Lustre Windows Client High Level Design

Sun Microsystems

April 8, 2008

Developed by:

OSR Open Systems Resources, Inc.

105 Route 101A, Suite 19

Amherst, New Hampshire 03031-2277

(603) 595-6500 ♦ (603) 595-6503

Revision History

Revision	**Date**	**Author**	**Description**
V0.1	1 March, 2008	WAM	Initial Internal Draft
V0.2	8 March 2008	WAM	Internal Draft Review
V0.8	10 March 2008	WAM	Draft Version for Final Internal Review
V0.9	11 March 2008	WAM	Draft Version to Sun
V0.91	25 March 2008	WAM	Update with Sun Feedback
V1.0	8 April 2008	WAM	Update with Sun Feedback

Documents used in the preparation of this specification

- Untitled Document “Non-compatible issues between gcc and VC”
- *Patchfree Lustre client Tiger Porting DLD*, dated April 20, 2006.
- *Patch free Lustre client in tiger*, Dated February 7, 2006.
- *Windows Lustre Client File System Design*, Version 0.4 dated July 2005.
- *Design and Implementation of XNU port of Lustre Client File System*, dated February 1, 2005.
- Miscellaneous Windows Driver Kit documentation and help files

Table Of Contents

1 Introduction

The primary purpose in creating this document is to create a basic working model for constructing a native Lustre Client for the Windows OS environment. While it will describe some of the key issues in that development, the primary focus here is to establish the conceptual framework for this work in order to foster clear understanding and discussions between the Lustre development team and the Windows Client design/implementation team. This communications can be complicated because the terminology employed in discussing similar concepts is often different.

The goal in creating this document is to provide the basic conceptual framework that will be used to guide the Windows Lustre Client implementation project. Key requirements for this project include:

- A native Windows Lustre Client that will appear to normal Windows applications as if it were a "typical" local file system (e.g., behaviors similar to those of FAT or NTFS file systems on Windows.)
- Support for Windows Server 2008 and Windows Vista, both 32 bit and 64 bit (x64) versions.
- Support for exporting the Lustre file system to other systems via the CIFS and NFS native Microsoft implementations.
- Emphasis on optimizing performance of the overall system. In the actual design, the emphasis is on making trade-offs that will provide generally better performance. We anticipate that actual implementation will require actual performance optimization (ergo, performance studies followed by detailed optimization) but the goal here is to avoid major architectural changes in performing that optimization.
- Moderate initial development time. An important goal for the project is to be able to deliver a Windows Lustre Client implementation in a reasonable time frame. Thus, an important goal for the design is to attempt to consider the final version, but also to look for intermediate milestones that can be used to provide earlier versions with restricted functionality and/or performance guarantees.

The design as set forth in this document is one that OSR believes will provide a solid base upon which to implement the Lustre for Windows Client. There are a number of variables that make this project more challenging than a "port" or other typical project and as such we expect that this design and the resulting implementation will likely change over its actual lifetime to accommodate the needs of the actual project. However, this document should serve as a basis for discussion for both concepts and issues.

Current Status: At the present time, this document is a draft document. The purpose of this document is to provide a basis of ongoing discussion with Sun Microsystems about the final design of the resulting product. It is provided to Sun for comment and feedback.

The goal here then is to ensure that all parties have reached a basic understanding of the proposed system; as a high level design document the goal of this document is to provide a basic road map for the implementation team. It is **not** a goal of this document to constitute a detailed design document – thus, it does not contain pseudo code, data structure descriptions or in-depth discussions of implementation level issues. Discussions of specific items contrary to this goal are included simply to address concerns raised by Sun during the review of this document.

Except as necessary to provide a conceptual framework, we have not tried to reiterate the contents of the other Lustre documents that were provided to OSR.

2 Architectural Model

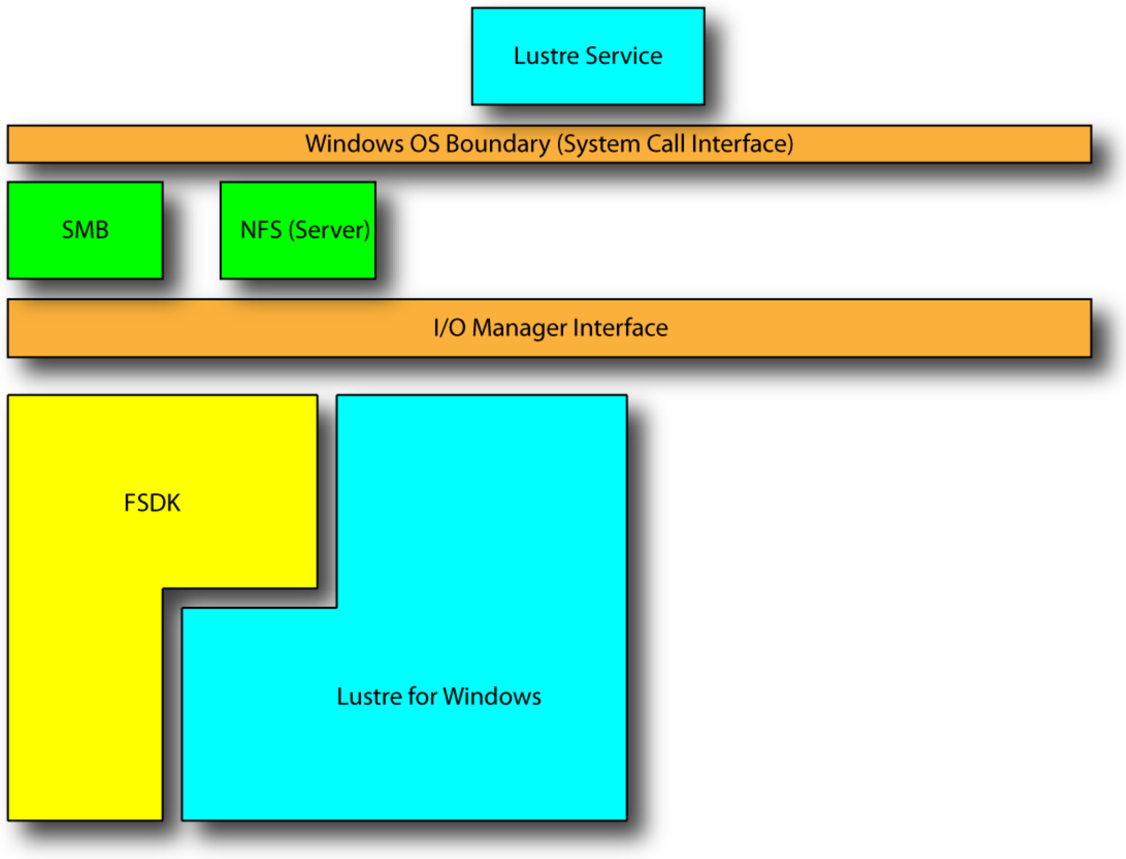
Lustre was originally designed and developed to be a highly scalable cluster file system in which information is maintained in a distributed fashion across multiple machines. In this way, the logical structure of the name space is maintained independently from the physical storage of the information. Further, this technique allows Lustre to "mix and match" systems to allow optimization of specific types of storage based upon the needs of the cluster in which it is being used.
Within Lustre, there are a number of important components:

- **Meta Data Server** (MDS) – these matinain Lustre meta-data information, including the structure of the name space, security information and actual data location information.
- **Object Storage Servers** (OSS) – these are responsible for managing the actual storage and retrieval of data. Typically this would be a native file system for the target storage device (e.g., NTFS.)
- **Object Storage Target** (OST) – these are responsible for managing the storage of file level data across one or more OSS. Thus (for example) a single logical file might be implemented by storing data on multiple distinct servers.
- **Client** – this is the component that allows access from the native system (in our case Windows) to the Lustre servers.

Since our goal is to construct a native Windows client for Lustre, we focus on the behavior of the system from the client perspective. Thus, when an application is running on a system in which the Windows Lustre Client is deployed, it will see an additional name space (likely an available drive letter, but this could also be displayed as mount point, or logical name, inside an existing file system – from the Client perspective these are indistinguishable.)

The Windows Lustre Client will interact with the MDS components to display the name space to applications. While the Windows Lustre Client may cache temporary namespace information, the actual namespace is maintained by the various MDS components. Actual file activity is managed by the OSTs and OSS components and the Windows Lustre Client must interact with them to perform actual I/O operation. We note that, particularly in the case of Windows, I/O activity to a file is relatively rare when compared to actual directory enumeration and basic information gathering activities.

The basic model that we are proposing for the Windows Lustre Client would utilize the OSR File Systems Development Kit (FSDK) along with a custom developed File Systems Driver and Windows Service. This basic architecture will allow a Windows native implementation to be developed in a reasonable timeframe and yet provide a solid base for further improving performance and balancing implementation via the service (which can often be easier) and the kernel driver (which is typically higher performance, but certainly far more demanding of correctness.)



The function of the service is to perform potentially complex operations for which there is an existing portable implementation (and for which performance is not a primary issue.) Operations we would normally consider to be an important part of this layer would include:

- Mapping Windows security credentials to Lustre security credentials
- Establishing connections with the Lustre services (the handles can be presented to the kernel component for further use, for example. The kernel service must properly convert them to file objects for further use.)
- Interactions with Lustre services. One advantage of this implementation model is that it allows leveraging the existing Lustre libraries in user mode (which has portability tools and libraries such as cygwin that would not apply to the kernel mode development environment.)
- Standard user/kernel communications using an inverted IOCTL service model (e.g., the service calls the driver, the driver suspends the I/O operation until needed.) This would include defining and handling the usual error conditions that can arise in this scenario: no threads available to service a kernel request, no kernel requests waiting for threads, allocation failures, timeouts (generally, the response time should be bounded. That bound can be configuration specified, but

without this the system will "hang" and this typically frustrates and annoys users,) and service terminations.

- Shared memory management; because it is quite likely that some shared data structures will be managed between the kernel driver and user mode service, it will be important to define how that is to be achieved. Because this creates a potential vulnerability in the OS, the kernel side component must be carefully written to validate all data access and handle potential error conditions carefully.[1]

Note that the interface between the kernel component and user components would be private. One important reason for this is that as the Windows Lustre Client implementation evolves, we would expect critical services to be moved into the kernel because of their need for performance.

With respect to Windows and performance in general, the two key areas are I/O performance and directory enumeration. The FSDK library will provide a certain amount of caching for the directory enumeration and supports an invalidation interface so that the cached contents of a directory can be purged as necessary. One disadvantage of this is that purging is for the entire cache (there is no selective update, for example.) Thus, in a high latency environment, it is likely to be useful for a secondary cache. This cache can be maintained in the service (where, presumably it may be able to process incremental updates to information in the name space, or at a minimum initiate a refresh of the directory while it is being actively used) and then shared with the kernel driver (presumably via the shared memory interface.)

[1] The important issue here is that whatever this shared format, it cannot safely contain data pointers – it can contain structures that are self defining but data points are by their nature not safe. Thus, for example, if directory contents are shared between the two components, the fields of the structure (e.g., length) should be capture and then used, rather than use in-place to avoid the risk of them changing during the operation. In addition, we strongly suggest that the interface between these two components be tightly restricted, such that the ACL on the device object only allows the distinguished account used to run the service be granted access. While this does not guarantee no compromise, it does create a barrier to such compromise being trivially achieved.

3 Windows Lustre Client Design

In approaching the Lustre design, we have decided to approach this from the perspective of key FSDK operations that would be implemented as part of this project. While this is certainly not the only possible means of analyzing this, we have taken this approach to attempt to make "bridging the gap" between the FSDK environment (and Windows) and the Lustre implementations on other platforms easier to understand and to map.

In doing this, we have considered the following key FSDK operations:

- FS_LOOKUP
- FS_RELEASE
- FS_CREATE
- FS_ACCESS
- FS_GET_ATTRIBUTES
- FS_SET_LENGTH
- FS_READ
- FS_WRITE
- FS_READ_DIRECTORY3
- FS_UPDATE_SHARE_ACCESS
- FS_REMOVE_SHARE_ACCESS
- FS_GET_NAME2

We note that this list of functions is not exhaustive and the final implementation would be expected to implement several other functions as well. However, our goal in picking out these functions was to explore some of the basic issues that we expect to observe in their implementation.

3.1 FS_LOOKUP

The purpose of the FS_LOOKUP operation is to take an existing, known handle and a new name component and determine if, based upon this information, a new object can be identified. Traditionally, this would imply that a file is being "looked up" in a directory. From the perspective of implementation, we would expect this to behave similar to existing lookup implementations for Lustre, as the logical model for implementation here is comparable (e.g., we would use something similar to the **ll_lookup_it** function that is in the Lustre lite implementation.)

In addition to the basic lookup, we would suggest that using the full path lookup scheme (a variant of FS_LOOKUP) also be employed. This would allow using something akin to the directory cache (dcache) for optimizing lookups of this type. While the FSDK does maintain a single entry cache (the last item looked up on this system,) it is not intended to replace a more scalable cache such as a traditional dcache scheme.

An open question for implementation would be how much of the interaction with the Lustre server can be done directly within the kernel driver component; from an implementation standpoint this can be split between the kernel driver and user mode service as necessary.

The FSDK model then allows the file system implementation to return a *handle* to the FSDK for subsequent operations. What we would suggest is using an extensible table scheme (e.g., similar to the object handle table used in Windows for generating handles) with some sort of arbitrary granularity. The low order bits inside that granularity can then be used to disambiguate handle reuse cases. While not necessary, it is helpful if these really are handles and not pointers to memory blocks. In addition, using these handles allows for easy communications between the kernel mode driver and user mode service, since they will also want to use a handle based scheme rather than an address based scheme.

3.2 FS_RELEASE

Handles in the FSDK environment are **not** reference counted by the FSD (Windows Lustre Client.) The FSDK does maintain reference counts and calls this function when a handle is no longer needed by the FSDK. From an implementation perspective, this normally means that the handle in question can be reused; with the presence of a dcache, it might be useful to maintain a reference to this handle (to avoid reconstructing state) as necessary.

Implementation caution: because the lookup versus release process can be done in a deserialized fashion, it is important that any calls *from* the FSD to the FSDK be properly serialized by the FSD itself (ergo, if the FSD performs an OwPurgeCache call, the FSDK may actually release the handle during that upcall. The FSD needs to be able to handle this case, as it is not protected against by the FSDK.)

3.3 FS_CREATE

The create operation within the FSDK actually corresponds to the creation of a new object – typically a file or directory. From an implementation standpoint it would be similar to the Lustre lite function ll_create_it.

3.4 FS_ACCESS

This function is used to perform a security check on the object in question. We would anticipate that this would be implemented by mapping the current entity (Windows SID based) into its Lustre analog. The access can then be verified (typically by actually opening the object in question.) We note that this is *not* the same as sharing control, but is rather the security check. It is also responsible for enforcing any attributes of the file, including the read-only attribute.

Note that this operation is always performed in the context of the originating thread. Thus, the security check can be done against the current thread security credentials. A trivial implementation (which may be sufficient for Lustre, in fact) would be to only process the read-only bit and defer the access check to opening the relevant object.

3.5 FS_GET_ATTRIBUTES

Given a handle to an already opened object, this retrieves the attributes of that object. This includes the various sizes, timestamps, attributes and link count of the target object. The information provided here is used by the FSDK to respond to a plethora of different operations and is normally cached (e.g., it is important that this be invalidated if the data associated with the file is invalidated.)

We would expect this to be equivalent to the normal **Getattr** implementation in the existing Lustre client codebase.

3.6 FS_SET_LENGTH

The Windows VM system provides a guarantee that paging write operations will **never** extend the size of the file. To honor this, they will always set the length of the file prior to the paging I/O (as well as protect against truncation during their paging I/O operation.) The purpose of this function is for the FSDK to ensure that the length of the file is sufficient for the incoming paging I/O operation.

For the Windows Lustre Client we would expect this to be an internal only value. It does need to be reported back properly, but need not actually represent the file size on the remote server. Thus, this would normally be a modest implementation.

3.7 FS_READ and FS_WRITE

We would expect that these functions will likely involve considerable work throughout the course of the project because they are typically the “performance sensitive” operations within the file. The implementation here will need to consider that as much work as possible should be done at most once (e.g., you can defer opening remote objects up to this point, but should only do so once, not on each I/O operation.) I/O sizes are typically modest in pre-Vista systems, with the maximum being 64KB for paging I/O. User applications can submit larger I/O operations, with a net effective limit of around 32MB (the maximum size that can be described in a single Memory Descriptor List or MDL – a form of page level scatter/gather data structure.)

One issue that may require further thought is that Windows does perform its own read-ahead. We notice that the Lustre lite implementation explicitly contains comments that suggest they disable this and perform their own read-ahead. This is possible with the FSDK, but doing a separate read-ahead cache may not make as much sense in Windows because of the heavy use of the VM integrated file data cache (and the normal “no share” semantics on files.) Read ahead could be disabled on shared access files (which likely don’t benefit from it anyway.)

We would suggest that the implementation here would likely look similar to the **ll_readpage** and **ll_writepage** functions.

3.8 FS_READ_DIRECTORY3

Directory enumeration is a heavily used function in Windows. While the basic model for directory enumeration is similar between Windows and UNIX systems (e.g., an iterative model) the Windows model includes attribute information (timestamps, sizes) in the directory enumeration. These values (notably sizes) need to be correct because applications actually rely upon them for proper behavior.

Open Question: How best to optimize this in the Lustre environment. If we must open and query the size for each file in the directory, we can do so, but typically this becomes a rather expensive exercise. Either a bulk stat or an augmented directory query operation would be critical to optimal performance of this very common operation in a directory being browsed from a Windows system using the Windows Lustre Client.[2]

3.9 FS_UPDATE_SHARE_ACCESS and FS_REMOVE_SHARE_ACCESS

Share access for a Windows system is the concept of shared mode read, write or delete access to the file. By default, sharing is disabled but may be optionally allowed when an application opens the file. Opens are only allowed if the share access is compatible with existing access on the file.

For the initial implementation, we would suggest that the locking be implemented in a “simple” fashion so that the file is locked on the Windows node. In that case, the Windows Lustre Client need not implement these functions. However, once shared file access is added, these two functions will need to be modified to

[2] The Windows Lustre Client will be presenting itself as if it were a local file system so that it can be re-exported via CIFS. Thus, the expectations of behavior will be those of FAT or NTFS.

work with the Lustre servers to coordinate the access of local processes using the file and the actual server controlling such access.[3]

3.10 FS_GET_NAME2

A Windows file may be opened using different variations of its name including:

- Short name (8.3 MS-DOS compatible)
- Long name (Win32 compatible, including POSIX compatible)
- File ID (equivalent of the "inode number" concept in a typical UNIX file system)
- Object ID (an application assigned GUID that can be used to open the object.)

In the initial implementation of the Windows Lustre client, we are proposing that it only implement long file names. In that case, there is no requirement for this API because there is no ambiguity with respect to the name.

We note however that to support the NFS implementation (in Services for UNIX and now natively included in the Windows Server 2008 distribution) it is necessary to support open by file ID. If this is the case, this function will be necessary in order to provide name information about the corresponding file (and its path.)

[3] It is not clear to us at this point which server would fill this role.

4 Potential Issue Discussion

The purpose of this section is to discuss specific issues that we envision affecting the Windows Lustre Client implementation. While we do not expect this list to be exhaustive, our goal is to identify key issues that we have considered; ideally, this list should be updated and refined throughout the course of the project, but at a minimum questions and issues raised during the draft design discussion and review will be added and addressed in this section (and thus captured as part of the final design.)

4.1 Security

An important consideration in any cross-system environment is the presentation of security information; this applies both to connections from the Windows client to the Lustre server as well as through the Windows APIs intended for managing and presenting such security information and attributes.
First, we note that the existing Lustre model already defines that security is the responsibility of the *servers* and not the client. Clients remain untrusted in this model.
Our expectation is that existing credential information (e.g., the SID of the entity issuing the operation) will be used to construct Lustre level credentials that can be used to authenticate the specific user with the Lustre server. An open issue (to be determined) is to define how this mapping is achieved.
Note: security at Windows kernel level is done via the Security Identifier. The expectation would be that this SID would be mapped to some equivalent Lustre structure as administratively defined. Presumably (using Kerberos) this would be achieved by interacting with the key distribution center to obtain a session ticket that is then used to communicate with the Lustre service. Our assumption has been that this is either well understood or will be defined by the existing Lustre implementation and that the code to implement this would be achieved via a user mode component. Kernel interactions would thus consist of converting the SID to the Lustre level credentials needed to establish the communications between the two endpoints.
The actual Lustre security controls (“ACL”) would be modified using some external tool – we do not recommend attempting to map a foreign ACL format into the Windows ACL format. However, should that be a goal, we would suggest that this be implemented as part of the user mode helper service and thus the ACLs can be presented to the FSDK as needed.

4.2 Windows File System Semantics

The semantics of file systems in Windows differs somewhat from those in a typical UNIX file system. We note that they do not differ dramatically – both support similar concepts of files, directories and I/O to and from a “byte stream” but they do differ in a number of subtle and important ways. This includes:

- Windows file systems traditionally are “case preserving, case insensitive.” However, file systems are allowed (and some do support) case sensitive naming semantics. This situation is complicated by a change that Microsoft introduced in Windows XP in which the default is that all file access is done using case **insensitive** behavior, regardless of what is requested by the application. This behavior is implemented in the Windows Object Manager (not the file system driver) and is controlled via a registry parameter. Installing certain optional products (e..g, Services for Unix) changes this behavior and enables case sensitive behavior.[4] Lustre comes from a heritage of case sensitivity. In the past when porting UNIX file systems to Windows we have used the case preserving semantics; any issues with respect to files that differ only in case are not unique to the ported file system and already exist with NTFS (for example.) Thus, preserving existing behavior is generally the best possible solution in this circumstance.

- Windows file systems traditionally support both a “long” and “short” file name. This is not required of file systems, but is provided to handle application compatibility issues – notably applications that only understand short names as well as applications that do not handle some characters (notably spaces) in the names of files and directories. However, the long term trend in Windows is to drop support for short file names (it is a considerable boost to performance for NTFS when short file name support is withdrawn, for example.) Thus, our suggestion here is to *not* support short file names in the Windows Lustre Client.
- UNIX directories typically do not maintain file stat information within them while NTFS and FAT do maintain such information. Directory enumeration operations in Windows thus return the name(s) and attributes of the files as a matter of course. This can be masked by iteratively opening each file and obtaining the attributes of the file but is typically expensive to perform. In such a case, this information should be cached locally whenever possible. In the alternative, a mechanism for obtaining this information efficiently from the MDS would be highly desirable.)
- Windows file systems do not traditionally support “symbolic links”, while UNIX file systems do support them. NTFS does support reparse points (beginning in Windows 2000) and symbolic links (beginning in Windows Vista) using what is known as a reparse point. At the present time, the FSDK does not provide explicit support for symbolic links (there are placeholders in the API for them, but we have never had call to support them.) One option here is that OSR can extend the FSDK to support symbolic links by mimicking the NTFS behavior (e.g., treat them like a specific form of reparse point.) In that fashion, reparse point queries and sets would be translated into symbolic link query/set operations, but only for those operations that match symbolic links. As this is not likely core to the functionality, this could be deferred from an initial release.

[4] Ostensibly, this was due to a security concern that arises when applications default to case insensitive behavior. In such asituation, the file the application wants to access can be “hidden” by creating a file that differs only in case and occurs lexographically prior to the file that the application actually wishes to access.

There are a number of other specialized Windows features (e.g., ACLs and Extended Attributes) that are either described elsewhere or are considered to be optional behavior for a Windows file system. We do not view them as being of consequence (except ACLs, which are discussed elsewhere in this document.)

4.3 FSDK Extensions

We note that the FSDK does not currently implement a handful of features that might be useful for the Windows Lustre Client. As such, these could be added to the FSDK as part of this project (or a logical extension of this project) and would then become part of the supported FSDK code base (ergo, this proposal is not for creating a customized FSDK version.) Specifically:

- Symbolic links. Windows Vista is the first version of Windows to include symbolic link support in a base file system (NTFS.) As such, the FSDK does not implement symbolic links internally, nor does it present them to the underlying FSD. The simplest model here would be for the FSDK to be modified to support reparse points of the relevant type (e.g., matching the behavior of the NTFS file system) so that applications that understand symbolic links would be able to properly interact with them.

At the present time we do not see any other features that might be useful for the Windows Lustre Client that are not supported in the FSDK. Note that we have not included some features that are specific to NTFS but for which there is no analog in Lustre (e.g., transaction support.)

4.4 Locking

Lustre has a complex locking model in keeping with its goal of broad scalability. However, to keep the Windows Lustre Client implementation simple, we would strongly suggest that the initial implementation of this “over compensate” for locking on files. While this might risk disallowing file sharing in cases in which it would otherwise work, it is the safest and most expedient approach to the initial implementation. As we gain further experience with the Windows Lustre Client, it would make sense to carefully move back from this implementation model.

Typically, a revocation would be a purge of the cached FSDK data. Normally this is done using the function OwPurgeCache. However, OwPurgeCache is an unsafe callback (we do not know the locking context) it will frequently fail to perform the operation immediately. In such cases, the FSDK queues a work item to a different thread in which it is now safe for the operation to block, acquire FSDK locks, and complete the operation. In that case the file will normally be released shortly thereafter. Thus, it may be desirable to note in the handle context that there is an outstanding revocation so that it can be satisfied once the release has been received. In that case, the revocation can then be satisfied and the handle discarded immediately.

This is certainly an important area in which we would expect to work with the Lustre team to better understand the proper implementation, as clearly the lock semantics in Lustre are very specific to the Lustre implementation and they need to be properly implemented into the Windows implementation.

4.5 Virtual Memory Differences

One important distinction between the Windows Virtual Memory implementations and the UNIX Virtual Memory implementations (including Linux) is that Windows does not maintain a list of virtual mappings to a physical page – there is no “inverted page table” from which all reference to a physical page can be located and invalidated. As such, the invalidation model that Lustre typically uses will not work in the Windows environment. This is a design philosophy decision on the part of Microsoft and we have previously discussed this with Landy Wang (the person at Microsoft responsible for the Windows Virtual Memory system)

Effectively, what this means is that when references to a file cannot be purged, the locks protecting that file cannot be released. The FSDK provides functions that will attempt to force any outstanding memory references on the given file to be deleted. Note that these can (and do) fail for a variety of reasons *including* the use of those files for memory mapped access by applications. Memory mapped access by the cache maanger (the Windows file system data cache) will certainly cause such purge attempts to fail, but in addition there are other situations in which it is not possible to immediately attempt purging the cache (this relates to issues involving observing lock hierarchy in order to avoid deadlock.) In such cases, the request to purge is often posted and thus may occur shortly after the initial request. To handle this situation, the Windows Lustre Client will need to request a purge on the file. Typically, this will lead to a subsequent release of the file indicating that it is no longer in use. Absent that, the Windows Lustre Client will not be able to release locks that it is holding against the file.

4.6 Third Party Product Interactions

A significant area of concern with respect to any Windows file systems development projects are related to third party product interactions. This includes such common applications as anti-virus products, data replication products, etc. While some of these will not be applicable in the Lustre environment (e.g., data replication products,) other products are very likely to be applicable.

Note that there is no “generic” mechanism for achieving this, and that even testing against a single product version is no guarantee that a new release of the same product will not exhibit some sort of problem or issue. Frequently, the types of issues that arise are not the “fault” of one component, either, but are often very scenario specific and require explicit expertise in analyzing and developing a mutually acceptable work-around.

Our suggestion here is that Sun should consider sending a development team, along with the Windows Lustre Client and any supporting infrastructure necessary for testing, to the Windows Filesystem Plugfests that are held roughly twice a year by Microsoft. Note that this is a development/engineering activity and as such is most productive when developers familiar with the code base under test are present.

5 Feedback/Discussion

This section is used to capture discussions based upon the feedback and discussion from Sun. It is an integral part of the document, but is captured here (separately) in order to ensure that they are addressed and that there is an opportunity to discuss them before they are actively integrated into the design itself.

5.1 Performance Goals

Sun indicates a hard performance goal of 250MB/s for simultaneous read and write from separate threads. We have no specific objection to the goal, but we note that achieving performance at this level is clearly going to require a configuration that can support the same (after all, that is a data rate of 3Gb/s, which would suggest that this is not achievable over a 1Gb/s Ethernet.) Thus, achieving a performance goal at this level will require further feedback from Sun as to the configuration to be tested (not to mention how to configure a Lustre environment that would meet this performance goal.)
We do note that achieving performance at this level will likely require performance study/analysis on the performance platform and under the relevant load platform.

5.2 User Mode Service Usage

Our model of user mode service usage was certainly not intended to indicate it would play any role in the I/O path. Indeed, the comments back (that it should only be part of authentication) is a perfectly reasonable dividing line. What we did not want to do in the design was over-constrain the implementation team as it is often necessary to balance out implementation between these two components for expedient implementation.
In addition, another reason to use a user mode service is to provide a secure communications channel with the kernel service. In that way, applications can speak to the server, it can perform the authentication and then send the relevant IOCTL. In addition, this model gives “free” distributed/remote administration (this is the standard Windows model.)

5.3 Management Tools

We would expect these to be implemented in the user mode service; administrative applications can then call an RPC interface in the UM service to implement the actual calls. This uses existing Windows security and will allow utilities to work either via a command line model or via other Windows like mechanisms (e.g., MMC and shell extensions.)

5.4 Existing Internal API Usage

We also realize that there is an extensive body of code; it was beyond the scope of a high level design to attempt to categorize them. We invite further conversation in this area, but consider this to be essential as part of the implementation effort.

5.5 Cache Invalidation

The FSDK caches attribute information via a write-through model. Thus, the FSD implementation (Windows Lustre Client) will always know file attributes (for example) and thus will not require any FSDK interaction. The *only* thing that is write-back cached is data, and that caching is done by the Windows OS. Thus, the design discusses that situation.

Other information may be cached by the FSDK, but the same API is used to invalidate that cache (including directories and files.)

5.6 Security/ACL Support

We believe that this mapping should be managed by the user mode service, but we can discuss this further with the Lustre team – precisely where the mapping is done should not materially impact the actual design.

5.7 Sharing Modes

The feedback mentions this issue, but we were unable to extract any substantive comments from the document.

5.8 Short Names

We did not expect the Windows Lustre Client would support short names; it was mentioned because it may create issues in the future.

5.9 Directory Query Operations

The FSDK allows the FSD to return a single query format. The FSDK then handles converting that into the format requested by the user. In general, we suggest that any implementation support the most general format for this information (e.g., more recent versions of Windows that use the file ID information format.) This is not required. Our concern was that this information needs to be inexpensive to obtain from Lustre because Windows queries directory information on a regular basis and applications rely upon it being correct.

5.10 Links

We would propose not supporting cross-volume links in the initial version.

The FSDK already detects when two links point to the same file and handles that as appropriate internally via the handle value that is returned to the FSDK.

5.11 Byte Range Locking

IRP_MJ_CREATE is available via an FSDK call (OwGetTopLevelIrp) although we generally try to avoid using that mechanism (the information is passed in via the various calls.)

5.12 Oplocks

We cannot envison any scenario in which oplocks would conflict with Lustre locks. We can, however, disable oplocks if it proves to be problematic.

5.13 Quota Support

We assume this implies an orthogonal implementation of quotas and not supporting the Windows quota model.

5.14 NLS/Codepage Support

We assume that mappings for these are otherwise defined. Names would then be mapped via library calls that are responsible for mapping between the two.

5.15 Read-ahead

We are certainly in agreement that read-ahead will likely be essential to good performance. Optimizing this is likely to involve modifications to the FSDK as well as performance tuning and analysis.

5.16 Directory Change Notification

In the FSDK model, directory change notification can be achieved by invalidating the entire directory contents or (as I recall) by simply invalidating the specific file that changed. However, it sounds like the whole directory invalidation model is consistent with what you have done in the past for Linux.

5.17 FS_LOOKUP_PATH

To facilitate the behavior of file systems that can more efficiently process path-level lookup operations, the FSDK provides a "lookup path" optimization. In this optimization, the entire path is provided to the FSD (in this case the Windows Lustre Client) and the FSD in turn returns a handle to use for both the parent directory and the child file (there are some special circumstances here as well, for example if the parent directory exists but the child file does not, where the parent directory handle is returned.) In its current implementation, the FSDK will then perform a second FS_LOOKUP call on the parent directory with the child file name in order to confirm (once it holds the correct FSDK locks) that the handle is valid (there are potential race conditions between the lookup and release paths and this is the manner in which they are resolved.)

We expect that for Lustre this is likely to be a useful optimization as it can be used to implement an FSD specific name cache (although it is not required for correctness, merely as a relatively orthogonal performance optimization.)

5.18 FS Handle

The FSDK uses a PVOID value for the handle. We would propose use an extensible table model for managing these handles. In keeping with Sun's suggestion that we use the 128 bit file identifier, we would suggest returning a location in the table to the FSDK and in turn storing the 128 bit file identifier in the table. In that way the handles can have an extremely slow recycle time.

5.19 FS_SET_LENGTH

The purpose of the FS_SET_LENGTH function is to ensure exactly what Lustre provides – effectively a guarantee that the size of the file will be *at least* a certain size. This is part of the Windows VM guarantee that it does not perform extending write operations (it sets the size first and locks against truncates below the level where it will write.) For Lustre this is unlikely to be a major concern (this is an issue for file systems that do not wish to perform allocation during paging write operations.)

5.20 FS_READ/FS_WRITE

In general, Windows writes will be done in units of pages (exceptions can push it down to the size of individual sectors, based upon the sector size reported by the device. The caveat here is that there are now known issues with Windows platforms in which devices report large sector sizes.)