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

Exercise 2 — Using Your First Process

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

Planned time: 90 minutes

Goal

Use the process abstraction from Lecture 3 before you implement it yourself for Checkpoint 1.

The course-code branch for this exercise provides:

```
lib/libprocess.a   precompiled implementation of the process API
context.h          saved CPU-context representation
context.S          visible RV64 context_switch()
```

There is deliberately **no `process.h` and no C source implementation of the process API**. You will create the public header yourself and link it against the provided implementation.

Your task is to write one process whose ordinary C execution:

```
starts
  ↓
works on a local character array
  ↓
permutes part of the array
  ↓
yields while a helper function is active
  ↓
resumes the same helper call
  ↓
finishes the permutation
  ↓
exits
```

The starter already contains `uart_puts()`. No additional output helper is required.

1. Get the Exercise Branch

Inside your existing `teaching-bvs1-code` clone:

```
git fetch origin
git switch exercise/02-process-use
git switch -c exercise-02
```

Work on your local `exercise-02` branch. Do not push student work to the course repository.

The course branch contains the Exercise 2 scaffold only. Your local branch is where you create `process.h` and the exercise application.

2. Inspect the Provided Boundary

Open:

```
context.h
context.S
Makefile
```

Answer:

1. In `context.h`, which CPU registers are represented by `struct context`?
2. In `context.S`, which registers are saved from the current execution and restored for the next execution?
3. In the `Makefile`, which variable names the provided process library, and where is that library added to the final linker command?
4. Which two process source/interface files are deliberately absent from the branch?

The purpose is to identify the boundary:

```
provided:     context representation + context switch + process binary
your work:    process.h + Exercise 2 application
```

3. Create the Process Interface

Create `process.h`.

Lecture 3 fixes the process representation and the public API used by the precompiled library. Recreate that interface:

- include `<stdint.h>`;
- define `STACK_SIZE` as 4096;
- define `process_entry_t` as a pointer to a function taking no arguments and returning `void`;
- define lifecycle states `READY`, `RUNNING`, and `FINISHED`;
- define `struct process` in this field order:
  1. entry function;
  2. `uint8_t stack[STACK_SIZE]`;
  3. `struct context context`;
  4. lifecycle state;
  5. integer exit status;
- declare:
  - `process_create(struct process *process, process_entry_t entry)`;
  - `process_run(struct process *process)`;
  - `process_yield(void)`;
  - `process_exit(int status)`.

The field order matters because `lib/libprocess.a` is already compiled against this representation.

Now build the unchanged scaffold once:

```
make
```

The provided `Makefile` automatically compiles the C files at the repository root and links `lib/libprocess.a` into the final ELF. You do not need to add the library manually or modify the linker command.

After you begin using `process.h` from `main.c`, continue to build with the same `make` command.

4. Predict the Permutation

Start with this local character array:

```
char value[] = "BVS1";
```

Your helper will perform two swaps:

```
first:
    swap value[0] and value[1]

yield

second:
    swap value[2] and value[3]
```

Before writing code, complete:

```
initial value          = BVS1
value before yield     = ____
final value            = ____
```

Do not run the program yet. Predict first.

5. Write the Helper That Yields

In `main.c`, include:

```
#include "process.h"
```

Write a helper with this shape:

```
static void permute(char value[])
{
    char tmp;

    /* swap value[0] and value[1] */

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

```
    uart_puts("\n");

    process_yield();

    /* swap value[2] and value[3] */
}
```

Important:

- `value` belongs to the active process execution;
- `tmp` is a local variable of `permute()`;
- when `process_yield()` is called, `permute()` has not returned.

Before running the program, predict:

> After the process is resumed, where should execution continue inside `permute()`?

6. Write the Process Entry Function

Write an entry function that owns the character array:

```
static void worker(void)
{
    char value[] = "BVS1";

    /* call permute(value) */

    /* print the final value */

    /* exit */
}
```

The array is deliberately local to `worker()`.

At the yield point, both `worker()` and `permute()` are still active. The process must later continue that same C call chain.

Use `process_exit(0)` when the work is complete.

7. Create and Run the Process

In `main()`, create one process:

```
struct process p;
```

Then:

1. call `process_create(&p, worker)`;
2. call `process_run(&p)` once;
3. confirm that control returns to the OS because the process yielded;
4. print `os: process yielded` only if the process state is `READY`;
5. call `process_run(&p)` a second time;
6. confirm that the process finishes;

7. print `os: process exited` only if the process state is `FINISHED`.

Your observable output should be:

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

Do not print these lines unconditionally. Use the actual process state to determine the OS-side messages.

8. Run It

Build and run:

```
make clean
make
make run
```

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

Compare the observed result with the permutation you predicted before running the program.

9. Explain What Happened

Use this control-flow diagram:

```
OS
 |
 | run
 v
worker -> permute -> yield
                       |
                       v
                      OS
                       |
                       | run again
                       v
            same permute call -> worker -> exit
                                                |
                                                v
                                               OS
```

At the yield point, answer:

1. Which stack is active while `worker()` and `permute()` are running?
2. Which application calls are still active when `permute()` yields?
3. Where does the local array `value` live while those calls are active?
4. What does the saved process `sp` reconnect the CPU to?
5. What role does the saved process `ra` play when the process is resumed?
6. Why is preserving the stack alone not enough to resume the same execution?

7. Why does the second `process_run(&p)` continue the suspended execution instead of starting `worker()` again?
8. Why can the same `context_switch()` mechanism transfer control both from OS to process and from process to OS?

Exit Question

In at most four sentences:

> What turns `worker()` from an ordinary C function into a resumable, OS-managed execution?

Connection to Checkpoint 1

For this exercise, `lib/libprocess.a` implements the process API for you.

For Checkpoint 1, that library is removed. You keep the same public interface and implement the process abstraction yourself around the provided `context_switch()`.