Document: WG 14/N1584
Date: 14-Oct-2011

We would like to discuss bringing the C memory model in line with the C++11 memory model. We have also spotted some other problems in the current draft and we have suggestions as to how they might be fixed. I (Mark Batty) am available to teleconference for part of the DC meeting if that would be appropriate.

Changes were made to the C++11 memory model in the final iterations of the standardisation process to fix some problems that our work brought to light. We made a mathematical semantics of the memory model of the language and some tools to test small examples with. Although many important problems were fixed, there was not enough time before the finalisation of the standard to fix everything. Looking at N1569, it appears that many of the changes that were made were not carried over to C, so in the second part of this document we list the textual changes that are needed. First, we describe the new issues that we would like to discuss at the meeting in Washington DC next month.

--- New issues to discuss ---

1.1 Visible sequences of side effects are redundant
1.2 SC fences do not restrict modification order enough
1.3 Should locks provide intra-thread synchronisation?

--- Changes to bring over from C++11 ---

2.1 Happens before cannot be cyclic
2.2 Coherence
2.3 Malloc and free in the memory model
2.4 A joke fragment remains in a footnote
2.5 The mutex specification
2.6 Do conflicting un-sequenced accesses have undefined behaviour?

--- 1.1 Visible sequences of side effects are redundant ---

We have mathematically proved that a simplification can be made to the memory model as it is specified in the final draft of the C++11 standard. Essentially, the restriction defining 'visible sequence of side effects' (vsse) is redundant and can be removed with no ill effects. The main motivation for doing this is that the current

restriction is misleading. 5.1.2.4p22 defines vsse's:

> 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.

The wording of this paragraph makes it seem as if the vsse identifies the writes that an atomic read is allowed to read from, but this is not the case. There can be writes in the vsse that cannot be read due to the coherence requirements (to be included in C, 1.10p15 through 1.10p18 in C++ N3291, the latest version I have access to). Consequently this is even more confusing than it at first appears.

We propose changing 5.1.2.4p22 to the following:

> The value of an atomic object M, as determined by evaluation B, shall be the value stored by some side effect A that modifies M, where B does not happen before A.

With a note to remind the reader of the coherence requirements:

> NOTE The set of side effects that a given evaluation might take its value from is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below

If the committee is concerned about allowing a differing text from C++11, then a note could be added to assure the reader:

> NOTE Although the rules for multi-threaded executions differ here from those of C++11, the executions they allow are precisely the same. Visible sequences of side effects are a redundant restriction.

--- 1.2 SC fences do not restrict modification order enough ---

C1x seems to omit the restriction imposed in C++11 in 29.3p7 (from N3291):

> For atomic operations A and B on an atomic object M, if there are memory_order_seq_cst fences X and Y such that A is sequenced before

> X, Y is sequenced before B, and X precedes Y in S, then B occurs
> later than A in the modification order of M.

Furthermore, it seems that both C1x and C++11 are missing the
following two derivatives of this rule:

> For atomic modifications A and B of an atomic object M, if there is
> a memory_order_seq_cst fence X such that A is sequenced before X,
> and X precedes B in S, then B occurs later than A in the
> modification order of M.
>
> For atomic modifications A and B of an atomic object M, if there is
> a memory_order_seq_cst fence Y such that Y is sequenced before B,
> and A precedes Y in S, then B occurs later than A in the
> modification order of M.

--- 1.3 Should locks provide intra-thread synchronisation ---

Most of the C++ standard, synchronisation is used exclusively
inter-thread, so in particular, synchronisation can't be used to avoid
undefined behaviour arising from conflicting un-sequenced memory
accesses, e.g.:

```
(x = 1)==(x = 2)
```

Firstly, C does not define this sort of thing as undefined
behaviour. Is this intentional? Secondly in C++ locks can currently be
used to fix up such programs and avoid undefined behaviour, e.g.:

```
(lock; x = 1; unlock; x)==(lock; x = 2; unlock; x)
```

The reason not to allow this sort of synchronisation in general is, I
believe, because it disallows some single threaded compiler
optimisations. Is intra-thread locking intended to be defined and
usable?

--- 2.1 Happens before cannot be cyclic ---

C++11 forbids happens before from being cyclic, but this change has
not made its way into C1x. In order to fix this, the following
sentence (taken from C++ N3291, 1.10p12) should be added to
5.1.2.4p18:

> The implementation shall ensure that no program execution

demonstrates a cycle in the ,Äúhappens before,Äù relation.

NOTE This cycle would otherwise be possible only through the use of consume operations.

--- 2.2 Coherence ---

The memory model described in N1569 matches an older version of the C++0x memory model, one that allowed executions that were not intended by the designers. We recommend matching the C++11 text by removing the sentence starting 'Furthermore' in 5.1.2.4p22, and including the following paragraphs in section 5.1.2.4 (Taken from C++ N3291, 1.10p15 through 18):

If an operation A that modifies an atomic object M happens before an operation B that modifies M , then A shall be earlier than B in the modification order of M .

NOTE The requirement above is known as write-write coherence.

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

NOTE The requirement above is known as read-read coherence.

If a value computation A of an atomic object M happens before an operation B on M , then A shall take its value from a side effect X on M , where X precedes B in the modification order of M.

NOTE The requirement above is known as read-write coherence.

If a side effect X on an atomic object M happens before a value computation B of M , then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M.

NOTE The requirement above is known as write-read coherence.

--- 2.3 Malloc and free in the memory model ---

The synchronisation afforded to malloc and free is missing some vital

ordering, and as the definition stands it makes little sense and creates cycles in happens before. C++11 includes a total order over the allocation and deallocation calls, which fixes the problem and seems to give a sensible semantics. From 18.6.1.4p1 in N3291:

> Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

Unfortunately, there is a typo here. Happens before edges are not transitively closed in to the happens before relation, but the edges here are meant to be. Instead the sentence above should create a synchronises with edge. In light of this, we suggest removing the last two sentences from 7.22.3p2 and replacing them with:

> Calls to these functions that allocate or deallocate a particular region of memory shall occur in a single total order, and each such deallocation call shall synchronise with the next allocation (if any) in this order.

--- 2.4 A joke fragment remains in a footnote ---

C1x seems to have inherited part of a joke from C++, which ought to be removed or made whole and annotated as such. Originally, C++0x had the footnotes:

> "Atomic objects are neither active nor radioactive" and
> "Among other implications, atomic variables shall not decay".

The first is pretty clearly a joke, but it's not obvious that the second doesn't have some technical meaning, and that is the one that remains in C1x in 7.17.3p13.

--- 2.5 The mutex specification ---

The C1x specification of mutexes is missing the total order over all the calls on a particular mutex. This is present in C++11. The following is from 30.4.1.2p5 in N3291:

> For purposes of determining the existence of a data race, these behave as atomic operations (1.10). The lock and unlock operations on a single mutex shall appear to occur in a single total order. [Note: this can be viewed as the modification order (1.10) of the

mutex. ,Äî end note]

The synchronisation in 7.26.4 is defined in terms of some order over these calls, even though none is specified, for instance 7.26.4.4p2 reads:

> Prior calls to mtx_unlock on the same mutex shall synchronize with this operation.

This seems like simple omission. We suggest adding a new paragraph to 7.26.4 that matches C++11:

> For purposes of determining the existence of a data race, mutex calls behave as atomic operations. The lock and unlock operations on a single mutex shall appear to occur in a single total order.
>
> NOTE This can be viewed as the modification order of the mutex.

--- 2.6 Do conflicting un-sequenced accesses have undefined behaviour? ---

See 1.3.