CS476 - PW7 Report

Julien de Castelnau (368509), Magnus Meyer (396302)

December 17, 2024

Part 1: Coroutines

Question 1

1. Concurrency vs parallelism: concurrency is the interleaving of multiple execution streams. Parallelism indicates that mutliple streams are happening at the same time.
2. Cooperative multitasking: Each task has to manually yield to give up execution to other tasks. Preemptive multitasking means that there is some automatic switching between tasks, usually through the use of a timer interrupt.
3. Threads vs coroutines: Coroutines are managed inside the application whereas threads are managed by the OS. The scheduling of coroutines is a context switch within the same program whereas switching to different threads is managed by a priviledged OS kernel, usually by preemption instead of a manual yield as is the case with coroutines.
4. Stackless vs stackful coroutines: Stackful coroutines are allocated their own stack and program state separate from the main coroutine. This allows them to yield in the middle of a deeper call stack. Stackless coroutines do not permit this as they share the stack of the main coroutine and thus the context upon switching would be invalid.

Question 2

We see that the program launches a coroutine, starting in the function *test_fun*. Since this was spawned by the function *part1*, it will not be run instantly. Instead, *part1* will keep executing, printing the first line of format "func = […]". After this, it will resume the execution of the function *test_fn*. This will print the first "func = […]" line with the *coro data*. Following this, it will enter the for-loop and print another of these functions with the _coro/*arg*. After this it will yield and let the main function continue. This will print another line of the format "func = […]". Again, it will allow *test_fn* to continue and enter the "*f*" function, where it will print "func = f, x = 15". After this, it yields again, and the main program prints once more, before allowing *test_fn* to run again. Hereafter, function "*f*" returns with 75, causing it to print "func = test_fn, f(15 + i) = 75". It will then keep looping and write "func = test_fn, i = 1, arg = 0xdeadbeef". This looping continues until the main function has looped 9 times. Hereafter, it will print the result returned from *test_fn*, which is 10.

`part1()`

Function call explanations:

- Line 30: `coro_glinit();` This simply initializes the pointer to the current coroutine, which is held in the Thread Local Storage (TLS) register `r10`, to zero.
- Line 32: `coro_init()`: This function initializes the coroutine, which sets the pointer to the global stack buffer in the struct, and also sets the arguments that will be passed to the coroutine.
- Line 33: `coro_data()`: This function returns the pointer to the start of the data section for that coroutine. The layout of the coroutine's stack is as follows: at the start is the `coro_data` struct itself, containing info such as the caller SP, the coroutine's SP, the return value, etc. Following that is the area which `core_data()` returns a pointer to. The rest of this area is considered the stack area, and the stack grows downward from the end of the buffer as given by `stack_sz` in `coro_init()`. So, the other end of the stack is used to store global data for coroutines.

- Line 37: `coro_resume()`: This switches to a coroutine from the main routine. The argument is the stack buffer of the coroutine.
- Line 41: `coro_completed()`: This checks the completed flag inside the `coro_data` struct and sets the result if the coroutine is complete.

`test_fn()`

- Line 14: `coro_data()`: used to get pointer to data area once again
- Line 18: `coro_arg()`: this uses the pointer to current coroutine, stored in `r10`, to access the `arg` field of the `coro_data` struct set by `coro_init()`
- Line 19: `coro_yield()`: Switch execution back to caller routine.
- Line 24: `coro_return()`: Set the completed flag and return value using `r10` as the current coroutine, then yield.

The program's execution essentially ping-pongs between the main function and the coroutine. The coroutine's loop switches back to the `for (int i = 0; i < 9; ++i)` loop on line 35 after every time it prints `i` and `arg` on line 18. That loop in `part1()` switches back to `test_fn()` using `coro_resume` in turn. Then, `f()` yields after printing the argument `x`. Since `test_fn()` yields twice per loop iteration while `part` only once, this is why the `part1` loop has twice as many iterations.

Question 3

These all make use of `coro_switch`. This function takes the stack pointer so that it can switch to the corresponding context, as well as a pointer to where we can save the stack pointer in the current context.

- `coro_resume` changes context from a non-coroutine to a coroutine. So, it accesses the `coro_sp` field of the passed coroutine stack buffer, and uses that as the new SP. The current stack pointer is saved in the `caller_sp` field of the `coro_data`. There are extra sanity checks reading `CORE_SELF()` to make sure this is not called from within a coroutine, as well as to ensure a coroutine which is already finished is not switched to. Note we need to set the current coroutine field in the global r10 register, and then set it back to NULL whenever we yield back. This is done using `CORO_SET_SELF` in the implementation.
- `coro_yield` is the reverse, so it gets the current `coro_data` using `CORE_SELF()` (reading r10), then switches to the caller_sp saved earlier, storing the current sp in `coro_sp`. This also has a sanity check to make sure it *is* being caled from a coroutine (`r10` should not be null.)
- `coro_return` is `coro_yield` with an extra step to set the completed flag and result fields, according to the arguments passed, of the `coro_data` before yielding.

Question 4

Register r10 stores the pointer to the beginning of the current coroutine's stack buffer. This corresponds to the `coro_data` struct.

Question 5

`coro__switch` is implemented by first saving the current context. Only the callee-saved registers need to be saved: `coro_switch` is a function call from the compiler's perspective, so any caller-saved registers used in the calling context would have been saved according to the OpenRISC calling convention. Thus, this function only saves the callee saved registers. `r10` is not saved because it is set before and after switching by `coro_resume` appropriately. Likewise, `r1` is set when switching itself, so it doesn't need to be saved. It also saves the current stack pointer in the location passed to it, so it can be returned back from in the future. Once the context saving is done, it restores from the stack pointed to by the `sp` provided, and then jumps to it using the link register on that stack.

Part 2.1: Single-core task manager

Question 7:

The purpose of the flag is to specify, whether or not a coroutine is currently being executed. We see that this is initially set to "false" (0), but updated to "1" whenever the coroutine is resumed. This is useful in a multi-core

implementation, as we do not want to execute the same coroutine in two cores at the same time. However, in a single core implementation, we know that when we are in the taskman_loop function no coroutine is running. If they were running, then we would not be able to execute the taskman_loop. Using this fact, we can simply remove the flag when simply treat it is always being set to "false" inside the taskman_loop.

Part 2.2: Dual-core task manager

Question 10:

The lock works by having 256 distinct locks that each can be acquired by a CPU, based on the ID of each CPU. Each lock has a unique memory location. When a CPU wants to require a lock, it provides the ID for the lock (any value between 0-255) and busy-waits, until the value at the memory location of the lock is zero. It uses an *atomic* compare and swap (CAS) operation to do this. When the value at the address equals zero, it will atomically swap this with the ID of the CPU that wants to require the lock. When a CPU wants to release the lock, it simply sets the value of the lock to zero, allowing another CPU to acquire the lock if needed.

Question 11:

For the Taskman implementation, we can identify 2 major race-conditions. Firstly, reading and updating task_data->running flag in a mutli core setting needs to be done inside a critical reading. Otherwise, two cores might run the taskman_loop at the same time, both read that a coroutine is not running and both try and run the coroutine.

Secondly, as mentioned in the project description, all handler functions need to be called from within a critical region. This is because the handlers are managed through a global struct that may be changed by calling these functions.

In both of these cases, they are relevant from within the taskman_loop. One simple way to account for this, is to make everything within the loop a critical region, except for the instruction of *coro_resume*, in which the loop switches to for coroutine for a CPU. However, it may be that two CPU's, both running the taskman loop, try and evaluate two different coroutines that each use two different handlers. In this case, the two loops should be able to run in parallel, as we don't risk overlapping critical regions. Even though this approach is more efficient, it also comes with greater complexity, as the CPUs need to somehow share which coroutine they are working on. When using TASKMAN_LOCK(), we use a single lock across all CPUs. One way to make this implementation would be to give each task a unique lock and each handler a unique lock and acquire those instead of the taskman lock.

When we implement the TASKMAN_LOCK() call before entering the taskman loop, while releasing before each *coro_resume*, re-acquiering just after, we see that the print tasks are called continously and from alternating CPUs. This is the expected behavior, and indicate that the multi core implementation works.

We also need to ensure that the modification of Taskman data structures is synchronized. For spawning and registering new tasks, we simply lock the entire function so that the shared state is updated atomically. Since we can only add tasks and handlers this approach is sufficient to ensure thread safety with respect to other cores that may be running taskman_loop/taskman_wait.

For taskman_wait, we want the on_wait function to be behind a critical section. Otherwise, a task could update the state while the wait happens. The update of the wait handler & arg, and the running flag also needs to be critical. Otherwise, there is a race condition between taskman_loop to read these fields and the updating inside taskman_wait.