CS695 – ASSIGNMENT 3

ANUBHAV JANA (22M2109)

1.a.'ioctl' system call takes three arguments:

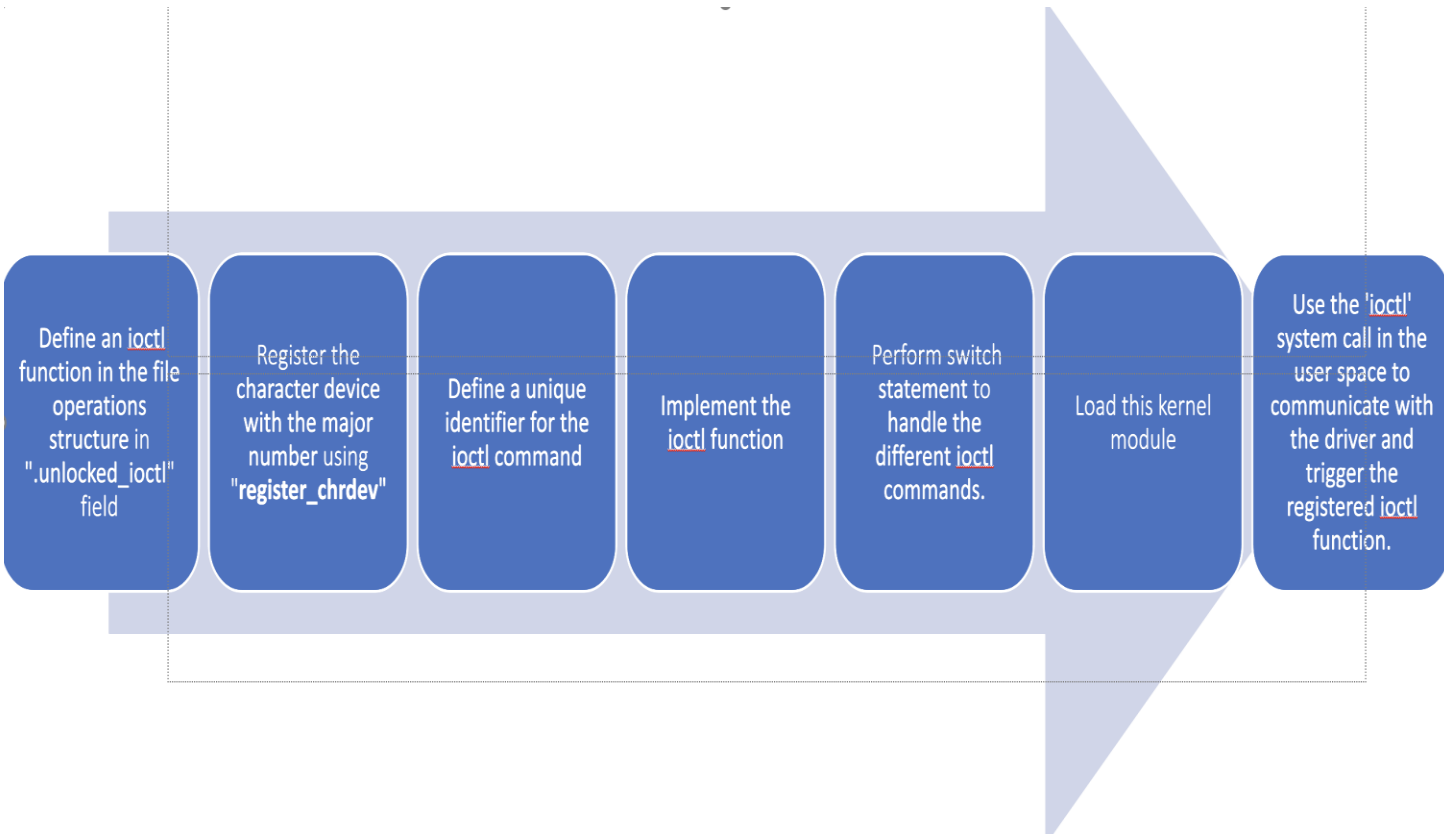
- **<u>file descriptor (int fd):</u>** This is the file descriptor of the device file that is being communicated with. It is returned by the 'open' system call when the device file is opened.

- **<u>request (unsigned long request)</u>:** This is the ioctl command number, which specifies the specific request being made to the device driver. The exact values and meaning of the command numbers are defined by the device driver and are unique to each device.

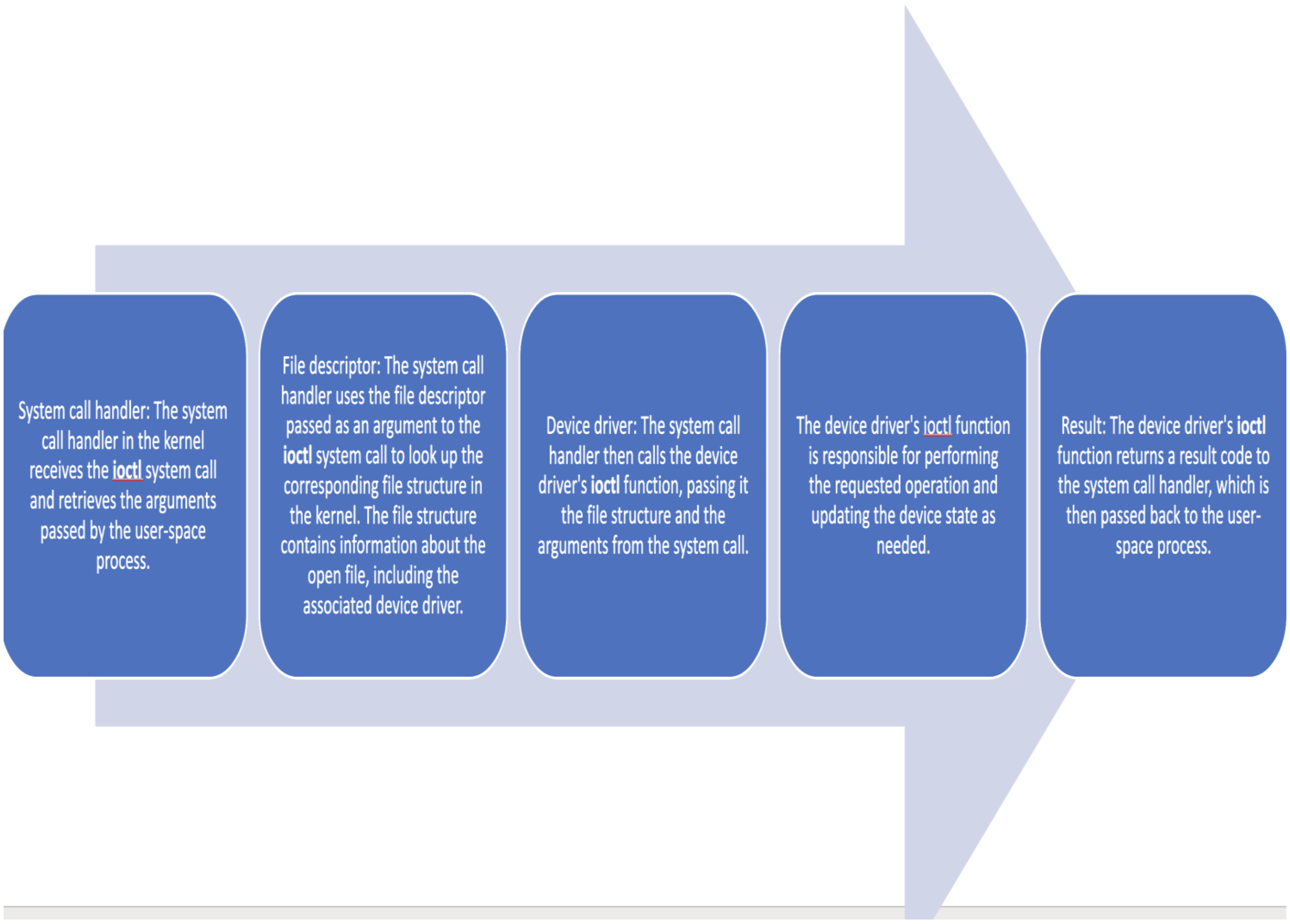
- **<u>argp (void *argp):</u>** This is a pointer to a data structure that is passed to the device driver with the ioctl request. The structure may contain additional information or data for the device driver to act upon, depending on the specific ioctl command being sent.

1.b.

<u>Kernel Space(REGISTRATION OF IOCTL FUCTION):</u>


Define an ioctl function in the file operations structure in ".unlocked_ioctl" field
Register the character device with the major number using "register_chrdev"
Define a unique identifier for the ioctl command
Implement the ioctl function
Perform switch statement to handle the different ioctl commands.
Load this kernel module
Use the 'ioctl' system call in the user space to communicate with the driver and trigger the registered ioctl function.

<u>Kernel Space(HANDLING OF IOCTL FUCTION):</u>


System call handler: The system call handler in the kernel receives the ioctl system call and retrieves the arguments passed by the user-space process.
File descriptor: The system call handler uses the file descriptor passed as an argument to the ioctl system call to look up the corresponding file structure in the kernel. The file structure contains information about the open file, including the associated device driver.
Device driver: The system call handler then calls the device driver's ioctl function, passing it the file structure and the arguments from the system call.
The device driver's ioctl function is responsible for performing the requested operation and updating the device state as needed.
Result: The device driver's ioctl function returns a result code to the system call handler, which is then passed back to the user-space process.

REGISTER THE IOCTL FUNCTION IN THE FILE OPERATIONS STRUCTURE

```
struct file_operations my_fops = {
   .unlocked_ioctl = my_ioctl,
   // Other file operations
};
```

DEFINE A UNIQUE IDENTIIFIER FOR THE IOCTL COMMAND

```
#define MY_IOCTL_CMD 0x123456
```

REGISTER THE CHARACTER DEVICE

```
ret = register_chrdev(MAJOR_NUM, DEVICE_NAME, &my_fops);
```

IMPLEMENT THE IOCTL FUNCTION WITH SWITCH CASE

```
long my_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
  switch (cmd) {
    case MY_IOCTL_CMD:
      // Perform desired operation
      break;
    default:
      return -EINVAL;
  }

  return 0;
}
```

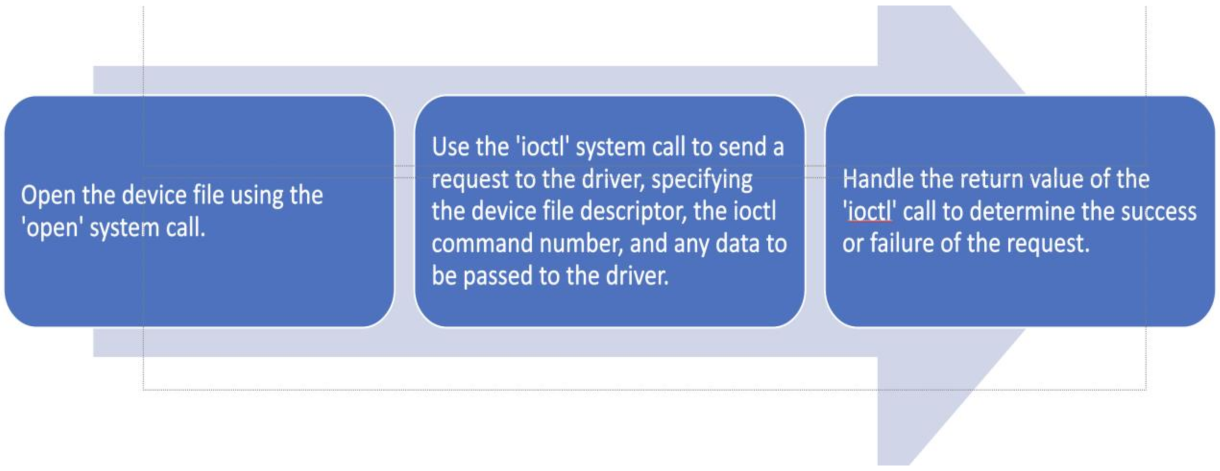
USER PROGRAM TO CALL THE IOCTL FUNCTION

```
int fd = open("/dev/mydevice", O_RDWR);
ioctl(fd, MY_IOCTL_CMD, arg);
close(fd);
```

User Space:

e.g.

```
int fd = open("/dev/mydevice", O_RDWR);
ioctl(fd, MY_IOCTL_CMD, arg);
```


Open the device file using the 'open' system call.
Use the 'ioctl' system call to send a request to the driver, specifying the device file descriptor, the ioctl command number, and any data to be passed to the driver.
Handle the return value of the 'ioctl' call to determine the success or failure of the request.

2.a. What KVM API calls does the hypervisor use to set up and run the guest OS?

KVM_CREATE_VM, KVM_SET_TSS_ADDR, KVM_SET_USER_MEMORY_REGION, KVM_CREATE_VCPU, KVM_RUN

2.b. List all the KVM API calls that the kvm-hello-world hypervisor uses and their purpose.

KVM_GET_API_VERSION: Gives the version of the KVM API in use. It is used to determine the features and capabilities of the KVM API.

KVM_CREATE_VM: Create a virtual machine. It returns a file descriptor that can be used to interact with the virtual machine.

KVM_SET_TSS_ADDR: Defines the physical address of a three-page region in the guest physical address space. 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. Sets the address of the **Task State Segment (TSS)** in the virtual machine.

It is a data structure used in x86 architecture operating systems to store information about a task, such as the current privilege level, stack pointers, and segment selectors. The TSS is used by the operating system to switch between tasks and to store the state of a task when it is not executing. The TSS is also used by the hardware to determine the location of the stack during an interrupt or exception, and to store the values of certain control registers.

KVM_SET_USER_MEMORY_REGION: This ioctl allows the user to create, modify or delete a guest physical memory slot .
Bits 0-15 of “slot” specify the slot id and this value should be less than the maximum number of user memory slots supported per VM.
Sets the memory regions that can be accessed by the virtual machine. It maps the physical memory of the host machine to the virtual memory of the virtual machine.

KVM_CREATE_VCPU: This KVM API request is used to create a virtual CPU for the virtual machine. It returns a file descriptor that can be used to interact with the virtual CPU. The vcpu id is an integer in the range [0, max_vcpu_id).

Each kvm_run structure corresponds to one vCPU, and we will use it to get the CPU status after execution.

KVM_RUN: This ioctl is used to run a guest virtual cpu. 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.

Transfers control from the host kernel to the guest operating system, and the virtual CPU starts executing the guest code.

KVM_GET_VCPU_MMAP_SIZE: The KVM_RUN ioctl communicates with userspace via a shared memory region. This ioctl returns the size of that region or in other words, this KVM API request returns the size of the memory region that is required for mapping the virtual CPU into the address space of the process that runs the virtual machine.

KVM_GET_REGS: This KVM API request retrieves the current state of the general purpose registers of the virtual CPU.

KVM_GET_SREGS: This KVM API request retrieves the current state of the special purpose registers of the virtual CPU.

KVM_SET_REGS: Writes the general-purpose registers into the vcpu.

KVM_SET_SREGS: Writes special registers into the vcpu

3a. Study these functions associated with these modes and explain how GVA is mapped to HPA in each of these modes.

Real mode: In real mode, the guest is limited to 1 MB of memory and doesn't have access to paging. The MMU is not used in this mode.

By default, the vCPU runs in **real mode**, which only executes **16-bit assembled code**.

Protected mode: Protected mode is be entered after the system software sets up one descriptor table and enables the Protection Enable (PE) bit in the control register 0 (CR0)

The protection enabled mode is made enabled by the setting the CR0 register in the following line.

sregs->cr0 |= CR0_PE; /* enter protected mode */

The **struct kvm_segment** is used to set up the guest system's segment registers and to control the memory segmentation of the guest operating system. It helps the hypervisor to translate the Guest Virtual Address (GVA) to the Host Physical Address (HPA) by defining the memory segments for the guest operating system.

Paged 32-bit mode: This mode extends protected mode to allow the guest to use paging using 32-bit instruction , which enables access to more memory. The MMU uses a table called the Page Directory to look up the physical address of a page, given its virtual address, and a table called the Page Table to map a virtual page to a physical page.

The PSE field of CR4 register is used to set the 4 MB page size instead of 4 KB.

The single page directory (4 MB page) to store mapping is being setup as well as the CR3 register to hold the base pgd address and segment registers in the following lines of code:

```
uint32_t pd_addr = 0x2000;
uint32_t *pd = (void *)(vm->mem + pd_addr);
/* A single 4MB page to cover the memory region */
pd[0] = PDE32_PRESENT | PDE32_RW | PDE32_USER | PDE32_PS;
/* Other PDEs are left zeroed, meaning not present. */
sregs->cr3 = pd_addr;
sregs->cr4 = CR4_PSE;
```

PSE field in the CR4 register is used to control the **Page Size Extension** which allows the processor to use larger page sizes for improved performance. When the PSE bit is set, the processor uses **4MB page sizes,** which can significantly reduce the number of page table entries that need to be accessed. When the PSE bit is clear, the processor uses smaller page sizes, typically **4KB**, which can negatively impact performance for applications that access large amounts of data.

This will also run in the protected mode on top of it.

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

Long mode: Long mode is similar to paged 32-bit mode, but is designed for 64-bit architectures. The MMU uses a similar mechanism to translate GVA to HPA as in paged 32-bit mode, but the Page Directory and Page Table have different structures and support larger address spaces.
This uses a memory management feature named PAE (Physical Address Extension), contains four kinds of tables: PML4T, PDPT, PDT, and PT

CR3 stores the base address of PML4T.

CR0_PE enables this mode to run in protected mode.

Each entry in the PML4T points to a PDPT, each entry in a PDPT to a PDT and each entry in a PDT to a PT. Each entry in a PT then points to the physical address. (4K paging)

But here in our implementation, we are using **2M paging**, with the PT (page table) removed. In this method the PDT entries point to physical address. [3 kinds of tables used]

The following lines of code sets up the 3 level paging as well as setting PAE and segment register for 64 bit]

uint64_t pml4_addr = 0x2000;

uint64_t *pml4 = (void *)(vm->mem + pml4_addr);

uint64_t pdpt_addr = 0x3000;

uint64_t *pdpt = (void *)(vm->mem + pdpt_addr);

uint64_t pd_addr = 0x4000;

uint64_t *pd = (void *)(vm->mem + pd_addr);

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

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

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

// PDE64_PS: stands for it's 2M paging instead of 4K. As a result, these page tables can map address below 0x200000 to itself (i.e. virtual address equals to physical address).

sregs->cr3 = pml4_addr;

The PAE field in the CR4 register is used to control the Physical Address Extension feature which allows the processor to address more than 4GB of physical memory. When the PAE bit is set, the processor uses 36-bit physical addresses, which allows it to access up to 64GB of physical memory. When the PAE bit is clear, the processor uses 32-bit physical addresses, which limits its physical memory address space to 4GB.

sregs->cr4 = CR4_PAE;

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

sregs->efer = EFER_LME | EFER_LMA;

setup_64bit_code_segment(sregs);

3.b. What is the size of the memory allocated to the guest VM in long mode?
<u>0x200000 bytes or 2^21 Bytes = 2 MB during vm_init(vm,memory size) call</u>

3.c. Which code line inside the hypervisor sets up the guest's page table?
What is the KVM API call used to make the KVM aware of the guest's page table?

For paged 32-bit mode : Line 335 calls the function to setup the guest page table with 32 bit entry.
setup_paged_32bit_mode(struct vm *vm, struct kvm_sregs *sregs)
For long mode: Line 416 calls the function setup the guest page table with 64 bit entry
setup_long_mode(struct vm *vm, struct kvm_sregs *sregs)

The KVM API call used to make the KVM aware of the guest's page table is
<u>KVM_SET_USER_MEMORY_REGION and before that using mmap() which returns a pointer to the starting virtual address of user space address.</u>

vm->mem = mmap(NULL, mem_size, PROT_READ | PROT_WRITE, AP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE, -1, 0);

if the guest_phys_addr field is set to 0, the mapping of the memory region to the guest physical address is performed dynamically by the KVM API at runtime.

It maps the physical memory of the host machine to the virtual memory of the virtual machine. (V2M : VA of guest -> Host PA)

3.d. Explain the role of CR4_PAE flag while setting the guest’s page table in the long mode.

The CR4_PAE (Physical Address Extension) flag in the guest's CR4 register is used to enable or disable the Physical Address Extension (PAE) feature in the long mode of x86-64 architecture. PAE allows a 32-bit operating system to use more than 4GB of physical memory. When setting up the guest's page table in the long mode, the hypervisor needs to set the CR4_PAE flag to enable PAE and ensure that the guest can access all of the allocated memory. If the CR4_PAE flag is not set, the guest may only be able to access 4GB of memory and the rest will be inaccessible. By setting the CR4_PAE flag, the hypervisor enables the guest to use a 36-bit physical address space, allowing it to access up to 64GB of memory.

4.a. At what (guest virtual) address does the guest start execution when it runs? Where is this address configured?

The starting address is specified by vm->mem returned as a result of :
vm->mem = mmap(NULL, mem_size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE, -1, 0)

This address is configured in RIP/EIP (Instruction Pointer) of struct kvm_regs

In the context of a virtual machine, setting the instruction pointer to zero may indicate that the virtual machine is starting up or being reset, and that the first instruction to be executed will come from a known and specified location in memory.

4.b. At which line of code does the guest VM start execution, and what is the KVM API used to start the guest execution?

The KVM API call to start guest execution is **KVM_RUN**. And is present in the line 162 in run_vm() function.

5. The guest uses the **outb** function to write an 8-bit value into a serial port.

a. How is a value written to or read from a serial port in the guest.c program?

This is done using inline assembly statement using “in” to read from a serial point and “out” to write to a serial port using “asm” from the kvm_run shared region between hypervisor and guest.

The KVM_RUN ioctl communicates with user space via a shared memory region.

Syntax for “out” :

asm("out %0,%1" : /* empty */: "a" (value), "Nd" (port): "memory");

Here, **value** is the value to be outputted, and **port** is the port number.

The first argument to the **out** instruction is the value to be outputted, and the second argument is the port number.

The **memory** constraint specifies that the memory may be modified by the assembly instruction, and the **“a” constraint** specifies that the **value** argument should be placed in the **%eax** (or **%rax**) register. The “**Nd” constraint** specifies that the **port** argument should be placed in the **%edx** (or **%rdx**) register, and it should be treated as a 32-bit (or 64-bit) signed integer.

Syntax for “in”:

asm("in %1, %0": "=a"(ret): "Nd"(port) : "memory");

Here “return” will store the value read from the serial port by the guest os

The **"=a" (value)** specifies that **value** will be an output and stored in the **%eax** register. The **"Nd" (port)** specifies that **port** will be an input and passed to the **%dx** register.

5.b. How does the hypervisor access and print the string?

The IO read/write in guest is a privileged instruction and will trigger a VM EXIT to the hypervisor.

The **kvm_run** data structure of vcpu in a KVM-based virtualized environment. It serves as a communication channel (shared memory region) between the guest and the hypervisor, and it is used to exchange information about the state of the virtual machine and the virtual devices that are being emulated.

Hypervisor------>Puts data in kvm_run [SHARED MEMORY] ------->fetched by guest using IN(triggered while IO EXIT)

Guest ----> Puts a value in kvm_run using OUT [SHARED MEMORY] -----> Hypervisor (fetches it while handling IO EXIT)

The hypervisor will check the EXIT reason and handle the VM EXIT and use fwrite to write to stdout

switch (vcpu->kvm_run->exit_reason) - will check the exit reason and handle based on it. In this case , it is the IO interrupt

```
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;
}
```

The guest OS will call the outb() which will be a hypercall to hypervisor and VMM will handle the IO EXIT by printing the string passed by the guest OS.

5.c. Code in *guest.c* uses the value 42, what is this value used for?

To make a HALT to trigger a VM EXIT and chexck whether vcpu is working fine or not.
The guest code stores the value 42 to memory address 0X400.
The asm "hlt" instruction copies the value 42 to eax register and is used to make a exit from VM mode. The hypervisor then calls the KVM_GET_REGS and checks the rax value, if it is not 42, then prints "Wrong Result" and if it is 42, then use the memcpy() to copy the value from memory address 0X400 to a variable "memval" and checks the condition of whether it is 42 or not.

Thus, it makes sure that the vcpu is working correctly with proper register and memory values.

The background instructions when
*(long *) 0x400 = 42;

asm("hlt" : /* empty */ : "a" (42) : "memory"); is called

The following is done :

```
.code 16
.global code16,code16_end
guest16:

        movw $42, %ax [moving the immediate operand 42 to accumulator (ax)
register]

   movw %ax, 0x400

   hlt

guest16_end:
```