Project 3-Memory Ordering Demonstration in Multicore Shared Environments and Implementation of Sequential consistency and Total Store Order

Project Description: Modern computer systems, particularly multicore chips or chip multiprocessors, incorporate hardware support for shared memory. In shared memory systems, each processor core has the capability to both read from and write to a single, shared address space. The concept of consistency, often referred to as the memory consistency model, plays a pivotal role in establishing the correctness of shared memory interactions. Consistency definitions lay down rules governing how memory operations, such as reads and writes, interact with the memory itself.

Consistency models serve as frameworks for determining the proper behavior of shared memory within a system, focusing exclusively on memory operations like reads and writes, independent of considerations about caches or coherence mechanisms. These models delineate the acceptable conduct of multithreaded programs running on shared memory. For a given multithreaded program operating with specific input data, the memory model outlines the potential values that dynamic loads can produce and the range of possible end states for the memory. Unlike the more straightforward scenario of single-threaded execution, the presence of multiple correct behaviors introduces complexity to the understanding of memory consistency models.

These memory consistency models, often referred to as memory models, perform the crucial role of defining how shared memory systems should function for both programmers and system implementors. By establishing criteria for correctness, these models provide programmers with expectations regarding the behavior of their code, and they offer implementors guidance on the necessary provisions to ensure proper system operation.

To see why shared memory behavior must be defined, consider the example execution of two cores depicted in Table 3.1 (assuming that the initial values of all variables are zero). Most programmers would expect that core C2's register r2 should get the value NEW. Nevertheless, r2 can be 0 in some of today's computer systems. Hardware can make r2 get the value 0 by reordering core C1's stores S1 and S2. Locally (i.e. if we look only at C1's execution and do not consider interactions with other threads), this reordering seems correct because S1 and S2 access different addresses. With the reordering of S1 and S2, the execution order may be S2, L1, L2, S1, as illustrated in Table 3.2.

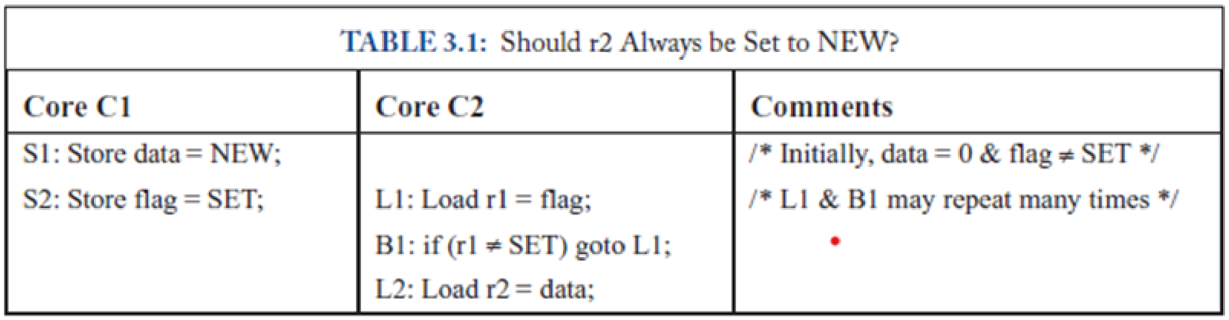
TABLE 3.1: Should r2 Always be Set to NEW?

Core C1	Core C2	Comments
S1: Store data = NEW;		/* Initially, data = 0 & flag ≠ SET */
S2: Store flag = SET;	L1: Load r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	L2: Load r2 = data;	

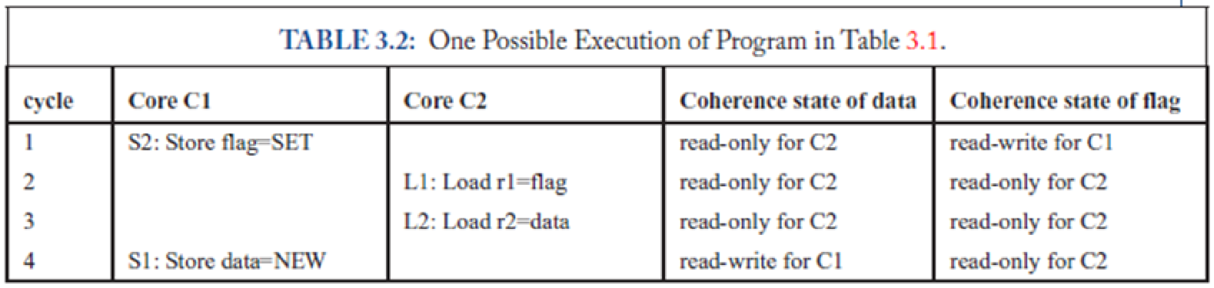
TABLE 3.2: One Possible Execution of Program in Table 3.1.

cycle	Core C1	Core C2	Coherence state of data	Coherence state of flag
1	S2: Store flag=SET		read-only for C2	read-write for C1
2		L1: Load r1=flag	read-only for C2	read-only for C2
3		L2: Load r2=data	read-only for C2	read-only for C2
4	S1: Store data=NEW		read-write for C1	read-only for C2

How a Core Might Reorder Memory Accesses:

Store-store reordering. Two stores may be reordered if a core has a non-FIFO write buffer that lets stores depart in a different order than the order in which they entered. This might occur if the first store misses in the cache while the second hits or if the second store can coalesce with an earlier store (i.e., before the first store). Note that these reordering are possible even if the core executes all instructions in program order. Reordering stores to different memory addresses has no effect on a single-threaded execution. However, in the multithreaded example of Table 3.1, reordering Core C1's stores allow Core C2 to see the flag as SET before it sees the store-to-data.

Load-load reordering. Modern dynamically scheduled cores may execute instructions out of program order. In the example of Table 3.1, Core C2 could execute loads L1 and L2 out of order. Considering only a single-threaded execution, this reordering seems safe because L1 and L2 are to different addresses. However, reordering Core C2's loads behave the same as reordering Core C1's stores; if the memory references execute in the order L2, S1, S2 and L1, then r2 is assigned 0. This scenario is even more plausible if the branch statement B1 is elided, so no control dependence separates L1 and L2.
Load-store and store-load reordering. Out-of-order cores may also reorder loads and stores (to different addresses) from the same thread. Reordering an earlier load with a later store (a load-store reordering) can cause many incorrect behaviors.
A memory consistency model, or, more simply, a memory model, is a specification of the allowed behavior of multithreaded programs executing with shared memory. For a multithreaded program executing with specific input data, it specifies what values dynamic loads may return and what the final state of memory is.

SEQUENTIAL CONSISTENCY (SC)

The most intuitive memory consistency model is sequential consistency (SC). Lamport first formalized sequential consistency. Lamport first called a single processor (core) sequential if "the result of an execution is the same as if the operations had been executed in the order specified by the program." He then called a multiprocessor sequentially consistent if "the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program." This total order of operations is called memory order. In SC, memory order respects each core's program order, but other consistency models may permit memory orders that do not always respect the program orders.

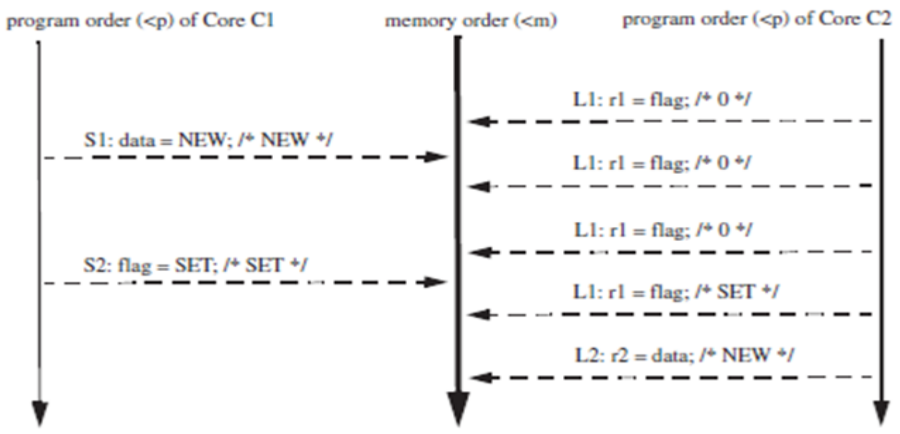

FIGURE 3.1: A Sequentially Consistent Execution of Table 3.1's Program.

Figure 3.1 depicts an execution of the example program from Table 3.1. The middle vertical downward arrow represents the memory order (<m) while each core's downward arrow represents its program order (<p). We denote memory order using the operator <m, so op1 <m op2 implies that op1 precedes op2 in memory order. Similarly, we use the operator <p to denote program order for a given core, so op1 <p op2 implies that op1 precedes op2 in that core's program order. Under SC, memory order respects each core's program order. "Respects" means that op1 <p op2 implies op1 <m op2. The values in comments (/* ... */) give the value loaded or stored. This execution terminates with r2 being NEW. More generally, all executions of Table 3.1's program terminate with r2 as NEW. This example illustrates the value of SC.

L(a) and S(a) represent a load and a store, respectively, to address a. Orders <p and <m define the program and global memory order, respectively. Program order <p is a per-core total order that captures the order in which each core logically (sequentially) executes memory operations. Global memory order <m is a total order on the memory operations of all cores.
An SC execution requires:

(1) All cores insert their loads and stores into the order <m respecting their program order, regardless of whether they are to the same or different addresses (i.e., a=b or a≠b). There are four cases:
If L(a) <p L(b) ⇒ L(a) <m L(b) /* Load→Load */
If L(a) <p S(b) ⇒ L(a) <m S(b) /* Load→Store */
If S(a) <p S(b) ⇒ S(a) <m S(b) /* Store→Store */
If S(a) <p L(b) ⇒ S(a) <m L(b) /* Store→Load */
(2) Every load gets its value from the last store before it (in global memory order) to the same address: Value of L(a) = Value of MAX <m {S(a) | S(a) <m L(a)}, where MAX <m denotes "latest in memory order."

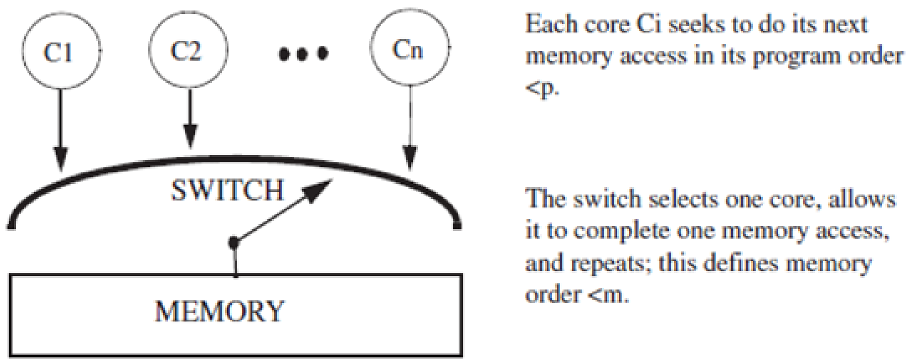
C1
C2
•••
Cn
SWITCH
MEMORY
Each core Ci seeks to do its next memory access in its program order <p.
The switch selects one core, allows it to complete one memory access, and repeats; this defines memory order <m.

FIGURE 3.2: A simple SC implementation using a memory switch

You can implement SC with a set of cores Ci, a single switch, and memory, as depicted in Figure 3.2. Assume that each core presents memory operations to the switch one at a time in its program order. Each core can use any optimizations that do not affect the order in which it presents memory operations to the switch. For example, a simple 5-stage in-order pipeline with branch prediction can be used.

Assume next that the switch picks one core, allows memory to satisfy the load or store fully, and repeats this process as long as requests exist. The switch may pick cores by any method (e.g.,random) that does not starve a core with a ready request. This implementation operationally implements SC by construction.

- Load → Load
- Load → Store
- Store → Store
- Store → Load /* Included for SC but omitted for TSO */

Total Store Order(TSO) includes the first three constraints but not the fourth. Detail definition of TSO can be found in [1] chapter 4.

OBJECTIVE:

1. Your project is to demonstrate the list of reordering mentioned above using a multithread operation (using 2 thread) in any programming language like (C , C++ etc) using a memory litmus test. You must design a separate memory litmus test for each reordering case.
2. You must draw the execution of the litmus tests.
3. Using proper synchronization primitive (e.g. lock, semaphores), you must demonstrate the implementation of sequential consistency and total store order(TSO)

Mid-Project Evaluation (MPE):

1. Demonstration of the reordering using memory litmus test:
    a. Store-store reordering.
    b. Load-load reordering
    c. Load-store and store load reordering.
2. Draw the execution of the litmus tests provide by you.

End-Project Evaluation (EPE):

Please ensure EPE completes the following:

1. Whatever you cannot implement in MPE has to be completed before EPE.
2. Implementation of naïve SC models to avoid reordering in the three litmus test.

3. Implementation of Total Store Order (TSO) memory model

In summary, this project aims to demonstrate memory ordering behaviors in a shared multicore environment practically. By designing memory litmus tests and implementing sequential consistency and TSO models, you will gain insight into memory consistency challenges and solutions. Your mid-project and end-project evaluations will assess the completeness and effectiveness of your demonstrations and implementations.

Further detail of memory consistency can be found in:

[1]A Primer on Memory Consistency and Cache Coherence, Second Edition