Homework3

Name	ID	Class
Qing Liu	6130116184	Class 164 of Computer Science and Technology

Requirements

> The hypercall you write should take one argument and output the information about virtual CPU in KVM.

> The prototype for your hypercall will be the following. The argument vcpu_id contains the CPU id in your VM.

```
int vcpu_info(int vcpu_id);
```

> Your hypervisor should output the following information to ftrace in your KVM host based on the input VCPU ID provided by the VM.

```
pid: the corresponding PID of the VCPU thread in KVM host
gp_regs: the values of the general purpose registers for the virtual CPU
num_exits: number of vm exits from the VCPU
```

I use the linux distribution Ubuntu in VMware, and the version is 16.04 LTS. And the guest in vm is Ubuntu 14.04. The linux kernel version which was used to be recompiled is 5.0.11, we can download it from here. It is faster than git from github. It need to be done in both guest and host. And we need to compile and install in both guest and host.

```
$ cd /usr/src
$ sudo wget https://mirrors.tuna.tsinghua.edu.cn/kernel/v5.x/linux-5.0.11.tar.xz
$ sudo tar -xvf linux-5.0.11.tar.xz
$ cd linux-5.0.11
```

Add the hypercall in host

- Define the hyper call number in `include/uapi/linux/kvm_para.h`

```
+ #define KVM_HC_VCPU_INFO               11
```

- Add handler for KVM_HC_VCPU_INFO in `arch/x86/kvm/x86.c`

```
/**
 * @brief      handler for KVM_HC_VCPU_INFO hypercall
 * @param[in]  vcpu     The vcpu
 * @param[in]  vcpu_id  The vcpu identifier
 * @return     status
 */
static int kvm_vcpu_info(struct kvm_vcpu *vcpu, ulong vcpu_id)
{
    int level;
    struct kvm *kvm = vcpu->kvm;
    struct kvm_vcpu *vcpup = NULL;

    struct upid *vcpu_upid = NULL;
    struct pid *vcpu_pid;

    ulong a0, a1, a2, a3;
    u64 vm_exits;

    vcpup = kvm_get_vcpu_by_id(kvm, vcpu_id);

    if (vcpup == NULL) {
        trace_printk("unknown cpu id: %lu\n", vcpu_id);
        return -1;
    }

    trace_printk("vcpu id: %lu\n", vcpu_id);

    vcpu_pid = vcpup->pid;

    level = vcpu_pid->level;
    trace_printk("vcpu pid level: %d\n", level);

    vcpu_upid = &vcpu_pid->numbers[level];
    trace_printk("pid: %d\n", vcpu_upid->nr);

    a0 = kvm_register_read(vcpu, VCPU_REGS_RBX);
    a1 = kvm_register_read(vcpu, VCPU_REGS_RCX);
    a2 = kvm_register_read(vcpu, VCPU_REGS_RDX);
    a3 = kvm_register_read(vcpu, VCPU_REGS_RSI);

    trace_printk("regs: rbx=%lu, rcx=%lu, rdx=%lu, rsi=%lu\n", a0, a1, a2, a

    vm_exits = vcpu->stat.exits;
    trace_printk("vm exits nums: %llu\n", vm_exits);

    return 0;
}
```

- Add a case in `kvm_emulate_hypercall` function in `arch/x86/kvm/x86.c`

```
switch (nr)
{

    ...
    case KVM_HC_VCPU_INFO:
        ret = kvm_vcpu_info(vcpu, a0);
        break;
    default:
        ret = -KVM_ENOSYS;
        break;
}
```

After finish this task, we need to compile the kernel and install it.

Add a new syscall in guest

- Enter the linux source directory, define the hyper call number in `include/uapi/linux/kvm_para.h`

```
+ #define KVM_HC_VCPU_INFO                11
```

- Register in `arch/x86/entry/syscalls/syscall_64.tbl`

```
548     64      vcpu_info               sys_vcpu_info
```

- Add declaration in `include/linux/syscalls.h`

```
asmlinkage long sys_vcpu_info(void);
```

- Add a new directory named vcpu_info with the command `mkdir vcpu_info`, and enter the directory, edit three new files -- `sys_vcpu_info.c`, `vcpu_info.h` and `Makefile`.

  `vcpu_info.h`:

```
asmlinkage long sys_vcpu_info(void);
```

  `vcpu_info.c`:

```
#include <linux/kernel.h>
#include <linux/syscalls.h>
```

```
#include <linux/init.h>
#include <linux/linkage.h>
#include "vcpu_info.h"
#include <uapi/linux/kvm_para.h>
#include <linux/cpumask.h>

asmlinkage long sys_vcpu_info(void)
{
    uint cpu_id;

    for_each_online_cpu(cpu_id) {
        kvm_hypercall1(KVM_HC_VCPU_INFO, cpu_id);
    }

    return 0;
}
```

`Makefile`:

```
obj-y := sys_vcpu_info.o
```

- Add `vcpu_info` to `core-y` target in `Makefile` in source directory

```
- core-y     += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/
+ core-y     += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ vcpu_in
```

Recompile the kernel and install(host and guest)

- copy config to your kernel source

```
$ cp /boot/config-$(uname -r) .config
$ yes "" | make oldconfig
```

- Install proper packages for the kernel compile

```
$ sudo apt-get install libssl-dev bc libncurses5-dev
$ sudo apt-get install build-essential openssl
$ sudo apt-get install zlibc minizip
$ sudo apt-get install libidn11-dev libidn11
$ sudo apt-get install flex bison
$ sudo apt-get install libelf-dev
```

- Run the following command to configure your kernel

```
$ make mrproper
$ make clean
$ make menuconfig
```

To enable ftrace in your KVM host, go to Kernel Hacking and select Tracers. Then enter Tracers, select both Kernel Function Tracer and Kernel Function Graph Tracer. Finally, save the config and exit.

- After you add your hypercall to your Linux/KVM source, use the following command to compile the kernel.

```
$ make -j4
```

`-j4` represents the number of the cpu threads supported. And this progress will take a long time. And many warnings or even errors will occurred and the errors will make influences to the next step.

- Finally, use the following command to install the new kernel to your system. The new kernel will be loaded in the next boot.

```
$ make modules_install
$ make install
```

Test the hypercall

- Write a program to call the syscall in guest, compile and extcute it.

```
/**
 * test.c
 * This is a test for the hypercall
 */
#include <stdio.h>
#include <unistd.h>

int main()
{
  long int r = syscall(548);
  printf("system call sys_vcpu_info result %ld\n", r);
  return 0;
}
```

```
$ gcc test.c -o test
$ ./test
```

Here is the output information:

```
system call sys_vcpu_info result 0
```

- Compile the program and execute it in guest, and check the trace in host

```
$ cat /sys/kernel/debug/tracing/trace
```

Here is the content for the trace output:

```
# tracer: nop
#
# entries-in-buffer/entries-written: 10/10   #P:4
#
#                              _------=> irqs-off
#                             / _-----=> need-resched
#                            | / _----=> hardirq/softirq
#                            || / _---=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
 qemu-system-x86-12398 [003] .... 10747.677382: kvm_emulate_hypercall: vcpu
 qemu-system-x86-12398 [003] .... 10747.677401: kvm_emulate_hypercall: vcpu
 qemu-system-x86-12398 [003] .... 10747.677401: kvm_emulate_hypercall: pid:
 qemu-system-x86-12398 [003] .... 10747.677402: kvm_emulate_hypercall: regs:
 qemu-system-x86-12398 [003] .... 10747.677402: kvm_emulate_hypercall: vm ex
 qemu-system-x86-12398 [003] .... 10747.677415: kvm_emulate_hypercall: vcpu
 qemu-system-x86-12398 [003] .... 10747.677415: kvm_emulate_hypercall: vcpu
 qemu-system-x86-12398 [003] .... 10747.677415: kvm_emulate_hypercall: pid:
 qemu-system-x86-12398 [003] .... 10747.677416: kvm_emulate_hypercall: regs:
 qemu-system-x86-12398 [003] .... 10747.677416: kvm_emulate_hypercall: vm ex
```

Thoughts

To be honest, I think this is a hard work so far. I will continue to learn how to fix this problem. Here are some new points I got at this period of time when doing this homework.

- **KVM**

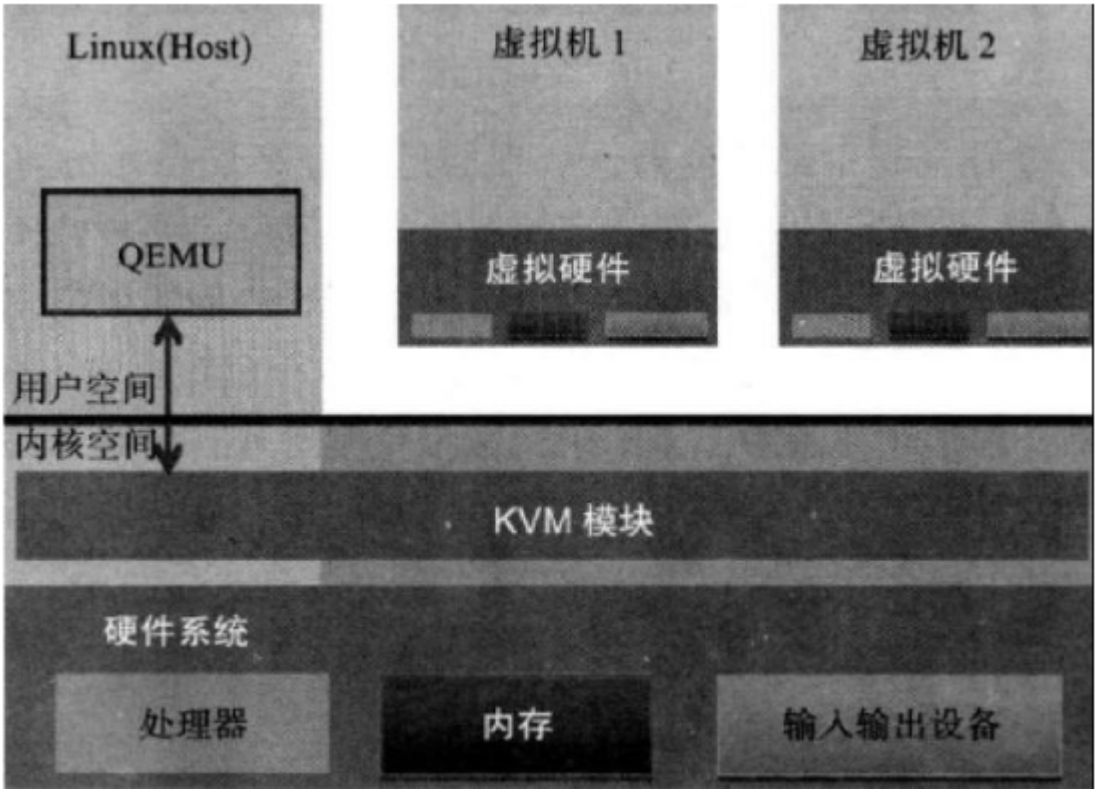
KVM (for Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko.

Using KVM, one can run multiple virtual machines running unmodified Linux or Windows images. Each virtual machine has private virtualized hardware: a network card, disk, graphics adapter, etc.

KVM is open source software. The kernel component of KVM is included in mainline Linux, as of 2.6.20. The userspace component of KVM is included in mainline QEMU, as of 1.3.

KVM itself does not perform any hardware simulation. User-space programs are required to set the address space of a client virtual server through the / dev / KVM interface, provide it with analog I/O, and map its video display back to the host's display screen. The current application is QEMU.

- **KVM Architecture**



  - Client systems: including CPU(vCPU), memory, driver(Console, network card, I/O device driver, etc.), are operated in a restricted CPU mode by KVM.
  - KVM: running in kernel space, providing virtual level CPU and memory, and client I/O interception. Guest's I/O is intercepted by KVM and handed over to QEMU.
  - QUME: the modified QEMU code used by KVM virtual machine runs in user space, provides hardware I/O virtualization, and interacts with KVM through IOCTL/dev/kvm devices.

- **QUME-KVM**

In fact, QEMU is not a part of KVM. It is a virtualization system implemented by pure software, so its performance is low. However, QEMU code includes a whole set of virtual machine

implementations, including processor virtualization, memory virtualization, and virtual device simulation (network card, graphics card, storage controller and hard disk) that KVM needs to use. In order to simplify the code, KVM has been modified on the basis of QEMU. During VM operation, QEMU enters the kernel through system calls provided by KVM module, and KVM is responsible for putting the virtual machine in a special mode of operation. When virtual machines perform I/O operations, KVM returns QEMU from the last system call exit, and QEMU is responsible for parsing and simulating these devices. From the point of view of QEMU, it can also be said that QEMU uses the virtualization function of KVM module to provide hardware virtualization acceleration for its virtual machine. In addition, the configuration and creation of virtual machines, the virtual devices on which virtual machines operate, the user environment and interaction when virtual machines run, and some specific technologies of virtual machines, such as dynamic migration, are all implemented by QEMU itself.

- **The creating progress of KVM**

1. qemu-kvm controls virtual machines through a series of ICOTL commands of / dev / kvm, such as

```
open("/dev/kvm", O_RDWR|O_LARGEFILE)    = 3
ioctl(3, KVM_GET_API_VERSION, 0)        = 12
ioctl(3, KVM_CHECK_EXTENSION, 0x19)     = 0
ioctl(3, KVM_CREATE_VM, 0)              = 4
ioctl(3, KVM_CHECK_EXTENSION, 0x4)      = 1
ioctl(3, KVM_CHECK_EXTENSION, 0x4)      = 1
ioctl(4, KVM_SET_TSS_ADDR, 0xfffbd000)  = 0
ioctl(3, KVM_CHECK_EXTENSION, 0x25)     = 0
ioctl(3, KVM_CHECK_EXTENSION, 0xb)      = 1
ioctl(4, KVM_CREATE_PIT, 0xb)           = 0
ioctl(3, KVM_CHECK_EXTENSION, 0xf)      = 2
ioctl(3, KVM_CHECK_EXTENSION, 0x3)      = 1
ioctl(3, KVM_CHECK_EXTENSION, 0)        = 1
ioctl(4, KVM_CREATE_IRQCHIP, 0)         = 0
ioctl(3, KVM_CHECK_EXTENSION, 0x1a)     = 0
```

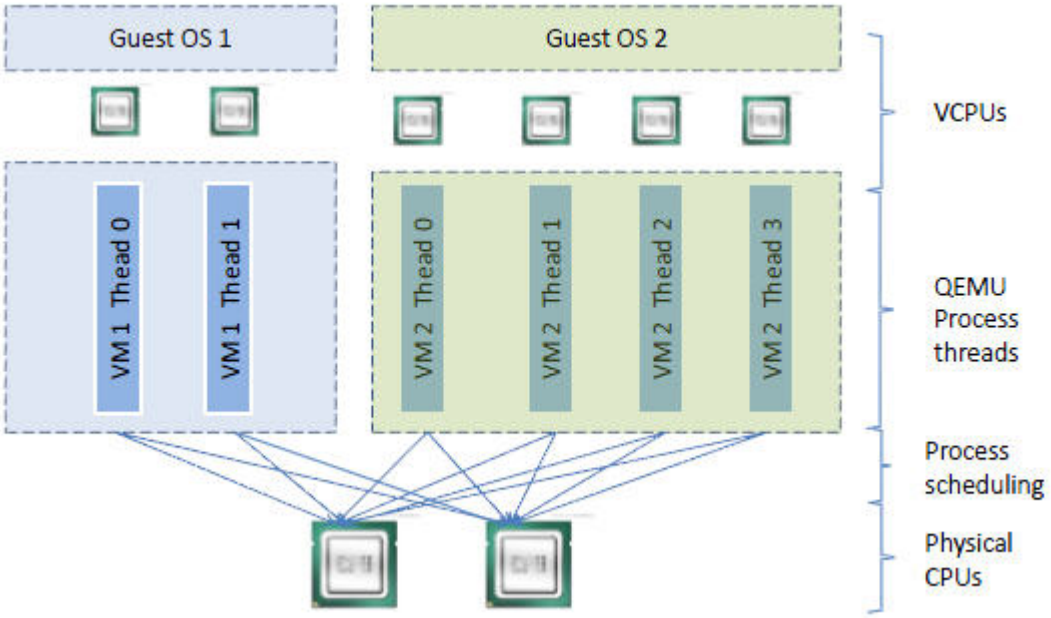
2. A KVM virtual machine, a Linux qemu-kvm process, is scheduled by the Linux process scheduler as other Linux processes.

3. KVM virtual machine includes virtual memory, virtual CPU and virtual machine I/O devices. The virtualization of memory and CPU is implemented by KVM kernel module, and the virtualization of I/O devices is implemented by QEMU.

4. The memory of the KVM host system is part of the address space of the qumu-kvm process.

5. The vCPU of the KVM virtual machine runs as a thread in the context of the qemu-kvm process.

Logical relationships among vCPU, QEMU process, LInux process scheduling and physical CPU:

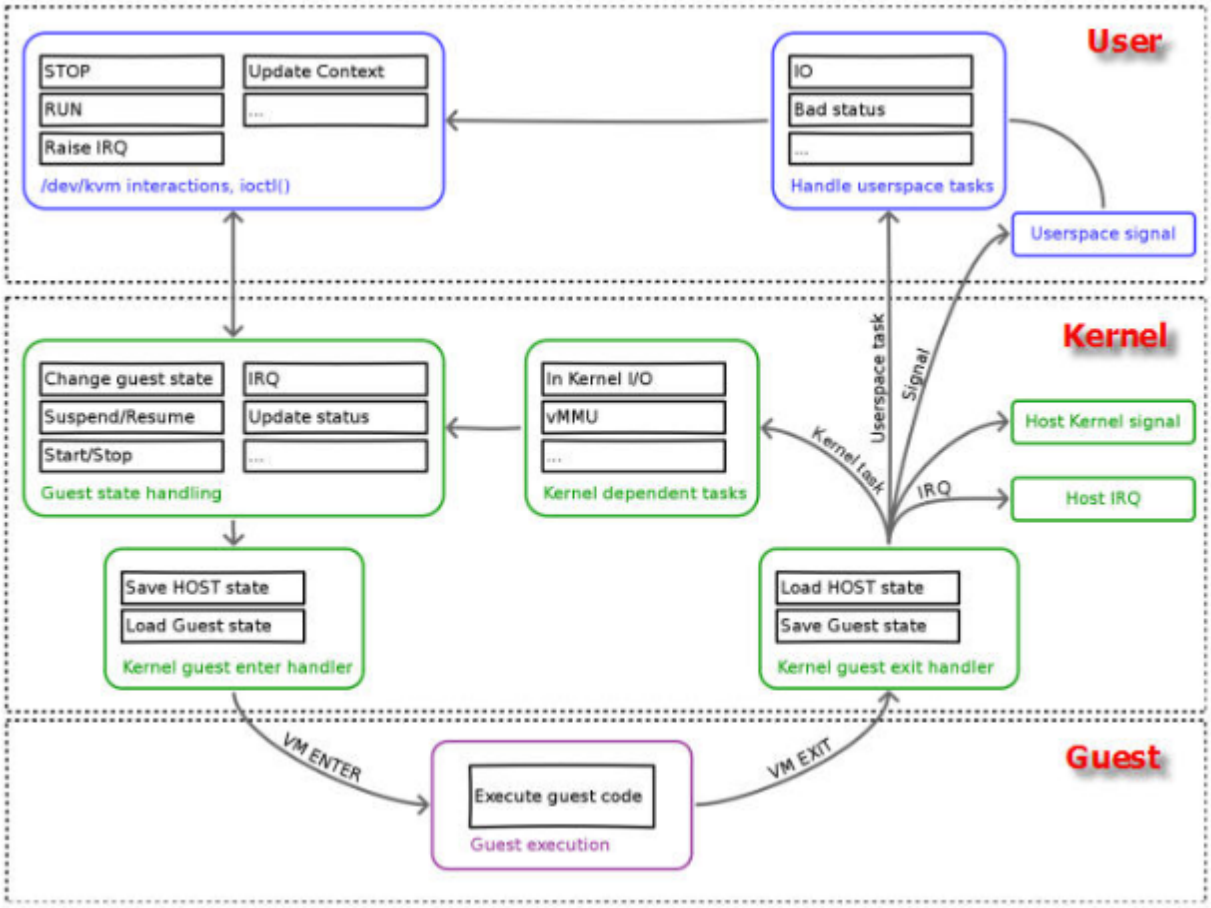


- **CPU Modes**

  The KVM kernel module serves as a bridge between user mode and Guest mode. QEMU-KVM in User mode runs virtual machines through ICOTL commands. After the KVM kernel module receives the request, it first does some preparatory work, such as loading the VCPU context into VMCS (virtual machine control structure), and then drives the CPU into VMX non-root mode to start executing client code.

  Three kinds of modes:

  - Guest mode: execute client system non-I/O code and drive the CPU out of the mode when needed.
  - Kernel mode: be responsible for switching CPU to Guest mode to execute Guest OS code and returning to Kenerl mode when CPU exits Guest mode.
  - User mode: Execute I/O operations on behalf of client systems.



References

- http://www.cs.columbia.edu/~nieh/teaching/e6998/homework/
- https://stackoverflow.com/questions/33590843/implementing-a-custom-hypercall-in-kvm#
- https://elixir.bootlin.com/linux/v4.10/source/arch/x86/kvm/x86.c#L6165
- https://blog.csdn.net/u012075739/article/details/86710067?tdsourcetag=s_pctim_aiomsg
- http://www.cnblogs.com/sammyliu/p/4543597.html