

The SC- Memory Model for Java

1/7/2004 3:07 AM

The memory model is the interface between the system and the programmer that defines the values that a read in a program is allowed to return. This document motivates and formalizes a memory model for Java, focusing only on accesses to volatiles, monitors, and ordinary variables.

1. High-level (Informal) Requirements for the Java Memory Model

The informal requirements for the Java Memory Model (JMM) are based on two observations on which there is widespread consensus:

- *Sequential consistency (SC) is the most intuitive programming model.*
With SC, programmers can assume that the system executes the instructions of the program one at a time in a total order such that (i) the instructions of a given thread occur in the order specified by its program and (ii) a read returns the value of the last write to the same location that occurs before the read in the total order.

Unfortunately, SC constrains several performance-enhancing system optimizations. While there has been much progress in high-performance SC hardware implementations, much remains to be done for compilers, runtime systems, and even hardware. Thus, today, pure SC is not a viable memory model.

- *Programmers must write data-race-free programs.*
For various reasons related to ease of understanding and debugging, it is widely considered that multithreaded code should be data-race-free, and programmers must be dissuaded from using data races.

Based on the above two observations, there is widespread consensus that the JMM must provide SC to data-race-free programs and only the minimal possible guarantees for programs with data races. The following summarizes the high-level goals for the JMM.

1.1 High-level goals for the Java memory model (JMM)

(1) *SC for data-race-free programs:* It is important to note that we say a program is data-race-free if *all SC executions* of the program are data-race-free (i.e., non-SC executions of a “data-race-free” program can possibly have data races). The reason is that it would not be very useful to provide an SC model if programmers had to reason with a non-SC model to determine if a program has a data race. (The formal definition of a data race follows later.)

(2) *Programmer-centric requirements for programs with data races:* Ensure safety/security properties of Java. These requirements are the key problem, since there has never before been a formal specification of these properties. We elaborate on these in the next section.

(3) *System (implementer)-centric requirements for programs with data races:* The model must allow all reasonable current system (including hardware, compiler, runtime software) optimizations and ideally all possible future optimizations. This essentially implies that the model must impose no more conditions than necessary to obey the programmer-centric requirements.

(4) *Understandability:* The model must be unambiguous and as simple as possible so it can be understood with as little effort as possible.

The challenge here is to formalize the properties for goal (2) in a way that will meet goals (3) and (4). There is consensus on the informal requirements for goal (2), which can be stated as in the next section.

Before we go ahead, it is worth emphasizing that the above list of goals does not include that “it should be easy to derive from the JMM specification whether a specific system optimization is allowed.” This would certainly be a good property to have, but the more important property from the system or implementer’s viewpoint is that the JMM not prohibit current or future system optimizations. Achieving both properties at the same time from the same specification is hard, so only the second is listed as the high-level system-centric goal. This does not mean that we want to abandon

implementers! Instead, we provide other specifications which give sufficient conditions to obey the JMM. These alternate specifications (e.g., release consistency, lazy release consistency, and the more aggressive specification described later) will be easier to map to real implementations. However, these alternate specifications are not the JMM, and implementers are free to violate and ignore these alternate specifications as long as they can show that their systems obey the JMM.

This document concerns the JMM. An accompanying document describes an alternate system-centric specification that obeys the JMM described here, allows aggressive optimizations (intended to include all optimizations I am aware of in current systems), and is easier to map to real systems.

1.2 Informal requirements to meet goal (2) (safety and security):

(2.1) SC for data-race-free parts of the program: A data race should not cause a violation of SC in parts of the program that are unrelated to the data race.

Example 1: Consider a program with four threads. Suppose the memory addresses accessed by threads 1 and 2 are completely disjoint from those accessed by threads 3 and 4. Then a data race between memory accesses in threads 1 and 2 should not affect the behavior of threads 3 and 4 (i.e., the code for threads 3 and 4 should appear to execute in a sequentially consistent fashion).

Example 2: Consider a program with 4 threads. Now consider adding thread 5 to this program where the thread simply performs a read to a location written in one of the four threads. This read clearly induces a data race with the write to that location. Introducing this single read should not affect the behavior of the program. This is the “non-intrusive reads” property identified by Gao and Sarkar.

(2.2) Data race causality: A violation of SC should not cause a data race which is then used to justify the violation of SC.¹

Example 3:

Initially A = B = 0	
Thread 1	Thread 2
if (A == 1)	if (B == 1)
B = 1	A = 1

Consider the outcome where the reads of A and B return the value 1. That is, the execution consists of the following accesses:

Thread 1: Read, A, 1
Thread 1: Write, B, 1
Thread 2: Read, B, 1
Thread 2: Write, A, 1

The above execution is not SC since there is no total order on its memory accesses that is consistent with program order and where a read returns the value of the last write to the same location.

The accesses to A and to B do form a data race in this example and so could potentially be used to justify the violation of SC. Note, however, that these data races would never have occurred in an SC execution (in fact, the writes to A and B would never be executed in an SC execution). In fact, these data races occur precisely because there is a violation of SC, and therefore using them to justify the violation of SC is a violation of causality that property 2.2 seeks to avoid.

Another way of looking at this example is that the program is data-race-free since it does not have any data races in any SC execution. So it should appear to execute in an SC fashion by goal 1 of Section 1.1. (Property 2.2 is more general than goal 1 because it imposes a requirement on programs that may have SC data races in some part of the program.)

¹ I use the term “causality” only because in the past, the Manson/Pugh model has been distinguished from SC- on the basis of causality. I use this term to emphasize that SC- also has a notion of causality, although neither the M/P model nor SC- obey the stricter notion of causality that has often been used in the literature.

(2.3) *Value causality (a.k.a. not out-of-thin-air values)*: A violation of SC should not cause a value which is then used to justify the violation of SC.

Example 4:

	Initially A = B = 0
Thread 1	Thread 2
r1 = A	r2 = B
B = r1	A = r2

Consider the outcome where the writes of A and B both write the value 42 and the reads also read 42. That is, the execution consists of the following accesses:

Thread 1: Read, A, 42
Thread 1: Write, B, 42
Thread 2: Read, B, 42
Thread 2: Write, A, 42

A violation of SC may allow these reads and writes to have the value 42; however, this value is also responsible for the violation of SC. This violates the value causality property 2.3.

Another way of thinking about this example is using the “out-of-thin-air” formalization. The values 42 are generated out of thin air, and property 2.3 does not allow that.

2. The SC- Model

Based on the informal requirements from the previous section, we seek to describe a model that behaves like SC for the most part. Informally, we can imagine an abstraction for an execution where memory accesses are executed one at a time in program order and a read returns the value of some conflicting² write in the execution. For SC, a read should return the value of the conflicting write that occurred last before it. However, in general, we could see non-SC behavior where a read returns the value of a write arbitrarily far from the past or even from the future. The memory model constrains which values are valid for a read. We call a read that does not return the value of the last conflicting write as a non-SC read and we refer to this as an SC violation.

Based on the three requirements from the previous section, we want to ensure that (1) a non-SC read occurs only due to a data race that directly affects it, (2) this data race could have occurred even without the SC violation, and (3) some write to the same address as the non-SC read could have written the value returned by the read even without the SC violation.

Informally, we ensure the above as follows. The last page of this document gives the full formalism. We require that we should be able to *validate* the value of all reads in an SC- execution in some total order as follows. For a read’s value to be validated, there should be a conflicting write with the same value in some valid execution E_v (we define valid executions in the next paragraph). Further, either (i) the value is an SC value for the read in E_v or (ii) the read and/or the write that wrote that value are involved in a data race in E_v and the write is hb-consistent (defined below) for the read in E_v . With this, we ensure that a validated read’s value is either an SC value or the accessed location is involved in a data race. The hb-consistency constraint ensures that the impact of the data race is bounded (i.e., the read is not allowed to return a value that was synchronized too long ago in the past nor one that will be synchronized in the future).

To ensure the causality properties (second and third properties above) are satisfied, we constrain the valid executions E_v that can be used to validate a read. An execution E_v used to validate a read must either have no non-SC reads itself, or any non-SC read in E_v should be a previously validated data read from E and return the same value from the same write as in E . These constraints ensure that the data race used to validate a non-SC read occurs even without the SC violation. They also ensure that the write whose value the non-SC read returned also occurs without this SC violation.

² Two accesses conflict if they are to the same location and at least one of them is a write.

More precisely, **an execution E obeys SC-** if all its reads can be validated in some total order. A read R' is validated for E if there exists an execution Ev such that the following holds:

- All non-SC reads in Ev are data reads that occur in E, have been previously validated for E, and return the same value in Ev and E.
- Let R be R' or a previously validated read in E. Let R return the value of W in E. Then
 - W and R occur in Ev, and W is hb-consistent for R in E and Ev.³
 - If R is a synchronization access, then W is the last conflicting write before R in E and Ev.
 - Either W and R form a data race in both E and Ev or W hb R in both E and Ev.

(Above, when we say an access in execution E1 occurs in execution E2, we not only mean that the instruction containing the access occurs in E2, but that this access has the same address and writes the same value (if it is a write) in E1 and E2.)

The above specification obeys the intuition set out so far as follows. The first bullet simply ensures that the validating execution Ev obeys the constraint outlined in the previous paragraph. The first two parts of the second bullet together ensure that the read being validated either returns the value of the last conflicting write before it in Ev (i.e., an SC value), or the location is involved in a data race (since otherwise it won't be happens-before consistent). We require that Ev obey this property for all reads validated in the past as well – this aspect and the last part of the second bullet above together with the previous constraints ensure that overall data-race-free programs appear SC.

3. SC- and the Goals in Section 1.

We next discuss how SC- obeys the four high-level goals described in Section 1.

(1) *SC for data-race-free programs:* An accompanying document provides the proof that SC- provides SC to data-race-free programs (<http://www.cs.uiuc.edu/~sadve/jmm/proof-sc-for-drf.pdf>).

(2) *Programmer-centric requirements for programs with data races:* Section 2 already motivated how SC- obeys the three requirements of SC for this part. Briefly, again, the SC for data-race-free parts of the program requirement is obeyed by ensuring that a read returns a non-SC value only if the read or writes that conflict with it are involved in a data race. The data race and value causality properties are met by ensuring that reads are validated one by one against executions that are either SC or exhibit non-SC behavior only for those reads that are already validated and return the validated values. The specification also requires that the validating executions contain all the previously validated reads, these reads continue to be race-consistent, and the hb relationship between the reads and the writes whose value they return be the same in the validated and validating executions. These requirements together ensure that a data-race-free program will appear SC.

(3) *System (implementer)-centric requirements for programs with data races:* In an accompanying document, we describe a system-centric specification that is sufficient to obey the SC- model and provide a proof for this (<http://www.cs.uiuc.edu/~sadve/jmm/system-centric-spec.pdf>). Informally, the document shows that if a system ensures that data reads return hb-consistent values, synchronization reads are executed in an SC manner, and writes are not executed “speculatively,” then the system will obey SC-. This set of requirements encompasses all the traditional reordering optimizations; e.g., as allowed by release consistency. It also allows elimination of thread-local locks and allows the less traditional techniques of lazy release consistency (where, for example, a processor may execute a write synchronization before its previous data accesses are done, as long as these accesses are made visible to a processor that issues a matching read synchronization). This also proves that speculative reads are allowed (e.g., a compiler may speculatively hoist a load across an acquire, as long as the value is checked later after the acquire, such as using IA-64 speculative load and chk instructions).

(4) *Understandability:* This is subjective, but I believe the SC- model is simpler to understand than the M/P model.

³ If R is a data access, then W is hb-consistent for R in E if there is no conflicting write W' in E such that $W \text{ hb } W' \text{ hb } R$ in E and if R is not ordered before W by hb in E. The hb relation is the irreflexive, transitive closure of program order and edges such as $W' \rightarrow R'$ where R' is a synchronization read that returns the value of W' .

4. The Full Formal SC- Memory Model for Java

Program: When we refer to a program, we also include an input set (i.e., the same code with a different input is a different program).

Execution E of a Java program P has the following properties:

- E specifies a set of read and write memory accesses in a total order, called the **serialization order**. Each write specifies a thread, an address, and a value. Each read specifies a thread, an address, and a write in the execution that is to the same address and whose value the read returns. The execution order restricted to accesses of the same thread is called the **program order**.
- The memory accesses and program order of an execution must represent a correct uniprocessor. That is, a correct uniprocessor running the program P for thread T would generate the accesses of thread T in E with the same program order as E, with the assumption that the reads in the uniprocessor return the same values as in E (e.g., from a memory system that magically generates these values).
- E may specify some addresses as volatiles or monitors. All accesses to these addresses are called **synchronization accesses** and others are called **data**.⁴
- E contains a special (*hypothetical*) **initialization** thread that writes initial values to all addresses, followed by a write to a unique volatile address. For all other threads, the first access (by program order) is a (*hypothetical*) read to this unique address that returns the value of the above write.

Conflicting accesses: Two accesses to the same address where at least one of them is a write.

Non-SC read: A read is non-SC in an execution E if it does not return the value of the conflicting write ordered last before it by the serialization order of E.

Happens-before relation (hb), defined on memory accesses of an execution: $I \text{ hb } J$ if (1) I is before J by program order, or (2) I is a synchronization write, J is a synchronization read, and J returns the value of I, or (3) $I \text{ hb } K \text{ hb } J$ for some K.

Data race: Two conflicting memory accesses form a data race if they are not ordered by hb.

Hb-consistency: A write W is hb-consistent for a conflicting read R in execution E if R is not before W by hb in E and if there is no conflicting write W' such that $W \text{ hb } W' \text{ hb } R$ in E.

Race-consistency: A write W is race-consistent for a conflicting read R in E if (1) W is hb-consistent for R in E and (2) if R is a synchronization access, then W is the last conflicting write before R in E's serialization order.

An execution E obeys SC- if all its reads can be validated in some total order. A read R' is validated for E if there exists an execution E' such that the following holds:

- All non-SC reads in E' are data reads that occur in E, have been previously validated for E, and return the value of the same write in E' and E.
- Let R be R' or a previously validated read in E. Let R return the value of W in E. Then
 - R and W occur in E' and W is race-consistent for R in E and E' .
 - Either W and R form a data race in both E and E' or $W \text{ hb } R$ in both E and E' .

Above, when considering two executions E1 and E2 of program P, we say an **access A1 in E1 occurs in E2 or is the same as access A2 in E2** if A1 is matched with A2 as follows. We can match two reads or two writes with each other if the set of all matched accesses (for E1 or E2) is a *maximal* set that obeys the following properties:

- Two accesses matched with each other must be from the same thread, access the same address, and write the same value (if they are writes).
- The sets of matched accesses should have the same program order relation in E1 and E2; i.e., if A1 and B1 in E1 are respectively matched with A2 and B2 in E2, then $A1 \text{ po } B1$ in E1 implies that $A2 \text{ po } B2$ in E2.

⁴ There are constraints on monitor accesses outside of those specified in the memory model (e.g., a lock must be held before it is freed, etc.). These can be integrated here as well.