I/O access and Interrupts

Lab objectives

- communication with peripheral devices
- implement interrupt handlers
- synchronizing interrupts with process context

Keywords: IRQ, I/O port, I/O address, base address, UART, request_region, release_region, inb, outb

Background information

A peripheral device is controlled by writing and reading its registers. Often, a device has multiple registers that can be accessed at consecutive addresses either in the memory address space or in the I/O address space. Each device connected to the I/O bus has a set of I/O addresses, called I/O ports. I/O ports can be mapped to physical memory addresses so that the processor can communicate with the device through instructions that work directly with the memory. For simplicity, we will directly use I/O ports (without mapping to physical memory addresses) to communicate with physical devices.

The I/O ports of each device are structured into a set of specialized registers to provide a uniform programming interface. Thus, most devices will have the following types of registers:

- **Control** registers that receive device commands
- **Status** registers, which contain information about the device's internal status
- **Input** registers from which data is taken from the device
- **Output** registers in which the data is written to transmit it to the device

Physical ports are differentiated by the number of bits: they can be 8, 16 or 32-bit ports.

For example, the parallel port has 8 8-bit I/O ports starting at base address 0x378. The data log is found at base address (0x378), status register at base + 1 (0x379), and control at base address + 2 (0x37a). The data log is both an entry and exit log.

Although there are devices that can be fully controlled using I/O ports or special memory areas, there are situations where this is insufficient. The main problem that needs to be addressed is that certain events occur at undefined moments in time and it is inefficient for the processor (CPU) to interrogate the status of the device repeatedly (polling). The way to solve this problem is using an Interrupt ReQuest (IRQ) which is a hardware notification by which the processor is announced that a particular external event happened.

For IRQs to be useful device drivers must implement handlers, i.e. a particular sequence of code that handles the interrupt. Because in many situations the number of interrupts available is limited, a device driver must behave in an orderly fashion with interruptions: interrupts must be requested before being used and released when they are no longer needed. In addition, in some situations, device drivers must share an interrupt or synchronize with interrupts. All of these will be discussed further.

When we need to access shared resources between an interrupt routine (A) and code running in process context or in bottom-half context (B), we must use a special synchronization technique. In (A) we need to use a spinlock primitive, and in (B) we must disable interrupts AND use a spinlock primitive. Disabling interrupts is not enough because the interrupt routine can run on a processor other than the one running (B).

Using only a spinlock can lead to a deadlock. The classic example of deadlock in this case is:

1. We run a process on the X processor, and we acquire the lock
2. Before releasing the lock, an interrupt is generated on the X processor
3. The interrupt handling routine will try to acquire the lock and it will go into an infinite loop

Accessing the hardware

In Linux, the I/O ports access is implemented on all architectures and there are several APIs that can be used.

Request access to I/O ports

Before accessing I/O ports we first must request access to them, to make sure there is only one user. In order to do so, one must use the `request_region()` function:

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

struct resource *request_region(unsigned long first, unsigned long n,
                                const char *name);
```

To release a reserved region one must use the `release_region()` function:

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

For example, the serial port COM1 has the base address 0x3F8 and it has 8 ports and this is a code snippet of how to request access to these ports:

```
#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;
}
```

To release the ports one would use something like:

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

Most of the time, port requests are done at the driver initialization or probe time and the port releasing is done at the removal of the device or module.

All of the port requests can be seen from userspace via the `/proc/ioports` file:

```
$ 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
...
```

Accessing I/O ports

After a driver has obtained the desired I/O port range, one can perform read or write operations on these ports. Since physical ports are differentiated by the number of bits (8, 16, or 32 bits), there are different port access functions depending on their size. The following port access functions are defined in asm/io.h:

- *unsigned inb(int port)*, reads one byte (8 bits) from port
- *void outb(unsigned char byte, int port)*, writes one byte (8 bits) to port
- *unsigned inw(int port)*, reads two bytes (16-bit) ports
- *void outw(unsigned short word, int port)*, writes two bytes (16-bits) to port
- *unsigned inl (int port)*, reads four bytes (32-bits) from port
- *void outl(unsigned long word, int port)*, writes four bytes (32-bits) to port

The port argument specifies the address of the port where the reads or writes are done, and its type is platform dependent (may be unsigned long or unsigned short).

Some devices may have problems when the processor is trying to transfer data too fast to and from the device. To avoid this issue we may need to insert a delay after an I/O operation and there are functions you can use that introduce this delay. Their names are similar to those described above, with the exception that it ends in _p: inb_p, outb_p, etc.

For example, the following sequence writes a byte on COM1 serial port and then reads it:

```
#include <asm/io.h>
#define MY_BASEPORT 0x3F8

unsigned char value = 0xFF;
outb(value, MY_BASEPORT);
value = inb(MY_BASEPORT);
```

5. Accessing I/O ports from userspace

Although the functions described above are defined for device drivers, they can also be used in user space by including the <sys/io.h> header. In order to be used, ioperm or iopl must first be called to get permission to perform port operations. The ioperm function obtains permission for individual ports, while iopl for the entire I/O address space. To use these features, the user must be root.

The following sequence used in user space gets permission for the first 3 ports of the serial port, and then releases them:

```
#include <sys/io.h>
#define MY_BASEPORT 0x3F8

if (ioperm(MY_BASEPORT, 3, 1)) {
     /* handle error */
}

if (ioperm(MY_BASEPORT, 3, 0)) {
     /* handle error */
}
```

The third parameter of the ioperm function is used to request or release port permission: 1 to get permission and 0 to release.

Interrupt handling

Requesting an interrupt

As with other resources, a driver must gain access to an interrupt line before it can use it and release it at the end of the execution.

In Linux, the request to obtain and release an interrupt is done using the `requests_irq()` and `free_irq()` functions:

```
#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);
```

Note that to get an interrupt, the developer calls `request_irq()`. When calling this function you must specify the interrupt number (*irq_no*), a handler that will be called when the interrupt is generated (*handler*), flags that will instruct the kernel about the desired behaviour (*flags*), the name of the device using this interrupt (*dev_name*), and a pointer that can be configured by the user at any value, and that has no global significance (*dev_id*). Most of the time, *dev_id* will be pointer to the device driver's private data. When the interrupt is released, using the `free_irq()` function, the developer must send the same pointer value (*dev_id*) along with the same interrupt number (*irq_no*). The device name (*dev_name*) is used to display statistics in */proc/interrupts*.

The value that `request_irq()` returns is 0 if the entry was successful or a negative error code indicating the reason for the failure. A typical value is *-EBUSY* which means that the interrupt was already requested by another device driver.

The *handler* function is executed in interrupt context which means that we can't call blocking APIs such as `mutex_lock()` or `msleep()`. We must also avoid doing a lot of work in the interrupt handler and instead use deferred work if needed. The actions performed in the interrupt handler include reading the device registers to get the status of the device and acknowledge the interrupt, operations that most of the time can be performed with non-blocking calls.

There are situations where although a device uses interrupts we can't read the device's registers in a non-blocking mode (for example a sensor connected to an I2C or SPI bus whose driver does not guarantee that bus read / write operations are non-blocking). In this situation, in the interruption, we must plan a work-in-process action (work queue, kernel thread) to access the device's registers. Because such a situation is relatively common, the kernel provides the `request_threaded_irq()` function to write interrupt handling routines running in two phases: a process-phase and an interrupt context phase:

```
#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 is the function running in interrupt context, and will implement critical operations while the thread_fn function runs in process context and implements the rest of the operations.

The flags that can be transmitted when an interruption is made are:

- *IRQF_SHARED* announces the kernel that the interrupt can be shared with other devices. If this flag is not set, then if there is already a handler associated with the requested interrupt, the request for interrupt will fail. A shared interrupt is handled in a special way by the kernel: all the associated interrupt handlers will be executed until the device that generated the interrupt will be identified. But how can a device driver know if the interrupt handling routine was activated by an interrupt generated by the device it manages? Virtually all devices that offer interrupt support have a status register that can be interrogated in the handling routine to see if the interrupt was or was not generated by the device (for example, in the case of the 8250 serial port, this status register is IIR - Interrupt Information Register). When requesting a shared interrupt, the dev_id argument must be unique and it must not be NULL. Usually it is set to module's private data.
- *IRQF_ONESHOT* interrupt will be reactivated after running the process context routine; Without this flag, the interrupt will be reactivated after running the handler routine in the context of the interrupt

Requesting the interrupt can be done either at the initialization of the driver (`init_module()`), when the device is probed, or when the device is used (e.g. during *open*).

The following example performs the interrupt request for the COM1 serial port:

```
#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;
     }
     [...]
}
```

As you can see, the IRQ for serial port COM1 is 4, which is used in shared mode (IRQF_SHARED).

Attention

When requesting a shared interrupt (IRQF_SHARED) the *dev_id* argument can not be NULL.

To release the interrupt associated with the serial port, the following operations will be executed:

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

During the initialization function (`init_module()`), or in the function that opens the device, interrupts must be activated for the device. This operation is dependent on the device, but most often involves setting a bit from the control register.

As an example, for the 8250 serial port, the following operations must be performed to enable interrupts:

```
#include <asm/io.h>
#define MY_BASEPORT 0x3F8

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

In the above example, two operations are performed:

1. All interruptions are activated by setting bit 3 (Aux Output 2) in the MCR register - Modem Control Register
2. The RDAI (Transmit Holding Register Empty Interrupt) is activated by setting the appropriate bit in the IER - Interrupt Enable Register.

Implementing an interrupt handler

Lets take a look at the signature of the interrupt handler function:

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

The function receives as parameters the number of the interrupt (*irq_no*) and the pointer sent to `request_irq()` when the interrupt was requested. The interrupt handling routine must return a value with a type of `typedef irqreturn_t`. For the current kernel version, there are three valid values: *IRQ_NONE*, *IRQ_HANDLED*, and *IRQ_WAKE_THREAD*. The device driver must return *IRQ_NONE* if it notices that the interrupt has not been generated by the device it is in charge. Otherwise, the device driver must return *IRQ_HANDLED* if the interrupt can be handled directly from the interrupt context or *IRQ_WAKE_THREAD* to schedule the running of the process context processing function.

The skeleton for an interrupt handler is:

```
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;
}
```

Typically, the first thing executed in the interrupt handler is to determine whether the interrupt was generated by the device that the driver ordered. This usually reads information from the device's registers to indicate whether the device has generated an interrupt. The second thing is to reset the interrupt pending bit on the physical device as most devices will no longer generate interruptions until this bit has been reset (e.g. for the 8250 serial port bit 0 in the IIR register must be cleared).

Locking

Because the interrupt handlers run in interrupt context the actions that can be performed are limited: unable to access user space memory, can't call blocking functions. Also, synchronization using spinlocks is tricky and can lead to deadlocks if the spinlock used is already acquired by a process that has been interrupted by the running handler.

However, there are cases where device drivers have to synchronize using interrupts, such as when data is shared between the interrupt handler and process context or bottom-half handlers. In these situations it is necessary to both deactivate the interrupt and use spinlocks.

There are two ways to disable interrupts: disabling all interrupts, at the processor level, or disabling a particular interrupt at the device or interrupt controller level. Processor disabling is faster and is therefore preferred. For this purpose, there are locking functions that disable and enable interrupts acquiring and release a spinlock at the same time: `spin_lock_irqsave()`, `spin_unlock_irqrestore()`, `spin_lock_irq()`, and `spin_unlock_irq()`:

```
#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);
```

The `spin_lock_irqsave()` function disables interrupts for the local processor before it obtains the spinlock; The previous state of the interrupts is saved in *flags*.

If you are absolutely sure that the interrupts on the current processor have not already been disabled by someone else and you are sure you can activate the interrupts when you release the spinlock, you can use `spin_lock_irq()`.

For read / write spinlocks there are similar functions available:

- `read_lock_irqsave()`
- `read_unlock_irqrestore()`
- `read_lock_irq()`
- `read_unlock_irq()`
- `write_lock_irqsave()`
- `write_unlock_irqrestore()`
- `write_lock_irq()`
- `write_unlock_irq()`

If we want to disable interrupts at the interrupt controller level (not recommended because disabling a particular interrupt is slower, we can not disable shared interrupts) we can do this with `disable_irq()`, `disable_irq_nosync()`, and `enable_irq()`. Using these functions will disable the interrupts on all processors. Calls can be nested: if disable_irq is called twice, it will require as many calls enable_irq to enable it. The difference between disable_irq and disable_irq_nosync is that the first one will wait for the executed handlers to finish. Because of this, `disable_irq_nosync()` is generally faster, but may lead to races with the interrupts handler, so when not sure use `disable_irq()`.

The following sequence disables and then enables the interrupt for the COM1 serial port:

```
#define MY_IRQ 4

disable_irq (MY_IRQ);
enable_irq (MY_IRQ);
```

It is also possible to disable interrupts at the device level. This approach is also slower than disabling interrupts at the processor level, but it works with shared interrupts. The way to accomplish this is device specific and it usually means we have to clear a bit from one of the control registers.

It is also possible to disable all interrupts for the current processor independent of taking locks. Disabling all interruptions by device drivers for synchronization purposes is inappropriate because races are still possible if the interrupt is handled on another CPU. For reference, the functions that disable / enable interrupts on the local processor are `local_irq_disable()` and `local_irq_enable()`.

In order to use a resource shared between process context and the interrupt handling routine, the functions described above will be used as follows:

```
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);
    ...
}
```

The *my_access function* above runs in process context. To synchronize access to the shared data, we disable the interrupts and use the spinlock *lock*, i.e. the `spin_lock_irqsave()` and `spin_unlock_irqrestore()` functions.

In the interrupt handling routine, we use the `spin_lock()` and `spin_unlock()` functions to access the shared resource.

Note

The *flags* argument for `spin_lock_irqsave()` and `spin_unlock_irqrestore()` is a value and not a pointer but keep in mind that `spin_lock_irqsave()` function changes the value of the flag, since this is actually a macro.

Interrupt statistics

Information and statistics about system interrupts can be found in */proc/interrupts* or */proc/stat*. Only system interrupts with associated interrupt handlers appear in */proc/interrupts*:

```
# cat /proc/interrupts
                CPU0
0:           7514294       IO-APIC-edge   timer
1:              4528       IO-APIC-edge   i8042
6:                 2       IO-APIC-edge   floppy
8:                 1       IO-APIC-edge   rtc
9:                 0       IO-APIC-level  acpi
12:             2301       IO-APIC-edge   i8042
15:               41       IO-APIC-edge   ide1
16:             3230       IO-APIC-level  ioc0
17:             1016       IO-APIC-level  vmxnet ether
NMI:               0
LOC:         7229438
ERR:               0
MIS:               0
```

The first column specifies the IRQ associated with the interrupt. The following column shows the number of interrupts that were generated for each processor in the system; The last two columns provide information about the interrupt controller and the device name that registered the handler for that interrupt.

The */proc/state* file provides information about system activity, including the number of interruptions generated since the last (re)boot of the system:

```
# cat /proc/stat | grep in
intr 7765626 7754228 4620 0 0 0 0 2 0 1 0 0 0 2377 0 0 41 3259 1098 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Each line in the */proc/state* file begins with a keyword that specifies the meaning of the information on the line. For information on interrupts, this keyword is intr. The first number on the line represents the total number of interrupts, and the other numbers represent the number of interrupts for each IRQ, starting at 0. The counter includes the number of interrupts for all processors in the system.

Further reading

Serial Port

- Serial Port
- Interfacing the Serial / RS232 Port

Parallel port

- Interfacing the Standard Parallel Port
- Parallel Port Central

Keyboard controller

- Intel 8042
- drivers/input/serio/i8042.c
- drivers/input/keyboard/atkbd.c

Linux device drivers

- Linux Device Drivers, 3rd ed., Ch. 9 - Communicating with Hardware
- Linux Device Drivers, 3rd ed., Ch. 10 - Interrupt Handling
- Interrupt Handlers

Exercises

Important

We strongly encourage you to use the setup from this repository.

To solve exercises, you need to perform these steps:

- prepare skeletons from templates
- build modules
- start the VM and test the module in the VM.

The current lab name is interrupts. See the exercises for the task name.

The skeleton code is generated from full source examples located in `tools/labs/templates`. To solve the tasks, start by generating the skeleton code for a complete lab:

```
tools/labs $ make clean
tools/labs $ LABS=<lab name> make skels
```

You can also generate the skeleton for a single task, using

```
tools/labs $ LABS=<lab name>/<task name> make skels
```

Once the skeleton drivers are generated, build the source:

```
tools/labs $ make build
```

Then, start the VM:

```
tools/labs $ make console
```

The modules are placed in /home/root/skels/interrupts/<task_name>.

You DO NOT need to STOP the VM when rebuilding modules! The local *skels* directory is shared with the VM.

Review the Exercises section for more detailed information.

Warning

Before starting the exercises or generating the skeletons, please run **git pull** inside the Linux repo, to make sure you have the latest version of the exercises.

If you have local changes, the pull command will fail. Check for local changes using `git status`. If you want to keep them, run `git stash` before `pull` and `git stash pop` after. To discard the changes, run `git reset --hard master`.

If you already generated the skeleton before `git pull` you will need to generate it again.

0. Intro

Using LXR, find the definitions of the following symbols in the Linux kernel:

- `struct resource`
- `request_region()` and `__request_region()`
- `request_irq()` and `request_threaded_irq()`
- `inb()` for the x86 architecture.

Analyze the following Linux code:

- Keyboard initialization function `i8042_setup_kbd()`
- The AT or PS/2 keyboard interrupt function `atkbd_interrupt()`

Keyboard driver

The next exercise's objective is to create a driver that uses the keyboard IRQ, inspect the incoming key codes and stores them in a buffer. The buffer will be accessible from userspace via character device driver.

1. Request the I/O ports

To start with, we aim to allocate memory in the I/O space for hardware devices. We will see that we cannot allocate space for the keyboard because the designated region is already allocated. Then we will allocate I/O space for unused ports.

The *kbd.c* file contains a skeleton for the keyboard driver. Browse the source code and inspect `kbd_init()`. Notice that the I/O ports we need are I8042_STATUS_REG and I8042_DATA_REG.

Follow the sections maked with **TODO 1** in the skeleton. Request the I/O ports in `kbd_init()` and make sure to check for errors and to properly clean-up in case of errors. When requesting, set the reserving caller's ID string (`name`) with `MODULE_NAME` macro. Also, add code to release the I/O ports in `kbd_exit()`.

Note

You can review the Request access to I/O ports section before proceeding.

Now build the module and copy it to the VM image:

```
tools/labs $ make build
tools/labs $ make copy
```

Now start the VM and insert the module:

```
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
```

Notice that you get an error when trying to request the I/O ports. This is because we already have a driver that has requested the I/O ports. To validate check the `/proc/ioports` file for the `STATUS_REG` and `DATA_REG` values:

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

Lets find out which driver register these ports and try to remove the module associated with it.

```
$ find -name \*.c | xargs grep \"keyboard\"

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
```

It looks like the I/O ports are registered by the kernel during the boot, and we won't be able to remove the associated module. Instead, let's trick the kernel and register ports 0x61 and 0x65.

Use the function `request_region()` (inside the `kbd_init()` function) to allocate the ports and the function `release_region()` (inside the `kbd_exit()` function) to release the allocated memory.

This time we can load the module and */proc/ioports* shows that the owner of these ports is our module:

```
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
```

Let's remove the module and check that the I/O ports are released:

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

2. Interrupt handling routine

For this task we will implement and register an interrupt handler for the keyboard interrupt. You can review the Requesting an interrupt section before proceeding.

Follow the sections marked with **TODO 2** in the skeleton.

First, define an empty interrupt handling routine named `kbd_interrupt_handler()`.

Note

Since we already have a driver that uses this interrupt we should report the interrupt as not handled (i.e. return `IRQ_NONE`) so that the original driver still has a chance to process it.

Then register the interrupt handler routine using `request_irq`. The interrupt number is defined by the *I8042_KBD_IRQ* macro. The interrupt handling routine must be requested with `IRQF_SHARED` to share the interrupt line with the keyboard driver (i8042).

Note

For shared interrupts, *dev_id* can not be NULL . Use `&devs[0]`, that is pointer to `struct kbd`. This structure contains all the information needed for device management. To see the interrupt in */proc/interrupts*, do not use NULL for *dev_name* . You can use the MODULE_NAME macro.

If the interrupt requesting fails make sure to properly cleanup by jumping to the right label, in this case the one the releases the I/O ports and continues with unregistering the character device driver.

Compile, copy and load module in the kernel. Check that the interrupt line has been registered by looking at */proc/interrupts* . Determine the IRQ number from the source code (see *I8042_KBD_IRQ*) and verify that there are two drivers registered at this interrupt line (which means that we have a shared interrupt line): the i8042 initial driver and our driver.

Note

More details about the format of the */proc/interrupts* can be found in the Interrupt statistics section.

Print a message inside the routine to make sure it is called. Compile and reload the module into the kernel. Check that the interrupt handling routine is called when you press the keyboard on the virtual machine, using **dmesg**. Also note that when you use the serial port no keyboard interrupt is generated.

Attention

To get access to the keyboard on the virtual machine boot with "QEMU_DISPLAY=gtk make boot".

3. Store ASCII keys to buffer

Next, we want to collect the keystrokes in a buffer whose content we will then send to the user space. For this routine we will add the following in the interrupt handling:

- capture the pressed keys (only pressed, ignore released)
- identify the ASCII characters.
- copy the ASCII characters corresponding to the keystrokes and store them in the buffer of the device

Follow the sections marked **TODO 3** in the skeleton.

Reading the data register

First, fill in the `i8042_read_data()` function to read the `I8042_DATA_REG` of the keyboard controller. The function just needs to return the value of the register. The value of the registry is also called scancode, which is what is generated at each keystroke.

Hint

Read the `I8042_DATA_REG` register using `inb()` and store the value in the local variable `val`. Revisit the Accessing I/O ports section.

Call the `i8042_read_data()` in the `kbd_interrupt_handler()` and print the value read.

Print information about the keystrokes in the following format:

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

Where scancode is the value of the read register using the `i8042_read_data()` function.

Notice that the scancode (reading of the read register) is not an ASCII character of the pressed key. We'll have to understand the scancode.

Interpreting the scancode

Note that the registry value is a scancode, not the ASCII value of the character pressed. Also note that an interrupt is sent both when the key is pressed and when the key is released. We only need to select the code when the key is pressed and then and decode the ASCII character.

Note

To check scancode, we can use the showkey command (showkey -s).

In this form, the command will display the key scancodes for 10 seconds after the last pressed key end then it will stop. If you press and release a key you will get two scancodes: one for the pressed key and one for the released key. E.g:

- If you press the ENTER key, you will get the 0x1c (0x1c) and 0x9c (for the released key)
- If you press the key a you will get the 0x1e (key pressed) and 0x9e (for the key release)
- If you press b you will get 0x30 (key pressed) and 0xb0 (for the release key)
- If you press the c key, you will get the 0x2e (key pressed) 0xae and 0xae (for the released key)
- If you press the Shift key you will get the 0x2a (key pressed) 0xaa and 0xaa (for the released key)
- If you press the Ctrl key you will get the 0x1d (key pressed) and 0x9d (for the release key)

  As also indicated in this article, a key release scancode is 128 (0x80) higher then a key press scancode. This is how we can distinguish between a press key scancode and a release scancode.

  A scancode is translated into a keycode that matches a key. A pressed scanned keycode and a released scancode have the same keycode. For the keys shown above we have the following table:

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)

  The press / release key is performed in the is_key_press() function and obtaining the ASCII character of a scancode takes place in the get_ascii() function.

In the interrupt handler check the scancode to see if the key is pressed or released then determine the corresponding ASCII character.

Hint

To check for press / release, use `is_key_press()`. Use `get_ascii()` function to get the corresponding ASCII code. Both functions expect the scancode.

Hint

To display the received information use the following format.

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

Where scancode is the value of the data register, and ch is the value returned by the get_ascii() function.

Store characters to the buffer

We want to collect the pressed characters (not the other keys) into a circular buffer that can be consumed from user space.

Update the interrupt handler to add a pressed ASCII character to the end of the device buffer. If the buffer is full, the character will be discarded.

Hint

The device buffer is the field `buf` in the device's `struct kbd`. To get the device data from the interrupt handler use the following construct:

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

The buffer's dimension is located in `struct kbd`'s field, `count`. The `put_idx` and `get_idx` fields specify the next writing and reading index. Take a look at the `put_char()` function's implementation to observe how the data is added to the circular buffer.

Attention

Synchronize the access to the buffer and the helper indexes with a spinlock. Define the spinlock in the device struct `struct kbd` and initialize it in `kbd_init()`.

Use the `spin_lock()` and `spin_unlock()` functions to protect the buffer in the interrupt handler.

Revisit the Locking section.

4. Reading the buffer

In order to have access to the keylogger's data, we have to send it to the user space. We will do this using the */dev/kbd* character device. When reading from this device, we will get the data from the buffer in the kernel space, where we collected the keys pressed.

For this step follow the sections marked with **TODO 4** in the `kbd_read()` function.

Implement `get_char()` in a similar way to `put_char()`. Be careful when implementing the circular buffer.

In the `kbd_read()` function copy the data from the buffer to the userspace buffer.

Hint

Use `get_char()` to read a character from the buffer and `put_user()` to store it to the user buffer.

Attention

In the read function, use `spin_lock_irqsave()` and `spin_unlock_irqrestore()` for locking.

Revisit the Locking section.

Attention

We cannot use `put_user()` or `copy_to_user()` while holding the lock, as userpace access is not permitted from atomic contexts.

For more info, read the Access to the address space of the process section in the previous lab.

For testing, you will need to create the */dev/kbd* character device driver using the mknod before reading from it. The device master and minor are defined as `KBD_MAJOR` and `KBD_MINOR`:

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

Build, copy and boot the virtual machine and load the module. Test it using the command:

```
cat /dev/kbd
```

5. Reset the buffer

Reset the buffer if the device is written to. For this step follow the sections marked with **TODO 5** in the skeleton.

Implement `reset_buffer()` and add the write operation to *kbd_fops*.

Attention

In the write function Use `spin_lock_irqsave()` and `spin_unlock_irqrestore()` for locking when resetting the buffer.

Revisit the Locking section.

For testing, you will need to create the */dev/kbd* character device driver using the mknod before reading from it. The device master and minor are defined as `KBD_MAJOR` and `KBD_MINOR`:

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

Build, copy and boot the virtual machine and load the module. Test it using the command:

```
cat /dev/kbd
```

Press some keys, then run the command **echo "clear"** > **/dev/kbd**. Check the buffer's content again. It should be reset.

Extra Exercises

1. kfifo

Implement a keylogger using the kfifo API.

Hint

Follow the API call examples from the kernel code. For example, the file bytestream-examples.c.