

Modulo I/O

G. Lettieri

10 Maggio 2024

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 (come tutti gli altri processi) 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

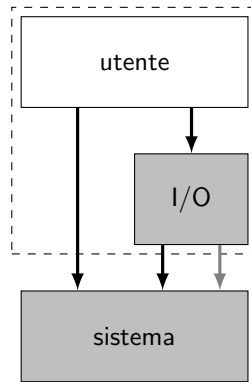


Figura 1: Relazioni tra i moduli

`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 e mappato nello spazio di indirizzamento di ogni processo, nella parte che abbiamo chiamato *IO/condivisa*. Il modulo `io` conterrà tutto ciò che è legato alle primitive di I/O, incluse le primitive di I/O invocabili dall'utente e le loro strutture dati.

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 a_p+$i*8, %rax
6       movq %rax, esecuzione
7       call carica_stato
8       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 DPL 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 `activate_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. La `activate_pe()`, dopo aver creato il processo, inserirà il corrispondente `des_proc` 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 `des_proc` da questa tabella. Quindi, se i è uno dei piedini dell'APIC, dovrà esistere un `handler_i` che metta in esecuzione il `des_proc` preso da `a_p[i]`. Questi handler non sono inizialmente associati a nessuna entrata della IDT. Il motivo è che l'entrata della IDT, ovvero il tipo dell'interruzione, determina anche la priorità che l'APIC assegna alla richiesta di interruzione. Vogliamo fare in modo che questa priorità sia consistente con la precedenza del corrispondente processo esterno, come assegnata dalla `activate_pe()`. La soluzione adottata prevede che tale precedenza debba avere la forma `MIN_EXT_PRIO + prio`, dove `MIN_EXT_PRIO` è una costante definita dal sistema (in `costanti.h`) e `prio` è un numero naturale minore di 256. La `activate_pe()` programmerà l'APIC in modo che il piedino i invii il tipo `prio` e all'entrata `prio` della IDT installerà un gate che punti a `handler_i`.

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-

```

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()`.

6 esegue l'equivalente del codice `esecuzione=a_p[i]`. La successiva coppia `"call carica_stato; iretq"` (righe 7 e 8) 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, ciascuna 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 specifica interfaccia (linea 4). Si noti che tale parametro era stato passato alla `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 (linee 6-9) che, dopo ogni iterazione, termina con una invocazione della primitiva di sistema `wfi()` (*wait for interrupt*), come si vede alla linea 8. 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 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 risposta ad una nuova richiesta di interruzione da parte della stessa interfaccia.

```

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()`.

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 6–7 di Figura 2) caricherà lo stato salvato alla linea 3 della `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, char* buf, natq 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'entrata `IO_TIPO_RN` della tabella IDT dovrà essere installato un gate con che punti a `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      call c_read_n
5      iretq
```

Notiamo che la funzione `a_read_n` *non* chiama le funzioni `salva_stato` e `carica_stato`, esattamente come l'analoga nel caso del driver. Il motivo è lo stesso: la primitiva `read_n()` non è atomica ed è eseguita dal processo che la invoca. In questo caso, però, non c'è modo di chiamare `salva_stato` o `carica_stato` per sbaglio, in quanto si tratta di funzioni definite nel modulo sistema, che non è collegato con il modulo I/O.

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      char* 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 (il suo gate è di tipo *trap*). Dobbiamo quindi preoccuparci di proteggere le risorse usate dalla primitiva rispetto ai possibili accessi da parte di altri processi che potrebbero interromperla e cercare di accedere alle stesse risorse. In questo caso le risorse sono il descrittore `des_io` e la periferica stessa (i suoi registri). In particolare, la primitiva potrebbe essere interrotta:

1. da altri processi che tentano di invocare `read_n()` sulla stessa interfaccia;
2. dal processo esterno associato all'interfaccia.

(Interruzioni da altri processi diversi da questi non potrebbero accedere alle risorse utilizzate dalla `read_n()` e quindi non ci interessano). Il primo problema è risolto tramite la 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 sia interamente completato. Chiunque volesse 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.

Il secondo problema è risolto sfruttando il fatto che il processo esterno potrà andare in esecuzione solo dopo che abbiamo abilitato l'interfaccia a generare richieste di interruzione (riga 10), quindi è sufficiente completare tutti gli accessi al `des_io` prima di quel momento (linee 8-9).

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
6     for (;;) {
7         d->quanti--;
8         if (d->quanti == 0)
9             outputb(0, d->iCTL);
10        char c = inputb(d->iRBR);
11        *d->buf = c;
12        d->buf++;
13        if (d->quanti == 0)
14            sem_signal(d->sync);
15        wfi();
16    }
17 }

```

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 8, in quanto dobbiamo eventualmente disabilitare le interruzioni (riga 9) *prima* di leggere il byte dall'interfaccia (riga 10);
2. alla riga 13, in quanto dobbiamo risvegliare il processo che aveva richiesto il trasferimento (riga 14) *dopo* aver effettivamente trasferito l'ultimo byte (righe 10–11).

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 richiesta, portando al trasferimento di un ulteriore byte che non era stato richiesto.

Il punto 2 è dovuto al fatto che, a differenza del driver, il processo esterno non è atomico. In particolare, la `sem_signal()` (linea 14) potrebbe mettere in esecuzione il processo risvegliato, se questo ha priorità maggiore del processo esterno. Il processo risvegliato (che era quello che aveva richiesto il trasferimento) potrebbe così cominciare ad usare i dati, prima che l'ultimo byte sia stato effettivamente trasferito.

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'interfaccia 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.

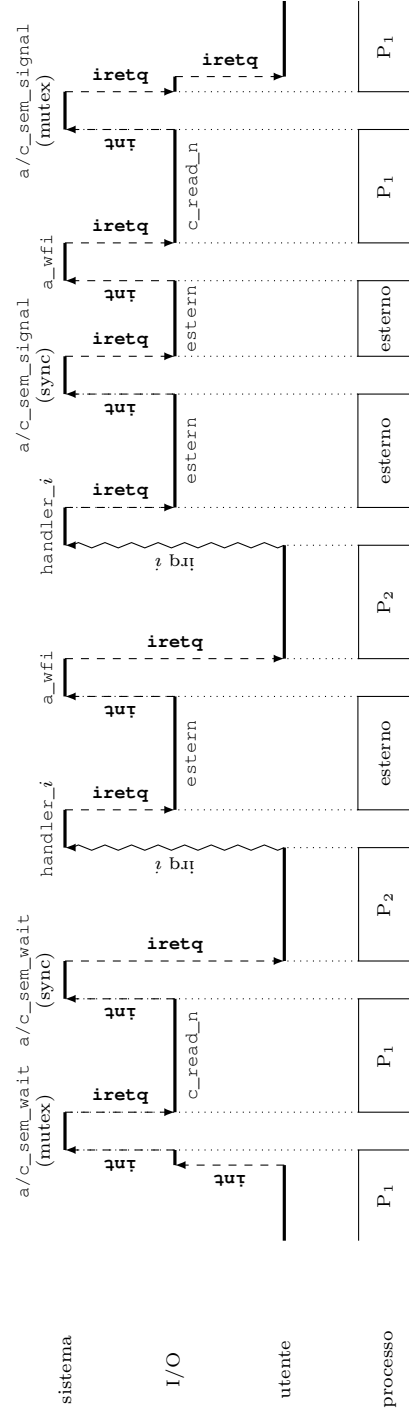


Figura 6: Esempio di esecuzione temporale in seguito all'invocazione di `read_n()` da parte di un processo `P1` (trasferimento di due byte)