Open Software Foundation T. Anderson (Transarc)
Request For Comments: 51.3
August 1996

DFS CHANGES TO SUPPORT A SCALAR 64-BIT TYPE

1. INTRODUCTION

This document describes a cleanup made to the Transarc DFS code base to make support for large data objects easier. This work was inspired by the earlier revisions of this RFC and specifically the concrete work at DEC and Cray to export large files with DFS.

The approach we took was to incorporate the wide ranging code changes described by Steve Strange in [RFC 51.1], which allowed 64-bit quantities to be represented efficiently using a scalar type when one is available. However, we needed to ensure backward compatibility with existing persistent data structures, which meant that the scalar type could not be used when an architecture independent format was needed. We also made several different choices for names of global types and macros to minimize the possibility of name space collisions.

We also incorporated the changes Steve suggested in [RFC 51.2]. Those changes which affected the DFS protocol were made earlier. There remained several internal changes, that should making ports to 64-bit architectures easier. These changes involve modifications to the DFS file exporter (PX) so it remembers the maximum file size supported by the client. Analogous changes allow the DFS cache manager (CM) to track the maximum file size supported by the server.

2. OUTLINE OF WORK

The work described here has several general components: type changes, hyper macro changes, platform independence considerations, and maximum file size tracking. The platform independence problem is further divided into: RPC interfaces, Episode disk structures, ubik databases for the fldb and backup system, and tape formats used by the backup system.

Existing DFS code often represented 64-bit quantities using a `hyper type that was implemented as a structure composed of two 32-bit integers. To hide this implementation, a collection of macros was provided to manipulate hypers`. However, the use of these macros was spotty, at best. To further confuse things, a similar type, called `afsHyper`, was used in some code and another set of macros existed to manipulate this type. Generally these types are called ‘‘hyper’’ ’s (as opposed to ‘‘`hyper`’’ ’s) and the macros are referred to collectively as ‘‘hyper macros’’.

2.1. Type Changes

An important part of the work was to provide a single hyper type, called `afs_hyper_t`, to represent 64-bit quantities wherever possible. To support both scalar and aggregate implementations of the type, hypers must be uniformly accessed via a consistent set of macros.

Several types were identified by [RFC 51.1] as containing 64-bit quantities that were not represented in a natural way. These were the `afsToken` and `afsRecordLock` types, which, for historical reasons, represented file offsets as two non-contiguous 32-bit integers.

The type `tkm_token_t` largely duplicated the functionality of the RPC-defined `afsToken` type, so these two types were combined in a new type called `afs_token_t`.

For consistency, the record lock type was renamed to `afs_recordLock_t`.

2.2. Hyper Macro Changes

The bulk of the changes of related to the use of hypers. To ensure that DFS code was portable between platforms with different representations of the hyper, all references to hypers were changed to use the appropriate macros. Most important, explicit reference to the ‘‘low’’ and ‘‘high’’ members of the (old) hyper structure were eliminated in favor of accessing macros `AFS_hgetlo()` and `AFS_hgethi()`. These members cannot exist on platforms that use a scalar 64-bit type. The accessing macros replace the awkward `hget32()` and `hget64()`.

To minimize name space collisions the hyper handling macros were all renamed to use the `AFS_` prefix.

The `hset()` macro was eliminated because the compiler can perform assignment efficiently for both scalar and non-scalar representations.

Several other minor changes were made to simplify the list of hyper macros to make them easier to understand and use. A full list of the macros appears below.

2.3. Platform Independence

The new hyper type is not suitable as an external representation for at least two reasons. First, the platform dependent implementation of the hyper implies that the byte order is not fixed. Second, the scalar type can have different alignment requirements from a structure of two 32-bit integers, so a structure containing a hyper will pack differently depending on whether the hyper is implemented as a scalar or an aggregate. A good external representation needs to have a stable, well-specified packing and byte order.

Therefore, to maintain upward compatibility another type was used to specifying externally visible or persistent formats. To meet these requirements, the type `dfsh_diskHyper_t` was defined in `file/util/hyper.h`. It comes with two sets of accessing macros depending on whether host byte order is acceptable (as in Episode) or whether platform independent byte order is necessary (as with ubik databases and the on-tape structures used by the backup system).

The new `afs_hyper_t` type is widely used in RPC functions. In that capacity the platform independence is provided by the RPC system using the `[represent_as]` mechanism specified in the `file/config/common_data.acf` file. This automatically maps between types as explained in [RFC 51.1] (e.g., `afsHyper` on the wire and `afs_hyper_t` in memory). Except for the type name changes this work was implemented as described.

2.4. Maximum File Size Tracking

A collection of changes was suggested in RFC 51.2. One change involves an enhancement to the protocol to exchange maximum supported file size information between the client and server. The minimal support for this feature was added to DFS some time ago and is present in the OSF DCE V1.2 code. This preliminary work was extended to make future ports to 64-bit platforms easier. Generally these changes followed those made by DEC, but several additional changes were made and a few things were implemented a bit differently than in the DEC code. These changes should interoperate with older 32-bit systems, and with the 64-bit systems deployed by DEC and Cray.

Additional members were added to the host structures used by the PX and CM to track the maximum file size supported by the other machine. This is used to provide reasonable behavior when clients and servers have different capabilities. This information allows enhanced clients to avoid writing a file longer than the server can support. The CM returns `EFBIG` when the application attempts to extend the file beyond this limit.

Several changes were made to token management. The special value used to represent a ''whole file'' was changed from 2^31−1 to 2^63−1. To make this work correctly with older systems, two special mappings are performed. On the client, the byte ranges of tokens returned from an old server are mapped from 2^31−1 to 2^63−1. On the server, the byte ranges of tokens being returned are modified so that 2^63−1 is mapped to 2^31−1.

Several bugs were fixed (e.g., OT13445) and shortcomings addressed (e.g., OT8872) which affected 64-bit operations.

3. ISSUES NOT ADDRESSED

Several related issues were not addressed by this work:

(a) On some platforms the type `int` and type `long` are different sizes. Because this is not true on any of the platforms in use at Transarc, some errors due to mixing these types are present. There has been no effort in this work to weed out those errors.

(b) Transarc's DFS does not support files longer than 2^31−1 bytes. There are several parts to remedying this. Part of the the problem is due to the native OS (e.g., neither SunOS 5.4 nor AIX 3.2 support large files in their virtual memory systems). Presumably DEC and Cray have removed these limitations. However, numerous components within DFS have limitations that would prevent large files from working immediately. Mostly these problems are minor, but a significant testing effort would be needed to verify large file support.

(c) No interoperation testing between Transarc DFS code and DEC and/or Cray products has been done.

4. DETAILED CHANGES

Next is a detailed description of the changes that were made, following the outline given above.

4.1. Changes to Types

Several important types that contained 64-bit quantities were renamed or combined. A few changes in the member names were also made. The types now have the following names:

```
   NEW TYPE              REPLACES
afs_hyper_t          hyper, afsHyper
afs_token_t          afsToken, tkm_token_t
afs_recordLock_t     afsRecordLock
```

Here are the changed member names:

```
   NEW MEMBER                      REPLACES
afs_token_t.expirationTime    tkm_token_t.expiration
afs_token_t.beginRange        tkm_token_t.startPosition
afs_token_t.endRange          tkm_token_t.endPosition
```

Obsolete members representing parts of hypers were removed:

```
   DELETED MEMBER                   NOW PART OF HYPER
afsToken.beginRangeExt           afs_token_t.beginRange
tkm_token_t.startPositionExt

afsToken.endRangeExt             afs_token_t.endRange
tkm_token_t.endPositionExt

afsRecordLock.l_start_pos_ext    afs_recordLock_t.l_start_pos
afsRecordLock.l_end_pos_ext      afs_recordLock_t.l_end_pos
```

4.2. Hyper Macro Descriptions

Here is the list of macros provided for manipulating hypers.

(a) `int AFS_hcmp(afs_hyper_t a, afs_hyper_t b)` — Returns a (negative, zero, or positive) value if `a` is (less, equal, or greater) `b`. This is an unsigned comparison. In other words, `(a` *`oper`* `b)` can be expressed as `(AFS_hcmp(a, b)` *`oper`* `0)` where *`oper`* is one of { `<`, `<=`, `==`, `>`, `>=` }.

(b) `int AFS_hcmp64(afs_hyper_t a, u_int32 hi, u_int32 lo)` — like `AFS_hcmp()` but compares `a` with `(hi<<32 + lo)`.

(c) `int AFS_hsame(afs_hyper_t a, afs_hyper_t b)` — Returns a non-zero value (TRUE) iff `a` has the same value as `b`.

(d) `int AFS_hiszero(afs_hyper_t a)` — Returns TRUE iff `a` is zero.

(e) `int AFS_hfitsinu32(afs_hyper_t a)` — Returns TRUE iff 0 <= `a` < 2^32.

(f) `int AFS_hfitsin32(afs_hyper_t a)` — Returns TRUE iff –2^31 <= `a` < 2^31.

(g) `void AFS_hzero(afs_hyper_t a)` — Sets `a` to zero.

(h) `u_int32 AFS_hgetlo(afs_hyper_t a)` — Returns the 32 least significant bits of `a`.

(i) `u_int32 AFS_hgethi(afs_hyper_t a)` — Returns the 32 most significant bits of `a`.

(j) `void AFS_hset64(afs_hyper_t a, u_int32 hi, u_int32 lo)` — sets `a` to `(hi<<32 + lo)`. So that `AFS_hset64(h, AFS_hgethi(h), AFS_hgetlo(h))` leaves `h` unchanged.

(k) `AFS_HINIT(u_int32 hi, u_int32 lo)` — An initializer of type `afs_hyper_t`.

(l) `void AFS_hleftshift(afs_hyper_t a, u_int amt)` — Shifts `a` left by `amt` bits; where 0 < `amt` < 64.

(m) `void AFS_hrightshift(afs_hyper_t a, u_int amt)` — Logically shifts `a` right by `amt` bits; where 0 < `amt` < 64.

(n) `void AFS_hset32(afs_hyper_t a, int32 i)` — Sets `a` to the 64-bit sign extended value of `i`. If `i` is unsigned use `AFS_hset64(a, 0, i)`.

(o) `void AFS_hadd32(afs_hyper_t a, int32 i)` — Adds `i` to `a`.

(p) `void AFS_hadd(afs_hyper_t a, afs_hyper_t b)` — Adds `b` to `a`.

(q) `void AFS_hsub(afs_hyper_t a, afs_hyper_t b)` — Subtracts `b` from `a`.

(r) `void AFS_hnegate(afs_hyper_t a)` — Sets `a` to its twos complement.

(s) `void AFS_HOP(afs_hyper_t a,` *op*`, afs_hyper_t b)` — like `a = a` *op* `b`, where *op* should be one of { `"|"`, `"&"`, `"^"`, `"&~"` }.

(t) `void AFS_HOP32(afs_hyper_t a, <op>, u_int32 u)` — Works like `AFS_HOP` except that `u` is logically extended to 64 bits by prepending 32 zero bits (i.e., no sign extension).

(u) `void AFS_hincr(afs_hyper_t a)` — Short for `AFS_hadd32(a, 1)`.

(v) `void AFS_hdecr(afs_hyper_t a)` — Short for `AFS_hadd32(a, -1)`.

(w) `int AFS_hissubset(afs_hyper_t a, afs_hyper_t b)` — Returns TRUE iff all the bits set in `a` are also set in `b` (`a` is a subset of `b`).

(x) `AFS_HGETBOTH(afs_hyper_t a)` — A short-hand for passing both halves of a hyper to a function, most significant half first. This is convenient for calling `printf()`, for instance.

The following macros were eliminated;

(a) `hset` — Compiler can handle assignments of both scalar and non-scalar types.

(b) `hget32` — Too awkward.

(c) `hget64` — Too awkward.

(d) `hones` — Rarely used; easily replaced with `AFS_hset64(a, -1, -1)`.

(e) `hdef64` — Replaced by `HINIT` which only provides an initializer.

4.3. Platform Independence

The basic tools used to achieve platform independence were defined in `file/util/hyper.h`. The type `dfsh_diskHyper_t` was used whenever 32-bit alignment was necessary to obtain the desired packing.

```
typedef struct {
    u_int32 dh_high;
    u_int32 dh_low;
} dfsh_diskHyper_t;
```

To convert back and forth between `afs_hyper_t` and `dfsh_diskHyper_t` two sets of macros were used. The first set preserves host order and is used by Episode.

```
#define DFSH_MemFromDiskHyper(h, dh) \
    AFS_hset64(h, (dh).dh_high, (dh).dh_low)
#define DFSH_DiskFromMemHyper(dh, h) \
    ((dh).dh_high = AFS_hgethi(h), \
     (dh).dh_low = AFS_hgetlo(h))
```

The second set uses `ntohl`/`htonl` on the halves and was used when architecture neutrality was needed: ubik databases and tapes.

```
#define DFSH_MemFromNetHyper(h, dh) \
    AFS_hset64(h, ntohl((dh).dh_high), ntohl((dh).dh_low))
#define DFSH_NetFromMemHyper(dh, h) \
    ((dh).dh_high = htonl(AFS_hgethi(h)), \
     (dh).dh_low = htonl(AFS_hgetlo(h)))
```

4.4. Episode Changes to Preserve On-disk Format

Several changes were made to the Episode code to insure that the disk representation was unaffected by the changes to the hyper type:

(a) In `fixed_anode.c`, modify `diskAnode` by changing length to be of type `dfsh_diskHyper_t` and renaming it to be `diskLength`. Also change `volId` similarly, though this member is not used.

(b) In `anode.p.h`, add a new length member to the `epia_anode` structure of type `afs_hyper_t`. This will be a copy of the `diskLength` member but maintained in host native format.

(c) Also in `fixed_anode.c`, copy the `diskLength` member to the length member using `DFSH_MemFromDiskHyper()` whenever an anode is initialized from disk: in `Open()` and `epia_Create()`. Make sure length-changing operations affect both members using `DFSH_DiskFromMemHyper()`: `epix_SetLength()`, `epix_MoveData()`, `epix_InsertInline()`, and `SalvageAnodeLength()`.

(d) In `volume.c`, modify the `diskVolumeHeader` structure to use the `dfsh_diskHyper_t` type to represent `ident.id`, `version`, `backingVolId`, and the `upLevelIds` array.

(e) Modify code in `epiv_Create()`, `epiv_GetStatus()`, `epiv_GetIdent()`, `epiv_GetVV()`, `epiv_SetStatus()`, and `epiv_NewVolumeVersion()` to use `DFSH_DiskFromMemHyper()` or `DFSH_MemFromDiskHyper()` as appropriate when copying between the disk volume header and in-memory structures such as `epiv_status` and `epiv_ident`.

(f) In `file.c`, modify the `diskStatus` structure member `volumeVersionNumber` to be a `dfsh_diskHyper_t`.

(g) In `file.h`, modify the fast accessing macros for status fields to use offsets in `diskStatus` which can no longer be assumed to be the same as the offsets in `epif_status`. This is done by defining explicit constants giving the offsets, then changing the asserts done by `epif_Init()` in `file.c` to verify that the offset are correct. Similarly, in `epif_GetStatus()` copy the auxiliary container lengths into the proper fields using a case statement, since the offset arithmetic no longer works.

(h) Modify `epif_CreateE()`, `epif_GetStatus()`, `epif_SetStatusAndMark()`, and `epiz_VerifyFileAux()` to use `DFSH_DiskFromMemHyper()` or `DFSH_MemFromDiskHyper()` as appropriate.

4.5. Fileset Location Server Changes to Preserve Interoperability

The ubik database used to store fileset location information is shared by all flservers using a byte-level replication protocol. This protocol has no knowledge of how the database is represented and so it cannot perform any transformations to fix up byte ordering or member packing differences between architectures. Therefore, the format of the database must be architecture-neutral. The convention with the design of ubik databases has been to use network-byte-order to represent 16 and 32 bit integers in the database. A similar convention is needed for hypers, both to ensure precise packing and to define consistent integer byte ordering.

The strategy was to clearly separate the structures used to represent the database from those used to transmit data to and from clients. The hypers in the database representation were changed to `dfsh_diskHyper_t`. A new header file called `flinternal.h` was created for definitions that are not used by clients of the flserver. The existing `vlentry` structure was moved there and a new `disk_vlheader` structure was defined to match the `vital_vlheader` structure already defined in `fldb_data.idl`. The `disk_vlheader` members `maxVolumeId` and `theCellId` became `dfsh_diskHyper_t`'s, as did the `vlentry` members `volumeId` (an array of length `MAXTYPES`) and `cloneId`.

The flserver code normally converts 16 and 32 bit integers in-place when reading from or writing to the database. However, because of differences in alignment, this will not work with hypers. Therefore, hypers were converted from `dfsh_diskHyper_t` to `afs_hyper_t` at the points of use, with the help of temporary variables when necessary. The conversions were accomplished using `DFSH_MemFromNetHyper()` or `DFSH_NetFromMemHyper()` which parallel the macros used in Episode but which also apply `ntohl()` or `htonl()` to the high and low halves of the 64-bit quantitiy.

Here are the points of use that must be converted:

(a) In `VL_GetNewVolumeId()` and `VL_GetNewVolumeIds()` `maxVolumeId` is increased for new volumes, by converting `maxVolumeId` to an `afs_hyper_t` using `DFSH_MemFromNetHyper()`, bumping it using `AFS_hadd32()` and storing it back into the database header using `DFSH_NetFromMemHyper()`.

(b) The functions `VL_ReplaceEntry()`, `VL_GetStats()`, `vldbentry_to_vlentry()`, `vlentry_to_vldbentry()`, and `vlentry_to_comvldbentry()` just copy structures to or from the database representation.

(c) A new database is constructed in `CheckInit()` and `theCellId` and `maxVolumeId` members are initialized here.

(d) The `FindByID()` function needs to consult the `vlentry`'s id, as do `HashVolid()`, `UnhashVolid()`, and `NextEntry()`.

4.6. Backup Changes to Preserve Ubik Database and Tape Formats

The changes to the backup system had two parts. The first was to ensure that the volume id stored in the ubik backup database was converted to and from a platform independent format. This parallels the changes made to the flserver. In addition, hypers are written to tape in two cases, once in the header of ordinary fileset dumps, and the other when the ubik database is dumped to tape. The dump and restore paths for the latter case are handled differently, but the basic strategy was the same as for the ubik database. New structures were defined to separate the structures recognized by the RPC marshaling code from the structures used to lay out the ubik database and the on-tape format.

The changes for the ubik database were simple because only a single hyper is stored there: the id member of the `volInfo` structure defined in `file/bakserver/database.h`. Its type was changed to `dfsh_diskHyper_t` and conversions were accomplished using `DFSH_MemFromNetHyper()` and `DFSH_NetFromMemHyper()` as appropriate. These conversions appear in `FillVolEntry()`, `VolInfoMatch()`, `GetVolInfo()`, `printVolInfo()`, and `volsToBudbVol()`. The test code duplicates a small amount of this logic. In `test/file/budb/database.h`, the `volInfo` structure must also be changed and the sole use of the member in `test/file/budb/budb_dump.c:print_volInfoBlock()` needs to use `DFSH_MemFromNetHyper()` before printing the volume's id.

The changes for the on-tape format of fileset dumps were also pretty easy because only a single member was affected: the `volumeID` member of the `volumeHeader` structure defined in `file/bubasics/tcdata.p.h`. This member was converted to net-order in `makeVolumeHeader()` instead of in `volumeHeader_hton()` where the other members are converted because hypers cannot be converted in-place as described earlier. The reverse conversion occurs in `PositionTape()` and `fillRestoreBuffers()`. Various routines in `file/butc/recoverDb.c` also need to be able to interpret backup tapes: `PrintVolumeHeader()`, `validVolumeHeader()`, `AddScanToDB()`, and `debugPrintVolumeHeader()` but not `VolHeaderToHost()`.

Saving the ubik database itself to tape is a process that uses completely separate data paths within the backup system. The dump is created by the bakserver using the `BUDB_DumpDB()` RPC which produces a byte stream suitable for writing directly to tape. The byte stream is not interpreted by the RPC marshaling code and so the structures that describe the stream must use types that pack correctly and, of course, network byte ordering is generated by the server. Previously the per volume information was dumped as a `budb_volumeEntry` (but with integers in network byte order). Instead, a new structure was defined in `file/bakserver/budb.idl` called `budb_dbVolume` which is similar to `budb_volumeEntry` except that the volume id is represented as a pair of `unsigned32: struct { unsigned32 dh_high; unsigned32 dh_low; }`. The member names are the same as for the `dfsh_diskHyper_t` type, but that type cannot be directly included in the IDL file (however, the same `DFSH_MemFromNetHyper()` and `DFSH_NetFromMemHyper()` macros will work).

The `budb_dbVolume` structure is filled in by the `bakserver`'s `BUDB_DumpDB()` function using `volsToBudbVol()`. When a ubik database dump is restored the client code reads the tape in `restoreDbDump()` and calls `volumeEntry_ntoh()` as a utility function (even though this function is linked into the bakserver it is never called by the server; probably these functions should be reorganized).

4.7. Maximum File Size Tracking

Three members were added to the structures used by the CM and PX to describe the hosts they communicate with:

```
unsigned32 maxFileParm;          /* value received from host */
afs_hyper_t maxFileSize;         /* max supported by host */
unsigned supports64bit:1;        /* host has 64bit fixes */
```

In the CM these are added to `cm_server` (in `file/cm/cm_server.h`). In the PX these are added to `fshs_host` (in `file/fshost/fshs_host.h`). The `maxFileParm` member preserves the value used to set the maximum file size (encoding described below) so that it can be easily returned in the response to the `SetParams()` call. The `maxFileSize` member is set to the largest file length than can be supported by the remote host.

The `supports64bit` boolean is set to one (TRUE) only if the host provides a valid indication of its maximum file size and claims that it does not need the backward compatibility features provided for older systems. This bit serves to differentiate hosts that can handle 64-bit quantities (whatever their maximum file size) from earlier systems that suffered from various bugs and shortcomings adversely affecting interoperation with 64-bit machines.

There are two, mostly independent, mechanisms for informing the client and server of the maximum file size of the remote host. The first involves the use of the `SetParams()`. The second involves passing this information via parameters to the `TKN_InitTokenState()` and `AFS_SetContext()` functions.

The `SetParams()` function is defined in both the AFS and TKN interfaces; however, while the roles of RPC client and server are reversed for the TKN interface, the definitions of the parameter words are fixed in terms of the DFS client (the cache manager, a.k.a. CM) and DFS server (the file exporter, a.k.a. PX). The `TKN_SetParams()` function recieves the maximum file size of the DFS server on input and returns its own limit as the client's value in the output parameter. The `AFS_SetParams()` function receives the DFS client's maximum on input and returns its limit as the server value in the output parameter.

Both functions take a flag argument, which is basically a sub-opcode. The other argument is a structure of twenty (20) 32-bit words plus a validity mask. Two new words are defined for specifying the maximum file size supported by the client and the server. These are added to `file/config/common_data.idl`:

```
const unsigned32 AFS_CONN_PARAM_MAXFILE_CLIENT = 4;
const unsigned32 AFS_CONN_PARAM_MAXFILE_SERVER = 5;

const unsigned32 AFS_CONN_PARAM_SUPPORTS_64BITS = 0x10000;
```

The `AFS_CONN_PARAM_MAXFILE_CLIENT` value, if valid and non-zero, specifies the maximum file size information for the DFS client. Similarly, `AFS_CONN_PARAM_MAXFILE_SERVER` provides the corresponding information about the DFS server.

The format of both the client and server words is the same. The least significant octet specifies one small integer; call it ‘‘a’’. The next least significant octet specifies another number; call it ‘‘b’’. Subsequent bits are interpreted as flag bits, only one of which is presently defined. The others are zero. Thus 17 bits are defined by this work for communicating the maximum file size; the remaining 15 bits could be used for some future purpose.

The value of the host's maximum file size is 2^a–2^b and is stored in the maxFileSize member of the appropriate host structure. If the `AFS_CONN_PARAM_SUPPORTS_64BITS` bit is set the `supports64bit` member is set to one (TRUE). In addition, if the `maxFileSize` value is not equal to 2^31−1 then `supports64bit` is also set to TRUE. The default value of `maxFileSize` is 2^31−1 and `supports64bit` is FALSE.

For example, DEC presently uses 0x132c which expresses a value of 0xffffff80000 (2^44−2^19), Cray uses 0x13f or 0x7ffffffffffffffe, and Transarc uses uses 0x1001f meaning 0x7fffffff with 64bit support. Older systems use 0x1f or provide no value; they are assumed to have a maximum file size of 2^31−1 and get the benefit of the backward compatibility features.

A new value for the flag parameter to `SetParams()` should be added to `file/fsint/afs4int.idl`:

```
const unsigned32 AFS_PARAM_SET_SIZE = 0x3;
```

The behavior of this flag value should be the same as for the value `AFS_PARAM_RESET_CONN (0x1)`. This new flag value is needed because the DEC and Cray ports only interpret the `MAXFILE` values if the flag has this value.

Regardless of the value of the `Flags` parameter, if the input value is valid (the corresponding bit in `afsConnParams.Mask` is set) the caller's host structure (`fshs_host` or `cm_server`) should be updated.

On output `SetParams()` should set both client and server words in the output structure if it knows them. It should do this regardless of the `Flags` value and whether or not an input value was specified. This returns its maximum file size to the caller and confirms receipt, possibly via some earlier call, of the caller's maximum.

When the `SetParams()` call returns the remote host's value is extracted from the output `afsConnParams` structure and processed as described above.

The `TKN_SetParams()` function is not called at present, but is instantiated in `file/cm/cm_tknimp.c`, `file/rep/rep_main.c`, `file/userInt/fts/volc_tokens.c`, and `test/file/itl/fx/itl_fxToken.c`. Only the first of these three and the `SAFS_SetParams()` function defined in `file/px/px_intops.c`, do the processing just described, the others just return `EINVAL`.

The DFS Client makes the `AFS_SetParams()` call to determine the server's maximum file size in `cm_RecoverTokenState()` if it does not already know the size via an earlier `AFS_SetContext()` / `TKN_InitTokenState()` exchange. This ensures that the CM knows whether the server can support 64-bit token ranges, before restoring its token state with that server. Because the server may have rebooted since we last contacted it, `cm_ConnAndReset()` resets `maxFileParm`, `maxFileSize`, and `supports64bit` before calling `cm_RecoverTokenState()`. That function calls a new function, `cm_GetServerSize()` defined in `file/cm/cm_tknimp.c`, which makes the actual call to `AFS_SetParams()` if `maxFileParm` is zero and passes the resulting server size parameter to the same function used by `STKN_SetParams()`, `STKN_InitTokenState()`, and `cm_QueuedRecoverTokenState()`.

The changes to `TKN_InitTokenState()` and `AFS_SetContext()` are simpler; they both have serveral spare parameters. One spare input parameter to each is used to pass the maximum file size information of the caller to the remote host.

In `file/fsint/tkn4int.idl` the description of `TKN_InitTokenState()` is changed so that the `spare1` input parameter becomes `serverSizesAttrs`:

```
error_status_t TKN_InitTokenState
(/* provider_version(1) */
    [in]  handle_t    h,
    [in]  unsigned32  Flags,
    [in]  unsigned32  hostLifeGuarantee,
    [in]  unsigned32  hostRPCGuarantee,
    [in]  unsigned32  deadServerTimeout,
    [in]  unsigned32  serverRestartEpoch,
```

```
        [in]  unsigned32  serverSizesAttrs,
        [in]  unsigned32  spare2,
        [in]  unsigned32  spare3,
        [out] unsigned32 *spare4,
        [out] unsigned32 *spare5,
        [out] unsigned32 *spare6
    );
```

This function is instantiated as `STKN_InitTokenState()` in four places where the function definition needs to be updated: `file/cm/cm_tknimp.c`, `file/rep/rep_main.c`, `file/userInt/fts/volc_tokens.c`, and `test/file/itl/fx/itl_fxToken.c`. The parameter is ignored in all of these except `cm_tknimp.c`. In that case, the received value is processed in the same way as described above for the `SetParams()` functions.

This function is called only from `tokenint_InitTokenState()` in `file/fshost/fshs_hostops.c`. The seventh parameter is changed from zero to the local maximum file size information encoded as described above.

The `TKN_InitTokenState()` call by the server is triggered when the client calls `AFS_SetContext()` to initialize a new connection and the server can find no information about the client. This function is defined in `file/fsint/afs4int.idl` and instantiated in `file/px/px_intops.c`. The new definition changes the spare input parameter `parm6` into `clientSizesAttrs`.

```
    error_status_t AFS_SetContext
    (/* provider_version(1) */
        [in] handle_t    h,
        [in] unsigned32  epochTime,
        [in] afsNetData *callbackAddr,
        [in] unsigned32  Flags,
        [in] afsUUID    *secObjectID,
        [in] unsigned32  clientSizesAttrs,
        [in] unsigned32  parm7
    );
```

The callers of `AFS_SetContext()` in `file/cm/cm_conn.c` (from both `cm_ConnAndReset()` and `cm_ConnByHost()`), `file/rep/rep_host.c`, `file/userInt/fts/volc_tokens.c`, and `test/file/itl/fx/itl_fxAPI.c` pass their local maximum file size information (encoded as above) as the second to last parameter. On receipt of the `AFS_SetContext()` the PX processes the `clientSizesAttrs` parameter as described for `AFS_SetParams()`.

4.8. Enforcing Maximum File Sizes

The maximum file size information communicated via the mechanisms just described are used in two different ways. The new CM uses the server's maximum length to prevent the client application from creating a file larger than can be stored back to the server. This is important because the store-back process happens largely in the background and errors cannot be reliably communicated to the application. To accomplish this `cm_write()` and `cm_setattr()` return `EFBIG` if these functions would try to extend the length past what the server can support.

The other area involves treatment of files larger than a client can handle. We follow the approach Cray took, which is also recommended by the Large File Summit in its proposal to X/Open [LFS 96]. This approach conservatively returns errors to applications that are unaware of the existence of files larger than 2^31−1 bytes. In DFS there are two layers at which we must apply this protection. If the DFS client is old it is protected by the PX which hides the large files from it. However, new clients see large files and protect their callers by hiding large files from them.

For old clients referencing large files, the server returns `DFS_EOVERFLOW` from `SAFS_FetchStatus()` and `SAFS_GetToken()`. In response to `SAFS_Lookup()`, `SAFS_LookupRoot()` and `SAFS_BulkFetchStatus()` calls, the server returns invalid status by setting `fileType` to `Invalid` and refuses to return tokens for these files. Other status returning operations (e.g., `SAFS_Rename()`) return invalid statuses for these files and other operations that can return tokens (e.g., `SAFS_FetchData()`) do not do so for these large files.

The new Transarc reference port of the CM, whose maximum file size remains 2ˆ31−1, is modified to remember which files have lengths too long to represent. It returns the appropriate errors to applications from vnode operations on those files.

To do this a new bit is defined for the scache states word in `file/cm/cm_scache.h` called `SC_LENINVAL`. This bit is set by `cm_MergeStatus()` in `file/cm/cm_vnodeops.c` when a valid status block is received. Its value is one if and only if the length is greater than 2ˆ31−1. The token management functions `cm_HaveTokensRange()` (which was called `cm_HaveTokens()`) and a new function `cm_HaveTokens()` report that the `TKN_STATUS_READ` token for these files is unavailable. In addition, the functions `cm_GetTokens()` and `cm_GetTokensRange()` return `EOVERFLOW` (or `EFBIG` if `EOVERFLOW` is undefined) when a `TKN_STATUS_READ` token is requested for such a file. This error is propagated up to the vnode operations such as `cm_getattr()`.

In a similar vein, `SAFS_Readdir()` and `SAFS_BulkFetchStatus()` are modified to return `DFS_EOVERFLOW` to old clients if the `NextOffsetp` parameter would be larger than 2ˆ31−1. New clients receive (the possibly too large) `NextOffset` intact, and return `EOVERFLOW` from `cm_FetchDCache()` in `file/cm/cm_dcache.c` and from `cm_BulkFetchStatus()` in `file/cm/cm_dnamehash.c` if `NextOffset` is larger than their local maximum file size.

As a safety check, the PX also checks for requests that would increase the length past what it can handle and rejects these with `EFBIG`. The checks are performed at the beginning of `SAFS_FetchData()`, `SAFS_StoreData()`, `SAFS_Readdir()`, `SAFS_BulkFetchStatus()`, and `px_PreSetExistingStatus()`. The latter function handles status setting operations that can change the length, such as `SAFS_StoreStatus()`.

The Large File Summit proposes returning `EOVERFLOW`, a new error code, when a file's length is too large to represent. At the present, the Solaris platform defines `EOVERFLOW` but the others do not. In any case, these other platforms will not use the same value, so DFS needs to define a platform independent value for this error as we have done with other error codes. `DFS_EOVERFLOW` was added to the list in `file/osi/osi_dfserrors.h` where it is defined as 94 (decimal). The mapping table for each platforms was updated so that this is mapped to `EOVERFLOW` on Solaris and `EFBIG` otherwise.

4.9. Token Byte Range Changes

Support for large files also depends upon being able to represent tokens covering any byte range in large files. While the token manager has no trouble with byte ranges beyond 2ˆ31−1, limits on this range do appear in other places. Some of these are due to limits of the local operating system but others are in platform independent code. This code needs to be made 64-bit ready.

4.9.1. Whole file tokens

Many places in the code use the value 2ˆ31−1 to represent the maximum possible file offset when specifying a byte range to cover a whole file. To remedy this the default byte range for tokens is changed from 0..2ˆ31−1 to 0..2ˆ63−1. This applies to whole file tokens requested by the CM (e.g., `cm_GetTokens()` defined in `file/cm/cm_tokens.c`), to tokens optimistically granted by the PX (e.g., using the macro `InitToken()` defined in `file/px/px_intops.c`; this macro was formerly called `tkm_initToken()` and was defined in `file/tkm/tkm_tokens.h`, even though it was only used in `px_intops.c`) and for non-range file and volume tokens.

4.9.2. TKC byte range representation

The TKC module manages tokens for access to local file systems that may also be exported. The representation it uses for byte ranges is changed to use a newly defined type consisting of a pair of hypers.

In `file/tkc/tkc.h` a new type is defined:

```
typedef struct {
    afs_hyper_t beginRange;
    afs_hyper_t endRange;
```

```
} tkc_byteRange_t;
```

This type replaces the use of hypers to represent byte ranges: in `struct tkc_sets` and as a parameter to `tkc_Get()`, `tkc_GetToken()`, `tkc_HaveTokens()`, `tkc_GetLocks()`, and `tkc_Putlocks()`, and in callers of `tkc_GetLocks()` and `tkc_PutLocks()` in the platform specific xvnode implementations of the vnode file lock functions.

Since the TKC does not use byte ranges on data tokens, the only significant changes are in `tkc_PutLocks()` defined in `file/tkc/tkc_locks.c`. Even in this function a straightforward mapping of the old representation to the new one is sufficient. At the same time, a few local variables used to compare byte ranges were changed from type `long` to type `afs_hyper_t`.

4.9.3. CM 64-bit byte range checks

The CM was not very good about checking all 64-bits of token byte ranges in some cases. In `RevokeDataToken()`, defined in `file/cm/cm_tknimp.c`, the comparison of cached chunk offsets was ignoring the high bits of the byte range. A similar problem existed in the slice-and-dice evaluation performed by `cm_TryLockRevoke()` in `file/cm/cm_lockf.c` and the loop over all dcache entries performed by `cm_UpdateDCacheOnLineState()` in `file/cm/cm_dcache.c`. These were fixed by changing local variables to be hypers and using hyper comparison macros throughout.

4.10. Backward Compatibility with Older Systems

The maximum file size of systems that do not provide one is assumed to be 2ˆ31−1. This assumption allows new (64-bit capable) hosts to accomodate most of the limitations of these systems. However, several problems require additional countermeasures. These countermeasures are employed whenever the remote host's maximum file size is equal to 2ˆ31−1 and the host hasn't explicitly said it supports 64-bit offsets by specifying `AFS_CONN_PARAM_SUPPORTS_64BITS` when communicating its maximum file size.

These countermeasures mostly consist of mapping between the value representing the largest possible file offset from 2ˆ63−1 used by new hosts and 2ˆ31−1 which was used by old hosts. This happens in these places:

(a) In `CM_EndPartialTokenGrant()` received tokens coming from old servers have their `endRanges` mapped from 2ˆ31−1 (or any larger value in case of truncation by the server) to 2ˆ63−1.

(b) In `SAFS_GetToken()` requests for tokens from old clients are adjusted so the `endRange` is mapped from 2ˆ31−1 to 2ˆ63−1.

(c) In `px_SetTokenStruct()` tokens being return to an old client have their `endRanges` mapped from 2ˆ63−1 to 2ˆ31−1. If the resulting token has an empty byte range (i.e., `beginRange` was also above 2ˆ31−1), the token is zeroed.

(d) In `fshs_RevokeToken()` the column A and B tokens offered to old clients during a revoke are eliminated if their range started beyond 2ˆ31−1 and have their `endRanges` truncated to 2ˆ31−1. Tokens with empty ranges after mapping are invalidated and the appropriate offered bit is cleared to withdraw the offer.

In addition, OT13445 describes a problem in which the most significant 32 bits of the start and end position in `afsRecordLock` were uninitialized. The two recently defined members `l_start_pos_ext` and `l_end_pos_ext`, were never being set. The changes described above that use the IDL `[represent_as]` mechanism fix this problem. However, older systems still produce lock ranges that may appear to contain garbage to a 64-bit host.

To address this the high 32 bits of these ranges are cleared when they come from old hosts. This should present no operational problem since old clients can not hold locks on files beyond 2ˆ31−1 nor can old servers contain files longer than that. The following functions contain this protection:

(a) In `fshs_RevokeToken()` when processing the output returned from `TKN_TokenRevoke()`.

(b) In `cm_GetTokensRange()` after the call to `AFS_GetToken()`.

(c) In `cm_GetHereToken()` after the call to `AFS_GetToken()`.

(d) In `cm_RecoverSCacheToken()` after the call to `AFS_GetToken()`.

4.11. Miscellaneaous Changes

4.11.1. Printed representation

Unfortunately there is no good way to handle printing hypers in a generic fashion. While platforms that support a 64-bit scalar type have some `printf()` control string to convert them, it was not feasible to parameterize all control strings.

So, as a compromise, we have tried to standardize on `%u,,%u` as the printed representation for hypers. The DFS code base contains a large variety of forms, many of which were converted to this standard form. Printing hypers with this control string requires passing a pair of arguments explicitly to `printf()`. This was simplified somewhat by liberal use of the `DFSH_HGETBOTH()` macro.

The util module now exports two new functions (declared in `<dcedfs/hyper.h>`) to help with string representations.

(a) `char *dfsh_HyperToStr(afs_hyper_t *h, char *s)` — Calls `sprintf()` with `"%u,,%u"` as the control string. Note that it takes the *address* of a hyper for historical reasons. As a convenience it returns its second argument.

(b) `int dfsh_StrToHyper(const char *numString, afs_hyper_t *hyperP, char **cp)` — Takes a string and converts it into a hyper if possible. If it succeeds, the value is returned in `*hyperP`, a pointer to the first unused character in `numString` is returned in `cp`, and the function returns zero. The `cp` argument may be NULL if no output string pointer is desired.

The function is liberal about the input it accepts, for instance, `"-1"`, `"4294967295,,-1"` and `"0xffFFffff,,037777777777"` all produce a hyper containing 64 one bits.

4.11.2. ICL logging

The ICL package has a hyper type, `ICL_TYPE_HYPER`, which takes the address of a hyper and inserts a pair of `u_int32`'s into the log. These integers are passed directly to `printf()` by `dfstrace()`, high half first, so the format string should contain two integer translation directives; typically a hyper is printed as `%u,,%u`. In several cases a pair of `ICL_TYPE_LONG`s were being passed to ICL traces where it made more sense to pass the hyper by reference. No changes were necessary to the print strings.

5. ACKNOWLEDGEMENTS

Thanks to Steve Strange (DEC), Steve Lord (Cray), Carl Burnett (IBM), Craig Everhart (Transarc) and Blake Lewis (Transarc) for very helpful comments both on this document and the code changes it describes.

REFERENCES

[RFC 51.1] S. Strange, ''DFS Source Code Cleanup to Support Both 32-Bit and 64-Bit Architectures'', DCE-RFC 51.1, February 1994.

[RFC 51.2] S. Strange, ''A 32-Bit/64-Bit Interoperability Solution for DFS'', DCE-RFC 51.2, June 1995.

[LFS 96] Large File Summit. "Adding Support for Arbitrary File Sizes to the Single UNIX Specification", 20 March 1996, http://www.sas.com:80/standards/large.file.

AUTHOR'S ADDRESS

Ted Anderson
Transarc Corporation
707 Grant St.
Pittsburgh, PA 15219
USA

Internet email: ota+@transarc.com
Telephone: +1-412-338-4410