Enhancing Thread Constructor Attributes – An Exploration of the Design Space

Document number: P0484R1

Revises: P0484R0

Date: 2017-06-18

Reply-to:

Patrice Roy, patricer@gmail.com

Billy Baker, billy.baker@flightsafety.com

Project: ISO JTC1/SC22/WG21: Programming Language C++

Audience: SG1

0. What happened since previous revisions

P0484R0 was presented during the Issaquah meeting, November 2016 in SG1. Brought as a complement to P0320R1, it examined alternatives to adding a constructor to `std::thread` in order to:

- take into account requested features of a thread that have to be fixed at construction time but are difficult to specify in terms of C++'s abstract machine, and
- provide feedback to calling code as to whether the request succeeded or did not.

As discussed in P0484R0, typical users of this feature are not necessarily exception users, which makes a strictly constructor-based approach inappropriate for the problem space.

SG1 feedback can be summarized as such:

- There was positive response to the idea of adding a `make_thread()` function which would take as arguments a set of thread attributes (see P0320R1 for details) and either create a thread respecting that request or signal failure to comply through a non-exception mechanism
- There was a desire for a mechanism that would allow client code to query for thread attributes for a given thread, regardless of how said thread was created. Note that such a feature was presented as part of P0320R1, but feedback was provided during the presentation of P0484R0
- There was interest in exploring the design space in two directions:
  - Using a `make_thread()` function that creates and returns a `std::thread` provided a (potentially empty) set of thread attributes, or signals failure to comply in some way
  - Adding a `std::thread` constructor that takes as argument a native_handle to a thread from the underlying platform, created through mechanisms that are not part of the C++ standard, thus leaving the issue of construction-time specifics to said platform

I. Motivation

Note: most of this section comes from P0484R0, and has been adapted from that proposal. It is provided here as a convenience.

Standard threading support has been part of C++ since C++11. Yet, part of the language's user base does not use standard threading facilities, preferring to rely on tools offered by the underlying platform instead. This is due to a number of factors, some of which are not technical.

One factor that *is* technical is that there are some customization operations these users need to perform on threads that have to be applied when the thread is created. The standard C++ threading facilities, as of this writing, do not offer ways to parameterize what happens at thread construction time. A reported use-case for the equivalent of thread constructor attributes in real-time applications involved supporting the following features (non-exhaustive example for illustrative purposes):

- Thread priority
- Affinity
- Scheduling
- Minimum and maximum stack size
- Create detached

As noted on P0320, many such features correspond to things that are not defined by the C++ standard and only really make sense at thread construction time. Not allowing users to specify these requirements in their code currently keeps them from using standard tools.

II. Summary

This proposal assumes the following:

- There is a desire to standardize a way to parameterize threading startup in order to cover the needs of C++ users for whom the lack of such a mechanism leads to not using standard threading and concurrency tools, and
- A mechanism such as what is being proposed in P0320 will eventually fill that role.

This proposal also supposes that users who request specific thread attributes need these requests to be serviced and will prefer failure to create the thread should this need not be met. From what is known of users currently requesting such a feature, exceptions are not expected to be the most appropriate mechanism to report failure to comply.

Finally, this proposal does not go into the details of how thread attributes are represented, limiting itself to supposing they exist and can be represented in a space-efficient manner.

III. Signaling Failure to Comply

As a reminder, P0320 proposes a per-implementation set of thread constructor attributes. This per-platform characteristic avoids a number of issues raised by the committee in prior discussions.

We consider the use of thread constructor attributes to be a request for a set of specific features on the part of a thread. Should the implementation not be able to service such a request, this can be seen as a failure to comply.

The following sections examine approaches to handle failure-to-comply situations.

III.a) Exceptions

The "easy" or "normal" way to signal failure to comply with a thread constructor attributes request is through exceptions. For example, raising `std::invalid_argument` is possible if a requested feature is not supported on the platform (although failure to compile is probably better in that case) or if the request cannot be serviced for such reasons as lack of resources or insufficient privileges.

Raising a generic exception type does not necessarily convey sufficient information to allow user code to perform diagnostics and handle this failure to comply, however. If the exception raised is intended to convey meaningful information and if the thread constructor attributes are per-platform, it remains unclear what exception types should be raised, particularly when more than one thread constructor attribute is used or when more than one request has not been serviced.

We also note that a significant portion of the expected user base for thread constructor attributes does not use exceptions, which suggests that the efforts invested to make these users more subject to using standard threading facilities might not come to fruition should this approach be preferred.

Note: there is no harm for this proposal if additional constructors are added to `std::thread` in order to accept as additional argument a set of thread attributes. However, since users of threads with construction-time specifics tend to write exception-blind code, we think `make_thread()` should not throw in order to signal failure to comply.

Should `make_thread()` be rejected in favor of platform-specific thread adoption, then the issue of how disappointments during thread creation should be handled also becomes out of scope for this proposal, at least in large part.

Since exceptions raised by constructors is orthogonal to our proposal, we will not explore it further.

III.b) Platform-Specific Thread Adoption

Another option is to add a thread constructor that accepts a native handle as argument. This would let users create their threads using platform-specific facilities and "adopt" them in `std::thread` afterward. In so doing, users would query the equivalent of thread attributes themselves through platform-specific code, and move on to standard threads afterward.

In P0484R0, we stated that this option seemed counterproductive to our objective of making standard threading facilities the preferred avenue for users. Yet, the current document explores this option which gathered some support from SG1 members.

III.c) Factory Function

The avenue we favor relies on a factory function, tentatively named `make_thread()`. The intent of such a function would be to let users provide a request for per-platform thread attributes, and obtain in return both the constructed thread and the thread attributes resulting from that construction.

This would be possible without changing the specification for `std::thread`, as implementations could add a non-specified private constructor for use with `make_thread()`, such as what is done today with `make_shared`/`allocated_shared`. Inspired by the committee's decision to add `basic_string_view` without modifying `std::string`, adding a `make_thread` function that

deals with the attributes and errors would ensure conformance wording would thereafter only apply to any new `make_thread`-like function as well as the attributes rather than all of `std::thread`, making this into a standalone library extension. Implementations might create a new constructor behind the scenes, as an implementation detail of `make_thread`, but that private constructor wouldn't need to be exposed to users.

We are aware that it might be difficult to find "room" for that other constructor unless it is standardized, given that everything of the form `thread(F&&, Args&&...)` is already claimed by the standard constructor. A solution could be to standardize that "this constructor/function does not participate in overload resolution unless the expression F(Args...) is well-formed".

A pseudo-code-like example of the approach would be:

```
template <class F, class ... Args>
  std::thread make_thread(attrs, F && f, Args && ... args) {
    std::thread t(f, std::forward<Args>(args)...);
    if (attrs.detached)
      t.detach();
    // ...
    return t;
  }
void foo() {
#ifdef __cpp_thread_attributes
  make_thread(detached_attr, f, args);
#else
  std::thread(f, args).detach();
#endif
}
```

This example abstracts the type of `attrs` and supposes we only return the newly constructed `std::thread`. Other possible approaches include taking `attrs` by reference, returning a tuple made of the thread and its attributes, and returning a tuple made of the thread and a subset of its attributes corresponding to the requests made through the arguments passed to `make_thread`.

Note: SG1 guidance asked to simplify error management to either it worked or it did not. Providing detailed information as to which attributes could not be applied was not considered essential, and was perceived as added complexity.

Should users eventually ask for a way to diagnose failure-to-comply situations, we suggest writing a proposal dedicated to this question.

Since this proposal assumes that thread constructor attributes requested are seen as mandatory by user code, being unable to service these requests should lead to returning a default (empty) `std::thread`. Failure to create a thread for any other reason than not being able to service the thread constructor attributes requests should be handled as it normally is without `make_thread`.

With a feature test macro, the presence of `thread_attributes` and any `make_thread`-like function could also be tested in a portable manner.

III.d) Querying Thread Attributes

SG1 expressed the desire for an API (tentatively named `get_attributes()`) that would let user code query thread attributes. Options to expose thread attributes include:

- Adding a `get_attributes()` member function to `std::thread`
- Adding a `std::thread::get_attributes(thread_id)` static function, and
- Adding a `this_thread::get_attributes()` function, which could simply be implemented as a call to `thread::get_attributes()` with the appropriate `thread_id`

This raises the question of where and how these per-thread attributes should be stored; we recommend making this unspecified to give vendors maximal freedom to use the underlying platform to their advantage.

IV. Exploring Design Space

SG1 has requested a study of two competing approaches, respectively: platform-specific thread adoption, and a thread attributes-aware factory function. What follows seeks to show the strengths and weaknesses of both approaches, in order to seek SG1 guidance.

IV.a) Impact of Platform-Specific Thread Adoption

SG1 expressed interest in platform-specific thread adoption since this approach seemed to lessen the impact on the existing standard facilities, in part by opening the possibility of not using thread attributes at all.

Should this approach be preferred, possible user code would look like:

```
// include platform-specific headers
#include <thread>
void adoption_example() {
   // create a platform-specific thread, with platform-specific
   // mechanisms and restrictions
   auto native_handle = ...
   if(is_valid(native_handle)) {
      std::thread adopting{ native_handle };
      // ...
```

```
    }
}
```

For user code, there are a number of annoyances with this approach:

- Having to include platform-specific headers reduces the interest of using standard C++ utilities
- Platform-specific threads tend to be functions with fixed signatures such as `void*(*)(void*)` or `unsigned long __stdcall(*)(void*)`. This problem can be mitigated somewhat with lambdas, but forces users to write significantly more type-unsafe code than what standard C++ facilities allow
- Some platforms allow threads to be created but not launched right away, whereas standard threads are assumed to be running once their constructor has completed
- It is possible that faulty user code provides invalid platform-specific handles
- There are ways to "handle" the case of invalid thread handles or handles to threads in an inappropriate state, including:
    - making it undefined behavior, using careful wording that does not let the platform-specific details leak into the standard
    - accepting thread attributes along with the native handle to inform the `std::thread` under construction of that fact, but that somewhat defeats the purpose of using platform-specific thread adoption
- There exists a risk that user code will release or otherwise impact the resources associated with the platform-specific handle, but that risk already exists with `std::thread` due to the exposure of the `native_handle()` once the thread has been created

In terms of design, this approach means exposing clearly what each platform's chosen native handle is in order to let user code create platform-specific threads appropriately, particularly on platforms where there is more than one way to represent a thread. This might be seen as impeding on vendor freedom since it leads to committing to a specific type.

On the upside, platform-specific thread adoption can be, in the simplest case (not considering error management due to invalid handles and such), reduced to adding a small set of constructors to `std::thread`. The need for thread attributes is lessened since the platform-specific effort of specifying construction-time requests can be done through platform-specific utilities, and failure to comply can also be diagnosed through platform-specific utilities.

IV.b) Impact of a Thread Attributes-Aware Factory Function

SG1 also expressed interest in a thread attributes-aware factory function since this approach seemed to allow for fine-grained parameterization of thread construction, without imposing exception handling on user code that might prefer not relying on that mechanism.

Should this approach be preferred, possible user code would look like:

```
#include <thread>

void factory_example() {

   auto th = make_thread( ... ); // attributes, function, args ...
```

```
    if(th.joinable()) {
        // ...
        th.join();
    }
}
```

The main downside of this approach is that it relies on thread attributes, which are tricky to define (see P0320 for a discussion of ways to achieve this). For the sake of this example, we have made `joinable()` the test of validity for the returned `std::thread`, but as has been noted previously, using thread attributes to request a detached thread could be envisioned.

On the upside, this approach removes the need for platform-specific headers and utilities to perform parameterized thread construction. It preserves the user's latitude with respect to the signature of the function used by the `std::thread`, thus avoiding the loss of type safety incurred by relying on such abstract argument types as `void*`.

Another strength of this approach is that it provides the vendor with full control over the thread creation process, including error recognition and management. Expressed otherwise, it avoids all issues related with user code providing a handle to a thread in an inappropriate state.

IV.c) A Word on `get_attributes()`

As mentioned in III.d) Querying Thread Attributes, SG1 has expressed interest in the addition of a mechanism to query the attributes of an existing thread.

Whilst not the focus of the current proposal, we note that using a thread attributes-aware factory function essentially implies that vendors can implement such a querying functionality, if only by storing the requested attributes in such a way as to be able to access them later.

Should platform-specific thread adoption be preferred, it might be possible to obtain information with respect to an existing thread's attributes by querying the underlying platform, but it's unclear whether we could guarantee that this is possible on all platforms. Thus, with this approach, an eventual `get_attributes()` function should be considered optionally provided.

V. Impact on the Standard

Both approaches discussed in this proposal are library extensions. Note that the factory function-based approach depends on the existence of a thread attributes mechanism, and could thus indirectly inherit its own dependencies.

VI. Acknowledgements

Thanks to Vicente J. Botet Escribá for taking the lead on the work for thread constructor attributes and contributing to the discussions that led to this paper, and to the SG14 community in general for support and inspiration. Thanks also to Arthur O'Dwyer for his work on the foundations of the factory function approach.

Thanks to SG1 for its interest and guidance.