The Coda Dir Package

Peter J. Braam Aug 31, 1998

This document describes the directory package used in Coda. Our aim is to describe the interface offered by the dir library, as well as typical usage in Venus and Vice. Areas of improvement are also pointed out. This system is currently being implemented and tested.

Contents

1 Introduction

1.1 Directory Entries

Unix directories are files containing lists of filenames and file identifiers. In Coda the directory entries consist of a name and a DirFed, a pair of a vnode number and uniqifier. The volume to which a directory belongs can be found back through its vnode.

Directories are transported over the net, hence the storage format of all data is in network order. Therefore two variants of the Dir Fid structure exist, DirFid's for use on the host,and Dir NFid's which hold the identifiers innetwork order.

The size of a directory entry is always a multiple of 32 bytes. The first entry contains the fid, the blob number of the next entry on the hash chain and a tag indicating it is the first entry. Subsequent entries contain the remainder of a potentially long name.

1.2 Directory Format

The Coda directory format is inherited from AFS 2 and is somewhat elaborate. Directories have sizes equal to an integral multiple of the directory page size (2048 bytes). They are always continguously allocated (this was changed in 1998). Pages are then subdivided in 64 blobs. Each page in the directory has a PageHeader and a directory has a DirHeader.

The first blob contains the PageHeader, the next 12 blobs in the first page contain the directory header and the remaining blobs are for directory entries. The pageheader contains a bitmap to manage the allocation of blobs, and a free count to indicate how many blobs are still free. The DirHeader contains an allocation map (allomap), which also indicates for each possible page in the direcotry how many blobs are available. It also contains a hastable. Has values are coputed for names and indicate the first blob used by entries of that name.

1.3 Persistent Storage for Directories

Ultimately, clients and servers hold directory data in RVM.

1.3.1 Directories on the client

Venus will store directory data as a contiguous blob of memory and expand such a region when a directory needs to grow. A Venus fso object contains a field of type VenusData, which is a union of file, directory and symlink data.

```
union VenusData {
    int havedata;       /* generic test for null pointer (pretty gross, eh) */
    CacheFile *file;    /* VnodeType == File */
    VenusDirData *dir;  /* VnodeType == Directory */
    char *symlink;      /* VnodeType == SymbolicLink */
};
```

For a directory Venus has:

```
struct VenusDirData {
        struct DirHandle dh; /* contains pointer to Coda format directory */
        unsigned udcfvalid : 1; /*T*/   /* Unix format directory in UFS. */
        CacheFile *udcf;   /*T*/
        int padding;       /*T*/
};
```

Venus will at appropriate times write a Unix style directory into a container file, so that the kernel can perform its readdir operations directly on the container.

1.3.2 Server directory storage

Server directory vnodes point to directory inodes. Such inodes contain the pages of the directories. Since directories and directory inodes are copy-on-write objects with respect to cloning volumes, several vnodes (in different volumes) can point to a single directory inode.

The **VnodeDiskObjectStruct** in RVM contains a pointer **Inode** to a directory inode structure:

```
struct DirInode {
        void *di_pages[DIR_MAXPAGES];
```

```
        int  di_refcount;             /* for copy on write */
};
```

The server stores the directory in pages in RVM, but never does directory operations on the RVM pages. A directory inode contains the page map `di_pages` for the directory. The array contains pointers to RVM storage for the individual pages of the directory. When concatenated into a single buffer they are the objects Venus fetches from the server.

The server makes a copy of the directory data in VM, operates on that copy and copies the entire directory back to RVM when finished.

Since directory contents on the server are copy on write, a directory handle is needed to count the references to the directory while it is in use. These handles are held in a hash table in the server.

These arrange for RVM storage of directories.

While the server is running vnodes are hashed in VM and the **Vnode** structure contains a pointer **dh** to a directory handle. The directory handles themselves sit in a hashtable:

```
struct dllist_head dcache[DCSIZE];
struct dllist_head dfreelist;

struct DCHashEntry {
        struct dllist_head    dc_listhead;
        int                   dc_count;  /*how many vnodes are referencing us */
        struct DirHandle      dc_dh;
        PDirInode             dc_pdi;
};
```

1.4 C Structures Describing the Directory Format

The following structures describe the directory format:

```
struct PageHeader {
    log tag;
    char freecount;   /* duplicated info: also in allomap */
    char freebitmap [EPP/81];
    char padding[PH_PADSIZE];
};

struct DirHeader {
    struct PageHeader dirh_ph;
    char dirth_allomap[MAXPAGES];  /* one byte per 2K page */
    short dirh_hashTable[NHASH];
};

/* A file identifier in host order */
struct DirFid {
    long dnf_vnode;   /* file vnode */
```

```
    long dnf_unique;  /* file uniguifier */
};

/* File identifier in network order */
struct DirNFid {
    long dng_vnode; /* file vnode */
    long dnf_unique;  /* file uniguifier */
};

struct DirEntry {
    char flag;
    char length;  /* currently unused */
    short next;
    struct DirNFid fid;
    char name[16];
};

struct DirXEntry {
    char name[32];
};
```

1.5 Example of the Directory Layout:

The dirtest program shows a simple directorylayout below:

```
DIR: 0x501121c, LENGTH: 2048

HASH TABLE:
(1 19) (46 13) (68 14) (104 20)

ALLOMAP:
(0 41)

PAGEHEADERS;
page 0, tag 1234, freecount 41, st 23, bitmap:
1111111101111111111111100000000100000000000000000000000000000000

CHAINS:
Chain: 1
thisblob: 19 next: 18, flag 1 fid: (8.8) 27
thisblob: 18 next: 17, flag 1 fid: (8.8) 50
thisblob: 17 next: 0, flag 1 fid: (8.8) 225
Chain: 46
thisblob: 13 next: 0, flag 1 fid: (9.9) .
Chain: 68
thisblob: 14 next: 0, flag 1 fid: (9.9) ..
Chain: 104
```

```
thisblob: 20 next: 0, flag 1 fid: (4.5) this is a veryveryveryverylongname
```

Careful investigation will show that more than a single blob is ocupied by the very long entry.

2 Interfaces

Venus and Vice will access directories through the directory handle api. A directory handle facilitates: - locking of directories - managing rvm and non rvm copies of the data

2.1 DirHandle API

Directory handles are opaque but have a simple internal structure:

```
struct DirHandle {
    lock           dh_lock;
    PDirHeader     dh_vmdata;
    PDirHeader     dh_rvmdata;
    int            dh_refcount;
};
```

2.1.1 void DH_LockW(PDirHandle);

Lock the direcotry for writing

2.1.2 void DH_LockR(PDirHandle);

2.1.3 void DH_UnLockW(PDirHandle);

release a write lock

2.1.4 void DH_UnLockR(PDirHandle);

release a read lock

2.1.5 void DH_Init(PDirHandle dh);

initializes the lock in a directory handle.

2.1.6 PDirHandle DH_New(int in_rvm, PDirHeader vmdata, PDirHeader rvmdata);

Instantiate a new directory handle in in VM or RVM and attach directory header pointers to the fields. This routine is probably not useful to Venus and Vice since DirHandle structures are embedded in other structures.

2.1.7 int DH_Length(PDirHandle);

return the length of the directory in bytes. This will always be a multiple of the DIR_PAGESIZE (2048 bytes)

2.1.8 int DH_Create (PDirHandle dh, char *entry, struct DirFid *fid);

Lookup an entry in the directory dh and fill in the fid. This call returns 0 upon success.

2.1.9 int DH_Delete(PDirHandle dh, char *entry);

Remove entry from dh. Return 0 upon success.

2.1.10 void DH_Free(PDirhandle dh, int in_rvm);

Free the data pointed toby (r)vm data field in the handle.

2.1.11 void DH_Alloc(PdirHandle dh, int size, int in_rvm);

Allocate (recoverable) memory of size size in the DirHeader pointers in the dh.

2.1.12 PDirHeader DH_Data(PDirHandle dh);

Return a pointer tothe dirheader in the dirhandle.

2.1.13 void DH_Print(PDirHandle dh);

Print the complete structure of the directory

2.1.14 int DH_MakeDir(PDirHandle dh, PDirFid me, PDirFid parent);

Create a new directory. This call will allocate memory in (r)vm based on the return vlaue of dir_rvm()

2.1.15 int DH_EnumerateDir(PDirHandle dh, int (*hookproc) (struct DirEntry *de, void *hook) , void *hook);

This call williterate through the directory entries and apply hookproc to each, with parameter the dir entry in question and the hook pointer. It will quit when all entries have been processed or when hookproc returns a nonzero value.

The final return value is returned by DH_EnumerateDir.

2.2 DIR api

The DIR api handles the actions on the directory data blobs. These routines are probably only used by the DH api and not elsewhere in the code. This important since it will make switching to a new directory type easy.

The internalhandling of the directory buffers is important. New entries can be created and the routine dir_FindBlobs will locate a sufficient number of blobs. If none are available anymore in the directory the routine dir_AddPage is called. AddPage reallocates a directory buffer which is one pagesize (2048 bytes) larger and then copies the old data into the new one (dir_Extend). Of course this requires a rvm_set_range call when the data is in RVM. After this is done, it adjusts the bitmaps and free counts.

2.2.1 extern int DIR_init(int data_location);

Tells Venus/Vice if directory data is manipulated in RVM or VM. Venus will directly doits transactions on the RVM data (following the pointer in DH). The server manipulates a VM handle and commits the entire blob at commit time.

2.2.2 extern void DIR_Free(struct DirHeader *, int);

2.2.3 extern int DirHash (char *);

Hash value of the name – could probably be static.

2.2.4 extern int DirToNetBuf(long *, char *, int, int *);

This routine is probably going to be obsolete.

2.2.5 int DIR_MakeDir(struct DirHeader **dir, struct DirFid *me, struct DirFid *parent

Make a new directory with "." and ".." entries as indicated by me and papa.

2.2.6 int DIR_LookupByFid (PDirHeader dhp, char *name, struct DirFid *fid);

The normal important lookup of a name in a directory.

2.2.7 int DIR_EnumberateDir(struct DirHeader *dhp,

2.2.8 int (*hookproc) (struct DirEntry *de, void *hook), void *hook);

Enumerate untilhookproc returns non-null. Return final return value of hookproc.

2.2.9 int DIR_Create(struct DirHeader **dh, char *entry, struct DirFid *fid);

Create a directory entry

2.2.10 int DIR_Length(struct DirHeader *dir);

Length of the directory

2.2.11 int DIR_Delete(struct DirHeader *dir, char *entry);

delete an entry

2.2.12 void DIR_PrintChain(PDirHeader dir, int chain);

print out a hash chain in a directory

2.2.13 int DIR_Convert (PDirHeader dir, char *file);

Convert a directory to Venus BSD format for parsing by the kernel.

sect2>static int dir_NameBlobs (char *name)

How many blobs are needed for this name?

2.2.14 static int dir_FindBlobs (struct DirHeader **dh, int nblobs

Find a bunch of contiguous entries; at least nblobs in a row.

2.2.15 static int dir_AddPage (struct DirHeader **dh)

Add a page to a directory. This routine has two components. It calls dir_Extend and when the new buffer is made available adjusts the bitmaps, freecounts, etc.

2.2.16 static struct DirHeader *dir_Extend(struct DirHeader *olddir, int in_rvm)

This is the routine which allocates a new buffer (exactly one page larger than the former size of the directory) and frees the previous buffer. Of ocurse it uses a transaction in RVM if this is where data is held.

2.2.17 static struct PageHeader *dir_Page(struct DirHeader *dirh, int page)

Return the PageHeader for the page `page` in the directory.

2.2.18 static void dir_Freeblobs(struct DirHeader *dhp, register int firstblob, int nblobs)

This is called delete. It makes a group of blobs available again for re-allocation to new entries.

2.3 FID api

There are a variety of fid supporting routines needed in the dir package. The primary

1. ViceFids in use by Venus and Vice are structures containing more information (such as cell and volume information) than fids stored in Coda directories which only contain a vnode and uniquifier.

2. Directories are stored on disk in network order. This means that conversion to host order is appropriate and needs a few supporting routines.

2.3.1 void FID_VFid2DFid(struct ViceFid *vf, struct DirFid *df);

Copy a ViceFid's vnode and unique fields into a directory fid.

2.3.2 void FID_DFiD2VFid(struct DirFid *df, struct ViceFid *vf);

Copy a directory Fid's vnode and unique field into a ViceFid.

2.3.3 void FID_CpyVol (struct ViceFid *taget, struct ViceFid *source);

2.3.4 void FID_Int2DFid (struct DirFid *fid, int vnode, int unique);

Load integers into a fid

2.3.5 void FID_NFid2Int (struct DirNFid *fid, long *vnode, long *unique);

Take vnode and unique out of a directory network fid.

2.3.6 void FID_PrintFid(struct DirFid *fid);

Print a fid

2.3.7 char *FID_s(struct ViceFid *fid);

2.3.8 char *FID_s2(struct ViceFid *fid);

Give a printed version of the fid in a string (statically allocated). Allows for ....

2.3.9 static void fid_Fid2NFid (struct DirFid *fid, struct DirNFid *nfid)

Convert fid to network order

2.3.10 static void fid_NFid2Fid (struct DirNFid *nfid, struct DirFid *fid)

Convert fid to host order

2.3.11 static int fid_FidEqNFid(struct DirFid *fid, struct DirFid *nfid)

Is a network fid equal to a fid?

2.4 Directory Inode API

2.4.1 PDirHeader DI_DiToDh(PDirInode pdi)

Copy the pages in a directory inode to a contiguous area in VM for which memory is malloced. The directory header (i.e. the first page of this data in VM) is returned.

2.4.2 DI_DhToDi(PDirHandle pdh, PDirInode pdi)

Store the pages found in a VM area in RVM. This involves setting ranges and potentially allocating new pages in persistent storage for the object.

2.4.3 DI_Inc(PDirInode)

Increase the reference count to the directory (after a clone for example) in persistent storage.

2.4.4 DI_Dec(PDirInode)

Decrease the RVM refcount on the directory. Remove the directory when this falls to 0.

2.4.5 DI_Count(PDirInode)

Returns the reference count on a directory.

2.4.6 DIPages(PDirInode)

Return the number of pages in the directory.

2.4.7 void *DI_Page(PDirInode pdi, int page)

Return a pointer to a page.

2.4.8 void DI_Copy(PDirInode oldinode, PDirInode *newinode)

Make a copy of a directory inode and the pages. Needed by copy on write.

2.5 Directory Handle Cache API

2.5.1 void DC_HashInit()

Set up the DH hashtable.

2.5.2 void DC_Put(PDirHandle dh)

Decrease the refcount on a hashed directory handle. Remove it when it falls to 0.

2.5.3 struct DirHandle *DC_Get(PDirInode pdi)

Increase the reference count on a cached directory handle or fetch the handle and the data from RVM if not hashed yet.

3 Lookup handling

Intimately related to directories is the handling of lookups. In Coda this is somewhat involved due to volume mountpoints, and objects in conflict. During regular use, Venus performs the lookups.

For every ViceFid an inode number can be computed. The routine coda_f2i does this translation. A lookup of a name will return this inode number *except* when:

1. The name looked up is ".." and it is contained in the root directory of a volume. Instead the lookup is returns the value returned by looking up "." in the directory containing the mount point. Note that this may be recursive.
2. The name looke dup is "." and it is contained in the root directory of a volume. In this case the value returned is the value obtained by looking up the mountpoint in the parent directory of ".".

4 Directory handling in the server

The server has a fairly complicated directory handling system which we describe here.

4.1 Persistent storage of directory data

A directory is identified by a vnode, and the vnode's data is stored in a VnodeDiskObject (cvnode.h). The `type` of the vnode will be directory and the `inodeNumber` will be an RVM pointer to a *DirInode*. The dir inode contains an array of page addresses and a reference count to identify the copy on write references to the directory.

When the pages are placed in a contiguous buffer, the directory has the standard structure described in `codadir.h`.

4.2 Directory handles in the server

A key difference between server and client handling of directories is that there can be more than a single vnode referencing the same directory inode. In the client only a single fso can point to a dir header. Generally the vnodes referencing the directory inode will lie in different volumes, one being a read only clone of another volume. As a result the directory data is an object which needs to be treated independently of the vnode. For this we use a cache of DirHandles.

The interface to the cache is extremely simple: `DC_HashInit` initializes the hash table. `DC_Get` takes as argument a pointer to a DirInode and enters it in the cache if it wasn't in yet, returning the pointer to the *DirHandle*. The cache entry has a reference count which indicates how many VM vnodes are referencing the directory inode. The in-core copy of the vnode will have its *DirHandle* field set to the result returned by DC_Get. When the directory data for a Vnode needs to be available, DC_Get is called with argument `vnode->disk.inode.` Previously this was achieved by calling SetDirHandle.

At this point in time a VM copy of the directory data is available for manipulation by the fileserver. The routines DH_Lookup, DH_Delete, DH_Create etc. are available to search and modify the VM copy of the directory. Such changes obtain a write lock on the DirHandle since the changes will be visible to other threads accessing the directory.