

Sistemi Operativi

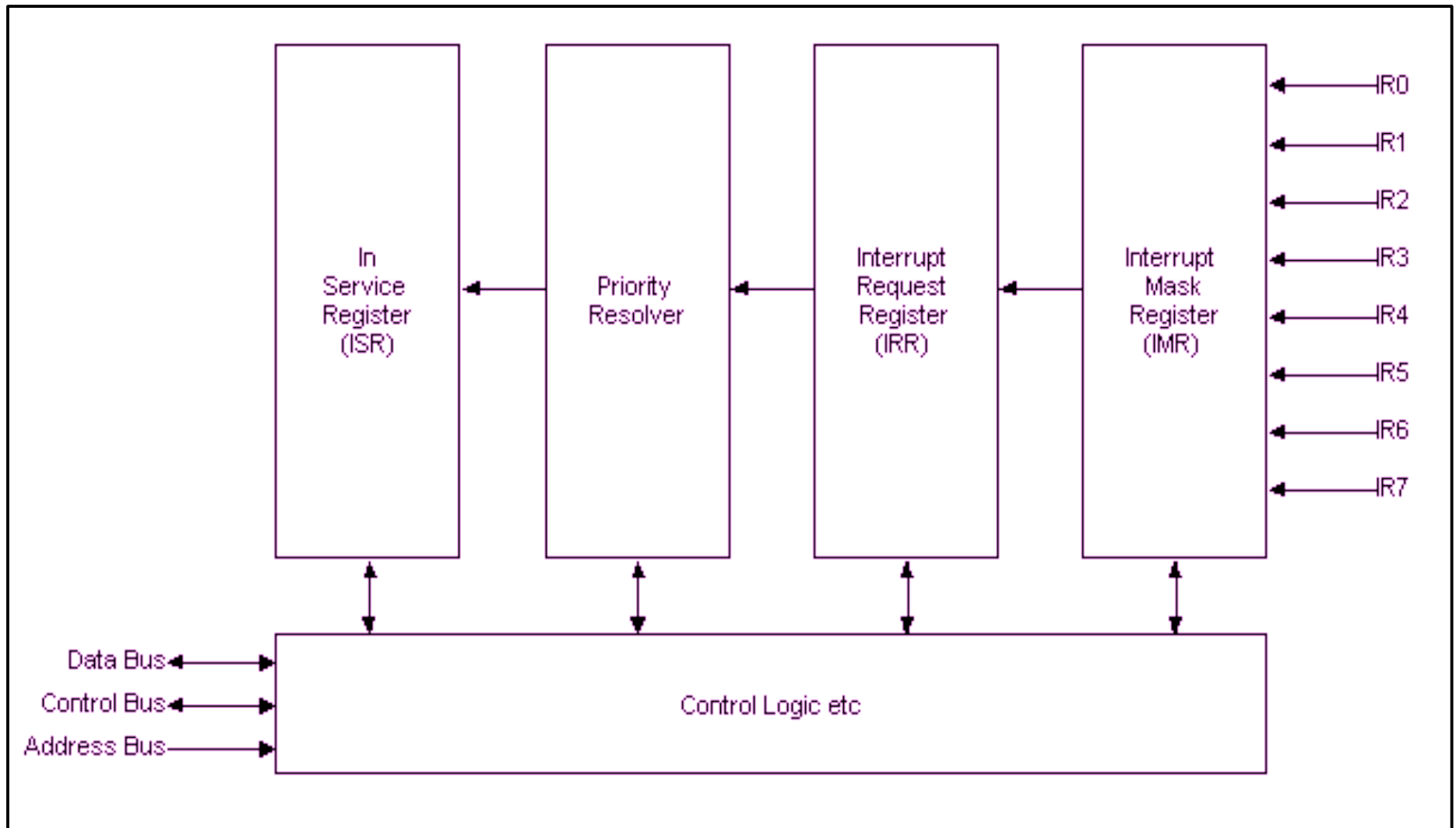
Lez. 22

La gestione degli interrupt

Introduzione

- Meccanismo per la comunicazione di eventi asincroni tra processori e periferiche
- Costituito essenzialmente da un segnale elettrico che viene intercettato dall'interrupt controller e passato al processore
- Interrompe il ciclo d'esecuzione del processore
fetch → decode → Execute

Schema interrupt controller

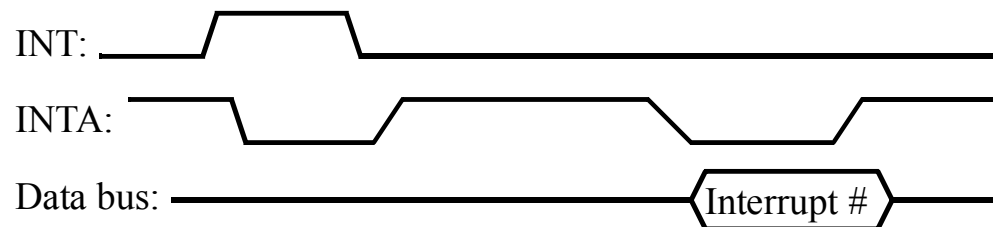


Funzionamento PIC

- Le linee di interrupt passano per prima cosa attraverso il registro IMR (Interrupt Mask Register) che verifica se sono state mascherate
- Nel caso di interrupt non mascherati, gli stessi sono registrati nel registro IRR (Interrupt Request Register), che li “conserva” finchè sono stati opportunamente gestiti
- Priority Resolver seleziona l' IRQ di più alta priorità
- Viene inviato al processore il segnale INT

Funzionamento PIC

- Il processore, appena sentito l'interrupt, risponde con il segnale INTA (Interrupt Acknowledge)
- Alla ricezione del segnale INTA, il segnale IRQ che il PIC sta gestendo viene memorizzato nel registro ISR (In Service Register ISR). Il bit corrispondente all'IRQ viene anche resettato all'interno di IRR e vengono disabilitati tutti gli interrupt di priorità minore o uguale a quello in corso
- Il processore invia un altro segnale INTA, a seguito del quale il PIC carica sul data bus un numero di 8 bit che corrisponde al vettore dell'interrupt



Registri di comando del PIC

- Programmable Interrupt Controller (PIC) gestisce gli interrupt hardware. Esistono in genere due PIC in un PC, a due diversi indirizzi
- Il primo PIC, posizionato all'indirizzo 0x20h controlla gli IRQ da 0 a 7
- Il secondo PIC posizionato all'indirizzo 0xA0 controlla gli IRQ da 8 a 15

Comandi del PIC

- Ai suddetti registri possono essere inviati comandi (usando l'istruzione out) che regolano il comportamento del PIC
- Ad esempio è possibile abilitare o disabilitare gli interrupt direttamente sul PIC invece che sul processore. La seguente sequenza abilita IRQ4:

```
in    al, 0x21    !Read existing bits.
and   al, 0xef    !Turn on IRQ 4 (COM1).
out   0x21, al    !Write result back to PIC
```

- E se voglio disabilitare IRQ 2?

EOI

- Un'altra istruzione estremamente importante nella gestione degli interrupt è l'istruzione EOI (End of Interrupt)
- Deve essere eseguita dal gestore dell'interrupt al termine della sua esecuzione e viene inviata dalla Cpu al PIC
- EOI consente la riabilitazione degli interrupt a livello PIC e consiste nell'invio del valore 0x20 al registro di comando interessato

```
mov     al, 0x20
out     0x20, al
```

```

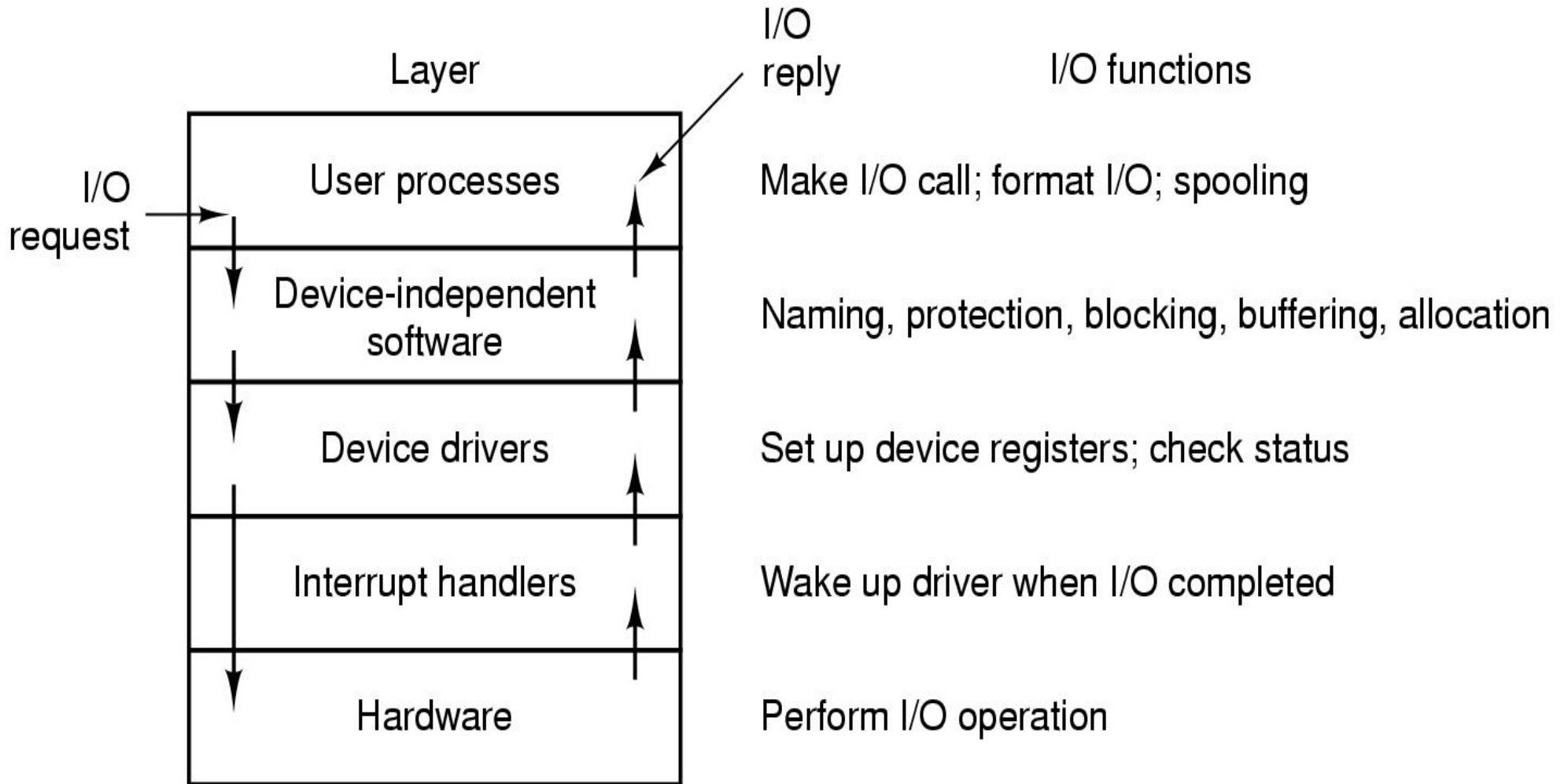
00029 /* Suitable irq bases for hardware interrupts. Reprogram the 8259(s) from
00030  * the PC BIOS defaults since the BIOS doesn't respect all the processor's
00031  * reserved vectors (0 to 31).
00032  */
00033 #define BIOS_IRQ0_VEC    0x08    /* base of IRQ0-7 vectors used by BIOS */
00034 #define BIOS_IRQ8_VEC    0x70    /* base of IRQ8-15 vectors used by BIOS */
00035 #define IRQ0_VECTOR      0x50    /* nice vectors to relocate IRQ0-7 to */
00036 #define IRQ8_VECTOR      0x70    /* no need to move IRQ8-15 */
00037
00038 /* Hardware interrupt numbers. */
00039 #define NR_IRQ_VECTORS    16
00040 #define CLOCK_IRQ        0
00041 #define KEYBOARD_IRQ     1
00042 #define CASCADE_IRQ      2    /* cascade enable for 2nd AT controller */
00043 #define ETHER_IRQ        3    /* default ethernet interrupt vector */
00044 #define SECONDARY_IRQ    3    /* RS232 interrupt vector for port 2 */
00045 #define RS232_IRQ        4    /* RS232 interrupt vector for port 1 */
00046 #define XT_WINI_IRQ      5    /* xt winchester */
00047 #define FLOPPY_IRQ       6    /* floppy disk */
00048 #define PRINTER_IRQ      7
00049 #define KBD_AUX_IRQ      12    /* AUX (PS/2 mouse) port in kbd controller */
00050 #define AT_WINI_0_IRQ    14    /* at winchester controller 0 */
00051 #define AT_WINI_1_IRQ    15    /* at winchester controller 1 */
00052
00053 /* Interrupt number to hardware vector. */
00054 #define BIOS_VECTOR(irq) \
00055     (((irq) < 8 ? BIOS_IRQ0_VEC : BIOS_IRQ8_VEC) + ((irq) & 0x07))
00056 #define VECTOR(irq) \
00057     (((irq) < 8 ? IRQ0_VECTOR : IRQ8_VECTOR) + ((irq) & 0x07))

```

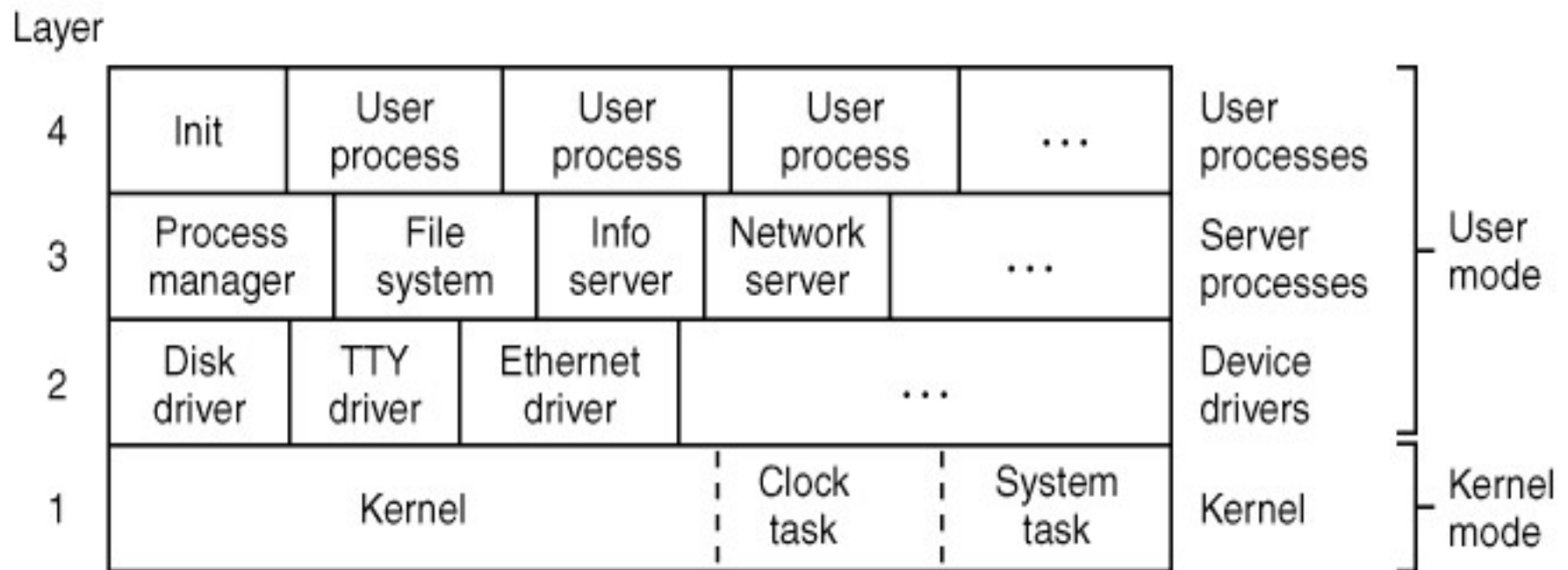
Risposta a Interrupt

- Quando la CPU riceve, a seguito del suo secondo segnale INTA il vettore dell'interrupt:
 - Disabilita gli interrupt (saranno riabilitati con IRET)
 - Recupera dal TSS del task che ha generato l'interrupt, il selettore di segmento e lo stack pointer relativi al nuovo stack
 - Su questo nuovo stack memorizza
 - lo stack segment selector
 - lo stack pointer del programma interrotto
 - EFLAGS, CS, e EIP correnti
 - Accede alla tabella IDT nella posizione indicata dal vettore stesso per richiamare il gestore dell'interrupt

Principio di funzionamento



C'è però un problema !!



Problema

- La struttura microkernel di Minix non consente la presenza a livello kernel dei driver di I/O, che per contro devono poter accedere a strutture di kernel
- Il problema viene risolto con due accorgimenti:
 - Inserendo delle opportune syscall usate dai driver per chiedere al kernel di operare su opportune strutture dati
 - Estrapolando dai driver le funzionalità di “interrupt handling” e inserendo le stesse in una procedura di Kernel nota come **generic_handler**, che poi richiamerà gli specifici driver

Interrupt handling

- Per tutti i device, ad eccezione del clock, viene richiamato il generic_handler denominato `intr_handle` che provvede a risvegliare i device driver di competenza il cui valore è contenuto nel campo `proc_nr_e` di un'opportuna struttura `hook`
- Il caricamento di questa struttura viene effettuato dalla procedura `put_irq_handler` richiamata, su richiesta dei singoli driver, dal system task

Strutture dati Minix

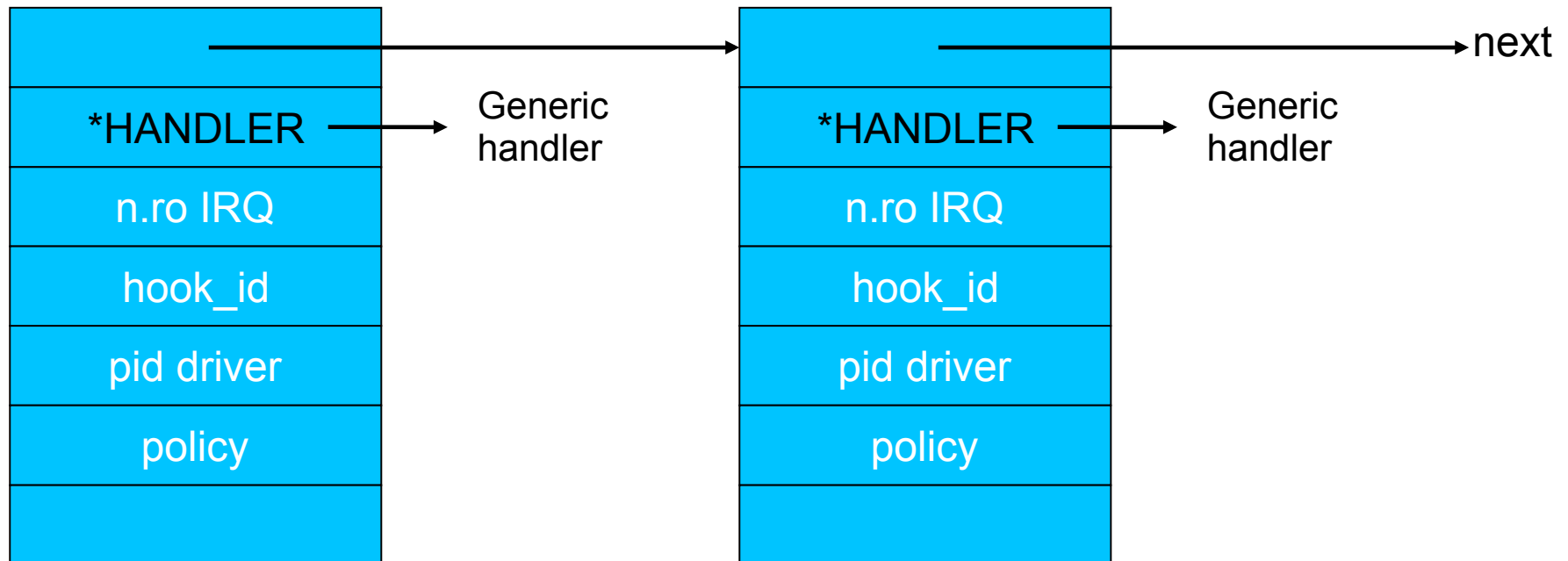
```
typedef unsigned long irq_policy_t;
typedef unsigned long irq_id_t;

typedef struct irq_hook {
    struct irq_hook *next;    /* next hook in chain */
    int (*handler)(struct irq_hook *); /* interrupt handler */
    int irq;                  /* IRQ vector number */
    int id;                   /* id of this hook */
    int proc_nr_e;           /* (endpoint) NONE if not in use */
    irq_id_t notify_id;      /* id to return on interrupt */
    irq_policy_t policy;     /* bit mask for policy */
} irq_hook_t;

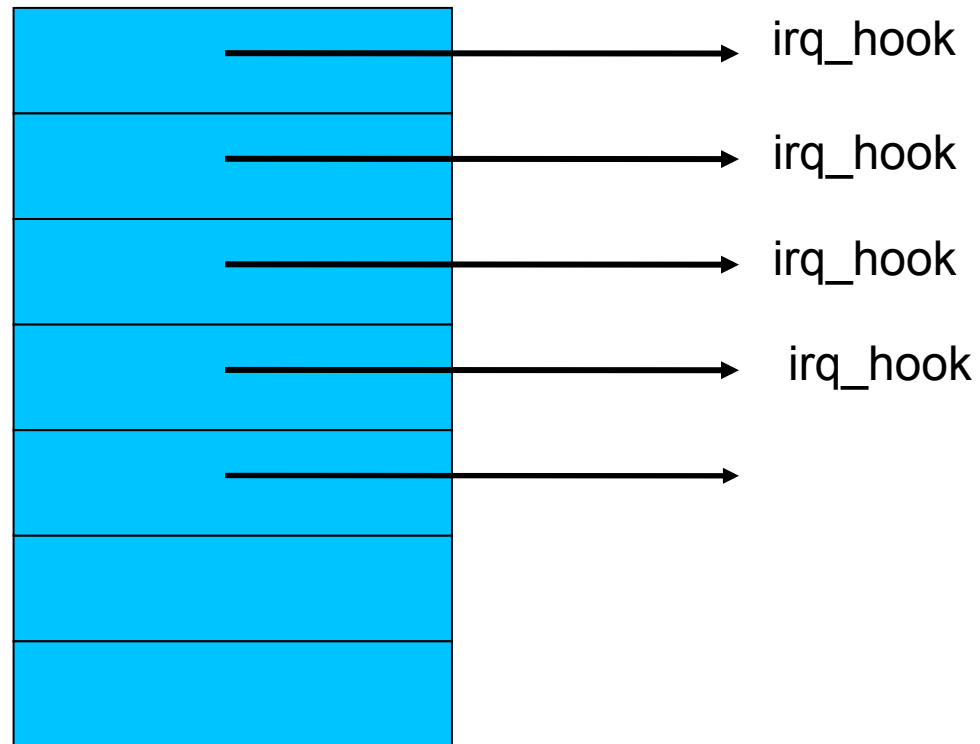
typedef int (*irq_handler_t)(struct irq_hook *);
```

Strutture dati: irq_hook

STRUTTURA DATI DI RIFERIMENTO PER LA GESTIONE DEGLI INTERRUPT, ESISTE ALMENO UN ELEMENTO irq_hook_t PER OGNI TIPO DI INTERRUPT



irq_handlers [nr_irq_hook]



Strutture dati Minix

```
00039 /* Interrupt related variables. */  
  
EXTERN irq_hook_t irq_hooks[NR_IRQ_HOOKS]; /* hooks for general use */  
  
EXTERN irq_hook_t *irq_handlers[NR_IRQ_VECTORS]; /* list of IRQ handlers */  
  
EXTERN int irq_actids[NR_IRQ_VECTORS]; /* IRQ ID bits active */  
  
EXTERN int irq_use; /* map of all in-use irq's */
```

Caricamento hook (do_irqctl)

```
/* Install the handler. */
00103 hook_ptr->proc_nr_e = m_ptr->m_source; /* process to notify */
00104 hook_ptr->notify_id = notify_id;      /* identifier to pass */
00105 hook_ptr->policy = m_ptr->IRQ_POLICY; /*policy interrupts */
00106 put_irq_handler(hook_ptr, irq_vec, generic_handler);
00107
00108 /* Return index of the IRQ hook in use. */
00109 m_ptr->IRQ_HOOK_ID = irq_hook_id + 1;
```

```

00089 /*=====
00090 *                               put_irq_handler                               *
00091 *=====*/
00092 PUBLIC void put_irq_handler(hook, irq, handler)
00093 irq_hook_t *hook;
00094 int irq;
00095 irq_handler_t handler;
00096 {
00097 /* Register an interrupt handler. */
00098     int id;
00099     irq_hook_t **line;
00100
00101     if (irq < 0 || irq >= NR_IRQ_VECTORS)
00102         panic("invalid call to put_irq_handler", irq);
00103
00104     line = &irq_handlers[irq];
00105     id = 1;
00106     while (*line != NULL) {
00107         if (hook == *line) return;           /* extra initialization */
00108         line = &(*line)->next;
00109         id <<= 1;
00110     }
00111     if (id == 0) panic("Too many handlers for irq", irq);
00112
00113     hook->next = NULL;
00114     hook->handler = handler;
00115     hook->irq = irq;
00116     hook->id = id;
00117     *line = hook;
00118
00119     irq_use |= 1 << irq;
00120 }

```

Contenuto IDT

```
00115     { hwint00, VECTOR( 0), INTR_PRIVILEGE },
00116     { hwint01, VECTOR( 1), INTR_PRIVILEGE },
00117     { hwint02, VECTOR( 2), INTR_PRIVILEGE },
00118     { hwint03, VECTOR( 3), INTR_PRIVILEGE },
00119     { hwint04, VECTOR( 4), INTR_PRIVILEGE },
00120     { hwint05, VECTOR( 5), INTR_PRIVILEGE },
00121     { hwint06, VECTOR( 6), INTR_PRIVILEGE },
00122     { hwint07, VECTOR( 7), INTR_PRIVILEGE },
00123     { hwint08, VECTOR( 8), INTR_PRIVILEGE },
00124     { hwint09, VECTOR( 9), INTR_PRIVILEGE },
00125     { hwint10, VECTOR(10), INTR_PRIVILEGE },
00126     { hwint11, VECTOR(11), INTR_PRIVILEGE },
00127     { hwint12, VECTOR(12), INTR_PRIVILEGE },
00128     { hwint13, VECTOR(13), INTR_PRIVILEGE },
00129     { hwint14, VECTOR(14), INTR_PRIVILEGE },
00130     { hwint15, VECTOR(15), INTR_PRIVILEGE },
00130 #if _WORD_SIZE == 2
00131     { p_s_call, SYS_VECTOR, USER_PRIVILEGE }, /* 286 system call */
00132 #else
00133     { s_call, SYS386_VECTOR, USER_PRIVILEGE }, /* 386 system call */
00134 #endif
00135     { level0_call, LEVEL0_VECTOR, TASK_PRIVILEGE },
00136 };
```

Qualche costante

```
/* 8259A interrupt controller ports. */
81 #define INT_CTL 0x20 /* I/O port for interrupt controller */
82 #define INT_CTLMASK 0x21 /* setting bits in this port disables
                             ints */
83 #define INT2_CTL 0xA0 /* I/O port for second interrupt
                             controller */
84 #define INT2_CTLMASK 0xA1 /* setting bits in this port
                             disables ints */
86 /* Magic numbers for interrupt controller. */
87 #define ENABLE 0x20 /* code used to re-enable after an
                             interrupt */
```

```

212 !*=====*
```

```

213 !*                hwint00 - 07                *
```

```

214 !*=====*
```

```

215 ! Note this is a macro, it just looks like a subroutine.
```

```

216 #define hwint_master(irq) \
217     call    save                /* save interrupted process state */;\
218     push    (_irq_handlers+4*irq) /* irq_handlers[irq]                */;\
219     call    _intr_handle        /* intr_handle(irq_handlers[irq]) */;\
220     pop     ecx                  ;\
221     cmp     (_irq_actids+4*irq), 0 /* interrupt still active?      */;\
222     jz      0f                  ;\
223     inb     INT_CTLMASK        /* get current mask */          ;\
224     orb     al, [1<<irq]       /* mask irq */                  ;\
225     outb    INT_CTLMASK        /* disable the irq che saranno
                                riabilitati dai driver                */;\
226 0:     movb  al, END_OF_INT     ;\
227     outb    INT_CTL            /* reenable master 8259         */;\
228     ret     /* restart (another) process */
229
```

```

230 ! Each of these entry points is an expansion of the hwint_master macro
231     .align 16
232 _hwint00:    ! Interrupt routine for irq 0 (the clock).
233     hwint_master(0)
234
235     .align 16
236 _hwint01:    ! Interrupt routine for irq 1 (keyboard)
237     hwint_master(1)

```

```

00148 /*=====
00149 *                intr_handle                *
00150 *=====*/
00151 PUBLIC void intr_handle(hook)
00152 irq_hook_t *hook;
00153 {
00154 /* Call the interrupt handlers for an interrupt with the given hook list.
00155 * The assembly part of the handler has already masked the IRQ, reenabled the
00156 * controller(s) and enabled interrupts.
00157 */
00158
00159 /* Call list of handlers for an IRQ. */
00160 while (hook != NULL) {
00161     /* For each handler in the list, mark it active by setting its ID bit,
00162     * call the function, and unmark it if the function returns true.
00163     */
00164     irq_actids[hook->irq] |= hook->id;
00165     if ((*hook->handler)(hook)) irq_actids[hook->irq] &= ~hook->id;
00166     hook = hook->next;
00167 }
00168
00169 /* The assembly code will now disable interrupts, unmask the IRQ if and only
00170 * if all active ID bits are cleared, and restart a process.
00171 */
00172 }

```

```

!*=====
316 !*                save                *
317 !*=====
318 ! Save for protected mode.
319 ! This is much simpler than for 8086 mode, because the stack already points
320 ! into the process table, or has already been switched to the kernel stack.
321
322         .align 16
323 save:
324         cld                ! set direction flag to a known value
325         pushad             ! save "general" registers
326         o16 push    ds      ! save ds
327         o16 push    es      ! save es
328         o16 push    fs      ! save fs
329         o16 push    gs      ! save gs
330         mov     dx, ss      ! ss is kernel data segment
331         mov     ds, dx      ! load rest of kernel segments
332         mov     es, dx      ! kernel does not use fs, gs
333         mov     eax, esp     ! prepare to return
334         incb   (_k_reenter) ! from -1 if not reentering
335         jnz    set_restart1 ! stack is already kernel stack
336         mov     esp, k_stktop
337         push   _restart      ! build return address for int handler
338         xor    ebp, ebp     ! for stacktrace
339         jmp    RETADR-P_STACKBASE(eax)
340
341         .align 4
342 set_restart1:
343         push   restart1
344         jmp    RETADR-P_STACKBASE(eax)

```

Generic Handler

- Sveglia il driver legato all'interrupt in oggetto, il cui valore è specificato attraverso il parametro hook
- Restituisce:
 - Un valore TRUE se al termine della sua esecuzione l'interrupt può essere riabilitato
 - FALSE: se l'interrupt deve rimanere disabilitato, in questo caso sarà il driver, precedentemente risvegliato, che provvederà a riabilitare l'interrupt

```

PRIVATE int generic_handler(hook)
00133 irq_hook_t *hook;
00134 {
00135 /* This function handles hardware interrupt in a simple and generic way. All
00136 * interrupts are transformed into messages to a driver. The IRQ line will be
00137 * reenabled if the policy says so.
00138 */
00139     int proc;
00140
00141     /* As a side-effect, the interrupt handler gathers random information by
00142     * timestamping the interrupt events. This is used for /dev/random.
00143     */
00144     get_randomness(hook->irq);
00145
00146     /* Check if the handler is still alive. If not, forget about the
00147     * interrupt. This should never happen, as processes that die
00148     * automatically get their interrupt hooks unhooked.
00149     */
00150     if(!isokendpt(hook->proc_nr_e, &proc)) {
00151         hook->proc_nr_e = NONE;
00152         return 0;
00153     }
00154
00155     /* Add a bit for this interrupt to the process' pending interrupts. When
00156     * sending the notification message, this bit map will be magically set
00157     * as an argument.
00158     */
00159     priv(proc_addr(proc))->s_int_pending |= (1 << hook->notify_id);
00160
00161     /* Build notification message and return. */
00162     lock_notify(HARDWARE, hook->proc_nr_e);
00163     return(hook->policy & IRQ_REENABLE);
00164 }
00165 A.A. 2010/2011
00166 #endif /* USE_IRQCTL */

```

```

00122 /*=====*
00123  *                rm_irq_handler                *
00124
*=====*/
00125 PUBLIC void rm_irq_handler(hook)
00126 irq_hook_t *hook;
00127 {
00128 /* Unregister an interrupt handler. */
00129     int irq = hook->irq;
00130     int id = hook->id;
00131     irq_hook_t **line;
00132
00133     if (irq < 0 || irq >= NR_IRQ_VECTORS)
00134         panic("invalid call to rm_irq_handler", irq);
00135
00136     line = &irq_handlers[irq];
00137     while (*line != NULL) {
00138         if ((*line)->id == id) {
00139             (*line) = (*line)->next;
00140             if (! irq_handlers[irq]) irq_use &= ~(1 << irq);
00141             return;
00142         }
00143         line = &(*line)->next;
00144     }
00145     /* When the handler is not found, normally return here. */
00146 }

```