A General Purpose Extended Attribute Service As A File System Layer

(with an example client that implements POSIX.6 Access Control Lists)

Comprehensive Project Documentation
for the Master of Science Degree

Jeff R. Weidner

Advisor: Gerald J. Popek

Computer Science Department

School of Engineering and Applied Sciences

University of California, Los Angeles

May 28, 1997

My master's project is an operating system extension called the EAFS (or extended attribute file system). It extends the UNIX file system to allow creation and manipulation of values associated with files (separate from the file's contents). This is an important capability for creating other new file system features. Without an attribute capability, each developer of a new file system feature is forced to provide their own ad-hoc attribute service. This lack of a solution has meant great effort and expense to create new services, thereby hindering innovation.

To prove my attribute service indeed does greatly aid novel file service development, I developed a new file system service from scratch in nine days. This new service is called the ACLFS (Access Control List File System). It provides a more flexible security model to file access by allowing a file's owner to grant or deny access permissions to individuals they name rather than to a pre-defined group. The security model I chose is a standard called POSIX.6 ACLs.

The documentation I wrote for this project is presented in the following sections:

Acknowledgments:

- Jerry Popek and the rest of the Ficus project for support and guidance.
- Ted Kim (an old office-mate) for lots of listening, help, and ideas.
- John Heidemann for the i405 interface upon which EAFS and ACLFS are built.

ACLFS User's Guide

Last Modified 05/03/97 Jeff Weidner

INTRODUCTION:

The ACLFS (Access Control List File System) implements POSIX.6 style ACLs. Each file (and directory) in an ACLFS has associated with it an ACL. Each ACL contains a number of entries, each of which consist of an id of a user or group and the access settings for that entity. For backwards compatibility, each file is required to have three entries in its ACL, one each for the file's owner, for the file's group, and for the general public. These entries correspond with the standard UNIX permissions arrangement. The access settings in each entry specify read, write, and execute privileges.

Access to a file is determined through evaluation of two factors: the ACL of the file, and the id and gid of the process accessing the file. This evaluation follows the algorithm specified by POSIX.6. The algorithm searches the ACL for a match between the process's uid (or gid) and each ACL entry's id. The search proceeds in the following order: the owner of the file, named users, the group of the file, named groups, and the general public 'other'. The process stops at the first match and allows or denies access based on the permission settings for the matching ACL entry.

FUNCTIONALITY:

Once the ACLFS is setup and initialized, its use can be transparent to the administrator, file owners, and users. Each file access within the ACLFS volume will be subject to the access restrictions and privileges as specified in each file's ACL. And, since newly created files and directories automatically inherit their ACLs from their parent directory, no extra commands are required until a change needs to be made. Once a shared file hierarchy is properly setup, it should not require additional maintenance until specific access policies change.

When access policies do change, they can be administered using the utility programs for the ACLFS: getfacl and setfacl. See the section "Utilities" below for a description of their use.

SETUP:

The ACLFS is implemented as a Stackable Layer, and its use requires a support layer, the EAFS (extended attribute file system), to be mounted beneath it. The EAFS provides the per-file attribute storage required to maintain and to access the ACLs. See the EAFS User Guide for the details of setting up an EAFS volume.

The process of mounting the filesystem stack for the ACLFS is as follows:

```
su                                ; obtain root privilege
mount_ea -C /raw.eafs.dir /mnt/eafs   ; mount the EA layer
mount_acl /mnt/eafs /mnt/aclfs        ; mount he ACL layer
exit                              ; relinquish root privilege
cd /mnt/aclfs                     ; change dir to the ACL volume
```

UTILITIES:

There are two utility programs associated with the ACLFS. These programs are called 'getfacl' and 'setfacl'. They are used for inspecting, and setting ACLs. These programs are modeled after the SunOS 5.5 programs of the same name and use the same ASCII representation for ACL's and their entries. Please read the man pages in /h/sys/aclfs/docs/SunOS for the details.

Usage is as follows:

```
getfacl file [file2 ...]

setfacl -f aclfile file [file2 ...]
```

The ASCII format for ACLs follows the following form:

```
# file: <filename>
# owner: <uid>
# group: <gid>
user::<perm>
user:<uid>:<perm>
group::<perm>
group:<gid>:<perm>
other:<perm>
```

where:

```
#            = comment line escape character
                   (info on rest of line is only a comment)
<filename>   = the filename the ACL applies to
<uid>        = a login name or numeric user id
<gid>        = a group name or numeric group id
<perm>             = permission settings,
                   either in 'rwx' format
                         where the letters indicate 'on'
                         and a dash "-" indicates 'off'
                         r = read
                         w = write
                         x = execute
                   or an octal digit where the bits are:
                         1 = 'x' = execute
                         2 = 'w' = write
                         4 = 'r' = read
```

This ASCII ACL format is the output of 'getfacl' and the input to 'setfacl'. Lines which begin with a hash mark '#' are ignored as comments. Each line is an element of the list. The output of 'getfacl' begins with three comment lines which indicate the filename, owner, and group of the file. This information is contained in comments so that the rest of the ACL can be copied to another file (to facilitate the reuse of ACLs - see the example of copying an ACL below). The lines which begin "user::", "group::", and "other:" represent the obligatory 'owner', 'group', and 'other' permissions settings and must appear exactly once per ACL. Lines which begin "user:<uid>" and "group:<gid>" represent named users and groups and can appear many times each (once for each <uid> or <gid>).

An example ASCII ACL file might look like this:

```
# file: foo
# owner: jeffw
# group: acldev
user::rwx
user:tek:rw-
user:johnh:r--
user:satan:---
group::r--
other::---
```

This ACL example indicates the access permissions from file 'foo' which is owned by 'jeffw' and is in group 'acldev'. It also specifies an ACL whose entries allow all permissions of the file's owner, read and write permissions for user 'tek' and read permission for user 'johnh'. Users who belong to group 'acldev' also have read permission. Everyone else is denied any access. The interesting part of this example are the named users, 'tek', 'johnh', and 'satan'. The 'tek' entry allows a specified user more access than the 'group' setting allows, but less than the 'owner' entry allows. Similarly, it allows user 'johnh' read permission even if he is not part of the 'acldev' group. The entry for 'satan' disallows any access to this user even if the user is part of the 'acldev' group.

Note that the above utilities do not currently support all of the options available in the SunOS versions. In particular: there is no support for 'default' or 'mask' ACL entries, 'getfacl' does not support any options, 'getfacl' merely outputs the ACLs of the named files, and 'setfacl' only supports the `-f` option. However, 'setfacl' DOES support STDIN via '-f -', so you may copy an ACL from one file to other(s) with the following command:

```
getfacl file1 | setfacl -f - file2
```

ACLFS Technical Guide

Last Modified 05/03/97 Jeff Weidner

ACL Algorithm

The algorithm used by the ACLFS is that specified by POSIX.6. That specification requires that the access checking proceeds in a certain order and that the first match between a process' credentials and a file's ACL resolves the access determination and terminates the algorithm. The order of checking proceeds as follows:

```
Owner of file
User named by ACL
Group of file
Group named by ACL
Other
```

Thus, if the process attempting access to a file has credentials for a user who is both named in the file's ACL and in the file's group, only the entry regarding the named user is considered. If the access for the named user is denied, then access is denied regardless of the setting for the group to which the user belongs. This is true in the reverse case when access is granted, too. In this case, the entry(s) for the file's group(s) is irrelevant, even though the user belongs to it.

For a more detailed description, see the source code and/or the POSIX.6 specification.

ACL Data Structures

The basic unit of data in the ACLFS is an "acl_entry", for these are the items that make up an access control list. Each acl_entry specificies two things: an entity and access permissions settings that apply to that entity.

An entity can be either a specific "user" or a "group". A "group" refers to a list of users that are defined to the system by a member list. If the entity refers to a specific user, than the "id" field of the acl_entry is that user's UID. Similarly, if the entity refers to a group, than the "id" field of the acl_entry is that of the group's GID. Bits in the "tag_and_mode" field of an acl_entry show whether the id refers to a USER or a GROUP.

The access permissions of an acl_entry are specified by three bits in the "tag_and_mode" field. These three mode bits signify READ, WRITE, and EXECUTE permissions (respectively) and are aligned to coincide with the mode bits defined in "inode.h". If a permission bit is a one, then that permission is granted to the entity. If a permission bit is zero, then that permission is denied to the specified entity.

```
struct acl_entry {
     u_short id;       // the UID or GID that entry applies to
     u_short tag_and_mode;   // type of 'id' & mode bits of entry
};

#define ACL_TAG_USERS   0100000   /* signals id is a UID */
#define ACL_TAG_GROUPS  0040000   /* signals id is a GID */
#define ACL_MODE_IREAD  0000400   /* aligned to match inode.h */
#define ACL_MODE_IWRITE 0000200   /* aligned to match inode.h */
#define ACL_MODE_IEXEC  0000100   /* aligned to match inode.h */

enum _idtype { M_USER = 15, M_GROUP, M_OTHER};
```

Function Description and Prototypes

This section lists the significant subroutines which together make up the bulk of the ACL service. The entry point to this code is through acl_dac_alg() which is used prolificly throughout the implementation of the vnodeops. A typical call from a vnodeop goes, for instance, like this:

```
/* require write and execute permissions on ap->a_vp */
mode = ACL_MODE_IWRITE | ACL_MODE_IEXEC;
rv_error = acl_dac_alg(ap->a_vp, mode, ap->a_cred);
if (rv_error) return (rv_error);
```

ACL_ACLSUBS.C

```
/* acl_dac_alg
 *
 * Discretionary Access Control Algorithm
 *     (as specified by POSIX.6)
 * check proceeds in these steps:
 *     (stop after first applicable match)
 *  OWNER   file's owner
 *  USERS   ACL's named users
 *  GROUP   file's group
 *  GROUPS  ACL's named groups
 *  PUBLIC  finally, file's public permissions settings
 */
int
acl_dac_alg(vp, mode, pcred)
     register struct vnode *vp;
     register int mode;
     register struct ucred *pcred;


/* acl_iaccess
 *
 * This is the ACL version of iaccess.  It is based on
 *  the official iaccess as in UFS, but it only checks
 *  one entity at a time (OWNER, GROUP, PUBLIC).
 *  We need to check only one at a time in order to
 *  implement the proper
 * Discretionary Access Control Algorithm as specified
 *  in POSIX.6 ACLs.
 */
int
acl_iaccess(vp, m, entity, pmatched, pva)
     register struct vnode *vp;
     register int m;               /* mode bits to check */
     enum acl_iaccess_entities entity; /* one of OWNER, GROUP,
PUBLIC */
     register int *pmatched;          /* returns whether we
matched or not */
     struct vattr *pva;
```

```
/* acl_aclaccess
 *
 * Look into the ACL for this file to try to find an
 *  entry referring to the current user.  Set *pmatched
 *  whether we found a relevant entry.
 * Return 0 for access granted
 *  or error (usually EACCES) for deny.
 */
int
acl_aclaccess(vp, mode, entity, pmatched)
     register struct vnode *vp;
     register u_short mode;
     enum acl_iaccess_entities entity;
     int *pmatched;


/* These *assume_root* func's are copied from ficus_ik_subr.c
 * They are used here in the ACLFS to allow access to named
 *  users and groups which under normal permission would be
 *  denied. So we promote their cred to root to bypass the
 *  lower layers' old style security scheme.  Obviously,
 *  it's important to revoke this privilege
 * on the way back up the call stack.
 */
void
acl_assume_root_uid(oldcredp)
        struct ucred **oldcredp;


void
acl_unassume_root_uid(oldcred)
        struct ucred *oldcred;


/* copy_acl
 *
 * copies an ACL from <svp> to <tvp> masking out bits
 *  specified by <mask>
 *   (e.g. the mask is used to drop execute bits from
 *    a directory to a new file)
 */
void
copy_acl(mask, svp, tvp)
     u_int mask;
     struct vnode *svp;
     struct vnode *tvp;
```

EAFS User's Guide

Last Modified 05/03/97 Jeff Weidner

INTRODUCTION:

The EAFS (Extended Attribute File System) provides a general way to associate arbitrary extra information with each file. This ability greatly simplifies the effort required to develop novel extensions to the UNIX file system. Together with the i405 stackable layers system, the EAFS creates an environment where a programmer can develop new filesystem capabilities in a matter of days rather than many months or even years. As proof of this statement, I single-handedly developed the ACLFS (an Access Control List File System) in about nine days.

The targeted use for the EAFS is by file system layers stacked above it. Although not the primary intended use, in addition, the EAFS is usable by programs and end-users through the i405 OOK interface and utility programs, respectively.

SETUP:

The EAFS is implemented as an i405 Stackable Layer and requires some special setup of the layer beneath, the UFS. This special setup entails creating a "raw storage" area and initializing the EAFS's persistent storage files there. In addition, the EAFS requires a kernel that supports the following modifications (in addition to I405 and EAFS): IACCESS and UFSATTR.

To create and initialize a new EAFS volume, follow these steps:

```
su                               ; obtain root privilege
mkdir /raw/eafs.dir              ; create the EAFS raw storage
mkdir /mnt/eafs                  ; create the EAFS mount point
cd /raw/eafs.dir                 ; change to raw storage dir
touch .eautoc                    ; create database files
touch .eactou
touch .eactoi
touch .eastoc
touch .eactos
touch .eacands
/h/sys/eafs/trunc8192 .eautoc     ; alloc space for DB files
/h/sys/eafs/trunc8192 .eactou
/h/sys/eafs/trunc8192 .eactoi
/h/sys/eafs/trunc8192 .eastoc
/h/sys/eafs/trunc8192 .eactos
cp /h/sys/eafs/dbfiles/.easb .    ; initialize superblock file
exit                              ; relinquish root privilege
```

To mount the EAFS created above, follow these steps:

```
su                                     ; obtain root privilege
mount_ea -C /raw/eafs.dir /mnt/eafs    ; mount the EA layer
exit                                   ; relinquish root
cd /mnt/eafs                           ; change dir EAFS
```

UTILITIES:

There are five main utility programs available for accessing the functionality of the EAFS. These functions are:

```
easetval    Creates and/or sets attribute values for a file
eagetval    Gets attribute values for a file
easetinfo   Sets meta data for an attribute
eagetinfo   Gets meta data for an attribute
eadel       Deletes an attribute from a file
```

The command line arguments for these functions are of these forms:

```
easetval  <file> <num> <uuid> <val> (etc.)
eagetval  <file> <num> <uuid> (etc.)
easetinfo <file> <num> <uuid> <type>.<info>.<interp>.<prior> ...
eagetinfo <file> <num> <uuid> (etc.)
eadel     <file> <uuid>
```

where:

```
<file>     the name of the file to access
<num>      the length of the attribute array being submitted:
<uuid>     attribute ID of the form: 9876543210a5.d.d.d.d.f.f.f.f
<val>      the value to store (can not contain whitespace)
<type>     attribute meta data: value from 1-255
<info>     attribute meta data: value from 1-255
<interp>   attribute meta data: value from 1-255
<priority> attribute meta data: value from 1-255
...        the rest of attribute array (placeholder)
```

USER PROGRAM API:

The user program API to the EAFS VOPs is defined by the EAFS interface definition file (ea_interface.int) and the i405 OOK interface. Below we show by example some specifics of how this interface is used. For more examples, see the source code of the above utilities (in /h/sys/eafs/tests/<program>.c). The example shown below is taken from eagetval.c. Error checking has been removed for clarity and brevity.

```
/*  parse attrid
 *  converts a string "dotted UUID" into
 *  a nca_uuid structure
 */
void
parse_attrid(string, attrid)
     char *string;
     struct nca_uuid *attrid;

/* hex_dump
 * print a hexdump of "buf" for length "len".
 * For showing attribute values
 */
void
hex_dump(buf, len)
     char *buf;
     int len;
```

```
/*
 * EAGETVAL
 * takes command line of form:
 *   eagetval  <file> <num> <uuid1> <uuid2> ... <uuidN>
 * outputs a hexdump of the attributes requested
 */
int
main(argc, argv)
     int argc;
     char **argv;
{
     int error, i, numelems;
     struct vnode *vp;              /* the vnode to operate on  */
     struct nca_uuid attrids[512]; /* a large number of ids    */
     struct attrelem *attrelems;   /* ptr to list of attr vals */
     struct avhack attrvalarg;     /* struct needed by intfc   */
     USES_EAVOP_GETEXATTRS;        /* the i405 EAFS macros     */

#define DISKBLKSIZE 8192
     /* allocate space for the returned attributes */
     attrelems = (struct attrelem *) malloc(DISKBLKSIZE);
     bzero((char *)attrelems, DISKBLKSIZE);
     struct_avhack_a_avbuf_space(&attrvalarg) = DISKBLKSIZE;

     /* prime the attrelem argument */
     struct_avhack_a_avbuf(&attrvalarg) = (u_char *) attrelems;

     /* lookup the file to operate on */
     ook_lookupname(argv[1], NO_FOLLOW, NULL, &vp);

     /* parse the attribute ids off the command line */
     sscanf(argv[2], "%d", &numelems);
     for(i = 0; i < numelems; i++) {
           parse_attrid(argv[3+i], &attrids[i]);
     }

     /* call the vnode operation to get the attribute values */
     EAVOP_GETEXATTRS(vp, numelems, attrids, &attrvalarg);

     /* examine the returned attributes */
     numelems = struct_avhack_a_avbuf_elems(&attrvalarg);
        printf("found (approximately) %d\n",
            (int)(numelems/sizeof(struct attrelem)));
        hex_dump(attrelems, numelems);

     /* close things up */
     ook_vn_rele(vp);
     return 0;
}
```

EAFS Technical Guide

Last Modified 05/03/97 Jeff Weidner

Introduction:

The standard UNIX file system provides for associating meta-data values with each file (these meta-data values are called "file attributes"). However, it only provides this ability for a small handful of specific values. These handful of attributes are closely related to the semantics of the standard UNIX file system, and, being so, can not be reused for other purposes without drastically affecting existing software and systems.

Creating new file system services which need additional file attributes has meant the need to spend significant effort to create ad-hoc file attribute mechanisms. These ad-hoc attribute mechanisms are expensive and time-consuming to develop and can result in file service extensions which are incompatible with one another.

Functionality:

The EAFS (extended attribute file system) solves these problems (expense, delay, and compatibility) by providing a backwardly compatible, general-purpose file attribute service with a single, simple interface and decentralized attribute naming and definitions.

The EAFS is backwardly compatible because it does not alter any of the existing file system functionality. Its implementation as new VOPs (vnode operations) in an i405 layer on top of the UFS makes compatibility simple. All of the original filesystem operations are still available, and unaltered.

It is general purpose because it allows run-time creation of any new attribute with any value for any file. Its single, simple interface is a small collection of VOPs which provide for attribute manipulation (creating, setting, getting, deleting, mapping names, etc).

Attribute definitions are easily decentralized through the use of NCA UUIDs as the ultimate unique names for attributes.

Mechanism:

The operations of adding attributes, changing attribute values, looking up attribute values, and deleting attributes from files are enabled through the use of several new vnode operations, or VOPs. Specifically, these VOPs are:

```
ea_getexattrs()          // get attributes for a file
ea_setexattrs()          // set attributes for a file
ea_delexattrs()          // delete attributes from a file
ea_getexattrinfo()       // get meta-data for an attribute
ea_setexattrinfo()       // set meta-data for an attribute
ea_exattr_reg_strtoid()  // register a name <--> id mapping
ea_exattr_strtoid()      // lookup an attr id from its name
ea_exattr_idtostr()      // lookup an attr name from its id
```

For more information about each of these operations, refer to the sections titled: "List of VOPs and Pseudo Code of Functions".

Data Structures (as used in interface):

```
/*
 * extended attribute structure.
 *      This structure just defines the front part.  The
 *      last part (the actual buffer) can be any length.
 */
struct attrelem {
  struct nca_uuid as_nameid; /* global id for the attribute    */
        u_long  as_length;   /* total length of attrelem       */
        char as_value[MINEXATTRVALLEN];  /* value of attribute */
        /* NOTE:
         *  the structure may actually be longer than this,
         *  since the value can be any length)
         */
};

/* extended attribute meta-data structure. */
struct attrinfo {
        struct nca_uuid ai_nameid; /* global id  the attribute */
        u_long  ai_type;
        u_long  ai_info;
        u_short ai_interp;   /* hint how to interpret attribute*/
        u_short ai_priority; /* determines if in fast attr area*/
};
```

Data Structures (as used internally):

```
struct ea_info_map_elem {
        caid_t  caid;  /* compact attribute identifier */
        unsigned long type;
        unsigned long info;
        unsigned short interp;
        unsigned short prior;
};


struct ea_val_map_elem {
        caid_t  caid;  /* compact attribute identifier */
        u_long length; /* total length of ea_val_map_elem
                        * (incl all of value) */
        char val[4];   /* the attribute value */
       /* NOTE:
         *  the structure may actually be longer than this,
         *  since the value can be any length)
         */
};
```

```
/*
 * this is the simple way we currently map
 *   attrids <--> strings
 * at this point, we're just linear searching through
 * a small file to access stringnames
 */
struct ea_str_map_elem{
        struct nca_uuid uuid;      /* global id the attribute  */
        unsigned short rec_length; /* to get to next entry    */
        char name[2];              /* must terminate with null */
       /* NOTE:
        *  the structure may actually be longer than this,
        *  since the value can be any length)
        */
};

struct ea_caid_map_elem {
        struct nca_uuid uuid;      /* global id the attribute */
        caid_t  caid;  /* machine identifier for attibrute    */
};
```

Mappings and Mapping Structures:

The EAFS often needs to map values from one namespace to another. The reason for mapping values between namespaces is to save space and to save time. For instance, the names for attributes used by humans and between computers are large. Humans prefer long, descriptive text names. To share names between computers requires the use of a UUID (16 bytes) to eliminate ambiguities. If any of these large names were used for each instance of a file attribute, much time and diskspace would be wasted on redundant information. To solve this problem, the EAFS maps large attribute names to shorter 4-byte CAIDs (Compact Attribute Identifier). The mechanism it uses to accomplish this mapping is called the 'Map Manager'.

The Map Manager generalizes mapping functionality to provide a simple, consistent interface and to provide services for expanding the map while continuing to provide quick access. The Map manager handles the three most important mappings:

```
UTOC  (UUID to CAID:    most important, used often)
CTOU  (CAID to UUID:    used by get_exattrs("all") )
CTOI  (CAID to AttrInfo: to access attribute meta-data)
```

These three mappings are the most important because they will be used when performance matters most: as an integral part of the regular operations of another file system (such as the ACLFS).

The Map Manager provides the following operations:

```
Query
Insert
Delete
GetNext
Replace
```

Mapping between descriptive text attribute names and CAIDs will usually be at the request of an interactive user program, so performance is less important. A delay of even a few hundred milliseconds to perform the necessary lookups will not even be noticed by the user. Also, since these descriptive text names may be variable length, they do not easily fit into the structure of the Map Manager. For these reasons and justifications, the mapping of attribute names to attribute identifiers is handled separately by the 'CANDS table'.

CANDS Table

The CANDS table maps both ways between UUIDs and descriptive text names for attributes. The on-disk structure of this table is like that used for directory entries (to support variable record lengths). Each entry is structured as follows:

```
int   rec_length  ; total length of record, inclusive
UUID  uuid        ; NCA_UUID
char *name        ; NULL term. padded to 4-byte boundary
```

Access to the CANDS table is provided through the following functions (see outline of algorithm in "Pseudo Code of Selected Functions", below):

```
ea_exattr_reg_strtoid() ; register a new mapping
ea_exattr_strtoid()     ; look up an id by name
ea_exattr_idtostr()     ; look up a name by id
```

Other Structures:

Policy Table

The policy table is used to store the criteria to determine which attributes are stored in the fast attribute storage area. These criteria are a priority level for the attribute and the length of the particular attribute. The structures that make up this table are:

```
u_long      priority_val
caid_t      caid
void *      val_ptr
```

where priority_val is used to sort the table and is made up of two parts: the priority of the attribute (from the AttrInfo value), and the length of the value. When sorted, the smallest of the most important attributes are stored in the fast attribute area. This arrangement's goal is to maximize the performance gains facilitated by the use of the fast attribute area.

Super Block

Each EAFS volume requires a number of volume-specific files which are used to store persistent volume variables and the databases for the various mappings. These variables and databases are located either in or by the volume's 'EA Super Block'. Currently the 'EA Super Block' (and the related database files) is implemented as a well-known named file (".easb" in the volume's root directory). The main work of creating a new, empty EAFS volume is initializing the super block and DB files. When the volume is mounted, the superblock and DB file information is loaded into an ea_info structure (a standard part of each I405 stackable layer).

```
struct superblock {
  u_long version;    /* superblock version number */
  u_long safe_caid; /* non-volatile safecaid > lastcaid */
  ino_t  utoc_fid;  /* the mapping DB files */
  ino_t  ctou_fid;
  ino_t  ctoi_fid;
  ino_t  cands_fid;
};
```

List of VOPs and Pseudo Code of Functions:

This section lists the vnodeops which provide the interface to the EAFS services. Each routine is described through the use of pseudo-code.

```
ea_getexattrs()         // get attributes for a file
ea_setexattrs()         // set attributes for a file
ea_delexattrs()         // delete attributes from a file
ea_getexattrinfo()      // get meta-data for an attribute
ea_setexattrinfo()      // set meta-data for an attribute
ea_exattr_reg_strtoid() // register a name <--> id mapping
ea_exattr_strtoid()     // lookup an attr id from its name
ea_exattr_idtostr()     // lookup an attr name from its id
```

EA_VNODEOPS.C

```
ea_getexattrs()
     validate attr parameters
     get the fast attribute storage's vnode
     read the fast attr storage into 'fast_attr_space' buffer
     retrieve the requested uuids (go through the request list)
           map the uuid to a caid
           look for this caid in the 'fast_attr_space'
           if found, put in 'outbuf' and
                 continue with next requested attr
           get the vnode of the large attribute storage area
           map a page of the large storage area into 'attr_buf'
                 look for this caid in the 'fast_attr_space'
                 if found,
                       put attr's value in 'outbuf'
                       continue with next requested attr
                 map next page and look there

ea_setexattrs()
     validate attr parameters
     get the fast attribute storage's vnode
     read the fast attr storage into 'fast_attr_space' buffer
     create a new 'policy table'
           put attrs in fast area into policy table
           put incoming attrs into policy table
     if cumulative length of policy table attrs > fast area
           put large attrs in policy table, too
                 get vnode of large attr area
                 if there isn't large attr storage, create some
                 setup 'large_buf' as pointer to large attr area
     sort the policy table
           (find the most important attrs to fit into fast area)
     copy most important attrs that will fit into fast attr area
     write out fast attr area
     if there are any attrs left
           copy them into the new large attr area
           write out the larage attr area
```

```
ea_getexattrinfo()
     validate parameters
     go through list of attrs to get info for:
           map UUID to CAID
           map CAID to Info
           copy Info to 'outbuf'
     return 'outbuf'

ea_setexattrinfo()
     validate parameters
     go through list of attrs to set info for:
           map UUID to CAID
           if mapping did not exist, create it.
           write Info out by CAID

ea_delexattrs()
     validate parameters
     get vnode for fast attr storage
     read fast atts into 'fast_attr_space'
     go through the list of attrs to delete
           map UUID to CAID
           look in 'fast_attr_space' for attr to delete
           if found
                 mark as logically deleted
           if not found
                 ;look in large attr storage area for this attr
                 get vnode of large attr area
                 map in the large attr area
                 look for this attr to delete
                 if found, mark as logically deleted
     write out storage areas

ea_exattr_reg_strtoid()
     validate parameters
     get length of CANDS DB file
     create a new CANDS record
           (format is {rec_length, UUID, name})
     append record to end of CANDS DB file

ea_exattr_strtoid()
     validate parameters
     get length of CANDS DB file
     read in a diskblock of records
     go through records one by one
     if record's 'name' matches the name we're looking for,
           copy record's UUID to the outbuffer

ea_exattr_idtostr()
     validate parameters
     get length of CANDS DB file
     read in a diskblock of records
     go through records one by one
     if record's 'uuid' matches the uuid we're looking for,
           copy record's name to the outbuffer
```

Analysis Of The EAFS And Its Environment

Last Modified 05/28/97 Jeff Weidner

Architecture:

My foremost thought about the architecture in which I have implemented my project is that it is not as flexible as it ought to be. The EAFS and I405's goals are to provide a general, flexible, and modular method to file system development. However, their success in achieving these goals is greatly diminished because they both are atop the "standard" UFS.

Because the "monolithic" UFS combines many things (a flat storage service with provisions for locality, an attribute service, a security mechanism and policy, and hierarchical naming) into one set of code, it is impossible to replace or omit any of them. We are forced to work with or to work around all of these aspects of the UFS. (For example, the ACLFS, EAFS, FLFS, and FPFS all need at some point to "assume root" identity in order to bypass UFS security).

To provide a truly flexible file system development environment, we need to decompose the UFS into several layers:

```
Flat Storage          (Inode/NewFID based file storage)
Attributes            (standard UFS and/or EAFS)
Security              (standard UFS and/or ACLFS)
Hierarchical Naming   (standard UFS and/or Ficus Replication)
```

Much of this work has already been accomplished. The basic functionality of the flat storage system is available through the IACCESS kernel option. The EAFS provides an attribute service. Security can be provided by the ACLFS (which could support "standard" security by merely commenting out a few lines). The remaining work to be done is:

1. expose the IACCESS capabilities independently of existing UFS semantics.
2. extend the EAFS to support the "standard" attributes
   (support `stat()`, `chown()`, `chmod()`, `link()`, `utimes()`, etc.
   by implementing VOPs `access()`, `setattr()`, `getattr()`, etc.)
3. top it all off with a hierarchical naming service
4. and glue it all together.

Taking these steps would create a truly flexible and modular file system development environment unhindered by the intertwined semantics of the UFS.

Why Our Architecture Has Not Been Adopted:

I believe the reason why our architecture has not been adopted is a combination of the following:

- I405 needs to be adopted in order for Ficus, EAFS, ACLFS to work
- Demise of the CSRG and the BSD hacker community as likely allies
- Migration of university C.S. departments away from UNIX
- We don't have enough of a "killer app" to make people convert
- Modern commercial UNIXs are mature and entrenched

To Make Our Work (I405, EAFS, Ficus, etc.) Successful:

- Need to provide several new, useful, stable, and easy (install and use) file services

  (Ficus, EAFS, ACLFS, Sloop, etc.)

- Or one seriously "killer app"

- Target a community most likely to adopt new development technology like ours:

  Linux, FreeBSD, NetBSD, etc.

- Have several influential "satisfied customers"

  (real users who aren't "involved" in our research project

  (not TIS, ISI))

- Provide good documentation for file system developers

  How to create a new file system service as a layer

  How to modify an existing VOP.

  How to add a new VOP.

- Provide a series of good, easy to understand samples:

  The NULL Layer.

  A Cleaned Up NULL Layer.

  The "Hello World" Layer.

- Free all code from any lingering source licenses

Productivity In The I405 Environment:

My productivity was initially low while I climbed a learning curve. This learning curve is significantly long because it is necessary to simultaneously learn all of the following:

```
1. kernel configuring
2. kernel building
3. kernel programming nuances
4. kernel debugging
```

```
5. I405 interface mechanism
6. I405 interface design
7. I405 interface development
8. the NULL layer and how to modify it
9. vnode/vfs interfaces
```

Once the climb up the learning curve is complete, the i405 environment is very productive. For example, I implemented the ACLFS in under nine days.

To make the learning curve shorter, need better documentation, training, and/or support in all or most of the above listed areas.

To make the environment more productive, we should improve the environment in the ways referred to above in the section "To Make Our Work Successful". We need more examples, to speedup the compile-debug cycle, train for kernel programming environment differences (could just be a readme file enumerating the pitfalls).

Problems With Kernel Environment:

- standard library routines supported BUT in a non-standard way
  (e.g. `bcopy( )` is supported but handles overlapping case incorrectly)
- the compile, test, debug cycle is slow:
  test machines hang and must be power cycled to un-freeze
  location of test machines inconvenient
  kernel builds take a long time
  kgdb is slow
- limited stack space for automatic variables

To Move The EAFS Forward:

1. Simplify creation of raw EAFS volumes by adding the initialization steps as an option of "mount_ea". (See the section “Setup“ in “EA User Guide”).
2. Hide EAFS mapping database files from users by:
   putting the EA mapping database files in the raw directory’s root

mounting only a subdirectory of the raw directory

```
e.g.  /raw/eafs.dir/           <-- DB files here
      /raw/eafs.dir/mountpt/  <-- raw files here
(only hierarchy below "mountpt/" would be visible)
```

3. Create and use a "copy" vnode operation

    so that extended attributes are preserved when a file is copied

    (so apps like "cp", "emacs", etc. could make copies of files including their attributes without knowing any EAFS specifics).

4. Tune (see the “Performance Of The Extended Attribute Service” section below).

Implementation Of The Attribute Areas:

There are two types of attribute storage areas: the fast attribute area and the large attribute area. The attribute areas are very similar in content, but not in size or speed of access. Attributes are not defined as either "fast" or "large" but are assigned to an area dynamically by the EAFS. To facilitate the EAFS' policy algorithm, each attribute ID is given a priority value. An attribute's priority and its value's length determine whether it is located in the fast area or the large area. The policy's goal is to maximize the number of high priority attributes in the fast area.

Contents-wise, both attribute areas are lists of

```
struct ea_val_map_elem {
        caid_t  caid;  /* 32 bit local attribute identifier */
        u_long length; /* length of entire attribute entry  */
        char val[4];   /* variable length value field       */
};
```

The caid stands for “compact attribute ID” and is a local equivalent of the attribute’s UUID, only smaller. The length field is used to find the next attribute entry. The val field is variable length and contains the attribute value.

Size-wise, the fast area is fixed at 128 bytes while the large area is 8192 bytes or longer (see the "Limitations" section below).

Speed-wise, the fast area is already in memory since to look up the file at all, we had to load its inode. The fast area is guaranteed to be in the same disk block as the file's inode. The large area is stored separately in a regular file that is accessed by inode number (converted to a NEWFID). Accessing a large attribute area usually incurs a disk seek to read in the needed disk block, and can incur an additional seek to first read in the inode of the large area file (if the large area's inode is not in the same disk block as the subject's inode and is not already in memory).

Fast Attribute Area

The fast attribute area is stored immediately adjacent to the file's inode in a reserved inode entry. This means that the fast attribute area is the size of a disk inode (128 bytes). This scheme halves the number of regular file inodes per disk block in order to make the room for storing attribute data. It is accomplished through the kernel options UFSATTR and IACCESS.

Large Attribute Area

The large attribute area is in all respects a regular file with the one exception that it does not have a name in the directory system. The large area is created, read, written, and deleted by the EAFS through the use of its inode number via the IACCESS kernel option and the `NVOP_NEWFID_TO_VP()` vnode operation. The large area's inode number is stored persistently as a NewFID in unused fields (at offset 112) of the file's **regular** inode (not the fast area of the doubled inode). Once we have the vnode of the large area, we map the disk pages containing the attribute values directly into memory with `segmap_getmap()`. Using segmap eliminates double buffering and greatly increases performance compared to, for example, `vn_rdwr()`.

Limitations Of Implementation:

The current implementation of the EAFS limits the number of attribute IDs that can exist in any volume and the number of attribute values that can be stored per file.

The limit on attribute IDs is due to the fact that only one disk block is currently supported for each of the mapping databases. This fact limits the number of attribute IDs that can exist in one EAFS volume to about 400.

```
400 ~= 8192/20 = DISK_BLK_SIZE / ((sizeof(NCA_UUID) + sizeof(CAID))
```

The limit on attribute values per file is set by the size of the fast attribute area and the size of the large attribute area. The maximum numbers for each area are:

```
max num fast attributes   10
max num large attributes 682
```

These limits are based on the minimum attribute value record size of twelve bytes. In practice, the maximum number of attributes for each area may be significantly less depending on the size of each attribute value.

HOWEVER, these limitations are not set by design. Rather, they are temporary restrictions based on the current state of the implementation. In fact, most of the code needed to remove these restrictions is already written.

- To allow an increased number of attributes per file,
  first try redefining `LARGE_ATTR_SIZE` to a multiple of `DISK_BLK_SIZE`.

- To allow an arbitrary number of attributes per file,
  rewrite the section of ea_setexattrs just after "`if (spillover) {` ".
  rewrite the section of ea_setexattrs at
    "`attr_filesize = DISK_BLK_SIZE;`"

- To allow an arbitrary number of attribute IDs per volume,
  add "`#define ENLARGEABLE_EADBS`" to ea_map_mgr.c
  and add a call to "`enlarge_map( )`"
    either at "`enter_newentry:`"
    or when "`map_mgr( )`" returns `mmrvs->expand_file == 1;`

Performance Of The Extended Attribute Service

Measurements And Expectations:

The Modified Andrew Benchmark was performed Feb. '97. The elapsed time for recursive attribute accesses was performed Oct. '93.

The figures below are rough first (or second) pass numbers. Time did not permit perfecting the testing method or to exhaust all tuning possibilities. Nevertheless, these numbers give some indication of the state of affairs in the EAFS and ACLFS.

Please note the anomalies present in the MAB such as the UFSATTR filesystem being 2% faster than the UFS and the 2NULLFS (two null layers stacked on top of each other) being the fastest of all. These anomalies cast some doubt on the accuracy and/or precision of these tests.

The short of it is that the extended attribute service adds a penalty of approximately six to ten percent when fast attributes are used. Accessing attributes in the large attribute area approximately doubles the elapsed time. These estimates are based on some preliminary performance testing shown here:

Modified Andrew Benchmark: **(tests performed Oct '93)**

Samp	mkdir	copy	rstat	grep	cc	total	filesystem/machine	diff%
n=47	4.9	13.8	14.3	17.5	103.3	154	ufs/warwick/	1.85
n=47	3.6	13.0	12.8	17.7	103.8	151.2	ufsattr/warwick/	base
n=24	3.9	14.0	13.0	18.1	103.7	152.9	nullfs/warwick/	1.01
n=17	3.5	12.6	13.5	17.2	103.4	150.4	2nullfs/warwick/	-.53
n=16	4.5	15.7	13.4	18.9	108.4	161.0	EAFS/warwick/	6.48
n=27	5.2	20.2	14.2	18.2	107.5	165.5	aclfs/warwick/	9.46

Recursive Attribute Access On Copy Of /usr/include: **(tests performed Feb '97)**

```
Accessing only fast  attributes adds about 10% elapsed time.
Accessing only large attributes adds about 120% elapsed time.
```

With additional effort it may be possible to improve the performance of the EAFS and ACLFS by using the MAB as an indication of where to focus tuning effort. For instance, EAFS tuning could focus on the approximately 15% (2 seconds out of 15.7) penalty that the copy phase incurs.

Performance Alternatives:

It would be much easier to maximize performance if we knew ahead of time what mix of attributes would be stored in an EAFS volume and the size and access patterns for each attribute. Given this information we would set the size of the fast attribute area appropriately to manage the tradeoff between inode locality during "standard" disk operations and attribute locality during EA operations.

Not knowing this information and to make a general-purpose attribute system we chose a trade-off of 128 bytes fast area per file. This means we "steal" half the inodes in a disk block for extended attribute storage.

To increase the locality of large attribute storage, we should try to ensure that the large area's inode is in memory at the same time as the file's inode. This could be accomplished in a variety of ways:

"best effort"

when allocating a large attribute area, try to get an inode in same disk block, or in same track, or same cylinder group.

"tripling" the inode:

1. file's regular inode
2. file's fast attribute area
3. file's large attribute area file's inode

"common" large area:

each diskblock's worth of inode's would contain one reserved inode which represents the large attribute area for all the other inode's on the diskblock. The fast area associated with this reserved inode could contain some sort of map between the other “regular” inodes in the inode block and attribute entries or large area diskblocks.

Amount Of Code Written:

The number of lines of C source code I wrote for the EAFS and ACLFS are listed here and broken into two categories: In-Kernel and Utilities:

```
In-Kernel (does not include code derived from NULL layer)
  EAFS    3639
  ACLFS    798

Utilities
  EAFS    4523
  ACLFS   1235

Total    10195
```

The remarkable thing to notice about the amount of code I wrote is that I implemented the ACLFS in less than 800 lines of code. This feat was accomplished in under nine days from start to finish (including design, implementation, testing, and debugging) and was made possible by using the attribute service provided by the EAFS. This shows that the EAFS greatly simplifies file system development.