Assignment #3
Hello KVM!

0. Introduction

This assignment aims to understand how QEMU+KVM support virtual machines using hardware-assisted virtualization as a Type2 (hosted) hypervisor setup.

I. Recap how hardware virtualisation works from the CS 695 lectures.
II. Modern hypervisors like QEMU, Virtualbox or VMWare run as Linux user space processes and communicate with the Linux kernel module KVM, which uses hardware virtualization for resource virtualization and providing the VM abstraction.

    References for the Linux KVM API are at: Link 1 and Link 2.
III. The following command should install the QEMU hypervisor and KVM kernel module on an Ubuntu-based system.

```
sudo apt-get install qemu-kvm libvirt-daemon-system libvirt-clients
bridge-utils
```

    Please look into the Ubuntu Documentation for more details.
IV. This assignment extends the kvm-hello-world example code given at this link.
    Do NOT clone this git repo, as we have changed the source code.
    Use the modified version of the code accompanying this description.
V. Link to skeleton code for the assignment is here.

1. Understanding the kvm-hello-world hypervisor

The kvm-hello-world hypervisor is implemented in the source files `kvm-hello-world.c` and the guest OS source code is in `guest.c`. For now the guest is a single stream of instructions, no separation between guest user space and the guest OS.

Understand source code and functions in these two files to get familiar with kvm-hello-world hypervisor uses the KVM API to run the guest code in a virtualized environment.

Answer the following questions in a file named Answers.txt (.docx, .pdf etc. file formats are also allowed) inside your assignment submission directory.

1. Familiarize yourself with *ioctl* system call.
   Refer to ioctl man pages and online examples for more info.
   Online resources: Link1, Link2 and an ioctl driver
   a. Describe the arguments (and purpose of each) of an ioctl system call?
   b. Draw a diagram listing the steps in the kernel and in the user space to register and use an ioctl call.
2. The KVM API
   a. What KVM API calls does the hypervisor use to set up and run the guest OS?
   b. List all the KVM API calls that the kvm-hello-world hypervisor uses and their purpose.

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.

    b. What is the size of the memory allocated to the guest VM in long mode?
    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?
    d. Explain the role of CR4_PAE flag while setting the guest's page table in the long mode.
4. VM execution
    a. At what (guest virtual) address does the guest start execution when it runs? Where is this address configured?
    b. At which line of code does the guest VM start execution, and what is the KVM API used to start the guest execution?
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?
    b. How does the hypervisor access and print the string?
    c. Code in *guest.c* uses the value 42, what is this value used for?

NOTE: This page table in the long mode of kvm-hello-world hypervisor translates to the guest's physical address to be the same as the guest's virtual address, so that the user space hypervisor can read from any guest's virtual address without actually walking through the page table.
You will need to read from the guest's virtual address inside the hypervisor in the assignment. Understand the page table structure carefully in the long mode to understand how the *gva* to *gpa* mapping and how the *gva* to *hva* mapping is set up and how the mappings work for gva access.

2. Implementing new hypercalls

In this section, we will add new hypercalls from the guest to the hypervisor. The hypercalls will communicate between the guest and the hypervisor using a serial port. The **in/out** instructions to read from and write to a serial port are privileged instructions which cause the guest OS to exit to the hypervisor.
KVM handles this IO exit and jumps to the user space hypervisor for further handling of the hypervisor.

The user space handler can determine the exit reason, direction of the data, i.e., read or write from the **io.direction** field equal to KVM_EXIT_IO_OUT or KVM_EXIT_IO_IN and has a pointer to a shared memory area for data transfers (io.data_offset).

For details check the kvm.h and the run_kvm function in the kvn_hello_world.c source file.

NOTE:
An example hypercall **hc_print8bit** is implemented in the skeleton code provided.

Further, the assembly code to write and read 32-bit values to and from a serial port inside the guest VM is also provided.

```
static inline void outb(uint16_t port, uint32_t value) {
```

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

static inline uint32_t inb(uint16_t port) {
  uint32_t ret;
  asm("in %1, %0" : "=a"(ret) : "Nd"(port) : "memory" );
  return ret;
}
```

Implement the following new hypercalls

1. **void hc_print32bit(uint32_t val)**
   Passes a 32-bit value to the hypervisor, and the hypervisor prints it on the terminal. Please print a single newline “\n” or “endl” after printing the 32-bit value inside the hypervisor.
2. **uint32_t hc_numexits()**
   Returns the number of times the guest VM has exited to the hypervisor (VM_EXIT) from the beginning of its execution. The hypervisor should maintain the count and return it to the guest. Since the guest can’t print directly on the terminal, please use HC_print32bit implemented in the previous step to print it out.
3. **void hc_printstr(char *str)**
   This will print a string pointed by str. But unlike the example code given in the guest.c, the HC_printStr hypercall should make only one exit to the hypervisor. To achieve this, you must pass the address of the string to the hypervisor, which will fetch the string from the guest’s memory. Use HC_numExits to count the number of exists before and after the hypercall. The count should defer by 1.
   NOTE: You should not add any extra newline. If the string passed by the guest has a newline, it should be printed.
4. **uint32_t hc_gvatohva(uint32_t gva)**
   This returns the Host Virtual Address (HVA) corresponding to a given Guest Virtual Address (GVA). If the host asks to translate an invalid GVA for which no HVA exists, the hypervisor should print “Invalid GVA” on the terminal and return 0.
   Demonstrate at least one valid and one invalid GVA to HVA conversion. Please use hc_print32bit to print the HVA value.

NOTE: To implement multiple hypercalls, you may use a different serial port for each hypercall or a struct with a field with a hypercall number and all arguments to the hypercall. In the second method, the struct will be known to the guest, and the hypervisor and the guest will pass the address to the struct for every hypercall with the correct hypercall number and the arguments set. The hypervisor can now read the hypercall number and the argument from the guest’s memory. The implementation choice is left to you.

Get more information from the KVM API

The hypervisor may communicate with the KVM API to get information such as the KVM API version etc. In this section, we will implement a few hypercalls which ask for information from the hypervisor. The hypervisor will fetch the info from KVM API and send it back to the guest.

1. uint32_t **hc_getTimestamp()**
   Gets the current timestamp of the kvm clock as seen by the current guest. Please use hc_print32bit to print the value.

Submission

You have to submit your modified guest.c hello-world-kvm.c and Answers.txt for evaluation. Please create a folder <your-roll-no>-CS695-a3 and put the above three files into the folder. Create a .tar.gz of the folder and submit a single <your-roll-no>-CS695-a3.tar.gz file in Moodle. You may use the following command to create the tarball.

```
tar -czvf <your-roll-no>-CS695-a3.tar.gz <your-roll-no>-CS695-a3/
```

Due date: 15th Feb. 5pm