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++:

```
#include <libce.h>
#define KBD_VECT 0x20

bool fine = false;

extern "C" void a_keyboard();
extern "C" void c_keyboard() {
  // leggiamo da tastiera
  natb code = kbd::get_code();

  if(code == 0x01) fine = true;

  char c = kbd::conv(code);
  vid::char_write(c);

  apic::send_EOI();
}

void main() {
  // attiva le interruzioni tastiera
  kbd::enable_intr();

  // imposta l'APIC
  apic::set_VECT(1, KBD_VECT);
  apic::set_TRGM(1, false); // false: fronte, true: livello
  apic::set_MIRQ(1, false);

  // imposta il gate nella IDT
  gate_init(KBD_VECT, a_keyboard);

  while(!fine);

  return;
}
```

e la parte assembly:

```
#include <libce.h>

.global a_keyboard
.extern c_keyboard

a_keyboard:
  salva_registri
  call c_keyboard
  carica_registri
  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 interruzini 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. Vediamo che in verità esiste un altra casistica di applicazione delle interruzioni che non abbiamo trattato, cioè quella delle *eccezioni*.

1.1 Eccezioni

Ci rimangono da vedere le **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 interpetando 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++:

```
#include <libce.h>

extern "C" void c_divzero(natq rip) {
  printf("E' successo qualcosa di brutto a %lx\n", rip);
}
extern "C" void a_divzero();

int main() {
  // imposta interruzione per fault divisione
  gate_init(0, a_divzero);

  volatile int a = 3;
  a /= 0; // il qualcosa di brutto

  return 0;
}
```

e la parte assembly:

```
.global a_divzero
a_divzero:
  // non abbiamo bisogno di salvare o caricare registr
  mov (%rsp), %rdi // restituisci IP
  call c_divzero
  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++:

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

che disassembla in:

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

```
#include <libce.h>

// conterra' il bit da salvare
char saved_byte;

void foo() {
  printf("sono la funzione foo\n");
}

// questa funzione interrompe all'int3
extern "C" void a_debug();
extern "C" void c_debug(void** p) {
  // p contiene il puntatore a %rsp

  // mette in pausa
  pause();

  // poi rimette tutto a posto
  auto func_p = reinterpret_cast<char**>(p);
  --(*func_p); // e' l'istruzione precedente
  **func_p  = saved_byte;
}


// questa funzione mette il breakpoint
void add_breakpoint(void (*func)(void)) {
  // salva in saved_byte
  auto func_p = reinterpret_cast<char*>(func);
  saved_byte = *func_p;
  printf("saved_byte: %x\n", saved_byte);

  // al suo posto mette 0xcc (int3)
  *func_p = 0xcc;
}

extern "C" void main() {
  // inizializza il gate int3 con a_debug()
  gate_init(3, a_debug);

```

```
  // aggiungi il breakpoint
  add_breakpoint(foo);

  // qui arresta
  foo();
  // qui no
  foo();

  pause();
}
```

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'instruction pointer (ricordiamo che la `INT3` è di tipo *fault*, quindi salva l'instruction pointer *dopo* l'ultima istruzione eseguita).

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

```
#include "libce.h"

.global a_debug
.extern c_debug

a_debug: # handler interruzione int3
  salva_registri

  # passa il puntatore alla pila come primo argomento
  leaq 120(%rsp), %rdi
  call c_debug

  carica_registri
  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'instruction pointer corrente al momento della gestione dell'interruzione `INT3`. In codice abbiamo quindi:

```
#include <libce.h>

char saved_byte;
char* saved_byte_addr;

```

```
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
       sara'
16   // l'indirizzo salvato nello stack che iretq andra' a riprenersi)
17   (*p)--;
18
19   // prende il vecchio %rip come indirizzo del byte salvato
20   saved_byte_addr = reinterpret_cast<char*>(*p);
21
22   pause();
23
24   // poi rimettette tutto a posto
25   *saved_byte_addr = saved_byte;
26 }
27
28 // questa funzione interrompe al single step
29 extern "C" void a_sstep();
30 extern "C" void c_sstep() {
31   // saved_byte_conterra' il byte che abbiamo reinserito
32
33   // rimette 0xcc (int3)
34   *saved_byte_addr = 0xcc;
35 }
36
37 // questa funzione mette il breakpoint
38 void add_breakpoint(void (*func)(void)) {
39   // salva in saved_byte
40   auto func_p = reinterpret_cast<char*>(func);
41   saved_byte = *func_p;
42   printf("saved_byte: %x\n", saved_byte);
43
44   // al suo posto mette 0xcc (int3)
45   *func_p = 0xcc;
46 }
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:

```
#include "libce.h"

.global a_debug, a_sstep
.extern c_debug, c_sstep

a_debug: # handler interruzione int3
  salva_registri

  # passa il puntatore alla pila come primo argomento
  leaq 120(%rsp), %rdi
  call c_debug

  carica_registri

  orw $0x100, 16(%rsp) # attiva la single step in eflags
                       # la pila e':
                       # $rsp   vecchio rip
                       # $rsp+8 vecchio cs
                       # $rsp+16  vecchio rflags
                       # a questo punto TF e' a 0x100 in rflags

  iretq

a_sstep: # handler single step
  salva_registri

  call c_sstep

  carica_registri

  andw $0xFEFF, 16(%rsp) # disattiva la single step in eflags
                         # come sopra, la maschera e' complementare

  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 mascherabli:** 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*.