Why does calling a coroutine allocate a lot of stack space even though the coroutine frame is on the heap?

devblogs.microsoft.com/oldnewthing/20231115-00

November 15, 2023

Raymond Chen

Consider the following:

```
#include <coroutine>
#include <exception>

// Define a coroutine type called "task"
// (not relevant to scenario but we need *something*.)
struct task { void* p; };

namespace std
{
    template<typename...Args>
    struct coroutine_traits<task, Args...>
    {
        struct promise_type
        {
            task get_return_object() { return { this }; }
            void unhandled_exception() { std::terminate(); }
            void return_void() {}
            suspend_never initial_suspend() { return {}; }
            suspend_never final_suspend() noexcept { return {}; }
        };
    };
}
// End of "task" boilerplate.

void consume(void*);

task sample()
{
    char large[65536];
    consume(large);
    co_return;
}
```

The `sample` coroutine function consumes a large buffer, but it never suspends, and the coroutine frame is destroyed before the function returns. This means that the function is a good candidate for *heap elision*, specifically Heap Allocation eLision Optimization (HALO), which permits the compiler to optimize out the entire heap allocation and put the coroutine frame on the stack:

```
# clang -O3
sample():
        sub     rsp, 65576
        lea     rdi, [rsp + 32]
        call    consume(void*)@PLT
        lea     rax, [rsp + 16]
        add     rsp, 65576
        ret
```

The Microsoft Visual C++ compiler's code generation for coroutines (in version 19.34.31931.0) performs the initialization in a function named `$InitCoro$2`, with the declaration

```
void sample$InitCoro$2(
    T* __coro_return_value,
    bool const& __coro_heap_ellision,
    void* __coro_frame_ptr,
    Args&&... args);
```

where `T` is the return type of the coroutine function (`task` in this example) and `Args&&...` are the parameters to the coroutine function (empty, in this case).

The `__coro_frame_ptr` points to a block of memory that will be used as the coroutine frame.

The `__coro_heap_ellision` parameter[1] is `true` if the frame is on the stack and `false` if the frame is on the heap.

The `__coro_return_value` points to the place to put the `T` returned by (or constructed from) the promise's `get_return_object()`.

The `$InitCoro$2` function initializes the coroutine frame, obtains the return object, and then runs the coroutine until its first suspension point, and then returns.

The code generation for the coroutine function goes roughly like this:

```
task sample()
{
    char __coro_elision_buffer[sizeof(coroutine_frame)];
    bool __coro_heap_elision = false;
    void* __coro_frame_ptr = __coro_heap_elision
        ? __coro_elision_buffer
        : operator new(sizeof(coroutine_frame));
    char alignas(task) __coro_return_value[sizeof(task)];

    sample$InitCoro$2(&__coro_return_value,
        __coro_heap_elision,
        __coro_frame_ptr);

    return reinterpret_cast<task&>(__coro_return_value);
}
```

At optimization level 2, the Microsoft Visual C++ compiler propagates the `__coro_heap_elision` constant into the ternary, resulting in

```
task sample()
{
    char __coro_elision_buffer[sizeof(coroutine_frame)];
    char alignas(task) __coro_return_value[sizeof(task)];

    sample$InitCoro$2(&__coro_return_value,
        false,
        operator new(sizeof(coroutine_frame)));

    return reinterpret_cast<task&>(__coro_return_value);
}
```

However, it fails to realize that the `__coro_elision_buffer` is now a dead variable, so the function allocates stack space for a buffer it never uses.[2]

This can be a problem if your coroutine has a large frame, because it will consume a lot of stack that may cause you to stack overflow prematurely. If you want to make sure a coroutine local variable goes on the heap, you should put it explicitly on the heap.

```
task sample()
{
    auto large = std::make_unique_for_overwrite<char[]>(65536);
    consume(large.get());
    co_return;
}
```

[1] Yes, the word *elision* is misspelled.

[2] This missed optimization is on the backlog.