

# 1 Lezione del 11-03-25

Riprendiamo la trattazione dell'controllore di interruzioni APIC.

## 1.0.1 Interruzione di livello o di fronte

Vediamo un dettaglio sul comportamento dell'APIC: questo può rilevare, in base alla sua configurazione, i **livelli** o i **fronti** delle variabili in ingresso.

Questo può avere delle implicazioni diverse a seconda dell'interfaccia. Ad esempio, avevamo detto che il timer in modalità 2 genera un onda quadra. Se si usa una routine lanciata dal timer a interruzione di programma, e si configura l'APIC per rilevare il livello, potrebbe essere che a routine concluse il livello del timer è sempre alto, e quindi l'interruzione viene lanciata nuovamente.

Questo è chiaramente diverso dal comportamento desiderato, ed è quindi opportuno configurare l'APIC per rilevare i soli fronti di salita.

Abbiamo quindi notato praticamente tutte le caratteristiche che ci interessavano dell'APIC, e possiamo procedere ad implementare un esempio di gestione di un'interfaccia a controllo di interruzione. Vediamo ad esempio il seguente programma, che gestisce la tastiera a controllo di interruzione, di cui la parte C++:

```
1 #include <libce.h>
2 #define KBD_VECT 0x20
3
4 bool fine = false;
5
6 extern "C" void a_keyboard();
7 extern "C" void c_keyboard() {
8     // leggiamo da tastiera
9     natb code = kbd::get_code();
10
11     if(code == 0x01) fine = true;
12
13     char c = kbd::conv(code);
14     vid::char_write(c);
15
16     apic::send_EOI();
17 }
18
19 void main() {
20     // attiva le interruzioni tastiera
21     kbd::enable_intr();
22
23     // imposta l'APIC
24     apic::set_VECT(1, KBD_VECT);
25     apic::set_TRGM(1, false); // false: fronte, true: livello
26     apic::set_MIRQ(1, false);
27
28     // imposta il gate nella IDT
29     gate_init(KBD_VECT, a_keyboard);
30
31     while(!fine);
32
33     return;
34 }
```

e la parte assembly:

```

1 #include <libce.h>
2
3 .global a_keyboard
4 .extern c_keyboard
5
6 a_keyboard:
7     salva_registri
8     call c_keyboard
9     carica_registri
10    iretq

```

Il meccanismo di chiamata dell'interruzione (macro per il salvataggio/caricamento registri, istruzione `iretq`, ecc...) è identico all'esempio precedente. Una novità è la presenza della funzione `send_EOI()` nel gestore di interruzione, che invia il segnale di End Of Interrupt all'APIC e gli fa capire, assieme alla lettura che facciamo sulla tastiera (con `kbd::get_code()`) che l'interruzione è stata effettivamente gestita. Inoltre, la parte di configurazione dell'interruzione è più complessa. Bisogna infatti:

- Attivare le interruzioni da tastiera con `kbd::enable_intr()`;
- Impostare l'APIC per inviare tali interruzioni al tipo interruzione `0x20`, configurandolo per riconoscere fronti, e disattivando la maschera (rispettivamente `set_TRGM()` e `set_MIRQ()`);
- Infine, inizializzare il gate corrispondente al tipo interruzione `0x20` come avevamo già visto.

Abbiamo quindi realizzato pienamente quanto ci eravamo posti di fare quando abbiamo iniziato a parlare di interruzione: la CPU è lasciata libera (nell'esempio specifico, esegue un loop infinito), e viene *interrotta* dalla periferica tastiera quando questa ha un nuovo dato disponibile.

Forniamo un altro esempio correggendo il codice dell'esempio presentato nella sezione 3.2.1. Qui l'intenzione era di scorrere a intervalli regolari un vettore, e avevamo usato un approccio piuttosto inelegante per farlo:

```

1 // i tick svolti
2 natl tick = 0;
3 natl next_song_update = 0;
4
5 natl last_value = read_timer(timer0_addr);
6
7 while (true) {
8     natl current_value = read_timer(timer0_addr);
9
10    // se il valore corrente e' maggiore del valore precedente
11    // si e' fatto un salto
12    if (current_value > last_value) {
13        tick++;
14    }
15
16    // aggiorna se necessario
17    if (tick > next_song_update) {
18        bool res = update_song();
19        if (!res) break;
20
21        // imposta il prossimo aggiornamento
22        next_song_update += TIMESTEP;

```

```

23 }
24
25 last_value = current_value;
26 }

```

Adesso che siamo in possesso delle conoscenze necessarie a usare il timer 0 come originariamente inteso, cioè sfruttando la linea di interruzione, facciamo. Introduciamo quindi una funzione `c_handler()`, e modifichiamo il `main()` come segue:

```

1 // temporizzazione
2 natl tick = 0;
3 natl next_song_update = 0;
4
5 bool fine = false;
6
7 // interruzioni
8 #define TIM_VECT 0x20
9
10 extern "C" void a_handler();
11 extern "C" void c_handler() {
12     tick++;
13
14     // aggiorna se necessario
15     if(tick > next_song_update) {
16         bool res = update_song();
17         if(!res) fine = true;
18
19         // imposta il prossimo aggiornamento
20         next_song_update += TIMESTEP;
21     }
22
23     apic::send_EOI();
24 }
25
26 void main() {
27     // setup dei timer
28
29     // imposta l'APIC
30     apic::set_VECT(2, TIM_VECT);
31     apic::set_TRGM(2, false);
32     apic::set_MIRQ(2, false);
33
34     // imposta IDT
35     gate_init(TIM_VECT, a_handler);
36
37     while(!fine);
38 }

```

L'implementazione di `a_handler()` sarà data in un file assembly separato, e sarà ancora una volta la banale:

```

1 #include <libce.h>
2
3 .extern c_handler
4 .global a_handler
5 a_handler:
6     salva_registri
7     call c_handler
8     carica_registri
9     iretq

```

Eseguendo questo programma si ha effettivamente lo stesso funzionamento dell'esempio in 3.2.1, con la differenza fondamentale che la CPU non si trova in uno stato continuo di attesa dell'avanzamento del timer, ma può fare altro mentre questo avanza, come ad esempio controllare la tastiera per arrestare su comando l'esecuzione.

In questo caso si reindirizza la linea di IREQ della tastiera ad un'interruzione di tipo diverso da quello del timer e gli si dedica un handler nell'IDT:

```

1 // ricordiamo che si e' usato #define TIM_VECT 0x20
2 #define KBD_VECT 0x21
3
4 extern "C" void a_keyb();
5 extern "C" void c_keyb() {
6     natb code = kbd::get_code();
7
8     if(code == 0x01) fine = true;
9
10    apic::send_EOI();
11 }
12
13 void main() {
14     // imposta il timer 0
15     // outputb(0x36, cwr_addr); // modo 3
16     natl div = 0; // 0 significa 65536
17     set_divisor(div, timer0_addr);
18
19     // imposta il timer 2
20     outputb(0xB6, cwr_addr); // modo 3
21
22     // imposta l'APIC
23     // timer
24     apic::set_VECT(2, TIM_VECT);
25     apic::set_TRGM(2, false);
26     apic::set_MIRQ(2, false);
27
28     // tastiera
29     apic::set_VECT(1, KBD_VECT);
30     apic::set_TRGM(1, false);
31     apic::set_MIRQ(1, false);
32
33     // imposta IDT
34     gate_init(TIM_VECT, a_timer);
35
36     kbd::enable_intr();
37     gate_init(KBD_VECT, a_keyb);
38
39     while(!fine);
40 }

```

Il codice assembly di `a_keyb()` non viene riportato in quanto è esattamente analogo a quello appena visto per `a_timer()`.

Il risultato di questa configurazione è che il processore controlla contemporaneamente, in qualche modo, timer e tastiera, permettendo all'utente di arrestare la riproduzione anche a metà del brano.

## 1.1 Eccezioni

Vediamo quindi che esiste un'altra casistica di applicazione delle interruzioni che non abbiamo trattato, cioè quella delle *eccezioni*. Queste sono particolari errori logici che il

processore potrebbe incontrare nel corso dell'esecuzione, come ad esempio la divisione per 0, il tentativo di eseguire un'istruzione non riconosciuta, ecc...

Una differenza fra le interruzioni esterne e le eccezioni è che le eccezioni possono essere sollevate *durante* la lettura e esecuzione di un'istruzione, quindi ad esempio mentre si stava interpretando un codice operativo (si pensi all'interruzione di operazione non riconosciuta). In verità, per assicurare l'atomicità dei cicli di esecuzione, la CPU ripristina automaticamente il suo stato a prima del lancio dell'interruzione. In particolare, possiamo distinguere 3 tipi di eccezione:

- **Fault:** l'esecuzione non viene ancora eseguita, lo stato IP prima della sua esecuzione viene salvato (quindi si rimane alla stessa istruzione), e si può riprovare ad eseguirla dopo aver risolto l'errore;
- **Trap:** l'esecuzione ormai è stata eseguita, e si salva l'IP successivo.
- **Abort:** raggruppa degli eventi particolarmente disastrosi in cui l'esecuzione si arresta completamente (ad esempio la tripla eccezione).

Quando viene lanciata una *fault* o una *trap*, il processore cerca nella IDT se esiste un handler corrispondente (segnalato attraverso un bit nell'IDT stessa, alla riga della tabella corrispondente all'eccezione considerata). Nel caso questo non esista, si riprova con la *fault* di *doppia eccezione*, che quindi rappresenta una *fault* a sé. Nel caso nemmeno questo handler esista, viene lanciata una *fault* di *tripla eccezione*, che è di tipo *abort* e comporta quindi l'arresto del programma.

Vediamo quindi un programma di esempio delle eccezioni, che gestisce ad esempio la divisione per zero (tipo 0x00 nella IDT), di cui la parte C++:

```

1 #include <libce.h>
2
3 extern "C" void c_divzero(natq rip) {
4     printf("E' successo qualcosa di brutto a %lx\n", rip);
5 }
6 extern "C" void a_divzero();
7
8 int main() {
9     // imposta interruzione per fault divisione
10    gate_init(0, a_divzero);
11
12    volatile int a = 3;
13    a /= 0; // il qualcosa di brutto
14
15    return 0;
16 }

```

e la parte assembly:

```

1 .global a_divzero
2 a_divzero:
3     // non abbiamo bisogno di salvare o caricare registr
4     mov(%rsp), %rdi // restituisci IP
5     call c_divzero
6     iretq

```

Notiamo che questo è il primo esempio che vediamo di valore di ritorno dal gestore di eccezione: il valore di RIP al momento dell'interruzione, che viene passato nel registro %RDI (come definisce l'ABI System V).

### 1.1.1 Eccezioni e debug

Un'interruzione particolare è quella rappresentata da **INT3**, l'interruzione di *debug*. Attraverso questa, un *debugger* è capace di interrompere l'esecuzione di un programma ad un certo indirizzo del suo codice macchina.

Un'altra interruzione di debug è data dalla *single step*, che viene lanciata ad ogni istruzione quando è attivo un certo flag (appunto, il flag *single step*). Questo permette al debugger di eseguire il programma in modalità *passo singolo*, cioè eseguendo un'istruzione e interrompendo, permettendo al programmatore di osservare il suo andamento passo per passo.

Possiamo sfruttare queste interruzioni di debug per realizzare il meccanismo dei **breakpoint**, cioè per interrompere un programma arbitrario ad una sua istruzione qualsiasi, per poi riprendere l'esecuzione esattamente da tale istruzione. Vediamo due esempi specifici:

- **Breakpoint con la sola INT3:** prendiamo la seguente funzione C/C++:

```
1 void foo(){
2     printf("sono la funzione foo\n");
3 }
```

che disassembla in:

1	0: 55	push	%rbp	# prologo
2	1: 48 89 e5	mov	%rsp,%rbp	
3	4: bf 00 00 00 00	mov	\$0x0,%edi	# chiama printf
4	9: b8 00 00 00 00	mov	\$0x0,%eax	
5	e: e8 00 00 00 00	call	13 <_Z3foov+0x13>	
6	13: 5d	pop	%rbp	# ritorna
7	14: c3	ret		

L'obiettivo potrebbe essere quello di interrompere la funzione nella fase di prologo, cioè all'istruzione **PUSH** `%RBP` di codifica 55. L'idea è quella di prendere tale istruzione, salvarla da qualche parte per poterla reintrodurre in seguito, e sostituirla con una **INT3**, in modo che si possa prestabilire un gestore dell'eccezione da questa lanciata che metta in pausa il programma e rimetta a posto il byte modificato. Potremo allora usare il seguente codice:

```
1 #include <libce.h>
2
3 // conterra' il bit da salvare
4 char saved_byte;
5
6 void foo() {
7     printf("sono la funzione foo\n");
8 }
9
10 // questa funzione interrompe all'int3
11 extern "C" void a_debug();
12 extern "C" void c_debug(void** p) {
13     // p contiene il puntatore a %rsp
14
15     // mette in pausa
16     pause();
17
18     // poi rimette tutto a posto
19     auto func_p = reinterpret_cast<char**>(p);
20     --(*func_p); // e' l'istruzione precedente
```

```

21  **func_p = saved_byte;
22  }
23
24
25  // questa funzione mette il breakpoint
26  void add_breakpoint(void (*func)(void)) {
27      // salva in saved_byte
28      auto func_p = reinterpret_cast<char*>(func);
29      saved_byte = *func_p;
30      printf("saved_byte: %x\n", saved_byte);
31
32      // al suo posto mette 0xcc (int3)
33      *func_p = 0xcc;
34  }
35
36  extern "C" void main() {
37      // inizializza il gate int3 con a_debug()
38      gate_init(3, a_debug);
39
40      // aggiungi il breakpoint
41      add_breakpoint(foo);
42
43      // qui arresta
44      foo();
45      // qui no
46      foo();
47
48      pause();
49  }

```

La variabile `saved_byte` conterrà il byte da reinserire dopo l'interruzione. La funzione `add_breakpoint()` si occuperà allora di salvare il byte giusto e sostituirlo con l'istruzione `INT3` (codice `0xcc`). A questo punto la funzione `a_debug()`, che avrà il compito mettere a primo argomento (registro `RDI` secondo l'ABI di System V) il puntatore allo stack di chiamare la `c_debug()`, che metterà in pausa l'esecuzione, rimetterà a posto `saved_byte` e decrementerà l'istruzione pointer (ricordiamo che la `INT3` è di tipo *fault*, quindi salva l'istruzione pointer *dopo* l'ultima istruzione eseguita).

Il funzionamento della parte assembler, cioè della `a_debug()`, si riduce a:

```

1  #include "libce.h"
2
3  .global a_debug
4  .extern c_debug
5
6  a_debug: # handler interruzione int3
7      salva_registri
8
9      # passa il puntatore alla pila come primo argomento
10     leaq 120(%rsp), %rdi
11     call c_debug
12
13     carica_registri
14     iretq

```

dove l'offset di 120 è dato dai registri, salvati dalla macro `salva_registri`, che occupano 120 byte sullo stack.

- **Breakpoint con INT3 e single step:** un problema dell'esempio precedente è che, come si riporta anche nei commenti, dopo la prima interruzione non si interrompe più in quanto il contenuto del byte d'istruzione interessato viene ristabilito e non più toccato. Potremmo invece voler rimettere la `INT3` dopo la sua esecuzione, così da permettere interruzioni ogni volta che si torna sull'istruzione (che è come funzionano i breakpoint di programmi reali come *GDB*). Facciamo questo sfruttando la modalità *single-step*: al momento dell'interruzione, la attiviamo, facciamo un singolo passo e rimettiamo la `INT3` sull'istruzione. Per fare ciò, salviamo l'indirizzo dell'istruzione, che non è altro dell'indirizzo precedente all'istruzione pointer corrente al momento della gestione dell'interruzione `INT3`. In codice abbiamo quindi:

```

1  #include <libce.h>
2
3  char saved_byte;
4  char* saved_byte_addr;
5
6  void foo() {
7      printf("foo e' in esecuzione\n\n");
8  }
9
10 // questa funzione interrompe all'int3
11 extern "C" void a_debug();
12 extern "C" void c_debug(void** p) {
13     // p contiene il puntatore a %rsp
14
15     // *p e' %rip, decrementa (vogliamo ripartire da rip - 1, e questo
16     // sarà
17     // l'indirizzo salvato nello stack che iretq andrà a riprendersi)
18     (*p)--;
19
20     // prende il vecchio %rip come indirizzo del byte salvato
21     saved_byte_addr = reinterpret_cast<char*>(*p);
22
23     pause();
24
25     // poi rimettette tutto a posto
26     *saved_byte_addr = saved_byte;
27 }
28
29 // questa funzione interrompe al single step
30 extern "C" void a_sstep();
31 extern "C" void c_sstep() {
32     // saved_byte_conterra' il byte che abbiamo reinserito
33
34     // rimette 0xcc (int3)
35     *saved_byte_addr = 0xcc;
36 }
37
38 // questa funzione mette il breakpoint
39 void add_breakpoint(void (*func)(void)) {
40     // salva in saved_byte
41     auto func_p = reinterpret_cast<char*>(func);
42     saved_byte = *func_p;
43     printf("saved_byte: %x\n", saved_byte);
44
45     // al suo posto mette 0xcc (int3)
46     *func_p = 0xcc;
47 }

```



```

47
48 extern "C" void main() {
49     // inizializza il gate int3 con a_debug()
50     gate_init(3, a_debug);
51     // inizializza il gate single_step() con a_sstep()
52     gate_init(1, a_sstep);
53
54     // aggiungi il breakpoint
55     add_breakpoint(foo);
56
57     // qui arresta
58     foo();
59
60     // qui pure
61     foo();
62
63     pause();
64 }

```

Le funzioni assembler (`a_debug()` e `a_sstep()`) modificano il registro `RFLAGS` per attivare e disattivare, rispettivamente, la modalità single-step, come segue:

```

1 #include "libce.h"
2
3 .global a_debug, a_sstep
4 .extern c_debug, c_sstep
5
6 a_debug: # handler interruzione int3
7     salva_registri
8
9     # passa il puntatore alla pila come primo argomento
10    leaq 120(%rsp), %rdi
11    call c_debug
12
13    carica_registri
14
15    orw $0x100, 16(%rsp) # attiva la single step in eflags
16                        # la pila e':
17                        # $rsp vecchio rip
18                        # $rsp+8 vecchio cs
19                        # $rsp+16 vecchio rflags
20                        # a questo punto TF e' a 0x100 in rflags
21
22    iretq
23
24 a_sstep: # handler single step
25     salva_registri
26
27     call c_sstep
28
29     carica_registri
30
31     andw $0xFEFF, 16(%rsp) # disattiva la single step in eflags
32                           # come sopra, la maschera e' complementare
33
34    iretq

```

## 1.2 Riassunto sui tipi di interruzioni

Abbiamo quindi visto tutti i tipi di interruzione, di cui riportiamo una lista completa:

- **Interruzioni esterne:** causate da interfacce esterne e gestite dall'APIC I/O, di cui distinguiamo:
  - **Interruzioni esterne mascherabili:** quelle che abbiamo visto finora, relative a normali eventi I/O;
  - **Interruzioni esterne non mascherabili:** cioè che non possono essere mascherate, solitamente rappresentano eventi particolarmente gravi o comunque la cui gestione ha alta importanza.
- **Interruzioni interne (*Eccezioni*):** eventi che non arrivano dall'esterno, ma si generano all'interno del processore stesso;
- **Interruzioni software:** interruzioni che vengono lanciate direttamente dal programma attraverso l'istruzione **INT**, la cui utilità è stata per ora dimostrativa, e verrà inquadrata meglio studiando il meccanismo della *protezione*, e in generale lo sviluppo del sistema multiprogrammato e delle relative *primitive*.