Document Number: P0162R0
Date: 2015-11-06
Reply To: Christopher Kohlhoff <chris@kohlhoff.com>
Audience: LEWG

A response to "P0055R0: On Interactions Between Coroutines and Networking Library"

1 Introduction

P0055R0 *On Interactions Between Coroutines and Networking Library* outlines design changes to, ostensibly, improve the performance of the proposed technical specification P0112R1 *Networking Library*. P0055R0 makes the claim that the overhead "can be made even lower if [the networking library] can take advantage of coroutines". P0055R0 claims to achieve this by using the coroutine's "stack" as a place to store per operation state, as opposed to this state being allocated for each operation.

This claim is not borne out by benchmarks. The property of coroutines that allows us to optimise allocation is that a coroutine represents a strictly sequential chain of event handlers. This property is not unique to coroutines. In fact, a similar optimisation has been available to users of the Boost.Asio library since 2005 through a custom allocation mechanism. Since 2013, this has been automatically enabled for the most common usage patterns even when no custom allocator is specified.

The proposed networking library TS provides a custom allocation mechanism based on this field experience.

In addition:

- The proposed design in P0055R0 breaks the conceptual design model where asynchronous operations mirror the semantics of normal synchronous functions.
- The proposed design in P0055R0 restricts the implementation freedom of authors of asynchronous operations, and indeed inhibits the development of low overhead abstractions.

Consequently, the author is of the opinion that the design changes proposed in P0055R0 should not be adopted.

2 Description of existing design

2.1 Transformation of CompletionToken into return type and value

In the proposed networking TS, asynchronous operations are launched using *initiating functions.* An initiating function has the following form:

```
template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive(const MutableBufferSequence& buffers,
                        socket_base::message_flags flags,
                        CompletionToken&& token);
```

Completion signature: `void(error_code ec, size_t n)`.

The *Completion signature* element specifies the form of the asynchronous operation result. A completion signature is to an asynchronous operation as a return type is to a normal function.

The initiating function's final argument is a *completion token*. The completion token is used to specify how the user of the asynchronous operation would like to receive the result. The transformation from completion token to the *DEDUCED* return type is performed using the `async_result` trait:

```
async_result<std::decay_t<CompletionToken>, Signature>::return_type
```

or, using the `async_completion` convenience wrapper:

```
async_completion<CompletionToken, Signature>::return_type
```

The result of the completed asynchronous operation is passed back from implementation to user via a *completion handler*. The completion handler is a function object, constructed from the completion token, that conforms to the completion signature.

Finally, the return value of the initiating function is determined using the `async_result` trait's `get()` member function. Thus, a complete initiating function looks like:

```
template<class MutableBufferSequence, class CompletionToken>
  typename async_result<std::decay_t<CompletionToken, void(error_code, size_t)>>::return_type
    async_receive(const MutableBufferSequence& buffers,
                  socket_base::message_flags flags,
                  CompletionToken&& token)
{
  typename async_result<std::decay_t<CompletionToken,
    void(error_code, size_t)>>::completion_handler_type
      completion_handler(forward<CompletionToken>(token));
  async_result<CompletionToken, void(error_code, size_t)> result(completion_handler);
  // do something to call completion_handler when the operation completes
  return result.get();
}
```

More commonly, an initiating function will make use of the `async_completion` convenience wrapper, so that it is simply:

```
template<class MutableBufferSequence, class CompletionToken>
  auto async_receive(const MutableBufferSequence& buffers,
                     socket_base::message_flags flags,
                     CompletionToken&& token)
{
  async_completion<CompletionToken, void(error_code, size_t)> init(token);
  // do something to call init.completion_handler when the operation completes
  return init.result.get();
}
```

2.2 Associated allocators

Asynchronous operations may allocate memory, such as a data structure to store copies of the completion handler object and the initiating function's arguments. To allow this allocation to be customised, the P0112R1 networking library says that every completion handler has an *associated allocator*. The implementation of an asynchronous operation is expected (or required, in the case of the asynchronous operations in the TS) to use the associated allocator to obtain memory.

The simplest way to specify an associated allocator is for a completion handler type to provide a nested typedef, `allocator_type`, and a member function `get_allocator`. For example:

```
struct my_completion_handler
{
  typedef my_custom_allocator allocator_type;
  allocator_type get_allocator() const noexcept;
  void operator()() { /* ... body of handler ... */ }
};
```

Specialising the `associated_allocator` *associator* trait enables the creation of more complex association relationships. By default, the trait uses the typedef and member function above, if they are present, with a default allocator to fall back to if they are not. Thus, completion handlers are not required to specify an associated allocator.

To obtain the associated allocator, an asynchronous operation implementation calls the `get_associated_allocator` function:

```
auto alloc = net::get_associated_allocator(completion_handler);
```

This single-argument form defaults to `std::allocator<void>` when the completion handler does not have an associated allocator. To specify a different default, a two-argument form is used:

```
auto alloc = net::get_associated_allocator(completion_handler, my_allocator<void>());
```

Finally, the asynchronous operation is required to de-allocate all memory obtained from the associated allocator prior to invoking the completion handler. As we shall see below, this requirement is a key element of the design.

3 P0055R0's proposed mechanism

P0055R0 proposes to combine the two customisation points:

- return type and value
- allocation of memory

into a single customisation point as shown below:

```
template<class CompletionToken>
auto async_xyz(T1 t1, T2 t2, CompletionToken&& token) noexcept(auto)
{
  return completion_token_transform<void(R1 r1, R2 r2)>(
       forward<CompletionToken>(token),
       [=](auto typeErasedHandler) { async_xyz_impl_raw(t1, t2, typeErasedHandler); });
}
```

4 Benchmarks

4.1 Tests

The following three approaches are benchmarked:

- **Callbacks with custom allocation strategy.** This test uses chains of completion handlers, implemented in terms of the proposed networking TS's existing design. Each handler has a custom associated allocator.
- **Callbacks with default allocation strategy.** This test uses chains of completion handlers, implemented in terms of the proposed networking TS's existing design. The handlers do not provide a custom allocator, but instead the implementation provides a default allocator that is optimised for asynchronous operations. This is based on the allocation strategy that has been employed in the Boost.Asio library for a number of years, and it is assumed that networking TS implementations will employ a similar strategy.
- **Coroutine using P0055R0 approach.** This test uses coroutines with a stack allocation strategy as proposed in that document.

4.2 Design

To accurately compare the performance of the approaches, we need to avoid high cost system calls. The execution times of these system calls are several orders of magnitude more than the facilities being tested, and consequently they may obscure the performance differences.

To this end, we will emulate asynchronous operating system facilities using a C-like API with a lightweight implementation. This API is based on the general shape described in P0055R0:

```
using OsResultType = int;
struct OsContext { };

using CallbackFnPtr = void(*)(OsResultType r, OsContext*);
OS_DECL void os_associate_completion_callback(CallbackFnPtr cb);
OS_DECL void os_trigger_completion();

using ParamType = int;
OS_DECL void os_xyz(ParamType p, OsContext* o);
```

Registers a persistent callback function

Invokes the persistent callback with the next queued OsContext

Emulates an asynchronous operation by enqueuing the specified OsContext

This C-like API is then wrapped in a C++ asynchronous operation to test each of the approaches. The pattern for each test is:

- At program start, register a persistent callback using the function `os_associate_completion_callback`.
- Wrap the `os_xyz` function in a C++ asynchronous operation, `async_xyz`. The `async_xyz` function allocates an `OsContext`-derived object to be passed to `os_xyz`. It is the method of allocation that is being benchmarked.
- Start 10 independent flows-of-control (i.e. callback chains or coroutines), which make repeated calls to `async_xyz`.
- Measure the time to make 1,000,000,000 calls to `os_trigger_completion`.

Finally, the tests are built in two separate modes:

- **OS as DLL.** The operating system emulation is built as a separate dynamically linked library. This inhibits optimisation across the OS boundary and more accurately reflects how operating system facilities are exposed to user programs.
- **OS as static library.** The operating system emulation is linked directly into the test executable, and link-time optimisation is enabled.

The complete test programs may be obtained from https://github.com/chriskohlhoff/p0162-bench.

4.3 Results

The following results show the time per triggered completion, using each of the approaches. The tests were compiled as 64-bit executables with Microsoft Visual C++ 2015 and run on Windows 7, using an Intel Core i7-3770, with hyper-threading and turbo boost disabled. Each test program was run at high priority and pinned to a specific CPU.

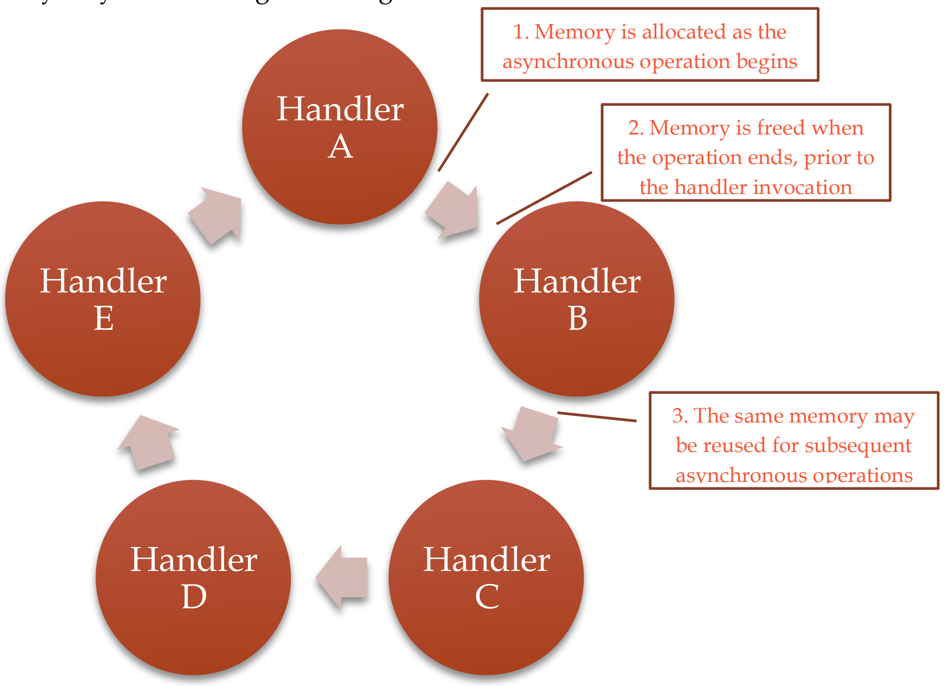
	OS as DLL	**OS as static library**
Callback with custom allocation strategy	16.4 ns	5.6 ns
Callback with default allocation strategy	16.5 ns	5.6 ns
Coroutine using P0055R0 approach	18.0 ns	6.1 ns

4.4 Analysis

These test results show that coroutines do not in fact make the overhead lower. However, except perhaps in the OS as DLL case, the differences are negligible.

As described above, a key requirement of the associated allocator mechanism is that an asynchronous operation deallocates the memory prior to invoking the completion handler. This ensures that the memory is immediately available for reallocation, so that if the completion handler launches another asynchronous operation, the memory may be reused.

If we structure our asynchronous control flow as sequential chains of completion handlers, then the same memory may be reused again and again.


1. Memory is allocated as the asynchronous operation begins
2. Memory is freed when the operation ends, prior to the handler invocation
3. The same memory may be reused for subsequent asynchronous operations
Handler A
Handler B
Handler C
Handler D
Handler E

Thus, it is having a strictly sequential chain of completion handlers that enables the optimisation. Coroutines are one way of representing a strictly sequential chain, but of course they are not the only such representation.

Furthermore, since sequential chains are the common case for asynchronous operations, experience shows that a library implementation can provide a default allocation strategy that enables this optimisation by default. Callbacks need not provide custom allocators to have low overhead. Indeed, the associated allocator mechanism allows implementers to specify an appropriate default, and it is expected that networking TS implementations will also take advantage of this to optimise the common case.

5 Design impacts of P0055R0

By combining the two customisation points:

- return type and value
- allocation of memory

into a single customisation point, P0055R0's approach has a number of disadvantages. We will explore these below.

5.1 All asynchronous operations must share a common base class

As specified, P0055R0's design requires all asynchronous operations to be implemented in terms of a common, type-erased type. This might be true if we limit our scope to some subset of operations, such as the overlapped I/O operations on Windows.

The intent of P0112R1's asynchronous model is to allow different types of asynchronous operations to be integrated. For example, the implementation of an asynchronous socket operation on a particular platform may be completely different from the implementation of a timer operation.

However, this is not necessarily an inherent limitation of the P0055R0 approach. It may be possible to adapt the approach to support different base types.

5.2 Asynchronous operations are restricted to one fixed-size allocation

P0055R0's mechanism is designed to allow the "allocated" object to reside on the coroutine stack. Consequently, the size of object must be known at compile time and we are limited to just one such object.

The proposed networking TS's associated allocator builds on the standard allocator facilities to enable allocation of arbitrary objects. For example, in addition to the memory for the completion handler, an asynchronous operation's implementation may require a `std::vector`, `std::string`, or other container. The associated allocator may also be used for this.

5.3 Allocation is forced on all asynchronous operations

Under the networking TS's asynchronous model we can write lightweight, efficient layers of abstraction. For example:

```
template <class DynamicBuffer, class InnerHandler>
struct on_dyn_read_done
{
  typedef associated_executor_t<InnerHandler> executor_type;
  executor_type get_executor() const noexcept { return get_associated_executor(handler_); }

  DynamicBuffer buffer_;
  InnerHandler handler_;

  void operator()(error_code ec, size_t n)
  {
    buffer_.commit(n);
    handler_(ec, n);
  }
};

template <class Stream, class DynamicBuffer, class CompletionToken>
auto async_dyn_read(
    Stream& stream,
    DynamicBuffer buffer,
    CompletionToken&& token)
{
  async_completion<CompletionToken, void(error_code, size_t)> init(token);
  auto b = buffer.prepare(1024);
  stream.async_read_some(b,
      on_dyn_read_done{
        std::move(buffer),
        std::move(init.completion_handler)});
  return init.result.get();
}
```

Here, `async_dyn_read` is a thin layer of abstraction that provides dynamic buffer semantics atop a raw, byte-oriented stream. The abstraction layer is not required to allocate storage of its own. Many of the existing freestanding "algorithms" in the networking TS are implemented in this way.

By combining allocation into the completion token mechanism, we are unable to write an efficient callback based composition like that above. With P0055R0's proposed design, allocations are forced to the asynchronous API boundaries. If implemented in terms of the proposed `completion_token_transform`, both the `async_dyn_read` and `stream.async_read_some` functions will separately allocate memory for the operation. For each layer of abstraction we add, we introduce an additional allocation.

5.4 Encodes internals of operation into return type of initiating function

As described in section 2.1, the networking TS's completion token mechanism is designed as a regular model that mirrors that of normal functions. The result of an asynchronous operation is specified purely in terms of its completion signature. Put another way, the completion signature is to an asynchronous operation as a return type is to a normal function. The implementation of a function has no bearing on its return type or completion signature.

On the other hand, P0055R0's design means that the internals of an asynchronous operation may influence its return type. That is, the lambda passed to `completion_token_transform` may be encoded into its return type.

```
template<class CompletionToken>
auto async_xyz(T1 t1, T2 t2, CompletionToken&& token) noexcept(auto)
{
  return completion_token_transform<void(R1 r1, R2 r2)>(
       forward<CompletionToken>(token),
       [=](auto typeErasedHandler) { async_xyz_impl_raw(t1, t2, typeErasedHandler); });
}
```

This lambda may be encoded into the return type

This is a deliberate design choice of P0055R0, and has a couple of negative consequences.

5.4.1 Inhibits composition

If we have two normal functions that return the same type:

```
int a();
int b();
```

we can compose them into a third function that also returns that same type:

```
int c()
{
  if (some_condition)
    return a();
  else
    return b();
}
```

We may likewise compose asynchronous operations that use the completion token model. For example:

```
template <class CompletionToken>
auto throttled_post(CompletionToken&& token)
{
  if (throttle_required())
    return my_simple_timer.async_wait(std::forward<CompletionToken>(token));
  else
    return post(std::forward<CompletionToken>(token));
}
```

where both `my_simple_timer.async_wait()` and `post()` have a completion signature of `void()`.

Or this example:

```
class http_client
{
  bool encrypt_;
  ssl::stream<tcp::socket> encrypted_;
  tcp::socket unencrypted_;

  template <class Buffers, class CompletionToken>
  auto async_read_some(Buffers buffers, CompletionToken&& token)
  {
    if (encrypt_)
      return encrypted_.async_read_some(buffers, std::forward<Token>(token));
    else
      return unencrypted_.async_read_some(buffers, std::forward<Token>(token));
  }
};
```

where all `async_read_some` operations share the completion signature `void(error_code, size_t)`.

As P0055R0's `completion_token_transform` may encode an asynchronous operation implementation into its return type, we can no longer compose operations in this way. For example, we can no longer assume that two different `async_read_some` operations return the same type.

5.4.2 Increases risk of ABI breakage

As P0055R0's model encodes implementation details into return types, there is a higher risk of making ABI breaking changes. For example, when a library implementer fixes a bug in the lambda passed to `completion_token_transform`, it may change size, and the return type of the asynchronous operation is similarly affected.

6 Conclusion

In summary, benchmarking shows that P0055R0's proposed design offers no tangible performance benefit over existing facilities. Moreover, the conflation of the two customisation points:

- return type and value
- allocation of memory

into a single customisation point carries significant design costs. Consequently, the design changes proposed in P0055R0 should not be adopted by the proposed networking TS.

7 Acknowledgements

The author would like to thank: Arash Partow for extensive discussion, feedback, and for his assistance in designing and running the benchmarks; and Jamie Allsop for providing feedback on drafts of this paper.

8 Appendix: Benchmark results

```
C:\build>start /b /wait /high /affinity 2 os_static\callback_custom_alloc.exe
5543 ms for 1e9 iterations, which is 5.543 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_custom_alloc.exe
```

```
5549 ms for 1e9 iterations, which is 5.549 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_custom_alloc.exe
5577 ms for 1e9 iterations, which is 5.577 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_custom_alloc.exe
5646 ms for 1e9 iterations, which is 5.646 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_custom_alloc.exe
5644 ms for 1e9 iterations, which is 5.644 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_custom_alloc.exe
5513 ms for 1e9 iterations, which is 5.513 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_custom_alloc.exe
5513 ms for 1e9 iterations, which is 5.513 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_custom_alloc.exe
5707 ms for 1e9 iterations, which is 5.707 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_custom_alloc.exe
5556 ms for 1e9 iterations, which is 5.556 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_custom_alloc.exe
5581 ms for 1e9 iterations, which is 5.581 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_default_alloc.exe
5545 ms for 1e9 iterations, which is 5.545 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_default_alloc.exe
5536 ms for 1e9 iterations, which is 5.536 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_default_alloc.exe
5526 ms for 1e9 iterations, which is 5.526 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_default_alloc.exe
5679 ms for 1e9 iterations, which is 5.679 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_default_alloc.exe
5548 ms for 1e9 iterations, which is 5.548 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_default_alloc.exe
5600 ms for 1e9 iterations, which is 5.600 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_default_alloc.exe
5551 ms for 1e9 iterations, which is 5.551 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_default_alloc.exe
5534 ms for 1e9 iterations, which is 5.534 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_default_alloc.exe
5563 ms for 1e9 iterations, which is 5.563 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\callback_default_alloc.exe
5699 ms for 1e9 iterations, which is 5.699 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\coroutine.exe
6075 ms for 1e9 iterations, which is 6.075 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\coroutine.exe
6076 ms for 1e9 iterations, which is 6.076 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\coroutine.exe
6066 ms for 1e9 iterations, which is 6.066 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\coroutine.exe
```

```
6075 ms for 1e9 iterations, which is 6.075 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\coroutine.exe
6102 ms for 1e9 iterations, which is 6.102 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\coroutine.exe
6065 ms for 1e9 iterations, which is 6.065 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\coroutine.exe
6076 ms for 1e9 iterations, which is 6.076 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\coroutine.exe
6076 ms for 1e9 iterations, which is 6.076 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\coroutine.exe
6066 ms for 1e9 iterations, which is 6.066 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_static\coroutine.exe
6074 ms for 1e9 iterations, which is 6.074 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_custom_alloc.exe
16581 ms for 1e9 iterations, which is 16.581 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_custom_alloc.exe
16230 ms for 1e9 iterations, which is 16.230 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_custom_alloc.exe
16236 ms for 1e9 iterations, which is 16.236 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_custom_alloc.exe
16615 ms for 1e9 iterations, which is 16.615 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_custom_alloc.exe
16632 ms for 1e9 iterations, which is 16.632 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_custom_alloc.exe
16632 ms for 1e9 iterations, which is 16.632 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_custom_alloc.exe
16303 ms for 1e9 iterations, which is 16.303 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_custom_alloc.exe
16294 ms for 1e9 iterations, which is 16.294 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_custom_alloc.exe
16278 ms for 1e9 iterations, which is 16.278 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_custom_alloc.exe
16297 ms for 1e9 iterations, which is 16.297 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_default_alloc.exe
16409 ms for 1e9 iterations, which is 16.409 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_default_alloc.exe
16431 ms for 1e9 iterations, which is 16.431 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_default_alloc.exe
16518 ms for 1e9 iterations, which is 16.518 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_default_alloc.exe
16424 ms for 1e9 iterations, which is 16.424 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_default_alloc.exe
16397 ms for 1e9 iterations, which is 16.397 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_default_alloc.exe
```

```
16395 ms for 1e9 iterations, which is 16.395 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_default_alloc.exe
16501 ms for 1e9 iterations, which is 16.501 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_default_alloc.exe
16507 ms for 1e9 iterations, which is 16.507 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_default_alloc.exe
16509 ms for 1e9 iterations, which is 16.509 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\callback_default_alloc.exe
16425 ms for 1e9 iterations, which is 16.425 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\coroutine.exe
18084 ms for 1e9 iterations, which is 18.084 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\coroutine.exe
18117 ms for 1e9 iterations, which is 18.117 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\coroutine.exe
17980 ms for 1e9 iterations, which is 17.980 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\coroutine.exe
18017 ms for 1e9 iterations, which is 18.017 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\coroutine.exe
17975 ms for 1e9 iterations, which is 17.975 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\coroutine.exe
18039 ms for 1e9 iterations, which is 18.039 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\coroutine.exe
17968 ms for 1e9 iterations, which is 17.968 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\coroutine.exe
17969 ms for 1e9 iterations, which is 17.969 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\coroutine.exe
18088 ms for 1e9 iterations, which is 18.088 ns per iteration

C:\build>start /b /wait /high /affinity 2 os_dll\coroutine.exe
18136 ms for 1e9 iterations, which is 18.136 ns per iteration
```