Thread attributes

Document #:	P2019R3
Date:	2023-05-12
Programming Language C++	
Audience:	LEWG

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose a way to set a thread stack size and name before the start of its execution, both of which are, as we demonstrate, current practices in many domains.

The absence of these features make `std::thread` and `std::jthread` unfit or unsatisfactory for many use cases.

Revisions

R3

We propose a `make_with_attributes` static factory function instead of an additional constructor. This is because LEWG expressed a slight preference for that design durting the 24th January 23 telecon. The new design, and the names should avoid confusion with the existing constructor, and make the intent clearer.

The design section add a note on the viability of NTTP and getter and setters as these questions came up during the meeting.

Fix some typos.

R2

Wording improvements.

R1

Rework the design to take a list of attributes as parameters, instead of a single type. This is done to alleviate ABI and extensibility concerns raised by SG1.

SG1 suggested exploring a property-based design, but this ship doesn't seem to still be sailing, So instead there is still a constructor interface, but with a separate type for each attribute.

Both `thread_name` and `thread_stack_size` are preserved in this revision despite the mixed feelings SG1 expressed for the `thread_stack_size` attribute (see polls section).

Example

The following code illustrates the totality of the proposed additions:

```
void f();
int main() {
    auto thread = std::jthread::make_with_attributes(f,
                      std::thread_name("Worker"),
                      std::thread_stack_size(512*1024));
    return 0;
}
```

This code suggests a thread name as well as a stack size the implementation should use when creating a new thread of execution.

Achieving the same result in C++20 requires duplicating the entire `std::thread` class, which would be difficult to fit in a Tony table.

Here is how to set the name and stack size of a thread on most POSIX implementation

```
int libcpp_thread_create(libcpp_thread_t *t, void *(*func)(void *),
                         void *arg,
                         size_t stack_size,
                         const libcpp_threadname_char_t* name)
{
    int res = 0;
    if(stack_size != 0) {
        pthread_attr_t attr;
        res = pthread_attr_init(&attr);
        if (res != 0) {
            return res;
        }
         // Ignore errors
        pthread_attr_setstacksize(&attr, stack_size);
        res = pthread_create(t, &attr, func, arg);
        // Ignore errors
        pthread_attr_destroy(&attr);
    }
    else {
        res = pthread_create(t, 0, func, arg);
    }
    if (res == 0) {
        // Ignore errors
        pthread_setname_np(*t, name);
    }
    return res;
}
```

Previous Polls

> SG1, July 2020 We want the ability to provide attributes to thread (full threads: thread, jthread) constructors, even if we can't specify their semantics fully.
>
Yay	Nay
12	1

> The name attribute.
>
Yay	Nay
14	1

> The stack size attribute.
>
Yay	Nay
7	7

Motivation

Threads have a name

Most operating systems, including real-time operating systems for embedded platforms, provide a way to name threads.

Names of threads are usually stored in the control structure the kernel uses to manage threads or tasks.

The name can be used by:

- Debuggers such as GDB, LLDB, WinDBG, and IDEs using these tools
- Platforms and third-party crash dump and trace reporting tools
- System task and process monitors
- Other profiling tracing and diagnostic tools
- Windows Performance Analyzer and ETW tracing

The Visual Studio documentation for SetThreadDescription explains:

> Thread naming is possible in any edition of Visual Studio. Thread naming is useful for identifying threads of interest in the Threads window when debugging a running process. Having recognizably-named threads can also be helpful when performing post-mortem debugging via crash dump inspection and when analyzing performance captures using various tools.

This non-exhaustive table shows that most platforms do in fact provide a way to set and often query a thread name.

Platform	At Creation	After	Query
Linux	`pthread_setname_np`[1]		`pthread_getname_np`
QNX	`pthread_setname_np`		`pthread_getname_np`
NetBSD	`pthread_setname_np`		`pthread_getname_np`
Win32		`SetThreadDescription`[2]	`GetThreadDescription`
Darwin		`pthread_setname_np`[3]	`pthread_getname_np`
Fuchsia	`zx_thread_create`		
Android	`JavaVMAttachArgs`[4]		
FreeBSD	`pthread_setname_np`		
OpenBSD	`pthread_setname_np`		
RTEMS [5]	`pthread_setname_np`	`pthread_setname_np`	`pthread_getname_np`
FreeRTOS	`xTaskCreate`		`pcTaskGetName`
VxWorks	`taskSpawn`		
eCos	`cyg_thread_create`		
Plan 9	`threadsetname`[6]	`threadsetname`[7]	
Haiku	`spawn_thread`	`rename_thread`	`get_thread_info`
Keil RTX	`osThreadNew`		`osThreadGetName`
WebAssembly			

Web assembly was the only platform for which we didn't find a way to set a thread name.

A cursory review of programming language reveals that at least the following languages/environments provide a way to set thread names:

Rust, Python, D , C# , Java , Raku , Swift , Qt , Folly

We also found multiple questions related to setting name thread on StackOverflow.

Thread names are also the object of a C proposal [**?**]

All of that illustrates that giving a name to os threads is standard practice.

Threads have a stack size

In the following, non-exhaustive table, we observe that almost all APIs across a wide range of environments expose a stack size that can either be queried or set. The necessity for such a parameter results from the unfortunate non-existence of infinite tape.

A stack size refers to the number of bytes an application can use to store variables of static storage duration and other implementation-defined information necessary to store the sequence of stack entries making the stack.

Because of that, all implementations which let a stack size be set, do so during the creation of the thread of execution.

We observe fewer variations of APIs across platforms (compared to names) as the parameter is a simple integer that can be no greater than the total system memory.

`pthread_attr_setstacksize` is part of the POSIX specification since Issue 5 (1997). However, platforms vary in the minimum and maximum stack sizes supported.

Platform	At Creation	Query
Linux	`pthread_attr_setstacksize`	`pthread_attr_getstacksize`
QNX	`pthread_attr_setstacksize`	`pthread_attr_getstacksize`
Win32	`CreateThread`	
Darwin	`pthread_attr_setstacksize`	`pthread_attr_getstacksize`
Fuchsia		
Android	`pthread_attr_setstacksize`	`pthread_attr_getstacksize`
FreeBSD	`pthread_attr_setstacksize`	`pthread_attr_getstacksize`
OpenBSD	`pthread_attr_setstacksize`	`pthread_attr_getstacksize`
NetBSD	`pthread_attr_setstacksize`	`pthread_attr_getstacksize`
RTEMS	`pthread_attr_setstacksize`	`pthread_attr_getstacksize`
FreeRTOS	`xTaskCreate`	
VxWorks	`taskSpawn`	
eCos	`cyg_thread_create`	
Plan 9	`threadcreate`	
Haiku		`get_thread_info`
Keil RTX		`osThreadGetStackSize`
WebAssembly		`pthread_attr_getstacksize`

We observe that Java, Rust, Python, C#, Haskell (through a compile-time parameter), D, Perl, Swift, Boost, Qt support constructing threads with a stack size.

There are many reasons why a program may need to set a stack size:

- Ensuring a consistent stack size across platforms for portability and reliability as some applications are designed to be run with a specific amount of stack size.

  More generally, such inconsistencies are a source of bugs and expensive testing.

- Ability to use less than the platform default (usually 1MB on windows, 2MB on many Unixes), which, when not used is a waste (on systems without virtual memory), especially if a large number of threads is started.

- Some applications will set a larger stack trace for the main thread, which is then inherited by spawn threads, which might be undesirable.

- Some applications, notably big games, and other large applications will require a stack larger than the default.

Motivation for standardization

Libc++ `std::thread` implementation is (very approximately) 1000 lines of code. Because stack size needs to be set before thread creation, an application wishing to use a non-default stack size has to duplicate that effort.

We found threads classes supporting names and stack size in many open source projects, including POCO, Chromium, Firefox, LLVM, Bloomberg Basic Development Environment, Folly, Intel TBB, Tensorflow... In many cases, these classes are very similar to `std::thread`, except they support a stack size.

Like thread names, adding this support to `std::thread` would be standardizing existing practices.

People working on AAA games told us that the lack of stack size support prevented them to use `std::thread`, which therefore fails to be a vocabulary type. As such this proposal is more about rounding an existing feature rather than proposing a new one.

FAQ

In which we try to answer all the questions we heard about this proposal

What about queries?

We observe that

- It is rarely useful to query the stack size (except to assert that it is in a range acceptable to the application).
  Querying the stack size could be done by storing a `std::size_t` within the `std::thread` instance, which is rather cheap, but we still don't think it is worth it.
- It is rarely useful to query the thread name, nor is there a portable way to do so (some platforms have API to do that). Use cases for querying a thread name include printing stack traces [3]
- It is also more difficult to design a query API for the name that would not pull in `<string>`
- While less convenient facility, it is at least possible to query available properties after creation from `native_handle`.

Threads should have names??? What next, `mutex` should have a name? `vector`?

Naming threads is standard practice across many operating systems and environments. This proposal merely proposes to expose this widely available and used system feature. We observe that it is common for threads to have names as processes do.

Windows indeed has the concept of named mutexes which are used to share mutexes across processes. However, `std::mutex` is not intended to be shared across processes and as such

does not need a name nor should it have one. A quick review of platforms reveals that it is not standard practice to use a mutex across processes (many UNIX systems rely on lock files).

`std::vector` and other C++ objects are not visible outside the program, except by debuggers which can identify them by their identifiers. Giving them a name would make little sense.

I don't need that and don't want to pay for it

None of the proposed attributes is stored in the thread object nor in any object associated with the thread or its associated `thread` object. The proposed `thread::attribute` object can be destroyed after the thread creation. The behavior of preexisting constructors remains unchanged.

On many implementations, including Linux, the space for the thread name is allocated regardless of whether it is used or not.

It's an ABI break ???

No. Because none of the attributes is stored in the thread or its associated `std::thread` object, the ABI is not changed. We proposed adding a single template constructor.

We cannot speak about stack size in the standard?

There exist a POSIX function that makes the wording more palatable. Setting a stack size insufficient for the correct execution of a well-formed program isn't different than if the default stack size is insufficient ([intro.compliance])

This is not something that the committee has the bandwidth to deal with?

We spent resources standardizing 2 (!) thread classes, which are not used in many cases. This proposal will help more people use `std::thread`.

The author of this proposal is aware of the limited resources of the committee, and that informed the design. The cost of re-implementing classes similar to `std::thread` is great for the industry.

I cannot implement that on my platform?

Here is a conforming minimal implementation

```
namespace std {

struct thread_attribute {};

struct thread_name : thread_attribute {
   constexpr thread_name(std::string_view str) noexcept {};
   constexpr thread_name(std::u8string_view str) noexcept {};
};
```

```
struct thread_stack_size : thread_attribute {
    constexpr thread_stack_size(std::size_t) noexcept {}
};

template <typename T>
concept __thread_attribute = std::derived_from<T, thread_attribute>;

class thread {
    template<class F, __thread_attribute... Attrs>
    static make_with_attributes(F&& f, Attrs&&...) {
     return thread(std::forward<F>(f));
    }
};


}
```

This belongs in a library?

Because the proposed attributes may need to be set during the thread creation, a library would have no choice but to reimplement all of `std::thread`. Besides the cost of doing that implementation, it poses composability challenges (cannot put a `custom_thread` in a `std::thread_poool` for example)

What about GPU threads?

While `std::thread` has no mechanism to specify an execution context, an implementation that wishes to use `std::thread` on a GPU or other hardware could ignore all attributes or the ones not relevant on their platform.

What about other properties

Depending on platforms, threads may have

- A CPU affinity such that they are only executed on a given CPU or set of CPU
- A CPU preference such that they preferentially executed on a given CPU
- A priority compared to the thread in the process
- A priority compared to threads in the system

The meaning of each value and parameter has more variation across implementations, as it is tied to the scheduler or the system.

It is also less generally useful and mostly used in HPC and embedded platforms, where there is the greatest variety of implementation.

As such, thread priorities and other properties are not proposed in this paper. However, the API is designed to allow adding support for more properties in the future.

Note that priorities can often be changed after the thread creation making it easier for third parties libraries to support thread priorities.

Proposed design

Constraints

- Some environments do not support naming threads.
- Thread names can be either narrow encoded or, in the case of win32, Unicode (UTF-16) encoded
- There is a platform-specific limit on thread name length (15(+1) on Linux, 32K on windows)
- All platforms expect names to be null-terminated.
- Some platforms set the name during the thread creation, while on Darwin (and plan 9) it can only be set in the thread in which the name is set.
- The stack size is always set prior to the thread creation.
- Platforms have minimum and maximum stack sizes that are not always possible to expose
- Implementation may allocate more stack size than requested (it is usually aligned on a memory page)
- Implementation may ignore stack size requests
- On some platforms, the thread stack size is not configurable.
- On some platforms, the thread stack size is not query-able.
- Defining these features in terms of wording may be challenging.
- **Users who do not care about these features should not have to pay for it**

Design

We propose adding a new factory function to `std::thread` and `std::jthread` of the form `thread::make_with_attributes(invocable, `*`thread-attribute`*`...)` where attribute is an instance of any type deriving from `std::thread_attribute` (where `std::thread_attribute` serves no other purposes than tagging a type as a thread attribute. Attributes that the implementation does not know how to support are ignored, and the effects of known attributes are implementation-defined.

Having separate types for individual attributes alleviates ABI concerns, and makes extension easier. Individual attributes - which only serve to hold a value, have constexpr constructors so they can be cheaply initialized.

`thread_name` makes a copy of this argument and supports non-null terminated strings, and `u8` strings, as to portably support environments with utf-8 execution encodings (Linux), environments where Unicode encodings can be preserved (Windows), and other environments.

This interface is made possible by the realization that lambdas alleviate the needs for `(j)thread` to support parameter forwarding, and so we don't need attributes + function + arguments but just function + attributes.

The attributes are put after, which is a subpar choice that can be revisited if P2347R2 [2] is adopted. A slightly different design would be to take a tuple of thread attributes as the first parameter.

During Initial LEWG review, a slight preference was expressed for a factory function over a constructor, to avoid any possible user confusion between the existing arg-taking overloads and the overloads that were proposed (and are now replaced by a factory function).

What about using NTTP?

During this review of this paper, it was suggested we could pass thread attributes as NTTP. However, in many cases these attributes may depend on runtime values (thread names depending on a counter would be a good example of that).

Can we add setters/getters later?

As explained, this would not be portable, however it would be possible to add the interface to support these later:

```
class thread {
          template <typename Attr>
          auto get_attribute() -> optional<typename Attr>;
          template <typename Attr>
          auto set_attribute(Attr);
};
```

This might require to extend attributes with (static, constexpr) members specifying

- Whether they can be retrieved/read after construction
- Whether they can be set
- The type of the attribute (int, string, etc), as returning the attribute itself may not be the best way to interact with its value

This would allow to place better compile-time constraints:

```
        class thread {
                template <typename Attr>
                requires is-readable-attribute
                auto get_attribute() -> optional<typename Attr::value_type>;
                template <typename Attr>
                requires is-settable-attribute
                auto set_attribute(Attr);
```

```
};
```

In that design, the attributes types are then only used by the `getter` as a tag to indicate to the implementation which attribute to return.

This could be added later (and isn't ABI breaking).

Implementation

A prototype implementation for libc++ (supporting only POSIX) threads has been created to validate the design. This is available on Compiler explorer, but of limited usefulness in the absence of debugging tools.

Alternatives considered

P0320R1 [1]

P0320R1 proposes `thread::attributes` holding a set of attributes that would all be implementation-defined (the standard would specify no attributes). This puts the burden on the user to check which attributes are present, presumably using `#ifdef`. We feel very strongly that such an approach fails to improve portability and only improves the status quo marginally. There is little value in standardizing a class without standardizing its members.

A class is also less extensible and portable than individually declared attributes and poses more ABI concerns.

Moreover, it proposes a `get_attributes()` function which would returned an implementation-defined object with all the supported attributes of that platform. The problem is that not all attributes that can be set can be queried (and reciprocally), and that interface would force and implementation to return all the attributes it supports, which is wasteful (would have to allocate for the name if a user wants to check the stack size).

P0484R1 [4]

P0484 proposes several solutions in the same design space:

- A constructor taking a native handle as a parameter

  ```
  std::thread thread::thread(native_handle_type h);
  ```

  This is probably a good idea, regardless of the attributes presented here, to interface with C libraries or third-party code.

  This solves the problem of having to rewrite an entire thread class just to set a stack size. However, it would still be painful to do so portably, as described in P0484. A standard library that targets a limited number of platforms can set the attributes more easily than a library that may desire to work in an environment where C++ is deployed.

Wording

🔲 Threads [thread.threads]

🔲 General [thread.threads.general]

?? describes components that can be used to create and manage threads. [*Note:* These threads are intended to map one-to-one with operating system threads. — *end note*]

🔲 Header <thread> synopsis [thread.syn]

```
#include <compare>              // see ??

namespace std {

    class thread_attribute {};
    class thread_name;
    class thread_stack_size;

    // ??, class thread
    class thread;

    void swap(thread& x, thread& y) noexcept;

    // ??, class jthread
    class jthread;

    // ??, namespace this_thread
    namespace this_thread {
        thread::id get_id() 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);
    }
}
```

🔲 Thread Attributes [thread.attributes]

A type which inherits from `thread_attribute` is a thread attribute. Thread attributes can be used to define additional implementation-defined behaviors on a `thread` or `jthread` object. Thread attributes not supported by an implementation are ignored.

🔲 Class `thread_name` [thread.thread.name.class]

```
class thread_name : thread_attribute {
public:
```

```
    constexpr thread_name(std::string_view name) noexcept;
    constexpr thread_name(std::u8string_view name) noexcept;
private:
    implementation-defined __name[implementation-defined]; // exposition only
};
```

Recommended practice:

The `thread_name` thread attribute, if supported by the implementation, can be used to set the name of a thread such that the name could be used for debugging or platform-specific display mechanisms. The name should not be stored in the `std::thread` or `std::jthread` object.

```
constexpr thread_name(std::string_view name) noexcept;
constexpr thread_name(std::u8string_view name) noexcept;
```

Effects: Initializes `__name` with `name` in an implementation-defined manner.

Recommended practice: If `__name` is not large enough to store the value of `name`, the thread name might be truncated. If the implementation performs a text conversion during the initialization of `__name` and if `name` is a valid code unit sequence in the encoding associated with its type, `__name` should be a valid code unit sequence in its associated encoding.

🔲 Class `thread_stack_size` [thread.thread.stack.size.class]

```
class thread_stack_size : thread_attribute {
public:
    constexpr thread_stack_size(std::size_t size) noexcept;
private:
    constexpr std::size_t __size; // exposition only
};
```

Recommended practice: Configure a desired stack size as if by POSIX `pthread_attr_setstacksize()`. The stack size set by the implementation may be adjusted up or down to meet platform-specific requirements

If `__size == 0` is `true` the thread attribute should be ignored.

```
constexpr thread_stack_size(std::size_t size) noexcept;
```

Effects: Initializes `__size` with `size` in an implementation-defined manner.

🔲 Class `thread` [thread.thread.class]

🔲 General [thread.thread.class.general]

The class `thread` provides a mechanism to create a new thread of execution, to join with a thread (i.e., wait for a thread to complete), and to perform other operations that manage and query the state of a thread. A `thread` object uniquely represents a particular thread of execution. That representation may be transferred to other `thread` objects in such a way that no two `thread` objects simultaneously represent the same thread of execution. A thread of

execution is *detached* when no `thread` object represents that thread. Objects of class `thread` can be in a state that does not represent a thread of execution. [*Note:* A `thread` object does not represent a thread of execution after default construction, after being moved from, or after a successful call to `detach` or `join`. — *end note*]

```
namespace std {
class thread {
    public:
    class thread::id;
    class id;
    using native_handle_type = implementation-defined;         // see ??

    // construct/copy/destroy
    thread() noexcept;
    template<class F, class... Args> explicit thread(F&& f, Args&&... args);

    ~thread();
    thread(const thread&) = delete;
    thread(thread&&) noexcept;
    thread& operator=(const thread&) = delete;
    thread& operator=(thread&&) noexcept;


    template <class F, class... Attrs>
    static thread make_with_attributes(F && f, Attrs&&... attrs);


    // ??, members
    void swap(thread&) noexcept;
    bool joinable() const noexcept;
    void join();
    void detach();
    id get_id() const noexcept;
    native_handle_type native_handle();                          // see ??

    // static members
    static unsigned int hardware_concurrency() noexcept;

    private:
    struct attribute-tag{}; // exposition only
    template <class F, class... Attrs>
    thread(attribute-tag, F && f, Attrs&&... attrs); // exposition only
};
}
```

🔲 Constructors [thread.thread.constr]

```
template<class F, class... Args>
explicit thread(F&& f, Args&&... args);
```

```
template<class F, class... Attrs>
requires (sizeof...(Attrs) != 0)
explicit thread(F&& f, Attrs&&... attrs); // exposition only
```

Constraints:

- `remove_cvref_t<F>` is not the same type as `thread`.

Mandates: The following are all `true`:

- `is_constructible_v<decay_t<F>, F>`,
- `(is_constructible_v<decay_t<Args>, Args> && ...)`,
- `is_move_constructible_v<decay_t<F>>`,
- `(is_move_constructible_v<decay_t<Args>> && ...)`, and
- `is_invocable_v<decay_t<F>, decay_t<Args>...>`.
- `is_constructible_v<decay_t<F>, F>` is `true`,
- `is_move_constructible_v<decay_t<F>>` is `true`.
- For the first overload, the following are all true:
  - `(is_constructible_v<decay_t<Args>, Args> && ...)`,
  - `(is_move_constructible_v<decay_t<Args>> && ...)`, and
  - `is_invocable_v<decay_t<F>, decay_t<Args>...>`.
- For the exposition-only overload, the following are all true:
  - `(is_base_of_v<thread_attribute, Attrs>) && ...)`,
  - `(is_invocable_v<decay_t<F>>`.

Each type representing a thread attribute known of the implementation occurs no more than once in `Attrs`,

Preconditions: `decay_t<F>` and each type in `decay_t<Args>` meet the *Cpp17MoveConstructible* requirements

Effects: The new thread of execution executes

`invoke(`*decay-copy*`(std::forward<F>(f)), `*decay-copy*`(std::forward<Args>(args))...)`

with the calls to *decay-copy* being evaluated in the constructing thread. Any return value from this invocation is ignored. [*Note:* This implies that any exceptions not thrown from the invocation of the copy of `f` will be thrown in the constructing thread, not the new thread. — *end note*] If the invocation of `invoke` terminates with an uncaught exception, `terminate` is called.

For the second exposition overload, `attrs` can be used to tailor the thread with additional implementation-defined behaviors. Thread attributes parameters unknown to the implementation are ignored. (see [thread.attributes]).

Synchronization: The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of `f`.

Postconditions: `get_id() != id()`. `*this` represents the newly started thread.

Throws: `system_error` if unable to start the new thread.

Error conditions:

- `resource_unavailable_try_again` — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

🔲 Factory functions [thread.thread.factory]

```
template<class F, class... Args>
static thread make_with_attributes(F&& f, Args&&... args);
```

Constraints: Mandates: The following are all `true`:

- `is_constructible_v<decay_t<F>, F>,`
- `(is_base_of_v<thread_attribute, Attrs>) && ...),`
- `(is_invocable_v<decay_t<F>>.`
- `(is_base_of_v<thread_attribute, Attrs>) && ...),`
- `(is_invocable_v<decay_t<F>>.`

Returns: `thread(`*`attribute-tag`*`{}, std::forward<F>(f), std::forward<Args>(args)...));`

`jthread`

[Editor's note: TODO: exact same wording as for thread] .

Feature test macros

[Editor's note: Add a new macro in `<version>`, `<thread>`: `__cpp_lib_thread_attributes` set to the date of adoption] .

Acknowledgments

Thanks to Martin Hořeňovský, Kamil Rytarowski, Clément Grégoire, Bruce Dawson, Patrice Roy, Ronen Friedman, Billy Baker, and others for their valuable feedback. Thanks to Jonathan Wakely and Mathias Stearn for their wording suggestions.

References

[1] Vicente J. Botet Escriba. P0320R1: Thread constructor attributes. https://wg21.link/p0320r1, 10 2016.

[2] Corentin Jabot and Bruno Manganelli. P2347R2: Argument type deduction for non-trailing parameter packs. https://wg21.link/p2347r2, 10 2021.

[3] Antony Polukhin and Antony Polukhin. P0881R5: A proposal to add stacktrace library. https://wg21.link/p0881r5, 6 2019.

[4] Patrice Roy, Billy Baker, and Arthur O'Dwyer. P0484R1: Enhancing thread constructor attributes. https://wg21.link/p0484r1, 6 2017.

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4892