OpenVMS Technical Journal V6

Threaded Tests in the OpenVMS Cluster Test Manager (CTM)

Threaded Tests in the OpenVMS Cluster Test Manager (CTM)

Richard Stammers

Overview

Testing extremely complex systems, such as large OpenVMS clusters involving SMP systems, is a race against time. It is daunting, in fact, when you view this testing as a "combination search" of all the various states that the cluster and the systems can be in. Moreover many of the hardware and software components of such systems are, by their nature, autonomous; they work together and co-operate heterarchically rather than having some "super system" or "super process" that controls their operations. As a result, most combinations of states are theoretically possible and hence must be tested. As Murphy would have it, "Anything that can go wrong will!" In practice, we have to cheat and connive to accomplish all the required testing in a reasonable time. We have to force the systems into all the nasty conditions that otherwise occur so infrequently that otherwise would be missed in the finite amount of time available for testing. There are limits to the extent that we can "cheat and connive." The components under test cannot be changed, and the tests have to be valid in the context of OpenVMS. Threads provide an effective way to contrive tests that explore various combinations of system states, and are essential to properly testing SMP systems. This article discusses in general terms how threaded tests are designed and used, showing specific examples of threaded tests that are employed in CTM, the OpenVMS Cluster Test Manager.

Introduction

Threaded tests in CTM are organized and designed so that multiple instances of routines used in the test can operate concurrently. During testing, it is important that we do as much as possible with the time that is available. As with all types of applications, threads can be used to provide a significant improvement in performance, especially on multiprocessor systems, where threads can run concurrently on separate processors. This parallelism also allows multiple execution threads to run concurrently in the context of the test, thus allowing us to verify that execution threads running on different processors are synchronized and maintain coherency.

Writing a test is a little different from writing other types of application programs. A good application programmer uses his knowledge of the system to make the application run as quickly and efficiently as possible. Tests are often written with exactly the opposite intention. Problems or defects show up much more quickly when the system is deliberately stressed and forced into difficult situations, so while application programmers use their knowledge of a system to avoid these situations, a person writing a test must approach things with exactly the opposite intention, designing the test so that nasty situations occur as frequently as possible.

CTM tests the OpenVMS operating system, including hardware and software components, at the application level. And because the threaded tests in CTM use it, they also test the multithreading runtime library. This is an important consideration.

No matter how zealous the tester is to try and break the system, the tests always have to abide by the rules! Moreover there are human factors to be considered. Tests are written and performed to ensure the quality of the final product; finding a problem or a defect is only the first step towards fixing it.

In some ways, the multithreaded runtime library is like a “mini operating system;” the results, logs, and traces from a threaded test are often complex and voluminous. The person writing the tests must guard against undue complexity and zeal must be tempered with the practical consideration of how the test will be used to pin-point and remedy any problems and defects that are found.

One final point to be considered when designing any test, particularly tests that make use of threads, is that a test that fails to find a problem or a defect, or that detects spurious problems that do not in fact exist, is worse than useless. On the one hand, time and effort is expended in writing and running the test, only to provide a false sense of security that everything is alright. On the other hand, that

time and effort would be wasted in tracking down a "problem" that doesn't exist. The design of any test should verify beyond any reasonable doubt that the test itself is working properly.

General Methodology

As described in an earlier OpenVMS Technical Journal article, CTM is not a test, but rather it is a framework which permits the control and running of multiple different tests in an OpenVMS cluster environment. To clarify this distinction, we refer to the framework in which the tests are run as the "test harness," as distinct from the actual tests that are run by the harness under the direction of the test engineer.

The threaded tests in CTM run under a "gang scheduling" architecture (see Figure 1). Central to each of the tests is the main thread, which communicates with and is controlled by the test harness. It receives from the harness any parameters that control the test, validates them, and uses them to perform as much of the test initialization as possible prior to the start of the test. In fact, as much of the work inside CTM is done in the initialization stage as is possible (for example, setting up all the data buffers and data patterns the test will use, using alternating data patterns instead of poisoning buffers between data cycles, and so forth). When the actual test runs, it can run as quickly as possible. The main thread is also responsible for logging performance information and key events to the CTM TEL (Cluster Test Manager Test Event Log).

Having completed the initialization, the main thread then creates a number of client threads according to a parameter passed to it from the harness. The main thread controls each of the client threads using a "gang scheduling" method. Prior to each phase of the test, each of the client threads reports to the main thread that it is ready. When all the client threads have reported that they are ready, they are told at the same time to start that phase of the test. The main thread then waits for each client thread to report back that it has completed that phase of the test. When all of the client threads have reported that they completed a phase of the test, the main thread examines the results from each of the client threads. If there are any errors or inconsistencies, the test is stopped. If there are no problems, the main thread gang schedules all the client threads through the next phase of the test, and this process continues until either an error is detected or the main thread is told to stop the test by the CTM test harness.

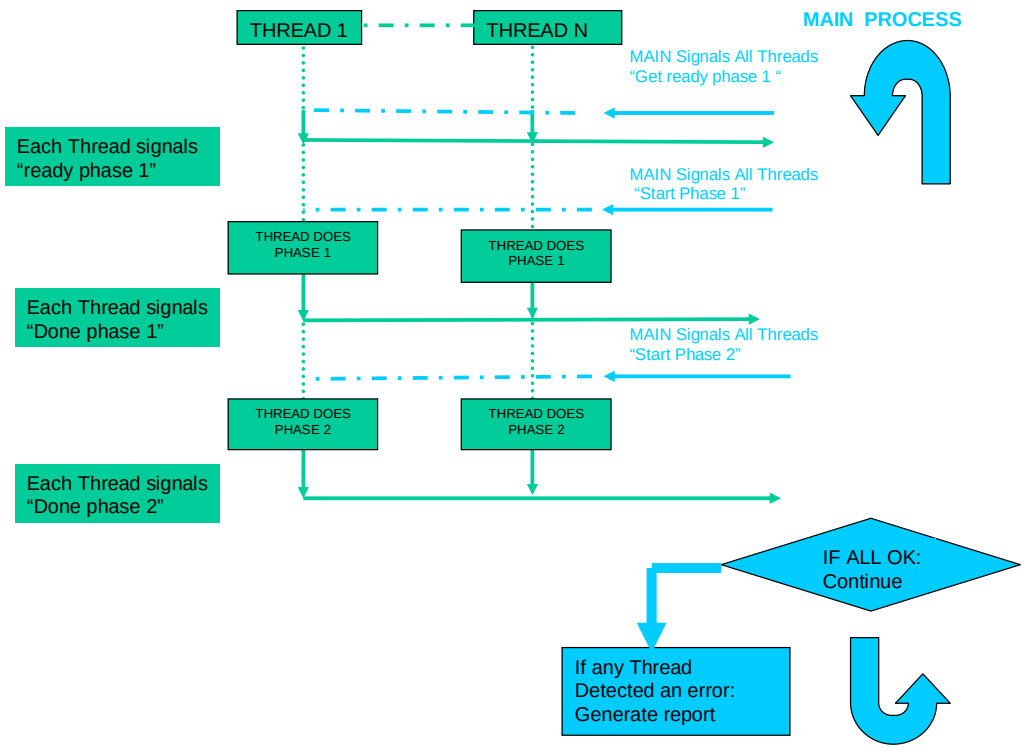

Figure 1 - Gang Scheduling

All of the threaded tests in CTM have only two phases, Phase I and Phase II, which are repeated indefinitely. The operations that are performed by each of the threads are the same, subject to specific differences, such as addresses of memory areas or names of files, which are set up for them and passed to them by the main thread. That is, in Phase 1 all the client threads do something, and in Phase II all the client threads check the results of what they did. Subject to the limitations of scheduling by the multithreaded runtime library, all the client threads do Phase 1 at the same time, and all the client threads do Phase II at the same time.

From a practical programming standpoint, this general methodology for the CTM threaded tests confers several benefits. Because the "gang scheduling – Phase 1 – Phase 2 methodology" is common to all the CTM threaded tests, a huge amount of code concerned with initializing and synchronizing the tests through these phases is common to all the tests. And because all the client threads are doing their work at the same time, we can contrive tests to make sure that memory coherency is maintained when multiple different threads are accessing the same areas of memory. In fact, all of the CTM threaded tests are designed like a framework within a framework. The tests themselves exist inside the framework of the CTM harness, and all the threaded tests are constructed inside a common "gang scheduling – Phase 1 – Phase II" methodology. In principle, a new test can be written by just changing the functionality performed in Phase 1 and Phase II. Of course, not all tests or ideas for testing lend themselves to this approach. This article only explains why and how this is done with CTM.

Existing Threaded Tests in CTM

The threaded tests in CTM were all designed and written for a particular purpose, but there is a considerable amount of commonality and even overlap between them because they were developed over time to meet testing needs as they arose. As the tests developed, we began to see how to write

better, more generalized, and more powerful tests. Commonality and overlap is not a bad thing where testing is concerned. Despite all efforts to approach testing from a logical and analytical point of view, testing in such a complex environment can never be a totally exact science. Sometimes one test will find something that might be missed or might take much longer to find with another similar test. By far, the best general testing strategy is to employ as many and as varied a repertoire of tests as possible.

The strange names for some of the tests warrant some explanation. The names do in general convey something of what the test is doing, and are certainly easier to remember than for example CTM_T001, CTM_T002 etc. etc. would be. Also, because of their common framework a lot of cutting and pasting and text substitutions were used to write them and in this context it is useful to have names that don't frequently occur in code. If nothing else then names like BOZO and YOYO definitely meet these criteria.

CTM_BYTEM (Byte Mode Memory Coherency Exerciser)

CTM_BYTEM was an early test designed to test native byte mode access on Ev6 machines, and by implication to test emulation of byte mode access on machines that do not provide native byte mode access. The test is extremely simple but serves to illustrate how the CTM threaded test structure operates.

In this test, multiple client threads target a contiguous area of memory. The principle parameters for the test are the size of the memory area and the number of threads. If there are N such client threads (thread0, thread1,…threadN), then thread 0 operates on bytes at offsets 0, N, 2N, etc., inside the memory area. Thread1 operates on bytes at offsets 1, N+1, 2N+1, etc., inside the memory area, and so on for each thread. This way, each client thread targets a unique set of bytes within the memory area.

Each thread has a unique 8-bit pattern assigned to it, which is the ones complement of its thread number. (For example, the bit pattern for thread 1 is "11111110".) Prior to Phase 1, the main thread zeros the memory area. Then, in Phase 1, each of the client threads iterates through the memory area writing the bytes it operates on to its own specific bit pattern.

In Phase II, each of the client threads reads the bit patterns in the entire memory region to ensure that they are set to the value corresponding to the client thread which operates on them

This makes sure that native byte-mode operations work OK, but a simpler test would suffice for this. The use of threads expands the nature of the test to ensuring that, in the process of modifying only a byte, the other bytes in the quad word containing the byte are not affected, and also, in a multiprocessor system, coherency is maintained when one or more threads is attempting to modify a byte in the same quad word.

Under OpenVMS Version 7.0 and later, a two-level scheduling model is employed. Threads are scheduled to run on virtual processors the same way that OpenVMS schedules virtual processors to run on physical processors. Most thread scheduling takes place in user mode, which is much more efficient than switching contexts, which involves the operating system. The OpenVMS scheduler can schedule virtual processes onto separate processors of a multiprocessor machine. An upcall mechanism handles the cases where OpenVMS detects an event that affects the scheduling of a thread, calling up to the threads scheduler to notify it of the change in the status of a thread.

In the case of this test, if the system under test has N processors, then the recommended value of the parameter for the number of client threads is also N. If only the client threads are running, very simple arithmetic is required to figure out how often, on average, they try to modify a byte in the same quad word. Considering the number of processors, then adjusting the size of the memory area and the number of client threads provides a powerful test mechanism that we can use to force this situation to occur frequently.

This test structure also allows us to verify that the test is working as expected and that clients threads are in fact trying to access the same quad word with the expected frequency. The tests are written in C. Normally, the memory area being accessed by the threads would be compiled with the volatile qualifier to ensure that the memory was protected between the client threads. If the volatile qualifier is

removed, then a number of corruptions should correspond to the number of times that two client threads were accessing a byte in the same quad word.

CTM_BOZO

CTM_BOZO uses threads to get as much testing work done as possible on a multiprocessor system. The purpose of the test is to have many threads performing memory tests (allocating and deallocating memory) and/or performing file I/O operations. Because the threads work in the context of a single process, they work more quickly than the corresponding number of processes; this allows them to test more allocation/deallocation sequences in the same amount of time.

As with the other threaded tests, the main thread communicates with the test harness, logs performance information, initializes data buffers and patterns, creates the client threads, and controls them using gang scheduling.

In Phase I, when the main thread instructs the client threads to start, they optionally allocate memory, write a data pattern to the memory and/or open a data file and write a data pattern to the file.

In Phase II, each of the client threads tests the results of the operations it performed in Phase I, either by verifying the data pattern in memory, and/or reading and verifying the data that was written to file. In Phase II, the client threads release the resources they were using in Phase I, thus assuring that resources are allocated and deallocated with each test iteration.

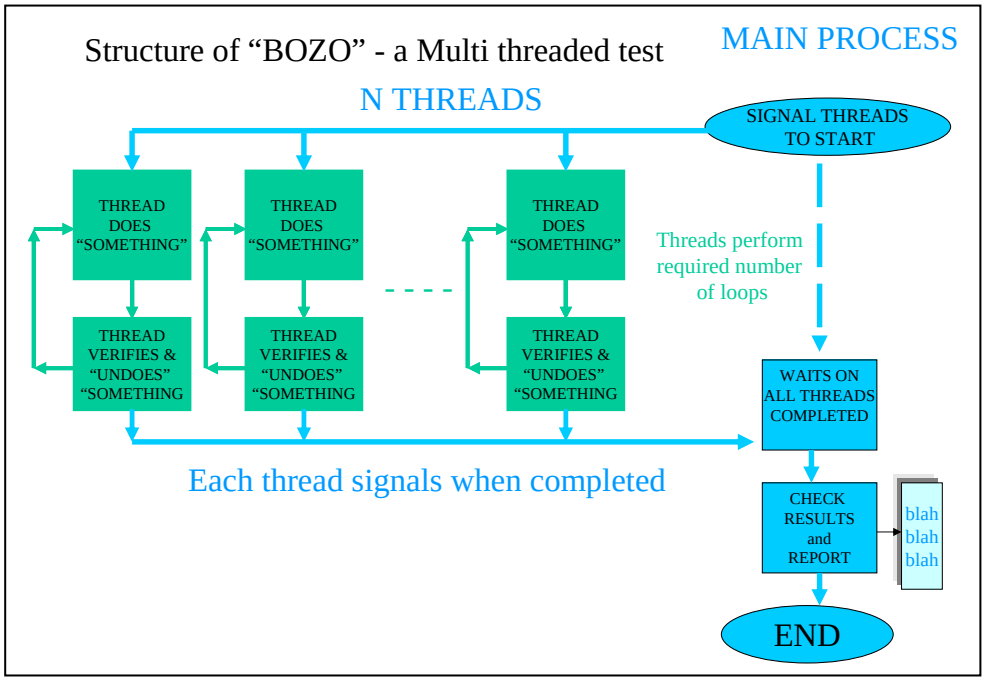

Figure 2 - Structure of a Multithreaded Test

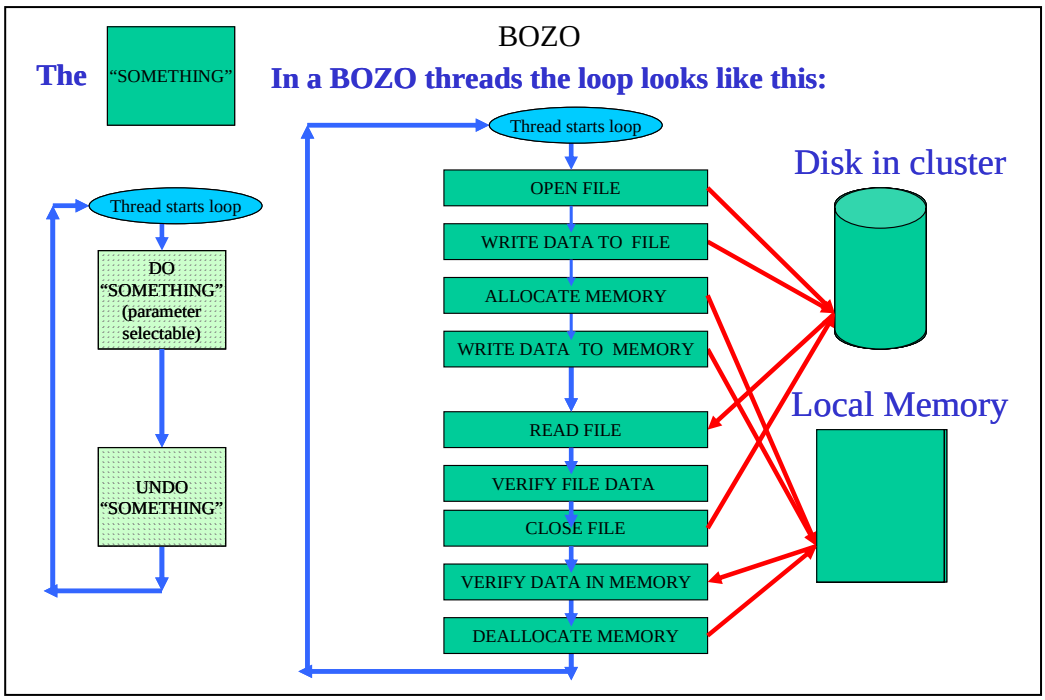

Figure 3 – The Loop

To permit the maximum utilization of the system, the SYNCH parameter specifies how many times the client thread should repeat the operations of Phase I and Phase II before it reports to the main thread that has completed.

The test parameters can specify any permutation of memory task to be performed (such as I/O only, memory only, I/O and memory, or neither I/O nor memory). The case where neither I/O nor memory operations are performed effectively tests the multithreading runtime library by causing a very large number of thread creates, joins, and terminations.

CTM_YOYO

CTM_YOYO is a threaded tool that was designed to stress threaded sequences and, in particular, to stress the thread creation, thread joins, thread termination, and upcall functionality.

The basic architecture of the test is the same as described under the General Methodology section above, except that the main thread, instead of just controlling a vector of client threads, actually controls a matrix of client threads. The main thread creates and synchronizes with the top level thread of what can be visualized as a series of columns of threads. The test uses recursive logic to test thread sequences. The top-level thread causes a series of threads to be created recursively underneath it. The number of columns of threads and the depth of recursion are specified by parameters.

As with the other threaded tests, the main thread communicates with the test harness, logs performance information, initializes data buffer and patterns, creates client threads, and controls them by gang scheduling. For speed and efficiency, the main thread creates all the data patterns for each of the client threads and the threads that they create, before the test starts.

The basic test unit in CTM_YOYO is a thread, and other than the top level thread (which communicates with the harness and reports back the results of all the threads created underneath it) and the lowest level thread which does not create a thread underneath it, the threads in CTM_YOYO operate as follows:

1. Thread starts a task that involves allocating resources.
2. Thread then recursively calls down to a copy of itself.
3. Thread waits for the completion of the recursive call.
4. .....

5. Recursive call completes.
6. Thread finishes the task it was doing.
7. Thread checks the results of what it did.
8. Thread releases the resources it was using.
9. Thread completes.

The name CTM_YOYO derives from the fact that the threads "yoyo" up and down through recursive calls. This is a very stressful on the mechanisms that are used to schedule the threads. It tests the up-calls and causes rapid context switching on the processors on which the threads are executing.

As mentioned above, a CTM_YOYO test is characterized by "WIDTH" and "DEPTH," which are test parameters. The depth is the level that the threads go down to recursively. The width is the number of sequences of threads that are simultaneously yoyoing up and down. Effectively, a matrix of thread execution sequences is involved in a test, each characterized by (X,Y), where X is the width position of the thread within the matrix, and Y is the depth position of the thread within the matrix. The matrix of threads is controlled by a main program, which interfaces to CTM, schedules and synchronizes the threads within the matrix, tests the results they report back, and does the TEL logging of performance information.

Again, the tasks that the thread sequences perform are related to allocating and testing memory regions, and to simple I/O operations, and again any permutation of memory task and I/O task may be specified (such as I/O only, memory only, I/O and memory, or neither I/O nor memory). The case where no memory testing and no I/O testing is specified is not only valid but particularly useful for this test as it causes rapid and stressful sequence of thread creation, termination, and synchronization to occur.

Specifically, if memory is to be tested by CTM_YOYO then each thread will do the following:-

1. Allocate a memory region of the specified number of pages (8192 bytes).
2. Write a data pattern into this memory region.
3. Call down recursively and wait for completion on the recursive call.
4. .....
5. Read back and verify the data pattern from the memory region.
6. De-allocate the memory region.

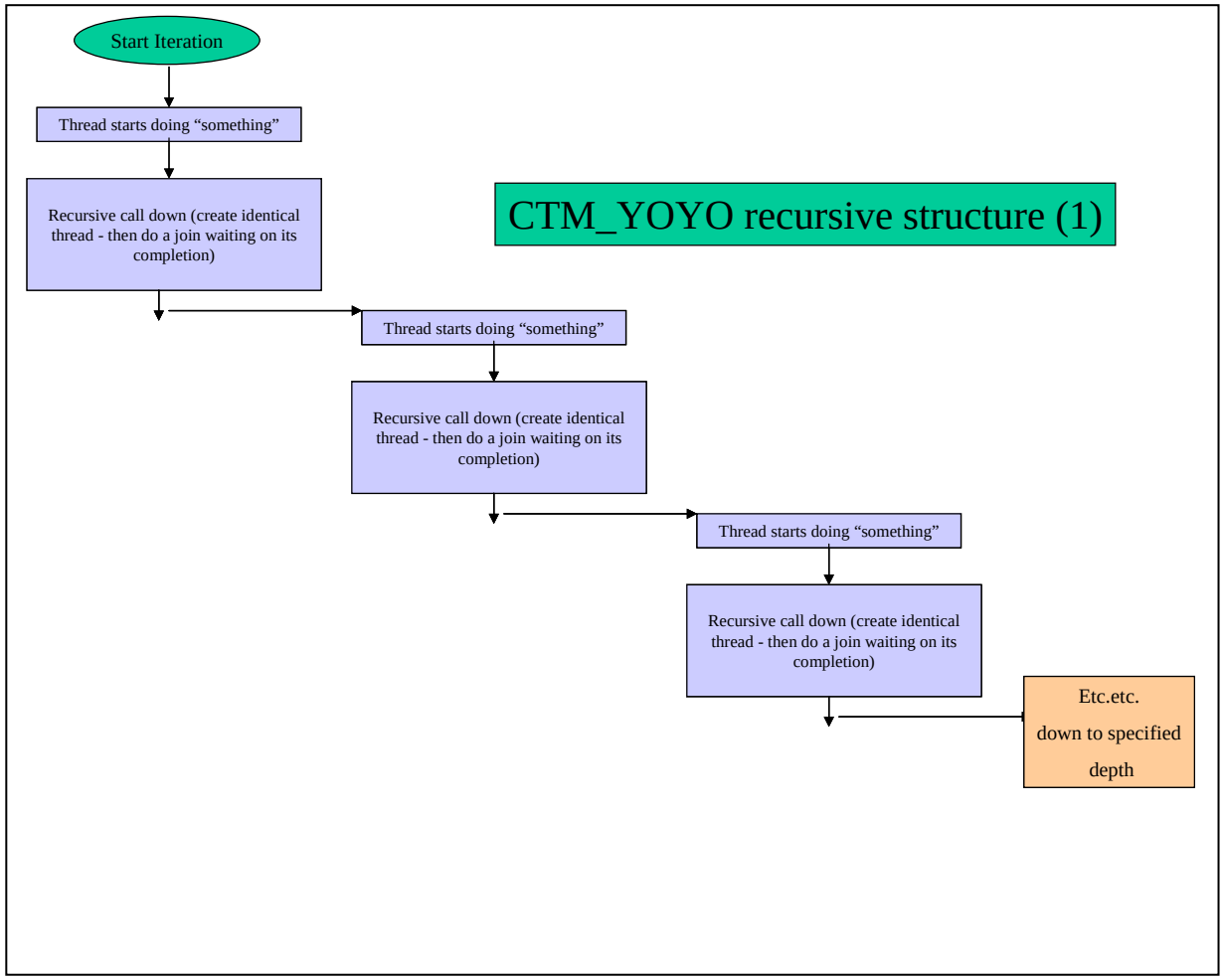

Figure 4 – The Recursive Structure (1)

If I/O is to be tested by CTM_YOYO each thread will do the following:-

1. Create and open a data file.
2. Write a data pattern of the specified number of blocks (512 bytes) to this data file.
3. Call down recursively and wait for completion on the recursive call.
4. .....
5. Read back and verify the data pattern from the data file.
6. Delete the data file.

For a large test, there may be 16 or more columns of threads, each recursing down to a depth of 16, which potentially generates a huge amount of data. Tracing, logging, and generally interpreting this amount of data when a problem is detected poses quite a challenge. When a thread detects a problem or a corruption, it generates its own corruption report, numbered and identified according to its position in the matrix, before it terminates. In this case, it does not call down to the client thread below it but sets a status to indicate to the thread that created it that it failed. All failures are bubbled up to the top level thread, which reports the failure to the main thread. The main thread puts a summary of the failure into the test log and then terminates the test.

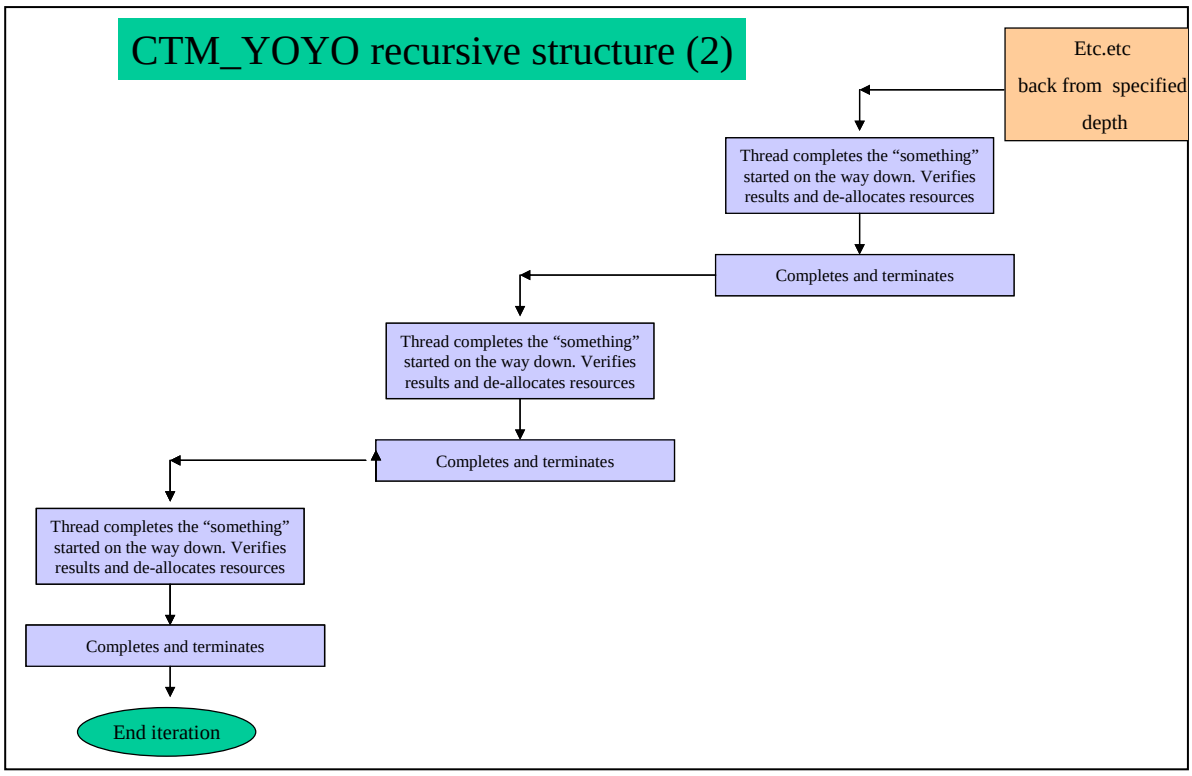

Figure 5 –The Recursive Structure (2)

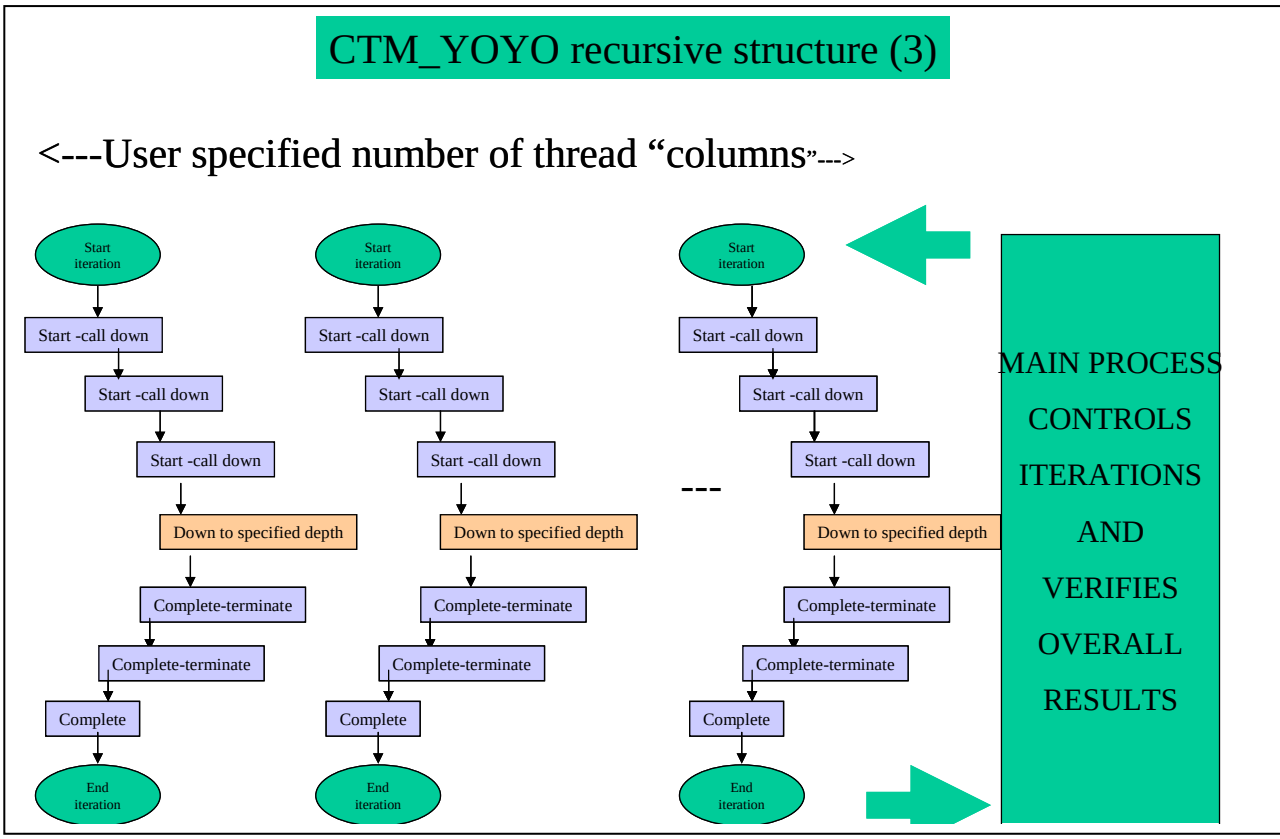

Figure 6 – The Recursive Structure (3)

CTM_LKNLD

The CTM_LKNLD test was specifically designed to test the load_lock/store_conditional sequence on multiprocessor Alpha systems. The test is designed for that Alpha architectures. This sequence was chosen for testing because of its importance for maintaining memory coherency on a multiprocessor system, and because rigorous testing in this area by implication tests other mechanisms that are involved in maintaining coherency on Alpha architectures.

To provide some context, a processor modifies the contents of physical memory by means of a LOAD/modify/STORE sequence. However in an SMP environment, where more than one processor has access to the memory location, this simple sequence will not suffice. For example, if there are two processors (A and B,) if A LOADS the memory location B LOADS the memory location, then B modifies the memory location and does a STORE, then A modifies the memory location and does a STORE, B's view of memory is now incorrect and something is going to get broken. The load locked/store conditional sequence provides a method whereby multiple CPUs can synchronize access.

The load locked/store conditional logic is the same for whatever granularity of memory is being accessed. CTM_LKNLD is based on quad word access (LDQ_L/STQ_C sequences). Roughly speaking, when a processor loads the memory with an LDQ_L instruction, it records the target address in a per-processor locked physical address register and sets a per-processor lock flag. If the per-processor lock flag is still set when it gets around to doing the store with a STQ_C instruction, then the store occurs; otherwise, the store does not occur. Whenever a processor successfully completes a store to within the locked address range of another processor, it clears the lock flag for that processor, causing that processor's subsequent store conditional to fail.

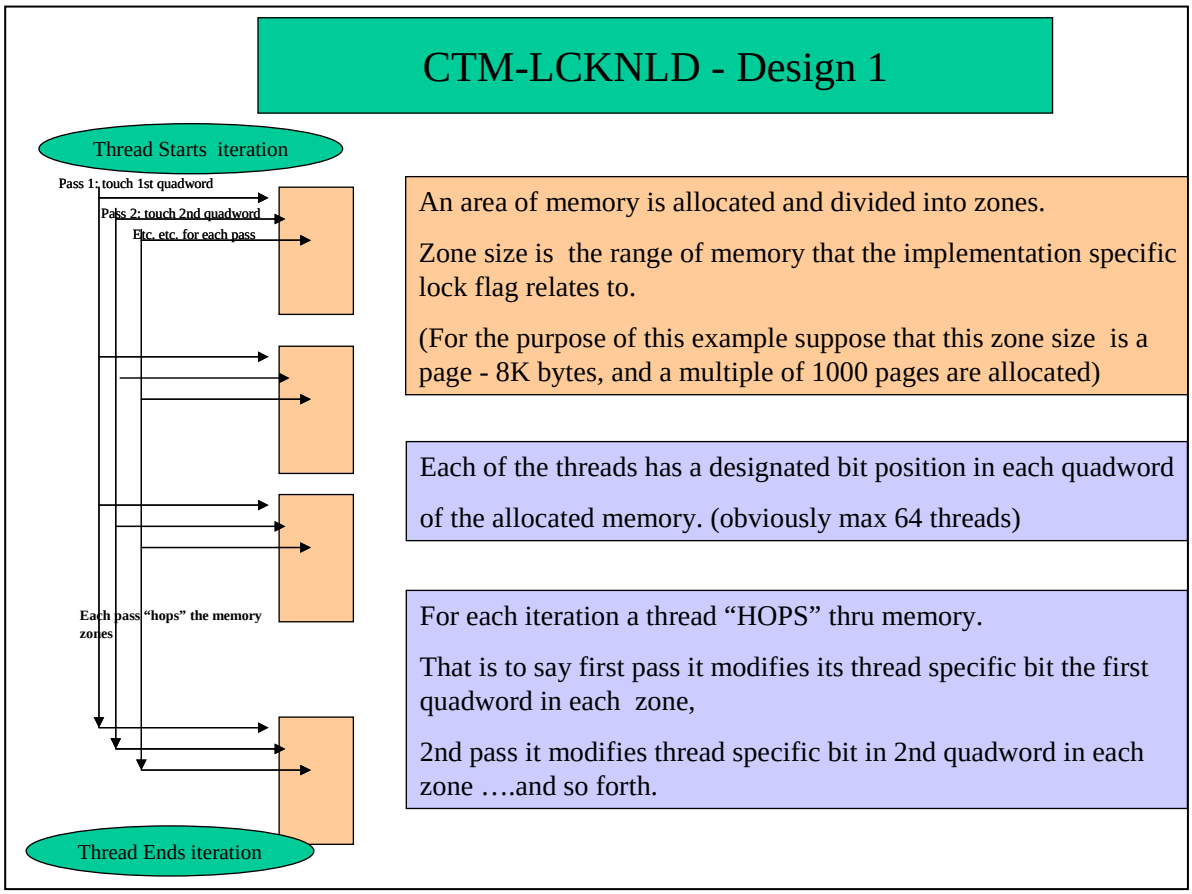

Figure 7 – CTM_LKNLD (1)

To use the example of processors A and B above:-

1. A does LDQ_L on address pa succeeds
2. B does LDQ_L on address pa succeeds.
3. A does STQ_C on address pa succeeds - clears B's lock flag.
4. B does STQ_C on address pa fails because B's lock flag is cleared.
5. B should now re-load pa and retry the sequence.

The basis of the test performed by CTM_LKNLD is similar to that used by CTM_BYTEM. The basic architecture is the same, with a main thread controlling a number of client threads by gang scheduling them through Phase 1 and Phase II. The result is a user-specified number of client threads concurrently looping through the same regions of memory. Each thread tries to modify a byte in a quad word with a data pattern specific to the thread. The threads access the quad words using LDQ_L/STQ_C sequences to access and store the modified quad words.

The data pattern that a thread writes to a byte within the quad word is the ones complement of the thread number. Hence, thread 0 writes binary 11111111, thread 1 writes binary 11111110, and so forth. All the memory regions are set to zero before a test iteration, so if a value of 0 is subsequently detected, it indicates that a thread did not write the pattern correctly.

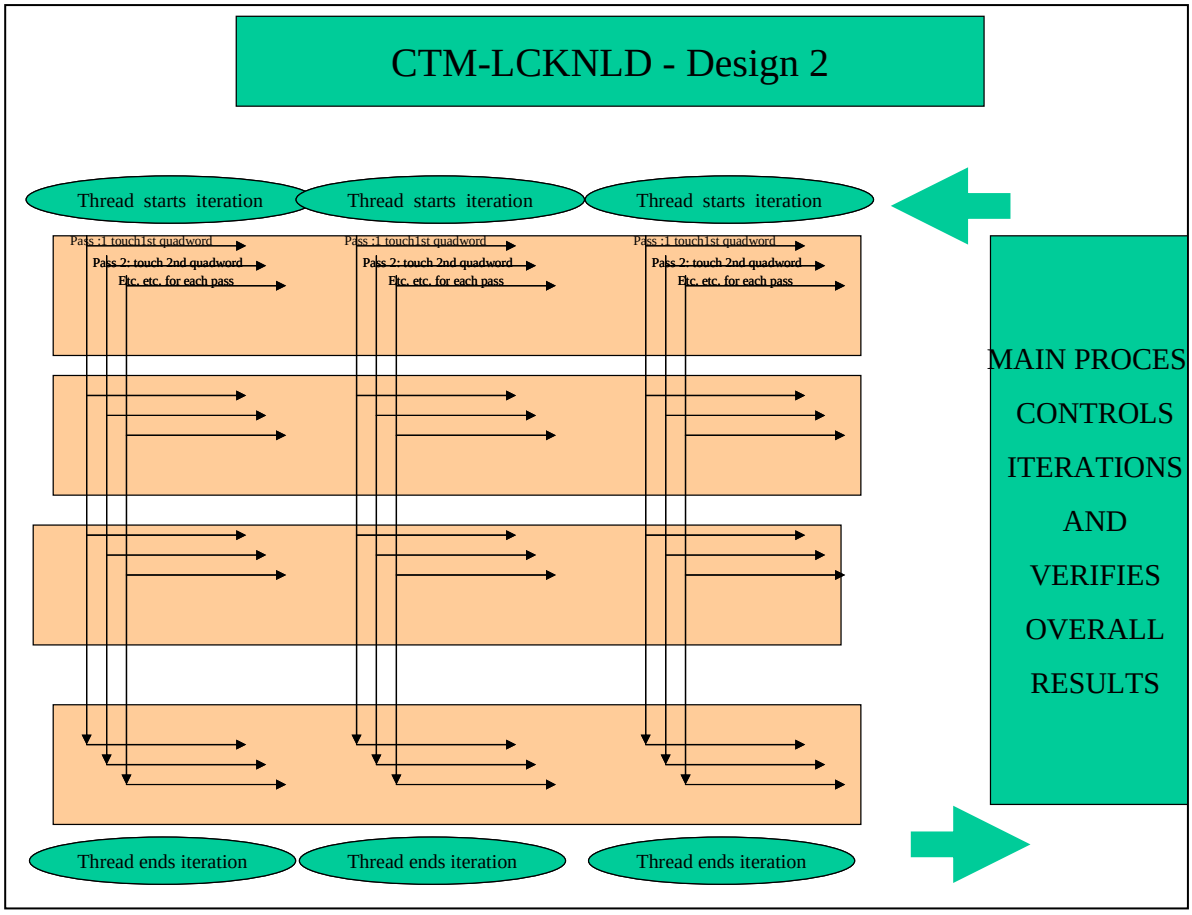

Figure 8 – CTM-LCKNLD (2)

Of course, the limits imposed by OpenVMS and the multithreaded runtime library make it impossible to control which thread will run on which CPU directly; however, it is a statistical certainty that there will be many times when different threads are simultaneously trying to access bytes in the same quad word, hence the LDQ_L /STQ_C sequence has to work correctly for the data patterns to be set correctly. By deliberately building the test to use LDQ_L/STQ_C sequences instead of regular Load and Store sequence, the test verifies that the number of reported corruptions corresponds to the

approximate number of times two client threads try to access the same quad word. In the case of this test, which is significantly more complex, this technique was invaluable for ensuring the test was doing what was intended.

For historical reasons, the CTM_LKNLD test was extended beyond the basic test described above, by varying the conditions when the STQ_C occurs. For each thread a varying number of memory accesses are interposed between the LDQ_L and the STQ_C, so that when the STQ_C occurs, the test causes the TB (Translation Buffer) entry to correspond to the target address in different positions in the TB. This is achieved by having the threads hop through memory as follows:

1. First memory location in first memory region
2. First memory location in second memory region
3. ...
4. First memory location in last memory region
5. Second memory location in first memory region
6. Second memory location in first memory region
7. ...
8. Second memory location in last memory region
9. ...
10. ...
11. Last memory location in last memory region

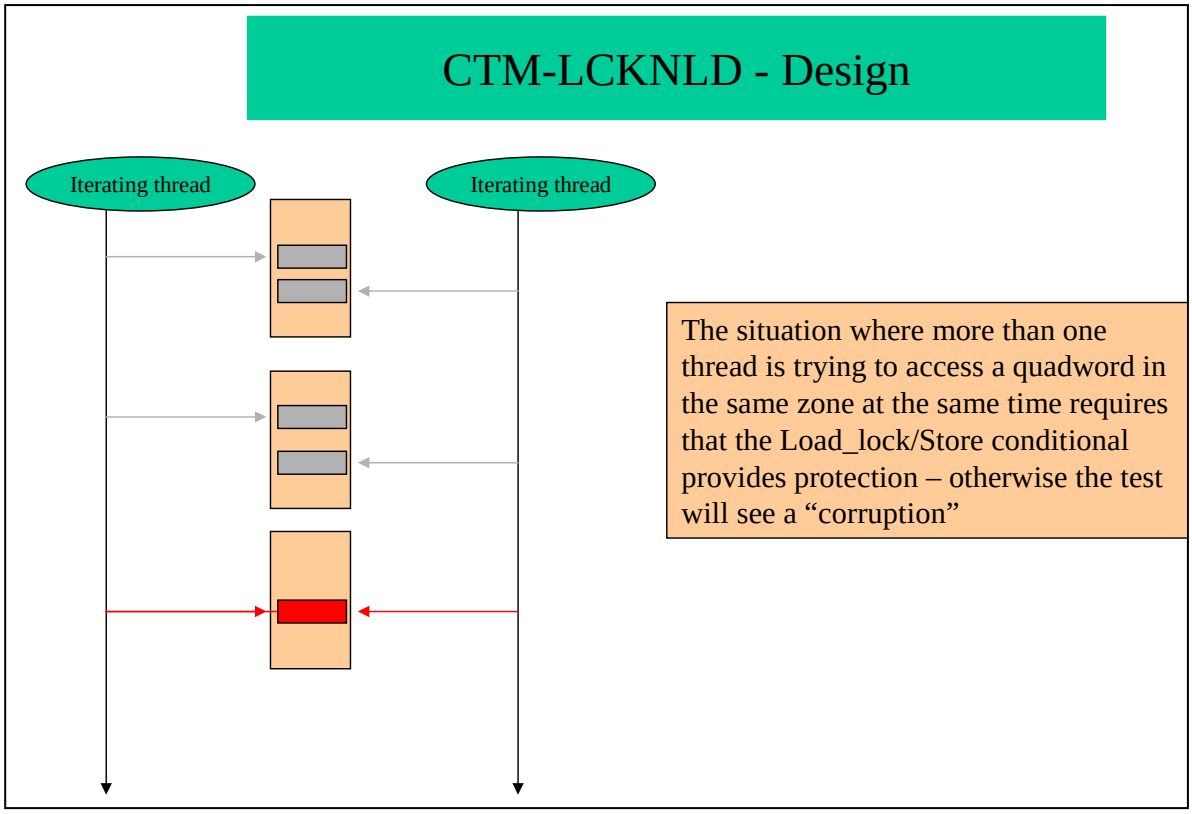

Figure 9 – CTM LCKNLD (3)

Two sets of memory regions are accessed alternately by the threads. For convenience, these are referred to as U regions and V regions. There are 128 U regions and the same number of V regions. Each memory region is the system-specific page size (currently set to 8192 bytes).

Each hop accesses a new page and results in a new entry in the TB. Having initiated an LDQ_L/STQ_C sequence with an LDQ_L, the thread then is then forced to access a varying number of memory locations in other memory regions before it completes the STQ_C. These are accessed with regular LDQ instructions; we consider them "noise," interposed between the LDQ_L and the STQ_C.

Because these noise memory accesses are to different pages, they have the effect of causing a new entry into the TB, and hence cause the TB entry for the target to be in a varying position in the TB when the STQ_C occurs. The number of noise accesses that are forced in this way varies from 0 to 128 entries plus one, with each iteration, so that eventually the thread completes the STQ_C with the target address entry in a variety of different positions within the TB.

On alternate iterations, each thread alternates as follows:

1. Uses U regions for LDQ_L/STQ_C (target) access.
2. Uses V regions for LDQ (noise) access to vary the position of the focus access in the TB.
3. Uses V regions for LDQ_L/STQ_C (target) access.
4. Uses U regions for LDQ (noise) access to vary the position of the focus access in the TB.

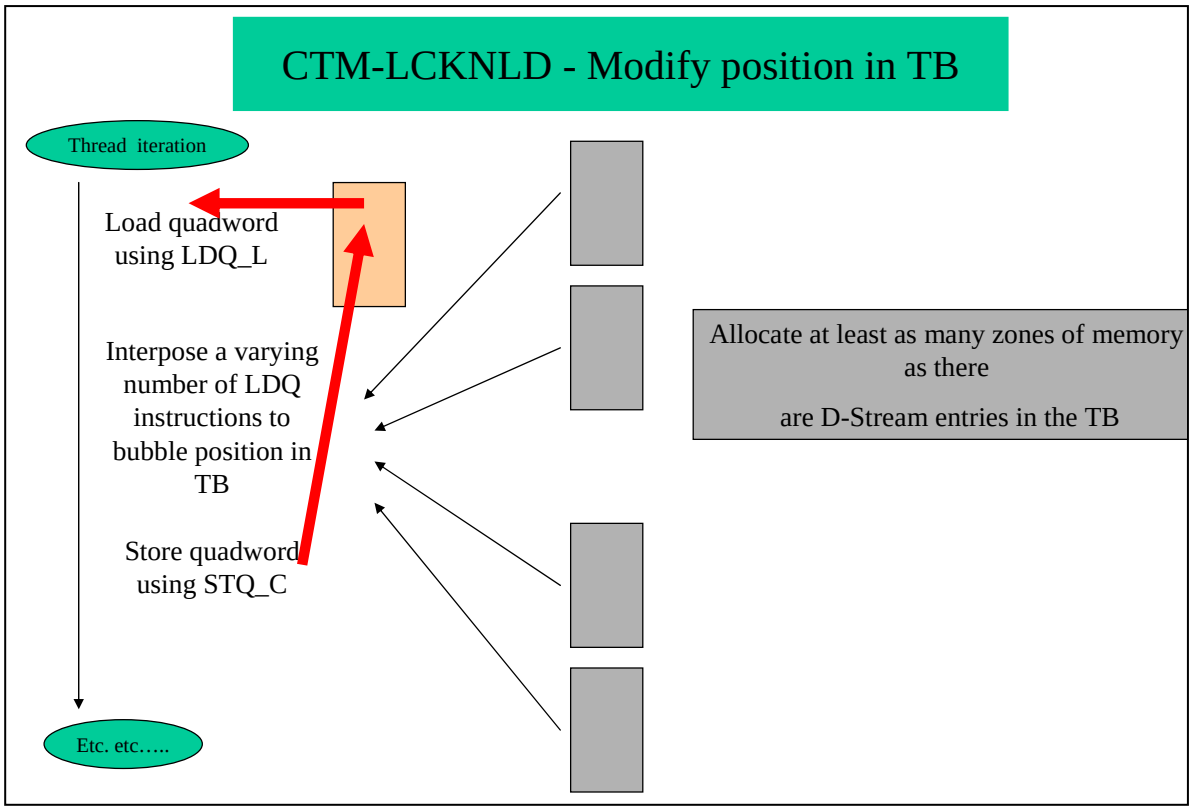

Figure 10 – CTM_LCKNLD (4)

CTM_FAST_IO

CTM_FAST_IO was written to test FAST_IO functionality, a new feature in OpenVMS Version 7.0 aimed at dramatically improving I/O performance.

Fast I/O is for systems and applications that require highly optimized paths for high volume I/O, such as large database systems. Fast I/O comprises a set of system services that provide an alternative to QIOs. The greatest benefit of Fast I/O is that it locks and maps user buffers and user I/O status areas permanently in system space. For direct I/O, this avoids per-I/O buffer lockdown and unlocking; for buffered I/O, it avoids allocation and deallocation of separate system buffers, permitting complete Fast I/O operations at IPL 8 and reducing spinlock acquisitions.

Threads are not a particularly important aspect of the FAST_IO test, other than providing the capability for improved performance on multiprocessor systems. A description of the CTM_FAST_IO test is provided here for completeness.

As with the other CTM threaded tests, the CTM_FAST_IO test comprises a main thread, which controls a user-specified number of client threads Two write buffers are used by each client thread: a pattern0 buffer and a pattern1 buffer. In addition to all the usual work, the main thread presets data patterns into the write buffers. Data pattern1 is the ones complement of the pattern0 buffer. Pattern0 and Pattern1 are alternately written to and read from a data file by the threads, using Fast I/O operations. Because the patterns are complements, the data integrity of the I/O operations can be checked without the need to poison buffers between reads and writes. The main process allocates read and write buffers for each of the threads. The size of these data buffers is the size of the actual data files that will be used by the threads, and is determined by the BLOCKS parameter (expressed in 512-byte blocks). Prior to starting the threads, the main process locks down all the read and write buffers that the threads will use in buffer objects, using the SYC$CREATE_BUFOBJ_64 system service call. The buffer objects remain locked down for use by the threads for the entire duration of the test.

Having prepared and locked down the buffers that the threads will use, the main process then creates and synchronizes the client threads through the actual test, which conforms to the Phase I/Phase II structure of all the CTM threaded tests.

A thread performs each test iteration as follows.

In Phase I, the thread:

1. Deletes any previous versions of its data file.
2. Creates and opens a new data file using RMS.
3. Performs the setup for each of the subsequent file operations using the SYS$IO_SETUP system service.
4. Loops a user-specified number of times (specified in the SYNCH parameter), by doing the following Fast I/O operations on the data file.
5. Writes pattern0 data to the data file using the SYS$IO_PERFORMW system service.
6. Reads the data back from the data file using the SYS$IO_PERFORMW system service.
7. Verifies the data read back against the data written.

In Phase II, the thread:

1. Writes pattern1 data to the data file using the SYS$IO_PERFORMW system service.
2. Reads the data back from the data file using the SYS$IO_PERFORMW system service.
3. Verifies the data read back against the data written.
4. Closes its data file, and cleans up the Fast I/O operations using the SYS$IO_CLEANUP system service.
5. Reports the results (success or failure of the data verification) back to the main process, and waits to be started again by the main process.

Conclusion

This article presents only one basic design for threaded tests, namely multiple client threads gang scheduled by a main thread, with the actual test functionality performed by the client threads and formally divided into two separate phases (usually the test operation phase and the verification phase). Of course, such tests can be organized in many ways, and there is no intent to suggest this is the best way to do it. However, this approach has worked well for us in terms of reducing the effort required to write such tests and so far has not presented any limitations for the tests we have needed to write. To be of use, tests for the operating system need to be very solid and reliable, and the results of the testing must be readily understandable so that problems can be debugged. Writing tests inside this framework has definitely helped us meet these objectives.

Threaded tests provide an invaluable means for testing on multiprocessor systems, not only in so far as they improve the performance of the test and permit more work to be done in a given time, but, more importantly, they provide the means to directly test that the required functionality works properly when functionality is distributed across multiple threads running concurrently on separate processors. Properly constructed, such tests also provide the means to subject the systems under test to extreme stress and to manipulate the systems so that situations that could potentially cause problems are created much more frequently than would otherwise be the case. There is no doubt that such tests will become increasingly important in ensuring that the very high level of reliability that OpenVMS is famous for will continue into the future.