Open Software Foundation
Request For Comments: 75.0
October 1995

Blake Lewis (Transarc)

EPISODE VM INTEGRATION

1. INTRODUCTION

This document describes the reorganization of the Episode I/O path for the 10/94 release. This work was undertaken with the following goals:

(a) Improve write latency (eliminate unnecessary synchronous writes).

(b) Eliminate disk block reservation system.

(c) Simplify source code (improved maintainability, portability).

(d) Reduce CPU usage.

(e) Maintain throughput; no deleterious side-effects.

(f) Get correct ‘‘zerofill’’ semantics (uninitialized parts of the file always contain zeroes).

The main result of the reorganization is the separation of disk block allocation and file promotion (i.e., changes in on-disk representation) from the write path, allowing us to perform these functions at a higher level in advance of scheduling any disk writes.

1.1. Advantages

(a) The strategy (disk I/O) path becomes much simpler. In-line files need a bit of special handling, but otherwise strategy becomes essentially a wrapper for the device driver strategy routine. There is no need for synchronous ‘‘slow path’’ writes.

(b) The reservation system disappears. We no longer need to reserve space since it is now allocated in advance.

(c) The new interfaces will let getpage detect holes and unwritable backing storage and turn off write permission when reading from these.

(d) Because they know how the disk blocks are laid out, getpage and putpage will only lock as many as pages at a time as they can transfer in a single disk I/O. Presently, they lock arbitrarily sized page lists, potentially over several disk I/O’s.

1.2. Drawbacks

The current lazy allocation scheme permits a simple block allocator (or at least minimizes its deficiencies). Doing allocation in advance requires a more complex mechanism so that concurrent updates will find contiguous regions of disk blocks without interfering with each other. UFS may offer some useful lessons. So far, the existing allocator appears adequate. However, we should do further investigation as part of any serious effort to improve performance.

1.3. Simplifying Assumptions

In order to simplify the implementation, we impose the following constraints on the configuration of the file system:

(a) Page size ≤ file system block size.

(b) Disk block (sector) size ≤ file system fragment size.

(c) All sizes are powers of two (so that every smaller unit divides evenly into every larger unit).

2. NEW INTERFACES

(a) int efs_MakeWritableBlocks(vp, off, len, flags);

Makes sure that writable disk storage backs vnode vp over the range specified by off and len, allocating new disk blocks if necessary. If COW blocks back any part of the range, their contents is copied into writable blocks (unless the flags say not to — see below). Returns an error if the allocation cannot be done. Called with the vnode's file and vm locks held.

efs_vmwrite (VOP_WRITE) and efs_getpage (VOP_GETPAGE) use this routine to do any necessary disk allocation before they initiate a write. vnm_Truncate also uses it to handle truncations into a COW block.

The flags contain three bits of information. MWB_ZEROFILL indicates that any newly allocated blocks should be initialized with zeroes. We set this flags at all times except during volume write. The volume operations always initialize allocated blocks before making them accessible.

If MWB_USE_VM is set, we do the zeroing by creating dirty, zero-filled pages to cover the allocated blocks, thus avoiding the cost of an immediate write and also a read when VOP_GETPAGE is subsequently called. If it is not set, we zero the blocks directly by a disk write. We use MWB_USE_VM on SunOS in all cases except volume writes (when we bypass the VM system); because of the limitations of the AIX VM system, we cannot use it on AIX.

MWB_OVERWRITE is set in cases where we know that we are going to replace the entire contents of the specified range. Thus, we do not bother copying any data from COW blocks.

Assumption: vp has the correct storage type (no promotion needed).

(b) int efs_FindBlocks(vp, off, blockP, wrLenP, rdLenP);

Find disk blocks corresponding to (vp, off), without allocating. *blockP is the disk address of the first block; *wrLenP is the number of bytes of contiguous, writable allocated disk space starting at this address and *rdLenP is the number of readable bytes. Note that the returned disk address is in units of *disk blocks*, not file system blocks, so that this interface can be used for both blocked and fragmented files. Also, the returned lengths need not be maximal. The implementation will likely stop searching when it reaches a natural boundary, e.g., at the end of an indirect block. Finally, if there is a hole at (vp, off), *blockP will contain EPIX_ANODE_BLOCKEMPTY, and both *rdLenP and *wrLenP will be zero. Called with the vnode's file lock held.

(c) int efs_FindWritableBlocks(vp, off, blockP, lenP);

Find disk blocks for the given vp and offset, without allocating. *blockP is the disk address of the first block; *lenP is the number of contiguous writable bytes starting at this address. Like FindBlocks, except that it only returns information about writable blocks, so that it can return immediately if it encounters a read-only block. Called with the vnode's file lock held.

(d) long efs_CheckAllocation(ap, off, dblkP, lenP, isCowP);

This is the lower-level interface used by the two preceding functions. It returns the disk address for anode ap at offset off (or EFS_HOLE if this is a hole), and the length in bytes of the number of "similar" blocks beginning at that address. That is, if there is a writable block at offset off, *lenP will be the number of contiguous writable bytes starting at that offset, and similarly for COW blocks and holes. isCowP indicates whether the region consists of COW blocks. Must not be used with in-line files.

(e) off_t efs_NextBlock(vp, off, len);
off_t efs_NextWritableBlock(vp, off, len);

Return the offset at which the next (writable) disk block appears. len specifies how far to search (0 means until EOF); returns -1 if no suitable block is found. These are intended to make operations on sparse files more efficient. Called with vnode's file lock held.

(f) int efs_Promote(vp, new_len, useVM);

Does any storage class conversion necessary so that vp can represent a file of the specified length, and then sets the length. The vnode must not have any allocated disk storage or pages beyond new_len when efs_Promote is called. Called with the vnode's vm lock held.

The useVM argument indicates whether efs_Promote should zero newly allocated storage by creating pages or by doing direct disk writes. It is true except during volume operations and on AIX.

(g) int efs_HasHoles(vp);

Calculate whether vp has any holes or COW blocks by comparing allocated blocks to length. Lets us avoid doing efs_FindBlocks on files that are completely allocated. Called with vnode's file lock held.

(h) int efs_DiscardVM(vp, oldLen, newLen, credp);

Invalidates all pages for vnode vp between offsets oldLen and newLen.

(i) void efs_CreateDirtyZeroPages(vp, start, end)

Cover the region of vp between start and end with dirty, zero-filled pages. This region must be backed by disk storage.

(j) int EFS_ZEROFILL_LAST_PAGE(vp, len);

Fill the page containing offset len with zeroes from len to the end of the page. Used when truncating a file downwards to make sure that the portion of the last page following EOF contains zeroes.

(k) struct buf *vol_efsGetBuf(vp, bflags, len, daddr);

Creates a struct buf to describe a volume I/O operation. The b_un.b_addr member will point to a buffer for the data.

(l) vol_efsStartIO(bp);

Interface for reading and writing without using either the VM or buffer systems, to be used by volume ops.

(m) int vol_efsBioWait(bp);

Waits for volume I/O on buf bp to complete; returns error status.

[The remaining interfaces are not currently implemented.]

(a) epia_RecordUndoZero(tranId, bp)

Record the disk blocks associated with the specified undo-zero transaction so that we can detect when the transaction can be completed.

(b) epia_UpdateUndoZero(bp, finishTran)

Following completion of a disk write, remove pending-block info. for the write and update the undo-zero pending-block count. If the count reaches zero and finishTran is true, end the transaction; if finishTran is false, put it on a queue for processing by the async daemon.

(c) epia_PurgeUndoZero(vp, off, len)

When removing allocated blocks by truncation, put any pending-block records on a special list and update the undo-zero pending-block count. Always ends the transaction if the count reaches zero.

(d) epia_ForceUndoZero(tranId)

Force completion of an undo-zero transaction by initializing all as yet unwritten disk blocks with zeroes and ending the transaction.

(e) epia_WaitForUndoZero()

Called by the transaction system when the log is full; sleeps until an undo-zero transaction has completed.

3. ZERO-FILL CORRECTNESS

Unix semantics require that reading from a part of the file where nothing has previously been written returns zeroes. Two consequences of this are that when we create a new page, any parts of it not initialized with user data must be filled with zeroes, and similarly, when we write a partially filled block to disk, the unused portions must contain zeroes. The new-block security mechanism handles situations where the machine crashes between the time that a disk block is allocated for user data and the time that the data is actually written, so we need only consider non-crash cases here, of which there are several:

(a) A file is extended into a new page (with EOF on a non-page boundary), and then extended farther via truncate.

(b) A few bytes are written into the middle of what was formerly a hole in the file.

(c) The file is truncated down to a non-page boundary and then extended via truncate.

The first two cases are handled correctly because efs_MakeWritableBlocks creates dirty zero-filled pages when it allocates new blocks. Any part of the page not subsequently initialized will therefore contain zeroes, and the zeroes will be written to disk when the page is cleaned.

EFS_ZEROFILL_LAST_PAGE handles the third case when we reduce a file's length by truncation.

We assume that these considerations only apply to ordinary user-level I/O. For instance, volume operations are sufficiently well-behaved that we do not have to worry about the cases described above.

4. LOCKING

We use three per-vnode locks (four on AIX). At the highest level, we have the per-vnode reader/writer lock. This lock is obtained upon entry to the various VOP operations. It allows concurrent access by operations that do not affect the on-disk representation of the vnode (read, getattr), while serializing those that do (write, setattr, truncate). The other two are described in RFC 78.0: the vm lock and the file lock. The vm lock serializes operations that create or destroy pages, preventing races, for instance, between page faults and file truncation; the file lock protects the contents of the anode (that is, its metadata). Finally, because of the difficulties of synchronizing between the file system and the AIX VMM, the AIX implementation requires a fourth per-vnode lock, the vmmLenLock, to protect the file length. This lock is at the top of the hierarchy.

In addition, we use an ''in-transit'' data base to keep track of disk blocks on which I/O is currently in progress. This is to prevent races where I/O is initiated through two different pathways, e.g., an ordinary asynchronous write followed by a write initiated by a volume operation. When we begin a disk write, we set an in-transit bit for each disk block and we clear this bit when the write completes. No read or write can begin until the in-transit bits for all the disk blocks involved are clear.

An awkward problem is that we cannot afford to block in VOP_PUTPAGE since delaying the pageout thread can deadlock the system. To solve this problem, the CM examines the putpage flags to distinguish pageout from other callers, and returns failure if the caller is pageout and it cannot obtain its locks without blocking, relying on pageout to make progress elsewhere. The approach that we have taken in Episode is to write efs_putpage so that it doesn't obtain any locks. Thus, we must face the prospect of pageouts happening concurrently with truncates. See the block comment preceding efs_putpage for the argument about why we can do this.

5. PSEUDO-CODE VM IMPLEMENTATION

```
/*
 * Generic interfaces
 */

/*
 * Do all length-setting & promotion here.
 *
 * Note: this function may cause a page fault when it zeroes out
 * the remainder of the final page.
```

```
 */
vnm_Truncate(vp, len, credp)
{
    if (new length too big)
        drop VMLock & return EFBIG;

    grab vm lock to prevent page creation & rival truncates;
    write-lock fileLock to change anode contents;
#ifdef AFS_AIX_ENV
    Iteratively write-protect pages beyond new EOF, flush them,
    and count write-protection faults until VM activity on this
    vnode stops.  AIX provides no way to avoid to synchronize
    with the VMM, so this is the best we can do.
#endif /* AFS_AIX_ENV */
    if (shrinking) {
        discard pages beyond new EOF;
        if (file is COW && not truncating to block boundary)
            efs_MakeWritableBlocks(vp, preceding block bndry,
                                   len, ...);
        discard on-disk storage beyond new EOF (epia_Truncate);
        if (vp does not require promotion && is not inline &&
            EOF is not in an allocation hole) {
               create zero-filled pages for the region between
               EOF and the end of allocated disk storage;
    }
    efs_Promote(vp, len);
    if (shrinking && not on page boundary)
        zero-fill last page following EOF;
    drop locks;
}

/*
 * Do the disk I/O described by a buf structure; handle in-line
 * data as a special case.  At this level, we assume that all the
 * allocation has already been done.
 *
 * Preconditions:
 *
 * Either the file is in-line or the region described by bp is backed
 * by a (physically) contiguous set of disk blocks.
 *
 * bp is handed to us initialized with count, blkno, vp, flags, and
 * whatever information is needed to identify the in-core data area
 * (b_pages or b_un.b_addr).  We will initialize the device number
 * from the anode, and set the iodone function.
 */
epia_Strategy(bp)
{
    ap = EVTOA(VTOEV(bp->b_vp));
    if (ap is in-line) {
        bp_mapin(bp);
        copy data to/from ap;
        bp_mapout(bp);
        return;
    }
```

```
    bp->b_dev = ap's device-number;
    bp->b_iodone = epia_Iodone;

    /*
     * Check in-transit state of each disk block in the transfer.
     * If we are writing, we must wait for any write already in
     * progress to finish.  If we are reading, the fact that we
     * are here means that there are no pages with the data; hence
     * the fact that a write is in progress implies that we must
     * be writing zeroes to an initialized block.  We can therefore
     * return zeroes immediately in the read case.  However, this
     * will mean partitioning the read to separate the in-transit
     * blocks from the rest.  So for now, we will just wait in
     * both cases.
     */
    wait for in-transit disk blocks;
    if (writing)
        mark all disk blocks in-transit;

    start disk I/O;
    if (!error && !synchronous)
        return;

    if (error)
        do error processing;
    else if (synchronous)
        biowait(bp);

    clear in-transit bits;
}

/*
 * Interrupt-time processing for strategy I/O.  The rules
 * are that if the I/O is synchronous, we simply wake up anyone
 * waiting for completion; if it is asynchronous, we must do
 * all necessary post-I/O cleanup.  Cleanup includes clearing
 * the in-transit bits for the blocks and marking the blocks
 * as initialized in the undo-zero database (for writes);
 * detecting completion of undo-zero transactions and waking
 * the async daemon; doing any system-specific cleanup such
 * as unlocking pages; and disposing of the buf.
 */
epia_Iodone(bp)
{
    if (bp->b_flags & B_ASYNC) {
        clear in-transit bits;
        epia_UpdateUndoZero(bp, 0);
    }
    osi_biodone(bp);
}

#ifdef AFS_SUNOS5_ENV
/*
 * SunOS-specific VM interfaces
 */
```

```
/*
 * Gather up vp's pages between off and off + len, and take some
 * action  depending on flags.  We may write out the dirty pages,
 * put them onto the free list, or destroy their identity.  All
 * allocation has already taken place by now.
 *
 * LOCKING NOTES --
 * Because we are called by the pageout daemon, we cannot afford to
 * block for extended periods here.  We therefore must not try to
 * obtain any of the per-vnode locks (the tlock, VM lock, and file
 * lock), since the holders of these locks may block, e.g., waiting
 * to start a new transaction.  The only Episode lock that we hold
 * transiently is the per-anode ap lock, whose purpose is to give us
 * a consistent view of the anode data structures (i.e., so that we
 * don't see them when they are partially updated).  We depend on
 * anode-level code not to block while holding this lock.  Of course,
 * we also acquire locks on pages, and it is these locks that
 * guarantee proper synchronization.
 *
 * We must make sure that a concurrent truncate cannot free disk
 * blocks that we are writing into, and also that promotion cannot
 * modify the on-disk representation of the anode as we are writing
 * to it.
 *
 * Before discarding any disk blocks, vnm_Truncate destroys all pages
 * beyond the new EOF.  This step will not complete until any
 * in-progress I/O on the pages has completed.  Because vnm_Truncate
 * holds the VM lock, we can be sure that the pages will not reappear
 * once they have been destroyed.  Thus, when it frees disk blocks by
 * calling epia_Truncate, vnm_Truncate is sure that no pageouts
 * beyond the new EOF are possible.
 *
 * Fragmented files present a special problem because the fragment
 * size may be less than PAGESIZE.  Thus, the last allocated fragment
 * may end in the middle of a page.  To avoid the possibility that a
 * pageout of the page covering EOF will write into newly freed
 * fragments, vnm_Truncate leaves the job of freeing fragments to
 * efs_Promote.
 *
 * efs_Promote may free fragments, change the on-disk representation
 * of the anode, and move data from one place on the disk to another.
 * To avoid races with efs_putpage, it locks all of the vnode's pages
 * before making any changes.  Thus, it cannot proceed until any
 * in-progress pageouts complete, and once it has locked the pages,
 * efs_putpage will block until it has finished updating the disk.
 * We do not risk deadlock or catastrophic performance degradation in
 * the latter case because efs_Promote does not start any
 * transactions or block for long periods.
 */
efs_putpage(vp, off, len, flags, credp)
{
    if (len == 0) /* Special case: go from off to EOF */
        return pvn_vplist_dirty(vp, off, efs_putapage, flags, credp);

    eoff = off + len;
```

```
    while (off <= o < eoff) {
        /*
         * page_lookup returns with the page exclusively locked.
         */
        pp = page_lookup(vp, o);
        if (pp == NULL)
            o += PAGESIZE;
        else {
            /*
             * pvn_getdirty tells us whether the page needs to be
             * written out; as a side-effect, it will destroy pages
             * when B_TRUNC is set and also does the appropriate
             * page lock operations.
             */
            if (pvn_getdirty(pp, flags)) {
               efs_putapage(vp, pp, &o, &delta, flags, credp);
               o += delta;
            }
            if (flags & B_TRUNC)
               epia_PurgeUndoZero(vp, pp->offset, PAGESIZE);
        }
    }
}

/*
 * Write out page pp and possibly other pages as well; return the
 * actual offset and length of the transfer in (offp, lenp).  We look
 * at the disk to see how many contiguous space exists, then try to
 * find enough pages to fill it.  At this stage, all allocation must
 * already have been done.
 */
efs_putapage(vp, pp, offp, lenp, flags, credp)
{
    efs_findWritableBlocks(vp, *offp, &blkno, &contig);
    assert(contig >= MIN(distance to EOF, PAGESIZE));
    /*
     * Find contiguous dirty pages between *offp and *offp + contig
     * and add these to the vp's page list.  Update offp and lenp to
     * describe the resulting page list.
     */
    pp = pvn_write_kluster(vp, offp, lenp, *offp, *offp + contig);
    bp = pageio_setup(vp, pp, ...);
    epia_Strategy(bp);
    if (synchronous or error) {
        epia_UpdateUndoZero(bp, 1);
        post-I/O cleanup;
    }
}

/*
 * Get vp's pages from off to off + len (possibly plus extra
 * adjoining pages).
 */
efs_getpage(vp, off, len, protp, pl, plsz, seg, addr, rw, credp)
{
```

```
    *protp = PROT_ALL;  /* default protection */
    if (efs_HasHoles(vp)) {
        if (rw != S_WRITE && rw != S_CREATE)
            turn off write permission in *protp;
        else {
        /*
         * We will already have taken care of storage class promotion
         * and extending allocation in efs_vmwrite, so the only case
         * that we handle here is filling in a hole or a COW block.
         */
            efs_MakeWritableBlocks(vp, offset, *lenp, 1);
            if (pp = page_lookup_nowait(vp, offset)) {
               /* efs_MakeWritableBlocks may have made the page for
                  us */
               pl[.] = pp;
               clean up & return;
            }
        }
    }
    eoff = off + len;
    while (off <= o < eoff) {
        delta = 0;
        while (contiguous pages starting at o exist) {
            lock next page & insert in pl array;
            delta += PAGESIZE;
        }
        if (no pages already exist) {
            /* Get page from disk (possibly plus extras) */
            delta = len;
            efs_getpage_io(vp, o, &pl[], plsz, &delta, protp, ...);
        }
        o += delta;
    }
    put a null at the end of the pl[] array;
}

/*
 * This function handles getpage requests where the required pages
 * do not already exist.  Creates a page for (vp, offset), usually
 * after reading the data from disk.  lenp is an in/out parameter;
 * it initially says how many bytes we want, and on return indicates
 * how many we actually got (PAGESIZE <= final value <= initial
 * value).
 */
efs_getpage_io(vp, offset, pl[], plsz, lenp, protp, ...);
{

    if (conditions warrant)
        start readahead;

    if (rw == S_CREATE) { /* Make a new page out of whole cloth */
        pl[.] = page_create(vp, offset, PAGESIZE);
        *lenp = PAGESIZE;
    } else {    /* Get as many contiguous pages as we can */
        efs_FindBlocks(vp, offset, &blkno, &writeLen, &readLen);
```

```
        if (rw != S_WRITE &&
            (blkno is a hole || writeLen != readLen))
            turn off write permission in *protp;
        if (blkno is a hole) {
            assert(rw != S_WRITE);
            pp = page_create(vp, offset, PAGESIZE);
            pagezero(pp, 0, PAGESIZE);
            pl[.] = pp;
        } else {
            io_off = first block boundary <= offset;
            io_len = (doing seq. reads on vp) ? readLen : block size;
            pp = read_kluster(vp, io_off, lenp, from io_off to io_off
                              + io_len);
            bp = pageio_setup(vp, pp, ...);
            epia_Strategy(bp);
            biowait(bp);
            post I/O cleanup;
        }
    }
}

/*
 * Map in vnode in MAXBSIZE (8KB) chunks, and use uiomove to copy the
 * data from the user's buffer.  We will do all the allocation in
 * efs_getpage, which will be called via segmap_getmapflt or as a
 * consequence of page faults during the uiomove; the only exception
 * is in the cases where we are creating a new page without calling
 * getpage, in which case we do the allocation here.
 */
efs_vmwrite(vp, uiop, ...)
{
    /*
     * Update length and change storage class of file if appropriate.
     */
    if (extending file) {
        if (new_length > max. file size allowed) {
            too_big = TRUE;
            new_length = max. file size;
        }
        vnm_Truncate(vp, new_length, ...);
    }

    for each 8K chunk {
        /*
         * We could set create_page slightly more aggressively, i.e.,
         * whenever the write starts in a page beyond the old EOF; in
         * that case we would also have to zero the beginning of the
         * page when not aligned.  Worth the effort?
         */
        create_page = extending the file && on an 8K boundary  ||
                      overwriting a whole chunk;
        create segmap mapping, faulting in pages if !create_page;
        if (create_page) {
            efs_MakeWritableBlocks(vp, off, 8K, 1);
            segmap_pagecreate(...);
```

```
        }
        error = uiomove(...);
        release mapping;
    }

    update mtime, etc.
    if (too_big && no other error occurred) {
        generate SIGXFSZ;
        return EFBIG;
    }
}

/*
 * Same idea as efs_vmwrite, only simpler because no allocation is
 * needed.
 */
efs_vmread(vp, uiop, ...)
{
    while (read not finished && not past EOF) {
        create mapping for 8K chunk;
        uiomove(...);
        release mapping;
     }
}

/*
 * Create dirty, zero-filled pages starting at off.
 * efs_MakeWritableBlocks uses this to make pages corresponding to
 * newly allocated blocks.
 *
 * Assumes that vp is locked so that no new pages are being created
 * concurrently; otherwise, we might overwrite valid data with
 * zeroes.
 *
 * Note: this will cause page faults.
 */
efs_CreateDirtyZeroPages(vp, off, len)
{
     set vd_owner to curthread;
     for each 8K chunk {
        fbzero(vp, off, size, &fbp);
        fbrelse(fbp, S_WRITE);
     }
     set vd_owner to NULL;
}

#elif defined(AFS_AIX_ENV)
/*
 * AIX-specific VM interfaces
 */

/*
 * Routines called by the episode VM handling daemons with a list of
 * bufs. Their job is to do all the work necessary to call
 * epia_Strategy.
```

```
 *
 * General Description
 *
 *      In AIX we have 3 daemons servicing pagefaults:
 *
 *              1. The pageout daemon. This process simply
 *      schedules the I/O to the disk. It asserts that all the pages
 *      that it is asked to write have backing disk space. It only
 *      read-locks the vnode's fileLock
 *
 *              2. The pagein daemon. This process reads data into
 *      the pages from disk. If the page that it read does not have
 *      allocated writable disk blocks, it returns the page write
 *      protected even if it was called with PF_STORE set. It locks
 *      the vnode's vmLock and read-locks its fileLock. It also
 *      increments the vnode's pagefault counter. If the pagein
 *      specified offset+len past EOF and it was called on behalf of
 *      shmat(), it again returns a write-protected page
 *
 *              3. The page-unprotect daemon. This process gets the
 *      pagefaults for the pages that the pagein daemon filled in but
 *      returned write protected. It does disk allocation, length
 *      setting and promotion if appropriate and returns. It write
 *      locks vnode's vd_tlock, vmLock and fileLock.
 *
 *      Notes: The third daemon is only used for mapped files. In the
 *      efs_vmwrite path we do all the length setting, promotion, and
 *      disk allocation, before calling vm_move, and we call vm_move
 *      while holding the vd_tlock because we know that the 3rd
 *      daemon won't be needed for our vm_move to complete.
 *
 *      Since AIX does not use pages but bufs to communicate with us
 *      and bufs don't have an offset field it puts the offset in
 *      logical disk block units in buf->b_blkno. Before we call
 *      epia_Strategy we must overwrite this field with the actual
 *      disk address of the block that we want the transfer to start
 *      at.
 */

/*
 * The VOP-level read/write interface.
 */
efs_vmrdwr(vp, uiop, ...)
{
    get vmmLenLock;
    make VM segment if needed;
    if (reading) {
        /* Move the data between user buffer and VM segment */
        vm_move(..., uiop, ...);
    } else { /* writing */
        if (extending file)
            vnm_Truncate(vp, new_length, ...);
            if (splitting a COW block) {
                push out any memory pages covering this block,
                since otherwise we may copy stale data to the
```

```
                new block.  Since there is synchronization in
                AIX, we must iteratively write-protect and flush
                until there are no more page faults.  On other
                systems, we would avoid the disk I/O entirely
                by temporarily locking the pages.
            }
            for each new page {
               efs_MakeWritableBlocks(vp, off, len, 0);
               create page;
            }
        }

        /* Move the data between user buffer and VM segment */
        vm_move(..., uiop, ...);

        if (file doesn't end on a page boundary)
            write-protect last page to handle extending mmaps;

        update mtime, etc.;

        if (synchronous)
            write out dirty pages;
    }
    drop vmmLenLock;
}

/*
 * The interrupt-time code that queues I/O requests from the VMM
 */
efs_vmstrategy(bp)
{
    while (bp != NULL) {
        if (bp->b_flags include B_PFSTORE or B_PFPROT)
            put bp on the pageunprotect daemon's queue;
        else if (reading)
            put bp on the pagein daemon's queue;
        else
            put bp on the pageout daemon's queue;

        if (we have just started on a new queue)
            issue wakeup to daemon for preceding queue;

        get next bp;
    }
    wake up daemon for final queue;
}

/*
 * The pagein daemon, whose job is to process the read requests
 * queued by efs_vmstrategy.
 */
efs_PageInDaemon()
{
    while (1) {
        get a request from the queue;
```

```
        if (volume is busy && can't do operation)
            save bp on "busy buffer" queue, to be placed back
            on the active queue when the volume is closed;
        } else
            efs_pagein(bp);
    }
}

efs_PageOutDaemon() {
   /* like efs_PageInDaemon */
}

efs_PageUnprotectDaemon() {
   /* like efs_PageInDaemon */
}

/*
 * Read a set of pages into memory.
 */
efs_pagein(struct buf *bp)
{
    flags = 0;
    /*
     * Page-fault on write; this may be because we are writing into
     * a hole on a mapped file or extending a mapped file.
     */
    if (PFSTORE) {
        if (PFEOF && bp would extend beyond EOF)
            return EXCEPT_EOF I/O error;
        if (extending file)
            flags = BP_PROTECT;
    }
    increment page fault count;
    efs_strategy(bp, flags);
}

/*
 * Write out a set of pages.
 */
efs_pageout(struct buf *bp)
{
    efs_strategy(bp, 0);
    if (synchronous write)
        epia_UpdateUndoZero(bp, 1);
}

efs_pageunprotect(bp)
{
    increment page fault count;
    if (writable mappings && extending file)
        efs_Promote(bp, new_length);
    efs_MakeWritableBlocks(vp, off, ..)
}

/*
```

```
 * Do I/O on a chain of buffers, breaking it into pieces that
 * correspond to contiguous regions on disk.  The buffer chain
 * is assumed to be logically contiguous.
 */
efs_strategy(bp, flags)
{
    while (bp != NULL) {
        get (vp, off) from bp;
        if (inline(vp)) {
           epia_Strategy(bp);
           break;
        } else if (reading) {
            efs_FindBlocks(vp, off, ...);
            if (reading from a hole) {
                bzero a block in buf & write-protect;
                adjust buf length and address;
                continue;
            } else if (reading from COW blocks)
                remember to write-protect pages after read;
        } else /* writing */
           efs_FindWritableBlocks(vp, off, ...);

        if (contiguous region > buf length)
            gather as many bufs as will fit;
        update length, addr of last buf;

        epia_Strategy(tbp);

        if ((flags & BP_PROTECT) || read from COW blocks)
            write-protect page;
    }

    do iodone processing on any unconsumed bufs;
}

efs_CreateDirtyZeroPages(vp, off, len)
{
     for each page {
        vm_makep(vm segment for vp, page no. for offset);
        offset += PAGESIZE;
     }
}
#endif /* AFS_AIX_ENV */

/*
 * New interfaces
 */

/*
 * Make sure that writable disk storage exists for the specified
 * range of offsets in vp.   If allocation is required, try to lay
 * out new blocks in minimum number of contiguous regions.
 */
efs_MakeWritableBlocks(vp, off, len, makePages)
{
```

```
    ap = EVTOA(vp);
    write-lock anode allocation info;
    epia_StartTran(ap, &tran);

    while (len != 0) {
        /* What is at this offset? */
        CheckAllocation(ap, off, &blkno, &alen)
        alen = MIN(len, alen);
        if (blkno == HOLE) {
            CreateBlocks(ap, tran, off, &alen);
            /*
             * We don't want to make pages beyond EOF, so if the
             * unused portion of the last block is a page or more in
             * size, we will write zeroes to it here.
             */
            if (next FS block boundary - (off + alen) >= PAGESIZE)
               ZeroBlock(ap, tran, off + alen);
            if (makePages)
               efs_CreateDirtyZeroPages(vp, off, alen);
        } else if (blkno is read-only) {
            if (makePages)
               CopyBlocks(ap, tran, off, &alen);
            else
               CreateBlocks(ap, tran, off, &alen);
        } else if (IsFragmented(ap) && off + len > alen)
            ExtendFragmentedFile(ap, blkno, alen, off + len);
        /* do nothing if already writable */

        len -= alen;
        off += alen;
    }

    unlock anode;
    epia_EndTran(ap, tran);
}

/*
 * Find the number of allocated blocks starting at the specified
 * offset.  wrLenP returns the number of contiguous writable blocks
 * starting at off, and rdLenP returns the number of contiguous
 * readable blocks.
 */
efs_FindBlocks(vp, off, blockP, wrLenP, rdLenP)
{
    int wlen = 0, rlen = 0;
    daddr_t first_blk, ablk;

    ap = EVTOA(vp);
    read-lock anode allocation info;

    CheckAllocation(ap, off, &ablk, &alen);
    first_blk = ablk;
    while (ablk != HOLE && first_blk == ablk + rlen) {
        if (blkno is read-only)
            rlen += alen;
```

```
        else if (wlen == rlen) {
            rlen += alen;
            wlen += alen;
        } else
            rlen += alen;
        off += alen;
        CheckAllocation(ap, off, &ablk, &alen);
    }
    unlock anode;

    *blockP = first_block;
    *wrLenP = wlen;
    *rdLenP = rlen;
}

/*
 * Like efs_FindBlocks, but returns only the number of consecutive
 * writable blocks.
 */
efs_FindWritableBlocks(vp, off, blockP, lenP)
{
    int wlen = 0;
    daddr_t first_blk, ablk;

    ap = EVTOA(vp);
    read-lock anode allocation info;

    CheckAllocation(ap, off, &ablk, &alen);
    first_blk = ablk;
    while (ablk is writable && first_blk == ablk + wlen) {
        wlen += alen;
        off += alen;
        CheckAllocation(ap, off, &ablk, &alen);
    }
    unlock anode;

    *blockP = first_block;
    *wrLenP = wlen;
}

/*
 * Return offset where next allocated block appears in the given
 * range, or -1 if none.
 */
off_t
efs_NextBlock(vp, off, len)
{
    ap = EVTOA(vp);
    read-lock anode allocation info;

    if (len == 0)
        len = distance from off to EOF;

    CheckAllocation(ap, off, &ablk, &alen);
```

```
    for (o = off; ablk == HOLE && o < off + len; o += alen)
        CheckAllocation(ap, off, &ablk, &alen);

    unlock anode;
    return (ablk == HOLE ? -1 : o);
}


/*
 * Like efs_NextBlock, but finds next writable block.
 */
off_t efs_NextWritableBlock(vp, off, len)
{
    /* Do the obvious */
}

/*
 * Convert vnode to whatever storage type is the "best" fit
 * for the specified length.
 */

/*
 * Table of conversion functions
 */
#define NALLOCTYPES     4      /* EMPTY, INLINE, FRAGS, BLOCKS */
static (*promote_func[NALLOCTYPES][NALLOCTYPES])
       (vnode_t *vp, int len) = {
    ...
};

/*
 * Convert storage class of vp if necessary so that it can grow to
 * desired_len.  In this function, we are only concerned with copying
 * whatever data the file already contains, e.g., from fragments to
 * blocks, and updating the anode.  We do not do any invalidate any
 * pages here or do any truncation.  After doing the necessary
 * conversion, we set the length to desired_len.
 *
 * While changing the disk storage for a file, we must prevent
 * any I/O that could result in stale data, either on disk or
 * in core.  On SunOS, this is done by locking the relevant
 * pages; on AIX, we must flush all the data to disk before
 * proceeding.
 *
 * LOCKS -- vnode's file lock should be write-locked on entry.
 */
efs_Promote(vp, desired_len)
{
    dstType = smallest_class_that_contains(desired_len);
    srcType = storage_class(vp);
    if (srcType != dstType)
        (*promote_func[srcType][dstType])(vp, desired_len);
}

/*
```

```
 * Check whether vnode has any holes or COW blocks.  We do this
 * by calculating the number of blocks needed for a completely
 * filled-in file of this length, and comparing that with the
 * number of blocks actually allocated.
 */
efs_HasHoles(vp)
{
    if (in-line(vp))
        return 0;
     else if (fragmented(vp))   /* it may be entirely COW */
        return (fragment is COW?);

    /* compute total # of data & metadata blocks from length */
    data_blks = howmany(anode length, fs block size);
    blks_per_indir = (usable space in indirect block) /
                     sizeof(daddr_t);

    indir_blks = 0;     /* No. of indirect blocks used */
    uncovered = data_blks - (number of direct blocks);

    while (uncovered > 0) {
        indir_blks += howmany(uncovered, blks_per_indir);
        uncovered = howmany(uncovered, blks_per_indir) - 1;
        /* The -1 is for the 1st block, which we leave uncovered */
    }
    return (# of allocated blocks in vp != data_blks + indir_blks);
}

/*
 * Internal interfaces
 */

/*
 * CheckAllocation returns the disk address of the block at the
 * specified offset (or HOLE if there is none), and the length
 * in bytes of contiguous storage starting at the offset.  Blocks
 * are only considered contiguous if they are of the same type
 * (i.e., writable or read-only).  The returned length is really
 * a lower bound on the size of the contiguous region, since the
 * function stops upon reaching the end of the current allocation
 * block.
 *
 * Should be called with the ap allocation info read-locked (at
 * least).
 */
CheckAllocation(ap, off, blkP, alenP)
{
    if (file is fragmented) {
        handle fragmented case separately (blech);
        return;
    }

    lbn = file system block containing off;
    if (lbn is a direct block) {
        startAlloc = &ap->db[lbn];
```

```
        endAlloc = &ap->db[# of direct blocks];
    } else {
        allocBlk = GetAllocationBlock(ap, lbn, &next_lbn);
        if (allocBlk == NULL) {        /* No allocation block */
            *blkP = HOLE;
            *alenP = fsbtob(next_lbn - lbn);
            return;
        }
        startAlloc = address of entry for lbn;
        endAlloc = address of end of allocation block (or EOF);
    }
    first_blk = *startAlloc;
    for (p = &startAlloc[1]; p != endAlloc; p++) {
        if (!same_type(first_blk, *p))
            break;
    }
    *blkP = first_blk;
    *alenP = p - first_blk;
    if (lbn is not a direct block)
        ReleaseAllocationBlock(allocBlk);
}

/*
 * CreateBlocks allocates a set of contiguous disk blocks for the
 * anode at the specified offset.  lenP is an in-out parameter.
 * On entry, it contains the number of bytes of storage desired.
 * If this much contiguous space cannot be found, CreateBlocks may
 * allocate to a smaller amount (but not zero); the caller is
 * responsible for calling CreateBlocks as many times as necessary
 * until all the allocation is performed.
 *
 * This routine handles both blocked and fragmented files.  However,
 * it does not do promotion.
 */
CreateBlocks(ap, tran, off, lenP)
{
   allocate up to *lenP of contiguous disk space;
   update ap & start undo-zero transaction;
   epia_RecordUndoZero(...);
   update *lenP;
}

/*
 * CopyBlocks produces writable copies of a set of read-only disk
 * blocks.  lenP has the same interpretation as for CreateBlocks.
 * The read-only blocks are guaranteed to be contiguous, and the new
 * writable copies must be contiguous also.
 *
 * This routine presents some tricky problems.  Initializing the new
 * copies synchronously here is easy but probably slow.  If we do the
 * write asynchronously, we must make sure there are pages containing
 * the data so that we don't read from the disk until the new blocks
 * are initialized.
 */
CopyBlocks(ap, tran, off, lenP)
```

```
{
#ifdef AFS_SUNOS5_ENV
   ...
   epia_RecordUndoZero(...);
   ...
   epia_Strategy(...);
   ...
#endif
}

/*
 * Return a pointer to the allocation block that covers logical
 * block blkno, or NULL if none exists.  We search through the
 * appropriate indirect block tree until we find the block that
 * we want or a hole.  *nextBlkP is set to the lowest following
 * block for which an allocation block may exist, i.e., the next
 * possible leaf in the tree.  We won't actually verify that the
 * leaf exists in all cases, e.g., if it is the next tree, so
 * this may be an underestimate.
 */
daddr_t *
GetAllocationBlock(ap, blkno, nextBlkP)
{
    calculate root of indirect tree from blkno;
    calculate span, etc.;
    iterate from root to leaf: {
        get allocation block & find entry corresponding to blkno;
        if (entry is a hole) {
            next_entry = first non-hole following entry;
            if (suitable next_entry not found)
               next_entry = first entry in following allocation
                            block;
            *nextBlkP = logical block no. corresponding to
                        next_entry;
            release allocation block;
            return NULL;
        }
        if (!leaf node)
            release allocation block;
    }
    calculate *nextBlkP as above;
    return allocation block;
}

/*
 * Undo-zero processing
 */

/*
 * This structure provides a handle for undo-zero transactions.  In
 * addition to the tranId, it contains a ref. count of allocated but
 * still uninitialized user data blocks, and a pointer so that
 * completed transactions can be linked together on a queue.
 */
struct epia_undoZero {
```

```
    struct undo_zero_tran *next;
    elbb_tranRec_t tranId;
    u_int pending_blocks;
};

typedef struct epia_undoZero epia_undoZero_t;

/*
 * Queue of undo-zero transactions that can be ended.
 */
epia_undoZero_t *epia_completedUndoZeroes;

/*
 * This structure associates a disk block with an undo-zero
 * transaction.  We create a instance of the structure for each
 * allocated, but not yet initialized, user data block and store them
 * in a hash table.  When a write completes, we remove the
 * corresponding blocks from the hash table and decrement the pending
 * block count on the undo-zero transaction.  When the ref. count
 * reaches zero, we either end the undo-zero transaction or queue it
 * for completion by the async daemon.
 */
struct epia_pendingBlock {
    struct epia_pendingBlock *next;
    epi_anode_t anode;
    daddr_t blkno;
    epia_UndoZero_t *tran;
}

typedef struct epia_pendingBlock epia_pendingBlock_t;

/*
 * Head for a hash chain
 */
struct pendingBlock_hashList {
    osi_mutex_t hash_mutex;
    epia_pendingBlock_t *blocks;
};

static struct pendingBlock_hashList
       pendingBlock_HashTab[HASHTABSIZE];

epia_RecordUndoZero(tranId, bp)
{
    tran = new epia_undoZero_t;
    tran->tranId = tranId;
    tran->pending_blocks = 0;

    for each disk block covered by bp {
        tran->pending_blocks++;
        p = new epia_pendingBlock_t;
        p->tran = tran;
        p->blkno = this block;
        p->anode = anode for this block;
        insert p in pendingBlock_HashTab;
```

```
    }
}

/*
 * Update any undo-zero transactions affected by completion of a
 * write on bp.  If the pending_block count on the transaction
 * reaches zero, we either end the transaction or queue it for
 * completion by the async daemon, depending on the value of
 * finishTran.
 *
 * This function is called at interrupt time when an asynchronous
 * write completes, and also from putpage after a sync write.
 */
epia_UpdateUndoZero(bp, finishTran)
{
    for each disk block covered by bp {
        RemovePendingBlock(block, anode, finishTran);
    }
}

/*
 * Remove a specific block from the hash table and update its
 * undo-zero transaction.
 */
RemovePendingBlock(block, anode, finishTran)
{
    p = lookup & remove from hash tab(block, anode);
    if (p != NULL) {
        if (--p->tran->pending_blocks == 0) {
            if (finishTran)
               EndUndoZero(...);
            else {
               add p->tran to completedUndoZeroes list;
               wake daemon if list was empty;
            }
        }
        discard p;     /* free list or osi_Free */
    }
}

/*
 * We use this interface to discard pending block records when
 * we destroy the corresponding blocks by truncation or deletion.
 * In this case, all we get is an offset and length, so we have
 * to find the disk blocks ourselves.
 */
epia_PurgeUndoZero(vp, off, len)
{
    int wlen = 0;
    for each block offset in range {
        if (wlen == 0)
            efs_FindWritableBlocks(vp, offset, &blkno, &wlen);
        RemovePendingBlock(block, anode, 1);
        blkno++;
        wlen -= block size;
```

```
    }
}

EndUndoZero(ap, tran)
{
     epia_EndTran(ap, tran);
     lock completedUndoZeroes list;
     EndCompletedUndoZeroes();  /* clean out the list */
}

EndCompletedUndoZeroes()
{
    foreach record on epia_CompletedUndoZeroes {
        remove record from list;
        epia_EndTran(anode, Tran);
        free record;
    }
}

epia_AsyncDaemon()
{
    while (1) {
        lock epia_CompletedUndoZeroes list;
        while (list is empty) {
            drop lock & sleep;
            reaquire lock;
        }
        EndCompletedUndoZeroes();
        if (transaction system is blocked by full log) {
            set waitingForUndoZero to false;
            wake up transaction system;
        }
    }
}

epia_ForceUndoZero(tranId)
{
}

epia_WaitForUndoZero()
{
    get lock on waitingForUndoZero c.v.;
    if (no unfinished undo-zero transactions) {
        drop lock;
        return failure;
    }
    set waitingForUndoZero to true;
    while (waitingForUndoZero) {
        drop lock & sleep;
        reacquire lock;
    }
    drop lock;
    return success;
}
```

AUTHOR'S ADDRESS

Blake Lewis
Transarc Corporation
The Gulf Tower
707 Grant Street
Pittsburgh, PA 15219

email: blake@transarc.com
Telephone: +1-412-338-6924