- Goal: run some instructions inside a VM sandbox. Not required to be a full VM!
- KVM is abstraction over different architectures but still lets you view / modify registers, setup memory layout etc.
- Need different program for different architecture.
- Focus on x86

Instructions:

- Add the initial contents of the `al` and `bl` registers (which we will pre-initialize to 2),
- convert the resulting sum (4) to ASCII by adding '0',
- output it to a serial port at `0x3f8`
- followed by a newline, and then
- halt.

Requirements:

- setup VM
- load the code into guest memory.
- Emulate a simple IO device: a serial port.
- run vm

Steps

- open /dev/kvm virtual file to start interacting with KVM
- check version
- check for the capability of setting up user memory `KVM_CAP_USER_MEM`
- Create a VM by making an ioctl call. Returns another file descriptor
- Ready to setup memory. This is guest physical memory. If we don't setup the memory, guest will exit.
- Allocate a page.
- Copy the assembly code into the page.
- Map page to guest physical address 0x1000. 0x0000 typically stores interrupt descriptor table: skip first page
- The slot field provides an integer index identifying each region of memory we hand to KVM; calling KVM_SET_USER_MEMORY_REGION again with the same slot will replace this mapping, while calling it with a new slot will create a separate mapping. You can create multiple memory mappings for setting up different flags. VM exit on some mappings (for setting up memory mapped IO).
- Create a vcpu.
- Each vcpu has CPU states: processor registers and other states. KVM allows view / modifying them too.

- Each virtual CPU has an associated struct kvm_run data structure, used to communicate information about the CPU between the kernel and user space. In particular, whenever hardware virtualization stops (called a "vmexit"), such as to emulate some virtual hardware, the kvm_run structure will contain information about why it stopped. We map this structure into user space using mmap(),
- Learn how much memory we need to allocate for kvm_run struct.
- mmap it to userspace. So userspace can access it directly.

Now need to setup registers: * standard registers * special registers. By default code segment, eip points to the place where intel starts reading the instructions after a reboot: 16 bytes below top of memory. We set cs.base to 0 and eip to 0x1000.

We also set up eax and ebx to 2 and 2. Eflags is set to 2. This means all flags are clear. (show eflags section 3.4.3, vol 1).

Now, after we are properly setup, we are ready to start the VM and start running instructions using KVM_RUN ioctl(). If you want to run multiple-CPUs, we must initialize mutliple vCPUs, and KVM_RUN them all on separate userspace threads.

ioctl(.. KVM_RUN ..) is a blocking call. It causes the atomic world switch. It does not return until vm has a reason to exit.

The `kvm_run` structure that we mmaped earlier contains the exit reason. In general there can be several dozen reasons to exit, here we will handle only a few of them.

- **KVM_EXIT_HLT**: Program has ended. Just finish.
- **KVM_EXIT_IO**: Program tried to do IO. Again kvm_run structure has information to further understand the exit reason. It is trying to output one character on port 0x3f8. Size is "the number of bytes in each output": 1 byte, 2 bytes, or 4 bytes. `count` is number of outputs.

The data offset is also offset from the start of kvm_run structure. Since, the size is 1 byte, we cast run pointer to char* and then add the data_offset. We dereference it to get the character and then print it on console.

- KVM_EXIT_FAIL_ENTRY mean some internal errors in setting up kvm_run / registers correctly.
- KVM_EXIT_INTERNAL_ERROR generally means that we saw an invalid instruction.

Rust

C was directly calling the KVM API. We had to remember many things like different ioctl calls.

Rust wrapper on top of the kvm API. Written by chromium OS and by Amazon Firecracker.

simple.rs

- kvm::new() does ioctl calls
- kvm.create_vm() does ioctl calls
- create_vpu() does ioctl calls and also map kvm_run structure
- set_user_memory_region does ioctl calls
- you can get a raw pointer. writing to raw pointer is **unsafe**. Pointers may be null. Rust references can never be null. You have to do unsafe things at times, but you try to create a safe abstraction over unsafe blocks. Minimize unsafe blocks.
- get_regs, get_sregs do ioctl calls
- Let's run, we get 4.
- Let's print registers after we halt.

```
rdx : 1016 = 0x3f8
rax : 10 = int ascii value of "\n"
rbx : 2. Unchanged
rip : 4108 = 0x100C. 0x1000 starting address, C = len(asm_code) = 12.
rflags : 2.
```

simple_in.rs

Let's take input. Set rax to 3. Let's print the registers again:

```
rflags : 6 = 4 + 2. 4 => Parity flag since the sum is "5". 2 bit is always 1.
```

simple_mm.rs

- Let's load / store stuff to / from memory and see the program loading storing stuff.
- Set a flag that we want to see memory reads and writes.
- See exits

simple_ss.rs

- Let's back the memory with a file.
- After halt we can see what was the state of the memory.