AdvFS Storage Allocation/Reservation

Design Specification

Version 1.0

DA

CASL

Building ZK3
110 Spit Brook Road
Nashua, NH 03062

Design Specification Revision History		
Version	**Date**	**Changes**
1.0	3/18/04	First draft for internal review

Preface

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

Author Name	Mailstop	Email Address
DA		

Sign-off review

Approver Name	Approver Signature	Date

1 Introduction

1.1 Abstract

This design describes a kernel implementation for a new storage allocator for AdvFS. The design provides the basis for the implementation of this allocator on HPUX.

1.2 Product Identification

Project Name	Project Mnemonic	Target Release Date
AdvFS Storage Allocation	AdvFS STGALLOC	

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 Storage allocation 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	Smart Store design Spec	
3		

1.5 Purpose of Document

This design presents a description of a new storage allocator to be implemented for HP-UX. This design introduces a policy based storage allocator. Some policies will have infrastructure in place but will not be implemented. The flexibility will exist for future policies to be added.

1.6 Acknowledgments & Contacts

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

TM, DB and CW.

1.7 Terms and Definitions

Term	**Definition**
Allocation unit	This term refers to the minimum increment of storage that will be handed out.
Storage Cluster	This term refers to the increment of storage that will be reserved per request if available.

2 Design Overview

2.1 Overview of existing storage code

The flow of code for storage involves extent maps and the SBM. This project is concerned with the SBM portion and below. The SBM storage model uses a static bitmap to keep track of all storage available on a volume. Each bit represents a unit of storage. The value of the bit determines if the storage is in use or free. In order to speed up searching thru the SBM for free bits, an in memory cache exists. This cache represents a portion of the SBM. As this cache is depleted, the next range of SBM bits are loaded into the cache.

All allocations will basically request storage from this cache on a first come first served basis. This approach leads to fragmentation of storage per file when there are simultaneous requestors. The nature of the SBM also makes it expensive to find free runs of storage since multiple pages of the SBM may need to be read into memory.

All calls to allocate and remove storage funnel thru a few routines. The new storage allocation model will leave these routines in place and maintain the same interfaces. In this way code above these routines will not need to change.

2.1.1 Stg_alloc_one_xtnt

This routine is the interface to the SBM for requesting storage. This will return a single contiguous extent of storage based on the requested passed in size. The routine will return NULL if there is not enough space (contiguous or not) to completely satisfy the request. Otherwise it will return the largest contiguous extent it can find that is within the requested size. This returned storage may be less than the requested size indicating to the caller that they must call back in (also indicating that there is enough space available to completely satisfy the request).

The call that does the work of finding space is sfm_find_space. This will be the routine that actually returns the storage. It does not however mark the storage as in use yet.

If there is no storage, then the code checks to see if vfast has reserved any storage for its use and if so releases it and retries. If no other storage can be found then this reserved range of storage will be stolen from VFAST. This reservation scheme will also be implemented in the new storage model.

Once storage is located in the SBM, we need to make sure that not too many SBM pages get pinned. This will be done by reducing the size of the request by the amount of pages in the SBM that we can safely pin. We will no longer need to do this.

The final step after potentially reducing the size of the request to avoid pinning too many pages is to call into sbm_remove_space to actually mark the storage as in use.

2.1.2 Sbm_find_space

This interface will be replaced in the new storage model. One of the first things sbm_find_space does is to check to see if there is any reserved space set aside for the BMT and if not it goes ahead and does this. We will no longer need to have this pre-reserved space for the BMT anymore or at least not in this manner.

Vfast sets up a reserved region of space in the SBM for migrating to and from when moving files around. If the caller needs to use this area, the BS_ALLOC_MIG_RSVD hint is passed in and we use this area of the SBM instead.

Next we have special code to deal with NFS requests. This will be removed. The new storage policy model will determine how to deal with allocating storage. NFS will have its own policy.

Next we search the in-memory cache of storage descriptors. To ease the pain of searching the SBM bits we keep an in-memory cache of offset, length pairs that describe the storage in the SBM. This cache only represents a portion of the SBM. If we find a descriptor that satisfies the request we return a pointer to it. Note that we also have a picky allocation flag that determines if we give back exact matches or first fit matches. This flag will only be used (currently) by VFAST since the reservation aspects of this design should otherwise alleviate the need for this.

If we did not find a descriptor in the in-memory cache that suits our needs, we will reload the in memory cache with the next set of SBM pages. We do some vfast special processing to force the cache load to be more aggressive about making sure that a contiguous extent of the requested size is loaded into the cache.

After calling load_x_cache we may find that there was nothing returned, this could mean that we dipped into reserved space set aside for the BMT. If so the code will move past this space and restart the scan. If there is still no space we then will steal space from the reserved descriptor.

2.1.3 Del_xtnt_array

This is the routine that all storage removal funnels thru. This routine will still exist but will now interact with the new storage allocation code.

This routine will process an Mcell. The caller of this routine (del_dealloc_stg) will parse a list of Mcells (on the deferred delete list) and call this routine for each Mcell. The Mcell will be read from disk and the subextentmap represented by the Mcell will be process to form ranges of storage that are to be deleted.

Each range of storage will call del_range to mark the bits in the SBM as free. Del_range will be replaced with a call to the new storage allocator. Since a range of storage will cause bits in the SBM to be changed, all of this must be under transaction control. The routine needs to keep track of the number of pages being pinned and back off and start a new subtransaction if we are pinning too many pages. This will no longer be necessary since a range of storage is represented by a single entry in the b-tree regardless of the size of the range.

2.2 Design Approach

The design approach will be to introduce storage policies. These policies will dictate how storage is handed out to the requestor. Depending on the policy storage may be handed out so as to be contiguous with previous requested storage. This will be achieved by using storage clusters. A cluster will be an amount of storage (currently 1MB) that will allow future requests to obtain contiguous on-disk storage. A given request will be granted but a cluster of storage will be set aside for this file so that future requests may draw from this. This will be a soft reservation in that once all unreserved storage is used then the soft reserved storage will become available to all files. Storage will be organized so that it may be looked up by disk offset.

Overhead is a concern and storage will therefore not be organized by size. A size will be attempted but storage will not be exhaustively search for a given size, unless the policy dictates this.

2.3 Overview of Operation

All storage allocation calls today funnel down to two routines (as discussed previously). These routines will operate on one chunk of contiguous space. This project will replace all of the code from this point down. Storage will still be allocated and deallocated in one single contiguous chunk. All code above this call should have minimal changes.

The storage allocator will operate using the transaction model that was used to manage the directory index b-trees. Much of this code will be leveraged. Allocating and deallocating of storage will not necessarily be faster but it will provide for better continuity of storage layout for files and thus provide overall better read and write performance.

2.3.1 Allocation Size

Storage will be given out in minimum allocation units. This will be configured at the domain creation time. Any storage that is granted will be in this minimum amount, and be contiguous in this amount. A request may contain multiple allocation units but always in multiples of this unit. We will allow for user data and metadata to have different allocation units. Currently user data will not be larger than the metadata allocation unit. This design will allow for this however.

2.3.2 Request Size

The request size is the amount of storage required to satisfy the request. This may or may not be one contiguous piece of storage. The request will always be rounded up to allocation units and it will be the responsibility of the upper layers to zero any storage handed to them that is beyond the requested amount.

2.3.3 Cluster Size

The cluster size refers to an amount of storage that will be set aside (if possible) for any given request. This will not be directly handed out but successive requests for storage that fall with in a cluster will be contiguous. The cluster size will be chosen as 1MB. Since a transfer of data to a disk benefits from contiguous disk space up to the maximum transfer size of the device, there is no point in making a cluster too large (except for minimizing the size of the extent map that maps the storage).

2.3.4 Soft Reservation of storage

A request for storage will be granted and at the same time a cluster of storage (minus the request) will be put aside for that file. This soft reservation will only be given out to this file when a request for storage falls within this cluster. It will not be used for other files until all unreserved space is exhausted. At this point the reserved storage will be reduced by half and handed out. Leaving half still reserved to the original file and half to the new file. If the request is for more than half of the reservation the caller will have to call back in for more storage. Storage will basically be reserved for ranges within a file.

2.3.5 Alignment of Storage

Unreserved storage will be aligned on cluster boundaries in cluster increments when possible. As storage is depleted and then given back as unreserved, it will be kept aligned. Cases will exist where this will not be possible but these will be rare and the design will handle this. The reason for this alignment is to help minimize reservations being broken up when storage is returned and a cluster is placed back into the free tree.

Reserved storage will be aligned based on the passed in policy. If a request for storage does not find any reserved storage, then storage will be reserved for this file and aligned according to the policy. For NFS server allocations for example, storage will be aligned to the file offset that corresponds to the size of the request if the request falls within the size being reserved. If the storage can not be aligned such that the entire request is encompassed, then the reservation will be aligned with the file offset of the request and the storage removed from the beginning of the reservation. This will of course always be rounded to the allocation unit.

For example, an NFS server request for storage at offset 32K in a file for size 16K that has no currently reserved storage for this range in the file. A 1MB chunk will be granted. This will be aligned with the nearest 1MB boundary that still encompasses the request. In this case offset 0 of the file. The request will then remove 16K from 32K into this reservation. The reservation tree will now contain a piece of storage of size 32K at disk block X and a piece of storage of size 1MB-16K-32K at disk block X+32K+16K. Now any request for this file in that range will find storage that is disk block contiguous to the original request.

2.3.6 Storage Allocation Policies

The storage allocator will accept as an argument the policy with which to allocate storage. This policy will determine the different characteristics with which storage is obtained. Multiple policies can be ORed together in some cases. There will be 2 main policies used in this design. OFFSET_RESERVATION and PACK_RESERVATION.

The OFFSET_RESERVATION policy will be used for requestors that exhibit out of order write behavior. This is NFS servers and CFS demons. This policy will attempt to allocate storage based on the file offset being requested. It will try to offset into the reservation such that other requests coming in out of order will find reserved storage that is contiguous.

The PACK_RESERVATION will be the default allocation policy. This will be used in conjunction with a new field in the access structure, lastAllocBlk. The last allocated disk block will be kept track off for every file. The next request for storage, regardless of file offset will attempt to allocate this disk block. This will provide sequential writers with contiguous allocations. This will also avoid the writers that are random or have strided patterns from fragmenting the disk.

- OFFSET_RESERVATION – This policy means that the storage given out should be taken from the beginning of the reservation offset by the specified amount. The caller will ask for a specific disk block and if this policy is set that disk block will be given back or if that disk block can not be found, then a new piece of storage will be reserved and the allocation will be removed from the specified offset into this reservation. This policy will be used for CFS and NFS servers.
- PACK_RESERVATION – This policy means that storage from the beginning of the reservation will be handed out. If the passed in disk block falls within this reservation the allocation will be given out starting at the beginning of the reservation. This will be the default policy.
- STATIC_RESERVATION – This policy will not be fully implemented but the infrastructure in place. This policy will keep a reservation around that will be used for files that don't need reservations. Files will be packed into this reservation until it fills up and then a new STATIC_RESERVATION cluster will be started. This could be used by VFAST if it determines a given file is not likely to have more allocations done to it. It could migrate the file over to this type of storage.
- LOOKUP_AND_SOFTHOLD– This policy is used to lookup free storage and put a hold on it for a future call to allocate it. This will be used by vfast to set aside a range of free_space that it will attempt to make larger by migrating adjacent storage out of the way. This will be a soft hold in the sense that other requests for storage will avoid this range until it is the only storage available at which time it will be stolen.
- PREFERED_ALLOCATION – This policy can be used when the allocation unit for user data is larger then the metadata allocation unit. The allocator will attempt to only hand out storage in allocation unit increments but may hand it out in less rather than returning ENOSPC. This policy will not be implemented.
- PICKY_ALLOCATION – This policy says to search more aggressively for a contiguous range of free space. This will return either the requested amount or the largest contiguous range that it could find. This will be a more expensive policy since the b-trees may need to be exhaustively searched.
- SOFTHOLD_ALLOCATION – This policy is used to obtain the storage that was looked up previously using LOOKUP_AND_SOFTHOLD.
- STORAGE_TREE_SPLIT1 – This policy is used exclusively when allocating space for the storage b-trees themselves.
- STORAGE_TREE_SPLIT2 – This policy is used exclusively when allocation space for the storage b-trees themselves and indicates that we need to avoid anymore splitting.
- FORFEIT_RESERVATION – We are deleting this storage and are willing to give up the reservation.
- ALLOW_OVERLAP – This is an existing flag that will be converted over to be included in the storage flags.

2.3.7 Finding reserved storage

A request will attempt to locate reserved storage by searching for storage at a specific disk block. Since storage is given out in clusters, a disk block will only be presented when the distance to the closest piece of storage already owned by this file is less than the cluster size. Other wise unreserved space will be handed out (which will also be looked up by disk block).

Reserved storage will be tracked in a b-tree that is ordered by disk block. A requested disk block will be searched for in the soft reservation tree. If a piece of storage is found that includes this disk block AND the tag associated with this reserved block matches the tag of the requestor, then the amount requested will be removed from this reserved piece of storage. If any storage remains, it will be put back in the reservation tree. If the request is for more than the reservation then only this amount will be returned.

2.4 Other Designs considered

Many other designs were considered for this. A bitmap approach was discarded because it was not scalable. The bitmap maps both free and used storage and is therefore difficult to quickly locate free storage without linearly searching thru the bitmap. Although a single bit is very cost effective for representing storage, we wanted to be able to potentially associate more information with a given range of storage for future flexibility.

A combination of b-tree and hash table was also considered but a hash table does not lend itself as an on disk structure very easily, plus and more importantly the hash table can only be looked up with a key where as the b plus tree can be looked up by offset for a specific element and the leaf nodes can be traversed for a candidate size.

Another approach that was considered was to create and in-memory b-tree to map space by size. This would allow for fast in-memory lookups and alleviate the need for the b-tree to be under transaction control. This was attractive however the initial creation of this b-tree at mount time could require lots of I/O and lots of memory for each domain in the system. For these reasons it was felt that an on-disk structure that was on demand was a better candidate.

Having a b-tree map reserved space and a mega-bitmap map free space was considered. The bitmap would only map in cluster sizes. The problem here was that once again we would have a linear search to find free space and could not tolerate an unaligned free chunk of storage.

Having the free space tree map by size rather than offset was also considered, this would allow the largest piece to always be given out when free space was available. This too was discarded since the minimum space in the b-tree was 1Mb anyway and this is considered the cluster size. The main drawback about order by size is that it is difficult to glom pieces of storage together since offset adjacent pieces are not necessarily adjacent in the b-tree.

2.5 Design Chosen

The design chosen is to use 2 b-plus-trees to manage the free disk space. One b-tree will map the free space and it will be searchable by disk block offset. It will contain pieces of space that are a cluster size or greater (currently a cluster is 1Mb) and in cluster size increments (except in some rare cases).

The second b-tree will map reserved free space by offset. This tree will contain pieces of storage that are less than a cluster size (except in some rare cases). This tree will be used to check if a desired disk block is available.

Each b-tree is mutually exclusive. Available space will either be in the free b-tree or the reserved b-tree but will not be mapped by both. Used space will not be mapped at all by the storage subsystem. This approach minimizes the number of b-tree operations that need to be performed for a single storage allocation request.

The free space tree will hand out space in 1Mb pieces, once this is exhausted all requests will need to go to the reserved tree. At this point however the storage has been evenly distributed amongst all existing files. Now a new storage request will generate a random offset and go to that node in the b-tree. This node will now be searched for the largest available piece of space. This space will then be broken up leaving some for original reserver and some for the current requestor.

It is thought that since the reserved tree does not contain a piece of storage larger that 1Mb and the storage has been spread across the disk, it is not vital that the largest available piece be handed out. By searching a b-tree node which should by design be half full there will be roughly 200 choices. If nothing is large enough to satisfy the request then the largest piece available will be handed out causing fragmentation of the file to occur. This is life at the races. A trade-off is being made here to spread allocations across the disk to increase the odds that extension of existing storage can be made into contiguous space. In order to achieve this space is actually being fragmented from the point of view of the entire disk. As storage is used up, the likelihood of fragmenting increases. This is true in any storage allocation scheme.

So the flexibility of b-trees will allow for the application of policies to the storage allocation in the future. As we run tests and monitor performance, we should be able to tune the storage subsystem. This design will keep in mind some of the future interfaces that may be need.

2.6 Major Modules

Storage allocation will be broken into a few major modules. The storage allocator itself will accept inputs and will return one contiguous chunk of storage based on the inputs. Certain rules will be followed by the allocator for choosing the storage. The inputs will help to influence the allocator but basically all requests will be funneled to the allocator.

The layer above the allocator will decide the policy. Policy will be driven by the requestor. The disk block number will be chosen based on this policy and in some cases cause the policy to be changed. Other than the policy choice, the rest of the allocation decisions will be made in the allocation layer.

The manipulation of the b-trees will use the existing index directory routines. Copies will be made of these routines and modifications made to tailor them to the storage trees. The infrastructure will remain basically the same. The transaction and storage allocation will remain the same. This document will not discuss how the b-tree will be internally managed. It will detail the modifications that need to be made to the existing b-tree routines.

2.7 Major Data Structures

A few new data structures and some new fields to existing data structures will be introduced. The SBM will be replaced by the storage map. Externally the storage map will look much like the SBM. It will have an access structure and be treated as a reserved meta-data file.

2.8 Design Considerations

3 Detailed Design

3.1 Data Structure Design

3.1.1 struct bfAccess

```
struct bfAccess {
    dyn_hashlinks_w_keyT hashlinks; /* dynamic hash table 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 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 directI/O */
    stateLkT stateLk;                 /* state field */

    struct vnode bfVnode;             /* The vnode for this file.  */

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

    bf_vd_blk_t bfaLastAllocBlk          /* The diskblock last allocated */

}
```

3.1.2 struct vd

```
typedef struct vd {
    ssVolInfoT ssVolInfo;      /* smartstore frag and free lists */
    /*
     ** Static fields (ie - they are set once and never changed).
     */
    ~~bf_vd_blk_t stgCluster;     /* num DEV_BSIZE blks each stg bitmap bit */~~
    struct vnode *devVp;        /* device access (temp vnode *) */
    uint32_t vdMagic;           /* magic number: structure validation */
    bfAccessT *rbmtp;           /* access structure pointer for RBMT */
    bfAccessT *bmtp;            /* access structure pointer for BMT */
    ~~bfAccessT *sbmp;            /* access structure pointer for SBM */~~
    bfAccessT *stgp;            /* access structure for storage map */
    domainT *dmnP;              /* domain pointer for ds */
    vdIndexT vdIndex;           /* 1-based virtual disk index */
    bs_meta_page_t bmtXtntPgs; /* number of pages per BMT extent */
    char vdName [BS_VD_NAME_SZ]; /* temp - should be global name */

        /*
     * The following fields are protected by the vdScLock semaphore
     * in the domain structure.  This lock is protected by the
     * domain mutex.  Use the macros VD_SC_LOCK and VD_SC_UNLOCK.
     */
    bf_vd_blk_t vdSize;         /* count of vdSectorSize blocks in vd */
    int vdSectorSize;           /* Sector size, in bytes, normally DEV_BSIZE */
    ~~bf_vd_blk_t vdClusters;     /* num clusters in vd (num bits in SBM) */~~
    bf_vd_blk_t vdClusterSz;    /* Size of reservation cluster */
    stgRecT stgBtreeParams;     /* Location of b-trees */
    serviceClassT serviceClass; /* service class provided */

    ftxLkT mcell_lk;            /* used with domain mutex */
    bs_meta_page_t nextMcellPg; /* next available metadata cell's page num */
    ftxLkT rbmt_mcell_lk;       /* This lock protects Mcell allocation from
                                 * the rbmt Mcell pool.  This pool is used
                                 * to extend reserved bitfiles.
                                 */
    bs_meta_page_t lastRbmtPg; /* last available reserved mcell's page num */
    int rbmtFlags;              /* protected by rbmt_mcell_lk */

    ftxLkT stgMap_lk;           /* used with domain mutex */
    ~~stgDescT *freeStgLst;       /* ptr to list of free storage descriptors */~~
    ~~uint32_t numFreeDesc;       /* number of free storage descriptors in list */~~
    ~~bf_vd_blk_t freeClust;      /* total num free clusters (free bits in sbm) */~~
    bf_vd_blk_t freeBlks;      /* total num free blocks */````````````
    ~~bf_vd_blk_t scanStartClust; /* cluster where next bitmap scan will start */~~
    ~~bs_meta_page_t bitMapPgs;   /* number of pages in bitmap */~~
    ~~uint32_t spaceReturned;     /* space has been returned */~~
    stgDescT *fill1;            /* ptr to list of reserved storage descriptors */
    stgDescT *fill3;            /* ptr to list of free, reserved stg descs */
    uint32_t fill4;             /* # of free, reserved stg descriptors in list */

    ftxLkT del_list_lk;         /* protects global □ransiti delete list */

    ADVRWL_DDLACTIVE_T ddlActiveLk; /* Synchs processing of deferred-delete */
                                    /* list entries */
                                /* used with domain mutex */

    bfMCIdT ddlActiveWaitMCId; /* If non-nil, a thread is waiting on entry */
                                /* Use domain mutex for synchronization */
    cv_t ddlActiveWaitCv;       /* Used when waiting for active ddl entry */

    struct dStat dStat;         /* collect device statistics */
    uint64_t vdIoOut;           /* There are outstanding I/Os on this vd */
    uint32_t vdRetryCount;      /* count of AdvFS initiated retries */
    uint32_t rdmaxio;           /* max read IO transfer byte size */
    uint32_t wrmaxio;           /* max write IO transfer byte size */
    uint32_t max_iosize_rd;      /*Disk device max IO transfer byte size */
    uint32_t max_iosize_wr;      /*Disk device max IO transfer byte size */
    uint32_t preferred_iosize_rd; /*Driver's preferred IO transfer byte size */
    uint32_t preferred_iosize_wr; /*Driver's preferred IO transfer byte size */

    ~~stgDescT freeRsvdStg;      /* desc for free rsvd stg for rsvd files */~~
    ~~stgDescT freeMigRsvdStg;   /* desc for free rsvd stg for migrating files */~~
    stgDescT freeSoftHoldStg; /* Range of storage currently on hold */
    stgOrphanList *stgOrphanList;   /* Storage that could not fit in b-trees */
    bf_vd_blk_t rand_blk_seed; /* seed for random block generator */
#   ifdef ADVFS_VD_TRACE
        uint32_t trace_ptr;
        vdTraceElmtT trace_buf [VD_TRACE_HISTORY];
#   endive
#   ifdef ADVFS_SS_TRACE
        uint32_t ss_trace_ptr;
        ssTraceElmtT ss_trace_buf [SS_TRACE_HISTORY];
#   endif
} vdT;
```

3.1.3 Struct stgBmtRec

```
typedef struct stgBmtRec
{
    bs_meta_page_t rsvd_page;
    bs_meta_page_t free_page;
    uint32_t rsvd_levels;
    uint32_t free_levels;
    bs_vd_blk_t static_file_cluster;
    bs_vd_blk_t orphan_list_head;

} stgBmtRecT;
```

3.1.4 Struct stgRec

```
typedef struct stgRec
{
    stgBmtRecT bmt;
    uint64_t prune_key_free;
    uint64_t prune_key_rsvd;
}stgRecT;
```

3.1.5 struct bsBfAttr

```
typedef struct bsBfAttr {
    bf_fob_t bfPgSz;              /* Bitfile area page size in 1k fobs  */
    ftxIdT transitioned;          /* ftxId when ds state is ambiguous */
    bfStatesT state;              /* bitfile state of existence */
    serviceClassT reqServices;
#ifdef SNAPSHOTS
    /* The following snapshot fields may be removed, modified, or relocated
     * depending on the design for snapshot support in HP-UX.
     */
    uint32_t snapShotId;
    uint32_t snapShotCnt;
    uint32_t maxSnapShotPgs;
#endif  /* SNAPSHOTS */
     bf_vd_blk_t lastAllocBlk     /* Last allocation made to file */

};
```

3.1.6 struct stgBtreeNode

```
typedef struct stgBtreeNode
{
   uint32_t total_elements;
   uint32_t reserved;
   bs_meta_page_t page_left;       /* Page number of left sibling node. */
   bs_meta_page_t page_right;
   stgBtreeNodeEntryT data[1];  /* Keep compiler happy */
```

3.1.7 struct stgBtreeNodeEntry

```
typedef struct idxNodeEntry
{
    uint64_t search_key;
    uint64_t free_size;
    bfTagT tag;
}idxNodeEntryT;
```

3.1.8 struct stgOrphanList

```
typedef struct stgOrphanList
{
   bfMCIdT mcellId;
   struct stgOrphanList *next;
}
```

3.1.9 typedef enum

```
{
  RESERVED_SPACE_TREE,
  FREE_SPACE_TREE
} stgMapT;
```

3.1.10 typedef enum

```
{


OFFSET_RESERVATION =      0x1,
PACK_RESERVATION =        0x2,
STATIC_RESERVATION =      0x4,
LOOKUP_AND_SOFTHOLD =     0x8,
PREFERED_ALLOCATION =     0x10,
PICKY_ALLOCATION =        0x20,
SOFTHOLD_ALLOCATION =     0x40,
STORAGE_TREE_SPLIT1 =     0x80,
STORAGE_TREE_SPLIT2 =     0x100,
FORFEIT_RESERVATION =     0x200,
ALLOW_OVERLAP =           0x400
} stgFlagsT;
```

3.1.11 #define STG_MAX_BTREE_LEVELS 5 /* ~300**5 entries */

3.1.12 #define STG_CLUSTER_SIZE 0x1000000/DEV_BSIZE

3.2 Module Design

This section will be divided into high level modules required for the new storage allocation.

3.2.1 Initializing the b-trees

At domain creation we do not have a log so we need to create the b-trees and account for any storage being used by metadata that is created at this time. Two meta-data pages will be needed for the b-trees. One for the reserved tree and one for the free tree. The storage will be handed out such that the storage b-trees have their own storage cluster (and therefore reserved space) and the BMT has its own storage cluster (and therefore reserved space). Both of these clusters will be obtained from the middle of this disk in an attempt to minimize seeking time. Each page of the b-tree will be manufactured to be the root node of the tree. The free space tree will be missing 3 storage clusters. The reserved tree will have the remainder of the BMT and storage b-tree clusters in it.

The concept of preallocated space is no longer needed. This new storage allocation/reservation design should alleviate the need to preallocate or explicitly reserve space for the BMT. If we find that we need this we can easily implement this in the allocation code by avoiding the BMT's tag when allocating.

3.2.1.1 Bs_disk_init

3.2.1.1.1 Interface

```
StatusT
bs_disk_init(
          char *diskName,              /* in - disk device name */
          bfDmnParamsT* bfDmnParams,   /* in - domain parameters */
          bsVdParamsT *bsVdParams,     /* in/out - vd parameters */
          serviceClassT tagSvc,        /* in - tag service class */
          serviceClassT logSvc,        /* in - log service class */
          ~~bf_fob_t bmtPreallocFobs,     /* in - fobs to be preallocated */~~
          onDiskVersionT  dmnVersion   /* in - version to store on-disk */
          )
```

3.2.1.1.2 Description

This will be modified to remove all SBM code and replace with the storage tree code.

3.2.1.1.3 Execution Flow

- Remove code relating to the SBM.
- Calculate where the two pages of the storage trees will go (in the middle of the disk as the SBM used to be).
- Call advfs_stg_init_trees to indicate the storage that is being used.
- Remove the BMT preallocation code.

3.2.1.2 advfs_stg_init_trees

3.2.1.2.1 Interface

```
static statusT
init_sbm(
    struct vnode *vp,               /* in */
    struct bsMPg *bmtpg,            /* in */
    struct bsVdAttr *vdattr,        /* in */
    bfFsCookieT dmnCookie,          /* in */
    bf_vd_blk_t rbmtBlks,           /* in */
    bf_vd_blk_t bmtBlks,            /* in */
~~    bf_vd_blk_t preallocBlks,       /* in */~~
    bf_vd_blk_t bootBlks,           /* in */
    bf_vd_blk_t tagBlks,            /* in */
    bf_vd_blk_t StgFirstBlk,        /* in */
    bf_vd_blk_t *freeBlks           /* out */
    )
```

3.2.1.2.2 Description

This routine will initialize the storage b-trees to map all the free space on the disk minus the space that is being used for the meta-data files that are being setup for domain creation. The storage for the b-tree's themselves also needs to be accounted for.

- Stg_blk = 2 * ADVFS_METADATA_PGSZ_IN_FOBS) / ADVFS_FOBS_PER_DEV_BSIZE,
- First_stg_chunk = MSFS_RESERVED_BLKS + CPDA_RESERVED_BLKS + rbmtBlks + bootBlks + tagBlks;
- First_stg_chunk_start = 0;
- Second_stg_chunk = bmt_blks;
- Second_stg_chunk_start = StgFirstBlk
- Third_stg_chunk = stg_blks;
- Third_stg_chunk_start = Second_stg_chunk_start + Cluster_size;
- Allocate memory for the free space root node
- Initialize the header
- Calculate the 3 chunks of FREE space and enter them into the node
- Allocate memory for the reserved space root node
- Initialize the header

- Calculate the 3 chunks of reserved space remaining in the 3 clusters that were handed out and enter them into the node.
- Write these two pages to disk using advfs_raw_io.
- Initialize the xtnt record in the passed in Mcell page.
- Initialize the StgBtree record in the passed in Mcell page marking the location if the two root nodes and levels.

3.2.2 The storage allocation layer

As mentioned above the storage allocator will provide an interface that will, based on its inputs, return a single contiguous chunk of storage. This will be made up of multiple underlying components that will be called from the single allocator interface.

Callers of stg_add_stg will all have their arguments modified to match stg_add_stg and their callers and their callers … when necessary. In the interest of clarity I will not include all of these here.

3.2.2.1 Stg_add_stg

3.2.2.1.1 Interface

```
statusT
stg_add_stg (
             ftxHT parentFtx,             /* in */
             bfAccessT *bfap,             /* in */
             bf_fob_t fob_offset,    /* in */
             bf_fob_t fob_cnt,       /* in */
             ~~int allowOverlapFlag     /* in */~~
             stgFlagsT storageFlags,        /* in */
             bf_fob_t *alloc_fob_cnt /* out */
             )
```

3.2.2.1.2 Description

This upper level storage allocation routine will not need to change very much. New flags will be passed thru to the storage allocator and the call to add_rsvd_stg will be removed.

3.2.2.2 Alloc_from_one_disk

3.2.2.2.1 Interface

```
statusT
alloc_from_one_disk (
             ftxHT parentFtx,             /* in */
             bfAccessT *bfap,             /* in */
             bf_fob_t fob_offset,    /* in */
             bf_fob_t fob_cnt,       /* in */
             ~~int allowOverlapFlag     /* in */~~
             stgFlagsT storageFlags,        /* in */
             bf_fob_t *alloc_fob_cnt /* out */
             )
```

3.2.2.2.2 Description

We can remove the special code for NFS_SERVER and instead set up the storage allocation policy to be OFFSET_RESERVATION for the NFS_SERVER and CFS demon cases. The code for calculating the block to ask for can all be removed. This will now be handled in the storage allocation code.

3.2.2.3 Stg_alloc_one_xtnt

3.2.2.3.1 Interface

```
Stg_alloc_one_xtnt (
        bfAccessT *bfap,          /* in  find stg for this file */
        vdT *vdp,                 /* in  this volume */
        bf_fob_t fob_cnt,        /* in Requested length */
        bf_vd_blk_t dstBlkOffset, /* in, chkd if policy == SOFTHOLD_ALLOCATION */
        ~~bsAllocHintT alloc_hint,  /* in */~~
        stgFlagsT storage_flags,
        ftxHT ftxH,               /* in */
        bf_fob_t *alloc_fob_cnt_p,   /* out Got this much stg. */
        bf_vd_blk_t *alloc_vd_blk_p,         /* in/out at this location.*/
        ~~Xint *pinCntp              /* in/out this many pins left */~~

    )
```

3.2.2.3.1.1 Description

This routine will no longer interface to the SBM. The old code would first find space and then remove the space from the SBM after reducing the size of the request due to log half full reasons. This is no longer necessary since the size of the request does not cause any more records to be pinned (unlike SBM bits).

If no policy is present this routine will decide the policy to pass to the storage allocation code. This will be based currently on the passed in NFS hint (BFS_ALLOC_NFS). This will cause the policy to be OFFSET_RESERVATION other wise it will set PACK_RESERVATION.

The disk block to attempt to allocate at will also be decided by this routine except when passed in (which is currently only VFAST). This too will be policy driven. If no storage is available to use as a disk block then a pseudo random number generator will be called to generate an offset that is between 0 and the last vd disk block. If the policy if OFFSET_RESERVATION then the extent map will be examined and if the nearest extent offset is within 1 cluster of the requested offset, the disk block of the extent map will be modified by the difference in offsets and this disk block will be used as the target disk block. If the policy is PACK_RESERVATION then the last allocated block from the access structure will be used as the target block (incremented by the allocation unit).

Once storage is obtained the last allocated field in the access structure will be updated with the start of last allocation unit of the storage returned. This will be updated for either storage policy. This will also be updated under transaction control in the primary Mcell.

It should no longer be necessary to reserve space for the BMT. The reservation scheme for all files should be sufficient. We may consider adding a priority to reservations so that the BMT can be scavenged last.

3.2.2.3.1.2 Execution Flow

- If storage_flags != STORAGE_TREE_SPLIT
    - Lock the storage map lock
- If hint == BFS_ALLOC_NFS or CFS_CLIENT
    - Storage_policy = OFFSET_RESERVATION
- Else
    - Storage_policy = PACK_RESERVATION
- Call advfs_get_space with the storage policy.
- If no storage was found
    - If storage_flags != STORAGE_TREE_SPLIT

        - Unlock the storage map
    - Return ENOSPC
- If storage_flags != STORAGE_TREE_SPLIT
    - Unlock the storage map.
- Return.

3.2.2.4 Advfs_get_space

3.2.2.4.1 Interface

statusT advfs_get_space (

```
vdT *vdp,                         /* The virtual disk to obtain space from */
bfAccessT *bfap,                  /* The file requesting storage */
bf_vd_blk_t *requested_blocks,    /* IN: The desired number of DEV_BSIZE blocks */
                                 /* OUT: The actual number of DEV_BSIZE blocks */
bf_vd_blk_t minimum_blocks,      /* The allocation unit. Storage in these units*/
offset_t file_offset,            /* Hint for aligning within cluster */
bf_vd_blk_t *disk_block,         /* IN: The desired starting VD disk block. */
                                 /* OUT: The actual starting VD disk block */
stg_flags  flags,                /* The allocation policy we would follow*/
FtxH ftxh)                     /* Transaction handle */
```

3.2.2.4.2 Description

This routine is the highest level storage allocation interface for requesting storage. It is the responsibility of the caller to select the policy. This routine will understand the passed in storage policy and attempt to obtain storage in a number of steps that will vary slightly based on the passed in policy. Depending on the policy the caller may pass in a target diskblock or this routine will pick the diskblock.

The caller expects to be handed back a single contiguous range of storage. This range may be less than the requested size for a variety of reasons. It is the callers' responsibility to call back into this routine for the next chunk. This behavior is not being changed.

The returned size may also be adjusted to match the passed in minimum blocks. The minimum block refers to the allocation unit for the file having storage added to. Storage will not be handed out that is not in units of this value. This means that if storage is available but no contiguous chunks can be found of this minimum size then no space will be returned. This situation could occur if the user data allocation unit is less than the meta-data allocation unit. We could have very fragmented space that would allow us to satisfy a user data request but not a meta-data request.

This routine will take advantage of the calling code's ability to handle glomming in-memory extents together that are actually adjacent on-disk. In other words, a request could find some reserved storage that it will return the caller and then the next call in could find storage that was actually adjacent to the previous request. This will simplify the searching greatly and should be infrequent.

A passed in disk block will indicate to this routine that it should first look to the reserved storage b-tree to see if it can locate space at this starting disk block. The caller has decided that placement of storage at this disk block will produce the optimal storage layout for this file. If the disk block is -1 this will indicate that no reserved storage is desired and space should be obtained from the free space b-tree if possible.

The passed in tag may be used (this is policy dependent) to determine if the reserved space we have looked up actually belongs to the requesting file. Regardless of policy, the tag will be stored with any space that is placed in the reserved b-tree.

If no reserved space can be found at the passed in disk block (or if none is requested), the free space b-tree will be searched next to see if there is any available space. The disk block will be used to index into the tree in an attempt to allocate storage once again at the desired disk block. The passed in file offset may be used (policy dependent) to determine which part of this reserved cluster, the storage to satisfy the request should actually be carved out of. If the file offset is to be used, the cluster will be aligned on the nearest file offset that is a multiple of the cluster size. The storage will then be carved out of this chunk relative to this offset and the passed in file offset. This will have the effect of allowing random/out-of-order writes to fall into contiguous storage.

If neither reserved or free storage can be found, then the reserved pool of storage will need to be scavenged. This will be done in a somewhat in-precise fashion that will attempt to reduce existing reservations without eliminating them. The original node that we went to look for the initial disk block will be revisited (optimizations will be added to avoid a re-walk of the b-tree) or a random offset will be generated. Once on a leaf node of the reservation tree, we will now search this node for the largest piece of space. The design choice here is that the absence of space in the free tree means that there exists no piece of space in the reservation tree that is larger than 1 cluster in size. Exhaustively searching this tree for the largest space is too expensive and will not produce space larger than 1 cluster anyway. So by limiting the search to the largest piece on a leaf node we will minimize anymore i/o's and have roughly 200 pieces to choose from (this is a balanced b-tree so nodes should be at least half full).

Once the largest piece of storage in the leaf node is located, we will divide it in half. The requestor will take the 2nd half leaving the reservations owner with half of its reservation. This half will then have alignment/policy decisions applied to it as described for free space. If this half is not enough to satisfy the request, the caller will have to call back in.

This routine will not return space that is less than the passed in minimum block size and space that is returned will be rounded down to the minimum block size as well.

Much of the b-tree management will not be covered in this description. The directory index code will be leveraged and very little of the b-tree management aspects will be modified and therefor discussed. This includes the transaction pieces. A transaction will be started across any call to advfs_get_space_int and done before calling out to any routines like advfs_put_space.

If at any point we find that we do not have enough space to split b-tree nodes then the error ENOSPACE will be returned to the caller. This could mean that the free blocks indicate that the callers request could have been satisfied. This also means that the next request for the same amount of space could actually succeed if it didn’t cause the b-tree to split. Keep in mind however that this situation can ONLY occur if we happen to have mixed block sizes where the metadata size is larger than user data size. This is also true for any metadata file that needs to grow. In a non-mixed environment the b-tree can’t need to split since it only maps free space!

Vfast will have its own storage policies that will be supported by this routine. Vfast can put a soft hold on a range of storage that it deemed a candidate for migrating to. This determination will be done external to this routine. A flag to this routine however will override this soft hold and give out this space to the caller. This will be the SOFTHOLD_ALLOCATION flag and will be used by vfast. All allocators will avoid this range when allocating new storage unless there is no storage left in which case they may steal it.

The STATIC_ALLOCATION policy allows for multiple files storage to be packed into a single reservation. This is for files that are deemed to not grow and this policy should only be used by Vfast. The offset of the next free piece of space in this reserved static space will be kept on-disk and in the vd structure. This routine will remove the space that is needed and update the offset. If the space is used up, then the offset will be set to -1 and the next call in will cause a new piece of static reserved space to be setup.

The OFFSET_RESERVATION allocator has a tendency to take storage from the middle of a reservation. This has the problem of placing two pieces of storage into the reservation tree which may potentially glom with another file’s reservation. The PACK_RESRVATION allocators don’t have this problem since they only have a single reservation that would never glom with another similar allocator. So the OFFSET_RESERVATION will not mark storage as theirs when glomming but they will be allowed to steal from other file’s reservations. This should hopefully even the score a little while favoring the most recent allocator.

3.2.2.4.3 Execution Flow

- `Round up request to allocation units and minimum blocks`
- `If policy & STATIC_RESERVATION`
    - `get the disk block from the vd structure.`
    - `If target disk block is -1`
        - `generate a random number between 0 and the last available disk block and use that as the diskblock`
- `If policy & OFFSET_RESERVATION`
    - `call the advfs_stg_get_nearest_diskblock to obtain the desired disk block for allocation.`
    - `If the nearest disk block is more than a cluster away. This can cause too much fragmentation.`
        - `Change the policy to PACK_RESERVATION`
- `If policy & PACK_RESERVATION`
    - `Obtain the target disk block from the bfap->lastAllocBlk`
- `If policy & LOOKUP_AND_SOFTHOLD`
    - `Call advfs_scan_trees for the specified size.`
    - `If a range is obtained`
        - `Set the offset and length in the soft hold fields of the VD structure.`
- `If policy & PICKY_ALLOCATION`
    - `Call advfs_find_space`
    - `If an allocation is located`
        - `Call advfs_get_space_int (btree, node) with the b-tree and node passed back from the lookup.`
        - `If storage was obtained return it`
        - `Else return ENOSPC.`
- `If offset is zero goto FREE SPACE TREE`
- `Call stg_find_node with reserved_tree root node and the offset obtained above.`
- `Search the entries for one that encompasses this disk block.`
- `If any entry is found and there is not a soft hold on this range (compare to the range stored in the VD – Should we steal this if it belongs to us?) and the size of the entry is at least the minimum allocation unit for this request.`
    - `If policy is PACK_RESERVATION`
        - `If the entry's tag matches`
            - `Set the offset equal to the offset of this entry (which it should be anyway).`
            - `Deref this page.`
            - `Call advdfs_get_storage_int the offset and min (requested_size, entry_size) passing in the node_page as a hint if it can be used` 1[1].
            - `If more space is returned than was requested then this means that we removed the last entry in the node and need to insert more space back. We can not allow the last entry in a node to have its offset increase without updating the parent so we must manually reinsert the rest back. We know this will fit in the node since it was the last piece of the node and we removed it.`
                - `Call advfs_put_space for the amount of space we are not going to use.`
            - `Update bfap->lastAllocBlk`

[1] The hint can be used if the element being inserted will not be the last element in the node or cause the node to split. Other wise a full walk of the b-tree is needed in order to update parent nodes.

        - Return this storage to the caller.
  - If policy is OFFSET_RESERVATION
    - Deref this page.
    - Call advdfs_get_storage_int with the offset and min (requested_size, entry_size) passing in the node_page as a hint if it can be used 1[2].
    - If more storage is returned than was requested
      - Call advfs_put_space with the offset and size of the remainder of the storage for the reservation tree with the FORFEIT_RESERVATION flag.
      - If we get an ENOSPC error back return all the space and set the policy to PACK_RESERVATION and start the search over. This will insure that we do not need to reinsert into the reservation tree.
    - Update bfap->lastAllocBlk
    - Return the space to the caller.
  - If policy is STATIC_RESERVATION
    - Call advfs_get_space for this offset and size.
    - Update the block offset in the vd reservation to be the offset + size of the space found.
    - Update bfap->lastAllocBlk
    - Update the VD record.
    - Return the space found.
- We were unable to find reserved space.
- If we have a page from above deref it now.
- FREE SPACE TREE – We either could not find the space we wanted in the reserved tree or the file is empty and this is the first allocation.
- If the offset is zero
  - Generate a random number between 0 and the last available disk block, rounded to a cluster size boundary and use that as the diskblock offset.
- Call stg_find_node with the offset calculated above.
- Search the node for the first entry that encompasses this disk block offset.
- If no entry encompasses the passed in disk block
  - Then locate the either the largest entry on the node or the first entry that satisfies the request.
- If there is no storage at all in the free tree (root node is empty)
  - Go to SCAVANGE RESERVESD TREE.
- Adjust the offset of the request down to be cluster aligned (unless this causes us to fall out of the entry’s range)
- Adjust the size of the request to be roundup(min(request_size,entry_size),cluster_size);
- If the resulting size happens to exceed the entry_size (in rare cases the entry size may not be a multiple of the cluster size) then choose the entry size.
- If the size that we would leave behind in the free space tree is less than a cluster then give this to the requestor.
- If the size is not the minimum allocation unit for this request or it has a soft hold on it then move to the next entry in the node. We could conceivable have to search all nodes in the tree (wrapping back around). Steal the soft hold piece if necessary. If we can’t find anything go to SCAVANGE RESERVED TREE.
- Deref the node page.
- Call advfs_get_space_int (free_btree) for the beginning offset and size from above.
- If the returned size is > requested_size

- If policy is PACK_RESERVATION or STATIC_RESERVATION
  - Call advfs_put_space (reserved_tree, tag) to place the remainder of the storage back into reserved tree for this file.
  - If policy is STATIC_RESERVATION
    - Update the block offset in the reservation to be the offset + size of the space found
  - Update bfap->lastAllocBlk
  - Return space to the caller.
- If policy is OFFSET_RESERVATION
  - If there is storage before the requested offset
    - Call advfs_put_space setting the FORFEIT_RESERVATION flag
  - If there is storage after the requested size
    - Call advfs_put_space setting the FORFEIT_RESERVATION flag
    - If this returns ENOSPC (Only on a second insertion could this happen)
      - Call advfs_get_space for the amount we just placed in the tree
      - Hand back to the caller space from the beginning of this chunk
      - Call advfs_put_space to place this single piece into tree. We know this will fit.
  - Update bfap->lastAllocBlk
  - Return space to the caller.

- Update bfap->lastAllocBlk
- return storage to the caller
- SCAVANGE RESERVED TREE
- If we do not still have a page reffed
  - Call stg_find_node with the offset from above
- Call stg_search_leaves(node, size)
- If no storage is found
  - Deref current node
  - return ENOSPC
- We have found space that will satisfy some or all, of our request. We are stealing another file’s reservation so we do not want to steal all of it. We will only take up to half of the reservation even if it means not satisfying the request.
- If the returned entry is greater than our minimum allocation unit
  - Set the starting diskblock offset to half of this entry.
  - Set the size to half of the entry size.
  - If policy == OFFSET_RESERVATION
    - Diskblock = diskblock + request_offset – rounddown(request_offset,size)
    - Set the FORFIET_RESERVATION flag
- Else
  - Set the offset and size to the entry.
- Call advfs_get_space (reserved_tree) for this offset and size.
- If we have any remaining size we need to put it back in the reservation tree.
  - Call advfs_put_space.
- Update the BMT record with the last diskblock in the storage that was obtained
- Update bfap->lastAllocBlk
- Check for b-tree pruning (see idx_directory_get_space).
- If storage was obtained return it to the caller.

3.2.2.5 Advfs_get_space_int

3.2.2.5.1 Interface

```
statusT advfs_get_space (
vdT *vdp,
bs_meta_page_t node_page,
uint32_t index,
bf_vd_blk_t *target_disk_block,
size_t *size,
stgFlagsT storage_flags,
ftxH ftxh
)
```

3.2.2.5.2 Description

This routine will use the b-tree routines that exist for the index directory code. Copies of the routines will be made and modified so that they are tailored to the storage trees. If it is found that there are enough similarities then this routine will be merged into a single b-tree utility routine. The majority of the b-tree processing should remain unchanged. This includes the transaction and storage allocation of the b-tree itself.

This routine expects that the offset that is being requested exists in the b-tree. The caller should have already examined the target leaf node and verified that the desired piece of storage is available. Locks will be held that prevent this node from changing prior to calling this routine.

Idx_directory_get_space_int will be leveraged. The routine will be changed to no longer search nodes for the passed in size but to go directly to the node corresponding to the passed in offset and index. It will then extract the space and pin the appropriate records as it does today.

All directory specific processing will be removed.

The caller will have specified the exact offset and size it desires. The caller will also have examined the leaf node to know if an update to the parent will need to occur and if not, the node itself will be passed in. This routine will then avoid walking the b-tree and operate only on the node passed in for efficiency. Although the code is set up for this, it should never be the case that we need to walk the b-tree. If we ever need to update a parent, a reinsertion will be done.

If possible this routine will perform reinsertions into the node as long as the node is not full and we are not increasing the offset of the last entry in a node. If either of these two conditions exists more than the request will be passed back and the caller will need to call advfs_put_space to reinsert the remainder.

The pinning of records will be modified to now include the tag as well.

This routine will never need to split the b-tree since we are only removing entries.

3.2.2.5.3 Execution Flow

```
• If a hint is passed in, pin the passed in node and skip walking the tree (This
  should always be the case)
• Else walk the b-tree until the leaf node is reached and pin it.
• Read the entry at the passed in index
• If the offset of this entry matched the passed in offset
    o If this entry is more that the requested size
        ▪ If we are the last entry in the node
            • Pin records
            • Return the entire size
        ▪ Else
            • Modify the offset and size to reflect the storage removed
              from this entry.
            • Pin records
```

```
                    •   Return requested size.
        o   Else we need to peel out the space
                ▪   If the node is at the max size
                    •   Pin records
                    •   Return the entire space
                ▪   Else
                    •   Update size of current entry
                    •   Add an entry with the offset and size that is beyond the
                        requested offset and size
                    •   Pin records
                    •   Return space
•   Else we want this entire entry
    o   Remove entry from the node
    o   Pin records
    o   Return size.
```

3.2.2.6 Advfs_find_node

3.2.2.6.1 Interface

```
advfs_find_space(
   vdT *vd;
   off_t offset_to_look_for,
   stgNodeT *ptr_to_node
   stgMapT tree)
```

3.2.2.6.2 Description

This will be called with an offset and root b-tree node to look in. It will walk the b-tree and return a pointer to the node in the b-tree that may contain the offset. It will not search the node itself to see if the offset exists. This will be derived from the existing idx_look_node_int function

3.2.2.6.3 Execution Flow

- Get the passed in trees root node from the vd
- Call bs_refpg on each node while traversing the tree and looking for the offset
- Return the node and its page number.

3.2.2.7 stg_search_leaves

3.2.2.7.1 Interface

```
stg_search_leaves (
   vdT *vd,
   stgNodeT *stg_node,
   uint32_t *index,
   stgFlagsT storage_flags,
   size_t size,
   pgRefHT *pgRefH
   )
```

3.2.2.7.2 Description

This needs to be called when there are mixed allocation units and we need to locate the best fit for a request for storage that has other than the minimum allocation unit (this currently is only meta-data). This routine will first search the passed in node for the largest piece of space on this node that is greater then or equal to the passed in size. It will also keep track of the smallest allocation that will satisfy the request. If none is found we will deref this page and move to the next sibling node. This will be repeated until either storage is found or the end of the tree is reached. We will then search in the other direction starting from the start node. If space is found we will return the index into the node of the entry containing the space.

If while searching the node we find a reservation that we already have, we will grab that one.

In the case where the largest size found on the node is the maximum minimum allocation unit and the request's minimum allocation unit is less than this then we will take the smallest entry in the node that satisfies the request. The idea here is to always try to keep at least one maximum minimum allocation unit available in each node so that for example a request to extend a directory will be able to satisfy its request without needing to search multiple leaf nodes

Note this is the routine where we can implement different policies about what storage we can steal. For example we can try to avoid stealing BMT pages here.

3.2.2.7.3 Execution Flow

- Save the prev sibling offset from the start node.
- Set direction forward
- Loop thru every entry in the node
    - keeping track of the index of the entry with the largest space
    - Keep track of the index of the smallest space that fulfills the request.
    - If this entry's tag matches our tag then steal this one.
- Do not steal space that has a soft hold on it until we are out of choices but remember the offset of the node that contains the beginning of this range.
- If space was not found
    - Deref current node
    - If direction == forward
        - Ref next sibling
    - Else ref prev sibling
    - If end of tree direction = reverse
- Keep looping
- If we found no space, try stealing the soft hold
- If the space we found is equal to the max min allocation unit and the request's min allocation unit is less than this then return the index of the smaller size to the caller.

3.2.2.8 Advfs_find_space

3.2.2.8.1 Interface

```
Advfs_find_space (
   vdT *vd,
   bs_meta_page_t *stg_node_page,
   uint32_t *index,
   size_t size,
)
```

3.2.2.8.2 Description

This routine will initially be used only by vfast (PICKY_ALLOCATION). This will lookup storage for a given size. It will aggressively search thru all nodes of the free space and reserved space trees until the desired space is found. It will start with the free tree first and then move to the reserved tree. This is i/o intensive and is not a high performance path for obtaining storage. This will not actually obtain the storage or make any modifications to the b-trees. The storage map lock must be held for the duration of this lookup.

3.2.2.8.3 Execution Flow

- Call advfs_find_node passing in the free space b-tree and an offset of zero as well as the desired size.
- Search this node until at least desired size is found
- If the size was not found
    - Deref the node
    - Ref the right sibling
- If the tree is exhausted and this not the reserved tree then repeat with the reserved tree.
- Return the index and node where the storage was found or ENOSPC.

3.2.2.9 advfs_put_space

3.2.2.9.1 Interface

```
statusT advfs_put_space (
vdT *vd,
bfAccessT *bfap,
off_t offset,
size_t size,
stgFlagsT *stg_flags,
```

3.2.2.9.2 Description

This routine is called to insert space back into the b-tree. This will be called when space is removed from a file or when space is being given out and there is remaining space to put back. As with the removal of space, the index directory b-tree routines will be modified to do the b-tree operations. This section will detail the operations that are unique to the storage trees.

The caller will not specify a btree to put the storage in. This routine will determine which tree to put it in. Calls to the lower routine advfs_put_space_int will accept a storage tree as an argument.

Generally speaking space can only be removed from a file by truncation. In other words most files will only have space removed from that last offset backwards. This is not a requirement but will lead to, especially with the default PACK_RESERVATION policy, space being glommed together as it is returned. The kernel meta-data b-trees are currently an exception to this since storage can be removed out from the middle of these files.

Unlike advfs_get_space where we will look up the node first and figure out what storage we want to ask for and then call advfs_get_space_int, here we will let advfs_put_space_int deal with most of the issues. Advfs_get_space is only operating on a single entry that may need to broken into pieces but will always live on a single node where as putting space back may involve multiple nodes that have entries glommed together. For this reason it makes more sense to move the complexity to the leaf processing.

This routine will reuse most of the code from the routine idx_directory_insert_space. The transaction/undo, pruning and splitting up to the root node will all be handled here as was in the idx_directory_insert_space routine.

Advfs_put_space_int may return a piece of storage which this routine will then call back into advfs_put_space_int in order to place it in the free space tree. The handling of ENOSPC will be discussed in the next section.

3.2.2.9.3 Execution Flow

- Obtain the root node for the reserved tree from the vd structure.
- Start a transaction.
- **TRY_AGAIN**
- Call advfs_put_space_int for the offset and size to be inserted into the reservation tree.
- If return_sts == ENOSPC we were not able to obtain storage for splitting the b-tree.
    - Check to see if space was returned for insertion into the free space tree.
        - We can use this space for the split.
        - Call advfs_put_space_int (free_tree) for the free_space that was returned to use.
        - If returned status == ENOSPC
            - We are unable to place space in either tree. Fail the transaction and return the error. This will be handled by del_dealloc_stg (see that section).
        - Goto **TRY_AGAIN**.
    - Check if the amount of storage needed to split (space_needed_to_split) the tree is available in what we are trying to put back
        - Peel out from the beginning of this piece of storage the amount of space needed to satisfy the split request.
        - Call advfs_put_space_int(free_tree) to place this amount into the free tree
        - If returned status == ENOSPC
            - We are unable to place space in either tree. Fail the transaction and return the error. This will be handled by del_dealloc_stg (see that section).
        - Goto **TRY_AGAIN** to insert the remaining amount back setting the STORAGE_TREE_SPLIT2 flag and passing the offset of the storage we just placed in the free tree in the free_space_offset argument.

- If a size and offset are returned then we must place this storage into the free tree. We should not get ENOSPC since the free tree contains chunks of large enough size to split.
    - Call advfs_putspace_int (free_tree) for this offset and size.
- Update vd->freeClust and domain->freeBlks (Note the undo will undo this).
-
- Finish the transaction.

3.2.2.10 Advfs_put_space_int

3.2.2.10.1 Interface

```
Advfs_put_space_int (
   bfTagT tag,
   Int root_node_offset,
   Off_t offset_to_insert,
        Size_t size_to_insert
        Off_t *free_space_offset,
        Size_t *free_space_size,
        Size_t *space_needed_to_split,
        stgFlagsT Storage_flags

    )
```

3.2.2.10.2 Description

This routine will walk the passed in tree by the disk block offset of the space being returned to determine if the space being returned is adjacent to any space already in the tree. This space will then be glommed together and if a cluster size that is disk block aligned on a cluster is produced then this amount will be returned to the caller (if we are not already inserting into the free space tree) for insertion into the free space tree.

This decision is based on the fact that allocations are originally given out in cluster sized chunks. These chunks by definition will start out on disk block boundaries. This is basically the unit of reservation. As time progresses and reserved space is scavenged, reservations may no longer be in cluster size chunks or aligned on cluster boundaries. So putting back space into the free tree may actually rob some one of a portion of their reservation. This seems to be acceptable since generally savaging only takes place within a cluster so for the cluster to become free means that some reservation was probably completely freed. If this were the wrong decision then the reserver will come back in with the disk block and most likely get the reservation back.

There are rare cases, when a non-aligned and less then the cluster size chunk can be put in the free space tree. The free space tree will operate correctly in these cases. There will be no alignment assumptions made, just attempts to keep the space aligned.

If glomming occurs such that the storage being added is combined with storage that precedes it then the reservation will be given to the preceding entry if a special flag is passed in (FORFEIT_RESERVATION). Otherwise the reservation will go to the tag of the file placing the storage into the tree. This flag will be used for example when deallocating since the storage will probably not be used again by this file.

If the glomming causes the space to be combined with existing reservations that are on either side of the space being returned then the preceding reservation will take over the new space. This is because PACK_RESERVATIONS start at the beginning of the reserved space so glomming to space in front of you means that either this is not a PACK_RESERVATION or all of the PACK_RESERVATION has been returned (in which case this may end up in the free space tree anyway). If it is an OFFSET_RESERVATION then any incoming request for storage will just steal the reservation back anyway.

Unlike obtaining storage, here we are giving it back and can not return an ENOSPACE error. This error can only occur if the insertion of space into the b-tree causes the b-tree to split. By definition the only way the b-tree can need to split is if there is free space. Unfortunately if we have mixed meta-data and user-data block sizes such that the user data block size is smaller then the meta-data block size we could conceivably find our self in a situation where there is only the smaller block sizes available. In these cases we will leave the space on the deferred delete list. See the section on deallocating storage for more details.

This routine will make an effort to avoid leaving storage on the deferred delete list. If we attempt to place storage into the reserved tree and we need to split the node but find that there is no space available for this (which can only happen if all the space currently in the reserved tree is of the smaller allocation unit) we will place the space in the free tree. If we find that the free space tree needs to split and we can’t get space (this is really rare since the free space tree should only contain cluster size or greater pieces) then we will leave the space on the deferred delete list.

When we find ourselves in the situation of placing space on the free tree because it wouldn’t fit in the reserved tree, we will immediately attempt to use that space if it is large enough. This is accomplished by removing from the piece we are attempting to place in the reserved tree, the amount of storage necessary to split the reserved node. We will then call back into add_stg and it will find this piece. Next we will reinsert the remainder of the space in the newly split node of the reserved tree.

We also need to avoid a cascading scenario of splitting nodes. This could happen in a mixed block size environment where the reserved tree once again encounters a situation in which there is not enough space to split a node. In this scenario we could successfully split the node using space that we obtained from the free tree. However we would then need to place the remainder of this space back into the reserved tree. The call to add_stg would not see that we actually have a piece large enough since we haven't yet put it into the reserved tree. We could then grab another piece from the free tree and this splitting scenario could theoretically ripple through the reserved tree.

In order to avoid this we will introduce two flags that will be passed to the storage allocator when splitting a node of the b-tree. The first flag STORAGE_TREE_SPLIT1 indicates this if an ENOSPC is returned from the call to add storage, and then if large enough, place the space into the free tree and try the call again. The second flag will be passed if the passed in flag is STORAGE_TREE_SPLIT1 and we need to allocate storage for another split. In this case we will pass STORAGE_TREE_SPLIT2 to indicate that if the insertion of this piece of space causes the reserved tree to split again, then before calling add_stg place the amount of space required for the split (we know at this point all the resources we need) into the free tree and then call add_stg with the STORAGE_TREE_SPLIT2 and the offset of this storage. Then add the remainder to the reserved tree.

3.2.2.10.3 Execution Flow

- Go Walk the passed in b-tree until the leaf node is reached that will potentially contain the passed in disk block.
- While walking the tree we will keep track of the number of nodes that will need to split if the node below them splits.
- Search the leaf node for the location that the insertion should take place.
- Call stg_glom_space_int to see if we can avoid an insert and glom the space into an existing entry or entries.
- If stg_glom_space_int returns and offset and length or we did not glom anything then we need to perform and insertion. This offset and length may or may not be the offset and length we started with (see the stg_glom_space_int section).
    - o Determine how many insertions we will need to make into this node. If the amount we are inserting is greater than a cluster then we could need to insert the front end and back end of this storage. Note also that we never have overlapping space so it is safe to place two entries in the node knowing that the second insertion will be less than the first entry in the next sibling node.
    - o If the node will need to split based on the number of insertions
        - ▪ If the STORAGE_TREE_SPLIT2 was passed in the use the offset passed in free_space_offset as the offset to request as well as the STORAGE_TREE_SPLIT1 flag.
        - ▪ Allocate all the storage for a full split up the tree (this is the same as in the existing index code) passing the STORAGE_TREE_SPLIT1 flag.
            - If we get the error ENOSPC
                - o Return the error to our caller and also return any space that was earmarked for the free tree and the amount of space we needed that we couldn't get.
        - ▪ Call the split routine.
    - o If the size we are to insert is larger than a cluster and we can pull out an cluster aligned piece of storage
        - ▪ Set up to return this cluster offset and length to the caller.
    - o Insert any remaining storage into the reserved node that we currently have pinned.
    - o If the RESERVE_PREVIOUS_ENTRY flag is set
        - ▪ If index > 0 then
            - store the tag in the previous entry.
            - pin the record.

        - Else
            - Pin the left sibling page.
            - Store the tag in the last entry in the node.
            - Pin the record.
- Follow the current b-tree code for updating parent nodes and pruning of the b-tree's
- Return.

3.2.2.11 Advfs_glom_space_int

3.2.2.11.1 Interface

```
 Advfs_glom_space_int (
    vdT *vd,
    bfAccessT *bfap,
    stgNodeT *node_ptr,
    bs_meta_data_pg tree,
    pgRefHT *pgref,
    Off_t *offset, /* IN/OUT offset to glom/offset to insert */
    Size_t *size,  /* IN/OUT size to glom/size to insert */
    Off_t *free_space_offset,
    Size_t free_space_size,
   Uint32_t index,
   stgFlagT storage_flag,
   ftxHT ftxH
)
```

3.2.2.11.2 Description

This routine will be based on the idx_glom_entry_int routine. It will be modified to remove all references to directories such as setting of the truncation flag and allowing entries to overlap. We will continue to panic with the all too familiar E_CANT_CLEAR_BITS_TWICE if a storage overlap is encountered.

The existing code handles the edge conditions of glomming entries that lie on different nodes and setting pruning flags. All of this functionality will remain.

Glomming will now understand a cluster size when glomming entries that live in the reservation tree. Entries that glom into a cluster or larger piece will be split up such that the cluster aligned piece can be returned to the caller. If after splitting out a cluster aligned piece, we find that an insertion is needed, this routine will return the offset and size that needs to be inserted to the caller.

When glomming storage the tag of the existing entry will not be changed if the FORFEIT_RESERVATION flag is set. Otherwise the passed in tag will always take over any glommed storage. If the passed in storage gloms both left and right and the FORFEIT_ RESERVATION flag is set then the tag to the left will be favored.

This routine will be changed slightly to no longer return the INSERT status to indicate that space needs to be inserted but rather to return in offset and size of the piece that needs to be inserted or zero to indicate that no insertion needs to be done

3.2.2.11.3 Execution Flow

- Determine the location of the left and right entries and the resulting size after the glom (this is existing code)
- If the resulting size if >=1 cluster
    - Free_space_offset = roundup(glom_offset,cluster)
    - Free_space_size = rounddown(offset+size – free_space_offset,cluster)

    - If free_space_size >= 1 cluster
        - If we glommed left and there is still an entry to the left after extracting the free space
            - Adjust the left entry's size (if needed)
            - Pin the record.
        - If we glommed right and there is still an entry to the right
            - Adjust the offset of this entry to reflect its size after the glom.
            - If this is the last entry in the node and its offset became larger
                - Set return value to UPDATE to indicate the parent node needs to change.
        - If after glomming we have to insert a piece into the tree then set the passed in offset and size to reflect this.
        - Return
- Perform any glomming
- If storage_flag != FORFEIT_STORAGE
    - Set the entry tag equal to the passed in tag.
- If glomming occurred set offset and size to zero
- Return.

3.2.2.12 stg_locate_last_used_block

3.2.2.12.1 Interface

```
Size_t Stg_locate_last_used_block (vdT vd)
```

3.2.2.12.2 Description

Examine the last entry in the last leaf node in each tree to determine the last disk block used. Pick the entry that has the larger starting offset and compare it to the size stored in the vd. The last used disk block is either the first block before this entry of the last block in the volume.

3.2.2.12.3 Execution Flow

- Call advfs_find_node (reserved_tree) with offset -1
- Call advfs_find_node(free_tree) with offset -1
- Chose the last entry in each node and compare their starting offsets choosing the larger of the two.
- If offset+size to < vd->vd_size
    - Return (vd->vd_size – 1)
- Else
    - Return(offset+size-1)

3.2.2.13 Stg_total_free_space

3.2.2.13.1 Interface

```
Bf_vd_lk_t_stg_total_free_space (
   vdT *vdp
)
```

3.2.2.13.2 Description

This routine replaces the sbm_total_free_space that was called at mount time. It will be used to fill in the field in the vd structure freeBlks (formerly freeClust). This will basically walk the leaf nodes of both the reserved and the free trees adding up the sizes. We will not include any space that happens to be in the DDL/orphan list since that is really not available until we can fit it into the storage trees. This seems like a heavy handed approach when we could just store the free bocks in the VD attributes record, however this would require a new pin record for every storage allocation/deallocation.

The concept of a cluster has now changed meaning. The cluster size this design refers to is the amount of reservation that we initially give out when space conditions permit. The cluster size that is referred to here is how much space an sbm bit maps. We now have allocation units that are potentially different for meta-data and user data. The analogous would be the minimum of these two although from the perspective of the storage allocator it is really meaningless. The allocation unit dictates that unit of contiguous space that is given out. How we manage the space in the b-tree is independent of this.

This routine will not obtain any locks since it is assumed that it is only being called at domain initialization time.

3.2.2.13.3 Execution Flow

- Walk the free space b-tree to the leftmost (zero offset) node.
- Loop thru all entries in this node adding up the sizes.
- Deref and ref the right node (using the right node page)
- Repeat until the rightmost node is reached.
- Repeat for the reserved tree.
- Return the total free space.

3.2.2.14 Advfs_get_nearest_diskblock

3.2.2.14.1 Interface

```
Status T Advfs_get_nearest_diskblock (
        bfAccessT *bfap,        /* IN - Access struct for file              */
        off_t *offset,          /* IN - offset in file to start allocation    */
        bf_vd_blk_t *target_diskblk, /* OUT – diskblock to request */
        bf_vd_blk_t *distance    /* OUT – distance to nearest block */
        )
```

3.2.2.14.2 Description

This will be called in the storage addition path. The xtntmap lock will already have been obtained so this routine will not need to hold it. This will examine the passed in bfap's extent maps using the passed in file offset to determine the diskblock that should be requested in order for the file to have a single extent if the file were to be written sequentially. This is intended to be used when the OFFSET_RESERVATION or out of order writes are requesting storage. The caller has guaranteed that the passed in offset has no backing storage.

3.2.2.14.3 Execution Flow

- Assert the extent map lock is held.
- Convert the offsets to fobs.
- if the passed in fob is > the file_size
    - set nearest_fob = file_size converted to fobs
    - Call imm_get_xtnt_desc for the nearest_fob.
    - Subtract from the ending fob of this extent from the requested_fob
    - Add this the ending disk block.
- else
    - Call imm_get_xtnt_desc for the requested_fob.
    - Assert returned extent descriptor is a hole.
    - Call imm_get_next_xtnt_desc to get the next extent
    - Call imm_get_prev_xtnt_desc to get the previous extent.
    - Calculate the distance from the end of the previous extent to the requested_fob
    - Calculate the distance from the beginning of the next extent to the requested_fob
    - Pick the smallest distance (if equal pick the previous)
    - If prev picked
        - Add this distance (converted to disk blocks) to the last diskblock of the extent map.
    - If next picked
        - Subtract this distance from this first disk block.
- Return the disk block and distance.

3.2.2.15 Imm_get_prev_xtnt_desc

3.2.2.15.1 Interface

```
statusT
imm_get_next_xtnt_desc (
                        bsInMemXtntMapT *xtntMap,  /* in */
                        bsInMemXtntDescIdT *xtntDescId,  /* in/out */
                        bsXtntDescT *xtntDesc  /* out */
                        )
```

3.2.2.15.2 Description

There already exists the imm_get_next_xtnt_desc routine. Take this routine and change the plus signs to minus signs. Not quite. There's a little more to it but not much. This will return the extent descriptor that this previous to the one we are working on.

3.2.2.15.3 Execution Flow

- Decrement the indexes

- Test for indexes < 0 instead of < max.
- If moving to previous subextent map start search at the end working backwards and ignoring the terminating descriptor.

3.2.2.16 Stg_get_rand_blk

3.2.2.16.1 Interface

```
Stg_get_rand_blk(
   vdT *vd

)
```

3.2.2.16.2 Description

This routine is borrowed from HPUX. This is used to generate random numbers in some subsystems in the kernel. The seed which is in the vd structure will be initialized at domain activation time to the low order bits of the current clock ticks. This will generate a random block offset that is within the volume. This will be used to generate a block for looking up storage for a request that does not have a target block preference.

3.2.2.16.3 Execution Flow

```
int return_value;
 vd->rand_block_seed = ((0x780341ab * rand_seed + 0x493f63d3) >> 2)%vd->vdSize;
return_value = vd->rand_block_seed_ & 0xffff;
 return (return_value);
```

3.2.2.17 Vd_extend

3.2.2.17.1 Interface

No change

3.2.2.17.2 Description

Compared to the current design, the use of b-trees makes this a much simpler operation. Basically we just need to call advfs_put_space for the amount the vd is being extended. This will place the new amount into the b-tree. In the rare case that the volume that the b-trees live on have no space left in the b-tree nodes and no space to split the nodes when we attempt to insert the new extended space, the advfs_put_space code will handle this situation. The space can always fit on the free space tree since if it were full then by definition there must be space available to split.

3.2.2.17.3 Execution Flow

- Remove the SBM bits calculations and replace with calculation of the starting disk block and size of the space being added to the VD.
- Remove the calls to allocate the SBM bits and write the raw pages to disk and replace with a call to advfs_put_space.

3.2.2.18 Advfs_check_vd_sizes

3.2.2.18.1 Interface

No Change

3.2.2.18.2 Description

This is called at mount time to determine if the size of the volume stored in our metadata matches the actual size of the volume as returned by an ioctl call to the volume. If we find the volume is smaller we will then locate the last used block on the disk and attempt to read it. If we are able to read this disk block we will allow the volume to be mounted in read-only mode.

The code to do this will be changed to no longer march thru the SBM. We will now call a dedicated function to determine this information and return the last used block.

3.2.2.18.3 Execution Flow

- Remove the loop of code that references the SBM trying to locate the first set bit. And replace it with a call to stg_locate_last_used_block.

3.2.3 Existing Indexed Directory Code

The existing indexed directory code will be utilized in the creation of the storage b-trees. This code will initially be copied over and modified to suit the needs to the storage b-trees. After the code has been stabilized, the routines will be revisited to see if common functions can be made and shared between the two subsystems. The return on investment for this is not very high.

All of the following routines and local variables and references to the directory indexes will be replaced with meaningful references to the storage trees.

3.2.3.1 Idx_bsearch_int

The handling of collisions within the search keys will be removed. There will be no collisions in the storage trees and this is just extra overhead.

3.2.3.2 Idx_index_get_free_pgs_int

The call to rbf_add_stg will not be passed the STORAGE_BTREE_SPLIT1 flag. Calls to update quotas will be removed since this is a reserved file and not charged to any pid. The updating of index statistics will also be replaced/

3.2.3.3 Idx_prune_start

Conditional filename tree processing will be removed. This tree did not have nodes doubly linked and extra processing was needed. We will not need this now. Due to locking restrictions we will no longer on an error call idx_prune_finish. The message to the thread must succeed.

3.2.3.4 Idx_prune_btree_int

Remove any special processing related to the FNAME tree which only has singly linked nodes.

3.2.3.5 Idx_remove_page_of_storage

Remove the updating of quotas.

3.2.3.6 Idx_undo

The sections of code relating to the creation/removal of the index file and the opening of the directory will be removed. The calls to undo operations involving the filename tree will be removed.

- Update VD and domain storage sizes to reflect the storage we have undone.
- Update the bfap->lastAllocBlk to reflect any storage we have undone.

3.2.4 Removal of storage and ENOSPC

A request to remove storage does not expect to get an ENOSPC error. Placing a piece of storage back into the b-tree could cause the b-tree to need to grow. This could result in an ENOSPC error. As mentioned earlier, to be in a situation that there is no space to grow then the b-trees should be pretty sparse. However this mixed allocation unit case could put us into a situation where this could happen.

To get around this, the deferred delete list will be taken advantage of. A deletion of storage is a 2 step process. The first step will remove the storage from the file's in-memory structures and place the storage (effectively) on the deferred delete list. The second step is to remove the storage from the deferred delete list. This 2 step process is needed for transactional reasons. Storage can not become available until the transaction that removed the storage has completed. Otherwise if the transaction was failed and the storage needed to be given back another file may have already claimed that storage.

This second step in the deletion process will be modified slightly. During this step, the storage that is being removed will be attempted to be placed into the storage b-tree. If the b-tree needs to grow and can not get storage to grow, an ENOSPC error will be returned. As detailed in the storage insertion sections, steps are being taken to make this event extremely rare. The existing code will be changed to place the Mcell ID of the entry on the deferred delete list and also onto an orphan list that is hanging off the domain.

Any more deletions that come in will look to see if the orphan list is non-empty and will also attempt to remove any storage that is on the orphan list as well. If any of these fail they will be placed back on the orphan list for the next deallocation to attempt.

This approach requires only in-memory changes, since at mount time the deferred delete list will be attempted to be cleaned up, there by also cleaning up any orphans. If during this processing of the deferred delete list we can not clean up storage, it will once again be placed back on the orphan list. Any call to walk the deferred delete list must first remove all entries from the orphan list with out processing them.

Vfast can also be taught to attempt to clean up any orphan entries periodically. This is not the optimal solution but due to the rarity of such an event occurring, it seems a reasonable approach. In the futures section of this document, there is a discussion of removing mixed block sizes all together that would eliminate this problem completely.

The pruning of the storage b-trees will be almost the same as the pruning of the directory index b-trees. After an entry is removed from a node the node will then be pruned before returning to the caller. The pruning will collapse the node with its neighbor node, and a call to stg_remove_stg_start will be made. This will insure that the storage is out of the b-tree. A message will then be sent to a kernel thread to actually perform the call to stg_remove_stg_finish. This is the existing directory index model and works especially well here. The thread will then call dealloc storage to actually place the storage back into the b-trees. Otherwise we would need to worry about the recursive nature of deallocating storage inside of the call to deallocate storage. With this model the recursive locking issues are not present.

3.2.4.1 Del_dealloc_stg

3.2.4.1.1 Interface

```
statusT del_dealloc_stg(
    bfMCIdT pmcid,      /* in - primary Mcell ID */
    vdT *pvdp,           /* in - virtual disk of primary Mcell */
```

```
            uint64_t storage_flags
        )
```

3.2.4.1.2 *Description*

Before processing the current deallocation request check the orphan list in the current vd. If there are any entries then process them as well. If any call to del_xtnt_array fails, create an orphan entry and place it in the vd.

Pass the storage flags down to del_xtnt_array.

3.2.4.1.3 *Execution Flow*

- If the vd's orphan list head is not -1
    - Attach the passed in mcellId to the front of the orphan list and remove the list from the vd.
    - Remove the list from the vd.
- Wrap the processing for a primary Mcell in a loop and loop thru the entire orphan list.
- Change the error checking for del_xtnt_array to:
    - Malloc an orphan entry and insert it on the orphan list.
- Release the memory for orphan entry.
- If the orphan list is not empty
    - Generate an event to indicating that the disk is in a fragmented state and would benefit from running defragment.

3.2.4.2 Del_clean_mcell_list

3.2.4.2.1 *Interface*

```
statusT del_clean_mcell_list(
    vdT *vdp,
    uint64_t flag
    )
```

3.2.4.2.2 *Description*

In order to avoid processing an entry twice this routine will need to check for any orphan list on the vd and release all the entries. After processing the ddl the orphan list will automatically be regenerated if an orphan still exists.

3.2.4.2.3 *Execution Flow*

- If the vd's orphan list head is not -1
    - Loop thru and release the memory for each entry on the list.
    - Set the orphan list head in the vd to -1.

3.2.4.3 Del_xtnt_array

3.2.4.3.1 *Interface*

```
statusT del_xtnt_array(
    bfMCIdT pmcid,          /* in - Mcell ID of primary Mcell */
    bfMCIdT mcid,           /* in - Mcell ID of extra or shadow xtnts record */
    vdT *pvdp,              /* in - pointer to virtual disk of primary */
```

```
        vdT *vdp,               /* in - pointer to virtual disk of xtra or shad */
        bfMCIdT *nextMCId,      /* out - Mcell ID of next Mcell */
        uint64_t storage_flag           /* in – storage flags */
        )
```

3.2.4.3.2 *Description*

This routine will be modified to accept a new storage flag that will be passed on to the storage routines.

The processing of the extents into contiguous ranges will not change. There is code in place to keep track of the number of pinned pages and to break out of processing an extent and start a new subftx if the number of pinned pages is too large. This processing can be removed since the number of pinned pages will not be a function of the size of storage being added into the b-tree and the b-tree operations are themselves wrapped in a sub transaction.

3.2.4.3.3 *Execution Flow*

- The while loop to process a single extent can be removed.
- We will call del_range once per extent and no longer pass in the pinPages argument.

3.2.4.4 Del_range

3.2.4.4.1 *Interface*

```
    del_range(
        bf_vd_blk_t start_vd_blk,  /* in - start of range to zero */
        bf~~_vd_blk_t *vd_blk_cnt,   /* in- number of blocks to free~~
~~                                    * out - number of blocks actually freed */~~
        ~~int *pinPages,       /* in/out - maximum number of pages that can be~~
~~                              pinned / number of pages that were pinned */~~
        bf_vd_blk_t vd_blk_cnt  /* In – number of blocks to free */
        vdT *vdp,           /* in - virtual disk containing range */
        bf_fob_t pgSz,           /* in - page size in 1k fobs */
        ftxHT ftxH          /* in transaction handle */
        uint64_t storage_flag           /* in – storage flags */
        )
```

3.2.4.4.2 *Description*

This routine will be modified to accept a new storage flag that will be passed on to the storage routines.

This routine will now call the new storage allocation interfaces.

3.2.4.4.3 *Execution Flow*

- Remove the call to sbm_howmany_blks.
- Replace the call to sbm_return_space_no_sub_ftx with advfs_put_space(vdp, start_vd_blk, vd_blk_count, storage_flags, ftxH);

3.2.5 Changes to existing storage routines

Unless specified here all sbm routines will be removed and replaced with corresponding storage allocation routines that are described in the preceding sections.

3.2.5.1 CANT_SET_TWICE

This routine will be renamed to CANT_DEALLOCATE_TWICE and the appropriate fields and panic text will be updated.

3.2.5.2 CANT_CLEAR_TWICE

This routine will be renamed to CANT_ALLOCATE_TWICE and the appropriate fields and panic text will be updated

3.2.6 Changes outside of kernel space

3.2.6.1 Vd parameter retrieval

The retrieval of volume parameters will need to either convert to the new model or we can leave the return fields alone and just return the values based on the b-trees. The idea of a storage cluster can be replaced by the minimum storage allocation unit.

3.2.6.2 Mkfset

We can remove the preallocation option.

3.2.6.3 Vods tools and fsck

The vods tools will need to change to understand the new layout of the on-disk b-trees. Unlike the static SBM, the b-trees will be dynamic and there layout unpredictable. The tools will need to understand how to walk the b-trees in order to obtain the information about the space layout. All information that was in the SBM can be found from the b-trees. There is also some extra information with the addition of the reserved tag.

Teaching these tools to understand the storage b-trees will also accomplish most of the work for examining the directory b-trees which currently is missing from the vods tools.

3.2.7 Changes in support of VFAST

Vfast tries to defragment a file at a time. Currently it keeps a list of active files. This list represents files that were recently closed. Vfast does not make any effort to defragment files on the disk that are not active. There are plans however to change vfast to do this in a background or foreground mode thereby replacing the need for the defragment utility.

The approach Vfast takes is to examine a file to determine if it needs to be defragmented. How it makes this determination is not relevant to this design document. Once it makes this determination it then attempts to create a single extent for this file. It does this by attempting to locate a run of free space on the disk that can contain the file in a single extent. If it can not find this, it will pick a portion of the file that is most fragmented and attempt to find a location on the disk that can hold all of these extents. If this fails it will then scan the free space looking for an area on disk that is mostly made up of free space. Once an area is found it will then lock down this range of storage and begin moving the allocated pieces of the storage out of the way in order to create a large contiguous chunk.

This methodology can work against the storage allocator proposed in this document. Initially it seemed like an all together different approach would be better. This approach would work with the existing extents in the file and try to move other extents adjacent to them. This avoids this moving data back on forth over regions of the disk. This approach however would basically mean a redesign of vfast. Given time constraints and the fact that this approach may not offer the return on investment, it is decided that we will attempt to work within the framework of the existing vfast design.

The interfaces that vfast uses to find storage for this migration will be maintained. The implementation will need to change but the basic design will remain. A new storage policy will be introduced that will indicate to the storage allocator that it is vfast. This policy will attempt to avoid stealing reservations from other files until the disk space is at point where the reservations are deemed detrimental to the overall performance of the file system. This would most likely be at a point where the disk is around the 75% full mark. At this point it would be nice to then make intelligent choices as to the storage that we are stealing.

It is not the intent of this design to also design vfast. But this design will attempt to put forth interfaces that should aid vfast in making the correct decisions about which storage to steal. Ultimately it will be up to vfast how to use these interfaces. For example, some file reservations may never get used. A file may get created and then storage never added to it again. This file would be a prime candidate for stealing its reservation. We could include in the storage metadata information the time of last allocation from the reserved space for a given tag and than make a more intelligent decision about stealing this space. The problem with this is that is reduces the number of storage descriptors that we can now store in the node of a b-tree as well as increase the amount of data that needs to be updated transactionally. We want to off load this type of overhead to vfast which is running as a kernel background thread. This type of overhead should not be in the mainline storage path.

A potential solution to the above example is to use the STATIC_RESERVATION policy. Vfast can attempt to identify files that are not growing by observing their ctimes. If the ctimes have not been updated then storage has not been added to the file. Vfast can call in with the last diskblock offset used by this file and query if there is any reserved space. If there is Vfast can then ask for an allocation of that size using the STATIC_RESERVATION policy. This allocation will come from cluster marked for no reservations. Storage will be given out from the first free space in this cluster. Thereby packing in all files into this storage. By migrating this file over to this type of storage the reservation for this file will become available. Of course guessing wrong can hurt the performance of this file.

In the meantime, vfast will use a storage policy of CONTIGUOUS_ALLOCATION which will cause the storage allocator to attempt to locate space of this size that is contiguous on disk. A range can be given. The storage allocator will first attempt to obtain space from the free space tree before scavenging the reservation tree. If it can not find space of this range then ENOSPC will be returned and vfast will have to resort to other methods.

The following section will detail the changes that need to be made to the existing vfast routines in order to interface with the new b-tree storage allocator.

3.2.7.1 Stg_scan_range

3.2.7.1.1 Interface

```
statusT stg_scan_reserved_range (
    off_t starting_offset,
     off_t *ending_offset,
    size_t *size,
    uint32_t index,
    stgBtreeNode node
)
```

3.2.7.1.2 Description

This routine will be called by stg_scan_tree as each entry is being processed. It will be used to walk the reserved tree either by itself or in conjunction with the free space tree. If it is being called in conjunction with the free space tree then the btree_node and index will be NULL. The starting offset and ending offset will be used to begin searching this range of diskblock offsets in the tree. The search will stop either when the size is found or the ending offset is reached. This routine will add up each entry in the reserved tree until one of these criteria is reached. Then either a smaller size or shorter offset will be returned.

3.2.7.1.3 Execution Flow

- If btree_node is NULL
    - Call advfs_stg_look_up for the passed in starting offset.
- Using the btree_node and index begin adding up each size entry while incrementing the index thru the node

- If either the size or ending offset is reached return updating the appropriate values.
- If the end of the node is reached.
    - Deref this b-tree page and ref the next b-tree page
    - Repeat the search.

3.2.7.2 Stg_scan_trees

3.2.7.2.1 *Interface*

```
statusT stg_scan_trees (
    off_t *starting_offset,
     size_t *size,
    )
```

3.2.7.2.2 *Description*

This routine will be called by vfast when it can not find a contiguous chunk of free space large enough to hold the file it is trying to migrate. This routine will scan thru all the entries in both free and reserved space trees in an attempt to find the shortest range on disk that contains the requested space. Vfast can then use this range by migrating all the storage within this range to one end thereby freeing up the other end with a chunk of contiguous space equal to its needs.

The method of scanning for this range will be somewhat sub optimal. We will start at the beginning of the disk and scan forward looking until we find a range that encompasses the amount of free space we want. We will then starting at the end of that range scan again. We will keep track of the shortest range that still encompasses the space we need. This requires one pass thru the storage trees and should produce a relatively good approximation of the sparest range on the volume. The exhaustive approach would be the scan the maps multiple times starting with each new chunk of free space. This is too expensive however and will not yield that much better of an answer.

We will start with offset zero and search the reserved tree until we either find the space we want of hit the offset of the first chunk of storage on the free space tree. If we still have not found the space we want, we will ass this free chunk in an search the reserved tree until the offset of the next chunk of free space (or the size is found). After finding a range that encompasses our needs we will start this search up again at the ending offset of the range we just found.

If no free chunks exist then only the reserved tree will be examined. The shortest distance will be chosen or the range and size that we did find if none was found that satisfied the requested size.

VFAST is prepared to receive a size back that is less than the requested size if we can not find that much available space.

3.2.7.2.3 *Execution Flow*

- Lookup of the leaf node of the free space tree
- Starting with the first entry in the node
- Set starting offset to zero.
- For all entries in the node
    - While the passed in size is not met.
        - Calculate the ending offset the of the free tree entry (or set to -1)
        - Call stg_scan_range with the starting offset and the ending offset from above.
        - If the returned size is equal to the size we are looking for then

      - If the length of the returned range is the smallest found
        - Save the offset and length.
        - Set starting offset to the ending offset of this range.
        - Break
    - Else move to the next entry and include this size into the range
      - Set starting offset to the ending offset of this range.
      - If the size is large enough
        - If smallest
          - Save offset and length
        - Break
  - Move to the next entry in the node.
- If there were any entries in the free space tree
  - Return the smallest range found
- Look up the first node in the reserved tree
- Starting with the first entry
- While entries exist in the reserved tree
  - Call stg_scan_range for this offset and a -1 ending offset
  - If a size was found
    - If it is the smallest range
      - Save the offset and length
  - Set the next entry to start to be the next entry in the b-tree (not the last entry in the range returned)
- Return the smallest range found (this may be the only range found and it may not satisfy the requested size)

3.2.7.3 Stg_lock_unlock_range

3.2.7.3.1 Interface

`No Changes.`

3.2.7.3.2 Description

This replaces the sbm_lock_unlock_range routine. This routine will mark a range as reserved in the VD structure. All allocations will avoid this range of storage when allocating storage if other storage is available. The previous code also removes this range from the SBM in memory cache. This will no longer need to be done. We will remove sbm_lock_range as well.

3.2.7.4 ss_find_space

3.2.7.4.1 Description

Replace the call to sbm_find_space with stg_find_space and sbm_lock_range with stg_lock_unlock_range.

3.2.7.4.2 ss_get_n_lk_free_space

3.2.7.4.3 Description

Replace all sbm calls with the corresponding stg calls. Replace the use of the vd->stgCluster calculations with the new freeBlks in the vd.

3.2.8 Future Enhancements

3.2.8.1 Vfast

Today vfast locates files and attempts to consolidate them. It will migrate storage around in an attempt to create a chunk of space large enough to hold the file that it is trying to coalesce. This will work against the new storage allocator design.

The approach VFAST should take is less file centered and more free space centered. Its goal should be not to try to reduce a files extent but to increase the ranges of free space by making files extents more contiguous. The new allocation philosophy is to create what would look like fragmented storage. We want to have files with free space adjacent to the file's used space in order for the file to be able to grow into contiguous space. This is more of a checker board appearance of storage and free space.

Vfast should make passes thru all files in the BMT for a given volume. It should then look for extents that are file offset contiguous but not disk offset contiguous. Vfast can then attempt to migrate the storage to make the extents contiguous. We would need to follow certain rules here so that vfast does not find itself remigrating extents back and forth.

- Gravity rule: always attempt to put to extents together such that the smaller extent combines with the larger extent.
- If two extents are the same size then migrate them towards the beginning of the file.

Vfast can then make a targeted storage request for a diskblock. We can introduce a new policy to the storage allocator that will cause it to fail if the targeted block can not be located. Vfast would notice that two extents could be disk block adjacent, calculates the disk block that is adjacent to the extent in question and requests that block. If it succeeds it can migrate the space to that block causing the old space to be freed. Successive passes thru the BMT should make more and more progress as blocks are freed.

This is a good phase 1 approach. We would not need to actually move storage around to make room which means that we do not need to search thru the BMT to find the extent map that maps some piece of storage that we are trying to move.

Future phases should be more aggressive. Vfast should try to locate those files that have storage reserved for them but are not likely to use it. This could be another storage policy (STATIC_ALLOCATION) that tells the storage allocator to pack all allocations into a storage cluster. If the request for storage is of this type then the known reserved storage that is being packed will be used for allocations. In this way vfast could pack away files that are static in size and do not benefit from reserved space.

3.2.8.2 Non-contiguous Meta-data Allocation Units

Allowing meta-data and user-data to exist in the same domain with different allocation units can lead to unexpected fragmentation issues. When the user data has an allocation unit that is less than the meta-data allocation unit, situations can arise where the disk appears to have free space yet allocations are failing. This can result because there are only contiguous pieces of storage left on-disk that are of the user-data size. A meta-data file will not be able to grow in this situation. If storage is requested and this will cause one of the meta-data files to need space the allocation will be failed.

This will also present a problem for vfast when attempting to pack a range in order to produce contiguous free space. Vfast will need to be aware of the allocation unit requirements for any file that it is attempting to move space for. This can become a difficult problem to solve.

In the investigation of this design it was noticed that keeping an allocation unit in terms of the minimum amount of storage that must be allocated for a given file can actually be independent of the allocation being contiguous on-disk. With some modifications to the storage allocator, getpage, putpage and the i/o completion code we should be able to allow for the allocation unit to actually be non-contiguous for meta-data.

This will remove the need for the special processing added around the orphan list that is kept when attempting to grow the storage b-trees. The on-disk recovery tools such as salvage and fixfdmn would need fairly wide spread changes.

3.2.8.3 Mixed allocation units

Running with mixed allocation units can put the domain in a potentially degraded state over time. This can happen when the reservation tree mainly consists of the smaller allocation unit size. In this case a request to allocate storage for a large allocation unit will need to begin exhaustively searching the b-tree for space that satisfies the request. Generally speaking as space is used in the domain the reserved tree should have fewer and fewer entries, so the search time should be reduced. However if the fragmentation of the domain is such that only many small allocations are available, we will have longer search times. Some approaches to solving this problem are to keep a hint in the VD that point to the offset where the last large allocation unit chunk was found. Then each search can start up where the last one left off.

Another consideration is to stop breaking reservations in half during scavenging when we hit a threshold. Instead we can begin stealing from the beginning of the reservation. This will help to keep available storage from fragmenting into the smaller allocation unit.

3.2.8.4 Allocation pattern prediction

Today in HPUX we have a sophisticated read ahead prediction module. This will identify patterns of reads and writes based on past behavior in an attempt to predict future behavior. Once a pattern is established we will attempt to read pages into the cache that we feel will be requested shortly. This allows the application to find its pages in memory without having to block on i/o.

A similar approach could be applied to the allocation of storage. If we notice a pattern of allocation requests we could attempt to reserve storage in this pattern in order to allow for the data to be contiguous on disk. This could be accomplished using a new policy definition.

3.2.8.5 Preferred Allocation Unit

This storage design will allow for different allocation units. This unit however will be enforced in the sense that if an allocation that is atleast this size can not be located then the request will be failed. We could introduce a second allocation unit type. The preferred allocation unit. Here the user could specify both a minimum allocation unit that we will never hand out space smaller than this, but also a larger preferred allocation unit that we will try to satisify. This space will be handed out in these larger units when ever possible (this is the PREFERRED_ALLOCTION policy) but rather than failing the request or if finding this larger unit will require a lot of searching, we could allocate a smaller piece. This would not be a major addition to this design.

4 Dependencies

4.1 System Administration

- No dependencies.

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

- No dependencies

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)

- No dependencies.

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

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

Low Priority

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