Changes since 2.5.0:

---

recommended

New helpers: sb_bread(), sb_getblk(), sb_find_get_block(), set_bh(), sb_set_blocksize() and sb_min_blocksize().

Use them.

(sb_find_get_block() replaces 2.4's get_hash_table())

---

recommended

New methods: ->alloc_inode() and ->destroy_inode().

Remove inode->u.foo_inode_i

Declare:

```
struct foo_inode_info {
        /* fs-private stuff */
        struct inode vfs_inode;
};
static inline struct foo_inode_info *FOO_I(struct inode *inode)
{
        return list_entry(inode, struct foo_inode_info, vfs_inode);
}
```

Use FOO_I(inode) instead of &inode->u.foo_inode_i;

Add foo_alloc_inode() and foo_destroy_inode() - the former should allocate foo_inode_info and return the address of ->vfs_inode, the latter should free FOO_I(inode) (see in-tree filesystems for examples).

Make them ->alloc_inode and ->destroy_inode in your super_operations.

Keep in mind that now you need explicit initialization of private data typically between calling iget_locked() and unlocking the inode.

At some point that will become mandatory.

mandatory

The foo_inode_info should always be allocated through alloc_inode_sb() rather than kmem_cache_alloc() or kmalloc() related to set up the inode reclaim context correctly.

---

mandatory

Change of file_system_type method (->read_super to ->get_sb)

->read_super() is no more. Ditto for DECLARE_FSTYPE and DECLARE_FSTYPE_DEV.

Turn your foo_read_super() into a function that would return 0 in case of success and negative number in case of error (-EINVAL unless you have more informative error value to report). Call it foo_fill_super(). Now declare:

```
int foo_get_sb(struct file_system_type *fs_type,
      int flags, const char *dev_name, void *data, struct vfsmount *mnt)
{
      return get_sb_bdev(fs_type, flags, dev_name, data, foo_fill_super,
                         mnt);
}
```

(or similar with s/bdev/nodev/ or s/bdev/single/, depending on the kind of filesystem).

Replace DECLARE_FSTYPE... with explicit initializer and have ->get_sb set as foo_get_sb.

---

mandatory

Locking change: ->s_vfs_rename_sem is taken only by cross-directory renames. Most likely there is no need to change anything, but if you relied on global exclusion between renames for some internal purpose - you need to change your internal locking. Otherwise exclusion warranties remain the same (i.e. parents and victim are locked, etc.).

---

informational

Now we have the exclusion between ->lookup() and directory removal (by ->rmdir() and ->rename()). If you used to need that exclusion and do it by internal locking (most of filesystems couldn't care less) - you can relax your locking.

---

mandatory

->lookup(), ->truncate(), ->create(), ->unlink(), ->mknod(), ->mkdir(), ->rmdir(), ->link(), ->lseek(), ->symlink(), ->rename() and ->readdir() are called without BKL now. Grab it on entry, drop upon return - that will guarantee the same locking you used to have. If your method or its parts do not need BKL - better yet, now you can shift lock_kernel() and unlock_kernel() so that they would protect exactly what needs to be protected.

---

mandatory

BKL is also moved from around sb operations. BKL should have been shifted into individual fs sb_op functions. If you don't need it, remove it.

---

informational

check for ->link() target not being a directory is done by callers. Feel free to drop it...

---

informational

->link() callers hold ->i_mutex on the object we are linking to. Some of your problems might be over...

---

mandatory

new file_system_type method - kill_sb(superblock). If you are converting an existing filesystem, set it according to ->fs_flags:

```
FS_REQUIRES_DEV         -       kill_block_super
FS_LITTER               -       kill_litter_super
neither                 -       kill_anon_super
```

FS_LITTER is gone - just remove it from fs_flags.

---

mandatory

FS_SINGLE is gone (actually, that had happened back when ->get_sb() went in - and hadn't been documented ;-/). Just remove it from fs_flags (and see ->get_sb() entry for other actions).

---

mandatory

->setattr() is called without BKL now. Caller _always_ holds ->i_mutex, so watch for ->i_mutex-grabbing code that might be used by your ->setattr(). Callers of notify_change() need ->i_mutex now.

---

recommended

New super_block field `struct export_operations *s_export_op` for explicit support for exporting, e.g. via NFS. The structure is fully documented at its declaration in include/linux/fs.h, and in Documentation/filesystems/nfs/exporting.rst.

Briefly it allows for the definition of decode_fh and encode_fh operations to encode and decode filehandles, and allows the filesystem to use a standard helper function for decode_fh, and provide file-system specific support for this helper, particularly get_parent.

It is planned that this will be required for exporting once the code settles down a bit.

mandatory

s_export_op is now required for exporting a filesystem. isofs, ext2, ext3, resierfs, fat can be used as examples of very different filesystems.

---

mandatory

iget4() and the read_inode2 callback have been superseded by iget5_locked() which has the following prototype:

```
struct inode *iget5_locked(struct super_block *sb, unsigned long ino,
                           int (*test)(struct inode *, void *),
                           int (*set)(struct inode *, void *),
                           void *data);
```

'test' is an additional function that can be used when the inode number is not sufficient to identify the actual file object. 'set' should be a non-blocking function that initializes those parts of a newly created inode to allow the test function to succeed. 'data' is passed as an opaque value to both test and set functions.

When the inode has been created by iget5_locked(), it will be returned with the I_NEW flag set and will still be locked. The filesystem then needs to finalize the initialization. Once the inode is initialized it must be unlocked by calling unlock_new_inode().

The filesystem is responsible for setting (and possibly testing) i_ino when appropriate. There is also a simpler iget_locked function that just takes the superblock and inode number as arguments and does the test and set for you.

e.g.:

```
inode = iget_locked(sb, ino);
if (inode->i_state & I_NEW) {
        err = read_inode_from_disk(inode);
        if (err < 0) {
                iget_failed(inode);
                return err;
        }
        unlock_new_inode(inode);
}
```

Note that if the process of setting up a new inode fails, then iget_failed() should be called on the inode to render it dead, and an appropriate error should be passed back to the caller.

---

recommended

->getattr() finally getting used. See instances in nfs, minix, etc.

---

mandatory

->revalidate() is gone. If your filesystem had it - provide ->getattr() and let it call whatever you had as ->revlidate() + (for symlinks that had ->revalidate()) add calls in ->follow_link()/->readlink().

---

mandatory

->d_parent changes are not protected by BKL anymore. Read access is safe if at least one of the following is true:

- filesystem has no cross-directory rename()
- we know that parent had been locked (e.g. we are looking at ->d_parent of ->lookup() argument).
- we are called from ->rename().
- the child's ->d_lock is held

Audit your code and add locking if needed. Notice that any place that is not protected by the conditions above is risky even in the old tree - you had been relying on BKL and that's prone to screwups. Old tree had quite a few holes of that kind - unprotected access to ->d_parent leading to anything from oops to silent memory corruption.

---

mandatory

FS_NOMOUNT is gone. If you use it - just set SB_NOUSER in flags (see rootfs for one kind of solution and bdev/socket/pipe for another).

---

recommended

Use bdev_read_only(bdev) instead of is_read_only(kdev). The latter is still alive, but only because of the mess in drivers/s390/block/dasd.c. As soon as it gets fixed is_read_only() will die.

---

mandatory

->permission() is called without BKL now. Grab it on entry, drop upon return - that will guarantee the same locking you used to have. If your method or its parts do not need BKL - better yet, now you can shift lock_kernel() and unlock_kernel() so that they would protect exactly what needs to be protected.

---

mandatory

->statfs() is now called without BKL held. BKL should have been shifted into individual fs sb_op functions where it's not clear that it's safe to remove it. If you don't need it, remove it.

---

mandatory

is_read_only() is gone; use bdev_read_only() instead.

---

mandatory

destroy_buffers() is gone; use invalidate_bdev().

---

mandatory

fsync_dev() is gone; use fsync_bdev(). NOTE: lvm breakage is deliberate; as soon as struct block_device * is propagated in a reasonable way by that code fixing will become trivial; until then nothing can be done.

mandatory

block truncatation on error exit from ->write_begin, and ->direct_IO moved from generic methods (block_write_begin, cont_write_begin, nobh_write_begin, blockdev_direct_IO*) to callers. Take a look at ext2_write_failed and callers for an example.

mandatory

->truncate is gone. The whole truncate sequence needs to be implemented in ->setattr, which is now mandatory for filesystems implementing on-disk size changes. Start with a copy of the old inode_setattr and vmtruncate, and the reorder the vmtruncate + foofs_vmtruncate sequence to be in order of zeroing blocks using block_truncate_page or similar helpers, size update and on finally on-disk truncation which should not fail. setattr_prepare (which used to be inode_change_ok) now includes the size checks for ATTR_SIZE and must be called in the beginning of ->setattr unconditionally.

mandatory

->clear_inode() and ->delete_inode() are gone; ->evict_inode() should be used instead. It gets called whenever the inode is evicted, whether it has remaining links or not. Caller does *not* evict the pagecache or inode-associated metadata buffers; the method has to use truncate_inode_pages_final() to get rid of those. Caller makes sure async writeback cannot be running for the inode while (or after) ->evict_inode() is called.

->drop_inode() returns int now; it's called on final iput() with inode->i_lock held and it returns true if filesystems wants the inode to be dropped. As before, generic_drop_inode() is still the default and it's been updated appropriately. generic_delete_inode() is also alive and it consists simply of return 1. Note that all actual eviction work is done by caller after ->drop_inode() returns.

As before, clear_inode() must be called exactly once on each call of ->evict_inode() (as it used to be for each call of ->delete_inode()). Unlike before, if you are using inode-associated metadata buffers (i.e. mark_buffer_dirty_inode()), it's your responsibility to call invalidate_inode_buffers() before clear_inode().

NOTE: checking i_nlink in the beginning of ->write_inode() and bailing out if it's zero is not *and never had been* enough. Final unlink() and iput() may happen while the inode is in the middle of ->write_inode(); e.g. if you blindly free the on-disk inode, you may end up doing that while ->write_inode() is writing to it.

---

mandatory

.d_delete() now only advises the dcache as to whether or not to cache unreferenced dentries, and is now only called when the dentry refcount goes to 0. Even on 0 refcount transition, it must be able to tolerate being called 0, 1, or more times (eg. constant, idempotent).

---

mandatory

.d_compare() calling convention and locking rules are significantly changed. Read updated documentation in Documentation/filesystems/vfs.rst (and look at examples of other filesystems) for guidance.

---

mandatory

.d_hash() calling convention and locking rules are significantly changed. Read updated documentation in Documentation/filesystems/vfs.rst (and look at examples of other filesystems) for guidance.

---

mandatory

dcache_lock is gone, replaced by fine grained locks. See fs/dcache.c for details of what locks to replace dcache_lock with in order to protect particular things. Most of the time, a filesystem only needs ->d_lock, which protects *all* the dcache state of a given dentry.

---

mandatory

Filesystems must RCU-free their inodes, if they can have been accessed via rcu-walk path walk (basically, if the file can have had a path name in the vfs namespace).

Even though i_dentry and i_rcu share storage in a union, we will initialize the former in inode_init_always(), so just leave it alone in the callback. It used to be necessary to clean it there, but not anymore (starting at 3.2).

---

recommended

vfs now tries to do path walking in "rcu-walk mode", which avoids atomic operations and scalability hazards on dentries and inodes (see Documentation/filesystems/path-lookup.txt). d_hash and d_compare changes (above) are examples of the changes required to support this. For more complex filesystem callbacks, the vfs drops out of rcu-walk mode before the fs call, so no changes are required to the filesystem. However, this is costly and loses the benefits of rcu-walk mode. We will begin to add filesystem callbacks that are rcu-walk aware, shown below. Filesystems should take advantage of this where possible.

---

mandatory

d_revalidate is a callback that is made on every path element (if the filesystem provides it), which requires dropping out of rcu-walk mode. This may now be called in rcu-walk mode (nd->flags & LOOKUP_RCU). -ECHILD should be returned if the filesystem cannot handle rcu-walk. See Documentation/filesystems/vfs.rst for more details.

permission is an inode permission check that is called on many or all directory inodes on the way down a path walk (to check for exec permission). It must now be rcu-walk aware (mask & MAY_NOT_BLOCK). See Documentation/filesystems/vfs.rst for more details.

---

mandatory

In ->fallocate() you must check the mode option passed in. If your filesystem does not support hole punching (deallocating space in the middle of a file) you must return -EOPNOTSUPP if FALLOC_FL_PUNCH_HOLE is set in mode. Currently you can only have FALLOC_FL_PUNCH_HOLE with FALLOC_FL_KEEP_SIZE set, so the i_size should not change when hole punching, even when puching the end of a file off.

---

mandatory

->get_sb() is gone. Switch to use of ->mount(). Typically it's just a matter of switching from calling `get_sb_...` to `mount_...` and changing the function type. If you were doing it manually, just switch from setting ->mnt_root to some pointer to returning that pointer. On errors return ERR_PTR(...).

---

mandatory

->permission() and generic_permission()have lost flags argument; instead of passing IPERM_FLAG_RCU we add MAY_NOT_BLOCK into mask.

generic_permission() has also lost the check_acl argument; ACL checking has been taken to VFS and filesystems need to provide a non-NULL ->i_op->get_acl to read an ACL from disk.

---

mandatory

If you implement your own ->llseek() you must handle SEEK_HOLE and SEEK_DATA. You can hanle this by returning -EINVAL, but it would be nicer to support it in some way. The generic handler assumes that the entire file is data and there is a virtual hole at the end of the file. So if the provided offset is less than i_size and SEEK_DATA is specified, return the same offset. If the above is true for the offset and you are given SEEK_HOLE, return the end of the file. If the offset is i_size or greater return -ENXIO in either case.

mandatory

If you have your own ->fsync() you must make sure to call filemap_write_and_wait_range() so that all dirty pages are synced out properly. You must also keep in mind that ->fsync() is not called with i_mutex held anymore, so if you require i_mutex locking you must make sure to take it and release it yourself.

---

mandatory

d_alloc_root() is gone, along with a lot of bugs caused by code misusing it. Replacement: d_make_root(inode). On success d_make_root(inode) allocates and returns a new dentry instantiated with the passed in inode. On failure NULL is returned and the passed in inode is dropped so the reference to inode is consumed in all cases and failure handling need not do any cleanup for the inode. If d_make_root(inode) is passed a NULL inode it returns NULL and also requires no further error handling. Typical usage is:

```
    inode = foofs_new_inode(....);
    s->s_root = d_make_root(inode);
    if (!s->s_root)
            /* Nothing needed for the inode cleanup */
            return -ENOMEM;
    ...
```

---

mandatory

The witch is dead! Well, 2/3 of it, anyway. ->d_revalidate() and ->lookup() do *not* take struct nameidata anymore; just the flags.

---

mandatory

->create() doesn't take `struct nameidata *`; unlike the previous two, it gets "is it an O_EXCL or equivalent?" boolean argument. Note that local filesystems can ignore tha argument - they are guaranteed that the object doesn't exist. It's remote/distributed ones that might care...

---

mandatory

FS_REVAL_DOT is gone; if you used to have it, add ->d_weak_revalidate() in your dentry operations instead.

---

mandatory

vfs_readdir() is gone; switch to iterate_dir() instead

---

mandatory

->readdir() is gone now; switch to ->iterate()

mandatory

vfs_follow_link has been removed. Filesystems must use nd_set_link from ->follow_link for normal symlinks, or nd_jump_link for magic /proc/<pid> style links.

---

mandatory

iget5_locked()/ilookup5()/ilookup5_nowait() test() callback used to be called with both ->i_lock and inode_hash_lock held; the former is *not* taken anymore, so verify that your callbacks do not rely on it (none of the in-tree instances did). inode_hash_lock is still held, of course, so they are still serialized wrt removal from inode hash, as well as wrt set() callback of iget5_locked().

---

mandatory

d_materialise_unique() is gone; d_splice_alias() does everything you need now. Remember that they have opposite orders of arguments ;-/

---

mandatory

f_dentry is gone; use f_path.dentry, or, better yet, see if you can avoid it entirely.

---

mandatory

never call ->read() and ->write() directly; use __vfs_{read,write} or wrappers; instead of checking for ->write or ->read being NULL, look for FMODE_CAN_{WRITE,READ} in file->f_mode.

---

mandatory

do _not_ use new_sync_{read,write} for ->read/->write; leave it NULL instead.

---

mandatory

->aio_read/->aio_write are gone. Use ->read_iter/->write_iter.

---

recommended

for embedded ("fast") symlinks just set inode->i_link to wherever the symlink body is and use simple_follow_link() as ->follow_link().

---

mandatory

calling conventions for ->follow_link() have changed. Instead of returning cookie and using nd_set_link() to store the body to

traverse, we return the body to traverse and store the cookie using explicit void ** argument. nameidata isn't passed at all - nd_jump_link() doesn't need it and nd_[gs]et_link() is gone.

---

mandatory

calling conventions for ->put_link() have changed. It gets inode instead of dentry, it does not get nameidata at all and it gets called only when cookie is non-NULL. Note that link body isn't available anymore, so if you need it, store it as cookie.

---

mandatory

any symlink that might use page_follow_link_light/page_put_link() must have inode_nohighmem(inode) called before anything might start playing with its pagecache. No highmem pages should end up in the pagecache of such symlinks. That includes any preseeding that might be done during symlink creation. __page_symlink() will honour the mapping gfp flags, so once you've done inode_nohighmem() it's safe to use, but if you allocate and insert the page manually, make sure to use the right gfp flags.

---

mandatory

->follow_link() is replaced with ->get_link(); same API, except that

- ->get_link() gets inode as a separate argument
- ->get_link() may be called in RCU mode - in that case NULL dentry is passed

---

mandatory

->get_link() gets struct delayed_call `*done` now, and should do set_delayed_call() where it used to set `*cookie`.

->put_link() is gone - just give the destructor to set_delayed_call() in ->get_link().

---

mandatory

->getxattr() and xattr_handler.get() get dentry and inode passed separately. dentry might be yet to be attached to inode, so do _not_ use its ->d_inode in the instances. Rationale: !@#!@# security_d_instantiate() needs to be called before we attach dentry to inode.

---

mandatory

symlinks are no longer the only inodes that do *not* have i_bdev/i_cdev/ i_pipe/i_link union zeroed out at inode eviction. As the result, you can't assume that non-NULL value in ->i_nlink at ->destroy_inode() implies that it's a symlink. Checking ->i_mode is really needed now. In-tree we had to fix shmem_destroy_callback() that used to take that kind of shortcut; watch out, since that shortcut is no longer valid.

---

mandatory

->i_mutex is replaced with ->i_rwsem now. inode_lock() et.al. work as they used to - they just take it exclusive. However, ->lookup() may be called with parent locked shared. Its instances must not

- use d_instantiate) and d_rehash() separately - use d_add() or d_splice_alias() instead.
- use d_rehash() alone - call d_add(new_dentry, NULL) instead.
- in the unlikely case when (read-only) access to filesystem data structures needs exclusion for some reason, arrange it yourself. None of the in-tree filesystems needed that.
- rely on ->d_parent and ->d_name not changing after dentry has been fed to d_add() or d_splice_alias(). Again, none of the in-tree instances relied upon that.

We are guaranteed that lookups of the same name in the same directory will not happen in parallel ("same" in the sense of your ->d_compare()). Lookups on different names in the same directory can and do happen in parallel now.

---

recommended

->iterate_shared() is added; it's a parallel variant of ->iterate(). Exclusion on struct file level is still provided (as well as that between it and lseek on the same struct file), but if your directory has been opened several times, you can get these called in parallel. Exclusion between that method and all directory-modifying ones is still provided, of course.

Often enough ->iterate() can serve as ->iterate_shared() without any changes - it is a read-only operation, after all. If you have any per-inode or per-dentry in-core data structures modified by ->iterate(), you might need something to serialize the access to them. If you do dcache pre-seeding, you'll need to switch to d_alloc_parallel() for that; look for in-tree examples.

Old method is only used if the new one is absent; eventually it will be removed. Switch while you still can; the old one won't stay.

---

mandatory

->atomic_open() calls without O_CREAT may happen in parallel.

---

mandatory

->setxattr() and xattr_handler.set() get dentry and inode passed separately. The xattr_handler.set() gets passed the user namespace of the mount the inode is seen from so filesystems can idmap the i_uid and i_gid accordingly. dentry might be yet to be attached to inode, so do _not_ use its ->d_inode in the instances. Rationale: !@#!@# security_d_instantiate() needs to be called before we attach dentry to inode and !@#!@##!@$!$#!@#$!@$!@$ smack ->d_instantiate() uses not just ->getxattr() but ->setxattr() as well.

---

mandatory

->d_compare() doesn't get parent as a separate argument anymore. If you used it for finding the struct super_block involved, dentry->d_sb will work just as well; if it's something more complicated, use dentry->d_parent. Just be careful not to assume that fetching it more than once will yield the same value - in RCU mode it could change under you.

---

mandatory

->rename() has an added flags argument. Any flags not handled by the filesystem should result in EINVAL being returned.

---

recommended

->readlink is optional for symlinks. Don't set, unless filesystem needs to fake something for readlink(2).

---

mandatory

->getattr() is now passed a struct path rather than a vfsmount and dentry separately, and it now has request_mask and query_flags arguments to specify the fields and sync type requested by statx. Filesystems not supporting any statx-specific features may ignore the new arguments.

---

mandatory

->atomic_open() calling conventions have changed. Gone is `int *opened`, along with FILE_OPENED/FILE_CREATED. In place of those we have FMODE_OPENED/FMODE_CREATED, set in file->f_mode. Additionally, return value for 'called finish_no_open(), open it yourself' case has become 0, not 1. Since finish_no_open() itself is returning 0 now, that part does not need any changes in ->atomic_open() instances.

---

mandatory

alloc_file() has become static now; two wrappers are to be used instead. alloc_file_pseudo(inode, vfsmount, name, flags, ops) is for the cases when dentry needs to be created; that's the majority of old alloc_file() users. Calling conventions: on success a reference to new struct file is returned and callers reference to inode is subsumed by that. On failure, ERR_PTR() is returned and no caller's references are affected, so the caller needs to drop the inode reference it held. alloc_file_clone(file, flags, ops) does not affect any caller's references. On success you get a new struct file sharing the mount/dentry with the original, on failure - ERR_PTR().

---

mandatory

->clone_file_range() and ->dedupe_file_range have been replaced with ->remap_file_range(). See Documentation/filesystems/vfs.rst for more information.

---

recommended

->lookup() instances doing an equivalent of:

```
if (IS_ERR(inode))
        return ERR_CAST(inode);
return d_splice_alias(inode, dentry);
```

don't need to bother with the check - d_splice_alias() will do the right thing when given ERR_PTR(...) as inode. Moreover, passing NULL inode to d_splice_alias() will also do the right thing (equivalent of d_add(dentry, NULL); return NULL;), so that kind of

special cases also doesn't need a separate treatment.

---

strongly recommended

take the RCU-delayed parts of ->destroy_inode() into a new method - ->free_inode(). If ->destroy_inode() becomes empty - all the better, just get rid of it. Synchronous work (e.g. the stuff that can't be done from an RCU callback, or any WARN_ON() where we want the stack trace) *might* be movable to ->evict_inode(); however, that goes only for the things that are not needed to balance something done by ->alloc_inode(). IOW, if it's cleaning up the stuff that might have accumulated over the life of in-core inode, ->evict_inode() might be a fit.

Rules for inode destruction:

- if ->destroy_inode() is non-NULL, it gets called
- if ->free_inode() is non-NULL, it gets scheduled by call_rcu()
- combination of NULL ->destroy_inode and NULL ->free_inode is treated as NULL/free_inode_nonrcu, to preserve the compatibility.

Note that the callback (be it via ->free_inode() or explicit call_rcu() in ->destroy_inode()) is *NOT* ordered wrt superblock destruction; as the matter of fact, the superblock and all associated structures might be already gone. The filesystem driver is guaranteed to be still there, but that's it. Freeing memory in the callback is fine; doing more than that is possible, but requires a lot of care and is best avoided.

---

mandatory

DCACHE_RCUACCESS is gone; having an RCU delay on dentry freeing is the default. DCACHE_NORCU opts out, and only d_alloc_pseudo() has any business doing so.

---

mandatory

d_alloc_pseudo() is internal-only; uses outside of alloc_file_pseudo() are very suspect (and won't work in modules). Such uses are very likely to be misspelled d_alloc_anon().

---

mandatory

[should've been added in 2016] stale comment in finish_open() nonwithstanding, failure exits in ->atomic_open() instances should *NOT* fput() the file, no matter what. Everything is handled by the caller.

---

mandatory

clone_private_mount() returns a longterm mount now, so the proper destructor of its result is kern_unmount() or kern_unmount_array().

---

mandatory

zero-length bvec segments are disallowed, they must be filtered out before passed on to an iterator.

---

mandatory

For bvec based itererators bio_iov_iter_get_pages() now doesn't copy bvecs but uses the one provided. Anyone issuing kiocb-I/O should ensure that the bvec and page references stay until I/O has completed, i.e. until ->ki_complete() has been called or returned with non -EIOCBQUEUED code.

---

mandatory

mnt_want_write_file() can now only be paired with mnt_drop_write_file(), whereas previously it could be paired with mnt_drop_write() as well.

---

mandatory

iov_iter_copy_from_user_atomic() is gone; use copy_page_from_iter_atomic(). The difference is copy_page_from_iter_atomic() advances the iterator and you don't need iov_iter_advance() after it. However, if you decide to use only a part of obtained data, you should do iov_iter_revert().

---

mandatory

Calling conventions for file_open_root() changed; now it takes struct path * instead of passing mount and dentry separately. For callers that used to pass <mnt, mnt->mnt_root> pair (i.e. the root of given mount), a new helper is provided - file_open_root_mnt(). In-tree users adjusted.