AdvFS User File Pre-Allocation

Design Specification

Version 2.0

GKS

CASL

Building ZK3
110 Spit Brook Road
Nashua, NH 03062

Design Specification Revision History

Version	Date	Changes
0.1	10/23/2003	First draft for internal review
0.2	12/15/2003	Second draft for internal review
1.0	02/25/2004	First version for general distribution
2.0	06/30/2004	Second version for general distribution

Table of Contents1 .... Introduction 6

1.1 Abstract.. 6

1.2 Product Identification 6

1.3 Intended Audience 6

1.4 Related Documentation 6

1.5 Purpose of Document 7

1.6 Acknowledgments & Contacts 7

1.7 Terms and Definitions 7

2 Design Overview 8

2.1 Design Approach 8

2.2 Design Background 8

2.3 Overview of Operation 8

2.4 Major Modules 9

2.4.1 fsadm prealloc.. 9

2.4.2 fsadm setattr... 9

2.4.3 fsadm getattr 9

2.4.4 advfs_ioctl 9

2.4.5 advfs_setext 9

2.4.6 advfs_getext.. 9

2.5 Major Data Structures.. 9

2.5.1 advfs_ext_attr_type_t 9

2.5.2 advfs_ext_attr_t 9

2.5.3 advfs_ext_prealloc_attr_t 9

2.6 Exception Conditions 10

2.6.1 Invalid Input Parameters.. 10

2.6.2 Resource Depletion.. 10

2.6.3 Insufficient Privilege 10

2.7 Design Considerations.. 10

3 Detailed Design 11

3.1 Data Structure Design... 11

3.1.1 advfs_ext_attr_type_t 11

3.1.2 advfs_ext_attr_t 11

3.1.3 advfs_ext_prealloc_attr_t 11

3.2 Existing Data Structure Modification 12

3.2.1 struct bfAccess... 12

3.2.2 struct bsBfAttr 12

3.3 Module Design 13
3.3.1 Commands 13
3.3.1.1 fsadm prealloc 13
3.3.1.2 fsadm setattr 14
3.3.1.3 fsadm getattr 14
3.3.2 Kernel Interface & Functions 15
3.3.2.1 advfs_ioctl 15
3.3.2.2 advfs_setext 16
3.3.2.3 advfs_getext 16
3.3.2.4 advfs_setext_prealloc 17
3.3.2.5 advfs_getext_prealloc 22
3.3.2.6 bs_map_bf 22
3.3.2.7 fs_trunc_test 22
3.3.2.8 bf_setup_truncation 23
4 Dependencies 25
4.1 File System Layout 25
4.2 File Systems 25
4.3 I/O System and Drivers 25
4.4 Auditing 25
4.5 Behavior in a cluster 25
4.6 Commands 25
4.7 Update/Rolling Upgrade 25
4.8 Learning Products (Documentation) 25
5 Issues 26
5.2 Additional work for recovery 26

Preface

The User Pre-Allocation Design Specification contains information gathered from team members and publicly available documentation on competing product functionality It is a proposal of what and how to implement user file pre-allocation in AdvFS. If you have any questions or comments regarding this document, please contact:

Author Name	Mailstop	Email Address
GKS		

Sign-off review

Approver Name	Approver Signature	Date

1 Introduction

1.1 Abstract

User file pre-allocation functionality is to be used to guarantee file storage of a specified size at file create time. Future additions to this functionality could include the ability to pre-allocate storage contiguously, pre-allocate space to a non-zero length file, and to alter allocation *policy*.

1.2 Product Identification

Project Name	**Project Mnemonic**	**Target Release Date**
AdvFS User File Pre-Allocation	AdvFS User Pre-Alloc	HP-UX

1.3 Intended Audience

The reader of this document is assumed to have a high level understanding of the AdvFS functionality. For more information, please refer to the Hitchhikers Guide to AdvFS, the admin guide, and/or related man pages. The intended audience is the HP-UX technical community.

1.4 Related Documentation

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

Item	**Document**	**URL**
1		
2		
3		
4		
5		
6		

1.5 Purpose of Document

This design is intended to document the results of the investigation into implementation of user file pre-allocation with AdvFS in the HP-UX operating system. It intends to elaborate on existing HP-UX functionality and propose user and kernel implementations.

1.6 Acknowledgments & Contacts

The following people have made useful contributions to the course of this investigation: TM, BT, BN, DB, JA, and DL.

1.7 Terms and Definitions

Term	Definition
.*_EXT_.*; .*_ext_.*	Data structures, flags, and functions which refer to file extents with the abbreviation “ext” are user-visible
.*_XTNT_.*; .*_xtnt_.*	Data structures, flags, and functions which refer to file extents with the abbreviation “xtnt” are internal to AdvFS.

2 Design Overview

2.1 Design Approach

The kernel interface design should be generic enough to provide for future extent based attributes to be modified without radically altering what has already been implemented. Thus, the kernel interface to extent attribute manipulation will be through an *ioctl(2)*.

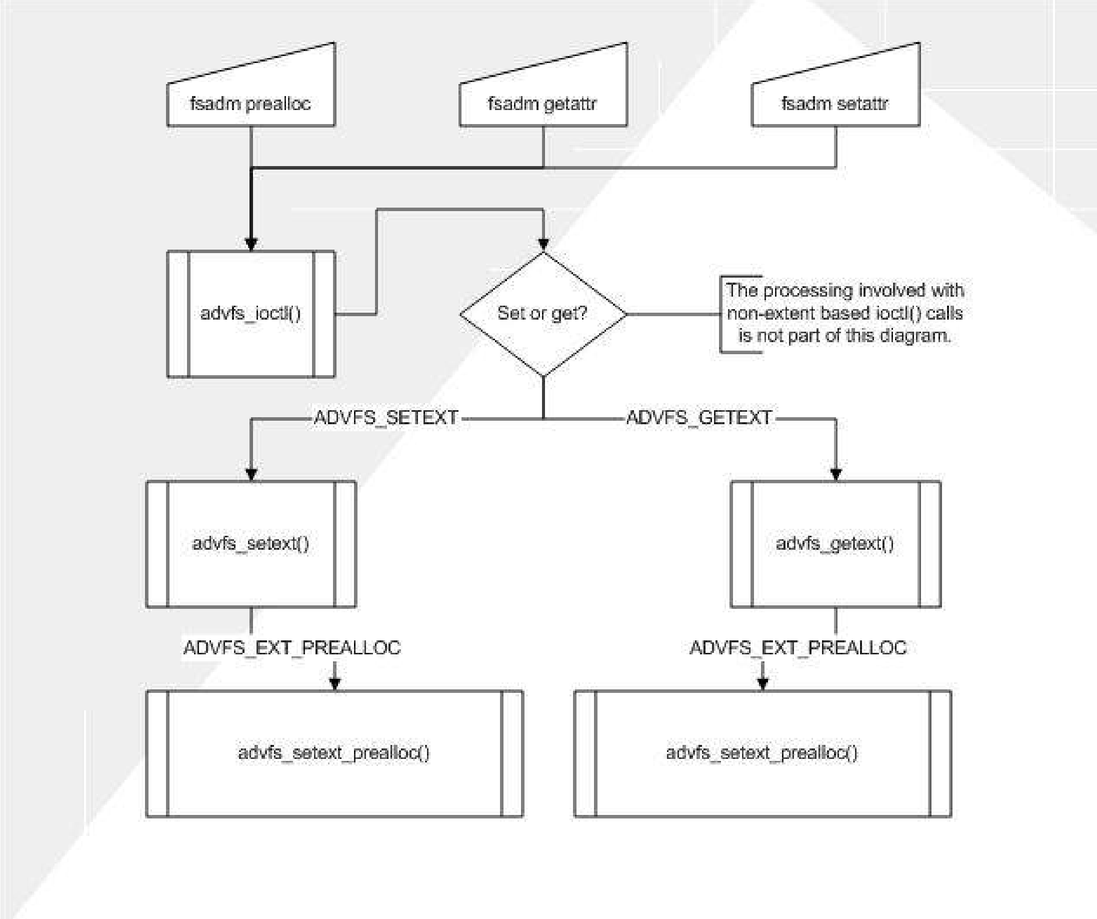
2.2 Design Background

2.3 Overview of Operation

AdvFS shall provide three major user space command interfaces for dealing with file pre-allocation. Those command interfaces will be through the *fsadm_advfs(1m)* modules called “prealloc”, “setattr”, and “getattr”. These command interfaces will all operate through the AdvFS *ioctl(2)* which shall serve as the sole interface to the kernel for file extent manipulation from user space.

The kernel entry point will be *advfs_ioctl*, which is the AdvFS vnode operation (VOP) for an *ioctl(2)* call upon an AdvFS file. The function *advfs_ioctl* will branch based upon two *ioctl(2)* commands – ADVFS_SETEXT or ADVFS_GETEXT. The former will call into a new function called *advfs_setext()* and then branch based on the extent attribute being modified. The latter will call into a new function called *advfs_getext()* to return the requested extent attribute information for the specified file back to user space.

The following diagram illustrates the overall execution flow.



2.4 Major Modules

Six major modules exist to implement user file pre-allocation.

2.4.1 fsadm prealloc

The *fsadm prealloc* command will provide a simple one way interface pre-allocate file storage. It will access a file as an argument and will allow root to pre-allocate non-zeroed storage. It will also allow space to be reserved without updating the file's size using the "reserveonly" option.

2.4.2 fsadm setattr

The *fsadm setattr* command will not initially be available, as it would serve the same purpose that *fsadm prealloc* provides. It will, however, serve as the generic interface to alter file extent attributes from user space in the future.

2.4.3 fsadm getattr

The *fsadm getattr* command will be updated to display file extend attributes, if any exist. Initially, the generic command 'du' will be used to report on-disk allocation (for reserved space). Eventually, *fsadm getattr* will serve as the generic interface to query file extent attributes from user space.

2.4.4 advfs_ioctl

The AdvFS *ioctl(2)* already exists, but it will be updated to accept new commands for querying and setting extent attributes – ADVFS_GETEXT and ADVFS_SETEXT respectively.

2.4.5 advfs_setext

The new function *advfs_setext* will branch to the appropriate function for setting the specified extent attribute. Initially it will only branch to the function *advfs_setext_prealloc()*.

2.4.6 advfs_getext

The new function *advfs_getext* will handle all queries of extent attributes originating from user space by branching to the appropriate handler function. This will initially be *advfs_getext_prealloc()*.

2.5 Major Data Structures

Three major data structures will be used to implement user file pre-allocation.

2.5.1 advfs_ext_attr_type_t

This is an enumeration of the valid extent attribute modification types.

2.5.2 advfs_ext_attr_t

This is one extent attribute. It is a key/value pair. The value is a union of extent attribute specific data structures.

2.5.3 advfs_ext_prealloc_attr_t

This is the pre-allocation specific attribute data structure.

2.6 Exception Conditions

2.6.1 Invalid Input Parameters

- ENOTSUP - Non-existent extent attribute requested modification
- EINVAL - Invalid number of bytes to pre-allocate

2.6.2 Resource Depletion

- ENOSPC – returned from *ioctl()*: pre-allocate would consume more than available disk space
- ENOMEM – returned from *ioctl()*: system memory insufficient for operation

2.6.3 Insufficient Privilege

- EPERM - Non-root attempt to pre-allocate non-zero storage

2.7 Design Considerations

This design takes into consideration future work allowing a user to modify file extent attributes. Therefore, initial implementation will not include *setext* or *getext*, even though they are discussed.

3 Detailed Design

User file pre-allocation is achieved by modifying file extent attributes. For this reason, it makes sense to have one generic way to alter extent attributes which can grow for future functionality as well as providing for current requirements.

3.1 Data Structure Design

3.1.1 advfs_ext_attr_type_t

This structure is an enum which contains supported extent attribute modifications. This structure need not be allocated or freed as it is embedded within the **advfs_ext_attr_t** structure. The values within this enumeration will be used to branch within the kernel functions **advfs_setext()** and **advfs_getext()**, described below.

```
typedef enum advfs_ext_attr_type {
        ADVFS_EXT_NOOP = 0,
        ADVFS_EXT_PREALLOC = 1
} advfs_ext_attr_type_t;
```

Enum Value	Description
ADVFS_EXT_NOOP	Default Initial Value, no attribute information.
ADVFS_EXT_PREALLOC	Attribute is pre-allocation information.

3.1.2 advfs_ext_attr_t

This data structure contains a type and value which represent one AdvFS extent attribute. The type is one of the supported extent attribute types (above). The value is a union of command specific data structures. At the moment, this is only **advfs_ext_prealloc_attr_t**.

```
typedef struct advfs_ext_attr {
        advfs_ext_attr_type_t type;
        union {
        advfs_ext_prealloc_attr_t prealloc;
} value;
} advfs_ext_attr_t;
```

Field Name	Description
type	Type of attribute (determines data structure in union)
value	Union of command specific argument data structures

3.1.3 advfs_ext_prealloc_attr_t

This structure is the specific argument structure for the pre-allocation extent attribute.

```
typedef struct advfs_ext_prealloc_attr {
                uint64_t flags;
                uint64_t bytes;
} advfs_ext_prealloc_attr_t;
```

Field Name	Description
flags	Pre-allocation specific flags (these are bitwise OR'd) Currently: **ADVFS_EXT_PREALLOC_NOZERO**: do not zero on-disk extents, **ADVFS_EXT_RESERVE_ONLY**: do not update the file's size; In the Future: **ADVFS_EXT_PREALLOC_CONTIG**: preallocate space contiguously, **ADVFS_EXT_PREALLOC_NOERR**: preallocate as much space, up to the requested amount, and therefore do not error;
bytes	Number of bytes to pre-allocate

3.2 Existing Data Structure Modification

The following additions are not visible to the user.

3.2.1 struct bfAccess

The *bfAccessT* structure will be modified to include a new field indicating the amount of reserved space the file contains and that it should not be truncated.

```
typedef struct bfAccess {
    …
    off_t file_size;
    off_t rsvd_file_size;
    …
} bfAccessT;
```

3.2.2 struct bsBfAttr

A corresponding field will be added to the *bsBfAttrT* structure indicating that a file contains reserved space and therefore should not be truncated.

```
typedef struct bsBfAttr {
    bf_fob_t    bfPgSz;              /* Bitfile area page size in 1k fobs  */
    ftxIdT      transitionId;        /* ftxId when ds state is ambiguous */
    bfStatesT   state;               /* bitfile state of existence */
    serviceClassT reqServices;
    int32_t     bfat_del_child_cnt;  /* Number of children to wait for before
                                        deleting the file. Used to defer delete
                                        of parent snapshots. */
    uint32_t    rsvd1;
    uint64_t    bfat_orig_file_size; /* filesize at time of snapshot creation */
    uint64_t    bfat_rsvd_file_size; /* minimum space to reserve for file */
    uint64_t    rsvd3;
    uint64_t    rsvd4;
} bsBfAttrT;
```

3.3 Module Design

3.3.1 Commands

The following commands provide a user interface to file pre-allocation. Initially, there will be only *fsadm prealloc* since *fsadm setattr* and *fsadm getattr* would provide no further functionality. It is, however, useful to note their design here as future functionality would use these commands.

3.3.1.1 fsadm prealloc

3.3.1.1.1 *Interface*

```
fsadm prealloc [-o option_list] file size
```

3.3.1.1.2 *Description*

The *fsadm prealloc* command accepts the following arguments:

-o	option list flag. valid options: nozero – do not zero allocated space (user must be root) reserveonly – do not update the file’s size when pre-allocating storage

A user will use the command *fsadm prealloc* to interface with the file pre-allocation functionality. The size can be specified as either a number of bytes, kilobytes, megabytes, or gigabytes using the suffixes ‘’, ‘K’ or ‘k’, ‘M’ or ‘m’, and ‘G’ or ‘g’ respectively. Specifying zero for the size will truncate the file to the last allocation unit at or before the end of the file (i.e. this will remove reserved storage). Its general program flow will be as follows.

3.3.1.1.3 *Execution Flow*

```
int fd = 0;
int err = 0;
int c = 0;
int Vflg = 0;
int oflg = 0;
char *options = NULL;
advfs_ext_attr_t attr = { 0 };
extern int optind;
extern char *optarg;

attr.type = ADVFS_EXT_PREALLOC;

while((c = getopt( argc, argv, "s:" )) != EOF) {
    switch(c) {
    case 'V':
        Vflg++;
        break;
    case 'o':
        oflg++;
        options = strdup(optarg);
        break;
    default:
        prealloc_usage();
        exit(1);
    }
}

if( (argc - optind) != 2) {
    prealloc_usage();
    exit(1);
}

if( oflg ) {
    if((parse_options( options, &(attr.value.prealloc.flags) )) == FALSE) {
        prealloc_usage();
        exit(1);
    }
}

attr.value.prealloc.bytes = str_to_bytes(argv[optind + 1]);
if(attr.value.prealloc.bytes < 0) {
    prealloc_usage();
    exit(1);
}

if((fd = open(argv[optind], O_RDWR | O_CREAT)) < 0) {
    perror("open");
    prealloc_usage();
    exit(1);
}

if(err = ioctl(fd, ADVFS_SETEXT, &attr)) {
    perror("ioctl");
    prealloc_usage();
    exit(1);
}

close(fd);
```

3.3.1.2 fsadm setattr

3.3.1.2.1 *Interface*

```
fsadm setattr -F advfs [-o <options>] [-p <size>] <file>
```

3.3.1.2.2 *Description*

The ***fsadm setattr*** command accepts the following flags:

-o	Option flag – options are specific to action being performed
-p	Specifies that the extent attribute being set is pre-allocation

The ***fsadm setattr*** command will be the generic interface to extent attribute manipulation. More options will be added in the future as requirements demand. User file pre-allocation options are 'nozero' and 'reserveonly'. They correspond to the options to *fsadm prealloc*.

3.3.1.2.3 *Execution Flow*

```
parse command options
allocate one advfs_ext_attr_t for each valid option
for each advfs_ext_attr_t
        ioctl(fd, ADVFS_SETEXT, &attr);
        check for error
        if error
                print error and continue
        free advfs_ext_attr_t
done
```

3.3.1.3 fsadm getattr

3.3.1.3.1 *Interface*

```
fsadm getattr -F advfs [-p] <file>
```

3.3.1.3.2 *Description*

The ***fsadm getattr*** command will accept the following new flags:

-p	Specify that the extent attribute the user wishes to query is file pre-allocated space.

The ***fsadm getattr*** command will be the generic interface for users to query file extent attributes. More options will be added in the future as requirements demand.

3.3.1.3.3 *Execution Flow*

```
parse command options
allocate one advfs_ext_attr_t for each valid extent option
for each advfs_ext_attr_t
        ioctl(fd, ADVFS_GETEXT, &attr);
        print attr.value;
        free advfs_ext_attr_t
done
```

3.3.2 Kernel Interface & Functions

3.3.2.1 advfs_ioctl

3.3.2.1.1 *Interface*

```
int
advfs_ioctl (
    struct vnode *vp,    /* in - vnode of interest */
    int           cmd,   /* in - Operation to be performed */
    caddr_t       data,  /* in/out - cmd dependent data in or out */
    int           fflag, /* in - file.f_flag from struct file */
    struct ucred *cred   /* in - Caller's Credentials */
);
```

3.3.2.1.2 *Description*

The ***advfs_ioctl*** function already exists. It will be modified to properly branch when the ioctl command is ADVFS_SETEXT or ADVFS_GETEXT. These additions will be made to the *advfs_ioctl.h* file:

```
#define ADVFS_SETEXT   _IOW(ADVFS_IOCTL, 5, struct advfs_ext_attr )
#define ADVFS_GETEXT   _IOWR(ADVFS_IOCTL, 6, struct advfs_ext_attr )
```

The ADVFS_SETEXT and ADVFS_GETEXT cases will verify the vnode is not on a read-only file system and is a regular file (VREG) before branching to *advfs_setext()* and *advfs_getext()* respectively.

3.3.2.1.3 *Execution Flow*

```
    switch(cmd) {

    case ADVFS_SETEXT:
    {
        /* must be a regular file */
        if (vp->v_type != VREG || VTOA(vp)->dataSafety != BFD_USERDATA) {
            retval = EINVAL;
            break;
        }

       /* can’t set attributes on a read-only filesystem */
       if (vp->v_vfsp->vfs_flag & VFS_RDONLY) {
               retval = EROFS;
               break;
       }

        /* now we can call advfs_setext() */
        retval = advfs_setext( vp, (advfs_ext_attr_t *)data );
        break;
    }
    case ADVFS_GETEXT:
    {
        /* must be a regular file */
        if (vp->v_type != VREG || VTOA(vp)->dataSafety != BFD_USERDATA) {
            retval = EINVAL;
            break;
        }

        /* now we can call advfs_getext() */
        retval = advfs_getext( vp, (advfs_ext_attr_t *)data );
        break;
    }
. . .
```

3.3.2.2 *advfs_setext*

3.3.2.2.1 *Interface*

```
mlStatusT
advfs_setext(
        struct vnode *vp,            /* in */
        advfs_ext_attr_t *attr       /* in */
);
```

3.3.2.2.2 *Description*

The function ***advfs_setext*** is used to set *one* extent attribute. If multiple attributes need to be set, this must be called within a loop. Its primary duty is to call the specific function associated with the extent attribute command being modified. The *advfs_ext_attr_t* is not modified.

3.3.2.2.3 *Execution Flow*

```
int
advfs_setext( struct vnode *vp, advfs_ext_attr_t *attr )
{
    int retval = ESUCCESS;

    switch(attr->type) {
    case ADVFS_EXT_NOOP:
        break;
    case ADVFS_EXT_PREALLOC:
        if (attr->value.prealloc.bytes >= 0) {
            retval = advfs_setext_prealloc(vp, &attr->value.prealloc);
        }
        break;
    default:
        return(ENOTSUP);
    }

    return(retval);
}
```

3.3.2.3 *advfs_getext*

3.3.2.3.1 *Interface*

```
mlStatusT
advfs_getext(
        advfs_ext_attr_type_t type,  /* in */
        advfs_ext_attr_t *attr       /* out */
);
```

3.3.2.3.2 *Description*

The function ***advfs_getext()*** is used to query *one* file extent attribute. If multiple attributes need to be queried, this must be called within a loop.

3.3.2.3.3 *Execution Flow*

```
int
advfs_setext( struct vnode *vp, advfs_ext_attr_t *attr )
```

```
{
    int retval = ESUCCESS;

    switch(attr->type) {
    case ADVFS_EXT_NOOP:
        break;
    case ADVFS_EXT_PREALLOC:
        retval = advfs_getext_prealloc(vp, &attr->value.prealloc);
        break;
    default:
        return(ENOTSUP);
    }

    return(retval);
}
```

3.3.2.4 *advfs_setext_prealloc*

3.3.2.4.1 *Interface*

```
int
advfs_setext_prealloc(
       struct vnode *vp,                   /* in */
       advfs_ext_prealloc_attr_t *prealloc /* in */
);
```

3.3.2.4.2 *Description*

The function ***advfs_setext_prealloc()*** sets up pre-allocated file storage for the file described by *vp*. It optionally modifies the BSR_ATTR mcell record to store reserved space for the file and updates the bfAccessT if the user has specified the ADVFS_EXT_PREALLOC_RESERVE_ONLY flag.

3.3.2.4.3 *Execution Flow*

```
int
advfs_setext_prealloc( struct vnode *vp, advfs_ext_prealloc_attr_t *prealloc )
{
    bfAccessT * bfap;
    domainT * dmnp;
    struct fsContext *contextp;
    struct advfs_pvt_param priv_param;
    fcache_map_dsc_t *fcmap;        /* UFC mapping for file */
    faf_status_t faultStatus = 0;
    ni_nameiop_t save_orig_nameiop;
    struct nameidata *ndp;
    size_t blks_needed;
    statusT error = 0;
    statusT error2 = 0;
    off_t offset = 0;
    size_t bytes = 0;
    int nozero = 0;             /* do not zero allocated space */
    int reserve_only = 0;       /* do not update the file_size */
    struct vnode *saved_vnode;
    uint64_t rlimit_fsize = 0;
    uint64_t file_size_limit = 0;

    bfap = VTOA(vp);
    dmnp = bfap->dmnP;
    contextp = VTOC(vp);

    /* first thing to do before continuing is to verify we own the file
     * to modify (or are root) */
    if (((kt_cred(u.u_kthreadp))->cr_uid != 0) &&
        ((kt_cred(u.u_kthreadp))->cr_uid != bfap->bfFsContext.dir_stats.st_uid)) {
        return (EPERM);
    }

    /*
```

```
 * Determine the maximum file size allowed by current ulimit() settings.
 */
rlimit_fsize = p_rlimit(u.u_procp)[RLIMIT_FSIZE].rlim_cur;

/*
 * The maximum file size this thread can write is the minimum of the
 * ulimit() setting for file size and the maximum offset that AdvFS
 * currently supports.
 */
file_size_limit = MIN(rlimit_fsize, advfs_max_offset + 1);

/* For the moment, we preallocate from offset 0 all the time, so
 * we only need to see if the bytes requested exceed the max file
 * size */
if (prealloc->bytes >= file_size_limit) {
    /*
     * Send the process a SIGXFSZ signal if we need to do so.
     */
    if ((prealloc->bytes > rlimit_fsize) &&
        (IS_XSIG(u.u_procp) || IS_SIGAWARE(u.u_kthreadp, SIGXFSZ))) {
            psignal(u.u_procp, SIGXFSZ);
    }
    return (EFBIG);
}

/* Check for the NOZERO flag, and verify that -- if set -- that the caller
 *  has root credentials
 */
if (prealloc->flags & ADVFS_EXT_PREALLOC_NOZERO) {
    if ((kt_cred(u.u_kthreadp))->cr_uid != 0) {
        return (EPERM);
    }
    nozero = 1;
}

/* determine if we are reserving space only, or if we are going to update
 * the file_size
 */
if (prealloc->flags & ADVFS_EXT_PREALLOC_RESERVE_ONLY) {
    reserve_only = 1;
}

/* Calculate the number of DEV_BSIZE disk blocks the caller wants to
 * preallocate.
 */
blks_needed = howmany(prealloc->bytes, DEV_BSIZE);

/* If howmany() told us zero blocks were needed but prealloc->bytes != 0,
 * then we probably have an overflow of the uint64_t.  Return EINVAL.
 */
if (prealloc->bytes != 0 && blks_needed == 0) {
    return (EINVAL);
}

/* Verify the domain has enough free DEV_BSIZE disk blocks for the request.
 * This is only a preliminary check as the domain's freeBlks may
 * change while performing the storage allocation.
 */
if (dmnp->freeBlks < blks_needed) {
    return (ENOSPC);
}

/* Advfs_getpage() via the fault requires the NI_RW flag be
 * set to determine caller is not mmapping and to allow file extentions.
 * Save the original nameidata value to restore later.
 */
ndp = NAMEIDATA();
MS_SMP_ASSERT(ndp);
save_orig_nameiop = ndp->ni_nameiop;
ndp->ni_nameiop = NI_RW;
saved_vnode = ndp->ni_vp;
```

```
ndp->ni_vp = vp;

/* initial values for getpage loop variables */
offset = 0;
bytes = prealloc->bytes;

/* Pass in the private parameter pointer so the advfs_getpage()
 * doesn't think this is a mmap file.
 */
bzero((char *)&priv_param, sizeof(struct advfs_pvt_param));
priv_param.app_starting_offset = offset;
priv_param.app_total_bytes = bytes;
priv_param.app_flags = APP_ADDSTG_NOCACHE;

ADVRWL_FILECONTEXT_WRITE_LOCK( contextp );

/* If prealloc->bytes is 0, just the set the rsvd file size to 0 in-memory
 * on on-disk in the BSR_ATTR record as this will "turn off"
 * any user reserved preallocated space */
if (prealloc->bytes == 0) {
    /* when the file is closed it will be truncated */
    reserve_only = 1; /* follow the reserve_only code path below */
    nozero = 0;  /* skip zeroing through raw_io */
}

/* if the file_size is not 0, we return EINVAL as we can only preallocate
 * for zero-length files.  The exception being if the bytes to preallocate
 * is 0, which will remove any reserved space */
else if (bfap->file_size != 0) {
    error = EFBIG;
    goto error;
}

/* call getpage to add storage.  This call to getpage bypasses
 * the UFC since we have no need to cache the pages we allocate.
 * We make this call in a loop since advfs_getpage() can only allocate
 * 2MB at a time due to log restrictions */
while( prealloc->bytes >
       ( (priv_param.app_stg_end_fob + 1) * ADVFS_FOB_SZ) ) {
    error = advfs_getpage( NULL,
                           &bfap->bfVnode,
                           (off_t *)&offset,
                           (size_t *)&bytes,
                           FCF_DFLT_WRITE,
                           (uintptr_t)&priv_param,
                           0 );
    if (error != EOK) {
        goto error;
    }
}

if (!nozero) {
    char *zeroed_memp = NULL;
    uint64_t extent_count = 0;
    uint32_t last_iosize = 0;
    off_t starting_fob = 0; /* start at beginning of file */
    struct vd *vdp = NULL;
    extent_blk_desc_t *fob_range = NULL, *fr = NULL;

    /* if nozero was NOT specified, we use raw_io to zero the storage.
     * First we need the migStgLk. */
    ADVRWL_MIGSTG_WRITE( bfap );

    /* get a list of real extents to zero */
    error = advfs_get_blkmap_in_range(bfap,
                                      bfap->xtnts.xtntMap,
                                      &priv_param.app_starting_offset,
                                      (priv_param.app_stg_end_fob + 1)
                                          * ADVFS_FOB_SZ,
                                      &fob_range,
                                      &extent_count,
                                      RND_NONE,
                                      EXB_ONLY_STG, /* storage only */
                                      XTNT_NO_WAIT);

    if (error != EOK || fob_range==NULL) {
        ADVRWL_MIGSTG_UNLOCK( bfap );
        goto error;
    }

    /* for each extent, call advfs_raw_io to zero it */
    for(fr = fob_range; fr != NULL; fr = fr->ebd_next_desc) {
        bf_vd_blk_t blocks_written = 0;
        bf_vd_blk_t starting_block = fr->ebd_vd_blk;
        bf_vd_blk_t blocks_to_write = fr->ebd_byte_cnt / ADVFS_FOB_SZ;
        vdp = VD_HTOP( fr->ebd_vd_index, bfap->dmnP );

        /* we should use the volume's preferred I/O size since we
         * know it, but we only malloc if the zeroed buffer is a
         * different size than the preferred_iosize -- this avoids
         * needless malloc's */
        if(last_iosize != vdp->preferred_iosize) {
            if(zeroed_memp != NULL) {
                ms_free(zeroed_memp);
            }
            zeroed_memp = ms_malloc(vdp->preferred_iosize);
            if (zeroed_memp == NULL) {
                error = ENOMEM;
                ADVRWL_MIGSTG_UNLOCK( bfap );
                error2 = advfs_free_blkmaps ( &fob_range );
                goto error;
            }
            last_iosize = vdp->preferred_iosize;
        }

        /* we loop and write some fixed amount of zeros each time --
         * this prevents a possible HUGE malloc.  The price is a
         * performance hit, but that's okay since someone really
         * concerned with performance would choose not to zero
         * this preallocated space */
        while (blocks_to_write > 0) {
            /* only write a MAX of vdp->preferred_iosize */
            if (blocks_to_write > (vdp->preferred_iosize / ADVFS_FOB_SZ)) {
                blocks_to_write = vdp->preferred_iosize / ADVFS_FOB_SZ;
            }

            error = advfs_raw_io ( vdp->devVp,
                                   starting_block,
                                   blocks_to_write,
                                   RAW_WRITE,
                                   zeroed_memp );

            if (error != EOK) {
                ADVRWL_MIGSTG_UNLOCK( bfap );
                error2 = advfs_free_blkmaps ( &fob_range );
                goto error;
            }

            /* now update the starting_block and blocks_to_write */
            starting_block += blocks_to_write;
            blocks_written += blocks_to_write;
            blocks_to_write = (fr->ebd_byte_cnt / ADVFS_FOB_SZ)
                              - blocks_written;
        }
    }

    ADVRWL_MIGSTG_UNLOCK( bfap );

    /* free the zeroed memory used for clearing on-disk storage */
    if(zeroed_memp != NULL) {
        ms_free(zeroed_memp);
    }
```

```
        /* free the blkmaps */
        error = advfs_free_blkmaps ( &fob_range );
        if (error) {
            goto error;
        }
    }

    if (reserve_only) {
        bsBfAttrT bfAttr = { 0 };
        ftxHT ftxH = { 0 };

        /* at this point we have allocated on-disk storage but it is not
         * persistent.  A call to fs_trunc_test will flag this file for
         * truncation, which is inappropriate if a user has asked for
         * reserved preallocated space.  So we update the metadata by
         * setting the BSR_ATTR record to indicate there is reserved
         * preallocated space. */

        if ( (error = FTX_START_N( FTA_BS_BMT_PUT_REC_V1,
                                   &ftxH, FtxNilFtxH, bfap->dmnP ) ) != EOK ) {
            goto error;
        }

        if ( (error = bmtr_get_rec_n_lk( bfap,
                                         BSR_ATTR,
                                         (bsBfAttrT *)&bfAttr,
                                         sizeof(bfAttr),
                                         TRUE ) ) ) {
            goto error;
        }

        bfAttr.bfat_rsvd_file_size = prealloc->bytes;


        if ( (error = bmtr_put_rec_n_unlk( bfap,
                                           BSR_ATTR,
                                           (bsBfAttrT *)&bfAttr,
                                           sizeof(bfAttr),
                                           ftxH,
                                           TRUE,
                                           0 ) ) ) {
            goto error;
        }

        /* now end the transaction */
        ftx_done_n( ftxH, FTA_BS_BMT_PUT_REC_V1 );

        /* now we can update the in-memory rsvd_file_size */
        bfap->rsvd_file_size = prealloc->bytes;
    }
    else {
        /* the user has asked us to preallocate some space and wants the
         * file_size updated. So we don't flag the tagdir or the bfap, since
         * the file won't truncate past the file_size */

        /* Update the file size to the requested size
         * only if it is larger than the previous file size.
         * Preallocate functionality always starts the preallocation from the
         * beginning of the file.
         */
        if (prealloc->bytes > bfap->file_size) {
            bfap->file_size = prealloc->bytes;
        }
    }

error:
    /* unlock the file context lock */
    ADVRWL_FILECONTEXT_UNLOCK(contextp);
    /* Restore the original value */
    ndp->ni_nameiop = save_orig_nameiop;
    ndp->ni_vp = saved_vnode;

    return (error2 ? error2 : error);
}
```

3.3.2.5 *advfs_getext_prealloc*

3.3.2.5.1 *Interface*

```
int
advfs_getext_prealloc(
       struct vnode *vp,                  /* in */
       advfs_ext_prealloc_attr_t *prealloc /* out */
);
```

3.3.2.5.2 *Description*

The function ***advfs_getext_prealloc()*** queries the amount of allocated on disk space. This is basically a front end to the existing function *bs_get_bf_fob_cnt()*.

3.3.2.5.3 *Execution Flow*

```
    return( bs_get_bf_fob_cnt( VTOA( vp ), &(prealloc->bytes) ) );
```

3.3.2.6 *bs_map_bf*

3.3.2.6.1 *Interface*

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

3.3.2.6.2 *Description*

The function ***bs_map_bf()*** sets up the in-memory bfAccessT structure. An addition has been made to map the new BSR_ATTR value for reserved space with a new flag in the bfAccess structure. This is used to prevent truncation when the file possesses reserved space without an updated file size (i.e. by using the ADVFS_EXT_PREALLOC_RESERVE_ONLY flag). With reserved storage, the file should never have less than the reserved amount allocated to it (it may have more storage).

3.3.2.6.3 *Execution Flow*

```
    …
    bfap->rsvd_file_size = bfAttrp->bfat_rsvd_file_size;
    …
```

3.3.2.7 *fs_trunc_test*

3.3.2.7.1 *Interface*

```
int
fs_trunc_test(
       struct vnode* vp
);
```

3.3.2.7.2 *Description*

The function ***fs_trunc_test()*** is used to determine whether the file should be truncated to the last used user data page. An addition has been made to indicate truncation should be performed only if the file has both a file size and a reserved file size smaller than the next FOB to be allocated.

3.3.2.7.3 *Execution Flow*

```
int
fs_trunc_test(struct vnode* vp)
{
    uint64_t next_alloc_unit_to_allocate,
             in_use_alloc_units,
             rsvd_alloc_units,
             file_alloc_unit_size;
    bfAccessT *bfap = VTOA(vp);
    bfSetT *bfSetp;
    struct fsContext* fileContext = VTOC(vp);


    if ( (fileContext == NULL) ||
         (fileContext->fs_flag & META_OPEN) ||
         (bfap->dataSafety != BFD_USERDATA) ) {
        /*
         * This file was opened thru the tag interface.  Its stats
         * area is not initialized so ignore it.  Alternately, the file is
         * metadata and file_size is not initialized.
         */
        return 0;
    }

    bfSetp = bfap->bfSetp;

    next_alloc_unit_to_allocate = bfap->bfaNextFob / bfap->bfPageSz;

    file_alloc_unit_size = bfap->bfPageSz * ADVFS_FOB_SZ;

    in_use_alloc_units  = (bfap->file_size + file_alloc_unit_size - 1L) /
                          file_alloc_unit_size;

    rsvd_alloc_units = (bfap->rsvd_file_size + file_alloc_unit_size - 1L) /
                       file_alloc_unit_size;

    return(bfap->trunc =
           ( (next_alloc_unit_to_allocate > in_use_alloc_units) &&
             (next_alloc_unit_to_allocate > rsvd_alloc_units) ) );
}
```

3.3.2.8 *bf_setup_truncation*

3.3.2.8.1 *Interface*

```
statusT
bf_setup_truncation (
        bfAccessT *bfap,     /* in */
        ftxHT ftxH,          /* in */
        void **delList,      /* out */
        uint32_t *delCnt     /* out */
)
```

3.3.2.8.2 *Description*

This function deallocates storage if there are any allocation units completely unused. This is, however, not what should be done if the user has specified that the file should contain reserved storage. And addition has been made to only deallocate storage up to the either the last used allocation unit or the last reserved allocation unit.

3.3.2.8.3 *Execution Flow*

```
statusT
bf_setup_truncation (
                     bfAccessT *bfap,     /* in */
                     ftxHT ftxH,          /* in */
                     void **delList,      /* out */
                     uint32_t *delCnt     /* out */
                     )
{
    statusT sts = EOK;
    bf_fob_t fobs_used = 0;
    bf_fob_t fobs_rsvd = 0;
    bf_fob_t fobs_to_keep = 0;

    *delCnt = 0;

    fobs_used = (bfap->file_size + ADVFS_FOB_SZ - 1L) / ADVFS_FOB_SZ;
    fobs_rsvd = (bfap->rsvd_file_size + ADVFS_FOB_SZ - 1L) / ADVFS_FOB_SZ;

    /*
     * stg_remove_stg_start expects fobs to be aligned on allocation unit
     * boundaries so we need to round fobs_used up to the next
     * bfap->bfPageSz boundary.  fobs_rsvd should always be aligned on
     * allocation unit boundaries
     */
    fobs_used = roundup( fobs_used, bfap->bfPageSz );
    fobs_to_keep = max( fobs_used, fobs_rsvd );

    if ( fobs_to_keep < bfap->bfaNextFob ) {

        /* We hsould be truncating in full allocation unit for this file */
        MS_SMP_ASSERT( (bfap->bfaNextFob - fobs_to_keep) % bfap->bfPageSz == 0 );
        sts = stg_remove_stg_start (
                                    bfap,
                                    fobs_to_keep,
                                    bfap->bfaNextFob - fobs_to_keep,
                                    1,          /* do rel quotas */
                                    ftxH,
                                    delCnt,
                                    delList,
                                    TRUE,      /* do COW */
                                               /* force alloc of mcell in */
                                    TRUE       /* bmt_alloc_mcell */
                                    );

    }

    return sts;
}
```

4 Dependencies

Any dependencies are noted below. The majority of this functionality is self contained and is not dependent upon non-standard portions of an HP-UX system.

4.1 File System Layout

- Pre-allocated storage is not dependent upon file system layout.

4.2 File Systems

- User file pre-allocation operations take place with the standard storage allocation interfaces in AdvFS. No new issues will be created as concern multi-volume AdvFS file systems.

4.3 I/O System and Drivers

- AdvFS must be present in the kernel for the *ioctl()* to succeed.

4.4 Auditing

- Space consumed by pre-allocated file storage is visible through the standard Unix command *du*.

4.5 Behavior in a cluster

- Functionality should be transparent to a cluster.

4.6 Commands

- Necessary commands are documented within this design document.

4.7 Update/Rolling Upgrade

- Pre-allocated storage is either flagged on disk (in the tag directory) or preserved with an updated file size and therefore should not be affected by an update/rolling upgrade

4.8 Learning Products (Documentation)

- Man pages for *fsadm* must be updated.

5 Issues

5.2 Additional work for recovery

- There is work necessary to recovery tools in order that they recognize file pre-allocation (if pre-allocation is to be preserved during a recovery or file system dump, restore, or salvage).