CSE 536: Advanced Operating Systems

Assignment 4: Trap and Emulate Virtualization

Deadline: 11:59 PM on Dec. 6, 2024

(100 points)

1 Brief Description

In this assignment, you will implement one way to design a virtual machine—*trap and emulate*. The high-level idea of this approach is that the virtual machine (VM) runs as a user-mode process on the host operating system, which also acts like a virtual machine monitor (VMM). The VM is allowed to execute user-level (unprivileged) instructions directly, while all supervisor-level (privileged) instructions *trap* to the VMM. It then *emulates* these trapped privileged instructions and maintains (in-memory) all privileged state of the VM. This state includes all privileged registers (e.g., satp, scause, stvec, etc. in RISC-V).

GitHub Repo. Please clone as follows:

git clone -b lab4-trap-and-emulate https://github.com/ASTERISC-Teaching/cse536-release.git

Important note

- **If your assignment does not compile or your upload is corrupted, you will get a zero.** Always double-check your submission.
- **Submit a zip file titled by your ASU username**. For instance, if your username is adil, your file should be adil.zip.

2 Description of Tasks

Please follow the steps below to complete this task:

1: Privileged Virtual Machine State Initialization (10 points). The VMM must keep in-memory instances of the VM's privileged state. Hence, your first task is to initialize and maintain a data structure that keeps the following privileged registers and VM state:

- Machine trap handling registers
- Machine setup trap registers
- Machine information state registers
- Machine physical memory protection registers
- Supervisor page table register (satp)
- Supervisor trap setup registers
- User trap handling registers
- User trap setup registers
- Current execution mode (e.g., U-mode, S-mode, M-mode).

Please initialize all the registers to 0, except for the `mvendorid` register, which you should initialize to the hexadecimal code of "cse536". Also, your VM should boot at Machine-mode (M-mode). Define different VM execution modes with different codes decided by you.

Suggested reading(s)

Please check this website to complete task 1: RISC-V Privileged Register Mappings

2. Track Virtual Machine Execution and Redirect Traps (5 points). All VM processes will start with "vm-". Hence, your job is to track whether the process is designated as a VM. If it is a VM, the VMM should redirect all user traps that are raised for the execution of privileged instructions to our function `trap_and_emulate()` (defined in trap-and-emulate.c). The test cases will be centered around 5 privileged instructions: `csrr`, `csrw`, `sret`, `mret`, and `ecall`. However, note that `ecall` will not be trapped as an error in `usertrap()` function. Make sure you handle the traps for `ecall` correctly too, and redirect its execution.

3. Decode Trapped Privileged Instruction (20 points). Once the VM executes a privileged instruction, it will be trapped to the VMM. In this task, your job is to *decode* the instruction so that your VMM can subsequently emulate it. For the decoding process, the VMM must read the instruction and determine (a) the opcode of the instruction (op), (b) source registers (src1 and src2), and (c) destination register (dst). One of the important parts of this decoding is to map between the codes of registers and the actual source and destination registers. Check the suggested reading #1 for this information.

Suggested reading(s)

Check this link for the layout of RISC-V instructions: RISC-V Instruction Layout

4. Emulate Decoded Instructions (40 points). Once you have decoded the privileged instruction executed by the VM, you must perform its operation in software. Provided below is a simple (and incomplete) description of what each instruction does:

- `csrr`: Moves the value from a privileged register to an unprivileged register.
- `csrw`: Moves the value from an unprivileged register to a privileged register.
- `sret`: Transfers control from S-mode to U-mode entry point.
- `mret`: Transfers control from M-mode to S-mode entry point.
- `ecall`: Transfers control from U-mode to S-mode or M-mode.
  Also, transfers control from S-mode to M-mode.

Note that all the above mentioned instructions have the same opcode. However, you can differentiate between each instruction by looking at the src and dst registers. Provided below are some important conditions you should take care of. The list is non-exhaustive; please refer the RISC-V manual.

- An instruction/register should only be accessible in its correct mode inside the VM (e.g., `scause` should not be read in U-mode). If the user process (in the VM) tries to execute a privileged instruction, the VMM should redirect execution to the guest OS' trap handler.
- Execution of `mret` and `sret` requires that certain registers are set before. For instance, returning from M-mode to S-mode using `mret`, the system writes certain bits of the `mstatus` register, indicating a return to S-mode (check kernel/start.c). If the corresponding registers are not correctly set, the execution of these instructions should fail, and the VM should be killed.
- Recall that user and kernel traps are redirected to addresses specified by registers like `stvec`, etc. If you do not remember this, check lectures where we talked about trap handling in xv6.
- When the VM writes 0x0 to `mvendorid`, you should shutdown the VM by killing the process. This is of course, not an actual condition, but one we implement to ensure graceful VM shutdown.

5. Emulating Physical Memory Protection (PMP) (25 points). There is no paging inside our trap and emulated VMs—all software inside the VM assume that they run directly on physical memory. However, the M-mode software inside the VM can still enable memory protections using PMP. In particular, if the M-mode software writes to the `pmpaddr0` register (e.g., using the instruction `csrw pmpaddr0, s3`) it assumes that part of its physical memory will become inaccessible in U/S-modes (recall assignment #1). Your task is to emulate this for the VM using xv6's process page tables.

The VMM must perform three sub-tasks to achieve PMP emulation. First, the VMM (xv6) should create a copy of the VM process' page tables for U/S-mode execution. Second, on the newly-created page tables (which we will call *PMP tables*), the VMM should unmap the regions of memory that are inaccessible (based on the PMP registers). Third, whenever the VM transitions into U/S-mode (e.g., using `sret`), the VMM should switch the VM process' page tables to PMP tables.

Note that for a created VM, the VMM automatically provides a 4MB memory region from 0x80000000 - 0x80400000. For the purpose of this assignment, assume that this is the only memory region where PMP will be leveraged for protection. For instance, in a testcase (which you will be provided later), you will be asked to remove permissions for the upper 1MB memory region.

Testcases. You are provided a test VM for this assignment (namely `vm-test`). You can change this VM to test different conditions in your code. The output of this VM is provided in outputs/vm-test.

3 Miscellaneous

3.1 Clarifications/Simplifications

Curated below is a list of clarifications:

- In the `mstatus` register, you only need to track the MPP (machine previous privilege) field (ignore all others). Based on this field, your system should return to U/S-mode.
- In the `sstatus` register, you only need to track the SPP (supervisor previous privilege) field (ignore all others). Based on this field, your system should return to U-mode or remain in S-mode.
- You can assume M-mode interrupt/exception delegation registers will always be set to 0xffff (i.e., all traps are delegated to S-mode).
- You can allow reading of the `mvendorid` register in all modes. Writing to `mvendorid` should only be allowed in M/S-mode.
- For PMP, you will only be asked to remove permissions in a top-of-range (TOR) manner. (Check the example given above and in assignment 1).
- In all testcases, you can assume that all writes to privileged registers will be valid.
- Your system should be able to access the `sie` register, but you can ignore it's features (i.e., it does nothing).
- If xv6 does not do anything special with a privileged register, our testcases will also not do anything with that register except for read/write from it.

3.2 Submitting your Assignment

Please zip the entire provided code directory and submit it to the Canvas under "Assignment 4: CODE".