Programmare con le interruzioni

G. Lettieri

2 Aprile 2020

Proviamo a modificare il programma di esempio della tastiera (Vol. II) 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.

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 10–21) 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.

Per eseguire queste due operazioni utilizzeremo delle funzioni già presenti nella libreria. Alla riga 31 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 32 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 si 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 33 riabilitiamo il piedino 1.

```
1  #include <libce.h>
2  #include <apic.h>
3
4  const ioaddr iSTR = 0x64;
5  const ioaddr iTBR = 0x60;
6  const ioaddr iRBR = 0x60;
7  const ioaddr iCMR = 0x64;
8
9  bool stop = false;
10 extern "C" void tastiera()
11 {
12         natb c;
13         inputb(iRBR, c);
14         if (c == 0x01)
15                 stop = true;
16         for (int i = 0; i < 8; i++) {
17                 char_write((c & 0x80) ? '1' : '0');
18                 c <<= 1;
19         }
20         char_write('\n');
21 }
22
23 const natb KDB_TYPE = 0x40
24
25 char spinner[] = { '|', '/', '-', '\\' };
26 extern volatile natw *video;
27 extern natb attr;
28 int main()
29 {
30
31         apic_set_VECT(1, KBD_TYPE);
32         gate_init(KBD_TYPE, tastiera);
33         apic_set_MIRQ(1, false);
34         outputb(0x60, iCMR);
35         outputb(0x61, iTBR);
36
37         int i = 0;
38         while (!stop) {
39                 video[12*80+40] = attr << 8 | spinner[i];
40                 i = (i + 1) % sizeof(spinner);
41         }
42 }
```

Figura 1: Versione 0.

Infine, alle righe 34–35 programmiamo il controllore della tastiera in modo che invii richieste di interruzione. Da questo momento in poi, mentre la CPU sta eseguendo il ciclo 37–41, la pressione di un tasto causerà l'esecuzione del codice in 10–21. 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 25 e 37–41).

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 `stop` (linea 9), settata a **`true`** quando l'utente preme ESC (linea 15) e controllata continuamente dal programma princiale (linea 38) per sapere quando terminare.

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). 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`**[1].

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 `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.

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.

Il problema ora è un altro: 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 22). La routine è dichiarata nel file `apic.h` della libreria, che includiamo alla linea 2.

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

```
1  #include <libce.h>
2  #include <apic.h>
3
4  const ioaddr iSTR = 0x64;
5  const ioaddr iTBR = 0x60;
6  const ioaddr iRBR = 0x60;
7  const ioaddr iCMR = 0x64;
8
9  bool stop = false;
10 extern "C" void a_tastiera();
11 extern "C" void c_tastiera()
12 {
13         natb c;
14         inputb(iRBR, c);
15         if (c == 0x01)
16                 stop = true;
17         for (int i = 0; i < 8; i++) {
18                 char_write((c & 0x80) ? '1' : '0');
19                 c <<= 1;
20         }
21         char_write('\n');
22 }
23
24 const natb KDB_TYPE = 0x40
25
26 char spinner[] = { '|', '/', '-', '\\' };
27 extern volatile natw *video;
28 extern natb attr;
29 int main()
30 {
31
32         apic_set_VECT(1, KBD_TYPE);
33         gate_init(KBD_TYPE, a_tastiera);
34         apic_set_MIRQ(1, false);
35         outputb(0x60, iCMR);
36         outputb(0x61, iTBR);
37
38         int i = 0;
39         while (!stop) {
40                 video[12*80+40] = attr << 8 | spinner[i];
41                 i = (i + 1) % sizeof(spinner);
42         }
43 }
```

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

Figura 2: Versione 1.

```
1  #include <libce.h>
2  #include <apic.h>
3
4  const ioaddr iSTR = 0x64;
5  const ioaddr iTBR = 0x60;
6  const ioaddr iRBR = 0x60;
7  const ioaddr iCMR = 0x64;
8
9  bool stop = false;
10 extern "C" void a_tastiera();
11 extern "C" void c_tastiera()
12 {
13         natb c;
14         inputb(iRBR, c);
15         if (c == 0x01)
16                 stop = true;
17         for (int i = 0; i < 8; i++) {
18                 char_write((c & 0x80) ? '1' : '0');
19                 c <<= 1;
20         }
21         char_write('\n');
22         apic_send_EOI();
23 }
24
25 const natb KDB_TYPE = 0x40
26
27 char spinner[] = { '|', '/', '-', '\\' };
28 extern volatile natw *video;
29 extern natb attr;
30 int main()
31 {
32
33         apic_set_VECT(1, KBD_TYPE);
34         gate_init(KBD_TYPE, a_tastiera);
35         apic_set_MIRQ(1, false);
36         outputb(0x60, iCMR);
37         outputb(0x61, iTBR);
38
39         int i = 0;
40         while (!stop) {
41                 video[12*80+40] = attr << 8 | spinner[i];
42                 i = (i + 1) % sizeof(spinner);
43         }
44 }
```

Figura 3: Versione 2.

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 17–20. 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.

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. 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.

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, modifichiamo lo script `compile`, che si trova nella sottodirectory `CE/bin` della directory *home*, aggiungendo `-O2` alle opzioni contenute nella variabile `COMPILER_OPTIONS`. Se ora ricompiliamo e rieseguiamo, vedremo che il nostro programma 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 40–43 vedrà che (secondo lui) niente può modificare `stop` dopo il primo test. Quindi il **`while`** verrà tradotto così:

```
if (!stop)
     while (true) { // ciclo infinito!
          ...
     }
```

```
1  #include "libce.s"
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: Versione 3.

Dobbiamo informare il compilatore del fatto che la variabile `stop` può cambiare il proprio valore anche se niente sembra modificarla nel ciclo 40–43. Questo si ottiene dichiarandola **`volatile`**.

Un ultimo problema è dato 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 usare il contenuto scorretto dei registri. Per rimediare a questo, salviamo 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.

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 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`**.

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 variable.