1 Lezione del 28-04-25

1.1 DMA

Vediamo di introdurre il meccanismo del **DMA** (*Direct Memory Access*).

Finora l'accesso a periferiche era fatto a *controllo programma* o *controllo interruzione*. Chiaramente il controllo programma era più veloce, in quanto il programma provvedeva a trasferire ogni byte immediatamente appena l'interfaccia era pronta, mentre nel controllo interruzione bisognava prima eseguire tutti i passaggi necessari all'esecuzione degli *handler* per portare in esecuzione il processo esterno, ecc...

Introduciamo quindi il DMA per delegare tutta l'operazione di trasferimento dati dalle interfacce alle interfacce stesse. Vorremo quindi avere delle primitive:

- `read_n`, trasferimento da interfaccia a RAM di un buffer $[b, b+n)$;
- `write_n`, trasferimento da RAM a interfaccia di un buffer $[b, b+n)$.

Se è l'interfaccia stessa a compiere l'operazione di trasferimento, al processore servirà nuovamente un modo di controllare l'esito dell'operazione, che potra essere ancora a *controllo programma* o a *controllo interruzione* (che è il caso più comune).

Aggiungiamo quindi un **controllore DMA** al bus visto finora. Questo sarà di base dotato di due registri, uno per il buffer corrente (**B**) e uno per la sua dimensione (**N**), e potrà accedere in lettura e scrittura allo spazio di memoria al pari di come farebbe la CPU. Notiamo che non c'è bisogno che la RAM distingua fra operazioni CP o DMA, in quanto per questa sono equivalenti.

Dotiamo quindi il controllore DMA di due linee di handshake `HOLD` e `HOLDA`. Al momento dell'inizio di un'operazione DMA, il controllore alza `HOLD`, e la CPU risponde con `HOLDA`. Da qui in poi il controllore lavora sul bus, in regime di `cycle-stealing`, cioè "rubando" cicli di accesso alla CPU. Finita l'operazione, il controllore abbassa nuovamente `HOLD`, a cui la CPU risponde abbassando `HOLDA` e riprendendo a scrivere sul bus.

Storicamente questo meccanismo era utile per interfacce più veloci della CPU stessa (così era la RAM, ed erano ad esempio i controllori video, che si dividevano circa la metà del tempo RAM). Oggi, lo stesso discorso vale ad esempio per interfacce di rete, che raggiungono velocità del Gigabit al secondo.

1.1.1 DMA e cache

Vediamo di reintrodurre nel bus così modificato la cache. Innanzitutto, i contendenti al bus non saranno più CPU e controllore DMA, ma cache e controllore BUS, per cui le linee di `HOLD` e `HOLDA` saranno fra questi due.

Il vantaggio immediato che abbiamo è che probabilmente la maggior parte dei dati necessari alla CPU saranno in cache, per cui il regime di cycle-stealing non sarà troppo dannoso all'attività della CPU (ricordiamo che la cache accede al bus solamente quando si ha una *miss*).

Un problema potrebbe invece essere che la cache potrebbe perdersi gli aggiornamenti effettuati dal controllore DMA.

Write-through

Supponiamo ad esempio che la cache adotti una politica *write-through*, e che chiaramente il programmatore si impegni a non toccare il buffer per tutto il tempo del trasferimento.

- In questo caso, per operazioni di scrittura su dispositivo non ci saranno problemi, in quanto la politicha write-trough mantiene sia la RAM che la cache in uno stato identico e consistente, e il controllore DMA non modifica la RAM in operazioni di uscita.
- Viceversa, per operazioni di lettura da dispositivo avremo problemi, in quanto potremmo intaccare una zona di memoria che era replicata in cache, e la cache non avrà modo di conoscere tale aggiornamento.

  Per ovviare a tale problema abbiamo effettivamente due soluzioni:

  – La prima soluzione è *lato hardware*, e consiste nel dotare la cache della possibilità di fare *snooping* del bus, cioè capire a quali indirizzi il DMA sta accedendo, farne un lookup esattamente nella maniera in cui si farebbe lookup degli indirizzi richiesti dalla CPU, e procedere ricopiando i dati modificati (*snarfing*) o direttamente invalidando tale porzione di cache (soluzione adottata dall'architettura Intel x86);
  – La seconda soluzione è *lato software*, e consiste nel dotare il controllore di cache di un'apposita istruzione per l'invalidazione di cache. Utilizzeremo quindi questa istruzione per invalidare i buffer forniti per la lettura da dispositivo al DMA, idealmente alla fine dell'operazione di trasferimento.

    Nel frattempo chiaramente non vorremmo toccare nessuna delle cacheline impegnate dal buffer, che possono definire una regione anche maggiore in dimensioni del buffer stesso. L'invalidazione a termine operazione viene effettuata a fine operazione proprio per questo motivo, ma una soluzione alternativa potrebbe anche essere l'adottare buffer allineati ai 64 KiB delle cacheline.

Write-back

Fatta l'ipotesi di *write-trough*, cioè di scritture effettuate in differita dalla cache, che mantiene le modifiche temporaneamente nella sua memoria, avremo problemi sia in lettura che in scrittura.

- In scrittura su dispositivo avremo chiaramente che il buffer in RAM potrebbe non essere stato aggiornato con le modifiche in RAM al momento dell'inizio dell'operazione da parte del controllore DMA.

  – Qui la soluzione *lato hardware* è di definire un protocollo per cui il controllore DMA deve prima parlare con la cache, fornendo l'indirizzo a cui intende accedere (questa fase viene detta sempre di *snooping*). In questo caso la cache ha il tempo di controllare l'indirizzo e quindi capire se la cacheline corrispondente è *dirty*, e quindi in RAM ce n'è una versione obsoleta. A questo punto potrà agire di conseguenza, fornendo lei stessa i dati aggiornati o effettuando una write-back;
  – La soluzione *lato software* sarà invece di fornire un'istruzione di pulizia, che permetta di forzare il write-back delle cacheline coinvolte nel buffer prima di iniziare l'operazione di lettura in RAM da parte del controllore DMA.

- In lettura da dispositivo avremo invece che il controllore DMA potrebbe intaccare zone di memoria per cui la cache stava pianificando scritture in differita.

  - In questo caso il chipset PIIX3 emulato da QEMU, ad esempio, ottiene dalla cache la versione più aggiornata (se questa esiste in cache), effettua il merge internamente al controllore DMA, che provvede poi ad effettuare la scrittura in RAM; Per cacheline complete, abbiamo che il DMA può adottare anche il protocollo *write invalidate*, per cui la cache attraverso un'operazione di snooping può verificare che un intera cacheline è stata modificata ed invalidarla;
  - La stessa istruzione di pulizia di cui abbiamo parlato nel caso precedente vale anche per risolvere questo problema *lato software*, sempre prima dell'operazione di trasferimento (in modo che la versione dei dati in RAMsia la più recente, cioè quella ottenuta dal controllore DMA).

1.1.2 DMA e MMU

Reintroduciamo infine la MMU. Qui il problema sarà chiaramente che la CPU conoscerà indirizzi virtuali, mentre la DMU avrà bisogno di indirizzi fisici.

Far passare la DMU attraverso la MMU non sarà una soluzione, in quanto non possiamo essere sicuri che durante l'operazione di trasferimento l'albero di traduzione resta lo stesso.

Abbiamo quindi che il problema dovrà essere risolto lato software, passando al controllore DMA direttamente indirizzi fisici, attraverso la finestra FM. A questo punto però non potremmo aspettarci il corretto trasferimento di regioni di memoria di dimensione superiore a quella di una pagina, in quanto il controllore DMA non ha modo di capire quando passare da una pagina all'altra, o dove queste pagine siano in primo luogo.

Vogliamo quindi che i buffer che passiamo al controllore DMA non superino i confini di una pagina, e saremo quindi costretti a segmentare buffer che passano per più confini.

Infine, vorremo che il kernel si impegni a mantenere costante l'impiego dei buffer forniti al controllore DMA, cioè non effettui swap in o swap out dei processi che li forniscono mentre l'operazione di trasferimento è ancora in corso, in quanto il controllore non ha modo di rilevare tali variazioni.

1.1.3 DMA nel bus PCI

Inseriamo quindi il controllore DMA nel bus PCI.

In questo caso sarà il ponte ospite-PCI ad occuparsi del DMA.

Avremo quindi bisogno che nei singoli bus PCI ogni interfaccia abbia la possibilità di prendere il controllo del bus, secondo il cosiddetto **bus mastering**. Il bus mastering nei bus PCI viene gestito da un **arbitro**, che è collegato con linee `REQ` e `ACK` ad ogni interfaccia. Le interfacce richiedono quindi accesso al bus attraverso la `REQ`, mentre questo gli risponde attraverso la `ACK`.

Le operazioni potranno essere poi *bufferizzate* dal ponte ospite-PCI, cioè questo potrà memorizzare le modifiche in RAM ottenute lato bus PCI, per poi riportarle in differita alla RAM vera e propria.

Questo potrebbe dare dei problemi per quanto riguarda la sincronizzazione fra interfacce e CPU, in quanto un'interfaccia potrebbe inviare un segnale di termine operazione attraverso l'APIC, quando essa *crede* l'operazione sia finita (e lo farà perche lato bus PCI questa effettivamente lo è), mentre lato bus locale il ponte non ha ancora attualizzato i dati in RAM.

- La soluzione *lato hardware* sarà quella di collegare l'APIC al ponte, in modo da poter ritardare le interruzioni esterne alla fine delle operazioni di trasferimento.

Possiamo anticipare che nei processori moderni gli interrupt si inviano attraverso scritture in regioni specifiche di memoria, che l'apparato CPU riconosce autonomamente come interruzione. Questo corrisponde in un delay naturale fra i trasferimenti e l'invio di interruzioni da una stessa interfaccia.

- lato software nulla?

Notiamo che entrambe le soluzioni richiedono che il ponte adotti una politica FIFO alla gestione dei trasferimenti.