A Boring Thread Attributes Interface

Document #:	P3022R1
Date:	2023-11-07
Project:	Programming Language C++
Audience:	Library Evolution

Reply-to:	David Sankel <dsankel@adobe.com> Darius Neațu <dariusn@adobe.com>

Contents

1 Revision history

- [P3022R0] (2023-10-14)
  - Highlight the problem and the proposed solution.
- P3022R1 (2023-11-07)
  - Update formatting (switch to PDF version).
  - Update comparison with latest revision - [P2019R4].
  - Add more wording/examples in API synopsis.
  - Add Conclusion.

2 A Boring Thread Attributes Interface

2.1 Abstract

The standard library lacks facilities to configure stack sizes or names for threads. This has resulted in a proliferation of code making use of unportable, platform-specific thread libraries. This paper argues that a simple, standardized, thread attributes API is a preferred solution to this problem.

Table 1: Solutions discussed in Library Evolution - 2023-11 Kona

P2019R4's API

```cpp
#include <thread>

void f(int);

int main() {
    // Can set attributes for std::jthread.
    std::jthread my_jthread(
        std::thread_name("Worker1"),
        std::thread_stack_size(512 * 1024),
        f,
        2023
    );

    // Can set attributes for std::thread.
    std::thread my_thread(
        std::thread_name("Worker2"),
        std::thread_stack_size(512 * 1024),
        f,
        2023
    );
    ...
};
```

Proposed API

```cpp
#include <thread>

void f(int);

int main() {
    // Can set attributes for std::jthread.
    std::jthread::attributes attrs;
    attrs.set_name("Worker");
    attrs.set_stack_size_hint(512 * 1024);
    std::jthread my_thread(attrs, f, 2023);

    // Cannot set attributes for std::thread!
    ...
}
```

2.2 Introduction

OSs provide many options for creating new threads. POSIX threads, for example, have over a dozen settings. Two of these stand out in their utility: thread names and thread stack sizes. Thread names simplify debugging workflows and configurable stack sizes are necessary for many-threaded and massive applications. Although most OSs support these settings, the C++ standard doesn't expose them.

In 2016, Vincente J. Botet Escribá proposed in [P0320R1] that OS-specific thread attributes be exposed. More recently, Corentin Jabot made the case in [P2019R3] that thread names and stack sizes in particular should be portably supported.

Although there is general agreement on the motivation for exposing thread names and stack sizes, it is unclear what form the API should take. [P2019R3] introduced a design involving a factory function and individual attribute classes which can be used as optional, unordered arguments to the factory. This complication was justified by the design's avoidance of ABI concerns when a future standard incorporates additional thread attributes.

While [P2019R3]'s design displays ingenuity and originality, we feel that C++ users are better served with a simpler design that has been battle tested over two decades. We speak of [Boost.Thread]'s thread attribute design.

2.3 Our design

In our experience users prefer familiar, simple APIs to novel, complicated ones. There are few things more familiar to C++ users than creating an object and calling some setters:

```
std::jthread::attributes attrs;
attrs.set_name("GUI Thread");
attrs.set_stack_size_hint(10 * 1024);
```

In our design, creating a thread is the same as it has always been except an attributes object can be added as the first argument.

```
std::jthread gui_thread(attrs, gui_function, gui_argument);
```

It's as simple as that. Compare that to the the equivalent in [P2019R3]':

```
auto gui_thread = std::jthread::make_with_attributes(
                      [&]{ gui_function( gui_argument ); },
                      std::thread_name("GUI Thread"),
                      std::thread_stack_size(1024 * 6)
                  );
```

Note that, * a factory function is used instead of a constructor, * the function call is wrapped in a lambda because thread function arguments are unsupported, and * non-positional arguments are dispatched based on type.

While experts familiar with those techniques may appreciate the beauty, it is our experience that most C++ engineers will not. An ergonomic interface will have a much more positive impact on our current and future C++ developers.

```
std::jthread gui_thread(
    std::thread_name("GUI Thread"),
    std::thread_stack_size(1024 * 6)
    [&]{ gui_function( gui_argument ); }
);
```

Moreover, [P2019R4]updates the proposal from [P2019R3] by removing the factory function and passing multiple attribute-like objects into the `std::thread` / `std::jthread` constructors. Although it is a step forward, we still consider it to be less effective to the `Boost.Thread`'s API.

2.4 Considerations

2.4.1 Existing practice

The oldest and the most used C threading library is probably `pthreads` ([man 7 pthreads]). Since the first POSIX standards were defined a few decades ago (around 1985), basic concepts regarding parallel processing were adopted worldwide. Some of the first concepts were: process, threads, global shared memory (data and heap segments), individual thread stack (automatic variables), and attributes shared by threads (PID, FDs).

The C pthread library defines `pthread_attr_t` as a configuration interface used at thread creation via `pthread_create`([man 7 pthreads]). The most used attributes refer to thread detaching, scheduler inheritance and, probably the most used one, the thread stack size. Downsides of known implementations before 2000 were primarily related to portability: `pthread` functioned on Unix variants while Windows used a different API. Additionally, thread utilities were absent from the C++98 standard.

Almost two decades later (2001), the first largely used portable threading library was published - [Boost.Thread]. In the following two decades many more followed such as `OpenMP`, `OpenThreads`, `TBB` and `QThread`.

The boost::thread::attributes API inspired our proposal:

```
template<typename Callable>
thread(attributes& attrs, Callable func);

class thread::attributes {
public:
    attributes() noexcept;
    ~attributes()=default;

    // stack
    void set_stack_size(std::size_t size) noexcept;
    std::size_t get_stack_size() const noexcept;

#if defined BOOST_THREAD_DEFINES_THREAD_ATTRIBUTES_NATIVE_HANDLE
    typedef platform-specific-type native_handle_type;
    native_handle_type* native_handle() noexcept;
    const native_handle_type* native_handle() const noexcept;
#endif
};
```

Here is a typical thread creation example with [Boost.Thread]:

```
boost::thread::attributes attrs;
// set portable attributes
// ...
attr.set_stack_size(10 * 4096);

#if defined(BOOST_THREAD_PLATFORM_WIN32)
    // ... windows version
#elif defined(BOOST_THREAD_PLATFORM_PTHREAD)
    // ... pthread version
    pthread_attr_setschedpolicy(attr.native_handle(), SCHED_RR);
#else
#error "Boost threads unavailable on this platform"
#endif

boost::thread thread(attrs, task_callable, /* arguments for task */ ...);
```

Similar thread attribute implementations are found in other open-source, widely-used C++ threading libraries:

— `LLVM` ([llvm::thread]): thread stack size APIs;
— `Chromium` ([base::Thread]): thread name and stack size APIs;
— `WebKit` ([wtf::Thread]): thread name, stack size and scheduling policy APIs;
— `folly`([folly::thread]): thread name APIs.

More often than not, thread attributes objects are passed to the thread constructor.

2.4.2 Vendor extensions

While non-standard, platform-specific extensions are outside the purview of the C++ standard, it is useful to note that our design does not obstruct such functionality. A vendor could, for example, create their own non-standard thread attributes class:

```
class posix_thread_attributes {
public:
    // Standard
    void set_name(std::string_view) noexcept;
    void set_stack_size_hint(std::size_t) noexcept;

    // Posix-specific
    void set_sched_policy(int) noexcept;
    void set_contention_scope(int) noexcept;

    // ...
};
```

Such classes could then be used in `jthread`'s constructor like `std::jthread::attributes`.

2.4.3 Future `std::jthread::attributes` enhancements

Additional thread attributes, such as priority, may find their way in future C++ revisions and we must not obstruct such additions. Fortunately, our proposed design does not force undesirable ABI breaks as one might expect. The migration path for such additions involves the creation of a backwards-compatible replacement class (e.g., `std::jthread::attributes2`) with additional attributes. Newly written code will utilize the new class and old code can be migrated when appropriate.

2.4.4 Allocation concerns

Many systems cannot use the default allocator. Will the thread name attribute pose a problem in these cases? Fortunately, it is not generally necessary to allocate a `string` on the heap for `thread::attributes` objects.

1. Systems without thread naming support can ignore this attribute completely.
2. Implementations of `thread::attributes` will likely use an OS-provided thread attribute handle for storing this data. `Boost.Thread` took this approach.
3. Many real-time system thread names have length limits which allows `thread::attributes` objects to store the string in place. Linux, for example, limits thread names to 15 characters (man pthread_setname_np).

2.4.5 `std::thread` vs `std::jthread`

We do not propose to add attributes support to `std::thread`. By adding this capability to only `std::jthread`, we give further encouragement to migrate to it.

2.5 API synopsis

The changes discussed in this section are relative to [N4964].

2.5.1 thread.jthread.class

We propose to update [thread.jthread.class.general] section to:

> The class jthread provides a mechanism to create a new thread of execution. The functionality is the same as for class thread ([thread.thread.class]) with the additional abilities to set attributes ([thread.jthread.attributes]) for the new thread before spawning, provide a stop_token (thread.stoptoken) to the new thread of execution, make stop requests, and automatically join.

We propose multiple additions to the `std::jthread` class:

```
namespace std {
  class jthread {
  public:
    // types
+   [thread.jthread.attributes.class], class jthread::attributes
+   class attributes;
    using id = thread::id;
    using native_handle_type = thread::native_handle_type;

    // [thread.jthread.cons], constructors, move, and assignment
    jthread() noexcept;
    template<class F, class... Args> explicit jthread(F&& f, Args&&... args);
+   template<class Attrs = attributes, class F, class... Args>
+      explicit jthread(Attrs&& attrs, F&& f, Args&&... args);
    ~jthread();
    ...

    // [thread.jthread.mem], members
    void swap(jthread&) noexcept;
    [[nodiscard]] bool joinable() const noexcept;
    void join();
    void detach();
    [[nodiscard]] id get_id() const noexcept;
+   [[nodiscard]] std::string_view get_name() const noexcept;
+   [[nodiscard]] std::size get_stack_size() const noexcept;
    [[nodiscard]] native_handle_type native_handle();   // see [thread.req.native]

    // [thread.jthread.stop], stop token handling
    ...
    // [thread.jthread.special], specialized algorithms
    ...
    // [thread.jthread.static], static members
    ...
  private:
    ...
  };
}
```

Usage example:

```
#include <thread>

int main() {
    std::jthread::attributes attrs;
    attrs.set_name("Worker"); // or use std::string_view / std::string
    attrs.set_stack_size_hint(512 * 1024);

    std::jthread t(attrs, [...](...){ ... }, ...);

    std::println("Created thread with name {}.", t.get_name());
    ...
}
```

Extra wording updates:

— Introduce an attributes class. Check thread.jthread.attributes.class.

— A new constructor which accepts an `attributes` object as the first argument. No need to overload the default constructor, as it does not represent a thread of execution.

```
template<class Attrs = attributes, class F, class... Args>
explicit jthread(Attrs&& attrs, F&& f, Args&&... args);
```

Same as `template<class F, class... Args> explicit jthread(F&& f, Args&&... args);`, but an extra attributes object is received as the first argument. Implementations are required to support `std::jthread::attributes` overlead and can add vendor extensions (e.g., `posix_thread_attributes`). The usage of values within passed attributes is implementation-defined (thread.jthread.attributes.class).

— A set of new getters: `get_name()` / `get_stack_size()` - naming inspired from `get_id()`. The behaviour of the getters is implementation-defined.

[[nodiscard]] std::string_view get_name() const noexcept;

Returns: A string representation of the thread name. Empty string if *this does not represent a thread, otherwise this_thread::get_name() for the thread of execution represented by* this.

[[nodiscard]] std::size get_stack_size() const noexcept;

Returns: The thread stack size. 0 if *this does not represent a thread, otherwise this_thread::get_stack_size() for the thread of execution represented by* this.

2.5.1.1 thread.jthread.attributes.class

We propose adding a `std::jthread::attributes` class that stores all attributes passed to `std::jthread`'s constructor.

Class jthread::attributes [thread.jthread.attributes]

```
namespace std {
  class jthread::attributes {
  public:
    attributes() noexcept;
    ~attributes() = default;

    // set thread name
    void set_name(const char* name) noexcept;
    void set_name(std::string_view name) noexcept;
    void set_name(std::string&& name) noexcept;
    // set thread stack size hint
    void set_stack_size_hint(std::size_t size) noexcept;

    // implementation-defined setters
  };
}
```

An object of type `jthread::attributes` can be used to set attributes to the new thread execution before its creation. The attributes are not stored inside the `jthread` object. A thread object that does not represent threads of execution does not have associated attributes.

- `set_name()`: Store a persistent string representation for the thread name. The `char*` and `string_view` overloads allocate a new string. The string lifetime is the same as for the attributes object. Implementations can truncate or ignore this value. Check [thread.this].
- `set_stack_size_hint()`: Store a hint for the initial thread stack size. Implementations can ignore this value. Check [thread.this].

Implementations can add other attributes / setters inside the `jthread::attributes` class.

Usage example:

```
#include <thread>

int main() {
    std::jthread::attributes attrs;
    attrs.set_name("Worker"); // or use std::string_view / std::string
    attrs.set_stack_size_hint(512 * 1024);
    ...
}
```

The public interfaces were inspired from existing practices used in `Boost.Thread` and it matches the `std::jthread`'s current API.

Note that more attributes can be added in a future C++ draft. Check Future std::jthread::attributes enhancements and Vendor extensions.

2.5.2 thread.thread.this

We propose minor updates to the `std::this_thread` namespace:

```
// Namespace this_thread[thread.thread.this]
namespace std::this_thread {
-   thread::id get_id() noexcept;
```

```
+   [[nodiscard]] thread::id get_id() noexcept;
+   [[nodiscard]] std::string_view get_name() noexcept;
+   [[nodiscard]] std::size get_stack_size() noexcept;

  void yield() noexcept;
  template<class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
  template<class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
}
```

Usage example:

```
#include <thread>

int main() {
  std::jthread::attributes attrs;
  attrs.set_name("Worker"); // or use std::string_view / std::string
  attrs.set_stack_size_hint(512 * 1024);

  std::jthread t(attrs, [](...) {
      std::println("Hello from {} thread!", std::this_thread::get_name();
  }, ...);

  std::println("Created thread with name {}.", t.get_name());
  ...
}
```

Extra wording updates:

— Add the same getters as for the `std::jthread` class.

[[nodiscard]] std::string_view this_thread::get_name() noexcept;

Returns: A string representation for the name of the current thread of execution. Every invocation from this thread of execution returns the same value. The returned value is implementation-defined. Recommended practice: implementations should return the name set via `jthread::attributes` (eventually truncated or encoded), or empty string if not possible.

[[nodiscard]] std::size get_stack_size() noexcept;

Returns: The current stack size for the current thread of execution. Multiple invocations from this thread of execution may not return the same value. The returned value is implementation-defined. Recommended practice: implementations should return the current thread stack size (which may be different from the value set via `jthread::attributes`), or 0 if not possible.

— Add the `[[nodiscard]]` specifier to existing member functions: `get_id()`.

[[nodiscard]] thread::id this_thread::get_id() noexcept;

Returns: An object of type thread::id that uniquely identifies the current thread of execution. Every invocation from this thread of execution returns the same value. The object returned does not compare equal to a default-constructed thread::id.

2.6 Conclusion

Criteria	P2019R4's API	Proposed solution
API simplicity	-	x
Attributes aggregation	-	x
Zero-overhead storage	x	x
Implementation experience	x	x
Vendor extension	-	x
Migration to modern C++ APIs	-	x

— `API simplicity`: There are few things more familiar to C++ users than creating an object and calling some setters.
— `Attributes aggregation`: Having all thread attributes into an class.
— `Zero-overhead storage`: Attributes are not stored inside the `std::jthread`. The underlaying/native threading support stores medata anyways (e.g., thread name). Zero extra allocations for attributes in the standard library.
— `Implementation experience`: As we have discussed in Existing practice, the proposed solution is just a standardization of heavily used implementation for more than two decades by the C++ community (e.g., [Boost.Thread]).
— `Vendor extension support`: `std::jthread` can receive non-standard attributes.
— `Migration to modern C++ APIs`: We encourage the C++ community to migrate from `std::thread` to `std::jthread` by adding standard thread attributes only for `std::jthread`.

Due to various advantages shown above, we propose to create a `std::jthread::attributes` class to set thread attributes for the modern C++ threading APIs only - `std::jthread`.

3 References

[base::Thread] base::Thread.
https://source.chromium.org/chromium/chromium/src/+/main:base/threading/thread.h

[Boost.Thread] Boost.Thread.
https://www.boost.org/doc/libs/1_83_0/doc/html/thread.html

[folly::thread] folly::thread.
https://github.com/facebook/folly/blob/main/folly/system/ThreadName.h

[llvm::thread] llvm::thread.
https://llvm.org/doxygen/thread_8h_source.html

[man 7 pthreads] pthreads(7) — Linux manual page.
https://man7.org/linux/man-pages/man7/pthreads.7.html

[N4964] Thomas Köppe. 2023-10-15. Working Draft, Programming Languages — C++.
https://wg21.link/n4964

[P0320R1] Vicente J. Botet Escriba. 2016-10-12. Thread Constructor Attributes.
https://wg21.link/p0320r1

[P2019R3] Corentin Jabot. 2023-05-18. Thread attributes.
https://wg21.link/p2019r3

[P2019R4] Corentin Jabot. 2023-10-15. Thread attributes.
https://wg21.link/p2019r4

[P3022R0] David Sankel, Darius Neațu. 2023-10-14. A Boring Thread Attributes Interface.
https://wg21.link/p3022r0

[wtf::Thread] wtf::Thread.
https://github.com/WebKit/WebKit/blob/main/Source/WTF/wtf/Threading.cpp#L265