Date: August 30, 1987
To: X3T9.2 Working Group
From: James McGrath
Quantum Corporation
1804 McCarthy Blvd
Milpitas, CA 93035
(408) 432-1100
Subject: Reordering of Commands in Command Queuing

This is an attempt to demonstrate the practicality of having the Target reorder commands which have been queued for a specific LU under its control with a minimum of explicit direction by the Initiator. This topic has been the subject of great debate in the past, and thus a clear and precise written explanation was deemed appropriate. While this discussion is only directly applicable to DASD, the same concepts can profitably be applied to any SCSI device.

This discussion is NOT intended to indicate how command queuing MUST be implemented by the Target in order to insure correct execution. Rather, it simply illustrates one possible implementation that does insure correctness at a reasonable cost (in overhead and performance) and is easy to analyze.

First, unless otherwise stated, all terms used are as defined by the SCSI-2 REV 2 document or the Command Queuing Proposal. The following terms are new:

Explicit Ordered Command - an ordered command, with an ordered tag, as defined in the Command Queuing proposal.

Implicit Ordered Command - an unordered command, with an unordered tag, but one that the target has determined that it will treat as an ordered command for the purposes of queuing.

Ordered Command - either an explicit or implicit ordered command.

Head of Queue Queue - the queue for a specific LU containing the Head of Queue commands for that LU.

Primary Queue - the queue for a specific LU containing the ordered and unordered commands for that LU.

(Primary) Queue Segment - each Primary queue can be divided into a series of one or more segments. Each segment normally consists of a sequence of commands containing zero or more unordered commands and one ordered command such that the ordered command is the last in the sequence and the unordered commands are those which arrived after the ordered command of the previous segment in the queue and before the ordered command in this segment. The last segment in the queue is a special case which may not include an ordered command. For example, a queue containing commands in the following order:

U U O O U O U O O O U U U U U

can be divided into segments as follows:

(U U O) (O) (U O) (U O) (O) (O) (U U U U U)

where U represents an unordered command and O represents an ordered command.

Reordering Rule - the algorithm used by a Target to reorder commands in the Primary queue of a LU.

Regeneration Point - the point in time when no command is under execution and the first command of a new segment in the Primary queue is the next command to be executed.

State of the Media - at any particular moment, the state is defined to be the complete mapping of Logical Block Addresses to the data stored in those LBAs. Thus the state is a measure of the contents of the device.

Correct Execution Sequence - any sequence of execution from the command queue(s) for a LU that both obeys the rules for command queuing and which results in the state of the media, and the data returned to the Initiator concerning the contents of the media, to be identical to those of a FIFO execution of the primary queue. (Note: the state of other components of the target, such as the buffer, are NOT gaurenteed to the be same under different reorderings that result in correct execution.).

THESIS: the Target can implement Reordering Rules which result in a Correct Execution Sequence at

1) low cost in command overhead,
2) high improvement in performance, and
3) without requiring the Initiator to explicitly order commands (although such ordering shall be allowed).

Under any reordering rule, only the reordering done within a queue segment can make the execution sequence incorrect.

This follows directly from the definitions given above and the entire philosophy of command queuing, under which the explicit ordering of a command or the use of a head of queue command indicates that the Initiator is removing any control of order of execution from the Target. Doing so shifts any risk that the resulting execution sequence may be "incorrect" from the Target to the Initiator.

A sequence of execution is correct if for each queue segment the execution of commands in that segment, if considered to be the total queue for the LU, would be considered to be correct.

Since the order of execution of head of queue commands and the order of execution of queue segments is restricted to a single ordering by the rules of command queuing, only reordering within a segment can create a deviation from the FIFO primary queue execution sequence which is always correct.

We will assume all unordered commands other than READ, READ EXTENDED, WRITE, and WRITE EXTENDED to be implicitly ordered by the target.

Note that this assumption does not significantly decrease the performance gains to be realized by reordering (since the remaining unordered commands still make up over 99.9% of the commands actually encountered during normal execution), nor increase the overhead (since a simple op code check is all that is required), but will significantly simplify the analysis of reordering rules. Targets might be able to insure correct execution sequence without this restriction, but allowing such commands as MODE SELECT, RESERVE/RELEASE, and FORMAT to be reordered obviously leads to potential difficulties and much complexity for little gain.

The test for correct execution is made at regeneration points. Note that commands cannot be reordered across regeneration points. This implies that halting execution (e.g. for an error) in the middle of a queue segment will leave the state of the media in an incorrect state. As always, it is up to the Initiator to successfully perform recovery operations.

All segments (except for the last, which we treat as a special case) are finite, and any reordering algorithm will eventually result in reaching a regeneration point. For the last segment, simply must insure that all commands are executed in a finite period of time (i.e. starvation does not occur). Many popular reordering algorithms (such as CSCAN) will prevent starvation, and we assume one such is implemented.

Thus we have finally be reduced to requiring that the reordering of commands within a segment does not result in the return of data which differs from that of a FIFO execution nor leaves the media in a different state. Note that under any reordering the ordered command is always constrained to be executed last. Thus as long as the data returned and the state of the media for the sequence of unordered commands meets the correctness criteria, then the commands in the segment as a whole will be correctly executed.

All unordered commands in a segments are either a variety of READ or WRITE. Consider the N unordered commands in a segment to be numbered 1..N. Then any reordering is uniquely defined by the N! ordered pairs of commands (x,y), where the each pair implies that command x comes before command y in the reordering. We will concentrate on these pairs.

If all the pairs were (READ,READ) pairs (i.e. all unordered commands were READs), then any reordering could not affect the state of the media (since it is never changed) nor the returned data. Similarly, if a pair was a (READ,WRITE), a (WRITE,READ), or a (WRITE,WRITE) then the reordering of these two commands could not affect correctness as long as the range of the specified LBAs for each command did not intersect.

Thus the above is both a necessary and sufficient condition for generating a correct execution sequence. However, the Target need not generate the N! pairs and perform the check required by theory. A more practical implementation of the above test would be the following.

First, any reordering of commands implies that a sorting operation (usually with respect to the LBA of the command) be performed. The sort may result in an explicit data structure (i.e. a binary tree of pointers) or an implicit structure (i.e. the CDBs are reordered in an array, or an array of pointers to CDBs are reordered). In any event, we will denote T as the time required to perform such sorting, and the resulting sequence of execution is denoted as A.

This list is now sorted so that the LBA+TL of the immediately preceeding command is <= the LBA of the next command. Note that LBA+TL is one more than the last LBA in the command, and this sort can be performed at a cost no greater than T (note that LBA+TL must be computed for each command anyway in order to perform a range check against the LU's maximum LBA, and that a more sophisticated data structure can reduce the incremental effort to perform this second sort considerably). This ordering is denoted as B.

[redacted]

For each segment, a command has a position in both queues denoted by the pair (a,b). The execution sequence is then determined as follows:

1) Attempt to execute command in the ordering determined by A.

2) If a = b, then execute the command.

3) If a < b, then scan A until you find a command equaling b. For all commands in A between a and this b, search B and keep track of the command that appears last in B (denote this c). Now scan A again, but use c as your search target instead of b. Continue the search process, alternating between A and B, until you run out of commands to search for. The result is a subsequence of commands in A and B such that each command in the subsequence in A appears in the subsequence in B and vica versa, but the orders differ between the subsequences. These commands should be executed in the original FIFO order (i.e. both reordering should be ignored).

4) when done, goto step 1) again until the queue is empty.

As an example, considering the following pairs of ordered LBA ranges:

(0,3) (6,8) (7,12) (8,15) (20,23) (28,32) (31,35) (36,39) (37,38)
(0,3) (7,12) (6,8) (8,15) (20,23) (31,35) (28,32) (37,38) (36,39)

Thus the execution order is:

(0,3)
(6,8) (7,12) in FIFO order
(8,15)
(20,23)
(28,32) (31,35) in FIFO order
(36,39) (37,38) in FIFO order

Note that other execution sequences may be defined that provide greater performance (i.e. (READ,READ) sequences can be freely reordered) at a cost of greater command overhead. But in the normal case of few intersections, the total overhead is 2*T plus a check per command (this can grow to N*N checks in the worse case).

Finally, command overhead should not be an issue in command queuing. Since overhead grows as the queue lengthens, but since the opportunity to overlap queuing tasks with seek time and rotational latency grows with the queue length, most if not all of the queuing overhead can be effectively hidden from the user.

30

Explicit ordering of commands by the Initiator can shift the the implementation burden from Target to Initiator, and this may have many practical benefits. Error recovery might prove easier to implement, and Target resources might be more profitably used.

A note of personal preference: the more I have examined this topic, the more I have reluctantly come to the conclusion that it may be best to allow targets to insure correct execution, but not require them to do so. It should be made easy for the initiator to determine if correctness is maintained by the target (i.e. another mode bit and/or information in inquiry).

In any event, I hope that the central thesis - that the Target can maintain proper order under command queuing - is clear, and that debate can now shift as to whether it is desirable to do so or not.