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 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 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 O O U U U U U U
```

can be divided into segments as follows:

```
(U U O) (O) (U O) (U U) (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 guaranteed to be the same under different reorderings that result in correct execution.)
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 ensure 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 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 segment 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) and the returned data. Similarly, if a pair was (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 preceding 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.
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 vice 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:

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

Thus the execution order is:

\[
(0,3) \quad (6,8) \quad (7,12) \quad \text{in FIFO order}
(8,15)
(20,23)
(28,32) \quad (31,35) \quad \text{in FIFO order}
(36,39) \quad (37,38) \quad \text{in FIFO order}
\]

Note that other execution sequences may be defined that provide greater performance (i.e. (READ,READ) sequences can be freely reordered) at the cost of greater command overhead. But in the normal case of few intersections, the total overhead is \(2*N\) 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.