Document number: N1883=05-0143
Date: 2005-08-29
Project: Programming Language C++, Library Subgroup
Reply to: Kevlin Henney <kevlin@curbralan.com>
Revision: 1

Preliminary Threading Library Proposal for TR2

I. Table of Contents

II. Motivation and Scope

A. The Need for Multithreading Support in Standard C++

Perhaps one of the most often requested features for inclusion in the C++ standard library — and, increasingly, the most noticeable by its absence — is support for multithreading. Multithreading is an increasingly common requirement and feature of modern software, a trend that is set to continue with the use of threading in frameworks, its availability in operating system APIs, and more direct support for threading at the hardware level.

Multithreaded programming is fraught with many challenges, and can rightly be considered something that the majority of programmers should steer clear of. However, C++ has a reputation as a language that embraces all levels of software architecture, from end-user GUI applications all the way down to the metal. Building on the portability of C and its own now-stable standard, C++ also has a reputation for portability. It is in multithreaded programming that C++ is falling short of expectations set by its reputation.

Programming support for multithreading is available as a commodity on platforms such Microsoft's .NET, for which various language mappings and extensions exist, and Sun Microsystems' Java, which enjoys binary compatibility across many operating system and hardware platforms. These higher-level, commercial platforms offer a certain level of portability where the threading API and model is, to most intents and purposes, invariant. For C, the picture is a little more diverse, with the POSIX threading API as the dominant de jure standard supported on many platforms and the Win32 threading API the de facto standard on Microsoft Windows. There are other APIs available, but these two are the ones most thread-aware programmers will run across and spend time with. The picture for C++ is even more diverse than that of C: C++ programmers will use the C APIs directly, or they will create their own thread wrappers (an activity that is now almost as popular as writing string classes used to be), or they will use existing wrapper libraries (e.g. Boost.Threads [Boost.Threads], MFC, Rogue Wave's Threads.h++ [Threads.h++]).

There is an expectation that standard C++ should at least acknowledge multithreading in its memory model. There is also a strong motivation that it should go further than this and provide some library facilities to support coarse-grained multithreaded programming. The diversity in existing practice for threading libraries is potentially an issue, but it would be a greater benefit to C++ programmers to standardize a threading library than not to.

B. Support for Multithreading in Language and Library

A conservative approach is taken in this proposal by recommending a solution that is based on extending the standard library rather than the core language. Although there is research experience in extending the core language (e.g. Concurrent C++ [Gehani+1989] and µC++ [uC++]), adding programmable threading facilities through library offers a less intrusive and less contentious approach that draws on a great deal of experience. This does not mean that all aspects of multithreading can be addressed by a library, but that the subset of features covered by this paper are more likely to be accepted as library than as language extension. It is possible to identify three basic levels of interest and support:

- *Level 0*: A well-defined memory model that addresses the question of concurrent access of memory locations and defines atomic, lock-free operations on primitive types. This affects the core language definition and the part of the library concerned with language support.

- *Level 1*: A standard set of library primitives for launching and managing threads and synchronizing access in the presence of threads. This level encapsulates the underlying platform, but is expressed as a library solution with some support from the core language. No more than Level 0 is assumed.
- *Level 2*: Higher-level facilities, such as message queues and thread pools, that hide the basic nuts and bolts of thread execution and synchronization behind more task-oriented interfaces. In principle there are no platform dependencies at this level, only a dependency on the features in Level 1, so were this level to be standardized it would affect only the library.

High-level threaded programming models and utilities are ultimately desirable, but must be built on a more primitive and portable layer. This paper therefore focuses on library primitives (Level 1) rather than higher-level facilities. It would be reasonable to also standardize Level-2 facilities, but they are considered to be a separate and additional consideration, layering on top of the library primitives. Defining a suitable memory model for threading is the subject of a separate proposal [Boehm+2005]. Until progress is made on the memory model and the library primitives, extending the proposal with higher-level facilities must be considered premature.

In considering a threading library for C++ based on a fairly standard set of primitives, there are essentially two parts to such a library — the actual threading part, i.e. the part of the library concerned with the actual threading and the part of the library concerned with locking — and it is worth considering these separately:

1. *Thread execution*: The creation, execution, and management of separate threads of execution. Whether and to what degree library primitives are offered to support manipulation of thread execution priorities, specification of scheduling model (e.g. FIFO or round-robin), and thread cancellation is currently open for discussion, and depends largely on whether a standardized library adopts a pure common subset approach, a common subset with optional extensions, or a required superset approach.
2. *Synchronization*: This is the traditional domain of locking-based abstractions. Atomic operations for lock-free programming are considered to be Level-0 primitives, i.e. part of the memory model. There is some range of variation in what locking primitives are supported (e.g. binary semaphores, counting semaphores, mutexes, condition variables, event variables, reader–writer locks) and in the way that they are supported (e.g. mutex reentrancy). A library needs to offer a reasonable minimum set of synchronization primitives and a way of dealing with the feature variability within each type.

The two parts are obviously related in terms of execution, but there is no necessary relationship between them in terms of interface dependencies. The

primary focus of this draft of the proposal is on the threading API model rather than on the synchronization API, although some brief notes on synchronization are included.

C. Design Goals

A standard C++ threading library should aim to be that primitive and portable layer rather than a C library, no matter how standard or well known a given C API is. It is most likely that any threading library implementation would build on an existing C threading API, but that is a matter for the library implementation rather than the user of the library. The design style of a threading library should reflect a style founded on modern C++ idioms, unchained from the view of thread objects as C API wrappers.

In principle, if a higher-level set of facilities were also offered as part of a standard threading library, they should be fully and portably implementable in terms of the library primitives. This is not to say that they would be required to be implemented in terms of them, but that they could be (cf. the relationship between I/O streams and the C standard I/O facilities).

For the C++ standard library a reasonable goal for both the threading and the synchronization parts would be to adopt a generic-programming style of design. A style based on that of the STL would be more in keeping with received modern C++ practice, and offer an open and orthogonal design framework that both library implementors and users could extend.

The goals for this paper are to present a rationale for and an outline of a standard C++ threading library, with a particular emphasis in the current revision on actual threading and less on synchronization. A future revision will outline the specific library components — code and concepts — in greater detail. Discussion and feedback are both welcome and encouraged.

D. Experience and Prior Art

The design of the threading API outlined here is all based on existing practice in C++ and other languages. For example, elements of the design can be considered similar to those in Boost.Threads and other popular thread library implementations. Concepts such as futures are established in the multithreading community, e.g. as IOUs in Rogue Wave's Threads.h+++, and have been standardized in Java [java.util].

The specifics of the design described in this paper have existed in prototype form for a couple of years, and have been implemented in private libraries based on presentations of the design (e.g. [Henney2004]). There is currently an effort underway to create a Boost library based on this design.

III. Impact on the Standard

Beyond a dependency on a suitable memory model, the threading library envisioned would be fairly self-contained, and therefore relatively non-intrusive with respect to its effect on other parts of the library.

There is currently no proposed text for the standard. The text proposed would most likely be in the form of a new library clause. Depending on the scope of concepts defined in the memory model, some changes to the core language may or may not be required.

The proposed library can be implemented in the C++ language as it stands today, so no extensions beyond the guarantees of the memory model are required.

IV. Design Decisions

A. Inheritance versus Delegation Approaches

There have a number of different higher-level threading APIs written and proposed. Some are influenced by object models that are not necessarily appropriate to C++'s own idioms, and some of them suffer from looking too obviously like C API wrappers, in spite of their specific goal of API independence. In many cases the resulting object model is less expressive, although far simpler and safer to use, than the underlying API.

Although the details of many C APIs differ, e.g. between POSIX and Win32, they tend to share a common underlying view of threading best characterized as an asynchronous function callback. A thread-launching function normally takes a function pointer, which defines the thread task, and auxiliary context data, e.g.

```
int thread_create(thread_t *, void *callback(void *), void *context);
```

Although simple, this model is surprisingly powerful, including the notion of a return type for the callback function. This result can be picked up by explicit synchronization with the thread's termination, e.g.

```
int thread_join(thread_t, void **result);
```

Threads that are quite function oriented, rather than those that represent daemon tasks, have a use for such a result value. However, the disadvantages of this API are clear to any practitioner: genericity is achieved by a Faustian deal with the type system, exchanging type safety and cohesion for the unquestioning openness of `void *`; error handling involves return-code football, so by definition any thread-based code that is correctly and robustly written will border on unreadable and any thread-based code that is readable will most likely not be robust.

There are two main alternative approaches to wrapping function APIs with respect to objects: inheritance and delegation.

The inheritance approach is best characterized as endowing an object with an independent thread of execution by virtue of its parentage. Object types are stereotyped as active objects (owning a thread of control) [Schmidt+2000] or passive objects (acted upon by other threads of control). A base class plays the role of an asynchronous command object and a derived class extends the base through what is effectively an asynchronous template method. With a few simplifications (which would be noted in more detail in the talk), something like the following is representative of the active object via inheritance approach:

```
class threaded
{
public:
    void execute()
    {
        thread_create(&handle, run, this);
    }
    void join();
    ...
protected:
    virtual void main() = 0;
private:
    static void *run(void *that)
    {
        static_cast<threaded *>(that)->main();
        return 0;
    }
    thread_t handle;
};
```

There is a simplicity to the inheritance approach that is appealing: one base class and an override is all that is needed to turn a task-oriented type into a threaded-task type, and there is a simple realization of the expression "this task *is a kind of* threaded object". However, this simplicity can be deceptive: the task type is now coupled to one and only model of execution (strict thread creation and ownership) and so its task logic cannot be tested simply and independently of threading, nor can it be easily adapted to other task execution models, such as thread pooling, timer driven, or synchronous execution. The base class is intrusive and presumptuous.

A common pitfall that the implementor of the base class can fall into is the combination of construction with execution. The realization of a threaded object as a form of command object makes a strong distinction between initialization and usage, which is good from a perspective of cohesion and clarity, but also sidesteps the problems of trying to call `virtual` functions in constructors.

The delegation-based approach can be expressed in number of ways. The command object metaphor is still present, but instead of a template method approach (as in the pattern rather than a reference to C++ templates) in a class hierarchy, an executor approach is used, where the act and mechanics of launching a thread are separated from the task to be undertaken in the thread.

The thread task object is no longer an active object type in its own right. A simple expression of its conformance would be through an interface class:

```
class threadable
{
public:
    virtual void execute() = 0;
    ...
};
```

And a simple thread executor can be expressed as a simple concrete class:

```
class threader
{
public:
    void execute(threadable *that);
    void join();
    ...
private:
    static void *run(void *that);
    thread_t handle;
};
```

This clearly represents a more loosely coupled model than the inheritance approach. Alternative executors can be used against a common interface for tasks, whether for testing, timer-based execution, thread pooling, etc. The separation between creation and execution is still worth preserving for clarity, flexibility, and the possibility of separately parameterizing the executor while still keeping the launch interface simple. However, there is more involved in setting up the delegation approach to run, a greater entourage of objects and types is involved. That said, the delegation approach is the one that offers the most flexibility and the greatest openness, especially if it is implemented based on the compile-time polymorphism of templates rather than the runtime polymorphism of inheritance and `virtual` functions.

B. A Generic Approach

There are many metaphors that can be used to consider threads in the context of C++, but not all of these are idiomatic. One of the most natural ways to view threads is not primarily as classic entity or controller objects but as functions. The lack of object orientation in the C threading approach is not the problem: it is ultimately the lack of cohesion and safety that are its weaknesses from a C++ programmer's perspective. The function provides the metaphor, and in C++ this extends naturally to include objects in the form of function objects. To constrain a C++-threading model to work only in terms of function pointers would be needlessly restrictive.

Therefore a delegation-based approach would most naturally be based on nullary function objects, which are by definition copy constructible, or function pointers. The function objects can be composed through binders or instantiated

at will from custom task types, whatever is appropriate for the task. This model offers the greatest familiarity and the least invasiveness.

The function metaphor can be taken further. The executor can be considered a unary function object that operates on a function, adapting a plain nullary function object as an asynchronous task:

```
class threader
{
public:
    template<typename nullary_function>
      void operator()(nullary_function threadable);
    ...
};
```

A further loosening in the design is possible by decoupling the executor from retaining the state of the tasks it has launched: the executor becomes responsible for encapsulating the policy of execution and act of launching for a threaded task [Crahen2002], and nothing more. However to ensure that the user can control the execution, an object can be returned that encapsulates the joining and other capabilities:

```
class threader
{
public:
    template<typename nullary_function>
      joiner operator()(nullary_function threadable);
    ...
};
```

This design has the benefit of keeping the role of the executor focused and the responsibilities of task management out of its interface. The joiner is a separate abstraction that represents such control, with the emphasis on joining with thread termination. The resulting threader is modeless, and joiners are only returned on successful creation.

Looking back over many threading libraries one omission that is subtle but apparent is the loss of the return type. In the C model the asynch function could return a value that could be picked up by the joining code. In typical object-oriented designs the result is lost and replaced with a `void`. Any communication of values following execution must be arranged through alternative synchronization objects. It is a simple to recover and reinstate this capability in the joiner itself:

```
class threader
{
public:
    template<typename nullary_function>
      joiner<result_of<nullary_function>::type>
        operator()(nullary_function threadable);
    ...
};
```

Worth noting in this design is that the return type does not intrude on the threader type as a whole, and the templating is local and with respect to the function-call operator. The same executor, with the same policies, may be used to launch different types of thread task.

The penultimate step in the design of the joiner is to make it a proper future. It blocks and yields a value (or propagating an exception on thread failure) when called. The personality of such a value-returning action can be modeled as a function object. Thus, threadable objects (commands), threader objects (executors), and joiner objects (futures) are all expressed functionally. A threader can considered a higher-order function that takes a nullary function and yields another nullary function.

A thread may terminate as the result of an uncaught exception. The most common and practical model is that a thread, but not its containing process, is killed on the escape of an exception. However, propagating exceptions across threads is unnecessarily tricky and creates more problems than it solves. On the other hand, a thread should not die silently, so the simplest approach is that on joining a failed thread, the joiner throws an exception (one indicating unexpected thread death, not the original thread's uncaught exception) instead of yielding a value. When it launches, a threader can also install an unexpected-exception handler for a thread. For uncaught exceptions, the handler function would be called before actual thread termination.

A related topic to unexpected termination of a thread is cancellation of a thread. This paper does not propose a cancellation model. In fact, this paper actively proposes not to discuss or standardize a cancellation model, leaving it as a rigidly defined area of doubt. The semantics of thread cancellation are subtle at the best of times, but agreeing on how they would be realized in a portable library seems to be a show stopper. Discussions concerning how to standardize cancellation for a C++ binding of POSIX threads have yet to reach a useful conclusion, and appear (repeatedly) to derail any further work on a binding. It is not clear that the standardization within the C++ library would fare any better, especially as the portability goal is broader than that of POSIX. At most, standardizing syntax for cancellation (via joiners) should be considered, but beyond a few non-normative notes, the semantics are perhaps best left as implementation defined.

A less contentious issue is that of non-blocking joins. A call to the function-call operator of a joiner would be a blocking call, i.e. it would not return until the target thread was joined or there was some kind of exception thrown to denote failure. For callers that wish to poll rather than block, the joiner would support a conversion to a `bool` or `bool`-like type, which would yield `true` if the thread had been joined and `false` if a call to the function-call operator would block,

The final step in considering a generic approach is to phrase the requirements for each of the roles in terms of concept categories rather than just as concrete class templates. It is this that allows greater extensibility, not just for threadable objects but also for different threaders and joiners. The standard library would

provide standard implementations. At the very least there would be an implementation similar to the one evolved in this section.

Standardizing the concepts as well as some Level-1 library interfaces also offers a model that Level-2 facilities can build on, whether standardized or not, and whether directly or just in terms of style. For example, message queues can follow a conventional named function model, or can follow a function object model:

```
channel(to_send); // enqueue on channel
...
received = channel(); // dequeue value from channel
```

This unifies the idea of sending and receiving with that of function calls, bringing it a little closer, in some ways, to the Ada rendezvous synchronization model and the CSP notion of channels.

In a different form it is also possible to generalize the use of futures as return values to allow a sequence of values to be returned, effectively forming a stream. An appropriate threader would result in a joiner that, on repeated calls, would pull the next value from the queue, rather than the mailbox behavior of the default joiner.

C. Comparison with the Boost.Threads Approach

Although Boost.Threads uses templates in expressing some of its capabilities, it is a relatively closed design with respect to extension, and hence not what would normally be termed generic. However, Boost.Threads does offer a starting point, and with only a few considerations and changes it is relatively easy to evolve the design described above from it.

Threading in Boost.Threads is currently based on the idea that a thread is identified with an object that launches it. This notion is somewhat confused by the idea that on destruction the thread object is destroyed but the thread is not — in other words the thread is not identified the thread object... except when it is.

Another appropriate separation is the distinction between initialization and execution. These are significantly different concepts but they are conflated in the existing thread-launching interface: the constructor is responsible both for preparing the thread and launching it, which means that it is not possible for one piece of code to set up a thread and another to initiate it separately at its own discretion, e.g. thread pools. Separating the two roles into constructor and executor function clears up both the technical and the conceptual issue. The executor function can be reasonably expressed as an overloaded function-call operator:

```
void task();
...
thread async_function;
...
```

```
asynch_function(task);
```

The separation also offers a simple and non-intrusive avenue for platform-specific extension of how a thread is to execute: configuration details such as scheduling policy, stack size, security attributes, etc, can be added as constructors without intruding on the signatures of any other function in the threading interface:

```
size_t stack_size = ...;
security_attributes security(...);
thread async_function(stack_size, security);
```

The default constructor would be the feature standardized, and an implementation would be free to add additional constructors as appropriate.

So far, this leads to an interface that looks something like the following:

```
class thread
{
public:
    thread();
    template<typename nullary_function>
      void operator()(nullary_function);
    void join();
    ...
};
```

Given that the same configuration might be used to launch other threads, and given the identity confusion of a thread being an object except when it's not, we can consider the interface not to be the interface of a thread but to be the interface of a thread launcher, i.e. an executor. A thread initiator can submit zero-argument functions and function objects to an executor for execution:

```
threader run;
...
run(first_task);
run(second_task);
```

A standard library would offer one or more standardized threader types, but as a concept an executor could be implemented in a variety of ways that still conform to the same basic launching interface, i.e. the function-call operator.

Given that a threader can be used to launch multiple threads, there is the obvious question of how to join with each separately run thread. Instead of returning `void`, the threader returns an object whose primary purpose is to represent the ability to join with the completion of a separately executing thread of control:

```
class threader
{
public:
    threader();
    template<typename nullary_function>
      joiner operator()(nullary_function);
```

```
    ...
};
```

The role played by the joiner in this fragment is that of an asynchronous completion token, a common pattern for synchronizing with and controlling asynchronous tasks [Schmidt+2000]. Via the joiner the initiator can poll or wait for the completion of the running thread, and control it in other ways, some of which may be platform specific extensions.

The joiner would be a *CopyConstructible*, *Assignable*, and *DefaultConstructible* handle. Its principal action, the act of joining, can be expressed as a function call:

```
joiner wait = run(first_task);
...
wait();
```

If there are no joiners for a given thread, that thread is considered detached, a role currently played in Boost.Threads by the thread destructor:

```
run(second_task); // runs detached because return value ignored
```

In common with many other objectified threading models, Boost.Threads does not return a value from a completed thread when joined. For many threaded tasks this makes sense, but where a thread is working towards a result then the idea that an asynchronously executed function can return a value for later collection should not be discarded. With a `void`-returning interface the programmer is forced to set up an arrangement for the threaded task to communicate a value to the party that wants the one-time result. This is tedious for such a simple case, and can be readily catered for by making the joiner a proper future variable that proxies the result. This leads to the threader interface looking like the following:

```
class threader
{
public:
    threader();
    template<typename nullary_function>
      joiner<result_of<nullary_function>::type>
        operator()(nullary_function);
    ...
};
```

This is the interface described in the previous section, but evolved via a different route.

For the common default configured threader, a wrapper function, `thread`, can be provided:

```
template<typename nullary_function>
  joiner<result_of<nullary_function>::type>
    thread(nullary_function);
```

And usage as follows:

```
void void_task();
int int_task();
...
joiner<void> wait_for_completion = thread(void_task);
joiner<int> wait_for_value = thread(int_task);
...
int result = wait_for_value();
wait_for_completion();
```

The benefit of programming with futures is that for a certain class of code that would use end-of-thread synchronization to pick up results, programmers are not presented with unnecessarily low-level synchronization APIs. The function-based model is applied consistently.

D. Notes on Synchronization

The variety of operations that a locked synchronization primitive can support (e.g. blocking lock, nonblocking lock, lock with timeout) is large enough and diverse enough that mandating their full support on all locking primitives might be considered too much of an overhead by both users and implementors alike. A hierarchical set of categories defining lockability, similar to the iterator categories in the current standard library, might offer a reasonable approach to addressing this variability. A library can define traits and tag types to allow discovery of details of a given primitive or to choose the best fit for their needs.

On their own, objects satisfying some category of lockability requirements, whether primitives defined in the library or higher-level objects written by the user, could be tedious and error prone to use. Inevitably their use would be wrapped up, using the scoped locking idiom (a specific and common application of the resource acquisition is initialization idiom). The approach is common enough that provision of some scoped-locking-based helpers would most likely be expected by users of a standard threading library. The set of such lockers is potentially unbounded, and is not restricted to the common scope lock: for example, smart pointers that wrap individual function calls can be defined in terms of a stable set of lockability requirements. In this sense, lockers are to lockable objects as algorithms are to iterators. A library could provide some common helper types, but the formalizing of lockability requirements would allow library implementors and users alike a uniform model for extension that would be nonintrusive on existing lockable objects, whether standardized synchronization primitives or other user-defined lockable objects.

In terms of synchronization the Boost.Threads library offers a number of primitives, such as mutexes, but unfortunately couples them to a relatively closed programming model. The synchronization primitives provided do not lead to a general model for locking, forcing the user into using resource acquisition objects when this is not always the best solution. Given that C++ should aim to be the language and API of choice for systems-level work, the restriction of a mandatory scoped-locking interface does not seem appropriate.

A more orthogonal approach based on the capabilities found in common across C++ synchronization libraries — Boost.Threads included — would be a good starting point: lockability and locking strategy are kept separate.

Taking a more generic approach, this means considering a related set of lock categories, e.g. *Lockable* and *TryLockable*. Something like *Lockable* would (syntactically) be little more than `a.lock()` and `a.unlock()`, and something like *TryLockable* would extend with `a.try_lock()`. Boost.Threads has some of this but restricts the interface of the lockable types themselves. With respect to these categories both primitives (e.g. mutex) and higher-level types (other externally locked objects) can be implemented.

There is no dependency on the locking strategies that can be used against lockable objects — scope bound [Schmidt+2000], transaction linked, smart-pointer based [Henney2000], etc. The library would be expected to provide at least scope lockers to simplify the tedious and error-prone business of ensuring that scope-related critical regions are exception safe, but programmers would be free to define additional ones based on specific needs. There is also no requirement to pollute the nested scope of a lockable class with special `typedefs`.

The Boost.Threads synchronization library does not currently appear to adequately satisfy the goals of openness and genericity. This is in part based on its well intended but perhaps mistaken belief that preventing programmers from using locks manually is the key to safe code: preventing users from using multiple threads and locks is the only way to do this. It is easy but annoying to work around the restriction (e.g. dynamically allocate the locker objects and control the lifetime on the heap instead of automatically with respect to scope), but it should not be something that a programmer should have to work around. The Boost.Threads documentation itself admits that its restricted approach is extreme, but a more open and proven — and less extreme — approach is probably a better fit for standardization.

In a language that allows programmer full control of memory allocation and deallocation, it seems inconsistent, although well intentioned, not to offer programmer control of lock acquisition and release. Convenience is not a replacement for completeness, so lock and unlock member functions should be public and reflect a standardized concept. The design most appropriate for standardization is one that recognizes that any use of explicit threading or any form of synchronization is the programmer's responsibility. It should make common tasks easier and less error prone than they might otherwise be — e.g. the provision of scope lockers — but it should also allow the programmer the freedom to implement locking strategies as they see fit rather than force them to work around the API or seek another.

E. Portability

A platform may or may not support preemptive a multithreading model natively. Non-preemptive threading models can be made to look and behave superficially like their preemptive counterparts in certain simple cases, but they

are sufficiently different to work with that this proposal focuses on what is these days presumed to be the default threading approach in programmers' minds, namely preemptive multithreading. As such, threading support in the library is not required to extend to non-preemptive models. Conversely, it may not be possible to use a standard threading library on all platforms that support other C++ standard library features.

Therefore, a question that needs to be resolved in proposing a threading library is to define its kind of conformance. A *freestanding* C++ implementation, by definition, need not include threading facilities in its library since its support for the standard library is minimal. However, it might be considered too much of a burden on what a *hosted* implementation must provide if, to conform to the standard, thread facilities must be supported. There are a number of possible approaches to consider:

1. Add a new kind of implementation, along the lines of *freestanding*, *hosted*, and *hosted with threads* (or some suitable synonym). The *hosted* category can be considered to be the library as it is defined in the current standard (along with any other nonthread-related extensions proposed for the next standard), whereas the *hosted with threads* category would include the *hosted* library plus the threading part of the library.
2. Provide the threading primitives library as an optional part of the library, perhaps as an appendix. Although this might conceptually be similar to approach (1), its spirit and practicalities are subtly different. Optionality implies feature testing, and so some feature testing mechanism is needed, such as macro testing (as used in POSIX), tag types, or traits. However, the inclusion of one optional library might set a precedent and be considered a cue for other optional libraries.
3. Require the definition of threading primitives within the standard library, but leave its viability as a QoI matter: code would compile but would not necessarily run successfully, although its execution would not be undefined. For example, an implementation on a single-threaded platform might chose to throw an exception for any attempted thread creation, or synchronization primitives might be implemented as stateless objects with null implementations of their locking functions. This approach could be complemented by feature-checking mechanisms.

The current paper does not state a preference; feedback on this issue would be welcomed.

F. Platform Variability and Extension

In addition to whether or not threading is supported fully in a given library implementation, there is also per-platform variation in the features supported. Because C++ is still considered a systems programming language, there is an expectation of a close (but no closer than necessary) correspondence between its primitives and the primitives of the platform. This means that any offering of library primitives must strike a balance between being a pure and common

subset of what is common across platforms — but perhaps too small a subset to be useful in real-world applications — and a constructed superset of what is available — demanding more of a library implementor. The superset approach may take an implementation too far from the correspondence between the platform and the library primitives, making constructs that are efficient on one platform indistinguishable from those that are inefficient, because the former can be realized directly and the latter must be constructed and might require elaborate support.

No matter what set of primitives is considered sufficiently portable, it must be recognized that underlying threading platforms are invariably richer, often providing specific features that a programmer may wish to take advantage of, e.g. interprocess synchronization and real-time scheduling. Therefore, it is probable rather than just possible that specific implementations will extend a core standardized threading library. Any chosen library design must take this into account and be open to these kinds of extensions.

For example, when threads are launched there may additional parameters that configure how a thread is run — scheduling policy, stack size, priority, etc — that are dependent on the details of the underlying platform. These extras can be accommodated in additional constructor overloads of a threader. These would be a pure extension that leaves the basic interface untouched.

Another area of variability is that of synchronization. Although mutexes are commonly supported, they tend to appear in different flavors (e.g. reentrant and non-reentrant). Platforms vary in their support for which flavors are supported — some support just one, some support many. Most users would likely expect that these were close to their underlying platform primitives. Types could be resolved at compile time or at runtime.

Another case that may warrant a quality of implementation license is that of deadlock detection. Although convenient, it is not universally supported and not necessarily efficient for all programmer's needs, especially when they have the choice and confidence of not using it. A reasonable quality of implementation constraint would be that, in the event of deadlock, an implementation may either block indefinitely (or until a timeout, if a lock has one specified) or throw an exception to indicate a deadlock condition.

In other cases there is a difference in the platform primitives on offer. For example, Win32 offers event variables but not conditional variables, and pthreads offers condition variables but not event variables. Assuming a mutex primitive, it is possible to implement one in terms of the other without too much infrastructure or surprise. However, in this particular case, condition variables are generally seen as the superior alternative and event variables as too primitive, so it would be reasonable for a standard library to support condition variables but not event variables.

Similarly, synchronization primitives vary in their scope, i.e. available only within a process or also visible to other processes. Win32 treats its *mutex* primitive as having interprocess visibility and its *critical section* primitive, which

has mutex semantics, as being process local only. In this case, because standard C++ does not have a concept of separate processes, it is sufficient to focus on single-process scope for standardization, but acknowledge that a library implementor may reasonably choose to extend the library to accommodate interprocess communication.

V. Acknowledgments

I would like to thank Andrei Alexandrescu, Beman Dawes, Ben Hutchings, Doug Lea, Eric Niebler, and Herb Sutter for encouraging me to put my thoughts on a threading library into a more concrete form, and Ion Gaztañaga for heading up the Boost implementation effort.

VI. References

[Boehm+2005] Hans Boehm et al, "Memory model for multithreaded C++: August 2005 status update", WG21/N1876=J16/05-0136.

[Boost.Threads] William E Kempf, "Boost.Threads", `http://boost.org/doc/html/threads.html`.

[Crahen2002] Eric Crahen, "Executor ", *Nordic Conference on Pattern Languages of Programs*, 2002, `http://www.cse.buffalo.edu/~crahen/papers/Executor.Pattern.pdf`.

[Gehani+1989] Narain Gehani and William D Roome, *The Concurrent C Programming Language*, Silicon Press, 1989.

[Henney2000] Kevlin Henney, "Executing Around Sequences", *European Conference on Pattern Languages of Programs*, 2000, `http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ExecutingAroundSequences.pdf`.

[Henney2004] Kevlin Henney, "More C++ Threading: From Procedural to Generic, By Example", *ACCU Spring Conference*, 2004, `http://www.two-sdg.demon.co.uk/curbralan/papers/accu/MoreC++Threading.pdf`.

[java.util] *J2SE 5.0 Concurrency Utilities*, `http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/index.html`.

[Schmidt+2000] Douglas Schmidt et al, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, Addison-Wesley, 2000.

[Threads.h++] *Threads.h++ User Guide*, `http://www.roguewave.com/support/docs/hppdocs/thrug/index.html`.

[uC++] *μC++ homepage*, `http://plg.uwaterloo.ca/~usystem/uC++.html`.