N3125: Omnibus Memory Model and Atomics Paper

ISO/IEC JTC1 SC22 WG21 N3125 = 10-0115 - 2010-08-22

Paul E. McKenney, paulmck@linux.vnet.ibm.com
Mark Batty, mjb220@cl.cam.ac.uk
Clark Nelson, clark.nelson@intel.com
Hans Boehm, hans.boehm@hp.com
Anthony Williams, anthony@justsoftwaresolutions.co.uk
Scott Owens, Scott.Owens@cl.cam.ac.uk
Susmit Sarkar, susmit.sarkar@cl.cam.ac.uk
Peter Sewell, Peter.Sewell@cl.cam.ac.uk
Tjark Weber, tw333@cam.ac.uk
Michael Wong, michaelw@ca.ibm.com
Lawrence Crowl, crowl@google.com

Introduction

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber recently analyzed a formalized variant of the C++ memory model, which uncovered a number of potential issues discussed in an email thread entitled “Some more memory model issues from Mark Batty” initiated by Hans Boehm on June 13, 2010, and in another email thread entitled “Further C++ concurrency discussion” initiated by Mark Batty on July 28, 2010. This paper summarizes the ensuing discussion, calling out changes that appear uncontroversial and summarizing positions on contended issues. Some of these issues have been captured as national-body comments, and the disposition of any remaining issues is to be determined.

Please note that only those issues related to the memory model are included in this paper. In addition, this version of the paper does not yet include issues raised in the July thread that were not also covered in the June thread.

More details on the work leading up to this paper may be found here and here.

Non-Controversial Issues Discussed in June Email Thread

CA 8: Inter-thread-happens-before is not acyclic [Clark]

Sections:

1.10p10, 1.10p11

Comment:

The following litmus test is generally agreed to be disallowed, however, Batty et al. uncovered a case where the standard does not forbid it, as shown by the following example from their paper:

Thread 0	Thread 1
`r1 = x.load(memory_order_consume);`	`r2 = y.load(memory_order_consume);`
`y.store(1, memory_order_release);`	`x.store(1, memory_order_release);`

The standard permits the counter-intuitive outcome `x == 1 && y == 1`, an outcome that cannot occur on any hardware platform that we are aware of.

Proposal:

In 1.10p10:

An evaluation *A* happens before an evaluation *B* if:

- *A* is sequenced before *B*, or
- *A* inter-thread happens before *B*.

The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation. [*Note:* This would otherwise be possible only through the use of consume operations. — *end note*]

Resolution:

Adopt the proposal.

CA 9: Imposed happens-befor e edges should be synchr onizes-with [Clark]

Sections:

1.10p7, 27.2.3p2, 29.3p1, 30.3.1.2p6, 30.3.1.5p7, 30.6.4p7, 30.6.9p5, and 30.6.10.1p23

Comment:

The happens-before relation is not transitive, and so it is not appropriate to specify happens-before for library functions that are intended to impose ordering because happens-before cannot always be extended using a trailing sequenced-before relation. Therefore, synchronized-with should be used in place of happens-before for this purpose.

Proposal:

Change 1.10p7:

Certain library calls *synchronize with* other library calls performed by another thread. ~~In particular, an atomic operation *A* that performs a release operation on an atomic object *M* synchronizes with an atomic operation *B* that performs an acquire operation on *M* and reads a value written by any side effect in the release sequence headed by *A*.~~ [*Example:* An atomic store-release synchronizes with a load-acquire that takes its value from the store. (29.3 atomics.order) — *end example*] [*Note: ...*

Insert a new paragraph following 29.3p1:

An atomic operation *A* that performs a release operation on an atomic object *M* synchronizes with an atomic operation *B* that performs an acquire operation on *M* and takes its value from any side effect in the release sequence headed by *A*.

Replace “happens-before” with “synchronizes-with” in 27.2.3p2:

If one thread makes a library call *a* that writes a value to a stream and, as a result, another thread reads this value from the stream through a library call *b* such that this does not result in a data race, then *a*'s write ~~happens before~~ synchronizes with *b*'s read.

Replace “happens-before” with “synchronizes-with” in 30.3.1.2p6:

Synchronization: The last store operation in the invocation of the constructor ~~happens before~~ synchronizes with the first read operation in the invocation of the copy of *f*.

Replace “happens-before” with “synchronizes-with” in 30.3.1.5p7:

Synchronization: The ~~completion of~~ last store operation carried out by the thread represented by **this* ~~happens before~~ synchronizes with (1.10) the corresponding successful *join()* return~~s~~. [Note: Operations on **this* are not synchronized. — end note]

Replace “happens-before” with “synchronizes-with” in 30.6.4p7:

Calls to functions that successfully set the stored result of an associated asynchronous state synchronize with (1.10) calls to functions successfully detecting the ready state

resulting from that setting. The storage of the result (whether normal or exceptional) into the associated asynchronous state ~~happens-before~~ synchronizes with (1.10) ~~that state is set to ready~~ the successful return from a call to a waiting function on the associated asynchronous state.

Replace “happens-before” with “synchronizes-with” in 30.6.9p5:

Synchronization: the invocation of async ~~happens-before~~ synchronizes with (1.10) the invocation of *f*. [Note: this statement applies even when the corresponding *future* object is moved to another thread. — end note] If the invocation is not deferred, a call to a waiting function on an asynchronous return object that shares the associated asynchronous state created by this *async* call shall block until the associated thread has completed. If the invocation is not deferred, the *join()* on the created thread ~~happens-before~~ synchronizes with (1.10) the first function that successfully detects the ready status of the associated asynchronous state returns or before the function that gives up the last reference to the associated asynchronous state returns, whichever happens first. If the invocation is deferred, the completion of the invocation of the deferred function ~~happens-before~~ synchronizes with the the successful return from a call to a waiting function on the associated asynchronous state. ~~calls to the waiting functions return.~~

Replace “happens-before” with “synchronizes-with” in 30.6.10.1p23:

Synchronization: a successful call to *operator()* synchronizes with (1.10) a call to any member function of a *future*, *shared_future*, or *atomic_future* object that shares the associated asynchronous state of **this*. The completion of the invocation of the stored task and the storage of the result (whether normal or exceptional) into the associated asynchronous state ~~happens-before~~ synchronizes with (1.10) the successful return from any member function that detects that the state is set to ready. [Note: *operator()* synchronizes and serializes with other functions through the associated asynchronous state. — end note]

Resolution:

Adopt the proposal.

CA 11: “Subsequent” in vsse de finition

Comment

Batty et al. propose removing the word “subsequent” from 1.10p12 (presumably instead meaning 1.10p13), stating that this will clarify the definition.

Discussion

This change has interesting consequences. The current wording is as follows, with the word being proposed for removal so marked:

The *visible sequence of side effects* on an atomic object `M`, with respect to a value computation `B` of `M`, is a maximal contiguous sub-sequence of side effects in the modification order of `M`, where the first side effect is visible with respect to `B`, and for every ~~subsequent~~ side effect, it is not the case that B happens before it. The value of an atomic object `M`, as determined by evaluation `B`, shall be the value stored by some operation in the visible sequence of `M` with respect to `B`. Furthermore, if a value computation `A` of an atomic object `M`happens before a value computation `B` of `M`, and the value computed by `A` corresponds to the value stored by side effect `X`, then the value computed by `B` shall either equal the value computed by `A`, or be the value stored by side effect `Y`, where `Y` follows `X` in the modification order of `M`. [Note: This effectively disallows compiler reordering of atomic operations to a single object, even if both operations are “relaxed” loads. This effectively makes the “cache coherence” guarantee provided by most hardware available to C++ atomic operations. — end note] [Note: The visible sequence depends on the

"happens before" relation, which depends on the values observed by loads of atomics, which we are restricting here. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the "happens before" relation derived as described above, satisfy the resulting constraints as imposed here. — end note]

The effect of the current wording is as follows:

1. The last side-effect in the modification order of `M` that happens before value computation `B` is the visible side effect. Call it `V`.
2. The first side-effect in the modification order of `M` such that `B` happen before it will be called `I`. `I` and all subsequent side-effects are *not* in the visible sequence of side effects.
3. The word "subsequent" adds the constraint that no side effect in the modification order of `M` that precedes `V` can be part of the visible sequence of side effects.

Does some hardware actually operate in this fashion, so that a value computation might return some value preceding the last side-effect in the modification order of `M` that happens-before that value computation?

Resolution:

Adopt the changes proposed for CA 8.

CA 12: The use of maximal in the definition of r elease sequence [Paul]

Sections:
1.10p6
Comment:

Batty et al. describe an interpretation of 1.10p6 that would only require that release sequences be extended back to the first release operation in a given thread out of a sequence of release operations on a given object.

This interpretation can be considered perverse in light of the wording of 1.10p7, however, the suggested modification is consistent with the intent.

Proposal:

Replace 1.10p6 with the following:

A *release sequence* from a release operation `A` on an atomic object `M` is a maximal contiguous sub-sequence of side effects in the modification order of `M`, where the first operation is ~~a release~~ `A`, and every subsequent operation

- is performed by the same thread that performed ~~the release~~ `A`, or
- is an atomic read-modify-write operation.

Please note that this has been modified slightly from that proposed by Batty et al.

Resolution:
Adopt the proposal.

CA 13: Wording of the r ead-read coherence condition [Paul]

Section:
1.10p13
Comment:

Batty et al. suggest that the following wording from 1.10p13:

> Furthermore, if a value computation A of an atomic object M happens before a value computation B of M, and the value computed by A corresponds to the value stored by side effect X, then the value computed by B shall either equal the value computed by A, or be the value stored by side effect Y, where Y follows X in the modification order of M.

be changed to the following:

> Furthermore, if a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from the side effect X, then the value computed by B shall either be the value stored by X, or the value stored by a side effect Y, where Y follows X in the modification order of M.

Proposal:

Use notation uniformly, as follows:

> Furthermore, if a value computation A of an atomic object M happens before a value computation B of M, and ~~the value computed by A corresponds to the value stored by~~ A takes its value from side effect X, then ~~the value computed by~~ B shall ~~either equal the value computed by A,~~ take its value either from X or ~~be the value stored by side effect~~ from a side effect Y, where Y follows X in the modification order of M.

Resolution:

Adopt the proposal.

CA 14: Initialization of atomics

Sections:

1.10p4

Comment:

Batty et al. suggest adding the following non-normative note to 1.10p4:

> [Note: There may be non-atomic writes to atomic objects, for example on intialization and re-initialization. — end note]

Discussion

There was some dissatisfaction with this approach expressed in the June email thread. It is quite possible specifying this would be encroaching on the prerogatives of implementors, who are in any case free to perform operations non-atomically when permitted by the as-if rule. Implementors may also perform initializations atomically, again, when permitted by the as-if rule.

Resolution

Adopt the update proposed by US 168.

CA 15: Intra-thread dependency-ordered-before [Paul]

Section:

1.10p9

Comment:

Batty et al. note that, unlike synchronizes-with, the dependency-ordered before relation can operate within a thread. This was not the intent. Instead, intra-thread operations are covered by the rules applying to execution of a single thread.

Proposal:

Update 1.10p9 as follows:

An evaluation `A` is dependency-ordered before an evaluation `B` if

- `A` performs a release operation on an atomic object `M`, and on another thread, `B` performs a consume operation on `M` and reads a value written by any side effect in the release sequence headed by `A`, or
- for some evaluation `X`, `A` is dependency-ordered before `X` and `X` carries a dependency to `B`.

Resolution:

Adopt the proposal.

CA 22: Control Dependencies for Atomics [Paul]

Sections:

N/A

Comment:

"Control dependencies for atomics
Given the examples of compilers interchanging data and control dependencies, and that control dependencies are architecturally respected on Power/ARM for load->store (and on Power for load->load with a relatively cheap isync), we're not sure why carries-a-dependency-to does not include control dependencies between atomics."

Proposal:

Please clarify.

Discussion

At the time that the memory model was formulated, there was considerable uncertainty as to what architectures respect control dependencies, and to what extent. It appears that this uncertainty is being cleared up, and our hope is that it will be ripe for standardization in a later TR.

Resolution

Not a Defect

US 10: Overlapping Atomics [Lawr ence]

Sections:

1.10/14

Comment:

The definition of a data race does not take into account two overlapping atomic operations.

Proposal:

Augment the first sentence: The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic (or both are atomic and operate on overlapping, but not-identical, memory locations), and neither happens before the other.

Discussion

The premise is incorrect; atomic objects may not overlap. The type argument to the `atomic` template must be a trivially-copyable type (29.5.3/1) and atomic objects are not trivially copyable. The atomic types provide no means to obtain a reference to internal members; all atomic operations are copy-in/copy-out.

Resolution

Not a Defect

US 12: N3074 [Paul]

Sections:
1.10p2, 1.10p14
Comment:
Adapt N3074.
Proposal:
The proposed change to 1.10p2 has been adopted. However, the proposed change to 1.10p14 has not, so the following modification needs to be made:

> The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior. [Note: It can be shown that programs that correctly use simple locks and `memory_order_seq_cst` operations to prevent all data races and that use no other synchronization operations behave as ~~the executions of~~ if the operations executed by their constituent threads ~~were~~ are simply interleaved, with each ~~observed value~~ value computation of an object being the ~~last value assigned~~ last side effect on that object in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data–race–free programs, and data–race–free programs cannot observe most program transformations that do not change single–threaded program semantics. In fact, most single–threaded program transformations continue to be allowed, since any program that behaves differently as a result must perform an undefined operation. — end note]

Resolution:
Adopt the proposal.

US 168, US 171: Initializing Atomics [Lawr ence]

Sections:
29.6/4
Comment:
The definition of the default constructor needs exposition.
Proposal:
Add a new paragraph: A::A() = default; Effects: Leaves the atomic object in an uninitialized state. [Note: These semantics ensure compatiblity with C. --end note]

Number
US 171
Sections:
29.6/6
Comment:
The atomic_init definition "Non-atomically assigns the value" is not quite correct, as the atomic_init purpose is intialization.
Proposal:
Change "Non-atomically assigns the value desired to *object." with "Initializes *object with value desired". Add the note: "[Note: This function should only be applied to objects that have been default constructed. These semantics ensure compatibility with C. --end note]"

Discussion

Adopt as recommended, but with more clarity.

Resolution

After 29.6 [atomics.types.operations] paragraph 4, add a new function description as follows.

```
A::A() = default;
```

Effects: Leaves the atomic object in an uninitialized state. [*Note:* These semantics ensure compatiblity with C. —*end note*]

Edit 29.6 [atomics.types.operations] paragraph 7 as follows, and then move it to just after the paragraph inserted above.

Effects: ~~Non-atomically assigns the value desired to *object.~~ Initializes *object with value desired. This function shall only be applied to objects that have been default constructed, and then only once. [*Note:* These semantics ensure compatibility with C. —*end note*] Initialization shall happen before (1.10) other operations on the object. [*Note:* Concurrent access from another thread, even via an atomic operation, constitutes a data race. —*end note*]

US 38: Generalized Infinite Loops [Clark]

Sections:

1.10p16, 6.5p5

Comment:

The statement that certain infinite loops may be assumed to terminate should also apply to go-to loops and possibly infinite recursion. We expect that compiler analyses that would take advantage of this can often no longer identify the origin of such a loop.

Proposal:

Insert new paragraph following 1.10p16:

The implementation is allowed to assume that any thread will eventually do one of the following:

- terminate,
- make a call to a library I/O function,
- access or modify a volatile object, or
- perform a synchronization operation.

[*Note:* This is intended to allow compiler transformations, such as removal of empty loops, even when termination cannot be proven. — *end note*]

Delete paragraph 6.5p5:

~~A loop that, outside of the *for-init-statement* in the case of a `for` statement,~~

- ~~makes no calls to library I/O functions, and~~
- ~~does not access or modify volatile objects, and~~
- ~~performs no synchronization operations (1.10) or atomic operations (Clause 29)~~

~~may be assumed by the implementation to terminate. [*Note:* This is intended to allow compiler transformations, such as removal of empty loops, even when termination cannot be proven. — *end note*]~~

Resolution:

Adopt the proposal.

GB 8, US 9, US 11: Mutexes versus Locks [Lawrence]

Sections:

1.10/4,7

Comment:

The text says that the library "provides ... operations on locks". It should say "operations on mutexes", since it is the mutexes that provide the synchronization. A lock is just an abstract concept (though the library types unique_lock and lock_guard model ownership of locks) and as such cannot have

operations performed on it. This mistake is carried through in the notes in that paragraph and in 1.10p7.

Proposal:

Change 1.10p4 as follows:

> "The library defines a number of atomic operations (Clause 29) and operations on mutexes (Clause 30) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics. [Note: For example, a call that acquires a lock on a mutex will perform an acquire operation on the locations comprising the mutex. Correspondingly, a call that releases the same lock will perform a release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on A. "Relaxed" atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. -- end note]"

Change 1.10p7 as follows:

> "Certain library calls synchronize with other library calls performed by another thread. In particular, an atomic operation A that performs a release operation on an atomic object M synchronizes with an atomic operation B that performs an acquire operation on M and reads a value written by any side effect in the release sequence headed by A. [Note: Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. -- end note] [Note: The specifications of the synchronization operations define when one reads the value written by another. For atomic objects, the definition is clear. All operations on a given mutex occur in a single total order. Each lock acquisition "reads the value written" by the last lock release on the same mutex. -- end note]"

Number

US 9

Sections:

1.10/4

Comment:

The "operations on locks" do not provide synchronization, as locks are defined in Clause 30.

Proposal:

Change "operations on locks" to "locking operations". (Covered by GB 8.)

Number

US 11

Sections:

1.10/7

Comment:

There is some confusion between locks and mutexes.

Proposal:

Change "lock" when used as a noun to "mutex". (Covered by GB 8.)

Discussion

Adopt the wording of GB 8, but with additional use of "mutex" for clarity.

Resolution

Edit 1.10 [intro.multithread] paragraph 4 as follows.

> The library defines a number of atomic operations (Clause 29) and operations on mutexes (Clause 30) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics. [*Note:* For example, a call that acquires a lock <u>on a mutex</u> will perform an acquire operation on the locations comprising the mutex. Correspondingly, a call that releases the same lock will perform a release operation on those same locations. Informally, performing a release operation on *A* forces prior side effects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on *A*. "Relaxed" atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. —*end note*]

Edit 1.10 [intro.multithread] paragraph 7 as follows.

> Certain library calls *synchronize with* other library calls performed by another thread. In particular, an atomic operation *A* that performs a release operation on an atomic object *M* synchronizes with an atomic operation *B* that performs an acquire operation on *M* and reads a value written by any side effect in the release sequence headed by *A*. [*Note:* Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. —*end note*] [*Note:* The specifications of the synchronization operations define when one reads the value written by another. For atomic objects, the definition is clear. All operations on a given ~~lock~~ <u>mutex</u> occur in a single total order. Each <u>mutex</u> lock acquisition "reads the value written" by the last <u>mutex</u> lock release on the same mutex. —*end note*]

GB 15: Control Dependencies and Dependency Ordering [Paul]

Number
GB 15
Sections:
N/A
Comment:
"Given the examples of compilers interchanging data and control dependencies, and that control dependencies are respected on Power/ARM for load->store (and on Power for load->load with a relatively cheap isync), we're not sure why carries-a-dependency-to does not include control dependencies between atomics."

Discussion

At the time that the memory model was formulated, there was considerable uncertainty as to what architectures respect control dependencies, and to what extent. It appears that this uncertainty is being cleared up, and our hope is that it will be ripe for standardization in a later TR.

Resolution

Not a Defect

CH 2: Observable Behavior of Atomics [Lawr ence]

Sections:
1.9 and 1.10
Comment:
It's not clear whether relaxed atomic operations are observable behaviour.
Proposal:
Clarify it.

Discussion

Normatively, the behavior is well-defined. We add a clarifying note.

Resolution

Edit paragraph 8 as follows.

> The least requirements on a conforming implementation are:
>
> - Access to volatile objects are evaluated strictly according to the rules of the abstract machine. [*Note:* Atomic objects may also be either volatile or non-volatile. —*end note*]
> - At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.
> - The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.
>
> These collectively are referred to as the observable behavior of the program. [*Note:* more stringent correspondences between abstract and actual semantics may be defined by each implementation. —*end note*]

Controversial Issues Discussed in June Email Thread

CA 17: 1.10p12 phrasing

The last note of 1.10p12 refers to data races “as defined here”. Batty et al. recommend that this change to “as defined below”. Given that data races are defined in 1.10p14, it is easy to argue for “below”, however, it is equally easy to argue that the scope of “here” is the whole of 1.10.

TBD National-Body Comments

Later revisions of this paper will also include the following national-body comments:

1. CA 18: “Non-unique visible sequences of side effects and happens-before orderings”. TBD Benjamin Kosnik and Michael Wong.
2. CA 19: “Alternative definition of the value read by an atomic operation”. TBD Benjamin Kosnik and Michael Wong.
3. CA 20: “Reading from last element in a vsse?” TBD Benjamin Kosnik and Michael Wong.
4. GB 5: “The evaluation of function arguments are now indeterminately sequenced, rather than left completely unspecified, as part of the new language describing the memory model. A clearer example of unspecified behavior should be used here.” TBD Clark Nelson.
5. GB 10: covering relationship of inter-thread happens before and and sequenced before. (Item C in Appendix 1 of ballot comments.) TBD Clark Nelson.
6. GB 11: covering relationship of `memory_order_consume` and modification order. (Item E in Appendix 1 of ballot comments.) TBD Benjamin Kosnik and Michael Wong.
7. GB 12: covering whether certain memory-ordering cycles are permitted. (Item F in Appendix 1 of ballot comments.) TBD Benjamin Kosnik and Michael Wong.