Open Software Foundation
Request For Comments: 78.0
January 1996
T. Anderson (Transarc)

EPISODE VNODE SYNCHRONIZATION

1. INTRODUCTION

This document describes a reorganization of the vnode synchronization code used by Episode planned for the OSF/DCE 1.2 release. It was developed for Transarc's September 1994 DCE/DFS 1.0.3 release[1] on Solaris. The changes made for the September 1995 DCE/DFS 1.1 release[2] were insignificant. This code is also present in the IBM DFS Product for DCE V1.1 and V1.0.3.

As used in this document, **vnode synchronization** refers to maintaining the consistency of vnodes and the data structures that surround them. It mostly excludes measures intended to protect the files and directories that the vnodes represent. For instance, the traditional vnode lock (`vd_tlock`) protects the directory contents and file link counts and similar concepts. This document makes only passing reference to this lock. Similarly, the management of virtual memory associated with vnodes is the province of another document.[3]

The core of the idea was to replace the existing **B&CV** (''bits and condition variable'') model with locks held over the appropriate interval. The use of locks is not universally preferred, however. In cases where it is common not to block on a resource but to take some other action or where the holder of the resource is not the same as the thread that releases it, a B&CV mechanism is probably clearer. For instance, waiting for an open volume[4] is not implemented by seeking a lock for the duration.

A related issue is that the old vnode synchronization model used the vnode lock for essentially everything. This gross overloading of a single lock imposes a good deal of complexity and confusion that can be avoided by using several locks each protecting a well-defined set of fields or operations.

I wanted to avoid assuming that volume operations are generally single threaded and to avoid, to the extent possible, depending on `vol_open()` to stop vnode operations. These assumptions are complicated, non-local, and violate the modular hierarchy. If Episode (or at least the vnode synchronization) can be made self-consistent without worrying about the workings of `StartVnodeOp()`, it would be more robust and easier to understand.

We use the vnode reference count freely within Episode to prevent vnodes from being transmogrified during operations that must drop locks when they sleep. This requires that a few routines (e.g., `Recycle()`) carefully check the refcount.

The old model had several fuzzy states, especially regarding open volumes, which are made explicit. This removes some confusing ambiguities and overloadings and allows more comprehensive assertions that the system is operating correctly.

Consideration of the quota reservation problem is deferred to the VM reorganization, which does away with reservation altogether (see RFC 75.0).

The solution to the problem of handling `VN_RELE()` on **ZLC** (zero link count) files in face of fileset operations[5] is to add careful volume glue[6] to `VOP_INACTIVE`. This allows `vnm_Inactive()` to safely avoid deleting

---

1. The V1.0.3 release is known internally as dfs-fwd-1.32.
2. The V1.1 release is known internally as dfs-perf-1.42.
3. See description of Episode's virtual memory integration in RFC 75.0.
4. The terms **volume** and **fileset** are used interchangeably.
5. Transarc defect db5505.
6. The term **glue** used in DFS refers to two mechanisms which add a layer of processing on all vnode operations. The first type of glue (**token glue**) obtains the proper type of tokens to provide single-site semantics to local operations on ufs or (locally mounted) Episode vnodes. The second type of glue (**volume glue**) ensures that no vnode operations run concurrently with incompatible volume operations. The file system independent portion of the VFS+ Interface interposes this glue as appropriate.

unlinked files during volume operations.

2. RATIONALE

An important goal of the reorganization was to improve the maintainability of the vnode synchronization mechanisms. There are two general components to the maintainability problem. First, the old mechanisms were so complex and non-local that understanding them was nearly impossible. Improvements here will reduce long term maintenance costs.

Second, analysis of past defects indicates that more bugs were expected as testing of Episode extended into new regimes. Plans for the DCE/DFS 1.1 release called for more intensive fileset operations and routine use of locally mounted filesets. Both of these were likely to expose new failure modes. A new mechanism with well-defined invariants and based on a deeper understanding of the operational environment faced by the vnode system would eliminate bugs and make diagnosis and repair of remaining problems easier and faster.

A primary reason for the shortcomings of the old mechanism is that it was developed incrementally, in response to an evolving understanding of its requirements. It was enhanced to handle virtual memory (VM) integration, phantomized and stale vnodes and use by fileset operations. These capabilities were *not* designed in. We now know far more about the requirements placed upon the vnode system than we did in 1990 when it was first developed based on the BSD 4.3 model used by the IBM RT AOS4 operating system.

3. VIRTUAL MEMORY REQUIREMENTS

I refer to `PageIn()` as the generic request from the VM system to fill a page by reading data from the file system. `PageIn()` also handles "minor page faults". Similarly, I use `PageOut()` to refer to the generic VM request to clean a page by writing it back to the file system. I use the terms **VM** and **page cache** synonymously.

On AIX, VM resources are locked before `PageIn()` is called. AIX vnodes that use VM contain an object called a **VM segment**. All VM accesses must use this segment, which must be created before the first such access. Both vnode reclamation and file deletion must delete this segment. Deleting the VM segment waits for all pending `PageIn()` calls to finish. On AIX, a zero reference count does not imply that there are no outstanding `PageIn()` requests. The existence of a segment is the only safe indicator. Further, deleting a VM segment is a destructive operation that must not be attempted on a vnode that is in use.

On AIX, `Recycle()`[7] must exclude new VM users, without blocking `PageIn()`, before deleting the VM segment. The inactive[8] procedure must also do this, but for the more obvious reason that it may delete the underlying file. When `PageIn()` is blocked by a volume operation, or if the reference count is non-zero, `Recycle()` must avoid this vnode. Otherwise it may safely delete the VM segment and proceed with the rest of the reclamation activities.

On AIX, `StopUse()`[9] cannot delete the VM segment in order to pacify the page cache since it must be able to operate on held vnodes. However, it must block `PageIn()` calls then clean or invalidate VM. Because it blocks `PageIn()`, it must carefully interact with `Recycle()`.

> *NOTE: We assume that prohibiting reclamation of all vnodes for open volumes would consume too many vnodes. The problem with banning* `Recycle()` *is that some volume operations are coded to use vnodes primitives (e.g., dump and restore). Since there is no acceptable bound on the size of the largest volume we also cannot specify a reasonable bound for the number of vnodes this would consume. However, prohibiting* `Recycle()` *during volume operations might be more attractive if the volume operations were*

---

7. The function `Recycle()` (described in Procedure Outlines) takes an unused vnode representing one file and removes its identity so that it can be used to represent a different file.

8. The function `vnm_Inactive()` (described in Procedure Outlines) is called by `VN_RELE()`, which calls `VOP_INACTIVE()` when the vnode's reference count drops to zero. For Episode vnodes this turns into a call to `vnm_inactive()`, which handles cleanup for referenced vnodes.

9. The function `vnm_StopUse()` (described in Procedure Outlines) is used to put a fileset's vnodes into a state compatible with a volume operation. Generally this involves cleaning or invalidating cached data for the vnode.

changed to avoid vnode primitives. This might be a future simplification.

On SunOS, the file system specific `PageIn()` procedure consults VM, locks pages, and so forth. The SunOS VM system calls `PageIn()` with a held vnode. This rules out any synchronization mechanism that uses a lock on the reference count to exclude VM users; the reference count lock must be at the bottom of the locking hierarchy.

On SunOS, the reference count is a reliable indicator of potential `PageIn()` requests. `Recycle()` and `StopUse()` can both block VM users in `PageIn()` and `Recycle()` can avoid vnodes with non-zero reference counts. There are no dependencies between `Recycle()` and `StopUse()`.

On both platforms, we assume `PageOut()` may be called spontaneously, but only on vnodes with dirty pages.

On both platforms, volume operations use vnodes to perform certain operations. They do not use VM and the page cache is clean or invalid during these operations.

4. FILESET OPERATIONS

When a volume is opened, inconsistent vnode operations are blocked. Operations *never* blocked by the volume glue are `PageOut()`,[10] `VOP_INACTIVE()` (although, see Deleting Files on special treatment for ZLC files), `VFS_ROOT()`, and `VOPX_GETVOLUME()`. The `PageIn()`[11] operation is glued on SunOS but not on AIX where it is called at page-fault level. Each volume operation places specific requirements on the cached vnode state. The semantics of each operation specify whether the cached data must be invalidated or not and which vnode operations may run concurrently. All volume operations assume that the fileset is consistent and quiescent for the duration.

An open volume is the exclusive province of the thread that opened it. This means that it may not be modified either implicitly or explicitly by other users. All vnode operations that modify status or data are inconsistent with open volumes regardless of their open mode and are blocked. Deleting ZLC files is therefore prohibited whenever a volume is open. To accomplish this we defer the last release (see Deleting Files, below). Implicit updates to a file's `atime` are discarded during volume operations.

The implementation of Episode imposes other constraints. Some volume operations utilize the anode layer functions directly, some use vnode primitives, and some make no use of per-file data at all. No volume operations reference or update VM. This means each volume operation[12] must be preceded by a step (`vnm_StopUse()`) which puts the vnode system, including VM, into an appropriate state.

Certain state information describing the current volume operation is maintained for each volume. This information is derived from the volume handle's file system private data pointer via the ''`vnvl_`'' module. The locking problems in the `xvolume` layer caused some difficulties in this area. In particular, it is not clear what lock protects the FS private data structure. However, the data in that structure changes infrequently (e.g., at the beginning and end of volume operations) and access is generally single threaded when it does. This can be cleaned up when the locking of volume handles is clarified.

The open volume state includes the following information:

(a) `vld_openbits` — These bits indicate the types of vnode caching that are allowed. They are derived from the current volume operation and the `vol_open()` modes. The appropriate open bits may change as an open volume progresses through a series of operations. The vnode system must be notified of these

---

10. `PageOut()` must not be blocked, otherwise `StopUse()` could not clean dirty pages.

11. On AIX, the vnode operation is `VOP_STRATEGY()`, which is called at page-fault level and so never blocks. The actual handling of `PageIn()` or `PageOut()` requests is done by a kernel process. It must do its own volume synchronization.

On SunOS, the glue treats `getpage()` as a `READWRITE` operation, so blocks it during most volume operations. The Episode code does enough volume synchronization to block writes and return unwritable page mappings during `READONLY` volume operations.

12. In the present system, `vnm_StopUse()` cannot be called during certain volume operations because of `xvolume` layer locking problems. As a result `vnm_StopUse()` is called in a rather *ad hoc* manner from places like `vol_efsOpen()` and `vol_efsScan()`. Assertions on the current `StopUse()` open bits are present in various volume operations to ensure that the correct level of VM/vnode/anode consistency is present.

changes by calls to `vnm_StopUse()`.

While these open bits allow for many possibilities, only a handful are actually used. These are given vaguely suggestive names with the following descriptions:

(i) `open-change-id` — The identity of the volume is being changed, but the contents are unaffected; this means swapid for clone or intra-server move. It requires writing all cached data through to the anode layer, write-protecting the page mappings, and closing the anode handle, but cached data do not have to be invalidated.

(ii) `open-change-anode` — The fileset's contents are being changed at the anode layer; this means swapid for replica-release, reclone (backing fileset), unclone (backing fileset), detach and destroy.[13] It requires writing through and invalidating VM and other cached state, such as the atime, and closing the anode handle.

(iii) `open-read-anode` — The fileset's contents are being examined by means of anode-layer primitives; this means clone and reclone (front fileset). It requires writing all cached data through to the anode layer, write-protecting the page mappings. The cached data do not have to be invalidated, since the contents of the fileset are not being changed.

(iv) `open-change-vnode` — The fileset's contents are being changed but by means of vnode-using primitives; this means restore. Since vnodes will be used during this operation, it is pointless to flush cached status or close the anode handle. However, since these modifications do not go through VM, the page cache must be cleaned and invalidated.

(v) `open-read-vnode` — The fileset's contents are only being examined by means of vnode-using primitives; this means dump. This state is analogous to open-change-vnode, except that the contents of the fileset are not being changed. The requirements are the same except that the existing VM does not have to be invalidated but must instead be write-protected. Unclone (front fileset) is also in this category.[14]

(vi) `open-noop` — The fileset header is being examined or modified (gently); this means getstatus, setquota and similar operations. These have no impact on the vnode pool.

(b) `sideRail` — On AIX, a list of deferred VM requests.

5. PER-VNODE STATE

This section describes some of the Episode vnode fields that relate to vnode synchronization. These fields describe the vnode's identity, its underlying anode representation, the data cached in the vnode or bits describing cached data.

These are protected using the `efs_lockvp()` and `efs_unlockvp()` macros to manipulate the operating system specific lock protecting the vnode reference count:

(a) `v_count` — The vnode's reference count. This field is really in the system independent part of the vnode, but Episode makes some references to it and the associated lock.

(b) (held) — This is not an explicit bit, but means that the `v_count` is at least one.

These are protected by the global `vntableLock`:

---

13. A special open mode could be recognized for destroy that knows that the containing files are being deleted. `StopUse()` could then expunge dirty pages without being required to write them through. This might provide a useful speedup for temporary filesets.

14. Ideally we could allow the front volume of a volume being uncloned to be opened in the weakest possible way. This is because we don't really care about the state of the fileset being uncloned, it doesn't need to represent a consistent snapshot so writes to the front fileset do not need to be interrupted. However, the locking that protects the VM system's use of the block map is not at a convenient level for the unclone operation. This could be fixed by locating each vnode in the front fileset, grabbing the `vd_vm.lock` and then doing the unclone.

(a) vd_fid — The file identifier for the vnode. The vd_idLock must also be held to change this field.

(b) (NoIdentity) — This isn't an explicit bit, but is true iff fid is invalid. If the vnode is held and NoIdentity, the vnode can be referred to as **stale**.[15] All vnodes with identities are in the vnode hash table so NoIdentity is synonymous with !vd_onHash. A vnode may not *lose* its identify unless the vd_idLock is locked. A NoIdentity vnode will not *gain* an identity while it is held, in other words, a stale vnode will stay that way until released.

(c) vd_volid — The containing volume's ID. Only valid if fid also valid.

(d) vd_onHash — In the vnode hash table. Implies !NoIdentity.

(e) vd_label — Vnode iteration label, see vnm_StopUse().

(f) vd_onLRU — On the LRU list.

(g) vd_avoidRecycle — Is set by StopUse() when it has blocked PageIn() for a vnode with a VM segment. Used by ObtainVnode() to traverse the LRU list without dropping the vntableLock.

The vd_cache substructure contains fields which describe frequently changing attributes cached in the vnode. This includes the access, modification and change times as well as the file contents data version number. The fileset version number (VV) is not cached because its value is maintained on a per-volume basis. The intent to change the VV is indicated by a dirty ctime.

The vd_cache fields are protected by the vd_cache.lock:

(a) noChange — atime updates are ignored.

(b) noStatus — Unused, parallels noAnode.

(c) noDirty — No unwritten status updates.

(d) new[AMC]time — A bit for each time value, set if dirty.

(e) last[AMC]time — Current time values.

(f) dataVersion — Current file data version number.

The vd_file substructure contains fields relating to the anode level representation of the file. These fields, especially the anode handle, are used by almost all vnode operations without the protection of the vd_file.lock (see Locks). The vd_file.lock must be locked when these fields are changed:

(a) noAnode — Anode handle must stay closed. Both volume and vnode operations that require the anode handle should fail.

(b) noDelete — Inactivation of vnodes for ZLC files does not reclaim space. This is set in various cases where deleting files would be a bad idea, for example during volume ops and when the fileset is readonly.

(c) readonly — File is in a readonly volume. This bit is derived from the volume header and is ignored if the volume is open. This is intended to cover replicated and backup filesets. Filesets which are locally mounted R/O are handled separately (see definition of EV_ISREADONLY). Filesets on R/O media will presumably need a new mechanism to inform Episode of the readonly-ness of the media or hardware.

(d) anode — Anode handle is open.

(e) anodeRO — File is unwritable due to the presence of a **COW** (copy on write) file (namely, (copies>0)). This is intended to handle cases where an interrupted fileset operation leaves some files in an otherwise R/W fileset with a COW reference.

15. Stale vnodes can be created by vnm_Delete() and vnm_StopUse(), if a file is held across certain fileset operations such as restore, reclone or destroy. All operations on all such vnodes fail with ESTALE, except VOP_INACTIVE().

(f) ap — Anode handle representing fid.

The vd_vm substructure contains various bits describing the state of the VM system as it relates to the vnode. These fields are protected by the vd_vm.lock:

(a) noReadable — Requests for new page mappings block. Implies noWritable.

(b) noWritable — Requests to return writable page mappings block.

(c) readonly — Requests for writable page mappings that cannot be satisfied with write-protected mappings fail.

(d) valid — Valid pages may exist.

(e) dirty — Modified pages may exist.

(f) seg — On AIX, has a VM segment (kept in vd_seg). The creation and deletion of the VM segment are protected by the vd_idLock. Never set on SunOS.

5.1. Open Volume Restrictions

When a volume is opened, six bits are specified (openbits) that describe the allowable states for cached data associated with the vnode. Each bit controls part of the vnode state space. Setting the bit will restrict the vnode from entering that state. Of course, it may already be in the restricted state, so the function that modifies these bits (vnm_StopUse()) can also move the vnode out of the restricted state when setting any bit. These bits then can refer both to state restrictions and to a process for forcing the vnode out of the restricted state. For example, specifying STOPUSE_NO_DIRTYVM prevents the creation of writable page mappings (by setting vd_vm.noWritable), and cleans all dirty pages (by calling vnvm_Clean()).

Operations that violate open volume restrictions must be blocked by volume synchronization (i.e., in vol_StartVnodeOp()), rejected by the volume ops dispatch code (in afscall_volser() using the VOLCHECK() macro) or handled by Episode (e.g., getpage() returns only R/O pages). For example, vnode-using primitives invoked by volume operations that conflict with the volume open bits must be avoided (e.g., specifying STOPUSE_NO_DIRTY is incompatible with restore operations).

When the volume is closed, vnm_StopUse() is called with an openbits value of zero which returns all vnodes to normal operation.

The function SetRestrictions() processes these openbits to set or clear the various restriction bits in the vnode:

(a) STOPUSE_NO_CHANGE — Containing fileset is open. Operationally this bit is assumed if any bits are specified. Attempts to delete zero link count files are ignored (calls to vnm_inactive() on such files should be deferred anyway). Implicit updates of atime are discarded.

(b) STOPUSE_NO_ANODE — Anode handle must be closed. Calls to PageIn() should *block* and attempts to open the anode handle should *panic*.[16] Implies STOPUSE_NO_DIRTY.

(c) STOPUSE_NO_STATUS — All cached status data must be written through and invalidated. Operations that reference status data should fail. Implies STOPUSE_NO_VM and STOPUSE_NO_DIRTY.

(d) STOPUSE_NO_DIRTY — Updates to cached status data must be written through. Operations that modify status data explicitly should fail. Implies STOPUSE_NO_DIRTYVM.

(e) STOPUSE_NO_VM — The VM must be written through and invalidated. Calls to PageIn() should *block*. Implies STOPUSE_NO_DIRTYVM.

16. Episode can view failures of the volume synchronization glue quite seriously since the consistency of these operations is restricted to the kernel. The xvolume layer will reject volume operations that are inconsistent with the volume open mode so user-space errors should not lead to panics.

(f) `STOPUSE_NO_DIRTYVM` — The VM must be written through. Calls to `PageIn()` that require writable page mappings must *block*; others may return write-protected mappings.

To summarize the open modes use these `openbits` bits in addition to `noChange`:

(a) `open-change-id` — `NO_ANODE+NO_DIRTY`
Swapid (for clone and intra-server move).

(b) `open-change-anode` — `NO_ANODE+NO_STATUS+NO_DIRTY`
Swapid (for replica release), reclone (backing fileset), unclone (backing fileset), destroy.

(c) `open-change-vnode` — `NO_VM`
Restore.

(d) `open-read-anode` — `NO_DIRTY`
Clone, reclone (front fileset).

(e) `open-read-vnode` — `NO_DIRTYVM`
Dump, unclone (front fileset).

(f) `open-noop` — `0`
Fileset header operations.

6. GLOBAL DATA STRUCTURES

In addition to per-vnode fields, vnode synchronization uses several global structures. They are all protected by the `vntableLock`:

(a) `vntable` — Hash table containing all vnodes with identities. The hash index is a function of the volid and the fid's index. A vnode on this list has `onHash` set.

(b) `vntableLabel` — Counter used by vnode iteration procedures. See `vnm_StopUse()`.

(c) `lruList` — Contains all unused vnodes in least recently used order. Vnodes are added to the list by inactive and removed by `ObtainVnode()`. They may be held or not and may have an identity or not. A vnode on this list has `onLRU` set.

(d) `staleList` — Contains vnodes with neither `onHash` nor `onLRU` set. This prevents us from completely losing track of stale but held vnodes. This fifo shares the lru fifo's thread.

(e) `vnCount` — Is the number of currently allocated vnodes.

(f) `vnCountTarget` — Is the preferred number of vnodes, which can be set at initialization time. Free unused vnodes if we have allocated more than this.

(g) `vnCountMax` — Never allocate more vnodes than this. Return `ENFILE` instead. This is also an configuration parameter; if it is zero no hard upper-bound on the number of vnodes is enforced.

7. PROCEDURE OUTLINES

This section briefly describes the primary functions involved with vnode synchronization:[17]

(a) `vnm_FindVnode()` — Returns a held vnode representing a fid. It locates an existing vnode or obtains an unused vnode (allocating a new one if necessary) by calling `ObtainVnode()` then initializes the vnode by calling `OpenVnode()`. The caller must not have started a transaction.

(b) `vnm_Allocate()` — Returns a vnode without an identity for use by a yet-to-be-created file. The caller must not have started a transaction.

---

17. For additional details see the code in `file/episode/vnops/efs_vnode.c`.

(c) `vnm_SetIdentity()` — Takes a `NoIdentity` vnode and the fid of a new file and its already opened anode handle, and makes the vnode refer to the file. We rely on the fact that the anode handle of a newly created file is inaccessible to other users. The case of racing `VGET`'s for colliding indexes is handled by waiting for those threads to notice that all fid's with this index are stale.

This is called when a transaction has already been started so we must avoid heavyweight operations that could start a transaction.

(d) `ObtainVnode()` — Locates a vnode for use by a file. If the fid is specified and a vnode for that fid already exists that vnode is returned. Otherwise a fresh vnode is obtained, either by calling `Recycle()` or by allocating a new one.

(e) `OpenVnode()` — Attaches a vnode without an identity to a specified fid. This can fail if a different vnode for that fid already exists. It can also fail if the fid does not match an existing file. If called from `vnm_SetIdentity()` an anode handle for a newly created file is provided which we use instead of calling `epif_Open()`.

This function uses the `vd_idLock` to ensure that there is never more than one vnode referring to a file.

(f) `Recycle()` — Attempts to obliterate an unused vnode's current identity by reclaiming its resources, then clearing its identity. If the specified vnode is being used or if, on AIX, its VM segment cannot be deleted, the vnode is left untouched.

This function holds the `vd_idLock` while checking the vnode's reference count. Since `OpenVnode()` grabs this lock also, this ensures that `Recycle()` has (nearly) exclusive use of the vnode and that destroying its identity is safe.

(g) `vnm_inactive()` — Does cleanup processing on a vnode when the caller is dropping the last reference to it. Has no effect if the reference count is greater than one. It deletes zero link count files on non-open R/W volumes. On AIX, it deletes the VM segment of a stale vnode.

(h) `vnm_Delete()` — This deletes the file corresponding to a particular `volp`/fid making the vnode stale if it is in use. Unlike inactive it takes no consideration of the vnode's `refCount` or the file's `linkCount`. Used by `vol_efsDelete()` during restore.

(i) `vnm_StopUse()` — Takes a volume and an open mode. It traverses all the vnodes for that volume and puts them into a state compatible with the open mode. It is called with an open mode of zero before these operations are released when the volume is being closed.

We assume that a higher level lock protects the fileset so that only one thread calls this routine at a time for each fileset. It makes a single pass over the vnodes assuming that there is no ongoing activity that would reestablish any inconsistent vnode state. Before this routine starts all incompatible vnode operations are already blocked.

The basic algorithm is to use the vnode hash table to locate all vnodes of interest. This is protected by the `vntableLock` which we may not hold while flushing vnode state. So, under the `vntableLock`, we identify and hold the *next* vnode to work on. Then we drop the `vntableLock` and flush the *current* vnode. The next vnode can become stale while the current vnode is being processed. In this rare case, the iteration is restarted.

After each vnode is made consistent with the current volume open mode it is marked with the label associated with the hash table when we started. This allows us to inexpensively skip it if we must restart the iteration. Any vnode created while `vnm_StopUse()` is running will be labeled with the current label (or a subsequent one) by `SetIdentity()` and its `openbits` are initialized to the same value by `SetRestrictions()` called from `OpenVnode()`. `vnm_StopUse()` can safely skip these also. Therefore all processed vnodes will remain consistent with the current volume open mode because their restriction bits have been set correctly and since only compatible vnode operations will be running.

If `STOPUSE_NO_ANODE` is being cleared, the anode handle of each vnode is, of course, reopened. If this fails (e.g., because a reclone operation removed files from the backing fileset) then the vnode is made stale.

(j) `SetRestrictions()` — Called by `OpenVnode()` to initialize a new vnode or by `StopUse()` to calculate the vnode state restrictions from the current volume open bits. It is called after the vnode has been put into a state consistent with the current open mode (e.g., if `STOPUSE_NO_ANODE` is specified it sets `vd_file.noAnode` and asserts that `vd_file.anode` is false).

8. DELETING FILES

Here is a summary of the steps we take to provide correct semantics for ZLC files. We carefully distinguish between *unlink* (or *remove*), which reduces a file's link count, and *delete*, which deallocates all the storage for a file and frees its anode.

When a file is unlinked and its link count reaches zero, it becomes a candidate for deletion. The glue functions that can unlink files and the file system independent fileset restore code passes the vnode to the ZLC Manager which attempts to obtain an ''open for delete'' token. It holds the vnode until it succeeds in this. When the delete token has been obtained, the ZLC Manager knows that no remote users are using the file, so it releases the vnode.

When the vnode's reference count reaches zero, `VOP_INACTIVE()` is invoked by the file system independent macro `VN_RELE()`. This will call a glue (file system independent) function which calls `vol_StartBusyOp()`, which fails if the containing volume is open. In this case the vnode is held on a per-volume (`volp->v_vp`) list until the volume is closed. The `vol_close()` operation will release these vnodes again. In any case, `VOP_INACTIVE()` returns successfully.

Once `StartBusyOp()` succeeds, `vnm_inactive()` is called. This operation will actually delete the file if its link count is zero unless the volume is open[18] or readonly.

The entire foregoing mechanism can be harmlessly invoked on files whose link count is not zero. Indeed, `VOP_INACTIVE()` cannot safely check the link count during some volume operations and so must defer inactivating vnodes released during those operations.

The above procedures ensure that, when `vnm_inactive()` receives a vnode whose link count is zero and which is not in a readonly volume, no volume operation is in progress and none will start while it is running, and the delete token has been obtained. Thus, it will be safe to delete the file.

The changes to add glue to `VOP_INACTIVE()` were done under db5505.

8.1. Undeleting Files During Restore

The old code used to delete each file before restoring it. This resulted in a requirement to undelete these files in the common case where the new file's fid matches the old file's fid and reattach the new file to the old vnode. This created all sorts of problems and this behavior was changed.[19]

8.2. Making Stale Vnodes

The only way to create a stale vnode (i.e., held and `NoIdentity`) is via a fileset operation. This statement, however, needs a slight qualification. Actually non-explicitly held vnodes, for instance those held during hash table traversal, can become stale. An explicit hold is a `FAST_VN_HOLD` followed by a check of the fid under the protection of the `vd_idlock` (see the description of the `vd_idlock` in Locks). The `vnm_FindVnode` function always returns explicitly held vnodes. A vnode with only an internal hold can become stale if `vnm_inactive()` or `Recycle()` is running concurrently.

All vnode operations operate on explicitly held vnodes, so any volume operation that can produce stale vnodes is inconsistent with (virtually) all vnode operations. Therefore vnode operations generally need only check for stale

---

18. `StartBusyOp()` always succeeds for the thread which has the volume open. This will allow `vnm_inactive()` to be called on vnodes in open volumes (e.g., during dump or restore). Under normal operation there should be no way that a volume operation could release a vnode for a file with a link count of zero since the vnode should have been passed to the ZLC Manager by restore for token management. Just to be safe, however, we avoid deleting files in open volumes.

19. Transarc defect db5449: ''Fileset restore need not delete file on creation always.''

vnodes on entry, typically using the EV_DEPHANTOM macro.[20] An example of an exception to this is VOPX_GETVOLUME(), which, because it is unglued, must carefully check for NoIdentity.

9. LOCKS

Here is a description of the locks used for vnode synchronization. They appear in resource hierarchy order. The StartTran resource and the SunOS page_lock are also listed to show their position in the hierarchy:

(a) rw_lock — vmmLenLock — per vnode

This is used only on AIX. See RFC 75.0 for details.

(b) rw_tlock — vd_tlock — per vnode

This lock protects the consistency of directories and some vnode interface properties such as the link count. It is held throughout most vnode ops. It is not used to protect the consistency of the vnodes themselves, but only the objects the vnodes represent.

(c) mutex — vd_idlock — per vnode

This lock protects a vnode's id from changing. Procedures that destroy a vnode's identity during normal operation, namely vnm_inactive() and Recycle(), act only on unused vnodes (see Making Stale Vnodes for a discussion of how a vnode that is in use can have its identity removed). These functions must check the reference count after grabbing the vd_idlock. This prevents races between OpenVnode() and vnm_inactive() or Recycle(). Users that have explicitly requested a vnode for a particular fid must call vnm_FindVnode(), which calls OpenVnode(). That routine must lock the vd_idlock while verifying that the vnode (from ObtainVnode()) contains the requested fid and that the fid matches an existing file. While the vd_idlock is held neither vnm_inactive() nor Recycle() can clear the vnode's identity, and after the vd_idlock is released both of these routines will notice the vnode is in use and skip it.

This contrasts with non-explicit (or internal) vnode holders (those which may bump the reference count from zero on a vnode without concern for its identity) which are of two types: hash table iterators and the VM system. The former obtain the vd_idlock after holding each vnode and check that the vnode is not stale. In the latter case the VM is cleared in the process of removing the vnode's identity, so VM requests on stale vnodes can safely be ignored. Synchronization with volume operations must be carefully considered in these cases. If vnode-destroying volume operations may be running concurrently then careful examination of the vnode state under protection of the vd_idlock is safe. If such operations are known to be excluded (which is the case for most vnode operations) then checking the identity is safe as long as the vd_idlock was grabbed at some point since the vnode was held.

Basically obtaining the vd_idlock converts an internal hold into an explicit hold.

To determine that a fid represents an existing file it must be passed to epif_Open() to obtain the anode handle (kept in vd_file.ap). This handle is always valid in vnodes with identities except during volume operations that require disconnection between the anode and vnode representations.

Because the ID lock is held for the duration of vnm_inactive() and Recycle(), it is a high level lock that can be held across VM operations. It must not be used in PageIn() or PageOut().

On AIX, StopUse() must also hold this lock to exclude vnm_inactive()[21] and Recycle() while it blocks incompatible VM operations. This prevents the VM segment deletion from deadlocking with blocked page faults.

---

20. The name of this macro is a historical artifact: the process of applying StopUse() to a vnode used to be called ''phantomization''.

21. Of course, vnm_inactive() should already be blocked by the volume glue in VOP_INACTIVE().

(d) mutex — `vd_vm.lock` — per vnode

This lock protects the vnode during state transitions related to the virtual memory page cache. It covers both the bits specifying restrictions as well as the advisory bits describing the current state of the page cache. It is used during `PageIn()` but not `PageOut()`. On AIX, the this lock protects the use of the VM segment. See RFC 75.0 for details.

(e) rw_lock — `vd_file.lock` — per vnode

The lock protects the consistency of the anode's length and block map. It used by functions that reference or modify a file's allocation map, for example `epia_Truncate()`. An exception to this rule is that `PageOut()` does not use this lock, as that would make `PageOut()` implicitly depend upon `PageIn()`. Instead `PageOut()` relies on the VM system's page lock when examining the block map while writing out a dirty page. See RFC 75.0 for details.

(f) *resource* — `StartTran` — global

(Call `elbb_StartTran()` to begin a transaction.)

(g) mutex — `vd_cache.lock` — per vnode

This lock protects the cached vnode status information, namely the three times and data version number. It is used when updating the times and when flushing them through to the anode.

As this lock is below `StartTran` in the resource hierarchy, we may not start a transaction while holding the lock. This leads to some awkward code in `vnm_UpdateAnode()`, which can be called without an already started transaction, but needs a transaction to write through dirty status.

(h) rw_lock — `page_lock` — per page [SunOS]

(The SunOS page lock.)

(i) mutex — `vntableLock` — global

This lock protects the fid, volid, iteration label, and `onHash` and `onLRU` bits of all vnodes. It also protects the global state such as the hash table and LRU and stale lists.

(j) mutex — `refCountMutex` — per vnode [AIX]
mutex — `v_lock` — per vnode [SunOS]

This protects the vnode's reference count and is referenced via the `efs_lockvp()` and `efs_unlockvp()` macros. On AIX, we added this mutex to protect our uses of the `v_count` field; the AIX kernel makes very little use of this field.

AUTHOR'S ADDRESS

Ted Anderson
Transarc Corporation
707 Grant St.
Pittsburgh, PA 15219
USA

Internet email: ted_anderson@transarc.com
Telephone: +1-412-338-4410