AdvFS 1024 ACLs

Design Specification

Version 2.0

DB
MD

Table of Contents

Preface

Version 1.0 of the Advfs 1024 ACLs Design Specification is being made available for comments and review by all interested parties.

If you have any questions or comments regarding this document, please contact:

Author Name	Mailstop	Email Address
DB		
MD		

Approver Name	Approver Signature	Date

1 Introduction

1.1 Abstract

This design document describes a design for Access Control Lists (ACLs) for AdvFS on HPUX which will support 1024 ACL entries per file. ACLs are a security mechanism for defining extended permissions on a file. This design describes an implementation for SystemV ACLs.

The goal of this design is to describe an implementation of ACLs that is fairly straight-forward, low risk and not reliant on an underlying Property List design.

1.2 Product Identification

Project Name	**Project Mnemonic**	**Target Release Date**
AdvFS 1024 ACL Design	AdvFS 1024 ACLs	

1.3 Intended Audience

This document is intended for review by those interested in AdvFS on HPUX and its interaction with other kernel subsystems. HPUX security teams may be interested in this document as it relates to file system security mechanisms.

1.4 Related Documentation

The following list of references was used in the preparation of this Design Specification. The reader is urged to consult them for more information.

Item	Document	URL
1		
2		
3		

1.5 Purpose of Document

This design will describe how AdvFS will implement large ACLs on HPUX. The intent is to provide sufficient design detail for the design to be implemented by AdvFS kernel engineers.

1.6 Acknowledgments & Contacts

The authors would like to gratefully acknowledge the contributions of the following people:

JA, DA, BH, TM, and BN.

1.7 Terms and Definitions

Term	Definition
ACL	Access Control List. A mechanism for storing privilege information for a file. AdvFS will implement System V ACLs.
ACL entry	A single element of the ACL. A single ACL entry defines the permissions for one group or user.
Tuple	A VFS term for an ACL entry. VFS deals with ACL tuples.
Base ACL	The basic UNIX permissions for owner, group and other.
Optional ACL Entry/ Optional tuple	ACL entries that are not part of the base ACL.

2 Design Overview

2.1 Design Approach

On Tru64, AdvFS ACLs were layered on top of Property Lists and were therefore constrained to a maximum of 62 ACL entries per file. The constraint was a result of the underlying Property List limitations. This design intends on removing the dependency of ACL code on underlying Property Lists. This decision has been made since support for Property Lists will be deferred.

The design uses a mechanism similar to the underlying Property List implementation on Tru64 whereby ACLs are stored in mcell records that are chained together. The ACL for a single file is composed by concatenating a set of mcell records stored in the mcell chain for a given file.

Because the mcell chain will be bounded in size and because it has a single purpose, much of the complexity of the underlying Property List infrastructure on Tru64 will not be required.

To minimize the number of IOs required for large numbers of ACL entries, the ACL for a file will be cached in a pointer off the bfAccess structure. For a large (1024) ACL, this cache may be up to 12K; however, up to 52 IOs may be required to read the entire ACL chain, so using the extra memory for caching is justified. Typical usage scenarios indicate that the cache will be between 36 and 120 bytes.

2.2 Overview of Operation

The ACL on a single file will be manipulated through the user commands setacl and getacl. These commands will interface with the kernel through the VFS layer routine `acl` in vfs_scalls.c. The VFS layer will resolve to `VOP_SETACL` and `VOP_GETACL` for the file being operated on. The AdvFS implementation of these vnode operations will be `advfs_setacl` and `advfs_getacl`. Much of the preprocessing of the ACL being operated on is done by VFS or in user space and is therefore not discussed in this design.

When `advfs_setacl` is called, a list of ACL entries is passed in. This ACL will replace the ACL already associated with the vnode on which `advfs_setacl` was called. The changing of the ACL will be done atomically. Prior to setting the new ACL, enough mcells will be initialized and chained together to store the entire ACL set. If an error occurs, the previous ACL will be restored by failing the entire transaction to set the new ACL.

`advfs_getacl` will generally rely on the ACL cache off of the bfap to read the ACL. Locking is necessary to insure that the cache is not replaced or modified by a call to `advfs_setacl` while the `advfs_getacl` is copying the cache to a local buffer. No locking exists to guarantee the ordering of a racing setacl and getacl user call.

2.3 Major Data Structures

2.3.1 In Memory Structures

2.3.1.1 bfAccess

Lock alignment concerns on PA will be considered at implementation time.

```
typedef struct bfAccess {
.
.
.
    rwlock_t           bfa_acl_lock;  /* Protects the file's ACL. Initialized in
                                        * advfs_access_constructor */
    /*
     * ACL cache pointer
     */
    advfs_acl_t        *bfa_acl_cache;        /* A cached copy of the ACL for this file */
    size_t             bfa_bfa_acl_cache_len; /* length of bfa_acl_cache */
```

```
    uint32_t largest_pl_num;    /* guarded by mcellList_lk */

} bfAccessT;
```

2.3.1.2 advfs_acl

```
/* AdvFS version of the aclv structure defined in aclv.h */
struct{
       int32_t aa_a_type;     /* acl entry type */
       int32_t aa_a_id;       /* user or group ID */
       int16_t aa_a_perm;     /* Permissions */
} advfs_acl;

typedef struct advfs_acl advfs_acl_t;
```

2.3.2 On Disk Structures (also used in memory) (64 bit aligned)

2.3.2.1 BSR_ACL

This record will replace BSR_PROPLIST_HEAD (19) and will represent a bsr_acl_rec in the BMT.

2.3.2.2 struct bsr_acl_rec

This structure will represent the ACL on disk. The structure will consume the majority of an mcell. The bar_acl_link_seg represents a sub-link segment value to help recovery. This is necessary because all mcells allocated for use by ACLs will have the same link segment value in the mcell header.

```
struct {
       uint32_t              bar_acl_cnt;          /* Number of acls in this record */
       uint16_t              bar_acl_link_seg      /* Place in ACL mcell list */
       uint16_t              bar_acl_type          /* SYSV_ACLS */
       advfs_acl_t           bar_acls[ADVFS_ACLS_PER_MCELL];
                                                  /* ACL array. */
} bsr_acl_rec

typedef struct bsr_acl_rec bsr_acl_rec_t
```

2.3.2.3 Exisiting PL ODS structures and definitions to be removed

BSR_PROPLIST_HEAD, BSR_PROPLIST_HEAD_SIZE, BSR_PROPLIST_HEAD_SIZE_V3, BSR_PL_LARGE, BSR_PL_DELETED, BSR_PL_PAGE, BSR_PL_RESERVED, struct bsPropListHead, struct bsPropListHead_v3, NUM_SEG_SIZE, BSR_PROPLIST_DATA, BSR_PROPLIST_DATA_SIZE, BSR_PL_MAX_SMALL, BSR_PROPLIST_PAGE_SIZE, BSR_PL_MAX_LARGE, struct bsPropListPage, and struct bsPropListPage_v3 will be removed as they are no longer needed.

2.3.3 Log Structures

2.3.3.1 struct acl_update_undo_hdr (64-bit aligned)

```
struct {
       uint64_t              auuh_undo_count;      /* Number of undo records */
       bfSetId               auuh_bf_set_id;       /* bf set id of file being undone */
       bfTagT                auuh_bf_tag;          /* Tag of file being undone */
} acl_update_undo_hdr

typedef struct acl_update_undo_hdr acl_update_undo_hdr_t
```

2.3.3.2 struct acl_update_undo (64-bit aligned)

```
struct {
```

```
        mcellIdT        auu_mcell_id
        char            auu_previous_record[ sizeof( bsr_acl_rec_t ) ];
                                               /* Byte array of changed data */
} acl_update_undo

typedef struct acl_update_undo acl_update_undo_t;
```

2.3.4 Macros

2.3.4.1 ADVFS_SETBASEMODE

```
/* Based on setbasemode from aclv.h */
#define ADVFS_SETBASEMODE( oldmode, mode, ugo ) \
               ( ( oldmode & ~( 7 << ugo ) ) | ( mode << ugo ) )
```

2.3.4.2 ADVFS_GETBASEMODE

```
/* Based on getbasemode from aclv.h */
#define ADVFS_GETBASEMODE( basemode, ugo ) \
               ( ( basemode >> ugo ) & 7 )
```

2.3.5 Constants

2.3.5.1 ADVFS_ACL_TYPE

```
/* SYSV_ACLS is the type of the ACLs AdvFS deals with */
#define ADVFS_ACL_TYPE        SYSV_ACLS
```

2.3.5.2 ADVFS_MAX_ACL_ENTRIES

```
/* The maximum number of ACL entries per file */
#define ADVFS_MAX_ACL_ENTRIES        1024
```

2.3.5.3 ADVFS_ACLS_PER_MCELL

```
/* Largest number of ACL entries that can be stored in a BMT mcell record.  *
 * ACL size / ADVFS_ACLS_PER_MCELL = number of mcells required to store the ACL.
 * This is a little over 20 (about 23) as of the mid October 2003. */
#define ADVFS_ACLS_PER_MCELL  (BSC_R_SZ - 2*sizeof(bsMRT))/sizeof(bsr_acl_rec_t)
```

This value is based on the maximum size of a BMT record and the size of the advfs_acl structure (12 bytes).

2.3.5.4 ADVFS_NUM_BASE_ENTRIES

```
/* number of base ACL entries, user, group, other and class */
#define ADVFS_NUM_BASE_ENTRIES       4
```

2.3.5.5 General ACL #defines from acl.h and aclv.h

```
/* The following defines are based on acl.h defines */
#define ADVFS_ACL_UNUSED      -35     /* ACLUNUSED */
#define ADVFS_ACL_USER        6       /* ACL_USER */
#define ADVFS_ACL_GROUP       3       /* ACL_GROUP */
#define ADVFS_ACL_OTHER       0       /* ACL_OTHER */

/* The following defines are based on aclv.h defines */
#define ADVFS_USER_OBJ        0x01                          /* owner of the object */
#define ADVFS_USER            0x02                          /* optional users */
#define ADVFS_GROUP_OBJ       0x04                          /* group of the object */
#define ADVFS_GROUP           0x08                          /* optional groups */
```

```
#define ADVFS_CLASS_OBJ       0x10                               /* file group class entry */
#define ADVFS_OTHER_OBJ       0x20                               /* other entry */
#define ADVFS_ACL_DEFAULT     0x10000                            /* default entry */
#define ADVFS_DEF_USER_OBJ    (ACL_DEFAULT | USER_OBJ)           /* default object owner */
#define ADVFS_DEF_USER        (ACL_DEFAULT | USER)               /* default optional users */
#define ADVFS_DEF_GROUP_OBJ   (ACL_DEFAULT | GROUP_OBJ)          /* default owning group */
#define ADVFS_DEF_GROUP       (ACL_DEFAULT | GROUP)              /* default optional groups */
#define ADVFS_DEF_CLASS_OBJ   (ACL_DEFAULT | CLASS_OBJ)          /* default class entry */
#define ADVFS_DEF_OTHER_OBJ   (ACL_DEFAULT | OTHER_OBJ)          /* default other entry */

#define ADVFS_DONTCARE        0x40000                            /* for pattern match */
#define ADVFS_IDDONTCARE      0x80000                            /* for pattern match */
```

2.3.6 Errors

2.4 Exception Conditions

The following general classes of exception conditions have been considered for this design.

- Invalid input parameters.
  - o Parameter out of range.
  - o Referenced data invalid.
- Resource depletion.
  - o Memory resources (free memory, physical memory, memory objects (buf structures, etc.)).
  - o Hardware resources.
- Race conditions.
  - o Sleeping for an event just as it occurs (or missing it completely).
  - o Locking protocol deadlocks.
  - o Consider cluster-wide races
- Insufficient privilege.
  - o User privilege level.
  - o Memory access rights.
  - o Privilege instructions.
- Hardware errors.
  - o Reported errors.
  - o Non-responding hardware.
- Power failure.
  - o System power failure.
  - o Device power failure.
- Multiprocessor.
  - o What the other processor is doing.
- Cluster exceptions
  - o What are other cluster members doing
  - o Failures during failover (i.e. multiple failures)

Specific cases of these general exception conditions that are applicable to this design are discussed in the following subsections.

3 Detailed Design

3.1 Major Modules

3.1.1 ACLs interface

3.1.1.1 VOP_SETACL (advfs_setacl)

Interface

```
int
advfs_setacl(  struct vnode*        vp,           /* File to set ACL on */
               int32_t              num_tuples,   /* Number of ACL entries requested */
               struct acl*          tupleset,     /* Buffer with new ACL entries */
               int32_t              acl_type      /* ACL type, should be SYSV_ACLS */
               )
```

Description

This routine is the AdvFS specific call out for `VOP_SETACL`. It takes the vnode pointer to the file whose ACL is to be set, the number of ACL entries requested to change, a buffer containing the new ACL to be set (tupleset), and the type of ACL, which should be `SYSV_ACLS`. Each ACL entry is referred to as a "tuple". Base tuples are those ACL tuples which reflect the base permissions (user, group, other and class). The base tuple "class" will be identical to "other" when there are no optional tuples. If there are optional tuples, the class tuple will specify the maximum permissions that can be granted by any of the optional user or group ACL entries. Optional tuples are any ACL tuples that are not base tuples.

The user command, setacl, and the VFS layer will preprocess the ACL before calling `VOP_SETACL`. Whenever an option to delete or add an entry is specified, the user command setacl will first call getacl to get the current ACL. It then rebuilds the ACL by either adding or removing those entries specified. It then calls setacl to insert the new ACL. The user command will also examine the user specified ACL for duplicate entries. If a duplicate is found, it will return an error to the user. This eliminates quite a bit of work for `advfs_setacl`. It needs only to delete the entire list or reinsert the new list as well as update mode bits and the BMT's stored base permissions.

Before any modifications to the current ACL is made, this routine will make basic checks for valid ACL type, number of ACL entries, and domain panics, as well as checking to make sure the proper owner or super user is attempting to modify the ACL of the file. The file system must not be read-only.

The ACL that is set will be stored in a contiguous chain of mcells in the mcell chain of the file being modified. If some mcells are already being used for an ACL on this file, the number of mcells will be increased or decreased as required. When adding additional mcells to the set of mcells reserved for ACLs in a file's mcell chain, the new mcells will be inserted after the already allocated mcells. This will keep the ACL mcells contiguous in the file's mcell chain. In an mcell chain without ACLs, each mcell has a unique (monotonically increasing) `linkSeg` identifier. In the case of the ACLs, all mcells allocated for ACLs will have an equal linkSeg value and will have a minor link segment value in the `bsr_acl_rec` structure (`bar_link_seg`). The linkSeg is only needed for recovery and can be used to guarantee correct ACL ordering when reconstructing a file on a corrupted system. Preallocating all the mcells up front, rather than as they are required, borrows the model of typical write where storage is allocated before the write is done. Additionally, allocating all necessary mcells before starting any modifications or writes makes the transactions more efficient.

A special case for this routine is when the number of tuples requested to change is less than zero. This indicates that the optional ACL entries should be removed. Since base tuples are stored and maintained in the `fs_stat` record, the ACL mcells and the associated records will be removed.

The next order of operation is to set up a new AdvFS specific ACL buffer that will be used to stored the ACL on disk. The `advfs_acl` data structure will be the on-disk structure used to store ACL entries. One by one, each tuple will be examined and inserted into the buffer. There is no need to sort the tuples since this is done in the user command setacl.

As each tuple is examined, if it is a base tuple, before it is inserted in the new ACL buffer, the mode is noted for later to change the base mode bits. If it is an optional tuple, it is simply inserted into the new buffer and `num_opt_tuples` (number of optional tuples) is incremented.

All changes to the ACL will be made under the `FTA_ACL_UPDATE` transaction agent. The allocation or deallocation of mcells for the ACL will happen under subtransactions. When modifying the ACL records in mcells that were not freshly allocated, a transaction will be used that will log a complete undo for the modification. The undo will consist of a before image of the previous ACL record in a given mcell. For mcells that are freshly allocated, an undo is not required as the mcell will be removed from the mcell chain when the transaction fails.

If there were no optional tuples specified, then all ACL mcells will be removed from the mcell chain. There is no need for the ACL mcells if only the base modes are stored.

`The bfa_acl_cache` pointer in the bfap will be freed and the `advfs_acl` buffer will be copied into a new buffer for the `bfa_acl_cache`. The routine is concluded by updating the mode bits and the BMT to indicate the new base tuple modes.

This is where the transaction is finished and changes can be committed.

This routine will use two primary locks to synchronize with other threads. The first lock, the bfa_acl_lock is a new lock that is being introduced to protect reads and writes to the subsection of the mcell chain that contains ACL data. The `bfa_acl_lock` will be held across the majority of this function. The second lock is the `mcellList_lk`. The `mcellList_lk` will be held for write when adding or removing mcells to the ACL chain. If any new mcells are added to the chain, the `mcellList_lk` must be held in write mode until those new mcells are either initialized or the transaction is `ftx_fail`'ed or `ftx_done`'d. In case of new mcells being added to the ACL chain, the `mcellList_lk` must be held in write to prevent other threads from assigning a record in one of the newly allocated mcells. The `mcellList_lk` will prevent a second thread from allocating a record in the new mcell for use by a record other than an ACL. If no new mcells are added, the `mcellList_lk` can be dropped since any racing threads will either need the `bfa_acl_lock` (to look at or modify the ACL portion of the mcell chain) or will see the ACL mcells as full and not try to assign records.

The `mcellList_lk` and `bfa_acl_lock` must both be held when making a call to `ftx_fail`. This is because failing a transaction may cause the undo routines to be executed and those undo routines expect these locks to be held. In recovery, the undo routines will be called in a single threaded context, so the locks will not need to be held.

The routine returns EOK when successful.

In a worst case scenario, this routine may use a number of transactions equal to:

$$1 + 1024 / ADVFS\_ACLS\_PER\_MCELL + \left\lceil \frac{1024 / ADVFS\_ACLS\_PER\_MCELL}{FTX\_MX\_PINP} \right\rceil +$$
$$1 + BMT\_EXTEND\_TRANSACTIONS * n$$
$$= 62 + BMT\_EXTEND\_TRANSACTIONS * n$$
$$n = number\_of\_extends\_required\_for\_52\_mcells$$

Assuming that no calls to `bmt_extend` are required to get all the necessary mcells for the transaction, this is typically about a maximum of 62 transactions. In the case of a replacement of a 1024 ACL with another 1024 ACL, the total transaction tree will require on the order of 32k in the log to log all the undo and redo records for the modification.

Execution Flow

```
/* Verify ACL type requested */
if( acl_type != ADVFS_ACL_TYPE ) {
        u.u_error = ENOSYS;
        return ENOSYS;
}

/* Convert vnode to bfAccess */
bfap = VTOA( vp );

/* filter out reserved files */
if( BS_BFTAG_RSVD( bfap->tag ) ) {
        u.u_error = EINVAL;
        return EINVAL;
}

/* If the file system is read only, advfs_setacl cannot be performed */
if( bfap->bfSetp->vfsp->vfs_flags & VFS_RDONLY ) {
        u.u_error = EROFS;
        return EROFS;
}

/* Check for domain panic */
if( bfap->dmnP->dmn_panic ) {
        u.u_error = EIO;
        return EIO;
}

/*
 * Check that either owner or su is attempting to make these changes
 *
 * If not, return u.u_error
 */

/* Verify the ACL is ordered properly and it contains all the base entries */
sts = advfs_verify_acl( tupleset, num_tuples );
if( sts != EOK ) {
        u.u_error = EINVAL;
        return EINVAL;
}

/*
 * START TRANSACTION
 *
 * Guarantee that all of the next steps either happen or do not happen:
 * 1. New ACLs are set or removed in the BMT.
 * 2. BMT is updated with new base permissions
 * 3. Base mode bits are set
 * 4. ACL cache is cleared out and reset with new ACL
 */

sts = FTX_START_N( FTA_ACL_UPDATE, &ftx_acl_update, FtxNilFtxH, bfap->dmnP );
if sts != EOK
        return sts
/* If num_tuples is zero, delete the ACL from the BMT */
if( num_tuples == 0 ) {
        mcells_required = 0
        new_mcells = 0
        lock_write( &bfap->bfa_acl_lock )
        lock_write( &bfap->mcellList_lk )
        sts = advfs_acl_resize_mcells( ftx_acl_update,
                                       bfap,
                                       mcells_required,
                                       &new_mcells,
                                       &acl_mcell_start)
        if (sts != EOK)
                free resources
                fail transactions
                unlock( bfa_acl_lock )
                unlock( mcellList_lk )
```

```
        return sts
    ASSERT( new_mcells == 0)
    unlock( &bfap->mcellList_lk )

    /* Remove the ACL cache if it exists */
    if( bfap->bfa_acl_cache != NULL )
        free( bfap->bfa_acl_cache );
        bfap->bfa_acl_cache = NULL
        bfap->bfa_acl_cache_length = 0

    ftx_done( ftx_acl_update );

    unlock( &bfap->bfa_acl_lock )
    return EOK
}

/* Initialize the new acl structure, malloc space for acl based on num_tuples size */
advfs_acl = ( advfs_acl_t *)malloc( sizeof( advfs_acl_t ) * num_tuples );

/* Make a temporary copy of the tuple set */
for( i=0, tp=tupleset; i<num_tuples; i++, tp++ ) {

    /* Are these valid tuples? */
    if( ( tp->a_id >= MAXUID || tp->a_id < 0 ) ||
      ( tp->a_perm > 7 ) ) {
        u.u_error = EINVAL;
        free( advfs_acl );
        fail transaction
        return EINVAL;
    }

    /*
     * BASE TUPLES:
     * In the following section, we have found one of the base tuples
     */
    chkbase = 0;
    if( tp->a_type == ADVFS_USER_OBJ ) {            /* Case: user base tuple */

        /* Error if it's already been set */
        if( chkbase & (1<<ADVFS_ACL_USER ) ) {
            u.u_error = EINVAL;
            return EINVAL;
        }
        chkbase |= 1<<ADVFS_ACL_USER;
        /* Set other base mode bits */
        imode = ADVFS_SETBASEMODE( imode, tp->a_perm, ADVFS_ACL_USER );
        advfs_acl[i].aa_a_type = tp->a_type;
        advfs_acl[i].aa_a_id = tp->a_id;
        advfs_acl[i].aa_a_perm = tp->a_perm;
        continue;

    } else if( tp->a_type == ADVFS_GROUP_OBJ ) {    /* Case: group base tuple */

        if( chkbase & ( 1<<ADVFS_ACL_GROUP ) ) {
            u.u_error = EINVAL;
            return EINVAL;
        }
        chkbase |= 1<<ADVFS_ACL_GROUP;
        /* Set the group base mode bits */
        imode = ADVFS_SETBASEMODE( imode, tp->a_perm, ADVFS_ACL_GROUP );
        advfs_acl[i].aa_a_type = tp->a_type;
        advfs_acl[i].aa_a_id = tp->a_id;
        advfs_acl[i].aa_a_perm = tp->a_perm;
        continue;

    } else if( tp->a_type == ADVFS_OTHER_OBJ ) {    /* Case: other base tuple */

        if( chkbase & ( 1<<ADVFS_ACL_OTHER ) ) {
    Consulted with Nils on various issues                u.u_error = EINVAL;
            return EINVAL;
        }
```

```
                chkbase |= 1<<ADVFS_ACL_OTHER;
                /* Set the user base mode bits */
                imode = ADVFS_SETBASEMODE( imode, tp->a_perm, ADVFS_ACL_OTHER );
                advfs_acl[i].aa_a_type = tp->a_type;
                advfs_acl[i].aa_a_id = tp->a_id;
                advfs_acl[i].aa_a_perm = tp->a_perm;
                continue;

        } else if( tp->a_type == ADVFS_CLASS_OBJ ) {        /* Case: class base tuple */

                advfs_acl[i].aa_a_type = tp->a_type;
                advfs_acl[i].aa_a_id = tp->a_id;
                advfs_acl[i].aa_a_perm = tp->a_perm;
        }

        /*
         * OPTIONAL TUPLE:            Case: (u,g)
         * In the following section we have found an optional tuple
         */
        advfs_acl[i].aa_a_type = tp->a_type;
        advfs_acl[i].aa_a_id = tp->a_id;
        advfs_acl[i].aa_a_perm = tp->a_perm;
        num_opt_tuples++;
}


/* Take the bfa_acl_lock to protect the modification of the ACL. This lock is not
 * acquired as part of the transaction, but must be dropped after the ftx_done or
 * ftx_fail.  When running recovery, the lock doesn't need to be held since recovery
 * is single threaded. In the non-recovery case, we will hold the lock until ftx_fail
 *  returns.  */
lock_write( &bfap->bfa_acl_lock )

acls_left_to_write = num_tuples
cur_acl_array_ptr = advfs_acl

mcellList_lk_held = FALSE

/*
 * There is no need to keep any ACL mcells if no optional tuples were specified
 * We'll change the mode bits and the BMT later for the base tuples
 */
if( num_opt_tuples == 0 ) {
        mcells_required = 0
        new_mcells = 0
        lock_write( &bfap->mcellList_lk )
        mcellList_lk_held = TRUE
        sts = advfs_acl_resize_mcells( ftx_acl_update,
                                        bfap,
                                        mcells_required,
                                        &new_mcells,
                                        &acl_mcell_start)
        if (sts != EOK)
                free resources
                fail transactions
                unlock( bfa_acl_lock )
                unlock( mcellList_lk )
                return sts


        ASSERT( new_mcells == 0 )
        unlock( mcellList_lk )
        mcellList_lk_held = FALSE

        /* Remove the ACL cache if it exists */
        if( bfap->bfa_acl_cache != NULL )
                free( bfap->bfa_acl_cache );
                bfap->bfa_acl_cache = NULL
                bfap->bfa_acl_cache_length = 0
```

```
} else {
        mcells_required = (num_tuples - 1) / ADVFS_ACLS_PER_MCELL + 1
        new_mcells = 0
        lock_write( &bfap->mcellList_lk )
        mcellList_lk = TRUE
        sts = advfs_acl_resize_mcells( ftx_acl_update,
                                        bfap,
                                        mcells_required,
                                        &new_mcells,
                                        &acl_mcell_start )
        if (sts != EOK)
                free resources
                fail transactions
                unlock( bfa_acl_lock )
                unlock( mcellList_lk )
                return sts


        /*
         * If there are any new mcells for the ACL, then we must hold the lock for write.
         * If we don't hold the lock for write, another thread (wanting a mcell for
         * something other than an ACL) might sneak in an assign a
         * record in one of the new mcells we just allocated but haven't put a record in.
         */
        if (new_mcells == 0)
                unlock( mcellList_lk )
                mcellList_lk_held = FALSE

        acl_link_seg = 0
        sts = FTX_START_N( ACL_UPDATE_WITH_UNDO,
                           cur_ftx, acl_update_ftx )
        if sts != EOK
                ftx_fail(acl_update_ftx)
                free resources
                return sts

        mcells_requiring_undo = mcells_required - new_mcells

        undo_rec_ptr = malloc( sizeof( acl_update_undo_hdr ) +
                        (FTX_MX_PINP * FTX_MX_PINR * sizeof( acl_update_undo ) )
        cur_undo_rec_ptr = undo_rec_ptr + sizeof( acl_update_undo_hdr )
        cur_undo_cnt = 0

        cur_mcell_id = acl_mcell_start

        /*
         * All the mcells that are being overwritten (weren't just allocated)
         * need to be completely logged so the modifications can be undone.
         */
        for (cur_mcell = 0; cur_mcell < mcells_requiring_undo; cur_mcell++)
                bmt_bfap = bfap->dmn->vdpTbl[cur_mcell_id.volume].bmtp
                sts = rbf_pinpg( &pghdl,
                                 page_ptr,
                                 bmt_bfap,
                                 cur_mcell_id.page,
                                 BS_NIL,
                                 cur_ftx,
                                 MF_VERIFY)
                if (sts == E_MAX_PINS_EXCEEDED)
                        ASSERT( cur_undo_cnt >= FTX_MX_PINP )
                        ((acl_update_undo_hdr)undo_rec_ptr)->auuh_undo_count = cur_undo_cnt
                        ((acl_update_undo_hdr)undo_rec_ptr)->auuh_bf_set_id = bfSetId
                        ((acl_update_undo_hdr)undo_rec_ptr)->auuh_bf_tag = bfap->tag
                        ftx_done_u( cur_ftx, undo_rec_ptr, sizeof( undo_records ) )
                        sts = ftx_start_n( cur_ftx, acl_update_ftx )
                        if sts != EOK
                                ftx_fail( acl_update_ftx )
                                free resources
                                return sts
                        cur_undo_rec_ptr = undo_rec_ptr + sizeof( acl_update_undo_hdr )
                        cur_undo_rec_cnt = 0
```

```
        sts = rbf_pinpg( &pghdl,
                page_ptr,
                bmt_bfap,
                cur_mcell_id.page,
                BS_NIL,
                cur_ftx,
                MF_VERIFY)
        if (sts != EOK)
                /* We may need to undo mcell list changes, so
                 * we need to hold the mcellList_lk for the fail
                 * case. */
                if (mcellList_lk_held == FALSE)
                        lock_write( &bfap->mcellList_lk )
                ftx_fail( cur_ftx )
                ftx_fail( acl_update_ftx )
                unlock( &bfap->mcellList_lk )
                unlock( &bfap->bfa_acl_lock )
                free resources
                return sts;


if (!rbf_can_pin_record( pghdl ) )
        ASSERT( cur_undo_cnt >= FTX_MX_PINR )
        ((acl_update_undo_hdr)undo_rec_ptr)->auuh_undo_count = cur_undo_cnt
        ((acl_update_undo_hdr)undo_rec_ptr)->auuh_bf_set_id = bfSetId
        ((acl_update_undo_hdr)undo_rec_ptr)->auuh_bf_tag = bfap->tag

        ftx_done_u( cur_ftx, undo_rec_ptr, sizeof( undo_records ) )
        sts = ftx_start_n( cur_ftx, acl_update_ftx )
        if sts != EOK
                ftx_fail( acl_update_ftx )
                free resources
                return sts
        cur_undo_rec_ptr = undo_rec_ptr + sizeof( acl_update_undo_hdr )
        cur_undo_rec_cnt = 0

        sts = rbf_pinpg( &pghdl,
                page_ptr,
                bmt_bfap,
                cur_mcell_id.page,
                BS_NIL,
                cur_ftx,
                MF_VERIFY)

        if (sts != EOK)
                /* We may need to undo mcell list changes, so
                 * we need to hold the mcellList_lk for the fail
                 * case. */
                if (mcellList_lk_held == FALSE)
                        lock_write( &bfap->mcellList_lk )

                ftx_fail( cur_ftx )
                ftx_fail( acl_update_ftx )
                unlock( &bfap->mcellList_lk )
                unlock( &bfap->bfa_acl_lock )
                free resources
                return sts;

/* We can go ahead and pin the record and log it for undo and
 * modify it. */
mcell_ptr = &((bsMPgT)page_ptr)->bsMCA[cur_mcell_id.cell]
record_ptr = mcell_ptr->bsMR0 + sizeof( struct bsMR )
rbf_pin_record( pghdl,
                record_ptr ),
                sizeof( bsr_acl_rec_t ) )

ASSERT( record type == BSR_ACL_REC )
/*
 * Setup the undo record for the changes we are about to make to
 * the date in this record.  We will copy out the entire record
```

```
         * prior to replacing the whole thing with a new record.
         */
        cur_undo_rec_ptr->auu_mcell_id = cur
        bcopy( record_ptr,
               &cur_undo_rec_ptr->auu_previous_record,
               sizeof( bsr_acl_rec_t )

        cur_undo_cnt++
        cur_undo_rec_ptr = undo_rec_ptr + sizeof( acl_update_undo )

        /*
         * Get the current acl_link_seg so we can make sure to set the new mcells
         * correctly.
         */
        acl_link_seg = record_ptr->bar_acl_link_seg
        ASSERT( record_ptr->bar_acl_type )
        acls_to_write = MIN( ADVFS_ACLS_PER_MCELL, acls_left_to_write )
        record_ptr->bar_acl_cnt = acls_to_write

        /* Set the number of acls in this record */
        ((bsr_acl_rec_t*)record_ptr)->bar_acl_cnt = acls_to_write;
        bcopy( cur_acl_array_ptr,
               &((bsr_acl_rec_t*)record_ptr)->bar_acls,

               acls_to_write * sizeof( advfs_acl ) )
        acls_left_to_write -= acls_to_write
        cur_acl_array_ptr += acls_to_write * sizeof( advfs_acl )

        cur_mcell_id = mcell_ptr->nextMcellId

/*
 * We have now completed all the work for the mcells that needed to be logged.
 * cur_mcell_id is pointing to the next mcell in the acl mcell chain, but
 * now transactions are a little easier since we don't need to log undos
 * (which really chew up space in the log ☹ )  We must finish the transaction
 * from above, then we will start doing the new mcells in a new transaction.
 */
ftx_done_u( cur_ftx, undo_rec_ptr, sizeof( undo_records ))


/* Before we start any more transactions, make sure there is something to do.
 * Note that the acl lock is still held at this point and no one will change
 * our contiguous range of acl mcells. */
if (acls_left_to_write)
        ASSERT( new_mcells != 0 )
        sts = FTX_START_N( ACL_UPDATE_WITHOUT_UNDO,
                           cur_ftx, acl_update_ftx )
        if sts != EOK
               ftx_fail( acl_updat_ftx )
               free resources
               return sts

        mcells_remaining = new_mcells
        /*cur_mcell_id is set to the current mcell already by the previous loop */

        /* Initialize all the remaining mcells to the new acl set. */
        for ( cur_mcell = 0; cur_mcell < mcells_remaining; cur_mcell++)
               ASSERT( cur_mcell_id != NilMcellId )
               bmt_bfap = bfap->dmnP->vdpTbl[cur_mcell_id.volume].bmtp

               sts = rbf_pinpg( &pghdl,
                           page_ptr,
                           bmt_bfap,
                           cur_mcell_id.page,
                           BS_NIL,
                           cur_ftx,
                           MF_VERIFY)
               if (sts == E_MAX_PINS_EXCEEDED)
                      ftx_done_n( cur_ftx)
                      sts = ftx_start_n( cur_ftx, acl_update_ftx )
                      if sts != EOK
```

```
                ftx_fail( acl_update_ftx)
                free resources
                return sts
        sts = rbf_pinpg( &pghdl,
                page_ptr,
                bmt_bfap,
                cur_mcell_id.page,
                BS_NIL,
                cur_ftx,
                MF_VERIFY)
        if (sts != EOK)
                ASSERT( mcellList_lk held for write )
                ftx_fail( cur_ftx )
                ftx_fail( acl_update_ftx )
                unlock( &bfap->mcellList_lk )
                unlock( &bfap->bfa_acl_lock )
                free resources
                return sts;


if (!rbf_can_pin_record( pghdl ) )
        ftx_done_n( cur_ftx )
        sts = FTX_START_N( cur_ftx, acl_update_ftx )
        if sts != EOK
                ftx_fail( acl_update_ftx )
                free resources
                return sts

        sts = rbf_pinpg( &pghdl,
                page_ptr,
                bmt_bfap,
                cur_mcell_id.page,
                BS_NIL,
                cur_ftx,
                MF_VERIFY)

        if (sts != EOK)
                ASSERT( mcellList_lk held for write )
                ftx_fail( cur_ftx )
                ftx_fail( acl_update_ftx )
                unlock( &bfap->mcellList_lk )
                unlock( &bfap->bfa_acl_lock )
                free resources
                return sts;

/* We can go ahead an pin the record modify it. */
mcell_ptr = &((bsMPgT)page_ptr)->bsMCA[cur_mcell_id.cell]

/* Pin the record header, and the terminator too.
 * The structure of this mcell is header, acl data, terminator
 * where the header has a record type of BSR_ACL and the
 * terminator has a type of BSR_NIL */
rbf_pin_record( pghdl,
                mcell_ptr,
                2 * sizeof( struct bsMR) +
                  ADVFS_ACLS_PER_MCELL * sizeof( advfs_acl ) )

/* Init header */
header_ptr = &mcell_ptr->bsMRO
ASSERT( header type is BSR_NIL )
header_ptr->bCnt = sizeof( struct bsMR ) +
        sizeof( bsr_acl_rec_t )
header_ptr->type = BSR_ACL

/* Init terminator */
term_ptr = &(mcell_ptr->bsMRO) +
                sizeof( bsr_acl_rec_t ) + sizeof( bsMR )
term_ptr->bCnt = sizeof( struct bsMR )
term_ptr->type = BSR_NIL

/* Copy data portion of new record */
```

```
                        record_ptr = &mcell_ptr->bsMCO + sizeof( struct bsMR )
                        record_ptr->bar_acl_link_seg = acl_link_seg
                        acl_link_seg++
                        record_ptr->bar_acl_type = SYSV_ACLS

                        acls_to_write = MIN (ADVFS_ACLS_PER_MCELL, acls_left_to_write )
                        record_ptr->bar_acl_cnt = acls_to_write
                        ((bsr_acl_rec_t*)record_ptr)->bar_acl_cnt = acls_to_write;

                        bcopy( cur_acl_array_ptr,
                                &((bsr_acl_rec_t*)record_ptr)->bar_acls,
                                acls_to-write * sizeof( advfs_acl ) )
                        acls_left_to_write -= acls_to_write
                        cur_acl_array_ptr += acls_to_write * sizeof( advfs_acl )

                        cur_mcell_id = mcell_ptr->nextMcellId
                ftx_done( cur_ftx )

/*
 * The ACL is now written out to the mcell records that it needs to be written to
 * at least with respect to the log. (The transactions are complete that describe the
 * data in the mcells for the ACL)
 */


/*
 * Get the primary mcell of the file and find the fs_stat record
 * Then reset the base modes
 */

sts = rbf_pinpg(        &pgRef,
                        (void **)&bmtp,
                        bfap->dmnP->vdpTbl[bfap->primMcell.volume]->bmtp,
                        bfap->primMcell.page,
                        BS_NIL,
                        acl_update_ftx,
                        MF_VERIFY_PAGE );
if( sts != EOK )
        /* We may need to undo mcell list changes, so
         * we need to hold the mcellList_lk for the fail
         * case. */
        if (mcellList_lk_held == FALSE)
                lock_write( &bfap->mcellList_lk )

        ftx_fail( acl_update_ftx )
        unlock( &bfap->mcellList_lk )
        unlock( &bfap->bfa_acl_lock )
        u.u_error = sts;
        free resources
        return sts;

mcellp = &bmtp->bsMCA[bfap->primMcell.cell];

/*
 * The mcellList_lk may already be held in write mode.  If we didn't take it for write
 * mode, we only need to take it for read now.
 */
if (!mcellList_lk_held)
        MCELLIST_LOCK_READ( &(bfap->mcellList_lk ) );
        mcellList_lk_held = TRUE
fsStats = (struct fs_stat *)bmtr_find( mcellp, BMTR_FS_STATS, bfap->dmnP );
if( fsStats == NULL )
        /* We may need to undo mcell list changes, so
         * we need to hold the mcellList_lk for the fail
         * case. */
        if (mcellList_lk_held == TRUE and !mcellList_lk held for write)
                if !upgrade( &bfap->mcellList_lk )
                        /* Upgrade failed.  Lock should be dropped */
                        lock_write( &bfap->mcellList_lk )

        ftx_fail( ftxAcl );
```

```
        unlock( &bfap->bfa_acl_lock )
        unlock( &bfap->mcellList_lk )
        free resources
        domain panic
        u.u_error = ENOSYS;
        return ENOSYS;

rbf_pin_record( pinPgH, fsStats, sizeof( struct fs_stat ) );
fsStats->st_mode = imode;

/* Set the mode bits */
context_ptr = VTOC( vp );
mutex_lock( &context_ptr->fsContext_mutex );
context_ptr->dir_stats.st_mode = imode;
mutex_unlock( &context_ptr->fsContext_mutex );

/* reset the acl cache with the new acl */
free( bfap->bfa_acl_cache );
bfap->bfa_acl_cache = advfs_acl;
bfap->bfa_bfa_acl_cache_len = sizeof( advfs_acl_t ) * num_tuples;

/* END TRANSACTION */
ftx_done_n( ftxAcl, FTA_ACL_UPDATE );

if (mcellList_lk_held)
        unlock( &bfap->mcellList_lk )
unlock( &bfap->acl_lock )

free undo record buffer


return EOK;
```

Exceptions

Could return `E_BAD_HANDLE`, `EIO`, `ENOSYS`, `EINVAL`, `EROFS`, or `ENOMEM`.

3.1.1.2 VOP_GETACL (advfs_getacl)

Interface

```
int
advfs_getacl(  struct vnode*          vp,            /* File to get ACL on */
               int                    num_tuples,    /* Number of ACL entries requested */
               struct acl_tuple_user* tupleset,      /* Buffer of new ACL entries */
               int                    acl_type       /* ACL type, should be SYSV_ACLS */
               )
```

Description

This routine is the AdvFS specific call out for `VOP_GETACL`. It takes the vnode pointer to a file, the number of ACL entries requested, and the type of ACL and returns a populated ACL array of the current list entries (`tupleset`). Each ACL entry is referred to as a "tuple".

The routine will make basic checks for valid ACL type, number of ACL entries, and domain panics before requesting ACL information from the BMT.

ACL information is stored in `BSR_ACL` records in the BMT. If this information was previously queried, then it should be stored in the bfap `bfa_acl_cache`. The routine can simply copy the data in the `bfa_acl_cache` in the bfap to a new `advfs_acl` buffer. If this cache pointer is null, then a look up on disk is necessary. This is accomplished by calling `advfs_getacl_bmt`.

The tuples are transferred from the `advfs_acl` buffer into the `tupleset` to be returned to the caller. If the number of tuples requested is zero, only the total number of tuples in the ACL is returned via `u.u_r.r_val1` and nothing is plugged into `tupleset`. If the total number of tuples found is more than the number requested, then we will return and error, `EINVAL`. This indicates that the caller provided a buffer too small for the ACL.

The routine returns `EOK` when successful.

Execution Flow

```
struct advfs_acl *advfs_acl;

/* Verify ACL type requested */
if( type != ADVFS_ACL_TYPE ) {
        u.u_error = ENOSYS;
        return ENOSYS;
}

/* Convert vnode to bfAccess */
bfap = VTOA( vp );

/* Check for domain panic */
if( bfap->dmnP->dmn_panic ) {
        u.u_error = EIO;
        return EIO;
}

/*
 * Verify that number of tuples requested is not more than max allowed by AdvFS
 * and that the number requested is not negative
 */
if( ( num_tuples > ADVFS_MAX_ACL_ENTRIES ) || ( num_tuples < 0 ) ) {
        u.u_error = EINVAL;
        return EINVAL;
}

/*
 * We can not take advantage of the acl cache.  We'll have to go to disk
 * to retrieve the ACL information.
 */
if( bfap->bfa_acl_cache == NULL ) {

        /*
         * Take the mcell list lock so we can read the ACL, and take the ACL
         * lock so we can reset the bfa_acl_cache once we get the ACL from disk
         */
        lock_write( &bfap->bfa_acl_lock );
        lock_read( &bfap->mcellList_lk );
        if( bfap->bfa_acl_cache == NULL ) {
                /* advfs_getacl_bmt will go to disk to get the ACL and reset the cache */
                sts = advfs_getacl_bmt( advfs_acl, &tuple_cnt, bfap );
                if( sts != EOK ) {
                        unlock(&bfap->mcellList_lk );
                        unlock(&bfap->bfa_acl_lock );
                        u.u_error = sts;
                        return sts;
                }

                unlock( &bfap->mcellList_lk );
                unlock( &bfap->bfa_acl_lock );
        } else {
                /* The acl cache is valid */
                tuple_cnt = bfap->bfa_acl_cache_len;
                advfs_acl = bfap->bfa_acl_cache;
        }
        unlock( bfap->bfa_acl_lock );

} else {

        lock_read( &bfap->bfa_acl_lock );

        /* Woops, we lost the acl_cahce while trying to grab the lock */
        if( bfap->bfa_acl_cache == NULL ) {
```

```
            /*
             * Take the mcell list lock so we can read the ACL, and upgrade the ACL
             * lock so we can reset the bfa_acl_cache once we get the ACL from disk
             */
            lock_read( &bfap->mcellList_lk );
            rwlock_upgrade( &bfap->bfa_acl_lock );

            /* advfs_getacl_bmt will go to disk to get the ACL and reset the cache */
            sts = advfs_getacl_bmt( advfs_acl, &tuple_cnt, bfap );
            if( sts != EOK ) {
                    unlock(&bfap->mcellList_lk );
                    unlock(&bfap->bfa_acl_lock );
                    u.u_error = sts;
                    return sts;
            }

            unlock( &bfap->mcellList_lk );

        } else {

            /* The ACL cache is vaild */
            tuple_cnt = bfap->bfa_bfa_acl_cache_len;
            advfs_acl = bfap->bfa_acl_cache;
        }

        unlock( bfap->bfa_acl_lock );

}

if( num_tuples >= 0 ) {

        /*
         * Error if number of tuples requested is less than number in the ACL
         * (i.e. They sent us a buffer that's too small to fit the ACL in)
         */
        if( num_tuples < tuple_cnt ) {
            u.u_error = EINVAL;
            return EINVAL;
        }


        /* plug the tuples into the tupleset */
        for( i = 0; i < tuple_cnt; i++ ) {
            tupleset[i].a_type = advfs_acl[i].aa_a_type;
            tupleset[i].a_id = advfs_acl[i].aa_a_id;
            tupleset[i].a_perm = advfs_acl[i].aa_a_perm;
        }
}

/* If the number of tuples requested was less than zero, only return the tuple_cnt */
u.u_r.r_val1 = tuple_cnt;

return EOK;
```

Exceptions

Could return `E_BAD_HANDLE`, `EIO`, `ENOSYS`, `EINVAL`, or `ENOMEM`

3.1.1.3 VOP_ACCESS (advfs_access)

Interface

```
int
advfs_access(  struct vnode*        vp,          /* in - vnode of the file to check */
               int                  mode,        /* in - requested mode of caller */
               struct ucred*        ucred        /* in - caller's credentials */
               )
```

Description

This routine is the AdvFS specific call out for VOP_ACCESS. It takes the vnode of the file to check (vp), the requested mode (mode), and credential structure of the caller (ucred). The routine first sets the statistic counter for the routine using FILESETSTAT, and then loads an fsContext structure from the vnode pointer and sets the file's permission mode (fmode) from the fs_stat structure.

If the requested mode includes WRITE access, and the file system is read only, then deny access and return EROFS. However, if the file is a block or character device resident on the file system, then do not return an error. If there is shared text associated with the vnode, the routine will try to free it up once. If that fails, then writing is not allowed.

Super user will always have read and write access, and super user will have execute access if the file is a directory, or if any execute bit is set.

After all of these initial checks, the routine calls advfs_bf_check_access to check further permissions, including the ACL entry. The mode according to what exists in the ACL is returned. If negative one is returned, there was an error, return EINVAL; otherwise, a number between zero and seven will be returned to indicate the caller's permissions. The requested mode will need to be shifted to compare with what is returned. If the modes match, access is granted, return zero. If they do not match, access is denied, and EACCES is returned.

Execution Flow

```
int m;
mode_t fmode;
struct fsContext *context_ptr;

/* Track the routine's access statistics */
FILESETSTAT( vp, advfs_access );

/* Get the fsContext info from the vnode */
if( ( context_ptr=VTOC( vp ) ) == NULL ) {
        u.u_error = EACCES;
        return EACCES;
}

/* Get this file's permission mode */
fmode = context_ptr->dir_stats.st_mode;

/* If the requested mode is WRITE, check a few things */
if( mode & S_IWRITE ) {

        /*
         * Do not allow write attempts on a read only file system, unless the
         * file is a block or character device resident on the file system
         */
        if( vp->v_vfsp->vfs_flag & VFS_READONLY ) {
                if( ( fmode & S_IFMT ) != S_IFCHR &&
                    ( fmode & S_IFMT ) != S_IFBLK ) {
                        u.u_error = EROFS;
                        return EROFS;
                }
        }

        /*
         * If there is shared text associated with the vnode, try to free it
         * up once.  If we fail, we can not allow writing.
         */
        if( vp->v_flag & VTEXT ) {
                xrele( vp );
                if( vp->v_flag & VTEXT ) {
                        u.u_error = ETXTBSY;
                        return ETXTBSY;
                }
        }
}
```

```
/*
 * If you are the super user, you always get read/write access.  Also you
 * get execute access if it is a directory, or if any execute bit is set.
 */
if( ( KT_EUID( u.u_kthreadp ) == 0 ) && ( ! ( mode & S_IEXEC ) ||
    ( ( fmode & S_IFMT ) == S_IFDIR ) || ( fmode & ANY_EXEC ) ) ) {
        return 0;
}

/*
 * Now that we've gotten some preliminary checks out of the way, let's
 * move on to the Access Control List (ACL).
 *
 * The routine advfs_bf_check_access will check the ACL and return the mode for
 * the caller.  The mode will be a number between 0 and 7. Representing the
 * following permissions:
 *
 *      0 ---          4 r--
 *      1 --x          5 r-x
 *      2 -w-          6 rw-
 *      3 -wx          7 rwx
 *
 * If -1 is returned, then there was an error, return EINVAL
 */
m = advfs_bf_check_access( vp, cred );
if( m == -1 ) {
        u.u_error = EINVAL;
        return EINVAL;
}

/*
 * We're only interested in VREAD, VWRITE, and VEXEC bits.
 *
 * Since the mode returned will be denoted by 0-7, we'll also need to shift
 * the bits of the requested mode to compare it to our caller's mode returned
 * from advfs_bf_check_access.
 */
mode &= ( VREAD | VWRITE | VEXEC );

/* Grant access */
if( ( m & mode ) == m ) {
        return 0;
}

/* Deny access */
u.u_error = EACCES;
return EACCES;
```

Exceptions

Could return `EROFS`, `ETXTBSY`, or `EACCES`

3.1.2 ACL support routines

3.1.2.1 advfs_verify_acl

Interface

```
statusT
advfs_verify_acl(      struct acl *tupleset,
                       int num_tuples )
```

Description

This routine will take the pointer to a buffer containing an ACL. It will verify that the ACL entries in the buffer are sorted in the acceptable order and that the buffer contains entries for all four of the base ACL entries, user, group, class, and other.

Execution Flow

```
int verify_flag = 0, i=0;

#define        VERIFY_USER    0x0001
#define        VERIFY_GROUP   0x0002
#define        VERIFY_CLASS   0x0004
#define        VERIFY_OTHER   0x0008
#define        VERIFY_ALL     (VERIFY_USER | VERIFY_GROUP | VERIFY_CLASS | VERIFY_OTHER)

/*
 * Base user should be the very first entry, if not, then we're already in violation of
 * the order, return error.
 */
if( tupleset[i].a_type != USER_OBJ )
       return EINVAL;
else
       verify_flag = verify_flag | VERIFY_USER;

/* Loop until we hit the base Group */
for( i=1; i<num_tuples, tupleset[i].a_type == GROUP_OBJ; i++ ) {
       /*
        * If we hit one of these before the Base group, then we're in violation
        * of the order
        */
       if(     ( tupleset[i].a_type == GROUP ) ||
               ( tuple_set[i].a_type == CLASS_OBJ ) ||
               ( tupleset[i].a_type == OTHER_OBJ ) )
               return EINVAL;
}

/* Make sure we stopped on the base group */
if( tupleset[i].a_type != GROUP_OBJ )
       return EINVAL;
else
       verify_flag = verify_flag | VERIFY_GROUP;

/* Loop until we hit the base class */
for( ; i<num_tuples, tupleset[i].a_type == CLASS_OBJ; i++ ) {
       /*
        * If we hit the base other before the Base class, then we're in violation
        * of the order return error
        */
       if( tupleset[i].a_type == OTHER_OBJ )
               return EINVAL;
}

/* make sure we stopped on the base class */
if( tupleset[i].a_type != CLASS_OBJ )
       return EINVAL;
else
       verify_flag = verify_flag | VERIFY_CLASS;

/* Next one should be base other */
i++;
if( tupleset[i].a_type != OTHER_OBJ )
       return EINVAL;
else
       verify_flag = verify_flag | VERIFY_OTHER;

/* Better not be any left in our buffer! */
if( i != num_tuples )
       return EINVAL;

/* Make sure we got all the base entries */
if( verify_flag != VERIFY_ALL )
```

```
        return EINVAL;

return EOK
```

Exceptions

This routine will return EINVAL if the ACL is not sorted in the acceptable order or if the buffer doesn't not contain all four of the base ACL entries.

3.1.2.2 advfs_acl_resize_mcells

Interface

```
statusT
advfs_acl_resize_mcells( ftxHT parent_ftx,
                         bfAccessT* bfap,
                         mcells_required,
                         &new_mcells,
                         &acl_mcell_start)
```

Description

This routine is intended as a mechanism for adjusting the number of mcells dedicated towards the ACL. The routine will either increase, decrease, or leave the same number of mcells that are usable by the ACL. On return, `new_mcells` will indicate the number of new mcells that are available for use by the ACL but are not yet initialized with a `BSR_ACL` record.

It is expected that the caller protect the mcell chain of the file being operated on by holding the `mcellList_lk` exclusively.

This routine will allocate up to `mcells_required` new mcells and each one will start a separate transaction. Additionally, the routine may attempt to extend a BMT one or more times while adding mcells. Thus the number of transactions started is on the order of `mcells_required` + transactions for BMT extends.

This routine assumes that all mcells with records allocated for the ACL are contiguous within the mcell chain and that mcells containing records for ACL contain no other records.

Execution Flow

```
Next_mcell_id =primMcell
Prev_mcell_id = NilMcellId
Last_acl_mcell = NilMcellId
acl_mcell_cnt = 0
*new_mcells = 0
*acl_mcell_start = NilMcellId
acl_link_seg = 0

while (nextMcellId.volume != 0 && (acl_mcell_cnt < mcells_required)
        /* Ref the next mcell and get the mcell pointer */
        sts = advfs_bmtr_mcell_refpg( bfap,
                                      next_mcell_id,
                                      mcell_ptr,
                                      page_ref);
        if (sts != EOK)
                return sts

        if (bmtr_find(BSR_ACL, mcell_ptr ) != NULL)
                /* found an ACL record, so we are within the contiguous ACL
                 * chain at this point  */
                acl_link_seg = mcell_ptr->linkSeg
                last_acl_mcell = next_mcell_id
                if (acl_mcell_start == NilMcellId)
                        acl_mcell_start = next_mcell_id
                acl_mcell_cnt++
```

```
                bs_derefpg( page_ref, BS_CACHE_IT )
        else if (last_acl_mcell != NilMcellId)
                /* done with chain of ACL mcells */
                bs_derefpg( page_ref, BS_CACHE_IT )
                break
        acl_link_seg = mcell_ptr->linkSeg+1
        prev_mcell_id = next_mcell_id
        next_mcell_id = mcell.nextMcellId
        deref mcell

if (mcells_required > acl_mcell_cnt)
        /* Append more acl mcells to contiguous chain */
        if (last_acl_mcell = NilMcellId)
                /* If there is not chain already, start at the
                 * end of the mcell chain. */
                last_acl_mcell = prev_mcell_id

        while (acl_mcell_cnt + *new_mcells < mcells_required)
                /* allocate_link_new_mcell will start one transaction
                 * for the mcell that is allocated and linked and may
                 * start additional transactions for a bmt_extend call. */
                allocate_link_new_mcell( bfap,
                                        last_acl_mcell,
                                        parent_ftx,
                                        new_mcell_id,
                                        BMT_NORMAL_MCELL,
                                        acl_link_seg)
                last_acl_mcell = *new_mcell_id
                if (acl_mcell_start == NilMcellId)
                        acl_mcell_start = *new_mcell_id

                *new_mcells++
else
        /* Remove any trailing acl mcells */
        if (last_acl_mcell = NilMcellId)
                /* There was no ACL found and none were added.*/
                ASSERT( mcells_required == 0)
                return EOK
        sts = advfs_bmtr_mcell_refpg( bfap,
                                next_mcell_id,
                                &mcell_ptr,
                                page_ref )
        if (sts != EOK)
                return sts

        page_refed = TRUE
        while (bmtr_find( BSR_ACL, mcell_ptr ) )
                next_mcell_id = mcell_ptr->nextMcellId
                bs_deref_pg ( pgref, BS_CACHE_IT )
                page_refed = FALSE
                /* bmt_unlink_mcells will start a transaction to unlink
                 * the mcell. */
                bmt_unlink_mcells( bfap->dmnP,
                                bfap->tag,
                                last_acl_mcell,
                                next_mcell_id,
                                next_mcell_id,
                                parent_ftx)
                if (next_mcell_id == NilMcellId)
                        /* We are at the end of the chain, must be done */
                        break
                else /* We may need to continue.  Ref the next mcell in the chain */
                        sts = advfs_bmtr_mcell_refpg( bfap,
                                                next_mcell_id,
                                                &mcell_ptr,
                                                page_ref )
                        if (sts != EOK)
                                return sts
                        page_refed = TRUE

        if (page_refed)
```

```
                bs_deref_pg( pgref, BS_CACHE_IT )

return EOK
```

Exceptions

This routine may return an error if `bmt_unlink_mcells` or `allocate_link_new_mcell` fails. Additionally, this routine may return errors if IO fails trying to read pages of the BMT.

3.1.2.3 advfs_acl_modification_undo

Interface

```
void
advfs_acl_update_undo( ftxHT ftx,
                                int32_t undo_rec_size,
                                void* undo_record_ptr )
```

Description

This is the undo routine for modifications to mcells. The undo routine is only required when the mcell being modified is not newly allocated. If the mcell were newly allocated, the undo routine for `allocate_link_new_mcell` would return the mcell to the BMT free list, so whether or not we modified the data that existed wouldn't matter, the next consumer will overwrite the mcell anyways.

In the case of a modified mcell, the entire previous record must be logged. The undo record that is passed in will consist of a header structure (`acl_update_undo_hdr_t`) followed by a variable number of `acl_update_undo_t` structures. The `acl_update_undo_t` structure contains the contents that need to be replaced in the mcell. This routine essentially uses a before image of the mcell to undo the changes.

This routine does not deal with locking. It is expected that if this routine is executed because of a call to `ftx_fail`, the call will hold the `bfa_acl_lock` and the `mcellList_lk` for write. If this routine is called during recovery, then the filesystem is running in a single threaded context and locking is not an issue.

This routine will unconditionally clear the acl cache in the bfAccess structure for which this undo is occurring.

Execution Flow

```
bcopy( undo_record_ptr, acl_undo_hdr, sizeof( acl_update_undo_hdr_t) )

if not in recovery mode
        ASSERT bfa_acl_lock held for write
        ASSERT mcellList_lk held for write

foreach undo record
        bcopy( undo_record_ptr + sizeof( acl_update_undo_hdr_t )
                + (cur_undo_index * sizeof( acl_update_undo_t ) ),
                cur_undo_rec, sizeof( acl_update_undo_t ) )
        /*
         * Pin the mcell in question
         */
        rbf_pinpg( cur_undo_rec.auu_mcell_id.page )
        if (sts != EOK)
                domain_panic(" advfs_acl_update_undo failed " )

        rbf_pin_record( pin the entire mcell record )
        bcopy( &cur_undo_rec.auu_previous_record,
                mcell record pointer,
                sizeof( bsr_acl_rec_t ) )

sts = bfs_access( acl_undo_hdr.auh_bf_set_id )
if (sts != EOK)
```

```
        domain panic
        return
sts = bs_access( acl_undo_hdr.auh_bf_tag, IN_MEM_ONLY )
if (sts != EOK)
        bfs_close( bfSet )
        domain panic
        return
if acl cache != NULL
        free acl cache
        acl cache = NULL
        acl cache length = 0
bs_close( bfap )
bfs_close( bfSet )

return
```

Exceptions

If this routine encounters an IO error while trying to pin the page which contains the mcell to be undone, then the routine in initiate a `domain_panic`.

3.1.2.4 advfs_getacl_bmt

Interface

```
statusT
advfs_getacl_bmt( struct advfs_acl*   advfs_acl,     /* out - buffer for ACL */
                  int*                        tuple_cnt,     /* out - number of tuples */
                  struct bfAccessT*           bfap           /* in/out – current file */
                  )
```

Description

This is a support routine for `advfs_getacl`. It will go to disk to look for the ACL. If there is a BMT record entry for an ACL, the tuples will be filled into the `advfs_acl` buffer, if not, base tuples are retrieved from the `fs_stat` structure stored in the BMT and inserted into the `advfs_acl` buffer. At this point, the bfap acl cache can be updated with the newly retrieved ACL.

It is expected that this routine is called with the `mcellList_lk` held for read or write and the `bfa_acl_lock` held for write.

The routine successfully completes by returning EOK.

Execution Flow

```
struct bsr_acl_rec *aclp;
struct bfMCIdT *mcellp;

/* Make sure we have the acl lock for writing and the mcellList lock for reading */
ASSERT( we have the bfap->bfa_acl_lock and the bfap->mcellList_lk );

/* Get the ACL from the BSR_ACL records in the BMT */
mcellp = bfap->primMCId;
if( ( aclp = (bsr_acl_rec *)bmtr_scan_mcells( &mcellp,
                                             &vdp,
                                             &bsrp,
                                             pgRef,
                                             recOffset,
                                             &rSize,
                                             BSR_ACL,
                                             bfap->tag ) ) != NULL ){
        /*
         * We found the first BSR_ACL record, if anymore exist, they
         * will be contiguous
```

```
         */
        Read the ACL from the records and plug it into the advfs_acl buffer.
        Increment tuple_cnt as we go along.

} else {

        /*
         * There was no BSR_ACL record.  Get the base modes from the stat structure
         */
        context_ptr = VTOC( vp );
        mutex_lock( &context_ptr->fsContext_mutex );
        fmode = context_ptr->dir_stats.st_mode;
        mutex_unlock( &context_ptr->fsContext_mutex );

        /* Malloc space for the base tuples in the ACL entry */
        advfs_acl = malloc( sizeof( advfs_acl ) * ADVFS_NUM_BASE_ENTRIES );
        *tuple_cnt = ADVFS_NUM_BASE_ENTRIES;

        One by one, we'll add each of the base tuples to the advfs_acl buffer
}

/* reset the acl cache with the new ACL */
ASSERT( bfap->bfa_acl_cache == NULL )
bfap->bfa_acl_cache = malloc( sizeof( advfs_tuple ) * (*tuple_cnt) );
bcopy( advfs_acl, bfap->bfa_acl_cache, ( sizeof(advfs_acl) * (*tuple_cnt) ));
bfap->bfa_bfa_acl_cache_len = *tuple_cnt;

return EOK;
```

Exceptions

This routine will return an error if it fails trying to read pages of the BMT.

3.1.2.5 advfs_bf_check_access

Interface

```
int
advfs_bf_check_access( struct vnode*          vp,             /* vnode of file to check */
                       struct ucred*          ucred           /* caller's credentials */
                       )
```

Description

This routine will perform more detailed checks on permissions for a file, by checking the ACL. The routine returns the caller's permissions as a number between zero and seven. The caller's uid and gid are taken from the credential struct passed in. The file's uid, gid, and mode (`fuid`, `fgid`, `fmode`) are taken from the `fsContext` structure built from the vnode pointer (`vp`).

If the `v_type` of this vnode is not one of the following, then access will be denied: `VDIR, VREG, VBLK, VCHR, VFIFO, VSOCK`, and `VLNK`.

The ACL will be plugged into a variable `acl` by calling `advfs_getacl`. The actual sorted ACL is returned by `advfs_getacl` along with the total number of entries via `u.u_r.r_val1`. If the return status is not `EOK`, then return negative one to indicate error.

The (u,g) tuples will be checked first. If there is a match, that mode is returned.

If no (u,g) entry is found, the routine moves on to (u,*) entries. The base entry is checked first to see if this caller is the file's owner, if not, the rest of the (u,*) entries in the ACL are checked.

The last set of entries to be checked is the (*,g) set. Again, if the caller is in the file's owning group, this base group tuple's mode is returned, otherwise, all (*,g) tuples are checked.

If all else fails, return the (*,*) base tuple entry, i.e. the base "other" mode.

Execution Flow

```
int32_t uid = cred->cr_uid;   /* the caller's uid */
int32_t gid = cred->cr_gid;   /* the caller's gid */
struct acl_tuple_user *acl;
int sts = 0, tuple_cnt=0, match = FALSE;
struct fsContext *context_ptr;
uid_t fuid;                   /* File's user id */
uid_t fgid;                   /* File's group id */
mode_t fmode;                 /* File's mode */

context_ptr = VTOC( vp );

/* Check the v_type, if it is not one of the following, deny access */
switch( vp->v_type ) {
        case VDIR:
        case VREG:
        case VBLK:
        case VCHR:
        case VFIFO:
        case VSOCK:
        case VLNK:
                break;
        default:
                /* access denied */
                return -1;
}

/* get the file's uid, gid, and base permissions */
mutex_lock( &context_ptr->fsContext_mutex );
fuid = context_ptr->dir_stats.st_uid;
fgid = context_ptr->dir_stats.st_gid;
fmode = context_ptr->dir_stats.st_mode;
mutex_unlock( &context_ptr->fsContext_mutex );

/*
 * To get the current ACL for this file (acl), we'll check the acl cache in the
 * bfAccess structure.  This will eliminate a trip to disk to get the info.  If
 * the cache does not exist, then we will call advfs_getacl which also returns
 * the base permissions in addition to the optional ones.  The ACL count will be
 * located in u.u_r.r_val1, making life much easier for this routine.
 */
if( bfap->bfa_acl_cache == NULL ) {

        acl = NULL;
        do {
                /*
                 * We've been through this loop before, make a note of
                 * the current acl buffer size
                 */
                if( acl != NULL )
                        acl_buf_size = tuple_cnt;

                /* Get the tuple count only by calling advfs_getacl with size zero */
                sts = advfs_getacl( vp, 0, acl, ADVFS_ACL_TYPE );
                if( sts != EOK )
                        return -1;
                tuple_cnt = u.u_r.r_val1;
                ASSERT( tuple_cnt > 0 )

                /*
                 * In an attempt to reduce memory fragmentation, we'll reuse
                 * the acl buffer if it is larger than the size we'll
                 * need this time around.  If it's not big enough, or
                 * if hasn't been allocated, malloc new space for it.
                 */
                if( ( acl == NULL ) || ( acl_buf_size < tuple_cnt ) ) {
                   if( acl != NULL )
                        free( acl );
                   ASSERT( tuple_cnt > 0 );
```

```
                acl = (acl_tuple_user *)malloc( sizeof( acl_tuple_user ) * tuple_cnt );
            }

            /*
             * Get the acl, if EINVAL is returned, something came along and changed
             * the ACL on us, the buffer wasn't big enough, let's loop and try again.
             */
            sts = advfs_getacl( vp, tuple_cnt, acl, ADVFS_ACL_TYPE );
            if( ( sts != EOK ) && ( sts != EINVAL ) )
                    return -1;

        } while( sts == EINVAL );
} else {

        lock_read( &bfap->bfa_acl_lock )

        /* Woops, we lost the acl cache while trying to grab the lock. */
        if (bfap->bfa_acl_cache == NULL) {

                unlock( &bfap->bfa_acl_lock );
                acl = NULL;
                do {
                        if( acl != NULL )
                                acl_buf_size = tuple_cnt;
                        sts = advfs_getacl( vp, 0, acl, ADVFS_ACL_TYPE );
                        if( sts != EOK )
                                return -1;
                        tuple_cnt = u.u_r.r_val1;
                        ASSERT( tuple_cnt > 0 )
                        if( ( acl == NULL ) || ( acl_buf_size < tuple_cnt ) ) {
                           if( acl != NULL )
                                free( acl );
                           acl = (acl_tuple_user *)malloc(
                                sizeof( acl_tuple_user ) * tuple_cnt );
                        }
                        sts = advfs_getacl( vp, tuple_cnt, acl, ADVFS_ACL_TYPE );
                        if( ( sts != EOK ) && ( sts != EINVAL ) )
                                return -1;
                } while( sts == EINVAL );

        } else {

                /* The acl cache is valid */
                acl = (acl_tuple_user *)malloc(
                        sizeof( acl_tuple_user ) * bfap->bfa_bfa_acl_cache_len );
                bcopy( bfap->bfa_acl_cache, acl, sizeof( bfap->bfa_bfa_acl_cache_len ) );
                tuple_cnt = bfap->bfa_bfa_acl_cache_len;
                unlock( bfa_acl_lock)
        }
}

/* Check if the caller's uid is the file's uid */
if( fuid == uid )
        return ADVFS_GETBASEMODE( fmode, ADVFS_ACL_USER );

/*
 * Loop through the user specified ids to see if there's an optional
 * entry for the caller's uid, break if we hit the group base
 */
for( i=0; i<tuple_cnt, acl[i].aa_a_type == ADVFS_GROUP_OBJ; i++ )
        if( ( acl[i].aa_a_type == ADVFS_USER ) && ( acl[i].aa_a_id == uid ) )
                return( acl[i].aa_a_perm & ADVFS_GETBASEMODE( fmode, ADVFS_ACL_CLASS ) );

/* Check if the caller's gid is the file's gid */
if( fgid == gid )
        return ADVFS_GETBASEMODE( fmode, ADVFS_ACL_GROUP );

/*
 * Loop through the group specified ids to see if there's an optional
 * entry for the caller's gid, break if we hit the other base.  There's
 * no need to reset i, since the acl is sorted.
```

```
 */
for( ; i<tuple_cnt, acl[i].aa_a_type == ADVFS_OTHER_OBJ; i++ )
        if( acl[i].aa_a_type == ADVFS_GROUP ) && ( acl[i].aa_a_id == gid ) )
                return( acl[i].aa_a_perm * ADVFS_GETBASEMODE( fmode, ADVFS_ACL_CLASS ) );

/* Return the file's other base mode since we didn't find anything else in the ACL */
return ADVFS_GETBASEMODE( fmode, ADVFS_ACL_OTHER );
```

Exceptions

None.

3.1.2.6 advfs_update_base_acls

Interface

```
int
advfs_update_base_acls(struct bfAccess*      bfap,          /* file to update */
                       uid_t                 new_owner      /* new owner */
                       uid_t                 new_group      /* new group */
                       mode_t                new_mode       /* Access mode */
                       )
```

Description

This routine will transactionally update the on disk ACL and in memory ACL cache with the given base ACLs. This routine must be called from advfs_chmod and advfs_chown to correctly update the base ACL entries in the ACL for the file. If this routine is not called, then only fsStat structure is updated and the ACL cache would reflect stale ACLs.

The routine will acquire the bfa_acl_lock for write and the mcellList_lk for read. This routine only needs to modify ACLs and does not ever need to grow or shrink the ACL. As a result, the routine is a specialized form of advfs_setacl that only sets the base ACL entries.

Execution Flow

```
lock_write( bfa_acl_lock )
if (bfap->bfa_acl_cache == NULL)
        /* Assumes bfa_acl_cache is NULL if we have no ACL except base.
         * Nothing to do */
        unlock bfa_acl_lock
        return EOK
lock_read mcellList_lk

sts = FTX_START_N( acl_owner_update )
if sts != EOK
        domain panic ( acls will be out of date )
        unlock locks
        return sts
first_acl_mcell = bfap->primMCId
sts = bmtr_scan_mcell( first_acl_mcell, BSR_ACL, pgref, record_ptr)
if sts != EOK
        domain panic( have an ACL cache but no ACL on disk??? )
        unlock locks
        return sts
bs_derefpg( pgref )
sts = advfs_bmtr_mcell_refpg( bfap, first_acl_mcell, mcell_ptr, page_ref)
if sts != EOK
        unlock locks
        domain panic( acls out of date )
        return sts
record_ptr = bmtr_find( BSR_ACL, mcell_ptr)
ASSERT( record_ptr != NULL) /* We already found it in this mcell, better still be there
*/
```

```
while record_ptr
        page_pinned = FALSE
        for i = 0; I < record_ptr->bar_acl_cnt; i++
                if bar_acls[i].aa_a_type == ADVFS_USER_OBJ ||
                        bar_acls[i].aa_a_type == ADVFS_GROUP_OBJ ||
                        bar_acls[i].aa_a_type == ADVFS_OTHER_OBJ
                        if !page_pinned
                                deref page
                                sts = rbf_pinpg
                                if error
                                        fail transaction
                                        unlock locks
                                        domain panic
                                        return sts
                                get pointer to were we are in acl record
                                page_pinned = TRUE
                                rbf_pin_record( bar_acls[i], sizeof( advfs_acl ) )
                if bar_acls[i].aa_a_type == ADVFS_USER_OBJ
                        bar_acls[i].aa_a_id = new_uid
                        bar_acls[i].aa_a_perm = user bits of new mode
                else if bar_acls[i].aa_a_type = ADVFS_GROUP_OBJ
                        bar_acls[i].aa_a_id = new_gid
                        bar_acls[i].aa_a_perm = group bits of new mode
                else if bar_acls[i].aa_a_type == ADVFS_OTHER_OBJ
                        bar_acls[i].aa_a_perm = other bits of new mode.
                        /* If we reached the other, then we are done processing */
                        record_ptr = NULL
                        break
        next_mcell = mcell_ptr->nextMcellId
        if (!page_pinned)
                bs_derefpg( pgref )
        sts = advfs_bmtr_mcell_refpg( bfap, next_mcell, mcell_ptr, page_ref)
        record_ptr = bmtr_find( BSR_ACL, mcell_ptr)
        if record_ptr == NULL
                bs_derefpg( page_ref )
                break
unlock mcellList_lk

/* On disk is now updated, so just update the bfa_acl_cache. */
for each acl in bfa_acl_cache
        if bar_acls[i].aa_a_type == ADVFS_USER_OBJ
                bar_acls[i].aa_a_id = new_uid
                bar_acls[i].aa_a_perm = user bits of new mode
        else if bar_acls[i].aa_a_type = ADVFS_GROUP_OBJ
                bar_acls[i].aa_a_id = new_gid
                bar_acls[i].aa_a_perm = group bits of new mode
        else if bar_acls[i].aa_a_type == ADVFS_OTHER_OBJ
                bar_acls[i].aa_a_perm = other bits of new mode.
                break

/*
 * Now update the fsStat ODS to make sure everything is in agreement
 */
fs_update_stats( acl_owner_update )

ftx_done( acl_owner_update )
unlock bfa_acl_lock

return EOK
```

3.1.3 Miscallaneous support routines

3.1.3.1 advfs_bmtr_mcell_refpg

Interface

```
statusT advfs_bmtr_mcell_refpg(  bfAccessT* bfap,
                              bfMCIdT mcell_id,
                              bsMCT* mcell_ptr,
```

```
                              bfPageRefHT *page_ref )
```

Description

This routine is just a wrapper for reading mcells more easily. The routine takes an `mcell_id` and returns a page reference of a page that must be derefed by the caller along with a pointer to the mcell in that page.

Execution Flow

```
sts = bs_refpg( page_ref,
               &page_ptr
               bfap->dmnP->vdpTbl[mcell_id.volume]->bmtp,
               mcell_id.page,
               FtxNilFtxH,
               MF_VERIFY_PAGE )
if (sts != EOK)
        return sts

mcell_ptr = ((bsMPgT*)page_ptr)->bsMCA[mcell_id.cell]

return EOK
```

3.1.3.2 rbf_pinpg

This routine will be modified to return the error `E_MAX_PINP_EXCEEDED` in the event that the maximum number of pinnable pages is exceeded. This will eliminate a system panic case in `rbf_pinpg`. Now, a transaction will fail gracefully if the number of transaction is exceeded rather than causing the system to panic.

3.1.3.3 rbf_can_pin_record

This will be a new routine that will check the number of records currently pinned in a given pinned page and return `TRUE` of `FALSE` depending on whether additional records can be pinned for that page. The routine allows transactions to maximize the number of pinned records on a single page and therefore conserve transactions.

3.1.3.4 advfs_access_constructor

This routine will initialize the `bfa_acl_lock`.

3.1.3.5 advfs_access_destructor

This routine will destroy the `bfa_acl_lock.`

3.1.3.6 advfs_dealloc_access

This routine will free the `bfa_acl_cache` if it is non-NULL.

3.1.3.7 advfs_process_access_list
3.1.3.8

This routine will free the `bfa_acl_cache` if it is non-NULL prior to recycling an access structure.

3.1.3.9 advfs_chown and advfs_chmod

These routines will have a call to advfs_update_base_acls before returning to the caller. advfs_chown will pass a new user and group id to advfs_update_base_acls and the previous mode while advfs_chmod will pass the previous user and group id to advfs_update_base_acls but a new mode. The call to advfs_update_base_acls will correct the ACL to reflect the permission changes and will update the dirty fsStat structure on disk.

3.1.3.10 fs_create

This routine will be modified to check the default ACL entries for directories. Any file or subdirectory created in a directory with an ACL will inherit those ACL entries.

3.1.3.11 advfs_getattr

This routine will be modified to set the va_aclv field to indicate that we support SYSV_ACLS. This is done by checking the acl_cache. If it is not null, then we have acls, set the va_aclv flag.

3.1.3.12 advfs_check_asserts

An assert will be added to verify that the sizeof a uid_t is equal to the size of an int32_t. This is to make sure that our on disk structure are valid if the uid_t structure is changed in a future release.

3.1.3.13 Recovery Changes

When linking contiguous mcells in the mcell chain for use by ACLs, each mcell will have the same linkseg value in the mcell header. This is done to prevent having to update the linkseg value of all trailing mcells. To allow recovery to continue to be able to reorder the mcells, a minor link seg value will be kept in the bar_acl_rec structure. Recovery code will need to be changed to correctly detect and use the minor link segment value. While the linkSeg values are only monotonically increasing, the bar_link_seg values will be contigous. No numbers should be skipped.

4 Dependencies

4.1 Behavior in a cluster

ACLs should behave well in a cluster.

4.2 Standards

This design should not impact standards.

4.3 Learning Products (Documentation)

Documentation will need to be updated to reflect the SYSV_ACL support for AdvFS.

5 Issues (Optional).

Medium Priority

- Add `BFA_BASE_ACL_ONLY` enumerated type to `bfa_flags_t` to indicate that only base tuples exist for this file. This will be used to eliminate a trip to disk to recover permissions in `advfs_bf_check_access`. It will need to be checked in `advfs_access`, set in `advfs_getacl`, and set or cleared in `advfs_setacl`.
  - Contact:
  - Status: Open.