Portable Systems Group

NT OS/2 Opportunistic Locking Design Note

Authors: *Darryl E. Havens, Chuck Lenzmeier and Brian Andrew*

Revision 0.4, June 12, 1991

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This design note describes the current implementation of *oplocks* in the LAN Manager product for Microsoft, its future plans, and the design and implementation of the support for this feature in the **NT OS/2** product.

2. Background

This section describes what an oplock is, its purpose, and the types of oplocks that exist today. Also explained are the features that are being planned for future LAN Manager products.

2.1. What is an Oplock

An oplock is an opportunistic lock. It gives client machines the ability to assume certain information about files that it has open on remote machines for the purposes of buffering information on the client machine. Because information can be buffered on the local client, the amount of network traffic is reduced. That is, the client does not have to write information into a file on the remote server if it knows that no other process is accessing the file because, by definition, no one else needs to see the data.

Likewise, the client can buffer readahead data from the file because, by definition, no one else can change the data that has been read.

Once a file is no longer locked by the client, due to someone else opening the file, for example, readahead data must be flushed and any write data or locks must be applied to the file. This keeps the file in a consistent state. This is referred to as *breaking* the oplock.

2.2. Current LAN Manager Product Features

The current LAN Manager product provides two different types of oplocks:

- *Exclusive oplocks* —This type of locking allows a client to open a file for exclusive access.
- *Batch oplocks* —This type of locking allows a client to keep a file open on the server even though the local accessor on the client machine has closed the file.

Exclusive oplocks are used to buffer lock information, readahead data, and write data on a client machine because the client knows that it is the only accessor to a file on a remote node. The basic protocol is that the redirector on the client opens the file on the remote node requesting that an oplock be given to the client. If the file is open by anyone else, then the client is refused the oplock and no local buffering may be performed on the local client. Notice that this also means that no readahead may be performed to the file, unless the redirector knows that a particular range of the file is locked by the client. Today, no readahead buffering is performed on locked ranges either.

If the client is the only accessor of the file, then the server grants the client an oplock on the file. This informs the client redirector that it is the file's only accessor. This means that the client can perform

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

certain optimizations for the file such as buffering lock and read/write data. This potentially greatly reduces the amount of network traffic between the client and the server.

Batch oplocks are designed to be used where common programs on a client behave in such a way that causes the amount of network traffic on a wire to go beyond an acceptable level for the functionality provided by the program.

For example, the command processor today executes commands from within a command procedure by performing the following steps:

- Opening the command procedure.
- Seeking to the "next" line in the file.
- Reading the line from the file.
- Closing the file.
- Executing the command.

This process is repeated for each command that is to be executed from the command procedure file. As is obvious, this type of programming model causes an inordinate amount of processing of files, thereby creating a lot of network traffic that could otherwise be curtailed if the program were to simply open the file, read a line, execute the command, and then read the next line.

Batch oplocking is designed to curtail the amount of network traffic by opening the command procedure file with an oplock. By having an oplock on the file, the local client redirector can simply skip the extraneous open and close requests. This is done by keeping the file open once it has been opened. When the command processor then asks for the next line in the file, the redirector can either ask for the next line from the server, or it may have already read the data from the file as readahead data. In either case, the amount of network traffic from the client is greatly reduced.

Once the server receives either a rename or a delete request for the file that is oplocked, it must inform the client that the oplock is to be broken if the client redirector's caller actually believes that the file has been closed. This keeps the semantics of the view of the system consistent with what would normally happen where the client redirector had actually closed the file each time its caller closed it.

2.3. Future LAN Manager Product Features

Future LAN Manager products will support several different types of oplocking. The five different types that have been seriously proposed are described in this section. Of these five, the first four have been agreed upon as the set that will be implemented in the future. Whatever design the **NT OS/2** system uses, however, must take into account the desire to perhaps one day implement all of the following types of oplocks.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

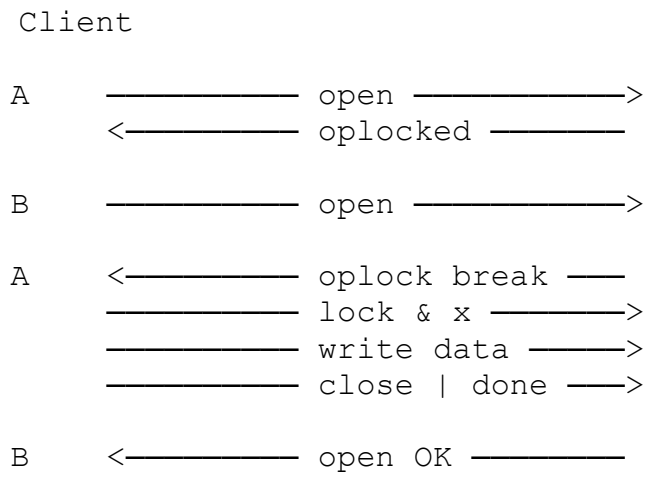
Each of the proposed oplocks is either the same, or builds on, those features currently in the LAN Manager product. These features are designed to further curtail the amount of network traffic on the wire for common situations.

The types of oplocks are:

Exclusive - The same exclusive oplocks that are part of LANMAN today.
Batch - The same batch-mode oplocks that are part of LANMAN today.
Level II - Level II oplocks allow multiple readers to a file.
Restoring - This feature allows broken oplocks to be restored.
Distributed - This feature allows distributed oplocks in the network.

2.3.1. Exclusive Oplocks

The exclusive oplocks proposed for future LAN Manager products is the same functionality that is in the current product. The protocol, in picture format, appears as follows:



As can be seen, when client A opens the file, it can request an oplock. Provided no one else has the file open on the server, then the oplock is granted to client A.

If, at some point in the future, another client, such as client B, requests an open to the same file, then the server must have client A break its oplock. Breaking the oplock involves client A sending the server any lock or write data that it has buffered, and then letting the server know that it has acknowledged that the oplock has been broken. This synchronization message informs the server that it is now permissible to allow client B to complete its open.

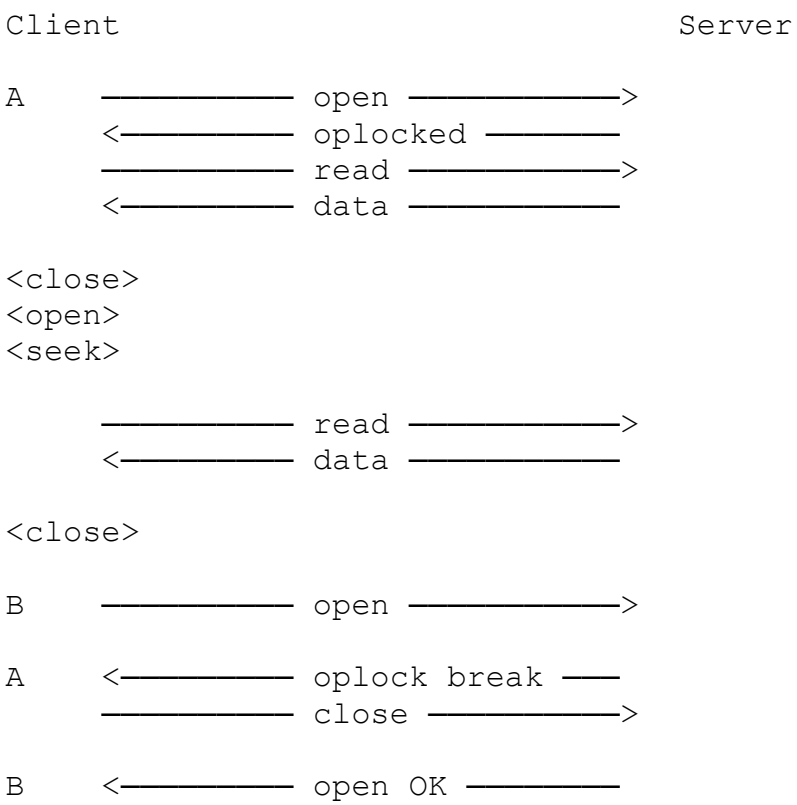
It should be noted that client A must also purge any readahead buffers that it has for the file. This is not shown in the above diagram since no network traffic is needed to do this. Future products may wish to continue to buffer any readahead data that client A knows is locked in the file for its caller.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

It is also possible for client A to complete the oplock break synchronization sequence with a close operation rather than a done. This simply short-circuits the logic in the server to allow it to optimize client B's open request and give it an oplock, provided that client B requested one.

2.3.2. Batch Oplocks

The batch oplock feature proposed for future LAN Manager products is very close to the functionality that is in the current product. The protocol, in picture format, appears as follows:



As can be seen, when client A opens the file, it can request an oplock. Provided no one else has the file open on the server, then the oplock is granted to client A.

Client A, in this case, keeps the file open for its caller across multiple open/close operations. Data may be read ahead for the caller and other optimizations, such as buffering locks, can also be performed.

When another client requests an open, rename, or delete operation to the server for the file, however, client A must cleanup its buffered data and synchronize with the server. Most of the time this involves actually closing the file, provided that client A's caller actually believes that he has closed the file. Once the file is actually closed, client B's open request can be completed.

The difference between this functionality and the feature as it is currently implemented is that in the future client A will be able to specify that it would like to have client B's open request fail. That is,

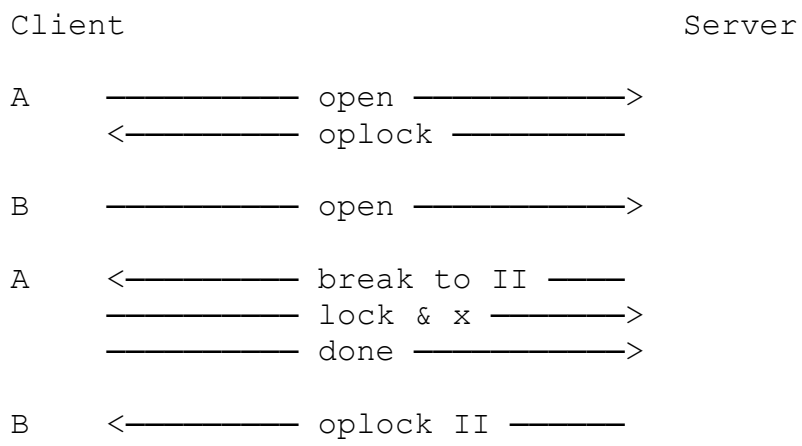
Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

client A would not drop the oplock to the file, so client B's operation should not be allowed to continue. In some senses, this is exactly as if client A had opened the file exclusively or in some mode that is incompatible with client B's request.

2.3.3. Level II Oplocks

A level II oplock is a new feature being proposed for future LAN Manager products. This feature allows multiple clients to have the same file open, providing that no client is performing write operations to the file. This is important for many environments because most compatibility mode opens from down-level clients map to an open request for shared read/write access to the file. While it makes sense to do this, it also tends to break oplocks for other clients even though neither machine actually intends to write to the file.

The protocol, in picture format, appears as follows:



It should be noted that this sequence of events is very much like an exclusive oplock. The basic difference is that the server informs the client that it should break to a level II lock when no one has been writing the file. That is, client A, for example, may have opened the file for a desired access of READ, and a share access of READ/WRITE. This means, by definition, that client A has not performed any writes to the file.

When client B opens the file, the server must synchronize with client A in case client A has any buffered locks. Once it is synchronized, client B's open request may be completed. Client B, however, is informed that he has a level II oplock, rather than an exclusive oplock to the file.

In this case, no client that has the file open with a level II oplock may buffer any lock information on the local client machine. This allows the server to guarantee that if any write operation is performed, it need only notify the level II clients that the lock should be broken without having to synchronize all of the accessors of the file.

> \\ *It would seem that a truly correct implementation of level II oplocks would require the oplock to be broken whenever anyone took out a byte-range lock. This would prevent*

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

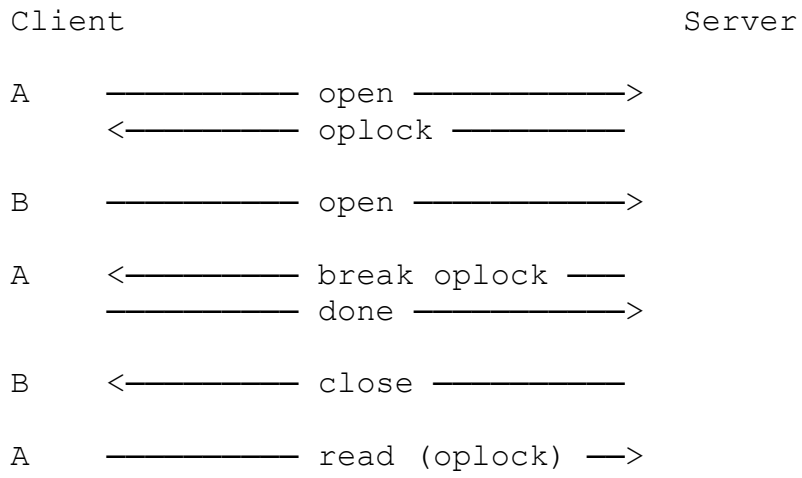
*clients from satisfying reads from previously obtained readahead data that may currently be locked. Perhaps the best approach here is heuristic that allows level II oplocks to be retained in the face of locks until the first write. *

The level II oplock may be *broken to none*, meaning that some client that had the file opened has now performed a write operation to the file. Because no level II client may buffer lock information, the server is in a consistent state. The writing client, for example, could not have written to a locked range, by definition. Read ahead data may be buffered in the client machines, however, thereby cutting down on the amount of network traffic required to the file. Once the level II oplock is broken, however, the buffering client must flush its buffers and degrade to performing all operations on the file across the network.

2.3.4. Restoring Oplocks

Restoring oplocks to a file once they have been broken is a feature being proposed for future LAN Manager products. This feature allows an oplock to be reenabled to a file once it has been broken. To cut down on the amount of network traffic required, this request is piggy-backed on top of other requests that are normally being sent to the server. It will most likely be implemented in the SMB protocol as a simple flag in the SMB header.

The protocol, in picture format, appears as follows:



The protocol for requesting that an oplock be taken out for a file is exactly the same as it is for the previous cases. In the case of restoring oplocks, however, once the oplock has been broken, the client, in this case client A, can request that the oplock be restored. Once client B closes the file, then the oplock can actually be restored to client A by the server.

In the above example, then, should the server determine that the conditions are right to restore client A's oplock, it can simply grant the oplock by sending client A an "oplock gained" message, just as it did when client A opened the file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Note, as well, that if client A closes the file, and client B attempts to take out an oplock on the file, then the server can choose to give the oplock to client B.

2.3.5. Distributed Oplocks

Distributed oplocks refer to the ability to have multiple clients accessing the same file, but giving an oplock to the "active" client. For example, in an airline reservation system, many reservation terminals open the central database file first thing in the morning. However, once the file is opened, the terminals are rarely used. This means that each time a terminal operator wishes to make a transaction, he must assume that someone else is accessing the file as well.

> \\ *The idea of distributed oplocks has been examined by the various individuals in both the LAN and NT groups. To date, no plans have been agreed upon to implement this type of oplocking. It is included herein to give the reader a broad picture of all of the possible oplocks that have been considered.* \\

If a distributed oplocking feature were added to the system, then the oplock could be given to the "active" terminal, if no other terminals were active and hadn't been for some period of time. (Of course, selecting the right "inactive" time is an issue.) This means that if all of the terminals were idle, and one operator started a transaction, that terminal would be given an oplock. It would own this oplock until it went inactive or until some other terminal attempted to access the database at the same time.

This would reduce the network traffic and the processing involved on the oplocked client machine because it would appear to the client as if it was the only accessor of the database file.

3. NT OS/2 Overview

The **NT OS/2** product provides users of the system with the ability to perform oplocking on either a remote node or on the local machine. That is, there are no "back doors" or hidden hacks in the system that are needed to support the oplocking functionality. All users can oplock files. Implementing this functionality in this manner is consistent with the design of the entire system. This also implies that no special code need be added to allow either redirectors or network servers to provide this functionality.

Oplocking functionality in **NT OS/2** is provided through the use of the **NtFsControlFile** system service. This service allows a user to pass file system specific requests to the file system that is servicing the file represented by the user's handle to it. The requests used to oplock a file are standard, non-privileged requests.

Because the **NT OS/2** I/O system is asynchronous by nature, the ability to make a request and then have it completed at a later time makes it natural for implementing oplocks. Further, because synchronization is required by the file system to determine when the caller has completed its oplock update transfers, the file system can use this feature to block open requests to a file by queueing the I/O Request Packet **(IRP)** to its internal file control structure until the oplock owner lets it know that it is finished.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

The user requests an oplock by submitting a request to the file system. If the return status from the system service is failure, then the oplock is not owned. If, on the other hand, the service return status is *STATUS_PENDING*, then the oplock is owned until the I/O operation is completed. At that time, the oplock may have been broken completely, or just to another level. This is indicated by the contents of the *Information* field of the I/O status block.

If synchronization is required by the file system because an oplock was broken to a level which requires this, then the user must flush his buffers, locks, etc. to the file system and then submit another I/O request that specifies that the operation that caused the oplock to break may now be continued.

Given this simple, straightforward design, all of the oplock types can be implemented. Note that because the user actually asks for the oplock after the file is open, rather than at the time the file is opened, oplock restoring falls out.

Implementing a redirector using this design is also straightforward. The redirector always requests an oplock when it attempts to perform an open operation on a remote file. It must also remember whether or not it has gotten the oplock (so it can perform the appropriate local buffering, etc.) When the local user asks for an oplock to the file, the redirector simply completes the request accordingly.

Implementing a server is done in much the same way. If the remote redirector requests an oplock to the file, then the server opens the file and requests an oplock. It then relays whether or not it gained the oplock to the remote redirector.

When it receives a request to open a file that is oplocked, the file system blocks the open request and begins the process of breaking the oplock. Because opening a file in **NT OS/2** is a synchronous operation, this means that the opener's thread is blocked while the oplock is broken. Openers can avoid having their thread blocked by specifying an option on their call to **NtCreateFile** or **NtOpenFile**. If this option is specified, and the file is oplocked, the file system starts the oplock break, then immediately releases the opener's thread, specifying the status code for the open operation and the opener receives a handle to the file. A distinguished success code of ***STATUS_OPLOCK_BREAK_IN_PROGRESS*** is used in this case. When this handle is used to access the file, the operation will block or return ***STATUS_PENDING*** if the requested operation cannot complete immediately due to the state of the oplock.

4. NT OS/2 Oplock Implementation

The **NT OS/2** I/O system provides users with the ability to oplock files by using three FS control functions to the file system servicing the open file, in addition to a file open option.

4.1. Obtaining an Oplock

All oplocks are requested on the open file by invoking the **NtFsControlFile** service with the handle to the open file and one of the following request codes. For more information on the **NtFsControlFile** system service, see the *NT OS/2 I/O System Specification*.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- *Request level I oplock*. —This function requests that an exclusive oplock to the file be granted. This type of request is consistent with the *exclusive* oplock discussed in previous sections. If the I/O request service status is an error, then the oplock was not granted. Otherwise, the request was granted and is held by the requestor until the file is closed or the I/O request completes later, indicating that the oplock has been broken to a level II oplock or no oplock.

  The file system control code for this function is ***FSCTL_REQUEST_OPLOCK_LEVEL_1***. The input and output buffers are not used. If the oplock was granted, then when the I/O request completes, the *Information* field of the I/O status block indicates whether the oplock has been broken to level II (*FILE_OPLOCK_BROKEN_TO_LEVEL_2*) or to none (*FILE_OPLOCK_BROKEN_TO_NONE*).

- *Request level II oplock*. —This function requests that a *Level II* oplock to the file be granted. Again, if the I/O request service status is an error, then the oplock was not granted. Otherwise, the request was granted and is held by the requestor until the file is closed or the I/O request completes at a later date. If the latter occurs, then the oplock has been broken to none.

  The file system control code for this function is ***FSCTL_REQUEST_OPLOCK_LEVEL_2***. The input and output buffers are not used.

- *Request batch oplock*. —This function requests that a *Batch Oplock* to the file be granted. The semantics of a batch oplock are the same as for level I oplocks except that a subsequent open of the file will initiate the oplock break before the access sharing check is made. Unless specified, all references to level I oplocks will also refer to batch oplocks.

  The file system control code for this function is *FSCTRL_REQUEST_BATCH_OPLOCK*. If the oplock was granted, then when the I/O request completes, the *Information* field of the I/O status block indicates whether the oplock has been broken to level II (*FILE_OPLOCK_BROKEN_TO_LEVEL_2*) or to none (*FILE_OPLOCK_BROKEN_TO_NONE*).

If the **NtFsControlFile** service (not the I/O request) completes with a status other than *STATUS_PENDING*, then the oplock was not granted. If the status is *STATUS_PENDING*, then the caller owns an oplock on the file. In this case, the I/O request does not complete unless and until the oplock is broken.

If a level I oplock is requested, but the file is already oplocked (at any level), the request is rejected. If a level II oplock is requested, but the file is already oplocked at level I, the request is rejected. If a level II oplock is request on a file that is already oplocked at level II, the request is accepted.

If the owner of a level I oplock requests a level II oplock, the request is rejected.

If the owner of a level II oplock requests a level I oplock and no one else has the file open, then the level II oplock will be broken and the level I oplock will be granted. The level II oplock is broken by completing the Irp used in granting the level II oplock.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

4.2. Opening an Oplocked File

When an oplocked file is opened again, the file system initiates an oplock break. This involves completing the FS control request(s) that created the oplock. The file system normally blocks the second open request (and subsequent opens) while the oplock is being broken.

For a level I oplock, the oplock is not considered broken until the owner of the lock issues an Accept Oplock Break FS control (see next section). This allows the owner to flush writebehind data and byte-range locks to the file before releasing the oplock.

For a level II oplock, the oplock is considered broken immediately; the owner has no writebehind data or locks to flush, so there is no need to wait for acknowledgement. The owner need not issue an Accept Oplock Break request, but the file system should not consider it an error if one is issued.

An opener can avoid having its thread blocked while waiting for a level I oplock to be broken by specifying the *FILE_COMPLETE_IF_OPLOCKED* option on the call to **NtCreateFile** or **NtOpenFile**. If this option is specified, and the file is oplocked at level I, the file system starts the oplock break, then immediately releases the opener's thread by completing the I/O request. The status of the I/O is *STATUS_OPLOCK_BREAK_IN_PROGRESS*, and the *Information* field of the I/O status block is determined by the result of the open, disregarding the oplock state of the file. The opener receives a handle to the file and may access the file using this handle. If the file is accessed and the operation cannot complete until the oplock break is completed, then the calling thread will block or ***STATUS_PENDING*** will be returned based on whether the operation is synchronous or asynchronous.

Batch oplocks present a problem in that a remote user may have opened a file with restricted share access (read-only). A batch oplock is obtained on this open file. The remote user may call to close the file, but the redirector holds the file open in anticipation of future open and read calls. When the remote user attempts to reopen the file with more liberal access (read-write), the open will fail unless the redirector acknowledges the oplock break and closes the file. The share access check for any subsequent opens of a file with a batch oplock will be blocked until the owner of the oplock has acknowledged the oplock break and closed the file (if it intends to do so). This subsequent open may elect not to block by specifying *FILE_COMPLETE_IF_OPLOCKED*, but this may cause the open to fail due to performing the share access check prematurely.

4.3. Accessing an Oplocked File

If an opener specifies that the open operation is not to block due to the oplock state of a file, then it is possible that the file may be accessed via a handle created in this way prior to the completion of the oplock break. This has no effect on files with a level 2 oplock, as they are broken immediately. Breaking a level 1 oplock requires an acknowledgement from the owner of the oplock after locally buffered data and lock requests are flushed. An operation requested on the file during the oplock break operation will be blocked pending the completion of the oplock break. If the requested operation is a synchronous operation, the thread will block pending the completion of the oplock break. Otherwise

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

STATUS_PENDING is returned and the IRP is completed when the oplock break acknowledgement is received.

The above conditions apply to the following operations:

- NtReadFile -- All read operations on the file will be blocked.
- NtLockFile -- All byte range lock requests on the file will be blocked.
- NtUnlockFile -- All byte range unlock requests on the file will be blocked.
- NtQueryInformationFile -- Any query operation involving the following file information classes will be blocked: ***FileBasicInformation, FileStandardInformation*** or ***FileAllInformation.***
- NtSetInformationFile -- Any set informtion operation involving the following file information classes will be blocked: ***FileBasicInformation, FileAllocationInformation*** or ***FileEndOfFileInformation.***
- NtWriteFile -- All write operations on the file will be blocked.

4.4. Releasing an Oplock

In response to the breaking of a level I oplock, the owner of the lock must flush any pending write behind data and lock requests. The owner then issues the following **NtFsControlFile** request:

- *Accept oplock break*. —This function is used to synchronize with the file system once an oplock has been broken. When this request is issued, the file system restarts any pending open requests and any operations blocked pending the completion of the oplock break. The file system control code for this function is ***FSCTL_OPLOCK_BREAK_ACKNOWLEDGE***.

  If the level 1 oplock is being broken to level 2, then the IRP used to acknowledge the oplock break is treated as a request for a level 2 oplock. If the level 2 oplock can be granted at this time, then ***STATUS_PENDING*** is returned. Any other return code indicates that the level 2 oplock is not granted. The IRP will be completed when the level 2 oplock is later broken.

- *Releasing a Batch Oplock*. —When a batch oplock break is initiated, the original Irp is is completed with a status indicating whether the oplock is being broken to none or to level II. The owner of the oplock will need to update the file with any locally buffered changes and then acknowledge the oplock break. The file system control code in this case is *FSCTL_OPLOCK_BREAK_ACKNOWLEDGE*.

  In many cases the oplock break acknowledgement is followed immediately by a close call. In this case, it is desirable to hold off any opens of the file until the close is performed. If the oplock break is acknowledged with the file system control code

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

FSCTL_OPBATCH_ACK_CLOSE_PENDING, then the in-process open and any subsequent opens will be blocked until remote close operation has been completed.

An oplock break may be initiated by the owner of either a level 1 or level 2 oplock by calling **NtClose** on the handle used to request the oplock. In the case of a level 1 oplock, the close operation also serves as the acknowledgement of the oplock break. The IRPs for the oplocks are completed and any operations pending due to the oplock state are continued. NOTE -- This action occurs in the "Cleanup" operation in the file systems, not the "Close" operation.

5. Design Issues

This section presents the issues that need to be resolved before the oplock design herein can be fully implemented.

5.1. Timeouts

In the LAN Manager product today, timeouts are implemented on the server once it has broken an oplock for a client redirector. The redirector has 45 seconds (or so) to cleanup its buffered data, flush its locks, etc., and then submit the packet that specifies that it is okay to continue. If this doesn't occur within the specified timeout period, then the redirector's session is closed.

The issue here is two-fold:

1. Should the local machine attempt to timeout a user in the same manner, and if so, what should the timeout value be and how does the file system efficiently implement this?

2. If the local file system times out users, what should the action be? In the remote case the redirector's session is lost. This is unfortunate because this means that the file is left in an inconsistent state. Perhaps in the local case the original accessor should be given exclusive access to the file?

~~5.2. Batch Oplocks~~

~~It's not clear whether NT OS/2 needs to implement specific support for batch oplocks. The distinction is made in OS/2 because open files in that system cannot be deleted or renamed. In NT OS/2, the target file *must* be open in order to be deleted and renamed. The ability to delete or rename a file that someone else has open is controlled by sharing modes.~~

~~This implies that exclusive vs. batch oplocks can be implemented in the following way:~~

- ~~An application can obtain OS/2 semantics for exclusive oplocks by not specifying *FILE_SHARE_DELETE*. Attempts to open the file for delete or rename will fail, and the oplock will be retained.~~

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

- An application can obtain OS/2 semantics for batch oplocks by specifying *FILE_SHARE_DELETE*. An attempt to open the file for delete or rename will cause the oplock to be broken.

The NT OS/2 LAN Manager server could implement batch oplocks in this manner. The server normally opens files without *FILE_SHARE_DELETE*, in order to obtain the sharing semantics required by the SMB protocol. However, if a file is opened with a batch oplock requested, the server could allow sharing for delete and rename.

This leads to a potential problem:

If the server allows delete sharing, but is then unable to obtain the oplock, then we have a situation in which the normal sharing rules are not being obeyed ——the file can be deleted or renamed out from under the client. Perhaps this is not a problem; perhaps it is a small price to pay. On the other hand, to maintain the sharing rules, perhaps the server should close the file and reopen it without delete sharing.

Now suppose that we do obtain the oplock, but it is subsequently broken. Again, we have the file open with delete allowed, and the questions above apply. This problem may not be important, however, because the response to a batch oplock break should be to close the file.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

6. Revision History

Original Draft Revision 0.1, April 2, 1990

Revision 0.2, August 15, 1990 -- Implementation details added

Revision 0.3, January 2, 1991 -- Implementation details of initial implementation added. Corrections to initial implementation details made.

Revision 0.4, June 12, 1991 -- Interface extended to support batch oplocks.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License