Programmare con le interruzioni (2)

G. Lettieri

12 Aprile 2018

Nelle Figure 1 e 2 riportiamo una versione diversa dell'esempio visto la volta precedente. Tutta l'elaborazione è stata spostata direttamente nel driver (funzione `c_tastiera()`). Questo comporta una serie di semplificazioni, in quanto la comunicazione tra il driver e il programma principale è ora limitata al solo riconoscimento del codice di ESC, che serve solo a far terminare il programma. In particolare, non è più necessario avere una variabile `nuovo_car` e condividere `c` tra il driver e il programma principale. Il programma principale può accettare interruzioni dalla tastiera in qualunque momento.

Si noti che abbiamo scritto in esadecimale (0x40) il tipo scelto per l'interruzione (righe 31 e 32). In questo modo la scomposizione in *priority class/* e *sub-priority class* è immediata: 4 per per la prima e 0 per la seconda.

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 38–42). Eseguendo il programma 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 come in Figura 3 (linee 20–22). 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

```
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  volatile 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 char spinner[] = { '|', '/', '-', '\\' };
26 extern volatile natw *video;
27 extern natb attr;
28 int main()
29 {
30
31         gate_init(0x40, a_tastiera);
32         apic_set_VECT(1, 0x40);
33         apic_set_MIRQ(1, false);
34         apic_set_TRGM(1, false);
35         outputb(0x60, iCMR);
36         outputb(0x61, iTBR);
37
38         int i = 0;
39         while (!stop) {
40                 video[40] = attr << 8 | spinner[i];
41                 i = (i + 1) % sizeof(spinner);
42         }
43 }
```

Figura 1: Versione 1, parte C++.

```
.extern         c_tastiera
.global         a_tastiera
a_tastiera
                salva_registri
                call c_tastiera
                carica_registri
                iretq
```

Figura 2: Versione 1, parte Assembler.

```
#include <libce.h>
#include <apic.h>

const ioaddr iSTR = 0x64;
const ioaddr iTBR = 0x60;
const ioaddr iRBR = 0x60;
const ioaddr iCMR = 0x64;

volatile bool stop = false;
extern "C" void a_tastiera();
extern "C" void c_tastiera()
{
        natb c;
        inputb(iRBR, c);
        if (c == 0x01)
                stop = true;
        for (int i = 0; i < 8; i++) {
                char_write((c & 0x80) ? '1' : '0');
                c <<= 1;
                volatile int j;
                for (j = 0; j < 10000000; j++)
                        ;
        }
        char_write('\n');
        apic_send_EOI();
}

... // come in Figura 1
```

Figura 3: Versione 2.

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.

Proviamo ora a introdurre una nuova fonte di richieste di interruzione, per vedere come il controllore APIC gestisce le priorità e come varia il flusso di controllo.

Come seconda fonte utilizziamo il contatore 0 dell'interfaccia di conteggio, il cui piedino OUT è collegato al piedino 2 del controllore APIC. In Figura 4 associamo la funzione `a_driver` al piedino 2, tramite il tipo 0x50 (linee 31–32). Abilitiamo quindi l'APIC ad accettare richieste sul piedino 2 (linea 33).

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 35–37). La costante scelta genererà una nuova richiesta ogni 50ms circa.

Alla linea 34 impostiamo il *trigger mode* del piedino 2 in modo che riconosca le richieste sul fronte, invece che sul livello. Questo perché il timer il timer attiva la richiesta e poi la disattiva autonomamente, ma se la routine di interruzione invia l'EOI prima della disattivazione l'APIC potrebbe vedere il segnale ancora attivo e generare una richiesta spuria. Il timer è una eccezione in questo modo di comportarsi: le altre periferiche disattivano la richiesta quando la routine di interruzione compie una certa azione. Alla routine basta eseguire questa azione prima di inviare l'EOI, e il problema non si pone. Il controllore della tastiera funziona in questo modo: disattiva la richiesta quando il software legge RBR, quindi possiamo usare il riconoscimento sul livello (linea 27).

La routine `a_driver` (Figura 5, linee 9–15) è del tutto simile alla funzione `a_tastiera`. L'animazione del nuovo spinner è realizzata dalla `c_timer()` (Figura 4, linee 13–19). Si noti che lo spinner del timer verrà mostrato più a destra rispetto a quello del programma principale.

Se proviamo a caricare ed eseguire questo programma vediamo i due spinner ruotare (il secondo più lentamente e più regolarmente). Se premiamo e rilasciamo 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 processore 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;

```
1  #include <libce.h>
2  #include <apic.h>
3
4  ... // tastiera come in Figura3
5
6  const ioaddr iCWR = 0x43;
7  const ioaddr iCTR0_LOW = 0x40;
8  const ioaddr iCTR0_HIG = 0x40;
9
10 extern volatile natw *video;
11 extern natb attr;
12 extern "C" void a_timer();
13 extern "C" void c_timer()
14 {
15         static int i = 0;
16         video[50] = attr << 8 | spinner[i];
17         i = (i + 1) % sizeof(spinner);
18         apic_send_EOI();
19 }
20
21 int main()
22 {
23
24         gate_init(0x40, a_tastiera);
25         apic_set_VECT(1, 0x40);
26         apic_set_MIRQ(1, false);
27         apic_set_TRGM(1, false);
28         outputb(0x60, iCMR);
29         outputb(0x61, iTBR);
30
31         gate_init(0x50, a_timer);
32         apic_set_VECT(2, 0x50);
33         apic_set_MIRQ(2, false);
34         apic_set_TRGM(2, true); // fronte
35         outputb(0x36, iCWR);
36         outputb((natb)50000, iCTR_LOW);
37         outputb((natb)(50000 >> 8), iCTR_HIG);
38
39         int i = 0;
40         while (!stop) {
41                 video[40] = attr << 8 | spinner[i];
42                 i = (i + 1) % sizeof(spinner);
43         }
44 }
```

Figura 4: Versione 3, parte C++.

```
1  .extern        c_tastiera
2  .global        a_tastiera
3  a_tastiera
4                 salva_registri
5                 call       c_tastiera
6                 carica_registri
7                 iretq
8
9  .extern        c_timer
10 .global        a_timer
11 a_timer
12                salva_registri
13                call       c_timer
14                carica_registri
15                iretq
```

Figura 5: Versione 3, parte assembler.

2. quando premiamo un tasto, il controllore invia l'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 bit 0x40 da IRR a ISR;

3. durante tutta la durata di `c_tastiera` il timer continua a mandare interruzioni; l'APIC registra la prima in IRR, bit numero 0x50 e, visto che priority class maggiore di quella in ISR, prova a inoltrarla al processore; questo però non l'accetta, e dunque la richiesta resta in IRR;

4. quando solleviamo il tasto (molto probabilmente quando ancora `c_driver` sta girando, vista la sua lentezza), il controllore della tastiera invia una nuova richiesta per il break code; anche questa non può essere accetta, ma l'APIC la registra in IRR (bit 0x40);

5. quando `c_driver` arriva ad eseguire `apic_send_EOI()` di `c_tastiera()`, l'APIC resetta il bit 0x40 di ISR e vede che in IRR ci sono due richieste: la 0x40 e la 0x50; inoltra al processore la 0x50, che ha priority class 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, usiamo `trap_init()` invece di `gate_init()` alla riga 24 di Figura 4. 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?