Assignment #3

CSD2180 Fall 2022

Due Date:	Refer to CSD2180 page. All submissions (programming) must be uploaded to the assignment page link.
Topics covered:	Threading, context switching, x86/64 assembly programming
Objectives:	To learn and understand context switching at some specific level using x86/64 assembly programming (). 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

If you copy, you are less prepared for the exams, you aren't learning, and you violate the principles of academic integrity.
For this Assignment, you should avoid copying anyone's (or Internet) code that implements user-level threading lib.

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 with a function.
- 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` and test cases. They should be exactly the same.
- To help you implementation, sample inline assembly and skeleton code are provided.

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. Try to figure out what is supposed to be initialized in this function.

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.

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 newly created.

(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 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 LIFO manner.

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

x86/64 Registers and 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 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. 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

```
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.)

Important information: The stack pointer in general should be kept aligned at 16 bytes.

Windows calling convention:

The registers RAX, RCX, RDX, R8, R9, R10, R11 are considered volatile and must be considered destroyed on function calls. The registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15 are considered nonvolatile and must be saved and restored by a function that uses them.

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


5


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

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++14 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++14 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.

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

void *spin1(void *a)
{
    int i;
    for(i=0; i< 20; i++)
    {
        printf("SPIN1\n");
        if((i+1)%4==0)
            CORO::thd_yield();
    }
    return NULL;
}

void* spin2(void *a)
{
    int i;
    for(i=0; i< 20; i++)
    {
        printf("SPIN2\n");
        if((i+1)%4==0)
            CORO::thd_yield();
    }
    return NULL;
}


int main()
{
    CORO::ThreadID id;
    CORO::thd_init();
    id = CORO::new_thd(spin2, NULL);
    spin1(NULL);
}
```

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++14 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);
    CORO::wait_thread(thread2, nullptr);

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

/**
 * print_message_function is used as the start routine for the threads
     used
 * it accepts a void pointer
```

```
45 **/
   void *print_message_function ( void *ptr )
47 {
       thdata *data;
49     data = (thdata *) ptr;  /* type cast to a pointer to thdata */

51     /* do the work */
       printf("Thread %d says %s \n", data->thread_no, data->message);
53
       CORO::thread_exit(0); /* exit */
55 } /* 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++14 double-wait-test.cpp coro-lib.a
```

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


5
  void *thd(void *param )
7 {
      std::cout << "Do nothing\n";
9     CORO::thd_yield();
      return nullptr;
11 }


13
  int main()
15 {
      CORO::ThreadID thd1_id;
17
      CORO::thd_init();
19    thd1_id =CORO::new_thd(thd, nullptr);
      printf("First wait is %d\n", CORO::wait_thread(thd1_id, nullptr)==
          CORO::WAIT_SUCCESSFUL);
21    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++14 return-test1.cpp coro-lib.a
```

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


void *get_return_value(void *param )
{
    static int count = 5;
    count--;
    return (void *)count;
}


int main()
{
    CORO::ThreadID thd_ids[5];
    int i;

    CORO::thd_init();

    for (i=0; i<5; i++)
        thd_ids[i]=CORO::new_thd(get_return_value, NULL);

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

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++14 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>
#include <cstdlib>
#include "new-coro-lib.h"

void *thd1(void *param )
{
    int *thd2_id, thd2_res;
    thd2_id = (int *) param;
    CORO::thd_yield();
    printf("This is thread 1\n");
    CORO::wait_thread(*thd2_id, (void**)(&thd2_res));
    CORO::thd_yield();
    printf("Thd3 returned %d\n", thd2_res);
    CORO::thread_exit((void *) (thd2_res+1));
    return nullptr;
}


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


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


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

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

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++14 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.

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

/*
   This program computes the fibonacci series, where F(0)=0 and F(1)=1.

   For n>1, F(n) = F(n-1) + F(n-2)

   Taken and modified from midterm. Instead of forking, we create new
       threads
   to compute F(n-1) and F(n-2) that we need.

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

   You should be able to run up to fib-threads
*/

void *fib_thd(void *param)
{
    long num;
    num = reinterpret_cast<long>(param);
    if (num > 1)
    {
        CORO::ThreadID t1, t2;
        long t1_res, t2_res;

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

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

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

```
47      }

49      num = atoi(argv[1]);

51      if(num < 0)
        {
53          printf("%s <number>", argv[0]);
            printf("number must be positive!");
55          return 1;
        }
57
        CORO::thd_init();
59      my_thd = CORO::new_thd(fib_thd, (void*)num);
        CORO::wait_thread(my_thd,(void**) &result);
61      printf("fib(%d) is %d\n", num,    result);
}
```

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++14 matrix-multiply-test-case.cpp coro-lib.a
```

```
#include <cstdio>
2 #include <cstdlib>
#include <ctime>
4 #include <memory>
#include "new-coro-lib.h"
6 /*
   This is a rather large test case that performs
8  matrix multiplication in multiple user threads.

10 The way to use this program:
     matrix-multiply-test-case.exe <row> <common dimension> <col> <
         max_num_thd>
12
   It generates two random matrices. One is a <row> by <common dimension>
       matrix. The other
14   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
16 would be to do all multiplication in one thread while the other way is
       to do the
   multiplication in several threads.
18
   The <max_num_thd> is the maximum number of threads you can generate to
       run this program.
20
   So far, in all my testing, the program results in printing "Correct
       Solution".
22
```

```
   The largest matrix I have tested so far is the following:
   matrix-multiply-test-case.exe 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.
*/

using MType = int;
using MTypePtr = std::unique_ptr<MType[]>;
using Matrix = std::unique_ptr<MTypePtr[]>;

const int MAX_MATRIX_ELEMENT = (2<<20);

struct MatrixInput
{
    int id, rows, cols, stride;
    int matrix2_rows;
    const Matrix& matrix1, &matrix2;
    Matrix& matrix_sol;
    MatrixInput(const Matrix & m1, const Matrix& m2, Matrix& 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);
}

void make_new_matrix(int row, int col, Matrix& new_matrix)
{
    int i;
    new_matrix = std::make_unique<MTypePtr[]>(row);

    for(i=0; i<row; i++)
    {
        new_matrix[i] = std::make_unique<MType[]>(col);
    }
}

void init_new_matrix(int row, int col, Matrix& new_matrix)
{
    int i,j;
    for(i=0; i<row; i++)
        for(j=0; j<col; j++)
            new_matrix[i][j] = rand()%MAX_MATRIX_ELEMENT;
}


void slow_multiply(const Matrix& matrix1, const Matrix& matrix2, Matrix&
    new_matrix,
                   int new_row, int new_col, int matrix2_row)
{
    int r,c, i;
```

```
80      for(r=0; r<new_row; r++)
        {
82          for(c=0; c<new_row; c++)
            {
84              MType acc;
                acc = 0;
86
                for(i=0; i<matrix2_row; i++)
88              {
                    acc += matrix1[r][i]*matrix2[i][c];
90              }
                new_matrix[r][c] = acc;
92          }
        }
94  }

96
    void *compute_mm_mult(void *data_ptr)
98  {
        int r,c;
100     int max_rows, max_cols;
        MatrixInput *ds_ptr = reinterpret_cast<MatrixInput*>(data_ptr);
102
        r = ds_ptr->id;
104     max_rows=ds_ptr->rows;
        max_cols=ds_ptr->cols;
106
        while(r < max_rows)
108     {
            for(c=0; c<max_cols; ++c)
110         {
                MType acc;
112             int i;
                acc=0;
114             for(i=0; i<ds_ptr->matrix2_rows; i++)
                {
116                 acc += (ds_ptr->matrix1)[r][i]*(ds_ptr->matrix2)[i][c];
                    CORO::thd_yield();
118             }
                (ds_ptr->matrix_sol)[r][c]=acc;
120         }
            r+=ds_ptr->stride;
122     }
        delete ds_ptr;
124     return nullptr;
    }
126

128 void mm_multiply(const Matrix &m1,Matrix &m2, Matrix& m_sol,
                    int max_num_threads, int m_sol_rows,
130                 int m_sol_cols, int matrix2_rows)
    {
132     CORO::ThreadID *thread_ids;
        int i;
134
        thread_ids = new CORO::ThreadID[max_num_threads];
136

138     for(i=0; i<max_num_threads; ++i)
```

```
    {
140         struct MatrixInput *mi = new MatrixInput(m1, m2, m_sol);
            mi->id = i;
142         mi->rows = m_sol_rows;
            mi->cols = m_sol_cols;
144         mi->stride = max_num_threads;
            mi->matrix2_rows = matrix2_rows;
146
            thread_ids[i] = CORO::new_thd(compute_mm_mult, reinterpret_cast<
                void*>(mi));
148     }

150     for(i=0; i<max_num_threads; ++i)
        {
152         CORO::wait_thread(thread_ids[i], NULL);
        }
154     delete thread_ids;
}
156
int matrix_compare(const Matrix& matrix1, const Matrix& matrix2,
158                    int rows, int cols)
{
160     int i, j;
        for(i=0; i<rows; i++)
162         for(j=0; j<rows; j++)
                if(matrix1[i][j]!=matrix2[i][j])
164                 return 0;
        return 1;
166 }

168 int main(int argc, char **argv)
{
170     int rows, common, cols;
        int max_num_threads;
172     Matrix matrix1, matrix2, single_sol, mm_sol;

174     if(argc!=5)
            error_msg(argv[0]);
176
        rows=atoi(argv[1]);
178     common=atoi(argv[2]);
        cols=atoi(argv[3]);
180     max_num_threads=atoi(argv[4]);

182     srand(time(NULL));
        make_new_matrix(rows, common, matrix1);
184     make_new_matrix(common, cols, matrix2);
        make_new_matrix(rows, cols, single_sol);
186     make_new_matrix(rows, cols, mm_sol);
        init_new_matrix(rows, common, matrix1);
188     init_new_matrix(common, cols, matrix2);

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


194
        if(matrix_compare(single_sol, mm_sol, rows, cols)==0)
```

```
196             printf("Incorrect solution\n");
        else
198             printf("Correct solution\n");
    }
```

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.

The Assignment

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 300 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.

Deliverables:

- Zip you folder and name the resulting file using the following convention:
  class_sitid_assignmet#.zip
  For example, if your sitid is foo and you are submitting assignment 3, your zipped file would be named as: csd2180_foo_3.zip
- Your folder should contain:
  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`.

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

- Passing test cases.
- Comments and Coding style.

The following table shows the rubrics measured for each part of the submission.

	Test Cases tested	Comments	Coding style and compilation
Submission	Tests 1 to 8 (90%)	Graded (5%)	Graded (5%)

Appendices

A new-coro-lib.h

Listing 9: new-coro-lib

```
#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();
    const int WAIT_SUCCESSFUL = 0;
    const int NO_THREAD_FOUND = -1;
    enum ThreadState : int;
}
#endif
```