Serbest Mansur

Coroutines in C++

Graduation work 2018-19

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT

This paper shows the research of what coroutines are about and investigates Unity Coroutines and existent C++ coroutine libraries. After the research, custom coroutines are implemented in C++ using the Boost.Context library for the context switch.

In the research, different libraries are compared against each other and the differences in characteristics are presented. The last part of the research shows the result of the profiling in speed and memory of all the libraries including the custom coroutine.

The case study shows how a custom coroutine interface is implemented in C++ and how the fiber class of the Boost.Context library is used to create a coroutine interface.

INTRODUCTION

The concept of coroutines was introduced long before, around 1960's. A coroutine is basically a function that should have the capability to be paused and resumed later while still preserving its state.

In a multithreading context, a thread can have its own state and can be scheduled to be executed. But switching between threads has an overhead problem because of the kernel switch. Sometimes it is not desired to use threads, but instead use lightweight threads inside the program. Coroutines are like lightweight threads in that way. Coroutines are contained within a thread and are basically functions that can preserve state across multiple function calls.

The purpose of this paper is to research the coroutines and implement a custom coroutine framework in C++. The research will start with investigating the context of coroutines in the multithreading system. After this is understood, the next step will be to research the characteristics of coroutines: what are the differences between stackful and stackless coroutines? What are the differences between a symmetric and asymmetric coroutines? How is memory handled? How is state preserved? How does context switch happen?

The next step will be to investigate Unity Coroutines and different existent C++ libraries which can be used to create coroutines or are implementing coroutines. The libraries which will be investigated are Boost, C++ TS Coroutines and Windows Fibers. The characteristics and the pros and cons will be determined of all these libraries.

And on top of that, the memory and speed of execution will be profiled of these libraries together with the custom coroutine after it is implemented in C++. Two test cases; generator and same fringe problem, will be used to test the performance and memory use of all these libraries and will be compared against each other, including the custom framework.

After the research is done, it will be decided which characteristics the custom coroutines will have and which possible libraries will be used for the context switch. The last step will be to finally implement the coroutines in C++.

RESEARCH

To understand the coroutines, it is important to understand the context of the coroutines within the threading system. Hence, the research began with researching how threading works. After understanding that, the place of coroutines can be properly placed in the context of threading.

1. THREADS

1.1. GENERAL

Coroutines are a way of concurrent programming. But it is not actually concurrent. There is real concurrency and pseudo concurrency. Real concurrent system will be executed simultaneously across multiple processors if the OS supports multithreading.

Pseudo concurrency gives the illusion of concurrency, but it is not. It is a way of programming and designing an application which can have advantages, because of interleaved executions. A function can be called, paused and resumed. This implies that the function's state is preserved across multiple function calls.

1.2. PROCESS

A process is an instance of a program which has a dynamically-allocated storage (=heap), a stack and an execution context. The execution context comprises the processor's registers and instruction pointer. The stack and the execution context are crucial for the for the execution flow of the program, because the instruction pointer keeps track of which instructions are going to be executed next and therefore those instructions affect the registers.

Within a program, different functions are being called. When a function is invoked or destroyed, it will affect the call stack by adding or removing a block of data on top the stack or from the top of the stack, and therefore also the stack pointer. This sequence of executions is called *thread of execution*.

A process may have multiple thread of executions, which is scheduled by the OS (=Operating System), if the OS supports multithreading.

1.3. KERNEL LEVEL THREAD

A process can consist of multiple thread of executions. These threads are concurrent and are, if the OS has a scheduler, scheduled by the OS. The scheduler is responsible to manage all the threads of different processes.

In a multithreaded process, the threads share the same heap and open files. But each thread has its own stack, this means its own instruction pointer and registers. This is illustrated in *figure 1*.

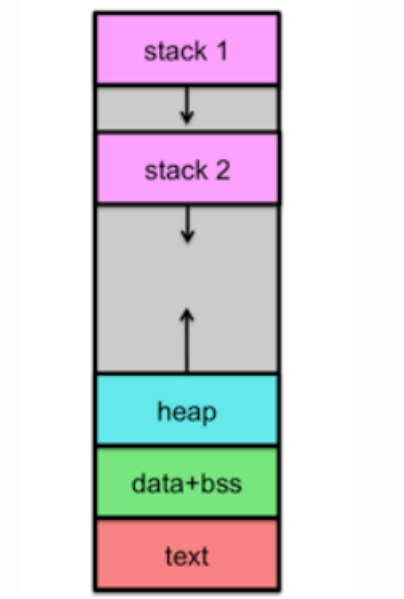


Figure 1: Memory map with two threads [1]

An OS keeps track of all the processes and all the threads they contain. The OS keeps all the information of a process in a block called process control block (=PCB). Since a process can contain multiple threads, each PCB will point to a list of blocks which contain information of the threads within the same process. These blocks of information are called Thread Control Block(=TCB).

Threading has its advantages; it makes some programs easier to write when you want to split the logic into different parts or if you want concurrency. You can create a thread which is meant to execute a specific task, this means that you can split the logic amongst different threads, and it will be executed concurrently. And because a thread has its own stack, all the necessary data can be put in the thread which is meant to use that data. A multithreaded program has also the advantage that the performance will increase with each extra processor. This way, the OS can schedule different threads simultaneously across multiple processes. If no extra processors are available, the OS will give each thread a certain amount of time to execute. This will give the illusion that threads are executed simultaneously in the program, but it is not. It is pseudo concurrency. Different threads will be executed simultaneously only if extra processors are available.

A thread-aware OS schedules threads only, not processes. A process which only has one thread is no different than a thread to the OS. The scheduler should be aware whether the threads belong to the same process or not.

1.4. USER LEVEL THREAD

The threads which were discussed up to this point are know as kernel-level threads. The OS can create multiple threads per process and the scheduler manager when and how they run. System calls are provided to control the creation, deletion and synchronization of threads.

But threads can also be implemented within a process itself, threads which are only known by the process itself and not by the OS. These threads are called user-level threads. With this kind of threads, the program will have its own threading system which will manage the threads within the process. These threads have the advantages of a cheap context switch and avoid the problems that multithreading creates because multiple kernel threads can access the same shared memory.

1.5. CONTEXT SWITCH

Switching from one thread to another is called a context switch. A thread context switch involves pushing all thread CPU registers and program counter to thread's private stack and saving the stack pointer. After storing all the important data, another thread will be executed from the point it was suspended the last time. Until the thread finishes or gets suspended, instruction will be executed. When the thread suspends or finishes, a context switch will be triggered again, and another thread will be activated. Which thread will be activated depends how the scheduling is implemented in the program.

1.6. COROUTINES AND THREADS

The place of coroutines within the threading infrastructure is within a kernel thread. This means that coroutines are not visible to the OS. The OS only sees the kernel thread which contains the coroutines. This means that if a coroutine blocks execution, it will block the current thread.

In order to suspend and reinvoke a coroutine, a context switch is necessary. A context switch is the process of storing the state of a thread, so that the thread can be continued from the same point later. Some context switches will involve the kernel, but in the context of coroutines the kernel is not involved. This is one of the advantages of coroutines; a cheap context switch.

2. COROUTINES ABSTRACT

2.1. GENERAL

Coroutines are generalizations of functions. A normal function can be called, and arguments can be passed through the parameter list in the definition of the function. These values can be used inside the function. After the whole body of the function is executed, the control will be given back to the caller. This implies that the caller's execution is suspended until the called function is finished.

The data of the functions are handled by the call stack. The call stack has a fixed size in memory. The data of the function are stored within this stack. The stack uses a First In First Out system.

A coroutine can be seen as a function that can be suspended and resumed later on, while maintaining its state after the suspension. It will save the values of its local data and know where it needs to continue.

2.2. STACKFUL VS STACKLESS

Coroutines can be divided into two different sorts: stackful and stackless. The difference lies in the way the coroutine frame data is stored and handled by the program. This also gives some implications to the use of coroutines.

2.2.1. STACKFUL

Stackful coroutines, may also be called green threads, fibers or goroutines. Stackful coroutines are similar to kernel threads in that they can be started, suspended, resumed and finish, but they are not pre-emptive. Pre-emptive threads are scheduled by the Operating System, this means that the application gives control to the OS for the thread scheduling. A thread can give control to another thread anytime if the OS decides it. Stackful coroutines are contained within a kernel thread, this means that if the kernel thread is scheduled, the coroutine that is contained in that kernel thread will also be executed if the code of the coroutine is reached.

Stackful coroutines have a separate stack that can be used to process subroutine calls. When a regular function calls a coroutine, a separate stack together with the activation frame gets allocated. The activation frame holds the data that is necessary for the context switch, such as stack pointer, program counter and link register. After the allocation, the coroutine may be suspended or executed immediately, that depends on the implementation. Either way, the control is given back to the function that is on the call stack and execution continues the call stack.

At a later point, a function in the call stack may resume the coroutine. When this call happens, the data in the activation frame are pushed into the registers of the processors which trigger a context switch. When executing

the coroutine, the coroutine may call other regular functions which will be pushed into the coroutine stack. At a certain point, on of these regular functions may suspend the coroutine. The activation frame data gets updated and the processor's registers are set to trigger a context switch back to the call stack.

When the coroutine ends, all the stacks that are active in the coroutine stacks will be unwinded and the return will happen from the bottom function. After the return finishes, the stack will be empty and the context switch will be triggered. The coroutine cannot be resumed anymore.

An example is given in *Figure 2,* the thread and the fiber(= coroutine) have their own separate stack, the green numbers are the ordering number in which actions happen.

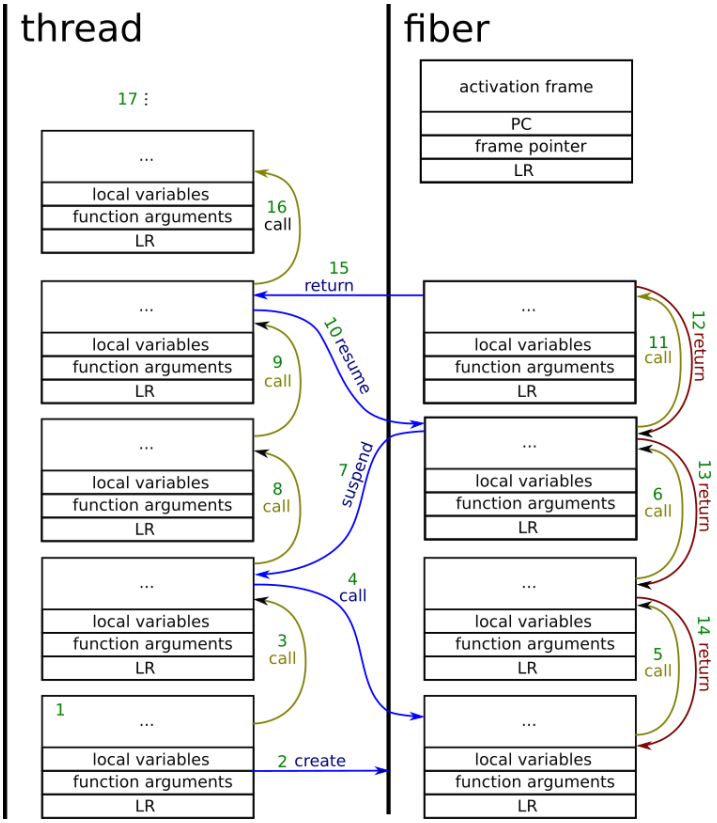


Figure 2: Example of Stackful Coroutine (fiber) [2]

2.2.2. STACKLESS

Stackless coroutines still have the main characteristics of stackful coroutines: they can be called, suspended, resumed and destroyed. But, stackless coroutines have a fixed relation with their callers. That is to say, the caller transfers control to the coroutine, and when the called coroutine yields, it will transfer execution back to it's caller.

A stackless coroutine can suspend itself only from the top-level function. The regular functions that are called in the coroutine are pushed onto the call stack. Stackless coroutines don't have their own separate stack, they only have dynamically allocated memory to store the essential data needed for suspension and resumption of the coroutine.

When a coroutine is created, the activation frame is allocated somewhere in the heap. Then it can be suspended or resumed immediately. That depends on the implementation of the coroutine. When the coroutine is resumed the first time, the body of the function will be allocated in the call stack, this means that it is as if a regular function is called.

When a regular function is called in the coroutine, that function will be pushed into the call stack. This implies that this function needs to return before the top-level function can yield to its caller.

If a coroutine suspends, all the necessary data that need to be preserved across the coroutine calls will be put into the activation frame. When the coroutine is resumed later, it will be as if a regular function is called. But the code will jump to the previous suspension pointer, which is saved on the heap. The values of variables will be restored from the activation frame.

After the coroutines returns, the coroutine won't be resumable anymore. An example is given below in *Figure 3*.

This all implies that stackless coroutines don't need to save as much data as the stackful coroutines do. But the downside is that the coroutine can suspend itself only from the top-level function.
A coroutine can also call other coroutines, which will allocate an activation frame on the heap for every coroutine that is created. A handle to the coroutine in the top level coroutine can be preserved in the activation frame of the top level coroutine. This way stackless coroutine can be chained.

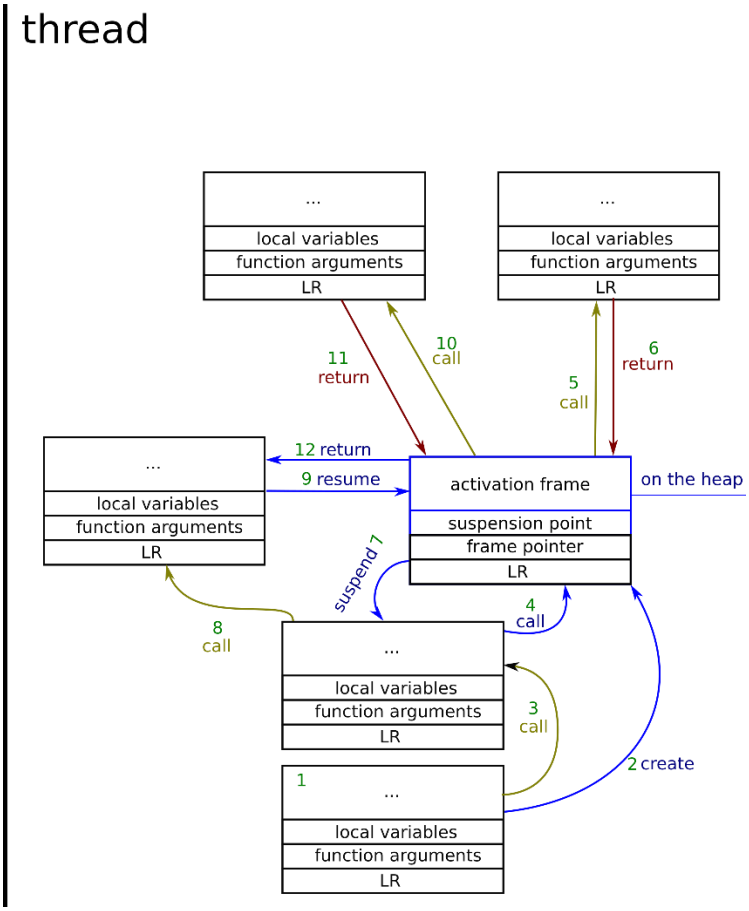


Figure 3: Example of Stackless Coroutine [2]

2.2.3. CONCLUSION

Stackful coroutines are more powerful than stackless coroutines in what they can achieve, like the nested function suspension. But stackful coroutines may have memory overhead, certainly if the implementation does not allow for a custom size of the stack.

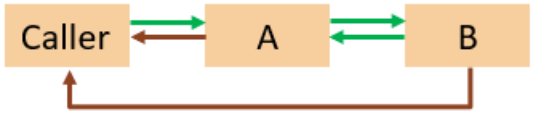
Stackless coroutines are more performant but have the disadvantage of not being able to suspend within a nested function. Although, a nested suspension can be simulated with the trampoline approach, it is not as convenient as the stackful coroutines' nested suspension.

Therefore, deciding which sort of coroutine to use depends on the problem which needs to be solved.

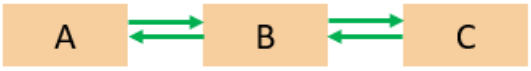
2.3 SYMMETRIC VS ASYMMETRIC

A coroutine can pass control in 2 ways: symmetric and asymmetric way.

In the symmetric way, a coroutine passes control by explicitly invoking another coroutine. For example: If coroutine **A** yields to coroutine **B**, the caller of **A** becomes the caller of **B.** If **B** ever does a normal yield, the control is given back to the caller of **A.** **B** can also of course give control back to **A**.



In the asymmetric way, a coroutine can invoke another coroutine, which will create a parent-child relation with the invoked coroutine. The other way an asymmetric coroutine can pass control is by passing control to its caller. The caller-callee relation between a coroutine's context and caller's context is fixed. The control flow must go from the caller context to the coroutine and again back to the caller. This implies that a coroutine can only yield directly to its caller.
An example of chained asymmetric coroutines: **A** yields to **B** yields to **C** yields to **B** yields to **A**



1. UNITY COROUTINE

1.1. ENGINE CONTROLLED COROUTINE

In Unity, a coroutine is created by creating a function with a return type of *IEnumerator* and with a yield statement included within the body of the function. When the coroutine reaches a yield statement, the control is given back to the engine and is continued the next frame. The coroutines are resumed between the *Update()* and *LateUpdate().* Coroutines being resumed every frame is the default way of handling the coroutines, but it can be customized by implementing a *CustomYieldInstruction.*

Behind the scenes, *YieldInstruction* is also a coroutine. By chaining coroutines, you can suspend a coroutine within a nested coroutines. Behind the scenes, every coroutine is yielding to its caller coroutine, until the top coroutine is

reached. This way, a stackful coroutine being suspended within a nested function can be imitated by using stackless coroutines. This mechanism is also called the *trampoline approach*.

A coroutine is destroyed when the Monobehaviour is destroyed. But, a coroutine may be destroyed manually by calling the *Monobehaviour.StopCoroutine* or *Monobehaviour.StopAllCoroutines*.

1.2. YIELD KEYWORD

The yield keyword is a language feature of C#. When a *yield* statement is used in a method, that method becomes an iterator. This is beneficial, because otherwise a separate class would be necessary to keep track of the iterator state when *IEnumerable* and *IEnumerator* are implemented.

1.3. STACKFULNESS AND SYMMETRY

The coroutines used in Unity are stackless and asymmetric coroutines, because the keyword *yield* can only give control back to the caller of the current coroutine. But, more complicated coroutines can be achieved by the previously discussed *trampoline* approach. With *CustomYieldInstruction*, a coroutine may wait for some certain time, may wait for an event to happen etc...

1.4. CONCLUSION

The coroutines in Unity are great to use in a game development context. In games, it is essential to process a certain task across many frames or wait for a certain event to happen. Coroutines are a proper and convenient solution to use in Unity for these problems.

2. C++ TS COROUTINE

2.1. GENERAL

C++ TS (TS = Technical Specification) Coroutines are stackless and asymmetric and introduce several keywords together with a library interface which needs to be used together with the keywords. When a function is created with a keyword used in it, the compiler will split the function in several function calls. The body of those functions can be customized, but the there is an order of call for all those functions.

The keywords can be seen as low-level assembly language. The new facilities that are introduced can be difficult to use directly in a safe way and are mainly intended to be used by library writers to build higher level abstraction that application developers can work with safely. The context switch is taken care of by the language itself, but it is up to the programmer to create different kind of coroutines by using the introduced API and keywords.

The three key words provided by the TS are: co_yield, co_return and co_await. Together with the keyword, several new types are provided: coroutine_handle<P>, coroutine_traits<Ts...>, suspend_always and suspend_never.

2.2. INTERFACES

C++ TS specifies a general mechanism for library code to customise the behaviour of the coroutine by implementing types that conform to a specific interface. The compiler then generates code that calls methods on instances of types provided by the library.

There are two interfaces that are defined by the C++TS: the *Promise* interface and the *Awaiter* interface.

The *Promise* interface customizes the behaviour of the *co_return* and *co_yield* expression within the coroutine. The programmer can customise how the coroutine behaves by implementing the Promise interface. When the coroutine is called, the promise type will handle how it is called. The promise type also handles how the coroutine returns .

The *Awaitable* interface is implemented to customize the behaviours of *co_await* expression within a coroutine. When a value is *co_awaited*, the code will be translated into a series of calls to methods on the awaitable object.

When *co_await* keyword is encountered in the coroutine, the code will be split into a series of calls to methods on the awaitable object. This awaitable object can specify whether to suspend the current coroutine, execute some logic after it has suspended to schedule the coroutine for later resumption, and execute some logic after the coroutine resumes to produce the results of the *co_await* expression.

2.3. KEYWORDS

To create a coroutine, a function needs to be created which returns a class that implements a promise type and uses one of the three keywords. The promise type certainly needs to implement *get_return_object(), initial_suspend(), final_suspend().* When a coroutine is created, the promise type object is created first. This will call the constructor of the promise type. After the construction of *promise_type*, its functions will be called. The *initial_suspend()* will be called first wherein the coroutine may be suspended or resumed immediately. There are two types that are provided for easy suspension and resumption: *suspend_always* and *suspend_never*.

After *initial_suspend()*, *get_return_object()* is called. This function will create the class that includes the inner class *promise_type*. That class can be created by passing itself in the constructor of the class. This way, a class will be created with a handle to the promise type. This handle will be used to access values.

If the function reaches a ***co_yield*** statement, the *yield_value()* method of the *promise_type* will be called. Here for example, the value may be changed, or other logic may be executed.

When a ***co_return*** statement is reached, the *return_value()* of *promise_type()* needs to be implemented. If no *co_return* is placed in the function, the *return_void()* needs to be implemented.

After the last *co_yield* / *co_await* or *co_return* is reached, the *final_suspend()* will be called after the *return_value()/return_void()*. Here can be chosen if the coroutine will be suspended or not and additional logic can be executed. When the coroutine is resumed after *final_suspend()*, the coroutine will be invalid and the *Done()* function of the *promise_type* will return false.

The class needs to implement *await_ready(), await_suspend()* and *await_resume()* if the coroutine is used in a ***co_await*** statement. The statement should look like "*co_await<expression>*". When this statement is reached

within the function, the program will jump to the *await_ready()* function of the object returned from the *expression*. This implies that the type returned by *expression* needs to implement the *co_await* function, if not the compiler will create an error.

After the *await_ready(),* the code will jump to *await_suspend()* if the return value of *await_read()* is false. In the *await_suspend(),* the programmer can decide what to do: suspend the awaiter, resume the object returned by *expression*, etc...
If *true*, the code will jump to *await_resume()* function, and the programmer can implement the function as he likes.

2.4. CONCLUSION

The C++ TS Coroutines is a powerful tool to create many different sorts of coroutines. The interfaces can be implemented in many ways, because every keyword can be implemented in a different way, conform to the API. But this also leads to a more complicated code and inconvenience of use, because for every keyword, functions of the API need to be implemented. Hence. this TS is meant to be used by library creators, who should create higher level coroutine libraries. These higher level coroutine libraries can be used by application builders with easier use.

3. BOOST COROUTINE

3.1. GENERAL

Another C++ implementation of coroutines is from Boost, which implements stackful coroutines. The library that implements the coroutines is *Boost.Coroutines2.* This library is based on Boost.Context, which uses the implementation of C++ proposal *P0876R0: fibers without a scheduler* and implementation of C++ proposal *P0534R3: call/cc*. Boost.Context uses assembly language to make context switch happen.

3.2. BOOST.CONTEXT

Boost.Context has 2 ways of context switch: fibers and call/cc. For fibers, the library provides a class *fiber*. For Call/cc, the library provides a class *continuation.*

3.2.1. CONTEXT SWITCH WITH FIBERS

The *fiber* class implements fibers without scheduler. The values of the registers at certain point of time represent a fiber, flow of execution. Different fibers can be executed synchronously by saving and restoring those register values. Context switch happens when the current fiber's register values are saved and the register values of the fiber that is going to be executed are restored. The current fiber will be suspended and the fiber that is activated will be resumed.

To create a fiber class, it needs a *context-function* with the signature *fiber (fiber && f)*. The parameter f represents the current fiber from which this fiber was resumed. When the current fiber wants to return, the *context-function* has to specify the fiber to which the control is transferred after termination of the current fiber.

A fiber can be continued by *resume()* and *resume_with().*The fiber class is a one-shot fiber, it means that it can be only used once. After calling *fiber.resume()* or *fiber.resume_with()* it is invalidated.

Because a fiber is a first-class object, it can be assigned to a variable, stored in a container, passed to a function and return by a function.

3.2.2. CONTEXT SWITCH CALL/CC

Another way the Boost.Context library switches context is by *call/cc (call with current continuation).* This concept explains a facility that permits a program to delegate processing to distinct lightweight execution agents nested within a thread. This agent may also be called context or execution context.

A *callcc()* captures the current continuation, which is the code that needs to be executed after *call/cc(),* and triggers a context switch. Like with fibers, the context switch is achieved by saving the register values of the current continuation and restoring the register values of the continuation which is going to be resumed.

A *callcc()* expects a *context-function* with the signature *continuation(continuation && c)* . The parameter c represents the current continuation from which this continuation was resumed. When the current continuation wants to return, the *context-function* has to specify the continuation to which the control is transferred after termination of the current continuation.

A continuation can be continued by *resume()* and *resume_with().*The continuation class is a one-shot continuation, it means that it can be only used once. After calling*.resume()* or *resume_with()* it is invalidated.

Because a continuation is a first-class object, it can be assigned to a variable, stored in a container, passed to a function and return by a function.

3.3. BOOST.COROUTINE2

Boost.Coroutine2 implements stackful and asymmetric coroutines. The Boost.Coroutine2 library has 2 important classes: coroutine<>::pull_type and coroutine<>::push_type. Both types can be used to create a coroutine, but both types are complementing each other as a coroutine. The other type is created together with the created type, which will be used as a coroutine.

The pull type pulls data from the coroutine function. Coroutines in the Boost.Coroutine2 use template type parameters to define the type of the value to transfer. The constructor of a pull type takes a function accepting a reference to a push type. This push type is used to give control back to the caller-context and transfer the value of type defined with the template parameter type. The control is passed to the push type by using *coroutine<>::pull_type::operator()*. This operator only does context switch, no value is passed to the coroutine function. The pull type provides a *.get()* function to retrieve the value pushed by the push type.

The push type pushes data to the coroutine function. Again, the template type parameter defines the type of the value that is transferred. The constructor of the push type takes a function accepting a reference to a pull type. This pull type is used to give control back to the main-context, but no value is transferred when this pull type is called inside the coroutine function.
Control is passed to the main-context by using *coroutine<>::push_type::operator()*.

3.4. CONCLUSION

Boost.Coroutine2 is a library that implements a coroutine interface. The fiber class is hidden behind this coroutine interface. The push type and pull type are easy to use and allow inversion of control.

4. WINDOWS FIBER

4.1. GENERAL

Windows Fibers implements fibers, which are user-level threads with their own stack. These fibers need to be scheduled by the application itself. Fibers share the same 'thread local storage' of the thread on which they are being executed. The state information of a fiber that is maintained consists of its stack, a subset of its registers, and the fiber data provided during fiber creation.

Because fibers aren't pre-emptive, the application itself need to schedule the fibers. A fiber is scheduled by switching to another fiber from the current active fiber. If a fiber is called from a context which is not a fiber yet, like the main function in C++, a function *ConvertThreadToFiber* needs to be called to create a fiber wherein the state information can be stored of the current execution.

A new fiber is created from another active fiber. At the creation of a fiber; the stack size, the starting address and the fiber data are passed. The starting address will be typically a user-supplied function. When this fiber function returns, it will terminate the thread on which it is being executed.

4.2. FIBER LOCAL STORAGE

In this library, a fiber can have a *Fiber Local Storage* to create a unique copy of a variable for each fiber. A fiber can access local variables of functions and global and static variables. But with *Fiber Local Storage,* you can also store values that are unique to a fiber.

The data of a fiber can be deleted by calling the *DeleteFiber* function. The deleted data includes the stack, subset of the registers and the fiber data.

4.3. USAGE

To use the Windows Fibers API, the header file *windows.h* needs to be included.

The way the Windows Fibers work is different then the fiber that boost.context uses. In Windows API, if the fiber function finishes and return before it switches to the main fiber, the main thread on which the fiber is being executed will return.

To make the first fiber switch, the current thread needs to be converted to a fiber, because only a fiber can switch to another fiber and a fiber can only switch to another fiber. There should be a fiber associated with the thread. This can be achieved by using *ConvertThreadToFiber(LPVOID lpParameter).* A pointer to a fiber variable can be passed. This data is accessible through *lpParameter* or using the function *GetFiberData().*

To create a fiber, the *LPVOID CreateFiber(SIZE_T dwStackSize, LPFIBER_START_ROUTINE lpStartAddress, LPVOID lpParameter)* can be used. The *dwStackSize* denotes the size in bytes of the created fiber. The *lpStartAddress* is a pointer to the application defined function to be executed by the fiber and represents the starting address of the fiber. The *lpParameter* is a pointer to the variable passed to the fiber. This pointer can be accessed in the application defined function, because this pointer is passed as an argument to the application defined function.

A fiber can switch to another fiber by using *SwitchToFiber(LPVOID lpFiber)*, with *lpFiber* the address of the fiber to be scheduled. When this function is called, the state information of the current fiber is saved and the state information of the fiber to be scheduled will be restored.

A fiber can be deleted by using *DeleteFiber(LPVOID lpFiber),* with *lpFiber* the address of the fiber to be deleted. When this function is called, all the data associated with the fiber will be deleted, including the stack, subset of the register and the fiber data.

4.4. CONCLUSION

Windows Fibers can be used to do what coroutines do, but it is still a fiber library. This means that if a programmer wants to create a coroutine, using Windows Fiber will not be the best choice if ease of use is considered. But this library can be used to create a coroutine library which has a better ease of use. Windows Fibers is a good choice if it needs to be used for the context switch of stackful coroutines.

CASE STUDY

1. INTRODUCTION

The implementation of a custom coroutine started by deciding what sort of coroutine to implement. It was decided to implement asymmetric stackful coroutines, because stackful coroutines are powerful and can solve recursive problems.
The next step was choosing the library which would be used for context switch. It was decided to use Boost.Context, which implements a fiber class, to use for the context switch.

The coroutine interface will be build on top of the Boost.Context (see figure 4). The interface is meant to be an easy to use interface with clear class names. Boost.Context can be used to do what coroutines do, but the programmer needs to handle fiber specific tasks, and that is not the purpose of a coroutine interface. A coroutine interface should hide the fiber specific functionalities and create a coroutine specific interface with easy to understand naming conventions.

The custom coroutine will also have extra functionalities. The coroutine class will be a template class, where the template type parameter will determine the type of the value that gets updated in the coroutine class.
Arguments may be passed to the coroutine function and the values may be reset together with the coroutine later.

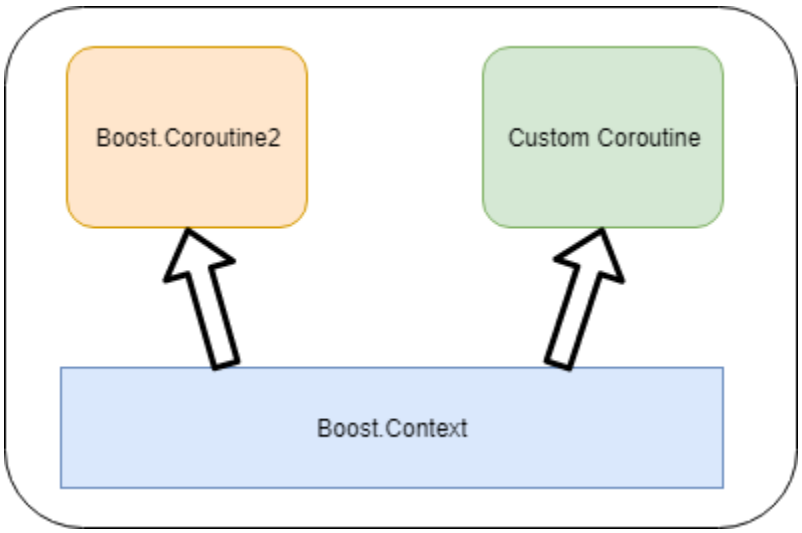


Figure 4: Boost.Context and Custom Coroutine

2. INITIALIZATION

The coroutine class is a template class which's type parameters denote the type of the value which is updated by the coroutine and the types of the optional parameter if arguments are passed to the constructor of the coroutine. The constructor accepts a lambda function which needs to at least accept a reference to a Caller object in its parameter list.

The coroutine class accepts parameters in two ways: passing arguments to the coroutine class constructor or passing the variables to the lambda capture list. Figure 5.1 and Figure 5.2 demonstrates an example where arguments are passed via the constructor parameters.

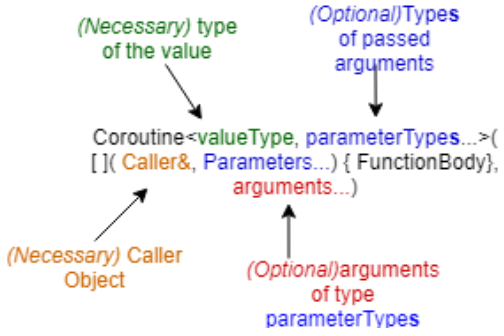


Figure 5.1: Coroutine Initialization

```
Coroutine<float, float, float>(
[ ]( Caller&, float minValue, float maxValue) {
    FunctionBody},
    10.0f, 100.0f)
```

Figure 5.2: Coroutine Initialization Example

Figure 6.1 and 6.2 demonstrates an example where arguments are passed via the capture list of the lambda. In figure 6.1 and 6.2, the initialization looks cleaner and more readable. But for the lambda to capture variables, variables need to be created outside the coroutine. Passing argument via the constructor is therefore another

option for the programmer to initialize the coroutine. It will prevent extra code to be created outside the coroutine initialization.

```
Coroutine<valueType>(
[ variables...]( Caller&) { FunctionBody})
```

Figure 6.1: Coroutine Initialization

```
float minValue = 10.0f
float maxValue = 100.0f

Coroutine<float>(
[ minvalue, maxValue]( Caller&) {
FunctionBody})
```

Figure 6.2: Coroutine initialization Example

3. CONTEXT SWITCHING

The Boost.Context implements a *fiber* class which represents a fiber (see 5.2.1). But this fiber class uses a *resume* function which only accepts rvalue fiber, hence the current fiber needs to be converted to an rvalue and assigned to the current fiber. The first step was to create a wrapper template class called *Coroutine<>*, which will hide all these fiber functionalities inside the class. This class will contain a fiber data member. The template type name is used to define the type of the return value of the coroutine.
This way the resume statement will be a lot shorter.
For example, `coro = std::move(coro).resume()` will become `coro.resume()`, which is just a simple statement that makes it clear the coroutine will be resumed.

At the creation of a *Coroutine* class, a *Caller* class will be passed as a reference to the coroutine function. This *Caller* class represents the main context from which the created coroutine is called. To context switch back to the main context from within the coroutine function, a simple *Caller::operator()* needs to be called. This is much cleaner and clearer than the fiber class method.
For example, `sink = std::move(sink).resume()` becomes `c()`.

4. SUSPEND/YIELD

Suspend is the context switch from within the coroutine function to the caller context. This suspend will be available by the *Caller* object that is passed as a reference to the coroutine function. The *Caller* class provides a *Caller::operator()* member function that will switch to the main context.

There are 2 versions of this function; one without a parameter list and one with a parameter list. When the one without the parameter list is used, only a context switch will happen. If the one with the value is used, the value with the type, determined by the template type parameter, will be set to the value passed to this function.

This value can be accessed later on by the caller context by using the *Coroutine<>::Get()* member function.

5. RESET

When the coroutine is finished, *Resume()* won't work anymore on the coroutine. To use the coroutine again, the *Reset()* function can be used to reset the coroutine. If variables are passed via the construction of the coroutine, new argument may be passed to the coroutine. The coroutine will start with new values.

For example; a coroutine generator is created. Two float values are passed as minimum value and max value. The coroutine generator will generate numbers from minimum value up until maximum value. When this coroutine is finished, the programmer can use this coroutine again, but it needs to be reset first. When resetting, new minimum value and maximum values may be passed.

PROFILE

1. INTRODUCTION

To compare the different libraries in terms of performance and memory, 2 test cases, generator and same fringe problem, were implemented with each of those libraries. The tested libraries are Boost, Windows Fibers, C++ TS Coroutines and the custom coroutine implementation.

The generator yields a value every time it is called until it reaches 1.000.000.
The same fringe problem checks all the leaves of a small binary tree from left to right, and compares it with another binary tree. Until one of the leaves differs from the other binary tree, the coroutine will continue executing.

The same fringe problem in C++ TS Coroutines doesn't have results in the tables because same fringe problem is a problem that can be solved with a stackful coroutine. But it is possible to simulate stackful coroutine by trampoline approach. Basically, it is a chain of coroutines calling other coroutines. The problem was a lack of time, because learning to use C++ TS Coroutine was a challenge on itself. Therefore, implementing the same fringe problem in C++ TS was omitted.

The tests were done on a pc with the following specs: Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz 2.50GHz and 16.0 GB RAM, running 64-bit Windows, x64 based processor.

2. SPEED

Peformance(x86)	Boost	Windows Fibers	C++ TS Coroutines	Custom Coroutine
Generator Debug(milliseconds)	1699,5	52,5	475,3	1407,7
Generator Release(milliseconds)	65,3	47,2	4,9	21,1
Same Fringe Debug(microseconds)	2944,3	135,1		490,8
Same Fringe Release(microseconds)	86,2	91,2		38

Peformance(x64)	Boost	Windows Fibers	C++ TS Coroutines	Custom Coroutine
Generator Debug(milliseconds)	1710,7	95,3	434,4	1608,5
Generator Release(milliseconds)	52,6	84	4,5	40,8
Same Fringe Debug(microseconds)	288,7	101		535,5
Same Fringe Release(microseconds)	143,9	81,7		25

From the results above, it is obvious that the stackless coroutine (C++ TS) perform better in respect of speed. Besides that, the custom coroutine achieves a better performance than the boost coroutines and the Windows Fiber.

The numbers above are useful to compare against each other, but in the context of game development, it is important to reach 60 frames per second. So, what do these numbers mean in the context of game performance?

A context switch certainly has an overhead, because the CPU needs to do extra work. But a single context switch's overhead is minimal and has no meaning in the context of disturbing the performance of the game. We are talking about tens of nanosecond (= 10^-6 milliseconds), according to boost performance tests [9].

But, the overhead of a context switch will be bigger when a higher level library is built, like a coroutine library, because the library class' function are also called. Coroutines will have an impact on the performance if there are significant amount of context switches happening in single frame.

The results below are the results of testing the custom coroutine generator we used for the results above, but with lesser number of calls, in release x64 mode. Every call includes 2 context switches: one to the context of the coroutine and then one back to the caller context.

Number of Calls	performance(microseconds)
1	12,9
10	15,6
100	17,6
1000	54,2
10000	595,5
100000	3851,8

A game that is run at 60 frames per second has 16ms reserved for every frame. This means that if the number of calls is limited to 1000, it may consume up to 0.0542ms of the 16ms (= +-1/300 of a frame) which begins to have a meaning in respect of performance. The result below are the performance of implementing what a generator does, but without a coroutine. Until 1000, it doesn't even reach a microsecond. The compiler probably also optimizes the code.

Number of Calls	performance(microseconds)
1	< 1
10	< 1
100	< 1
1000	< 1
10000	3,5
100000	32,4

This means that theoretically, if we called a coroutines 100.000 times, it will be 100 times slower (in a simple test case), which is a lot slower. But practically, coroutines won't be called that many times in a single frame. This should also make it clear to the programmer that one shouldn't abuse the coroutines. If the number of calls to coroutines are limited to 1000, it is safe to say that using coroutines are a good option if it makes your code easier to write and read.

What happens if the size of the value, that is transferred at context switch, is bigger? The numbers below are results of a custom coroutine which's value is of type vector<float>. When the coroutine is called the first time, the coroutine will create a vector of starting from 1 to 100000 elements. The coroutine is called 100000 times, and at each suspend, the vector is set to the vector created at the start of the coroutine. So the bigger the vector, the bigger the size that is being transferred.

size	Performance(millisecond)
1	11,2
10	11
100	15,7
1000	25,2
10000	349,8
100000	3655,3

3. MEMORY

The difference between a stackful and stackless coroutine becomes more obvious when it's about memory usage. From the results below, we can see that the boost and custom coroutine use 30 times more memory than the stackless coroutine. What's interesting is that Windows Fibers uses a lot less memory. This may be caused because the custom coroutine and boost coroutine have more classes and functionalities on top of a fiber class while Windows Fibers is only a fibers library.

Memory(x84)	Boost	Windows Fibers	C++ TS Coroutines	Custom Coroutine
Generator Release(kb)	128	16	4	136
Same Fringe Release(kb)	264	12		272

Memory(x64)	Boost	Windows Fibers	C++ TS Coroutines	Custom Coroutine
Generator Release(kb)	52,6	24	16	144
Same Fringe Release(kb)	268	16		256

CONCLUSION

Coroutines are a good way to execute a task across different calls (or frames in the context of game development), because it keeps all the logic inside the coroutine without cluttering the outside of the coroutine.

When writing code, the way the code is organized should make sense to the programmer and using coroutines makes sense in certain situations.

The research shows that stackful coroutines are easier to use if suspension is desired within a nested function, and that's why it's decided to implement a stackful coroutine in C++.

Because implementing a context switch is not a simple task, an existent library is used to take care of the context switch and a stackful coroutine is built upon the Boost.Context's fiber class. The interface built upon the fiber class has an easy to use and understand interface and has the functionalities corresponding to the concept of coroutines.

REFERENCES

[1] Threads: multiple flows of execution within a process, by Paul Krzyzanowski
https://www.cs.rutgers.edu/~pxk/416/notes/05-threads.html

[2] Coroutines, stackless vs stackful, by Dawid Pilarski
https://blog.panicsoftware.com/coroutines-introduction/

[3]yield c# reference,
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/yield

[4] Unity Coroutines, Manual
https://docs.unity3d.com/Manual/Coroutines.html

[5] Windows Fibers
https://docs.microsoft.com/en-us/windows/desktop/procthread/fibers

[6] Boost.Coroutine, symmetric coroutines
http://www.crystalclearsoftware.com/soc/coroutine/coroutine/symmetric_coroutines.html

[7] C++ Coroutines: Understanding operator co_await, by Lewiss Baker
https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await

[8] C++ Coroutines: Understanding the promise type, by Lewiss Baker
https://lewissbaker.github.io/2018/09/05/understanding-the-promise-type

[9] Boost.Context, context switch performance
https://www.boost.org/doc/libs/1_70_0/libs/context/doc/html/context/performance.html

[10] What is stack, execution context, stackoverflow
https://stackoverflow.com/questions/7721200/using-javascript-closures-in-settimeout/7722057#7722057

[11] Lightweight cooperative multitasking with boost.context, by Tolon
https://www.tolon.co.uk/2012/08/boost-context/

[12] fibers without schedulers, paper by Oliver Kowalke
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0876r0.pdf

[13] Distinguishing coroutines and fibers
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4024.pdf

APPENDICES