1 Correctness

Correctness: A sequence of transactions are said to be consistent if and only if a total ordeing of them according to precedence relationship can be established that demonstrates the same behavior as the execution of the program. Behavior for an operation means the data it has read or wants to write. Demonstrating the same behavior thus means all the read operations should still see the same data in the new sequence as they have seen in the actual sequence. However writs always writes the same value no matter what. Hence the behavior of a write operation is not alterable.

Def 1- *Set of Primitive Operations*: Operations are taken from the set {force-readoffset(filedescriptor), writeoffset(filedescriptor), readdata(inode, offset, length), writedata(inode, offset, length), commit}.

Def 2- *Set of User-Level Operations*: User-Level Operations are taken from the set {Read(fildescriptor), GetFilePointer(filedescriptor), Write(filedescriptor), Seek(filedescriptor), EndTransaction}.

Note 1- *Assignment Operations Need Not Be Shown In* $OP_{excecuted}$: Operations like offset = offset + length and other assignment operations in OP_{T_i} need not be shown in the actual sequence of operations namely $OP_{executed}$ that consists of operations executed by different transaction so far, the reason is simply all such operations are local to the transaction and do not affect any other transaction's state and hence do not restrict the commution of other operations in any manner.

Note 2 *Forced-Readoffset(fd)*: Reads the offset for the filedescriptor and makes the transaction bound to this value.

2 FileDescriptor Offset State per Transaction

:

Each filedescriptor has an associated offset with it, within each transaction this offset can be in 4 different states, these states indicate the dependency the transaction has on the value of this offset:

1- **No Access**: This is the default state for all filedescriptors in a transaction and is changed as soon as there is an access to the the descriptor within the transaction (any of the use-level operations are invoked).

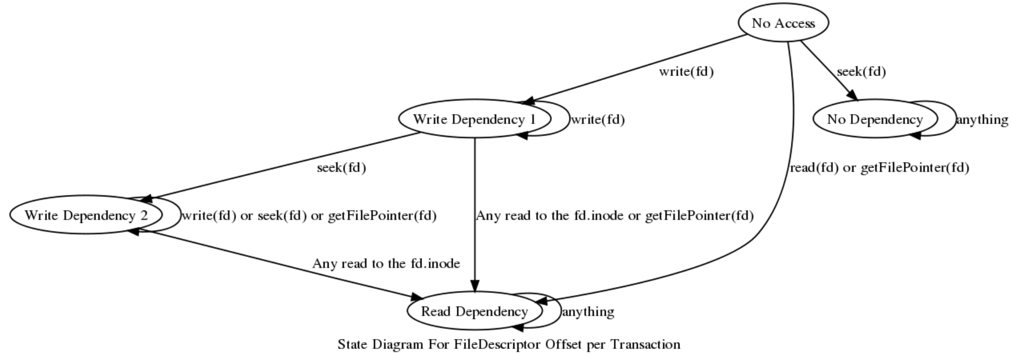
2- **None**: Meaning there is no dependency for any operation in this

transaction on the value of the offset associated with this descriptor regarding other transactions.

3- **Write Dependency**: This kind of dependency means there is at least one operation in OP_{T_i} having an unknown offset(essentially a write operation) value as argument. The value of this unknown offset will be determined at commit instant.

4- **Read Dependency**: This kind of dependency means there is at least one operation acting on an offset value for fd, where the value for fd has been determined by a previously committed transaction.

The state machine below depicts the behavior of the user level operations regarding how the offset corresponding to that transaction changes:



Explanation: Whenever the offset status for the transaction goes to "Read Dependence", a "forced-readoffset(fd)" operation is issued immediately preceding the operation that caused this transformation. The forced-radoofset(fd) is only issued if there is not a forced-readoffset(fd) in the OP_{T_i} already.

Axiom For Diagram: If the first access to a filedescriptor is a $Seek(fd)$, then the following operation on fd gets the offset value from the assignment made by the $Seek(fd)$ and advances the offset. Hence, the following operations get this offset and the offset value the filedescriptor had had before this transaction accesses fd is never referenced (thats why it is an absorbing state). This conforms to the definition of "None" state.

If $Read(fd.inode, offset, length)$ or $GetFilePointer(fd)$ is the first access made by this transaction, the offset is read (since the data needs to be read at this instant), this offset should be the one committed by a previously committed transaction (as this is the first access to fd in this transaction).

Once the offset is read, it is always dependent on this value (hence an absorbing state). For all the following operations, the offset value is known. This conforms to the definition that there is at least (the first read ever on fd by this transaction) one operation that acts on the offset value for fd and rules out the "Write Dependency" and "None" states.

If $Write(fd.inode, offset, length)$ is the first access made to fd by this transaction, then the offset to write to, can be decided at commit instant since the Write functions means start writing at the most recent committed fd.offset, hence offset realization can be postponed till commit instant. Any Writes or Seeks would still leave this dependency, since operations after a Seek act on absolute offset, and Writes preceding any Seek can all determine the offset at commit instant for the same reason as before.

However, if a Read on the same fd in the transaction is invoked there are two possibilities:

1- A Seek precedes this Read, hence the offset value is absolute and is not read, however the ranges that are supposed to be written by Writes preceding the Seek, may overlap with the range Read is willing to Read from, and according to Rules(Most Recent Changes Should Be Visible) if thats the case the Read should be able to see this data, this suggests the ranges that all the Writes are going to write to should be realized now and this requires settling down on a value for all file descriptors offsets for this inode at this instant. Based on these, the most recent committed offset value for all these descriptors should be assigned to the offsets for the Writes that for the first time accessed the descriptor. Other for writes preceding the Seek, get this value as being advance by prior writes.

2- No Seek precedes the Read, hence the offset value the Read has to read from is unknown, since preceding Writes to fd have all used unknown offsets, the offset value given to Read is an unknown once, however it has to be known, follows that the offset value for the Write that for the first time accessed this fd should be decided upon and as shown before, the value should be the most recent committed offset value for fd. The offset value for this read or other writes, is the offset value obtained as being advanced by those operations.

The two same possibilities exist when a GetFilePointer operation is invoked on the same fd:

1- If a Seek precedes it, then the offset value becomes absolute and hence the getFilePointer could retrieve the value assigned by Seek.

2- Otherwise, the offset value is still unknown, hence to be able to determine the offset value at this instant, the value obtained by reading the last committed offset value should be assigned to the offset value. for the first Write to fd.

3 User-Level Operations Structure

The user-level operation cab be broken as follows:

1- Seek(fd): This operation sets the offset for filedescriptor. We define it as demonstarted below:

Just an internal assignment in the transaction, $\{fd.offset = value\}$.

2- Write(fd):

1- $\{writedata(fd.inode, offset, length), fd.offset = fd.offset + length\}$

3- GetFilePointer(fd):

$\{\{forcedreadoffset(fd)$ issued as demonstrated at the state diagram if any, it is issued when the state for fd in this transaction is not in "No Dependency" or "Write Dependency 2"$\}\}$

4- Read(fd):

$\forall filedescriptor fd_i$ where $fd_i.inode = fd.inode$ and the state for fd_i in this transaction is not $NoDependency$, Read(fd) $=\{\{forcedreadoffset(fd_i)\}\}, readdata(fd.in$
$offset + length\}$

5- EndTransaction:
$\forall fd_i$ such that the state for fd_i in this transaction is not "No State", $EndTransaction = \{\{writeoffset(fd_i)\}, commit\}$

Essentially for any fd that the correspondent state diagram in the transactions shows is in a state other than "No Access", a writeoffset(fd) is issued while committing.

4 FileDescriptor Offset State per writedata Operation

Any writedata operation within a transaction gets fd and an offset as arguments. There writes are reflected in the commit instant, however the offset to write to as we saw earlier for some writes i s determined at commit instant and for other is bound to a specific value before commit instant. We should have a policy to be able to diffrenciate between these two. The 3 rules below odes this.

1- If the state for fd a given distinguish in a transaction is "Write Dependency 1" all writes by that transaction to that fd will get the value of the offset to write to at commit instant (since in "Write Dependency 1" all writes are at unknown offsets) .

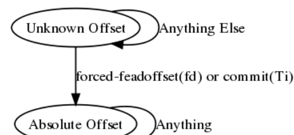
2- If the state for fd is "Read Dependency" or "No Dependency" then all writes on fd within this transaction should be done at offset determined for them when they were invoked (since all writes are to known offsets either determined by the transaction itself or a previously committed transaction).

3- Otherwise, if the state is "Write Dependency 2", then if the $writedata(fd.inode, ...)$ operation precedes a $Seek(fd)$ then the offset is determined at commit instant (since all such writes are at unknown offset). Otherwise, the writes should be done at the offset decided upon earlier (the write after a Seek write the offset determined by Seek and hence to a known offset).

We could also think of this as a state machine for each writedata operation. The state diagram is created for each operation when it is first invoked and is subject to two things:

1- If there $\exists Seek(fd) or forced-readoffset(fd) \in OP_{T_i}$ such that those precede the $writedata(fd.inode, ...)$ then the initial state in the state diagram for this writedata is Absolute.

2- Otherwise the initial state is Unknown offset.



State diagram for FileDescriptor Offset per writedata(fd.inode, offset, lenght) operation in Ti

The final state for all $writedata$ operations is Absolute, since the write should be perfomed at a specefic offset eventually. However, depending on

the prevuious circumstances the system would immediately prior to commit determine the offset or would have realized it earlier.

5 Guidelines for Implementaion

As we saw earlier in Rule 4, any two operations can commute across each other unless they are subject to one of the two conditions in Rule 3.

Guideline 1: *Commuting forced-readoffset Operations*: A $forced - readoffset(fd) \in OP_{T_i}$ can go past a $commit_{T_j}$ if and only $\nexists writeoffset(fd) in OP_{T_j}$.

Axiom: It follows immediately from Rule 4 and conditions in Rule 3, that this $forced - readoffset(fd)$ can go pats the commit. What remains to be proven is $\not\exists writeoffset(fd) \in OP_{T_i}$ such that it precedes $forced - readoffset(fd)$, as this would mean even if $\exists writeoffset \in OP_{T_j}$ still the $forced - readoffset(fd)$ could commute with $commit_{T_j}$.

This stems from the definition of EndTransaction operation, and the state diagram. A $forced - readoffset$ can be issued at any place in OP_Ti however it would always precede the $writeoffset(fd) \in OP_{T_i}$ since this is last operation in OP_{T_i} before *commit*.

Guideline 2: *Commuting readdata Operations*: A $readdata(fd_i.inode, offset_i, lenght_i) \in OP_{T_i}$ can go past a $commit_{T_j}$ if and only if

1- $\nexists writedata(fd_j.inode, offset_j, length_j) in OPT_j$ such that $fd_j.inode = fd_i.inode$ and the two ranges $(offset_i, offset_i + length_i)$ and $(offset_i, offset_j + length_j)$ have an intersection.

OR

2- $\exists$ a set of $writedata(fd_j.inode, offset_j, lenght_j) in OP_{T_i}$ such that for all of them $writedata \rightarrow readdata$ and $fd_i.inode = fd_j.inode$ and the range $(offset_i, offset_i + length)$ is a subrange for a combiantion of ranges for these *writedata* operations.

Axiom: This follows imeediately from Rule 4 (1 and 2 are concrete representations for conditions 1 and 2 in Rule 3). However, it should be noted that whenever a *readata* operation wants to commute across a *commit* operation $\in T_j$, if $\exists writedata \in OP_{T_j}$, the offset for the *wtitedata* operation is known at this instant (according to the state diagram demonstarted below and regardless of whether $i = j$ or not), hence it would be trivial to see

whether these *writedata* operations intersect (or include in case they are in the same transaction as the *readata*) with the *readdata* or not.

Guideline 3: *Reads Should be Validated At Commit Instant*: If an operation a that "reads" some data r_n (reads the data at some $t < t_{commit-of-the-transaction}$(the commit instant)), it should be ensured that the data r_n is still valid in the actual file system at $t_{commit-of-the-transaction}$ (commit instant) or T_i has to abort.

Axiom: If the data is not valid anymore it means the data r_n has been written since t_{i-1}. This implies at least one operation b has written the data since the use and b does not precede a and a precedes b. b is either an operation in the same transaction or a different one. However, if b is in the same transaction, then the data read is valid at commit instant by defenition. On the other hand, assume b belongs to a different transaction namely T_j. Since T_j should have already commited, according to Corrolary 2 all operation in OP_{T_j} should precede those in OP_{T_i}, however, we know there is at least an operation b that does not precede a (since a has not seen the "writes" made to r_n by b). Hence, T_i can not commit.

Guideline 4: *If Reads Are Valid At Commit Instant, the Transaction Commits*: If by applying Guideline 3 for T_i it is ensured that $\forall r_i \in RT_i$ (all data read by T_i) is still valid at commit instant, then $\forall T_j \in T_{committed} T_j \rightarrow T_j$ and hence according to Rule 4 T_i commits.

proof: If all data read is still valid at commit instant, means all operation in the set of operations belonging to committed transactions, precede those in T_i (since no writes have been seen), and consequently all those transactions precede T_i. Rule 5 ensures such transaction would be able to commit.