CS-473: System Programming for SOCs

Practical Work 8:
Multitasking using Coroutines

Report

Andrea Grillo

371099

Index

Part 1: Coroutines

Question 1

1. Concurrency and parallelism

   Both terms refer to the possibility of having multiple tasks run on a machine at the same time.

   Concurrency means that the tasks are run in overlapping time periods, with the illusion of parallelism that is given by the speed of the CPU, but are in fact run one at a time on a single thread CPU.

   Parallelism means that the tasks are actually run at the same time, on a multi thread CPU or on a system with multiple CPUs.

2. Cooperative multitasking and preemptive multitasking

   In multitasking, the main issue is the sharing of the resources, above all the CPU. Cooperation and preemption are the two main ways of managing CPU time in a multitasking environment.

   In cooperation, each task is responsible for executing a chunk of code and yielding control to the caller by initiating a context switch. The caller is usually a process manager, who will resume another task who is waiting.

   Instead, in a preemptive multitasking environment, the operating system is responsible for the interruption of the execution of each thread. A time slot is allocated for each task, and at the end of each time slot the process is interrupted and the operating system initiates the context switch towards another task.

3. Subroutines and coroutines

   A subroutine is a simple function, it can be called, with or without arguments to it, then the function runs, it can store a return value, and then it resumes the code that started its execution.

   A coroutine is like a subroutine, but with two main extra capabilities. A coroutine can be suspended, and it can be resumed. This allows for the implementation of multitasking systems, by suspending and resuming multiple coroutines that are in this manner executed concurrently.

4. Stackful and stackless coroutines

   Stackful coroutines use a separate stack frame for each coroutine, similar to a normal function call. A downside of this approach is that managing separate stacks for each coroutine can be more resource-intensive, as it requires memory allocation for each stack.

   Stackless coroutines, on the other hand, share a single stack among multiple coroutines. Instead of having a dedicated stack for each coroutine, they use a mechanism to save and restore their execution state within a shared stack. This approach is often more lightweight in terms of memory usage, as it avoids the overhead of maintaining separate stacks. However, the mechanism to manage the execution flow can be more challenging to write.

Question 2

At the beginning of the *part1* function, *coro_init* is called to initialize a coroutine, using the stack *stack0*, the *test_fn* as the task to be run, and passing an hex argument to the function.

Then the *coro_data* function is used to get the address at which save data for the coroutine.

Then the *coro_resume* function is used to start or resume the coroutine which has been initialized at stack *stack0*.

The *coro_resume* therefore starts the test_fn coroutine, which gets the data passed to it using *coro_data* and the argument using *coro_arg*. The test_fn has a call to the *coro_yield* function to suspend its execution and a call to the *f* function, which in turn calls the *coro_yield* function to suspend again the execution until the resume is called from *part1*.

At the end of the execution, the *part1* function checks if the execution of the coroutine has finished using the *coro_completed* function, which also allows saving the result of the execution.

Question 3

The library implements coroutines in a stackful manner, as each coroutine has its own stack frame that is saved and restored in the stack by the *coro__switch* function.

Question 4

1. *coro_resume*

   The resume function receives as a parameter a void pointer, which is then casted to a pointer to a *coro_data* struct. This structure contains the data of the coroutine that has to be resumed. The function checks if it has been called by a coroutine, which is not possible, and if the coroutine is already completed. Then it uses the *CORO_SET_SELF* macro to set the R10 registers and calls the *coro__switch* function to do the context switching to the coroutine.

2. *coro_yield*

   The *coro_yield* function first checks if it has been called from a coroutine, checking the *R10* register, then just calls the *coro__switch* function to do the context switching.

3. *coro_return*

   The return function sets the coroutine as completed and saves the result, and then it just calls the *coro_yield* function to do the context switching to the caller.

Question 5

According to the OpenRISC manual, the register R10 is used for the TLS (Thread Local Storage). This register hosts the address of the thread local storage structure.

In the implementation of this library, it stores the pointer to a *coro_data* structure, holding all the data of the coroutine.

The address is set using the *CORO_SET_SELF* macro and is read using the *CORO_SELF* macro.

Question 6

The *coro__switch* function has to do the context switch between different coroutines. To do so, it has to save all the registers used by the *old* coroutine to the stack, restore the *new* coroutine data to registers, between them the stack pointer and the link register, then do the jump to the *new* coroutine.

Note that, as the library is using a cooperative approach, the context switching happens always when coroutine yields, so it is not necessary to save the temporary registers (*odd* registers). In a preemptive context switching, as it may happen at any point of the execution, all registers have to be saved.

Part 2: Task Manager

Question 7

In the *part2.c* file there is the code of the task manager testing program. Two tasks are defined, *wait_task* and *uart_task*. The first one consists in an infinite loop in which it just waits for a time set by the argument passed to it and then does a printf. The second one is used to interact with the UART bus, reading line by line.

The first task can be spawned multiple times, but the UART can be spawned only once, as only one task can wait for UART input at a time.

So the part2 function spawns 3 *wait_task* with respectively 1s, 3s, and 9s time periods, and then a *uart_task*. The 4 tasks are executed concurrently.

The execution of the scheduled tasks is done in the *taskman_loop* function, which acts as a kind of scheduler but with a cooperative approach, resuming the tasks and regaining control after the tasks yield.

Question 8

The library implements multitasking using a *cooperative* approach, in which every task executes a small chunk of code, and then calls the *yield* function. If a task does not call the *yield* function and keeps running, all other tasks will not be run.

Question 9

The code for the Task Manager has been implemented in the file *src/taskman/taskman.c*.

Question 10

The code for the UART task of the Task Manager has been implemented in the file *src/taskman/uart.c*.