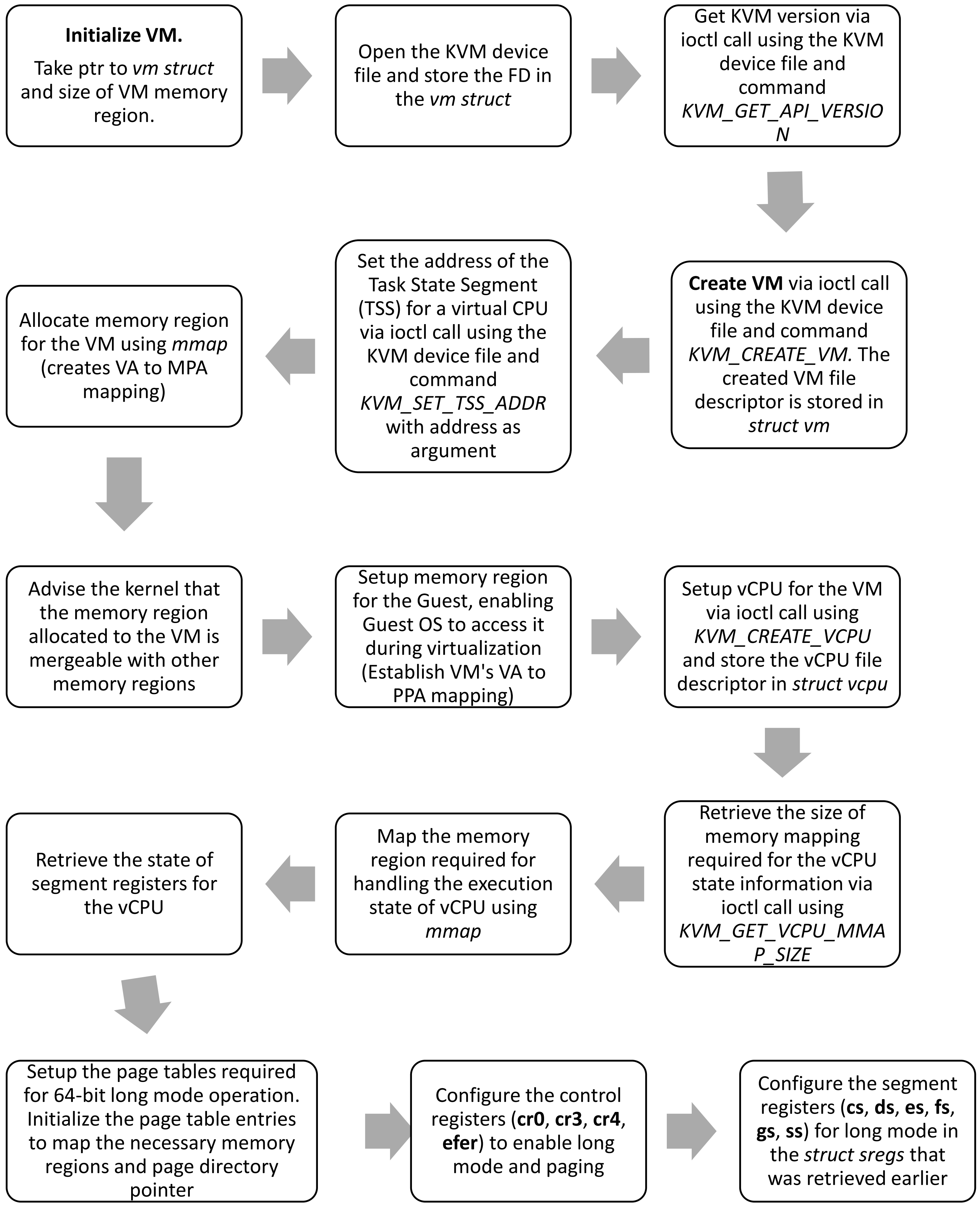
Shantanu Welling 210010076 CS695 A2

Q1a.1



Initialize VM. Take ptr to *vm struct* and size of VM memory region.

Open the KVM device file and store the FD in the *vm struct*

Get KVM version via ioctl call using the KVM device file and command *KVM_GET_API_VERSION*

Create VM via ioctl call using the KVM device file and command *KVM_CREATE_VM.* The created VM file descriptor is stored in *struct vm*

Set the address of the Task State Segment (TSS) for a virtual CPU via ioctl call using the KVM device file and command *KVM_SET_TSS_ADDR* with address as argument

Allocate memory region for the VM using *mmap* (creates VA to MPA mapping)

Advise the kernel that the memory region allocated to the VM is mergeable with other memory regions

Setup memory region for the Guest, enabling Guest OS to access it during virtualization (Establish VM's VA to PPA mapping)

Setup vCPU for the VM via ioctl call using *KVM_CREATE_VCPU* and store the vCPU file descriptor in *struct vcpu*

Retrieve the size of memory mapping required for the vCPU state information via ioctl call using *KVM_GET_VCPU_MMAP_SIZE*

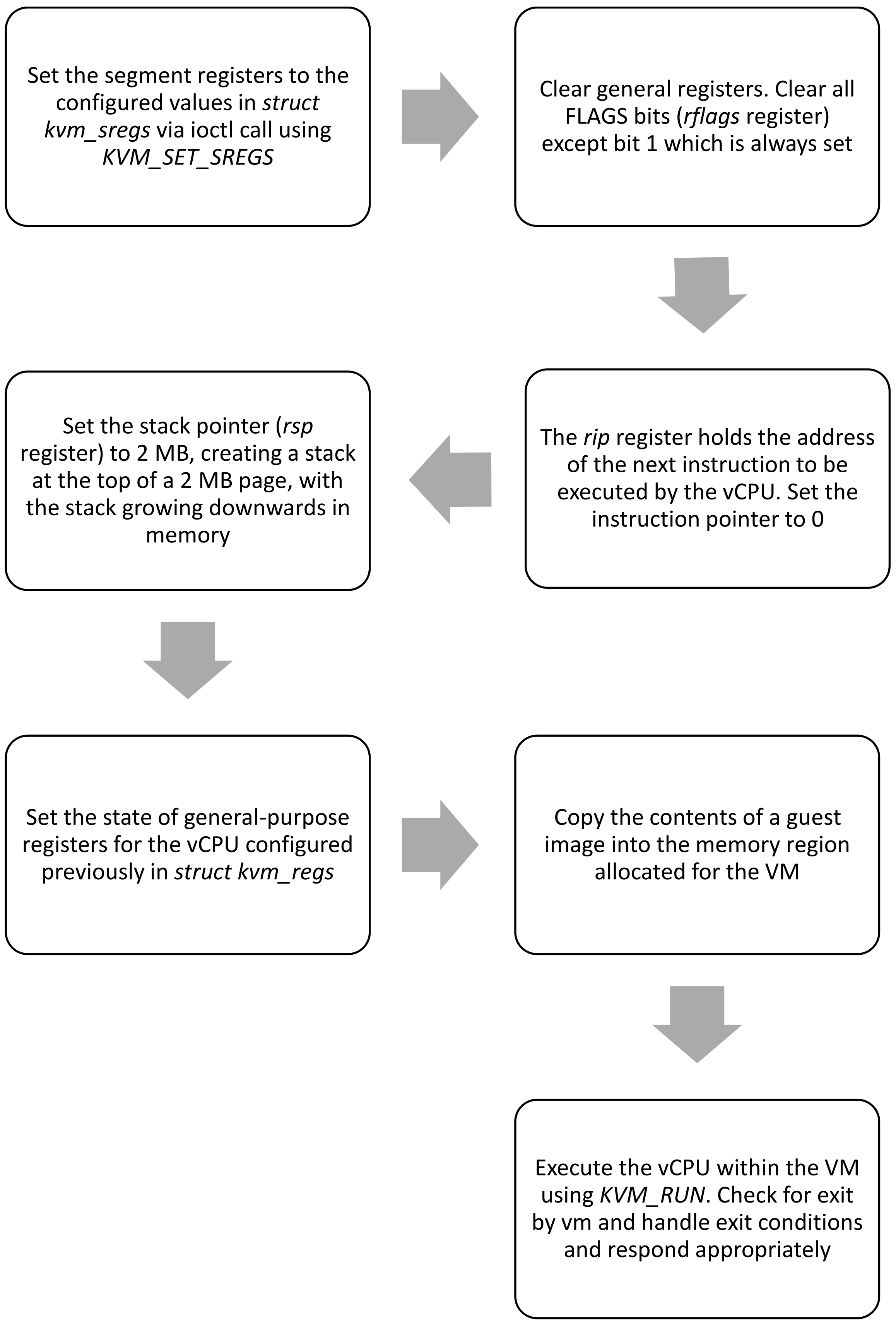
Map the memory region required for handling the execution state of vCPU using *mmap*

Retrieve the state of segment registers for the vCPU

Setup the page tables required for 64-bit long mode operation. Initialize the page table entries to map the necessary memory regions and page directory pointer

Configure the control registers (**cr0**, **cr3**, **cr4**, **efer**) to enable long mode and paging

Configure the segment registers (**cs**, **ds**, **es**, **fs**, **gs**, **ss**) for long mode in the *struct sregs* that was retrieved earlier



Set the segment registers to the configured values in *struct kvm_sregs* via ioctl call using *KVM_SET_SREGS*

Clear general registers. Clear all FLAGS bits (*rflags* register) except bit 1 which is always set

The *rip* register holds the address of the next instruction to be executed by the vCPU. Set the instruction pointer to 0

Set the stack pointer (*rsp* register) to 2 MB, creating a stack at the top of a 2 MB page, with the stack growing downwards in memory

Set the state of general-purpose registers for the vCPU configured previously in *struct kvm_regs*

Copy the contents of a guest image into the memory region allocated for the VM

Execute the vCPU within the VM using *KVM_RUN*. Check for exit by vm and handle exit conditions and respond appropriately

Q1a.2

1. To get the KVM version, `ioctl` call `KVM_GET_API_VERSION` is used
    a. input arguments:
        i. KVM device file descriptor
    b. Returns KVM version
2. To create VM, `ioctl` call `KVM_CREATE_VM` is used
    a. input args:
        i. KVM device file descriptor
    b. Creates VM and the VM file descriptor is returned
3. To set the address of the Task State Segment (TSS) for a virtual CPU, `ioctl` call `KVM_SET_TSS_ADDR` is used
    a. input args:
        i. The file descriptor associated with the KVM virtual machine
        ii. Address where the TSS should be located in the virtual address space
    b. Sets the address of TSS which stores information like process register state, I/O ports etc about a task
4. VM memory region is allocated using `mmap()` system call. It maps files or devices into memory. In our case, it is used to allocate a memory region.
    a. input args:
        i. `NULL`: Starting address of the allocation (the system chooses the address).
        ii. Size of the memory region to allocate.
        iii. `PROT_READ` | `PROT_WRITE`: Protection flags indicating that the memory region should be readable and writable.
        iv. `MAP_PRIVATE` creates a private copy-on-write mapping, `MAP_ANONYMOUS` creates an anonymous mapping, and `MAP_NORESERVE` indicates that no swap space should be reserved.
        v. -1: File descriptor
        vi. 0: Offset
    b. Maps memory region of the given size and returns virtual address to the start of memory region
5. To give advice or make requests about the behaviour of memory mappings to the kernel, `madvise()` system call is used
    a. input args:
        i. Pointer to the start of the memory region.
        ii. Size of the memory region.
        iii. Flags: `MADV_MERGEABLE`
    b. Advice to the kernel that the memory region is mergeable with other memory regions, potentially improving memory utilization. Enable Kernel Same Page Merging for the pages in the range specified by address and size. The kernel regularly scans those areas of user memory that have been marked as mergeable, looking for pages with identical content. These are replaced by a single write-protected page (which is automatically copied if a process later wants to update the content of the page).

6. Set up guest VM VA to PPA mapping i.e. set up the memory region for VM, enabling the guest operating system to access it during virtualization. The `kvm_userspace_memory_region` data structure used to define and manage memory regions allocated by user-space processes for VM. Its fields are:
   a. `slot`: Specifies the slot index for the memory region
   b. `flags`: Flags for the memory region
   c. `guest_phys_addr`: Specifies the guest physical address where the memory will be mapped. In our case, set to 0, indicating that the memory will be mapped at the beginning of the guest's physical memory space.
   d. `memory_size`: Size of the memory region
   e. `userspace_addr`: Address of the memory region in userspace process's address space where the memory region is located

7. Set memory region for VM using `ioctl` call KVM_SET_USER_MEMORY_REGION
   a. input args:
      i. Pointer to the `kvm_userspace_memory_region` structure
      ii. VM file descriptor
   b. Sets up memory region using the configuration given in the input struct

8. Create vCPU using `ioctl` call KVM_CREATE_VCPU
   a. input args:
      i. VM file descriptor. It serves as a reference to the virtual machine with which the operation is to be performed.
      ii. The index of the vCPU to be created
   b. Returns file descriptor of the new vCPU added to the VM. After this call, the virtual machine will have an additional vCPU available for execution, which can be utilized for running guest operating system processes. Each vCPU created in this manner represents a separate execution context within the virtualized environment.

9. Retrieve the size of memory mapping required for the vCPU state information using `ioctl` call KVM_GET_VCPU_MMAP_SIZE
   a. input args:
      i. VM file descriptor
      ii. The index of the vCPU
   b. The KVM_RUN `ioctl` communicates with userspace via a shared memory region. This `ioctl` call returns the size of that region.

10. Map the memory region required for handling the execution state of vCPU in `struct vcpu's` attribute which a `struct kvm_run`, using mmap syscall with flags PROT_READ, PROT_WRITE (These flags specify that the memory region should be readable and writable), MAP_SHARED (This flag indicates that the mapped memory region will be shared between multiple processes). The `struct kvm_run` is used to communicate information between the kernel and user space when running a vCPU. It serves as a container for various data fields that represent the state of the vCPU and provide information about the execution context of the guest operating system running within the VM. It stores info like headers, state of CPU registers and exit reason. Additional data fields provide context-specific information about the vCPU's execution state, such as the status of virtual devices, interrupt vectors, or pending operations.

11. To retrieve the state of segment registers for the vCPU, `ioctl` call KVM_GET_SREGS is used
    a. It takes as input a pointer to a structure (`sregs`) where the segment registers state will be stored after the operation completes.
    b. It also takes vCPU file descriptor as input. It is used to identify and communicate with the specific vCPU for which the operation is to be performed.

   c. It gets the segment registers state and stores them in the struct argument provided.

12. **setup_long_mode**:
    a. set up the page tables required for 64-bit long mode operation.
    b. initialize the Page Map Level 4 (PML4), Page Directory Pointer Table (PDPT), and Page Directory (PD) entries to map the necessary memory regions.
    c. configure the control registers (`cr0, cr3, cr4, efer`) to enable long mode and paging.
    d. call `setup_64bit_code_segment()` to configure the code and data segment registers.
        i. This function configures the segment registers `(cs, ds, es, fs, gs, ss)` in the `struct sregs` for long mode operation.
        ii. It sets up the code segment with attributes for code execution and read access.
        iii. It sets up the data segments (`ds, es, fs, gs, ss`) with attributes for read/write access.
        iv. The segment selector and other attributes like base and limit of segment are set accordingly to define the segment characteristics

13. Setting the state of segment registers for the vCPU, `ioctl` call KVM_SET_SREGS is used
    a. It takes as input a pointer to a structure (`sregs`) containing the desired segment register state set by previous steps
    b. It also takes vCPU file descriptor as input. It is used to identify and communicate with the specific vCPU for which the operation is to be performed.
    c. It sets the segment registers state to the state specified by the input structure

14. Setting the state of general-purpose registers for the vCPU, `ioctl` call KVM_SET_REGS is used
    a. It takes as input a pointer to a structure (`regs`) containing the desired general purpose register state
    b. It also takes vCPU file descriptor as input. It is used to identify and communicate with the specific vCPU for which the operation is to be performed.
    c. It sets the general-purpose registers' state to the state specified by the input structure

15. Copy the contents of a guest image into the memory region allocated for the VM using `memcpy`
    a. `vm->mem`: This represents the starting address of the memory region allocated for the VM. It is a pointer to the beginning of the memory region (destination pointer where the data will be copied).
    b. `guest64`: This is a pointer to the beginning of the guest image in memory (source pointer from where the data will be copied).
    c. `guest64_end - guest64`: The size of the guest image obtained by subtracting the starting address from the ending address. It represents the number of bytes to be copied.

16. Finally execute the vCPU within the VM using `ioctl` call KVM_RUN which takes the file descriptor of the vCPU as input argument. Runs the VM in a loop and checks for exit by VM and handles exit conditions appropriately. The exit reason is stored in `kvm_run struct` in the `vcpu struct`.
    a. Check if KVM_EXIT_HLT i.e. halt instruction by guest which allows the hypervisor to take control
    b. Check if KVM_EXIT_IO i.e. vCPU encountered an I/O operation
        i. If this check passes, further check if exit reason is KVM_EXIT_IO_OUT and port number is `0xE9`, if yes then it means that an output operation to port `0xE9` is being performed. In response, retrieve the output data from the `kvm_run` structure and write it to the `stdout`.