Computer Labs: I/O and Interrupts

2º L.EIC

Pedro F. Souto (`pfs@fe.up.pt`)

February 27, 2025

I/O Operation

- I/O devices are the interface between the computer and its environment
- Most of the time, the processor is not synchronized with its environment
  - I/O operations are **asynchronous** wrt the processor operation
- Usually, I/O devices are much slower than the processor
  - The processor **must wait** for an I/O device to complete its current operation before it can request a new one

How Does the Processor Know about an I/O event?

Polling The processor polls the I/O device, i.e. reads a status register, to find out

Interrupts The I/O device notifies the processor, via the interrupt mechanism

How Does the Processor Know about an I/O event?

Polling The processor polls the I/O device, i.e. reads a status register, to find out

- **Response time** Highly variable – depends on what the processor has to do between consecutive polls.

Interrupts The I/O device notifies the processor, via the interrupt mechanism

- **Response time** Usually responsive – depends on the time:
  - interrupts are disabled or
  - higher priority interrupts take to be served

How Does the Processor Know about an I/O event?

Polling The processor polls the I/O device, i.e. reads a status register, to find out

Response time Highly variable – depends on what the processor has to do between consecutive polls.

Efficiency/Overhead Depends on the frequency of the event

- The more frequent the more efficient
  - Assuming, polling at a constant rate

Interrupts The I/O device notifies the processor, via the interrupt mechanism

Response time Usually responsive – depends on the time:

- interrupts are disabled or
- higher priority interrupts take to be served

Efficiency/Overhead Depends on the frequency of the event

- The more frequent the less efficient
  - Overhead per interrupt is higher than that per poll

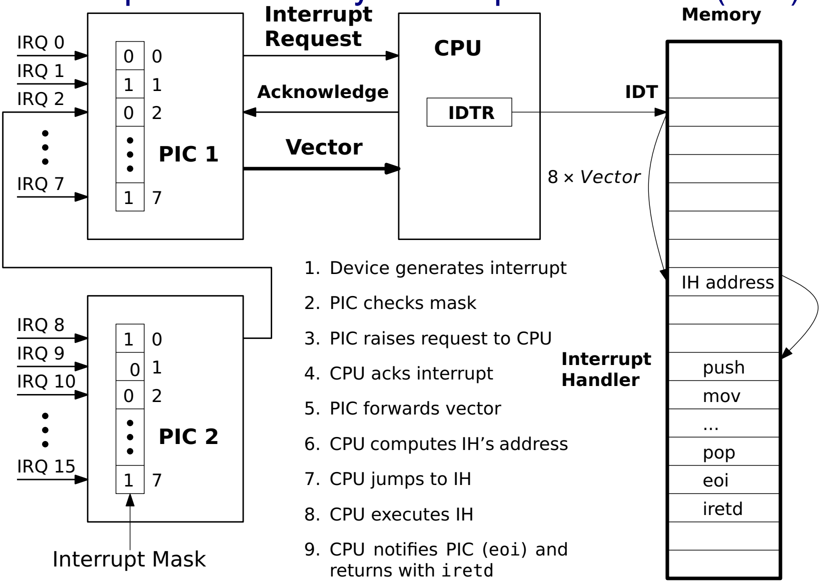
Lab 2: `timer_test_int()`

What to do? Print one message per second, for a time interval whose duration is specified in its argument, by using:

- Timer 0 interrupts
- LCF function:

```
void timer_print_elapsed_time()
```

PC Interrupt HW: Priority Interrupt Controller (PIC)



1. Device generates interrupt
2. PIC checks mask
3. PIC raises request to CPU
4. CPU acks interrupt
5. PIC forwards vector
6. CPU computes IH's address
7. CPU jumps to IH
8. CPU executes IH
9. CPU notifies PIC (`eoi`) and returns with `iretd`

Imp: If a bit of the Interrupt Mask is set, the corresponding IRQ is disabled.

PC Interrupts: IRQ Lines and Vectors

PIC 1	PIC 2	Device	Vector
IRQ0		Timer	0x08
IRQ1		Keyboard	0x09
IRQ2		PIC2	0x0A
	IRQ0	Real Time Clock	0x70
	IRQ1	Replace IRQ2	0x71
	IRQ2- IRQ7	Reserved	0x72-0x77
IRQ3		Serial port COM2	0x0B
IRQ4		Serial port COM1	0x0C
IRQ5		Reserved/Sound card	0x0D
IRQ6		Floppy disk	0x0E
IRQ7		Parallel port	0x0F

IRQ line Determined by the HW designer (IBM)

Vector Specified also by IBM, but can be configured at boot time. All that is needed is:

1. Configure the PIC
2. Configure the IDT (Interrupt Descriptor Table)

Interrupt Handlers (IH)

- IHs are executed by the HW upon an interrupt
  - They run **asynchronously** wrt other code
  - They take no arguments
  - They return no values
- IHs used to be written in assembly
  - Need to perform I/O operations

```
isr_name:
    push ..              ; save all registers used
    ...                  ; IH instructions
    mov al, EOI          ; signal EOI
    out PIC1_CMD, al     ;  to PIC1
    pop ...              ; restore all registers used
    iretd
```

- But nowadays, they are usually written in C (for reasons of portability)

Terminology Interrupt handlers are also called interrupt service routines (ISR) and are part of the respective **device driver**

Interrupt Handling in Minix 3

- In Minix, device drivers are implemented as **user-level processes**, rather than at the kernel-level
  - This was an important design decision in Minix 3

Interrupt Handling in Minix 3

- In Minix, device drivers are implemented as **user-level processes**, rather than at the kernel-level
  - This was an important design decision in Minix 3

Issue How do you do interrupt handling?

- Interrupt handling requires performing operations that usually require special privileges

Interrupt Handling in Minix 3

- In Minix, device drivers are implemented as **user-level processes**, rather than at the kernel-level
  - This was an important design decision in Minix 3

Issue How do you do interrupt handling?

- Interrupt handling requires performing operations that usually require special privileges

Solution

1. Perform only the bare minimum in the kernel: this is done by the **generic interrupt handler** (GIH)
2. Device specific operations are performed by the device drivers themselves at user level
   - Using kernel calls to perform privileged operations

Minix 3: The Generic Interrupt Handler (GIH)

- Upon an interrupt, the GIH:
  1. Masks, in the PIC, the respective IRQ line.
  2. Notifies all the device drivers (DD) **interested** in that interrupt
  3. If possible, unmasks, in the PIC, the respective IRQ line.
  4. Acknowledges the interrupt by issuing the `EOI` command to the PIC.
  5. Issues the `IRETD` instruction

Issue 1 How does the GIH know that a DD is interested in an interrupt?

Issue 2 How does the GIH notify a DD?

Issue 3 How does a DD receive the notification of the GIH?

Issue 4 How does the GIH know if it can unmask the IRQ line in the PIC?

Issue 5 If the GIH does not unmask the IRQ line in the PIC, when, how and whom does it?

Issue 1

How does the GIH know that a DD is interested in an interrupt?

Issue 1

How does the GIH know that a DD is interested in an interrupt?

Answer The DD tells it, using kernel call:

```
int sys_irqsetpolicy(int irq_line, int policy, int *hook_id)
```

where

`irq_line` is the IRQ line of the device

`policy` use `IRQ_REENABLE` to inform the GIH that it can unmask the IRQ line in the PIC.

- This answers Issue 4: How does the GIH know if it can unmask the IRQ line in the PIC?

`hook_id` is both:

input an id to be used by the kernel on interrupt notification

output an id to be used by the DD in other kernel calls on this interrupt

- `sys_irqsetpolicy()` can be viewed as an interrupt notification subscription

Minix 3: Other Interrupt Related Kernel Calls

`sys_irqrmpolicy(int *hook_id)` Cancels a previous interrupt notification subscription, by specifying a pointer to the `hook_id` returned by the kernel in `sys_irqsetpolicy()`

`sys_irqenable(int *hook_id)` Unmasks at the PIC an interrupt line associated with a previously subscribed interrupt notification, by specifying a pointer to the `hook_id` returned by the kernel in `sys_irqsetpolicy()`

`sys_irqdisable(int *hook_id)` Masks at the PIC an interrupt line associated with a previously subscribed interrupt notification, by specifying a pointer to the `hook_id` returned by the kernel in `sys_irqsetpolicy()`

Issue 2

How does the GIH notify the DD of the occurrence of an interrupt?

Issue 2

How does the GIH notify the DD of the occurrence of an interrupt?

Answer It uses the standard interprocess communication (IPC) mechanism used for communication:

- between processes;
- between the (micro) kernel and a process

More specifically, it uses **notifications**

Minix 3 IPC This is essentially a message based mechanism

- Processes send and receive messages to communicate with one another, and with the kernel.
- A **notification** is a special kind of message, used by the kernel to unsolicited communication with a user-level process.

Issue 3 (1/2)

How does the DD receive the notification of the GIH?

Issue 3 (1/2)

How does the DD receive the notification of the GIH?

Short Answer Just use the IPC mechanism.

Useful Answer Use some library calls provided by the `libdrivers` library

```
#include <lcom/lcf.h>
int ipc_status;
message msg;
while( 1 ) { /* You may want to use a different condition */
    /* Get a request message. */
    if( (r = driver_receive(ANY, &msg, &ipc_status)) != 0 ) {
        printf("driver_receive failed with: %d", r);
        continue;
    }
    if (is_ipc_notify(ipc_status)) { /* received notification
        switch (_ENDPOINT_P(msg.m_source)) {
        case HARDWARE: /* hardware interrupt notification */
            if (msg.m_notify.interrupts & irq_set) { /* subscri
                ... /* process it */
            }
            break;
        default:
            break; /* no other notifications expected: do nothi
        }
    } else { /* received a standard message, not a notification
        /* no standard messages expected: do nothing */
    }
}
```

Why: `msg.m_notify.interrupts`?

- Interrupt handlers take no arguments (and return no values)

Answer True, but usually an IH knows which interrupt request it is handling

- Minix 3 allows a DD to subscribe notifications on several interrupt lines

What is its value?

Answer It is based on the input value of `hook_id` passed by the DD in the corresponding `sys_irqsetpolicy()`.

- If a given interrupt is pending then the corresponding `hook_id` bit of `msg.m_notify.interrupts` is set.
- Why not just the `hook_id`?

What should `irq_set` **value be?**

- `irq_set` is used as a mask to test which interrupts are pending

Issue 3 (2/2)

Key Observation In Minix 3, a DD is an event driven service that receives and processes messages

- either interrupt notifications from the kernel (GIH)
- or service requests from other processes

However, the programs in LCOM are not DD: they do not receive requests from other processes

Lab 2: `timer_test_int()`

What to do? Print one message per second, for a time interval whose duration is specified in its argument.

1. Subscribe Timer 0 interrupts
2. Print message at 1 second intervals, by calling the LCF function:

   `void timer_print_elapsed_time()`
3. Unsubscribe Timer 0 at the end

How to design it? It is not easy to come up with an API that can be used in the project

- Implement `int timer_subscribe_int()` to hide from other code i8254 related details, such as the IRQ line used
  - It returns, via its argumens, the bit number, that will be set in `msg.m_notify.interrupts` upon a TIMER 0 interrupt
- Implement the interrupt handler also in `timer.c`
- Implement the "interrupt loop" in `timer_test_int()`

Issue 5 (and Last)

What if the GIH does not unmask the IRQ line in the PIC?

Issue 5 (and Last)

What if the GIH does not unmask the IRQ line in the PIC?

- I.e., if a DD does not set the `IRQ_REENABLE` policy in its interrupt subscription request (`sys_irqsetpolicy()`)

Answer The DD will have to do it, as soon as possible

- In most cases, you'll want to set the `IRQ_REENABLE` policy
  - In Lab 2, certainly

How can a DD unmask the IRQ line in the PIC??

- By calling `sys_irqenable(int *hook_id)`
  - Note that here `hook_id` should point to a variable with the value returned by the kernel in `sys_irqsetpolicy()`

That is, the kernel will unmask the IRQ line, upon request of the DD.

Minix 3: Interrupt Sharing

- Minix 3 already includes its own Timer 0 IH
- By subscribing interrupts on IRQ line 0, the IH of your driver will not replace the IH of the kernel
  - Upon an interrupt generated by Timer 0, the kernel:
    1. executes its own IH, and
    2. notifies your driver
- This behavior stems from the need to share the interrupt lines among devices
  - In systems with the PIC (i8259), there are only 15 interrupt lines available
  - And many of them are actually hardwired, e.g. IRQ 0, which means that they cannot be shared among devices

IMP Using two IH for the same device is seldom what you want

- But is just what we need for Lab 2.

Further Reading

- Lab 2 Handout, Section 4, The PC's Interrupt Hardware
- 8259A- Interrupt Priority Controller- Data Sheet, by Intel
- Using Interrupts
- Lab 2 Handout, Subsection 5.2 (Minix 3) Interrupt Handling