directory does not match the stored origin, that directory will not be merged with the upper directory.

NFS export

When the underlying filesystems supports NFS export and the "nfs_export" feature is enabled, an overlay filesystem may be exported to NFS.

With the "nfs_export" feature, on copy_up of any lower object, an index entry is created under the index directory. The index entry name is the hexadecimal representation of the copy up origin file handle. For a non-directory object, the index entry is a hard link to the upper inode. For a directory object, the index entry has an extended attribute "trusted.overlay.upper" with an encoded file handle of the upper directory inode.

When encoding a file handle from an overlay filesystem object, the following rules apply:

1. For a non-upper object, encode a lower file handle from lower inode
2. For an indexed object, encode a lower file handle from copy_up origin
3. For a pure-upper object and for an existing non-indexed upper object, encode an upper file handle from upper inode

The encoded overlay file handle includes:

- Header including path type information (e.g. lower/upper)
- UUID of the underlying filesystem
- Underlying filesystem encoding of underlying inode

This encoding format is identical to the encoding format file handles that are stored in extended attribute "trusted.overlay.origin".

When decoding an overlay file handle, the following steps are followed:

1. Find underlying layer by UUID and path type information.
2. Decode the underlying filesystem file handle to underlying dentry.
3. For a lower file handle, lookup the handle in index directory by name.
4. If a whiteout is found in index, return ESTALE. This represents an overlay object that was deleted after its file handle was encoded.
5. For a non-directory, instantiate a disconnected overlay dentry from the decoded underlying dentry, the path type and index

inode, if found.

6. For a directory, use the connected underlying decoded dentry, path type and index, to lookup a connected overlay dentry.

Decoding a non-directory file handle may return a disconnected dentry. copy_up of that disconnected dentry will create an upper index entry with no upper alias.

When overlay filesystem has multiple lower layers, a middle layer directory may have a "redirect" to lower directory. Because middle layer "redirects" are not indexed, a lower file handle that was encoded from the "redirect" origin directory, cannot be used to find the middle or upper layer directory. Similarly, a lower file handle that was encoded from a descendant of the "redirect" origin directory, cannot be used to reconstruct a connected overlay path. To mitigate the cases of directories that cannot be decoded from a lower file handle, these directories are copied up on encode and encoded as an upper file handle. On an overlay filesystem with no upper layer this mitigation cannot be used NFS export in this setup requires turning off redirect follow (e.g. "redirect_dir=nofollow").

The overlay filesystem does not support non-directory connectable file handles, so exporting with the 'subtree_check' exportfs configuration will cause failures to lookup files over NFS.

When the NFS export feature is enabled, all directory index entries are verified on mount time to check that upper file handles are not stale. This verification may cause significant overhead in some cases.

Note: the mount options index=off,nfs_export=on are conflicting for a read-write mount and will result in an error.

Note: the mount option uuid=off can be used to replace UUID of the underlying filesystem in file handles with null, and effectively disable UUID checks. This can be useful in case the underlying disk is copied and the UUID of this copy is changed. This is only applicable if all lower/upper/work directories are on the same filesystem, otherwise it will fallback to normal behaviour.

Volatile mount

This is enabled with the "volatile" mount option. Volatile mounts are not guaranteed to survive a crash. It is strongly recommended that volatile mounts are only used if data written to the overlay can be recreated without significant effort.

The advantage of mounting with the "volatile" option is that all forms of sync calls to the upper filesystem are omitted.

In order to avoid a giving a false sense of safety, the syncfs (and fsync) semantics of volatile mounts are slightly different than that of the rest of VFS. If any writeback error occurs on the upperdir's filesystem after a volatile mount takes place, all sync functions will return an error. Once this condition is reached, the filesystem will not recover, and every subsequent sync call will return an error, even if the upperdir has not experience a new error since the last sync call.

When overlay is mounted with "volatile" option, the directory "$workdir/work/incompat/volatile" is created. During next mount, overlay checks for this directory and refuses to mount if present. This is a strong indicator that user should throw away upper and work directories and create fresh one. In very limited cases where the user knows that the system has not crashed and contents of upperdir are intact, The "volatile" directory can be removed.

User xattr

The the "-o userxattr" mount option forces overlayfs to use the "user.overlay." xattr namespace instead of "trusted.overlay.". This is useful for unprivileged mounting of overlayfs.

Testsuite

There's a testsuite originally developed by David Howells and currently maintained by Amir Goldstein at:

https://github.com/amir73il/unionmount-testsuite.git

Run as root:

```
# cd unionmount-testsuite # ./run --ov --verify
```