I/O nel nucleo

G. Lettieri

10 Maggio 2022

Come abbiamo visto nell'esempio che motivava l'introduzione della protezione, le operazioni di I/O offrono un'ottima opportunità per sfruttare appieno l'ambiente multiprogrammato. Ricordiamo brevemente gli aspetti essenziali dell'esempio. Tipicamente, un processo che inizia una operazione di I/O non può proseguire finché l'operazione non è terminata, e d'altro canto l'operazione stessa è normalmente lenta e non ha bisogno della CPU per essere portata avanti, o ne ha bisogno in minima parte. L'idea è quindi di *bloccare* i processi che iniziano una operazione di I/O e sbloccarli (riportarli in coda pronti) quando l'operazione è terminata. In questo modo altri processi potranno andare in esecuzione mentre è in corso l'operazione di I/O e la CPU sarà sfruttata più efficientemente. Si noti che il sistema deve sapere che l'operazione è completata mentre è in esecuzione un processo che, in generale, è completamente scorrelato da essa. Il completamento dell'operazione dovrà essere segnalato da una interruzione, in modo che il sistema possa riacquistare il controllo della CPU e svolgere le operazioni necessarie, tra cui sbloccare il processo che aveva richiesto l'operazione.

Quanto appena descritto è un esempio di *sincronizzazione*: vogliamo che il processo che ha iniziato l'I/O possa proseguire solo dopo che l'operazione di I/O è stata completata.

In ambiente multiprocesso il sistema deve anche preoccuparsi di coordinare l'accesso alle periferiche. Tipicamente, mentre una periferica è impegnata in una operazione di I/O, non può essere usata per altre operazioni. Se un processo vuole usare una periferica mentre questa è occupata, deve aspettare che la precedente operazione termini e la periferica ritorni libera. A differenza del precedente, questo non è un problema di sincronizzazione, ma di *mutua esclusione*: se due processi, P_1 e P_2, vogliono entrambi usare la stessa periferica, è per noi indifferente se la usa prima P_1 e poi P_2 o viceversa: l'unica cosa che ci interessa è che non la usino contemporaneamente. Si noti infine che la mutua esclusione possiamo garantirla separatemene per ogni periferica: se P_1 e P_2 devono usare due periferiche distinte non hanno bisogno di coordinarsi.

Ricordiamo ora che i processi di cui stiamo parlando sono processi *utente* e, come al solito, sono da considerarsi non fidati. Non possiamo dunque aspettarci che gli utenti si coordinino tra di loro per usare le periferiche uno alla volta, o che sospendano volontariamente i propri processi per far andare avanti quelli

degli altri. Per imporre la mutua esclusione e la sincronizzazione di cui sopra procediamo come al solito:

- impediamo agli utenti di parlare direttamente con le periferiche e
- forniamo delle primitive per svolgere le operazioni di I/O sotto il controllo del sistema.

Per realizzare il primo punto facciamo in modo che il campo IOPL del registro **rflags** dei processi utente specifichi il valore "sistema". In questo modo, mentre il processore si trova a livello utente, genera una eccezione ogni volta che si prova ad eseguire una istruzione di `in` o `out` [1] Notare che, indipendentemente da questo problema, siamo sostanzialmente obbligati a settare il campo IOPL a "sistema", in quanto questo è anche il modo in cui vietiamo agli utenti l'utilizzo delle istruzioni `sti` e `cli`, che abilitano e disabilitano le interruzioni esterne mascherabili. Per vietare l'accesso alle periferiche che hanno i registri mappati in memoria ricorriamo invece alla MMU, in uno dei due modi possibili: o non inserendo traduzioni che portino ai registri delle periferiche, o proteggendo le traduzioni con i bit "S/U" nei descrittori.

Occupiamoci ora della struttura generale delle primitive di I/O che il sistema deve fornire. Una tipica primitiva di lettura avrà una interfaccia simile a questa:

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

La primitiva riceve un parametro `id`, che serve ad identificare la periferica da cui il processo vuole leggere, e un indirizzo `buf` che punta ad un buffer in cui l'utente vuole ricevere i dati. Faremo l'ipotesi che i dati siano sempre una sequenza di byte. Il parametro `quanti` specifica il numero di byte che l'utente vuole leggere. È l'utente che deve preoccuparsi di dichiarare un buffer grande a sufficienza per contenere i byte richiesti. Vedremo che il sistema può imporre altre limitazioni su questo buffer.

Analogamente, la tipica primitiva di scrittura avrà questa interfaccia verso gli utenti:

```
extern "C" void write_n(natl id, const char* buf,
      natq quanti);
```

I parametri hanno lo stesso significato del caso precedente, ma ora il buffer puntato da `buf` deve contenere i dati che l'utente vuole scrivere (la primitiva si limita a leggerli, da cui il **const**).

1 Realizzazione con primitiva e driver

Concentriamoci su una generica operazione di lettura, con interfaccia utente

[1] Per vietare `in` e `out` sono necessari anche altri accorgimenti, che tralasciamo per semplicità; maggiori dettagli si trovano nel codice.

```
extern "C" void read_n(natl id, char* buf, natq quanti)
```

Assumiamo che nel sistema siano installate diverse periferiche simili, identificate dunque dal parametro `id`. Assumiamo inoltre che queste periferiche siano in grado di trasferire un byte alla volta, inviando una richiesta di interruzione ogni volta che è disponibile un nuovo byte. L'interfaccia di ogni periferica avrà un registro di controllo per abilitare e disabilitare le interruzioni (CTL) e un registro di ingresso da cui leggere il nuovo byte (RBR). La lettura da RBR funziona da risposta alla richiesta di interruzione: l'interfaccia non genera una nuova richiesta di interruzione fino a quando non ha avuto una risposta a quella precedente.

L'operazione sarà svolta in parte dalla primitiva e in parte da un *driver*, che andrà in esecuzione ad ogni richiesta di interruzione da parte dell'interfaccia:

- la primitiva ha lo scopo di avviare l'operazione di I/O e bloccare il processo, garantendo anche la mutua esclusione;
- il driver ha il compito di trasferire effettivamente i byte e sbloccare il processo quando l'operazione si è conclusa.

Per gestire la mutua esclusione e la sincronizzazione vogliamo usare i semafori, ma questo comporta che la primitiva non può essere atomica, e dunque non deve salvare e caricare lo stato del processo all'entrata e all'uscita dal sistema. In questo caso ha senso immaginare che sia il processo stesso ad eseguire la primitiva, sospendendosi e risvegliandosi al suo interno in corrispondenza delle invocazioni delle primitive semaforiche.

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  .global read_n
2  read_n:
3        int $IO_TIPO_RN
4        ret
```

Nell'entrata corrispondente al tipo `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 `sistema.s`:

```
1  .extern c_read_n
2  a_read_n:
3        call c_read_n
4        iretq
```

Si noti che, come dicevamo, la `a_read_n` non salva lo stato, lasciando che la primitiva venga eseguita nel contesto del processo P_1 che l'ha invocata.

Passiamo ora alla parte C++ della primitiva. Dato l'identificatore `id`, la primitiva ha bisogno di ottenere alcune informazioni sulla corrispondente periferica (l'indirizzo dei suoi registri, ma non solo). Per memorizzare tutte queste informazioni prevediamo un descrittore di operazione di I/O definito come segue:

```
1  struct des_io {
2    ioaddr iRBR, iCTL;
3    char* buf;
4    natq quanti;
5    natl mutex;
6    natl sync;
7  };
```

È sufficiente avere un array di tali descrittori e usare `id` come indice al suo interno. Ogni descrittore contiene tre tipi di informazioni: gli indirizzi dei registri (riga 2), le informazioni (ricevute dall'utente) su dove i byte vanno trasferiti (righe 3–4) e due identificatori di semafori (righe 5–6). Per realizzare la sincronizzazione e la mutua esclusione, come abbiamo anticipato, la `c_read_n` utilizzerà i semafori, che servono proprio a questo. Il semaforo `mutex` è inizializzato a 1 all'avvio del sistema e serve a garantire la mutua esclusione tra i processi che vogliono usare la periferica `id`. Il semaforo `sync` è inizializzato a 0 all'avvio del sistema e serve a realizzare la sincronizzazione tra il processo che ha richiesto l'operazione e la conclusione dell'operazione stessa.

La `c_read_n` è strutturata nel modo seguente:

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

Alla riga 4 ottiene un puntatore al descrittore della interfaccia, per comodità di scrittura. Le righe 6 e 11 garantiscono la mutua esclusione: solo un processo alla volta può eseguire le righe 7–10

Le righe 7 e 8 trasferiscono le informazioni sul buffer dell'utente all'interno del descrittore. Da qui le leggerà il driver quando andrà in esecuzione.

La riga 9 abilita le interruzioni (supponiamo che sia sufficiente scrivere 1 nel registro CTL). Da questo momento l'interfaccia può inviare richieste di interruzione ogni volta che ha un nuovo byte da trasferire. Si noti che per il momento le interruzioni esterne sono mascherate, in quanto ci troviamo nel modulo sistema.

La riga 10 blocca P_1 sul semaforo di sincronizzazione. A questo punto il sistema può passare ad eseguire altri processi. Mentre sono in esecuzione questi altri processi le interruzioni sono abilitate. Si noti che P_1 è bloccato all'interno della `sem_wait` alla riga 10 e dunque ancora dentro la zona di mutua esclusione. Fino a quando P_1 non verrà sbloccato e non eseguirà la `sem_signal(d->mutex)` alla riga 11 l'interfaccia non potrà essere usata da altri processi, come volevamo. Se un altro processo (andato in esecuzione in seguito al blocco di P_1) invocherà la `read_n` sulla stessa interfaccia, arriverà alla riga 6 e si bloccherà.

Passiamo ora ad esaminare il driver. Il driver andrà in esecuzione per effetto di una richiesta di interruzione da parte dell'interfaccia (richiesta che arriverà alla CPU tramite l'APIC). Il driver gira a livello sistema e, per poter tornare a livello utente, deve terminare anch'esso con una **`iretq`**. Anche il driver, dunque, avrà una parte scritta in assembler:

```
1  .extern c_driver
2  a_driver_i:
3        call salva_stato
4        movq $i, %rdi
5        call c_driver
6        call apic_send_EOI
7        call carica_stato
8        iretq
```

Per fare in modo che `a_driver_`i vada in esecuzione ogni volta che l'interfaccia genera una interruzione, occorre conoscere il tipo dell'interruzione generata e preparare il corrispondente gate della IDT in modo che punti a `a_driver_`i.

Chiamiamo P_2 il processo in esecuzione all'arrivo dell'interruzione.

Il driver salva e ripristina lo stato di P_2 (linee 3 e 7) perché può dover cambiare il processo in esecuzione. Infatti, quanto l'ultimo byte è stato trasferito, il driver deve risvegliare P_1 (che ora è bloccato nella `sem_signal(d->sync)` dentro `c_read_n`). Se P_1 ha una priorità maggiore di P_2, è P_1 che deve andare in esecuzione, mentre P_2 deve tornare in coda pronti. La `carica_stato` alla riga 7, quindi, carica lo stato di P_2 o di P_1, a seconda dei casi.

Alle righe 4–5 si chiama il driver vero e proprio, `c_driver`, scritto in C++. Dal momento che stiamo assumendo di trattare interfacce simili, `c_driver` può essere una funzione generica e ricevere un parametro che gli indichi l'interfaccia da gestire (linea 4). Si noti che il parametro che `a_driver_`i passa a `c_driver` è una costante. Questo perché ogni interfaccia genera una richiesta di interruzione di tipo diverso, quindi è sufficiente associare ad ogni tipo di interruzione una diversa copia di `a_driver_`i, ciascuna con una costante diversa.

Alla riga 6 inviamo l'End Of Interrupt al controllore APIC, in modo che lasci passare le interruzioni a priorità minore o uguale di quella appena gestita dal driver.

Si noti che il driver usa le risorse di P_2. In particolare usa la sua pila sistema. Usa anche la memoria virtuale di P_2, dal momento che non stiamo cambiando

il valore di **cr3**.

Vediamo ora il codice di `c_driver`:

```
1   extern "C" void c_driver(natl id)
2   {
3     des_io *d = &array_des_io[id];
4
5     d->quanti--;
6     if (d->quanti == 0) {
7       outputb(0, d->iCTL)
8       des_sem *s = &array_dess[sem];
9       s->counter++;
10
11      if (s->counter <= 0) {
12        des_proc* lavoro = rimozione_lista(s->pointer);
13        inspronti();    // preemption
14        inserimento_lista(pronti, lavoro);
15        schedulatore(); // preemption
16      }
17    }
18    char c = inputb(d->iRBR);
19    *d->buf = c;
20    d->buf++;
21  }
```

Alla riga 3 il driver ottiene un puntatore al descrittore della interfaccia, per comodità di scrittura.

Lo scopo principale del driver è leggere il nuovo byte dall'interfaccia e copiarlo nel buffer dell'utente, cosa che viene fatta alle righe 18–19. Alla riga 20 il puntatore al buffer viene incrementato, in modo che il prossimo byte venga copiato nella locazione successiva.

Alla riga 5 il driver decrementa `d->quanti`, che così contiene sempre il numero di byte ancora da leggere. Se `d->quanti` è arrivato a zero il byte letto alla riga 18 è l'ultimo byte richiesto dall'utente. Si deve quindi disabilitare l'interfaccia a generare interruzioni (riga 7) e risvegliare il processo che aveva iniziato l'operazione (righe 8– 16).

Ci sono alcune cose da notare:

1. la disabilitazione delle interruzioni (riga 7) è eseguita *prima* della lettura del byte (riga 18);

2. invece di chiamare `sem_signal()` ripetiamo la parte centrale del suo codice (riga 8–16);

3. la scrittura in `d->buf` (riga 19) è eseguita mentre è attiva la memoria virtuale di P_2, anche se il buffer era stato allocato da P_1.

Il punto 1 è una conseguenza del fatto che la lettura del byte fa da risposta alla richiesta di interruzione da parte dell'interfaccia, che a quel punto può generarne un'altra se ha un nuovo byte disponibile. Quindi, se leggiamo l'ultimo byte mentre le interruzioni sono abilitate, è possibile che l'interfaccia generi una nuova interruzione, rimandando in esecuzione il driver, anche se nessun processo ha iniziato una operazione di lettura. Il driver leggerebbe questo byte e lo copierebbe dove punta `d->buf`, andando a sovrascrivere parti casuali della memoria.

Il punto 2 è una conseguenza del fatto che `sem_signal()` salva e ripristina lo stato, ma il driver non è un processo e non ha un suo descrittore di processo. In particolare, se `c_driver` chiamasse `sem_signal()` salverebbe lo stato del driver nel descrittore processo attivo, che è P_2. Per di più sovrascriverebbe lo stato salvato alla riga 3 di `a_driver_`i (avremmo violato la regola di non avere due `salva_stato` senza una `carica_stato` in mezzo).

Dal momento che il driver fa parte del modulo sistema, potremmo pensare di chiamare direttamente `c_sem_signal()`, evitando il salvataggio e caricamento dello stato. Purtroppo, però, il controllo sulla validità del semaforo (`sem_valido()`) fallirebbe, in quanto `liv_chiamante()` assume che `c_sem_signal()` sia stata invocata per effetto di una istruzione **`int`**. Nel nostro caso, invece, `liv_chiamante()` accederebbe alla pila sistema di P_2 e scambierebbe il livello di P_2, che molto probabilmente è utente, per il livello del chiamante della `c_sem_signal()`; dal momento che il semaforo `ce->sync` era stato allocato da livello sistema, segnalerebbe un errore. Non ci resta dunque che ricopiare il codice della `c_sem_signal()` escludendo i controlli dei parametri (o definire una ulteriore funzione che contenta solo tale codice). Si noti che il driver, in questo modo, manipola le code dei processi, e dunque deve essere eseguito con le interruzioni disabilitate. In altre parole, il driver deve essere atomico. Questo comporta che anche le richieste di interruzione a precedenza maggiore dovranno aspettare che il driver termini, prima di poter essere gestite.

Il punto 3 è un problema se P_1 ha allocato il buffer nella sua sezione utente/privata, dal momento che gli indirizzi virtuali delle sezioni private hanno significati diversi per ogni processo. In questo caso `c_driver` scriverebbe nella sezione utente/privata di P_2, non di P_1, causando un doppio problema: P_1 non riceverebbe i suoi dati e P_2 si vedrebbe sovrascrivere parti casuali della sua memoria. Dobbiamo quindi richiedere che i buffer per l'I/O vengano sempre allocati nella parte utente/condivisa. Dal punto di vista del linguaggio C++, nel nostro caso, questo comporta che i buffer devono essere dichiarati globali o allocati nello heap, e mai dichiarati come variabili locali. Questo perché le variabili locali vengono allocate in pila e abbiamo deciso che le pile si trovano in parti private. Nei sistemi che bloccano i processi che causano page fault, il buffer deve anche essere residente (non rimpiazzabile), in quanto il driver, non essendo un processo, non può essere bloccato.

Esaminiamo infine come devono essere impostati i campi del gate che porta a `a_read_n` e di quello che porta a `a_driver_`i. Per il primo avremo:

- il campo DPL (Descriptor Privilege Level) che indica che il gate può essere utilizzato da livello utente;
- il campo L (Level) che indica che, dopo il salto, il processore si deve trovare a livello sistema;
- il campo I/T che può indifferentemente indicare "Interrupt" o "Trap", dal momento che la primitiva non manipola direttamente le code e i descrittori dei processi, ma noi assumeremo "Interrupt" per uniformità con il resto del codice del modulo sistema.

Per il gate della `a_driver_`*i* avremo:

- il campo DPL (Descriptor Privilege Level) che indica che il gate può essere utilizzato solo da livello sistema;
- il campo L (Level) che indica che, dopo il salto, il processore si deve trovare a livello sistema;
- il campo I/T che indica "Interrupt", per quanto detto prima.

L'impostazione del campo DPL deriva da questa constatazione: tutti i gate della IDT possono essere usati indifferentemente da tutti e tre i meccanismi di interruzione (interruzioni esterne, eccezioni e istruzione **`int`**). Impostando DPL a livello sistema impediamo all'utente di invocare il driver in modo spurio, tramite una **`int`**.

1.1 Controllo sui parametri

Alla riga 3 della `c_read_n()` abbiamo lasciato un commento in cui si dice che è necessario controllare i parametri. Questo è sempre vero per tutte le primitive, in quanto i parametri arrivano dal livello utente che, per definizione, non è fidato. In particolare, prima di usare `id` alla riga 4, dobbiamo assicurarci che `id` sia un valido identificatore di periferica. Se abbiamo raggruppato tutti i descrittori all'inizio dell'array è sufficiente ricordare l'indice dell'ultimo e confrontarlo con `id`.

Si noti che, invece, non c'è bisogno di controllare che `id` sia valido alla riga 3 di `c_driver_`*i*, in quanto in quel caso `id` è stato passato da `a_driver_`*i* che, facendo parte del modulo sistema, è fidata.

In `c_read_n()` è molto importante controllare i parametri `buf` e `quanti`, per evitare il cosiddetto problema del *Cavallo di Troia*. L'utente, invece di passare l'indirizzo di un buffer che ha correttamente allocato, potrebbe passare (usando dei cast o scrivendo direttamente in assembler) l'indirizzo di qualunque cosa, anche di parti della memoria che appartengono al sistema o ad altri processi. I controlli di protezione eseguiti dalla CPU e dalla MMU sono inefficaci in questo caso: il passaggio dell'indirizzo "cattivo" alla primitiva comporta solo la scrittura di un numero in un registro e la CPU non controlla alcunché, in quanto non può conoscere lo scopo di questa operazione. L'apparentemente innocuo Cavallo di Troia attraversa dunque le mura del sistema. Quando,

successivamente, il sistema tenta di usare l'indirizzo per leggere o scrivere (nel nostro caso ciò accade alla riga 19 di `c_driver_`i), ecco che il cavallo si apre, in quanto la primitiva gira a livello sistema e dunque anche la MMU non esegue alcun controllo. Se non prendiamo provvedimenti, l'utente riesce a costringere il sistema a scrivere su (o leggere da) una zona di memoria a cui lui, normalmente, non avrebbe accesso.

Dobbiamo anche controllare che il buffer che l'utente ci passa non contenga indirizzi che non sono mappati, o non sono normalizzati, in quanto questi causerebbero una eccezione nel driver alla riga 19, ma abbiamo detto che il driver deve essere atomico e, dunque, non deve generare eccezioni.

Il modulo sistema mette a disposizione seguente funzione che può essere usata in questi casi:

```
bool c_access(vaddr begin, natq dim,
              bool writeable, bool shared);
```

La funzione controlla che l'intervallo [`begin`, `begin` + `dim`) non attraversi il massimo indirizzo (non ci sia un *wrap around*) e non contenga indirizzi che fanno parte del "buco" (e dunque non sono normalizzati). Inoltre, percorre l'albero di traduzione e controlla che tutti gli indirizzi dell'intervallo siano mappati, accessibili da livello utente e, se `writeable` è **`true`**, anche scrivibili. Inoltre, se `shared` è **`true`**, controlla anche che tutto l'intervallo sia contenuto nella zona utente/condivisa.

La funzione `c_read_n()`, dunque, può eseguire il seguente codice

```
if (!c_access(begin, quanti, true)) {
  flog(LOG_WARN, "buf non valido");
  abort_p();
}
```

Si noti che passiamo **`true`** come parametro `writeable`, perché il driver dovrà *scrivere* nel buffer (in una `c_write_n()` avremmo passato **`false`**). Inoltre, dal momento che ci troviamo in una primitiva non atomica (che dunque non salva e carica lo stato), non è sufficiente chiamare `c_abort_p();` per abortire il processo, ma si deve chiamare `abort_p()`, che salva lo stato, chiama `c_abort_p()` e poi carica lo stato.