Document number:	P0320R0
Date:	2016-05-22
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group/Concurrency Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

Thread Constructor Attributes

Abstract

This paper presents an extension of `thread` construction allowing to pass some additional attributes, like the stack size.

Table of Contents

Introduction

This paper presents an extension of `thread` construction allowing to pass some additional attributes, like the stack size.

Motivation and Scope

Today we can construct an instance of `thread` with a function or callable object , e.g:

```
void find_the_question(int the_answer);

std::thread deep_thought_2(find_the_question, 42);
```

Threads launched in this way are created with implementation defined thread attributes as stack size, scheduling, priority, ... or any platform specific attributes.

However in some specific domains it is important to be able to be more specific so that the resources are used in an optimal way.

As each platform has its own specific thread construction attributes, it is not evident how to provide a portable interface that allows the user to set the platform specific attributes. This paper stay in the middle road through the class `std::thread::attributes` which allows to set at least in a portable way the stack size and provides some mechanisms to

Proposal

This paper proposes then to

- add a `thread::attributes` class with a single portable stack size attribute, and possible getter of a `native_handle_type` when available on the platform.
- add `thread` constructors taking a `thread::attributes` parameter.

How to set the stack size?

The stack size attribute of a thread can be set as follows:

```
std::thread::attributes attrs;
attrs.set_stack_size(4096*10);
std::thread deep_thought_2(attrs, find_the_question, 42);
```

Even for this simple attribute there could be portable issues as some platforms could require that the stack size should have a minimal size and/or be a multiple of a given page size. The library implementation could adapt the requested size to the platform constraints so that the user doesn't need to take care of it.

This is the single attribute that is provided in a portable way. In order to set any other `thread` attribute at construction time the user needs to use non portable code.

Using a native*handle*type

On Posix platforms the user will need to get the thread attributes native handle and use it for whatever attribute.

Next follows how the user could set the stack size and the scheduling policy on PThread platforms.

```
std::thread::attributes attrs;
// set portable attributes
// ...
attr.set_stack_size(4096*10);
#if defined(THREAD_PLATFORM_PTHREAD) // replace THREAD_PLATFORM_PTHREAD by any macro
    // ... pthread version
    pthread_attr_setschedpolicy(attr.native_handle(), SCHED_RR);  // non portable
#endif
std::thread th(attrs, find_the_question, 42);
```

Extending the `thread::attributes` interface

On Windows platforms it is not so simple as there is no type that compiles the thread attributes. There is one attribute linked to the creation of a thread on Windows that is emulated via the `thread::attributes` class, this is the `LPSECURITY_ATTRIBUTES lpThreadAttributes`. The implementation can provide a non portable `set_security` function so that the user can provide it before the thread creation as follows

```
std::thread::attributes attrs;
// set portable attributes
attr.set_stack_size(4096*10);
#if defined(THREAD_PLATFORM_WINDOWS) // replace THREAD_PLATFORM_WINDOWS by any macro
// set non portable attribute
LPSECURITY_ATTRIBUTES sec;
// init sec
attr.set_security(sec); // non portable
#endif
std::thread th(attrs, find_the_question, 42);
// Set other thread attributes using the native_handle_type.
//...
```

Design rationale

Making the stack size really a portable attribute

As the stack size is given as a hint, there is no portability issues. In platforms where the the stack size can not be set, this hint is just ignored, and the result of getting it would be the value '0', meaning the default stack size.

Using setters versus constructors

About `thread::attributes::native_handle`

As the Posix based platforms have associated a `pthread_attribute_t` to the thread creation, is seems natural to provide a native handle accessor so that the user can in a non-portable way make use of this handle. This is in line with the other native handle types.

About `thread::attributes` non portable setters

Users of platforms as Windows that don't have a native handle for the thread attributes, could need to set some attributes in a non-portable way. This paper let the implementation the possibility to add implementation defined functions to achieve this goal.

Proposed wording

The wording is relative to P0159R0.

Thread library

Update Class thread [thread.thread.class] section with

Class thread [thread.thread.class]

```
namespace std {
namespace experimental {
inline namespace concurrency_v3 {

  class thread {
  public:
    // add after id
    class attributes;

    // add after thread construtor
    template <class F, class ...Args>
    explicit thread(attributes cosnt& attr, F&& f, Args&&... args);

  };

}}}
```

Class `thread::attributes`

```
namespace std {
namespace experimental {
inline namespace concurrency_v3 {

class thread::attributes {
public:
    attributes() noexcept;
    // stack
    void set_stack_size(std::size_t size) noexcept;
    std::size_t get_stack_size() const noexcept;

    typedef `implementation-defined` native_handle_type; // See 30.2.3
    native_handle_type native_handle() noexcept; // See 30.2.3

};

}}}
```

Implementations are free to add other functions not specified in this document.

```
attributes() noexcept;
```

Effects: Constructs a thread attributes instance with its default values. A thread constructed with such a default `attributes` shall behave as if there was no `attributes` parameter.

Postconditions: `this-> get_stack_size()` returns `0`.

Throws: Nothing

```
void set_stack_size(std::size_t size) noexcept;
```

Effects: Stores the stack size to be used to create a thread. This is a hint that the implementation can choose a better size if to small or too big or not aligned to a page. `0` means the default.

Postconditions: `this-> get_stack_size()` returns the chosen stack size.

Throws: Nothing.

```
std::size_t get_stack_size() const noexcept;
```

Returns: The stack size to be used on the creation of a thread. Note that this function can return 0 meaning the default. *Throws*: Nothing.

```
    native_handle_type native_handle() noexcept;
```

Returns: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying thread attributes implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Update thread constructors [thread.thread.constr] adding

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

As before

Remarks: The first overload constructor shall not participate in overload resolution if `decay_t<F>` is the same type as `std::thread` or `std::thread::attributes`.

Effects: Constructs an object of type `thread`, taking in account the passed attributes. The first overload behaves as if a default attributes was passed. ....

Implementability

This proposal can be implemented as pure library extension, without any compiler magic support. Boost.Thread provides it since version 1.51

Open points

The authors would like to have an answer to the following points if there is at all an interest in this proposal:

- Do we want the stack size as portable attribute?
- Do we want the `thread::attributes::native_handle_type` ?
- Could an implementation provide additional functions in class `thread::attributes` ?
- Should the default value for the stack size be implementation defined?

Acknowledgements

Thanks to all that commented this proposal helping me to improve globally the paper.

References

- P0159R0 P0159 - Draft of Technical Specification for C++ Extensions for Concurrency

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0159r0.html

- Boost.Thread http://www.boost.org/doc/libs/1*60*0/doc/html/thread.html