A Cooperative Scheduled User-Level Thread Library

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).

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 task, you will write functions that allow the creation, termination, suspension and simple scheduling of the threads. 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.
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:
   1. By calling the thread exit function.
   2. 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.

1. **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:
2. 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:
   1. Stack pointer: Points to thread’s stack in the process (this is stored within ucontext\_t)
   2. Program counter
   3. State of the thread (running, ready, waiting, start, done)
   4. register values for thread
3. 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.
4. 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:

* 1. Puller is the thread that created the Pusher.
  2. Pusher is the created thread.
  3. Puller and Pusher is strictly a 1:1 relationship.
  4. 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:

1. A queue of threads that are created and not yet ready to run e.g., stack is not allocoated yet, but they have TCB.
2. A queue of threads that are ready to run i.e., they have a context and stack allocated already.
3. Which thread is running right now.
4. A collection of threads that has called wait\_thread but the thread they have waited for has not yet completed.
5. A collection of threads that has called pull\_value but the Pusher has not yet pushed any value yet.
6. 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:

1. 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.
2. If the ready queue is empty, we will run a newly created thread. The newly created threads are chosen in a FIFO manner.
3. The behaviour is undefined if both queues are empty.

**Context Switching**

In this section, we shalldiscuss 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 task, 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.

**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

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.

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.

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

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

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.

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.

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

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

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

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.

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

Please 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.