

VFS internals

Sommaire

1. General description of responsibilities
2. General architecture
3. Worker threads
4. Locking
5. Recovery from driver crashes

1. General description of responsibilities

VFS implements the file system in cooperation with one or more File Servers (FS). The File Servers take care of the actual file system on a partition. That is, they interpret the data structure on disk, write and read data to/from disk, etc. VFS sits on top of those File Servers and communicates with them. Looking inside VFS, we can identify several roles. First, a role of VFS is to handle most POSIX system calls that are supported by Minix. Additionally, it supports a few calls necessary for libc. The following system calls are handled by VFS:

access, chdir, chmod, chown, chroot, close, creat, fchdir, fcntl, fstat, fstatfs, fstatvfs, fsync, ftruncate, getdents, ioctl, link, llseek, lseek, lstat, mkdir, mknod, mount, open, pipe, read, readlink, rename, rmdir, select, stat, statvfs, symlink, sync, truncate, umask, umount, unlink, utime, write.

Second, it maintains part of the state belonging to a process (process state is spread out over the kernel, VM, PM, and VFS). For example, it maintains state for select(2) calls, file descriptors and file positions. Also, it cooperates with the Process Manager to handle the fork, exec, and exit system calls. Third, VFS keeps track of endpoints that are supposed to be drivers for character or block special files. File Servers can be regarded as drivers for block special files, although they are handled entirely different compared to other drivers.

The following diagram depicts how a read() on a file in /home is being handled:

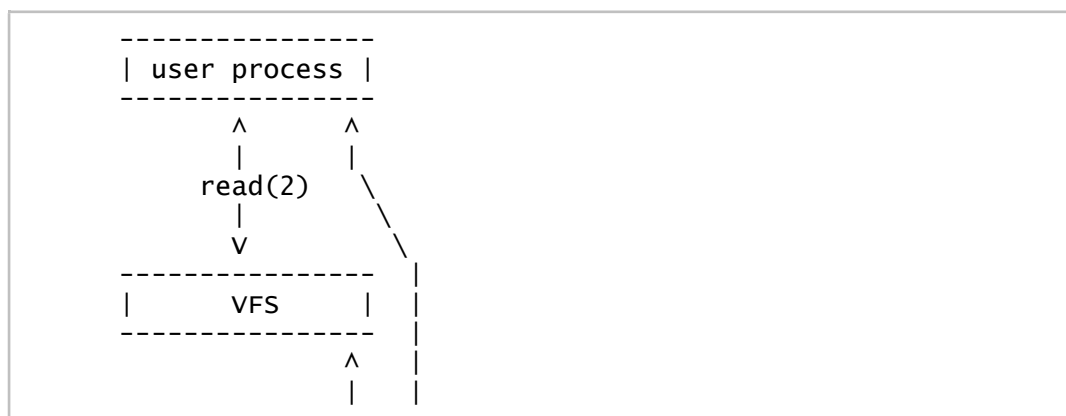




Diagram 1: handling of read(2) system call

The user process executes the read system call which is delivered to VFS. VFS verifies the read is done on a valid (open) file and forwards the request to the FS responsible for the file system on which the file resides. The FS reads the data, copies it directly to the user process, and replies to VFS it has executed the request. Subsequently, VFS replies to the user process the operation is done and the user process continues to run.

2. General architecture

VFS works roughly identical to every other server and driver in Minix; it fetches a message (internally referred to as a job in some cases), executes the request embedded in the message, returns a reply, and fetches the next job. There are several sources for new jobs: from user processes, from PM, from the kernel, and from suspended jobs inside VFS itself (suspended operations on pipes, locks, or character special files). File Servers are regarded as normal user processes in this case, but their abilities are limited. This is to prevent deadlocks. Once a job is received, a worker thread starts executing it. During the lifetime of a job, the worker thread might need to talk to several File Servers. The protocol VFS speaks with File Servers is fully documented on the Wiki at [0]. The protocol fields are defined in `<minix/vfsif.h>`. If the job is an operation on a character or block special file and the need to talk to a driver arises, VFS uses the Character and Block Device Protocol. See [1]. This is sadly not official documentation, but it is an accurate description of how it works. Luckily, driver writers can use the `libchardriver` and `libblockdriver` libraries and don't have to know the details of the protocol.

3. Worker threads

Upon start up, VFS spawns a configurable amount of worker threads. The main thread fetches requests and replies, and hands them off to idle or reply-pending workers, respectively. If no worker threads are available, the request is queued. There are 3 types of worker threads: normal, a system worker, and a deadlock resolver. All standard system calls are handled by normal worker threads. Jobs from PM and notifications from the kernel are taken care of by the system worker. The deadlock resolver handles jobs from system processes (i.e., File Servers and drivers) when there are no normal worker threads available; all normal threads might be blocked on a single worker thread that caused a system process to send a request on its own. To unblock all normal threads, we need to reserve one thread to handle that situation. VFS drives all File Servers and drivers asynchronously. While waiting for a reply, a worker thread is blocked and other workers can keep processing requests. Upon reply the worker thread is unblocked. As mentioned above, the main thread is responsible for retrieving new jobs and replies to current jobs and start or unblock the proper worker thread. Given how many sources for new jobs and replies there are, the work for the main thread is quite complicated. Consider Table 1.

From	normal	deadlock	system
msg is new job			
PM			X
Notification from the kernel			X
Notification from DS or system process	X	X	
User process	X		
Unuspended process	X		
msg is reply			
File Server reply	resume		
Sync. driver reply	resume		
Async. driver reply	resume/X	X	

Table 1: VFS' message fetching main loop. X means 'start thread'.

The reason why asynchronous driver replies get their own thread is for the following. In some cases, a reply has a thread blocked waiting for it which can be resumed (e.g., open). In another case there's a lot of work to be done which involves sending new messages (e.g., select replies). Finally, DEV_REVIVE replies unblock suspended processes which in turn generate new jobs to be handled by the main loop (e.g., suspended reads and writes). So depending on the reply a new thread has to be started. Having all this logic in the main loop is messy, so we start a thread regardless of the actual reply contents. When there are no worker threads available and there is no need to invoke the deadlock resolver (i.e., normal system calls), the request is queued in the fproc table. This works because a process can send only one system call at a time. When implementing kernel threads, one has to take this assumption into account. The protocol PM speaks with VFS is asynchronous and PM is allowed to send as many request to VFS as it wants. It is impossible to use the same queueing mechanism as normal processes use, because that would allow for just 1 queued message. Instead, the system worker maintains a linked list of pending requests. Moreover, this queueing mechanism is also the reason why notifications from the kernel are handled by the system worker; the kernel has no corresponding fproc table entry (so we can't store it there) and the linked list has no dependencies on that table. Communication with drivers is asynchronous even when the driver uses the synchronous driver protocol. However, to guarantee identical behavior, access to synchronous drivers is serialized. File Servers are treated differently. VFS was designed to be able to send requests concurrently to File Servers, although at the time of writing there are no File Servers that can actually make use of that functionality. To identify which reply from an FS belongs to which worker thread, all requests have an embedded transaction identification number (a magic number + thread id encoded in the mtype field of a message) which the FS has to echo upon reply. Because the range of valid transaction IDs is isolated from valid system call numbers, VFS can use that ID to differentiate between replies from File Servers

and actual new system calls from FSes. Using this mechanism VFS is able to support FUSE and ProcFS.

4. Locking

To ensure correct execution of system calls, worker threads sometimes need certain objects within VFS to remain unchanged during thread suspension and resumption (i.e., when they need to communicate with a driver or File Server). Threads keep most state on the stack, but there are a few global variables that require protection: the fproc table, vmnt table, vnode table, and filp table. Other tables such as lock table, select table, and dmap table don't require protection by means of exclusive access. There it's required and enough to simply mark an entry in use.

4.1. Locking requirements

VFS implements the locking model described in [2]. For completeness of this document we'll describe it here, too. The requirements are based on a threading package that is non-preemptive. VFS must guarantee correct functioning with several, semi-concurrently executing threads in any arbitrary order. The latter requirement follows from the fact that threads need service from other components like File Servers and drivers, and they may take any time to complete requests.

1. Consistency of replicated values. Several system calls rely on VFS keeping a replicated representation of data in File Servers (e.g., file sizes, file modes, etc.).
2. Isolation of system calls. Many system calls involve multiple requests to FSes. Concurrent requests from other processes must not lead to otherwise impossible results (e.g., a chmod operation on a file cannot fail halfway through because it's suddenly unlinked or moved).
3. Integrity of objects. From the point of view of threads, obtaining mutual exclusion is a potentially blocking operation. The integrity of any objects used across blocking calls must be guaranteed (e.g., the file mode in a vnode must remain intact not only when talking to other components, but also when obtaining a lock on a filp).
4. No deadlock. Not one call may cause another call to never complete. Deadlock situations are typically the result of two or more threads that each hold exclusive access to one resource and want exclusive access to the resource held by the other thread. These resources are a) data (global variables) and b) worker threads.
 - a. Conflicts between locking of different types of objects can be avoided by keeping a locking order: objects of different type must always be locked in the same order. If multiple objects of the same type are to be locked, then first a "common denominator" higher up in the locking order must be locked.
 - b. Some threads can only run to completion when another thread does work on their behalf. Examples of this are drivers and file servers that do system calls on their own (e.g., ProcFS, PFS/UNIX Domain Sockets, FUSE) or crashing components (e.g., a driver for a character special file that crashes during a request; a second thread is required to handle resource clean up or driver restart before the first thread can abort or retry the

request).

5. No starvation. VFS must guarantee that every system call completes in finite time (e.g., an infinite stream of reads must never completely block writes). Furthermore, we want to maximize parallelism to improve performance. This leads to:
6. A request to one File Server must not block access to other FS processes. This means that most forms of locking cannot take place at a global level, and must at most take place on the file system level.
7. No read-only operation on a regular file must block an independent read call to that file. In particular, (read-only) open and close operations may not block such reads, and multiple independent reads on the same file must be able to take place concurrently (i.e., reads that do not share a file position between their file descriptors).

4.2. Three-level Lock

From the requirements it follows that we need at least two locking types: read and write locks. Concurrent reads are allowed, but writes are exclusive both from reads and from each other. However, in a lot of cases it possible to use a third locking type that is in between read and write lock: the serialize lock. This is implemented in the three-level lock [2]. The three-level lock provides: TLL_READ: allows an unlimited number of threads to hold the lock with the same type (both the thread itself and other threads); N * concurrent. TLL_READSER: also allows an unlimited number of threads with type TLL_READ, but only one thread can obtain serial access to the lock; N * concurrent + 1 * serial. TLL_WRITE: provides full mutual exclusion; 1 * exclusive + 0 * concurrent + 0 * serial. In absence of TLL_READ locks, a TLL_READSER is identical to TLL_WRITE. However, TLL_READSER never blocks concurrent TLL_READ access. TLL_READSER can be upgraded to TLL_WRITE; the thread will block until the last TLL_READ lock leaves and new TLL_READ locks are blocked. Locks can be downgraded to a lower type. The three-level lock is implemented using two FIFO queues with write-bias. This guarantees no starvation.

4.3. Data structures subject to locking

VFS has a number of global data structures. See Table 2.

Structure	Object
description	
+-----+	
fproc	Process (includes process's file descriptors)
+-----+	
vmnt	virtual mount; a mounted file system
+-----+	
vnode	virtual node; an open file
+-----+	

filp file	File position into an open file
-----+-----	

lock file	File region locking state for an open file
-----+-----	

select call	State for an in-progress select(2) call
-----+-----	

dmap driver	Mapping from major device number to a device driver
-----+-----	

Table 2: VFS object types.

An fproc object is a process. An fproc object is created by fork(2) and destroyed by exit(2) (which may, or may not, be instantiated from the process itself). It is identified by its endpoint number ('fp_endpoint') and process id ('fp_pid'). Both are unique although in general the endpoint number is used throughout the system. A vmnt object is a mounted file system. It is created by mount(2) and destroyed by umount(2). It is identified by a device number ('m_dev') and FS endpoint number ('m_fs_e'); both are unique to each vmnt object. There is always a single process that handles a file system on a device and a device cannot be mounted twice. A vnode object is the VFS representation of an open inode on the file system. A vnode object is created when a first process opens or creates the corresponding file and is destroyed when the last process, which has that file open, closes it. It is identified by a combination of FS endpoint number ('v_fs_e') and inode number of that file system ('v_inode_nr'). A vnode might be mapped to another file system; the actual reading and writing is handled by a different endpoint. This has no effect on locking. A filp object contains a file position within a file. It is created when a file is opened or anonymous pipe created and destroyed when the last user (i.e., process) closes it. A file descriptor always points to a single filp. A filp always point to a single vnode, although not all vnodes are pointed to by a filp. A filp has a reference count ('filp_count') which is identical to the number of file descriptors pointing to it. It can be increased by a dup(2) or fork(2). A filp can therefore be shared by multiple processes. A lock object keeps information about locking of file regions. This has nothing to do with the threading type of locking. The lock objects require no locking protection and won't be discussed further. A select object keeps information on a select(2) operation that cannot be fulfilled immediately (waiting for timeout or file descriptors not ready). They are identified by their owner ('requestor'); a pointer to the fproc table. A null pointer means not in use. A select object can be used by only one process and a process can do only one select(2) at a time. Select(2) operates on filps and is organized in such a way that it is sufficient to apply locking on individual filps and not on select objects themselves. They won't be discussed further. A dmap object is a mapping from a device number to a device driver. A device driver can have multiple device numbers associated (e.g., TTY). Access to a driver is exclusive when it uses the synchronous driver protocol.

4.4. Locking order

Based on the description in the previous section, we need protection for fproc,

vmnt, vnode, and filp objects. To prevent deadlocks as a result of object locking, we need to define a strict locking order. In VFS we use the following order:

fproc -> [exec] -> vmnt -> vnode -> filp -> [block special file] -> [dmap]

That is, no thread may lock an fproc object while holding a vmnt lock, and no thread may lock a vmnt object while holding an (associated) vnode, etc. Fproc needs protection because processes themselves can initiate system calls, but also PM can cause system calls that have to be executed in their name. For example, a process might be busy reading from a character device and another process sends a termination signal. The `exit(2)` that follows is sent by PM and is to be executed by the to-be-killed process itself. At this point there is contention for the fproc object that belongs to the process, hence the need for protection. The `exec(2)` call is protected by a mutex for the following reason. VFS uses a number of variables on the heap to read ELF headers. They are on the heap due to their size; putting them on the stack would increase stack size demands for worker threads. The `exec` call does blocking read calls and thus needs exclusive access to these variables. However, only the `exec(2)` syscall needs this lock. Access to block special files needs to be exclusive. File Servers are responsible for handling reads from and writes to block special files; if a block special file is on a device that is mounted, the FS responsible for that mount point takes care of it, otherwise the FS that handles the root of the file system is responsible. Due to mounting and unmounting file systems, the FS handling a block special file may change. Locking the vnode is not enough since the inode can be on an entirely different File Server. Therefore, access to block special files must be mutually exclusive from concurrent `mount(2)/umount(2)` operations. However, when we're not accessing a block special file, we don't need this lock.

4.5. Vmnt (file system) locking

Vmnt locking cannot be seen completely separately from vnode locking. For example, `umount(2)` fails if there are still in-use vnodes, which means that FS requests [0] only involving in-use inodes do not have to acquire a vmnt lock. On the other hand, all other request do need a vmnt lock. Extrapolating this to system calls this means that all system calls involving a file descriptor don't need a vmnt lock and all other system calls (that make FS requests) do need a vmnt lock.

----- -----	
Category calls	System
+-----+-----	
-----+	
System calls with creat, dumpcore+, a path name open, readlink, argument truncate, umount, utime	access, chdir, chmod, chown, chroot, exec, link, lstat, mkdir, mknod, mount, rename, rmdir, stat, statvfs, symlink, unlink,
+-----+-----	
-----+	
System calls with ftruncate, a file descriptor select, write	close, fchdir, fcntl, fstat, fstatvfs, getdents, ioctl, llseek, pipe, read,

argument	
+-----+-----	
-----+	
System calls with fsync++, sync,	
umask	
other or no	
arguments	

Table 3: System call categories. + path name argument is implicit, the path name is "core.<pid>" ++ although fsync actually provides a file descriptor argument, it's only used to find the vmnt and not to do any actual operations on

Before we describe what kind of vmnt locks VFS applies to system calls with a path name or other arguments, we need to make some notes on path lookup. Path lookups take arbitrary paths as input (relative and absolute). They can start at any vmnt (based on root directory and working directory of the process doing the lookup) and visit any file system in arbitrary order, possibly visiting the same file system more than once. As such, VFS can never tell in advance at which File Server a lookup will end. This has the following consequences:

- In the lookup procedure, only one vmnt must be locked at a time. When moving from one vmnt to another, the first vmnt has to be unlocked before acquiring the next lock to prevent deadlocks.
- The lookup procedure must lock each visited file system with TLL_READSER and downgrade or upgrade to the lock type desired by the caller for the destination file system (as VFS cannot know which file system is final). This is to prevent deadlocks when a thread acquires a TLL_READSER on a vmnt and another thread TLL_READ on the same vmnt. If the second thread is blocked on the first thread due to it acquiring a lock on a vnode, the first thread will be unable to upgrade a TLL_READSER lock to TLL_WRITE.

We use the following mapping for vmnt locks onto three-level lock types:

Lock type	Mapped to	Used
for		
+-----+-----		
-----+		
VMNT_READ	TLL_READ	Read-only operations and fully
independent write		
operations		
+-----+-----		
-----+		
VMNT_WRITE	TLL_READSER	Independent create and modify
operations		
+-----+-----		
-----+		
VMNT_EXCL	TLL_WRITE	Delete and dependent write
operations		

Table 4: vmnt to tll lock mapping

The following table shows a sub-categorization of system calls without a file descriptor argument, together with their locking types and motivation as used by VFS.

Group Motivation	System calls	Lock type	
File open ops. vnodes can be (non-create) concurrently, and open affect state.	chdir, chroot, exec, open and open	VMNT_READ	These operations do not interfere with each other, as opened operations do not replicated
File create-require mutual and-open concurrent file ops file already VMNT_WRITE lock the lookup is upgraded	creat, open(O_CREAT) that	VMNT_EXCL for create VMNT_WRITE for open	File create ops. exclusion from open ops. If the existed, the is necessary for not
File create-nameless inodes unique-and-opened by means creation interfere with else	pipe	VMNT_READ	These create which cannot be of a path. Their therefore does not anything
File create-not affect only ops. can therefore concurrently with operations	mkdir, mknod, slink with open	VMNT_WRITE	These operations do any VFS state, and take place

File info not interfere retrieval or do not modify modification state	access, lstat readlink,stat utime	VMNT_READ	These operations do with each other and replicated
-----+-----+-----+-----			
File not interfere modification They do need the vnode level	chmod, chown, truncate	VMNT_READ	These operations do with each other. exclusive access on
-----+-----+-----+-----			
File link create-only ops. operations	link	VMNT_WRITE	Identical to file
-----+-----+-----+-----			
File unlink interfere with ops. operations, to avoid inodes are However, due checks, the locked VMNT_WRITE upgraded	rmdir, unlink avoid	VMNT_EXCL	These must not file create the scenario where reused immediately. to necessary path vmnt is first and then
-----+-----+-----+-----			
File rename unlink ops. operations	rename	VMNT_EXCL	Identical to file
-----+-----+-----+-----			
Non-file involve the file ops. not need not alter state atomic at the FS level	sync, umask	VMNT_READ or none	umask does not system, so it does locks. sync does in VFS and is
-----+-----+-----+-----			

Table 5: System call without file descriptor argument sub-categorization

4.6. Vnode (open file) locking

Compared to vmnt locking, vnode locking is relatively straightforward. All read-only accesses to vnodes that merely read the vnode object's fields are allowed to be concurrent. Consequently, all accesses that change fields of a vnode object must be exclusive. This leaves us with creation and destruction of vnode objects (and related to that, their reference counts); it's sufficient to serialize these accesses. This follows from the fact that a vnode is only created when the first user opens it, and destroyed when the last user closes it. A open file in process A cannot be closed by process B. Note that this also relies on the fact that a process can do only one system call at a time. Kernel threads would violate this assumption.

We use the following mapping for vnode locks onto three-level lock types:

Lock type for	Mapped to	Used
VNODE_READ vnodes	TLL_READ	Read access to previously opened vnodes
VNODE_OPCL destruction of vnodes	TLL_READSER	Creation, opening, closing, and destruction of vnodes
VNODE_WRITE vnodes	TLL_WRITE	Write access to previously opened vnodes

Table 6: vnode to tll lock mapping

When vnodes are destroyed, they are initially locked with VNODE_OPCL. After all, we're going to alter the reference count, so this must be serialized. If the reference count then reaches zero we obtain exclusive access. This should always be immediately possible unless there is a consistency problem. See section 4.8 for an exhaustive listing of locking methods for all operations on vnodes.

4.7. Filp (file position) locking

The main fields of a filp object that are shared between various processes (and by extension threads), and that can change after object creation, are filp_count and filp_pos. Writes to and reads from filp object must be mutually exclusive, as all system calls have to use the latest version. For example, a read(2) call changes the file position (i.e., filp_pos), so two concurrent reads must obtain exclusive access. Consequently, as even read operations require exclusive access, filp object don't use three-level locks, but only mutexes.

System calls that involve a file descriptor often access both the filp and the corresponding vnode. The locking order requires us to first lock the vnode and then the filp. This is taken care of at the filp level. Whenever a filp is locked, a lock on the vnode is acquired first. Conversely, when a filp is unlocked, the corresponding vnode is also unlocked. A convenient consequence is that whenever a vnode is locked exclusively (VNODE_WRITE), all corresponding filps

are implicitly locked. This is of particular use when multiple filps must be locked at the same time:

- When opening a named pipe, VFS must make sure that there is at most one filp for the reader end and one filp for the writer end.
- Pipe readers and writers must be suspended in the absence of (respectively) writers and readers.

Because both filps are linked to the same vnode object (they are for the same pipe), it suffices to exclusively lock that vnode instead of both filp objects.

In some cases it can happen that a function that operates on a locked filp, calls another function that triggers another lock on a different filp for the same vnode. For example, `close_filp`. At some point, `close_filp()` calls `release()` which in turn will loop through the filp table looking for pipes being select(2)ed on. If there are, the select code will lock the filp and do operations on it. This works fine when doing a select(2) call, but conflicts with `close(2)` or `exit(2)`. `Lock_filp()` makes an exception for this situation; if you've already locked a vnode with `VNODE_OPCL` or `VNODE_WRITE` when locking a filp, you obtain a "soft lock" on the vnode for this filp. This means that `lock_filp` won't actually try to lock the vnode (which wouldn't work), but flags the vnode as "skip unlock_vnode upon unlock_filp." Upon unlocking the filp, the vnode remains locked, the soft lock is removed, and the filp mutex is released. Note that this scheme does not violate the locking order; the vnode is (already) locked before the filp.

A similar problem arises with `create_pipe`. In this case we obtain a new vnode object, lock it, and obtain two new, locked, filp objects. If everything works out and the filp objects are linked to the same vnode, we run into trouble when unlocking both filps. The first filp being unlocked would work; the second filp doesn't have an associated vnode that's locked anymore. Therefore we introduced a plural `unlock_filps(filp1, filp2)` that can unlock two filps that both point to the same vnode.

4.8. Lock characteristics per request type

For File Servers that support concurrent requests, it's useful to know which locking guarantees VFS provides for vmnts and vnodes, so it can take that into account when protecting internal data structures. `READ = TLL_READ`, `READSER = TLL_READSER`, `WRITE = TLL_WRITE`. The vnode locks applies to the `REQ_INODE_NR` field in requests, unless the notes say otherwise.

request notes	vmnt	vnode	
REQ_BREAD and writes to files		READ	VFS serializes reads from block special
REQ_BWRITE and writes to files		WRITE	VFS serializes reads from block special

REQ_CHMOD file is not opened	READ 	WRITE 	vmnt is only locked if already
REQ_CHOWN file is not opened	READ 	WRITE 	vmnt is only locked if already
REQ_CREATE file is locked	WRITE 	WRITE 	The directory in which the created is write
REQ_FLUSH REQ_BREAD and REQ_BWRITE			Mutually exclusive to
REQ_FSTATFS 			
REQ_FTRUNC file is not opened	READ 	WRITE 	vmnt is only locked if already
REQ_GETDENTS file is not opened	READ 	READ 	vmnt is only locked if already
REQ_INHIBREAD 		READ 	
REQ_LINK READ WRITE	READSER 	WRITE 	REQ_INODE_NR is locked REQ_DIR_INO is locked
REQ_LOOKUP 	READSER 		
REQ_MKDIR 	READSER 	WRITE 	
REQ_MKNOD 	READSER 	WRITE 	

REQ_MOUNTPOINT	WRITE	WRITE	
+-----+		+-----+	
+-----+			
REQ_NEW_DRIVER			
+-----+		+-----+	
+-----+			
REQ_NEWNODE			Only sent to
PFS			
+-----+		+-----+	
+-----+			
REQ_PUTNODE		READSER	READSER when dropping all
but one			
		or WRITE	references. WRITE when
final reference			
			is dropped (i.e., no
longer in use)			
+-----+		+-----+	
+-----+			
REQ_RDLINK	READ	READ	In some circumstances
stricter locking			
			might be applied, but not
guaranteed			
+-----+		+-----+	
+-----+			
REQ_READ		READ	
+-----+		+-----+	
+-----+			
REQ_READSUPER	WRITE		
+-----+		+-----+	
+-----+			
REQ_RENAME	WRITE	WRITE	
+-----+		+-----+	
+-----+			
REQ_RMDIR	WRITE	WRITE	
+-----+		+-----+	
+-----+			
REQ_SLINK	READSER	READ	
+-----+		+-----+	
+-----+			
REQ_STAT	READ	READ	vmnt is only locked if
file is not			
			already
opened			
+-----+		+-----+	
+-----+			
REQ_STATVFS	READ	READ	vmnt is only locked if
file is not			
			already
opened			
+-----+		+-----+	
+-----+			
REQ_SYNC	READ		
+-----+		+-----+	
+-----+			
REQ_UNLINK	WRITE	WRITE	

REQ_UNMOUNT	WRITE		
REQ_UTIME	READ	READ	
REQ_WRITE		WRITE	

Table 7: VFS-FS requests locking guarantees

5. Recovery from driver crashes

VFS can recover from block special file and character special file driver crashes. It can recover to some degree from a crashed File Server (which we can regard as a driver).

5.1. Recovery from block drivers crashes




When reading or writing, VFS doesn't communicate with block drivers directly, but always through a File Server (the root File Server being default). If the block driver crashes, the File Server does most of the work of the recovery procedure. VFS loops through all open files for block special files that were handled by this driver and reopens them. After that it sends the new endpoint to the File Server so it can finish the recover procedure. Finally, the File Server will retry pending requests if possible. However, reopening files can cause the block driver to crash again. When that happens, VFS will stop the recovery. A driver can return ERESTART to VFS to tell it to retry a request. VFS does this with an arbitrary maximum of 5 attempts.

5.2. Recovery from character driver crashes

Character special files are treated differently. Once VFS has found out a driver has been restarted, it will stop the current request (if there is any). It makes no sense to retry requests due to the nature of character special files. If a character special driver can restart without changing endpoints, this merely results in the current request (e.g., read, write, or ioctl) failing and allows the user process to reissue the same request. On the other hand, if a driver restart causes the driver to change endpoint number, all associated file descriptors are marked invalid and subsequent operations on them will always fail with a bad file descriptor error.

5.3. Recovery from File Server crashes

At the time of writing we cannot recover from crashed File Servers. When VFS detects it has to clean up the remnants of a File Server process (i.e., through an `exit(2)`), it marks all associated file descriptors as invalid and cancels ongoing and pending requests to that File Server. Resources that were in use by the File Server are cleaned up.

- [0]  <http://wiki.minix3.org/en/DevelopersGuide/VfsFsProtocol>
- [1]  <http://www.cs.vu.nl/~dcvmoole/minix/blockchar.txt>
- [2]  <http://www.minix3.org/theses/moolenbroek-multimedia-support.pdf>

MinixWiki: DevelopersGuide/VfsInternals (dernière édition le 2013-03-31 13:59:15 par LokeBharti)