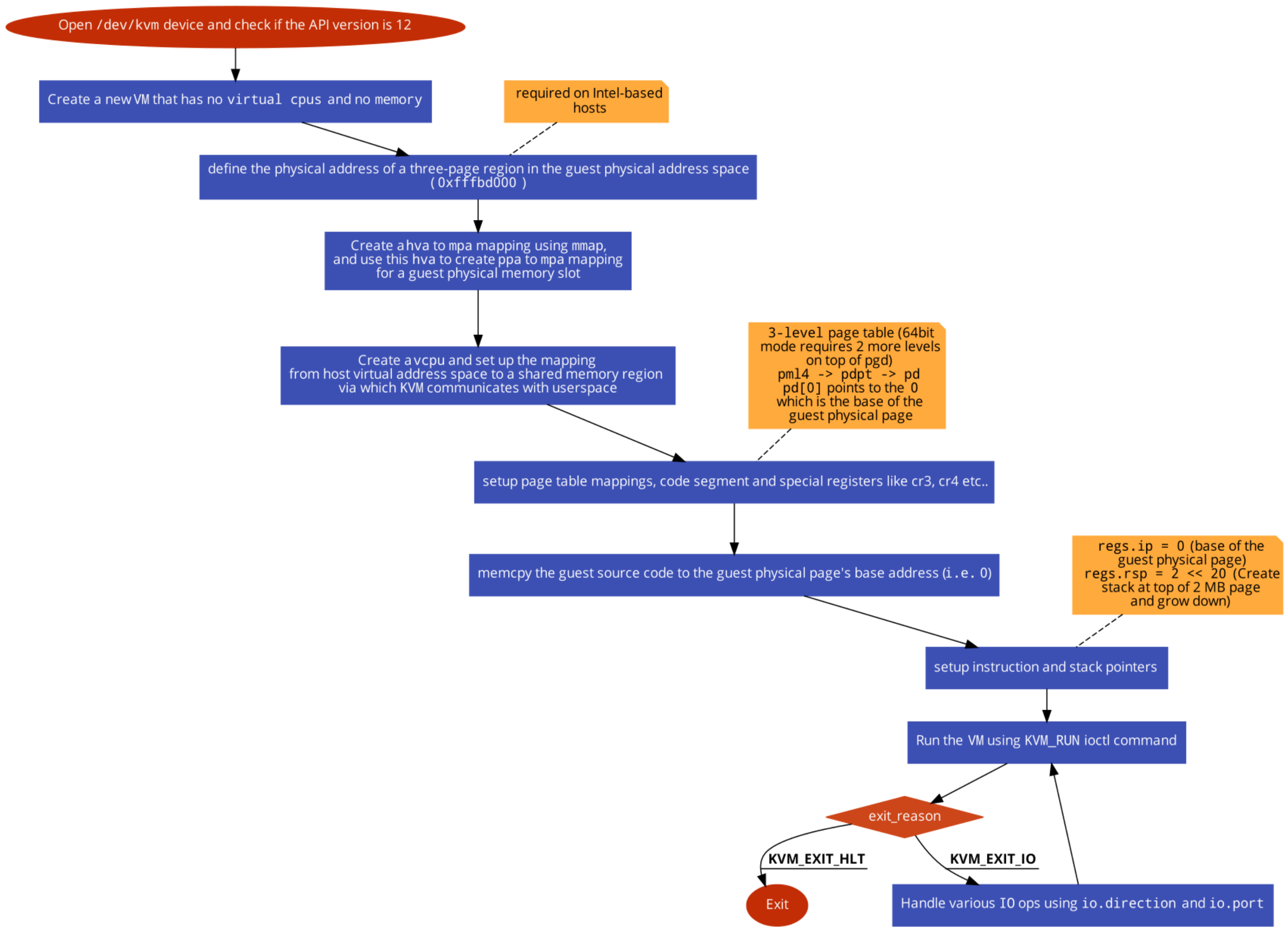
Open /dev/kvm device and check if the API version is 12
Create a new VM that has no virtual cpus and no memory
required on Intel-based hosts
define the physical address of a three-page region in the guest physical address space (0xfffbd000)
Create a hva to mpa mapping using mmap, and use this hva to create ppa to mpa mapping for a guest physical memory slot
Create a vcpu and set up the mapping from host virtual address space to a shared memory region via which KVM communicates with userspace
3-level page table (64bit mode requires 2 more levels on top of pgd) pml4 -> pdpt -> pd pd[0] points to the 0 which is the base of the guest physical page
setup page table mappings, code segment and special registers like cr3, cr4 etc..
memcpy the guest source code to the guest physical page's base address (i.e. 0)
regs.ip = 0 (base of the guest physical page) regs.rsp = 2 << 20 (Create stack at top of 2 MB page and grow down)
setup instruction and stack pointers
Run the VM using KVM_RUN ioctl command
exit_reason
KVM_EXIT_HLT
KVM_EXIT_IO
Exit
Handle various IO ops using io.direction and io.port

Note: In a few points I've clubbed 2-3 actions together, so the total count of points has become less than 10 (i.e. 9)

1. Open /dev/kvm Device and Check if the API Version is 12
KVM_GET_API_VERSION: This ioctl call retrieves the version of the KVM API supported by the kernel. It takes no input parameters. The returned value indicates the version number of the KVM API. Applications should refuse to run if it returns a value other than 12.
2. Create a New VM
KVM_CREATE_VM: It creates a new VM that has no virtual CPUs and no memory. It takes a machine type identifier as a parameter (usually 0) and returns a file descriptor (VM fd) that can be used to control the new virtual machine.
3. Define the Physical Address of a Three-Page Region in the Guest Physical Address Space
KVM_SET_TSS_ADDR: It takes the TSS address (physical address) as the input. The region must be within the first 4GB of the guest physical address space and must not conflict with any memory slot or any MMIO address. This ioctl is required on Intel-based hosts. This is needed on Intel hardware because of a quirk in the virtualization implementation.
4. **Create an HVA to MPA Mapping Using mmap, and Use this HVA to Create PPA to MPA Mapping for a Guest Physical Memory Slot**
KVM_SET_USER_MEMORY_REGION: This ioctl call sets up memory regions for the VM. More specifically, this ioctl allows the user to create, modify, or delete a guest physical memory slot. It takes struct kvm_userspace_memory_region as input. It contains slot number, guest_phys_addr, userspace_addr (start of the userspace allocated memory), etc.
5. **Create a VCPU and Set Up the Mapping from Host Virtual Address Space to a Shared Memory Region via which KVM Communicates with Userspace**
KVM_CREATE_VCPU: This ioctl call creates a new virtual CPU (vCPU) for the VM. It returns a file descriptor representing the vCPU. Takes vCPU ID as input (0 in our program).
KVM_GET_VCPU_MMAP_SIZE: This ioctl call retrieves the size of memory required for memory-mapping the vCPU. It helps allocate memory for communication between the vCPU and user-space (during KVM_RUN ioctl).
6. **Setup Page Table Mappings, Code Segment, and Special Registers like CR3, CR4, etc. Also, memcpy the Guest Source Code to the Guest Physical Page**
KVM_SET_SREGS: This ioctl call sets the special registers (e.g., segment registers, control registers) for the vCPU. It includes setting up the page table mappings (CR3), code segment (CS), and other control registers (CR0, CR4, EFER). It takes kvm_sregs struct as input.
7. Setup Instruction and Stack Pointers
KVM_SET_REGS: This ioctl call sets the general-purpose registers (e.g., instruction pointer, stack pointer) for the vCPU. It initializes the instruction pointer (RIP) to start executing instructions from the guest physical page and sets up the stack pointer (RSP) to point to the top of the stack. It takes struct kvm_regs as input (contains values of all the regs).
8. Run the VM using KVM_RUN ioctl command
KVM_RUN: This ioctl is used to run a guest virtual CPU. While there are no explicit parameters, there is an implicit parameter block that can be obtained by mmap()ing the vCPU fd at offset 0, with the size given by KVM_GET_VCPU_MMAP_SIZE. The parameter block is formatted as a

'struct kvm_run'. When a VM exits, it populates that kvm_run struct with relevant info which can be used by the userspace program to do say IO handling (in case the exit happened due to IO).

9. Translate Guest Virtual Address to Guest Physical Address

KVM_TRANSLATE: Translates a guest virtual address according to the vCPU's current address translation mode. You pass a kvm_translation struct in which you assign the linear_address field to the GVA which you want the translation of. On success, the rest of the fields in the struct are populated.