Pathname lookup

This write-up is based on three articles published at lwn.net:

- <https://lwn.net/Articles/649115/> Pathname lookup in Linux
- <https://lwn.net/Articles/649729/> RCU-walk: faster pathname lookup in Linux
- <https://lwn.net/Articles/650786/> A walk among the symlinks

Written by Neil Brown with help from Al Viro and Jon Corbet. It has subsequently been updated to reflect changes in the kernel including:

- per-directory parallel name lookup.
- `openat2()` resolution restriction flags.

Introduction to pathname lookup

The most obvious aspect of pathname lookup, which very little exploration is needed to discover, is that it is complex. There are many rules, special cases, and implementation alternatives that all combine to confuse the unwary reader. Computer science has long been acquainted with such complexity and has tools to help manage it. One tool that we will make extensive use of is "divide and conquer". For the early parts of the analysis we will divide off symlinks - leaving them until the final part. Well before we get to symlinks we have another major division based on the VFS's approach to locking which will allow us to review "REF-walk" and "RCU-walk" separately. But we are getting ahead of ourselves. There are some important low level distinctions we need to clarify first.

There are two sorts of ...

Pathnames (sometimes "file names"), used to identify objects in the filesystem, will be familiar to most readers. They contain two sorts of elements: "slashes" that are sequences of one or more "`/`" characters, and "components" that are sequences of one or more non-"`/`" characters. These form two kinds of paths. Those that start with slashes are "absolute" and start from the filesystem root. The others are "relative" and start from the current directory, or from some other location specified by a file descriptor given to "`*at()`" system calls such as openat().

It is tempting to describe the second kind as starting with a component, but that isn't always accurate: a pathname can lack both slashes and components, it can be empty, in other words. This is generally forbidden in POSIX, but some of those "`*at()`" system calls in Linux permit it when the `AT_EMPTY_PATH` flag is given. For example, if you have an open file descriptor on an executable file you can execute it by calling execveat() passing the file descriptor, an empty path, and the `AT_EMPTY_PATH` flag.

These paths can be divided into two sections: the final component and everything else. The "everything else" is the easy bit. In all cases it must identify a directory that already exists, otherwise an error such as `ENOENT` or `ENOTDIR` will be reported.

The final component is not so simple. Not only do different system calls interpret it quite differently (e.g. some create it, some do not), but it might not even exist: neither the empty pathname nor the pathname that is just slashes have a final component. If it does exist, it could be "`.`" or "`..`" which are handled quite differently from other components.

If a pathname ends with a slash, such as "`/tmp/foo/`" it might be tempting to consider that to have an empty final component. In many ways that would lead to correct results, but not always. In particular, `mkdir()` and `rmdir()` each create or remove a directory named by the final component, and they are required to work with pathnames ending in "`/`". According to POSIX:

> A pathname that contains at least one non-<slash> character and that ends with one or more trailing <slash> characters shall not be resolved successfully unless the last pathname component before the trailing <slash> characters names an existing directory or a directory entry that is to be created for a directory immediately after the pathname is resolved.

The Linux pathname walking code (mostly in `fs/namei.c`) deals with all of these issues: breaking the path into components, handling the "everything else" quite separately from the final component, and checking that the trailing slash is not used where it isn't permitted. It also addresses the important issue of concurrent access.

While one process is looking up a pathname, another might be making changes that affect that lookup. One fairly extreme case is that if "a/b" were renamed to "a/c/b" while another process were looking up "a/b/..", that process might successfully resolve on "a/c". Most races are much more subtle, and a big part of the task of pathname lookup is to prevent them from having damaging effects. Many of the possible races are seen most clearly in the context of the "dcache" and an understanding of that is central to understanding pathname lookup.

More than just a cache

The "dcache" caches information about names in each filesystem to make them quickly available for lookup. Each entry (known as a "dentry") contains three significant fields: a component name, a pointer to a parent dentry, and a pointer to the "inode" which contains further information about the object in that parent with the given name. The inode pointer can be `NULL` indicating that the name doesn't exist in the parent. While there can be linkage in the dentry of a directory to the dentries of the children, that linkage is not used for pathname lookup, and so will not be considered here.

The dcache has a number of uses apart from accelerating lookup. One that will be particularly relevant is that it is closely integrated with the mount table that records which filesystem is mounted where. What the mount table actually stores is which dentry is mounted on top of which other dentry.

When considering the dcache, we have another of our "two types" distinctions: there are two types of filesystems.

Some filesystems ensure that the information in the dcache is always completely accurate (though not necessarily complete). This can allow the VFS to determine if a particular file does or doesn't exist without checking with the filesystem, and means that the VFS can protect the filesystem against certain races and other problems. These are typically "local" filesystems such as ext3, XFS, and Btrfs.

Other filesystems don't provide that guarantee because they cannot. These are typically filesystems that are shared across a network, whether remote filesystems like NFS and 9P, or cluster filesystems like ocfs2 or cephfs. These filesystems allow the VFS to revalidate cached information, and must provide their own protection against awkward races. The VFS can detect these filesystems by the `DCACHE_OP_REVALIDATE` flag being set in the dentry.

REF-walk: simple concurrency management with refcounts and spinlocks

With all of those divisions carefully classified, we can now start looking at the actual process of walking along a path. In particular we will start with the handling of the "everything else" part of a pathname, and focus on the "REF-walk" approach to concurrency management. This code is found in the `link_path_walk()` function, if you ignore all the places that only run when "`LOOKUP_RCU`" (indicating the use of RCU-walk) is set.

REF-walk is fairly heavy-handed with locks and reference counts. Not as heavy-handed as in the old "big kernel lock" days, but certainly not afraid of taking a lock when one is needed. It uses a variety of different concurrency controls. A background understanding of the various primitives is assumed, or can be gleaned from elsewhere such as in Meet the Lockers.

The locking mechanisms used by REF-walk include:

dentry->d_lockref

This uses the lockref primitive to provide both a spinlock and a reference count. The special-sauce of this primitive is that the conceptual sequence "lock; inc_ref; unlock;" can often be performed with a single atomic memory operation.

Holding a reference on a dentry ensures that the dentry won't suddenly be freed and used for something else, so the values in various fields will behave as expected. It also protects the `->d_inode` reference to the inode to some extent.

The association between a dentry and its inode is fairly permanent. For example, when a file is renamed, the dentry and inode move together to the new location. When a file is created the dentry will initially be negative (i.e. `d_inode` is `NULL`), and will be assigned to the new inode as part of the act of creation.

When a file is deleted, this can be reflected in the cache either by setting `d_inode` to `NULL`, or by removing it from the hash table (described shortly) used to look up the name in the parent directory. If the dentry is still in use the second option is used as it is perfectly legal to keep using an open file after it has been deleted and having the dentry around helps. If the dentry is not otherwise in use (i.e. if the refcount in `d_lockref` is one), only then will `d_inode` be set to `NULL`. Doing it this way is more efficient for a very common case.

So as long as a counted reference is held to a dentry, a non-`NULL` `->d_inode` value will never be changed.

dentry->d_lock

`d_lock` is a synonym for the spinlock that is part of `d_lockref` above. For our purposes, holding this lock protects against the dentry being renamed or unlinked. In particular, its parent (`d_parent`), and its name (`d_name`) cannot be changed, and it cannot be removed from the dentry hash table.

When looking for a name in a directory, REF-walk takes `d_lock` on each candidate dentry that it finds in the hash table and then checks that the parent and name are correct. So it doesn't lock the parent while searching in the cache; it only locks children.

When looking for the parent for a given name (to handle "`..`"), REF-walk can take `d_lock` to get a stable reference to `d_parent`, but it first tries a more lightweight approach. As seen in `dget_parent()`, if a reference can be claimed on the parent, and if subsequently `d_parent` can be seen to have not changed, then there is no need to actually take the lock on the child.

rename_lock

Looking up a given name in a given directory involves computing a hash from the two values (the name and the dentry of the directory), accessing that slot in a hash table, and searching the linked list that is found there.

When a dentry is renamed, the name and the parent dentry can both change so the hash will almost certainly change too. This would move the dentry to a different chain in the hash table. If a filename search happened to be looking at a dentry that was moved in this way, it might end up continuing the search down the wrong chain, and so miss out on part of the correct chain.

The name-lookup process (`d_lookup()`) does *not* try to prevent this from happening, but only to detect when it happens. `rename_lock` is a seqlock that is updated whenever any dentry is renamed. If `d_lookup` finds that a rename happened while it unsuccessfully scanned a chain in the hash table, it simply tries again.

`rename_lock` is also used to detect and defend against potential attacks against `LOOKUP_BENEATH` and `LOOKUP_IN_ROOT` when resolving ".." (where the parent directory is moved outside the root, bypassing the `path_equal()` check). If `rename_lock` is updated during the lookup and the path encounters a "..", a potential attack occurred and `handle_dots()` will bail out with `-EAGAIN`.

inode->i_rwsem

`i_rwsem` is a read/write semaphore that serializes all changes to a particular directory. This ensures that, for example, an `unlink()` and a `rename()` cannot both happen at the same time. It also keeps the directory stable while the filesystem is asked to look up a name that is not currently in the dcache or, optionally, when the list of entries in a directory is being retrieved with `readdir()`.

This has a complementary role to that of `d_lock`: `i_rwsem` on a directory protects all of the names in that directory, while `d_lock` on a name protects just one name in a directory. Most changes to the dcache hold `i_rwsem` on the relevant directory inode and briefly take `d_lock` on one or more the dentries while the change happens. One exception is when idle dentries are removed from the dcache due to memory pressure. This uses `d_lock`, but `i_rwsem` plays no role.

The semaphore affects pathname lookup in two distinct ways. Firstly it prevents changes during lookup of a name in a directory. `walk_component()` uses `lookup_fast()` first which, in turn, checks to see if the name is in the cache, using only `d_lock` locking. If the name isn't found, then `walk_component()` falls back to `lookup_slow()` which takes a shared lock on `i_rwsem`, checks again that the name isn't in the cache, and then calls in to the filesystem to get a definitive answer. A new dentry will be added to the cache regardless of the result.

Secondly, when pathname lookup reaches the final component, it will sometimes need to take an exclusive lock on `i_rwsem` before performing the last lookup so that the required exclusion can be achieved. How path lookup chooses to take, or not take, `i_rwsem` is one of the issues addressed in a subsequent section.

If two threads attempt to look up the same name at the same time - a name that is not yet in the dcache - the shared lock on `i_rwsem` will not prevent them both adding new dentries with the same name. As this would result in confusion an extra level of interlocking is used, based around a secondary hash table (`in_lookup_hashtable`) and a per-dentry flag bit (`DCACHE_PAR_LOOKUP`).

To add a new dentry to the cache while only holding a shared lock on `i_rwsem`, a thread must call `d_alloc_parallel()`. This allocates a dentry, stores the required name and parent in it, checks if there is already a matching dentry in the primary or secondary hash tables, and if not, stores the newly allocated dentry in the secondary hash table, with `DCACHE_PAR_LOOKUP` set.

If a matching dentry was found in the primary hash table then that is returned and the caller can know that it lost a race with some other thread adding the entry. If no matching dentry is found in either cache, the newly allocated dentry is returned and the caller can detect this from the presence of `DCACHE_PAR_LOOKUP`. In this case it knows that it has won any race and now is responsible for asking the filesystem to perform the lookup and find the matching inode. When the lookup is complete, it must call `d_lookup_done()` which clears the flag and does some other house keeping, including removing the dentry from the secondary hash table - it will normally have been added to the primary hash table already. Note that a `struct waitqueue_head` is passed to `d_alloc_parallel()`, and `d_lookup_done()` must be called while this `waitqueue_head` is still in scope.

If a matching dentry is found in the secondary hash table, `d_alloc_parallel()` has a little more work to do. It first waits for `DCACHE_PAR_LOOKUP` to be cleared, using a wait_queue that was passed to the instance of `d_alloc_parallel()` that won the race and that will be woken by the call to `d_lookup_done()`. It then checks to see if the dentry has now been added to the primary hash table. If it has, the dentry is returned and the caller just sees that it lost any race. If it hasn't been added to the primary hash table, the most likely explanation is that some other dentry was added instead using `d_splice_alias()`. In any case, `d_alloc_parallel()` repeats all the look ups from the start and will normally return something from the primary hash table.

mnt->mnt_count

`mnt_count` is a per-CPU reference counter on "`mount`" structures. Per-CPU here means that incrementing the count is cheap as it only uses CPU-local memory, but checking if the count is zero is expensive as it needs to check with every CPU. Taking a `mnt_count` reference prevents the mount structure from disappearing as the result of regular unmount operations, but does not prevent a "lazy" unmount. So holding `mnt_count` doesn't ensure that the mount remains in the namespace and, in particular, doesn't stabilize the link to the mounted-on dentry. It does, however, ensure that the `mount` data structure remains coherent, and it provides a reference to the root dentry of the mounted filesystem. So a reference through `->mnt_count` provides a stable reference to the mounted dentry, but not the mounted-on dentry.

mount_lock

`mount_lock` is a global seqlock, a bit like `rename_lock`. It can be used to check if any change has been made to any mount points.

While walking down the tree (away from the root) this lock is used when crossing a mount point to check that the crossing was safe. That is, the value in the seqlock is read, then the code finds the mount that is mounted on the current directory, if there is one, and increments the `mnt_count`. Finally the value in `mount_lock` is checked against the old value. If there is no change, then the crossing was safe. If there was a change, the `mnt_count` is decremented and the whole process is retried.

When walking up the tree (towards the root) by following a ".." link, a little more care is needed. In this case the seqlock (which contains both a counter and a spinlock) is fully locked to prevent any changes to any mount points while stepping up. This locking is needed to stabilize the link to the mounted-on dentry, which the refcount on the mount itself doesn't ensure.

`mount_lock` is also used to detect and defend against potential attacks against `LOOKUP_BENEATH` and `LOOKUP_IN_ROOT` when resolving ".." (where the parent directory is moved outside the root, bypassing the `path_equal()` check). If `mount_lock` is updated during the lookup and the path encounters a "..", a potential attack occurred and `handle_dots()` will bail out with `-EAGAIN`.

RCU

Finally the global (but extremely lightweight) RCU read lock is held from time to time to ensure certain data structures don't get freed unexpectedly.

In particular it is held while scanning chains in the dcache hash table, and the mount point hash table.

Bringing it together with `struct nameidata`

Throughout the process of walking a path, the current status is stored in a `struct nameidata`, "namei" being the traditional name - dating all the way back to First Edition Unix - of the function that converts a "name" to an "inode". `struct nameidata` contains (among other fields):

`struct path path`

A `path` contains a `struct vfsmount` (which is embedded in a `struct mount`) and a `struct dentry`. Together these record the current status of the walk. They start out referring to the starting point (the current working directory, the root directory, or some other directory identified by a file descriptor), and are updated on each step. A reference through `d_lockref` and `mnt_count` is always held.

`struct qstr last`

This is a string together with a length (i.e. *not* `nul` terminated) that is the "next" component in the pathname.

`int last_type`

This is one of `LAST_NORM`, `LAST_ROOT`, `LAST_DOT` or `LAST_DOTDOT`. The `last` field is only valid if the type is `LAST_NORM`.

`struct path root`

This is used to hold a reference to the effective root of the filesystem. Often that reference won't be needed, so this field is only assigned the first time it is used, or when a non-standard root is requested. Keeping a reference in the `nameidata` ensures that only one root is in effect for the entire path walk, even if it races with a `chroot()` system call.

It should be noted that in the case of `LOOKUP_IN_ROOT` or `LOOKUP_BENEATH`, the effective root becomes the directory file descriptor passed to `openat2()` (which exposes these `LOOKUP_` flags).

The root is needed when either of two conditions holds: (1) either the pathname or a symbolic link starts with a "'/'", or (2) a "`..`" component is being handled, since "`..`" from the root must always stay at the root. The value used is usually the current root directory of the calling process. An alternate root can be provided as when `sysctl()` calls `file_open_root()`, and when NFSv4 or Btrfs call `mount_subtree()`. In each case a pathname is being looked up in a very specific part of the filesystem, and the lookup must not be allowed to escape that subtree. It works a bit like a local `chroot()`.

Ignoring the handling of symbolic links, we can now describe the "`link_path_walk()`" function, which handles the lookup of everything except the final component as:

> Given a path (`name`) and a nameidata structure (`nd`), check that the current directory has execute permission and then advance `name` over one component while updating `last_type` and `last`. If that was the final component, then return, otherwise call `walk_component()` and repeat from the top.

`walk_component()` is even easier. If the component is `LAST_DOTS`, it calls `handle_dots()` which does the necessary locking as already described. If it finds a `LAST_NORM` component it first calls "`lookup_fast()`" which only looks in the dcache, but will ask the filesystem to revalidate the result if it is that sort of filesystem. If that doesn't get a good result, it calls "`lookup_slow()`" which takes `i_rwsem`, rechecks the cache, and then asks the filesystem to find a definitive answer.

As the last step of walk_component(), step_into() will be called either directly from walk_component() or from handle_dots(). It calls handle_mounts(), to check and handle mount points, in which a new `struct path` is created containing a counted reference to the new dentry and a reference to the new `vfsmount` which is only counted if it is different from the previous `vfsmount`. Then if there is a symbolic link, step_into() calls pick_link() to deal with it, otherwise it installs the new `struct path` in the `struct nameidata`, and drops the unneeded references.

This "hand-over-hand" sequencing of getting a reference to the new dentry before dropping the reference to the previous dentry may seem obvious, but is worth pointing out so that we will recognize its analogue in the "RCU-walk" version.

Handling the final component

`link_path_walk()` only walks as far as setting `nd->last` and `nd->last_type` to refer to the final component of the path. It does not call `walk_component()` that last time. Handling that final component remains for the caller to sort out. Those callers are path_lookupat(), path_parentat() and path_openat() each of which handles the differing requirements of different system calls.

`path_parentat()` is clearly the simplest - it just wraps a little bit of housekeeping around `link_path_walk()` and returns the parent directory and final component to the caller. The caller will be either aiming to create a name (via `filename_create()`) or remove or rename a name (in which case `user_path_parent()` is used). They will use `i_rwsem` to exclude other changes while they validate and then perform their operation.

`path_lookupat()` is nearly as simple - it is used when an existing object is wanted such as by `stat()` or `chmod()`. It essentially just calls `walk_component()` on the final component through a call to `lookup_last()`. `path_lookupat()` returns just the final dentry. It is worth noting that when flag `LOOKUP_MOUNTPOINT` is set, path_lookupat() will unset LOOKUP_JUMPED in nameidata so that in the subsequent path traversal d_weak_revalidate() won't be called. This is important when unmounting a filesystem that is inaccessible, such as one provided by a dead NFS server.

Finally `path_openat()` is used for the `open()` system call; it contains, in support functions starting with "open_last_lookups()", all the complexity needed to handle the different subtleties of O_CREAT (with or without O_EXCL), final "`/`" characters, and trailing symbolic links. We will revisit this in the final part of this series, which focuses on those symbolic links. "open_last_lookups()" will sometimes, but not always, take `i_rwsem`, depending on what it finds.

Each of these, or the functions which call them, need to be alert to the possibility that the final component is not `LAST_NORM`. If the goal of the lookup is to create something, then any value for `last_type` other than `LAST_NORM` will result in an error. For example if `path_parentat()` reports `LAST_DOTDOT`, then the caller won't try to create that name. They also check for trailing slashes by testing `last.name[last.len]`. If there is any character beyond the final component, it must be a trailing slash.

Revalidation and automounts

Apart from symbolic links, there are only two parts of the "REF-walk" process not yet covered. One is the handling of stale cache entries and the other is automounts.

On filesystems that require it, the lookup routines will call the `->d_revalidate()` dentry method to ensure that the cached information is current. This will often confirm validity or update a few details from a server. In some cases it may find that there has been change further up the path and that something that was thought to be valid previously isn't really. When this happens the lookup of the whole path is aborted and retried with the "`LOOKUP_REVAL`" flag set. This forces revalidation to be more thorough. We will see more details of this retry process in the next article.

Automount points are locations in the filesystem where an attempt to lookup a name can trigger changes to how that lookup should be handled, in particular by mounting a filesystem there. These are covered in greater detail in autofs.txt in the Linux documentation tree, but a few notes specifically related to path lookup are in order here.

The Linux VFS has a concept of "managed" dentries. There are three potentially interesting things about these dentries corresponding to three different flags that might be set in `dentry->d_flags`:

DCACHE_MANAGE_TRANSIT

If this flag has been set, then the filesystem has requested that the `d_manage()` dentry operation be called before handling any possible mount point. This can perform two particular services:

It can block to avoid races. If an automount point is being unmounted, the `d_manage()` function will usually wait for that process to complete before letting the new lookup proceed and possibly trigger a new automount.

It can selectively allow only some processes to transit through a mount point. When a server process is managing automounts, it may need to access a directory without triggering normal automount processing. That server process can identify itself to the `autofs` filesystem, which will then give it a special pass through `d_manage()` by returning `-EISDIR`.

DCACHE_MOUNTED

This flag is set on every dentry that is mounted on. As Linux supports multiple filesystem namespaces, it is possible that the dentry may not be mounted on in *this* namespace, just in some other. So this flag is seen as a hint, not a promise.

If this flag is set, and `d_manage()` didn't return `-EISDIR`, `lookup_mnt()` is called to examine the mount hash table (honoring the `mount_lock` described earlier) and possibly return a new `vfsmount` and a new `dentry` (both with counted references).

DCACHE_NEED_AUTOMOUNT

If `d_manage()` allowed us to get this far, and `lookup_mnt()` didn't find a mount point, then this flag causes the `d_automount()` dentry operation to be called.

The `d_automount()` operation can be arbitrarily complex and may communicate with server processes etc. but it should ultimately either report that there was an error, that there was nothing to mount, or should provide an updated `struct path` with new `dentry` and `vfsmount`.

In the latter case, `finish_automount()` will be called to safely install the new mount point into the mount table.

There is no new locking of import here and it is important that no locks (only counted references) are held over this processing due to the very real possibility of extended delays. This will become more important next time when we examine RCU-walk which is particularly sensitive to delays.

RCU-walk - faster pathname lookup in Linux

RCU-walk - faster pathname lookup in Linux

RCU-walk is another algorithm for performing pathname lookup in Linux. It is in many ways similar to REF-walk and the two share quite a bit of code. The significant difference in RCU-walk is how it allows for the possibility of concurrent access.

We noted that REF-walk is complex because there are numerous details and special cases. RCU-walk reduces this complexity by simply refusing to handle a number of cases -- it instead falls back to REF-walk. The difficulty with RCU-walk comes from a different direction: unfamiliarity. The locking rules when depending on RCU are quite different from traditional locking, so we will spend a little extra time when we come to those.

Clear demarcation of roles

The easiest way to manage concurrency is to forcibly stop any other thread from changing the data structures that a given thread is looking at. In cases where no other thread would even think of changing the data and lots of different threads want to read at the same time, this can be very costly. Even when using locks that permit multiple concurrent readers, the simple act of updating the count of the number of current readers can impose an unwanted cost. So the goal when reading a shared data structure that no other process is changing is to avoid writing anything to memory at all. Take no locks, increment no counts, leave no footprints.

The REF-walk mechanism already described certainly doesn't follow this principle, but then it is really designed to work when there may well be other threads modifying the data. RCU-walk, in contrast, is designed for the common situation where there are lots of frequent readers and only occasional writers. This may not be common in all parts of the filesystem tree, but in many parts it will be. For the other parts it is important that RCU-walk can quickly fall back to using REF-walk.

Pathname lookup always starts in RCU-walk mode but only remains there as long as what it is looking for is in the cache and is stable. It dances lightly down the cached filesystem image, leaving no footprints and carefully watching where it is, to be sure it doesn't trip. If it notices that something has changed or is changing, or if something isn't in the cache, then it tries to stop gracefully and switch to REF-walk.

This stopping requires getting a counted reference on the current `vfsmount` and `dentry`, and ensuring that these are still valid - that a path walk with REF-walk would have found the same entries. This is an invariant that RCU-walk must guarantee. It can only make decisions, such as selecting the next step, that are decisions which REF-walk could also have made if it were walking down the tree at the same time. If the graceful stop succeeds, the rest of the path is processed with the reliable, if slightly sluggish, REF-walk. If RCU-walk finds it cannot stop gracefully, it simply gives up and restarts from the top with REF-walk.

This pattern of "try RCU-walk, if that fails try REF-walk" can be clearly seen in functions like filename_lookup(), filename_parentat(), do_filp_open(), and do_file_open_root(). These four correspond roughly to the three `path_*()` functions we met earlier, each of which calls `link_path_walk()`. The `path_*()` functions are called using different mode flags until a mode is found which works. They are first called with `LOOKUP_RCU` set to request "RCU-walk". If that fails with the error `ECHILD` they are called again with no special flag to request "REF-walk". If either of those report the error `ESTALE` a final attempt is made with `LOOKUP_REVAL` set (and no `LOOKUP_RCU`) to ensure that entries found in the cache are forcibly revalidated - normally entries are only revalidated if the filesystem determines that they are too old to trust.

The `LOOKUP_RCU` attempt may drop that flag internally and switch to REF-walk, but will never then try to switch back to RCU-walk. Places that trip up RCU-walk are much more likely to be near the leaves and so it is very unlikely that there will be much, if any, benefit from switching back.

RCU and seqlocks: fast and light

RCU is, unsurprisingly, critical to RCU-walk mode. The `rcu_read_lock()` is held for the entire time that RCU-walk is walking down a path. The particular guarantee it provides is that the key data structures - dentries, inodes, super_blocks, and mounts - will not be freed while the lock is held. They might be unlinked or invalidated in one way or another, but the memory will not be repurposed so values in various fields will still be meaningful. This is the only guarantee that RCU provides; everything else is done using seqlocks.

As we saw above, REF-walk holds a counted reference to the current dentry and the current vfsmount, and does not release those references before taking references to the "next" dentry or vfsmount. It also sometimes takes the `d_lock` spinlock. These references and locks are taken to prevent certain changes from happening. RCU-walk must not take those references or locks and so cannot prevent such changes. Instead, it checks to see if a change has been made, and aborts or retries if it has.

To preserve the invariant mentioned above (that RCU-walk may only make decisions that REF-walk could have made), it must make the checks at or near the same places that REF-walk holds the references. So, when REF-walk increments a reference count or takes a spinlock, RCU-walk samples the status of a seqlock using `read_seqcount_begin()` or a similar function. When REF-walk decrements the count or drops the lock, RCU-walk checks if the sampled status is still valid using `read_seqcount_retry()` or similar.

However, there is a little bit more to seqlocks than that. If RCU-walk accesses two different fields in a seqlock-protected structure, or accesses the same field twice, there is no a priori guarantee of any consistency between those accesses. When consistency is needed - which it usually is - RCU-walk must take a copy and then use `read_seqcount_retry()` to validate that copy.

`read_seqcount_retry()` not only checks the sequence number, but also imposes a memory barrier so that no memory-read instruction from *before* the call can be delayed until *after* the call, either by the CPU or by the compiler. A simple example of this can be seen in `slow_dentry_cmp()` which, for filesystems which do not use simple byte-wise name equality, calls into the filesystem to compare a name against a dentry. The length and name pointer are copied into local variables, then `read_seqcount_retry()` is called to confirm the two are consistent, and only then is `->d_compare()` called. When standard filename comparison is used, `dentry_cmp()` is called instead. Notably it does *not* use `read_seqcount_retry()`, but instead has a large comment explaining why the consistency guarantee isn't necessary. A subsequent `read_seqcount_retry()` will be sufficient to catch any problem that could occur at this point.

With that little refresher on seqlocks out of the way we can look at the bigger picture of how RCU-walk uses seqlocks.

`mount_lock` and `nd->m_seq`

We already met the `mount_lock` seqlock when REF-walk used it to ensure that crossing a mount point is performed safely. RCU-walk uses it for that too, but for quite a bit more.

Instead of taking a counted reference to each `vfsmount` as it descends the tree, RCU-walk samples the state of `mount_lock` at the start of the walk and stores this initial sequence number in the `struct nameidata` in the `m_seq` field. This one lock and one sequence number are used to validate all accesses to all `vfsmounts`, and all mount point crossings. As changes to the mount table are relatively rare, it is reasonable to fall back on REF-walk any time that any "mount" or "unmount" happens.

`m_seq` is checked (using `read_seqretry()`) at the end of an RCU-walk sequence, whether switching to REF-walk for the rest of the path or when the end of the path is reached. It is also checked when stepping down over a mount point (in `__follow_mount_rcu()`) or up (in `follow_dotdot_rcu()`). If it is ever found to have changed, the whole RCU-walk sequence is aborted and the path is processed again by REF-walk.

If RCU-walk finds that `mount_lock` hasn't changed then it can be sure that, had REF-walk taken counted references on each vfsmount, the results would have been the same. This ensures the invariant holds, at least for vfsmount structures.

`dentry->d_seq` and `nd->seq`

In place of taking a count or lock on `d_reflock`, RCU-walk samples the per-dentry `d_seq` seqlock, and stores the sequence number in the `seq` field of the nameidata structure, so `nd->seq` should always be the current sequence number of `nd->dentry`. This number needs to be revalidated after copying, and before using, the name, parent, or inode of the dentry.

The handling of the name we have already looked at, and the parent is only accessed in `follow_dotdot_rcu()` which fairly trivially follows the required pattern, though it does so for three different cases.

When not at a mount point, `d_parent` is followed and its `d_seq` is collected. When we are at a mount point, we instead follow the `mnt->mnt_mountpoint` link to get a new dentry and collect its `d_seq`. Then, after finally finding a `d_parent` to follow, we must check if we have landed on a mount point and, if so, must find that mount point and follow the `mnt->mnt_root` link. This would imply a somewhat unusual, but certainly possible, circumstance where the starting point of the path lookup was in part of the filesystem that was mounted on, and so not visible from the root.

The inode pointer, stored in `->d_inode`, is a little more interesting. The inode will always need to be accessed at least twice, once to determine if it is NULL and once to verify access permissions. Symlink handling requires a validated inode pointer too. Rather than revalidating on each access, a copy is made on the first access and it is stored in the `inode` field of `nameidata` from where it can be safely accessed without further validation.

`lookup_fast()` is the only lookup routine that is used in RCU-mode, `lookup_slow()` being too slow and requiring locks. It is in `lookup_fast()` that we find the important "hand over hand" tracking of the current dentry.

The current `dentry` and current `seq` number are passed to `__d_lookup_rcu()` which, on success, returns a new `dentry` and a new `seq` number. `lookup_fast()` then copies the inode pointer and revalidates the new `seq` number. It then validates the old `dentry` with the old `seq` number one last time and only then continues. This process of getting the `seq` number of the new dentry and then checking the `seq` number of the old exactly mirrors the process of getting a counted reference to the new dentry before dropping that for the old dentry which we saw in REF-walk.

No `inode->i_rwsem` or even `rename_lock`

A semaphore is a fairly heavyweight lock that can only be taken when it is permissible to sleep. As `rcu_read_lock()` forbids sleeping, `inode->i_rwsem` plays no role in RCU-walk. If some other thread does take `i_rwsem` and modifies the directory in a way that RCU-walk needs to notice, the result will be either that RCU-walk fails to find the dentry that it is looking for, or it will find a dentry which `read_seqretry()` won't validate. In either case it will drop down to REF-walk mode which can take whatever locks are needed.

Though `rename_lock` could be used by RCU-walk as it doesn't require any sleeping, RCU-walk doesn't bother. REF-walk uses `rename_lock` to protect against the possibility of hash chains in the dcache changing while they are being searched. This can result in failing to find something that actually is there. When RCU-walk fails to find something in the dentry cache, whether it is really there or not, it already drops down to REF-walk and tries again with appropriate locking. This neatly handles all cases, so adding extra checks on rename_lock would bring no significant value.

`unlazy walk()` and `complete_walk()`

That "dropping down to REF-walk" typically involves a call to `unlazy_walk()`, so named because "RCU-walk" is also sometimes referred to as "lazy walk". `unlazy_walk()` is called when following the path down to the current vfsmount/dentry pair seems to have proceeded successfully, but the next step is problematic. This can happen if the next name cannot be found in the dcache, if permission checking or name revalidation couldn't be achieved while the `rcu_read_lock()` is held (which forbids sleeping), if an automount point is found, or in a couple of cases involving symlinks. It is also called from `complete_walk()` when the lookup has reached the final component, or the very end of the path, depending on which particular flavor of lookup is used.

Other reasons for dropping out of RCU-walk that do not trigger a call to `unlazy_walk()` are when some inconsistency is found that cannot be handled immediately, such as `mount_lock` or one of the `d_seq` seqlocks reporting a change. In these cases the relevant function will return `-ECHILD` which will percolate up until it triggers a new attempt from the top using REF-walk.

For those cases where `unlazy_walk()` is an option, it essentially takes a reference on each of the pointers that it holds (vfsmount, dentry, and possibly some symbolic links) and then verifies that the relevant seqlocks have not been changed. If there have been changes, it, too, aborts with `-ECHILD`, otherwise the transition to REF-walk has been a success and the lookup process continues.

Taking a reference on those pointers is not quite as simple as just incrementing a counter. That works to take a second reference if you already have one (often indirectly through another object), but it isn't sufficient if you don't actually have a counted reference at all. For `dentry->d_lockref`, it is safe to increment the reference counter to get a reference unless it has been explicitly marked as "dead" which involves setting the counter to `-128`. `lockref_get_not_dead()` achieves this.

For `mnt->mnt_count` it is safe to take a reference as long as `mount_lock` is then used to validate the reference. If that validation fails, it may *not* be safe to just drop that reference in the standard way of calling `mnt_put()` - an unmount may have progressed too far. So the code in `legitimize_mnt()`, when it finds that the reference it got might not be safe, checks the `MNT_SYNC_UMOUNT` flag to determine if a simple `mnt_put()` is correct, or if it should just decrement the count and pretend none of this ever happened.

Taking care in filesystems

RCU-walk depends almost entirely on cached information and often will not call into the filesystem at all. However there are two places, besides the already-mentioned component-name comparison, where the file system might be included in RCU-walk, and it must know to be careful.

If the filesystem has non-standard permission-checking requirements - such as a networked filesystem which may need to check with the server - the `i_op->permission` interface might be called during RCU-walk. In this case an extra "`MAY_NOT_BLOCK`" flag is passed so that it knows not to sleep, but to return `-ECHILD` if it cannot complete promptly. `i_op->permission` is given the inode pointer, not the dentry, so it doesn't need to worry about further consistency checks. However if it accesses any other filesystem data structures, it must ensure they are safe to be accessed with only the `rcu_read_lock()` held. This typically means they must be freed using `kfree_rcu()` or similar.

If the filesystem may need to revalidate dcache entries, then `d_op->d_revalidate` may be called in RCU-walk too. This interface *is* passed the dentry but does not have access to the `inode` or the `seq` number from the `nameidata`, so it needs to be extra careful when accessing fields in the dentry. This "extra care" typically involves using READ_ONCE() to access fields, and verifying the result is not NULL before using it. This pattern can be seen in `nfs_lookup_revalidate()`.

A pair of patterns

In various places in the details of REF-walk and RCU-walk, and also in the big picture, there are a couple of related patterns that are worth being aware of.

The first is "try quickly and check, if that fails try slowly". We can see that in the high-level approach of first trying RCU-walk and then trying REF-walk, and in places where `unlazy_walk()` is used to switch to REF-walk for the rest of the path. We also saw it earlier in `dget_parent()` when following a "`..`" link. It tries a quick way to get a reference, then falls back to taking locks if needed.

The second pattern is "try quickly and check, if that fails try again - repeatedly". This is seen with the use of `rename_lock` and `mount_lock` in REF-walk. RCU-walk doesn't make use of this pattern - if anything goes wrong it is much safer to just abort and try a more sedate approach.

The emphasis here is "try quickly and check". It should probably be "try quickly *and carefully*, then check". The fact that checking is needed is a reminder that the system is dynamic and only a limited number of things are safe at all. The most likely cause of errors in this whole process is assuming something is safe when in reality it isn't. Careful consideration of what exactly guarantees the safety of each access is sometimes necessary.

A walk among the symlinks

There are several basic issues that we will examine to understand the handling of symbolic links: the symlink stack, together with cache lifetimes, will help us understand the overall recursive handling of symlinks and lead to the special care needed for the final component. Then a consideration of access-time updates and summary of the various flags controlling lookup will finish the story.

The symlink stack

There are only two sorts of filesystem objects that can usefully appear in a path prior to the final component: directories and symlinks. Handling directories is quite straightforward: the new directory simply becomes the starting point at which to interpret the next component on the path. Handling symbolic links requires a bit more work.

Conceptually, symbolic links could be handled by editing the path. If a component name refers to a symbolic link, then that component is replaced by the body of the link and, if that body starts with a '/', then all preceding parts of the path are discarded. This is what the "`readlink -f`" command does, though it also edits out "`.`" and "`..`" components.

Directly editing the path string is not really necessary when looking up a path, and discarding early components is pointless as they aren't looked at anyway. Keeping track of all remaining components is important, but they can of course be kept separately; there is no need to concatenate them. As one symlink may easily refer to another, which in turn can refer to a third, we may need to keep the remaining components of several paths, each to be processed when the preceding ones are completed. These path remnants are kept on a stack of limited size.

There are two reasons for placing limits on how many symlinks can occur in a single path lookup. The most obvious is to avoid loops. If a symlink referred to itself either directly or through intermediaries, then following the symlink can never complete successfully - the error `ELOOP` must be returned. Loops can be detected without imposing limits, but limits are the simplest solution and, given the second reason for restriction, quite sufficient.

The second reason was outlined recently by Linus:

> Because it's a latency and DoS issue too. We need to react well to true loops, but also to "very deep" non-loops. It's not about memory use, it's about users triggering unreasonable CPU resources.

Linux imposes a limit on the length of any pathname: `PATH_MAX`, which is 4096. There are a number of reasons for this limit; not letting the kernel spend too much time on just one path is one of them. With symbolic links you can effectively generate much longer paths so some sort of limit is needed for the same reason. Linux imposes a limit of at most 40 (MAXSYMLINKS) symlinks in any one path lookup. It previously imposed a further limit of eight on the maximum depth of recursion, but that was raised to 40 when a separate stack was implemented, so there is now just the one limit.

The `nameidata` structure that we met in an earlier article contains a small stack that can be used to store the remaining part of up to two symlinks. In many cases this will be sufficient. If it isn't, a separate stack is allocated with room for 40 symlinks. Pathname lookup will never exceed that stack as, once the 40th symlink is detected, an error is returned.

It might seem that the name remnants are all that needs to be stored on this stack, but we need a bit more. To see that, we need to move on to cache lifetimes.

Storage and lifetime of cached symlinks

Like other filesystem resources, such as inodes and directory entries, symlinks are cached by Linux to avoid repeated costly access to external storage. It is particularly important for RCU-walk to be able to find and temporarily hold onto these cached entries, so that it doesn't need to drop down into REF-walk.

While each filesystem is free to make its own choice, symlinks are typically stored in one of two places. Short symlinks are often stored directly in the inode. When a filesystem allocates a `struct inode` it typically allocates extra space to store private data (a common object-oriented design pattern in the kernel). This will sometimes include space for a symlink. The other common location is in the page cache, which normally stores the content of files. The pathname in a symlink can be seen as the content of that symlink and can easily be stored in the page cache just like file content.

When neither of these is suitable, the next most likely scenario is that the filesystem will allocate some temporary memory and copy or construct the symlink content into that memory whenever it is needed.

When the symlink is stored in the inode, it has the same lifetime as the inode which, itself, is protected by RCU or by a counted reference on the dentry. This means that the mechanisms that pathname lookup uses to access the dcache and icache (inode cache) safely are quite sufficient for accessing some cached symlinks safely. In these cases, the `i_link` pointer in the inode is set to point to wherever the symlink is stored and it can be accessed directly whenever needed.

When the symlink is stored in the page cache or elsewhere, the situation is not so straightforward. A reference on a dentry or even on an inode does not imply any reference on cached pages of that inode, and even an `rcu_read_lock()` is not sufficient to ensure that a page will not disappear. So for these symlinks the pathname lookup code needs to ask the filesystem to provide a stable reference and, significantly, needs to release that reference when it is finished with it.

Taking a reference to a cache page is often possible even in RCU-walk mode. It does require making changes to memory, which is best avoided, but that isn't necessarily a big cost and it is better than dropping out of RCU-walk mode completely. Even filesystems that allocate space to copy the symlink into can use `GFP_ATOMIC` to often successfully allocate memory without the need to drop out of RCU-walk. If a filesystem cannot successfully get a reference in RCU-walk mode, it must return `-ECHILD` and `unlazy_walk()` will be called to return to REF-walk mode in which the filesystem is allowed to sleep.

The place for all this to happen is the `i_op->get_link()` inode method. This is called both in RCU-walk and REF-walk. In RCU-walk the `dentry*` argument is NULL, `->get_link()` can return -ECHILD to drop out of RCU-walk. Much like the `i_op->permission()` method we looked at previously, `->get_link()` would need to be careful that all the data structures it references are safe to be accessed while holding no counted reference, only the RCU lock. A callback `struct delayed_called` will be passed to `->get_link()`: file systems can set their own put_link function and argument through set_delayed_call(). Later on, when VFS wants to put link, it will call do_delayed_call() to invoke that callback function with the argument.

In order for the reference to each symlink to be dropped when the walk completes, whether in RCU-walk or REF-walk, the symlink stack needs to contain, along with the path remnants:

- the `struct path` to provide a reference to the previous path

- the `const char *` to provide a reference to the to previous name
- the `seq` to allow the path to be safely switched from RCU-walk to REF-walk
- the `struct delayed_call` for later invocation.

This means that each entry in the symlink stack needs to hold five pointers and an integer instead of just one pointer (the path remnant). On a 64-bit system, this is about 40 bytes per entry; with 40 entries it adds up to 1600 bytes total, which is less than half a page. So it might seem like a lot, but is by no means excessive.

Note that, in a given stack frame, the path remnant (`name`) is not part of the symlink that the other fields refer to. It is the remnant to be followed once that symlink has been fully parsed.

Following the symlink

The main loop in `link_path_walk()` iterates seamlessly over all components in the path and all of the non-final symlinks. As symlinks are processed, the `name` pointer is adjusted to point to a new symlink, or is restored from the stack, so that much of the loop doesn't need to notice. Getting this `name` variable on and off the stack is very straightforward; pushing and popping the references is a little more complex.

When a symlink is found, walk_component() calls pick_link() via step_into() which returns the link from the filesystem. Providing that operation is successful, the old path `name` is placed on the stack, and the new value is used as the `name` for a while. When the end of the path is found (i.e. `*name` is `'\0'`) the old `name` is restored off the stack and path walking continues.

Pushing and popping the reference pointers (inode, cookie, etc.) is more complex in part because of the desire to handle tail recursion. When the last component of a symlink itself points to a symlink, we want to pop the symlink-just-completed off the stack before pushing the symlink-just-found to avoid leaving empty path remnants that would just get in the way.

It is most convenient to push the new symlink references onto the stack in `walk_component()` immediately when the symlink is found; `walk_component()` is also the last piece of code that needs to look at the old symlink as it walks that last component. So it is quite convenient for `walk_component()` to release the old symlink and pop the references just before pushing the reference information for the new symlink. It is guided in this by three flags: `WALK_NOFOLLOW` which forbids it from following a symlink if it finds one, `WALK_MORE` which indicates that it is yet too early to release the current symlink, and `WALK_TRAILING` which indicates that it is on the final component of the lookup, so we will check userspace flag `LOOKUP_FOLLOW` to decide whether follow it when it is a symlink and call `may_follow_link()` to check if we have privilege to follow it.

Symlinks with no final component

A pair of special-case symlinks deserve a little further explanation. Both result in a new `struct path` (with mount and dentry) being set up in the `nameidata`, and result in pick_link() returning `NULL`.

The more obvious case is a symlink to "`/`". All symlinks starting with "`/`" are detected in pick_link() which resets the `nameidata` to point to the effective filesystem root. If the symlink only contains "`/`" then there is nothing more to do, no components at all, so `NULL` is returned to indicate that the symlink can be released and the stack frame discarded.

The other case involves things in `/proc` that look like symlinks but aren't really (and are therefore commonly referred to as "magic-links"):

```
$ ls -l /proc/self/fd/1
lrwx------ 1 neilb neilb 64 Jun 13 10:19 /proc/self/fd/1 -> /dev/pts/4
```

Every open file descriptor in any process is represented in `/proc` by something that looks like a symlink. It is really a reference to the target file, not just the name of it. When you `readlink` these objects you get a name that might refer to the same file - unless it has been unlinked or mounted over. When `walk_component()` follows one of these, the `->get_link()` method in "procfs" doesn't return a string name, but instead calls nd_jump_link() which updates the `nameidata` in place to point to that target. `->get_link()` then returns `NULL`. Again there is no final component and pick_link() returns `NULL`.

Following the symlink in the final component

All this leads to `link_path_walk()` walking down every component, and following all symbolic links it finds, until it reaches the final component. This is just returned in the `last` field of `nameidata`. For some callers, this is all they need; they want to create that `last` name if it doesn't exist or give an error if it does. Other callers will want to follow a symlink if one is found, and possibly apply special handling to the last component of that symlink, rather than just the last component of the original file name. These callers potentially need to call `link_path_walk()` again and again on successive symlinks until one is found that doesn't point to another symlink.

This case is handled by relevant callers of link_path_walk(), such as path_lookupat(), path_openat() using a loop that calls link_path_walk(), and then handles the final component by calling open_last_lookups() or lookup_last(). If it is a symlink that needs to be followed, open_last_lookups() or lookup_last() will set things up properly and return the path so that the loop repeats, calling link_path_walk() again. This could loop as many as 40 times if the last component of each symlink is another symlink.

Of the various functions that examine the final component, open_last_lookups() is the most interesting as it works in tandem with do_open() for opening a file. Part of open_last_lookups() runs with `i_rwsem` held and this part is in a separate function: lookup_open().

Explaining open_last_lookups() and do_open() completely is beyond the scope of this article, but a few highlights should help those interested in exploring the code.

1. Rather than just finding the target file, do_open() is used after open_last_lookup() to open it. If the file was found in the dcache, then `vfs_open()` is used for this. If not, then `lookup_open()` will either call `atomic_open()` (if the filesystem provides it) to combine the final lookup with the open, or will perform the separate `i_op->lookup()` and `i_op->create()` steps directly. In the later case the actual "open" of this newly found or created file will be performed by vfs_open(), just as if the name were found in the dcache.
2. vfs_open() can fail with `-EOPENSTALE` if the cached information wasn't quite current enough. If it's in RCU-walk `-ECHILD` will be returned otherwise `-ESTALE` is returned. When `-ESTALE` is returned, the caller may retry with `LOOKUP_REVAL` flag set.
3. An open with O_CREAT **does** follow a symlink in the final component, unlike other creation system calls (like `mkdir`). So the sequence:

   ```
   ln -s bar /tmp/foo
   echo hello > /tmp/foo
   ```

   will create a file called `/tmp/bar`. This is not permitted if `O_EXCL` is set but otherwise is handled for an O_CREAT open much like for a non-creating open: lookup_last() or open_last_lookup() returns a non `NULL` value, and link_path_walk() gets called and the open process continues on the symlink that was found.

Updating the access time

We previously said of RCU-walk that it would "take no locks, increment no counts, leave no footprints." We have since seen that some "footprints" can be needed when handling symlinks as a counted reference (or even a memory allocation) may be needed. But these footprints are best kept to a minimum.

One other place where walking down a symlink can involve leaving footprints in a way that doesn't affect directories is in updating access times. In Unix (and Linux) every filesystem object has a "last accessed time", or "`atime`". Passing through a directory to access a file within is not considered to be an access for the purposes of `atime`; only listing the contents of a directory can update its `atime`. Symlinks are different it seems. Both reading a symlink (with `readlink()`) and looking up a symlink on the way to some other destination can update the atime on that symlink.

It is not clear why this is the case; POSIX has little to say on the subject. The clearest statement is that, if a particular implementation updates a timestamp in a place not specified by POSIX, this must be documented "except that any changes caused by pathname resolution need not be documented". This seems to imply that POSIX doesn't really care about access-time updates during pathname lookup.

An examination of history shows that prior to Linux 1.3.87, the ext2 filesystem, at least, didn't update atime when following a link. Unfortunately we have no record of why that behavior was changed.

In any case, access time must now be updated and that operation can be quite complex. Trying to stay in RCU-walk while doing it is best avoided. Fortunately it is often permitted to skip the `atime` update. Because `atime` updates cause performance problems in various areas, Linux supports the `relatime` mount option, which generally limits the updates of `atime` to once per day on files that aren't being changed (and symlinks never change once created). Even without `relatime`, many filesystems record `atime` with a one-second granularity, so only one update per second is required.

It is easy to test if an `atime` update is needed while in RCU-walk mode and, if it isn't, the update can be skipped and RCU-walk mode continues. Only when an `atime` update is actually required does the path walk drop down to REF-walk. All of this is handled in the `get_link()` function.

A few flags

A suitable way to wrap up this tour of pathname walking is to list the various flags that can be stored in the `nameidata` to guide the lookup process. Many of these are only meaningful on the final component, others reflect the current state of the pathname lookup, and some apply restrictions to all path components encountered in the path lookup.

And then there is `LOOKUP_EMPTY`, which doesn't fit conceptually with the others. If this is not set, an empty pathname causes an error very early on. If it is set, empty pathnames are not considered to be an error.

Global state flags

We have already met two global state flags: `LOOKUP_RCU` and `LOOKUP_REVAL`. These select between one of three overall approaches to lookup: RCU-walk, REF-walk, and REF-walk with forced revalidation.

`LOOKUP_PARENT` indicates that the final component hasn't been reached yet. This is primarily used to tell the audit subsystem the full context of a particular access being audited.

`ND_ROOT_PRESET` indicates that the `root` field in the `nameidata` was provided by the caller, so it shouldn't be released when it is no longer needed.

`ND_JUMPED` means that the current dentry was chosen not because it had the right name but for some other reason. This happens when following "`..`", following a symlink to `/`, crossing a mount point or accessing a "`/proc/$PID/fd/$FD`" symlink (also known as a "magic link"). In this case the filesystem has not been asked to revalidate the name (with `d_revalidate()`). In such cases the inode may still need to be revalidated, so `d_op->d_weak_revalidate()` is called if `ND_JUMPED` is set when the look completes -

which may be at the final component or, when creating, unlinking, or renaming, at the penultimate component.

Resolution-restriction flags

In order to allow userspace to protect itself against certain race conditions and attack scenarios involving changing path components, a series of flags are available which apply restrictions to all path components encountered during path lookup. These flags are exposed through `openat2()`'s `resolve` field.

`LOOKUP_NO_SYMLINKS` blocks all symlink traversals (including magic-links). This is distinctly different from `LOOKUP_FOLLOW`, because the latter only relates to restricting the following of trailing symlinks.

`LOOKUP_NO_MAGICLINKS` blocks all magic-link traversals. Filesystems must ensure that they return errors from `nd_jump_link()`, because that is how `LOOKUP_NO_MAGICLINKS` and other magic-link restrictions are implemented.

`LOOKUP_NO_XDEV` blocks all `vfsmount` traversals (this includes both bind-mounts and ordinary mounts). Note that the `vfsmount` which contains the lookup is determined by the first mountpoint the path lookup reaches -- absolute paths start with the `vfsmount` of `/`, and relative paths start with the `dfd`'s `vfsmount`. Magic-links are only permitted if the `vfsmount` of the path is unchanged.

`LOOKUP_BENEATH` blocks any path components which resolve outside the starting point of the resolution. This is done by blocking `nd_jump_root()` as well as blocking ".." if it would jump outside the starting point. `rename_lock` and `mount_lock` are used to detect attacks against the resolution of "..". Magic-links are also blocked.

`LOOKUP_IN_ROOT` resolves all path components as though the starting point were the filesystem root. `nd_jump_root()` brings the resolution back to the starting point, and ".." at the starting point will act as a no-op. As with `LOOKUP_BENEATH`, `rename_lock` and `mount_lock` are used to detect attacks against ".." resolution. Magic-links are also blocked.

Final-component flags

Some of these flags are only set when the final component is being considered. Others are only checked for when considering that final component.

`LOOKUP_AUTOMOUNT` ensures that, if the final component is an automount point, then the mount is triggered. Some operations would trigger it anyway, but operations like `stat()` deliberately don't. `statfs()` needs to trigger the mount but otherwise behaves a lot like `stat()`, so it sets `LOOKUP_AUTOMOUNT`, as does "`quotactl()`" and the handling of "`mount --bind`".

`LOOKUP_FOLLOW` has a similar function to `LOOKUP_AUTOMOUNT` but for symlinks. Some system calls set or clear it implicitly, while others have API flags such as `AT_SYMLINK_FOLLOW` and `UMOUNT_NOFOLLOW` to control it. Its effect is similar to `WALK_GET` that we already met, but it is used in a different way.

`LOOKUP_DIRECTORY` insists that the final component is a directory. Various callers set this and it is also set when the final component is found to be followed by a slash.

Finally `LOOKUP_OPEN`, `LOOKUP_CREATE`, `LOOKUP_EXCL`, and `LOOKUP_RENAME_TARGET` are not used directly by the VFS but are made available to the filesystem and particularly the `->d_revalidate()` method. A filesystem can choose not to bother revalidating too hard if it knows that it will be asked to open or create the file soon. These flags were previously useful for `->lookup()` too but with the introduction of `->atomic_open()` they are less relevant there.

End of the road

Despite its complexity, all this pathname lookup code appears to be in good shape - various parts are certainly easier to understand now than even a couple of releases ago. But that doesn't mean it is "finished". As already mentioned, RCU-walk currently only follows symlinks that are stored in the inode so, while it handles many ext4 symlinks, it doesn't help with NFS, XFS, or Btrfs. That support is not likely to be long delayed.