AdvFS Snapshots (Kernel)

Design Specification

Version 2.0

DB

CASL

Building ZK3
110 Spit Brook Road
Nashua, NH 03062

Design Specification Revision History		
Version	**Date**	**Changes**
1.0	11/20/03	First draft for internal review

Table of Contents

Preface

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

Author Name	Mailstop	Email Address

Sign-off review

Approver Name	Approver Signature	Date

1 Introduction

1.1 Abstract

This design describes a kernel implementation for multiple-writeable snapshots for AdvFS. The design provides the basis for the implementation of AdvFS Snapshots on HPUX; however, in the first release AdvFS will only expose an interface for a single, read-only snapshot. The multiple-writable features described in this design will only be provided as infrastructure for future releases and will not be fully qualified. This design will be fully tested with respect to the single, read-only features.

1.2 Product Identification

Project Name	Project Mnemonic	Target Release Date
AdvFS Read-Only Snapshots	AdvFS RO Snaps	

1.3 Intended Audience

This design assumes a good deal of familiarity with AdvFS kernel internals and with the mechanisms that AdvFS uses to interface with the UFC. As a result, the design is intended to be read and reviewed by AdvFS kernel engineers and those interested in the internals of AdvFS Snapshots on 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	AdvFS Integration with UFC Design Specification	
2	AdvFS Snapshots User Design	
3		

1.5 Purpose of Document

This design presents a description of multiple-writeable snapshots to be implemented. While the entire design is complete, the implementation of the design will be phased and will focus on achieving fully tested, single, read-only snapshot functionality for the first release of AdvFS in HPUX. This design focuses on the main functional paths of snapshots along with the locking mechanism used to prevent race

conditions. Where applicable, the design compares the design of AdvFS Snapshots on HPUX to the implementation of AdvFS Clones on Tru64.

1.6 Acknowledgments & Contacts

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

DA, TM, BT, and DL

1.7 Terms and Definitions

Term	Definition
COW	Copy-on-Write. This term refers to delaying copy until the source data is about to be modified.
Snapshot	A logical copy of a file at a moment in time. The snapshot file stores original data for each change in the parent file.
Snapset	A copy of a fileset at a moment in time. Used to refer to the entire set of snapshot files as opposed to a single snapshot[1].
RO snapshots	Read only snapshot. This term refers to the Tru64 model of clones where the snapset is read only and cannot be modified.
MW snapshots	Multiple-writable snapshots. In the first release, the infrastructure will be in place for MW snapshots, but they will not be enabled or fully tested.
Snapshot Tree	In the case of multiple snapshots, the snapshot tree refers to the hierarchical structure of the snapshots. Each file is either the root (and in the first fileset to be snapped) or is a child of some file. Any file may have any number of snapshot children.
Parent	The parent snapshot is the file or fileset which contains the “original” data or the data to be COWed.
Child	The child snapshot is the file or fileset which was created as a snapshot and will receive COWed data. In a MW snapshot context, a file or fileset can be both a child and a parent.
Snap maps	The term snap maps is used to describe a two-dimensional list of extent maps for children snapshots of

[1] The term snapset is used in this design to differentiate between a fileset of snapshot files and a single snapshot file. The there is no plan on exporting the term snapset to documentation or user commands.

	a specific file. The list is organized by file then by extent offset. The goal of the snap maps is to concisely represent all storage that must participate in a COW operation.
Unmapped extents	In a snapshot child, an extent can have one of three states, mapped as storage, mapped as a hole, or unmapped. An unmapped extent has not yet been COWed and requires the parent snapshot to determine what that extent represents.
Sympathetic reference count	A reference placed on an object (bfSet or a bfAccess structure) as a result of a reference put on another object.
Out of Sync Snapshot	An out of sync snapshot is a file that does not correctly represent its parent at the moment in time that the snapshot was created. Once a file becomes out of sync, it is forever out of sync and no further IO to the file will succeed.

2 Design Overview

2.1 Design Approach

In developing this design, the author attempted to consider both the current requirement to provide snapshot capability that is functionally comparable to Tru64 and the future goals of AdvFS on HPUX to support multiple, writable (MW) snapshots. This document describes a complete design for multiple, writeable snapshots. Only RO snapshots will be tested and qualified for the first release.

In some cases, optimizations that existed in Tru64 were eliminated because of the complexity on Tru64. The ability of AdvFS snapshots to support the transfer of extents from parents to children snapshots on deletes and truncates is not supported by this design. Additionally, the ability to delay the deletion of a parent snapshot until the deletion of the child snapshot has also been deferred in favor of a simpler model of forcing a COW on delete. The optimizations will be added back in subsequent releases.

2.2 Overview of Operation

On Tru64, as a page was pinned, it was COWed before it was modified. Since all pages were pinned prior to being modified, the COW processing could be abstracted outside of the normal write paths and put into COW routines.

Because of the design of AdvFS when using the UFC, only advfs_getpage is able to have access to all the data required to perform COWing efficiently. As a result, the COWing operations on HPUX will conditionally happen in the main line code paths. There are two basic types of COW that are necessary for snapshots. The first type, done once the first time a snapshot's parent file is touched, involves the copying of metadata for the snapshot. When a snapset is first created, only the tag directory for the filesystem is copied to the new snap set. Each snapshot file shares its metadata with its parent. The first time a parent file is touched after making a snap of the filesystem, the metadata of the parent is COWed over to the snapshot file. The metadata COWing makes a copy of all non-extent mcells and links the bfap of the snapshot to that new mcell chain.

In addition to the COWing of metadata, extent data also must be COWed. Extent data will be COWed in advfs_getpage whenever a fault request for write permission comes into advfs_getpage and the range being requested has not already been COWed. Because metadata COWing must occur before extent data COWing can occur, advfs_getpage will check to see if the snapshot already has its own set of metadata before doing any extent data COWing.

Synchronization between creating new snap sets and operations on the parent set will be handled by a combination of a new flag in the bfSet and the file lock of the files in the parent filesystem. When a filesystem starts the snapping processing, it will set a flag indicating that all advfs_getpage callers should block and wait for the snap to complete if they are trying to do a write. In advfs_getpage, writers will synchronize with the new flag in the file set. The synchronization may, however, allow the advfs_getpage caller to continue with the write request if the file lock is already held for write.

Removing a filesystem will not be allowed if a snapshot exists of the filesystem. It will be required that all child snapshots of a filesystem are removed before the filesystem is removed. This notion will carry over for MW snapshots as well, were the concept is more important when one considers removing a snap set that itself has a snap set.

2.3 Scalability

This design describes an implementation of snapshots based on filesets in a single domain. All filesets in a single domain share a common log and must be mounted and served from the same node in a cluster. As a result, having a large number of related, writeable snapshots concurrently mounted may affect the scalability of a cluster since all of the snapsets will be served from the same node.

Additionally, as the number of snapshots on the same level increases (the number of child snapshots), the number of IOs required to perform a single COW will increase thereby causing a linear decrease in performance. This solution is likely to have scalability consequences with respect to number of children. Section 3.2.16 discusses future enhancements to deal with scalability issues.

2.4 Performance

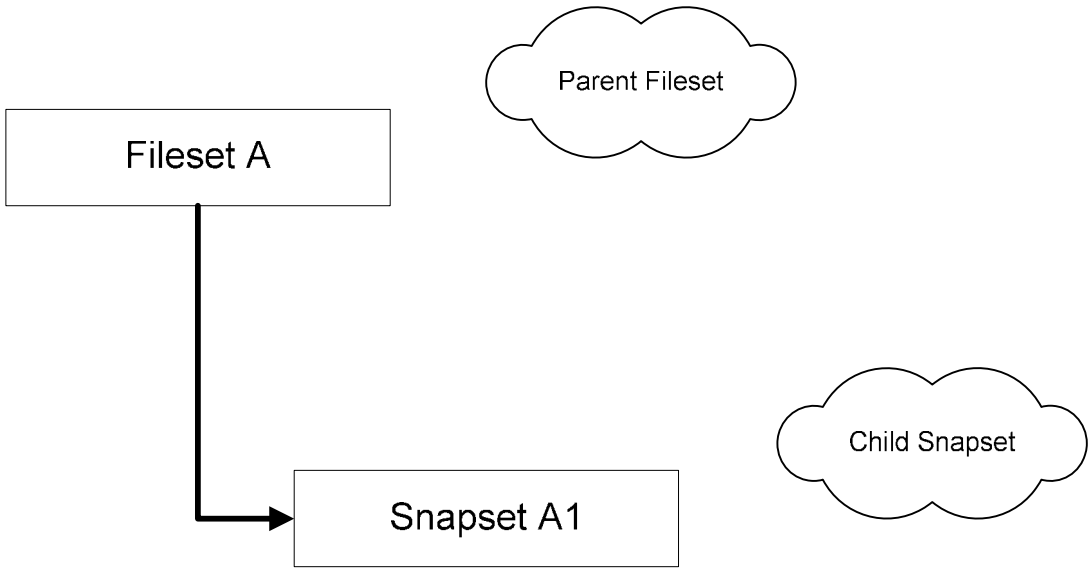
The time required to create a snapshot will be proportional to the amount of dirty data and the number of files in the filesystem to be snapped. The process of creating a snapshot requires a domain flush and copy of a single file. Therefore, the largest contributor to time to flush will be the time required to flush the domain. Section 3.2.16 discussed future enhancements to address the time-to-snap issue. The time to snap should be no worse than Tru64 and will allow all reads to proceed unhindered during the snapping process.

Real time performance will be impacted by creating a snapshot. Snapshots will force synchronous IOs on all writes that require any copy-on-write. As a result, real time performance that relies on asynchronous IO will be impacted.

2.5 The Snapshot Model

Before describing the design for AdvFS Snapshots on HPUX, it is useful to understand the model that will be used. The following pictures and description illustrate various concepts and build up to the model described by this document which is a multiple-writeable snapshot model.

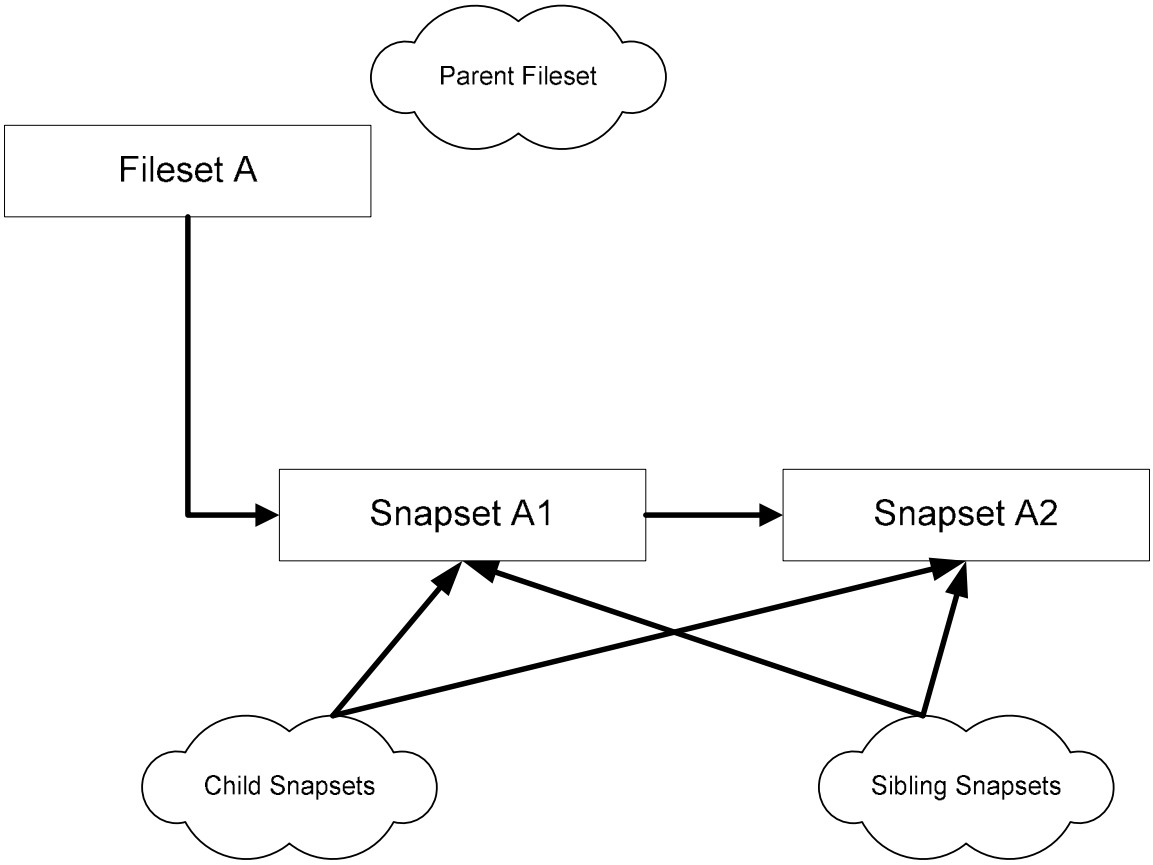
2.5.1 Single, Read-Only Snapshots



This single, read-only model of snapshots is nearly equivalent to clones on Tru64. The model allows for at most one child snapshot which is directly related to one parent. More than one snapshot child can never exist. In order to create a second snapshot, the first snapshot child must first be removed.

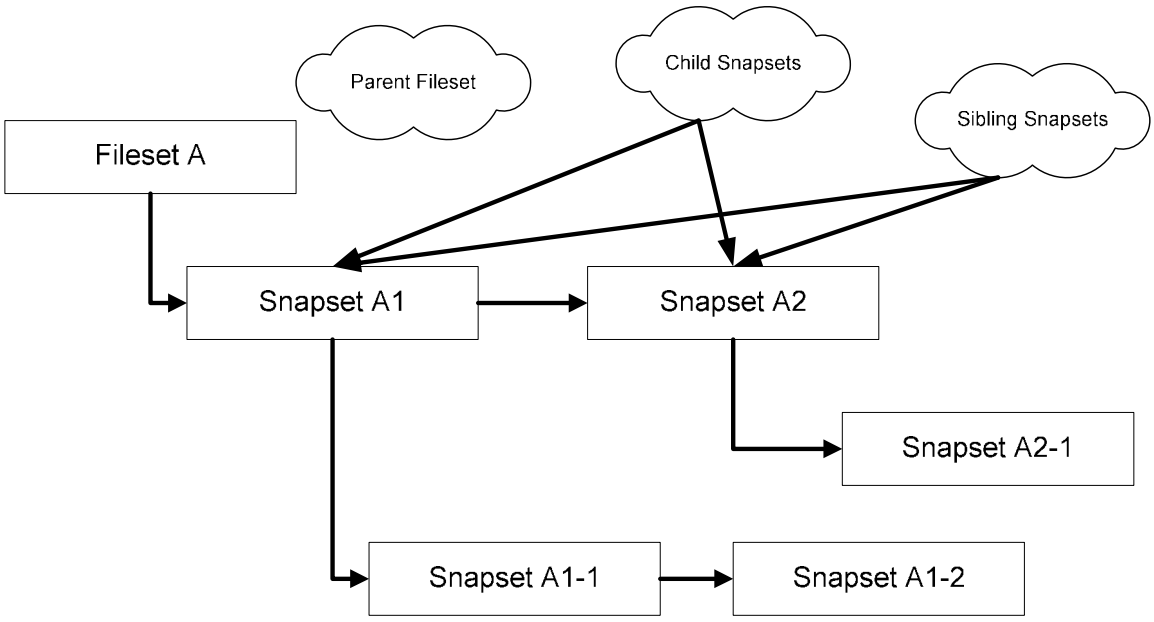
In addition to the limitation of having only a single snapshot child, the snapshot child cannot be modified. While reads are allowed in snapset A1 (the child) any mounts are done read-only and all writes will fail.

2.5.2 Multiple, Read-Only Snapshots



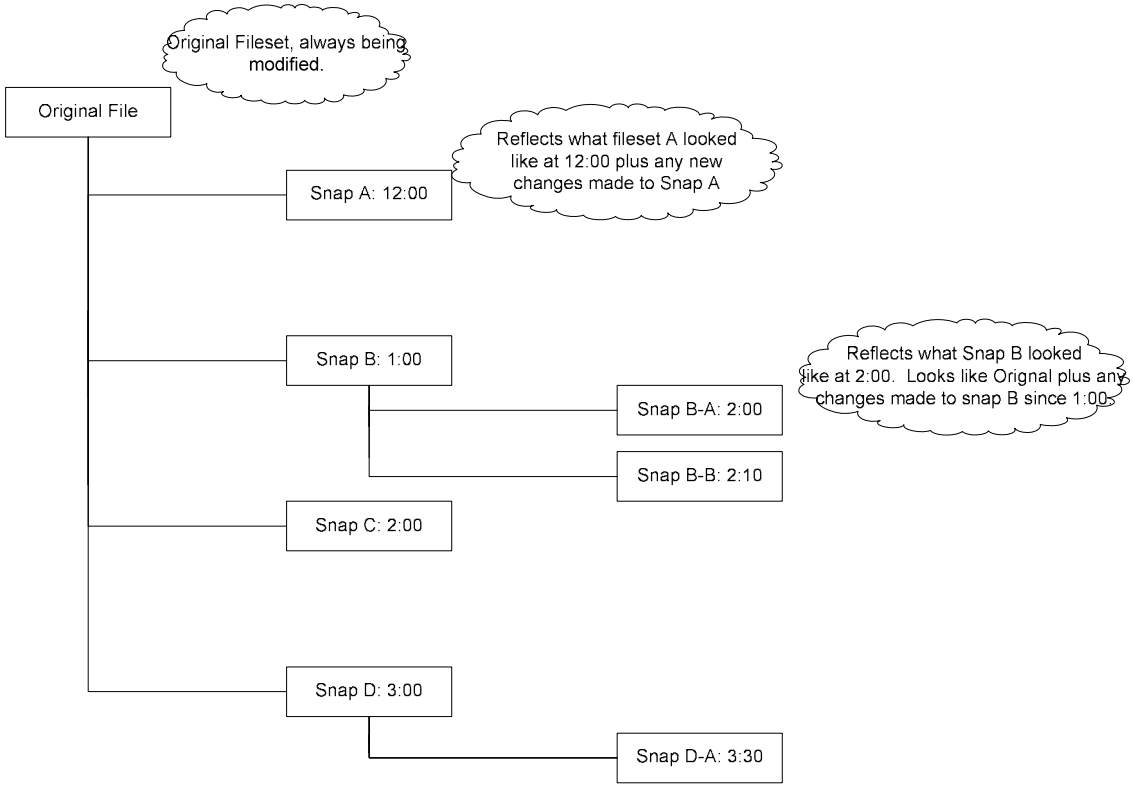
In the multiple, read-only snapshot model, a fileset can have multiple snapshot children at the same time. Each snapshot child is a sibling to each other snapshot child. Only one level of snapshot children can exist, and all snapshot children are mounted read only. All writes will fail on any snapshot child.

2.5.3 Multiple, Writeable Snapshots



In the multiple, writeable model, a fileset can have any number of concurrent snapshot children and each of those children can have additional children snapshots. In this model, any snapset can be mounted writeable and snapshot children can be modified independently of their parents. A sibling snapshot in this model is defined as a snapshot which shares a common immediate parent snapshot.

2.5.4 The Multiple, Writeable Snapshots Model through Time



The diagram above provides an example of how a multiple-writeable model of snapshots might evolve over time. The example is purly theoretical and is provided only as a means to help understand the potential uses for multiple, writeable snapshots.

In the above example, a snapshot has been taken each hour of the original file. Additionally, Snap B and Snap D have been snapped themselves and have versions that reflec them at a certain time. There is no indication in this diagram of why or when files changed and what was COWed as a result.

2.6 Major Modules

This design considered a few key areas. The first, and the one with the most impact on AdvFS overall, is advfs_getpage, the heart of the snapshot system. advfs_getpage will be responsible for making sure that extents are correctly COWed as data is modified and for making sure that a snapshot has metadata in which to put COWed extent data.

The next major area is in fileset creation. Fileset creation requires careful locking to ensure that once a filesystem is snapped, all data on that filesystem is correctly COWed when it is modified.

Finally, the access structure and fileset management code is impacted by snapshots. When opening or closing a snapshot or a snapset, care must be taken to correctly open the parent or child snapshot.

The impact of CFS on snapshots is discussed explicitly in a subsection of this document; however, where CFS impacts AdvFS code, it is discussed in line.

2.7 Major Data Structures

The bfAccess structure and the bfSet structures bear the majority of the burden for managing snapshots. Both of these structures are being modified to remove old Tru64 clone fields that are not longer relevant and are being modified to remove superfluous locks. The bsBfSetAttrT structure is also changing to support MW snapshots.

On Tru64, out-of-space errors were handled somewhat clumsily by clones. To improve the granularity of out-of-sync snapshots, the information about when a file is not in agreement with the original file (at the time of the snapshot) will be moved into the tag directory of the snap set. At the time of creating the snap set, the entire tag directory is copied, so the tag directory provides a persistent and reliable mechanism for tracking snapshot state information.

On HPUX, when a file is marked as out-of-sync, all further attempts to read or write from the file will fail with EACCESS.

2.8 Design Considerations

This design considers the impact of AdvFS snapshots on CFS.

See Section 3.2.14 for a description of the impact of snapshots on recoverability and the policies that will govern how and when snapshots are recovered.

3 Detailed Design

3.1 Data Structure Design

3.1.1 struct bfAccess

```
struct bfAccess {
    dyn_hashlinks_w_keyT hashlinks; /* dynamic hashtable links */
    struct bfAccess *freeFwd;
    struct bfAccess *freeBwd;
    advfs_list_state_t freeListState;   /* Determines if the access structure
                                         * is on the closed list, free list,
                                         * or no list. */
    uint32_t accMagic;                /* magic number: structure validation */
                                      /* guard next two with bfSetT.accessChainLock */
    struct bfAccess *setFwd;          /* fileset chaining */
    struct bfAccess *setBwd;
    mutexT bfaLock;                   /* lock for many of fields in this struct */
                                      /* next 3 guarded by bfaLock */
    int32_t    refCnt;                /* number of access structure references */
    int32_t    dioCnt;                /* threads having file open for direct I/O */
    stateLkT   stateLk;               /* state field */

    struct vnode *bfVp;               /* Pointer to the vnode for this file */
    struct vnode bfVnode;             /* The vnode for this file.  */

    struct bfNode    bfBnp;           /* This files bfVnode */
    struct fsContext bfFsContext;     /* This files fsContext area */

    int32_t    bfaWriteCnt;           /* Count of number of writes in progress for
                                       * synchronization with snapset creation.  Protected
                                       * by the atomic increment macros.  Waiters are
                                       * notified using the bfaLock and the bfaSnapCv */
    /* Snapshot related fields.  Protected by the bfaSnapLock. */
    rwlock_t   bfaSnapLock            /* Protects snapshot related fields and provides
                                       * synchronization */
    cv_t       bfaSnapCv              /* Used for synchronization, protected by
                                       * bfaSnapLock */
    bfAccessT* bfaParentSnap          /* Parent bfap of this bfap.  NULL if
                                       * parent not open or doesn't exist */
    bfAccessT* bfaFirstChildSnap      /* First child snapshot of this bfap. NULL
                                       * if no children */
    bfAccessT* bfaNextSiblingSnap     /* Next sibling snapshot on the same level
                                         NULL if this is the last bfap on the level */
    int32_t    bfaRefsFromChildSnaps  /* Indicates the number of refCnts caused by
                                       * child snaps accessing the parent */
    size_t     bfa_orig_file_size     /* Highest byte offset to be COWed in a snapshot
                                       * This is always zero if no parent exists.
                                       * If a parent exists, this is the file_size of
                                       * the parent at the time of the snapshot. When a
                                       * snapshot has its own metadata, this is stored in
                                       * the bsBfAttr bfat_orig_file_size field. */



    rwlock_t trunc_xfer_lk;      /* prevent clone reads during orig truncation */
    rwlock_t cow_lk;
    rwlock_t clu_clonextnt_lk;  /* clusters only lock protecting clone xtnts */

    uint32_t cloneXtntsRetrieved; /* Indicates cluster client obtained xtnts */
                                  /* Protected by either clonextnt_lk or */
                                  /* the cow_lk */

    off_t migrate_starting_offset; /* range being migrated */
    off_t migrate_ending_offset;
    struct bfAccess *nextCloneAccp; /* link to next clone's access struct */
    struct bfAccess *origAccp;  /* link to orig bitfile's access struct */
```

```
    uint64_t cowPgCount;         /* Protected by the migStgLk; number of */
                                 /* times bs_cow_pg() has added storage */

    /* the following are valid only if mapped == 1 */
    uint32_t cloneId;            /* 0 ==> orig; "> 0" ==> clone */
    uint32_t cloneCnt;           /* set's clone cnt last time bf changed */
    uint32_t maxClonePgs;        /* max pages in clone */
    bfDataSafetyT dataSafety;    /* bitfile's data safety attribute */

    /* flags */
    uint32_t noClone;            /* flag true if bitfile has no clone */
    uint32_t deleteWithClone;    /* true if bf should be deleted with clone */
    uint32_t outOfSyncClone;     /* clone may not be accessed */
    uint32_t trunc;              /* truncate bitfile on last bitfile close */
    bf_fob_t bfaNextFob;         /* 1 past the highest allocated */
                                 /* file offset block */
    bf_fob_t bfaLastWrittenFob; /* Last FOB written by advfs_fs_write(). */

    bfMCIdT primMCId;            /* primary metadata cell id */
    /* Analysis of the code reveals that the following locks
     * seem to be protecting the xtntmap in the following way:
     *
     * trunc_xfer_lk - used for clone access
     * migStgLk - Held across adding stg, removing stg, migrating stg.
     * mcellList_lk - Protects the on-disk Mcells exclusively but also
     *                allows read access to the xtntMap.
     *
     * The following lock gives exclusive access to the xtntMap
     *
     * xtntMap_lk - Must be held when merging xtntmaps and across any call
     *              to imm_extent_xtnt_map.
     *
     * NOTE: The use of these appears to be fairly inconsistent and
     *       needs to be investigated.
     *
     */

    bsInMemXtntT xtnts;          /* extent descriptors */
    void *dirTruncp;             /* possible ptr to dtinfoT struct */
…

}
```

3.1.2 struct bsBfAttr

```
typedef struct bsBfAttr {
    bfStatesT state;          /* bitfile state of existence */
    bf_fob_t bfPgSz;          /* Bitfile area page size */
    ftxIdT transitionId;      /* ftxId when ds state is ambiguous */
    uint64_t   bfat_orig_file_size;          /* filesize at time of snapshot creation */
    int32_t    bfat_del_child_cnt;           /* Number of children to wait for before
                                              * deleting the file.  Used to defer delete
                                              * of parent snapshot */
    bf_od_flags_t     bfat_flags             /* On disk flags for a file. */
    bsBfClAttrT cl;           /* client attributes */
} bsBfAttrT;
```

3.1.3 struct bfSet

```
struct bfSet {
    dyn_hashlinks_w_keyT hashlinks;   /* dyn_hashtable links */
    char bfSetName[BS_SET_NAME_SZ];   /* bitfile-set's name */
    bfSetIdT bfSetId;                 /* bitfile-set's ID */
    uint32_t bfSetMagic;              /* magic number: structure validation */
    int32_t fsRefCnt;                 /* number of bfs_access() accessors */
    domainT *dmnP;                    /* pointer to BF-set's domain structure */
    bfsQueueT bfSetList;              /* list of bfSetT's in this domain */
    mutexT accessChainLock;           /* protects the next two fields */
    bfAccessT *accessFwd;             /* list of access structures */
    bfAccessT *accessBwd;
```

```
    dev_t bfs_dev;                  /* set's dev_t; used for statfs() and stat() */

    bfTagT     dirTag;              /* tag of bitfile-set's tag directory */
    bfAccessT *dirBfAp;

    mutex_t    bfsSnapMutex         /* Used to protect snapshot fields */
    cv_t       bfsSnapCv            /* Used to synchronize IO with making snap set */
    bfSetT     *bfsParentSnapSet    /* Parent set. NULL if not open yet */
    bfSetT     *bfsFirstChildSnapSet /* First child snap set in snap tree */
    bfSetT     *bfsNextSiblingSnapSet/* Next snapset on the same level */
    uint16_t   bfsSnapLevel         /* 0 = root, greater 0 indicates number of
                                     * parents back to root */
    uint32_t   bfsSnapRefs          /* Number of refs from other related snapsets. */

    bfSetT *cloneSetp;              /* pointer to clone set */
    bfSetT *origSetp;               /* for clones, this is parent set desc ptr */
    uint32_t cloneId;               /* 0 ==> orig; "> 0" ==> clone */
    uint32_t cloneCnt;              /* times orig has been cloned */
    uint32_t numClones;             /* current number of clones */

    /*
     * The following state lock is used to coordinate the deletion of
     * a clone fileset and the transfer of extents from files in the original
     * fileset to files in the clone fileset.
     */
    mutex_t  cloneDelStateMutex;    /* Protects cloneDelState and xferThreads */
    stateLkT cloneDelState;         /* State of clone fileset deletion        */
    int32_t     xferThreads;        /* Number of threads doing transfer of    */
                                    /* storage from an original file to a     */
                                    /* clone file in this fileset.            */

    uint32_t infoLoaded;      /* true if correct tagdir info has been loaded */
    mutexT setMutex;          /* protects dirLock & fragLock lock header fields */
    ftxLkT dirLock;           /* tag dir lock */
    bfsStateT state;          /* state */

    /* tagdir info - valid iff infoLoaded == TRUE */
    int32_t bfCnt;            /* number of bitfiles in the bitfile set */
    bs_meta_page_t tagFrLst;  /* page no of head of free list + 1 */
    bs_meta_page_t tagUnInPg; /* first uninitialized page in tag dir */
    bs_meta_page_t tagUnMpPg; /* first unmapped page in tag dir */

    fileSetNodeT *fsnp;       /* file set node pointer */

    mutex_t  bfSetMutex;      /* protect bfSetFlags */
    bfs_flags_t       bfsFlags      /* The high-order 16-bits of this field holds */
                              /* in-memory attributes and the low-order */
                              /* 16-bits holds on-disk flags */
    uint32_t bfSetFlags;

};
```

3.1.4 struct bsBfSetAttrT

```
typedef struct {
    bfSetIdT bfSetId;                   /* bitfile-set's ID */
    bfTagT nxtDelPendingBfSet;          /* next delete pending bf set */
    uint16_t state;                     /* state of bitfile set */
    uint16_t flags;
    adv_dev_t fsDev;                    /* Unique ID */
    adv_uid_t uid;                      /* set's owner */
    adv_gid_t gid;                      /* set's group */
    adv_mode_t mode;                    /* set's permissions mode */
    char setName[BS_SET_NAME_SZ];       /* bitfile set's name */
    uint64_t numberFiles;               /* ~ Number of file in fileset */
    uint64_t numberDirs;                /* ~ Number of dirs in fileset */
    uint64_t numberExtents;             /* ~ Number of extents in fileset */

    bfTagT   nextSnapShotTag;
    bfTagT   origSnapShotTag;
```

```
    uint32_t snapShotId;                /* 0 --> orig; "> 0" --> snapshot */
    uint32_t snapShotCnt;               /* number of times snapshots have been created */
    uint32_t numSnapShots;              /* current number of snapshots */

    bfSetIdT bfsaParentSnapSet       /* Parent fileset to this fileset with respect to
                                      * the snapshot tree */
    bfSetIdT bfsaFirstChildSnap;     /* Head of the chain of child snap sets */
    bfSetIdT bfsaNextSiblingSnap     /* List of snaps on this level of the snap tree */
    uint16_t bfsaSnapLevel;          /* 0 = root.  Greater than 0 indicates number of
                                      * parents back to root. */

    uint16_t rsvd1;
    adv_time_t bfsaFilesetCreate     /* Time of fileset creation */
    uint64_t rsvd3;
} bsBfSetAttrT;
```

3.1.5 struct advfs_pvt_param

```
struct advfs_pvt_param {
    struct bsBuf      *app_bp;              /* Associated bsBuf */
    off_t             app_total_bytes;      /* Total read() or write() length */
    off_t             app_starting_offset;  /* Starting Offset of original request */
    pvt_param_flags_t app_flags;            /* Flags */
    uint32_t          app_started_readahead; /* TRUE if read-ahead was started */
    bf_fob_t          app_ra_first_fob;     /* Read-ahead: first FOB to read */
    bf_fob_t          app_ra_num_fobs;      /* Read-ahead: number of FOBs to read */
    ftxHT             app_parent_ftx        /* parent transaction for metadata COWs */
};
```

3.1.6 struct extent_blk_desc

```
struct extent_blk_desc {
    struct extent_blk_desc *ebd_next_desc;   /* Next desc. in a list, null term */
    struct extent_blk_desc *ebd_snap_fwd;    /* Next desc for use by snapshots */
    bfAccessT*        ebd_bfap;              /* Used by snapshots to chain snap_maps,
                                              * extent maps of snapshots requiring
                                              * COWing */
    off_t             ebd_offset;            /* Starting offset in bytes in file*/
    size_t            ebd_byte_cnt;          /* length of the range described */
    bf_vd_blk_t       ebd_vd_blk;            /* Disk block the mapping begins at
                                              * -1 = hole, -2 = start perm hole */
    vdIndexT          ebd_vd_index;          /* volume index this mapping is on */
};
```

3.1.7 struct ioanchor

```
typedef struct  ioanchor  {
    spin_t anchr_lock;          /* Coordinate changes to anchor using lock. */
                                /* advfs_iodone() always gets spin lock. */
    int64_t anchr_iocounter;    /* Single IO request callers set to 1. Else, */
                                /* set to number of IO's in multi-IO set */
    uint64_t anchr_flags;       /* Anchor flags for advfs_iodone to check */
    struct buf *anchr_origbuf;  /* Set to the original UFC IO buf structure*/
                                /* for advfs_iodone to use. */
    cv_t anchr_cvwait;          /* Optionally allows caller to sleep on this */
                                /* condition variable until IO completes. */
                                /* Caller can also use with */
                                /* IOANCHORFLG_WAKEUP_ON_ALL_IO flag */
    struct ioanchor *anchr_listfwd; /* Caller can link multiple anchors to */
    struct ioanchor *anchr_listbwd; /* take responsibility for freeing anchors. */
                                /* Caller must set the */
                                /* IOANCHOR_KEEP_ANCHOR flag to use the link.*/
    actRangeT *anchr_actrangep; /*Active range pointer when using */
                                /* active range locking Otherwise, set to 0.*/
    struct buf *anchr_aio_bp;   /* Asynchronous IO buffer for directIO only.*/
    uint32_t anchr_magicid;     /* Unique structure validation identifier */
    struct buf *anchr_buf_copy; /* A copy of the original buf struct.  Used by
                                 * snapshots.  */
    struct adviodesc_t *anchr_error_ios      /* A chain of adviodesc structs that had
```

```
                                        * errors occur during IO.  The chain is maintained
                                        * in the advio_fwd pointer in the adviodesc_t on if
                                        * the IOANCHORFLG_CHAIN_ERRORS flag is set. */
} ioanchor_t;
```

3.1.8 struct adviodesc_t

```
typedef struct adviodesc {
    blkDescT advio_blkdesc;     /* Virtual disk location and index */
    bfAccessT *advio_bfaccess;  /* File access structure */
    ioanchor_t *advio_ioanchor; /* All callers initiating IO supply an anchor*/
    struct bsBuf *advio_bsbuf;  /* Only metadata/log data write IO set ptr */
    int(*advio_save_iodone)__((struct buf *));
                                /* Save IO caller's buf iodone() */
    uint64_t advio_flags;       /* Flags set by advfs_bs_startio() caller */
                                /* for advfs_iodone() processing. */
    /* Following fields are mainly for AdvFS IO retry. */
    off_t advio_foffset;        /* 1KB aligned file byte block offset. */
    caddr_t advio_targetaddr;   /*starting data virtual address for I/O */
    int32_t advio_ioretrycount; /* Count of AdvFS initiated I/O retries */
    uint32_t advio_magicid;     /* Unique structure validation identifier */
    struct adviodesc *advio_fwd /* Used to link io descriptors */
} adviodesc_t;
```

3.1.9 struct bfTag

```
typedef struct {
    tagNumT tag_num;          /* tag number, 1 based */
    tagSeqT tag_seq;          /* sequence number */
    uint32_t padding;         /* structure alignment by compiler */
    uint32_t bft_tag_flags;   /* Flags for the tag.  Used to return flags from
                               * tagdir_lookup_next. Not always valid. */
} bfTagT
```

3.1.10 Enumerations

3.1.10.1 enum bfs_flags_t

Flags for bitfile sets are currently divided into two groups, in memory flags and on disk flags. The flags are defined in separate places as #defines. This enumeration will merge the set of flags into one enumeration but maintain the quality of having the low order 16 bits represent on disk flags and the high order 16 bits represent in memory flags.

```
typedef enum {
       BFS_OD_OUT_OF_SYNC     0x0001  /* Snapshot could not allocate storage
                                       * for a copy-on-write, so it is now
                                       * out-of-sync with the original
                                       * fileset.
                                       */
       BFS_OD_HSM_MANAGED     0x0004  /*
                                       * The fileset is set to be managed
                                       * by an HSM.
                                       */
       BFS_OD_HSM_MANAGED_REGIONS 0x0008
                                        /*
                                         * An HSM-managed fileset has had
                                         * managed regions set, and so
                                         * the BFS_OD_HSM_MANAGED flag must not
                                         * be unset.
                                         */

       BFS_OD_OBJ_SAFETY      0x0010  /* enables/disables forcing zeroed
                                       * newly allocated storage in a file
                                       * to disk before allowing the file to
                                       * have the storage.
```

```
                                        */
        BFS_OD_ROOT_SNAPSHOT   0x0020  /* This flag indicates that this fileset has no
                                        * logical parent snapset.  Either it is the root of
                                        * a snapset tree or has been cleved from its
                                        * parent */
        BFS_IM_ON_DISK_MASK    0x0000FFFF     /* Used to select the on-disk flags */
        BFS_IM_DIRECTIO        0x00010000     /* Default direct I/O */
        BFS_IM_SNAP_IN_PROGRESS   0x00020000  /* The fileset is currently being snapped,
                                               * all new getpage write requests must
                                               * synchronize */
        BFS_IM_NOATIMES        0x000400000    /* Previously a mount flag, not a fileset
                                               * flag */
} bfs_flags_t
```

3.1.10.2 enum bfa_flags_t

This enumeration provides a set of flags for bfAccess structures. The flags are stored in the bfaFlags field of the bfAccess structure and are protected by the bfaLock.

```
typedef enum {
        BFA_NO_FLAGS        = 0x0,    /* Not open, not mapped */
        BFA_EXT_OPEN        = 0x1,    /* 1 or more external opens */
        BFA_INT_OPEN        = 0x2,    /* 2 or more internal opens */
        BFA_MAPPED          = 0x4     /* The bfap is inited from disk */
                                      /* same as BSRA_VALID essentially */
        BFA_VIRGIN_SNAP     = 0x8     /* This snapshot has not yet been given its
                                       * own metadata.  */
        BFA_OUT_OF_SYNC     = 0x10    /* The bfap is a snapshot and is out of sync
                                       * with the parent */
        BFA_CFS_HAS_XTNTS   = 0x80    /* This field is set whenever a CFS client has
                                       * successfully acquired a copy of the extent maps
                                       * for this bfap.  */
        BFA_XTNTS_IN_USE    = 0x100   /* This flag indicates that the extent maps are
                                       * being manipulated and CFS clients
                                       * cannot get a copy of them */
        BFA_SNAP_IN_COW_MODE= 0x200   /* Indicates that CFS_COW_MODE_ENTER has been
                                       * called on the bfap */
        BFA_SNAP_CHANGE     = 0x400   /* This flag is used to get around a deadlock
                                       * between advfs_getpage and starting an exclusive
                                       * transaction in advfs_create_snapset.  The
                                       * field lets getpage detect that a snapshot may
                                       * have been created and attempt to do late
                                       * hole COWing and page protection. The flag
                                       * is set every time a file has
                                       * its pages protected during the
                                       * snapping process.  The flag is set whenever a
                                       * file is open when a snapshot child is created and
                                       * is cleared any time all children snapshots are
                                       * opened. */
        BFA_QUICK_CACHE     = 0x800   /* This flag is set on snapshots whose access
                                       * patterns tend to be mostly one time reads.  If
                                       * the flag is set, the access structure will be
                                       * aged in half the time of normal access
                                       * structures. */
        BFA_PARENT_SNAP_OPEN   =0x1000 /* The parent snapshot is opened by the child */
        BFA_OPENED_BY_PARENT   =0x2000 /* The parent snapshot has opened this child */
        BFA_ROOT_SNAPSHOT      =0x4000 /* This file is a root of a snapshot tree */
} bfa_flags_t;
```

3.1.10.3 enum bf_od_flags_t

```
typedef enum {
        BOF_ROOT_SNAPSHOT      = 0x1,  /* No dependency on parent snapshots */
        BOF_DEL_WITH_CHILDREN  = 0x2,  /* The deletion of the last child will cause this
                                        * file to be deleted */
} bf_od_flags_t
```

3.1.10.4 enum acc_open_flags

```
enum acc_open_flags {
    BF_OP_NO_FLAGS      = 0x0,    /* No Flags */
    BF_OP_IGNORE_DEL    = 0x1,
    BF_OP_OVERRIDE_SMAX = 0x2,    /* override acc_ctrl_soft_max to ref bfap */
    BF_OP_BFA_LOCK_HELD = 0x4,    /* bfaLock held on entry to advfs_ref_bfap */
    BF_OP_INMEM_ONLY    = 0x8,    /* Don't init a new bfap, doesn't bump v_count */
    BF_OP_FIND_ON_DDL   = 0x10,   /* Find it on DDL */
    BF_OP_INTERNAL      = 0x20,   /* Doesn't set v_count */
    BF_OP_IGNORE_CLOSED_LIST = 0x40     /* If the bfap is on the free or closed
                                           list, do not remove it.  This is required
                                           to prevent sync code from interfering with
                                           cache aging */
    BF_OP_IGNORE_BFS_DELETING = 0x80    /* If this flag is set, then bs_access_one will
                                         * allow a file to be opened on a fileset that is
                                         * actively being deleted */
    BF_OP_SNAP_REF            = 0x100   /* If set, whenever refCnt is bumped,
                                         * will also be bumped bfaRefsFromChildSnaps */
};
```

3.1.10.5 enum acc_close_flags

```
enum acc_close_flags {
    MSFS_CLOSE_NONE           = 0x0,  /* No Flags */
    MSFS_INACTIVE_CALL        = 0x1,  /* Called from msfs_inactive */
    MSFS_BFSET_DEL            = 0x2,  /* */
    MSFS_DO_VRELE             = 0x4,  /* Call to VN_RELE required*/
    MSFS_SS_NOCALL            = 0x8,  /* */
    MSFS_CLOSE_DEALLOC        = 0x10, /* Dealloc bfap on DEC_REFCNT */
    MSFS_CLOSE_SYNCING        = 0x20, /* The close was issued from a flush/sync */
    MSFS_SNAP_DEREF           = 0x40  /* While holding the bfaLock, and before calling
                                       * DEC_REFCNT, the bfaRefsFromChildSnaps must be
                                       * decremented. */
    MSFS_SNAP_PARENT_CLOSE    = 0x80  /* While holding the bfaLock, and before calling
                                       * DEC_REFCNT, the bfaFlag BFA_OPENED_BY_PARENT
                                       * flag must be cleared */
};
```

3.1.10.6 enum round_type_t

```
typedef enum {
   RND_ALLOC_UNIT      =0x1,
                        /* Round to complete allocation units if the offset is
                         * in the middle of a hole.  This is primarily
                         * intended for writes that start in a hole.  If the
                         * end of a range falls in the middle of a 4k
                         * boundary, the end will be rounded up to 4k. (not
                         * the allocation unit). */
   RND_VM_PAGE         =0x2,
                        /* Round to a vm page boundary (4k).  This is intended
                         * for use when a read is occuring.   This round
                         * type prevents processing of entire allocation
                         * units when only part of the unit is required.
                         * The start of the range will be rounded down to a
                         * 4k boundary and the end will be rounded up to a
                         * 4k boundary */
   RND_MIGRATE         =0x4,
                        /* Round to adjacent leading (left) holes and truncate
                         * trailing (right) holes in order to guarantee
                         * that a hole accompanies its trailing storage.
                         * If the range is completely contained within
                         * a hole, return the entire hole.  Logical adjacent
                         * blocks are coalesced into one extent block descriptor.
                         */
   RND_NONE            =0x8,
                        /* No rounding.  The passed in offset and length must be
```

```
                        * a multiple of DEV_BSIZE. */
   RND_ENTIRE_HOLE     =0x10
                        /* This rounding type will be used to indicate that rounding
                           * should encompass entire holes.  If the range is 1 byte in
                           * in the middle of a 1 GB hole, the entire 1 GB hole will be
                           * returned.  This is for COW operations to COW entire holes
                           * at once. */
} round_type_t;
```

3.1.10.7 enum extent_blk_map_type_t

```
typedef enum {
    EXB_COMPLETE       =0x1,   /* Return a map of holes and storage. */
    EXB_ONLY_HOLES     =0x2,   /* Return only a map of holes.  Do not include storage */
    EXB_ONLY_STG       =0x4,   /* Return only a map of storage. Do not include holes */
    EXB_DO_NOT_INHERIT =0x8    /* Return only local xtnt maps, none from parent snaps */
} extent_blk_map_type_t;
```

3.1.10.8 enum snap_flags_t

This enumeration is used for passing flags into snapshot related routines.

```
typedef enum {
    SF_SNAP_NOFLAGS    =0x0
    SF_SNAP_READ       =0x1,   /* Indicates a read operation is occurring on a snapshot */
    SF_SNAP_WRITE      =0x2,   /* Indicates a write operation is occurring */
    SF_HAD_PARENT      =0x4    /* Used to indicate to advfs_unlink_snapset that a parent
                                * DID exist, but may have been closed.  */
    SF_FAST_SNAP       =0x8    /* Indicates that only metadata should be flushed during
                                * snapset creation */
    SF_FOUND_SNAPSET   =0x10   /* Indicates that the fileset being opened has been
                                * traversed while accessing other related snapsets
                                * see advfs_access_snapset_recursive */
    SF_OUT_OF_SYNC     =0x20   /* Used to indicate a fileset that is out of sync has
                                * been found and children must be marked out of sync */
    SF_NO_UNLINK       =0x40   /* Used to indicate to advfs_force_cow_and_unlink that the
                                * unlink is not desired */
} snap_flags_t
```

3.1.11 Constants and Macros

3.1.11.1 #define BS_TD_OUT_OF_SYNC_SNAP 0x8

This #define is used to set a flag in the tag directory if and when a snapshot file becomes out of sync with the parent file.

3.1.11.2 #define BS_TD_VIRGIN_SNAP 0x10

This #define is used to indicate that the file described by this tag directory entry does not have it's own metadata. No COW has occurred to this file yet. This is set in each tag entry when a snapset is created (during the copying of the tag directory) and is cleared when advfs_access_snap_children COWs the metadata for a given file.

3.1.11.3 uint64_t advfs_cow_alloc_units 8

This global is used to determine the number of allocation units to be checked for COWing when a write request occurs on a file with a snapshot. COWing will be done on aligned allocation units of advfs_cow_alloc_units, so if a write request comes in on the second allocation unit, the COW would happen over the byte range:

[0..advfs_cow_alloc_units*bfap->bfPageSz*ADVFS_FOB_SZ].

This value is made dynamic to allow for changing on a live kernel if necessary.

3.1.11.4 #define ADVFS_FORCE_COW_MAX_ALLOC_UNITS 64

This #define is used when the COWing of a file is being forced (the entire file will be faulted in for write to force a COW to all children snapshots). This value represents the maximum number of allocation units that will be brought COWed in a single iteration. This value is used to reduce the stress on UFC memory that might result from forced COWs bringing in large files.

3.1.11.5 #define ADVIOFLG_SNAP_READ 0x8

This constant is a flag to advfs_start_blkmap_io to indicate that the IO to be issued is a READ for a READ/WRITE serial operation on a snapshot. The flag indicates that the IO anchor should have an IO Count of 2 rather than 1 and that the IOANCHORFLG_WAKEUP_ON_ALL_IO is set.

3.1.11.6 #define IOANCHORFLG_CHAIN_ERRORS 0x10

This IO Anchor flag is used to indicate to IO completion that any adviodesc_t structures on which an error has occurred should be chained to the IO Anchor via the anchr_error_ios field. Additionally, the multiple adviodesc_t's will be chained via the advio_fwd field of the adviodesc_t. This is used to return IO error information to advfs_getpage when dealing with snapshots and multiple writes.

3.1.11.7 #define ADVFS_MAX_SNAP_DEPTH 5

This constant defines the maximum depth of snapshots. A snapshot can have at most ADVFS_MAX_SNAP_DEPTH parents.

3.1.11.8 #define ADVFS_FILES_BEFORE_PREEMPTION_POINT 128

This constant is the number of files that can be freed before a preemption point will be hit when trying to remove all files in a fileset.

3.1.11.9 #define ADVFS_CFS_COW_IS_COMPLETE 1<<63

This flag is used to indicate to CFS that an extent of a file has already been COWed and that it is not necessary to send direct IO writes through the server node. If this flag is set in the high order bit of the bsExtentDescT bsed_fob_offset field, the extent can be safety directly read and written as long as the direct IO token is held.

3.1.11.10 #define ADVFS_ROOT_SNAPSHOT (-1)

This #define is used to initialize the bfat_orig_file_size in files that do not have parent snapshots.

3.1.11.11 APP_MARK_READ_ONLY

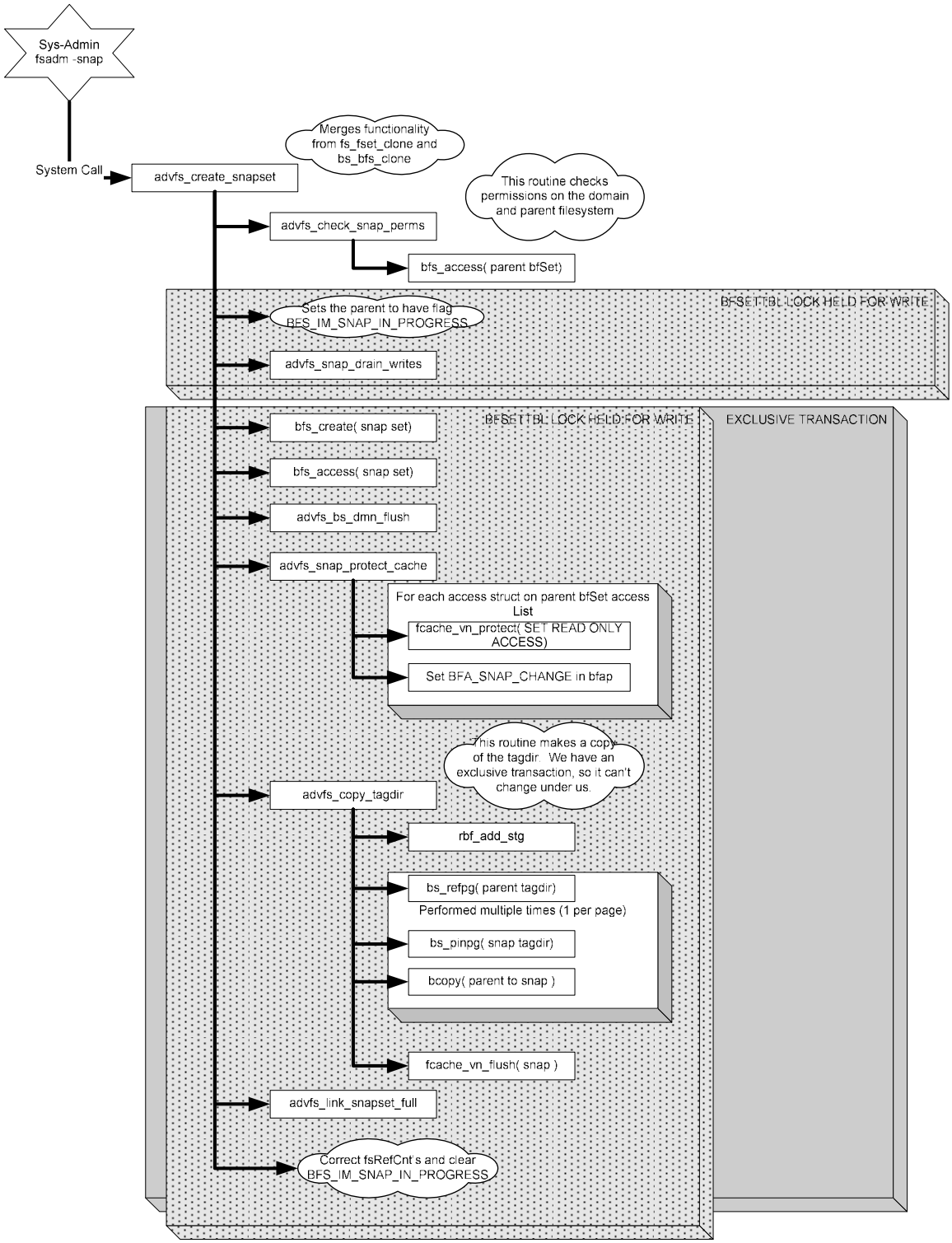
This new flag for advfs private parameters to advfs_getpage and advfs_putpage is used to indicate that all pages should but flushed and marked as read only.

3.2 Module Design

This section is divided into subsections based on high level function. The basic functions include creating a snap set, opening a fileset (or snapset), opening a file (or snapshot), and writing to a file (invoking COW processing). The sections are primarily non-intersecting, but some overlap occurs with respect to locking issues.

3.2.1 Creating a Snap Set

3.2.1.1 Function Call Tree Overview



3.2.1.2 Basic Operations of Creating a Snap Set

When the user issues a command to create a snapshot of a filesystem, the command will resolve to advfs_create_snapset in the kernel. advfs_create_snapset provides the kernel interface for creating a snapset. On entry, the parent set can be either mounted or unmounted, and on successful exit from the routine, the parent will be actively COWed on any writes. If the parent file system is mounted, the snap set will be activated on return and will remain open until the close of the parent filesystem. If the parent file system is not mounted or otherwise open, both the parent and snap set will be closed on exit.

On Tru64, fs_fset_clone provided the highest level kernel interface. fs_fset_clone performed access checks prior to resolving to bs_bfs_clone which did the majority of the work for creating a clone. On HPUX, advfs_create_snapset will make a call to advfs_check_snap_perms to make sure that the fileset can be snapped. Once permissions have been verified, advfs_create_snapset will perform the majority of the work required to create the snapset.

advfs_create_snapset will start an exclusive transaction in which to perform the majority of the snapping work. The transaction synchronizes with rmfset which starts an exclusive transaction and with all modifications to metadata. No transactions can be in progress at the moment when data begins to be COWed since this would provide the potential to COW half of a transaction (half an atomic update). Starting an exclusive transaction has the added benefit of preventing any further modifications to the tag directory of the parent file system. Therefore, no files can be created, deleted, or migrated.

Before the transaction is started, advfs_create_snapset will open the parent fileset and set the BFS_IM_SNAP_IN_PROGRESS flag in the parent fileset. The setting of the BFS_IM_SNAP_IN_PROGRESS flag will block write activity on the parent file system by holding up new writers in advfs_fs_write and advfs_getpage. Once the BFS_IM_SNAP_IN_PROGRESS flag is set, any new mmap writes will be stopped in advfs_getpage while any new syscall writes will be stopped in advfs_fs_write. advfs_drain_snap_writes will be called to wait for all in progress writes to complete.

After the parent is accessed, and the active writes are drained (all while holding the bfSetTbl lock), the exclusive transaction will be started to perform the remainder of the snapset creation. The creation of the snap set is done through a call to bfs_create. After bfs_create succeeds, the new snap set is accessed and a ref count is placed on the snap set. At this point, all outstanding metadata on the domain is flushed. Since the thread is in an exclusive transaction, flushing the entire domain makes sure the log is on disk and that any metadata that is COWed will not be undone if the system panics. The flushing of the domain will ensure that all metadata on disk is consistent; however mmappers may still be able to modify cached pages. Next, a call to advfs_snap_protect_cache will protect each page in cache to be read-only, thereby requiring a fault to allow any writes. If a dirty page is found while protecting the page, it will be flushed again.

advfs_snap_protect_cache will call fcache_vn_flush on every file on the parent filesets access set list and pass in the FVF_PPAGE flag along with a new private parameter flag APP_MARK_READ_ONLY. This will set the pi_pg_ro flag on each pfdat, thereby marking the file write protected (read only). In addition, advfs_snap_protect_cache will acquire the bfaLock and set the BFA_SNAP_CHANGE flag in the bfaFlags of the protected file.

Next, advfs_create_snapset will call advfs_copy_tagdir which will allocate storage for and synchronously copy the tag directory of the parent file system to the new snap set.

On return from advfs_copy_tagdir, the new snapset exists on disk and has a complete copy of the tag directory of the parent file system, also on disk). Once advfs_copy_tagdir has completed, it is safe to link the parent and child fileset together in memory and through the bfsAttr structures on disk.

Before returning to the caller, advfs_create_snapset will clear the BFS_IM_SNAP_IN_PROGRESS flag and broadcast on the bfsSnapCv to wakeup any waiters.

3.2.1.3 Functional Call Detail

3.2.1.3.1 advfs_create_snapset

3.2.1.3.1.1 Interface

```
statusT advfs_create_snapset(
       char *parent_dmn_name,        /* Name of parent fileset's domain */
       char *parent_fset_name,       /* Name of parent fileset */
       char *snap_dmn_name,          /* Name of snap's domain */
       char *snap_fset_name,         /* Name of snap's fileset */
       bfSetIdT* snap_set_id         /* bf set id of the new snap set. */
       snap_flag_t    snap_flags     /* If SF_FAST_SNAP is set, only flush metadata */
       ftxIdT  cfs_xid )             /* CFS transaction ID */
```

3.2.1.3.1.2 Description

This routine is the highest level kernel interface for creating a snapshot of a filesystem. This routine is called with no locks held. On successful return, a fileset named snap_fset_name exists on the domain snap_dmn_name and is a snap shot of the fileset parent_fset_name on parent_dmn_name domain.

At a high level, this routine will check for permissions to create a snapshot fileset, and then start an exclusive transaction under which to create the snapshot. The transaction will open the parent file system and create a new filesystem with the name snap_fset_name. The opened parent fileset will be checked for the BFS_DELETING state. If the state is BFS_DELETING, the creation of a snapshot will fail, the transaction will be failed and the function will return. After opening the parent filesystem, a flag will be set (BFS_IM_SNAP_IN_PROGRESS) to indicate that a snapshot is currently being made of the parent and that all modification operations should block including writes and deletes. The BFS_IM_SNAP_IN_PROGRESS flag will also be set in the new snapset. If the parent filesystem is mounted, the CFS callback CLU_CFS_SNAP_NOTIFY[2] will be sent using the SNAP_CREATE argument. On any error that causes the snapshot creation to fail, the CFS callback CLU_CFS_SNAP_NOTIFY will be called passing the SNAP_DELETE parameter. Once the snapshot filesystem is created, advfs_create_snapset will wait for all outstanding writes (mmap and syscall) to complete before continuing. After all writes have been drained via a call to advfs_drain_snap_writes, the parent domain will be flushed. The flush of the domain will make sure that all pages in cache are clean with the exception of user data pages mmapped for write. advfs_snap_protect_cache will be called to protect every page in cache and to flush any additional dirty pages. This will block out any mmap writers who will need to fault into advfs_getpage in order to get write permission on the page.

A call to advfs_snap_protect_cache will walk through each file on the parent filesystems access set list and call fcache_vn_flush on each of the files with the FVF_PPAGE and APP_MARK_READ_ONLY flags. The FVF_PPAGE flag will insure that advfs_putpage is called and the APP_MARK_READ_ONLY will indicate to advfs_putpage that each page in cache should be marked read only. On successful return from advfs_snap_protect_cache, each file in the parent filesystem will have had all its pages write protected.

Once the flush and protect is complete, any COWed data will be consistent before and after a system failure and it is, therefore, safe to call advfs_copy_tagdir to replicate the tag directory for the snapshot filesystem.

On successful return from advfs_copy_tagdir, the set of files that the snapshot file system will track has been established on disk. Additionally, since the setting of BFS_IM_SNAP_IN_PROGRESS and the start of an exclusive transaction, all new writes to files have blocked in advfs_getpage (mmap) or in rbf_add_stg (metadata) or in advfs_fs_write (user data writes).

Once the tag directory is successfully copied to the snapshot fileset, advfs_link_snapset_full will be called to link the child into the parents list of snapset children and to copy the necessary fields of the parent's bfSetAttr record to the child. advfs_link_snapset_full will pin records under the exclusive root transaction, so after advfs_link_snapset_full, the transaction cannot fail. advfs_link_snapset_full will also set the snapset child's on disk state to BFS_ODS_VALID.

Once advfs_snap_protect_cache is called, all open files in the parent snapset have had the BFA_SNAP_CHANGE flag set. Before returning, the fsRefCnt of the child snapset will be set to match

[2] Previously the *_SNAP_* names were *_CLONE_* for CFS. They will be renamed.

the fsRefCnt of the parent through a series of calls to bfs_access. Finally, the BFS_IM_SNAP_IN_PROGRESS flag will be cleared in the parent and child filesets.

Before returning, a broadcast will occur on the bfsSnapCv to wake up any waiters on the create. Once the parent has broadcast, the child will issue a broadcast in a similar fashion. The parent may have waiters in advfs_getpage, the child could only have waiters in code paths attempting to access the new fileset.

advfs_create_snapset will hold the bfSetTbl lock in write mode across the majority of the routine. This lock will synchronize with callers of bfs_access and bfs_open that are trying to open the parent fileset while a snap is being created. In order to allow advfs_bs_dmn_flush to be called while holding the lock, advfs_bs_dmn_flush will be modified to take a flag indicating that the bfSet table lock is already held and need not be acquired.

In the event of any error, the exclusive transaction will be failed, any resources that were allocated will be freed, and the error status of the subroutine that failed will be returned. The child snapshot will be removed as part of the failing of the transaction.

3.2.1.3.1.3 Execution Flow

- Call advfs_check_snap_perms
- If permissions check fails
    - Return E_ACCESS_DENIED
- If clu_is_ready
    - CLU_CFS_SNAP_NOTIFY[3] of snapset create
- /* If advfs_check_snap_perms succeeded, the bfSet parent is open */
- write lock bfSetTbl lock
- if parent fileset is BFS_DELETING, close parent fileset, fail transaction, unlock bfSetTbl lock, propogate error. If this is the first snapset in the domain, call CLU_CFS_SNAP_NOTIFY to notify CFS of the snapset delete.
- Set BFS_IM_SNAP_IN_PROGRESS flag in parent fileset
- Call advfs_drain_snap_writes
- If advfs_snap_drain_writes fails, clear BFS_IM_SNAP_IN_PROGRESS flag in parent fileset, propogate error. If this is the first snapset in the domain, call CLU_CFS_SNAP_NOTIFY to notify CFS of the snapset delete.
- Start an exclusive transaction (if it fails, return the error)
- write lock bfSetTbl lock
- bfs_create the child fileset
- if bfs_create fails, fail transaction, drop bfSetTbl Lock and propogate error If first_snapset_in_domain CLU_CFS_SNAP_NOTIFY of snapset delete
- bfs_access child fileset
- Set BFS_IM_SNAP_IN_PROGRESS flag in child fileset
- if bfs_access fails, fail transaction, drop bfSetTbl Lock and propogate error, clear BFS_IM_SNAP_IN_PROGRESS flag. If this is the first snapset in the domain, call CLU_CFS_SNAP_NOTIFY to notify CFS of the snapset delete.
- Flush the flush the entire domain (advfs_bs_dmn_flush)
- Call advfs_snap_protect_cache
- If advfs_snap_protect_cache fails, finish transaction, drop bfSetTbl Lock and propogate error, clear BFS_IM_SNAP_IN_PROGRESS flag, and delete the child fileset. If this is the first snapset in the domain, call CLU_CFS_SNAP_NOTIFY to notify CFS of the snapset delete.
- Call advfs_copy_tagdir

---

[3] The CLU_CFS_SNAP_NOTIFY with the SNAP_CREATE flag will be modified so that in addition to draining any direct IO writes on clients, it will also invalidate the extent maps on those clients. Invalidating the clients is necessary to make a new optimization for direct IO cluster writes to filesets with snapshots function correctly.

- If advfs_copy_tagdir fails, fail transaction, drop bfSetTbl Lock and propogate error, clear BFS_IM_SNAP_IN_PROGRESS flag. If this is the first snapset in the domain, call CLU_CFS_SNAP_NOTIFY to notify CFS of the snapset delete.
- Call advfs_link_snapset_full passing snap_flags
- If advfs_link_snapset_full fails, fail transaction, drop bfSetTbl Lock and propogate error, clear BFS_IM_SNAP_IN_PROGRESS flag. If this is the first snapset in the domain, call CLU_CFS_SNAP_NOTIFY to notify CFS of the snapset delete.
- Finish exclusive transaction
- Adjust the fsRefCnt of the child snapset to match the parent fileset by calling bfs_access on child until they match.
- Clear BFS_IM_SNAP_IN_PROGRESS flag in parent and child
- cv_broadcast on parent's bfsSnapCv
- cv_broadcast on child's bfsSnapCv
- bfs_close the parent fileset
- Unlock bfSetTbl lock

Note that any time the BFS_IM_SNAP_IN_PROGRESS is cleared in an error condition, a broadcast will occur to wake all waiters.

3.2.1.3.2 advfs_check_snap_perms

3.2.1.3.2.1 Interface

```
statusT advfs_check_snap_perms (
        char    *parent_dmn_name,      /* Name of parent fileset's domain */
        char    *parent_fset_name )    /* Name of parent fileset */
        bfSetT  *parent_bf_set )       /* Returns bf set struct of parent fileset */
```

3.2.1.3.2.2 Description

advfs_check_snap_perms is intended to be a routine used by advfs_create_snapset to open the parent fileset and validate that the parent fileset can be snapped. Validation includes making sure the parent fileset is not managed by an HSM, verifying that the caller has write access to the domain, and verifying that the caller has read access to the parent filesystem.

To verify permissions, advfs_check_snap_perms will first activate the domain of the parent filesystem. Next, the parent fileset will be accessed. If the parent fileset is HSM managed (BFS_OD_HSM_MANAGED in the bfsFlags field of the bfSet structure), then the file system is closed and the domain deactivated and ENOT_SUPPORTED is returned.

Otherwise, the domain parameters of the parent filesystem domain are read via a call to bs_get_dmn_params and the domain is checked to see if the caller has write permissions. If permission is denied, the fileset is closed, the domain is deactivated, and E_ACCESS_DENIED is returned.

Next, the bfSetParams structure is read from the parent fileset and the fileset is checked for read access. If permission is denied, the fileset is closed, the domain is deactivated, and E_ACCESS_DENIED is returned. If the depth of the fileset to be snapped is equal to ADVFS_MAX_SNAP_DEPTH, then the create will be denied with the ENOT_SUPPORTED flag. This indicates that the maximum depth of snapshots has been exceeded.

If no error is returned, this routine will return with the domain activated, the parent fileset accessed, and will return a status of EOK.

3.2.1.3.2.3 Execution Flow

- Activate the domain
- Read the domain attributes (bs_get_dmn_params)
- Call bs_accessible to check that domain is writeable
- If domain is not writeable (ie user does not have write permission)

        - Close domain
        - Return E_ACCESS_DENIED
    - Activate the parent fileset
    - Bfs_open the parent fileset (and related snapsets)
    - Read the bfSetAttr record of the parent fileset
    - If parent fileset has BFS_OD_HSM_MANAGED set
        - Close fileset and domain
        - Return E_NOT_SUPPORTED
    - Call bs_accessible to check that parent fileset is readable
    - If parent fileset is not readable
        - Close fileset and domain
        - Return E_ACCESS_DENIED
    - If parent fileset snapset depth == ADVFS_MAX_SNAP_DEPTH
        - Return ENOT_SUPPORTED
    - parent_bf_set = accessed fileset
    - return EOK

3.2.1.3.3 advfs_copy_tagdir

3.2.1.3.3.1 Interface

```
statusT advfs_copy_tagdir (
       bfSetT  *parent_bf_set_ptr,    /* bfSetT pointer of parent fileset */
       bfSetT* *snap_bf_set_ptr       /* bfSetT pointer of snap set */
       ftxH     parent_ftx )          /* Transaction of parent_ftx */
```

3.2.1.3.3.2 Description

This routine is called to make a replica of the tag directory of parent_bf_set_ptr in the snap_bf_set_ptr's fileset. On entrance to this routine, it is expected that an exclusive transaction is underway and that no modifications can be made to the parent_bf_set_ptr's tag directory. Additionally, it is expected that all changes made to the source tag directory are on disk and will not be undone during recovery.

On successful return from this function, the snap_bf_set_ptr has a separate, but equivalent tag directory file and all the fields of the snap_bf_set_ptr structure are correctly mapped to the new file.

This routine does a synchronous copy of the tag directory and does not use any transactions to do the copy (however a transaction may be started to add storage to the new tag directory file)[4].

This routine will first check to see if the bfaNextFob value of the dirBfap field of the snap_bf_set_ptr structure is less than the same field of the parent_bf_set_ptr. If it is, then the tag file of the snap is smaller than the tag file of the parent and storage must be added to the snap. If required, rbf_add_stg will be called to allocate the difference between the snapshot tag file and the parent tag file.

Once storage has been successfully added, the routine will loop over each bfPageSz unit of the tag directories and call bs_refpg on the parent's tag file and bs_pinpg on the snapshot tag file. Next, the routine will do a bcopy from the parent to the snapshot. After the bcopy is complete, each tag directory entry in the copied page will have the flag field marked as BS_TD_VIRGIN_SNAP to indicate that it has not yet been given its own copy of metadata. Next, the pages will be derefed and unpinned. The unpin with be called with the BS_DIRTY flag which will cause the page to be cached.

[4] In the event that there were very few files on the file system, the single page of the snapsets tag directory file that was allocated by bfs_create may be sufficient to hold the copy of the tag directory. As a result, a transaction may not be started by rbf_add_stg.

Once all pages have been bcopied, a call to fcache_vn_flush will be made on the tag directory of the snapshot with the FVF_WRITE and FVF_SYNC flags. Because the dirty pages were pinned with bs_pinpg and not rbf_pinpg, they were never given LSNs, but they were given bsBufs by advfs_get_metapage. advfs_putpage will note that the writeRef on the bsBuf is 0 and the bsb_metafwd field is NULL and will therefore consider the pages eligible for IO. Once fcache_vn_flush completes successfully, the tag directory of the snapshot will be on disk and consistent with the parent.

Finally, tagdir_get_info will be called to initialize the tag directory related fields of the snap_bf_set_ptr. On success, the routine will return to the caller with the tag directory fully copied.

3.2.1.3.3.3 Execution Flow

- parent_tagdir = parent_bf_set_ptr->tagBfap
- snap_tagdir = snap_bf_set_ptr->tagBfap
- if parent_tagdir->bfaNextFob > snap_tagdir->bfaNextFob
    - rbf_add_stg (parent_tagdir->bfaNextFob – snap_tagdir->bfaNextFob) to snap_tagdir
    - if rbf_add_stg fails, propogate the error
- /* The parent and child tag directories are now the same size. */
- foreach bfPageSz page in parent_tagdir
    - bs_refpg range in parent_tagdir
    - bs_pinpg range in snap_tagdir
    - bcopy from parent_tagdir to snap_tagdir
    - for each tag entry in range
        - set BS_TD_VIRGIN_SNAP in tag flags
    - bs_derefpg range in parent_tagdir
    - bs_unpinpg range in snap_tagdir (with BS_DIRTY flag)
- fcache_vn_flush snap_tagdir from offset 0 to size 0 with FVF_SYNC (entire file and synchronously)
- if fcache_vn_flush returns an error, propogate the error and return
- call tagdir_get_info to initialize tag fields of snap_bf_set_ptr
- return EOK

3.2.1.3.4 advfs_snap_protect_cache

3.2.1.3.4.1 Interface

```
statusT advfs_snap_protect_cache (
       bfSetT  *parent_bf_set_ptr )   /* bfSetT pointer of parent fileset */
```

3.2.1.3.4.2 Description

advfs_snap_protect_cache is required to protect every page that is currently in cache from being written. This will cause all mmappers to fault into advfs_getpage to allow for COW processing. Until each page is protected, there is no locking that can be done by AdvFS to prevent modifications to the data. This routine was initially designed to use fcache_vn_protect but it was determined that the fcache_vn_protect interface had negative consequences with respect to performance and CFS. As a result, pages will be protected by flushing the pages with the FVF_PPAGE flag and a new private flag APP_MARK_READ_ONLY. The FVF_PPAGE flag will force a call into advfs_putpage even for clean pages while the APP_MARK_READ_ONLY flag will indicate to advfs_putpage that it should flush dirty data and set the pi_pg_ro flag in all in-cache pages.

advfs_snap_protect_cache is primarily responsible for making sure that all files of the parent fileset are set to be read-only and that the BFA_SNAP_CHANGE flag has been set on every file that may need to be COWed in advfs_getpage. The setting of files to be read-only is done via a call to fcache_vn_flush.

The routine will walk the access set list of the parent_bf_set_ptr (accessFwd) and for each file that is not in ACC_INVALID, ACC_DEALLOC, ACC_RECYCLE and is not a bfAccess magic marker (accMagic == ACCMAGIC_MARKER), it will call fcache_vn_flush on the file with the FVF_PPAGE and APP_MARK_READ_ONLY flags and then set the bfap's BFA_SNAP_CHANGE flag. The bfap will need the bfaLock to be held while setting the BFA_SNAP_CHANGE flag. Unfortunately, since fcache_vn_flush cannot be called while holding the accessChainLock, the same marker mechanism will be used to traverse the access set list as is used by advfs_bs_bfs_flush. Since any new files that are opened will be added to the front of the list, and since any new files opened will not be allowed to have writes performed (they will block in advfs_getpage or advfs_fs_write), it is safe to traverse the access set list and still allow new files to be added to the list.

advfs_snap_protect_cache is called in the context of an exclusive transaction and after having flushed the entire domain. Therefore, it is expected that the vast majority of pages found in cache in advfs_putpage will be clean. However, if a user data page is mmapped and if the mmapper modifies the page between the domain flush and the call to advfs_vn_flush to protect the page, a page may be found dirty. A metadata page should never been found dirty and in cache while in advfs_putpage with the APP_MARK_READ_ONLY flag.

On any error from fcache_vn_flush, this routine will return the error status to the caller. The routine will not unprotect previously protected pages. The snapshot fileset creation will be failed by the caller, and the pages that were already protected will suffer a potential performance impact until they have all faulted through advfs_getpage to reset the write permissions.

3.2.1.3.4.3 Execution Flow

- Lock parent_bf_set_ptr->accessChainLock
- cur_bfap = head of setList
- While not at the end of the setList
    - If cur_bfap is ACC_INVALID, ACC_DEALLOC, ACC_RECYCLE or is a ACCMAGIC_MARKER
        - cur_bfap = cur_bfap->setFwd
        - continue
    - Insert marker after cur_bfap
    - Drop accessChainLock
    - Try to lock the bfap flush lock for read.
    - if failure, skip the bfap and continue
    - fcache_vn_flush with FVF_PPAGE and APP_MARK_READ_ONLY on cur_bfap to make it READ only
    - if fcache_vn_flush fails,
        - lock accessChainLock
        - remove marker
        - unlock accessChainLock
        - free marker
        - return error
    - lock bfaLock
    - set BFA_SNAP_CHANGE flag in bfaFlags
    - unlock bfaLock
    - unlock bfap flush lock
    - lock accessChainLock
    - cur_bfap = marker->setFwd
    - remove marker from setList
- unlock accessChainLock
- return EOK

3.2.1.3.5 advfs_snap_drain_writes

3.2.1.3.5.1 Interface

```
statusT advfs_snap_drain_writes (
       bfSetT  *parent_bf_set_ptr )   /* bfSetT pointer of parent fileset */
```

3.2.1.3.5.2 Description

This routine is used to wait for all in progress writes that may cause data to become dirty after a domain flush. The routine will walk the list of access structures in the parent fileset and wait until the bfaWriteCnt to go to zero. If any files have writes in progress, the bfaLock will be acquired and the bfaSnapCv will be waited on. The last writer to decrement the bfaWriteCnt to zero will wake up the snapset creation thread.

The routine will use the same model as advfs_snap_protect_cache to walk the access chain list. For each file, if the bfaWriteCnt is not equal to zero, the bfaLock will be acquired, and the thread will sleep on the bfaSnapCv.

3.2.1.3.5.3 Execution Flow

- Lock parent_bf_set_ptr->accessChainLock
- cur_bfap = head of setList
- While not at the end of the setList
    - If cur_bfap is ACC_INVALID, ACC_DEALLOC, ACC_RECYCLE or is a ACCMAGIC_MARKER or is metadata
        - cur_bfap = cur_bfap->setFwd
        - continue
    - if cur_bfap->bfaWriteCnt == 0
        - cur_bfap = cur_bfap->setFwd
        - continue
    - Insert marker after cur_bfap
    - Drop accessChainLock
    - Lock cur_bfap->bfaLock
    - While cur_bfap->bfaWriteCnt
        - Cv_wait cur_bfap->bfaSnapCv
    - Unlock cur_bfap->bfaLock
    - lock accessChainLock
    - cur_bfap = marker->setFwd
    - remove marker from setList
- unlock accessChainLock
- return EOK

3.2.1.3.6 advfs_link_snapsets_full

3.2.1.3.6.1 Interface

```
statusT advfs_link_snapsets (
       bfSetT  *parent_bf_set_ptr,    /* bfSetT pointer of parent fileset */
       bfSetT  *child_bf_set_ptr,     /* bfSetT pointer of child fileset */
       snap_flags_t   snap_flags      /* Flags to indicate read or write snapshot */
       ftxHT   parent_ftx )           /* Transaction to use for updates */
```

3.2.1.3.6.2 Description

This routine will insert (on disk and in memory) child_bf_set_ptr into the list of children snapshots of parent_bf_set_ptr and copy necessary fields from the parent bfSetAttr to the child bfSetAttr and set the

child's on disk state to BFS_ODS_VALID. The update will be done in a root transaction and cannot be undone. After this routine, the parent_ftx cannot be failed.

This routine assumes that it is being called from an exclusive transaction and that the bfSetTbl lock is held for write mode.

The parent fileset will be linked to the snapshot fileset by following the bfsaFirstChildSnap pointer and then the bfsaNextSiblingSnap pointer to the end of the chain of child snapshots filesystems. If bfsaFirstChildSnap is a NULL bfSetId, then the parent will have bfsaFirstChild snap pointer to the new snapshot. The new snapset will be linked with the parent by having bfsaParentSnapSet set to the bfSetId of the parent filesystem and by having the bfsaSnapLevel set to one greater than the same value in the parent file set attributes field. The updates will happen to the fields in the bfSetAttr record of the tag directory file for each fileset.

At the same time that the on disk linkage is established, the parent and child will be linked in memory using the pointers in the bfSet structure.

3.2.1.3.6.3 Execution Flow

- ASSERT the bfSetTbl lock is held for write
- Read parent fileset's bfSetAttr record
- if parent_bf_set_ptr has a child snapset
    - prev_child = parent_bf_set_ptr->bfsFirstSnapChild
    - next_child = prev_child->bfsNextSnapSibling
    - while (next_child)
        - prev_child = next_child
        - next_child = next_child->bfsNextSnapSibling
    - read prev_child's bfSetAttr record
    - modify bfsaNextSnapSibling pointer to have child_bf_set_ptr's setId
    - update prev_child's on disk bfSetAttr (done under parent_ftx)
    - if update failed, the domain has paniced, return
    - modify the prev_child's bfsNextSnapSibling pointer to point to the child
- else
    - modify the parent's bfsaFirstSnapChild to have child_bf_set_ptr's setId
    - update parent's on disk bfSetAttr (done under parent_ftx)
    - if the update failed, the domain has paniced, return
    - modify the parent's bfsFristSnapChild pointer to point to the child
- read child's bfSetAttr
- modify bfsaParentSnapSet to have parent's setId
- modify bfsaSnapLevel to be one greater than parent's level
- set on disk state to BFS_ODS_VALID
- if snap_flags & SF_SNAP_READ
    - set mode bits in bfSetAttr to READ only
- else
    - ASSERT snap_flags & SF_SNAP_WRITE
    - Set mode bits in bfSetAttr to WRITE and READ
- update the child's on disk bfSetAttr record (done under parent_ftx)
- if the update failed, the domain has paniced, return error
- modify the child's bfsParentSnapSet pointer to point to the parent
- return EOK

3.2.1.3.7 advfs_fs_write

3.2.1.3.7.1 Interface

```
statusT advfs_fs_write ( … )
```

3.2.1.3.7.2 Description

This routine will be modified to synchronize with advfs_create_snapset. Before the file lock or the cachemode lock are acquired, the routine will synchronize with snapset creation by checking for the BFS_IM_SNAP_IN_PROGRESS flag. If the flag is set, the routine will block on the bfsSnapCv using the bfsSnapMutex to synchronize. If the BFS_IM_SNAP_IN_PROGRESS flag is not set, then bfap->bfaWriteCnt will be incremented atomically and the BFS_IM_SNAP_IN_PROGRESS flag will be checked again. If the flag is now set, the bfaWriteCnt field will be decremented and the bfaSnapCv will be broadcast before going to sleep on the bfsSnapCv (using the bfsSnapMutex to synchronize).

Before returning to the caller, and after dropping all locks, the bfaWriteCnt will be atomically decremented. The bfaWriteCnt will synchronize direct IO with snapset creation since t direct IO is called from advfs_fs_write.

3.2.1.3.7.3 Execution Flow

- …
- incr bfap->bfaWriteCnt
- while BFS_IM_SNAP_IN_PROGRESS
    - decr bfap->bfaWriteCnt
    - broadcast bfap->bfaSnapCv
    - mutex_lock bfSet->bfsSnapMutex
    - if BFS_IM_SNAP_IN_PROGRESS
        - cv_wait on bfsSnapCv NO_RELOCK
    - else
        - unlock bfSet->bfsSnapMutex
    - incr bfap->bfaWriteCnt
- Lock cachemode lock and file lock
- Perform normal write
- …
- decr bfap->bfaWriteCnt
- if bfap->bfaWriteCnt == 0 && BFS_IM_SNAP_IN_PROGRESS
    - mutex_lock bfap->bfaLock
    - cv_broadcast bfap->bfaSnapCv
    - mutex_unlock bfap->bfaLock
- return

3.2.1.3.8 advfs_putpage

3.2.1.3.8.1 Interface

```
statusT advfs_putpage ( … )
```

3.2.1.3.8.2 Description

advfs_putpage will be modified to handle the APP_MARK_READ_ONLY flag. When this flag is set, advfs_putpage must do a fcache_page_scan for both clean and dirty pages. All pages in cache must have the pi_pg_ro flag set and, additionally, dirty pages must be flushed. While the APP_MARK_READ_ONLY flag is set, a metadata file should never be found to have a dirty page.

3.2.1.3.8.3 Execution Flow

- …

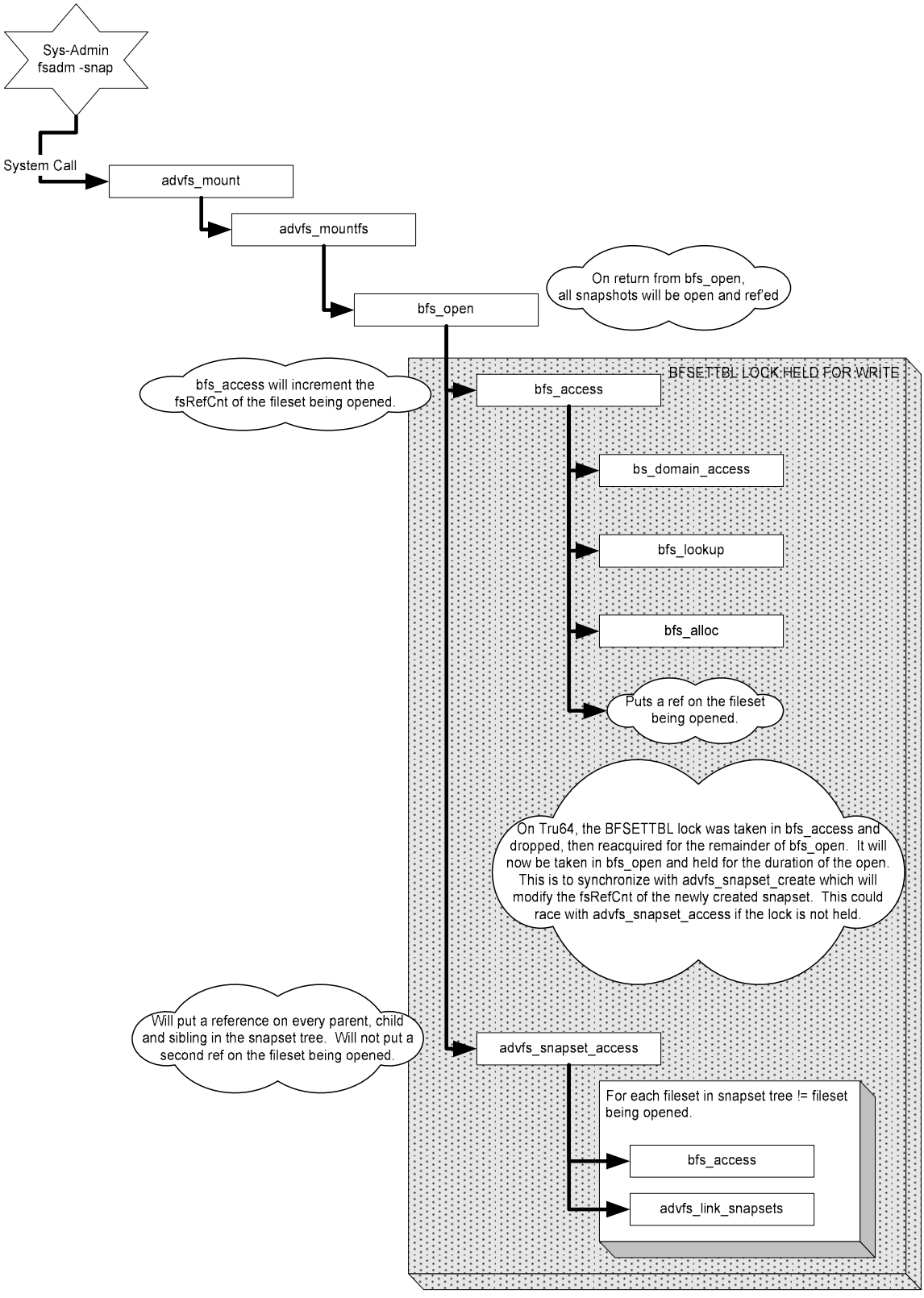
- if private_params exist & app_flags & APP_MIGRATE_FLUSH_ALL|APP_MIGRATE_FLUSH_WAIT|APP_MARK_READ_ONLY
    - ptype = FPS_GET_DIRTY|FPS_GET_CLEAN
- …
- while current_request < ending_request
    - …
    - fcache_page_scan
    - adjust for large pages
    - if APP_MARK_READ_ONLY
        - for each pfdat in plist
            - set pi_pg_ro flag
        - if FPS_ST_CLEAN
            - /* There is no need to continue in this routine in this loop, just move on to the next plist. */
            - fcache_page_release the pages
            - goto loop_end
        - else ASSERT(!metadata)
    - …

3.2.1.3.9 Miscellaneous Changes

When creating a new fileset, fs_fset_create will set the BFS_OD_ROOT_SNAPSHOT flag on disk to indicate the fileset has no dependency on a parent snapset. This flag is in support of potential future work to allow a snapshot child to be cleved from the parent. See Section 3.2.16 for further descriptions.

3.2.2 Opening a fileset

3.2.2.1 Function Call Tree Overview



3.2.2.2 Basic Operations of Opening a Snap Set

When a user issues a command that will cause a fileset to be opened (in the above example, a mount system call), the call will eventually resolve to bfs_open[5]. On returning from bfs_open, the fileset being opened will have a reference count incremented on it, and all filesets in the snapshot tree will have a reference count placed on them. In the case of RO snapshots, only the parent or child snapshot needs to have an additional reference placed on it. In the case of MW snapshots, the tree must be walked from top to bottom and every fileset needs to have bfs_access called to put a reference on the fileset.

Any call to bfs_open will result in all snapsets in the snapshot tree being opened and accessed. On entrance to bfs_open, the BFSETTBL lock will be held for write or not held at all. If the lock is not held for write, it will be acquired. The BFSETTBL lock will be held for write for the majority of the time bfs_open is processing the open request. The lock will synchronize with advfs_snapset_create which will match the number of accesses on the parent and the child filesets. If the BFSETTBL lock were dropped after bfs_access (as is done in Tru64) a call to advfs_snapset_create would result in an extra reference on the child fileset.

Since advfs_snapset_create will release the BFSETTBL lock while creating the snapset, if the lock is not held on entrance to this routine, it will be acquired and the fileset will be accessed. If, after accessing the fileset, it is determined that the BFS_IM_SNAP_IN_PROGRESS flag is set in the parent or the accessed fileset, the fileset will be closed and the lock will be dropped. In the event that the fileset is closed, the thread will wait on the bfsSnapCv of the fileset to be accessed. Once the BFS_IM_SNAP_IN_PROGRESS flag is cleared, the thread will be woken up and will re-access the fileset and continue. If a thread holds the BFSETTBL lock for write when calling bfs_access, it must handle making sure that the BFS_IM_SNAP_IN_PROGRESS flag is not set.

bfs_open will first call bfs_access on the fileset being opened. This fileset may be either a parent fileset or a child snapset. In either case, on return from bfs_access, only the fileset requested will be open (assuming successful return). bfs_access will have no knowledge of snapshots or responsibility for setting them up. On successful return from bfs_access, advfs_snapset_access will be called to correctly open and access all related filesets.

advfs_snapset_access will call bfs_access on every fileset in the snapshot tree that is not the fileset that was already opened. In addition to opening the filesets, advfs_snapset_access will link the filesets together using the bfsParentSnapSet, bfsFirstChildSnapSet, and bfsNextSiblingSnapSet fields.

Before returning, bfs_open will drop the BFSETTBL lock if it acquired it. On error, any filesets that were accessed will be closed and all resources will be freed. In the event that opening a child snapshot fails, that child will be marked as out of sync.

3.2.2.3 Function Call Details

3.2.2.3.1 bfs_open

3.2.2.3.1.1 Interface

```
~~static~~ statusT
bfs_open( bfSetT      **retBfSetp,   /* out - pointer to open bitfile-set */
        bfSetIdT      bfSetId,       /* in - bitfile-set id to open */
        uint32_t      options,       /* in - options flags */
        ftxHT         ftxH           /* in - parent transaction handle */
```

[5] On Tru64, bfs_open is called via two wrappers, rbf_bfs_open and rbf_bfs_access. The two routines call the same function with a different flag and have no value-add, so they will be removed in favor of a direct call to bfs_open.

3.2.2.3.1.2 Description

This routine opens a fileset and all associated filesets in the snapshot tree. On Tru64, the first line of this routine called bfs_access to lookup, create and put a reference on the fileset to be opened. bfs_access would conditionally acquire the BFSETTBL lock for write and drop it before returning (if it had acquired the lock). The routine will be modified so that the BFSETTBL lock is acquired before bfs_access is called. This will synchronize with advfs_snapset_create so that snapsets do not end up with the wrong number of fsRefCnt's.

The remainder of the routine (the part not involved in acquiring the BFSETTBL lock or calling bfs_access) on Tru64 was responsible for correctly opening clones. This code will all be removed and the routine advfs_snapset_access will be called instead. On successful return from advfs_snapset_access, all parent, sibling and child snapsets will have an fsRefCnt on them for this open and the system will be ready to have data COWed.

Before returning, the BFSETTBL lock will be dropped if it was acquired at the beginning of the routine.

3.2.2.3.1.3 Execution Flow

- If the bfSetTbl lock is not held for write
  - Write lock the bfSetTbl lock
- Call bfs_access to access the fileset bfSetId
- If bfs_access fails
  - If bfSetTbl lock was acquired in this routine, drop the lock
  - Return error from bfs_access
- advfs_snapset_access on bf_set
- if advfs_snapset_access fails
  - /* Some serious error occurred on the domain and the children that couldn't be opened also could not be marked out of sync. */
  - close bf_set
  - drop bfSetTbl lock if acquired in this routine
  - return error
- if fileset is BFS_OD_OUT_OF_SYNC after advfs_snapset_access
  - Report that fileset is out of sync
- if bfSetTbl lock was acquired in this routine
  - drop bfSetTbl lock
- return EOK

3.2.2.3.2 bfs_access

3.2.2.3.2.1 Interface

```
static statusT
bfs_access( bfSetT     **retBfSetp,   /* out - pointer to open bitfile-set structure */
        bfSetIdT       bfSetId,       /* in - bitfile-set id to access */
        uint32_t       options,       /* in - options flags */
        ftxHT          ftxH )         /* in - parent transaction handle */
```

3.2.2.3.2.2 Description

This routine will be modified to no longer deal with Tru64 clones. All snapshot field initialization will occur in bfs_open or bfs_alloc. bfs_access will be a routine to access a single fileset and not its associated snapshots.

3.2.2.3.2.3 Execution Flow

No significant logic changes. Code dealing with clones will be removed.

3.2.2.3.3 bfs_alloc

3.2.2.3.3.1 Interface

```
static statusT
bfs_alloc( bfSetIdT    bfSetId,       /* in - bitfile-set id */
        domainT        *dmnP,         /* in - BF-set's domain's struct pointer */
        bfSetT         **retBfSetp    /* out - ptr to BF-set's descriptor */
```

3.2.2.3.3.2 Description

This routine will be modified to no longer initialize the obsolete cloneDelStateMutex and related fields. The routine will also be modified to initialize the snapshot related fields of the bfSet to be NULL pointers. The fields will be fully setup by advfs_snapset_access.

3.2.2.3.3.3 Execution Flow

No significant logic changes. Changes will be made to initialize new fields as described.

3.2.2.3.4 advfs_snapset_access

3.2.2.3.4.1 Interface

```
statusT advfs_snapset_access(
        bfSetT  *bf_set_ptr,   /* The fileset that is being opened.  This will not be
                                * re-accessed by this routine */
        ftxHT   parent_ftx)    /* The parent transaction to be used during this access */
```

3.2.2.3.4.2 Description

This routine is a wrapper for calling bfs_access on every fileset in the snapset tree that is not the same fileset as the bf_set_ptr passed in. The basic algorithm will be to follow the parent snapshot pointer in the bitfile set attributes record of the tag directory until the parent of the snapshot tree is reached. Once the root of the tree is reached, the entire tree will be traversed in a prefix-order and each fileset will have bfs_access called on it. After having bfs_access called, the fileset will be linked to its parent and children. When a child is accessed, it will be pointed to by either its parent or one of its siblings on the same level. After each call to bfs_access, the bfsSnapRefs field of the bfSet structure will be incremented to reflect that it has been accessed by a related snapset.

It is expected that the calling routine will hold the BFSETTBL lock for the domain in write mode. The routine will return with the lock still held.

If this routine encounters an error while trying to open a fileset, if bf_set_ptr being opened has not yet been reached in the traversal of the snapshot tree, then a hard error must be returned and the bfs_open must fail (we are a snapshot and our parents weren't opened). If, however, the fileset being opened has already been traversed in the snapshot tree, and a failure occurs, the fileset that failed to open and all its child snapsets will have the BFS_OD_OUT_OF_SYNC_SNAP set in the bfSetAttr record of the fileset. If the setting of the out of sync flag fails, then a hard error will be returned and the bfs_open will fail. Otherwise, all snapsets that can be opened will be opened and those that cannot be opened will be marked as out of sync.

While traversing the snapset tree, if a snapset is found that has the BFS_OD_OUT_OF_SYNC set, all it's children will have the BFS_OD_OUT_OF_SYNC set. On return from this routine, bf_set_ptr may be marked as out of sync.

This routine will be optimized to use the in memory snapset pointers if bfsSnapRefs is non-zero. In the case of a non-zero bfsSnapRefs count, IO can be avoided reading the bfSetAttr record for each fileset since the pointers are valid in cache (and will remain so as long as the bfSetTbl lock is held).

3.2.2.3.4.3 Execution Flow

- ASSERT bfSetTbl Lock is held for write.
- Read the bfSetAttr of bf_set_ptr to get

- parent_set_id = bfSetAttr->bfsaParentSnapSet
- /* Walk up the snapset tree to the root */
- while (parent_set_id != NilBfSetId)
    - top_level_parent = parent_set_id
    - bs_access_one the tag file of the parent set
    - if bs_access_one gets an error, return the error
    - read the bfSetAttr of the tag file
    - parent_set_id = bfSetAttr->bfsaParentSnapSet
    - bs_close_one the tag file of the parent set
- /* Top level parent is now the root */
- advfs_snapset_access_recursive(bf_set_ptr, top_level_parent, parent_ftx)
- if advfs_snapset_access_recursive fails, return the error
- return EOK

3.2.2.3.5 advfs_snapset_access_recursive

3.2.2.3.5.1 Interface

```
statusT advfs_snapset_access_recursive(
       bfSetT  *bf_set_ptr,   /* The fileset that is being opened.  This will not be
                               * re-accessed by this routine */
       bfSetT *parent_set_p   /* Parent of cur_bf_set_id */
       bfSetId cur_bf_set_id  /* The current bfSet.  The top level call should have this
                               * as NULL */
       snap_flags_t   *snap_flags    /* Flags */
       ftxHT   parent_ftx)    /* The parent transaction to be used during this access */
```

3.2.2.3.5.2 Description

This routine will recursively access each fileset in the snapset tree that is not the same fileset as bf_set_ptr. If this routine encounters an error, it will attempt to mark any children filesets as out of sync. If the children cannot be marked out of sync, all filesets already open will be closed and an error will be returned.

The recursion will be done in a pre-order traversal where the parent is opened before any children. As a fileset is accessed, it will be linked to its parent and the end of its parents list of children.

This routine relies on recursion to open any number of snapsets; however, in practice, the number of snapsets ought to be bounded to prevent deep recursive calls on the kernel stack.

3.2.2.3.5.3 Execution Flow

- Read bfSetAttr for cur_bf_set_id
- if snap_flags & SF_OUT_OF_SYNC
    - mark bfSet as out of sync on disk (advfs_snap_out_of_sync)
- if bfSetAttr has BFS_OD_OUT_OF_SYNC set
    - snap_flags & SF_OUT_OF_SYNC
- if cur_bf_set_id != bf_set_ptr->bfSetId
    - bfs_access(cur_bf_set_id, cur_set_ptr)
- else
    - cur_set_ptr = bf_set_ptr
    - *snap_flags &= SF_FOUND_SNAPSET
- if bfs_access fails and snap_flags & SF_FOUND_SNAPSET
    - Attempt to open tagdir file for cur_bf_set_id and update with BFS_OD_OUT_OF_SYNC.
    - If update fails a domain panic occurred.
    - Return error

- Else if bfs_access fails
    - /* We haven't yet reached the fileset we are opening in the
      * snapset tree so we can't mark it's children out of sync. Just
      * fail the open completely */
    - if SF_OUT_OF_SYNC was set in this call, clear it
    - return error
- if cur_set_ptr != bf_set_ptr
    - cur_set_ptr->bfsSnapRefs++
- advfs_link_snapsets(parent_set_p, cur_set_ptr)
- /* recurse to first child if it exists */
- if bfSetAttr->bfsaFirstSnapChild != bfSetNilId
    - advfs_snapset_access_recursive(bf_set_ptr,
      cur_set_ptr,
      bfSetAttr->bfsaFirstSnapChild,
      parent_ftx)
    - if advfs_snapset_access_recursive fails
        - cur_set_ptr->bfsSnapRefs--
        - bfs_close cur_set_ptr
        - if SF_OUT_OF_SYNC was set in this call, clear it
        - return the error
- /* walk list of siblings if any exist */
- if bfSetAttr->bfsaNextSnapSibling != bfSetNilId
    - advfs_snapset_access_recursive(bf_set_ptr,
      cur_set_ptr,
      bfSetAttr->bfsaNextSnapSibling,
      parent_ftx)
    - if advfs_snapset_access_recursive fails
        - cur_set_ptr->bfsSnapRefs—
        - For each child up to the child that failed
            - advfs_snapset_close_recursive(bf_set_ptr, child_set, parent_ftx)
        - bfs_close cur_set_ptr
        - if SF_OUT_OF_SYNC was set in this call, clear it
        - return the error
- if SF_OUT_OF_SYNC was set in this call, clear it
- return EOK

3.2.2.3.6 advfs_link_snapsets

3.2.2.3.6.1 Interface

```
statusT advfs_link_snapsets(
       bfSetT  *parent_set_ptr,       /* The parent to be linked to child. */
       bfSetT  *child_set_ptr         /* The child to be linked to parent */
```

3.2.2.3.6.2 Description

This routine is intended to link the child_set_ptr to it's parent and the child_set_ptr to the end of the parent's snapset list. The linking is done only in memory since it was already done on disk at snapset creation. This routine assumes the bfSetTbl lock is held for write access.

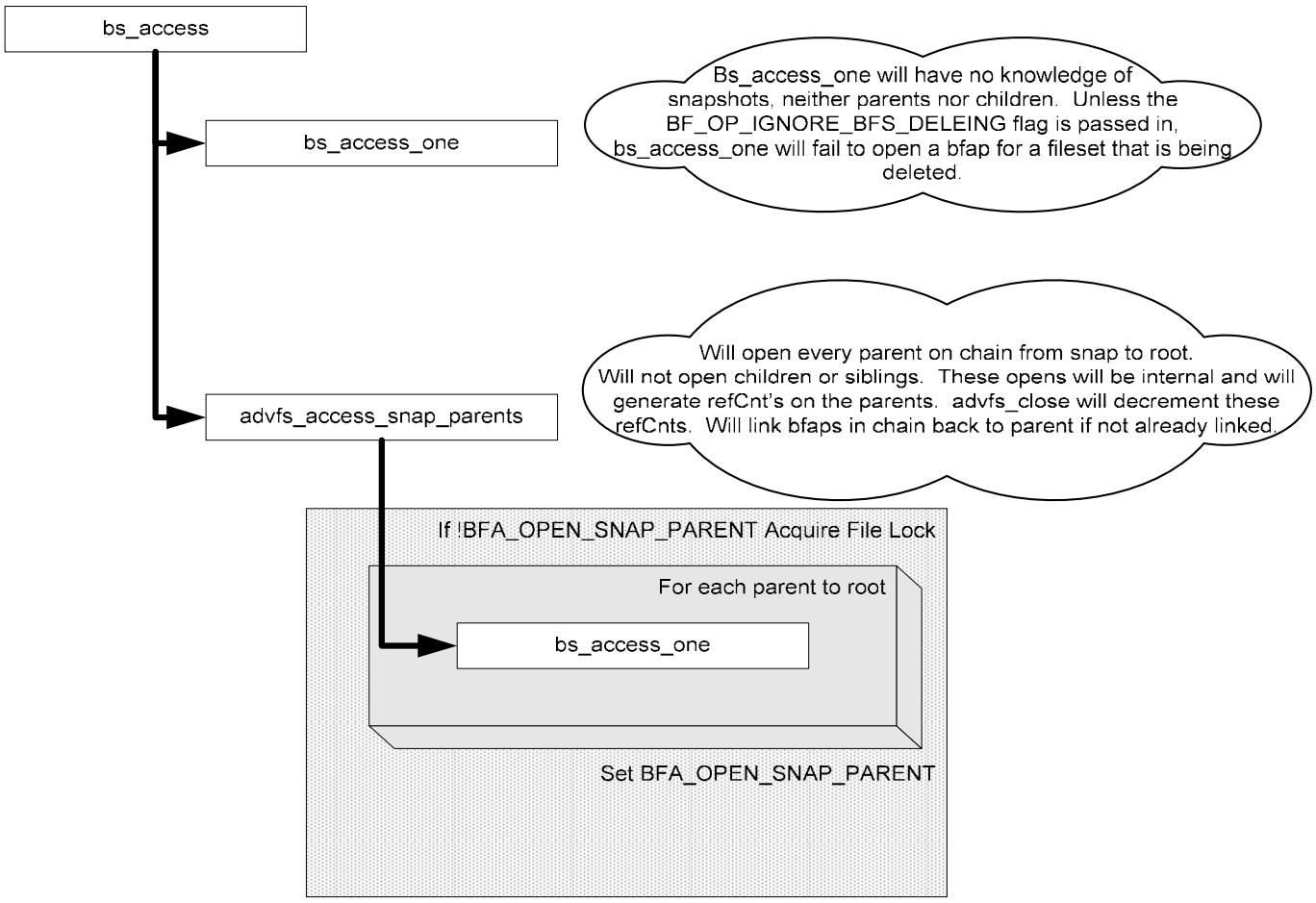
3.2.2.3.6.3 Execution Flow

- ASSERT the bfSetTbl lock is held for write access
- child_set_ptr->bfaParentSnap = parent_set_ptr

- `if parent_set_ptr->bfsFirstSnapChild == NULL`
    - `parent_set_ptr->bfsFirstSnapChild = child_set_ptr`
    - `return EOK`
- `else`
    - `prev_child = parent_set_ptr->bfsFirstSnapChild`
    - `next_child = prev_child->bfsNextSnapSibling`
    - `while next_child`
        - `prev_child = next_child`
        - `next_child = next_child->bfsNextSnapSibling`
    - `prev_child->bfsNextSnapSibling = child_set_ptr`
    - `return EOK`

3.2.3 Opening a file

3.2.3.1 Function Call Tree Overview



3.2.3.2 Basic Operations of Opening a File

Opening a file internally or externally enters a common code path at bs_access. On Tru64, bs_access opens the original file if the fileset of the file being opened is an original fileset and opens the original and then the clone if the file being opened is a clone.

On HPUX, the logic will be simplified somewhat. bs_access will begin by attempting to open the file being opened (a call to bs_access_one) whether that file is a snapshot, a parent or both a snapshot and a parent of a snapshot. If bs_access_one succeeds in opening the file requested, then advfs_access_snap_parents will be called to open all the parents of the requested file. Since a fileset cannot gain a new parent, the advfs_access_snap_parents will have a fast exit case if there are no parent filesets, and there is no need to hold locks during this check.

If advfs_access_snap_parents fails to open any parents (by calling bs_access_one on each of them) it will close all parent files that were successfully opened and return an error to bs_access. advfs_access_snap_parents will also output an error message to the console and to the terminal to indicate that the open failed because of the parent file open error.

If advfs_access_snap_parent fails, bs_access will close the opened file and return an error.

It is not necessary to open any child snapshots at this point since we may only be opening the file to read from it, in which case there is no need for us to open the child snapshots.

3.2.3.3 Function Call Details

3.2.3.3.1 bs_access

3.2.3.3.1.1 Interface

```
statusT
bs_access(
          bfAccessT **outbfap,     /* out - access structure pointer */
          bfTagT tag,              /* in - tag of bf to access */
          bfSetT *bfSetp,          /* in - BF-set descriptor pointer */
          ftxHT ftxH,              /* in - ftx handle */
          enum acc_open_flags options,  /* in - options flags */
          struct vnode **vp        /* in/out - from bs_access_one */
```

3.2.3.3.1.2 Description

This routine will be simplified to no longer check for BS_BFSET_ORIG and make conditional decisions on which files to open. Instead, the routine will always call bs_access_one on the file being requested to be open. The call to bs_access_one will open the file either externally or internally depending on the options parameter to bs_access.

If the bs_access_one call is successful, then bs_access will call advfs_access_snap_parents to open each file starting at the file just opened and moving up to the root of the snapshot tree. If any open fails along the way to the parent, bs_access will get an error from advfs_access_snap_parents and will close the file and return an error along with outputting an error message to the console and terminal.

The parameters origBfap and fsvp are to be removed from bs_access_one. bs_access will set *vp to &outbfap->bfVnode before returning successful.

In the event that an access is attempted on a file that is out of sync, the access will be failed. Since reads of out of sync files will fail, opens will also fail. Since the check is racy, it is possible to open a file which immediately becomes out of sync. Any attempts to access data in the file will fail.

3.2.3.3.1.3 Execution Flow

- `bs_access_one tag in bfSetp to get bfap`
- `if bs_access_one fails`
    - `return error`
- `if !(bfaFlags & BFA_ROOT_SNAPSHOT)`
    - `advfs_access_snap_parents`
    - `if advfs_access_snap_parents fails`
        - `bs_close_one bfap`
        - `return error`
- `*outbfap = bfap`
- `*vp = &bfap->bfVnode`

3.2.3.3.2 bs_access_one

3.2.3.3.2.1 Interface

```
statusT
bs_access_one(
              bfAccessT **outbfap, /* out - access structure pointer */
              bfTagT tag,          /* in - tag of bf to access */
```

```
                bfSetT *bfSetp,      /* in - BF-set descriptor pointer */
                ftxHT ftxH,          /* in - ftx handle */
                enum acc_open_flags options, /* in - options flags */
                struct vnode** fsvp, /* out - The vnode for this access structure
                                                This is redundant with the outbfap */
                bfAccessT *origBfap  /* in - Orig access (clone open) */
```

3.2.3.3.2.2 Description

bs_access_one is the interface for retrieving a single files access structure without incurring any penalties for opening related snapshot files. The routine will not know anything about snapshots except for a check to see if the bfSetp->bfsParentSnapSet is non NULL and checking for an original file size for a snapshot. If the value of bfsParentSnapSet is non NULL, then the bfaFlags BFA_QUICK_CACHE will be set. This flag will allow the snapshot to be processed early if it is on the free or closed list[6].

The routine will have the parameters origBfap and fsvp removed. It will be the responsibility of the caller to get the vnode from the outbfap->bfVnode field. Previously, the origBfap parameter was used to test whether a clone was being opened and the clone fileset was being deleted. If it was the case that the clone fileset was BFS_DELETING, origbfap was set (indicating that a clone was being opened) and the clone shared metadata with the parent (the primary mcell id's were equal) then ENO_SUCH_TAG was returned. Now, if BFS_DELETING is set, then ENO_SUCH_TAG will be returned unless a new acc_open_flags flag is passed in options parameter to indicate that the delete should be ignored. The new flag, BF_OP_IGNORE_BFS_DELETING is necessary in the fileset delete code path when the file must be accessed for rbf_delete is called.

3.2.3.3.2.3 Execution Flow

- advfs_lookup_valid_bfap
- if bfap state is ACC_VALID
    - No logic changes
- else if state is ACC_INIT_TRANS or ACC_CREATING
    - unlock bfaLock
    - tagdir lookup
    - if bfSet flags & BFS_DELETING and not acc_open_flags & BF_OP_IGNORE_BFS_DELETING
        - sts = ENO_SUCH_TAG
        - goto err_setinvalid
    - if bfState is BSRA_INVALID
        - bs_map_bf
    - lock bfaLock
- No logic changes for bfState conditions
- Remove origBfap processing after BF_OP_INTERNAL conditions
- if bfSet->bfsParentSnapSet != NULL
    - if parent snapset is read only
        - bfap->bfaFlags |= BFA_QUICK_CACHE
    - if tagFlags & BS_TD_VIRGIN_SNAP set for bfap
        - bfap->bfaFlags |= BFA_SNAP_VIRGIN
- unlock bfaLock
- remove setting of fsvp and outbfap

3.2.3.3.3 advfs_lookup_valid_bfap

---

[6] The entire free and closed lists will not be walked looking for BFA_QUICK_CACHE access structures. Instead, if they are encountered, they will be aged more quickly, but they may age the full length of time if other bfaps without the BFA_QUICK_CACHE flag are ahead of them on the free or closed list.

3.2.3.3.3.1 Interface

```
bfAccessT*
advfs_lookup_valid_bfap(
           bfSetT *bfSetp,           /* in - bitfile-set handle */
           bfTagT tag,               /* in - bitfile tag */
           enum acc_open_flags options  /* in - options flags */
           )
```

3.2.3.3.3.2 Description

advfs_lookup_valid_bfap will be modified to conditionally increment bfaRefsFromChildSnaps if the acc_open_flags BF_OP_SNAP_REF flag whenever refCnt is bumped. bfaRefsFromChildSnaps will only be conditionally bumped if refCnt is also bumped.

advfs_lookup_valid_bfap also initialize bfa_orig_file_size to ADVFS_ROOT_SNAPSHOT (-1).

3.2.3.3.3.3 Execution Flow

- find_bfap
- if found
    - if in valid state
        - if BF_OP_INTERNAL
            - advfs_ref_bfap
            - if BF_OP_SNAP_REF
                - bfap->bfaRefsFromChildSnaps++
        - else
            - if not bfap->bfaFlags & BFA_EXT_OPEN
                - advfs_ref_bfap
                - if BF_OP_SNAP_REF
                    - bfap->bfaRefsFromChildSnaps++
        - return bfap
- advfs_get_new_access
- initialize access structure
- if BF_OP_SNAP_REF
    - bfap->bfaRefsFromChildSnaps++
- bfap->bfa_orig_file_size = ADVFS_ROOT_SNAPSHOT
- return

3.2.3.3.4 advfs_access_snap_parents

3.2.3.3.4.1 Interface

```
statusT
advfs_access_snap_parents(
             bfAccess* bfap,      /* in - file to be accessed */
             bfSetT *bf_set_ptr,  /* in - bfSet pointer of fileset to start
                                   * walk at.  */
             ftxHT ftxH)          /* in - ftx handle */
```

3.2.3.3.4.2 Description

This routine will conditionally call bs_access_one on all parent snapshots of bfap. If bfap->bfaFlags & BFA_PARENT_SNAP_OPEN is set, then there is no work to be done and advfs_access_snap_parents will return success. If the BFA_OPENING_PARENTS flag is set, then another thread is racing to open the parent files and this thread will block on the bfaSnapCv.

In the event that the BFA_OPENING_PARENTS flag is not set, the flag will be set while holding the bfaSnapLock for write. This flag will cause any racing openers to block until the BFA_PARENT_SNAP_OPEN flag is set and the BFA_OPENING_PARENTS flag is cleared.

Accessing the parents will consist of opening the immediate parent, acquiring the bfaSnapLock of the child and pointing the child to the parent. The child's bfaSnapLock will be dropped and the process will move up one level of the snapshot tree (the parent will be the child and its parent will be opened).

When accessing the parents, the call to bs_access_one will pass the BF_OP_SNAP_REF flag to force the bfaRefsFromChildSnaps field of the parents to be incremented along with the refCnt. In most cases, an error will cause all previously opened parents to be closed and an error will be returned. The one exception to this would be if ENO_SUCH_TAG was returned. In the case of MW snapshots, a file may exists at level 1, but not level 0, therefore, when ENO_SUCH_TAG is encountered, walking up the parent filesets will stop.

This routine will link the access structure to their parents as the parents are opened. On error, any links already setup will be left setup. There is no need to tear down links. Once all parents are opened, the bfaSnapLock will be acquired and the BFA_PARENT_SNAP_OPEN flag will be set in the bfap passed in to indicate that a reference has been placed on all the parent bfaps. The BFA_OPENING_PARENTS flag will be clear and a cv broadcast will occur to wake up any other threads trying to open the same access structure.

As the chain of parents is being traversed and opened, if a bfap is accessed and has a bfa_orig_file_size of ADVFS_ROOT_SNAPSHOT (-1), then the bfa_orig_file_size must be initialized. The bfa_orig_file_size will be initialized in bs_map_bf inside the call to bs_access_one.

After accessing all parent snapshots of a file, it is necessary to set the file's bfaNextFob field. Setting this field for metadata ensures that the file_size is correctly calculated. The bfaNextFob will be set to the max of the bfa_orig_file_size in fobs rounded up to an allocation unit, and either the bfa_orig_file_size of the first parent to have a bfa_orig_file_size, or the bfaNextFob of the first parent to be a root snapshot.

It is expected that bfap being accessed was reference prior to this routine be called. As a result, this routine will synchronize with bs_close since bs_close will not try to process a "last close" and will therefore not try to close the parent files.

3.2.3.3.4.3 Execution Flow

- If BFA_PARENTS_OPEN is set
    - return EOK
- write lock bfap->bfaSnapLock
- if BFA_OPENING_PARENTS is set
    - while BFA_PARENT_SNAP_OPEN is not set
        - cv_wait on bfaSnapCv
    - unlock bfaSnapLock
    - if BFA_OPENING_PARENTS not set, start over (another thread failed the open)
    - return EOK
- else
    - lock bfaLock
    - set BFA_OPENING_PARENTS
    - unlock bfaLock
    - unlock bfaSnapLock
- /* Start accessing parents until parent is NULL */
- child = bfap
- while (child->bfSet->bfsParentSnapSet != NULL)
    - parent = bs_access_one(child->bfSet->bfsParentSnapSet, bfap->tag, BF_OP_SNAP_REF)
    - if parent open returns ENO_SUCH_TAG, break

        - If parent open return any other error
            - Close any parents already opened by this loop
            - Lock bfaSnapLock
            - Lock bfaLock
            - Clear BFA_OPENING_PARENTS
            - Unlock locks
            - cv_broadcast bfaSnapCv
            - return error
        - write lock child->bfaSnapLock
        - child->bfaParentSnapShot = parent
        - unlock child->bfaSnapLock
        - child = parent
        - if (child->bfaFlags & BFA_ROOT_SNAPSHOT)
            - break
- if (bfap->bfa_orig_file_size != 0)
    - ASSERT(bfa_orig_file_size != ADVFS_ROOT_SNAPSHOT)
    - bfap->bfaNextFob = roundup(OFFSET_TO_FOB_UP(bfap->bfa_orig_file_size), bfap->bfPageSz)
- else
    - while parent bfa_orig_file_size == 0 && (bfap->primMCId == parent->primMCId) && parent's parent != NULL
        - parent = parent's parent
    - if parent->bfa_orig_file_size == ADVFS_ROOT_SNAPSHOT
        - bfaNextFob = parent->bfaNextFob
    - else
        - ASSERT(bfa_orig_file_size != ADVFS_ROOT_SNAPSHOT)
        - bfap->bfaNextFob = roundup(OFFSET_TO_FOB_UP(bfap->bfa_orig_file_size), bfap->bfPageSz)
- write lock bfap->bfaSnapLock
- lock bfaLock
- set BFA_PARENT_SNAP_OPEN
- clear BFA_OPENING_PARENTS
- unlock bfaLock
- unlock bfap->bfaSnapLock
- cv_broadcase on bfaSnapCv
- return EOK

3.2.3.3.5 bs_map_bf

3.2.3.3.5.1 Interface

```
statusT
bs_map_bf(
       bfAccessT*     bfap,          /* in/out - ptr to bitfile's access struct */
       enum acc_open_flags options,  /* in - options flags (see bs_access.h) */
       bfTagFlagsT    tagFlags       /* in - flags to set various bfap values */
```

3.2.3.3.5.2 Description

bs_map_bf will have a few slight changes made so that it correctly initializes the bfa_orig_file_size field of snapshot children.

If a snapshot has its own metadata, the bsBfAttr fied bfat_orig_file_size will be looked up and the original file size set based on that record. If the file does not yet have its own metadata, then the original file size will be initialized based on the parent's file size for userdata and the parents bfaNextFob for metadata.

In order to make sure that the parents file size is valid, the initialization of the fsContext structure will be moved from bf_get_l into bs_map_bf. On Tru64, the fsContext structure did not necessarily exist in bs_map_bf and could not be initialized for internal opens. On HPUX, the fsContext structure is embedded and can be initialized in bs_map_bf whether the file is being opened internally or externally.

3.2.3.3.5.3 Execution Flow

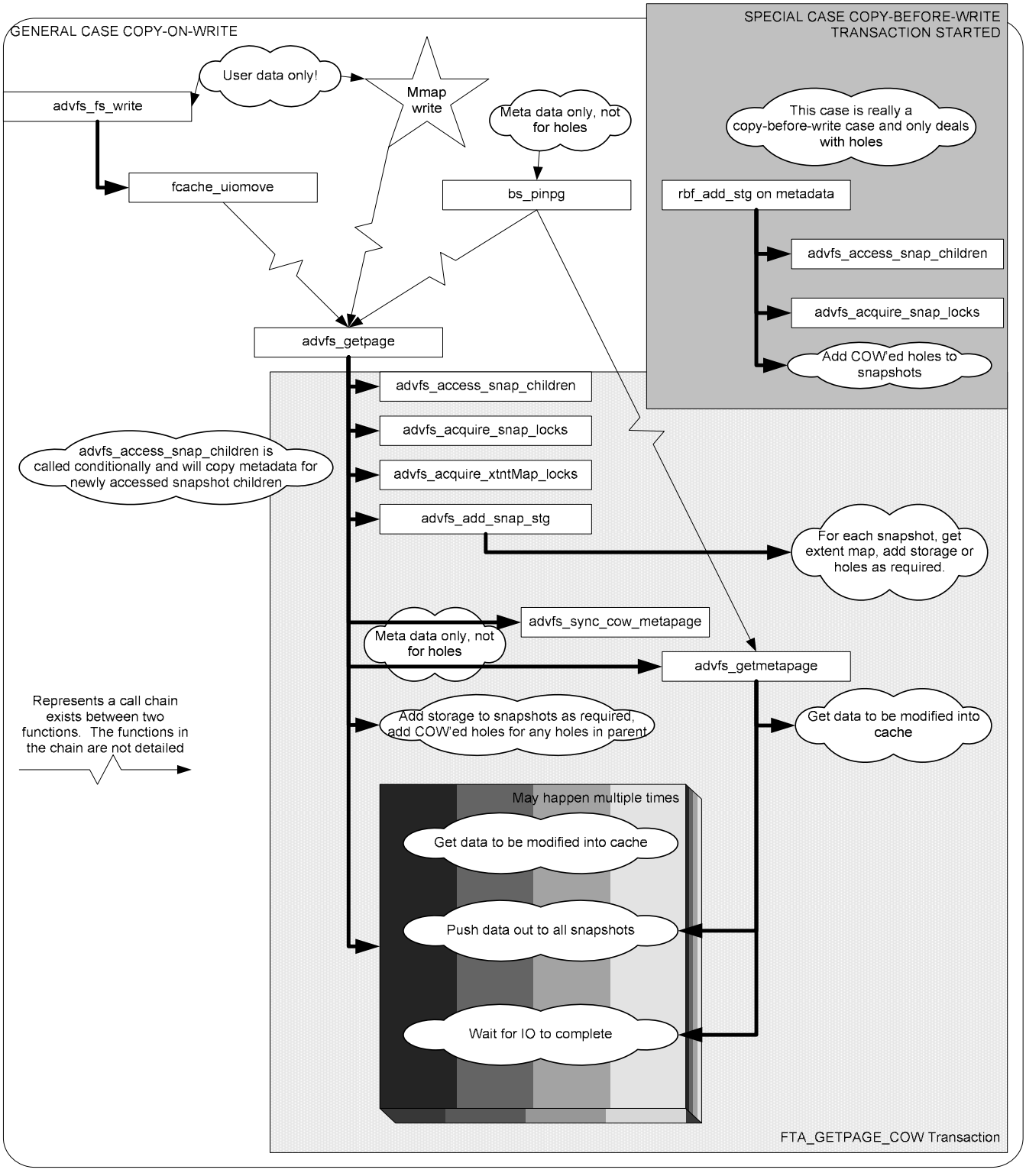
- When reading the bfAttr, if BOF_ROOT_SNAPSHOT is set, set BFA_ROOT_SNAPSHOT.
- …
- if bfSet->bfsParentSnapSet != NULL and bfap is COW-able (not reserved or a tag dir)
    - if tagFlags & BS_TD_VIRGIN_SNAP set for bfap
        - if bfap is metadata
            - bfap->orig_file_size = bfap->bfaNextFob * ADVFS_FOB_SZ
        - else
            - bfap->orig_file_size = bfap->file_size
    - else
        - get for BSR_BFATTR
        - bfap->bfa_orig_file_size = bfAttr->bfat_orig_file_size

3.2.3.3.6 Miscellaneous Changes

advfs_init_access will be modified to initialize snapshot pointers in the bfAccess structure.

3.2.4 Writing to a file (Copy-on-Write processing)

3.2.4.1 COW Overview



3.2.4.2 Basic Operation of Copy-On-Write

Copy-On-Write (COW) processing occurs primarily in advfs_getpage. The basic set of operations required to complete a COW includes synchronizing with any snapshot creations, acquiring locks to protect the range to be COWed, adding storage to any child snapshots as required, issuing any necessary writes to children snapshots and waiting for all IOs to complete. The majority of these steps are performed under transaction control to ensure that a system crash leaves the snapshots in a valid state.

Copy-on-Write operations will happen in one of several possible ways. The most common COW path will be through advfs_getpage when being called from advfs_fs_write. In this general case, the data to be COWed will necessarily be user data (metadata can't come through advfs_fs_write). The case of advfs_fs_write is nearly the same case as an mmap fault for write. In both cases, the bfaSnapLock (write) and the migStg_lk (read) for all child snapshots will be acquired by a call to advfs_acquire_snap_locks. Once the locks are acquired, it is safe to potentially add storage to the snapshot children.

Storage will be added to all child snapshots via a call to advfs_add_snap_stg. advfs_add_snap_stg will compare the extent maps of the faulted-on bfap and the unmapped extents of each child snapshot and add storage or COWed holes as appropriate. Any holes that exist in the file to be faulted (logical holes, not unmapped regions) will be inserted into the snapshot children as COW'ed holes[7]. Storage will only be added for parts of the snapshot that are unmapped and had storage in a parent.

Once storage is allocated for all the snapshot children, advfs_getpage will process the fault request but will use IO anchors to push out writes to all child snapshots. In the event that a write to a snapshot requires a COW from the parent and to the children snapshots, the write to the children will happen first, and the write to the faulted-on file will occur once storage has been allocated in the normal advfs_getpage code path for the faulted-on file.

Before advfs_getpage acquires any necessary locks, the routine will synchronize with the creation of snapshots by waiting on the BFS_IM_SNAP_IN_PROGRESS flag. Also, prior to acquiring locks, advfs_getpage will check the BFA_SNAP_CHANGE flag in the bfap being faulted on. If the flag is set or if the bfaFirstSnapChild pointer is NULL, then advfs_access_snap_children will be called to put a reference on any children that are not already linked into the faulted-on bfaps snapshot tree. This routine will handle linking in any new snapshot children and making a copy of the parent's metadata for the snapshot child.

When advfs_getpage is called from bs_pinpg, it may require COWs on metadata. advfs_getpage will never need to create holes in the extent maps of snapshot children for metadata. Instead, advfs_getpage will only need to COW storage-backed metadata. For metadata, the process of getting storage for snapshot children will be similar to user data files with the exception that the faulted-on file may have storage acquired for it if it is an unmapped range of a snapshot that is being written[8]. For a child metadata snapshot being written to in a region that is unmapped, storage will be added to the snapshot child and any of its children prior to calling advfs_getmetapage. advfs_getmetapage will be called to bring the data into the cache and the data to be COWed. If the page to be written is found in cache, advfs_getmetapage will kick off an IO for each child snapshot. The IOAnchor will be returned to advfs_getpage and the necessary COW processing will be finished there. If the data isn't found in the cache, then a read will be initiated in advfs_getmetapage and the IO anchor will be returned to advfs_getpage. In this case, advfs_getpage will be responsible for kicking off writes to all child snapshots and potentially newly allocated storage for the file on which the fault is occurring.

---

[7] COWed holes are replacing permanent holes since holes in a writable snapshot are not really permanent. A mechanism is required to distinguish between an unmapped region of a snapshot and a hole that has been COWed. In a snapshot, a normal hole represents an unmapped region whereas a COWed hole represents a hole in the parent that has been COWed and not filled.

[8] advfs_getmetapage is not designed to allow a write to a hole in a metadata page. advfs_bs_add_stg will synchronously COW a hole in a metadata page so that advfs_getmetapage always sees storage backing for meta writes.

Write optimization for files that require any COW processing will be ignored. If a file requires any COW processing and any pages are not already in cache, then it is necessary to do a read from disk prior to doing the COW.

On exit from advfs_getpage, all locks acquired in advfs_getpage will be dropped.

The final scenario for transferring extent information from a parent to a child snapshot is in the case of adding storage on a metadata file. Because metadata have storage added before bs_pinpg is called to fault in the page, waiting until advfs_getpage to do COW processing on metadata holes would be too late. From advfs_getpage's perspective, there is no way to tell the difference between newly allocated storage and a fault for a page that is not in cache. advfs_getpage has no way to differentiate uninitialized storage from initialized storage, and adding a flag to be passed down through bs_pinpg and into advfs_getpage leaves too much room for programming error that may cause data corruption or data reuse problems. Instead, when adding storage to a metadata file, rbf_add_stg will insert COWed holes into each of its snapshot children. Since rbf_add_stg is done in a transactional context, and since inserting COWed holes also requires a transaction, synchronization with creating new snapshots is provided by the fact that advfs_create_snapset starts an exclusive transaction.

rbf_create_stg will call advfs_access_snap_children if bfaFirstSnapChild is NULL or if BFA_SNAP_CHANGE is set in the bfaFlags. Once the children are setup, each child's migStg_lk will be acquired and it will have a COWed hole inserted into its extent map via advfs_make_cowed_hole if the extent map does not already have storage. For each snapshot that has a COWed hole inserted, the transaction will have a special done mode so that the COWed hole is left in the event that the higher level add storage transaction fails.

3.2.4.3 Function Call Detail

3.2.4.3.1 advfs_getpage

3.2.4.3.1.1 Interface

```
int
advfs_getpage(
    fcache_vminfo_t *fc_vminfo,   /* An opaque pointer to a vm data struct*/
    struct vnode * vp,           /* The vnode pointer */
    off_t *off,                   /* The offset in the file of the fault */
    size_t *size,                 /* The size in bytes to fault in */
    fcache_ftype_t ftype,            /* The fault type */
    struct advfs_pvt_param *fs_priv_param, /* File system private parameter */
    fcache_pflags_t pflags )     /* Options or modifiers to the function */
```

3.2.4.3.1.2 Description

advfs_getpage will be the primary hub for all copy-on-write activity in AdvFS. advfs_getpage will be responsible for making sure that any snapshots that may need opening are opened and that any of the opened snapshots have a copy of the parent snapshot's metadata. To help minimize the impact that COW processing has on code paths when snapshots are not enabled, advfs_getpage will attempt to make racy checks for snapshots before doing any snapshot processing.

Before advfs_getpage can do any COW processing, it must first synchronize mmap writers with any in progress advfs_create_snapset calls. If advfs_create_snapset has already started execution, then the BFS_IM_SNAP_IN_PROGRESS flag will be set in the bfSet of the bfap being faulted on. If the flag is set and the fault is an mmap write request, then the thread will wait on the bfsSnapCv in the bfSet. On waking up from sleeping on the bfsSnapCv, the BFS_IM_SNAP_IN_PROGRESS flag should be cleared unless another snapset was racing the create. advfs_getpage will wait until the BFS_IM_SNAP_IN_PROGRESS flag is cleared before continuing. For non-mmap writers, the synchronization with advfs_create_snapset was done either in advfs_fs_write (for userdata) or through an exclusive transaction (for metadata).

Once passed the synchronization loop for BFS_IM_SNAP_IN_PROGRESS, advfs_getpage will start a transaction to contain the entire COW effort. The entire COW process will happen under transactional control so that a system crash does not leave any snapshots with a data reuse case that could render the

snapshot corrupt. For a write fault on metadata, the app_parent_ftx field must contain a non-Nil transaction handle to be the parent of the COW transaction. The only exception to this rule is for metadata that is not COWed (reserved metadata and tag directories.) This is required to correctly synchronize with snapset creation. In the event that some parts of the COW operation fail, an attempt will be made to not fail the entire transaction. Instead of failing the entire COW operation, snapshots will be marked as out of sync in their bfAccess structure and in the flags field of their tag directory entry. The fault will be allowed to proceed without COWing. In the case of multiple snapshots, as many snapshots will be COWed as is possible.

After having started a transaction, advfs_getpage will see if either the faulted-on bfap, or any of its children snapshots need a copy of the parent's metadata . Otherwise, if the BFA_VIRGIN_SNAP flag is set, then advfs_setup_cow will be called to make a copy of the parent's metadata for the faulted-on file. If either the BFA_SNAP_CHANGE flag is set or the bfSet has a child snapset and the bfap->bfaFirstSnapChild field is NULL, then a call will be made to advfs_access_snap_children. advfs_access_snap_children will verify that all the snapshots children of the bfap being faulted on are open and will open and ref any children that are not already open.

When advfs_getpage is servicing a write fault on a file that may need COW processing the need for COW operations will be checked for in ADVFS_COW_ALLOC_UNITS*bfap->bfPageSz byte units. To accomplish this, the fault offset will be rounded down to an ADVFS_COW_ALLOC_UNITS*bfap->bfPageSz boundary and the offset+size will be rounded up to an ADVFS_COW_ALLOC_UNITS*bfap->bfPageSz boundary (not exceeding file_size). These ranges will be used to check whether COWing is required. This will help reduce fragmentation in snapshots.

The first step in COW processing is to acquire all necessary locks. advfs_acquire_snap_locks will handle acquiring locks in the correct order for any potential COW or extent map operations. In either a read or write case, the migStg_lk must be acquired for each ancestor of the faulted on bfap. In the case of a write, the bfaSnapLock must be acquired for write access for each child, and the migStg_lk must be acquired for read for each child. The migStg_lk will protect reads from parent snapshots from having the storage migrated during the read operation.

In the write case, advfs_get_blkmap_in_range will be called to determine if any storage is required for this write. If storage is required, the file lock will be acquired in write mode and advfs_getpage will either start over or continue. In either case, once the file lock is held for write and advfs_acquire_snap_locks has been called, advfs_add_snap_stg will be called to allocate any storage or COWed holes required to process this fault. On successful return from advfs_add_snap_stg, any unmapped regions in the children snapshots that will require storage will either have storage or will be a COWed hole.

advfs_add_snap_stg will return an extent_blk_desc for each range of storage added for a child snapshot. Any snapshot children that have no storage added will have their locks dropped. The extent_blk_desc returned will be organized in a two-dimensional list by file then extent. The extent_blk_desc for a single child will be linked via the ebd_next_desc pointer while the list of each child snapshot's extents will be linked through the ebd_snap_fwd pointer. advfs_add_snap_stg will also return a min and max fob that was added to children. If snap_maps is NULL, these values are undefined. Otherwise, the write request will be brought back in to the min and max fob offset. If min_fob is not equal to the request, then either a hole existed in the original or no storage was added. In either case, we do not need to extend the fault in this case since a COWED_HOLE was already dealt with and already COWed storage does not need to be dealt with. The same logic applies for max fob.

Any extent_blk_descs that represent storage in a child snapshot have a pointer to the bfap for which the migStg_lk and the bfaSnapLock must be dropped after all IOs are completed[9]. For user data, storage for the faulted on bfap will be added in the normal advfs_getpage path for userdata. For metadata, advfs_sync_cow_metadata will be called to force a synchronous COW so that no special handling is necessary in advfs_getpage or advfs_getmetapage.

[9] If no COW was required for a given snapshot child, the locks were released by advfs_add_snap_stg. If any COWing was required, then the snap map has a extent map for the snapshot child.

If the write is for metadata, advfs_getmetapage will be called. The call to advfs_getmetapage will pass in the snap maps through a new snap_map parameter. advfs_getmetapage may start multiple IOs and will pass back an IO Anchor that represents all IOs that were started. For metadata, COWing will be started in advfs_getmetapage and finished in the IO waiting section of advfs_getpage.

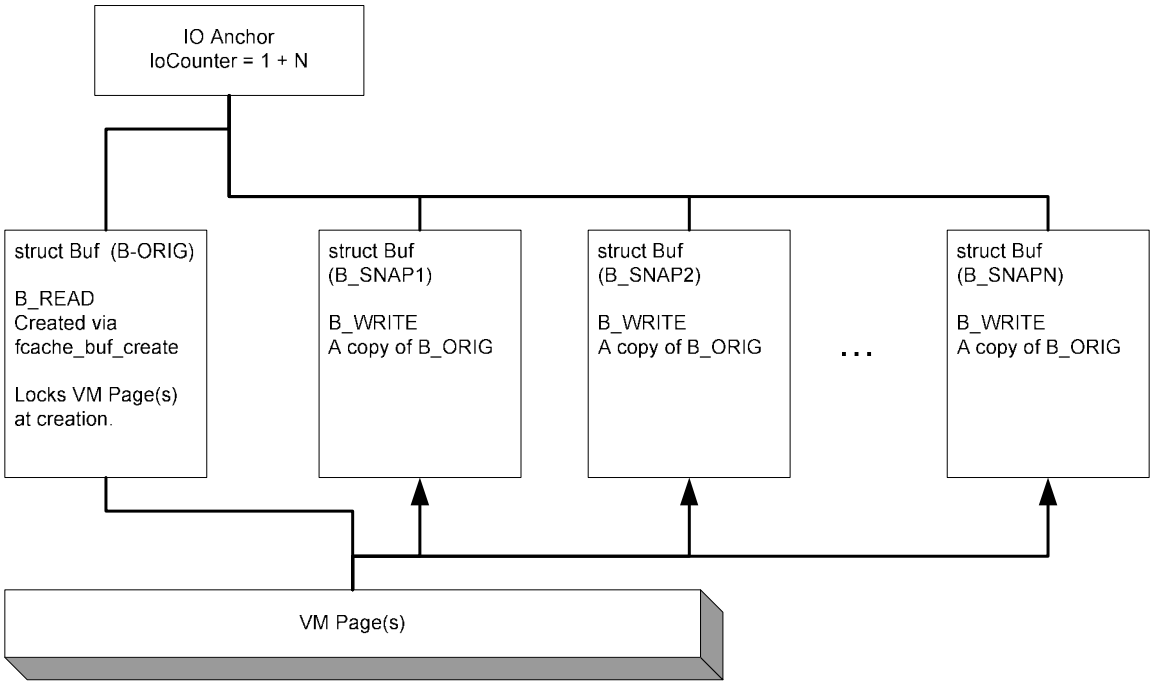
At the start of the fcache_page_alloc loop, where x_load_inmem_xtnt_map is called, a call will be made to advfs_acquire_xtntMap_locks. This routine will call x_load_inmem_xtnt_map on the chain of parents of the faulted on file and acquire the xtntMap_lks for read. This is necessary for both reads and writes in case the faulted-on bfap has unmapped regions that come from the parent snapshot.

The READ case of the fcache_page_alloc loop will change very little; however, it is necessary to make sure any pages that have advfs_start_blkmap_io called on them are set to be write protected on IO completion if any snapshots exist. The test for a snapshot existing will be based on the bfSet that the faulted on file belongs to since a read to a snapshot will not cause child snapshots to be opened. The protecting of these pages is necessary since read operations are allowed to proceed through advfs_getpage while a snapset create operation is occurring.

The WRITE case of the fache_page_alloc loop will change more significantly to initiate IO to snapshots. When dealing with snapshots, large pages will always be demoted if any COWing needs to be done. In the WRITE case code path, a check will be made to see if the snapshot unmapped maps are NULL. If the list is NULL, then no COWing is required to the children, so large pages can be processed as usual, otherwise, any pages outside of the range to be faulted are released and advfs_getpage will skip past all of the large page processing.

In the case of WRITE's in which the pages from fcache_page_alloc are found in cache, if the page is already writeable, then the page must have been COWed previously and can just be released. Otherwise, if the page is still read only, then it must be COWed. If the page is read only, then it may be either dirty or clean. Dirty pages must be pushed out to all snapshot children and the file being faulted on. Clean pages must be pushed out to all child snapshots but not the bfap being faulted on.

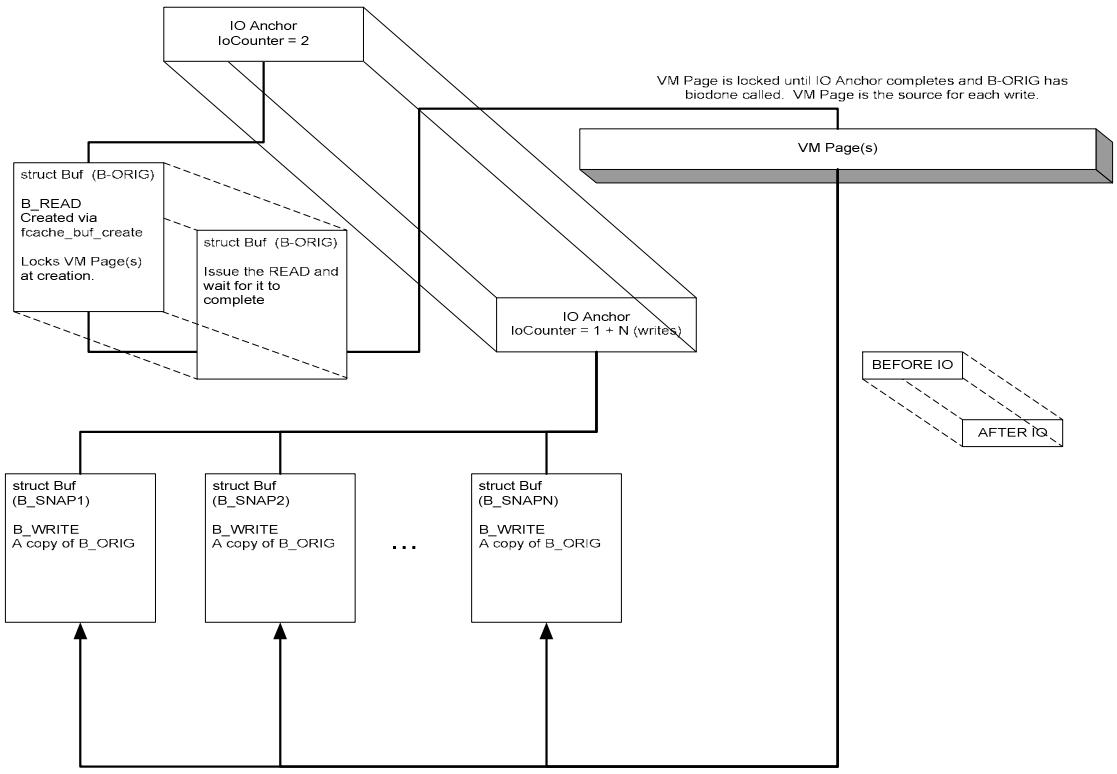
If the page is in cache and read only (and COWing must be done) then, for each contiguous range of the plist returned from fcache_page_alloc, fcache_buf_create will be called for a read. This read will be a fake read and will just be used to synchronize with an IO Anchor that will track all necessary writes. An IO Anchor will be created for each fake read with an IO count of 1. The IO Anchor will be set to not be freed at IO completion time. The IO Anchor will have the IOANCHORFLG_CHAIN_ERROR_BUFS flag set to keep track of all errant disk IOs. If the page is dirty, a copy of the fake read buf will be created for a write. The fake buf will have the disk block for the write set in it along with the write flag. The IO count of the IO anchor will be bumped by one and the buf will be issued as an IO. Next, a call will be made to advfs_issue_snap_io passing the snap maps and the IO Anchor. advfs_issue_snap_io will issue an IO to any extents in snapshots that overlap the IO that the buf structure represents. Once all writes to the current bfap and the children snapshots have been started, the fake read will be issued as an IO. The fake IO has already bumped the IO Count. The IO Anchor will be added to the list of IO Anchors on which to wait at the end of advfs_getpage. The picture below illustrates the relationship between buf structures and IO Anchors in the "in-cache" case.



VM Page is locked until IO Anchor completes and B-ORIG has biodone called. VM Page is the source for each write.

In the case that the plist returned from fcache_page_alloc is not found in cache, both a read from disk and a write will be required to complete the COW. Since these are serial operations (the write cannot be started until the read completes), the IO Anchor will be used to provide a serial notification of a completed READ without having processed the IO completion. The goal is to wait until the READ completes, then issue the WRITEs before the READ pages are unlocked.[10] If the pages are unlocked, then the COW process is compromised since another thread could modify the data before the write occurs. The picture below illustrates the case of a page (or list of pages) not found in cache. The IO Anchor is described before and after the READ IO completes. The write IOs will not be issued until after the READ IO has completed (but the VM page will not be unlocked until all writes have completed).

[10] The reads from disk may be coming from the parent of the faulted on bfap if the faulted on bfap is a child snapshot.



When calling advfs_start_blkmap_io, a new flag (ADVIOFLG_SNAP_READ) will be passed in to indicate that the IO is for the READ portion of a READ/WRITE COW pair. Advfs_start_blkmap_io will create an IO anchor(s) for the IO that will have the IO Count set to 2 and the IOANCHORFLG_WAKEUP_ON_ALL_IO set. Additionally, the IOANCHORFLG_CHAIN_ERRORS will be set so that all bufs that have errors generated will be accessible.

In the objectsafety case, if a transaction must be started, it will be started as a subtransaction of the COW transaction. When the objectsafety transaction is completed, it will be completed with a special done mode so that it is not undone if the overall transaction fails.

When calling advfs_start_blkmap_io for read ahead on a fileset that has any snapshot children, or is itself a snapshot child, a special flag will be passed to indicate that any pages brought into cache must be marked read-only.

The final step in the COW processing in advfs_getpage is to wait for any remaining WRITE or READs from disk, and for each READ that completes, to issue any necessary WRITEs to snapshot children or the current bfap. All started IOs are associated with an IO Anchor and the READ IO anchors are at the front of the list, so those will be processed first and their corresponding WRITEs issued before advfs_getpage begins to wait on any synchronous WRITEs.

If the caller specified FP_ASYNC, advfs_getpage can only return without waiting for IO if no COWing was done. Otherwise, if the snap maps are non-NULL, for each IO Anchor, the anchor will be locked. If the anchor has the IOANCHORFLG_WAKEUP_ON_ALL_IOs flag set and the IO count is 1, then the read has already completed. If the read has already completed advfs_issue_snap_io will be called to kick of any corresponding WRITEs. Before the IOs are initiated, the IOANCHORFLG_WAKEUP_ON_ALL_IO will be cleared since advfs_getpage will only be concerned on waiting for the last IO to complete. Once all writes have been issued, the IO count will be decremented by 1 to remove the extra IO count put on when the read was issued. Each write that is issued will need to make a copy of the buf structure used to issue the read. This buf structure will be acquired from a field in the IO Anchor and will represent a copy of the read buf structure before the READ IO was issued. A copy

of the read buf structure before the IO was issued is required (as opposed to making a copy after the IO is issued) because the low level drivers may modify the buf structure during IOs. It is necessary to have a buf that can still have IO issued on it and not one that looks as if IO has already been issued.

If the IOANCHORFLG_WAKEUP_ON_ALL_IO is not set and the IOANCHORFLG_IODONE flag is set, then all IOs have been completed on this anchor and we can remove it from the list. If the IO count is non-zero, and IOANCHORFLG_WAKEUP_ON_ALL_IO is not set, then advfs_getpage will wait on the IO Anchor's condition variable and wakeup without relocking. The IO Anchor will be reprocessed once the condition variable signal is received.

In the event of an IO error for any of the writes, the snapshot to which the IO was directed will have advfs_snap_out_of_sync called to mark the snapshot on its fileset as out of sync.

Once all IOs have completed, the COW transaction will be ftx_done'd.

Before returning the migStg_lk of all parents and the file lock and migStg_lk of the child snapshots (along with the migStg_lk of the faulted on bfap if necessary) will be dropped.

For direct IO, advfs_getpage may be called directly with the APP_ADDSTG_NOCACHE set in the private parameters structure. Since the pages that storage is added to will be forced out of cache anyways, it is safe to allow the storage to be added. If storage is added, any cache writers will need the file lock and direct IO will upgrade to write mode, so there will not be a race. Trying to block out direct IO threads during snapset creation would required dropping the file lock to prevent a dead lock. In the event that the APP_ADDSTG_NOCACHE is set and the upgrade of the file lock from read to write fails, then after the file lock is reacquired for write, if a snapshot may exist, advfs_getpage will do a fault on the range that may required a COW. This case will be very rare and will represent a direct IO write operation to a sparse range in a file that was racing with a snapset creation. The fault is necessary since the snapset creation may have succeeded while the direct IO thread was blocked on the file lock. The fault will occur with the APP_ADDSTG_NOCACHE flag which will cause all pages brought into cache by the fault to be invalidated before returning.

When the APP_ADDSTG_NOCACHE flag is set, and snapshots may exist, the full advfs_getpage path will be followed (not the short cut for just adding storage). Additionally, if any pages are faulted in when the APP_ADDSTG_NOCACHE flag is set, they will be invalidated before returning from advfs_getpage.

advfs_getpage will be modified to support an APP_FORCE_COW flag. When this flag is set, no new storage will be allocated for the file being faulted on, but the range will be COWed. Any pages that are backed by a hole will not be brought into cache. If the APP_FORCE_COW flag is set and the faulted on file is metadata, the user data write path will be followed. This will prevent the metadata from being brought into cache with a bsBuf and a writeRef. The page must be clean since it has not yet been COWed.

3.2.4.3.1.3 Execution Flow

- if an mmapper, get the file lock for read

Start_over:

- if (mmap writer
  - incr bfap->bfaWriteCnt
- if (bfSet BFS_IM_SNAP_IN_PROGRESS) & mmap writer
  - While (BfSet BFS_IM_SNAP_IN_PROGRESS is set)
    - ASSERT bfap is not metadata
    - Decr bfap->bfaWriteCnt
    - Lock bfSet->bfsSnapMutex
    - if BFS_IM_SNAP_IN_PROGRSS
      - cv_wait(bfSet->bfsSnapCv, NO_RELOCK)
    - incr bfap->bfaWriteCnt
- if metadata && pvt_params == NULL
  - /* Do no COW processing. This is an implicit fault for write on metadata, the COWing must have already been done. This is not detailed in the

following execution flow, but all COW processing and transactions will not be done (starting a transaction would likely cause a deadlock). */

- Kick out mmappers that are beyond EOF (goto popsicle stand if EFAULT)
- if (parent or child snapset exists && FCF_DFLT_WRITE)
    - FTX_START(FTA_GETPAGE_COW, cow_ftx, pvt_params->app_parent_ftx)
    - if (bfap->file lock held for write)
        - ASSERT(!bfap->BFA_VIRGIN_SNAP)
            /* If the file lock is held for write, we should already have COWed metadata for bfap. */
    - Else /* In this case, we need to access any child snapshots and possibly get a copy of metadata for bfap */
        - If (bfap->BFA_VIRGIN_SNAP)
            - write lock bfap->bfaSnapLock
            - advfs_cow_setup(bfap)
            - unlock bfap->bfaSnapLock
        - if (bfap->BFA_SNAP_CHANGE || (bfSet->bfsFirstSnapChild && bfap->bfaFirstSnapChild)
            - advfs_access_snap_children(bfap)
- if parent or child snapset exists
    - read lock bfap->bfaSnapLock
    - if bfap->bfaFlags & BFA_OUT_OF_SYNC
        - fail cow_ftx
        - unlock bfap->bfaSnapLocks
        - return EACCESS (is this the right error?)
- /* Deal with CFS if necessary */
- if clu_is_ready and (filelock held for read or not held) and not metadata
    - if child snapset sets
        - if child snapset exists
            - for each child
                - write lock child->bfaSnapLock
                - while bfap->bfaFlags & BFA_IN_COW_MODE
                    - cv_wait bfaSnapCv
                - lock child's bfaLock
                - set BFA_XTNTS_IN_USE flag in bfap
                - set BFA_IN_COW_MODE flag in bfap
                - unlock child's bfaLock
                - unlock child->bfaSnapLock
            - unlock bfap->bfaSnapLock
            - for each child
                - CLU_CFS_COW_MODE_ENTER
                - if CLU_CFS_COW_MODE_ENTER fails
                    - lock bfaLock
                    - clear BFA_IN_COW_MODE flag in bfap
                    - unlock bfaLock
                - else exit_child_cow_mode = TRUE
            - Read lock bfap->bfaSnapLock
    - if parent snapset exists
        - unlock bfaSnapLock
        - write lock bfaSnapLock
        - lock bfaLock
        - set BFA_XTNTS_IN_USE flag in bfap
        - set BFA_IN_COW_MODE flag in bfap

            - unlock bfaLock
            - drop bfap->bfapSnapLock
            - CLU_CFS_COW_MODE_ENTER
            - If CLU_CFS_COW_MODE_ENTER fails
                - Lock bfaLock
                - Clear BFA_IN_COW_MODE flag in bfap
                - Unlock bfaLock
            - Else exit_bfap_cow_mode = TRUE
            - Read lock bfap->bfaSnapLock
- Else if clu_is_ready and file lock held for write or bfap is metadata
    - If child snapsets exists
        - For each child
            - ASSERT child->bfaFlags BFA_CFS_HAS_XTNTS is not set
            - ASSERT child->bfaFlags BFA_XTNTS_IN_USE flag is set
    - If parent snapset exists
        - ASSERT bfap->bfaFlags BFA_CFS_HAS_XTNTS is not set
        - ASSERT bfap->bfaFlags BFA_XTNTS_IN_USE flag is set
- Adjust sizes (including rounding down to ADVFS_COW_ALLOC_UNIT boundaries)
- Do read-ahead processing
- /* Acquire necessary locks for COWing */
- if parent or child snapsets exist
    - advfs_acquire_snap_locks (FCF_DFLT_WRITE ? SF_SNAP_WRITE : SF_SNAP_READ)
- If FCF_DFLT_WRITE
    - advfs_get_blkmap_in_range (XTNT_NO_MAPS & LOCK_NOT_ALREADY_HELD)
    - if storage required
        - if (!metadata) && file lock not held for write
            - Try to upgrade to write
                - ftx_done the cow_ftx
                - If we failed, drop read fileLock, get write Lock,
                - advfs_drop_snap_locks FCF_DFLT_WRITE ? SF_SNAP_WRITE : SF_SNAP_READ
                - drop bfap->bfaSnapLock
                - goto start_over
    - if child snapsets exists
        - advfs_add_snap_stg (bfap, snap_maps, offset, size, min_fob, max_fob)
        - if (min_fob > offset)
            - Set offset to the minimum of the initial request size or min_fob.
        - if (max_fob < offset + size)
            - Set end of write to max of max_fob or initial request end.
    - /* Setup request_needs_storage flags */
- if metadata & FCF_DFLT_WRITE & !APP_FORCE_COW
    - If bfap is snapshot child
        - call advfs_sync_cow_metapg(bfap, offset, size, ftx)
    - if parent or child snapsets exist
        - advfs_acquire_xtntMap_lks
    - else x_load_inmem_xtnt_map
    - call advfs_getmetapage(bfap, snap_maps)
    - Goto popsicle stand
- /* Calculate loop control variables */
- if snap_maps == NULL

        - no_cow_required = TRUE
        - re-adjust fault range so ADVFS_COW_ALLOC_UNIT rounding is undone
- else
    - pflags &= ~FP_CREATE
- bfap_smap_storage_head = NULL
- bfap_smap_storage_tail = NULL
- while remaining_loop_size > 0
    - if parent or child snapsets exist
        - advfs_acquire_xtntMap_locks
    - else x_load_inmem_xtnt_map
    - fcache_page_alloc
    - if FCF_DFLT_READ
        - if FP_ST_DATAFILL
            - advfs_get_blkmaps_in_range
            - for each extent
                - if extent is a hole
                    - No change in logic
                - Else
                    - If child snapshots exist
                        - For each pfdat in extent range
                            - Set pi_pg_ro flag
                        - advfs_start_blkmap_io
        - else FP_ST_EXISTS
            - No change in logic
    - Else FCF_DFLT_WRITE
        - if no_cow_required
            - Process large pages
        - Else
            - /* Demote any large pages outside of COW adjusted fault range on the left. */
            - if alloc_offset < cow_adjusted_fault_offset
                - if alloc_status == FPA_ST_DATAFILL
                    - inv_flags = pflags&FP_INVAL(~FP_DEFAULT)
                - fcache_page_release(current_loop_offset - alloc_offset
        - if FP_ST_EXISTS
            - if no_cow_required
                - /* Non-snapshot case, logic unchanged */
                - fcache_page_release
            - else
                - while plist != NULL and plist->pfs_offset < request_end
                    - pfdat = plist
                    - is_read_only = pfdat->pi_pg_ro
                    - is_dirty = pfdat->pi_pg_dirty
                    - pfdat = pfdat->pfs_next
                    - contiguous_range = VM_PAGE_SZ
                    - while pfdat != NULL and pfdat->pi_pg_ro == is_read_only and pfdat->pi_pg_dirty == is_dirty
                        - pfdat = pfdat->pfs_next
                        - contiguous_range += VM_PAGE_SZ

                - if !is_read_only
                    - fcache_page_release
                - else
                    - /* May need to do a COW */
                    - ASSERT !metadata && dirty
                    - ioanchor = advfs_get_io_anchor
                    - ioanchor->anchr_iocounter = 1
                    - ioanchor->anchr_flags &= IOANCHOR_KEEP_ANCHOR|IOANCHOR_WAKEUP_ON_ALL_IO|IOANCHOR_CHAIN_ERRORS
                    - fcache_buf_create a buf passing in plist and a size of contiguous_range for a READ
                    - ioanchor->anchr_orig_buf = buf
                    - malloc a temp_buf buf structure
                    - bcopy buf to temp_buf
                    - ioanchor->anchr_buf_copy = temp_buf
                    - ASSERT the plist is not dirty
                    - advfs_issue_snap_io passing the ioanchor and the snap_maps
                    - issue the fake read by calling advfs_bs_startio with the ADVIOFLG_FAKEIO flag
        - else FP_ST_DATAFILL
            - advfs_get_blkmap_in_range to get extent maps
            - for each extent
                - if extent is not a hole
                    - if !FP_CREATE
                        - When calling advfs_start_blkmap_io, pass ADVIOFLG_SNAP_READ.
                        - /* Remaining logic remains the same */
                    - else
                        - No change to logic for the FP_CREATE case
                - Else
                    - /* The extent is a hole */
                    - if APP_FORCE_COW
                        - invalidate pages
                    - No change to logic for dealing with holes
- if parent or child snapset exists and plist != NULL
    - /* Deal with any large pages to the right */
    - if alloc_status == FP_ST_EXISTS
        - fcache_page_release(pflags)
    - else fcache_page_release (pflags &FP_INVAL)
- if parent or child snapset exists
    - advfs_unlock_xtntMap_locks
- else unlock bfap->xtntMap_lk
- if request_needs_storage and !APP_FORCE_COW
    - advfs_get_blkmap_in_range
    - for each sparseness map
        - advfs_bs_add_stg
        - if advfs_bs_add_stg fails

- for each child snapshot
  - advfs_snap_out_of_sync
- if BFS_OD_OBJ_SAFETY
  - advfs_bs_zero_fill_pages
  - if !no_cow_required
    - ftx_special_done_mode to skip undo
  - ftx_done_n
- else if FP_CREATE
  - no changes to FP_CREATE logic
- if parent snapset exists and extent is unmapped (not a COWed hole)
  - advfs_get_blkmap_in_range for sparseness map to get storage_extents (Get extent map for storage just added)
  - if bfap_smap_storage_head == NULL
    - bfap_smap_storage_head = storage_extents
  - else bfap_smap_storage_tail->ebd_next_desc = storage_extents
  - if bfap_smap_storage_tail == NULL
    - bfap_smap_storage_tail = bfap_smap_storage_head
  - while bfap_smap_storage_tail->ebd_next_desc != NULL
    - bfap_smap_storage_tail = bfap_smap_storage_tail->ebd_next_desc

- if need_storage & !BFS_OD_OBJ_SAFETY
  - if FP_CREATE
    - No change to logic for FP_CREATE
  - If !FP_CREATE || alloc_status == FPA_ST_EXISTS
    - if bfap_smap_storage_head == NULL
      - advfs_unprotect_range all allocated storage
    - else
      - Call advfs_unprotect range on portions of allocated storage not in bfap_smap_storage list. (The page locks are still held for the pages in the bfap_smap_storage_head since the IO has not completed).
  - Else
    - No change to logic for the FP_CREATE case

- if bfap_smap_storage_head != NULL
  - /* If we allocated storage for a child snapshot, add it's storage to the snap maps so that writes go out to the new storage */
  - bfap_smap_storage_head->ebd_snap_fwd = snap_maps
  - snap_maps = bfap_smap_storage_head
- if child snapset exists
  - advfs_kickoff_readahead with flag to mark pages as READ ONLY
- else advfs_kickoff_readahead

popsicle_stand:

- if release_pages
  - fcache_page_release
- if xtntmap_locked
  - unlock xtntMap_lk
- if unlock_filelock
  - unlock file lock
- if ioAnchor_head != NULL

- while (ioAnchor_head != NULL)
  - if snap_maps == NULL
    - Current io wait logic
  - else
    - lock ioanchor
    - if ioanchor->anchr_flags & IOANCHORFLG_WAKEUP_ON_ALL_IOs
      - /* This was a read, must issue writes */
      - if ioanchor->anchr_iocounter == 1
        - /* The read has completed */
        - clear IOANCHORFLG_WAKEUP_ON_ALL_IOS
        - unlock ioanchor
        - if ioanchor->anchr_error_ios != NULL
          - for each child in snap maps that would have required a write
            - advfs_snap_out_of_sync
          - finish the io associated with the ioanchor (biodone) to free the VM page.
          - free the adviodesc
          - free the ioanchor
          - continue
        - call advfs_issue_snap_ios to issue writes
        - lock ioanchor
        - ioanchor->anchr_iocounter—
        - put ioanchor at end of ioAnchor list
        - unlock ioanchor
      - else
        - /* The read isn't done yet, wait */
        - cv_wait ioanchor->anchr_cvwait NO_RELOCK
    - else
      - /* Waiting for writes */
      - if ioanchor->anchr_flags & IOANCHORFLG_IODONE
        - ioAnchor_head = ioAnchor_head->anchr_listfwd
        - ASSERT ioanchor->anchr_iocounter == 0
        - Unlock ioanchor
        - if ioanchor->anchr_error_ios != NULL
          - for each adviodesc
            - if advio_bfaccess == bfap
              - if bfap is a snapshot, set_mark_bfap_out_of_sync = TRUE
              - For each child of bfap, call advfs_snap_out_of_sync to mark it out of sync.
              - Execute error logic already in advfs_getpage (adjust return size).
            - Else
              - Call advfs_snap_out_of_sync to mark advio_bfaccess out of sync.

                                        - Free the adviodesc
                                - advfs_bs_free_ioanchor
                        - else
                                - /* Writes not done yet */
                                - cv_wait ioanchor->anchr_cvwait NO_RELOCK
- for each bfap in the snap_maps
    - if bfap == faulted-on bfap
        - continue
    - unlock file's migStg_lk
    - unlock file's bfaSnap_lk
- if exit_child_cow_mode
    - for each child snapshot
        - if BFA_IN_COW_MODE
            - lock bfaLock child
            - clear BFA_XTNTS_IN_USE
            - unlock bfaLock
- if exit_bfap_cow_mode
    - if BFA_IN_COW_MODE
        - lock bfaLock
        - clear BFA_XTNTS_IN_USE
        - unlock bfaLock
- unlock bfap->bfaSnapLock
- if parent snapsets exist
    - for each parent snapshot
        - unlock migStg_lk
- if mark_bfap_out_of_sync == TRUE
    - write lock bfap->bfaSnapLock
    - advfs_snap_out_of_sync
    - unlock bfap->bfaSnapLock
- if exit_child_cow_mode
    - for each child snapshot[11]
        - if child's bfSet is BFS_DELETING, skip the file
        - else
            - CLU_CFS_COW_MODE_LEAVE
            - Lock child->bfaLock
            - Clear BFA_IN_COW_MODE
            - Unlock child->bfaLock
        - broadcast bfaSnapCv
- If exit_bfap_cow_mode
    - CLU_CFS_COW_MODE_LEAVE
    - Lock bfaLock
    - Clear BFA_IN_COW_MODE
    - Unlock bfaLock
    - Broadcast bfaSnapCv
- if mmap writer && BFS_IM_SNAP_IN_PROGRESS
    - decr bfap->bfaWriteCnt

[11] It is safe to walk the child list without holding the bfaSnapLock since the file will be skipped if the state of its fileset is BFS_DELETING. The only way the file could be removed from the list would be if its fileset were deleted.

        - if bfap->bfaWriteCnt == 0
            - broadcast on bfaSnapCv
- free the snap_maps (free each list of extents per file)
- return sts

3.2.4.3.2 advfs_getmetapage

3.2.4.3.2.1 Interface

```
statusT
advfs_getmetapage(
    fcache_vminfo_t *fc_vminfo,      /* An opaque pointer to a vm data struct*/
    struct vnode *vp,                /* The access structure pointer */
    off_t off,                       /* The offset in the file of the fault */
    struct advfs_pvt_param *fs_priv_param,  /* File system private parameter */
    ioanchor_t **ioAnchor_head,       /* list of anchors for wait on */
    fcache_pflags_t pflags            /* Flags passed to VOP_GETPAGE */
    extent_blk_desc* snap_maps)       /* Block maps for snaps needing COWing */
```

3.2.4.3.2.2 Description

advfs_getmetapage will be responsible for setting up the IO Anchors to correctly process metadata. Metadata does not have the same synchronization issues that user data have with respect to snapset creation since metadata must be modified under a transaction and snapset creation will be done under an exclusive transaction. It is assumed that even for writeable metadata snapshots, the child has already had storage allocated and initialized before this routine is called.

The new parameters snap_maps will indicate whether or not any storage was allocated to children snapshots that need to have data written to it. The snap_maps will also include storage for the faulted on file if that file were a metadata file that is having a COW occur from the parent to the child.

In the case of advfs_getmetapage where the page or pages are found in cache, if snap_maps is NULL, the pages are released. If the snap_maps are non-NULL, then IOs must be issued to any overlapping ranges in the snap map. An assertion can be made that any pages found in cache and needing to be pushed out to disk are not dirty. This is because the pages should have been flushed and protected under an exclusive transaction. To issue the writes, a fake read will be setup and an IO anchor will be created with an IO count of 1. A call to advfs_issue_snap_io will be made to issue the writes to overlapping regions in the snap_maps. On return, the fake IO will be issued and the IO Anchor will be chained to the end of the returned ioAnchor list.

After having issued an IO for each snapshot child, the fake IO will be issued and the IO Anchor will be linked to the end of the IO Anchor list to be returned to advfs_getpage.

In the case of FP_ST_DATA_FILL, if any COWing is required, then it is necessary that the data first be read into cache. This will be done in a similar manner to advfs_getpage. The READ operation will be started in advfs_getmetapage and the corresponding WRITE operations will be issued at the end of advfs_getpage once advfs_getmetapage returns. If snap_maps is non-NULL, then advfs_start_blkmap_io will be passed ADVIOFLG_SNAP_READ to indicate that it needs to set the IO Count on the IO Anchor to 2, set the WAKEUP_ON_ALL_IO flag in the IO anchor, and that it needs to link the anchor to the front rather than the back of the IO Anchor list.

The IO Anchor(s) returned by advfs_getmetapage will be further processed or waited on by advfs_getpage.

3.2.4.3.2.3 Execution Flow

- while loop_size > 0
    - fcache_page_alloc
    - if loop_size == ADVFS_METADTA_PGSZ
        - no change to logic

- if snap_maps
  - ASSERT bfap->bfaSnapLock locked
  - ASSERT plist->pi_pg_dirty == 0
- if plist->pi_pg_dirty == 0
  - no change to logic
- if alloc_status == FPA_ST_EXISTS
  - if snap_maps == NULL
    - fcache_page_release
  - else
    - /* May need to do a COW */
    - ioanchor = advfs_get_io_anchor
    - ioanchor->anchr_iocounter = 1
    - ioanchor->anchr_flags &= IOANCHOR_KEEP_ANCHOR|IOANCHOR_WAKEUP_ON_ALL_IO
    - fcache_buf_create a buf passing in plist and a size of contiguous_range for a READ
    - ioanchor->anchr_orig_buf = buf
    - malloc a temp_buf buf structure
    - bcopy buf to temp_buf
    - ioanchor->anchr_buf_copy = temp_buf
    - advfs_issue_snap_io passing the ioanchor and the snap_maps
    - issue the fake read by calling advfs_bs_startio with the ADVIOFLG_FAKEIO flag
- else
  - /* FPA_ST_DATAFILL */
  - advfs_get_blkmap_in_range
  - if error
    - no change to error logic
  - call advfs_start_blkmap_io, if snap_maps is not NULL, pass ADVIO_FLG_SNAP_READ to put an IO count of 2 on the ioanchor and to set the IOANCHRFLG_WAKEUP_ON_ALL_IO flag.
  - if error
    - no change to current error logic

- return sts

3.2.4.3.3 rbf_add_stg

3.2.4.3.3.1 Interface

```
statusT
rbf_add_stg(
            bfAccessT *bfap,                /* in */
            bf_fob_t fob_offset,            /* in */
            bf_fob_t fob_cnt,               /* in */
            ftxHT parentFtx,                /* in */
            int checkmigstglock             /* in */
            )
```

3.2.4.3.3.2 Description

rbf_add_stg must perform copy-before-write operations whenever a hole of a metadata file is being filled and that file has a child snapshot. In the event that storage is being added to a metadata file and a snapshot exists, an assertion can be made that the parent transaction handle passed into rbf_add_stg is not FtxNilFtxH. If the parent transaction were FtxNilFtxH, then the modification of the metadata would not be correctly synchronized with the advfs_create_snapset's exclusive transaction.

If the file to which storage is added may have a child snapshot and the file is metadata, the children snapshots will be opened via a call to advfs_access_snap_children.

To add holes to children snapshots, rbf_add_stg will acquire the sparseness map for the file to which storage is being added. To acquire the sparseness maps, advfs_get_blkmap_in_range will be called with the round_type RND_ENTIRE_HOLE and the extent_blk_map_type EXB_ONLY_HOLES. Each hole returned will be inserted into all children snapshots. The hole that is being written to may be mapped in either the bfap having storage added or the parent snapshot. If a race occurs and the snapshot has storage added before the COWed hole is added, then the storage wins and the hole will only be created where no storage exists.

The hole will be inserted into the snapshot's extent maps via a call to advfs_make_cow_hole. Prior to calling advfs_make_cow_hole, the migStg_lk will be acquired in READ mode.

3.2.4.3.3.3 Execution Flow

- ASSERT not COWable metadata & bfSet is BFS_SNAP_IN_PROGRESS
- If COWable metadata (not a tag dir and not a reserved file)
    - ASSERT parentFtx != FtxNilFtx
    - if & bfap->bfaFlags & BFA_SNAP_CHANGE
        - advfs_access_snap_children(bfap, parentFtx)
    - advfs_get_blkmap_in_range for bfap using RND_ENTIRE_HOLE and EXB_ONLY_HOLES
    - for each child snapshot
        - acquire child's bfaSnapLock for write
        - acquire migStg_lk of child for read access
        - for each hole extent
            - advfs_make_cow_hole in child snapshot
        - drop child's bfaSnapLock
        - drop migStg_lk of child
- stg_add_stg on bfap
- return status

3.2.4.3.4 advfs_access_snap_children

3.2.4.3.4.1 Interface

```
statusT
advfs_access_snap_children(
                      bfAccessT*     bfap,
                      ftxHT          parent_ftx )
```

3.2.4.3.4.2 Description

advfs_access_snap_children will be called from advfs_getpage or rbf_add_stg before any COWing is done to the children bfaps. The routine will open any children snapshots that are not already opened. Additionally, each child will have a single reference put on it. The accessing of children snapshots is postponed until a COW is required so as to reduce the number of access structures in cache.

The basic algorithm for this routine is to walk the set of child bfSets of the bfSet that bfap belongs to, and for each one, make sure a child bfap exists in bfap's child list. If the child does not exist in the list, the child snapshot will be opened via bs_access_one and the BFA_OPENED_BY_PARENT flag will be set. If the child does not exist in the child snapset, it will be skipped, if an error occurs, the child bfap will have the BS_TD_OUT_OF_SYNC_SNAP set in its tag dir flags field. If any other error occurs, the routine will return the error (most likely EIO). Any snapshot that is marked as out of sync will also cause the BFS_OD_OUT_OF_SYNC flag to be set in the fileset that the snapshot belongs to.

After a child is accessed, if the BFA_SNAP_VIRGIN flag is set, then the bfaSnapLock of the child snapshot will be acquired in write mode and advfs_setup_cow will be called to make a copy of the metadata. If advfs_setup_cow fails to copy the metadata, it will mark the child as BS_TD_OUT_OF_SYNC_SNAP in the tag directory of its bfSet and will set the BFA_OUT_OF_SYNC flag in the bfaFlags.

If advfs_access_snap_children is able to open all child snapshots that are not ENO_SUCH_TAG, then it will return EOK. A return status of ENO_SUCH_TAG may indicate a file that was deleted in a child snapset, or a child in a snapset that is in state BFS_DELETING.

This routine will acquire the bfaSnapLock of bfap while walking and modifying the list of children.

3.2.4.3.4.3 Execution Flow

- ASSERT bfSet is not BFS_IM_SNAP_IN_PROGRESS
- ASSERT bfap->bfaFlags BFA_OPENING_PARENTS is not set
- write lock bfaSnapLock for bfap
- if BFA_SNAP_CHANGE not set in bfaFlags
    - unlock bfaSnapLock
    - return
- cur_bf_set = bfap->bfSet->bfsFirstSnapChild
- cur_bfap = bfap->bfaFirstSnapChild
- prev_bfap = NULL
- while cur_bf_set != NULL
    - if cur_bfap->bfSet == cur_bf_set
        - cur_bf_set = cur_bf_set->bfsNextSnapSibling
        - cur_bfap = cur_bfap->bfaNextSnapSibling
    - else
        - ASSERT cur_bfap->bfaNextSnapSibing == NULL
        - /* This assumes that bfaps are chained in the
          * same order as the bfSets */
        - if cur_bf_set is BFS_DELETING
            - /* If we are deleting the fileset, skip opening the file */
            - cur_bf_set = cur_bf_set->bfsNextSnapSibling
        - else
            - /* Open the child for cur_bf_set */
            - bs_access_one bfap->tag in cur_bf_set to get cur_bfap
            - if bs_access_one with ENO_SUCH_TAG
                - cur_bf_set = cur_bf_set->bfsNextSnapSibling
            - else
                - call advfs_snap_out_of_sync cur_bfap, cur_bf_set
                - On error, domain has paniced.
            - if cur_bfap->bfaFlags & BFA_ROOT_SNAPSHOT
                - bs_close_one cur_bfap
                - cur_bf_set = cur_bf_set->bfsNextSnapSibling
            - Lock cur_bfap->bfaLock
            - Set BFA_OPENED_BY_PARENT flag in bfaFlags
            - Unlock cur_bfap->bfaLock
            - if prev_bfap
                - ASSERT bfap->bfaFirstSnapChild == NULL
                - bfap->bfaFirstSnapChild = cur_bfap
            - else
                - ASSERT bfap->bfaNextSnapSibling = NULL

        - bfap->bfaNextSnapSibling = cur_bfap
    - if cur_bfap->bfaFlags & BFA_SNAP_VIRGIN
        - write lock cur_bfap->bfaSnapLock
        - call advfs_cow_setup on cur_bfap
        - unlock cur_bfap->bfaSnapLock
        - if advfs_cow_setup fails
            - call advfs_snap_out_of_sync cur_bfap, cur_bf_set
            - lock cur_bfap->bfaLock
            - set BFA_OUT_OF_SYNC in bfaFlags
            - unlock cur_bfap->bfaLock
    - ASSERT cur_bfap->bfa_orig_file_size != ADVFS_ROOT_SNAPSHOT
    - prev_bfap = cur_bfap
- lock bfap->bfaLock
- clear BFA_SNAP_CHANGE flag
- unlock bfap->bfaLock
- unlock bfap->bfaSnapLock
- return EOK

3.2.4.3.5 advfs_acquire_snap_locks

3.2.4.3.5.1 Interface

```
statusT
advfs_acquire_snap_locks(
                      bfAccessT*     bfap,
                      snap_flags_t   snap_flags)
```

3.2.4.3.5.2 Description

This routine will acquire the bfaSnapLock and migStg_lk for the snapshots in the snapshot tree as required. This routine will acquire different locks depending on whether the snap_flags indicate a read or a write operation (SF_SNAP_READ or SF_SNAP_WRITE). If bfap is a reserved metadata file or a tag directory file, no locks need to be acquired for reserved files since those files are not COWed. As a result, if the file represents a tag file (is in a fileset with bfSetId of negative two) or is a reserved file (has a negative tag), no work needs to be done by this routine.

For a read of userdata or non-reserved metadata, the migStg_lk will be acquired in read mode for all parent snapshots of bfap[12]. The bfaSnapLock will be held for read on entrance to this routine. For userdata, the file lock will be held for read or write on entrance to this routine.

For a write, the bfaSnapLock for bfap is already held if bfap has a parent snapshot. It is necessary to acquire the bfaSnapLock for each child snapshot for write mode. The locking will proceed in order of the bfaNextSnapSibling chain starting at the bfaFirstSnapChild pointer in bfap. The bfaSnapLock will protect uninitialized data in the snapshots from being read by other threads.

For both reads and write, once the bfaSnapLocks are acquired, it is necessary to acquire the migStg_lk for all parents in read mode. The acquisition of the migStg_lk in read mode will protect against migrate moving the storage during a read. This is necessary in case any reads are required from the parent. If the migStg_lk were not held, then the reads would need to be waited on synchronously while holding the

[12] The migStg_lk will protect against migrate moving the physical storage. It will not protect against the removal of the storage from the parent files. As a result, it is necessary for storage removal to force a COW or transfer of the extents to the child. In any case, it will be necessary to acquire the file lock of the child, so we only need to synchronize with migrate moving the storage.

xtntMap_lks of the parents. For writes, after acquiring the migStg_lk of each parent, the migStg_lk must be acquired for each child. The migStg_lk will synchronize with any migrations of the child snapshot data.

The migStg_lk of the faulted-on file does not need to be acquired since it synchronizes with migrate via the page locks. Once the migStg_lk is acquired for the children, the bfaSnapLock will be acquired in read mode for each of the child snapshots.

Both the bfaSnapLock and the migStg_lk will be dropped at the end of advfs_getpage or as soon as it is determined that a snapshot does not require any COW operations.

Any snapshots marked as BFA_OUT_OF_SYNC will be skipped (no locking will be done).

3.2.4.3.5.3 Execution Flow

- If rsvd metadata or tag file
  - Return EOK
- bfsetT* parent_sets[ADVFS_MAX_SNAP_DEPTH]
- bzero parent_sets
- ASSERT bfaSnapLock of bfap held for read
- cur_parent = bfap->bfSet->bfsParentSnapSet
- high_parent_idx = 0
- while cur_parent != NULL
  - parent_sets[high_parent_idx] = cur_parent
  - high_parent_idx++
  - cur_parent = cur_parent->bfsParentSnapSet
- high_parent_idx--
- /* parent sets now has parent chain from root to bfap's fileset. All locks will be acquired going from high_parent_idx down to 0 */
- if snap_flags & SF_SNAP_WRITE
  - /* Acquire bfaSnapLocks. The lock for bfap
    * is already held (acquired by caller) for read */
  - For each child of bfap
    - If child is BFA_OUT_OF_SYNC continue
    - Lock bfaSnapLock for write
- For high_parent_idx downto zero
  - ASSERT parent is not BFA_OUT_OF_SYNC
  - Read lock parent_sets[i]->migStg_lk
- If snap_flags & SF_SNAP_WRITE
  - For each child
    - If child is BFA_OUT_OF_SYNC continue
    - Read lock migStg_lk
- return EOK

3.2.4.3.6 advfs_drop_snap_locks

3.2.4.3.6.1 Interface

```
statusT
advfs_drop_snap_locks(
                      bfAccessT*     bfap,
                      snap_flags_t   snap_flags)
```

3.2.4.3.6.2 Description

This routine will drop the migStg_lk of all parents of bfap along with the migStg_lk and the bfaSnapLock of each child of bfap. It is assumed that the bfaSnapLock of bfap is held on entrance. The routine cannot be used unless all children (not marked BFA_OUT_OF_SYNC) and all parents' locks are held.

3.2.4.3.6.3 Execution Flow

- ASSERT bfap->bfaSnapLock is held
- cur_parent = bfap->bfaParentSnap
- while cur_parent
    - unlock cur_parent->migStg_lk
    - cur_parent = cur_parent->bfaParentSnap
- if snap_flags & SF_SNAP_READ
    - return EOK
- cur_child = bfap->bfaFirstSnapChild
- while cur_child
    - if cur_child->bfaFlags & BFA_OUT_OF_SYNC
        - continue
    - else
        - ASSERT migStg_lk (read) and bfaSnapLock (write) are held
    - unlock cur_child->migStg_lk
    - unlock cur_child->bfaSnapLock
    - cur_child = cur_child->bfaNextSnapSibling
- return EOK

3.2.4.3.7 advfs_acquire_xtntMap_locks

3.2.4.3.7.1 Interface

```
statusT
advfs_acquire_xtntMap_locks(
                      bfAccessT*     bfap)
```

3.2.4.3.7.2 Description

This routine will acquire the xtntMap_lks for bfap and all of the parents of bfap. The xtntMap_lk will be acquired in read mode. The locks will be acquired from the root down to bfap so that the locking order always proceeds from parent to child. It is assumed that the bfaSnapLock of bfap is held for read access when this routine is called.

3.2.4.3.7.3 Execution Flow

- bfsetT* parent_sets[ADVFS_MAX_SNAP_DEPTH]
- bzero parent_sets
- ASSERT bfaSnapLock of bfap held for read
- cur_parent = bfap->bfSet->bfsParentSnapSet
- high_parent_idx = 0
- while cur_parent != NULL
    - parent_sets[high_parent_idx] = cur_parent
    - high_parent_idx++
    - cur_parent = cur_parent->bfsParentSnapSet
- high_parent_idx--
- /* parent sets now has parent chain from root to bfap's fileset. All locks will be acquired going from high_parent_idx down to 0 */
- for i=high_parent; I > 0; i—

        - x_load_inmem_xtnt_map(parent_sets[i], X_LOAD_REFERNCE)
        - On error
            - Unlock locked parents
            - Return error
- x_load_inmem_xtnt_map(bfap, X_LOAD_REFERNCE)
- On error
    - Unlock locked parents
    - Return error
- return EOK

3.2.4.3.8 advfs_drop_xtntMap_locks

3.2.4.3.8.1 Interface

```
statusT
advfs_drop_xtntMap_locks(
                      bfAccessT*     bfap)
```

3.2.4.3.8.2 Description

This routine will drop the xtntMap_lk of each of the parents of bfap and for bfap itself.

3.2.4.3.8.3 Execution Flow

- Unlock bfap->xtntMap_lk
- Cur_parent = bfap->bfaParentSnap
- While cur_parent != NULL
    - Unlock cur_parent->xtntMap_lk
    - Cur_parent = cur_parent->bfaParentSnap
- return EOK

3.2.4.3.9 advfs_add_snap_stg

3.2.4.3.9.1 Interface

```
statusT
advfs_add_snap_stg(
                      bfAccessT*            bfap,
                      off_t                 offset,
                      size_t                size,
                      extent_blk_desc**     snap_maps,
                      bf_fob_t*             min_storage,
                      bf_fob_t*             max_storage,
                      ftxHT                 parent_ftx )
```

3.2.4.3.9.2 Description

This routine will handle adding storage or COWed holes to each snapshot child of bfap. The work performed by this routine will all be done under the parent_ftx transaction. The extent map of bfap will be examined to determine if a COWed hole or storage should be allocated in the child snapshots. For each storage extent in bfap, each child snapshot will have storage added to it and an extent_blk_desc will be created for the newly allocated storage. No storage will be added to a snapshot child beyond the snapshots bfa_orig_file_size[13]. For each hole in bfap, a COWed hole will be inserted into each child snapshot extent

[13] A writeable snapshot may extend its file_size, but the bfa_orig_file_size is the size of the file at the time the snapshot was taken (or the time the original metadata was created).

map. If storage or a COWed hole already exists in the child, no action is taken (the range has already been COWed).

For each child snapshot, a chain of extent_blk_desc for the storage added will be returned. The extent_blk_desc list for a given snapshot will be chained together using the ebd_next_desc while the lists for separate files will be chained together through the ebd_snap_fwd field of the extent_blk_desc structure.

If storage allocation fails for any child snapshot because of a lack of disk space, that child snapshot will be marked out of sync in the tag directory of its bfSet and in the access structure for the snapshot. If an error other than ENO_SPACE occurs, that error will be returned. If any attempt to add storage to a snapshot child fails, advfs_snap_out_of_sync will be called to mark the snapshot child as out of sync and the locks associated with that file will be dropped. On error, the snapshot will not have any snap maps returned.

This routine assumes that the bfaSnapLock is held for read for bfap and for write for each child of bfap. The routine also assumes that the migStg_lk is held for read on each child snapshot. If any snapshot is found to not require any storage (no COWing required) the bfaSnapLock and migStg_lk will be dropped for that child snapshot.

This routine will return in min_storage and max_storage the lowest and highest fob added to snapshot children. If snap_maps are NULL then these values are undefined. This will be used by the caller to determine if it is possible to un-round the COW value. If the range passed in starts or ends in a hole, the original file does not need to have a fault occur over those fobs since the storage will have been COWed as a hole.

On an error other than ENO_SPACE, the file locks and migStg_lks of the child snapshots will be dropped and the routine will propagate the error.

3.2.4.3.9.3 Execution Flow

- advfs_get_blkmaps_in_range of bfap requesting RND_ENTIRE_HOLE and EXB_COMPLETE and XTNT_LOCKS_HELD
- snap_map_head = snap_map_tail = NULL
- for each child of bfap
    - ASSERT the child's bfaSnapLock is held for write
    - child_extent_head = child_extent_tail = NULL
    - advfs_get_blkmap_in_range on child using RND_NONE and EXB_ONLY_HOLES|EXB_DO_NOT_INHERIT (get unmapped maps and COWed holes) and XTNT_LOCKS_HELD. The request will be over the range of the request, but not beyond the child's bfa_orig_file_size.
    - for each extent of bfap
        - for each extent in child that is unmapped
            - if range is a hole in parent
                - advfs_make_cow_hole on overlapping range (rounded up or down to cover the entire hole in parent)
                - if advfs_make_cow_hole fails
                    - advfs_snap_out_of_sync child_bfap and child_bfap's bfSet
                    - free child_extent_head extent list
                    - break
            - else
                - rbf_add_stg to child bfap
                - if rbf_add_stg fails
                    - advfs_snap_out_of_sync child_bfap and child_bfap's bfSet
                    - free child_extent_head extent list
                    - break
                - create extent_blk_desc over range of added storage
                - if child_extent_head = NULL

                - child_extent_head = extent_blk_desc
            - if child_extent_tail != NULL
                - child_extent_tail->ebd_next_desc = extent_blk_desc
            - child_extent_tail = extent_blk_desc
    - free child extent maps (not child_extent_head list)
    - if child_extent_head != NULL
        - if snap_map_head = NULL
            - snap_map_head = child_extent_head
        - snap_map_tail != NULL
            - snap_map_tail->ebd_snap_fwd = child_extent_head
        - snap_map_tail = child_extent_head
    - else
        - /* No storage was added, so the locks will be dropped */
        - drop bfaSnapLock of child bfap
        - drop migStg_lk of child bfap
- free bfap's extent maps
- *snap_maps = snap_map_head
- return EOK

3.2.4.3.10 advfs_issue_snap_io

3.2.4.3.10.1 Interface

```
statusT
advfs_issue_snap_io(
                        ioanchor_t*             io_anchor,
                        extent_blk_desc**       snap_maps)
```

3.2.4.3.10.2 Description

This routine will examine the anchr_buf_copy field of the io_anchor to determine what range of the snapshot a read was issued to. Once the range is determined, the snap_maps will be examined and WRITE IOs will be issued to each contiguous range that overlaps the range of the anchr_buf_copy buf. In any given snapshot child, multiple IOs may be issued. If the anchr_buf_copy has no overlapping ranges in the snap_maps, no IOs will be issued.

It is assumed that external locking is protecting the snap_maps and the storage they map to from being migrated or removed.

This routine should never be called on reserved metadata.

3.2.4.3.10.3 Execution Flow

- ASSERT bfap is not reserved metadata to tag directory.
- cur_buf = io_anchor->anchr_buf_copy
- ASSERT cur_buf != NULL
- io_start = cur_buf->foffset
- io_end = io_start + cur_buf->size
- cur_extent_maps = snap_maps
- prev_extent_maps = NULL
- while cur_extent_maps != NULL
    - cur_extent = cur_extent_map
    - cur_io_start = io_start
    - while cur_extent != NULL

- if io_end < cur_extent->ebd_offset
  - break
- if io_start > cur_extent->ebd_offset + ebd_size
  - cur_extent = cur_extent->ebd_next_desc
  - continue
- ASSERT io_start >= cur_extent->ebd_offset
- offset_into_extent = cur_extent->ebd_offset – io_start
- blks_into_extent = offset_in_extent / (ADVFS_FOBS_PER_BLK * ADVFS_FOB_SZ)
- if io_end > cur_extent->ebd_offset + ebd_size
  - cur_write_end = cur_extent->ebd_offset + ebd_size
- else cur_write_end = io_end
- malloc temp_buf
- bcopy cur_buf into temp_buf
- temp_buf->b_flags &= ~B_READ
- temp_buf->b_flags &= B_WRITE|B_PHYS|B_CALL
- temp_buf->b_foffset = io_start
- temp_buf->b_blkno = cur_extent->ebd_vd_blk + blks_into_extent
- temp_buf->b_un.b_addr += cur_io_start – io_start
- temp_buf->b_bcount = cur_write_end – cur_io_start
- Lock ioanchor
- Increment iocounter
- Unlock ioanchor
- call advfs_bs_startio with VD_HTOP(cur_extent->ebd_vd_index, cur_extent->bfap->dmnP)->devVp->v_rdev and NULL bsBuf passing in temp_buf for IO.
- cur_extent = cur_extent->ebd_next_desc

o cur_extent_maps = cur_extent_maps->ebd_snap_fwd

- return EOK

3.2.4.3.11 advfs_setup_cow

3.2.4.3.11.1 Interface

```
statusT
advfs_setup_cow(
               bfAccessT*     parent_bfap,
               bfAccessT*     child_bfap,
               snap_flags_t   snap_flags,
               ftxHT          parent_ftx     )
```

3.2.4.3.11.2 Description

This routine will handle making a copy of a parent snapshots metadata for the child. It is assumed that the child snapshot has its bfaSnapLock held for write. Furthermore, it is assumed that the BFA_SNAP_VIRGIN flag is set in the bfAccess structure of the file to receive a copy of the metadata (the child_bfap).

If the copy of metadata fails, advfs_snap_out_of_sync will be called to mark the snapshot and its fileset as out of sync. If the copy succeeds, the BS_TD_VIRGIN_SNAP flag will be cleared in the tag directory.

A transaction handle will be passed in under which to perform the copy of metadata. The basic routine structure, locking and transaction control will be the same as clone in Tru64. The biggest exception to this is that the bfaSnapLock will now provide synchronization. The transaction type FTA_BS_BFS_CLONE_V1 will become FTA_META_SNAP, the new_clone_mcell routine will become advfs_new_snap_mcell, bmtr_clone_recs will become bmtr_snap_recs, stripe processing will be removed,

and field names will be changed to correctly reflect the new snapshot field names in the bfAccess and bfSet structures.

After the chain of mcells has been copied to the child snapshot, a the bsBfAttr field bfat_orig_file_size will be set in the new metadata for the child. The field will be set to the file_size of the parent snapshot at the time the metadata is copied. This field will provide an upper boundary on the amount of data to COW and will be used to initialize bfa_orig_file_size when the snapshot is opened.

Before returning success, the BFA_VIRGIN_SNAP flag will be cleared.

3.2.4.3.11.3 Execution Flow

- If ! child_bfap & BFA_SNAP_VIRGIN
    - /* child_bfap already has its own metadata */
    - return
- write lock child_bfap->bfaSnapLock
- if !child_bfap & BFA_SNAP_VIRGIN
    - unlock bfaSnapLock and return
- /* Start a transaction so that we can finish it with a special done mode to prevent undos */
- /* Copy the parent's mcell list to the child */
- set child_bfap state to ACC_INIT_TRANS to prevent new accesses
- FTX_START_N(FTA_META_SNAP, parent_ftx, snap_ftx)
- get bfAttr (BSR_ATTR) record for parent_bfap
- Initialize as in clone(.) on Tru64
- Read lock parent_bfap->mcellList_lk
- FTX_LOCKWRITE(child_bfap->mcellList_lk, snap_ftx)
- Call advfs_new_snap_mcell (new_clone_mcell on Tru64) to get a primary mcell for child_bfap
- If advfs_new_snap_mcell fails
    - fail snap_ftx
    - Call advfs_snap_out_of_sync(child_bfap, parent_bfap, parent_ftx)
    - Return error
- Call tagdir_lookup_full to get tagdir entry for undo
- If tagdir_lookup_full fails
    - Fail snap_ftx
    - Call advfs_snap_out_of_sync
    - Return error
- Call bmtr_snap_recs (bmtr_clone_recs on Tru64)
- If bmtr_snap_recs fails
    - Fail snap_ftx
    - Call advfs_snap_out_of_sync
    - Return error
- Update the tagdir entry with the new primary mcell (see clone(.) on Tru64). The BS_TD_SNAP_VIRGIN flag will be cleared.
- Call bs_map_bf with the BS_REMAP flag to re-init child_bfap with new primary mcell.
- Reset bsBfAttr->bfat_orig_file_size with parent_bfap->file_size stored in it.
- ftx_special_done_mode(snap_ftx, FTXDONE_SKIP_SUBFTX_UNDO)
- ftx_done snap_ftx
- unlock parent_bfap->mcellList_lk
- bfaFlags &= ~ BFA_SNAP_VIRGIN
- Transition child_bfap back to previous state (before ACC_INIT_TRANS)
- Return EOK

3.2.4.3.12 advfs_sync_cow_metapage

3.2.4.3.12.1 Interface

```
statusT
advfs_sync_cow_metapage(
               bfAccessT*     bfap,
               off_t          offset,
               size_t         size,
               ftxHT          parent_ftx     )
```

3.2.4.3.12.2 Description

This routine is called to synchronously allocate and initialize a metadata page prior to calling advfs_getmetapage. This routine will only have any action to do if bfap is a child snapshot and if the page in the range [offset, size] has not already been COWed to bfap. If the page is unmapped in bfap, then storage will be allocated for the page in bfap's extent maps and the page will be read in and written out to bfap before returning from this function.

This routine assumes that the bfaSnapLock of bfap is held for read on entrance. If the page is unmapped in bfap, then the bfaSnapLock will be upgraded to write mode. If the upgrade fails, then the lock will be dropped and acquired in write mode. This routine also assumes that advfs_acquire_snap_locks has already been called to acquire the migStg_lk of the parent's of bfap. This routine may drop and reacquire any locks taken by advfs_acquire_snap_locks.

If the page is unmapped in bfap, the migStg_lk of all parents and bfap will be dropped (along with the migStg_lk of any children) and the lock will be reacquired for write mode. This will protect the parents' storage against changing while storage is added to bfap. Once the migStg_lks are reacquired for the parents and bfap in write mode, the extents will be queried to find out where in the parent extent maps the page is located. The extent map locks will be dropped and storage will be added to bfap. Next, a READ will be issued to the disk location that the page exists at in the parent. Once the read completes, a write will be issued to the newly allocated storage in bfap. On successful completion of the write, all the parent migStg_lks will be downgraded to read and the routine will return.

This routine will be optimized in the future to do more than a single page of COWing at a time.

3.2.4.3.12.3 Execution Flow

- if bfap->bfaParentSnap == NULL
    - return EOK
- ASSERT that bfap->bfaSnapLock is held for read
- advfs_get_blkmap_in_range(bfap, offset, size, EXB_ONLY_HOLES)
- if extents don't have any unmapped holes
    - free extents
    - return EOK
- if bfap->bfaSnapLock is not held for write
    - Try to upgrade bfap->bfaSnapLock
    - If upgrade fails
        - Drop parent's migStg_lk's
        - Drop child's migStg_lk's and bfaSnapLocks
        - Drop read lock
        - Write lock bfaSnapLock
        - advfs_acquire_snap_locks
        - Start over
- If bfaSnapLock was dropped
    - Advfs_get_blkmap_in_range to reacquire block maps

        - If extents don't have any unmapped holes
            - Free extents
            - If bfaSnapLock was upgraded
                - Downgrade to read mode
            - Return EOK
- rbf_add_stg to bfap over range [offset..offset+size]
- Create extent_blk_desc to describe page storage
- Do a loop similar to advfs_getmetapage to fault in range
    - Fcache_page_alloc
        - Demote any large pages
    - Fcache_buf_create
    - Create ioAnchor with IO Count of 2
    - advfs_start_blkmap_io to do READ
    - wait for READ to complete
    - Call advfs_start_snap_io passing extent_blk_desc of new storage
    - Wait for WRITE to complete
    - On any error
        - advfs_snap_out_of_sync(bfap, bfap->bfSet, parent_ftx)
- if bfaSnapLock was upgraded
    - downgrade to read mode
- Return EOK

3.2.4.3.13 advfs_snap_out_of_sync

3.2.4.3.13.1 Interface

```
int
advfs_snap_out_of_sync(
        struct bfAccess* bfap,        /* in - bfap of out of sync snapshot */
        struct bfSet   * bf_set_ptr   /* in - bfSet to mark out of sync */
        ftxHT  parent_ftx             /* in - transaction to update metadata under */ )
```

3.2.4.3.13.2 Description

This routine will mark the snapshot file bfap and the snapshot fileset bf_set_ptr as out of sync.

To mark bfap as out of sync, the BS_TD_OUT_OF_SYNC_SNAP flag will be transactoinally set in the tag flags of the tag directory entry for bfap and the BFA_OUT_OF_SYNC flag will be set in the bfaFlags field of bfap.

To mark the bf_set_ptr as out of sync, the BFS_OD_OUT_OF_SYNC flag will be set in the bfsFlags of bfap's fileset and will be transactionally written to the bitfile set attributes record for the fileset. If an IO error occurs during this routine, the domain will panic.

Transactions started by this routine will be ftx_done'd with a special done mode that will cause the transaction to not be undone.

It is acceptable for one of the two parameters bfap or bf_set_ptr to be NULL. If bfap or bf_set_ptr is NULL, that parameter is not marked out of sync. In this way, it is possible to mark a file but not a snapset, or a snapset but not a file as out of sync.

3.2.4.3.13.3 Execution Flow

- If (bfap & bfap is already out of sync) and (bfSet and bfSet is already out of sync)
    - return EOK
- FTX_START out_of_sync_ftx
- if FTX_START fails, panic

- If bf_set_ptr != NULL and not bf_set_ptr->bfSetFlags & BFS_OD_OUT_OF_SYNC
    - Get bfSetAttr record for bf_set_ptr (bmtr_get_rec_ptr)
    - rbf_pin_record bfSetAttr->flags field
    - bfSetAttr->flags |= BFS_OD_OUT_OF_SYNC
    - lock bfSetMutex
    - bfSetp->bfSetFlags |= BFS_OD_OUT_OF_SYNC
    - unlock bfSetMutex
- if bfap != NULL and not bfap->bfaFlag & BFA_OUT_OF_SYNC
    - tagdir_lookup_full(bf_set_ptr, bfap->tag, &tag_flags)
    - create a new tag entry with tag flag BS_TD_OUT_OF_SYNC set and all other fields copied from tagdir lookup
    - tagdir_stuff_tagmap
    - lock bfap->bfaLock
    - bfap->bfaFlags |= BFA_OUT_OF_SYNC
    - unlock bfap->bfaLock
- ftx_special_done_mode(out_of_sync_ftx, FTXDONE_SKIP_SUBFTX_UNDO)
- ftx_done
- return EOK

3.2.4.3.14 advfs_fs_write

3.2.4.3.14.1 Interface

```
int
advfs_fs_write(
         struct vnode *vp,              /* in - vnode of file to write */
         struct uio *uio,               /* in - structure for uiomove */
         enum uio_rw rw,                /* in - read/write flags */
         int ioflag,                    /* in - flags - append, sync, etc. */
         struct ucred *cred             /* in - credentials of caller */
         )
```

3.2.4.3.14.2 Description

advfs_fs_write must guard against potentially large transactions which could cause a log half full system panic. In the event of a very large write, storage may need to be acquired for a large sparse range in a file. Currently, the amount of storage allocated in a single transaction will be bounded by MAX_ALLOC_FOB_CNT, however, if a file has multiple children snapshots, then storage may need to be allocated for each of the children in addition to the file being written to.

In the write case, a transaction must be started to allocate storage to the children snapshots. If the transaction were to allocate MAX_ALLOC_FOB_CNT fobs for each child plus MAX_ALLOC_FOB_CNT for the file being written, the total amount of storage allocated in a single transaction would exceed MAX_ALLOC_FOB_CNT. Therefore, advfs_fs_write must limit the size of a single call to fcache_as_uiomove to no more than MAX_ALLOC_FOB_CNT / (number of children + 1) if number of child is greater than 1. This will guard against significantly exceeding MAX_ALLOC_FOB_CNT.

3.2.4.3.15 advfs_start_blkmap_io

3.2.4.3.15.1 Interface

```
statusT
advfs_start_blkmap_io(
    fcache_vminfo_t *fc_vminfo,          /* opaque vm pointer for buf creation */
    struct vnode * vp,                   /* The vnode pointer of the file */
    off_t     offset,                    /* starting offset */
```

```
size_t     length,                    /* length to write */
extent_blk_desc_t * primary_blkmap,      /* The blkmap for the main i/o*/
extent_blk_desc_t ** passed_secondary_blkmap,    /* if multiple destinations */
ioanchor_t **ioAnchor_head,      /* List to add any i/o's started to (SYNC only) */
page_fsdata_t  ** plist,             /* list of PFDATS to write */
struct bsBuf ** bsBufList_ptr,       /* list of bsBufs for meta-data i/o */
fcache_pflags_t pflags,              /* flags need for buf creation */
int32_t         io_flags             /* ADVIOFLG_* flags */
```

3.2.4.3.15.2 Description

This routine will be modified to take a flag ADVIOFLG_SNAP_READ which will cause the routine to initialize the ioanchor_t to have an iocounter value of 2. Additionally, when ADVIOFLG_SNAP_READ is set, then the IOANCHORFLG_WAKEUP_ON_ALL_IO and IOANCHORFLG_CHAIN_ERRORS flags will be set in the anchr_flags field of the ioanchor.

3.2.4.3.16 Miscellaneous Changes

On Tru64, bs_cow was called in a number of places to force only the metadata portion of a file to be COWed to its clone. In HPUX, these calls to bs_cow will be replaced with locking the bfaSnapLock for read, calling the advfs_access_snap_children and dropping the bfaSnapLock.

bmtr_put_rec_n_unlk, bmtr_update_rec, and advfs_setacl will be modified to perform special calls to advfs_access_snap children to force a metadata COW.

bmtr_clone_mcell will be renamed to bmtr_snap_mcell.

When doing a truncate of a writeable snapshot file, it is necessary to reduce the orig file size record in the bsBfAttr field bfat_orig_file_size to match the truncated size. If the truncation extends the file, no work needs to be done. Also in the truncate code path, if a snapshot child exists, then for each snapshot child, advfs_force_cow_and_unlink will be called on truncated. The SF_NO_UNLINK flag will be passed to advfs_force_cow_and_unlink to prevent the truncated file from be unlinked from its children.

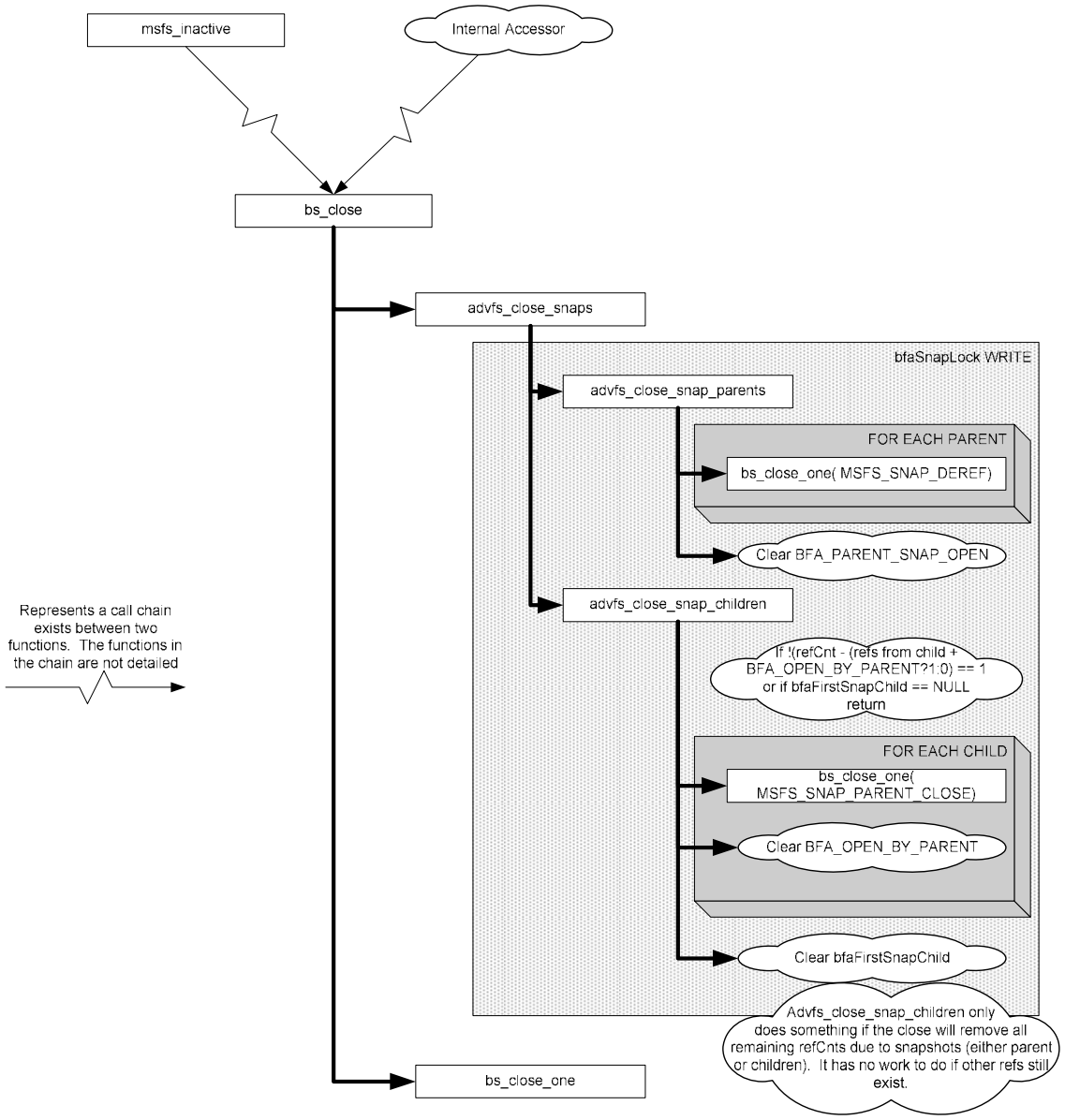
advfs_start_blkmap_io will be modified to handle the ADVIOFLG_SNAP_READ option to create a IO Anchor with an iocounter value of 2 and the IOANCHORFLG_WAKEUP_ON_ALL_IO and IOANCHORFLG_CHAIN_ERROR flags set.

advfs_iodone will be modified so that if IOANCHORFLG_CHAIN_ERRORS is set in the IO Anchor associated with an IO, the adviodesc_t will not be freed but will be chained off of the IO Anchor's anchr_error_ios field and linked through the advio_fwd pointer in the adviodesc_t. If IOANCHORFLG_CHAIN_ERRORS is set, an assertion will be made that the IOANCHORFLG_KEEP_ANCHOR is also set.

advfs_bs_get_ioanchor will be modified to set anchr_error_ios to NULL.

3.2.5 Closing a File

3.2.5.1 Closing a File Overview



3.2.5.2 Basic Operation of Closing a File

In a filesystem with snapshots, the refCnt of a bfAccess structure is incremented and decremented for all of the reasons that it would be incremented in a non-snapshot filesystem. In addition, when snapshots are enabled, it is necessary to put sympathetic refCnts on parent and children snapshots. When closing a file, and when the only remaining references on the file are sympathetic refCnts (either from parent or child snapshots), it is necessary to remove any references on other access structures that the file being closed has acquired. In other words, any time a “last close” would be done in a non-snapshot environment, it is necessary to remove any sympathetic, snapshot related references.

To accomplish this goal, if snapshots are enabled, bs_close will advfs_close_snaps which will in turn call advfs_close_snap_parents and advfs_close_snap_children if needed.

On successful return from advfs_close_snaps, all references placed on parents or children which resulted from opening or writing to the file being closed will be undone. The undoing of the references will synchronize with other accesses of the file via the bfaSnapLock.

Once all sympathetic links are removed from the file being closed, bs_close_one will be called to decrement the refCnt. If no parent or children snapshots have sympathetic reference counts on the file being closed, then it will do last close processing as per the non-snapshot model. Otherwise, last close processing will occur when the last sympathetic refCnt is removed.

3.2.5.3 Function Call Detail

3.2.5.3.1 bs_close

3.2.5.3.1.1 Interface

```
statusT
bs_close(
         bfAccessT            *bfap,         /* in */
         enum acc_close_flags options        /* in */
         )
```

3.2.5.3.1.2 Description

bs_close is the primary routine for causing a close to occur on a file. bs_close will handle closing any related snapshots and will cause the close to occur on the bfap passed in. A quick check to see if snapshots exist will be performed in bs_close before making a decision as to whether advfs_close_snaps needs to be called. If snapshots exist, advfs_close_snaps will close any necessary related snapshots and handle synchronizing with new accesses of the file.

Whether advfs_close_snaps is called or not, bs_close_one will be called before returning to do the actual close of bfAccessp.

3.2.5.3.1.3 Execution Flow

- If bfap == NULL
  - return EINVALID_HANDLE
- if bfap->bfSet has parent or child fileset
  - advfs_close_snaps
- call bs_close_one(bfap, options, FtxNilFtx)

3.2.5.3.2 bs_close_one

3.2.5.3.2.1 Interface

```
statusT
bs_close_one(
         bfAccessT            *bfap,         /* in */
         enum acc_close_flags options        /* in */
         )
```

3.2.5.3.2.2 Description

bs_close_one performs the close of a file. If the refCnt is going from 1 to 0, the close is the last close, whether or not the refCnt is a sympathetic reference from an associated snapshot. bs_close_one is primarily unconcerned as to why the refCnt is being decremented and whether the refCnt is from a "normal" referencer or a snapshot. The only change bs_close_one must make is to take action on the MSFS_SNAP_DEREF and MSFS_SNAP_PARENT_CLOSE. While holding the bfaLock and before calling DEC_REFCNT, if MSFS_SNAP_DEREF is passed in, then the bfaRefsFromChildSnaps field must

also be decremented. If the MSFS_SNAP_PARENT_CLOSE flag is passed in, the bfaFlags BFA_OPENED_BY_PARENT flag must be cleared.

Both of these updates are done in bs_close_one so that they are always consistent with the refCnt for threads that need to determine whether the refCnts are from sympathetic references or "normal" references.

The only other change to bs_close_one will be to remove its support for the MSFS_DO_VRELE option. This option is inefficient on HPUX and callers of bs_close with the MSFS_DO_VRELE flag should change to call VN_RELE directly or do an internal open and close call. Removing the MSFS_DO_VRELE flag removes the need to artificially bump the refCnt in bs_close_one before calling VN_RELE and make it more clear which refCnts are real opens and which are sympathetic references from other snapshots.

3.2.5.3.2.3 Execution Flow

The code will not be modified until the close_it label.

```
close_it:
```

- `/* close_it is always entered with the bfaLock held.  */`
- `release migStg_lk`
- `MSFS_DO_VRELE handling will be removed`
- `If (options & MSFS_SNAP_PARENT_CLOSE)`
  - `Clear BFA_OPENED_BY_PARENT in bfaFlags`
- `Else if (options & MSFS_SNAP_DEREF)`
  - `Decrement bfaRefsFromChildSnaps`
- `Perform DEC_REFCNT`
- `Finish transaction`

3.2.5.3.3 advfs_close_snaps

3.2.5.3.3.1 Interface

```
statusT
advfs_close_snaps(
         bfAccessT            *bfap,  /* bfap to have associated snapshots closed */
)
```

3.2.5.3.3.2 Description

advfs_close_snaps will close any snapshot parents or children that were opened/ref'ed as a result of a call to advfs_access_snap_parents or advfs_access_snap_children. The routine will first check to see if any snapshots exist. If snapshots do not exist on the bfSet of bfap, then the routine will return. Next, advfs_close_snaps will check to see if this is the last close of a normal (non-snapshot related) open. If refCnt – bfaRefsFromChildSnaps == 1 and the BFA_OPENED_BY_PARENT flag is not set, or if refCnt – bfaRefsFromChildSnaps == 2 and the BFA_OPENED_BY_PARENT flag is set, then this is the last close and advfs_close_snaps will continue. Otherwise, it will return success since there is no work to be done.

If snapshots exist, then the bfaSnapLock of bfap will be acquired (in write mode) and bfaParentSnap field will be checked. If NULL, then no work needs to be done to close the parents. If bfaParentSnap is non-NULL, then advfs_close_snap_parents will be called. On returning from advfs_close_snap_parents, all references on parent snapshots will have been removed. BFA_PARENT_SNAP_OPEN will be cleared, but bfaParentSnap will still point to the parent snap on return from advfs_close_snap_parents.

Once advfs_close_parent_snaps has been called, if bfaFirstChildSnap is non-NULL, then advfs_close_snap_children will be called. advfs_close_snap_children will remove the reference counts on any immediate children snapshots that were opened as a result of a write to this file.

Before returning, the bfaSnapLock will be dropped.

3.2.5.3.3.3 Execution Flow

- If bfSet is not a child or parent snapset
    - Return
- write lock the bfap bfaSnapLock
- If bfap->refCnt –
        (bfaRefsFromChildSnaps +
        (bfap->bfaFlags & BFA_OPENED_BY_PARENT ? 1 : 0)) != 1
        /* Not the last non-snap closer, so just return */
    - Unlock bfaSnapLock
    - Return
- If bfap->bfaParentSnap
    - call advfs_close_snap_parents(bfap)
- if bfap->bfaFirstSnapChild
    - call advfs_close_snap_children(bfap)
- unlock bfap bfaSnapLock
- return

3.2.5.3.4 advfs_close_snap_parents

3.2.5.3.4.1 Interface

```
statusT
advfs_close_snap_parents(
         bfAccessT            *bfap,  /* bfap to have parent snapshots closed */
)
```

3.2.5.3.4.2 Description

If the bfaFlag BFA_PARENT_SNAP_OPEN flag is not set, this routine has no work to do and can return. Otherwise, advfs_close_snap_parents will call bs_close_one on each of the parents of bfap (following the bfaParentSnap points in the bfAccess structure) passing in the MSFS_SNAP_DEREF flag to indicate that the bfaRefsFromChildSnaps should be decremented in addition to the refCnt of the parent access structure. It is assumed that the bfaSnapLock of bfap is held in write mode during this call to synchronize with an access trying to access the snap parents.

Before returning, the BFA_PARENT_SNAP_OPEN flag must be cleared so that any future calls to access the file correctly open the parents.

3.2.5.3.4.3 Execution Flow

- ASSERT(bfap->bfaSnapLock held for write)
- If bfap->bfaFlags & BFA_PARENT_SNAP_OPEN not true
    - Return EOK
- Current parent = bfap->bfaParentSnap
- While current parent != NULL
    - Next parent = current parent->bfaParentSnap
    - Bs_close_one(current parent, MSFS_SNAP_DEREF)
    - Current parent = next parent
- Lock bfaLock
- Clear BFA_PARENT_SNAP_OPEN
- Unlock bfaLock
- Return EOK

3.2.5.3.5 advfs_close_snap_children

3.2.5.3.5.1 Interface

```
statusT
advfs_close_snap_children(
        bfAccessT           *bfap,  /* bfap to have parent snapshots closed */
)
```

3.2.5.3.5.2 Description

advfs_close_snap_children must close and dereference any child snapshots that were opened as a result of a write to bfap. If bfap->bfaFirstSnapChild is NULL, then this routine has no work do to and can return.

It is assumed that this routine is called with the bfaSnapLock held for write. Once it is determined that bfaFirstSnapChild is non-NULL, the list of child snapshots will be walked and bs_close_one will be called on each child passing the MSFS_SNAP_PARENT_CLOSE flag to indicate that the bfaFlag BFA_OPENED_BY_PARENT flag must be cleared in the child bfap when the refCnt is decremented.

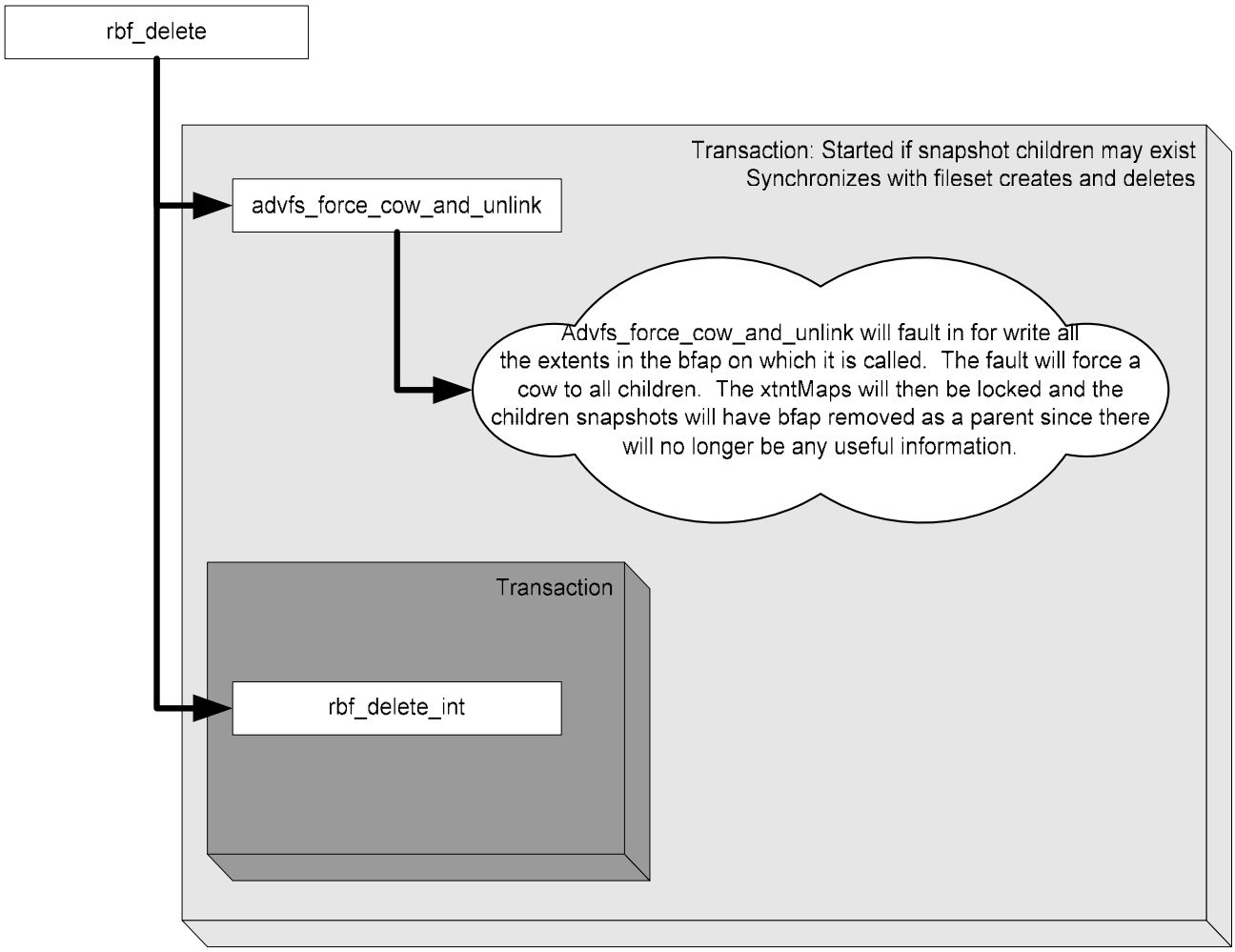
Before returning, the bfaFirstSnapChild pointer must be set to NULL.

3.2.5.3.5.3 Execution Flow

- `ASSERT( bfap->bfaSnapLock held for write)`
- `If bfap->bfaFirstSnapChild == NULL`
    - `return EOK`
- `current child = bfap->bfaFirstSnapChild`
- `while current child != NULL`
    - `next child = current child->bfaNextSnapSibling`
    - `bs_close_one( current child, MSFS_SNAP_PARENT_CLOSE )`
    - `current child = next child`
- `bfaFirstSnapChild = NULL`
- `return EOK`

3.2.6 Deleting a file

3.2.6.1 Deleting a File Overview



3.2.6.2 Basic Operation of Deleting a File

Deleting a file in a parent fileset or in a writeable snapshot will be the same basic operation. Because multiple filesets make transferring extents to snapshot children both more difficult than it already was and less effective, the ability to transfer extents directly to snapshots will not be initially supported. As a result, the process of deleting a file can be simplified to fault in a file's storage at delete time, thereby causing the storage to be created in the children snapshots.

The basic operation for deleting a file will start by calling rbf_delete. If the fileset of the file to be deleted has any children snapshots, then advfs_force_cow_and_unlink will be called. advfs_force_cow_and_unlink will fault in (for write) the entirety of the deleted file's extent maps, thereby causing a complete COW to the children (metadata will also be created for the children if it does not exist).

After faulting in all necessary data, the pages can be invalidated from the cache and the parent file can be unlinked from the child bfaps. To unlink the parent file, the bfaParentSnap in each child will be set to NULL. Additionally, the refCnt in the file being deleted and all its parent snapshots will be decremented by bfaRefsFromChildSnaps to remove the dependency on the parents from the children. On all the parents, the close will be done via multiple calls to bs_close_one with the MSFS_SNAP_DEREF. On the

file being closed, the refCnt will simply be decremented since at least one non-snapshot reference must exist for the thread calling rbf_delete, therefore last close processing will not be required.

Once advfs_force_cow_and_unlink has been called, the file effectively exists as a stand alone file in the snapshot fileset and has no dependency on the parent snapshot. Therefore, rbf_delete_int can be called on the parent. On last close, the parent bfap will be removed as normal and no additional extent or snapshot processing will be required (except closing parent snapshots as appropriate).

3.2.6.3 Function Call Detail

3.2.6.3.1 rbf_delete

3.2.6.3.1.1 Interface

```
statusT
rbf_delete (
        bfAccessT       *bfap,          /* bfap to be marked for deletion */
        ftxHT           parent_ftx      /* Parent FTX */
)
```

3.2.6.3.1.2 Description

rbf_delete is the interface routine for marking a file for deletion. Since a file can be deleted by one thread but still opened by another thread, rbf_delete does not actually remove the file from a fileset, it simply sets the on disk state to BSRA_DELETING which prevents further opens. On last close the file is removed from the fileset.

On Tru64, if a file in an original fileset was being deleted, it would be marked as "delete with clone" to indicate that when the associated clone file was deleted, the parent should also be deleted. In the context of multiple, writeable snapshots, "delete with clone" is a complex idea that would require checking all siblings and children of those siblings to determine if it was safe to delete a file. To simply the logic (which can be expanded at a later date), rbf_delete will simply remove all dependencies that children snapshots may have on any parents when the parent file is deleted. Removing the dependency will include forcing all data in the file to be deleted to be COWed to any children. While the approach is less efficient than the Tru64 model, it is significantly simpler to implement and will reduce the technical risk associated with read only snapshots and would become obsolete with multiple-writeable snapshots.

3.2.6.3.1.3 Execution Flow

- If a snapshot exists
    - ASSERT parent_ftx is FtxNilFtxH
    - Call advfs_force_cow_and_unlink
    - Start a transaction
- call rbf_delete_int
- If a transaction was started, finish the transaction

3.2.6.3.2 advfs_force_cow_and_unlink

3.2.6.3.2.1 Interface

```
statusT
advfs_force_cow_and_unlink (
        bfAccessT       *bfap,          /* bfap to have all data cowed. */
        off_t           offset,         /* Starting offset to COW from */
```

```
        size_t          size,           /* Size to COW. 0 for the entire file */
        snap_flags_t    snap_flags      /* SF_NO_UNLINK if no unlink is desired */
        ftxHT           parent_ftx      /* Parent FTX */
)
```

3.2.6.3.2.2 Description

This routine will force any data in bfap (mapped in bfap or its parent's extents) to be COWed to the children snapshots of bfap. Once all the data is COWed, the dependency between the child snapshots and bfap and all parents of bfap can be severed. To sever the connection, all parents of bfap will be closed a number of times equal to bfap->bfaRefsFromChildSnaps, then bfap itself will have its refCnt decremented by bfaRefsFromChildSnaps. Since the thread performing the close will necessarily have a refCnt on bfap, this routine can safely decrement the refCnt directly (while holding the bfaLock) without calling bs_close.

To force the COW from bfap to its children, advfs_force_cow_and_unlink will get a copy of the extent maps for bfap and fault in each contiguous range for write. On return from the fault, the data in the faulted on range will be invalidated to help reduce the amount of memory consumed for force COWing. If any errors occur during the force COW, all child snapshots will be marked as out of sync in the tag directory.

Once the forced COW has occurred, each parent of bfap will be closed bfap->bfaRefsFromChildSnaps times. After all parents have been closed, the refCnt of bfap will be directly decremented by bfaRefsFromChildSnaps and bfaRefsFromChildSnaps will be set to zero. Because the child snapshots have a complete copy of the parent snapshots, there is no longer any connection with the parents. The bfaParentSnap pointer in each child will be set to NULL.

When calling fcache_as_fault to force the COW, a private flag APP_FORCE_COW will be passed to indicate that no storage should be allocated to the file being deleted. This should simply cause the COW to the children without affecting the parent.

This routine must not be called in the context of a transaction since it may cause a log half full transaction.

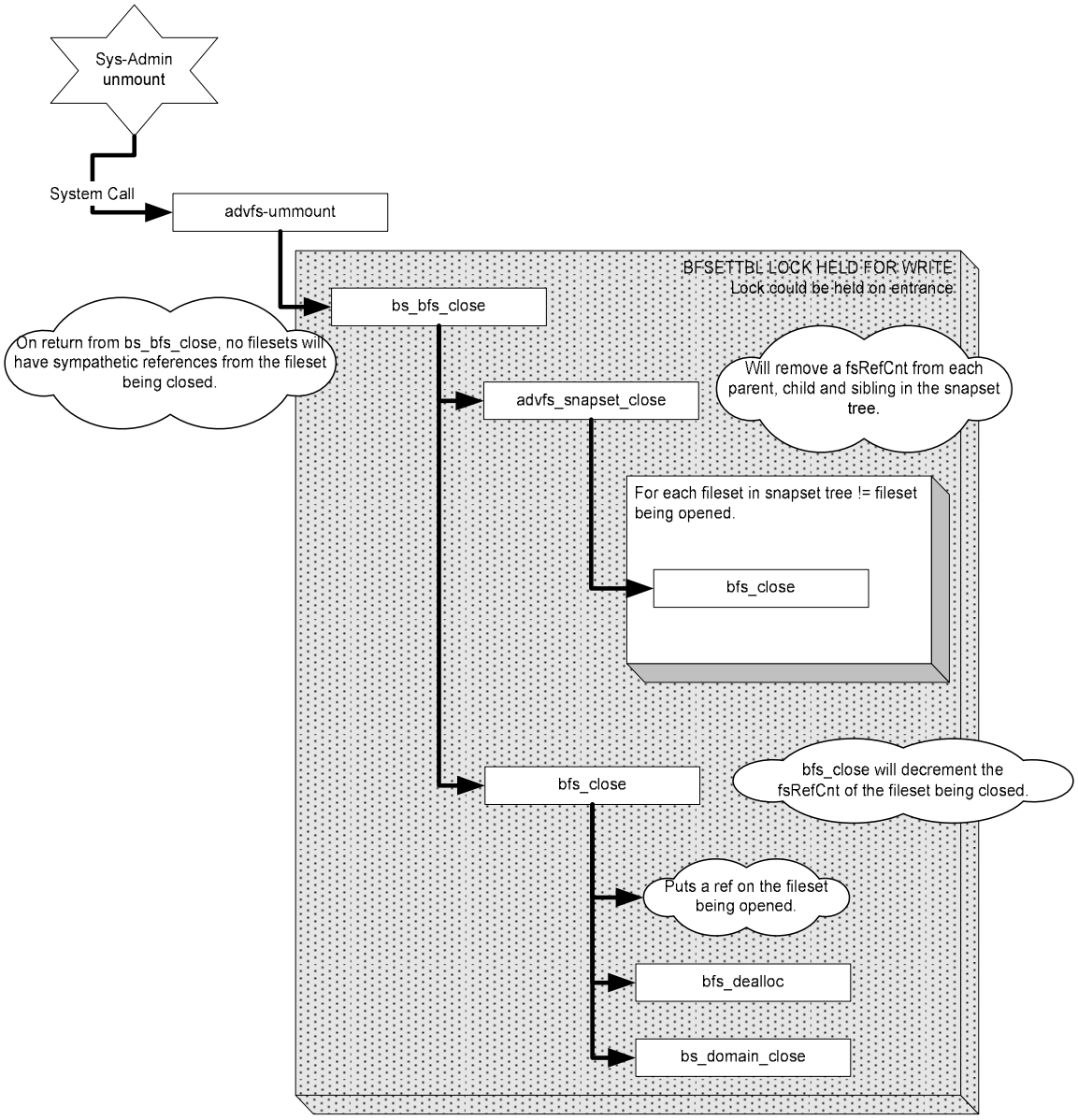
3.2.6.3.2.3 Execution Flow

- if bfap has no children snapshots
    - return
- ASSERT(bfap->refCnt > 1)
- Get extent maps for bfap in range [offset..offset+size] if size is 0, request the entire file (use bfaNextFob for metadata and file_size for userdata).
- While not at the end of extent list
    - Fcache_as_map(min(extent size, ADVFS_FORCE_COW_MAX_ALLOC_UNITS)) (meta-vas or user-vas as appropriate)
    - Sts = Fcache_as_fault(min(extent size, ADVFS_FORCE_COW_MAX_ALLOC_UNITS), FCF_DFLT_WRITE passing in APP_FORCE_COW
    - If an error occurred during fault
        - For each child
            - Transactionally set BS_TD_OUT_OF_SYNC_SNAP in tagdir flags field, and set BFA_OUT_OF_SYNC in child's bfaFlags field.
            - unmap and invalidate range
            - break
    - Fcache_as_unmap
    - fcache_vn_invalidate(faulted range)
    - If extent_size > amount faulted
        - Advance extent start (advance offset value of extent)
    - Else
        - Advance to next extent
- If !SF_NO_UNLINK
    - For each parent of bfap
        - For I = 0; I < bfap->bfaRefsFromChildSnaps; i++

            - bs_close_one(parent, MSFS_SNAP_DEREF)
        - lock bfap->bfaLock
        - bfap->refCnt -= bfap->bfaRefsFromChildSnaps
        - bfap->bfaRefsFromChildSnaps = 0
        - for each child
            - child->bfaParentSnapShot = NULL
        - unlock bfap->bfaLock
- return EOK

3.2.7 Closing a fileset

3.2.7.1 Closing a Fileset Overview



3.2.7.2 Basic Operation of Closing a Fileset

Closing a fileset will be very similar to opening a fileset. bs_bfs_close is the counterpart routine to bfs_access. Any calls to bs_bfs_close will be responsible for removing any references on related snapshots that bfs_access put on those related snapsets. bfs_close operates only on single filesets without respect to related snapsets.

To close a fileset, the bfSetTbl lock will be acquired for write. The lock may be held for write on entrance to bs_bfs_close, or may be acquired for write in bs_bfs_close. In either case, once the lock is acquired, no new filesets in the domain can be created or opened. advfs_snapset_close will remove all the references on filesets that advfs_snapset_access placed. advfs_snapset_close does not need to undo the actions of advfs_link_snapsets since the snapsets may still be opened. If all the snapsets are closed, they will be deallocated and the links will become invalid.

Once all related snapsets have been closed, bfs_close will be called on the fileset being unmounted or otherwise closed. The call to bfs_close may deallocate the fileset if no related filesets still have external fsRefCnts on them (i.e. if no other filesets are mounted to keep the fileset being closed referenced, it will be deallocated).

3.2.7.3 Function Call Detail

3.2.7.3.1 bs_bfs_close

3.2.7.3.1.1 Interface

```
statusT
bs_bfs_close (
        bfSetT          *bf_set_p,      /* bfSet to be closed. */
        ftxHT           parent_ftx      /* Parent FTX */
)
```

3.2.7.3.1.2 Description

bs_bfs_close is responsible for undoing all the actions of bfs_access. Specifically, bs_bfs_close must remove any references put on snapsets related to the one being closed, and it must close the fileset on which it was called.

bs_bfs_close will first remove any references on related snapsets by calling advfs_snapset_close, then it will close bf_set_p by calling bfs_close. bs_bfs_close may be called with the bfSetTbl lock held for write mode, or not held at all. If the lock is not held when this routine is called, then the lock will be acquired in write mode before calling advfs_snapset_close and will be dropped before returning. Holding the bfSetTbl lock in write mode synchronizes with opening other snapsets and with creating new snapsets.

Although bs_bfs_close must undo the actions of bfs_access, it does not need to unlink the filesets. If the fileset being closed represents the last non-sympathetic fsRefCnt on any of the snapshots, then it will cause each snapset to be deallocated in turn as bfs_close is called on that snapset. When bfs_close is called on bfs_set_p it too will be deallocated. As a result, destroying the links between snapsets will not be required since they will be created next time any snapset is accessed in the snapset tree.

3.2.7.3.1.3 Execution Flow

- ASSERT(bf_set_p is a valid fileset)
- ASSERT(bf_set_p->fsRefCnt > 0)
- If bfSetTblLock is not held for write
    - Acquire bfSetTblLock for write
- Call advfs_snapset_close to close all parents, children and siblings
- Call bfs_close to close the bf_set_p
- If bfSetTblLock was acquired
    - Drop bfSetTblLock

3.2.7.3.2 advfs_snapset_close

3.2.7.3.2.1 Interface

```
statusT
advfs_snapset_close (
       bfSetT         *bf_set_p,     /* bfSet to have all related snapsets closed */
       ftxHT          parent_ftx     /* Parent FTX */
)
```

3.2.7.3.2.2 Description

This routine is responsible for calling bfs_close on every related snapset (parent, child or sibling) that advfs_snapset_access called bfs_access on when bf_set_p was opened (via bfs_open). The routine does not need to undo the inter-fileset linking that was done between snapsets in advfs_snapset_access.

Like advfs_snapset_access, this routine relies on a recursive helper routine to walk the snapset tree. The recursive routine will not call bfs_close on the fileset being closed (only on the filesets that it has put a reference on).

The bfSetTblLock must be held for write when calling advfs_snapset_close to synchronize with the creation of new snapsets and the opening of snapsets that are related to bf_set_p.

3.2.7.3.2.3 Execution Flow

- ASSERTbfSetTblLock held for write
- root_set = bf_set_p
- next_parent_set = bf_set_p->bfaParentSnapSet
- While (next_parent_set)
    - root_set = next_parent_set
    - next_parent_set = parent_set->bfaParentSnapSet
- advfs_snapset_close_recursive(bf_set_p, root_set, parent_ftx)
- if advfs_snapset_close_recursive fails,
    - ADVFS_SAD
    - /* Optionally, the filesets could be unlinked in memory and the system could continue, but a memory leak will have occurred and problems could arise in the future. */
- return EOK

3.2.7.3.3 advfs_snapset_close_recursive

3.2.7.3.3.1 Interface

```
statusT
advfs_snapset_close_recursive (
       bfSetT         *bf_set_p,     /* bfSet to have all related snapsets closed */
       bfSetT         *cur_set_p     /* Parent of cur_set_p */
       ftxHT          parent_ftx     /* Parent FTX */
)
```

3.2.7.3.3.2 Description

This routine is a helper routine to advfs_snapset_close. The routine will do a post-order traversal of the entire snapset tree calling bfs_close on each snapset that is not bf_set_p (that will be closed by bs_bfs_close.

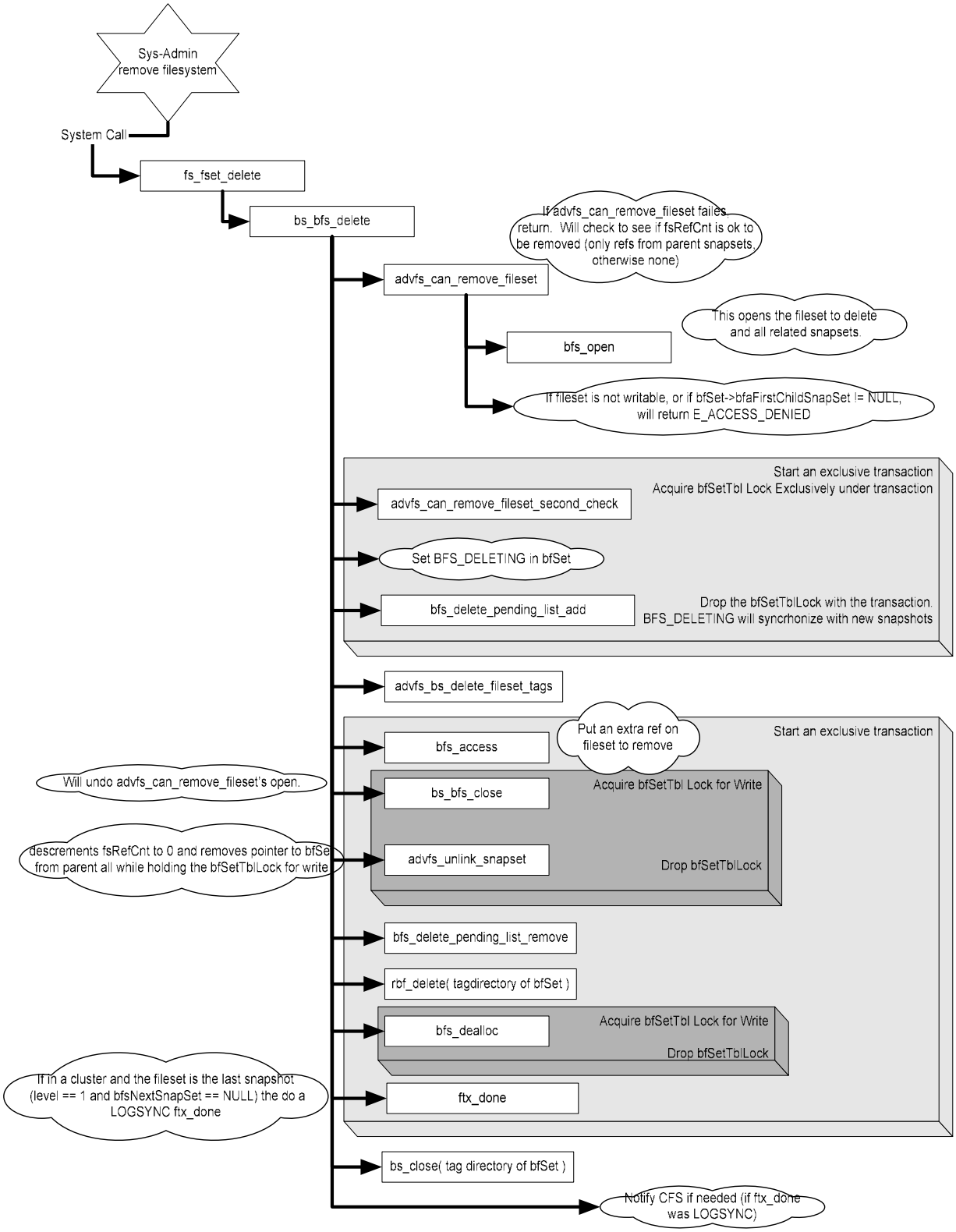
This routine relies on recursion to close any number of snapsets; however, in practice, the number of snapsets ought to be bounded to prevent deep recursive calls on the kernel stack.

3.2.7.3.3.3 Execution Flow

- `ASSERTbfSetTblLock held for write`
- `if cur_set_p->bfaFirstSnapChild != NULL`
    - `for each child`
        - `advfs_snapset_close_recursive( bf_set_p, child_set_p, parent_ftx)`
        - `ASSERT advfs_snapset_close returns EOK`
- `if bf_set_p != cur_set_p`
    - `bfs_close( cur_set_p )`
- `return EOK`

3.2.8 Removing a fileset

3.2.8.1 Removing a Fileset Overview



3.2.8.2 Basic Operation of Removing a Fileset

The process of removing a fileset in a snapshot environment will be simplified so that removing a parent or child snapset is essentially the same process as removing a single fileset with a few extra checks. The process of removing a snapshot child or parent will not be nearly as different as they were on Tru64.

A user thread attempting to remove a fileset will resolve to fs_fset_delete in the kernel. fs_fset_delete will do some minor error checking at the domain level to make sure the calling thread has permissions to remove the fileset. Next, bs_bfs_delete will be called to do the majority of the fileset removal work.

bs_bfs_delete will call advfs_can_remove_fileset to perform racy checks to see if the fileset can be removed, and to open the fileset to be removed and any related snapsets. advfs_can_remove_fileset will deny the removal of the fileset if the calling thread does not have write access to the fileset, or if the fileset has any children filesets. Additionally, advfs_can_remove_fileset will return an error if there are accessors to the fileset that are not from other snapsets in the snapset tree. bfsSnapRefs is used to determine the number of accesses caused by other snapsets.

If advfs_can_remove_fileset returns successfully, then all related snapsets will have been opened and must be closed before returning. Next, an exclusive transaction will be started under which the BFS_DELETING state will be set in the fileset to delete. The exclusive transaction blocks any snapshots from being created while the BFS_DELETING state is set. As soon as the exclusive transaction is started, the fileset to be opened will be checked to see if any snapshot children were created while waiting to start an exclusive transaction. If advfs_can_remove_fileset_second_check finds a child snapshot, an error is returned, all filesets that were opened are closed and the bs_bfs_delete will return an error. Once the exclusive transaction is completed, the BFS_DELETING state will prevent new snapshots from being created. Under the exclusive transaction, the fileset to be removed is added to the fileset delete pending list so that bs_bfs_delete will be called again in the event of a system failure.

Once on the fileset delete pending list and in state BFS_DELETING, the exclusive transaction is completed. The transaction must be stopped so that the (potentially huge number of) files in the fileset can be deleted. The files in the fileset will be removed via a call to advfs_bs_delete_fileset_tags. advfs_bs_delete_fileset_tags will remove any files in the fileset and will unlink any open files from their parents so that no new COWing is done. The unlinking will require the bfaSnapLock.

After advfs_bs_delete_fileset_tags has completed, another exclusive transaction will be started. Under the new exclusive transaction, the fileset to be deleted will have bfs_access called on it so that a second fsRefCnt is put on it. This allows bs_bfs_close to be called, thereby forcing closed[14] all other related snapsets that are open because of the fileset to be removed. Before bs_bfs_close is called, however, the fileset to be removed will be removed from the list of its parent's child snapshots if a parent exists. This will be done while holding the bfSetTbl Lock.

advfs_unlink_snapset will be called to adjust the fsRefCnt of the fileset to be removed to be 0 (removing any references from related snapsets) and to unlink the fileset from its parent and any sibling snapsets. advfs_unlink_snapset will be called while still holding the bfSetTbl lock, but the lock will be dropped on return from advfs_unlink_snapset.

After the fileset has been unlinked, bfs_delete_pending_list_remove will be called to take the fileset back off of the fileset delete pending list. At this point, the fileset is essentially a stand alone fileset whether or not it previously was a snapshot child. The remaining process for deleting the fileset includes removing the tag directory for the fileset, deallocating the fileset and closing the tag directories fileset. If the snapshot being removed is the last snapshot child in a snapset tree and the snapset is in a cluster environment, CFS will be notified via a callout that there are no snapshots remaining.

---

[14] The call to bs_bfs_close will decrement the fsRefCnt on each related snapset. The snapsets will only actually be closed if the fileset to be deleted was keeping the other fileset open.

3.2.8.3 Function Call Detail

3.2.8.3.1 fs_fset_delete

3.2.8.3.1.1 Interface

```
statusT
fs_fset_delete(
    char *domain,       /* in - name of set's domain table */
    char *setName,      /* in - name of set to delete */
    ftxIdT xid          /* in - CFS transaction id */
    )
```

3.2.8.3.1.2 Description

fs_fset_delete is the high level interface for deleting a fileset. The routine will activate the fileset to be deleted and call bs_bfs_delete to perform the bulk of the work associated with deleting the fileset. fs_fset_delete can be called on any fileset, whether or not it has associated snapsets. If the fileset on which fs_fset_delete is called is mounted or if it has any snapshot children, the delete operation will fail and an error will be returned.

3.2.8.3.1.3 Execution Flow

- Call bs_bfset_activate to get bfSetId
- Call bs_bfs_delete passing the bfSetId
- Call bs_bfdmn_deactivate to deactivate the domain (undoing the bs_bfset_activate call above)

3.2.8.3.2 bs_bfs_delete

3.2.8.3.2.1 Interface

```
statusT
bs_bfs_delete(
    bfSetIdT bfSetId,   /* in - bitfile set id */
    domainT *dmnP,      /* in - set's domain pointer */
    ftxIdT  xid         /* in - CFS transaction id */
    )
```

3.2.8.3.2.2 Description

bs_bfs_delete is the primary internal interface for removing a fileset. The routine will do some basic error checking before initiating the removal of the fileset. Before beginning the active removal of the fileset specified by bfSetId, bs_bfs_delete will put the fileset on the fileset delete pending list. As a result of this operation, this routine may be called multiple times on the same fileset so it cannot do any damage to the domain that would prevent it from completing a removal.

To validate that the calling thread has permission to remove the requested fileset, and that the fileset is safe to remove (no external references or child snapsets), advfs_can_remove_fileset is called. If advfs_can_remove_fileset succeeds, it will return the open bfSet structure to be removed. If advfs_can_remove_fileset fails, it will have cleaned up any operations that it performed and bs_bfs_delete will simply propogate the error.

If advfs_can_remove_fileset verifies that the fileset can safely be removed and returns success, an exclusive transaction must be started to put the fileset on the fileset deferred delete list. The exclusive transaction is required to synchronize with the creation and opening of related snapsets (and snapset children). Once the exclusive transaction is started, advfs_can_remove_fileset_second_chance will be called to verify that the fileset is still a valid candidate to be removed. The two level verification approach is used to avoid starting an exclusive transaction if possible, but a check is required after the exclusive transaction starts to synchronize with new snapsets. Within the context of the exclusive transaction, the

fileset state will be set to BFS_DELETING and the fileset will be put onto the fileset delete pending list. When the transaction is finished, new accesses will be allowed on the fileset, and no snapshots will be able to be created on the fileset because of the BFS_DELETING state. The transaction will be ftx_done'd with the LOGSYNC option to make sure the transaction is on disk before the links between parent and child snapshots are broken. Once a files link from its parent is severed, any remaining writes to the parent will not be COWed. It is necessary to have the transaction that puts the fileset on the fileset delete pending list on disk so that the potentially out of sync fileset is definitely deleted prior to the next activation of the domain.

Once the transaction is completed, advfs_bs_delete_fileset_tags will be called to delete each file in the tag directory of the fileset being removed. advfs_bs_delete_fileset_tags must be called without holding any locks and outside of the context of a transaction as it may take a very long time to remove each file in the fileset.

Once every file in the fileset is deleted, another exclusive transaction is started. The exclusive transaction makes sure that metadata is not modified for other filesets in the domain while this fileset is being removed. Once the exclusive transaction is started, an additional reference is placed on the fileset by calling bfs_access. The call to bfs_access does not reference any related snapsets and only bumps the fsRefCnt of the fileset to remove. Next, the bfSetTbl lock is acquired for write mode and bs_bfs_close is called to remove any references on related snapsets that were placed as a result of this snapset. If all other related snapsets were open only because of this fileset, then they will be closed and deallocated. If, however, any references already existed, the other snapsets may continue to exist. In either case, the fileset to be removed will still be open because of the call to bfs_access. Next, advfs_unlink_snapset will be called to decrement the fsRefCnt on the fileset to be removed to 0 and to unlink the fileset from its parent and sibling snapsets. On returning from advfs_unlink_snapset, the fileset to be removed is no longer logically associated with any other fileset and the bfSetTbl lock can be dropped.

Next, the fileset will be removed from the fileset delete pending list via a call to bfs_delete_pending_list_remove and the tag directory file will be deleted via a call to rbf_delete.

Finally, the bfSetTbl lock will be reacquired to synchronize with any lookups and the fileset structure (bfSetT) will be deallocated via a call to bfs_dealloc. The transaction will be ftx_done'd and a call to bs_close on the bfap of the tag directory will complete the deletion of the tag directory out of the root bitfile set.

In a cluster environment, if the fileset being removed is the last snapset child in a snapset tree, the ftx_done will be done as a synchronous log flush and a callout will be issued to CFS to notify it that direct writes can be initiated.

3.2.8.3.2.3 Execution Flow

- call advfs_can_remove_fileset get bf_set_ptr on success
- if sts != EOK
    - return the error
- start an exclusive transaction
- call advfs_can_remove_fileset_second_check
- if sts != EOK
    - bs_bfs_close(bf_set_ptr)
    - fail transaction
    - return sts
- Set BFS_DELETING in bf_set_ptr flags field
- Call bfs_delete_pending_list_add to add to delete pending list
- Finish the exclusive transaction doing a synchronous log flush (LOGSYNC)
- Call advfs_bs_delete_fielset_tags to remove all fileset in fileset
- Start an exclusive transaction
- Call bfs_access on fileset to remove to put an extra fsRefCnt on it.
- Acquire bfSetTbl lock for write

- Call bs_bfs_close on fileset to remove fsRefCnts on all associated snapsets.
- Call advfs_unlink_snapset to remove fileset from parents list of children snapsets
- Unlock bfSetTblLock
- Call bfs_delete_pending_list_remove to remove bf_set_ptr from delete pending list
- Call rbf_delete on tag directory of bf_set_ptr
- Call bfs_dealloc on bf_set_ptr to deallocate the bfSet structure
- Finish the exclusive transaction
- Call bs_close to close the tag directory file and cause it to be deleted.
- If the last child snapset in a domain was just removed and this is a cluster, the transaction would have been finished with a synchronous log flush and CFS will be notified that no snapshots remained.

3.2.8.3.3 advfs_can_remove_fileset

3.2.8.3.3.1 Interface

```
statusT
advfs_can_remove_fileset(
    bfSetIdT   bf_set_id,     /* in - bitfile set id */
    domainT    *dmnP,         /* in - set's domain pointer */
    bfSetT**   bf_set_ptr     /* out - point of fileset to delete (valid on EOK return)*/
)
```

3.2.8.3.3.2 Description

advfs_can_remove_fileset will open all filesets related to the fileset specified by bf_set_id and will verify that the fileset can be safely removed. If the fileset has any snapshot children, or if the fileset is not writable, then the fileset cannot be removed. Additionally, if after opening the fileset and it's related snapsets the fileset has a fsRefCnt of more than one, then the fileset cannot be removed because it is in use.

This routine will encapsulate much of the error checking that was done in fs_fset_delete on Tru64[15]. On error, advfs_can_remove_fileset will undone all its own actions including closing the fileset that was opened and all its related snapshots.

On successful return, bf_set_ptr will point to the newly opened fileset.

3.2.8.3.3.3 Execution Flow

- Call bfs_open passing in the fileset id and getting back the bfSet pointer
- Get the domain and fileset parameters
- If caller doesn't have write access to either domain or fileset
    - bs_bfs_close the fileset
    - bf_set_ptr = NULL
    - return E_ACCESS_DENIED
- if fileset has any children snapshots
    - bs_bfs_close the fileset
    - bf_set_ptr = NULL
    - return E_HAS_SNAPSHOT
- if fsRefCnt of fileset is > # of sibling snapshots
    - bs_bfs_close the fileset
    - bf_set_ptr = NULL
    - return E_TOO_MANY_ACCESSORS
- return EOK

[15] The call to bs_bfset_activate will remain in fs_fset_delete, but the call to bfs_open will occur in advfs_can_remove_fileset.

3.2.8.3.4 advfs_can_remove_fileset_second_check

3.2.8.3.4.1 Interface

```
statusT
advfs_can_remove_fileset_second_check(
    bfSetT**   bf_set_ptr     /* in - point of fileset to delete */
)
```

3.2.8.3.4.2 Description

This routine is a lightweight version of advfs_can_remove_fileset. It is to be called in the context of an exclusive transaction in which the fileset state will be set to BFS_DELETING. This routine will verify that the fileset was not transitioned to a state from which it cannot be deleted while the exclusive transaction was starting. The routine will check to make sure that no snapshots were created during as children of this fileset, and that the fsRefCnt is still acceptable to allow the deletion of the fileset.

It is assumed that the fileset is already open and valid when this routine is called. On error, this routine will not take any action against the fileset passed in (it will not close the fileset).

3.2.8.3.4.3 Execution Flow

- ASSERT bf_set_ptr is BFS_DELETING
- if fileset has any children snapshots
    - Return E_HAS_SNAPSHOT
- if fsRefCnt of fileset is > # of sibling snapshots
    - Return E_TOO_MANY_ACCESSORS
- return EOK

3.2.8.3.5 advfs_bs_delete_fileset_tags

3.2.8.3.5.1 Interface

```
statusT
advfs_bs_delete_fileset_tags(
    bfSetT**   bf_set_ptr     /* in - pointer to fileset to cleanup*/
)
```

3.2.8.3.5.2 Description

This routine will delete all files in a fileset and severe any connections between parents and sibling snapshots for each file. No file will have any snapshot children while it is being deleted since the fileset would not have been allowed to be removed if any snapshot children existed.

It is assumed that the fileset on which this routine is called is in the BFS_DELETING state and on the fileset delete pending list. Additionally, it is assumed that the transaction under which the fileset was put on the fileset delete pending list is on disk. The last assumption allows advfs_bs_delete_fileset_tags to break the link between parent and child snapshots (thus ceasing any further COWing) without the risk that the fileset will not end up deleted.

The goal of this routine is to process each entry in the tag directory of the fileset to be deleted. Processing of the tag directory entry consists of opening the file identified by the tag, calling rbf_delete on the file, disconnecting it from it's parents and sibling snapshots, and closing the file to allow for final close processing and storage deallocation. Since the deletion of all files in a fileset may potentially be a very long operation, the routine holds no locks while walking the tag directory and processing the files.

advfs_bs_delete_fileset_tags will merge the functionality of the routines delete_orig_set_tags and delete_clone_set_tags on Tru64. Changes in the way snapshots are designed have made the differences between snapshots and original filesets less severe and more easily handled in a common routine.

As each tag directory entry in a fileset to be deleted is read, the flags will be checked for the BS_TD_VIRGIN_SNAP flag. If set, the tag entry represents a snapshot that has not been COWed to by a parent (no metadata has been created for the child). Since the fileset state is BFS_DELETING, the parent access structure will not be able to open this file to force a COW. Deleting such a file consists of simply deleting the tag directory entry as long as the file is not already in cache. If bs_access_one with the BF_OP_INMEM_ONLY flag does not return an access structure, then deleting the tag is sufficient. If bs_access_one with the BF_OP_INMEM_ONLY flag does return an access structure, then the access structure will be unlinked from its parents and closed to complete the delete.

As this routine processes each tag and opens the file, the BFA_XTNTS_IN_USE and BFA_IN_COW_MODE flags will be examined. If either of these flags is set, advfs_bs_delete_fileset_tags will block and wait for those flags to clear. This is necessary because while these flags are set, the chain of snapshot children may be walked without holding any locks. It is unsafe to unlink a snapshot child from its parent while these flags are set. The wait will occur in the advfs_unlink_snapshot routine.

In the event that BS_TD_VIRGIN_SNAP is not set, the file must be accessed so that it can be deleted. bs_access_one will be called to open only the file to be deleted and not its parents. The call to bs_access_one will pass in the BF_OP_IGNORE_BFS_DELETING flag to indicate that the file must be opened even though the fileset is being deleted. Once open, the bfap will be checked to see if a parent pointer or a sibling pointer exists. If one does, then the file was already open and will be unlinked from parents and siblings.

Once all unlinking is completed, the file can have rbf_delete called on it and bs_close_one can be called to do the final processing and storage deallocation. For a fileset without any snapshots, the overhead attributed to snapshots is simply checking the BS_TD_VIRGIN_SNAP flag and checking a pointer to see if the access structure has a parent or sibling.

3.2.8.3.5.3 Execution Flow

- ASSERT bf_set_ptr is BFS_DELETING
- cur_tag = NilBfTag
- file_count = 0
- While (true)
    - tagdir_lookup_next(cur_tag)
    - if tagdir_lookup_next return ENO_SUCH_TAG
        - break
    - if tagdir_lookup_next fails
        - domain panic
    - if cur_tag->bft_tag_flags & BS_TD_VIRGIN_SNAP
        - /* Tag does not have it's own metadata */
        - call bs_access_one on cur_tag with BF_OP_INMEM_ONLY|BF_OP_IGNORE_BFS_DELETING|BF_OP_INTERNAL flag to get cur_bfap /* If found in cache, the parents must also be open */
        - if bs_access_one fails
            - output error message
            - continue
        - ASSERT bfap->bfaFlags & BFA_SNAP_VIRGIN
        - call tagdir_remove_tag on cur_tag
        - if cur_bfap != NULL
            - /* snapshot was in cache */
            - advfs_unlink_snapshot cur_bfap
            - /* Setting state to ACC_INVALID will make sure cur_bfap is freed after last close */
            - lock cur_bfap->bfaLock
            - set state to ACC_INVALID
            - unlock cur_bfap->bfaLock

                - bs_close_one cur_bfap (use MSFS_BFSET_DEL flag)
        - else
            - bs_access_one cur_tag to get cur_bfap (use BF_OP_IGNORE_BFS_DELETING|BF_OP_INTERNAL flag)
            - advfs_unlink_snapshot cur_bfap
            - bs_delete cur_bfap
            - bs_close_one cur_bfap (use MSFS_BFSET_DEL flag)
        - file_count++
        - if file_count && file_count % ADVFS_FILES_BEFORE_PREEMPTION_POINT == 0
            - preemption_point
            - /* Allow other processes to run since this may be a long loop */
- return EOK

3.2.8.3.6 advfs_unlink_snapshot

3.2.8.3.6.1 Interface

```
statusT
advfs_unlink_snapshot(
    bfAccessT* bfap           /* in - bfap to unlink from parents. */
)
```

3.2.8.3.6.2 Description

This routine will unlink a bfap from its parent's list of snapshot children. If bfap has no parent, then this routine has nothing to do. If bfap does have a parent, the parent's bfaSnapLock will be acquired in write mode and the child list will be relinked without bfap in it. After being removed from its parent's list of children, bfap will have it's refCnt adjusted to 1 so that a last close can occur.

It is assumed that bfap has no child snapshots and that it is only being held open by its parent bfap.

3.2.8.3.6.3 Execution Flow

- ASSERT bfap->bfaFirstSnapChild == NULL
- If bfap->bfaParentSnap == NULL
    - return EOK
- write lock bfap->bfaParentSnap->bfaSnapLock
- while bfap->bfaFlags & BFA_XTNTS_IN_USE || bfap->bfaFlag & BFA_IN_COW_MODE
    - cv_wait on bfaSnapCv using bfaSnapLock to synchronize
- if bfap->bfaParentSnap->bfaFirstSnapChild == bfap
    - bfap->bfaParentSnap->bfaFirstSnapChild = bfap->bfaNextSnapSibling
- else
    - prev_child = bfap->bfaParentSnap->bfaFirstSnapChild
    - cur_child = prev_child->bfaNextSnapSibling
    - while cur_child != bfap
        - prev_child = cur_child
        - cur_child = cur_child->bfaNextSnapSibling
    - prev_child->bfaNextSnapSibling = bfap->bfaNextSnapSibling
- unlock bfap->bfaParentSnap->bfaSnapLock
- /* There should be one access for the bs_access_one that opened this and one from the parent bfap */
- ASSERT refCnt == 2
- Lock bfap->bfaLock
- refCnt = 1
- unlock bfap->bfaLock
- ASSERT bfap->bfaFlags & BFA_OPENED_BY_PARENT

- ASSERT bfap->bfaFlags & BFA_EXT_OPEN is not set
- return EOK

3.2.8.3.7 advfs_unlink_snapset

3.2.8.3.7.1 Interface

```
statusT
advfs_unlink_snapset(
    bfSetT**          bf_set_ptr     /* in - pointer to fileset to unlink*/
    snap_flags_t      snap_flags     /* in - SF_HAD_PARENT is set if a parent exists */
    ftxHT             parent_exc_ftx /* in - an exclusive transaction handle */
)
```

3.2.8.3.7.2 Description

This routine will remove the fileset described by bf_set_ptr from its parents list of child snapsets.

If the parent fileset's bfaFirstSnapChild is equal to bf_set_ptr's bfSetId, then the parent's bfaFirstSnapChild will be set to the next sibling of bf_set_ptr (potentially NULL meaning there was only one child). If the parent's bfaFirstSnapChild is not equal to bf_set_ptr, then the list of child snapsets will be walked until one is found that precedes bf_set_ptr. The fileset that precedes bf_set_ptr will be adjusted so that the next sibling pointer points to the current next sibling pointer of bf_set_ptr (potentially NULL). All the adjustments will be made in memory and on disk. The on disk changes will be in terms of file set ids while the in memory changes will be in terms of pointers.

When this routine is called from bs_bfs_delete, the reference put on the parent and sibling snapsets has already been removed. As a result, there is no guarantee that the related snapsets will still exist in memory. If the SF_HAD_PARENT flag is set, then this is a snapshot with a parent that needs to have at least the on disk structures unlinked. If the SF_HAD_PARENT flag is set and the pointers in bf_set_ptr to next and parent snapsets are non-NULL, then the in memory versions also need to be updated. Since this routine is called in the context of an exclusive transaction, it is safe to modify the parent and sibling bfSet structures without an fsRefCnt on them.

It is assumed that this routine is called in the context of an exclusive transaction. Since the transaction is exclusive, no new snapsets can be created or deleted from the snapset chains. As a result, it is safe to manipulate the snapset lists in this routine without explicitly locking.

3.2.8.3.7.3 Execution Flow

- if SF_HAD_PARENT and bf_set_ptr->bfaParentSnapSet == NULL
    - on_disk_update_only = TRUE
    - /* This is the slow path. We only need to update on disk, but we need to go to disk to find out what to update. */
    - read bfSetAttr for bf_set_ptr to get set id of parent
    - open the tag directory file of the parent
    - current_open_tagdir_bfap = parent's tagdir bfap
    - read BSR_BFS_ATTR record from parent's tag directory
    - if bf_set_ptr->set_id == parent's bfsaFirstChildSnapSet
        - is_first_child = TRUE
    - else
        - Find the set that precede bf_set_ptr in the child list
        - prev_set_tag_bfap = open tag dir file of first child snapset

            - prev_bfs_attr = read BSR_BFS_ATTR record from prev_set_tag_bfap
            - cur_set_id = prev_bfs_attr bfsaNextSiblingSnapSet
            - close current_open_tagdir_bfap
            - while cur_set_id.dirTag != bf_set_ptr->dirBfap.tag
                - close prev_set_tag_bfap
                - prev_set_tag_bfap = open prev_bfs_attr bfsaNextSiblingSnapSet
                - prev_bfs_attr = read BSR_BFS_ATTR record from prev_set_tag_bfap
                - cur_set_id = prev_bfs_attr bfsaNextSiblingSnapSet
            - next_snap_sibling_id = read BSR_BFS_ATTR of bf_set_ptr tag dir file to get next snap sibling id.
            - current_open_tagdir_bfap = prev_set_tag_bfap
            - close_curent_open_tag_bfap = TRUE
            - /* Now prev_bfs_attr is the field that needs to be updated on disk, and next_snap_sibling_id is what it's next field needs to point to */
- else if SF_HAD_PARENT and bf_set_ptr->bfaParentSnapSet != NULL
    - /* Fast path, in memory snapset structure is still setup so we can take advantage */
    - on_disk_update_only = FALSE
    - if bf_set_ptr-:bfsParentSnapSet->bfsFirstSnapChild = bf_set_ptr
        - is_first_child = TRUE
        - current_open_tagdir_bfap = parent's tagdir bfap
    - else
        - /* Need to search for previous snap sibling to update */
        - prev_bf_set = parent->bfaFirstChildSnapSet
        - cur_bf_set = prev_bf_set->bfaNextSiblingSnapSet
        - while cur_bf_set != bf_set_ptr
            - prev_bf_set = cur_bf_set
            - cur_bf_set = cur_bf_set->bfaNextSiblingSnapSet
        - current_open_tagdir_bfap = prev_bf_set's tagdir bfap
        - prev_bfs_attr = read BSR_BFS_ATTR from current_open_tagdir_bfap
    - next_snap_sibling_id = bf_set_ptr's next sibling's bfSetId
    - close_current_open_tag_bfap = FALSE
- /* Setup for the actual updates is not complete. */
- if SF_HAD_PARENT
    - /* Update parent to point to next sibling of bf_set_ptr. Since we are in an exclusive transaction, no one else could be modifying the ODS.   */
    - start transaction
    - pin the page and record of the prev_bfs_attr
    - setup undo record with previous snapshot info in the fileset attributes
    - if is_first_child
        - set bfsaFirstChildSnap = next_snap_sibling_id

        - else set bfsaNextSiblingSnapSet = next_snap_sibling_id
        - if !on_disk_update_only
            - lock bfaSnapMutex of parent
            - if first_child
                - parent->bfaFirstSnapChild = bf_set_ptr->bfaNextSnapSibling
            - else prev_bf_set->bfaNextSibling = bf_set_ptr->bfaNextSiblingSnap
            - unlock bfaSnapMutex
        - finish transaction
- if close_current_open_tag_bfap
        - close current_open_tag_bfap

3.2.8.3.8 Miscellaneous Changes

delete_clone_set_tags will be removed and the logic will be merged into a common routine for snapshots and non-snapshot files.

tagdir_lookup_next will be modified to return the tagFlags value in the bft_tag_flags field of the bfTagT structure that it returns.

3.2.9 Locking Overview

3.2.9.1 Predicted Lock Hierarchy

The locks listed first are acquired before those listed second. These numbers only indicate lock hierarchy relative to AdvFS and not other subsystems. Tru64 list is below.

Complex Locks

1. DmnTblLock
2. InitLock
3. kdmLock (fsContext)
4. cnode lock (CFS)
5. file_lock (fsContext)
6. rmvolTruncLk (domainT)
7. **bfaSnapLock (bfap)**
8. ~~clu_clonextnt_lk (bfap)~~
9. ~~trunk_xfer_lk (bfap)~~
10. migStg_lk (bfap)
11. ddlActiveLk
12. ftxSlotLock (domainT)
13. bfSetTblLock
14. quotaInfoT_qiLock
15. ~~fragLock (bfSetT)~~
16. dirLock (bfSetT)
17. ~~cow_lk (bfap)~~
18. ~~clone_migStg_lk (bfap)~~
19. mcellList_lk (bfap)
20. xtntMap_lk (bfap)
21. FilesetLock
22. mcell_lk (vdT)
23. del_list_lk (vdT)
24. BMT mcellList_lk (BMT bfap)
25. BMT xtntMap_lk (BMT bfap)
26. scLock (domainT)
27. rbmt_mcell_lk (vdT)

28. stgMap_lk (vdT)
29. xidRecoveryLk (domainT)

Complex locks out of hierarchy

- TraceLock
- dqLock
- ~~cow_lk~~
- descLock
- flushLock

3.2.10 Extent Manipulation

3.2.10.1 advfs_get_blkmap_in_range

3.2.10.1.1 Interface

```
statusT
advfs_get_blkmap_in_range (
        bfAccessT *bfap,        /* IN - Access struct for file                */
        bsInMemXtntMapT *xtnt_map,      /* IN - Extent map to use to          *
                                 * generate range maps                       */
        off_t *offset,          /* IN - offset in file to start range map     */
                                /* OUT - offset adjusted to correct           *
                                 * alignment                                  */
        size_t length,          /* IN - length of range to map                */
        extent_blk_desc_t **extent_blk_desc,
                                /* IN - pointer to an extent_blk_desc         *
                                 * OUT - pointer to head of list that maps    *
                                 * the given range                            */
        uint64_t *xtnt_count,   /* IN - a pointer or NULL                     *
                                 * OUT - Overloaded meaning. See Above        */
        round_type_t round_type,/* IN - type of rounding to be performed      */
        extent_blk_map_type_t extent_blk_map_type,
                                /* IN - determines the map type. (sparse,     *
                                 * stg, both                                  */
        int blkmap_flags        /* In - flags for the function                */
        )
```

3.2.10.1.2 Description

This routine generates a linked list of extents representing the extent maps of bfap. For snapshots, this routine will compose the extent maps from the mapped extents of bfap and the mapped extents of the parents. If the extent_blk_type is EXB_DO_NOT_INHERIT, then the extents returned will only be composed from the extent maps of bfap and will not look to parents for extent information. If EXB_DO_NOT_INHERIT is set, then EXB_ONLY_HOLES will include both unmapped regions (XTNT_TERM) and COWed holes (COWED_HOLE).

A new round_type_t will be supported that rounds extents to include entire holes regardless of the requested offset and length. This is used to allow holes to be completely COWed in one operation rather than COWing parts of holes.

3.2.10.1.3 Execution Flow

- ASSERT only one of EXB_COMPLETE, EXB_ONLY_HOLES and EXB_ONLY_STG is set
- ASSERT that EXB_DO_NOT_INHERIT and RND_ENTIRE_HOLE are not both set
- If XTNT_LOCKS_HELD
    - ASSERT xtnts are XVT_VALID
- Else

- if extent_type & EXB_DO_NOT_INHERIT || bfap->bfaParentSnap == NULL
  - x_load_inmem_xtnt_map
- else
  - advfs_acquire_xtntMap_locks
- no change to error logic

- switch (round_type)
  - case RND_MIGRATE,RND_VM_PAGE, RND_ALLOC_UNIT, RND_ENTIRE_HOLE:
    - no change to logic
  - case RND_NONE
    - if *offset & DEV_BSIZE
      - if unlock_xtntlock
        - if extent_type & EXB_DO_NOT_INHERIT || bfap->bfaParentSnap == NULL
          - unlock xtntMap_lk
        - else
          - advfs_drop_xtntMap_locks

- source_bfap = bfap
- if EXB_DO_NO_INHERIT is set or bfap->bfaParentSnap == NULL
  - sts = imm_get_xtnt_desc(bfap, start_fob, &xtnt_desc)
- else
  - sts = advfs_get_snap_xtnt_desc(bfap, start_fob, &xtnt_desc,&source_bfap)
- if sts != E_RANGE_NOT_MAPPED
  - if RND_MIGRATE
    - no change to logic
  - if RND_ALLOC_UNIT
    - no change to logic
  - if RND_ENTIRE_HOLE and xtnt_desc is a hole (XTNT_TERM or COWED_HOLE)
    - cur_fob = start_fob = xtnt_desc. bsx_fob_offset
    - *offset = ADVFS_FOB_TO_OFFSET(start_fob)
  - do {
    - if RND_MIGRATE
      - no change to logic
    - else if… the logic will remain the same, but the extent_blk_map_type will now be check by bitwise AND with the EXB_ONLY_HOLES, EXB_ONLY_STG and EXB_COMPLETE types.
      - if !XTNT_NO_MAPS
        - if XTNT_NO_WAIT
          - malloc cur_range (extent_blk_desc)
          - if cur_range == NULL
            - if unlock_xtntlock
              - if extent_type & EXB_DO_NOT_INHERIT || bfap->bfaParentSnap == NULL
                - unlock xtntMap_lk
              - else
                - advfs_drop_xtntMap_locks
            - return E_WOULD_BLOCK
        - else
          - cur_range = malloc extent_blk_desc
        - /* Initialize the cur_range */
        - cur_range->ebd_snap_fwd =NULL
        - cur_range->ebd_bfap = source_bfap

- ASSERT (!EXB_DO_NOT_INHERIT) || (EXB_DO_NOT_INHERIT && source_bfap == bfap)
- if (EXB_ONLY_HOLES || EXB_COMPLETE) && (XTNT_TERM || COWED_HOLE)
  - if RND_ENTIRE_HOLE
    - cur_range->ebd_byte_cnt = ADVFS_FOB_TO_OFFSET(xtnt_desc.bsxdFobOffset+xtnt_desc.bsxdFobCnt - cur_fob)
  - else
    - no change to logic
  - cur_range->ebd_vd_index = 0
- else if (EXB_ONLY_STG || EXB_COMPLETE) && (!XTNT_TERM && !COWED_HOLE)
  - no change to logic

- else (XTNT_NO_MAPS is TRUE)
  - if EXB_ONLY_HOLES
    - if unlock_xtntlock
      - if extent_type & EXB_DO_NOT_INHERIT || bfap->bfaParentSnap == NULL
        - unlock xtntMap_lk
      - else
        - advfs_drop_xtntMap_locks
  - else
    - no change to logic

- else
  - /* This extent is to be skipped */
  - no change to logic
- if bfap->bfaParentSnap or EXB_DO_NO_INHERIT is set
  - sts = imm_get_xtnt_desc(bfap, start_fob, &xtnt_desc)
- else
  - sts = advfs_get_next_snap_xtnt_desc(bfap, start_fob, &xtnt_desc,&source_bfap)

- while cur_fob < end_fob && sts == EOK && !error

- if cur_fob < end_fob && (EXB_COMPLETE || EXB_ONLY_HOLES)
  - if !XTNT_NO_MAPS
    - no change to logic for malloc of cur_range
    - cur_range->ebd_snap_fwd =NULL
    - cur_range->ebd_bfap = bfap
    - ASSERT EXB_DO_NOT_INHERIT || bfap->bfaParentSnap == NULL
    - No change in logic
  - else
    - if EXB_ONLY_HOLES
      - no change to logic
      - if unlock_xtntlock
        - if extent_type & EXB_DO_NOT_INHERIT || bfap->bfaParentSnap == NULL
          - unlock xtntMap_lk
        - else
          - advfs_drop_xtntMap_locks
      - return EOK
    - else if EXB_COMPLETE
      - no change to logic

- no change in logic
- if !XTNT_LOCKS_HELD
    - if extent_type & EXB_DO_NOT_INHERIT || bfap->bfaParentSnap == NULL
        - unlock xtntMap_lk
    - else
        - advfs_drop_xtntMap_locks
- return EOK

3.2.10.2 advfs_get_snap_xtnt_desc

3.2.10.2.1 Interface

```
statusT
advfs_get_snap_xtnt_desc (
               bf_fob_t              fob_offset,    /* in */
               bfAccessT             *bfap,         /* in */
               bsInMemXtntDescIdT    *xtnt_desc_id, /* out */
               bsXtntDescT           *xtnt_desc     /* out */
               bfAccessT             *source_bfap   /* out */
                   )
```

3.2.10.2.2 Description

This routine performs the same basic operation as imm_get_xtnt_desc. The routine can be called to return a bsXtntDescT structure which represents an extent in a file. If bfap has a parent snapshot, and if the extent in bfap is unmapped (an XTNT_TERM extent) then the parent bfap will be examined for its extent descriptor. The bsXtntDescT will represent the first mapped (either hole or storage) extent in which fob_offset is described.

Logically, this routine will return the extent descriptor which, in the extent maps composed by repeatedly collapsing the extent maps of the child snapshot up to the root of the snapshot tree, contains the fob_offset requested. As the extent maps are collapsed, unmapped ranges of the child will be replaced by mapped regions of the parents.

The parameter source_bfap is a pointer to the bfap from which the extent descriptor was acquired.

This routine assumes the extent maps of bfap and all its parents are locked for read access.

The xtnt_desc_id returns by this routine is the id of the extent descriptor in bfap that would map fob_offset if it were mapped in bfap.

3.2.10.2.3 Execution Flow

- max_fob_from_parent = ADVFS_OFFSET_TO_FOB_UP(bfap->bfa_orig_file_size)
- sts = imm_get_xtnt_desc(fob_offset, bfap->xtnts->xtnt_map, bfap_xtnt_desc_id, bfap_xtnt_desc)
- if bfap->bfaSnapParent == NULL
    - return
- if sts == E_RANGE_NOT_MAPPED && fob_offset > bfap->bfa_orig_file_size
    - return sts
- else if sts != E_RANGE_NOT_MAPPED
    - return sts
- child_xtnt_not_mapped = (sts == E_RANGE_NOT_MAPPED)
- cur_xtnt_desc = child_xtnt_desc
- cur_bfap = bfap->bfaParentSnap

- while (sts == E_RANGE_NOT_MAPPED || cur_xtnt_desc is XTNT_TERM) && (cur_bfap != NULL)
    - sts = imm_get_xtnt_desc(fob_offset, cur_bfap->xtnts->xtnt_map, &cur_xtnt_desc, &cur_xtnt_desc_id)
    - if sts != EOK cur_bfap->bfapSnapParent == NULL || sts != E_RANGE_NO_MAPPED
        - return sts
    - if cur_bfap->bfaSnapParent != NULL && sts == E_RANGE_NOT_MAPPED
        - cur_bfap = cur_bfap->bfaSnapParent
        - continue
    - if cur_bfap->bfaSnapParent == NULL || cur_xtnt != XTNT_TERM (not unmapped hole)
        - /* At root, clip extent to child_xtnt_desc and return */
        - if child_xtnt_not_mapped
            - /* If the child extent lookup returned E_RANGE_NOT_MAPPED, clip the parent's xtnt_desc at bfap's orig_file_size so we don't get extent maps from the parents beyond the COWable region */
            - cur_xtnt_desc->bsxdFobCnt = MIN(bfa_orig_file_size (in fobs) - cur_xtnt_desc->bsxdFobOffset, cur_xtnt_desc->bsxdFobCnt)
            - xtnt_desc = cur_xtnt_desc
            - return EOK
        - else
            - /* If the child had an extent, that extent must have been a hole (an unmapped region as opposed to a COWed hole). Clip the parent's extent descriptor to the unmapped region of the child */
            - clip_fob_offset = MAX(child_xtnt->bsxdFobOffset, cur_xtnt_desc->bsxdFobOffset)
            - clip_fob_cnt = MIN(child_xtnt->bxsdFobOffset+child_xtnt->bsxdFobCnt, cur_xtnt->bsxdFobOffset+cur_xtnt->bsxdFobCnt)
            - cur_xtnt_desc->bsxdFobOffset = clip_fob_offset
            - cur_xtnt_desc->bsxdFobCnt = clip_fob_cnt
            - xtnt_desc = cur_xtnt_desc
            - return EOK
    - else
        - /* cur_bfap is still unmapped, need to go up another level */
        - cur_bfap = cur_bfap->bfaParentSnap
        - /* The child xtnt desc fob count will be clipped since the parent's unmapped range may be smaller than the child's */
        - clip_fob_cnt = MIN(child_xtnt->bxsdFobOffset+child_xtnt->bsxdFobCnt, cur_xtnt->bsxdFobOffset+cur_xtnt->bsxdFobCnt)
        - child_xtnt_desc->bsxdFobCnt = clip_fob_cnt
        - if child_not_mapped
            - /* If the child was unmapped, setup the child_xtnt so the clipping is correctly done when a mapped extent is found */
            - child_xtnt_desc->bsxdFobOffset = cur_xtnt->bsxdFobOffset
            - child_not_mapped = FALSE
        - continue
- return sts

3.2.10.3 advfs_get_next_snap_xtnt_desc

3.2.10.3.1 Interface

```
statusT
advfs_get_next_snap_xtnt_desc (
               bfAccessT             *bfap,         /* in */
               bsInMemXtntDescIdT    *xtnt_desc_id, /* in/out */
               bsXtntDescT           *xtnt_desc     /* out */
               bfAccessT             *source_bfap   /* out */
                   )
```

3.2.10.3.2 Description

This routine is the snapshot equivalent of imm_get_next_xtnt_desc. The routine will take an extent descriptor (xtnt_desc) and an extent descriptor id (xtnt_desc_id) and return the next extent in the snapshot. If the next extent in bfap is unmapped, this routine will look to the parent snapshots of bfap to find the correct extent information. This routine expects xtnt_desc to contain the last extent that was examined.

This routine will currently find the next extent descriptor by calling advfs_get_snap_xtnt_desc on the fob after the last fob mapped by xtnt_desc. In the future, this routine can be optimized to more intelligently traverse the extent maps.

3.2.10.3.3 Execution Flow

- `next_fob = xtnt_desc->bsxdFobOffset + xtnt_desc->bsxdFobCnt + 1`
- return advfs_get_snap_xtnt(next_fob, bfap, &xtnt_desc_id, &xtnt_desc, source_bfap)

3.2.10.4 advfs_make_cow_hole

This routine is adapted from the routine make_perm_hole in Tru64. It inserts a COWed hole anywhere in an extent map (in the middle or the end). This routine uses advfs_append_cow_hole and advfs_insert_cow_hole.

It is assumed that the migStg_lk is held for READ mode when this routine is called.

This should just return success if there storage already exists in the extent maps.

3.2.10.5 advfs_append_cow_hole

This routine is adapted from the routine append_perm_hole in Tru64. It appends a COWed hole to the end of an extent map.

3.2.10.6 advfs_insert_cow_hole

This routine is adapted from the routine insert_perm_hole in Tru64. It insert a COWed in the middle of extents.

3.2.10.7 advfs_get_xtnt_map (previously bs_get_clone_xtnt_map, bs_get_bf_xtnt_map, and bs_get_bkup_xtnt_map)

advfs_get_xtnt_map will merged version of bs_get_clone_xtnt_map and bs_get_bf_xtnt_map that will use advfs_get_blkmap_in_range and build extent maps for the requested file. Since advfs_get_blkmap_in_range will correctly compose the extent maps of child snapshots with those of its parent snapshots, there is no need to maintain two versions of this routine (for snapshots and not for

snapshots). The routine will lock the extent maps using the advfs_acquire_xtntMap_locks routine and will drop them using the advfs_drop_xtntMap_locks routine.

This routine will acquire the bfaSnapLock for read while trying generating the extents to return. If the BFA_XTNTS_IN_USE flag is set in the bfap's bfaSnapFlags, this routine will drop the bfaSnapLock and return an error indicating that the extents cannot be acquired at present. The BFA_XTNTS_IN_USE flag allows other threads that need to modify the extents and revoke the CFS token to set the flag and revoke the CFS token without holding any locks. The thread trying to acquire the extent maps cannot block since it currently holds the CFS token.

3.2.10.8 load_inmem_xtnt_map

When load_inmem_xtnt_map is called on a bfap that has BFA_SNAP_VIRGIN set, it will return EOK. This is so that the extent maps of the parent are not loaded into the child's extents. If the parent's extents were loaded, then the parent's storage was migrated, the child snapshot would have incorrect extents. This was a known issue on Tru64.

3.2.10.9 COWED_HOLES in child snapshots

Any time a hole is inserted into a writeable snapshot and that hole is not the result of an explicit COW operation, the hole will be a COWED_HOLE. By example, if a snapshot is writeable and has an original file size of 1k and is extended via a truncate to a size of 2k, the range from 1k to 2k will be a COWED_HOLE even though it was not explicity COWed. This modification will be made to all extent map routines that insert holes. An XTNT_TERM hole will never be inserted into a child as that extent would appear to be "unmapped" rather than an actual hole.

3.2.11 CFS Related Changes

3.2.11.1 Direct IO Writes from clients

To improve direct IO writes from cluster clients when snapshots exist, the advfs_get_xtnt_map routine will return information to clients to indicate whether or not an extent in a file has already been COWed. For ranges that have already been COWed, direct IO writes can occur as normal. For ranges that have not already been COWed, direct IO writes must be sent to the server to have the COW completed. To pass information about what ranges must be COWed, the high order bit of the bsed_fob_offset field of the bsExtentDescT structure will be set to 0 when a COW is required. On a fileset with no snapshots, the bit will always be 0. On CFS clients, if the CFS_EXP_HAS_CLONE is set in the cms_dbentry_t associated with a filesystem, then any direct IO writes to extents that have a 0 in the high order bit must be shipped to the server. Any extents that have a 1 in the high order bit can do a direct IO write as long as the direct IO token is held.

The flag is an advisory. If the flag is incorrectly set, it will only be incorrectly set to 0 and will therefore cause a write to be shipped to the server when that write could have occurred directly from the client.

The flag will be set in advfs_get_xtnt_maps by looking at the last child's extent maps and subdividing the extent maps to be returned based on unmapped ranges of the child. Whenever a new snapset is added, the CFS_SNAP_NOTIFY callout will invalidate the extent maps of all children, thereby clearing the flag and preventing incorrectly set hints. Since advfs_get_xtnt_maps must look at the last snapshot child, if BFS_IM_SNAP_IN_PROGRESS is set, it will return an error indicating that the direct IO token must be dropped.

3.2.11.2 advfs_get_xtnt_map

CFS clients will be modified to correctly handle an error when trying to acquire the extent maps for a file while holding that files direct IO token. In the event of an error, the client must back out far enough to drop the token, and then try again. If an error is returned, it means that the server was trying to get the token for exclusive access so that a COW can be performed. If the client succeeded it getting the extent

maps, they would be immediately invalidated, so preference is given to the server by forcing the client to try again.

3.2.11.3 advfs_getpage callers holding the file lock

If advfs_getpage must do any COW operations on userdata, it must invalidate the extent maps of any child snapshots. To invalidate the extent maps, advfs_getpage must call CLU_CFS_COW_MODE_ENTER. The call to CLU_CFS_COW_MODE_ENTER will acquire the cnode lock which is before the file lock in the hierarchy. advfs_getpage cannot safely drop a file lock that is held for write access, however it can drop a lock that is held for read access. Any callers of advfs_getpage that hold the file lock for write must invalidate the extent maps of the child and set the BFA_XTNTS_IN_USE flag in the children's bfaFlags.

For callers that hold the file lock for read on entrance, the file lock will be dropped and the BFA_XTNTS_IN_USE flag will be set in each child snapshot as the child has CLU_CFS_COW_MODE_ENTER called on it. Once each child has BFA_XTNTS_IN_USE set, the file lock will be reacquired for read. If the file lock is still held for read on exit, and if any children had CLU_CFS_COW_MODE_ENTER called on them, then before exiting advfs_getpage, the file lock will be dropped and CLU_CFS_COW_MODE_LEAVE will be called on each child. Additionally, the BFA_XTNTS_IN_USE flag will be cleared. If the file lock was held for read on entrance, it will be reacquired before exiting. If the file lock was not held for read on entrance, the setting of BFA_XTNTS_IN_USE flag and the calling of CLU_CFS_COW_MODE_ENTER will occur before the file lock is acquired.

If any calls to advfs_getpage return with the file lock held for write, the caller must call CLU_CFS_COW_MODE_LEAVE on each child snapshot and clear the BFA_XTNTS_IN_USE flag.

Any file lockers that set the BFA_XTNTS_IN_USE flag must be responsible for clearing the flag and broadcasting on the bfaSnapCv as the flag is cleared.

3.2.11.4 Migrate

Migrate must deal with invalidating any cached extent maps of any children snapshots of the file to be migrated.

3.2.11.4.1 migrate_clu_handling

3.2.11.4.1.1 Interface

```
static
statusT
migrate_clu_handling(
                bfAccessT *bfap,
                int32_t *do_cluster_cleanup,
                int32_t *clear_child_wait_for_xtnt_flag,
                )
```

3.2.11.4.1.2 Description

This routine is responsible for revoking the direct IO token for the file to be migrated, and for making sure that any snapshot descendants (children and grandchildren) have any cached extents revoked. To protect the extent maps of children during the migrate of the parent's extents, the BFA_WAIT_FOR_XTNTS flag will be set while holding the bfaSnapLock for write. Setting the BFA_XTNTS_IN_USE flag will cause CFS clients that are trying to acquire a copy of the extent maps to block in advfs_get_xtnt_map and wait for the migrate to complete. The waiters in advfs_get_xtnt_map will be woken up by a broadcast on the bfaSnapCv.

If the fileset that contains the file to be migrated is not mounted, then it is not necessary to revoke the CFS direct IO token, however, the children snapshots still must have their extents revoked if they are cached (if child_bfap->bfaFlags & BFA_CFS_HAS_XTNTS).

3.2.11.4.1.3 Execution Flow

- if clu_is_ready()
  - if fileset is mounted and bfap is not metadata and vnode is VREG
    - CC_CFS_CONDIO_EXCL_MODE_ENTER on bfap->bfVnode
    - On error, return ENO_MORE_MEMORY
    - do_cluster_cleanup = TRUE
  - if bfap is metadata
    - ASSERT (foreach descendant, BFA_CFS_HAS_XTNTS is not set)
    - clear_child_wait_for_xtnt_flag = FALSE
    - return EOK
  - if (bfSet->bfaFirstSnapCHild != NULL and bfap->bfaFirstSnapChild == NULL) || (bfap->bfaFlags & BFA_SNAP_CHANGE)
    - advfs_access_snap_children
  - if bfap->bfaFirstChildSnap != NULL
    - for each descendant of bfap
      - write lock bfaSnapLock
      - lock bfaLock
      - set BFA_XTNTS_IN_USE
      - unlock bfaLock
      - unlock bfaSnapLock
      - call CLU_CFS_COW_MODE_ENTER to revoke snapshots extent maps
      - write lock bfaSnapLock
      - lock bfaLock
      - clear BFA_CFS_HAS_XTNTS
      - unlock bfaLock
      - unlock bfaSnapLock
      - clear_child_wait_for_xtnt_flag = TRUE
- else return EOK

3.2.11.5 Future CFS Enhancements

3.2.11.5.1 Function Shipped COWs

To further optimize direct IO writes on a cluster, cluster clients will be provided a routine that will force a COW over a range of a file. When a client needs to do a direct IO write to a range in the file that is not already COWed (the ADVFS_CFS_COW_IS_COMPLETE flag is not set in those extents), the client will send a request to the serve to COW the range. On successful return, the client will be able to acquire the direct IO token and perform the write.

3.2.11.5.2 Optimized reads from client nodes

When performing frequent reads to child snapshots from a client node, significant performance degradation has been observed when the parent snapshot is being actively and frequently written (causing frequent COWing). Each modification to the parent file will invalidate any cached extent maps for child snapshots and will potentially introduce a new extent (increasing fragmentation) in the child snapshot. If the child is being frequently read while its extent maps are being invalidated, the CFS client will make frequent calls to the server to request extent maps that are ever increasing in size.

Tru64 resolved this issue by having a global flag to force all reads to be function shipped to the server. This flag could be set on a single cluster node and would cause all filesets with snapshot children to have reads function shipped to the server.

In the future, AdvFS will provide a mechanism for monitoring the frequency of requests from clients for extents maps and will enable or disable direct IO reads on a per file basis.

For HPUX 11.31, AdvFS will not provide a mechanism for disabling direct IO reads.

3.2.12 Miscellaneous Changes

3.2.12.1 fs_setattr

When a truncate occurs that extends the file, the hole must be a COWed hole. More generically, any hole inserted into a child snapshot must be a COWed hole type. This is to make sure behavior is correct after a snapshot child is truncated and then written to again. If the new holes weren't COWed holes, attempts could be made to COW into storage that is not really correctly associated.

A truncate of a parent file will trigger a call to advfs_force_cow_and_unlink prior to acquiring any locks and performing the truncate. In this model, if the truncate were to fail, the COW may have already occurred and will not be undone.

3.2.12.2 advfs_access_mgmt_thread

If advfs_access_mgmt_thread encounters a bfap that has the BFA_QUICK_CACHE flag set, it will halve the age time when making a decision as to whether or not to advance the access structure in the cache.

3.2.12.3 Migrate

When migrate calls advfs_get_blkmap_in_range, it will pass in the EXB_DO_NOT_INHERIT flag so that it only examines extents that are mapped by the bfap it is trying to migrate. This will prevent migrate from attempting to move storage that is mapped by a parent bfap.

The migStg_lk for write will protect the child from having storage allocated for COW operations during the migrate. The migStg_lk will no longer be dropped and reacquired for snapshots. Because of simplified locking and transaction management for snapshots, it is no longer necessary to acquire the migStg_lk in a different order for snapshots and parents. As a result, the starting of a transaction when migrating a snapshot child is no longer required.

mig_migrate will be modified so that the early exit condition is based on the BFA_SNAP_VIRGIN flag rather than the BS_BFSET_ORIG flag.

3.2.12.4 fs_fset_create

This routine will be modified to initialize the bfsaFilesetCreate field to contain the time at which the fileset was created. It will also be modified to initialize the new bfsaSnap* fields to 0 when not creating a snapset.

3.2.12.5 advfs_putpage

advfs_putpage will be modified to assert that if a file is a snapshot child, any writes that are issued to disk are mapped in an extent map that has ebd_bfap == bfap. This is to make sure that any dirty pages to be written during a migrate are actually mapped in the bfap that is being flushed. If a dirty page was mapped in a parent, it would indicate that the parent's storage was being migrated under it. This would cause a significant locking problem.

3.2.12.6 fs_create_file

When creating a new file, the BOF_ROOT_SNAPSHOT flag should be set in the bsBfAttr bfat_flags field. Additionally, the bfat_orig_file_size should be initialized to ADVFS_ROOT_SNAPSHOT value of (-1) and the bfat_del_child_cnt should be initialized to 0.

3.2.13 IO Completion

IO completion code will be modified slightly to deal with errors during COW operations. If the IOANCHORFLG_CHAIN_ERRORS flag is set on an IO anchor associated with a buf structure that is being processed by advfs_iodone, and if an error occurred on that buf structure, the freeiodesc flag will be cleared and the iodesc structure will be chained to the IO Anchor. Multiple errant IOs associated with the same IO Anchor will all be chained so that advfs_getpage can determine which IOs failed and can correctly mark snapshot children as out of sync.

The change in logic will occur after IO retry processing and should not impact IO retry.

3.2.14 Recovery Concerns

Recovery of a snapshot filesystem will present a significant challenge since it is necessary that a snapshot only be recovered if the exact original data is intact. If a snapset is marked as out-of-sync, it should not be recovered but should be removed from the domain. In the event of an out-of-sync snapset that is corrupted, there is no reliable way to know what is recoverable corruption and what are errors caused by an out-of-sync condition. For snapsets that are not out-of-sync, recovery will consist of repairing the snapset in a similar fashion to a fileset on Tru64. Any corrupt extents that are identified in a snapshot should cause the file to be marked as out of sync along with the snapset, but the snapset should not be removed.

3.2.15 On-Disk Impact

The basic set of information maintained on disk has not changed significantly from Tru64. Tools that directly examine on disk structures must continue to expect to find COWED_HOLES and unmapped holes in snapshot children and never in parent filesets. If the BFS_OD_ROOT_SNAPSHOT flag is set on a fileset, a COWED_HOLE should never be seen. If the BFS_OD_ROOT_SNAPSHOT is not set, then it may be the case that a COWED_HOLE or an XTNT_TERM (unmapped) hole is found in a file.

The links between parent and child snapshots have been expanded to have the flexability of mapping a tree in constant space. As a result, related snapsets should be seen as linked through the bfsaFirstSnapChildSnapShot, bfsaNextSiblingSnap, and bfsaParentSnapShot pointers which are the bfSetId's of the related snapshots.

3.2.16 Future Enhancements

3.2.16.1 Enhanced Out-Of-Sync handling

A flag could be introduced to indicate that it is more important to keep a snapshot in sync with the parent than it is to successfully complete writes to the parent. With this flag, a write that would cause a child snapshot to become out of sync would instead cause the write to the parent to fail.

3.2.16.2 Deferred deletion of parent snapshots

On deletion of parent snapshots, it is preferable that the entire file is not COWed to the children but is, instead, marked as, "Delete with last child." Marking the parent as deleted but not actually deleting it prevents the need to COW a large amount of data at the time of deletion. A counter has been introduced into the on-disk structure to allow for this future expansion. The counter indicates the number of children

that existed at the time the file was deleted. As children are removed, they must decrement the parent's counter if the parent is marked as "Delete with last child." The child to decrement the counter from one to zero must delete the parent file.

3.2.16.3 Forced Independence of Snapshot Child

It may be useful to provide the functionality to convert a snapshot into a snap clone and break all dependencies between the parent and child. In order to support such future work, a flag, BFS_OD_ROOT_SNAPSHOT is being introduced. Any fileset that has this flag set on disk will have been completely COWed or will have no dependency on any parent snapshot.

3.2.16.4 Inter-domain snapshots

A future enhancement might be to support inter-domain snapshots. Since snapshots on linked on-disk only through the bfSetId, it is theoretically possible to link snapsets that exist in different domain. Some locking issues exist with respect to creation and removal of filesets. The advantage of inter-domain snapsets would be the ability to mount filesets on different members of a cluster and the ability to prevent out of sync conditions be providing a disk that is large enough to hold the entire original. Mounting on a separate cluster client would likely required direct write capabilities.

3.2.16.5 ASYNC and NOWAIT support for snapshots

Currently, no simple solution has been developed to handle an asynchronous write that requires COWing. At some future point, some thought may need to be given into how to deal with an asynchronous write request in such a way that the write does not become synchronous.

4 Dependencies

4.1 System Administration

- Dependencies related to this area will be fully discussed in a design specification for AdvFS Snapshots in User Space

4.2 Memory Management

- No dependencies

4.3 ccNUMA

- No dependencies

4.4 Process Management

- No dependencies

4.5 File System Layout

- No dependencies

4.6 File Systems

- No dependencies

4.7 I/O System and Drivers

- No dependencies

4.8 Security

- No dependencies

4.9 Auditing

- No dependencies

4.10 Multiprocessor

- No dependencies

4.11 Behavior in a cluster

- Dependencies related to this area were fully discussed previously in this design.

4.12 Kernel Instrumentation/Measurement Systems

- No dependencies

4.13 Diagnostics

- No dependencies

4.14 Panic/HPMC/TOC

- No dependencies

4.15 Commands

- Dependencies related to this area will be fully discussed in a design specification for AdvFS Snapshots in User Space

4.16 Standards

- No dependencies

4.17 Kernel Debugger

- No dependencies

4.18 Boot Kernel

- No dependencies

4.19 Install Kernel

- No dependencies

4.20 Update/Rolling Upgrade

- No dependencies

4.21 Support Products

- No dependencies

4.22 Learning Products (Documentation)

- AdvFS online support documents must be updated to reflect new snapshot terminology.
- AdvFS manuals must be updated to reflect new snapshot terminology.
- AdvFS man pages must be updated.

5 Issues (Optional)

High Priority

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

Medium Priority

- Kernel recursion can pose risks. The design currently limits the number of recursive calls in the kernel but still uses recursion. There is some concern about continuing to use recursion. The design will use recursion initially but may be changed later to use a non-recursive algorithm. The recusion is contained in routines such that the implementation can change without impacting most of the design.
  - Owner:
  - Contact:
  - Status: Open

Low Priority

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