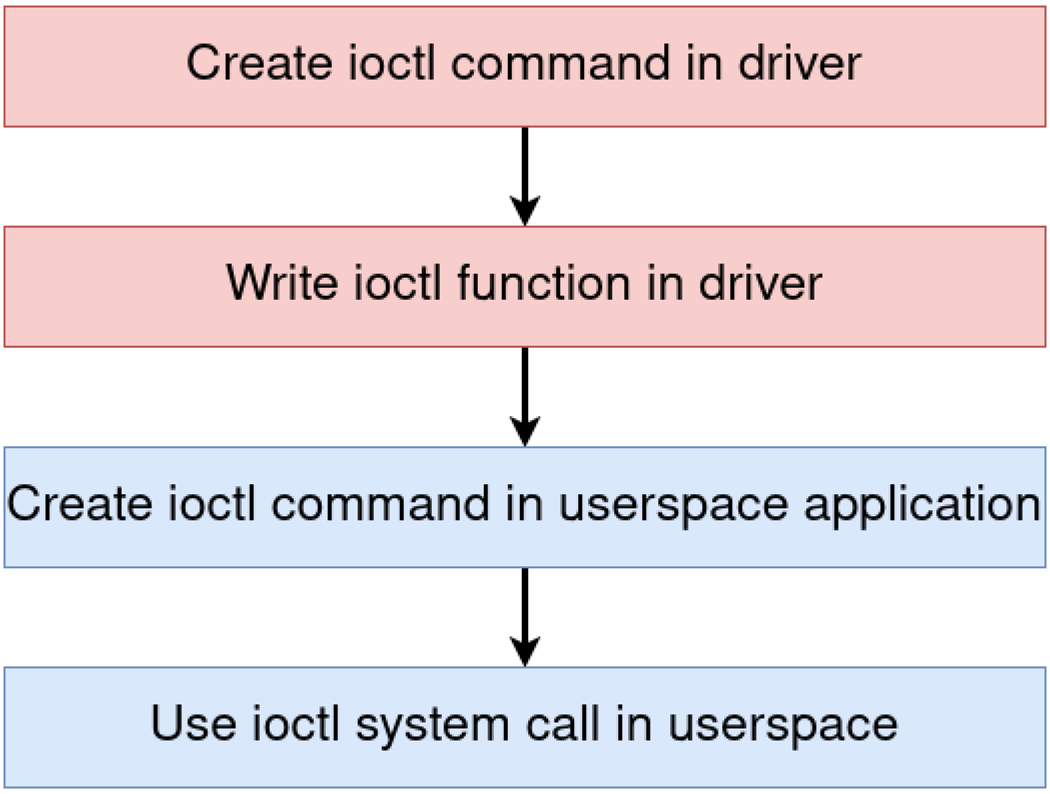
1. Familiarize yourself with ioctl system call.

- **a) Describe the arguments (and purpose of each) of an ioctl system call?**
    - Controling anything related to i/o devices can be done using the generic ioctl syscall, by using its two arguments carefully.
    - int fd: open file descriptor of which underlying device parameters are to be manipulated.
    - unsigned long request: this is a request code which is device dependent
    - untyped pointer to memory
    - return 0 on success, few requests use return value as an output parameter

- **b) Draw a diagram listing the steps in the kernel and in the user space to register and use an ioctl call.**
    - Kernel space steps:
        - Create ioctl command in the driver
        - Write ioctl function in driver
    - User space steps:
        - Create ioctl command in userspace application
        - Use ioctl system call in a userspace


Create ioctl command in driver
Write ioctl function in driver
Create ioctl command in userspace application
Use ioctl system call in userspace

2. The KVM API

- **a) What KVM API calls does the hypervisor use to set up and run the guest OS?**
    - KVM_GET_API_VERSION
        - This gets the API version of KVM
    - KVM_CREATE_VM
        - This returns file descriptor to a VM
    - KVM_SET_TSS_ADDR
        - This ioctl defines the physical address of a three-page region in the guest physical address space.
    - KVM_SET_USER_MEMORY_REGION
        - This ioctl allows the user to create, modify or delete a guest physical memory slot
    - KVM_RUN
        - VM execution starts here, after this call kernel uses this thread to execute guest code on vCPU.
- **b) List all the KVM API calls that the kvm-hello-world hypervisor uses and their purpose.**
    - KVM_GET_API_VERSION
        - This gets the API version of KVM
    - KVM_CREATE_VM
        - This returns file descriptor to a VM
    - KVM_SET_TSS_ADDR
        - This ioctl defines the physical address of a three-page region in the guest physical address space.
    - KVM_SET_USER_MEMORY_REGION
        - This ioctl allows the user to create, modify or delete a guest physical memory slot
    - KVM_RUN
        - VM execution starts here, after this call kernel uses this thread to execute guest code on vCPU.
    - Above list is for setting up and running the guestOS
    - KVM_TRANSLATE
        - This ioctl call is used to translate guest virtual address(gva) to guest physical address(gpa)
        - linear_address is passed to the structure and physical_address is output
    - KVM_GET_CLOCK
        - This call is used to get current clocktime of vm. A structure kvm_clock_data is passed to the call and we get the current time.
    - KVM_GET_REGS and KVM_SET_REGS
        - This ioctl call gets or sets the register context from vCPU
        - rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8, r9, r10, r11, r12, r13, r14, r15;
    - KVM_GET_SREGS and KVM_SET_SREGS
        - This ioctl call gets or sets the SEGMENT registers context from vCPU
        - cs, ds, es, fs, gs, ss, cr0, cr2, cr3, cr4, cr8

3. The memory layout and how MMU converts the GuestVirtual Address (GVA) to the Host Physical Address (HPA) depends on the mode the guest is running. These modes are named real mode, protected mode, paged 32-bit mode and long mode in the main function of the kvm-hello-world.c file.

- **a) Study these functions associated with these modes and explain how GVA is mapped to HPA in each of these modes.**
  - KVM_CREATE_VM: Physical address size is initialized here
  - KVM_SET_USER_MEMORY_REGION : This API call sets GVA to HPA Mapping dynamically at run time if guest_phys_addr is set to 0
  - Line 112

- **b) What is the size of the memory allocated to the guest VM in long mode?**
  - 2^{21} Bytes: 2MB
  - vm_init(&vm, 0x200000);
  - Second parameter of vm_init is mem_size which is used to initialise vm->mem

- **c) Which code line inside the hypervisor sets up the guest's page table?**
  - For real mode and protected mode; no page table
  - For **paged_32bit_mode, single level page table**
    - CR3 register is set to pd_addr which has address to first level of page.
    - Line 335 calls function setup_paged_32bit_mode
    - This function has following code which sets up page table directory address, CR3, CR2, CR0 registers

    - 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;
sregs->cr0
= CR0_PE | CR0_MP | CR0_ET | CR0_NE | CR0_WP | CR0_AM | CR0_PG;
sregs->efer = 0;
```

  - For **long mode, 3 level page table**
    - Function setup_long_mode function is being called which sets up the page table directory address, CR3, CR2, CR0 registers along with multiple level of paging
    - CR3 is set to pml4_addr which is first level of paging

```
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;

sregs->cr3 = pml4_addr;
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;
```

- **d) Explain the role of CR4_PAE flag while setting the guest's page table in the long mode.**
    - PAE stands for Page Address Extension. CR4_PAE flag enables page address extension feature in X86. Normally guest OS will have 32 bit addresses. I.e. it can use Hence host os can set the PAE flag for guest os in order to have longer virtual addresses in guest OS.

4. VM execution

- a) At what (guest virtual) address does the guest start execution when it runs? Where is this address configured?
    - Guest execution is started at guest virtual address 0 as RIP register is set to 0
    - regs.rip = 0;

- b) At which line of code does the guest VM start execution, and what is the KVM API used to start the guest execution?
    - Line 162
    - if (ioctl(vcpu->fd, KVM_RUN, 0) < 0) {

```
perror("KVM_RUN");
exit(1);
}
```

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?
    - Using out and in functions. This is a family of functions which is used to do low level port input and output. The out* functions do port output, the in* functions do port input. T he b-suffix functions are byte-width
    - In this guest.c code, we have used outb, out, inb function to do port output and input.

- b) How does the hypervisor access and print the string?
    - Guest.c writes the string to port number 0xE9 (233) using outb function using following code

    - Using outb function we pass the memory address of str, which is guest virtual address (gva). We need host virtual address (hva) corresponding to the gva. We use VM_TRANSLATE to get gpa and then add that offset to vm→mem ro access the hva address. Then we using that address in printf function with %s specifier, which in return prints the string

    - Hypervisor uses following code to access the gva sent from guest.

        -
          ```
          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;
              }
          ```
        - This code runs when vm io exit case is there, I.e   KVM_EXIT_IO
        - When io.direction is outb, the **data_offset** has the data to output.

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

    - 42 is stored at memory location 0x400 from guest program

    - When kvm_exit reason is KVM_EXIT_HLT

        - Then value at memory location 0x400 i.e 42 that will be copied to memval varibale using memcpy command

        - The value 42 is also stored at rax register

        - Code is as follows:

        -
          ```
          memcpy(&memval, &vm->mem[0x400], sz);
              if (memval != 42) {
              printf("Wrong result: memory at 0x400 is %lld\n",
              (unsigned long long)memval);
              return 0;
          ```

```
}
```

-
  ```
  for (;;)
      asm("hlt" : /* empty */ : "a" (42) : "memory");
  ```

- Using this check, we can verify if the vm->mem object is created properly and if we are able to store and retrieve in the vm’s memory. Same with vCPU. We store 42 to a “A” register and try to retrieve it back. So that we can verify if vCPU object is working.