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

Exercise 3 — Tracing Code and Runtime State

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

Planned time: 90 minutes

Guiding Question

> Where are our instructions and runtime data, and how does restoring a context continue the same execution?

What You Should Be Able to Do Afterwards

- locate editing, building, and execution in the physical-host / Ubuntu-VM / RV64-target setup;
- make and verify a saved source change through the complete edit–build–run workflow;
- distinguish instructions, immediate constants, constant/static data, stack storage, and saved CPU context in the Exercise 2 example;
- trace and diagnose the same execution across yield, resume, and exit, including how the program counter is re-established.

Starting Point — 5 Minutes

Use your existing **Exercise 2** work in `~/teaching-bvs1-code`, including `worker()`, `permute()`, and the provided process library. Keep your existing local branch and changes. This exercise adds no process implementation or new checkpoint mechanism.

If that application is not ready, work with a partner's running version for Task 2 and ELF inspection. The reference trace and observation card below let you complete the reasoning tasks without first finishing Exercise 2. Continue your own implementation during lab support time.

On the physical host, enter the VM:

```
multipass shell bvs1
```

Inside the VM:

```
cd ~/teaching-bvs1-code
pwd
ls
```

Check that `ls` lists the `Makefile`. This is the directory in which you run `make`.

1. Draw the Working Environment — 10 Minutes

Draw the physical host, Ubuntu VM `bvs1`, QEMU process, and emulated RV64 target. Label the relationships **hosts via virtualization**, **runs**, and **emulates**.

Place these actions or artefacts in the drawing:

```
editor       main.c on disk       compiler/linker
linked ELF   make                 target CPU and RAM
worker() executing               UART output
```

Add arrows showing how a saved change in `main.c` reaches execution on the RV64 target.

Evidence: your annotated diagram and one sentence explaining why running QEMU in Ubuntu does not mean that `worker()` is compiled for Ubuntu's CPU architecture.

2. Edit, Save, Build, Observe — 10 Minutes

Predict what changes if this prefix in `permute()`:

```
uart_puts("process: before yield: ");
```

becomes:

```
uart_puts("process: pause value: ");
```

Open `main.c` in your usual editor. With nano:

```
nano main.c
```

Save with **Ctrl-O**, **Enter**; leave with **Ctrl-X**. Then:

```
git diff -- main.c
make
make run
```

Exit QEMU with **Ctrl-a**, release, then **x**. Restore the original output prefix in the editor, save, and rebuild before continuing.

Evidence: the saved source diff, the changed output line, and an explanation of why the array values remain the same. If the old label appears, identify the first failed link in **saved source → build input → successful build → executable → QEMU run**.

3. Connect the Source to Target Memory — 20 Minutes

Use these commands inside the VM after a successful build, or use the observation card below:

```
riscv64-unknown-elf-objdump -h build/hello-bvs1.elf
riscv64-unknown-elf-nm -n build/hello-bvs1.elf
riscv64-unknown-elf-objdump -d \
  --disassemble=context_switch build/hello-bvs1.elf
```

```
riscv64-unknown-elf-objdump -s -j .rodata build/hello-bvs1.elf
```

`objdump -h` lists sections; `nm -n` lists symbols by address; `objdump -d` decodes instructions; `objdump -s -j .rodata` shows the bytes stored in `.rodata`. In `nm`, `T/t` denotes code and `B/b` denotes BSS-related symbols. A symbol may name an address, not an allocated C object.

Classify each item and justify your choice:

Item	Location / role
instructions of `worker()` / `permute()`	
constant output label `"process: result: "`	
local `char value[] = "BVS1"` in `worker()`	
`struct process p` local to `main()`	
embedded array `p.stack`	
saved register fields in `p.context`	
current CPU registers `pc` and `sp`	
`_stack_top`	

Then answer:

1. Is the `.data` section of the inspected build empty? If so, does that indicate an error?
2. Does `nm` have to list the local array `value` or a separate `permute()` symbol? Explain the difference between an automatic object and a function the compiler may inline.
3. Do these file-inspection commands show the live contents of the process stack?
4. The compiler can initialize `value` using immediate constants in instructions. Does this make the writable array itself part of `.text` or `.rodata`?
5. Find one constant encoded as an immediate in the `context_switch` disassembly, and find the bytes of one output label in `.rodata`. Which of the two is part of an instruction, and which is data that instructions must address?

Evidence: the completed table plus one section or symbol observation from the ELF or the card. Sketch `p.stack` inside `p`, and show where `p` is allocated in this example.

Observation Card — One Exercise 2 Build

These are selected observations from an instructor build, not addresses or byte counts you must reproduce:

```
section     role / observed contents
.text       target instructions
.rodata     constant output labels
.bss        os_context, current_process
.data       empty in this build
```

```
instruction       observation
sd s0,16(a0)      offset 16 is an immediate in the instruction

symbol            nm type
context_switch    T
worker            t
os_context        b
current_process   b
_stack_top        B  (linker-defined initial stack address)
```

For the label lookup, these are the bytes of `"process: result: "`, including its terminating zero, in `.rodata` (hexadecimal, in increasing address order; absolute addresses omitted):

```
70 72 6f 63 65 73 73 3a 20 72 65 73 75 6c 74 3a 20 00
 p  r  o  c  e  s  s  :     r  e  s  u  l  t  :     \0
```

`objdump -s` prints the same bytes in a different layout: with an address at the start of each line, in groups of four bytes, and with an ASCII column on the right. A label can start in the middle of a group or line.

The normal `-O2` build folds `permute()` into `worker()`. The lecture's observation build retains the function boundary using `-O0 -g -fno-inline`. Neither result requires function instructions to be placed on the stack. Frame pictures describe runtime roles rather than fixed compiler offsets.

4. Trace One Array Across the Pause — 25 Minutes

Use the Exercise 2 application. Its essential operations are:

```c
/* worker() */
char value[] = "BVS1";
permute(value);
/* print the final value, then process_exit(0) */

/* permute(value): the parameter points to that same array */
/* swap positions 0 and 1; print; process_yield(); */
/* swap positions 2 and 3; return to worker(); */
```

Complete the snapshots. Include the array's terminating zero byte.

Moment	Array contents	Application calls still active	Execution currently using CPU
A: `worker()` initialized the array			
B: first swap done, before entering yield			

Moment	Array contents	Application calls still active	Execution currently using CPU
C: yield returned control to the OS			
D: resumed process completed the second swap, before helper return			
E: the process has exited via `process_exit(0)`			

For snapshot C, draw three separate boxes:

1. current CPU registers, with `pc` and `sp` used by the OS;
2. the preserved process stack, including the array;
3. `p.context`, with its saved stack position, continuation, and additional registers.

Add arrows from the saved context to the process stack and the continuation in code. Show `.text` as the location of the instructions for both OS and process execution.

Explain why:

- passing `value` to `permute()` does not create another array or another process;
- restoring the context first continues the suspended yield path and then returns into the same helper call;
- `p.context` has no `pc` field, yet the process continues at the right instruction;
- `value` is not initialized to `BVS1` again on resume;
- FINISHED prevents another run even if old stack bytes remain.

Evidence: the completed snapshots, your three-box diagram, and a short explanation connecting them.

5. Diagnose and Transfer — 15 Minutes

For each case, state the wrong assumption, predict a consequence, and explain a correction:

1. After yield, `process_run()` calls `p.entry()` again instead of restoring the process context.
2. A switch restores the continuation but leaves `sp` selecting the OS stack.
3. A student sees `worker frame` in a diagram and concludes that entering `worker()` copies its instructions onto the stack.
4. The process reaches `for (;;) {}` before its yield, and the student expects our current OS-side code to resume automatically.

For case 4, distinguish the **RV64 target's OS control flow** from Ubuntu's ability to manage the QEMU process. Name the kind of future mechanism that could allow the target OS to regain control, and explain why it must preserve the interrupted `pc` explicitly although our cooperative context has no `pc` field. No interrupt implementation is required.

Evidence: four brief diagnoses with predicted behaviour, not just true/false answers.

Exit Question — 5 Minutes

In four to six sentences, explain:

> At the yield point, which instructions and data exist where, what does the OS preserve, and why does the next run continue the same execution?

Use `.text`, **process stack**, **saved context**, and **lifecycle state**. Then point to the file you edited and the machine that executed the resulting RV64 instructions.