Locking

The text below describes the locking rules for VFS-related methods. It is (believed to be) up-to-date. *Please*, if you change anything in prototypes or locking protocols - update this file. And update the relevant instances in the tree, don't leave that to maintainers of filesystems/devices/ etc. At the very least, put the list of dubious cases in the end of this file. Don't turn it into log - maintainers of out-of-the-tree code are supposed to be able to use diff(1).

Thing currently missing here: socket operations. Alexey?

dentry_operations

prototypes:

```
int (*d_revalidate)(struct dentry *, unsigned int);
int (*d_weak_revalidate)(struct dentry *, unsigned int);
int (*d_hash)(const struct dentry *, struct qstr *);
int (*d_compare)(const struct dentry *,
                 unsigned int, const char *, const struct qstr *);
int (*d_delete)(struct dentry *);
int (*d_init)(struct dentry *);
void (*d_release)(struct dentry *);
void (*d_iput)(struct dentry *, struct inode *);
char *(*d_dname)((struct dentry *dentry, char *buffer, int buflen);
struct vfsmount *(*d_automount)(struct path *path);
int (*d_manage)(const struct path *, bool);
struct dentry *(*d_real)(struct dentry *, const struct inode *);
```

locking rules:

ops	rename_lock	->d_lock	may block	rcu-walk
d_revalidate:	no	no	yes (ref-walk)	maybe
d_weak_revalidate:	no	no	yes	no
d_hash	no	no	no	maybe
d_compare:	yes	no	no	maybe
d_delete:	no	yes	no	no
d_init:	no	no	yes	no
d_release:	no	no	yes	no
d_prune:	no	yes	no	no
d_iput:	no	no	yes	no
d_dname:	no	no	no	no
d_automount:	no	no	yes	no
d_manage:	no	no	yes (ref-walk)	maybe
d_real	no	no	yes	no

inode_operations

prototypes:

```
int (*create) (struct inode *,struct dentry *,umode_t, bool);
struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
int (*link) (struct dentry *,struct inode *,struct dentry *);
int (*unlink) (struct inode *,struct dentry *);
int (*symlink) (struct inode *,struct dentry *,const char *);
int (*mkdir) (struct inode *,struct dentry *,umode_t);
int (*rmdir) (struct inode *,struct dentry *);
int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
int (*rename) (struct inode *, struct dentry *,
               struct inode *, struct dentry *, unsigned int);
int (*readlink) (struct dentry *, char __user *,int);
const char *(*get_link) (struct dentry *, struct inode *, struct delayed_call *);
void (*truncate) (struct inode *);
int (*permission) (struct inode *, int, unsigned int);
struct posix_acl * (*get_acl)(struct inode *, int, bool);
int (*setattr) (struct dentry *, struct iattr *);
int (*getattr) (const struct path *, struct kstat *, u32, unsigned int);
ssize_t (*listxattr) (struct dentry *, char *, size_t);
int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start, u64 len);
void (*update_time)(struct inode *, struct timespec *, int);
int (*atomic_open)(struct inode *, struct dentry *,
                        struct file *, unsigned open_flag,
                        umode_t create_mode);
int (*tmpfile) (struct inode *, struct dentry *, umode_t);
```

```
int (*fileattr_set)(struct user_namespace *mnt_userns,
                    struct dentry *dentry, struct fileattr *fa);
int (*fileattr_get)(struct dentry *dentry, struct fileattr *fa);
```

locking rules:

all may block

ops	i_rwsem(inode)
lookup:	shared
create:	exclusive
link:	exclusive (both)
mknod:	exclusive
symlink:	exclusive
mkdir:	exclusive
unlink:	exclusive (both)
rmdir:	exclusive (both)(see below)
rename:	exclusive (all) (see below)
readlink:	no
get_link:	no
setattr:	exclusive
permission:	no (may not block if called in rcu-walk mode)
get_acl:	no
getattr:	no
listxattr:	no
fiemap:	no
update_time:	no
atomic_open:	shared (exclusive if O_CREAT is set in open flags)
tmpfile:	no
fileattr_get:	no or exclusive
fileattr_set:	exclusive

Additionally, ->rmdir(), ->unlink() and ->rename() have ->i_rwsem exclusive on victim. cross-directory ->rename() has (per-superblock) ->s_vfs_rename_sem.

See Documentation/filesystems/directory-locking.rst for more detailed discussion of the locking scheme for directory operations.

xattr_handler operations

prototypes:

```
bool (*list)(struct dentry *dentry);
int (*get)(const struct xattr_handler *handler, struct dentry *dentry,
           struct inode *inode, const char *name, void *buffer,
           size_t size);
int (*set)(const struct xattr_handler *handler,
           struct user_namespace *mnt_userns,
           struct dentry *dentry, struct inode *inode, const char *name,
           const void *buffer, size_t size, int flags);
```

locking rules:

all may block

ops	i_rwsem(inode)
list:	no
get:	no
set:	exclusive

super_operations

prototypes:

```
struct inode *(*alloc_inode)(struct super_block *sb);
void (*free_inode)(struct inode *);
void (*destroy_inode)(struct inode *);
void (*dirty_inode) (struct inode *, int flags);
int (*write_inode) (struct inode *, struct writeback_control *wbc);
int (*drop_inode) (struct inode *);
void (*evict_inode) (struct inode *);
void (*put_super) (struct super_block *);
int (*sync_fs)(struct super_block *sb, int wait);
```

```
int (*freeze_fs) (struct super_block *);
int (*unfreeze_fs) (struct super_block *);
int (*statfs) (struct dentry *, struct kstatfs *);
int (*remount_fs) (struct super_block *, int *, char *);
void (*umount_begin) (struct super_block *);
int (*show_options)(struct seq_file *, struct dentry *);
ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
```

locking rules:

All may block [not true, see below]

ops	s_umount	note
alloc_inode:		
free_inode:		called from RCU callback
destroy_inode:		
dirty_inode:		
write_inode:		
drop_inode:		!!!inode->i_lock!!!
evict_inode:		
put_super:	write	
sync_fs:	read	
freeze_fs:	write	
unfreeze_fs:	write	
statfs:	maybe(read)	(see below)
remount_fs:	write	
umount_begin:	no	
show_options:	no	(namespace_sem)
quota_read:	no	(see below)
quota_write:	no	(see below)

->statfs() has s_umount (shared) when called by ustat(2) (native or compat), but that's an accident of bad API; s_umount is used to pin the superblock down when we only have dev_t given us by userland to identify the superblock. Everything else (statfs(), fstatfs(), etc.) doesn't hold it when calling ->statfs() - superblock is pinned down by resolving the pathname passed to syscall.

->quota_read() and ->quota_write() functions are both guaranteed to be the only ones operating on the quota file by the quota code (via dqio_sem) (unless an admin really wants to screw up something and writes to quota files with quotas on). For other details about locking see also dquot_operations section.

file_system_type

prototypes:

```
struct dentry *(*mount) (struct file_system_type *, int,
              const char *, void *);
void (*kill_sb) (struct super_block *);
```

locking rules:

ops	may block
mount	yes
kill_sb	yes

->mount() returns ERR_PTR or the root dentry; its superblock should be locked on return.

->kill_sb() takes a write-locked superblock, does all shutdown work on it, unlocks and drops the reference.

address_space_operations

prototypes:

```
int (*writepage)(struct page *page, struct writeback_control *wbc);
int (*readpage)(struct file *, struct page *);
int (*writepages)(struct address_space *, struct writeback_control *);
bool (*dirty_folio)(struct address_space *, struct folio *folio);
void (*readahead)(struct readahead_control *);
int (*write_begin)(struct file *, struct address_space *mapping,
                        loff_t pos, unsigned len, unsigned flags,
                        struct page **pagep, void **fsdata);
int (*write_end)(struct file *, struct address_space *mapping,
                        loff_t pos, unsigned len, unsigned copied,
                        struct page *page, void *fsdata);
```

```
sector_t (*bmap)(struct address_space *, sector_t);
void (*invalidate_folio) (struct folio *, size_t start, size_t len);
int (*releasepage) (struct page *, int);
void (*freepage)(struct page *);
int (*direct_IO)(struct kiocb *, struct iov_iter *iter);
bool (*isolate_page) (struct page *, isolate_mode_t);
int (*migratepage)(struct address_space *, struct page *, struct page *);
void (*putback_page) (struct page *);
int (*launder_folio)(struct folio *);
bool (*is_partially_uptodate)(struct folio *, size_t from, size_t count);
int (*error_remove_page)(struct address_space *, struct page *);
int (*swap_activate)(struct file *);
int (*swap_deactivate)(struct file *);
```

locking rules:

All except dirty_folio and freepage may block

ops	**PageLocked(page)**	**i_rwsem**	**invalidate_lock**
writepage:	yes, unlocks (see below)		
readpage:	yes, unlocks		shared
writepages:			
dirty_folio	maybe		
readahead:	yes, unlocks		shared
write_begin:	locks the page	exclusive	
write_end:	yes, unlocks	exclusive	
bmap:			
invalidate_folio:	yes		exclusive
releasepage:	yes		
freepage:	yes		
direct_IO:			
isolate_page:	yes		
migratepage:	yes (both)		
putback_page:	yes		
launder_folio:	yes		
is_partially_uptodate:	yes		
error_remove_page:	yes		
swap_activate:	no		
swap_deactivate:	no		

->write_begin(), ->write_end() and ->readpage() may be called from the request handler (/dev/loop).

->readpage() unlocks the page, either synchronously or via I/O completion.

->readahead() unlocks the pages that I/O is attempted on like ->readpage().

->writepage() is used for two purposes: for "memory cleansing" and for "sync". These are quite different operations and the behaviour may differ depending upon the mode.

If writepage is called for sync (wbc->sync_mode != WBC_SYNC_NONE) then it *must* start I/O against the page, even if that would involve blocking on in-progress I/O.

If writepage is called for memory cleansing (sync_mode == WBC_SYNC_NONE) then its role is to get as much writeout underway as possible. So writepage should try to avoid blocking against currently-in-progress I/O.

If the filesystem is not called for "sync" and it determines that it would need to block against in-progress I/O to be able to start new I/O against the page the filesystem should redirty the page with redirty_page_for_writepage(), then unlock the page and return zero. This may also be done to avoid internal deadlocks, but rarely.

If the filesystem is called for sync then it must wait on any in-progress I/O and then start new I/O.

The filesystem should unlock the page synchronously, before returning to the caller, unless ->writepage() returns special WRITEPAGE_ACTIVATE value. WRITEPAGE_ACTIVATE means that page cannot really be written out currently, and VM should stop calling ->writepage() on this page for some time. VM does this by moving page to the head of the active list, hence the name.

Unless the filesystem is going to redirty_page_for_writepage(), unlock the page and return zero, writepage *must* run set_page_writeback() against the page, followed by unlocking it. Once set_page_writeback() has been run against the page, write I/O can be submitted and the write I/O completion handler must run end_page_writeback() once the I/O is complete. If no I/O is submitted, the filesystem must run end_page_writeback() against the page before returning from writepage.

That is: after 2.5.12, pages which are under writeout are *not* locked. Note, if the filesystem needs the page to be locked during writeout, that is ok, too, the page is allowed to be unlocked at any point in time between the calls to set_page_writeback() and end_page_writeback().

Note, failure to run either redirty_page_for_writepage() or the combination of set_page_writeback()/end_page_writeback() on a

page submitted to writepage will leave the page itself marked clean but it will be tagged as dirty in the radix tree. This incoherency can lead to all sorts of hard-to-debug problems in the filesystem like having dirty inodes at umount and losing written data.

->writepages() is used for periodic writeback and for syscall-initiated sync operations. The address_space should start I/O against at least `*nr_to_write` pages. `*nr_to_write` must be decremented for each page which is written. The address_space implementation may write more (or less) pages than `*nr_to_write` asks for, but it should try to be reasonably close. If nr_to_write is NULL, all dirty pages must be written.

writepages should _only_ write pages which are present on mapping->io_pages.

->dirty_folio() is called from various places in the kernel when the target folio is marked as needing writeback. The folio cannot be truncated because either the caller holds the folio lock, or the caller has found the folio while holding the page table lock which will block truncation.

->bmap() is currently used by legacy ioctl() (FIBMAP) provided by some filesystems and by the swapper. The latter will eventually go away. Please, keep it that way and don't breed new callers.

->invalidate_folio() is called when the filesystem must attempt to drop some or all of the buffers from the page when it is being truncated. It returns zero on success. The filesystem must exclusively acquire invalidate_lock before invalidating page cache in truncate / hole punch path (and thus calling into ->invalidate_folio) to block races between page cache invalidation and page cache filling functions (fault, read, ...).

->releasepage() is called when the kernel is about to try to drop the buffers from the page in preparation for freeing it. It returns zero to indicate that the buffers are (or may be) freeable. If ->releasepage is zero, the kernel assumes that the fs has no private interest in the buffers.

->freepage() is called when the kernel is done dropping the page from the page cache.

->launder_folio() may be called prior to releasing a folio if it is still found to be dirty. It returns zero if the folio was successfully cleaned, or an error value if not. Note that in order to prevent the folio getting mapped back in and redirtied, it needs to be kept locked across the entire operation.

->swap_activate will be called with a non-zero argument on files backing (non block device backed) swapfiles. A return value of zero indicates success, in which case this file can be used for backing swapspace. The swapspace operations will be proxied to the address space operations.

->swap_deactivate() will be called in the sys_swapoff() path after ->swap_activate() returned success.

file_lock_operations

prototypes:

```
void (*fl_copy_lock)(struct file_lock *, struct file_lock *);
void (*fl_release_private)(struct file_lock *);
```

locking rules:

ops	inode->i_lock	may block
fl_copy_lock:	yes	no
fl_release_private:	maybe	maybe[1]_

lock_manager_operations

prototypes:

```
void (*lm_notify)(struct file_lock *);  /* unblock callback */
int (*lm_grant)(struct file_lock *, struct file_lock *, int);
void (*lm_break)(struct file_lock *); /* break_lease callback */
int (*lm_change)(struct file_lock **, int);
bool (*lm_breaker_owns_lease)(struct file_lock *);
```

locking rules:

ops	flc_lock	blocked_lock_lock	may block
lm_notify:	no	yes	no
lm_grant:	no	no	no
lm_break:	yes	no	no
lm_change	yes	no	no
lm_breaker_owns_lease:	yes	no	no

buffer_head

prototypes:

```
void (*b_end_io)(struct buffer_head *bh, int uptodate);
```

locking rules:

called from interrupts. In other words, extreme care is needed here. bh is locked, but that's all warranties we have here. Currently only RAID1, highmem, fs/buffer.c, and fs/ntfs/aops.c are providing these. Block devices call this method upon the IO completion.

block_device_operations

prototypes:

```
int (*open) (struct block_device *, fmode_t);
int (*release) (struct gendisk *, fmode_t);
int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
int (*direct_access) (struct block_device *, sector_t, void **,
                      unsigned long *);
void (*unlock_native_capacity) (struct gendisk *);
int (*getgeo)(struct block_device *, struct hd_geometry *);
void (*swap_slot_free_notify) (struct block_device *, unsigned long);
```

locking rules:

ops	open_mutex
open:	yes
release:	yes
ioctl:	no
compat_ioctl:	no
direct_access:	no
unlock_native_capacity:	no
getgeo:	no
swap_slot_free_notify:	no (see below)

swap_slot_free_notify is called with swap_lock and sometimes the page lock held.

file_operations

prototypes:

```
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
int (*iopoll) (struct kiocb *kiocb, bool spin);
int (*iterate) (struct file *, struct dir_context *);
int (*iterate_shared) (struct file *, struct dir_context *);
__poll_t (*poll) (struct file *, struct poll_table_struct *);
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, loff_t start, loff_t end, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                loff_t *, int);
unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                unsigned long, unsigned long, unsigned long);
int (*check_flags)(int);
int (*flock) (struct file *, int, struct file_lock *);
ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
                size_t, unsigned int);
ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
                size_t, unsigned int);
int (*setlease)(struct file *, long, struct file_lock **, void **);
long (*fallocate)(struct file *, int, loff_t, loff_t);
void (*show_fdinfo)(struct seq_file *m, struct file *f);
unsigned (*mmap_capabilities)(struct file *);
ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
                loff_t, size_t, unsigned int);
loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
                struct file *file_out, loff_t pos_out,
                loff_t len, unsigned int remap_flags);
int (*fadvise)(struct file *, loff_t, loff_t, int);
```

locking rules:

All may block.

->llseek() locking has moved from llseek to the individual llseek implementations. If your fs is not using generic_file_llseek, you need to acquire and release the appropriate locks in your ->llseek(). For many filesystems, it is probably safe to acquire the inode mutex or just to use i_size_read() instead. Note: this does not protect the file->f_pos against concurrent modifications since this is something the userspace has to take care about.

->iterate() is called with i_rwsem exclusive.

->iterate_shared() is called with i_rwsem at least shared.

->fasync() is responsible for maintaining the FASYNC bit in filp->f_flags. Most instances call fasync_helper(), which does that maintenance, so it's not normally something one needs to worry about. Return values > 0 will be mapped to zero in the VFS layer.

->readdir() and ->ioctl() on directories must be changed. Ideally we would move ->readdir() to inode_operations and use a separate method for directory ->ioctl() or kill the latter completely. One of the problems is that for anything that resembles union-mount we won't have a struct file for all components. And there are other reasons why the current interface is a mess...

->read on directories probably must go away - we should just enforce -EISDIR in sys_read() and friends.

->setlease operations should call generic_setlease() before or after setting the lease within the individual filesystem to record the result of the operation

->fallocate implementation must be really careful to maintain page cache consistency when punching holes or performing other operations that invalidate page cache contents. Usually the filesystem needs to call truncate_inode_pages_range() to invalidate relevant range of the page cache. However the filesystem usually also needs to update its internal (and on disk) view of file offset -> disk block mapping. Until this update is finished, the filesystem needs to block page faults and reads from reloading now-stale page cache contents from the disk. Since VFS acquires mapping->invalidate_lock in shared mode when loading pages from disk (filemap_fault(), filemap_read(), readahead paths), the fallocate implementation must take the invalidate_lock to prevent reloading.

->copy_file_range and ->remap_file_range implementations need to serialize against modifications of file data while the operation is running. For blocking changes through write(2) and similar operations inode->i_rwsem can be used. To block changes to file contents via a memory mapping during the operation, the filesystem must take mapping->invalidate_lock to coordinate with ->page_mkwrite.

dquot_operations

prototypes:

```
int (*write_dquot) (struct dquot *);
int (*acquire_dquot) (struct dquot *);
int (*release_dquot) (struct dquot *);
int (*mark_dirty) (struct dquot *);
int (*write_info) (struct super_block *, int);
```

These operations are intended to be more or less wrapping functions that ensure a proper locking wrt the filesystem and call the generic quota operations.

What filesystem should expect from the generic quota functions:

ops	FS recursion	Held locks when called
write_dquot:	yes	dqonoff_sem or dqptr_sem
acquire_dquot:	yes	dqonoff_sem or dqptr_sem
release_dquot:	yes	dqonoff_sem or dqptr_sem
mark_dirty:	no	•
write_info:	yes	dqonoff_sem

FS recursion means calling ->quota_read() and ->quota_write() from superblock operations.

More details about quota locking can be found in fs/dquot.c.

vm_operations_struct

prototypes:

```
void (*open)(struct vm_area_struct*);
void (*close)(struct vm_area_struct*);
vm_fault_t (*fault)(struct vm_area_struct*, struct vm_fault *);
vm_fault_t (*page_mkwrite)(struct vm_area_struct *, struct vm_fault *);
vm_fault_t (*pfn_mkwrite)(struct vm_area_struct *, struct vm_fault *);
int (*access)(struct vm_area_struct *, unsigned long, void*, int, int);
```

locking rules:

ops	mmap_lock	PageLocked(page)

ops	mmap_lock	PageLocked(page)
open:	yes	
close:	yes	
fault:	yes	can return with page locked
map_pages:	yes	
page_mkwrite:	yes	can return with page locked
pfn_mkwrite:	yes	
access:	yes	

->fault() is called when a previously not present pte is about to be faulted in. The filesystem must find and return the page associated with the passed in "pgoff" in the vm_fault structure. If it is possible that the page may be truncated and/or invalidated, then the filesystem must lock invalidate_lock, then ensure the page is not already truncated (invalidate_lock will block subsequent truncate), and then return with VM_FAULT_LOCKED, and the page locked. The VM will unlock the page.

->map_pages() is called when VM asks to map easy accessible pages. Filesystem should find and map pages associated with offsets from "start_pgoff" till "end_pgoff". ->map_pages() is called with page table locked and must not block. If it's not possible to reach a page without blocking, filesystem should skip it. Filesystem should use do_set_pte() to setup page table entry. Pointer to entry associated with the page is passed in "pte" field in vm_fault structure. Pointers to entries for other offsets should be calculated relative to "pte".

->page_mkwrite() is called when a previously read-only pte is about to become writeable. The filesystem again must ensure that there are no truncate/invalidate races or races with operations such as ->remap_file_range or ->copy_file_range, and then return with the page locked. Usually mapping->invalidate_lock is suitable for proper serialization. If the page has been truncated, the filesystem should not look up a new page like the ->fault() handler, but simply return with VM_FAULT_NOPAGE, which will cause the VM to retry the fault.

->pfn_mkwrite() is the same as page_mkwrite but when the pte is VM_PFNMAP or VM_MIXEDMAP with a page-less entry. Expected return is VM_FAULT_NOPAGE. Or one of the VM_FAULT_ERROR types. The default behavior after this call is to make the pte read-write, unless pfn_mkwrite returns an error.

->access() is called when get_user_pages() fails in access_process_vm(), typically used to debug a process through /proc/pid/mem or ptrace. This function is needed only for VM_IO | VM_PFNMAP VMAs.

Dubious stuff

(if you break something or notice that it is broken and do not fix it yourself - at least put it here)