INDEX

1. Memory consistency models

These are formal frameworks used to define the expected behavior of memory operations in a shared-memory system. These models specify the order in which memory operations (loads and stores) must appear to execute to ensure predictable and correct program behavior in multiprocessor environments.

1.1. Types of Approaches

1. **Axiomatic Models**
2. **Operational Models**

2. Axiomatic Models

Definition: Axiomatic models describe memory consistency through a set of formal rules (axioms) that must be satisfied by any valid execution of a program. They specify constraints on the order of memory operations.

Purpose:

To define the allowable behaviors of memory operations in a formal and mathematical way.

To provide a high-level, abstract way of reasoning about memory consistency.

2.1 Example Models

Sequential Consistency (SC)

Axiom: All operations appear to execute in a single, global order that is consistent with the program order on each processor.

Example:

```
Processor 1:      Processor 2:
x = 1;            y = 1;
r1 = y;           r2 = x;
```

Possible outcomes for (r1, r2) under SC: (0,1), (1,0), or (1,1).
(0,0) is not possible because it implies that both writes happened after both reads, which violates the global order.

SC requires that all processors see a single, consistent global order of operations. Results like (1,0) and (0,1) are not valid under SC because they imply conflicting global orders where one processor's reads and writes do not align consistently with another processor's operations. Only (1,1) maintains a consistent global order where all operations are observed in a way that respects the program order on each processor.

Why (1,0) **and** (0,1) **Are Not Valid Under SC**

1. Scenario Analysis

Consider the operations for each processor:

Processor 1: x = 1; r1 = y;

Processor 2: y = 1; r2 = x;

Outcome (1,0)

Meaning: Processor 1 reads y = 1 (after y = 1 is written), and Processor 2 reads x = 0 (before x = 1 is written).

Issues with (1,0) **under SC:**

Program Order for Processor 1: Processor 1's read of y is consistent with seeing y = 1 written before the read. However, for Processor 2, reading x = 0 implies that Processor 1's write to x was not observed by Processor 2's read. This situation requires that Processor 1's write to x happened after Processor 2's read of x, which would conflict with SC's requirement for a single global order where all processors see operations in a consistent sequence.

Outcome (0,1)

Meaning: Processor 1 reads y = 0 (before y = 1 is written), and Processor 2 reads x = 1 (after x = 1 is written).

Issues with (0,1) **under SC:**

Program Order for Processor 2: Processor 2's read of x is consistent with seeing x = 1 written before the read. However, for Processor 1, reading y = 0 implies that Processor 2's write to y was not seen, which would mean that Processor 1's read of y happened before Processor 2's write to y was completed. This inconsistency indicates that the operations did not follow a single global order where all operations respect the program order.

Why (1,1) Is Valid Under SC

Outcome (1,1):

Meaning: Both Processor 1 and Processor 2 read 1 for their respective variables, indicating that both writes (x = 1 and y = 1) were completed before either read occurred.

Consistency with SC:

Program Order for Both Processors: This result maintains a single global order where both writes are visible to both reads. It respects the program order on each processor and ensures that all processors observe operations in a consistent sequence.

Total Store Order (TSO)

Axiom: Writes by a single processor appear in the order issued, but reads may see writes from other processors out of order.

Example:

```
Processor 1:      Processor 2:
x = 1;            y = 1;
r1 = y;           r2 = x;
```

Possible outcomes for (r1, r2) under TSO: (0,1), (1,0), (1,1), and potentially (0,0) if reads are reordered with respect to writes.

3. **Operational Models**

Definition: Operational models describe memory consistency by modeling the actual execution of memory operations as state transitions in a system. They focus on how operations are processed, propagated, and observed in real-time.

Purpose:

To simulate and understand the dynamic behavior of memory operations.
To illustrate how the state of the system changes with each memory operation.

3.1. Example Models

Sequential Consistency (SC)

Mechanism: Operations are executed in a single, global sequence that is consistent with program order.

Example:

```
Processor 1:      Processor 2:
x = 1;            y = 1;
r1 = y;           r2 = x;
```

Operations must be interleaved in a way that respects the program order on each processor.

Total Store Order (TSO)

Mechanism: Writes are seen in program order by all processors, but reads can be out of order.

Example:

```
Processor 1:       Processor 2:
x = 1;             y = 1;
r1 = y;            r2 = x;
```

Writes are ordered, but reads may observe writes from different stages, potentially allowing outcomes like (0,0).

4. Key Concepts of axiomatic model

Consistency Axioms: These are the rules that define the allowed orderings of memory operations. For example, a common axiom is "reads-from," which determines the value returned by a read operation.
Memory Order: Defines the sequence in which memory operations are observed. Common memory orders include sequential consistency, total store order, and relaxed memory order.
Visibility and Propagation: Rules that define when and how the effects of memory operations (writes) become visible to other processors.
Fences and Barriers: Special instructions that enforce ordering constraints between memory operations.

4.1. Consistency Axioms

Consistency axioms define the rules that dictate how memory operations should be ordered and how the results of these operations are observed by different threads or processors.

4.1.1 Reads-From (RF) Axiom

The Reads-From (RF) axiom describes how a read operation relates to previous write operations. It ensures that a read operation must read from a preceding write operation.

Example:

Suppose you have two processes, P1 and P2, and a shared variable x.

P1:

```
x = 1;
```

P2:

y = x;

If P2 reads the value of x after P1 has written 1 to x, then according to the RF axiom, the value y read by P2 must be 1.

4.1.2 Writes-From (WF) Axiom

The Writes-From (WF) axiom ensures that if a process writes to a memory location, subsequent reads of that location by the same or different processes will see the written value, provided there are no intervening writes.

Example:

P1:

x = 2;

P2:

x = 3;
y = x;

In this example, if P1 writes 2 to x and P2 writes 3 to x and then reads x, the value of y read by P2 should be 3 according to the WF axiom.

4.1.3. Program Order (PO) Axiom

The Program Order (PO) axiom ensures that instructions in a single thread are executed in the order they appear in the program.

Example:

P1:

x = 1;
y = 2;

In this case, the value assigned to y cannot be read before x is assigned 1 due to the program order constraint.

4.1.4. Read-After-Write (RAW) Axiom

The Read-After-Write (RAW) axiom ensures that a read operation cannot see an outdated value if it follows a write operation to the same memory location.

Example:

P1:

```
x = 5;
```

P2:

```
y = x;
```

Here, if P1 writes 5 to x, P2's read of x should see 5 and not an earlier value if it follows the RAW axiom.

4.1.5. Write-After-Read (WAR) Axiom

The Write-After-Read (WAR) axiom stipulates that a write operation should not occur to a memory location before a previous read to that location is completed.

Example:

P1:

```
y = x;
```

P2:

```
x = 6;
```

In this scenario, P1 must complete its read of x before P2 performs its write to x, adhering to the WAR axiom.

4.1.6. Write-After-Write (WAW) Axiom

The Write-After-Write (WAW) axiom ensures that if two writes occur to the same memory location, the final value seen should be the result of the last write operation.

Example:

P1:

```
x = 4;
```

P2:

x = 7;

If P2's write to x happens after P1's write, any subsequent read of x should reflect the value 7 from P2, ensuring WAW consistency.

4.1.7. Initialization (INIT) Axiom

The Initialization (INIT) axiom defines the initial state of memory locations before any operations take place.

Example:

Before any processes start, assume all memory locations are initialized to 0. This ensures that if no writes have occurred yet, any read operation should return 0.

Initial State:

x = 0;

Any read of x before any writes will return 0, according to the INIT axiom.

4.1.8. Consistency (CONS) Axiom

The Consistency (CONS) axiom enforces that observed memory states must comply with the overall consistency model (e.g., sequential consistency).

Example:

If you have a sequentially consistent memory model and processes are executing as follows:

P1:

x = 1;

P2

y = x;

P2 should observe y as 1 if P1 wrote 1 to x before P2 reads x, ensuring that memory operations appear as if they are executed in a sequential order.

4.2 Consistency models

Here’s an overview of some common consistency models:

4.2.1. Sequential Consistency (SC)

Definition: Sequential Consistency requires that the results of execution appear as if all operations were executed in some sequential order, and each process sees the operations in the same order.

We have two processes, P1 and P2, and two shared variables x and y:

P1:

```
x = 1;
y = 2;
```

P2:

```
a = x;
b = y;
```

Here, P1 writes 1 to x and then 2 to y. P2 reads the values of x and y and stores them in a and b, respectively

Sequential Consistency requires that the result of execution is as if all operations were executed in some sequential order, and each process sees the operations in that order.

1. **Execution Order:**

   Suppose P1 executes x = 1 followed by y = 2.
   Then P2 reads x and y.
2. **Possible Observations under SC:**

   If P1 finishes executing both operations before P2 starts, P2 will see a = 1 and b = 2 because SC ensures that operations appear in a global, sequential order.
   The order of operations across processes is strictly preserved.
3. **Result:**

   P2 will definitely observe a = 1 and b = 2 if the reads occur after P1 has completed its writes.

4.2.2. Total Store Order (TSO)

Total Store Order allows some reordering of writes to different locations but maintains the order of writes to the same location. Reads might see stale values due to the reordering.

1. **Execution Order:**

   Suppose P1 writes x = 1 and then writes y = 2.
   P2 reads x and y.

2. **Possible Observations under TSO:**

   Write Reordering: TSO allows writes to different locations (x and y) to be observed in a different order than they were issued. For example, P2 might observe y = 2 before x = 1 if there's reordering.
   Write Order Preservation: Writes to the same location are observed in the order they were issued. If P1 writes x = 1 and then x = 2, P2 will see x = 2 if it reads after the write.

3. **Possible Results:**

   Scenario A: If P1 writes x = 1 and then y = 2, but the writes are reordered, P2 might see a = 1 and b = 2 if it reads x after x = 1 and y after y = 2.
   Scenario B: Due to potential reordering, P2 might see a = 1 and b = 0 if P1's write to y was delayed or not observed yet when P2 reads y.

4.2.3. Release Consistency (RC)

Definition: Release Consistency is an optimization of sequential consistency that differentiates between acquire and release operations. It requires that all operations between an acquire and a release operation be seen as consistent, but does not require a global ordering of all operations.

Example:

Consider:

P1:

```
acquire(lock);
x = 1;
release(lock);
```

P2:

```
acquire(lock);
y = x;
release(lock);
```

Under release consistency, P2 will read x = 1 only if P1 has released the lock after writing x = 1, ensuring proper synchronization.

4.2.4. Weak Consistency (WC)

It is a memory consistency model that provides a more relaxed guarantee compared to Sequential Consistency (SC) and Total Store Order (TSO). It is designed to improve performance by allowing more flexibility in the ordering of memory operations. However, this relaxation comes at the cost of weaker guarantees about the order of operations observed by different processes.

Key Characteristics of Weak Consistency:

1. **Relaxed Ordering:**

   WC allows for significant reordering of reads and writes across different memory locations. This flexibility helps in optimizing performance but can lead to less predictable behavior in terms of memory visibility.

2. **Consistency Constraints:**

   WC ensures consistency only when certain conditions are met. It introduces additional synchronization mechanisms (such as locks or barriers) to enforce consistency.

3. **No Global Order Guarantee:**

   Unlike SC, which provides a global order of operations, WC does not guarantee a single, global order of operations. The order in which operations are observed can vary based on the synchronization mechanisms used.

Example of Weak Consistency:

Let’s consider two processes, P1 and P2, and two shared variables, x and y:

P1:

```
x = 1;
sync();  // Synchronization point
y = 2;
```

P2:

```
sync();  // Synchronization point
a = x;
b = y;
```

The sync() operation acts as a point where memory operations before the sync() are guaranteed to be visible to all processes or threads after the sync(). It ensures that all preceding writes are observed by other processes before any operations that follow the synchronization point.

In Weak Consistency:

1. **Consistency at Synchronization Points:**

   Before Synchronization: The model allows reordering of operations before synchronization points, which means P2 can see y = 2 before x = 1 if the sync() is not yet encountered.
   After Synchronization: Once synchronization points are reached, the model ensures that the operations before the synchronization in P1 are visible to P2 after the synchronization point.

Types of Synchronization Operations

Memory Barriers (Fence Instructions):

Low-level operations that enforce ordering constraints between memory reads and writes. They ensure that all operations before the barrier are completed before any operations after the barrier.

Locks and Semaphores:

Higher-level synchronization mechanisms that control access to shared resources and ensure proper ordering of operations.

Barrier Synchronization:

A synchronization point where all participating threads or processes must reach before any can proceed.

4.2.5. Data Race-Free Consistency (DRF)

Definition: Data Race-Free Consistency is a model where memory operations are consistent if the program is data race-free. It means that if the program is free of data races, its behavior will be consistent as if it were sequentially consistent.

Example:

If a program is guaranteed to be data race-free:

P1:

```
x = 1;
```

P2:

```
y = x;
```

If there are no data races (no concurrent writes to x without proper synchronization), then P2 should consistently read x = 1 after P1 writes it.

4.2.6. Causal Consistency

Causal Consistency is a memory consistency model that guarantees a certain level of ordering for operations based on their causal relationships. It is more relaxed than Sequential Consistency but provides stronger guarantees than models like Weak Consistency.

Key Concepts of Causal Consistency:

1. **Causal Relationships:**

   Causal Relation: Operations are causally related if one operation (A) causally influences or affects another operation (B). For instance, if operation A must happen before operation B because A sets a value that B reads, then A and B are causally related.

Non-Causal Relation: Operations that do not have a direct cause-and-effect relationship are not causally related. They can be observed in any order relative to each other.

2. **Causal Ordering:**

   Causal Consistency ensures that all operations that are causally related are observed by all processes in the same order. This means that if operation A causally affects operation B, then every process or thread will see A and B in that order.

   Non-Causal Operations: Operations that are not causally related can be seen in a different order by different processes. The model does not enforce a specific global order on these operations, providing flexibility in how they are observed.

Consider:

P1:

```
x = 1;
```

P2:

```
y = x;
```

If P2 reads x after P1 has written 1 to x, it should see y = 1. However, if P1 and P2 are not synchronized, P2 might see different values based on the causal relationship between operations.

4.2.7. Processor Consistency (PC)

Definition: Processor Consistency is a model where writes to a single location are observed in the same order by all processors, but different locations can be seen in different orders.

Example:

Consider:

P1:

x = 1;
y = 2;

P2:

z = y;

Under processor consistency, if P1 writes 1 to x and 2 to y, P2 should see the writes to x and y in the same order. However, it does not guarantee the order of operations to different locations.

4.3 Memory Order

Memory order determines how memory operations (loads and stores) are perceived by different threads or processors. It affects the visibility and ordering of operations and can influence the behavior of concurrent programs.

4.3.1. Sequential Consistency (SC)

Definition: Sequential consistency is a strong memory model that ensures all operations appear to be executed in a single, global order that is consistent with the program order of each thread. In other words, the results of memory operations are as if all operations were executed one after another in some global sequence.

Key Points:

Global Order: All threads observe memory operations in a globally agreed-upon order.
Program Order: The operations appear in the same order to all threads as they appear in each individual thread's program.

Example: Consider two threads:

Thread 1:

x = 1;
y = 1;

Thread 2:

```
if (y == 1) {
    print(x);
}
```

Expected Behavior:

With SC, if Thread 2 sees y as 1, it should observe x as 1 because all operations are perceived in a globally consistent order.

Reordering:

There is no reordering allowed between memory operations. If Thread 1 writes x = 1 and then y = 1, Thread 2 must see y as 1 before it can observe x.

4.3.2. Total Store Order (TSO)

Definition: Total Store Order (TSO) is a weaker memory model compared to SC. It ensures that stores are globally ordered, but allows more flexibility for the ordering of loads relative to stores.

Key Points:

Store Ordering: Stores are globally ordered and cannot be reordered with other stores.
Load-Store Reordering: Loads can be reordered with other loads and stores, which means that loads might see stores in a different order than they were issued.

Example: Consider two threads:

Thread 1:

```
x = 1;
y = 1;
```

Thread 2:

```
if (y == 1) {
    print(x);
}
```

Expected Behavior:

With TSO, Thread 2 might still print x = 1 if it sees y = 1, but it is possible for Thread 2 to observe a different value for x if the load from Thread 2 is reordered with the stores from Thread 1.

Reordering:

Stores: Stores are globally ordered, but **loads** might see the effects of stores in a different order than they were issued.

4.3.3. Relaxed Memory Order

Definition: Relaxed memory order allows the greatest flexibility in how memory operations are reordered. It does not impose a strong global ordering of operations, which can lead to various observable orders depending on the memory model and architecture.

Key Points:

Flexible Ordering: Allows significant reordering of operations for performance optimization.
Explicit Synchronization: Requires explicit synchronization mechanisms (like memory barriers) to enforce specific ordering constraints when needed.

Example: Consider two threads:

Thread 1:

```
x = 1;
```

Thread 2:

```
if (x == 1) {
    print(y);
}
```

Expected Behavior:

With relaxed memory order, Thread 2 might not see the updated value of y even if x is 1, because loads and stores are not strictly ordered. It's up to the programmer to use synchronization mechanisms to ensure the desired ordering.

Reordering:

Both **loads** and **stores** can be reordered extensively. The memory model does not enforce any global ordering beyond what is explicitly specified by memory barriers.

4.4. Visibility and Propagation

Visibility and **Propagation** address how changes made by one processor (such as writes to memory) become observable by other processors. These concepts are crucial for understanding how memory operations can be seen across different threads or processors.

4.4.1 Visibility

Definition: Visibility refers to the ability of one processor to see the updates made by another processor. It deals with when and how a write operation performed by one processor becomes visible to other processors.

Key Points:

> **Write Visibility**: When Processor 1 writes to a memory location, Processor 2 must see the latest value written if it reads from the same location.
> **Memory Coherence**: Ensures that all processors have a consistent view of memory. Coherence protocols manage visibility by ensuring that writes are propagated correctly across caches and memory.

Example: Imagine two processors, A and B, accessing a shared variable x.

> **Processor A** writes x = 1;
> **Processor B** reads x;

Expected Behavior:

> **Immediate Visibility**: Processor B should read x as 1 if Processor A's write has been made visible. If the memory model or cache coherence protocol delays the visibility, Processor B might see an old value of x.

Reordering Example:

> If Processor B's read operation is delayed or reordered, it might not immediately reflect Processor A's latest write, leading to inconsistencies if Processor B observes an outdated value.

4.4.2 Propagation

Definition: Propagation is the process by which changes (such as writes) are communicated and updated across different parts of the memory system, ensuring all processors eventually observe the latest values.

Key Points:

Cache Coherence Protocols: Ensure that all caches and memory are updated with the latest values. Common protocols include MESI (Modified, Exclusive, Shared, Invalid).
Latency: The time it takes for a write to propagate through the memory system to other processors.

Example:

Processor A writes x = 1; to a shared variable.
Processor B reads x; from the same location.

Expected Behavior:

If the cache coherence protocol and propagation mechanisms are working correctly, Processor B will see the updated value of x as 1.

Propagation Delays:

Cache Misses: If Processor B's cache does not have the updated value, it must fetch it from main memory or another cache, which introduces delay.

4.5. Fences and Barriers

Fences and **barriers** are special instructions used to control the ordering of memory operations and ensure that memory operations are observed in the desired order.

Definition: Fences are instructions that enforce ordering constraints between memory operations. They ensure that certain memory operations are completed before others begin, which is essential for maintaining consistency in concurrent programs.

Key Points:

Types of Fences: Different architectures have different types of fences, such as **mfence in x86, dmb in ARM, and fence in RISC-V.**
Scope: Fences typically affect memory operations performed by the processor they are executed on, and sometimes they can influence the behavior of other processors through coherence protocols.

Example (x86):

Code:

```
mov [x], 1;
mfence; // Memory fence
mov [y], 2;
```

Expected Behavior:

The mfence ensures that the write to x is completed and visible to other processors before the write to y begins. Without the fence, there is no guarantee about the order in which these writes are observed.

Reordering Example:

Without mfence, the processor might reorder the instructions, potentially writing y before x is fully visible.

5. Key concepts of Operational Model

Overview

The operational model describes memory consistency by modeling the actual execution of memory operations as state transitions in a system. It focuses on how operations are processed and their effects on the state of memory over time.

Key Concepts

State Transitions: Each memory operation (load, store, atomic) causes a transition from one state to another.

Execution Order: Specifies the order in which memory operations are executed and how they affect the memory state.

Processor and Memory Interaction: Describes how individual processors interact with memory and how changes propagate.

Sequential Consistency: In its strictest form, all operations appear to execute in some sequential order that is consistent with the program order.

Let's explore each key concept of the operational model in detail:

5.1. State Transitions

Definition: State transitions refer to the changes in the status of a memory location or cache line as a result of memory operations. Each type of operation (read, write, or atomic) can cause these states to change based on the current state of the memory or cache line.

5.1.1 States

Invalid (I)

A state indicating that the cache line does not contain valid data.

Shared (S)

A state where the cache line is valid and possibly held by multiple caches. The data is consistent with the main memory.

Modified (M)

A state where the cache line is valid and contains data that has been modified. The data is not consistent with main memory and is exclusive to this cache.

Exclusive (E)

A state where the cache line is valid and contains data that is exclusive to this cache. The data matches main memory, but no other cache holds this line.

5.1.2 Transitions

Read Miss (Invalid to Shared/Exclusive)

When a read operation is issued for a cache line that is in the Invalid state, the cache line transitions to Shared if other caches have the line or Exclusive if no other cache has it.

Write Miss (Invalid/Shared to Modified)

When a write operation is issued for a cache line that is in the Invalid or Shared state, the cache line transitions to Modified. Other caches may need to invalidate their copies.

Read Hit (Shared/Exclusive)

When a read operation hits a cache line in Shared or Exclusive state, the cache can return the data without changing the state.

Write Hit (Exclusive to Modified)

When a write operation hits a cache line in Exclusive state, the state transitions to Modified, indicating that the line has been updated.

Invalidate (Shared/Modified to Invalid)

If another processor writes to a memory location that a cache has in Shared or Modified state, the line transitions to Invalid in the current cache.

5.2. Execution Order

Definition: Execution order defines the sequence in which memory operations are executed and observed. This includes both the order of operations within each processor and the order in which operations are observed across different processors.

Details:

Program Order: Each processor has a specific order in which it executes its operations (i.e., the order in which instructions are issued). Execution order must respect this order within each processor.
Global Order: In models like Sequential Consistency, all operations must appear to execute in a global order that respects each processor's program order.
Consistency: Different consistency models allow different degrees of flexibility in how operations are ordered and observed. For instance, Sequential Consistency requires strict adherence to a global order, while Total Store Order allows some degree of reordering.

5.3. Processor and Memory Interaction

Definition: This concept describes how individual processors interact with memory and how these interactions affect memory states and coherence.

Details:

Memory Operations: Processors issue read and write operations to memory. The interaction can involve fetching data from main memory to a cache, updating data in a cache, or writing modified data back to main memory.

Caches: Processors use caches to store frequently accessed data. The cache must maintain coherence with the main memory and with other caches in a multi-processor system.

Propagation: Changes made by one processor (e.g., writing data) must be propagated to other caches and memory to maintain consistency. This involves notifying other caches of changes and ensuring that all caches have a coherent view of the memory.

5.4. Sequential Consistency (SC)

Definition: Sequential Consistency (SC) is a memory consistency model where the results of execution are as if all memory operations were executed in some sequential order that is consistent with the program order on each processor.

Key Aspects:

Global Order: All operations appear to execute in a single, consistent global order. This global order respects the program order of each individual processor.

Program Order: Each processor's operations are seen in the order they were issued, and this order is respected across all processors.

Strict Consistency: SC ensures that no matter how operations are interleaved, all processors observe a coherent view of memory where operations appear in a consistent global sequence.

Example: Consider two processors with the following operations:

Processor 1: write(x = 1); read(y);

Processor 2: write(y = 1); read(x);

In SC, if Processor 1 reads y and sees 1, then Processor 2 must also see the write to x (i.e., (1,1) is a valid outcome). Any outcome where the reads do not see the latest writes in a globally consistent manner (like (1,0) or (0,1)) is not valid under SC.