VM HW2 B10902078

Instructions on compiling your kernel module

1. go to the `linux` directory, and run `git apply b10902078_hw2_kernel.patch` to update the kernel code
2. run
   - `make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig`, and
   - `make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j4` to compile the kernel into an image
     (Note: since I run on a M3 Mac, so I omit the cross_compile argument)
3. go to the `kernel_module` folder, and run
   - `make KDIR=/PATH/TO/kernel-source ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-`
     to compile the kernel module into a `virt_walker.ko` file
     (Note: since I run on a M3 Mac, so I omit the cross_compile argument)
4. Enter the KVM host, and
   - scp `virt_walker.ko` from Ubuntu into KVM
   - mount the disk image of the guest VM in `/mnt`, and cp `virt_walker.ko` from KVM to the disk image of the guest VM
5. Enter the guest VM, and run
   - `insmod virt_walker.ko` to insert the kernel module into the Linux kernel of the guest VM (use `lsmod` to check if the kernel module was successfully inserted)

Explanations:

Kernel patch:

I handle the MMIO fault in `io_mem_abort() [arch/arm64/kvm/mmio.c]`, which gets called by `kvm_handle_guest_abort() [arch/arm64/kvm/mmu.c]`.

First, I create a callback function, which is similar to `hyp_map_walker()` in `arch/arm64/kvm/hyp/pgtable.c`. This is a callback function that will be called at the end of `kvm_pgtable_walk()`.

```
static int hide_seek_walker(u64 addr, u64 end, u32 level, kvm_pte_t *ptep, enum kv
        struct hide_seek_data *data = arg;
```

```
        // only interested in leaf entries mapping 0x40000000
        if (flag != KVM_PGTABLE_WALK_LEAF || addr != 0x40000000)
                return 0;

        if (data->is_write) {
                // clear bits [58:51] and set them to target value
                *ptep &= ~(0xFFULL << 51);
                *ptep |= ((u64)data->value << 51);
        }
        else{
                // read bits [58:51]
                data->value = (*ptep >> 51) & 0xFF;
        }

        data->found = true;
        return 0;
}
```

When it gets called, it will check whether the current stage-2 page table level is at the leaf, and whether the fault IPA/GPA is equal to 0x40000000. If it is a write operation, it will clear bits [58:51] of *ptep (which is the page table entry that maps GPA 0x40000000), and set them to the target value. If it is a read operation, then it will read bits [58:51] and store them to data->value.

Then, I create a helper function, its main objective is to setup the related data to call `kvm_pgtable_walk()`. I get the stage-2 table of the VM (which is managed by KVM) by calling `vcpu->kvm->arch.mmu.pgt`

```
static int handle_hide_seek_mmio(struct kvm_vcpu *vcpu, phys_addr_t fault_ipa, boo
        struct hide_seek_data walk_data = {
                .is_write = is_write,
                .found = false
        };

        // setup walk data
        if (is_write)
                walk_data.value = *data;


        struct kvm_pgtable_walker walker = {
                .cb = hide_seek_walker,
                .flags = KVM_PGTABLE_WALK_LEAF,
                .arg = &walk_data
        };

        int ret;

        // walk the stage-2 page tables
        ret = kvm_pgtable_walk(vcpu->kvm->arch.mmu.pgt, 0x40000000, PAGE_SIZE, &wa
        if (ret)
                return ret;
```

```
        if (!walk_data.found)
                return -EFAULT;

        if (!is_write)
                *data = walk_data.value;

        return 0;
}
```

Finally, I will call this helper function inside `io_mem_abort()`. If it is a write operation to the HIDE register, I will get the value from the register of the vcpu by calling `vcpu_get_reg()`, and pass this value to the helper function. If it is a read operation from the SEEK register, I will store the value returned by the helper function in the register of the vcpu by calling `vcpu_set_reg()`.

After this, I will call `kvm_incr_pc(vcpu)` to move the PC of the vcpu to the next instructon. I handle this whole process in KVM (not in QEMU) because read and write are simple operations that don't need to be emulated by QEMU in userspace.

```
if (fault_ipa >= 0x0b000000 && fault_ipa <= 0x0b000002) {
        is_write = kvm_vcpu_dabt_iswrite(vcpu);
        len = kvm_vcpu_dabt_get_as(vcpu);

        printk("MMIO to HIDE and SEEK register detected, len: %d\n", len);

        if (len != 1) { // only allow one byte access
                printk("more than one byte being read/write, ERROR\n");
                return -EINVAL;
        }

        if (fault_ipa == 0x0b000000 && is_write) {
                rt = kvm_vcpu_dabt_get_rd(vcpu);
                data = vcpu_get_reg(vcpu, rt);
                data_buf[0] = data & 0xFF; // only care about one byte

                ret = handle_hide_seek_mmio(vcpu, fault_ipa, true, &data_buf[0]);
        }
        else if (fault_ipa == 0x0b000001 && !is_write) {
                ret = handle_hide_seek_mmio(vcpu, fault_ipa, false, &data_buf[0]);
                if (ret == 0) {
                        data = data_buf[0];
                        rt = kvm_vcpu_dabt_get_rd(vcpu);
                        vcpu_set_reg(vcpu, rt, data);
                }
        }

        if (ret == 0) {
                kvm_incr_pc(vcpu);
                return 1; // handled in KVM (kernel), no need to handle it in user
        }
        else {
```

```
                return -EFAULT;
        }
}
```

Kernel module:

First, I will map the MMIO region using

```
// Map the MMIO region
static void __iomem *mmio_base;

static int __init virt_walker_init(void) {
    ...
    // Map MMIO region: create a mapping from the virtual device's physical addres
    // to a virtual address that the kernal can use
    mmio_base = ioremap(HIDE_REG_ADDR, 2); // map 2 bytes for both registers
    if (!mmio_base) {
            pr_err("virt_walker: ioremap() failed\n");
            ret = -ENOMEM;
            goto out_device;
    }

}
```

This allow the guest kernel to use a virtual address that maps to the virtual devices' physical addresses (0x0b000000 & 0x0b000001).

Then, in `virt_walker_read()`, the guest kernel will call `readb()` to read one byte from the MMIO address. This operation will trap to KVM. After this operation completes and the guest kernel gets the value, it will copy this value to the user process by calling `copy_to_user()`.

Similarly, in `virt_walker_write()`, the guest kernel will first get the value from user process by calling `copy_from_user()`. Then, it will call `writeb()` to write one byte to the MMIO address, which will also trap to KVM.

Note: I always perform `readb()`, `writeb()`, and `return 1` to ensure the kernel module always read/write one byte.

```
static ssize_t virt_walker_read(struct file *file, char __user *buffer, size_t cou
    /* TODO: Add your code here. */
    u8 value;
    printk("read length: %ld bytes\n", count);

    // read from SEEK register, readb(): read one byte from a MMIO address
    value = readb(mmio_base + (SEEK_REG_ADDR - HIDE_REG_ADDR));

    // copy value to user space
    if (copy_to_user(buffer, &value, 1))
```

```
        return -EFAULT;

    return 1;
}

static ssize_t virt_walker_write(struct file *file, const char __user *buffer, siz
    /* TODO: Add your code here. */
    u8 value;
    printk("write length: %ld bytes\n", count);

    // get value from user space
    if (copy_from_user(&value, buffer, 1))
        return -EFAULT;

    // write to HIDE register
    writeb(value, mmio_base);

    return 1;
}
```