实验三: I/O 虚拟化

[522031910557] [冯海桐]

1 调研部分

1. PIO: x86 架构提供了 IN/OUT、INS/OUTS 等指令访问与设备寄存器相关的 I/O 端口。Hypervisor 将这四条指令设定 为会触发 VM-Exit 的敏感指令，当客户机通过发起这四条指令进行端口 I/O 操作时会触发 VM-Exit，陷入 Hypervisor。同时 Hypervisor 会保存访问的端口号、访问数据宽度、数据传输方向等相关信息，以达到截获客户机端口 I/O 操作的目的。 MMIO: MMIO 地址空间属于物理地址空间中的高地址部分，程序可以使用内存访问指令进行 MMIO。与端口 I/O 涉及的 IN/OUT 指令不同的是，内存访问指令并不属于 MMIO 的专属指令，Hypervisor 不能将其设为敏感指令。为了截获客户机的 MMIO 操作，Hypervisor 不会将虚拟机中 MMIO 所在的物理地址范围映射到主机的物理地址空间，即影子页表中不存在相应的页表项。每次当客户机操作系统发起 MMIO 时，都会产生一个缺页异常，产生 VM-Exit，这样就可以截获客户机的 MMIO 访问并交由相关处理函数模拟。
2. 当计算机启动时，固件首先进行初始化，包括 CPU、内存和基本硬件的初始化。固件会扫描系统中的各种总线，以发现连接到这些总线上的设备。对于 PCI 总线，固件会读取每个可能的设备地址来检测设备的存在。一旦发现设备，固件会读取设备的配置信息，以确定设备的类型、制造商和设备 ID。这些信息通常存储在设备的配置寄存器中，固件通过标准的 I/O 操作读取这些寄存器。根据设备的类型和 ID，固件会加载相应的驱动程序，以便与设备进行通信。这些驱动程序可以是固件内置的，也可以从外部存储设备加载。驱动程序加载后，固件会调用驱动程序的初始化函数，对设备进行初始化。初始化过程可能包括设置设备的工作模式、分配资源（如内存和 I/O 端口）等。初始化完成后，设备会被注册到系统中，供操作系统和其他软件使用。固件会将设备的信息存储在系统表中（如 ACPI 表），以便操作系统在启动时可以访问这些信息。

2 实验目的

1. 理解 I/O 虚拟化的基本概念；
2. 理解 I/O 虚拟化的基本原理；
3. 使用测试程序访问虚拟设备并打印过程，加以分析。

3 实验步骤

1. 启动并登陆到 L1 虚拟机中。
2. 进入 L1 虚拟机的 cpu_lab 目录下：`$ cd ~/labs/io_lab`。
3. 启动并登陆到 L2 虚拟机中。
4. 通过 `$ make` 编译目标程序，通过 `sudo insmod pci.ko` 安装驱动程序并查看设备状态。
5. 通过 `lspci` 查看 PCI 设备，并用 `lspci -s 00:04.0 -vvv -xxxx` 查看详细信息。
6. 通过 `cat /proc/devices | grep edu` 查看驱动信息以及主设备号。
7. 通过 `sudo mknod /dev/edu c <major> 0` 构造设备节点，其中 `<major>` 为主设备号。通过 `ls /dev/edu` 查看设备节点文件。
8. 通过 `sudo dmesg --clear` 清空消息，然后通过 `sudo ./test` 运行测试代码。
9. 通过 `sudo dmesg` 打印测试过程，记录结果后通过 `sudo poweroff` 退出 L2 虚拟机。
10. 根据记录的翻译过程进行分析，描述虚拟 I/O 过程。

4 实验分析

4.1 代码理解

4.1.1 pci.c

宏定义设备的各种寄存器地址、命令和常量。

```
#define FACTORIA_VAL 0x8
#define IO_FACTORIA_IRQ 0x20
#define IO_IRQ_STATUS 0x24
#define IO_IRQ_RAISE 0x60
#define IO_IRQ_ACK 0x64

// 其他宏定义省略...
```

定义支持的PCI设备ID。

```
static struct pci_device_id pci_ids[] = {
    { PCI_DEVICE(QEMU_VENDOR_ID, EDU_DEVICE_ID), },
    { 0, }
};
MODULE_DEVICE_TABLE(pci, pci_ids);
```

`edu_open()`、`edu_read()`、`edu_write()` 和 `edu_ioctl()` 函数分别实现了设备文件的打开、读取、写入和 `ioctl` 操作。

其中 `edu_ioctl()` 函数处理了5种 `ioctl` 操作：`DMA_WRITE_CMD` 启动 DMA 传输，命令包括从内存读取和中断；`DMA_READ_CMD` 启动 DMA 传输，命令包括写入内存和中断；`PRINT_EDUINFO_CMD` 打印设备资源的长度，读取并打印设备的配置空间前 64 个字节，打印设备的中断号，读取并打印设备的 I/O 寄存器值；`SEND_INTERRUPT_CMD` 向设备的中断寄存器写入值 0x12345678，触发中断；`FACTORIAL_CMD` 向设备的阶乘中断寄存器写入值 0x80，等待 1 秒后向设备的阶乘值寄存器写入值 0xA，再等待 1 秒后读取并打印计算结果。

`irq_handler()` 函数处理设备的中断请求。

`pci_probe()` 和 `pci_remove()` 在设备插入和移除时被调用。

`edu_init()` 和 `edu_exit()` 函数在模块加载和卸载时被调用，注册和注销PCI驱动程序。

4.1.2 test.c

进行宏定义。这些宏定义了不同的 `ioctl` 命令，每个命令都有一个唯一的标识符。

```
#define MAGIC 'k'
#define DMA_READ_CMD    _IO(MAGIC,0x1a)
#define DMA_WRITE_CMD    _IO(MAGIC,0x1b)
#define PRINT_EDUINFO_CMD    _IO(MAGIC,0x1c)
#define SEND_INTERRUPT_CMD    _IO(MAGIC,0x1d)
#define FACTORIAL_CMD    _IO(MAGIC,0x1e)
```

以读写模式打开设备文件 `/dev/edu`。如果打开失败，打印错误信息并返回。

```
fd = open("/dev/edu",O_RDWR,S_IRUSR | S_IWUSR);
if(fd == -1) {
    printf("open edu device failure/n");
    return -1;
}
```

依次发送 `PRINT_EDUINFO_CMD`、`SEND_INTERRUPT_CMD`、`FACTORIAL_CMD` 和 `DMA_WRITE_CMD` 命令给设备。如果 `ioctl` 调用失败，打印错误信息。

```
ret = ioctl(fd, PRINT_EDUINFO_CMD);
if (ret < 0) {
    printf("ioctl: %d\n", ret);
}

ret = ioctl(fd, SEND_INTERRUPT_CMD);
if (ret < 0) {
    printf("ioctl: %d\n", ret);
}

ret = ioctl(fd, FACTORIAL_CMD);
if (ret < 0) {
    printf("ioctl: %d\n", ret);
}

ret = ioctl(fd, DMA_WRITE_CMD);
if (ret < 0) {
    printf("ioctl: %d\n", ret);
}
```

等待 1 秒，关闭设备文件，然后重新打开设备文件。

```
sleep(1);
close(fd);
fd = open("/dev/edu",O_RDWR,S_IRUSR | S_IWUSR);
```

发送 `DMA_READ_CMD` 命令给设备。如果 `ioctl` 调用失败，打印错误信息。 等待 1 秒，关闭设备文件并返回。

```
ret = ioctl(fd, DMA_READ_CMD);
if (ret < 0) {
    printf("ioctl: %d\n", ret);
}

sleep(1);
```

```
    close(fd);
    return 0;
```

4.2 实验结果

运行的结果如附录 2 所示。

先发送了 PRINT_EDUINFO_CMD 命令，打印了相关信息。

```
[  677.240300] length 100000
[  677.240364] config 0 34
[  677.240404] config 1 12
[  677.240443] config 2 e8
...
[  677.242873] dev->irq a
[  677.242926] io 0 10000ed
[  677.242948] io 4 0
[  677.242969] io 8 0
...
```

然后发送了 SEND_INTERRUPT_CMD 命令，中断寄存器内的值为 0x12345678。

```
[  677.243765] irq_handler irq = 10 dev = 245 irq_status = 12345678
```

然后发送了 FACTORIAL_CMD 命令，计算 10 的阶乘并输出，结果为 0x375f00，即 3628800。

```
[  678.272507] receive a FACTORIAL interrupter!
[  678.272508] irq_handler irq = 10 dev = 245 irq_status = 1
[  679.296482] computing result 375f00
```

然后发送了 DMA_WRITE_CMD 命令，但是却输出了读取相关的结果。

```
[  679.396643] receive a DMA read interrupter!
[  679.396645] irq_handler irq = 10 dev = 245 irq_status = 100
```

最后发送了 DMA_READ_CMD 命令，输出了读取相关的结果。

```
[  681.420767] receive a DMA read interrupter!
[  681.420770] irq_handler irq = 10 dev = 245 irq_status = 100
```

5 遇到的问题及解决方案

不知道为什么 DMA_WRITE_CMD 命令不能正确执行，询问助教。助教说，是因为 qemu 源码里的 edu.c 没有对 DMA 的读写做区分。

附录 1

实验代码

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <stdio.h>

#define MAGIC 'k'
#define DMA_READ_CMD    _IO(MAGIC,0x1a)
#define DMA_WRITE_CMD    _IO(MAGIC,0x1b)
#define PRINT_EDUINFO_CMD    _IO(MAGIC,0x1c)
#define SEND_INTERRUPT_CMD    _IO(MAGIC,0x1d)
#define FACTORIAL_CMD    _IO(MAGIC,0x1e)



int main(void)
{
    int fd, num, arg, ret;
    fd = open("/dev/edu",O_RDWR,S_IRUSR | S_IWUSR);

    if(fd == -1) {
        printf("open edu device failure/n");
        return -1;
    }
    ret = ioctl(fd, PRINT_EDUINFO_CMD);
    if (ret < 0) {
        printf("ioctl: %d\n", ret);
    }

    ret = ioctl(fd, SEND_INTERRUPT_CMD);
    if (ret < 0) {
        printf("ioctl: %d\n", ret);
    }

    ret = ioctl(fd, FACTORIAL_CMD);
    if (ret < 0) {
        printf("ioctl: %d\n", ret);
    }

    ret = ioctl(fd, DMA_WRITE_CMD);
    if (ret < 0) {
        printf("ioctl: %d\n", ret);
    }
```

```
    sleep(1);
    close(fd);
    fd = open("/dev/edu",O_RDWR,S_IRUSR | S_IWUSR);

    if(fd == -1) {
        printf("open edu device failure/n");
        return -1;
    }

    ret = ioctl(fd, DMA_READ_CMD);
    if (ret < 0) {
        printf("ioctl: %d\n", ret);
    }

    sleep(1);
    close(fd);
    return 0;

}
```

附录 2

实验结果

```
[  677.240300] length 100000
[  677.240364] config 0 34
[  677.240404] config 1 12
[  677.240443] config 2 e8
[  677.240483] config 3 11
[  677.240522] config 4 3
[  677.240560] config 5 1
[  677.240600] config 6 10
[  677.240637] config 7 0
[  677.240677] config 8 10
[  677.240716] config 9 0
[  677.240755] config a ff
[  677.240795] config b 0
[  677.240834] config c 0
[  677.240873] config d 0
[  677.240913] config e 0
[  677.240952] config f 0
[  677.240992] config 10 0
[  677.241031] config 11 0
[  677.241070] config 12 a0
[  677.241109] config 13 fe
[  677.241149] config 14 0
[  677.241187] config 15 0
[  677.241226] config 16 0
[  677.241266] config 17 0
[  677.241322] config 18 0
[  677.241360] config 19 0
```

```
[  677.241415] config 1a 0
[  677.241455] config 1b 0
[  677.241494] config 1c 0
[  677.241533] config 1d 0
[  677.241573] config 1e 0
[  677.241612] config 1f 0
[  677.241651] config 20 0
[  677.241690] config 21 0
[  677.241730] config 22 0
[  677.241769] config 23 0
[  677.241808] config 24 0
[  677.241848] config 25 0
[  677.241887] config 26 0
[  677.241925] config 27 0
[  677.241964] config 28 0
[  677.242003] config 29 0
[  677.242042] config 2a 0
[  677.242080] config 2b 0
[  677.242119] config 2c f4
[  677.242159] config 2d 1a
[  677.242198] config 2e 0
[  677.242237] config 2f 11
[  677.242280] config 30 0
[  677.242319] config 31 0
[  677.242358] config 32 0
[  677.242397] config 33 0
[  677.242445] config 34 40
[  677.242483] config 35 0
[  677.242521] config 36 0
[  677.242569] config 37 0
[  677.242607] config 38 0
[  677.242645] config 39 0
[  677.242682] config 3a 0
[  677.242720] config 3b 0
[  677.242758] config 3c b
[  677.242796] config 3d 1
[  677.242834] config 3e 0
[  677.242872] config 3f 0
[  677.242873] dev->irq a
[  677.242926] io 0 10000ed
[  677.242948] io 4 0
[  677.242969] io 8 0
[  677.242991] io c ffffffff
[  677.243012] io 10 ffffffff
[  677.243033] io 14 ffffffff
[  677.243054] io 18 ffffffff
[  677.243079] io 1c ffffffff
[  677.243103] io 20 0
[  677.243124] io 24 0
[  677.243145] io 28 ffffffff
[  677.243166] io 2c ffffffff
[  677.243189] io 30 ffffffff
[  677.243225] io 34 ffffffff
[  677.243245] io 38 ffffffff
```

```
[  677.243265] io 3c ffffffff
[  677.243289] io 40 ffffffff
[  677.243310] io 44 ffffffff
[  677.243331] io 48 ffffffff
[  677.243351] io 4c ffffffff
[  677.243371] io 50 ffffffff
[  677.243392] io 54 ffffffff
[  677.243412] io 58 ffffffff
[  677.243433] io 5c ffffffff
[  677.243453] io 60 ffffffff
[  677.243473] io 64 ffffffff
[  677.243493] io 68 ffffffff
[  677.243514] io 6c ffffffff
[  677.243534] io 70 ffffffff
[  677.243554] io 74 ffffffff
[  677.243575] io 78 ffffffff
[  677.243595] io 7c ffffffff
[  677.243615] io 80 0
[  677.243636] io 84 ffffffff
[  677.243656] io 88 0
[  677.243676] io 8c ffffffff
[  677.243696] io 90 0
[  677.243717] io 94 ffffffff
[  677.243765] irq_handler irq = 10 dev = 245 irq_status = 12345678
[  678.272507] receive a FACTORIAL interrupter!
[  678.272508] irq_handler irq = 10 dev = 245 irq_status = 1
[  679.296482] computing result 375f00
[  679.396643] receive a DMA read interrupter!
[  679.396645] irq_handler irq = 10 dev = 245 irq_status = 100
[  681.420767] receive a DMA read interrupter!
[  681.420770] irq_handler irq = 10 dev = 245 irq_status = 100
```