CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Coroutines in a nutshell

Péter Kardos

Coroutines in C++

Split into two layers:

1. Compiler support layer
   - A standardized interface to talk to the compiler
   - You have to define a coroutine promise
   - You have to define classes that use the promise (e.g. future, generator)
   - The compiler translates `co_return`, `co_yield` and `co_await` statements to code that calls your defined classes
   - This is complicated, so we'll skip this
2. Library layer
   - Assume someone did the complicated stuff above
   - Now you have a usable coroutine library
   - This is what we'll talk about

Coroutine libraries

- You can not practically use the barebones coroutines in C++
- You must implement a library on top
  - This means you can have different interfaces and capabilities for different coro libraries
- The catch: C++ does not have a standard one yet
  - Worse: there are not many open source ones either
- We'll use a fictional library for code examples:
  - It defined `task<T>`
  - It also defined `generator<T>`

What is a coroutine?

In short: a function that can be paused and resumed later

This is the simplest coroutine:

```
task<int> zero(int value) {
    std::cout << "Hi from coroutine!" << std::endl;
    co_return 0;
}
```

You can "call" it from a function:

```
void function() {
    task<int> tsk = zero();
    std::cout << "Hi from function!" << std::endl;
    int value = tsk.get();
}
```

What happens here:

1. `zero()` spawns the coroutine, but it's initially paused
2. The code of `function` continues immediately, without executing the coroutine
3. The coroutine executes somewhere else (on another thread maybe)
4. `function` retrieves the result of the coroutine
    - It may also blocks to wait if the result is not ready yet

Possible output:

```
Hi from function!
Hi from coroutine!
```

Combining coroutines

Consider two coroutines:

```
task<int> produce_value() {
    int value = ...; // Complex stuff
    co_return value;
}

task<int> process_value() {
    int init = ...; // Complex stuff
    task<int> tsk = produce_value();
    int value = co_await tsk;
    int result = ...; // Complex stuff
    co_return result;
}
```

What happens in `process_value`:

1. Does initialization
2. Spawns coroutine
3. Pauses itself
    - And tell `tsk` it's waiting for it

Later, possibly in another thread:

1. `produce_value` finishes executing, passes its result in `tsk`
2. `tsk` sees `process_value` is waiting for it
3. `tsk` resumes `process_value`
4. `process_value` retrieves the result in `tsk` (`value = co_await ...`)
5. `process_value` does its complex stuff

Coroutines as user space threads

A thread can be in these states:

- Running: a CPU core is currently walking through its instructions
- Paused: no CPU *should* be working on it
- Ready: a CPU core can pick it up, if it has capacity

Observe: a coroutine can be in these exact same states!

Difference:

- A CPU core switching from one thread to the other is expensive
  - Must go through the operating system kernel and thread scheduler
- A CPU core switching from one coroutine to the other is cheap
  - Just save the state of the current one, load the state of the next, and go
- Coroutines basically realize a cooperative thread scheduler purely in user space
  - This is much lighter weight than the preemptive kernel scheduler

Use cases for task-like coroutines

- Scaling event-driven systems: large number of small jobs, does not saturate CPU capacity
  - Think of a web server
  - You could launch a thread for each client
  - Threads are resource heavy, this doesn't scale
  - Launch a coroutine for each client instead. You can have millions of coroutines at the same time without issue.
- Efficiently scheduling parallel work: medium number of heavy jobs, spins all cores at 100%
  - Think of a game engine
  - Many tasks, many of them parallel-decomposable:
    - Process sound: convolution reverb, mixing...
    - Update physics engine: space partitioning, detect collisions...
    - Update graphics engine: occlusion culling, issue render commands...
  - You can reframe the tasks as coroutines that `co_await` each other
  - Set a thread-pool to eagerly *resume* 'ready' coroutines from a global queue

Generators

Coroutines also help lazy-generating sequences:

```
generator<int> fibonacci() {
    int a = 1, b = 1;
    while (true) {
        co_yield a;
        std::tie(a, b) = std::tuple(b, a + b);
    }
}
```

```
generator<int> g = fibonacci()
auto fib_n = begin(g); // Iterator!
for (int i=0; i<100; ++i) {
    std::cout << *fib_n << std::endl;
    ++fib_n;
}
```

- `co_yield` :
  - Pauses the coroutine like `co_await`
  - Creates a result like `co_return`
  - Not waiting for another coro, manually resumed
- `generator<T>` :
  - It's a *range*: has `begin()` and `end()` methods
  - These return iterators to the values in the range
  - The range is made up of the values `co_yield` ed
- Lifecycle generators:
  1. Spawn the coroutine: start in a running state
  2. Hits `co_yield` , produces a value and pauses
  3. Dereference iterator: extract the latest `co_yield` 'ed value
  4. Increment iterator: resume coroutine until next `co_yield`

Use cases for generators

- Express procedurally generated infinite sequences (e.g. random numbers)
- Iterate over data structures: may be more convenient to express it as a coroutine than directly with `iterator` classes
- Awaitable generators:
  - Instead of dereferencing the iterator, you `co_await` it
  - Your web-client's requests can be expressed as a generator

Remarks

- This was a very brief introduction, there is a lot more to it:
  - How to implement `task<T>` and `generator<T>` yourself
  - How to implement a coroutine scheduler
  - Theory of operation of threads
  - Theory of operation of stackful and stackless coroutines
- To use coroutines in practice, you need a library that implements `task<T>` and `generator<T>`
  - You can write your own
  - You can use:
    - cppcoro: https://github.com/lewissbaker/cppcoro
    - libcoro: https://github.com/jbaldwin/libcoro