AdvFS ACLs and Property Lists

Design Specification

Version 1.0

DB
MD

Copyright (C) 2008 Hewlett-Packard Development Company, L.P.

Table of Contents

Preface

Version 1.0 of the Advfs ACLs and Property Lists 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 the design for Property Lists and Access Control Lists (ACLs) for AdvFS on HPUX. Property Lists are a mechanism for storing general information that is associated with a specific file. Property Lists elements are pairs of name and data. ACLs are a security mechanism for defining extended permissions on a file. This design describes an implementation for SystemV ACLs.

The goals of this design include providing an efficient, scalable implementation of property lists that supports up to 1024 elements of ACLs and can be implemented within the schedule constraints facing AdvFS. The design does not describe user level interfaces for property lists. The design allows for arbitrarily large PL elements up to the limitations of AdvFS' transaction management system.

This design describes a mechanism for implementing large ACLs on top of the Property List infrastructure. Smaller ACLs are stored directly in a file's metadata.

1.2 Product Identification

Project Name	**Project Mnemonic**	**Target Release Date**
AdvFS Property List and ACL Design	AdvFS PL/ACL	

1.3 Intended Audience

This document is intended for review by those interested in the AdvFS on HPUX and its interaction with other kernel subsystems. Portions of the design describe ACLs on AdvFS and may be related to security engineers for HPUX.

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 property lists and ACLs on HPUX. The intent is to provide sufficient design detail for the design to be implemented by AdvFS kernel engineers. This design describes a single implementation for Property Lists and does not propose a mechanism that will be space efficient for small property list elements. Trade offs in the design are discussed inline.

This design does not attempt to implement the initial design of having two implementations for Property Lists (for small and large PL elements; see reference 1 in section 1.4).

1.6 Acknowledgments & Contacts

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

DA, TM, and SB.

1.7 Terms and Definitions

Term	Definition
PL	Abbreviation for property list
PL file	Refers to the data entity that exists for each file that has any associated property lists. The PL file has a vnode associated with it when open, but shares the access structure of the file who owns the property list.
PL block	The block size of the PL file. The PL block is determined at creation of the PL file based on the DEV_BSIZE of the device the PL file is created on. The block size allows VAU support and greater space efficiency that requiring entire pages to be formatted consistently.
PL Flat File	An ODS layout for the PL file that is very simple. While the design is not scalable, it will be quick to implement and will allow debugging of in memory structures without worrying about the ODS complexities. It will increase parallelism of debugging to have a simplified layout.
PL Indexed File	An ODS layout that mixes PL blocks of data and B-tree elements. The B-Tree and data elements may exist in any combination on an allocation unit for the PL file.
PL element	A single PL entry consisting of a name and data pair each of arbitrary length.
B-tree Node	A single PL block that represents a node in the btree
B-tree element	A single entry in a B-Tree node that contains a hash value for a PL element and an offset for the PL element in the PL file.
ACL	Access Control List. A mechanism for storing privledge information for a file. AdvFS will implement System V

	ACLs.

2 Design Overview

2.1 Design Approach

This design aims at having one implementation for property lists. The design also describes a means of directly mapping the storage for PL elements in extents associated with a single file. A vnode is created for the storage that maps PL elements, but only a single tag and access structure exist for the storage for both the PL and the parent file. Storage for PL files must be allocated in allocation units which will initially be 8k (ADVFS_METADATA_PGSZ). This design treats PL elements as metadata and requires all operations on the PL file to be transactionally logged.

Although PL files require at minimum 8k, AdvFS could support 4k PL metadata files without significant modification. This design does not describe those changes, but the PL file layout was designed to allow allocation units of 1k and up. If AdvFS implements an efficient small file support mechanism, the PL file should be able to take advantage of this to be more space efficient.

Discussions with security engineers indicates that typical usage for ACLs does not include very large ACLs. As a result, ACLs will be stored in a single BMT record until the ACL exceeds ADVFS_BMT_ACL_THRESHOLD. ACLs greater than ADVFS_BMT_ACL_THRESHOLD will be stored in a PL file which will require a full allocation unit.

This design made several tradeoffs to simplify implementation. One tradeoff was not allowing PL files to remove storage unless the parent file is deleted or the last PL element is removed. The design does not preclude future enhancements in this area. The design also does not call for atomic deletion of PL elements following an insertion (a modification of a PL element). This may leave two copies of a PL element in the PL file, however one will be marked invalid and cleaned up at some point in the future.

2.2 Overview of Operation

Each file will be able to have two extent chains. The first extent, starting in the primary mcell, will be the normal storage for the file. Open the file and doing reads and writes to the file will use the bfAccess structure bfVnode as a mapping handle for the UFC. For files that have PL elements associated with them, a second chain of extents will exist. The PL extents will begin with a record in the mcell chain for the parent file. The extent chain for the PL files has no primary mcell, only a record that points directly to an extra extents mcell.

Because the PL file will be mapped with its own extents, starting at offset 0, the UFC must have a way of differentiating between PL storage data and regular file data. A second vnode will exist for files with PL storage. The second vnode will be dynamically allocated and will provide a mapping handle for the UFC to cache PL data. The PL vnode (bfAccess.pl_vp) will be created when the parent file is opened and will be freed when the parent file is deallocated. When the storage associated with the PL file is read or written to, the pl_vp will be passed to getpage and putpage.

Direct read access to the PL file storage will be made through the .tags file. The PL storage of a file will be accessed using the notation <tag>PL. Because metadata will be eligible for PLs to support CPDA, metadata may need to be accessed as M6PL. Therefore, the PL is a suffix to prevent confusion with the prefix M for metadata.

No public interface is being made available for PL elements in the initial design. VOP_SETACL and VOP_GETACL may use the PL if the ACLs are sufficiently large, but a generic PL interface will not be made available. If and when a public interface is made available, advfs_pl_get and advfs_pl_set will provide the interface to the lower level PL routines.

2.3 Major Modules

2.3.1 Property List Kernel API

2.3.1.1 advfs_pl_get & advfs_pl_set

These routines will provide the highest level interface to the PL. Initially, these will not be made public. The routines are modeled after Tru64's high level interface routines and expect a uio structure to be available to pass PL information to and from user space.

2.3.2 Property List Internal API

2.3.2.1 advfs_pl_lookup, advfs_pl_insert & advfs_pl_delete

These routines provide interfaces for operations on single PL elements. The routines can work either on kernel memory or user space memory depending on the flags passed in.

2.3.3 Lookups of PL files via .tags

Lookups of PL files via .tags will be read only. No Direct IO or mmapping of PL files will be supported. Access to the PL files will be made using the parent file tag and the suffix "PL." For reserved metadata files, access will be made by using both the "M" prefix and the "PL" suffix. Opening of the PL file through .tags translates into an open of the parent file, but the vnode for the PL file is returned to the user. On last close of the vode for the PL file, the parent file will also be closed.

2.3.4 Extent Maps for Property List Files

Extents for PL files will be stored in a chain of mcells pointed to by a new BMT record in the parent's mcell chain. Once created, the BMT record will always exist, but will be marked valid and invalid as the PL file is created and destroyed (all storage is removed when the last PL element is removed).

The extent chain for PL elements will not have a primary mcell. Therefore, when storage is removed, it will be treated like a truncation rather than a deletion of a file. A truncation will create a pseudo primary mcell for the chain of extra extents records and will attach the psuedo primary mcell to the DDL.

2.3.5 Property List File Locks

The extents for the PL file will be locked with a rwlock for the PL file. The bfAccess.pl_file_lock will be held in write mode whenever modifications are being made to the PL file. The lock will be taken before modification transactions begin and released after the transacations end. The lock will not be required during recovery since recovery is single threaded.

2.3.6 Access Structure Modifications for Property List Files

Several fields will be added to the bfAccess structure to support the PL file. A pointer to the vnode for the PL file will exist along with several fields to describe the PL file.

2.3.7 ODS Property List File Layout

Most of the Property List will be self descriptive in that the first block of the PL file will contain information about the file itself. It is intended that the PL file support an index of the PL elements for quick searches, however, because of the complexity of implementing the index, only a framework is described. This design details a flat file layout for the PL element where the PL file is subdivided into blocks each of which maintains a free list of storage on that block.

PL elements will be chained together from head to head and each PL element may span one or more blocks.

2.3.8 Migrating PL Storage

Migration of PL storage will be very similar to migration of normal file storage with some differences in locking.

2.4 Major Data Structures

2.4.1 New Types

2.4.1.1 enum pl_file_layout_version

```
typedef enum {
       PLV_FLAT       = 0x1  /* ODS is a flat file of data */
       PLV_B_TREE     = 0x2  /* ODS is b-tree/data mix */
} pl_file_layout_version_t
```

2.4.1.2 enum pl_block_type

```
typedef enum {
       PLB_UNCLAIMED          = 0x1  /* An unused PL block, no data or index. */
       PLB_DATA               = 0x2  /* PL element data resides in the block */
       PLB_B_TREE_NODE        = 0x3  /* A b-tree node of an indexed PL file */
} pl_block_type_t
```

2.4.1.3 enum pl_control

```
typedef enum {
       PLC_NOT_ON_LIST= -0x1         /* PL block not on free list */
       PLC_END_OF_LIST= -0x2         /* End of PL free list */
       PLC_NO_CHILD   = -0x4         /* No child nodes for b-tree */
} pl_control_t
```

2.4.1.4 enum pl_flags

```
typedef enum {
       PLF_INSERT             = 0x1         /* Lookup called from advfs_pl_insert */
       PLF_DELETE             = 0x2         /* Lookup called from advfs_pl_delete */
       PLF_LOOKUP             = 0x4         /* Lookup called for read of PL.     */
       PLF_FOUND              = 0x8         /* Found the requested element        */
       PLF_NOT_FOUND          = 0x10        /* Did not find the requested element */
       PLF_ACL                = 0x20        /* We are dealing with an ACL, so we need to
                                             * do bcopy's instead of uio moves.  The uio
                                             * structure is just dummied up. */
       PLF_CREATE             = 0x40        /* Request that advfs_pl_access create the
                                             * PL file if it does not exist. *
       PLF_CREATING           = 0x80        /* advfs_pl_create is in process of creating
                                             * the PL file. */
       PLF_CLEANUP            = 0x100       /* Flag to advfs_pl_delete.  Only delete the
                                             * element at the specified offset if it is
                                             * PLST_ELEM_INVALID */
       PLF_RELOAD_MAPS        = 0x200       /* Force the extent maps to be reloaded.
                                             * They should already have been deleted */
} pl_flags_t
```

2.4.1.5 enum meta_flags_t

```
typedef enum {
       MF_NO_FLAGS         = 0x0     /* No flags */
       MF_NO_VERIFY        = 0x1     /* Do not verify data integrity */
       MF_VERIFY_PAGE      = 0x2     /* MF_NO_VERIFY is sufficient to make
```

```
                                   * a VERIFY/NO VERIFY decision, but this
                                   * provides clarity */
        MF_PL_FILE          = 0x4     /* Pin/ref the PL file, not the bfap's data */
} meta_flags_t
```

2.4.1.6 enum pl_trans_types

```
typedef enum {
        PL_INS_HDR_UNDO,              /* Undo a block header change */
        PL_INS_SEG_UNDO,              /* Undo segment header change */
        PL_DEL_HDR_UNDO,              /* Undo a block header change */
        PL_DEL_SEG_UNDO,              /* Undo a segment header change */
        PL_DEL_FILE_HDR_UNDO          /* Undo a file header change */
} pl_trans_types_t
```

2.4.1.7 enum pl_segment_type

```
typedef enum {
        PLST_ELEM_HDR,
        PLST_ELEM_CONT,
        PLST_FREE_SEG,
        PLST_ELEM_INVALID
} pl_segment_type_t
```

2.4.1.8 enum stg_flags_t

```
typedef enum {
    STGF_REL_QUOTA    =       0x1     /* Release quota blks when doing stg removal */
    STGF_PL_FILE      =       0x2     /* Remove storage for the PL file */
    STGF_DO_COW       =       0x4     /* 1 or more external opens */
} stg_flags_t;
```

2.4.1.9 enum pl_ods_flags_t

```
typedef enum {
    PLOF_VALID        =       0x1     /* The PL file is currently valid (at least 1
                                       * of initialized storage */
    PLOF_INVALID      =       0x2     /* The PL file has no storage. */
} pl_ods_flags_t;
```

2.4.1.10 enum lvl_flags_t

```
typedef enum {
    LF_PL_FILE        =       0x1     /* The redo records described in this level are
                                       * for the PL file */
} lvl_flags_t;
```

2.4.1.11 enum mig_flags_t

```
typedef enum {
        MIG_FORCE     =       0x1     /* Force the migrate */
        MIG_PL_FILE   =       0x2     /* The migrate is for the PL file storage */
} mig_flags_t;
```

2.4.1.12 enum advfs_handyman_msg_type_t

```
typedef enum {
    AHME_FINISH_DIR_TRUNC,      /* Finish a directory trunction */
    AHME_RETRY_IO,              /* Perform an IO Retry */
    AHME_DELETE_PL_ELEMENT,     /* Perform async delete of PL element
    AHME_THREAD_GO_AWAY,        /* Kill the thread */
    AHME_CFS_XID_FREE_MEMORY    /* Clean up memory on failover recovery */
} advfs_handyman_msg_type_t;
```

2.4.2 In Memory Structures

2.4.2.1 bfAccess

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

```
typedef struct bfAccess {
.
.
.

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

~~    bfNode bfNp                       /* To be removed.  */~~
    bfVnode                           /* Bfap's vnode */
       v_data = &bfap                 /* Now, v_data will point to vnode */
.
.
.

/*
 * Property List related fields
 */
    rw_lock_t pl_file_lock            /* Take for write when inserting or deleting a PL
                                       * from the PL file.  Taken for read for any lookups
                                       * that are successful and not for modification.
                                       */
    pl_file_layout_version_t pl_layout  /* File layout of the PL file {b-tree,flat} */
    size_t pl_file_size               /* Size in bytes of the proptery list */
    size_t pl_block_size              /* Size in bytes of each pl block, determined at
                                       * pl file creation or off disk when opened */
    bf_fob_t pl_alloc_size            /* Allocation unit size of the PL file */
    off_t pl_free_blk_off             /* Offset of first free block in PL file */
    uint32_t pl_element_cnt           /* Number of elements in the PL file */
    size_t pl_free_space              /* Bytes available for PL elements */
    uint64_t pl_free_blocks           /* Number of completely free PL blocks */
    struct vnode* pl_vp               /* Vnode for property list file if one exists
                                       * and has been accessed during this open of the
                                       * file.  Set to NULL at recycle and deallocate
                                       * and initialize at first access to the pl file
                                       */
    /*
     * ACL cache pointer
     */
    advfs_acl_tuple_t  *acl_cache;    /* set up on open, points to access control list */
    size_t             acl_cache_len; /* length of acl_cache */

} bfAccessT;
```

2.4.2.2 struct bsInMemXtnt

This structure is a wrapper for the bsInMemXtntMapT structures that are actually the extent maps. The PL file's extent maps will be contained in this structure along with the regular extent maps.

```
typedef struct bsInMemXtnt {
    xtnt_valid_type_t validFlag;      /* XVT_VALID if extents are valid.    *
                                       * XVT_INVALID if extents are invalid */
    bsInMemXtntMapT *xtntMap;         /* Link to primary extent map */
    ~~bsStripeHdrT *stripeXtntMap;      /* Link to stripe extent maps */~~
    bsInMemXtntMapT *copyXtntMap;     /* Link to copy extent maps */
    bsInMemXtntMapT *plXtntMap        /* Xtnt maps for PL file.  Valid if bfap->pl_vp  */

    rwlock_t migStgLk;            /* Serialize migrate and add/remove of stg */

    bsXtntMapTypeT type;          /* The type of extent maps */
    bf_fob_t bimxAllocFobCnt;     /* Number of allocated 1k file offset blks */
} bsInMemXtntT;
```

2.4.2.3 struct bfNode

The bfNode will be removed from the bfAccess structure. The information contained in the bfNode is redundant with the information kept in the bfAccess structure. For this reason, the bfNode is not required. Additionally, for the VTOA macro to function, the v_data, either the pl_vp needs to have its own bfNode, or both the bfVnode and the pl_vp need to point directly to the access structure via the v_data field.

2.4.2.4 struct domain

```
typedef struct domain {
.
.
.
       mutex_t pl_cv_mutex    /* Mutex for the pl_cv and the pl_big_insert field */
       int64_t pl_big_insert_cnt      /* Number of "big" inserts occurring */
       cv_t    pl_cv;         /* Condition variable to wake PL inserts that are
                               * blocking on another transaction to complete */


} domainT;
```

2.4.2.5 advfs_handyman_thread_msg_t

```
struct advfs_handyman_thread_msg {
    advfs_handyman_msg_type_t ahm_msg_type;     /* Message type */
    union {
       dtinfoT        ahm_dtinfo;            /* Dir truncation info structure */
       struct buf *   ahm_io_retry_bp;       /* buf struct for io retry */
       bfDomainIdT    ahm_xid_free_dmnId;    /* domainId from cfs_xid_free_memory */
       pl_cleanup_t   ahm_pl_cleanup;        /* Message to cleanup pl element */
    } ahm_data;
};
typedef struct advfs_handyman_thread_msg advfs_handyman_thread_msg_t;
```

2.4.2.6 pl_cleanup_t

```
struct pl_cleanup {
       bfTagT         plc_tag;
       bfSetIdT       plc_bf_set_id;
       off_t          plc_element_offset;
       int64_t        plc_element_hash;
};
typedef struct advfs_pl_clean advfs_pl_cleanup_t;
```

2.4.2.7 pl_imm_element

```
struct pl_imm_element {
       size_t plie_name_len  /* Size of name buffer in bytes */
       size_t plie_data_len  /* Size of data buffer in bytes */
       char * plie_name      /* Pointer to name. */
```

```
        char * plie_data       /* Pointer to data.  Only set in some occasions */
}

typedef struct pl_imm_element pl_imm_element_t;
```

2.4.2.8 pl_data

```
struct pl_data {
        pl_imm_element_t      pld_element    /* The PL element to be worked on */
        off_t                 pld_offset     /* Offset of pl_segment_hdr in PL file */
        struct uio*           pld_uiop       /* UIO structure for PL element data */
}
typedef struct pl_data pl_data_t
```

2.4.2.9 advfs_acl_tuple

```
/* AdvFS version of the acl_tuple_user structure defined in acl.h */
struct{
        int32_t uid;          /* user ID */
        int32_t gid;          /* group ID */
        unsigned char mode;   /* permission mode */
} advfs_acl_tuple;

typedef struct advfs_acl_tuple advfs_acl_tuple_t;
```

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

2.4.3.1 pl_file_hdr

```
struct {
        pl_file_layout_version_t pfh_layout;  /* Flat or b-tree layout */
        int64_t       pfh_elem_start;         /* Start of element list in PL file */
        int64_t       pfh_index_start;        /* Root node of index */
        int64_t       pfh_next_free_blk_off;  /* Offset of first free pl block */
        int64_t       pfh_free_byte_cnt       /* Bytes available in the file */
        int64_t       pfh_wasted_byte_cnt     /* Total wasted bytes in the file */
        int64_t       pfh_free_blk_cnt        /* Full free blocks (PLB_UNCLAIMED)
                                               * available */
        int64_t       pfh_file_size;          /* byte offset of last allocated FOB */
        int64_t       pfh_element_cnt         /* Number of PL elements in the PL file */
        int64_t       pfh_pl_block_size;      /* Size in bytes of pl block, determined
                                               * based on DEV_BSIZE at pl file creation
                                               * time */
} pl_file_hdr
typedef struct pl_file_hdr pl_file_hdr_t;
```

2.4.3.2 pl_block_hdr

```
struct {
        int64_t pbh_next_free_blk_off;        /* Offset of next block in free list*/
        int64_t pbh_prev_free_blk_off         /* Offset of prev block in free list */
        int64_t pbh_free_chunk_off;           /* block relative offset of free data in
                                               * this block*/

        pl_block_type_t pbh_block_type;       /* Type of block */
} pl_block_hdr
typdef struct pl_block_hdr pl_block_hdr_t;
```

2.4.3.3 pl_element_hdr

```
struct {
        uint64_t pleh_name_len;               /* Length of name of entry */
        uint64_t pleh_data_len;                /* Length of data of entry */
        uint64_t pleh_name_hash;               /* Hash of name for quicker lookups */
        int64_t pleh_flags;                    /* PL_ELEMENT_VALID | PL_ELEMENT_INVALID */
} pl_element_hdr;
typdef struct pl_element_hdr pl_element_hdr_t;
```

2.4.3.4 pl_segment_hdr

```
struct {
        pl_segment_type_t pls_type     /* PLST_FREE_SEG or PLST_ELEM_SEG or PLST_CONT_SEG
*/
        int64_t pls_next_offset        /* Offset of next element segment or free chunk */
        int64_t pls_prev_offset        /* Offset of prev element segment or free chunk
                                        * Not maintained for PLST_FREE_SEG types. */
        int64_t pls_cont_off;          /* Offset of element continuation.  */
        int64_t pls_size               /* Size of segment including hdr */
        pl_element_hdr_t pl_element_hdr
} pl_segment_hdr

typedef struct pl_segment_hdr pl_segment_hdr_t
```

2.4.3.5 pl_bt_node_hdr

This structure will defined when indexing is supported of PL files.

2.4.3.6 pl_bt_element

This structure will defined when indexing is supported of PL files.

2.4.3.7 BSR_PL_FILE

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

2.4.3.8 struct bsr_pl_rec

```
struct {
        bfMCIdT         bpr_xtnt_mcell;         /* mcell ID of PL file xtnts mcell */
        bf_fob_t        bpr_alloc_sz            /* Allocation unit for pl file */
        pl_ods_flags_t bpr_flags;               /* On disk flags {VALID, INVALID} */
} bsr_pl_rec

typedef struct bsr_pl_rec bsr_pl_rec_t
```

2.4.3.9 BSR_BMT_ACL

This record will replace BSR_PROPLIST_DATA (20) and will represent a bsr_bmt_acl_rec in the BMT.

2.4.3.10 struct bsr_bmt_acl_rec

```
struct {
        int32_t         bba_acl_cnt;            /* mcell ID of PL file xtnts mcell,
                                                 * -1 indicates invalid record
                                                 *(not used) */
        advfs_acl_tuple_t bpr_acls[ADVFS_BMT_ACL_THRESHOLD];
                                                /* ACL array. */
} bsr_pl_rec

typedef struct bsr_pl_rec bsr_pl_rec_t
```

2.4.3.11 Exisiting PL ODS structures and definitions

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.

2.4.4 Log Structures

2.4.4.1 pl_generic_ hdr_undo

This structure is a generic undo header for variable length undo records. The structure tells the undo routine how many repeating structures follow the header and which file the undo is associated with.

```
struct {
       uint64_t plgu_multi_cnt;      /* Number of repeated structures after this record*/
       bfTagT   plgu_bf_set_tag;     /* Tag of bfSet of parent of PL file */
       bfTagT   plgu_bf_tag;         /* Tag of parent file of PL file */


} pl_generic_hdr_undo
typedef struct pl_generic_hdr_undo pl_generic_hdr_undo_t
```

2.4.4.2 pl_ins_del_multi_undo

This structure may occur between 1 and *n* times after the pl_generic_hdr_undo record, once for each free chunk header that may need to be redone.

```
struct {
       pl_trans_types_t plidu_type;  /* Type of undo record. */
       uint64_t       plidu_offset;  /* Offset of structure to be restored/undone */
       union {
              pl_block_hdr_t plidu_prev_pl_block_hdr
              pl_segment_hdr plidu_prev_pl_seg_hdr
              pl_file_hdr_t plidu_prev_file_hdr
       } plidu_u

} pl_ins_del_multi_undo

typedef struct pl_ins_del_multi_undo pl_ins_del_multi_undo_t
```

2.4.4.3 lvlPinTblT

```
typedef struct {
    int32_t ftxPinS;            /* ftx pin page slot */
    int32_t numXtnts;           /* number of record extents */
    ftxRecXT recX[FTX_MX_PINR]; /* record extent list */
    lvlFlagsT lvl_flags         /* Flags for this level transaction */
} lvlPinTblT;
```

2.4.5 Macros

2.4.5.1 VTOA

This macro will be modified to directly access the bfap from the v_data field. It will have the same semantics with respect to shadow and real bfaps, but will directly reference the bfap through the v_data field rather than through the bfNode.

2.4.5.2 advfs_setbasemode

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

2.4.5.3 advfs_getbasemode

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

2.4.6 Constants

2.4.6.1 ADVFS_PL_FILE

This constant is a flag for the high order bits of the sequence. Since bit will be bit 63 in the 64 bit sequence field of the tag. This field will be set for the pageRedo structure and several undo and redo storage structures to indicate that the file being operated on was the PL file and not the regular file data.

2.4.6.2 ADVFS_PL_MIN_FREE_CHUNK_SZ

This constant represents that size at which a free chunk will be abandoned and not tracked on the free list. Chunks smaller than this constant are counted as wasted space in the PL file. At a later date, autotune (vfast) or some other mechanism may try to deal with wasted space if it becomes a problem. Allowing wasted space allows a simple implementation and relies on the assumption that removals and updates are infrequent and reads are common place.

2.4.6.3 ADVFS_PL_EXCLUSIVE_INSERT_THRESHOLD

This value represents the size over which a PL element insert will block other large PL inserts in order to prevent stressing the log with PL modifications. PL elements inserts which are greater than this size will be exclusive on a per domain basis. The idea is somewhat analogous to an exclusive transaction but applies only to PL insert transactions. Note that if a PL element that is greater than the threshold is being inserted, small PL element inserts will still be allowed, only large inserts will be blocked.

2.4.6.4 ADVFS_PL_BIG_INSERT_MAX

This defines the number of “big” inserts that can be occurring simultaneously. The value will be 1 initially which will make “big” inserts exclusive.

2.4.6.5 ADVFS_PL_FULL_BLOCK_THRESHOLD

This defines the size over which a PL element will only be spanned over completely free PL blocks. Elements under this size will use a first available algorithm over free space. Above this size, PL elements will only be written into fully free blocks. This is necessary to calculate an upper bound on the number of transactions started on large writes.

2.4.6.6 ADVFS_PL_MAX_ELEMENT_SIZE

This macro defines the maximum size of a PL element. The size will be limited by the number of transactions that can occur as subtransactions of a root transaction. This value will be determined empirically but is expected to be in the range of 200k-500k.

2.4.6.7 ADVFS_ACL_TYPE

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

2.4.6.8 ADVFS_MAX_ACL_ENTRIES

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

2.4.6.9 ADVFS_BMT_ACL_THRESHOLD

```
/* Largest amount of ACL entries that can be stored in a BMT record */
#define ADVFS_MAX_ACL_ENTRIES_BMT    20
```

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

2.4.6.10 ADVFS_NUM_BASE_ENTRIES

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

2.4.6.11 General ACL #defines from acl.h

```
/* The following defines are based on acl.h defines*/
#define ADVFS_ACL_UNUSED      -35     /* ACLUNUSED */
#define ADVFS_ACL_BASE_USER   -36     /* ACL_NSUSER, Non-Specified user */
#define ADVFS_ACL_BASE_GROUP  -36     /* ACL_NSGROUP, Non-Specified group */
#define ADVFS_ACL_USER        6       /* ACL_USER */
#define ADVFS_ACL_GROUP       3       /* ACL_GROUP */
#define ADVFS_ACL_OTHER       0       /* ACL_OTHER */
```

2.4.6.12 acl_tuple_unused

```
/* taken from ufs_setacl.c, used for initialization */
struct acl_tuple_user acl_tuple_unused = {
       ACLUNUSED, ACLUNUSED, 0
};
```

2.4.6.13 advfs_acl_tuple_unused

```
/* AdvFS version of the above structure, used for initialization */
struct advfs_acl_tuple_t advfs_acl_tuple_unused = {
       ADVFS_ACL_UNUSED, ADVFS_ACL_UNUSED, 0
};
```

2.4.6.14 struct vnodeops advfs_pl_vnodeops

This is a callout structure for PL file vnode operations. The structure will only define VOP_LOOKUP, VOP_READ, VOP_CLOSE, and VOP_OPEN. All other VOPs are invalid on PL files opened through .tags and will be set to advfs_pl_vop_not_supported which will return E_NOT_SUPPORTED.

2.4.7 Errors

2.4.7.1 E_MX_PINP_EXCEEDED

rbf_pinpg will be modified to return this error when FTX_MX_PINP is exceed. This will cause an error situation instead of a system-wide panic.

2.4.7.2 E_TOO_BIG

This error is returned when a PL element exceeds PL_MAX_ELEMENT_SIZE.

2.5 Exception Conditions

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

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

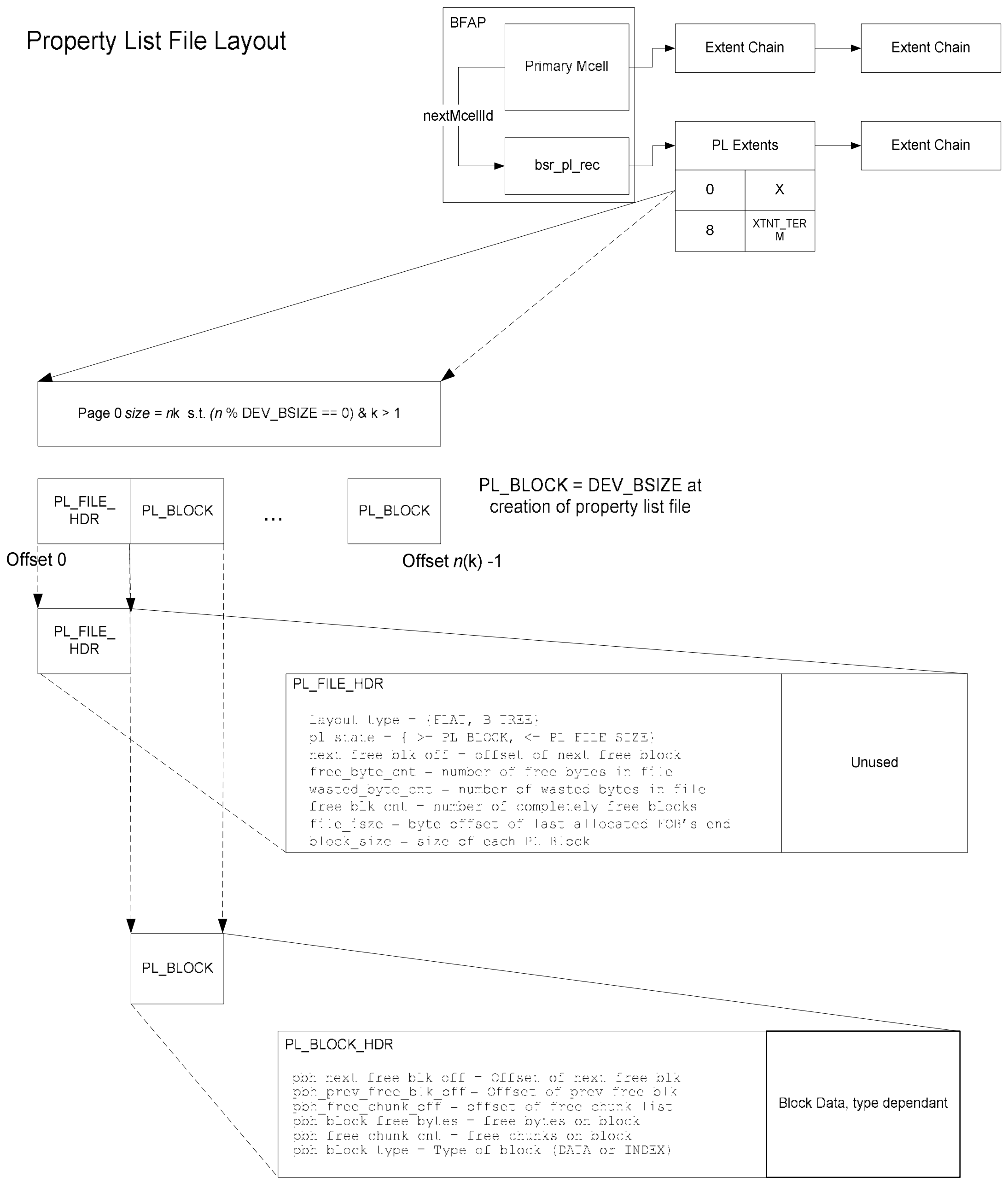
Specific cases of these general exception conditions that are applicable to this design are discussed in the following subsections.

3 Detailed Design

3.1 Data Structure Design

This section provides an overview of how the on disk data structures will be layed out. Section 3.2 will describe the algorithms for modifying the data structures on disk and in memory and will detail algorithms for accessing and protecting (locking) the property list file.

3.1.1 Basic PL file Layout



Property List File Layout
BFAP
Primary Mcell
nextMcellId
bsr_pl_rec
Extent Chain
Extent Chain
PL Extents
Extent Chain
0
X
8
XTNT_TERM
Page 0 *size* = *n*k s.t. (*n* % DEV_BSIZE == 0) & k > 1
PL_FILE_HDR
PL_BLOCK
…
PL_BLOCK
PL_BLOCK = DEV_BSIZE at creation of property list file
Offset 0
Offset *n*(k) -1
PL_FILE_HDR
PL_FILE_HDR
layout type – {FLAT, B TREE}
pl state – { >= PL BLOCK, <= PL FILE SIZE}
next free blk off – offset of next free block
free_byte_cnt – number of free bytes in file
wasted_byte_cnt – number of wasted bytes in file
free blk cnt – number of completely free blocks
file_isze – byte offset of last allocated FOB's end
block_size – size of each PL Block
Unused
PL_BLOCK
PL_BLOCK_HDR
pbh next free blk off – Offset of next free blk
pbh_prev_free_blk_off – Offset of prev free blk
pbh_free_chunk_off – offset of free chunk list
pbh block free bytes – free bytes on block
pbh free chunk cnt – free chunks on block
pbh block type – Type of block (DATA or INDEX)
Block Data, type dependant

Figure 1

The PL file is organized as a series of PL blocks. The size of a PL block is determined at PL file creation time (the first time a PL element is created) based on the sector size of the device that will hold the PL file's primary extent. In the event that AdvFS supports variable sector sizes in the future, the PL block size will remain the same even if the PL file is migrated to a volume with a different PL block size than the original. Selecting the DEV_BSIZE of the initial block will optimize performance for that sector size and will allow the property list to support space efficient indices by having both b-tree index nodes and PL element data on the same allocation unit, thereby reducing IO for small and medium PL file sizes.

Each PL file will have the first PL block start at offset 0. The first PL block will contain PL file information related to the file layout, file size, and PL block size (see Figure 1). The first PL block will also contain the head pointer for the free list of available storage, and will contain an offset of the first PL block of data for a flat PL file, or the root node of a B-tree for an indexed PL file. The first PL block will have a header structure of type pl_file_header_t and the remainder of the PL block will be 0 filled and unused.

Free space in the PL file is maintained in chunks. A PL file free list is maintained which chains together all PL blocks that contain any free chunks available for PL elements. The free list is maintained using the pl_block_hdr_t.pbh_next_free_blk_off field. Within a PL block, free chunks are chained together using the pl_block_hdr_t.pbh_free_chunk_off as a list head pointer with a PL block relative offset. The free chunks are chained together using the pl_free_hdr_t.plfh_next_free_off field. Both the PL file free list and the PL block free chunk lists are terminated with the PLC_END_OF_LIST offset.

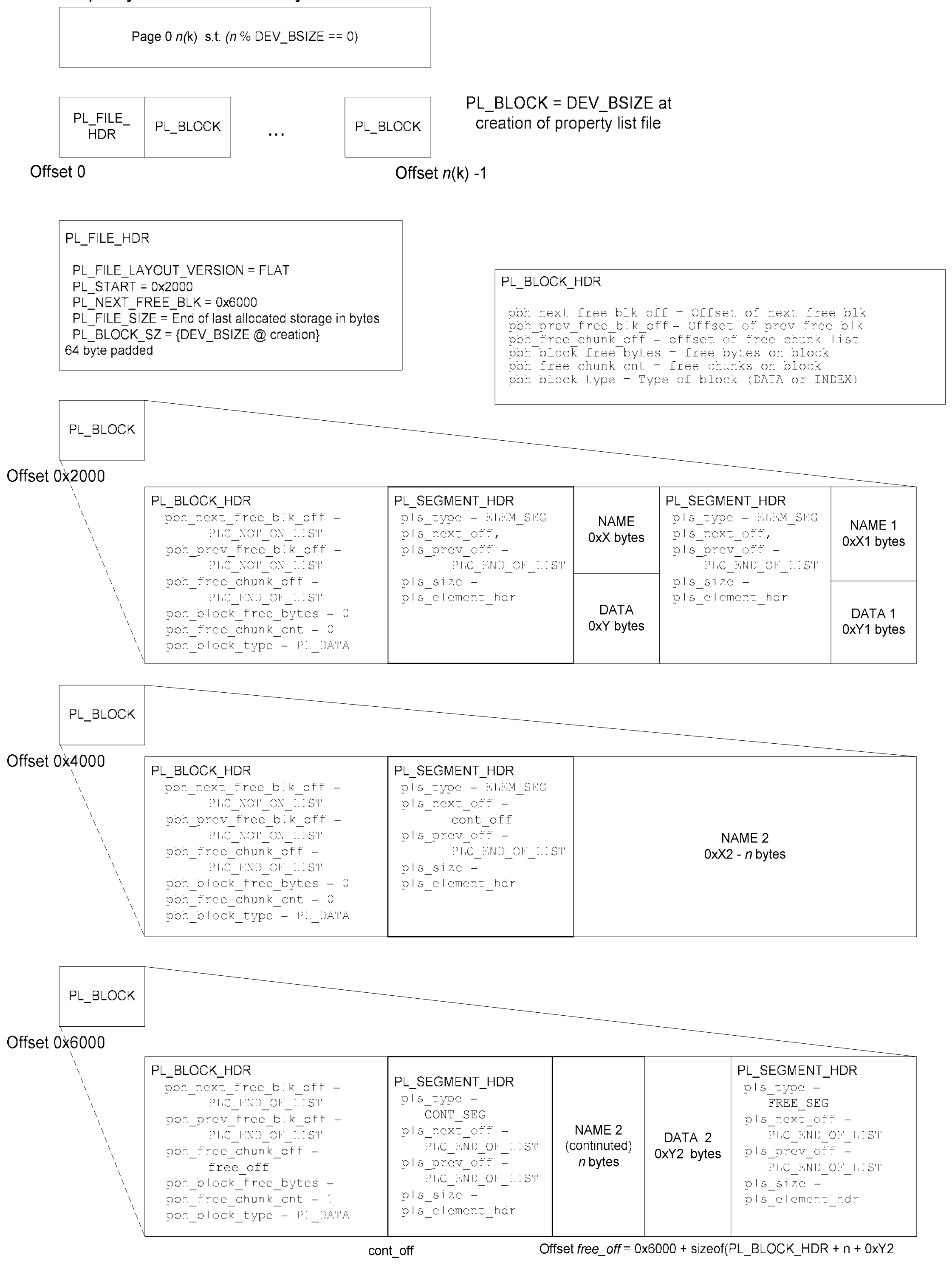
Because free space is maintained as flat lists, insertion of a new element into the PL file, or replacement of a PL element with larger data will require an O(n) walk of the free list and O(m) walk of each PL block's free space where n=Number of PL blocks on the PL file free list and m=number of non-contiguous free chunks in each PL block. It is not a goal of this design to provide an efficient free space mechanism for PLs since it is assumed that common usage will be heavy on lookups and not inserts and updates.

Each PL block will be in one of several states: on the free list with no PL elements, on the free list with some PL elements, not on the free list with PL elements or not on the free list and representing a B-tree node. If a PL block is on the free list, it has some number of free chunks available for additional PL elements.

Since the PL block size will be a multiple of the sector size of the volume the PL file is created on, and since sector size will be both a limiting factor for how small, and what multiples, allocation units can be in a domain, having PL blocks be equal to sector size will allow PL files to have allocation units equal to their parent files. However, since PL files will be maintained separately, they could also have allocation units that are independently defined. No interface will be exported for users to modify the PL file allocation unit and it will default to ADVFS_METADATA_PGSZ_IN_FOBS.

3.1.2 Flat File Layout

Property List Flat File Layout


Page 0 n(k) s.t. (n % DEV_BSIZE == 0)
PL_FILE_ HDR
PL_BLOCK
...
PL_BLOCK
PL_BLOCK = DEV_BSIZE at creation of property list file
Offset 0
Offset n(k) -1
PL_FILE_HDR
PL_FILE_LAYOUT_VERSION = FLAT
PL_START = 0x2000
PL_NEXT_FREE_BLK = 0x6000
PL_FILE_SIZE = End of last allocated storage in bytes
PL_BLOCK_SZ = {DEV_BSIZE @ creation}
64 byte padded
PL_BLOCK_HDR
pbh next free blk off - Offset of next free blk
pbh_prev_free_blk_off - Offset of prev free blk
pbh_free_chunk_off - offset of free chunk list
pbh block free bytes - free bytes on block
pbh free chunk cnt - free chunks on block
pbh block type - Type of block (DATA or INDEX)
PL_BLOCK
Offset 0x2000
PL_BLOCK_HDR
pbh_next_free_blk_off - PLC_NOT_ON_LIST
pbh_prev_free_blk_off - PLC_NOT_ON_LIST
pbh_free_chunk_off - PLC_END_OF_LIST
pbh_block_free_bytes - 0
pbh_free_chunk_cnt - 0
pbh_block_type - PL_DATA
PL_SEGMENT_HDR
pls_type - ELEM_SEG
pls_next_off,
pls_prev_off - PLC_END_OF_LIST
pls_size -
pls_element_hdr
NAME 0xX bytes
DATA 0xY bytes
PL_SEGMENT_HDR
pls_type - ELEM_SEG
pls_next_off,
pls_prev_off - PLC_END_OF_LIST
pls_size -
pls_element_hdr
NAME 1 0xX1 bytes
DATA 1 0xY1 bytes
PL_BLOCK
Offset 0x4000
PL_BLOCK_HDR
pbh_next_free_blk_off - PLC_NOT_ON_LIST
pbh_prev_free_blk_off - PLC_NOT_ON_LIST
pbh_free_chunk_off - PLC_END_OF_LIST
pbh_block_free_bytes - 0
pbh_free_chunk_cnt - 0
pbh_block_type - PL_DATA
PL_SEGMENT_HDR
pls_type - ELEM_SEG
pls_next_off - cont_off
pls_prev_off - PLC_END_OF_LIST
pls_size -
pls_element_hdr
NAME 2 0xX2 - n bytes
PL_BLOCK
Offset 0x6000
PL_BLOCK_HDR
pbh_next_free_blk_off - PLC_END_OF_LIST
pbh_prev_free_blk_off - PLC_END_OF_LIST
pbh_free_chunk_off - free_off
pbh_block_free_bytes -
pbh_free_chunk_cnt - 1
pbh_block_type - PL_DATA
PL_SEGMENT_HDR
pls_type - CONT_SEG
pls_next_off - PLC_END_OF_LIST
pls_prev_off - PLC_END_OF_LIST
pls_size -
pls_element_hdr
NAME 2 (continuted) n bytes
DATA 2 0xY2 bytes
PL_SEGMENT_HDR
pls_type - FREE_SEG
pls_next_off - PLC_END_OF_LIST
pls_prev_off - PLC_END_OF_LIST
pls_size -
pls_element_hdr
cont_off
Offset free_off = 0x6000 + sizeof(PL_BLOCK_HDR + n + 0xY2

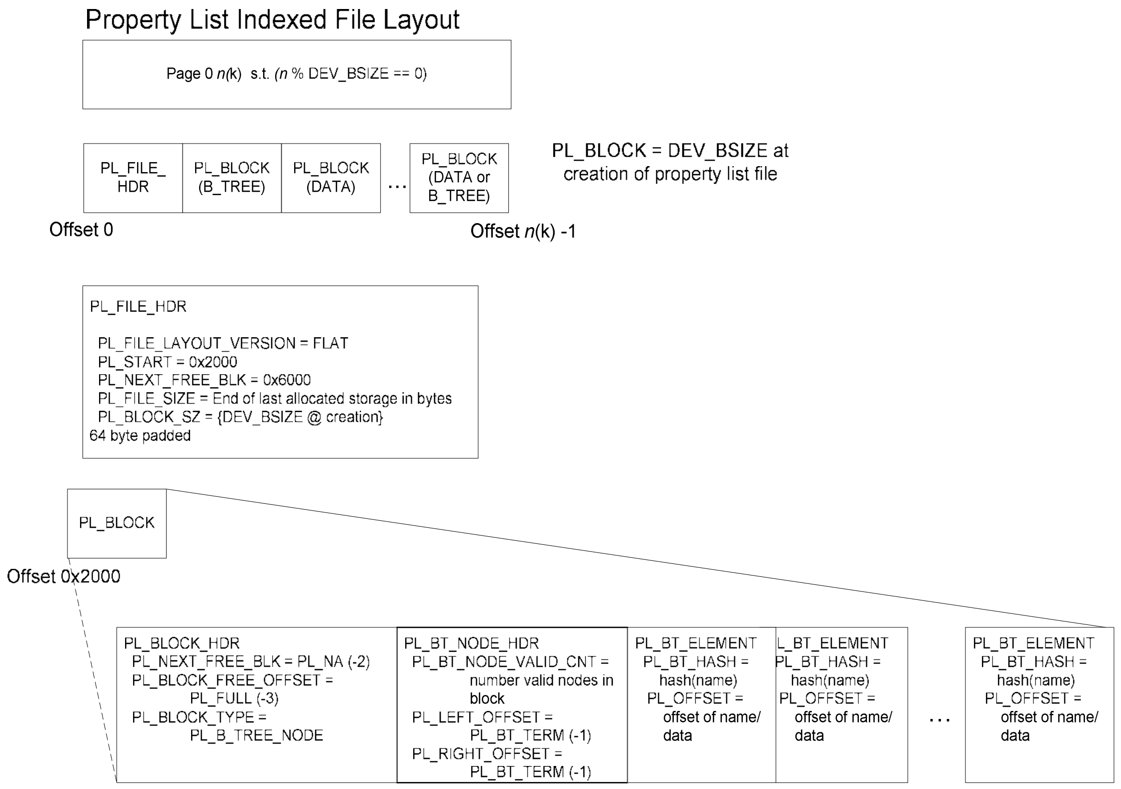
The flat file layout for the PL file will provide a simple implementation and be useful for debugging. Providing a flat file implementation will reduce the risk that PLs will miss the target schedule for functionality while providing the benefit of removing some complex on disk manipulations and making debugging in-memory extent and locking issues easier.

The flat file layout will provide linear time searches and linear time insertions. As a result, this layout is not suitable for large numbers of PLs and ACLs. For small numbers of PLs, this solution may provide some slight performance benefit over the indexed PL files. Assuming that both the B-Tree node and the PL block containing the data exist on the first page of an indexed PL file, the difference for small PL files would be minor but in favor of the flat file. For medium sized PLs, the advantage may fall either towards the flat file or the B-tree depending on the example. For large numbers of PLs, the indexed solution would provide better performance.

The basic layout of the flat PL file will be a list of PL elements. Each element of the list in a flat file contains a pl_segment_hdr_t (of type PLST_ELEM_HDR) and a variable length name and data segment. For any given PL element, any number of continuations may exist if the PL element was unable to fit in a single PL block and required spanning to another block. When a PL element is unable to fit in a given free chunk, the element will have it's pl_segment_hdr_t structure's pls_cont_off field contain the offset in the PL file that contains another pl_segment_hdr_t (of type PLST_ELEM_CONT) structure. The pl_segment_hdr_t structure will serve as a header for more of the PL element's data but may point to another pl_segment_hdr_t structure (ie a chain of pl_segment_hdr_t PLST_ELEM_CONT types may exist for a given PL element). The chaining of pl_segment_hdr_t structures will continue until enough free space to accommodate all of the name and data portions of the PL element is acquired. PL element data will be chained using a first available algorithm for free space unless the PL element size exceeds ADVFS_PL_FULL_BLOCK_THRESHOLD in which case the PL element will only be allowed to occupy full PL blocks.

PL elements will terminate their continuations by defining a pls_cont_off as PLC_END_OF_LIST (-0x2). The list of PL elements (chained together through the pls_next_offset field of the pl_segment_hdr_t) will also terminate with an offset of PLC_END_OF_LIST.

3.1.3 B-Tree Layout


Property List Indexed File Layout
Page 0 n(k) s.t. (n % DEV_BSIZE == 0)
PL_FILE_HDR
PL_BLOCK (B_TREE)
PL_BLOCK (DATA)
...
PL_BLOCK (DATA or B_TREE)
PL_BLOCK = DEV_BSIZE at creation of property list file
Offset 0
Offset n(k) -1
PL_FILE_HDR
PL_FILE_LAYOUT_VERSION = FLAT
PL_START = 0x2000
PL_NEXT_FREE_BLK = 0x6000
PL_FILE_SIZE = End of last allocated storage in bytes
PL_BLOCK_SZ = {DEV_BSIZE @ creation}
64 byte padded
PL_BLOCK
Offset 0x2000
PL_BLOCK_HDR
PL_NEXT_FREE_BLK = PL_NA (-2)
PL_BLOCK_FREE_OFFSET = PL_FULL (-3)
PL_BLOCK_TYPE = PL_B_TREE_NODE
PL_BT_NODE_HDR
PL_BT_NODE_VALID_CNT = number valid nodes in block
PL_LEFT_OFFSET = PL_BT_TERM (-1)
PL_RIGHT_OFFSET = PL_BT_TERM (-1)
PL_BT_ELEMENT
PL_BT_HASH = hash(name)
PL_OFFSET = offset of name/ data
L_BT_ELEMENT
PL_BT_HASH = hash(name)
PL_OFFSET = offset of name/ data
...
PL_BT_ELEMENT
PL_BT_HASH = hash(name)
PL_OFFSET = offset of name/ data

The indexed PL file will contain a mix of PL B-tree nodes and data PL blocks. The PL blocks representing data will be organized similar to the flat file, but will not chain from one PL element to the next. A single PL Entry will chain continuations of the data exactly like the flat PL file and the free list will be maintained in the same manor.

The largest differentiator between indexed and flat PL files is the lookup. While flat files start from the PL block at offset pl_file_hdr.pl_start (always 0x{*sector size*} for flat files) and walk through each PL element looking for a match on the name to be looked up, the indexed B-Trees start at the B-Tree node pointed to by pl_file_hdr.pl_index_start and find the leaf node which contains the hash value for the name to be looked up. The B-Tree element containing the hash value provides an offset for the PL element. The offset found in the B-Tree element is read and the PL element's name is compared to the name to be looked up. If the name does not match, there was a collision and the next B-Tree element is examined (much like the directory index code).

The indexed PL file searches are logarithmic and therefore will perform significantly better for large PLs.

Because B-Trees are more complex to implement, the B-tree implementation will be done after the flat file implementation. As an aspiration goal, the B-tree implementation will be completed prior to releasing AdvFS to any PL consumers, however, if necessary, the flat file can be used for functionality and the B-tree implementation will be provided later for better performance. This design describes the B-Tree design at a very high level and leaves room for its implementation later. It is believed that moving between an indexed and non-indexed PL file should be supported.

3.2 Module Design

3.2.1 PL API – advfs_proplist.c & advfs_proplist.h

3.2.1.1 advfs_pl_set

3.2.1.1.1 Interface

```
statusT
advfs_pl_set(struct vnode *   vp,             /* Parent file of PL file */
             struct uio*     uiop,           /* UIO structure for PL to be set. The is
                                              * only 1 iovec, but the list may contain
                                              * any number of PL elements, one after
                                              * the other.  */
             struct ucred*  cred            /* Credentials of caller */
             )
```

3.2.1.1.2 Description

This routine is the highest level interface for setting one or more PL elements in a PL. The interface takes a vnode and a uio structure pointer and uses this to set the PL elements specified in the uio structure. The vnode passed in must be the vnode of the parent file of the PL file. If a user were to do a .tags lookup of a PL file, the user cannot attempt to use that vnode to set PL elements.

The uio structure may have multiple PL elements and each will be set in series. If a uio structure has multiple elements, the second structure must start immediately after the first element ends.

This is fairly high level because the interface is not completely defined. Any translation between the external API and internal API will happen in this routine. The “struct pl_user_element_hdr” is a place holder for an external property list header that is presumed necessary. A generic interface for property lists is being developed and will influence the final version of this routine.

This routine will interpret the setting of a PL element with a 0 length data field as a request to delete the PL element.

3.2.1.1.3 Execution Flow

```
Check for permission to set proplists.
Make sure the vnode is the bfVnode of the bfap it is associated with and not a pl_vp.
        If a pl_vp, return E_BAD_HANDLE

cur_resid = uio->resid

/* Loop through all the passed in PL elements and insert them one at a time into the
 * PL file.  */
while (cur_resid > 0)
        /* copyin will advance the uiop->offset. */
        cur_pl_element_hdr = copyin(sizeof(struct pl_user_element_hdr), uiop)

        /*
         * We need the name to do lookups and hash calculations, but we will want to
         * get the start of the uio structures for the insert.  So we will do a copyin,
         * then reset the uio->offset value since copyin will advance the count */
        cur_offset = uiop->offset
        char * name = copyin(cur_pl_element_hdr->namelen, uiop)
        uiop->offset = cur_offset
        next_offset = cur_offset + cur_pl_element_hdr->namelen +
                                     cur_pl_element_hdr->datalen
        pl_data->pld_element.plie_namelen = cur_pl_element_hdr->namelen
        pl_data->pld_element.plie_name = name
        pl_data->pld_uio = uio
        if (cur_pl_element_hdr->datalen)
                pl_flags = PLF_INSERT
                sts = advfs_pl_insert( bfap, pl_data, ftxNilFtx )
        else
                pl_flags = PLF_DELETE
                sts = advfs_pl_delete( bfap, pl_data, pl_flags, ftxNilFtx)
        if (sts != EOK)
                return sts
        cur_resid -= next_offset – cur_offset

free name
return EOK
```

3.2.1.1.4 Exceptions

Could return any status from advfs_pl_insert or E_PERM for permissions errors.

3.2.1.2 advfs_pl_get

3.2.1.2.1 Interface

```
statusT
advfs_pl_get(struct vnode *   vp,             /* Parent file of PL file */
             char *name,                      /* Name of element to lookup */
             struct uio*      uiop,           /* UIO structure for PL to be set. The is
                                               * only 1 iovec, but the list may contain
                                               * any number of PL elements, one after
                                               * the other.  */
             struct ucred*    cred            /* Credentials of caller */
             )
```

3.2.1.2.2 Description

This routine is the highest level interface for PL lookups. It takes a vnode for the parent file whose PL is to be searched, and a uio vector to hold the PL element that is found. As with advfs_pl_set, this routine cannot be passed the vnode of the PL file itself.

The basic flow of control is to offset the uio by the sizeof the user space header, then to do a lookup that will copyout the name and data. After the lookup, the user header will be initialized with the sizes of the name and data buffers and copied out. It is assumed that there is only one iovec for the uio structure.

This is fairly high level because the interface is not completely defined. Any translation between the external API and internal API will happen in this routine. The “struct pl_user_element_hdr” is a place holder for an external property list header that is presumed necessary. A generic interface for property lists is being developed and will influence the final version of this routine.

3.2.1.2.3 Execution Flow

```
Check for permission to set proplists.
Make sure the vnode is the bfVnode of the bfap it is associated with and not a pl_vp.
        If a pl_vp, return E_BAD_HANDLE

start_offset = uio->offset
init_resid = uio->resid
cur_offset = start_offset + sizeof(pl_user_element_hdr)
cur_resid = init_resid – sizeof(pl_user_element_hdr)

pl_data->name = name
pl_data->namelen = strlen(name)
pl_data->pld_uiop = uio
pl_data->pld_cur_uio_off = cur_offset
pl_data->pld_cur_uio_resid = cur_resid
pl_flags = PLF_LOOKUP

sts = advfs_pl_lookup( bfap,
                      pl_data,
                      pl_flags )
```

```
if (sts != EOK)
        return sts

pl_user_element_hdr->namelen = pl_data->namelen
pl_user_element_hdr->datalen = pl_data->datalen

cur_offset = uio->offset
uio->offset = start_offset
copyout(pl_user_element_hdr, uio)
uio->offset = cur_offset
uio->resid = cur_resid

return EOK
```

3.2.1.2.4 Exceptions

EIO and E_NOT_FOUND can be returned from this routine.

3.2.1.3 advfs_pl_lookup

3.2.1.3.1 Interface

```
statusT
advfs_pl_lookup(bfAccessT*    bfap,          /* Parent file of PL file */
                pl_data_t*    pl_data,       /* Comes in with Element name and lookup
                                              * hints.  Different fields initialized for
                                              * different callers */
                pl_flags_t*   pl_flags       /* Flags contain hints and return status */
               )
```

3.2.1.3.2 Description

This function is a centerpiece for all operations on the PL file. advfs_pl_lookup will be called from both advfs_pl_insert and advfs_pl_delete to find the location of the PL element to modify/insert or remove. Additionally, the routine will be called when a read of PL element is requested. If called from insert or delete, the pl_file_lock should be held for write before calling into the routine. If called for a lookup, the pl_file_lock will be acquired in read mode. If no PL file exists, then the pl_vp in the bfap will be NULL and EOK will be returned with PLF_NOT_FOUND set in the pl_flags parameter.

Internally, advfs_pl_lookup will always have the bfap->pl_file_lock for either read or write. When (pl_flags & PLF_LOOKUP), the lock will be taken for read access and will be held while the PL file is searched. The pl_file_lock will be held across potentially multiple IOs. In the PLF_LOOKUP case, the fl_file_lock will be dropped before returning to the caller. If the lookup did not find an element, then the pl_flags field PLF_NOT_FOUND will be set on return. The return status will be EOK or will reflect an error from lower levels but will not indicate anything about the existence of the element in the PL file.

When advfs_pl_lookup is called from advfs_pl_insert, (pl_flags & PLF_INSERT) will be TRUE. In this case, advfs_pl_insert will take the pl_file_lock for write access before calling advfs_pl_lookup. On insert, the pl_data structures pl_data.pld_element.pl_name will contain the name of the element to be inserted. advfs_pl_lookup will return with the pl_file_lock held whether or not the PL element is found.

When advfs_pl_lookup is called from advfs_pl_delete, (pl_flags & PLF_DELETE) will be TRUE. advfs_pl_delete will setup the pl_data.pl_element.pl_name field to specify which PL Entry should be searched for. As with the PLF_INSERT case, advfs_pl_delete will acquire the pl_file_list lock for write mode before calling advfs_pl_lookup. If the PL element is found, advfs_pl_lookup will setup the pld_offset field of the pl_data structure and will initialize the pl_data.pld_element.plie_namelen and plie_datalen fields from the header of the found PL element. If the requested element is not found, the routine will return a status of EOK and will set the PLF_NOT_FOUND flag in the pl_flags parameter.

The flow of control in advfs_pl_lookup will be the same for both PL file layout types (indexed and flat) up until the point that the search actually occurs. All locking and error checks will be done in advfs_pl_lookup prior to layout specific search calls.

Before calling any advfs_pl_* routines, the parent bfap must be accessed and referenced. The pl_vp will be initialized in the bfap when the file is first accessed (in bs_map_bf), so on entrance to advfs_pl_lookup, it is expected that pl_vp is non-NULL if the PL file exists.

3.2.1.3.3 Execution Flow

```
/* Handle some error checking and getting the pl_file_lock if necessary.  Also, create
 * the PL file if necessary.  */
if ( (bfap->pl_vp == NULL) && ((PLF_INSERT) || (PLF_DELETE)
        /* We are trying to do an insert or a delete and there is no PL file.
         * therefore, there is nothing to lookup.  Return EOK. */
        *pl_flags |= PLF_NOT_FOUND
        return EOK
else
        ASSERT( pl_flags & PLF_LOOKUP)
        read_lock bfap->pl_file_lock
        if (bfap->pl_vp == NULL)
                unlock pl_file_lock
                *pl_flags |= PLF_NOT_FOUND
                return EOK



switch bfap->pl_layout
case PLV_FLAT:
        /*
         * This call will search the flat PL file element chain and
         * set PLF_FOUND or PLF_NOT_FOUND accordingly.  If PLF_FOUND,
         * then the pl_data->pld_offset field will also be set to
         * indicate where the found element is located in the file.
         */
        sts = advfs_pl_flat_lookup(bfap, pl_data, pl_flags)
        if sts != EOK
                unlock bfap->pl_file_lock
                return sts
        break;
case PLV_B_TREE:
        /*
         * This call will not count free space file wide but will setup the pl_data
         * fields for insert if there is enough space on the PL block that the PL element
         * is found on (if found).
         */
        sts = advfs_pl_index_lookup(bfap, pl_data, pl_flags)
        if sts != EOK
                unlock bfap->pl_file_lock
                return sts
        break

/*
 * At this point, pl_flags has PLF_FOUND or PLF_NOT_FOUND set and on an insert
 * indicate whether enough storage was found if the PLF_STG_AVAILBLE flag is set
 */

if (pl_flags & PLF_NOT_FOUND)
        if (pl_flags & PLF_LOOKUP)
                /* In the lookup case, there is nothing to do but unlock*/
                unlock pl_file_lock
        else /* Insert or delete case, return with lock held for write */
                ASSERT( pl_flags & PLF_INSERT || PLF_DELETE)
else if (pl_flags & PLF_FOUND)
        if (pl_flags & PLF_LOOKUP)
                /* If we are doing a lookup, then we need to do the uio
                 * move to the uio structure in the pl_data.  */
                advfs_pl_uiomove_read(bfap, pl_data, pl_flags)
                /* In lookup case, we can always just unlock */
                unlock bfap->pl_file_lock
        else /* If we found it for insert or delete, keep the lock */
                /* The insert or delete will handle the uio move. */
                ASSERT( (pl_flags & PLF_INSERT) || (pl_flags & PLF_DELETE) )


return EOK
```

3.2.1.3.4 *Exceptions*

advfs_pl_lookup may encounter an IO error searching the PL file and will return EIO.

3.2.1.4 advfs_pl_insert

3.2.1.4.1 *Interface*

```
statusT
advfs_pl_insert(bfAccessT* bfap,
                pl_data_t* pl_data,
                pl_flags_t * pl_flags,
                ftxHT parent_ftx
               )
```

3.2.1.4.2 *Description*

The insert function of the PL API allows the caller to insert a new PL element or modify an existing PL element. This design assumes that insertions and deletes occur significantly less frequent than lookups. Because inserts use a first available algorithm for free space since, and because a write lock is required for inserts and deletes, frequent insertions and deletions may cause fragmentation and performance degradation. Additionally, because inserts must be atomic, large insertions may have a significant impact on the log. To help offset this problem when modifications are being made to PL elements, data is not logged on deletions, only header changes. As a result, a modification is treated as an insert of a new element followed by a deletion of the old element. This causes free space to exist in the PL file that is equal to the size of the old PL element. However, the log does not need to record the changes to the deleted data.

advfs_pl_insert begins by doing basic access checks to make sure the file system is not read only and that the user doing the insert has the permissions to modify the file. Next, the routine will try to access the PL file if it is not already initialized. The routine will pass the PLF_CREATE flag to advfs_pl_access to create the file if it does not already exist. A lookup is then performed on the PL file to see if the PL element being inserted already exists. If the element does exist, it will need to be deleted after we insert the new element (this is a modification of a PL element). To delete the element later, the offset of the element to be deleted is stored away and will be used to tell advfs_pl_delete which element to remove. Next, the file header is checked to see if a sufficient amount of free space exists to insert the new element. A heuristic calculation is used to determine if additional storage will be required. If storage will be required, the storage is added and initialized in a transaction that is separate from the transaction protecting the actual insert. The storage addition transaction and insert transaction will be separate root transactions; however, in the case of ACLs, a parent transaction is passed in that will tie the two together. For the general case, the two transactions are separated to reduce the amount of stress on the log and increase the maximum PL element size. In the case of ACLs, the maximum size is limited, so the two transactions can safely occur under the same root transaction.

When storage is added, the storage will be deducted from the quota of the owner of the file. After the storage is added, the pages will be initialized via a call to advfs_pl_init_range. The newly added storage will be the first storage to be used when doing the insert.

Once storage is added, the PL element is inserted via a call to advfs_pl_insert_int. If the PL file is indexed, the index is also updated to point to the new element. Finally, if the insert is a modification, the original PL element is deleted via a call to advfs_pl_delete. At the end of the routine, the pl_file_hdr is updated to reflect the number of bytes that were consumed by the insert and the pl file is closed.

advfs_pl_insert expects the pld_element structure of the pl_data parameter to be initialized with a non-zero length name and a non-NULL name buffer. The data length can be 0 if the data buffer is NULL. This will allow internal PL elements with 0 length data, but advfs_pl_set will screen out any external callers trying to set PL elements with 0 length data.

Because all data in a insert must be put into the log so that the insert can be redone if necessary, large inserts may put a substantial strain on the log. To help protect the log against large property lists, this routine will only allow PL_BIG_INSERT_MAX "large" inserts to occur simultaneously. A new cv in the domainT structure will be used to block multiple inserts from occurring at the same time when the size exceeds the PL_EXCLUSIVE_INSERT_THRESHOLD.

3.2.1.4.3 Execution Flow

```
if ((pl_data->pld_element.plie_name == NULL ||
     (pl_data->pld_element.plie_namelen == 0) || (
     ( (pl_data->pld_element.plie_data == NULL) &&
        (pl_data->pld_element.plie_datalen != 0) )
        return E_BAD_PARAMS

if (bfap->bfSetp->fsnp->vfsp->vfs_flags & VFS_RDONLY)
        return E_ROFS

/* Do access checks to make sure the user has perms to modify/write PL list. */
if (!access_allowed)
        return E_PERM

pl_flags = PLF_INSERT

if (pl_data->plie_namelen + plie_datalen) > ADVFS_PL_MAX_ELEMENT_SIZE
        return E_TOO_BIG

if (pl_data->plie_namelen + plie_datalen) > ADVFS_PL_FULL_BLOCK_THRESHOLD
        full_blocks_required = TRUE
else full_blocks_required = FALSE

/*
 * Check if this is a "big" insert that needs to block others.
 */
if (pld_element.plie_namelen + pld_element.plie_datalen > PL_EXCLUSIVE_INSERT_THRESHOLD)
        mutex_lock(bfap->dmnP->pl_cv_mutex)
        while (bfap->dmnP->pl_big_insert_cnt >= PL_BIG_INSERT_MAX)
                cv_wait(bfap->dmnP->pl_cv)
        bfap->dmnP->pl_big_insert_cnt++
        mutex_unlock(bfap->dmnP->pl_cv_mutex)
        is_big_insert = TRUE

/* We will be doing an insertion, so we will need the pl_file_lock in
 * write mode.  */
write_lock( bfap->pl_file_lock )

/*
 * pl_vp would have been setup at open of the bfap if it existed.  So we will
 * create it here before we continue.
 */
if (bfap->pl_vp == NULL)
        sts = advfs_pl_access(bfap, PLF_CREATE)

        /*
         * We failed to create the PL file, so return the error.
         */
        if (sts != EOK)
                unlock pl_file_lock
                return sts
```

```
/* Passed initial checks, now start the insert */
sts = advfs_pl_lookup( bfap, pl_data, &pl_flags )
if (sts != EOK)
        if (is_big_insert)
                mutex_lock(bfap->dmnP->pl_cv_mutex)
                bfap->dmnP->pl_big_insert_cnt—
                cv_signal(bfap->dmnP->pl_cv)
                mutex_unlock(bfap->dmnP->pl_cv_mutex)
        return sts

pl_element_size = pl_data.pld_element.plie_namelen +
                        pl_data.pld_element.plie_datalen;
pl_max_blks_required = pl_element_size / (pl_block_size – sizeof(pl_block_hdr_t))

if (pl_element_size > bfap->pl_free_space ||
        (full_blocks_required && pl_blocks_free < pl_max_blks_required)

        /* Storage was required. We will add enough storage for this
         *entire PL element even if we found an existing one that will be removed.  */
        /* This transaction will cover the addition and initialization of the new storage
         * and linking it into the free list.  The transaction will be ftx_done'd prior to
         * starting a transaction for the insertion. */
        FTX_START_N(FTA_PL_INSERT_ADD_STG, &ftxH, FtxNilFtxH, bfap->dmnP)

        /* First update/check quotas */
        fobs_to_add = roundup(pl_element_size, bfap->pl_alloc_size) / ADVFS_FOB_SZ
        if (error = chk_blk_quota(&bfap->bfVnode,
                                (int64_t) QUOTABLKS(fobs_to_add * ADVFS_FOB_SZ),
                                pl_data->pld_cred, 0, ftxH )) {
                /*
                 * Couldn't deduct quotas for this user by fobs_to_add.  We need to fail
                 * the transaction.  This could be weird for a user since he won't
                 * be able to modify a PL if he is near is quota maximum.  It's sort
                 * of like trying to do something that requires a tmp file, so it
                 * should be ok.
                 */
                if (is_big_insert)
                        mutex_lock(bfap->dmnP->pl_cv_mutex)
                        bfap->dmnP->pl_big_insert_cnt—
                        cv_signal(bfap->dmnP->pl_cv)
                        mutex_unlock(bfap->dmnP->pl_cv_mutex)

                ftx_fail(ftxH)
                unlock (bfap->pl_file_lock )
                return error
         }


        /*
         * Add storage for PL element
         */
        sts = stg_add_stg(ftxH,
                        bfap,
                        bfap->pl_file_size / ADVFS_FOB_SZ,
                        roundup(pl_element_size, ADVFS_FOB_SZ),
                        NO_OVERLAP | PL_ADD_STG,
                        &actual_storage_fobs);
        if (sts != EOK)
                ftx_fail( ftxH )
                if (is_big_insert)
                        mutex_lock(bfap->dmnP->pl_cv_mutex)
                        bfap->dmnP->pl_big_insert_cnt—
                        cv_signal(bfap->dmnP->pl_cv)
                        mutex_unlock(bfap->dmnP->pl_cv_mutex)
                unlock (bfap->pl_file_lock )

                return sts
        /* We haven't updated the bfap->pl_file_size yet!  Advfs_pl_init_range will init
         * the new storage as free pl blocks and link the storage as free storage pointing
         * to the head of the current free list. advfs_pl_insert will actually update the
         * the head pointer to point to this new storage.  We need to update the
```

```
 * file_size so getpage will read it when we fault. */
old_size = bfap->pl_file_size
bfap->pl_file_size += actual_storage_fobs * ADVFS_FOB_SZ;
sts = advfs_pl_init_range(bfap,
                          old_size,
                          actual_storage_fobs * ADVFS_FOB_SZ,
                          ftxH)
if (sts != EOK)
        bfap->pl_file_size = old_size
        /* This is a problem since our storage was added but we didn't initialize
         * it.  We must fail the transaction to undo the storage add. */
        ftx_fail( ftxH )
        if (is_big_insert)
                mutex_lock(bfap->dmnP->pl_cv_mutex)
                bfap->dmnP->pl_big_insert_cnt—
                cv_signal(bfap->dmnP->pl_cv)
                mutex_unlock(bfap->dmnP->pl_cv_mutex)

        advfs_pl_close( bfap )
        unlock (bfap->pl_file_lock )
        return sts

/* We've added the storage to our file, now update the file size and check to see
 * if we got enough storage to insert the new PL element. We need to pin page 0
 * to update the on disk file size too. */

sts = rbf_pinpg( page_hdl,
                 &pl_file_hdr_p
                 bfap,
                 0,    /* Page 0 */
                 BS_NIL,
                 ftxH,
                 MF_VERIFY_PAGE| MF_PL_FILE)
if (sts != EOK)
        ftx_fail( ftxH )
        bfap->pl_file_size = old_size
        if (is_big_insert)
                mutex_lock(bfap->dmnP->pl_cv_mutex)
                bfap->dmnP->pl_big_insert_cnt—
                cv_signal(bfap->dmnP->pl_cv)
                mutex_unlock(bfap->dmnP->pl_cv_mutex)

        return sts

undo_rec = malloc(sizeof(pl_generic_hdr_undo_t) + sizeof(pl_ins_del_undo_t) )
/* The undo record will just be for the file hdr, so it's a hdr followed by one
 * repeating record */
undo_rec.plgu_multi_cnt = 1
undo_rec.plgu_bf_set_tag = bfap->bfSetId
undo_rec.plgu_bf_tag = bfap->tag
undo_rec.plidu_type = PL_DEL_FILE_HDR_UNDO
undo_rec.plidu_offset = 0
undo_rec.plidu_u.plidu_prev_file_hdr = pl_file_hdr_p

/*
 * First link the new storage to the head of the free list
 */
rbf_pin_record ( page_hdl,
                 &pl_file_hdr_p,
                 sizeof(pl_file_hdr_t) )
pl_file_hdr_p->pdh_next_free_blk_off = old_file_size
new_blocks = (actual_storage_fobs * ADVFS_FOB_SZ)
                                    / bfap->pl_block_size
pl_file_hdr_p->pdh_free_blk_cnt += new_blocks
pl_file_hdr_p->free_byte_cnt += (pl_block_size –sizeof(pl_block_hdr_t)) *
                                    new_blocks
pl_file_hdr_p->pdh_file_size = bfap->pl_file_size;

update bfap pl fields to match file header files.
```

```
        /* Can't fail this transaction now! */
        ftx_done_u( ftxH,
                    FTA_PL_INSERT_ADD_STG,
                    undo_rec, sizeof(undo_rec))


/* Start a transaction to handle the insert.  This transaction may have subtransactions
 * for a removal, and an update to the index if the file is indexed.
 */
FTX_START_N(FTA_PL_INSERT_PL_ELEMENT, &ftxH, FtxNilFtxH, bfap->dmnP)
/*
 * We can't remove the element yet because we will overflow the transaction log on too-
 * large of writes.  Additionally, we would not be able to do the insert/remove
 * atomically if we do the remove first since we would have to put the storage back
 * on the free list and the new element would risk over writing it.  The log is
 * not suitable for logging all data to be removed.
 */
delete_old_element = FALSE
if (pl_flags & PLF_FOUND)
        delete_old_element = TRUE
        pl_element_to_del_offset = pl_data->pld_offset

/*
 * The element does not exist in the PL file at this time.  We either removed it or it
 * wasn't found.  Either way, the insert is the same from here on out. We call
 * advfs_pl_insert_int to do the actual data insertion and return to use the amount
 * of space consumed (including overhead) and the number of chunks it used in addition to
 * the number of clean/new blocks consumed. If we are in an indexed PL file, we will
 * update the b-tree with the new PL element.  We will then update the counts in the file
 * header.
 */
sts = advfs_pl_insert_int( bfap,
                           pl_data,
                           pl_flags,
                           ftxH,
                           &clean_pl_blocks_used,
                           &free_bytes_consumed,
                           &bytes_wasted )

if (sts != EOK)
        ftx_fail( ftxH )
        unlock ( bfap->pl_file_lock )

/*
 * Add the new PL element to the index.  This shouldn't need to add storage
 * since we got enough storage to grow the b-tree above
 */
if (bfap->pl_layout == PLV_B_TREE)
        sts = advfs_pl_index_add_element( bfap,
                                    pl_data,
                                    ftxH,
                                    &clean_index_blocks_used,
                                    &free_index_bytes_used)
        if (sts != EOK)
                ftx_fail( ftxH )
                unlock ( bfap->pl_file_lock )
        clean_pl_blocks_used += clean_index_blocks_used
        free_bytes_consumed += free_index_bytes_used


/*
 * Remove the old element now.  We must reset the offset in the pl_data
 * in case it got modified somehow.  advfs_pl_delete will recognize this
 * as an INSERT related operation and will remove the PL element at offset
 * rather than searching for the name and finding the new element.
 */
if (delete_old_element)
        /* pl_flags & PLF_INSERT is already set to key remove.  */
        pl_data->pld_offset = pl_element_to_del_offset
        sts = advfs_pl_delete(bfap,
                              pl_data,
                              &pl_flags,
                              ftxH)
        if (sts == E_MAX_PINP_EXCEEDED)
                /* if delete fails because it couldn't do enough transactions to remove
                 * the old element, we don't necessarily want to fail the insert.  We will
                 * instead try to transactionally mark the old PL element as invalid under
                * this root transaction.  The invalid element can be cleaned up by an
                * async thread.
                */
                /* On successfully marking the old element as invalid, sts == EOK */
                Setup a message for the advfs_handyman_thread to cleanup the element at
                the offset we tried to delete.  Send the message.  If this was a large
                insert, advfs_handyman_thread will synchronize so it doesn't try to start
                the delete until the undo is completed.

        if (sts != EOK)
                if (is_big_insert)
                        mutex_lock(bfap->dmnP->pl_cv_mutex)
                        bfap->dmnP->pl_big_insert_cnt—
                        cv_signal(bfap->dmnP->pl_cv)
                        mutex_unlock(bfap->dmnP->pl_cv_mutex)
                ftx_fail( ftxH )
                unlock ( bfap->pl_file_lock )

                return sts



/*
 * Pin the first page of the PL file to update the stats.
 */
sts = rbf_pinpg( page_hdl,
                &pl_file_hdr_p
                bfap,
                0,      /* Page 0 */
                BS_NIL,
                ftxH,
                MF_VERIFY_PAGE| MF_PL_FILE)
if (sts != EOK)
        ftx_fail( ftxH )
        if (is_big_insert)
                mutex_lock(bfap->dmnP->pl_cv_mutex)
                bfap->dmnP->pl_big_insert_cnt—
                cv_signal(bfap->dmnP->pl_cv)
                mutex_unlock(bfap->dmnP->pl_cv_mutex)

        return sts

rbf_pin_record( page_hdl,
                 &pl_file_hdr_p,
                 sizeof(pl_file_hdr)
pl_file_hdr_p->pdh_free_blk_cnt -= clean_pl_blocks_used
pl_file_hdr_p->pdh_free_byte_cnt -= free_byts_consumed;
pl_file_hdr_p->pdh_wasted_byte_cnt += total_bytes_wasted
pl_file_hdr_p->pdh_element_cnt++
/* Can't fail this transaction now */
ftx_done_n( ftxH, FTA_PL_INSERT_PL_ELEMENT)

bfap->pl_free_blocks = pl_file_hdr-P->pdh_free_blk_cnt
bfap->pl_free_space = pl_file_hdr_p->pdh_free_byte_cnt
bfap->pl_element_cnt++;
ASSERT( bfap->pl_element_cnt == pl_file_hdr_p->pdh_element_cnt)

if (is_big_insert)
        mutex_lock(bfap->dmnP->pl_cv_mutex)
        bfap->dmnP->pl_big_insert_cnt—
        cv_signal(bfap->dmnP->pl_cv)
        mutex_unlock(bfap->dmnP->pl_cv_mutex)

unlock ( bfap->pl_file_lock )
return EOK;
```

3.2.1.4.4 Exceptions

advfs_pl_insert may return ENOSPC if the PL file requires additional storage and none can be acquired. EIO may be returned if an IO error occurs during the lookup or storage allocation. EROFS may be returned if this is a read only file system and EPERM may be returned if the caller does not have permissions to modify the PL file.

3.2.1.5 advfs_pl_delete

3.2.1.5.1 Interface

```
statusT
advfs_pl_delete(bfAccessT* bfap,
                pl_data_t* pl_data,
                pl_flags_t pl_flags,
                ftxHT parent_ftx
               )
```

3.2.1.5.2 Description

advfs_pl_delete will remove a PL element from the PL file and return its storage to the PL file's free list. The routine can be called either on its own to remove a PL element defined by the pl_data->pld_name field, or from advfs_pl_insert to remove an entry starting at pl_data->pld_offset after inserting an element that already existed. In the case of a call from advfs_pl_insert, a transaction handle will be passed down and the bfap->pl_file_lock will already be held in write mode.

If the function is called directly, then a lookup will be performed to initialize the pl_data with the starting offset of the PL element to be removed.

If the function is called from advfs_pl_insert, then (pl_flags & PLF_INSERT) will be TRUE and the pl_file_lock will be held in write mode. When these conditions are TRUE, the pl_data structure will already be initialized with the start offset of the PL element to be removed and no lookup will be performed.

The remove will be done in a transaction or sub transaction that is actually a series of transactions. Since there is a fixed limit of the number of pages that can be pinned in a single sub transaction, the removal will work on subsets of the PL element, finishing sub transactions and starting new ones as the limit is reached. The sub transactions will be tied together under one single transaction to insure atomicity. Because a large element may need to be removed, providing an undo for the actual data of a PL element is not feasible and would put undo strain on the log. During a removal, only PL block, and segment headers are modified and reattched to the free list. Additionally, the file header is modified to reflect the new free list makeup. The undo of a remove consists only of reconnecting the segments and removing PL blocks from free lists that they should not be on. None of the PL element itself is actually provided a redo or undo. For this reason, the remove of an element on an insert case must occur after the insert of the new element that will replace it.

The storage that is freed by the advfs_pl_delete operation will be returned to the front of the free list. Returning the storage to the front of the list is faster than adding it to the end but may cause fragmentation if frequent inserts and deletions are performed for PL elements that change size. Putting the storage at the front of the free list will help to pack the PL file, by putting smaller elements in free chunks rather than full blocks. This will improve caching of PL elements.

If the file layout is indexed, once the element is removed, the index entry will be removed and the b-tree updated under a separate transaction.

On entry, it is expected that the pl_data->pld_element.plie_namelen is non-zero and the plie_name is non-NULL and represents the PL element to be removed. The routine will check for access rights and read only file systems and return an error if the delete operation is not allowed.

3.2.1.5.3 Execution Flow

```
if ((pl_data->pld_element.plie_name == NULL ||
     (pl_data->pld_element.plie_namelen == 0)
        return E_BAD_PARAMS

if (bfap->bfSetp->fsnp->vfsp->vfs_flags & VFS_RDONLY)
        return E_ROFS

/* Do access checks to make sure the user has perms to modify/write PL list. */
if (!access_allowed)
        return E_PERM

if (!(pl_flags & PLF_INSERT)
        write_lock( bfap->pl_file_lock )

/*
 * Check to see if we even have a PL file to delete from.  If we don't, then
 * the element being deleted couldn't exist.
 */
if (bfap->pl_vp == NULL)
        if (!(pl_flags & PLF_INSERT))
                unlock (bfap->pl_file_lock )
        /* If we failed because the PL file doesn't exist for this file,
         * then the requested PL element no longer exists, thus it is deleted */
        return EOK

if (!(pl_flags & PLF_INSERT) && !(pl_flags & PLF_CLEANUP) )
        pl_flags = PLF_DELETE
        /* Passed initial checks, now start the insert */
        sts = advfs_pl_lookup( bfap, pl_data, &pl_flags )
        if (sts != EOK)
                if ( !pl_flags & PLF_INSERT)
                        unlock( pl_file_lock )
                return sts

/*
 * Deleting the element consists of marking each segment header as free.
 * Our undo records will consist of
 * the offset of each of the segment headers so that if we must undo,
 * we can walk modify the records and modify them to be in use again.
 */

FTX_START_N(FTA_PL_DELETE_PL_ELEMENT_TOP, &top_ftx, parent_ftx, bfap->dmnP)
FTX_START_N(FTA_PL_DELETE_PL_ELEMENT, &cur_ftx, top_ftx, bfap->dmnP)

pl_flags |= PLF_PL_ELEMENT_HEAD;

undo_buf = malloc (sizeof(pl_generic_hdr_undo_t) +
                        (FTX_MX_PINR * sizeof(pl_ins_del_multi_undo_t ))
undo_buf->plgu_bf_set_tag = bfap->bfSetId
undo_buf->plgu_bf_tag = bfap->tag
cur_undo = undo_buf + sizeof(pl_generic_hdr_undo_t)
undo_cnt = 0
bytes_freed = 0
full_blocks_freed = 0


for (element_offset = pl_data->pld_offset; element_offset != PLC_END_OF_LIST;;) {
        pl_pg = element_offset / (bfap->pl_alloc_unit * ADVFS_FOB_SZ)
        /*
         * Pin the page of the element segment to be marked free.  Setup the
         * undo. If this is the first free chunk on a PL block, we need to
         * update the block too.
         * We may need to start a new sub transaction here to continue.
         */
        sts = rbf_pinpg(page_hdl,
                              pg_ptr,
                              bfap,
                              pl_pg,
                              BS_NIL,
                    cur_ftx,
                    MF_VERIFY | MF_PL_FILE)

if (sts == E_MX_PINP_EXCEEDED || !rbf_can_pin_record(page_hdl) )
        undo_buf->plgu_multi_cnt = undo_cnt
        ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                    undo_buf,
                    sizeof(pl_generic_hdr_undo_t) +
                            undo_cnt * sizeof(pl_ins_del_multi_undo_t) )
        undo_cnt = 0
        cur_undo = undo_buf + sizeof(pl_generic_hdr_undo_t)
        FTX_START_N(FTA_PL_DELETE_PL_ELEMENT, &cur_ftx, top_ftx, bfap->dmnP)

        sts = rbf_pinpg(page_hdl,
                    pg_ptr,
                    bfap,
                    pl_pg,
                    BS_NIL,
                    cur_ftx,
                    MF_VERIFY | MF_PL_FILE)
if (sts != EOK)
        if (undo_cnt)
                ftx_fail(cur_ftx)
        else
                ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                    undo_buf,
                    sizeof(pl_generic_hdr_undo_t) +
                            undo_cnt * sizeof(pl_ins_del_multi_undo_t) )
        ftx_fail( top_ftx )
        if (!PLF_INSERT)
                unlock( pl_file_lock )
        return sts

In the future, here is where we would check for free space that follows the PL
element segment.  If the free space is
There, extend the size of the pl segment being deleted and remove the free segment
from the free list.

pl_block_offset = rounddown(element_offset, bfap->pl_block_size)
pl_block_ptr = pg_ptr + (pl_block_offset –
                        (pl_pg * bfap->pl_alloc_size * ADVFS_FOB_SZ))
pl_segment_hdr = pl_block_ptr + (element_offset – pl_block_offset)

cur_undo->plidu_offset = pl_block_offset
cur_undo->plidu_type = PL_DEL_HDR_UNDO
/* structure copy the old header */
cur_undo->plidu_u.plidu_prev_block_hdr = *pl_block_ptr
undo_cnt++
previous_free_chunk_off = pl_block_ptr->pls_free_chunk_off
rbf_pin_record( pg_hdl,
                pl_block_ptr,
                sizeof( pl_block_hdr_t ))
pl_block_ptr->pls_free_chunk_off = element_offset;
bytes_freed += pl_segment_ptr->pls_size
if (pl_segment_ptr->pls_size == bfap->pl_block_size – sizeof(pl_segment_hdr_t))
        full_blocks_freed++


if (pl_block_ptr->pbh_next_free_blk_off == PLC_NOT_ON_LIST)
        /* We already pinned the entire pl_block header, so we can
         * just modify it, but we also need to modify the next elements
         * back pointer and the file hdr’s next pointer.  Those may
         * require pinning of additional pages.
         */
        pl_block_ptr->pbh_next_free_block_off = bfap->pl_free_blk_off
        pl_block_ptr->pbh_prev_free_block_off = PLC_END_OF_LIST

        /* Pin the file hdr and modify the free list pointer */
        sts = rbf_pinpg( pl_pg_zero_hdl,
                    pg_file_hdr_ptr,
                    bfap,
                0,
                BS_NIL,
                cur_ftx,
                MF_VERIFY | MF_PL_FILE)

        if (sts == E_MX_PINP_EXCEEDED || !rbf_can_pin_record(page_hdl) )
                undo_buf->plgu_multi_cnt = undo_cnt
                ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                                undo_buf,
                                sizeof(pl_generic_hdr_undo_t) +
                                        undo_cnt * sizeof(pl_ins_del_multi_undo_t) )
                undo_cnt = 0
                cur_undo = undo_buf + sizeof(pl_generic_hdr_undo_t)
                FTX_START_N(FTA_PL_DELETE_PL_ELEMENT, &cur_ftx,
                                                top_ftx, bfap->dmnP)

                sts = rbf_pinpg(pl_pg_zero_hdl,
                                pg_file_hdr_ptr,
                                bfap,
                                0,
                                BS_NIL,
                                cur_ftx,
                                MF_VERIFY | MF_PL_FILE)
        if (sts != EOK)
                if (undo_cnt)
                        ftx_fail(cur_ftx)
                else
                        ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                                undo_buf,
                                sizeof(pl_generic_hdr_undo_t) +
                                        undo_cnt *
                                        sizeof(pl_ins_del_multi_undo_t) )
                ftx_fail( top_ftx )
                if (!PLF_INSERT)
                        unlock (pl_file_lock )
                return sts

        /* Now update the file header pointer to the new free block */
        cur_undo->plidu_offset = 0
        cur_undo->plidu_type = PL_DEL_FILE_HDR_UNDO
        /* Structure copy the entire structure */
        cur_undo->plidu_u.plidu_prev_file_hdr = *pl_file_hdr_ptr
        rbf_pin_record( pl_pg_zero_hdl,
                                pl_file_hdr_ptr,
                                sizeof(pl_file_hdr_t))
        pl_file_hdr_ptr->pfh_next_free_blk_off = pl_block_offset


        /* Pin the prev PL block hdr that was the head of the free list */
        prev_pl_blk_pg_off = rounddown(bfap->pl_free_blk_off,
                                        bfap->pl_alloc_sz * ADVFS_FOB_SZ)
        rbf_pinpg( prev_pg_hdl,
                        prev_pg_ptr,
                        bfap,
                        prev_pl_blk_pg_off /
                                (bfap->pl_alloc_sz * ADVFS_FOB_SZ),
                        BS_NIL,
                        cur_ftx,
                        MF_VERIFY | MF_PL_FILE)

        if (sts == E_MX_PINP_EXCEEDED || !rbf_can_pin_record(prev_page_hdl) )
                undo_buf->plgu_multi_cnt = undo_cnt
                ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                                undo_buf,
                                sizeof(pl_generic_hdr_undo_t) +
                                        undo_cnt * sizeof(pl_ins_del_multi_undo_t) )
                undo_cnt = 0
                cur_undo = undo_buf + sizeof(pl_generic_hdr_undo_t)
                FTX_START_N(FTA_PL_DELETE_PL_ELEMENT, &cur_ftx,
                                                top_ftx, bfap->dmnP)
                sts = rbf_pinpg(prev_pg_hdl,
                               prev_pg_ptr,
                               bfap,
                               prev_pl_blk_pg_off /
                                       (bfap->pl_alloc_sz * ADVFS_FOB_SZ),
                               BS_NIL,
                               cur_ftx,
                               MF_VERIFY | MF_PL_FILE)
        if (sts != EOK)
                if (undo_cnt)
                        ftx_fail(cur_ftx)
                else
                        ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                               undo_buf,
                               sizeof(pl_generic_hdr_undo_t) +
                                       undo_cnt *
                                       sizeof(pl_ins_del_multi_undo_t) )
                ftx_fail( top_ftx )
                if (!PLF_INSERT)
                        unlock (pl_file_lock )
                return sts

        prev_blk_ptr = prev_pg_ptr + (bfap->pl_free_blk_off - prev_pl_blk_pg_off)
        /* Now update the file header pointer to the new free block */
        cur_undo->plidu_offset = 0
        cur_undo->plidu_type = PL_DEL_HDR_UNDO
        /* Structure copy the entire structure */
        cur_undo->plidu_u.plidu_prev_file_hdr = *pl_file_hdr_ptr
        rbf_pin_record( pl_pg_zero_hdl,
                               prev_blk_ptr,
                               sizeof(pl_file_hdr_t))
        pl_file_hdr_ptr->pfh_prev_free_blk_off = pl_block_offset


        bfap->pl_free_blk_off = pl_block_offset



/* The PL block has been added back to the free list, continue processing the
 * the segment
 */
if (rbf_can_pin_record(page_hdl) )
        undo_buf->plgu_multi_cnt = undo_cnt
        ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                       undo_buf,
                       sizeof(pl_generic_hdr_undo_t) +
                               undo_cnt * sizeof(pl_ins_del_multi_undo_t) )
        undo_cnt = 0
        cur_undo = undo_buf + sizeof(pl_generic_hdr_undo_t)
        FTX_START_N(FTA_PL_DELETE_PL_ELEMENT, &cur_ftx, top_ftx, bfap->dmnP)

        sts = rbf_pinpg(page_hdl,
                       pg_ptr,
                       bfap,
                       pl_pg,
                       BS_NIL,
                       cur_ftx,
                       MF_VERIFY | MF_PL_FILE)
element_ptr = pg_ptr + (element_offset –
                       (pl_pg * bfap->pl_alloc_size * ADVFS_FOB_SZ))

/*
 * We should now be able to pin a record successfully.  Setup the undo for the
 * change we are about to make.
 */
cur_undo->plidu_offset = element_offset
cur_undo->plidu_type = PL_DEL_SEG_UNDO
/* Structure copy the old structure */
cur_undo->plidu_u->plidu_prev_seg_hdr = *element_ptr
undo_cnt++
if (pl_flags & PLF_PL_ELEMENT_HEAD)
        prev_elem_off = element_ptr->pls_prev_offset
        next_elem_off = element_ptr->pls_next_offset

rbf_pin_record( pg_hdl, element_ptr, sizeof(pl_segment_hdr_t))
element_ptr->pls_type = PLST_FREE
element_ptr->pls_prev_offset = PLC_END_OF_LIST
element_ptr->pls_next_offset = previous_free_chunk_offset

/*
 * If this is the first segment of the PL element chain,
 * we need to update the previous and next elements to remove
 * this from the chain.
 */
if (pl_flags & PLF_PL_ELEMENT_HEAD)
        /* Pin the previous and next elements to remove them from the chain
         * if necessary  */
        if (prev_elem_off != PLC_END_OF_LIST)
                prev_pg = prev_elem_off / (bfap->pl_alloc_sz * ADVFS_FOB_SZ)
                sts = rbf_pinpg( prev_pg_hdl,
                                      prev_pg_ptr,
                                      bfap,
                                      prev_pg,
                                      BS_NIL,
                                      cur_ftx,
                                      MF_VERIFY | MF_PL_FILE)
                if (sts == E_MX_PINP_EXCEEDED || !rbf_can_pin_record(page_hdl) )
                        undo_buf->plgu_multi_cnt = undo_cnt
                        ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                                      undo_buf,
                                      sizeof(pl_generic_hdr_undo_t) +
                                      undo_cnt * sizeof(pl_ins_del_multi_undo_t) )
                        undo_cnt = 0
                        cur_undo = undo_buf + sizeof(pl_generic_hdr_undo_t)
                        FTX_START_N(FTA_PL_DELETE_PL_ELEMENT,
                                &cur_ftx, top_ftx, bfap->dmnP)

                        sts = rbf_pinpg(page_hdl,
                                prev_pg_ptr,
                                bfap,
                                prev_pg,
                                BS_NIL,
                                cur_ftx,
                                MF_VERIFY | MF_PL_FILE)

                if (sts != EOK)
                        if (undo_cnt)
                                ftx_fail(cur_ftx)
                        else
                                ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                                                undo_buf,
                                                sizeof(pl_generic_hdr_undo_t) +
                                        undo_cnt * sizeof(pl_ins_del_multi_undo_t) )
                        ftx_fail( top_ftx )
                        if (!PLF_INSERT)
                                unlock (pl_file_lock )
                        return sts


                /* The prev_pg is now pinned, update the prev hdr, creating
                 * the appropriate undo */
                prev_element_ptr = prev_pg_ptr + (prev_elem_off -
                        (prev_pg * bfap->pl_alloc_size * ADVFS_FOB_SZ))

                cur_undo->plidu_offset = prev_element_offset
                cur_undo->plidu_type = PL_DEL_SEG_UNDO
                /* Structure copy the old structure */
                cur_undo->plidu_u->plidu_prev_seg_hdr = *prev_element_ptr
                undo_cnt++

                rbf_pin_record( pg_hdl, prev_element_ptr, sizeof(pl_segment_hdr_t))
```

```
                        element_ptr->pls_next_offset = next_elem_offset

                /* Now process the next element on the PL element chain.  */
                if (next_elem_off != PLC_END_OF_LIST)
                        next_pg = next_elem_off / (bfap->pl_alloc_sz * ADVFS_FOB_SZ)
                        sts = rbf_pinpg( next_pg_hdl,
                                            next_pg_ptr,
                                            bfap,
                                            next_pg,
                                            BS_NIL,
                                            cur_ftx,
                                            MF_VERIFY | MF_PL_FILE)
                        if (sts == E_MX_PINP_EXCEEDED || !rbf_can_pin_record(page_hdl) )
                                undo_buf->plgu_multi_cnt = undo_cnt
                                ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                                            undo_buf,
                                            sizeof(pl_generic_hdr_undo_t) +
                                            undo_cnt * sizeof(pl_ins_del_multi_undo_t) )
                                undo_cnt = 0
                                cur_undo = undo_buf + sizeof(pl_generic_hdr_undo_t)
                                FTX_START_N(FTA_PL_DELETE_PL_ELEMENT,
                                        &cur_ftx, top_ftx, bfap->dmnP)

                                sts = rbf_pinpg(page_hdl,
                                        next_pg_ptr,
                                        bfap,
                                        next_pg,
                                        BS_NIL,
                                        cur_ftx,
                                        MF_VERIFY | MF_PL_FILE)

                        if (sts != EOK)
                                if (undo_cnt)
                                        ftx_fail(cur_ftx)
                                else
                                        ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
                                                    undo_buf,
                                                    sizeof(pl_generic_hdr_undo_t) +
                                            undo_cnt * sizeof(pl_ins_del_multi_undo_t) )
                                ftx_fail( top_ftx )
                                if (!PLF_INSERT)
                                        unlock (pl_file_lock )
                                return sts


                        /* The next_pg is now pinned, update the next hdr, creating
                         * the appropriate undo */
                        next_element_ptr = next_pg_ptr + (next_elem_off -
                                (next_pg * bfap->pl_alloc_size * ADVFS_FOB_SZ))

                        cur_undo->plidu_offset = next_element_offset
                        cur_undo->plidu_type = PL_DEL_SEG_UNDO
                        /* Structure copy the old structure */
                        cur_undo->plidu_u.plidu_prev_seg_hdr = *prev_element_ptr
                        undo_cnt++

                        rbf_pin_record( pg_hdl, prev_element_ptr, sizeof(pl_segment_hdr_t))
                        element_ptr->pls_next_offset = prev_elem_offset

                /*
                 * We've updated the PL element chain to link around the PL element being
                 * removed.
                 */
                pl_flags &= ~PLF_PL_ELEMENT_HEAD


        /*
         * The PL element to delete should now have all of its segments marked as
         * as free and should have been removed from the list of PL elements.
         */
```

```
/*
 * We are all done setting up root done records, so finish the last cur_ftx.  The
 * index update can be done as a sub trans of the top_ftx.
 */
ftx_done_u( cur_ftx, FTA_PL_DELETE_PL_ELEMENT,
               undo_buf,
               sizeof(pl_generic_hdr_undo_t) +
                       undo_cnt * sizeof(pl_ins_del_multi_undo_t) )


/*
 * Remove the PL element from the index.
 */
if (bfap->pl_layout == PLV_B_TREE)
       pl_temp_bytes_freed = 0
       advfs_pl_index_remove_element( bfap,
                                      pl_data,
                                      top_ftx,
                                      &pl_temp_bytes_freed,
                                      &pl_temp_blocks_freed)
       pl_bytes_freed += pl_temp_bytes_freed
       if (pl_flags & PLF_FREED_PL_BLOCK)
               pl_blocks_freed += pl_temp_blocks_freed
               pl_flags &= ~ PLF_FREED_PL_BLOCK

/*
 * Update the PL file header stats.
 */
sts = rbf_pinpg( pl_pg_zero_hdl,
               pg_file_hdr_ptr,
               bfap,
               0,
               BS_NIL,
               cur_ftx,
               MF_VERIFY | MF_PL_FILE)

undo_rec = malloc(sizeof(pl_generic_hdr_undo_t) + sizeof(pl_ins_del_undo_t) )
/* The undo record will just be for the file hdr, so it’s a hdr followed by one
 * repeating record */
undo_rec.plgu_multi_cnt = 1
undo_rec.plgu_bf_set_tag = bfap->bfSetId
undo_rec.plgu_bf_tag = bfap->tag
undo_rec.plidu_type = PL_DEL_FILE_HDR_UNDO
undo_rec.plidu_offset = 0
undo_rec.plidu_u.plidu_prev_file_hdr = pl_file_hdr_p

pl_file_hdr_ptr->plfs_free_byte_cnt += bytes_freed
pl_file_hdr_ptr->plfs_free_blk_cnt += full_blocks_freed


ftx_done_n( ftxH, FTA_PL_DELETE_PL_ELEMENT_TOP)

if (!PLF_INSERT)
       unlock (pl_file_lock )
return EOK
```

3.2.1.5.4 Exceptions

advfs_pl_delete may return EIO if an IO error occurs during the lookup or storage allocation. EROFS may be returned if this is a read only file system and EPERM may be returned if the caller does not have permissions to modify the PL file.

If this routine fails because it is unable to start a transaction, it will propogate the error from FTX_START back to the caller as E_MAX_PINP_EXCEEDED. If the caller is insert, the delete will need to be done asynchronously via a message to the advfs_handyman_thread.

3.2.2 PL support routines – advfs_proplist.c & advfs_proplist.h

3.2.2.1 advfs_pl_access

3.2.2.1.1 Interface

```
statusT
advfs_pl_access(bfAccessT* bfap,
                struct vnode **pl_vpp,
                pl_flags_t pl_flags,
                mcellIDt pl_mcell_in )
```

3.2.2.1.2 Description

This routine is meant to be a mechanism for initializing the in memory extents and vnode of a PL file. If the PL file does not already exist, and pl_flags PLF_CREATE is set, then this routine will call advfs_pl_create to create the ODS structures for the PL file. If the PL file does exist, but has not yet been mapped (extent maps read into memory and the bfap PL fields initialized) then this routine will load the extent maps and initialize the bfap's pl_vp.

The parameter is a bfAccess pointer of the parent access structure of the PL file. The parent access structure will already be opened and referenced while opening the PL file.

Before calling any property list routines, the parent bfap must already be open. As a result, when calling advfs_pl_access or advfs_pl_close, the bfap is referenced, and therefore races between deallcation and recycling do not need to be considered by this routine. This routine will be called from bs_map_bf when the parent file is accessed. On returning from this routine, the bfap->pl_vp will be malloced and initialized if the PL file exists.

This routine may call advfs_pl_create to create the ODS structures for the PL file, however, to add storage, advfs_pl_create needs a vnode, so it may in turn call advfs_pl_access with the PLF_CREATING flag set. This flag indicates that the extent maps should be loaded and the vnode initialized despite the bsr_pl_file records flags indicating the PL file is invalid. The PLF_CREATING flag also indicates that the bfap->pl_file_lock is already held. In the PLF_CREATING case, pl_mcell will have the mcell of the PL file since the bsr_pl_rec will not be initialized yet.

It is assumed that the pl_file_lock is held in write mode when calling this function. If the PLF_CREATE or PLF_RELOAD_MAPS flags are set in the pl_flags parameter, then the extent maps will be loaded on return from this call.

If the PLF_RELOAD_MAPS field is set, then the routine will reload the extent maps from disk without initializing a new vnode.

3.2.2.1.3 Execution Flow

```
/* The parent bfap should already be opened and referenced at this point.  We don't
 * need to worry about accessing it. */
if (bfap->pl_vp != NULL)
        /* The PL file is already accessed.  Just return success */
        return EOK

/*
 * Read the BMT to see if we have a PL file already created.  We will look for a PL file
 * record.  If we don't find one, we will create the file.  If we do find one, then check
 * the valid field to see if the PL file was previously deleted.  If it was, create it.
 * Otherwise, we have a PL file and we can just open it.
 */
prim_mcell = bfap->primMcell
pl_mcell = nilMcell

rwlock_rdlock( &bfap->mcellList_lk )
sts = bmtr_scan_mcells( &prim_mcell,
                              &vdp,
                              &bsr_pl_rec_ptr,
                              &bmt_pg_ref,
                              &rec_pg_offset,
                              &rec_size,
                              BSR_PL_FILE_RECORD,
                              parent_tag)
if (sts == EBMTR_NOT_FOUND)
        ASSERT( !PLF_RELOAD_MAPS)
        if (!PLF_CREATE)
                return E_NO_SUCH_TAG
        ASSERT( !(pl_flags & PLF_CREATING )
        bs_derefpg( bmt_pg_ref )
        sts = advfs_pl_create( bfap, pl_mcell )
        /* advfs_pl_create will have called back in here to init the bfap */
        return sts;
else {
        /* We found the record, but we need to make sure it is valid. */
        if (bsr_pl_rec_ptr->bpr_flags & PLOF_INVALID) && (!(pl_flags & PLF_CREATING))
                ASSERT( !PLF_RELOAD_MAPS )
                /* The PL file not valid and needs to be created. */
                bs_derefpg( bmt_pg_ref )
                if (!PLF_CREATE)
                        return E_NO_SUCH_TAG
                sts = advfs_pl_create( bfap, pl_flags )
                return sts
        else
                /* We are ready to use the PL file */
                if (PLF_CREATING)
                        pl_mcell = pl_mcell_in
                        pl_alloc_sz = ADVFS_METADATA_PGSZ
                else
                        pl_mcell = bsr_pl_rec_ptr->bpr_xtnt_mcell
                        pl_alloc_sz = bsr_pl_rec_ptr-> bpr_alloc_sz

bfap->pl_alloc_sz = pl_alloc_sz

/*
 * Now just read in the xtnts and the file header info. This is like a
 * scaled down bs_map_bf.
 */
sts = imm_create_xtnt_map( dmnp, 1, pl_xtnt_maps)
if (sts != EOK)
        return sts

/* This is new for a header type.  No one that I can find has had BSR_XTRA_XTNTS as
 * the header
 */
pl_xtnt_maps->hdrType = BSR_XTRA_XTNTS
pl_xtnt_maps->hdrMcellId = pl_mcell
pl_xtnt_maps->allocVdIndex = -1


sub_xtnt_idx = 0

while(pl_mcell.volume != bsNilVdIdex)
        sts = load_from_xtra_xtnt_rec( bfap,
                                      pl_mcell,
                                      &pl_xtnt_maps->subXtntMap[sub_xtnt_idx],
                                      0, /* Not needed ??? *//
                                      &next_pl_mcell)
        if (sts != EOK)
                return sts
        pl_xtnt_maps->cnt++

Initialize the remaining pl_xtnt_maps fields.

if (!(pl_flags & PLF_CREATING)
        /* At this point, the pl_xtnt_maps are in memory.  We can now read from the PL
         * file and initialize bfap fields.  */
        sts = bs_refpg( pl_pg_ref,
                pl_pg_ptr,
                bfap,
                0,      /* read the first page of the PL file to get meta info */
                ftxNilFtx,
                MF_VERIFY_PAGE | MF_PL_FILE )
        If (sts != EOK)
                if (downgrade_lock )
                downgrade pl_file_lock to read
                return sts
        pl_file_hdr = pl_pg_ptr;
        bfap->pl_layout = pl_file_hdr->plfh_layout
        bfap->pl_block_size = pl_file_hdr->plfh_block_size
        bfap->pl_file_size = pl_file_hdr->plfh_file_size
        bfap->pl_alloc_size = pl_file_hdr->plfh_alloc_size
        bfap->pl_free_blk_off = pl_file_hdr->plfh_free_blk_off
else
        bfap->pl_layout = 0
        bfap->pl_block_size = 0
        bfap->pl_file_size = 0
        bfap->pl_alloc_size = 0
        bfap->pl_free_blk_off = 0

if (PLF_RELOAD_MAPS)
        /* We don't' want to initialize a new vnode */
        ASSERT( bfap->pl_vp )
        return sts


vnode_p = malloc( sizeof(struct vnode) )

vnode_p->v_vfsp = bfap->bfSetp->fsnp->vfsp
vnode_p->v_count = 1
vnode_p->v_scount = 0
vnode_p->v_nodeid = -1        /* Not valid ? */
vnode_p->v_flags = 0          /* We don't want soft counts for these */
vnode_p->v_shlockc = 0
vnode_p->v_exlockc = 0
vnode_p->v_tcount = 0
vnode_p->v_op = &advfs_pl_vnodeops
vnode_p->v_socket = NULL
vnode_p->v_stream = NULL
vnode_p->v_writecount = 0
vnode_p->v_type = VREG
vnode_p->v_fstype = VADVFS
vnode_p->v_vmdata = NULL
vnode_p->v_vfsmountedhere = NULL
vnode_p->v_shrlocks = NULL
bfNode = malloc (sizeof (bfNode) )
vnode_p->v_data = bfNode
bfNode->accessp = bfap
bfNode->fsContext = NULL
bfNode->tag = bfap->tag
bfNode->bfSetId = bfap->bfSetId

sts = fcache_vn_create( vnode_p,
                       FVC_DEFAULT,
                       0);
if (sts != EOK)
        return sts

bfap->pl_vp = vnode_p

return sts
```

3.2.2.1.4 Exceptions

3.2.2.2 advfs_pl_close

3.2.2.2.1 Interface

```
statusT
advfs_pl_close(bfAccess* bfap,
               pl_flags pl_flags,
               ftxHT parent_ftx,
               uint32_t *mcell_del_cnt,
               void *mcell_del_list
               )
```

3.2.2.2.2 Description

This routine is the counterpart to advfs_pl_access. When accessing a PL file is completed, this routine should be called to close the PL file and potentially remove the storage for the PL. If the PL file is open through .tags, the v_count will be non-0 and will prevent this routine from removing the underlying file.

If is assumed that the pl_file_lock is held for write on entrance to this function. This routine can be called multiple times on the same PL file without adverse effect. Unless the PL file is empty or the PLF_REMOVE_FILE flag is set, this routine will have no effect.

This routine will be called from bs_close_one when deleting the file. In that case, the parent_ftx will be non-FtxNilFtxT and pl_flags will have PLF_REMOVE_FILE set.

If parent_ftx is non-NULL, then the routine will not be able to call stg_remove_stg_finish since this routine requires a root transaction. In the event that the storage is freed and parent_ftx is non-NULL, the parameters mcell_del_cnt and mcell_del_list will be passed back to the caller. These will represent chains of storage on the DDL that must be passed to stg_remove_stg_finish once the root transaction is closed.

3.2.2.2.3 Execution Flow

```
pl_vp = bfap->pl_vp
if (pl_vp == NULL)
        /* If the pl_vp isn't set, the file doesn't exist, so there is nothing
         * to close. */
        return EOK


/* Check to see if we have anything to do on this close.  */
if (bfap->pl_element_cnt == 0 || PLF_REMOVE_FILE)

        if (pl_vp->v_count == 0 and bfap->pl_element_cnt == 0)

                FTX_START_N( FTA_PL_FILE_DESTROY, ftx )
                chk_blk_quota(
                        &bfap->bfVnode,
                        - QUOTABLKS( bfap->pl_file_size),
                        kt_cred(u.u_kthreadp),
                        0,
                        ftx )
                chk_bf_quota( &bfap->bfVnode,
                              -1,
                              kt_cred(u.u_kthreadp),
                              0,
                              ftx )

                stg_remove_stg_start( bfap,
                                      0,
                                      ADVFS_OFFSET_TO_FOBS_UP(bfap->pl_file_size),
                                      STGF_PL_FILE,
                                      ftx,
                                      &mcell_del_cnt,
                                      &mcell_del_list)

                /* Now set the state of the PL file to invalid before we drop the lock.
                 * Once we set it to invalid, we can drop the lock and finish freeing the
                 * storage.  Until we set it to invalid, we can't let anyone else see the
                 * storage. */
                read_lock( bfap->mcellList_lk )
                pl_mcell = bfap->primMcell
                sts = bmtr_scan_mcells( &pl_mcell,
                                      pl_mcell.volume,
                                      bsr_pl_rec,
                                      bmt_pg_ref,
                                      &rec_offset,
                                      &rec_size,
                                      BSR_PL_FILE,
                                      bfap->tag)
                ASSERT( sts != EBMTR_NOT_FOUND )
                unlock (bfap->mcellList_lk )
                derefpg (bmt_pg_ref )
                if (sts != EOK)
                        return sts or domain_panic?
                sts = rbf_pinpg( bmt_pg_hdl,
                               mcell_pg_ptr,
                               bfap->dmnP->vdTbl[pl_mcell.volume].bmtp,
                               pl_mcell.page,
                               BS_NIL,
                               ftx,
                               MF_VERIFY_PAGE | MF_PL_FILE)
                if (sts != EOK)
                        return sts or domain_panic?
                mcell_ptr = mcell_pg_ptr->bsMCA[ pl_mcell.cell ]
                bsr_pl_rec = mcell_ptr + rec_offset
                rbf_pin_record ( bmt_pg_hdl,
                                      bsr_pl_rec,
                                      sizeof(pl_ods_flags_t))
                bsr_pl_rec->bpr_flags = PLOF_INVALID

                ftx_done( ftx )

                unlock (bfap->pl_file_lock )

                if (parent_ftx == ftxNilFtx)
                        /* Now the storage is on the DDL, we can now finish the removal
                         * since we've dropped all our locks. */
                        stg_remove_stg_finish( dmnP,
                                      mcell_del_cnt,
                                      mcell_del_list)
                        mcell_del_cnt = 0
                        mcell_del_list = NULL
                /* If parent_ftx, then we will return the DDL storage to free */


                return EOK


        else
                /* We raced someone, just return */
                unlock( bfap->pl_file_lock )
                return EOK
return EOK
```

3.2.2.2.4 Exceptions

3.2.2.3 advfs_pl_create

3.2.2.3.1 Interface

```
statusT
advfs_pl_create(bfAccess* bfap, pl_flags *pl_flags, ftxH parent_ftx)
```

3.2.2.3.2 Description

This function is responsible for creating a PL file's on disk layout. The routine will create a transaction under which to make all necessary on disk changes for the PL file to be created. The routine will create a record in the primary mcell of the parent file (represented by bfap) that points to a new chain of mcells for the PL file. If the record already exists, then flags field of the record will be updated to show that the PL file is valid. The new chain of mcells will contain extents for the PL file and will initially have one allocation unit of storage. The allocation unit will default to ADVFS_METADATA_PGSZ. The first page will have a pl_file_hdr_t structure initialized in the first PL block and the remaining PL blocks will be initialized to be PL_FREE blocks.

It is expected that the caller holds the bfap->pl_file_lock for write access.

3.2.2.3.3 Execution Flow

```
ASSERT( bfap->pl_file_lock held in write mode )

FTX_START_N( FTA_PL_CREATE, ftx, parent_ftx )

/*
 * Search for the BSR_PL_FILE_RECORD to see if we need to create one, or just
 * modify the existing record.
 */
ftx_write lock ( &bfap->mcellList_lk )
sts = bmtr_scan_mcells( &prim_mcell,
                              &vdp,
                              &bsr_pl_rec_ptr,
                              &bmt_pg_ref,
                              &rec_pg_offset,
                              &rec_size,
                              BSR_PL_FILE_RECORD,
                              parent_tag)
if (sts == EBMTR_NOT_FOUND)
        bs_derefpg( bmt_pg_ref )
        /*
         * We need to create the record
         */
        pl_rec_mcell = bfap->primMcell
        sts = bmtr_create_rec( bfap,
                              pl_rec_mcell,
                              bfap->dmnP->vdTbl[pl_rec_mcell.volume],
                              pl_rec_ptr,
                              sizeof( bsr_pl_rec ),
                              BSR_PL_FILE,
                              ftx )
        if (sts != EOK)
                ftx_fail
                unlock mcellList_lk
                return sts


        /* We will modify the record at the end of the routine to reflect that
         * it is valid. */

else {
        /* We found the record, and will need to make it valid at the end of
         * this routine. */

/* We now have a record and it is not valid, and it's mcell pointer is not accurate.
 * we need to add storage, initialize the storage, then set the mcell pointer in the
 * record and set the valid flag. */
mcell_u_id.tag = bfap->tag.num
mcell_u_id.seg = bfap->tag.seq | PL_FILE_ID
mcell_u_id.ut.bfsid = bfap->bfSetp->bfSetId

bfAttr.cl.reqServices = bfap->
bfAttr.bfPgSz = bfap->bfPageSz
```

```
/* This call comes back with the mcell_lk locked by this ftx.  */
sts = new_mcell ( ftx,
               mcell_u_id
               &bfAttr,
               bfap->dmnP,
               STGF_PL_FILE )

if (sts != EOK)
       ftx_fail( ftx )
       unlock mcellList_lk
       return sts

sts = advfs_pl_access( bfap->tag,
                       bfap->bfSetp,
                       &pl_vp,
                       PLF_CREATING,
                       mcell_u_id.mcell)

Check for error, fail transaction accordingly

/* Now we have an empty mcell attached to the PL file.  We should now be able to add
 * storage to the PL file */
sts = stg_add_stg( ftx,
               bfap,
               0,
               ADVFS_METADATA_PGSZ_IN_FOBS,
               STGF_PL_FILE | STGF_NO_OVERLAP,
               &alloc_fob_cnt )

Check for error, fail transaction accordingly

/*
 * Now we can initialize page 0 of the PL file. First we'll just init the
 * entire range as pl blocks, then we'll lay down the pl_file_hdr structure.
 */
sts = advfs_pl_init_range( ftx,
                           bfap,
                           0,
                           ADVFS_METADATA_PGSZ_IN_FOBS * ADVFS_FOB_SZ)

Check for error and fail transaction accordingly


/* We need to initialize the pl_file_hdr_t in page 0 of the pl file, and
 * the bsr_pl_file record in the BMT.  We can not fail after we pin a record,
 * so we'll pin both pages, then pin the records to modify rather than doing it
 * one page modification at a time. */
sts = rbf_pinpg( pl_pg_hdl,
               pl_file_hdr_ptr,
               bfap,
               0,
               BS_NIL,
               ftx,
               MF_VERIFY_PAGE | MF_PL_FILE )

Check for error and fail transaction accordingly

sts = rbf_pinpg( bmt_pg_hdl,
               bmt_pg_ptr,
               bfap,
               0,
               BS_NIL,
               ftx,
               MF_VERIFY_PAGE | MF_PL_FILE )

/* Check for error and fail transaction accordingly.  Also unpin the pl_pg_hdl page from
 * above. */


rbf_pin_record ( pl_pg_hdl,
                       pl_file_hdr_ptr,
                      sizeof( pl_file_hdr_t )

pl_file_hdr_ptr->pfh_layout = PLV_FLAT
pl_file_hdr_ptr->pfh_elem_start = ADVFS_PL_BLOCK_SIZE
pl_file_hdr_ptr->pfh_next_free_blk_off = ADVFS_PL_BLOCK_SIZE
pl_file_hdr_ptr->pfh_free_byte_cnt = alloc_fob_cnt * ADVFS_FOB_SZ - ADVFS_PL_BLOCK_SIZE
pl_file_hdr_ptr->pfh_wasted_byte_cnt = 0
pl_file_hdr_ptr->pfh_free_blk_cnt = ((alloc_fob_cnt * ADVFS_FOB_SZ) /
                                             ADVFS_PL_BLOCK_SIZE) - 1
pl_file_hdr_ptr->pfh_file_size = ADVFS_METADATA_PGSZ_IN_FOBS * ADVFS_FOB_SZ
pl_file_hdr_ptr->pfh_pl_block_size = ADVFS_PL_BLOCK_SIZE

bfap->pl_layout = pl_file_hdr_ptr->plfh_layout
bfap->pl_block_size = pl_file_hdr_ptr->plfh_block_size
bfap->pl_file_size = pl_file_hdr_ptr->plfh_file_size
bfap->pl_alloc_size = pl_file_hdr_ptr->plfh_alloc_size
bfap->pl_free_blk_off = pl_file_hdr_ptr->plfh_free_blk_off

mcell_ptr = bmt_pg_ptr->bsMCA[ mcell_u_id.mcell.cell ]
bsr_pl_rec_ptr = bmtr_find( mcell_ptr,
                            BSR_PL_FILE,
                            bfap->dmnP )

ASSERT (bsr_pl_rec_ptr != NULL )


rbf_pin_record( bmt_pg_ptr,
               bsr_pl_rec_ptr,
               sizeof( bsr_pl_rec_t ))
bsr_pl_rec_ptr->pl_mcell = mcell_u_id.mcell
bsr_pl_rec_ptr->pl_flags = PLOF_VALID

ftx_unlock mcelList_lk

ftx_done( ftx )
```

3.2.2.3.4 Exceptions

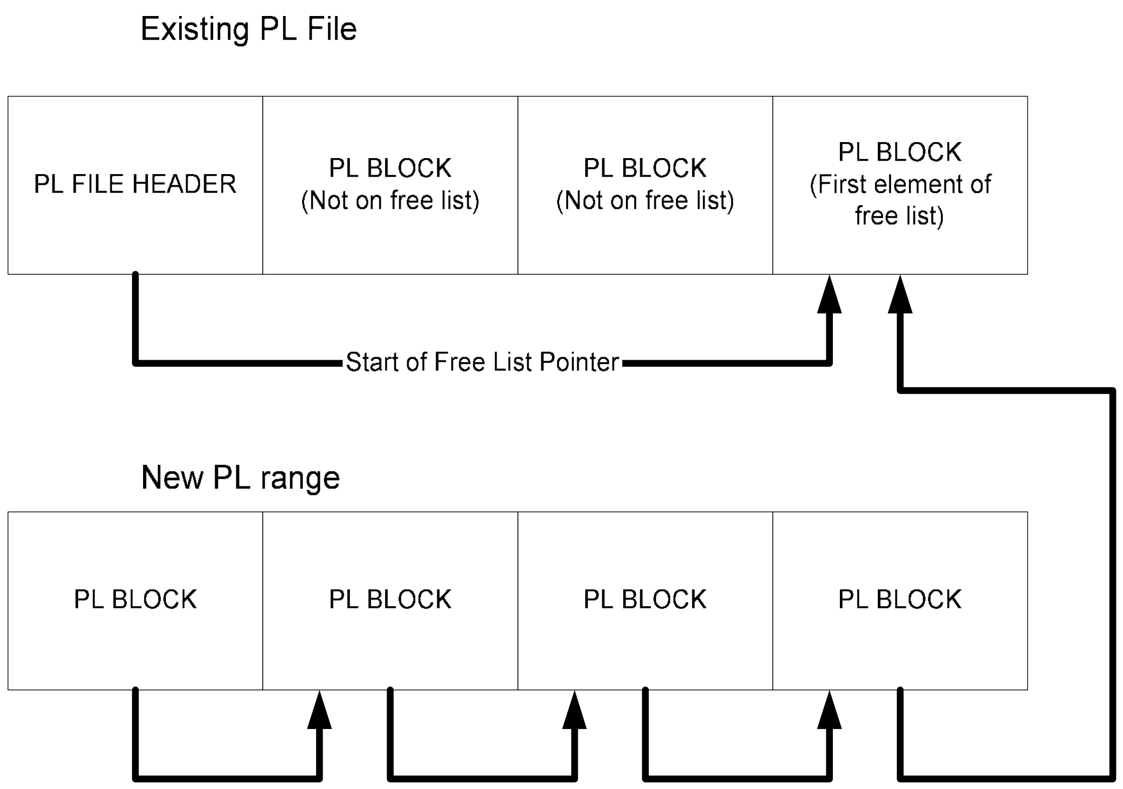
3.2.2.4 advfs_pl_init_range

3.2.2.4.1 Interface

```
statusT
advfs_pl_init_range(bfAccessT *bfap,
                      off_t start_offset,
                      size_t init_size
                      ftxHT parent_ftx)
```

3.2.2.4.2 Description

advfs_pl_init_range will initialize the range from [start_offset..start_offset+init_size] as a chain of free pl blocks. The chain will point to the current start of the PL file's free list but will not update the PL file headers free list pointer. At the end of this routine, the following diagram illustrates the file layout.


Existing PL File
PL FILE HEADER
PL BLOCK (Not on free list)
PL BLOCK (Not on free list)
PL BLOCK (First element of free list)
Start of Free List Pointer
New PL range
PL BLOCK
PL BLOCK
PL BLOCK
PL BLOCK

Although the routine will chain PL blocks, it must be aware of the number of pages that are pinned under a single transaction. When more than one allocation unit of the PL file is initialized, advfs_pl_init_range will need to pin pages between one allocation unit and the next.

It is required that the range to initialize start and end aligned on the PL file's allocation unit (bfap->pl_alloc_sz).

PL block initialized in this routine will have their type set to PLB_UNCLAIMED to indicate they are not yet claimed for any specific purpose.

3.2.2.4.3 Execution Flow

```
if ( parent_ftx == nilFtx  ||
        (start_offset < 0) ||
        (start_offset % (bfap->pl_alloc_sz * ADVFS_FOB_SZ) != 0) ||
        (init_size % (bfap->pl_block_size * ADVFS_FOB_SZ) != 0)
        return E_BAD_PARMS

if (init_size == 0)
        return E_BAD_PARAMS
/*
 * Start a transaction to tie together multiple sub transaction if necessary.
 * Similar in design to storage addition model
 */
FTX_START_N( FTA_PL_INIT_RANGE_TOP, &ftx_top, parent_ftx, bfap->dmnP )

/*
 * Start the first init range transaction
 */
FTX_START_N( FTA_PL_INIT_RANGE, &cur_ftx, ftx_top, bfap->dmnP )
current_offset = start_offset
```

```
end_offset = start_offset + init_size
prev_pl_pg_ptr = NULL
prev_pl_pg_offset = -1
pl_blocks_per_alloc = bfap->pl_alloc_sz / bfap->pl_block_sz
pinned_a_record = FALSE

while (current_offset < end_offset)
        /*
         * Pin the page to initialize
         */
        sts = rbf_pinpg( page_hdl,
                        &pl_page_ptr,
                        bfap,
                        current_offset / (bfap->pl_alloc_sz * ADVFS_FOB_SZ),
                        BS_NIL,
                        cur_ftx,
                        MF_NO_VERIFY | MF_PL_FILE )
        if (sts == E_MAX_PINP_EXCEEDED || !rbf_can_pin_record( page_hdl ) )
                ftx_done( cur_ftx )
                FTX_START_N( cur_ftx, top_ftx )
                pinned_a_record = FALSE
                sts = rbf_pinpg( page_hdl,
                        &pl_page_ptr,
                        bfap,
                        current_offset / (bfap->pl_alloc_sz * ADVFS_FOB_SZ),
                        BS_NIL,
                        cur_ftx,
                        MF_NO_VERIFY | MF_PL_FILE )


        if (sts != EOK)
                if (pinned_a_record)
                        ftx_done( cur_ftx )
                else ftx_fail (cur_ftx)
                ftx_fail( top_ftx )
                return sts


        /*
         * Initialize the page and point the previous page to this page.
         */
        rbf_pin_record( page_hdl, pl_page_ptr, bfap->pl_alloc_sz * ADVFS_FOB_SZ )
        bzero(pl_page_ptr, bfap->pl_alloc_sz * ADVFS_FOB_SZ)
        pinned_a_record = TRUE

        current_block_ptr = pl_page_ptr
        prev_block_ptr = NULL
        for (i = 0; i < pl_blocks_per_alloc; i++)
                /*
                 * Setup the pl_block header.
                 */
                pl_block = (pl_block_hdr_t*)current_block_ptr;
                pl_block->pbh_next_free_blk_off = PLC_END_OF_LIST
                pl_block->pbh_free_chunk_off = sizeof(pl_block_hdr_t)
                pl_block->pbh_block_type = PLB_UNCLAIMED

                /* Initialize the free space */
                pl_seg_hdr = &current_block_ptr + sizeof(pl_block_hdr_t)
                pl_seg_hdr.pls_size = bfap->pl_block_size - sizeof(pl_block_hdr_t)
                pl_seg_hdr.pls_cont_off = PLC_END_OF_LIST
                pl_seg_hdr.pls_next_offset = PLC_END_OF_LIST
                pl_seg_hdr.pls_prev_offset = PLC_END_OF_LIST
                pl_seg_hdr.pls_type = PLST_FREE_SEG

                if (prev_block_ptr)
                        prev_block_ptr->pbh_next_free_blk_off = current_offset +
                                                             (i * bfap->pl_block_sz)
                prev_block_ptr = current_block_ptr
                current_block_ptr += bfap->pl_block_sz
        /*
         * This allocation unit of the PL file is internally linked correctly,
         * if we initialized a previous allocation unit, we need to point it's last
         * pl_block at the first block of this allocation unit.
         */
        if (prev_pl_pg_offset != -1 )
               sts = rbf_pinpg( page_hdl,
                      &pl_prev_page_ptr,
                      bfap,
                      prev_pl_pg_offset / (bfap->pl_alloc_sz * ADVFS_FOB_SZ),
                      BS_NIL,
                      cur_ftx,
                      MF_NO_VERIFY | MF_PL_FILE )
               if (sts == E_MAX_PINP_EXCEEDED || !rbf_can_pin_record( page_hdl ) )
                      ftx_done ( cur_ftx )
                      FTX_START_N( cur_Ftx, top_ftx )
                      pinned_a_record = FALSe
                      sts = rbf_pinpg( page_hdl,
                                   &pl_prev_page_ptr,
                                   bfap,
                                   prev_pl_pg_offset /
                                     (bfap->pl_alloc_sz * ADVFS_FOB_SZ),
                                   BS_NIL,
                                   cur_ftx,
                                   MF_NO_VERIFY | MF_PL_FILE )
               if (sts != EOK)
                      if (pinned_a_record)
                                   ftx_done( cur_ftx )
                      else ftx_fail (cur_ftx)
                      ftx_fail(top_ftx)
                      return sts

               pin the record of the last PL block in the prev pg and point the
               pls_next_offset to current_offset.


        current_offset += bfap->pl_alloc_sz * ADVFS_FOB_SZ


/*
 * Wrap up by finishing transactions
 */
ftx_done_n(cur_ftx, FTA_PL_INIT_RANGE)

ftx_done_n(cur_ftx, FTA_PL_INIT_RANGE_TOP)

return EOK
```

3.2.2.4.4 Exceptions

The routine may return E_BAD_PARAMS if the init_size is 0 or if start_offset or init_size are not allocation unit aligned. The routine may also return EIO or any error status from rbf_pinpg.

3.2.2.5 advfs_pl_insert_int

3.2.2.5.1 Interface

```
statusT
advfs_pl_insert_int( bfAccessT * bfap,
                      pl_data_t *pl_data,
                      pl_flags_t* pl_flags,
                      ftxHT parent_ftx,
                      uint64_t* clean_pl_blocks_used,
                      uint64_t *free_bytes_consumed,
                      uint64_t *bytes_wasted)
```

3.2.2.5.2 Description

This routine manages inserting an element in the PL file. The caller has already taken the PL file lock and added enough storage for the PL element to be added. This routine will manage spreading the PL data across a sufficient number of PL blocks and PL free chunks to hold the entire element. On success, the routine will return in clean_pl_blocks_used the number of totally empty PL blocks that were used (transitioned from PLB_UNCLAIMED to PLB_DATA). In free_bytes_consumed, the routine will return the number of bytes that were used in total to store the PL element. This will include all free chunks used in PL blocks that were already in use (PLB_DATA) along with any new blocks used. The free_bytes_consumed will include overhead for the pl_segment_hdr_t structures used to chain the PL element across multiple free chunks or PL blocks plus any wasted bytes.

This routine may need to pin a large number of pages in order to write the PL element. Because more than FTX_MX_PINP may need to be pinned, the routine will start a top level transaction and do all the pins as subtransactions of that top level transaction.

The basic flow for this routine is to start a top level transaction and then loop, pinning free blocks as they occur in the free list. Every time FTX_MX_PINP pages are pinned, ftx_done will be called on the sub transaction and a new transaction will be started to continue the write.

To simplify the routines transaction management, each block will be pinned separately. The transaction code will correctly handle this if two blocks land on the same page. rbf_pinpg has been modified to return E_MX_PINP_EXCEEDED when the maximum number of pages is reached, and it will not count a double pin of a page as two pins, so pinning a page several times will have no impact on the transaction.

If the PLF_ACL flag is set in the pl_flags parameter, then it is expected that the uio structure will have been dummied up to represent a normal PL element. In the case of PLF_ACL, a bcopy will be used from the address in uio->iovec, rather than a fcache_as_uiomove. The PLF_ACL's fake uio structure should point to a buf that contains the name and data, that is {SYSV_ACL, <*acl_data*>}. The uio->offset should be 0 on entry, and the uio->resid should represent the length in bytes of the name and data pair.

3.2.2.5.3 Execution Flow

```
ASSERT(pl_flags & PLF_INSERT)
ASSERT(pl_data->pld_element)

*clean_pl_blocks_used = 0
*free_bytes_consumed = 0
*bytes_wasted = 0

/*
 * malloc a buffer big enough to hold the undo record.  This is a variable sized
 * undo record with a bounded size (can't have more that FTX_MX_PINR records to undo)
 */
undo_buf = malloc (sizeof( pl_generic_hdr_undo ) +
                      (FTX_MX_PINR * sizeof( pl_ins_del_multi_undo ) )
undo_record_cnt = 0;
cur_undo_record = undo_buf + sizeof( pl_generic_hdr_undo )


/*
 * Get the size of the data to store, not counting overhead
 */

remaining_insert_size = pl_data->pld_element.plie_name_ +
                            pl_data->pld_element.plie_data_len

/*
 * Start a top level insert_int transaction and the first sub transaction
 */
FTX_START_N( FTA_PL_INSERT_INT_TOP, &ftx_top, parent_ftx, bfap->dmnP )
FTX_START_N( FTA_PL_INSERT_INT, &cur_ftx, ftx_top, bfap->dmnP )

/*
 * Keep looping until we have done the entire write
 */
wrote_hdr = FALSE
pin_rec_count = 0

uio = pl_data->pld_uiop

/*
 * Consume free space from the front of the free list without try to be intelligent about
 * fragmentation
 */
while (remaining_insert_size > 0)
        /* Get the pg num/alloc num of the PL block at the head of the free list */
        pl_alloc_num = bfap->pl_free_blk_off / (bfap->pl_alloc_sz * ADVFS_FOB_SZ)
        sts = rbf_pinpg( page_hdl,
                        pl_pg_ptr,
                        bfap,
                        pl_alloc_num,
                        BS_NIL,
                        cur_Ftx,
                        MF_VERIFY_PAGE | MF_PL_FILE)
        if (sts == E_MX_PINP_EXCEEDED)
                undo_hdr = undo_buf
                undo_hdr->plgu_multi_cnt = undo_record_cnt
                ftx_done_u( cur_ftx,
                                FTA_PL_INSERT_INT,
                                sizeof( pl_generic_hdr_undo ) +
                                        (undo_record_cnt *
                                        sizeof( pl_ins_del_multi_undo ),
                                undo_buf)
                undo_record_cnt = 0;
                cur_undo_record = undo_buf + sizeof( pl_generic_hdr_undo )

                FTX_START_N( FTA_PL_INSERT_INT, &cur_ftx, ftx_top, bfap->dmnP )
                sts = rbf_pinpg( page_hdl,
                                pl_pg_ptr,
                                bfap,
                                pl_alloc_num,
                                BS_NIL,
                                cur_Ftx,
                                MF_VERIFY_PAGE | MF_PL_FILE)
                pin_rec_count = 0

        if (sts != EOK)
                if (pin_rec_count > 0)
                        undo_hdr = undo_buf
                        undo_hdr->plgu_multi_cnt = undo_record_cnt
                        ftx_done_u( cur_ftx,
                                        FTA_PL_INSERT_INT,
                                        sizeof( pl_generic_hdr_undo ) +
                                                (undo_record_cnt *
                                                sizeof( pl_ins_del_multi_undo ),
                                        undo_buf)
                        undo_record_cnt = 0;
                        cur_undo_record = undo_buf + sizeof( pl_generic_hdr_undo )
                else
                        ftx_fail( cur_ftx )
                ftx_fail( top_ftx )
                return sts

        pl_block_ptr = pl_pg_ptr + (bfap->pl_free_blk_off –
                                (pl_alloc_num * bfap->pl_alloc_sz * ADVFS_FOB_SZ))



        /*
         * Loop over all free chunks on this PL block and consume whatever
         * we can.
         */
        pin_rec_count = 0
        wasted_space = 0
        free_chunk_offset = pl_block_ptr->pbh_free_chunk_off
```

```
while (free_chunk_offset != PLC_END_OF_LIST) && (remaining_insert_size > 0)
        free_chunk_ptr = pl_block_ptr + free_chunk_offset
        next_free_chunk_offset = free_chunk_ptr->pls_next_offset
        free_size = free_chunk_ptr->pls_size - sizeof( pl_segment_hdr_t )

        /*
         * This free chunk is too small to be useful and should not have gone
         * on the free list to begin with.  It is wasted space.
         */
        ASSERT(free_size > PL_MIN_FREE_CHUNK_SZ)


        /*
         * Pin the free space we want to use.
         */
        if (!rbf_can_pin_record(page_hdl))
                undo_hdr = undo_buf
                undo_hdr->plgu_multi_cnt = undo_record_cnt
                ftx_done_u( cur_ftx,
                                FTA_PL_INSERT_INT,
                                sizeof( pl_generic_hdr_undo ) +
                                        (undo_record_cnt *
                                        sizeof( pl_ins_del_multi_undo ),
                                undo_buf)
                undo_record_cnt = 0;
                cur_undo_record = undo_buf + sizeof( pl_generic_hdr_undo )

                FTX_START_N( FTA_PL_INSERT_INT, &cur_ftx, ftx_top, bfap->dmnP )
                sts = rbf_pinpg( page_hdl,
                                pl_pg_ptr,
                                bfap,
                                pl_alloc_num,
                                BS_NIL,
                                cur_Ftx,
                                MF_VERIFY_PAGE | MF_PL_FILE)
                if (sts != EOK)
                        ftx_fail( cur_ftx )
                        ftx_fail( top_ftx )
                        return sts
                pin_rec_count = 0


        rbf_pin_record( page_hdl,
                        free_chunk_ptr,
                        free_size)
        pin_rec_count++

        /*
         * Setup an undo for the change we are about to make, and
         * do a structure copy of the previous free chunk header.
         */
        cur_undo_record->plidu_type = PL_INS_SEG_UNDO;
        cur_undo_record->plidu_offset = free_chunk_offset
        cur_undo_record->plidu_data.prev_pl_free_hdr = free_chunk_ptr
        undo_record_cnt++
        cur_undo_record += sizeof( pl_insert_int_multi_undo )

        /*
         * If this is the first element segment, use the pl_segment_hdr type
         * PLST_ELEM_HDR
         * Otherwise, use type PLST_ELEM_CONT
         */
        pl_buf_start = NULL
        pl_buf_offset = 0
        if (!wrote_hdr)
                pl_segment_hdr = free_chunk_ptr
                pl_segment_hdr->pls_type = PLST_ELEM_HDR
                pl_segment_hdr->pls_element_hdr.pleh_name_len = name_len
                pl_segment_hdr->pls_element_hdr.pleh_data_len = data_len
                pl_segment_hdr->pls_element_hdr.pleh_name_hash =
                        advfs_pl_name_hash(pl_data->pld_name,
                                      pl_data->pld_namelen)
        pl_segment_hdr->pleh_cont_off = PLC_END_OF_LIST
        pl_segment_hdr->pleh_flags = PLF_ELEMENT_VALID
        pl_buf_start = pl_free_chunk_ptr + sizeof (pl_segment_hdr_t)
        pl_buf_offset = free_chunk_offset + sizeof( pl_segment_hdr_t)
        free_bytes_consumed += sizeof(pl_segment_hdr_t)
        free_size -= sizeof(pl_segment_hdr_t)
        free_chunk_offset += sizeof(pl_segment_hdr_t)
        wrote_hdr = TRUE
else
        pl_segment_hdr->pls_cont_off = PLC_END_OF_LIST
        pl_buf_start = pl_free_chunk_ptr + sizeof (pl_segment_hdr_t)
        pl_buf_offset = free_chunk_offset + sizeof( pl_segment_hdr_t)
        free_bytes_consumed += sizeof(pl_segment_hdr_t)
        free_size -= sizeof(pl_segment_hdr_t)
        /*
         * Link the previous segment to this one.
         */
        prev_pl_seg_hdr->pls_cont_off = free_chunk_offset

        free_chunk_offset += sizeof(pl_segment_hdr_t)

        prev_pl_seg_hdr = pl_segment_hdr

cur_buf_len = remaining_insert_size
bytes_to_write = min(cur_buf_len, free_size)

pl_segment_hdr->pls_size = roundup( bytes_to_write, sizeof( uint64_t ) )

if (pl_flags & PLF_ACL) {
        /* The uio structure is really just dummied up.  We
         * have to bcopy out of the buffer it points to, then
         * then increment the offset. */
        bcopy( uio->iovec->iov_base + uio->offset,
                pl_buf_start, bytes_to_write)
        uio->offset += bytes_to_write
        uio->resid -= bytes_to_write
} else {
        /* The normal PL element case.  This will almost always only
         * execute once, so there is no need to be “smart” about mapping.
         * */
        error = EOK
        while ((bytes_to_write > 0 && (error == EOK))
                if !(fcmap = fcache_as_map( bfap->dmnP->metadataVas,
                                bfap->pl_vp,
                                pl_buf_offset,
                                bytes_to_write,
                                0,
                                FAM_WRITE, &error) )
                        If (error == EINVAL)
                                ADVFS_SAD ☹
                        break
                bytes_written = 0
                /*
                 * The uio structure should be setup to point to the
                 * start of the name buffer we are going to write.  The
                 * data buffer should immediately follow that, so
                 * we just do a write of as much as we can in this
                 * map handle which covers exactly the free chunk we
                 * can fill (or the amount remaining to write, which is
                 * less. The uiomove will update the resid and offset of
                 * the uio structure. This will fault on pl_vp…*/
                if (error = fcache_as_uiomove(fcmap,
                                             NULL, /* We don’t need to
                                                     specify the address,
                                                     even though we have
                                                     it */
                                             &bytes_written,
                                             uio,
                                             NULL, /* No private params */
                                             FAUI_WRITE)
                        If (error == EINVAL)
                                ADVFS_SAD ☹
                        break
                bytes_to_write -= bytes_written
                if (error = fcache_as_unmap(fcmap, NULL, 0, FAU_CACHE)
                        if (error = EINVAL || error = EFAULT)
                                ADVFS_SAD ☹
                        Break
        /*
         * If we got an error, we need to fail the transaction and return
         */
        if (error != EOK)
                undo_hdr = undo_buf
                undo_hdr->plgu_multi_cnt = undo_record_cnt
                ftx_done_u( cur_ftx,
                        FTA_PL_INSERT_INT,
                        sizeof( pl_generic_hdr_undo ) +
                                (undo_record_cnt *
                                sizeof( pl_ins_del_multi_undo ),
                        undo_buf)
                ftx_fail( top_ftx )
                return error

/*
 * If we had any free space left and data to write, we wrote more.  At
 * this point, if cur_buf_len is < free_size, we must have completed
 * the write.  If cur_buf_len > free_size then we will continue on keep
 * writing.  Otherwise, we'll add any remaining free space back to the
 * free list.
 */
if (cur_buf_len < free_size)
        pl_buf_start += bytes_to_write

        /* For alignment issues, we will round the size to 64 bits.
         */
        free_size -= bytes_to_write
        rounddown( free_size, sizeof( uint64_t ) )
        free_chunk_offset += bytes_to_write
        roundup( free_chunk_offset, sizeof( uint64_t ) )

        new_free_offset_start = pl_buf_start – free_chunk_ptr
        if (free_size < PL_MIN_FREE_CHUNK_SZ)
                wasted_space += free_size
                free_bytes_consumed += wasted_space
                free_chunk_offset = next_free_chunk_offset
        else
                free_hdr = pl_buf_start
                free_hdr->plfh_size = free_size + sizeof( pl_segment_hdr_t)
                free_hdr->plfh_next_free_off = free_chunk_offset
                /*
                 * We don't update free chunk offset, but we better
                 * be done with the write at this point
                 */
                ASSERT(remaining_insert_size == 0)
else
        /* We used the entire free chunk, so advance to the next one */
        free_chunk_offset = next_free_chunk_offset
        cur_buf_len -= bytes_to_write


/* We've used all the free space on this pl block or have finished the insert.
 * Now, update the pl block hdr and see if we are approaching our pin limit.
 * Additionally, if the PL block is type PLB_UNCLAIMED then we need to transition
 * it to data since it is now in use and we need to bump clean_pl_blocks_used */
if (rbf_can_pin_record(page_hdl))
        rbf_pin_record( page_hdl,
                        pl_block_ptr,
                        sizeof(pl_block_hdr_t) )
        pin_rec_count++;
else
```

```
                undo_hdr = undo_buf
                undo_hdr->plgu_multi_cnt = undo_record_cnt
                ftx_done_u( cur_ftx,
                               FTA_PL_INSERT_INT,
                               sizeof( pl_generic_hdr_undo ) +
                                       (undo_record_cnt *
                                       sizeof( pl_ins_del_multi_undo ),
                               undo_buf)
                undo_record_cnt = 0;
                cur_undo_record = undo_buf + sizeof( pl_generic_hdr_undo )

                FTX_START_N( FTA_PL_INSERT_INT, &cur_ftx, ftx_top, bfap->dmnP )
                sts = rbf_pinpg( page_hdl,
                               pl_pg_ptr,
                               bfap,
                               pl_alloc_num,
                               BS_NIL,
                               cur_Ftx,
                               MF_VERIFY_PAGE | MF_PL_FILE)
                if (sts != EOK)
                        ftx_fail( cur_ftx )
                        ftx_fail( ftx_top )
                        return sts

                pin_rec_count = 0
                rbf_pin_record( page_hdl,
                               pl_block_ptr,
                               sizeof(pl_block_hdr_t) )
                pin_rec_count = 1
        /*
         * Setup an undo for the change we are about to make, and
         * do a structure copy of the previous free chunk header.
         */
        cur_undo_record->plidu_type = PL_INS_HDR_UNDO;
        cur_undo_record-> plidu_offset = bfap->pl_free_blk_off;
        cur_undo_record-> plidu_u.plidu_prev_pl_seg_hdr = pl_block_ptr
        undo_record_cnt++
        cur_undo_record += sizeof( pl_ins_del_multi_undo )


        /*
         * Update the free list for this block.  If free_chunk_offset
         * is PLC_END_OF_LIST, then we used all free space on this PL
         * block so we remove it from the list and update the
         * pl_free_blk_off in the bfap to point to the next element
         * on the free list. We always update the pbh_free_chunk_off
         * although this may set it to PLC_END_OF_LIST
         */
        if (free_chunk_offset == PLC_END_OF_LIST)
                bfap->pl_free_blk_off = pl_block_ptr->pbh_next_free_blk_off
                pl_block_ptr->pbh_next_free_blk_off = PLC_NOT_ON_LIST
        pl_block_ptr->pbh_free_chunk_off = free_chunk_offset


        if (pl_block_ptr->pbh_block_type == PLB_UNCLAIMED)
                if (bfap->pl_layout == PLV_FLAT)
                        pl_block_ptr->pbh_block_type = PLB_FLAT_DATA
                else
                        pl_block_ptr->pbh_block_type = PBL_B_TREE_DATA








/*
 * The insert is now completed.  The caller is responsible for updating the file hdr
 * structure.  The new file head of the free list is set in bfap->pl_free_blk_off.
 */

/*
 * Wrap up by finishing transactions
 */
undo_hdr = undo_buf
undo_hdr->plgu_multi_cnt = undo_record_cnt
ftx_done_u( cur_ftx,
               FTA_PL_INSERT_INT,
               sizeof( pl_generic_hdr_undo ) +
                       (undo_record_cnt *
                       sizeof( pl_ins_del_multi_undo ),
               undo_buf)
free(undo_buf)
ftx_done_n(cur_ftx, FTA_PL_INSERT_INT_TOP)

return EOK
```

3.2.2.5.4 Exceptions

This routine could fail with EIO or other return status from rbf_pinpg.

3.2.2.6 advfs_pl_uiomove_read

3.2.2.6.1 Interface

3.2.2.6.2 Description

This function takes a pl_data structure with a uio structure and an offset and moves the name and data elements of the PL element starting at pl_data->pld_offset to buffer represented by the uio structure. This routine only does reads from the pl_element and does not handle writes since writes are handled by the insert code.

3.2.2.6.3 Execution Flow

```
uio = pl_data->pld_uio

/* We need to loop over the PL element starting at the offset defined in pl_data->offset.
 */
next_seg_offset = pl_data->pld_offset
error = EOK
while (next_seg_offset != PLC_END_OF_LIST && (error == EOK))

        /* Ref the page we want to read from */
        cur_seg_offset = next_seg_offset
        cur_seg_pg = cur_seg_off / (bfap->pl_alloc_sz * ADVFS_FOB_SZ)
        /* Ref the page with the segment on it. */
        sts = bs_refpg( pg_hdl,
                        pg_ptr,
                        bfap,
                        nilFtxH,
                        MF_VERIFY | MF_PL_FILE)
        if (sts != EOK)
                return sts
        cur_seg_ptr = pg_ptr + (cur_seg_offset –
                                (cur_seg_pg * bfap->pl_alloc_sz * ADVFS_FOB_SZ))
        next_seg_offset = cur_seg_ptr->pls_next_offset

        /*
         * If we can’t fit the entire PL into the buffer they’ve given us, then fail.
         */
        if (uio->resid < pl_data->datalen + pl_data->namelen)
                return E_BUFFER_TOO_SMALL
        bytes_to_read = cur_seg_ptr->pls_size – sizeof(pl_segment_hdr_t)
        if (pl_flags & PLF_ACL) {
                /* The uio structure is really just dummied up.  We
                 * have to bcopy out of the buffer it points to, then
                 * then increment the offset. */
                bcopy(pl_buf_start, uio->iovec->iov_base + uio->offset, bytes_to_read)
                uio->offset += bytes_to_read
                uio->resid -= bytes_to_read
        } else {
                /* The normal PL element case.  This will almost always only
                 * execute once, so there is no need to be "smart" about mapping.
                 * */
                error = EOK
                while ((bytes_to_read > 0 && (error == EOK))
                        if !(fcmap = fcache_as_map( bfap->dmnP->metadataVas,
                                        bfap->pl_vp,
                                        pl_buf_offset,
                                        bytes_to_read,
                                        0,
                                        FAM_READ, &error) )
                                If (error == EINVAL)
                                        SAD ☹
                                break
                        bytes_written = 0
                        /*
                         * The uio structure should be setup to point to the
                         * start of the buffer we are going to read into.  The
                         * data buffer should immediately follow that, so
                         * we just do a write of as much as we can in this
                         * map handle which covers exactly the free chunk we
                         * can fill (or the amount remaining to read, which is
                         * less. The uiomove will update the resid and offset of
                         * the uio structure. */
                        if (error = fcache_as_uiomove(fcmap,
                                                      NULL, /* We don't need to
                                                               specify the address,
                                                               even though we have
                                                               it */
                                                      &bytes_read,
                                                      uio,
                                                      NULL, /* No private params */
                                                      FAUI_WRITE)

                        If (error == EINVAL)
                                SAD ☹
                        break
                bytes_to_read -= bytes_read
                if (error = fcache_as_unmap(fcmap, NULL, 0, FAU_CACHE)
                        if (error = EINVAL || error = EFAULT)
                                SAD ☹
                        Break

        bs_derefpg( pg_hdl)
return error
```

3.2.2.6.4 Exceptions

3.2.2.7 advfs_pl_ins_del_undo

3.2.2.7.1 Interface

```
statusT
advfs_pl_ins_del_undo( ftxHT ftx,
                       int32_t undo_rec_sz,
                       void* undo_buf)
```

3.2.2.7.2 Description

This routine must undo all the changes done by either an insert_int or a delete subtransaction. The undo operation will process a set of pl_ins_del_multi_undo records and undo the changes described by them. This will generally change the segment header from one state to another or modify the free list in some way.

One of five types of multi record is expected in this routine, PL_DEL_HDR_UNDO, PL_DEL_SEG_UNDO, PL_DEL_FILE_HDR_UNDO, PL_INS_HDR_UNDO or PL_INS_SEG_UNDO. These five records represent an undo of a pl block header structure, a segment header structure and a file header structure. It is assumed that no more than the maximum number of records pinnable on a page are passed into the routine to be undone.

Note that the undo routine does not need to deal with any actual PL element data. A redo will record the PL element data, but undos only need to restore the headers so the data looks valid. Deletion operations must be careful to do a delete by only modifying headers of the PL element and not the data itself, that way, restoring the headers in an undo will be sufficient to restore an entire PL element.

3.2.2.7.3 Execution Flow

```
undo_hdr = (pl_generic_hdr_undo) undo_buf
cur_cnt = 0



/* Make sure we open the pl file before we update it. We don't want to create the
 * PL file if it doesn't already exist. */
sts = bs_access( &bfap,
                       rtdn_hdr->plfs_bf_set_tag,
                       rtdn->plfs_bf_tag,
                       ftxNilFtx,
                       BF_OP_NO_FLAGS,
                       )
if (sts != EOK)
        return

/*
 * Process all the undo records.  We may pin the same page multiple times,
 * but that should be ok with the transaction code.
 */
for(cur_cnt = 0; cur_cnt < undo_hdr->plds_mutli_cnt, cur_cnt++)
        cur_undo = (pl_ins_del_multi_undo) undo_buf +
                               cur_cnt * sizeof(pl_generic_hdr_undo)
        alloc_unit = cur_undo->plidu_offset / (bfap->pl_alloc_unit * ADVFS_FOB_SZ)
        alloc_unit_off = rounddown( cur_undo->plidu_offset,
                                     bfap->pl_alloc_unit *ADVFS_FOB_SZ)
        undo_rec_pg_off = cur_undo->plidu_offset - alloc_unit_off
        rbf_pinpg( page_hdl,
                       pg_ptr,
                       bfap,
                       alloc_unit,
                       BS_NIL,
                       cur_ftx,
                       MF_VERIFY | MF_PL_FILE)
        switch (cur_undo)
        PL_DEL_HDR_UNDO, PL_INS_HDR_UNDO:
                pl_block_ptr = pg_ptr + undo_rec_pg_off
                rbf_pin_record( page_hdl, pl_block_ptr, sizeof(pl_block_hdr_t))
                *pl_block_ptr = *cur_undo->plidu_prev_block
                break
        PL_DEL_SEG_UNDO, PL_INS_SEG_UNDO:
                pl_seg_ptr = pg_ptr + undo_rec_pg_off
                rbf_pin_record( page_hdl, pl_seg_ptr, sizeof(pl_segment_hdr_t))
                *pl_seg_ptr = *cur_undo->plidu_prev_seg
                break
        PL_DEL_FILE_HDR_UNDO:
                pl_file_hdr_ptr = pg_ptr + undo_rec_pg_off
                rbf_pin_record( page_hdl, pl_file_hdr_ptr, sizeof(pl_file_hdr_t))
                *pl_file_hdr_ptr = *cur_undo->plidu_prev_file_hdr
                update bfap fields to match file header fields
                break
bs_close(bfap)
```

3.2.2.7.4 *Exceptions*

In the event that the PL file has already been deleted and cannot be opened, this routine will just return without doing anything.

3.2.3 Flat file support routines

3.2.3.1 advfs_pl_flat_lookup

3.2.3.1.1 *Interface*

statusT
```
advfs_pl_flat_lookup( bfAccessT *bfap,
                       pl_data_t *pl_data,
                       pl_flags_t *pl_flags)
```

3.2.3.1.2 *Description*

This routine does a linear search of the PL file for the PL element defined by pl_data->pld_name. The routine assumes that the PL file has been protected through some form of locking by the caller. The only expected caller of this routine is advfs_pl_lookup. In either an indexed or a flat layout PL file, this routine should succeed in finding an element if it exists in the PL file, however it should not be required to call this routine in an indexed file layout.

If this routine successfully finds the PL element defined by pl_data->pld_name, the pl_data fields pl_data->pld_datalen and the pl_data->pld_offset fields setup.

The basic flow of this routine is to start at the file header and walk the chain of PL segments that are linked together starting at the pl_file_hdr->pfh_elem_start offset and continuing through the pl_segment_hdr->pls_next_offset fields. All encountered segments on this chain should be of type PLST_ELEM_HDR or PLST_ELEM_INVALID. If a PLST_ELEM_INVALID type is encountered, it means it couldn’t be correctly removed during a previously attempted modification. We will send a message to the advfs_handyman_thread to have it cleaned up asynchronously.

When a new header is examined, the hash value in the header is first compared to the hash of the name being searched for. If the hash values match, a buffer is malloced which is big enough to hold the entire name of the PL element. The name is read in and if a match if found, the name is set in the pl_data->pld_name field and the PLF_FOUND flag is set in pl_flags. If the hash matches, but the name does not, then the buffer is freed and the search continues.

3.2.3.1.3 *Execution Flow*

```
name_hash = advfs_pl_name_hash(pl_data->pld_name,
                              pl_data->pld_namelen)
name = NULL
data = NULL
next_hdr_offset = bfap->pl_start_off

while ( (next_hdr_offset != PLC_END_OF_LIST)
        hdr_offset = next_hdr_offset
        /*
         * Do a page lookup at hdr_offset’s page.
         */
        pl_pg_off = rounddown(bfap->pl_start_off, bfap->pl_alloc_sz * ADVFS_FOB_SZ)
        bs_refpg( pg_hdl,
                  pg_ptr,
                  bfap,
                pl_pg_off / (bfap->pl_alloc_sz * ADVFS_FOB_SZ),
                ftxNilFtx,
                MF_VERIFY | MF_PL_FILE)
        pl_hdr_ptr = pg_ptr + (hdr_offset - pl_pg_off)
        next_hdr_offset = pl_hdr_ptr->pls_next_offset
        namelen = pl_hdr_ptr->pls_element_hdr->pleh_namelen

        if (pl_hdr_ptr->pls_type == PLST_ELEM_INVALID)
            create a message for the advfs_handyman thread.
            Setup the tag, the bf set id, the offset and the hash that was
            found.  Advfs_pl_delete will validate that the element is
            still PLST_ELEM_INVALID.
            Send message to the advfs_handyman_thread
            continue


        if (pl_hdr_ptr->pleh_name_hash == name_hash)
            /* Read the entire name */
            remaining_name_len = _hdr_ptr->pls_element.pleh_namelen
            namebuf = malloc(remaining_name_len)
            namebuf_off = 0
            pl_seg_hdr = pl_hdr_ptr
            do {
                size_to_copy = min( pl_seg_hdr->pls_size -
                                    sizeof( pl_segment_hdr_t ),
                                    remaining_name_len)
                bcopy(pl_seg_ptr + sizeof(pl_segment_hdr_t),
                    namebuf + namebuf_off,
                    size_to_copy)
                namebuf_off += pl_seg_ptr->pls_size - sizeof(pl_segment_hdr_t
                remaining_name_len -= size_to_copy
                next_seg = pl_seg_ptr->pls_cont_off
                if (next_seg != PLC_END_OF_LIST)
                    bs_derefpg(pg_hdl)
                    pl_pg_off = rounddown(next_seg,
                                    bfap->pl_alloc_sz * ADVFS_FOB_SZ)
                    bs_refpg( pg_hdl,
                                    pg_ptr,
                                    bfap,
                                    pl_pg_off /
                                            (bfap->pl_alloc_sz * ADVFS_FOB_SZ),
                                    ftxNilFtx,
                                    MF_VERIFY | MF_PL_FILE)
                    pl_seg_hdr = pg_ptr + (pl_seg_off - pl_pg_off)
            while (namebuf_off <= namelen)
            if (strncmp(pl_data->pld_name, namebuf, namelen == 0)
                pl_data->pld_offset = hdr_offset
                pl_flags &= PLF_FOUND
                return EOK
            else free namebuf

        bs_derefpg( pg_hdl )

pl_flags &= PLC_NOT_FOUND
return EOK
```

3.2.3.1.4 Exceptions

This routine could return EIO or any other error status from bs_refpg.

3.2.3.2 advfs_pl_name_hash

3.2.3.2.1 Interface

statusT
```
advfs_pl_name_hash( char *name,
                    int32_t name_len)
```

3.2.3.2.2 Description

This routine will calculate a simple hash of the name. The hash will be stored in the PL element header will and speed up searches of the PL file.

3.2.4 Index file support routines

3.2.4.1 advfs_pl_index_lookup

The index layout is to be designed and implemented at a later date.

3.2.4.2 advfs_pl_index_remove_element

The index layout is to be designed and implemented at a later date.

3.2.5 Getpage and Putpage modifications

3.2.5.1 advfs_getpage

3.2.5.1.1 Description

This routine will not need to change significantly. When testing for metadata, an OR condition will be added such that a file is metadata if bfap->dataSafety == BFD_LOG, bfap->dataSafety == BFD_METADATA *or* vp == bfap->pl_vp.

Additionally, calls to x_load_inmem_xtnt_map in the metadata write case and the read case will be skipped if vp == bfap->pl_vp since the pl file's extents will always be valid if a fault is occurring on the file.

No locking will be impacted because metadata is not locked by getpage and the PL file's extent maps will be protected by the pl_file_lock which will have been taken prior to the fault into getpage.

When making calls to advfs_get_blkmaps_in_range, advfs_getpage will conditionally pass the plXtntMap rather than the xtntMap for the file. The condition for using the plXtntMaps will be if pl_vp == VTOA(vp)->pl_vp. In other words, if the vp passed to putpage is the pl_vp of the bfap it is associated with, use the plXtntMap.

3.2.5.2 advfs_getmetapage

3.2.5.2.1 Description

If the PL files are modified to work with 4k allocation units, advfs_getmetapage will need to have asserts modified to allow for 4K metadata. When making calls to advfs_get_blkmaps_in_range, advfs_getmetapage will conditionally pass the plXtntMap rather than the xtntMap for the file. The condition for using the plXtntMaps will be if pl_vp == VTOA(vp)->pl_vp. In other words, if the vp passed to advfs_getmetapage is the pl_vp of the bfap it is associated with, use the plXtntMap.

3.2.5.3 advfs_putpage

3.2.5.3.1 Description

If the PL files are modified to work with 4k allocation units, advfs_putpage will need to have asserts modified to allow for 4K metadata. When making calls to advfs_get_blkmaps_in_range, putpage will conditionally pass the plXtntMap rather than the xtntMap for the file. The condition for using the plXtntMaps will be if pl_vp == VTOA(vp)->pl_vp. In other words, if the vp passed to putpage is the pl_vp of the bfap it is associated with, use the plXtntMap.

If 0,0 is passed in as the size and offset to flush for the PL file vnode, then the bfap->pl_file_size will provide the actual size to flush.

3.2.5.4 bs_refpg

This routine will be modified to use the bfap->pl_vp instead of the bfap->bfVnode if the MF_PL_FILE flag is set.

3.2.5.5 bs_pinpg

This routine will be modified to use the bfap->pl_vp instead of the bfap->bfVnode if the MF_PL_FILE flag is set.

3.2.6 Storage Subsystem Changes

3.2.6.1 Overview of transaction changes for multiple extent maps

Most of the storage subsystem will remain unchanged. The changes described below are primarily related to locking that is unnecessary with PL files, or with changes related to the fact that PL storage does not have a primary mcell.

3.2.6.2 stg_add_stg

3.2.6.2.1 Interface

```
sts = stg_add_stg(ftxHT parentFtx,           /* in */
             bfAccessT *bfap,                /* in */
             bf_fob_t fob_offset,            /* in */
             bf_fob_t fob_cnt,               /* in */
             stg_flags_t stg_flags,          /* in */
             bf_fob_t *alloc_fob_cnt         /* out */
             )
```

3.2.6.2.2 Description

This routine will be required to add storage to the PL file rather than always adding storage to the parent bfap. The routine will have the allowOverlapFlag changed to a stg_flags_t type which will have STGF_NO_OVERLAP or STGF_OVERLAP_OK set to model the previous behavior. If STGF_PL_FILE is passed in, the routine will add the requested storage to the PL file rather than the parent bfap. If STGF_PL_FILE is passed in, the bfap->xtnts->plXtntMaps will be used instead of the bfap->xtnts->xtntMap extents. Additionally, it is assumed that a caller passing in the STGF_PL_FILE flag has already handled locking the pl file extent maps.

Because storage addition and remove is exclusive for the PL file, and since all the storage for the PL extents is in a separate mcell chain, the taking and releasing of the xtntMap_lk and the mcelList_lk is not required. Additionally, the extent maps for the PL file will always be valid when acquiring storage since the advfs_pl_insert would have loaded them after taking the pl_file_lock for exclusive access. As a result, the PL file can be fast tracked through stg_add_stg to avoid x_load_inmem_xtnt_maps and all the locking. The add_stg_undo routine will clear reload the extent maps for the PL file if the transaction fails. Since the transaction would either fail before the pl_file_lock is released or be recovered during single threaded recovery, the locking is safe for the PL file storage.

When initializing the undo record for the add storage transaction, the routine will bitwise-OR in ADVFS_PL_FILE to the bfap's tag to indicate to the add_stg_undo routine that it must undo a storage addition to the PL file and not the regular storage.

3.2.6.3 add_stg

3.2.6.3.1 Interface

```
static
```

```
statusT
add_stg (
         bfAccessT *bfap,       /* in */
         bf_fob_t fob_offset,  /* in */
         bf_fob_t fob_cnt,     /* in */
         int allowOverlapFlag,  /* in */
         bsInMemXtntMapT *xtnts,   /* in, modified */
         ftxHT parentFtx,       /* in */
         int updateDisk,        /* in */
         bf_fob_t *alloc_fob_cnt /* out */
        stg_flags_t stg_flags        /* in */
         )
```

3.2.6.3.2 Description

This routine will be modified to take a bsInMemXtntMapT instead of a bsInMemXtntT. The call to imm_get_hole_size will now pass the xtntMap rather than the xtnts. The routine will also take a stg_flags parameter. If STGF_PL_FILE is set, then the checks hole_fob_cnt will be set to fob_cnt since the PL file will not be sparse.

3.2.6.4 new_mcell

3.2.6.4.1 Interface

```
statusT
new_mcell(
          ftxHT parFtx,          /* in - parent ftx */
          mcellUIdT* mcellUIdp, /* in/out - ptr to global mcell id */
          bsBfAttrT* bfAttr,    /* in - ds attributes */
          domainT* dmnP,        /* in - domain ptr */
          stg_flags_t stg_flags /* in – flags to indicate if this is the PL file */
          )
```

3.2.6.4.2 Description

This routine is currently only called by rbf_create and new_clone_mcell. The routine gets a new mcell and initializes it with a BSR_XTNTS (primary extent) record. The routine will be modified so that the oxtntp parameter is removed and replaced by a stg_flags_t parameter. If the stg_flag_t parameter is set to STGF_PL_FILE, then the record in the mcell will be initialized to a BSR_XTRA_XTNTS record instead of a BSR_XTNTS record. This routine will become the interface to get the first mcell for the PL file from advfs_pl_create.

3.2.6.5 add_stg_undo

This routine will be modified to check the sequence of the tag in the addStgUndoRec for the ADVFS_PL_FILE flag and take action on the PL file storage rather than the original file storage if necessary.

The call to bs_access will load both the parent bfap and the pl file. If the call to bs_access fails, or if ADVFS_PL_FILE is set in the tag_seq field of the undoRec.tag field and pl_vp is NULL, then the pl file was deleted and the routine can return. The locking of the xtntMap_lk is no longer necessary since either the pl_file_lock will be held exclusively if we are failing the transaction, or we will be recoverying and be in a single threaded environment.

If the PL file is being worked on, the xtnts used will come from the bfap->plXtntMap. Rather than invalidating extents, imm_delete_xtnt_map will be called on the plXtntMap and plXtntMap will be set to NULL. fcache_vn_invalidate will be called on the range being undone as in the normal case, however, a call to advfs_pl_access with the PLF_RELOAD_MAPS flag will be made instead of another call to x_load_inmem_xtnt_maps.

3.2.6.6 stg_remove_stg_start

3.2.6.6.1 Interface

```
statusT
stg_remove_stg_start (
                bfAccessT     *bfap,         /* in */
                bf_fob_t      fob_offset,    /* in */
                bf_fob_t      fob_cnt,       /* in */
                ftxHT         parentFtx,     /* in */
                uint32T       *delCnt,       /* out */
                void          **delList,     /* out */
                stg_flags_t   stg_flags      /* in */
                )
```

3.2.6.6.2 Description

This routine will be modified so that the relQuota flag is an enum stg_flags_t and the parameter will take either STGF_PL_FILE or STGF_REL_QUOTA. The STGF_REL_QUOTA will act just as the relQuota flag did previously. The STGF_PL_FILE flag will indicate that the stg code should be dealing with storage for the PL file and not the bfap passed in. The doCow flag will also be rolled into the stg_flags parameter as the flag STGF_DO_COW. This may become obsolete when the snapshot design is completed, but will be left for now.

It is expected that, until PL files support sparse storage, cowing of PL data will happen with the rest of the metadata for a file, thus, this routine does not need to handle cowing of PL data.

The routine will handle storage removal much like stg_add_stg handles the addition of storage. The locking and unlocking of the mcelList_lk and the xtntMap_lk is not required, nor is calling x_load_inmem_xtnt_map, when dealing with PL files. The call to remove_stg will be modified to pass bfap->xtnts->plXtntMap rather than the bfap->xtnts.

The call to merge_xtnt_maps will be replaced with a call directly to merge_sub_xtnt_maps passing the plXtntMap pointer, and the tag of the rmStgUndoRec will be initialized so that the sequence has the ADVFS_PL_FILE bit set. These changes will only be for cases when the STGF_PL_FILE flag is set in stg_flags.

3.2.6.7 stg_remove_stg_finish

This routine should not need to be modified. It will be processing the storage associated with the PL file, but that storage will be on the DDL and will look like a normal truncation.

3.2.6.8 del_dealloc_stg

This routine should not need to be modified. It will be processing the storage associated with the PL file, but that storage will be on the DDL and will look like a normal truncation.

3.2.6.9 remove_stg

3.2.6.9.1 Interface

```
static
statusT
remove_stg (
            bfAccessT *bfap,        /* in */
            bf_fob_t fob_offset,  /* in */
            bf_fob_t fob_cnt,     /* in */
            bsInMemXtntMapT *xtntMap,   /* in, modified */
            ftxHT parentFtx,        /* in */
            bf_fob_t *del_fob_cnt,   /* out */
            uint32T *delCnt,        /* out */
            mcellPtrT **delList     /* out */
```

)

3.2.6.9.2 Description

This routine will be modified to take a bsInMemXtntMap instead of a bsInMemXtnts structure. The PL file's extent maps will be passed in when being truncated and the file destroyed. This routine will return after having put the chain of mcells described by the xtntMap on the DDL. Because the DDL requires a primary mcell so that the delRstT structure exists for recovery, remove_stg may create a pseudo primary mcell for the delete. For normal files, a psuedo primary mcell is only required for truncations. As a result, the removal of all storage for a PL file will behave like a truncation for a normal file.

3.2.6.10 remove_stg_undo

Much like add_stg_undo, this routine will check the undoRec.tag.tag_seq for the ADVFS_PL_FILE flag to determine whether the storage removal to be undo is associated with the PL file or the regular file. The call to bs_access will load the pl file and set pl_vp if the PL file exists. If either the call to bs_access fails, or pl_vp is not set after the call, then the pl file has been deleted and the routine can return.

If the pl_vp is set after the call to bs_access and if tag_seq & ADVFS_PL_FILE, then the extents will be taken from the bfap->xtnts->plXtntMap. The call to imm_delete_xtnt_map will pass in the plXtntMap. When dealing with the plXtntMap, the xtntMap_lk does not need to be acquired.

Once the in memory plXtntMap is cleared, a call to advfs_pl_access will be made with the PLF_RELOAD_MAPS flag to indicate that the extent maps should be read in, but the vnode already exists. This is necessary so that the pl_vp does not change if a user is looking at the PL file through .tags.

3.2.6.11 merge_xtnt_maps

This routine will be removed and merge_sub_xtnt_maps will be called directly passing either the xtntMap or plXtntMap bsInMemXtntMapT pointer accordingly.

3.2.7 Access structure interaction

3.2.7.1 Access structure Init

At initialization time (bs_access_one and bs_map_bf), a call will be made to advfs_pl_access without the create flag. This will map in the PL file if it exists and allow it to be ready for access when required. Whenever the bfAccess structure is open and a PL file exists, the pl_vp will be initialized and valid. If the pl_vp is accessed through the .tags directory, the pl_vp will have a v_count on it for each open of the file. When the pl_vp is opened because of the bfap being opened, the v_count will be 0.

After having initialized the PL file via a call to advfs_pl_access, a call will be made to advfs_getacl. This call will initialize the ACL cache in the bfap and will make later calls to advfs_access faster.

3.2.7.2 Access dealloc and recycle

At deallocation and recycle time, the pl_vp's v_count field should be 0 unless the PL file is being access via .tags. If the v_count is 0, then the pl_vp will be freed just before returning the bfap to the memory arena or before recycling the bfap. If the pl_vp exists, the vnode will be flushed and invalidated before returned the vnode to the arena.

advfs_process_list will flush and invalidated the pl_vp if it exists before recycling the access structure. This flush and invalidate will be safe from racing threads dirtying the pages because the bfap will already be in state ACC_RECYCLE and access to the bfap would be required before access to the pl_vp would be granted.

advfs_dealloc_access will flush and invalidate the pl_vp before deallocating the bfap. This flush and invalidate will be safe from racing threads dirtying the pages because the bfap will already be in state ACC_DEALLOC and access to the bfap would be required before access to the pl_vp would be granted.

advfs_access_mgmt_thread will flush the pl_vp before moving the bfap from the closed list to the free list and will invalidate the pl_vp before freeing the bfap to the arena. This flush and invalidate will be safe from racing threads dirtying the pages because the bfap will already be in state ACC_VALID_EXCLUSIVE when moving from the closed list to the free list and access to the bfap would be required before access to the pl_vp would be granted. When invalidating and moving from the free list to the arena, the routine advfs_dealloc_access will be used and will handle races.

Before deallocating or recycling the bfap, the memory consumed by the ACL cache will be freed.

3.2.7.3 Access structure/file deletion

When an access structure is deleted, the PL file will also be deleted. Freeing the access structures mcells will be sufficient to cleanup the PL file, but the storage must also be freed. To synchronize with deletion, advfs_pl_close will be passed with the PLF_REMOVE_FILE flag and the transaction handle for the delete. The call to advfs_pl_close will move the storage to the DDL. On return from the call to advfs_pl_close (PLF_REMOVE_FILE), bs_close_one will ftx_done its close transaction and then remove the storage associated with the file itself and call stg_remove_stg_finish on the mcell_del_list and mcell_del_cnt parameters returned from advfs_pl_close. If the transaction were to fail, the redo of the close would cause the file's data storage to be cleaned up and the processing of the DDL would cause the PL file's storage to be cleaned up.

3.2.8 Transaction management

3.2.8.1 Overview of transaction changes for multiple extent maps

Transactions dealing with PL elements and the PL file storage must be aware of the fact that the storage modified under transaction was the storage described by a file's plXtntMap extents and not the normal bfap->xtnts->xtntMap extents. Although the PL file does not have it's own tag (it is just an extension of the parent file's metadata), the image redo code must be made aware of which storage is to be modified. Tru64 had a 32 bit tag and 32 bit sequence. On HPUX, the tag will be 64 bits and the sequence will be extended to 64 bits. The high order bits of the sequence will be used to indicate that the storage is associated with the PL file and not the parent bfap.

3.2.8.2 rbf_pinpg

rbf_pinpg will be modified to return E_MX_PINP_EXCEEDED when the maximum number of pages pinnable in a single transaction is exceeded. This change has two significant advantages. First, it simplifies processing adjacent PL blocks in insert and remove, and it removes a potential source of a system wide panic on an exceptional transaction error. rbf_pinpg will now return an error causing a transaction to fail and be propogated up, rather than bringing down the entire system.

This routine will also now set the lvl_flags LF_PL_FILE flag in the lvlPinTblT structure to indicate that the page pinned was for the PL file. This flag need only be set the first time the page is pinned for this transaction (or sub transaction). If the page handle is already found in the per level page table, the flag will already be set.

When setting the tag in the ftxPinTblT.ftxRecRedoT structure, the ADVFS_PL_FILE flag will be set in the tag sequence number to indicate to ftx_recovery_pass that it is dealing with the PL file when applying image redo records.

3.2.8.3 rbf_can_pin_record

This is a new routine being introduced to maximize records pinned per page. The routine will take a page handle (rbfPgRefHT) and will return a positive number if the page can still have more pages pinned without exceeding FTX_MAX_PINR.

3.2.8.4 ftx_bfdata_recovery_pass

This routine will be modified to handle redoing transactions that occurred on the PL file. The basic change to this routine is to check the pageredoT structure's tag.seq field to see if the ADVFS_PL_FILE flag is set. If it is set, then the redo operations will be the PL file pages rather than the regular file pages.

The assert which verifies the file is metadata will be modified to assert that either dataSafety == BFD_METADATA || pgredop->pgdesc.tag.seq & ADVFS_PL_FILE. The call to bs_access will open the pl file if it exists, so it is not necessary to do a advfs_pl_access; however, since the pl file may have been deleted, if ADVFS_PL_FILE is set in the tag.seq, the bfap->pl_vp will be checked. If NULL, we will just return.

The call to bs_pinpg will conditionally pass the MF_PL_FILE flag to indicate that the page to pin is in the PL storage of the file.

3.2.9 .tags lookup

3.2.9.1 Overview

Access of the pl file through .tags will allow for direct reads of the PL file storage (not direct IO). Writes to the pl storage will not be allowed through .tags. Mmapping of PL storage will also not be allowed. Manipulation of the PL storage through .tags will be strictly limited to reads.

Access of the PL file through .tags will be accomplished by appending "PL" to the end of any tag that could normally be accessed through .tags (reserved metadata is valid and can be accessed as M7PL). Opening of the PL file through .tags will be semantically the same as accessing the parent file, except that the vnode returned will be the vnode of the PL file rather than the vnode of the parent. The parent bfap will still be referenced and will look as if it has been externally opened. The vnode of the PL file will also have a v_count for opens through .tags. On last close of the PL vnode, the v_count will be 1 and advfs_inactive will be called to decrement the v_count. When advfs_inactive is called on the vnode of the PL file, the parent file will be closed and the v_count of the PL file will be decremented to 0.

To guarantee that PL files are not accidentally accessed incorrectly via .tags, the PL vnode will have a advfs_pl_vnodeops structure that uses a function that returns E_NOT_SUPPORTED for everything but read, close, open and inactive, and lookup.

3.2.9.2 advfs_lookup

advfs_lookup will not need to be changed significantly to support .tags access of the PL file. In the block of code that deals with .tags lookups, the string will be checked for a suffix of "PL." If the suffix exists, it will be stripped from the name and a flag will be set to indicate to the routine that this is a PL file access.

In the two places where advfs_lookup sets ndp->ni_vp to found_vp, a check will be made. If flag set in the .tags lookup path was set, then the ni_vp will be assigned VTOA(found_vp)->pl_vp. If there is no pl_vp for the file, then the parent file will be closed and the function will return as if the the file had not been found. If pl_vp is set, then the vnode lock will be acquired and the v_count will be incremented.

3.2.9.3 advfs_inactive

advfs_inactive will be modified to check if the vnode passed in is a PL vnode. If the vnode is a PL vnode, then the vnode lock will be acquired and the v_count decremented. The routine will then call VN_RELE on the parent vnode (VTOA(vp)->bfVnode). The call to VN_RELE will cause the v_count on the parent to be decremented and may cause another call to advfs_inactive on the parent file. In the PL file case, advfs_inactive will return after calling VN_RELE.

3.2.10 Migrate Impact

3.2.10.1 Overview of changes

Migrate will operate in the same basic manner for PL files as it does for regular file storage. The migration of PL file data will happen with a flag to the migrate routines to indicate that the PL file extents should be migrated rather than the normal file data. The basic algorithm for migrate is to allocate storage at the target for migration, fault in the storage to be migrated, then flush the data to the new and old storage if necessary. The flushing of the data requires the copy extents to exist for the new storage. Once the storage is flushed, the storage is swapped and the copy extents are overlayed over the original extents.

Because the copy extents are required, the migration of the PL file storage and the migration of the parent file storage will be synchronized. To synchronize between the migrations, the migStgLk will be taken in write mode for either migration. Unlike migrating the parent file regular extents storage, migrating the PL file storage will only required the pl_file_lock and the migStgLk, not the xtntMap_lk and the mcellList_lk.

In the migration step where the storage is swapped from the copy extents to the original extents, the transaction that handles the storage swap will also update the PL file record if the first fob of the PL file is being migrated (if the migrate starts at fob 0).

3.2.10.2 advfs_real_syscall

The routine will get the vnode from the file descriptor and use this to specify the storage to be migrated. This will allow the PL data file to be migrated by referencing the file via .tags.

3.2.10.3 msfs_syscall_op_migrate

This routine will be modified to take a vnode instead of a bfap. The vnode will be checked to see if it is the pl_vp. If the vnode is a PL file vnode, the call to bs_migrate will pass the MIG_PL_FILE flag.

3.2.10.4 bs_migrate

This routine will be modified to take a mig_flags_t parameter instead of a forceFlag parameter. The previous forceFlag will be converted into the mig_flags_t MIG_FORCE flag. The MIG_PL_FILE flag will indicate that the storage to be migrated is the PL file storage and not the regular file storage.

This routine will check the src_fob_offset and src_fob_cnt for alignment on the PL file's allocation unit if the migrate is for the PL file.

3.2.10.5 mig_migrate

This routine will be modified to take a mig_flags_t parameter instead of a forceFlag parameter. The previous forceFlag will be converted into the mig_flags_t MIG_FORCE flag. The MIG_PL_FILE flag will indicate that the storage to be migrated is the PL file storage and not the regular file storage.

This routine will conditionally call x_load_inmem_xtnt_map or lock the pl_file_lock for write depending on whether the the MIG_PL_FILE flag is set. If the pl_file_lock is taken, it will be unlocked where the xtntMap_lk would otherwise have been unlocked.

The srcXtntMap will be taken from either the bfap->xtntMap or the plXtntMap depending on whether the MIG_PL_FILE flag is set.

3.2.10.6 migrate_normal

This routine will be modified to take a mig_flags_t parameter instead of a forceFlag parameter. The previous forceFlag will be converted into the mig_flags_t MIG_FORCE flag. The MIG_PL_FILE flag will indicate that the storage to be migrated is the PL file storage and not the regular file storage.

If MIG_PL_FILE is set, this routine does not need to take an active range since the pl_file_lock will already be taken. The migStgLk will synchronize with all migrates of the normal storage so that it is safe to use the copyXtntMap. Additionally, for MIG_PL_FILE, the call to x_load_inmem_xtnt_map can be skipped.

The routine will verify that the src_fob_offset and src_fob_cnt are multiples of the pl_alloc_sz if MIG_PL_FILE. The call to advfs_get_blkmap_in_range to get block maps for the file will pas the bfap->xtnts.plXtntMap if the MIG_PL_FILE flag is set.

The call to mig_fill_cache will be modified to pass in the MIG_PL_FILE which will cause the calls to fcache_* routines to use the bfap->pl_vp rather than the bfap->bfVnode. This will cause the PL file storage to be brought into cache in preparation for the migrate.

The call to fcache_vn_flush will pass the bfap->pl_vp if MIG_PL_FILE is set. This will cause the storage to be flushed to the new copy storage.

The call to switch_stg will be passed the MIG_PL_FILE flag to indicate that the PL record must be updated if the 0th fob is changed. Additionally, if MIG_PL_FILE is set, then the bfap->xtnts.plXtntMap will be passed rather than the bfap->xtnts.xtntMap.

3.2.10.7 migrate_normal_one_disk

This routine will be modified to take a mig_flags_t parameter instead of a forceFlag parameter. The previous forceFlag will be converted into the mig_flags_t MIG_FORCE flag. The MIG_PL_FILE flag will indicate that the storage to be migrated is the PL file storage and not the regular file storage.

migrate_normal_one_disk is implemented on top of migrate_normal. As a result, this routine will pass the MIG_PL_FILE down to migrate_normal when calling to migrate a subrange of the file. This routine currently sets xtntMap to bfap->xtnts.xtntMap. The local variable will be conditionally set to bfap->xtnts.plXtntMap if MIG_PL_FILE is set. In the case of the MIG_PL_FILE, the pl_file_lock will have been acquired before calling this routine and the xtntMap_lk will not be required to be locked.

3.2.10.8 switch_stg

This routine will be modified to take a mig_flags_t parameter. The flag may be set to MIG_PL_FILE when the PL file is being migrated.

This routine will remain essentially unchanged with one caveat: if MIG_PL_FILE is set and the *origXtntMapAddr extent map maps fob 0, then the bsr_pl_rec record in the bfap will need to be transactionally updated to point to the mcell of the copy extent map. This transactional update will occur just before the final ftx_done in the routine. The record will be pinned and brought into cache as part of a sub transaction.

3.2.11 Miscellaneous Changes

3.2.11.1 advfs_handyman_thread

This routine will be modified to accept a message of type AHME_DELETE_PL_ELEMENT. The message will contain the tag of a file and an offset into the PL file of that tag that is in an invalid state and should be removed. Typically, the message will be sent when a remove fails following a large insertion. If an insert of a PL element requires that a previous element be deleted, and if the delete fails because of transaction limitations, the old element will be marked invalid and a message will be sent to the advfs_handyman_thread to clean up the invalid element. This thread may also receive a message from advfs_pl_lookup if lookup comes across an invalid element that was not previously removed because of a crash. The lookup will trigger an asynchronous cleanup of any invalid elements it finds.

The message to the handyman thread will consist of the offset of the PL element, the tag of the parent file (including the bfSetId), and the name of the PL element.

Because this routine will most commonly be called following a large insert, the routine will always synchronize with large PL insertions by using domain's pl_cv_mutex and the domain's pl_cv.

3.2.12 ACLs interface

3.2.12.1 VOP_SETACL (advfs_setacl)

Interface

```
int
advfs_setacl(  struct vnode*          vp,             /* File to set ACL on */
               int                    ntuples,        /* Number of ACL entries requested */
               struct acl_tuple_user* tupleset,       /* Buffer with new ACL entires */
               int                    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 parent file (cannot take the vnode of the property list file), 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".

The user command, setacl, will handle a few things for the kernel routine before it is called. 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 the following routine. 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 property list interactions, the 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 system. The file system must not be read-only.

ACL information may be stored as an AdvFS property list element in a property list file hidden to the user. This information will be inserted or deleted from this property list subsystem using the advfs_pl_insert and advfs_pl_delete routines. If the number of ACL entries is less than or equal to ADVFS_BMT_ACL_THRESHOLD, then the ACL will be stored in a BMT record.

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 BMT, the property list entry for the ACL can simply be removed. A pl_data structure must be initialized with the SYSV_ACLS name key and the strlen of this key. Also, the pl_flags must indicate that the SYSV_ACLS entry for this file will be deleted from the property list by setting the flag to PLF_DELETE and PLF_ACL. A call to advfs_pl_delete with the initialized pl_data and pl_flags will remove the SYSV_ACLS entry from the property list. Nothing further is necessary since base modes are already stored in the BMT. However, the BMT will need to be checked for any ACL record. If an ACL record exists in the BMT, it will be marked as invalid.

The next order of operation is to set up a new AdvFS specific ACL buffer that will be passed to the property list interface for insertion. The advfs_acl_tuple data structure will be used. The space for all ntuples will be allocated and initialized using the advfs_acl_unused constant for this new buffer. Then, 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 nopttuples (number of optional tuples) is incremented.

At this point, the start of a new transaction, FTA_UPDATE_ACL with handle ftxAcl, will guarantee that the next few steps either complete or do not complete.

If there were no optional tuples specified, then the property list entry will be deleted. There is no need for the PL entry if only the base modes are stored. Check the BMT for an ACL record and mark it invalid if it exists. If optional tuples were specified, pl_data and pl_flag will need to be prepared to insert the newly constructed AdvFS ACL buffer into the property list. The pl_flags will indicate PLF_INSERT and PLF_ACL. The uio structure of pl_data will need the name field initialized to indicate SYSV_ACLS and the strlen of this type will be inserted in the name length. The offset will be zero to indicate there is no header information at the beginning of the uio vector. The property list insert routine can simply insert the information as given with no offset. The length and resid will be the size of the name key length added to the size of ntuple number of advfs_acl_tuple structures. The iov_base will include the key name "SYSV_ACLS" and the new ACL buffer. Once these initializations are made, a call to routine advfs_pl_insert can be made to add the new ACL information to the property list file. If there is an error during the insertion, status is returned in sts.

The acl_cache pointer in the bfap will be freed and the advfs_acl buffer will be copied into a new buffer for the 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.

The routine returns EOK when successful.

Execution Flow

```
struct acl_tuple_user *tp;
struct advfs_acl_tuple *advfs_acl;
struct pl_data_t *pl_data;
pl_flags_t pl_flags;
struct fsContext *contex_ptr;
int chkbase=0, imode=0, nopttuples=0, mcellListPage=0;

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

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

/* 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;
}

/*
 * Make sure the vnode is the bfVnode of the bfap it is associated with
 * and not a pl_vp
 *
 * If a pl_vp, return E_BAD_HANDLE
 */

/* 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
 */

/* Initialize pl_data */
pl_data = ( pl_data_t *)malloc( sizeof( pl_data_t ) );
pl_data->pld_element.plie_name_len = strlen( ADVFS_ACL_TYPE );
pl_data->pld_element.plie_name = strdup( ADVFS_ACL_TYPE );

/* Special case:  If ntuples is less than zero, delete the ACL from the PL */
if( ntuples < 0 ) {
        pl_flags = PLF_DELETE | PLF_ACLS;
        sts = advfs_pl_delete( bfap, pl_data, &pl_flags, FtxNilFtxH );
        if( sts != EOK ) {
                u.u_error = sts;
                return sts;
        }


        Check the BMT for a BSR_BMT_ACL record, if one exists AND is valid,
        Start a transaction, mark the record as invalid by setting bba_acl_cnt to -1,
        and finish the transaction.

        /* Remove the ACL cache if it exists */
        /* LOCK bfap */
        if( bfap->acl_cache != NULL )
                free( bfap->acl_cache );
        /* UNLOCK bfap */

        return EOK
}

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

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

        /* Are these valid tuples? */
        if( tp->uid >= MAXUID || tp->gid >= MAXUID ||
          ( tp->uid < 0 && tp->uid != ADVFS_ACL_BASE_USER ) ||
          ( tp->gid < 0 && tp->gid != ADVFS_ACL_BASE_GROUP ) ||
          ( tp->mode > 7 ) ) {
                u.u_error = EINVAL;
                return EINVAL;
        }

        /*
         * BASE TUPLES:
         * In the following section, we have found one of the base tuples
         */
        if( tp->uid == ADVFS_ACL_BASE_USER ) {

                if( tp->gid == ADVFS_ACL_BASE_GROUP ) {              /* Case: (*,*) */
                        /* Error if it has already been set */
                        if( chkbase & (1<<ADVFS_ACL_OTHER ) ) {
                                u.u_error = EINVAL;
                                return EINVAL;
                        }
                        chkbase |= 1<<ADVFS_ACL_OTHER;
                        /* Set other base mode bits */
                        imode = advfs_setbasemode( imode, tp->mode, ADVFS_ACL_OTHER );
                        advfs_acl[i].uid = tp->uid;
                        advfs_acl[i].gid = tp->gid;
                        advfs_acl[i].mode = tp->mode;
                        continue;
                } else {                                             /* Case: (*,g) */
                        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->mode, ADVFS_ACL_GROUP );
                        advfs_acl[i].uid = tp->uid;
                        advfs_acl[i].gid = tp->gid;
                        advfs_acl[i].mode = tp->mode;
                        continue;
                }
        } else {

                if( tp->gid == ADVFS_ACL_BASE_GROUP ) {               /* Case: (u,*) */
                        if( chkbase & ( 1<<ADVFS_ACL_USER ) ) {
                                u.u_error = EINVAL;
                                return EINVAL;
                        }
                        chkbase |= 1<<ADVFS_ACL_USER;
                        /* Set the user base mode bits */
                        imode = advfs_setbasemode( imode, tp->mode, ADVFS_ACL_USER );
                        advfs_acl[i].uid = tp->uid;
                        advfs_acl[i].gid = tp->gid;
                        advfs_acl[i].mode = tp->mode;
                        continue;
                }
        }

        /*
         * OPTIONAL TUPLE:            Case: (u,g)
         * In the following section we have found an optional tuple
         */
        advfs_acl[i].uid = tp->uid;
        advfs_acl[i].gid = tp->gid;
        advfs_acl[i].mode = tp->mode;
        nopttuples++;
}

/*
         * START TRANSACTION
         *
         * Guarantee that all of the next steps either happen or do not happen:
         * 1. New ACLs are set in the BMT or the property list (depending on number of entries)
         *     or PL entry is deleted
         * 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
         */
FTX_START_N( FTA_ACL_UPDATE, &ftxAcl, FtxNilFtxH, bfap->dmnP );

/*
         * There is no need to keep the PL entry if no optional tuples were specified
         * We'll change the mode bits and the BMT later for the base tuples
         */
if( nopttuples == 0 ) {
        pl_flags = PLF_DELETE | PLF_ACLS;
        sts = advfs_pl_delete( bfap, pl_data, &pl_flags, ftxAcl );
        if( sts != EOK ) {
                ftx_fail( ftxAcl );
                u.u_error = sts;
                return sts;
        }


        Check the BMT for an BSR_BMT_ACL record, if one exists AND is valid,
        transactionally mark it as invalid by setting bba_acl_cnt to -1.


        /* LOCK the bfap */
        if( bfap->acl_cache != NULL )
                free( bfap->acl_cache );
        /* UNLOCK the bfap */
```

```
} else {
        if( ntuples <= ADVFS_BMT_ACL_THRESHOLD ) {
                Check to see if a BMT record for ACLs exists, if not, create one
                and add the advfs_acl buffer to the record.  If a record does exist,
                reset the record with the new acl (and update the bba_acl_cnt to reflect
                the record is now valid).  This will be done transactionally using
                rbf_pinpg and rbf_pin_record.

        } else {

                /* Prep pl_data and pl_flags for insertion of the new ACL into the PL*/
                pl_flags = PLF_INSERT | PLF_ACLS;
                pl_data->uiop->uio_iov->iov_len =
                    strlen( ADVFS_ACL_TYPE ) + ( sizeof(advfs_acl_tuple) * ntuples);
                pl_data->uiop->uio_resid =
                    strlen( ADVFS_ACL_TYPE ) + ( sizeof(advfs_acl_tuple) * ntuples);
                pl_data->uiop->uio_offset = 0;
                pl_data->uiop->uio_iov->iov_base = advfs_acl;

                /* insert the new ACL */
                sts = advfs_pl_insert( bfap, pl_data, &pl_flags, ftxAcl );
                if( sts != EOK ) {
                        ftx_fail( ftxAcl );
                        u.u_error = sts;
                        return sts;
                }
        }
}

/*
 * 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,
                        ftxAcl,
                        MF_VERIFY_PAGE );
if( sts != EOK ) {
        ftx_fail( ftxAcl );
        u.u_error = sts;
        return sts;
}
mcellp = &bmtp->bsMCA[bfap->primMcell.cell];

MCELLIST_LOCK_WRITE( &(bfap->mcellList_lk ) );
fsStats = (struct fs_stat *)bmtr_find( mcellp, BMTR_FS_STATS, bfap->dmnP );
if( fsStats == NULL ) {
        ftx_fail( ftxAcl );
        u.u_error = ENOSYS;
        return ENOSYS;
}
fsStats->st_mode = imode;
MCELLIST_UNLOCK( &(bfap->mcellList_lk ) );

/* 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 */
/* LOCK THE bfap */
free( bfap->acl_cache );
bfap->acl_cache = malloc( sizeof( advfs_acl_tuple ) * ntuples );
bcopy( advfs_acl, bfap->acl_cache, ( sizeof( advfs_acl_tuple ) * ntuples ) );
bfap->acl_cache_len = sizeof( advfs_acl_tuple_t ) * ntuples;
```

```
/* UNLOCK THE bfap */

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

/*
 * Free up all space allocated for the pl_data structure
 */

return EOK;
```

Exceptions

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

3.2.12.2 VOP_GETACL (advfs_getacl)

Interface

```
int
advfs_getacl(  struct vnode*         vp,           /* File to get ACL on */
               int                   ntuples,      /* Number of ACL entries requested */
               struct acl_tuple_user* tupleset,     /* Buffer of new ACL entries */
               int                   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 the parent file (cannot take the vnode of the property list 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 either the property list file or the BMT (for base tuples if no ACLs are in the property lists).

ACL information will either be stored as an AdvFS property list element in a property list file hidden to the user, or if it is less that ADVFS_BMT_ACL_THRESHOLD, the ACL will be stored in an ACL MBT record. If this information was previously queried, then it should be stored in the bfap acl_cache. The advfs_acl can simply copy the data in the acl_cache in the bfap to a new buffer. If this cache pointer is null, then a look up on disk is necessary. If there is a BMT record entry for an ACL, they will be filled intot he advfs_acl buffer, if not, this ACL information will be queried from the property list subsystem using the advfs_pl_lookup routine. Both PLF_LOOKUP and PLF_ACL flags will be set in the pl_flags field which indicates that the uio structure is a kernel memory address and a bcopy will need to be done instead of a uio move. Several fields in the pl_data structure will be initialized before the call. The name key will be set to "SYSV_ACLS" and the length will be string length of "SYSV_ACLS", the uio structure will need an offset of zero to indicate that there is no header, and the information should start at the beginning of the uio vector. The length of the data will need to be big enough to accommodate the largest possible ACL (sizeof an AdvFS ACL tuple * 1024), and the uio_resid field will be the same as the uio vector length.

The ACL tuples will be returned in the uio vector base field and the pl_data->pld_data pointer will point to where the actual data begins in the uio vector. The advfs_acl pointer will be set to point to the data pointer of the pl_data structure, and this data will be copied into a new buffer pointed to by the bfap's acl_cache. If pl_flags indicates PLF_NOT_FOUND, then the base tuples will be retrieved from the BMT since there is no property list entry for ACLs at this point. Base tuples will be inserted into advfs_acl.

The tuples are finally plugged into the tupleset. 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 less than the number requested, then we will return and error, EINVAL.

The routine returns EOK when successful.

Execution Flow

```
struct advfs_acl_tuple *advfs_acl;

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

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

/*
 * Make sure the vnode is the bfVnode of the bfap it is associated with
 * and not a pl_vp
 *
 * If a pl_vp, return E_BAD_HANDLE
 */

/* 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( ( ntuples > ADVFS_MAX_ACL_ENTRIES ) || ( ntuples < 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->acl_cache == NULL ) {

        if (BMT has a BSR_BMT_ACL record and its valid)
                Read the ACL from the record and plug it into the advfs_acl buffer.
        else {
                /*
                 * Initialize the pl_data structure and set the
                 * pl_flags to indicate we are looking up ACLs
                 */
                pl_data = (pl_data_t *)malloc( sizeof( pl_data_t ) );
                pl_data->pld_element.plie_name_len = sizeof( ADVFS_ACL_TYPE );
                pl_data->pld_element.plie_name = strdup( ADVFS_ACL_TYPE );
                pl_data->uiop->uio_iov->iov_len =
                        strlen( ADVFS_ACL_TYPE ) + ( sizeof( advfs_acl_tuple ) *
                        ADVFS_MAX_ACL_ENTRIES );
                pl_data->uiop->uio_resid =
                        strlen( ADVFS_ACL_TYPE ) + ( sizeof( advfs_acl_tuple ) *
                        ADVFS_MAX_ACL_ENTRIES );
                pl_data->uiop->uio_offset = 0;
                pl_data->uiop->uio_iov->iov_base =
                        malloc( strlen( ADVFS_ACL_TYPE ) +
                                  sizeof( advfs_acl_tuple ) * ADVFS_MAX_ACL_ENTRIES ) );
                pl_flags = PLF_LOOKUP | PLF_ACLS;

                /* Request ACL from AdvFS property list file */
                sts = advfs_pl_lookup( bfap, pl_data, &pl_flags );
                if( sts != EOK ) {
                        .u_error = sts;
                        return sts;
                }

                /*
                 * Make sure something was found, if not, check the BMT for the
                 * base tuples
                 */
                if( pl_flags & PLF_NOT_FOUND ) {

                        /*
                         * 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,
                                          bfap->dmnP->vdpTbl[bfap->primMcell.volume]->bmtp,
                                          BS_NIL,
                                          ftxAcl,
                                          MF_VERIFY_PAGE );
                        if( sts != EOK ) {
                                ftx_fail( ftxAcl );
                                u.u_error = sts;
                                return sts;
                        }
                        mcellp = &bmtp->bsMCA[bfap->primMcell.page];

                        /* Get the file's mode from the fs_stat structure */
                        MCELLIST_LOCK_READ( &(bfap->mcellList_lk ) );
                        fsStats = (struct fs_stat *)bmtr_find( mcellp, BMTR_FS_STATS,
                                bfap->dmnP);
                        if( fsStats == NULL ) {
                                ftx_fail( ftxAcl );
                                u.u_error = ENOSYS;
                                return ENOSYS;
                        }
                        fmode = fsStats->st_mode;
                        MCELLIST_UNLOCK( &(bfap->mcellList_lk ) );

                        /*
                         * Malloc space for the base tuples in the ACL entry, we'll
                         * need four: user, group, other, and class entries
                         */
                        advfs_acl = malloc( sizeof( advfs_acl_tuple ) *
                                ADVFS_NUM_BASE_ENTRIES );
                        tuple_cnt = ADVFS_NUM_BASE_ENTRIES;

                        /*
                         * Add the base tuples to the advfs_acl buffer
                         */

                } else {

                        /* Make sure we got correct information from our property list */
                        ASSERT( strncmp( pl_data->uiop->uio_iov->iov_base, ADVFS_ACL_TYPE,
                                strlen( ADVFS_ACL_TYPE ) ) == 0 );
                        advfs_acl = pl_data->pld_element.plie_data;
                        tuple_cnt = pl_data->pld_element.plie_data_len /
                                sizeof(advfs_acl_tuple);
                }
        }

        /* Set the bfap acl cache with the new ACL */
        /* LOCK THE bfap */
        bfap->acl_cache = malloc( sizeof( advfs_acl_tuple ) * tuple_cnt );
        bcopy( advfs_acl, bfap->acl_cache, ( sizeof( advfs_acl_tuple ) * tuple_cnt ) );
        bfap->acl_cache_len = tuple_cnt;
        /* UNLOCK THE bfap */

} else {
        /* Get ACL from bfap acl_cache */
        advfs_acl = malloc( sizeof( advfs_acl_tuple ) * tuple_cnt );
        bcopy( bfap->acl_cache, advfs_acl, ( sizeof( advfs_acl_tuple ) * tuple_cnt ) );
        tuple_cnt = bfap->acl_cache_len;
}

if( ntuples != 0 ) {

        /* Error if number of tuples requested is less than number in the ACL */
        if( ntuples < tuple_cnt ) {
                u.u_error = EINVAL;
                return EINVAL;
        }

        /* initialize tupleset */
        tupleset = (acl_tuple_user *)malloc( sizeof(acl_tuple_user) * tuple_cnt );
        advfs_init_acl( tupleset, tuple_cnt );

        /* plug the tuples into the tupleset */
        for( i = 0; i < tuple_cnt; i++ ) {
                tupleset[i].uid = advfs_acl[i].uid;
                tupleset[i].gid = advfs_acl[i].gid;
                tupleset[i].mode= advfs_acl[i].mode;
        }
}

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

/*
 * Free up all space allocated for the pl_data structure and advfs_acl
 */

return EOK;
```

Exceptions

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

3.2.12.3 advfs_access

Interface

```
void
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 (cred). The routine first sets the statistic counter for the routine using FILESETSTAT, and then loads a 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 no 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 );
mode >>= 6;

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

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

Exceptions

Could return EROFS, ETXTBSY, or EACCES

3.2.13 ACLs support routines

3.2.13.1 advfs_init_acl

Interface

```
void
advfs_init_acl( struct acl_tuple_user*       acl,          /* Buffer for ACL entries */
                int                          ntuples       /* Num of tuples to init */
                )
```

Description

This routine initializes the acl buffer before it is used by AdvFS acl routines. It loops through the tuple set, initializing each tuple to the global acl_tuple_unused.

Execution Flow

```
int i;

for( i = 0; i < ntuples; i++ ) {
        acl[i] = acl_tuple_unused;
}
```

Exceptions

None

3.2.13.2 advfs_init_acl

Interface

```
void
advfs_init_acl_internal( struct advfs_acl_tuple *    advfs_acl,      /* Buffer for ACL */
                         int                         ntuples         /* Num of tuples */
                         )
```

Description

This routine initializes the acl buffer before it is used by AdvFS acl routines. It loops through the acl buffer, initializing each tuple to the global advfs_acl_tuple_unused. This is an internal routine because an AdvFS specific data structure is used for the ACL buffer.

Execution Flow

```
int i;

for( i = 0; i < ntuples; i++ ) {
       advfs_acl[i] = advfs_acl_tuple_unused;
}
```

Exceptions

None

3.2.13.3 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 called "acl" by calling advfs_getacl. The maximum possible of acl entries, ADVFS_MAX_ACL_ENTRIES as well as the ACL type (ADVFS_ACL_TYPE) will be passed in. The actual sorted ACL is returned 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 );

/* 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 );

/* 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;
}

/*
 * 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->acl_cache != NULL ) {
        acl = bfap->acl_cache;
        tuple_cnt = bfap->acl_cache_len;
} else {
        sts = advfs_getacl( vp, ADVFS_MAX_ACL_ENTRIES, acl, ADVFS_ACL_TYPE );
        if( sts != EOK )
                return -1;
        tuple_cnt = u.u_r.r_val1;
}

/*
 * Check the (u,g) tuples first, if there is no ACL entry for this caller, then
 * We'll use the base tuples to determine accessability.
 */
for( i = 0; ( i < tuple_cnt ) && ( (unsigned)acl[i].uid < ADVFS_ACL_BASE_USER )
                              && ( (unsigned)acl[i].gid < ADVFS_ACL_BASE_GROUP ); i++ ) {

        if( acl[i].uid == uid ) {

                /* Found a (u,g) tuple */
                if( acl[i].gid == gid ) {
                        match = TRUE;
                        mode |= acl[i].mode;
                        continue;
                }

                /* make sure this group id is valid */
                if( groupmember( acl[i].gid ) ) {
                        match = TRUE;
                        mode |= acl[i].mode;
                }

        }
}

if( match )
        return mode;

/* Check all of the (u,*) tuples, start with the base tuple
 *
 * Since the ACL is presorted with the (u,g) tuples first, followed by
 * the (u,*) tuples then the (*,g) tuples, we do not need to reset i
 */
if( fuid ==uid )
        return advfs_getbasetuple( fmode, ADVFS_ACL_USER );
for( i = 0;    ( ( i < tuple_cnt ) && ( acl[i].gid == ADVFS_ACL_BASE_GROUP ) ); i++ ) {
        if( acl[i].uid == uid )
                return acl[i].mode;
}

/* Check all of the (*,g) tuples, start with the base tuple
 *
 * No need to reset i since the ACL is presorted.
 */
if( ( fgid == gid ) || ( groupmember( fgid ) ){
        match = TRUE;
        mode = getbasetuple( fmode, ADVFS_ACL_GROUP );
}
for( i = 0;    ( ( i < tuple_cnt ) && ( acl[i].uid == ADVFS_ACL_BASE_USER ) ); i++ ) {
        if( ( acl[i].gid == gid ) || ( groupmember( acl[i].gid ) ) {
                match = TRUE;
                mode |= acl[i].mode;
        }
}

if( match )
        return mode;

/* Return the file's "other" mode, i.e. the (*,*) base tuple mode */
return advfs_getbasemode( fmode, ADVFS_ACL_OTHER );
```

Exceptions

None.

4 Dependencies

4.1 Behavior in a cluster

PL files and ACLs should behave well in a cluster.   CFS will release vnodes on last close, so PL files open through .tags should behave consistently in a cluster and in a standalone system.

4.2 Kernel Instrumentation/Measurement Systems

o

4.3 Standards

This design should not impact standards.

4.4 Learning Products (Documentation)

Documentation will need to be updated to reflect the new interface through .tags and the SYSV_ACL support for AdvFS. The PL documentation is dependant on the export UI.

5 Issues (Optional).

High Priority

- Small PL files require 8k storage. Could be reduced to 4k if it is deemed worthwhile.
    - Owner: Contact:
    - Status: Open.
- Currently, wasted space is never reclaimed. Can this wait for future releases or does it need to go into first release?
    - Contact: Status: Open.

Medium Priority

- Issue...
    - Owner:
    - Contact:
    - Status: Closed/Open. If closed, resolution:

Low Priority

- Issue...
    - Owner:
    - Contact:

Status: Closed/Open. If closed, resolution: