In this practical work, you will first explore *coroutines* and then use them to develop a cooperative multitasking environment managed by a *task manager*. You can think of this as building a very basic operating system. First, you will implement the task manager for a single CPU core first, then extend your implementation to use two CPU cores by using synchronization primitives.

This is a two-week assignment. You will complete some programming tasks that require you to upload (1) your source code and (2) the program output. We will provide a Moodle activity to upload your code and your report. Create a ZIP file with name `SCIPER_surname_name_tp7.zip`, which contains:

- your report with file name `report.pdf`
- the project directory `project/`
- the program output `output.txt`

Part 1: Coroutines (20 points)

Most programmers are familiar with *subroutines*, which are functions that execute sequentially, starting from the beginning and running until they return control to their caller. They cannot be paused or resumed midway. In contrast, *coroutines* are generalizations of subroutines that allow multiple entry points. A coroutine can pause its execution (yield) at a specific point and later resume from that same point, maintaining its state between invocations. While subroutines follow a strict call-return relationship, coroutines collaborate and transfer control back and forth, enabling cooperative multitasking and stateful operations. Coroutines are user-controlled entities; they are explicitly created, managed, and scheduled by the programmer, offering fine-grained control over execution. Many major operating systems and programming languages provide support for coroutines under different names. For example: **fibers** in Windows, **coroutines** in C++ (since C++20), and **generators and async functions** in Python.

To help you get started, we provide a **primitive coroutine library** (`include/coro/coro.h`) designed for the OpenRISC platform. Later, we will ask you to complete a **task manager** framework (`include/taskman/taskman.h`) that extends the use of coroutines. This setup enables you to understand coroutines and cooperative multitasking in a low-level context.

Question 1. (4 points) Briefly explain the difference between the following concepts:

1. *Concurrency* and *parallelism*
2. *Cooperative multitasking* and *preemptive multitasking*
3. *Threads* and *coroutines*
4. *Stackful* and *stackless* coroutines

Question 2. (5 points) Compile and execute the project. Compare and relate the program output with the program source code `src/part1.c`. Explain the role of each `coro_` function call.

You can use the following command to compile the project:

```
cd $PROJECT_DIR
export PATH=$OR1K_ROOT/bin:$PATH
make TARGET=OR1300 DEBUG=0  # the output folder is build-release-or1300
```

Question 3. (4 points) Explain how `coro_resume`, `coro_yield`, and `coro_return` are implemented. These functions are defined in `src/coro/coro.c`.

Question 4. (3 points) What is the intended role of processor register `r10` according to the architecture manual? What purpose does it serve in the coroutine library implementation?

Question 5. (4 points) Explain how `coro__switch` is implemented. What registers does it save? It is declared in `src/coro/coro.c` and defined in `src/coro/coro.s`.

Part 2: Task Manager (80 points)

In a fairly complex system, many coroutines coexist and are executed in a loop. Managing these coroutines manually can become cumbersome for the user due to several challenges:

- **Large numbers of coroutines:** There might be many coroutines to track, making manual management inefficient and error-prone.
- **Dynamic creation:** Coroutines might be created dynamically, such as from within another coroutine during execution, leading to unpredictable runtime behavior.
- **Conditional continuation:** A coroutine's continuation might depend on specific events, such as receiving data from a UART or the expiration of a timer.
- **Memory management:** Each coroutine is associated with a stack that must be allocated from a region.

A *task manager* is an abstraction built on top of coroutines to address these challenges. It provides a higher-level interface that simplifies the management of coroutines by handling their lifecycle, scheduling, and event-driven execution. In this abstraction a *task* is a corutine managed by the task manager. This abstraction enables the system to scale while maintaining clarity and control over the coroutines. The task manager allows the user to spawn tasks that are executed concurrently. While one task waits for an event (such as a timer or user input), other tasks can advance. In this part of the practical work, you will complete the implementation of a task manager for a single core.

Using the Task Manager

- **Initialization:** The user must first follow these steps before using the task manager:
  1. Call `coro_glinit` for each CPU core to initialize the coroutine system.
  2. Call `taskman_glinit` once to set up the task manager library globally.
  3. Register necessary handlers. Handlers will be explained later in this document.
  4. Spawn the initial tasks using the `taskman_spawn` function.
- **Executing the Task Manager Loop:** Each CPU core intended to execute the task manager's main loop should call the `taskman_loop` function. This function blocks until the loop is killed via `taskman_stop`.
- **Lifecycle Management of Tasks:** The following functions manage the lifecycle of tasks, which might be called from within a task:
  - `taskman_spawn`: Dynamically spawns new tasks. This function does not block and allows the system to create additional tasks at runtime.
  - `taskman_yield`: Temporarily yields control back to the task manager's main loop. This allows the task manager to continue iterating and potentially schedule other tasks for execution.
  - `taskman_wait`: Similar to `taskman_yield`, takes a `struct taskman_handler *` as an argument, which determines when the task can resume execution. This mechanism enables tasks to pause and wait for specific conditions, such as an event or a signal. Please note that `taskman_yield` and `taskman_wait` have very similar implementations; therefore, the library simply calls `taskman_wait` with `NULL` arguments for yielding.

Task Manager Data Structures

- **Global State:** The task manager library is implemented as a singleton, i.e., it relies on global state defined in `src/taskman/taskman.c` as a variable with name `taskman`. This variable (1) maintains an array of taskman handlers, which we will describe later, (2) holds and manages a large memory region used as stack by tasks, (3) maintains an array of pointers to stacks of individual tasks, and (4) should-stop flag.

- **Handlers:** A handler (described with `struct taskman_handler`, defined in `include/taskman/taskman.h`) enables tasks to wait on a particular condition. For example, waiting on a timer event is handled by the tick handler (defined in `src/taskman/tick.c`) or waiting on UART data is handled by the UART handler, which you will implement as part of this assignment. A handler instance holds function pointers to three functions: `on_wait`, `can_resume`, and `loop`. These functions are called by the task manager APIs (like `taskman_wait` and `taskman_loop`) so that tasks can wait for a specific condition to realize.

  - `on_wait` is called when the task intends to yield control to the main loop via `taskman_wait`. If it returns `0`, yield is complete, and the main loop keeps iterating. It takes as arguments a pointer to the handler itself, a pointer to the stack of the current task, and the argument passed to `taskman_wait`.

  - `can_resume` is called by the task manager main loop (i.e., `taskman_loop`) to see if the task can be resumed. If it returns `1`, the task is resumed. It takes as arguments a pointer to the handler itself, a pointer to the stack of the current task, and the argument passed to `taskman_wait`.

  - `loop` is called at the beginning of each main loop iteration.

  For the dual-core task manager implementation, all the handler functions are called from within a critical section.

  Handlers must be first registered to task manager using `taskman_register` function before calling `taskman_loop`. User argument passed to `on_wait` and `can_resume` functions allows the handler to implement specific functionality. For instance, handlers can (1) wait on a semaphore by passing a pointer to the semaphore as argument or (2) wait for a time interval by passing a time point as argument. You can read the source code of the tick handler (implemented in `src/taskman/tick.c`) as an example.

- **Task Data:** Task data (described with a `struct task_data`, defined in `src/taskman/taskman.c`) describes the individual task state. It is stored as the coroutine data (you can get a pointer to which using `coro_data`). It keeps information such as (1) if the task is being executed on any CPU core right now, (2) on what handler the task is waiting, and (3) the argument passed to the handler.

For the programming tasks below, do not forget to read the comments in source files and the function declarations in header files. This document might not specify the precise behavior expected in the implementation.

Part 2.1: Single-core Task Manager Implementation (60 points)

In this section of the assignment, you will implement missing parts of the task manager implementation for a single-core implementation.

Question 6. (20 points) Complete the task manager implementation in `src/taskman/taskman.c`. Missing parts of the code are marked with `IMPLEMENT_ME`.
You can test your implementation by uncommenting the "`basic test`" section of the `entry_task`.

Question 7. (5 points) What can you say about `task_data->running` flag? For a single-core implementation, can you optimize it out?

Question 8. (15 points) Complete the semaphore implementation in `src/taskman/semaphore.c`. Missing parts of the code are marked with `IMPLEMENT_ME`.
You can test your implementation by uncommenting the "`semaphore test`" section of the `entry_task`.

Question 9. (20 points) Complete the UART implementation in `src/taskman/uart.c`. Missing parts of the code are marked with `IMPLEMENT_ME`.
You can test your implementation by uncommenting the "`uart test`" section of the `entry_task`.

Part 2.2: Dual-core Task Manager Implementation (20 points)

In this section of the assignment, you will use the synchronization primitives `TASKMAN_LOCK()` and `TASKMAN_RELEASE()` to sequentialize the critical sections of the task manager. The dual-core implementation must ensure that any function call that might alter the internal state of the task manager is safe.

Even if your dual-core implementation appears to work correctly without using locks, it does not guarantee that your program is free of errors. Locks are essential for ensuring program correctness by preventing rare and hard-to-detect race conditions that could lead to unpredictable behavior or data corruption.

Question 10. (5 points) Read `support/src/locks.c` file and explain how the lock mechanism works.

Question 11. (15 points) For your single-core task manager implementation, identify possible race conditions. What lines in the source code should you insert `TASKMAN_LOCK()` and `TASKMAN_RELEASE()` primitives to prevent these race conditions? Explain your reasoning. Implement your changes and verify that your code works.

For Questions 6, 8, 9, and 11, you will receive full points only if your program output clearly demonstrates that the corresponding parts of the implementation are functional, even if the implementation is not fully optimized.