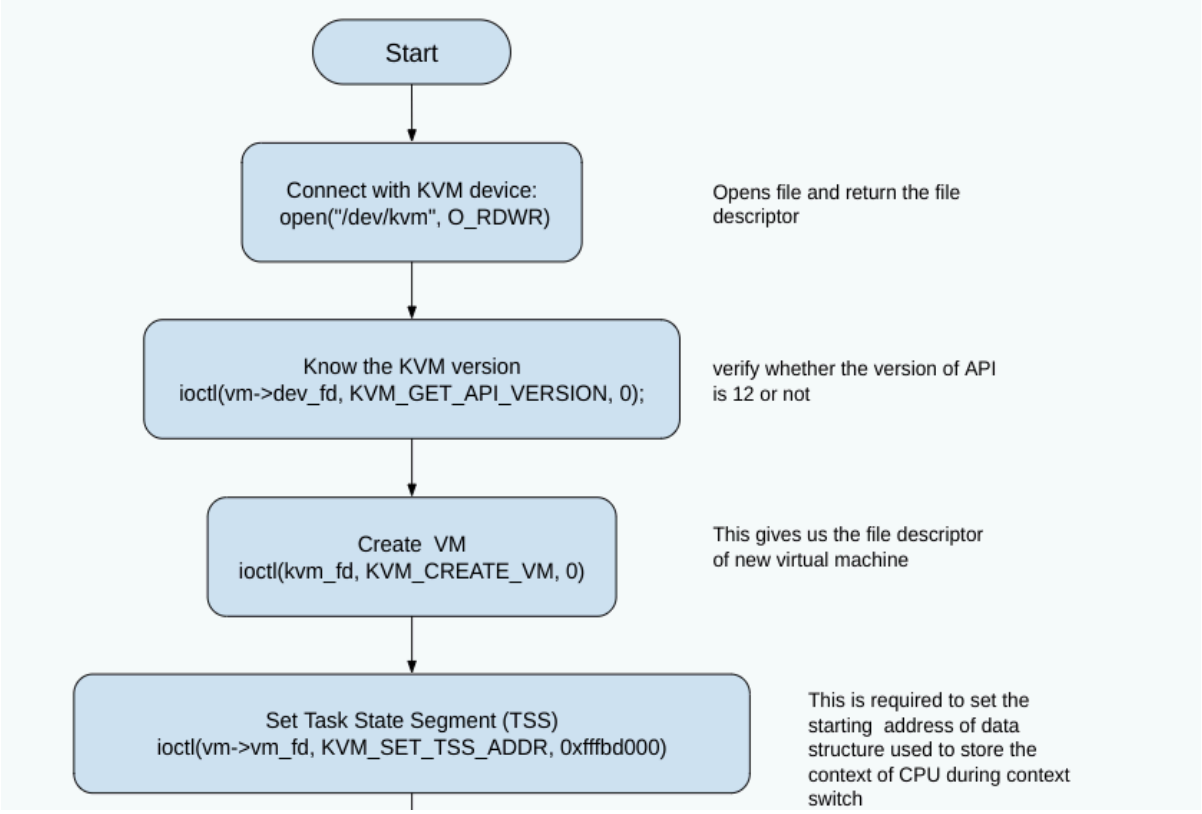
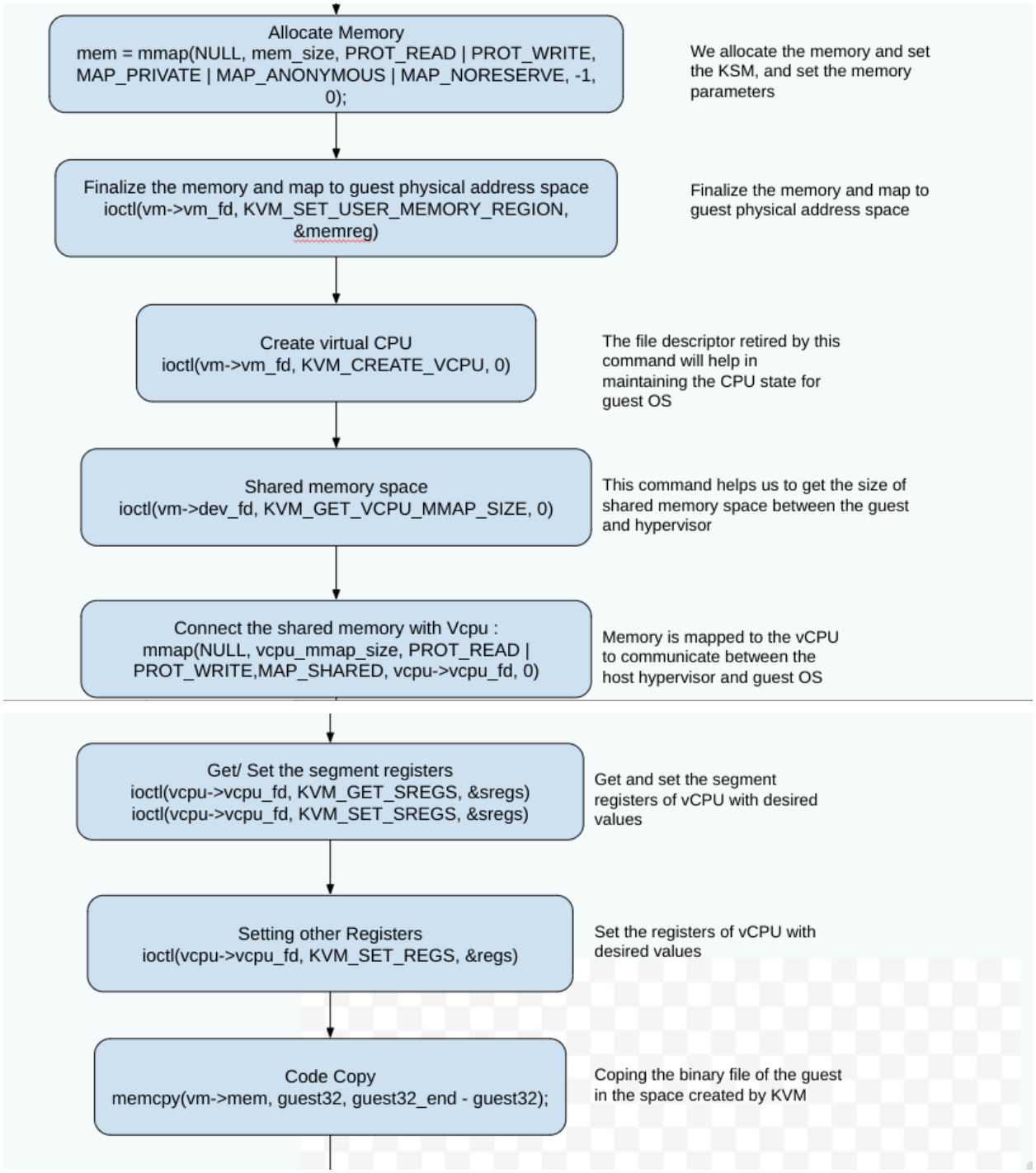
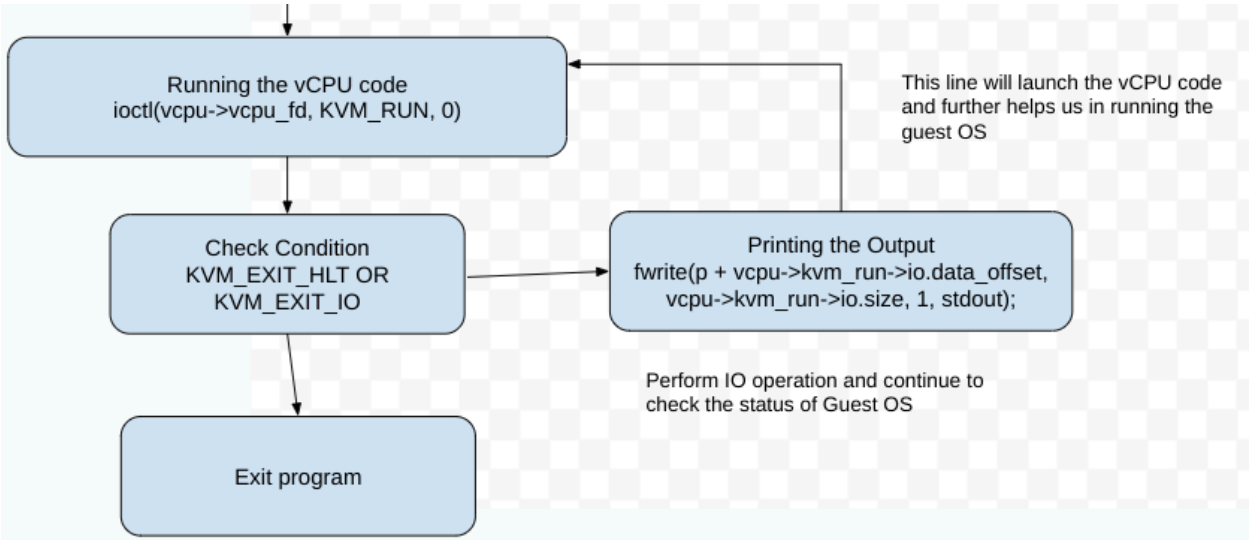
Question 1.a1: Flow chart




Allocate Memory
mem = mmap(NULL, mem_size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE, -1, 0);
We allocate the memory and set the KSM, and set the memory parameters
Finalize the memory and map to guest physical address space
ioctl(vm->vm_fd, KVM_SET_USER_MEMORY_REGION, &memreg)
Finalize the memory and map to guest physical address space
Create virtual CPU
ioctl(vm->vm_fd, KVM_CREATE_VCPU, 0)
The file descriptor retired by this command will help in maintaining the CPU state for guest OS
Shared memory space
ioctl(vm->dev_fd, KVM_GET_VCPU_MMAP_SIZE, 0)
This command helps us to get the size of shared memory space between the guest and hypervisor
Connect the shared memory with Vcpu :
mmap(NULL, vcpu_mmap_size, PROT_READ | PROT_WRITE,MAP_SHARED, vcpu->vcpu_fd, 0)
Memory is mapped to the vCPU to communicate between the host hypervisor and guest OS
Get/ Set the segment registers
ioctl(vcpu->vcpu_fd, KVM_GET_SREGS, &sregs)
ioctl(vcpu->vcpu_fd, KVM_SET_SREGS, &sregs)
Get and set the segment registers of vCPU with desired values
Setting other Registers
ioctl(vcpu->vcpu_fd, KVM_SET_REGS, &regs)
Set the registers of vCPU with desired values
Code Copy
memcpy(vm->mem, guest32, guest32_end - guest32);
Coping the binary file of the guest in the space created by KVM



Question 1.a2:

Explanation:

(a)
extern const unsigned char guest64[], guest64_end[];

This variable are referring to the external variable, which are not present in the file but present in some other file which got compiled along with simple.c
They are read_only variables and guest64[] refer to the start of the program and the guest62_end[] refer to the end of the program. The difference will help us in determining the size of guest program.

(b)

pml4[0] = PDE64_PRESENT | PDE64_RW | PDE64_USER | pdpt_addr;

This points to page directory pointer table, and pdpt_addr contains the physical address and other parameters are
PDE64_PRESENT : The page is valid.
PDE64_RW : Read/Write permissions.
PDE64_USER : Allows user-mode access.

pdpt[0] = PDE64_PRESENT | PDE64_RW | PDE64_USER | pd_addr;

This points to page table directory table

pd[0] = PDE64_PRESENT | PDE64_RW | PDE64_USER | PDE64_PS;

This points to actual physical memory, after skipping 2MB because of PDE64_PS

Here we are configuring the CPU control registers

sregs->cr3 = pml4_addr;

Here we are configuring the CR3 registers, which points to the top level page table. This helps in memory mapping.

sregs->cr4 = CR4_PAE;

We are enabling support for the 4 level page table here, without this the CPU would not recognize the 64 bit page table format PAE extension is required in x86-64 paging.

sregs->cr0 = CR0_PE | CR0_MP | CR0_ET | CR0_NE | CR0_WP | CR0_AM | CR0_PG;

CR0 manages the cpu mode, CR0_PE enable the protected mode , CR0_PG enables the paging mode.

sregs->efer = EFER_LME | EFER_LMA;

Flags description:
EFER_LME (Long Mode Enable) Enables 64-bit paging.
EFER_LMA (Long Mode Active) depicts that the CPU is in 64-bit mode.

(c)

vm->mem = mmap(NULL, mem_size, PROT_READ | PROT_WRITE, MAP_PRIVATE |

MAP_ANONYMOUS | MAP_NORESERVE, -1, 0);

Here we are making the memory which guests will use.

Description about arguments in mmap()
NULL: Any available memory space will be picked
Mem_size: memory size we want to demand.

PROT_READ PROT_WRITE: read write allowed
MAP_PRIVATE: Any changes in this area will not be visible to other process.
MAP_ANONYMOUS: This is RAM level , it is not having the file
MAP_NORESERVE: Reverse swap space is not allowed to avoid overcommit issues

madvise(vm->mem, mem_size, MADV_MERGEABLE);

Here we are optimizing the Memory with KSM,
Kernel Samepage Merging.
The function will look if there is any other pages of the different VM which are similar to the one which the first VM is having, then it'll combine those pages. Resulting in less RAM usage.

(d)

```
case KVM_EXIT_IO:

if (vcpu->kvm_run->io.direction == KVM_EXIT_IO_OUT &&

   vcpu->kvm_run->io.port == 0xE9) {

char *p = (char *)vcpu->kvm_run;

fwrite(p + vcpu->kvm_run->io.data_offset,

      vcpu->kvm_run->io.size, 1, stdout);

fflush(stdout);

continue;

}
```

Here we are handling the case where the VM exits for the IO operation, then we are checking the VM wants to write or read on the specific port. We further take the data form the kvm_run structure which is shared between the guest and host. During the data fetch we also give the offset and size upto which we want to read the data. Then this data is flushed to the host terminal.

Then again we continue the execution of VM.

(e)
memcpy(&memval, &vm->mem[0x400], sz);

This function is going into the memory of the VM and fetching the value from the offset of 0x400 and storing it in the variable memval. It also specifies the size in bytes upto which we want to read from the memory.