Doc No:	ISO/IEC JTC1/SC22 N4276 = ISO/IEC JTC1/SC22/WG21 N2420 = INCITS CT22/J16 07-0280	
Date:	2007-09-20	
Projects:	JTC1.22.21.04.01.01 (POSIX), JTC1.22.32 (C++)	
Reply to:	Herb Sutter Microsoft Corp. 1 Microsoft Way Redmond WA USA 98052 hsutter@microsoft.com	Nick Stoughton USENIX Association 2560 Ninth Street #215 Berkeley CA USA 94710 nick@usenix.org

POSIX/C++ Liaison Report

Nick Stoughton, Herb Sutter

1. Memory Model, Alignment & Atomic Data Types

The Austin Joint Working Group have reviewed the proposals in WG21 papers N2334 (J16/07-0194) for the concurrency memory model, N2341 (J16/07-0201) for alignment, and N2381 (J16/07-0241) for atomic types and operations, and believe that this is an appropriate and satisfactory foundation for concurrency. The Austin Group are actively encouraging WG14 to follow up this work with a C binding.

2. Thread APIs

The threads API proposal presented in WG21/J16 paper N2320 for the next revision of ISO/IEC IS 14882 (C++) (aka "C++0x") is intended to be highly compatible with existing threading APIs, including specifically ISO/IEC IS 9945 (POSIX) threads (aka "pthreads"), without exclusively endorsing any single existing API. Note that N2320 is one of several proposals being considered by WG21.

Members of the Austin Group and of WG21 have identified potential differences between POSIX pthreads and the N2320 proposal, and these differences were discussed at length in email, and during the large group discussion at the WG21 Toronto meeting (July 2007). They were also discussed again during the recent Austin Group plenary meeting in Reading, UK (September 2007) where three WG21 members participated by teleconference.

The major difference between the POSIX pthreads design and the N2320 proposal is in the area of thread cancellation. There are three main issues: naming, semantics, and integration. A question was also raised regarding adding additional cancellation points. The following subsections discuss these topics, as well as a statement from the Austin JWG.

2.1 Naming: "Cancellation" vs. "interruption"

The N2320 proposal uses the term "cancellation" for a feature by which a thread can be asked to co-operatively enter exception processing. The long established and well known pthread interfaces included in POSIX use the term "cancellation" for a similar concept, but with semantics that are incompatible with the model in the N2320 proposal. Other established and well known thread interfaces include ISO/IEC IS 23271 (CLI) and Java, which use the term "interruption" for the concept approximately corresponding to the N2320 proposal.

The N2320 proposal should consider renaming "cancellation" to some other name, possibly "interruption." This would remove the naming conflict with POSIX threads and conform to existing naming practice in other major languages and ISO standards.

However, the Austin JWG has expressed concern that this would only be of benefit if WG21 can commit to *never* extend the current list of "cancellation/interruption points" to include blocking I/O (discussed in 2.3 below). And both WGs have concerns that such an interruption facility would be too limited to support shutdown of arbitrary threads, and that this separation of POSIX "cancellation" and C++ "interruption" would lead to two non-interoperable thread shutdown mechanisms for C/POSIX and C++. Each one of these would perform only some of the necessary cleanup actions, potentially making neither fully usable in a mixed language POSIX application.

2.2 Semantics: POSIX pthreads cancellation vs. exceptions and destructors

At the time the POSIX pthread interfaces were designed, some 15 or more years ago, they were primarily designed for a language (namely C) that does not have exceptions or class types. Forward-looking language features like exception handling and class types with destructors were discussed, and the thread cleanup handlers that exist in the 9945 standard today were based in part on the desire to permit class types with destructors and exception handling as it was understood at that time. However, we note that other libraries have encountered difficulties reconciling pthreads cancellation with the way that modern languages now actually support both of these features.

Existing experience includes the following:

- **POSIX pthreads:** Pthreads implements several distinct methods for "thread interruption", including asynchronous signaling, and both synchronous and asynchronous cancellation. The pthreads cancellation model includes the ability to call an arbitrary number of "cleanup handlers" (intended to map into destructors or stack unwinding mechanisms), but does not include any mechanism to abort the cancellation once started. The majority of C libraries and applications choose synchronous thread cancellation as their preferred shutdown mechanism.
- **Boost.Threads:** The N2320 proposal is based, at least in part, on the Boost C++ thread library (aka "Boost.Threads"). The Boost.Threads library does not include cancellation, which we understand is in part because Boost is a pure library that cannot mandate language changes, and Boost have been unable to determine how to map POSIX pthread cancellation onto the C++ exception mechanism without changes to language semantics that involve breaking both source and binary compatibility.
- **Gnu gcc and Solaris pthreads:** The Gnu gcc and Solaris pthreads implementations are two known implementations that attempt to map POSIX pthread cancellation onto C++ exception handling, but both do so at the cost of breaking the exception model (i.e., they no longer conform to ISO C++) because the alternative appears to be that C++ destructors and catch blocks would not be invoked for cancellation which would mean that resources would be leaked.
- **Java and ISO CLI:** Both Java and ISO CLI support exceptions and "using" blocks or "try/finally" conventions that simulate C++ stack-based lifetimes and destructors. Both have diverged from pthreads cancellation semantics in the same ways as the N2320 proposal, favoring an interruption model instead. Neither supports POSIX cleanup handlers in C code called from Java or CLI. Like C++, both environments include facilities to integrate with existing C code (e.g. JNI, P/Invoke), but experience the same issues with thread cancellation.

2.3 Integration

The C++ language shares much in common with its predecessor, C, and has striven to retain compatibility with that language ever since its inception. It is simple in a C++, Java, or CLI application to call functions written in C, and to use their results. Many applications are linked with third party C libraries. From the library perspective, the C++, Java, or CLI caller does not appear in any way different than a C caller would.

In turn, it is becoming common for such libraries to use "helper" threads to implement their functionality, utilizing multi-processor architectures etc.

Integration with existing C functions remains a critical requirement of C++, Java, and CLI, and any threads proposal should take this into account. Thread aware C libraries expect that the threading model in use is that of the underlying OS, and may require recoding to interoperate correctly with non-C callers.

2.4 Additional cancellation points

The current list of "cancellation/interruption points" listed in N2320 is:

- void std::this_thread::cancellation_point()
- template <class ElapsedTime>
  void std::this_thread::sleep(const ElapsedTime& rel_time)
- void std::thread::join()
- void std::condition<Lock>::wait(Lock&)
- template<class Predicate>
  void std::condition<Lock>::wait(Lock&, Predicate)
- bool std::condition<Lock>::timed_wait(Lock&, const utc_time&)
- template<class Predicate>
  bool std::condition<Lock>::timed_wait(Lock&, const utc_time&, Predicate)

The next revision of the N2320 proposal could include additional "cancellation points" that more closely match the concept of POSIX cancellation, in particular for blocking I/O.

The Austin JWG has expressed concern that adding cancellation points in blocking I/O (e.g., file stream read and write) would be a major problem, requiring almost all operating system vendors to implement two distinct forms of thread cancellation or interruption to support both C++ and traditional POSIX applications.

WG21/J16 experts, who include representatives from all major operation system vendors, do not agree with that evaluation. They believe that two threading paradigms will not be needed, and further that if this were so then the problem would already exist with facilities in Java and ISO CLI which correspond directly to the N2320 proposal. Operating System specialists from Sun, IBM, HP and Linux (all represented within the Austin JWG) have all categorically stated that this would cause them problems, and that these problems do indeed exist within Java (with both the JNI and CNI interfaces) and CLI.

3. Notes

3.1 Austin JWG statement

The following is a statement from the Austin JWG:

> Multi-threaded programming at the low level is an inherently system specific operation, closely coupled to I/O, scheduling, process and memory management. To attempt to abstract such a low level concept into a high level language is likely to be fraught with difficulty.
>
> The Austin JWG feels strongly that C++ needs to supply two things to its end-users:
>
> 1. A good memory model, coupled with atomic operations, that permit multi-threaded programs to have predictable behavior.
> 2. A high level parallelism paradigm (possibly including features from OpenMP, or the current Futures proposals).
>
> However, the Austin JWG feels that there is no requirement for a fully portable thread implementation between the two. On a POSIX (or Windows) platform, parallel tasks could be implemented using pthreads. On other operating systems there might be a different thread library used to implement the portable parallel tasks.
>
> If WG21/J16 does not agree with this direction, it is imperative that whatever is implemented is compatible with existing thread implementations. It is also imperative that the functionality in C++0x can be efficiently implemented on top of and integrated with existing commercial system thread implementations.
>
> Liaison representatives from both groups have been meeting and co-operating to resolve this issue, and the Austin JWG welcomes this opportunity to work together. As it is written, N2320 presents a major roadblock for POSIX implementers, and the Austin JWG resolved at the September 4-7 meeting:
>
> > AGREED: Ask ISO/IEC JTC1/SC22 to request that WG21 continue to work with their POSIX liaisons to resolve the difficulties represented in their thread API proposal to the mutual satisfaction of both working groups before any FCD ballot of C++0x commences.
>
> The IEEE Portable Applications Standards Committee (PASC) has recently approved the formation of a new working group to develop a C++ binding to IEEE 1003.1 (ISO/IEC 9945). One item that is clearly in scope for this new project (which will eventually be forwarded through the appropriate channels to become an ISO/IEC IS) is the mechanism for using POSIX threads within a C++ application, in a way that is fully interoperable with C. The Austin JWG encourages WG21 to work with this project, and notes that the majority of those who expressed interest in joining the new working group were also WG21/J16 members.
>
> The Austin JWG believes that this will be a better all round solution for the problem that N2320 is attempting to solve.
>
> The Austin JWG is happy for the opportunity to have had, and to continue, close liaison with the WG21 and J16 groups working on a C++/POSIX binding, and appreciates all groups' ongoing efforts to collaborate with and inform each other.

3.1 Additional WG21 notes

WG21 has not had a meeting since the above Austin JWG statement was produced, but for completeness the authors of this paper are able to add the following notes based on discussion and decisions from recent WG21 meetings:

- WG21 does not agree with the comment that "there is no requirement for a fully portable thread implementation." During WG21's meetings, several SC22 national bodies have repeatedly expressed that they would vote against any revision of C++ that does not include a portable thread implementation. This is clearly an essential feature.
- WG21 has always expressed strong agreement with avoiding incompatibility wherever possible with existing thread implementations. This includes not only compatibility with POSIX pthreads, but also compatibility with other ISO standards (e.g., ISO CLI) and industry de facto standards (e.g., Java) as well as the broad array of operating systems and environments that C++ serves.
- WG21 is very happy that the N2320 proposal is already (and by intent) compatible with POSIX pthreads, both in semantics and in efficiency, thanks to much effort including feedback received during the close liaison with the Austin JWG members. To avoid confusion, WG21 will consider renaming "cancellation" to "interruption" in its proposal (see 2.1 above) which would render the proposal even more consistent with POSIX and all other ISO standards.
- The WG21 proposal currently does not attempt to directly specify how POSIX pthreads cancellation interacts with C++ exceptions and class types with destructors (see 2.2). However, there is concern that the next C++ standard needs to provide sufficient guidance to enable portable and safe C++ code for at least simple calls to I/O operations. WG21 is open to developing this guidance in close liaison with another body, such as the Austin JWG or the PASC working group on C++/POSIX bindings.
- WG21 is happy for the opportunity to have had, and to continue, close liaison with the Austin JWG and the PASC working group working on a C++/POSIX binding, and appreciates all groups' ongoing efforts to collaborate with and inform each other.