# Assignment #2.1

CSD 2180/2182 FALL 2023

| | |
|---|---|
| Due Date: | As specified on the moodle |
| Topics covered: | Threading, context switching |
| Deliverables: | To submit all relevant files that will implement the user-level thread library, including `new-coro-lib.cpp` that contains the implementation of the thread library. |
| Objectives: | To learn and understand context switching at some specific level using setcontext or inline assembly. The student who successfully completes this assignment will understand some analogy between the management of threads and the management of processes. |

## Programming Statement: A Cooperative Scheduled User-Level Thread Library

In this assignment, you will implement a cooperatively scheduled user-level thread library under Linux. The library to be implemented consists of a set of functions that allow management of threads in user space (i.e., the operating system is not aware of your threads). By way of introduction, we shall describe some of the key concepts here:

- Cooperatively scheduled – this means that each thread will be using the CPU until one of the following events happen.

    - The thread voluntarily gives up the CPU by calling `thd_yield()`.
    - The thread has finished its execution.
    - The thread has chosen to wait (or block) until a specific thread has completed execution.
    - The thread is being used as a co-routine, and **pushes** a value to the **puller** thread.

- User-level - The management of the threads are done without employing system calls, save the exception of calling `malloc` and `free` that may make use of the memory management facilities of the OS.

In this assignment, you will write functions that allow the creation, termination, suspension and simple scheduling of the threads. In order to complete this assignment, you need to accomplish the following:

- Export the user thread library API. This is already done in the provided header file that includes the function prototypes of the exported API.

- Implement any functions (including any auxillary functions and the exported API) to ensure that the library is running according to specifications. Please compare the behaviour of your library with the provided sample static library `coro-lib.a`. They should be exactly the same.

## The TCB data structure and Exported API

Each thread that is created will have its own TCB – Thread Control Block. The TCB is a data structure maintained *internally* by the library that contain the necessary information for the execution of each thread. Your thread library should be able to support an indefinite number of threads, as long as there is enough memory space for them. Note that each of the following functions has been implemented in the namespace `Coro`.

In the following, the application programming interface (API) of your library will include the following routines (The APIs are also given in the header files listed in the Appendix):

1. **Initialization of the thread library:**

   ```
   void thd_init();
   ```

   The `thd_init` function initializes all necessary data structures for use in the user-level thread library. All test cases will call this function before calling the other thread library API. You are required to figure out what is supposed to be initialized in this function. That's part of the assignment.

2. **Creation of a user-level thread:**

   ```
   thread_id_  new_thd(void *(*thd_function_t )(void *), void* param );
   ```

   Conceptually, calling this function creates a new thread that begins by calling function `thd_function_t` with arguments `param`. It should be noted that `thd_function_t` takes in a `void *` parameter and returns a `void *`. Additionally, `new_thd` returns a thread id uniquely identifies the new thread.

   Implementation-wise, this means that you may need to allocate the stack and the TCB for the new thread in this function. Please assume a working size of 1MB for each stack allocated for each thread.

   For the assignment of the thread ID, we assume that the primary thread (i.e., the original, uncreated thread of the process) has an ID of 0. The other threads are assigned in order of their creation e.g., the 4th created thread will have a thread ID of 4. Once the ID reaches maximum, you can wrap around and recycle it.

3. **Termination of a user-level thread:**

   ```
   void thread_exit(void *ret_value);
   int wait_thread(thread_id_ id, void **value);
   ```

   `thread_exit` is a function that exits the thread with a `ret_value` as a return value of the thread. There are two ways by which a thread may terminate:

   - By calling the `thread_exit` function.
   - By calling `return` in the thread starting function i.e., the function passed as a parameter into `new_thd`.

We need to ensure that all threads, except the primary thread, *must* call this function upon exit. Bad things may happen if this `thread_exit` is not called when a thread terminates. Therefore, we need to handle the case when the user choose to exit the thread via the latter way described above. To do this, we should ensure that the thread starting function is called by a wrapper function that will call `thread_exit` on behalf of the user. Finally, since thread termination is one of the ways the thread gives up its use of the CPU, you should reschedule another thread to run after thread termination.

`wait_thread` is a function that waits for a thread to be completed and obtain the return value of the thread. `id` identifies the thread to be waited upon and `value` should be changed to the return value of the thread after `wait_thread` successfully completes.

When a thread X terminates, there are two possibilities.

- First, there could an existing thread that have called `wait_thread` on thread X. Given that a waiting thread cannot be scheduled to run during the entire period of waiting, the library at this point should ensure that the waiting thread is schedulable and able to receive the return value of the thread properly.

- Second, there are no waiting threads. In this case, the library should ensure that the data structures for the exiting thread are not released in case some other threads call `wait_thread` at a later time. In this library, it is expected that there is only one `wait_thread` call for each created thread. The results of multiple `wait_thread` calls to the same thread is undefined.

Conversely, when thread Y calls `wait_thread` to wait for thread X, there are three possibilities.

- First, the thread to be waited for has already terminated. In this case, `wait_thread` should free up the data structures associated with X and obtain the return value of X.

- Second, X has not yet terminated. In this case, Y will be suspended until X completes and resumes execution *inside* `wait_thread`. Y should go on to free up the data structures associated with X and obtain the return value of X.

- Thirdly, the thread to be waited for is no longer a valid thread (i.e., the associated TCB does not exist anymore).

`wait_thread` should return `WAIT_SUCCESSFUL` in the former two cases and `NO_THREAD_FOUND` in the latter case.

4. **Yield the processor:**

```
void thd_yield()
```

This function causes the current thread to yield the CPU for another thread (if any) to be scheduled. Roughly, the logic of this function should be the following:

(a) Save the context of the current thread. You will need to determine what would be the "context" of a thread. An example of information contained in the TCB:

- Stack pointer: Points to thread's stack in the process
- Program counter
- State of the thread (running, ready, waiting, start, done)
- register values for thread

(b) Call the scheduler or perform scheduling task. More details on scheduling in the next section. Depending on the scheduler, another thread might run as a result.

(c) If execution reaches beyond the call to the scheduler, it means that we can simply return from the `thd_yield()` function.

## Push and Pull

```
void pull_value(void **pulled_value);
void push_value(void *pushed_value);
```

These functions are that which causes the user-thread library to imitate co-routine behaviours in C++. The functions assume that there are 2 individual threads existing in a `Puller-Pusher` relationship. The `Puller-Pusher` follows the following definition and constraints:

- Puller is the thread that created the Pusher.
- Pusher is the created thread.
- Puller and Pusher is strictly a 1:1 relationship.
- The user must ensure that the number of `pull_value` calls and the `push_value` calls must be the same i.e., 1 pull for 1 push.

The `pull_value` function is called by the Puller to retrieve one pushed value from the Pusher. There are two possibilities: 1) The Pusher has already pushed the value or 2) The Pusher hasn't.

In the former case, the behaviour is trivial. The `pull_value` function indicates that it's ready to receive another pushed value and returns the current pushed value. In the trivial case, the scheduler is not invoked.

In the latter case, the Puller has to be suspended until the next pushed value is ready for retrieval. In this case, the Puller should invoke the scheduler to schedule the next thread.

The `push_value` function is called by the Pusher to push the value to the Puller. There are two possibilities: 1) The Puller has asked for the value and is now suspended or 2) The Puller hasn't.

In the latter case, the behaviour is trivial. The `push_value` function simply pass the value to the Puller and then call the `thd_yield`. The assumption is that the Puller should run before the Pusher runs next. This assumption should be guaranteed by the user of the library, otherwise, the behaviour is undefined.

In the former case, the Pusher pass the value to the Puller and then add the Pusher to the Ready Queue (more about the Ready Queue in the next section) before calling `thd_yield`.

## Scheduling

You need to maintain the states of the thread in a manner akin to the process states. In this thread library, the scheduler is invoked under 3 possible scenarios:

- Termination of a thread
- `thd_yield` is called
- `wait_thread` resulted in a thread waiting for another thread to complete.

You will need to maintain the following information:

(a) A queue of threads that are created and not yet ready to run e.g., stack is not allcoated yet, but they have TCB.

(b) A queue of threads that are ready to run i.e., they have a context and stack allocated already.

(c) Which thread is running right now.

(d) A collection of threads that has called `wait_thread` but the thread they have waited for has not yet completed.

(e) A collection of threads that has called `pull_value` but the Pusher has not yet pushed any value yet.

(f) A collection of threads that has completed but no threads have called `wait_thread` on them yet.

For our scheduling purposes, we assume that there exists two separate queues for the threads created but not ready to run and the threads ready to run.

The algorithm for scheduling is quite straightforward:

(a) If the ready queue is not empty, always run the threads in the ready queue first. The threads are selected from the ready queue in a FCFS manner.

(b) If the ready queue is empty, we will run a newly created thread. The newly created threads are chosen in a FIFO manner.

(c) The behaviour is undefined if both queues are empty.

## Context Switching

We have discussed the idea of a context during the lectures. In this section, we shall discuss the specific details of a context in terms of threads running in the e.g. x86-64 architectures. The context of a thread can be captured with the values of the following:

- Stack Pointer `rsp`, Frame pointer `rbp`
- The general purpose registers i.e., `rax, rbx, rcx, rdx, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15`
- Program Counter `rip`
- Conditional Flags `rflags`

At any point in time during the execution of the thread, the context of the thread can be captured by all the values of the listed items. To save the context of a thread, you need to keep these listed items in the memory somewhere. To restore the context of a thread, you need to copy the values stored previously in the memory back into these registers.

Out of these items, the Stack Pointer register can be addressed directly, while the program counter can only be accessed indirectly. To be specific, before any call to the scheduler (which may switch context), we need to save the context of the thread.

The general purpose registers of the x86-64 architecture listed in Table 1. Although these registers are used for "general purpose", the original x86-64 architecture was built with the intention of having specific purposes as listed in the Table 1.

| Register Name | Description of register |
|---------------|-------------------------|
| rax | Accumulator Register |
| rbx | Base Register (Frame Pointer) |
| rcx | Counter Register |
| rdx | Data Register |
| rsi | Source Index |
| rdi | Destination Index |
| rbp | Base Pointer |
| rsp | Stack Pointer |
| r8-r15 | Other General Purpose Registers |

Table 1:   Table of x86-64 General Purpose Registers (64 bits)

For this assignment, the important registers to note would be the stack pointer `rsp` and the base pointer `rbp`. The `rbp` is often used to compute the addresses of local variables relative to the stack frame. So, if the value of `rbp` gets corrupted, bad things might happen. Also, it is important to note that the stack pointer `rsp` will point to the current stack. Remember that a stack grow downwards for x86-64, from higher addresses to lower addresses. This has implications as to how the stack pointer of a new thread ought to be initialized.

The program counter is the register `rip` that points to the next instruction to be executed after the current one. However, if you have designed your code carefully, there is actually no need to store the rip value explicitly.

### x86-64 Instruction References and Assembly Inlining

In this section, we shall enumerate a few instructions that are relevant for this assignment. You are free to choose whatever instructions that may suit your purposes, should you find the need to use instructions not enumerated below. Each of the instructions below are explained in terms of assembly inlining. To recap, the key word `__asm` is used for the inlining of assembly code in Visual Studio. For instance, the statement

In this section, we shall enumerate a few instructions that are relevant for this assignment. You are free to choose whatever instructions that may suit your purposes,

should you find the need to use instructions not enumerated below. You are expected to use `gcc` assembly inlining. The syntax for gcc assembly inlining is a bit more complicated than Visual Studio. We will do a brief introduction here. The general form of the assembly inlining is the following:

```
asm [volatile] ( AssemblerTemplate
                 : OutputOperands
                 [ : InputOperands
                 [ : Clobbers ] ])
```

We give a brief introduction to each part of the assembly inlining syntax.

- `Assembler Template`. This refers to the assembly instructions that the programmer wish to insert into his program. These instructions must be written in the AT&T syntax.

  That is, consider the following:

  ```
  addq $8, %rax
  movq $0x100, 8(%rdi)
  ```

  The above shows assembly code that is written in 2 separate instructions - `add` and `mov`. Note that I did not write addq and movq. The q is a suffix for the instructions that indicate the size of the data being copied around. `l, w, b` are respectively the suffixes for the size of 4, 2 and 1 bytes. So the first instruction reads as incrementing the register `rax` by 8 while the second instruction reads as copying the values `0x100` into the address pointed to by register `rdi` plus 8. That is, we take that the register `rdi` contains an address as value, add 8 to that address, and take the dereference of the computed address as the destination for the value `0x100`. In the AT&T Syntax, the prefixes indicates the types of the operands. $ means that it is a constant while % means that what comes next is a register.

  To include the above same instructions using the assembly inline syntax for `gcc/g++`, we can write the following:

  ```
  asm volatile("addq $8, %%rax\n\t"
               "movq $0x100, 4(%%rdi)\n\t");
  ```

  That is, the so-called OutputOperands, InputOperands and Clobbers can be left blank if you so desire, syntactically speaking. Leaving them blank in general might not be a good idea.

- `Input/Output Operands`. Operands are the ways in which the assembler inline syntax in `gcc` allows us to use variables or identifiers in scope as part of the assembly. This is especially done so that we leave the tricky issue of allocating registers to particular variables to the compiler, trusting the compiler to do a better job. For example, we can consider the following code:

  ```
  int p, q;
  {
      int *x=&q;
      asm volatile("addq %2, %1\n\t"
  ```

```
                "movq $0x100, %0\n\t",
                : "=m" (*x), "+r" (p)
                : "r"(q) );
    }
```

Now, with similar instructions as before, the above code bears some explanation. First of all, `%0` and `%1` refers to the identifiers in the operand lists. Here, we have 3 operands: 2 output operands and 1 input operand.

The first output operand, referred to by `%0`, is `"=m"` `(*x)`. The operand proper is `*x`, which is the memory location pointed to by `x`. `"=m"` means that the operand is stored in memory and the compiler is meant to generate code for `%0` that will ensure that the memory location is written to. `"="` here means that this operand is write-only.

The second output operand, referred to by `%1`, is `"+r"` `(p)`. The operand proper is `p`. `"+r"` means that the operand should be stored in register and the compiler is meant to generate code for `%1` that will ensure that the register representing `p` is written to. `"+"` here means that this operand is both read and write.

The first input operand, referred to by `%2`, is `"r"` `(q)`. The operand proper is `q`. `"r"` means that the operand should be stored in register. Since it is an input operand, it does not need additional constraints to specify that it is read-only.

The reader should note that the numbering for the operands begins with the `OutputOperand` list and carries into the `InputOperand` list.

- `Clobber List`. This is the list to inform the compiler about registers that could be affected by side effects. An example of this in x86 is the following:

```
    {
        asm volatile("pushq $5\n\t"
                        :
                        :
                        : "rsp"
                        );
    }
```

Now the `push` instruction is (`"q"` here is a suffix indicating that the constant 5 takes up 8 bytes.) affects the value of the stack pointer indirectly. That is, `rsp` is not included as part of the specified operands of the instruction. However, its value is changed. So we include it in the clobber list here to inform the compiler of this fact.

Now we shall briefly describe each of the instructions relevant for this assignment. For more detailed information, the reader is referred to the Intel manuals here.

**mov**

In the x86-64 assembly language, the `mov` instruction is a mnemonic for the copying of data from one location to another. The `mov` instruction allows the movement of data from register to register, and register to memory and memory to register. To be sure, the following statement

8

```
asm volatile( "movq %%rbx, %%rax" );
```

will copy the contents of `rbx` and put it into `rax`. Also, the statement

```
int *my_var=foo();
asm volatile (
            "movl %0, %%ebx"
            :
            : "m"(*my_var)
            );
```

will copy the contents of the memory location of `my_var` into the register `rbx`. This is then a case of moving data from memory into register. The reverse is supported as well in the `mov` instruction i.e., moving from register into memory. However, the following statement

```
int *my_var=foo();
int *his_var=goo();
asm volatile (
            "movl %0, %1"
            : "=m"(*his_var),
            : "m"(*my_var),
            );
```

is not supported. This is because `mov` does not support a memory to memory movement of data. It should be noted that registers can contain addresses (especially in the case of registers like `rsp` and `rbp`). So a statement like

```
asm volatile( "movl %%eax, (%%rsp)" );
```

does not mean moving contents of register `eax` into register `esp`. Rather, it is copying the contents of `eax` into the address contained in the `esp` register. So the round brackets mean that you should use the contents of the register as a memory address.

**push**
The `push` instruction takes in one operand (it could either be a constant, register or memory location) onto the stack. The exact memory location written to depends on the stack pointer itself. More specifically, the stack pointer is decremented by 8 bytes before storing the value into the particular location pointed to by the stack pointer.

**pop**
The `pop` instruction takes in one operand (it could either be a register or memory location) and basically copies the 8 bytes pointed by the stack into the operand. The stack pointer is subsequently incremented by 8 bytes.

**pushfq, popfq**
`pushfq` decrements the `rsp` by 8 and pushes the entire contents of the `rflags` register onto the stack. `popfq` performs the reverse: pops the entire contents of the 8 byte addresses starting from where `rsp` is pointing to and then increment the `rsp` by 8.

**add/sub**

> add/sub instructions performs additions/subtraction on the given operands of the instruction. While the reader is free to use any form of add/sub if it suits his/her purposes, the form that might be relevant to this assignment is like the statement
>
> ```
> asm volatile( "addq $8, %%rsp" );
> ```
>
> which means that the rsp register is incremented by 8. This is the form where we can increment a general purpose register by a constant. (Note that the constant can be negative too.)

## Test Cases

In this section, we introduce some of the basic test cases to showcase some of the expected behaviour of the thread library. We provide a sample library `coro-lib.a` for your to test the expected behaviour.

### Test 1 - only one thread

```
1  #include <stdio.h>
   #include "new-coro-lib.h"
3


5
   int main()
7  {
       CORO::thd_init();
9      CORO::thd_yield();
        printf("This should print\n");
11 }
```

Listing 1: Test 1 - only one thread.

`only-one-thread.cpp` is provided. To compile with the given static library, use the following command:

```
g++ -std=c++17 only-one-thread.cpp coro-lib.a
```

The thread library should handle the case when the primary thread is the only thread in the system i.e., thd_yield() ought to be able to schedule the primary thread to run when it is the only thread in the process.

### Test 2 - switching threads

`switching-threads-example.cpp` is provided. To compile with the given static library, use the following command:

```
g++ -std=c++17 switching-threads-example.cpp coro-lib.a
```

In this example, the main thread creates a thread that will call `spin2` and a thread that will call `spin1`. The thread library should be able to rotate the scheduling between the two threads, causing SPIN1 to be printed with SPIN2 in an alternating manner.

```
1  #include <stdio.h>
   #include "new-coro-lib.h"
3  #pragma GCC diagnostic ignored "-Wunused-parameter"
   void *spin1(void *a)
5  {
       int i;
7      for(i=0; i< 20; i++)
       {
9          printf("SPIN1\n");
           if((i+1)%4==0)
11             CORO::thd_yield();
       }
13     return NULL;
   }
15
   void* spin2(void *a)
17 {
       int i;
19     for(i=0; i< 20; i++)
       {
21         printf("SPIN2\n");
           if((i+1)%4==0)
23             CORO::thd_yield();
       }
25     return NULL;
   }
27

29 int main()
   {
31     CORO::thd_init();
       CORO::ThreadID id = CORO::new_thd(spin2, NULL);
33     printf("Thread ID = %d\n", id);
       spin1(NULL);
35     printf("switching threads test DONE\n");
   }
```

Listing 2: Test 2 - switching threads.

### Test 3 - Thread waiting with parameter passing

`wait-thread-example.cpp` is provided. To compile with the given static library, use the following command:

`g++ -std=c++17 wait-thread-example.cpp coro-lib.a`

This test case creates two threads and the primary thread waits for the two threads before terminating the process.

```
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include "new-coro-lib.h"

/* prototype for thread routine */
void *print_message_function ( void *ptr );

/* struct to hold data to be passed to a thread
   this shows how multiple data items can be passed to a thread */
typedef struct str_thdata
{
    int thread_no;   char message[100];
} thdata;

int main()
{
    CORO::ThreadID thread1, thread2;  /* thread variables */
    thdata data1, data2;             /* structs to be passed to threads */

    /* initialize data to pass to thread 1 */
    data1.thread_no = 1;   strcpy(data1.message, "Hello!");

    /* initialize data to pass to thread 2 */
    data2.thread_no = 2;   strcpy(data2.message, "Hi!");

    /* create threads 1 and 2 */
    CORO::thd_init();
    thread1 = CORO::new_thd(print_message_function, (void *) &data1);
    thread2 = CORO::new_thd(print_message_function, (void *) &data2);

    /* Main block now waits for both threads to terminate, before it
         exits
         If main block exits, both threads exit, even if the threads have
             not
         finished their work */
    CORO::wait_thread(thread1, nullptr);
    printf("waiting thread1 finished\n");
    CORO::wait_thread(thread2, nullptr);

    printf("waiting thread2 finished\n");

    /* exit */
    exit(0);
} /* main() */

/**
 * print_message_function is used as the start routine for the threads
     used
 * it accepts a void pointer
**/
void *print_message_function ( void *ptr )
{
    thdata *data;
    data = (thdata *) ptr;  /* type cast to a pointer to thdata */

    /* do the work */
    printf("Thread %d says %s \n", data->thread_no, data->message);
```

```
56
      CORO::thread_exit(0); /* exit */
58      return nullptr;
} /* print_message_function ( void *ptr ) */
```

Listing 3: Test 3 - Thread waiting with parameter passing.

## Test 4 - waiting on the same thread twice

**double-wait-test.cpp** is provided. To compile with the given static library, use the following command:

g++ -std=c++17 double-wait-test.cpp coro-lib.a

```
1  #include <iostream>
   #include <cstdio>
3  #include "new-coro-lib.h"
   #pragma GCC diagnostic ignored "-Wunused-parameter"
5
   void *thd(void *param )
7  {
       std::cout << "Do nothing\n";
9      CORO::thd_yield();
       printf("after yield() but before thread function returns\n");
11      return nullptr;
   }
13

15  int main()
   {
17      CORO::ThreadID thd1_id;

19      CORO::thd_init();
       thd1_id =CORO::new_thd(thd, nullptr);
21      printf("First wait is %d\n", CORO::wait_thread(thd1_id, nullptr)==
           CORO::WAIT_SUCCESSFUL);
       printf("back to main\n");
23      printf("Second wait is %d\n", CORO::wait_thread(thd1_id, nullptr)==
           CORO::NO_THREAD_FOUND);
   }
```

Listing 4: Test 4 - waiting on the same thread twice.

This test case shows what happens when you call wait on the same thread twice. The first **wait_thread** is supposed to return **WAIT_SUCCESSFUL**. The second **wait_thread** is supposed to return **NO_THREAD_FOUND**.

## Test 5 - return value

**return-test1.cpp** is provided. To compile with the given static library, use the following command:

g++ -std=c++17 return-test1.cpp coro-lib.a

```
#include <cstdio>
2 #include <cstdlib>
#include "new-coro-lib.h"
4 #pragma GCC diagnostic ignored "-Wunused-parameter"

6 void *get_return_value(void *param )
{
8     static long count = 5;
    count--;
10     return (void *)count;
}
12

14 int main()
{
16     CORO::ThreadID thd_ids[5];
    int i;
18
    CORO::thd_init();
20
    for (i=0; i<5; i++)
22         thd_ids[i]=CORO::new_thd(get_return_value, NULL);

24     for (i=0; i<5; i++)
    {
26         long long thd_ret_value;
        CORO::wait_thread(thd_ids[i], (void**)&thd_ret_value);
28         printf("Thread %u returns %lld\n", thd_ids[i], thd_ret_value);
    }
30 }
```

Listing 5: Test 5 - return value.

This test case tests whether the main thread is able to obtain the correct return values from the created threads.

### Test 6 - scheduling

`schedule-test.cpp` is provided. To compile with the given static library, use the following command:

```
g++ -std=c++17 schedule-test.cpp coro-lib.a
```

This test case tests the scheduling mechanism of the thread library. You ought to ensure that your implemented library schedules in the same way as the provided static library.

```
#include <cstdio>
2 #include <cstdlib>
#include "new-coro-lib.h"
4 #pragma GCC diagnostic ignored "-Wunused-parameter"

6 void *thd1(void *param )
{
8     int *thd2_id;
```

```
        int thd2_res;
10      thd2_id = (int *) param;
        CORO::thd_yield();
12       printf("This is thread 1\n");
        CORO::wait_thread(*thd2_id, (void**)(&thd2_res));
14      CORO::thd_yield();
         printf("Thd2 returned %d\n", thd2_res);
16      CORO::thread_exit((void *) (long)(thd2_res+1));
        return nullptr;
18 }


20
   void *thd2(void *param )
22 {
        int *thd3_id;
24      int thd3_res;
        thd3_id = (int *) param;
26       printf("This is thread 2\n");
        CORO::wait_thread(*thd3_id, (void**)&thd3_res);
28      CORO::thd_yield();
         printf("Thd3 returned %d\n", thd3_res);
30      CORO::thread_exit((void *) (long)(thd3_res+1));
        return nullptr;
32 }


34
   void *thd3(void *param )
36 {
        int a=1024;
38      CORO::thd_yield();
         printf("This is thread 3\n");
40      CORO::thread_exit((void*)(long)a);
        return nullptr;
42 }


44
   int main()
46 {
        CORO::ThreadID thd1_id, thd2_id, thd3_id;
48      int *thd1_res;
        CORO::thd_init();
50      thd3_id =CORO::new_thd(thd3, NULL);
        thd2_id =CORO::new_thd(thd2, (void *)&thd3_id);
52      thd1_id =CORO::new_thd(thd1, (void *)&thd2_id);

54      CORO::wait_thread(thd1_id, (void**)&thd1_res);
         printf("Thd1 returned %ld\n", (long)thd1_res);
56 }
```

Listing 6: Test 6 - scheduling.


## Test 7 - fibonacci numbers

`fib-threads.cpp` is provided. To compile with the given static library, use the following command:

```
g++ -std=c++17 fib-threads.cpp coro-lib.a
```

This is a complex test case where the threads are creating more threads *recursively*.
See the comments within the code for more details.

```
1  #include <cstdio>
   #include <cstdlib>
3  #include "new-coro-lib.h"

5  /*
      This program computes the fibonacci series, where F(0)=0 and F(1)=1.
7
      For n>1, F(n) = F(n-1) + F(n-2)
9
      Taken and modified from midterm. Instead of forking, we create new
         threads
11    to compute F(n-1) and F(n-2) that we need.

13    This test case tests the case when threads create more threads.

15    Sample outputs:

17    $fib-threads 0
      fib(0) is 0
19
      $fib-threads 1
21    fib(1) is 1

23    $fib-threads 2
      fib(2) is 1
25
      $fib-threads 5
27    fib(5) is 5

29    $fib-threads 6
      fib(6) is 8
31
      $fib-threads 7
33    fib(7) is 13

35    $fib-threads 10
      fib(10) is 55
37
      $fib-threads 12
39    fib(12) is 144

41    $fib-threads 13
      fib(13) is 233
43
      $fib-threads 14
45    fib(14) is 377

47    You should be able to run up to fib-threads
   */
49
   void *fib_thd(void *param)
51  {
       long num;
53     num = reinterpret_cast<long>(param);
       if (num > 1)
55     {
```

```
            CORO::ThreadID t1, t2;
57          long t1_res, t2_res;

59          t1 = CORO::new_thd(fib_thd, (void*)(num-1));
            t2 = CORO::new_thd(fib_thd, (void*)(num-2));
61          CORO::wait_thread(t1, (void**) &t1_res);
            CORO::wait_thread(t2, (void**) &t2_res);
63          CORO::thread_exit((void*)(t1_res+t2_res));
            return nullptr;
65      }
        else
67          return (void*)num;
}

69
int main(int argc, char **argv)
71 {
        long num;
73      CORO::ThreadID my_thd;
        int result;

75
        if(argc != 2)
77      {
            printf("%s <number>", argv[0]);
79          return 1;
        }

81
        num = atoi(argv[1]);

83
        if(num < 0)
85      {
            printf("%s <number>", argv[0]);
87          printf("number must be positive!");
            return 1;
89      }

91      CORO::thd_init();
        my_thd = CORO::new_thd(fib_thd, (void*)num);
93      CORO::wait_thread(my_thd,(void**) &result);
        printf("fib(%ld) is %d\n", num,    result);
95 }
```

Listing 7: Test 7 - fibonacci numbers.

## Test 8 - matrix multiplication

matrix-multiply-test-case.cpp is provided. To compile with the given static library, use the following command:

g++ -std=c++17 matrix-multiply-test-case.cpp coro-lib.a

```
1 #include <cstdio>
  #include <cstdlib>
3 #include <ctime>
  #include <memory>
5 #include "new-coro-lib.h"
```

```cpp
/*
    This is a rather large test case that performs
    matrix multiplication in multiple user threads.

    The way to use this program:
        matrix−multiply−test−case <row> <common dimension> <col> <max_num_thd
            >

    It generates two random matrices. One is a <row> by <common dimension>
            matrix. The other
        is  <common dimension> by <col>. We compute the multiplication of the
                two and put
    it into a <row> by <col> solution matrix. The multiplication is done in
            two ways: one way
    would be to do all multiplication in one thread while the other way is
            to do the
    multiplication in several threads.

    The <max_num_thd> is the maximum number of threads you can generate to
            run this program.

    So far, in all my testing, the program results in printing "Correct
            Solution".

    The largest matrix I have tested so far is the following:
    matrix−multiply−test−case 1000 2000 1000 200
    Correct solution

    But this will take a long time so it is not recommended that you should
            be doing this test
    all the time.
*/


const int MAX_MATRIX_ELEMENT = (2<<8);

struct intInput
{
    int id, rows, cols, stride;
    int matrix2_rows;
    const int* matrix1;
    const int* matrix2;
    int* matrix_sol;
    intInput(const int * m1, const int * m2, int * m_sol)
        : matrix1(m1), matrix2(m2), matrix_sol(m_sol)
    {}
};

void error_msg(char *exec_filename)
{
    fprintf(stderr, "Usage: %s", exec_filename);
    fprintf(stderr, " <row> <common dimension> <col> <max_num_threads>");
    exit(-1);
}

int * make_new_matrix(int row, int col)
{
    int *new_matrix = new int[row*col];
```

```
57     return new_matrix;
   }

59
   void init_new_matrix(int row, int col, int* new_matrix)
61 {
       int i,j;

63
       for(i=0; i<row; i++)
65         for(j=0; j<col; j++)
               *(new_matrix + i*col + j) = rand()%MAX_MATRIX_ELEMENT;
67 }


69
   void slow_multiply( const int * matrix1,
71             const int * matrix2,
               int * new_matrix,
73             int new_row,
               int new_col,
75             int matrix2_row)
   {
77     int r,c, i;

79     for(r=0; r<new_row; r++)
       {
81         for(c=0; c<new_col; c++)
           {
83             int acc = 0;

85             for(i=0; i<matrix2_row; i++)
               {
87                 acc += matrix1[r*matrix2_row+i]*matrix2[i*new_col + c];
               }
89             new_matrix[r*new_col+c] = acc;
           }
91     }
   }

93

95 void *compute_mm_mult(void *data_ptr)
   {
97     int r,c;
       int max_rows, max_cols;
99     intInput *ds_ptr = reinterpret_cast<intInput*>(data_ptr);

101    r = ds_ptr->id;
       max_rows=ds_ptr->rows;
103    max_cols=ds_ptr->cols;

105    while(r < max_rows)
       {
107        for(c=0; c<max_cols; ++c)
           {
109            int acc=0;
               int i;
111            for(i=0; i<ds_ptr->matrix2_rows; i++)
               {
113                acc += (ds_ptr->matrix1)[r*ds_ptr->matrix2_rows+i]*(
                       ds_ptr->matrix2)[i*max_cols+c];
                   CORO::thd_yield();
```

```
115                         }
                           (ds_ptr->matrix_sol)[r*max_cols+c]=acc;
117                 }
               r+=ds_ptr->stride;
119
           }
121     delete ds_ptr;
        return nullptr;
123  }


125

     void mm_multiply( const int *m1,
127              const int *m2,
                 int* m_sol,
129              int max_num_threads,
                 int m_sol_rows,
131              int m_sol_cols,
                 int matrix2_rows)
133  {
          CORO::ThreadID *thread_ids;
135       int i;

137       thread_ids = new CORO::ThreadID[max_num_threads];

139       for(i=0; i<max_num_threads; ++i)
          {
141           struct intInput *mi = new intInput(m1, m2, m_sol);
              mi->id = i;
143           mi->rows = m_sol_rows;
              mi->cols = m_sol_cols;
145           mi->stride = max_num_threads;
              mi->matrix2_rows = matrix2_rows;

147           thread_ids[i] = CORO::new_thd(compute_mm_mult, reinterpret_cast<
                  void*>(mi));
149       }

151       for(i=0; i<max_num_threads; ++i)
          {
153           printf("wait thread %d \n", i);

155           CORO::wait_thread(thread_ids[i], NULL);
          }
157       delete thread_ids;
      }
159
     int matrix_compare(const int* matrix1,
161              const int* matrix2,
                 int rows,
163              int cols)
     {
165       int i, j;
          int e = 1;
167       for(i=0; i<rows; i++)
          {
169           for(j=0; j<cols; j++)
              {
171               if(matrix1[i*cols+j]!=matrix2[i*cols+j])
                  {
```

```
173                      e = 0;
                         return e;
175                  }
                 }
177          }
        return e;
179 }

181 int main(int argc, char **argv)
   {
183     int rows, common, cols;
        int max_num_threads;
185
        if(argc!=5)
187         error_msg(argv[0]);

189     rows=atoi(argv[1]);
        common=atoi(argv[2]);
191     cols=atoi(argv[3]);
        max_num_threads=atoi(argv[4]);
193
        srand(time(NULL));
195     int *matrix1 = make_new_matrix(rows, common);
        int *matrix2 = make_new_matrix(common, cols);
197     int *single_sol = make_new_matrix(rows, cols);
        int *mm_sol = make_new_matrix(rows, cols);
199
        init_new_matrix(rows, common, matrix1);
201     init_new_matrix(common, cols, matrix2);

203     slow_multiply(matrix1, matrix2, single_sol, rows, cols,  common);
        CORO::thd_init();
205     mm_multiply(matrix1, matrix2, mm_sol, max_num_threads, rows, cols,
            common);

207     if(matrix_compare(single_sol, mm_sol, rows, cols)==0)
            printf("Incorrect solution\n");
209     else
            printf("Correct solution\n");
211 }
```

Listing 8: Test 8 - matrix multiplication

This is a relatively complex test case where the resultant matrix is computed by different threads according to their allocated partition. See the comments within the code for more details.

### Test 9 - Fibonacci Numbers Push and Pull

`fib-source-sink.cpp` is provided. To compile with the given static library, use the following command:

`g++ -std=c++17 fib-source-sink.cpp coro-lib.a`

```
1 #include <iostream>
  #include "new-coro-lib.h"
```

```cpp
3  //#define RANDOM

5  void *fib(void *a)
   {
7      int num = *reinterpret_cast<int *>(a);
       long long first=0;
9      long long second=1;
       for(auto i=0; i<num; ++i)
11     {
           CORO::push_value(reinterpret_cast<void*>(first));
13         int tmp=second;
           second+=first;
15         first=tmp;
       }
17     return nullptr;
   }
19
   int main(int argc, char **argv)
21 {
       CORO::ThreadID id;
23     CORO::thd_init();
       int num;
25
#ifdef RANDOM
27     srand(time(NULL));
       num = rand()%10 + rand()%5;
29 #else
       if(argc != 2)
31     {
           printf("%s <number>", argv[0]);
33         return 1;
       }
35     num = atoi(argv[1]);
       if(num < 0)
37     {
           printf("%s <number>", argv[0]);
39         printf("number must be positive!");
           return 1;
41     }
#endif
43     id = CORO::new_thd(fib, &num);
       printf("Thread ID = %d\n", id);
45     for(long long i=0; i<num; ++i)
       {
47         long long value;
           CORO::pull_value(reinterpret_cast<void**>(&value));
49         std::cout << value << std::endl;
       }
51   return 0;
   }
```

Listing 9: Test 9 - Fibonacci Numbers Push and Pull

This is a test case where we have one puller and one pusher. The `fib` function, instead of restarting each time, continues one from the point where it pushed value until it's so repeatedly scheduled and reaches it is completion.

**Test 10 - context saving**

`context-saving-test-case.cpp` is provided. To compile with the given static library, use the following command:

`g++ -std=c++17 context-saving-test-case.cpp coro-lib.a`

```cpp
#include <stdio.h>
#include <cstdlib>
#include "new-coro-lib.h"
void *t(void *a)
{
  long  v_rbx, v_r12, v_r13, v_r14, v_r15;

  v_rbx = rand()%100;
  v_r12 = rand()%100;
  v_r13 = rand()%100;
  v_r14 = rand()%100;
  v_r15 = rand()%100;
  asm volatile
  (
      "movq %0, %%rbx\n\t"
      "movq %1, %%r12\n\t"
      "movq %2, %%r13\n\t"
      "movq %3, %%r14\n\t"
      "movq %4, %%r15\n\t"
    :
    : "r" (v_rbx),"r" (v_r12), \
      "r" (v_r13),"r" (v_r14),"r" (v_r15)
    : "rbx", "r11","r12", "r13", "r14"
  );

  CORO::thd_yield();

  long  rbx, r12, r13, r14, r15;

  asm volatile
  (
      "movq %%rbx, %0\n\t"
      "movq %%r12, %1\n\t"
      "movq %%r13, %2\n\t"
      "movq %%r14, %3\n\t"
      "movq %%r15, %4\n\t"
      : "+m" (rbx), "+m" (r12),\
      "+m" (r13), "+m" (r14), "+m" (r15)
    :
    :
  );

  if (rbx!=v_rbx || \
    r12!=v_r12||r13!=v_r13||r14!=v_r14||r15!=v_r15)
    printf ("%s: ERROR \n", (char*)a);
  else
    printf("%s: %ld %ld %ld %ld %ld\n", (char*)a, \
        rbx, r12, r13, r14, r15);

  CORO::thd_yield();
```

```
52    return NULL;
}
54
int main()
56 {
        CORO::ThreadID id;
58      CORO::thd_init();
        char p1[] = "t1";
60      char p2[] = "t2";

62      id = CORO::new_thd(t, (void*)p2);
        printf("Thread ID = %d\n", id);
64      t( (void*)p1);
}
```

Listing 10: Test 10 - context saving

This is a simple test case where we can verify whether context saving for some registers is correct. See the comments within the code for more details.

## Rubrics

Your job is to try to implement the thread library described above with all the information given to assist you in this task. All coding should be done in $C++$. If you understood the material and implemented your library carefully, you should not be required to write more than 250 lines of code (not including comments and the access functions for the TCB data structure).

The code that you write should use the provided header in `new-coro-lib.h` that is listed in the appendices section.

What to submit:

- A file `new-coro-lib.cpp` that contains the implementation of the user-level library.
- You **should not** submit `new-coro-lib.h` i.e., your code should work with the provided `new-coro-lib.h`.

Submission of these files should be according to the stipulations already stated in the syllabus.

In general, the rubrics for this assignment are the following:

- Passing test cases.
- Comments or documentation.
- Coding style and compilation without warnings.

Students are required to provide succinct comments to some of the design decisions made. Example of design decisions include:

- The contents of the TCB structure.

- The mechanism of how the exit or return value of a thread is passed to the thread waiting for it.

- Whether the scheduler is a separate function from `thd_yield()` - explain why you chose to do the way you did it. (Hint: "I like it this way" is not good enough an explanation.)

The list above is non-exhaustive. The lecturer reserves the right to impose reasonable penalties for code that violates general practices or does not match the specification in an obvious way that has not been mentioned above. In exceptional cases, the lecturer reserves a discretionary right to allow resubmission or submission after the deadline.

# Appendices

## A    new-coro-lib.h

<div align="center">Listing 11: new-coro-lib</div>

```cpp
#ifndef NEW_CORO_LIB_H
#define NEW_CORO_LIB_H
namespace CORO
{
using ThreadID = unsigned;
void thd_init();
ThreadID new_thd( void*(*)(void*), void *);
void thread_exit(void *);
int wait_thread(ThreadID id, void **value);
void thd_yield();
void push_value(void*);
void pull_value(void**);
const int WAIT_SUCCESSFUL = 0;
const int NO_THREAD_FOUND = -1;
enum ThreadState : int;
}
#endif
```