Usability improvements for `std::thread`

Document #:	P2019R0
Date:	2020-06-29
Project:	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.

ABI considerations

This paper proposes a class `std::thread::attributes` which is meant to be extended in future standard versions. However, such extension capabilities are not be possible without the ability to break ABI.

The proposed design is therefore not viable over time within the current stability constraints. We do not propose an alternative design at this time.

Example

The following code illustrate the totality of the proposed additions:

```
void f();
int main() {
    std::thread thread(
     std::thread::attributes()
        .name("Worker")
        .stack_size(512*1024),
     f
    );
    thread.join();
    return 0;
}
```

This code suggest a thread name as well as a stack size the implementation should used 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;
}
```

Motivation

Threads have a name

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

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

[1] GLIBC 2.12+, MUSL

[2] Since Windows 10 1607 - In older versions a name can be set only when a debugger is attached, by throwing an exception from the calling thread. See Windows Documentation and this article by Bruce Dawson

[3] Can only be called from the new thread

[4] See `https://stackoverflow.com/a/59490438/877556`

[5] since 2017

[6] Can only be called from the new thread

[7] Can only be called from the new thread

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

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 size 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 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 for 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 question 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 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 includes printing stack traces [**?**]

- 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 name ? `vector`?

Naming threads is standard practice across many operating system and environments. This proposal merely proposes to expose this widely available and use 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.

What about domains? Colors?

These are not features provides by systems nor it is existing practice or implemented in any of the languages and libraries we surveyed. While it is true that these features may be offered by some tools they either use a heuristic to set these information or rely on tool-specific methods.

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 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 it's 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 which 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 have 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

```
class thread {
    class attributes  {
    public:
        attributes& stack_size(std::size_t) noexcept {return *this;}
        attributes& name(std::span<const char>)  {return *this;}
        attributes& name(std::span<const char8_t>)  {return *this;}
    };
template<class F, class... Args>}
explicit thread(const attributes &, F&& f, Args&&... args)
: thread(std::forward<F>(f), std::forward<Args>(args)...) {}
};
```

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 GPUs 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 which name is set.
- The stack size is always set prior to the thread creation.
- Platforms have minimum and maximum stack size 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 `thread::attribute` on which can be set a stack size and a name. This class can then be passed to the constructor of `std::thread` and `std::jthread`.

The setters `name` and `stack_size` can be used to set each attributes, which are ignored if not set.

`name` can be either utf-8 or narrow encoding. This is because most posix implementations are UTF-8 (but take a narrow encoding parameter) and windows is UTF-16 (`wchar_t*`). `name` take a span whose header is cheaper than `string_view` and it otherwise makes no difference.

The name has to be copied by implementations which do support that attribute, however, that name is short, and limited to 16 or 32 bytes in many implementations. Still, this is why `name` is not `noexcept`

The setters are preferred to constructors arguments as it makes it easier to add other attributes in the future, such as thread priority, while maintaining a small surface API.

Implementation

A prototype implementation for libc++ (supporting only POSIX) threads has been created to validate the design.

Alternatives considered

P0320 [?]

P0320R1 proposes a mechanism such as the `thread::attributes` class proposed in this paper, except all attributes would be implementation-defined (the standard would specify no setter). 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.

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).

P0484 [?]

P0484 proposes several solutions in the same design space:

- A constructor taking a native handle as 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 on an environment where C++ is deployed.

- A factory function for creating a thread with attributes

```
template <class F, class ... Args>
unicorn<std::thread, ??> make_thread(thread::attributes, F && f, Args && ... args);
```

We think this is trying to solve two problems:

- Threads cannot be used without exceptions support
- Some users want the stack size to be guaranteed

We are sympathetic to the first concern, however, it seems orthogonal to thread attributes. If a unicorn type (`expected`?) or a cheaper exception mechanism is ever standardized, such a factory function will be welcome, but it doesn't prevent a thread constructor to support attributes. As for guaranteed stack size:

- Some platforms do no support stack size at all - doesn't mean they won't use the desired amount
- Some platforms may ignore stack size requests silently
- Some platforms may allocate more than request to align with memory pages
- Trying to check after the thread has started is not possible (aka it would throw an exception even though the new thread has started)

As such, we allow but do not require and implementation to throw when a stack size request cannot be fulfilled.

Wording

For illustrative purposes only

Class `thread` **[thread.thread.class]**

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

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

        ~thread();
        //...
    };
}
```

🔒 Class `thread::attributes` [thread.thread.attr]

```
class thread::attributes  {
public:
    attributes() noexcept = default;
    attributes(const attributes&);
    attributes(attributes&& other) = default;

    attributes& stack_size(std::size_t size) noexcept;
    attributes& name(std::span<const char> name);
    attributes& name(std::span<const char8_t> name);

    implementation-defined  __name; // exposition only
    std::size_t __size; // exposition only
};
```

An object of type `thread::attributes` provides a way to set different parameters that may be used by the constructor of `std::thread` and `std::jthread`.

The behavior of each attribute is implementation-defined. Attributes that are not handled by the implementation are ignored.

Name	Description
`stack_size`	[*Note:* The intent is to configure a stack size as if by POSIX `pthread_attr_setstacksize()`. — *end note*] If a non-zero value is outside of the range of values supported by the implementation, `thread` and `jthread` constructors may throw `system_error`.
`name`	`name` represents a string convertible to the native character encoding which may be used to non-uniquely identify a thread of execution. When the value of `name` is outside of the range of values supported by the implementation, an implementation can ignore the attribute entirely or use any `subspan` of the value to non-uniquely identify the thread of execution.

```
attributes& stack_size(std::size_t size) noexcept;
```

Effects: Initialize the `stack_size` attribute with the value of `size`.

Returns: `*this;`

```
attributes& name(std::span<const char> name);
```

Requires: `name` represents a valid string of character in the narrow literal encoding

Effects: Initialize the `name` attribute with the value of `name`.

Throws: `bad_alloc`

Returns: `*this;`

```
attributes& name(std::span<const char8_t> name);
```

Effects: Initialize the `name` attribute with the value of `name`.

Returns: `*this;`

Throws: `bad_alloc`

🔒 Constructors [thread.thread.constr]

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

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

Constraints:

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

Expects: `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.

In the second form, `attrs` can be used to specify implementation-defined thread attributes (see [thread.thread.attr]).

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

Ensures: `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.

Acknowledgments

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

[N2419] Kamil Rytarowski *Add methods for setting and getting the thread name* (WG14 proposal)
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2419.htm