Using Coroutine TS with zero dynamic allocations

Document Number: **P1365 R0** Date: 2018-11-24
Reply-to: Gor Nishanov (gorn@microsoft.com)
Audience: WG21, EWG

Abstract

It is not immediately obvious that Coroutines TS supports environments where no dynamic memory allocation is allowed. This paper outlines how coroutines can be used in those environments.

1 Overview

Coroutine TS is composed of several layers.

Syntax Layer	User facing syntax of the coroutine
Library Layer	Library guided transformation imbuing the coroutine with high-level semantics (generator, async task, etc.)
Mechanism Layer	Internal coroutine mechanism of transforming a function into a state machine

The Coroutines TS does not expose coroutine state machine object directly and uses compiler-based wrapping of the coroutine state machine guided by a set of customization points.

Syntax Layer	**Library layer**

`task<int> async_sum(channel& s) {`
`  int sum = 0;`
`  for (;;)`
`  sum += co_await s.async_read<int>();`
`  co_return sum;`
`}`
	`template <typename T>`
	`struct task {`
	`  struct promise_type { // defines coroutine semantics`
	`  …`
	`  void *operator new(size_t sz);`
	`  void operator delete(void* p, size_t sz);`
	`  };`
	`  …`
	`};`

To control the placement of the coroutine state, the coroutine designer must provide `operator new` and `operator delete` in the `promise_type` of the coroutine. If provided, the coroutine will acquire storage for its state (if needed[1]) using provided operators `new` and `delete`. The designers of the coroutine type have freedom in how the control can be exposed to the users of the coroutine

In this paper we will outline how to use coroutine library layer in environments where dynamic memory allocations are not allowed.

[1] Compiler is free to elide allocation of the state under the conditions described in Halo paper [P0981R0].

2 Create all coroutines on boot/initializaiton time

JSF coding guidelines rule 206 [JSF] states that no dynamic allocations should happen after initialization.

In this environment, all required coroutines need to be precreated at initialization time. Coroutines have an option to start suspended or, for example, to start and suspend while reading for instructions from a command channel.

Initialization time	**Sample coroutine**

void boot_time_init() {
  // precreate all coroutines on boot
  for (auto &ch: channels)
  command_procesor(ch);
}
	task<> command_processor(channel& s) {
	  for (;;) {
	  auto cmd = co_await s.read_command();
	  …
	  }
	}

In this example, promise_type of the task would overload operator new and will place the coroutines one after another in one contiguous buffer.

3 Embed the coroutine state in the return type

More in-your face approach would involve embedding the coroutine state in the return type.

```
auto f(int a, int b) {
  return make_on_stack<64>([=](auto) -> generator<int> {
    for (int i = a; i < b; ++i)
      co_yield element;
  });
}
```

In this case, make_on_stack helper creates a coroutine wrapper that contains an array of 64[2] bytes in which the coroutine described by the lambda will be placed. If coroutine size exceeds 64 bytes, the code won't compile.

We envision that as a part of the integration of Coroutines TS and contracts there will be small addition to the coroutine TS wording that makes the program ill-formed.

```
template <size_t MaxSize>
void *operator new(size_t sz, Storage<MaxSize> put_it_here) [expects:sz <= MaxSize]] {
  return put_it_here.addr();
}
```

Note that in "[[expects: sz <= MaxSize]]" contract sz is a compile time constant, but, it is a constant that is known at translation time, but not necessarily during semantic analysis time. Here is a rough wording to make it a guaranteed compile time error:

In [dcl.fct.def.coroutine]/7 (paragraph defining how coroutine storage is obtained), add the following sentence:

> If operator new has a precondition contract AND that contract condition would be a constant expression if "sz" argument were a constant time expression, then, the contract will be checked even in the Contract checking "Off" mode and violation will make the program ill-formed.

[2] The [P1362R0] goes into detail why Coroutines TS does not offer an ability to estimate the size of the coroutine during semantic analysis. This may change in the future as compiler technology evolves.

4 Rely on Halo optimization

Heap allocation elision optimization [P0981R0], can be used in combination with other approaches described in this section. By declaring and not defining the body of the operator new and operator delete in the coroutine promise, the developer can guarantee a compile time error if a compiler failed to remove the heap allocation for the coroutine state.

Halo paper [P0981R0] describes conditions and patterns under which heap allocation can be removed by the compiler. To summarize it in one sentence: if a lifetime of the coroutine does not escape the lifetime of the caller, the coroutine state can be placed as a local object in its caller.

Here is an example of a generator this optimization applies:

```
generator<int> range(int from, int to) {
  for (int i = from; i < to; ++i)
    co_yield i;

int main() {
  auto s = range(1, 10); // lifetime contained in main
  return std::accumulate(s.begin(), s.end(), 0);
}
```

Same applies for tasks:

```
task<> subtask(Executor ex) {
  ... co_await execute_on(ex);
  ...
}

task<> big_task(Executor ex) {
  ... co_await subtask(ex); // subtask frame allocation elided
  ...
}
```

For more comprehensive example, see [Godbolt] (scroll to the bottom).

5 Bibliography

[P0981R0] Richard Smith, Gor Nishanov. “Halo: coroutine Heap Allocation eLision Optimization” (WG21 paper, 2018-03-18).

[P1362R0] Gor Nishanov. “Incremental Approach: Coroutines TS + Core Coroutines” (WG21 paper, November 2018).

[JSF] “Join Strike Fighter C++ Coding Guidelines” (WG21 paper, December 2005).

[Godbolt] “Generator composition example” (https://godbolt.org/g/26viuZ).