Computer Labs: I/O and Interrupts

2º MIEIC

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

September 29, 2011

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 requests the I/O device a new one

How Does the Processor Know that an I/O op is done?

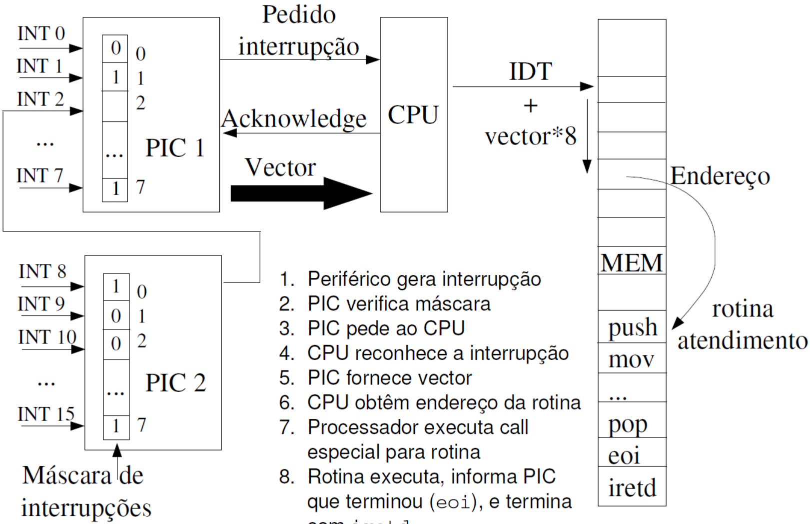
Polling The processor polls the I/O device, i.e. reads a status register, to find out

- **Response time** Highly variable – depends on what the processor has to do between consecutive polls.
- **Bandwidth** May be high, if:
  - the interface bus is fast
  - the I/O device has a high-bandwidth or a large buffer, e.g. a disk
  - and the processor polls the I/O device frequently

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
- **Bandwidth** Medium to low. It depends on the amount of data ready to transfer on each interrupt

The PC Interrupt Hardware



1. Periférico gera interrupção
2. PIC verifica máscara
3. PIC pede ao CPU
4. CPU reconhece a interrupção
5. PIC fornece vector
6. CPU obtêm endereço da rotina
7. Processador executa call especial para rotina
8. Rotina executa, informa PIC que terminou (`eoi`), e termina com `iretd`

PC Interrupts and Lines

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	Reserved	0x72
	IRQ3	Reserved	0x73
	IRQ4	Mouse	0x74
	IRQ5	Math coprocessor	0x75
	IRQ6	Hard disk	0x76
	IRQ7	Reserved	0x77
IRQ3		Serial port COM2	0x0B
IRQ4		Serial port COM1	0x0C
IRQ5		Reserved/Sound card	0x0D
IRQ6		Floppy disk	0x0E
IRQ7		Parallel port	0x0F

Interrupt Handlers (IH) in Assembly

- IHs are executed by the HW upon an interrupt
  - They run **asynchronously** with other code execution
  - They take no arguments
  - They return no values
- IHs are usually 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
```

Terminology Interrupt handlers are also called interrupt service routines (ISR).

Interrupt Handling in Minix 3

- An important design decision in Minix 3 was to move the device drivers to user space
  - Device drivers are implemented as user-level processes.

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**
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)

1. Notifies all the device drivers (DD) **interested** in an interrupt, when that interrupt occurs
2. If possible, acknowledges the interrupt by issuing the `EOI` command to the PIC.
3. 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 receives the notification of the GIH?

Issue 4 How does the GIH know if it can send the `EOI` to the PIC?

Issue 5 If the GIH does not send the `EOI`, who, when and how is the `EOI` sent to the PIC?

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 interrupt

`policy` use `IRQ_REENABLE` to inform the GHI that it can give the `EOI` command

- This answers Issue 4: How does the GIH know if it can send the `EOI` to the PIC?

`hook_id` is both:

- **input** an id to be used by the kernel on the 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

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 to communicate:

- 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
- A **notification** is a special kind of message, used to communicate from the kernel to a user process.

Issue 3

How does the DD receives the notification of the GIH?

Short Answer Just use the IPC mechanism.

Useful Answer Use some library calls provided by the `libdrivers` library (see next slide)

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

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

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

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

Why: `msg.NOTIFY_ARG`?

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

Answer True, but usually the IH know which interrupt request they are 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 `sys_irqsetpolicy()`.

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

Lab3: `timer_subscribe_int()`

- This function should subscribe Timer 0 notifications from the kernel.
- The handler main loop will be in `lab3()`
  - It is not very generic
- Thus, `timer_subscribe_int()` is supposed to return a bit-mask with the bit corresponding to the `hook_id` value passed to the kernel in `sys_irqsetpolicy()` set to one.

Issue 5 (and Last)

What if the GIH does not send the EOI?

- 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 3, certainly

How can a DD send the EOI to the PIC?

- By calling sys_irqenable(int *hook_id)
  - Note that here hook_id should point to the value returned by the kernel.

That is, the EOI will be sent by the kernel, upon request of the DD.

Minix 3: Other Interrupt Related Kernel Calls

`sys_irqrmpolicy(int *hook_id)` Unsubscribes a previous interrupt notification, by specifying a pointer to the `hook_id` returned by the kernel

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

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 3.

Further Reading

- Using Interrupts