LWN.net Linux info from the source

Weekly edition Kernel Security Distributions Contact Us Search
Archives Calendar Subscribe Write for LWN LWN.net FAQ Sponsors

Not logged in
Log in now
Create an account
Subscribe to LWN

Weekly Edition
Return to the Kernel page

Recent Features
LWN.net Weekly Edition for July 16, 2015
Python 3.5 is on its way
Why Debian returned to FFmpeg
LWN.net Weekly Edition for July 9, 2015
A preview of PostgreSQL 9.5

Printable page

Union file systems: Implementations, part I

March 25, 2009
This article was contributed by Valerie Aurora (formerly Henson)

In last week's article, I reviewed the use cases, basic concepts, and common design problems of unioning file systems. This week, I'll describe several implementations of unioning file systems in technical detail. The unioning file systems I'll cover in this article are Plan 9 union directories, BSD union mounts, Linux union mounts. The next article will cover unionfs, aufs, and possibly one or two other unioning file systems, and wrap up the series.

For each file system, I'll describe its basic architecture, features, and implementation. The discussion of the implementation will focus in particular on whiteouts and directory reading. I'll wrap up with a look at the software engineering aspects of each implementations; e.g., code size and complexity, invasiveness, and burden on file system developers.

Before reading this article, you might want to check out Andreas Gruenbacher's just published write-up of the union mount workshop held last November. It's a good summary of the unioning file systems features which are most pressing for distribution developers. From the introduction: "All of the use cases we are interested in basically boil down to the same thing: having an image or filesystem that is used read-only (either because it is not writable, or because writing to the image is not desired), and pretending that this image or filesystem is writable, storing changes somewhere else."

Plan 9 union directories

The Plan 9 operating system (browseable source code here) implements unioning in its own special Plan 9 way. In Plan 9 union directories, only the top-level directory namespace is merged, not any subdirectories. Unconstrained by UNIX standards, Plan 9 union directories don't implement whiteouts and don't even screen out duplicate entries - if the same file name appears in two file systems, it is simply returned twice in directory listings.

A Plan 9 union directory is created like so:

```
    bind -a /home/val/bin/ /bin
```

This would cause the directory `/home/val/bin` to be union mounted "after" (the `-a` option) `/bin`; other options are to place the new directory before the existing directory, or to replace the existing directory entirely. (This seems an odd ordering to me, since I like commands in my personal `bin/` to take precedence over the system-wide commands, but that's the example from the Plan 9 documentation.) Brian Kernighan explains one of the uses of union directories: "This mechanism of union directories replaces the search path of conventional UNIX shells. As far as you are concerned, all executable programs are in /bin." Union directories can theoretically replace many uses of the fundamental UNIX building blocks of symbolic links and search paths.

Without whiteouts or duplicate elimination, `readdir()` on union directories is trivial to implement. Directory entry offsets from the underlying file system correspond directly to the offset in bytes of the directory entry from the beginning of the directory. A union directory is treated as though the contents of the underlying directories are concatenated together.

Plan 9 implements an alternative to `readdir()` worth noting, `dirread()`. `dirread()` returns structures of type `Dir`, described in the `stat()` man page. The important part of the `Dir` is the `Qid` member. A `Qid` is:

> ...a structure containing `path` and `vers` fields: `path` is guaranteed to be unique among all path names currently on the file server, and `vers` changes each time the file is modified. The `path` is a long long (64 bits, `vlong`) and the `vers` is an unsigned long (32 bits, `ulong`).

So why is this interesting? One of the reasons `readdir()` is such a pain to implement is that it returns the `d_off` member of `struct dirent`, a single `off_t` (32 bits unless the application is compiled with large file support), to mark the directory entry where an application should continue reading on the next `readdir()` call. This works fine as long as `d_off` is a simple byte offset into a flat file of less than 2^{32} bytes and existing directory entries are never moved around - not the case for many modern file systems (XFS, btrfs, ext3 with htree indexes). The 96-bit `Qid` is a much more useful place marker than the 32 or 64-bit off_t. For a good summary of the issues involved in implementing `readdir()`, read Theodore Y. Ts'o's excellent post on the topic to the btrfs mailing list.

From a software engineering standpoint, Plan 9 union directories are heavenly. Without whiteouts, duplicate entry elimination, complicated directory offsets, or merging of namespaces beyond the top-level directory, the implementation is simple and easy to maintain. However, any practical implementation of unioning file systems for Linux (or any other UNIX) would have to solve these problems. For our purposes, Plan 9 union directories serve primarily as inspiration.

BSD union mounts

BSD implements two forms of unioning: the "`-o union`" option to the `mount` command, which produces a union directory similar to Plan 9's, and the `mount_unionfs` command, which implements a more full-featured unioning file system with whiteouts and merging of the entire namespace. We will focus on the latter.

For this article, we use two sources for specific implementation details: the original BSD union mount implementation as described in the 1995 USENIX paper Union mounts in 4.4BSD-Lite [PS], and the FreeBSD 7.1 `mount_unionfs` man page and source code. Other BSDs may vary.

A directory can be union mounted either "below" or "above" an existing directory or union mount, as long as the top branch of a writable union is writable. Two modes of whiteouts are supported: either a whiteout is always created when a directory is removed, or it is only created if another directory entry with that name currently exists in a branch below the writable branch. Three modes for setting the ownership and mode of copied-up files are supported. The simplest is `transparent`, in which the new file keeps the same owner and mode of the original. The `masquerade` mode makes copied-up files owned by a particular user and supports a set of mount options for determining the new file mode. The `traditional` mode sets the owner to the user who ran the union mount command, and sets the mode according to the umask at the time of the union mount.

Whenever a directory is opened, a directory of the same name is created on the top writable layer if it doesn't already exist. From the paper:

> By creating shadow directories aggressively during lookup the union filesystem avoids having to check for and possibly create the chain of directories from the root of the mount to the point of a copy-up. Since the disk space consumed by a directory is negligible, creating directories when they were first

traversed seemed like a better alternative.

As a result, a "`find /union`" will result in copying every directory (but not directory entries pointing to non-directories) to the writable layer. For most file system images, this will use a negligible amount of space (less than, e.g., the space reserved for the root user, or that taken up by unused inodes in an FFS-style file system).

A file is copied up to the top layer when it is opened with write permission or the file attributes are changed. (Since directories are copied over when they are opened, the containing directory is guaranteed to already exist on the writable layer.) If the file to be copied up has multiple hard links, the other links are ignored and the new file has a link count of one. This may break applications that use hard links and expect modifications through one link name to show up when referenced through a different hard link. Such applications are relatively uncommon, but no one has done a systematic study to see which applications will fail in this situation.

Whiteouts are implemented with a special directory entry type, `DH_WHT`. Whiteout directory entries don't refer to any real inode, but for easy compatibility with existing file system utilities such as `fsck`, each whiteout directory entry includes a faux inode number, the `WINO` reserved whiteout inode number. The underlying file system must be modified to support the whiteout directory entry type. New directories that replace a whiteout entry are marked as opaque via a new "opaque" inode attribute so that lookups don't travel through them (again requiring minimal support from the underlying file system).

Duplicate directory entries and whiteouts are handled in the userspace `readdir()` implementation. At `opendir()` time, the C library reads the directory all at once, removes duplicates, applies whiteouts, and caches the results.

BSD union mounts don't attempt to deal with changes to branches below the writable top branch (although they are permitted). The way `rename()` is handled is not described.

An example from the `mount_unionfs` man page:

```
The commands

    mount -t cd9660 -o ro /dev/cd0 /usr/src
    mount -t unionfs -o noatime /var/obj /usr/src

mount the CD-ROM drive /dev/cd0 on /usr/src and then attaches /var/obj on
top.  For most purposes the effect of this is to make the source tree
appear writable even though it is stored on a CD-ROM.  The -o noatime
option is useful to avoid unnecessary copying from the lower to the upper
layer.
```

Another example (noting that I believe source control is best implemented outside of the file system):

```
The command

    mount -t unionfs -o noatime -o below /sys $HOME/sys

attaches the system source tree below the sys directory in the user's
home directory.  This allows individual users to make private changes to
the source, and build new kernels, without those changes becoming visible
to other users.
```

Linux union mounts

Like BSD union mounts, Linux union mounts implement file system unioning in the VFS layer, with some minor support from underlying file systems for whiteouts and opaque directory tags. Several versions of these patches exist, written and modified by Jan Blunck, Bharata B. Rao, and Miklos Szeredi, among others.

One version of this code is merges the top-level directories only, similar to Plan 9 union directories and the BSD `-o union` mount option. This version of union mounts, which I refer to as union directories, are described in some detail in a recent LWN article by Goldwyn Rodrigues and in Miklos Szeredi's recent post of an updated patch set. For the remainder of this article, we will focus on versions of union mount that merge the full namespace.

Linux union mounts are currently under active development. This article describes the version released by Jan Blunck against Linux 2.6.25-mm1, util-linux 2.13, and e2fsprogs 1.40.2. The patch sets, as quilt series, can be downloaded from Jan's ftp site:

> Kernel patches:
> ftp://ftp.suse.com/pub/people/jblunck/patches/
>
> Utilities:
> ftp://ftp.suse.com/pub/people/jblunck/union-mount/

I have created a web page with links to git versions of the above patches and some HOWTO-style documentation at http://valerieaurora.org/union.

A union is created by mounting a file system with the `MS_UNION` flag set. (The `MS_BEFORE`, `MS_AFTER`, and `MS_REPLACE` are defined in the `mount` code base but not currently used.) If the `MS_UNION` flag is specified, then the mounted file system must either be read-only or support whiteouts. In this version of union mounts, the union mount flag is specified by the "`-o union`" option to `mount`. For example, to create a union of two loopback device file systems, /img/ro and /img/rw, you would run:

```
# mount -o loop,ro,union /img/ro /mnt/union/
# mount -o loop,union /img/rw /mnt/union/
```

Each union mount creates a `struct union_mount`:

```
struct union_mount {
    atomic_t u_count;              /* reference count */
    struct mutex u_mutex;
    struct list_head u_unions;     /* list head for d_unions */
    struct hlist_node u_hash;      /* list head for searching */
    struct hlist_node u_rhash;     /* list head for reverse searching */
    struct path u_this;            /* this is me */
    struct path u_next;            /* this is what I overlay */
};
```

As described in `Documentation/filesystems/union-mounts.txt`, "All union_mount structures are cached in two hash tables, one for lookups of the next lower layer of the union stack and one for reverse lookups of the next upper layer of the union stack."

Whiteouts and opaque directories are implemented in much the same way as in BSD. The underlying file system must explicitly support whiteouts by defining the `.whiteout` inode operation for directories (currently, whiteouts are only implemented for ext2, ext3, and tmpfs). The ext2 and ext3 implementations use the whiteout directory entry type, `DT_WHT`, which has been defined in `include/linux/fs.h` for years but not used outside of the Coda file system until now. A reserved whiteout inode number, `EXT3_WHT_INO`, is defined but not yet used; whiteout entries currently allocate a normal inode. A new inode flag, `S_OPAQUE`, is defined to mark directories as opaque. As in BSD, directories are only marked opaque when they replace a whiteout entry.

Files are copied up when the file is opened for writing. If necessary, each directory in the path to the file is copied to the top branch (copy-on-demand of directories). Currently, copy up is only supported for regular files and directories.

`readdir()` is one of the weakest points of the current implementation. It is implemented the same way as BSD union mount `readdir()`, but in the kernel. The `d_off` field is set to the offset within the current underlying directory, minus the sizes of the previous directories. Directory entries from directories underneath the top layer must be checked against previous entries for duplicates or whiteouts. As currently implemented, each `readdir()` (technically, `getdents()`) system call reads all of the previous directory entries into an in-kernel cache, then compares each entry to be returned with those already in the cache before copying it to the user buffer. The end result is that `readdir()` is complex, slow, and potentially allocates a great deal of kernel memory.

One solution is to take the BSD approach and do the caching, whiteout, and duplicate processing in userspace. Bharata B. Rao is designing support for union mount `readdir()` in glibc. (The POSIX standard permits `readdir()` to be implemented at the libc level if the bare kernel system call does not fulfill all the requirements.) This would move the memory usage into the application and make the cache persistent. Another solution would be to make the in-kernel cache persistent in some way.

My suggestion is to take a technique from BSD union mounts and extend it: proactively copy up not just directory entries for directories, but all of the directory entries from lower file systems, process duplicates and whiteouts, make the directory opaque, and write it out to disk. In effect, you are processing the directory entries for whiteouts and duplicates on the first open of the directory, and then writing the resulting "cache" of directory entries to disk. The directory entries pointing to files on the underlying file systems need to signify somehow that they are "fall-through" entries (the opposite of a whiteout - it explicitly requests looking up an object in a lower file system). A side effect of this approach is that whiteouts are no longer needed at all.

One problem that needs to be solved with this approach is how to represent directory entries pointing to lower file systems. A number of solutions present themselves: the entry could point to a reserved inode number, the file system could allocate an inode for each entry but mark it with a new `S_LOOKOVERTHERE` inode attribute, it could create a symlink to a reserved target, etc. This approach would use more space on the overlying file system, but all other approaches require allocating the same space in memory, and generally memory is more dear than disk.

A less pressing issue with the current implementation is that inode numbers are not stable across boot (see the previous unioning file systems article for details on why this is a problem). If "fall-through" directories are implemented by allocating an inode for each directory entry on underlying file systems, then stable inode numbers will be a natural side effect. Another option is to store a persistent inode map somewhere - in a file in the top-level directory, or in an external file system, perhaps.

Hard links are handled - or, more accurately, not handled - in the same way as BSD union mounts. Again, it is not clear how many applications depend on modifying a file via one hard-linked path and seeing the changes via another hard-linked path (as opposed to symbolic link). The only method I can come up with to handle this correctly is to keep a persistent cache somewhere on disk of the inodes we have encountered with multiple hard links.

Here's an example of how it would work: Say we start a copy up for inode 42 and find that it has a link count of three. We would create an entry for the hard link database that includes the file system id, the inode number, the link count, and the inode number of the new copy on the top level file system. It could be stored in a file in CSV format, or as a symlink in a reserved directory in the root directory (e.g., "`/.hardlink_hack/<fs_id>/42`", which is a link to "`<new_inode_num> 3`"), or in a real database. Each time we open an inode on an underlying file system, we look it up in our hard link database; if an entry exists, we decrement the link count and create a hard link to the correct inode on the new file system. When all of the paths are found, the link count drops to one and the entry can be deleted from the database. The nice thing about this approach is that the amount of overhead is bounded and will disappear entirely when all the paths to the relevant inodes have been looked up. However, this still introduces a significant amount of possibly unnecessary complexity; the BSD implementation shows that many applications will happily run with not-quite-POSIXLY-correct hard link behavior.

Currently, `rename()` of directories across branches returns `EXDEV`, the error for trying to rename a file across different file systems. User space usually handles this transparently (since it already has to handle this case for directories from different file systems) and falls back to copying the contents of the directory over one by one. Implementing recursive `rename()` of directories across branches in the kernel is not a bright idea for the same reasons as rename across regular file systems; probably returning `EXDEV` is the best solution.

From a software engineering point of view, union mounts seem to be a reasonable compromise between features and ease of maintenance. Most of the VFS changes are isolated into fs/union.c, a file of about 1000 lines. About 1/3 of this file is the in-kernel readdir() implementation, which will almost certainly be replaced by something else before any possible merge. The changes to underlying file systems are fairly minimal and only needed for file systems mounted as writable branches. The main obstacle to merging this code is the readdir() implementation. Otherwise, file system maintainers have been noticeably more positive about union mounts than any other unioning implementation.

A nice summary of union mounts can be found in Bharata B. Rao's union mount slides for FOSS.IN [PDF].

Coming next

In the next article, we'll review unionfs and aufs, and compare the various implementations of unioning file systems for Linux. Stay tuned!

(Log in to post comments)

Union file systems: Implementations, part I
Posted Mar 26, 2009 11:22 UTC (Thu) by ebiederm (subscriber, #35028) [Link]

mini_fo anyone?

Given it's use in openwrt it may be one of the more widely
used
union filesystems.

fall-through dentries
Posted Mar 26, 2009 16:10 UTC (Thu) by arnd (subscriber, #8866) [Link]

AFAICT, the fall-through idea is really interesting but may
get
problematic when the lower-level file system is modified. The
original
union mount code should be able to handle readdir (after a
new opendir)
and lookup even if the lower file system is bind-mounted to
another
location and updated concurrently.

If you duplicate the entire directory structure in the top
level of the
union, this would not even easily work in the case where you
unmount the
top level, modify the lower level and then recreate the union
mount.

fall-through dentries
Posted Mar 26, 2009 17:03 UTC (Thu) by vaurora (guest, #38407) [Link]

I think the key to getting a maintainable unioning file system is limiting the feature set. Okay, Linux union mounts won't slice, dice, AND puree your files - but they will cover many common cases.

If you don't agree, just wait for the aufs article - it is almost certain to implement any feature you want.

fall-through dentries
Posted Mar 27, 2009 14:36 UTC (Fri) by arnd (subscriber, #8866) [Link]

I absolutely agree that the feature set needs to be
limited, and that's
what makes the Plan9 way of union mounts so beautiful
(thanks for
describing it here, I didn't know how it works before).

Limiting the implementation so that you can never change
the underlying
file system any more may be worth it but is still quite
drastic, so I

thought it should be mentioned more explicitly.

fall-through dentries
Posted Mar 30, 2009 8:33 UTC (Mon) by bharata (subscriber, #7885) [Link]

> If you duplicate the entire directory structure in the top level of the
> union, this would not even easily work in the case where you unmount the
> top level, modify the lower level and then recreate the union mount.

Not only that but since you cache the consolidated directory entries on disk, you will not be able to union mount your top layer later on any other lower layer filesystem.

I know that supporting all sorts of corner cases and features has caused major pains for union mount, but this restriction sounds a bit too restrictive to me :)

fall-through dentries
Posted Mar 30, 2009 8:45 UTC (Mon) by bharata (subscriber, #7885) [Link]

> I know that supporting all sorts of corner cases and features has caused > major pains for union mount, but this restriction sounds a bit too
> restrictive to me :)

This also will cause problems if you want to use union mount in server consolidation environments where you have multiple servers working out of a common base distribution as their lower layer. With your scheme, I will not be able to do updates (like security updates) to the base distribution and see it getting effected in all the servers.

Union file systems: Implementations, part I
Posted Apr 7, 2009 23:33 UTC (Tue) by sbelmon (subscriber, #55438) [Link]

Userspace whiteouts -- what about security? Doesn't that mean (at the very least) that I could see the names of deleted files? Or even maybe be able to open them? That doesn't seem to work -- there are too many things that will assume that once a file is deleted, it's deleted...