N/A

Internet-Draft

Intended status: Informational

Expires: November 27, 2011

Matt Benjamin

Linux Box Corporation

April 27, 2011

AFS Byte-Range Locking

draft-mbenjamin-afs-file-locking-06

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

Copyright Notice

Abstract

The AFS-3 protocol supports file locks, but only on whole files, only in advisory mode. Efficient support for byte-range file locking, together with the stronger semantics with which they are associated, are required to improve the suitability of AFS as a LAN file-sharing protocol for both Unix and Windows clients. Applications on the Windows platform, in particular (e.g., Microsoft Office), actually require byte-range locking to function correctly. Emulation in the client has alleviated most serious problems, albeit, with reduced semantics. We propose protocol enhancements facilitating server-coordinated byte-range locks, atomic lock up/down-grade support, improved semantics for files under byte-range lock control, protocol support for wait-on-lock with fairness, and mandatory lock enforcement for clients on request. The delegation proposal, included within this document in previous drafts, has been split out into a separate proposal, based on feedback from reviewers.

Table of Contents

1 Introduction

While AFS-3 does support file locking, it permits locking of whole-files only, and provides this support inefficiently. AFS clients can take locks on any file object, with the granularity of an entire file, using the RXAFS_SetLock procedure, and release them with the RXAFS_ReleaseLock procedure. AFS uses a poll-based locking model. AFS file locks, once issued, are considered to persist only for 5 minutes, unless extended by the requesting client using the RXAFS_ExtendLock procedure. The OpenAFS file server implementaion, based on the original Transarc AFS file server, tracks locks directly in its on-disk volume structures. The disk package tracks lock type (LockRead or LockWrite), numbers of clients holding locks, and a timestamp. Lock ownership, which in many cases may be reliably inferred, is not recorded. Hence, a broken or malicious client might release locks it never set (i.e., locks set by other clients). The AFS protocol also does not permit atomic lock upgrades (or downgrades).

2 Conventions Used in this Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

3 Byte-Range Locking Interfaces

3.1 Dependencies

The byte-range lock feature depends on support for extended callback notifications and extended host tracking support in client and server.

3.2 Backward Compatibility

AFS clients and servers will indicate their support for byte-range locking through new client and file server capability flags:

```
const CLIENT_CAPABILITY_BYTE_RANGE_LOCK = 0x0008;

const VICED_CAPABILITY_BYTE_RANGE_LOCK = 0x0010;
```

Additionally, a client capability flag is provided to indicate when a client is prepared to accept async lock issue notifications, as a substitute for polling:

```
const VICED_CAPABILITY_ASYNC_LOCK_ISSUE = 0x0020;
```

3.3 Concepts

3.3.1 General

An AFS file server is responsible to coordinate byte-range locking requests and, optionally, enforce mandatory locking semantics relative to file operations, initiated at different clients. By contrast with the traditional AFS file locking protocol, the proposed byte-range locking protocol makes an attempt to associate locks with a unique subject, specifically, a ViceID and unique identifier which could correspond to a unique session or process executing on the client machine.

Clients (cache-manager processes not co-located in memory) request and release byte-range locks through a pair of interfaces (RequestByteRangeLock, ReleaseByteRangeLock) similar to those provided by the traditional AFS locking implementation. Two base lock types (read and write, in general regarded as "shared" or "exclusive") locks, plus a new share reservation lock type, are defined. Additional arguments and flags are provided to permit selection of desired lock ranges, intention to "wait" on the lock (i.e., willing to accept a deferred issue of the lock at such time as the file server can grant the lock, if it cannot be granted immediately), and desired special semantics--currently, the client may request mandatory enforcement. Clients already holding a read or write lock on a range may atomically upgrade or downgrade the lock to the orthogonal type, i.e., they need not release a lock of one

type before requesting the other type, avoiding the race condition present in the traditional AFS locking protocol. Byte-range locks are permanently associated with an owner, the client which requested the lock. A lock may not be released by a client which never owned it.

A file server may revoke locks granted to any client, for any reason. The file server may also request clients to re-assert their interest in outstanding locks, at any time--in particular, if a client holding locks has not been heard from for a long period (e.g., 10 minutes). Provision is made for re-establishment of state after server restarts or other service interruptions.

Administrative users may under various circumstances have need to identify the owner and state of locks on a locked file, and to revoke file locks administratively. This proposal includes RPCs allowing administrative users to perform these operations, and suggests exposure through new AFS pioctls and the fs command.

3.3.2 Lock Management

Lock management in the proposed interface is completely redefined relative to the file locking in AFS-3. Concepts are borrowed from AFS cache management, including the callback concept. A byte-range lock may be regarded as a special-purpose callback. A file server may use the ExtendedCallBack interface to request re-assertion of existing locks or revoke (cancel) locks completely. These indications re-use the existing AFSCB_Event_Cancel extended callback notification, adding new cancellation types defined below.

3.3.3 Share Reservations

To support platforms in which use mandatory locking and other enhanced sharing semantics, in particular, to support Microsoft Windows sharing semantics, a new share reservation mechanism is proposed. AFS-3 share reservations serve a purpose similar to the correspondingly named facility in NFSv4. Share reservations provide a means by which clients can reserve, in advance of any I/O or ordinary locking operations, a specific set of sharing semantics. For example, a client would use a share reservation to request mandatory enforcement semantics, or to request a specific share mode. AFS-3 share reservations are locks acquired and released by clients using the SetByteRangeLock and ReleaseByteRangeLock procedures defined in this document, with special meaning. A share reservation may be taken only at whole-file granularity.

3.3.4 POSIX Conventions

In addition to having (as presently standardized) advisory semantics, the application interfaces for file locking on Unix-like platforms are not entirely uniform (cf. fcntl, flock, lockf) and not uniformly compatible with those on Windows operating systems. In particular, a POSIX file locking

implementation may consolidate adjacent lock ranges taken in different lock requests. In addition, POSIX permits unlocking of potentially non-overlapping locked ranges (including locks of different types) in a range in a single operation and permits splitting of a locked range by unlocking an intervening range. A POSIX client may request a lock spanning any future end-of-file by setting a lock length of 0. None of these behaviors is permitted using Windows file locking interfaces. Consolidation of adjacent locked ranges, in particular, would be unexpected and incorrect behavior for a Windows file locking client. ("Two adjacent regions of a file cannot be locked separately and then unlocked using a single region that spans both locked regions.") The listed behaviors are not visible to other (possibly non-Unix) clients independently operating on the same file, however, and each is bounded by the scope of a specific operation (e.g., SetByteRangeLock or ReleaseByteRangeLock). Hence a per-call flag is sufficient to allow a cache manager to select appropriate semantics for its platform. The present document attempts to provide a uniform interface for a superset of POSIX file locking facilities. For each operation where a choice of operational semantics is available, the client may specify POSIX semantics, defined as supporting the above-listed behaviors for both shared and exclusive locks, using the AFSLock_Flag_Posix flag. The unmarked semantics are those of the corresponding Windows file locking operation.

3.3.5 Deferred Locks

Where possible, locks are granted immediately with the completion of the SetByteRangeLock request. A file server MAY, on explicit request and subject to client capability, agree to prospectively issue a lock to an interested client at a future time, when the requested lock becomes available. Such deferred locks constitute a promise to issue the lock with best-effort consideration of fairness. A new extended callback notification (AFSCB_Event_Flock) is provided to effect asynchronous issue of a deferred lock to a waiting client, for clients configured to accept it. Clients not configured for async issue processing must alternatively poll for desired locks. Deferred locks may themselves be canceled.

3.3.6 Server Restarts

When a byte-range locking capable client receives one of the InitCallBackState RPCs from a byte-range locking capable file server, it must assume that any byte-range locks it held prior to receipt must be re-asserted or bulk-released at the file server, using the server's AssertExtendLocks RPC. A conformant file server may, but need not, be prepared to validate locks previously issued to clients, across server restarts. In future revisions, the Token attribute of AFSByteRangeLock may allow file servers to reliably recognize locks they issued in these circumstances, using cryptographic or other mechanisms.

3.4 Constants

3.4.1 Lock Type

AFS-3 defines the following lock types:

```
%#define LockRead 0

%#define LockWrite 1

%#define LockExtend 2

%#define LockRelease 3
```

The current draft adds the following new lock type:

```
const LockShareReservation = 4;
```

3.4.2 Lock Flags

The following flag constants are defined for use in the Flags member of the AFSByteRangeLock structure and equivalently in the Flags argument of the SetByteRangeLock procedure, with the same semantics:

```
const AFSLock_Flag_Mand = 0x0001;  /* Request mandatory
enforcement */

const AFSLock_Flag_Wait = 0x0002;  /* Request async wait on
lock */

const AFSLock_Flag_Posix = 0x0004; /* Request posix semantics
*/

const AFSLock_Flag_TestLock = 0x0100; /* test for conflicts
only */

const AFSLock_Flag_EReturn = 0x1000; /* error return flag */
```

AFSLock_Flag_Mand

Requests mandatory enforcement when sent with a SetByteRangeLock request or in a deferred AFSByteRangeLock instance. Asserts mandatory enforcement in an AFSByteRangeLock instance.

AFS_Lock_Flag_Wait

Requests deferred lock if immediate lock cannot be granted when sent with a SetByteRangeLock request. Indicates deferred lock in an AFSByteRangeLock instance. The SetByteRangeLock procedure may return locks in this state, subject to client capability and if so requested in the Flags argument.

AFSLock_Flag_TestLock

Requests that the current lock request be evalutated only for conflict information.

AFS_Lock_Flag_EReturn

When set on return from a lock request, coincides with an error return and non-zero members in Lock describe a conflicting lock which was in effect at the time of the request and obstructed it.

3.4.3 Lock Flags for Share Reservation

The following flag constants are defined for use in the Flags member of the AFSByteRangeLock structure and equivalently in the Flags argument of the SetByteRangeLock procedure, and specifcally, identify share reservations:

```
const AFSLock_Flag_Share_Read = 0x0008;       /* Allow Share
mode READ (Share Reservation) */
```

```
const AFSLock_Flag_Share_Write = 0x0010;      /* Allow Share
mode WRITE (Share Reservation) */

const AFSLock_Flag_Share_Exclusive = 0x0020;  /* Assert
EXCLUSIVE sharing (Share Reservation) */
```

AFSLock_Flag_Share_Read

Share reservation. Allow future clients to open this file for reading. Opens for writing mode and requests for write locks will fail unless the AFSLock_Flag_Share_Write flag is set.

AFSLock_Flag_Share_Write

Share reservation. Allow future clients to open this file for writing. Opens for reading mode and requests for read locks will fail unless the AFSLock_Flag_Share_Read flag is set.

AFSLock_Flag_Share_Exclusive

Share Reservation. Requests exclusive access to the file by the requesting process at the requesting client.

3.4.4 Lock Status

The following flag constants are provided to coordinate advanced lock-management operations:

```
const AFSLock_Flag_Extend = 4;  /* request extension, or server
ack extended */

const AFSLock_Flag_Discard = 8; /* discard lock, or server ack
discarded */
```

AFSLock_Flag_Extend

Sent with AssertExtendLocks indicates request to assert/extend the corresponding lock. Returned from AssertExtendLocks in OutStatus array, indicates lock confirmation.

AFSLock_Flag_Discard

Sent with AssertExtendLocks indicates intention to discard the corresponding lock. Returned from AssertExtendLocks in OutStatus array, acknowleges lock discard.

3.4.5 Extended Callback Constants

The following extended callback cancellation types and flags are provided, to facilitate lock management through the ExtendedCallback interface:

```
const AFSCB_Cancel_ExtendLocks = 7; /* re-assert locks, or lose
them */

const AFSCB_Cancel_RevokeLocks = 8; /* locks on Fid revoked */
```

These cancellation types are intended to be sent with notifications of the existing AFSCB_Event_Cancel type.

3.4.6 Extended Callback Extra Flags

AFSCB_Lock_Flag_All

Sent as the value of ExtraFlags when the notification type is AFSCB_Cancel_ExtendLocks or AFSCB_Cancel_RevokeLocks, the notification shall apply to all eligible objects, in which a 0 value has also been set for one or more of Volume, Fid, Uniq in the corresponding callback, with the following intepretation:

- If Volume is non-zero, and is published from the sending file server, while Fid and Uniq are 0, then all outstanding locks on files in the volume are requested to be re-asserted or revoked, depending on the value of the corresponding notification
  - If the notification type is AFSCB_Cancel_ExtendLocks, all corresponding locks are requested to be extended
  - If the notification type is AFSCB_Cancel_RevokeLocks, all corresponding locks are revoked
- If all of Volume, Fid, and Uniq are 0, then all outstanding locks on files published from this server are requested to be re-asserted or revoked, depending on the value of the corresponding notification
  - If the notification type is AFSCB_Cancel_ExtendLocks, all corresponding locks are requested to be extended
  - If the notification type is AFSCB_Cancel_RevokeLocks, all corresponding locks are revoked

AFSCB_Cancel_ExtendLocks

When sent as the reason for cancellation in an ExtendedCallback notification, indicates the server requires re-assertion of all locks on FID using the file server's AssertExtendLocks procedure. The client MUST execute the procedure for all locks it asserts on FID prior to the Expiration in the callback, else it MUST consider any locks it held on FID to be canceled.

AFSCB_Cancel_RevokeLocks

When sent as the reason for cancellation in an ExtendedCallback notification, indicates administrative cancellation of all locks on FID.

```
const AFSCB_Flag_AssertLocks = 4; /* request ExtendLock */

const AFSCB_Flag_RevokeLocks = 8; /* locks cancelled */
```

AFSCB_Flag_ExtendLocks

Has the same meaning and effect as AFSCB_Cancel_ExtendLocks, but may be sent with an arbitrary extended callback message.

AFSCB_Flag_RevokeLocks

Has the same meaning and effect as AFSCB_Cancel_RevokeLocks, but may be sent with an arbitrary extended callback message.

3.5 Data Types

3.5.1 AFSByteRangeLock

The AFSByteRangeLock data type represents a byte-range lock issued by an AFS file server:

```
struct AFSByteRangeLock {

    AFSFid Fid;

    afs_uint32 Type;

    afs_uint32 Owner;

    afs_uint64 Uniq;

    afs_uint32 Flags;

    afs_uint64 Offset;

    afs_uint64 Length;

    afs_uint64 Expiration;

    AFSOpaque Txid;

    AFSOpaque Token;
```

```
};
```

Fid

The Fid on which the lock is held.

Type

The type of lock requested, LockRead or LockWrite. A byte-range read lock is a non-exclusive read assertion on the stated range, which may be shared by any number of readers and no writers. A byte-range lock is an exclusive write assertion on the stated range.

Owner

The ViceID in use by the client requesting the lock.

Uniq

Value uniquely identifying a session or process context at the client. The representation of Uniq is intended to be able to uniquely represent the most relevant process or thread context on modern platforms.

Offset

The distance in bytes from beginning-of-file to the start of the locked range.

Length

Length in bytes of the locked range.

Expiration

AFSByteRangeLock instances may be regarded as a special-purpose callback. Instances persist until canceled, or until Expiration is reached.

Txid

An arbitrary counted bytestring originating at the client with the original request granting a lock. Defined for this revision of the specification to have a maximum length of 0.

Token

An arbitrary counted bytestring originating at the server when the lock is issued. Defined for this revision of the specification to have a maximum length of 0. In future revisions it may be used to store an "irrefutable"

cryptographic object which may be used to re-assert locks after server restart, or similar scenarios.

3.5.2 AFSByteRangeLockSeq

A variable-length array of type AFSByteRangeLock used for bulk calls for asserting and locks.

```
const AFSFLOCKMAX = 512;

typedef AFSByteRangeLock AFSByteRangeLockSeq<AFSFLOCKMAX>;
```

3.5.3 AFSByteRangeLockPointer

A convenience typedef for a pointer to an AFSByteRangeLock object.

3.5.4 AFSByteRangeLockPointerSeq

An array of pointers to objects of type AFSByteRangeLock, used by AssertExtendLocks to return locks with updated expiration time.

```
typedef AFSByteRangeLockPointer
AFSByteRangeLockPointerSeq<AFSFLOCKMAX>;
```

3.5.5 AFSLockHostIdentifierSeq

```
typedef HostIdentifier AFSLockHostIdentifierSeq<AFSFLOCKMAX>;
```

An array of HostIdentifier structures used by the GetByteRangeLockStatus procedure to report client machines holding locks.

3.5.6 AFSCB_NotificationData Redefinition

The AFSCB_NotificationData union defined in the Callback Extended Information draft is redefined (upward compatibly), as the following:

```
const AFSCB_Event_Flock = 14;


ext-union AFSCB_NotificationData switch (afs_uint32 Event_Type)
{


 ...
```

```
 case AFSCB_Event_Flock:

    AFSCB_Data_Flock u_xlock;

 };
```

```
  AFSCB_Data_Flock
```

A struct AFSCB_Data_Flock is defined to be the type of AFSCB_NotificationData at AFSCB_Event_Flock. The structure is equivalent to an AFSByteRangeLock except it omits Fid.

```
struct AFSCB_Data_Flock {

    afs_uint32 Type;

    afs_uint32 Owner;

    afs_uint64 Uniq;

    afs_uint32 Flags;

    afs_uint64 Offset;

    afs_uint64 Length;

    afs_uint64 Expiration;

    AFSOpaque Txid;

    AFSOpaque Token;

};
```

3.6 Procedures

3.6.1 SetByteRangeLock<sub:SetByteRangeLock>

Requests a lock of type Lock.Type on Fid, on the range [Lock.Offset, Lock.Offset+Lock.Length). Lock.Type must be one of LockRead or LockWrite. Lock.Owner shall be set to the ViceID corresponding to the requesting process or equivalent, or to 0 if this is not known. Lock.Uniq shall be set to a value uniquely identifying the requesting process or equivalent. On Unix-like systems, Lock.Uniq could be set to the PID of the requesting process. Lock.Txid shall be a counted bytestring corresponding to the AFSByteRangeLock attribute of the same name. Lock.Txid is defined at this revision to have length 0.

```
proc SetByteRangeLock(

    IN AFSFid *Fid,

    INOUT AFSByteRangeLock *Lock

) = 65601;
```

Notes

On successful return the file server has granted the requested lock, and Lock points to the server's asserted AFSByteRangeLock structure. If the client has requested and the server agrees to issue a deferred lock, Lock points to the server's asserted deferred AFSByteRangeLock structure. The client may safely determine if it has been granted a deferred lock by inspecting the value of Lock->Flags.

The returned Lock structure MAY differ from request with respect to Flags. The returned Lock structure MUST NOT differ from the request with respect to range, unless AFSLock_Flag_EReturn is to be returned (below), or unless POSIX semantics are in effect.

If a SetByteRange lock request would fail due to conflict, or if AFSLock_Flag_Test is set in the request and the request would have failed due to conflict, the server MAY provide conflict information. To do so, the server returns EAGAIN, setting flag AFSLock_Flag_EReturn. Non-zero members in the returned Lock, less Expiration, describe a conflicting lock which was in effect at the time of the request and obstructed it. The returned value for Lock->Expiration indicates the earliest time the client should resend the lock request (see polling, below).

The value of the Flags argument may alter the semantics and/or processing of the call:

- if (Flags & AFSLock_Flag_Mand), file server is requested to provide mandatory locking semantics as defined below--if the file server is willing to provide mandatory enforcement, it MAY set the corresponding flag in Lock, and if so MUST restrict writes on the asserted range to the holding client for the duration of the lock. It is expected that clients will request mandatory enforcement in a share reservation request.

- if (Flags & AFSLock_Flag_Wait), file server is requested to issue a deferred lock if the requested lock may not be immediately granted--the file server MAY grant a deferred lock in response to this request, indicating its agreement by setting the corresponding flag in Lock. Lock is in this

instance an indicator only of the deferred lock promise

- if (Flags & AFSLock_Flag_Posix), POSIX lock semantics for
  byte range locks will be observed for the current request

  Async Lock Issue vs Polling

A file server MUST be prepared to support clients unable to use the async lock issue mechanism. That is, it publishes the VICED_CAPABILITY_BYTE_RANGE_LOCK capabilty, but not VICED_CAPABILITY_ASYNC_LOCK_ISSUE. This situation may arise when a client is operating in an unsecured environment and also lacks a secured callback channel.

A file server MUST distingush clients not eligible for async issue, and for these clients:

1. **send the AFSCB_Event_ReleaseLock extended call back notification to such clients when the client would be** eligible to acquire a lock it has previously requested, under specific conditions

2. **notify clients selectively in the presence of contention, so as to achieve a best-effort degree of fairness**

3. **in consideration of read/write semantics, notify potentially multiple clients with compatible shared lock requests**

To improve the efficiency of polling, clients are provided with a lock expiration hinting mechanism, as follows:

1. **when a client SetByteRangeLock (UpgradeByteRangeLock) cannot be granted, the value for Expiration in the returned Lock** object indicates

   (a) the expiration time of the corresponding deferred lock

      i. if the client had sent AFSLock_Flag_Wait, and a deferred lock was successfully issued by the server, and the client has published the capability to accept AFSCB_Event_Flock (async lock issue) notifications

   (b) the minimum interval which the client MUST delay polling for the requested lock,

      i. if the client as not published the capability to accept AFSCB_Event_Flock (async lock issue) notifications

2. **if a client is unable to to accept async lock issue notifications but expresses willingness to wait for a** SetByteRangeLock (UpgradeByteRangeLock) to complete, the server MUST

(a) send an AFSCB_Event_ReleaseLock notification when it
believes the client would be eligible to receive the lock
if such a condition arises prior to the Expiration it sent
with the corresponding lock request, or

(b) send an AFSCB_Event_Cancel notification for the
corresponding Fid, to inform the client that its pending
notification is cancelled

POSIX Semantics

The following behaviors are specified only when POSIX file lock semantics are in effect:

- If a process has existing locks on a file F and requests a new lock in a range overlapping existing locks and the type of each existing lock is LockRead or LockWrite, the type of the existing lock(s) shall be replaced by the new lock type

- If a process requests a lock adjacent to an existing lock of the same type it already holds, the locks SHOULD be consolidated into a single lock, this will be indicated in the returned structure

- If a process requests a lock with a length of 0, the lock, if granted, extends through any future end-of-file

Share Reservations

A share reservation is a file lock which is logically and operationally distinct from traditional read and write locks, but interact with lock requests of these types according to the following rules, (as noted, share reservations are only issued at whole-file granularity):

- if a client holds an exclusive share reservation on a file F, the following assertions hold for the duration of the reservation:

  - no other client, nor the same client, may be granted a share reservation of any type on F

  - no other client may be granted a byte-range or whole-file lock of any type on F

  - the same client may be granted byte-range or whole-file locks on F, according to ordinary rules

- if a client holds a read share reservation (but no write share reservation) on a file F, we assert:

  - other clients may be granted a read share reservation on F

  - other clients may be granted byte-range or whole-file read
    locks on F, according to ordinary rules

  - no other client may be granted a byte-range or whole-file
    write lock on F

  - no other client may be granted an exclusive share
    reservation on F

  - no other client may be granted a write share reservation on
    F

    * the same client may be granted such a reservation,
      provided its read share reservation is the only existing
      on F

- if a client holds a write share reservation (but no read
  share reservation) on a file F, we assert:

  - other clients may be granted a write share reservation on F

  - other clients may be granted byte-range or whole-file write
    locks on F, according to ordinary rules

  - no other client may be granted a byte-range or whole-file
    read lock on F

  - no other client may be granted a read share reservation on
    F (but the same client may be granted such a reservation)

  - no other client may be granted an exclusive share
    reservation on F

- if a client holds a read and write (AFSLock_Share_Read |
  AFSLock_Share_Write) share reservation on a file F, we assert

  - other clients may be granted read or write or (read |
    write) share reservations on F

  - other clients may be granted read or write locks on F,
    according to ordinary rules

  - no other client may be granted an exclusive share
    reservation on F

- a client which holds a read or write byte-range or whole-file
  lock on F but holds no share reservation on F, may be
  following POSIX semantics (although such a client could also
  have requested a read and write share reservation)

- in such a case, no other client may be granted a share reservation of any type on F

In addition, the AFSLock_Share_Mand flag may be included in a share reservation to request mandatory enforcement of byte-range locks, as described in this document. Clients which prefer mandatory enforcement are expected to take a corresponding share reservation to assert this preference whenever appropriate. It is believed that the above rules permit a correct client implementation to achieve Windows file sharing semantics, by taking/releasing appropriate share reservations when files are opened/closed by applications at the client. As noted, the share reservation may be used by any client implementation.

Error Codes

EACCES

The caller does not have the necessary rights.

EAGAIN (EWOULDBLOCK)

The server is unable to grant the request due to conflicting locks. If a deferred lock was requested, a Flags value of AFSLock_Flag_Wait indicates the deferred lock is granted.

EDEADLK

The server declines to grant the requested lock (or deferred lock) because granting it would cause a deadlock.

EINVAL

An illegal lock type was specified.

ENAVAIL

The server unable to grant the request due to a conflicting share reservation. If a deferred lock was requested, a Flags value of AFSLock_Flag_Wait indicates a deferred lock is granted.

ENOLCK

The server has insufficient resources to grant the lock, or the requesting client or file has too many locks outstanding. (No specific limits are mandated or suggested by this document.)

3.6.2 ReleaseByteRangeLock

Releases the byte-range lock represented in Lock.

```
proc ReleaseByteRangeLock(

  IN AFSByteRangeLock *Lock

) = 65602;
```

  Notes

When an AFS client intends to release a byte-range write lock, it MUST ensure that any changed data in the effected range has been sent to the file server with the appropriate StoreData RPC, and that the RPC completed successfully. This requirement is based on an implied assertion that holding a lock on some region of a file implies, invariantly, an up-to-date view on the locked region.

The value of the Flags argument may alter the semantics and/or processing of the call:

- if (Flags & AFSLock_Flag_Posix), POSIX lock semantics for byte range locks will be observed for the current request

  POSIX Semantics

The following behaviors are specified only when POSIX file lock semantics are in effect:

- an arbitrary number of previously-locked ranges, of type LockRead or LockWrite, may be released with a single ReleaseByteRangeLock request

- if Lock.Length is 0, the released range extends to the current end-of-file

By contrast, when default file locking semantics are in effect, the range is asserted to be held by the calling client with the supplied lock type.

  Error Codes

  EINVAL

The caller does not own the corresponding lock.

3.6.3 UpgradeByteRangeLock

Upgrades the byte-range lock represented in Lock, asserted to be held by the calling client, from its current type (which should be LockRead) to LockWrite. The upgrade is executed atomically (no opportunity exists for another client to set a conflicting lock in the upgraded range while the upgrade is being executed).

On unsuccessful return the file server MAY set flag AFSLock_Flag_EReturn. In this case, non-zero members in Lock describe a conflicting lock which was in effect at the time of the request and obstructed it.

```
proc UpgradeByteRangeLock(

  IN AFSByteRangeLock *Lock,

    afs_uint32 Type

) = 65603;
```

```
  Error Codes

  EINVAL
```

The caller does not own the corresponding lock or it is not of the correct type.

```
  EWOULBLOCK
```

The lock could not be granted due to conflicting locks.

```
  EDEADLK
```

The lock could not be granted because granting it would cause deadlock.

3.6.4 DowngradeByteRangeLock

Downgrades the byte-range lock represented in Lock, asserted to be held by the calling client, from its current type (which should be LockWrite) to LockRead. The downgrade is executed atomically (no opportunity exists for another client to set a conflicting lock in the downgraded range while the downgrade is being executed).

```
proc DowngradeByteRangeLock(

    IN AFSByteRangeLock *Lock,
```

```
    afs_uint32 Type

) = 65604;
```

Notes

When an AFS client intends to downgrade a byte-range write lock, it MUST ensure that any changed data in the effected range has been sent to the file server with the appropriate StoreData RPC, and that the RPC completed successfully. This requirement is based on an implied assertion that holding a lock on some region of a file implies, invariantly, an up-to-date view on the locked region.

Error Codes

EINVAL

The caller does not own the corresponding lock or it is not of the correct type.

3.6.5 AssertExtendLocks

A file server may, at any time, request a client to re-assert its interest in oustanding locks, or revoke those locks altogether. It is expected that clients not heard from for a long period (e.g., 10 minutes) would be requested to re-assert any outstanding locks they hold. To request re-assertion of outstanding locks, the file server may send the client an extended callback notification on the corresponding Fids of type AFSCB_Cancel_ExtendLocks, or it may set the flag AFSCB_Flag_ExtendLocks on a notification of another type it was already intending to send.

On receipt of an AFSCB_Cancel_ExtendLocks or AFSCB_Flag_ExtendLocks notification through the extended callback interface, a client MUST either:

- return any locks it asserts in AssertedLocks_Array, the type of union AFSCB_ResultData for these calls
  - if the server rejects any locks asserted by the client, it will so notify client in a subsequent cancellation message
- set a result of AFSCB_Result_ResponseDeferred, and execute the AssertExtendLocks bulk call before the Expiration in the AFSExtendedCallback structure sent with the callback

Fid is the file for which locks are being extended. Flags contains indication of special semantics (e.g., mandatory enforcement) being asserted, if any. AssertedLocks_Array points to a variable length array of AFSByteRangeLock structures the

client asserts to hold. At the completion of the call, the parallel array OutResult indicates the server's confirmation (or refusal) to extend each asserted lock--a value of (Flags & AFSLock_Flag_Extend_Ok) indicates confirmation.

```
/* Assert locks on Fid, on request */

proc AssertExtendLocks (

     IN afs_uint32 Flags,

        AFSByteRangeLockSeq *AssertedLocks_Array,

     OUT AFSByteRangeLockPointerSeq *ConfirmedLocks_Array,

         AFSLockCodeRSeq *Result_Array

) = 65607;
```

3.6.6 GetByteRangeLockStatus

This is a diagnostic procedure provided to permit system administrators to identify client machines and software running on those clients that are currently holding locks on a file. Fid is the file to report on. The call returns parallel variable-length arrays of locks and their associated hosts. The procedure may only be executed by the AFS super user or members of the system:administrators group.

```
proc GetByteRangeLockStatus(

    IN AFSFid Fid,

    OUT AFSByteRangeLockSeq *AssertedLocks_Array,

        AFSLockHostIdentifierSeq *Clients_Array

) = 65605;
```

Error Codes

EACCES

The caller does not have the necessary rights.

3.6.7 CancelByteRangeLock

The CancelByteRangeLock procedure permits system administrators to revoke active locks that may be obstructing normal operations, perhaps due to a system or network problem. Fid is the file on which to revoke locks. If successful, all locks in range [Offset, Offset+Length) are canceled If a value of 0 is given for Offset and Length the range is taken to span the entire file. The procedure may only be executed by the AFS super user or members of the system:administrators group.

```
proc CancelByteRangeLocks(

    IN AFSFid *Fid,

       afs_uint64 Offset,

       afs_uint64 Length

) = 65606;
```

3.6.8 CreateFileLocked

The CreateFileLocked procedure is to be regarded as if it consisted of of two actions, an initial CreateFile action, and a subsequent SetByteRangeLock action, taken atomically. The CreateFile action is taken first, and if the request succeeds, then the AFSByteRangeLock INOUT parameter (ignoring any supplied value for Expiration, Txid, or Token) is evaluated by the server as a byte-range lock request. The creating client is assured that no other client can be granted a conflicting lock on the file during the execution of the procedure. It is expected that clients will typically request a lock of the LockShareReservation type, and use a valid combination of AFSLock_Share_Exclusive, AFSLock_Share_Read, AFSLock_Share_Write, and AFSLock_Share_Mand flags to specify desired sharing semantics. In particular, the CreateFileLocked procedure provides a way to support Windows share mode opens including atomic open and lock semantics assumed by the Windows CreateFile() function. However, a client may request a lock of any valid type and range.

```
proc CreateFileLocked(

    IN  AFSFid *Fid,

        string Name<AFSNAMEMAX>,

        AFSStoreStatus *InStatus,

    OUT AFSFid *OutFid,

        AFSFetchStatus *OutFidStatus,

        AFSFetchStatus *OutDirStatus,

        AFSCallBack *CallBack,

        AFSVolSync *Sync,
```

```
    INOUT

        AFSByteRangeLock *Lock,

) = 65607;
```

Error Codes

The CreateFileLocked procedure shall return error codes corresponding to those of an equivalent CreateFile request. If the CreateFile is successful, and if Lock->Fid != OutFid, then Lock->Fid.Uniq is an error return for the requested lock operation, and may be any valid return from SetByteRangeLock. Otherwise OutFid is locked and Lock describes the lock.

3.7 Windows File Locking Semantics

Implementation of interoperable locking behavior presents challenges for a distributed file system like AFS, which must support clients on platforms which do not agree precisely on the semantics desirable or possible to enforce.

3.7.1 Byte-Range Locking vs. Byte-Range Lock Emulation

As byte-range locking is effectively required for correct behavior of Windows applications, the OpenAFS for Windows client has been forced to implement a locally-enforced byte-range locking mechanism. In the Windows client today, local byte-range are shadowed by a whole-file lock in AFS. With the introduction of server-coordinated byte-range locking, the Windows client is expected to use server byte-range locks when possible.

3.7.2 Atomic Lock Open

Windows provides applications with the ability to open and lock a file in a single operation. As noted elsewhere in this document, the correct use of share reservations and byte-range (or whole-file) lock facilities at clients permits correct implementation of this behavior. The CreateFileLocked procedure is used by clients seeking to atomically create and lock a file in a single operation.

3.8 Lock Enforcement

Mandatory enforcement of file locks is considered a requirement for Windows interoperation. Lock enforcement on Unix-like platforms generally is advisory. The rules proposed here reflect some consideration and discussion of unique features in AFS, and also compromises made in competing systems intended to support mixed Windows and Unix clients, particularly NFSv4.

3.8.1 Governing Ideas

- Byte-range locks may be taken out on a file under the same circumstances under which a whole file might be taken out in traditional AFS

- The mechanism of lock enforcement is to fail the operation being attempted, a hint shall be sent in the return code of the reason for failure

- An operation which fails due to conflict with an existing lock fails completely

- When mandatory enforcement is in effect, attempts by other than owner to write within a range protected by a byte-range or whole-file lock, are asserted to fail

- When mandatory enforcement is in effect, attempts by other than owner to truncate a file such that the truncation overlaps a range protected by a byte-range or whole-file read or write lock, or by a read or exclusive share reservation, are asserted to fail

- Attempts to write outside any conflicting locked range on a file F with at least one mandatory locked range and not conflicting with any share reservation on F, considering the view of locks on the file at the fileserver when the write request is processed, are considered valid (this is the documented behavior on Windows platforms)

- Since applications exist, particularly for the command line (e.g., tar) which know nothing about locks, and may have legitimate reason to read (though not write) data protected by mandatory locks, relaxed semantics are enforced for reads by clients reading outside any range they have themselves locked--such reads never conflict with lock enforcement, nor with conflicting share reservations. The view of data provided to such a client shall be whatever is available, conforming to regular AFS semantics

- Mandatory enforcement of a read or write lock is asserted to govern only the StoreData operation (by other clients), and not, e.g., the various directory change operations or FetchData[footnote:
Mandatory read lock enforcement is silly, Eisler 2006. More importantly, it causes difficulties for the AFS cache consistency model.
]

3.8.2 Enforcement Rules

- If a client A has a mandatory lock of any type on a range R in a file F, then StoreData operations by any other client B which would alter data in any overlapping range or truncate F such as to reduce or eliminate R, the conflicting operation (initiated by B) fails

3.8.3 Implementation Note

An AFS implementation MAY provide mechanisms, in addition to share reservations, by which administrators or users could specify that files or groups of files in a volume require mandatory enforcement semantics.

4 Security Considerations

Unless explicitly requested by a client, a sever implementation MUST send AFSCB_Event_Flock notifications only over a secured callback channel. By contrast, AFSCB_Event_Release Lock notifications may be sent over any channel.

5 IANA Considerations

This document makes no request of the IANA.

6 Appendix A: XDR Grammar (afsint.xg)

```
const VICED_CAPABILITY_BYTE_RANGE_LOCK = 0x0010;

const VICED_CAPABILITY_ASYNC_LOCK_ISSUE = 0x0020;



const LockShareReservation = 4;



const AFSLock_Flag_None =   0x0000;

const AFSLock_Flag_Mand =   0x0001; /* request mandatory
enforcement */

const AFSLock_Flag_Wait =   0x0002; /* request wait on lock */

const AFSLock_Flag_Posix =  0x0004; /* request posix semantics
*/



const AFSLock_Flag_Share_Read = 0x0008;       /* allow Share
mode READ (Share Reservation) */
```

```
const AFSLock_Flag_Share_Write = 0x0010;      /* allow Share
mode WRITE (Share Reservation) */

const AFSLock_Flag_Share_Exclusive = 0x0020;  /* assert
exclusive sharing (Share Reservation) */



const AFSLock_Flag_Assert_Read = 0x0040; /* assert intention to
READ */

const AFSLock_Flag_Assert_Write = 0x0080; /* assert intention
to WRITE */

const AFSLock_Flag_TestLock = 0x0100;  /* test for conflicts
only */



const AFSLock_Flag_EReturn = 0x1000;          /* error return
flag */



struct AFSByteRangeLock {

    AFSFid Fid;

    afs_uint32 Type;

    afs_uint32 Owner;

    afs_uint64 Uniq;

    afs_uint32 Flags;

    afs_uint64 Offset;

    afs_uint64 Length;

    afs_uint64 Expiration;

    AFSOpaque Txid;

    AFSOpaque Token;

};
```

```
/* Request byte-range file lock */

proc SetByteRangeLock(

    IN AFSFid *Fid,

    INOUT AFSByteRangeLock *Lock

) = 65601;


/* Release byte-range file lock */

proc ReleaseByteRangeLock(

    IN AFSByteRangeLock *Lock

) = 65602;


/* Upgrade byte-range file lock (i.e., from Read to Write) */

proc UpgradeByteRangeLock(

    IN AFSByteRangeLock *Lock,

    afs_uint32 Type

) = 65603;


/* Downgrade byte-range file lock (i.e., from Write to Read) */

proc DowngradeByteRangeLock(

    IN AFSByteRangeLock *Lock,

    afs_uint32 Type

) = 65604;


/* Request lock status report (system:administrators) */

proc GetByteRangeLockStatus(
```

```
    IN Fid,

    OUT AFSByteRangeLockSeq *AssertedLocks_Array,

        AFSLockHostIdentifierSeq *Clients_Array

) = 65605;


/* administratively cancel locks (system:administrators) */

proc CancelByteRangeLocks(

    IN Fid,

       afs_uint64 Offset,

       afs_uint64 Length

) = 65606;


const AFS_LOCK_SEQ_MAX = 10000;

typedef AFSByteRangeLock AFSByteRangeLockSeq
<AFS_LOCK_SEQ_MAX>;

typedef AFSLockFlagsSeq <AFS_LOCK_SEQ_MAX>;


const AFSLock_Flag_Extend = 4; /* client request extend, server
ack extended */

const AFSLock_Flag_Discard = 8; /* client request disard,
server ack discarded */


/* Assert locks on Fid, on request */

proc AssertExtendLocks (

     IN afs_uint32 Flags,

        AFSByteRangeLockSeq *AssertedLocks_Array,
```

```
     OUT AFSByteRangeLockPointerSeq *ConfirmedLocks_Array,

         AFSLockCodeRSeq *Result_Array

) = 65607;
```

7 Appendix A: XDR Grammar (afscbint.xg)

```
const CLIENT_CAPABILITY_BYTE_RANGE_LOCK = 0x0008;


const AFSCB_Result_ResponseDeferred = 2;

const AFSCB_Result_ReturnLocks = 3;


/* Byte-Range Locking Cancellation Types */

const AFSCB_Cancel_ExtendLocks = 7; /* re-assert locks, or lose
them */

const AFSCB_Cancel_RevokeLocks = 8; /* locks on Fid revoked */


/* Cancellation Flags */

const AFSCB_Flag_AssertLocks = 4; /* request ExtendLock */

const AFSCB_Flag_RevokeLocks = 8; /* locks cancelled, sorry */


const AFSCB_Flock_IssueLocks = 1;  /* locks issued on Fid */

const AFSCB_Flock_RevokeLocks = 2; /* locks on Fid revoked */

const AFSCB_Flock_ExpireLocks = 3; /* locks expired */


/* confirm issue of deferred lock requests */

proc AssertExtendLocks (

    IN afs_uint32 Flags,
```

```
       AFSByteRangeLockSeq *AssertedLocks_Array,
    OUT AFSByteRangeLockPointerSeq *ConfirmedLocks_Array,
       AFSLockCodeRSeq *Result_Array
) = 65607;
```

Author's Address

Matt Benjamin

Linux Box Corporation

206 **S. Fifth Ave, Ste 150**

Ann Arbor, MI 48104 USA

Phone: +1 734 761 4689

Email: matt@linuxbox.com