Appunti Calcolatori Elettronici

Machkour Mohamed
calcolatori.elettronici@gmail.com

29 aprile 2009

Indice

1 Organizzazione dell'informazione all'interno di un calcolatore 5
1.1 Elaborazione & informazione 5

2 Architettura di un calcolatore con riferimento al processore PC 6
2.1 Aspetti fisici del Processore 6
2.1.1 BU 6
2.1.2 PU 6
2.1.3 ALU & FPU 7
2.1.4 MMU 7
2.1.5 Organizzazione degli spazi di I/O 7
2.1.6 Operandi a 16bit e 32bit 9
2.1.7 ciclo di bus 9
2.2 Circuito di pilotaggio 9
2.2.1 latch 10
2.2.2 Transceiver 10
2.3 Circuito di controllo 11
2.3.1 circuito di abilitazione 11
2.3.2 Circuito di comando 11
2.4 Spazio di I/O 12
2.4.1 Spazio esterno a 16 bit 12
2.4.2 Spazio esterno a 8 bit 13
2.5 Memoria fisica e interfaccia 13
2.5.1 Memoria fisica 13
2.5.2 Interfacce 14
2.5.3 Memoria dinamica 15
2.6 Memoria cache 16
2.6.1 Organizzazione della memoria cache 17
2.6.2 Controllore della memoria cache 19
2.6.3 Montaggio del controllore cache 20
2.6.4 Memoria cache associativa a insieme (set-associativa) 21
2.7 Interruzione di programma 21
2.7.1 Istruzioni asincrone e sincrone 22
2.7.2 Tipo d'interruzione 22
2.7.3 Tabella delle interruzioni e azioni del processore 23
2.7.4 Descrittore della IDT 23
2.7.5 Riconoscimento del tipo per le richieste di interruzioni esterne mascherabili 24
2.7.6 Gate non presente 25
2.7.7 Routine di servizio 25
2.7.8 Controllore delle interruzioni 26
2.7.9 Montaggio del controllore dell'interruzioni . . . 27
2.7.10 Controllore dell'interruzione e memoria cache . 28
2.7.11 Fully Nested delle interruzioni mascherabili . . 28
2.7.12 Schema di una routine di servizio 28
2.7.13 Collegamento di piú controllori in cascata . . . 30
2.7.14 Modalitá di funzionamento Special Fully Nested 31
2.7.15 Operazioni di I/O a interruzione di programma 32
2.8 DMA 37
2.8.1 Politica di gestione del bus 37
2.8.2 Controllore DMA 38

2.8.3 Cablaggio del controllore DMA 38
2.8.4 Trasferimenti in DMA 40
2.8.5 Modo di trasferimto del controllore DMA . . . 42
2.8.6 Tipo di ciclo del controllore DMA 43
2.8.7 Problema del corto circuito 43
2.8.8 Trasferimento memoria-memoria in DMA . . . 44
2.8.9 Controllore Cache e controllore DMA 45
2.8.10 Interfaccia gestita in DMA 45
2.8.11 Operazione di ingresso in DMA 47

3 Aspetti Architetturali avanzzati 51
3.1 Memoria virtuale Paginata 51
3.1.1 Tipologia di spazi di memoria 51
3.1.2 Paginazione 52
3.1.3 Meccanismo per la paginazione 53
3.1.4 Generalizzazione della paginazione 55
3.1.5 Memory Management Unit e Page-fault 55
3.1.6 Riesecuzione delle istruzioni 56
3.1.7 Memoria fisica in memoria virtuale 57
3.1.8 Paginazione nel processore PC 57
3.2 Gestione della tabella di corrispondenza 59
3.2.1 Descrittore di tabella e di pagina 59
3.2.2 Trasferimento delle pagine 60
3.2.3 Trasferimento delle tabelle delle pagine 61
3.2.4 Buffer dei descrittori di pagina 61
3.2.5 Controllore DMA e indirizzi fisici 64

4 Nucleo di un sistema multiprogrammato 65
4.1 Multiprogrammazione 65
4.1.1 Spazio di indirizzamento di un processo 66
4.1.2 Descrittori di processo nel processore PC . . . 67
4.1.3 Commutazione hardware fra processi. 68
4.1.4 Processi bloccati 69
4.2 Protezione 70
4.2.1 Stati di funzionamento 70
4.2.2 Protezione nel processore PC 72
4.2.3 Meccanismo di interruzione e protezione 72
4.2.4 Il problema del cavallo di Troia 75
4.3 Sistemi Multiprogrammati 76
4.3.1 Modello di un processo 76
4.3.2 Commutazione di contesto 78
4.3.3 Nucleo e interruzioni 79
4.4 Preccessi e interruzioni 80
4.4.1 Realizzazione delle primitive 81
4.4.2 Realizzazione dei processi 83
4.4.3 Processo dummy 85
4.4.4 Corpo delle primitive 86
4.4.5 Atomicitá 90
4.4.6 Semafori 91
4.4.7 Realizzazione dei semafori 92
4.4.8 Mutua esclusione (impiego d) 94
4.4.9 Sincronizzazione 94
4.4.10 Memoria dinamica 95
4.4.11 Utilizzo delle interfacce di conteggio 96

4.5 Operazione di I/O 99
4.5.1 Ingresso/Uscita a interruzione di programma . 101
4.5.2 Descrittori di operazione di I/O 102
4.5.3 Operazione di lettura 104
4.5.4 Operazione di scrittura 106
4.6 Processi Esterni 108
4.6.1 Driver e processi esterni 108
4.6.2 Struttura di un processo esterno 109
4.6.3 La primitiva wfi () 111
4.6.4 Lettura con i processi esterni 112
4.6.5 Scrittura con i processi esterni 113
4.7 Ingresso / Uscita in DMA 114
4.7.1 Accesso diretto alla memoria 115
4.7.2 Lettura in DMA 117
4.7.3 Scrittura in DMA 118
4.8 Multiprogrammazione e Personal Computer 120
4.8.1 Interruzione e la primitiva nwfi() 120
4.8.2 Gestione dell'interfaccia seriale 120
4.8.3 Operazione di lettura con la porta seriale . . . 122
4.8.4 Operazione di scrittura con la porta seriale . . 123

5 Segmentazione e Multiprogrammazione 125
5.1 Memoria Segmentazione 125
5.1.1 Segmenti nel processore PC 126
5.1.2 Registri e istruzioni in ambiente segmentato . . 126
5.1.3 Traduzione degli indirizzi 127
5.1.4 Traduzione degli indirizzi nel processore PC . . 128
5.1.5 Struttura dei registri selettore 129
5.1.6 Selettore nullo 129
5.1.7 Trasferimento di segmenti 130
5.1.8 Attivazione del modo protetto 131
5.1.9 Meccanismo di interruzione 132

6 Conclusione 133

A Appendice 134
A.1 Gestione della Memoria Virtuale Paginata 134

1 Organizzazione dell'informazione all'interno di un calcolatore

1.1 Elaborazione & informazione

I calcolatori digitali sono un aggregato di meccanismi e circuiti in grado di elaborare informazioni in modo performante. Lo scambio di informazioni con l'esterno avviene mediante la codifica esterna dell'informazione sequenza di caratteri:

* Lettere
* Cifre
* Segni di interpunzione
* ecc.

La codifica interna é costituita da sequenza di simboli binari (BInary digiT). Ogni carattere esterno viene codificato (su 7 o 8 bit) in maniera automatica in ingresso(l'operazione inversa é effettuata del medesimo agente di conversione verso l'uscita/e).

Vengono in genere compiute ulteriori conversioni di codifica visto che la legge di codifica non e' una sola. L'elaborazione dunque deve avvenire, in maniera consistente sulla codifica.

I numeri , una classe di informazione, vengono convertiti in base 2 su un numero fisso di bit. I numeri naturali sono rappresentati da una configurazione di N bit in notazione posizionale

$$A = a_{n-1}a_{n-2}\ldots a_0 = \sum_{i=0}^{N-1} 2^i * a_i \tag{1}$$

Dove A e' il un numero il cui valore é dato dalla somma pesata dei vari bit a_i con peso 2^i. Risulta dunque possibile rappresentare con N bit i seguenti numeri naturali $0\ldots$ a $2^N - 1$.

.

2 Architettura di un calcolatore con riferimento al processore PC

Per famiglia 32/PC si intende una serie di moduli circuitali mediante i quali é possibile strutturare un semplice calcolatore. Si esamineranno i sistemi **monoprocessore**. :

I moduli sono una rappresentazione schematica dei reali integrati con visione logica di funzionamento

1. Processore.
2. Circuito di pilotaggio.
3. Circuito di controllo.
4. Spazio di I/O.
5. Memoria principale e interfacce.
6. Memoria cache.
7. Controllore delle interruzioni.
8. DMA.

2.1 Aspetti fisici del Processore

Il processore PC é costituito da cinque unitá

1. BUS Unit (**BU**)
2. Prefech Unit ((**PU**)
3. Arithmetic & Logic Unit (**ALU**)
4. Floating Point Unit (**FPU**)
5. Memory Management Unit (**MMU**)

2.1.1 BU

Effuttua gli accessi ai spazi esterni mediante dei cicli di bus per :

- Prelevare Istruzioni (in fase di chiamata)
- Prelevare / Immettere Dati (in fase di esecuzione)
- Effettuare cicli di risposta alle richieste di interruzione (Vedi avanti *)

2.1.2 PU

Gestisce una coda interna di istruzioni effettuando il prelievo anticipato delle stesse, e li invia alla ALU o FPU in base al discriminante rappresentato dalla combinazione di bit 11011 (ESC) con cui il codice operativo (OPCODE) inizia. Le istruzioni il cui OPCODE inizia con ESC vengono inviati alla FPU. Ogni volta che non é richiesto l'utilizzo del bus la BU effettua la lettura dalla memoria, in base a criteri che verranno analizzati in seguito, una porzione di codice e la memorizza all'interno della coda di *Prefech*. La *PU* preleva le istruzioni dalla coda man mano che queste devono essere eseguite, svuotandola. Provvede a svuotarla la coda ogni volta che viene caricato un nuovo valore in ESP, per effetto di un istruzione di salto, o per il meccanismo di interruzioni.

2.1.3 ALU & FPU

Queste due unita operano in parallelamente in concomitanza, ed eseguono due sottoinsiemi di istruzioni disgiunti. La PU preleva le istruzioni da un unico flusso e li invia all'apposita unit: Viene rispettato il paradigma del Produttore-Consumatore, ovvero l'invio della successiva istruzione avviene soltanto quando l'unitá coinvolta ha terminato l'esecuzione della precedente istruzione. Quindi viene rispettata una certa sequanzialitá . Esiste un sincronismo fra le due unitá. Quando un istruzione richiede/produce un dato dalla/verso la memoria o in un registro della ALU, la FPU attende che il prelievo o memorizzazione sia terminata prima di procedere.

2.1.4 MMU

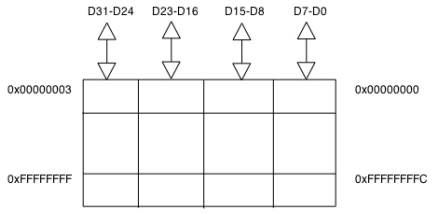
Opera in presenza della *memoria virtuale*.

2.1.5 Organizzazione degli spazi di I/O

Il processore PC rientra nella categoria di quelli a 32 bit, ovvero che puó trasferire in unico ciclo fino a 4 byte. Esso prevede pertanto 32 piedini $d_{31} - d_0$ Gli spazi esteri a cui il processore é in grado di accedere sono :

- spazio di memoria 4Gbyte ($A_{31} - A_0$)
- spazio di I/O 64Kbyte($A_{15} - A_0$)

Dove $A_i - A_0$ sono i piedini di indirizzo. Il processore vede lo spazio esterno organizzato a 32bit, questo puó comportare problemi di compatibilitá. Un insieme di 4 byte consecutivi di cui il primo byte ha un indirizzo multiplo di 4 si dice *linea*.



Nei singoli cicli di bus possono essere trasferite singole linee o porzioni di essa (una piú byte consecutive all'interno della linea)
L'accesso allo spazio di memoria avviene durante la fase di prelievo delle istruzioni e durante la fase di esecuzione della stessa che hanno almeno un operando in memoria.

La linea coinvolta nel trasferimento é individuata dai piedini A31-A2. I byte coinvolti nel trasferimento all'interno della linea vengono abilitati singolarmente dai piedini /BE3-/BE0 (BE = BUS ENABLE), se sono piú di un byte necessariamente sono consecutivi.

il processore genera gli indirizzi come l'insieme di A31-A2, /BE3-/BE0, lo spazio di I/O prevede peró A31-A0

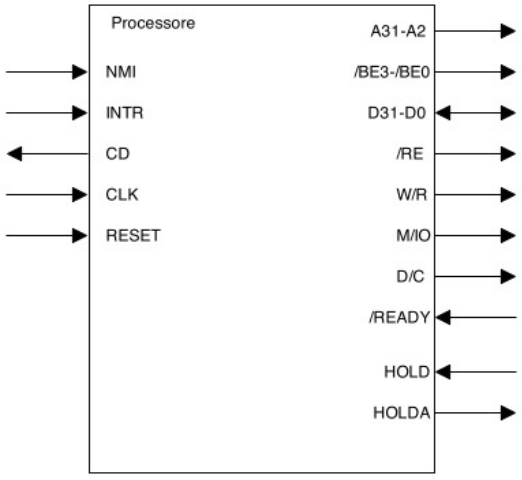
l'indirizzo di linea
+
i segnali di abilitazione
⇓

Individuano univocamente l'indirizzo del primo byte (quello di indirizzo piú piccolo) coinvolto nel trasferimento Tale indirizzo e' deteminabile secondo la seguente tabella

/BE3	/BE2	/BE1	/ BE0	A1	A1
-	-	-	0	0	0
-	-	0	1	0	1
-	0	1	1	1	0
0	1	1	1	1	1

***Nota* 1** *in base al valore dei non specifici si determina quali sono i byte che seguono quello di indirizzo minore sono coinvolti nel trasferimento purché siano consecutivi.*

- A31-A2: sopportano l'indirizzo di una linea durante i cicli di bus.
- /BE3-/BE0: (BUS Enable) abilitano i singoli parti del bus dati (D31-D24,D23-D16,D15-D8,D7-D0)
- D31-D0: sopportano i dati durante i cicli bus.
- /RE: specifica l'inizio di un ciclo di bus.
- W/R : specifica il tipo di ciclo bus (lettura/ scrittura)
- M/IO: specifica a quale spazio interessa il tipo di ciclo bus.
- D/C: (data/control) durante un ciclo di lettura dello spazio di memoria specifica se il trasferimento é di tipo dati o istruzioni.
- /READY: (BUS RAEADY) indica che il processore puó terminare un ciclo di bus.
- NMI: (Not Maskable Interrupt request) indica una richiesta di interruzione non mascherabile.
- INTR: (INTerrupt Request) indica un richiesta di interruzione mascherabile.



- HOLD: (BUS Hold Request) indica una richiesta di utilizzo del bus(Una qualunque unitá che avverte il processore della necessitá di utilizzo del bus).
- HOLDA: (bus Hold Acknowledge) indica l'accettazione di una richiesta di utilizzo del bus.
- CD: (Cache Disable) Serve a disabilitare la memoria cache.

Durante il prefech viene letta sempre un intera linea di codice. Durante l'esecuzione un istruzione puó richiedere operandi che stanno su linee differenti e in tal caso il processore effettua il trasferimento in cicli distinti specificando per ogni ciclo l'indirizzo della linea e i byte da trasferire all'interno della linea stessa mediante i segnali di abilitazione del bus.

Nota 2 *Le quantitá che si trovano o vengo posti nei registri del processore devono subire opportune traslazioni rispetto alla porzione del bus dati coinvolto nel trasferimento.*

2.1.6 Operandi a 16bit e 32bit

⇒ una *parola* é allineata se l'indirizzo del primo byte é multiplo di 2.

⇒ una *parola lunga* é allineata se l'indirizzo del primo byte é multiplo di 4.

Parole e parole lunghe si possono sempre trasferire in unico ciclo di bus. Il piedino /RE segnala l'inizio di un ciclo di bus. W/R, M/IO, D/C, segnalano il tipo di ciclo.

D/C	M/IO	W/R	ciclo
1	1	1	mem. operand write
⋮	⋮	⋮	
0	**0**	**0**	interrupt ack

La risposta ad una richiesta di interruzioni equivale alla lettura nello spazio di I/O.

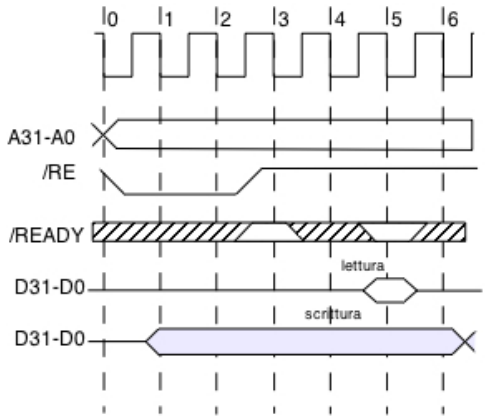
2.1.7 ciclo di bus

Un ciclo di bus dura un numero prefissato di tempi di clock. Il ciclo viene allungato di un ulteriore tempo di clock se alla fine del ciclo il segnale /READY non e' attivo. Si suppone che la durata di un ciclo si di 4 tempi, e che l'esame del segnale di /READY venga effettuata a partire dal terzo.

2.2 Circuito di pilotaggio

Nota: la funzione del latch e del trasceiver é quella di rigenerare i segnali elettrici.

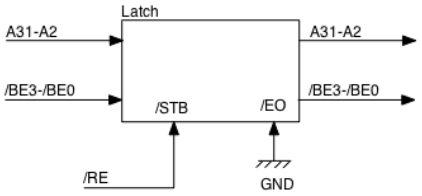
Sul bus indirizzi (A31-A2, /BE3-/BE0) viene montato un registro *latch* che memorizza un nuovo indirizzo ogni volta che inizia un ciclo di bus. Sul bus dati (D31-D0) viene montato un buffer bidirezionale *transceiver* che lascia passare i dati nel senso e nel tempo richiesto



dal tipo di ciclo di bus. Il latch e il transceiver costituiscono il circuito di pilotaggio: fra il processore e questo circuito vengono montati dispositivi particolari (DMA), mentre a valle si ha il bus vero e proprio.

2.2.1 latch

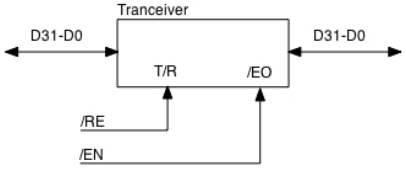
il lactch memorizza i dati di ingresso e li trasferisce in uscita quando arriva il segnale sul piedino di strobe (piedino STB). Inoltre l'uscita diviene inattiva (Hz) quando il piedino /EO (Output Enable) diviene inattivo. In sistemi monoprocessore questo il piedino /EO viene



collegato a massa visto che non ci sono dispositivi che producono indirizzi oltre al processore stesso. Il DMA viene collegato a monte del latch ne segue che necessariamente il piedino /EO deve essre sempre attivo.

2.2.2 Transceiver

Il transceiver lascia passare i dati in una delle due direzioni a seconda del valore del piedino T/R (Transmit/Receve). Inoltre l'uscita viene posto in alta impedenza quando il piedino /EO diviene inattivo. A monte del transceiver vengono collegati altri dispositivi che scambiano informazioni con il processore utilizzando il bus dati (DMA).



Pertanto le uscite del transceiver devono essere poste in alta impedenza ogni volta che avviene un tale utilizzo e il segnale /EN (ENABLE) viene generato esplicitamente.

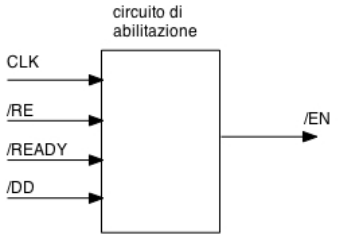
2.3 Circuito di controllo

I cicrcuiti di abilitazione e comando costituiscono i circuiti di controllo. il circuito di abilitazione ha il compito di generare il segnale /EN che abilita il transceiver presente sul bus dati.

2.3.1 circuito di abilitazione

Il segnale /EN dipende da:

- /RE : che diviene attivo all'inizio di un ciclo di bus (inizio).
- /READY : che determina la fine del ciclo di bus (fine).
- CLK: ovvio.
- /DD: (Data Disable) in caso di trasferimenti a monte del circuito di pilotaggio. La variabile /DD diviene attivo quando avviene tale trasferimento.



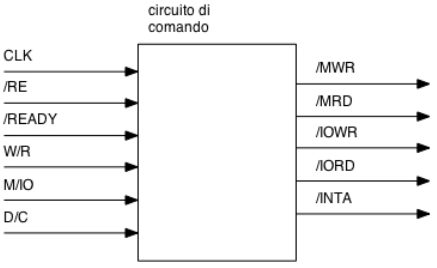
2.3.2 Circuito di comando

I segnali W/R, M/IO, D/C che specificano il tipo di ciclo, e vengono collegati ad un circuito di comando, che genera in forma temporizzata i segnali di comando presenti sul bus.

***Nota* 3** *la risposta alla richiesta di interruzione /INTR viene generata dal cicrcuito di comando.*

D/C	M/IO	W/R	Ciclo
1	1	1	M:op. Write
1	1	0	M. op. Read
1	0	1	IO Write
1	0	0	IO Read
0	1	0	Memory Code Read
0	0	0	Inerrupt Acknowledgement

***Nota* 4** *D/C vale Control solo quando il segnale W/R vale Read.*

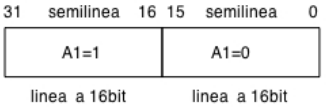


2.4 Spazio di I/O

2.4.1 Spazio esterno a 16 bit

Nell'ipotesi che lo spazio di I/O sia di 64Kbyte ($2^6 * 2^{10} = 2^{16}$, puó essere organizzato a 16bit, ne segue che il bus dati a 32bit una linea risulta suddivisa in due semilinee

- la semilinea *bassa* D15-D0.
- la semilinea *alta* D31-D16.

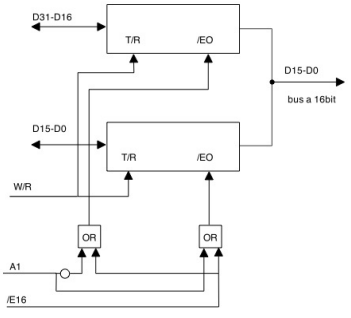


individuate dall'indirizzo di linea A31-A2 e dal bit A1. Nel bus dati a 16bit una linea é un insieme di due byte consecutivi. Il primo dei quali ha un indirizzo multiplo di 2.

***Nota* 5** *In base al valore di A1 deve essere effettuata una commutazione di contesto fra la linea del bus a 16 bit e una delle semilinee del bus a 32 bit*

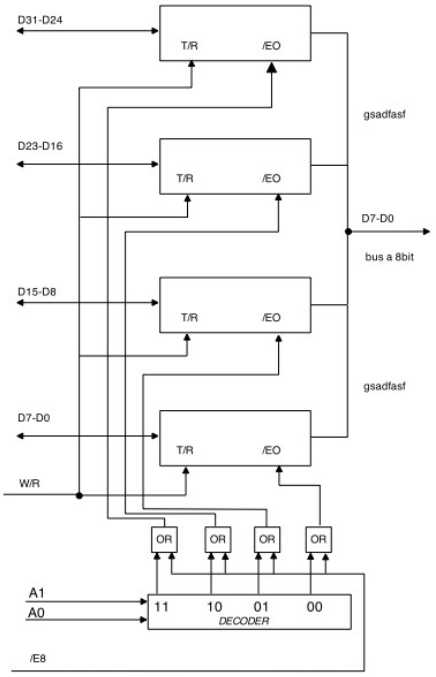
transceiver plexato
I singoli byte del bus dati a 16 bit vengono abilitati da due segnali /BHE, /BLE, che possono essere ricavati dai /BE3-/BE0,secondo la seguente tabella.

/BE3	/BE2	/BE1	/BE0	/BHE	/BLE=A0
-	-	1	0	1	0
-	-	0	1	0	1
-	-	0	0	0	0
1	0	1	1	1	0
0	1	1	1	0	1
0	0	1	1	0	0



2.4.2 Spazio esterno a 8 bit

Una porzione dello spazio esterno puó essere organizzato a 8 bit. Ogni linea del bus dati a 32 bit risulta suddivisibile in 4 sottolinee : D31-D24, D23-D16, D15-D8, D7-D0.Tale sottolinee sono individuate dai bit A1-A0.



Le quatto sottolinee corrispondono ad una linea di un byte (D7-D0) nel bus dati a 8 bit. Con la solita regola vista prima un circuito deve effettuare la commutazione in base ai valori di A1-A0.

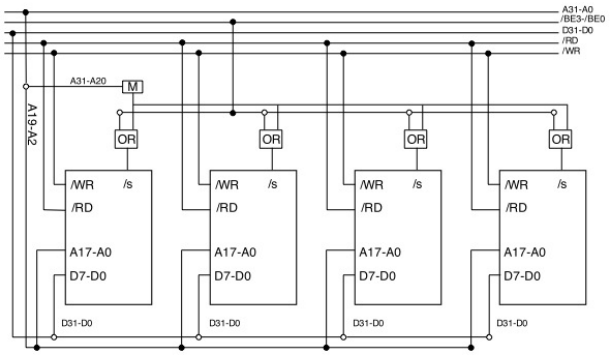
2.5 Memoria fisica e interfaccia

2.5.1 Memoria fisica

I banchi di memoria vengono montati nello spazio di memoria e montati sul bus dati a 32 bit.

***Nota* 6** *Un banco di memoria deve essere costruito in modo tale da essere accessibile a uno o piú byte consecutivi di una stessa linea.*

***Esempio* 2.1** *Memoria da 1024 kbyte suddivisa in 4 sotto banchi da 256 byte ciascuno, accessibili al byte. La maschera M riconosce*



una delle possibili combinazioni dei 12 bit piú significativi dell'indirizzo di linea A31-A20 determinando in quale delle 2^{12} porzioni dello spazio ciascuno da 256 klinee é effettivamente posizionato il banco di memoria. L'indirizzo all'interno della memoria é determinato dai 18 bit meno significativi dell'indirizzo di linea A19-A2. I byte sono selezionati utilizzando i segnali di abilitazione /BE3-/BE0 che in OR con l'uscita della maschera M (uscita attiva bassa) costituiscono il select del banco.

2.5.2 Interfacce

Le interfacce possono essere

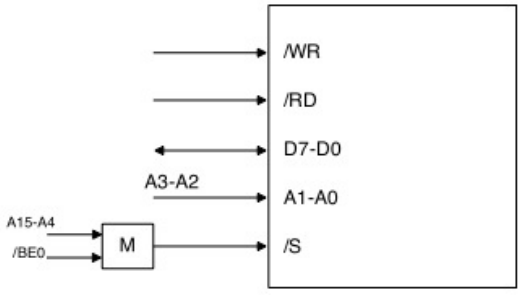
- memory mapped I/O
- I/O mapped I/O

I registri interni vengono riferiti nel primo caso con istruzioni che operano nella spazio di memoria, e nel secondo caso con istruzioni che operano nello spazio di I/O. Le interfacce piú semplici sono accessibili al byte(D7-D0). Vengono collegati alla parte meno significativa a qualsiasi tipo di bus dati. Gli indirizzi dei registri sono visibili al programmatore come segue:

$$\text{ind. programmatore} = \text{ind. base} + b * \text{ind. interno} \tag{2}$$

dove indirizzo_base si ottiene aggiungendo in testa i bit mancanti pensati uguali a 0 all'indirizzo riconosciuto dalla maschera. b é il numero di byte del bus.

***Esempio* 2.2** *Interfaccia di un interfaccia accessibile al byte ad un bus di 32 bit. I registri si trovano su parti corrispondenti di piu' linee consecutive a partire dall'indirizzo di base della prima linea individuata dai valori di A15-A2. Se sta parlando direttamente ad un bus a 32 bit , ne segue che ogni registro viene mappato su linee come descritto precedentemente. L'indirizzo all'interno della linea (A1-A0) viene dato dal segnale di abilitazione /BE0.Consideriamo un interfaccia con 4 registri interni collegata (ovviamente) alla parte piú bassa del bus dati. I bit A3-2 dell'indirizzo di linea indirizzano il registro*



Una linea é indirizzato dai A31-A2

coinvolto nel trasferimento. Se la maschera produce 0 quando A15-A4 = 0b011000111110 e /BE0 = 0 (A1-A0 = 0b00) l'indirizzo

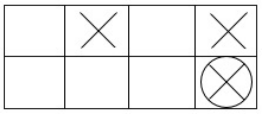
- *base = 0x63E0*
- *reg(0) = 0x63E0*
- *. . . .*

- *reg(3) = 0x63EC*

Per il bus dati a 16 si possono fare considerazioni analoghe. L'interfaccia é accessibili al byte per problemi di compatibilitá sono montate sul bus dati a *8 bit*, in modo che i registri interni abbiano indirizzi consecutivi. Consideriamo l'interfaccia accessibile al byte, collegata ad un bus dati a 8 bit. A15-A2 costituiscono gli ingressi della maschera M (ovvio ci si aspetta che sia coinvolta una sola linea), A1-A0 vanno connessi con gli omonimi ingressi dell'interfaccia, per indirizzare i vari registri. Gli indirizzi dei registri sono quindi consecutivi (0x63E0,. . ,0x63E3). e il valore dei registri viene a trovarsi, nel bus dati a 32 bit, in una delle 4 sottolinee di quest'ultimo. Questo ci é garantito dal segnale di abilitazione /BE8 e ai valori di A1-A0 necessari nel pilotaggio dei vari transceiver per un corretto demultiplexing sulla sottolinea interessata nel trasferimento. Il programma

il motivo per cui sono previsti i bus dati a 16 e 8 bit

un programma scritto per un bus a 32 bit ha problemi a girare su 16bit



che gira si aspetta il valore del registro nella seconda linea invece il processore appartiene alla famiglia a 16 bit, e il valore si trova su parti corrispondenti di linee composte da due byte consecutive.

Nota 7 *I banchi di memoria e le interfacce devono avere i pin dati con logica 3-state.*

2.5.3 Memoria dinamica

- le memorie statiche utilizzano come elemento di cella un elemento attivo
- le memorie dinamiche utilizzano come elemento di cella un condensatore (elemento passivo), con consumo ridotto, e ha un organizzazione a piani Nx1. Le celle del piano sono organizzati a matrice e l'indirizzo, di una di esse é suddivisa in due componenti
  - ***** l'indirizzo di riga
  - ***** l'indirizzo di colonna

  e vengono forniti alla memoria in sequenza utilizzando gli stessi collegamenti, in modo multiplexato collegati in base ai segnali /RAS (row)e /CAS (column)

Sono solo 10 bit grazie a /RAS /CAS.
Operazioni di lettura:

- W/R=0
- si specifica l'indirizzo di riga (/RAS = 0) tutta la riga viene selezionata.
- viene specificato l'indirizzo di colonna é attivo /CAS.
- il piedino D0 (in alta impedenza) assume il valore immagazzinato nella cella.

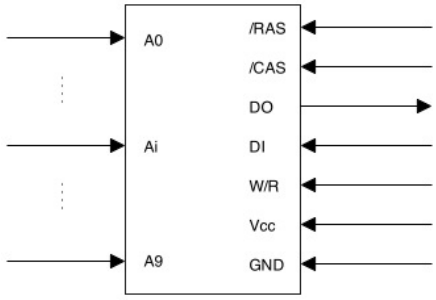


Figura 1: memoria dinamica 1Mx1

Operazioni di scrittura:
analoga all'operazione di lettura.

***Nota* 8** *W/R va a 1 dopo che si é attivato il /RAS consentendo la lettura di tutta la riga, e prima che si attivi il segnale /CAS consentendo al piedino D0 di rimanere nello stato di alta impedenza.*

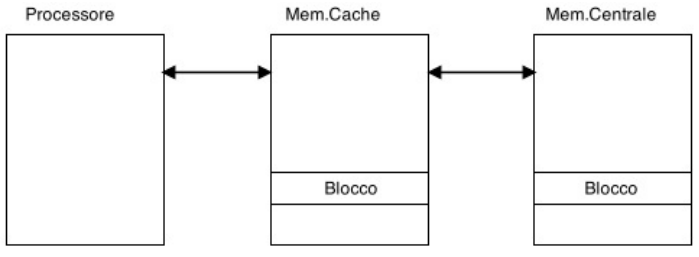
La memoria dinamica richiede un controllore:
questo si interfaccia con il bus comportandosi come una memoria statica e gestisce la memoria dinamica effettuando

- operazioni di lettura / scrittura comandate dal processore
- operazioni di *rinfresco* (con una certa frequenza il controllore deve rinfrescare tutte le righe della memoria. deve essere previsto un contatore che genera tutti gli indirizzi in sequenza delle righe.

***Nota* 9** *se il processore vuole effettuare operazioni di lettura / scrittura durante questo periodo (di rinfresco) il controllore* non *deve attivare il segnale /READY. Ne segue un allungamento del ciclo comandato dal processore.*

2.6 Memoria cache

Nel caso generale i programmi rispettano il cosiddetto *principio di localitá* che si esplicita nello spazio e nel tempo.



- **localitá sequenziale:** quando é probabile che venga riferita la locazione sequenzialmente successiva alla locazione o piú locazioni precedentemente riferite.
- **localitá spaziale:** come al punto recedente con l'assenza del vincolo di sequenzailitá.

- **localitá temporale** probabilitá che venga riferita la stessa locazione in tempi strettamente vicini.

In accordo a questo principio si stabilisce una gerarchia nella memoria con almeno due livelli:

- **livello inferiore:** memoria centrale vera e propria
  le informazioni presenti sul livello superiore sono presenti in genere anche sul livello inferiore.
- **livello superiore:** memoria cache
  piú veloce, piú costosa. piú piccola contiene le informazione che hanno la maggior probabilitá di essere riferite

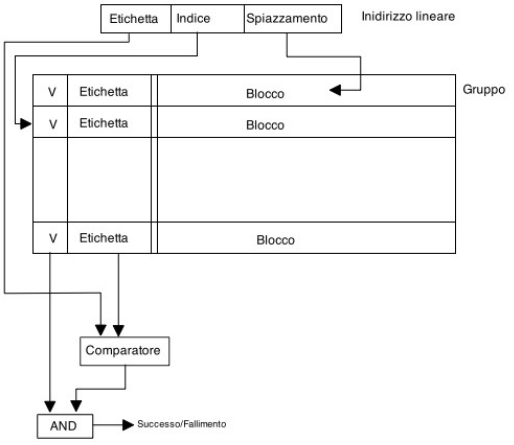
***Nota* 10** *si possono considerare come ulteriore livelli, i registri del processore e la coda della PU(prefech unit)*

Nel livello superiore e inferiore le informazioni sono *raggruppati* in *blocchi* Il trasferimento di informazioni dal livello inferiore a livello superiore avviene a livello di blocco.

blocco: insieme di locazioni consecutive allineati di qualche decina di byte (32 byte)

2.6.1 Organizzazione della memoria cache

La memoria cache é organizzata a gruppi prevede un indirizzamento diretto(non é previsto nessun registro speciale per poterla indirizzare).



L'indirizzo generato dal processore é pensato spartito in tre parti:

- *indice* : seleziona un gruppo all'interno della memoria cache.
- *spiazzamento* : individua una specifica locazione all'interno del blocco informazione.
- *etichetta* : in caso di validitá (bit V) la componente etichetta dell'indirizzo viene confrontata con l'etichetta del gruppo selezionato mediante un comparatore, per verificane l'uguaglianza.

tag: bit di validitá + campo etichetta.

***Esempio* 2.3** *Una memoria cache di 256 kbyte organizzata a blocchi di 32byte (8 linee)*

Nota *11* *La memoria cache essendo un componente hardware, comporta che la partizione dell'indirizzo lineare sia un un parametro di progetto. Il processore genera l'indirizzo sui piedini A31-A2, /BE3-/B0.*■

Ne segue che:

- *lo spiazzamento ćostituito dai 3 bit A4-A2 per le linee e dai /BE3-/BE0 per lo locazioni all'interno delle linee.*
- *indice :* $(256Kbyte/32byte) = 8k \quad blocchi(2^3 * 2^{10} \quad blocchi)$ *ne segue che sono necessari 13 bit (A17-A5) per indirizzare 8k blocchi.*
- *l'etichetta é composta dai rimanenti bit (A31-A18).*

Operazione di lettura

Quando il processore compie un operazione di *lettura* avviene:

1. un accesso in lettura al gruppo della memoria cache individuato mediante *l'indice* (componente dell'indirizzo).
2. l'esame del bit di validitá e confronto fra etichetta dell'indirizzo e l'etichetta del gruppo selezionato.

Queste due operazioni possono dar luogo a :

- **hit** (successo): l'entitá informativa riferita viene fornita al processore.
- **miss** (fallimento):
  * avviene un accesso in scrittura al gruppo selezionato della memoria cache con la memorizzazione nel campo tag. e il trasferimento di un nuovo blocco dalla memoria principale alla memoria cache. Se il bit V = 0 (nessun rimpiazzamento) altrimenti si effettua un rimpiazzamento del blocco selezionato.
  * trasferimento dell'entitínformativa riferita verso il processore.

Operazione di scrittura

Quando il processore compie operazioni di scrittura effettue:

1. un accesso in lettura al gruppo della memoria cache individuato mediante *l'indice* (componente dell'indirizzo).
2. l'esame del bit di validitá e confronto fra etichetta dell'indirizzo e l'etichetta del gruppo selezionato.

Queste due operazioni possono dar luogo a :

- **hit** (successo): In base alla politica di gestione della memoria cache dell'operazione di scrittura, si hanno i seguenti casi.

  - WRITE TROUGH: l'entitá viene memorizzata sia sulla cache che in memoria principale.
  - WRITE BACK: l'entitá informativa viene memorizzata solo all'interno del blocco della memoria cache .

  ***Nota* 12** *Con la regola di scrittura* WRITE BACK *la memoria principale non viene continuamente aggiornata.*
  L'aggiornamento avviene solo in caso di reimpiazzamento. ***Queso richiede un particolare tipo di gestione della consistenza dell'informazione quando si hanno piú entitá che accedono alla memoria principale, ossia all'informazione stessa(e. g. DMA).***

- **miss** (fallimento): L'entitá informativa da scrivere viene memorizzata nella memoria centrale.
  * **V = 0:** con la regola WRITE BACK avviene anche il trasferimento del nuovo blocco della memoria principale alla memoria cache
    (rispetta il principio della localitá).
  * **V = 1:** RIMPIAZZAMENTO il trasferimento del vecchio blocco della memoria cache alla memoria principale con conseguente aggiornamento del blocco nella memoria centrale, e trasferimento del nuovo blocco dalla memoria principale alla memoria cache.

Dal punto di vista costruttivo la memoria cache é costruttivo da due banchi di memoria. Il primo contiene il campo *tag*, il secondo banco contiene il blocco. In questo modo si possono effettuare accessi diversi nei due blocchi, in tempi diversi. Avviene sempre un preliminare accesso in lettura del primo blocco indipendentemente dal tipo di ciclo comandato dal processore.
In caso di successo avviene l'operazione comandata dal processore (lettura/scrittura).
All'interno della memoria cache ad indirizzamento diretto, un blocco pu'øessere solo in una entrate, in dipendenza dalla componente *indice*. Dualmente non si puó avere due blocchi con lo stesso indice (ovvero lo stesso blocco con indice differente ed etichetta diversa).
La memoria cache puó contenere una *pagina* di memoria fisica, pari alla dimensione della memoria cache.

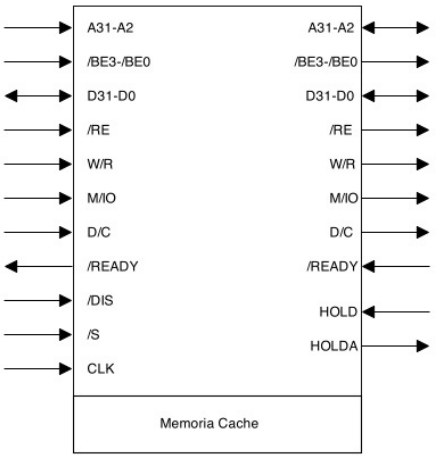
pagina di memoria: insieme consecutivo di locazioni di memoria con la stessa etichetta.

2.6.2 Controllore della memoria cache

La memoria cache prevede un controllore che effettua risposte alle operazioni ddi lettura/scrittura comandate dal processore, ed effettua le operazioni necessarie sulla memoria cache e memoria principale, in accordo alle operazioni descritte precedentemente.

I piedini omonimi del controllore e del processore possono essere messi in comunicazione diretta (viene esclusa la memoria cache). Disabilitare la cache é necessario nei seguenti casi.

- vengono effettuate operazioni di I/O (cicli di I/O)



- viene attivato il pin /DIS (dipendente dal pin del processore CD). Tipicamente la memoria cache viene disabilitata quando si indirizza la memoria ROM (contenente il programma di bootstrap) e CD viene gestito dalla MMU.

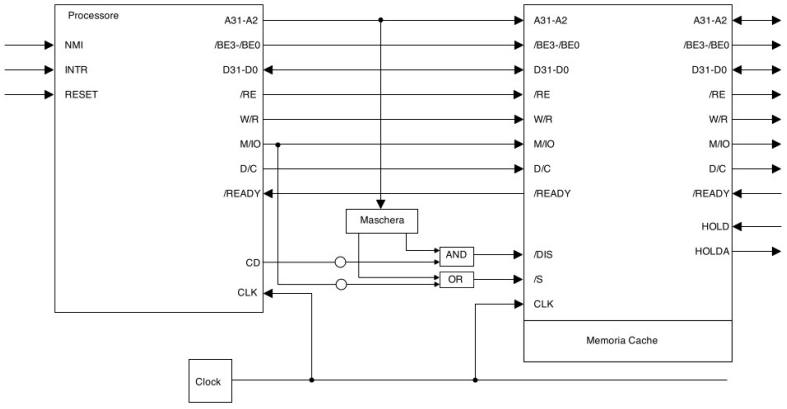
I pin HOLD e HOLDA vengono utilizzati nel meccanismo di accesso diretto alla memoria, in tal caso A31-A2: possono essere anche di ingresso al controllore, necessarie ai fini della consistenza delle informazioni (vedi piú avanti).

Uno o piú gruppi possono essere invalidati ponendo a 0 il valore del bit V (flush). Quest'operazione viene effettuata dal controllore abitualmente con comandi software:

- due registri sono destinati a contenere gli indici di inizio e fine del gruppo da invalidare, e un terzo registro per ricevere il comando di flush. Ai fini di queste operazioni il controllore viene abilitato tramite il piedino /s.

2.6.3 Montaggio del controllore cache

Il controllore viene montato subito a valle del processore e prima del circuito di pilotaggio del bus. Il piedino /s é sopportato dall'uscita



di una maschera e dal segnale M/IO (in OR) determinando in tal modo l'indirizzo base e lo spazio in cui il controllore viene montato(si assume lo spazio di memoria). Il segnale di disabilitazione della memoria cache /DIS viene attivato o da una maschera o dal segnale CD proveniente dal processore.

2.6.4 Memoria cache associativa a insieme (set-associativa)

Una memoria cache associativa a insiemi con cardinalitá n (n-way) é costituita da n parti uguali. Ogni singola parte é identica alla memoria cache analizzata, con l'aggiunta di un campo per la gestione del rimpiazzamento. L'indirizzo viene suddiviso, anche in questo caso, in tre componenti. La componente indice rappresenta l'indirizzo di un set contenente n gruppi: ∀ gruppo avviene la stessa analisi vista (n comparatori). In caso di successo per un gruppo la componente spiazzamento rappresenta l'indirizzo della locazione all'interno del blocco per quel gruppo. Si ha fallimento globale, se si ha ∀ gruppo del set selezionato. Un blocco puó essere memorizzato, in relazione al suo indice in uno degli n gruppi. Il campo previsto per ogni set e il campo R, utile ai fini statistici per il rimpiazzamento, nel caso in cui campo di validitá vale 1.

Regola di rimpiazzamento

Si rimpiazza il gruppo non riferito da piú tempo LRS(Least Recently Used).
Il campo R viene aggiornato dopoun'operazione andata a buon fine che coinvolge un insieme, in base al valore precedente, e al numero d'ordine del gruppo dell'insieme coinvolto nell'operazione.

campo R: deve contente un informazione d'ordine relativa a tutti i gruppi dell'inseme

***Esempio* 2.4** *Memoria cache associativa a insieme con cardinalitá 2, il campo R é costituito solo da 1bit. e la gestione risulta semplicemente nel far assumere al bit il valore che individua il gruppo non coinvolto.*

2.7 Interruzione di programma

L'interfaccia quando é pronta un trasferimento invia un segnale al processore mediante un'apposita linea di collegamento. Il processore viene interrompe momentaneamente l'esecuzione del programma in corso e prevede a effettuare il trasferimento richiesto. Esistono altri eventi che producono la temporanea interruzione del programma in corso. Tali eventi sono prodotti dal verificarsi di particolari condizioni all'interno del processore, verificarsi di particolari condizioni all'interno del processore (*eccezioni*), oppure nell'esecuzione di istruzione da parte dell'utente che non puó eseguire. Un'interruzione é una sospensione forzata del programma in esecuzione e il trasferimento del controllo ad un'apposita *routine di servizio* il cui compito é soddisfare le esigenze che hanno provocato l'interruzione stessa. Al termine la routine deve restituire il controllo al programma sospeso. In base alla causa che determina tali eventi si hanno:

- *interruzioni esterne* : vengono determinate da richieste che giungono al processore sui piedini
  - NMI : una richiesta su tale piedino si traduce in un effettiva interruzione
  - INTR : una richiesta su tale piedino si traduce in effettiva interruzione o meno a seconda che il bit IF del registro EFLAG vale 1 ovvero 0.Tale flag puó essere modificato via software mediante le due istruzioni STI e CLI.

- *Interruzioni software* : sono prodotte dall'esecuzione dell'istruzione INT.
- *Single step trap* : sono prodotte al termine della fase di esecuzione di ogni istruzione se TF (bit 8 di EFLAG) vale 1. **fa eccezione l'istruzione che ha provveduto a porre a 1 TF**.
- *Eccezioni nel processore* : prodotte da circuiti interni al processore, ogni volta che si verifica un condizione anomala che impedisce il completamento dell'istruzione in corso.

2.7.1 Istruzioni asincrone e sincrone

- Le istruzioni giungono al processore in qualsiasi momento.
- Al termine della fase di esecuzione di un'istruzione qualsiasi, a microprogramma, viene controllata l'eventuale presenza di richieste di interruzioni esterne, prima di passare alla fase di fetch dell'istruzione sequenzialmente successiva nel flusso di esecuzione.

un interruzione esterna non puó mai interrompere un istruzione quando questa é in esecuzione

- Le altre interruzioni sono prodotte dall'esecuzione di istruzioni (quindi sono sincrone rispetto al programma in esecuzione). Le interruzioni software e single step trap sono prodotte ad esecuzione terminata di una qualsiasi istruzione, non producendo l'annullamento dell'effetto dell'istruzione eseguita.
- Le eccezioni sospendono l'esecuzione annullando l'effetto dell'eventuale esecuzione parziale e questo richiede un meccanismo di gestione per mandare in porto questo vincolo(vedi avanti).

2.7.2 Tipo d'interruzione

IL processore associa ad ogni interruzione un identificatore a 8 bit in base al quale determina l'indirizzo della routine di servizio dell'interruzione stessa. Il tipo viene determinato secondo quanto segue.

i tipi d'interruzione possibili sono quindi 256, e altrettante sono le routine di servizio.

- *interruzioni esterne*:
  - NMI ⇒ il tipo implicito pari a 2.
  - INTR ⇒ il tipo viene prelevato dalla parte bassa del bus dati, durante un ciclo di risposta ad una richiesta di interruzioni esterna (vedi avanti). **Ne segue che l'invio di una richiesta su questo deve essere accompagnata dal tipo che puó essere uno dei 256 possibili.**
- *Interruzioni software*: il tipo é dato dall'operando immediato dell'istruzione INT. Compito del programmatore specificare il tipo.
- *Single step trap*: tipo implicito pari a 1.
- *Eccezioni interne al processore*: il tipo é implicito legato alla causa che ha determinato l'interruzione.

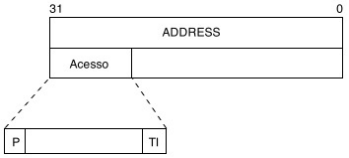
2.7.3 Tabella delle interruzioni e azioni del processore

Il processore precede che a partire da un certo indirizzo base (indirizzo base é contenuto nel registro speciale IDTR) vi sia la cosiddetta *tabella dell'interruzione* IDT (Interrupt Descriptor Table) che ha tanti descrittori quanti sono i tipi 256.

2.7.4 Descrittore della IDT

Ogni descrittore (**gate**) é constituito da 8 byte.

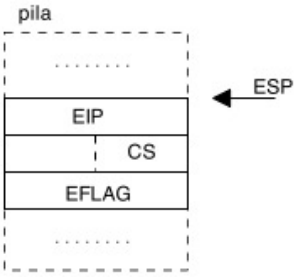
$$8\text{byte} * 256 = 2^3 * 2^8 = 2^{11}\text{byte} = 1\text{Kbyte} \tag{3}$$



Accettazione di una richiesta di interruzione

Il processore quando accettazione una richiesta di interruzione

1. riconosce il tipo dell'interruzione
2. preleva un gate, e un entrata della IDT in funzione del tipo, il nuovo indirizzo e il byte d'accesso.
3. se il but P vale 0 $\Rightarrow$ gate non presente, genera un eccezione.
4. immettere in pila tre parole lunghe EFALG,CS,EIP.
5. carica l'indirizzo contenuto nel gate in EIP
6. pone 0 :
   - TF se TI vale 1 (GATE di tipo TRAP)
   - TF e IF se TI vale 0 (GATE di tipo INTERRUPT)



Per effetto del punto 5 il processore passa ad eseguire le istruzione della routine di servizio. A fine esecuzione la routine di servizio. A fine esecuzione la routine ritorna al programma interrotto eseguendo una IRET.

IRET preleva 3 parole lunghe dalla pila e la trasferisce la prima in EIP e la terza in EFLAG

Il *meccanismo di interruzione* si dice *vettorizzato*, un interruzione ha associato un tipo che determina una specifica entrata della IDT. Il processore esegue la routine di servizio

- con le interruzione **disabilitate** se il gate é di tipo interrupt.
  - **in questo modo non si hanno annidamenti delle interruzioni mascherabili, a meno che la routine non provvede esplicitamente a porre ad 1 IF**.
- senza che si verifichi il single step trap sia che il gate coinvolto e' interrupt o trap, a meno che non si provvede a porre ad 1 TF(in questo caso si esegue la routine in single step trap).

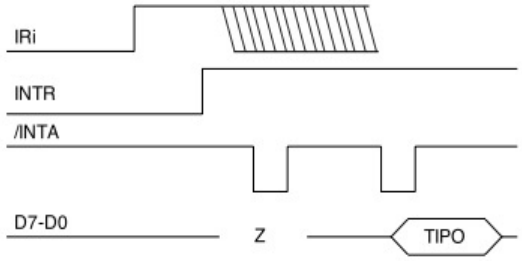
2.7.5 Riconoscimento del tipo per le richieste di interruzioni esterne mascherabili

Per le interruzioni *esterne mascherabili* il tipo é dato dal un byte che deve essere inviato al processore sui piedini D7-D0 dopo che accetta la richiesta stessa.

ciclo di risposta

viene effettuata una lettura nello spazio di I/O (due cicli di lettura) agli indirizzi 0 e 4 con D/C= 0.

- il primo ciclo serve a segnalare che la richiesta é stata accetta
- li secondo ciclo serve a informare il controllore delle interruzioni che il processore é pronto a prelevare il tipo. Il circuito di comando quando riceve dal processore il comando
  IO READ CONTROL (M/IO = 0 D/C = 0 W/R = 0 A31-A0 = 0x0.. . . 0 oppure 0x0.. . 04) invia sul piedino /INTA un segnale (impulso temporizzato).



Indirizzo di ritorno

a) Nel caso di interruzioni Software o esterne e single step trap il valore di EIP che viene memorizzato in pila é l'indirizzo:

- sequenzialmente successivo rispetto al punto i interruzione.
- determinato da un eventuale salto (le interruzioni e di single step trap vengono valutate alla fine della fase di esecuzione dell'istruzione corrente)

Se l'istruzione é di salto quando viene eseguita in EIP viene caricato l'indirizzo di salto. Prima che termina l'esecuzione dell'istruzione di salto se viene riscontrata la presenza di richiesta di interruzione allora viene salvato in pila il valore relativo al punto di salto contenuto in EIP.

b) le *eccezioni* si dividono in :

- trap: si comportano come nel caso al punto a.
- abort: terminazione forzata dell'istruzione attuale senza che sia significativo l'indirizzo di ritorno.

- fault: produce la memorizzazione dell'indirizzo d'istruzione che ha prodotto l'eccezione e riesecuzione della stessa al termine della routine di servizio che esegue la IRET.

La possibilita di poter rieseguire un'istruzione comporta la necessitá che non venga alterato il contenuto dei registri fino a quando l'istruzione non termina.

⇓

Registri d'appoggio

2.7.6 Gate non presente

Un descrittore della tabella IDT contiene, come abbiamo visto un byte d'accesso. Tale byte contiene a sua volta il bit P (Present). In caso in cui P indica l'assenza del gate viene generato un fault (tipo implicito 11). Non ha senso un fault su gati coinvolti dal verificarsi di interruzioni esterne. La routine che va in esecuzione negli altri casi, puó inserire nella tabella IDT il gate mancante aggiornando il bit P. All'esecuzione della IRET il programma riparte dall'istruzione che ha provocato fault.

2.7.7 Routine di servizio

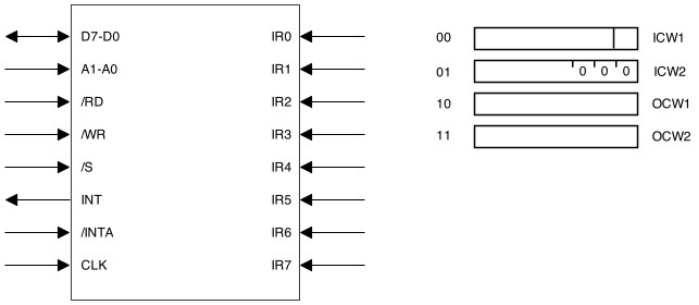
Le routine sono fornite con il software di base, del sistema, e vengono caricate in memoria all'atto dell'inizializzazione del sistema di bootstap.

classi di interruzioni

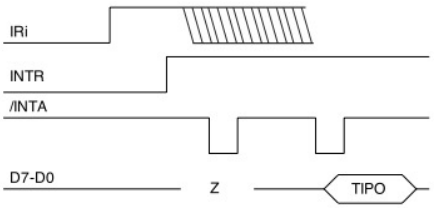
- *interruzione esterne mascherabili* sono prodotte dalle interruzioni mascherabili. Le routine di servizio provvedono a trasferire dati.
- *interruzioni esterne non mascherabili* sono eventi di cause critiche. Le routine provvedono a effettuare azioni di recupero(tipo 2).
- *interruzioni software* prodotte in modo volontario;
  - offrono un meccanismo alternativo alle call: usare l'istruzione INT $tipo consente di riferire le primitive di sistema mediante un numero d'ordine e non un nome(non si ha necessita di collegamento fra la primitiva e il programma che ha necessitá di utilizzarla.
  - le primitive servono a svolgere funzioni che l'utente non é autorizzato a svolgere.
- *le eccezioni* prodotte da un'anomalia. le routine provvedono ad impedire che il programma continui ad operare in modo scorretto.
- *single step trap.* La routina consentono all'utente (cha pone ad 1 TF) di eseguire il proprio programma in single step trap. Tele routine viene eseguita con TF = 0 e provvede a memorizzare il contenuto dei registri in un'apposita zona di memoria per la gestione di operazioni di visualizzazione/ modifica. La routine eseguendo la IRET ripristina il vecchio valore di EFLAG ponendo nuovamente a 1 TF.

2.7.8 Controllore delle interruzioni

Il processore possiede solo il piedino INTR per le richieste mascherabili. Su un sistema di elaborazione vengono montati piú sorgenti di interruzioni. Si prevede un circuito esterno al processore che stabilisce la precedenza in caso di richieste multiple.



- ll controllore per essere collegato al bus possiede gli usuali pin.
- i pin IR7-IR0 servono a ricevere le richieste di interruzioni provenienti da sorgenti esterne(transizione del segnale da 0 a 1).
- il controllore gestisce le richieste con livello di prioritá crescente
  - massimo per IR0
  - minimo per IR7
- tramite il pin INT il controllore inoltra le richieste al processore (INT = 1) e attende la risposta sul piedino /INTA (/INT = 0).
  - la risposta a una richiesta di interruzione é rappresentata da due impulsi sul piedino /INTA
  - il circuito di comando collegato al processore riconosce il ciclo di risposta (IO READ CONTROL) e produce due impulsi richiesti per il controllore.



Il controllore é un interfaccia con 4 registri che possono essere modificati durante il sistem:

- ICW (Initialization Control Word)
  - *ICW1* : il bit meno significativo serve a stabilire il modo di funzionamento
    - Normal : 0
    - Fully Nested: 1
  - *ICW2* : serve a contenere il tipo base a partire dal quale il controllore ricava il tipo associato alla varie richieste.

***Nota* 13** *Per una rapida determinazione del tipo, il tipo base é multiplo di 8 in tal modo non si ha riporto nell'operazione di somma.*

$$\text{tipo}_{IRi} = \text{tipo}_{base(=ICW2)} + i \tag{4}$$

- OCW (Operand Control Word)
  - *OCW1* : serve a mascherare le singole richieste di interruzione che provengono al controllore. Ogni bit é associato a un piedino IRi e quando vale 1 la richiesta viene scartata.
  - *OCW2* : viene usato per la modualitá *fully nested* (vedi avanti)

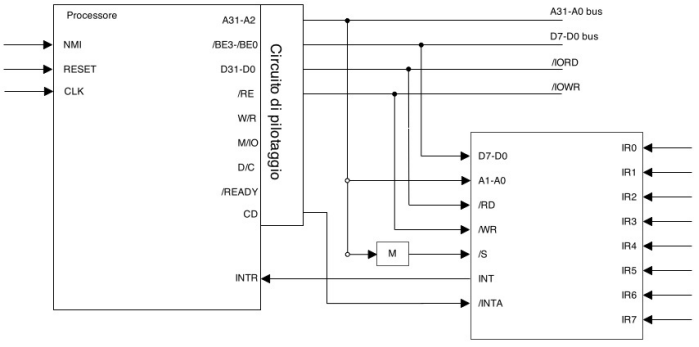
Il controllore utilizza un registro interno IRR(Interrupt Request Register) non visibile al programmatore, in cui memorizza le richieste pendenti.
continuamente:

1. memorizza in IRR la richiesta proveniente dal piedino IRi.
2. - attende che IRR sia diverso da 0.
   - invia una richiesta di interruzione attivando il piedino INT.
   - attende un primo impulso su /INTA
     - Seleziona la richiesta di interruzione a piú alta prioritá fra quelle memorizzate in IRR, e pone a 0 l'i-mo bit di IRR se la richiesta selezionata é la i-ma.
   - attende un secondo impulso di risposta e quando lo riceve determina il tipo inviandolo su D7-D0.E rimuove la richiesta di interruzione disattivando INT.

***Nota* 14** *l'i-ma sorgente invia un impulso al controllore altrimenti per disattivare la richiesta verso il controllore é necessario che la routine di servizio acceda al registro candidato come il registro OK dell'interfaccia(una lettura su tale registro comporta la disattivazione del segnale in ingresso sul piedino IRi)*

2.7.9 Montaggio del controllore dell'interruzioni

Il controllore viene montato nello spazio di I/O e sul bus dati a 8 bit, in tal modo i registri hanno indirizzi consecutivi.



Il doppio ciclo di lettura ha gli indirizzi 0 e 4 nello spazio di I/O non vieta che per questi indirizzi venga abilitato qualunque bus dati, il *tipo* viaggia correttamente dal controllore al processore. Il verso é determinato dal valore 0 del piedino W/R.

2.7.10 Controllore dell'interruzione e memoria cache

- In presenza di memoria cache il ciclo di risposta deve essere effettuato quando il bus non é impegnato.
- Il controllore della memoria cache gestisce tutti i cicli di bus in base alle specifiche elettriche che giungono dal processore

2.7.11 Fully Nested delle interruzioni mascherabili

Si ha un annidamento delle interruzioni quando una routine di servizio é relativa ad un interruzione mascherabile esterna viene interrotta a sua volta.

- Le interruzioni mascherabili esterne si possono annidare in presenza di gate di tipo trap ovvero di interrupt quando la routine di servizio pone esplicitamente a uno IF.
- Il modo di funzionamento *normal* del controllore di interruzioni non é idoneo a gestire correttamente l'annidamento. Infatti puó accadere che una routine di servizio sia interrotta da una richiesta di interruzioni a piú bassa prioritá accettata dal processore.

***Nota* 15** *Questo puó accadere solo nella condizione citata precedentemente(quando si riabilita il processore ad accettare richieste mascherabili esterne da parte di una routine di servizio relativa ad una medesima richiesta).*

Ne segue che é necessaria una diversa modalitá di funzionamento.

Fully Nested

- Ogni volta che una richiesta di interruzione viene accettata (risposta sul piedino /INTA) il controllore cessa di gestire (ma non di memorizzare le eventuali richieste di interruzioni a prioritá minore o uguale fino a quando non riceve il comando **EOI** (End Off Interrupte) dalla routine di servizio.
- Le interruzioni di prioritá maggiori o uguali vengono regolarmente gestite.
- Per tanto una routine di servizio puó essere interrotta quando IF é posto ad 1 dalla stessa,ma solo da una richiesta di interruzione a piú alta prioritá.
- L'invio di EOI avviene alla fine dalla routine di servizio dopo aver disabilitato il processore a ricevere richieste di interruzioni mascherabili esterne: si scrive il valore 0x20 nel registro OCW2.

2.7.12 Schema di una routine di servizio

```
. SET OCW2, . . .
. SET EOI, 0x20
. text
rout_j:
```

```
pushl . . .
#abilitazione del processore ad
#accettare richieste per un gate interrupt
#per un gate interrupt
STI
. . .
elaborazioni
. . . .
CLI
#invio del comando
# EOI
movb $EOI,%al
movw $OCW2,%dx
outb %al, %dx
popl . . .
iret
```

Quando il processore riceve una richiesta di interruzioni:

1. riconoscere il tipo di interruzione
2. . .
3. . .
4. immettere in pila tre parole tra cui EFALG(IF=1)
5. . .
6. porre a 0 TF se TI = Trap, IF e TF se TI = Interrupt

***Nota* 16** *La routine prima di terminare invia EOI e prima di inviare tale comando disabilita il processore ad accettare richieste di interruzioni esterne mascherabili. Questo permette alla routine di terminare senza essere interrotta dalle routine di prioritá minore o uguali rispetto a quella sotto servizio.*

Quando viene eseguita la IRET viene ripristinato EFLAG quindi si riabilita il processore a ricevere richieste di interruzioni. Il controllore adopera due registri interni (non visibili al programmatore):

- IRR
- ISR (In Service Register)

continuamente:

1. memorizza in IRR la richiesta proveniente dal piedino IRi.
2. - attende che IRR sia diverso da 0.Se una richiesta é sotto servizio, attende una richiesta a prioritá maggiore di quest'ultima.
   - invia una richiesta di interruzione attivando il piedino INT.
   - attende un primo impulso su /INTA
     - Seleziona la richiesta di interruzione a piú alta prioritá fra quelle memorizzate in IRR, e pone a 0 l'i-mo bit di IRR se la richiesta selezionata é la i-ma, e a 1 in ISR.

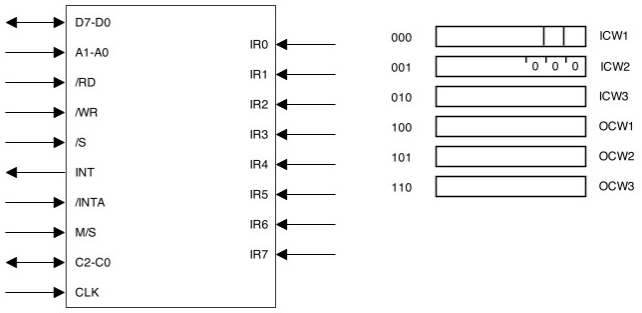
- attende un secondo impulso di risposta e quando lo riceve determina il tipo inviandolo su D7-D0.E rimuove la richiesta di interruzione disattivando INT.

3. Il controllore ogni volta che riceve la parola EOI azzera il bit di ISR a piú alta prioritá tra quelle sotto servizio.

2.7.13 Collegamento di piú controllori in cascata

Quando il numero di di sorgenti é superiore a quello che puó gestire un singolo controllore si ricorre al cosiddetto montaggio in cascata di pi'čontrollori. In tal caso si ha:

- un controllore master.
- un o piú controllore slave.



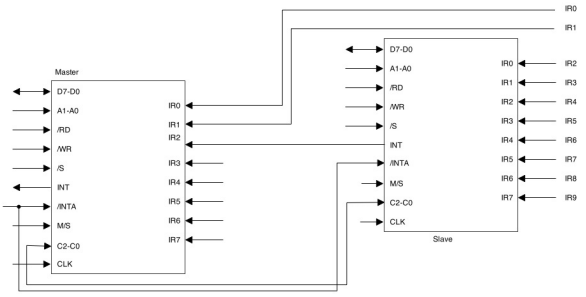
Il controllore ha le seguente caratteristiche:

- *M/S* : il piedino M/S stabilisce se il controllore é master oppure slave.
- *C2-C0* : sono di uscita quando il controllore é master e di ingresso.
- *ICW1* : modalitá di funzionamento
  - Normal
  - Special Fully Nested (bit n1 = 1 vedi avanti).
- *ICW2* : contiene il tipo base.
- *ICW3* : viene utilizzata dl controllore:
  - *master*: per indicare i piedini a cui sono collegati i vari controllori slave(si inizializza in fase di init).
  - *slave*: contiene un identificatore rappresentato su 3 bit il che rappresenta il numero d'ordine del piedino del master a cui é collegato.
- *OCW3* : registro di stato e consente di leggere il contenuto di ISR.

Funzionamento

- Il controllore *master* inoltra le richieste di interruzione al processore con le analizzati precedentemente.
- quando riceve il primo impulso di risposta su /INTA seleziona la richiesta a piú alta prioritá:
  - Se questa é stata ricevuta su un piedino, IRj,a cui é connesso un controllore slave, invia sui piedini C2-C0 il numero j.
  - Ogni controllore slave all'arrivo del secondo segnale su /INTA preleva il numero j dai piedini di ingresso C2-C0.Il controllore slave che riconosce tale numero coincidente con il valore del proprio registro ICW3, invia il tipo su D7-D0 del bus dati una volta calcolato.

***Esempio* 2.5**



Esaminiamo il caso in cui una richiesta di interruzione giunge al controllore slave, il quale la inoltra al controllore master sul piedino a cui é collegato (IR2 del master). Al primo segnale di risposta sul piedino /INTA (impulso che arriva a tutti i controllori) il master vede che sul piedino IR2 é collegato un controllore slave , esaminando il registro ICW3, sul quale ha ricevuto una richiesta di interruzione. Visto che non il controllore master non é in grado di calcolare il tipo da inviare al processore, inoltra il valore 2 sui piedini C2-C0. All'arrivo del secondo segnale su /INTA il controllore slave confronta il valore ricevuto su C2-C0 con il valore del suo registro ICW3, e riconoscendo tal valore provvede ad inviare al processore, una volta calcolato, il tipo immettendolo in D7-D0.

2.7.14 Modalitá di funzionamento Special Fully Nested

Nel montaggio in cascata la modalitá fully nested puó dar luogo ad un funzionamento globale che non rispetta le regole di precedenza: Il controllore master quando riceve una richiesta di interruzione proveniente da un controllore slave,che viene accettata dal processore una volta inoltrata ad esso, **cessa di gestire tutte le richieste di interruzioni provenienti da quello slave (ma non quelli che arrivano al master), anche quelle con prioritá piú alta rispetto a quella sotto servizio.** Si rende necessaria una modalitá di funzionamento che gestisca in modo corretto le prioritá.

Special Fully Nested

Ogni volta che una richiesta vien accettata dal processore, il controllore master cessa di gestire eventuali richieste di interruzioni di *minor*

priorità finche non viene ricevuto il comando EOI, dalla routine di servizio in esecuzione. Le richieste di interruzioni con prioritá *maggiore o uguale* vengono regolarmente gestite.
Purché

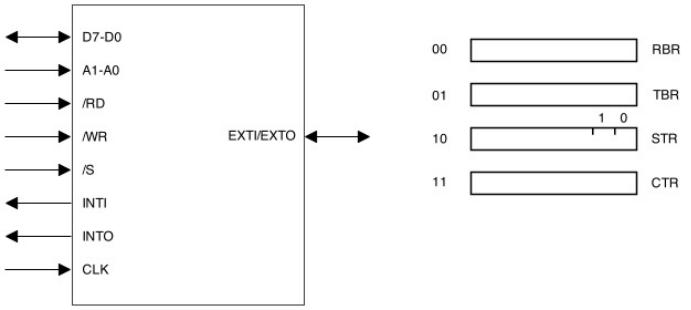
- tutte le richieste provengono al master sono tutte richieste provenienti da controllori slave, in tal caso le richieste sono gestite con prioritá. Le richieste con ugual prioritá vengo bloccate. Se cosi non fosse si rischia che le richieste con ugual prioritá a quella sotto servizio la interrompano.
- oppure il controllore master gestisce le richieste con prioritá uguale , solo se queste provengono dai piedini a cui sono collegati dei controllori slave.

Una routine di servizio relativa ad una richiesta gestita unicamente da un controllore slave deve

- inviare il comando EOI a quest'ultimo in modo che possa gestire richieste con prioritá minore o uguale alla richiesta gestita.
- leggere il valore di OCW3(ISR) e se non ci sono richieste pendenti ($OCW3! = 0$) gestite dallo slave associato alla routine che sta girando, deve inviare il comando EOI anche al controllore master.

2.7.15 Operazioni di I/O a interruzione di programma

Gestione dell'interfaccia a interruzione di programma.



- INTI (Interrupt Input)
- INTO (interrupt output)

L'interfaccia invia le richieste di interruzioni al controllore mediante questi piedini, quando si verifica uno di questi eventi:

- il buffer di ingresso pieno (RBR contiene un nuovo dato prelevato dal dispositivo esterno) e il bit FI del registro STR assume il valore 1.
- il buffer di uscita diviene vuoto(FO assume il valore 1).

Un accesso a RBR (lettura) o a TBR(scrittura) da parte del processore informa l'interfaccia che il servizio richiesto é stato svolto cosiché essa possa iniziare un nuovo trasferimento e alla fine inviare una nuova richiesta. L'interfaccia puó essere abilitata o disabilitata a inviare richieste di interruzioni a seconda del valore di alcuni bit del registro CTR:

- se il bit n. 0 di CTR vale 1 l'interfaccia di ingresso puó generare richieste di interruzioni.
- se il bit n. 1 di CTR vale 1 l'interfaccia di uscita puó generare una richieste di interruzioni.

Una volta abilitata viene generata immediatamente una richiesta di interruzioni se il buffer di ingresso é pieno o quello di uscita é vuoto.

***Nota* 17** *Alcune interfacce una volta abilitate generano una richiesta di interruzione sono quando il buffer di ingresso diviene pieno o quello di uscita diviene vuoto. Quindi é necessario riempire o svuotare rispettivamente i due buffer.*

Driver di interruzione = routine di servizio

Ingresso di n byte ad interruzione di programma

Ogni volta che va in esecuzione un driver verifica che l'operazione attuale non si l'ultimo dato da trasferire. Se ció si verifica si disabilita l'interfaccia a generare richieste di interruzioni. Nell'ipotesi che il controllore delle interruzioni sia stato inizializzato per operare in modalitá *fully nested* é necessario inviare il comando EOI a quest'ultimo.

***Nota* 18** *IL trasferimento di un singolo dato richiede piú tempo se effettuato a interruzione di programma rispetto a controllo di programma. Il vantaggio risiede nel fatto che si* l'attesa attiva.

Si adotta una forma rudimentale di sincronismo tra driver e il programma che richiede l'operazione di ingresso, in modo da permettere al programma stesso di capire se l'operazione di ingresso é globalmente terminata.

```
. include "servizio"
. global
. data
num_in: . word 0 # numero dati da trasferire
buffer_in: . fill 100,1 # buffer di memoria per i dati di ingresso.
. text
start:
main_in:
  movw num_in, %cx
  movl $buffer_in,%esi
  #chaimata della primitiva star_in
  int $250
  . . .
  altre operazioni # qui pu\'o andare in esecuzione driver_in
  . . .
  #chaimata dwlla primitva wait_in
  int $251
  . . .
  utilizzo dei dati
  . . .
  jmp dos

rooutine ini.
. SET . . .
. SET OCW2, . . .
. SET RBR, . . .
. SET CTR, . . .
# dati comuni a start_in , wait_in, driver_in
```

```
. data
  . couter_in: . word  0
  . pointer_in: . long 0
  . sincr_in: . byte 0
. text
ini:
  #inizializzazione del controllore delle interruzioni
  call ini_contr
  #iniziallizazione della tabella IDT
  movl $start_in, %eax
  movb $250,%bl
  call ini_tab
  movl $wait_in, %eax
  movb $251,%bl
  call ini_tab
  movl $driver_in, %eax
  movb $12,%bl # interfaccia collegata al piedino 4 e il valore di ICW2 = 8 (8+4 = 12)
  call ini_tab
  ret

ini_tab:
  . . .
  . . .
  ret
ini_contr:
  . . .
  . . .
  ret

start_in:
rout_250:
  pushl %edx
  pushl %eax
  #trasferimento paramteri nei dati comuni
  movw %cx, couter_in
  movl %esi, pointer_in
  #abilitare interfaccia a generare richieste di interruzioni
  movw $CTR, %dx
  inb %dx, %al
  orb $0x01, %al
  outb %al, %dx # la routine non pu\'o essre interrotta visto che IF = 0
  #inizializzatione variabile sincronizzazione
  movb $0, sincr_in
  popl %eax
  popl %edx
  iret

wait_in:
rout_12:
  #abilito processore ad accettare richieste di interruzioni
  STI # visto che IF = 0  altrimenti driver_in non pu\'o andare in esecuzione
ciclo_in:
  cmpl $0, sincr_in # necessario visto che il programma raggiunto tale punto non pu\'o
  je ciclo_in #fare altro ma pu\'o essere interrotto.
  IRET
driver_in
rout_12:
  pushl %eax
  pushl %edx
  pushl %edi
  STI
  #verifica se si tratta dell'ultimo trasferimento
  decw cont_in
  jnz step_in
  #disabilita l'interfaccia a effettuare richieste di interruzioni.
end_in:
```

```
  inb %dx, %al
  adn $0xFE, %al
  outb %al, %dx
step_in:
  movl pinter_in, %edi
  movw $RBR, %dx
  inb %dx, %al
  movlb %al,, (%edi)
  incl pointer_in
  #verifica se l'operazione \'e terminata
  cmpw $0, couter_in
  jne endstep_in
  movb $1, sinc_in
endstep_in:
  CLI
  movb $0x20, %al
  movw $OCW2, %dx
  outb %al,%dx # EOI
  popl %edi
  popl %edx
  popl %eax
  IRET
```

***Nota* 19**

- Le interruzioni del processore devono essere abilitate consentendo al driver_in di andare in esecuzione (altrimenti sincr_in non puo mai assumere il valore 1).
- Quando il dato da trasferire Í'ultimo. occorre disabilitare l'interfaccia a generare richieste di interruzioni prima di prelevare il dato altrimenti si inizia un nuovo ciclo di prelievo dal dispositivo esterno, producendo una nuova richiesta, prima di riuscire a disabilitarla. Tale richiesta non viene gestita subito visto che proviene da una fonte a ugual prioritá rispetto a quella sotto servizio, ma dopo l'invio del comando EOI(nel rischio di scrittura di zona di memoria non controllato).
- a interruzioni abilitate é opportuno porre a 1 il valore di sincr_in dopo aver effettuato il trasferimento altrimenti puó accadere che vada in esecuzione la wait_in e si opera sull'ultimo dato non significativo.

Uscita dati

Si ipotizza che il controllore opera in modalitá fully nested. Quindi in alcuni contesti é necessario abilitare il processore a ricevere richieste di interruzioni esterne mascherabili. Si suppone inoltre che il piedino INTO sia collegato al piedino IR5 del controllore delle interruzioni, e che ICW2 sia inizializzato a 08.

```
 . include "servizio"
 . global start
 . data
 num_aout: . word 0
 buffer_out: . fill 100,1
 . text
 start:
 main_aut:
   . . .
   movw num_out, %cx
   movl $buffer_out, %esi
```

```
   #chiamata della primitiva start_out
   int $252
   . . .
   altre elaborazioni
   #chiamata della primitiva wait_out
   int $253
   altre operazioni
   jmp dos

. SET . . .
. SET OCW2,. . .
. SET TRB,. . .
. SET CTR,. . .
#dati comuni alle routine
. data
counter_out: . word 0
pointer_out: long 0
sincr_out: . byte 0
. text
ini: #inizializzazione del controllore di interruzioni
  call ini_contr
  #predisposizione della tabella IDT
  movl $start_out,%eax
  movb $252,%bl
  call ini:_ab
  movl $wait_out,%eax
  movb $253,%bl
  call ini_tab
  movl $driver_out,%eax
  movb $13,%bl
  call ini_tab
  ret

ini_contr:
  . . .
  ret

ini_tab:
  . . .
  ret

start_out:
rout_252:
  pushl %edx
  pushl %eax
  #trasferimento parametri nei dati comuni
  movw %cx,%counter_out
  movl %esi,pointer_out
  #abilitazione dell'interfaccia
  movw $CTR,%DX
  inb %DX,%al
  orb $0x02,%al
  outb %al,%dx
  #inizializzazione variabile sincronizzazione
  movb $0,sincr_out
  pop %eax
  popl $edx
  iret

wiat_out:
rout_253:
  #abilitazione del processore
  STI
cilco_out:
  cmpb $0, sincr_out
  je ciclo_out
```

```
  iret

driver_out:
rout_13:
  pushl %eax
  pushl %edx
  pushl %edi
  STI
  #verifica se il si tratta dell'ultimo trasferimento
  decw couter_out
  jnz step_out
  #disabilito l'interfaccia
  movw $CTR,%DX
  inb %DX,%al
  andb $0xFD, %al
  outb %al,%DX
  #
step_out:
  movl pointer_out,%edi
  movb (%edi),%al
  movw $TBR, %dx
  outb %al,%dx
  incl pointer_out
  #verifica se l'operazione \'e terminata
  cmpw $0,couter_out
  jne endstep_out
  movb $1,sincr_out
  #disabilitare il processore a ricevere
  # richieste di interruzioni mentre si invia il comando EOI
endstep_out:
  CLI
  movb $0x020,%al
  movw $OCW2. %DX
  outb %al,%dx
  popl %edi
  popl %edx
  popl %eax
  iret
```

2.8 DMA

Come nel trasferimento di dati da e verso il disco ottico, la velocitá di trasferimento é superiore a quella dell'esecuzione di un istruzione. Si utilizza il DMA (Direct Memory Access control) tale controllore si *impossessa* temporaneamente del bus sostituendosi al processore e comanda il trasferimento diretto tra i registri delle interfacce e la memoria principale.

***Nota* 20** *La memoria di massa é gestita da un interfaccia.*

Un operazione di Ingresso/Uscita che prevede il trasferimento di piú dati, comporta che il controllore DMA sostituisca il processore piú volte nel controllo del bus.

2.8.1 Politica di gestione del bus

Si necessita di una politica di gestione del bus, essendo una risorsa condivisa, per concedere l'utilizzo del bus in mutua esclusione a chi ne fa richiesta, compreso il processore. Nel caso piú semplice l'utilizzo viene gestito dal processore visto che le due entitá che utilizzano il bus sono il processore stesso e il controllore DMA.

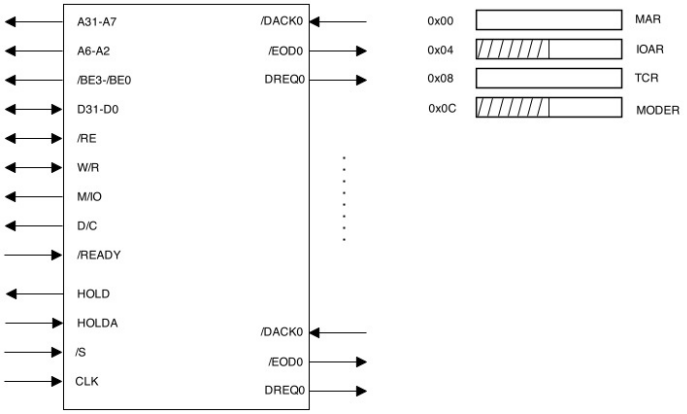
2.8.2 Controllore DMA

Il controllore DMA puó operare in modalitá:

- *Slave*: in fase di inizializzazione il controllore si comporta da slave, alla pare di ogni interfaccia.
- *master*: quando interagisce con le interfacce.

Gestisce a divisione di tempo, fino a 8 interfacce (8 *canali* DMA identici).

- $\forall$ canale sono previsti 4 registri da 32 bit.
- 3 piedini di connessione con le interfaccia che gestisce il canale stesso.



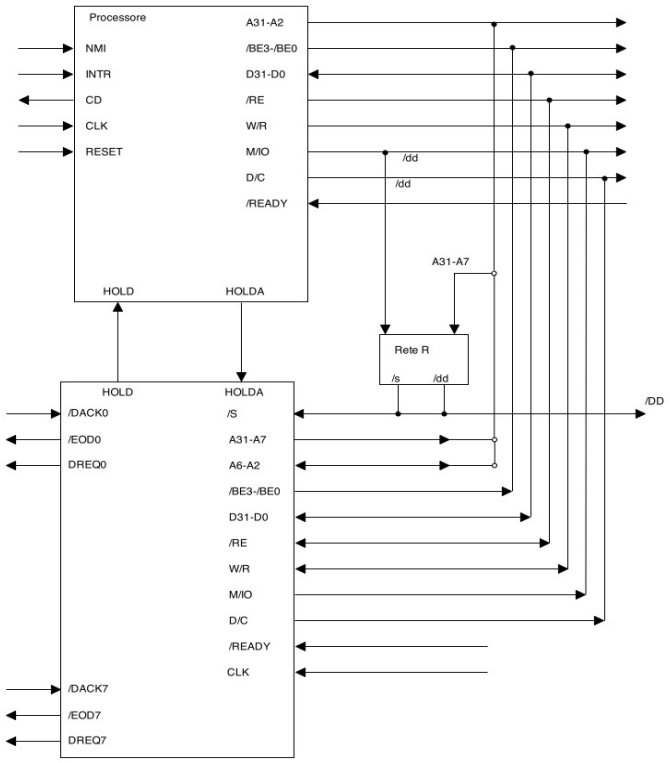
Quando il controllore é slave i trasferimenti avvengono sempre per linea (quindi /BE3-/BE0 del controllore devono essere in alta impedenza). I piedini A6-A2 sono necessari ad indirizzare i 32 registri interni dei canali. I comandi di lettura/scrittura dei registri si ottengono decodificando /RE e W/R, visto che non c'e informazione relativa allo spazio esterno. Quando il DMA si impossessa del bus effettua dei cicli di lettura/scrittura mediante comandi simili a quelli che genera il processore, nello spazio esterno interessato. A tale scopo esso possiede i piedini di :

- A31-A2
- /BE3-/BE0
- D31-D0 (bidirezionali)
- /RE, W/R, M/IO, D/C (D/C vale sempre 1).
- /READY (di ingresso) serve per stabilire la lunghezza del dei cicli di bus.

2.8.3 Cablaggio del controllore DMA

Il controllore viene montato vicino al processore a monte dei circuiti di pilotaggio nello spazio di memoria.

I collegamenti del processore e del controllore DMA sono collegati insieme, e solo uno dei due circuiti puó avere il controllo del bus.



La rete R

abilita il controllore quando questo é slave. /s diviene attivo quando A31-A2 assume una data configurazione che determina l'indirizzo base a partire dal quale si possono individuare i registri del controllore.

***Nota* 21** • *La rete R, se gli indirizzi sono minori di* 2^{16}*, avendo supposto che lo spazio di I/O si di 64Kbyte, necessita anche del collegamento M/IO: l'informazione di spazio é necessaria per stabilire che il DMA é montato nello spazio di memoria.*

- *Il controllore DMA riceve i comandi di lettura/scrittura da /RE e W/R e non da M/IO.*
- *Nel caso in cui il controllore fosse montato a indirizzi maggiori od ugali a* 2^{16} *l'informazione di spazio non é pi'ňecessaria come ingresso alla rete che genera il segnale di selezione del controllore.*

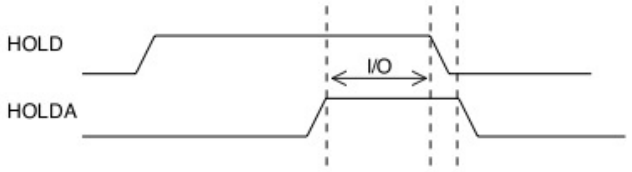
DD:

- Serve a come informazione ai circuiti di controllo (circuito di abilitazione) per disabilitare i transceiver che sopportano il bus dati.
- coincide con /ss: vale anche quando gli indirizzi sono minori di 2^{16}. In tal caso non sorge nessun problema quando il processore genera indirizzi minori di 2^{16} grazie all'informazione data dal segnale M/IO che specifica lo spazio interessato. C'e' da dire

anche che se gli indirizzi non sono riconosciuti dalla maschera R il bus dati non deve essere disabilita.

***Nota* 22** *Quando il controllore é master non deve generare indirizzi coincidenti con quelli a cui sono montati i sui registri interni. Altrimenti si rischia di scrivere dati casuali (A6-A2 sono in alta impdenza) nei registri stessi.*

- HOLD e HOLD sono i piedini mediante il quale il processore riceve le richieste di utilizzo del bus (HOLD) e invia la risposta di accettazione della richiesta stessa (HOLDA).
- il controllore DMA é dotato die medesimi piedini ma duali rispetto a quelli del processore e collegati a quest'ultimo, rendendo possibile il colloquio per concordare chi dei due ha il controllo del bus.
- partendo da una condizione iniziale in cui HOLD e HOLDA sono entrambi disattivati (il processore ha il controllo del bus) si ha:
  1. il controllore attiva il piedino HOLD, chiedendo al processore l'utilizzo del bus.
  2. il processore dopo aver terminato, l'eventuale; ultimo ciclo di bus si disconnette (ponendo il alta impedenza i piedini in comune) attivando HOLDA.
  3. il controllore si impossessa del bus e dopo averlo utilizzato, disattiva il compie le stesse azioni al punto precedente disattivando il piedino HOLD.
  4. il processore conclude il colloquio con il controllore ponendo HOLDA come disattivo e assume di nuovo il controllo del bus.



***Nota* 23** *il processore privilegia le richieste esterne rispetto alle proprie necessitá interne, per evitare che la possibilitá di perdere i dati che provengono dal DMA che gestisce operazioni di I/O a elevata velecitá.*

2.8.4 Trasferimenti in DMA

Il controllore riceve le varie richieste richieste sui piedini DREQ7-DREQ0.e risponde alla stesse attraverso /DACK-/DACK0 e /EOD7-EOD0.Eventuali richieste contemporanee vengono gestite dal controllore DMA con livello di priorit'a decrescente da DREQ0 fino DREQ7.

Il controllore inoltre possiede 4 registri da 32 bit, per ogni canale utilizzabili per la gestione dei vari trasferimenti richieste per portare a termine un'operazione. Un trasferimento puó interessare un byte, una parola o parola lunga allineata, in modo che ogni trasferimento dalla / in memoria possa essere effettuato con un unico ciclo di bus.

Registri del controllore DMA

I registri hanno indirizzi multipli di 4 a partire da una linea base. Gli indirizzi dei registri del canale i-mo si ottengono a partire dagli indirizzi die registri omonimi sommandovi $i * 16$.

- *MAR* : (Memory Address Register)
  Viene inizializzato con *l'indirizzo* ***fisico*** della prima locazione di memoria coinvolta nel trasferimento. Dopo ogni singolo trasferimento il contenuto viene automaticamente incrementato (di 1,2 o 4 a seconda del trasferimento).

- *IOAR* : (I/O Address Registr)
  Rimane invariato durante l'intera operazione di trasferimento. e va inizializzato con l'indirizzo della porta di I/O (l'indirizzo é solo di 16 bit).

- *TCR* : (Transfer Counter Register)
  Va inizializzato con il numero dei dati che si intende trasferire diminuito di uno. Viene decrementato automaticamente dopo ogni trasferimento. A fine operazione si emette un impulso su /EODi e disabilitato il canale, restituendo il controllo del bus al processore.

- *MODER* : (Mode Register)
  Ne viene utilizzato solo un byte.
  Permette di :
  - abilitare il canale ad accettare richieste.
  - fissare le modalitá di trasferimento:
    - direzione di trasferimento.
      memoria → interfaccia
      interfaccia → memoria
    - lunghezza dei dati da trasferire.
    - modo di trasferimento
      - **Cycle Stealing mode** (modo singolo)
      - **Burst mode** (modo continuo)
    - tipo di ciclo
      - **ciclo doppio**
      - **ciclo unico**
  - **bit:**
    - n. 0: 1 abilita, 0 disabilita il canale.
    - n. 1:
      1 interfaccia → memoria
      0 memoria → interfaccia.
    - n. 3. . n. 2: lunghezza degli operandi
      - 00 byte

        - 01 parola
        - 10 parola lunga
    - n. 4: 1 Cycle Stealing, 0 Burst
    - n. 5: 1 ciclo doppio, 0 ciclo unico
    - n. 6: 1 specifca che il canale viene utilizzato per trasferimento
      memoria → memoria.

I contenuti di questi registri (TCR) possono essere letti per ottenere informazioni sullo stato di avanzamento di un'operazione. Quando il bit n. 0 passa da 0 a 1 i canale diviene master e comincia a colloquiare con l'interfaccia e con il processore. Il passaggio di TCR da 0x0000 0000 a 0xFFFF FFFF determina il ritorno del canale nello stato slave.

Il passaggio da 0x00000000 → 0xFFFFFFFFFF permette di controllare un solo bit, CF, per determinare se l'operazione é terminata.

2.8.5 Modo di trasferimto del controllore DMA

- Cycle Stealing
- Burst

Questi due modi sono previsti per ogni canale.

- Se un canale é programmato in *modo di trasferimento singolo.* l'interfaccia deve effettuare una richiesta di servizio per ogni dato da trasferire. Il controllore DMA chiede ed assume il controllore del bus.
- Nel *modo di trasferimento continuo* l'interfaccia fá un'unica richiesta di servizio (indipendentemente dal numero di dati da trasferire). Il controllore una volta acquisito il controllo del bus, compie tutti i trasferimenti prima di restituire il controllo dello stesso al processore.

Il controllore DMA possiede due ulteriori registri.

- DR (DMA Request): per le richieste di si servizio.
- EO (End Of Operation): per indicare fine operazioe.

in parallelo:

1. Il DMA memorizza in DR le richieste di servizio (transizione da 0 a 1) che provengono tramite i piedini DREQ7-DRE0 dei canali abilitati se questi sono abilitati.
2. (a) esamina il contenuto di DR. se diverso da 0 invia una richiesta al processore la richiesta di utilizzare il bus ponendo HOLD uguale a 1.
   (b) attende che HOLDA valga 1.
   seleziona la richiesta a piú alta prioritá fra quelle memorizzate in DR, sia l'i-ma, ponendo a 0 i-mo bit di DR.
   (c) Decrementa il contenuto di TCRi
   Se questo passa da 0x0000 0000 a 0xFFFF FFFF, pone a 1 il bit i-mo di EO ed emette un impulso sul piedino EODi ad indicare che il prossimo trasferimento sará l'ultimo.

(d) Comanda il trasferimento di un dato tra la locazione di memoria e l'interfaccia coinvolte nel trasferimento. Durante tale trasferimento emette un impulso sul piedino /DACKi informando l'interfaccia ad emettere / prelevare il dato dal bus.

(e) incrementa il contenuto del registro MARi di (1, 2, 4 a seconda . . .) e

- se il bit i-mo di EO vale 1, completa il colloquio con il processore (disattiva HOLD e attende che HOLDA sia disattivato) e con l'interfaccia (attende che DREQi sia disattivo). Disabilita il canale azzerando anche il bit i-mo di EO e torna al punto a.
- se il bit i-mo di EO vale 0 e il canale i-mo é programmato in modo singolo, completa il colloquio con il processore, e con l'interfaccia (attende che DREQi si disattivo) e torna al punto a.
- se il bit i-mo di EO vale 0 e il conale i-mo é programmato in modo continuo torna al punto c.

2.8.6 Tipo di ciclo del controllore DMA

Sia nel modo singolo che continuo il controllore puó effettuare il trasferimento(supponiamo memoria → interfaccia) come segue:

- **ciclo doppio**:

  1. preleva il dato dalla memoria(comanda un ciclo di lettura nello spazio di memoria all'indirizzo specificato dal contenuto di MARi e memorizza il dato in un registro d'appoggio.
  2. emette un impulso sul piedino /DACKi per notificare all'interfaccia che la richiesta di servizio é stata accettata e trasferisce il dato, comandando un operazione di scrittura all'indirizzo specificato da IOARi nello spazio di I/O.

- **ciclo unico** comanda un 'operazione di lettura nello spazio di memoria all'indirizzo specificato dal contenuto di MARi. Contestualmente emette un impulso nel piedino /DACKi notificando all'interfaccia che puó prelevare il dato presente sul bus dati.

  ***Nota* 24** *Se il trasferimento avviene nel seguente modo, interfaccia → memoria, i transceiver vanno disabilitati.*

2.8.7 Problema del corto circuito

Ci sono due casi in cui si potrebbe avere un corto circuito:

1. Qunado il processore seleziona il controllore DMA per andare a leggere il contenuto dei suoi registri.

2. Quando il controllore DMA effettua un trasferimento da interfaccia verso la memoria principale, con il tipo di ciclo unico.

Analizziamo il funzionamento del transceiver. Per semplicitá assumeremo i transceiver che sopportano il bus dati come l'equivalente di un unico transceiver che supporta l'intero bus dati.

/EO	INTPUT	OUTPUT
0	0	0
0	1	1
0	Z	-
1	1	Z
1	0	Z
1	Z	Z

1. Il procesore seleziona il controllore DMA e va al leggere il contenuto die registri. Le varie interfacce e la memoria saranno disabilitate dato che avendo indirizzi diversi da quelli del controllore (/DD), le loro uscite saranno poste in alta impedenza (Z). Il processore effettua un operazione di lettura quindi il transceiver, si porta nello stato di far passare i dati da valle verso il processore. L'input al tranceiver avviene dunque dal lato della memoria ed interfacce, e L'output verso il processore e il controllore DMA. Per input si ha Z che produce un uscita non specificato (0 / 1 casuale). Ma il controllore produce delle uscite che sono collegati agli stessi collegamenti. quindi si é in presenza di un potenziale cortocircuito. Ne segue che il transceiver deve essere disabilitato, con l'esplicita generazione del segnale /DD verso il circuito di abilitazione.

2. Il controllore DMA comanda un'operazione di scrittura nello spazio di memoria all'indirizzo specificato dal contenuto MARi. Contemporaneamente i piedini D31-D0 del controllore saranno in alta impedenza. Il transceiver si trova nella condizione di far passare i dati da monte a valle (operazione di scrittura). L'input al transceiver dal lato del processore e controllore DMA e l'output dalla parte della memoria principale e interfacce. Di nuovo si il transceiver si vede in ingresso Z e produce in uscita un non specificato (0/1 casuale). L'uscita é collegata sullo stesso collegamento tra memoria e interfacce. Ne segue che anche in questa condizione si ha un potenziale circuito. Si puó utilizzare DAKi del controllore per disabilitare i transcever (funge da /DD)

Le interfacce gestite attraverso il controllore DMA devono, specialmente nel casi di trasferimento a ciclo unico. essere compatibili con i piedini del controllore DMA:

- **interfaccia d'uscita** : Non richiede il comando di scrittura per prelevare il dato dal bus dati. Deve intendere il segnale /DACKi come abilitazione e indirizzo del suo buffer di uscita.
- **interfaccia d'ingresso**: mutatis mutandis.

2.8.8 Trasferimento memoria-memoria in DMA

Analogo assembler del comando *MOVSlung.* . In questo caso vengono utilizzati due canali consecutivi (pari-dispari).

- nel canale pari non viene utilizzato il registro IOAR.
- nel canale dispari viene adoperato solo il registro MAR.

- é previsto solo il modo continuo e ciclo doppio.
  - modo continuo perché é piu efficiente.
  - ciclo doppio perché non si puó leggere e/o scrivere la memoria allo stesso istante.
- l'inizio del trasferimento avviene secondo opportuni valori dei bit del registro MODER, del canale pari.

2.8.9 Controllore Cache e controllore DMA

- Il controllore DMA viene montato a valle del controllore cache e a monte del circuito di pilotaggio dei bus dati.
- Il possesso del bus viene richiesto dal controllore DMA, sui piedini HOLD e HOLDA, al controllore Cache in presenza di memoria cache, visto che quest'ultimo é quello che gestisce i cicli bus, quando é attivo.
- Il controllore Cache si comporta alla pari del processore, ovvero pone le sue uscite in alta impedenza concedendo al controllore DMA il controllo del bus. Se é necessario, in vista di una richiesta di utilizzo del bus, il controllore cache puó allungare i cicli di bus. Il processore non si accorge affatto della gestione che ne viene fatta del bus, quindi non é necessario che esso ponga in alta impedenza le sue uscite.
- Nell'operazione di I/O i trasferimenti da interfaccia verso la memoria principale vengono comunemente effettuate utilizzando un buffer in memoria dedicata a tale operazione. Ne segue che la cache deve essere disabilitata per tale zona dove risiede il buffer. Una modifica del contenuto del buffer non ha nessuna conseguenza sulla memoria cache. Altrimenti si ha un eccessivo costo di operazioni di trasferimento tra memoria e processore.
- Il controllore DMA puó essere utilizzato per effettuare i trasferimenti tra la memoria centrale e quelle di massa, interessando una qualunque zona di memoria. Ne segue che é necessario invalidare la memoria cache visto che si altera la consistenza tra il contenuto della memoria cache e quella principale (anche di una sola parte in base al range di indici).
- Il controllore cache puó anche essere realizzato in modo che i piedini di indirizzo a valle siano bidirezionali. Il tal caso quando il DMA si vede arrivare sul piedino HOLDA, gli indirizzi da esso generati producono un
  1. accesso a un blocco della memoria cache.
  2. in casi di successo, si invalida il blocco stesso. (come fá???)

2.8.10 Interfaccia gestita in DMA

- L'interfaccia deve avere delle caratteristiche funzionali compatibili con il controllore DMA. Possedere dei piedini con cui effettuare delle richieste di servizio al controllore DMA e ricevere risposta alle richieste e segnale di fine operazione.

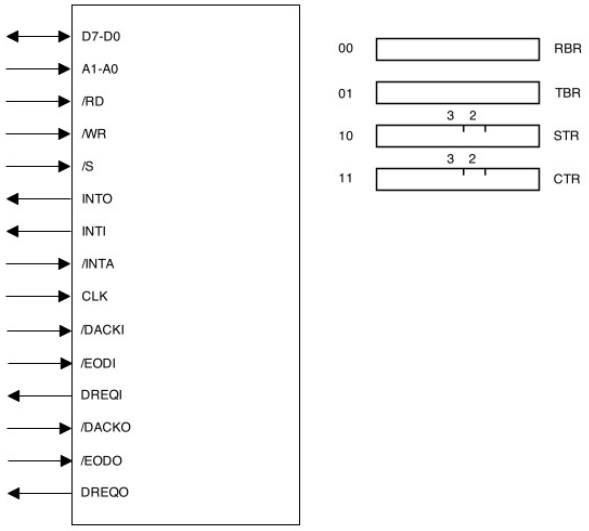
- Un interfaccia capace di operare con il ciclo doppio puó essere collegata a qualunque bus dati. con ampiezza pari alla lunghezza dei dati da trasferire. In sintesi si effettua un ciclo di I/O abilitando in tempi diversi il bus dati.

  ***Nota* 25** *un interfaccia che funziona con tipo di ciclo doppio trasferisce singoli byte e viene collegata al bus dati a 8 bit.*

- Un'interfaccia che funziona a ciclo unico puó essere collegata solo al bus a 32 bit dove si trova la corrispondente memoria. Infatti per ogni trasferimento il controllore DMA solo un unico ciclo di memoria e l'interfaccia deve avere i collegamenti compatibili con il bus a 32 bit. Al piú anche i segnali di abilitazione dei singoli byte.

Interfaccia a 8 bit

L'interfaccia va abilitata, via software a colloquiare con il DMA:



- effettua richieste di servizio solo se il bit n. 2 n. 3 del registro CTR valgono 1.
- l'interfaccia si disabilita automaticamente e mette a 0.
  - bit n. 2 quando riceve /EODI.
  - bit n. 3 quando riceve un impulso su /EODO.

STR

: puó essere utilizzato per verificare se é in corso un'operazione di ingresso / uscita.

- bit n. 2 segnala fine operazione di ingresso.
- bit n. 3 segnala fine operazione di uscita.

L'interfaccia puó essere inoltre abilitata a inviare una richiesta di interruzione (mediante i piedini INTI e INTO) per segnalare la fine di operazione di ingrasso / uscita. Per realizzare quanto detto si pone il bit n. 6 e n. 7 del registro CTR.
Tali bit vanno azzerati via software dal sottoprogramma di servizio associato all'interfaccia. Tale operazione si rende necessaria per informare che la richiesta é stata accettata. Supponendo che l'interfaccia operi con il modo di trasferimento singolo e tipo di ciclo doppio (per le ragioni analizzati prima).

Gestione di un'Interfaccia ad interruzione di programma

Una volta abilitata
ciclicamente

1. preleva un dato dal trasduttore esterno e lo immette nel buffer di ingresso RBR.
2. invia una richiesta di servizio al controllore attivando DREQi.
3. attende una risposta dal controllore (sul piedino /DACK) nel frattempo il controllore provvede ad effettuare un ciclo di lettura che preleva in contenuto di RBR e un ciclo di scrittura in memoria.
4. attende che /DACKI divenga disattivo e quindi chiude il colloquio con il controllore (disattivando DREQI)
   - se ha ricevuto un impulso sul piedino /EOI passa al punto 5.
   - passa al punto 1.
5. l'interfaccia si auto-disabilita a inviare ulteriori richieste di servizio per i trasferimenti in ingresso, e pone ad 1 il flag di fine operazione di ingresso del registro STR.
   - Se l'interfaccia é abilitata invia una richiesta di interruzioni (INTI) quando l'intera operazione di ingresso é terminata.

2.8.11 Operazione di ingresso in DMA

Dopo aver come viene organizzata un operazione di I/O in DMA dal punto di vista fisico. Si analizza ora come si organizza tali operazioni anche a livello software. Si effettuano operazioni di ingresso / uscita sfruttando il DMA nelle seguenti situazioni.

- l'operazione deve avviene in modo veloce, e una gestione a controllore di programma non é compatibile con i tempi di trasferimento richiesti (la gestione a interruzioni di programma sarebbe ancora piú lenta).
- il programma puó utilizzare a pieno il processore in attesa che l'operazione venga compiuta senza essere interrotto dal driver che effettua i singoli trasferimenti.

Ingresso dati

(Modo singolo, tipo ciclo doppio. fine operazione segnalata a interruzione di programma) Vanno effettuate i seguenti passi

- inizializzare il canale i-mo del DMA a cui é collegata l'interfaccia
- immettere nel registro TCR-i in numero di byte da trasferire diminuito di 1.
- immettere in MAR l'indirizzo fisico dalla prima locazione di memoria coinvolta nel trasferimento.
- inizializzare l'interfaccia l'interfaccia a operare in modalitá prevista con inizializzazione del canale stesso, inizializzando opportunamente il registro MODER.
- abilitare l'interfaccia a inviare richieste di servizio ogni volta che l'interfaccia ha un nuovo dato in RBR.
- abilita l'interfaccia a inviare richieste di interruzione alla fine dell'intera operazione di ingresso.
- azzerare una variabile di sincronizzazione.

***Nota* 26** *Il DMA montato nello spazio di memoria per cui se si vuol scrivere nei suoi registri si usano le istruzioni che operano sullo spazio di memoria.*

```
#programma main_in
. include "servizio"
. global start
. data
num_in: . long 0
. buffer_in: . fill 100,1
. text
start:
main_in;
   . . .
   #dmastat_in
   movl num_in, %ecx
   movl $buffer_in, %esi
   int $248
   . . .
   altre operazioni
   . . .
   #chiamata del dmawait_in
   int $249
   utilizzo dati
   . . .
   jmp dos

#nomi sembolici
. SET . . .
. SET OCW2, . . .
. SET MAR, . . .
. SET IOAR, . . .
. SET TCR, . . .
. SET MODER, . . .
. SET  RBR, . . .
. SET CTR, . . .
#dati comuni
. data
sincr_in: . byte 0
. text
ini: #inizializzazione del controllore delle interruzioni
```

```
   call ini_contr
   #inizializzazione della tabella IDT
   . . .
   movl $dmastart_in, %eax
   movb $248,%bl
   call ini_tab
   movl $dmawait_in, %eax
   movb $249,%bl
   call ini_tab
   movl $dmastart_in, %eax
   movb $9,%bl
   call ini_tab
   ret
ini_contr:
   . . .
   ret
ini_tab:
   . . .
   ret

dmastart_in:
rout_248:
    pushl %eax
    pushl %ecx
    pushl %edx
    #caricamento in MAR l'indirizzo contenuto in %esi
    movl %esi,MAR
    movl $RBR,IOAR
    #caricamento in TCR il contenuto di ECX-1
    decl %ecx
    movl %ecx,TCR
   #abilitazione del canale 1 e sua inizializzazione con
   #modo di trasferimento singolo e tipo ciclo
   #doppio
   movl $33,MODER
   #abilitazione interfaccia a operare in DMA
   # e inviare richieste di interruzione.
   movw $CTR, %dx
   inb %dx,%al
   orb $0x44, %al
   outb %al,%dx
   #inizializzazione della variabile sincr_in
   movb $0,sincr_in
   popl %edx
   popl %ecx
   popl %eax
   iret

dmawait_in:
rout_249:
#abilitazione processore ad accettare richieste
STI
# attesa di sincronizzazione
dmaciclo_in:
  cmpb $0,sincr_in
  je dmaciclo_in
  iret

dmadriver_in;
rout_9: #8  +1
   pushl %eax
   pushl %edx
   STI
   #segnalazione che l'operazione \'e finita
   movb $1,sincr_in
   #rimozione di interruzione (CTR, bit n 6 = 0)
```

```
movw $CTR,%dx
inb %dx,%al
andb $0xBF, %al
outb %al, %dx
#invio del comando EOI
CLI
movb $0x20, %al
movw $OCW2, %dx
outb %al,%dx
popl %edx
popl %eax
iret
```

3 Aspetti Architetturali avanzzati

- Memoria virtuale (virtualizzazione della memoria centrale)
- Multiprogrammazione
- Protezione

3.1 Memoria virtuale Paginata

Premesso che

- memoria principale = memoria centrale = memoria fisica,
- memoria secondaria = memoria di massa.

Il processore per poter eseguire un qualsiasi programma (programma utente , routine di servizio ecc.) necessita che una copia (codice + dati) dello stesso ve nga trasferita dalla memoria di massa alla memoria centrale. Questo ci suggerisce l'esistenza di una limitazione dovuta alle dimensioni della capacita della memoria centrale che deve ospitare, oltre alle primitive di sistema, anche il programma da eseguire. Tale problema si presenta nel caso si voglia, per esigenze, mandare in programmi che occupano una dimensione maggiore della dimensione della memoria centrale disponibile sul calcolatore, e piú di un programma ,con tali caratteristiche, alla volta. Come verrá analizzato in seguito si ricorre ad meccanismo di virtualizzazione della memoria centrale, per realizzare quanto premesso. Per realizzare tale virtualizzazione si utilizza la memoria di massa come estensione della memoria centrale.
Le due tecniche di virtualizzazione possibili sono

Si separano i concetti di indirizzamento e memoria fisica

- *paginazione*
- *segmentazione*

3.1.1 Tipologia di spazi di memoria

Un programma composto da istruzioni eterogenei (istruzioni operative e di controllo) per riferire la memoria, utilizzano degli *indirizzi logici.*

***definizione* 1** *L'insieme degli indirizzi logici costituiscono lo* SPAZIO LOGICO*.*

Il processore quando preleva istruzioni o tratta operandi effettua via HARDWARE lettura/scrittura in memoria utilizzando degli *indirizzi fisici.*

***definizione* 2** *L'insieme degli indirizzi fisici costituiscono lo* SPAZIO FISICO*.*

La memoria fisica (quella effettivamente cablata) é un sottoinsieme dello spazio fisico

- **caso 1**: L'indirizzo logico coincide con l'indirizzo fisco.
  In questo caso un programma puøutilizzare al massimo uno spazio di memoria avente dimensione pari a quella della memoria fisica (*sistema monoprogrammata*).

- **caso 2**: Si rende differenti l'indirizzo logico e quello fisico quando la dimensione del programma (codice + dati + nucleo(codice + dati)) da eseguire é maggiore rispetto alla dimensione della memoria fisica.

$$\Downarrow$$
meccanismo di virtualizzazione
memoria fisica
+
memoria di massa

3.1.2 Paginazione

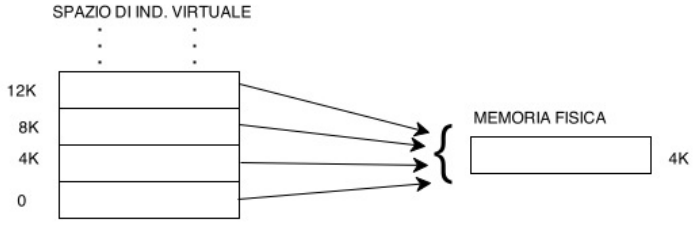
Un calcolatore con capacitá di indirizzamento pari a 16 bit é in grado di indirizzare 64 Kbyte (o locazione).

Lo spazio di indirizzamento

$$\text{S\_indirizzi} = \{0, \ldots, 65535\} \tag{5}$$

viene pensato suddiviso in blocchi funzionali della stessa dimensione. al piú, pari a quella della memoria fisica.Il calcolatore puó avere una memoria fisica costituita da un numero di locazioni molto minori rispetto alla dimensione dello spazio di indirizzamento.Supponiamo per semplicitá 4kbyte. Si presentano allora due casi, in cui si puó o meno ricorrere al meccanismo di virtualizzazione.

- **Nessuna virutalizzazione** In questo caso é necessario distinguere gli indirizzi compresi tra 0 e 4k-1 (indirizzabili) da quelli superiori od uguali a 4k(indirizzamento senza una corrispondenza in memoria fisica. In caso di accesso a locazioni superiori alla (4k-1)-i-ma locazione si ha un errore.
- **Virtualizzazione paginata** Il programma utilizza indirizzi virtuali (quando la paginazione é attiva) che vengono tradotti in indirizzi fisici poi.
  - {0,4k,8k,...} $\rightsquigarrow$ nell'indirizzo fisico 0.
  - {1,4k+1,8k+1,...} $\rightsquigarrow$ nell'indirizzo fisico 1.



- La memoria principale viene salvata su disco.
- Viene caricato dal disco il blocco da 4k in base all'indirizzo virtuale (generato dal processore)

- L'indirizzo virtuale viene tradotto in modo da accedere all'opportuna locazione di memoria.

***definizione* 3** *Il blocco elementare della suddivisione in blocchi dello spazio di indirizzamento virtuale prende il nome di* PAGINA.

Il meccanismo di virtualizzazione paginato, richiede, oltre alla traduzione degli indirizzi, anche lo spostamento delle pagine dalla memoria secondaria (o di massa) alla memoria principale e viceversa (*Swap*).

Vanttaggi:

- I programmi possono essere scritti come si il sistema di disponesse di una quantitá di memoria principale pari allo spazio di indirizzamento virtuale.
- La dimensione dei programmi é limitata solo dalla capacitá della memoria di massa.
- Il meccanismo di paginazione é trasparente al programmatore.

Lo spazio di indirizzamento virtuale é diviso in un certo numero di pagine di dimensione fissa (da 512 a 64 Kbyte).

***definizione* 4** *Lo spazio di memoria fisica é suddiviso in blocchi della stessa dimensione pari alla dimensione di una pagina, e prende il nome di* PAGE FRAME.

***Esempio* 3.1** *Supponiamo di aver un sistema con capacitá di indirizzamento virtuale pari a 32 bit (ad esempio la capacitá dei registri per l'indirizzamento, un su tutti IP), 32K locazioni di memoria fisica cablata e pagine dimensione 4k.*

L'indirizzamento a 32 bit deve essere tradotto in fisico a 15 bit.

$$32kbyte = 2^5 * 2^{10}Byte = 2^{15}Byte$$

3.1.3 Meccanismo per la paginazione

La *MMU*, una delle 5 unitá componenti il processore, é l'effettore ultimo della traduzione degli indirizzi rispettando tale sequenza nel caso piú generale (tenendo conto, in maniera semplificata, di entrambe le tecniche di virtualizzazione).

- *indirizzo logico* $\rightsquigarrow$ *indirizzo virtuale* $\rightsquigarrow$ *indirizzo fisico.*

Tale traduzione viene effettuata utilizzando in due passi successivi. Nel primo viene utilizzata una tabelle par passare da indirizzo logico a indirizzo virtuale. La seconda per passare da indirizzo virtuale a fisico. definito

$\mathbf{L^2}$: spazio di indirizzamento logico (bidimensionale)

$\mathbf{V}$: spazio di indirizzamento virtuale

$\mathbf{F}$: spazio di indirizzamento fisico

$\underline{x} = \underline{\text{indirizzo\_logico}}$

$v : \mathbf{L^2} \rightarrow \mathbf{V}$

$f : \mathbf{V} \rightarrow \mathbf{F}$

indirizzo_fisico $= f(v(\underline{x}))$

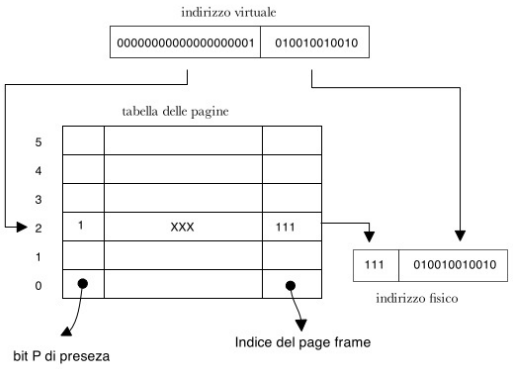
Si nota che la funzioni matematiche corrispondono alle leggi di corrispondenza e definiti mediante delle tabelle.No é stato definito un dominio visto che questo dipende dalla realizzazione vera e proprio dei meccanismi di virtualizzazione, in base all'architettura coinvolta.

Vediamo una realizzazione di quanto esposto, riprendendo come riferimento l'esempio precedente. Verrá analizzata solo la paginazione. La tabella di corrispondenza contiene gli indirizzi fisici dei page-frame. Nel processo di traduzione , una parte dell'indiririzzo virtuale individua un entrata nella tabella dicorrispondenza , e l'altra parte per individuare la locazione riferita all'interno della pagina stessa. Scegliendo di suddividere lo spazio di indirizzamento virtuale (2^32)in pagine di dimensione pari a 4K locazioni, si ha che lo spazio virtuale é composto da 1M di pagine. Ne segue che per indirizzare 1M di pagine é necessario disporre di 20 bit di indirizzo, da cui la partizione:

Lo spazio di indirizzamento fisico é composto da 8 page-frame essendo di 32 Klocazioni

- *indice*: MSB 20 bit per individuare un entrata della tabella.
- *spiazzamento*: 12 bit indirizzo all'interno della pagina($4k = 2^{12}$) . per individuare l'indirizzo della prima locazione coinvolta nel trasferimento.

Ovviamente non tutte le pagine della memoria virtuale possono essere presenti in memoria fisica.



3.1.4 Generalizzazione della paginazione

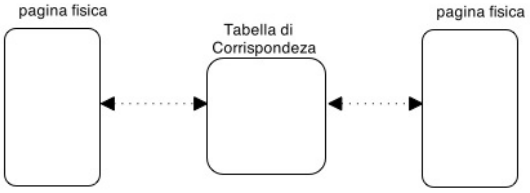
Come analizzato prima Il meccanismo di virtualizzazione della memoria principale consente di ottenere la memoria virtuale a partire dalla memoria fisica e memoria di massa. L'indirizzo virtuale viene pensato partizionato in due componenti.:

- Indirizzo virtuale di pagina (a bit).
- Indirizzo di riga (b bit).

$$\Downarrow$$

Ne segue che la memoria virtuale risulta suddivisa in 2^a di ampiezza 2^b.

Anche la memoria fisica viene pensata suddivisa in page-frame di dimensione pari 2^b. L'indirizzo virtuale di pagina viene utilizzato per

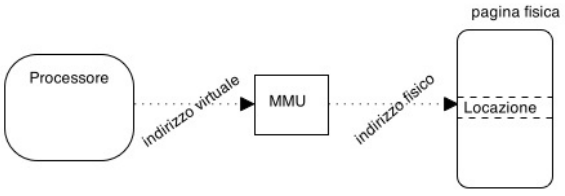


accedere ad una determinata entrata della tabella.

- **descrittore di pagina**: Occupa 4byte, e specifica l'indirizzo fisico della page fisica.
- **l'indirizzo virtuale di riga**: Non necessita di essere tradotto, e rimane inalterato, visto che le pagine sono allineati.

3.1.5 Memory Management Unit e Page-fault

Ogni volta che il processore genera un indirizzo di memoria la MMU provvede via hardware a effettuare la traduzione di indirizzo da virtuale a fisico. Ogni descrittore di pagina contiene, oltre all'indirizzo



fisico anche alcuni informazioni sulla presenza di una pagina virtuale nella memoria fisica. Se la pagina non é presente la MMU

- genera un *page_fault*.

Paginazione su domanda

- memorizza in un registro dedicato, detto *registro speciale* l'indirizzo non tradotto.
- memorizza nella pila il registro EIP (non ancora aggiornato, in modo da gestire correttamente i fault).

La routine che va in esecuzione esecuzione (routine di trasferimento) utilizza il contenuto del registro speciale, per completare a livello software la gestione della memoria virtuale(vedremo come viene effettuata tale gestione).

- provvede a trasferire la pagina dalla memoria di massa alla memoria fisica. Il campo del descrittore riservato all'indirizzo fisico pu'øessere utilizzato dalla routine di trasferimento, come coordinate per individuare la pagina nella memoria di massa.(vedremo come).
- aggiornare la tabella di corrispondenza. (vedremo come).
- mediante la IRET si rimanda in esecuzione l'istruzione che ha generato il page-fault.

L'istruzione viene rieseguita, e la MMU traduce l'indirizzo senza nessuna conseguenza.

Vantaggi
:

- come accennato non é necessaria la rilocazione di un programma quando la paginazione é attiva, poiché per ogni programma viene prevista una tabella di corrispondenze e la memoria risulta sempre libera a partire dall'indirizzo 0.

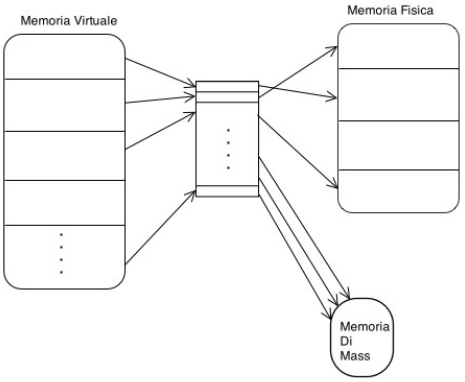
***Nota* 27** *La gestione della memoria virtuale é affidata alle routine del sistema operativo.*

3.1.6 Riesecuzione delle istruzioni

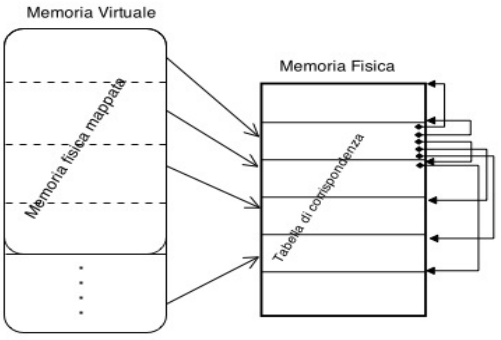
Un page-fault, un eccezione di tipo fault, é generata in assenza di una pagina in memoria fisica.

- il programma viene interrotto dalla routine di trasferimento.
- l'effetto della chiamata, produce la memorizza in pila EIP (non ancora aggiornato visto, visto che si é trattato di un interruzione di tipo fault e che l'esecuzione dell'istruzione non é terminata, il valore dei registri d'appoggio non vengono memorizzati nei registri del processore) che punta all'istruzione che ha generato fault.
- quando viene trasferita la pagina mancante si riesegue l'istruzione, come effetto della IRET della routine di trasferimento.

3.1.7 Memoria fisica in memoria virtuale



Via software é possibile accedere solo alla memoria virtuale, e le operazioni di lettura/scrittua in memoria avvengono solo a traduzione di indirizzo eseguita. Ne segue che i page-frame della memoria fisca memoria fisica per essere acceduta direttamente devo risiedere in memoria virtuale. L'accesso a tale pagine fá si che la traduzione sia un operazione simile a quanto esposto, oppure si si posizionano i page-frame nelle prime pagine della memoria virtuale, l'operazione di traduzione é superflua.
Nel secondo caso dunque si ha che l'indirizzo fisico coincide con l'indirizzo virtuale, fintanto che l'indirizzo é minore di 2^k, se 2^k é la dimensione della memoria realmente cablata nel calcolatore. In sintesi si scegli di mappare la memoria fisica in memoria virtuale a partire dalla locazione 0. In tal modo il processo di traduzione porta al medesimo indirizzo. Il vantaggio di questo tipo di scelta risiede nel continuitá nell'indirizzamento.
Occorre dunque prevedere che nella tabella di corrispondenza vi siano descrittori che hanno come indirizzo fisico quello dei page-frame con l'informazione di presenza persistente(la pagina non deve essere rimpiazzabile). Come vedremo in seguito se la memoria fisica é totalmente occupata e si verifica un page-fault si ha necessita di effettuare uno swap-out di una *pagina* che risiede nella memoria fisica ed uno swap-in della pagina per la quale si ha avuto il fault.



Non ha nessun senso rimpiazzare una pagina che mappa un page-frame. infatti il page-frame é sempre presente, e un tentativo di swap-out equivale ad un tentativo di rimozione fisica di un page-frame.

***Nota* 28** *Un page-frame mappato in memoria, mediante un qualche descrittore, puó contenere una qualunque pagina, purché rimpiazzabile, per realizzare la memoria virtuale stessa.*

***Nota* 29** *Se si sceglie di mappare la memoria fisica in memoria virtuale si avranno tanti descrittori quanti sono i page-frame da mappare. In caso di un page-fault un descrittore di pagina coinvolto, dopo la chiamata della funzione di trasferimento, conterrá l'indirizzo fisico del page-frame che mappa la pagina trasferita dalla memoria di massa alla memoria fisica. Si noti allora che si hanno almeno due descrittori che hanno lo stesso indirizzo fisico, ma con una sostanziale differenza.*

3.1.8 Paginazione nel processore PC

Si adoperano 4 registri dedicati da 32bit ciascuno, accessibili al programmatore:

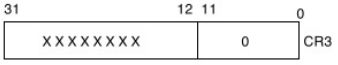
- CR0: PG = bit n.31 e posto ad uno abilita il meccanismo di paginazione.
- CR1: usi futuri.
- CR2: Contiene l'indirizzo virtuale non tradotto in caso di page-fault.
- CR3: Contiene l'indirizzo del direttorio.

La paginazione si abilita ponendo ad 1 PG. Si é scelto che la dimensione di ogni pagina sia pari a 4kbyte, ne segue che lo spazio della memoria virtuale é composto da 1M di pagine.

***Nota* 30** *La tabella di corrispondenza tra pagine virtuali e pagine page-frame é realizzata in due livelli. Ovvero La tabella di corrispondenza é paginata a sua volta.*

La realizzazione della tabella in due livelli é data da:

- **direttorio delle pagine**: é un page-frame il cui indirizzo fisico é dato dal valore del registro dedicato CR3.

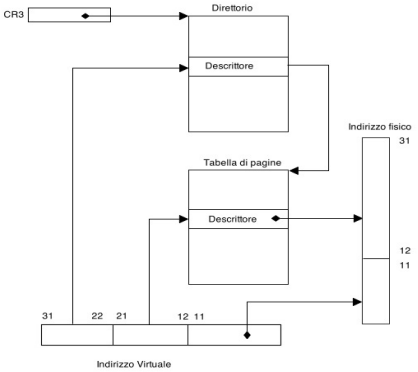


  - é costituito da 1024 *descrittori di tabella delle pagine* (ogni descrittore é lungo 4 byre).
  - Ogni descrittore contiene l'indirizzo fisico (20 bit) di un page-frame. ed un byte di accesso.
- **tabelle delle pagine**: é una pagina il cui indirizzo fisico é dato dal valore della parte di un descrittore di tabella.
  - é costituita da 1024 *descrittori di pagina.*
  - Ogni descrittore contiene l'indirizzo fisico (20bit) di un page-frame. ed un byte di accesso.

Traduzione a due livelli

La traduzione prevede due accessi in memoria come descritto in figura:

- 10 bit piú significativi dell'indirizzo virtuale servono ad individuare un descrittore di tabella delle pagine.
- 10 bit successivi dell'indirizzo virtuale servono ad individuare un descrittore di pagina (all'interno della tabella delle pagine).



La tabella é formata da $1 + 1k$ pagine ($1 + 1024$ pagine) e quindi al piú occupa una zona di memoria pari a (4Kbyte + 4Mbyte).

3.2 Gestione della tabella di corrispondenza

Avevamo visto che per effettuare l'accesso diretto é necessario mappa la memoria fisica in quella virtuale se l'accesso é software. Quando si verifica un page-fault la funzione di trasferimento deve aggiornare la tabella di corrispondenza dopo il trasferimento. Ne segue che sia il direttorio che la tabella delle pagine siano presenti nella memoria virtuale.

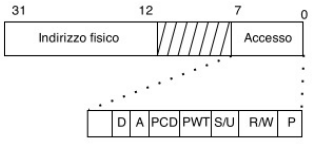
Questo indirizzo virtuale si deve stabilire a priori? E la routine come fa a conoscerlo? E il sistema stesso che decide lindirizzo virtuale: ci sará una routine di inizializzazione che, prima di attivare la paginazione, predispone la tabella che mappa il direttorio in memoria virtuale. Quindi, é sufficiente che la routine di inizializzazione lasci questo indirizzo in una variabile globale, accessibile alla routine di page fault. Se, peró, si mappa tutta la memoria fisica in memoria virtuale, la routine di page fault puó usare direttamente lindirizzo fisico contenuto in CR3, come se fosse virtuale.

La paginazione delle tabelle delle pagine comporta trasferimenti tra memoria di massa e memoria centrale, ne segue che tale tabelle devono risiedere a indirizzi virtuali diversi da quelli in cui risiede la memoria fisica, le cui pagine sono sempre presenti in modo persistente. (vedi pagina 46 domande e risposte)

3.2.1 Descrittore di tabella e di pagina

Byte di accesso

Il byte di accesso contiene i seguenti bit:



- P bit di presenza.
- D,A utili ai fini del rimpiazzamento.
- PCD, PWT gestione della cache.
- S/U, R/W gestione della protezione.

Descrittore di pagina

- P (present): se vale 0 viene generato il page-fault dalla MMU.
- A (accessed) : viene posto a 1 dalla MMU. Ogni volta che avviene un accesso alla pagina (traduzione di indirizzo).
- D (Dirty): viene posto dalla MMU quando avviene un accesso in scrittura alla pagina stessa (scrittura in memoria).
- PCD (Page Cache Disable): specifica se per quella pagina la memoria cache deve essere disabilitata e determinano lo stato del piedino CD del processore.
- PWT (Page WriteThrough) : specifica se per quella pagina la memoria cache deve essere gestita in modalitá write-trough e determina lo stato del piedino WT del processo.

- S/U (System / User): Specifica il livello di privilegio della pagina.
- R/W: Specifica se la pagina é solo leggibile o anche screvibile.

Descrittore di tabella

Per i descrittori di tabella sono significativi solo i bit P ed A. Il bit D non é significativo in quanto accessi in scrittura a una tabella delle pagine puó avvenire via software solo se questa risiede in memoria virtuale. Se cosí fosse viene gestito (come logico sia) il bit D del descrittore di pagina che fa risiedere tale pagina (tabella delle pagine per intero). I bit PCD e PWT non vengono utilizzati perché la memoria cache riguarda solo le pagina.Infine il bit S/U vale System. La tabella di corrispondenza risulta paginata:

- il bit P é previsto anche nel descrittore di tabella delle pagine. Se vale 0 viene generato un page-fault dalla MMU.
- in caso di page-fault la routine di trasferimento deve esaminare due descrittori :
  - nel direttorio
  - nella tabella delle pagine coinvolta.

L'esame software dei due bit determina se si tratta di mancanza di tabella o pagina.

3.2.2 Trasferimento delle pagine

Si parte da una situazione iniziale in cui sono presenti in memoria fisica (mappata nella virtuale) solo il direttorio (tanti direttori quanti sono programmi) e le tabelle delle pagine che consentono all'intera tabella di corrispondenza di essere posizionata nella memoria virtuale (direttorio e tabelle di pagine) e la tabelle che consentono di mappare la memoria fisica nella virtuale.

Il trasferimento di una pagina da memoria di massa a memoria centrale avviene su domanda, quando viene generato un indirizzo che causa un page-fault (non viene tradotto). La routine di trasferimento che va in esecuzione esamina CR2 (quindi i descrittori coinvolti) e determina dove si trova in memoria di massa la pagina da trasferire in memoria fisica. In caso di rimpiazzamento la routine sceglie la pagina da rimpiazzare in base al valore delle variabili associate alla pagina che contengono una statistica di utilizzo.

- LFU (least frequently used)
- LRU (least recently used)

Viene rimpiazzata la pagina la cui variabile associata ha valore minimo. L statistica si effettua in base al valore assunto dal bit A dei vari descrittori di pagina e di tabelle delle pagine quando va in esecuzione la routine di timer per effetto di un interruzione prodotto da un timer. La routine di timer effettua le statistiche ed azzera il bit A, di ogni descrittore.

- LFU: si associa ad ogni pagina una variabile contatore. Viene incrementata solo se il valore di A vale 1.
- LRU: si associa ad ogni pagina una variabile registro a scorrimento. Viene inserito in testa il valore di A.

***Nota* 31** *Le variabile associata a una tabella delle pagine nono puó mai avere valore inferiore rispetto al valore della variabili associate alle pagine da esse riferite.*

- a prioritá di condizione viene scelta una pagina e non una tabella: altrimenti non si puó accedere via software a tutte le pagine da essa riferite.
- se la tabella non riferisce nessuna pagina, tale tabella non viene ricopiata nella memoria di massa (la copia esistente é ancora valida).

Se viene selezionata una pagina vuol dire che mentre era in memoria non é stata modificata (D=0), e non va salvata nella memoria di massa, la copia in memoria di massa é ancora valida.

Quando si sceglie di rimpiazzare una tabella delle pagine si sceglie di non riferire per un certo intervallo di tempo 4 kbyte di memoria virtuale. Rimpiazzando una tabella delle pagine si sceglie di non riferire per un certo intervallo 4 Mbyte di memoria con un alta probabilitá che questa venga riferita.

3.2.3 Trasferimento delle tabelle delle pagine

Quando una routine di trasferimento seleziona per il rimpiazzamento una tabella delle pagine, questa non viene ricopiata in memoria di massa.

Il bit D del descrittore della tabella delle pagine é non significativo. Il descrittore della tabella che fa risiedere questa tabella in memoria virtuale che bisogna modificare, in particolare il bit D del suo byte di accesso .

- modifiche software e hardware avvenuti mentre la tabella delle pagine si trovava in memoria fisica vengono perse.
- Questo fatto non produce alcun danno, dal momento che la tabella, quando viene caricata in memoria fisica. non vi sono pagine da essa riferite (nella memoria fisica, ma contiene le informazioni per individuare le pagine nella memoria fisica).
  Per ogni descrittore
  - Nel byte di accesso pone i bit P, A, D a zero (dalla routine di trasferimento)
  - L'indirizzo fisico non é significativo.

Quando si rimpiazza una tabella delle pagine si richiede che venga messo a 0 il bit P sia nel descrittore nel direttorio che nel descrittore della tavella che la fa risiedere in memoria virtuale.

I descrittori da modificare, ovviamente, sono individuati dalle componenti dell'indirizzo virtuale.

3.2.4 Buffer dei descrittori di pagina

Il meccanismo di traduzione (corrispondenza) richiede due accessi in memoria fisica,

- al direttorio
- alla tabella della pagine.

Quest'operazione richiede inevitabilmente del tempo, prima che venga generato l'indirizzo fisico vero e proprio per compiere il ciclo comandato dal processore.

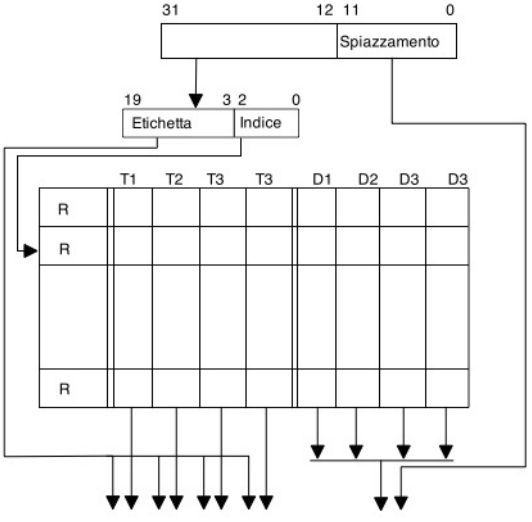
TLB (Translation Lookaside Buffer)

Una sorta di memoria cache associativa a insiemi dei descrittori.

- Il processore é dotato di un buffer interno destinato a memorizzare i descrittori di pagine utilizzati recentemente (non ha senso memorizzare i descrittori di tabella delle pagine).
- La MMU nel tradurre un indirizzo da virtuale a fisico accede prima al TLB.
  - # **hit** usa l'indirizzo fisico contenuto nel descrittore individuato in TLB.
  - # **Miss** traduce l'indirizzo come avevamo analizzato. A partire da CR2 si effettua l'accesso al direttorio e alla tabella delle pagine.

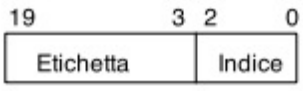
Organizzazione del TLB

- Il TLB é costituito da 8 insiemi ciascuno dei quali prevede
  - un campo R.
  - 4 gruppi
    - ogni gruppo é composto dal campo Ti(Tag) e Di(Dati)
- dal punto di vista fisico il TLB é composto da 3 banchi di memoria ciascuno di 8 righe.
  Una riga contiene
  - il campo R.
  - il campo Ti.
  - il campo Di.



La MMU considera i 20 bit piú significativi dell' indirizzamento virtuale cosí partizionato:

- indice (3bit)
- etichetta (17bit)



- L'indice (di 3 bit) seleziona un insieme all'interno del TLB.
- L'etichetta é confrontata con ognuno dei 4 campi Ti dell'insieme.
  - **hit** In caso di successo il campo Di contiene l'indirizzo fisico.

– **miss** il campo R relativo ad ogni insieme viene utilizzato per il rimpiazzamento e gestitio con un pseudo LRU.

Il TLB copre puó contenere al piú 32 descrittori di pagina, quindi copre fino a *32 x 4 Kbyte = 128 Kbyte*. Circa il 98-99% la MMU trova l'informazione di corrispondenza.

Vediamo piú in dettaglio

- **Campo Tag**: (contiene)
  - bit V (validitá)
  - Ec (etichetta)
  - D (Dirty)
  - S/U
  - R/W

  S/U e R/W vengono utilizzati dal meccanismo di protezione.
- **Campo Di**: (contiene)
  - indirizzo fisico della pagina.
  - bit PCD
  - bit PWT

  PCD, PWT vengono utilizzati per il controllo della memoria cache.

Il TLB produce successo se :

- V = 1 (E = Ec).
- D vale 0 e l'accesso é in lettura.
- D vale 1, l'accesso puó essere qualunque .
- la regole di protezione sono soddisfatte.

La routine di trasferimento che esamina il bit non puó accedere alla cache (TLB) visto che non ci sono istruzioni che permettano tale accesso.

Consideriamo i descrittori in TLB:

- Questi hanno il bit A = 1 nei descrittori nelle tabelle delle pagine
- Quando avviene un accesso in scrittura ma il bit D é 0 si ha un fallimento. ⇒ traduzione classica: il descrittore viene modificato (D= 1) ⇒ viene trasferito in TLB alla stessa posizione. Se il fallimento é dovuto a :

  L'aggiornamento del bit D nono avviene direttamente.

  – V = 1, $E \neq E_c$ ⇒ Causa una traduzione di indirizzo.

  – S/U , R/W ⇒ si genera un'eccezione per il non rispetto diprotezione.
- Se via software avviene una qualunque modifica ai descrittori di pagina presenti anche nel TLB, il gruppo relativo va invalidato. (INVLPGindirizzo virtuale). Si é in questa situazione quando tutti i bit A vengono posti a 0 dalla routine di timer.

  Accede ad esempio quando una pagina viene rimpiazzata e il suo descrittore viene modificato (P=1 → P=0)
- Per invalidare l'intero TLB si puó caricare una nuova quantitá in CR3.

3.2.5 Controllore DMA e indirizzi fisici

Avevamo visito che il controllore DMA é dotato di 4 registri per ogni calale per il trasferimento. Ogni canale é dotato di un registro MARi che contiene l'indirizzo della locazione di memoria coinvolta nel trasferimento. In presenza di memoria virtuale tale indirizzo é quello fisico. In questo modo il controllore non é costretto ad effettuare nessuna traduzione di indirizzo. La possibilitá che il controllore possa operare direttamente con indirizzi fisici, é data dal fatto che gli indirizzi coinvolti sono consecutivi, e la consecutiviá all'interno del page-frame si mantiene anche dopo il processo di traduzione dell'indirizzo.
Quanto detto comporta :

- la primitiva che inizializza un canale deve accedere alla tabella delle pagine per determinare l'indirizzo fisico della locazione a partire dalla quale si ha il trasferimento.
- le pagine su cui opera il controllore DMA devono risiedere (permanentemente) in memoria fisica, cioé non soggette al meccanismo di rimpiazzamento.Per una garanzia di consistenza tale pagine non devono essere soggette neanche al meccanismo di memoria cache.

4 Nucleo di un sistema multiprogrammato

- Multiprogrammazione
- Processi.
- Protezione
- Primitive di nucleo.
- Primitive semaforiche
- Primitive di I/O
  - ad interruzione di programma
  - ad accesso diretto alla memoria (DMA)
- driver
- processi esterni

4.1 Multiprogrammazione

Un sistema di elaborazione monoprocessore puó eseguire un solo programma alla volta. A divisione di tempo, esegue in *concorrenza* piú di un programma, dando luogo ai sistemi di elaborazione *multiprogrammati*. I programmi possono appartenere :

- allo stesso utente
- ad utenti diversi.

Per realizzare sistemi multiprogrammati si prevede una forma di *virtualizzazione* del processore.

descrittore di processo

Si associa ad ogni programma una *tabella di stato*, che contiene una copia dei registri del processore (registri virtuali) ed altri informazioni.

***definizione* 5** *Un programma in esecuzione su un processore virtuale prende il nome di* PROCESSO *(o task).*

***definizione* 6** LA TABELLA DI STATO *di un processo prende il nome di descrittore di processo (o di task).*

Il processore reale quando abbandona l'esecuzione di un programma Pa per un'altra del programma Pb, si dice che avviene una *commutazione di contesto*. Per realizzare tale commutazione:

- i valori attuali del processore reale vengono salvati nel descrittore associato al programma Pa
- nei medesimi registri si caricano nuovi valori a partire dal descrittore del processore Pb

Un programma quando viene ripreso per essere eseguito si parte dal punto di arresto precedente (valore di EIP salvato individua l'istruzione successiva da eseguire), e questo é dovuto ai passi di salvataggio / ripristino del valore dei singoli registri.

Si hanno cosí tanti processi virtuali quanti sono i programmi in esecuzione, e ogni processore virtuale é caratterizzato dalla tabella di stato. Un processore virtuale alla volta ha il possesso del processore reale, in base alla politica di gestione del sistema che si vuol realizzare.

4.1.1 Spazio di indirizzamento di un processo

Un processo ha un proprio spazio di indirizzamento puó coincidere:

- con lo spazio di indirizzamento del processore
- con un sottoinsieme dello stesso.

In presenza di memoria virtuale, tutti i processi hanno:

- lo stesso spazio virtuale
- spazio fisico differente
- lo stesso spazio di I/O, in quanto non é virtualizzato.

Una commutazione di contesto porta ad una commutazione di spazio di memoria fisico. Per realizzare quanto detto, per una consistenza globale di sistema:

- Si prevedono delle tabelle di corrispondenza diverse da processo a processo
- Sono individuate in memoria dal descrittore del processo stesso
  - Ogni descrittore di processo prevede anche il registro virtuale CR3v che specifica l'indirizzo fisico del direttorio delle pagine.
  - Quando viene caricato CR3v in CR3 (commutazione di contesto) si ha la commutazione di contesto, nuovo direttorio nuove tabelle delle pagine.
  - A paritá di indirizzi virtuali si hanno indirizzi fisici diversi da processo a processo.
- Si prevede anche una zona comune dove risiede il sistema operativo e una zona per la comunicazione fra i processi.
  - Tale zone hanno gli stessi indirizzi virtuali a cui corrispondono gli stessi indirizzi fisici.

    ***Nota* 32** *La legge di corrispondenza deve essere la medesima per la zone di memoria condivisa. Questo si concretizza dicendo che nelle tabelle delle pagine dei singoli processi deve comparire, per uno stesso indirizzo di pagina virtuale lo stesso indirizzo fisico. E questo non ha nulla a che fare con il fatto che la memoria fisica venga mappata agli stessi indirizzi virtuali.*

    I processi possono condividere

    * singole pagine

    * tabella delle pagine e quindi tutte le pagine individuate da quella tabella: nei direttori dei vari processi compare lo stesso indirizzo di tabella delle pagine.

La *multiprogrammazione* viene gestita interamente dalle routine del sistema operativo. Tutti i processi hanno nella zona di memoria comune agli stessi indirizzi virtuali, le pagine relative al sistema operativo. Ne segue che una commutazione di contesto (cambio del valore CR3) non influenza il funzionamento del sistema operativo.

In teoria é possibile che una *page-frame condivisa* corrisponda ad una pagina virtuale avente indirizzo differente da processo a processo. Questo puó comportare ad un inconsistenza globale del sistema.

Nei descrittori di pagina dei vari processi se compare ad indirizzi virtuali differenti l'indirizzo fisico del page-frame condiviso

***Esempio* 4.1** *I processi Pa , Pb possono riferire la stesso page-frame rispettivamente con l'indirizzo aaa e bbb. Se la pagina contiene al suo interno un indirizzamento un indirizzamento virtuale assoluto che riferisce la stessa pagina questo deve essere allo stesso tempo sia aaa che bbb, con conseguente contraddizione. Ne segue che se una pagina é condivisa, e contiene indirizzamenti assuluti, deve avere necessariamente lo stesso indirizzo virtuale per tutti i processi.*

4.1.2 Descrittori di processo nel processore PC

Nel processore PC, un descrittore di processo é individuato da un identificatore ID

- rappresentato su 16 bit
- multiplo di 8 (0xXXX8)
- costituisce l'identificatore del processo
  - ID viene utilizzato come indice della tabella GDT (Global Descriptor Table) e seleziona un'entrata di 64 bit.
    * GDT[ID]. Tale entrata é di 64 bit e contiene:
      · *base* l'indirizzo virtuale inisiale (base di 32 bit)
      · *limite* spiazzamento relativo alla base dell'ultimo byte (20 bit).

      del descrittore del processo.
  - *base , limite* della GDT sono contenuti nel registro **GDTR**
  - Nel processore PC il registro **TR** (Task Register)
    * la parte visibile contiene l'identificatore ID del processore attualmente in esecuzione.
    * la parte nascosta contiene la base e limite del descrittore di tale processo.

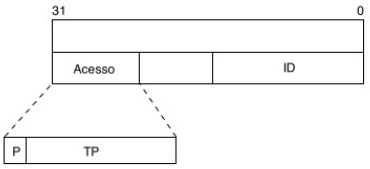
      Il sistema operativo, quando effettua una commutazione di contesto. provvede a caricare in TR l'identificatore del nuovo processo mediante l'istruzione LDTR, che prevede da microgramma a caricare in automatico nella parte nascosta di TR la base e limite del descrittore del nuovo processo prelevandoli dalla tabella GDT.

4.1.3 Commutazione hardware fra processi.

La commutazione fra processi puó essere ottenuta mediante il meccanismo di interruzione. Ogni gate della tabella IDT ha un tipo specificato dal byte di accesso. I tipi sono

- Iterrupt/Trap: giá analizzati nella sezione delle interruzioni.
- **Task**

Un *gate* di tipo *Task* contiene un identificatore di processo. Il discriminante del tipo di gate é dato dal campo TP nel descrittore stesso. Quando si verifica un'interruzione e viene coinvolto un gate di tipo Task (in base al tipo di interruzione) avviene una *commutazione automatica* fra processi. Questo comporta l'annidamento fra processo. In presenza di un annidamento fra processi viene gestito il bit NT del registro EFLAG. Il *processo uscente* (processo attuale) viene sostituito con un nuovo processo o *processo entrante*.



Un descrittore di processo ha la seguente struttura necessariamente statica. Questo é dovuto agli accessi via hardware.

- *link* : contiene un descrittore di processo.
- gli altri campi ordinatamente corrispondo, ordinatamente, a contenuti dei registri del processore.

Passi compiuti dal meccanismo di interruzioni

Se il gate coinvolto é di tipo Task:

- salvataggio della parte visibile di TR (identificatore del processo uscente) in un registro d'appoggio.
- salvataggio dello stato (valore di alcuni registri) del processore nel descrittore del processo uscente.

  Solo alcuni registri sono significativi a caratterizzare lo stato di avanzamento di un processo.
- caricamento nella parte visibile di TR con l'identificatore contenuto nel campo ID del gate, e nella parte nascosta di TR con base e limite del descrittore del processo (prelevati dalla tabella GDT utilizzando il contenuto del campo ID del gate come indice nella tabella GDT).
- caricamento di un nuovo stato del processore a partire dal descrittore del processo entrante, individuato dal contenuto della parte nascosta di TR (tra cui EIP).
- salvataggio dell'identificatore del processo uscente, contenuto nel registro d'appoggio, nel campo link del descrittore del processo entrante.

- settaggio del bit NT (Nested Task) del registro EFLAG.
- microsalto alla microoperazione di chiamata, utilizzando il contenuto del registro EIP.

Il processo entrante diviene cosí attivo a partire da :

- una situazione iniziale (il processo entrante viene messe in esecuzione per la prima volta)
- dalla situazione nella quale era stato precedentemente sospeso (si ricorda che prima di passare alla fase di chiamata di una nuova istruzione, a microprogramma si verifica la presenza o meno di una richiesta di interruzione)

Il corpo di un processo termina con l'istruzione IRET, dal momento che viene invocato con un'iterruzione come una routine di servizio, con la sola differenza che questo ha associato un descrittore di processo. La commutazione inversa si ottiene dunque con l'istruzione IRET in presenza di processi annidati (NT = 1)

- Il processo uscente diviene é quello attivo
- Il processo entrante é puntato dal campo link memorizzato nel descrittore del processo uscente.
- la commutazione inversa é effettuata con passi analoghi ai precedenti, salvo l'azzeramento del bit NT prima di memorizzare lo stato del processore nel descrittore del processo esterno.
- Il meccanismo di interruzione pone a 0 il bit NT se il gate coinvolto é di tipo Interrupt / Trap dopo aver salvato il registro EFLAG (NT vale 1) una parola lunga e EIP .
  - Quest'azione é necessaria nel caso in cui avviene un interruzione di tipo Interrupt / Trap e il processo in esecuzione ha il bit NT (nel registro EFLAG) pari a 1, e la routine di servizio associata a tale interruzione che pur terminando con l'istruzione IRET non provoca una commutazione di contesto.

    ***Nota* 33** *L'istruzione IRET della routine che interrompe un processo per il quale il bit NT vale 1, dopo che é stato azzerato dal meccanismo di interruzione, provvede a ripristinare il valore del registo EFLAG salvato nella pila, ponendo ad 1 il bit NT.*

4.1.4 Processi bloccati

Per evitare la ricorsione dei processi il meccanismo automatico di commutazione gestisce il il bit B (busy) presente in ogni entrata della tabella GDT. Ogni processo attivo ha associato un solo descrittore di processo. Il processo in esecuzione e i processi sospesi da un'interruzione hanno i loro descrittori in una coda. Riattivando un processo, giá attivo:

- nel campo link del suo descrittore viene scritto un nuovo valore, spezzando il collegamento della coda stessa.

- L'esecuzione dell'istruzione IRET prova la scrittura di nuove informazioni nel descrittore del processo uscente, distruggendo cosí lo stato dell'esecuzione precedente (non ancora conclusa) del processo stesso.

Il processo sospeso per ultimo, il cui descrittore é individuato dal campo link del descrittore del processo in esecuzione, puó essere riattivato dall'istruzione IRET (NT =1) .

In conclusione unprocesso per poter essere attivato dal meccanismo di interruzione deve essere marcato come *libero* (bit B = 0) . Il processo in esecuzione (quindi *occupato*) rimane occupato anche quando viene sospeso a causa di una nuova interruzione. Il processo sospeso puó tornare in esecuzione marcato ancora come occupato , supponiamo l'ultimo sospeso, quando il processo in esecuzione esegue l'istruzione IRET. Quest'ultimo viene marcato invece come *libero* quando termina la sua esecuzione eseguendo l'istruzione IRET.

4.2 Protezione

4.2.1 Stati di funzionamento

Il processore puó operare in due stati di funzionamento possibili:

- ***stato sistema***
- ***stato utente***

Per i due stati di operativitá del processo si definiscono altrettanto due *livelli (stati) di privilegio*:

- livello di privilegio **sistema**
- livello di privilegio **utente**
- e rispettano la seguente relazione:
  livello p. sistema > livello p. utente.

Si possono valutare i seguenti aspetti per una distinzione dei due stati

a) Il processore nello

stato sistema puó eseguire tutte le istruzioni (livello di privilegio massimo) anche le istruzioni *privlieggiate.*

stato utente puó eseguire un sottoinsieme di istruzioni (livello di privilegio ridotto) solo le istruzioni *non privileggiate.*

Se nel corpo testo di un programma compare un istruzione privilegiata, e il processore si trova ad evolvere nello stato utente, e si appresta ad eseguirla, genera un ' eccezione di protezione. e.g. sono le istruzioni che operando nei registri speciali del processore.

b) Ogni descrittore di pagina specifica, nel suo byte di accesso, il livello di privilegio per quella pagina. Il processore puó accedere alla pagina per un'operazione di :

- *execute* : prelevare istruzioni.
  - Un accesso per prelevare istruzioni é lecito solo se il processore ha lo stesso livello di privilegio della pagina.

- *read*: leggere operandi,
- *write*: scrivere operandi.
  - Un accesso per trattare operandi (lettura / scrittura) é lecito solo se il processore ha un livello di privilegio maggiore o uguale al quello della pagina.

***Nota* 34** *Su una pagina possono essere consentiti operazione di lettura o anche scrittura, e questi si mantengono indipendentemente dallo stato di privilegio del processore. E se queste* regole di protezione *vengono violate, il processore genera un'eccezione di protezione.*

c) Per i due stati di operativitá del processore, livello sistema e livello utente, é previsto l'utilizzo di pile differenti.

- **pila sistema.**
- **pila utente.**

***Nota* 35**

- *Nell'ipotesi di disporre di un'unica posta pila a livello sistema, e secondo quanto detto riguardo le regole di protezione, tale pila non é accessibile al processore da stato utente (livello delle pagine contente la pila é maggiore del livello utente).*
- *Nell'ipotesi di disporre della stessa pila al punto precedente, ma posta a livello utente, si hanno problemi di protezioni per le informazioni relative allo stato sistema.*

Le istruzioni che operano sulla pila (push e pop) richiedono che il livello di privilegio del processore sia uguale a quello della pila. Ogni processo ha un livello di privilegio, memorizzato nel relativo descrittore. Tale livello viene trasferito (come vedremo pú avanti) in un registro del processore che stabilisce il livello di privilegio del processore. La modifica del livello di privilegio del processore puó essere effettuata solo da istruzioni privilegiate. Questo ci suggerisce che si possono avere processi utente e processi sistema. **In conclusione ogni processo utente puó avere pagine private sia a livello utente, che a livello sistema.**

Il livello di privilegio del processore puó essere modificato in modo controllato, mediante l'ausilio del meccanismo de interruzione.

- Un processo utente puó portarsi volontariamente a livello sistema, eseguendo un istruzione di interruzione software, mandando in esecuzione una routine del sistema operativo che fornisce servizi che l'utente non puó effettuare autonomamente.
- Il medesimo processo puó essere interrotto da un interruzione hardware, e mandare in esecuzione una routine di servizio che per compiere determinate operazioni necessita di essere eseguita con livello di privilegio sistema.

4.2.2 Protezione nel processore PC

Io stato di privilegio del processore é dato dal valore del registro speciale del processore **CPL** (Current Privilege Level) :

- CPL = 1 ⇒ stato sistema.
- CPL = 0 ⇒ stato utente.

Avevamo visto che nel byte di accesso del descrittore di una pagina il bit:

- S/U specifica il livello di privilegio della pagina. (S/U = 1 System , S/U=0 User)
- R/W specifica se la pagina é leggibile o anche scrivibile.

***Nota* 36** *Il direttorio e le tabelle delle pagine, la tabella GDT e i descrittori dei processi sono in pagine che hanno livello di privilegio sistema.*

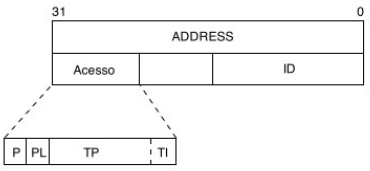
Le istruzioni di I/O possono essere privilegiate, e questo viene stabilito scrivendo un opportuno valore nel campo IOPL (I/O Privilege Level) del registro EFLAG. Tale modifica é applicabile se il processore é nello stato sistema.

L'istruzione STI e CLI

Le istruzioni di abilitazione e disabilitazione delle interruzione esterne mascherabile sono privilegiato allo stesso livello di quelle di I/O.

4.2.3 Meccanismo di interruzione e protezione

Il registro **IDTR** contiene base e limite della tabella IDT. Un gate della tabella IDT ha la seguente struttura, piú dettagliata rispetto a quella analizzata in precedenza.

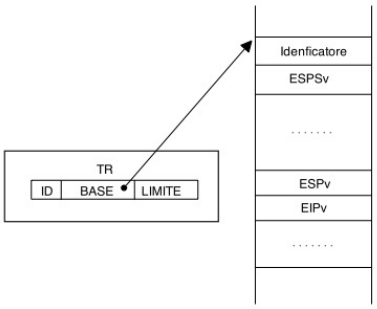


Il tipo del gate (campo TP) puøessere :

- Interrupt / Trap (bit IT) : in tal caso il gate specifica l'indirizzo virtuale della routine di interruzione.
- Task: il bit fa parte del tipo, in quel caso il gate specifica l' identificatore del nuovo processo.

Un descrittore di processo contiene in posizioni prefissate i valori dei registri del processore virtuale. Dopo la parola lunga (campo *link*) segue il valore di ESPSv, puntatore della pila del livello sistema del processo stesso. Il puntatore della pila utente é dato dal valore del registro ESPv del processore se tale processo é in esecuzione. Sappiamo anche che il registro TR del processore contiene, nella parte nascosta, *base* e*limite* del processo attualmente in esecuzione, ne segue che il valore di **ESPSv** é accessibile dal meccanismo hardware di interruzione.

Alla luce di quanto esposto riguardo i livelli di privilegio analizzati, bisogna rivalutare le azioni che il meccanismo di interruzione deve effettuare. Visto che in gioco ci sono i seguenti fattori:



- Interruzioni nel senso classico : interruzione di tipo Interrupt/ Trap (gestione dei bit P TF, IF) mandare in esecuzione una routine di sistema operativo, quindi tenere traccia dell'indirizzo di ritorno.
- Commutazione hardware fra processo. (gestione del bit NT e B)
- Possiblitá di un cambio di privilegio (controllato) del processore. (gestione dei bit S/U CPL PL)

Quando si verifica un'interruzione, via hardware viene esaminata l'entrata specificata dal tipo dell'interruzione come segue:

- esame del bit P (*Present*) : se vale 0 viene generata un eccezione per tipo di interruzione non implemtento.
- esame del bit PL : devono sussistere le seguenti relazioni altrimenti viene generata un' eccezione di protezione.
  - per interruzione software e eccezioni PL $\leq$ CPL
  - interruzione hardware PL puó essere qualsivoglia.
-

Per un gate di tipo Interrupt / Trap:

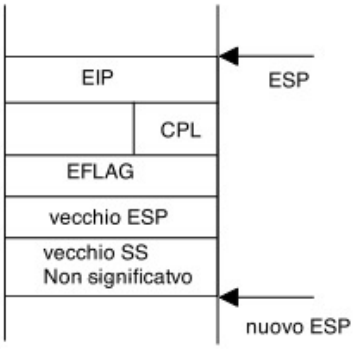
- Prelievo dell'indirizzo virtuale della routine, sia VA, ed esame del bit S/U del descrittore di pagina corrispondente a tale indirizzo.
  - se **S/U = CPL** (nessun cambiamento di stato) :
    1. Memorizzazione in pila dei valori attuali dei registri EFALG, CPL, EIP.
    2. Caricamento di CPL con il valore S/U (operazione inutile) e di EIP con VA.
    3. Azzeramento del flag TF se il gate é di tipo Trap, oppure TF e IF se é di tipo Interrupt.
    4. Azzeramento del flag . NT.
    5. Microsalto alla microoperazione della fase di fach.
  - se **S/U > CPL** (passaggio dallo stato utente allo stato sistema)
    1. Salvataggio momentaneo del valore di ESP in un registro d'appoggio.

Il valore di NT é salvato nel registro EFLAG, e si azzera per evitare che quando la routine di servizio esegua l'istruzione IRET si abbia una commutazione automatica fra processi, azione per altro insensata visto che la routine di servizio non ha un descittore

2. Caricamento in ESP il valore di ESPSv. prelevato dal descrittore del processo mediante il registro TR.
3. Memorizzazione nella nuova pila di una prima parola lunga (non significativa SS) e del valore di ESP momentaneamente salvato.
4. Memorizzazione nella nuova pila dei valore attuali dei registri EFLAG, CPL, EIP (indirizzo di ritorno) .
5. caricamento di CPL con il valore S/U e di EIP con VA.
6. Azzeramento del flag TF se il gate é di tipo Trap, oppure TF e IF se é di tipo Interrupt.
7. Azzeramento del flag NT.
8. Microsalto alla microoperazione della fase di fach.

– Se **S/U < CPL** viene generata una eccezione per tentativo di cambiamento di stato non lecito. In altre parole si tenta di abbassare il livello di privilegio in modo non controllato.

Quando avviene una cambiamento di livello, secondo quanto esposto, la nuova pila (pila sistema) assume la seguente forma.



L'istruzione IRET e protezione

Nota 37 *L'istruzione IRET, in assenza di processi sistema, non puó che essere eseguita dalle routine del sistema operativo che possono essere invocati mediante un numero d'ordine dai programmi utente, o eseguito in risposta ad interruzione hardware, o eccezioni.*

Nell'ipotesi che non ci siano processi annidati (NT = 0), e notando che la sommitá della pila, che sia avvenuto o meno un cambio di livello di privilegio, é composta da EIP CS EFLAG, l'istruzione IRET effettua le seguenti azioni:

- Estrazione dalla pila attuale dei vecchi valori di EIP, CPL, EFLAG, indichiamoli per comoditá con EIPv, CPLv, EFLAGv
- se CPLv = CPL :

  - trasferimento dei vecchi valori EIPv, CPLv, EFLAGv nei corrispettivi registri. (operazione superflua)

- altrimenti CPLv < CPL
  - trasferimento dei vecchi valori EIPv, CPLv, EFLAGv nei corrispettivi registri, ripristinando del livello utente.
  - estrazione dalla pila attuale di due doppie parole (ESPv, SSv) e trasferimento delle stesse nei corrispettivi registri, ripristinando cosí la pila utente.

Una routine che va in esecuzione con il meccanismo di interruzione. per poter essere eseguita, deve trovarsi in pagine che hanno lo stessa livello di privilegio uguale a quello in cui si porta il processore. In conclusione un'interruzione (hardware o software), che comporta il trasferimento di una nuova quantitá nel registro CPL, puó provocare il passaggio dallo stato utente allo stato sistema. Il viceversa non é ammesso, se non con l'esecuzione della IRET da parte della routine di sistema. Con il principio di protezione non é congruente ipotizzare che il sistema operativo possa invocare routine utente. Questo a priori non é ammesso dal meccanismo di interruzioni, ovvero l'abbassamento di livello di privileggio. *Un processo utente*

- non puó portarsi ad un livello di privilegio superiore se non invocando una routine di sistema mediante il suo numero d'ordine.
- puó essere interrotto da cause esterne, con conseguente modifica del livello di privilegio stesso.

  ***Nota* 38** *La routine che va in esecuzione effettuando un cambio di privilegio, comporta un cambio di pila, da pila utente a pila sistema. Questo é sempre vero anche se l'interruzioni esterne o eccezioni non interessano il processo utente in esecuzione. Presta la sua pila sistema.*

- la tabella delle interruzioni deve essere posta a livello sistema.

4.2.4 Il problema del cavallo di Troia

Un processo utente, invocando una primitiva che lavora a livello sistema, trasmettendogli dei parametri, se questa accetta parametri. Se uno o piú parametri sono indirizzi l'utente malintenzionato, puó trasmettere degli indirizzi che si riferiscono a pagine con livello di privilegio sistema. In stato utente non é possibile accedere a tale pagine.

La possibile soluzione, é la primitiva che viene invocata controlla preventivamente il campo CPL memorizzato nella pila e lo confronta il livello esistente al momento della sua invocazione e, accedendo alla tabella delle pagine, il livello di privilegio della pagina a cui l'indirizzo trasmesso si riferisce, in tal caso puó verificare la consistenza o meno.

Una soluzione pratica risulta data da un convenzione sugli indirizzi virtuali. Se l'indirizzo virtuale inizia con 0 si ha un livello di privilegio utente altrimenti sistema. In tal caso le primitive non devono accedere alle tabelle delle pagine per esaminare il livello di privilegio della pagina corrispondente ad un certo indirizzo virtuale in quanto questo é implicito.

4.3 Sistemi Multiprogrammati

Analizziamo piú da vicino la realizzazione di un sistema multiprogrammato (*multitasking*) . Come modello di riferimento prendiamo un elaboratore basato:

- sul processore PC.
- su piú processori virtuali.
- un unico processore reale.
- su memoria virtuale paginata.
- su due livelli di privilegio, utente e sistema

4.3.1 Modello di un processo

Ogni processo ha un proprio livello di privilegio e in base a questo si hanno

- *processi sistema*
- *processi utente*

Attraverso il meccanismo di interruzione un processo utente puó innalzare il proprio livello di privilegio a livello sistema per poi tornare al livello di principio eseguendo l'istruzione IRET. Ogni processo ha un proprio spazio di indirizzamento suddiviso in

- spazio comune a tutti i processi,
- spazio privato proprio di ogni processo.

I vari spazi di indirizzamento sono ulteriormente suddivisi come segue:

- Per i processi utente lo spazio comune é costituito
  - da una parte a livello utente,
  - da una parte a livello sistema.
- Per i processi sistema sia lo spazio comune che privato sono a livello sistema.
- Nello spazio comune a tutti i processi a livello sistema sono posizionati le routine del sistema operativo, necessarie per la gestione della multiprogrammazione .

***definizione* 7** *Un processo é caratterizzato dalle seguenti entitá*

- *il descrittore. Si trova nello spazio comune a livello sistema, e contiene*
  - *il puntato di pila per il livello di sistema (ESPSv) se é un processo utente.*
  - *il contenuto dei registri generali del processore, fra cui il puntatore di pila per il proprio livello.*
  - *il contenuto del registro CPL e del registro CR3.*
- *il corpo (programma che ne definisce le elaborazioni) costituito da un codice e da un' area dati privata. L'area dati contiene*

  - *le due pile per i due livelli di privilegio per un processo utente.*
  - *pila sistema per un processo sistema.*

- *un area dati comune. Tale zona puó essere utilizzata dai processi per*
  - *accedere a informazioni comuni,*
  - *a scambiare informazioni fra processi visto che non si hanno altri meccanismi per effettuare ció.*

***definizione* 8** *Il codice di un programma o di una routine detto rientrante se progettato in modo che una singola copia del codice in memoria possa essere condivisa (quindi risiede nella parte comune) ed eseguita contemporaneamente da piú processi separati, con lo stesso livello di privilegio. Affinch una routine o comunque una parte di codice sia rientrante deve soddisfare questi requisiti:*

1. *Nessuna porzione del codice possa essere alterata durante l'esecuzione (codice non automodificante);*
2. *Il codice non deve richiamare nessuna routine che non sia a sua volta rientrante*
3. *Il codice deve usare, se necessarie, solo variabili temporanee allocate sullo stack.*
4. *Il codice non deve modificare n variabili globali n aree di memoria condivisa.*

Se una data porzione di codice non rispetta queste regole, non possibile farla eseguire da piú processi contemporaneamente ma necessario regolarne l'accesso tramite semafori o sezioni critiche, per assicurarsi che venga eseguita da un solo processo alla volta. La parte di codice del nucleo che implementa la sincronizzazione interprocesso (semafori, sezioni critiche ecc.) non rientrante per definizione.

Per implementare un processo si puó utilizzare come modello comune a tutti i processi una funzione C++.la funzione , sia funz (), costituisce il *corpo* comune a piú processi. Ricordiamo l'implementazione di una funzione C++ prevede per ogni istanza

- una zona testo, dove viene memorizzato il codice.
- una pila dove destinata a contenere le variabili automatiche (variabili locali) e i parametri attuali,
- una zona dati destinata a contenere le variabili statiche.

Ne segue che piú processi allo stesso livello possono avere come *corpo* istanze della stessa funzione:

- la zona testo e la zona dati memorizzati nello spazio comune a tutti i processi
- la pila (o pile) memorizzate nello spazio privato di ciascun processo.

Un processo puó iniziare o terminare.

- Quando inizia vengono creati il suo descrittore e la pila o pile a seconda che sia un processo sistema o processo utente , il suo codice puó venir creato oppure se condiviso pu'øgiá esistere.
- Quando termina vengono distrutti codice e pila / pile. Il codice viene distrutto sono se non é condiviso.

4.3.2 Commutazione di contesto

La commutazione di contesto é l'insieme di azioni che vengono svolti dalle routine del sistema operativo dedicate se viene realizzata vai software , mandando in esecuzione un nuovo processo, e venga sospeso quello in esecuzione al verificarsi di qualche evento. Se l'evento si manifesta per via di un interruzione (hardware o software) .
Facciamo le seguenti ipotesi

La commutazione hardware fra processi, viene gestita appunto via hardware.

1. La commutazione di contesto fra processi é gestita dalle routine del sistema operativo

   ***Nota* 39** *La commutazione avviene sempre ed esclusivamente a livello sistema.*

2. Il gate coinvolto sia di tipo Interrupt.
3. Le precedenti ipotesi sono valide fintanto che non verrá specificato diversamente.

La commutazione, che avviene a livello sistema, comporta le seguenti macro azioni:

1. Salvataggio dello stato attuale del processo in esecuzione (*processo uscente*).
   - salvare nel descrittore del processo le informazioni di stato, fra cui il contenuto dei registri del processore, che interessano il processo stesso.
   - inserire l'identificatore del processo, in una coda in base a determinate condizioni. Le possibili code sono
     - la coda dei *processi pronti*
     - la coda dei *processi bloccati*
     - la coda dei *processi in esecuzione*, logicamente é costituita da un solo elemento.

   l meccanismo delle interruzioni memorizza in pila il contenuto dei registri EFLAG, CPL, EIP, quindi non richiedono spazio nel descrittore del processo.
2. La scelta del nuovo processo da mandare in esecuzione (*processo entrante*).
   - La scelta del nuovo processo entrante é compito della routine detta *schedulatore*. Tale routine opera sulla coda dei *processi pronti*, ed effettua la scelta del processo da mandare in esecuzione basandosi su informazioni di prioritá.
3. Caricamento dello stato relativo al processo entrante.
   - caricare nel registro TR il valore dell'identificatore del processo entrante, prelevato dalla coda dei *processi pronti*.

- caricare il valore dei registri del processore con i valori dei registri virtuali contenti nel descrittore del processo individuato dalla base e limite della parte nascosta del registro TR.

***Nota* 40**

- *Una commutazione di contesto produce il caricamento di un nuovo valore di CR3, e quindi il cambiamento dello spazio fisico di memoria.*
- *Il nuovo valore di ESP (= ESPSv) , si riferisce alla pila sistema realizzata nella parte privata del processo entrante.*
- *Il nuovo valore di EIP (contatore di programma) si riferisce al punto di esecuzione nel corpo posto nello spazio comune.*

In conclusione un processo puó trovarsi in ogni istante in uno dei seguenti stati:

- nello stato di *esecuzione*
- nello stato *pronto* : in attesa di essere mandato in esecuzione.
- nello stato *bloccato*: in attesa del verificarsi di un certo evento di attivazione. Quando diviene attivo il suo identificatore viene rimosso dalla coda dei processi bloccati e inserito nella coda dei processi pronti.

4.3.3 Nucleo e interruzioni

Il *nucleo* di un sistema operativo multiprogrammato é l'agglomerato di un insieme di routine e strutture dati in grado di gestire la multiprogrammazione. Come anticipato questo risiede nella parte comune a livello di privilegio sistema.

- SISTEMA PERSONALE (monoutente ma multitasking): i corpi dei processi sono costituiti dai vari programmi attivati dall'utente, e quello in esecuzione ha il controllo della tastiera e video. Lo schedulatore invocato esplicitamente dall'utente, seleziona il nuovo processo.
- SISTEMA TIME-SHARING: molti utenti condividono lo stesso elaboratore in maniera interattiva. mediante un terminale o una richiesta di servizio. I corpi dei processi sono costituiti dai programmi dei vari utenti. L'elaboratore esegui il primo processo pronto. Nel caso in cui piú processi sono pronti, per mezzo di un contatore di tempo lo schedulatore va in esecuzione a intervalli regolari (sfruttando il meccanismo di interruzione), e assegna l'elaboratore a un utente alla volta in modo ciclico.
- SISTEMA TEMPO REALE L'esecuzione dei processi avviene in seguito al verificarsi di certi eventi esterni, e deve rispettare dei vincoli temporali. Alcuni processi sono piú importanti di altri, e lo schedulatore sceglie il processo a precedenza piú alta. Nel caso in cui un processo diviene pronto a partire dallo stato bloccato, puó accadere che la precedenza del processo in esecuzione sia inferiore rispetto al processo sbloccato, con conseguente commutazione forzata di contesto (prelazione).

Faremo esplicito riferimento ai sistemi tempo reale.

Le routine del nucleo si dividono in:

- *primitive* invocate da un processo utente con interruzione software, con innalzamento di livello di privilegio.
- *driver* mandati in esecuzione da un'interruzione esterna, che operano in stato sistema.

* Alcuni di queste routine possono determinare una commutazione di contesto. *

Sezioni critiche

Le strutture dati, posti a livello sistema, piú rilevanti sono le code dei processi:

- sono costituiti da *elementi* che sono identificatori di processo.
- ogni coda é individuata da un *puntatore*

Molte routine del nucleo lavorano sulle code dei processi. Le code sono risorse *astratte*, e si trovano in uno stato consistente sono all'inizio e alla fine dell'operazione su di esse eseguita. Le interruzione esterne sono per definizione asincrone rispetto all'esecuzione di una qualsiasi programma. Puó accadere che vada in esecuzione una routine che opera sulle code dei processi mentre sono in uno stato non consistente. Si presenta la necessitá che le operazioni sulle code dei processi avvenga in modo *indivisibile*, pertanto le routine di cui sopra, vengono eseguite con le interruzioni esterne mascherabili *disabilitate*. Le routine del nucleo in genere sono considerate delle sezioni critiche corte e vengono eseguite con le interruzione esterne mascherabili disabilitate.

***Nota* 41** *Il meccanismo di interruzione (hardware o software) utilizzato per mandare in esecuzione una routine del nucleo fa uso di gate di tipo interrupt, e questo fá si che le interruzioni esterne mascherabili vengano automaticamente disabilitate.*

4.4 Preccessi e interruzioni

Un processo in esecuzione (ha il controllo del processo reale) puøprodurre un cambiamento di contesto:

- *volontario* : mediante l'invocazione di una primitiva, effettua delle richieste che non possono essere soddisfatte. In tal caso il processo diviene bloccato e viene chiamato lo schedulatore.
- *involontario*: si verifica un'interruzione esterna. Viene mandato in esecuzione un driver che puó :
  - forzare l'esecuzione di un altro processo.
  - far divenire pronto un processo bloccato.
  - chiamare lo schedulatore.
  - effettuare le sue elaborazioni e terminare (in tal caso il processo interrotto torna in esecuzione) .

Per poter realizzare un sistema real-time occorre che tutti i processi, quando non eseguono routine del nucleo, vengano eseguiti con le interruzioni esterne mascherabili abilitate.

4.4.1 Realizzazione delle primitive

Per strutturare un nucleo si fará riferimento ai seguenti moduli, scritti in assembler e C++:

- due moduli *utente.s utente.cpp* con livello di privilegio utente.
- due moduli *sistema.s sistema.cpp* con livello di privilegio sistema.

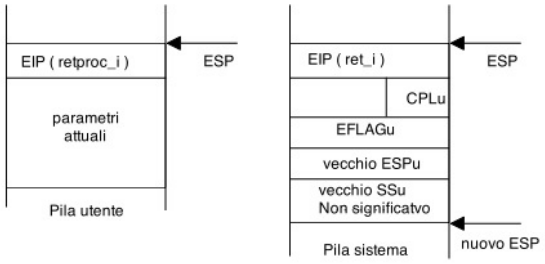
Un processo puó invocare delle primitive di nucleo, che si trovano a livello sistema. I linguaggi ad alto livello non permettono di invocare direttamente le primitive (routine)

- non esistono dei costrutti che permetto di generare interruzioni software.
- Si ovvia con sottoprogramma di interfaccia scritto in assembler, il quale invoca la primitiva vera e propria.

La primitiva accede al record di attivazione per il prelievo dei parametri. Nel caso di un processo utente il record di attivazione é situato nella pila utente. La a_primitiva termina con un'istruzione di ritorno da interruzione, e se non si sono avute commutazione di contesto, restituisce il controllo al sottoprogramma di interfaccia che a sua volta ritorna al programma chiamante.

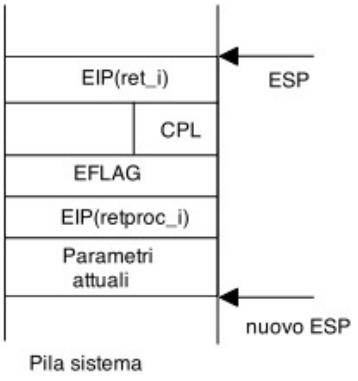
```
// utente.cpp
...
extern "C" void primitiva_i  (/* parametri formali */) ;

// corpo del processo
void proc_body (/*parametri formali*/)
{
...
primitiva_i (/*parametri attuali*/);
// retproc_i
...
}

#// utente.s
...
.TEXT
.GLOBAL _primitiva_i
_primitiva_i:
       INT $tipo_i # sottogramma di interfaccia
ret_i:        RET

#// sistema.s
...
.TEXT
...
a_primitiva_i:
rout_$tipo_i:
        ...
        IRET
...
```

Quando un processo utente chiama una routine di interfaccia, memorizza in pila utente i parametri attuali, e e con l'istruzione di chiamata (che si traduce in una call) il valore di EIP relativo al punti di ritorno (retproc_i) . La routine di interfaccia _primitiva_i contiene l'istruzione *INT $tipo_i*, che invoca la routine vera e propria, provocando un variazione di livello. Il meccanismo di interruzione prevede un cambio di pila, il cui puntatore viene prelevato dal descrittore del processo in esecuzione. Come prevede ancora una volta il meccanismo di interruzione, viene copiato nella nuova pila il valore



dei registri (parola non significati va SSu), ESPu, EFLAGu, CPLu, EIPu (ret_i).

L'invocazione di una primitiva da parte di un processo sistema non produce nessun cambiamento di livelli di privilegio. Nel caso in



cui una a_primitiva possa provocare una commutazione di contesto, deve

- salvare preventivamente lo stato del processo al momento della sua invocazione.
- caricare lo stato del nuovo processo prima di terminare.
  - il nuovo stato puó coincidere con quello salvato.

Dopo il caricamento dello stato la IRET agisce sulla pila sistema del nuovo processo. Come avevamo accennato tutti i processi hanno la sommitá della pila nello stessa configurazione (tre parole lunghe) in quanto hanno abbandonato l'esecuzione

- per effetto di un'interruzione software (invocazione di una a_primitiva),
- per effetto di un'interruzione hardware.

All'atto della creazione di un processo bisogna predisporre tale configurazione (EIP, CPL, EFLAG). L'istruzione IRET provoca il ritorno

- ad un nuovo processo se il processo é stato creato ex-novo, e posto nella coda dei processi pronti.

- alla routine dell'interfaccia del nuovo processo che precedentemente aveva invocato una a_primitiva che ha provocato una commutazione di contesto.

  ***Nota* 42** *Quando un processo che ha invocato una a_primitiva ritorna in esecuzione, riparte dal punto successivo rispetto all'invocazione della a_primitiva stessa (rispetto a INT $tipo)*

4.4.2 Realizzazione dei processi

Il *descrittore di un processo (des_proc)* é una struttura si fatta:

```
// sistema.cpp
struct des_proc
{
        short id; // id su 8 bit
        int punt_nucleo; // puntatore alla pila del nuclo
        int contesto[N_REG]; // conteunto dei registri del processore
        int precdenza;
        char cpl;
        int cr3;
};
```

I descrittori di processo sono variabili definiti nel nucleo, ed individuato mediante un identificatore che tramite la tabella GDT determina la base e il limite. Un processo é costituito da un descrittore da un corpo (codice e pila) .

***Nota* 43** *Nella pila relativa al privilegio del processo si trovano i parametri attuali del corpo del processo.*

I processi vengono gestite mediante le strutture dati code. Ogni coda é individuata da un puntatore, e i sui elementi contengono le seguenti informazioni

- l'identificatore del processo
- la sua precedenza

```
// sistema.cpp
...
struct proce_elem { short identifier, int priority; proc_elem* puntatore;}
struct proce_elem* esecuzione;
struct proce_elem* pronti;
```

```
# i puntatore vengon utilizzati anche a livello assembler
#sistema.s
...
.DATA
...
.GLOBAL _esecuzione
_esecuzione:  .LONG
.GLOBAL _pronti
```

A livello di nucleo sono definiti i puntatori

- *esecuzione*: contiene l'indirizzo di un elemento di tipo proc_elem
- *pronti*: contiene il puntatore alla coda dei processi pronti.
- e bloccati? perche non é previsto un puntatore per la coda dei processi bloccati? Un processo si puó bloccare su una risorsa se questa non é disponibile. in tal caso il processo si blocca su una coda associata alla risorsa stessa. Quindi non é necessaria una coda generica dei processi bloccati.

Ogni *processo* deve essere creato (attivato) :

- selezione di un identificatore.
- inizializzazione del corrispettivo descrittore.
- allocazione nello heap di un tipo proc_elem, in cui vengono memorizzati l'identificatore, e la sua precedenza.
- inserimento nella coda dei processi pronti.

Creazione dei processi

La creazione di un processo avviene

- in fase di inizializzazione da un processo principale main (), che supponiamo venga eseguito a livello utente (questa é la strada che viene presa in seguito).
- un processo padre crea dinamicamente un processo figlio.

Un processo viene creato con la primitiva activate_p

```
void activate_p (void f (int), int par, int prio, char livel, short& id, bool &ris);
```

Richiede

- l'intestazione di una funzione (f (int)), e il parametro attuale della funzione (int par),
- livello di privileggio proprio,
- variabile riferimento per l'identificatore del processo,
- variabile riferimento per il risultato dell'attivazione del processo.

La terminazione di un processo, al momento facciamo quest'ipotesi, avviene per mezzo della primitiva terminate_p (), che non ha paramentri

- distrugge le strutture dati allocate per il processo puntato dal processo esecuzione
- richiama lo schedulatore
- *terminate_p ()* costituisce l'ultima istruzione di ogni processo.

Il programma principale, una volta attivato tutti i processi, esegue la primitiva begin_p () che

- trasforma il programma del processo a piú bassa precedenza
- lo inserisce nella coda dei processi pronti,
- selezionare il processo a piú alta precedenza e caricane lo stato.

```
// utente.cpp
extern "C" void activate_p (void f (int), int par, int prio, char livel, short& id, bool &ris);
extern "C" void terminate_p ();
extern "C" void begin_p ();
...
short processo_i ; // ID del processo
// corpo del processo
void p (int h)
{
// parametro h risiede nel record di attivazione rilativo a livello LIV
...
...
terminate_p ();
}
...
int main ()
{
  bool ris;
  ...
  activate_p (p, A, PRIO, LIV, processo_i, ris);
  ...
  begin_p ();
}

#utente.s
...
.TEXT
...
.GLOBAL _activate_p:
_activate_p:
  INT$tipo_a
  RET
.GLOBAL _terminate_p:
_terminate_p:
  INT$tipo_t
  RET
.GLOBAL _begin_p:
_begin_p:
  INT$tipo_b
  RET

#sistema.s
...
.TEXT
...
a_activate_p:
  #routine INT $tipo_a
  #...
  IRET
a_terminate_p:
  #routine INT $tipo_t
  #...
  IRET
a_begin_p:
  #routine INT $tipo_b
  #...
  IRET
```

Come avevamo sottolineato con una nota, i parametri attuali, in questo caso uno solo, le variabili locali nel corpo del processo fanno parte del record di attivazione del processo, e quindi vanno inseriti nella pila di livello di privilegio proprio del processo stesso.

***Nota* 44** *Gli identificatori dei processo sono variabili condivise, e un processo puó utilizzare primitive che agiscono esplicitamente su un altro processo, ad esempio una terminazione forzata di un altro processo.*

4.4.3 Processo dummy

Nella coda dei processi pronti, viene accodato il processo *dummy*, a precedenza inferiore a quella di tutti i processi utente, ma superiore a

quella del processo main che crea i processi, che ha lo scopo di andare in esecuzione quando tutti i processi utente sono bloccati in attesa del verificarsi di un evento (interruzione esterna). Se i tutti i processi utente sono bloccati, e avviene un'interruzione esterna, la routine che va in esecuzione oltre a sbloccare un processo, deve richiamare lo schedulatore e caricare lo stato del processo selezionato. Nel nucleo viene gestita la variabile *processi* come segue

- viene inizializzata a 0, in fase di init,
- viene incrementata dalla primitiva active_p (),
- viene decrementata dalla primitiva terminate_p ().

Il processo dummy puó terminare solamente se la variabile *processi*, testata per via della primitiva give_num (int&), é uguale a zero. Lo schedulatore manda di nuovo in esecuzione il processo main (), che effettua le operazioni di chiusura.

```
// utente.cpp
...
extern "C" void give_num (int& lav);

void dd (int i)
{
 int lavoro;
 do
 give_num (lavoro);
  while (lavoro != 0);
 terminate_p ();
}
#sistema.s
...
.TEXT
...
a_give_num:
 #routine INT $tipo_g
 # ...
 IRET
```

4.4.4 Corpo delle primitive

Una a_primitiva puó, se puó causare una commutazione di contesto, deve

1. effettuare il salvataggio dello stato attuale,
2. adempire al proprio compito per la richiesta di servizio fatta,
3. caricamento del nuovo stato.

La prima e terza operazione sono effettuate a livello assembler, visto che coinvolgono registri del processore. Il corpo della a_primitiva puó essere invece, scritto in C++ (c_primitiva). Devono essere rispettate le regole di aggancio dei sottoprogrammi ad alto livello, ovvero vanno ricopiati in testa alla pila sistema i parametri che per i processi utente si trovano nella pila utente.

```
# sistema.s
.DATA
.balign 16
gdt:
        // spazio per 5 descrittori piu' i descrittori di TSS
        // i descrittori verrano costruiti a tempo di esecuzione
        .space 8 * 8192, 0

.TEXT
...
// Estrae la base del segmento da un descrittore.
```

```
// Si aspetta l'indirizzo del descrittore in %eax,
// lascia la base del segmento in %ebx
// NOTA: il formato dei descrittori di segmento dei
//       processori Intel x86, per motivi di compatibilita'
//       con i processori Intel 286 (che erano a 16 bit),
//       e' piu' complicato di quello visto a lezione.
//       In particolare, i byte che compongono il campo base
//       non sono consecutivi
.macro estrai_base

        movb 7(%eax), %bh        // bit 31:24 della base in %bh
        movb 4(%eax), %bl        // bit 23:16 della base in %bl
        shll $16, %ebx                // bit 31:16 nella parte alta di %ebx
        movw 2(%eax), %bx        // bit 15:0 nella parte basse di %ebx

.endm

// offset, all'interno della struttura des_proc, dei campi
// destinati a contere i registri del processore
.set EAX, 40
.set ECX, 44
.set EDX, 48
.set EBX, 52
.set ESP, 56
.set EBP, 60
.set ESI, 64
.set EDI, 68
.set ES, 72
.set SS, 80
.set DS, 84
.set FS, 88
.set GS, 92

.set CR3, 28

.set FPU, 104

// salva lo stato del processo corrente nel suo descrittore
//
salva_stato:
                pushl %ebx
        pushl %eax

        movl _esecuzione, %eax
        movl $0, %ebx
                movw (%eax), %bx                // esecuzione->identifier in ebx
                leal gdt(, %ebx, 8), %eax       // ind. entrata della gdt relativa in eax
        estrai_base                         // ind. TSS -> %ebx

                popl %eax

        movl %eax, EAX(%ebx)                // salvataggio dei registri
        movl %ecx, ECX(%ebx)
        movl %edx, EDX(%ebx)
        popl %eax                          // vecchio valore di %ebx in %eax
        movl %eax, EBX(%ebx)
        movl %esp, %eax
        addl $4, %eax                       // salviamo ind. rit. di salva_stato...
        movl %eax, ESP(%ebx)                // ... prima di memorizzare %esp
        movl %ebp, EBP(%ebx)
        movl %esi, ESI(%ebx)
        movl %edi, EDI(%ebx)
        movw %es, ES(%ebx)
        movw %ss, SS(%ebx)
        movw %ds, DS(%ebx)
        movw %fs, FS(%ebx)
        movw %gs, GS(%ebx)

        movw $SEL_DATI_SISTEMA, %ax               // selettori usati dal nucleo
        movw %ax, %ds
        movw %ax, %es
        // ss contiene gia' il valore corretto
        movw $0, %ax
        movw %ax, %fs
        movw %ax, %gs
```

```
        fsave FPU(%ebx)

        ret

// carica lo stato del processo in _esecuzione
//
carica_stato:
        movl _esecuzione, %edx
         movl $0, %ebx
        movw (%edx), %bx                // esecuzione->identifier in ebx

              leal gdt(, %ebx, 8), %eax       // ind. entrata della gdt relativa in eax
        estrai_base                        // ind. del TSS in %ebx
              andl $0xfffffdff, 4(%eax)       // bit busy del TSS a zero

               ltr %cx                                 // nuovo valore in TR

                frstor FPU(%ebx)

        movw GS(%ebx), %ax                // ripristino dei registri
        movw %ax, %gs
        movw FS(%ebx), %ax
        movw %ax, %fs
        movw DS(%ebx), %ax
        movw %ax, %ds
        movw SS(%ebx), %ax
        movw %ax, %ss
        movw ES(%ebx), %ax
        movw %ax, %es

        popl %ecx                       // toglie dalla pila l' ind. di ritorno

        movl CR3(%ebx), %eax               // cambio di direttorio
        movl %eax, %cr3                       // NOTA: siamo sicuri della continuita'
                                        // dell'indirizzamento, in quanto il sistema
                                        // e' mappato agli stessi indirizzi in tutti
                                        // gli spazi di memoria

        movl ESP(%ebx), %esp            // nuovo punt. di pila...
        pushl %ecx                        // salvataggio ind. di ritorno nella nuova pila

        movl ECX(%ebx), %ecx
        movl EDI(%ebx), %edi
        movl ESI(%ebx), %esi
        movl EBP(%ebx), %ebp
        movl EDX(%ebx), %edx
        movl EAX(%ebx), %eax
        movl EBX(%ebx), %ebx

        ret

.EXTERN _c_primitiva_i
a_primitiva:
        #routine INT $tipo_i
        CALL  salva_stato
        #ricopiamento dei paramenti
        CALL  _c_primitiva_i
        #ripulitura della pila
        CALL  carica_stato
        IRET
##

// sistema.cpp
extern "C" void c_primitiva_i (/*parametri formali*/)
{
/*implementazione del corpo della primitiva_i*/
}
```

Ricopiamento Dati

Nella realizzazione a_primitiva, il ricopiamento dei parametri, richiesto dallo standard di aggancio Assembler C++, avviene dalla pila utente o pila sistema in base al vecchio valore di CPL (memorizzato

in pila sistema) . Se l'accesso interessa una pila utente da livello sistema, non comporta nessuna violazione delle regole di protezione, in quanto a livello sistema la pila utente é contenuta in pagine dati, il cui livello di privilegio é minore al livello di privilegio del processore.

***Nota* 45** *La ripulitura della pila dopo la chiamata al sottoprogramma _c_primitiva non é necessaria, a causa del caricamento di uno nuovo stato (precedentemente salvato), e con esso si ha in caricamento di un nuovo valore di ESP, che si riferisce ad un livello di riempimento della nuova pila cosí come prevede il meccanismo di interruzioni.*

Slava / ripristina stato

- **salva_stato** tale routine salva il contenuto dei registri del processore del descrittore del processore in cui identificatore é contenuto nell'elemento puntato da *esecuzione*.

  # il valore di EFLAG, CPL, EIP si trovano nella pila sistema del processo che ha invocato la a_primitiva e non si é previsto uno spazio apposito nel descrittore del processo.

  # il salvataggio del valore CR3 e ESP producono il salvataggio dell'intera pila.

  # ***Nota* 46** *Il valore di ESP che deve essere salvato, non corrisponde al valore attuale di ESP ma va opportunamente incrementato, poiché in cima alla pila é memorizzato l'indirizzo di ritorno del sottoprogramma* salva_stato.

- **cairca_stato**. Una c_primitiva deve lasciare in esecuzione l'indirizzo dell'elemento contenente l'identificatore del processo da mandare in esecuzione. e la carica_stato mediante *esecuzione*

  # copia in TR il calore dell'identificatore del processo.

  # caricare i registri del processore, con i valore dei registri virtuali contenuti nel descrittore del processo, con l'ausilio dell'identificatore del processo, e la tabella GDT.Quest'operazione si effettua sia nel caso di commutazione contesto che non.

  ***Nota* 47** *Nel caso di commutazione di contesto, la carica_stato deve effettuare il trasferimento dell'indirizzo, per poter terminare correttamente dalla vecchia pila sistema alla nuova pila sistema.*

Nel caso non avviene mai una commutazione di contesto, nella a_primitiva, sia l i-ma, il salvataggio e il ripristino dello stato del processo, puó essere sostituito con il salvataggio e ripristino dei registri utilizzati.

```
# sistema.s
...
.TEXT
...
.EXTERN _c_primitiva_i
a_primitiva_i:
 #routine INT $tipo_i
 #salvataggio dei registri
 #ricopiamento dei parametri
```

```
CALL _c_primitiva_i
#ripulitura della pila
#ripristino dei registri
IRET

// sisitema.cpp
...
extern "C" void c_primitiva_i (/*parametri formali*/)
{
 /**/
}
```

Per realizzare una primitiva che opera sulle code dei processi si fa uso delle seguenti routine:

- *inserimento_coda ()*: inserisce un nuovo elemento di tipo *proc_elem*, con ordinamento prioritario decrescente. In caso di ugual priorità fra gli elementi l'inserimento avviene dopo tutti gli elementi con tale priorità.
- *rimozione_coda ()*: rimuovo dalla coda il primo elemento.
- *shedulatore ()*: seleziona dalla coda dei processi pronti, il primo elemento e lo inserisce nella coda esecuzione.

```
// sistema.cpp
...
void inserimento_ coda ( proc_elem*& p_coda, proc_elem* elem)
{/**/}
void rimozione_ coda ( proc_elem*& p_coda, proc_elem*& elem)
{/**/}
void schedulatore (void)
{/**/}
```

4.4.5 Atomicitá

In un sistema multiprogrammato alcune operazioni, eseguite su una risorsa da parte dei processi deve essere eseguita in modo atomico . L'insieme delle operazioni che devono essere eseguite in modo atomico su una risorsa condivisa, prende il nome di *sezione critica*. Un'azione *non interrompibile* é un'azione atomica. Un 'azione atomica puó essere interrompibile. L'interruzione puó mandare in esecuzione un nuovo processo, con il vincolo che questo non compia elaborazioni che causano danno all'azione iniziata dal processo interrotto. Ne segue subito il concetto di *mutua esclusione*, ovvero il controllore di una certa risorsa, una volta assegnata ad un processo, é proprio finche non termina la propria sezione critica relativa alla risorsa stessa. Per evitare che un processo trovi una risorsa in uno stato inconsistente, si protegge la sezione critica intervenendo sulle interruzioni mascherabili come segue.

atomico: nessun altro processo puó eseguire nessuna operazione finche non si completa l'esecuzione di tutte le operazioni su una risorsa.

```
disabilitare_interruzioni
sezione_critica
riabilitazione_interruzioni
```

All'interno di una sezione critica non ci sará mai una sospensione volontaria del processo in esecuzione, (non si possono invocare primitive che possono causare commutazione di contesto), per cui la sezione critica é sicuramente eseguita in modo atomico. Risulta evidente che una soluzione del genere porta alla conseguenza di esclusione mutua fra tutte le sezioni critiche di tutte le risorse. Tale soluzione risulta conveniente per sezioni critiche corte. Le routine del nucleo che

non operano sulle code dei processi vengono considerati sezioni critiche corte, e vengono protette anche essi disabilitando le interruzioni. Quindi tutte le routine del nucleo vengono eseguite con le interruzioni disabilitate. Si ricorre a soluzioni sofisticate nei seguenti casi:

- Nel caso in cui un processo utente si appresta ad eseguire una sezione critica lunga si ricorre a meccanismi piú sofisticati.
- Le sezioni critiche devono essere mutamente esclusivi solamente a gruppi, in relazione ad una certa risorsa.

4.4.6 Semafori

Un possibile modo della gestione della mutua esclusione sono i semafori. Quindi per proteggere una risorsa si ricorre a

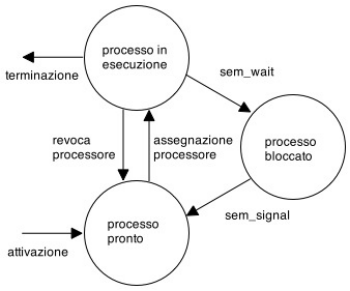
- un *semaforo*
- le primitive *sem_wait (), sem_signal ()*

Un processo che vuol utilizzare tale risorsa, se trova tale risorsa occupata perché un altro processo vi sta compiendo altre operazioni, si blocca volontariamente, dando luogo ad una commutazione di contesto. Quando una risorsá viene liberata, il processo bloccato diviene di nuovo pronto per l'esecuzione.

semaforo di mutua esclusione

Per un semaforo di mutua esclusione

- viene inizializzato a 1,
- la primitiva **sem_wait ()** ha come parametro il semaforo su cui opera.
  - # viene eseguita da un processo che vuole utilizzare una risorsa,
  - # essa decrementa il contatore, e se $contatore < 0$, blocca sul semaforo il processo. che ha invocato la primitiva stessa, viene quindi mandato in esecuzione un nuovo processo.
- la primitiva **sem_signal** ha come parametro su cui opera.
  - # essa incrementa il contatore, e se $contatore \leq 0$, viene sbloccato il processo a precedenza piú alta fra quelli a bloccati sul semaforo.
  - # se $contatore \leq 0 \Rightarrow |contatore| + 1$ sono i processi bloccati sul semaforo.
  - # Sbloccando un processo la cui prioritá é piú alta rispetto a quella del processo in esecuzione, puó verificarsi una *preemption* (prelazione). Tale vincolo non é presente quando non si hanno vincoli temporali sui processi.
  - # Un processo puó divenire pronto per effetto di una *sem_signal* eseguita da un altro processo.



4.4.7 Realizzazione dei semafori

Il descrittore di un semaforo é un struttura con due campi

- *counter* : per la gestione del semaforo, mediante la *sem_wiat ()* *e la sem_signal ()*
- *pointer* : a cui si accodano i processi che si bloccano sul semaforo stesso.

I descrittori di semaforo sono variabili definiti nel nucleo.

```
// sistema.cpp
...
struct des_sem
{
 int couter;
 proc_elem* pointer;
};

# sistema.s
...
.DATA
...
.GLOBAL  _array_dess
_array_dess:        .SPACE  BYTE_SEM  #MAX_SEM*sizeof (des_sem)
// sistema.cpp
...
extern des_sem array_dess[MAX_SEM];
```

Un semaforo é un indice di uno di questi descrittori, con un dato iniziale del campo *counter*. Il semaforo puó essere inizializzato con la primitiva *sem_ini ()*, invocabile dal programma main ().

```
void sem_ini (int& index_des_s, int val, bool& ris);
```

e ha come parametri

- variabile destinata a contenere l'indice del semaforo utilizzato,
- il valore iniziale del contatore.
- una variabile risultato.

```
// utente.cpp
extern "C"void sem_ini (int& index_des_s, int val, bool& ris);
extern "C" void sem_wait (int sem);
extern "C" void sem_signal (int sem);
...
int semaforo_i; // indice destinato a contenere un descrittore di semforo
...
int main ()
{
  bool ris;
  ...
  sem_ini (semaforo_i, VALORE, ris);
  ...
```

```
  begin_p ();
  ...
}

# utente.s
...
.TEXT
...
.GLOBAL _sem_ini
_sem_ini:
 INT $tipo_si
 RET
.GLOBAL _sem_wait
_sem_wait:
 INT $tipo_w
 RET
.GLOBAL _sem_signal
_sem_signal:
 INT $tipo_s
 RET

# sistema.s
...
.TEXT
...
a_sem_ini:
 #routine INT $tipo_si
 # ...
 IRET
 .EXTERN _c_sem_wait
a_sem_wait:
 #routine INT $tipo_w
 CALL salva_stato
 #ricopiamento del parametro sem
 CALL _c_sem_wait
 #ripulitura della pila
 CALL carica_stato
 IRET

 .EXTERN _c_sem_signal
a_sem_signal:
 #routine INT $tipo_s
 CALL salva_stato
 #ricopiamento del parametro sem
 CALL _c_sem_signal
 #ripulitura della pila
 CALL carica_stato
 IRET

// sistema.cpp
...
extern "C" void c_sem_wait (int sem)
{
 des_sem* s;
 s = &array_dess[sem];
  (s->counter)--;
 if ( (s->counter) < 0)
 {
  iserimento_coda (s->pointer, esecuzione);
  schedulatore ();
 }
}
...
extern "C" void c_sem_signal (int sem)
{
 des_sem* s; proc_elem* lavoro;
 s= &array_dese[sem];
  (s->counter)++;
 if  ( ( s->counter) <= 0)
 {
  rimozione_coda (s->pointer, lavoro);
  if ( (lavoro->priority) <=  (esecuzione->priority))
   inserimento_coda (pronti, lavoro);
  else
  { // preemption
   inserimento_coda (pronti, esecuzione);
```

```
    esecuzione = lavoro;
   }
}
```

I semafori possono utilizzati per

- la mutua esclusione
- la sincronizzazione

4.4.8 Mutua esclusione (impiego d)

Per proteggere una risorsa in mutua esclusione una risorsa, su cui vengono fatte da parte die processi op_1, op_2,..., op_n, viene utilizzato un semaforo, inizializzato ad 1.

```
// utente.cpp
...
int mutex;
short procm_a, procm_b;
void pma (int i)
{
 sem_wait (mutex);
 op_1;        // sezione critica
 sem_signal (mutex);
 ...
}

void pmb (int i)
{
 sem_wait (mutex);
 op_2;        // sezione critica
 sem_signal (mutex);
 ...
}
...
int main ()
{
 bool ris;
 ...
 sem_ini (mutex,1,ris);
 ...
 activate_p (pma,Ama,PRIOma,LIVma,procm_a,ris);
 activate_p (pmb,Amb,PRIOmb,LIVmb,procm_b,ris);
 ...
 begin_p ();
 ...
}
```

4.4.9 Sincronizzazione

Un semaforo se viene inizializzato con un valore pari a 0, puó essere utilizzato per scambiare segnali temporali tra due processi (*sincronizzazione*) .

```
// utente.cpp
...
```

```
int sincr;
short procs_a, procs_b;
void psa (int i)
{
 ...
 sem_wait (sincr);
 ...
}
void psb (int i)
{
 ...
 sem_signal (sincr);
 ...
}
...
int main ()
{
 bool ris;
 ...
 activate_p (psa,Asa,PRIOsa,LIVsa,procs_a,ris);
 activate_p (psb,Asb,PRIOsb,LIVsb,procs_b,ris);
 ...
 begin_p ();
}
```

Il processo procs_a si blocca quando esegue la sem_wait (), e diviene nuovamente pronto quando il processo procs_b esegue la sem_signal (). Se il processo procs_b esegue la sem_signal. per primo il processo procs_a non si blocca quando esegue la sem_wait () (l'evento si e' giá verificato). Tale sincronizzazione non é simmetrica.

4.4.10 Memoria dinamica

In un sistema multiprogrammato, la memoria dinamica deve essere gestita come una risorsa in mutua esclusione. Tale gestione si effettua mediante due routine di nucleo, mem_alloc () e free_free ().

```
void* mem_alloc (int dim);
void mem_free (void* pv);
```

Le routine girano a interruzioni disabilitate, e quindi é garantita la mutua esclusione. La mem_alloc ()

- alloca una struttura di dim byte,
- restituisce il puntatore al primo byte allocato
- il numero di byte allocati viene associato alla struttura allocata.

La mem_free ()

- libera la struttura indirizzata da pv, che deve essere stata allocata con la mem_alloc () e quindi ha associata l'informazione sul numero di byte.

```
// utente.s
...
.TEXT
```

```
...
.GLOBAL _mem_alloc
_mem_alloc:
 INT  $tipo_ma
 RET
...
.GLOBAL _mem_free
_mem_free:
 INT  $tipo_mf
 RET

#sisitema.s
...
.TEXT
...
.EXTERN _c_mem_alloc
a_mem_alloc:
 #routine INT tipo_ma
 #salvataggio dei registri man non di EAX
 #ricopiamento dei parametri
 CALL _c_mem_alloc
 #ripulitura della pila
 #ripristino dei registri  (non di EAX, che contiene il risultato)
 IRET
...
.EXTERN _c_mem_free
a_mem_free:
 #routine INT tipo_mf
 #salvataggio dei registri
 #ricopiamento dei parametri
 CALL _c_mem_free
 #ripulitura della pila
 #ripristino dei registr
 IRET

// sisitema.cpp
...
extern "C" void c_mem_alloc (int dim)
{ /**/ }
extern "C" void c_mem_free (void* pv)
{ /**/ }
```

Un processo utente deve effettuare la conversione di tipo per poter utilizzare tali routine.

```
// utente.cpp
extern "C" void* mem_alloc (int dim);
extern "C" void mem_free (void* pv);
...
void pro_mem (int i) // corpo del processo
{
 struct struttura{ /* */};
 struct* ps;
 ...
 ps = static_cast<struttura*> (mem_alloc ( sizefo (*ps)) );
 mem_free (static_cast<void*> (ps));
}
```

4.4.11 Utilizzo delle interfacce di conteggio

Un interfaccia di conteggio possiede i seguenti registri

- **CTR**: contiene la costante di conteggio,
- **STR**: consente di leggere il valore attuale del conteggio,
- **CWR**: specifica le modalitá di funzionamento. Le piú importanti modalitá di funzionamento sono:
  * *conta tempi* il decremento del contatore avviene con la frequenza di pilotaggio.

* *conta eventi* il decremento del contatore avviene con geli impulsi provenienti dall'esterno su apposito piedino.
* *ciclo singolo* quando il conteggio termina il funzionamento del contatore termina.
* *ciclo continuo* quando il conteggio termina, viene ricaricata nuovamente la costante nel contatore.
* *trigger hardware*: l'inizio del conteggio avviene quando arriva un fronte su un apposito piedino.
* *trigger software*:l'inizio del conteggio avviene quando viene scritta una nuova quantitá nel registro CTR.

L'interfaccia di conteggio possiede un contatore che

- inizializzato con il contenuto del registro CTR (16 bit)
- viene decrementato con una certa sequenza di impulsi
- quando passa da 1 a 0 (termina il conteggio) l'interfaccia attiva un piedino di uscita, che puó essere utilizzato
  - per generare una richiesta di inetrruzione,
  - per pilotato un dispositivo esterno.

Timer di sistema

Carichiamo una costante di tempo in modo da vare un'interruzione ogni 50ms, si ottiene in questo modo un *timer di sistema*. In un sistema tempo reali, i processi possono sospendersi per tante unita ciascuna pari a 50ms. Per gestire la sospensione programmata si dispone di un descrittore di timer (puntatore descittore_timer), sul quale si possono accodare elementi che individuano processi che si sospendono sul timer.

Per potersi, un processo deve invocare la primitiva *delay ()*, specificando il numero di intervalli di tempo in cui vuol rimanere bloccato. Lo stesso processo viene rimesso diviene pronto, non appena trascorre il tempo di blocco.

```
// sistema.cpp
...
struct richiesta
{
 int d_attesa;
 richiesta* p_rich;
 proc_elem* pp;
};
richiesta* descrittore_timer;

// utente.cpp
...
extern "C" void delay (int n);
void procd (int i) // corpo del processo
{
 ...
delay (N);
 ...
}
#utente.s
...
.TEXT
...
.GLOBAL _delay
_delay:
```

```
 INT $tipo_d
 RET
# sistema.s
...
.TEXT
...
.EXTERN _c_delay
a_delay:
 #routine INT $tipo_d
 CALL salva_stato
 #ricopiamento dei parametri
 CALL _c_delay
 #ripulitura delle pila
 CALL carica_stato
 IRET
// sistema.cpp
...
void inserimento_coda_timer (richiesta* p);
extern "C" void c_delay (int n)
{
 richiesta* p;
 p  = new richiesta;
 p->d_attesa = n;
 p->pp = esecuzione;
 inserimento_coda_timer (o);
 schedulatore ();
}
```

La primitiva *delay ()* utilizza la funzione di utilitá *insermimento_coda_timer ()*. La coda del timer é organizzata in modo efficiente, e prevede che il numero di intervalli di attesa di un certo elemento é dato dalla somma die campi d_attesa di tutti gli elementi che lo precedono e quello dell'elemento stesso. La gestione della coda da parte del driver di interruzione

- si limita a decrementare il contenuto del cammpo *d_attesa* del primo elemento.
- inserisce nella coda dei processi pronti gli elementi gli elementi che hanno il campo *d_attesa* uguale a 0.
- ad ogni nuovo inserimento, viene decrementato il valore del campo *d_attesta* dell'elemento da inserire delle unitá *d_attesa* e aggiustati i valori dei campi rimanenti successivi all'elemento inserito.

```
// sistema.cpp
...
// inserisce P nella coda delle richieste al timer
void inserimento_coda_timer (richiesta *p)
{
 richiesta *r, *precedente;
 bool ins;
 r = descrittore_timer;
 precedente = 0;
 ins = false;
 while (r != 0 && !ins)
   if (p->d_attesa > r->d_attesa)
   {
     p->d_attesa -= r->d_attesa;
     precedente = r;
     r = r->p_rich;
    } else
         ins = true;
   p->p_rich = r;
   if (precedente != 0)
     precedente->p_rich = p;
   else
     descrittore_timer = p;
  if (r != 0)
  r->d_attesa -= p->d_attesa;
}
```

Ongi 50 ms arriva un'interruzione timer e va in esecuzione un driver che

- salva lo stato (salva_stato)
- pone il processo interrotto in testa alla coda dei processi pronti (si tratta del processo a piú alta prioritá), utilizzando la routine *inspronti*.
- decrementa d_attesa del primo elemento in coda e immette nella coda dei processi pronti gli elementi con d_attesa pari a 0.
- richiama lo schedulatore.

```
# sistema.s
...
.TEXT
...
inspronti:
 #...
 ret
.EXTERN _c_driver_t
driver_t:
 CALL saslva_stato
 CALL inspronti
 CALL _c_driver_t
 CALL carica_stato
 IRET
// sistema.cpp
...
extern "C" void c_driver_t (void)
{
 richiesta* p;
 if (descrittore_timer != 0)
  descrittore_timer->d_attesa--;
 while (descittore_timer != 0 && descittore_time->d_attesa == 0)
 {
  inserimento_coda (pronti, descittore_timer->pp);
  p = descrittore_timer;
  descittore_timer = descittore_timer->p_rich;
  delete p;
 }
 schedulatore ();
}
```

La c_delay e c_driver vengono eseguite con le interruzioni disabilitate e vengono utilizzati gli operatore *new* e *delete* nelle routine di nucleo, nell'ipotesi che questi non riabilitino le interruzioni.

4.5 Operazione di I/O

Nel corpo di un processo non si possono utilizzare routine predefinite per le operazione di I/O, perché non gestiscono le operazioni di I/O in modo atomica. Occorre dunque definire nuove routine di I/O, dette *primitive di I/O*. Questo costituisce un vantaggio:

- Le primitive di I/O non effettuano commutazione di contesto in modo esplicito, in quanto non operano sulle code dei processi, quindi possono girare con le interruzioni mascherabili abilitate.
- Le operazioni di I/O possono avvenire anche ad interruzione di programma.

***definizione* 9** *Le operazioni di I/O si dicono* SINCRONE *o* ASINCRONE *a seconda che il processo le richiede, attenda la loro terminazione, oppure prosegue eseguendo altre elaborazioni.*

Nella multiprogrammazione si utilizzando in genere le operazioni di I/O sincrone, e prevedono che il processo che le richiede si blocchi per divenire di nuovo pronto quando l'operazione é terminata. La gestione della sincronizzazione (attesa di fine operazione di I/O) e mutua esclusione é affidata alle primitive di I/O.

Le primitive di I/O hanno lo stesso livello di privilegio delle istruzioni di I/O, visto che utilizzano queste per effettuare un operazione di I/O. Si supporrá che queste abbiano livello di privilegio sistema ma che non facciano parte del nucleo, visto che girano con le interruzioni mascherabili abilitate. L'implementazione delle primitive di I/O é la medesima vista per le routine di nucleo, salvo il seguente

- le interruzione software utilizzate per primitive di I/O coinvolgono gate di tipo Trap, che quindi non disabilita le interruzione.
  - # possono essere coinvolti anche i gate di tipo Interrupt purche le primitive provvedano a riabilitare le interruzioni mascherabili.

```
// utente.cpp
...
extern "C" prim_io_i (/*parametri formali*/);
...
void proc_io (/*parametri formali*/)
 ... prim_io_i (/*parametri attuali*/); ...
#utente.s
...
.TEXT
...
.GLOBAL _prim_io_i
_priim_io_i:
 INT $io_tipo_i
ret_io_i:
 RET
 ...
#sistema.s
...
.TEXT
...
.EXTERN _c_prim_io_i
a_prim_io_i:
 #routine INT $io_tipo_i
 #STI per gate di tipo Interrupt
 #salvataggio dei registri
 #ricopiamento parametri
 CALL _c_prim_io_i
 #ripulitura della pila
 #ripristino dei registri
 IRET
 ...
 //sistema.cpp
 ...
 extern "C" void c_prim_io_i (/*parametri formali*/)
 /**/
```

Le primitive di I/O utilizzano le primitive semaforiche.

```
//sistema.cpp
...
extern "C" void sem_wiat (int sem);
extern "C" void sem_signal (int sem);
...
#sistema.s
...
.TEXT
...
.EXTERN _sem_wait
_sem_wait:
 INT $tipo_w
 rit_w:
  RET
.EXTERN _sem_signal
_sem_signal:
 INT $tipo_s
 rit_s:
  RET
```

4.5.1 Ingresso/Uscita a interruzione di programma

Si considera che le operazioni di I/O

- trasferiscono byte,
- utilizzano il meccanismo di interruzione per il trasferimento.

Una primitiva di I/O

- ha come parametri,
  - indirizzo di un buffer di memoria
  - numero di byte da trasferire
- attiva una specifica interfaccia (routine start_io ()); l'interfaccia attivata
  - scambia i singoli byte con il dispositivo esterno,
  - genera richieste di interruzione in presenza di un byte da scambiare con il processore,
  - riceve la risposta, come operazione di lettura / scrittura nel buffer della stessa, e ne segue un nuovo ciclo del meccanismo di scambio di un byte.

  Se un'operazione comporta il trasferimento di n byte, comporta la generazione di n richieste di interruzione da parte dell'interfaccia.

  Ogni interruzione manda in esecuzione lo specifico *driver*, che
  - utilizza un puntatore (punt) contenete l'indirizzo del byte da trasferire, e un contatore (cont) che specifica il numero di byte da trasferire.
  - gestisce i trasferimenti, che costituiscono la risposta all'interfaccia,

    - effettuare il trasferimento,
    - aggiornare il puntatore e il contatore, incrementandoli.
    - verifica che l'ultima operazione non sia terminata.
  - esegue un *signal*, se l'operazione é terminata, sul semaforo su cui il processo utente si era bloccato, dopo aver disattivato l'interfaccia (*hatl_io ()*), e richiama lo schedulatore.

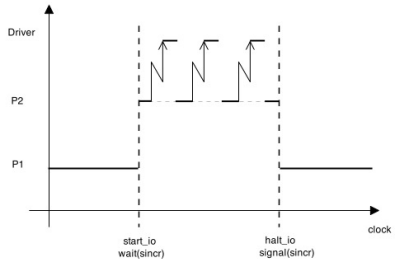
- esegue un'operazione di *wait* su semaforo,bloccandosi in attesa che l'operazione termini. Una volta che viene eseguita la *signal* sul semaforo dal driver, il processo utente diviene pronto, e puó essere schedulato.

***Esempio* 4.2** *Consideriamo due processi P1 e P2, e supponiamo che P1 abbia effettuato un'operazione di I/O, secondo le modalitá specificate. P1 per effetto dell'operazione di* wait *si blocca sul semaforo di sincronizzazione, e lo schedulatore manda in esecuzione il processo P2. Il driver interrompe il processo P2, ogni volta che viene mandato in esecuzione come effetto di un interruzione prodotta dall'interfaccia che porta avanti l'operazione di I/O. A fine operazione, fa divenire pronto P1.*

Nota *48*

- *come previsto dal meccanismo dell'interruzione, il driver, per essere eseguito, necessita di un cambio di livello se il processo P2 interrotto e un processo utente. Si puó dedurre qualunque sia il livello di privilegio di P2, il meccanismo dell'interruzione opera sulla pila sistema del processo P2, se questo é il processo in esecuzione.*

- *Il buffer di memoria, specificato da P1, deve risiedere nello spazio di indirizzamento comune, e non deve essere soggetto a rimpiazzamento, dovendo essere utilizzato dal driver (il buffer) quando é in esecuzione P2.*

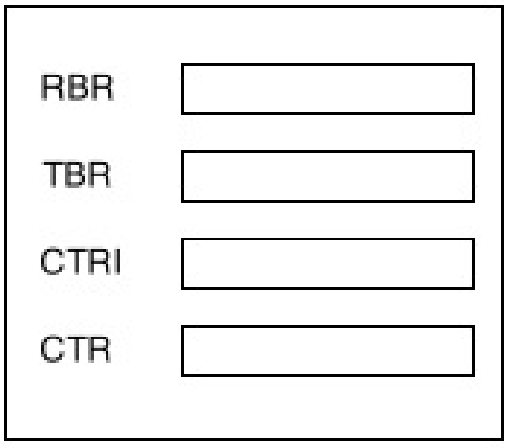


4.5.2 Descrittori di operazione di I/O

Le interfacce sono risorse condivise, pertanto devono essere utilizzate in mutua esclusione, per precludere situazioni di non consistenza delle operazioni di I/O nel caso piú di un processo effettua la stessa operazione di I/O, utilizzando la medesima interfaccia. Si prevede dunque un descrittore di I/O contenente

- informazioni per la gestione dell'interfaccia in grado di effettuare sia operazioni di ingresso che di uscita,
    * l'interfaccia é dotata di due piedini per effettuare richieste distinte, in base al tipo di operazione, uno per la scrittura e uno per la lettura.
- un semaforo di muta esclusione,
- un semaforo per la sincronizzazione.

Funzionalmente l'interfaccia é vista come segue



- *RBR*: registro buffer di lettura,
- *TBR*: registro buffer di scrittura,
- *CTRI*: registro di controllo per l'abilitazione di richieste di interruzione in ingresso,
- CTRO: registro di controllo per l'abilitazione di richieste di interruzioni in uscita.

```
// sistema.cpp
...
typedef char* ind_b;
struct interf_reg
{
 union { ind_b iRBR, iTBR;} in_out;
 union {ind_b iCTRI, iCTRO;} ctr_io;
};
struct des_io
{
 interf_reg indreg; // indirizzi dei registri
 int mutex;
 int sincr;
 int cont;
 ind_b punt;
};
extern des_io desinti[T];
extern des_io desinto[T];
...
#sistema.s
.DATA
...
.GLOBAL _desinti
.GLOBAL _desinto
_desinti:        .SPACE BYTE_IO        #T*sizeof (des_io)
_desinto:        .SPACE BYTE_IO
```

I descrittori devono essere inizializzati:

- inizializzazione del campo *indreg*, con gli indirizzi di iRBR e iCTRI, per le operazioni di ingresso, e iTBR e iCTRO per le operazioni di uscita,
- inizializzazione del campo mutex con il valore 1 (sem_ini ())
- inizializzazione del campo sincr con il valore 0 (sem_ini ()).

Sottoprogrammi di utilitá

```
# sistema.s
...
.TEXT
.GLOBAL _inputb
_inputb:
 #...
 RET
.GLOBAL go_inputb
go_inputb:
 #...
 RET
 .GLOBAL halt_inputb
halt_inputb:
 #...
 RET
.GLOBAL _outputb
_outputb:
 #...
 RET
.GLOBAL go_outputb
go_outputb:
 #...
 RET
 .GLOBAL halt_outputb
halt_outputb:
 #...
 RET
 // sistema.cpp
 ...
extern "C" void inputb (ind_b i_ctr, char& a);
extern "C" void go_inputb (ind_b i_ctr);
extern "C" void halt_inputb (ind_b i_ctr);
extern "C" void outputb (ind_b i_ctr, char a);
extern "C" void go_outputb (ind_b i_ctr);
extern "C" void halt_outputb (ind_b i_ctr);
```

4.5.3 Operazione di lettura

La primitiva di I/O per la lettura di n byte é *a_read_n ()*, la quale richiede come parametri:

- il numero d'ordine dell'interfaccia da utilizzare,
- l'indirizzo del buffer di memoria nel qual devono essere scritti i byte letti,
- il numero di byte da leggere.

Un processo utente necessita di un programma di interfaccia per l'invocazione della primitiva.

```
//utente.cpp
...
char buffer_r[N_r];
extern "C" void read_n (int interf, char vetti[], int quanti);
void proc_int (int h)
```

```
{
 int interf , nn;
 ...
 read_n (interf, buffer_r, nn);
 ...
}
...
#utente.s
...
.TEXT
...
.GLOBAL _read_n
_read_n:
 INT $io_tipo_r
 rit_r:
  RET
 #sistema.s
...
.TEXT
...
.EXTERN _c_read_n:
a_read_n:
 #routine INT $io_tipo_r
 #STI per gate di tipo intrrupt
 #salvataggio dei registri
 #ricopiamento parametri
 CALL _c_read_n
 #ripulitura delle pile
 #ripristino dei registri
 IRET
...
// sistema.cpp
...
void start_in (des_io* p_desi, char vetti[], in quanti);
extern "C" void c_read_n (int interf, char vetti[], int quanti)
{
 des_io* p_desi;
 p_desi = & desinti[interf]; // i descrittori sono inizializzati
 sem_wait (p_desi->mutex);
 start_in (p_desi, vetti, quanti);
 sem_wait (p_desi->sincr);
 sem_signal (p_desi->mutex);
}
```

La funzione *start_in ()* trasferisce i parametri nel corrispondente descrittore di I/O, e abilita l'interfaccia a effettuare richieste di interruzione.

```
//sistema.cpp
...
void start_in (des_io* p_desi, char vetti[], int quanti)
{
 p_desi->cont = quanti;
 p_desi->punt = vetti;
 go_input (p_desi->indreg.ctr_io.iCTRI);
}
```

Il driver che viene mandato in esecuzione quando viene generata un'interruzione dall'interfaccia di ingresso, opera sul corrispondente descrittore di I/O. Questo opera anche sulle code dei processi, quindi deve girare con le interruzioni esterne mascherabili disabilitate.

```
#sistema.s
...
.TEXT
...
.EXTERN _c_driverin_i
int_tipoi_i:
 CALL salva_stato
 CALL inspronti
 CALL _c_driverin_i
 CALL carica_stato
 IRET
 ...
//sistema.cpp
...
```

```
extern "C" void c_driverin_i (void) // opera su desinti[i]
{
 cost i = ....;
 char cc;
 proc_elem* lavoro;
 des_sem* s;
 des_io* p_desi;
 p_desi = &desinti[i];
 p_desi->count--;
 if (p_desi->cont == 0)
 {
  halt_input (p_desi->indreg.ctr_io.iCTRI);
  s = & array_dess[p_desi->sincr];
  s->counter++;
  if  (s->counter <= 0)
  {
   rimozione_coda (s->pointer,lavoro);
   inserimento_coda (pronti, lavoro);
  }
 }
 inputb (p_desi->indreg.in_out.iRBR,cc);
 * (p_desi->punt) = cc;
 p_desi->punt++;
 schedulatore ();
}
```

***Nota* 49** *Le ultime operazioni, equivalgono alla sem_signal senza preemption. Visto che il driver non ha descrittore la primitivia sem_signal () non puó essere utlizzata. Infatti sia la sem_wait che la sem_singal, effettuano prima il salvataggio dello stato del processo in esecuzione, e in questo caso il driver non é un processo.*

Un processo utente, invocando una primitiva di I/O, abilita l'interfaccia a generare richieste di interruzione per mezzo della sem_ini (). Puó accadere che il driver vada in esecuzione per l'ultima volta, eseguendo la sem_signal () (incrementando il contatore del semaforo portandolo al valore 1), prima che il processo l'utente possa eseguire la sem_wait (). In tal caso il processo utente, eseguendo la sem_wait () su sincr, non si blocca.

4.5.4 Operazione di scrittura

Mutatis mutandis dall'operazione di lettura si ottiene l'operazione di scrittura.

```
//utente.cpp
...
char buffer_r[N_w];
extern "C" void write_n (int interf, char vetto[], int quanti);
void proc_int (int h)
{
 int interf , nn;
 ...
 write_n (interf, buffer_w, nn);
 ...
}
...
#utente.s
...
.TEXT
...
.GLOBAL _write_n
_write_n:
 INT $io_tipo_w
 rit_w:
  RET
 #sistema.s
...
.TEXT
...
```

```
.EXTERN _c_write_n:
a_write_n:
 #routine INT $io_tipo_r
 #STI per gate di tipo intrrupt
 #salvataggio dei registri
 #ricopiamento parametri
 CALL _c_write_n
 #ripulitura delle pile
 #ripristino dei registri
 IRET
...
// sistema.cpp
...
void start_out (des_io* p_deso, char vetto[], in quanti);
extern "C" void c_write_n (int interf, char vetto[], int quanti)
{
 des_io* p_deso;
 p_deso = & desinto[interf]; // i descrittori sono inizializzati
 sem_wait (p_deso->mutex);
 start_in (p_deso, vetto, quanti);
 sem_wait (p_deso->sincr);
 sem_signal (p_deso->mutex);
}
```

La funzione *start_ou ()* trasferisce i parametri nel corrispondente descrittore di I/O, e abilita l'interfaccia a effettuare richieste di interruzione.

```
//sistema.cpp
...
void start_out (des_io* p_deso, char vetto[], int quanti)
{
 p_deso->cont = quanti;
 p_deso->punt = vetto;
 go_output (p_deso->indreg.ctr_io.iCTRO);
}
```

Il driver che viene mandato in esecuzione quando viene generata un'interruzione dall'interfaccia di uscita, opera sul corrispondente descrittore di I/O. Questo opera anche sulle code dei processi, quindi deve girare con le interruzioni esterne mascherabili disabilitate.

```
#sistema.s
...
.TEXT
...
.EXTERN _c_driverout_i
int_tipoo_i:
 CALL salva_stato
 CALL inspronti
 CALL _c_driverout_i
 CALL carica_stato
 IRET
 ...
//sistema.cpp
...
extern "C" void c_driverout_i (void) // opera su desinto[i]
{
 cost i = ....;
 char cc;
 proc_elem* lavoro;
 des_sem* s;
 des_io* p_deso;
 p_deso =&desinto[i];
 p_deso->count--;
 if (p_deso->cont == 0)
 {
  halt_output (p_deso->indreg.ctr_io.iCTRO);
  s = & array_dess[p_deso->sincr];
  s->counter++;
  if  (s->counter <= 0)
  {
   rimozione_coda (s->pointer,lavoro);
   inserimento_coda (pronti, lavoro);
  }
 }
```

```
 cc = * (p_desi->punt) ;
 outputb (cc, p_deso->indreg.in_out.iTBR);
 p_desi->punt++;
 schedulatore ();
}
```

4.6 Processi Esterni

La sezione critica di un driver puó essere lunga, tale che il tempo di esecuzione della stessa, non é accettabile come vincolo temporale in cui il processore puó avere le interruzioni mascherabili disabilitate. Il vincolo dato nel nostro caso, é il tempo massimo accettabile per l'implementazione di un sistema real-time. Se tale problema si presentasse puó essere risolto, con la seguente implementazione, del driver

* un insieme di *handler**
+
*un processo *esterno* (o processo driver)*

Un ulteriore vantaggio che ne consegue, é che il processo esterno puó fare uso, liberamente, delle primitive di nucleo , cosa non praticabile con il solo driver.

4.6.1 Driver e processi esterni

Quando arriva un'interruzione esterna va in esecuzione uno specifico *handler*, che provoca una commutazione di contesto.

- Mette come primo processo pronto il processo interrotto,
- manda in esecuzione il *processo esterno*, il quale esegue i trasferimenti richiesti.

Si hanno pertanto

- processi esterni (processo a livello di privilegio I/O), associati ai descrittori di I/O. Sia le primitive di I/O che i descrittori di I/O verranno posizionati a livello di
- processi interni
  - processi utente
  - processi sistema

Un handler é una routine di sistema, pertanto gira con le interruzioni disabilitate. Invece un processo esterno gira con le interruzioni mascherabili esterne abilitate. Quando vá in esecuzione un driver vengono disattivate le interruzione, come previsto dal meccanismo di interruzione, ma vengono riabilitate non appena viene mandato in esecuzione il processo esterno, in quando si carica in EFLAG un nuovo valore dal descrittore del processo esterno stesso.

Si prevede in EFLAGv nel descrittore del processo esterno che il bit IF sia pari a 1.

***Nota* 50** *Questo modo di realizzare i driver, permette l'annidamento delle interruzioni esterne. Il controllore delle interruzioni deve essere programmato per gestire l'annidamento delle interruzioni, quindi ricevere il comando il comando EOI alla terminazione di un processo esterno.*

I processi esterni, come tutti i processi hanno un proprio identificatore, che tramite la GDT, individua il descrittore. L'attivazione dei processi esterni avvengono in fase di inizializzazione. La primitiva per l'attivazione di un processo esterno é *activate_pe ()* e ha la seguente firma

```
void activate_pe (void f (int), int a, int prio, char liv,
        short& identifier, proc\_elem*& p, bool& risu);
```

Il processo una volta attivato, il processo esterno viene inserito in una particolare coda, costituita da un solo elemento, in attesa di un tipo idoneo di interruzione. La coda i-ma é individuata dal puntatore p-i

La struttura prototipo di un handler é

```
#sistema.s
...
.TEXT
...
handler_i:
 CALL salva_stato
 CALL inspronti
 #mettere in esecuzione il contenuto del puntatore p_i che
 #individua un determinato processo esterno.
 CALL carica_stato
 IRET
```

Esistono tanti handler quanti sono i tipi delle interruzioni prodotte dalle interfacce di ingresso / uscita, come é previsto dal meccanismo di interruzione. Il processo esterno puó utilizzare le due primitive sem_wait () e sem_signal ()

```
// utente.cpp
...
extern "C" void sem_wait (int sem);
extern "C" void sem_signal (int sem);
...

# io.s
...
.TEXT
...
.GLOBAL _sem_ini
_sem_ini:
 INT $tipo_si
 RET
.GLOBAL _sem_wait
_sem_wait:
 INT $tipo_w
 RET
.GLOBAL _sem_signal
_sem_signal:
 INT $tipo_s
 RET
```

4.6.2 Struttura di un processo esterno

Un processo esterno, per un operazione di I/O che comporta il trasferimento di n byte, va in esecuzione n volte. All'ultimo trasferimento pone nella coda dei processi pronti il processo utente che si era bloccato, sul semaforo di sincronizzazione in attesa del termine dell'operazione di I/O. Al termine di ogni esecuzione del processo esterno

- invio della parola EOI, in modo che possa servire anche richieste di interruzioni a precedenza inferiore.
- salvataggio dello stato attuale

- richiama lo schedulatore, che seleziona un processo ad alta priorità in testa alla coda dei pronti
  - il processo interrotto nelle esecuzioni intermedie
  - il processo esterno interrotto
  - un processo divenuto pronto nell'esecuzione finale.
- caricamento dello stato del nuovo stato.

Una volta inviato il comando di EOI il processo esterno si blocca in attesa che venga riattivato nuovamente. L'operazione di blocco non é esplicita, ovvero non avviene per via di un semaforo, ma provocando una commutazione di contesto volontaria. Le operazioni di epilogo vengono realizzate della primitiva *wifi ()* (Wait For Interrupt). Ne segue che struttura di un processo risulta la seguente .

```
..
extern "C" void wfi ();
...
vodi extern (int h) // corpo del processo esterno
{
...
 while (1) // ciclo infinito
 {
   ...
   //implementazione del driver
   ...
   // epilogo
   wfi ();
 }
}
```

***Nota* 51** *Un interruzione a precedenza maggiore rispetto a quella sotto servizio modifica il descrittore del processo in esecuzione, che lo fa terminare in uno stato S_i, che dipende dal punto in cui é stato interrotto. Appena il processo riprende l'esecuzione partirá dallo stato S_i. E se il processo si bloccasse senza salvare nel descrittore uno nuovo stato pari a S_1 (EIP punta alla prossima istruzione che effettua il trasferimento), punto in cui inizia il trasferimento, succede che nelle interruzioni successive partirá sempre dallo stato aleatorio S_i senza alcun significato per l'operazione di I/O. Da qui la necessitá della primitiva wfi (), che prevede il* salvataggio del suo stato *prima che si blocchi. Tale salvataggio dello stato ,sia S_k risolve il problema dello stato aleatorio. Affinché venga salvato lo stato S_k corrisponda a S_1 la struttura del processo esterno deve essere ciclica. Il processo si blocca quando le si sono concluse le operazioni di epilogo, per il singolo trasferimento, e una nuova esecuzione forzata del processo (che avviene per effetto di un'interruzione esterna dello stesso tipo) fa ripartire il processo dallo stato S_k (viene riporestinato EIP che punta alla successiva), punto successivo al blocco in cui deve comparire un salto da cui inizia il trasferimento, per cui lo stato S_k corrisponde esattamente con lo stato desiderato.*

Sempre per un'interruzione a precedenza maggiore, rispetto a quella sotto servizio, lo handler pone il processo interrotto nella coda dei processi pronti. Ne segue che i processi esterni sono presenti nella coda dei processi pronti.

```
// io.cpp
...
typedef char* ind_b;
struct interf_reg
{
```

```
 union { ind_b iRBR, iTBR;} in_out;
 union {ind_b iCTRI, iCTRO;} ctr_io;
};
struct des_io
{
 interf_reg indreg; // indirizzi dei registri
 int mutex;
 int sincr;
 int cont;
 ind_b punt;
};
extern des_io desinti[T];
extern des_io desinto[T];
 ...
extern "C" void inputb (ind_b i_ctr, char& a);
extern "C" void go_inputb (ind_b i_ctr);
extern "C" void halt_inputb (ind_b i_ctr);
extern "C" void outputb (ind_b i_ctr, char a);
extern "C" void go_outputb (ind_b i_ctr);
extern "C" void halt_outputb (ind_b i_ctr);
...
#io.s
.DATA
...
.GLOBAL _desinti
.GLOBAL _desinto
_desinti:        .SPACE BYTE_IO        #T*sizeof (des_io)
_desinto:        .SPACE BYTE_IO

...
.TEXT
.GLOBAL _inputb
_inputb:
 #...
 RET
.GLOBAL go_inputb
go_inputb:
 #...
 RET
 .GLOBAL halt_inputb
halt_inputb:
 #...
 RET
.GLOBAL _outputb
_outputb:
 #...
 RET
.GLOBAL go_outputb
go_outputb:
 #...
 RET
 .GLOBAL halt_outputb
halt_outputb:
 #...
 RET
```

4.6.3 La primitiva wfi ()

La wfi (), in realtá é un sottoprogramma di interfaccia, verso la vera primitiva a_wfi che

- effettua il salvataggio dello stato del processo del processo esterno che la invoca,
- invia la parola EOI al controllore delle interruzioni,
- richiama lo schedulatore,
- carica lo stato del nuovo processo.

```
#io.s
...
.TEXT
```

```
.GLOBAL _wfi
_wfi: INT $tipo_wfi
rit_wfi: RET
...
#sistema.s
....
.TEXT
...
.EXTERN _schedulatore
.set EOI, ...
.set OCW2, ...
a_wfi: # routine int $tipo_wfi
 CALL salva_stato
 MOVB $EOI, %AL
 MOVW $OCW2, %DX
 OUTB %AL, %DX
 CALL _schedulatore
 CALL carica_stato
 IRET
```

***Nota* 52** *L'invio della parola EOI da parte del processo esterno, deve far parte della wfi (). Se cosí non fosse, poterebbe giungere al processore una nuova richiesta dalla stessa fonte, prima che esso possa bloccarsi. Non si possono avere piú istanze contemporanee dello stesso processo, visto che ogni processo ha un solo descrittore. Tuttavia se un processo esterno potesse essere interrotto, da una seconda richiesta di interruzione proveniente dalla stessa fonte, lo handler salverebbe lo stato relativo al punto di interruzione del processo. La seconda istanza dello stesso processo riprenderebbe dal punto successivo all'interruzione, e si sospenderebbe per effetto della della wfi (), senza effettuare nessuna elaborazione. La prima elaborazione riprende dal punto successivo alla wfi, avendo cosi un ricupero delle operazioni che spettavano alla seconda istanza.*

4.6.4 Lettura con i processi esterni

La primitiva di I/O read_n (), é sottoprogramma d'interfaccia.

```
#io.s
...
.TEXT
...
.EXTERN _c_read_n:
a_read_n:
 #routine INT $io_tipo_r
 #STI per gate di tipo intrrupt
 #salvataggio dei registri
 #ricopiamento parametri
 CALL _c_read_n
 #ripulitura delle pile
 #ripristino dei registri
 IRET
// sistema.cpp
...
void start_in (des_io* p_desi, char vetti[], in quanti);
extern "C" void c_read_n (int interf, char vetti[], int quanti)
{
 des_io* p_desi;
 p_desi = & desinti[interf]; // i descrittori sono inizializzati
 sem_wait (p_desi->mutex);
 start_in (p_desi, vetti, quanti);
 sem_wait (p_desi->sincr);
 sem_signal (p_desi->mutex);
 }

 void start_in (des_io* p_desi, char vetti[], int quanti)
{
 p_desi->cont = quanti;
 p_desi->punt = vetti;
```

```
 go_input (p_desi->indreg.ctr_io.iCTRI);
}
```

Quando avviene un'interruzione causata dall'interfaccia abilitata, va in esecuzione lo specifico handler associato al tipo di interruzione. Il processo esterno mandato forzatamente, se tutte le interfacce sono uguali, il suo corpo é rientrante e puó essere condiviso, salvo il fatto che questo opera su descrittori diversi, e discriminato dal parametro del processo.

```
#sisitema.s
...
.TEXT
...
handler_i:
 CALL salva_stato
 CALL inspronti
 #mette in esecuzione il contenuto del puntatore p_i
 #che individua lo specifico processo esterno
 CALL carica_stato
 IRET

// io.cpp
...
// copo comune a tutti i processi
void esterni (int h)
{
 char cc;
 des_sem* s;
 des_io* p_desi;
 p_desi = & desinti[h];
 while (1)
 {
  p_desi->count--;
  if (p_desi->cont == 0)
   halt_input (p_desi->indreg.ctr_io.iCTRI);
 inputb (p_desi->indreg.in_out.iRBR,cc);
 * (p_desi->punt) = cc;
  if (p_desi->cont == 0)
   sem_signal (p_desi->sincr);
 p_desi->punt++;
 wfi ();
}
```

***Nota* 53** *Il valore p_desi si trova nella pila a livello di privilegio I/O del processo che esegue la read_n (), quindi é una variabile privata. Questo in accordo all'ipotesi che il corpo del processo é un codice rientrante.*

4.6.5 Scrittura con i processi esterni

Per le operazioni di scrittura si ottiene mutatis mutandis dall'operazione di lettura La primitiva di I/O write_n (), é sottoprogramma d'interfaccia.

```
#io.s
...
.TEXT
...
.EXTERN _c_write_n:
a_write_n:
 #routine INT $io_tipo_w
 #STI per gate di tipo intrrupt
 #salvataggio dei registri
 #ricopiamento parametri
 CALL _c_write_n
 #ripulitura delle pile
 #ripristino dei registri
 IRET
// sistema.cpp
...
void start_out (des_io* p_deso, char vetto[], in quanti);
extern "C" void c_write_n (int interf, char vetto[], int quanti)
```

```
{
 des_io* p_deso;
 p_deso = & desinto[interf]; // i descrittori sono inizializzati
 sem_wait (p_deso->mutex);
 start_in (p_deso, vetto, quanti);
 sem_wait (p_deso->sincr);
 sem_signal (p_deso->mutex);
 }

 void start_out (des_io* p_deso, char vetto[], int quanti)
{
 p_deso->cont = quanti;
 p_deso->punt = vetto;
 go_out (p_deso->indreg.ctr_io.iCTRO);
}
```

Quando avviene un'interruzione causata dall'interfaccia abilitata, va in esecuzione lo specifico handler associato al tipo di interruzione. Il processo esterno mandato forzatamente, se tutte le interfacce sono uguali, il suo corpo é rientrante e puó essere condiviso, salvo il fatto che questo opera su descrittori diversi, e discriminato dal parametro del processo.

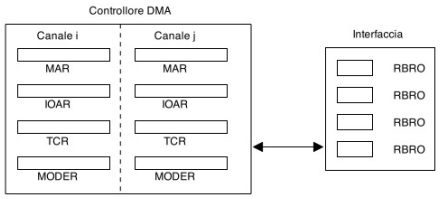
```
#sisitema.s
...
.TEXT
...
handler_i:
 CALL salva_stato
 CALL inspronti
 #mette in esecuzione il contenuto del puntatore p_i
 #che individua lo specifico processo esterno
 CALL carica_stato
 IRET

// io.cpp
...
// copo comune a tutti i processi
void esterni (int h)
{
 char cc;
 des_sem* s;
 des_io* p_deso;
 p_deso = & desinto[h];
 while (1)
 {
  p_deso->count--;
  if (p_deso->cont == 0)
   halt_output (p_deso->indreg.ctr_io.iCTRI);
 cc= * (p_deso->punt) ;
 outputb (cc,p_deso->indreg.in_out.iTBR);
  if (p_deso->cont == 0)
   sem_signal (p_deso->sincr);
 p_deso->punt++;\begin{}

\end{}
 wfi ();
}
```

***Nota* 54** *Il valore p_deso si trova nella pila a livello di privilegio I/O del processo che esegue la write_n (), quindi é una variabile privata. Questo in accordo all'ipotesi che il corpo del processo é un codice rientrante.*

4.7 Ingresso / Uscita in DMA

Descriviamo le operazioni di I/O effettuate con il meccanismo di accesso diretto alla memoria (DMA). Si suppone di fare una analogia con il trasferimento a interruzione di programma, e quindi si trasferiscono solo byte, con il tipo di ciclo doppio. I due canali e l'interfaccia ad essi collegata appare al programmatore come in figura

Ciclo doppio: ciclo di memoria e ciclo di I/O per ogni trasferimento.



4.7.1 Accesso diretto alla memoria

- I registri del DMA sono a 32 bit,
- I registri dell'interfaccia a 8 bit.

Un canale del DMA é si fatto

- *MAR* contiene l'indirizzo fisico del byte da trasferire.
- *IOAR* contiene, nei due byte meno significativi, l'indirizzo del registro buffer.
- *TCR* contiene il numero di byte-1 da trasferire.
- *MODER* contiene nel byte meno significativo, il valore per stabilire la modalitá di funzionamento

Dopo ogni trasferimento il valore di MAR viene incrementato e quello di TCR decrementato (automaticamente via hardware). L'interfaccia é analoga a tipo di interfacce introdotte nella sezione relativa al DMA.

L'interfaccia viene disabilitata ad effettuare trasferimenti in DMA, automaticamente (viene generato il segnale di fine operazione End Of Dma da parte del controllore DMA), a fine operazione.

La disabilitazione dell'interfaccia a effettuare richieste di interruzione a fine operazione deve essere effettuata esplicitamente a cura del processo esterno, e questo notifica anche all'interfaccia che la sua richiesta é stato servita.

Se M sono le interfacce che effettuano trasferimenti in DMA, si avranno due array di M descrittori DMA.

```
//io.cpp
...
typedef int* ind_l;
struct dma_reg
{
 ind_l iMAR;
 ind_l iTCR;
 ind_l iIOAR;
 ind_l iMODER;
};
struct dmadescrittore
{
 dma_reg ind_dma;
 interf_reg ind_interf;
 int mutex, sincr;
};
extern dmadescrittore dmadesi[M];
extern dmadescrittore dmadeso[M];
...
#io.s
...
.DATA
...
.GLOBAL _dmadesi
.GLOBAL _dmadeso
```

```
_dmadesi: .SPACE DMA_BYTE # sizeof(dmadescrittore) * M
_dmadeso: .SPACE DMA_BYTE
...
```

Nell'ipotesi che la i-ma intefaccia sia collegata al j-mo e k-mo canale del DMA l'inizializzazione del descrittore dmadesi i-mo, comporta

- inizializzazione del campo ind_dma, con gli indirizzi iMARj, iTCRj iIORAj, iMODERj;
- inizializzazione del campo ind_interf, con gli indirizzi iRBRi, iCTRIi
- inizializzazione del campo mutex, facendo uso della sem_ini(), ponendolo ad 1.
- inizializzazione del campo sincr, facendo uso della sem_ini(), ponendolo a 0.

Le operazione analoghe vanno fatte per il descrittore dmadeso i-mo. Ogni canale é connesso, via hardware, ad un'interfaccia, con un ben determinata operazione, quindi la predisposizione delle informazioni avviene in due tempi:

- in fase di inizializzazione del sistema, viene stabilito il valore di IOAR, e in parte del registro MODER(modalitá di funzionamento tranne il passaggio da slave a master).
- quando viene utilizzato il canale, tramite la dmastart().
  - scrive nei registri MAR e TCR,
  - modifica il bit per il passaggio da slave a master.

```
#io.s
...
.TEXT
...
.GLOBAL _dmago_in
_dmago_in:
 #...
 RET

.GLOBAL _dmahalt_in
_dmahalt_in:
 #...
 RET

 .GLOBAL _go_moder_in # riguarda DMA
_go_moder_in:
 #...
 RET

.GLOBAL _dmago_out
_dmago_in:
 #...
 RET

.GLOBAL _dmahalt_out
_dmahalt_in:
 #...
 RET

 .GLOBAL _go_moder_out # riguarda DMA
_go_moder_in:
 #...
 RET
```

```
#io.cpp
extern "C" void dmago_in(ind_b i_ctr);
extern "C" void dmahalt_in(ind_b i_ctr);
extern "C" void go_moder_in(ind_b i_moder);
extern "C" void dmago_out(ind_b i_ctr);
extern "C" void dmahalt_out(ind_b i_ctr);
extern "C" void go_moder_out(ind_b i_moder);
```

***Nota* 55** *Le operazioni in DMA coinvolgono un buffer di memoria, il cui indirizzo fisico costituisce il contenuto iniziale di MAR, deve risiedere nello spazio di indirizzamento comune, e non deve essere soggetto ne al meccanismo di rimpiazzamento ne a quello di cache.*

4.7.2 Lettura in DMA

La primitiva di lettura in DMA é struttura come segue (dmaleggi())

```
//utente.cpp
// spazio comune a tutti i processi
char buff_r[RR];
extern "C" void dmaleggi(int dmainterf, char vetti[], int quanti);
void proc_dma(int h) // corpo del processo utente
{
 int interf_dma, nn;
 // ...
 dmaleggi(interf_dma, buff_r, nn);
 // ...
}
//...
#utente.s
.TEXT
...
.GLOBAL _dmaleggi
_dmaleggi:
 INT $dma_tipo_r
rit_dr:
 RET
 ...
#io.s
...
.TEXT
.GLOBAL a_dmaleggi
a_dmaleggi:
 # routine INT $dma_tipo_r
 # STI per gate di tipo Interrupt
 # salvataggio dei registri
 #ricopiamento parametri
 CALL _c_dmaleggi
 #ripulitura della pila
 IRET
 ...
 // io.cpp
 ...
 void dmastart_in(dmadescrittore* p_dmadesi, char vetti[], int quanti);
 extern "C" void c_dmaleggi(int dmainterf, char vetti[], int quanti)
 {
  dmadescrittore* p_dmadesi;
  p_dmadesi = & dmadesi[dmainterf];
  sem_wati(p_dmadesi->mutex);
  dmastart_in(p_dmadesi, vetti, quanti);
  sem_wait(p_dmadesi->sincr);
  sem_signal(p_dmadesi->mutex);
 }
...
typedef int ind_fisico;
extern "C" void trasforma(char* a, ind_fisico& b);
void dmastart_in(dmadescrittore* p_dmadesi, char vetti[], int quanto)
{
 inid_fisico iff;
 trasforma(vetti, iff);
 *(p_dmadesi->ind_dma.iMAR) = iff;
 *(p_dmadesi->ind_dma.iTCR) = quanti -1 ;
 go_moder_in(p_dmadesi->ind_dma.iMODER);
```

```
 dmago_in(p_dmadesi->ind_interf.ctr_io.iCTRI);
}
#io.s
...
.TEXT
...
.GLOBAL _trasforma
_trasforma:
 INT $tipo_tra
rit_tra:
 RET
#sisitema.s
...
.TEXT
...
a_trasforma:
 #routine INT $tipo_tra
 #salvataggio dei registri
 #...
 #ripristino dei registri
 IRET
```

L'interfaccia genera una richiesta di interruzione, quando il controllore DMA notifica alla stessa che l'operazione di ingresso é globalmente terminata. Come effetto dell'interruzione viene eseguito, l'handler che forza l'esecuzione del processo esterno che

- disabilita l'interfaccia a inviare richieste di interruzione,
- sbloccare il processo utente (che si é bloccato sul semaforo di sincronizzazione) che ha richiesto l'operazione di ingresso.
- epilogo mediante la wfi (come avevamo analizzato nella sezione di processo esterno).

```
//io.cpp
...
void dmaesterni(int h)
{
 dmadescrittore* p_dmadesi;
 p_dmadesi = &dmadesi[h];
 while(1)
 {
  dmahalt_in(p_dmadesi->ind_interf.ctr_io.iCTRI);
  sem_signal(p_dmadesi->sincr);
  wfi();
 }
}
```

4.7.3 Scrittura in DMA

La primitiva di scrittura in DMA é struttura come segue (dmascrivi())

```
//utente.cpp
// spazio comune a tutti i processi
char buff_w[WW];
extern "C" void dmascrivi(int dmainterf, char vetto[], int quanti);
void proc_dma(int h) // corpo del processo utente
{
 int interf_dma, nn;
 // ...
 dmascrivi(interf_dma, buff_w, nn);
 // ...
}
//...
#utente.s
.TEXT
...
.GLOBAL _dmascrivi
_dmascrivi:
 INT $dma_tipo_w
rit_dw:
```

```
 RET
 ...
#io.s
...
.TEXT
.GLOBAL a_dmascrivi
a_dmascrivi:
 # routine INT $dma_tipo_w
 # STI per gate di tipo Interrupt
 # salvataggio dei registri
 #ricopiamento parametri
 CALL _c_dmascrivi
 #ripulitura della pila
 IRET
 ...
 // io.cpp
 ...
 void dmastart_out(dmadescrittore* p_dmadesi, char vetto[], int quanti);
 extern "C" void c_dmascrivi(int dmainterf, char vetto[], int quanti)
 {
  dmadescrittore* p_dmadeso;
  p_dmadeso = & dmadeso[dmainterf];
  sem_wati(p_dmadeso->mutex);
  dmastart_in(p_dmadeso, vetto, quanti);
  sem_wait(p_dmadeso->sincr);
  sem_signal(p_dmadeso->mutex);
 }
...
typedef int ind_fisico;
extern "C" void trasforma(char* a, ind_fisico& b);
void dmastart_out(dmadescrittore* p_dmadeso, char vetto[], int quanto)
{
 inid_fisico iff;
 trasforma(vetto, iff);
 *(p_dmadeso->ind_dma.iMAR) = iff;
 *(p_dmadeso->ind_dma.iTCR) = quanti -1 ;
 go_moder_in(p_dmadeso->ind_dma.iMODER);
 dmago_in(p_dmadeso->ind_interf.ctr_io.iCTRI);
}
#io.s
...
.TEXT
...
.GLOBAL _trasforma
_trasforma:
 INT $tipo_tra
rit_tra:
 RET
#sisitema.s
...
.TEXT
...
a_trasforma:
 #routine INT $tipo_tra
 #salvataggio dei registri
 #...
 #ripristino dei registri
 IRET
```

L'interfaccia genera una richiesta di interruzione, quando il controllore DMA notifica alla stessa che l'operazione di uscita é globalmente terminata. Come effetto dell'interruzione viene eseguito, l'handler che forza l'esecuzione del processo esterno che

- disabilita l'interfaccia a inviare richieste di interruzione,
- sbloccare il processo utente (che si é bloccato sul semaforo di sincronizzazione) che ha richiesto l'operazione di uscita.
- epilogo mediante la wfi (come avevamo analizzato nella sezione di processo esterno).

```
//io.cpp
...
void dmaesterno(int h)
{
 dmadescrittore* p_dmadeso;
 p_dmadeso = &dmadeso[h];
 while(1)
 {
  dmahalt_in(p_dmadeso->ind_interf.ctr_io.iCTRI);
  sem_signal(p_dmadeso->sincr);
  wfi();
 }
}
```

4.8 Multiprogrammazione e Personal Computer

Il personal computer é dotato di due controllori collegati in cascata (8259), di cui uno master e uno slave. Entrambi i controllori sono montati nel bus a 8 bit, con indirizzo base 0x0020 per master e 0x000A per lo slave. In dos sono inizializzati in modalitá *fully nested* con il tipo base 0x08 e 0x07.

4.8.1 Interruzione e la primitiva nwfi()

Il comando di EOI, valore 0x20, viene scritto nel registro OCW2 di ogni controllore. Occorre dunque definire una nuova wfi(), la nwfi, la quale invia il comando EOI come segue

- al controllore master, se la richiesta di interruzione proviene dal master stesso,
- al controllore master che allo slave se la richiesta proviene, tramite master, dallo slave.

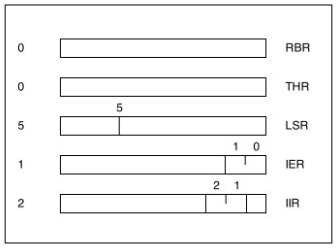
Occorre definire un parametro per la nwfi che discrimina fra le due situazione.

```
// io.cpp
...
enum contrllore{ master, slave};
extern "C" void nwfi(controllore);
...
int estern(int h) corpo comune
{
 ...
 while(1)
 {
 switch(h)
  {
   case ...: nwfi(master); break;
   case ...: nwfi(slave); break;
  }
 }
}
```

4.8.2 Gestione dell'interfaccia seriale

Il Personal Comuputer possiede, un interfaccia seriale, e software compatibile con l'integrato 8259(COMi). Questa viene montata nel bus a 8 bit, e puó essere configurata per avere un prefissato indirizzo base(COM1 ha indirizzo base 0x03F8, COM2 0x2F8H). Ogni COMi, possiede un solo piedino per effettuare richieste di interruzione di I/O, e collegato ad un ben preciso piedino del controllore di interruzioni. L'interfaccia si presenta al programmatore come segue

- *RBR* buffer di ingreso,



- *THR* (Trasmitter Holding Register) buffer di uscita.
- *LSR* (Line Status Register)
  - # contiene i due flag di buffer di ingresso pieno, e buffer di uscita vuoto,
  - # alcune condizioni di errore codificati come segue
    - * *bit 0* flag di buffer di ingresso pieno
    - * *bit 1* $= 1 \Rightarrow$ errore overrun
    - * *bit 2* $= 1 \Rightarrow$ errore paritá
    - * *bit 3* $= 1 \Rightarrow$ errore framing
    - * *bit 4* $= 1 \Rightarrow$ é stato ricevuto un break.
    - * *bit 5* flag di buffer di uscita.
- *IER* (Interrupt Enable Register) abilita l'interfaccia a generare richieste di interruzione
  - # *bit 0* $= 1 \Rightarrow$ abilita condizione di buffer pieno,
  - # *bit 1* $= 1 \Rightarrow$ abilita condizione di buffer vuoto.

Nell'ipotesi di effettuare operazioni di I/O che prevedono

- lettura di n caratteri o lettura fino a ritorno carrello,
- scrittura di n caratteri o scrittura fino al carattere nullo,
- verifica degli errori per le operazione di lettura.

si prevede un descrittore per la seriale che ha, in aggiunta rispetto al descrittore di I/O, due campi

- *funzione* che specifica il tipo di operazione.

  ***Nota* 56** *Ogni COMi ha un solo piedino per generare un interruzione, e alcuni registri sono comuni per le operazioni di lettura/scrittura, giustificando l'uso di un unico descrittore per entrambe le funzioni. L'interfaccia viene gestita in mutua esclusione con un unico semaforo. E ad ogni interruzione viene mandato uno specifico handler che forza l'esecuzione di un processo esterno di ingresso o di uscita a seconda del valore di IIR.*

- *stato* per memorizzare eventuali errori nel risultato.

Nell'ipotesi che le interfacce siano S avremo 2*S processi esterni, e i loro descrittori, di tipo proc_elem, sono puntati degli array **array_dessei[S] *array_desseo[S], che nell'implementazione del nucleo risultano sotto un unica variabile array * proc_esterni.*

```
// io.cpp
...
enum funz{input_n, input_ln, output_n, output_0};
struct interfse_reg
{
 ind_b iRBR;
 ind_b iTHR;
 ind_b iLSR;
 ind_b iIER;
 ind_b iIIR;
};
strcut des_se
{
 interfse_reg indreg;
 int mutex;
 int sincr;
 int cont;
 ind_b punt;
 funz funzione;
 char stato;
};
extern "C" des_se com[S];
```

Lo handler relativo a COMi é il seguente

```
#sistema.s
.TEXT
...
handler_COMi:
 CALL salva_stato
 CALL inspronti
 #leggere IRR_COMi esaminare la quantit\'a letta:
 #mettere in esecuzione il contenuto del puntatore array_dessei[i] o array_desso[i]
 #che individua un determinato processo esterno
 CALL carica_stato
 IRET
```

4.8.3 Operazione di lettura con la porta seriale

Supponiamo di eseguire un'operazione di lettura, annunciata prima, con le porte seriali utilizzando un buffer di memoria di 81 caratteri, utilizzando le seguente due routine.

```
// lettura di un numero prefissato di caratteri
void readse_n(int serial, char vetti[], int quanti, char& errore);
// lettura fino al ritorno carattere CR.
void readse_ln(int serial, char vetti[], int& quanti, char& errore);

//io.cpp
...
void startse_in(des_se* p_des, char vetti[], int quanti, funz op);
extern "C" void c_readse_n(int serial, char vetti[], int quanti, char& errore)
{
des_se* p_des;
p_des = &com[serial];
sem_wait(p_des->mutex);
startse_in(p_des, vetti, quanti, input_n);
sem_wait(p_des->sincr);
errore = p_des->stato;
sem_signal(p_des->mutex);
}
...
// io.cpp
void startse_in(des_se* p_des, char vetti[], int quanti, funz op)
{
 p_des->cont = quanti;
 p_des->punt = vetti;
 p_des->funzione = op;
 go_inputse(p_des->indreg.iIER);
}
```

Il corpo del processi esterni di ingresso relativo alle interfacce seriali.

```
// io.cpp
...
// corpo comune a tutti i processi esterni di ingresso per le porte seriali
void input_com(int h)
 {
  char c, s;
  bool fine;
  des_se* p_des;
  p_des = &com[h];
  while(1)
   {
    fine = false;
    halt_inputse(p_des->indreg.iIER);
    inputb(p_des->indreg.iRBR,c);
    inputb(p_des->indreg.iLSR,s);
    p_des->stato = s & 0x1E;
    if ( p_des->stato != 0) fine = turno;
    else
      {
       if(p_des->funzione = input_n)
        {
         *p_des->punt = c;
         p_des->punt++;
         p_des->cont--;
         if ( p_des->cont == 0) fine = true;
        }
       else
        {
          if (p_des->funzione == input_ln)
            if(c == "\r")
             {
              fine = true;
              p_des->cont = 80 - p_des->cont;
             }
            else
             {
              *p_des->punt = c;
              p_des->punt++;
              p_des->cont--;
              if(p_des->cont == 0)
               {
                fine = true;
                p_des->cont = 80;
               }
             }
        }
      }//
     if(fine == true)
      {
       *p_des->punt = 0;
       sem_signl(p_des->sincr);
      }
     else
      {
       go_inputse(p_des->indreg.iIER);
      }
     switch()
      {
       case ...: nwfi(master); break;
       case ...: nwfi(slave); break;
      }
   } // while
 } // input_com
```

4.8.4 Operazione di scrittura con la porta seriale

Supponiamo di eseguire un'operazione di scrittura, annunciata prima, con le porte seriali utilizzando un buffer di memoria di 81 caratteri, utilizzando le seguente due routine.

```
// scrittura di un numero prefissato di caratteri
void writese_n(int serial, char vetto[], int quanti, char& errore);
// scrittura fino al ritorno carattere CR.
void writese_ln(int serial, char vetto[], int& quanti, char& errore);
```

```
//io.cpp
...
void startse_out(des_se* p_des, char vetto[], int quanti, funz op);
extern "C" void c_writese_n(int serial, char vetto[], int quanti, char& errore)
{
des_se* p_des;
p_des = &com[serial];
sem_wait(p_des->mutex);
startse_in(p_des, vetti, quanti, output_n);
sem_wait(p_des->sincr);
errore = p_des->stato;
sem_signal(p_des->mutex);
}
...
// io.cpp
void startse_out(des_se* p_des, char vetto[], int quanti, funz op)
{
 p_des->cont = quanti;
 p_des->punt = vetto;
 p_des->funzione = op;
 go_outputse(p_des->indreg.iIER);
}
```

Il corpo del processi esterni di uscita relativo alle interfacce seriali.

```
// io.cpp
...
// corpo comune a tutti i processi esterni di uscita per le porte seriali
void output_com(int h)
 {
  char c;
  bool fine;
  des_se* p_des;
  p_des = &com[h];
  while(1)
   {
    fine = false;
    if(p_des->funzione = output_n)
      {
        p_des->cont--;
        if ( p_des->cont == 0)
         {
          fine = true;
          halt_outputse(p_des->indreg.iIER);
         }
        c = *p_des->punt;
        outputb(c, p_des->indreg.iTHR);
        p_des->punt++;
       }
     else
       {
        if ( p_des->funzione == output_0)
         {
          c = *p_des->punt;
          if ( c == 0)
           {
            fine = true;
            halt_outputse(p_des->indreg.iIER);
           }
         else
           {
             outputb(c, p_des->indreg.iTHR);
             p_des->cont++;
             p_des->punt++;
           }
         }
       }
      if(fine == true)
        sem_signl(p_des->sincr);
      switch()
       {
        case ...: nwfi(master); break;
        case ...: nwfi(slave); break;
       }
   } // while
 } // input_com
...
```

5 Segmentazione e Multiprogrammazione

- Segmentazione
- Multiprogrammazione e protezione in ambiente segmentato (Parte mancante)

5.1 Memoria Segmentazione

Avevamo giá analizzato un meccanismo di virtualizzazione della memoria principale. Una seconda tecnica di virtualizzazione della memoria, é la segmentazione.

La tecnica della segmentazione, comporta che lo spazio logico abbia un'organizzazione in segmenti.

- uno o piú segmenti dati,
- un segmento per la pila,
- un segmento codice,
- uno per ogni sottoprogramma.

Quest'organizzazione ricorda molto, molto l'organizzazione modulare di un'architettura software e per poter essere realizzata, necessita che lo *spazio logico* si *bidimensionale*. L'indirizzo logico viene suddiviso in due componenti

- **selettore**. Il *selettore* individua un segmento,
- **offset**. L'*offset* seleziona una locazione all'interno del segmento riferito.

Sia a il numero di bit per individuare un segmento, e b sia il numero di bit per individuare una locazione nel segmento, si ha che

- il numero massimo di segmenti sia pari a 2^a,
- la dimensione massima di un segmento sia pari a 2^b.

In analogia con la virtualizzazione paginata, un programma utilizza solo alcuni segmenti, e diversamente dalla virtualizzazione paginata la dimensione effettiva di un segmento puó essere inferiore o uguale alla dimensione massima.

***Nota* 57** *Il calcolatore ha un spazio di indirizzamento lineare, quindi un indirizzo logico necessita di una traduzione ad un indirizzo linerare.*

La traduzione avviene a livello hardware, per via della MMU, ogni volta che viene generato un indirizzo di memoria. L'indirizzo lineare puó costituire l'indirizzo fisico, ovvero soggetto ad un'ulteriore traduzione qualora sia abilitata la paginazione.

1. *indirizzo logico* $\rightsquigarrow$ *indirizzo lineare* $\rightsquigarrow$ *indirizzo fisico.*
2. *indirizzo logico* $\rightsquigarrow$ *indirizzo lineare* $\rightsquigarrow$ *indirizzo virtuale* $\rightsquigarrow$ *indirizzo fisico.*

In ambiente segmentato sono si hanno le seguenti visione della memoria

- *memoria segmentata (o memoria logica)*, occupa un sotto insieme dello spazio logico, puó risiedere nella memoria lineare, oppure, quando la sua dimensione effettiva é maggiore di quella lineare, viene virtualizzata appoggiandosi sulla memoria di massa.
- *memoria lineare* occupa un sottoinsieme della memoria lineare, puó risiedere tutta nella memoria fisica, oppure essere virtualizzata appoggiandosi sulla memoria di massa.

5.1.1 Segmenti nel processore PC

Nel processore PC

- il *selettore* di un indirizzo logico é rappresentato su 16 bit
  - 14 MSB individuano il segmento, e lo spazio logico é costituito da 2^{14} segmenti.
  - 2 bit rimanenti utilizzati nel meccanismo di protezione.

  I segmenti possono avere
  - granularitá di byte
    - la lunghezza effettiva é multiplo di un byte, e la lunghezza massima é di 1 Mbyte.
  - granularitá di pagina
    - la lunghezza effettiva é multipla di 4 Kbyte (una pagina), e la lunghezza massima é di 4 Gbyte (1 Mpagine).
- *l'offset* di un indirizzo logico, é rappresentato su 32 bit. Per la grunularitá di pagina la dimensione massima di un segmento é di 2^{32} byte, pertanto lo spazio logico ha una capacitá pari a 2^{46} byte.

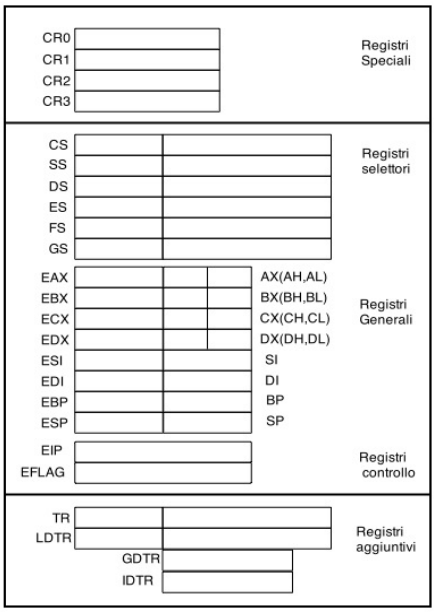
5.1.2 Registri e istruzioni in ambiente segmentato

Il processore PC appare al programmatore come in figure I registri selettore presentano una parte visibile e una parte nascosta gestita via hardware. La parte visibile dei registri selettore contiene i selettori dei segmenti correnti

- *CS* (Code Segment) contiene il selettore del segmento codice.
- *SS* (Stack Segment) contiene il selettore del segmento pila.
- *DS*, *ES*, *FS*, *GS* contengono i segmenti dati. Alcuni possono avere lo stesso valore.

Si ha che

- l'istruzione successiva da eseguire é data dal valore CS:EIP (selettore : offset), pertanto durante la fase di chiamata delle istruzione il valore di CS rimane costante, mentre viene aggiornato il valore di EIP.
- Il puntatore di pila é determinato dal valore SS:ESP, e le istruzioni di PUSH e POP aggiornano solo ESP.



- per un istruzione operativa che prevede un operando in memoria, si deve, in primo luogo specificare il selettore. L'istruzione operativa specifica la componente l'offset.
- Per un istruzione di salto
  - *intra-segment* (stesso segmento) l'istruzione specifica solo l'offset
  - *inter-segment* l'istruzione specifica sia il selettore che l'offset.

Il livello di privilegio corrente del processore (CPL) é dato dal valore dei 2 bit del registro CS. I livelli di privilegio possibili sono quattro.

5.1.3 Traduzione degli indirizzi

La traduzione di indirizzo da logico a lineare, se la paginazione non é attivata, restituisce un indirizzo fisico. Per effettuare la traduzione, si adopera una tabella di corrispondenza DT (*Descriptor Table*) che risiede nella memoria lineare, e il suo indirizzo (lineare) é contenuto nel registro DTR del processore.

- la tabella DT contiene i *descrittori* dei segmenti, e la componente selettore é l'indice di uno dei descrittori.
- Ogni descrittore contiene
  - un byte di access
  - l'*indirizzo base* del segmento in memoria lineare.
  - il limite nel segmento stesso (offset dell'ultima locazione).

La traduzione ha luogo solo se *componente offset* $\leq$ al limite

- *hit*: l'indirizzo lineare si ottiene come somma dell'indirizzo base e e la componente offset.
- *miss*: viene generata un'eccezione.

5.1.4 Traduzione degli indirizzi nel processore PC

La traduzione di indirizzo nel processore PC viene effettuata mediante l'ausilio di due tabelle di descrittore di segmento

- *GDT* (Global Descriptor Table) la medesima per tutti i processi. La GDT é situata nella memoria lineare e il registro GDTR contiene la base e il limite della stessa.
- *LDT* (Local Descriptor Table) si riferisce ad un singolo processo. La LDT é situata nella memoria logica e il registro LDTR contiene il selettore e il descrittore(base e limite e byte di accesso) della stessa.

Il registro CS viene pensato suddiviso in tre parti

- *indice*: (i 13 piú significativi), individua il descrittore del segmento all'interno della tabella GDT o LDT. Tale tabelle hanno
  - al piú 2^{13} descrittori.
  - ogni descrittore é costituito da 8 byte. Il byte meno significativo ha indirizzo relativo *indice * 8*.
  - dimensione massima di 64 kbyte.
- *CPL*: 2 bit (n.0, n.1) e rappresenta il livello di privilegio attuale del processore.
- *bit T*: (Table indicator, bit n.2) in base al valore si determina quale tabella coinvolgere nella traduzione di indirizzo.
  - T = 0 ⇒ GDT.
  - T = 1 ⇒ LDT.

Descrittore di segmento

Il descrittore di semento é composto da

- campo *base* (32 bit) che specifica l'indirizzo a partire dal quale é allocato il segmento in memoria lineare. Se un segmento é allineato alla pagina, i 12 bit meno significativi dell'indirizzo base valgono 0.
- campo *limite*: viene espresso in forma compatta su 20 bit. Specifica l'offset dell'ultimo byte o dell'ultima pagina. Viene espanso su 32 bit prima di essere comparato con l'offset. Se la granularitá di segmento é
  - di byte ⇒ i 12 bit piú significativi vengono posti a 0.
  - di pagina ⇒ i 12 bit meno significativi vengono posti a 1.
- bit *G* : indica la granularitá del segmento.
- campo *accesso*: contiene
  - il bit P ad indicare se il segmento é presente in memoria lineare.
  - il campo TP che indica il tipo di segmento.

– il campo DPL rappresenta il livello di privilegio del descrittore.

I segmenti piú utilizzati sono il segmento codice e segmento dati (solo leggibili o anche scrivibili).

5.1.5 Struttura dei registri selettore

Per ridurre l'accesso alle tabelle di corrispondenza GDT o LDT, i descrittore dei segmenti correnti vengono trasferiti nel processore. Ogni registro selettore predispone di una parte nascosta al programmatore, e accessibile via hardware, per memorizzare il descrittore del segmento di cui il selettore é l'indice.

***Nota* 58** *L'insieme delle parti nascoste dei registri selettore, costituiscono la memoria cache dei descrittori*

CS	Selettore	Base	Limite	G accesso
SS	Selettore	Base	Limite	G accesso
DS	Selettore	Base	Limite	G accesso
ES	Selettore	Base	Limite	G accesso
FS	Selettore	Base	Limite	G accesso
GS	Selettore	Base	Limite	G accesso

Ogni volta che viene caricato il selettore in un registro selettore, avviene un accesso via hardware alla tabella dei GDT o LDT, ricopiando il descrittore nella parte nascosta del registro selettore coinvolto. Se avviene un accesso ad un segmento il cui selettore si trova in un registro selettore, l MMU ricava l'indirizzo fisico sommando semplicemente la base del segmento, contenuta nella parte nascosta del registro selettore, con la componente offset dell'indirizzo logico. Quindi la traduzione avviene in modo efficiente.

5.1.6 Selettore nullo

Alcuni registri selettori, quando non sono utili vanno caricati con il cosiddetto *selettore nullo*(i 14 bit piú significativi del selettore posti a 0). Quest'operazione si rende necessaria per evitare i seguenti casi

- Se il selettore non utilizzato contenesse un selettore non significativo, poterebbe essere generata un'eccezione. Questo pu'øaccadere in fase di aggiornamento della cache dei descrittori.
- Situazione analoga alla precedente, puó presentarsi nel caso in cui un sottoprogramma salva e poi ripristina il contenuto dei registri selettore.

Caricando il selettore nullo nei registri non utilizzati, questi problemi non accadono, visto che il selettore nullo viene gestito in questo modo

- é consentito il suo caricamento in un registro selettore
- una volta caricato non avviene l'aggiornamento dell'entrata cache relativa al registro coinvolto.
- non é consentito il suo utilizzo per riferire una locazione di memoria.

Ne segue che nelle tabelle GDT e LDT un indice 0 non individua alcun descrittore.

5.1.7 Trasferimento di segmenti

In alcuni casi, la memoria lineare non puó contenere tutta la memoria logica, , e questo comporta che solo una parte dei segmenti risiedono nella memoria lineare. Per analogia con la memoria virtuale paginata, la memoria logica é una memoria virtuale segmentata.

In termini di spazio di indirizzamento la capacitá della memoria logica é di gran lunga piú ampia della memoria lineare

Viene realizzato il meccanismo di trasferimento dinamico dei segmenti tra la memoria di massa e la memoria lineare, in modo che ad ogni istante, i segmenti attualmente riferiti siano presenti in memoria lineare. Il trasferimento dinamico é analogo a quello trattato nella paginazione.

- nel byte di accesso di un descrittore di segmenti, il bit P indica la presenza del segmento nella memoria lineare
  - P = 1 segmento presente in memoria lineare,
  - P = 0 segmento assente,
- viene generata l'eccezione di *segment-fault* quando nell'eseguire un'istruzione che carica un registro selettore, trasferendo nella parte nascosta il descrittore, rileva che P = 0.

Un *segment-fautl* si puó verificare solo in fase di esecuzione, quando viene scritto un operando in DS, ES, FS, GS, SS da un'istruzione operativa o in CS da un'istruzione di controllo

In caso di eccezione va in esecuzione un'opportuna routine di trasferimento

- tramite la pila dove é stato memorizzato l'indirizzo logico dell'istruzione che ha provocato fault, determina il registro selettore RS relativo al segmento non presente, e attraverso il descrittore contenuto in GDT o LDT determina, la lunghezza del segmento.
- cerca lo spazio in memoria lineare per il nuovo segmento. Puó essere necessario un rimpiazzamento ricopiando uno o piú segmenti in memoria di massa.
- trasferisce da memoria di massa a memoria lineare, una copia del nuovo segmento.
- aggiorna (in GDT e LDT) il descrittore del nuovo segmento, modificando il campo base e ponendo a 1 il bit P.

Quando termina la routine, viene rieseguita l'istruzione che ha prodotto fault. I segmenti hanno in genere lunghezza diversa, il meccanismo di rimpiazzamento porta ad una *frammentazione* della memoria lineare, e in alcuni casi si richiede una ricompattazione dei segmenti presenti, aggiornando gli indirizzi base nei descrittori di segmento.

Nell'operazione di rimpiazzamento vanno ricopiati in memoria solo i segmenti che sono stati modificati. Per poter implementare la politica di rimpiazzamento viene gestito il bit A (Accessed), il quale viene posto, via hardware, ad 1 ogni volta che viene eseguita un'istruzione che carica un registro selettore, con il trasferimento nella parte nascosta del registro.

Ad intervalli regolari va in esecuzione la routine di timer, che effettua le opportune statistiche ed azzera il bit A.

***Nota* 59** *I segmenti il cui selettore é contenuto in un registro selettore, quando viene riferito , non determina alcuna modifica del bit A nel descrittore memorizzato nella tabella GDT o LDT, in quanto non vi é alcun caricamento del descrittore medesimo. Pertanto la routine*

dopo aver posto a 0 tutti i bit A, deve porre nuovamente a 1 i bit A di qui descrittori giá contenuti nelle parte nascoste dei registri selettore, caricando nuovamente in essi il loro valore, riprestinando il vecchio valore di CS con l'istruzione IRET.

5.1.8 Attivazione del modo protetto

Il processore PC, al reset, parte nel modo reale. L'indirizzamento é bidimensionale, ma con segmenti aventi lunghezza massima pari a 64 Kbyte, realizzando la tabella di corrispondenza via hardware, con indirizzo base dato da *selettore**16 e limite 0xFFFF. La paginazione é disattivata. Dopo le operazioni di inizializzazione il processore passa ad operare nel modo protetto, agendo sul bit n.0 del bit CR0.

- In fase di inizializzazione vengono predisposte le tabelle GDT e IDT, con i relativi registri. In secondo luogo viene attivata la paginazione ponendo ad 1 il bit n.31 del registro CR0. Nel passare dal modo reale a quello protetto non si ha discontinuitá nel riferire i segmenti, in quanto la MMU utilizza solo informazioni presenti nella parte nascosta dei registri selettore.
- Si effettua poi un salto inter-segment passando ad eseguire un programma che utilizza le potenzialitá della segmentazione e paginazione previste dal modo protetto. L'istruzione di salto comporta anche lo svuotamento della coda di prefech interna al processore, in modo che le successive istruzioni vengono prelevati dalla memoria con gli opportuni di controllo.
- Vengono inizializzati i registri selettori, e quelli non utilizzati con il selettore nullo.
- Se intende operare in ambiente multiprogrammato si inizializza i registri TR e LDTR, e predisporre i segmenti di stato (TSS) e le tabelle dei segmenti private di ogni processo (LDT).

Modello Flat

Il modo protetto, nel caso piú semplice, utilizza il cosiddetto modello di memoria flat.

- é costituito da tre segmenti (il cui selettore ha bit T = 0), avente granularitá di pagina.
  - uno per il code
  - uno per per i dati
  - uno per la pila.

  tutti con indirizzo base 0x00000000 e limite 0xFFFFF. Ne segue che la GDT é costituita da tre descrittori che individuano i segmenti sovrapposti in memoria lineare. I strumenti di sviluppo utilizzabili in ambiente flat, allocano codice, pila e dati in zone diverse nel corrispondente segmento. Cosí facendo si fa apparire una memoria lineare.

***Nota* 60** *La tabella GDT che la IDT non sono segmenti di memoria pertanto non possono essere modificate. Nelle applicazione in cui si rende necessario modificare la GDT, si deve prevedere un segmento nello spazio logico comune a tutti i processi, pertanto il descrittore della tabella GDT deve risiede nella tabella stessa e inserito prima dell'attivazione del modo protetto, con gli stessi valori di base e limite contenuti nel GDTR. Lo stesso vale per la IDT.*

5.1.9 Meccanismo di interruzione

Nel modo protetto, il registro IDTR del processore PC, contiene la base e limite della IDT. Questa contiene al piú 256 gate, relativi a tipi di interruzioni. Il tipo del gate (campo TP) puó essere

- Interrupt/Trap ⇒ il gate specifica l'indirizzo logico (selettore e offset) della routine di interruzione.
- Task ⇒ il gate specifica il selettore del segmento TSS del nuovo processo.

Quando avviene un'interruzione il processore puó generare un'eccezione se

- se viene superato il limite di IDT.
- se il gate non é presente (tipo di interruzione non implementato).

Se il gate é di tipo Interrupt/Trap e non si ha variazione di livello di privilegio,

- vengono immesse in pila EFLAG, CS e EIP.
- Il selettore e l'offset prelevati dal gate vengono trasferiti nei registri CS e EIP.
- vengon azzerati i bit NT TF ed eventualmente IF del registro EFLAG.

L'operazione di ripristino avviene quando la routine termina e viene eseguita la IRET.

6 Conclusione

NOTHING IS IMPOSSIBLE! LaTeX

La volontá in determinati casi é piú forte di ogni pregiudizio, e aiuta a vincere le proprie fobie(M.M).

[]

A Appendice

A.1 Gestione della Memoria Virtuale Paginata

```
//sistema.cpp

////////////////////////////////////////////////////////////////////////
// INTRODUZIONE GENERALE ALLA MEMORIA VIRTUALE                        //
////////////////////////////////////////////////////////////////////////
//
// SPAZIO VIRTUALE DI UN PROCESSO:
// Ogni processo possiede un proprio spazio di indirizzamento, suddiviso in
// cinque parti:
// - sistema/condivisa: contiene l'immagine di tutta la memoria fisica
// installata nel sistema
// - sistema/privata: contiene la pila sistema del processo
// - io/condiviso: contiene l'immagine del modulo io
// - utente/condivisa: contiene l'immagine dei corpi di tutti i processi, dei
// dati globali e dello heap
// - utente/privata: contiene la pila utente del processo
// Le parti sistema e io non sono gestite tramite memoria virtuale (i bit P
// sono permanentemente a 1 o a 0) e sono accessibili solo da livello di
// privilegio sistema. Le parti utente sono gestite tramite memoria virtuale e
// sono accessibili da livello utente.
// Le parti condivise sono comuni a tutti i processi. La condivisione viene
// realizzata esclusivamente condividendo le tabelle delle pagine: i direttori
// di tutti i processi puntano alle stesse tabelle nelle entrate relative alla
// parte sistema/condivisa, io/condivisa e utente/condivisa.  In questo modo,
// ogni pagina (sia essa appartenende ad uno spazio privato che ad uno spazio
// condiviso) e' puntata sempre da una sola tabella. Cio' semplifica la
// gestione della memoria virtuale: per rimuovere o aggiungere una pagina e'
// necessario modificare una sola entrata (altrimenti, per rimuovere o
// aggiungere una pagina di uno spazio condiviso, sarebbe necessario modificare
// le entrate relative alla pagina in tutte le tabelle, di tutti i processi,
// che la puntano). Inoltre, le tabelle condivise sono sempre preallocate e non
// rimpiazzabili (altrimenti, per aggiungere o rimuovere una tabella condivisa,
// sarebbe necessario modificare l'entrata relativa nel direttorio di ogni
// processo).
//
// MEMORIA FISICA:
// La memoria fisica e' suddivisa in due parti: la prima parte contiene il
// codice e le strutture dati, statiche e dinamiche, del nucleo, mentre la
// seconda parte e' suddivisa in pagine fisiche, destinate a contenere i
// direttori, le tabelle delle pagine, le pagine del modulo io e le pagine
// virtuali dei processi.  Poiche' l'intera memoria fisica (sia la prima che la
// seconda parte) e' mappata nello spazio sistema/condiviso di ogni processo,
// il nucleo (sia il codice che le strutture dati) e' mappato, agli stessi
// indirizzi, nello spazio virtuale di ogni processo.  In questo modo, ogni
// volta che un processo passa ad eseguire una routine di nucleo  (per effetto
// di una interruzione software, esterna o di una eccezione), la
// routine puo' continuare ad utilizzare il direttorio del processo e avere
// accesso al proprio codice e alle proprie strutture dati.
// La seconda parte della memoria fisica e' gestita tramite una struttra dati
// del nucleo, allocata dinamicamente in fase di inizializzazione del sistema,
// contenente un descrittore per ogni pagina fisica. Per le pagine fisiche
// occupate (da un direttorio, una tabella o una pagina virtuale) e
// rimpiazzabili, tale descrittore contiene il contatore per le statistiche di
// accesso (LRU o LFU) e il numero del blocco, in memoria di massa, su cui
// ricopiare la pagina, in caso di rimpiazzamento. Per motivi di efficienza, i
// descrittori delle pagine occupate e rimpiazzabili contengono anche le
// informazioni utili a recuperare velocemente l'entrata del direttorio o della
// tabella delle pagine che puntano alla tabella o alla pagina descritta (senza
// questa informazione, per rimpiazzare il contenuto di una pagina fisica,
// occorrerebbe scorrere le entrate di tutti i direttori o di tutte le tabelle,
// alla ricerca delle entrate che puntano a quella pagina fisica).
//
// MEMORIA DI MASSA:
// La memoria di massa (swap), che fa da supporto alla memoria virtuale, e'
// organizzata in una serie di blocchi, numerati a partire da 0. I primi
// blocchi contengono delle strutture dati necessarie alla gestione dello swap
// stesso, mentre i rimanenti blocchi contengono direttori, tabelle e pagine
// virtuali.
// Se una pagina virtuale non e' presente in memoria fisica, il proprio
```

```
// descrittore di pagina (all'interno della corrispondente tabella delle
// pagine) contiene il numero del blocco della pagina nello swap (analogamente
// per le tabelle non presenti) e le informazioni necessarie alla corretta
// creazione del descrittore di pagina, qualora la pagina dovesse essere
// caricata in memoria fisica (in particolare, i valori da assegnare ai bit US,
// RW, PCD e PWT). Quando una pagina V, non presente, viene resa presente,
// caricandola in una pagina fisica F, l'informazione del blocco associato alla
// pagina viene ricopiata nel descrittore della pagina fisica F (altrimenti,
// poiche' nel descrittore verra' scritto il byte di accesso e l'indirizzo di
// F, tale informazione verrebbe persa). Se la pagina V dovesse essere
// successivamente rimossa dalla memoria fisica, l'informazione relativa al
// blocco verra' nuovamente copiata dal descrittore della pagina fisica F al
// descrittore di pagina di V, all'interno della propria tabella.
//
// CREAZIONE DELLO SPAZIO VIRTUALE DI UN PROCESSO:
// Inizialmente, lo swap contiene esclusivamente l'immagine della parte
// utente/condivisa e io/condivisa, uguale per tutti i processi. Lo swap
// contiene, infatti, un solo direttorio, di cui sono significative solo le
// entrate relative a tali parti, le corrispondenti tabelle e le pagine puntate
// da tali tabelle.
//
// All'avvio del sistema, viene creato un direttorio principale, nel seguente
// modo:
// - le entrate corrispondenti allo spazio sistema/condiviso, vengono fatte
// puntare a tabelle, opportunamente create, che mappano tutta la memoria
// fisica in memoria virtuale
// - le entrate corrispondenti agli spazi io/condiviso e utente/condiviso,
// vengono copiate dalle corrispondenti entrate che si trovano nel direttorio
// nello swap. Le tabelle puntate da tali entrate vengono tutte caricate in
// memoria fisica, cosi' come le pagine puntate dalle tabelle relative allo
// spazio io/condiviso
// - le rimanenti entrate sono non significative
//
// Ogni volta che viene creato un nuovo processo, il suo direttorio viene prima
// copiato dal direttorio principale (in questo modo, il nuovo processo
// condividera' con tutti gli altri processi le tabelle, e quindi le pagine,
// degli spazio sistema, io e utente condivisi). Gli spazi privati (sistema e
// utente), che sono inizialmente vuoti (fatta eccezione per alcune parole
// lunghe) vengono creati dinamicamente, allocando nuove tabelle e lo spazio
// nello swap per le corrispondenti pagine. Se la creazione degli spazio
// privati di un processo non fosse effettuata durante la creazione del
// processo stesso, lo swap dovrebbe essere preparato conoscendo a priori il
// numero di processi che verranno creati al momento dell'esecuzione di main.
//


//////////////////////////////////////////////////////////////////////
// PAGINAZIONE                                                      //
//////////////////////////////////////////////////////////////////////
// Vengono qui definite le strutture dati utilizzate per la gestione della
// paginazione. Alcune (descrittore_pagina, descrittore_tabella, direttorio,
// tabella_pagina) hanno una struttura che e' dettata dall'hardware.
//
// In questa implementazione, viene utilizzata la seguente ottimizzazione:
// una pagina virtuale non presente, il cui descrittore contiene il campo
// address pari a 0, non possiede inizialmente un blocco in memoria di massa.
// Se, e quando, tale pagina verra' realmente acceduta, un nuovo blocco verra'
// allocato in quel momento e la pagina riempita con zeri. Tale ottimizzazione
// si rende necessaria perche' molte pagine sono potenzialmente utilizzabili
// dal processo, ma non sono realmente utilizzate quasi mai (ad esempio, la
// maggior parte delle pagine che compongono lo heap e lo stack). Senza questa
// ottimizzazione, bisognerebbe prevedere uno dispositivo di swap molto grande
// (ad es., 1 Giga Byte di spazio privato per ogni processo, piu' 1 Giga Byte
// di spazio utente condiviso) oppure limitare ulteriormente la dimensione
// massima dello heap o dello stack. Tale ottimizzazione e' analoga a quella
// che si trova nei sistemi con file system, in cui i dati globali
// inizializzati a zero non sono realmente memorizzati nel file eseguibile, ma
// il file eseguibile ne specifica solo la dimensione totale.
//
// Se il descrittore di una tabella delle pagine possiede il campo address pari
// a 0, e' come se tutte le pagine puntate dalla tabella avessero il campo
// address pari a 0 (quindi, la tabella stessa verra' creata dinamicamente, se
// necessario)

struct descrittore_pagina {
        // byte di accesso
```

```
        unsigned int P:                 1;        // bit di presenza
        unsigned int RW:        1;        // Read/Write
        unsigned int US:        1;        // User/Supervisor
        unsigned int PWT:        1;        // Page Write Through
        unsigned int PCD:        1;        // Page Cache Disable
        unsigned int A:                 1;        // Accessed
        unsigned int D:                 1;        // Dirty
        unsigned int pgsz:        1;        // non visto a lezione
        // fine byte di accesso

        unsigned int global:        1;        // non visto a lezione
        unsigned int avail:        3;        // non usati

        unsigned int address:        20;        // indirizzo fisico/blocco
};

// il descrittore di tabella delle pagine e' identico al descrittore di pagina
typedef descrittore_pagina descrittore_tabella;

// Un direttorio e' un vettore di 1024 descrittori di tabella
// NOTA: lo racchiudiamo in una struttura in modo che "direttorio" sia un tipo
struct direttorio {
        descrittore_tabella entrate[1024];
};

// Una tabella delle pagine e' un vettore di 1024 descrittori di pagina
struct tabella_pagine {
        descrittore_pagina  entrate[1024];
};

// Una pagina virtuale la vediamo come una sequenza di 4096 byte, o 1024 parole
// lunghe (utile nell'inizializzazione delle pile)
struct pagina {
        union {
                unsigned char byte[SIZE_PAGINA];
                unsigned int  parole_lunghe[SIZE_PAGINA / sizeof(unsigned int)];
        };
};

// una pagina fisica occupata puo' contenere un direttorio, una tabella delle
// pagine o una pagina virtuale.  Definiamo allora pagina_fisica come una
// unione di questi tre tipi
union pagina_fisica {
        direttorio        dir;
        tabella_pagine        tab;
        pagina                pag;
};

// dato un puntatore a un direttorio, a una tabella delle pagine o a una pagina
// virtuale, possiamo aver bisogno di un puntatore alla pagina fisica che li
// contiene (poiche' gli indirizzi di partenza coincidono, si tratta solo di
// eseguire una conversione di tipo)
inline pagina_fisica* pfis(direttorio* pdir)
{
        return reinterpret_cast<pagina_fisica*>(pdir);
}

inline pagina_fisica* pfis(tabella_pagine* ptab)
{
        return reinterpret_cast<pagina_fisica*>(ptab);
}

inline pagina_fisica* pfis(pagina* ppag)
{
        return reinterpret_cast<pagina_fisica*>(ppag);
}

// dato un indirizzo virtuale, ne restituisce l'indice nel direttorio
// (i dieci bit piu' significativi dell'indirizzo)
short indice_direttorio(void* indirizzo)
{
        return (uint(indirizzo) & 0xffc00000) >> 22;
}

// dato un indirizzo virtuale, ne restituisce l'indice nella tabella delle
// pagine (i bit 12:21 dell'indirizzo)
```

```
short indice_tabella(void* indirizzo)
{
        return (uint(indirizzo) & 0x003ff000) >> 12;
}

// dato un puntatore ad un descrittore di tabella, restituisce
// un puntatore alla tabella puntata
// (campo address del descrittore esteso a 32 bit aggiungendo 12 bit a 0)
tabella_pagine* tabella_puntata(descrittore_tabella* pdes_tab)
{
        return reinterpret_cast<tabella_pagine*>(pdes_tab->address << 12);
}

// dato un puntatore ad un descrittore di pagina, restituisce
// un puntatore alla pagina puntata
// (campo address del descrittore esteso a 32 bit aggiungendo 12 bit a 0)
pagina* pagina_puntata(descrittore_pagina* pdes_pag)
{
        return reinterpret_cast<pagina*>(pdes_pag->address << 12);
}

// il direttorio principale contiene i puntatori a tutte le tabelle condivise
// (degli spazi sistema, io e utente). Viene usato  nella fase inziale, quando
// ancora non e' stato creato alcun processo e dal processo main.  Inoltre,
// ogni volta che viene creato un nuovo processo, il direttorio del processo
// viene inzialmente copiato dal direttorio principale (in modo che il nuovo
// processo condivida tutte le tabelle condivise)
direttorio* direttorio_principale;

// carica un nuovo valore in cr3 [vedi sistema.S]
extern "C" void carica_cr3(direttorio* dir);

// restituisce il valore corrente di cr3 [vedi sistema.S]
extern "C" direttorio* leggi_cr3();

// attiva la paginazione [vedi sistema.S]
extern "C" void attiva_paginazione();

////////////////////////////////////////////////////////////////////////
// GESTIONE DELLE PAGINE FISICHE                                      //
// ////////////////////////////////////////////////////////////////////
// La maggior parte della memoria principale del sistema viene utilizzata per
// implementare le "pagine fisiche".  Le pagine fisiche vengono usate per
// contenere direttori, tabelle delle pagine o pagine virtuali.  Per ogni
// pagina fisica, esiste un "descrittore di pagina fisica", che contiene:
// - un campo "contenuto", che puo'assumere uno dei valori sotto elencati
// - il campo contatore degli accessi (per il rimpiazzamento LRU o LFU)
// - il campo "blocco", che contiene il numero del blocco della pagina o
//   tabella nello swap
// I descrittori di pagina fisica sono organizzati in un array, in modo che il
// descrittore in posizione "i" descrive la pagina "i-esima" (considerando il
// normale ordine delle pagine fisiche in memoria fisica)
//
//se la pagina e' libera, il descrittore di pagina fisica contiene anche
//l'indirizzo del descrittore della prossima pagina fisica libera (0 se non ve
//ne sono altre)
//
//se la pagina contiene una tabella, il destrittore di pagina fisica contiene
//anche il puntatore al direttorio che mappa la tabella, l'indice del
//descrittore della tabella all'interno del direttorio e un campo "quante", che
//contiene il numero di entrate con P != 0 all'interno della tabella
//(informazioni utili per il rimpiazzamento della tabella)
//
//se la pagina contiene una pagina virtuale, il descrittore di pagina fisica
//contiene un puntatore all'(unica) tabella che mappa la pagina e l'indirizzo
//virtuale della pagina (informazioni utili per il rimpiazzamento della pagina)

// possibili contenuti delle pagine fisiche
enum cont_pf {
        LIBERA,                         // pagina allocabile
        DIRETTORIO,                  // direttorio delle tabelle
        TABELLA_PRIVATA,          // tabella appartenente a un solo processo
        TABELLA_CONDIVISA,          // tabella appartenente a piu' processi
                                  // (non rimpiazzabile, ma che punta a pagine
                                  // rimpiazzabili)
        TABELLA_RESIDENTE,          // tabella non rimpiazzabile, che punta a
```

```
                            // a pagine non rimpiazzabili
        PAGINA,                     // pagina rimpiazzabile
        PAGINA_RESIDENTE };        // pagina non rimpiazzabile

// descrittore di pagina fisica
struct des_pf {
        cont_pf               contenuto;        // uno dei valori precedenti
        unsigned int        contatore;        // contatore per le statistiche LRU/LFU
        union {
                struct { // informazioni relative a una pagina
                        unsigned int        blocco;               // blocco in memoria di massa
                        tabella_pagine*        tabella;        // tabella che punta alla pagina
                        void*                ind_virtuale;   // indirizzo virtuale della pagina
                } pag;
                struct { // informazioni relative a una tabella
                        unsigned int        blocco;               // blocco in memoria di massa
                        direttorio*     dir;                // direttorio che punta alla tabella
                        short                indice;                // indice della tab nel direttorio
                        short                quante;                // numero di pagine puntate
                } tab;
                struct  { // informazioni relative a una pagina libera
                        des_pf*                prossima_libera;
                } avl;
                // non ci sono informazioni aggiuntive per una pagina
                // contenente un direttorio
        };
};


// puntatore al primo descrittore di pagina fisica
des_pf* pagine_fisiche;

// numero di pagine fisiche
unsigned long num_pagine_fisiche;

// testa della lista di pagine fisiche libere
des_pf* pagine_libere = 0;

// indirizzo fisico della prima pagina fisica
// (necessario per calcolare l'indirizzo del descrittore associato ad una data
// pagina fisica e viceversa)
pagina_fisica* prima_pagina;

// restituisce l'indirizzo fisico della pagina fisica associata al descrittore
// passato come argomento
pagina_fisica* indirizzoPF(des_pf* p)
{
        return static_cast<pagina_fisica*>(add(prima_pagina, (p - pagine_fisiche) * SIZE_PAGINA));
}

// dato l'indirizzo di una pagina fisica, restituisce un puntatore al
// descrittore associato
des_pf* strutturaPF(pagina_fisica* pagina)
{
        return &pagine_fisiche[distance(pagina, prima_pagina) / SIZE_PAGINA];
}

// init_pagine_fisiche viene chiamata in fase di inizalizzazione.  Tutta la
// memoria non ancora occupata viene usata per le pagine fisiche.  La funzione
// si preoccupa anche di allocare lo spazio per i descrittori di pagina fisica,
// e di inizializzarli in modo che tutte le pagine fisiche risultino libere
bool init_pagine_fisiche()
{

        // allineamo mem_upper alla linea, per motivi di efficienza
        salta_a(allineav(mem_upper, sizeof(int)));

        // calcoliamo quanta memoria principale rimane
        int dimensione = distance(max_mem_upper, mem_upper);

        if (dimensione <= 0) {
                flog(LOG_ERR, "Non ci sono pagine libere");
                return false;
        }

        // calcoliamo quante pagine fisiche possiamo definire (tenendo conto
```

```
        // del fatto che ogni pagina fisica avra' bisogno di un descrittore)
        unsigned int quante = dimensione / (SIZE_PAGINA + sizeof(des_pf));

        // allochiamo i corrispondenti descrittori di pagina fisica
        pagine_fisiche = static_cast<des_pf*>(occupa(sizeof(des_pf) * quante));

        // riallineamo mem_upper a un multiplo di pagina
        salta_a(allineav(mem_upper, SIZE_PAGINA));

        // ricalcoliamo quante col nuovo mem_upper, per sicurezza
        // (sara' minore o uguale al precedente)
        quante = distance(max_mem_upper, mem_upper) / SIZE_PAGINA;

        // occupiamo il resto della memoria principale con le pagine fisiche;
        // ricordiamo l'indirizzo della prima pagina fisica e il loro numero
        prima_pagina = static_cast<pagina_fisica*>(occupa(quante * SIZE_PAGINA));
        num_pagine_fisiche = quante;

        // se resta qualcosa (improbabile), lo diamo all'allocatore a lista
        salta_a(max_mem_upper);

        // costruiamo la lista delle pagine fisiche libere
        des_pf* p = 0;
        for (int i = quante - 1; i >= 0; i--) {
                pagine_fisiche[i].contenuto = LIBERA;
                pagine_fisiche[i].avl.prossima_libera = p;
                p = &pagine_fisiche[i];
        }
        pagine_libere = &pagine_fisiche[0];

        return true;
}

// funzione di allocazione generica di una pagina
// Nota: restituisce un puntatore al *descrittore* della pagina, non alla
// pagina
des_pf* alloca_pagina()
{
        des_pf* p = pagine_libere;
        if (p != 0)
                pagine_libere = pagine_libere->avl.prossima_libera;
        return p;
}

// perche' la struttura dati composta dai descrittori di pagina fisica sia
// consistente, e' necessario che, ogni volta che si alloca una pagina, si
// aggiorni il campo "contenuto" del corrispondente descrittore in base all'uso
// che si deve fare della pagina.  Per semplificare questa operazione
// ripetitiva, definiamo una funzione di allocazione specifica per ogni
// possibile uso della pagina fisica da allocare (in questo modo, otteniamo
// anche il vantaggio di poter restituire un puntatore alla pagina del tipo
// corretto)
direttorio* alloca_direttorio()
{
        des_pf* p = alloca_pagina();
        if (p == 0) return 0;
        p->contenuto = DIRETTORIO;

        // tramite il descrittore puntato da "p", calcoliamo l'indirizzo della
        // pagina descritta (funzione "indirizzo"), che puntera' ad una "union
        // pagina_fisica". Quindi, restituiamo un puntatore al campo di tipo
        // "direttorio" all'interno della union
        direttorio *pdir = &indirizzoPF(p)->dir;
        return pdir;
}

// le tre funzioni per l'allocazione di una tabella differiscono solo per il
// valore da scrivere nel campo tipo. Poiche' e' buona norma di programmazione
// evitare il copia e incolla, ne definiamo una sola, parametrizzata sul tipo.
// Le altre si limitano a richiamare questa con il tipo corretto
tabella_pagine* alloca_tabella(cont_pf tipo = TABELLA_PRIVATA)
{
        des_pf* p = alloca_pagina();
        if (p == 0) return 0;
        p->contenuto = tipo;
        p->tab.quante = 0;
```

```
        tabella_pagine *ptab = &indirizzoPF(p)->tab;
        return ptab;
}

tabella_pagine* alloca_tabella_condivisa()
{
        return alloca_tabella(TABELLA_CONDIVISA);
}

tabella_pagine* alloca_tabella_residente()
{
        return alloca_tabella(TABELLA_RESIDENTE);
}


// analogo discorso per le pagine
pagina* alloca_pagina_virtuale(cont_pf tipo = PAGINA)
{
        des_pf* p = alloca_pagina();
        if (p == 0) return 0;
        p->contenuto = tipo;
        return &indirizzoPF(p)->pag;
}

pagina* alloca_pagina_residente()
{
        return alloca_pagina_virtuale(PAGINA_RESIDENTE);
}

// rende libera la pagina associata al descrittore di pagina fisica puntato da
// "p"
void rilascia_pagina(des_pf* p)
{
        p->contenuto = LIBERA;
        p->avl.prossima_libera = pagine_libere;
        pagine_libere = p;
}

// funzioni di comodo per il rilascio di una pagina fisica
// dato un puntatore a un direttorio, a una tabella o ad una pagina, ne
// calcolano prima l'indirizzo della pagina fisica che le contiene (semplice
// conversione di tipo) tramite la funzione "pfis", quindi l'indirizzo del
// descrittore di pagina fisica associato (funzione "struttura") e, infine,
// invocano la funzione "rilascia_pagina" generica.
inline
void rilascia(direttorio* d)
{
        rilascia_pagina(strutturaPF(pfis(d)));
}

inline
void rilascia(tabella_pagine* d)
{
        rilascia_pagina(strutturaPF(pfis(d)));
}

inline
void rilascia(pagina* d)
{
        rilascia_pagina(strutturaPF(pfis(d)));
}

// funzioni che aggiornano sia le strutture dati della paginazione che
// quelle relative alla gestione delle pagine fisiche:

// collega la tabella puntata da "ptab" al direttorio puntato "pdir" all'indice
// "indice". Aggiorna anche il descrittore di pagina fisica corrispondente alla
// pagina fisica che contiene la tabella.
// NOTA: si suppone che la tabella non fosse precedentemente collegata, e
// quindi che il corrispondente descrittore di tabella nel direttorio contenga
// il numero del blocco della tabella nello swap
descrittore_tabella* collega_tabella(direttorio* pdir, tabella_pagine* ptab, short indice)
{
        descrittore_tabella* pdes_tab = &pdir->entrate[indice];

        // mapping inverso:
```

```
        // ricaviamo il descrittore della pagina fisica che contiene la tabella
        des_pf* ppf  = strutturaPF(pfis(ptab));
        ppf->tab.dir    = pdir;
        ppf->tab.indice = indice;
        ppf->tab.quante = 0; // inizialmente, nessuna pagina e' presente

        // il contatore deve essere inizializzato come se fosse appena stato
        // effettuato un accesso (cosa che, comunque, avverra' al termine della
        // gestione del page fault). Diversamente, la pagina o tabella appena
        // caricata potrebbe essere subito scelta per essere rimpiazzata, al
        // prossimo page fault, pur essendo stata acceduta di recente (infatti,
        // non c'e' alcuna garanzia che la routine del timer riesca ad andare
        // in esecuzione e aggiornare il contatore prima del prossimo page
        // fault)
        ppf->contatore       = 0x80000000;
        ppf->tab.blocco      = pdes_tab->address; // vedi NOTA prec.

        // mapping diretto
        pdes_tab->address    = uint(ptab) >> 12;
        pdes_tab->D          = 0; // bit riservato nel caso di descr. tabella
        pdes_tab->pgsz       = 0; // pagine di 4KB
        pdes_tab->P          = 1; // presente

        return pdes_tab;
}

// scollega la tabella di indice "indice" dal direttorio puntato da "ptab".
// Aggiorna anche il contatore di pagine nel descrittore di pagina fisica
// corrispondente alla tabella
descrittore_tabella* scollega_tabella(direttorio* pdir, short indice)
{
        // poniamo a 0 il bit di presenza nel corrispondente descrittore di
        // tabella
        descrittore_tabella* pdes_tab = &pdir->entrate[indice];
        pdes_tab->P = 0;

        // quindi scriviamo il numero del blocco nello swap al posto
        // dell'indirizzo fisico
        des_pf* ppf = strutturaPF(pfis(tabella_puntata(pdes_tab)));
        pdes_tab->address = ppf->tab.blocco;

        return pdes_tab;
}

// collega la pagina puntata da "pag" alla tabella puntata da "ptab",
// all'indirizzo virtuale "ind_virtuale". Aggiorna anche il descrittore di
// pagina fisica corrispondente alla pagina fisica che contiene la pagina.
// NOTA: si suppone che la pagina non fosse precedentemente collegata, e quindi
// che il corrispondente descrittore di pagina nella tabella contenga
// il numero del blocco della pagina nello swap
descrittore_pagina* collega_pagina(tabella_pagine* ptab, pagina* pag, void* ind_virtuale)
{

        descrittore_pagina* pdes_pag = &ptab->entrate[indice_tabella(ind_virtuale)];

        // mapping inverso
        des_pf* ppf = strutturaPF(pfis(pag));
        ppf->pag.tabella        = ptab;
        ppf->pag.ind_virtuale   = ind_virtuale;
        // per il campo contatore, vale lo stesso discorso fatto per le tabelle
        // delle pagine
        ppf->contatore          = 0x80000000;
        ppf->pag.blocco        = pdes_pag->address; // vedi NOTA prec

        // incremento del contatore nella tabella
        ppf = strutturaPF(pfis(ptab));
        ppf->tab.quante++;
        ppf->contatore                |= 0x80000000;

        // mapping diretto
        pdes_pag->address        = uint(pag) >> 12;
        pdes_pag->pgsz                = 0; // bit riservato nel caso di descr. di pagina
        pdes_pag->P                 = 1; // presente

        return pdes_pag;
}
```

```
// scollega la pagina di indirizzo virtuale "ind_virtuale" dalla tabella
// puntata da "ptab". Aggiorna anche il contatore di pagine nel descrittore di
// pagina fisica corrispondente alla tabella
descrittore_pagina* scollega_pagina(tabella_pagine* ptab, void* ind_virtuale)
{

        // poniamo a 0 il bit di presenza nel corrispondente descrittore di
        // pagina
        descrittore_pagina* pdes_pag = &ptab->entrate[indice_tabella(ind_virtuale)];
        pdes_pag->P = 0;

        // quindi scriviamo il numero del blocco nello swap al posto
        // dell'indirizzo fisico
        des_pf* ppf = strutturaPF(pfis(pagina_puntata(pdes_pag)));
        pdes_pag->address = ppf->pag.blocco;

        // infine, decrementiamo il numero di pagine puntate dalla tabella
        ppf = strutturaPF(pfis(ptab));
        ppf->tab.quante--;

        return pdes_pag;
}

// mappa la memoria fisica, dall'indirizzo 0 all'indirizzo max_mem, nella
// memoria virtuale gestita dal direttorio pdir
// (la funzione viene usata in fase di inizializzazione)
bool mappa_mem_fisica(direttorio* pdir, void* max_mem)
{
        descrittore_tabella* pdes_tab;
        tabella_pagine* ptab;
        descrittore_pagina *pdes_pag;

        for (void* ind = addr(0); ind < max_mem; ind = add(ind, SIZE_PAGINA)) {
                pdes_tab = &pdir->entrate[indice_direttorio(ind)];
                if (pdes_tab->P == 0) {
                        ptab = alloca_tabella_residente();
                        if (ptab == 0) {
                                flog(LOG_ERR, "Impossibile allocare le tabelle condivise");
                                return false;
                        }
                        pdes_tab->address   = uint(ptab) >> 12;
                        pdes_tab->D            = 0;
                        pdes_tab->pgsz      = 0; // pagine di 4K
                        pdes_tab->D            = 0;
                        pdes_tab->US            = 0; // livello sistema;
                        pdes_tab->RW            = 1; // scrivibile
                        pdes_tab->P             = 1; // presente
                } else {
                        ptab = tabella_puntata(pdes_tab);
                }
                pdes_pag = &ptab->entrate[indice_tabella(ind)];
                pdes_pag->address = uint(ind) >> 12;   // indirizzo virtuale == indirizzo fisico
                pdes_tab->pgsz    = 0;
                pdes_pag->global  = 1; // puo' restare nel TLB, anche in caso di flush
                pdes_pag->US      = 0; // livello sistema;
                pdes_pag->RW          = 1; // scrivibile
                pdes_pag->PCD     = 0;
                pdes_pag->PWT     = 0;
                pdes_pag->P       = 1; // presente
        }
        return true;
}
```

```
////////////////////////////////////////////////////////////////////////
// MEMORIA VIRTUALE                                                   //
////////////////////////////////////////////////////////////////////////
//
// ACCESSO ALLO SWAP
//
// lo swap e' descritto dalla struttura des_swap, che specifica il canale
// (primario o secondario) il drive (primario o master) e il numero della
```

```
// partizione che lo contiene. Inoltre, la struttura contiene una mappa di bit,
// utilizzata per l'allocazione dei blocchi in cui lo swap e' suddiviso, e un
// "super blocco".  Il contenuto del super blocco e' copiato, durante
// l'inizializzazione del sistema, dal primo settore della partizione di swap,
// e deve contenere le seguenti informazioni:
// - magic (un valore speciale che serve a riconoscere la partizione, per
// evitare di usare come swap una partizione utilizzata per altri scopi)
// - bm_start: il primo blocco, nella partizione, che contiene la mappa di bit
// (lo swap, inizialmente, avra' dei blocchi gia' occupati, corrispondenti alla
// parte utente/condivisa dello spazio di indirizzamento dei processi da
// creare: e' necessario, quindi, che lo swap stesso memorizzi una mappa di
// bit, che servira' come punto di partenza per le allocazioni dei blocchi
// successive)
// - blocks: il numero di blocchi contenuti nella partizione di swap (esclusi
// quelli iniziali, contenenti il superblocco e la mappa di bit)
// - directory: l'indice del blocco che contiene il direttorio
// - l'indirizzo virtuale dell'entry point del programma contenuto nello swap
// (l'indirizzo di main)
// - l'indirizzo virtuale successivo all'ultima istruzione del programma
// contenuto nello swap (serve come indirizzo di partenza dello heap utente)
// - l'indirizzo virtuale dell'entry point del modulo io contenuto nello swap
// - l'indirizzo virtuale successivo all'ultimo byte occupato dal modulo io
// - checksum: somma dei valori precedenti (serve ad essere ragionevolmente
// sicuri che quello che abbiamo letto dall'hard disk e' effettivamente un
// superblocco di questo sistema, e che il superblocco stesso non e' stato
// corrotto)
//

// descrittore di una partizione dell'hard disk
struct partizione {
        int             type;        // tipo della partizione
        unsigned int first;        // primo settore della partizione
        unsigned int dim;        // dimensione in settori
        partizione*  next;
};

// funzioni per la lettura/scrittura da hard disk
extern "C" void readhd_n(short ind_ata, short drv, void* vetti,
                unsigned int primo, unsigned char quanti, char &errore);
extern "C" void writehd_n(short ind_ata, short drv, void* vetti,
                unsigned int primo, unsigned char quanti, char &errore);
// utile per il debug: invia al log un messaggio relativo all'errore che e'
// stato riscontrato
void hd_print_error(int i, int d, int sect, short errore);
// cerca la partizione specificata
partizione* hd_find_partition(short ind_ata, short drv, int p);


// super blocco (vedi sopra)
struct superblock_t {
        char                magic[4];
        unsigned int        bm_start;
        int                blocks;
        unsigned int        directory;
        int                (*user_entry)(int);
        void*                user_end;
        int                (*io_entry)(int);
        void*                io_end;
        unsigned int        checksum;
};

// descrittore di swap (vedi sopra)
struct des_swap {
        short channel;                // canale: 0 = primario, 1 = secondario
        short drive;                // dispositivo: 0 = master, 1 = slave
        partizione* part;        // partizione all'interno del dispositivo
        bm_t free;                // bitmap dei blocchi liberi
        superblock_t sb;        // contenuto del superblocco
} swap;         // c'e' un unico oggetto swap


// legge dallo swap il blocco il cui indice e' passato come primo parametro,
// copiandone il contenuto a partire dall'indirizzo "dest"
bool leggi_blocco(unsigned int blocco, void* dest)
{
        char errore;
```

```
        // ogni blocco (4096 byte) e' grande 8 settori (512 byte)
        // calcoliamo l'indice del primo settore da leggere
        unsigned int sector = blocco * 8 + swap.part->first;

        // il seguente controllo e' di fondamentale importanza: l'hardware non
        // sa niente dell'esistenza delle partizioni, quindi niente ci
        // impedisce di leggere o scrivere, per sbaglio, in un'altra partizione
        if (blocco < 0 || sector + 8 > (swap.part->first + swap.part->dim)) {
                flog(LOG_ERR, "Accesso al di fuori della partizione");
                // fermiamo tutto
                panic("Errore interno");
        }

        readhd_n(swap.channel, swap.drive, dest, sector, 8, errore);

        if (errore != 0) {
                flog(LOG_ERR, "Impossibile leggere il blocco %d", blocco);
                hd_print_error(swap.channel, swap.drive, blocco * 8, errore);
                return false;
        }
        return true;
}


// scrive nello swap il blocco il cui indice e' passato come primo parametro,
// copiandone il contenuto a partire dall'indirizzo "dest"
bool scrivi_blocco(unsigned int blocco, void* dest)
{
        char errore;
        unsigned int sector = blocco * 8 + swap.part->first;

        if (blocco < 0 || sector + 8 > (swap.part->first + swap.part->dim)) {
                flog(LOG_ERR, "Accesso al di fuori della partizione");
                // come sopra
                panic("Errore interno");
                return false;
        }

        writehd_n(swap.channel, swap.drive, dest, sector, 8, errore);

        if (errore != 0) {
                flog(LOG_ERR, "Impossibile scrivere il blocco %d", blocco);
                hd_print_error(swap.channel, swap.drive, blocco * 8, errore);
                return false;
        }
        return true;
}

// lettura dallo swap (da utilizzare nella fase di inizializzazione)
bool leggi_swap(void* buf, unsigned int first, unsigned int bytes, const char* msg)
{
        char errore;
        unsigned int sector = first + swap.part->first;

        if (first < 0 || first + bytes > swap.part->dim) {
                flog(LOG_ERR, "Accesso al di fuori della partizione: %d+%d", first, bytes);
                return false;
        }

        readhd_n(swap.channel, swap.drive, buf, sector, ceild(bytes, 512), errore);

        if (errore != 0) {
                flog(LOG_ERR, "\nImpossibile leggere %s", msg);
                hd_print_error(swap.channel, swap.drive, sector, errore);
                return false;
        }
        return true;
}

// inizializzazione del descrittore di swap
char read_buf[512];
bool swap_init(int swap_ch, int swap_drv, int swap_part)
{
        partizione* part;
```

```
        // l'utente *deve* specificare una partizione
        if (swap_ch == -1 || swap_drv == -1 || swap_part == -1) {
                flog(LOG_ERR, "Partizione di swap non specificata!");
                return false;
        }

        part = hd_find_partition(swap_ch, swap_drv, swap_part);
        if (part == 0) {
                flog(LOG_ERR, "Swap: partizione non esistente o non rilevata");
                return false;
        }

        // se la partizione non comprende l'intero hard disk (swap_part > 0),
        // controlliamo che abbia il tipo giusto
        if (swap_part && part->type != 0x3f) {
                flog(LOG_ERR, "Tipo della partizione di swap scorretto (%d)", part->type);
                return false;
        }

        swap.channel = swap_ch;
        swap.drive   = swap_drv;
        swap.part    = part;

        // lettura del superblocco
        flog(LOG_DEBUG, "lettura del superblocco dall'area di swap...");
        if (!leggi_swap(read_buf, 1, sizeof(superblock_t), "il superblocco"))
                return false;

        swap.sb = *reinterpret_cast<superblock_t*>(read_buf);

        // controlliamo che il superblocco contenga la firma di riconoscimento
        if (swap.sb.magic[0] != 'C' ||
            swap.sb.magic[1] != 'E' ||
            swap.sb.magic[2] != 'S' ||
            swap.sb.magic[3] != 'W')
        {
                flog(LOG_ERR, "Firma errata nel superblocco");
                return false;
        }

        flog(LOG_DEBUG, "lettura della bitmap dei blocchi...");

        // calcoliamo la dimensione della mappa di bit in pagine/blocchi
        unsigned int pages = ceild(swap.sb.blocks, SIZE_PAGINA * 8);

        // quindi allochiamo in memoria un buffer che possa contenerla
        unsigned int* buf = new unsigned int[(pages * SIZE_PAGINA) / sizeof(unsigned int)];
        if (buf == 0) {
                flog(LOG_ERR, "Impossibile allocare la bitmap dei blocchi");
                return false;
        }

        // inizializziamo l'allocatore a mappa di bit
        bm_create(&swap.free, buf, swap.sb.blocks);

        // infine, leggiamo la mappa di bit dalla partizione di swap
        return leggi_swap(buf, swap.sb.bm_start * 8, pages * SIZE_PAGINA, "la bitmap dei blocchi");
}
```

Riferimenti bibliografici

[1] Mailing list curata da ING. G. Lettieri http://lettieri.iet.unipi.it/mailman/listinfo/calcolatori

[2] Sorgente nucleo di riferimento http://calcolatori.iet.unipi.it/deep/nucleo.tar.gz

[3] Calcolatori Elettronici Domande & risposte A cura di A. Nannetti e A. Marzia, fotocopisteria il telaio.

[4] Paolo Corsini, Dalle porte AND OR NOT al sistema calcolatore, Edizioni ETS

[5] P. Corsini, G. Frosini, B. Lazzerini. *Architettura dei calcolatori con riferimento al Personal Computer*, MacGraw-Hill, Milano 1988.

[6] G. Frosini, B. Lazzerini *Calcolatori elettronici Volume IV* Edizione ETS.