Programmare con le interruzioni

G. Lettieri

19 Marzo 2024

Scriviamo dei semplici programmi che sfruttino il meccanismo delle interruzioni, prima considerando una singola sorgente di interruzioni e poi due sorgenti contemporaneamente.

1 Singola sorgente (tastiera)

Modifichiamo il programma di esempio della tastiera[1] in modo da utilizzare il meccanismo delle interruzioni. Vogliamo mostrare quali sono i problemi a cui bisogna prestare attenzione quando si scrivono questo tipo di programmi, quindi scriveremo prima una versione apparentemente corretta, ma che in realtà contiene diversi errori, e poi la miglioreremo un po' alla volta. I sorgenti di questo esempio si trovano in `interrupt-1` nel pacchetto `esempiIO`. In particolare, la sottodirectory `errato` contiene il file riportato parzialmente in Figura 1.

Il controllore della tastiera può essere programmato per generare una richiesta di interruzione quando dispone di un nuovo dato nel registro RBR. Per farlo occorre scrivere il byte 0x60 nel registro CMR (indirizzo 0x64 dello spazio di I/O) seguito dal byte 0x61 nel registro TBR (indirizzo 0x60 dello spazio di I/O). Una volta abilitate, le interruzioni possono essere disabilitate scrivendo 0x60 sia in CMR che in TBR. Dopo aver inviato una richiesta di interruzione, il controllore non ne invia una nuova fino a quando il software non legge il registro RBR.

In Figura 1 vediamo la prima versione del codice. L'idea è di fare in modo che la funzione `tastiera()` (righe 12–22) venga eseguita ogni volta che il controllore della tastiera invia una richiesta di interruzione. Per farlo è necessario sapere a quale piedino del controllore APIC la tastiera è collegata: nel nostro caso è il piedino numero 1. Quindi si deve scegliere una entrata libera della Interrupt Descriptor Table, per esempio la numero 0x40 (si noti che le prime 32 non sono utilizzabili, per motivi che vedremo). Il numero 0x40 sarà il *tipo* (o vettore) dell'interruzione. Dobbiamo:

- configurare l'APIC in modo che possa inviare tale tipo alla CPU ogni volta che inoltra una richiesta di interruzione proveniente dal piedino 1;
- scrivere l'indirizzo della funzione `tastiera()` nel gate 0x40 della IDT.

[1] Programma `tastiera-1` di `esempiIO`.

```
1  // ...
2  namespace kbd {
3  void enable_intr()
4  {
5          outputb(0x60, iCMR);
6          outputb(0x61, iTBR);
7  }
8  // ...
9  }
10 // ...
11 bool fine = false;
12 void tastiera()
13 {
14         natb c = inputb(iRBR);
15         if (c == 0x01)  // make code di ESC
16                 fine = true;
17         for (int i = 0; i < 8; i++) {
18                 vid::char_write('0' + !!(c & 0x80));
19                 c <<= 1;
20         }
21         vid::char_write('\n');
22 }
23 const natb KBD_VECT = 0x40;
24 void main()
25 {
26         char spinner[] = { '|', '/', '-', '\\' };
27
28         vid::clear(0x0f);
29         apic::set_VECT(1, KBD_VECT);
30         gate_init(KBD_VECT, tastiera);
31         apic::set_MIRQ(1, false);
32         kbd::enable_intr();
33
34         natq spinpos = 0;
35         while (!fine) {
36                 vid::video[12*80+40] =
37                         vid::attr | spinner[spinpos % 4];
38                 spinpos++;
39         }
40 }
```

Figura 1: File `interrupt-1/errato/intr.cpp`. Versione 0.

Per eseguire queste due operazioni utilizzeremo delle funzioni già presenti nella libreria. Alla riga 29 creiamo l'associazione tra il piedino 1 e il tipo 0x40. La funzione `apic::set_VECT()` scrive nell'opportuno registro interno dell'APIC. Alla riga 30 inseriamo il puntatore alla funzione nell'entrata numero 0x40 della IDT. Si noti che il gate preparato dalla funzione `gate_init()` ha il flag TI a 0, quindi il processore azzererà IF in **`rflags`** prima di saltare alla routine.

L'APIC può disabilitare ogni piedino in modo indipendente e, quando il programma viene caricato, la libreria *libce* provvede a disabilitare tutti i piedini. Alla riga 31 riabilitiamo il piedino 1.

Infine, alle righe 5–6 programmiamo il controllore della tastiera in modo che invii richieste di interruzione. Da questo momento in poi, mentre la CPU sta eseguendo il ciclo 34–39, la pressione di un tasto causerà l'esecuzione del codice in 12–22. Per rendere visivamente il flusso di controllo nel programma principale, facciamo in modo che questo animi uno *spinner* in una certa posizione dello schermo. L'animazione è ottenuta sovrascrivendo continuamente una certa posizione dello schermo con una sequenza di caratteri che somiglia ad una barra in rotazione (linee 26 e 34–39).

Dobbiamo pensare di avere ora a disposizione due flussi di controllo: quello normale, del programma principale, e quello della routine di interruzione, che si inserisce in modo imprevedibile in quello principale. Proprio per via di questa imprevedibilità non abbiamo altro modo di passare informazioni tra il codice principale e la routine di interruzione, se non tramite variabili globali. Questo è lo scopo della variabile `fine` (linea 11), settata a **`true`** quando l'utente preme ESC (linea 16) e controllata continuamente dal programma principale (linea 35) per sapere quando terminare.

1.1 Primo errore: `ret` vs `iretq`

Se si prova a caricare ed eseguire questo programma sulla macchina virtuale, noteremo subito un problema: la prima pressione di un tasto mostrerà la stampa del corrispondente codice di scansione, ma premendo altri tasti non verrà mostrato più niente (non viene mostrato neanche il break code che la tastiera ha inviato in seguito al rilascio del primo tasto che abbiamo premuto).

1.1.1 Debugging

Proviamo a collegarci con il debugger: avviamo QEMU con `boot -g`, apriamo un altro terminale e ci portiamo nella stessa directory, quindi eseguiamo `debug`. Osserviamo la voce "`intr: abilitati`" nella sezione "protezione": questa ci dice che, in questo momento, la CPU della macchina QEMU ha il bit IF settato, e dunque è pronta ad accettare interruzioni. Impostiamo un breakpoint all'inizio della funzione `tastiera()` (comando `b *tastiera`) e lasciamo proseguire l'esecuzione (comando `c`), selezioniamo la finestra di QEMU (facendo attenzione a cliccare solo sulla barra della finestra, per evitare che il mouse venga catturato da QEMU) e premiamo un tasto. L'esecuzione di QEMU dovrebbe bloccarsi e dovrebbe ricomparire il prompt nel debugger. Il debugger ci mostra che la CPU

```
1  // ...
2  extern "C" void a_tastiera()
3  extern "C" void c_tastiera()
4  {
5    // decodifica e stampa del codice in RBR
6  }
7  const natb KBD_VECT = 0x40;
8  void main()
9  {
10         char spinner[] = { '|', '/', '-', '\\' };
11
12         vid::clear(0x0f);
13         apic::set_VECT(1, KBD_VECT);
14         gate_init(KBD_VECT, a_tastiera);
15         apic::set_MIRQ(1, false);
16         kbd::enable_intr();
17
18         // animazione dello spinner
19 }
```

```
1  .extern        c_tastiera
2  .global        a_tastiera
3  a_tastiera:
4                 call c_tastiera
5                 iretq
```

Figura 2: Tastiera, versione 1.

di QEMU sta per eseguire la prima istruzione di `tastiera()`, e dunque ci troviamo un istante dopo l'accettazione della richiesta di interruzione da parte della tastiera. Notiamo che il debugger ci dice "`intr: disabilitate`". Lasciamo proseguire la funzione `tastiera()` fino alla sua conclusione (comando `fin`). Ora l'esecuzione è tornata al programma principale, ma le interruzioni risultano ancora disabilitate.

Il problema è che il processore ha disabilitato le interruzioni nel momento in cui è saltato a `tastiera()` (TI=1 nel gate 0x40) e poi nessuno le ha riabilitate. Si noti che il processore, prima di porre IF=0 in **`rflags`**, aveva salvato il vecchio valore del registro in pila, ma la **`ret`** che si trova in fondo a `tastiera()` preleva soltanto il valore salvato di **`rip`**. Per prelevare anche **`rflags`** serve una **`iretq`**[2].

Modifichiamo allora il programma come in Figura 2. Spezziamo il driver in due parti, una scritta in assembler (`a_tastiera`) e una scritta in C++ (`c_tastiera`). Non inseriamo nel gate 0x40 direttamente il puntatore a

[2]Vedremo in seguito che il processore salva anche altre informazioni oltre a **`rip`** e **`rflags`**. L'istruzione **`iretq`** preleva correttamente anche queste.

`c_tastiera()`, ma passiamo prima dalla funzione `a_tastiera`, che chiama `c_tastiera()` e poi invoca **`iretq`**. Si faccia attenzione a non omettere la `q` finale: l'istruzione **`iret`** esiste anch'essa ed è la versione a 32 bit che non può funzionare nel nostro caso.

1.2 Secondo errore: EOI

Se proviamo a eseguire questo nuovo programma vediamo che il comportamento è apparentemente lo stesso: non viene stampato più niente dopo la prima pressione di un tasto.

1.2.1 Debugging

Avviamo nuovamente la macchina con `boot -g` e colleghiamoci con il debugger. Inseriamo un breakpoint in `a_tastiera` (`b *a_tastiera`), lasciamo proseguire l'esecuzione (`c`) e premiamo un tasto nella finestra di QEMU, come prima. Riottenuto il prompt nel debugger, usiamo il comando `apic` per farci mostrare lo stato dell'APIC. Notiamo che, nel registro ISR, il bit associato al tipo 0x40 (primo bit del quinto gruppo contando da destra) è settato: l'APIC pensa (correttamente) che sia in corso la gestione della richiesta di interruzione proveniente dalla tastiera. Lasciamo terminare la funzione `a_tastiera` (`fin`) in modo che l'esecuzione torni al programma principale. Se proviamo ad eseguire di nuovo il comando `apic`, vedremo che l'APIC pensa che la gestione dell'interruzione 0x40 sia *ancora* in corso.

Il problema è dunque questo: non abbiamo inviato l'End Of Interrupt al controllore APIC. Se non riceve l'EOI, il controllore APIC non inoltra al processore le altre interruzioni che arrivano dalla tastiera.

Corregiamo dunque il programma come in Figura 3. Invochiamo la funzione `apic::send_EOI()` prima di terminare la routine di interruzione (linea 23).

Questo programma sembra ora funzionare. Eseguendolo si vedrà lo spinner ruotare in continuazione. Contemporaneamente, ogni volta che si preme un tasto sulla tastiera, si vedranno apparire i soliti codici di scansione, stampati dalla funzione `c_tastiera()`.

È importante ricordare, però, che il processore sta eseguendo un solo programma ad ogni istante. Ogni volta che riceve una interruzione dal controllore della tastiera (tramite il controllore APIC), salta all'indirizzo della funzione `a_tastiera()`. Quando poi incontra la **`iretq`**, ritorna al programma principale. Per tutto il tempo in cui sta girando il driver della tastiera il programma principale è fermo. Possiamo rendere la cosa molto più evidente se allunghiamo artificialmente la durata della funzione `c_driver()`, aggiungendo un ciclo che incrementa un milione di volte (o più) una variabile `j`, dentro il ciclo delle linee 18–21. Se ora proviamo a caricare ed eseguire il nuovo programma vedremo di nuovo lo spinner ruotare. Appena premiamo un tasto vediamo apparire le cifre del codice di scansione corrispondente e lo spinner fermarsi visibilmente. Lo spinner riprende a ruotare solo quando il driver della tastiera è terminato.

```
1  // ...
2  namespace kbd {
3  void enable_intr()
4  {
5          outputb(0x60, iCMR);
6          outputb(0x61, iTBR);
7  }
8  // ...
9  }
10 // ...
11 bool fine = false;
12 extern "C" void a_tastiera()
13 extern "C" void c_tastiera()
14 {
15         natb c = inputb(iRBR);
16         if (c == 0x01)  // make code di ESC
17                 fine = true;
18         for (int i = 0; i < 8; i++) {
19                 vid::char_write('0' + !!(c & 0x80));
20                 c <<= 1;
21         }
22         vid::char_write('\n');
23         apic::send_EOI();
24 }
25 const natb KBD_VECT = 0x40;
26 void main()
27 {
28         char spinner[] = { '|', '/', '-', '\\' };
29
30         vid::clear(0x0f);
31         apic::set_VECT(1, KBD_VECT);
32         gate_init(KBD_VECT, a_tastiera);
33         apic::set_MIRQ(1, false);
34         kbd::enable_intr();
35
36         natq spinpos = 0;
37         while (!fine) {
38                 vid::video[12*80+40] =
39                         vid::attr | spinner[spinpos % 4];
40                 spinpos++;
41         }
42 }
```

Figura 3: Tastiera, versione 2.

```
1  #include <libce.h>
2  .extern        c_driver
3  .global        a_driver
4  a_driver:
5                 salva_registri
6                 call c_driver
7                 carica_registri
8                 iretq
```

Figura 4: Tastiera, versione 3.

Se premiamo un tasto e poi lo rilasciamo prima che la stampa del make code sia finita, il driver farà in tempo a ripartire una seconda volta senza che il programma principale abbia alcuna possibilità di andare in esecuzione: la tastiera genererà l'interruzione dovuta al rilascio del tasto (break code pronto) mentre ancora è in esecuzione il driver. Se il riconoscimento è sul livello il controllore APIC non la vedrà subito, ma la vedrà appena il driver avrà inviato l'EOI, quindi la inoltrerà al processore. Il processore potrebbe anche esso non accettarla subito, in quando il driver gira a interruzioni disabilitate (le interruzioni erano state disabilitate quando era stata accettata l'interruzione precedente), ma arrivato alla **iretq** le interruzioni saranno nuovamente abilitate (grazie al ripristino del contenuto di **rflags** precedente). Dunque, subito dopo la **iretq**, il processore salterà nuovamente al driver. Vedremo dunque lo spinner restare fermo mentre vengono stampati sia il make code che il break code.

1.3 Terzo errore: registri *scratch*

Anche se sembra funzionare, il programma contiene in realtà alcuni errori molto insidiosi. Un errore si manifesta se abilitiamo le ottimizzazioni nel compilatore (opzione `-O`n con $1 \leq n \leq 3$). Per esempio, lanciamo lo script `compile` nel seguente modo

```
CFLAGS=-O2 compile
```

Se ora rieseguiamo, vedremo che il nostro programma mostra strani caratteri al posto dello spinner[3]

Questo problema nasce dall'uso dei registri da parte di `c_tastiera()`. Il compilatore non salverà e ripristinerà il valore dei registri **rsi**, **rdi** etc., perché per lui `c_tastiera()` è una normalissima funzione, che segue le regole di aggancio di tutte le funzioni C++. Il problema è che ora questa funzione può inserirsi tra due istruzioni qualunque del programma principale (se le interruzioni sono abilitate). Che succede se si inserisce tra il caricamento di uno di questi registri e il suo successivo uso? Il programma principale si troverà ad

[3]L'errore può manifestarsi in questa forma o in altre (compreso nessuna) in base ai dettagli della compilazione del programma, che dipendono anche dalla versione del compilatore.

usare il contenuto scorretto dei registri. In particolare, il programma ottimizzato conserva in `esi` la variable `attr`, che contiene l'attributo colore nella parte alta e 0 nella parte bassa, ma la funzione `vid::char_write()`, invocata da `c_tastiera()`, scrive in questo registro senza prima salvarne e poi ripristinarne il contenuto, in quanto si tratta di un registro *scratch*. Il valore scritto, in particolare, contiene dei bit a 1 anche nella parte bassa, quindi l'OR alla linea 39 modificherà il codice ASCII del carattere da mostrare[4] Per rimediare a questo, salviamo preventivamente e poi ripristiniamo tutti i registri intorno alla chiamata di `c_tastiera()` in `a_tastiera` (righe 5 e 7 in Figura 4). Per farlo utilizziamo due *macro* definite nella libreria, che includiamo alla riga 1. Si noti che, in generale, sarebbe sufficiente salvare solo i registri effettivamente utilizzati dalla routine di interruzione (e dalle eventuali routine da essa chiamate, direttamente o indirettamente), ma occorrerebbe una accurata analisi per identificarli tutti. Come compromesso, in caso di routine scritte in C++, potremmo salvare solo i registri scratch.

1.4 Quarto errore: `volatile`

Una volta risolto questo problema, ne osserviamo un altro: il nostro programma, quando compilato con le ottimizzazioni, non termina più quando premiamo ESC. Il problema è che il compilatore C++ non sa dell'esistenza delle interruzioni e assume che niente possa interferire con l'esecuzione di una funzione. Osservando il ciclo 37–41 vedrà che (secondo lui) niente può modificare `fine` dopo il primo test, e dunque è inutile rileggere la variabile dopo la prima volta. Quindi il nostro **`while`** verrà tradotto così:

```
if (!fine) while (true) { // ciclo infinito!
          // animazione dello spinner
}
```

Dobbiamo informare il compilatore del fatto che la variabile `fine` può cambiare il proprio valore anche se niente sembra modificarla nel ciclo 37–41. Questo si ottiene dichiarandola **`volatile`**.

Ricapitolando:

- le funzioni che vogliamo eseguire in risposta ad una richiesta di interruzione devono:
  - salvare e ripristinare *tutti* i registri che usano (o tutti i registri, per semplicità);
  - inviare l'EOI al controllore APIC subito prima di terminare;
  - eseguire una **`iretq`** come ultima istruzione.
- variabili usate sia da una routine di interruzione, sia dal programma principale devono essere dichiarate **`volatile`**.

[4] Il codice che si ottiene in questo caso potrebbe anche non essere ASCII, se il bit più significativo diventa 1; in quel caso verrebbero stampati caratteri presi dalla *code page 437*, che era quella usata per default dalle schede VGA in modalità testo.

Per quanto riguarda l'ultimo punto, si noti che in realtà il problema è molto più complesso e lo affronteremo più avanti. Per il momento è bene limitarsi a condividere al più una singola variabile.

2 Singola sorgente (timer)

Proviamo ora a utilizzare, come singola sorgente di richieste di interruzioni, il contatore 0 dell'interfaccia di conteggio, il cui piedino OUT è collegato al piedino 2 del controllore APIC. Il codice di questo esempio si trova nella directory `interrupt-2` di `esempiIO` ed è riportato in Figura 5.

Associamo la funzione `a_timer` al piedino 2, tramite il tipo 0x50 (linee 23–24). Abilitiamo quindi l'APIC ad accettare richieste sul piedino 2 (linea 27). Programmiamo il contatore 0 in modo 3: trigger software, ciclo continuo (ricaricamento automatico del contatore a fine conteggio), generazione di un impulso a fine conteggio (linee 7-9). La costante scelta genererà una nuova richiesta ogni 50ms circa.

La routine che andrà in esecuzione per effetto della richiesta periodica di interruzione si limita ad incrementare un contatore (linea 15), che viene poi stampato dal programma principale dopo che è passato un po' di tempo (linea 31).

In questo esempio ci vogliamo concentrare su una fondamentale differenza, per quanto riguarda le richieste di interruzione, tra le interfacce tipo il timer e le interfacce del tipo della tastiera. Se ripensiamo alla tastiera, ricordiamo che essa, come molte altre interfacce, disattiva la propria richiesta di interruzione quando il software esegue l'azione attesa (nel caso della tastiera, quando il software legge RBR). Possiamo dunque dedurre che, se all'arrivo dell'EOI il controllore APIC vede la richiesta proveniente dalla tastiera come ancora attiva, ci troviamo di fronte ad una *nuova* richiesta di interruzione, che deve essere inoltrata al processore. Ha dunque senso, in questo caso, usare il riconoscimento delle richieste di interruzione "sul livello". A differenza della tastiera, il timer attiva e disattiva il suo piedino OUT senza coordinarsi in alcun modo con il software (in particolare, non attende nessuna risposta alle sue richieste di interruzione). Questo comporta che vedere la richiesta attiva all'arrivo dell'EOI non ha alcun significato (l'APIC potrebbe star vedendo sempre lo stesso impulso già visto). Per questo motivo, se vogliamo evitare richieste di interruzione spurie, conviene che l'APIC usi il riconoscimento *sul fronte* per le richieste provenienti dal timer. Questo si ottiene settando un opportuno flag nella tabella interna dell'APIC. La libreria `libce` fornisce la funzione `apic::set_TRGM()` che ci permette di farlo facilmente: il primo argomento è il numero del piedino e il secondo è un booleano che vale **`true`** per il riconoscimento sul livello e **`false`** per il riconoscimento sul fronte (linea 26). Possiamo osservare facilmente l'effetto dei due modi di riconoscimento facendo girare il programma con l'uno o l'altro valore. Con il riconoscimento sul fronte il programma stamperà un valore di `counter` pari a qualche decina, mentre con il riconoscimento sul livello `counter` assumerà un valore molto più alto, anche di centinaia di migliaia.

```
1  #include <libce.h>
2
3  namespace timer {
4  // ...
5  void start(natw N)
6  {
7          outputb(0b00110111, iCWR);
8          outputb(N, iCTR0_LOW);
9          outputb(N >> 8, iCTR0_HIG);
10 }
11 }
12 volatile natq counter = 0;
13 extern "C" void c_timer()
14 {
15         counter++;
16         apic::send_EOI();
17 }
18
19 extern "C" void a_timer();
20 const natb TIM_VECT = 0x50;
21 void main()
22 {
23         apic::set_VECT(2, TIM_VECT);
24         gate_init(TIM_VECT, a_timer);
25         timer::start(59660);
26         apic::set_TRGM(2, false);
27         apic::set_MIRQ(2, false);
28
29         for (volatile int i = 0; i < 100000000; i++)
30                 ;
31         printf("counter = %d\n", counter);
32         pause();
33 }
```

```
1  .extern        c_timer
2  .global        a_timer
3  a_timer:
4                 salva_registri
5                 call c_timer
6                 carica_registri
7                 iretq
```

Figura 5: File contenuti in `interrupt-2` di `esempiIO`.

Si noti che, almeno in teoria, non possiamo usare il riconoscimento sul fronte anche per la tastiera, perché potrebbe verificarsi il problema opposto: se la tastiera ha già un nuovo codice pronto nel momento in cui il software legge RBR, l'interfaccia potrebbe lasciare attiva la propria richiesta di interruzione senza prima disattivarla. Questa seconda richiesta, dunque, non genererebbe alcun fronte e non verrebbe riconosciuta dall'APIC. In pratica, però, l'emulazione delle periferiche in QEMU genera sempre un nuovo fronte quando è necessario inviare una nuova richiesta di interruzione, quindi possiamo limitarci ad usare sempre soltanto il riconoscimento sul fronte.

3 Due sorgenti (tastiera e timer)

Proviamo ora a usare contemporaneamente entrambe le sorgenti di richieste di interruzione, per vedere come il controllore APIC gestisce le priorità e come varia il flusso di controllo. Il codice di questo esempio si trova nella directory `interrupt-3` di `esempiIO`. La parte C++ è riportata in Figura 6, mentre la parte Assembler non è altro che l'unione di Figura 4 e della parte Assembler di Figura 5.

Anche la parte C++ è sostanzialmente l'unione dei due esempi di Sezione 1 e 2, in particolare per quanto riguarda l'inizializzazione (linee 28–37). Il programma principale e la routine `c_tastiera()` sono sostanzialmente identiche a quelle dell'esempio (corretto) di Sezione 1. A differenza del precedente esempio sul timer, però, questa volta la routine del timer anima anch'essa uno spinner, a destra sullo schermo rispetto a quello animato dal programma principale (linee 13–15). Si noti la necessità di dichiarare **`static`** la variabile locale `timer_spinpos`, in modo che il suo valore sia preservato da un'istanza all'altra della funzione `c_timer()`.

Se proviamo a caricare ed eseguire questo programma vediamo i due spinner ruotare (il secondo più lentamente e più regolarmente). Se premiamo e rilasciamo velocemente un tasto notiamo che entrambi si fermano mentre viene stampato il make code, quindi lo spinner del timer fa un singolo passo, e poi entrambi si fermano di nuovo mentre viene stampato il break code. Questo è ciò che accade:

1. Mentre non stiamo premendo tasti, il programma fa ruotare in continuazione lo spinner a sinistra e, ogni 50ms, fa avanzare anche quello di destra; tutto è molto veloce e non si notano rallentamenti nel primo spinner, ma ogni avanzamento del secondo comporta un salto alla routine del timer;

2. quando premiamo un tasto, l'interfaccia della tastiera invia la richiesta di interruzione, l'APIC la registra in IRR, bit 0x40, e la inoltra al processore, che prima o poi l'accetta, salta a `c_tastiera` e *disabilita le interruzioni*; l'APIC sposta il flag 0x40 da IRR a ISR;

3. durante tutta la durata di `c_tastiera` il timer continua a mandare richieste di interruzione; l'APIC registra la prima in IRR, bit numero 0x50

```
1  // ...
2  volatile bool fine = false;
3  char spinner[] = { '|', '/', '-', '\\' };
4
5  extern "C" void c_tastiera()
6  {
7          // come in Figura 3, ma con un
8          // ciclo for che ne rallenta l'esecuzione
9  }
10
11 extern "C" void c_timer()
12 {
13         static natq timer_spinpos = 0;
14         vid::char_put(spinner[timer_spinpos % 4], 60, 12);
15         timer_spinpos++;
16         apic::send_EOI();
17 }
18
19 extern "C" void a_tastiera();
20 extern "C" void a_timer();
21
22 const natb KBD_VECT = 0x60;
23 const natb TIM_VECT = 0x50;
24 void main()
25 {
26         vid::clear(0x0f);
27
28         apic::set_VECT(1, KBD_VECT);
29         gate_init(KBD_VECT, a_tastiera);
30         apic::set_MIRQ(1, false);
31         kbd::enable_intr();
32
33         apic::set_VECT(2, TIM_VECT);
34         gate_init(TIM_VECT, a_timer);
35         timer::start0(59660);
36         apic::set_TRGM(2, false);
37         apic::set_MIRQ(2, false);
38
39         natq spinpos = 0;
40         while (!fine) {
41                 vid::char_put(spinner[spinpos % 4], 40, 12);
42                 spinpos++;
43         }
44         return 0;
45 }
```

Figura 6: Tastiera e timer, parte C++.

e, visto che ha una *priority class* maggiore di quella in ISR, prova a inoltrarla al processore; questo però non l'accetta perché ha le interruzioni disabilitate, e dunque la richiesta resta in IRR;

4. quando solleviamo il tasto (molto probabilmente quando `c_tastiera` sta ancora girando, vista la sua lentezza), l'interfaccia della tastiera invia una nuova richiesta per il break code; se il riconoscimento è sul fronte, l'APIC la vede e la registra in IRR (bit 0x40), altrimenti la vedrà al prossimo EOI;

5. quando `c_tastiera` arriva ad eseguire `apic::send_EOI()`, l'APIC resetta il bit 0x40 di ISR, registra in IRR la richiesta della tastiera (se non lo aveva già fatto prima) e vede che in IRR ci sono ora due richieste: la 0x40 e la 0x50; inoltra al processore la 0x50, che ha priorità maggiore;

6. il processore la accetta quando raggiunge la **`iretq`** di `a_tastiera`, salta alla routine del timer e fa avanzare di un passo il secondo spinner;

7. il processore arriva alla `apic::send_EOI()` di `c_timer();` è molto probabile che nel frattempo non sia arrivata una nuova richiesta dal timer (50ms sono tanti), dunque l'APIC si ritrova tra le richieste pendenti solo quella in 0x40, e la inoltra al processore;

8. quando il processore arriva alla **`iretq`** di `a_timer` accetta la nuova richiesta, salta alla routine della tastiera e stampa il break code; durante questo tempo le interruzioni sono nuovamente disabilitate e dunque lo spinner del timer è fermo;

9. è molto probabile che in tutto questo tempo il processore non sia mai riuscito a tornare al programma principale (c'è sempre una interruzione pendente quando arriva alle **`iretq`**) e dunque il primo spinner resta fermo tutto il tempo.

È possibile fare in modo che il processore non disattivi automaticamente le interruzioni quando ne accetta una. L'opzione può essere decisa tipo per tipo, in base ad un flag nella corrispondente entrata della tabella IDT. Per vedere cosa succede in questo caso, passiamo **`true`** come terzo parametro di `gate_init()` alla linea 29 di Figura 6. In questo modo le interruzioni non saranno disabilitate mentre è in esecuzione il driver della tastiera. Se proviamo a caricare il programma così modificato vedremo i due spinner ruotare come al solito. Se premiamo un tasto, il primo spinner (quello del programma principale) si ferma, ma il secondo continua a ruotare indisturbato. Questo perché ora il processore accetta le richieste per il tipo 0x50 che arrivano dall'APIC, interrompendo quindi periodicamente il driver della tastiera per saltare a quello del timer. Abbiamo dunque un annidamento delle interruzioni.

Altre cose da provare:

- che succede se diamo il tipo 0x40 al timer e 0x50 alla tastiera?

- che succede se diamo il tipo 0x55 al timer e 0x50 alla tastiera?
- che succede se diamo il tipo 0x50 al timer e 0x55 alla tastiera?
- che succede se omettiamo `apic::send_EOI()` nella routine del timer o della tastiera, nelle varie combinazioni di tipi?