Modulo I/O

G. Lettieri

21 Maggio 2018

La gestione delle interruzioni con il meccanismo del driver è poco flessibile ed efficiente, per due motivi:

- il driver deve essere eseguito con le interruzioni disabilitate, in quanto manipola direttamente le code dei processi;
- il driver non si può bloccare, in quanto non è un processo.

La disabilitazione delle interruzioni può causare problemi, in quanto costringe anche le interruzioni ad alta priorità ad aspettare che il driver termini la propria esecuzione. Il fatto che il driver non si possa bloccare limita fortemente le cose che può fare, soprattutto in un sistema più complicato in cui si debba gestire anche la rete o i file system.

Il secondo problema si può risolvere "trasformando" il driver in un processo. Più precisamente, facciamo in modo che l'interruzione non mandi in esecuzione l'intero driver, ma solo un piccolo *handler* che ha il solo scopo di mandare in esecuzione un processo, il quale si preoccuperà di svolgere le istruzioni che prima erano svolte dal driver.

Il problema delle interruzioni disabilitate può essere ridotto facendo girare questo processo a interruzioni abilitate, identificando tutti i punti in cui accede a strutture dati condivise (nel nostro caso, le code dei processi) e disabilitando le interruzioni solo in quei punti, usando le istruzioni `cli` e `sti`. Questa soluzione, per quanto utilizzata in sistemi reali, comporta però grandi complicazioni nella scrittura del codice. Possiamo invece adottare una soluzione molto più semplice: se il processo non fa parte del nucleo e appartiene invece ad un modulo distinto, che non è collegato con il modulo sistema, non ha modo di accedere alle code dei processi se non invocando delle primitive di sistema. Poiché le primitive di sistema girano a interruzioni disabilitate, ecco che l'accesso alle code dei processi è protetto.

Introduciamo allora un nuovo modulo, detto modulo *I/O*, distinto dal modulo sistema e dal modulo utente. Lo scopo di questo modulo è di realizzare le primitive per l'accesso alle periferiche, gestendo le richieste di interruzione tramite processi, detti processi "esterni" (nel senso che sono esterni al modulo sistema, anche se svolgono funzioni di sistema). Nella nostra implementazione, i file che contengono il codice di questo nuovo modulo si trovano nella cartella

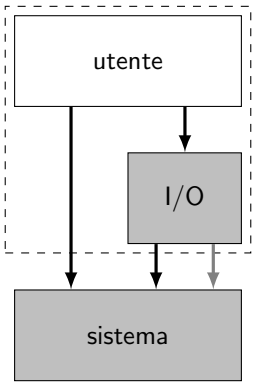

Figura 1: Relazioni tra i moduli. I moduli in grigio sono eseguiti con la CPU a livello sistema. I moduli dentro il riquadro tratteggiato sono eseguiti con le interruzioni mascherabili abilitate. Le frecce rappresentano possibili invocazioni di primitive: le nere sono accessibili anche da livello utente e le grigie solo da livello sistema.

`io` e sono `io.cpp` e `io.s`. La loro compilazione e collegamento produrrà il file `build/io`, che poi verrà caricato in memoria durante l'avvio del sistema.

La Figura 1 mostra le relazioni tra i moduli che compongono il sistema. Si noti che ora l'utente ha accesso sia a primitive che sono realizzate direttamente nel modulo sistema (`sem_ini()`, `sem_wait()`, `sem_signal()`, `delay()`, `activate_p()`, `terminate_p()`), sia a nuove primitive che sono realizzate nel modulo I/O. L'accesso a queste nuove primitive avviene sempre tramite la tabella IDT, con l'esecuzione di una istruzione **`int`**. Le relative entrate della tabella, però, punteranno a funzioni che sono definite nel modulo I/O.

Idealmente vorremmo che il codice contenuto in questo nuovo modulo girasse ad un livello intermedio tra utente e sistema in quanto deve avere più diritti degli utenti (deve poter interagire direttamente con le interfacce di I/O), ma non deve accedere direttamente alle strutture dati del sistema (IDT, GDT, code dei processi, tabelle della paginazione, etc.) Purtroppo il processore dispone di soli due livelli e scegliamo, dunque, di far girare anche questo modulo a livello sistema (in Figura, entrambi i moduli sono su sfondo grigio). Il fatto di compilare separatamente il modulo I/O e il modulo sistema protegge comunque le strutture dati del sistema da molti errori, ma sicuramente non da tutti (per esempio, un accesso errato in memoria da parte del codice I/O può comunque modificare la memoria riservata al modulo sistema).

A differenza del codice del modulo sistema, il codice del modulo I/O girerà a interruzioni abilitate, come il codice del modulo utente. Questo vale sia per il codice dei processi esterni, sia per il codice delle nuove primitive realizzate nel modulo I/O. Eventuali problemi di mutua esclusione dovranno essere risolti utilizzando i semafori forniti dal modulo sistema. Infatti, nella Figura si vede che anche il modulo I/O fa uso di primitive fornite dal modulo sistema; in

```
1    # file: sistema.s
2    handler_i:
3        call salva_stato
4        call inspronti
5        movq $i, %rcx
6        movq a_p(, %rcx, 8), %rax
7        movq %rax, esecuzione
8        call carica_stato
9        iretq
```

Figura 2: Schema generale di un handler.

particolare, usa le primitive semaforiche.

Il modulo I/O ha però accesso anche a primitive ad esso dedicate e non accessibili da livello utente. Per evitare che gli utenti possano invocare queste primitive è sufficiente che il bit GL dei corrispondenti gate sia impostato a Sistema.

Una delle primitive riservate al modulo I/O è la primitiva `activate_pe()`, che serve ad attivare un processo esterno. Questa primitiva ha gli stessi parametri della normale `active_p()`, con un ulteriore parametro che è il numero del piedino dell'APIC da cui arriveranno le richieste a cui il processo deve rispondere. Il modulo sistema avrà una tabella `a_p` con una entrata per ogni piedino dell'APIC. Al termine della `activate_pe()`, dopo aver creato il processo, inserirà il corrispondente `proc_elem` nella opportuna entrata di questa tabella, invece di inserirlo in coda pronti.

Per ogni possibile interruzione, il modulo sistema deve predisporre un handler che si preoccupa di mettere in esecuzione il corrispondente processo esterno, prendendo il `proc_elem` da questa tabella. In fase di inizializzazione il modulo sistema deve programmare l'APIC e riempire le entrate della tabella IDT in modo che ogni richiesta causi un salto al giusto handler.

L'handler associato alla richiesta di interruzione proveniente dal piedino i-esimo avrà la forma mostrata in Figura 2. Alla riga 3 salva lo stato del processo che stava girando quando la richiesta è stata accettata e alla riga 4 lo inserisce forzatamente in testa alla coda pronti (se era in esecuzione, era sicuramente quello a priorità maggiore tra quelli in coda pronti). Nelle righe 5–7 esegue l'equivalente del codice `esecuzione=a_p[i]`. La successiva coppia "**`call`** `carica_stato;` **`iretq`**" (righe 8 e 9) cederà il controllo al processo esterno.

I processi esterni seguiranno tutti lo schema generale mostrato in Figura 3. In Figura si assume che nel sistema ci siano più periferiche simili, ciasuna gestita da un diverso processo esterno, il cui codice può essere però scritto una volta per tutte. Tramite il parametro `i` il codice accede al descrittore di una specificia interfaccia (linea 4). Si noti che tale parametro era stato passato alla

```
1   // file: io.cpp
2   extern "C" estern(natl i)
3   {
4       des_io *d = &array_des_io[i];
5
6       for (;;) {
7           ...
8           wfi();
9       }
10  }
```

Figura 3: Schema generale di un processo esterno.

```
1   # file: io.s
2   .global wfi
3   wfi:
4         int $TIPO_WFI
5         ret
```

Figura 4: Funzione di interfaccia per la primitiva `wfi()`.

`activate_pe()` al momento della creazione del processo, in modo del tutto analogo a quanto avviene con la `activate_p()`. Dopo la loro creazione i processi esterni non terminano più e, dopo aver risposto ad una richiesta di interruzione, si limitano a sospendersi in attesa della prossima. Il loro corpo è composto dunque da un ciclo infinito (line 6–9) che, dopo ogni iterazione, termina con una invocazione della primitiva di sistema `wfi()` (*wait for interrupt*). Anche questa primitiva è riservata al modulo I/O.

La Figura 4 mostra il codice della funzione di interfaccia `wfi()`, usata dal modulo I/O per invocare la primitiva di sistema vera e propria, mostrata in

```
1   # file: sistema.s
2   a_wfi:
3       call salva_stato
4       call apic_send_EOI
5       call schedulatore
6       call carica_stato
7       iretq
```

Figura 5: La primitiva di sistema `wfi()`.

Figura 5. La primitiva salva lo stato del processo esterno (linea 3), invia l'EOI al controllore APIC (linea 4) e mette in esecuzione un altro processo (linee 5–7), di fatto sospendendo il processo esterno, che a questo punto potrà andare nuovamente in esecuzione solo quando il corrispondente handler verrà nuovamente invocato, in riposta ad una nuova richiesta di interruzione da parte della stessa interfaccia.

Si noti che possiamo affermare con certezza che, dopo la prima volta che il processo esterno è andato in esecuzione, la coppia "**call** `carica_stato`**; iretq**" al termine dell'handler associato (linee 8–9 di Figura 2) caricherà lo stato salvato dalla `a_wfi`. Infatti, se l'handler è in esecuzione vuol dire che l'interfaccia ha inviato una richiesta che l'APIC ha fatto passare, e dunque l'APIC deve aver ricevuto l'EOI per la richiesta precedente, e dunque l'ultima cosa che il processo esterno aveva fatto in precedenza era proprio chiamare la `wfi()`.

Non possiamo invece sapere quale processo verrà messo in esecuzione al termine della `a_wfi`. Questo perché il processo esterno gira a interruzioni abilitate e dunque, mentre è stato in esecuzione, vari processi potrebbero essere finiti in coda pronti (per esempio, processi sospesi su una `delay()`, o processi che attendevano la conclusione di operazioni di I/O su altre periferiche a maggiore priorità, etc.).

1 Esempio di primitiva di lettura

Torniamo a considerare una generica operazione di lettura, con interfaccia utente

```
extern "C" void read_n(natl id, natb *buf, natl quanti);
```

L'operazione sarà svolta in parte dalla primitiva e in parte da un processo esterno messo in esecuzione da un handler ad ogni richiesta di interruzione da parte dell'interfaccia.

Consideriamo ora un processo P_1 che invoca la primitiva `read_n()`. Questa, come per tutte le primitive, è in realtà solo una piccola funzione scritta in Assembler nel file `utente.s`. La funzione invoca la primitiva vera e propria tramite una istruzione **int**, che permette l'innalzamento del livello di privilegio:

```
1  # file: utente.s
2  .global read_n
3  read_n:
4        int $IO_TIPO_RN
5        ret
```

All'offset `IO_TIPO_RN` della tabella IDT dovrà essere installato un gate con GP=1 e IND=`a_read_n`. Questa sarà una funzione scritta in assembler nel file `io.s`:

```
1  # file: io.s
2  .extern c_read_n
3  a_read_n:
```

```
4         cavallo_di_troia %rsi
5         cavallo_di_troia2 %rsi %rdx
6         call c_read_n
7         iretq
```

Ci sono alcune cose da notare:

- la funzione `a_read_n` *non* chiama `salva_stato` e `carica_stato`, come l'analoga nel caso del driver; in questo caso non avrebbe proprio potuto, in quanto si tratta di funzioni definite nel modulo sistema, che non è collegato con il modulo I/O;
- anche qui è necessario controllare il problema del Cavallo di Troia.

Passiamo ora alla parte C++ della primitiva. Anche in questo caso la primitiva ha bisogno di ricordare alcune informazioni relative alla periferica, quindi prevediamo un descrittore di operazioni di I/O, identico a quello visto precedentemente:

```
1   // file: io.cpp
2   struct des_io {
3     natw iRBR, iCTL;
4     natb *buf;
5     natl quanti;
6     natl mutex;
7     natl sync;
8   };
```

Prevederemo come al solito un array di tali descrittori e useremo `id` come indice al suo interno. In questo caso, però, l'array sarà definito in `io.cpp`.

Anche la `c_read_n` è identica a quella vista precedentemente, che riportiamo qui per comodità:

```
1   // file: io.cpp
2   extern "C"
3   void c_read_n(natl id, natb *buf, natl quanti)
4   {
5     des_io *d = &array_des_io[id];
6
7     sem_wait(d->mutex);
8     d->buf = buf;
9     d->quanti = quanti;
10    outputb(1, d->iCTL);
11    sem_wait(d->sync);
12    sem_signal(d->mutex);
13  }
```

C'è però una differenza rispetto al caso precedente, anche se non si vede: questa primitiva è ora definita nel modulo I/O (file `io.cpp`) e gira ad interruzioni

abilitate. Non ci sono comunque problemi di interferenza con altri processi che potrebbero interromperla, grazie alla mutua esclusione garantita dal semaforo `d->mutex`. Si noti come la mutua esclusione è rilasciata dopo aver atteso, sul semaforo `d->sync`, che il trasferimento fosse interamente completato. Chiunque voglia utilizzare la periferica mentre è già un corso un altro trasferimento dovrà dunque mettersi in fila alla riga 7, aspettando che il trasferimento sia finito.

Vediamo ora il codice del processo esterno:

```
1   // file: io.cpp
2   extern "C" void estern(natl id)
3   {
4       des_io *d = &array_des_io[id];
5       char c;
6
7       for (;;) {
8           d->quanti--;
9           if (d->quanti == 0)
10              outputb(0, d->iCTL);
11          inputb(d->iRBR, c);
12          *d->buf = c;
13          d->buf++;
14          if (d->quanti == 0)
15              sem_signal(d->sync);
16          wfi();
17      }
18  }
```

Lo scopo principale del processo esterno, come quello del driver, è di leggere il nuovo byte dall'interfaccia e copiarlo nel buffer dell'utente. Il codice del processo esterno è dunque molto simile a quello del driver, ma ci sono delle differenze dovute al fatto che ora ci troviamo in un vero processo. In particolare, il test su `d->quanti == 0` deve essere ripetuto due volte:

1. alla riga 9, in quanto dobbiamo eventualmente disabilitare le interruzioni (riga 10) *prima* di leggere il byte dall'interfaccia (riga 11);

2. alla riga 14, in quanto dobbiamo risvegliare il processo che aveva richiesto il trasferimento (riga 15) *dopo* aver effettivamente trasferito l'ultimo byte (righe 11–12).

Per il punto 1 vale esattamente lo stesso discorso già fatto nel caso del driver: se lasciassimo le interruzioni abilitate e leggessimo l'ultimo byte, l'interfaccia potrebbe generare una nuova richesta, portando al trasferimento di un ulteriore byte che non era stato richiesto.

Il punto 2 è dovuto al fatto che, a differenza del driver, ora stiamo invocando la vera `sem_signal()` (linea 15) e questa potrebbe mettere in esecuzione il processo risvegliato, se questo ha pririt̀a maggiore del processo esterno. Il

processo risvegliato (che era quello che aveva richesto il trasferimento) potrebbe così cominciare ad usare i dati, prima che l'ultimo byte sia stato effettivamente letto.

La Figura 6 mostra un esempio di evoluzione del sistema supponendo che un processo P_1 invochi una `read_n()` chiedendo di trasferire 2 byte dall'iterfaccia i. Si suppone che sia pronto anche un processo P_2, a priorità più bassa, e che non ci siano altre richieste di interruzioni oltre quelle dell'interfaccia i durante tutto il trasferimento.

2 Interazioni con gli altri meccanismi

Vediamo ora come le operazioni di I/O interagiscono con il meccanismo della memoria virtuale e del Bus Mastering. Per quanto riguarda il Bus Mastering, la cosa a cui prestare attenzione è che i trasferimenti eseguiti dalle periferiche in grado di operare in questa modalità non attraversano la MMU, e dunque sono eseguiti ad indirizzi *fisici*. Per quanto riguarda la memoria virtuale, la cosa a cui prestare attenzione è che i byte sono traferiti mentre in esecuzione c'è, in generale, un processo *diverso* da quello che aveva richiesto il trasferimento.

Pensiamo ora al fatto che il processo utente che ha iniziato il trasferimento ha passato alla `read_n()` l'indirizzo di un suo buffer. Il buffer si trova ovviamente nella memoria *virtuale* del processo.

Nel caso di primitiva di sistema con driver, teniamo presente che il driver userà le tabelle di traduzione del processo che ha interrotto. Se vogliamo che il driver possa accedere al buffer anche tramite queste tabelle, il buffer dovrà trovarsi nella parte utente/condivisa, in modo che la traduzione sia la stessa per tutti i processi. Nel nostro caso questo comporta che deve trovarsi nella sezione `.data` (o nello heap). Se il processo utente è scritto in C++, questo implica he il buffer dovrà essere dichiarato a livello globale (o allocato nello heap). Inoltre, abbiamo detto che consideriamo un errore se codice del modulo sistema genera un page fault; per questo motivo il buffer dovrà anche non essere rimpiazzabile. Nel nostro caso, visto che abbiamo reso residente tutta la parte utente/condivisa, questo non comporta ulteriori limitazioni.

Consideriamo ora il caso di primitiva di I/O con handler e processo esterno. Il buffer dovrà essere accessibile sia al processo utente, sia al processo esterno. Nel nostro caso, questo comporta, di nuovo, che il buffer deve trovarsi nella sezione `.data` (o nello heap). Il processo esterno, in quanto processo, può generare page fault senza problemi. Non è dunque necessario che il buffer sia residente. Se però si vogliono gestire i trasferimenti dallo swap tramite interruzioni, si deve garantire che tali interruzioni abbiano la priorità massima, altrimenti l'APIC potrebbe non farle passare mentre è in esecuzione un processo esterno.

Pensiamo infine al caso del Bus Mastering. L'indirizzo del buffer passato alla primitiva non può essere direttamente utilizzato dalla perifica, ma va prima tradotto in fisico. Dal momento che le tabelle di traduzione sono gestite dal modulo sistema e non vogliamo che il modulo I/O vi acceda direttamente, forniamo al modulo I/O la seguente primitiva di sistema:

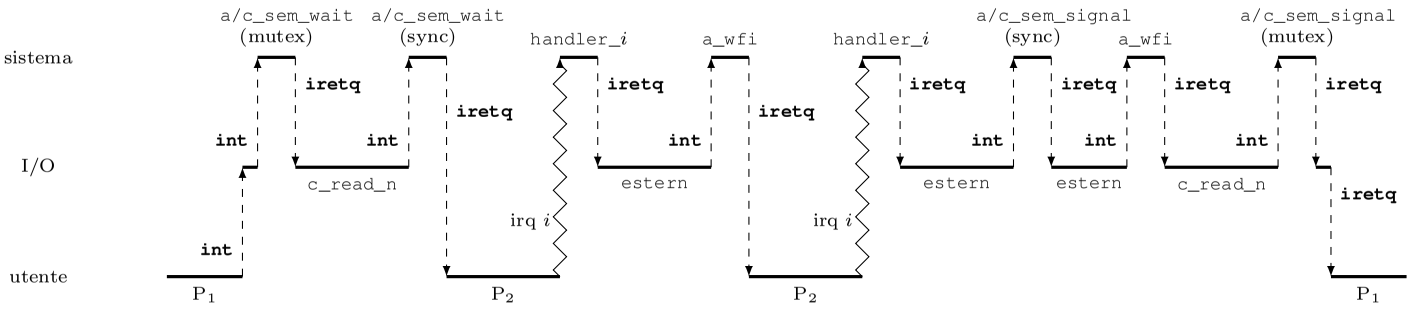

Figura 6: Esempio di esecuzione di `read_n()`. Il processo P_1 richiede il trasferimento di due byte.

```
addr trasforma(addr ind_virt);
```

Questa primitiva, dato un indirizzo virtuale `ind_virt`, restituisce il corrispondente indirizzo fisico (0 se la pagina o una qualunque delle tabelle non è presente). Non è necessario che il buffer si trovi nella parte utente/condivisa, proprio perché la periferica accederà direttamente alla memoria fisica, senza bisogno di traduzioni di indirizzi. Per lo stesso motivo, però, è fondamentale che il buffer non sia rimpiazzato dalla memoria per tutta la durata del trasferimento (la periferica non ha modo di sapere che il frame in cui sta scrivendo, o da cui sta leggendo, ora contiene qualcos'altro). Se il buffer attraversa più pagine, infine, non sarà in generale possibile completare tutto il traferimento in un'unica operazione, in quanto i frame che contengono il buffer potrebbero non essere contigui in memoria.