

Calcolatori Elettronici

Francesco Bonistalli, Luca Minuti

Anno accademico 2023/2024

Indice

1 Introduzione	6
1.1 Lezione 1 26/02/2024	8
1.1.1 Grandezza (in byte) dei tipi in c++	9
1.1.2 Funzioni	9
1.2 Lezione 2 27/02/2024	10
1.2.1 Come realizzare questa (utile) supposizione	12
1.2.2 Meccanismo di chiamata	12
1.2.3 Registri nel processore a 64 bit	13
1.2.4 I parametri	14
2 Indirizzi e oggetti	16
2.1 Lezione 4 01/03/2024	16
2.1.1 Indirizzi	16
2.1.2 Offset	17
2.1.3 Intervalli	17
2.1.4 Allineamento e scomposizione	18
3 La memoria centrale	20
3.1 (Continuo Lezione 4)	20
3.1.1 Come viene usata la memoria	20
3.1.2 Esempio di come avviene una richiesta di operazione di lettura	21
3.1.3 Montaggio della RAM	21
3.1.4 Allineamento dei dati in C++	22
4 Memoria Cache	24
4.1 Lezione 7 11/03	24
4.1.1 CACHE	24
4.1.2 Cache ad indirizzamento diretto	25
4.1.3 Cache associative ad insiemi	27
5 Periferiche	28
5.1 Lezione 8 12/03	28
5.1.1 Tastiera	28
5.2 Lezione 9 14/03	28
5.2.1 Scheda video	29
5.2.2 Memoria video	30
5.2.3 Timer	30
5.3 Lezione 10 15/03	31

5.3.1 Hard Disk	32
6 Interruzioni	35
6.1 Lezione 11 18/03	35
6.1.1 Programmazione a eventi	35
6.2 Lezione 12 19/03	40
7 Eccezioni	41
7.1 Lezione 14 22/03	41
7.1.1 Eccezioni	41
8 Protezione	43
8.1 Lezione 15 25/03	43
8.1.1 Protezione	43
9 Introduzione al sistema multiprogrammato	47
9.1 Lezione 16 26/03	47
9.1.1 Processi	47
10 Realizzazione dei processi	50
10.1 (Continuo Lezione 16)	50
10.2 Lezione 17 04/04	54
10.2.1 Atre note sui processsi	54
11 Realizzazione delle primitive	56
11.1 (Continuo Lezione 17)	56
11.1.1 Realizzazione delle primitive	56
11.1.2 Meccanismo di chiamata	57
11.1.3 Realizzazione di una nuova primitiva	57
11.1.4 Funzioni di supporto per le primitive	58
11.1.5 Chi esegue le primitive	58
12 Semafori	60
12.1 Lezione 18 05/04	60
12.1.1 Meccanismo dei Semafori	61
13 Sospensione dei processi	64
13.1 Lezione 19 08/04	64
13.1.1 Sospensione dei processi	64
13.1.2 Driver del timer	66
14 Paginazione	68
14.1 Lezione 22 12/04	68
14.1.1 Memoria virtuale	68
14.1.2 Indirizzi virtuali	71
14.1.3 Paginazione	71
14.1.4 Come si realizza l'hardware che fa questo (architettura intel-amd)	72
14.1.5 Funzioni aggiuntive:	73
14.2 Lezione 23 15/04	74

15 Tabelle multilivello	75
15.1 Continuo Lezione 23	75
15.1.1 Irrealizzabilità della super-MMU	75
15.1.2 TRIE-MMU	76
16 Paginazione: complementi	79
16.1 Lezione 24 18/04	79
16.1.1 Come gestire la memoria	80
16.1.2 TLB	81
16.2 Lezione 25 19/04	82
17 Funzioni di supporto per la paginazione	84
17.1 Lezione 26 22/04	84
17.1.1 Funzioni di supporto per la paginazione	84
18 Implementazione della memoria virtuale	87
18.1 Lezione 28 29/04	87
18.1.1 Implementazione della memoria virtuale nel nucleo	87
18.1.2 Creazione delle parti condivise	88
19 Il bus PCI	90
19.1 Lezione 29 30/04	90
19.1.1 Modulo I/O	90
20 I/O nel nucleo	94
20.1 Lezione 30 03/05	94
20.1.1 I/O nel nucleo	94
20.1.2 Realizzazione con primitiva e driver	95
21 Modulo I/O	99
21.1 Lezione 31 06/05	99
21.1.1 Modulo I/O	99
21.1.2 Funzionamento effettivo del modulo I/O	99
22 DMA e PCI Bus Mastering	103
22.1 Lezione 32 13/05	103
22.1.1 DMA	103
22.1.2 Interazione DMA con la cache	104
22.1.3 Cache con politica write-through	105
22.1.4 Cache con politica write-back	106
22.1.5 Interazione con la memoria virtuale	107
22.1.6 PCI Bus Mastering	108
23 La pipeline	110
23.1 Architettura interna del processore	110
23.1.1 Pipeline	110
23.1.2 Processori RISC	111
23.1.3 Le e-istruzioni	111
23.1.4 Alee	111

23.1.5 Esecuzione fuori ordine	112
23.1.6 Organizzazione interna del processore	112
23.1.7 Esecuzione speculativa	113
23.1.8 Dipendenze nelle istruzioni load e store	114
24 Lezioni di Laboratorio	115
24.1 Lezione 3 (Laboratorio) 29/02/2024	115
24.2 Lezione 5 (Laboratorio) 05/03/2024	116
24.2.1 Mangling dei nomi	116
24.3 Lezione 6 (Laboratorio) 07/03	117
24.4 Lezione 13 (Laboratorio) 21/03	117
24.5 Lezione 20 (Laboratorio) 09/04	118
24.6 Lezione 21 (Laboratorio) 11/04	118
24.7 Lezione 27 (Laboratorio) 23/04	118
25 Appunti laboratorio Leonardi	120
25.1 Lab 6	120
25.1.1 Parte 1	120
25.1.2 Parte 2	120
25.2 Lab 10	120
25.2.1 Parte 1	120
25.2.2 Parte 2	121
25.3 Lab 14	121
25.3.1 Parte 1	121
25.3.2 Parte 2	121
25.4 Lab 24 (semafori mutex)	122
25.5 Lab 26 (05/02/2011 concessione di permessi scrittura/lettura)	123
25.6 Lab 34 (02/07/2015 Memoria virtuale)	123
25.7 Lab 38 (settembre 2021 Memoria virtuale)	124
25.8 Lab 42 (15/06/2016 I/O)	124
25.9 Lab 45 (06/07/2016 I/O)	125
25.10Lab 46	125
26 Alcune domande orale	126

Prefazione

Il documento è stato realizzato a partire dagli appunti presi durante l'anno accademico 2023/2024 e integrati con le dispense del solito. Minori modifiche sono state apportate in riferimento alle dispense dell'anno accademico 2024/2025.

Capitolo 1

Introduzione

I tre principali argomenti di cui tratteremo sono:

$$\left\{ \begin{array}{l} \text{interruzioni} \\ \text{protezione} \\ \text{memoria virtuale} \end{array} \right. \rightarrow \text{multiprogrammazione (fa riferimento a un solo processore (core))}$$

Nota: *In questo corso trattiamo sempre il caso in cui si ha 1 processore (il caso di più processori si vede alla magistrale). È importante non confondere il fatto che vediamo sul nostro pc molti programmi con l'avere più processori sulle nostre macchine (core), la multiprogrammazione infatti **non** fa riferimento all'avere più CPU.*

Domanda fondamentale che ci porremo durante il corso: "**chi fa cosa**"?

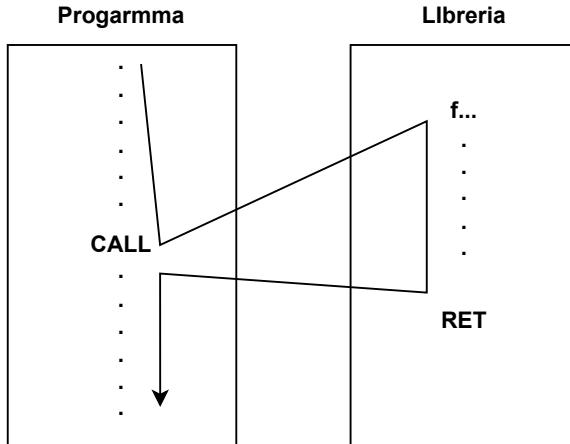
Per capire bene il concetto è bene partire da: "*chi comanda un computer dentro un computer?*" La risposta più corretta è **il software**, ovvero quell'insieme di dati che sta in memoria. Questo è ciò che distingue un computer, costituito da un'architettura progettata per eseguire software, dalle altre macchine (ad esempio una lavatrice) in quanto permette di essere programmato. Avendo quindi bisogno di dati è utile chiedersi "*chi sa cosa?*", questo perché chi comanda il processore è il software: esso infatti decide il flusso di programmazione.

Nota: *Non è il processore a comandare un computer in quanto si limita a fare quello che gli viene detto da ciò che c'è scritto in memoria: in ogni istante la CPU è "concentrata" sull'istruzione corrente e non sa nulla di tutto il resto (quello che è successo prima e quello che succederà dopo).*

Nota : Memoria: *non confondersi su questi aspetti:*

- *Tenere bene a presente il fatto che le celle di memoria hanno degli indirizzi ma quest'ultimi non sono scritti da nessuna parte.*
- *Essa contiene sempre qualcosa*
- *Il significato dei byte nella memoria non sta scritto da nessuna parte, il contenuto assume un significato in base all'uso che se ne fa (byte visto come istruzione o come elemento di memoria ha due significati differenti)*

Esempio 1: Nel processore che studiamo c'è un solo flusso di controllo (dato che abbiamo un solo processore (core)).



Il software con la CALL cede il controllo alla libreria (il processore, che esegue e basta, non si accorge di nulla). □

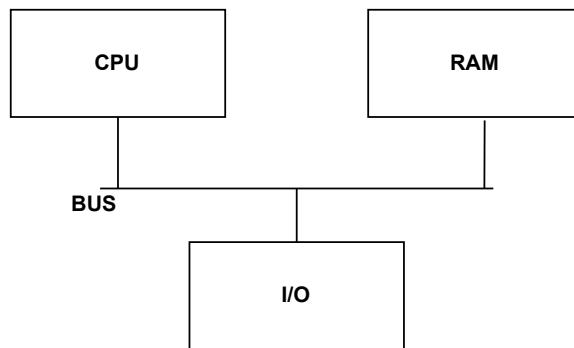
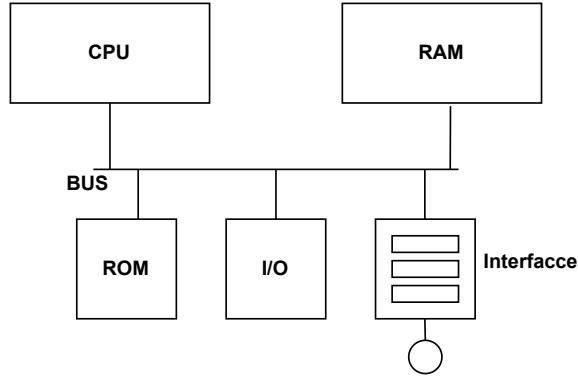


Figura 1.1: Schema essenziale calcolatore

- **BUS:** una sola scatola alla volta può parlare mentre tutte possono ascoltare. Tramite gli indirizzi (sfruttati per creare maschere) le scatole che ascoltano capiscono chi deve rispondere.
- **RAM:** è la parte più importante perché è quella attorno alla quale tutto ruota e, data la sua struttura, tutto il resto è stato costruito. Essa è in grado di accedere velocemente, tramite gli indirizzi, a qualunque dei byte di cui è composta, tutto ciò indipendentemente da cosa è successo prima. In questa architettura **programmare** la macchina significa infatti **decidere cosa c'è inizialmente nella RAM**, noto anche come il suo stato iniziale. L'idea centrale di questa architettura, perciò, sono gli indirizzi (in quanto si riferiscono alla RAM).
- **I/O:** l'idea di base per la sua realizzazione è "portare tutto a qualcosa che somigli alla memoria" (in modo da sfruttare ancora gli indirizzi).



Per programmare nello stato iniziale qualcosa nella RAM ho bisogno di una ROM contenente un programma di bootstrap. Questo nome viene dal modo di dire tipicamente americano "by his own bootstrap", ovvero "fare da soli" (riferito anche cose impossibili). Situazioni paradossali infatti si ritrovano spesso nell'informatica, ad esempio il compilatore del *C* è scritto in *C* (ci verrebbe da chiederci *"chi compila il compilatore?"*). La soluzione del mistero è che qualcuno prima ha scritto in assembler il compilatore del compilatore del *C*, che a sua volta è stato scritto prima in linguaggio macchina ecc.

Nota: *Programmare questa architettura significa decidere il contenuto iniziale della RAM.*

Nota: *Il software fa qualcosa solo se il flusso di controllo passa da esso, non è sufficiente che stia in RAM e basta.*

Nota: *In memoria ci può scrivere solo la CPU, sui registri dell'IO possono avvenire scritture anche da parte delle periferiche.*

Mentre stiamo studiando ci domanderemo spesso cosa è fatto in hw e cosa in sw, per capirlo dobbiamo tenere in mente che:

- se pensiamo sia fatto in hw: quali elementi conosce questo hw? (da rivedere)
- se pensiamo sia fatto in sw dobbiamo chiederci: *"so scrivere questa cosa?"*

1.1 Lezione 1 26/02/2024

Il termine compilatore non è esattamente corretto: sarebbe più opportuno parlare di traduttore.



Il traduttore non ci serve per far girare in senso stretto i nostri programmi (se devo leggere un libro in una lingua che non conosco non mi serve il traduttore, può anche essere morto per quanto riguarda i miei scopi, basta avere il libro tradotto).

Prendiamo quindi come esempio un file c++ che contiene dichiarazioni e definizioni di variabili, ad esempio:

```

1 int i;
2 long j;

```

Queste due dichiarazioni e definizioni hanno come effetto di riservare dello spazio in memoria per un `int` e per un `long` e ricordarsi come si chiamano.

In assembler quindi nelle sezione `.data` verrà scritto:

```
1 .data
2 i: .dword 0
3 j: .quad 0
```

1.1.1 Grandezza (in byte) dei tipi in c++

	ABI	Windows
<code>char</code>	1	1
<code>short</code>	2	2
<code>int</code>	4	8
<code>long</code>	8	8
<code>bool</code>	1	1

Nota: Noi useremo un ambiente linux e quindi le dimensioni ABI

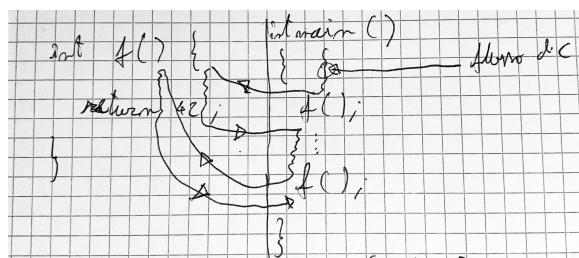
Nota: Il traduttore (compilatore) se trova dei byte che rappresentano una istruzione la traduce, altrimenti ignora; ad esempio:

```
1 .text
2 f:
3 .
4 .
5 .
6 .long 0xCC
```

Quando il flusso di controllo incontra quella "istruzione" alla riga 6 prende quel "cc" e lo dimentica: non è una istruzione eseguibile.

1.1.2 Funzioni

Per affrontare questo argomento bisogna tenere ben presente l'unicità del flusso di controllo.



Il motivo della scelta (ai tempi) di questa architettura è che il codice potesse modificare se stesso: prima di chiamare la funzione modifico l'indirizzo della JMP finale che contiene

l'indirizzo di ritorno con l'indirizzo dell'istruzione successiva alla chiamata di funzione, una volta fatto ciò eseguo la chiamata di quest'ultima.

Nota: questo metodo ora è completamente o quasi abbandonato.

In una prima evoluzione si sfruttavano i registri per salvare gli indirizzi di ritorno, più recentemente si è iniziato a sfruttare la pila.

1.2 Lezione 2 27/02/2024

Nota: Noi usiamo il compilatore gcc/g++.

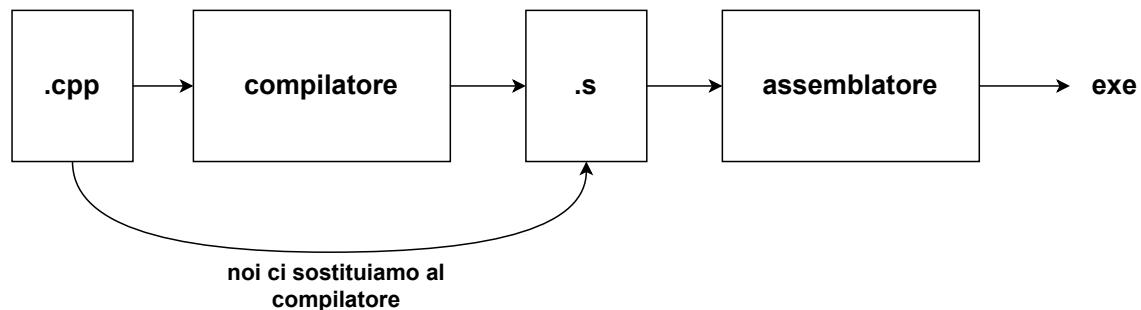


Figura 1.2: "exe" contiene il contenuto iniziale della memoria (programmare questa macchina infatti significava fare questo).

Nota : Differenza tra sezioni `.data` e `.text`: sono sempre e comunque byte copiati in memoria ma in `.text` il compilatore se riesce protegge dalla scrittura.

Il meccanismo di chiamata di base sulla **pila**:

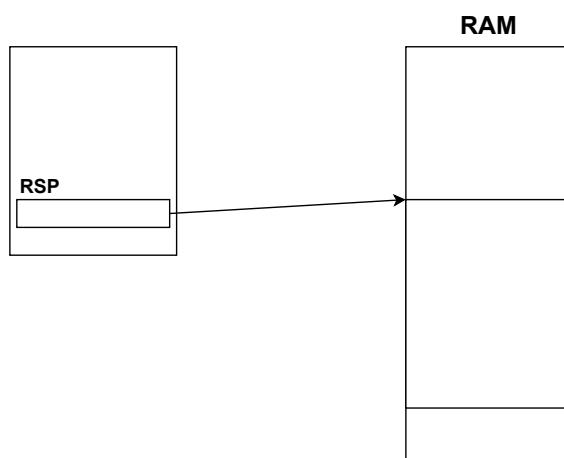


Figura 1.3: RSP è il registro puntatore al top della pila

Nota: la pila è una parte della RAM.

Ci sono istruzioni (PUSH, CALL PUSHF, POP, RET, POPF) che usano implicitamente questo registro (RSP), vale a dire lo usano senza nominarlo esplicitamente ma tramite altri meccanismi. In particolare abbiamo:

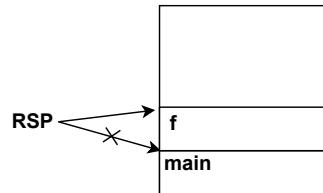
- le istruzioni PUSH, CALL e PUSHF che decrementano **sempre** di 8 l'indirizzo puntato (8 perché siamo a 64 bit non più a 32).
- le altre (POP, RET, POPF) incrementano **sempre** di 8.

Per il resto RSP è un normale registro e in quanto tale posso inizializzarlo (es. tramite una MOV), anche se di solito a fare questa operazione è una libreria.

Nota: *Il fatto che queste istruzioni vengano usate a coppie (PUSH con la POP ecc.) è una convenzione del programmatore, al compilatore non interessa niente di ciò in quanto vede semplicemente delle istruzioni.*

Negli anni 70 quando si dichiaravano delle funzioni si riservava ad esse spazio in memoria, al loro interno poi si riservava spazio per le variabili che usavano. Questo è però uno spreco di memoria in quanto non è detto che verranno mai usate contemporaneamente tutte le funzioni (e con esse le loro variabili). Inoltre in caso di funzione ricorsiva, non essendoci un modo assoluto di sapere a priori quante istanze servissero di essa, non si sapeva per quante riservare spazio in memoria.

Si è quindi giunti a una idea migliore sapendo che l'ordine di utilizzo delle funzioni è del tipo LIFO. Dato questo comportamento si può infatti sfruttare la pila:



In questo modo a partire dal *main* si alloca spazio per una funzione alla volta, supportando in questo modo anche la ricorsione (si alloca ogni volta spazio per ogni chiamata).

Esempio 2:

```

1 int fact(n){
2     if (n == 0) return 1;
3     return n * fact (n - 1);
4 }
```

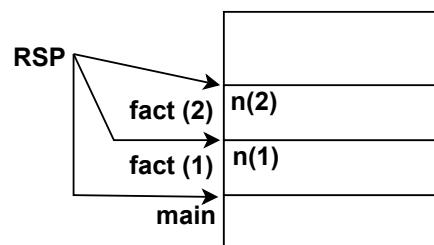


Figura 1.4: supponiamo che *n* si trovi sempre al solito offset rispetto a RSP

□

1.2.1 Come realizzare questa (utile) supposizione

Ogni volta in cima alla pila si aggiunge un record di attivazione. Per farlo usiamo un registro, RBP ovvero "base pointer", che punta sempre al record di attivazione rispetto al quale gli offset rimangono costanti.

Nota: Quest'ultima considerazione vale per il codice che stiamo scrivendo, non è un concetto assoluto, è infatti una convenzione non ottimizzata ma (noi) la useremo sempre.

Si deve quindi realizzare per ogni funzione un prologo e un epilogo:

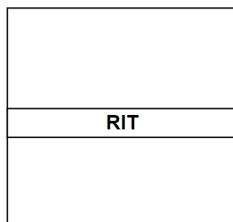
```
1 prologo: PUSH %RBP
2         MOV %RSP, %RBP
3         .
4         .
5         .
6 epilogo: MOV %RBP, %RSP
7         POP %RBP
8         RET
```

Nota: Le prime due righe dell'epilogo possono essere sostituite da una sola istruzione (LEAVE)

In questo modo abbiamo creato il record di attivazione.

1.2.2 Meccanismo di chiamata

Si parte da una situazione iniziale del tipo:



Che si evolve, alla chiamata della funzione, in questo modo:

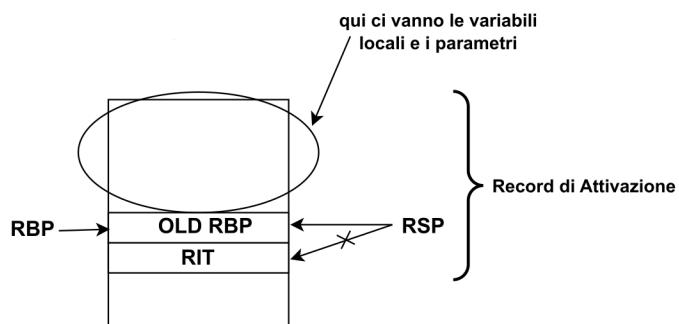


Figura 1.5: RSP all'inizio punta all'indirizzo di ritorno salvato dalla *call*, poi cambia l'indirizzo puntato; RBP invece sta fermo in modo tale che tutto quello che sta sopra ha un offset costante rispetto ad esso.

Per allocare lo spazio per le variabili locali decremento RSP: questo si fa a multipli di 8 per mantenere l'allineamento in memoria (in realtà si fa a multipli di 16 perché linux vuole che prima della chiamata ci sia un allineamento rispetto ai multipli di 16).

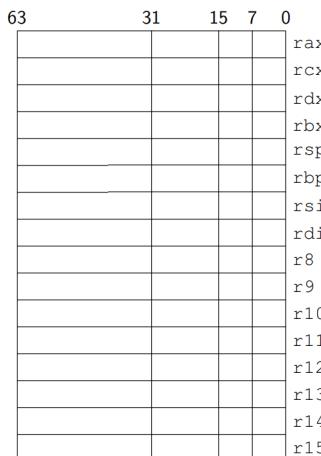
Esempio 3: se n è di 4 byte essa sta da qualche parte in quei 16 allocati per mantenere l'allineamento, non so esattamente dove in quei 16 byte (in realtà ci sono delle regole) perché lo deve sapere solo la funzione, che sfrutta l'offset con RBP, e nessun altro. \square

La PUSH di RBP si fa perché una funzione è stata chiamata da un'altra che a sua volta era stata chiamata ecc. quindi devo salvare il vecchio RBP da qualche parte.

Lo stesso problema si potrebbe presentare per altri registri non potendo sapere con esattezza quali sono stati usati dalla funzione chiamante e che quindi contengono informazioni da non perdere. Si distinguono per questo motivo, secondo una convenzione, i registri **scratch** (che servono alla funzione chiamata) da quelli non scratch. I registri schratch sono quelli che vengono utilizzati di solito nelle traduzioni (non serve fare push e pop di essi), sono temporanei e vengono utilizzati dal compilatore per immagazzinare valori intermedi. I registri **preservati**, invece, devono essere preservati durante la chiamata di una funzione in modo tale che il loro valore non venga modificato dalla funzione stessa. Se li voglio usare all'interno di una traduzione di funzione devo prima fare una push di essi e infine una pop.

1.2.3 Registri nel processore a 64 bit

Schema:



Nota: A tutti i registri si può accedere a 64, 32, 16 e 8 bit; solo RAX, RBX, RCX e RDX però conservano i nomi AH, BH,... ma è meglio non usarli per via di complesse regole.

Nota : Registri scratch: sono:

- RAX
- R8
- RCX
- R9
- RSI
- R10
- RDI
- R11

1.2.4 I parametri

Stanno in pila, come le variabili, ma a differenza di quest'ultime vanno passati da una funzione all'altra. Per attuare questo meccanismo si usano i registri in modo regolamentato da una convenzione che stabilisce quali e con che ordine i registri vadano adoperati, ovvero secondo l'ordine¹:

- | | |
|--------|--------|
| 1. RDI | 4. RCX |
| 2. RSI | 5. R8 |
| 3. RDX | 6. R9 |

Nel prologo quindi i parametri passati vengono copiati dai registri alla memoria (pila).

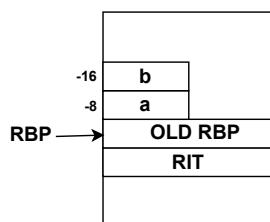
Esempio 4: .

```
1 void f (int a, int b){  
2 }  
3 void g (){  
4     f(3, 5);  
5 }
```

Prima di fare CALL "f" si devono passare i parametri attuali nei registri:

```
1 MOV $3, %RDI  
2 MOV $5, %RSI  
3 CALL f  
4  
5  
6 f: PUSH %RBP  
7     MOV %RSP, %RBP  
8     SUB $16, %RSP //mi basta 16 perche' b sta a indirizzo -16  
9     .  
10    .  
11    .  
12    MOV %RDI, -8(%RBP)  
13    MOV %RSI, -16(%RBP)
```

□

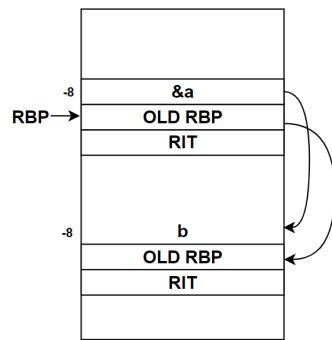


Questo era il passaggio dei parametri per valore, analizziamo ora quello per riferimento:

```
1 void f (int &a){  
2     a = 1;
```

¹Ordine trovato in modo sperimentale, se (non succederà nel nostro corso) servano più registri si usa la pila.

```
3 }
4
5 void g () {
6     int b;
7     f(b);
8     //b == 1
9 }
```



Alla chiamata di f devo passare l'indirizzo di b:

```
1 LEA -8(%RBP), RDI
```

Questo mi permette di fare 0 accessi in memoria (uso la LEA perché devo fare un calcolo).

Capitolo 2

Indirizzi e oggetti

2.1 Lezione 4 01/03/2024

Definizione 1 - Spazio di indirizzamento

È l'insieme di tutti gli indirizzi, **solo** al suo interno questi hanno senso.

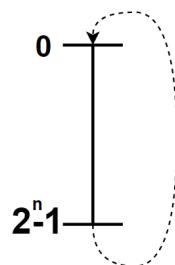
□

Nota:

- Esso è deciso dal BUS.
- In tutte le istruzioni (tranne IN e OUT) ci si riferisce allo spazio di memoria (altrimenti allo spazio I/O).
- Sul BUS ci sono fili per indirizzi, dati, di controllo ecc.

2.1.1 Indirizzi

In particolare i fili di indirizzo sono n , possiamo quindi avere 2^n indirizzi possibili (che vanno da 0 a $2^n - 1$).



Gli indirizzi sul calcolatore hanno rappresentazione binaria, tuttavia è più comodo (per i programmati) rappresentarli in base 8 o 16; essendo queste potenze di 2 si ha come vantaggio una immediata conversione, per la base 8 infatti basta raggruppare 3 bit alla volta ($8 = 2^3$) mentre per la base 16 4 bit alla volta ($16 = 2^4$).

Nota: Altri vantaggi sono:

- $2^a = 10\dots0$ con a zeri
- $2^a - 1 = 1\dots1$ con a uno

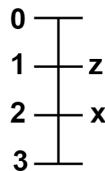
2.1.2 Offset

L'offset tra gli indirizzi è dato da: $|y - x|_{2^n}$, è il numero di indirizzi che devo saltare per andare da un indirizzo ad un altro.



Figura 2.1: Nel caso di questa figura abbiamo: da x a y 3 di offset (normalmente si conta 3 e non 4 ma non è una regola universale), da x a z invece si ha -2

Esempio 5: Consideriamo:



Se voglio calcolare l'offset da x a z ho due modi:

- indietreggio di 1 (-1)
- avanzo di 3 (+3)

Solo se l'offset è rappresentato sullo stesso numero di bit degli indirizzi è corretto sia se interpretato come negativo (in complemento alla radice) che come positivo, infatti in complemento a 2 si ha: $|-1|_{2^2} = |3|_{2^2} = 11$ \square

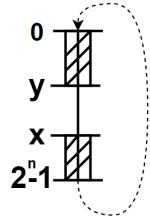
2.1.3 Intervalli

Per rappresentarli vanno bene tutti i modi (intervalli aperti, chiusi, aperti solo in un estremo ecc.) ma il modo più coerente con la definizione di offset è:

$$[x, y) = \{i \mid x \leq i < y\}$$

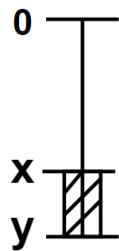
Nota: All'interno di questo intervallo ci sono $y - x$ indirizzi.

Fino ad ora abbiamo parlato di numeri naturali, nel calcolatore però abbiamo $|\mathbb{N}|_{2^n}$.



Caso particolare

Caso particolare (che ci interessa) in cui si verifica la sopra illustrata situazione:



Quello che si verifica è che per far rientrare y nell'intervallo, seguendo il metodo prima designato, quello che io devo scrivere deve comprendere come estremo destro l'indirizzo successivo; ci troviamo però a fondo scala. Fortunatamente (per $x \neq 0$) non si hanno problemi nello scrivere l'intervallo come:

$$[x, 0)$$

2.1.4 Allineamento e scomposizione

Definizione 2 - Allineamento

Un indirizzo è allineato a 2^k se è multiplo di 2^k . Questa condizione si verifica se l'indirizzo finisce con k zeri. \square

Definizione 3 - Confini

Tutti gli indirizzi allineati a 2^b sono detti confini di 2^b ; essi si trovano a partire da 0 procedendo di offset successivi di 2^b . \square

Nota:

- 0 è allineato a qualunque cosa (eccezione).
- Un intervallo è allineato a 2^k se la base ha k zeri.
- Un intervallo è allineato alla dimensione dello stesso:

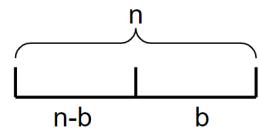
$$y - k = 2^k \rightarrow x \text{ allineato a } 2^k$$

Ogni spazio di indirizzamento è fisso a $b \rightarrow$ ognuno allineato a 2^b .

Definizione 4 - Regione naturale

Ognuno di questi intervalli, delimitati da dei confini, lo chiamiamo regione naturale. \square

Ogni regione naturale è così composta:



Ogni regione naturale avrà un univoco $n - b$ (ovvero il numero di bit più significativi dell'indirizzo di quella regione) detto *numero della regione*, in ogni b invece ci sarà l'offset della regione.

Capitolo 3

La memoria centrale

3.1 (Continuo Lezione 4)

Per la RAM ogni modulo (individuabile all'interno di essa tramite una maschera) è allineato naturalmente, quindi l'offset può essere usato per analizzare un sottocomponente della RAM.

Definizione 5 - RAM

Sottosistema che risponde alle letture e alle scritture del processore. □

Esempio 6: Date le due istruzioni:

```
1 MOV %AL, (%RBX)  
2 MOV %RAX, (%RBX)
```

la RAM deve essere in grado di farle entrambe in modo efficiente (vorrei che facesse sempre una sola operazione e non più di una per quelle che, ad esempio, riguardano più byte). Si può ottenere questo risultato **solo se** ho allineamento. □

3.1.1 Come viene usata la memoria

Storicamente ci sono due metodi per posizionare i dati in memoria:

- **Big endian:** memorizzare in memoria in modo che il byte meno significativo si trovi nell'indirizzo più grande, partendo quindi dal più significativo (non ci interessa, è usato nei pacchetti web).
- **Little endian:** il byte meno significativo si trova nell'indirizzo più piccolo (è quello utilizzato dal processore). **Si utilizza questa.**

Questo comportamento si può dunque osservare sia nei registri (Figura 3.1) che nella RAM (Figura 3.2).

MSB								RAX
-----	--	--	--	--	--	--	--	-----

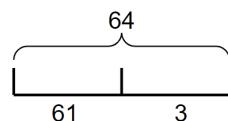
Tabella 3.1:

+7	+6	+5	+4	+3	+2	+1	+0	
88	77	66	55	44	33	22	11	x
								x+8

Tabella 3.2: i numeri sopra le celle di memoria indicano gli offset della riga

3.1.2 Esempio di come avviene una richiesta di operazione di lettura

Per permettere l'accesso a più byte (necessità vista nell'esempio precedente con le due MOV), il processore presenta solo il numero di regione (61 bit, ovvero $n - 3$ data l'architettura a 64 bit dei processori intel/AMD che studiamo), ad esempio:



Al posto dei fili collegati ai 3 bit meno significativi, in cui sarebbe potuto esserci l'offset del primo byte, sono previsti 8 fili di *byte-enable* che permettono di specificare, più comodamente, più casi di un semplice offset e di specificare quali byte devono essere letti/scritti all'interno della riga selezionata. Con queste due informazioni (numero di regione e *byte-enable-wires*) riesco quindi a costruire facilmente una RAM date le specifiche volute.

3.1.3 Montaggio della RAM

Quando compro un chip di RAM esso avrà i classici ingressi e uscite (ingresso dati, /mr, /mw, uscita dati, select), una dimensione (prendiamo come esempio 2^k) e sarà montata su un certo intervallo di indirizzi.

Per ottenere una RAM a 8 byte monto in parallelo 8 chip da 8 bit ciascuno, usando ognuno di questi moduli come elemento di una delle colonne che formano la memoria, la quale sarà dunque composta da righe (di 8 moduli da 8 bit in parallelo) montate in serie.

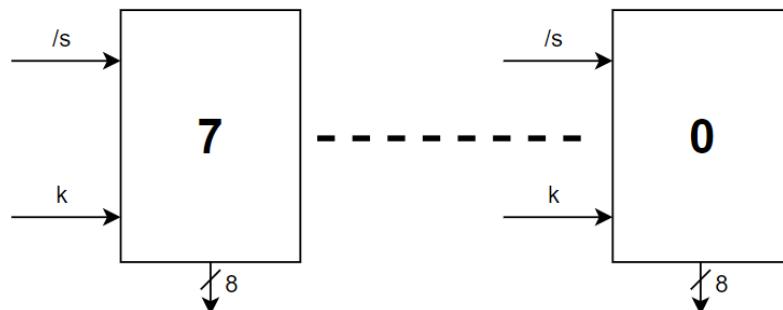


Figura 3.1: Esempio di riga di RAM a 8 byte

La prima cosa da realizzare per la RAM è la **maschera**: devo guardare se sono sugli indirizzi che gli ho riservato montandola. Per fare ciò, se ho n fili di indirizzo, devo prenderne

$$n - (k + 3) \rightarrow n - k - 3$$

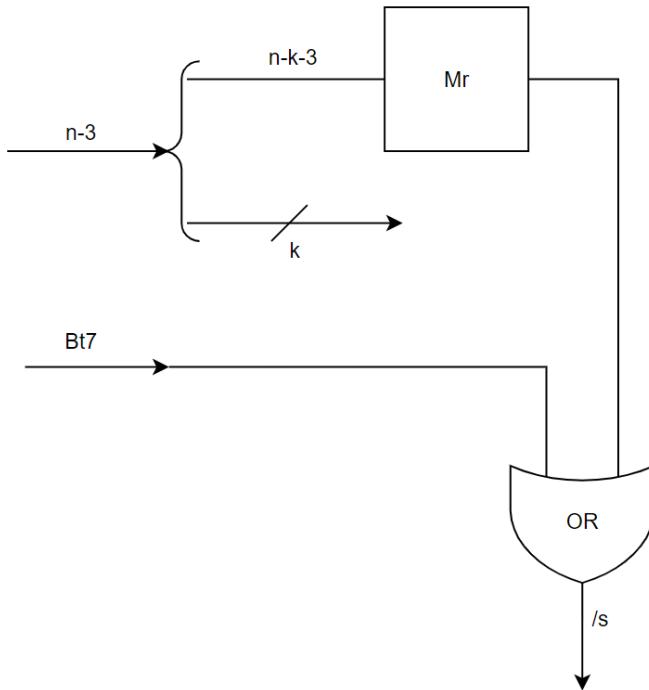


Figura 3.2: La maschera nell'esempio produce il segnale di select per il modulo 7 (byte 7)

Nota : Caso particolare: *Se un dato sta in parte su una riga e in parte su un'un'altra con questo circuito non posso fare la lettura richiesta con una sola operazione; questo circuito infatti sfrutta il fatto di avere i byte-enable e compie quindi 2 operazioni (le due letture sono accompagnate da uno shift per posizionare nella giusta cifra significativa il byte letto).*

3.1.4 Allineamento dei dati in C++

Tutti i tipi (short, int ecc.) sono allineati naturalmente per facilitare i conti: questo significa che stanno a indirizzi pari su ogni riga (ovvero offset +2, +4, +6).

	size-of	align-of
char	1	1
short	2	2
int	4	4
long	8	8

Il problema si ha quando si tratta di array e strutture.

Array

Se si ha un array

tipo a[dim]

allora si ha:

- $align-of(a)$ dato da: $align-of(\text{tipo})$ nota¹
- $size-of(a)$ dato da: $dim \times size-of(\text{tipo})$

Struct

In questo caso si ha il problema di dover garantire che ogni elemento sia allineato e analogamente ogni istanza deve essere allineata alle altre.

- l' $align-of$ è dato dal più grande tra quelli degli elementi della struttura.
- il $size-of$ va invece studiato disegnando gli elementi in memoria (si ha infatti la possibilità di presenza di spazi vuoti, fenomeno detto *internal padding*, che influiscono sulla dimensione finale della struttura)

Esempio 7: data:

```
1 struct a {  
2     char c;  
3     int i;  
4     long L;  
5 }
```

la situazione in memoria sarà:

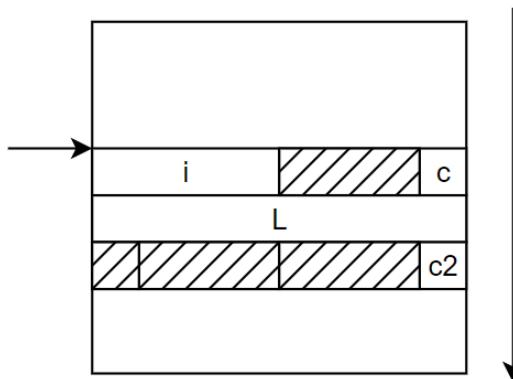


Figura 3.3: In figura si può osservare come, se nella struttura viene aggiunto un char c2, tutto il resto di quella riga di memoria rimane inutilizzato

□

¹Questo è dato dal fatto che l'ABI garantisce che per ogni tipo il suo sizeof sia multiplo del suo alignof.

Capitolo 4

Memoria Cache

4.1 Lezione 7 11/03

4.1.1 CACHE

Partiamo dall'idea di base: il processore è molto più veloce della memoria, quindi possiamo anche avere un processore velocissimo ma che dovrà comunque accedere in memoria per ricavare le istruzioni, sarà dunque rallentato da essa. Sarebbe comodo avere una memoria economica e veloce, tuttavia per ora disponiamo della RAM, ovvero una memoria economica (è infatti grande) ma lenta. Per arrivare al concetto di cache ci basiamo su **due principi** che seguono quasi **ogni programma**:

1. Località spaziale
2. Località temporale

Il principio 1 afferma che se un programma accede ad un certo indirizzo è molto probabile che in breve tempo acceda anche al successivo (prendere come esempio l'accesso ad un array). Il 2 afferma invece che, se un programma accede ad un certo indirizzo, è molto probabile che debba accedervi di nuovo in breve tempo (ad esempio se abbiamo un ciclo). Quindi la conclusione è che, anche se un programma può avere complessivamente bisogno di molta memoria, se lo si osserva in un breve periodo di tempo, vediamo che si concentra su una parte più piccola. L'obiettivo è quindi quello di scoprire quale è questa parte e di copiarla in una memoria che, seppur costosa, sia veloce.

Quello che si fa è di aggiungere nell'architettura tra CPU e BUS la **cache**. Questa è composta da un **controllore cache** e dalla **memoria cache** vera e propria (che è costosa e piccola ma veloce). Il **ruolo del controllore cache** è quello di intercettare quasi tutte le operazioni di lettura e scrittura iniziate dal processore verso la memoria principale, controllare se i dati cercati si trovano in cache e, in caso affermativo, restituircieli, altrimenti accedere lui stesso in RAM, restituire i dati e tenerne una copia in cache. In questo modo si riducono gli accessi in memoria e si velocizza il lavoro complessivo.

Nota: All'inizio ovviamente la cache è vuota, man mano si riempie.

L'unità di trasferimento è detta **cacheline** ed è solitamente di 64 byte (per via della memoria delle etichette, aspetto analizzato in seguito). Quando la cache si riempie alcune cacheline dovranno essere rimpiazzate per far spazio ad altre: per questo vengono utilizzati algoritmi euristici che non sempre portano alla soluzione ottima, ma rimangono la soluzione migliore. Infatti la soluzione ottima sarebbe tenere sempre cacheline che verranno accedute

in seguito, il processore però vede un'istruzione alla volta e non può prevedere ciò che avverrà, mentre il controllore cache vede solamente le operazioni iniziate dal processore.

Nota: Molto importante è notare che tutto ciò si svolge completamente in hardware ed è trasparente al software, infatti i programmatore possono scrivere i loro programmi ignorando la presenza della cache, con l'unica differenza che potrebbero andare più veloci. Inoltre (e questa è una caratteristica abbastanza generale per tutti i moduli che aggiungiamo) la presenza della cache non influenza il comportamento degli altri moduli dell'architettura (ad esempio non influenza il lavoro del processore).

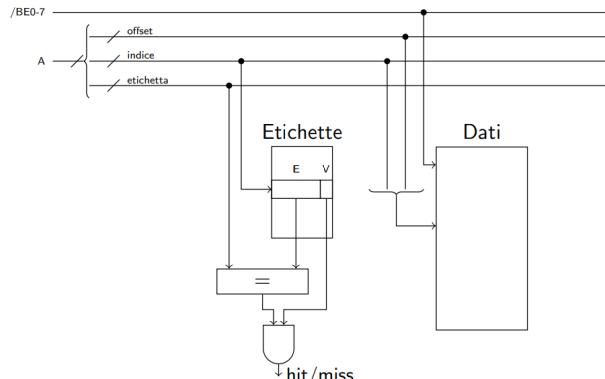
Nota: Altra cosa da notare è che il controllore cache lascia passare tutte le operazioni di I/O, in quanto queste hanno effetti collaterali che non devono essere cancellati. Inoltre si deve disabilitare la cache quando bisogna accedere in lettura o scrittura ad indirizzi della memoria principale in cui sono mappati registri relativi ad interfacce di I/O. Vedremo con la paginazione che c'è la possibilità di settare/resettare dei flag per gestire la cache in questi casi.

Vediamo ora 2 tipi di cache:

- Cache ad indirizzamento diretto
- Cache associative ad insiemi

4.1.2 Cache ad indirizzamento diretto

Il primo tipo che vediamo è la cache ad indirizzamento diretto.



Dal punto di vista funzionale la cache è identica ad una memoria RAM, solo più piccola, sulla quale si possono effettuare operazioni di lettura e scrittura ad indirizzi (detti **indirizzi di cache**). In particolare se prendiamo una cache con cacheline a 64 byte l'indirizzo di riga generato dal processore verrà scomposto in questo modo:

- i 3 bit meno significativi rappresentano l'offset
- il resto il **numero di cacheline**

Quest'ultimo basta per determinare se un dato è presente in cache. Viene inoltre scomposto a sua volta in *indice* (composto da a bit se la cache è grande 2^a cacheline, va a comporre l'**indirizzo di cache** di tale chacheline) e *etichetta*. Tutte le cacheline che distano 2^a hanno lo stesso indice e si differenziano dunque solo per l'etichetta. Viene inoltre utilizzata una

memoria detta **memoria delle etichette** dove in ogni riga è presente l'etichetta relativa ad ogni indice ed un bit di validità posto ad 1 se l'etichetta è valida.

Quando si esegue una scansione all'interno della cache si va a prelevare, nella memoria delle etichette, l'etichetta relativa all'indice, si confronta poi con l'etichetta generata dal numero di cacheline e si fa un AND con il bit di validità, generando così il segnale di *hit/miss*. In caso di *hit* si usano l'offset e l'indice per ricavare l'indirizzo da cui prelevare il dato che ci serve nella memoria dati.

Politiche lettura

Per quanto riguarda la lettura, se si genera un *hit* si preleva il dato e si restituisce al processore. Se si è generato un *miss* il controllore cache legge la cacheline dalla memoria, la copia in cache e poi si procede come in una *hit*.

Nota: *Il Problema principale della cache ad indirizzamento diretto è che 2 cacheline che hanno lo stesso indice non possono convivere contemporaneamente in cache, anche se il resto della cache è vuota. Si verifica infatti che più cacheline possono essere caratterizzate dallo stesso indice ma nella memoria delle etichette se ne può salvare solo una alla volta, questo per via del tipo di "hashing" che si effettua per riservare loro un indirizzo durante il salvataggio in memoria delle etichette.*

Politiche scrittura

Per quanto riguarda la scrittura in caso di *miss* il controllore completa la scrittura in memoria senza caricare la cacheline (*write no allocate*) o carica la cacheline in cache e prosegue come in una *write hit* (*write allocate*).

Nel caso di *write hit* il controllore aggiorna solo la cache (*write back*) o aggiorna sia la cache che la memoria (*write through*).

Nota: *Le cache moderne sono write back e write allocate.*

Se si usa la politica *write back* solitamente si aggiunge un bit alle cacheline detto *bit dirty*, il quale viene settato se la cacheline viene sporcata. Prima di venire rimpiazzata, la cacheline, se ha questo bit settato viene ricopiata in memoria principale.

Avevamo anticipato il fatto che le cacheline sono grandi 64 byte nonostante ne servano molto meno, questo tuttavia è un buon compromesso perché più le cacheline sono grandi più piccola è la memoria delle etichette e questo è un bene perché è costosa; allo stesso tempo però non devono essere troppo grandi altrimenti diventa complicato effettuare le operazioni di lettura e scrittura.

4.1.3 Cache associative ad insiemi

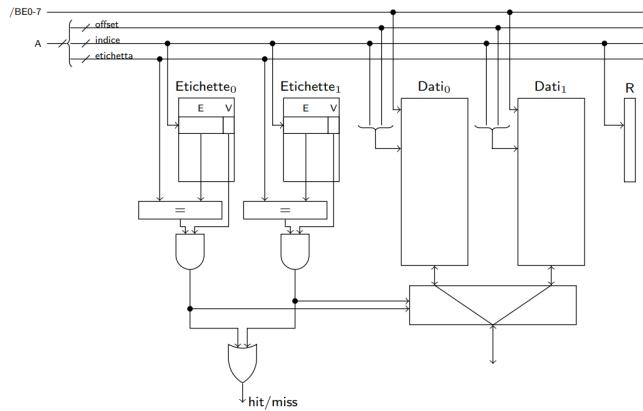
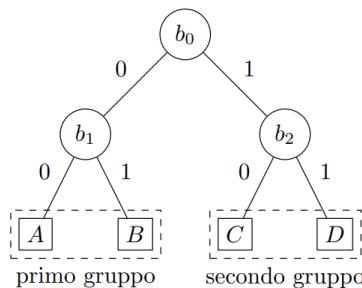


Figura 4.1: Cache associativa a insiemi (a 2 vie).

Sono più costose di quelle ad indirizzamento diretto ma alleviano il problema dei conflitti. Possono essere a più vie, se sono ad n vie posso memorizzare n cacheline allo stesso indice. Sono composte da n cache ad indirizzamento diretto messe in parallelo. Ora ogni indice individua un insieme di possibili locazioni per la cacheline. Infatti, prendendo ad esempio una cache a 2 vie, si genera una *hit* se una delle due cache produce una *hit*: il segnale di *hit/miss* è l'*or* dei segnali di *hit/miss* generati dalle varie cache ad indirizzamento diretto; l'*or* riceve quindi le uscite delle 2 porte AND.

In caso di miss bisogna decidere quale delle cacheline rimpiazzare. Per fare ciò si utilizza un algoritmo pseudo-LRU: l'idea è di rimpiazzare la cacheline che non viene acceduta da più tempo. Per implementare questo algoritmo si utilizza anche un'altra piccola memoria detta R che contiene le informazioni per implementare lo pseudo-LRU e viene aggiornata ad ogni accesso. Ad esempio se si utilizza una cache a 4 vie, R è organizzata in 3 bit ed a ogni accesso basta aggiornare 2 di questi 3 bit. Il difetto principale dell'algoritmo LRU è che non porta alla soluzione ottima e in alcuni casi porta alla peggiore.



La cache è un elemento dell'architettura fondamentale, senza di essa il computer sarebbe praticamente inutilizzabile, questi concetti verranno infatti ripresi con la paginazione e con il meccanismo del DMA.

Nota : Dove si trova questo argomento all'esame: *Le politiche della cache si possono trovare anche in esercizi come quelli sull'hard disk e all'orale viene chiesta molto, specialmente a chi non ha un voto alto.*

Capitolo 5

Periferiche

5.1 Lezione 8 12/03

5.1.1 Tastiera

La prima periferica che vediamo è la tastiera, il cui scopo è quello di individuare i tasti che vengono premuti e in seguito rilasciati. Ogni tasto possiede un *makecode* e un *breakcode*, su 8 bit, rispettivamente per quando viene premuto e rilasciato. Questi 2 codici si differenziano solo per il MSB:

- 0 per il *make*
- 1 per il *break*

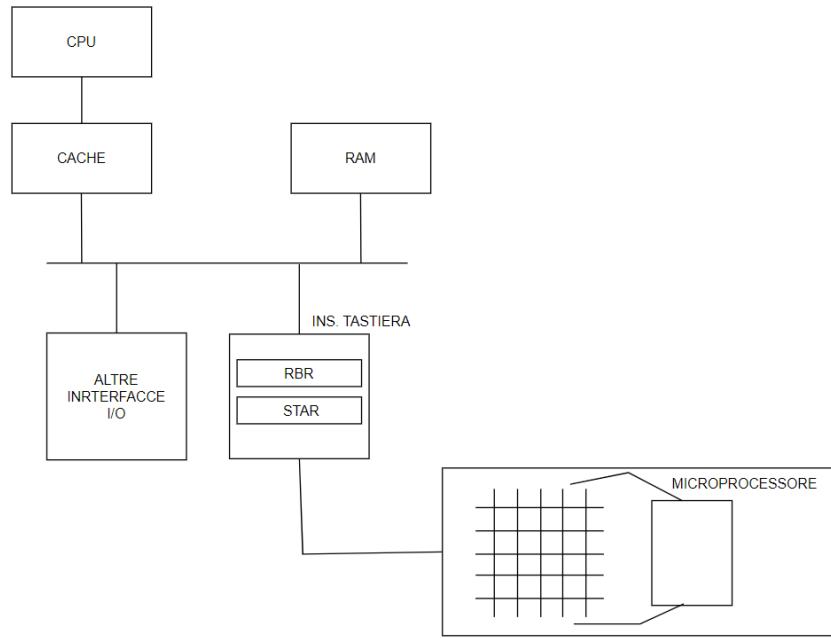
La tastiera è composta da una matrice di collegamenti elettrici distribuiti su 3 strati: il primo ha i collegamenti verticali, il terzo ha quelli orizzontali e il secondo serve a tenerli separati. Quest'ultimo presenta però dei buchi in corrispondenza dei tasti: in tal modo, quando un tasto viene premuto, si mette un comunicazione lo strato 1 e 3 e si genera un impulso elettrico.

La tastiera contiene un microcontrollore, un computer vero e proprio con propri Ram, I/O e processore, il quale ha lo scopo di esaminare continuamente la matrice e, quando viene premuto un tasto, inviare il suo *makecode* al PC. A livello pratico, infatti, la tastiera si limita a inviare i codici e a gestire il *typematic* di quasi ogni tasto (ovvero quando un tasto viene premuto per più tempo), per il resto fa tutto il software. Per sapere quando un tasto è stato rilasciato il microcontrollore ricorda lo stato di ogni tasto nella propria RAM, così quando un tasto viene premuto e poi rilasciato se ne accorge.

Per quanto riguarda l'interfaccia, per il programmatore sono disponibili 4 registri, 2 di sola lettura e 2 di sola scrittura. I registri per la lettura sono RBR e STS. RBR serve per leggere il codice dell'ultimo tasto premuto, STS funziona da flag busy: va letto RBR solo quando STS è a 1. I registri TBR e CMR funzionano allo stesso modo ma per la scrittura.

5.2 Lezione 9 14/03

Continuiamo a vedere le periferiche, aggiornando la struttura del calcolatore:

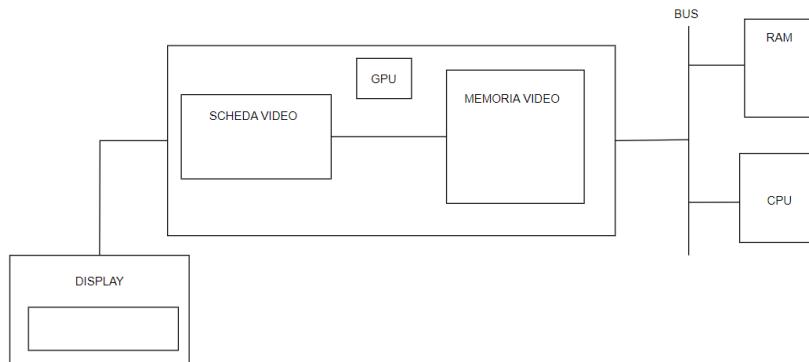


Finora abbiamo visto come:

- Il processore esegue continuamente istruzioni (non può fare altrimenti).
- Il microprocessore scansiona continuamente la macchina (rileva quindi quando un tasto viene premuto).
- La cache lavora con il processore.
- Il blocco dell'inserimento tastiera viene letto quando ci sono IN.
- Il cursore viene fatto lampeggiare in hardware dalla scheda video, altrimenti dovrebbe esserci stato del software messo da noi (ricordare lezione 1) che "dice" questa cosa.

5.2.1 Scheda video

Il suo ruolo è quello di inviare informazioni digitali al monitor, per ogni pixel infatti deve indicare che colore esso deve assumere. La sua struttura collegata al calcolatore è:



5.2.2 Memoria video

È una RAM particolare: può essere letta solo da una parte alla volta (?).

Si trova anch'essa nello spazio di memoria (come la RAM) il che risulta importante per la cache: non vogliamo che una scrittura rimanga in cacheline per chissà quanto tempo (l'informazione video la vogliamo vedere subito sullo schermo).

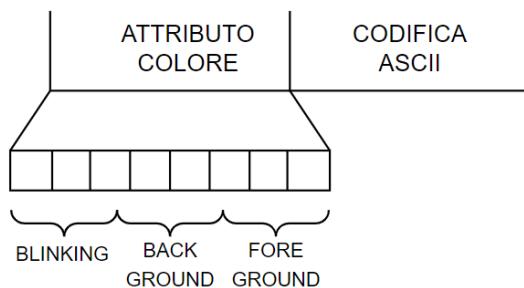
Nota: La scheda video (quella vecchia della IBM) lavora in 2 modalità che interpretano come bisogna leggere quello che sta in memoria video:

1. testo

2. grafica

Alla memoria video possono accedere il software, solitamente in scrittura, e il controllore video, il quale invece si occupa di interpretare il contenuto della memoria e di inviare i segnali al monitor. Software e monitor possono comunicare, è in questo modo infatti che possiamo selezionare la modalità che ci interessa.

Nella memoria video, che è organizzata in righe e colonne (con questa organizzazione dove una cosa sta è dove viene visualizzata), ogni cella è composta da 2 byte:



Nota: Il controllore video espone sul bus solo due registri

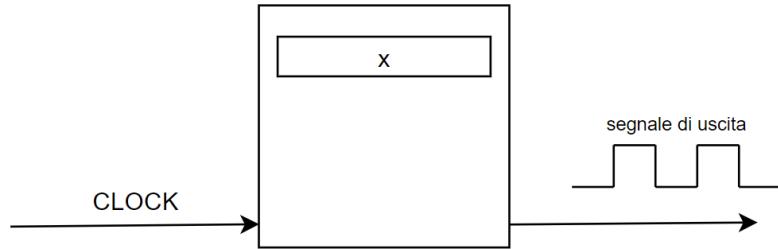
5.2.3 Timer

Si tratta di una periferica che in generale viene chiamata "conta eventi". Sono periferiche "di conteggio", in quanto hanno al loro interno un contatore, che può essere inizializzato via software, e che viene decrementato ogni verificarsi di un evento (ad esempio ogni clock). Al termine del conteggio si genera un determinato output e il contatore:

- resta a 0 se è a ciclo unico
- si ricarica e riparte se è a ciclo continuo.



Il conta eventi diventa timer quando l'ingresso è un clock:



Nota: In uscita dal timer (dell'IBM) si ha un'onda quadra (utile per comandare il semplice audio del pc IBM).

5.3 Lezione 10 15/03

Continuando a parlare del timer:

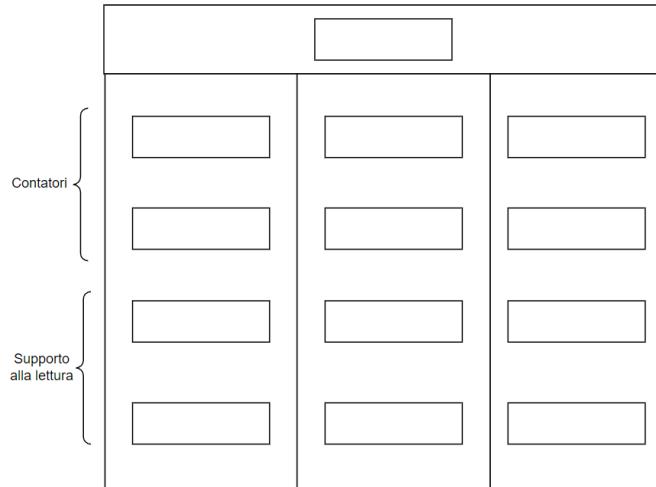


Figura 5.1: Chip di interfaccia del timer

Guardando alla figura:

- La prima colonna (la 0) è collegata al meccanismo delle interruzioni, ed è la cosa importante del timer. Solitamente viene inviata una richiesta di interruzione quando il contatore arriva a 0.
- La 1 è collegata al refresh della memoria dinamica.
- La 2 va in AND con 1 bit del registro SPR all'indirizzo di I/O 61.

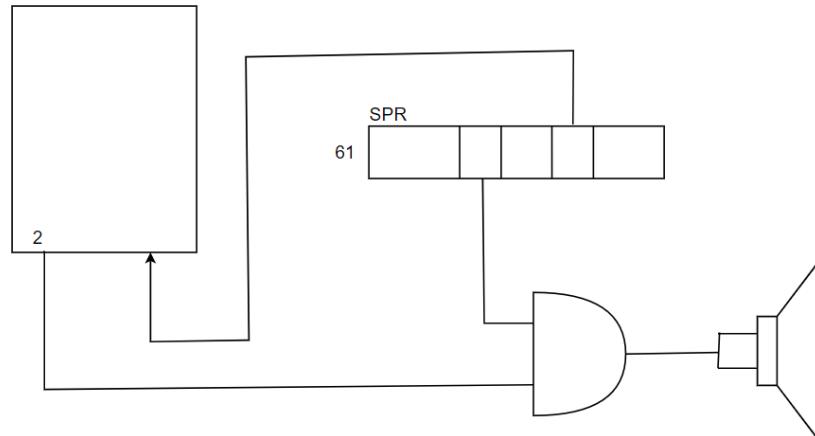
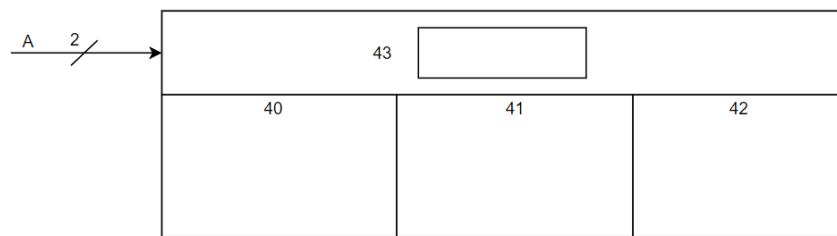


Figura 5.2: L'uscita in AND può essere usata per collegare quella che è una semplice uscita audio (cicalino). Il bit 0 del registro SPR può essere usato per mettere in pausa il conteggio.

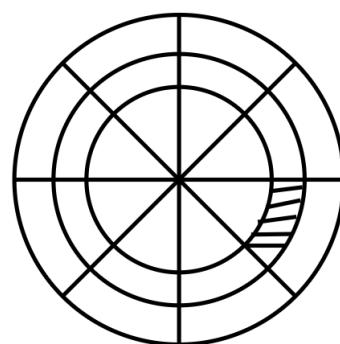
Questa interfaccia ha solo 2 piedini (si tratta di una interfaccia vecchia):



(aggiungere commento sull'immagine)

5.3.1 Hard Disk

Inizialmente era costruito secondo lo standard ATA (evolutosi poi in ATAPI, SATA, PATA). Questa tecnologia è basata su dischi di materiale ferromagnetico in cui l'informazione è organizzata in tracce (le varie corone circolari) e settori (le varie parti di corona circolare delimitate da confini).



Nell'immagine sopra la parte tratteggiata è detta **blocco** e tradizionalmente è di 512 byte. I blocchi sono troppo grandi per essere elaborati dal processore, vanno quindi trasferiti in RAM mediante lo *swap-in*, questo però richiede l'utilizzo di una interfaccia.

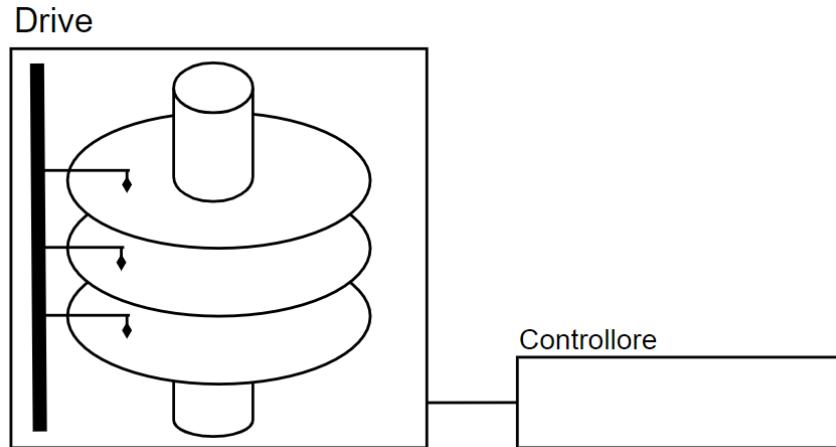


Figura 5.3: I drive sono stati al più formati da 3 dischi

Se voglio leggere un blocco devo identificare:

1. Faccia (quindi spostare la testina sul disco desiderato)
2. Traccia
3. Settore

Il tempo di accesso a un blocco sarà quindi determinato da:

$$\text{seek} + \text{latency} + \text{read/write}$$

I primi due sono tempi meccanici:

- *seek* dipende da dove è la testina, mediamente quindi possiamo misurare un tempo nell'ordine dei *ms*.
- *latency* dipende da quando il settore passa sotto la testina (i dischi infatti ruotano costantemente). Questo tempo è sempre dell'ordine dei *ms* in quanto la velocità massima raggiungibile è di circa 15/20 mila rpm.
- Il tempo di *read/write* dipende invece dal tempo che impiega il blocco a passare sotto la testina, è quindi molto breve, dell'ordine di *ns*.

Nota: Definiamo "cilindro" le varie tracce corrispondenti poste su dischi diversi in colonna tra loro.

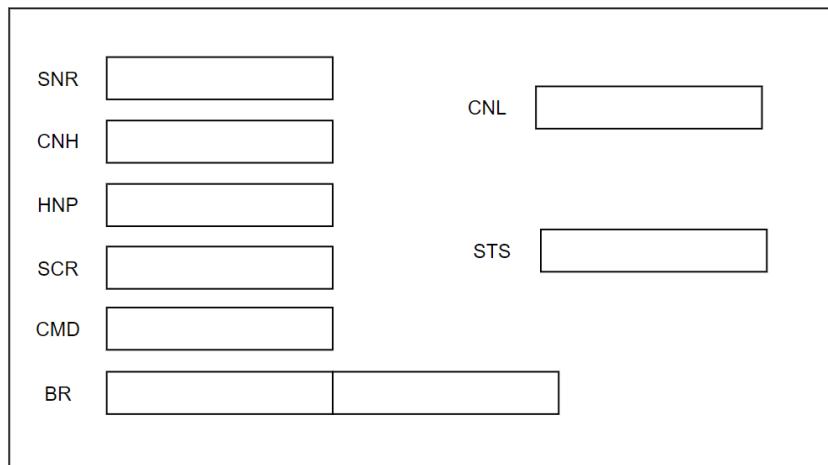
Nota: Gli hard disk moderni sono organizzati in zone in modo da avere settori tutti della solita dimensione; questo porta ad avere molti più blocchi nelle tracce esterne e quindi, data una velocità angolare costante, una maggiore velocità di lettura/scrittura.

I dati che vengono letti dai settori vengono messi dal controllore in una memoria interna per poter essere utilizzati.

Definizione 6 - Formattare

In origine significava creare la struttura interna di tracce e settori con tanto di indentificatori per i vari blocchi. Ora questo non è più possibile farlo con gli hard disk commerciali e quindi il significato del termine è virato verso l'indicare il ripristino del contenuto dei dati dell'hard disk. □

Per permettere al programmatore di accedere all'hard disk si ha una interfaccia:



Come al solito, per garantire retrocompatibilità, si è mantenuta la struttura mostrata nell'immagine sopra. Questa ha però perso in parte di significato dal momento che sono comparse le aree che con esse hanno portato all'utilizzo dell'LBA (logical block address).

Capitolo 6

Interruzioni

6.1 Lezione 11 18/03

```
1 do {  
2     s = input(ist);  
3 } while ( !(s & 1) );
```

In questo ciclo *do while* di esempio cicliamo finché non leggiamo nel registro...

In questo modo, se il programma sta scrivendo sull'hard disk, l'utente per poter scrivere dovrebbe aspettare la fine dell'operazione. Vorremmo però che ciò **non** accadesse, introduciamo quindi la **programmazione ad eventi**.

6.1.1 Programmazione a eventi

Il programmatore può associare una propria funzione a un evento, quindi la macchina, all'accadere dell'evento, in automatico smette di fare cosa stava facendo e salta alla funzione associata.

L'idea è che il processore inizi eseguendo in sequenza le istruzioni del programma principale *p* e, mentre lo esegue, controlla se si è verificato uno degli eventi previsti. In caso affermativo salterà automaticamente alla routine associata a quell'evento, interrompendo il programma principale *p*. L'interruzione è comunque momentanea e una volta finita il processore riprende a eseguire istruzioni del programma principale dal punto in cui era stato interrotto.

Nota: *Guardiamo prima il caso di singola sorgente/singolo evento.*

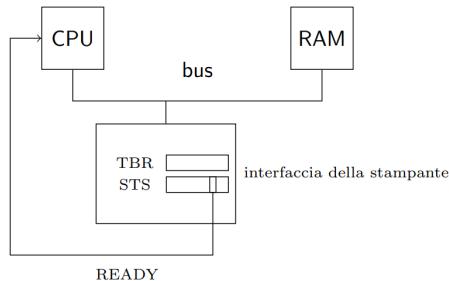
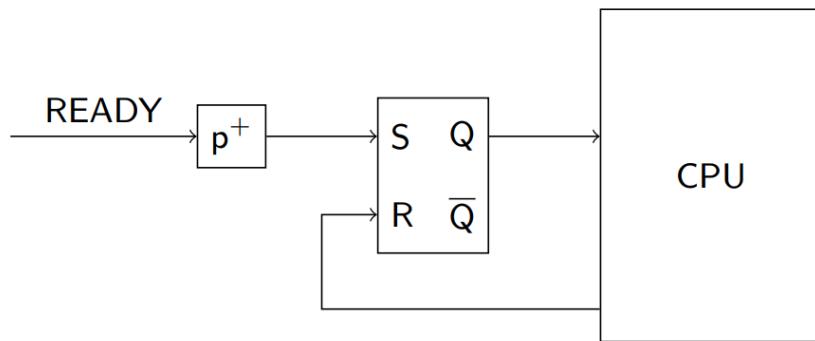


Figura 6.1: Quando STS passa a 1 la CPU smette di eseguire il programma principale e passa alla "stampa" (funzione di esempio chiamata dall'evento).

Modifichiamo il microprogramma della CPU in modo che dopo **ogni** istruzione controlli il flag in ingresso (si tratta di un salto fittizio). Operando in questo modo si risolve il problema di attuare il controllo troppo raramente (ad esempio ponendolo prima e dopo un ciclo molto lungo).

Dove salviamo l'indirizzo di ritorno di questo programma lanciato dall'evento? **in pila** (il programma termina infatti con una variante dell'istruzione RET: la IRETQ). La IRETQ permette di estrarre dalla pila ciò che ci avevamo salvato, ovvero il registro RIP e il registro dei flags. Inoltre se consideriamo il caso singola sorgente, in cui una volta che viene accettata la richiesta di interruzione il processore disabilita il flag IF, la IRETQ riabilita quest'ultimo.

Dal punto di vista hardware questo modello è troppo semplicistico: rischiamo di leggere due volte il solito evento. Infatti quando la CPU vede il bit *ready* a 1 salta alla routine legata all'interruzione e qualora la prima istruzione della routine non sia esattamente quella che scrive nel registro TBR della stampante, e dunque non risponde alla richiesta, la CPU dopo averla eseguita troverà ancora *ready* ad 1 e, pensando che ci sia una nuova richiesta, risalterà all'inizio della routine eseguendo l'istruzione che avevamo appena eseguito e generando un loop. Infatti per come abbiamo progettato per ora il meccanismo la CPU controlla a seguito di ogni istruzione lo stato del bit *ready*. Il fatto è che, prendendo questo esempio, la stampante quando è pronta a stampare un carattere vuole sapere quale valore deve stampare, perciò invia la richiesta di interruzione alla CPU settando il bit *ready*. La CPU accetta la richiesta e fa partire la routine di interruzione relativa alla stampante, la quale ha lo scopo di scrivere in TBR il valore da stampare. Proprio la scrittura in TBR funge da risposta alla richiesta e solo a seguito di ciò viene resettato *ready*. Si risolve questo problema con l'aggiunta di un latch-SR con in ingresso un impulsivo:



Il segnale di ready, quando settato crea un impulso che setterà il latch SR, poi il microprogramma della CPU provvederà a resettare il latch dopo aver visto la richiesta, in modo che il latch sia di nuovo attivo solo se arriva una nuova richiesta di interruzione.

Nota: Il segnale di ingresso alla CPU è detto **segnale di interruzione**, il meccanismo è detto **di interruzione** e quella che va in funzione si chiama **routine di interruzione**.

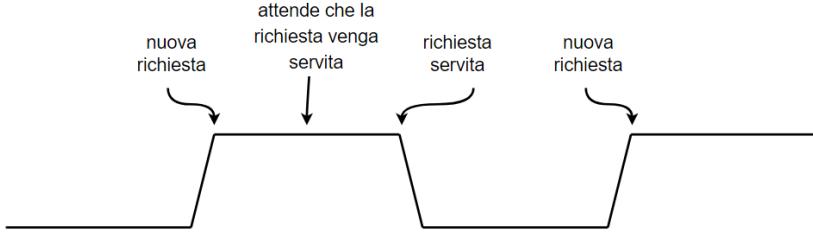


Figura 6.2: Segnale di richiesta

Nota: Non tutte le interfacce si comportano come mostrato in figura: ad esempio il timer va avanti senza attendere risposta in quanto deve "far passare il tempo" a prescindere. Nel timer si ha riconoscimento sul fronte, se si avesse sul livello ci sarebbe il rischio di leggere sempre la stessa richiesta. Nella tastiera si ha invece riconoscimento sul livello.

È utile che il programmatore possa voler **non essere interrotto in certi momenti** (ad esempio durante la routine di interruzione stessa), è quindi presente un flag che disabilita l'interruzione: IF (Interruption Flag). Questo flag può essere rispettivamente resettato e settato via software mediante le istruzioni **cli** e **sti**.

Nota: Nell'intel molti aspetti della gestione delle interruzioni sono gestiti in hardware.

Il sottoprogramma (chiamato dall'interruzione) **deve**, come già detto, essere terminato da IRETQ. Questa istruzione infatti ripristina RIP e i flag salvati al momento dell'arrivo dell'interruzione, in particolare ripristina il valore di IF che era stato modificato in modo di non ricevere altre interruzioni. Abbiamo visto a proposito un esempio in cui al posto di IRETQ viene utilizzata la RET e si nota come dopo la prima richiesta di interruzione non ne vengono accettate altre, questo perché la RET a differenza della IRETQ non ripristina i flag che erano stati salvati in pila.

Nota: Per la CPU il meccanismo delle interruzioni si esaurisce nel momento in cui fa il salto: dopo (ad esempio al momento della RET) non si ricorda minimamente di aver visto una interruzione, la CPU infatti, come sempre, esegue il flusso e basta.

Gestione di più richieste

Nella realtà si possono avere più sorgenti di richieste di interruzione, tuttavia se ne potrà gestire solo una per volta, per questo la cpu:

- ha **un unico flusso di controllo** e può eseguire le istruzioni o del programma principale o della routine.
- ha **un solo piedino sul quale può ricevere le richieste**. Serve dunque un oggetto che raccolga le varie richieste che arrivano dalle sorgenti, le ordini e le invii alla CPU una alla volta.

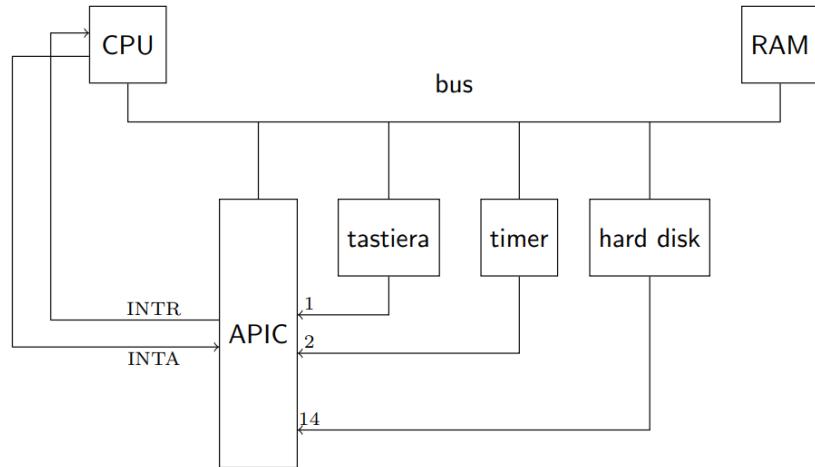


Figura 6.3: Esempio di controllore delle richieste di interruzione di più interfacce.

Come anticipato nella descrizione dell'immagine si avrà un **controllore delle interruzioni**. Si nota inoltre, sempre dall'immagine, che l'intel non si limita a una semplice gestione ma aggiunge dei piedini tra controllore e CPU che servono a far passare la codifica di chi ha fatto la richiesta. I collegamenti INTR e INTA servono a gestire l'handshake, la comunicazione tra processore e controllore APIC (spiegato bene su dispense). Si ha poi un terzo collegamento per passare il tipo della richiesta di interruzione.

Nota: *Il tipo può essere passato anche sul bus.*

Il controllore delle interruzioni usato dall'intel è l'APIC. L'APIC ha 24 piedini di ingresso numerati da 0 a 23, uno per ogni possibile sorgente. Ad ogni sorgente è quindi associato un tipo numerico, il quale viene passato alla CPU insieme alla richiesta. Una volta che la CPU ha ricevuto il tipo lo usa per consultare la **IDT**. Quest'ultima è una tabella allocata in RAM, il cui indirizzo viene scritto nel registro IDTR del processore, che contiene gli indirizzi ai quali saltare per eseguire le routine associate alle richieste di interruzione. In particolare basta scrivere l'indirizzo della routine nell'entrata corrispondente al tipo da associare. Nell'APIC l'associazione tra sorgente e tipo può avvenire via software attraverso i registri che il controllore contiene per ogni piedino. Per quanto riguarda la comunicazione tra CPU e APIC, questa, come già anticipato, avviene tramite handshake mediante i collegamenti INTR e INTA:

1. L'APIC quando vuole mandare una richiesta alla CPU attiva INTR e aspetta che sia attivo INTA.
2. La CPU dopo ogni istruzione se IF=1 controlla se INTR è attivo, in caso affermativo attiva INTA.
3. A questo punto l'APIC invia il tipo alla CPU e abbassa INTR. A questo punto la richiesta è accettata.
4. Infine la CPU vede INTR disattivo, legge il tipo e disattiva INTA.

Nota: *Le interruzioni esterne non sospendono l'esecuzione di un'istruzione, come vedremo solo le eccezioni di tipo fault vengono sollevate durante l'esecuzione di un'istruzione, sospendendola.*

Per quanto riguarda la fine dell'interruzione, **è compito del software far sapere che la routine è terminata** e lo fa scrivendo un valore all'interno del registro EOI dell'APIC.

L'APIC possiede altri 2 importanti registri: IRR e ISR, entrambi a 256 bit, uno per ogni tipo. Il primo tiene conto di tutte le richieste pendenti, ovvero arrivate ma che devono ancora essere inoltrate alla CPU. Il secondo tiene conto delle richieste attualmente in servizio per cui non è ancora stato inviato l'*end of interrupt*.

Vediamo meglio il funzionamento di questi due registri (assumiamo che ci sia il riconoscimento sul fronte):

1. Quando arriva una richiesta sul piedino i , associato al tipo t , viene settato, se non lo è già, il bit relativo a t nel registro IRR.
2. A questo punto, se t in ISR è 0, inizia l'handshake con la CPU e se la richiesta viene accettata si setta il bit t del registro ISR, resettando quello di IRR.
Nota: *Una eventuale nuova richiesta dello stesso tipo può arrivare, in quel caso verrà settato il bit t del registro IRR, creando così una coda di 2 richieste dello stesso tipo (una eventuale terza verrebbe persa).*
3. Quando viene inviato l'EOI l'APIC provvede ad azzerare t in ISR e se in IRR è a 1 riavvia l'handshake.

Passando al discorso del riconoscimento, l'APIC può ricevere le richieste di interruzione e di conseguenza riconoscerle in 2 modi:

1. **Riconoscimento sul fronte:** una nuova richiesta di interruzione viene riconosciuta ogni volta che il segnale su un piedino passa da 0 a 1 (o da 1 a 0 configurabile).
Nota: *Viene utilizzato questo metodo ad esempio dal timer.*
2. **Riconoscimento sul livello:** viene riconosciuta una prima richiesta sul fronte, poi continua a riconoscere una nuova richiesta se il segnale è ancora attivo nel momento in cui l'APIC riceve un EOI.

Gestione annidata delle richieste di interruzione

Per arrivare a considerare il caso reale manca da analizzare il caso di gestione annidata. In generale infatti le richieste possono arrivare da più sorgenti (anche contemporaneamente) e una eventuale routine può essere interrotta da un'altra. Per rendere possibile ciò bisogna:

- che il processore **non** resetti IF quando accetta una richiesta
- introdurre il concetto di **priorità**, in modo tale che una routine possa essere interrotta solo da una richiesta che abbia priorità strettamente maggiore.

Nota: *Il timer ha la priorità più alta.*

L'APIC interpreta il tipo associato ad ogni piedino come la priorità: in particolare i 4 bit più significativi del tipo indicano la classe di priorità. Quando arriva una nuova richiesta, come già spiegato nel paragrafo precedente, questa viene registrata in IRR. A quel punto l'APIC controlla se la sua classe di priorità, chiamiamola p , sia > della classe di priorità della richiesta più a sinistra in ISR; in caso affermativo si inizia l'handshake.

L'APIC si comporta come già detto con i registri IRR e ISR, con l'unica differenza che ora in ISR possono esserci più bit a 1 contemporaneamente e quello ad 1 più a sx sarà quello

relativo alla richiesta che si sta gestendo con priorità maggiore. Quando il software scriverà qualcosa in EOI, si assume che segua la politica LIFO e quindi si resetta il bit ad 1 più a sx in ISR. Ora l'APIC controlla le classi di priorità di ISR e IRR e eventualmente si inizia un nuovo handshake.

Ultima cosa da notare è che per decidere se inoltrare o no una richiesta l'APIC controlla solo le classi di priorità, mentre quando deve inviarla guarda tutti i bit del tipo, compresi i 4 bit meno significativi. Questo perché a parità di classi di priorità si guardano appunto tutti i bit.

Riassunto di "chi fa cosa" nelle interruzioni esterne

- **Periferica:** invia la richiesta di interruzione all'APIC.
- **APIC:** prima inoltra la richiesta alla CPU, poi quando viene accettata gestisce IRR e ISR, e invia il tipo.
- **CPU:** all'inizio esegue istruzioni, se IF = 1 dopo ognuna di esse controlla se INTR è attivo, se non lo è continua normalmente, altrimenti continua l'handshake con l'APIC. Dopo che ha ricevuto il tipo consulta la IDT, salva le cose in pila, eventualmente aggiorna IF, salta all'indirizzo scritto nel gate e una volta che ha chiuso l'handshake continua con il normale flusso di istruzioni, probabilmente quelle della routine.
- **Software:** Invia l'*end of interrupt* scrivendo qualcosa nel registro EOI dell'APIC.

6.2 Lezione 12 19/03

(esempi vari sulle interruzioni)

Capitolo 7

Eccezioni

7.1 Lezione 14 22/03

Note esempi al pc:

- per gestire le interruzione nel debugger usare il comando *c* : il comando *n* non le fa funzionare
- *q* per uscire dal debugger

7.1.1 Eccezioni

Le eccezioni sono situazioni speciali o di errore che sono rilevate dal processore mentre esegue le normali istruzioni e ne interrompono il flusso principale (ad esempio una divisione per 0). Il meccanismo che seguono è in genere molto simile a quello delle interruzioni, in particolar modo nell'intel è esattamente lo stesso: quando il processore solleva una eccezione fa partire una routine associata dal programmatore (sono queste che ci fanno mostrare a video i vari messaggi di errore).

Nota: *Anche le routine legate alle eccezioni devono terminare con IRETQ, invece che con RET, in quanto il processore salva in pila le stesse informazioni che salvava per le interruzioni.*

Nota: *Nel caso delle eccezioni la sorgente dell'interruzione non è esterna (come abbiamo visto fino ad ora), ma è interna al processore stesso. La condizione si verifica mentre il processore sta eseguendo. Il meccanismo invece è analogo: l'unica differenza è che il tipo è implicito (mentre prima veniva passato dall'APIC).*

Nota: *Nell'intel le prime 32 righe dell'IDT sono riservate alle eccezioni (in particolare nella riga 0 si trova l'eccezione per la divisione per 0, alla riga 14 quella legata al page fault).*

Tipi di eccezioni

Le eccezioni hanno 3 tipi:

1. **fault:** salva il RIP dell'istruzione fallita in pila e riprova a eseguirla (quasi tutte sono di questo tipo).

2. **trap**: salva in pila il RIP dell'istruzione successiva a quella che ha sollevato l'interruzione, come fanno anche le interruzioni esterne.
3. **abort**: sono la situazione peggiore in quanto resta solo che spegnere il processore (infatti avviene questo).

Nota: *Un modo per causare un abort è avere un triplo fault.*

Le *fault* non permettono che l'istruzione che le ha generate finisca (ad esempio divisione per zero), mentre le *trap* verranno gestite al termine dell'istruzione che le ha generate. Per le *fault* l'idea è che la routine legata all'eccezione possa risolvere il problema e che quindi, quando si torna ad eseguire l'istruzione che aveva causato l'eccezione, non si abbiano problemi o almeno che ne crei uno diverso. La maggior parte delle eccezioni sono di tipo *fault* e rappresentano situazioni di errore, al contrario le *trap* rappresentano più situazioni particolari come una chiamata di sistema o eventi programmati.

Altra particolarità attuata per gestire correttamente le *fault* è questa: il processore può utilizzare delle copie di registri e apportare le modifiche inizialmente a questi, alla fine di ogni istruzione verrà poi trasferito il contenuto di queste copie ai veri registri del processore. In questa maniera, se si verifica un *fault*, basta ignorare le copie dei registri, dato che i veri registri non sono stati sporcati durante l'istruzione che ha generato l'eccezione, e si può tranquillamente rieseguire un'istruzione che era stata eseguita fino ad un certo punto e, allo stesso tempo, il processore è in grado di ritornare allo stato precedente l'inizio dell'istruzione.

Capitolo 8

Protezione

8.1 Lezione 15 25/03

8.1.1 Protezione

Concetto nato in seguito a problemi notati nei computer degli anni '60 (ad esempio l'IBM 7090 che era a Pisa). Questi computer, potendo eseguire 1 solo programma alla volta, avevano il problema di avere molti tempi morti quando andavano in attesa di tempi meccanici (ad esempio il riavvolgimento del nastro, che durava qualche decina di secondi). Per non far perdere tempo ai costosissimi computer si è pensato quindi di far partire l'esecuzione, durante i tempi di attesa, di altri programmi sfruttando due caratteristiche delle macchine:

- il processore non ricorda nulla di ciò che è successo prima se non quanto è salvato nei registri.
- il programma¹ non ricorda nulla di ciò che è successo prima se non quanto c'è scritto nei registri e in memoria.

Per realizzare questo comportamento è quindi necessario avviare delle routine nei momenti opportuni (ad esempio quando JOB1 va in attesa e quando JOB1 è pronto a riprendere la sua esecuzione).

Questo comportamento però rende vulnerabile il funzionamento del programma ad azioni "sbagliate", eseguite ad esempio in JOB2 mentre JOB1 è in attesa, effettuate dall'utente che ha scritto il secondo programma. Il processore deve quindi poter distinguere il codice degli operatori, che sapendo cosa stanno facendo e operando nell'interesse del corretto funzionamento non hanno limitazioni, da quello degli utenti, che devono essere soggetti a limitazioni.

All'avvio il processore si trova nel contesto privilegiato, in questo modo l'utente non può eseguire le istruzioni:

- | | |
|-------|-------|
| • CLI | • IN |
| • STI | • OUT |

e ovviamente anche la HLT. **Non** può inoltre accedere allo spazio di memoria riservato ai programmi degli operatori.

¹in questo contesto i vari programmi prendono il nome di JOB1, JOB2 ecc.

La richiesta e la conseguente accettazione di un'interruzione, insieme al rilevamento di un'eccezione, sono gli unici (poi vedremo il terzo) modo per innalzare il livello di privilegio (**passaggio da livello utente a sistema**): questo avviene tramite il passaggio per la IDT che fa partire la routine collegata all'evento.

In altre parole la **protezione** è un meccanismo aggiunto ai calcolatori per fare in modo che il processore non obbedisca completamente all'utente e ai suoi programmi, evitando così che l'utente possa svolgere alcune operazioni, tra cui quelle elencate sopra. Per rendere possibile questo è stato aggiunto il concetto di **contesto**: quando la CPU si trova nel contesto privilegiato può eseguire tutto, altrimenti no.

Guardando un caso specifico, ovvero quello della **protezione nei processori INTEL/AMD a 64 bit**, funziona proprio in questa maniera: il processore può trovarsi in 2 livelli possibili:

- **Livello sistema**
- **Livello utente**

Il **livello sistema** corrisponde al contesto **privilegiato** appena illustrato, quello **utente** a quello **non privilegiato**. La discriminazione del livello avviene attraverso un registro del processore: il registro **CS**. In particolare se gli ultimi 2 bit di tale registro sono 00 il processore si trova a livello sistema, altrimenti a livello utente. Si può innalzare il livello di privilegio solo tramite il passaggio attraverso un gate della IDT e abbassarlo tramite l'istruzione IRETQ. Se si ha il processore a livello utente e si prova a svolgere una delle operazioni vietate, il processore solleva una **eccezione di protezione**, con conseguente esecuzione di una routine di sistema che termina il programma dell'utente e ne mette in esecuzione uno nuovo.

Nota: *La protezione serve per proteggere un utente da un altro, non a proteggere il sistema. Serve, come detto, per evitare che gli utenti eseguano operazioni che danneggerebbero altri utenti, ad esempio accedere e modificare la memoria M1.*

Le linee della IDT si chiamano gate, per passare quindi da livello utente a sistema bisogna attraversare un gate. Ogni linea è di 16 byte di cui 8 sono di indirizzo e 8 di informazione. Più nello specifico si ha:

- **P**: presenza. Indica se ci sono informazioni significative.
- **Un puntatore** alla routine a cui saltare (8 byte).
- **I/T**: *interrupt/trap*. Indica se il gate è di tipo *interrupt* o *trap*.
- **L**: livello. Indica il livello a cui il processore deve essere dopo l'attraversamento (non lo usiamo, riguarda il processore 80286 che aveva 4 livelli).
- **DPL**: *utente/sistema*. Indica il livello minimo di privilegio a cui si deve essere per poter attraversare il gate (solo in caso di interruzione software, vedi dopo). Questo campo serve in particolare per assicurarsi che l'utente invochi solamente routine di sistema e non eccezioni o routine per interruzioni esterne.

Ci sono 3 modi per attraversare i gate:

1. **Interruzioni esterne**.
2. **Eccezioni** (ovvero eventi interni alla CPU).

3. Interruzioni software.

Le interruzioni software si creano (a livello utente) tramite l'istruzione:

```
INT $tipo
```

Nota: Il valore di "tipo" è compreso tra 0 e 255 e indica il tipo del gate che si vuole attraversare.

Le routine che vanno in esecuzione tramite una INT prendono il nome di **primitive di sistema**. Le interruzioni software hanno lo stesso meccanismo delle interruzioni esterne e delle eccezioni di tipo *trap*, la differenza è che ora il tipo è il parametro. Il processore utilizza dunque il tipo per consultare la IDT e ricavare l'indirizzo della routine da mandare in esecuzione, la quale deve anch'essa terminare con IRETQ. A livello logico l'istruzione INT è molto simile alla CALL, ma la **differenza** molto **importante** è che la CALL richiede di specificare l'indirizzo al quale saltare, mentre la INT solo il tipo. Infatti se così non fosse gli utenti potrebbero saltare, ad esempio, in mezzo al codice di primitive, scavalcando così tutti i controlli.

Vediamo nel dettaglio cosa succede quando il processore accetta un'interruzione:

1. Si procura il tipo: se l'interruzione è esterna glielo passa l'APIC, in caso di eccezione è implicito mentre se l'interruzione è software è il parametro.
2. Usa il tipo per accedere al gate corrispondente della IDT, controlla subito il bit P: se è 0 genera un'eccezione per gate non presente.
3. Se sta gestendo un'interruzione software confronta il livello corrente con il campo DPL, se il livello corrente è meno privilegiato di quello indicato da DPL → eccezione di protezione.
4. Controlla il CS con L, se L è meno privilegiato → eccezione.
5. Salva il valore di RSP in un registro di appoggio SRSP.
6. Se si ha un innalzamento del livello di privilegio, da utente a sistema, si ha un cambio di pila.
7. Salva in pila 5 parole lunghe, ovvero:
 - (1) Una parola non significativa
 - (2) Il contenuto di SRSP
 - (3) Il contenuto di RFLAGS
 - (4) Il contenuto di CS
 - (5) Il contenuto di RIP (in cima)
8. Azzera in ogni caso il single-step (TF=0) e se il gate è di tipo *interrupt* resetta anche IF.
9. Salta all'indirizzo della routine puntata dal gate.

Il controllo al punto 3 viene fatto perché sostanzialmente non si vuole che l'utente possa accedere a gate riservati alle eccezioni e alle interruzioni esterne ai quali si può accedere solo da livello sistema. L'utente deve usare l'istruzione INT solo con tipi associati a primitive di sistema.

Il controllo al punto 4 in realtà nel nostro sistema è un po' ridondante e non è fondamentale. Serve comunque ad evitare che si abbassi il livello di privilegio quando si attraversa un gate. Come già spiegato il livello può solo alzarsi o restare uguale: se ad esempio abbiamo il processore a livello sistema, vuol dire che sta eseguendo istruzioni importanti, quindi non voglio che venga interrotto da una routine di livello utente, anche perché degli utenti non mi posso fidare e non so quando libereranno il processore.

Nel punto 6 si parla del cambio di pila: quando avviene un cambio di livello di privilegio da utente a sistema si ha un cambio di pila tramite il caricamento in RSP di un nuovo valore e il salvataggio nella nuova pila di 5 parole quadruple. Il motivo per cui si fanno questi salvataggi in una nuova pila è per evitare che l'utente possa, inserendo in RIP l'indirizzo di una funzione voluta e in CS il livello di privilegio sistema, creare problemi. Il cambio di pila, se si ha un innalzamento di livello, si ha perché il processore deve poter garantire di scrivere le 5 parole lunghe senza sovrascrivere altre cose e non può fidarsi del contenuto di RSP controllato dall'utente². Inoltre, anche se meno importante, è bene che queste informazioni siano salvate nella memoria di sistema M1, in modo che l'utente non le possa corrompere. Il puntatore alla nuova pila viene prelevato dal TSS attivo, puntato dal registro TR.

Per quanto riguarda l'istruzione IRETQ sappiamo che svolge le operazioni opposte rispetto a quelle del meccanismo dell'interruzione. Confronta il valore corrente di CS con quello salvato in pila, assicurandosi che quello salvato sia meno privilegiato di quello corrente, altrimenti si ha un'eccezione di protezione. Questo controllo viene fatto perché dopo la IRETQ, che ripristina il vecchio CS, il processore torna nella modalità in cui era prima. Ma, come detto in precedenza, l'istruzione IRETQ può solo abbassare (o lasciare inalterato) il livello di privilegio. Se non venisse effettuato questo controllo sarebbe facile per l'utente poter eseguire codice in modalità sistema. Inoltre essa ripristina i valori di RIP, CS, RFLAGS e RSP, leggendo i corrispondenti valori dalla pila.

Nota: Le istruzioni IN, OUT, CLI, STI non sono automaticamente vietate a livello utente ma sono abilitate se il flag IOPL (flag del registro dei flag) è uguale a CS.

Nota: I flag IF e IOPL non si possono modificare in nessun modo a livello utente (né con una IRETQ né con una PUSHF seguita da una POPF né in nessun altro modo). Se si prova a farlo, l'istruzione viene ignorata dal processore.

Nota: Avevamo visto che all'inizializzazione il processore si portava a livello sistema, inizializzava tutte le strutture dati sistema e poi passava a livello utente. Per fare ciò basta preparare una pila nello stesso stato in cui l'avrebbe lasciata una interruzione proveniente da livello utente ed esegua una IRETQ. Da quel momento il controllo passa agli utenti. Il sistema interverrà solo in risposta ad interruzioni esterne, eccezioni ed interruzioni software. Con l'attivazione dei processi vediamo questo meccanismo.

²Tenere sempre a mente che in modalità sistema non ci si può fidare di quello che è stato fatto dall'utente.

Capitolo 9

Introduzione al sistema multiprogrammato

9.1 Lezione 16 26/03

9.1.1 Processi

Un processo è un programma in esecuzione su dei dati in ingresso. In particolare è qualcosa di più astratto: un processo è la sequenza di stati attraverso cui il sistema

processore + memoria

passa eseguendo il programma, su quei dati, dall'inizio fino alla conclusione. L'esecuzione di ogni istruzione fa passare il processo da uno stato al successivo.

Logicamente un processo può dunque sembrare simile ad un programma, e infatti lo è, tuttavia ci sono delle differenze: un programma può generare più processi, in particolare anche più istanze del solito processo possono essere generate. Un processo invece fa riferimento ad un unico programma.

Non è solo il programma a far avanzare il processo ma anche l'input. Un processo può eseguire in sequenza più programmi.

Se prendiamo un programma con dei cicli, nel programma le istruzioni sono scritte una volta sola, nel processo le operazioni sono svolte più volte.

Un processo, quindi, si evolve nel tempo e durante la sua evoluzione possiamo fermarlo e "scattargli una foto": quella foto deve contenere tutte le informazioni che ci permettano di capire come si evolverà il processo, infatti, le informazioni, rappresentano lo stato del processo in quel momento.

Definizione 7 - Stato di un processo

Insieme delle informazioni dei registri del processore, memoria ecc. di un momento preciso di un processo. □

Lo stato, così inteso, è quello di cui parlavamo riguardo il passaggio dall'esecuzione di JOB1, in attesa di un tempo meccanico, all'esecuzione di JOB2. Prima di caricare quest'ultimo infatti si salva in memoria lo stato di JOB1 per poi passare all'esecuzione nel contesto di JOB2; quando poi bisogna riprendere l'esecuzione di JOB1, si prende l'ultimo stato di JOB2, si salva in memoria, si ricarica lo stato precedentemente salvato di JOB1 e si riprende la sua esecuzione.

I processi sono rappresentati da una serie di strutture dati, tra le quali la più importante probabilmente è il **descrittore di processo** in quanto conterrà lo stato del processo. Come già anticipato il nostro scopo è quello di poter eseguire più processi congiuntamente. Per fare ciò sfruttiamo appunto il descrittore di processo. Quando si vuole portare avanti un processo P1, carichiamo il suo stato all'interno dei veri registri e della vera memoria del processore. La normale esecuzione delle istruzioni farà avanzare lo stato di P1, quando poi si vuole passare ad un altro processo P2, carichiamo lo stato dei registri del processore e della memoria all'interno del descrittore di processo di P1 e, successivamente, carichiamo P2 caricando il suo stato.

La storia di un processo è indipendente da quella degli altri. Noi vogliamo però eseguire più processi con un solo processore. Questo accade perché per ora la RAM contiene un solo programma alla volta e deve accedere all'hard disk ogni volta che deve cambiare programma.

Nota: *I registri, nel momento del cambio di processo, non si salvano: ci sono delle strutture appropriate per questo scopo.*

Nota: *L'unica cosa privata di un processo è la pila.*

Nota: *Noi usiamo linux per compilare e poi carichiamo il programma sulla macchina QEMU con il bootloader.*

Se vogliamo poter portare avanti più processi bisogna **estendere il concetto di contesto**. Infatti ogni processo avrà un proprio contesto e quando il processo è in esecuzione stiamo operando implicitamente sul suo contesto. Il **contesto** di un processo conterrà tutta la sua **memoria privata** e una **copia dei registri** del processore. Quando carichiamo un processo rendiamo corrente il suo contesto.

I contesti dei singoli processi possono essere manipolati quando il processore si trova nel contesto privilegiato: più in generale si potranno manipolare non solo i contesti dei singoli processi ma anche tutte le parti di memoria condivisa tra i processi, le strutture dati sistema e si potrà interagire con le periferiche.

Nota: *Nel contesto privilegiato gira il software detto nucleo o kernel.*

Definizione 8 - Kernel

Codice che sta sempre in memoria eseguito in modalità sistema. □

Il compito del kernel è quello di realizzare l'astrazione tra i processi, gestire il cambio di processo, fornire le primitive che possono essere utilizzate dai processi utente per accedere in modo controllato alle risorse condivise.

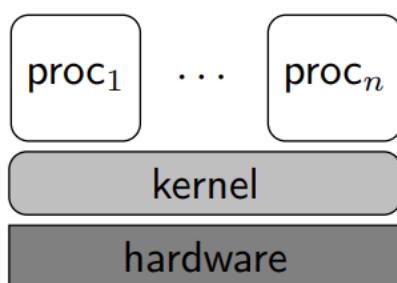


Figura 9.1: Schema "sistema" linux

Dalla figura possiamo vedere come i processi accedono all'hardware solo attraverso il kernel. Inoltre si ha che il processore si trova solo in un processo alla volta, dato l'**unico** flusso di controllo e dato il fatto che tra un processo e l'altro si deve sempre passare dal kernel. Quest'ultimo gestisce quindi i cambi di processi che avvengono, come già visto, in 3 casi:

1. Interruzione esterna
2. Eccezione
3. Interruzione

Il modulo sistema riprende quindi il controllo solo quando si attraversa un gate (3 casi appena descritti).

Nota : Anticipazione: *un esempio di interruzione esterna può essere un certo setting di tempo del timer.*

Per quanto riguarda il nostro sistema distinguiamo **3 moduli**:

1. **Modulo sistema**
2. **Modulo I/O**
3. **Modulo utente**

Il modulo sistema lo abbiamo appena visto mentre il modulo I/O contiene tutte le routine che permettono di interagire con le periferiche collegate al sistema. Questi 2 moduli gireranno con la CPU a livello sistema, il modulo utente, al contrario, con la CPU a livello utente. I primi 2 moduli sostanzialmente forniscono un supporto, sotto forma di primitive, agli utenti. Nel modulo utente si possono creare più processi, i quali verranno eseguiti concorrentemente.

Capitolo 10

Realizzazione dei processi

10.1 (Continuo Lezione 16)

Stati di esecuzione dei processi

Innanzitutto non bisogna confondere lo stato di un processo con il suo stato di esecuzione. Lo stato del processo indica il contenuto dei suoi registri e della sua memoria in un determinato momento nella vita di un processo, diversamente lo stato di esecuzione indica "la fase" in cui si trova un processo. I vari stati di esecuzione si possono vedere dalla figura sottostante.

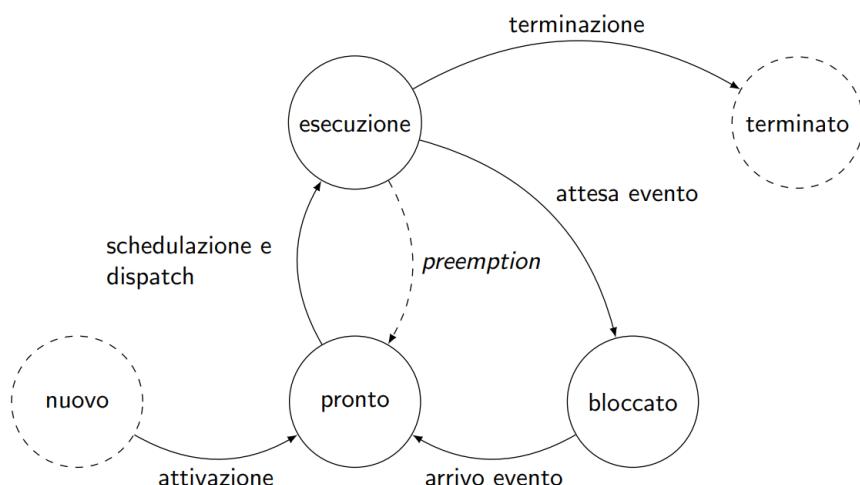


Figura 10.1: Grafo descrittivo dello stato di un processo

Un processo deve innanzitutto essere attivato. Con l'attivazione si ha la creazione di tutte le strutture dati necessarie come le pile e il descrittore di processo. Il primo processo viene creato dal sistema durante l'inizializzazione, poi i processi possono essere creati anche dinamicamente da altri processi, con il vincolo che un processo non ne possa creare un altro con priorità più alta della propria. **Un solo processo alla volta** può essere **in esecuzione**, caso in cui si rende corrente il suo contesto e il suo stato cambia nel tempo. Lo stato di un processo che non si trova in esecuzione resta costante nel tempo. Un processo in esecuzione può chiedere di terminare o di bloccarsi.

Definizione 9 - Processo bloccato

Processo in attesa di un evento, quando l'evento avverrà il processo ritornerà in coda pronti.

□

I processi **pronti** sono processi che potrebbero andare in esecuzione se ci fossero abbastanza processori. Un processo passa dalla coda pronti a esecuzione tramite l'operazione di *schedulazione* e *dispatch*. Un processo in esecuzione può dover lasciare il controllo del processore per via della **preemption**. Questa non è in tutti i sistemi (in quello che usiamo noi c'è in parte): se deve andare in esecuzione un altro processo quello corrente torna ad essere pronto. Essa può avvenire a seguito di un'interruzione, di un'eccezione o perché c'è un processo in coda pronti che ha priorità maggiore. La preemption è diversa dal caso di blocco perché in questo caso il processo torna in coda pronti.

Come già accennato, noi utilizziamo un sistema in cui ogni processo ha una priorità e ci si impegna per far sì che in esecuzione ci sia sempre il processo con priorità maggiore. Ad esempio negli esercizi quando bisogna inserire un processo in lista pronti bisogna sempre gestire il caso della preemption perché il processo inserito potrebbe avere priorità maggiore di quello che è attualmente in esecuzione.

Per quanto riguarda la memoria nel nostro sistema i processi hanno una parte di memoria privata (pila sistema, pila utente, descrittore di processo) e possono anche condividere zone di memoria. Lo scopo del **descrittore di processo** è contenere tutte le informazioni relative al processo, in particolare ha:

- Il campo **id** per identificare il processo.
- Il campo **livello** che indica se è di livello utente o livello sistema.
- Il campo **precedenza** che indica la priorità del processo.
- Un campo **puntatore** che punta alla pila sistema del processo.
- Un campo **contesto** con la copia dei registri del processore.
- Un campo **CR3** che indica la radice del Trie associato al processo (vedi memoria virtuale).
- Un campo *puntatore* per realizzare le code dei processi.

Il campo attualmente in esecuzione sarà puntato dalla variabile globale **esecuzione** di tipo puntatore a **des_proc**. Un'altra variabile globale dello stesso tipo è **pronti** che punta alla testa della lista dei processi pronti.

Un dettaglio è che avevamo visto che l'indirizzo a cui veniva trovata la pila sistema del processo è dentro il TSS attivo, identificato dal registro TR. Per fare in modo che il meccanismo delle interruzioni utilizzi la pila sistema del processo corrente dobbiamo sovrascrivere il segmento TSS ogni volta che cambiamo processo. Un'anticipazione è che con il cambio di processo viene sovrascritto anche il registro CR3 che indica la radice del Trie corrente.

Per evitare di dover gestire il caso in cui tutti i processi sono bloccati, nel quale il processore sarebbe in attesa e non farebbe niente, è stato creato un processo che è sempre pronto ed è a priorità minima, il processo **dummy**. Questo processo va quindi in esecuzione quando non ci sono altri processi pronti. La sua funzione varia da sistema a sistema, nel nostro conta i processi attualmente esistenti e se non ce ne sono esegue lo shutdown del sistema.

Cambio di processo

Quando il sistema deve scegliere un processo sceglie quello a priorità maggiore. Il cambio di processo può avvenire solo se il processo corrente si porta a livello sistema e lo può fare tramite i 3 metodi già visti. Quindi in tutti e 3 i casi il processore consulta la IDT, disabilita interruzioni, innalza il livello di privilegio e come già spiegato cambia pila. La nuova pila sarà la pila di livello sistema del processo in esecuzione e il processore ci salva le 5 parole quadruple già viste. Poi salta ad eseguire la routine, il cui indirizzo è scritto all'interno del gate. Faremo in modo che il codice a cui si salta in tutti i casi sia della forma:

```
1 CALL SALVA_STATO
2 .
3 .
4 .
5 CALL CARICA_STATO
6 IRETQ
```

La funzione **SALVA_STATO** ha lo scopo di salvare nel descrittore del processo corrente lo stato del processore, sostanzialmente fa la "fotografia" citata nella lezione precedente. Una cosa importante è che la **SALVA_STATO** salva anche il contenuto del registro RSP nel descrittore di processo, come lasciato dal processore dopo il cambiamento di pila e il salvataggio in questa delle 5 parole lunghe.

La funzione **CARICA_STATO** carica nel processore lo stato del processo puntato dalla variabile *esecuzione*, rendendo così corrente il suo contesto. La **CARICA_STATO** ripristina anche il valore di RSP (eventualmente del nuovo processo), in modo tale che la **IRETQ** vada ad estrarre le informazioni dalla pila sistema del processo che al momento della **CARICA_STATO** è puntato da *esecuzione*. Quindi per un eventuale cambio di processo bisogna cambiare il valore della variabile *esecuzione* tra la salva e la carica, in modo tale che a seguito della **IRETQ** si possa riprendere l'esecuzione del processo puntato correttamente dal punto in cui era stato interrotto.

Nota: Importante è notare che il cambio di processo vero e proprio si ha a seguito dell'esecuzione della coppia di istruzioni **CARICA_STATO-IRETQ** e non dopo il cambio di valore di *esecuzione*, quindi neanche dopo di una semplice chiamata di **schedulatore()**.

Altre note sulla **CARICA_STATO** sono:

- Si occupa di sovrascrivere il segmento TSS attivo, nel caso sia avvenuto un cambio di processo.
- Nel caso la primitiva invocata sia la **terminate_p()** o la **abort_p()**, nel caso in cui il processo da distruggere sia il processo corrente, la primitiva non deve provvedere a distruggere la pila sistema del processo invocante, perché potrebbe servire. Questo è compito della **CARICA_STATO**. In particolare la **CARICA_STATO** controlla che se il processo invocante è terminato, quindi è avvenuto un cambio di processo, a seguito del cambio del valore di RSP la carica si occupa di distruggere la pila sistema del processo che l'aveva invocata.

Si noti inoltre che quando il processore sta eseguendo codice del modulo sistema, tutti i processi si trovano in uno stato simile. Se questi erano stati precedentemente in *esecuzione*, l'unico modo in cui possono esserci usciti è passando dal meccanismo delle interruzioni, in uno dei tre modi possibili. Tutti saranno dunque passati dalla **SALVA_STATO** e quindi saranno adatti ad essere interpretati da una **CARICA_STATO** seguita da **IRETQ**.

Vediamo alcune situazioni di errore per capire l'importanza della `SALVA_STATO` e della `CARICA_STATO`.

1. Caso in cui si chiama la salva ma non la carica. Quindi salviamo correttamente lo stato del processo che era in esecuzione, chiamiamolo P1, poi cambiamo valore ad *esecuzione*, facendolo puntare ad un nuovo processo P2 e infine faccio `IRETQ`. Quello che succede è che il processo che effettivamente è in esecuzione è P2, però il contesto corrente è quello di P1 perché non abbiamo caricato i valori dei registri di P2 all'interno dei registri del processore. Così facendo la `IRETQ` andrà ad estrarre le parole dalla pila sistema puntata da `RSP`, il quale non essendo stato cambiato punta alla pila sistema di P1. La conseguenza è quindi una situazione confusa, di instabilità ed errore.
2. Caso è quello in cui non si chiama la salva, cambiamo il valore di *esecuzione* facendola puntare a P2 e infine eseguo la coppia carica e `IRETQ`. Per quanto riguarda P2 non ci sono problemi in quanto la `IRETQ` andrà a prelevare dalla pila sistema di P2 e quest'ultimo riprenderà dal punto in cui era stato interrotto. Il problema è che non facendo la salva si perde tutto l'aggiornamento dello stato di P1 dall'ultimo salvataggio fino ad ora, quindi se poi P1 dovesse essere ripreso non si riprenderebbe dal punto corretto e con un contesto non coerente.
3. Caso che si potrebbe analizzare è quello in cui non si chiama né la salva né la carica, si cambia *esecuzione* e si fa `IRETQ`. In questo caso probabilmente va in crash il sistema. In realtà questa situazione si fa con le primitive di I/O ma si parla di un argomento totalmente diverso e quindi si rimanda al paragrafo relativo.

Riassunto del meccanismo di cambio di processo Tramite la `INT` si attraversa un gate della IDT che porta all'esecuzione di una routine che avrà la forma prima illustrata.

Ciò che accade nello specifico in questa primitiva è:

1. Il salvataggio dello stato del processore nel descrittore di processo attivo in quel momento (lo si fa tramite la `SALVA_STATO`).
2. Cambio del valore di *esecuzione* .
3. Caricamento dello stato, salvato sempre in `des_proc`, del nuovo processo puntato da *esecuzione* all'interno del processore tramite `CARICA_STATO`.
4. Effettiva messa in esecuzione del nuovo processo, il cui stato è appena stato caricato, tramite la `IRETQ` ovvero l'estrazione del nuovo RIP e dei nuovi flags.

Attivazione di un processo

Nel nostro sistema un processo può creare un altro mediante la primitiva `activate_p()` che chiamerà la funzione `crea_processo()` che si occupa della creazione vera e propria del processo. La `activate` accetta come parametri:

- Un puntatore alla funzione che il nuovo processo deve eseguire.
- Un eventuale parametro per la funzione.
- Un'espressione che indichi la priorità del processo e il livello che deve avere il nuovo processo.

La activate controlla che un processo utente non provi a creare un processo di livello sistema e che non provi a creare un processo con priorità maggiore della propria. Poi quello che fa è chiamare la `crea_processo()` e se tutto è andato a buon fine inserisce il processo in lista pronti e restituisce l'identificatore del processo appena creato, 0xFFFFFFFF in caso di errore.

La `crea_processo()` deve fare in modo che il nuovo processo quando viene messo in esecuzione la prima volta parta dalla prima istruzione della funzione *f* passata, inoltre deve allocare ed inizializzare tutte le sue strutture dati. Vogliamo fare in modo che il nuovo processo si comporti come tutti gli altri che si trovano in coda pronti, possiamo inizializzare il descrittore di processo e pila sistema come se anch'esso si fosse precedentemente portato da livello utente a livello sistema, subito prima di eseguire la sua prima istruzione. In particolare la crea farà le seguenti cose:

- Alloca un nuovo descrittore di processo con tutte le caratteristiche richieste per questo processo, e aggiorna la `proc_table` e il numero di processi, l'indice nella `proc_table` sarà anche il suo id.
- Inserisce in RDI del contesto il valore del parametro della funzione di partenza.
- Alloca la radice dell'albero di traduzione.
- Alloca la pila sistema del processo e se il processo è a livello utente alloca anche quella utente.
- Mette dentro la pila le 5 parole quadruple, quelle che il meccanismo di interruzione avrebbe dovuto salvare. Come RIP metto l'indirizzo della funzione passata come argomento, poi inserisco il livello, il campo RFLAGS con *IF* = 1 e *IOPL* = *sistema* e un campo puntatore che punta alla pila utente opportunamente allocata (se processo di livello sistema metto puntatore alla pila sistema).
- Importante inizializzare il campo `contesto[I-RSP]` in modo tale che, in seguito alla carica_stato, l'RSP del processore punti alle informazioni che abbiamo scritto in pila sistema.

In questo modo la `IRETQ`, che segue la `CARICA_STATO`, estrae correttamente le informazioni dalla pila e fa saltare il processore alla prima istruzione della funzione passata, con le interruzioni abilitate e pronto a usare la pila utente (o sistema) puntata dal puntatore che aveva messo in pila.

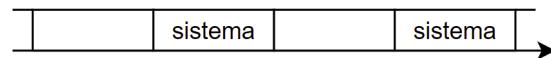
10.2 Lezione 17 04/04

10.2.1 Atre note sui processi

- Creando un processo dentro *main* questo avrà comunque priorità minore di *main* qualunque priorità gli si assegna.
- Le variabili globali sono condivise tra tutti i processi.
- La tabella dei processi contiene tutti i processi attivi.

- Un processo è la storia dell'esecuzione di un programma (concetto dei tanti fotogrammi in cui ognuno corrisponde alla foto scattata nel momento della fine dell'esecuzione di una istruzione).
- Per ogni processo abbiamo un descrittore di processo: ognuno contiene 1 foto del processo a cui è legato.

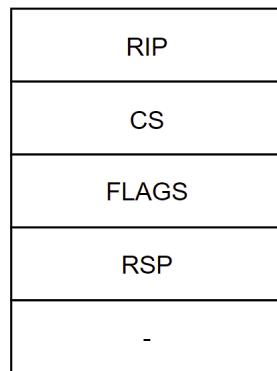
La CPU, durante l'esecuzione, o è a livello sistema o è fuori:



Quando il processore si trova fuori dalla linea sistema sta eseguendo il processo, sempre 1 alla volta, portandone avanti la storia.

Riguardo alla storia del processo è bene ricordare che la foto scattata e salvata nel descrittore non è aggiornata durante l'esecuzione del processo: l'aggiornamento avviene infatti al momento di passaggio a livello sistema (che avviene sempre in uno di quei 3 casi), a seguito della salva stato; non è dunque possibile recuperare stati intermedi del processo.

Per **saltare da un processo ad un altro** la CPU salva nella **pila sistema** le seguenti informazioni:



Nota: l'ultima cosa, segnata con il trattino, al momento non ci interessa

Nota: La foto di un processo è composta in parte dalle informazioni salvate in pila sistema dal meccanismo delle interruzioni e in parte dal contenuto del descrittore di processo come aggiornato dalla `salva_stato`.

Capitolo 11

Realizzazione delle primitive

11.1 (Continuo Lezione 17)

11.1.1 Realizzazione delle primitive

Partiamo da un problema generale: le primitive del nostro sistema devono lavorare su un insieme di strutture dati globali, come i descrittori di processo e le code dei processi. Se mentre una primitiva sta operando su una di queste strutture viene interrotta da un'altra primitiva che deve operare sulla stessa struttura si crea confusione ed inconsistenza. Si parla dunque del problema di **interferenza** tra 2 flussi di esecuzione che lavorano sulla stessa struttura dati. Il nostro interesse è che una struttura si trovi in uno stato consistente all'inizio e alla fine dell'operato su di essa. Un esempio di interferenza può essere una **SALVA_STATO** che viene interrotta da un'altra **SALVA_STATO**: si creerebbero vari problemi. Abbiamo 2 modi per evitare i problemi dell'interferenza:

1. il primo sarebbe scrivere le primitive tenendo conto di tutti i possibili casi di interruzione, ma è complicato.
2. Il **secondo metodo è quello utilizzato** e consiste nell'eliminare tutte le possibili sorgenti di interruzione durante l'esecuzione delle primitive. In particolare faremo in modo che tutti i gate della IDT siano di tipo *interrupt*, quindi le primitive gireranno con le interruzioni esterne disabilitate e staremo attenti a non usare interruzioni software e a non sollevare eccezioni. In questo modo le nostre primitive avranno la proprietà dell'**atomicità**, cioè una volta iniziata verranno portate a compimento senza che vengano interrotte.

Nota: *Essendo le primitive atomiche, dall'esterno non si può vedere lo stato intermedio di esse.*

Definizione 10 - Primitiva

CONTROLLARE Una primitiva è una routine (codice già pronto) fornito dal modulo sistema ed utilizzabile dall'utente per svolgere operazioni in modo protetto. Ad esempio gestire il cambio di processo o il trasferimento da/verso una periferica. Dal livello utente le primitive sono chiamabili tramite una interruzione software. □

Prima di vedere più nel dettaglio come si realizza una nuova primitiva (che è quello che la maggior parte delle volte è richiesto nel compito) vediamo velocemente il **meccanismo di chiamata**.

11.1.2 Meccanismo di chiamata

Questo meccanismo coinvolge sia il linguaggio assembler che il linguaggio c++, in particolare l'utente chiama la primitiva come se fosse una funzione c++, con eventuali parametri, da *utente.cpp*. La dichiarazione della funzione viene fatta come *extern* (così non si prevede l'overloading delle funzioni e il loro nome non richiede traduzione) in *sys.h*. A questo punto va in esecuzione un piccolo programma assembler che si trova in *utente.s* e si limita ad eseguire la INT con il tipo associato e ciò farà sì che si attraversi il gate relativo della IDT e si passi al modulo sistema. Come già visto, all'interno del gate è presente l'indirizzo della routine a cui saltare, questa routine quasi sempre conterrà solo la **SALVA_STATO**, la **CARICA_STATO**, la **IRETQ** e tra la salva e la carica la chiamata alla primitiva vera e propria che per comodità viene scritta in c++ in *sistema.cpp*. Attraverso la **IRETQ** il processore verrà riportato a livello utente e si continuerà ad eseguire il processo il cui descrittore è puntato dalla variabile globale *esecuzione*. Nel caso in cui una primitiva debba restituire un valore bisogna modificare il campo **contesto[I_RAX]** del processo che deve restituire il valore; non si può semplicemente ritornare il valore con una **return risultato**. Infatti così facendo il compilatore metterebbe risultato dentro RAX, ma la **CARICA_STATO** sovrascriverebbe il valore di RAX scrivendoci il valore contenuto nel registro RAX che andrà in esecuzione. Questo è anche il motivo per cui in *sistema.cpp* anche se la primitiva è definita con un certo tipo, qui si dichiara *void*. Infatti in questo modo se devo restituire un valore lo metto dentro RAX, se la dichiarassi ad esempio INT si aspetterebbe un valore di ritorno attraverso una **return**, ma per i motivi appena detti è sbagliato.

Nota: Le primitive di I/O che non chiamano salva e carica stato possono anche essere definite con un tipo diverso da void.

Nota: Altro caso è quello in cui siamo sicuri che la primitiva non causi cambi di processo. Se così fosse non serve utilizzare la **SALVA_STATO** e la **CARICA_STATO** e si potrebbe ritornare il valore con una semplice **return**.

11.1.3 Realizzazione di una nuova primitiva

1. Innanzitutto bisogna assegnare un tipo di interruzione alla primitiva, si può definire un nuovo tipo nel file *costanti.h*.
2. Bisogna dichiarare anche la primitiva in *sys.h* come abbiamo visto prima.
3. Bisogna definire il nuovo gate della IDT, lo si può fare in *sistema.s* attraverso la macro **carica_gate** che vuole come parametri:
 - Il tipo della primitiva.
 - L'indirizzo della routine a cui saltare quando qualcuno usa il gate.
 - Il livello di privilegio minimo che bisogna avere per attraversare il gate.
4. Sempre in *sistema.s* viene scritta la parte assembler della routine come abbiamo visto prima.
5. In *sistema.cpp* si scrive la primitiva vera e propria in c++.
6. In *utente.s* scriviamo la INT con il nuovo tipo associato. In questo modo l'utente l'unica cosa che deve fare è chiamare la routine come se fosse una primitiva in *utente.cpp*.

Una possibile domanda è: l'utente può in qualche modo accedere al modulo sistema e scavalcare la INT? La risposta è no. Anche se io utente sapessi gli indirizzi di dove sono salvati i dati che mi servono non potrei accederci, perché la protezione è fatta in hardware e la MMU non mi permette di accedere agli indirizzi riservati al sistema (vedi parte su memoria virtuale).

11.1.4 Funzioni di supporto per le primitive

- **des_proc* des_p(nat1 id)**

Restituisce un puntatore al descrittore del processo di identificatore **id** (**nullptr** se tale processo non esiste).

- **void schedulatore()**

Sceglie il prossimo processo da mettere in esecuzione (cambia il valore della variabile **esecuzione**).

- **void inserimento_lista(des_proc& p_lista, des_proc* p_elem)**

Inserisce **p_elem** nella lista **p_lista**, mantenendo l'ordinamento basato sul campo **precedenza**. Se la lista contiene altri elementi che hanno la stessa precedenza del nuovo, il nuovo viene inserito come ultimo tra questi.

- **des_proc* rimozione_lista(des_proc& p_lista)**

Estrae l'elemento in testa alla **p_lista** e ne restituisce un puntatore (**nullptr** se la lista è vuota). (Dal codice: estrazione del processo a maggiore priorità dalla lista passata come argomento).

- **void inspronti()**

Inserisce il **des_proc** puntato da **esecuzione** in testa alla coda **pronti**.

- **void c_abort_p()**

Distrugge il processo puntato da **esecuzione** e chiama **schedulatore()**.

11.1.5 Chi esegue le primitive

Un ultimo concetto molto importante su questa parte è **chi esegue le primitive**. Si potrebbe pensare che le primitive siano eseguite dal processo che era in esecuzione quando è stata chiamata la primitiva, ma non è così. Riprendendo la definizione di processo, ovvero "*un processo è la sequenza di stati che il sistema processore + memoria attraversano durante l'esecuzione di un programma con dei dati di ingresso*", notiamo che durante l'esecuzione di una primitiva il processo però non si evolve: non si ha un'evoluzione dei suoi stati. Tutta l'esecuzione di una primitiva si svolge tra 2 stati del processo, quest'ultimo va quindi pensato congelato.

Più nel dettaglio, all'ingresso di una primitiva il meccanismo delle interruzioni e la funzione **SALVA_STATO** scattano un fotogramma dello stato del processo invocante. Fino a che il processo non torna in esecuzione, quindi per tutta la durata della primitiva, il processo resta fermo a quel fotogramma e il suo stato non si evolve, non sta compiendo azioni. Di conseguenza le primitive vengono eseguite da un'entità diversa dai processi, anche se prendono in prestito le risorse dei processi che le invocano, come ad esempio l'albero di traduzione della memoria virtuale corrente.

Infine riprendiamo un concetto già affrontato. All'inizio della primitiva la variabile globale esecuzione punta al descrittore del processo invocante e se non viene cambiata così sarà anche alla fine. Facciamo conto che all'interno della primitiva chiamiamo la funzione `schedulatore()`, la quale cambia il valore a *esecuzione*, ciò non significa che stiamo cedendo il controllo del processore in quel momento. Come abbiamo appena visto le primitive non sono eseguite da nessun processo, come faccio allora a cambiare il processo corrente se non c'è nessun processo corrente?. Se questo non bastasse le primitive atomiche non possono essere interrotte e `schedulatore()` non contravviene a questa regola. Un eventuale cambio effettivo del processo in esecuzione avviene a seguito della coppia di istruzioni `call carica_stato-iretq`.

(Esempio al proiettore *nucleo-7.1.1*).

Capitolo 12

Semafori

12.1 Lezione 18 05/04

Esempi sui processi visti con il debugger, comandi utili:

- *process dump 5*: stampa il dump del processo 5, si vede quindi cosa c'è nel descrittore di quel processo
- *process list*: stampa la lista dei processi
- *idt*: fa vedere tutti i gate
- *si*: esegue una singola istruzione del linguaggio macchina
- *context*: fa vedere tutte le info di contesto

Gli utenti possono programmare il sistema e più processi ma non hanno il controllo di come vengono eseguiti: non si possono quindi fare supposizioni su come i processi vengano eseguiti.

Esempio 8 - Presentazione del problema: Abbiamo due processi e un contatore globale: vogliamo contare quanti processi vengono eseguiti.

```
1 int counter = 0;
2
3 P1
4 //counter++;
5 MOV counter, %RAX
6 //processo 2
7 INC %RAX
8 MOV %RAX, counter
```

Nel codice di esempio supponiamo dunque che, al momento del commento, parta il processo 2, anch'esso intento ad incrementare la variabile *counter*:

```
1 P2
2 //counter++;
3 MOV counter, %RAX
4 INC %RAX
5 MOV %RAX, counter
```

I due processi stanno parlando a due $\%RAX$ differenti in quanto ogni processo ha il proprio contesto. Se quindi si verifica un'esecuzione come quella nell'esempio, alla fine *counter* varrà 1, quando invece ci aspettiamo che valga 2 (in quanto sono stati eseguiti 2 processi). \square

Il problema appena visto viene quindi detto **dell'interferenza dei processi** ed è quasi impossibile da debuggare, va quindi **evitato** a priori. Questo succede perché i processi hanno sia una parte di memoria privata (come le pile utente e sistema o il descrittore di processo) sia zone di memoria condivise, come strutture dati globali o variabili globali (come il *counter* dell'esempio). Condividendo parti di memoria bisogna evitare situazioni come quella dell'esempio: un processo sta operando su una struttura dati condivisa e viene interrotto da un altro processo che opera sulla stessa struttura. Non si può operare come nelle primitive garantendo atomicità, i processi devono essere eseguiti con le interruzioni abilitate. Altrimenti per evitare questo tipo di problemi si potrebbe pensare di usare le primitive **CLI()** e **STI()**, ma dato che per definizione non ci possiamo fidare degli utenti questo non è possibile, una eventuale dimenticanza di **STI()** disabiliterebbe le interruzioni per sempre.

Analisi delle possibili situazioni Si distinguono due casi:

1. **Mutua esclusione**: i due processi vanno avanti per fatti loro senza interferire con l'altro (in questo caso non ci interessa l'ordine). In generale con la mutua esclusione si fa sì che operazioni diverse non si mescolino tra loro.
2. **Sincronizzazione**: produzione, da parte di P1, di un dato nuovo da porre in un buffer dal quale viene letto da P2 solo quando è nuovo. In generale l'utente può volere che una determinata operazione sia svolta prima di un'altra.

Nota: *In questo caso ci interessa l'ordine.*

Unico metodo risolutivo (ideato da Dijkstra): **meccanismo dei semafori**.

12.1.1 Meccanismo dei Semafori

Questo metodo si basa sull'**idea** di fondo delle *scatole contenenti gettoni* sulle quali si possono effettuare solo 2 operazioni:

- Prendere un gettone.
- Lasciare un gettone.

con il **vincolo aggiuntivo** di poter prendere solo se c'è, aspettare altrimenti.

Per risolvere i problemi di mutua esclusione (caso 1) è sufficiente 1 scatola con all'interno 1 gettone.

```

1 S =INI(1)
2
3 P1
4 PREND0(S)
5 SEZ. CRITICA1
6 LASCIO(S)
7
8 P2
9 "

```

```
10 | P3
11 |
12 | "
13 |
14 | ecc.
```

Nota: *PREND0(S)* è una primitiva di sistema.

Chi arriva per primo alla scatola prende il gettone e una volta che ha eseguito l'operazione ce lo rimette. Se qualcuno arriva e non trova il gettone viene bloccato in attesa che qualcuno ce lo inserisca.

Nota: chi lascia un gettone potrebbe avercelo "già in tasca": non deve per forza averlo preso.

Per risolvere i problemi di sincronizzazione (caso 2) è sufficiente 1 scatola con all'interno 0 gettoni. L'idea è che se devo svolgere prima un'operazione A rispetto ad un'operazione B, dopo aver eseguito l'operazione A bisogna lasciare il gettone nella scatola, così B può prenderlo ed eseguire l'operazione. Nel caso in cui arrivi prima B di A non ci saranno gettoni e quindi B viene sospeso fino a che non è stata fatta l'operazione A.

In generale per risolvere questo tipo di problemi bisogna decidere quante scatole utilizzare, quanti gettoni devono contenere e stabilire delle regole su chi deve inserire/estrarre gettoni e quando. Nel nostro sistema le strutture dati che rappresentano le scatole con i gettoni sono, come già anticipato, i semafori. Le primitive che il sistema mette a disposizione sono:

- **natl sem_ini(natl v)**

Permette di allocare un nuovo semaforo con v gettoni, restituisce l'identificatore del semaforo.

- **void sem_wait(natl sem)**

Permette di estrarre un gettone dal semaforo di identificatore *sem*, se non ci sono gettoni il processo viene sospeso.

- **void sem_signal(natl sem)**

Permette di inserire un gettone nel semaforo di identificatore *sem*, se c'è un processo in attesa lo risveglia.

In questo modo possiamo garantire **sia la mutua esclusione sia la sincronizzazione** attraverso l'uso di queste primitive.

Nota: Specialmente negli esercizi di I/O è richiesto di utilizzare i semafori per gestire la mutua esclusione su una periferica.

Tutte queste 3 sono primitive di sistema, chiamabili anche da livello utente. In quanto tali prevedono l'attraversamento di un gate della IDT e presentano al loro interno la salva e la carica stato, seguite dalla iretq.

Nota: Sia la wait che la signal prima di usare il parametro *sem* devono controllare che sia valido, se non lo è si fa una abort del processo. Nei compiti si può utilizzare la funzione ausiliare *sem_valido* per fare questo controllo.

Nota: Il meccanismo delle interruzioni è una cosa globale, nel senso che quando va in esecuzione una routine o una primitiva tutti i processi vanno pensati congelati, è una cosa che comprende tutto il sistema, mentre se un processo viene bloccato su un semaforo, viene sospeso solo quel processo e ne verrà messo in esecuzione un altro.

Struttura dati "semaforo"

Analizziamo la struttura dati utilizzata per i semafori:

```
1 struct des_sem{  
2     int counter;  
3     des_proc* pointer;  
4 };
```

Il campo *counter* conta i gettoni presenti nel semaforo, viene incrementato a seguito di una *signal* e decrementato a seguito di una *wait*. Nel caso in cui un processo che fa una *wait* non trova gettoni, questo campo viene comunque decrementato, anche in senso negativo. In quel caso il suo valore assoluto indica quanti processi sono sospesi su quel semaforo. Il campo *pointer* indica la lista di processi sospesi, a seguito di una *signal* un eventuale processo sospeso viene risvegliato, si estrae dalla testa, e si gestisce un'eventuale preemption.

Nota: *In totale nel nostro sistema si può avere $2 \times MAX_SEM$ semafori, i primi MAX_SEM sono dedicati all'utente, mentre la seconda metà sono dedicati al modulo I/O.*

Capitolo 13

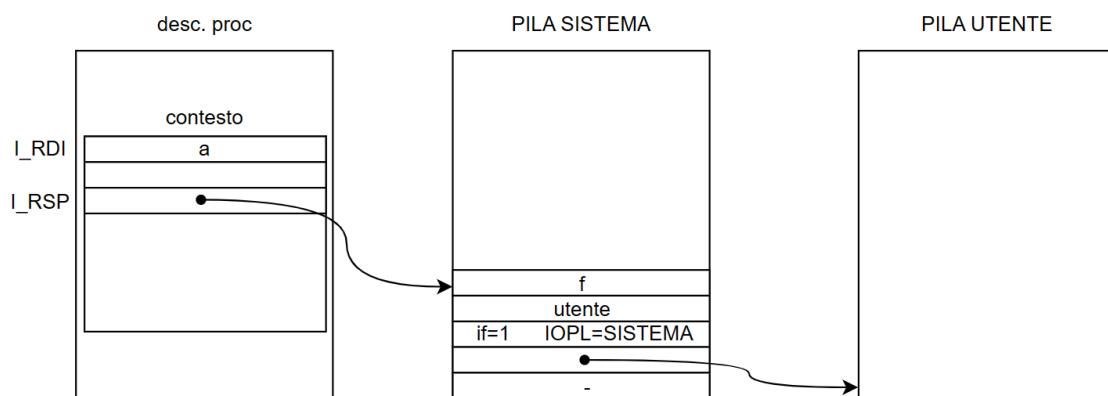
Sospensione dei processi

13.1 Lezione 19 08/04

Prima parte: esempi con il debugger. Comandi utili:

- *semaphore*: fa vedere lo stato dei semafori.

Tramite la primitiva `activate_p(f, a, prec, liv)` cerchiamo, via software, di ricreare quello che succede via hardware:



Vedi immagine sopra: mi comporto in modo che la IRETQ trovi "le cose che vuole al posto giusto", ad esempio dove ci va RIP ci trova *f* ovvero l'indirizzo della prossima istruzione da eseguire ovvero la funzione *f*.

13.1.1 Sospensione dei processi

Quando abbiamo analizzato i possibili stati di un processo avevamo visto che un processo in esecuzione può sospendersi o in attesa di una determinata risorsa o per un determinato periodo di tempo. Nel tempo in cui un processo è sospeso il sistema può eseguire altri processi.

Esempio 9: Primitiva `delay(nat1 n)`: sospende un processo per *n* cicli del timer. □

Definizione 11 - Processi sospesi

Sono quei processi bloccati che aspettano un evento per riprendere. □

Per attuare la primitiva dell'ultimo esempio abbiamo quindi bisogno di misurare il tempo: programmiamo il timer per interrompersi periodicamente (inizialmente questo tempo era di $50ms$). Facciamo sì che ad ogni ciclo del timer, venga inviata una richiesta di interruzione e vada in esecuzione la routine associata: il **driver del timer**. Questa routine ha lo scopo di decrementare il campo n dei processi che sono sospesi e se $n = 0$ rimetterli in coda pronti, gestendo un eventuale preemption.

Nota: *Quindi possiamo avere più processi sospesi.*

Ordiniamo questi in una lista in modo tale che ogni elemento della lista ricordi solo quanti cicli n deve attendere in più rispetto all'elemento precedente in lista. In questo modo si ha il vantaggio che il driver dovrà decrementare il campo n solo del primo elemento in lista (e non percorrere tutta la lista ogni volta per modificare tutti i campi). Se $n = 0$ andrà svegliato sicuramente il primo elemento in lista più eventualmente altri elementi (vedi figura 1). Eventualmente se bisogna inserire un nuovo elemento, quindi si fa una nuova richiesta, bisogna eseguire un inserimento ordinato e bisogna decrementare il contatore dell'elemento che si trova in lista subito dopo all'elemento appena inserito.

Per implementare questo meccanismo e gestire le varie richieste di *delay* utilizziamo la struttura "richiesta", in particolare essendo queste diverse operiamo con una lista:

```

1 struct richiesta{
2     natl d_attesa;
3     richiesta* p_rich;
4     des_proc* pp;
5 };
6 richiesta *sospesi;
```

Il campo d_attesa indica il contatore di quanti cicli di timer deve attendere il processo sospeso, p_rich punta al prossimo processo in lista e serve ad implementare la lista dei processi sospesi mentre pp punta al descrittore di processo sospeso.

Nota: *La testa della lista si chiama "richiesta* sospesi".*

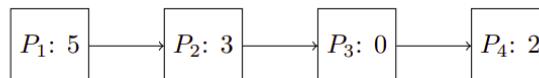


Figura 1: Esempio di lista di processi sospesi.

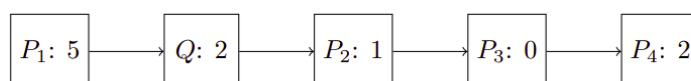
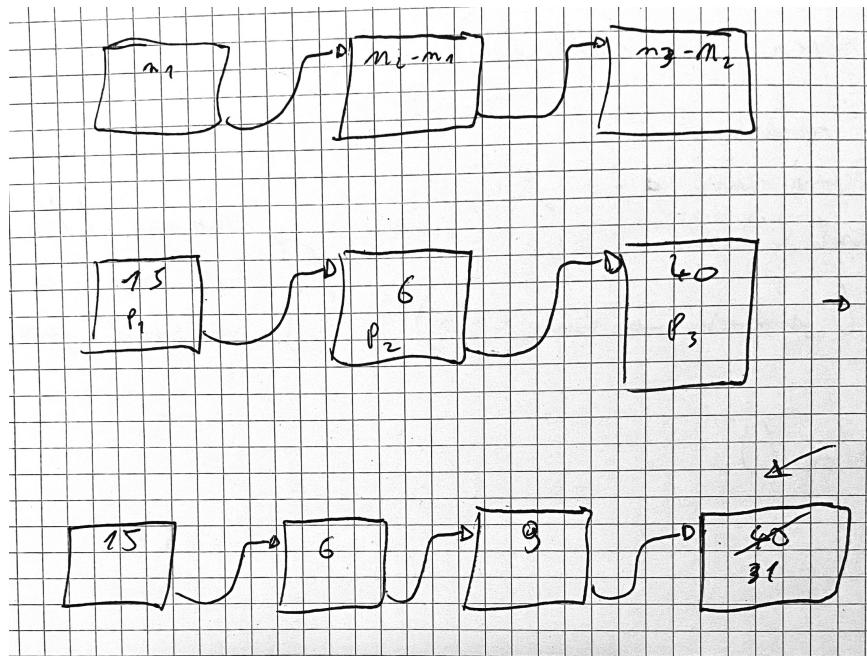
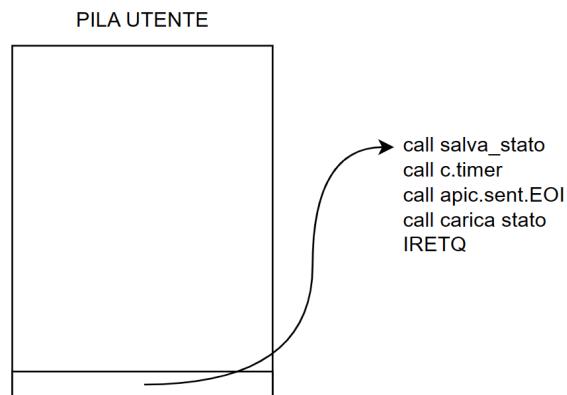


Figura 2: La lista di Figura 1 dopo l'inserimento di un nuovo processo.



Nota: nell'esempio nella foto dal quaderno P_2 deve aspettare $15 + 6 = 21$, e allo stesso modo si devono comportare gli altri: quanto infatti andiamo a inserire il processo che vuole aspettare 30 il suo ritardo richiesto diventerà 9 poiché $15 + 6 + 9 = 30$.



13.1.2 Driver del timer

Analizziamo ora il driver del timer, che, come già detto, è la routine di che va in esecuzione quando il timer invia una richiesta di interruzione. Ricordiamo che il timer nel nostro sistema ha priorità massima. Il driver del timer è eseguito atomicamente, quindi non può essere interrotto e si ha la garanzia di mutua esclusione sulla lista.

```

1 //Parte assembler:
2 driver_td:
3     call salva_stato
4     call c_driver_td
5     call apic_send_EOI
6     call carica_stato
7     iretq

```

Per quanto riguarda la parte assembler notiamo che si comporta come una normale primitiva, l'unica cosa è che viene mandato l'end of interrupt. Questo è importante perché altrimenti non passerebbero richieste di interruzioni con priorità minore o uguale di quella del timer, quindi nel nostro sistema nessuna. Se non venisse mandato nessuno resetterebbe il bit a 1 relativo nel registro ISR e quindi non passerebbe più nulla. In particolare abbiamo visto che quando si parla di interruzioni esterne vengono settati bit nel registro IRR e ISR e la richiesta di interruzione viene inoltrata solo se ha priorità + alta di quella + a sinistra nel registro ISR.

Questo problema non si ha ovviamente quando si parla di interruzioni software perché in quel caso sono atomiche e si torna al normale flusso di controllo quando termina la primitiva.

```

1 //Parte C++:
2 extern "C" void c_driver_td(void){
3     inspronti();
4     if(sospesi)
5         sospesi->d_attesa--;
6     while(sospesi && sospesi->d_attesa == 0){
7         inserimento_lista(pronti, sospesi->pp);
8         richiesta* p = sospesi;
9         sospesi = sospesi->p_rich;
10        delete p;
11    }
12    schedulatore();
13 }
```

Chi esegue il driver

In questo caso è più facile arrivare al fatto che il driver non è eseguito da nessun processo e che quando si sta eseguendo il driver non è in esecuzione nessun processo. Il processo che era in esecuzione non ha volontariamente chiamato il driver. Proprio come per le primitive, il driver prende in prestito le risorse del processo che era in esecuzione prima dell'attraversamento del gate (come la pila sistema e la memoria privata) e anche il driver utilizza la salva e la carica stato seguite da una IRETQ.

Capitolo 14

Paginazione

14.1 Lezione 22 12/04

14.1.1 Memoria virtuale

Riprendiamo i concetti che abbiamo visto fino ad ora sulla memoria:

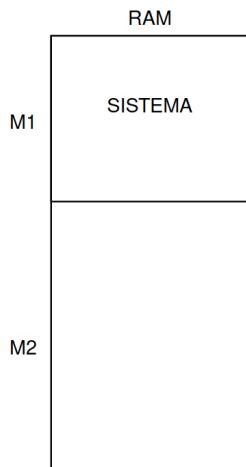


Figura 14.1: La protezione viene effettuata in hardware

Lo stato di un processo non è solo lo stato dei registri ma anche cosa c'è in memoria, comprende infatti sia lo stato della CPU sia lo stato della memoria privata del processo: quando quindi si cambia processo bisogna cambiare anche memoria e la precedente deve essere dunque salvata sull'hard disk (noi non vedremo questo comportamento ma comunque era usato su alcuni sistemi). In questo modo ogni processo pensa di avere una CPU e una memoria M2 tutta per sé, ma in realtà ha una CPU e una memoria M2 virtuale usate anche da altri.

La vera CPU e la vera memoria M2 incarnano ad ogni istante la CPU virtuale e la memoria M2 virtuale del processo attualmente in esecuzione. Mentre l'ultimo stato delle CPU virtuali e delle memorie virtuali dei processi non in esecuzione sono al sicuro nei campi dei descrittori di processo e nell'hard disk.

Vediamo i **vantaggi** di usare questo metodo:

1. **Isolamento tra i processi:** ogni processo ha accesso solo alla sua memoria privata e eventualmente alla zona condivisa. (es. P1 non ha accesso al descrittore di processo di P2, dato che questo sta nella parte sistema della memoria, e alla memoria di P2, dato che sta nell'hard disk).
2. **Collegamento semplificato:** dato che l'utente deve programmare, ovvero mettere in memoria lo stato iniziale della macchina, deve sapere a partire da quale indirizzo lo può caricare. In particolare qualunque processo pensa di poter avere l'intera memoria a sua disposizione.
3. **Swap semplificato:** scambio dei processi semplificato. La memoria viene ricaricata esattamente nella stessa posizione, tutti gli indirizzi che un processo usa sono sempre validi.
4. **Condivisione della memoria possibile:** Si può condividere zone di memoria evitando di sostituire le parti di memoria condivise quando si cambia processo.

L'obiettivo è cercare di eliminare o quantomeno ridurre il più possibile le operazioni di *swap*. Vediamo man mano dei metodi sempre più sofisticati fino ad arrivare a quello che è il metodo utilizzato, ovvero quello della paginazione.

Registri INF e SUP

(spiegato approssimativamente) I programmi scritti ad alto livello hanno delle strutture complesse che vengono gestite in questo modo tramite l'aggiunta dei registri INF e SUP:

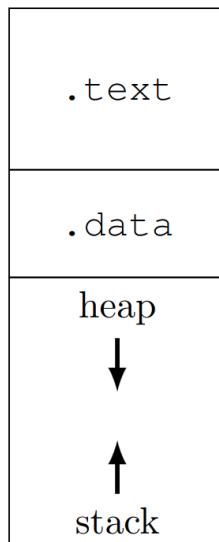


Figura 14.2: I puntatori registri INF e SUP contengono rispettivamente l'indirizzo dell'inizio della sezione .text e l'indirizzo della sezione stack/heap.

Come si vede dall'immagine sopra, un processo ha in memoria una dimensione massima: i primi 2 settori sono decisi a tempo di compilazione, l'altro lo decide qualcun altro.

Nota: Se si prova ad accedere fuori da SUP si genera una eccezione.

Caratteristiche della dimensione fissa:

- Devo copiare solo quel blocco e non tutta la RAM quando si fa uno *swap*.
- Per ogni processo devo avere un INF e un SUP da caricare nei rispettivi registri quando il processore va in esecuzione.
- Con INF e SUP si perde il vantaggio 2: non sapendo l'indirizzo di un processo finché non viene caricato in memoria.

Soluzione per risolvere questa perdita:

- se l'architettura del processore lo permette si usa un indirizzamento relativo (il programma può essere traslato in memoria e funzionare sempre).
- altrimenti (ad esempio nei sistemi MS-DOS) si usa la scrittura di un programma rilocabile.

Vediamo come mantenere il vantaggio 3 (swap semplificato):

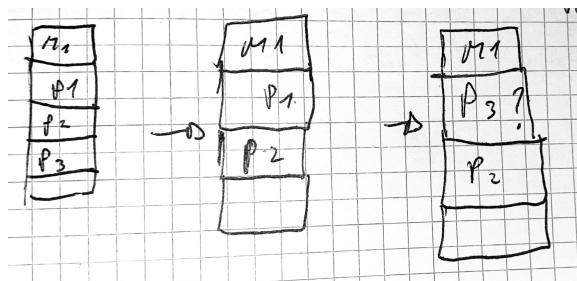


Figura 14.3:

Ciò che avviene nell'immagine non si può fare perché non so quali indirizzi assoluti aveva scritto P3 prima di essere terminato. Lo *swap* non è quindi impossibile, per realizzarlo in con questo metodo devo stare attente a caricare ogni processo negli indirizzi in cui era stato caricato la prima volta (non vantaggioso).

Nota: Anche il vantaggio 4 (condivisione della memoria) non viene mantenuto.

Registri base e limite Analizziamo ora i vantaggi nel caso di utilizzo, al posto di INF e SUP, dei registri BASE e LIM. Con questi si ha che tutti gli indirizzi generati dal programma, prima di accedere alla ram vengono sommati al contenuto di BASE, e confrontati con il contenuto di LIM.

Nota: In particolare BASE contiene lo stesso valore che conteneva INF, solo che invece di usarlo come limite inferiore viene usato come somma.

Si ha quindi che il primo vantaggio (isolamento tra processi) si mantiene, seguendo lo stesso comportamento prima descritto per INF e SUP, come anche il secondo (collegamento semplificato), in quanto basta dire ai programmatori di partire con i loro indirizzamenti da 0, sarà poi l'hardware ad occuparsi di modificare gli indirizzi opportunamente (la somma con BASE è infatti eseguita dall'hardware). Così i programmatori partiranno sempre con il loro indirizzamento da 0 e ci penserà appunto l'hardware a fare la somma.

14.1.2 Indirizzi virtuali

Questo particolare modo di intendere gli indirizzi fa prendere ad essi il nome di **indirizzi virtuali**: i programmi, in questo modo, non sanno di trovarsi a partire da un particolare indirizzo della RAM (ottenuto con la somma) o che BASE esista, "vivono" nel loro "mondo virtuale". Questo fa sì che i programmi non possano usare indirizzi assoluti (vivendo nel loro mondo senza neppure la conoscenza di BASE non sanno quali questi possano essere) e questo risolve il precedente problema di non poter usufruire del vantaggio 2 (non potendo in questo modo modificare cose in indirizzi che non siano i suoi, o meglio quelli del suo mondo). In altre parole, consideriamo che un processo P1 usa un indirizzo x , quell'indirizzo x non corrisponderà allo stesso indirizzo x che usa un processo P2, in quanto verranno sommati 2 valori di base diversi. Quindi anche se l'indirizzo virtuale x è lo stesso, corrisponderà a 2 diversi indirizzi fisici. In questo modo ogni processo ha acquisito una sua memoria virtuale: l'unica a cui ha accesso. Tutti gli indirizzi generati durante l'esecuzione di un processo sono relativi alla memoria virtuale del processo e sono quindi indirizzi virtuali.

Nota: *Riguardo però al mantenere il vantaggio 3 si può dire che viene fatto in parte: ad esempio se, guardando all'esempio mostrato nell'immagine 14.3, P3 è più grande di P1 e quindi non entra nello spazio riservato al processo. Non viene mantenuto nemmeno il vantaggio 4.*

14.1.3 Paginazione

Sviluppiamo quindi quella che è l'idea finale: **la paginazione**. I problemi visti possono essere risolti se spezzettiamo la memoria di un processo in più parti e li gestiamo separatamente. In particolare raggruppiamo gli indirizzi generabili da un processo in regioni naturali da $4KiB$ dette pagine e gestiamo ogni pagina separatamente. L'insieme di questi indirizzi è lo spazio di indirizzamento virtuale di un processo, i suoi indirizzi indirizzi virtuali. Lo spazio di indirizzamento che comprende gli indirizzi della RAM, delle periferiche, dell'APIC e altri è lo spazio di indirizzamento che comprende gli indirizzi fisici. Gli indirizzi fisici possono essere raggruppati in regioni naturali da $4KiB$ dette frame: lo scopo è sempre quello di far corrispondere ad indirizzi virtuali indirizzi fisici.

$$f(v) \rightarrow f$$

dove v è l'indirizzo virtuale. Consideriamo un indirizzo virtuale, questo cadrà all'interno di una certa pagina p ad un offset o , quindi avremo (p, o) . L'indirizzo fisico corrispondente cadrà all'interno di un frame n allo stesso offset o , avremo (n, o) . Per eseguire questa traduzione abbiamo una struttura dati che ci permette di trovare il frame corrispondente alla pagina, abbiamo una tabella di corrispondenza, ovvero un array indicizzato, chiamiamolo a che dato p mi restituisce n , con $n = a[p]$. In sostanza partendo da v che si trova ad (p, o) , troveremo f che si trova ad $(a[p], o)$.

Nota: *Poi vedremo che in realtà al posto delle tabelle di corrispondenza abbiamo alberi di traduzione.*

Nota: *Per eseguire queste traduzioni è stato introdotto un oggetto hardware che è la MMU, la vediamo nella prossima lezione.*

Tramite questo metodo si vanno a coprire con pieno successo tutti i **4 vantaggi**:

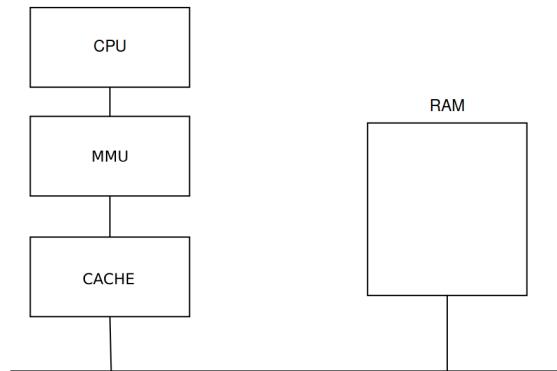
1. **Isolamento tra i processi:** basta utilizzare una tabella di traduzione diversa per ogni processo, impedendo che queste possano essere modificate da livello utente. In

questo modo ogni processo può accedere solamente alle zone di memoria fisica che si trovano nel codominio della funzione di traduzione.

2. **Semplicità nel collegamento dei programmi:** con una tabella di traduzione per ogni processo si ha l'illusione che tutti gli indirizzi siano disponibili e quindi si può assegnare liberamente gli indirizzi.
3. **Semplicità nel caricamento/scaricamento dei programmi:** Utilizzando solo indirizzi virtuali, questi non cambiano anche se un processo viene rimosso e poi ricaricato in memoria. Inoltre, ogni pagina può essere caricata ovunque ci sia un frame libero.
4. **Possibilità di condivisione della memoria:** basta inserire gli stessi numeri di frame nelle entrate delle opportune tabelle.

Nota: *si avrebbe ora anche un quinto vantaggio (quello di potersi comportare come se si avesse memoria infinita) ma non lo vediamo.*

14.1.4 Come si realizza l'hardware che fa questo (architettura intel-amd)



La MMU

La MMU, ovvero *memory management unit*, trasforma gli indirizzi da virtuali a fisici tramite una traduzione f configurabile. A partire da questo momento tutto quello che fa la CPU viene considerato virtuale.

La trasformazione fatta dalla MMU

La trasformazione fatta dalla MMU si fa byte per byte:

- Si prendono tutti gli indirizzi fisici e tutti quelli virtuali.
- Li dividiamo in, rispettivamente, **frame** e **pagine**.
- Si associa ogni pagina a un frame.

Gli indirizzi virtuali v (ovviamente a 64 bit) saranno quindi composti da 12 bit (i meno significativi) di offset della pagina e dai restanti bit indicanti il numero di pagina, mentre gli indirizzi fisici p saranno composti da 12 bit di offset del frame e i restanti per il numero di frame.

Nota: mentre il numero di pagine viene trasformato da f , l'offset rimane uguale.

Possiamo scrivere una tabella che descrive ogni processo come viene tradotto. All'interno del primo esempio di MMU che vediamo, detta **super-MMU**, sono presenti tutte le tabelle di traduzione, memorizzate nella memoria della super-MMU. Solo una tabella di traduzione sarà usata in un determinato momento, quella relativa al processo in esecuzione.

Nota: Attenzione: la MMU reale non è così, verrà vista più avanti.

Chi crea le tabelle di traduzione? **il sistema**. Caricare un programma significa quindi portare le cose nella RAM e creare le tabelle nella MMU (si ha infatti una tabella per ogni processo).

Nota: la creazione delle tabelle avviene in software, quello che c'è dentro è deciso dal software.

Chi decide quando cambiare tabella? (con il cambio di processo deve infatti avvenire il cambio di tabella su quella relativa al processo in esecuzione). Lo fa sempre **il sistema**, in particolare lo fa la **carica_stato**.

Nota: La MMU in hardware fa solo la traduzione degli indirizzi da virtuale a fisico, il resto fa tutto il software.

Esempio 10: Ad esempio si ha una tabella del tipo

$$a[0] = 2$$

$$a[1] = 0$$

$$a[2] = 4$$

.

.

.

in cui il numero nelle quadre indica il numero di pagina mentre quello a destra dell'uguale il numero di frame. \square

14.1.5 Funzioni aggiuntive:

La MMU si trova in una posizione strategica, intercetta tutti gli indirizzi generati dal processore: possiamo quindi utilizzarla per aggiungere alcune informazioni lungo il percorso di traduzione dei vari indirizzi:

- Per ogni pagina associamo un bit p : ci dice se c'è stata traduzione o no.
- Per ogni pagina associamo un bit r/w : dice se si può accedervi in modalità scrittura.
- Per ogni pagina associamo un bit u/s : dice il livello di accesso, utente o sistema.

Nello specifico il bit P serve per marcare le pagine che il processo non usa. In particolare se la MMU riceve un indirizzo che porta ad un'entrata con $P = 0$ smette di tradurre e dà il controllo al sistema. Il flag R/W serve invece per proteggere alcune sezioni, in particolare alcuni indirizzi dalla scrittura. In particolare se la MMU intercetta un'operazione di scrittura e lungo il percorso di traduzione incontra R/W=0 fa sollevare un'eccezione. Infine vediamo il flag U/S, che ci dice a quale livello sono accessibili determinati indirizzi in memoria. Nel

nostro sistema la MMU è sempre attiva, quindi se devo permettere il meccanismo delle interruzioni e delle eccezioni e di conseguenza devo permettere di poter innalzare il livello di privilegio, devo far sì che tutta la memoria M1 sia nel codominio delle funzioni di traduzione di tutti i processi. Allo stesso tempo devo vietare agli utenti di poter accedere in scrittura e di modificare questa zona di memoria, per farlo metto $U/S = \text{sistema}$ in tutta la parte alta (tutta prima del buco) dello spazio di indirizzamento. Se la MMU rileva un accesso da livello utente ad una pagina con $U/S = \text{sistema}$ allora solleva una eccezione.

14.2 Lezione 23 15/04

Sono inoltre presenti anche i flag:

- PWT
- PCD

che servono entrambi per agire sulla cache. In particolare il bit PCD se è settato ordina alla cache di lasciar passare le operazioni con destinatari quegli indirizzi. Avremo $PCD = 1$ per tutte le pagine che contengono indirizzi di registri di I/O mappati in memoria. Un esempio è l'APIC, i cui indirizzi sono appunto mappati nello spazio di memoria. Il bit PWT indica alla cache di seguire la politica *write through* per questo accesso (un esempio di utilizzo è la memoria video).

Nota: Se $PCD = 1$ PWT non ha significato.

Per ogni pagina la MMU riserva 2 bit:

- **A:** va a 1 quando si verifica un accesso su quella pagina, sia in lettura che in scrittura.
- **D:** va a 1 quando si verifica accesso in scrittura.

I bit A e D sono gli unici che vengono letti e scritti dal software e dalla MMU, gli altri vengono scritti all'inizializzazione tramite la `activate_p()` e poi solo letti dalla MMU. Questi 2 bit sono utili prevalentemente per implementare la paginazione su domanda, che noi non trattiamo. Tuttavia il bit D può essere **utile** anche nel nostro sistema: quando il sistema carica le pagine di un processo in RAM, lo fa mettendo $D = 0$, quando una pagina verrà modificata verrà messo $D = 1$. Al momento dello *swap-out* del processo (togliere le pagine del processo dalla RAM e salvarle in HD) il sistema può evitare di ricopiare sull'hard-disk tutte le pagine per cui $D = 0$, perché quelle non sono state modificate.

Capitolo 15

Tabelle multilivello

15.1 Continuo Lezione 23

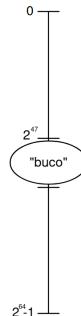
15.1.1 Irrealizzabilità della super-MMU

Questa è dovuta dalla dimensione delle tabelle. Trattandosi infatti di indirizzi a 64 bit la tabella dovrebbe contenere:

$$2^{64-12} = 2^{52} \cdot 2^3 = 2^{55}$$

indirizzi, il che è impossibile dato che 1 Tera è 2^{40} . Questo perché le tabelle sono grandi $4KiB \rightarrow 2^{12}$, quindi avremo 2^{52} pagine, con ogni entrata grande 8 byte.

I processori attuali sono però, per questi motivi, a 48 bit di indirizzamento (e non 64). Questo porta ad avere uno spazio di memoria diviso in 2 (dovuto al fatto che, tolto i 48 LSBs, i 16 più significativi seguono il comportamento del 47° bit).



In particolare noi ci muoveremo considerando la prima parte della memoria riservata al sistema mentre la seconda all’utente, andando, per comodità, in contrasto con le politiche storiche sia di Windows che di Linux. Questa suddivisione ci permette di discriminare velocemente la parte sistema da quella utente guardando il MSB. La parte di indirizzi in mezzo, dove c’è il buco, non va quindi considerata e il processore solleva un’eccezione se si prova ad accedervi. Considerando questo tipo di indirizzamento si riduce sensibilmente la dimensione richiesta delle tabelle che quindi passa da irrealistica a:

$$2^{48-12} = 2^{36} \cdot 2^3 = 2^{39}$$

che corrisponde a 512 Gib, dimensione realistica ma comunque troppo grande per essere utilizzata, visto che ce ne vorrebbe una per processo.

15.1.2 TRIE-MMU

Passiamo quindi ad analizzare la TRIE-MMU, che aggiunge alcune caratteristiche tra cui, una su tutte, l'utilizzo di una struttura ad albero al posto della singola tabella: ci sono delle chiavi per guidare la ricerca mentre le informazioni sono solo sulle foglie. In particolare utilizziamo un *bitwise-trie*, dove la chiave di ricerca all'interno dell'albero sono gruppi di bit della chiave. Il meccanismo è molto simile ai trie originali in cui i caratteri guidano la ricerca. La TRIE-MMU, come la super, ha ancora una memoria al suo interno dove memorizzare le tabelle, inoltre si ha un registro del processore, il CR3 che ci dà informazioni su quale tabella è attiva.

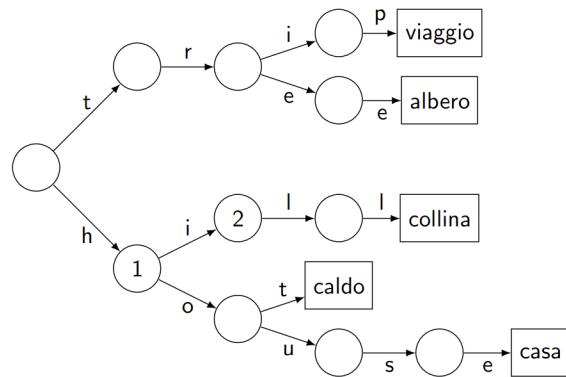
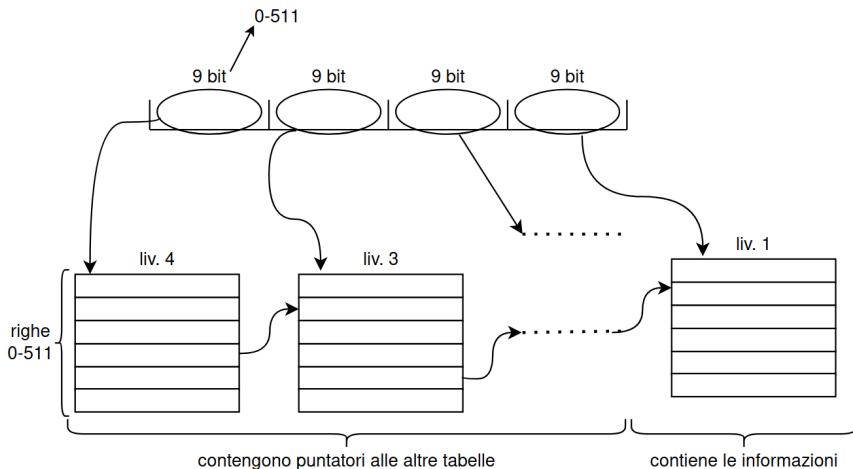


Figura 15.1: Esempio di struttura ad albero con chiavi di ricerca e informazioni sulle foglie.

Nel nostro caso la chiave di ricerca sarà il numero di pagina, che è a 36 bit, e il valore che vi vogliamo associare sarà il corrispondente numero di frame, contenuto nei nodi foglia:



Nota: Ogni tabella è grande 4KB, ogni nodo conterrà una tabella con 512 entrate con puntatori, eventualmente nulli, al nodo successivo:

$$2^9 \cdot 2^3 = 2^{12} = 4k$$

Come si vede dalla figura dividiamo il numero di pagina in 4 gruppi da 9 bit, di conseguenza l'albero sarà al massimo di 4 livelli. L'informazione è mantenuta nel nodo foglia, gli altri nodi ci danno solo informazioni su dove trovare il numero di frame.

Analisi dei livelli della TRIE-MMU

Guardiamo come sono fatte le entrate dei vari livelli. Partendo dal livello 1, le entrate delle tabelle di livello 1 sono dette descrittori di pagina virtuale o descrittori di livello 1, conterranno tutti i bit che abbiamo visto. Le entrate delle tabelle di livello 2,3 e 4 hanno tutte lo stesso formato, a differenza di quelle di livello 1 queste descrivono tabelle e non pagine. Contengono i bit visti esclusi: D, PWT, PCD e contengono il bit PS, che vedremo.

Nota: *Ogni riga contiene il numero di riga dopo i 12 bit meno significativi: questi sono relativi ai vari flag illustrati in precedenza (r/w, u/s, ...). In particolare si ha che sono significativi:*

Tutte	Liv. 1
P	PWT
R/W	PCD
U/S	D
A	

Nota: *Per ogni processo abbiamo:*

- 1 tabella di livello 4 per un totale di 4KiB
- 512 tabelle di livello 3 per un totale di 2MiB
- 256 Ki tabelle di livello 2 per un totale di 1GiB
- 128 Mi tabelle di livello 1 per un totale di 512GiB

Percorso di traduzione

Andiamo a vedere cosa deve fare la MMU e in particolare il **percorso di traduzione** (si vede anche dalla figura 16.1 della lezione successiva). Partiamo dall'indirizzo virtuale, che sarà composto da 12 bit meno significativi di offset e 36 di numero di pagina. Innanzitutto si esegue una lettura in memoria, in particolare va letto il valore del CR3 per trovare il corretto descrittore di livello 4, poi per trovare il corretto indice bisogna concatenare il valore del CR3 con i bit dell'indirizzo 47 – 39 e moltiplicare per 8, per rispettare l'allineamento naturale. Eseguiamo la stessa operazione per trovare il descrittore di livello 3, 2 e 1, ogni volta concatenando un gruppo di 9 bit dell'indirizzo virtuale. Una volta arrivati al livello 1 estraiamo il numero di frame (bit 12 – 51), che concatenato con i 12 bit dell'offset ci dà l'indirizzo fisico.

Durante il percorso di traduzione la MMU fa anche altro:

- Controlla tutti i bit R/W, permettendo un'operazione di scrittura solo se tutti e 4 la permettono.
- Controlla tutti i bit U/S e permette una determinata operazione solo se tutti la permettono.
- Controlla i bit PWT e PCD del descrittore di livello 1 e eventualmente interagisce con la cache.

- Pone ad 1 tutti i bit A che incontra e se l'operazione è di scrittura setta il bit D nell'entrata di livello 1.
- Analizza anche il bit PS (lo vedremo).

Analisi sulle dimensioni della struttura ad albero

Analizziamo adesso le dimensioni di un TRIE di questo tipo. Se tutto l'albero fosse allocato si occuperebbe più spazio della SUPER-MMU, infatti la sola tabella di livello 1 ha la stessa dimensione della SUPER-MMU. Quindi sembra che abbiammo peggiorato il problema. Anche in termini di tempo prima serviva un solo accesso in memoria, mentre ora 4. Questa struttura ad albero ci permette però di avere alcuni vantaggi:

- La struttura ad albero funziona anche se non completamente allocata (la tabella MMU funzionava invece solo se tutta allocata).
- Posso "tagliare" alcuni rami senza influire sul funzionamento; in particolare questo permette di risparmiare molto spazio (basti pensare quanto spazio si risparmia tagliando un ramo della tabella di liv. 4). I processi non useranno quasi mai tutta gli indirizzi a loro disposizione.
- Si può condividere memoria, basta condividere un sottoalbero.
- C'è un altro vantaggio che non vediamo.

Capitolo 16

Paginazione: complementi

16.1 Lezione 24 18/04

Arriviamo alla vera MMU che come abbiamo visto non ha una memoria interna. Bisogna quindi spostare le informazioni in RAM. Abbiamo all'interno del descrittore di un processo un registro CR3: questo contiene l'indice della radice dell'albero di un processo. Quindi quando un processo è in esecuzione il registro CR3 del processore conterrà il valore del CR3 del descrittore di processo. Questo valore è passato dalla CPU alla MMU insieme all'indirizzo da tradurre. Da esso inizia il processo di prelievo da RAM o cache delle entrate della tabella di livello 4, poi della tabella di livello 3 ecc. Questo procedimento è detto *table walk*.

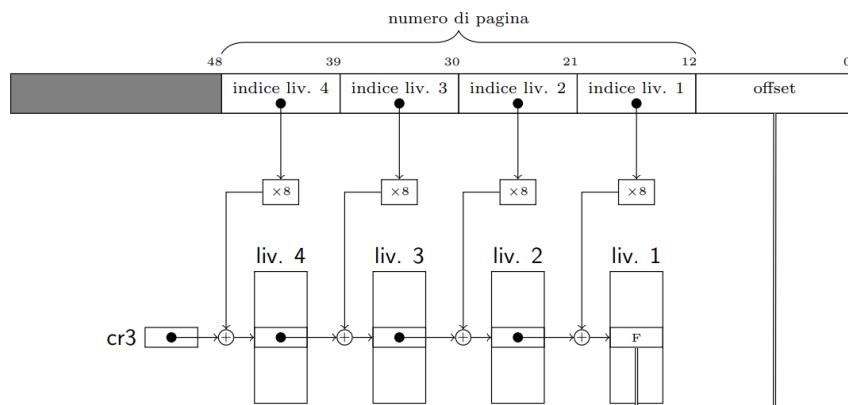


Figura 16.1: Il primo indirizzo sarà quindi dato dalla somma di quello contenuto in CR3 sommato ai 9 bit dell'indice di liv.4, da questo si ottiene l'indirizzo della tabella di livello 3 a cui va sommato l'indice di liv.3 ecc.

Si nota dunque che, considerando le operazioni da fare eventualmente sul bit A si hanno, nel caso peggiore 8 accessi in memoria per ogni indirizzo richiesto. Di conseguenza ora dobbiamo pensare a come gestiamo la memoria e come rendere efficiente questo meccanismo, visto che come abbiamo appena visto, a primo impatto, sembra abbiano quasi peggiorato le cose.

16.1.1 Come gestire la memoria

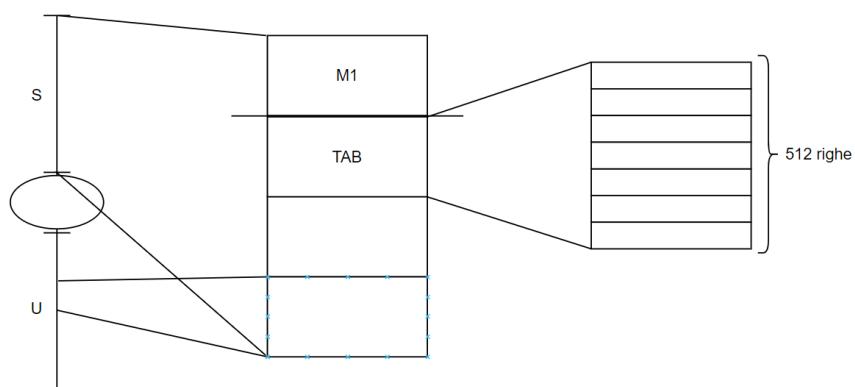
Come abbiamo già visto abbiamo diviso la memoria fisica in 2 sezioni: M1 per il sistema e M2 per gli utenti. Logicamente le tabelle dovrebbero essere memorizzate in M1, in quanto sono inaccessibili e non modificabili dagli utenti. In realtà, per motivi di comodità, le memorizziamo in M2 in quanto grandi $4KiB$ come le pagine, in questo modo un frame di M2 può contenere o una pagina o una tabella senza dover decidere a priori quale spazio di memoria riservare alle une e alle altre. Ciò comporta che il sistema debba poter accedere anche a tutta M2 per poter consultare/modificare le tabelle di traduzione, e deve ovviamente anche poter accedere a tutta M1: deve poter quindi accedere a tutta la memoria fisica. La soluzione migliore sarebbe dunque avere la MMU disattivata quando ci troviamo a livello sistema: questo però nell'architettura intel/AMD non è possibile, la MMU resta infatti sempre attiva.

La soluzione che allora si adotta è creare delle traduzioni identità (1:1), ovvero tradurre gli indirizzi virtuali in sé stessi, per tutti gli indirizzi della memoria fisica e inserire queste traduzioni nello spazio di indirizzamento dei processi. In particolare l'inserimento avviene nella parte alta dello spazio di indirizzamento dei processi con la protezione delle traduzioni tramite il bit $U/S = \text{sistema}$ (in questo modo gli utenti non possono accedervi).

Questo corrisponde all'avere creato, nella parte alta dello spazio di indirizzamento dei processi, una **finestra sulla memoria fisica**. Questa finestra viene creata **prima** di attivare la memoria virtuale. In particolare all'avvio del sistema si parte con la memoria virtuale disattivata, vengono poi allocate tutte le tabelle necessarie alla definizione della finestra e infine viene attivata la memoria virtuale. A questo punto la routine di inizializzazione può continuare ad usare indirizzi fisici come prima dell'attivazione della memoria virtuale. Quando poi si passa a livello utente queste traduzioni diventano inaccessibili, ritornano invece accessibili ogni volta che si passa a livello sistema, ed è come se la MMU "si attivasse".

Nota : Importante: *Tutto quello che fa a livello di traduzione la MMU (ovvero l'uso delle tabelle) è invisibile lato software sia per il sistema che per l'utente. Il sistema ha però la possibilità di accedere in lettura e in scrittura alle tabelle.*

Nota: *Nel passaggio da livello utente a livello sistema l'albero usato per la traduzione attivo in quel momento non cambia.*



Dall'immagine si vede come dallo spazio degli indirizzi di quel processo, da quelli del sistema si ha la traduzione per tutta la RAM in modo da poterne accedere nella sua totalità. Ogni riga infatti, se valida, contiene un indirizzo **fisico** (in quanto serve alla MMU).

Nota: Ogni processo all'inizio del suo tree a livello sistema ha una traduzione identità.

Nota: Gli indirizzi che si trovano lungo l'albero di traduzione (nelle tabelle) servono alla MMU per trovare la traduzione, quindi sono indirizzi fisici.

16.1.2 TLB

Vediamo ora di rendere questo meccanismo efficiente in termini di tempo. Quello che si fa è utilizzare una cache esclusiva per la MMU, il **TLB** che ha lo scopo di ricordare le traduzioni (dato il numero di pagina) più recenti. In particolare ricorda solo il descrittore di livello 1, quelli precedenti servono solo per il percorso di traduzione. La cache TLB è così fatta:

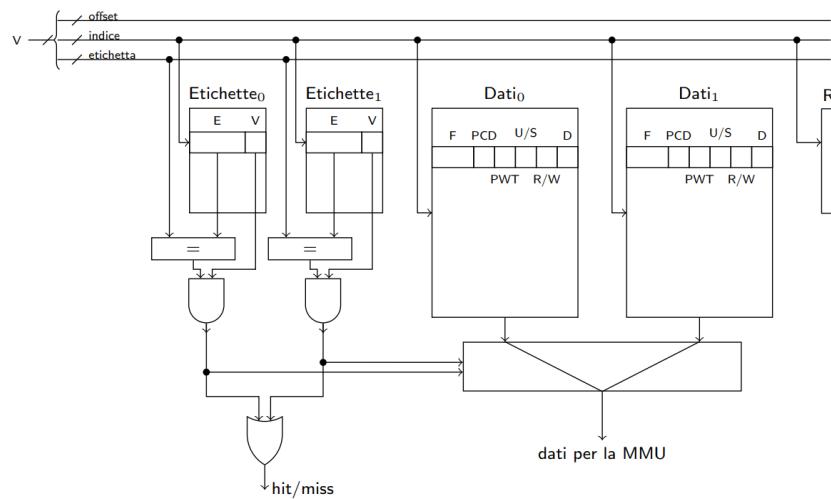


Figura 16.2: Notare quali flag sono presenti (mancano A e P)

Senza questa cache la memoria virtuale non sarebbe utilizzabile.

Funzionamento della TLB

Quando la MMU deve eseguire una traduzione di un indirizzo prima guarda se il descrittore cercato sta nel TLB, se non c'è deve eseguire il *table walk*, fa poi una copia del descrittore di livello 1 e la inserisce nel TLB, eventualmente rimpiazzando un altro descrittore, scelto tramite un algoritmo pseudo-LRU.

Il TLB come tutte le cache è **trasparente al software**: non si hanno istruzioni che ci permettono di vedere cosa c'è dentro.

Le uniche istruzioni che si hanno sono per invalidarlo tutto o in parte. Queste operazioni sono necessarie quando viene cambiato qualcosa nel percorso di traduzione, quindi il TLB non contiene più informazioni valide. Il TLB può essere invalidato tutto cambiando il CR3, infatti quando si cambia processo in esecuzione, e di conseguenza si ha un cambio del valore del CR3, si ha lo svuotamento automatico del TLB. Questo ci va bene perché il TLB contiene le informazioni sulle traduzioni del processo in esecuzione.

Il TLB deve permettere alla MMU di svolgere tutte le funzioni senza dover fare il *table-walk*, quindi a livello di bit abbiamo che:

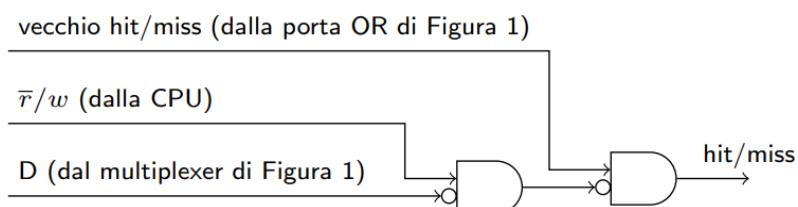
- Bit P non memorizzato nel TLB, se la traduzione c'è vuol dire che esiste.

- Dei bit R/W e U/S viene memorizzato l'AND dei 4 bit delle tabelle.
- Bit PWT e PCD vengono memorizzati e eventualmente passati al controllore cache.
- Bit A non memorizzato, se c'è la traduzione vuol dire che un accesso a quella pagina c'è stato.
- Bit D memorizzato (spiegazione dopo).

Per quanto riguarda il bit A la soluzione è quella di aggiornare l'albero solo se si ha una miss nel TLB. Il problema sussiste quando il software per qualche motivo decide di azzerarli nelle tabelle. In quel caso la soluzione deve essere in software e quando vengono azzerati i bit A allora bisogna invalidare in parte o del tutto il TLB.

Per il bit D il discorso è più complicato, può succedere che ci sia un primo accesso in lettura, a seguito del quale la MMU non porta D a 1 e la traduzione viene caricata nel TLB. Poi c'è un nuovo accesso in scrittura, la traduzione viene trovata all'interno del TLB e, senza provvedimenti, non si accederebbe all'albero e si lascerebbe $D = 0$ pur essendovi stato un accesso in scrittura, rendendo così inaffidabile l'informazione offerta dal bit D. La soluzione che si adotta è di tipo hardware: il TLB ricorda anche il valore del bit D al momento del caricamento della traduzione e causa una miss per gli accessi in scrittura con D=0. In questo modo si forza la MMU a percorrere l'albero aggiornando di conseguenza il bit D. A quel punto la MMU ricaricherà la traduzione nel TLB, questa volta con il bit D a 1 e i successivi accessi in scrittura non causeranno dunque ulteriori miss.

Nota: Cambiare D solo nel TLB non avrebbe senso: *in primo luogo perché a livello software non ho istruzioni per leggere cosa c'è dentro il TLB e in secondo luogo anche se avessi l'opportunità di leggerne il contenuto avrebbe senso solo se il TLB si comportasse come cache write-back e ricopiasse nella tabella opportuna il contenuto del bit D prima di rimpiazzare un'entrata, ma per fare ciò si dovrebbe fare il table walk anche nel TLB, perché la tabella andrebbe ritrovata a partire dall'unica informazione di cui il TLB dispone, che è il numero di pagina.*



Nota: Se abilitiamo la scrittura (facciamo un accesso in scrittura) e prima era disabilitata, il TLB causerà sicuramente miss sull'operazione in scrittura e quindi non c'è bisogno di invalidare l'entrata, questa verrà sovrascritta dalla MMU. Se invece disattiviamo la scrittura ed era abilitata bisogna invalidare l'entrata del TLB.

16.2 Lezione 25 19/04

Per ottimizzare gli accessi in memoria, riducendo i tempi di *table walk* e occupare meno spazio in memoria fisica per i TRIE, vogliamo pagine più grandi, rispetto a quelle di $4KiB$. Per realizzarle dobbiamo aumentare i bit di offset:

- Fino ad ora abbiamo visto 48 bit di numero di pagina e 12 di offset, quindi pagine grandi $4KiB$.
- Si è passati poi ad avere anche 27 bit per numero di pagina e 21 di offset (bit da 0 a 20), quindi pagine grandi $2MiB$.
- Nei processori più moderni si ha anche 18 per numero di pagina e 30 di offset (30 bit significa avere pagine grandi $1Gb$). In quel caso $PS = 1$ nei descrittori di livello 3.

Per implementare questo meccanismo utilizziamo il già accennato **bit PS** nelle entrate delle tabelle. In particolare nei descrittori di livello 2 e, nei processori moderni anche in quelli di livello 3, si può inserire questo bit.

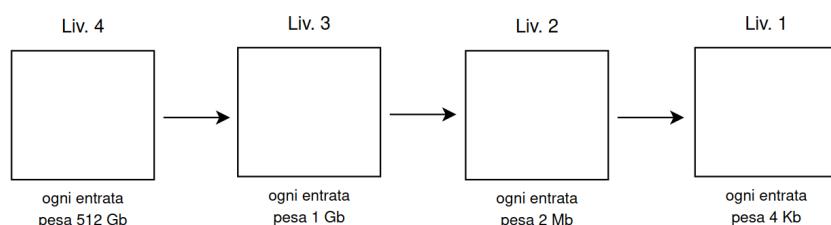
Il meccanismo è il seguente: la MMU inizia a tradurre e percorre l'albero di traduzione, se quando arriva al descrittore di livello 2 (o 3) trova il bit $PS = 1$ vuol dire che l'albero è finito e che il numero di frame è contenuto nella tabella di livello 2, in questo modo possiamo evitare di allocare 512 tabelle di livello 1. Facendo così il numero di offset sarà di 21 bit (o 30) e il descrittore di livello 2 nel caso deve memorizzare anche i bit D, PWT e PCD.

Il TLB per pagine di $4KB$ non funziona però per pagine più grandi: si usano quindi 3 TLB, uno per ogni dimensione di pagina possibili (3 dimensioni specificate sopra). La MMU deve cercare la traduzione in ognuno dei 3 TLB, perché ogni indirizzo può essere tradotto usando pagine di qualunque dimensione e non è possibile saperlo in anticipo.

Nota: *Con la presenza di 3 TLB la ricerca avviene in tutti e 3, senza un ordine particolare, ma in parallelo.*

La struttura a 3 TLB è anche un incentivo a usare pagine più grandi, le quali hanno una maggiore influenza sulle prestazioni anche della cache. Se una pagina di memoria è più grande, quando i dati vengono letti dalla memoria, una porzione maggiore di dati viene caricata nella cache. Ciò significa che è più probabile che future richieste di dati nella stessa pagina siano servite dalla cache, minore sarà il numero di miss, riducendo i tempi di accesso alla memoria. Così si sfruttano i TLB aggiuntivi e si alleggerisce la pressione sul TLB principale.

Nota: *Uno svantaggio dell'utilizzo di pagine più grandi è che si può avere un maggiore spreco di memoria all'interno delle pagine stesse.*



Nota:

- Quando si usa il debugger con la funzione `vm ...` per distinguere i megabyte degli indirizzi in esadecimale basta guardare la prima cifra dopo i primi 5 zeri meno significativi.
- Nella nostra macchina virtuale, a $32Mb$ di RAM, se andiamo a vedere i flag delle tabelle di livello 2 vediamo come il bootloader dopo aver finito tutta la RAM utilizzi gli indirizzi rimanenti per periferiche ecc. (si vede perché sono attivi di bit PWT e PCD).

Capitolo 17

Funzioni di supporto per la paginazione

17.1 Lezione 26 22/04

Esempi.

17.1.1 Funzioni di supporto per la paginazione

Vediamo alcune funzioni utili per la gestione della memoria virtuale e della paginazione. Le funzioni più importanti per il pratico sono spiegate anche nel file dedicato al pratico. Riportiamo solo una piccola spiegazione, gli esempi si trovano nella dispensa del prof.

Tipi vaddr e paddr

La libreria *libce* definisce 2 nuovi tipi: **vaddr** e **paddr**, che vengono utilizzati per rappresentare indirizzi virtuali e indirizzi fisici. Entrambi i tipi sono equivalenti ad interi senza segno su 64 bit.

Funzione vaddr norm(vaddr)

Serve a normalizzare un indirizzo virtuale, rende i 16 bit più significativi uguali al bit 47. Viene usata anche se un indirizzo virtuale è normalizzato. Un indirizzo virtuale per poter essere utilizzato deve essere normalizzato. Funzione non molto usata.

Funzione natq dim_region(int liv)

Restituisce la dimensione in byte di una regione di livello liv. Non molto usata.

Funzioni vaddr base(vaddr v, int liv) e vaddr limit(vaddr e, int liv)

Restituiscono rispettivamente la base della regione di livello liv in cui cade un indirizzo v e la base della prima regione di livello liv che si trova a destra di un intervallo [b,e). Quasi mai usate, al massimo la base.

Viene aggiunto anche il tipo tab-entry

Rappresenta un'entrata di una tabella di qualunque livello. Se lo utilizziamo per avere un riferimento ad un'entrata della tabella si può utilizzare per vari scopi, ad esempio esaminare i bit dell'entrata, resettarli o settarli.

Funzioni int i_tab(vaddr v, int liv)

Estrae dall'indirizzo virtuale v l'indice che la MMU usa per consultare le tabelle di livello liv. La funzione **tab_entry get_entry(paddr t, int i)** dato l'indice fisico di una tabella e un indice i restituisce un riferimento all'entrata i-esima della tabella stessa.

Funzioni copy_des() e set_des()

Permettono di copiare una o più entrate da una tabella ad un'altra e di sovrascrivere una o più entrate di una tabella con uno stesso valore.

Funzioni readCR3()/loadCR3() e readCR2()

Permettono di leggere i registri cr3 e cr2 e di scrivere nel registro cr3. loadCR3() può essere usata per attivare un nuovo albero di traduzione, gli viene passato l'indirizzo fisico della tabella radice -> viene invalidato tutto il TLB.

Funzione invalida_entrata_TLB(vaddr v)

Serve ad invalidare la traduzione associata all'indirizzo virtuale v, nel caso il TLB ne stesse conservando una copia. Va usata ogni volta che si cambia qualcosa nel percorso di traduzione di v. Invece **invalida_TLB()** invalida tutto il contenuto del TLB, conviene chiamarla se sono stati fatti tanti cambiamenti.

Iteratore tab_iter

Viene introdotto anche l'iteratore **tab_iter** che permette di visitare tutte le entrate dell'albero di traduzione coinvolte, a tutti i livelli, nella traduzione di tutti gli indirizzi di un dato intervallo. La visita può essere fatta sia in ordine anticipato, come fa la *map* per la creazione di alberi/sottoalberi, o posticipato, come fa la *unmap* per la distruzione di alberi/sottoalberi.

L'operatore può essere spostato con il metodo *next()*. Le funzioni membro **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 di questa tabella.

La visita in ordine anticipato è utile quando vogliamo costruire l'albero di traduzione per un certo intervallo di indirizzi. Ogni volta che l'iteratore si ferma si esamina il bit P e se è a 0 e non siamo ancora arrivati al livello 1 si può allocare e agganciare una nuova tabella di livello inferiore.

La visita in ordine posticipato è utile quando vogliamo distruggere un albero di traduzione, in quanto ci permette di eliminare i livelli inferiori dell'albero prima di esaminare i livelli superiori.

trasforma(paddr root, vaddr v)

Permette di tradurre l'indirizzo virtuale *v* nel corrispondente indirizzo fisico in base al TRIE con radice *root*. La funzione usa internamente un tab-iter per visitare il TRIE in software come farebbe la MMU in hardware. Se non si esplicita il primo parametro prende implicitamente `esecuzione->cr3`. Funzione **importante**.

Funzioni map e unmap

Le funzioni **più importanti** sono **map** e **unmap**. La funzione *vaddr map()* riceve in ingresso l'indirizzo fisico della tabella radice di un trie, gli estremi di un intervallo $[begin, end]$ di indirizzi virtuali, gli eventuali bit da settare nelle entrate e un parametro *getpaddr* che si comporta come una funzione da *vaddr* a *paddr*, solitamente si usa un funtore per questo. La funzione *map* utilizza il metodo della visita anticipata per allocare le traduzioni per gli indirizzi specificati nell'intervallo. Se va a buon fine e ha creato tutte le traduzioni desiderate restituisce il prossimo indirizzo virtuale di cui doveva allocare la traduzione (*end*). Solitamente dopo la chiamata di una *map* si fa un controllo con cui si verifica che sia andata a buon fine, in caso contrario si fa una *unmap* da *begin* fino a dove siamo arrivati e si dà un messaggio di errore. La funzione *unmap()* esegue l'operazione inversa di *map*: distrugge tutte le traduzioni in un dato intervallo di indirizzi virtuali. La funzione si preoccupa di deallocare anche tutte le tabelle che diventano vuote dopo aver eliminato le traduzioni dell'intervallo. Viene passato un parametro *putpaddr* che l'utente può usare per decidere cosa fare di ogni indirizzo fisico che prima era mappato da qualche indirizzo virtuale, anche per questo si può utilizzare un funtore.

Capitolo 18

Implementazione della memoria virtuale

18.1 Lezione 28 29/04

18.1.1 Implementazione della memoria virtuale nel nucleo

L'ultima cosa che rimane da vedere sulla memoria virtuale è l'implementazione nel nostro sistema. Abbiamo un sistema in cui i processi possono condividere zone di memoria e hanno anche zone di memoria private. Inoltre abbiamo visto che la memoria virtuale di un processo si divide in 2 sezioni: la prima parte dall'indirizzo 0 all'indirizzo 0000 7FFF FFFF FFFF (2 alla 47 -1) è riservata al sistema, la seconda parte dall'indirizzo FFFF 8000 0000 0000 all'ultimo (tutte F) è riservata agli utenti. Queste parti sono divise in ulteriori sezioni, ovvero:

- **Sistema condivisa:** contiene la finestra sulla memoria fisica, la sua dimensione è decisa a priori mentre la dimensione della finestra dipende dalla memoria fisica installata.
- **Sistema privata:** contiene la pila sistema privata del processo. Questa pila è utilizzata sia dalle funzioni di sistema sia da quelle del modulo I/O.
- **I/O condiviso:** il modulo I/O (che vedremo dalle prossime lezioni).
- **Utente condivisa:** contiene il modulo utente, le sezioni `.text` e `.data` e l'`heap` utente.
- **Utente privata:** contiene la pila utente privata del processo.

Nota: *Le sezioni condivise contengono le stesse traduzioni in tutti i processi, le sezioni private no. Ciò vuol dire che nelle sezioni condivise gli stessi indirizzi virtuali sono tradotti negli stessi indirizzi fisici, mentre nelle sezioni private gli stessi indirizzi virtuali sono mappati in indirizzi fisici diversi. Si usano traduzioni diverse in ciascun processo e corrispondono a diverse parti della memoria fisica.*

Come abbiamo già visto la memoria è divisa in 2 parti M1 (sistema) e M2 (utente). M2 non ha un ordinamento specifico: le tabelle e le pagine vengono allocate dove ci sono frame liberi. La M1 invece ha un ordinamento: 1 MB viene riservato per ragioni storiche, poi si ha l'`heap` di sistema, dove vengono allocate tutte le strutture come i `des_proc` o richieste che

hanno bisogno di essere allocate dinamicamente. Infine si ha la parte dedicata al modulo sistema (.text e .data).

Per quanto riguarda la suddivisione dello spazio di indirizzamento dei processi nelle sue varie parti è stabilita da costanti in *costanti.h* (per maggiori dettagli guardare la dispensa). Una tabella di livello 4 ha 512 entrate, 256 riservate al modulo sistema e 256 riservate al modulo utente, di queste 128 per la parte utente condivisa e 128 per la parte utente privata.

Parlando ora della gestione della memoria fisica, il sistema deve sapere quali frame della memoria fisica sono liberi per poterci allocare le tabelle o le pagine dei processi, inoltre per ogni frame si associano anche altre informazioni. Ogni frame è equivalente, quindi basta avere una lista di frame e fare inserimento e estrazione dalla testa. In particolare viene utilizzato un descrittore di frame per ogni frame della memoria fisica, contenente il numero del prossimo frame nella lista dei frame liberi e quante entrate con P diverso da 0 si hanno se nel frame è stata allocata una tabella.

```

1 struct des_frame{
2     union{
3         natw nvalide;
4         natl prossimo_libero;
5     };
6 };

```

I descrittori sono raccolti in un array, *vdf[]*, indicizzato dal numero di frame. Questo array viene inizializzato in fase di inizializzazione con tutti i frame di M2 nella lista dei frame liberi. Si può allocare un frame tramite la funzione `paddr alloca_frame()` e dealloarlo tramite la funzione `void rilascia_frame(paddr p)`. Infine si può manipolare il contatore *nvalide* di una tabella di cui conosciamo l'indirizzo fisico tramite le funzioni `inc_ref()`, `des_ref()` e `get_ref()`.

18.1.2 Creazione delle parti condivise

Tutte le traduzioni relative alle parti condivise dei processi possono essere create una sola volta all'avvio del sistema, i processi possono condividere tutto il sottoalbero che implementa queste traduzioni a partire dal livello inferiore rispetto al livello della radice: il livello 3.

Nota: *Ogni processo ha la sua tabella di livello 4, altrimenti non potrei avere le parti private.*

Ad esempio la finestra di memoria viene creata una sola volta dal boot-loader tramite la funzione `crea_finestra_FM()`, prima di abilitare la paginazione. Il modulo sistema ritrova il TRIE allocato dal boot-loader tramite l'indirizzo contenuto in CR3, il boot-loader mappa poi tutti gli indirizzi da *DIM_PAGINA* a *MEM_TOT* in sé stessi, creando la finestra sulla memoria fisica. Le traduzioni della finestra hanno il bit *R/W* = 1 e *U/S* = 0.

Alcuni dettagli sono che: nella traduzione degli indirizzi della memoria video viene anche settato *PWT* = 1 e che nelle traduzioni che permettono di accedere ai registri dell'APIC si ha *PCD*=1. Le traduzioni delle parti I/O e utente/condivisa sono create invocando la funzione `carica_modulo()`, invocata una volta per il modulo I/O e una per il modulo utente. Tutte queste traduzioni vengono create nello stesso albero di traduzione che aveva inizialmente creato il boot-loader e la cui radice si trova ancora in cr3. Abbiamo visto che la creazione dei processi avviene tramite la `crea_processo()`, questa oltre ad allocare un nuovo descrittore e a svolgere le operazioni viste deve anche creare la memoria virtuale del processo. Per farlo alloca una nuova tabella radice e ci copia tutte le entrate relative alle parti condivise, prendendole dalla tabella radice puntata dal cr3 (probabilmente quella del

processo genitore). In questo modo le entrate delle parti condivise punteranno tutte allo stesso sottoalbero. Poi bisogna creare le parti private, ovvero la pila sistema e la pila utente, che sono create allocando un numero prestabilito di frame mappandoli contiguamente. La funzione deve anche inizializzare la pila sistema con le 5 parole quadruple in modo tale che il processo possa andare in esecuzione a seguito di una *iretq*. La funzione non può scrivere le 5 parole utilizzando gli indirizzi della parte sistema/privata, in quanto questi saranno tradotti usando lo spazio di indirizzamento del processo genitore, quindi eventualmente si sovrascriverebbe la pila sistema del processo genitore. Per scrivere nella pila sistema del processo da creare si utilizza la finestra sulla memoria fisica, accedendo direttamente ai frame della pila tramite il loro indirizzo fisico.

Quando lavoriamo con le tabelle, le allochiamo, le deallochiamo o le modifichiamo bisogna **sempre** mantenere in uno stato **consistente** il **campo delle entrate libere**, il quale va aggiornato ogni volta che si cambia un bit *P* all'interno di una tabella. Di seguito altre funzioni di supporto per la memoria virtuale:

- **paddr alloc_a_tab()**
Alloc a frame destinato a contenere una tabella, azzera sia il frame che il contatore *nvalide*.
- **void rilascia_tab()**
Rilascia un frame che conteneva una tabella, controlla che il contatore *nvalide* non sia diverso da 0.
- **void set_entry(paddr tab, natl j, tab_entry se)**
Il contrario della `get_entry()`, setta l'entrata *j*-esima della tabella di indirizzo fisico *tab* con il nuovo valore *se*; aggiorna opportunamente il contatore *nvalide* se il bit *P* cambia.

Capitolo 19

Il bus PCI

19.1 Lezione 29 30/04

19.1.1 Modulo I/O

Bus PCI

Per introdurre il problema facciamo un passo indietro a livello storico. Il PC AT era un pc espandibile (come lo sono i pc di ora), ovvero avevano un bus di espansione con degli slot in cui si possono inserire delle schede di espansione che aggiungevano funzionalità al pc (scheda video, di rete, etc). Il problema era che allora non esisteva uno standard definito: ogni scheda riconosceva gli indirizzi dei propri registri. Quindi se un'azienda *A* produceva una scheda che riconosceva un certo range di indirizzi e un'azienda *B* ne produceva un'altra che riconosceva lo stesso range, queste 2 schede non potevano coesistere contemporaneamente sul pc. Inoltre non c'era un modo affidabile per il software per vedere se una determinata scheda fosse presente o no. Questo standard si chiamava ISA.

Nel 1992 venne introdotto dall'Intel lo standard che tutt'oggi è presente, lo **standard PCI**. Questo stabilisce i collegamenti del bus e il protocollo che le schede devono seguire. In particolare definisce **3 spazi di indirizzamento**: lo spazio di memoria, lo spazio di I/O e un nuovo spazio detto **spazio di configurazione**, il quale veniva utilizzato in fase di configurazione delle schede per far sì che si possano utilizzare correttamente a regime quando si utilizzano gli altri 2 spazi. I registri delle periferiche non devono avere indirizzi sovrapposti, per fare ciò le periferiche che rispettano lo standard devono avere dei comparatori programmabili attraverso i quali il software assegna gli indirizzi; lo spazio di configurazione viene quindi utilizzato dal software per programmare questi comparatori. Di conseguenza ora gli indirizzi dei vari registri delle periferiche vengono assegnati via software e quindi si evita sovrapposizioni. Un'altra cosa che aggiunge lo standard è che ora abbiamo un dispositivo detto **ponte ospite-PCI** tra il bus locale e il nuovo bus detto *bus-pci*. Inoltre si possono avere fino a 256 bus pci collegati tramite ponti, su ogni bus poi possono essere collegati 32 dispositivi e ogni dispositivo può svolgere fino ad 8 funzioni.

Operazioni di lettura e scrittura

Sul bus PCI possono transitare operazioni di lettura o scrittura in uno dei tre spazi. Un'operazione sul bus pci è detta **transazione**, ogni transazione ha un **iniziatore**, colui che inizia la transazione, e un **obiettivo**, dispositivo su cui si opera. Ogni dispositivo può comportarsi, in transazioni diverse, o da iniziatore o da dispositivo. Questo è ciò che permette

il DMA, che vedremo. Per ora facciamo conto che l'iniziatore è sempre il ponte ospite-pci che traduce le operazioni iniziate dalla CPU sul bus principale.

Ogni transazione è composta da una fase di indirizzamento e una o più fasi di scambio dati. Durante la fase di indirizzamento l'iniziatore specifica tipo dell'operazione e indirizzo, se un dispositivo, al massimo uno, collegato riconosce l'indirizzo e il tipo diventa obiettivo della transazione. Se nessun dispositivo risponde entro un certo numero di clock la transazione termina e viene dato un messaggio di errore. I principali collegamenti sono:

- **FRAME**: delimita l'inizio e la fine di una transazione.
- **DEVSEL**: viene attivato dal dispositivo che diventa obiettivo della transazione, se nessuno lo attiva in tempo succede quello visto prima.
- **C/BE[3:0]**: C usato per codificare il tipo dell'operazione e BE usato come byte enabler.
- **AD[31:0]**: per codificare prima l'indirizzo e poi per trasportare i dati.
- **IRDY e TRDY**: usati per implementare l'handshake tra iniziatore e obiettivo nella fase di scambio dati.
- **STOP**: serve a terminare prematuramente un'operazione.
- **CLK**: segnale di clock, i dispositivi campionano i dati sul fronte di salita. Non è lo stesso clock della CPU.

Tutti questi segnali sono attivi bassi tranne CLK e AD.

Vediamo il protocollo La fase di indirizzamento è uguale sia per le operazioni di lettura che per quelle di scrittura: l'iniziatore pilota C e AD con il tipo dell'operazione e con l'indirizzo del primo byte da leggere o da scrivere, poi attiva FRAME, iniziando una transazione. Se uno dei dispositivi collegati riconosce il tipo e l'indirizzo attiva DEVSEL. A questo punto si passa alle fasi di dati che sono diverse a seconda del tipo dell'operazione.

Operazioni di lettura L'obiettivo pilota le linee AD (dati), segnala la loro validità attivando TRDY e attende che IRDY sia attivo. L'iniziatore che deve leggere i dati dall'obiettivo attende che TRDY sia attivo (ovvero che i dati siano validi), li campiona e quando è pronto attiva IRDY. A questo punto con IRDY e TRDY attivi contemporaneamente sullo stesso fronte di salita del clock si è conclusa una fase dati.

Operazioni di scrittura L'iniziatore pilota BE e le linee AD con i dati che deve scrivere nell'obiettivo e attiva IRDY. L'obiettivo attiva TRDY quando è pronto a ricevere e quando vede IRDY attivo può campionare i dati.

Per entrambi i tipi di transazione è l'iniziatore che decide quante fasi di scambio dati eseguire, se quando IRDY è attivo lo è anche FRAME si prosegue con una nuova fase dati, al contrario se FRAME è disattivo quando viene campionato IRDY vuol dire che ci sarà l'ultima fase di trasferimento dati. Ovviamente come già detto l'obiettivo può fermare prematuramente la transazione con STOP.

Nota : importante: *La CPU non si accorge del bus PCI, questo è trasparente alla CPU.*

Funzionamento effettivo del ponte ospite-pci e delle transazioni

Abbiamo appena considerato il caso in cui è sempre il ponte ospite-pci a fare da iniziatore per le transazioni, tuttavia non le inizia di sua spontanea volontà. Il ponte infatti non fa altro che tradurre le operazioni iniziate dalla CPU (o dal controllore cache) sul bus principale. Il ponte risponde a tutte le operazioni nello spazio di I/O e ad alcune nello spazio di memoria. In particolare il ponte ha alcuni registri (che vedremo) nello spazio di I/O del processore che possono essere usati per comandarlo. Quindi il ponte traduce tutte le operazioni ricevute in analoghe operazioni sul bus pci, dialogando con l'obiettivo per conto del processore o del controllore cache. La **traduzione è trasparente**: le operazioni nello spazio di I/O del processore si traducono in transazioni nello spazio di I/O PCI, allo stesso indirizzo, uguale per lo spazio di memoria. Dal punto di vista della CPU (quindi anche del software) è come se i dispositivi che saranno obiettivo di queste transazioni si trovassero sul bus locale. Questi dispositivi, per far funzionare questo meccanismo, devono essere però prima configurati.

Nelle **transazioni nello spazio di configurazione** si può accedere ai vari registri di configurazione dei dispositivi, per configurarli. In questi casi l'iniziatore è sempre il ponte ospite-pci. Ogni dispositivo che è collegato al bus pci, diverso dal ponte, possiede un ingresso detto **IDSEL**, il ponte ha un collegamento per ognuno di questi ingressi. Per accedere ad uno di questi registri il ponte deve attivare uno di questi collegamenti, attivando IDSEL, e successivamente passare: il numero della funzione, l'offset e i byte enable. Per il resto le transazioni nello spazio di configurazione sono molto simili a quelle di lettura e scrittura, con la differenza che oltre ad attivare FRAME il ponte deve anche attivare l'IDSEL appropriato. Se l'IDSEL è effettivamente collegato ad un dispositivo, questo deve attivare il DEVSEL, se non viene attivato vuol dire che quello slot è vuoto.

Abbiamo detto che l'iniziatore è sempre il ponte ospite, tuttavia anche per le transazioni nello spazio di configurazione non inizia le transazioni di sua spontanea volontà ma solo per tradurre le operazioni iniziate dalla CPU. Infatti la CPU non dispone di istruzioni che permettono di interagire con lo spazio di configurazione, utilizza il ponte ospite come tramite. Infatti il ponte ha 2 registri nello spazio di I/O della CPU tramite i quali la CPU accede indirettamente allo spazio. Questi registri sono:

- **CAP**: permette di selezionare una funzione e l'offset della parola a cui si vuole accedere.
- **CDP**: permette di accedere alla parola selezionata tramite CAP, è qua che si legge o si scrive tramite istruzioni IN e OUT.

In particolare se si vuole accedere ad una determinata funzione di un dispositivo bisogna scrivere in CAP il numero del bus, il numero del dispositivo, il numero della funzione del dispositivo a cui si vuole accedere e l'offset della parola. Una volta impostato CAP il ponte ospite tradurrà tutte le letture e le scritture in CDP in corrispondenti transazioni nello spazio di configurazione. Il ponte attiva l'uscita IDSEL, che corrisponde al numero del dispositivo, copia il numero di funzione e l'offset su AD e usa i byte enable. Nel caso in cui l'operazione in CDP sia di lettura, si può sfruttare per vedere un'altra cosa. Se nessun dispositivo risponde alla transazione di lettura nello spazio di configurazione (nessuno attiva DEVSEL) il ponte restituisce un valore di tutti 1 al processore. In questo modo il software può capire se un dispositivo è presente nella posizione selezionata. La lettura di un valore diverso da 0xFFFF indica la presenza di un dispositivo.

Abbiamo visto che lo spazio di configurazione è stato introdotto per poter assegnare indirizzi univoci negli spazi di I/O e memoria, in modo tale che a seguito della fase di con-

figurazione il software possa utilizzare direttamente i registri dei dispositivi, registri diversi da quelli per la configurazione (es. RBR per una stampante). Il costruttore della scheda deve definire uno o più blocchi di indirizzi, ciascuno con una dimensione, all'interno dei quali rendere disponibili i registri specifici della funzione. Inoltre il costruttore deve decidere per ciascun blocco se deve trovarsi nello spazio di I/O o di memoria. Quello che non può decidere è l'indirizzo di partenza dei blocchi, il quale è assegnato dal software, in modo tale che non ci siano sovrapposizioni tra i vari blocchi. In particolare per ogni blocco si prevede un **registro BAR** nello spazio di configurazione, usato dal software per assegnare l'indirizzo di partenza dei blocchi. I registri BAR sono solitamente grandi 32 bit, i primi b bit, se il blocco è grande 2^b , non sono scrivibili e danno informazioni sulla dimensione e sul tipo di blocco. In particolare il primo bit se è 0 vuol dire che il blocco è pensato per lo spazio di memoria e 1 se è pensato per lo spazio di I/O. Il software può scoprire quanti bit non sono scrivibili provando a scrivere tutti 1 e guardando quanti bit sono stati effettivamente scritti. I blocchi possono essere assegnati a regioni allineate, il software sceglie la regione e ne scrive il numero nei bit scrivibili del BAR. In questo modo una volta azzerati i 2 o 4 bit meno significativi il BAR conterrà l'indirizzo di partenza del blocco. Infine il software di inizializzazione setterà il bit 0 o 1 del registro di comando a seconda che il blocco sia di I/O o di memoria. A questo punto la funzione risponderà alle transazioni i cui indirizzi cadono in quella regione.

Interruzioni dei dispositivi PCI

I dispositivi PCI possono generare richieste di interruzione. Ogni dispositivo ha fino a 4 piedini di uscita per le richieste di interruzione. Ogni funzione può essere collegata al massimo a uno di questi piedini e più funzioni possono essere collegate ad uno stesso piedino.

Capitolo 20

I/O nel nucleo

20.1 Lezione 30 03/05

20.1.1 I/O nel nucleo

Abbiamo visto sia quando abbiamo parlato della protezione sia con le interruzioni che quando bisogna eseguire un'operazione di I/O il processo che la inizia tipicamente viene bloccato per un tempo perché l'operazione è lenta, verrà sbloccato una volta che l'operazione è terminata. Avevamo visto che quindi la soluzione che si adotta è che quando un processo P1 è bloccato perché è in corso un'operazione di I/O se ne mette in esecuzione un altro, P2, in modo tale che venga sfruttata efficientemente la CPU; eventualmente quando P1 viene sbloccato se ha priorità maggiore di P2 viene messo subito in esecuzione. Il completamento dell'operazione viene segnalato da una richiesta d'interruzione da parte della periferica, in modo tale che il sistema acquisisca il controllo del processore e eventualmente sblocchi P1. Quindi stiamo considerando un caso di *sincronizzazione*, in quanto il processo che ha iniziato l'operazione di I/O deve essere bloccato e sbloccato solo dopo il completamento dell'operazione.

Un'altra cosa che dobbiamo garantire è la *mutua esclusione* per le periferiche: non si vuole che mentre la periferica è occupata da un processo venga usata anche da un processo differente. Questi 2 problemi avevamo visto che possono essere risolti tramite l'utilizzo di semafori, che, anche in questo caso vengono infatti utilizzati, ora ci arriviamo.

Importante notare che i processi di cui stiamo parlando sono processi *utente* e di conseguenza non sono fidati. Un processo utente non ha alcun interesse a lasciare libero il processore dopo aver avviato un'operazione o a garantire la mutua esclusione di una periferica. Inoltre avevamo visto che i processi utente non possono interagire direttamente con le periferiche, quello spetta al sistema. Per impedire questi problemi, garantire la mutua esclusione e la sincronizzazione facciamo sì che:

- Impediamo agli utenti di interagire direttamente con le periferiche.
- Forniamo delle primitive per eseguire le operazioni di I/O sotto il controllo del sistema.

Per il primo punto basta vietare le operazioni di IN e OUT quando siamo a livello utente settando *IOPL = sistema* nel registro dei flags. Invece per vietare l'accesso alle periferiche che hanno registri mappati in memoria usiamo l'MMU, o non inserendo traduzioni che portino ai registri del processore o proteggendo le traduzioni con il bit *U/S = sistema*. Vediamo come sono fatte le tipiche primitive di lettura e scrittura su una periferica:

```
1 |     extern "C" void read_n(nat1 id, char* buf, natq quanti);
```

```
2 |     extern "C" void write_n(nat1 id, const char* buf, natq quanti);
```

Il campo *id* è l'id della periferica, ci sarà un array con gli indici di tutte le periferiche, *buf* puntatore al buffer da cui prendere o inserire i dati e *quanti* numero di byte da leggere/scrivere.

Regola Le primitive **non atomiche** possono chiamare al loro interno altre primitive, quindi possono essere interrotte, dato che non devono fare salva_stato e carica_stato. Verranno eseguite dal processo che era in esecuzione quando sono state invocate, è come se questo processo si innalzasse a livello sistema. Le primitive **atomiche** non possono essere interrotte, in quanto chiamano salva_stato e carica_stato, non sono eseguite da alcun processo.

20.1.2 Realizzazione con primitiva e driver

Vediamo un primo esempio di realizzazione di primitiva di lettura che sfrutta un driver.

Nota : Attenzione: *in pratica non si utilizzano i driver perché presentano alcuni problemi, lo vediamo solo per capire.*

L'interfaccia di ogni periferica ha un registro di controllo che viene utilizzato per abilitare/disabilitare le interruzioni e un registro per leggere/scrivere byte. Solitamente la lettura di questo registro funge da risposta alla richiesta di interruzione, ma questo dipende da esercizio a esercizio, non viene mandata una nuova richiesta di interruzione finché quella precedente non ha ricevuto risposta. L'operazione viene svolta in parte dalla primitiva e in parte dal driver che va in esecuzione ad ogni richiesta di interruzione da parte dell'interfaccia. La primitiva avvia l'operazione di I/O e blocca il processo, il driver si occupa del trasferimento vero e proprio dei byte e di sbloccare il processo una volta che è terminata l'operazione. Per gestire la mutua esclusione e la sincronizzazione utilizziamo i semafori, con il meccanismo visto quando li abbiamo studiati. Quindi la primitiva deve essere atomica, utilizza primitive semaforiche, deve essere interrotta e non farà salva e carica stato. Consideriamo che la primitiva venga invocata da un processo P1, che invoca la funzione *read_n*. Vediamo anche un descrittore, abbastanza standard, che viene utilizzato per la periferica (varia ovviamente in base all'esercizio):

```
1 read_n:
2     int $IO_TIPO_RN
3     ret
4     //Si attraversa il gate della IDT
5
6     a_read_n:
7         call c_read_n
8         iretq
9
10    struct des_io{
11        ioaddr iRBR, iCTL;
12        char* buf;
13        natq quanti;
14        natl mutex;
15        natl sync;
16    };
```

Utilizziamo un array di tali descrittori di IO mentre *id* sarà l'indice della periferica.

```

1 extern "C" void c_read_n(natl id, natb *buf, natl quanti){
2     //controllo parametri
3     des_io *d = &array_des_io[id]; //ricavo periferica
4
5     sem_wait(d->mutex); //mutua esclusione
6     //imposto parametri
7     d->buf = buf;
8     d->quanti = quanti;
9     outputb(1, d->iCTL); //abilito interruzioni
10    sem_wait(d->sync); //blocco processo per sincronizzazione
11    sem_signal(d->mutex); //fine mutua esclusione
12 }

```

Ora si esamina il driver che va in esecuzione per effetto di una richiesta di interruzione, che passa dall'APIC e poi arriva alla CPU. Il driver gira a livello sistema, deve terminare con una *iretq*. Il driver *non* è un processo, quindi è **forzatamente atomico**.

Nota: *Il driver usa le risorse del processo che ha interrotto, usa la sua pila sistema e anche la sua memoria virtuale visto che non si sta cambiando il valore di cr3. Inoltre gira con le interruzioni disabilitate perché manipola le code dei processi.*

```

1 a_driver_i:
2     call salva_stato //perche potrebbe cambiare il processo in
3         esecuzione
4     movq $i, %rdi
5     call c_driver //funzione generica, cambia il parametro passato
6         in rdi
7     call apic_send_EOI //per far passare le richieste di
8         interruzione con priorita piu bassa o uguale a quella
9         gestita dal driver.
10    call carica_stato
11    iretq
12
13
14    extern "C" void c_driver(natl id){
15        des_io *d = &array_des_io[id];
16        d->quanti--;
17
18        if(d->quanti == 0){ //se vero trasferisco l'ultimo byte
19            outputb(0, d->iCTL);
20            des_sem * s = &array_dess[d->sync];
21            s->counter++;
22
23            if(s->counter <= 0){
24                des_proc* lavoro = rimozione_lista(s->pointer);
25                inspronti(); preemption
26                inserimento_lista(pronti, lavoro);
27                schedulatore(); preemption
28            }
29        }
30        //leggo il nuovo byte e lo copio nel buffer dell'utente.
31        char c = inputb(d->iRBR);
32        *d->buf = c;
33        d->buf++;
34    }

```

Da notare alcune cose:

- La prima è che la disabilitazione delle interruzioni avviene prima di trasferire l'ultimo byte, se così non fosse e leggessimo prima il byte è possibile che l'interfaccia generi una nuova interruzione rimandando in esecuzione il driver, anche se nessuno ha iniziato una operazione di lettura. Il driver leggerebbe questo byte e lo copierebbe dove punta `d->buf`, sovrascrivendo parti casuali della memoria.
- Altra cosa è che non si può chiamare direttamente la `sem_signal()`, il driver è atomico e non può essere interrotto da una primitiva semaforica.
- Da notare anche che la scrittura in `d->buf` è eseguita mentre è attiva la memoria virtuale del processo interrotto, ciò comporta che il buffer deve essere allocato come variabile globale, nella parte utente/condivisa. Se così non fosse si userebbe la memoria virtuale del processo interrotto, le sue traduzioni, andando a scrivere in una parte di memoria casuale del processo interrotto. Inoltre il processo che aveva invocato la primitiva non riceverebbe effettivamente i dati. I buffer devono essere sempre dichiarati come variabili globali o nello heap, mai come variabili locali, in quanto queste si trovano in pila che sta nella parte privata dei processi.
- Il gate che porta alla `a_read_n` deve avere:
 - DPL che indica che può essere chiamata da livello utente, infatti le primitive di I/O devono poter essere chiamate dagli utenti.
 - Campo $L = sistema$ perché devono essere eseguite con il processore a livello sistema.
 - I/T non importa se girano o no con le interruzioni disabilitate o abilitate.
- Il gate che porta a `a_driver_i` deve avere:
 - $DPL = sistema$, il drive non può essere chiamato da un processo utente.
 - $L = sistema$, drive eseguito con il processore a livello sistema.
 - $I/T = interrupt$, deve essere eseguito, come detto, con le interruzioni disabilitate.

In generale in tutte le primitive bisogna controllare i parametri, in quanto questi arrivano dagli utenti che non sono fidati. Nel caso delle primitive di I/O bisogna controllare che *id* sia valido (non superi MAX-ID), che *quanti* non sia 0 e bisogna controllare *buf* per evitare il problema del **cavallo di troia**.

Questo problema avviene quando l'utente invece di passare l'indirizzo di un buffer che ha correttamente allocato passa l'indirizzo di qualsiasi cosa: parti della memoria che appartengono al sistema o ad altri processi. In questi casi i controlli fatti dalla CPU e dalla MMU non sono efficaci. Il passaggio del parametro in sé comporta solo la scrittura di un numero in un registro e la CPU non controlla nulla, non sa lo scopo di questa operazione. Quando si tenta di utilizzare l'indirizzo, il quale viene usato nella primitiva che gira a livello sistema, il cavalo si apre. Infatti la MMU non esegue nessun controllo perché la primitiva gira a livello sistema. Quindi se non si prendono provvedimenti e non si controllano i parametri l'utente può obbligare il sistema a scrivere/leggere su/da una zona di memoria a cui l'utente normalmente non ha accesso e non dovrebbe averne.

Nota: Nel caso particolare dell' I/O il cavallo si apre quando si ha la scrittura o la lettura sul buffer, quindi nel processo esterno.

Per controllare il buffer passato dall'utente, ma in generale per controllare gli indirizzi passati dall'utente, si può utilizzare una funzione fornita dal modulo sistema:

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

Questa funziona controlla che tutti gli indirizzi nell'intervallo $[begin, begin + dim]$ non attraversino il massimo indirizzo (no wrap-around) e che non ci siano indirizzi che fanno parte del buco. Percorre anche tutto l'albero di traduzione e controlla che tutti gli indirizzi dell'intervallo siano mappati, accessibili da livello utente, se *writeable* è *true* che siano scrivibili e se *shared* è *true* che facciano parte della zona utente/condivisa. Gli indirizzi devono essere mappati e normalizzati, altrimenti verrebbe sollevata un'eccezione di *page fault*, ma il driver è atomico, non può essere interrotto. Solitamente se questa funzione restituisce false si abortisce il processo. Il processo va abortito chiamando *abort_p()* che fa salva e carica stato e non *c_abort_p()* perché siamo in una primitiva non atomica.

Capitolo 21

Modulo I/O

21.1 Lezione 31 06/05

21.1.1 Modulo I/O

Abbiamo visto la scorsa lezione come gestire le primitive di I/O utilizzando un driver che va in esecuzione a seguito di una richiesta di interruzione. In realtà non è il meccanismo utilizzato, in quanto il driver presenta dei problemi con il meccanismo delle interruzioni, per 2 motivi:

1. Il driver deve essere eseguito con le interruzioni disabilitate in quanto manipola le code dei processi, quindi anche le richieste di interruzione a priorità alta devono aspettare.
2. Il driver non si può bloccare, non è un processo, è forzatamente atomico. Questa è una limitazione forte.

Il secondo punto viene risolto trasformando il driver in un processo, in particolare la richiesta di interruzione manderà in esecuzione un handler che ha lo scopo di mandare in esecuzione un processo, che svolge le operazioni che eseguiva prima il driver. Gestirà i trasferimenti e eventualmente sblocca il processo che aveva invocato la primitiva.

Il problema delle interruzioni è più serio, si potrebbe far girare questo processo con le interruzioni abilitate e disattivarle nei punti in cui si accede alle code dei processi, in quel caso ovviamente il processo deve girare a livello sistema. Si avrebbero però problemi nella scrittura del codice. La soluzione adottata è che questo processo faccia parte di un nuovo modulo distinto dal modulo utente o sistema, gira con le interruzioni abilitate e quando deve accedere alle code dei processi utilizza delle primitive di sistema, le quali girano con le interruzioni disabilitate, quindi si ha un accesso protetto. Questo modulo distinto sarà il **modulo I/O**.

21.1.2 Funzionamento effettivo del modulo I/O

Quindi la richiesta di interruzione da parte di una periferica manderà in esecuzione un piccolo handler (scritto in assembler), il quale manda in esecuzione il processo vero e proprio, questo processo sarà un **processo esterno**, nel senso che svolge funzioni di sistema pur non facendo parte di esso. Il processo esterno farà infatti parte del modulo I/O, girerà con le interruzioni abilitate, CPU a livello sistema (ora lo vediamo) e quando deve accedere a risorse condivise, quali le code dei processi, utilizzerà primitive di sistema in modo tale che l'accesso sia protetto.

Il modulo I/O conterrà tutto ciò che è legato alle primitive di I/O, le primitive stesse e le loro strutture dati. Quindi adesso l'utente ha la possibilità sia di chiamare le primitive di sistema viste fino ad ora, *sem_signal/wait*, *delay*, *activate* e *terminate_p*, sia nuove primitive che sono realizzate nel modulo I/O. Da esso si può accedere al modulo sistema sia attraverso primitive di I/O dedicate inaccessibili da livello utente, sia dalle classiche primitive di sistema, in particolare anche il modulo I/O userà le primitive semaforiche.

Come già anticipato questo nuovo modulo girerà con la CPU a livello sistema. In realtà idealmente vorremmo che ci fosse un livello intermedio, visto che il codice di questo nuovo modulo ha più diritti rispetto agli utenti e meno rispetto al sistema (non può accedere alle strutture dati del sistema), ma avendo solo 2 livelli a disposizione è stato scelto il livello sistema.

Una delle primitive riservate al modulo I/O è la **activate_pe()** che ha lo scopo di attivare un nuovo processo esterno. Ha gli stessi parametri della activate normale + il piedino dell'APIC da cui arriveranno le richieste di interruzione a cui il processo deve rispondere. Nel modulo sistema abbiamo una tabella *a_p* con una entrata per ogni piedino dell'APIC. La **activate_pe()** dopo aver creato il processo esterno inserisce il **des_proc** nell'entrata opportuna di questa tabella. Quello che farà l'handler, ce n'è uno per ogni possibile interruzione, è mettere in esecuzione il corrispondente processo esterno, prendendo il **des_proc** da questa tabella, farà **esecuzione = a_p[i]** con i piedino su cui è arrivata la richiesta di interruzione.

Nota: *Gli handler sono inseriti all'interno del modulo sistema, devono poter usare la salva e la carica stato, le quali non sono definite nel modulo I/O e devono manipolare le code dei processi.*

La precedenza del processo esterno avrà la forma: *MIN_EXT_PRIO+prio*, con *prio* numero naturale < 256. I processi esterni avranno una priorità molto alta, solitamente maggiore (forse sempre) dei processi utente.

Schema handler:

```

1 handler_i:
2     call salva_stato
3     call inspronti
4     movq a_p+i*8, %rax
5     movq %rax, esecuzione
6     call carica_stato
7     iretq

```

Viene salvato lo stato del processo che era in esecuzione quando è stata accettata la richiesta, perché potrebbe cambiare, e viene inserito in cima alla lista pronti. Le successive 2 righe servono a fare **esecuzione = a_p[i]** e la successiva coppia carica stato e **iretq** cederanno il controllo al processo esterno. Lo schema di un processo esterno è uguale per tutte le periferiche (cambiano delle parti in base agli esercizi) ed è il seguente.

```

1 extern "C" estern_ce(natl id){
2     des_io* d = &array_des_io[id];
3
4     for(;;){
5         ...
6         wfi();
7     }
8 }

```

La cosa particolare è che dopo la loro creazione i processi esterni non terminano più, si bloccano in attesa della prossima richiesta di interruzione. Il processo esterno non muore mai, quando ha finito il suo compito si blocca. Infatti il loro corpo è composto da un ciclo infinito che termina con una `wfi()` ovvero *wait for interrupt*, primitiva riservata al modulo I/O.

```

1 a_wfi:
2     call salva_stato
3     call apic_send_EOI
4     call schedulatore
5     call carica_stato
6     iretq

```

La primitiva salva lo stato del processo esterno, invia l'*EOI* all'APIC e mette in esecuzione un altro processo, sospendendo così il processo esterno che andrà nuovamente in esecuzione solo quando il corrispondente handler verrà nuovamente invocato.

Si può dire sicuramente che dopo la prima volta che il processo esterno è andato in esecuzione, la coppia `carica_stato-iretq` al termine dell'handler caricherà lo stato salvato dalla `wfi()`. Se l'handler è andato in esecuzione vuol dire che è passata una richiesta di interruzione, dunque l'APIC deve aver ricevuto l'*EOI* per la richiesta precedente, e l'ultima cosa che il processo esterno aveva fatto era chiamare la `wfi()`, non potrebbe non essere così, l'*EOI* viene mandato in fondo. Analizziamo meglio questa cosa: noi sappiamo che l'handler, in particolare la coppia `carica_stato-iretq` dell'handler, manderanno in esecuzione il processo esterno. Vediamo da che punto partirà: la primissima volta partirà dall'inizio mentre dalla seconda volta in poi so che partirà da subito dopo la `wfi()`, come detto prima. La `wfi` manda l'*EOI*, se è partito l'handler è perché l'APIC ha inviato una richiesta di interruzione, quindi l'APIC sa che non era ancora pendente una interruzione alla stessa priorità. Finché non si arriva alla `wfi` non viene mandato l'*eoI*, quindi finché non si arriva alla `wfi` per l'APIC è ancora in corso la gestione dell'ultima interruzione arrivata da quel piedino e quindi l'APIC non invierà richieste di quel tipo e non andrà in esecuzione il relativo handler. Di conseguenza prima o poi il processo esterno arriva alla `wfi` e prima di allora non può essere rimesso in esecuzione da un handler per quello che abbiamo detto, non possono arrivare richieste. La `wfi` salva lo stato del processo esterno, metterà in esecuzione un altro processo e poi dalla seconda volta in poi l'handler, la coppia `carica_stato-iretq` in particolare, metteranno in esecuzione il processo esterno dal punto in cui si era interrotto, quindi verrà caricato lo stato salvato dalla `wfi` e si riparte da lì.

Non possiamo dire con certezza, invece, quale processo andrà in esecuzione al termine della `wfi`. Infatti il processo esterno gira con le interruzioni esterne abilitate, mentre è in esecuzione vari processi potrebbero essere finiti in coda pronti.

Differenza tra driver e processo esterno

Adesso vediamo lo stesso esempio di primitiva di lettura che avevamo visto la scorsa lezione con l'uso del driver, vedendo le differenze tra l'utilizzo del driver e del processo esterno. L'operazione adesso sarà svolta in parte dalla primitiva di I/O e in parte dal processo esterno. Consideriamo un processo P1 che invoca la funzione `read_n()`. La funzione invocherà la primitiva vera e propria tramite una `int`, e avremo:

```

1 a_read_n:
2     call c_read_n

```

La `a_read_n` non chiama la salva e la carica stato, sia perché deve essere non atomica ed è eseguita dal processo che la invoca, sia perché adesso la primitiva fa parte del modulo I/O, la carica e la salva non ha modo di usarle, sono definite nel modulo sistema. Il descrittore di operazioni di I/O e la parte c++ della `read` sono le stesse dell'esempio della scorsa lezione. La differenza è che ora la primitiva è definita nel modulo I/O e quindi gira ad interruzioni abilitate (gate di tipo trap). Dobbiamo proteggere le risorse usate dalla primitiva rispetto ad accessi di altri processi. In particolare da altri processi che tentano di invocare la stessa primitiva e dal processo esterno associato all'interfaccia. Il primo caso si risolve utilizzando la mutua esclusione con il semaforo `mutex`. Il secondo caso si risolve facendo tutti gli accessi alle risorse prima di abilitare le interruzioni, il processo esterno può andare in esecuzione solo se le interruzioni sono abilitate. Di seguito il codice del processo esterno per questo esempio:

```

1 extern "C" void estern(nat1 id){
2     des_io* d = &array_des_io[id];
3
4     for(;;){
5         d->quanti--;
6         if(d->quanti == 0)
7             outputb(0, d->iCTL);
8         char c = inputb(d->iRBR);
9         *d->buf = c;
10        d->buf++;
11        if(d->quanti == 0)
12            sem_signal(d->sync);
13        wfi();
14    }
15 }
```

Il processo esterno, proprio come il driver, legge il nuovo byte dell'interfaccia e lo copia nel buffer utente, sbloccando il processo P1 quando sono terminati i byte da leggere. Bisogna disabilitare le interruzioni prima di trasferire l'ultimo byte, come con il driver, altrimenti l'interfaccia potrebbe inviare una nuova richiesta che manderebbe in esecuzione il driver, il quale copierebbe un byte indesiderato. Dobbiamo risvegliare il processo che aveva richiesto il trasferimento dopo aver trasferito effettivamente l'ultimo byte. Questo perché il processo esterno non è atomico, risvegliando P1 questo potrebbe andare in esecuzione, se ha priorità maggiore del processo esterno. Il processo risvegliato potrebbe iniziare ad usare i dati prima che il trasferimento sia completato e l'ultimo byte sia stato trasferito.

Capitolo 22

DMA e PCI Bus Mastering

22.1 Lezione 32 13/05

22.1.1 DMA

Finora le modalità di trasferimento dati dalla memoria ad un dispositivo e viceversa sono 2:

1. La modalità a controllo di programma.
2. La modalità tramite meccanismo delle interruzioni.

Il secondo è più lento del primo ma è più facile da implementare, entrambi i metodi però richiedono il coinvolgimento diretto del processore ma vogliamo a questo punto introdurre un metodo che non lo richiede. La **modalità DMA** è un nuovo metodo di trasferimento dati che prevede la capacità del dispositivo, una volta istruito dal software, di riuscire a eseguire il trasferimento da/verso lo stesso alla/dalla memoria, senza un coinvolgimento diretto del processore. In particolare viene detto al dispositivo quanti byte trasferire e il primo indirizzo da/su cui copiarli, una volta fatto il dispositivo esegue autonomamente l'operazione e quando ha trasferito tutti i byte solitamente invia una richiesta di interruzione.

Nota: *Da notare che ora l'interruzione è inviata dopo aver trasferito tutti i byte, prima era inviata per ogni byte da trasferire.*

Nota : Per gli esercizi: *Negli esercizi in cui le periferiche operano il trasferimento in DMA basta configurare i campi (come si faceva prima) e avviare l'operazione attivando le interruzioni, il trasferimento poi lo fanno da sole; nel processo esterno non bisogna fare manualmente il trasferimento.*

Vediamo ora il meccanismo DMA per passi, aggiungendo un componente alla volta. Inizialmente consideriamo il caso in cui abbiamo solo la CPU, la RAM e un dispositivo DMA. Esso deve coordinarsi con il processore per l'accesso al bus, essendo questo condiviso e visto che tutti e 2 vorranno accedere in RAM. Si implementa un handshake tra dispositivo a CPU, molto simile a quello che avevamo visto tra APIC e CPU, con collegamenti HOLD (da dispositivo a proc.) e HOLDA (da proc. a disp.).

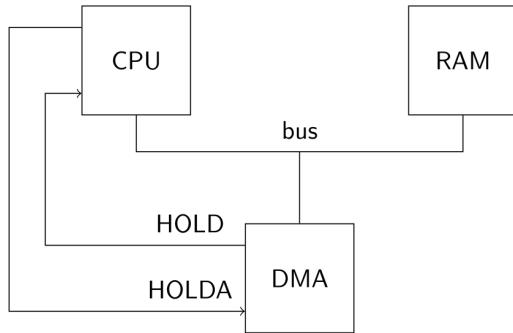


Figura 22.1: Architettura di base con DMA

Il dispositivo normalmente ha le sue uscite in alta impedenza, quando invece vuole compiere un trasferimento:

1. Attiva HOLD.
2. La CPU quando vede HOLD attivo smette di fare quello che stava facendo, mette le sue uscite in HI e attiva HOLDA.
3. Il dispositivo attiva le sue uscite, esegue il trasferimento, quando ha finito rimette le uscite in HI e disattiva HOLD.
4. La CPU riattiva le sue uscite riprendendo il controllo del BUS, disattiva HOLDA.

In questo handshake si vede che quindi la CPU dà precedenza al dispositivo DMA, in quanto può interrompersi anche in mezzo ad una istruzione, si parla dunque di *cycle stealing* per via del fatto che il dispositivo "ruba" cicli di bus alla CPU.

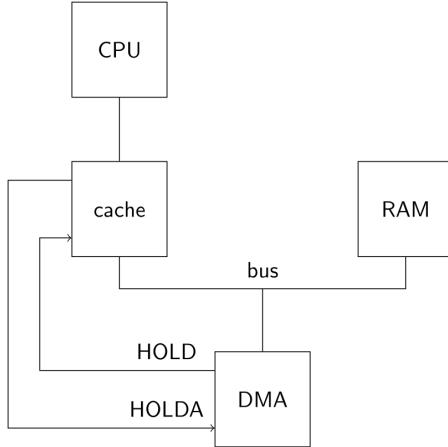
Nota: È importante notare due cose:

- In un singolo trasferimento non vengono trasferiti tutti i byte, il dispositivo probabilmente richiederà e farà più trasferimenti per trasferire tutti i byte richiesti.
- Mentre è in corso un trasferimento DMA la CPU potrà svolgere solo istruzioni già prelevate, in quanto non potrà accedere in RAM per prelevarne altre, è rallentata dal dispositivo DMA.

Ora vediamo che con l'introduzione della cache questo rallentamento non è significativo.

22.1.2 Interazione DMA con la cache

Aggiungiamo la cache allo schema e vediamo cosa cambia.



Innanzitutto con l'introduzione della cache non è più la CPU che accede al bus, ora la CPU non sa nemmeno che c'è il bus, ma vi accede il controllore cache, quindi l'handshake, analogo al precedente, ora si ha tra dispositivo DMA e controllore cache.

L'introduzione della cache porta vantaggi e svantaggi, il vantaggio è che ora la CPU può prelevare istruzioni direttamente in cache senza passare dal bus, di conseguenza è anche in grado di eseguire istruzioni mentre è in corso un'operazione DMA. Lo svantaggio è che ora viene meno l'ipotesi con la quale avevamo introdotto la cache, ovvero che tutte le possibili fonti di scrittura in RAM arrivano dalla CPU. Adesso è possibile che in memoria non ci siano i dati aggiornati e quindi non devono essere presi (situazione che può accadere se ad esempio la cache utilizza la politica write-back). Questi problemi possono comunque essere risolti sia in hardware che in software.

22.1.3 Cache con politica write-through

Se la cache che abbiamo introdotto usa una politica *write-through*¹ non ci sono problemi per le operazioni di lettura dalla RAM. Infatti la RAM contiene i dati già aggiornati, in quanto con questa politica si aggiorna sia la cache che la RAM.

Per i trasferimenti di ingresso bisogna invece assicurarsi che tutte le cacheline coinvolte siano o invalidate o aggiornate. La **soluzione hardware** è che il controllore cache sfrutti il bus condiviso osservando tutte le possibili sorgenti di scrittura in RAM. In particolare il controllore cache farà lo **snooping**² e quando vede che il dispositivo DMA vuole scrivere in RAM allora utilizzerà le linee di indirizzo per fare una scansione della cache, in caso di *hit* dunque può provvedere autonomamente a invalidare la corrispondente cacheline. Se il controllore oltre alle linee di indirizzo riceve anche le linee di dati può provvedere ad aggiornare la cacheline, in questo caso si parla di **snarfing**. Lo *snarfing*³ è molto più complicato di quel che sembra infatti non è previsto dai processori intel. La **soluzione software** è che il programmatore utilizzi delle istruzioni che permettono tipicamente di invalidare un intervallo di indirizzi in cache, quindi il software le deve eseguire prima che inizi il trasferimento o dopo che è terminato.

¹Ovvero se la cacheline è presente (*write hit*) il processore può aggiornare sia la cache che la memoria.

²Significa *ficcanasare*: fa riferimento al fatto che il controllore cache “ficca il naso” in quello che sta facendo il DMA.

³Significa *ingurgitare*.

22.1.4 Cache con politica write-back

Consideriamo ora il caso in cui la cache introdotta utilizzi la politica *write-back*⁴. Prima consideriamo il caso di **trasferimento di intere cacheline**. Nel caso di cacheline non dirty i casi sono gli stessi che per la politica write-through e questo vale anche per trasferimenti generici sempre con cache *write-back*. Nel caso invece di cacheline dirty abbiamo problemi sia nelle operazioni di ingresso, come prima, sia nelle operazioni di uscita. La cache contiene la versione più aggiornata dei dati e sarebbe errore se il dispositivo DMA li leggesse dalla RAM. La **soluzione hardware** per le operazioni di uscita in DMA è che il dispositivo DMA faccia uno snooping nello stato della cache prima di eseguire la lettura in RAM. Lo snooping è svolto su collegamenti dedicati oppure acquisendo il controllo del bus tramite l'handshake visto precedentemente. Tramite questo snooping il dispositivo DMA passa al controllore cache l'indirizzo interessato e il controllore risponde con il segnale di *hit* o *miss*. In caso di *hit* o il controllore cache invia direttamente i dati al dispositivo, se questo è in grado di fare lo snarfing, oppure prende il controllo del bus esegue una *write back* e infine il dispositivo può prelevare i dati aggiornati in RAM. Per i trasferimenti in ingresso (da dispositivo a RAM) si ha lo snooping da parte del controllore cache (come per le cache *write-through*), ma il contenuto della cacheline dirty va invalidato senza prima essere ricopiatato in RAM, perché il valore più aggiornato della cacheline deve essere quello prodotto dal dispositivo (trascrivere l'intera cacheline).

Per quanto riguarda la **soluzione software** il programmatore deve ordinare al controllore cache di eseguire il *write-back* di un certo intervallo di indirizzi **necessariamente prima** del trasferimento. Questo è necessario per le operazioni di uscita ma anche per quelle di ingresso per evitare che cacheline dirty non vengano ricopiate in RAM durante l'operazione o dopo il trasferimento col rischio di sovrascrivere quanto scritto dal dispositivo. Nel caso di operazioni in ingresso bisogna invalidare la cacheline in RAM, come abbiamo visto nella soluzione hardware. Solo nel caso di trasferimento di intere cacheline nel caso di operazioni in ingresso si un'ottimizzazione: si ha un'istruzione che il processore può usare per invalidare l'intera cacheline senza prima effettuare il *write-back*. L'importante è che anche con questa ottimizzazione l'invalidazione vada fatta prima dell'inizio del trasferimento.

Consideriamo ora il caso di **trasferimenti generici**, ovvero trasferimenti che potrebbero coinvolgere anche solo parti di cacheline e non necessariamente tutta. La cacheline dovrà contenere il valore attualmente in cache per i byte non necessari e il valore scritto dal dispositivo per i rimanenti. Per la **soluzione hardware** nel caso di trasferimento in uscita è uguale al caso visto prima solo che il dispositivo sarà interessato solo ad una parte di cacheline e non a tutta. Per il trasferimento in ingresso se il controllore cache è in grado di aggiornare (snarfing) autonomamente la cache con i nuovi dati scritti dal dispositivo il trasferimento resta immutato con snooping da parte del controllore. Se invece il controllore può solo invalidare si risolve il problema con lo snooping da parte del dispositivo. Prima di eseguire il trasferimento il dispositivo compie lo snooping, in caso di *hit* si hanno 2 opzioni:

1. La prima è che il dispositivo si stacca momentaneamente il controllore esegue il write back, poi il dispositivo scrive in RAM.
2. La seconda è che il controllore invia al dispositivo il contenuto della cacheline, il dispositivo combina questi dati con quelli che deve scrivere e poi scrive l'intera cacheline in RAM.

⁴Ovvero se la cacheline è presente (*write hit*) il processore può aggiornare solo la cache.

In ogni caso il controllore deve invalidare la sua cacheline perché la sua versione aggiornata si troverà ora in RAM. Per quanto riguarda la **soluzione software** l'unica cosa che cambia è che non si può usare l'ottimizzazione per non fare il write back.

Nota: Lo snooping da parte del controllore cache è più efficiente dello snooping da parte del dispositivo in quanto il primo avviene in parallelo con l'operazione in RAM, il secondo, invece, deve essere completato prima di poter dialogare con la RAM.

22.1.5 Interazione con la memoria virtuale

Aggiungiamo l'ultimo elemento all'architettura, l'MMU. La MMU come abbiamo visto si trova tra la CPU e la cache e ha il compito di intercettare tutti gli indirizzi generati dalla CPU e tradurli in indirizzi fisici. I dispositivi di I/O, quindi anche quelli che operano in DMA vedono ed utilizzano indirizzi fisici. Facciamo conto che il dispositivo DMA debba operare un trasferimento da/verso un buffer che si trova agli indirizzi $[b, b + n]$, bisogna far sì che:

1. Al dispositivo vada comunicato l'indirizzo fisico corrispondente a b , ovvero $f(b)$, e non l'indirizzo virtuale b .
2. Se l'intervallo $[b, b + n]$ attraversa più pagine che non sono tradotte in frame contigui allora il trasferimento va spezzato in più parti, in modo tale che in ciascuna parte si coinvolga solo frame contigui.
3. La traduzione degli indirizzi coinvolti in un trasferimento non deve cambiare mentre il trasferimento è in corso.

Per capire la richiesta data dal **primo punto** basta prendere l'esempio in cui comunichiamo l'indirizzo virtuale b al dispositivo, questo lo usa come se fosse un indirizzo fisico andando a scrivere in parti di memoria casuale (tranne se $f(b) = b$). Per il **secondo punto** supponiamo che l'intervallo $[b, b + n]$ attraversi 2 pagine che sono mappate in 2 frame non contigui F1 e F3, intervallati dal frame F2. Se non spezzassimo il trasferimento il dispositivo andrebbe a leggere/scrivere da $f(b)$ per n byte invadendo il frame F2. Per il **terzo punto** immaginiamo di trovarci in un sistema multiprocesso che esegue lo *swap-in* e lo *swap-out* dei processi. Supponiamo che un processo P1 avvii un trasferimento in DMA verso un suo buffer privato e mentre è in corso venga fatto lo *swap-out* di P1 e lo *swap-in* di P2. Il dispositivo DMA è ignaro di ciò e continuerà a leggere o scrivere agli indirizzi fisici precedentemente occupati dal buffer di P1 ma che ora sono occupati da parti di memoria di P2.

Talvolta negli esercizi è richiesto di fare attenzione alla memoria virtuale, nel caso in cui si abbiano periferiche che operano in BUS mastering.

Esempio 11:

```

1 paddr p = trasforma(ce->buf);
2 natq rimanenti = DIM_PAGINA - (p & 0xFFFF);
3 if (rimanenti > quanti)
4     rimanenti = quanti;
5 ce->buf = buf + rimanenti;
6 ce->quanti = quanti - rimanenti;
7 outputl (p, ce->iBM PTR);
8 outputl (rimanenti, ce->iBML EN);
9
10 //NEL PROCESSO ESTERNO

```

```

11 if(ce->quanti){
12     natq rimanenti = ce->quanti;
13     if(rimanenti > DIM_PAGINA)
14         rimanenti = DIM_PAGINA;
15     paddr p = trasforma (ch->buf);
16     ch->buf += rimanenti;
17     ch->quanti = ch->quanti-rimanenti;
18     outputl (p, ce->iBMPTR);
19     outputl (rimanenti, ce->iBMLEN);
20 }else{
21     sem_signal(ce->sync);
22 }
23 wfi();

```

□

Nota: Il meccanismo DMA è importante per gli scritti di I/O ed è molto chiesto anche all'orale.

22.1.6 PCI Bus Mastering

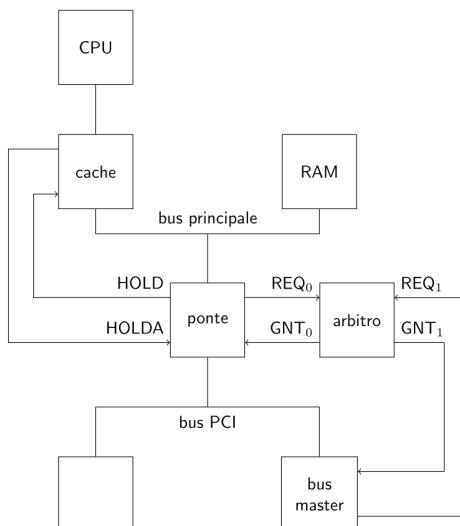


Figura 22.2: Esempio di architettura con bus PCI

Fino ad ora avevamo considerato che l'iniziatore delle transazioni sul bus pci fosse sempre il ponte, che agiva per conto della CPU. In realtà qualsiasi dispositivo collegato al bus pci potrebbe essere iniziatore, quelli in grado di esserlo si dicono **bus master**. Questi dispositivi devono essere collegati ad un nuovo oggetto che inseriamo, che in realtà è un'espansione del ponte, ovvero **l'arbitro**. Qualsiasi dispositivo bus master, compreso il ponte, avrà dei collegamenti RET (da dispositivo ad arbitro) e GNT (da arbitro a dispositivo) con l'arbitro. Quest'ultimo ha infatti lo scopo di gestire i vari accessi al bus pci.

In particolare, prima di ogni transazione, si ha una fase di arbitraggio: il dispositivo attiva RET e quando riceve GNT vuol dire che sarà il prossimo a poter iniziare una transazione. L'arbitraggio può avvenire anche mentre è in corso una transazione pci, poi quando il dispositivo che ha ricevuto il GNT vede che il bus è libero (FRAME e IRDY disattivi)

può iniziare la transazione. Dal punto di vista software è come se il dispositivo fosse collegato direttamente al bus principale, il software comunica direttamente con il dispositivo, ad esempio il numero di byte da trasferire e l'indirizzo del primo byte. Dal punto di vista hardware avviene tutto per tramite del ponte. In particolare questo è configurato in modo tale da essere l'obiettivo di ogni transazione iniziata da un dispositivo bus master. Nel caso di operazioni di uscita, ovvero dispositivo che vuole leggere dalla RAM, è il ponte che pensa ad accedere in RAM, a prelevare i dati e a fornirli al dispositivo. In caso di operazioni di ingresso, il dispositivo vuole scrivere in RAM, il ponte memorizza i dati in un suo buffer interno e poi fa lui la copia dei dati in RAM. Il ponte si comporta come un normale dispositivo DMA e ciò che abbiamo visto fino ad ora non cambia.

Ora bisogna analizzare e risolvere un problema che si crea a causa della bufferizzazione interna del ponte, ovvero quando il dispositivo DMA che vuole scrivere in RAM scrive nel buffer interno del ponte e poi ci penserà lui a scrivere i dati in RAM. Quando il dispositivo DMA ha terminato di scrivere i dati nel buffer del ponte invia una richiesta di interruzione e a quel punto si crea una corsa: da una parte la richiesta di interruzione che passa dall'APIC e poi arriva alla CPU e dall'altra i dati che passano dal ponte e poi arrivano in RAM. Se arriva prima la richiesta di interruzione e viene accettata il software potrebbe iniziare ad utilizzare i dati prima che siano stati effettivamente scritti tutti e quindi una parte non sono aggiornati.

La **soluzione software** al problema è far sì che la routine associata alla richiesta di interruzione prima di permettere l'accesso al buffer, potrebbe eseguire una lettura di un registro del dispositivo. La risposta a questa lettura sarebbe accodata dopo l'ultimo trasferimento di dati del bus master e quindi l'istruzione di lettura verrebbe completata dalla CPU necessariamente dopo che il ponte ha finito di trasferire i dati in RAM.

La **soluzione hardware** prevede che si crei un handshake tra ponte e APIC, il ponte aspetta di aver svuotato il buffer dai dati che erano presenti al momento della richiesta di interruzione prima di dare il permesso all'apic, il quale aspetta l'OK del ponte prima di inoltrare la richiesta di interruzione.

Capitolo 23

La pipeline

23.1 Architettura interna del processore

La CPU che abbiamo visto fino ad ora abbiamo detto che compie ciclicamente

$$fetch \rightarrow decode \rightarrow esecuzione$$

e che fra un'istruzione e l'altra controlla se ci sono richieste di interruzione. I processori moderni in realtà fanno la stessa cosa ma sono molto più veloci. Questi infatti riescono anche a fare più cose e magari portare avanti più istruzioni contemporaneamente. Vediamo dunque dei metodi che permettono di velocizzare il lavoro del processore.

23.1.1 Pipeline

Idea: si scomponete l'azione che svolge la CPU e ci si ferma dopo ognuna di queste azioni. Ogni azione è svolta da un circuito combinatorio diverso, si ha anche un registro in mezzo alle RC. In particolare si va a scomporre l'esecuzione dell'istruzione in 5 fasi che sono: *prelievo dell'istruzione, decodifica, prelievo operandi, esecuzione e scrittura del risultato*. Ognuna di queste azioni viene svolta da una parte diversa della CPU, da più circuiti combinatori. L'idea è che, se va tutto bene, nel momento in cui abbiamo prelevato un'istruzione e siamo in fase di decodifica, la parte del circuito che si occupa del prelievo in quel momento non è usata, quindi possiamo utilizzare quella parte di circuito per prelevare la successiva istruzione, e così via. I registri tra le RC sono indispensabili perché le RC sono reti senza memoria e cambiano subito lo stato di uscita quando c'è un nuovo stato di ingresso. Adesso occorrono 5 cicli di clock per ogni istruzione, mentre prima serviva in un ciclo si faceva tutto. La soluzione è quindi aumentare la velocità del clock. Questo in realtà non è un problema perché con questa configurazione la distanza tra un registro ed un altro (da cui dipende la durata del clock) diminuisce. Si perde un po' ma non è un problema, in questo modo si possono portare avanti più istruzioni contemporaneamente.

Quindi la pipeline effettivamente ci fa riuscire nel nostro intento, tuttavia sorgono altri problemi. Infatti, specialmente nei processori INTEL, può capitare che si debba utilizzare per fasi diverse la stessa parte di circuito nella CPU. Ad esempio se la CPU deve fare un prelievo operandi di una istruzione e scrittura di un'altra istruzione, entrambe devono accedere in memoria.

23.1.2 Processori RISC

Sono dei tipi di processori (non molto usati) diversi dai processori CISC, come lo sono i processori INTEL. Questi processori utilizzano istruzioni elementari e tutte della stessa dimensione. Con questo tipo di processori si può utilizzare la pipeline, mentre con i processori CISC, quindi anche con gli INTEL, sono stati dovuti prendere accorgimenti, quali le e-istruzioni, per poterla usare. Nei processori RISC tutte le istruzioni hanno tutte la stessa dimensione e si ha 2 istruzioni per accedere in memoria (*load* e *store*), tutte le altre operazioni sono sui registri.

23.1.3 Le e-istruzioni

Sono un set di istruzioni derivanti da una trasformazione delle istruzioni base dell'assembly, tramite un circuito, usate dai processori INTEL per poter utilizzare la pipeline. Il problema era che le istruzioni assembly base avevano dimensioni diverse e sono complesse dal punto di vista del calcolo. Le e-istruzioni hanno tutte la stessa lunghezza (4 byte) e sono semplici.

23.1.4 Alee

Vediamo altri problemi legati all'introduzione della pipeline. Ci sono alcune situazioni che impediscono di eseguire un'istruzione ogni ciclo di clock. Queste situazioni sono molto frequenti e prendono il nome di **alee** e possono causare quelli che si chiamano **stalli della pipeline**. Gli stalli sono cicli di clock in cui non viene iniziata una nuova istruzione e di conseguenza si perde 1 ciclo di clock. Abbiamo 3 tipi di alee:

- **Alee strutturali.**
- **Alee sui dati.**
- **Alee sul controllo.**

Le alee **strutturali** si presentano quando si hanno istruzioni che fanno uso delle stesse risorse del processore contemporaneamente (es. 2 istruzioni hanno bisogno della ALU). Viene bloccato il flusso della pipeline per almeno 1 ciclo di clock fino a quando la risorsa non è di nuovo accessibile.

Le alee **sui dati** si hanno quando si hanno istruzioni che usano un risultato che viene calcolato dall'istruzione precedente. Ad esempio una istruzione j deve aspettare il risultato dell'istruzione precedente i . La soluzione che si adotta in questo caso è un filo di bypass in modo tale che l'istruzione j prenderà il risultato direttamente dalla fase di esecuzione dell'istruzione i .

Le alee **sul controllo** si hanno nelle istruzioni di salto in cui il successivo indirizzo da prelevare non è quello immediatamente successivo ma dipende dal risultato del salto. Si risolve cercando di indovinare dove porterà l'istruzione di salto con una predizione statica o dinamica. Nella predizione statica l'istruzione viene predetta sempre nello stesso modo, ad esempio se il salto è all'indietro mi aspetto che ci sia un ciclo e quindi torno direttamente indietro. Per la predizione dinamica si sfrutta una piccola cache che serve a ricordare cosa ha fatto nel passato una istruzione, in base a ciò predire il prossimo indirizzo da prelevare. In caso di predizione sbagliata vengono invalidate le 3 istruzioni successive. Per queste istruzioni è presente un flag che se attivo quando si arriva allo stato di scrittura non conclude l'operazione.

23.1.5 Esecuzione fuori ordine

Abbiamo visto la pipeline che è una tecnica elementare per cercare di velocizzare il funzionamento di un processore. Esistono altri metodi, vediamone un altro.

Supponiamo che ad un certo punto un'istruzione non può passare perché c'è un alea strutturale e quell'istruzione ha bisogno dell'ALU in 2 stadi e non possiamo farla partire subito dopo la precedente. Invece di sprecare un clock si può vedere se si può far passare l'istruzione successiva. In generale l'idea dell'**esecuzione fuori ordine** consiste nel far eseguire istruzioni successive ad istruzioni che magari sono bloccate (stallo) a patto che le istruzioni successive non abbiano dipendenze con le istruzioni bloccate. Si può eseguire una e-istruzione nel primo momento in cui ci sono tutte le risorse necessarie tenendo conto delle dipendenze. Le dipendenze sono 3:

- **dipendenza sui dati.**
- **dipendenza sui nomi.**
- **dipendenza sul controllo.**

Una e-istruzione j dipende sui dati di una e-istruzione i se l'istruzione i produce un risultato che j deve usare. Può dipendere anche transitivamente da i .

Delle dipendenze sui nomi si possono avere 2 tipi: l'istruzione i legge dallo stesso registro in cui scrive l'istruzione j (successiva ad i). SUB R4, R1, R6 // i legge R1 ADD R1, R2, R3 // j scrive in R1. Oppure se 2 istruzioni scrivono nello stesso registro, non può passare prima la seconda: ADD R1, R2, R3 e SUB R1, R4, R5.

Le dipendenze sul controllo si hanno se un'istruzione può essere eseguita o meno in base al risultato della istruzione di salto.

Le dipendenze sui dati non possono essere risolte perché sono quello che il programma sta facendo e non posso modificarne il significato. Le dipendenze sui nomi possono essere risolte tramite una rinominazione dei registri. La rinomina dei registri può essere svolta autonomamente dal processore, basta aggiungere un componente che lo fa.

23.1.6 Organizzazione interna del processore

L'architettura di un processore moderno, in particolare se si vuole implementare l'esecuzione fuori ordine, ha bisogno di individuare le dipendenze e cercare una soluzione. Per riconoscere le dipendenze e per capire se un'istruzione va fermata usiamo delle stazioni di prenotazione, una posizionata davanti ad ogni ALU. In queste stazioni si vanno ad accodare tutte le e-istruzioni in attesa che siano pronti i loro risultati, inoltre per ogni registro avremo:

- bit W , vale 1 se c'è una e-istruzione in esecuzione che deve scriverci dentro.
- $cont$, conta quante istruzioni vogliono leggere dentro il registro.
- il contenuto del registro.

Quando si vuole emettere una e-istruzione si guarda la sua destinazione e innanzitutto il flag W , se è a 1 abbiamo dipendenza sui dati. Se queste code sono piene non c'è modo di risolvere la alea strutturale se non con il blocco del flusso della pipeline.

Come già accennato le dipendenze sui nomi possono essere eliminate attraverso la rinomina dei registri. Distinguiamo registri fisici e registri logici. I logici sono quelli che compaiono

nel flusso di programma, quelli su cui si fanno davvero i conti sono i fisici, sono molti di più dei logici. Per ogni registro logico viene specificato quale registro fisico contiene il suo valore, tramite puntatori. Quando una e-istruzione viene emessa, si crea una nuova corrispondenza fra il registro logico destinatario e un registro fisico libero. Avremo W a 1 nel registro fisico se deve scriverci almeno una e-istruzione emessa, $cont$ numero delle e-istruzioni emesse che deve leggere dal registro fisico F , C a 1 se vi è una corrispondenza fra un registro logico e il registro fisico F a cui il bit C appartiene. Il registro fisico F è libero se C , W e $cont$ sono a 0. Quindi il processore ora per risolvere le dipendenze sui nomi può usare la corrispondenza

registro logico - registro fisico

23.1.7 Esecuzione speculativa

Proprio come per l'esecuzione fuori ordine l'obiettivo dell'esecuzione speculativa è quella di evitare che la pipeline vada in stallo. Tramite l'esecuzione speculativa e in particolare tramite la tecnica della **speculazione** si cercano di risolvere sia le dipendenze sui nomi sia le dipendenze sul controllo. In particolare questa tecnica prevede che i risultati di istruzioni che dipendono da istruzioni di salto non ancora risolte vengano scritti in registri temporanei e trasferiti nei registri reali solo quando abbiamo la certezza che quelle istruzioni andavano realmente eseguite. Per implementare questa tecnica bisogna aggiungere un nuovo stadio detto *stadio di ritiro delle istruzioni*. I risultati delle istruzioni diverranno effettivi solo se la e-istruzione che li ha prodotti passa dallo stadio di ritiro. Questo stadio fa uso di una struttura dati detta **ROB**¹, una coda di descrittori di e-istruzioni. Il ROB per ogni e-istruzione ne memorizza il tipo, se è stata completata o se è ancora in fase di esecuzione e se la e-istruzione è di controllo anche l'esito previsto per il salto. Quando la e-istruzione in testa al ROB risulta completata viene ritirata compiendo le seguenti azioni:

- Se l'istruzione è operativa il risultato prodotto diventa effettivo e l'istruzione viene estratta dalla coda
- Se l'istruzione è di controllo: se la previsione è sbagliata viene svuotato il ROB con annullamento delle istruzioni eventualmente completate. Se la previsione è corretta la e-istruzione viene estratta dalla testa del ROB.

Per poter implementare questa tecnica bisogna anche avere 2 tipi di corrispondenza per ogni registro logico:

- corrispondenza **non speculativa**: puntatore al registro fisico che contiene l'ultimo valore buono calcolato.
- corrispondenza **speculativa**: puntatore al registro fisico che contiene il valore corrente.

Quindi si può risolvere il problema delle dipendenze sul controllo in questo modo: se la previsione del salto è corretta, i valori non speculativi vengono aggiornati con gli attuali valori speculativi. Se la previsione è sbagliata i valori speculativi vengono ripristinati con i corrispondenti valori non speculativi che sono gli ultimi che sicuramente andavano calcolati. Il valore non speculativo relativo ad un registro logico viene aggiornato quando l'istruzione esce dal ROB con il valore del registro nell'istruzione, non con il valore speculativo attuale.

¹Sta per *ReOrder Buffer*, ovvero buffer di riordino

Nota: Le istruzioni nel ROB possono essere eseguite in qualsiasi ordine, una volta che hanno pronti i dati, ma devono essere ritirate dal ROB nell'ordine che rappresenta il flusso delle istruzioni voluto dal programma, nell'ordine in cui sono immagazzinate in esso. Quando una istruzione esce dal ROB siamo sicuri che andava eseguita

Nota : Altre note generali:

- Il processore traduce le istruzioni in e-istruzioni man mano che le preleva, ma non ha una visione completa di tutto il flusso di esecuzione.
- Prese 2 istruzioni i e h con j successiva a i , la dipendenza è una cosa che impedisce di scambiare le due e-istruzioni.
- I registri di destinazione vengono sempre rinominati anche quando non c'è alcun tipo di dipendenza.

Ne concludiamo che le uniche dipendenze che danno fastidio sono le **dipendenze sui dati** perché le altre le possiamo risolvere nei modi visti.

23.1.8 Dipendenze nelle istruzioni load e store

Non si può sapere se ci sono dipendenze, in quanto non c'è modo di sapere se gli indirizzi sono gli stessi. Per poter scoprire la dipendenza divido le istruzioni in una che calcola gli indirizzi e in una che sposta il contenuto, aggiungo poi dei load e store buffer al cui interno conservo gli indirizzi delle store e delle load in esecuzione e così scopro la dipendenza. Per quanto riguarda la speculazione invece non posso completare le load e le store in quanto andrei a cambiare la memoria, salvo quindi il valore nei load e store buffer e poi quando l'istruzione deve essere completata, completo anche la scrittura. Che succede se eseguo speculativamente una load che mi genera Page Fault? Il Page Fault non può essere generato subito ma devo segnare nel ROB che è stato generato un Page Fault e poi quando l'istruzione esce dal ROB allora lo mando.

Capitolo 24

Lezioni di Laboratorio

24.1 Lezione 3 (Laboratorio) 29/02/2024

(Appunti di laboratorio)

- **pwd** dice in che cartella sei
- **touch** crea file vuoto
- **file "nome file"** dice il tipo del file
- con **>** mando l'output di un programma in un file (che gli passo)
- con **|** mando l'output di un programma come input di un altro
- **man** apre il manuale (q per uscirne)
-

vim ha 3 modalità: quando si apre è in modalità normale ovvero a ogni tasto è buildato un comando ad esempio:

- se schiaccio **i** vado in modalità inserimento (ci si muova con le frecce)
- con **esc** si torna in modalità normale
- con **w** mi sposto avanti di una parola
- con **b** indietro di una parola
- con **cw** faccio change word (cancella la parola e mi mette in modalità inserimento)
- schiaccio **w/v (?)** e con **w** salvo, con **q** esco e **qw** salvo ed esco

Nota: *gcc compila il c, g++ compila il c++*

Nota: *GNU è il sistema operativo che sta sopra linux ecc.*

Come si compila:

```
gcc main.c -o main
```

per compilare l'assembler:

```
as somma.s -o somma
```

per creare un file oggetto:

```
gcc -c main.c -o main.o  
as somma.s -o somma.o
```

make main per compilare (?)

Essendoci una convenzione tra chiamante e chiamato, mettendo nel file somma.s solo una nop, quando vado a collegare i due file e eseguo ./main ottengo un segmentation fault; già mettendo una ret, anche se non faccio nulla nel file non ho più segmentation fault.

Comando su gdb per vedere dove punta rsp:

```
x/10g $rsp
```

si = comando che compie una sola istruzione (single instruction)

L'intel x86 64 ha come calling convention che se il valore di ritorno è un intero o un puntatore (per noi è sempre così) deve stare in RAX.

24.2 Lezione 5 (Laboratorio) 05/03/2024

Si allinea lo stack a 16 byte perché in alcune istruzioni avanzate di S.I.D. è richiesto (noi non le vediamo ma ci atteniamo comunque a questo standard).

Nota: se non si fanno *CALL* potremmo non farlo, occhio all'eventualità di passaggio per flussi di controllo non "conosciuti"

Nella cartella CE/bin troviamo:

- compile : prende tutti i file *cpp* e li compila, come target dell'eseguibile creato ci mette una vm... in *a.output* c'è un minimo kernel per supportare tutte le istruzioni (supporta l'esecuzione del programma)

se lanciamo **boot -g** la vm parte con il debugger (con gdb).

24.2.1 Mangling dei nomi

Mangling dei nomi: codifica dei nomi di funzioni e variabili in nomi univoci (ci sono le slide ma sono 80, in un file di testo di 40 righe del prof c'è tutto)

Inizia sempre con: _Z Poi ci va il nome della funzione preceduto dal numero di caratteri del nome (in ASCII): es. *funcion* *funz* diventa _Z4funzic (i e c sono i tipi dei parametri che contiene la funzione)

C'è un tool "c++filt" che fa il demangling: mi da il nome iniziale (da riga di comando)

Vediamo le strutture:

```
1 struct elem{  
2     int val;  
3     char boh;  
4 }  
5 funz(int, char)  
6 _z4funzic  
7  
8 ciao(elem, int)  
9 _z4ciao4elemi
```

Tipi composti: puntatore:

```
1 _z4ciaoP4elemPi --> ciao(elem*, int*)
```

```
1 objdump -d a.^C
```

per disassemblare e cercare poi il mangling se non ci si ricorda

code . per aprire vscode nella cartella corrente

Lavoreremo solo con strutture più piccole di 16 byte (se più grandi ne crea 2 ecc), se sono più piccole...

24.3 Lezione 6 (Laboratorio) 07/03

Unica differenza delle funzioni delle classi da quelle globali è che c'è sempre il parametro nascosto this che viene sempre passato nel primo registro e si indica con E

Per le classi:

$_ZN + numchar + nome + param\dots$

Il costruttore è C1, se questo prende per esempio in ingresso 2 interi si aggiunge C1ii

Strutture più grandi di 8 byte ma minore di 16 (più di 16 si va nello stack) si passano in 2 registri, lo stesso vale per il ritorno (al ritorno non lo vedremo mai).

Quando dichiaro e definisco nel main una istanza di una classe **non** usando la new questa starà nella zona di memoria del main, se avessi usato new sarebbe andata nello heap.

LEA, a differenza della MOV, non sposta il contenuto di un qualcosa contenuto a un certo indirizzo ma calcola (fa quindi dei conti) un indirizzo e lo mette in dest.

24.4 Lezione 13 (Laboratorio) 21/03

Laboratorio, esercizi traduzione primi due appelli di Giugno 2023.

Note sui comandi del debugger:

- x esamina memoria: vuole un indirizzo esadecimale (inizia con 0x)
- x/ esamina memoria passandogli il tipo (es x/8b, x/8c)
- c continua
- b breakpoint (immagino)
- s step
- n next

Nota: Nota su esercizi con interruzioni: quando si hanno variabili che sono condivise tra la routine di interruzione e il programma principale devono essere volatile. La non presenza di volatile presentava errori se si compilava con ottimizzazioni nell'es 1 sulle interruzioni visto a lezione. Anche le variabili globali si mettono volatile.

24.5 Lezione 20 (Laboratorio) 09/04

Laboratorio, appunti esercizio 1:

Esempio 12 - Aggiungere una primitiva al nucleo:

- In *costanti.h* definire il nuovo tipo e dare il gate corrispondente (si fa tramite `#define NOME numero gate exadecimal`).
- Dichiarazione delle primitive in *sys.h* tramite *extern "C"* così quando l'utente deve chiamare sa dove sono.
- Scrivere poi le primitive a livello sistema (in *sistema.s*): caricare il gate (con *carica_gate NOME nome_funzione_assembler LIVELLO*, il livello indica il livello necessario per attraversare il gate).
- In *sistema.s* bisogna definire la chiamata alla primitiva in c++ (quindi quella alla *c_syscall*) (fine **livello sistema**).
- Poi, passando a **livello utente**, in *utente.s* si avrà la chiamata alla primitiva che permette di passare a livello sistema (*INT \$NOME*).
- In *utente.cpp* si ha la chiamata vera e propria della primitiva da parte dell'utente: nel *main* si crea un processo con *activate_p* a cui si passa la funzione che chiama le primitive dichiarate prima in *sys.h*.

Nota: Lo schedulatore è una riga di codice: rimuove da lista pronti.

□

24.6 Lezione 21 (Laboratorio) 11/04

Note laboratorio:

- Page fault richiede l'intervento..
- Il debugger per funzionare usa la int3: scrive al posto dell'istruzione in cui scriviamo il breakpoint 0xCC (ovvero int3) → non c'è nulla di magico nel debugger.
- Es1: Scriviamo un programma che cattura l'eccezione della divisione e se vede una divisione per 0 ritorna un numero che ci pare
- Es2: Scriviamo un programma che inserisce un br

24.7 Lezione 27 (Laboratorio) 23/04

Laboratorio, soluzione esame 01/02/2012:

```
1 01/02/2012
2
3 sistema.cpp
4
5     bool rr = false;
```

```

7     extern "C" void a_abilita_rr(){
8         rr = true;
9     }
10
11    extern "C" void a_disabilita_rr(){
12        if(!rr) return;
13        rr = false;
14
15        for(natl i = 0; i < MAX_PROC_ID; i++){
16            des_proc* p = des_p(i);
17            if(p){
18                p->quanto = MAX_QUANTO;
19            }
20        }
21    }
22
23    ///////////////////////////////////////////////////
24
25    p->quanto = MAX_QUANTO
26
27    ///////////////////////////////////////////////////
28
29    nel driver del timer
30
31    if(rr){
32        esecuzione->quanto--;
33        if(esecuzione->quanto == 0){
34            esecuzione->quanto = MAX_QUANTO;
35            inserimento_lista(pronti, esecuzione);
36            schedulatore();
37        }
38    }
39
40 sistema.s (si fa prima qui)
41
42 nei gate
43
44     carica_gate TIPO_A_RR a_abilita_rr LIV_UTENTE
45     carica_gate TIPO_d_RR a_disabilita_rr LIV_UTENTE
46
47 nelle funzioni:
48     si dichiarano .extern le primitive che scriviamo in
        sistema.cpp
49
50     call c_abilita_rr
51
52     call c_disabilita_rr

```

Capitolo 25

Appunti laboratorio Leonardi

25.1 Lab 6

25.1.1 Parte 1

-

25.1.2 Parte 2

- Tenere sempre a mente che i riferimenti sono dei puntatori, bisogna riservare quindi 8 byte per l'indirizzo.
- È importante ricordarsi l'indirizzamento tramite

(base, indice, scala)

in cui l'indirizzo ottenuto è $base + indice * scala$. Questo è utile negli array.

25.2 Lab 10

25.2.1 Parte 1

- Le classi salvate in memoria hanno, a differenza dello stack, gli indirizzi nel verso opposto (sulla riga da sx a dx e in verticale dall'alto (0) verso il basso (+8/16/24/...))
- Nel mangling:
 - `_ZN` per nome della classe, deve essere seguito in ordine da numero caratteri e nome.
 - `C1` per il costruttore. Va posto dopo il nome della classe e prima del carattere `E` (vedi prossimo punto).
 - `E` indica la fine della sezione del nome.
 - a seguire la sezione parametri. Indicare in ordine i loro tipi seguendo le indicazioni:
 - * per i tipi semplici si usano i caratteri minuscoli

- * per puntatori, riferimenti e costanti si usano, rispettivamente P, R, K. Queste lettere maiuscole devono poi essere seguite da numero caratteri e nome se si riferiscono a strutture definite dall'utente.

- In tutte le funzioni membro di una classe c'è il parametro nascosto **this**, va incluso nello stack come tutte le altre variabili.

Nota: *Essendo un parametro viene passato in %RDI.*

- Agli elementi della classe ci accediamo sempre tramite il puntatore **this** (si può per esempio copiare in un registro per sfruttare l'indirizzamento (b, i, s) per un array della classe).

Nota: *Per capire bene come utilizzare la classe è bene capire che nello stack abbiamo this, questo, in quanto puntatore, punta all'inizio della classe (che è salvata in memoria), gli indirizzi all'interno della classe possono essere quindi acceduti, ad esempio, spostando il contenuto di this in un registro e poi utilizzare i parametri definiti con .set per la classe per riferirsi ai singoli elementi rispetto all'offset indicato sul registro in cui abbiamo copiato this.*

- Per il *mangling* di una funzione membro di una classe dopo ZN si ha numero cifre e nome della classe seguito da numero cifre e nome del metodo. Si chiude poi la sezione dei nomi con E e si inizia la sezione dei parametri.

- Quando allochiamo memoria nello stack per una funzione che contiene elementi di tipo classe possiamo vedere quanta memoria occupano guardando quanta ne avevamo allocata in precedenza per scrivere il costruttore.

Nota: *Quando poi andiamo a settare gli indirizzi all'interno delle righe riservate alla classe utilizzeremo, come visto sopra, indirizzi crescenti. Per riferirci all'elemento di tipo classe bisogna salvarsi l'indirizzo più in alto.*

- Per inizializzare un elemento di tipo classe basta fare la **call** al nome in *mangling* dato al costruttore prima (ricordarsi di passare prima i parametri in cui è sempre incluso il **this**. È sempre il primo parametro da passare ed è l'indirizzo "più alto" della classe, lo stesso che abbiamo utilizzato per la **.set**).

25.2.2 Parte 2

•

25.3 Lab 14

25.3.1 Parte 1

•

25.3.2 Parte 2

•

25.4 Lab 24 (semafori mutex)

Le prove sul nucleo possono essere divise in 3 tipologie:

- sincronizzazione/mutua esclusione
- I/O
- memoria virtuale

Distinzione tra funzioni e primitive

Tenere ben presente che la funzione indicata sul testo (es. `mutex_wait(..)`) è la funzione utente che chiama la INT, mentre `c_mutex_wait` è la primitiva nel nucleo.

Verifica parametri

- Quando si fa la verifica dei parametri dell'utente sfruttare la funzione `sem_valido/mutex_valido` che di solito si trova già pronta.
- Ricordarsi di abortire il processo che non ha superato il controllo degli input (`c_abort_p();`).

Gestione processi

- L'id del processo in esecuzione può essere ricavato tramite: `esecuzione->id`.
- Per inserire un processo in una lista c'è la funzione `inserimento_lista(lista, processo)` che fa inserimento ordinato.
- Lo schedulatore non fa altro che mettere il primo processo in coda pronti (tramite la funzione `rimozione_lista(des_proc*)` dentro la variabile `esecuzione`).
- Bisogna controllare che il processo in esecuzione non tenti 2 volte di acquisire il controllo sullo stesso mutex: `if(mutex->owner == esecuzione->id)`.
- Tramite la funzione `inspronti()`; si inserisce in lista pronti il processo attualmente in esecuzione.
- Per garantire la preemption si può fare l'estrazione di un processo dalla lista di attesa, fare `inspronti()`; e poi `inserimento_lista(pronti, proc_estratto)`.
- Dopo aver estratto dalla lista un processo, ricordarsi di aggiornare l'owner.
- Funzioni di gestione delle liste:
 - `rimozione/inserimento_lista(...)` eseguono il loro compito in modo ordinato.
 - `inspronti()` esegue l'inserimento sempre in testa.

Gestione degli errori

Per la gestione degli errori e mostrare a schermo messaggi di errore si utilizza la funzione `flog`, seguendo lo schema: `flog(LOG_WARN, "messaggio di errore \n");`

25.5 Lab 26 (05/02/2011 concessione di permessi scrittura/-lettura)

Note varie

- Quando una funzione (primitiva?) non chiama `schedulatore()` si può evitare l'utilizzo di `salva_stato` e `carica_stato`.
- Per lasciare il messaggio da leggere bisogna sovrascrivere il campo `esecuzione->contesto[I_RAX]`.

Parte 2 (18/01/2017 Broadcaster/Listener)

•

25.6 Lab 34 (02/07/2015 Memoria virtuale)

Parte 1

- Controlli da fare per la memoria virtuale:
 - validità degli indirizzi (controllo dello spazio di indirizzamento) verificabili tramite: `if(end < begin || end >= fin_utn_c)`.
- L'utilizzo della funzione `map` è supportato da un ampio commento. In sostanza la funzione deve mappare un indirizzo virtuale in un indirizzo fisico (deve quindi conoscere per ogni pagina quale indirizzo fisico punta), per farlo chiama la funzione `getpaddr` per creare la tabella di corrispondenza. Per i flag si possono usare le macro come `BIT_RW|BIT_US`.
- Per il campo `getpaddr` si può creare la classe `Funtore` con un metodo pubblico che restituisce un `paddr` dato un `vaddr`. Si ridefinisce l'operatore `()` come metodo pubblico.
- Nella classe `Funtore` si usa un campo privato `natl fn` per l'id del primo frame. Va aggiunta una variabile di appoggio:

```
natl cfn = fn;
fn = vdf[fn].next_shmem;
return cfn * DIM_PAGINA;
```

- Costruttore: `Funtore(natl n) : fn(n) {};`
- La funzione `unmap` può ricevere come argomento un secondo funtore vuoto.
- La `map` restituisce il primo indirizzo non mappato.
- Nella `unmap` l'indirizzo `end` sarà il return value della `map`.
- In questo esercizio non va invalidato il TLB perché `map` e `unmap` sono consecutive.

Parte 2 (23/07/2015 shmem anche questo)

- In questo caso bisogna invalidare il TLB poiché essendo stato mappato in precedenza non sappiamo se qualcuno ci ha acceduto.
- I funtori possono essere usati per invalidare il TLB durante una `unmap` tramite:

```
invalida_entrata_TLB(v)
```

25.7 Lab 38 (settembre 2021 Memoria virtuale)

Parte 1

- Per creare una view si deve prendere una view esistente e assegnarla al processo.
- Per convertire un `void*` in `vaddr`:

```
vaddr v = reinterpret_cast<vaddr>(vv);
```

- Quando si deve restituire `nullptr` basta fare `return`.

Parte 2

- Il funtore è una funzione che viene chiamata ogni volta che viene fatta la `unmap` su una pagina.
- Quello che fa una lambda non è altro che ricreare la sintassi di una classe con un funtore.
- Il TLB contiene informazioni sulle traduzioni del processo in esecuzione, quindi bisogna distinguere i casi in cui va invalidato.
- Con `invalida_entrata_TLB()` si deve passare un indirizzo virtuale.

25.8 Lab 42 (15/06/2016 I/O)

Parte 1

- `fill_io_gate` sono di tipo trap quindi non disabilitano le interruzioni.
- Il tipo delle primitive si può prendere da `utente.s`.
- Le primitive I/O non chiamano `carica_stato` e `salva_stato`.
- Si usa `abort_p` e non `c_abort_p()` perché siamo in un modulo diverso.
- Utilizzo della primitiva `access(...)` per verificare i problemi del cavallo di Troia.
- In `access` il parametro `shared` deve essere `true` se la memoria è condivisa.
- Quando operiamo in DMA lavoriamo sempre con indirizzi fisici.
- Controllare la versione della trasformata.
- Ricordarsi che nell'I/O le primitive girano a interruzioni abilitate.

Parte 2

- Si trova una `activate_pe(...)` che ha come primo parametro la funzione del processo esterno.
- La struttura interna del processo esterno è: `for(;;){... wfi();}`.
- Se `quanti == 0`, fare `sem_signal` sul `sync`.
- I processi esterni hanno priorità maggiore.

25.9 Lab 45 (06/07/2016 I/O)

Parte 1

- Trucco: usare la funzione `ce_init(...)` per capire i campi della struct.
- Usare un semaforo mutex per proteggere l'accesso alla struttura dati.
- Prima di leggere dal registro dei dati bisogna disattivare le int.

Parte 2

- Bisogna fare il controllo (con la `access`) anche su `quanti` perché viene passato come riferimento
- In questo caso non ci conviene utilizzare indirizzi fisici perché la periferica non opera in DMA, farlo è lecito ma complica inutilmente le cose
Nota: *Operare in DMA è necessario solo quando si ha che il dispositivo accede direttamente al bus e scrive direttamente in memoria, non si ha quindi nessuna traduzione.*

Prova 27/07/2016

- Per verificare che `buf` sia allineato alla pagina basta fare un `&` con gli ultimi 16 bit (`0xFFFF`).

25.10 Lab 46

Parte 1 (12/06/2019 Breakpoint)

- La funzione va dichiarata `void` perché i valori di ritorno vanno messi nel contesto.
- Creare una lista di attesa come indicato nel punto 2 del testo.

Parte 2 (16/02/2019 I/O head tail)

-

Capitolo 26

Alcune domande orale

Di seguito alcune domande fatte all'orale del primo appello estivo 2024. Le risposte non è detto siano giuste e sono solo uno spunto, la risposta deve essere più ampia. Altre domande si trovano su git-hub.

1. MECCANISMO DELLE INTERRUZIONI
 - a. A COSA SERVONO I COLLEGAMENTI TRA CPU ED APIC?
 - b. PERCHE' LA CPU NON DOVREBBE ACCETTARE IMMEDIATEMENTE LE INTERRUZIONI? Innanzitutto se le interruzioni sono abilitate controlla se è arrivata una richiesta e eventualmente l'accetta solo dopo aver completato l'esecuzione di un'istruzione; l'interruzione esterna non blocca il completamento di una istruzione come ad esempio fa un'eccezione di tipo fault. Inoltre può non accettare richieste di interruzione se IF=0, quindi le interruzioni sono disabilitate e questo può essere perché il processore sta eseguendo una serie di istruzioni importanti e non vuole essere interrotto.
 - c. SE IL PROCESSORE È A LIVELLO UTENTE ACCETTA SEMPRE LE INTERRUZIONI? Generalmente a livello utente siamo con le interruzioni abilitate e quindi queste vengono accettate, però nei sistemi INTEL potrei dare il permesso all'utente di eseguire le istruzioni cli e sti modificando da livello sistema il flag IOPL e quindi l'utente potrebbe disattivare le interruzioni.
 - d. E SE È A SISTEMA? Da livello sistema dipende, se abbiamo attraversato un gate di tipo interrupt staremo eseguendo una routine di sistema con le interruzioni disabilitate e in quel caso non le accetto. Se invece ad esempio voglio garantire la gestione annidata delle richieste di interruzione farò sì che dopo che ne ho accettata una e quindi starò eseguendo la routine relativa a livello sistema, questa girerà con le interruzioni abilitate e quindi ne potrei accettare un'altra con priorità maggiore.
 - e. QUANDO È CHE L'APIC INVIA UNA NUOVA RICHIESTA DI INTERRUZIONE? Si sfrutta i registri IRR e ISR dell'APIC, se arriva una richiesta dal piedino i si setta a 1 il piedino i-esimo nel registro IRR e se il bit corrispondente in ISR non è attivo l'apic inizia l'handshake con la CPU.
 - f. COME FA A SAPERE IL TIPO DELL'INTERRUZIONE? Nell'APIC l'associazione sorgente-tipo viene configurata via software dal programmatore. Invece la CPU saprà il tipo perché: gli viene passato dall'apic per le int. esterne, parametro della int per interruzione sw e implicito per le eccezioni.
2. CI SONO DELLE OCCASIONI IN CUI LA CACHE DEVE ESSERE DISATTIVATA QUALI? Penso intenda le operazioni con destinatario il modulo I/O, per cui il controllore cache non deve agire, e le operazioni che coinvolgono indirizzi ai quali sono mappati registri delle periferiche di I/O o registri dell'APIC. Forse si può rispondere anche parlando del DMA ma in quel caso si parla più di invalidare la cache secondo me.

 - a. COME FA LA CACHE A SAPERE QUALI IGNORARE? La MMU quando traduce gli indirizzi obiettivo di queste

operazioni da virtuali a fisici, nelle tabelle di livello 1 vede PCD=1 e ordina al controllore cache di lasciar passare inalterate queste operazioni.

1. (VOTO ALTO) COSA DEVE FARE E QUANDO VA IN ESECUZIONE L'HANDLER? Piccolo codice assembler che va in esecuzione quando arriva una richiesta di interruzione esterna da parte di una periferica, salva lo stato del processo che ha interrotto e manda in esecuzione il corretto processo esterno, prendendo il des-proc dall'entrata i della tabella a-p, con i piedino dell'apic su cui è arrivata la richiesta. a. GIRA AD INTERRUZIONI ABILITATE? No, fa parte del modulo sistema e farà salva e carica stato. b. È UNA BUONA IDEA FARLO GIRARE AD INTERRUZIONI ABILITATE? No, deve essere atomico, manipola le code dei processi. c. MA STIAMO FORZANDO IL PROCESSO ESTERNO AD ANDARE IN ESECUZIONE MA NON AVEVAMO GARANTITO CHE FOSSE SEMPRE IN ESECUZIONE QUELLO A MASSIMA PRIORITÀ? Si, ma è corretto perché avrò MIN-EXT-PRIO + prio con prio numero naturale minore di 256 (tipo della richiesta di interruzione). d. Perché METTIAMO IN TESTA A PRONTI IL PROCESSO PRECEDENTE E NON CI PREOCCUPIANO DI METTERLO IN MEZZO ALLA LISTA PRONTI? Perché dovrei metterlo in mezzo, devo gestire la preemption. La risposta credo sia perché se quel processo era in esecuzione era quello a priorità maggiore quindi lo rimetto in cima alla lista pronti per gestire la preemption (preemption che verrà completamente gestita poi con la wfi()).

1. (VOTO ALTO) WFI COS'È COSA FA? a. CODICE IN ASSEMBLY

```
1 a_wfi:  
2 call salva_stato  
3 call apic_send_eoi  
4 call schedulatore  
5 call carica_stato  
6 iretq
```

b. TOGLIAMO CALL SEND-EOI E LO METTIAMO NEL CODICE IN C SOPRA LA WFI() COSA CAMBIA? Si può fare, verrà inviato prima l'eoil all'apic, il quale assume che quella richiesta non è più pendente, è stata servita. c. COSA SUCCIDE SE ARRIVA UN INTERRUZIONE DALLA STESSA PERIFERICA TRA LA SEND-END-EOI E LA WFI()? Il flusso di esecuzione si ferma e si attraversa un gate, si salvano i valori e il RIP che viene salvato punta alla wfi no eoi(). Salva lo stato del processo e inserisce il processo in coda pronti e poi il nuovo processo esterno in esecuzione. Il problema ora è questo l'interruzione arrivata era della stessa dispositivo quindi l'handler ha messo in esecuzione lo stesso processo esterno. Ora avrà il processo esterno sia in esecuzione che in coda pronti. L'handler continua e mi carica lo stato nel descrittore e poi fa la iretq, mettendo in RIP l'indirizzo della wfi no eoi() salvata prima. Quindi il processo esterno chiama la wfi no eoi() e salva in pila RIP che punta dopo la wfi, quindi all'inizio del ciclo for, e chiama schedulatore che metterà in esecuzione lo stesso processo esterno che però era rimasto in lista pronti. Fa la carica stato e la iretq e riparte il ciclo. Quindi passando per stati inconsistenti ma alla fine siamo finiti in uno stato consistente. d. QUALE RIP VIENE SALVATO? IL PROCESSO ESTERNO SARÀ ANDATO IN CODA PRONTI È UN PROBLEMA? NO PERCHE IL PROCESSO ESTERNO PUÒ ESSERE MESSO IN CODA PRONTI DA UN ALTRO PROCESSO ESTERNO A PRIORITA' MAGGIORE e. CHE SUCCIDE POI? ESECUZIONE PUNTERÀ ALLO STESSO PROCESSO ESTERNO IN CODA PRONTI f. ALLA FINE ABBIAMO AVUTO DEI PROBLEMI? NO

1. MECCANISMO INTERRUZIONI a. DISEGNO APIC 24 piedini di ingresso, per ogni

piedino associato un tipo, se il segnale deve essere riconosciuto sul fronte o sul livello e se alto o basso e se interruzione mascherabile o no. Poi disegno collegamenti con CPU. b. IDT c. CHI DECIDE IL TIPO? IL SISTEMA LO SA GIA VUOL DIRE CHE QUALCUNO L'HA SCRITTI OSSIA CHI HA PROGETTATO LA CPU d. COME GESTISCE LA PRIORITÀ I 4 bit più significativi del tipo indicano la classe di priorità. e. COME FA A SAPERE CHE QUELLA PRECEDENTE HA TERMINATO Quando il software scrive in EOI l'APIC assume che è terminata la richiesta di interruzione che era in corso con priorità più alta. f. QUANDO è CHE L'APIC SI ACCORGERÀ CHE È IN ARRIVO UNA NUOVA RICHIESTA SULLO STESSO PIEDINO Se i in ISR è attivo e ne arriva un'altra verrà settato i in IRR, una terza verrebbe persa. Oppure rispondere con riconoscimento sul fronte o sul livello -> meglio questo. g. QUANDO è CHE LA PERIFERICA CONSIDERA CHE LA RICHIESTA È STATO SERVITA Per la periferica quando si ha la risposta alla richiesta, ad esempio a seguito della lettura di un registro della periferica. Invece poi la richiesta sarà completamente servita quando viene scritto un valore in EOI.

1. (VOTO ALTO) COSA È IL TLB E PERCHÉ DOBBIAMO INVALIDARLO Piccola cache esclusiva per la MMU che contiene le traduzioni accedute più di recente, per traduzioni si intendono solo i descrittori di livello 1, contenenti il numero di frame. Memorizza oltre alla traduzione anche alcuni flag che servono alla MMU per eseguire tutte le sue funzioni senza dover fare il table-walk. a. Se cambia il diritto di scrittura in una tabella di liv 1 quante entrate del TLB VANNO INVALIDATE? 1. In realtà dipende come lo cambio, se era attivo e poi lo disattivo devo invalidare l'entrata. Se era disattivato e lo attivo non devo invalidare l'entrata perché l'operazione di scrittura avrà comunque causato una miss. Altro caso in cui devo invalidare le entrate è quando faccio una unmap su un intervallo di indirizzi del processo in esecuzione, in quel caso faccio un for e faccio unmap delle entrate relative a quell'intervallo di indirizzi. b. SE LA INVALIDO IN UNA ENTRATA DI LIV 2? 512 c. È VERO CHE DEVO INVALIDARE SEMPRE TUTTO IL TLB QUANDO CAMBIO IL PROCESSO? Il TLB contiene le traduzioni accedute più di recente del processo attualmente in esecuzione, se cambia il processo in esecuzione e cambia quindi l'albero di traduzione attivo devo invalidare tutto il TLB -> quindi direi di sì (?).

Processo esterno con privilegio utente, cosa non puo' fare senza modifiche al codice? Non può accedere ai registri della periferica, sono mappati nella zona di memoria riservata all'I/O alla quale lui non ha accesso, quindi non può inviare l'EOI, non può disabilitare le interruzioni. Dovrei settare il flag IOPL per permettere all'utente di fare cli e sti e mappare i registri nella zona di memoria accessibile all'utente, quindi in conclusione non posso avere un processo esterno a livello utente perché non posso dare all'utente la possibilità di fare queste cose.

Ottimizzazioni e rallentamenti nella coda con i contatori già decrementati? Ottimizzo il driver del timer che così deve decrementare solo il primo elemento, ma peggioro la delay che deve fare l'inserimento ordinato in lista.

DMA in serie alla mmu risolve il problema degli indirizzi fisici dell'io? Non funziona perché in un sistema di multiprocesso la radice dell'albero di traduzione punta all'albero del processo in esecuzione, quindi il DMA mentre scrive e passa per l'MMU e avviene un cambio di processo gli indirizzi non sarebbero più validi. Se mentre è in corso il trasferimento DMA cambia il processo in esecuzione, cambia l'albero di traduzione e quindi gli indirizzi virtuali che stava usando il dispositivo DMA non sono più validi. Può essere risolvibile solo se i processi usano le stesse traduzioni o se il dispositivo scrive solo nelle parti condivise.