vollmann engineering gmbh
>

C++20 Coroutines for Small Embedded Systems

An Interesting New Mechanism

emBO++ 2020

March 2020

Detlef Vollmann

vollmann engineering gmbh

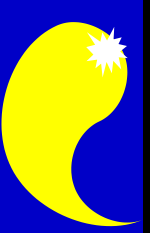
vollmann engineering gmbh
>

C++20 Coroutines for Small Embedded Systems

An Interesting New Mechanism

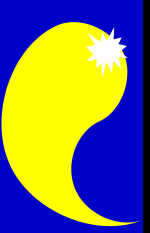
Detlef Vollmann
vollmann engineering gmbh
Luzern, Switzerland

dv@vollmann.ch
http://www.vollmann.ch/



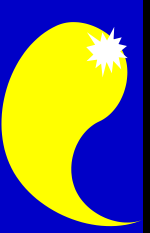
Overview

- References
- Motivation
- Example
- Stackless
- Mechanics
- Memory
- Stackful

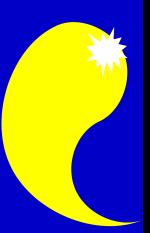


Finding Information

- The Standard http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2020/n4848.pdf
- The TS http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2018/n4775.pdf
- The (original) proposal http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4286.pdf
- Lewis Baker
  - https://lewissbaker.github.io/
- I still learn about coroutines

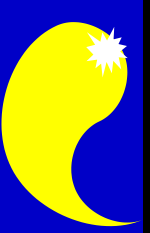


Overview



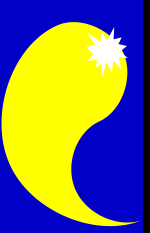
ASIO Without Coroutines

```
void start() { // start async read;
  socket.async_read_some(net::buffer(data),
    [] (size_t length) { handleRead(length); });
}

void handleRead(size_t length) {
  // start async write
  net::async_write(socket,
    net::buffer(data),
    [] () { handleWrite(); });
}

void handleWrite() { // start async read
  socket.async_read_some(net::buffer(data),
    [] (size_t length) { handleRead(length); });
}
```

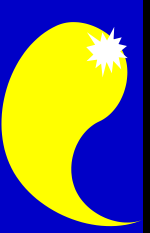
ASIO With Coroutines

```
awaitable<void> echo(tcp::socket socket
                     , await_context ctx) {
  size_t length;
  char data[128];
  while (true) {
    length = co_await socket.async_read_some(
                       net::buffer(data), ctx);
    co_await async_write(socket
                , net::buffer(data, length)
                , ctx);
  }
}
```

Overview

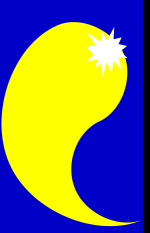
References
Motivation
Example
Stackless
Mechanics
Memory
Stackful



Data Concentrator

```
RTExecutor rtExec(80);

  ConcentratorT dataConcentrator{
      pool
      , readDev1
      , ReadDev(rtExec, in2)
      , StoreData(out)};
```

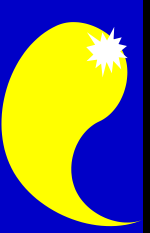
- Data concentrator
  - Two producers, one filter, one consumer
  - One producer has higher priority



Producer

```
Dummy read1(CoQueue<DataT> &q)
{
  // starts the reader (timer for our demo)
  TimerAwaitable read1{evLoop
                       , itimerspec{{2, 0}, {2, 0}}};
  while (1) {
    uint64_t dummy = co_await read1; // wait for next item to a
    // now create a value to push
    DataT val = { 1, dummy };
    co_await q.push(val);
  }
}
```

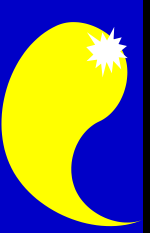
- co_await for data
- co_await for push
- This coroutine never ends!



Consumer

```
Dummy log(CoQueue<DataT> &q)
{
  int count = 0;
  while (1)
  {
    DataT v = co_await q.pull();
    std::clog << "Data from " << v.id << ": " << v.data << '\n'
    ++count;
    if (count == 10)
    {
      evLoop.stop();
    }
  }
}
```

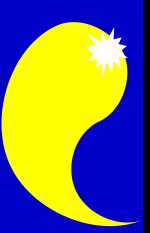
- co_await for pull



Putting Together

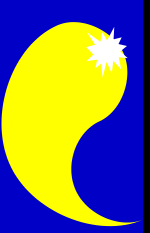
```
int main()
{
    CoQueue<DataT> queue;

    read1(queue); // start producer 1
    read2(queue); // start producer 2
    log(queue);   // start consumer

    evLoop.loop<TimerAwaitable>(); // event loop

    return 0;
}
```

- Not a coroutine
- Call the couroutines
- Start the loop



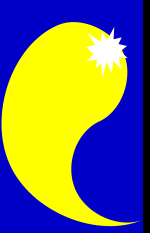
Initial Control Flow

- Calling `read1`
- `read1` suspends on first `co_await`
- `read1` returns!
- Same for `read2` and `log`
- `read1`, `read2` and `log` are now suspended
- `evLoop.loop()` blocks
- An event that unblocks `evLoop.loop()` will then resume `read1` or `read2`



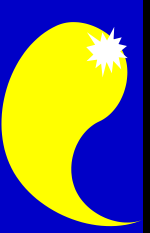
Overview

References
Motivation
Example
Stackless
Mechanics
Memory
Stackful



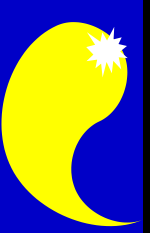
Stackless Coroutines

- Stackless coroutines don't have a full stack
- Stackless coroutines use the normal stack of the calling (resuming) function
- Stackless coroutines only set aside the state needed for resumption



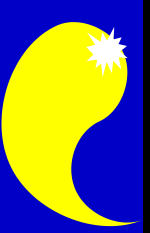
Coroutine Frame

- A (suspended or running) coroutine is represented by a coroutine frame
- Contains all local state
  - not more (no stack reserve for calling functions)!
- Represented by `std::coroutine_handle<>`



Overview

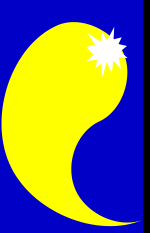
References
Motivation
Example
Stackless
Mechanics
Memory
Stackful



co_await

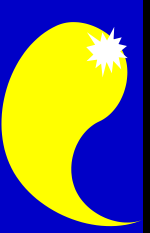
- co_await <expr>
- <expr> must be of Awaitable type
- Simplified co_await sequence:

```
ResultT do_await(Awaitable a)
{
  if (!a.await_ready()) {
    // suspend coroutine magic (store resumption point in frame)

    if (a.await_suspend(coroFrame)) {
      // return to caller or resumer
    }

    // <- this is the resumption point
  }
  return a.await_resume();
}
```

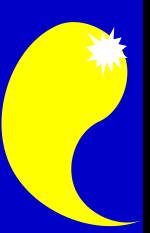
Awaitable

- `await_ready()` is an initial check whether to suspend at all
- `await_suspend()` gets the handle for storage
- `await_suspend()` can start the asynchronous operation
- `await_resume` provides the value for `co_await`



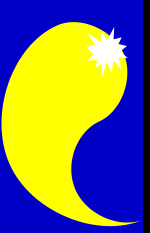
Queue Implementation

```
template <typename T> class CoQueue {
public:
  PushAwaiter push(T val) {
    //assert(!full);
    data = val;
    full = true;
    return {this};
  }

  PullAwaiter pull() { return {this}; }

private:
  CoHandleT consumer;
  T data;
  bool full = false;
};
```

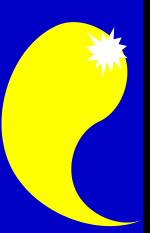
PushAwaiter

```
class PushAwaiter {
public:
  PushAwaiter(CoQueue *q_)
    : q{q_}
  {}
  bool await_ready() const noexcept {
    return !(q->consumer);
  }
  void await_suspend(CoHandleT) noexcept {
    // resume reader
    if (q->consumer) { q->consumer(); }
  }
  void await_resume() noexcept { }

private:
  CoQueue<T> *q;
};
```

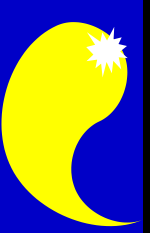
PullAwaiter

```
class PullAwaiter {
public:
  PullAwaiter(CoQueue *q_)
    : q{q_}
  {}
  bool await_ready() const noexcept {
    return q->full;
  }
  void await_suspend(CoHandleT coro) noexcept {
    q->consumer = coro;
  }
  T await_resume() {
    // this only works for a push-driven queue
    q->full = false;
    return q->data;
  }
private:
  CoQueue<T> *q;
};
```

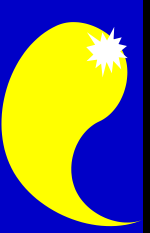
Loop Implementation

```
class EpollLoop {
public:
  EpollLoop()
    : fd{epoll_create(1)}
  {}
  template <class EpollAwaitable>
  void add(EpollAwaitable &aw) {
    epoll_event ev = {EPOLLIN, {&aw}};
    epoll_ctl(fd, EPOLL_CTL_ADD, aw.fd, &ev);
  }
  void stop() {
    doStop = true;
  }
private:
    int fd;
    bool doStop = false;
};
```

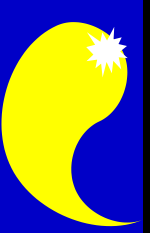
Loop Implementation

```
class EpollLoop {
public:
  template <class EpollAwaitable>
  void loop() {
    epoll_event waitEv;
    while (!doStop)
    {
      int cnt = epoll_wait(fd, &waitEv, 1, -1);
      if (cnt > 0) {
        EpollAwaitable *src
           = static_cast<EpollAwaitable *>(waitEv.data.ptr);
        src->handleEvent();
      }
    }
  }
};
```

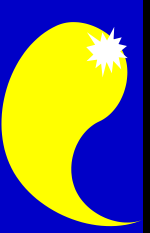
Timer Implementation

```
class TimerAwaitable {
public:
  TimerAwaitable(EpollLoop &ep, itimerspec tSpec)
    : fd{timerfd_create(CLOCK_MONOTONIC, TFD_NONBLOCK)}
  {
    timerfd_settime(fd, 0, &tSpec, 0);
    ep.add(*this);
  }
  void handleEvent() {
    size_t cnt = ::read(fd, &timeValue, sizeof(timeValue));
    coro();    // now resume ourselves
  }
private:
  coroHandleT coro;
  uint64_t timeValue;
  int fd;
};
```

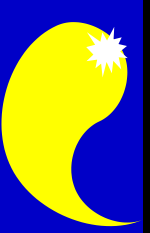
Timer Implementation

```
class TimerAwaitable {
public:
  bool await_ready() const noexcept {
    return false;
  }
  bool await_suspend(coroHandleT self) noexcept {
    coro = self;
    // we would start the next interval here
    // but we have a self-repeating timer

    return true; // stay suspended
  }
  uint64_t await_resume() noexcept {
    return timeValue;
  }
};
```

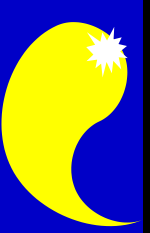
Loop Control Flow

- `epoll()` blocks (system call)
- After (system) wakeup calls respective `handleEvent()`
- Resumes `read` function
- `co_awaits CoQueue::push()`
- Suspends `read` function
- Resumes `log` function
- `co_awaits CoQueue::pull()` again
- Returns to `handleEvent()`
- Returns to `loop()`
- Back to `epoll()`



Overview

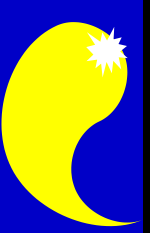
References
Motivation
Example
Stackless
Mechanics
Memory
Stackful



Initial Coroutines Calling

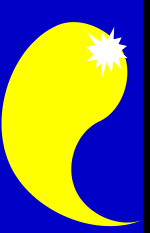
- No difference on call site from normal function
- Coroutine must return special type
  - must contain internal `promise_type`
- Simplified procedure for `Dummy read1(queue)`:

```
void callCoro()
{
    h = coroutine_handle<>{new CoroFrameT{queue}};
    Dummy::promise_type p;
    Dummy ret = p.get_return_object();
    co_await p.initial_suspend();
    // execute coroutine body
}
```



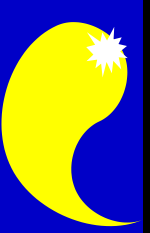
Promise Requirements

- `initial_suspend()` allows to always suspend a coroutine immediately (for lazy evaluation)
- `final_suspend()` is called whenever the coroutine ends
- `return_void()` is called for `co_return`
  - there's also `return_value()`
- `yield_value()` is called for `co_yield`
- `unhandled_exception()` is called whenever an exceptions leaves the coroutine body
- `get_return_object()` must provide the actual object to be returned from the coroutine initially



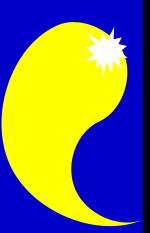
MinimalPromise

```
struct Dummy
{
  typedef DummyPromise promise_type;
};

class DummyPromise
{
  typedef std::suspend_never NoSuspend;
public:
  NoSuspend initial_suspend() { return {}; }
  NoSuspend final_suspend() { return {}; }
  void return_void() { }
  Dummy get_return_object() { return {}; }
  void unhandled_exception() noexcept { }
};
```

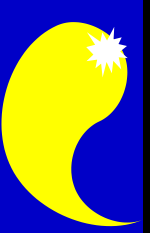
Memory Allocation

- A coroutine is compiled to first allocate memory for the coroutine frame
- Normally by calling global `operator new()`
- Can be customized by overloading `operator new()` for your promise type
- Sometimes the compiler can elide the memory allocation completely
  - if it knows that the lifetime of the coroutine is bounded by the calling function anyway
  - then the coroutine frame is allocated on the stack of the calling function
  - fairly simple if the promise return is an RAII type
  - may require inline coroutine



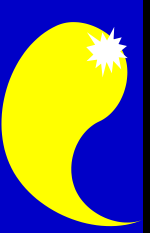
Memory Allocation at Setup

- Many embedded systems allow memory allocation only in a start phase
- Our example creates (initially calls) all coroutines before entering the event loop
- Inside the loop no memory allocation happens



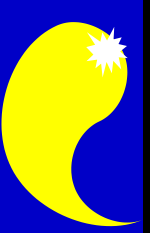
Repeated Memory Allocation

- Any (initial) call of a coroutine may allocate memory
  - not resumption
- Calling a coroutine inside a loop may allocate memory repeatedly
  - if not elidable
- E.g. async function calls deep inside your logic



Destruction Coroutine Frames

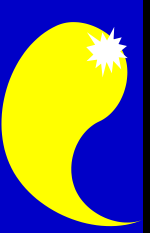
- `coroutine_handle<>` is non-owning!
- The coroutine frame is destructed when
  - the coroutine body is left
  - `destroy()` is called on the `coroutine_handle`



Coroutine Frame Cleanup

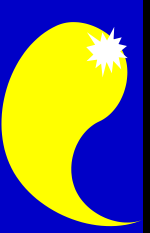
- Our example leaks the coroutine frames
- The promise type can work as RAII type for the coroutine frame

```
struct CoHandle {
    typedef DummyPromise promise_type;
    typedef std::coroutine_handle<promise_type> CoHandleT;

    ~CoHandle() {
        handle.destroy();
    }

    CoHandleT handle;
};

CoHandle DummyPromise::get_return_object() {
    return {CoHandle::CoHandleT::from_promise(*this)};
}
```

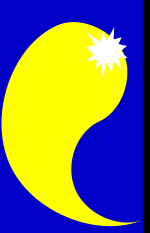
Dangling Coroutine Frames

- `coroutine_handle<>` is non-owning!
- You may destroy it while somebody else still has another handle
- This may happen with multi-threading
  - the `epoll()` loop may run in its own thread
  - `await_suspend()` may start an asynchronous read
  - now your own handle may become dangling immediately
- Be careful!
- Same for double delete!



Overview

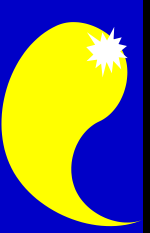
References
Motivation
Example
Stackless
Mechanics
Memory
Stackful



Stackless vs.Stackful

- Stackless coroutines have only the coroutine frame
  - not the full call stack
- Stackless coroutines may allocate heap memory whenever they are called

- Each stackful coroutine has it's own (complete) stack
- Much easier to setup at startup time
- Always has full stack available
- Wastes a lot of RAM
- Not standardized (yet)



Questions

- ??????????????????????????????????

