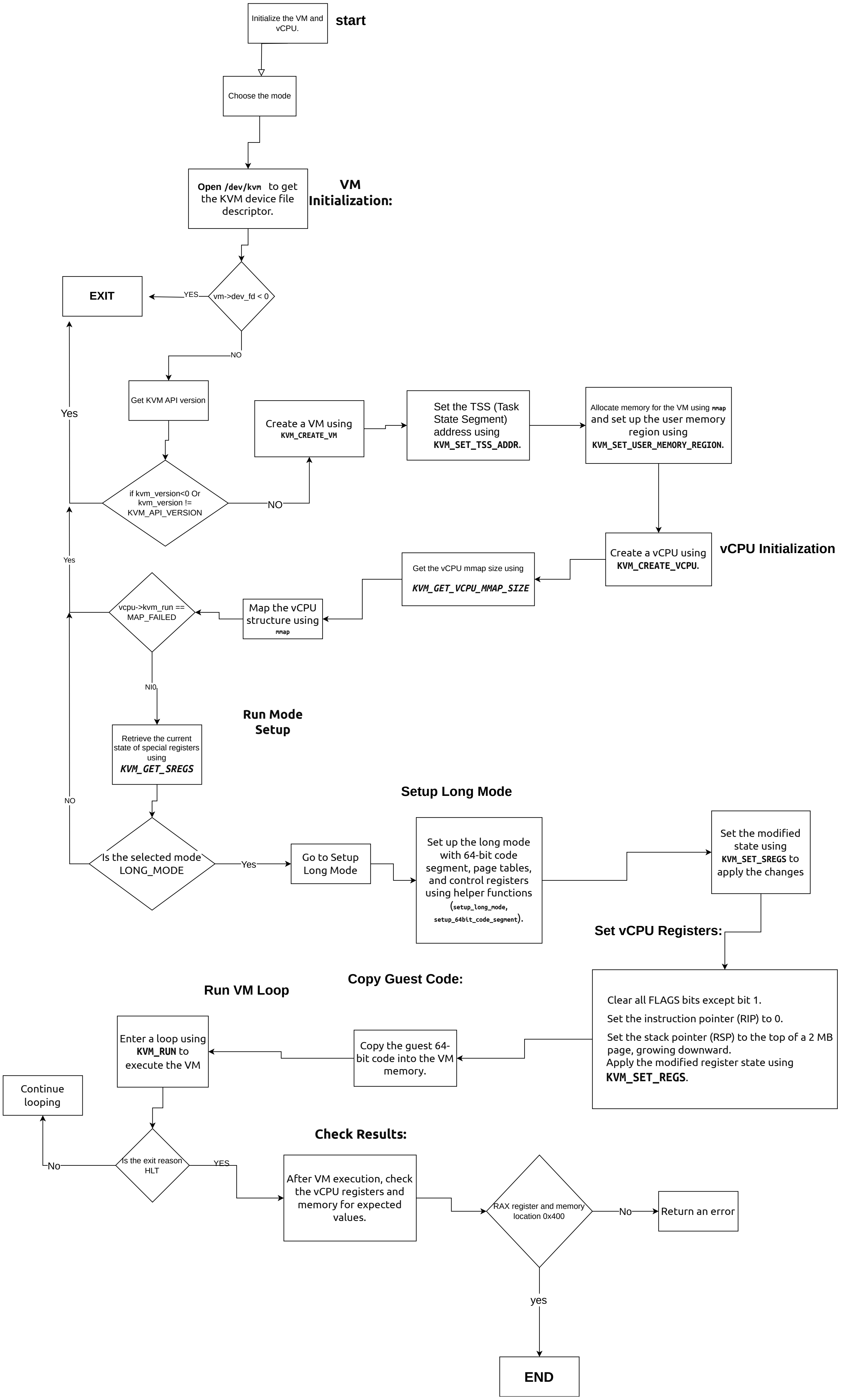
22M0749


start
Initialize the VM and vCPU.
Choose the mode
VM Initialization:
Open /dev/kvm to get the KVM device file descriptor.
vm->dev_fd < 0
YES
EXIT
NO
Get KVM API version
if kvm_version<0 Or kvm_version != KVM_API_VERSION
Yes
NO
Create a VM using KVM_CREATE_VM
Set the TSS (Task State Segment) address using KVM_SET_TSS_ADDR.
Allocate memory for the VM using mmap and set up the user memory region using KVM_SET_USER_MEMORY_REGION.
vCPU Initialization
Create a vCPU using KVM_CREATE_VCPU.
Get the vCPU mmap size using KVM_GET_VCPU_MMAP_SIZE
Map the vCPU structure using mmap
vcpu->kvm_run == MAP_FAILED
Yes
NI0
Run Mode Setup
Retrieve the current state of special registers using KVM_GET_SREGS
NO
Is the selected mode LONG_MODE
Yes
Go to Setup Long Mode
Setup Long Mode
Set up the long mode with 64-bit code segment, page tables, and control registers using helper functions (setup_long_mode, setup_64bit_code_segment).
Set the modified state using KVM_SET_SREGS to apply the changes
Set vCPU Registers:
Clear all FLAGS bits except bit 1.
Set the instruction pointer (RIP) to 0.
Set the stack pointer (RSP) to the top of a 2 MB page, growing downward.
Apply the modified register state using KVM_SET_REGS.
Copy Guest Code:
Copy the guest 64-bit code into the VM memory.
Run VM Loop
Enter a loop using KVM_RUN to execute the VM
Is the exit reason HLT
No
Continue looping
YES
Check Results:
After VM execution, check the vCPU registers and memory for expected values.
RAX register and memory location 0x400
No
Return an error
yes
END

1a 2nd part

Initialize VM and vCPU:

- KVM API: `open("/dev/kvm")`, `ioctl(vm_fd, KVM_CREATE_VM)`, `ioctl(vcpu_fd, KVM_CREATE_VCPU)`

- Description: We start by opening the KVM device file using `open("/dev/kvm")`. Then, we create a VM and a vCPU using `ioctl` calls with `KVM_CREATE_VM` and `KVM_CREATE_VCPU` on the respective file descriptors.

Verify KVM API Version:

- KVM API: `ioctl(dev_fd, KVM_GET_API_VERSION)`

- Description: We check the version of the KVM API using `ioctl` with `KVM_GET_API_VERSION` on the device file descriptor (`dev_fd`).

Set TSS Address:

- KVM API: `ioctl(vm_fd, KVM_SET_TSS_ADDR)`

- Description: We set the address of the Task State Segment (TSS) for our VM using `ioctl` with `KVM_SET_TSS_ADDR` on the VM file descriptor (`vm_fd`).

Allocate Memory for VM:

- KVM API: `mmap`, `ioctl(vm_fd, KVM_SET_USER_MEMORY_REGION)`

- Description: We allocate memory for our VM using `mmap` and then inform the kernel about this memory region by making an `ioctl` call with `KVM_SET_USER_MEMORY_REGION` on the VM file descriptor (`vm_fd`).

Retrieve VM Special Registers:

- KVM API: `ioctl(vcpu_fd, KVM_GET_SREGS)`

- Description: We retrieve the current state of certain special registers for the VM by using `ioctl` with `KVM_GET_SREGS` on the vCPU file descriptor (`vcpu_fd`).

Set VM Special Registers:

- KVM API: `ioctl(vcpu_fd, KVM_SET_SREGS)`
- Description: We inform the kernel about changes in the special registers of our VM using `ioctl` with `KVM_SET_SREGS` on the vCPU file descriptor (`vcpu_fd`).

Set vCPU Registers:

- KVM API: `ioctl(vcpu_fd, KVM_SET_REGS)`
- Description: We set the registers for our virtual CPU using `ioctl` with `KVM_SET_REGS` on the vCPU file descriptor (`vcpu_fd`). This includes specifying values like the instruction pointer (RIP) and stack pointer (RSP).

Run VM Loop:

- KVM API: `ioctl(vcpu_fd, KVM_RUN)`
- Description: We enter a loop where the VM executes its code. The kernel helps us with this by providing a way to run the VM using `ioctl` with `KVM_RUN` on the vCPU file descriptor (`vcpu_fd`).

Check Exit Reason:

- KVM API: Accessing `vcpu->kvm_run->exit_reason`
- Description: After each run of the VM, we check why it exited the loop by accessing `vcpu->kvm_run->exit_reason`. This information helps us decide the next steps in our program.