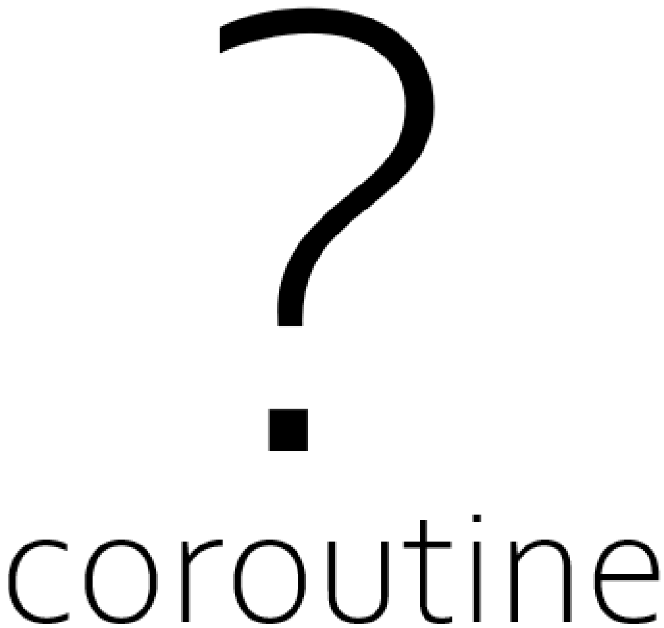
Coroutine in C Language



Reading Time: 8 minutes

It's been quite a while that I haven't published anything on my blog. But that's due to the job change. I hope you understand that it has never been easy to re-settle in a new environment with new people while maintaining a steep technical learning curve. It takes time to tune yourself accordingly. Anyways, I wrote on "Coroutine in C Language" as a pre-pend to my upcoming post on C++20 Coroutine. Today we will see "How Coroutine Works Internally?".

Contents [hide]

Prologue

If you are an absolute beginner, then go through the below pre-requisites. And if you are not a beginner, you better know what to skip!

Note: Context switching APIs `getcontext`, `setcontext`, `makecontext` and `swapcontext` were obsoleted in POSIX.1-2004 and removed in POSIX.1-2008 citing portability issues. So, please do not use it. Here I have used it for demonstration purpose.

Coroutine Basics

What Is Coroutine?

- A coroutine is a function/sub-routine(co-operative sub-routine to be precise) that can be suspended and resumed.
- In other words, You can think of coroutine as an in-between solution of normal function & thread. Because, once function/sub-routine called, it

executes till the end. On other hand, a thread can be blocked by synchronization primitives(like mutex, semaphores, etc) or suspended by an OS scheduler. But again you can not decide on suspension & resumption on it. As it is done by the OS scheduler.

- While coroutine on other hand, can be suspended on a pre-defined point & resumed later on a need basis by the programmer. So here programmer will be having complete control of execution flow. That too with minimal overhead as compared to thread.
- A coroutine is also known as native threads, fibres(in windows), lightweight threads, green threads(in java), etc.

Why Do We Need Coroutine?

- As I usually do, before learning anything new, you should be asking this question to yourself. But, let me answer it:
- Coroutines can provide a very high level of concurrency with very little overhead. As it doesn't need OS intervention in scheduling. While in a threaded environment, you have to bear the OS scheduling overhead.
- A coroutine can suspend on a pre-determined point, so you can also avoid locking on shared data structures. Because you would never tell your code to switch to another coroutine in the middle of a critical section.
- With the threads, each thread needs its own stack with thread local storage & other things. So your memory usage grows linearly with the number of threads you have. While with co-routines, the number of routines you have doesn't have a direct relationship with your memory usage.
- For most use cases coroutine is a more optimal choice as it is faster as compared to thread.
- And if you are still not convinced then wait for my C++20 Coroutine post.

To-the-point Context Switching API Theory

- Before we dive into a implementation of Coroutine in C, we need to understand the below foundation functions/APIs for context switching. Off-course, as we do, with less to-the-point theory & with more code examples.

  1. `setcontext`
  2. `getcontext`
  3. `makecontext`
  4. `swapcontext`

- If you are already familiar with `setjmp/longjmp`, then you might have ease in understanding these functions. You can consider these functions as an advanced version of `setjmp/longjmp`.
- The only difference is `setjmp/longjmp` allows only a single non-local jump up the stack. Whereas, these APIs allows the creation of multiple cooperative threads of control, each with its own stack or entry point.

Data Strucutre To Store Execution Context

- `ucontext_t` type structure that defined as below is used to store the execution context.
- All four(`setcontext`, `getcontext`, `makecontext` & `swapcontext`) control flow functions operates on this structure.

```
typedef struct {
    ucontext_t *uc_link;
    stack_t     uc_stack;
    mcontext_t  uc_mcontext;
    sigset_t    uc_sigmask;
    ...
} ucontext_t;
```

- `uc_link` points to the context which will be resumed when the current context exits, if the context was created with `makecontext` (a secondary context).
- uc_stack is the stack used by the context.
- uc_mcontext stores execution state, including all registers and CPU flags, frame/base pointer(i.e. indicates current execution frame), instruction pointer(i.e. program counter), link register(i.e. stores return address) and the stack pointer(i.e. indicates current stack limit or end of current frame). `mcontext_t` is an opaque type.
- `uc_sigmask` is used to store the set of signals blocked in the context. Which isn't the focus for today.

int setcontext(const ucontext_t *ucp)

- This function transfers control to the context in `ucp`. Execution continues from the point at which the context was stored in `ucp`. `setcontext` does not return.

int getcontext(ucontext_t *ucp)

- Saves current context into `ucp`. This function returns in two possible cases:

  1. after the initial call,
  2. or when a thread switches to the context in `ucp` via `setcontext` or `swapcontext`.

- The `getcontext` function does not provide a return value to distinguish the cases (its return value is used solely to signal error), so the programmer must use an explicit flag variable, which must not be a register variable and must be declared `volatile` to avoid constant propagation or other compiler optimisations.

void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...)

- The `makecontext` function sets up an alternate thread of control in `ucp` , which has previously been initialised using `getcontext`.
- The `ucp.uc_stack` member should be pointed to an appropriately sized stack; the constant `SIGSTKSZ` or `MINSIGSTKSZ` is commonly used.
- When `ucp` is jumped to using `setcontext` or `swapcontext`, execution will begin at the entry point to the function pointed to by `func`, with `argc` arguments as specified. When `func` terminates, control is returned to the context specified in `ucp.uc_link`.

int swapcontext(ucontext_t *oucp, ucontext_t *ucp)

- Saves the current execution state into `oucp` and then transfers the execution control to `ucp`.

[Example 1]: Understanding Context Switching With `setcontext` & `getcontext` Functions

- Now, that we have read lot of theory. Let’s create meaningful out of it.
- Consider below program that implements plain infinite loop printing “Hello world” every second.

```
#include <stdio.h>
#include <ucontext.h>
#include <unistd.h>
#include <stdlib.h>

int main( ) {
    ucontext_t ctx = {0};

    getcontext(&ctx);   // Loop start
    puts("Hello world");
    sleep(1);
    setcontext(&ctx);   // Loop end

    return EXIT_SUCCESS;
}
```

- Here, `getcontext` is returning with both possible cases as we have mentioned earlier i.e.:

  1. after the initial call,
  2. when a thread switches to the context via `setcontext`.

- Rest is I think self-explanatory.

[Example 2]: Understanding Control Flow With `makecontext` & `swapcontext` Functions

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <signal.h>
#include <ucontext.h>

void assign(uint32_t *var, uint32_t val) {
    *var = val;
}

int main( ) {
    uint32_t var = 0;
    ucontext_t ctx = {0}, back = {0};

    getcontext(&ctx);

    ctx.uc_stack.ss_sp = calloc(1, MINSIGSTKSZ);
    ctx.uc_stack.ss_size = MINSIGSTKSZ;
    ctx.uc_stack.ss_flags = 0;

    ctx.uc_link = &back; // Will get back to main as `swapcontext` call will populate `back` with curren
    // ctx.uc_link = 0;  // Will exit directly after `swapcontext` call

    makecontext(&ctx, (void (*)())assign, 2, &var, 100);
    swapcontext(&back, &ctx);    // Calling `assign` by switching context

    printf("var = %d\n", var);
```

```
    return EXIT_SUCCESS;
}
```

- Here, the `makecontext` function sets up an alternate thread of control in `ctx`. And when jump made with `ctx` by using `swapcontext`, execution will begin at `assign`, with respective arguments as specified.
- When `assign` terminates, control will be switch to `ctx.uc_link`. Which points to `back` & will be populated by `swapcontext` before jump/context-switch.
- If the `ctx.uc_link` is made to 0, then current execution context is considered as the main context, and the thread will exit when `assign` context gets over.
- Before a call is made to `makecontext`, the application/developer needs to ensure that the context being modified has a pre-allocated stack. And `argc` matches the number of arguments of type `int` passed to `func`. Otherwise, the behavior is undefined.

Coroutine in C Language

- Initially, I have created single file to demonstrate the example. But then I realised It will be too much to stuff into the single file. Hence, I splited implementation & usage example into different file which will make the example more comprehensible & easy to understand.

Implementation of Coroutine in C

- So, here is the simplest coroutine in c language:

coroutine.h

```
#pragma once

#include <stdint.h>
#include <stdlib.h>
#include <ucontext.h>
#include <stdbool.h>

typedef struct coro_t_ coro_t;
typedef int (*coro_function_t)(coro_t *coro);

/*
    Coroutine handler
*/
struct coro_t_ {
    coro_function_t     function;           // Actual co-routine function
    ucontext_t          suspend_context;    // Stores context previous to coroutine jump
    ucontext_t          resume_context;     // Stores coroutine context
    int                 yield_value;        // Coroutine return/yield value
    bool                is_coro_finished;   // To indicate the current coroutine status
};

/*
    Coroutine APIs for users
*/
coro_t *coro_new(coro_function_t function);
int coro_resume(coro_t *coro);
void coro_yield(coro_t *coro, int value);
void coro_free(coro_t *coro);
```

- Just ignore the coroutine APIs as of now.
- The main thing to focus here is coroutine handler that has following field

- `function` : That holds the address of actual coroutine function supplied by user.
- `suspend_context` : That used to suspend the coroutine function.
- `resume_context` : That holds the context of actual coroutine function.
- `yield_value`: To store the return value between intermediate suspension point & also final return value.
- `is_coro_finished` : An indicator to check status on coroutine lifetime.

coroutine.c

```c
#include <signal.h>
#include "coroutine.h"

static void _coro_entry_point(coro_t *coro) {
    int return_value = coro->function(coro);
    coro->is_coro_finished = true;
    coro_yield(coro, return_value);
}

coro_t *coro_new(coro_function_t function) {
    coro_t *coro = calloc(1, sizeof(*coro));

    coro->is_coro_finished = false;
    coro->function = function;
    coro->resume_context.uc_stack.ss_sp = calloc(1, MINSIGSTKSZ);
    coro->resume_context.uc_stack.ss_size = MINSIGSTKSZ;
    coro->resume_context.uc_link = 0;

    getcontext(&coro->resume_context);
    makecontext(&coro->resume_context, (void (*)())_coro_entry_point, 1, coro);
    return coro;
}

int coro_resume(coro_t *coro) {
    if (coro->is_coro_finished) return -1;
    swapcontext(&coro->suspend_context, &coro->resume_context);
    return coro->yield_value;
}

void coro_yield(coro_t *coro, int value) {
    coro->yield_value = value;
    swapcontext(&coro->resume_context, &coro->suspend_context);
}

void coro_free(coro_t *coro) {
    free(coro->resume_context.uc_stack.ss_sp);
    free(coro);
}
```

- The most used APIs for coroutine is `coro_resume` & `coro_yield` that drags the actual work of suspension & resumption.
- If you already have consciously gone through the above Context Switching API Examples, then I don't think there is much to explain for `coro_resume` & `coro_yield`. It's just `coro_yield` jumps to `coro_resume` & vice-versa. Except for the first call to `coro_resume` which jumps to `_coro_entry_point`.
- coro_new function allocates memory for the handler as well as stack & then populates the handler members. Again `getcontext` & `makecontext` should be clear by this point. If not then please re-read the above section on Context Switching API Examples.
- If you genuinely understand the above coroutine API implementation, then the obvious question would be why do we even need `_coro_entry_point`? Why can't we directly jump to the actual coroutine function?.

- But then my argument will be “How do you ensure the lifetime of coroutine?”.
- Which technically means, number of call to coro_resume should be similar/valid to number of call to `coro_yield` plus one(for actual return).
- Otherwise, you can not keep track of yields. And behaviour will become undefined.

- Nonetheless, `_coro_entry_point` function is needed otherwise there is no way by which you can deduce the coroutine execution finished completely. And next/subsequent call to coro_resume is not valid anymore.

Coroutine Lifetime

- By the above implementation, **using the coroutine handler**, you should only be able to execute coroutine function completely once throughout program/application life.
- If you want to call the coroutine function again, then you need to create a new coroutine handler. And rest of the process will remain the same.

Coroutine Usage Example

coroutine_example.c

```
#include <stdio.h>
#include <assert.h>
#include "coroutine.h"

int hello_world(coro_t *coro) {
    puts("Hello");
    coro_yield(coro, 1);    // Suspension point that returns the value `1`
    puts("World");
    return 2;
}

int main() {
    coro_t *coro = coro_new(hello_world);
    assert(coro_resume(coro) == 1);     // Verifying return value
    assert(coro_resume(coro) == 2);     // Verifying return value
    assert(coro_resume(coro) == -1);    // Invalid call
    coro_free(coro);
    return EXIT_SUCCESS;
}
```

- Usecase is pretty straight forward:
  - First, you create a coroutine handler.
  - Then, you start/resume the actual coroutine function with the help of the same coroutine handler.
  - And, whenever your actual coroutine function encounters a call the `coro_yield`, it will suspend the execution & return the value passed in the 2nd argument of coro_yield.

- And when actual coroutine function execution finishes completely. The call to coro_resume will return -1 to indicate that the coroutine handler object is no more valid & the lifetime is expired.
- So, you see coro_resume is a wrapper to our coroutine `hello_world` which executes `hello_world` in parts(obviously by context switching).

Compiling

- I have tested this example in WSL with gcc 9.3.0 & glibc 2.31.

```
$ gcc -I./ coroutine_example.c coroutine.c  -o myapp && ./myapp
Hello
World
```

Parting Words

You see there is no magic if you understand “How CPU Executes The Code..!” well-given Glibc provided a rich set of context switching API. And, from the perspective of low-level developers, it’s merely a well-arranged & difficult to organize/maintain(if used raw) context switching function calls. My intention here was to put the foundation for C++20 Coroutine. Because I believe, if you see the code from CPU & compiler’s point of view, then everything becomes easy to reason about in C++. See you next time with my C++20 Coroutine post.