Appunti Calcolatori Elettronici

Luca Seggiani
2025

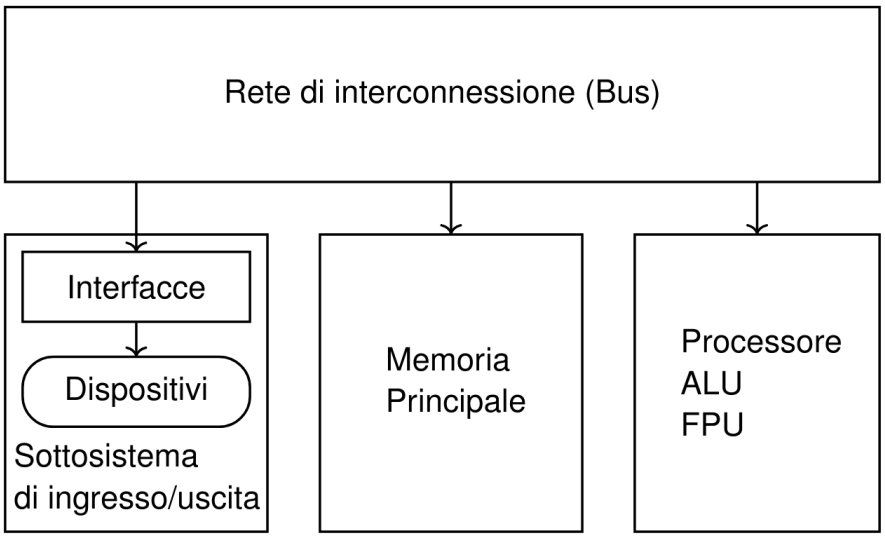
1 Lezione del 24-02-25

1.1 Introduzione al corso

Continuiamo lo studio di una particolare architettura per calcolatori, a partire da quanto detto riguardo alle reti logiche, introducendo i concetti di **interruzione**, **protezione** e **memoria virtuale**. Questi 3 strumenti ci permetterano di realizzare il paradigma della **multiprogrammazione**, cioè di far eseguire ad una macchina con un singolo processore più programmi contemporaneamente. Non si pensi questo significhi avere più processori, in quanto il corso riguarda esclusivamente processori *single-threading*.

1.2 Architettura

L'architettura di riferimento è quella classica, composta da **CPU**, **memoria** e **I/O** interconnessi da un **bus**:



Durante lo studio di un archiettura è oppurtuno porsi la domanda *"chi fa cosa?"*, che fornisce determinati *chi* ai determinati *cosa* forniti da un opportuno livello di astrazione (transistor, porte logiche, diagrammi funzionali, ecc...).

La domanda che potremo porci adesso è *"chi comanda?"* all'interno dell'architettura vista. La risposta più giusta è quella del **software**: l'architettura è fatta per *eseguire* software.

Per convincerci di questo possiamo sostituire la domanda *"chi fa cosa?"* con la domanda *"chi sa cosa?"*.

- La **CPU** conosce lo stato corrente dei registri e l'istruzione in esecuzione. Fra un'istruzione e l'altra non c'è alcun bisogno di sapere cosa è accaduto finora, e cosa accadrà in futuro, ma solamente l'istruzione corrente. Quindi si può pensare che la CPU non *sa* qual'è l'obiettivo della computazione, ma si limita a portarla avanti.
- La **memoria** è un oggetto passivo, che contiene il programma, ma si limita a restiture i dati richiesti quando sono richiesti. Notiamo che le memorie che usiamo

sono ad **accesso casuale**, ergo nessuno scorre alla ricerca di indirizzi, ma si può leggere e scrivere in posizioni arbitrarie in tempo pressoché costante. La memoria contiene **sempre** qualcosa, che questo sia significativo o meno, e la sua tipizzazione dipende solamente dalle intenzioni del programmatore.

- L'**I/O** è il componente più variegato dell'architettura. L'unica costante che rende la comunicazione con le periferiche più facile è la presenza di un interfaccia, che riduce tale comunicazione ad una semplice lettura o scrittura nello spazio di I/O. La differenza fra le letture e scritture nello spazio di I/O e lo spazio di memoria è la possibile presenza di **effetti collaterali**, cioè effetti non riconducibili alla sola variazione di stato di una locazione di memoria. Inoltre la CPU non è l'unica a scrivere nello spazio di I/O, in quanto questo può essere fatto anche dalle periferiche stesse.

- Il **bus** è un insieme di linee (*fili*), che trasportano ciò che ogni componente sta comunicando in un dato momento. Ogni componente vede ciò che viene scritto sul bus in qualsiasi momento, e l'indirizzamento di locazioni specifiche nello spazio di memoria o nello spazio di I/O viene fatto attraverso **maschere** di indirizzo.

1.2.1 Flusso di controllo

Abbiamo visto come la CPU si limita a prelevare ed eseguire istruzioni nel ciclo di **fetch-execute**. L'istruzione successiva alla corrente, il cui indirizzo viene scritto nell'**instruction pointer**, viene decisa dall'istruzione corrente stessa (si pensi alle istruzioni di salto). Il **flusso di controllo** è quindi deciso dall'istruzioni stesse, cioè dal programma.

Vedremo che mentre la struttura del calcolatore va a complicarsi, il flusso di controllo smette di essere completamente sequenziale (anche oltre alle istruzioni di salto), sopratutto grazie al meccanismo che introdurremo di *interruzione*.

1.2.2 Bootstrap

Il **bootstrap** è un processo secondo il quale si porta il sistema in un certo stato di esecuzione, apparentemente impossibile, o comunque molto difficile, da raggiungere. Ad esempio, il compilatore del linguaggio C è scritto esso stesso in linguaggio C. La domanda naturale è *"come è stato compilato il compilatore?"*. La risposta è un processo di boostrap, usando o un compilatore presesistente, magari che implementa un sottoinsieme parziale del C, o scrivendo l'intero compilatore in linguaggio macchina, cioè assemblando codice assembler.

Il boostrap si rende necessario anche all'avvio del calcolatore, per il caricamento del programma all'interno della memoria e l'inizio dell'esecuzione. Nei calcolatori moderni questo viene fatto attraverso la **ROM**, cioè una memoria a sola lettura che contiene un programma di bootstrap. All'avvio il processore è impostato in modo che al reset prenda come indirizzo proprio quello della ROM, e quindi inizi ad eseguire il programma di boostrap. All'interno della ROM si trova, nei calcolatori moderni, il **BIOS** (o *UEFI*, nei sistemi moderni), che ha il solo compito di impostare alcune periferiche di base e caricare il sistema operativo.

Iniziamo quindi ad approfondire, uno per uno, i moduli dell'architettura.

1.3 Memoria

La memoria è un insieme contiguo di locazioni di memoria, che nelle architetture moderne sono rappresentate da byte. Storicamente, la memoria era indirizzata a *parole*, cioè insiemi di bit coincidenti in dimensioni coi registri del processore. Una parola poteva essere di più byte, mentre oggi le memorie sono accessibili ai singoli byte. Ad esempio, le memorie usate nell'architettura Intel x86 sono accessibili ad 1 byte (`MOVB`), 2 byte (`MOVW`), 4 byte (`MOVL`), e 8 byte (`MOVQ`).

Il fatto che la memoria dell'architettura x86 sia organizzata a *parole* da 8 byte (in particolare è cosi nella versione a 64 bit, x86_64, che chiameremo semplicemente x86) comporta che cambi il modo stesso in cui vediamo la memoria. Invece di vedere solo byte contigui, infatti, possiamo immaginare la memoria come organizzata in **righe**, a loro volta divise in 8 byte:

Numero di riga	+7	+6	+5	+4	+3	+2	+1	+0	**Indirizzo di riga**
0									0
1									8
2									16
...									...

1.3.1 Endianess

Notiamo che la posizione in memoria del byte più significativo di una parola (in questo caso consideriamo una "parola" da 8 byte, da cui si ricavano tutte le altre misure) determina l'*endianess* dell'architettura. In particolare, se l'ultimo byte sta in fondo nella memoria, si dice **big-endian**, mentre se viceversa l'ultimo byte viene per primo nella memoria, si dice **little-endian**.

L'architettura Intel x86 che andiamo a considerare è little-endian, come lo sono la maggior parte delle architetture moderne. Un esempio di utilizzo del big-endian e nella trasmissione di dati attraverso il protocollo IP, usato nelle comunicazioni Internet.

Il formalismo introdotto nel paragrafo precedente si dimostra utile anche da questo punto di vista: scrivendo le parole da destra verso sinistra, cioè a offset nella parola crescenti verso sinistra, si ha che il MSB finisce a sinistra, come suggerirebbe il senso di scrittura naturale.

1.3.2 Allineamento

Indicheremo con **offset** la distanza in byte fra due locazioni di memoria, intesa come il numero di locazioni che vanno saltate per raggiungere un indirizzo a partire dall'altro. In questo ha senso parlare anche di offset *negativi*.

Visto che lo spazio di memoria è effettivamente ciclico, cioè si ha *wrap-around* ai suoi capi, si ha che gli offset rimangono validi **modulo** la dimensione dello spazio di memoria, che è sempre 2^n, con n nel nostro caso uguale a 64.

Il *wrap-around* si comporta bene con gli offset, ma lo stesso non si può dire per quanto riguarda **intervalli** di byte. Preso un certo intervallo $[x, y)$, quindi, si ha che questo contiene gli indirizzi $\{n \mid x \leq n < y\}$, ammesso che $x < y$, cosa che risulta falsa nel caso di intervalli che hanno *wrap-around*. Decidiamo di non considerare intervalli di questo tipo. Questo rende necessaria un'eccezione per intervalli che comprendono l'ultimo byte: in questo caso è concesso $[x, 0)$, con 0 che indica il fondo dello spazio di memoria.

Veniamo quindi all'**allineamento**. Dire che un indirizzo è allineato ad un numero n significa dire che quell'indirizzo è un multiplo di n. Chiaramente, conviene scegliere n potenze di 2. In questo caso, per riconoscere se un indirizzo è allineato a 2^k, basta guardare i suoi ultimi k bit.

Si dice spesso che oggetti sono *allineati alla parola*, ecc... Questo significa che sono allineati alla *dimensione* della parola specificata. Altrimenti, si può dire che un oggetto è allineato *naturalmente*, nel caso in cui sia allineato alla dimensione di stesso.

Infine, il **confine** di un oggetto è l'indirizzo che lo delimita dal resto dello spazio di memoria.

2 Lezione del 25-02-25

2.1 Interazione fra CPU e memoria

Nell'architettura Intel x86 la CPU interroga la RAM in due situazioni:

- Durante la lettura di un istruzione;
- Durante la lettura di *eventuali* operandi in memoria richiesti dall'istruzione. Notiamo che per ogni istruzione è previsto un solo indirizzo esplicito di un operando in memoria (non è permesso scrivere qualcosa come `MOV (%RBP), (%RDI)`). Alcune istruzioni possono però avere comunque più di un operando in memoria (ad esempio le istruzioni di stringa, `MOVS`, ecc... o la stessa istruzione di pila `POP`).

Dal punto di vista pratico, il collegamento fra CPU e RAM è quindi rappresentato da:

- Un **bus dati** a 64 bit;
- Un certo numero di linee per il **numero di riga**. Questo non corrisponde all'indirizzo del primo byte contenuto in ogni riga, ma l'indice proprio di ogni regione (intesa come riga) da 64 bit all'interno della RAM. Si noti inoltre che queste non sono necessariamente disposte per indirizzare un numero di byte pari a 2^{64}, o 2^{57} (il massimo spazio indirizzabile secondo l'architettura x86), ma più spesso intorno ai 2^{36}-2^{37};
- Determinate **linee di controllo** che segnalano l'operazione in corso da parte del processore.
- 8 linee di **byte enable**, attive basse, che rappresentano i byte di interesse all'interno di ogni locazione da 64 bit della RAM. Dal punto di vista della lettura, queste linee non sono particolarmente utili in quanto tutta la locazione verrà comunque riportata sul bus dati, o comunque le locazioni non selezionate potranno essere invalide o in alta impedenza, senza avere effetto sulla CPU (che non le leggerà). Per quanto riguarda la scrittura, invece, la RAM lascerà inalterati i byte con byte enable alto.

2.1.1 Struttura della RAM

Modellizziamo un modulo di RAM come una rete provvista di:

- Una linea di **select**, attiva bassa;

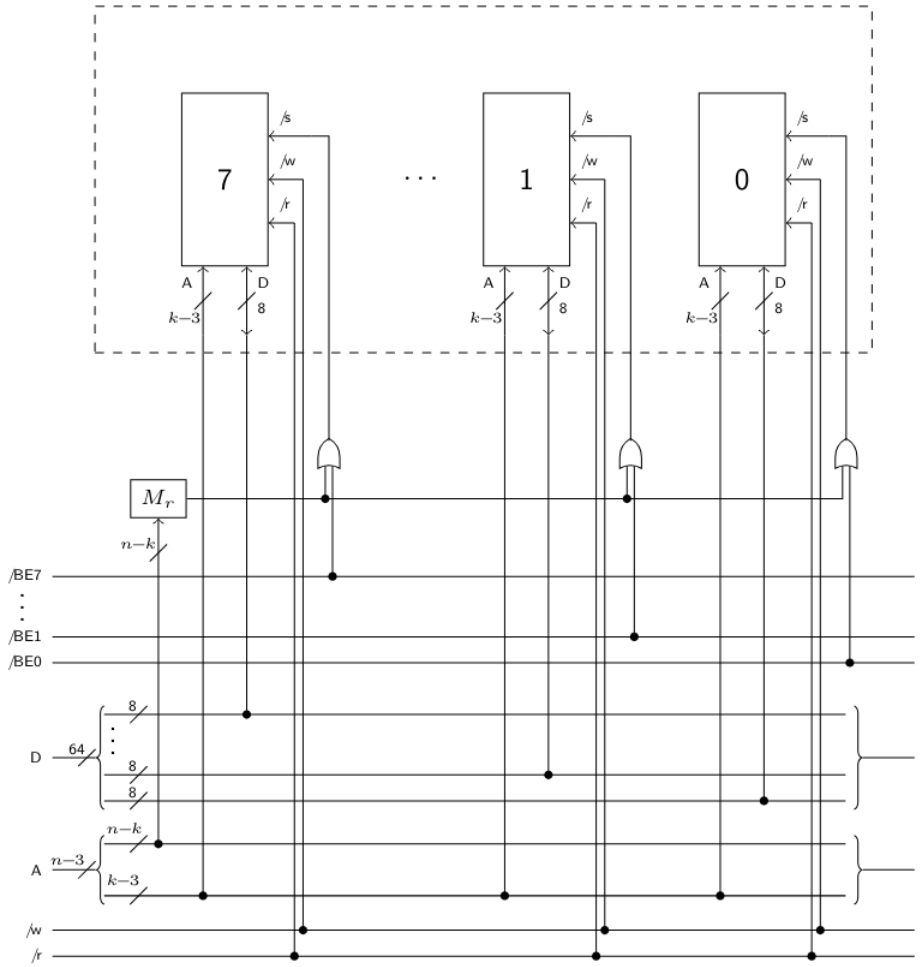
- Le **linee di indirizzo**;
- Una linea di *memory read* e una linea di *memory write*, o comunque un certo numero di **linee di controllo** necessarie all'accesso in scrittura e lettura;
- Un **bus dati** di ingresso/uscita.

Dalla CPU arriveranno, come abbiamo detto, i **numeri di riga**, i **byte enable**, il **bus dati** e le **linee di controllo**.

I numeri di riga si collegano direttamente alle linee di indirizzo di ogni modulo (o lo fanno le prime k nel caso di banchi di dimensione 2^k). Ogni modulo rappresenterà allora un byte della locazione (avremo quindi, nell'architettura descritta, 8 moduli per 8 byte per banco di memoria, quindi 64 bit). I byte enable dovranno quindi smistarsi nelle linee di select di ogni modulo di RAM, a selezionare il modulo corrispondente. Il bus dati verrà composto, analogamente, concatenando le linee di uscita da 8 bit di ogni modulo di RAM. Notiamo che avevamo chiamato questo montaggio **parallelo**.

Vorremo poter estendere la memoria disponibile oltre il numero di locazioni fornite da ogni banco di RAM. Pensiamo di fare questo attraverso più banchi di memoria con locazioni da 64 bit (che possono tranquillamente essere realizzati anch'essi unendo 8 banchi da 1 byte). In questo caso avremo bisogno di montaggio in **serie**, e quindi di generare un segnale di select a partire non solo dalle line di byte enable, ma anche da una **maschera** generata a partire dagli $n - k$ numeri di riga. Questo si potrà fare agevolmente mettendo il segnale di uscita della maschera in OR (ricordiamo segnali attivi bassi, quindi si applica De Morgan) con il byte enable di ogni modulo di RAM compreso nel banco di memoria associato a tale maschera.

La struttura complessiva della RAM sarà quindi la seguente:



dove la rete M_r è quella che si occupa di generare la maschera, prendendo banchi di dimensione 2^k righe.

2.1.2 Allineamento e RAM

Quanto discusso finora rende più chiaro l'importanza del corretto allineamento degli oggetti in memoria. Leggere un oggetto da 8 byte non allineato nel montaggio di RAM descritto, infatti, richiederà necessariamente 2 accessi, contro il singolo accesso necessario per un oggetto allineato. Inoltre, alcuni dei byte più significativi risulteranno invertiti di posto rispetto ai byte meno significativi, cioè si richiede un operazione di shift interna al processore.

Questa combinazione di operazioni, eseguite in **hardware**, rende gli accessi in memoria non allineati molto poco performanti, e quindi sconsigliati (anche se l'architettura Intel x86 li permette comunque, probabilmente malvolentieri).

2.1.3 Posizione dei confini in RAM

Un altro problema che potrebbe interessarci è, data una regione di memoria $[x, y)$ di dimensione b uguale a un singolo banco di RAM, ottenere gli indici della prima regione

in cui cade l'intervallo, e la prima in cui non cade più, cioè gli indici delle regioni di appartenenza suoi *confini*.

Vediamo come calcolare la prima regione di appartenenza. In **hardware**, questo può essere calcolato semplicemente prendendo gli $n - b$ bit più significativi dei numeri di riga x e y. In **software**, questo equivarrà ad uno shift a destra che conservi i soli $n - b$ bit più significativi.

Mascherando gli stessi bit, invece, si può ottenere l'indirizzo (offset) all'interno del banco del confine della regione. Per la precisione, vogliamo una maschera fatta da $n - b$ 0 e b 1. Questa si può ricavare agevolmente prendendo 2^b come `1UL << b` e sottraendogli 1 (`UL` è da intendersi come il suffisso di letterale *unsigned long* del C), ottenendo la maschera desiderata (si avranno borrow propagati dal bit in b fino al LSB).

Infine, vediamo come calcolare la prima regione di non appartenenza. In questo caso potremo calcolare la regione in cui cade $y - 1$, e aggiungervi 1 (tenendo conto di eventuali *wrap-around*). Il -1 è richiesto dal fatto che y potrebbe cadere sul confine. In questo caso avremo `((y - 1) >> b) + 1`, considerata somma modulo $n - b$. Alternativamente, si può prendere $y + b - 1$ e calcolarne la regione di appartenenza. Il -1 è necessario per evitare il caso limite dove y è già allineato a b (in quel caso $y + b$, ricordando che è y è il primo indirizzo *non più contenuto* di $[x, y)$, punterebbe alla regione ancora dopo a quella desiderata).

Faremo considerazioni simili a queste quanto studieremo il meccanismo della *memoria paginata*, quando avremo bisogno di individuare i frame di appartenenza di regioni in memoria.

2.2 Spazio di I/O

Veniamo quindi alla trattazione dello spazio di I/O e delle interfacce ivi connesse. L'accesso alle periferiche viene fatto attraverso le istruzioni `IN` e `OUT`, ammesso che non ci sia nessun sistema operativo in esecuzione, ma solo il nostro programma, e appositi sottoprogrammi di ingresso/uscita, la cui struttura non è al momento importante.

Le periferiche che studieremo, per semplicità di trattazione, derivano in parte da quelle disponibili sui PC **IBM AT** (famiglia *IBM 5170*). I PC di questa categoria (compresi tutti i vari *IBM compatible*) si basavano sullo standard per periferiche **ISA** (*Industry Standard Architecture*). Visto che i PC moderni derivano dai vecchi IBM compatible, anche oggi si cerca di emulare (almeno in parte) questo standard.

Le periferiche, nello specifico saranno:

- La **tastiera**;
- Il **video** su VGA;
- Il **timer**;
- Gli **hard disk**.

2.3 Tastiera

Dal punto di vista funzionale, la tastiera deve solo scoprire quali tasti sono premuti e comunicarlo al calcolatore. In particolare, noi studieremo tastiere IBM che trasmettono secondo lo standard PS/2.

Nei PC IBM il tasto non restituisce il carattere ASCII del carattere premuto, ma un codice associato ad ogni tasto che va convertito in software. Questo codice viene ottenuto per *scansione* dell'intero piano della tastiera. Dal punto di vista meccanico, ci sono

tracce orizzontali e verticali disposte, rispettivamente, su ogni riga o colonna di tasti. La pressione di un tasto comporta una deformazione delle tracce che chiude un circuito fra la riga e la colonna del tasto corrispondente. Un **microcontrollore** (originariamente un Intel 8042) collegato sia alle tracce orizzontali che alle tracce verticali scansiona ciclicamente, con impulsi, o le righe leggendo le colonne, o le colonne leggendo rige, cercando un circuito chiuso. Un cortocircuito viene quindi rilevato dal microcontrollore, che aggiorna una (piccola) memoria interna con il tasto premuto. Di conseguenza, invia al calcolatore un segnale che codifica quali tasti sono stati premuti rispetto al precedente istante temporale, e quali tasti sono stati rilasciati rispetto al precedente istante temporale.

La tastiera non restituisce solo pressioni di tasti, ma anche i loro rilasci, cosa che può essere utile per ottenere combinazioni di tasti, pressioni estese nel tempo, ecc... I codici di pressione si dicono **make code**, mentre i codici di rilascio si dicono **break code** La stessa pressione ripetuta di un tasto quando l'utente lo tiene premuto per un certo istante temporale era, nei PC IBM, realizzata direttamente nella tastiera (tecnologia *type-matic*), tra l'altro con periodo configurabile. Tramite il *type-matic*, su appositi tasti abilitati, si ha infatti una ripetizione dell'evento di *pressione* (non rilascio) di un tasto a frequenza costante dopo un intervallo di pressione continua.

Abbiamo quindi che i make code e i break code dei diversi tasti in una classica tastiera layout US sono i seguenti, nel cosiddetto *set 0* (man mano che le tastiere si sono evolute, sono stati introdotti nuovi set con codifiche diverse):

ESC 0x01 0x81	\ | 0x2b 0xab	1 ! 0x02 0x82	2 @ 0x03 0x83	3 # 0x04 0x84	4 $ 0x05 0x85	5 % 0x06 0x86	6 ^ 0x07 0x87	7 & 0x08 0x88	8 * 0x09 0x89	9 (0x0a 0x8a	0) 0x0b 0x8b	- _ 0x0c 0x8c	= + 0x0d 0x8d	← 0x0e 0x8e
Tab 0x0f 0x8f	q Q 0x10 0x90	w W 0x11 0x91	e E 0x12 0x92	r R 0x13 0x93	t T 0x14 0x94	y Y 0x15 0x95	u U 0x16 0x96	i I 0x17 0x97	o O 0x18 0x98	p P 0x19 0x99	[{ 0x1a 0x8a	] } 0x0b 0x8b	ʕ•ᴥ•ʔ	
Caps Lock 0x3a 0xba	a A 0x1e 0x9e	s S 0x1f 0x9f	d D 0x20 0xa0	f F 0x21 0xa1	g G 0x22 0xa2	h H 0x23 0xa3	j J 0x24 0xa4	k K 0x25 0xa5	l L 0x26 0xa6	; : 0x19 0x99	' " 0x1a 0x8a	Enter 0x1c 0x9c		
Left Shift 0x2a 0xaa	z Z 0x2c 0xac	x X 0x2d 0xad	c C 0x2e 0xae	v V 0x2f 0xaf	b B 0x30 0xb0	n N 0x31 0xb1	m M 0x32 0xb2	. < 0x33 0xb3	. > 0x34 0xb4	/ ? 0x35 0xb5	Right Shift 0x36 0xb6		
Left Ctrl 0x1d 0x9d	Left Alt 0x38 0xb8	Space 0x39 0xb9							Right Alt 0xe0/38 0xe0/b8	Right Ctrl 0xe0/1d 0xe0/9d				

Lato calcolatore, il segnale prodotto dal microcontrollore della tastiera viene letto da un interfaccia provvista dei seguenti registri:

`0x60`	**RBR**, *Receive Buffer Register* **TBR**, *Transmit Buffer Register*
...	
`0x64`	**STR**, *Status Register* **CMR**, *Command register*

RBR e TBR, come STR e CMR, condividono gli indirizzi, rispettivamente `0x60` e `0x64`. Il RBR conterrà i make e break code, mentre l'STR conterrà i flag di stato sia per RBR che per TBR (rispettivamente ai bit 0 e 1).

Potremmo chiederci il significato di un registro di trasmissione TBR. Questo serve, ad esempio, a governare i led di stato per funzioni speciali quali Caps-Lock, Num-Lock, Scroll-Lock ecc... nonché a modificare le impostazioni del type-matic e, in maniera completamente slegata alla tastiera, a provocare il reset del PC, scrivendo `0xFE` in CMR.

Vediamo quindi un programma C++ per l'interazione con l'interfaccia di tastiera. Notiamo che la libreria all'header `libce.h` definisce alcuni tipi (qui `natb`, un naturale su 8 bit, e `ioaddr`, un indirizzo nello spazio di I/O) e funzioni (qui `inputb`, ottieni byte dallo spazio di I/O, e `vi::char_write()`, stampa un carattere a schermo).

```
#include <libce.h>
#define NUM_CODES 28

// indirizzi porte tastiera
const ioaddr rbr_addr = 0x60;
const ioaddr str_addr = 0x64;

// tabella make code
natb make_codes[] = {
  0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19,
  0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26,
  0x2c, 0xd, 0x2e, 0x2f, 0x30, 0x31, 0x32,
  0x1c, 0x39
};

// tabella caratteri minuscoli
char l_table[] = {
  'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p',
  'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l',
  'z', 'x', 'c', 'v', 'b', 'n', 'm',
  '\n', ' '
};

// tabella caratteri maiuscoli
char u_table[] = {
  'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P',
  'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L',
  'Z', 'X', 'C', 'V', 'B', 'N', 'M',
  '\n', ' '
};

// make code ESC
natb esc_code = 0x01;

// make code shfit
natb shift_down = 0x2A;
natb shift_up = 0xAA;

// gestione case
enum cas { lower, upper };
cas cur_cas = lower;

natb get_key() {
  natb status;

  // ciclo di lettura per il flag FI in str_addr
  do {
    status = inputb(str_addr);
  } while(!(status & 0x01));

  // FI alto, leggi da rbr_addr
  return inputb(rbr_addr);
}

char get_char(natb make_code) {
```

```
  // cerca il carattere per scansione lineare
  for(int i = 0; i < NUM_CODES; i++) {
    if(make_code == make_codes[i]) {
      // trovato, controlla il case corrente
      if(cur_cas == upper) {
        return u_table[i];
      } else {
        return l_table[i];
      }
    }
  }

  // carattere nullo come default
  return '\0';
}

void main() {
  while(true) {
    // otteni make code
    natb make_code = get_key();

    // se ESC, esci
    if(make_code == esc_code) {
      break;
    }

    // gestisci shift
    if(make_code == shift_down) {
      cur_cas = upper;
    }
    if(make_code == shift_up) {
      cur_cas = lower;
    }

    // ottieni carattere e stampa
    char c = get_char(make_code);
    vid::char_write(c);
  }
}
```

Dal programma si evincono subito gli indirizzi dei registri di Receive Buffer (RBR) e di stato (STR), a cui i registri trasmettitore e comando (TBR e CMR) sono sovrapposti. Il funzionamento è quindi ottenuto attraverso una lettura ciclica dei make code dall'interfaccia, e una scansione per il rilevamento del carattere selezionato a partire dal make code stesso. Altro codice è usato per gestire il tasto shift, e il termine dell'esecuzione alla pressione del tasto ESC.

3 Lezione del 03-03-25

3.1 Video

Il supporto principale al video è la **memoria video**, che lato software si comporta perlopiù come una normale memoria ad accesso casuale.

Questo è quindi il primo esempio di un oggetto che si trova nello spazio di memoria, senza necessariamente *essere* memoria: ciò che vi viene scritto non viene memorizzato, ma visualizzato sullo schermo.

Inoltre, la memoria video supporta un accesso *bidirezionale*: cioè vi si può accedere sia lato CPU che lato **adattatore video**, cioè la rete che si occupa di gestire tale memoria e visualizzarla sul *display*. Lo standard VGA usato dal PC IBM prevede che l'adattatore sia configurabile e utilizzabile in due modalità:

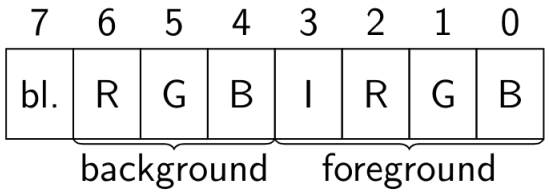
- **Modalità testo:** ogni locazione viene associata ad un carattere ASCII da visualizzare sullo schermo, diviso in 80 colonne $\times$ 25 righe (con un carattere di 9 $\times$ 16 pixel, per una risoluzione totale di 720 $\times$ 400 pixel a 70 Hz). È questa la modalità di default in cui si avvia l'adattatore.

  In questo caso il compito dell'adattatore è quello di leggere i 4 KB di memoria, e convertire ogni codice nel carattere principale. Questo viene fatto consultando una ROM di caratteri che contiene quello che è effettivamente il carattere (*font*) dell'adattatore. Solitamente si può anche redirezionare la lettura in ROM ad una certa regione della RAM, modificando così il font.

  La faccenda è veramente più complicata: si dedicano non 1 ma 2 byte ad ogni carattere, dove il byte più significativo rappresenta informazioni riguardo al **colore** del carattere:

  - I 4 bit meno significativi rappresentano il colore del *foreground*;
  - I 3 bit successivi rappresentano il colore del *background*;
  - Il bit più significativo rappresenta il *blinking*, cioè indica all'adattatore di far *lampeggiare* quel carattere nel tempo.

  Il layout dei flag è quindi:



  Parliamo quindi del **cursore** comunemente visualizzato sullo schermo nelle interfacce a riga di comando. La modalità testo non ha idea della posizione del cursore sullo schermo: attraverso registri si può indicare la posizione del cursore, e modificando la regione di memoria interessata si possono cambiare i caratteri in qualsiasi zona dello schermo. Il comportamento del cursore (spostamento, ritorno a capo, ritorno carrello, ecc...) è quindi gestito interamente lato software.

- **Modalità grafica:** programmando i registri dell'adattatore si possono ottenere diverse modalità grafiche, che permettono al progammatore di colorare singoli pixel sul display. Le modalità più popolari di questo tipo su schede VGA erano o a 640 $\times$ 400 pixel a 70 Hz (fase di boot grafica), o a 640 $\times$ 480 pixel a 60 Hz non interlacciati (la modalità di default di Microsoft Windows). Nella macchina virtuale usata incapsuliamo tale operazione di conversione in un apposita libreria, e scriviamo pixel con colori su 8 bit (per 256 colori diversi). Nei sistemi moderni la memoria video non viene scritta dalla CPU, ma da un *coprocessore grafico* che esegue un suo programma, mentre la CPU può dedicarsi ad altro.

3.1.1 Indirizzamento dei registri dell'adattatore video

Vediamo nel dettaglio come si possono indirizzare i registri interni dell'adattatore video. Questo dispone infatti di una vasta gamma di registri, ma una sola linea di ingresso da un byte per indirizzamento e scrittura. Le scritture vengono quindi eseguite inserie:

- Prima specificando l'**indirizzo** del registro da aggiornare;
- Poi inserendo i **dati** da scrivere a tale indirizzo.

Vediamo quindi un programma di esempio che sfrutta l'adattatore grafico in modalita testo a 25×80 caratteri. Dalla libreria `libce` si è importata la variabile `video`, che rappresenta l'intero buffer da 4 KB di memoria video (2 byte per cella) a disposizione dall'adattotore. Le scritture vengono fatte, come avevamo detto, sovrascrivendo dati in tale buffer, mantenendo *lato software* un cursore che ci indica la posizione sullo schermo.

```
#include <libce.h>
// dimensioni memoria video (2K)
#define SIZE 4000

namespace vid {
  // dichiara l'array video di libce
  extern volatile natw* video;
}

#define video vid::video

char mess[] = "- x86 rules ";
int cursor = 0;

// attributi carattere: 0x00001111-ASCII-
// significa bianco su sfondo nero
natl attr = 0x0F00;

// stampa una stringa
void prt_string(char* mess) {
  while(*mess != '\0') {
    video[cursor] = *mess | attr;
    cursor = (cursor + 1) % SIZE;
    mess++;
  }
}

void main() {
  for(int i = 0; i < 167; i++) {
    prt_string(mess);
  }

  do {
    // esci su ESC
    natb k = kbd::get_code();
    if(k == 0x01) break;
  } while(true);
}
```

3.2 Timer

Il timer è realizzato come un interfaccia ad eventi, che riceve in ingresso un clock e aggiorna ciclicamente un registro contatore. Al raggiungimento di 0 da parte del contatore, si resetta e si invia un certo evento (un impulso).

Nel PC IBM in particolare troviamo 3 contatori:

- **Contatore 0:** è collegata al controllore delle interruzioni;
- **Contatore 1:** era storicamente usato per il refresh della RAM, oggi non viene più usato;
- **Contatore 2:** era collegato all'unico dispositivo audio presente sull'IBM, cioè il beeper speaker.

Le porte del timer sono le seguenti:

0x40	Timer 0 (registro contatore)
0x41	Timer 1 //
0x42	Timer 2 //
0x43	**CWR**, *Command Word Register*
...	
0x61	**SPR**, *Speaker Register*

Il timer dispone di una vasta gamma di parametri e di solo 2 piedini di indirizzo, quindi 4 locazioni nello spazio di I/O (il controllo speaker si considera a parte), che sono **CWR**, un registro comune di configurazione delle interfacce, più 4 registri per timer collegati ad ognuna delle 3 porte rimanenti. L'accesso ai registri di un singolo timer va quindi fatto, cosa interessante del chip, in serie, modificando la stessa porta 1 o 2 volte consecutivamente.

3.2.1 Sonoro

Vediamo in particolare il lato sonoro del PC IBM. Essendo stato questo un calcolatore pensato per l'uso da ufficio, le capabilità audio erano molto limitata: si disponeva di un beeper speeker a frequenza modulabile dal timer (contatore 2). Inoltre, un particolare registro in memoria era collegato direttamente in AND con l'uscita del contatore 2, permettendo la modulazione on/off del segnale allo speaker. La modulazione in volume del pc speaker, in particolare, viene fatta attraverso un ulteriore registro detto **SPR**.

Questo tipo di modulazione permetteva effettivamente di sfruttare, in maniera non prevista dalla IBM, per riprodurre segnali generici.

Mostriamo un esempio di un programma per la riproduzione di un brano musicale monofonico, inteso come una sequenza di note:

```
#include "song.h" // include libce
#define TIMESTEP 3

// indirizzi timer
const ioaddr timer0_addr = 0x40;
const ioaddr timer2_addr = 0x42;
const ioaddr cwr_addr = 0x43;
const ioaddr spr_addr = 0x61;

// frame corrente di esecuzione
```

```
natl frame = 0;

// imposta il divisore di un timer
void set_divisor(natw divisor, ioaddr addr) {
  // metti divisor in addr in 2 passate
  outputb(divisor, addr);
  outputb(divisor >> 8, addr);
}

// leggi il conteggio di un timer
natl read_timer(ioaddr addr) {
  // comando di latch
  outputb(0x00, cwr_addr);

  natb low = inputb(addr);
  natb high = inputb(addr);
  return (high << 8) | low;
}

// abilita lo speaker
void note_on() {
  outputb(3, spr_addr);
}

// disattiva lo speaker
void note_off() {
  outputb(0, spr_addr);
}

bool update_song() {
  song_frame cur_frame = song[frame];

  switch(cur_frame.mode) {
    case 0:
      // off
      note_off();
      break;
    case 1: {
      // on
      note_on();

      // divisore della nota
      natl divisor = cur_frame.get_divisor();
      set_divisor(divisor, timer2_addr);
      break;
    }
    case 2: {
      // legato
      natl divisor = cur_frame.get_divisor();
      set_divisor(divisor, timer2_addr);

      break;
    }
  }

  frame++;

  if(frame == length + 1) return false;
  return true;
}
```

```
71
72 void main() {
73   // imposta il timer 0
74   // outputb(0x36, cwr_addr); // modo 3
75   natl div = 0; // 0 significa 65536
76   set_divisor(div, timer0_addr);
77
78   // imposta il timer 2
79   outputb(0xB6, cwr_addr); // modo 3
80
81   // i tick svolti
82   natl tick = 0;
83   natl next_song_update = 0;
84
85   natl last_value = read_timer(timer0_addr);
86
87   while (true) {
88     natl current_value = read_timer(timer0_addr);
89
90     // se il valore corrente e' maggiore del valore precedente
91     // si e' fatto un salto
92     if (current_value > last_value) {
93       tick++;
94     }
95
96     // aggiorna se necessario
97     if(tick > next_song_update) {
98       bool res = update_song();
99       if(!res) break;
100
101      // imposta il prossimo aggiornamento
102      next_song_update += TIMESTEP;
103    }
104
105    last_value = current_value;
106  }
107 }
```

In particolare, il programma si basa sullo scorrimento di un array di strutture `song_frame` (definite in `song.h`, la cui struttura non ci interessa) ad intervalli regolari. Questi intervalli vengono ottenuti configurando il timer 0 per oscillare in modalità 2 (oscillazione a onda quadra) con un divisore di 65536, che per un clock a 1.19 MHz fa:

$$\frac{1.19\,\text{MHz}}{65536} \approx 17.158\,\text{Hz}$$

Lato software, si aspettano 3 di queste oscillazioni, per ottenere una frequenza di $\approx$ 6Hz (abbastanza vicina a quella delle crome a 120 BPM).

Una volta ottenuto il frame, quindi, si aggiorna il timer 2 di conseguenza, sfruttando il registro SPR per il volume della nota (qui solo on/off) e il suo registro di scrittura per la nota stessa.

4 Lezione del 04-03-25

4.1 Hard disk

Gli **hard disk** (*dischi rigidi*) sono effettivamente, seppur memorie, **periferiche**, collegate al bus attraverso la loro interfaccia. La CPU non puo' eseguire programmi direttamente

dall'hard disk, ma deve prima caricarli in memoria principale (memoria RAM).

Questo perchè letture e scritture in hard disk vengono effettuate per **blocchi** (storicamente di 512 byte), e richiedono molto più tempo di quanto sia possibile aspettare al prelievo di istruzioni o operandi.

Dal punto di vista elettromeccanico venivano realizzati attraverso dischi di materiale ferromagnetico imperniati ad un asse centrale, con testine mobili che scandivano il raggio dei dischi, rilevando o modificando la loro magnetizzazione per accedere all'informazione. Il complesso di dischi e testine viene detto **drive**.

L'informazione viene disposta su ogni disco in **settori** e **tracce**. Le tracce sono concentriche e i settori formano degli "spicchi" di ogni faccia. Notiamo che entrambe le facce di ogni disco possono memorizzare informazione. Un **blocco** è quindi formato dalla regione di una traccia compresa in un certo sensore.

I dischi vengono tenuti continuamente in rotazione (negli ordini delle centinaia/migliaia di RPM). Il tempo che la testina impiega a raggiungere una tracca viene detto **tempo di seek**, t_{seek}, il tempo che alla velocità di rotazione del disco l'informazione si trovi sotto la testina **latenza** $t_{latency}$ e il tempo necessario ad effettuare l'operazione vera e propria **tempo di lettura/scrittura** $t_{r/w}$, per cui il tempo di lettura/scrittura complessivo risulta:

$$t_{seek} + t_{latency} + t_{r/w} \sim 1\,\text{ms}$$

nell'ordine del millisecondo, per la CPU estremamente (milioni di volte) più lento della RAM.

Quello che accade al tempo di lettura è che il blocco viene copiato in un buffer di memoria nell'interfaccia che viene poi reso disponibile alla CPU. Viceversa, al tempo di scrittura il buffer viene riempito dalla CPU, e l'interfaccia si occupa poi di copiarlo all'interno del settore giusto.

Per effettuare un operazione dobbiamo quindi sapere:

- Quale *testina* individuare;
- Quale *traccia* individuare;
- Quale *regione* (quindi quale *blocco*) individuare.

Storicamente queste informazioni erano gestite lato software, concedendo la possibilità di alterare la *formattazione* del disco. Oggi la formattazione è definita in fabbrica, e l'interfaccia offre una sua astrazione. In questa astrazione ogni blocco è quindi indirizzato da un indirizzo logico, il **Logical Block Address**, **LBA**.

4.1.1 Interfaccia ATA

Nello standard PC AT gli hard disk usano interfacce **ATA** (capaci di gestire 2 drive, in configurazione *master/slave*). L'interfaccia ATA è dotata di diversi registri a 8 bit e uno a 16 bit:

- **Registri di selezione** del blocco:
  - **SNR** (Sector Number);
  - **CNL** (Cylinder Number Low);
  - **CNH** (Cylinder Number High);

  - **HND** (Head And Drive): solo gli ultimi 4 bit di questo registro formano l'informazione sulla testina da utilizzare. Gli altri bit vengono usati diversamente, ad esempio per selezionare quale drive usare in configurazioni master/slave, o per abilitare il LBA, usando quindi i registri di selezione per specificare un indirizzo logico (su $3 \cdot 8 = 4 = 28$ bit) anzichè un informazione geometrica sulla posizione del blocco desiderato.

  Vediamo che dalla dimensione dell'LBA (assumiamo che per indirizzamento geometrico si trova la stessa cosa) si ha una dimensione del disco:

$$2^{28} \cdot 2^9 = 2^{37} = 128\,\text{GB}$$

  Per questo si puo' abilitare la modalità **LBA48** (che non è un gruppo di idol giapponesi), dove ci si aspetta che l'LBA venga specificato in due passate, una da 24 bit e una da 20 bit sugli stessi registri.

- **SCR** (Section Counter): permette di specificare su quanti settori contigui a partire da quello specificato prima eseguire l'operazione;
- **BR** (Buffer Register): l'unico registro a 16 bit, permette di accedere al buffer 2 byte alla volta;
- **STS** (Status Register): il classico registro di stato che ci notifica se un'operazione è conclusa o si puo' effettuare;
- **CMD** (Command): serve a specificare l'operazione da effettuare (lettura, scrittura, ecc...).

Questi registri sono disposti in memoria come segue:

0x01f0	**BR**, *Buffer Register*
0x01f1	**ERR**, *Error Register*
0x01f2	**SCR**, *Section Counter*
0x01f3	**SNR**, *Sector Number*
0x01f4	**CNL**, *Cylinder Number Low*
0x01f5	**CNH**, *Cylinder Number High*
0x01f6	**HND**, *Head And Drive*
0x01f7	**CMD**, *Command Register*
0x01f8	**STS**, *Status Register*

Vediamo quindi un ultimo programma di esempio delle periferiche, che permette di scrivere un buffer di caratteri da 512 byte, stampandolo a schermo, e scriverlo/leggerlo su un settore di memoria ad un indirizzo LBA (prendiamo 1).

```
#include <libce.h>
#include "keyboard.h"
#include "video.h" // definisce il buffer video
#define BUF_SIZE 512

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

```
11
12 // registri indirizzo LBA (sarebbero SNR CNL CNH HND)
13 const ioaddr disk_lba0 = 0x01F3;
14 const ioaddr disk_lba1 = 0x01F4;
15 const ioaddr disk_lba2 = 0x01F5;
16 const ioaddr disk_lba3 = 0x01F6;
17
18 // dai indirizzo LBA al controllore disco
19 void give_lba(natl lba) {
20   // dividi in 4 byte
21   natb lba0 = lba;
22   natb lba1 = lba << 8;
23   natb lba2 = lba << 16;
24   natb lba3 = lba << 24;
25
26   // il byte piu' significativo deve attivare l'LBA,
27   // lba stava comunque su 28 bit
28   lba3 = (lba3 & 0x0F) | 0xE0; // 1110-LBA-
29
30   outputb(lba0, disk_lba0);
31   outputb(lba1, disk_lba1);
32   outputb(lba2, disk_lba2);
33   outputb(lba3, disk_lba3);
34 }
35
36 // dai comando al controllore disco
37 void give_command(natl lba, natb sectors, natb cmd) {
38   give_lba(lba);
39   outputb(sectors, disk_sectors);
40   outputb(cmd, disk_command);
41 }
42
43 // aspetta il disco
44 void wait_for_disk() {
45   natb s;
46   do {
47     s = inputb(disk_status);
48   } while ((s & 0x88) != 0x08);
49 }
50
51 // scrivi un settore sul disco
52 void write_sector(natb* sector) {
53   wait_for_disk();
54
55   // reinterpret_cast per mandare 2 byte per volta (ripetuti a 256 * 2 =
     512)
56   outputbw(reinterpret_cast<natw*>(sector), 256, disk_buffer);
57 }
58
59 // leggi un settore dal disco
60 void read_sector(natb* sector) {
61   wait_for_disk();
62
63   // come sopra
64   inputbw(disk_buffer, reinterpret_cast<natw*>(sector), 256);
65 }
66
67 // make code salva (1) e carica (2)
68 const natb save_code = 0x02;
69 const natb load_code = 0x03;
```

```
70
71 // indirizzo lba disco
72 natl lba = 1;
73
74 // buffer testo
75 natb buffer[BUF_SIZE];
76
77 // svuota il buffer
78 void init_buffer() {
79   for(int i = 0; i < BUF_SIZE; i++) {
80     buffer[i] = 0x00;
81   }
82 }
83
84 // cursore buffer testo
85 natl cursor = 0;
86
87 // sposta il cursore senza uscire dal buffer
88 inline void mov_cursor(int d) {
89   if(cursor == 0 && d < 0) return;
90
91   cursor += d;
92   if(cursor >= BUF_SIZE) cursor = BUF_SIZE - 1;
93 }
94
95 // salva buffer testo
96 void save() {
97   give_command(lba, 1, hd::WRITE_SECT);
98   write_sector(buffer);
99 }
100
101 // carica buffer testo
102 void load() {
103   give_command(lba, 1, hd::READ_SECT);
104   read_sector(buffer);
105 }
106
107 void main() {
108   // inizia svuotando il buffer
109   init_buffer();
110
111   // vai in un ciclo di lettura
112   while(true) {
113     natb make_code = get_key();
114
115     if(make_code == esc_code) break;
116     if(make_code == back_code) mov_cursor(-1);
117
118     if(make_code == save_code) save();
119     if(make_code == load_code) load();
120
121     char c = get_char(make_code);
122     if(c != '\0') {
123       buffer[cursor] = c;
124       mov_cursor(1);
125     }
126
127     // aggiorna schermo
128     prt_screen(buffer, BUF_SIZE);
129     set_cursor(cursor);
```

```
130     }
131 }
```

Gli header `keyboard.h` e `video.h` contengono funzioni simili a quelle viste negli esempi precedenti per l'interfacciamento con tastiera e video (ci sono due funzioni video non viste, `prt_screen()` per la scrittura di tutto il buffer video, e `set_cursor()`, che imposta la posizione del cursore hardware agendo su registri specifici).

La scrittura viene effettuata alla pressione del tasto `"1"`, e la lettura alla pressione del tasto `"2"`. Entrambe le operazioni si riassumono fondamentalmente nell'invio di un comando (`give_command()`), che include la scrittura dell'indirizzo LBA (`give_lba()`), e nella successiva scrittura o lettura di un settore (`write_sector()` o `read_sector()`), che comprende di aspettare un certo bit di stato del disco (`wait_for_disk()`). Il bit particolare si può verificare consultando i manuali appositi.

Le funzioni viste finora su periferiche di I/O sono disponibili nella cartella `/code` degli appunti del corso, assieme a vari esperimenti e la definizione completa di tutti gli header usati. Si noti che questi non sempre corrispondono con `libce`, ma spesso riprendono, ridefiniscono o usano (probabilmente in maniera erronea) funzioni e oggetti ivi definiti.

4.2 Caching

Abbiamo detto che la memoria RAM è molto più veloce dei dischi rigidi. Questo è vero, ma non significa che non ci sia comunque un certo dislivello tra la velocità della CPU e la velocità della RAM: un operazione puo' comunque richiedere un tempo nell'ordine dei ~ 100 circa cicli di clock.

Per questo motivo si inframezzano fra la CPU e la RAM più memorie, relativamente piccole ma veloci, dette **memorie di cache**. L'idea è che la RAM in sè è costituita da memoria dinamica (DRAM), quindi a condensatori, relativamente lenta e con tempo di refresh, mentre le memorie di cache vengono implementate con memorie statiche, più veloci ma più costose da realizzare su larga scala (per cui le dimensioni ridotte).

Vediamo che ci sono due modi principali di organizzare queste memorie: in *grandezza* delle singole memorie di cache, o in *distanza* dal processore, implementando una gerarchia di memorie sempre più grandi allontanandosi dal processore e avvicinandosi alla RAM. Vedremo nel dettaglio solo il primo metodo, introducendo le **cache ad indirizzamento diretto** e le **cache associative ad insiemi**, mentre ci limiteremo solo ad accennare al secondo metodo.

4.2.1 Principi di località

Le piccole dimensioni delle memorie vengono aiutate dalla **località** del codice in memoria: istruzioni che compongono le stesse funzioni avranno istruzioni vicine fra di loro, le strutture definite dal programmatore conterranno dati locali, ecc... In particolare, potremo distinguere fra due **principi di località**:

- **Località temporale:** una volta visto un indirizzo, è probabile che questo o indirizzi ad esso vicini siano visti di nuovo;
- **Località spaziale:** solitamente si accede ad indirizzi vicini fra di loro.

La cache avrà quindi il compito di memoizzare i valori prelevati con frequenza dalla DRAM. Possiamo immaginare che la prima lettura di un dato richiederà il tempo com-

pleto di accesso, ma la lettura successiva, ammesso che quel dato sia stato salvato nella cache, richiederà un tempo di accesso significativamente minore.

L'importante è che questo processo sia **trasparente** per la CPU, cioè che questa non si debba preoccupare di quali indirizzi sono stati visti dalla cache e memoizzati e quali no. Il risultato finale è la velocizzazione di un qualsiasi programma senza dover agire in nessun modo sul programma stesso. Di contro, non è detto che il programmatore non possa sfruttare la presenza della memoria cache, cercando di sviluppare algoritmi e strutture dati che rispettano il più possibile i principi di località (tecniche *data driven*).

4.2.2 Cache ad indirizzamento diretto

Vediamo un primo esempio di memoria cache. Abbiamo che lato processore ci arriveranno le linee di byte enable (BE) e le linee di indirizzo (A). Inoltre avremo a disposizione un bus dati (D) di un certo numero di linee.

Vorremo porre fra CPU e DRAM una cache, connessa a quest'ultima dalle linee di indirizzo A. La memoria interna della cache, di dimensione complessiva 64 KB, sarà rappresentata da una serie di blocchi, o **cacheline** da 64 byte.

In fase di lettura, invece di leggere l'unica riga richiesta dal processore, si procederà alla lettura di un certo numero di righe (poniamo 8). Questo significa che per un tempo di lettura di riga di t, ci vorrà un tempo $\sim 8t$ (solitamente meno). La speranza è che queste righe verranno lette successivamente dal processore.

Inoltre, ad ogni blocco di memoria letto dalla cache si dovrà associare dell'informazione riguardo alla posizione in memoria: questa viene contenuta in un altra memoria, dette **memoria delle etichette**. E' quindi più conveniente leggere regioni relativamente più grandi di memoria, in modo da non sprecare *overhead* per piccole quantità di dati.

4.2.3 Principio di funzionamento

La divisione della DRAM sulle cacheline è quindi realizzata giocando sulle scomposizioni degli indirizzi. Si divide ogni indirizzo in tre parti:

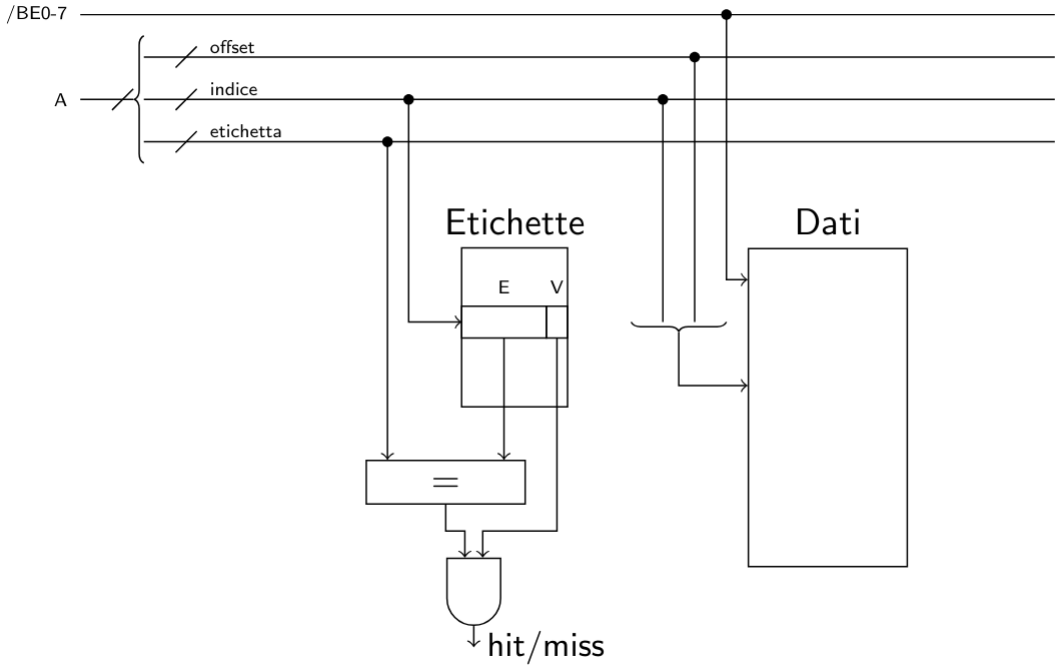
- L'**etichetta**, formata dai bit più significativi del bus;
- L'**indice**, formato dai 10 bit centrali (per indirizzare la totalità dei 64 KB di cache);
- L'**offset**, formato dai 3 bit meno significativi di A (per ottenere cache line da 64 byte, cioe $8 = 2^3$ parole quadruple da 8 byte.).

Noto l'offset, l'**indice** verrà calcolato per indirizzare la totalità delle cacheline come stante su un numero di linee tali a:

$$\text{bit}_{\text{indice}} = \frac{\text{dimensione cache}}{\text{dimensione cacheline}}$$

Per ottenere la regione corrispondente ad un indirizzo (il numero di cacheline) si realizza una sorta di *funzione di hash*, prendonendo l'etichetta e usandola come chiave per la regione di dati di indice corrispondente. Inoltre, alla regione selezionata si associa solitamente un singolo bit di validità. Un comparatore fra etichetta e gli n bit piu significativi messo in AND a questo bit di validità ci assicurerà quindi la presenza nella cacheline del dato richiesto, detta **hit/miss**.

La struttura complessiva è quindi la seguente:



4.2.4 Lettura

A questo punto, in fase di lettura, nel caso di hit basterà ricavare una linea di offset dai bit meno significativi di A, e leggere dalla memoria cache a tale offset, all'indice indicato dall'etichetta. Nel caso di miss si dovrà invece svolgere la lettura in memoria RAM, e poi riportare l'informazione nella cacheline di indice giusto della cache aggiornando l'etichetta.

4.2.5 Scrittura

Per quanto riguarda le scritture invece, potremo muoverci in due strade: **write allocate** e **write no allocate**.

- **Write allocate:** ci comportiamo in maniera simile alla lettura nel caso di hit. Nel caso di miss, invece, riportiamo il dato in cache.

  A questo punto potremmo pensare di svolgere la scrittura in RAM e in cache contemporaneamente (regola *write-through*), mantenendo entrambe aggiornate.

  Una tecnica più intelligente può invece essere quella di aggiornare il solo dato in cache, e rimandare la scrittura in RAM alla rimozione del dato dalla cache (per l'introduzione di un nuovo dato allo stesso indice) (regola *write-back*). In questo caso dovremo dotarci di un nuovo bit nella memoria delle etichette, il bit *dirty*, che segnalerà il bisogno di ricopiare il dato in cache nella RAM in occasione del suo deallocamento dalla cache. La difficoltà principale di questo metodo è l'avere un agente che non è la CPU che scrive in RAM, e come vedremo richiede soluzioni tecniche particolari.

- **Write no allocate:** in questo caso ignoriamo le scritture in cache e la sfruttiamo solamente per le letture.

Notiamo che questa cache soffre di problemi di **collisione**: infatti ci sarà un numero di regioni con lo stesso indice ed etichetta diversa, pari alla dimensione della RAM fratto la dimensione della cache.

5 Lezione del 07-03-25

Riprendiamo il discorso della memoria cache.

5.0.1 Cache e I/O

Avevamo che la memoria cache è montata fra la CPU e lo spazio di memoria: più propriamente, si trova fra la CPU e il bus. Può quindi vedere non solo le operazioni sulla memoria, ma anche sullo spazio di I/O. In questo caso, però, dovrà ovviamente comportarsi sempre in maniera *read-through* e *write-through*, quindi effettivamente disattivarsi e lasciare che il processore interagisca direttamente con l'I/O.

Questo è dovuto al fatto che allo spazio di I/O potrebbero accedere e modificare dati dispositivi esterni alla CPU (le interfacce), operazione che invaliderebbe immediatamente qualsiasi cosa venga scritta in memoria cache.

Inoltre, ogni operazione di lettura può comportare di per sé un aggiornamento delle interfacce, che comporterà un aggiornamento della memoria, motivo per cui un operazione di caching sarebbe superflua se non addirittura dannosa.

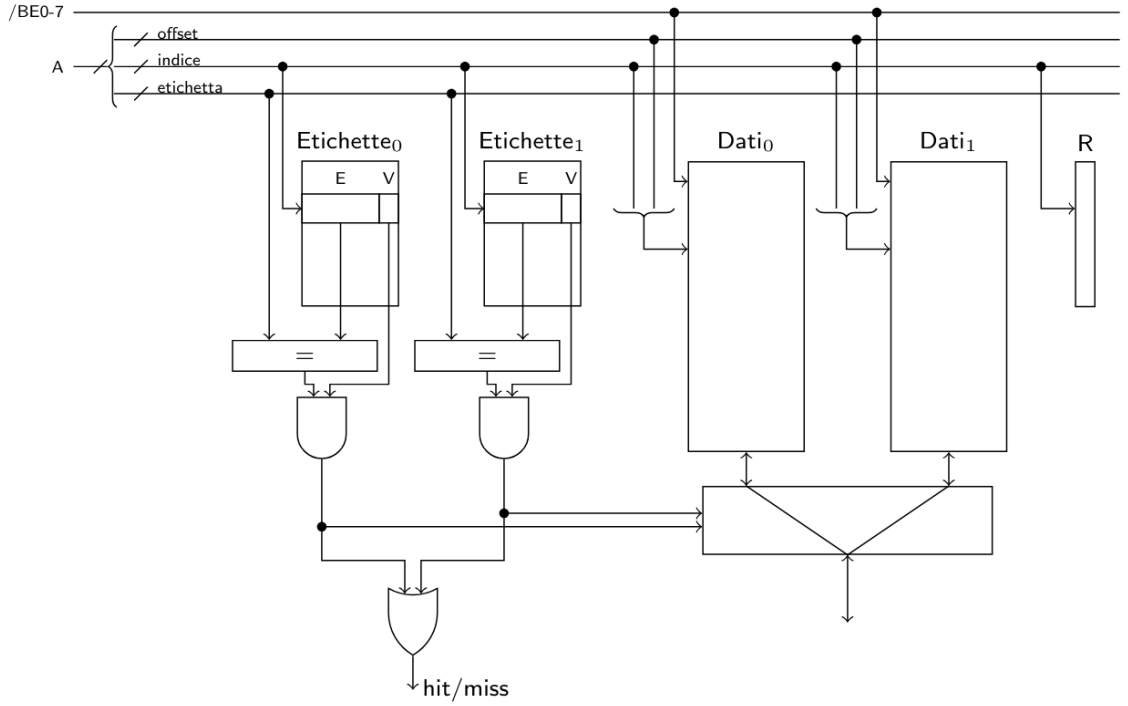
Operazione simile varrà effettuata per la memoria video (che non sta nello spazio di I/O). Questa facoltà verra realizzata dalla cache attraverso, probabilmente, *maschere* o *tabelle* che tengono conto di dove si trova la memoria video, e quindi quali richieste di lettura e scrittura vi hanno luogo.

5.0.2 Cache associative ad insiemi

Avevamo visto come il difetto principale della cache ad indirizzamento diretta è quello delle *collisioni*. Presentiamo un metodo, quello delle **cache associative ad insiemi**, che risolve il problema permettendo di allocare più cacheline allo stesso indirizzo.

Duplichiamo quindi la struttura vista per la cache ad indirizzamento diretto (qui solo 1 volta, anche se nei sistemi moderni si va dai 4-8 insiemi per le cache di primo livello e 8-16 insiemi per le cache di secondo e terzo livello), e sfruttiamo le uscite hit/miss delle singole memorie delle etichette per pilotare un multiplexer con in ingresso le linee dati delle memorie di cache corrispondenti.

La struttura così modificata sarà la seguente:



In questo caso a letture allo stesso indice le cache potranno rispondere diversamente (magari la prima in miss e la seconda in hit), e il processore vedrà ritornarsi il dato corretto (in questo caso quello della seconda).

Compito di scegliere quale cache sfruttare nel caso di collisioni è quello del **controllore** di cache (nella cache ad indirizzamento diretto non c'era scelta). La scelta migliore possibile sarebbe quella di scegliere la cacheline al cui i accederà più tardi nel futuro (per mantenere i dati immediatamente utili nella cache).

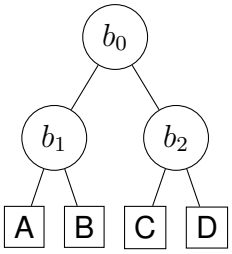
Chiaramente, visto che non si può prevedere il futuro (o almeno non lo possono fare né la CPU né il controllore di cache), occorre adottare un euristica. Una di queste euristiche è la politica **LRU** (*Least Recently Used*), dove si sceglie la cacheline al quale non si accede da più tempo.

Per realizzare tale politica si sfrutta una memoria, che chiamiamo R. Con solo due vie, basterà memorizzare su R l'ultima via usata, e quella su cui scrivere sarà immediatamente l'altra.

Con più di due vie sarebbe necessario mantenere l'ordine degli accessi, cioè per n vie ricordare informazione necessaria a controllare $n!$ diverse possibilità. Nella pratica, però, conviene usare politiche approssimate.

5.0.3 Pseudo-LRU dell'80486

Vediamo una di queste politiche approssimate, implementata nella cache del processore Intel 486, che gestiva 4 insiemi attraverso 3 bit b_0, b_1 e b_2. Si usava un albero binario per la selezione di una delle vie, disposto come:



dove i valori 1 sono i rami a destra, viceversa i valori 0 sono i rami a sinistra, e gli $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}$ rappresentano i 4 insiemi associativi.

In fase di rimpiazzamento, si sceglie la via seguendo l'albero. In fase di accesso, si modificano i b_i in modo da portare la via a cui si è fasso accesso in fondo all'ordinamento che si ottiene visitando l'albero. L'errore può essere dato dal fatto che la via che si trova nello stesso gruppo della via a cui si è fatto accesso potrebbe trovarsi ad un indice più alto del necessario, visto che si abbassa cumulativamente l'intero gruppo aggiornando b_0.

Ad oggi, anche per cache più grandi si sfruttano sempre algoritmi ad albero di questo tipo, magari tagliando i rami più bassi per lasciare spazio a scelte completamente casuali.

5.0.4 Cache ed accessi sequenziali

Notiamo poi che le memorie cache di questo tipo incontrano sempre difficoltà quando si fanno accessi ciclici ad indici che si ripetono con un modulo con il numero di vie diverso da zero: ad esempio se si leggono ciclicamente 5 indirizzi che corrispondono allo stesso indice, la cache non riuscirà mai a mantenere tutti e 5 in una delle cacheline delle vie, e quindi ogni accesso comporterà un miss.

5.0.5 Livelli di cache

Come abbiamo accennato, nei processori moderni si hanno solitamente più livelli di cache (3 o 4), che crescono in dimensioni e associatività più si vanno a disporre "lontano" dal processore e "vicini" alla RAM. Le cache di livello più basso saranno quindi più veloci ma più piccole, mentre le cache di livelo alto saranno più lente ma più grandi.

Il controllore di cache provvederà a gestire i livelli di cache, effettuando gli accessi controllando a partire dal livello più basso (più veloce) per arrivare al livello più basso, fino alla RAM.

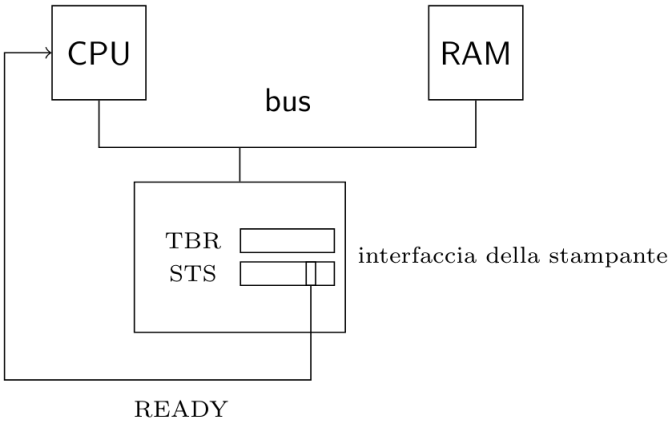
5.1 Interruzioni

La limitazione principale del processore studiato finora è che il flusso di controllo è completamente determinato dal programma in esecuzione. L'**interruzione** viene introdotta per allontanarsi da questo paradigma e introdurre nel processore la possibilità di gestire **eventi**. Infatti, attraverso il meccanismo dell'interruzione, il sistema definisce $e_1, ..., e_n$ di questi eventi, e il programmatore $r_1, ..., r_n$ **routine** per la loro gestione. Da qui in poi il processore continua ad eseguire il suo normale flusso di controllo, ma monitorando in

qualche modo lo stato di questi eventi. Nel caso uno degli eventi e_i effettivamente si verifichi, il processore provvederà a sospendere il flusso di controllo attuale e ad eseguire la routine r_i.

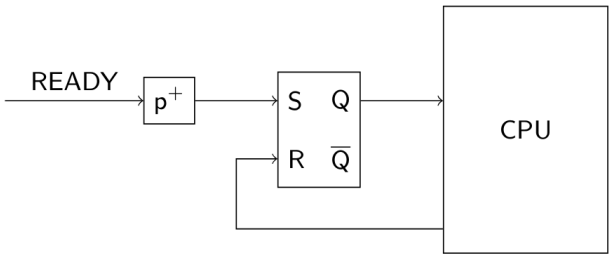
Un esempio classico dell'utilità di un meccanismo di questo tipo è dato dalle fasi di stampa che avevamo definito per dispositivi come schermi o stampanti: attraverso l'approccio visto finora dovremmo controllare periodicamente un certo registro di stato per verificare la possibilità di scrivere un nuovo dato in un certo registro di buffer. Questo occupa la CPU con operazioni inutili, che potrebbe saltare se fosse la stampante stessa ad avvertirla di quando è pronta a ricevere un nuovo dato.

L'idea di base è quella di avere una nuova operazione da svolgere in fase di esecuzione di un instruzione da parte della CPU, dopo l'esecuzione dell'istruzione stessa. Ad esempio, potremmo riportarci un bit di validità, `READY`, da parte della stampante, e controllarlo ad ogni istruzione per la chiamata di una routine di stampa, cioè aggiungere la seguente circuiteria:



La chiamata sarà semplicemente un aggiornamento condizionato a RIP, con scrittura del contenuto attuale di RIP in pila (che è compatibile con le regole di chiamata dei sottoprogrammi a cui siamo abituati).

Un problema di questo approccio potrebbe essere che, se il bit che segnala l'evento non si aggiorna immediatamente, la CPU andrà in un ciclo continuo di arresto dell'esecuzione e inizio di una routine. Una soluzione potrebbe essere dotare la CPU di una *rete di accettazione* della rihiesta: il bit di segnalazione dell'evento va in un generatore di impulsi che setta un SR flip-flop, cioè aggiungere la seguente circuiteria:



A questo punto la CPU può rispondere (livello hardware, nella nuova fase di esecuzione appena descritta) con un segnale di reset nel momento in cui riesce a rilevare l'evento e spostarsi nella routine.

In verità la situazione è più complicata: ad esempio potremmo voler ignorare nuovi eventi quando stiamo già cercando di soddisfarne uno. Per questo i processori x86 prevedono un apposito flag, il flag **IF** (*Interruption Flag*), che determina se le nuove interruzioni dovranno essere soddisfatte o meno. Il processore può essere quindi configurato per attivare automaticamente il flag IF in fase di risposta ad una richiesta di interruzione. Per effettuare il corretto ritorno, si usa la funzione `IRETQ`, che ripristina, oltre ad altre cose, lo stato dei flag (che era stato salvato in pila).

Inoltre, probabilmente vorremo gestire più di un interruzione. Per fare ciò, il processore supporta più tipi di interruzioni, codificati su 8 bit (per un totale di 256 tipi di interruzione). Vedremo che i primi 32 di questi sono riservati alle cosiddette *eccezioni*, mentre i successivi 224 sono disponibili al programmatore. Il codice da mandare in esecuzione (nel caso più banale, il semplice valore di RIP da inserire) per ogni interruzione viene mantenuto in una certa struttura dati in memoria, che viene detta **IVT**, *Interrupt Vector Table*, in modalità reale, e **IDT**, *Interrupt Descriptor Table*, in modalità protetta.

Abbiamo quindi che il meccanismo delle interruzioni, per citare Dijkstra, *"apre un vaso di pandora"* all'interno dell'architettura dei calcolatori. Infatti, la loro utilità ha fatto sì che esse non venissero usate solo per l'I/O, ma anche per altre applicazioni. Possiamo infatti distinguere i tipi di interruzione:

- **Esterne**, quelle introdotte adesso, che come vedremo si dividono a loro volta in *mascherabili* e *non mascherabili*;
- **Software**, lanciate dal processore stesso attraverso l'istruzione `INT`;
- **Interne** (*eccezioni*), lanciate sempre dal processorein caso di errori ad esso interni.

Vediamo l'esempio basilare di un'interruzione software (per noi analoga, almeno per adesso, alle altre 2 in funzionamento), per osservare il funzionamento del meccanismo di interruzione sopratutto riguardo alla pila. Scriviamo un breve programma misto C++/assembly, di cui la parte C++:

```
#include <libce.h>

extern "C" void a_handler();
extern "C" void c_handler() {
  printf("sto gestendo l'interruzione\n");
}

void main() {
  // imposta il gestore
  // extern "C" void gate_init(natb num, void routine(), bool trap, int
    liv)
  gate_init(0x40, a_handler, false, 3);

  // chiama l'interruzione interna
  asm("int $0x40");
  pause();

  return;
}
```

e la parte assembly:

```
#include <libce.h>

.global a_handler
.extern c_handler
```

```
5
6  a_handler:
7    salva_registri
8    call c_handler
9    carica_registri
10   iretq
```

Questo tipo di programmazione mista si rende necessario in quanto la gestione dell'interruzione richiede meccanismi accessibili solo all'assembly (in particolare la `IRETQ`, che serve per ritornare dal gestore di interruzione).

Ignoriamo quindi per adesso la riga 11 della funzione `main()`, che ha il solo compito di impostare il gestore di interruzione per l'interruzione `0x40`, e vediamo cosa accade alla riga `14`, dove si solleva effettivamente l'interruzione. In questo caso, il processore sospende l'esecuzione del programma e salta alla funzione `a_handler()`, che provvede a salvare i registri, stampare un messaggio, ricaricare i registri e ritornare. Il contenuto della pila al momento della chiamata di `a_handler()` sarà il seguente:

`0x20a208`	RIP (main alla riga 15)
`0x20a210`	Riservato (CS, per adesso non significativo)
`0x20a218`	RFLAGS

Quindi, come avevamo introdotto, si salva il registro dei flag, un valore per adesso non significativo, e il puntatore corrente (in particolare, quello all'istruzione successiva alla `INT`, vedremo significa che si trattava di un interruzione di tipo *trap*).

L'esecuzione del programma dà quindi complessivamente il seguente output:

```
>> sto gestendo l'interruzione
>> Premere ESC per proseguire
```

cioè siamo riusciti ad interrompere il normale flusso di controllo della funzione `main()`, interrompendola a metà esecuzione per eseguire un operazione secondaria, e restituendogli in seguito il normale controllo.

6 Lezione del 10-03-25

Torniamo sull'argomento delle interruzioni, specificando il modo in cui dobbiamo definire dei *gestori* per ogni interruzione.

Il calcolatore visto finora dispone di 4 interfacce:

- L'interfaccia *tastiera*, letta finora in controllo di programma, valutando la validità di un bit FI sul registro di stato;
- L'interfaccia *timer*, dotata di 3 singoli timer, di cui abbiamo detto il primo viene usato per generare interruzioni, il secondo non è più usato, e il terzo e connesso al *beeper speaker*;
- L'interfaccia a blocchi per *hard disk*, che accede ad un drive pilotando in base al suo stato un registro di stato (per noi era utile implementare la funzione di attesa del drive a controllo programma `wait_for_br()`).

Ignoriamo, per adesso, il video. Ognuna di queste interfacce può trarre beneficio dalla presenza di interruzioni:

- La tastiera potrebbe *avvertirci* dei nuovi tasti premuti, anziché costringerci a controllare;

- Il timer ci deve avvisare, al termine del conteggio del timer 0, attraverso un interruzione;
- L'hard disk, come la tastiera, ci può avvisare con un interruzione quando è pronto ad una nuova scrittura.

Questo comportamento, delle cosiddette **interruzioni esterne**, è definito nella macchina studiata dal **controllore delle interruzioni**, che è l'Intel **APIC** (*Advanced Progammable Interruption Controller*). Questo scansiona periodicamente tutte le linee di richiesta d'interruzione (le **IRQ**) ottenute dalle varie interfacce, e invia le interruzioni corrispondenti, una per volta, alla CPU.

Avevamo già reputato necessario specificare un **tipo di interruzione**, su 8 bit (per 256 tipi) per ogni interruzione lanciata. L'APIC, allora, fornirà semplicemente la possibilità di assegnare un tipo di interruzione diverso ad ogni piedino di ingresso dall interfacce, in modo che si possa assegnare ad ogni interruzione la routine di gestione più adatta. In questo, la configurazione dell'APIC si svolge come la configurazione di una qualsiasi periferica attraverso un'apposita interfaccia.

La comunicazione fra CPU e APIC in fase di interruzione viene effettuata attraverso un *handshake* su due linee, **INTR** (*Interrupt Request*) e **INTA** (*Interrupt Acknowledge*), che comporta anche una lettura da parte della CPU di quanto l'APIC metterà sul bus (cioè il tipo di interruzione).

A questo punto, le routine vere e proprie verrano definite nell'**IDT**, (*Interrupt Descriptor Table*), contenente in sequenza gli indirizzi delle prime istruzioni di ogni routine per ogni tipo di interruzione, e specificata a partire da un certo indirizzo indicato nel registro **IDTR**.

Come abbiamo visto, la reazione o meno della CPU ad una interuzione è data dall'attivazione del flag IF. Nel caso si passi effettivamente ad eseguire l'interruzione, ricordiamo che sia l'IP che lo stato dei flags verrà salvato in pila, e ripristinato a fine routine attraverso l'istruzione `IRET`.

6.0.1 Rilevamento di interruzioni da parte dell'APIC

Potremmo chiederci come fa il controllore APIC a capire quando un'interfaccia sta richiedendo una nuova richiesta.

Un primo approccio potrebbe essere di non rileggere il piedino di ingresso di quell'interfaccia, all'ottenimento e successivo invio alla CPU di un interruzione, fino alla segnalazione, sempre da parte della CPU, di avvenuta gestione dell'interruzione. Questo può essere effettuato dotando l'APIC di un opportuno registro (**EOI**, *End Of Interrupt*), che la CPU andrà a modificare conclusa la gestione dell'interruzione.

Un approccio più sicuro può essere ottenuto dotando il controllore delle interruzioni di due registri, entrambi su 256 bit (un bit per ogni tipo di interruzione):

- **IRR** (*Interrupt Request Register*): indica con bit alti quali interruzioni sono state inviate dalle interfacce attualmente;
- **ISR** (*Interrupt Service Register*): indica con bit alti a quali interruzioni sta rispondendo il processore attualmente. In un processore single-threaded come quello che studiamo al più uno solo dei suoi bit sarà alto in un dato momento (escluso il caso della *gestione annidata*).

Si avrà quindi la seguente organizzazione:

IRR, *Interrupt Request Register*	256 bit
ISR, *Interrupt Service Register*	256 bit

Un interruzione generata lato hardware si tradurrà nell'innalzamento (se non era già alto) di un bit nell'IRR, e l'inizio (o schedulazione dell'inizio) di un handshake con la CPU. Al termine dell'handshake (quindi all'abbassamento di INTA successivo ad un suo innalzamento per acknowledge) il bit dell'interruzione corrente passa dall'IRR all'ISR. Infine, la transizione dal bit presente nell'ISR all'interruzione gestita (bit nuovamente basso) si ha sempre con il segnale EOI da parte della CPU (con successivo inizio, se necessario, di un nuovo ciclo di handshake per una nuova richiesta di interruzione).

6.0.2 Priorità delle interruzioni e gestione annidata

Ci rendiamo quindi conto che alcune richieste sono più importanti di altre: ad esempio, la pressione di un tasto su tastiera può essere ignorata, se ad esempio nel frattempo arriva una richiesta di interruzione da parte di un timer. La pressione del tasto non si ripeterà infatti in tempo utile, mentre il timer potrebbe inviarci nuove richieste mentre ancora non siamo pronti a riceverle, e continuerà a farlo a scadenze regolari (potremmo finire per gestire solo un sottoinsieme delle richieste che ci vengono effettivamente inviate).

Possiamo quindi chiederci come l'APIC si comporta in caso di più richieste concorrenti. Un idea potrebbe essere di assegnare una priorità ad ogni richiesta, e rispondere prima alle richieste di priorità più alta. Questa priorità può essere implementata chiamando i 4 bit più significativi del tipo dell'interruzione **classe di precedenza** dell'interruzione: a classi di precedenza maggiore abbiamo gestione prioritaria delle richieste di interruzione. Il trasferimento da IRR a ISR avverrà quindi prima per richieste di classe di precedenza più alta, e poi per quelle di classe di precedenza *uguale* o più bassa, con la possibilità per le prime di *interrompere* i gestori di interruzione delle ultime.

La precedenza delle interruzioni è quindi necessaria all'implementazione corretta della **gestione annidata** delle interruzioni, dove un interruzione di precedenza più altra può interrompere (a patto che IF sia alto) un gestore di interruzione in esecuzione. Questo è il caso a cui accennavamo prima, dove più bit di ISR possono essere alti contemporaneamente (a patto che si dispongano nel tempo, da destra verso sinistra, cioè da minore priorità a maggiore priorità), e che si risolvano da sinistra verso destra (cioè da maggiore priorità a minore priorità).

7 Lezione del 11-03-25

Riprendiamo la trattazione dell controllore di interruzioni APIC.

7.0.1 Interruzione di livello o di fronte

Vediamo un dettaglio sul comportamento dell'APIC: questo può rilevare, in base alla sua configurazione, i **livelli** o i **fronti** delle variabili in ingresso.

Questo può avere delle implicazioni diverse a seconda dell'interfaccia. Ad esempio, avevamo detto che il timer in modalità 2 genera un onda quadra. Se si usa una routine lanciata dal timer a interruzione di programma, e si configura l'APIC per rilevare il

livello, potrebbe essere che a routine concluse il livello del timer è sempre alto, e quindi l'interruzione viene lanciata nuovamente.

Questo è chiaramente diverso dal comportamento desiderato, ed è quindi opportuno configurare l'APIC per rilevare i soli fronti di salita.

Abbiamo quindi notato praticamente tutte le caratteristiche che ci interessavano dell'APIC, e possiamo procedere ad implementare un esempio di gestione di un interfaccia a controllo di interruzione. Vediamo ad esempio il seguente programma, che gestisce la tastiera a controllo di interruzione, di cui la parte C++:

```
#include <libce.h>
#define KBD_VECT 0x20

bool fine = false;

extern "C" void a_keyboard();
extern "C" void c_keyboard() {
  // leggiamo da tastiera
  natb code = kbd::get_code();

  if(code == 0x01) fine = true;

  char c = kbd::conv(code);
  vid::char_write(c);

  apic::send_EOI();
}

void main() {
  // attiva le interruzioni tastiera
  kbd::enable_intr();

  // imposta l'APIC
  apic::set_VECT(1, KBD_VECT);
  apic::set_TRGM(1, false); // false: fronte, true: livello
  apic::set_MIRQ(1, false);

  // imposta il gate nella IDT
  gate_init(KBD_VECT, a_keyboard);

  while(!fine);

  return;
}
```

e la parte assembly:

```
#include <libce.h>

.global a_keyboard
.extern c_keyboard

a_keyboard:
  salva_registri
  call c_keyboard
  carica_registri
  iretq
```

Il meccanismo di chiamata dell'interruzione (macro per il salvataggio/caricamento registri, istruzione `iretq`, ecc...) è identico all'esempio precedente. Una novità è la presenza della funzione `send_EOI()` nel gestore di interruzione, che invia il segnale di

End Of Interrupt all'APIC e gli fa capire, assieme alla lettura che facciamo sulla tastiera (con `kbd::get_code()`) che l'interruzione è stata effettivamente gestita. Inoltre, la parte di configurazione dell'interruzione è più complessa. Bisogna infatti:

- Attivare le interruzioni da tastiera con `kbd::enable_intr();`
- Impostare l'APIC per inviare tali interruzini al tipo interruzione `0x20`, configurandolo per riconoscere fronti, e disattivando la maschera (rispettivamente `set_TRGM()` e `set_MIRQ`);
- Infine, inizializzare il gate corrispondente al tipo interruzione `0x20` come avevamo già visto.

Abbiamo quindi realizzato pienamente quanto ci eravamo posti di fare quando abbiamo iniziato a parlare di interruzione: la CPU è lasciata libera (nell'esempio specifico, esegue un loop infinito), e viene *interrotta* dalla periferica tastiera quando questa ha un nuovo dato disponibile. Vediamo che in verità esiste un altra casistica di applicazione delle interruzioni che non abbiamo trattato, cioè quella delle *eccezioni*.

7.1 Eccezioni

Ci rimangono da vedere le **eccezioni**. Queste sono particolari errori logici che il processore potrebbe incontrare nel corso dell'esecuzione, come ad esempio la divisione per 0, il tentativo di eseguire un istruzione non riconosciuta, ecc...

Una differenza fra le interruzioni esterne e le eccezioni è che le eccezioni possono essere sollevate *durante* la lettura e esecuzione di un istruzione, quindi ad esempio mentre si stava interpetando un codice operativo (si pensi all'interruzione di operazione non riconosciuta). In verità, per assicurare l'atomicità dei cicli di esecuzione, la CPU ripristina automaticamente il suo stato a prima del lancio dell'interruzione. In particolare, possiamo distinguere 3 tipi di eccezione:

- **Fault:** l'esecuzione non viene ancora eseguita, lo stato IP prima della sua esecuzione viene salvato (quindi si rimane alla stessa istruzione), e si può riprovare ad eseguirla dopo aver risolto l'errore;
- **Trap:** l'esecuzione ormai è stata eseguita, e si salva l'IP successivo.
- **Abort:** raggruppa degli eventi particolarmente disastrosi in cui l'esecuzione si arresta completamente (ad esempio la tripla eccezione).

Quando viene lanciata una *fault* o una *trap*, il processore cerca nella IDT se esiste un handler corrispondente (segnalato attraverso un bit nell'IDT stessa, alla riga della tabella corrispondente all'eccezione considerata). Nel caso questo non esista, si riprova con la fault di *doppia eccezione*, che quindi rappresenta una fault a sé. Nel caso nemmeno questo handler esista, viene lanciata una fault di *tripla eccezione*, che è di tipo *abort* e comporta quindi l'arresto del programma.

Vediamo quindi un programma di esempio delle eccezioni, che gestisce ad esempio la divisione per zero (tipo `0x00` nella IDT), di cui la parte C++:

```
#include <libce.h>

extern "C" void c_divzero(natq rip) {
  printf("E' successo qualcosa di brutto a %lx\n", rip);
```

```
}
extern "C" void a_divzero();

int main() {
  // imposta interruzione per fault divisione
  gate_init(0, a_divzero);

  volatile int a = 3;
  a /= 0; // il qualcosa di brutto

  return 0;
}
```

e la parte assembly:

```
.global a_divzero
a_divzero:
  // non abbiamo bisogno di salvare o caricare registr
  mov (%rsp), %rdi // restituisci IP
  call c_divzero
  iretq
```

Notiamo che questo è il primo esempio che vediamo di valore di ritorno dal gestore di eccezione: il valore di RIP al momento dell'interruzione, che viene passato nel registro `%RDI` (come definisce l'ABI System V).

7.1.1 Eccezioni e debug

Un interruzione particolare è quella rappresentata da `INT3`, l'interruzione di *debug*. Attraverso questa, un *debugger* è capace di interrompere l'esecuzione di un programma ad un certo indirizzo del suo codice macchina.

Un'altra interruzione di debug è data dalla single step, che viene lanciata ad ogni istruzione quando è attivo un certo flag (appunto, il flag single step). Questo permette al debugger di eseguire il programma in modalità *passo singolo*, cioè eseguendo un istruzione e interrompendo, permettendo al programmatore di osservare il suo andamento passo per passo.

Possiamo sfruttare queste interruzioni di debug per realizzare il meccanismo dei **breakpoint**, cioè per interrompere un programma arbitrario ad una sua istruzione qualsiasi, per poi riprendere l'esecuzione esattamente da tale istruzione. Vediamo due esempi specifici:

- **Breakpoint con la sola INT3**: prendiamo la seguente funzione C/C++:

  ```
  void foo(){
      printf("sono la funzione foo\n");
  }
  ```

  che disassembla in:

  ```
     0: 55                    push   %rbp                # prologo
     1: 48 89 e5              mov    %rsp,%rbp
     4: bf 00 00 00 00        mov    $0x0,%edi           # chiama printf
     9: b8 00 00 00 00        mov    $0x0,%eax
     e: e8 00 00 00 00        call   13 <_Z3foov+0x13>
    13: 5d                    pop    %rbp                # ritorna
    14: c3                    ret
  ```

L'obiettivo potrebbe essere quello di interrompere la funzione nella fase di prologo, cioè all'istruzione `PUSH %RBP` di codifica `55`. L'idea è quella di prendere tale istruzione, salvarla da qualche parte per poterla reintrodurre in seguito, e sostituirla con una `INT3`, in modo che si possa prestabilire un gestore dell'eccezione da questa lanciata che metta in pausa il programma e rimetta a posto il byte modificato. Potremo allora usare il seguente codice:

```
#include <libce.h>

// conterra' il bit da salvare
char saved_byte;

void foo() {
  printf("sono la funzione foo\n");
}

// questa funzione interrompe all'int3
extern "C" void a_debug();
extern "C" void c_debug(void** p) {
  // p contiene il puntatore a %rsp

  // mette in pausa
  pause();

  // poi rimette tutto a posto
  auto func_p = reinterpret_cast<char**>(p);
  --(*func_p); // e' l'istruzione precedente
  **func_p  = saved_byte;
}


// questa funzione mette il breakpoint
void add_breakpoint(void (*func)(void)) {
  // salva in saved_byte
  auto func_p = reinterpret_cast<char*>(func);
  saved_byte = *func_p;
  printf("saved_byte: %x\n", saved_byte);

  // al suo posto mette 0xcc (int3)
  *func_p = 0xcc;
}

extern "C" void main() {
  // inizializza il gate int3 con a_debug()
  gate_init(3, a_debug);

  // aggiungi il breakpoint
  add_breakpoint(foo);

  // qui arresta
  foo();
  // qui no
  foo();

  pause();
}
```

La variabile `saved_byte` conterrà il byte da reinserire dopo l'interruzione. La funzione `add_breakpoint()` si occuperà allora di salvare il byte giusto e sostituirlo con

l'istruzione `INT3` (codice `0xcc`). A questo punto la funzione `a_debug()`, che avrà il compito mettere a primo argomento (registro `RDI` secondo l'ABI di System V) il puntatore allo stack di chiamare la `c_debug()`, che metterà in pausa l'esecuzione, rimetterà a posto `saved_byte` e decrementerà l'instruction pointer (ricordiamo che la `INT3` è di tipo *fault*, quindi salva l'instruction pointer *dopo* l'ultima istruzione eseguita).

Il funzionamento della parte assembler, cioè della `a_debug()`, si riduce a:

```
#include "libce.h"

.global a_debug
.extern c_debug

a_debug: # handler interruzione int3
  salva_registri

  # passa il puntatore alla pila come primo argomento
  leaq 120(%rsp), %rdi
  call c_debug

  carica_registri
  iretq
```

dove l'offset di 120 è dato dai registri, salvati dalla macro `salva_registri`, che occupano 120 byte sullo stack.

- **Breakpoint con INT3 e single step**: un problema dell'esempio precedente è che, come si riporta anche nei commenti, dopo la prima interruzione non si interrompe più in quanto il contenuto del byte d'istruzione interessato viene ristabilito e non più toccato. Potremmo invece voler rimettere la `INT3` dopo la sua esecuzione, così da permettere interruzioni ogni volta che si torna sull'istruzione (che è come funzionano i breakpoint di programmi reali come *GDB*). Facciamo questo sfruttando la modalità *single-step*: al momento dell'interruzione, la attiviamo, facciamo un singolo passo e rimettiamo la `INT3` sull'istruzione. Per fare ciò, salviamo l'indirizzo dell'istruzione, che non è altro dell'indirizzo precedente all'instruction pointer corrente al momento della gestione dell'interruzione `INT3`. In codice abbiamo quindi:

```
#include <libce.h>

char saved_byte;
char* saved_byte_addr;

void foo() {
  printf("foo e' in esecuzione\n\n");
}

// questa funzione interrompe all'int3
extern "C" void a_debug();
extern "C" void c_debug(void** p) {
  // p contiene il puntatore a %rsp

  // *p e' %rip, decrementa (vogliamo ripartire da rip - 1, e questo
    sara'
  // l'indirizzo salvato nello stack che iretq andra' a riprenersi)
  (*p)--;

```

```
  // prende il vecchio %rip come indirizzo del byte salvato
  saved_byte_addr = reinterpret_cast<char*>(*p);

  pause();

  // poi rimettette tutto a posto
  *saved_byte_addr = saved_byte;
}

// questa funzione interrompe al single step
extern "C" void a_sstep();
extern "C" void c_sstep() {
  // saved_byte_conterra' il byte che abbiamo reinserito

  // rimette 0xcc (int3)
  *saved_byte_addr = 0xcc;
}

// questa funzione mette il breakpoint
void add_breakpoint(void (*func)(void)) {
  // salva in saved_byte
  auto func_p = reinterpret_cast<char*>(func);
  saved_byte = *func_p;
  printf("saved_byte: %x\n", saved_byte);

  // al suo posto mette 0xcc (int3)
  *func_p = 0xcc;
}

extern "C" void main() {
  // inizializza il gate int3 con a_debug()
  gate_init(3, a_debug);
  // inizializza il gate single_step() con a_sstep()
  gate_init(1, a_sstep);

  // aggiungi il breakpoint
  add_breakpoint(foo);

  // qui arresta
  foo();

  // qui pure
  foo();

  pause();
}
```

Le funzioni assembler (`a_debug()` e `a_sstep()`) modificano il registro `RFLAGS` per attivare e disattivare, rispettivamente, la modalità single-step, come segue:

```
#include "libce.h"

.global a_debug, a_sstep
.extern c_debug, c_sstep

a_debug: # handler interruzione int3
  salva_registri

  # passa il puntatore alla pila come primo argomento
  leaq 120(%rsp), %rdi
  call c_debug
```

```
12
13    carica_registri
14
15    orw $0x100, 16(%rsp) # attiva la single step in eflags
16                         # la pila e':
17                         # $rsp   vecchio rip
18                         # $rsp+8 vecchio cs
19                         # $rsp+16  vecchio rflags
20                         # a questo punto TF e' a 0x100 in rflags
21
22    iretq
23
24  a_sstep: # handler single step
25    salva_registri
26
27    call c_sstep
28
29    carica_registri
30
31    andw $0xFEFF, 16(%rsp) # disattiva la single step in eflags
32                           # come sopra, la maschera e' complementare
33
34    iretq
```

7.2 Riassunto sui tipi di interruzioni

Abbiamo quindi visto tutti i tipi di interruzione, di cui riportiamo una lista completa:

- **Interruzioni esterne:** causate da interfacce esterne e gestite dall'APIC I/O, di cui distinguiamo:
  - **Interruzioni esterne mascherabli:** quelle che abbiamo visto finora, relative a normali eventi I/O;
  - **Interruzioni esterne non mascherabili:** cioè che non possono essere mascherate, solitamente rappresentano eventi particolarmente gravi o comunque la cui gestione ha alta importanza.
- **Interruzioni interne** (*Eccezioni*): eventi che non arrivano dall'esterno, ma si generano all'interno del processore stesso;
- **Interruzioni software:** interruzioni che vengono lanciate direttamente dal programma attraverso l'istruzione `INT`, la cui utilità è stata per ora dimostrativa, e verrà inquadrata meglio studiando il meccanismo della *protezione*, e in generale lo sviluppo del sistema multiprogrammato e delle relative *primitive*.

8 Lezione del 17-03-25

8.1 Protezione

Tutti i programmi che abbiamo visto finora hanno il pieno controllo su la macchina su cui sono in esecuzione. Questo significa che possono impattare qualsiasi regione di memoria, incluso il loro stesso codice macchina, o i frame di stack di programmi lanciati prima di loro.

Un approccio di questo tipo non è ideale quando più programmi, magari di utenti diversi, vengono lanciati ed eseguiti *quasi* in contemporanea (*time-sharing*) sulla stessa macchina.

Un esempio di questa situazione può verificarsi nel caso di esecuziono *batch*, cioè di esecuzione successiva di più programmi, magari scritti da più utenti. Vorremmo massimizzare l'uso della CPU sospendendo un programma e iniziandone un altro nel caso il primo fra questi inizi un'operazione che richiede una quantità significativa di tempo (ad esempio un accesso a un dispositivo di I/O). In questo caso, visto che non possiamo fidarci della benevolenza degli utenti nell'inserire istruzioni esplicite per il cambio da un programma all'altro, vorremo agire sull'hardware per, ad esempio, vietare all'utente l'uso di certe istruzioni (qui `IN` e `OUT`) e costringerlo ad usare primitive messe a disposizione dal sistema.

Chiaramente, però, le primitive dovranno poter usare `IN` e `OUT` per fare l'I/O vero e proprio con i dispositivi. Per permettere questo doppio comportamento introduciamo l'idea di **protezione**.

8.1.1 Contesti di esecuzione

Il programma nella memoria potrà essere in esecuzione, in un momento qualsiasi, in uno di due **contesti**, o *modi* (vedremo nell'architettura x86 corrente, si parla di protezione a *ring*): il contesto **sistema** e il contesto **utente**. Le istruzioni di cui permetterà l'esecuzione saranno quindi determinate dal contesto corrente.

Forniamo allora il processore di un apposito registro, il **CS** (*Code Segment*), a 2 bit. I 2 bit sono necessari in quanto storicamente (il meccanismo descritto viene introdotto nell'architettura x86 a partire dal 286) si definivano quattro contesti, o **ring**:

CS	Ring	Tipo
00	Ring 0	Kernel (sistema)
01	Ring 1	Driver
10	Ring 2	/ /
11	Ring 3	Utente

Il nome CS deriva dal fatto che questo registro era pensato per gestire la *segmentazione* della memoria. Sia questo meccanismo, che i due ring interni (l'1 e il 2) sono pressoché inutilizzati nell'architettura x86-64 moderna, e quindi li ignoreremo, portandoci effettivamente alla situazione dove CS rappresenta un flag che distingue fra contesto *sistema* e contesto *utente*, come avevamo ipotizzato.

8.1.2 Transizioni fra contesti

Ipotizziamo quindi che all'avvio si parta in contesto sistema, e che si passi al contesto utente quando si esegue un programma utente, Per permettere all'utente di "accedere" alle istruzioni privilegiate, vogliamo che questo disponga di un modo di tornare al contesto sistema, ma lasciando il controllo al sistema operativo (altrimenti sarebbe inutile introdurre l'idea di un contesto utente in primo luogo). Di contro, vogliamo un modo per il sistema operativo di restituire in sicurezza il controllo al programma, previa transizione del processore in contesto utente.

Vediamo come il meccanismo dell'interruzione fornisce un metodo per gestire questa situazione.

Introdurremo un tipo di interruzione apposito, che restituisce il controllo al sistema operativo (semplicemente passando ad un gestore di interruzione definito dal sistema operativo) passando a contesto sistema. Il tipo di operazione che stiamo richedendo al sistema operativo potrà essere passato in qualche registro specifico, solitamente `%EAX`. Il problema potrebbe essere chiaramente che l'utente ha la possibilità di modificare tutta la memoria, e quindi la stessa IDT e il gestore impostato.

8.1.3 Protezione di memoria

Si rende quindi necessario un meccanismo di gestione degli accessi in memoria. In contesto utente, quindi, oltre a permettere l'utilizzo di solo alcune istruzioni *non privilegiate*, il processore dovrà permettere l'accesso solo a determinate regioni di memoria. Visto che non abbiamo ancora introdotto l'idea di *memoria virtuale*, modellizziamo temporaneamente questa configurazione con un apposito registro a controllo sistema che decide quali regioni di memoria sono o non sono accessibili.

Abbiamo quindi l'immagine completa del meccanismo della protezione, che avevamo introdotto per privilegiare le sole istruzioni, ma ci rendiamo adesso conto deve consistere in:

- Protezione delle **istruzioni** attraverso il loro privilegiamento al contesto sistema, come avevamo visto;
- Protezione della **memoria** definendo regioni accessibili in sola modalità sistema.

8.1.4 Transizione da contesto utente a contesto sistema

Vediamo nel dettaglio come si passa dal contesto utente al contesto sistema. Per questo sfrutteremo l'istruzione x86 `INT`, che permette di generare un interruzione software sulla base del tipo fornito come operando. Si potrà quindi implementare il meccanismo della *chiamata a sistemo*, secondo una modalità del tipo:

```
mov $0x00, %eax # tipo chiamata
int $0x80       # chiamata sistema (per x86, in x86-64 esiste syscall)
```

Notiamo che questo è l'approccio normalmente supportato dai moderni sistemi operativi x86 (specialmente Linux, anche se oggi si usa l'istruzione apposita `syscall`). Vedremo che il passaggio del tipo chiamata nel registro `%EAX` non si verifica nel kernel che studieremo, dove invece ognuno dei 224 tipi di interruzioni liberi potrà rappresentare una chiamata a sistema diversa.

Questo si tradurrà a livello processore nel salvataggio dello stato corrente di esecuzione, la transizione al contesto sistema e lo spostamento in IP della prima istruzione di un apposito sottoprogramma di servizio atto a gestire l'eccezione (e quindi soddisfare, se possibile, la richiesta del programma per cui questo ha sollevato in primo luogo l'interruzione).

Per capire nel dettaglio cosa accade nel processore è necessario:

- Capire come è strutturata la Interrupt Descriptor Table (IDT) all'interno della memoria del sistema, che supponiamo essere privilegiata (altrimenti l'utente potrebbe manometterla);
- Capire come viene gestita un interruzione software, cioè come si conserva lo stato al momento dell'interruzione, e come si inizia l'esecuzione del gestore in contesto sistema.

Vediamo questi dettagli in ordine.

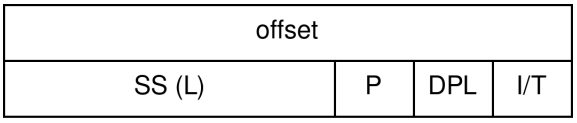
8.1.5 Struttura della IDT

Vediamo quindi nel dettaglio la struttura di un entrata della IDT. Questa viene a trovarsi nella memoria privilegiata a partire da un indirizzo, come avevamo detto, contenuto nel registro IDTR. L'impostazione di questo registro si fa attraverso apposite istruzioni, sempre ad accesso privilegiato.

Le entrate dell'IDT si chiamano **gate IDT**, che si distinguono in 3 tipi, *Task Gate*, *Interrupt Gate* e *Trap Gate*, che al momento non vediamo. La struttura a livello di memoria contiene le seguenti informazioni:

- L'offset della routine di gestione dell'interruzione, in alcune modalità comprendente dell'indice di segmento, ecc...;
- **P:** un flag di **presenza**, indica se il descrittore è effettivamente abilitato;
- **L:** il livello di protezione (contesto sistema o utente) a cui deve essere eseguito il gestore. Notiamo che questa sembra essere una semplificazione del corso (il professore si è rivelato ombroso a riguardo). In verità, l'IDT mantiene un riferimento al CS dell'istruzione, che anche se ora abbiamo assunto come un semplice flag sistema/utente, rappresenta invece un riferimento al *segmento* vero e proprio all'interno del cui è allocata la routine. Informazioni riguardo al livello di ring di ogni segmento sono contenute in altre tabelle specifiche, dette **GDT** (*Global Descriptor Table*) e **LDT** (*Local Descriptor Table*). Il salto al livello L viene quindi fatto automaticamente in base al livello del segmento in cui è allocato il gestore (vediamo che con considerazioni simili si capisce come mai viene allocato, oltre a RIP, anche il CS corrente in fase di chiamata);
- **I/T:** il tipo di interruzione fra quelli sopra definiti.
- **DPL:** il livello minimo da cui si può accedere al gestore come interruzione interna (attraverso una `INT`). Questo non significa che tale gestore non possa essere lanciato da un eccezione.

Sorvolando su alcuni dettagli non immediatamente rilevanti (il valore del *Segment Selector* SS è piuttosto complesso, ma è quello che va a definire quello che noi intediamo con L), la struttura generale di un'entrata dell'IDT è quindi la seguente:



8.1.6 Gestione dell'interruzione software

Avevamo visto come il meccanismo dell'interruzione, definito un gate nella IDT, si riduceva al caricamento in RIP dell'indirizzo del gestore e dell'immissione in pila dei seguenti dati:

0	RIP
+1	CS
+2	RFLAGS

Cioè si impostava un nuovo frame sulla pila con i seguenti dati:

- L'instruction pointer **RIP**, da dove si vorrà ripartire nell'esecuzione una volta gestita l'interruzione. Notiamo che in verità questo indirizzo, che è fra l'altro in memoria virtuale, è corredato a seconda del tipo di gate dall'**SS** (*Stack Segment*) o dal **TSS** (*Task State Segment*), utili alla memoria segmentata che come abbiamo visto non ci è di interesse. La caratteristica importante è che si conserva un riferimento a dove ripartire, in memoria, nell'esecuzione una volta gestita l'interruzione;
- Il contenuto attuale di **CS**, cioè il contesto al momento della chiamata, che chiaramente vorremo ristabilire in seguito;
- Come abbiamo visto, anche **RFLAGS** viene memorizzato, in quanto gli interrupt mascherabili vengono mascherati in fase di gestione di un interrupt sistema (attraverso il flag **IF**), e vogliamo resettare questo comportamente al termine della gestione.

A questo punto l'unica differenza nella chiamata di interrupt in caso di cambio di contesto sta effettivamente nella transizione fra due **pile**: la separazione fra contesto utente e contesto sistema viene infatti resa possibile anche dalla presenza di due pile separate, di cui l'ultima chiaramente sta in memoria protetta. Il programma è normalmente in esecuzione nella pila utente: al momento del sollevamento di un interruzione software, si passa all'esecuzione (se alcune condizioni che vedremo fra poco sono rispettate) della routine di gestione definita dal sistema operativo. Questo richiede un modo per preservare la posizione della pila utente, da cui ci spostiamo quando passiamo alla pila sistema. Facciamo ciò conservando il vecchio **RSP**, immettendolo in pila prima dei registri visti prima, cioè creando un frame del tipo:

0	RIP
+1	CS
+2	RFLAGS
+3	RSP (Pila utente)

Il vecchio valore di RSP permetterà, fra l'altro, di accedere e modificare il contesto del *processo* in esecuzione con la sua pila utente.

Un caso particolare ma permesso è rappresentato dalla situazione dove **L**, il livello di destinazione, corrisponde allo stato attuale (ad esempio, sono permesse chiamate di interruzioni da contesto utente a contesto utente, o da contesto sistema a contesto sistema). In questo caso, chiaramente, tutta questa operazione verrà svolta su un unica pila (sia questa la pila utente o la pila sistema). Noteremo fra poco come questa possibilità rivela delle falle di sicurezza che vanno gestite.

8.1.7 Transizione da contesto sistema a contesto utente

La transizione inversa a quella vista adesso viene fatta semplicemente ritornando dall'interruzione attraverso la `IRETQ`. In questo caso si preleva dalla pila sistema (utente se eravamo in un interruzione a gestione livello utente) le informazioni che vi avevamo inserito al momento della chiamata dell'interruzione (RIP, CS ed EFLAGS) e si ristabilisce lo stato precedente al sollevamento dell'istruzione. Anche qui vi sono delle particolarità, che verranno spiegato, assieme a quelle annunciate in precedenza, nel paragrafo seguente.

8.1.8 Particolarità della gestione delle interruzioni software

Notiamo una particolarità riguardo alla transizione di contesto in fase di chiamata dell'interruzione (nota osservando il contesto attuale e l'L dell'interruzione lanciata), e riguardo alla transizione di contesto in fase di ritorno dall'interruzione (nota osservando il contesto attuale e il contesto salvato in pila).

Infatti, in fase di chiamata (quando si usa la `INT`), se L è minore del contesto corrente, viene lanciato un errore. La motivazione è principalmente una questione di simmetria nel meccanismo di chiamata delle interruzioni, piuttosto che una ragione di sicurezza: si vuole che le interruzioni ci portino in contesti maggiori o uguali del livello presente in CS.

Viceversa, se si prova a passare ad un livello superiore in fase di ritorno dall'interuzione (cioè quando si usa la `IRETQ`), viene lanciato un altro errore. La motivazione è che, visto che prevediamo nell'IDT il flag L, livello di destinazione, che permette di chiamare interruzioni in contesto utente, l'utente potrebbe impostare un frame di pila dove si richiede effettivamente l'accesso ad un livello di protezione superiore, e poi usare `IRETQ` per ritornare da tale frame di pila e passare quindi a tale livello di accesso.

9 Lezione del 18-03-25

9.1 Multiprogrammazione

Abbiamo accennato al funzionamento dei calcolatori in modalità *batch*, dove più programmi vengono eseguiti in sequenza, uno dopo l'altro.

Un paradigma sicuramente più piacevole per l'utente, e più diffuso al giorno d'oggi, è quello del **time-sharing**, dove il processore dà l'illusione agli utenti di portare avanti più attività contemporaneamente, mentre il tempo della CPU è in verita diviso in frammenti temporali ridotti dove si dedica a ogni attività singolarmente.

Il meccanismo stesso della protezione che abbiamo introdotto alla lezione precedente serve appunto a difendere i programmi l'uno dall'altro in caso di esecuzione "parallela" (da non confondere col *multithreading*). Infatti, anche se è un concetto nato nei *mainframe* a uso pubblico, la protezione si è subito diffusa anche nelle macchine personali degli utenti, in modo da difendere non più programmi di diversi utenti ma più programmi dello *stesso* utente, magari soggetti a bug che potrebbero corrompere lo stato di altri programmi o dell'intero sistema.

Oggi il meccanismo di protezione si trova in tutti i calcolatori moderni, dai telefoni cellulari ai supercomputer, ed è risparmiato solo nel caso dei microcontrollori più semplici.

La domanda che ci poniamo adesso è quindi quella di *come* realizzare un sistema capace di dare quest'illusione dell'esecuzione "parallela" di più programmi, che avevamo introdotto all'inizio del corso come **multiprogrammazione**.

9.1.1 Processo

Chiamiamo **processo** un programma in esecuzione. Ciò che vorremo eseguire in parellelo sono, più propriamente, non programmi ma *processi*.

Intendiamo quindi un processo non come il codice che definisce un programma, ma come il programma stesso una volta che viene messo in esecuzione nel calcolatore, quindi tutti gli stati di elaborazione (disposti nel tempo) del calcolatore nell'esecuzione di tale programma.

Il modo in cui andremo a definire il paradigma della multiprogrammazione è assumendo un processo come un insieme di operazioni **atomiche**, che possono essere interrotte al loro termine o prima del loro inizio, e che bastano insieme all'istruzione successiva del codice a determinare lo stato successivo di esecuzione del processo.

9.1.2 Contesto

Un altro concetto chiave nella multiprogrammazione sarà il **contesto** di un processo. Avevamo parlato di contesto in termini di protezine: adesso diamo un significato leggermente diverso. Ogni processo si aspetterà infatti di trovarsi nel *suo* contesto personale: le operazioni intaccheranno i suoi registri, che si aspetta essere l'unico a modificare, ecc... Il sistema operativo dovrà quindi essere in grado di fornire a ogni processo il suo contesto specifico.

Vediamo che questa idea si può tradurre già lato software. Il **cambio di contesto** può essere infatti effettuato, prendendo l'esempio dei soli registri, mantenendo una struttura data che contiene un'entrata per ogni registro. Al momento del cambio basterà copiare l'insieme dei registri corrispondenti al contesto di un certo processo nei registri veri e propri del processore.

Un discorso analogo sarà quella della memoria: ogni processo si aspetterà che al suo contesto corrisponda una sua copia della memoria. Possiamo mantenere un'altra struttura dati, simile a quella posta per i registri, che si occupa di mantenere informazioni riguardo alle regioni di memoria corrispondenti ad ogni contesto, e caricare quindi queste in una sezione dedicata su e della memoria stessa, o per semplicità su e dall'hard disk (così erano i primi sistemi time-sharing). Vedremo più nel dettaglio questo aspetto quando introdurremo la *memoria virtuale*.

Facciamo un ultima nota sulla *comunicazione* fra processi: nel caso più semplice, ogni processo non è al corrente dell'esistenza degli altri processi, e gestisce la sua *memoria privata*. Il sistema che studieremo dispone invece anche di una *memoria condivisa*, che permette ai processi di condividere informazioni fra di loro.

9.1.3 Kernel

Il programma che si occupa di effettuare queste operazioni di cambio di contesto si chiama **kernel** o *nucleo*. E' sempre in esecuzione in modo sistema e gestisce i contesti e le risorse assegnate ad ogni processo.

Immaginiamo quindi il kernel come un intermediario fra **processi** e **hardware**. Notiamo che questo non significa che kernel e processi sono *contemporaneamente* in esecuzione: questo è impossibile, in quanto la CPU è una sola. Kernel e processi sono infatti in esecuzione singolarmente, l'uno alla volta, e l'unico modo in cui si restituisce il controllo al kernel da un processo è attraverso i 3 tipi di interruzioni:

- Interruzioni esterne (dai dispositivi);
- Eccezioni (errori e altri malfunzionamenti, non necessariamente dati da errori di programmazione);
- Interruzioni interne (sollevate dall'istruzione `INT`).

Nel caso dei sistemi in time-sharing di cui abbiamo brevemente parlato prima, il cambio di processo viene eseguito ad intervalli regolari sfruttando interruzioni esterne periodiche generate da un timer.

10 Lezione del 21-03-25

Andiamo a definire più nei dettagli la struttura di un processo e le modalità secondo le quali questi si possono creare e distruggere.

10.1 Descrittori di processo

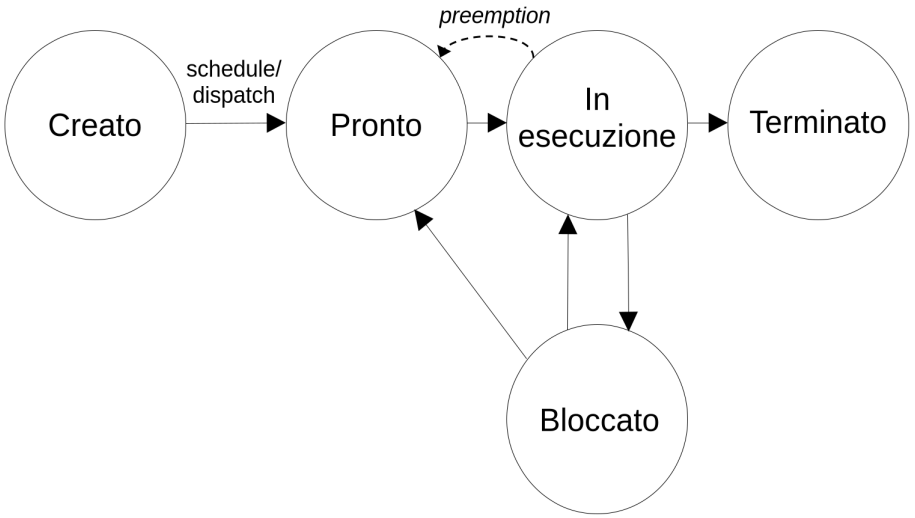
Un processo è descritto fondalmente da un astrazione, detta **descrittore di processo**, idealmente contenuta in una qualche locazione contigua, assieme ad altri descrittori, in memoria. Questo può essere schematizzato come una struttura C++ del tipo:

```
struct des_proc {
  // identificatore numerico del processo
  natw id;
  // livello di privilegio (LIV_UTENTE o LIV_SISTEMA)
  natw livello;
  // precedenza nelle code dei processi
  natl precedenza;
  // indirizzo della base della pila sistema
  vaddr punt_nucleo;
  // copia dei registri generali del processore
  natq contesto[N_REG];

  // prossimo processo in coda
  des_proc* puntatore;

  // informazioni di debug
};
```

dove quindi si tiene conto di:

- Un **indice** numerico unico ad ogni processo;
- Il **livello di privilegio** di un processo, che può essere *utente* o *sistema* (non è altro che la distinzione che avevamo fatto trattando la protezione di *processi utente* e *processi sistema*);
- La **precedenza** del processo, la cui motivazione ci sarà chiara fra poco;
- Un puntatore alla **pila sistema** del processo;
- Il **contesto** del processo, inteso come la copia di tutti i registri del processore.
- Infine, un puntatore al **prossimo processo**, in quanto intendiamo organizzare questi processi in linked liste ordinate per precedenza decrescente.

Ogni processo ha quindi un ciclo di vita che rispetta l'andamento del seguente grafico:



Vediamo nel dettaglio il significato delle diverse fasi. In fase di creazione del proesso, il suo descrittore viene posto in una struttura dati che ne consente **schedulazione** e **dispatch**:

- **Schedulazione:** effettivamente la scelta che il kernel fa, assunto il controllo, su qual'è il prossimo processo da portare in esecuzione (passaggio da processo **pronto** a processo in **esecuzione**);
- **Dispatch:** l'esecuzione effettiva di una serie di operazioni di tale processo.

I processi possono anche **bloccarsi**, cioè mettersi in attesa di qualche evento.

Infine, un processo può **terminare**, cioè sparire dal sistema (lui e il suo descrittore). Anche in questo caso il processo deve essere attualmente in esecuzione.

Una transizione che non è prevista da tutti i sistemi è quella di **preemption**, cioè di ritorno allo stato **pronto** a controllo dello scheduler. La maggior parte dei sistemi operativi supporta tale funzionalità, il nucleo che vedremo solo parzialmente.

10.1.1 Code di processi

L'esistenza di processi bloccati e pronti richiede l'esistenza di una struttura dati che ne tenga conto. Questa struttura dati, come abbiamo accennata, è rappresentata nel kernel studiato da linked list ordinate per precedenza decrescente. Una lista viene definita per i processi pronti:

```
// i processi pronti
des_proc* pronti;
```

mentre vedremo i processi bloccati lo sono in relazione a particolari oggetti, detti *semafori*.

Notiamo quindi l'esistenza della variabile `esecuzione`, che tiene conto del processo correntemente in esecuzione:

```
// il processo in esecuzione (sempre 1)
des_proc* esecuzione;
```

Visto che bisognerà lavorare con liste di processi, si definiscono funzioni per la loro manipolazione:

- **Inserimento di processo:** prende la forma di un semplice inserimento ordinato in lista.

```
void inserimento_lista(des_proc*& p_lista, des_proc* p_elem)
{
// inserimento in una lista semplice ordinata
//   (tecnica dei due puntatori)
  des_proc *pp, *prevp;

  pp = p_lista;
  prevp = nullptr;
  while (pp && pp->precedenza >= p_elem->precedenza) {
    prevp = pp;
    pp = pp->puntatore;
  }

  p_elem->puntatore = pp;

  if (prevp)
    prevp->puntatore = p_elem;
  else
    p_lista = p_elem;

}
```

- **Rimozione di un processo:** prende la forma dell'estrazione della testa (cioè del processo a priorità più alta).

```
des_proc* rimozione_lista(des_proc*& p_lista)
{
// estrazione dalla testa
  des_proc* p_elem = p_lista;   // nullptr se la lista e' vuota

  if (p_lista)
    p_lista = p_lista->puntatore;

  if (p_elem)
    p_elem->puntatore = nullptr;

  return p_elem;
}
```

- **Inserzione forzata:** è usata in casi particolari, inserisce il processo corrente in testa alla lista ignorando il suo livello di precedenza. La motivazione di tale comportamento è quella di non "svantaggiare" inutilmente il processo corrente se, ad esempio, ne si è interrotta l'esecuzione con preemption per la gestione di un interruzinoe esterna.

```
extern "C" void inspronti()
{
  esecuzione->puntatore = pronti;
  pronti = esecuzione;
}
```

10.2 Prima vista dell'esecuzione del kernel

Dopo il boot della macchina, il kernel si impadronisce della macchina e lancia il primo processo (il processo utente). Da qui in poi il kernel avrà il controllo solo fra un processo

e l'altro, in caso di interruzioni (interne, esterne o eccezioni), e potrà restituirlo solo attraverso il ritorno da gestore con `IRETQ`.

Come abbiamo visto, ad ogni chiamata di gestore di interruzione lascia RIP, CS, RFLAGS e RSP al tempo di chiamata dell'interruzione (facendo le opportune distinzioni fra *fault* e *trap*) in pila. A questo punto il gestore fa una copia dei registri generali, e si ha a quel punto una *"foto"* del processore al momento di attraversamento del gate, che rappresenterà quindi il *contesto* del processo stesso al momento della chiamata dell'interruzione.

In questo, sfrutteremo delle routine (`salva_stato` e `carica_stato`) all'avvio e al termine di ogni gestore, che si occupano di salvare e caricare il contesto del processo attualmente in esecuzione. Per conoscere quale questo processo sia, sfruttiamo la variabile globale nel sistema introdotta prima, `esecuzione`, che punta al descrittore del processo (che è dove vogliamo mettere il contesto stesso).

Un gestore di interruzione di base, quindi, si potrebbe magari occupare di passare al contesto e all'esecuzione del processo di priorità più alta a intervalli regolari, magari regolato da un timer (cosiddetto *timeslicing*).

Altre situazioni, più vicine a noi, sono quelle del termine di una gestione di un interruzione esterna, o bloccaggio automatico di un processo, dove il kernel deve selezionare il prossimo processo da eseguire, scegliendo chiaramente quello a priorità più alta.

10.2.1 Processo dummy

Inseriamo un processo fittizio, *dummy*, nella lista dei processi pronti con la priorità più bassa possibile. Questo ci assicurerà di non trovarci mai una situazione dove nessun processo è pronto all'esecuzione, e quindi avere sempre qualcosa a cui il kernel può passare (idealmente il processo dummy effettua solo un ciclo a vuoto).

10.2.2 Inizializzazione di un processo

Un ulteriore dettaglio è quello dello stato del processo alla sua creazione. Non è infatti realistico pensare di controllare se quel processo richiede inizializzazione ogni volta che si ritorna da un interruzione gestita a livello sistema. Alla creazione del processo, quindi, vogliamo svolgere le seguenti azioni in modo che il processo venga eseguito per la prima volta già in uno stato completo:

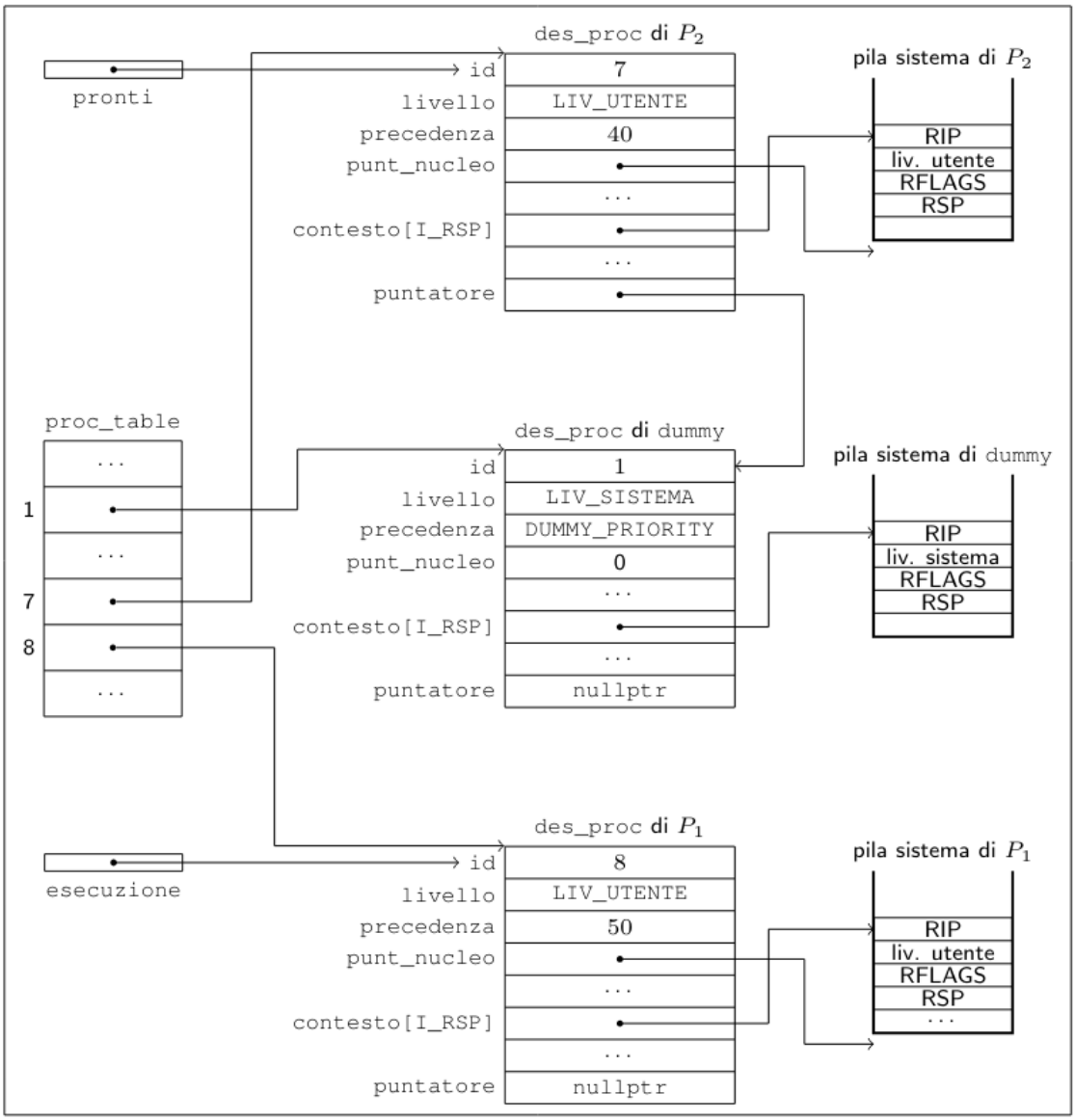
- Allocare una **pila sistema** dedicata al processo;
- Inizializzare la pila sistema. Questo consisterà nell'inizializzare a loro volta:
  - RIP alla prima istruzione del processo;
  - CS al segmento livello utente dove si trova il processo;
  - RFLAG a quanto viene richiesto dallo standard C++ al momento di avvio (solitamente tutto a 0), con l'eccezione di IF a 1.
- Allocare il **descrittore** di processo, e mettere quel processo fra i processi pronti;
- Inizializzare il descrittore. Questo consiste nell'inizializzare a loro volta:
  - Un puntatore alla pila sistema appena definita;
  - Il contesto del processo;
  - L'**argomento** di chiamata del processo, utile al debug;

– L'**IOPL**, *IO Privilege Level*, che specifica la possibilità o meno del processo di accedere all'IO.

10.2.3 Vista riassuntiva dei processi

Possiamo quindi riassumere brevemente l'intero meccanismo dei processi come definiti (e spesso sovrapposti) con la loro definizione in 10.1, che interagiscono col sistema attraverso le *primitive* (che vedremo più nel dettaglio alla prossima lezione), attraverso il meccanismo introdotto alla 10.2, sfruttando le interruzioni e i layout di pila descritti in 8.1.6.

Il tutto può essere spiegato bene dal seguente schema:



Tralasciando per adesso la `proc_table`, che non abbiamo visto, vediamo come la lista pronti punta alla radice di una lista di processi, dove manteniamo i campi dati notati sopra:

- L'*indice*, `id`;
- Il *livello di privilegio*, `livello`, che può essere `liv_utente` o `liv_sistema`;
- La *precedenza*, `precedenza`;
- Un puntatore alla *pila sistema*, `punt_nucleo`;
- Il *contesto*, `contesto`, del processo. Dallo schema, in particolare, si nota come il campo `I_RSP`, cioè il registro `RSP` salvato al momento della gestione dell'interruzione di primitiva, punta effettivamente alla pila sistema: per ottenere un riferimento alla pila utente bisogna prendere l'elemento dopo il `CS` l'`RFLAGS` salvati in pila sistema. Questo significa che avremo effettivamente due campi, `contesto[I_RSP]` e `punt_nucleo`, che puntano alla pila sistema: rispettivamente, il primo alla cima dell'ultimo frame (quello allocato in fase di gestione dell'ultima interruzione), quindi alla cima della pila, e il secondo alla base di tutta la pila sistema.

  Tutto questo comportamento deriva sostanzialmente dal fatto che parte dell'informazione che ci serve riguardo al processo nel caso di gestione di un interruzione di primitiva è salvata automaticamente dal processore al momento del salto da pila utente a pila sistema, con annesso cambio di contesto, proprio nella pila sistema, mentre l'altra informazione (i registri operativi operativi, ecc...) devono essere salvati manualmente dal kernel.
- Il puntatore al *prossimo processo*, `puntatore`.

Infine, notiamo come `esecuzione` punta al processo attualmente in esecuzone (non in contemporanea al kernel, ma che riprenderà l'esecuzione nel momento in cui si fara ritorno dalla primitiva con `IRETQ`).

11 Lezione del 24-03-25

11.1 Primitive

Abbiamo introdotto il concetto di primitiva, cioè di routine svolte dal sistema al servizio di un dato programma. Queste verranno implementate come gestori di interruzioni, quindi non propriamente funzioni, in quanto implicano un passaggio di contesto. Ciò nonostante, in un linguaggio come il C++ le primitive saranno comunque rappresentate da funzioni, dette *funzioni interfaccia*, scritte in assembly e che hanno il solo compito di usare la funzione `INT` con i parametri necessari alla chiamata di una specifica primitiva primitiva.

11.1.1 Primitiva di creazione di un processo

Abbiamo visto che la creazione di un processo consiste nell'inizializzazione dalla memoria ad esso dedicata in contesto utente e sistema, alla creazione del suo descrittore e all'inserzione di questo in pila *"pronti"*. Se la pila è rappresentata come una linked list, l'operazione dovra quindi essere quella di un *inserimento in testa*.

Notiamo che questa operazione non può essere divisa da altre interruzioni, in quanto richiede necessariamente almeno due passaggi, dove fra un passaggio e l'altro la lista viene lasciata in uno stato inconsistente:

- Prima si fa puntare il processo al resto della lista;

- Poi si fa puntare il puntatore della lista pronti al processo inserito.

anche invertendo l'ordine delle operazioni, dopo la prima la lista è inconsistente (in questo caso perché il processo inserito non viene effettivamente visto, nel caso opposto perché non vengono visti tutti gli altri).

Nel caso di routine di sistema basterà abbassare il flag `IF`, disabilitando effettivamente le interruzioni, durante tutta la durata della routine. A questo punto basterà evitare di generare eccezioni, e non usare mai l'istruzione `INT`, per ottenere una routine che viene eseguita dal processore nella sua interezza senza il rischio di interruzioni. Chiamiamo codice di questo tipo **codice atomico**. Il kernel Linux, ad esempio, *non* è atomico.

11.1.2 Disposizione delle primitive

La memoria del calcolatore conterrà in qualsiasi momento la tabella IDT, di cui abbiamo detto le prime 32 entrate rappresentano le eccezioni. Siamo quindi liberi di usare i gate dal 33 in poi per implementare le primitive. Per queste primitive dobbiamo impostare i parametri:

- **P:** 1, per attivare il gate;
- **L:** sistema, in quanto le primitive devono essere svolte a livello sistema;
- **DPL:** utente, in quanto le primitive devono essere accessibili all'utente;
- L'indirizzo effettivo della routine, implementata (in assembly, serve `IRET`), che deve trovarsi da qualche altra parte;
- **I/T:** tipo interrupt (interruzioni esterne mascherabili disabilitate).

Notiamo che l'interruzione esterna non mascherabile 2 è comunque in grado di bloccare le nostre istruzioni atomiche. Questo non è importante, in quanto abbiamo detto la useremo per casi particolarmente catastrofici (dove magari la salvaguardia dei dati dell'utente e del sistema e di maggiore priorità rispetto allo stato dei processi).

La struttura della routine sarà quindi tipicamente:

```
primitiva:
  CALL salva_stato
  CALL c_primitiva
  CALL carica_stato
  IRETQ
```

dove `c_primitiva` è una funzione, scritta in C++, che termina con una `RET` e lascia quindi che `primitiva` restituisca il controllo all'utente con `IRETQ`.

Per chiamare la primitiva da C++, come abbiamo detto, ci doteremo di una funzione di interfaccia del tipo:

```
primitiva_i:
  INT $ tipo %il tipo di primitiva
  RET
```

11.1.3 Passaggio di parametri alla primitiva

Supponiamo di voler passare dei parametri alla nostra primitiva. La funzione di interfaccia dovrà semplicemente essere modificata per accettare dati parametri (`primitiva_i(params...)`).

A questo punto la `primitiva_i` potrà svolgere il passaggio effettivo sfruttando i registri, solitamente il solo registro `%EAX` (in quanto `salva_stato` non modifica i registri).

11.1.4 Passaggio di parametri dalla primitiva

Per avere una restituzione di parametri da parte della primitiva la situazione è più complicata, in quanto abbiamo una chiamata a `carica_stato` prima del ritorno della primitiva per `IRETQ`.

Abbiamo però accesso al contesto di processo, nel descrittore di processo, e possiamo quindi modificare i registri che ci interessano direttamente lì.

11.1.5 Implementazione delle primitive processo

Vediamo quindi l'implementazione effettiva delle primitive relative a creazione e terminazione dei processi, cioè le `activate_p` e `terminate_p`. Queste chiaramente vengono chiamate da un handler scritto in assembly, che si occupa di salvare e caricare il contesto correttamente, in modo da non intaccare i registri in uso dal processo in `esecuzione`.

- `activate_p`: questa sfrutta una funzione, `crea_processo` (per adesso non significativa), che si occupa di creare effettivamente il descrittore di processo. Il suo compito è quindi solo quello di controllare che i parametri siano validi, chiamare `crea_processo()`, inserire il descrittore in lista pronti e restituire l'id del processo creato.

```
// crea un nuovo processo
extern "C" void c_activate_p(void f(natq), natq a, natl prio, natl
    liv)
{
  des_proc* p; // descrittore per il nuovo processo
  natl id = 0xFFFFFFFF; // id da restituire in caso di fallimento

  // seguono controlli di sicurezza sul livello
  [...]

  // crea effettivamente il descrittore di processo
  p = crea_processo(f, a, prio, liv);

  if (p != nullptr) {
    inserimento_lista(pronti, p);
    processi++;
    id = p->id;     // id del processo creato
                    // (allocato da crea_processo)
  }

  esecuzione->contesto[I_RAX] = id; // restituisci l'id del processo
}
```

- `terminate_p`: questa viene chiamata direttamente dal processo in `esecuzione`, quando questo desidera essere terminato. In questo, sfrutta una funzione, `distruggi_processo()` (anche questa al momento non significativa), che si occupa di ripulire il descrittore del processo (e quindi la sua pila, ecc...).

```
// termina il processo attuale
extern "C" void c_terminate_p()
{
  des_proc* p = esecuzione;

  distruggi_processo(p);
  processi--;
  schedulatore();
}
```

11.2 Semafori

Per gestire l'accesso condiviso ad una risorsa, nel nostro kernel adotteremo il meccanismo dei **semafori**.

Introdotti da Dijsktra nel 1962, questi si possono meglio modellizzare come una scatola piena di gettoni: ogni utente può mettere un gettone o prelevare un gettone dalla scatola, con la condizione che questa operazione sia atomica: se si tenta di prendere un gettone che non esiste, si resta in attesa finché quel gettone non viene effettivamente immesso nella scatola.

I problemi che vogliamo risolvere sfruttando i semafori sono effettivamente die due categorie:

- Problemi di **mutua esclusione:** assicurarsi che solo un processo possa accedere ad una risorsa in un dato momento.

  In questo caso si associa un gettone alla risorsa: accedere alle risorsa significa prendere il gettone, restituire la risorsa significa reinserire il gettone. L'esistenza di un singolo gettone assicura che solo un processo abbia accesso alla risorsa in un dato momento. Al momento della reimmissione del gettone, il processo che ne vince l'accesso sarà nel nostro kernel quello a priorità più alta.

  Notiamo inoltre che un processo che cerca di estrarre un gettone da una scatola vuota (tenta l'accesso ad una risorsa occupata o comunque non disponibile) dovrà aspettare che questa risorsa si renda disponibile: rappresenterà quindi il caso perfetto di **blocco** del processo, che può essere realizzato con **preemption** nei sistemi che la supportoìano;

- Problemi di **sincronizzazione:** esistono più attività, e ci interessa che alcune attività vengano fatte prime di altre (ordinamento *parziale*).

  Prendiamo l'esempio di avere due processi, A e B, e di volerci assicurare che A $\rightarrow$ B. In questo caso creiamo un semaforo associato al processo A, che parte vuoto. A mette il suo gettone nel semaforo quando finisce la sua esecuzione. A questo punto, B preleva il gettone ed esegue. Se B avesse provato ad entrare in esecuzione prima che A avesse terminato, non sarebbe riuscito a prelevare il gettone e avrebbe fallito.

  Nel caso di 2 processi (sempre A e B, con A che scrive e B che legge) che devono scambiare dati fra di loro ciclicamente, potremmo usare 2 semafori per realizzare un *handshake*. Ad esempio, definiamo quelle che effettivamente sono due variabili logiche sfruttando i semafori, che intendiamo come "buffer scritto" e "buffer letto". Il processo A dovrà semplicemente attivare il semaforo "buffer scritto" in fase di scrittura, e il processo B attivare il semaforo "buffer letto" in fase di lettura. Abbassando questi semafori al termine delle rispettive operazioni, e assicurandosi, osservando l'altro semaforo, di poter effettivamente procedere ad una nuova operazione, potremmo realizzare il paradigma desiderato.

Dal punto di vista di implementazione, il kernel fornisce una primitiva `sem_ini(int val)` che inizializza un semaforo con `val` gettoni iniziali, restituendone l'indirizzo. Da qui in poi i processi hanno accesso alle primitive `sem_wait()` e `sem_signal()`, che si occupano rispettivamente di richiedere e restiture un gettone.

11.2.1 Implementazione delle primitive semaforiche

Vediamo quindi l'implementazione delle primitive relative ai semafori, `sem_ini()` `sem_wait()` e `sem_signal()`.

Innanzitutto, un semaforo viene descritto dalla struttura:

```
// descrittore di semaforo
struct des_sem {
  // se >= 0, numero di gettoni contenuti;
  // se < 0, il valore assoluto e' il numero di processi in coda
  int counter;
  // coda di processi bloccati sul semaforo
  des_proc* pointer;
};
```

Si definisce quindi, sulla base di un paramero `MAX_SEM` che definisce il numero massimo di semaforo in ogni contesto:

```
des_sem array_dess[MAX_SEM * 2];
```

Il `* 2` è motivato dal fatto che si forniscono due array separate di semafori, una al contesto utente e una al contesto sistema.

L'array dei semafori non viene mai ripulita, e i semafori correntemente attivi vengono mantenuti invece da due indici:

```
// numero di semafori allocati per il livello utente
natl sem_allocati_utente  = 0;

// numero di semafori allocati per il livello sistema (moduli sistema e I/O)
natl sem_allocati_sistema = 0;
```

Le primitive vere e proprie sono quindi:

- `sem_ini()`: questa si serve di una funzione, `alloca_sem()`, che svolge gli opportuni controlli e incrementa l'indice nel vettore dei semafori corretto:

  ```
  // alloca un semaforo
  natl alloca_sem()
  {
    // i semafori non vengono mai deallocati, quindi e' possibile allocarli
    // sequenzialmente. Per far questo e' sufficiente ricordare quanti ne
    // abbiamo gia' allocati (variabili sem_allocati_utente e
    // sem_allocati_sistema)

    int liv = liv_chiamante();
    natl i;
    if (liv == LIV_UTENTE) { // semaforo utente
      if (sem_allocati_utente >= MAX_SEM)
        return 0xFFFFFFFF;
      i = sem_allocati_utente;
      sem_allocati_utente++;
    } else { // semaforo sistema
      if (sem_allocati_sistema >= MAX_SEM)
        return 0xFFFFFFFF;
      i = sem_allocati_sistema + MAX_SEM;
      sem_allocati_sistema++;
    }
    return i;
  }
  ```

```
24
25 // inizializza un semaforo
26 extern "C" void c_sem_ini(int val)
27 {
28   natl i = alloca_sem();
29
30   if (i != 0xFFFFFFFF)
31     array_dess[i].counter = val;
32
33   esecuzione->contesto[I_RAX] = i;
34 }
```

- `sem_wait()`: è semplicemente:

```
1  extern "C" void c_sem_wait(natl sem)
2  {
3    // controlli sulla validita' del semaforo
4    [...]
5
6    des_sem* s = &array_dess[sem];
7    s->counter--;
8
9    if (s->counter < 0) {
10     inserimento_lista(s->pointer, esecuzione);
11     schedulatore();
12   }
13 }
```

- `sem_signal()`: una particolarità di questa è l'uso della funzione `inspronti()`, che si rende necessario, come avevamo detto, per non svantaggiare inutilmente il processo corrente alla chiamata di `schedulatore()`:

```
1  extern "C" void c_sem_signal(natl sem)
2  {
3    // controlli sulla validita' del semaforo
4    [...]
5
6    des_sem* s = &array_dess[sem];
7    s->counter++;
8
9    if (s->counter <= 0) {
10     des_proc* lavoro = rimozione_lista(s->pointer);
11     inspronti();  // preemption
12     inserimento_lista(pronti, lavoro);
13     schedulatore(); // preemption
14   }
15 }
```

12 Lezione del 25-03-25

12.1 Attesa

Esiste un altra primitiva, la `delay`, che viene usata per sospendere un processo per un certo istante temporale.

Il kernel sfrutta di per sé il timer 1 per generare interruzioni periodiche, che lo assistano anche solamente a tenere traccia del tempo trascorso durante l'esecuzione. A

questo punto la `delay(natl n)` si limita ad aspettare `n` clicli del timer. Un'implementazione naive del timer è quindi quella di una lista di strutture, che rappresentano **richieste**, che tengono conto del loro conteggio attuale e del processo che le ha invocate. Un processo crea una richiesta sfruttando la primitiva `delay`, che risulta nella creazione di una richiesta e dello spostamento del processo nella lista bloccati. Il kernel dovrà quindi limitarsi ad aggiornare ad ogni ciclo di timer le richieste, decrementandole, e quindi ad riportare il processo in esecuzione una volta che il conteggio raggiunge 0.

Un modo più efficiente di fare la stessa cosa è quello di memorizzare non il conteggio di ogni richiesta, ma il conteggio *successivo* alla richiesta precedente nella lista d'attesa. Questo, chiaramente, implicherà un possibile riordinamento della lista in fase di inserzione (chi arriva prima sta in testa). In questo caso basterà decrementare solo il primo elemento della lista, e in occasione di raggiungimento di 0 eliminare quel processo e i successivi con conteggio aggiuntivo pari a 0.

12.1.1 Implementazione delle primitive d'attesa

Vediamo quindi l'implementazione vera e propria della primitiva `delay()`, secondo quanto detto finora. Questa si basa prima di tutto sulla definizione di una struttura `richiesta`, e dal mantenimento di una lista di tali richieste:

```
struct richiesta {
  // tempo di attesa aggiuntivo rispetto alla richiesta precedente
  natl d_attesa;
  // puntatore alla richiesta successiva
  richiesta* p_rich;
  // descrittore del processo che ha effettuato la richiesta
  des_proc* pp;
};

// Coda dei processi sospesi
richiesta* sospesi;
```

A questo punto serviranno due primitive, la `delay()` vera e propria, e la `driver_td()`, che si occupa effettivamente di avanzare temporalmente le richieste quando ha luogo un impulso di timer.

- `delay()`:

```
// primitiva di delay
extern "C" void c_delay(natl n)
{
  // caso particolare: se n e' 0 non facciamo niente
  if (!n)
    return;

  richiesta* p = new richiesta;
  p->d_attesa = n;
  p->pp = esecuzione;

  inserimento_lista_attesa(p);
  schedulatore();
}
```

- `driver_td()`:

```
// driver del timer
extern "C" void c_driver_td(void)
```

```
{
  inspronti();

  if (sospesi != nullptr) {
    sospesi->d_attesa--;
  }

  while (sospesi != nullptr && sospesi->d_attesa == 0) {
    inserimento_lista(pronti, sospesi->pp);
    richiesta* p = sospesi;
    sospesi = sospesi->p_rich;
    delete p;
  }

  schedulatore();
}
```

12.2 Memoria dinamica

Vediamo alla gestione della memoria dinamica, in particolare alla parola chiave `new` fornita dal linguaggio. Per noi le `new` non si tradurranno in altro che chiamate di funzione, che cercano una zona di memoria libera dove allocare il dato desiderato. Di contro, la `delete` si occuperà di deallocare lo stesso dato.

Una domanda che potremmo porci è dove si trova questa memoria. Per quanto riguarda il **sistema**, una porzione dedicata viene inizializzta all'avvio e resta tale durante l'esecuzione dello stesso. Le allocazioni e deallocazioni si fanno quindi con le `alloc()` e `dealloc()` (che ridefiniscono gli operatori corrispondenti, `new` e `delete`), definite all'interno di `libce.h`.

Per quanto riguarda l'**utente**, invece, si dedica un altra porzione di memoria, condivisa fra i processi. Questa condivisione implica che più processi non possono fornire le loro versioni della funzione `new` e `delete`, in quanto se queste venissero interrotte (le funzioni utente non sono mai atomiche), lascerebbero la memoria dinamica in uno stato inconsistente per altri processi intenzionati a modificarla.

Si usa quindi un semaforo che tiene conto di chi sta scrivendo in memoria. In particolare, vediamo le:

- `new`: implementata per l'utente come:

```
// alloca un oggetto nello heap utente
void* operator new(size_t s)
{
  void* p;

  sem_wait(userheap_mutex);
  p = alloc(s);
  sem_signal(userheap_mutex);

  return p;
}
```

- `delete`: implementata per l'utente come:

```
// dealloca un oggetto dallo heap utente
void operator delete(void* p)
{
  sem_wait(userheap_mutex);
```

```
5   dealloc(p);
6   sem_signal(userheap_mutex);
7 }
```

Queste vengono semplicemente fornite in un apposita libreria (`lib.h`) al programma utente, che può servirsene per scrivere, assieme agli altri processi, nell'heap utente.

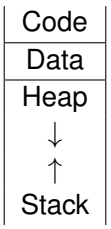
12.3 Memoria virtuale

Veniamo quindi all'ultimo argomento chiave del corso. Abbiamo detto che la memoria di sistema è divisa fra sistema e utente. In ogni momento ci aspettiamo che la memoria utente occupata da due porzioni: una **parte pubblica**, che rappresenta l'heap condiviso fra processi, e la **parte privata**, che rappresenta la memoria relativa ad un *singolo* processo, quello attualmente in esecuzione. La memoria privata degli altri processi è stata quindi intesa finora come memorizzata separatamente, magari nel disco rigido, con conseguente scaricamento del processo corrente e caricamento del successivo in memoria in fase di cambio di contesto fra processi.

Per i nostri scopi, possiamo assumere anche l'heap come parte della memoria privata. Il problema principale sarà infatti quello di poter memorizzare le immagini della memoria di *più* processi contemporaneamente. Infatti, storicamente, per *sistema multiprogrammato* si intendeva proprio il sistema in grado di mantenere più processi *in memoria* (il sistema visto finora sarebbe stato detto *multiprocesso*).

Decidiamo quindi di dividere la porzione di memoria utente in più sezioni, associate ad ogni processo. Potremmo intanto chiederci qual'è la memoria da dedicare ad ogni processo. La porzione dati e il codice di un programma sono infatti fissi in dimensioni, mentre pila e heap non lo sono. Storicamente, la memoria richiesta veniva specificata dal programmatore in fase di scrittura del programma. Quali metodologie si usino oggi non ci è immediatamente di interesse.

Si crea quindi per ogni processo una struttura di questo tipo:



Dove lo stack e l'heap si espandono in una sola regione, da estremità opposte.

13 Lezione del 28-03-25

Riprendiamo il discorso della memoria virtuale.

13.0.1 BASE e LIMIT

Avevamo detto che intendeveamo dividere la memoria utente fra processi, senza dover ricorrere al caricamento da disco della memoria relativa ad ognuno di essi.

Decidiamo quindi di dotare la CPU di due registri, **BASE** e **LIMIT**, che puntano rispettivamente all'inizio e alla fine della memoria dedicata al proesso, che questa dovrà

controllare per prevenire accessi all'esterno della zona definita quando ci si trova in modalità utente.

Chiaramente potrebbero esserci problematiche rispetto a quali indirizzi i singoli programmi vogliono usare: questi non potranno chiaramente usare salti a posizioni arbitrarie in memoria.

Una prima soluzione può essere quella di rendere il **PIC** (*Position Indipendent Code*), cioè codice indipendente dalla posizione (dove ad esempio le `CALL` e le `JUMP` saltano ad *offset*, e non ad indirizzi assoluti). Un primo problema di questo approccio è che costringe i programmi di stare all'intero di una zona di $\sim$ 4GB, in quanto gli offset sono su 32 bit (e non è nemmeno detto che il kernel dedicherà ad ogni processo la stessa quantità di memoria).

Un altro approccio può essere quello di realizzare un *caricatore rilocante* (ad esempio impementato in MS-DOS): si fa in modo che il collegatore lasci gli indirizzi non completamente specificati, e si definiscono una volta nota la posizione al partire da cui il processo verrà caricato (semplicemente incrementando gli indirizzi a partire da 0 in modo che puntino alla stessa posizione relativa al nuovo punto di inizio del programma).

Notiamo però che una problematica si presenterà sempre se intendiamo spostare processi fra memoria e disco in posizioni diverse, in quanto un processo potrebbe ad esempio poter aver messo un indirizzo assoluto in un registro, pianificando di effettuarci successivamente un accesso.

Un'altra problematica è che abbiamo perso l'accesso alla *memoria condivisa*, a meno di non sovrapporre le regioni definite dal BASE e LIMIT di due processi, sempre però limitandosi a due regioni molto specifiche di soli due processi.

Decidiamo quindi di usare il seguente approccio: ogni accesso in memoria ad un indirizzo x richiesto dal programma viene trasformato in un accesso a BASE $+ x$. In questo modo il collegatore potrà far partire ogni programma dall'indirizzo 0: ogni indirizzo usato da quel processo non sarà quindi altro che un offset a partire dall'inizio della regione di memoria dedicata a tale processo.

Risolveremo così i problemi relativi agli indirizzi assoluti, ma resterà il problema della dimensione del codice e della memoria condivisa.

Trascurando per adesso questi due dettagli, vediamo che abbiamo effettivamente realizzato una **memoria virtuale**, dove una certa funzione f mappa indirizzi *virtuali* $x_1, ...$ ad indirizzi *fisici* $v_1, ...$:

$$\begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix} \xrightarrow{f(x)=\text{BASE}+x} \begin{matrix} v_1 \\ v_2 \\ v_3 \end{matrix}$$

Vediamo però che spostare processi nella memoria comporta comunque un gran dispendio di risorse in quanto la memoria dedicata ad un processo può raggiungere dimensioni considerevoli, problema che viene solo moltiplicato quando si inizia a lanciare sempre più processi.

13.1 Paginazione

Questo problema, assieme in qualche modo agli altri due che avevamo lasciato in sospeso, può essere risolto agendo sulla funzione f. Decidiamo infatti di dividere la memoria processo in una serie di **pagine**, di dimensione fissa (prendiamo 4 KB), che possono prese in qualsiasi ordine dalla memoria centrale, ad unità sempre da 4 KB che chiamiamo **frame** .

A questo punto non avremo più bisogno di **continuità** nella memoria dedicata ai processi, cioè non avremo problemi di *frammentazione esterna*, anche se in qualche modo avremo introdotto *frammentazione interna* dove ogni processo dovrà ottenere memoria a "pacchetti" di 4 KB (che è comunque più vantaggioso).

BASE e LIMIT non saranno chiaramente abbastanza per gestire una situazione di questo tipo, e avremo quindi bisogno di una **tabella di corrispondenza**, allocata da qualche parte in memoria, che contenga una riga per ogni pagina, contenente il frame corrispondente alla pagina. Ogni indirizzo x sarà quindi scomposto in due valori, il **numero di pagina** e l'**offset di pagina** al suo interno. Il numero di pagina verrà quindi trasformato nel frame corrispondente alla pagina, e si potrà procedere all'accesso.

Ogni processo avrà quindi bisogno della sua tabella di corrispondenza personale, che pensiamo per adesso di poter semplicemente caricare e scaricare da memoria assieme al processo stesso.

Risolveremo quindi anche il problema della memoria condivisa, in quanto basterà mettere alcuni frame in comune fra più processi (starà al kernel tenere conto di quali frame sono in uso da quali processi e cosi via). Inoltre, agendo sulle tabelle, possiamo anche immaginare come questo sistema porterebbe almeno via di un livello di astrazione l'accesso a regioni di memoria più grandi di 4 GB.

13.1.1 Memory Management Unit

Aggiungiamo quindi, lato hardware e posta fra processore e cache, un nuovo componente detto **MMU**, *Memory Management Unit*, che tiene conto delle tabelle di corrispondenza e trasforma tramite esse le pagine degli indirizzi nei frame giusti.

Assumeremo che tutti gli indirizzi che la CPU genera saranno indirizzi virtuali, e che tutti gli indirizzi che escono dalla MMU saranno indirizzi fisici. Decidere che tutti gli indirizzi generati dalla CPU sono virtuali solleva una questione riguardo al kernel: idealmente, vorremmo che questo veda l'interezza della memoria, senza paginazione. Prendiamo quindi la sua memoria come in testa allo spazio di memoria, con le regioni successive dedicate ai frame di pagina dei processi.

Potremmo allora avere una tabella dedicata al solo kernel, che tiene conto di tutto lo spazio indirizzabile. Inoltre, potremmo prendere la tabella del kernel come *identiva*, cioè dare al kernel la visione della sua memoria *così com'è*.

Corrediamo allora la tabella di pagina introducendo:

- **P:** un bit di presenza, che definisce l'esistenza o meno di una traduzione per quell'indirizzo: nel caso di accesso a pagine non traducibili si genera un ecceione, detta **page fault**, che comporta il caricamento della pagina richiesta o la terminazione forzata del programma per **segmentation fault**.

  Ad esempio, se scegliamo 0 come la codifica del null pointer, vogliamo che la prima pagina (o le prime pagine, se vogliamo essere più larghi con accessi a strutture puntate da null pointer, che potrebbero avere offset negli struct anche considerevoli) sia non presente, e quindi si traduca in eccezione prima di effettuare accessi chiaramente erronei;

- **S/U:** *Sistema/Utente*, indica se una pagina è accessibile o meno ad un processo utente;

- **R/W:** *Read/Write*, indica se una pagina è accessibile solo in scrittura o solo in lettura per un certo processo. Questa può essere utile ad esempio per la sezione *text* del programma, che ricordiamo contiene il codice e non vogliamo venga modificata;

- **PCD** e **PWT**: indicano se ignorare completamente la cache (PCD) o se adottare una politica di scrittura *write-through* (PWT). Questo può essere utile nel caso di dispositivi mappati in memoria (come l'APIC o l'adattatore video);
- **A** e **D**: due flag che danno indicazioni agli accessi che la MMU ha individuato sulla pagina.

14 Lezione del 31-03-25

Continuiamo il discorso sulla paginazione.

14.0.1 Funzionamento della MMU

Avevamo definito una MMU che definiva tabelle di corrispodenza fra pagine e frame, una per ogni processo in esecuzione. Avevamo quindi detto che il processo (diciamo P_1) in esecuzione ha più sezioni di dati, cui potremmo assegnare ad esempio un frame ciascuna:

Sezione	Frame
Text (Code) P_1	2
Data P_1	3
...	//
Stack P_1	4

I *frame* di memoria scelti e sono in posizioni arbitrarie, l'unica cosa importante è che la MMU li possa rintracciare attraverso le sue tabelle di corrispondenza.

Potremo quindi assumere che la pagina 0 sia riservata, la pagina 1 riservata al sistema, e vedere che una tabella di corrispondenza per P_1 potrebbe essere la seguente:

Pagina	Sezione	Frame
0	Null	//
1	Sistema	1
2	Text P_1	2
3	Data P_1	3
...	...	//
7	Stack P_1	4

dove si è scelto di disporre lo stack in fondo allo spazio di memoria.

Nel momento in cui un altro processo (diciamo P_2) entra in esecuzione, potremmo assegnargli le seguenti pagine:

Sezione	Frame
Text P_2	5
Data P_2	6
...	//
Stack P_2	7

e disporre una tabella di corrispondenza:

Pagina	Sezione	Frame
0	Null	0
1	Sistema	1
2	Text P_2	5
3	Data P_2	6
...	...	//
7	Stack P_2	7

Vediamo che la pagina sistema resta nella tabella, ergo quella pagina è **condivisa** fra più processi. Cambiare contesto significherà quindi, oltre che caricare i registri, passare da una tabella di corrispondenza di processo all'altra. Il fatto che la pagina sistema sia sempre la 1 ci assicura che i suoi indirizzi siano per il programmatore sempre gli stessi.

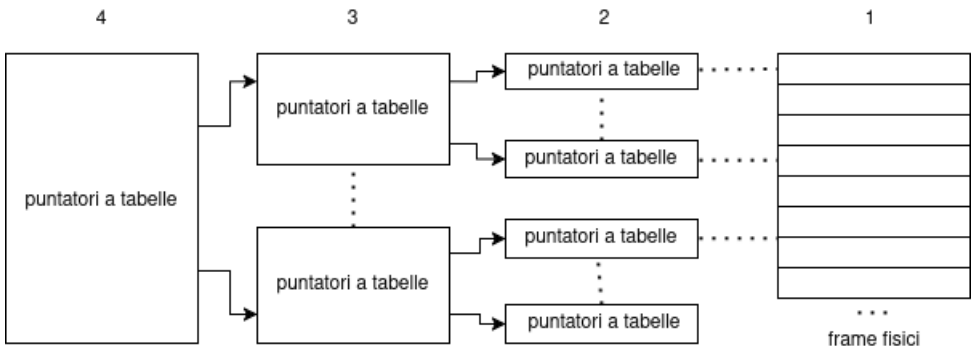
14.0.2 Verso la MMU reale

Vediamo che la MMU come l'abbiamo definita adesso è effettivamente di impossibile realizzazione. Vediamo infatti le dimensione di queste tabelle: se si prendono 12 bit di offset, per pagine da 4 KiB, si ha che nei 48 bit indirizzabili dall'architettura x86 (senza estendere a 57) lasciano 36 bit, e quindi 2^{36}, cioè 64 miliardi (64 Gi, *Gibi*) circa di pagine. Se vogliamo dedicare 8 byte ad ogni entrata di una tabella di corrispondenza, quindi, abbiamo bisogno di 512 GiB di spazio, che non è chiaramente fattibile (considerando poi che vogliamo una tabella per processo, quindi moltiplicando questo valore per un n arbitrariamente grande).

Chiamiamo quindi il modello fittizio visto finora **S-MMU** (da *Super MMU*) e ne introduciamo una versione più vicina alla realtà, che adotta una struttura dati diversa: la **T-MMU** (da *Trie MMU*).

La **trie** è una struttura dati che nasce per effettuare ricerche chiave-valore. Sono simili agli alberi binari, con la differenza che la chiave non è memorizzata nei nodi, ma nella posizione stessa dei nodi all'interno dell'albero.

Utilizziamo le trie per realizzare una struttura dati detta **bitwise tree**, o *albero bitwise*. L'idea è quella di dividere i 36 bit di pagina in 4 porzioni da 9 bit ciascuna, sulle quali costruire delle trie. La radice della struttura che costruiamo sarà quindi una tabella di $2^9 = 512$ entrate, corrispondente alle 512 possibili configurazioni dei primi 9 dei 36 bit di pagina. Ognuna di queste entrate punterà ad un altra tabella di 512 entrate, relative ai 9 bit successivi. Si hanno quindi 4 livelli di accesso, ordinati dal 4 all'1, che bisogna attraversare per arrivare fino al frame corrispondente alla pagina che ci interessa:



Nei sistemi operativi moderni, quali Linux, ogni livello di accesso ha un nome specifico e quindi ogni segmento da 9 bit dei 36 bit di pagina rappresenta un informazione denominata diversamente, che riportiamo per completezza (dal più significativo):

Bit	Nome	Significato
9 bit	PML4I	*Page Map Level 4 Index*
9 bit	PDPTI	*Page Directory Pointer Table Index*
9 bit	PDI	*Page Directory Index*
9 bit	PTI	*Page Table Index*
12 bit	//	Offset di pagina

Resta comunque il fatto che quale nome decidiamo di assegnare alle tabelle di ogni livello di accesso non è importante, in quanto le strutture ad ogni livello sono sovrapposte (puntare a una tabella o a un frame è la stessa cosa).

Il procedimento che ci porta dai bit di pagina all'indirizzo del frame si chiama **table walk**, o *cammino della tabella*. Ogni entrata delle tabelle di trie sarà grande 8 byte (almeno 7 bit per i flag, più ~ 48 bit di indirizzo, ricordando che lo spazio indirizzabile nell'x86_64 non corrisponde al massimo di 64 bit), per cui $2^9 \cdot 2^3 = 2^{12} = 4$ KiB di memoria ciascuna.
La memoria massima *raggiunta ad* ogni livello e il numero di entrate *raggiunta ad* ogni livello sono quindi:

Livello	Memoria massima usata	Numero di entrate
4	$2^9 \cdot 2^3 = 2^{12} = 4$ KiB	$2^9 = 512$
3	$2^9 \cdot 2^9 \cdot 2^3 = 2^{21} = 2$ MiB	$2^9 \cdot 2^9 = 2^{18} = 256$ Ki
2	$2^9 \cdot 2^9 \cdot 2^9 \cdot 2^3 = 2^{30} = 1$ GiB	$2^9 \cdot 2^9 \cdot 2^9 = 2^{27} = 128$ Mi
1	$2^9 \cdot 2^9 \cdot 2^9 \cdot 2^9 \cdot 2^3 = 2^{39} = 512$ GiB	$2^9 \cdot 2^9 \cdot 2^9 \cdot 2^9 = 2^{36} = 64$ Gi

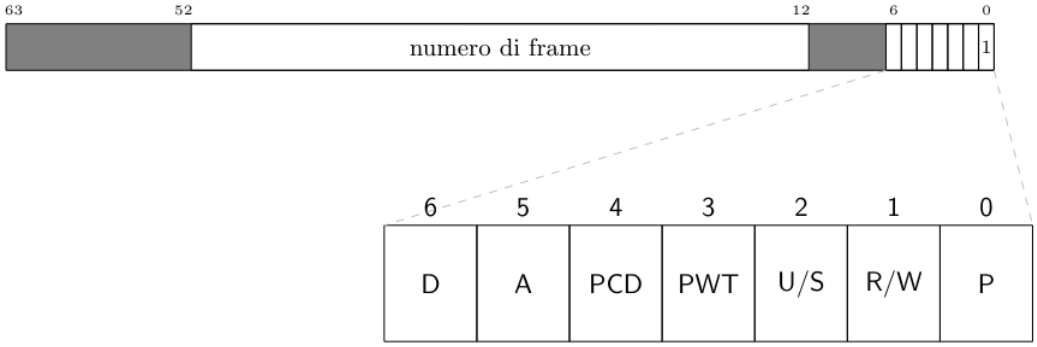
Per cottenere l'informazione inversa, cioè la memoria massima *raggiungibile da* ogni livello e il numero di entrate *raggiungibili da* ogni livelo basterà invertire l'ordine verticale delle ultime 2 colonne.

Potremmo allora chiederci dov'è il guadagno di un sistema di questo tipo, in quanto a memoria l'ultimo livello di trie necessiterà degli stessi 512 GiB, più lo spazio necessario ai livelli precedenti. Il vantaggio sarà però quello di poter tagliare arbitrariamente rami dall'albero che abbiamo formato, cioè non tenere conto di pagine di cui non abbiamo attualmente bisogno.

14.0.3 Descrittori nella T-MMU

Vediamo come si evolvono i descrittori che avevamo messa corredo delle tabelle di corrispondenza, nella T-MMU. Avremo che ci dovrà essere una distinzione fra i descrittori di primo e di secondo, terzo e quarto livello:

- **Descrittori di primo livello:** qui vogliamo usare l'intero insieme di descrittori, che riportiamo:



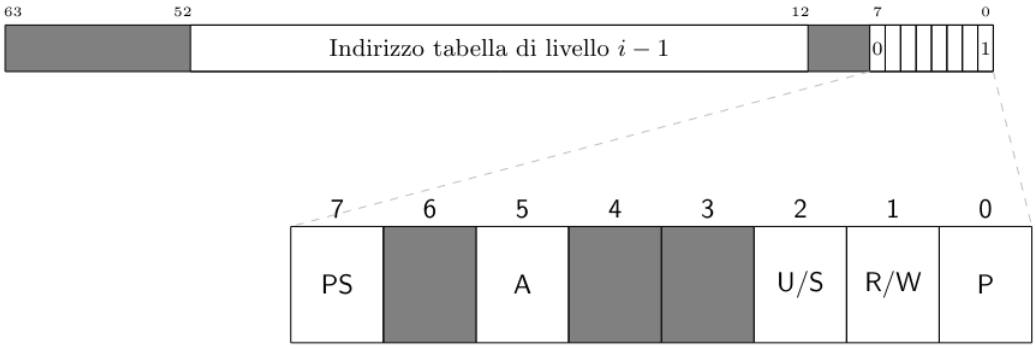
  - **P:** un bit di presenza, che definisce l'esistenza o meno di una traduzione per quell'indirizzo: nel caso di accesso a pagine non traducibili si genera un eccezione, detta **page fault**, che comporta il caricamento della pagina richiesta o la terminazione forzata del programma per **segmentation fault**.

    Ad esempio, se scegliamo 0 come la codifica del null pointer, vogliamo che la prima pagina (o le prime pagine, se vogliamo essere più larghi con accessi a strutture puntate da null pointer, che potrebbero avere offset negli struct anche consideravoli) sia non presente, e quindi si traduca in eccezione prima di effettuare accessi chiaramente erronei;
  - **S/U:** *Sistema/Utente*, indica se una pagina è accessibile o meno ad un processo utente;
  - **R/W:** *Read/Write*, indica se una pagina è accessibile solo in scrittura o solo in lettura per un certo processo. Questa può essere utile ad esempio per la sezione *text* del programma, che ricordiamo contiene il codice e non vogliamo venga modificata;
  - **PCD** e **PWT**: indicano se ignorare completamente la cache (PCD) o se adottare una politica di scrittura *write-through* (PWT). Questo può essere utile nel caso di dispositivi mappati in memoria (come l'APIC o l'adattatore video);
  - **A** e **D**: due flag che danno indicazioni agli accessi che la MMU ha individuato sulla pagina.

    Vediamo solo adesso la loro utilità: la MMU setta questi bit per dare informazioni al kernel su cosa è successo alle pagine fino all'ultimo accesso. Il bit **A**, quindi, indica che una certa pagina è stata usata (*attraversata*), mentre il bit **D** (*Dirty*) indica che si è scritto su una certa pagina. Abbiamo quindi una situazione dove è l'*hardware* ad informare il *software* del suo funzionamento, e non viceversa (come eravamo abituati). L'informazione può quindi essere usata per gestire meglio il caricamento su e da memoria delle pagine, soprattutto in sistemi che supportano la *memoria di swap*, cioè una certa porzione di memoria sul disco rigido che viene impiegata nella memorizzazione delle immagini dei processi in esecuzione (che è come, in origine, avevamo ipotizzato funzionasse il meccanismo della multiprogrammazione). In questo caso conoscere il flag D può evitare una scrittura su disco quando una pagina non è stata modificata, mentre conoscere il flag A può dare un euristica su quali

pagine conviene spostare nello swap e quali mantenere nel caso di spostamento di pagine da e su disco. Per la precisione, in sistemi di questo tipo i pagefault sono normali, e vengono sfruttati per realizzare la *paginazione su domanda*: può essere che la pagina richiesta da un processo non esista, quindi comporti un'eccezione, che viene gestita caricando la pagina corrispondente (e quindi verificando i flag A se altre pagine vanno rimosse per fare spazio).

- **Descrittori di secondo, terzo e quarto livello:** il descrittore ha questo aspetto:



in questo caso non abbiamo bisogno di **PWT**, **PCT** e **D**, mentre introduciamo un nuovo bit, **PS**, *Page Size*, che distingue due situazioni: se PS è basso, si procede come si è detto finora, altrimenti, quella entrata punta ad un unica pagina contigua di entrate (e non al livello successivo della trie). Il numero di entrate delle pagine contigue, dette **huge page**, cambia quindi in base al livello:

Livello	Memoria indirizzata in huge page
4	//
3	1 GiB
2	2 MiB
1	4 Kib (default)

in quanto, ad ogni livello, stiamo effettivamente "passando" 9 bit dall'indirizzo di pagina all'offset nella pagina, cioè stiamo adottando indirizzi e offset di dimensione:

Livello	Dimensione indirizzo	Dimensione offset
4	//	//
3	18 bit	30 bit
2	27 bit	21 bit
1	36 bit	12 bit

Come si vede poi dalla tabella, il flag PS è effettivamente ignorato al livello 4 (avremmo pagine da 512 GB, che ad oggi non tornano particolarmente utili) e al livello 1 (è la dimensione di default delle pagine).

14.0.4 T-MMU e memoria condivisa

La struttura ad albero delle trie ci permette, ad esempio, di far puntare un entrata di un sottoalbero della trie associata ad un processo, ad un sottoalbero di una trie di un

altro processo. Questo ci permette effettivamente di realizzare pagine, o tabelle di pagine, condivise fra processi. Potremo liberamente assegnare la stessa pagina in posizioni diverse dello spazio di memoria di ogni processo, in quanto l'unica cosa importante è il *percorso* che ci porta alla tabella condivisa, che può variare di processo in processo (o meglio di trie di processo in trie di processo).

15 Lezione del 01-04-25

Riprendiamo la trattazione dei moduli MMU. Eravamo partiti dalla S-MMU, che prendevamo solo come esempio funzionale, e avevamo in seguito introdotto la T-MMU, che sfrutta una struttura dati ad albero (la *trie*) per mantenere in aniera più efficiente le associazioni pagina-frame. Passiamo adesso alla descrizione della **MMU** vera e propria, senza le semplificazioni che avevamo assunto per la S-MMU e la T-MMU.

15.0.1 MMU Reale

Abbiamo che le tabelle della MMU stanno in RAM, assieme ai dati stessi cui la MMU vuole accedere. La struttura *trie* sta quindi in memoria, e l'MMU è dotata di un registro **CR3** che mantiene la posizione della prima tabella (avevamo visto, quella di livello 4). Il processore si occupa quindi di creare la trie, e di fornirne l'indirizzo alla MMU, su tale registro.

Notiamo che gli indirizzi che stanno nelle tabelle della MMU, che sono comunque in memoria, sono indirizzi *fisici*, e anzi il contenuto dello stesso registro CR3 è un indirizzo *fisico*.

Perchè il kernel possa accedere a tali indirizzi senza problemi (ricordiamo che tutto ciò che esce dalla CPU è considerato dalla MMU come un indirizzo *virtuale*), si mette tale struttura nella memoria sistema, in modo che gli indirizzi ivi contenuti si traducano in loro stessi.

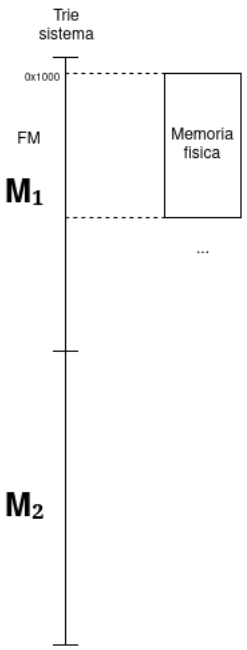
15.0.2 Bootstrap della MMU

Nello specifico, possiamo immaginare che il kernel faccia, all'avvio, le seguenti operazioni:

1. Preso lo spazio indirizzabile come diviso in 2 (che abbiamo visto essere il caso nell'architettura x86_64), dedica una delle due parti (la superiore in Windows, l'inferiore in Linux) alla memoria sistema;

2. Inizializza un primo trie per la MMU in memoria sistema, e vi crea una serie di pagine (una **finestra** sulla memoria virtuale, detta **FM**), corrispondenti ai loro frame, di tipo sistema, in modo da poter indirizzare la memoria con indirizzi che si traducono in loro stessi.

Si andrà quindi a creare una struttura del tipo:



dove notiamo che non si indirizza la prima pagina (fino a `0x1000`), in quanto questa sarà dovè andranno a finire gli indirizzi nullptr, che vogliamo catturare con un'eccezione di pagefault.

3. Carica tale trie in CR3 ed abilita la MMU.

Da qui in poi il kernel avrà accesso, attraverso tale finestra, all'interezza della memoria fisica mappata con indirizzi fisici (cioè con indirizzi virtuali mappati con l'identità ad indirizzi fisici).

Notiamo di poter tranquillamente creare tale finestra, in quanto in 2^{48} possibili indirizzi virtuali mappiamo anche più di una volta tutta la finestra della memoria fisica.

15.0.3 Translation Lookaside Buffer

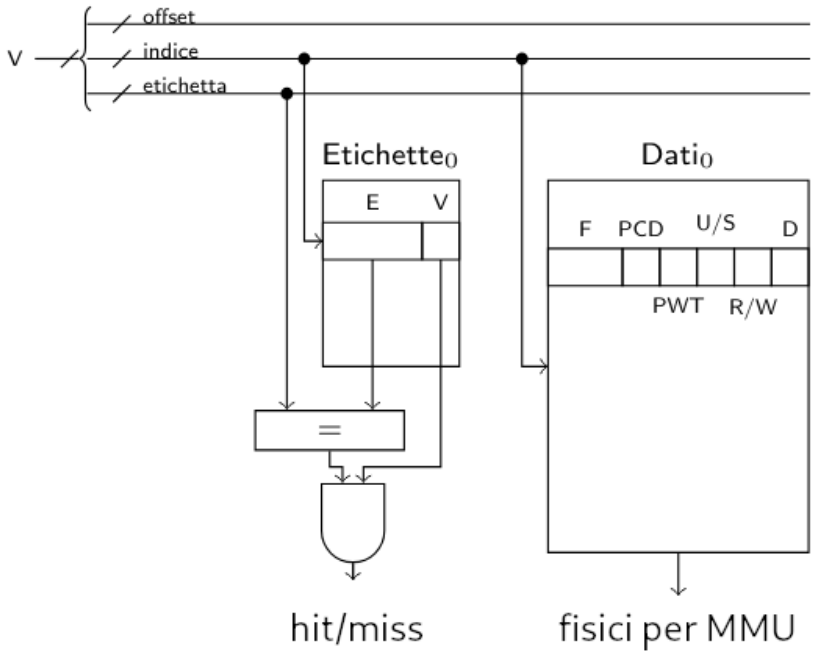
Abbiamo che sfruttando il modello visto finora ogni accesso alla RAM si traduce in realtà a diversi accessi (fino a 5, accesso alle tabelle dal 4 all'1 della trie corrente, e l'accesso all'indirizzo fisico desiderato).

Visto che era proprio la RAM ad essere, come avevamo detto, la parte più lenta del sistema, è necessario introdurre una cache a parte dedicata alla MMU, che viene detta **TLB**, *Translation Lookaside Buffer*. Il TLB tiene traccia delle coppie indirizzo virtuale - indirizzo fisico più usate, limitando la necessità dei table walk alle sole istanze dove sono strettamente necessari.

Il TLB è perlopiù trasparente alla CPU. Esistono istruzioni, però, come la `INVLPG`, che permettono di modificare lo stato del TLB. Questa infatti permette di invalidare un indirizzo virtuale, se questo è contenuto nel TLB (costringendo a un nuovo page walk in fase di ricerca di tale indirizzo).

La struttura del TLB è analoga a quella della cache: si prendono i 48 bit di indirizzo virtuale, di cui i 12 più bassi saranno come sempre l'offset, e si dividono i 36 rimanenti in due parti che usiamo come indice e come numero di pagina in una memoria delle etichette. Una and fa il bit di validità nella tabella delle entrate e un comparatore fra il numero di pagina e l'etichetta fissata all'indice corrente ci darà quindi gli hit di cache.

La struttura è quindi la seguente:



Dove una memoria a sé stante conterrà quindi gli indirizzi fisici veri e propri, oltre ad alcuni flag. Notiamo che questa non ha bisogno di conoscere A (se è nel TLB ci è già stato fatto accesso). Ad esempio però vogliamo sapere PCD e PWT, in quanto questi regoleranno l'accesso o meno via cache. Abbiamo poi un solo bit dedicato a U/S e R/W, in quanto comunque tutto ciò che vorremo sapere è se a quell'indirizzo si può accedere o meno nel contesto corrente. Infine, manteniamo il bit D. Effettuare il write back di tale bit è complicato in quanto occorre effettivamente ritrovare la pagina effettuando il table walk. Possiamo risolvere questa situazione fingendo di non conoscere (praticamente invalidando) una pagina, anche se conosciuta, quando il suo bit D è basso, costringendo l'MMU ad effettuare il page walk ed alzarlo.

Notiamo infine di poter applicare tutte le ottimizzazioni che avevamo visto per le cache tradizionali, inclusa l'introduzione di memorie aggiuntive e una memoria per l'LRU, in modo da realizzare effettivamente una cache a contenitori associativi.

15.0.4 Pagine di grandi dimensioni

Vediamo un ultimo dettaglio sulla MMU approfondendo il discorso delle pagine di grandi dimensioni (*huge page*). Anche se teoricamente si potrebbe procedere ignorando questi oggetti, il significato della MMU è quello di mantenere *pagine*, e quindi offset all'interno di tali pagine, e dobbiamo quindi mantenere informazione riguardo al fatto che gli indirizzi in una huge page sono più grandi di quelli in una pagina normale da 4 KiB (come abbiamo riportato in 14.0.3).

Storicamente, trovata una huge page, la MMU la caricava interamente all'interno della cache TLB. Oggi, si mantengono 3 TLB separati per ogni dimensione di pagina (1 GiB, 2 MiB e 4 KiB).

16 Lezione del 04-04-25

16.1 Funzioni di supporto alla paginazione

Riprendiamo la trattazione della memoria virtuale paginata, discutendo quali funzioni il kernel studiato mette a supporto della sua operazione.

16.1.1 Funzioni sugli indirizzi

Nell'architettura x86_64 un indirizzo, virtuale o fisico, sta su 64 bit, e quindi definiamo i tipi `vaddr` e `paddr`:

```
typedef natq /* (uint64_t) */ vaddr; // indirizzo virtuale
typedef natq /*    idem    */ paddr; // indirizzo fisico
```

Dotiamoci quindi di alcune funzioni per la gestione di questi indirizzi. Di base, vorremmo un modo per verificare la normalizzazione di un indirizzo (quindi il fatto che i bit dal 48 al 63 siano uguali al bit 47), che potrebbe essere la `norm(vaddr)`:

```
if(norm(v) != v) {
  // errore: indirizzo non normalizzato
}
```

Definiamo poi due funzioni per trovare l'indirizzo della prima pagina contenuta e della prima pagina immediatamente dopo una regione di memoria $[x, y]$. Avevamo trattato questo problema nella sezione 2.1.3, e usando quanto avevamo detto possiamo definire le funzioni `base(vaddr)` e `limit(vaddr)`.

```
vaddr base(vaddr v, int liv)
{
  natq mask = dim_region(liv) - 1;
  return v & ~mask;
}

vaddr limit(vaddr v, int liv)
{
  natq dr = dim_region(liv);
  natq mask = dr - 1;
  return (v + dr - 1) & ~mask;
}
```

dove notiamo che l'indice di pagina resta comunque nei 36 (o meno nel caso di huge page) bit dedicati al numero di pagina (nel caso più semplice, azzeriamo i 12 bit di offset di pagina).

La funzione helper `dim_region(int)` calcola la dimensione di una pagina ad un certo livello di paginazione (sui i livelli $4, 3, 2, 1$, indicati come $3, 2, 1, 0$), ed è definita come:

```
natq dim_region(int liv)
{
  natq v = 1ULL << (liv * 9 + 12);
  return v;
}
```

16.1.2 Funzioni sulle tabelle

Un singolo descrittore di tabella viene rappresentato dal tipo `tab_entry`, che entra (come abbiamo visto nella sezione 14.0.3) in 64 bit, per cui possiamo dire:

```
typedef natq /* (uint64_t) */ tab_entry;
```

L'interazione vera e propria con il descrittore avverrà attraverso maschere e funzioni che applicano maschere. Ad esempio, per modificare il bit di presenza di una pagina si può dire:

```
// e e' un riferimento ad una tabella
e |= BIT_P; // BIT_P maschera il bit di presenza
```

mentre se si vuole estrarre o modificare l'indirizzo fisico si possono sfruttare rispettivamente le funzioni `extr_IND_FISICO(tab_entry)` e `set_IND_FISICO(tab_entry)`:

```
// e e' un riferimento ad una tabella
paddr p;

// prendi l'indirizzo fisico di e
p = extr_IND_FISICO(e);

// per esempio, rimetticelo
set_IND_FISICO(e, p);
```

Esiste poi le funzione `i_tab(vaddr v, int liv)` per l'accesso alla regione da 9 bit che indirizza le tabelle di livello `liv` di un indirizzo `v`. Questa potrà essere usata con le funzioni `get_entry(paddr, int)` e `set_entry(paddr, int, tab_entry)`, che si occupano rispettivamente di ottenere un entrata di una tabella e modificarne una, sostituendo l'intero descrittore.

Ad esempio, potremo dire:

```
// questo e' un indirizzo, diciamo che la tabella tab e' di livello 2
vaddr v;

// e e' un descrittore di tabella di livello 1
tab_entry& e = get_entry(tab, i_tab(v, 2));

// creiamo un descrittore per sostituire e
tab_entry se;
// ... imposta se

// modifica la stessa entrata
set_entry(tab, i_tab(v, 2), se);
```

Copie o sovrascritture in massa si possono fare poi con le funzioni `copy_des()` e `set_des()`, che per adesso non vediamo nel dettaglio.

16.1.3 Funzioni sulla MMU

Notiamo l'esistenza della `loadCR3()` per l'attivazione di un nuovo albero di traduzione, che carica un indirizzo fisico nel registro `CR3`, e `readCR3()`, che permette successivamente di rileggerlo.

Esiste anche la `readCR2()`, che permette la lettura di `CR2`, dove si trova l'ultimo indirizzo la cui traduzione ha causato un pagefault.

Esistono poi le funzioni di interazione con il TLB: `invalida_entrata_TLB(vaddr v)` permette di invalidare l'indirizzo virtuale `v` sfruttando l'istruzione assembler `INVLPG`, mentre `invalida_TLB()` invalida l'intero albero di traduzione (cosa che ricordiamo faceva già la `loadCR3()`).

16.1.4 Iteratori di tabella

La gestione ad alto livello dei trie (*alberi di traduzione*) viene effettuata sfruttando l'iteratore `tab_iter`.

Questo serve per effettuare visite in diversi ordini (che vederemo fra poco), a livello di *pagina*: l'offset di pagina andrà comunque conservato a parte, in quanto l'iteratore ci porterà, al massimo, solo fino all'indirizzo fisico della pagina giusta. Sui `tab_iter` sono definite alcune funzioni membro: le `get_e()`, `get_tab()` e `get_l()` permettono di ottenere, rispettivamente, un riferimento all'entrata su cui si trova l'iteratore, l'indirizzo fisico della tabella che contiene questa entrata e il livello (4, 3, 2 o 1) di questa tabella. La funzione `get_v()`, invece, restituisce il più piccolo indirizzo virtuale la cui traduzione passa da questa entrata.

Il `tab_iter` viene inizializzato attraverso un indirizzo virtuale, o una coppia di questi in modo da esplorare un intera regione di indirizzi virtuali. Una volta definito un'oggetto `tab_iter`, si possono sfruttare le funzioni `up`, `down` e `right` per spostarsi rispettivamente nella tabella di livello superiore, inferiore, e a destra fra le tabelle di livello corrente. La funzione `done()`, che si ottiene anche dall'operatore di conversione a `bool`, restituisce falso quando la visita è terminata (siamo arrivati alla pagina o non possiamo proseguire).

Attraverso queste funzioni, e alle `next()` (che avanza l'iteratore in avanti in visita anticipata, cercando di raggiungere il prossimo indirizzo) e `next_post()` (che avanza l'iteratore in avanti in visita posticipata) si possono realizzare quindi diversi tipi di visita ad un trie, fra cui:

- **Visita anticipata di un singolo indirizzo:** si percorre l'intero percorso di traduzione di un indirizzo `v` come segue:

```
for(tab_iter it(tab, v); it; it.next()) {
  // it e' l'elemento corrente
}
```

  Una soluzione alternativa si ha sfruttando direttamente `down()`:

```
tab_iter it(tab, v);
while (it.down()) {
  // it e' l'elemento corrente
}
```

- **Visita anticipata di una regione di indirizzi:** analoga a sopra, ma si fornisce l'indirizzo base e la dimensione della regione al costruttore del `tab_iter`:

```
for(tab_iter it(tab, v_lo, v_hi); it; it.next()) {
  // it e' l'elemento corrente
}
```

- **Visita posticipata di una regione di indirizzi:** ancora analoga a sopra, ma si scende del tutto sfruttando la `post()`, e poi si prosegue con la `next_post()`:

```
tab_iter it(tab, v);
for (it.post(); it; it.next_post()} {
  // it e' l'elemento corrente
}
```

  Notiamo che la `post()` non implementa altro che la visita anticipata che abbiamo visto prima:

```
void tab_iter::post()
{
  // controlli di validita'
  if (done())
    return;

```

```
  while (down())
    ;
}
```

16.1.5 Trasformazione

Possiamo quindi definire la funzione `trasforma(paddr, vaddr)` che usa l'albero di traduzione puntato dall'indirizzo fisico al primo argomento per tradurre, in un altro indirizzo fisico, l'indirizzo virtuale al secondo argomento. Solitamente l'albero di traduzione che ci interesserà sara quello attualmente caricato in `readCR3()`, cioè:

```
// v e' un indirizzo virtuale
paddr p = trasforma(readCR3(), v);
```

L'implementazione della `trasforma()` si riduce effettivamente a:

1. Esegui una visita in ordine anticipato fino alla pagina corretta;
2. Verifica se la pagina è stata effettivamente ottenuta (altrimenti restituisci l'indirizzo 0);
3. Combina l'indirizzo fisico di pagina con l'offset di pagina.

cioè in codice:

```
paddr trasforma(paddr root_tab, vaddr v)
{
  // punto 1
  tab_iter it(root_tab, v);
  while (it.down())
    ;

  // punto 2
  tab_entry e = it.get_e();
  if (!(e & BIT_P))
    return 0;

  // punto 3 (con un dettaglio riguardante le huge page: si prende come
    offset la maschera ottenuta da dim_region())
  int l = it.get_l();
  natq mask = dim_region(l - 1) - 1;
  return (e & ~mask) | (v & mask);
}
```

16.1.6 map() e unmap()

Vediamo infine due funzioni che permettono di mappare e liberare regioni di memoria contigue nello spazio virtuale, ottenendo attraverso un qualche helper `alloca_tab()` nuovi frame fisici (anche non contigui) e rilasciandoli con `rilascia_tab()`. Queste sono la `map()` e la `unmap()`.

La chiamata della `map()` è:

```
vaddr map(paddr tab, vaddr begin, vaddr end, natl flags, T& getpaddr, int
    ps_lvl = 1)
```

cioè dobbiamo specificare la tabella madre, gli indirizzi virtuali di confine della regione, una double word contenente eventuali flag R/W o U/S, e una funzione per l'ottenimento sequenziale di indirizzi fisici *di pagina*, che viene passata come *funtore* o come *lambda*

(non si entra nei dettagli dello standard C++, va bene una funzione come una classe che ridefinisce l'operatore di chiamata).

La chiamata della `unmap()` è invece:

```
void unmap(paddr tab, vaddr begin, vaddr end, T& putpaddr)
```

dove vediamo servono principalmente gli stessi parametri, più la funzione `putpaddr()`, che può essere usata per fare pulizia degli indirizzi fisici non più usati.

Ad esempio, se questi vengono posti in memoria dinamica attraverso l'helper passato a `getpaddr` della `map()`, e si sfrutta qualche struttura dati che tiene conto di quali regioni di memoria fisica sono effettivamente in utilizzo, il parametro `putpaddr()` ci permette di definire una funzione che ripulisca i frame utilizzati una volta che questi vengono liberati, liberando le pagine di memoria corrispondenti.

Un'ultima precisazione va fatta riguardo alle funzioni usate da `map()` e `unmap()`: queste saranno la `copy_des()` e la `set_des()` che avevamo visto in 16.1.2, più ottimizzate per la modifica in massa.

17 Lezione del 07-04-25

17.0.1 Implementazione di map() e unmap()

Vediamo brevemente come sono implementate in pratica la `map()` e la `unmap()`.

- `map()`: si basa su una visita *anticipata* del trie:

```
vaddr map(paddr tab, vaddr begin, vaddr end, natl flags, T& getpaddr,
     int ps_lvl = 1)
{
  vaddr v;  /* indirizzo virtuale corrente */
  int l;    /* livello (del TRIE) corrente */
  natq dr;  /* dimensione delle regioni di livello ps_lvl */
  [...]

  // controlli
  [...]

  // usiamo un iteratore di tabella per effettuare una visita
    anticipata
  tab_iter it(tab, begin, end - begin);
  for ( /* niente */ ; it; it.next()) {
    tab_entry& e = it.get_e();
    l = it.get_l();
    v = it.get_v();

    // new verra' popolato dal nuovo indirizzo fisico da collegare a
    una tabella o a una traduzione, indistintamente
    paddr new_f = 0;

    if (l > ps_lvl) {
      // nodo non foglia

      if (!(e & BIT_P)) {
        // va allocata una tabella, chiama alloca_tab()
        new_f = alloca_tab(); // caso 1) si crea una nuova tabella

        // controlli
        [...]

```

```
      } else if (e & BIT_PS) {
        // errore: e' una huge page
      }
    } else {
      // va allocata una traduzione, chiama get_paddr()
      new_f = getpaddr(v);  // caso 2) si crea una nuova traduzione

      // controlli
      [...]

      // configura i flag
      if (l > 1)
        e |= BIT_PS;

      e |= (flags & (BIT_PWT|BIT_PCD));
    }

    if (new_f) {
      // siam qui per il caso 1) o per il caso 2), cioe':
      // - caso 1) bisogna creare una tabella
      // - caso 2) bisogna creare una traduzione
      // in entrambi i casi si fanno le stesse operazioni

      // 'e' non puntava a niente e ora deve puntare a new_f
      set_IND_FISICO(e, new_f);
      e |= BIT_P;

      // dobbiamo incrementare il contatore delle entrate
      // valide della tabella a cui 'e' appartiene
      inc_ref(it.get_tab());
    }

    // configura altri flag
    e |= (flags & (BIT_RW|BIT_US));
  }
  return end;
```

Vediamo quindi che la situazione rispetto all'ultima volta si complica: non abbiamo bisogno soltanto della `getpaddr()`, per l'ottenimento degli indirizzi fisici, ma anche della `alloca_tab()` per l'allocazione di tabelle del trie. Questo ha senso, in quanto la tabella si distingue dal semplice frame di collegato a una pagina, per il fatto che necessita di un contatore di entrate valide che ne facilità la pulizia in caso di inutilizzo. Come vedremo, in ogni caso, sia la `getpaddr()` che la `alloca_tab()` vengono spesso definite, ad esempio nel nucleo, sulla base della stessa funzione helper per l'ottenimento di memoria libera (`allocaframe()`);

- `unmap()`: si basa su una visita *posticipata* del trie:

```
void unmap(paddr tab, vaddr begin, vaddr end, T& putpaddr)
{
  // usiamo un iteratore di tabella per effettuare una visita
    posticipata
  tab_iter it(tab, begin, end - begin);
  for (it.post(); it; it.next_post()) {
    tab_entry& e = it.get_e();

    // non eliminare tabelle non allocate
    if (!(e & BIT_P))
      continue;
```

```
11
12      paddr p = extr_IND_FISICO(e);
13      if (!it.is_leaf()) {
14        // l'entrata punta a una tabella.
15
16        // qui entra in gioco il numero di entrate valide:
17        // la get_ref() ci permete di ottenere le sottotabelle con P
      alto
18
19        if (!get_ref(p)) {
20          // se la tabella non contiene piu' entrate
21          // valide la deallochiamo
22          rilascia_tab(p);
23        } else {
24          // altrimenti non facciamo niente
25          // (la tabella serve per traduzioni esterne
26          // all'intervallo da eliminare)
27          continue;
28        }
29      } else {
30        // l'entrata punta ad una pagina (di livello it.get_l())
31        vaddr v = it.get_v();
32        int l = it.get_l();
33
34        // controlli
35        [...]
36
37        putpaddr(v, p, l);
38      }
39
40      // azzeriamo l'entrata di tabella
41      e = 0;
42      // decrementiamo i riferimenti
43      dec_ref(it.get_tab());
44    }
45  }
```

Vediamo qui ancor meglio come è necessario mantenere separatamente il numero di sottotabelle occupate di una tabella, in modo da capire quando si può procedere alla tabella con `rilascia_tab()`, o quando questa mantiene ancora sottotabelle utili ad altre traduzioni. L'eliminazione delle traduzioni stesse, e quindi delle locazioni fisiche allocate, invece, viene svolta dalla `putpaddr()`.

17.1 Gestione della memoria fisica

Osservando come la `map()` e la `unmap()` hanno bisogno di funzioni (`alloca_tab()` e `dealloca_tab()`, nonché `getpaddr()` e `putpaddr()`, comunque queste siano implementate) che si occupano di ottenere effettivamente memoria fisica. Vediamo come queste vengono implementate.

17.1.1 Descrittori di frame

Ci rendiamo quindi conto di aver bisogno di una struttura dati, contenuta in memoria sistema (M_1), che gestisce i frame di memoria nella parte alta (M_2). Questa struttura è implementata come un'array:

```
1 // descrittore di frame
2 struct des_frame {
```

```
  union {
    // numero di entrate valide (se il frame contiene una tabella)
    natw nvalide;
    // prossimo frame libero (se il frame e' libero)
    natl prossimo_libero;
  };
};

// array dei descrittori di frame
des_frame vdf[N_FRAME];
```

17.1.2 Gestione dei frame

Definiti descrittori di frame, si potrà allocare e deallocare come segue:

- **Allocazione:** prendiamo il primo frame libero, che manteniamo in un apposita variabile (appositamente inizializzata), sostituiamo il suo puntatore a frame libero con il numero di entrate valide nullo, e prendiamo il suo puntatore a prossimo frame libero come nuovo puntatore globale, ovvero:

```
paddr alloca_frame() {
  if (!num_frame_liberi) {
    flog(LOG_ERR, "out of memory");
    return 0;
  }
  natq j = primo_frame_libero;
  primo_frame_libero = vdf[primo_frame_libero].prossimo_libero;
  vdf[j].prossimo_libero = 0;
  num_frame_liberi--;
  return j * DIM_PAGINA;
}
```

- **Deallocazione:** prendiamo il frame come primo frame libero e impostiamo il suo puntatore a prossimo frame libero al puntatore a frame libero corrente, ovvero:

```
void rilascia_frame(paddr f) {
  natq j = f / DIM_PAGINA;
  if (j < N_M1) {
    fpanic("tentativo di rilasciare il frame %lx di M1", f);
  }
  // dal momento che i frame di M2 sono tutti equivalenti, e'
  // sufficiente inserire in testa
  vdf[j].prossimo_libero = primo_frame_libero;
  primo_frame_libero = j;
  num_frame_liberi++;
}
```

17.1.3 Gestione di tabelle

Vorrremo usare le `alloca_frame()` e `rilascia_frame()` per allocare e deallocare intere tabelle di frame, attraverso le funzioni:

- **Allocazione:**

```
paddr alloca_tab() {
  paddr f = alloca_frame();
  if (f) {
    memset(voidptr_cast(f), 0, DIM_PAGINA);
```

```
    vdf[f / DIM_PAGINA].nvalide = 0;
  }
  return f;
}
```

- **Deallocazione:**

```
void rilascia_tab(paddr f) {
  if (int n = get_ref(f)) {
    fpanic("tentativo di deallocare la tabella %lx con %d entrate
    valide", f, n);
  }
  rilascia_frame(f);
}
```

Queste funzioni sono proprio quelle che davamo in argomento a le `map()` e `unmap()` per la gestione del trie. In altre parole, stiamo gestendo l'albero di traduzione attraverso le funzioni `map()` e `unmap()`, le tabelle attraverso le `alloca_tab()` e `rilascia_tab()`, e i frame di memoria fisica attraverso le `alloca_frame()` e `rilascia_frame()`.

17.2 Bootloader

Vediamo quindi il **bootloader**, cioè quella parte del kernel che si occupa di effettuare il *bootstrap* e portare il sistema in uno stato operativo.

Riguardo alla memoria virtuale, avremo che dovremo in sequenza:

1. Creare la radice dell'albero di traduzione;
2. Creare la finestra FM;
3. Prima di attivare la paginazione, caricare l'indirizzo fisico radice dell'albero di traduzione nel registro `CR3`;
4. Attivare la paginazione.

Cosa che facciamo come:

```
// punto 1
paddr root_tab = alloca_tab();
if (!root_tab) {
  flog(LOG_ERR, "ATTENZIONE: impossibile allocare la tabella radice");
  return;
}

// punto 2
if (!crea_finestra_FM(root_tab, mem_tot)) {
  flog(LOG_ERR, "ATTENZIONE: fallimento in crea_finestra_FM()");
  return;
}

// punto 3
loadCR3(root_tab);

// punto 4
// (equivale a comunicare con un interfaccia)
attiva_paginazione(info, info->mod[0].entry_point, MAX_LIV);
```

Potrebbe interessarci l'implementazione della `crea_finestra_FM()`. Questa parte creando una traduzione identità attraverso una *lambda*:

```
auto identity_map = [] (vaddr v) -> paddr { return v; };
```

e quindi mappando diverse regioni di memoria in base al loro scopo:

```
  // prima regione non mappata, interecetta nullptr
  natq first_reg = dim_region(1);

  // [0, DIM_PAGINA): non mappato
  // [DIM_PAGINA, 0xa0000): memoria normale
  if (map(root_tab, DIM_PAGINA, 0xa0000, BIT_RW, identity_map) != 0xa0000)
    return false;
  // [0xa0000, 0xc0000): memoria video
  if (map(root_tab, 0xa0000, 0xc0000, BIT_RW|BIT_PWT, identity_map) != 0
    xc0000)
    return false;
  // [0xc0000, first_reg): memoria normale
  if (map(root_tab, 0xc0000, first_reg, BIT_RW, identity_map) != first_reg
    )
    return false;

  // mappiamo il resto della memoria, se esiste, con PS settato
  if (mem_tot > first_reg) {
    if (map(root_tab, first_reg, mem_tot, BIT_RW, identity_map, 2) !=
    mem_tot)
      return false;
  }

  flog(LOG_INFO, "Creata finestra sulla memoria centrale:  [%16llx, %16llx
    )", DIM_PAGINA, mem_tot);

  // qui la memoria e' finita, tutto quello che segue interessa al bus PCI

  // mappiamo tutti gli altri indirizzi, fino a 4GiB, settando sia PWT che
     PCD.
  // questa zona di indirizzi e' utilizzata in particolare dall'APIC per
    mappare i propri registri.
  vaddr beg_pci = allinea(mem_tot, 2*MiB),
    end_pci = 4*GiB;
  if (map(root_tab, beg_pci, end_pci, BIT_RW|BIT_PCD|BIT_PWT, identity_map
    , 2) != end_pci)
    return false;

  flog(LOG_INFO, "Creata finestra per memory-mapped-IO:    [%16llx, %16llx
    )", beg_pci, end_pci);
  return true;
```

Un dettaglio interessante è nella `attiva_paginazione()`. Questa è scritta in assembler come:

```
# settiamo il bit 31 di CR0
  movl %cr0, %eax
  orl $0x80010000, %eax # paging & write-protect
  movl %eax, %cr0
  # da qui in poi la MMU e' attiva
```

Visto che dall'esecuzione della `MOVL` in poi il processore emetterà indirizzi che verrano tradoti dalla MMU, sarà necessario che l'indirizzo puntato in quel momento dal RIP sia contenuto nella finestra creata prima, così che si mantenga la continuità fra le istruzioni del programma.

17.3 Partizione della memoria nel nucleo

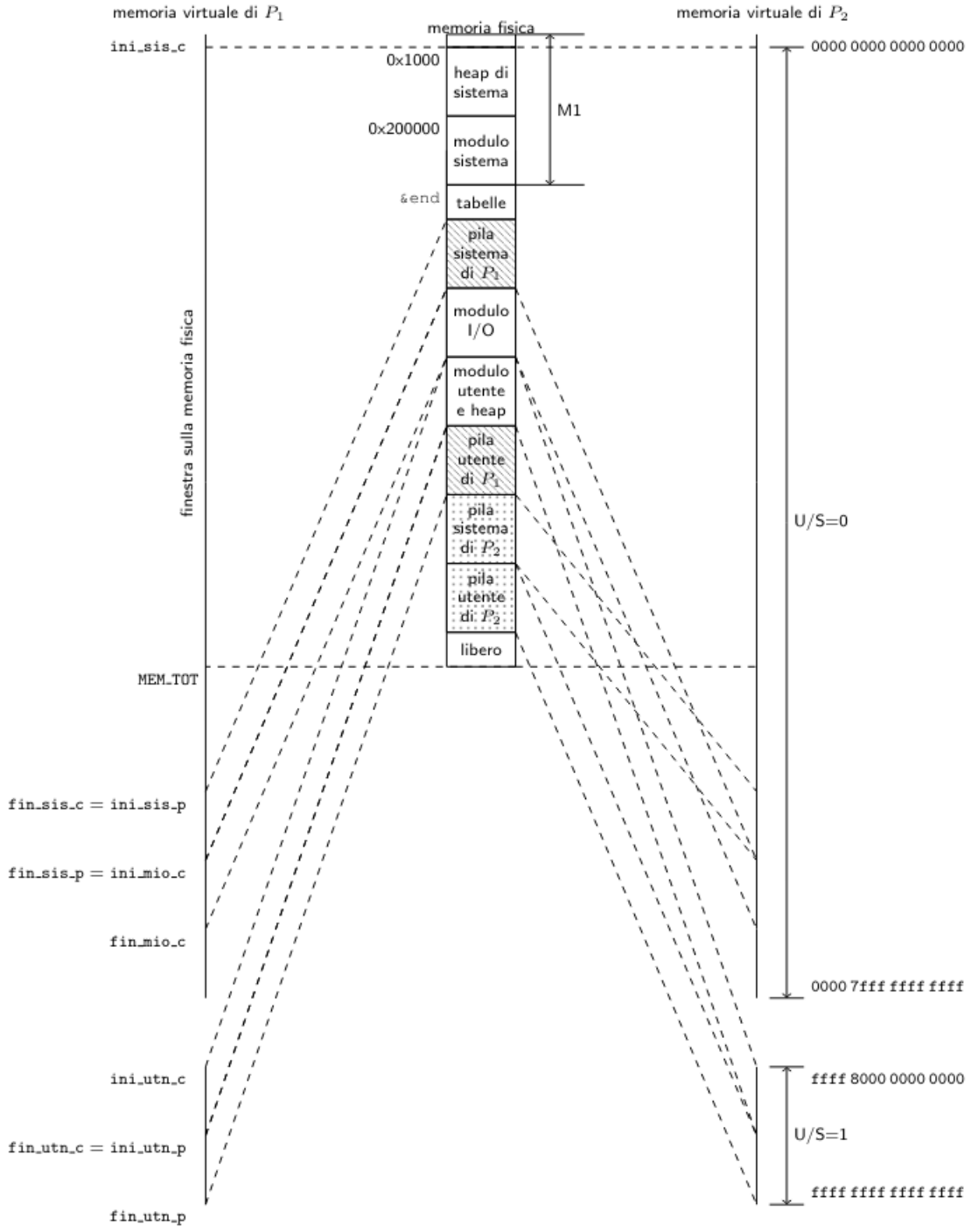
Abbiamo quindi visto come la memoria indirizzabile è divisa in due regioni da 2^{47} bit ciascuna (cioè la divisione data dagli indirizzi a 48 bit normalizzati). Vediamo come questa memoria è divisa nel nucleo. Abbiamo che nella regione bassa allochiamo memoria sistema, come segue:

- **Memoria sistema:**
  - **Memoria sistema condivisa**: qui si manterranno informazioni riguardo ai frame della memoria M_2 (quella al di sopra dela partizione), e alle tabelle, in un apposita struttura dati (un array). La struttura dati contiene, fra l'altro, anche il contatore delle entrate valide di ogni tabella (che abbiamo visto prima viene consultato dalla `unmap()` per effettuare rimozioni di tabelle);
  - **Memoria sistema privata** (**pila sistema**);
  - **Memoria sistema del modulo I/O condivisa**.

La parte alta alloca invece memoria utente, come segue:

- **Memoria utente:**
  - **Memoria utente condivisa** (**codice** e **heap**): questa è organizzata perché qualsiasi processo in esecuzione la mappi sempre nella stessa regione;
  - **Memoria utente privata** (**pila utente**): questa è organizzata perché ogni processo mappi la *sua* pila nella stessa regione.

Lo schema complessivo è quindi del tipo seguente:



in questo caso relativa ai trie di due processi, da cui notiamo ancora come la prima pagina non è tradotta, in quanto rappresenta l'area raggiungibile dai nullptr.

Facciamo allora alcune semplificazioni riguardo a questa struttura, in modo da avvicinarci ad un'implementazione effettiva:

- Tutte le parti di livello più alte vengono create come multipli di 512 GiB, in modo che occupino intere entrate di livello 4;

- Le parti condivise sono *"fisse"*, riferite da tabelle di livello 3 che vengono puntate nuovamente in ogni tabella di livello 4 che creiamo come radice degli alberi di traduzione di ogni processo (e che sono le stesse dell'albero di traduzione del nucleo).

Vediamo che le tabelle di livello 3 della parte utente e sistema condivise si possono quindi creare una volta sola all'avvio del sistema (si dovrebbe ricaricare la parte codice dei processi nel caso di un sistema che carica software dal disco). Questa allocazione viene fatta usando la `map()` aiutata da `alloca_frame()`, per ottenere memoria e disporre traduzioni per regioni di memoria di indirizzi prestabiliti. Queste regioni seguono lo schema visto finora, che viene definito in codice come:

```
#define I_SIS_C   0   // prima entrata sistema/condivisa (inizio M1)
#define I_SIS_P   1   // prima entrata sistema/privata
#define I_MIO_C   2   // prima entrata modulo IO/condivisa
#define I_UTN_C   256 // prima entrata utente/condivisa (inizio M2)
#define I_UTN_P   384 // prima entrata utente/privata

#define N_SIS_C   1   // numero entrate sistema/condivisa
#define N_SIS_P   1   // numero entrate sistema/privata
#define N_MIO_C   1   // numero entrate modulo IO/condivisa
#define N_UTN_C   128 // numero entrate utente/convidisa
#define N_UTN_P   128 // numero entrate utente/privata
```

Le uniche cose che vanno quindi create da zero ogni volta che si crea un processo sono la **pila sistema** in memoria sistema privata e la **pila utente** in memoria utente privata.

Avremo quindi che alla creazione di un nuovo processo dovremo creare una *nuova* tabella di livello 4, che puntera alle tabelle di livello 3 delle parti condivise (memoria utente e sistema condivisa), già esistenti, e che creerà nuove tabelle di livello 3, e quindi di livello 2, ecc... per le parti private (pila utente e pila sistema).

17.3.1 Sequenza di boot

Possiamo quindi vedere nel dettaglio la sequenza di boot e avvio del kernel, anticipando alcuni dettagli sull I/O che verranno meglio spiegati nella sezione 18. Abbiamo che il processo di boot si autodocumenta stampando messaggi sulla porta seriale (attraverso la funzione di `libce.h` chiamata `flog()`). Vediamo quindi l'intero processo dividendolo in fasi e controllando cosa viene stampato.

1. **Bootloader:**

```
INF - Boot loader di Calcolatori Elettronici, v1.0
INF - Memoria totale: 32 MiB, heap: 636 KiB
INF - Argomenti: ~/CE/lib/ce/boot.bin
INF - Il boot loader precedente ha caricato 3 moduli:
INF - - mod[0]: start=10d000 end=117ba0 file=boot/0-sistema
INF -   . seg[1]: off      0 vaddr           200000 size   6973
    memsize   6973
INF -   . seg[2]: off   7000 vaddr           207000 size   2f24
    memsize   2f24
INF -   . seg[3]: off   a000 vaddr           20a000 size    180
    memsize  17404
INF - - mod[1]: start=118000 end=11e5d8 file=boot/1-io
INF -   . seg[1]: off      0 vaddr      10000000000 size   3ddf
    memsize   3ddf
```

```
11 INF -  . seg[2]: off   4000 vaddr      10000004000 size   19ac
       memsize   19ac
12 INF -  . seg[3]: off   6000 vaddr      10000006000 size    190
       memsize    200
13 INF - - mod[2]: start=11f000 end=1224b0 file=boot/2-utente
14 INF -  . seg[1]: off      0 vaddr ffff800000000000 size   1a69
       memsize   1a69
15 INF -  . seg[2]: off   2000 vaddr ffff800000002000 size    974
       memsize    974
16 INF -  . seg[3]: off   3000 vaddr ffff800000003000 size     60
       memsize   1690
17 INF - Copio mod[0] agli indirizzi specificati nel file ELF:
18 INF - - copiati   6973 byte da   10d000 a   200000
19 INF - - copiati   2f24 byte da   114000 a   207000
20 INF - - copiati    180 byte da   117000 a   20a000
21 INF - - azzerati ulteriori 17284 byte
22 INF - - entry point 20569b
23 INF - Creata finestra sulla memoria centrale:  [            1000,
            2000000)
24 INF - Creata finestra per memory-mapped-IO:    [         2000000,
          100000000)
25 INF - Inizializzo l'APIC
26 INF - Inizializzo video e tastiera
27 INF - Attivo la modalita' a 64 bit e cedo il controllo a mod[0]...
```

Inizia il bootloader, definito in `libce.h`. Questo non è il primo bootloader che viene lanciato sulla macchina virtuale: QEMU dispone infatti del suo bootloader, che si occupa di ottenere informazioni di base sulla memoria (che vengono visualizzate alla riga 2) e caricare dal disco le immagini dei moduli da caricare (riga 3).

Questi moduli sono, nel dettaglio:

- Il modulo sistema, contenuto in `mod[0]`;
- Il modulo di I/O, contenuto in `mod[1]`;
- Il modulo utente, contenuto in `mod[2]`.

Ogni modulo si porta dietro le sue sezioni definite nell'**ELF** (*Executable and Linkable Format*), divise in 3 *segmenti*:

1) Il primo segmento è di tipo **testo** e contiene il codice del modulo, quindi va caricato in sola lettura;
2) Il secondo segmento è di tipo **dati in sola lettura**, quindi va caricato in sola lettura;
3) Il terzo segmento è di tipo **dati** (variabili inizializzate, fra cui lo spazio da dedicare a tutte le varie tabelle di descrittori, da cui la grande quantità di spazio `memsize` da liberare senza scrivere), quindi va caricato in lettura/scrittura.

Il bootloader si occupa quindi di caricare solo il primo modulo, quello sistema, da `200000` in poi (righe 17-22), in quanto non sa come questo vorrà gestire I/O e utente.

Procede quindi alla creazione della finestra FM, come abbiamo visto dalla `crea_finestra_FM()` (righe 23 e 24). Infine, inizializza l'APIC alla riga 25, video e tastiera alla riga 26, semplicemente configurando i registri giusti (ad esempio attiva le interruzioni tastiera e disattiva il PIC), attiva la paginazione e cede il controllo a `mod[0]`, cioè al modulo sistema, alla riga 27.

2. **Kernel:**

```
1  INF 0 Nucleo di Calcolatori Elettronici, v8.2
2  INF 0 Numero di frame: 546 (M1) 7646 (M2)
3  INF 0 Suddivisione della memoria virtuale:
4  INF 0 - sis/cond [               0,       8000000000)
5  INF 0 - sis/priv [      8000000000,      10000000000)
6  INF 0 - io /cond [     10000000000,      18000000000)
7  INF 0 - usr/cond [ffff800000000000, ffffc00000000000)
8  INF 0 - usr/priv [ffffc00000000000,                0)
9  INF 0 Carico il modulo I/O
10 INF 0 - segmento sistema read-only  mappato a [     10000000000,
       10000004000)
11 INF 0 - segmento sistema read-only  mappato a [     10000004000,
       10000006000)
12 INF 0 - segmento sistema read/write mappato a [     10000006000,
       10000007000)
13 INF 0 - heap:                                 [     10000007000,
       10000107000)
14 INF 0 - entry point: _start [io.s:17]
15 INF 0 Carico il modulo utente
16 INF 0 - segmento utente  read-only  mappato a [ffff800000000000,
       ffff800000002000)
17 INF 0 - segmento utente  read-only  mappato a [ffff800000002000,
       ffff800000003000)
18 INF 0 - segmento utente  read/write mappato a [ffff800000003000,
       ffff800000005000)
19 INF 0 - heap:                                 [ffff800000005000,
       ffff800000105000)
20 INF 0 - entry point: _start [utente.s:15]
21 INF 0 Frame liberi: 7116 (M2)
22 INF 0 Heap del modulo sistema: aggiunto [100000, 200000)
23 INF 0 Attivo il timer (DELAY=59659)
24 INF 0 Creo il processo main I/O
25 INF 0 Attendo inizializzazione modulo I/O...
```

Il kernel prende quindi il controllo della macchina. A questo punto ha accesso alla finestra FM, quindi può controllare il numero di frame liberi e creare la partizione di memoria è l'albero sistema come definito in 17.3 (righe da 2 a 8). Notiamo come le regione che crea coincidono con la dimensione delle entrate di livello 4 (512 GiB ciascuna), Inoltre, notiamo il salto alla prima tabella utente, da M_1 a M_2 (indirizzi che iniziano con `0xffff...`).

Si inizia quindi il caricamento del modulo I/O (riga 9), mappato nella regione I/O condivisa, e del modulo utente (riga 15), mappato nella memoria utente condivisa.

Si sceglie poi l'entry point del modulo utente (riga 20), si aumenta la dimensione dell'heap sistema (inizializzato dal primo bootloader) (riga 22), si attiva il timer (riga 23) e si crea il processo I/O (riga 24), che da qui in poi prenderà il controllo.

3. **I/O:**

```
1 INF 1 Heap del modulo I/O: 100000B [0x10000007000, 0x10000107000)
2 INF 1 Inizializzo la console (kbd + vid)
3 INF 1 estern=2 entry=estern_kbd(unsigned long) [io.cpp:197](0) prio
      =1104 (tipo=50) liv=0 irq=1
4 INF 1 kbd: tastiera inizializzata
5 INF 1 vid: video inizializzato
6 INF 1 Inizializzo la gestione dell'hard disk
7 INF 1 bm: 00:01.1
```

```
8 INF 1 estern=3 entry=estern_hd(unsigned long) [io.cpp:557](0) prio
    =1120 (tipo=60) liv=0 irq=14
9 INF 1 Processo 1 terminato
```

Il modulo di I/O fa un pò di configurazione aggiuntiva: inizializza la console avviando il processo `estern_kbd` (riga 3). In seguito, deve *trovare* l'unico dispositivo finora non considerato, cioè l'hard disk (riga 6) che trova sul bus principale al dispositivo 1, funzione 1 (vedremo in seguito che questo corrisponde al bus PCI-ATA del PIIX3). Una volta trovato l'hard disk, lo attiva gestendolo col processo `estern_hd` (riga 8).

In seguito, termina e restituisce il controllo al kernel.

4. **Utente:**

```
1 INF 0 Creo il processo main utente
2 INF 0 Cedo il controllo al processo main utente...
3 INF 4 Heap del modulo utente: 100000B [0xffff800000004690, 0
    xffff800000104690)
```

L'ultimo a prendere il controllo è l'utente, attraverso il kernel che lo attiva dall'entry point trovato prima. Da qui in poi ad eseguire è il codice che definiamo noi, interfacciandosi con le primitive di kernel e i processi di I/O.

17.4 Creazione di processi in memoria

Possiamo quindi vedere più nel dettaglio la creazione di processi, in particolare riguardo alla memoria e alle tabelle e traduzioni create.

17.4.1 Albero di traduzione

Abbiamo detto avevamo bisogno di creare nuove tabelle di livello 4 per ogni processo. Facciamo questo come segue:

```
des_proc* crea_processo(void f(natq), natq a, int prio, char liv) {
  [...]

  p->cr3 = alloca_tab(); // la nuova tabella di livello 4
  if (p->cr3 == 0)
    goto err_rel_id;
  init_root_tab(p->cr3);

  [...]
}
```

dove la `init_root_tab()`, come avevamo detto, si limita a copiare le tabelle di livello 3 delle parti condivise:

```
void init_root_tab(paddr dest) {
  // cr3 del processo corrente
  paddr pdir = esecuzione->cr3;

  // copia le tabelle di livello 3
  copy_des(pdir, dest, I_SIS_C, N_SIS_C);
  copy_des(pdir, dest, I_MIO_C, N_MIO_C);
  copy_des(pdir, dest, I_UTN_C, N_UTN_C);
}
```

Questa ha una duale, che semplicemente libera le entrate create:

```
void clear_root_tab(paddr dest) {
  // eliminiamo le entrate create da init_root_tab()
  set_des(dest, I_SIS_C, N_SIS_C, 0);
  set_des(dest, I_MIO_C, N_MIO_C, 0);
  set_des(dest, I_UTN_C, N_UTN_C, 0);
}
```

17.4.2 Pila

Veniamo quindi all'inizializzazione della pila. Questa si fa, sia per la pila utente che per la pila sistema, attraverso la `crea_pila()`:

```
bool crea_pila(paddr root_tab, vaddr bottom, natq size, natl liv)
{
  vaddr v = map(root_tab,
    bottom - size,
    bottom,
    BIT_RW | (liv == LIV_UTENTE ? BIT_US : 0),
    [](vaddr) { return alloca_frame(); });

  // caso di errore
  if (v != bottom) {
    unmap(root_tab, bottom - size, v,
      [](vaddr, paddr p, int) { rilascia_frame(p); });
    return false;
  }
  return true;
}
```

che ottiene una pila di una dimensione prestabilita allocando i frame necessari.

Questa ha ancora una duale, `distruggi_pila()`:

```
void distruggi_pila(paddr root_tab, vaddr bottom, natq size) {
  unmap(
    root_tab,
    bottom - size,
    bottom,
    [](vaddr, paddr p, int) { rilascia_frame(p); });
}
```

che si limita a liberare i frame usati.

Dal punto di vista della `crea_processo()`, quindi, vogliamo prima inizializzare la pila sistema, e poi:

- Se siamo in contesto utente:
  1. Inizializzare la pila sistema;
  2. Creare la pila utente;
  3. Inizializzare la pila utente.
- Se invece siamo in contesto sistema, ci limitiamo ad inizializzare la pila sistema.

Questo in codice si traduce come:

```
// creazione della pila sistema
static_assert(DIM_SYS_STACK > 0 && (DIM_SYS_STACK & 0xFFF) == 0);

// siamo in un altro processo, quindi dobbiamo accedere alla pila sistema
    tramite la finestra FM
```

```
pila_sistema = trasforma(p->cr3, fin_sis_p - 1) + 1;

// convertiamo a puntatore a natq, per accedervi piu' comodamente
pl = ptr_cast<natq>(pila_sistema);

if (liv == LIV_UTENTE) {
  // processo di livello utente
  // inizializzazione della pila sistema
  pl[-5] = int_cast<natq>(f);          // RIP (codice utente)
  pl[-4] = SEL_CODICE_UTENTE;          // CS (codice utente)
  pl[-3] = BIT_IF;                     // RFLAGS
  pl[-2] = fin_utn_p - sizeof(natq);   // RSP
  pl[-1] = SEL_DATI_UTENTE;            // SS (pila utente)

  // eseguendo una IRET da questa situazione, il processo
  // passera' ad eseguire la prima istruzione della funzione f,
  // usando come pila la pila utente (al suo indirizzo virtuale)

  // creazione della pila utente
  static_assert(DIM_USR_STACK > 0 && (DIM_USR_STACK & 0xFFF) == 0);

  // inizialmente, il processo si trova a livello sistema, come
  // se avesse eseguito una istruzione INT, con la pila sistema
  // che contiene le 5 parole quadruple preparate precedentemente
  p->contesto[I_RSP] = fin_sis_p - 5 * sizeof(natq);

  p->livello = LIV_UTENTE;

  // dal momento che usiamo traduzioni diverse per le parti sistema/
    private
  // di tutti i processi, possiamo inizializzare p->punt_nucleo con un
  // indirizzo (virtuale) uguale per tutti i processi
  p->punt_nucleo = fin_sis_p;

  // tutti gli altri campi valgono 0
} else {
  // processo di livello sistema
  // inizializzazione della pila sistema
  pl[-6] = int_cast<natq>(f);          // RIP (codice sistema)
  pl[-5] = SEL_CODICE_SISTEMA;         // CS (codice sistema)
  pl[-4] = BIT_IF;                     // RFLAGS
  pl[-3] = fin_sis_p - sizeof(natq);   // RSP
  pl[-2] = 0;                          // SS
  pl[-1] = 0;                          // ind. rit.

  // inizializzazione del descrittore di processo
  p->contesto[I_RSP] = fin_sis_p - 6 * sizeof(natq);

  p->livello = LIV_SISTEMA;

  // tutti gli altri campi valgono 0
}
```

18 Lezione del 08-04-25

Concluso il discorso sulla memoria virtuale, ci concentreremo nuovamente sull'hardware, nella prospettiva di approfondire l'interazione fra nucleo e dispositivi di I/O.

18.1 Bus PCI

Il bus **PCI**, che sta per *Peripheral Component Interconnect*, è uno standard per bus sviluppato da IBM per consentire l'espansione dei loro calcolatori attraverso schede apposite, supportando quindi una cosiddetta *architettura aperta*.

Possiamo quindi immaginare che ogni scheda di espansione sia provvista dei suoi registri, delle sue interruzioni, ecc... che non devono sovrapporsi con quelli di altre schede. Storicamente, questo rappresentava un problema, in quanto potevano crearsi *conflitti* fra più schede.

Inoltre, un problema era rappresentato dai *driver*, in quanto non esisteva un modo standardizzato per rilevare se una certa scheda era installata o no, e quindi se si poteva usare un certo driver.

I produttori stabilirono quindi una sorta di standard *de facto*, che abbiamo già nominato: l'**ISA** (*Industry Standard Architecture*).

L'idea fondamentale è che la scheda non può avere registri fissi, ma deve essere programmabile in questo dal calcolatore. Inoltre, deve esistere una qualche modalità per rilevare le schede correntemente installate nel sistema.

Nei PC moderni sfruttiamo uno standard di derivazione dal vecchio PCI, compatibile con esso, che è il *PCI Express* (e che non studieremo).

18.1.1 Indirizzamento dei dispositivi

Secondo lo standard PCI, separiamo il **bus locale** (quello che abbiamo visto finora) da un eventuale **albero di bus**, collegati fra di loro dai cosiddetti **ponti**. Il **ponte ospite-PCI**, in particolare, collega il **bus principale** (il più vicino al bus locale) al bus locale, mentre questo a sua volta viene collegato ad altri bus attraverso **ponti PCI-PCI**. Ad esempio, molti calcolatori dell'epoca erano dotati di *bus ISA* collegati con appositi ponti al bus principale, per la gestione di vecchie interfacce ISA.

A ogni bus è associato un numero su 8 bit, col bus principale che si prende il numero 0.

Per indirizzare un dispositivo usiamo invece 16 bit, disposti come:

Scopo	**Bit**	**Max**
Bus	8 bit	256
Device	5 bit	32
Function	3 bit	8

dove il numero di bus è lo stesso di prima.

Il numero di dispositivo differisce dal *Device ID*, che vedremo fra poco, e deriva dalla posizione fisica del dispositivo nel bus.

Il numero di funzione, invece, è reso necessario da schede che implementano più funzionalità, quali ad esempio le schede grafiche moderne, che si occupano anche dell'audio. In ogni caso, la funzione 0 deve essere implementata obbligatoriamente.

Come vediamo dalla tabella, poi, si possono avere direttamente dalle codifiche fino a 256 bus diversi, con 32 dispositivi ciascuno e 8 funzioni per dispositivo.

18.1.2 Operazioni coi dispositivi

Veniamo quindi a come funzionano le operazioni più semplici sul Bus PCI. Ogni operazione sul PCI viene detta **transazione**, ed ha un **iniziatore** e un **obiettivo**, cioè il dispositivo che inizia la transazione e il dispositivo che gli risponde.

L'iniziatore delle richieste che il nostro programma invia all'I/O, notiamo, non sarà più il processore, ma il ponte ospite-PCI. Inoltre, il bus PCI permette in verità che anche dispositivi esterni al ponte ospite-PCI facciano da iniziatori per transazioni, rendendo possibili meccanismi come il **DMA** (*Direct Memory Access*).

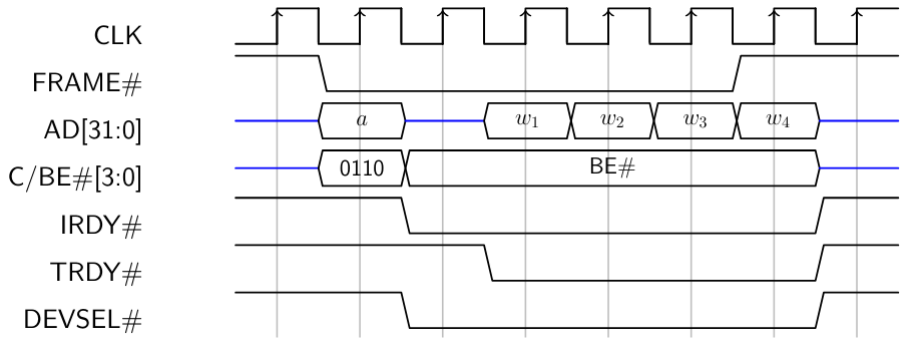
Sul bus vero e proprio troviamo quindi:

- La linea di **clock**, che tutte le interfacce vedono, originariamente intorno ai 33 MHz. Sul fronte di salita del clock tutte le interfacce (idealmente) campionano i segnali sul bus;
- **FRAME#**, che "incornicia" la transazione corrente: l'iniziatore lo alza quando la transazione è finita. Al massimo, a handshake avvenuto si possono avere un numero indefinito di cili di trasmissione da 4 byte: il FRAME si alza ad avvenuta trasmissione dell'ultimo fra questi;
- **AD**, la linea *condivisa* di indirizzo o dati, a 32 bit;
- **C/BE#**, *controllo* e *byte-enable*, codificano il tipo di operazione in fase di indirizzamento e fanno da byte-enable nel trasferimento dati;
- **IRDY#** e **TRDY#**, rispettivamente *Initiator Ready* e *Target Ready*, supportano l'handshake nella fase di scambio dati: l'iniziatore abbassa IRDY quando è pronto a ricevere dati o quando inizia a scriverli, mentre l'obiettivo abbassa TRDY quando inizia a inviare dati o a riceverli;
- **DEVSEL#**, viene attivato dal dispositivo che riconosce, controllando C/BE e l'indirizzo su AD, una chiamata a sé stesso, quindi dal presunto *obiettivo*;
- **STOP#**, viene attivato dall'*obiettivo* per terminare prematuramente una transazione.

dove il # indica *attivi bassi*.

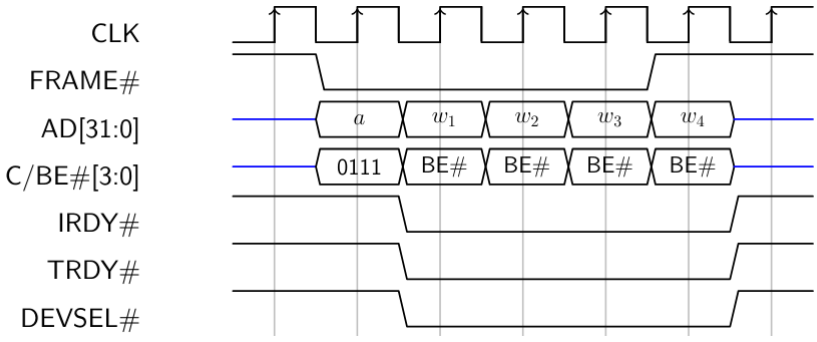
Vediamo nel dettaglio due transazioni complete, di lettura e scrittura a 4 byte sequenziali, per capire a pieno la politica di handshake:

- **Lettura:**



Abbiamo che l'iniziatore come prima cosa porta il FRAME in basso e scrive l'indirizzo in AD e la modalità di indirizzamento in C/BE (qui `0110`, che significa memoria). L'obiettivo risponde abbassando DEVSEL, al cui l'iniziatore risponde abbassando IRDY. Fra l'abbassamento di IRDY e l'inizio della trasmissione (l'abbassamento di TRDY) c'è un ciclo di clock a vuoto, detto *ciclo di turnaround*, che permette all'iniziatore di rilasciare AD e C/BE perchè l'obiettivo possa assumerne il controllo. Si trasmettono quindi 16 byte in mandate da 4, con FRAME che torna alto al quarto byte per segnalare la fine dell'operazione, e le linee di handshake (IRDY, TRDY e DEVSEL) tornano alte.

- **Scrittura:**



Le cose si svolgono in maniera pressoché identica, con la differenza che non si necessita di un ciclo di turnaround in quanto l'iniziatore sarà l'unico a scrivere su AD e C/BE. Inoltre, possiamo immaginare che C/BE venga effettivamente modificato nel corso della scrittura, in quanto magari non vogliamo impattare tutti i byte ad ogni ciclo di scrittura.

18.1.3 Spazio di configurazione

Per poter lavorare con il bus PCI, poi, introduciamo un nuovo spazio indirizzabile, quello di **configurazione**. Avremo quindi che il processore può indirizzare:

- **Memoria**;

- **I/O**;
- **Configurazione**.

Questo spazio è rilevante solo all'avvio del calcolatore (all'esecuzione del BIOS, o come definito originariamente dallo standard, *PCI BIOS*), appunto per effettuare la configurazione delle interfacce PCI installate nel sistema. Ogni bus PCI trasporta quindi messaggi in uno qualsiasi di questi 3 spazi, e si discrimina lo spazio specifico controllando cosa l'iniziatore mette in C/BE.

Useremo lo spazio di I/O come sempre, ma i segnali del processore viaggeranno attraverso i vari ponti fino all'interfaccia desiderata. Per quanto riguarda i dispositivi mappati in memoria (si pensi al video), invece, possiamo assumere che all'avvio il ponte ospite-PCI deve solo sapere la dimensione della memoria RAM installata, in modo da rispondere da lì in poi solo agli indirizzi *di memoria* posti al di *sotto* di essa. Avevamo visto questo meccanismo nella sezione 17.2, vedendo come il bootloader ignora la zona di memoria superiore alla RAM e dedicata al bus PCI.

Notiamo che tutto questo sistema è comunque strutturato per essere trasparente al processore, e quindi invisibile lato software.

18.1.4 Configurazione dei dispositivi

Ogni dispositivo sul bus è obbligato a fornire, per ogni funzione e ad una certa locazione predefinita, un numero di registri che formano 64 righe da 32 bit, che devono contenere informazioni di configurazione. I primi due dati, su 16 bit (quindi una riga), saranno il **Vendor ID** e il **Device ID**, seguiti da altri dati che non ci sono immediatamente rilevanti:

<table>
<tr><th>+3</th><th>+2</th><th>+1</th><th>+0</th><th>offset</th></tr>
<tr><td colspan="2">Device ID</td><td colspan="2">Vendor ID</td><td>0x00</td></tr>
<tr><td colspan="2">Status</td><td colspan="2">Command</td><td>0x04</td></tr>
<tr><td colspan="3">Class Code</td><td>Revision ID</td><td>0x08</td></tr>
<tr><td>...</td><td>Header Type</td><td>...</td><td>Cache Line Size</td><td>0x0c</td></tr>
<tr><td colspan="4">Base Address Register 0</td><td>0x10</td></tr>
<tr><td colspan="4">Base Address Register 1</td><td>0x14</td></tr>
<tr><td colspan="4">Base Address Register 2</td><td>0x18</td></tr>
<tr><td colspan="4">Base Address Register 3</td><td>0x1c</td></tr>
<tr><td colspan="4">Base Address Register 4</td><td>0x20</td></tr>
<tr><td colspan="4">Base Address Register 5</td><td>0x24</td></tr>
<tr><td colspan="4">...</td><td>0x28</td></tr>
<tr><td colspan="2">...</td><td colspan="2">...</td><td>0x2c</td></tr>
<tr><td colspan="4">...</td><td>0x30</td></tr>
<tr><td colspan="3">...</td><td>...</td><td>0x34</td></tr>
<tr><td colspan="4">...</td><td>0x38</td></tr>
<tr><td>...</td><td>...</td><td>Intr. Pin</td><td>Intr. Line</td><td>0x3c</td></tr>
</table>

Il Vendor ID è determinato da un'autorità centrale (la *PCI-SIG*): ad esempio, il vendor ID della Intel è `0x8086`.

Per permettere quindi alla CPU di configurare i dispositivi, cioè accedere ai loro registri di configurazione, il ponte ospite-PCI rende disponibili alla CPU due registri, entrambi su 32 bit:

- Il **CAP**, *Configuration Address Port*, che permette di selezionare una funzione e l'offset della parola a cui si vuole accedere;
- Il **CDP**, *Configuration Data Port*, che permette di accedere alla parola selezionata con *CAP*.

Le operazioni effettuate dalla CPU attraverso questi due registri verranno trasformate automaticamente dal ponte ospite-PCI in operazioni di configurazione sui bus PCI.

La posizione di questi registri in memoria è la seguente:

`0xcf8`	**CAP**, *Configuration Address Port*
`0xcfc`	**CDP**, *Configuration Data Port*

Vediamo quindi cosa deve fare il BIOS per la configurazione dei dispositivi PCI, cioè per collocarne nello spazio di I/O o in memoria eventuali registri o porzioni di memoria, rispettivamente. Ogni dispositivo ha una **dimensione naturale** che occupa nello spazio, sia questo di memoria o di I/O. Fornisce quindi al processore un registro, detto **BAR** (*Base Address Register*), che è scrivibile solo in parte: la parte meno significativa, infatti, è fissa a 0 e determina la dimensione naturale della regione che questo occuperà. Per rilevare la dimensione naturale, quindi, basta scrivere tutti 1 sul BAR e controllare quali bit vengono effettivamente modificati.

Il PCI BIOS dovrà quindi, attraverso i registri CAP e CDP, controllare il BAR di tutte le interfacce, determinarne la dimensione naturale e trovare una regione libera nello spazio di I/O o in memoria, a seconda del tipo di dispositivo, dove collocarle.

18.1.5 Interruzioni PCI

Per la gestione delle interruzioni, lo standard si ferma al dire che ogni dispositivo deve specificare per ogni funzione quale, di quattro linee di interruzione, tale funzione usa, in un apposito registro di configurazione. Queste linee sono dette **INTA**, **INTB**, **INTC** e **INTD**, e lo spazio di configurazione della funzione contiene il dato rispetto a quale linea usa in *Intr. Pin*, con 0 che significa nessuna interruzione, 1 *INTA*, e così via.

18.1.6 Struttura di un bus PCI

Per fissare i concetti di quest'ultima sezione, cerchiamo di esporre la struttura del calcolatore emulato su cui abbiamo studiato finora. Scriviamo quindi un programma che cerchi tutti i dispositivi presenti nella configurazione attuale, sfruttando le funzioni per la gestione dello spazio di configurazione definite in `libce.h` (`read_conf...()`):

```
1 #include <libce.h>
2
3 bool check_dev(natb bus, natb dev, natb fun) {
4   // entrate tabella configurazione
5   natw vendorID, deviceID;
6   natb class_code;
7
```

```
  // ottieni vendor ID
  vendorID = pci::read_confw(bus, dev, fun, 0);

  // 0xffff significa inesistente
  if(vendorID == 0xffff) return false;

  // ottieni device ID e codice classe
  deviceID = pci::read_confw(bus, dev, fun, 2);
  class_code = pci::read_confb(bus, dev, fun, 11);

  // stampa informazioni dispositivo
  printf("%02x:%02x.%1d   %04x:%04x [%s]\n",
    bus, dev, 0,
    vendorID, deviceID,
    pci::decode_class(class_code)); // decodifica il codice classe

  return true;
}

void main() {
  for(natb bus = 0; bus < 100; bus++) { // 100 bastano
    for(natb dev = 0; dev < 32; dev++) {
      // cerchiamo il dispostivo bus:dev:xxx
      if(!check_dev(bus, dev, 0)) continue;

      // controlla le altre funzioni
      for(natb fun = 1; fun < 8; fun++) {
        check_dev(bus, dev, fun);
      }
    }
  }

  pause();
}
```

e vediamo cosa stampa:

```
00:00.0   8086:1237 [bridge device]             % ponte ospite-PCI
00:01.0   8086:7000 [bridge device]             % ponte PCI-ISA
00:01.1   8086:7010 [mass storage controller]   % ponte PCI-ATA
00:01.3   8086:7113 [bridge device]             % altre funzioni PIIX3
00:02.0   1234:1111 [display controller]        % scheda VGA virtuale
```

dove i commenti sono stati aggunti successivamente.

Vediamo quindi che abbiamo 3 dispositivi, tutti sul bus 0 (bus principale):

0: Il ponte ospite-PCI;

1: Un dispositivo che fornisce altri bus, che possiamo individuare nel *southbridge* Intel **PIIX3** (*PCI IDE ISA Xcelerator*). Questi sono:

   0: Un bus ISA accessibile attraverso il ponte PCI-ISA, che emula le interfacce di tastiera e di timer dell'ISA;

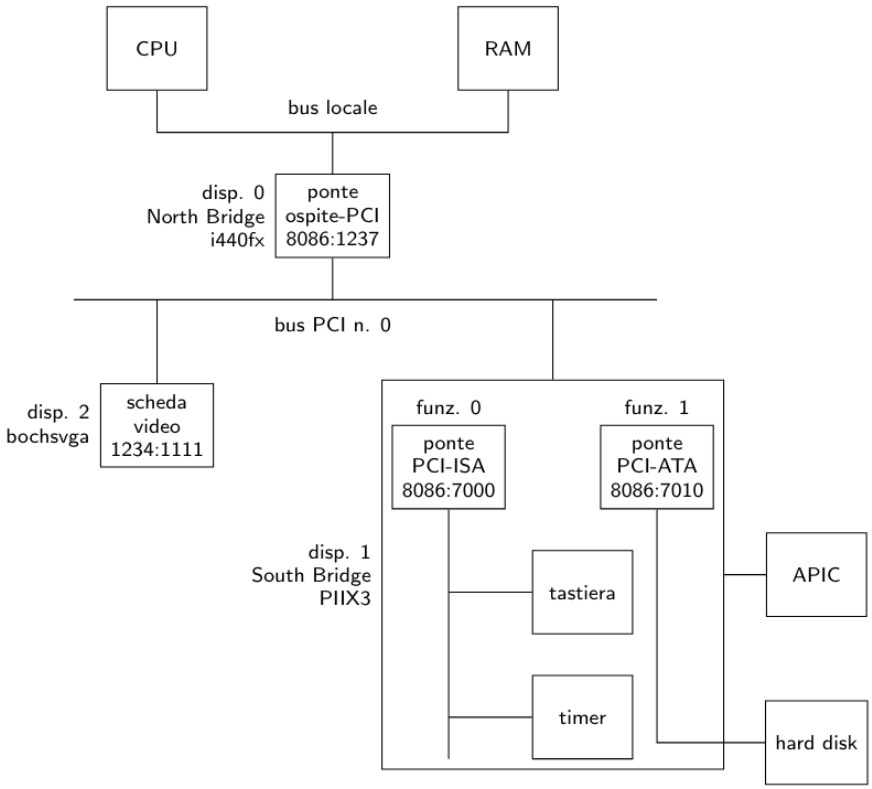
   2: Un bus ATA accessibile attraverso il ponte PCI-ATA, che permette l'interazione con il disco rigido (hard disk);

   3: L'interfaccia **ACPI** (*Advanced Configuration and Power Interface*), non di interesse a questo corso.

Il southbridge si interfaccia poi con altri componenti, quali ad esempio l'APIC (ed emula il PIC sul bus ISA).

2: La scheda video VGA emulata di *bochs*, un emulatore precedente a QEMU.

La struttura complessiva sarà quindi la seguente:



18.2 I/O nel kernel

La maggior parte delle modifiche allo spazio di I/O apportate introducendo il bus PCI, abbiamo detto, sono effettivamente trasparenti al processore, se non per l'introduzione dei registri CAP e CDP, che devono essere comunque usati una volta sola nel PCI BIOS per la configurazione dei dispositivi. Possiamo quindi sfruttare la maggior parte delle funzioni definite in `libce.h` per l'input/output senza particolari problemi.

Vediamo quindi com'è implementata la gestione dell'I/O da parte del *kernel*.

Avevamo detto che la motivazione principale dietro lo sviluppo del sistema multiprogrammato era che questa permetteva l'interruzione in qualsiasi momento di un *processo* in esecuzione, per permettere al processore di effettuare altre operazioni. Le interruzioni sollevate da i dispositivi esterni, vediamo, possono essere gli eventi che provocano tale cambio di contesto, e quindi la gestione del segnale in ingresso al sistema.

Ad esempio, un processo potrebbe, con la funzione `readconsole()`, specificare un buffer e un numero di caratteri che vuole leggere da tastiera. A questo punto, il sistema lo metterà in attesa e si occuperà di altro, riempiendo sequenzialmente il buffer via via che i tasti vengono effettivamente premuti (e quindi le relative interruzioni sollevate). Una volta che il buffer sia riempito dal numero di caratteri richiesti dal processo, potrà quindi rimettere il processo in esecuzione, o nella lista pronti, e proseguire.

Per tenere conto di più processi che possono voler leggere contemporaneamente, poi, ci dotiamo di un semaforo che tenga conto di chi sta usando quella risorsa in un certo momento. Un problema qui sarà come usare i semafori da lato sistema, e verrà discusso nella prossima lezione.

19 Lezione del 11-04-25

Riprendiamo il discorso dell'I/O nel kernel.

19.1 Primitive di I/O

L'ipotesi generale è che una primitiva di I/O vuole trasferire dati da e alla memoria o da e allo spazio di I/O. La forma generica della primitive che vorremo fornire all'utente sarà quindi del tipo:

- **Lettura:**

```
read_n(id, char* buf, natq quanti);
```

  dove prendiamo la periferica `id`, e la usiamo per leggere `quanti` byte e inserirli nel buffer (in memoria condivisa) `buf`;

- **Scrittura:**

```
write_n(id, const char* buf, natq quanti);
```

  dove prendiamo la periferica `id`, e ci scriviamo `quanti` byte dal buffer (in memoria condivisa) `buf`.

Avremo quindi che un certo processo P_1, a qualche punto della sua esecuzione, chiama una primitiva di I/O. A questo punto il controllo passa al kernel, che si occupa quindi di gestire l'operazione, sfruttando prima di tutto le istruzioni privilegiate a cui ha accesso `IN`, `OUT`, ecc... e le possibilità di scheduling di cui dispone per eseguire, mentre attende per la sincronia col dispositivo, altri processi (mettendo quindi quello che aveva chiamato la primitiva in attesa). Quando P_1 viene rimesso in esecuzione, non vede niente di diverso nel suo contesto privato, se non il fatto che il buffer contiene adesso l'input desiderato.

Le primitive di sistema che gestiscono un certo dispositivo prendono il nome, abbiamo detto, di **driver**.

1. Per realizzare il meccanismo di **sincronizzazione** sfruttiamo *semaforo*: vogliamo che il la primitiva di I/O chiami la `sem_wait()` al momento dell'inizio dell'operazione di I/O, e che la `sem_signal()` venga chiamata sullo stesso semaforo alla fine dell'operazione per segnalare che l'operazione è finita (dal driver stesso).

2. Per realizzare invece la **mutua esclusione** si sfrutta il procedimento inverso: si parte con un semaforo inizializzato a `sem_wait()`, e successivamente ogni processo che inizia un'operazione di I/O ne prende un *"gettone"*, restituendolo a termine operazione, così che nessun processo possa iniziare una operazione di I/O contemporaneamente a un altro.

19.1.1 Primitive nel kernel

Avevamo visto che il problema era usare le `sem_wait()` e le `sem_signal()` lato kernel, in quanto queste effettivamente interrompono routine sistema e ne violano l'atomicità.

Vediamo però che, se ci limitiamo a non manipolare le strutture dati sistema nelle primitive `read_n()` e `write_n()`, non incappiamo nei rischi per cui avevamo introdotto l'atomicità delle routine sistema in primo luogo (le uniche a manipolare la lista processi saranno le `read_n()` e `write_n()`).

L'unico problema resterà il discorso del contesto, che verrebbe salvato così 2 volte. Rimuoviamo allora le `salva_stato` e `carica_stato`.

Trasformiamo quindi le primitive di I/O in primitive effettivamente *non atomiche*, che eseguono nello stesso contesto (a livello di registri) del processo chiamante.

19.1.2 Driver

Vediamo come si svolge la situazione lato driver. Potremmo pensare iniziatutto di sfruttare un certo descrittore di *dispositivo*, `des_io`, realizzato ad esempio come:

```
des_io {
  char* buf;
  natq quanti;

  // semafori
  natl mutex;
  natl sync;
};
```

cioè contenente tutte le informazioni necessarie al driver per soddisfare la richiesta del processo che ha richiesto l'I/O, scrivendo i dati ottenuti nel buffer `buf` giusto all'indice `quanti` (che incrementerà o decrementerà da solo) giusto.

- `buf` e `quanti` vengono forniti alla primitiva di I/O e cambiano per ogni richiesta da parte dei processi.
- `mutex` e `sync` sono gli indici dei semafori e vengono inizializzati una sola volta per ogni dispositivo. In particolare, stando a quanto abbiamo detto nello scorso paragrafo, vorremo:

```
// sync si inizializza a 0
if ( (ce->sync = sem_ini(0)) == 0xFFFFFFFF) {
  flog(LOG_WARN, "ce%d: impossibile allocare sync", next_ce);
  break;
}
// mutex si inizializza a 1
if ( (ce->mutex = sem_ini(1)) == 0xFFFFFFFF) {
  flog(LOG_WARN, "ce%d: impossibile allocare mutex", next_ce);
  break;
}
```

Notiamo che in ogni caso il buffer rappresenta i dati lato utente, cioè come vedremo contenuti nella memoria condivisa dei processi, e nulla riguardo al dispositivo vero e proprio: i suoi dati verranno ottenuti, a controllo d'interruzione, dalla primitiva stessa attraverso i suoi procedimenti specifici. In particolare, il buffer della `read_n` sarà quello dove il driver dovrà scrivere cosa legge così che il processo lo veda, mentre il buffer (costante) della `write_n` sarà quello dove il driver dovrà leggere per restituire poi al dispositivo.

Chiamando `proc` la primitiva che si occupa di realizzare la richiesta I/O (nell'esempio in 19.1, potrebbe essre sia `read_n` che `write_n`), e `driver` la primitiva del driver, si ha che i semafori di un certo descrittore `ce` si evolvono quindi come segue (in relazione alla lista di 19.1):

1. `proc` aspetta che arrivi `ce->mutex` (2);
2. `proc` prende `ce->sync` e viene messo in attesa (1);
3. `driver` restituisce `ce->sync` (1);
4. `proc` restituisce `ce->mutex` (2)

e che quindi `proc` e `driver` sono strutturati pressappoco come segue, prima `proc`:

```
extern "C" void proc(natl id, ...) {
  // ottieni ce
  sem_wait(ce->mutex); // (2)
  sem_wait(ce->sync); // (1)

  // qui saremo messi in attesa, e' compito di driver restituire ce->sync

  sem_signal(ce->mutex); // (2)
}
```

e poi `driver`:

```
extern "C" void driver() {
  // ottieni ce

  // intanto restituirai un byte
  ce->quanti--;

  if(ce-quanti == 0) {
    // appunto, restituiamo
    c_sem_signal(ce->sync); (1)
  }

  // qui gestiamo l'interruzione esterna del dispositivo
  // e restituiamo effettivamente il byte
}
```

Nei prossimi paragrafi andremo via via a definire i dettagli tecnici e raffinare l'implementazione.

19.1.3 Driver e primitive

La domanda è se il driver può chiamare le primitive semaforo che gestiscono i semafori di indice `mutex` e `sync`. In particolare, avevamo detto che il driver avrà il compito di chiamare la `sem_signal(sync)` per segnalare al processo che l'operazione di I/O è finita. Abbiamo però il problema della `salva_stato`, che andrà a sovrascrivere, quando chiamata da una routine sistema (e quindi anche da un driver) il contesto del processo in `esecuzione` (che potrebbe essere arbitrario) con i valori correnti della routine sistema, facendo evidentemente danni.

Potremmo allora pensare di usare direttamente la sua implementazione, cioè chiamare la `c_sem_signal()`. Il problema sarà che la `c_sem_signal()` usa la `sem_valido()` per controllare la validità del semaforo cercato, usando la `liv_chiamante()`, che non sarebbe significativa se chiamata senza passare da un interruzione (sfrutta il CS salvato presumibilmente in pila dalla `INT`). In particolare, vediamo che si ha:

```
extern "C" void c_sem_signal(natl sem)
{
  // una primitiva non deve mai fidarsi dei parametri
  if (!sem_valido(sem)) {
    flog(LOG_WARN, "semaforo errato: %u", sem);
    c_abort_p();
    return;
  }

  // corpo di esecuzione effettiva, alla 11.2.1
}
```

percorrendo il backtrace al contrario:

```
bool sem_valido(natl sem)
{
  // dal momento che i semafori non vengono mai deallocati,
  // un semaforo e' valido se e solo se il suo indice e' inferiore
  // al numero dei semafori allocati

  int liv = liv_chiamante(); // qui vogliamo distinguere il contesto
  return sem < sem_allocati_utente ||
    (liv == LIV_SISTEMA && sem - MAX_SEM < sem_allocati_sistema);
}
```

e infine:

```
int liv_chiamante()
{
  // (ci aspettiamo che) salva_stato ha salvato il puntatore
  // alla pila sistema subito dopo l'invocazione della INT
  natq* pila = ptr_cast<natq>(esecuzione->contesto[I_RSP]);

  // -> peccato che nessuno ha chiamato la salva_stato!
  //    siamo passati da c_sem_signal() e non a_sem_signal()

  // la seconda parola dalla cima della pila contiene il livello
  // di privilegio che aveva il processore prima della INT
  return pila[1] == SEL_CODICE_SISTEMA ? LIV_SISTEMA : LIV_UTENTE;
}
```

Saremo quindi costretti a replicare in qualche modo la `c_sem_signal()` nel codice del driver, cioè dire:

```
extern "C" void c_driver() {
  // ottieni ce

  // intanto restituirai un byte
  ce->quanti--;

  if(ce-quanti == 0) {
    // qui (!) facciamo la sem_signal()
    des_sem *s = &array_dess[ce->sync];

    s->counter = 0;
    des_proc* lavoro = rimozione_lista(s->pointer);
    inspronti();  // preemption
    inserimento_lista(pronti, lavoro);
    schedulatore(); // preemption
  }

  // qui gestiamo l'interruzione esterna del dispositivo
  // e restituiamo effettivamente il byte
```

```
}
```

In questo modo, il codice del driver dovrà essere effettivamente atomico (`c_sem_signal()` manipola le liste di processi, che sono strutture dati sensibili). Questo potrebbe essere problematico in quanto ci impedisce di gestire interruzioni innestate a priorità più alta. Decidiamo di continuare con questa limitazione.

19.1.4 Cavalli di Troia

Altre questioni di sicurezza potrebbero riguardare i cosiddetti **cavalli di Troia**: un utente potrebbe sfruttare la `write_n()` per scrivere in locazioni di memoria arbitrarie, dove lui da solo non avrebbe potuto scrivere. Si rende quindi necessario controllare gli indirizzi passati alle primitive di I/O lato software.

Vediamo i problemi che vogliamo controllare:

- Gli indirizzi potrebbero fare wraparound, costringendo il driver a saltare da M_2 a M_1, e quindi sovrascrivendo più memoria di quanta probabilmente si voleva impattare;
- Regioni di indirizzi virtuali (di per sé contigue) che potrebbero avere flag di scrivibilità non contigui (cioè si potrebbero incontrare flag di traduzioni modalità sistema o sola lettura);
- Indirizzi virtuali non tradotti, che quindi causerebbero page fault (e visto che abbiamo detto il driver deve essere atomico, non possiamo permetterci nessuna eccezione);
- Non vogliamo accedere alla memoria privata del processo richiedente l'I/O, perché l'albero di traduzione caricato al momento dell'esecuzione del driver non sarà sicuramente il suo (è stato messo in attesa dalla primitiva), ma quello di un altro processo (quello in `esecuzione`) con la sua memoria privata (che sicuramente non vogliamo toccare).

L'unica regione valida resta quindi quella che avevamo già detto, cioè la **memoria condivisa**. Notiamo che questa era comunque una necessità, in quanto vogliamo che il processo passi per forza un buffer corrispondente ad un indirizzo in memoria condivisa, appunto perchè chiedere alla primitiva driver di scrivere nel suo spazio privato sarebbe complesso (a meno di non passare un indirizzo fisico, e quindi passare dalla finestra FM, cosà però abbastanza complicata dal punto di vista dei controlli).

Per tutti questi controlli, quindi, siamo quindi costretti a controllare tutte le traduzioni.

Scriveremo quindi un'apposita funzione di controllo del buffer del tipo:

```
// restituisce true (nell'I_RAX del contesto) se la regione va bene,
// false altrimenti
extern "C" bool c_access(vaddr begin, natq dim, bool writeable, bool
    shared = true)
{
  esecuzione->contesto[I_RAX] = false;

  // l'intervallo e' valido?
  if (!tab_iter::valid_interval(begin, dim))
    return false;

  // siamo nella regione di memoria utente condivisa?
```

```
12   if (shared && (!in_utn_c(begin) || (dim > 0 && !in_utn_c(begin + dim -
       1))))
13     return false;
14
15   // usiamo un tab_iter per percorrere tutto il sottoalbero relativo
16   // alla traduzione degli indirizzi nell'intervallo [begin, begin+dim).
17   for (tab_iter it(esecuzione->cr3, begin, dim); it; it.next()) {
18     tab_entry e = it.get_e();
19
20     // interrompiamo il ciclo non appena troviamo qualcosa che non va
21     if (!(e & BIT_P) || !(e & BIT_US) || (writeable && !(e & BIT_RW)))
22       return false;
23   }
24   esecuzione->contesto[I_RAX] = true;
25   return true;
26 }
```

che fa prima i controlli detti su intervallo e regione processi, e poi percorre l'intero albero di traduzione, controllando che ogni traduzione sia scrivibile.

19.1.5 Implementazione di un driver

Vediamo quindi l'implementazione di un semplice driver per un dispositivo fasullo cioè il dispositivo *CE*, dotato di soli 3 registri (e relative porte):

`0x?? + 0`	**CTL**, *Control Register*
`0x?? + 4`	**STR**, *Status Register*
`0x?? + 8`	**RBR**, *Receive Buffer Register*

Il `0x??` è dato dal fatto che il dispositivo è montato sul bus PCI, ergo per disporre i suoi registri nello spazio di I/O bisogna regolarne il **BAR**.

In ogni caso, il funzionamento del dispositivo è banale: si limita a controllare periodicamente CTL, e quando vi trova 1, stampare un carattere alfanumerico.

Disponiamo per tale dispositivo il descrittore:

```
1  struct des_ce {
2    // i registri, variabili perche in PCI
3    ioaddr iCTL, iSTS, iRBR;
4
5    // gia' visto
6    char *buf;
7    natl quanti;
8    natl sync;
9    natl mutex;
10 };
```

Avremo quindi una funzione `ce_init()`, che viene lanciata all'avvio del kernel, e che si occupa di impostare il BAR e i descrittori dispositivo.

A questo punto, forniremo all'utente la primitiva `c_ceread_n()`, per la lettura:

```
1  // questa non puo' toccare i descrittori di processo, praticamente e' una
       chiamata di funzione,
2  // quindi e' interrompibile (non atomica)
3  extern "C" void c_ceread_n(natl id, char *buf, natl quanti)
4  {
5    if (id == id_ce) {
6      flog(LOG_WARN, "ce non riconosciuto: %d", id);
7      abort_p(); // e' quella normale, nessuno ha ancora chiamato
       salva_stato
```

```
  }

  if (!quanti)
    return;

  if (!c_access(reinterpret_cast<vaddr>(buf), quanti, true, false)) {
    flog(LOG_WARN, "buf non valido\n");
    abort_p();
  }

  des_ce *ce = ce;
  sem_wait(ce->mutex);
  ce->buf = buf;
  ce->quanti = quanti;
  outputb(1, ce->iCTL); // qui attivi
  sem_wait(ce->sync);
  sem_signal(ce->mutex);
}
```

mentre imposteremo nella IDT una funzione assembly che chiama il seguente gestore per l'interruzione esterna associata a *CE*:

```
// questa e' atomica, e reimplementa in qualche modo sem_signal()
extern "C" void c_driver_ce(int id)
{
  // qui otteniamo il descrittore di dispositivo
  des_ce *ce = ce;
  ce->quanti--;

  // qui si termina se si e' finito
  if (ce->quanti == 0) {
    // qui disattiviamo le interruzioni
    outputb(0, ce->iCTL);

    // qui (!) facciamo la sem_signal()
    des_sem *s = &array_dess[ce->sync];

    s->counter = 0;
    des_proc* lavoro = rimozione_lista(s->pointer);
    inspronti();  // preemption
    inserimento_lista(pronti, lavoro);
    schedulatore(); // preemption
  }

  // qui gestiamo effettivamente il dispositivo

  // questa va fatta dopo che disattivi le interruzioni,
  // altrimenti potresti avere un'altra interruzione sotto (nell'IRR)
  char b = inputb(ce->iRBR);
  *ce->buf = b;
  ce->buf++;
}
```

dove notiamo il dettaglio che la gestione del dispositivo si fa dopo aver disattivato, eventualmente, il dispositivo impostando CTL a 0. In caso contrario, si potrebbe ottenere una nuova interruzione, messa in IRR dall'APIC, che andrebbe erroneamente a richiamare il driver una volta di troppo rispetto a quelle previste.

Lato assembler, il driver verrà impostato come segue:

```
% setup IDT
carica_gate INTR_TIPO_CE  a_driver_ce LIV_SISTEMA
```

dove la `a_driver_ce` è:

```
a_driver_ce:
  call salva_stato
  movq $0, %rdi
  call c_driver_ce
  call apic_send_EOI
  call carica_stato
  iretq
```

La `c_ceread_n()` avrà invece la controparte assembler:

```
  .extern c_ceread_n
a_ceread_n:
  % qui nessuno chiama salva_stato
  call c_ceread_n
  % qui nessuno chiama carica_stato
  iretq
```

Dove notiamo ancora meglio che praticamente si ha una semplice chiamata di funzione che ritorna con la `IRETQ` invece che con la `RET`.

20 Lezione del 15-04-25

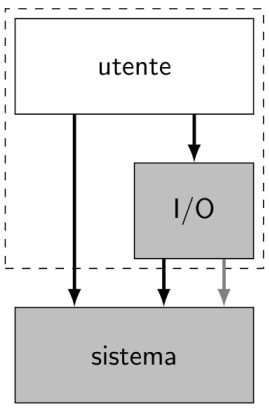
20.1 Gestione di primitive di I/O quasi atomiche nel kernel reale

Abbiamo che quanto abbiamo visto finora sulle primitive di I/O non corrisponde propriamente al kernel reale: avevamo infatti il problema di dover realizzare primitive *"quasi"* atomiche, nel senso che potevano essere interrotte dalle primitive semaforiche (che usavano per sincronizzazione e mutua esclusione), ma avevano anche accesso alle strutture dati sistema (in particolare le code processi), per cui bisogna fare attenzione nella loro implementazione di non toccare parti sensibili, in modo da evitare errori di difficile rilevazione.

Possiamo migliorare leggermente questa architettura separando il **modulo** sistema (inteso come un insieme di file compilati e collegati insieme) dal modulo di I/O, cioè stabilendo la seguente distinzione:

- **Modulo sistema**, implementato in `sistema.cpp` e `sistema.s` (parti C++ e assembler);
- **Modulo di I/O**, implementato in `io.cpp` e `io.s` (come sopra).

Avremo quindi che la parte I/O dipenderà dalla parte sistema, ma non avrà accesso a tutte le sue struttura dati: ad esempio non avra accesso alle code processi. Si andrà quindi a creare la seguente struttura:



dove le frecce nere indicano primitive accessibili a livello utente e le frecce grigie indicano primitive accessibili a livello sistema.

Prima di capire il perché di questa distinzione vediamo di capire a quale livello lavora il modulo di I/O.

Abbiamo visto in 8.1.1 come il registro CS ci permette di definire 4 livelli, o *ring*, e di come storicamente i ring intermedi (1 e 2) venivano usati per i driver (quindi, per i nostri scopi, i moduli di I/O):

CS	Ring	Tipo
00	Ring 0	Kernel (sistema)
01	Ring 1	Driver *(non più supportato)*
10	Ring 2	/ /
11	Ring 3	Utente

Oggi, però, questa distinzione non esiste più (assieme alla memoria segmentata, si perde anche questa funzionalità del registro CS), e gli unici livelli permessi sono 0 (che abbiamo assunto come livello sistema) e 3 (che abbiamo assunto come livello utente).

Dovremo quindi porre il modulo di I/O in livello sistema, in modo che questo possa avere accesso alle funzioni privilegiate di cui vogliamo privare l'utente. L'unico vantaggio dell'architettura adottata sarà quindi quello di impedire al modulo di I/O di intaccare le strutture dati sensbili lato sistema. Ad esempio, non si potranno nemmeno usare erroneamente le `salva_stato()` o `carica_stato()` nelle primtive di I/O, in quanto queste non saranno nemmeno definite nel modulo di I/O.

20.1.1 Processi esterni

Avremo quindi che i driver saranno rappresentati da processi, detti **processi esterni**: le primitivi accessibili solo da livello sistema dal modulo di I/O saranno appunto le primitive per la creazione di processi esterni (la `activate_pe()`), e altre primitive utili al modulo di I/O (a cui l'utente non dovrà avere accesso). Queste saranno Il sistema avrà quindi il compito di definire, su richiesta del modulo di I/O **handler** caricati nella IDT, che si occuperanno di gestire le interruzioni esterne semplicemente indirizzandole ai processi di I/O relativi, creati anch'essi dal modulo di I/O.

21 Lezione del 15-04-25

Avevamo quindi visto come si era introdotto un nuovo modulo, il modulo di I/O, che aveva il compito di gestire da gestire l'I/O da livello sistema, fornendo primitive di I/O e processi (driver) al modulo utente e sfruttando primitive, sia accessibili a livello utente che accessibili a livello sistema (quindi solo a lui) del sistema.

21.0.1 Cambi di contesto a processi esterni

Avremo quindi che in un dato momento sulla macchina sono in esecuzione i processi utente (P_1, P_2, ecc...) e i processi *esterni* di I/O. Assumiamo per adesso di avere un solo processo di I/O, chiamato appunto IO.

Nel momento in cui uno dei processi utente (diciamo P_1) chiama una primitiva di I/O, come ad esempio la `read_n()` già nominata, viene messo in attesa e si mettono in esecuzione altri processi. Di qui in poi, all'arrivo di un'interruzione esterna relativa all'operazione di I/O corrente, viene eseguito l'handler (definito in sistema) e quindi da questo

messo in esecuzione il processo IO, che riempie ad ogni chiamata il buffer, e all'ultima chiamata rimette in lista pronti (attraverso i meccanismi implementati coi semafori visti in 19.1.5) il processo P_1.

21.0.2 Implementazione degli handler

Un'idea di base potrebbe essere quella di dedicare un *processo esterno* ad ogni tipo di interruzione esterna nell'IDT. Abbiamo visto come l'APIC supporta 24 piedini di interruzione esterna (`IREQ`), il cui tipo di interruzione nell'IDT può essere scelto via software modificando appositi registri. Vorremo quindi creare l'associazione:

$$\text{piedino di interruzione} \rightarrow \text{processo esterno}$$

cioè all'arrivo dell'interruzione esterna, il processore deve eseguire l'handler corretto come definito nella IDT, e questa deve mettere in esecuzione il processo corrente.

Servira quindi anche un *handler* per ogni tipo di interruzione esterna, corrispondente al suo processo esterno (definiremo quindi `handler_0`, `handler_1`, ..., `handler_23`).

La struttura di un handler generico sarà la seguente:

```
handler_i:
  # gestiamo l'interruzione
  call salva_stato

  # sospensdiamo il processo in esecuzione
  call inspronti

  # trova il processo esterno corrispondente
  mov a_p+i*8, %rax
  # e mettilo in esecuzione
  movq %rax, esecuzione

  call carica_stato
  iretq
```

`a_p` conterrà l'indirizzo della prima entrata di una lista di puntatori ai descrittori dei processi esterni:

	a_p
0	Processo esterno 0
⋮	//
23	Processo esterno 23

Questa si definisce nel sistema come:

```
// MAX_IRQ e' il numero di linee di interruzione (24)
des_proc* a_p[apic::MAX_IRQ];
```

21.0.3 Implementazione dei processi esterni

I processi esterni avranno quindi struttura simile. Di base, avremo bisogno (come abbiamo visto) di descrittori associati a periferiche che tengono conto dei tipi specifici alle periferiche, e delle informazioni riguardanti l'ultima richiesta di I/O:

```
des_io array_des_io[MAX_DES_IO];
```

Avremo quindi che un processo esterno è implementato come:

```
void estern_i(natq id) {
  des_io *d = &array_des_io[id];
  // non servono controlli, id e' fidato

  // i processi esterni non terminano mai, quindi facciamo un ciclo
    infinito
  for(;;) {
    // qui si gestisce l'interruzione
    // ...

    // aspetta prossima interruzione
    wfi();
  }
}
```

Notiamo che chiaramente non possiamo chiamare, finita la gestione, `schedulatore()`, in quanto questo processo (non atomico) non deve avere accesso a strutture dati sensibili, e proprio per questo è stato messo nel modulo di I/O.

Si rende quindi disponibile un'altra di quelle primitive non accessibili all'utente, cioè la `wfi()` (*Wait For Interrupt*), che appunto sospende il processo mettendolo in attesa fino alla prossima interruzione di sua competenza, e mettendo in esecuzione un altro processo (cioè facendo la chiamata a `schedulatore()` che volevamo fare in primo luogo).

Possiamo vedere l'implementazione, in assembler, della `wfi()`:

```
a_wfi:
  call salva_stato

  # intanto rispondiamo all'APIC
  call apic_send_EOI

  # non dobbiamo fare niente col processo esterno,
  # e' gia' nella tabella a_p

  call schedulatore

  call carica_stato
  iretq
```

Vediamo quindi il dettaglio di poter usare, questa volta, le primitive semaforiche all'interno del processo esterno (in quanto il processo deve essere interrompibile), cioè dire:

```
extern "C" void estern_i(natq id)
{
  des_io *d = &array_des_io[id];
  // non servono controlli, id e' fidato

  for (;;) {
    // gestisci l'interruzione

    if (d->quanti == 0) {
      sem_signal(ce->sync); // qui puoi usare la sem_signal()
    }

    wfi(); // ecco la wfi() per il ritorno
  }
}
```

Infine, vediamo a scopo di esempio il processo esterno per una periferica nota, la *CE* del 19.1.5:

```
extern "C" void estern_ce(natq id)
{
  // trova il descrittore
  des_ce *ce = &array_ce[id];

  for (;;) {
    // gestisci l'interruzione
    ce->quanti--;
    if (ce->quanti == 0) {
      outputb(0, ce->iCTL);
    }
    char b = inputb(ce->iRBR);
    *ce->buf = b;
    ce->buf++;

    if (ce->quanti == 0) {
      sem_signal(ce->sync); // qui puoi usare la sem_signal()
    }

    wfi(); // ecco la wfi() per il ritorno
  }
}
```

Tastiera e hard disk dispongono di processi esterni simili, definiti rispettivamente in `estern_kbd()` e `estern_hd()`.

Vediamo un dettaglio: durante l'esecuzione di un processo esterno, non potrà accadere che questo viene interrotto da un interruzione esterna relativa allo stesso processo: questo perchè le interruzioni esterne relative a quel dispositivo sono effettivamente disattivate fino alla chiamata di `wfi()`, cioè quando viene chiamata la `apic_send_EOI()` (e non si può interrompere la `wfi()` perchè è una primitiva di sistema, quindi atomica).

Esiste invece la possibilità che un interruzione di livello più alto nella IDT vada ad interrompere il processo esterno (quindi si metta in esecuzione il processo esterno relativo a *quella* interruzione), ma questo è desiderabile in quanto corrisponde in maniera naturale alla priorità di gestione delle interruzioni esterne secondo l'APIC.

21.0.4 Creazione di processi esterni

Vediamo quindi nel dettaglio la `activate_pe()`, di uso concesso solo al modulo di I/O, che viene usata per creare i processi esterni. Questa sarà diversa dalla comune `activate_p()`, in quanto dovrà anche impostare la tabella `a_p` per gli handler.

La funzione avrà quindi firma: `activate_pe(void (*f)(natq), natq id, natl prio, nat livello, natb irq)`, cioè si specifica:

- La **funzione** (`f`) che realizza il processo stesso;
- L'**indice** (`id`) di processo;
- La **priorità** (`prio`) del processo (che per i processi determinerà anche il tipo di interruzione nell'IDT);
- Il **livello** (`livello`) del processo, fin qui tutto normale;
- La **linea di interruzione** (`IRQ`) che tale processo esterno gestisce.

Come anticipato, un discorso importante va fatto sulla priorità di questi processi. Abbiamo infatti la priorità *da noi* definita, cioè `prio`, e la priorità nell'APIC (quindi nella

IDT), data dal codice di interruzione assegnato ad una linea `IRQ`. Questa seconda priorità, che chiamiamo **tipo**, viene ricavata direttamente dalla proprietà `prio` secondo la seguente formula:

$$\texttt{prio} = \texttt{MAX\_PRIO\_UTENTE} + \texttt{tipo}$$

Da qui in poi, le parti di creazione vera e propria del processo potranno essere messe in comune con la `activate_p()`, e la `activate_pe()` avrà il solo compito aggiuntivo di predisporre un handler, con relativo setup dell'APIC e caricamento dell'handler nell'IDT.

21.0.5 Implementazione del modulo I/O

Vediamo quindi la struttura generaled del modulo I/O stesso. Questo dispone delle sue primitive, che definisce per l'utente e carica nella IDT attraverso la primitiva sistema `fill_gate()` (potrebbe farlo da sé, in quanto gira a livello utente, ma non lo fa per *"educazione"* nei confronti del modulo sistema, cioè si centralizza la gestione dell'IDT per evitare errori).

Le primitive definite gestiscono quindi essenzialmente tastiera, video ed hard disk, cioè si dispone delle primitive:

```
# inserimento delle primitive di I/O nell'IDT
fill_io_gate  IO_TIPO_HDR     a_readhd_n
fill_io_gate  IO_TIPO_HDW     a_writehd_n
fill_io_gate  IO_TIPO_DMAHDR  a_dmareadhd_n
fill_io_gate  IO_TIPO_DMAHDW  a_dmawritehd_n
fill_io_gate  IO_TIPO_RCON    a_readconsole
fill_io_gate  IO_TIPO_WCON    a_writeconsole
fill_io_gate  IO_TIPO_INIC    a_iniconsole
fill_io_gate  IO_TIPO_GMI     a_getiomeminfo
```

I processi esterni per le periferiche (tastiera e hard disk) sono quindi definiti come nel paragrafo precedente, cioè si ha:

```
// processo esterno associato alla tastiera
void estern_kbd(natq) {
  // ...
}

// processo esterno per le richieste di interruzione dell'hard disk
void estern_hd(natq) {
  // ...
}
```

21.0.6 Inizializzazione del modulo I/O

Possiamo quindi vedere un ulteriore dettaglio riguardo alla sequenza di boot, cioè quella che riguarda l'inizializzazione dell'I/O.

Avremo infatti nel `main()` del modulo sistema la seguente chiamata all'entry point del modulo I/O:

```
extern "C" void main(natq) {
  flog(LOG_INFO, "Creo il processo main I/O");
  main_io = crea_processo(mio_entry, int_cast<natq>(&io_init_done),
    MAX_EXT_PRIO, LIV_SISTEMA);
  if (main_io == nullptr) {
    flog(LOG_ERR, "impossibile creare il processo main I/O");
    goto error;
```

```
  }
  processi++;
  flog(LOG_INFO, "Attendo inizializzazione modulo I/O...");

  // cediamo il controllo al modulo I/O e aspettiamo che setti
  // la variable io_init_done
  cedi_controllo(main_io);

  // sostanzialmente una sorta di sem_wait()
  while (!io_init_done)
    halt(); // abilita temporaneamente le interruzioni esterne
}
```

dove la funzione `cedi_controllo()` ha il compito di salvare lo stato corrente del kernel in pila in modo che vi si possa ritornare dopo l'esecuzione del *processo di inizializzazione* (è proprio un processo) di un altro modulo.

A questo punto il modulo di I/O prenderà il controllo dal suo entry point:

```
extern "C" void main_io(natq p)
{
  // questo e' il flag che sta guardando il main() di sistema
  int *p_io_init_done = ptr_cast<int>(p);

  // riempi la IDT delle primitive di I/O
  fill_io_gates();

  // inizializza l'heap
  ioheap_mutex = sem_ini(1);
  if (ioheap_mutex == 0xFFFFFFFF) {
    flog(LOG_ERR, "impossible creare semaforo ioheap_mutex");
    abort_p();
  }
  char* end_ = allinea_ptr(_end, DIM_PAGINA);
  heap_init(end_, DIM_IO_HEAP);
  flog(LOG_INFO, "Heap del modulo I/O: %llxB [%p, %p)", DIM_IO_HEAP,
      end_, end_ + DIM_IO_HEAP);

  // inizializza la console (tastiera + video)
  flog(LOG_INFO, "Inizializzo la console (kbd + vid)");
  if (!console_init()) {
    flog(LOG_ERR, "inizializzazione console fallita");
    abort_p();
  }

  // inizializza l'hard disk
  flog(LOG_INFO, "Inizializzo la gestione dell'hard disk");
  if (!hd_init()) {
    flog(LOG_ERR, "inizializzazione hard disk fallita");
    abort_p();
  }

  // avverti main che hai finito
  *p_io_init_done = 1;

  // termina
  terminate_p();
}
```

Abbiamo quindi una serie di funzioni (`heap_init()`, `console_init()`, `hd_init()`, ...) che hanno il compito di inizializzare ogni dispositivo. Queste, oltre ad inizializzare tutte le

informazioni relative al descrittore di dispositivo, hanno il compito di creare il processo esterno di gestione di tale dispositivo.

Ad esempio, riguardo alle periferiche di tipo *CE* descritte in 19.1.5 (e ammesso quindi di avere più periferiche dello stesso tipo nel bus), potremmo definire la funzione di inizializzazione:

```
// trova le periferiche CE installate e crea i rispettivi processi esterni
bool ce_init()
{
  // scansiona il bus PCI per le periferiche di tipo CE
  for (natb bus = 0, dev = 0, fun = 0;
       pci::find_dev(bus, dev, fun, 0xedce, 0x1234);
       pci::next(bus, dev, fun))
  {
    if (next_ce >= MAX_CE) {
      flog(LOG_WARN, "troppi dispositivi ce");
      break;
    }

    // crea il descrittore
    des_ce *ce = &array_ce[next_ce];

    // configuralo
    ioaddr base = pci::read_confl(bus, dev, fun, 0x10);
    base &= ~0x1;
    ce->iCTL = base;
    ce->iSTS = base + 4;
    ce->iRBR = base + 8;

    if ( (ce->sync = sem_ini(0)) == 0xFFFFFFFF ) {
      flog(LOG_WARN, "ce%d: impossibile allocare semaforo sync", next_ce);
      break;
    }
    if ( (ce->mutex = sem_ini(1)) == 0xFFFFFFFF ) {
      flog(LOG_WARN, "ce%d: impossibile allocare semaforo mutex", next_ce)
    ;
      break;
    }

    // trova la linea di interruzione del dispositivo
    natb irq = pci::read_confb(bus, dev, fun, 0x3c);

    // attiva il procsso esterno
    if (activate_pe(estern_ce, next_ce, MIN_EXT_PRIO + 0x80, LIV_SISTEMA,
    irq) == 0xFFFFFFFF) {
      flog(LOG_WARN, "ce%d: impossibile attivare processo esterno",
    next_ce);
      break;
    }
    // log...

    next_ce++;
  }

  // restituisci true se non sei riuscito ad inizializzare
  return next_ce != 0;
}
```

Implementazioni simili si trovano per la `console_init()`, che chiama a sua volta `vid_init()` e `kbd_init()`, e la `hd_init()`.

Come ultimo dettaglio, ricordiamo che il timer è comunque gestito da sistema, utilizzando i driver come definiti in `19.1.5` (in particolare, `driver_td()`).

22 Lezione del 28-04-25

22.1 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...

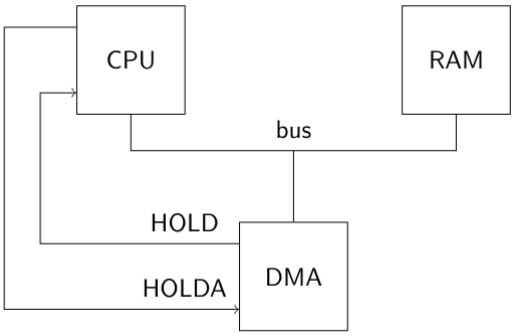
Introduciamo quindi il DMA per delegare tutta l'operazione di trasferimento dati dalle interfacce alle interfacce stesse. Vorremo quindi avere delle primitive:

- `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 effettuate da CPU o DMA, in quanto per questa sono equivalenti.

Dotiamo quindi il controllore DMA di due linee di handshake `HOLD` e `HOLDA` che lo connettono alla CPU:

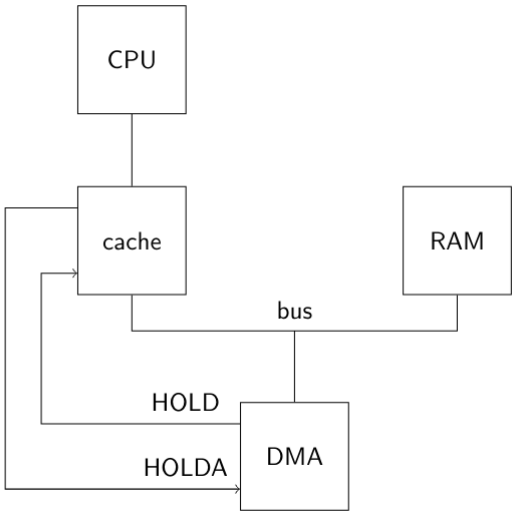


Al momento dell'inizio di un'operazione DMA, il controllore alza `HOLD`, e la CPU risponde alzando `HOLDA` e portando i suoi pin di uscita in alta impedenza. Da qui in poi il controllore lavora sul bus, in regime di `cycle-stealing`, cioè "rubando" cicli di accesso alla CPU. Finita l'operazione, il controllore 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 con la CPU). Oggi, lo stesso discorso vale nel senso opposto, ad esempio per le interfacce di rete, che raggiungono velocità del Gigabit al secondo (quindi molto più veloci della CPU).

22.1.1 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 DMA, 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. Vediamo nel dettaglio quali problemi possono apparire, e quali soluzioni si possono adottare. in regime di *write-through* e *write-back*.

Write-through

Supponiamo quindi come primo 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 nell'introdurre un'apposita istruzione per forzare il controllore di cache ad effettuare 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 invece l'ipotesi di *write-back*, cioè di scritture effettuate in differita dalla cache alla RAM da parte del controllore di 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 cache 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, che viene invalidata (se questa esiste in cache), effettua il *merge* fra questa e quanto ottenuto dal dispositivo 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 e limitarsi ad invalidarla: ci si aspetterà che il controllore DMA la modificherà integralmente e non ci sarà bisogno di merge;
  - 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 RAM sia la più recente, cioè quella ottenuta dal controllore DMA).

22.1.2 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 delle regioni di memoria dedicate ai 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 e potrebbe continuare operazioni di scrittura o lettura con effetti disastrosi.

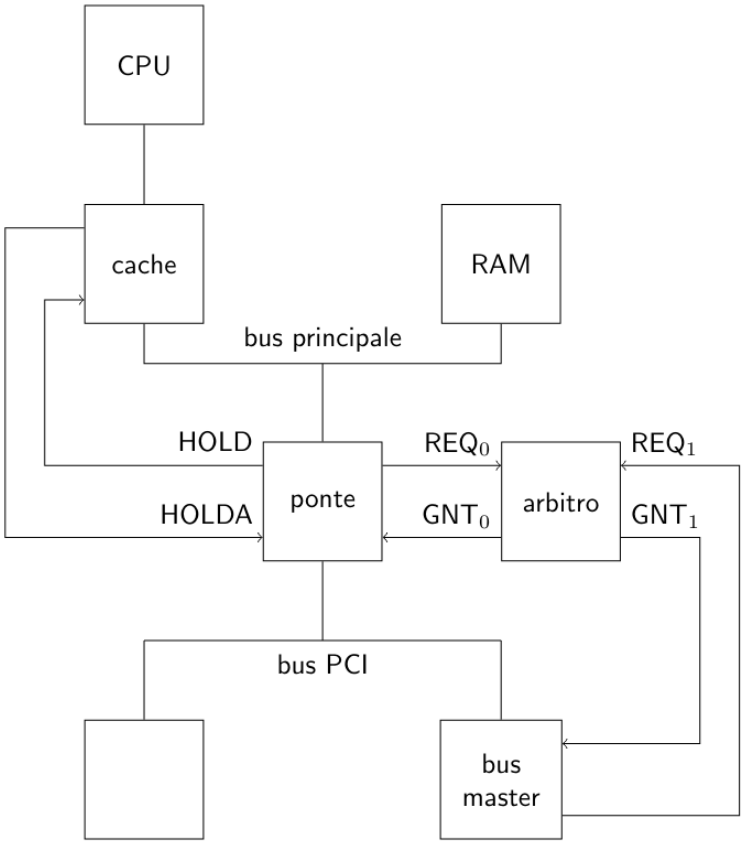
22.1.3 DMA nel bus PCI

Inseriamo quindi il controllore DMA nel bus PCI.

In questo caso sarà il ponte ospite-PCI ad occuparsi del DMA sul bus principale, secondo le regole e le linee di connessione col controllore di cache appena viste.

Avremo quindi bisogno che nei singoli bus PCI ogni interfaccia abbia la possibilità di prendere il controllo del bus, secondo il cosiddetto meccanismo di **bus mastering** (avevamo già visto come ogni interfaccia, e anzi più propriamente ogni *funzione* PCI poteva iniziare transazioni sul bus PCI nella sezione 18.1).

Il bus mastering nei bus PCI viene gestito da un **arbitro**, che è collegato con linee REQ e GNT ad ogni interfaccia:



Le interfacce richiedono quindi accesso al bus attraverso la REQ, mentre questo gli risponde attraverso la GNT. Una volta ottenuta la conferma su GNT, il dispositivo in bus mastering può comportarsi come se si trovasse sul bus principale (passando attraverso il ponte ospite-PCI). Le operazioni che questo svolge 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 ultimo dettaglio 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 sarà quella di collegare l'APIC al ponte, in modo da poter ritardare le interruzioni esterne alla fine delle operazioni di trasferimento.

Infine, 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.

Notiamo che entrambe le soluzioni richiedono che il ponte adotti una politica strettamente FIFO alla gestione dei trasferimenti: il primo trasferimento iniziato e completato lato bus PCI è il primo trasferimento a essere riportato dal buffer interno al ponte in RAM.

23 Lezione del 29-04-25

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

23.1 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:

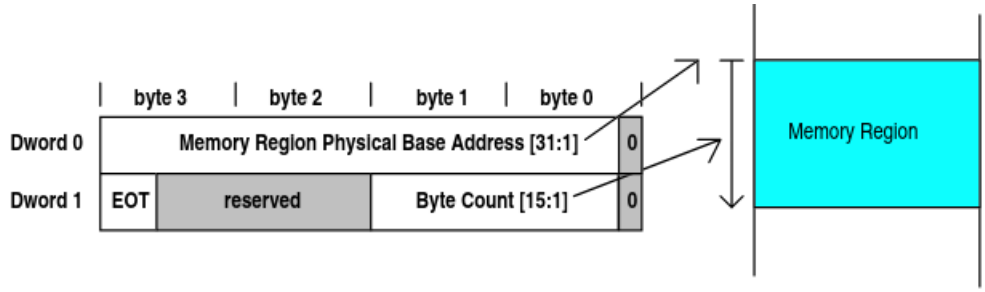
	Primario
`0x?? + 0`	**BMCMD**, *Bus Master Command*
`0x?? + 1`	Specifico al dispositivo
`0x?? + 2`	**BMSTR**, *Bus Master Status Register*
`0x?? + 3`	Specifico al dispositivo
`0x?? + 4-7`	**BMDTPR**, *Bus Master Descriptor Table Pointer*
	Secondario
`0x?? + 8`	**BMCMD**, *Bus Master Command*
`0x?? +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

23.1.1 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.

23.1.2 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.

```
// fondamentalmente un wrapper per hd::start_cmd di libce, che
    aggiorna il descrittore
void starthd_in(des_ata* d, natb vetti[], natl primo, natb quanti)
{
  d->cont = quanti;
  d->punt = vetti;
  d->comando = hd::READ_SECT;
  hd::start_cmd(primo, quanti, hd::READ_SECT);
}

// la primitiva vera e propria
extern "C" void c_readhd_n(natb vetti[], natl primo, natb quanti)
{
  des_ata* d = &hard_disk;

  // controlli (c_access)

  sem_wait(d->mutex);
  starthd_in(d, vetti, primo, quanti);
  sem_wait(d->sincr);
  sem_signal(d->mutex);
}
```

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

```
void dmastarthd_in(des_ata* d, natb vetti[], natl primo, natb quanti)
{
  // passo 1 della scaletta
  if (!prepare_prd(d, vetti, quanti)) {
    flog(LOG_ERR, "dmastarthd_in: numero di PRD insufficiente");
    sem_signal(d->sincr);
    return;
  }

  d->comando = hd::READ_DMA;
  d->cont = 1;

  // passo 2
  paddr prd = trasforma(d->prd);
  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 }
```

24 Lezione del 05-05-25

24.1 Architettura interna del processore

Vediamo più nel dettaglio dell'architettura interna dei processori Intel x86.

Per velocizzare l'operazione del processo la via principale potrebbe essere quello di aumentare le prestazioni dei componenti, cioè dei transistor, che lo compongono. Si ha che questo approccio però non è scalabile all'infinito, in quanto negli ultimi anni si è raggiunto un *plateu* delle prestazioni.

La soluzione che vediamo è quindi **architetturale**, e consiste nell'uso di una **pipeline** particolare per l'esecuzione delle istruzioni. Un normale ciclo di esecuzione di un'istruzione si svolge come:

Prelievo istruzione	Decodifica	Prelievo operandi	Esecuzione	Scrittura

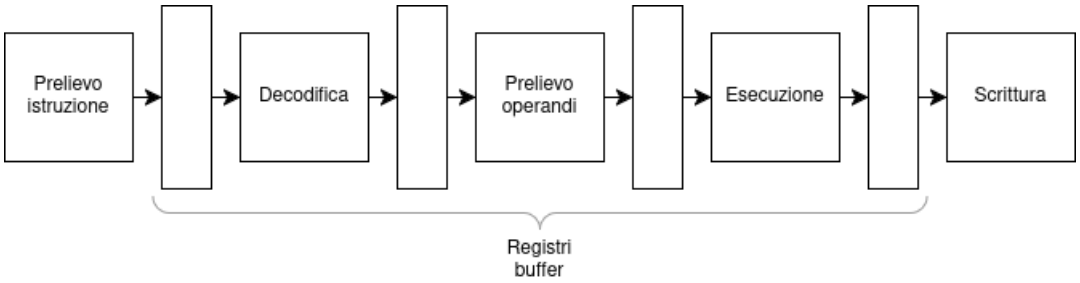
24.1.1 Pipeline

Se ognuna di queste fasi è svolta da una certa circuiteria, possiamo far passare in parallelo ogni istruzione da ogni circuiteria, cioè avere che l'istruzione gestita ad ogni istante temporale sucessivo t_0, t_1, ... è:

	Prelievo istruzione	Decodifica	Prelievo operandi	Esecuzione	Scrittura
t_0	i				
t_1	$i+1$	i			
t_2	$i+2$	$i+1$	i		
t_3	$i+3$	$i+2$	$i+1$	i	
t_4	$i+4$	$i+3$	$i+2$	$i+1$	i

Questo approccio chiaramente non modifica il tempo necessario ad eseguire una istruzione (e anzi vedremo lo aumenta un po'), ma di contro permette di aumentare la frequenza delle istruzioni eseguite, per l'esattezza di un fattore pari al numero di fasi in cui si divide l'esecuzione (qui 5).

Realizzaremo infatti questo tipo di struttura frapponendo fra ogni blocco funzionale un registro, che campiona sul rising edge del clock, rallentando leggermente la velocità della pipeline per accomodare il tempo di setup dei registri. Inoltre, il periodo del clock dovrà essere determinato dal più lungo dei percorsi (in termini temporali) Δ_1, Δ_2, ..., Δ_5 fra un registro e un altro, cioè dall'elemento più lento della pipeline. Si avrà quindi una configurazione del tipo:



Di base, questa configurazione risulterà comunque un'accelerazione del clock. Infatti, se l'intera pipeline richiedeva prima un tempo Δ, ci aspettiamo che ogni componente in cui la dividiamo richieda un tempo nell'ordine di $\sim \frac{\Delta}{5}$, ed esattamente $\frac{\Delta}{5}$ se ogni circuiteria ha lo stesso tempo di attraversamento, per cui il clock può essere accelerato di un fattore di 5.

Il bottleneck è però chiaro per la fase di esecuzione, che potrebbe andare dalla somma naturale alla divisione in virgola mobile, con evidenti differenze in tempo di esecuzione.

Inoltre, la stessa fase di prelievo potrebbe variare in requisiti temporali per via del tipo di codifica delle istruzioni, a lunghezza variabile, adottata dai processori Intel x86 (instruction set **CISC**, *Complex Instruction Set Computer*, contro gli instruction set **RISC**, *Reduced Instruction Set Computer*, adottati da ARM).

Ci troveremo quindi di fronte a situazioni dove, con una sola circuiteria di prelievo, non si può sapere quando un'istruzione è veramente finita prima di decodificarla, e quindi non si può procedere con una nuova fase di prelievo.

Chiaramente, tutto questo procedimento è semplificato per i processori ad architettura RISC, in quanto la dimensione delle istruzioni è standardizzata. Inoltre, si ottengono vantaggi nella gestione della pipeline eliminando la possibilità di avere operandi in me-

moria: si dedicano istruzioni dedicate alla lettura/scrittura in memoria da registri, cioe le LOAD e STORE, più semplici da gestire.

24.1.2 Alee

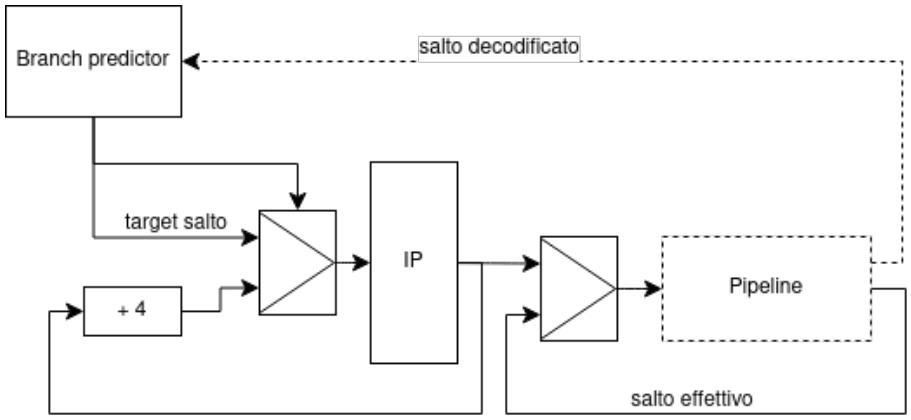
Altri problemi sono dati dalle **alee**, cioè legati ai salti condizionali (*alee di controllo*), o a casi dove istruzioni hanno bisogno di risultati di istruzioni ancora in pipeline (*alee di dati*), se non a casi dove la pipeline non permette in primo luogo l'esecuzione successiva di due istruzioni (*alee strutturali*).

- Questi problemi possono sempre essere risolti dall'introduzione di **bolle**: si modificano i registri intermedi perché possano conservare il loro stato, e in caso una certa rete di controllo rilevi situazioni a rischio di alee, si introducono nella pipeline *bolle*, cioè si lasciano stadi di elaborazione vuoti, o se vogliamo si introducono istruzioni `nop`, a effetto nullo (che chiaramente rappresentano throughput spreacto). I registri che alimentano gli stadi rimasti in attesa manterranno quindi una copia dell'istruzione allo scorso ciclo di clock, e invieranno invece avanti istruzioni nulle. In questo modo si torna effettivamente al processore prima della pipeline.
- Un'altra soluzione per alee dati e alee strutturali può essere quella di dotare l'ultima fase della pipeline di una linea di **bypass**, che porti il risultato a termine esecuzione a fronte della fase di esecuzione: un istruzione che richiede un operando non ancora scritto dall'istruzione precedente può ottenerlo direttamente da questa, attraverso il bypass.
- Per la gestione delle alee di controllo possiamo sfruttare la cosiddetta *esecuzione predittiva*, in particolare **branch prediction**: si fa un ipotesi sul risultato dell'istruzione di salto condizionale, e si riempie la pipeline con istruzioni che provengono dalla regione corrispondente di programma. Al momento della fase di esecuzione dell'istruzione di salto, si capisce quindi se l'ipotesi si è avverata o meno, e si procede ripulendola completamente introducendo bolle (in caso di *miss*, pagando un prezzo pari al numero di fasi della pipeline) o non facendo nulla (in caso di *hit*).

  Esistono politiche leggermente diverse in caso di salti **diretti** (con indirizzo noto) o **indiretti** (ad indirizzi calcolati).

  - **Salti diretti**: in questo caso si fa una predizione **statica**, cioè si sceglie sostanzialmente a caso fra gli esiti del salto. Esistono comunque alcune euristiche che possiamo usare: nel caso di salti *all'indietro* ci si aspetta di entrare in un loop, e quindi si assume che il salto verrà eseguito; di contro per salti *in avanti* l'ipotesi è meno forte e ci si aspetta che il salto non verrà eseguito.

Abbiamo quindi a grandi linee la struttura funzionale:

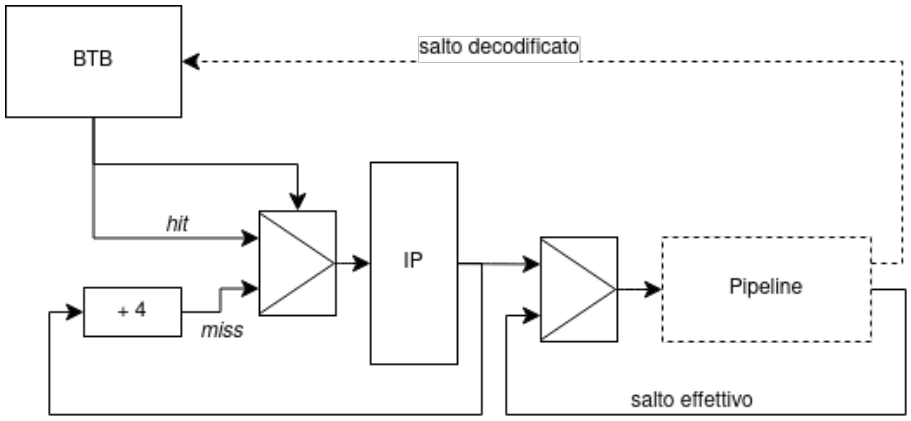


dove una qualche rete di branch prediction valuta le regole appena valutate per ogni istruzione di salto decodificata dalla pipeline, e invia al primo stadio della pipeline stessa la prossima istruzione che ne ricava. Notiamo che il $+4$ corrisponde al calcolo dell'indirizzo della prossima istruzione sequenziale: si assume che l'architettura abbia istruzioni RISC fisse a 4 byte (vedremo che nelle architetture moderne la semplificazione non è inopportuna).

– **Salti indiretti:** Per questi ci si aspetta che i salti si comporteranno come si sono comportati in precedenza, cioè si fa una predizione **dinamica**. Questo compito è associato ad una componente detta **BTB**, *Branch Target Buffer*, che si occupa di capire se un salto verrà effettuato o meno sulla base della sua storia precedente, e a quale locazione si salterà, sempre basandosi sulla storia precedente, nel caso di salti indiretti.

  Questi circuiti assumono essenzialmente la forma di cache, e visto che il loro insuccesso (*miss*) risulta solo in un annullamento di un ciclo di pipeline, non si preoccupano di gestire le collisioni.

  In questo caso la struttura funzionale è del tipo:



  cioè sostanzialmente analoga alla precedente ma dove l'ipotesi sul prossimo salto viene fatta dall'hit/miss della cache BTB.

24.1.3 Architettura del Pentium Pro

Nei processori di oggi, la gestione della pipeline è effettivamente quella che Intel sviluppa dal Pentium Pro del 1995 (che ha continuato ad evolvere fino ad oggi, se non per una deviazione che fu esplorata nel Pentium 4).

In questa architettura, il processore si occupa di tradurre intermante le istruzioni CISC in istruzioni RISC, e quindi a gestire la pipeline con sole istruzioni RISC. Vorremo quindi rimuovere le fasi di prelievo e decodifica dalla pipeline vista finora, per porle esternamente come parte del ciclo di traduzione da CISC a RISC, e aspettarci che la pipeline vera e propria si veda arrivare istruzioni già decodificate.

Notiamo infine che spesso nemmeno l'esecuzione sequenziale del codice non è necessaria. Prendendo ad esempio il frammento di codice:

```
for(int i = 0; i < 1000; i++) {
  a[i] = v1[i] * v2[i];
}
```

potremmo "srotolarlo" in:

```
a[0] = v1[0] * v2[0];
a[1] = v1[1] * v2[1];
// ...
a[999] = v1[999] * v2[999];
```

24.1.4 Esecuzione asincrona

Vediamo che nessuna di queste istruzioni dipende dalle altre, ergo l'ordine in cui vengono eseguite non è importante. In questo caso il miss di cache per alcune di queste può tradursi semplicemente in un ritardo nella loro esecuzione, mentre altre che invece si trovano in cache (magari relative ad indirizzi successivi) possono essere eseguite da subito, cioè l'esecuzione asincrona è non solo possibile, ma può risultare anche utile.

24.1.5 Esecuzione predittiva

Un'ultima tecnica che il processore moderno può usare per accelerare l'esecuzione delle istruzioni è eseguire istruzioni prima che questo sia necessario, eventualmente scartando i risultati nel caso risultino inutili.

25 Lezione del 06-05-25

Riprendiamo più nel dettaglio i concetti che abbiamo solamente accennatto alla scorsa lezione.

25.0.1 Ordinamento delle istruzioni

Abbiamo che l'*ordinamento totale* delle istruzioni del programma è solo un artefatto di come questo è compilato: non necessariamente le istruzioni vanno eseguite in tale ordine per arrivare allo stesso risultato finale.

Le istruzioni si trovano infatti solo in un *ordinamento parziale*, dove solo alcune istruzioni hanno bisogno, per motivi di sincronia, di essere eseguite successivamente ad altre.

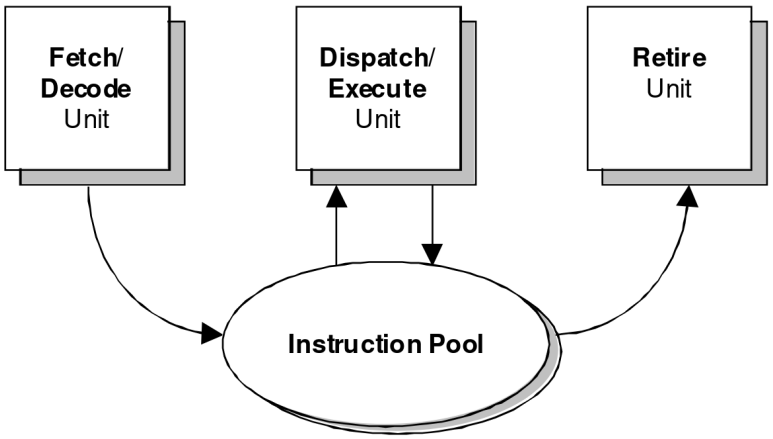
Prendiamo quindi l'esempio `op1, src1, sec2, dst`, con sintassi simile all'assembly ARM (e in generale delle istruzioni RISC, dove ricordiamo gli operandi sono sempre registri). Una volta che i sorgenti `src1` e `src2` sono definiti, questa può essere eseguita.

Ipotizziamo quindi un'architettura dove sono previste un numero arbitrario di ALU, preceduta da componenti che denominiamo **stazioni di prenotazione**, fondamentalmente registri capaci di contenere la codifica macchina di una istruzione.

Ogni istruzione che viene decodificata dal processore viene spostata in una stazione di prenotazione. Non appena gli operandi saranno pronti, l'istruzione potrà quindi essere messa in esecuzione.

Viene da sé che questa architettura ci permette di ottenere un'esecuzione delle istruzioni che è *fuori ordine* e in *parallelo*.

Vediamo quindi nel dettaglio lo schema funzionale del Pentium Pro:



I 3 blocchi funzionali si occupano rispettivamente di:

1. **Fetch/Decode** Unit: come abbiamo detto in 24.1.3, si occupa di convertire le istruzioni CISC in micro-operazioni RISC da 4 KiB ciascuna. Le istruzioni CISC più complesse fornite dall'instruction set x86 vengono tradotte in 2 o più micro-operazioni.

   Le micro-operazioni vengono poi introdotte nella cosiddetta **instruction pool**, cioè una struttura che mantiene le micro-operazioni decodificate. L'instruction pool viene riempita con un certo grado di *look-ahead*, cioè si ottengono, in maniera speculativa, istruzioni successive al punto di esecuzione corrente.

   Infine, notiamo che nella pratica l'instruction pool viene implementata da un buffer detto **ROB**, *Reorder Buffer*;

2. **Dispatch/Execute** Unit: si occupa di prelevare le istruzoni dal ROB ed effettuarne il *dispatch* (**emissione**) all'interno delle stazioni di prenotazione. Queste si occupano quindi di eseguire l'istruzione ottenuta, secondo le regole sulle dipendenze che vedremo fra poco, e probabilmente attraverso l'arbitrio di una qualche unità di controllo superiore;

3. **Retire** Unit: questo è il componente finale della catena, che si occupa di prelevare dal ROB le istruzioni terminate, ed effettuarne il **ritiro** nell'ordine in cui sono state scritte dal programmatore.

25.0.2 Dipendenze

Chiaramente, resterà da definire le regole secondo le quali le istruzioni devono essere eseguite prima o dopo di altre nello schema di esecuzione fuori ordine. Chiamiamo queste condizioni **dipendenze**, fra cui distinguiamo:

- Dipendenze **dati**;
- Dipendenze **nomi**;
- Dipendenze **controllo**;

dove la prima e l'ultima non vanno confuse con le alee: adesso parliamo di conseguenze di come è fatto il *programma*, non di come è fatto il *processore*.

Le dipendenze **dati** sono il caso che abbiamo già visto. Se abbiamo due istruzioni:

```
add r1, r2, r3
...
sub r4, r3, r5
```

si ha che la `sub` dipende per dati dalla `add`, in quanto questa intacca `r3`, operando.

Per ogni registro dovremo quindi mantenere delle informazioni associate: quando una istruzione viene messa in una stazione (si dice viene **emessa**), si alza un certo flag associato al registro che intaccherà. Questo flag viene poi abbassato in fase di esecuzione vera e propria, insieme all'aggiornamento dei dati del registro stesso. Un istruzione successiva che vuole usare tale registro dovrà quindi controllare tale flag per capire se quell'operando è pronto.

Le dipendenze sui **nomi** si dividono in due categorie:

- **Antidipendenze**: Poniamo di avere le due istruzioni:

  ```
  add r1, r2, r3
  ...
  sub r4, r5, r1
  ```

  in questo caso non si può eseguire la `sub` prima della `add`, in quanto la prima necessita del *vecchio* contenuto di `r1`, non quello che avremo dopo la `sub`. Si deve quindi evitare di mettere in attesa la `sub` finché la `add` non è conosciuta.

  Per risolvere questo tipo di dipendenza dovremmo quindi dotare ogni registro, oltre al flag W, un contatore C che conti quante stazioni contengono istruzioni che usano quel registro come sorgente. Ogni istruzione emessa alza i contatori dei suoi sorgenti, e ogni istruzione eseguita fino in fondo li abbassa. Avremo quindi che un istruzione potrà scrivere sulla sua destinazione quando il numero di lettori, e quindi il contatore, è pari a 0.

- Dipendenze in **uscita**: Poniamo di avere le due istruzioni:

  ```
  add r1, r2, r3
  ...
  sub r4, r5, r3
  ```

  che può sembrare strano, ma può succedere nel caso qualcuno usi `r3` fra le due istruzioni, o il compilatore lo faccia comunque per motivi di ottimizzazione. Ad esempio questo succede spesso per quanto riguarda il registro dei flag, che viene aggiornato costantemente ma letto solamente dalle istruzioni di salto.

Per risolvere questo tipo di dipendenze si controlla il flag associato al registro di uscita prima dell'emissione: in caso questo sia già preso in scrittura si mette l'istruzione in attesa (simile alle bolle che avevamo visto per la pipeline), per poi emetterla solo quando `r3` è stato modificato.

Possiamo dire che, date due istruzioni a, b da eseguire in quest'ordine, le dipendenze si classificano come nel seguente schema:

	a in scrittura	a in lettura
b in scrittura	Dipendenza in uscita	Antidipendenza
b in lettura	Dipendenza sui dati	//

Le dipendenze sul **controllo** si verificano quando istruzioni possono o meno essere eseguito sulla base dell'esito di istruzioni di salto. Ad esempio, preso:

```
  cmp
  ja fine

  add ...
  sub ...

fine:
  mul ...
```

vorremo che le `add` e `sub` non fossero eseguite in caso positivo della `ja`, mentre la `mul` venga eseguita comunque. Chiaramente questo non è facile, in quanto potremmo avere:

```
  cmp
  ja fine

  add ...
  sub ...

  jmp fine2

fine:
  mul ...

fine2:
```

dove la `mul` non può essere eseguita comunque, ma questo non si può sapere finché non si entra nel blocco `add ... sub`. Il processore non ha quindi speranze per risolvere il problema, se non assumere *per eccesso* che tutto ciò che viene dopo la jump dipende dalla jump.

25.0.3 Traduzione da CISC a RISC

Vediamo come può essere effettuata la traduzione da un istruzione CISC al corrispondente insieme di istruzioni RISC. Prendiamo ad esempio:

```
add %rax, 1000(%ebx, %ecx, 8)
```

questa dovrà essere tradotta in qualcosa come:

```
shl %ecx, $3, tmp1
add %ebx, tmp1, tmp1
ld 1000(tmp1), tmp2
add %rax, tmp2, tmp2
st tmp2, 1000(tmp1)
```

Vediamo come abbiamo bisogno di registri dedicati, `tmp1` e `tmp2`, interni al processore e non accessibili al programmatore. Questo chiaramente perchè non vogliamo che la trasformazione in RISC delle istruzioni CISC sporchi i registri programmatore.

25.0.4 Registri fisici

Dotando il processore di più registri fisici, oltre a quelli programmatore, possiamo rimuovere le dipendenze sui nomi, dette anche *false dipendenze*.

Questo si fa attraverso il meccanismo di **rinominazione** dei registri. Facciamo in modo che ogni registro logico punti ad un registro fisico, con il numero di registri fisici anche maggiore di quello dei registri logici.

Quando il processore incontra un'istruzione, chiama un registro fisico non appena usato come come il registro di uscita (per tutti, non solo per se stesso), e prende i registri fisici puntati dai registri logici sorgenti come registri sorgenti. Ad esempio, assunto di partire da una mappatura identità, prendiamo l'istruzione:

```
1 add r1, r2, r3
```

che viene trasformata in:

```
1 add f1, f2, f6 # assunto f6 libero
```

dove *libero* significa non usati per niente, quindi non puntati e con contatore C e flag W a 0.

Notiamo che questa fase di traduzione si svolge *dopo* la fase di traduzione in RISC effettuata nei componenti addetti alla fetch e decode, e quindi solo in fase di emissione.

Abbiamo quindi rimosso tutti gli stalli necessari alle dipendenze fra i nomi (le cosiddette *false dipendenze*), in quanto ogni istruzione emessa ha come destinazione un registro fisico libero.

Un'istruzione che ha già avuto i suoi registri logici sorgenti tradotti in registri fisici non si preoccupa di ulteriori aggiornamenti ai registri sorgenti, in quanto queste verranno fatte su altri registri, liberi, e non su quelli che lei ha scelto.

Vediamo quindi come usare il meccanismo di ridenominazione dei registri per risolvere anche gli stalli sulle dipendenze di controllo.

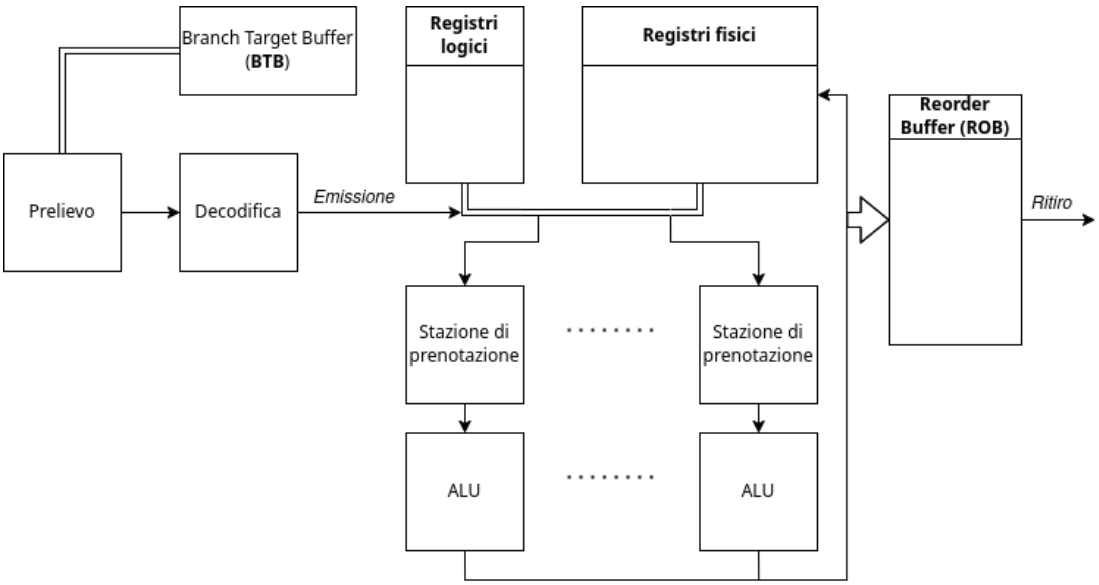
Avevamo già introdotto l'esecuzione speculativa delle istruzioni. Nella terminologia Intel, questa viene eseguita attraverso un **ROB**, *Reorder Buffer*. All'interno del Reorder Buffer viene mantenuta una lista ordinata delle istruzioni, assieme ad un flag che indica che quell'istruzione è terminata (ma non che il risultato è stato scritto!).

Prima di un'istruzione di salto, si possono eseguire e ritirare tutte le istruzioni. Dopo l'istruzione di salto, immaginiamo che le istruzioni successive vengano eseguite ma non ritirate (scritte nei registri fisici), secondo una predizione fatta come descritto in 24.1.2. A seguito del salto, potremmo quindi verificare se la predizione è stata corretta: in caso di hit, avremo che potremo ritirare le istruzioni già eseguite, altrimenti dovremo svuotare il Reorder Buffer.

In ogni caso, informazioni più dettagliate sul funzionamento dell'esecuzione fuori ordine nel Pentium Pro si possono trovare nel manuale dello stesso, reperibile al link: `https://download.intel.com/design/PentiumII/manuals/24400101.pdf`.

26 Lezione del 09-05-25

Riprendiamo il discorso sull'architettura del processore. Possiamo fare un riassunto dell'architettura descritta finora:



Abbiamo quindi che le istruzioni provengono dal Branch Target Buffer, **BTB**, dalle quali vengono poste in una componente di **prelievo** e una di **decodifica**.

Da qui i poi le istruzioni passano per le stazioni di prenotazione, che poi le inviano alle relative ALU, sulla base delle tabelle associate ai registri fisici.

In fine, in fondo a questo meccanismo si trova il Reorder Buffer, **ROB**, che contiene le istruzioni terminate ma non ancora scritte e permette l'esecuzione speculativa. Il significato del Reorder Buffer è sostanzialmente quello di realizzare un'ulteriore smistamento delle istruzioni a termine della loro esecuzione, che permette o meno il loro write back nei registri (che sappiamo coi registri ridenominati significa semplicemente aggiornare i registri logici) solo sotto le condizioni dell'esecuzione speculativa.

26.0.1 Istruzioni LOAD e STORE

L'esecuzione speculativa vista presenta problemi per quanto riguarda le istruzioni `LOAD` e `STORE`.

Prendiamo ad esempio il framento di codice:

```
  cmp $1000, %rbx
  jae fine
  mov off(%rbx), %rax

fine:
```

Qui la scrittura va eseguita solo se `%rbx` contiene un valore inferiore a mille. Per le regole dei salti in avanti, il salto in `jae fine` si da per non fatto, e quindi il BTB fornisce l'istruzione successiva, cioè quella di accesso in memoria (che verrebbe tradotta in una `load`). capito 0, per le store come fa?

26.0.2 Istruzioni IN e OUT

Infine, una nota va fatta sulle istruzioni di I/O `in` e `out`: queste infatti hanno conseguenze su dispositivi esterni al processore per cui non ne si può fare l'esecuzione parallela o speculativa.

In fase di emissione di istruzioni di I/o, quindi, il processore si aspetta che tutte le istruzioni attualmente in esecuzione siano state eseguite, e solo dopo procede con l'emissione. e poi?

26.1 Vulnerabilità Meltdown e Spectre

Meltdown e **Spectre** sono state vulnerabilità dell'architettura Intel x86, scoperte nel 2017, che permettevano di superare le limitazioni normalmente imposte al codice in esecuzione in modalità utente.

Abbiamo detto che il programma dovrebbe poter ignorare ciò che accade nel processore a livello **microarchitetturale**, cioè che tutte le operazioni che l'architettura descritta finora compie ai fini di ottimizzare l'esecuzione devono risultare **invisibili** al programma.

Ciò che queste vulnerabilità hanno rivelato è che lo stato microarchitetturale, invece, non è veramente invisibile.

26.1.1 Meltdown

Ad esempio, per quanto riguarda la cache, abbiamo che si può in qualche modo capire se qualcosa è scritto in cache: bastera invalidare una cacheline, far eseguire il processo che vogliamo studiare, e quindi tentare un'accesso alla stessa cacheline: valutando il tempo necessario all'accesso (e magari facendo una media statistica) si potrà capira se quella cacheline è piena oppure no.

Questa era l'approccio adottato dalla vulnerabilità Meltdown.

```
togli_cacheline

mov indirizzo_vietato, %al # qui il programma muore
# da qui in poi si esegue in speculativa
shl $12, %al
mov vettore (%rax), %rbx

# questo si mette nell'handler del segmentation fault
controlla_cacheline
```

L'istruzione illegale `mov indirizzo_vietato, %al` causerà chiaramente l'arresto del programma, ma le due istruzioni successive verranno comunque eseguite dal processore in modalità speculativa, prima che si manifesti il fault.

La `controlla_cacheline`, invece, viene eseguita comunque attraverso una ridefinizione dell'handler di segmentation fault.

A questo punto, si ha quindi che l'indirizzo letto in `%al`, che il processo non sarebbe autorizzato a vedere, viene trasformato in un indice in un vettore, vettore il cui accesso si traduce nel riempimento di una cachline che rileviamo in `controlla_cacheline`.

Iterando questo processo su tutta la memoria kernel si riesce a ricostruirne una copia, e quindi effettivamente leggere tutta la memoria fisica.

Il problema oggi è stato risolto forzando l'invalidazione della cache ad ogni passaggio al kernel.

26.1.2 Spectre

La vulnerabilità Spectre si basa sulla natura stessa del branch prediction: l'utente può sfruttare la struttura del BTB, che abbiamo detto è effettivamente una cache senza controlli sulle collisioni, per "addestrare" il predittore di branch a fare scelte arbitrarie.

In questo modo si potrà in qualche modo "direzionare" l'esecuzione del kernel in luogo dei salti condizionali, e poi usando metodi come Meltdown capire che cosa il kernel ha fatto.

Anche questo problema si può risolvere invalidando una cache in fase di passaggio al kernel, e in particolare invalidando il BTB.