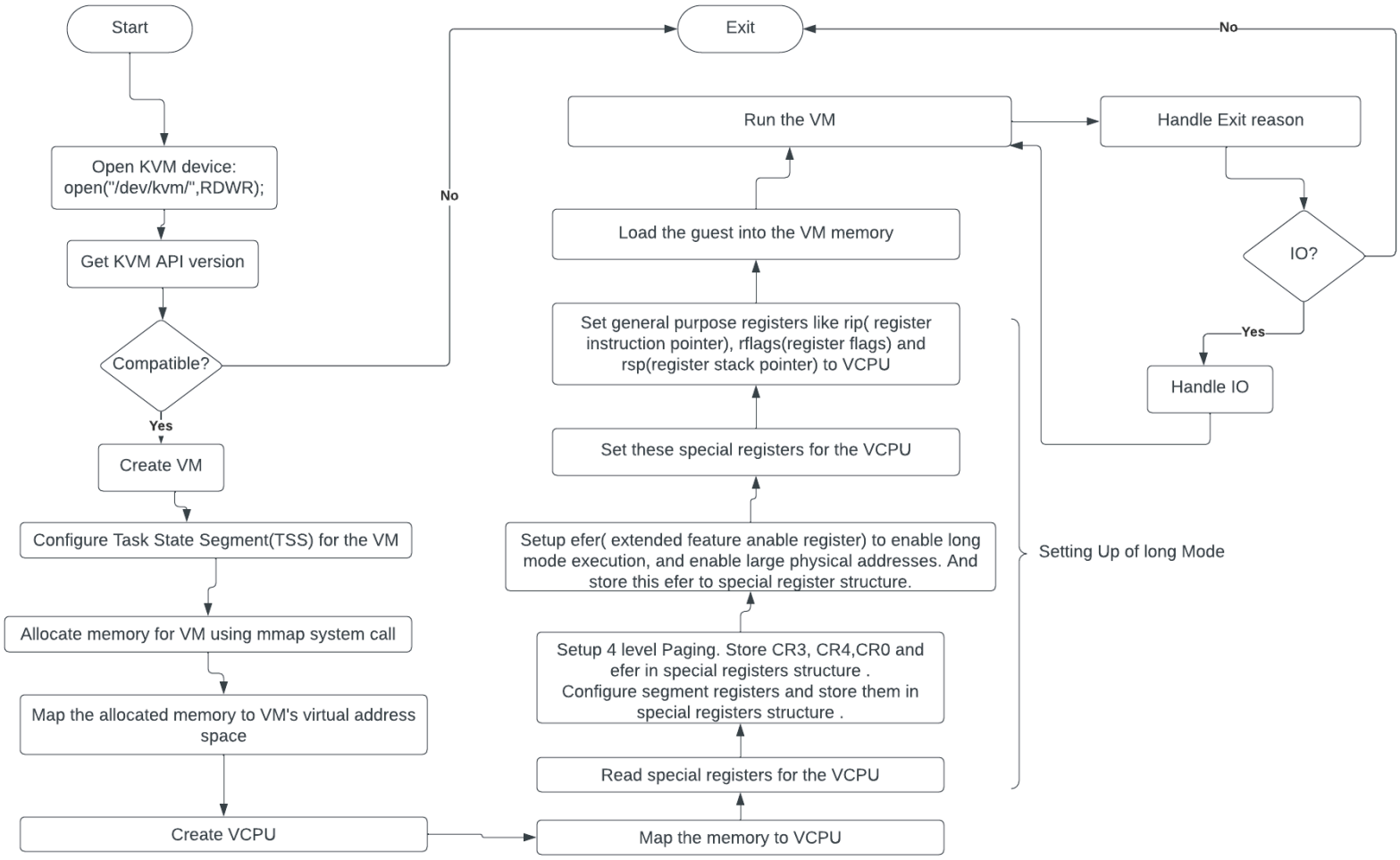
1) The flowchart to setup and execute a VM in the long mode of operation.



2)
Various KVM APIs associated with the actions mentioned above are:

I. **Get KVM API version**: To check the KVM API version currently we are using.
Earlier versions have unstable API , hence it is necessary to work with the right KVM version.The KVM_GET_API_VERSION ioctl call is used.

KVM_GET_API_VERSION:
Inputs to the ioctl call: vm device fd, KVM_GET_API_VERSION and NULL(or 0).

If it returns 12(i.e. the version number) , then we are good to work with the KVM otherwise, we need to update the KVM.

II. **Create VM**: This creates a new virtual machine instance for us. KVM_CREATE_VM ioctl call is used.

KVM_CREATE_VM:

Inputs to the ioctl call: vm device fd, KVM_CREATE_VM and machine type identifier(in our case it will be NULL or 0 , since new VM has no virtual cpus or memory)

This returns a vm fd. This creates a virtual machine for us which we will associate with the memory and one or more virtual cpus.

III. **Configure Task State Segment(TSS) for the VM**: Sets the guest's Task State Segment (TSS) address. TSS is a data structure used by the CPU to store information about the current task, including stack pointers, segment registers, and other context switch details. KVM_SET_TSS_ADDR ioctl call is used.

KVM_SET_TSS_ADDR:

Inputs to the ioctl call: vm fd, KVM_SET_TSS_ADDR and tss_address (unsigned long).

This returns 0 on success and -1 on error. This API defines the physical address space (usually within the first 4 GB of guest physical memory) to set up the TSS address. This is for x86 architectures.

IV. **Map the allocated memory to VM's virtual address space:** For this the KVM_SET_USER_MEMORY_REGION ioctl call is used.

KVM_SET_USER_MEMORY_REGION

Inputs to the ioctl call: vm fd , KVM_SET_USER_MEMORY_REGION and address of a struct kvm_userspace_memory_region data type.

struct kvm_userspace_memory_region is defined as:

```
struct kvm_userspace_memory_region {
	__u32 slot;
	__u32 flags;
	__u64 guest_phys_addr;
	__u64 memory_size; /* bytes */
	__u64 userspace_addr; /* start of the userspace allocated memory */
};
//source https://docs.kernel.org/virt/kvm/api.html
```

This returns 0 on success and -1 on error. This allocates the memory for the VM's guest OS. After this step we would have successfully allocated memory space to our vm.

V. **Create VCPU**:To create vcpu for the guest OS, we need KVM_CREATE_VCPU ioctl call.

KVM_CREATE_VCPU

Input to the ioctl call: vm fd, KVM_CREATE_VCPU and vcpu id(it is an integer).

On success it returns vcpu fd and -1 on error. This API will add the vcpu to the VM.

VI. **Map the memory to VCPU:** First we need to know the amount of memory we need to be mapped for kvm run. This can be done by using the KVM_GET_VCPU_MMAP_SIZE ioctl call. Then after this we can map the required memory to kvm run using mmap system call.

KVM_GET_VCPU_MMAP_SIZE

Inputs to the ioctl call: vm device fd , KVM_GET_VCPU_MMAP_SIZE and NULL (or 0).

This returns the size of the vcpu mmap area, in bytes. This return value can be further used to mmap the memory region to the vcpu->kvm_run.

VII. **Read special registers for the VCPU:** For this we need the KVM_GET_SREGS ioctl call.

KVM_GET_SREGS

Input to the ioctl call: vcpu fd , KVM_GET_SREGS and address of struct kvm_sregs data type.

struct kvm_sregs is defined as:

```
struct kvm_sregs {
struct kvm_segment cs, ds, es, fs, gs, ss;
struct kvm_segment tr, ldt;
struct kvm_dtable gdt, idt;
__u64 cr0, cr2, cr3, cr4, cr8;
__u64 efer;
__u64 apic_base;
__u64 interrupt_bitmap[(KVM_NR_INTERRUPTS + 63) / 64];
};
//source https://docs.kernel.org/virt/kvm/api.html
```

This returns 0 on success and -1 on error. This reads special registers from the VCPU. The read values are stored in struct kvm_sregs.

VIII. **Set these special registers for the VCPU**: This can be done using KVM_SET_SREGS ioctl call.

KVM_SET_SREGS

Input to the ioctl call: vcpu fd , KVM_SET_SREGS and address of struct kvm_sregs data type.

The struct kvm_sregs is defined the same as above.

On success it returns 0 and on error it returns -1. This API writes the special register values to the VCPU. Here we are writing cr3,cr0,cr4, efer, struct kvm_segment and other special registers to set up the long mode.

IX. **Set General purpose registers for the VCPU:** For this we are using KVM_SET_REGS ioctl call.

KVM_SET_REGS

Input to the ioctl call: vcpu fd, KVM_SET_REGS and address of struct kvm_regs data type.

struct kvm_regs is defined as:

```
struct kvm_regs {
/* out (KVM_GET_REGS) / in (KVM_SET_REGS) */
__u64 rax, rbx, rcx, rdx;
__u64 rsi, rdi, rsp, rbp;
__u64 r8,  r9,  r10, r11;
__u64 r12, r13, r14, r15;
__u64 rip, rflags;
};
//source https://docs.kernel.org/virt/kvm/api.html
```

On success it returns 0 and on error it returns -1.This sets the general purpose registers like flags, instruction pointer and stack pointer to the vcpu.

X. **Run the VM:** For this we need the KVM_RUN ioctl call.

KVM_RUN

Input to the ioctl call: vcpu fd, KVM_RUN and NULL(or 0) .

On success it returns 0 and on error it returns -1. This API is used to run the guest virtual cpu. This starts the vcpu execution.