Q1-A) Below are the KVM_APIs, their explanation, and their ioctl arguments.

1. **KVM_CREATE_VM**: This API facilitates the creation of a new virtual machine (VM) instance within the KVM subsystem. When invoked, it allocates the necessary resources, such as memory and data structures, to represent the VM within the kernel. Additionally, it initializes the VM to an initial state, enabling subsequent configuration and execution.

   ioctl(vm->dev_fd, KVM_CREATE_VM, 0);

   Here,
   a. **vm->dev_fd**: This file descriptor refers to the KVM device. The KVM device file is typically obtained by opening the /dev/kvm device file using the open system call. This file descriptor is used to communicate with the KVM subsystem in the kernel.

   b. **0**: This argument is not utilized in this particular ioctl call. It signifies that no additional parameters or data structures are required to create the VM. The VM creation does not require any additional configuration parameters in this context.

2. **KVM_SET_USER_MEMORY_REGION**: With this API, the host system configures a memory region accessible to the guest operating system running within the VM. It allows the host to define the memory layout and permissions for the guest, specifying attributes such as the starting address, size, and access permissions of the memory region. This API plays a crucial role in accurately setting up the guest's memory environment.

ioctl(vm->vm_fd, KVM_SET_USER_MEMORY_REGION, &memreg)

Here,

a. **vm->vm_fd**: This file descriptor refers to the virtual machine (VM) instance. The vm_fd represents the file descriptor associated with the VM, typically obtained after creating the VM using the KVM_CREATE_VM ioctl call. This file descriptor is used to communicate with the VM in the KVM subsystem.

b. &**memreg**: This argument is a pointer to a structure (memreg) containing information about the memory region to be configured. The structure typically includes fields such as slots, flags, etc. By passing a pointer to this structure, the ioctl call receives the necessary parameters to define the memory region within the guest's address space.

3. **KVM_CREATE_VCPU**: This API creates a new virtual CPU (VCPU) instance within a VM. Each VCPU represents a virtual processor that executes guest instructions. Upon invocation, this API sets up the necessary data structures and resources associated with the VCPU, enabling it to execute guest code within the VM environment.

   ioctl(vm->vm_fd, KVM_CREATE_VCPU, 0);
   The explanation for args is similar to the above KVM_CREATE_VM

4. **KVM_GET_VCPU_MMAP_SIZE**: This API retrieves the size of the memory-mapped region required for communication between the user-space application and the VCPU. Memory mapping efficiently exchanges data between the user-space application and the VCPU during VM execution.

   ioctl(vm->dev_fd, KVM_GET_VCPU_MMAP_SIZE, 0);
   The explanation for args is similar to the above KVM_CREATE_VM

5. **KVM_GET_SREGS**: When called, this API retrieves the current values of special CPU registers associated with a VCPU. These registers, such as control and segment registers, determine the execution state of the VCPU, including the privilege level and memory segmentation. By querying these registers, the host system can obtain insights into the current execution context of the VCPU.

   ioctl(vcpu->vcpu_fd, KVM_GET_SREGS, &sregs)
   Here,
   a. **vcpu->vcpu_fd**: This file descriptor refers to the virtual CPU (VCPU). The vcpu_fd represents the file descriptor associated with the VCPU, typically obtained after creating the VCPU using the KVM_CREATE_VCPU ioctl call. This file descriptor communicates with the VCPU in the KVM subsystem.

   b. &**sregs**: This argument is a pointer to a structure (sregs) where the state of the special CPU registers will be stored after the ioctl call. The structure typically includes fields representing various special CPU registers, such as control registers, segment registers, and descriptor tables. By passing a pointer to this structure, the ioctl call can populate it with the current state of the special CPU registers associated with the VCPU.

6. **KVM_SET_SREGS**: This API allows the host system to modify the values of special CPU registers for a VCPU. By updating these registers, the host can alter the execution state of the VCPU, potentially changing parameters such as the operating mode (e.g., real mode, protected mode, long mode) or memory segmentation. This API is crucial for configuring and controlling the behavior of the VCPU.

   ioctl(vcpu->vcpu_fd, KVM_SET_SREGS, &sregs)
   The explanation is similar to the above call.

7. **KVM_RUN**: Invoking this API initiates the execution of a VCPU within its associated VM. It starts the execution loop, allowing the VCPU to execute guest instructions until an event occurs, such as an interrupt, exception, or system call, or until the VM exits. This API plays a fundamental role in the overall execution flow of the virtual machine.

   ioctl(vcpu->vcpu_fd, KVM_RUN, 0)
   The explanation is similar to the above call.

8. **KVM_TRANSLATE**: This API translates a guest virtual address to a host physical address during VM execution. It ensures that memory accesses performed by the guest are correctly mapped to corresponding physical addresses on the host system. By facilitating this address translation, the API maintains proper memory isolation and access control within the virtualized environment, enhancing the security and integrity of the system.

   ioctl(vcpu->vcpu_fd, KVM_TRANSLATE, &tr)
   Here,
   a. &**tr**: This argument is a pointer to a structure (tr) representing the translation request. The structure typically includes information about the guest's virtual address that needs to be translated and provides space to store the resulting host's physical address after the ioctl call. By passing a pointer to this structure, the ioctl call can process the translation request and populate it with the corresponding host physical address.

9. **KVM_GET_API_VERSION**: Through this API, applications can retrieve the version information of the KVM kernel module currently loaded in the system. It provides details about the specific version of KVM being utilized, including version numbers, release dates, and potentially additional feature support. This allows applications to adapt their behavior or configuration based on the capabilities of the underlying KVM implementation.

ioctl(vm->dev_fd, KVM_GET_API_VERSION, 0);
The explanation is similar to the above calls.

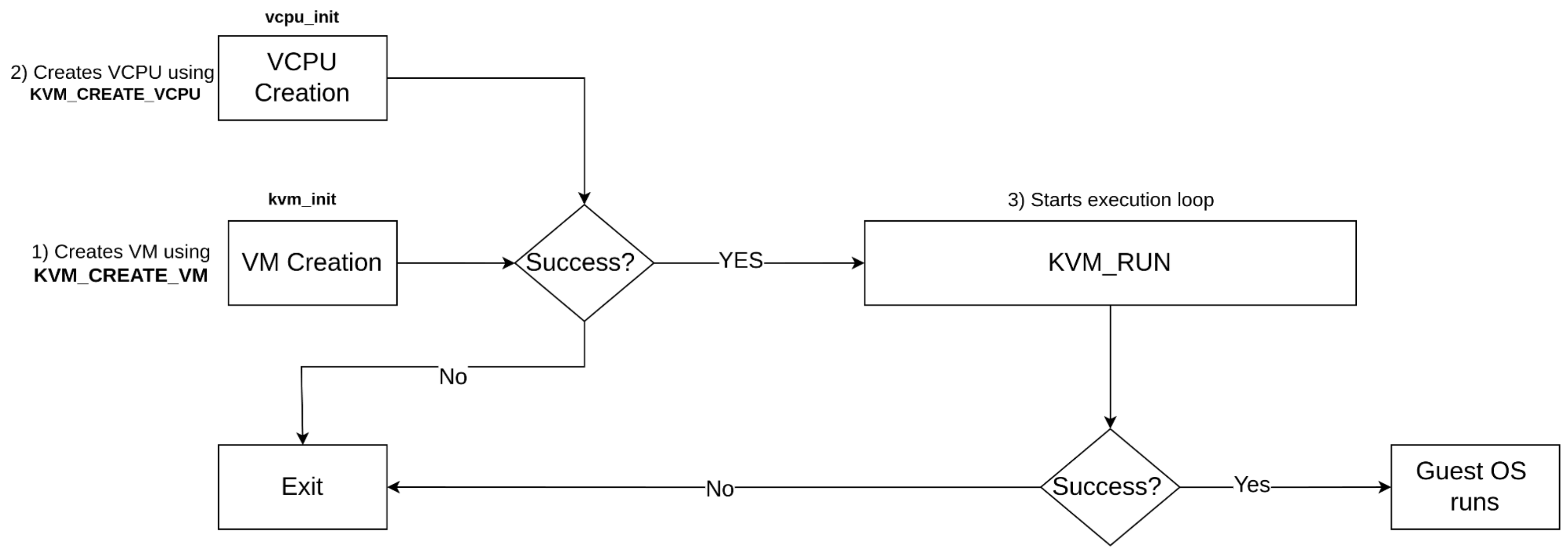

Fig-1 - KVM calls for the Hypervisor code.