STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVII**, Number 2, 2022
DOI: 10.24193/subbi.2022.2.03

COROUTINES COMUNICATIONS. DESIGN AND IMPLEMENTATION ISSUES IN C++20

RADU LUPŞA AND DANA LUPŞA

Abstract. This paper explores the communication mechanisms and patterns available to coroutines to cooperate with one another. It investigates the issues in designing and implementing a framework for using C++20 coroutines effectively, for generators, asynchronous function calls, and especially asynchronous generators.

1. Introduction

Coroutines are a programming concept that allows execution to be suspended and resumed. A coroutine can transfer the control to other piece of code and, when getting back the control at a later time, it continues from the next instruction and with all the data restored, including the execution stack.

Coroutines were originally proposed in 1963 and also were studied quite thoroughly in 1960's to beginning of 1980's. The interest to coroutines has resurfaced in recent years, with several mainstream languages offering some support to coroutines.

Even though coroutines dates back to 1963 [9] , they are still found useful in modern applications. For example, a recent work on coroutine study the use of corotuine for a web crawler [15]. Coroutines were proven to perform better than single-threaded and multi-threaded versions. Performance improvement was also demonstrated for coroutine use in `I/O` requests [14]. When used on Android, coroutines achieve better performance when compared to some existing concurrency frameworks [8]. Also, lot of recent academic works study the application of coroutines on resource-constrained platforms, in the Internet of Things and Embedded Systems [7].

Received by the editors: 4 December 2022.

2010 *Mathematics Subject Classification.* 68N19, 68Q85.

1998 *CR Categories and Descriptors.* D.3.3 [**Software**]: Programming Languages – *Language Constructs and Features*; D.1.3 [**Software**]: Programming Techniques – *Language Concurrent Programming*.

Key words and phrases. asynchronous programming, coroutines, language design.

While there are a lot of works exploring the low level aspects and the applicable aspects of using coroutine in asyncronous processing, not so much is studied about higher-level abstractions and patterns. However, in [13] there is a review about the following patterns: The Producer–Consumer Problem, Generator, Goal-oriented programming, Cooperative Multitasking. They also discuss Exception Handling, but this is more of CPS than coroutines. [10] also present an implementation approach to asynchronous programming and generator based on coroutines.

The coroutine primitives provided in C++20 are very powerful, but they are very complicated to use directly. In this paper, we explore how to use them to build the producer-consumer patterns and cooperative multitasking.

The rest of the paper is organized as follows. The next section makes an overview of the most known coroutine classification and presents the characteristics of coroutines existing in some programming languages. Section 3 details two main scenarios for inter-coroutine communication. Section 4 proposes a framework over C++20 coroutines allowing several use cases: to make an asynchronous call and switch to some other coroutine until the asyncronous call completes, to have a coroutine yielding a sequence of values (to implement a generator) and to combine those two features together. The paper ends with a short review over the main things that are presented in this paper.

2. What is a Coroutine

Essentially, a coroutine is an execute thread, together with the notion of current instruction, execution stack together with arguments, local and temporary variables. However, unlike threads, switching to or from a coroutine is done at the current coroutine request, instead of unpredictable, whenever the operating system decides to.

According to [12], the characteristic property of a coroutine is that it can transfer the control (yield) to other code and, when getting back the control at a later time, it continues from the next instruction and with all the data restored, including the execution stack.

There are 3 important classification criteria for coroutine support in a coroutine implementation [13], [10]:

symmetric vs asymetric: : in symmetric coroutine implementation, any coroutine can transfer control to any other coroutine; with asymmetric coroutines, a caller coroutine can transfer control to a subordinate coroutine, while the subordinate can only transfer back to the caller (or to own subordinates).

stackfull vs stackless: : a stackless coroutine can transfer control only from its main function; a stackfull one can transfer control from within a called function at any depth. However, as a partial workaround, a stackless coroutine can create and call a child stackless coroutine, which can then transfer control to some other coroutine.

first class vs constrained: : a first-class coroutine is explicitly manipulated by the programmer, via handles that can be passed around and stored in variables; constrained coroutines exist only implicit, within some programming constructs such as a *generator* created based on *yield* statements and iterated within a *foreach* loop, or the *async-await* mechanism, both introduced in C# [4], [5] and also available in other languages such as Python.

2.1. **Existing coroutine support in programming languages.** *Windows Fibers* [6] are symmetric, stackfull, first-class coroutines. The API is constructed for the C language. The basic operations are: `CreateFiber()`, that creates a coroutine, given its main function, `SwitchToFiber()`, that suspends the current coroutine giving control to the specified coroutine, and `DeleteFiber()`, that deletes the specified coroutine.

Lua coroutines [11] are asymmetric, stackfull, first-class coroutines. The basic operations are: `coroutine.create()` that creates a child coroutine, `coroutine.resume()`, that transfers control to the specified child coroutine, and `coroutine.yield()`, that transfer control to the parent coroutine. An interesting feature is that an additional parameter given to `coroutine.resume()` (beside the coroutine identifier) gets retrieved inside the coroutine as the return value of `coroutine.yield()` and, vice-versa, any parameter given to `coroutine.yield()` can be retrieved in the parent coroutine as the returned value from `coroutine.resume()`.

C# coroutines [4], [5] are stackless, constrained coroutines (symmetry is a bit unclear). They occur under two forms: *generators* and the *async-await* mechanism. A generator looks like a function returning an `IEnumerable`. The parent coroutine transfers the control to the child by a call to `MoveNext()` on the corresponding iterator; the control is transferred back to the parent by a `yield return` statement in the generator. In an *async-await* scenario, the mechanism is more complex and involves threads and some thread pool mechanism in addition to the coroutines. A coroutine is marked with the `async` keyword and must return a `Task` or a `Task<T>`. An `await` statement inside a coroutine suspends it and may transfer control to a coroutine that is runnable at that time. The coroutine is resumed when the awaited future is completed.

C++20 coroutines [1] are symmetric, stackless, first-class coroutines, although the mechanism for controlling them is complex, poses some constraints and offers distinct mechanisms for symmetric and asymmetric transfer of control. A coroutine main function is identified by having one or more of the `co_await`, `co_yield` or `co_return` statements within its body. The declared return type of a coroutine main function is a user-defined type that serves two purposes: on one hand, it is created when starting the coroutine and should be a wrapper over the coroutine handle, and, on the other hand, it must declare an inner class, `promise_type`, containing some member functions that control the behavior of the coroutine. It is those functions that decide if and to which coroutine should the control be transferred as a result of `co_await`, `co_yield` or `co_return`. These functions also allow data to be transmitted between coroutines in a user-customised way, allowing the construction of higher level mechanisms.

3. Communication between coroutines

Regardless of the lower level mechanisms of handling coroutines, there are two higher level patterns that cover most uses of coroutines:

- Producer-consumer scenarios, where one coroutine produces values for the consumption of another, with passing control together with the values.
- As "poor man's threads", that is to switch from a task, that cannot be continued because it depends on some external data, to another task, that can be continued.

3.1. **Producer-consumer scenario.**

In this scenario, there is a producer and a consumer, and both are written as in full control, each having a main loop. Thus, the producer will have the main loop and will repeatedly execute a statement (usually called *yield*) that pushes a value to the consumer; the consumer also has the main loop and repeatedly pulls data from the producer, often just through a special form of a *for* loop. The producer push (*yield*) operation needs to both give data and switch control to the consumer coroutine; the consumer pull operation needs to give control to the producer and, when the control is transferred back, to return the data pushed by the producer.

3.2. **Poor man's threads scenario.**

This scenario considers several independent operations. We want that, when one operation is blocked waiting for an asynchronous operation, to schedule another. Each operation has its own coroutine and decides when it can pass the control to another. This way, coroutines are used as a cooperative multitasking mechanism.

Each coroutine acts as a line of execution, similar to a thread. Its main advantage, though, is that it is handled in user space, which means that it is cheaper to switch from a coroutine to another. Also, since control is yielded only at some specific points within the code, concurrency control can be relaxed.

When a coroutine executes an asynchronous operation (an operation that has to wait for an external event, such as read from input, from a socket, sleep for an amount of time, wait for a computation executed on some other thread), the coroutine must do three operations:

(1) start the asynchronous operation;
(2) arrange so that the completion notification of the operation marks the coroutine runnable again;
(3) invokes a scheduler that switches to a runnable coroutine.

It is important to note that, in this scenario, coroutines are mostly independent from one another, so, switching from a coroutine to the next is not accompanied with passing information. On the contrary, even deciding which coroutine is the next one is independent of the main bussines of the current coroutine, so, it is delegated to a scheduling mechanism. Furthermore, the scheduler can also move a coroutine from one thread to another.

4. Designing asynchronous generators

The basic usages of constrained coroutines in programming languages like C# and Python is for *generators* and for the *async-await* pattern.

We will look into creating these mechanisms with C++20 coroutines and combining them to form asynchronous generators.

Note that the C# language, that invented the async-await mechanism, does not support asynchronous generators yet, although the work in this direction is under way. Python has it implemented [3], and it comes in a straightforward way from the generators and the async-await. Both use a scheduler mechanism to decide which coroutine gets the control when the current one gets suspended, but neither gives the programmer explicit control to choose the scheduler.

There also exists an implementation for asynchronous generators using C++20 coroutines — the `CppCoro` project [2]. However, our framework has two important differences over what `CppCoro` provides:

(1) we provide primitives for conveying data from sources or to destinations that are not implemented as coroutines, while `CppCoro` only provides the possibility for a non-coroutine source to signal to a coroutine that it can proceed, and similar for a coroutine source to a non-coroutine consumer;
(2) we allow a scheduler to decide which coroutine would take over the current thread when the current coroutine gets suspended, thus using the symmetric coroutines mechanism in C++20 coroutines; `CppCoro`

returns the control to the parent (or latest resumer), using the asymmetric coroutine mechanism in C++20 coroutines.

4.1. **Implementing a *generator* mechanism.** The common way the communications and coroutines work together is the *generator* mechanism. We implemented a small C++20 framework allowing an easy way to write a generator — as a coroutine function that produces values and pushes them via the `co_yield` statement to the consumer — and the consumer code — that can consume the elements via the standard iterators mechanism, for example as a simple *foreach* style `for` loop.

The proposed implementation has the following elements:

- The returned type for the producer coroutines is a template over the produced objects, `Generator<T>`;
- The `Generator<T>` object contains only the coroutine handle;
- The `Generator<T>::promise_type` holds the value between the producer and the consumer, as an `std::optional<T>`. An empty optional signifies the end of data.
- The producer coroutine starts suspended (`initial_suspend()` in `promise_type` returns `std::suspend_always`). This way, we have a lazy evaluation mechanism — the values are produced on demand.
- The consumer calls a function `next()` defined in `Generator<T>`. This resumes the coroutine (suspending the caller).
- The producer coroutine sends the data items via the `co_yield`, invoking the `yield_value()` in the `promise_type`, which stores the value and suspends the producer coroutine, resuming the caller (consumer) code.
- The `next()` call returns in the value in the consumer code.
- At the end, the producer coroutine ends, leading to the runtime invoking `return_void()` in the `promise_type`. This stores a null optional, suspends the producer coroutine and resumes the consumer. The consumer must not invoke `next()` after a null was returned.
- The `next()` call is wrapped in a standard STL iterator.

The above mechanism allows, for instance, a straightforward way to implement a permutations generator. Note that the coroutine function here is recursive, so it acts both as a producer and as a consumer, so it demonstrates both usages:

```
Generator<std::vector<int> >
permutations_rec(std::vector<int> const& prefix, int n) {
    std::vector<int> newPrefix = prefix;
    newPrefix.push_back(0);
    for (int i = 0; i < n; ++i) {
        if (std::find(prefix.begin(), prefix.end(), i) ==
        prefix.end()) {
            newPrefix.back() = i;
            if (newPrefix.size() == n) {
                co_yield newPrefix;
            } else {
                for (auto perm : permutations_rec(newPrefix, n)) {
                   co_yield perm;
                }
            }
        }
    }
}
```

Like for other coroutine-based generators, the downside is that the generator is not copyable, so, the user cannot make a copy at a certain point and have both the original and the copy generate independently the remaining elements.

4.2. **Async-await pattern.** The next element we need is to implement a framework allowing to make an asynchronous call and switch to some other coroutine until the asyncronous call completes.

Its goal is to be able to write something like `v = co_await func()`, where `func()` is an asynchronous operation returning a *future*, with the following effect:

(1) `func()` is called, starting some asynchronous operation;
(2) our coroutine is suspended and the control is passed to some coroutine that is runnable;
(3) when the asynchronous operation completes, our coroutine is marked runnable again;
(4) eventually, when some other coroutine gets suspended or finishes, our coroutine is resumed;
(5) the result of the asynchronous operation is returned and assigned to the variable `v`.

Additionally, to make the mechanism composable, our framework allows the same `v = co_await func()` statement to be executed with `func()` being a coroutine returning a single value. In this case, the effect is similar to a single asynchronous call, but there will be multiple suspends and resumes between the initial call and getting the final result.

The implementation uses two auxiliary objects:

- a `CoroutineScheduler`, for keeping track of runnable coroutines.
- a `PromiseFuturePair`, that will hold the return value from the asynchronous call.

The `CoroutineScheduler` needs to offer two functions, `markRunnable()`, that puts the specified coroutine into a set of coroutines ready to be executed, and `schedule()`, which picks and returns a runnable coroutine or waits until such a coroutine exists.

The `PromiseFuturePair` offers three operations: `set()`, that can be called only once and sets the return value of the asynchronous operation, `get()`, that waits for the result and returns it, and `addCallback()`, that sets a callback to be called when the operation completes; this is needed to mark the coroutine that waits for the result runnable.

With the above, the PromiseFuturePair can be wrapped into an *awaiter* object. The `await_suspend()` will set the callback for the `PromiseFuturePair` to mark the awaiting coroutine runnable again and will invoke the scheduler to schedule some other coroutine. The `await_resume()` will return the value from the future.

Note that the PromiseFuturePair awaiter needs to be linked to the CoroutineScheduler that will provide the next coroutine for the current thread and will also schedule the current coroutine when the awaited condition is fullfilled. The way it is done is by having the coroutine promise object holding a pointer to the CoroutineScheduler and the `await_transform()` that creates the awaiter out of the PromiseFuturePair embed the CoroutineScheduler into the awaiter object. We took two basic assumptions behind this design:

(1) each coroutine is handled by a single scheduler (it cannot go from one scheduler to another);
(2) no coroutine may outlive its scheduler.

The coroutine return object is also an awaiter. Its `await_suspend()` operation, invoked by the caller coroutine, makes the called coroutine runnable, so that the called coroutine will eventually run. It also memorizes the handle of the coroutine invoking it (that gets suspended) as well as its scheduler, so that the caller coroutine is marked ready in its scheduler when the called coroutine returns a value. The `return_value()` of the `promise_type` object marks runnable the coroutine mentioned above.

This design allows control of the scheduler for each coroutine, allowing, among other use cases, to control the thread used by each coroutine. This is important for some GUI frameworks that insist that GUI related functions can be called only on the UI thread.

4.3. **A pipe mechanism.**

To go from asynchronous operations returning single values to asynchronous operations returning multiple values, the mechanism for conveying the result must be changed from `PromiseFuturePair` to a pipe (queue).

While the basics of a pipe are very simple, with the two basic operations, `send()` and `recv()`, there are a few needed additions and clarifications needed.

First, there is the issue of how to signal the end of communication. On the producer end, we have to either call a different function (say, `pipe.close()`) or send a special *EOF* data value. On the consumer end, since the consumer cannot usually know beforehand when the communication will end, the only possibility is to have `pipe.recv()` return the special *EOF* value.

Second, like for the `PromiseFuturePair`, we need a callback to be called when an element is put into the queue. As the pipe is fed from an asynchronous operation, setting the callback can be at race with putting an element into the queue. To simplify the operations, we make the following assumptions:

- When setting a callback to be called for enqueued elements, the callback will be called on each element that is not consumed yet (via `recv()`).
- For this to not create a race condition, the functions `recv()` and `setDataAvailableCallback()` must not be called concurrently.
- To simplify the behavior of `setDataAvailableCallback()`, the capacity of the pipe will be 1 element.

4.4. **Asynchronous generator.**

We combined the elements described above to enable the creation of *asynchronous generators.*

Our design offers the user the possibility to implement an asynchronous generator as a coroutines function having the following elements:

- like a regular generator, it returns multiple values via `co_yield`;
- like an async-await coroutine, when it executes `co_await`, it gets suspended until the result of the function invoked with `co_await` is available;
- the argument of `co_await` may be an asynchronous function producing multiple values. In this case, the calling coroutine gets suspended until the next value is generated and `co_await` returns that value;
- the argument of `co_await` may be another asynchronous generator, in which case `co_await` returns the next generated value. Note that this may lead to an arbitrary sequence of nested asyncronous generators calls and the use of `co_await` leads to a potentially complex sequence of suspending and resuming of their coroutines.

To support the above functionalities, our framework defines a class template, `AsyncGenerator<T>`, that represents an asynchronous generator coroutines that generates objects of type `T`. Given the way coroutines work in C++20, the coroutine function that acts as an asynchronous generator should have a declared return type which is `AsyncGenerator<T>`

AsyncGenerator<T> is designed to be usable in two contexts:

- in a coroutine, as an argument of a co_await statement. Here, it must act as an awaiter.
- in a regular function. Here, it must offer a blocking receive function.

To implement that, the AsyncGenerator<T>::promise_type acts as pipe. The pipe is fed by the yield_value(), which puts a value, and return_void(), which closes the pipe (that is, it adds a special EOF object). After feeding the pipe, the control is transmitted back to the calling coroutine.

For the case AsyncGenerator<T> is the operand of co_await, its method await_suspend() memorizes the handle of the calling coroutine and resumes the own (called) coroutine. This is done so that the owned coroutine can produce the next value. When the next value is produced (via yield_value() or return_void() of AsyncGenerator<T>::promise_type), the memorized calling coroutine is set to runnable again.

For the case AsyncGenerator<T> is used from a regular function, it offers a blocking recv() function that repeatedly calls resume() on the owned coroutine, until a value is made available by the coroutine.

To demonstrate the capabilities of our framework, we present below an implementation of an asynchronous generator that uses coroutines. It takes a sequence of characters produced by another asynchronous generator named source, parses it as a sequence of numbers and returns them to its caller.

```
AsyncGenerator<unsigned> parseFlow(CoroutineScheduler* pScheduler,
        AsyncGenerator<char>& source) {
    unsigned v = 0;
    bool parseStarted = false;
    while (true) {
        ElementOrEof<char> el = co_await source;
        if (el.isEof()) {
            if (parseStarted) {
                co_yield v;
            }
            co_return;
        }
        char c = el.value();
        if (c >= '0' && c <= '9') {
            parseStarted = true;
            v = 10 * v + (c - '0');
        } else if(c == ' ' || c == 10 || c == 13) {
            co_yield v;
            v = 0; parseStarted = false;
        }
    }
}
```

It is worth noting that the above example shows a possible typical usage of our framework, for instance in some networked server or client. The data source in the example would encapsulate an asynchronous read operation from a network connection, the coroutine in the example would do some parsing or preprocessing of the requests, and the user of the data would contain the main processing loop. Classical alternative mechanisms would be threads or callbacks. Threads consume more resources than coroutines. Callbacks are harder to understand because of the inversion of control — in the callback, the programmer needs to keep track of a state and to update it at each incoming callback; with coroutines, the programmer writes the main processing loop.

5. Conclusions

The asynchronous coroutines, proposed in this paper, allow a very natural way of writing code that processes a flow of data coming asynchronously, from some external source, without resorting to threads for this purpose.

This paper also demonstrates how to create, each by itself, the generators and the async-await mechanism, which exist in other languages, but only at its beginning using C++20 coroutines.

It also shows that the C++20 coroutines mechanism, while quite a bit hard to use directly, is very powerful and allows very diverse use scenarios.

References

[1] C and c++ reference, coroutines.
`https://en.cppreference.com/w/cpp/language/coroutines`. Accessed: 2022.

[2] A library of c++ coroutine abstractions for the coroutines ts.
`https://github.com/lewissbaker/cppcoro`. Accessed: 2022.

[3] Pep 525 – asynchronous generators.
`https://peps.python.org/pep-0525/` . Accessed: 2022.

[4] `.NET/C#` guide/language reference. await operator - asynchronously await for a task to completes.
`https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await`. Accessed: 2022.

[5] `.NET/C#` guide/language reference. yield statement - provide the next element.
`https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/yield`. Accessed: 2022.

[6] Windows app development documentation. processes and threads. fibers.
`https://learn.microsoft.com/en-us/windows/win32/procthread/fibers`. Accessed: 2022.

[7] Belson, B., Xiang, W., Holdsworth, J. J., and Philippa, B. W. C++20 coroutines on microcontrollers—what we learned. *IEEE Embedded Systems Letters 13* (2021), 9–12.

[8] Chauhan, K., Kumar, S., Sethia, D., and Alam, M. N. Performance analysis of kotlin coroutines on android in a model-view-intent architecture pattern. In *2021 2nd International Conference for Emerging Technology (INCET)* (2021), IEEE, pp. 1–6.

[9] Conway, M. E. Design of a separable transition-diagram compiler. *Commun. ACM 6*, 7 (jul 1963), 396–408.

[10] Elizarov, R., Belyaev, M., Akhin, M., and Usmanov, I. Kotlin coroutines: Design and implementation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (2021), Association for Computing Machinery, p. 68–84.

[11] Ierusalimschy, R. Programming in `Lua`, 2003.

[12] Marlin, C. D. *Coroutines: A Programming Methodology, a Language Design and an Implementation*, vol. 95 of *Lecture Notes in Computer Science*. Springer, 1980.

[13] Moura, A. L. D., and Ierusalimschy, R. Revisiting coroutines. *ACM Trans. Program. Lang. Syst. 31*, 2 (feb 2009).

[14] von Merzljak, L., Fent, P., Neumann, T., and Giceva, J. What are you waiting for? use coroutines for asynchronous `I/O` to hide `I/O` latencies and maximize the read bandwidth! In *International Workshop on Accelerating Data Management Systems (ADMS)* (2022).

[15] Wang, Z. Web crawler scheduler based on coroutine. *2019 International Conference on Intelligent Computing, Automation and Systems (ICICAS)* (2019), 540–543.

Babeş-Bolyai University, Faculty of Mathematics and Computer Science, 1 Mihail Kogălniceanu, Cluj-Napoca 400084, Romania

Email address: `radu.lupsa@ubbcluj.ro, dana.lupsa@ubbcluj.ro`