Otto-Friedrich-Universität Bamberg
Fakultät WIAI

Lehrstuhl für Praktische Informatik, insbes.
Systemnahe Programmierung – Prof. Dr. Michael Engel
https://www.uni-bamberg.de/sysnap
michael.engel@uni-bamberg.de

Exercise 3

The stack and cooperative multitasking

Discussion on Monday, 21.6.2022

The code skeleton for this exercise can be found at
https://github.com/michaelengel/OSE2022/tree/main/lecture5/multitasking

3.1 Loadable Processes

Based on the process swapping code in the git repository, add the following functionality (check the `TODO 3.1` and `TODO 3.2` comments in the code):

- Extend the exception handler in `kernel.c` to distinguish between system calls and other sources. Print an error message indicating the values of the `mcause`, `mepc` and `mtval` CSRr. Refer to sections 3.1.14–3.1.16 in the RISC-V Instruction Set Manual, Volume II: Privileged Architecture (Document Version 20211203 at https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf) for details.
- Change the PMP configuration in setup.c so that user mode programs can no longer access the I/O address range (0x0000_0000–0x7FFF_FFFF) or the kernel address range (0x8000_0000–0x800F_FFFF). Check that the protection works: accessing addresses in the I/O and kernel space from a user program should now result in an exception message from your exception handler.
- Fix the problem with stack — create a separate user stack for each of the user mode programs and set the stack pointer accordingly at the start of the user program in `userentry.S`.

3.2 Cooperative Multitasking

You are going to introduce a new system call, `yield`, with syscall number 23. Using this system call, a user program gives up control of the processor voluntarily. The kernel eventually returns to the user program, so that it now can be continued *where it left off*, i.e. at the instruction following the system call.

- We now need to ensure that all user programs are in memory at the same time. So we have to give up our process swapping again and now configure the code to load the first program at address 0x8010_0000, the second at 0x8020_0000, and so on. To do this, you need separate linker scripts for each program.
- Now implement the `yield` system call. We already save registers in the assembler stub function in `ex.S`. What you need to do here is:
  - Create a data structure that holds the PC and SP registers of each program (or process, to be precise – we will discuss the difference in the next lecture).
  - Save the PC and SP registers when the `yield` system call is executed
  - Switch to a different user program (e.g. by implementing a simple *round robin* scheduler).
  - Restore PC and SP when control returns to a process after a `yield` call.

Remember that the initial values for each user program's PC and SP have to be initialized accordingly, so the first start of each program (not from `yield`) should be handled as a special case.