1 Lezione del 29-04-25

Riprendiamo il discorso del DMA nella prospettiva di un esempio concreto.

1.1 Hard disk e DMA

Fra i dispositivi visti finora solo l'hard disk è quello capace di fare DMA nel kernel. Dentro la macchina virtuale QEMU è disponibile un'emulazione dell'hard disk del PC AT (l'HD ATA visto in 4.1). Questo non era capace di fare DMA in autonomia, ma era bensì collegato ad un controllore DMA.

Fra i comandi disponibili per comunicare con l'hard disk ci sono quindi comandi dedicati a letture e scritture in DMA. Quando tali comandi vengono inviati all'hard disk, questo si occupa di coinvolgere il controllore DMA.

Questa non è più la situazione odierna: l'hard disk ATA con cui comunica la macchina emulata è situato sul bus ATA, che si collega al bus PCI con un ponte PCI-ATA. E' quindi il ponte a comportarsi come il controllore DMA, lato bus ATA.

Considerazioni storiche a parte, vediamo la struttura del controllore DMA dell'hard disk ATA, come descritto nella specifica reperibile a `https://calcolatori.iet.unipi.it/deep/idems100.pdf`. Abbiamo che questo può gestire due dischi separati, denominati *primario* e *secondario*, con relativi registri:

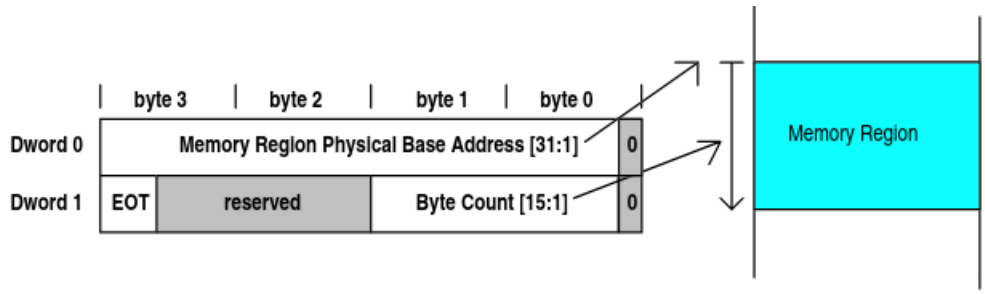
	Primario
`0x?? + 0`	**BMCMD**, *Bus Master Command*
`0x?? + 1`	Specifico al dispositivo
`0x?? + 2`	**BMSTR**, *Bus Master Status Register*
`0x?? + 3`	Specifico al dispositivo
`0x?? + 4-7`	**BMDTPR**, *Bus Master Descriptor Table Pointer*
	Secondario
`0x?? + 8`	**BMCMD**, *Bus Master Command*
`0x?? + 9`	Specifico al dispositivo
`0x?? + a`	**BMSTR**, *Bus Master Status Register*
`0x?? + b`	Specifico al dispositivo
`0x?? + c-f`	**BMDTPR**, *Bus Master Descriptor Table Pointer*

Riguardo a ogni registro avremo:

- **BMCMD**, *Bus Master Command*: questo specifica il tipo di operazione che vogliamo eseguire (lettura o scrittura), e ne specifica l'inizio. Per lanciare un'operazione, infatti, il software dovrà impostare il bit di *Read or Write Control* (bit 3), e successivamente alzare il bit *Start/Stop Bus Master* (bit 0);
- **BMSTR**, *Bus Master Status Register*: indica lo stato corrente del dispositivo a cui corrisponde. In particolare ci sono di interesse i primi 3 bit meno significativi (gli altri danno principalmente informazioni rispetto alle funzioni supportate dai dispositivi). Questi saranno:
  - Bit 2: rappresenta l'**interruzione**, viene alzato quando la trasmissione di dati in DMA è stata completata;
  - Bit 1: rappresenta uno stato di **errore**;
  - Bit 0: indica se il bus mastering è attivo o meno, cioè viene alzato quando il software scrive 1 sul bit start/stop bus master del BMCMD.

Abbiamo poi che i bit 1 e 2 possono essere resettati scrivendovi 1 (ed è questo passo che termina l'handshake col controllore DMA).

- **BMDTPR**, *Bus Master Descriptor Table Pointer*: questo punta alla prima entrata della cosiddetta tabella **PRD**, *Physical Region Descriptor Table*. Questa è una tabella di entrate da 8 byte, allineate ai 4 byte, che indicano l'indirizzo base della regione da trasferire, il numero di byte da trasferire e se l'entrata corrente è l'ultima della tabella (il controllore DMA continua a scorrere le entrate finchè non raggiunge l'ultima). La struttura delle entrate PRD è la seguente:



Notiamo che le regioni indicate dall'indirizzo base dell'entrata PRD può essere al massimo di 64 KiB. Per questo lato hardware si può usare un sommatore a sole 16 cifre. In ogni caso, questo non sarà un problema in quanto vorremo trasferire buffer in memoria virtuale una pagina (4 KiB) alla volta.

A questo punto basterà definire i passaggi di un operazione di trasferimento:

1. Si prepara una tabella PRD in memoria;
2. Si carica l'indirizzo base della tabella PRD nel registro BMDTPR, quindi si ripuliscono i bit di interruzione ed errore del registro di stato BMSTR;
3. Si fornisce il comando appropiato sul registro BMCMD;
4. Si attiva il bit 0 del registro BMCMD per attivare il bus mastering;
5. Il controllore DMA trasferisce i dati secondo quanto disposto finora;
6. Alla fine della trasmissione il controllore segnala la fine dell'operazione su una linea di interruzinoe;
7. In risposta all'interruzione, si resetta il bit 0 del registro BMCMD, e si legge lo stato dal controllore e dal disco per capire se l'operazione è andata a buon fine.

Vediamo quindi un semplice programma, che reinterpreta effettivamente quanto fatto in 4.1, usando il controllore DMA anzichè il controllo programma per effettuare gli accessi al disco rigido. Altre modifiche sono state fatte rispetto all'esempio in 4.1 per usare il più possibile le utilità fornite da `libce`.

```
#include <libce.h>
#include "video.h"

// definiti in buffer.s
extern natl prd[]; // la tabella PRD
extern natb buffer[]; // il buffer dati
```

```
#define BUF_SIZE 512

/*
 * Interrupt
 */
// codice interruzione hard disk
#define HD_VECT 0x60

// handler
extern "C" void a_int();
extern "C" void c_int() {
  // fai l'acknowledge (passo 7)
  bm::ack(); // ack bus mastering
  hd::ack(); // ack disco
  apic::send_EOI();
}

void init_int() {
  // imposta l'IREQ 14 al codice HD_VECT
  apic::set_VECT(14, HD_VECT);
  // carica l'handler
  gate_init(HD_VECT, a_int);
  // smaschera l'IREQ 14
  apic::set_MIRQ(14, false);
}

/*
 * Disco rigido
 */

// registri disco
const ioaddr disk_buffer = 0x01F0;
const ioaddr disk_status = 0x01F7;
const ioaddr disk_sectors = 0x01F2;
const ioaddr disk_command = 0x01F7;

// registri indirizzo LBA (sarebbero SNR CNL CNH HND)
const ioaddr disk_lba0 = 0x01F3;
const ioaddr disk_lba1 = 0x01F4;
const ioaddr disk_lba2 = 0x01F5;
const ioaddr disk_lba3 = 0x01F6;

// indirizzo lba disco
natl lba = 1;

// dai indirizzo LBA al controllore disco
void give_lba(natl lba) {
  // dividi in 4 byte
  natb lba0 = lba;
  natb lba1 = lba << 8;
  natb lba2 = lba << 16;
  natb lba3 = lba << 24;

  // il byte piu' significativo deve attivare l'LBA,
  // lba stava comunque su 28 bit
  lba3 = (lba3 & 0x0F) | 0xE0; // 1110-LBA-

  outputb(lba0, disk_lba0);
  outputb(lba1, disk_lba1);
  outputb(lba2, disk_lba2);
```

```
  outputb(lba3, disk_lba3);
}

// dai comando al controllore disco
void give_command(natl lba, natb sectors, natb cmd) {
  give_lba(lba);
  outputb(sectors, disk_sectors);
  outputb(cmd, disk_command);
}

/*
 * Controller DMA
 */

// indirizzo dispositivo bus mastering
natb bus = 0, dev = 0, fun = 0;

// inizializza dispositivo bus masetering
void init_bm() {
  bm::find(bus, dev, fun);
  bm::init(bus, dev, fun);
}

// prepara tabella PRD
void prepare_prd() {
  prd[0] = reinterpret_cast<natq>(buffer);
           // byte EOT | dim. buffer
  prd[1] = 0x80000000 | (512UL & 0xFFFF);
}

// effettua un operazione disco in bus mastering
void bm_op(bool write) {
  // il PRD e' gia pronto (passo 1)

  // carica il PRD (passo 2)
  bm::prepare(reinterpret_cast<paddr>(prd), write);

  // dai il comando (passo 3)
  give_command(lba, 1, write ? hd::WRITE_DMA : hd::READ_DMA);

  // inizia il bus mastering (passo 4)
  bm::start();

  // adesso il controllore DMA effettuera' i passi 5 e 6
}

/*
 * Console (video/tastiera)
 */

// svuota il buffer
void init_buffer() {
  for(int i = 0; i < BUF_SIZE; i++) {
    buffer[i] = 0x00;
  }
}

// make code salva (1) e carica (2)
const natb save_code = 0x02; // sarebbe 1
const natb load_code = 0x03; // sarebbe 2
```

```
127
128 // make code esc
129 natb esc_code = 0x01;
130
131 // make code backspace
132 natb back_code = 0x0E;
133
134 // cursore buffer testo
135 natl cursor = 0;
136
137 // sposta il cursore senza uscire dal buffer
138 inline void mov_cursor(int d) {
139   if(cursor == 0 && d < 0) return;
140
141   cursor += d;
142   if(cursor >= BUF_SIZE) cursor = BUF_SIZE - 1;
143 }
144
145 void main() {
146   // inizializza il gestore di interrupt
147   init_int();
148
149   // attiva gli interrupt disco
150   hd::enable_intr();
151
152   // inizializza il controllore in bus mastering
153   init_bm();
154
155   // prepara il prd
156   prepare_prd();
157
158   // svuota il buffer
159   init_buffer();
160
161   // vai in un ciclo di lettura
162   while(true) {
163     // aggiorna schermo
164     prt_screen(buffer, BUF_SIZE);
165     set_cursor(cursor);
166
167     // ottieni stato tastiera
168     natb make_code = kbd::get_code();
169
170     if(make_code == esc_code) break;
171     if(make_code == back_code) {
172       mov_cursor(-1);
173       buffer[cursor] = 0x00;
174       continue;
175     }
176
177     if(make_code == save_code) {
178       bm_op(true); // scrivi
179       continue;
180     }
181     if(make_code == load_code) {
182       bm_op(false); // leggi
183       continue;
184     }
185
186     char c = kbd::conv(make_code);
```

```
187     if(c != '\0') {
188       buffer[cursor] = c;
189       mov_cursor(1);
190     }
191   }
192 }
```

La tabella PRD e il buffer hanno dei prerequisiti particolari sui confini che possono attraversare:

- Le entrate PRD devono essere di 8 byte allineate ai 4 byte;
- Il buffer deve essere allineato ai 2 byte, e non attraversare confini allineati ai 64 KiB.

Per questo motivo li definiamo in un file a parte, in assembler, `buffer.s`:

```
1  .data
2
3  // prd
4  .balign 4
5  .global prd
6  prd:
7    .fill 16384, 4
8
9  // buffer
10 .balign 65536
11 .global buffer
12 buffer:
13   .fill 512, 1
```

Infine, definiamo a parte anche la parte assembler del gestore d'interuzione `a_int`:

```
1 #include <libce.h>
2
3 .extern c_int
4 .global a_int
5 a_int:
6   salva_registri
7   call c_int
8   carica_registri
9   iretq
```

1.1.1 Controller IDE su bus PCI

Per l'inserzione di un controllore di questo tipo in un bus PCI dobbiamo renderci conto di alcuni dettagli: Nei registri dello spazio di configurazione del dispositivo si devono attivare dei flag particolari per segnalare la possibilità che questo lavori in bus mastering.

1.1.2 Controller IDE nel kernel

Vediamo infine come il controllore DMA dell'hard disk ATA viene gestito nel kernel. La libreria `libce` definisce i registri del controllore:

```
1 namespace bm {
2   extern ioaddr iBMCMD; // Bus Master Command
3   extern ioaddr iBMSTR; // Bus Master Status Register
4   extern ioaddr iBMDTPR; // Bus Master Descriptor Table Pointer
5 }
```

e le relative funzioni per l'inizializzazione, l'acknowledge, ecc...

L'unica interfaccia ATA montata nel sistema è quindi descritta dal descrittore:

```
// descrittore di interfaccia ATA
struct des_ata {
  // Ultimo comando inviato all'interfaccia
  natb comando;
  // Indice di un semaforo di mutua esclusione
  natl mutex;
  // Indice di un semaforo di sincronizzazione
  natl sincr;
  // Quanti settori resta da leggere o scrivere
  natb cont;
  // Da dove leggere/dove scrivere il prossimo settore
  natb* punt;
  // Array dei descrittori per il Bus Mastering
  natl* prd;
};
```

che tiene conto dell'operazione corrente.

A questo punto il processo esterno dedicato all'hard disk dovrà limitarsi ad inviare i comandi corretti seguendo la scaletta appena riportata. Unica parte di interesse è quella della preparazione della tabella PRD, per cui bisogna tenere conto che il controllore DMA necessita di indirizzi fisici, e che legge sequenzialmente a partire da tali indirizzi fisici (perciò non si possono superare i 4 KiB della dimensione di pagina). Per fare questo, e tenere conto di buffer in memoria che iniziano potenzialmente a metà pagina, si sfrutta la funzione `prepare_prd()`:

```
bool prepare_prd(des_ata *d, natb* vett, natb quanti)
{
  // ottieni il numero di byte da trasferire
  natq n = quanti * DIM_BLOCK;

  int i = 0;

  // scorri
  while (n && i < MAX_PRD) {
    // ottieni l'indirizzo fisico dell'indirizzo corrente
    paddr p = trasforma(vett);

    // otteni il numero di byte nella pagina corrente
    // sarebbe dimensione_pagina - scarto
    natq  r = DIM_PAGINA - (p % DIM_PAGINA);

    // se eccede il numero di byte, taglia
    if (r > n)
      r = n;

    // imposta l'entrata PRD
    d->prd[i] = p;
    d->prd[i + 1] = r;

    // rimuovi da n il numero di byte presi
    n -= r;
    // avanza il vettore del numero di byte presi
    vett += r;

    // passa alla prossima entrata PRD
    i += 2;
  }
```

```
33
34    // se non hai coperto tutti i byte e' errore
35    if (n)
36      return false;
37
38    // imposta il bit end of table
39    d->prd[i - 1] |= 0x80000000;
40    return true;
41  }
```

Un dettaglio interessante è che si usa la `trasforma()` per ogni entrata PRD che si va a generare, in quanto chiaramente ognuna di queste avrà bisogno di un nuovo indirizzo fisico. Per questo motivo si mantiene oltre al numero di byte mancanti anche l'indirizzo corrente all'interno del vettore (in `vett`).

A questo punto si possono fornire all'utente primitive per l'accesso all'hard disk sia a controllo interruzione (come avevamo già visto, implementato in `libce`) sia in DMA. Queste saranno:

- **Controllo interruzione:** vediamo ad esempio l'operazione di ingresso.

```
1  // fondamentalmente un wrapper per hd::start_cmd di libce, che
       aggiorna il descrittore
2  void starthd_in(des_ata* d, natb vetti[], natl primo, natb quanti)
3  {
4    d->cont = quanti;
5    d->punt = vetti;
6    d->comando = hd::READ_SECT;
7    hd::start_cmd(primo, quanti, hd::READ_SECT);
8  }
9
10 // la primitiva vera e propria
11 extern "C" void c_readhd_n(natb vetti[], natl primo, natb quanti)
12 {
13   des_ata* d = &hard_disk;
14
15   // controlli (c_access)
16
17   sem_wait(d->mutex);
18   starthd_in(d, vetti, primo, quanti);
19   sem_wait(d->sincr);
20   sem_signal(d->mutex);
21 }
```

- **DMA:** vediamo sempre l'operazione di ingresso:

```
1  void dmastarthd_in(des_ata* d, natb vetti[], natl primo, natb quanti)
2  {
3    // passo 1 della scaletta
4    if (!prepare_prd(d, vetti, quanti)) {
5      flog(LOG_ERR, "dmastarthd_in: numero di PRD insufficiente");
6      sem_signal(d->sincr);
7      return;
8    }
9
10   d->comando = hd::READ_DMA;
11   d->cont = 1;
12
13   // passo 2
14   paddr prd = trasforma(d->prd);
15   bm::prepare(prd, false);
```

```
16
17    // passo 3
18    hd::start_cmd(primo, quanti, hd::READ_DMA);
19    bm::start();
20  }
```

A operazioni terminate, il processo esterno dovrà chiaramente interpretare correttamente le interruzoni che riceve in base al tipo di comando dato:

```
1  void estern_hd(natq)
2  {
3    des_ata* d = &hard_disk;
4    for(;;) {
5      d->cont--;
6      hd::ack();
7      switch (d->comando) {
8      // questi sono i casi gia visti
9      case hd::READ_SECT:
10       hd::input_sect(d->punt);
11       d->punt += DIM_BLOCK;
12       break;
13     case hd::WRITE_SECT:
14       if (d->cont != 0) {
15         hd::output_sect(d->punt);
16         d->punt += DIM_BLOCK;
17       }
18       break;
19     case hd::READ_DMA:
20     case hd::WRITE_DMA:
21       // qui si fa l'acknowledge, passo 7 della scaletta
22       bm::ack();
23       break;
24     }
25     if (d->cont == 0)
26       sem_signal(d->sincr);
27     wfi();
28   }
29 }
```