Lab 3 Precise Trap Handling with System Call

> Due: 2023/12/18 (Mon) 23:59:59

This lab takes 2 points. After previous labs, you should be familiar with the simulator now. You might notice that it lacks a trap mechanism from user space to kernel space. For this lab, you are asked to add a **precise** trap handling mechanism by implementing a system call (syscall) to the simulator.

Please refer to following instructions for your implementation.

1. Kernel Code

In the previous labs, you have learned how the emulator loads and runs programs. You should notice that it has no operating system. So there is no kernel-space code in it. Nevertheless, you can write a piece of assembly code as *kernel code* as the handler for a syscall.

In this lab, you are responsible for simulating a system call which

1. stores/loads integer data in a contiguous array (with no less than 8 integers randomly placed) at a particular memory address and,
2. finds out and returns the largest integer of the array.

Your *kernel code* snippets need to be made as a binary format and executed by the pipeline of simulator. The arguments are *given* by passing arguments to `ecall` from user-space code, which should follow the calling convention of RISC-V.

> **HINT**:
>
> After you write the *kernel code* as a trap handler, you may want to put it in a defined memory address space. In the current simulator, the 32-bit address space is used. The stack address starts at `0x80000000`, where the stack will expand at the direction of decreasing address. So, you can consider using the address space `0x80000000-0xFFFFFFFF`.

2. Trap handler

When doing with the aforementioned handler, you need to cover following items.

0. Use the `ecall` as a trigger for the syscall. Pick an `ecall` operand that, when triggered, saves the current program's state and jumps to the kernel code. Restore the previous state after completing the kernel code.
1. The program state can be saved in one of the *known* manners. Typically the program's state is stored in a dedicated location, but on some systems it can also be stored on a stack.
2. Make sure that all instructions before the instruction that triggers the syscall have successfully executed and committed.
3. Make sure that no instruction beyond the instruction that triggers the syscall is factually committed.

4. You should track and save the state before executing the instruction that triggers the syscall. You jump to the kernel-space handler and return to continue running your original user-space codes after handling the syscall. For this project, the mark of returning from trap handler should be `ret`.
5. Basically, you need to implement and run all tests without out-of-order execution (scoreboard).
6. (Optional) One bonus point would be given to students who implement and run tests with out-of-order execution (scoreboard).

> **HINT**:
> Note that there are incoming and outgoing values for the `ecall` directive, which should be additionally considered when saving/restoring state.

3. Test

You need to complete your testing process according to your design and show meaningful test(s).

4. Report

The following sections must be included in your report.

1. Design and implementation details on how you make the precise trap handling mechanism for the syscall and your test(s) with test case(s).
2. How your test(s) illustrate the correctness of your implementation.

All rules enforced to previous labs are applicable to this lab.