Enhancing Thread Constructor Attributes

Document number: P0484R0

Date: 2016-10-16

Reply-to:

Patrice Roy, patricer@gmail.com

Billy Baker, billy.baker@flightsafety.com

Arthur O'Dwyer, arthur.j.odwyer@gmail.com

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

Audience: Library Evolution Working Group/Concurrency Working Group

Note: this document can be seen as a complement to P0320R1.

I. Motivation

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
- Preventing stack growth

As remarked 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.

P0320R1 offers an avenue to address this issue by adding platform-specific arguments, named thread constructor attributes, representing specific feature requests to a thread's constructor. In the "Open points" section of that proposal, one finds "Do we want a way to get the current attributes of a thread?" Indeed, we expect users that have written code requesting specific attributes to want to know to what extent their requests succeeded.

II. Summary

This proposal is meant as a complement to P0320 and tries to offer ways to address the question of how calling code can obtain the current attributes of a thread. Even though we favor the approach described in III.c) Factory Function, we are seeking guidance as to whether pursuing this avenue is seen as worthwhile and if not, whether the committee favors another approach. For this reason, section III. Signaling Failure to Comply discusses other approaches in addition to the one we are proposing.

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. Another possibility, allowing for mandatory and non-mandatory requests, is briefly discussed in IV. Alternative Approach.

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, which can be particularly important should the committee favor III.d) Querying Thread Attributes.

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 or 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.

III.b) Platform-Specific Thread Adoption

Another option that 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.

This option seems counterproductive to our objective of making standard threading facilities the preferred avenue for users.

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`.

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

Yet another approach, which can be coupled with `make_thread()`, is to add an API (tentatively named `get_attributes()`) that would let user code query thread attributes. If this approach is preferred, options to expose thread attributes would 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`

Should this approach be preferred, it will raise the question of where and how these per-thread attributes should be stored. We note that this approach can be complementary with III.c) Factory Function but that both are independent from one another.

IV. Alternative Approach

The current proposal is restricted to cases where thread constructor attributes requests are mandatory, leading to a simplified binary worked/failed resolution of `make_thread()` calls.

A possible generalization of this approach would be to qualify thread attributes as mandatory or not. If this avenue is seen as desirable, then failure to comply with a non-mandatory feature request would not prevent thread startup.

Thread attributes as returned by `make_thread()` or obtained from other facilities such as the ones described in III.d) Querying Thread Attributes would let user code know whether non-mandatory feature requests have been serviced or not.

V. Impact on the Standard

The impacts on the standard for the factory function approach are discussed in III.c) Factory Function. The actual impacts of this proposal on the standard will depend on guidance provided by the Library Evolution and Concurrency Working Groups.

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.