WG21/N2275 **J16/07-0135** **2007-05-02**

Nick Maclaren
University of Cambridge Computing Service,
New Museums Site, Pembroke Street,
Cambridge CB2 3QH, England.
Email: nmm1@cam.ac.uk
Tel.: +44 1223 334761
Fax: +44 1223 334679

Non-Memory Actions (Library)

1.0. Introduction

This is another go to try to explain some aspects of the non-memory actions, and how they cause trouble with concurrency. The original paper was N1947, which contains a **lot** more detail, but was effectively ignored because of incomprehensibility. This document has quite a lot of wording in common with N2273, to make them both self-contained.

It relates to standard- or implementation-defined behaviour and side-effects that are not clearly modifications to or accesses of a memory location, hidden or otherwise. It includes objects that have strange properties, of the sort that are not normal for a simple object stored in memory.

Note that this is **not** a purely theoretical problem, as I have seen it cause trouble on a fair number of systems. A great many implementations (at all levels, from hardware up to libraries) remember to honour the synchronisation and serialisation properties for memory accesses, but tend to forget about the non-memory ones. This is not helped by most languages that mention them effectively specifying them as undefined behaviour, because that makes it almopst impossible for customers to report problems as bugs through the usual support interfaces.

1.1. Examples and the Problem

C++ as of today has relatively few that have any effects outside the library, though several in that, but POSIX and C99 introduce many more. Here is a partial list of the most relevant ones, of which only the first three are actually required in C++ implementations today, but where several others are **required** for C99 and POSIX 2001 support (see later).

1: C++ exceptions. These are objects with some extra restrictions, a certain amount of hidden state, and the property that rethrowing an exception could be regarded as a modification action on that hidden state.

2: Timestamps etc. <`ctime`> is the only one that I know of in C++, but POSIX adds more. Quite a lot of code gets really, really unhappy when time appears to run backwards, as can happen even with sequential consistency if the timestamp extraction is not synchronised with the memory accesses.

3: C++, C (and POSIX) I/O. This is a soluble problem, but will become an insoluble one if it is ignored – see later.

4: C++ (and POSIX) signals. The existing C++ standard specifies the syntax and some vague intent, but leaves the semantics undefined. POSIX relies on the undefined handling.

5: IEEE 754 exception flags and modes, or at least the subset of features supported in C99.

6: Code-generated interrupts (e.g. `SIGFPE`), whether handled as C++ exceptions, signals or otherwise. The Language Independent Arithmetic standard should also be mentioned.

7: Some of POSIX concurrency modes, most especially PTHREAD_CANCEL_ASYNCHRONOUS, signal handling and the thread scheduling options.

8: WG14 Technical Report 18037 provides some hardware I/O register support for C99, but I have no idea what C++'s position on it is.

9: Implementation-specific versions of the above, which are probably more widely used than the standardised ones, and other implementation-modes, states, counts and flags. A class that is extremely important to many users and often gets forgotten is hardware counters (including IEEE 754 exception counts).

Some of those are apparently specific to a particular thread, but that is deceptive. POSIX allows the handlers of code-generated interrupts to be handled in a newly created thread, and there are other, more esoteric, issues that can affect some people. E.g., for arcane hardware and operating system reasons, some issues can impact a whole system (let alone a process!); this is not obvious even to the library implementor, let alone the developer, but is clearly of great importance to real-time and embedded developers, and even HPC ones.

There is also the problem that the handler of an event (whether a C++ exception or some other kind of event) may do something that is visible to other threads or a non-memory action may become visible in other ways. For these reasons, such actions need some suitable wording. The remaining sections refer to wording that I think will be needed.

1.1. Proposal

There is obviously no hope of specifying all the above in any detail; many of the areas have been left undefined for good reasons (whether or not you agree with me that the standard should have made them implementation-defined). In N2273, I propose that some informative wording should be included, such as the following::

- *Where an implementation includes extensions that have side-effects (in the most general sense), as far as is practical an implementation should define the synchronisation of such behaviour in terms of the basic memory model.*

But the library also needs to specify some primitives that are clearly documented to ensure that non-memory actions are synchronised with memory ones, especially for those actions that are not clearly associated with any particular memory location. This could then be used by other parts of the library as well as programmers. I can see little point in having separate "low-level" versions, as non-memory actions rarely are clearly "acquire-like" or "release-like". For example, the same IEEE 754 exception flag can be set multiple times, unsequenced. So everything here should be assumed to be *ordered* in the memory model sense, because

Upon thinking it over, what I said in N1947 doesn't make sense in the context of the current memory model. What is needed is some primitives like the following:

- A synchronisation call that ensures that the current thread's view of memory and non-memory actions is consistent (as defined by the implementation!) There would be no implication on ordering of semi-asynchronous events, or on other threads. This is a classic fence.
- A synchronisation call between threads A and B that ensures that their views are consistent and consistent with each other. That would include semi-asynchronous events caused by one and affecting the other or that affect both (e.g. `pthread_kill`). This is a pairwise barrier.
- A synchronisation call between all threads that ensures that their views are consistent and consistent with each other. That would include all semi-asynchronous events (i.e. all threads would see the effect or none). This is a classic barrier.

In the latter two cases, I can see uses both for versions that wait for the other thread or threads, and for ones that don't. Producing precise wording for the latter may be a bit tricky, but I think is worth trying.

2.0. Specific Issues

2.1. C++ Exceptions

I mention this in N2273. I don't think that the library issues are significant enough to consider separately. What I propose there is:

Proposal: *It should be undefined behaviour to access a C++ exception from any thread except the one in which it was thrown.*

2.2. Timestamps

I think that functions like `clock()` and `time()` should imply fences of some sort (e.g. the one I mention above). A lighter weight approach is probably possible, but is it worth the effort? What I can witness is that merely making them "read-like" actions is likely to cause confusion, and not saying anything in the standard unquestionably will lead to implementors and users interpreting it incompatibly.

2.3. I/O etc.

This is a problem. The problem isn't technical, so much as POSIX has taken some decisions that have catastrophic effects on performance and usability, and I can witness that this already causes serious confusion. I will try to write another paper on the details.

The performance problem is that POSIX has specified that the I/O functions are thread safe, which means that a large number of threads all reading `stdin` or writing small units to `cout`, `cerr` or `clog` will run like a drain – and there are reasonable programs that do that. But, more importantly, those are used for diagnostics in all sorts of places, and POSIX says that is defined even in threaded code. If C++ forbids threads to write to `cerr` or `clog` without explicit synchronisation, then a lot of existing non-portable but conforming code will become undefined.

There are many usability problems, but threading introduces several new ones. One is that it is not specified what the unit of interleaving is. POSIX defines "thread-safe" as "*A function that may be safely invoked concurrently by multiple threads*", which does not help with deciding whether (say) `printf` should be atomic at the top level, the basic built-in scalar value level or the `fputc` level. C++ I/O is no different in this respect – for example when writing a `complex` object to `cout`! Consider the following:

```
std::complex<double> a = std::complex<double>(1.23,4.56),
     b = std::complex<double>(9.89,0.12);
```

Thread A: `std::clog << a;`

Thread B: `std::clog << b;`

Another is that I/O buffers are inherently asynchronous objects, and the rules for them are unclear.

This is all made a **lot** more complicated by wording such as in "*27.3.1 Narrow stream objects ... The object cerr controls output to a stream buffer associated with the object stderr, declared in*

<*cstdio*>." Does that imply that they must honour the same "thread-safety' rules or doesn't it? And does that mean that the implementation may use different rules for C and POSIX and C++, or must they be the same?

Specifying all of this better is known technology, and not even hard, but C++ should not attempt to follow POSIX either in both going overboard on demanding "thread-safety' or in leaving that largely unspecified. The following proposal is almost certainly redundant, as it will already be on board, but I shall make it anyway.

Proposal: *This should be regarded as an outstanding work item, to clarify what the C++ standard defines.*

2.4. IEEE 754 Flags and Modes

This is too complicated an issue for me to cover here, but I will try to write another paper on the details.

The current draft of C++0X is seriously confusing, in that it includes the function interfaces of C99 but not the language changes. And, despite being in the library, C99 <`fenv.h`> is really a compiler feature, with its primary consequences being on code generation.

I believe that this needs to be clarified, though I am not at all sure what should (or even can!) be said. What I can guarantee is that virtually every implementor will choose a different interpretation of the C++0X standard unless **something** is said.

Proposal: *This should be regarded as an outstanding work item, but only to clarify what the C++ standard defines.*

2.5. Non-C++ I/O, SHMEM etc.

There are a lot of I/O-like interfaces that are heavily used within C++, even if not all are actually described as I/O. For example, there is MPI (the Message Passing Interface) so heavily used on clusters, various "SHMEM" interfaces derived from the Cray designs and used in high-performance computing on shared-memory machines, several GUI interfaces, many data-passing protocols used in commercial code and communications, and so on.

All of the respectable ones have their own synchronisation models, and all they really need from the C++ library is a suitable interface to the memory model, as described above. But they really **do** need that. It doesn't really matter whether it is in terms of atomics or separate functions, as long as it has the right semantics and a clear specification.