Objectives

In this practical work, you will be experimenting with multitasking using coroutines.

Requirements

Make sure that:
1. the project files are downloaded and extracted,
2. your Gecko board has the final version of the firmware,
3. the toolchain is functional (it should be `or1k-elf-gcc`).

All the file paths mentioned in this document are relative to the downloaded archive.

Submission

This practical work contains some programming tasks. You will be provided with a Moodle activity to upload your code and your report. Please create a ZIP file with name `SCIPER_surname_name_tp6.zip`, which contains your report with file name `report.pdf` and the project directory `coopmt/`.

Part 1: Coroutines

Question 1. Use your favorite search engine to learn about and (concisely) explain:
1. *Concurrency* and *parallelism*
2. *Cooperative multitasking* and *preemptive multitasking*
3. *Subroutines* and *coroutines*
4. *Stackful* and *stackless* coroutines

We provide a coroutine library for OpenRISC 1000. In this part of the practical work, you are required to understand how it is implemented. The source code of the library is provided in `taskman/include/coro/` and `taskman/src/coro/`. For the following questions, you need to analyze the source code of the library. Comments in the source code are your friend, so please read them (for both parts).

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

Question 3. What kind of coroutines are implemented in the library: stackless or stackful?

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

Question 5. 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 6. Explain how `coro__switch` is implemented. What registers does it save? It is declared in `taskman/src/coro/coro.c` and defined in `taskman/src/coro/coro.s`.

Part 2: Task Manager

In this part of the practical work, you will complete the implementation of a task manager library built on top of the coroutine library. The task manager library allows the user to spawn tasks that are executed concurrently. While one task is waiting for an event (like a timer or user input), other tasks can advance.
You can find its source code in `taskman/include/taskman/` and `taskman/src/taskman/`. Missing parts of the source code (that you are supposed to fill) are marked by `IMPLEMENT_ME`. You might find comments describing the desired functionality. The following files have missing parts in them:

Question 7. Read the program source code `taskman/src/part2.c`. Explain the role of each `taskman_` function call. What do you expected the program do? Hint: look at Figure 1 to have some idea.

```
now executing: part1
func = part1
func = test_fn, data = 0xaaaabeef
func = test_fn, i = 0, arg = 0xdeadbeef
func = part1
func = f, x = 15
func = part1
func = test_fn, f(15 + i) = 75
func = test_fn, i = 1, arg = 0xdeadbeef
func = part1
func = f, x = 16
func = part1
func = test_fn, f(15 + i) = 76
func = test_fn, i = 2, arg = 0xdeadbeef
func = part1
func = f, x = 17
func = part1
func = test_fn, f(15 + i) = 77
func = test_fn, i = 3, arg = 0xdeadbeef
func = part1
func = f, x = 18
func = part1
func = test_fn, f(15 + i) = 78
done.
result = 10
now executing: part2
wait_task: now = 1001 ms (period = 1000)
wait_task: now = 2002 ms (period = 1000)
wait_task: now = 3001 ms (period = 3000)
wait_task: now = 3006 ms (period = 1000)
wait_task: now = 4007 ms (period = 1000)
wait_task: now = 5008 ms (period = 1000)
wait_task: now = 6002 ms (period = 3000)
wait_task: now = 6009 ms (period = 1000)
wait_task: now = 7010 ms (period = 1000)
wait_task: now = 8011 ms (period = 1000)
wait_task: now = 9001 ms (period = 9000)
wait_task: now = 9006 ms (period = 3000)
wait_task: now = 9012 ms (period = 1000)
```

Figure 1: Expected output. Includes parts for both parts.

Question 8. What kind of multitasking does the library implement: cooperative or preemptive?

Question 9. Complete the missing parts in `src/taskman/taskman.c`, which constitutes as the core of the task manager library. Verify the functionality by (1) commenting out `taskman_spawn(&uart_task, NULL, 5120);` in `taskman/src/part2.c` and (2) executing the program. Note that the expected output should be as shown in Figure 1. We strongly suggest that you read `src/taskman/tick.c` to better understand how the library internals work.

Question 10. Complete the missing parts in `src/taskman/uart.c`. Verify the functionality by (1) uncommenting `taskman_spawn(&uart_task, NULL, 5120);` in `taskman/src/part2.c` and (2) executing the program. Try sending data over CuteCom (or whatever software you use), the program should echo back what you wrote.