BVS1 · Winter Semester 2026/2027 · TH Köln · ICCT

Exercise 5 — The Timer Path

Except where otherwise noted, © 2026 Stavros Kousidis · CC BY-SA 4.0

Planned time: 90 minutes

Guiding Question

> How does the OS get control back from a process that never yields, and how can you show where the process was interrupted?

What You Should Be Able to Do Afterwards

- predict and explain the output of a never-yielding process with and without timer interrupts;
- locate a reported trap `pc` in the disassembly and explain why the OS adds nothing to it;
- name the three conditions for taking a timer interrupt and show where each is set in our system;
- compute a time slice in `mtime` counts and relate it to the number of timer interrupts;
- diagnose a broken timer path from its observable behaviour.

Starting Point — 5 Minutes

You need your **Exercise 4** files `process.h` and `main.c` on your local `exercise-04` branch. Commit any open work there first.

Inside the VM, in your `teaching-bvs1-code` clone:

```
cd ~/teaching-bvs1-code
git status
git fetch origin
git switch exercise/05-timer
git switch -c exercise-05
git checkout exercise-04 -- process.h main.c
```

`git status` must show no uncommitted changes before you switch. Work on your local `exercise-05` branch; do not push student work to the course repository.

The branch provides the Exercise 4 scaffold and five libraries:

```
lib/libkernel.a            Exercise 5: timer on, one line per tick
lib/libkernel-ex04.a       Exercise 4: no timer
lib/broken/libkernel-a.a   broken library A (Task 5)
lib/broken/libkernel-b.a   broken library B (Task 5)
lib/broken/libkernel-c.a   broken library C (Task 5)
```

`make run` uses `lib/libkernel.a`; `make run KERNEL=ex04` uses the Exercise 4 library, `make run KERNEL=a` the broken library A, and so on. QEMU runs with `-icount shift=4,sleep=off` (Lecture 6).

In `main.c`, keep `worker()` and `permute()`; Lecture 7 needs them again. Add this process above `main()`:

```
/* CPU-bound, never yields. */
static void spinner(void)
{
    volatile unsigned long counter = 0;

    uart_puts("spinner: start\n");
    while (counter < 1000000UL) {
        ++counter;
    }
    uart_puts("spinner: done, counter = ");
    uart_put_dec((int64_t)counter);
    uart_puts("\n");
    process_exit(7);
}
```

In `main()`, create the spinner instead of the worker, and add one line that keeps the unused `worker()` in the build without a warning:

```
    (void)worker;   /* kept for Lecture 7 */
    process_create(&p, spinner);
    os_run();
```

If your Exercise 4 files are not ready, work with a partner's copy.

1. Predict, Then Run — 15 Minutes

Before you run anything, write down your prediction for both libraries:

- Does the spinner reach `spinner: done`? Which counter does it print?
- Are there `os:` lines between `spinner: start` and `spinner: done`? Which ones?
- Which lines come from the process, and which from the OS?

Then run both:

```
make
make run KERNEL=ex04
make run
```

Exit QEMU with **Ctrl-a**, release, then **x**.

Evidence: your predictions, both outputs, each line marked *process* or *OS*, and one sentence on what the counter in the last spinner line tells you about the interruptions.

2. Where Was the Spinner Interrupted? — 20 Minutes

Disassemble the spinner:

```
riscv64-unknown-elf-objdump -d --disassemble=spinner
build/hello-bvs1.elf
```

Find the loop: the instructions between the backward branch and its target. For each address in your `os: tick` lines, find the instruction.

1. Has the instruction at a reported address executed when the OS prints the tick line? Justify with the trap rule (Lecture 5, recalled in Lecture 6).
2. In Exercise 4 you traced an `ecall` at address E, and the OS added 4. Why does the OS add nothing after a timer interrupt? What would the spinner do next if it added 4 at one of your reported addresses? Name the instructions that would be skipped.
3. At one reported address, which registers hold values the loop still needs? Are they caller-saved or callee-saved? What would happen to the spinner if the trap path saved only the 14 values of the CP1 context?
4. Count the instructions of one loop iteration. With `-icount shift=4`, one instruction takes 16 ns of virtual time. Estimate how long the loop runs and how many timer interrupts you expect with a 10 ms time slice. Compare with your output.
5. Run `make run` a second time. Are the tick lines identical? Then run `make run ICOUNT=` (no `-icount`) twice. What changes, and why?

Evidence: your tick lines with the matching instructions, five short answers, and your estimate with the calculation.

3. The Three Conditions in Our System — 15 Minutes

A timer interrupt is taken only if it is pending, the timer switch is on, and the global switch is on (Lecture 6). Complete the table. For "set where", point to a line in `riscv.h` or `trap.S`, or to an instruction in the disassembly of `os_run()`:

```
riscv64-unknown-elf-objdump -d --disassemble=os_run build/hello-bvs1.elf
```

Condition	CSR and bit	Set by whom, where
pending		
timer switch on		
global switch on		

Then give the value of each bit while the spinner runs:

Condition	Exercise 4 library	Exercise 5 library
pending		
timer switch on		
global switch on		

Then answer:

1. In the disassembly of `os_run()`, find the load from `mtime`, the store to `mtimecmp` and the instruction that sets the bit in `mie`. Give their addresses. How do you recognise the device addresses in the instructions?
2. While `os_run()` itself runs, can a timer interrupt be taken? Why does the order of the store to `mtimecmp` and the `csrs` not matter here?
3. Every handler run has `mstatus.MIE` = 0. Suppose the time slice expires while the OS handles the spinner's `exit` system call. What happens to that timer interrupt? Why does the same not happen while the OS prints a tick line?

Evidence: both tables and three short answers.

4. The Time Slice — 5 Minutes

1. `mtime` counts at 10 MHz. How many counts are 10 ms?
2. Find this number in the disassembly of `os_run()`. In which instructions, and in which form?
3. `timer_arm(delta)` in `riscv.h` writes `mtime + delta` into `mtimecmp`. What would go wrong if the OS wrote only `delta`, for example 100000?

Evidence: three short answers.

5. Diagnose Three Broken Libraries — 20 Minutes

Each of the libraries A, B and C has exactly one defect in its timer path. Run your spinner against each:

```
make run KERNEL=a
make run KERNEL=b
make run KERNEL=c
```

For each library:

- describe the observable behaviour (tick lines, their addresses, the counter, the end of the run);
- name the defect: which condition or step of the timer path is wrong;
- give the evidence: which line of the output, which instruction in your `objdump`, which condition of Task 3;
- state the correction.

For the library that makes the spinner skip instructions, the exact outcome depends on where the ticks land: explain, with your disassembly, which instructions were skipped at each tick and what the CPU found at the address where the spinner continued.

Evidence: three diagnoses, each with behaviour, defect, evidence and correction.

Exit Question — 5 Minutes

In four to six sentences, explain:

> What happens between `mtime` reaching `mtimecmp` and the spinner executing its next instruction?

Use **pending**, `mie.MTIE`, `mstatus.MIE`, **trap**, `mepc`, **re-arm** and `mret`. Then complete the sentence: *Preemption means that …, and in today's OS the scheduling decision is trivial because …*