The Venus kernel interface

Peter J. Braam v1.2, Mar 18, 1998

This document describes the communication between Venus and kernel level file system code needed for the operation of the Coda filesystem. This version document is meant to describe the current interface (version 1.0) as well as improvements we envisage.

Contents

1 Introduction

A key component in the Coda Distributed File System is the cache manager, *Venus*.

When processes on a Coda enabled system access files in the Coda filesystem, requests are directed at the filesystem layer in the operating system. The operating system will communicate with Venus to service the request for the process. Venus manages a persistent client cache and makes remote procedure calls to Coda file servers and related servers (such as authentication servers) to service these requests it receives from the operating system. When Venus has serviced a request it replies to the operating system with appropiate return codes, and other data related to the request. Optionally the kernel support for Coda may maintain a minicache of recently processed requests to limit the number of interactions with Venus. Venus possesses the facility to inform the kernel when elements from its minicache are no longer valid.

This document describes precisely this communication between the kernel and Venus. The definitions of so called upcalls and downcalls will be given with the format of the data they handle. We shall also describe the semantic invariants resulting from the calls.

Historically Coda was implemented in a BSD file system in Mach 2.6. The interface between the kernel and Venus is very similar to the BSD VFS interface. Similar functionality is provided, and the format of the parameters and returned data is very similar to the BSD VFS. This leads to an almost natural environment for implementing a kernel level filesystem driver for Coda in a BSD system. However, other operating systems such as Linux and Windows 95 and NT have virtual filesystem with different interfaces.

To implement Coda on these systems some reverse engineering of the Venus/Kernel protocol is necessary. Also it came to light that other systems could profit significantly from certain small optimizations and modifications to the protocol. To facilitate this work as well as to make future ports easier, communication between Venus and the kernel should be documented in great detail. This is the aim of this document.

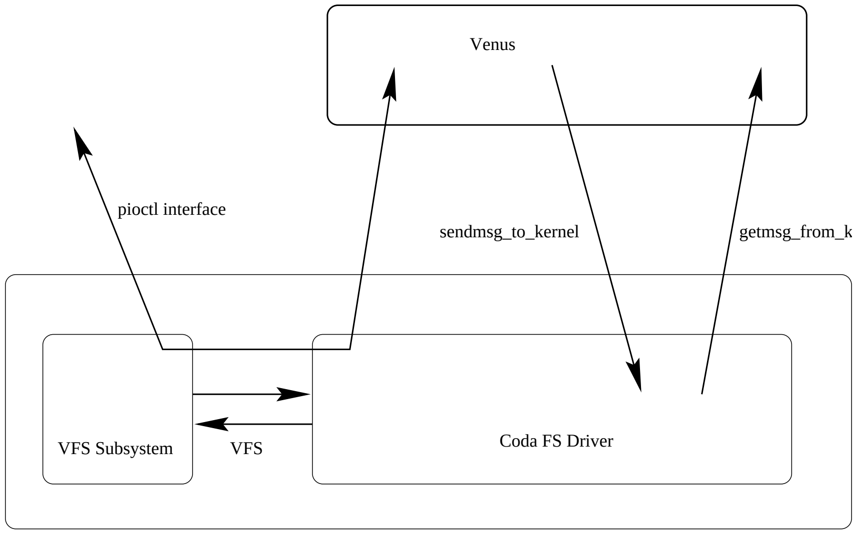

Figure 1: Interfaces of the Coda FS Driver

2 Servicing Coda filesystem calls

The service of a request for a Coda file system service originates in a process **P** which accessing a Coda file. It makes a system call which traps to the OS kernel. Examples of such calls trapping to the kernel are *read, write, open, close, create, mkdir, rmdir, chmod* in a Unix context. Similar calls exist in the Win32 environment, and are named *CreateFile,* .

Generally the operating system handles the request in a virtual filesystem (VFS) layer, which is named I/O Manager in NT and IFS manager in Windows 95. The VFS is responsible for partial processing of the request and for locating the specific filesystem(s) which will service parts of the request. Usually the information in the path assists in locating the correct FS drivers. Sometimes after extensive pre-processing, the VFS starts invoking exported routines in the FS driver. This is the point where the FS specific processing of the request starts, and here the Coda specific kernel code comes into play.

The FS layer for Coda must expose and implement several interfaces. First and foremost the VFS must be able to make all necessary calls to the Coda FS layer, so the Coda FS driver must expose the VFS interface as applicable in the operating system. These differ very significantly among operating systems, but share features such as facilities to read/write and create and remove objects. The Coda FS layer services such VFS requests by invoking on or more well defined services offered by the cache manager Venus. When the replies from Venus have come back to the FS driver, servicing of the VFS call continues and finishes with a reply to the kernels VFS. Finally the VFS layer returns to the process.

As a result of this design a basic interface exposed by the FS driver must allow Venus to manage message traffic. In particular Venus must be able to retrieve and place messages and to be notified of the arrival of a new message. The notification must be through a mechanism which does not block Venus since Venus must attend to other tasks even when no messages are waiting or being processed.

Furthermore the FS layer provides for a special path of communication between a user process and Venus, called the pioctl interface. The pioctl interface is used for Coda specific services, such as requesting detailed information about the persistent cache managed by Venus. Here the involvement of the kernel is minimal. It identifies the calling process and passes the information on to Venus. When Venus replies the response is passed back to the caller in unmodified form.

Finally Venus allows the kernel FS driver to cache the results from certain services. This is done to avoid excessive context switches and results in an efficient system. However, Venus may acquire information, for example from the network which implies that cached information must be flushed or replaced. Venus then makes a downcall to the Coda FS layer to request flushes or updates in the cache. The kernel FS driver handles such requests synchronously.

Among these interfaces the VFS interface and the facility to place, receive and be notified of messages are platform specific. We will not go into the calls exported to the VFS layer but we will state the requirements of the message exchange mechanism.

3 The message layer

At the lowest level the communication between Venus and the FS driver proceeds through messages. The synchronization between processes requesting Coda file service and Venus relies on blocking and waking up processes. The Coda FS driver processes VFS- and pioctl-requests on behalf of a process P, creates messages for Venus, awaits replies and finally returns to the caller. The implementation of the exchange of messages is platform specific, but the semantics have (so far) appeared to be generally applicable. Data buffers are created by the FS Driver in kernel memory on behalf of P and copied to user memory in Venus.

The FS Driver while servicing P makes `upcall`'s to Venus. Such an upcall is dispatched to Venus by creating a message structure. The structure contains the identification of P, the message sequence number, the size of the request and a pointer to the data in kernel memory for the request. Since the data buffer is re-used to hold the reply from Venus, there is a field for the size of the reply. A flags field is used in the message to precisely record the status of the message. Additional platform dependent structures involve pointers to determine the position of the message on queues and pointers to synchronization objects. In the `upcall` routine the message structure is filled in, flags are set to 0, and it is placed on the *pending* queue. The routine calling `upcall` is responsible for allocating the data buffer; it's structure will be described in the next section.

A facility must exist to notify Venus that the message has been created, and implemented using available synchronization objects in the OS. This notification is done in the `upcall` context of the process P. When the message is on the pending queue, process P cannot proceed in `upcall`. The (kernel mode) processing of P in the filesystem request routine must be suspended until Venus has replied. Therefore the calling thread in P is blocked in `upcall`. A pointer in the message structure will locate the synchronization object on which P is sleeping.

Venus detects the notification that a message has arrived, and the FS driver allow Venus to retrieve the message with a `getmsg_from_kernel` call. This action finishes in the kernel by putting the message on the queue of processing messages and setting flags to READ. Venus is passed the contents of the data buffer. The `getmsg_from_kernel` call now returns and Venus processes the request.

At some later point the FS driver receives a message from Venus, namely when Venus calls `sendmsg_to_kernel`. At this moment the Coda FS driver looks at the contents of the message and decides if:

- the message is a reply for a suspended thread P. If so it removes the message from the processing queue and marks the message as WRITTEN. Finally, the FS driver unblocks P (still in the kernel mode context of Venus) and the `sendmsg_to_kernel` call returns to Venus. The process P will be scheduled at some point and continues processing its `upcall` with the data buffer replaced with the reply from Venus.
- The message is a *downcall*. A downcall is a request from Venus to the FS Driver. The FS driver processes the request immediately (usually a cache eviction or replacement) and when finishes `sendmsg_to_kernel` returns.

Now P awakes and continues processing `upcall`. There are some subtleties to take account off. First P will determine if it was woken up in `upcall` by a signal from some other source (for example an attempt to terminate P) or as is normally the case by Venus in its `sendmsg_to_kernel` call. In the normal case, the upcall routine will deallocate message structure and return. The FS routine can proceed with its processing.

In case P is woken up by a signal and not by Venus, it will first look at the flags field. If the message is not yet READ, the process P can handle it's signal without notifying Venus. If Venus has READ, and the request should not be processed, P can send Venus a signal message to indicate that it should disregard the

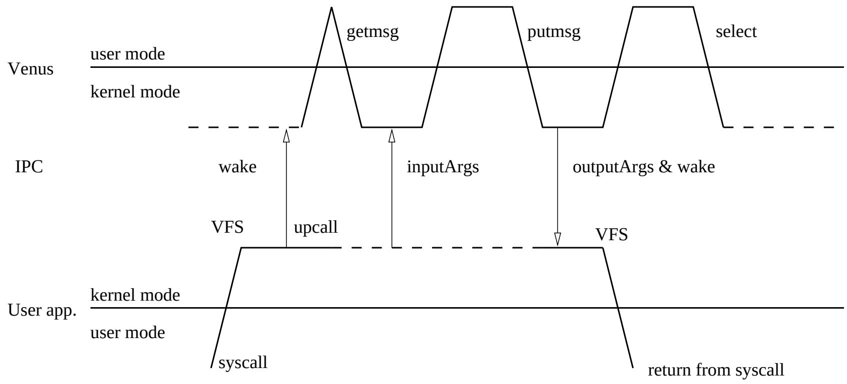

Figure 2: Sleeping and IPC arrangements

previous message. Such signals are put in the queue at the head, and read first by Venus. If the message is already marked as WRITTEN it is too late to stop the processing. The VFS routine will now continue. [1]

3.1 Implementation details

The Unix implementation of this mechanism has been through the implemenation of a character device associated with Coda. Venus retrieves messages by doing a `read` on the device, replies are sent with a `write` and notification is through the `select` system call on the file descriptor for the device. The process P is kept waiting on an interruptible wait queue object.

In Windows NT and the DPMI Windows 95 implementation a DeviceIoControl call is used. The DeviceIoControl call is designed to copy buffers from user memory to kernel memory with OPCODES. The `sendmsg_to_kernel` is issued as a synchronous call, while the `getmsg_from_kernel` call is asynchrounous. Windows EventObjects are used for notification of message arrival. The process P is kept waiting on a KernelEvent object in NT and a semaphore in Windows 95.

[1] If a VFS request involves more than one upcall, this can lead to complicated state, an extra field "handle_signals" could be added in the message structure to indicate points of no return have been passed.

4 The interface at the call level

This section describes the upcalls a Coda FS driver can make to Venus. Each of these upcalls make use of three structures: `inputArgs` for communication from kernel to Venus, and `outputArgs` back from Venus to the kernel and finally downcalls which are calls from Venus to the kernel initiated by Venus.

```
union inputArgs {
    struct cfs_in_hdr ih;       /* NB: every struct below begins with an ih */
    struct cfs_open_in cfs_open;
    struct cfs_close_in cfs_close;
    struct cfs_ioctl_in cfs_ioctl;
    struct cfs_getattr_in cfs_getattr;
    struct cfs_setattr_in cfs_setattr;
    struct cfs_access_in cfs_access;
    struct cfs_lookup_in cfs_lookup;
    struct cfs_create_in cfs_create;
    struct cfs_remove_in cfs_remove;
    struct cfs_link_in cfs_link;
    struct cfs_rename_in cfs_rename;
    struct cfs_mkdir_in cfs_mkdir;
    struct cfs_rmdir_in cfs_rmdir;
    struct cfs_readdir_in cfs_readdir;
    struct cfs_symlink_in cfs_symlink;
    struct cfs_readlink_in cfs_readlink;
    struct cfs_fsync_in cfs_fsync;
    struct cfs_inactive_in cfs_inactive;
    struct cfs_vget_in cfs_vget;
    struct cfs_rdwr_in cfs_rdwr;
    struct cfs_open_by_path_in cfs_open_by_path;
};

union outputArgs {
    struct cfs_out_hdr oh; /* NB: every struct below begins with an oh */
    struct cfs_root_out cfs_root;
    struct cfs_open_out cfs_open;
    struct cfs_ioctl_out cfs_ioctl;
    struct cfs_getattr_out cfs_getattr;
    struct cfs_lookup_out cfs_lookup;
    struct cfs_create_out cfs_create;
    struct cfs_mkdir_out cfs_mkdir;
    struct cfs_readdir_out cfs_readdir;
    struct cfs_readlink_out cfs_readlink;
    struct cfs_vget_out cfs_vget;
    struct cfs_purgeuser_out cfs_purgeuser;
    struct cfs_zapfile_out cfs_zapfile;
    struct cfs_zapdir_out cfs_zapdir;
    struct cfs_zapvnode_out cfs_zapvnode;
    struct cfs_purgefid_out cfs_purgefid;
    struct cfs_rdwr_out cfs_rdwr;
    struct cfs_replace_out cfs_replace;
    struct cfs_open_by_path_out cfs_open_by_path;
```

```
};

union cfs_downcalls {
    /* CFS_FLUSH  is also a down call */
    struct cfs_purgeuser_out purgeuser;
    struct cfs_zapfile_out zapfile;
    struct cfs_zapdir_out zapdir;
    struct cfs_zapvnode_out zapvnode;
    struct cfs_purgefid_out purgefid;
    struct cfs_replace_out replace;
};
```

The headers are common to all calls and contain process, authentication and opcode information:

```
struct cfs_in_hdr {
    unsigned long opcode;
    unsigned long unique;    /* Keep multiple outstanding msgs distinct */
    u_short pid;                 /* Common to all */
    u_short pgid;                /* Common to all */
    u_short sid;                 /* to become the PAG */
    struct coda_cred cred;       /* to become a PAG */
};

/* Really important that opcode and unique are 1st two fields! */
struct cfs_out_hdr {
    unsigned long opcode;
    unsigned long unique;
    unsigned long result;
};
```

Before going on let us elucidate the role of the various fields. The inputArgs start with the `opcode` which defines the type of service requested from Venus. There are approximately 30 upcalls at present which we will discuss. The `unique` field labels the inputArg with unique number which will identify the message uniquely. A process and process group id are passed. Finally the credentials of the caller are included.

Before delving into the specific calls we need to discuss a variety of data structures shared by the kernel and Venus.

4.1 Data structures shared by the kernel and Venus

The `CodaCred` structure defines a variety of user and group id's as they are set for the calling process. The `vuid_t` and `guid_t` are 32 bit unsigned integers. It also defines group member ship in an array. On Unix the CodaCred has proven sufficient to implement good security semantics for Coda but the structure may have to undergo modification for the Windows environment when these mature.

```
struct CodaCred {
    vuid_t cr_uid, cr_euid, cr_suid, cr_fsuid; /* Real, efftve, set, fs uid*/
    vgid_t cr_gid, cr_egid, cr_sgid, cr_fsgid; /* same for groups */
    vgid_t cr_groups[NGROUPS];        /* Group membership for caller */
};
```

NOTE It is questionable if we need CodaCreds in Venus. Finally Venus doesn't know about groups, although it does create files with the default uid/gid. Perhaps the list of group membership is superfluous.

The next item is the fundamental identifier used to identify Coda files, the `ViceFid`. A fid of a file uniquely defines a file or directory in the Coda filesystem within a *cell*. [2]

```
typedef struct ViceFid {
    VolumeId Volume;
    VnodeId Vnode;
    Unique_t Unique;
} ViceFid;
```

Each of the constituent fields: `VolumeId, VnodeId` and `Unique_t` are unsigned 32 bit integers. We envisage that a further field will need to be prefixed to identify the Coda cell; this will probably take the form of a Ipv6 size IP address naming the Coda cell through DNS.

The next important structure shared between Venus and the kernel are the attributes of the file. The following structure is used to exchange information. It has room for future extensions such as support for device files (currently not present in Coda).

```
struct coda_vattr {
        enum coda_vtype va_type;        /* vnode type (for create) */
        u_short         va_mode;        /* files access mode and type */
        short           va_nlink;       /* number of references to file */
        vuid_t          va_uid;         /* owner user id */
        vgid_t          va_gid;         /* owner group id */
        long            va_fsid;        /* file system id (dev for now) */
        long            va_fileid;      /* file id */
        u_quad_t        va_size;        /* file size in bytes */
        long            va_blocksize;   /* blocksize preferred for i/o */
        struct timespec va_atime;       /* time of last access */
        struct timespec va_mtime;       /* time of last modification */
        struct timespec va_ctime;       /* time file changed */
        u_long          va_gen;         /* generation number of file */
        u_long          va_flags;       /* flags defined for file */
        dev_t           va_rdev;        /* device special file represents */
        u_quad_t        va_bytes;       /* bytes of disk space held by file */
        u_quad_t        va_filerev;     /* file modification number */
        u_int           va_vaflags;     /* operations flags, see below */
        long            va_spare;       /* remain quad aligned */
};
```

4.2 The pioctl interface

Coda specific requests can be made by application through the pioctl interface. The pioctl is implemented as an ordinary ioctl on a ficticious file `/coda/.CONTROL`. The piocl call opens this file, gets a file handle and makes the ioctl call. Finally it closes the file.

The kernel involvement in this is limited to providing the facility to open and close and pass the ioctl message *and* to verify that a path in the pioctl data buffers is a file in a Coda filesystem.

[2] A *cell* is a group of Coda servers acting under the aegis of a single system control machine or SCM. See the Coda Administration manual for a detailed description of the role of the SCM.

The kernel is handed a data packet of the form:

```
    struct {
        const char *path;
        struct ViceIoctl vidata;
        int follow;
    } data;
```

where

```
struct ViceIoctl {
        caddr_t in, out;        /* Data to be transferred in, or out */
        short in_size;          /* Size of input buffer <= 2K */
        short out_size;         /* Maximum size of output buffer, <= 2K */
};
```

The `path` must be a Coda file, otherwise the `ioctl` upcall will not be made.

NOTE The data structures and code are a mess. We need to clean this up.

We now proceed to document the individual calls:

4.3 root

Arguments

in

empty

out

```
struct cfs_root_out {
    ViceFid VFid;
} cfs_root;
```

Description This call is made to Venus during the initialization of the Coda filesystem. If the `result` is zero, the `cfs_root` structure contains the ViceFid of the root of the Coda filesystem. If a non-zero result is generated, its value is a platform dependent error code indicating the difficulty Venus encountered in locating the root of the Coda filesystem.

4.4 lookup

Summary Find the ViceFid and type of an object in a directory if it exists.

Arguments

in

```
struct  cfs_lookup_in {
    ViceFid     VFid;
    char        *name;          /* Place holder for data. */
} cfs_lookup;
```

out

```
struct cfs_lookup_out {
    ViceFid VFid;
    int vtype;
} cfs_lookup;
```

Description This call is made to determine the ViceFid and filetype of a directory entry. The directory entry requested carries name `name` and Venus will search the directory identified by `cfs_lookup_in.VFid`. The `result` may indicate that the name does not exist, or that difficulty was encountered in finding it (e.g. due to disconnection). If the `result` is zero, the field `cfs_lookup_out.VFid` contains the targets ViceFid and `cfs_lookup_out.vtype` the `coda_vtype` giving the type of object the name designates.

The name of the object is an 8 bit character string of maximum length CFS_MAXNAMLEN, currently set to 256 (including a 0 terminator.)

It is extremely important to realize that Venus bitwise or's the field `cfs_lookup.vtype` with `CFS_NOCACHE` to indicate that the object should not be put in the kernel name cache. Files in conflict or uncovered mountpoints should never be cached by the kernel and each lookup should make it to Venus.

NOTE The type of the vtype is currently wrong. It should be `coda_vtype`. Linux does not take note of `CFS_NOCACHE`. It should.

4.5 getattr

Summary Get the attributes of a file.

Arguments

in

```
struct cfs_getattr_in {
    ViceFid VFid;
    struct coda_vattr attr; /* XXXXX */
} cfs_getattr;
```

out

```
struct cfs_getattr_out {
    struct coda_vattr attr;
} cfs_getattr;
```

Description This call returns the attributes of the file identified by fid.

Errors Errors can occur if the object with fid does not exist, are unaccessible or if the caller does not have permission to fetch attributes.

Note Many kernel FS drivers (Linux, NT and Windows 95 need to acquire the attributes as well as the Fid for the instantiation of an internal "inode" or "FileHandle". A significant improvement in performance on such systems could be made by combining the *lookup* and *getattr* calls both at the Venus/kernel interaction level and at the RPC level.

The `vattr` structure included in the input arguments is superfluous and should be removed.

4.6 setattr

Summary Set the attributes of a file.

Arguments

in

```
struct cfs_setattr_in {
    ViceFid VFid;
    struct coda_vattr attr;
} cfs_setattr;
```

out

empty

Description The structure `attr` is filled with attributes to be changed in BSD style. Attributes not to be changed are set to -1, apart from `vtype` which is set to `VNON`. Other are set to the value to be assigned. The only attributes which the FS driver may request to change are the mode, ownner, groupid, atime, mtime and ctime. The return value indicates success or failure.

Errors A variety of errors can occur. The object may not exist, may be inaccessible, or permission may not be granted by Venus.

4.7 access

Summary

Arguments

in

```
struct cfs_access_in {
    ViceFid     VFid;
    int flags;
} cfs_access;
```

out

empty

Description Verify if access to the object identified by `VFid` for operations described by `flags` is permitted. The result indicates if access will be granted. It is important to remember that Coda uses ACL's to enforce protection and that ultimately the servers, not the clients enforce the security of the system. The result of this call will depend on wether a *token* is held by the user.

Errors The object may not exist, or the ACL describing the protection may not be accessible.

4.8 create

Summary Invoked to create a file

Arguments

in

```
struct cfs_create_in {
    ViceFid VFid;
    struct coda_vattr attr;
    int excl;
    int mode;
    char        *name;          /* Place holder for data. */
} cfs_create;
```

out

```
struct cfs_create_out {
    ViceFid VFid;
    struct coda_vattr attr;
} cfs_create;
```

Description This upcall is invoked to request creation of a file. The file will be created in the directory identified by `VFid`, its name will be `name`, and the mode will be `mode`. If `excl` is set an error will be returned if the file already exists. If the size field in attr is set to zero the file will be truncated. The uid and gid of the file are set by converting the CodaCred to a uid using a macro `CRTOUID` (this macro is platform dependent). Upon success the VFid and attributes of the file are returned. The Coda FS Driver will normally instantiate a vnode, inode or filehandle at kernel level for the new object.

Errors A variety of errors can occur. Permissions may be insufficient. If the object exists and is not a file the error `EISDIR` is returned under Unix.

NOTE The packing of parameters is very inefficient and appears to indicate confusion between the system call creat and the VFS operation create. The VFS operation create is only called to create new objects. This create call differs from the Unix one in that it is not invoked to return a file descriptor. The trunctate and exclusive options, together with the mode, could simply be part of the mode as it is under Unix. There should be no flags argument; this is used in `open` (2) to return a filedescriptor for READ or WRITE mode.

The attributes of the directory should be returned too, since the size and mtime changed.

4.9 mkdir

Summary Create a new directory.

Arguments

in

```
struct cfs_mkdir_in {
    ViceFid     VFid;
    struct coda_vattr attr;
    char        *name;          /* Place holder for data. */
} cfs_mkdir;
```

out

```
struct cfs_mkdir_out {
    ViceFid VFid;
    struct coda_vattr attr;
} cfs_mkdir;
```

Description This call is similar to `create` but creates a directory. Only the `mode` field in the input parameters is used for creation. Upon successful creation, the `attr` returned contains the attributes of the new directory.

Errors As for create.

NOTE The input parameter should be changed to `mode` instead of attributes.

The attributes of the parent should be returned since the size and mtime changes.

4.10 link

Summary Create a link to an existing file.

Arguments

in

```
struct cfs_link_in {
    ViceFid sourceFid;          /* cnode to link *to* */
    ViceFid destFid;            /* Directory in which to place link */
    char        *tname;         /* Place holder for data. */
} cfs_link;
```

out

empty

Description This call creates a link to the `sourceFid` in the directory identified by `destFid` with name `tname`. The source must reside in the targets parent, i.e. the source must be have parent `destFid`, i.e. Coda does not support cross directory hard links. Only the return value is relevant. It indicates success or the type of failure.

Errors The usual errors can occur.

4.11 symlink

Summary create a symbolic link

Arguments

in

```
struct cfs_symlink_in {
    ViceFid     VFid;          /* Directory to put symlink in */
    char        *srcname;
    struct coda_vattr attr;
    char        *tname;
} cfs_symlink;
```

out

none

Description Create a symbolic link. The link is to be placed in the directory identified by `VFid` and named `tname`. It should point to the pathname `srcname`. The attributes of the newly created object are to be set to `attr`.

Errors

NOTE The attributes of the target directory should be returned since its size changed.

4.12 remove

Summary Remove a file

Arguments

in

```
struct cfs_remove_in {
    ViceFid     VFid;
    char        *name;          /* Place holder for data. */
} cfs_remove;
```

out

none

Description Remove file named `cfs_remove_in.name` in directory identified by `VFid`.

Errors

NOTE The attributes of the directory should be returned since its mtime and size may change.

4.13 rmdir

Summary Remove a directory

Arguments

in

```
struct cfs_rmdir_in {
    ViceFid     VFid;
    char        *name;          /* Place holder for data. */
} cfs_rmdir;
```

out

none

Description Remove the directory with name `name` from the directory identified by `VFid`.

Errors

NOTE The attributes of the parent directory should be returned since its mtime and size may change.

4.14 readlink

Summary Read the value of a symbolic link.

Arguments

in

```
struct cfs_readlink_in {
    ViceFid VFid;
} cfs_readlink;
```

out

```
struct cfs_readlink_out {
    int count;
    caddr_t     data;           /* Place holder for data. */
} cfs_readlink;
```

Description This routine reads the contents of symbolic link identified by `VFid` into the buffer `data`. The buffer `data` must be able to hold any name up to `CFS_MAXNAMLEN` (PATH or NAM??).

Errors No unusual errors.

4.15 open

Summary Open a file.

Arguments

in

```
struct cfs_open_in {
    ViceFid     VFid;
    int flags;
} cfs_open;
```

out

```
struct cfs_open_out {
    dev_t       dev;
    ino_t       inode;
} cfs_open;
```

Description This request asks Venus to place the file identified by `VFid` in its cache and to note that the calling process wishes to open it with `flags` as in `open(2)`. The return value to the kernel differs for Unix and Windows systems. For Unix systems the Coda FS Driver is informed of the device and inode number of the container file in the fields `dev` and `inode`. For Windows the path of the container file is returned to the kernel.

Errors

NOTE Currently the `cfs_open_out` structure is not properly adapted to deal with the windows case. It might be best to implement two upcalls, one to open aiming at a container file name, the other at a container file inode.

4.16 close

Summary Close a file, update it on the servers.

Arguments

in

```
struct cfs_close_in {
    ViceFid     VFid;
    int flags;
} cfs_close;
```

out

none

Description Close the file identified by `VFid`.

Errors

NOTE The `flags` argument is bogus and not used. However, Venus' code has room to deal with an `execp` input field, probably this field should be used to inform Venus that the file was closed but is still memory mapped for execution. There are comments about fetching versus not fetching the data in Venus `vproc_vfscalls`. This seems silly. If a file is being closed, the data in the container file is to be the new data. Here again the `execp` flag might be in play to create confusion: presently Venus might think a file can be flushed from the cache when it is still memory mapped. This needs to be understood.

4.17 ioctl

Summary Do an ioctl on a file. This includes the piocl interface.

Arguments

in

```
struct cfs_ioctl_in {
    ViceFid VFid;
    int cmd;
    int len;
    int rwflag;
    char *data;                 /* Place holder for data. */
} cfs_ioctl;
```

out

```
struct cfs_ioctl_out {
    int len;
    caddr_t     data;           /* Place holder for data. */
} cfs_ioctl;
```

Description Do an ioctl operation on a file. The `command, len` and `data` arguments are filled as usual. `flags` is not used by Venus.

Errors

NOTE Another bogus parameter. `flags` is not used. What is the business about PREFETCHING in the Venus' code?

4.18 rename

Summary Rename a fid.

Arguments

in

```
struct cfs_rename_in {
    ViceFid     sourceFid;
    char        *srcname;
    ViceFid destFid;
    char        *destname;
} cfs_rename;
```

out

none

Description Rename the object with name `srcname` in directory `sourceFid` to `destname` in `destFid`. It is important that the names `srcname` and `destname` are 0 terminated strings. Strings in Unix kernels are not always null terminated.

Errors

4.19 readdir

Summary Read directory entries.

Arguments

in

```
struct cfs_readdir_in {
    ViceFid     VFid;
    int count;
    int offset;
} cfs_readdir;
```

out

```
struct cfs_readdir_out {
    int size;
    caddr_t     data;           /* Place holder for data. */
} cfs_readdir;
```

Description Read directory entries from `VFid` starting at `offset` and read at most `count` bytes. Returns the data into `data` and indicates the size returned `size`.

Errors

NOTE This call is not used. Readdir operations exploit container files. We will re-evaluate this during the directory revamp which is about to take place.

4.20 vget

Summary instructs Venus to do an `FSDB->Get`.

Arguments

in

```
struct cfs_vget_in {
    ViceFid VFid;
} cfs_vget;
```

out

```
struct cfs_vget_out {
    ViceFid VFid;
    int vtype;
} cfs_vget;
```

Description This upcall asks Venus to do a `get` operation on an `fsobj` labelled by `VFid`.

Errors

NOTE This operation is not used. However, it is extremely useful since it can be used to deal with read/write memory mapped files. These can be "pinned" in the Venus cache using `vget` and release with `inactive`.

4.21 fsync

Summary Tell Venus to update the RVM attributes of a file.

Arguments

in

```
struct cfs_fsync_in {
    ViceFid VFid;
} cfs_fsync;
```

out

none

Description Ask Venus to update RVM attributes of object `VFid`. This should be called as part of kernel level `fsync` type calls. The `result` indicates if the synching was successful.

Errors

NOTE Linux does not implement this call. It should.

4.22 inactive

Summary Tell Venus a vnode is no longer in use.

Arguments

in

```
struct cfs_inactive_in {
    ViceFid VFid;
} cfs_inactive;
```

out

none

Description This operation returns `EOPNOTSUPP`.

Errors

NOTE This should perhaps be removed.

4.23 rdwr

Summary Read or write from a file

Arguments

in

```
struct cfs_rdwr_in {
    ViceFid     VFid;
    int rwflag;
    int count;
    int offset;
    int ioflag;
    caddr_t     data;           /* Place holder for data. */
} cfs_rdwr;
```

out

```
struct cfs_rdwr_out {
    int rwflag;
    int count;
    caddr_t     data;   /* Place holder for data. */
} cfs_rdwr;
```

Description This upcall asks Venus to read or write from a file.

Errors

NOTE It should be removed since it is against the Coda philosophy that read/write operations never reach Venus. I have been told the operation does not work. It is not currently used.

4.24 odymount

Summary Allows mounting multiple Coda "filesystems" on one Unix mount point.

Arguments

in

```
struct ody_mount_in {
    char        *name;          /* Place holder for data. */
} ody_mount;
```

out

```
struct ody_mount_out {
    ViceFid VFid;
} ody_mount;
```

Description Asks Venus to return the rootfid of a Coda system named `name`. The fid is returned in `VFid`.

Errors

NOTE This call was used by David for dynamic sets. It should be removed since it causes a jungle of pointers in the VFS mounting area. It is not used by Coda proper. Call is not implemented by Venus.

4.25 ody_lookup

Summary Looks up something.

Arguments

in

irrelevant

out

irrelevant

Description

Errors

NOTE Gut it. Call is not implemented by Venus.

4.26 ody_expand

Summary expands something in a dynamic set.

Arguments

in

irrelevant

out

irrelevant

Description

Errors

NOTE Gut it. Call is not implemented by Venus.

4.27 prefetch

Summary Prefetch a dynamic set.

Arguments

in

Not documented.

out

Not documented.

Description Venus worker.cc has support for this call, although it is noted that it doesn't work. Not surprising, since the kernel does not have support for it. (ODY_PREFETCH is not a defined operation).

Errors

NOTE Gut it. It isn't working and isn't used by Coda.

4.28 signal

Summary Send Venus a signal about an upcall.

Arguments

in

none

out

not applicable.

Description This is an out-of-band upcall to Venus to inform Venus that the calling process received a signal after Venus read the message from the input queue. Venus is supposed to clean up the operation.

Errors No reply is given.

NOTE We need to better understand what Venus needs to clean up and if it is doing this correctly. Also we need to handle multiple upcall per system call situations correctly. It would be important to know what state changes in Venus take place after an upcall for which the kernel is responsible for notifying Venus to clean up (e.g. open definitely is such a state change, but many others are maybe not).

5 The minicache and downcalls

The Coda FS Driver can cache results of `lookup` and `access` upcalls, to limit the frequency of upcalls. Upcalls carry a price since a process context switch needs to take place. The counterpart of caching the information is that Venus will notify the FS Driver that cached entries must be flushed or renamed.

The kernel code generally has to maintain a structure which links the internal file handles (called vnodes in BSD, inodes in Linux and FileHandles in Windows) with the ViceFid's which Venus maintains. The reason is that frequent translations back and forth are needed in order to make upcalls and use the results of upcalls. Such linking objects are called **cnodes**.

The current minicache implementations have cache entries which record the following:

1. the name of the file
2. the cnode of the directory containing the object
3. a list of CodaCred's for which the lookup is permitted.
4. the cnode of the object

The `lookup` call in the Coda FS Driver may request the cnode of the desired object from the cache, by passing it's name, directory and the CodaCred's of the caller. The cache will return the cnode or indicate that it cannot be found. The Coda FS Driver must be careful to invalidate cache entries when it modifies or removes objects.

When Venus obtains information that indicates that cache entries are no longer valid, it will make a downcall to the kernel. Downcalls are intercepted by the Coda FS Driver and lead to cache invalidations of the kind described below. The Coda FS Driver does not return an error unless the downcall data could not be read into kernel memory.

5.1 INVALIDATE

No information is available on this call.

5.2 FLUSH

Arguments None

Summary Flush the name cache entirely.

Description Venus issues this call upon startup and when it dies. This is to prevent stale cache information being held. Some operating systems allow the kernel name cache to be switched off dynamically. When this is done, this downcall is made.

5.3 PURGEUSER

Arguments

```
struct cfs_purgeuser_out {/* CFS_PURGEUSER is a venus->kernel call */
    struct CodaCred cred;
} cfs_purgeuser;
```

Description Remove all entries in the cache carrying the Cred. This call is issued when tokes for a user expire or are flushed.

5.4 ZAPFILE

Arguments

```
struct cfs_zapfile_out {  /* CFS_ZAPFILE is a venus->kernel call */
    ViceFid CodaFid;
} cfs_zapfile;
```

Description Remove all entries which have the (dir vnode, name) pair. This is issued as a result of an invalidation of cached attributes of a vnode.

NOTE Call is not named correctly in NetBSD and Mach. The minicache `zapfile` routine takes different arguments. Linux does not implement the invalidation of attributes correctly.

5.5 ZAPDIR

Arguments

```
struct cfs_zapdir_out {   /* CFS_ZAPDIR is a venus->kernel call */
    ViceFid CodaFid;
} cfs_zapdir;
```

Description Remove all entries in the cache lying in a directory `CodaFid`, and all children of this directory. This call is issed when Venus receives a callback on the directory.

5.6 ZAPVNODE

Arguments

```
struct cfs_zapvnode_out { /* CFS_ZAPVNODE is a venus->kernel call */
    struct CodaCred cred;
    ViceFid VFid;
} cfs_zapvnode;
```

Description Remove all entries in the cache carrying the `cred` and `VFid` as in the arguments. This downcall is probably never issued.

5.7 PURGEFID

Summary

Arguments

```
struct cfs_purgefid_out { /* CFS_PURGEFID is a venus->kernel call */
    ViceFid CodaFid;
} cfs_purgefid;
```

Description Flush the attribute for the file. If it is a dir (odd vnode), purge its children from the namecache remove the file from the namecache.

5.8 REPLACE

Summary Replace the Fid's for a collection of names.

Arguments

```
struct cfs_replace_out { /* cfs_replace is a venus->kernel call */
    ViceFid NewFid;
    ViceFid OldFid;
} cfs_replace;
```

Description This routine replaces a ViceFid in the name cache with another. It is added to allow Venus during reintegration to replace locally allocated temp fids while disconnected with global fids even when the reference count on those fids are not zero.

6 Initialization and cleanup

This section gives brief hints as to desirable features for the Coda FS Driver at startup and upon shutdown or Venus failures. Before entering the discussion it is useful to repeat that the Coda FS Driver maintains the following data:

1. message queues
2. cnodes
3. name cache entries

The name cache entries are entirely private to the driver, so they can easily be manipulated. The message queues will generally have clear points of initialization and destruction. The cnodes are much more delicate. User processes hold reference counts in Coda filesystems and it can be difficult to clean up the cnodes.

It can expect requests through:

1. the message subsystem
2. the VFS layer
3. pioctl interface

Currently the *pioctl* passes through the VFS for Coda so we can treat these similarly.

6.1 Requirements

The following requirements should be accomodated:

1. The message queueus should have `open` and `close` routines. On Unix the opening of the character devices are such routines.
   - Before opening, no messages can be placed.
   - Opening will remove any old messages still pending.
   - Close will notify any sleeping processes that their upcall cannot be completed.
   - Close will free all memory allocated by the message queues.
2. At open the namecache shall be initialized to empty state.
3. Before the message queues are open, all VFS operations will fail. Fortunately this can be achieved by making sure than mounting the Coda filesystem cannot succeed before opening.
4. After closing of the queues, no VFS operations can succeed. Here one needs to be careful, since a few operations (lookup, read/write, readdir) can proceed without upcalls. These must be explicitly blocked.
5. Upon closing the namecache shall be flushed and disabled.
6. All memory held by cnodes can be freed without relying on upcalls.
7. Unmounting the file system can be done without relying on upcalss.
8. Mounting the Coda filesystem should fail gracefully if Venus cannot get the `rootfid` or the attributes of the `rootfid`. The latter is best implemented by Venus fetching these objects before attempting to mount.

NOTE NetBSD in particular but also Linux have not implemented the above requirements fully. For smooth operation this needs to be corrected.

7 Inode numbers

The Unix stat system call returns inode numbers for files. These should be correct for Coda files for library calls like getwd and tar to function correctly.

The mount points of volumes complicate the values which Venus assigns to the inode numbers in the statbuf structure. The rules are as follows:

1. A fid is converted to an integer (vol<<20) + (vnode<<10) + (uniq).
2. If a fid is the root of a volume, then it is assigned the inode number of the symlink which is its mountpoint.

The kernel should never cache the symlink mountpoints but may see them in case they cannot be covered. Also downcalls are made to purge the symlink fid from the cache, despite the fact that the kernel has never seen the mountpoint fid before.

Kernel code for Coda will need to implement a lookup mechanism to find vnodes (BSD) inodes (Linux) or FileHandles (Win) based on Fid, and it will need to be careful to get things right for volume roots.