# A Boring Thread Attributes Interface


| | |
|---|---|
| Document #: | P3022R0 |
| Date: | 2023-10-14 |
| Project: | Programming Language C++ |
| Audience: | Library Evolution |


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


## Contents


**1 Revision History** **2**

**2 A Boring Thread Attributes Interface** **2**

2.1 Abstract . . . 2
2.2 Introduction . . . 2
2.3 Our design . . . 3
2.4 Considerations . . . 3
2.4.1 Existing Practice . . . 3
2.4.2 Vendor Extensions . . . 4
2.4.3 Future `std::jthread::attributes` enhancements . . . 5
2.4.4 Allocation concerns . . . 5
2.4.5 `std::thread` vs `std::jthread` . . . 5
2.5 API synopsis . . . 5
2.5.1 Create `std::jthread::attributes` Class . . . 5
2.5.2 Update `std::jthread` Class . . . 5
2.5.3 Update `std::this_thread` namespace . . . 6
2.6 References . . . 7

**3 References** **7**

# 1 Revision History

— R0
  — Highlight the problem and the proposed solution- [P3022R0].

# 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 attribute API is a preferred solution to this problem.

P2019R3's API

```
void f(int);

int main() {
    auto thread = std::jthread::make_with_attributes(
      []{ f(3); },
      std::thread_name("Worker"),
      std::thread_stack_size(512 * 1024)
    );
}
```

Proposed API

```
void f(int);

int main() {
    std::jthread::attributes attrs;
    attrs.set_name("Worker");
    attrs.set_stack_size_hint(512 * 1024);

    auto thread = std::jthread(attrs, f, 3);
}
```

## 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[1] that OS-specific thread attributes be exposed. More recently, Corentin Jabot made the case in P2019R3[2] 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(1024 * 6);
```

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.

## 2.4 Considerations

### 2.4.1 Existing Practice

Probably the oldest and the most used C threading library is pthreads. Since the first POSIX standards were defined 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. 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 large-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 typical thread creation example with `Boost.Thread`:

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

#if defined(BOOST_THREAD_PLATFORM_WIN32)
    // ... window 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 (thread stack size APIs), Chromium(thread name & stack size APIs), WTF (thread name, stack size & scheduling policy APIs), folly (thread name APIs). In almost every case, 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 N4950[3].

### 2.5.1 Create `std::jthread::attributes` Class

We propose adding a class that stores all attributes passed to `std::jthread`'s constructor (before actually starting the thread).

```
+ Class thread::attributes[thread.jthread.attributes]
+ namespace std {
+    class jthread::attributes {
+        attributes() noexcept;
+        ~attributes() = default;
+
+    public:
+        // 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;
+    };
+ }
```

The public interfaces were inspired from existing practices used in `Boost.Thread` (e.g., `boost::thread::attributes`) and it matches the `std::jthread`'s current API.

### 2.5.2 Update `std::jthread` Class

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

— an attribute class:`attributes` - described in previous section.

— 2 new constructors: `thread(attributes&& attrs)` / `thread(attributes&& attrs, F&& f, Args&&... args)` - constructors which accept a `std::jthread::attributes` object as the first argument.
— a set of new getters: `get_name()` / `get_stack_size()` - the most common thread attributes used by industry (as described above); naming inspired from `get_id()`. The behaviour of the getters is implementation-defined.
— add the `[[nodiscard]]` specifier for existing member functions: `get_id()`.

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

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

Usage example:

```
#include <thread>

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());
```

### 2.5.3 Update `std::this_thread` namespace

We propose minor additions to the `std::this_thread` namespace: * add same getters as in the `std::jthread` class: `get_name()` / `get_stack_size()`. The behaviour of the new getters is implementation-defined. * add the `[[nodiscard]]` specifier to existing member functions: `get_id()`.

```
// Namespace this_thread[thread.thread.this]
namespace std::this_thread {
+   [[nodiscard]] thread::id get_id() noexcept; // existent function, add [[nodiscard]] specifier
+   [[nodiscard]] std::string_view get_name() noexcept;
+   [[nodiscard]] std::size get_stack_size() noexcept;
```

```
+
    void yield() noexcept;
    ...
}
```

Usage example:

```cpp
#include <thread>

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());
```

### 2.6 References

[1] Vicente J. Botet Escribá. P0320R1: Thread Constructor Attributes. https://wg21.link/p0320r1, 10/2016. [2] Corentin Jabot. P2019R3: Thread attributes. https://wg21.link/p2019r3, 5/2023. [3] Thomas Köppe. N4950: Working Draft, Standard for Programming Language C++. https://wg21.link/N4950, 10/2023 [4] pthreads API docs: man 7 pthreads [5] Boost.Thread API docs: boost::thread [6] LLVM thread implementation: llvm::thread [7] Chromium thread implementation: base::Thread [8] WTF thread implementation: folly::ThreadName

## 3 References

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