Azzolini Riccardo 2019-10-02

Gestione dell'I/O

1 Dispositivi I/O e controller

Si distingue tra:

- **controller** (detto anche **adapter**), la componente elettronica che comunica con la CPU (e le altre unità) tramite il bus di sistema;
- dispositivo vero e proprio, gestito dal controller.

Solitamente, un controller può gestire contemporaneamente più dispositivi uguali o simili.

Le interfacce tra controller e dispositivo sono solitamente standard (SATA, USB, ecc.).

2 Interfaccia tra CPU e controller

L'interfaccia tra CPU è controller è composta da:

- **registri**[1] **di controllo**, detti anche **porte di I/O**, che vengono usati:
  - dalla CPU, per inviare comandi al controller;
  - dal controller, per comunicare alla CPU i risultati dei comandi e lo stato del dispositivo;
- un **buffer**, usato per memorizzare i dati durante le operazioni di I/O.

Tale interfaccia viene usata dai **driver**, i programmi del SO che gestiscono i dispositivi.

[1] Questi registri sono situati nel controller: non sono registri della CPU.

3 Comunicazione tra CPU e porte di I/O

Ci sono due soluzioni per la comunicazione tra la CPU e le porte di I/O.

1. **Istruzioni macchina ad hoc**. Ad esempio, queste potrebbero essere `IN R, P`, che carica il valore della porta `P` nel registro `R` della CPU, e `OUT P, R`, che carica il valore del registro `R` della CPU nella porta `P`.
    - Queste istruzioni devono essere privilegiate, perché le interazioni con i dispositivi devono essere gestite solo dal SO.
    - Le parti dei driver che usano queste istruzioni devono essere scritte in assembly, perché i linguaggi ad alto livello non hanno istruzioni corrispondenti.
2. **Memory mapped I/O**: si assegna a ogni porta di I/O un indirizzo di memoria. Non servono quindi istruzioni ad hoc.
    - Questi indirizzi non sono visibili dai programmi, in modo che l'accesso ai dispositivi possa avvenire solo mediante il SO.
    - I driver possono essere scritti in linguaggi ad alto livello, come ad esempio il C.
    - Si complica la gestione della cache, perché il contenuto di queste locazioni di memoria può essere modificato dal dispositivo, e non solo dalla CPU.

4 Esecuzione dell'I/O

Si suppone che un programma P voglia trasferire `n` byte da un buffer di memoria `b` verso il buffer del controller di un dispositivo. Esistono tre possibili soluzioni per eseguire tale operazione di I/O.

4.1 Programmed I/O

Su richiesta di P, il SO (driver) esegue:

```
for (i = 0; i < n; i++) {
    while (device_status_reg != READY) {} // busy waiting
    buffer = b[i];
}
```

Viene quindi trasferito un byte alla volta. Per ogni byte:

1. finché la porta di I/O `device_status_reg` non indica che il dispositivo è pronto, il SO aspetta facendo **busy waiting** (*attesa attiva*);

2. quando il dispositivo è pronto, viene trasferito un singolo byte nel `buffer` del controller, poi il ciclo si ripete.

In questo modo, mentre il dispositivo non è pronto la CPU viene utilizzata solo per controllarne continuamente lo stato, mentre sarebbe più opportuno che essa continuasse a eseguire altri programmi.

4.2 Interrupt driven I/O

Il codice del driver eseguito su richiesta di P è:

```
while (device_status_reg != READY) {}
buffer = b[0];
i = 1;
scheduler();
```

1. *Solo inizialmente*, si effettua il busy waiting.
2. Viene scritto il primo byte nel buffer del controller.
3. Si imposta a 1 la variabile contatore/indice `i`.
4. Viene chiamato lo scheduler: la CPU viene sottratta al programma che ha richiesto l'operazione di I/O e assegnata a un altro, evitando il busy waiting.
5. Il resto del trasferimento viene eseguito da un interrupt handler:

```
if (i < n) {
    buffer = b[i];
    i++;
} else {
    unblock_user();
}
return_from_interrupt();
```

Quando il dispositivo segnala, mediante un interrupt, che ha finito di trasferire un byte, l'interrupt handler:

- se mancano ancora dei byte, trasferisce il prossimo e incrementa il contatore/indice;
- altrimenti, alla fine, rende nuovamente schedulabile il programma che aveva richiesto l'operazione di I/O.

Questa soluzione evita di usare la CPU solo per testare lo stato del dispositivo e aspettare, ma si hanno `n` interrupt (uno per ogni byte), la cui gestione introduce un overhead.

4.3 Direct Memory Access

L'architettura prevede un controller **Direct Memory Access** (**DMA**), capace di accedere direttamente alla memoria e di lavorare in parallelo con la CPU.

Su richiesta di P, il driver deve semplicemente:

1. richiedere il trasferimento, specificando il numero di byte e l'indirizzo di partenza negli appositi registri del controller DMA;
2. invocare lo scheduler per assegnare la CPU a un altro programma.

```
set_up_DMA_controller();
scheduler();
```

L'invio degli `n` byte viene eseguito in parallelo dal DMA, che interagisce con il controller del dispositivo al posto della CPU:

- imposta i registri di controllo del controller per inviare ciascun byte;
- riceve gli `n` interrupt che segnalano la fine di ciascun trasferimento.

Quando ha finito, il controller DMA invia un interrupt alla CPU. Allora, l'interrupt handler rende nuovamente schedulabile il programma:

```
unblock_user();
return_from_interrupt();
```

Siccome la CPU riceve solo un singolo interrupt, si ha una diminuzione dell'overhead rispetto all'interrupt driven I/O.

Questa soluzione è quella usata nei sistemi attuali.