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

Checkpoint 1 — I Have a Process

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

Mandatory individual demonstration

Goal

Implement the C-level process abstraction from Lecture 3.

Exercise 2 used your public process interface together with the provided `libprocess.a`. For Checkpoint 1, that binary implementation is gone.

The low-level RV64 context switch remains provided.

Your goal is therefore:

```
same Exercise 2 application
          +
your process implementation
          +
provided context_switch()
          ↓
same observable behaviour
```

1. Get the Checkpoint Branch

First make sure your Exercise 2 work is committed on your local `exercise-02` branch.

Then fetch the current course branches and create a local CP1 work branch from the checkpoint scaffold:

```
git fetch origin
git switch lab/cp1-process-api
git switch -c cp1
```

The CP1 scaffold deliberately does not contain your Exercise 2 files. Bring your own Exercise 2 interface and application onto the new branch:

```
git checkout exercise-02 -- process.h main.c
```

You should now have:

- the CP1 scaffold from `lab/cp1-process-api`;
- your own `process.h` from Exercise 2;
- your own permutation application in `main.c`.

Do not push student work to the course repository.

2. Provided Infrastructure

The CP1 scaffold provides:

- startup and linker support;
- UART string output support;
- `context.h`;
- `context.S`;
- the RV64 `context_switch(current_context, next_context)`;
- Make/QEMU integration.

It deliberately does **not** provide:

- `process.h`;
- `process.c`;
- `libprocess.a`;
- an implementation of the process API.

You are not required to write the RV64 context-switch assembly.

3. Keep Your Exercise 2 Interface

Reuse the `process.h` that you created in Exercise 2.

It must still represent the four pieces introduced in Lecture 3:

- entry function;
- owned stack storage;
- saved CPU context;
- lifecycle state and exit status.

It must still expose the four operations used by your Exercise 2 application:

- `process_create`;
- `process_run`;
- `process_yield`;
- `process_exit`.

For CP1, create `process.c` and implement that interface yourself.

4. Required Semantics

`process_create()`

Prepare a process without running it.

After creation:

- the entry function is known;
- the process owns its stack storage;
- the initial stack pointer is correctly aligned near the high-address end of that stack;
- an initial CPU context exists that can enter the process for the first time;
- the process is `READY`.

A process that has never run must be made to look, to `context_switch()`, like an execution that can be restored.

`process_run()`

Give the CPU to a runnable process.

It must:

- refuse to run a `FINISHED` process;
- record which process is `current_process`;
- mark it `RUNNING`;
- save the OS context and restore the process context.

The same operation must work both for first start and for later resume.

`process_yield()`

The currently running process voluntarily returns control to the OS.

It must:

- mark `current_process` as `READY`;
- save the process CPU context;
- restore the OS CPU context.

When the process is run again, execution must continue after the suspended yield rather than restart its entry function.

`process_exit(status)`

Finish the currently running process.

It must:

- record the exit status;
- mark `current_process` as `FINISHED`;
- return control to the OS through the context switch.

A finished process must not run again.

First entry

A newly created process has never yielded, so it has no naturally saved continuation yet.

Your implementation therefore needs an internal first-entry path that connects the prepared initial context to the process entry function. If the entry function returns normally, treat that as a normal exit with status 0.

The internal helper is implementation machinery; it is not another public process operation.

5. Reuse Your Exercise 2 Application

Reuse the permutation application from Exercise 2 unchanged.

Its observable behaviour should still be:

```
process: before yield: VBS1
os: process yielded
process: result: VB1S
os: process exited
```

This is the important test: the application should not care whether its process API is backed by the provided Exercise 2 library or by your own CP1 implementation.

Build and run with:

```
make clean
make
make run
```

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

6. What You Must Be Able to Explain

At the checkpoint, each student must individually explain their own implementation.

Be prepared to explain:

- why an entry function is not itself a process;
- why the process needs its own stack memory;
- what our CPU context contains;
- why the context is not the stack;
- what `sp` reconnects the CPU to;
- what role the restored `ra` plays;
- why `s0-s11` are preserved by the provided switch;
- how a process that has never run gets an initial context;
- what `current_process` records and why `yield` and `exit` need it;
- why first start and later resume can use the same `context_switch()`;
- why resume continues the same C call chain instead of invoking the entry function again;
- why yield and exit use the same low-level switch but have different lifecycle meaning;
- why a `FINISHED` process must not be run again.

You may also be asked to predict an execution, diagnose a fault, or make a small equivalent modification inside this announced competence family.

7. A Useful Construction Order

A productive order is:

```
1. bring your Exercise 2 process.h and application onto the CP1 branch
2. create OS-context and current-process bookkeeping
3. prepare a new process stack and initial CPU context
4. implement process_run()
5. implement process_yield()
6. prove that the permutation helper resumes
```

```
7. implement process_exit()
8. handle normal return from the entry function
9. reject attempts to run a FINISHED process
10. build and run from a clean state
```

This is guidance, not an assessed source-file structure.

Not Required

Do not add these mechanisms for CP1:

- timer-driven preemption;
- interrupts or trap handling;
- several runnable processes;
- a scheduler;
- privilege separation;
- virtual memory;
- dynamic executable loading;
- dynamic memory allocation;
- a filesystem;
- a shell.

These belong to later stages of the course.

Demonstration

Bring your local semester repository in a clean, buildable state.

At the appointment:

1. build and run your implementation;
2. demonstrate the complete permutation / yield / resume / exit behaviour;
3. identify the code representing entry, stack, context, and lifecycle state;
4. explain your process API and the provided assembly boundary;
5. respond individually to a small equivalent question, diagnosis, prediction, or modification.

Pair work is allowed during construction, but the checkpoint assessment is individual.

Checkpoint Dates

Group	Checkpoint 1
M1	Wed **04 Nov 2026**
M2	Wed **18 Nov 2026**
F1	Fri **06 Nov 2026**
F2	Fri **13 Nov 2026**

Only your booked checkpoint appointment is mandatory. Weekly lab sessions remain available as optional support time.

Before Your Appointment

Verify that:

- the repository builds from a clean state;
- the process demonstrably resumes rather than restarts;
- the permutation progresses from `BVS1` to `VBS1`, yields, and later reaches `VB1S`;
- a finished process cannot be resumed;
- you can identify entry, stack, context, and lifecycle state in your own code;
- you can explain why the same context switch works in both directions;
- your local Git history contains a working CP1 milestone;
- you have a backup outside the course VM.