Università di Pisa

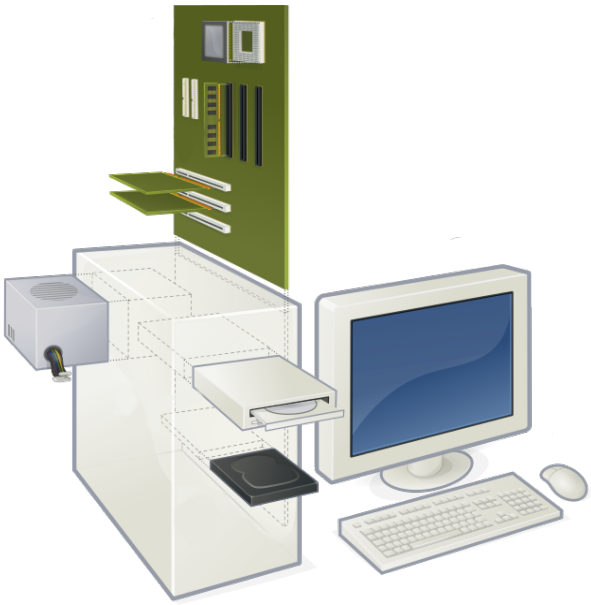
Dipartimento di Ingegneria dell'informazione

Corso di Laurea in Ingegneria Informatica

Calcolatori Elettronici

- Appunti -

a cura di
Marco Soricelli



Anno Accademico 2014/2015

Indice

Prefazione

Questa dispensa è il frutto di una commistione tra appunti presi alle lezioni del professor Giuseppe Lettieri e del professor Graziano Frosini con quanto scritto nei testi *Architettura dei Calcolatori, vol. I, II, III* i cui autori sono i professori stessi. L'ordine degli argomenti e gli esempi contenuti all'interno di questa dispensa, rispecchiano quelli forniti dai professori durante lo svolgimento del corso nell'anno accademico 2014/1015. Questo lavoro, pertanto, può essere utile come sostegno a chi deve ancora seguire il corso o come ripasso in vista dell'avvicinarsi della sessione d'esame. Però attenzione: nessun libro e tanto meno nessuna dispensa scritta da studenti può sostituire le lezioni frontali di un professore che sarà sicuramente più esauriente e pronto a rispondere ad eventuali domande di chiarimento.

Marco Soricelli

Capitolo 1

Memoria Cache

La ***memoria cache*** è un oggetto che fisicamente si trova fra la CPU e la memoria fisica (o centrale, o RAM). Il fatto di avere "due" memorie vuole servire a dare l'illusione di avere le tre caratteristiche ideali di una memoria ma che quasi sempre non coesistono e cioè capienza, velocità e basso costo.

1.1 Principio di località

La maggior parte dei programmi che vengono eseguiti da un calcolatore, rispetta il principio della località. Abbiamo tre tipi di località:

1. località ***sequenziale***: è probabile che se viene riferita una locazione di memoria, debba essere riferita anche quella immediatamente (sequenzialmente) successiva;

2. località ***spaziale***: è probabile che se viene riferita una locazione di memoria, debba essere riferita anche una locazione "nelle sue vicinanze" (in questo tipo di località è compreso quello sequenziale);

3. località ***temporale***: è probabile che se viene riferita una locazione, questa debba essere utilizzata nuovamente poco dopo.

Per dare corpo a questo principio, viene spesso adottata la gerarchia di memorie con almeno due livelli (Figura 1.1). Abbiamo quindi un *livello inferiore* dato dalla memoria fisica (o centrale) e un *livello superiore* dato dalla memoria cache. Anche nella memoria cache le locazioni sono raggruppate in blocchi (blocco: insieme allineato di locazioni consecutive).

La memoria cache deve essere in grado di "rispondere" alla domanda ‹‹quello che il processore cerca è in cache oppure no?››.

Il trasferimento delle informazioni dal livello inferiore a quello superiore avviene sempre a livello di blocco.

1.2 Memoria fisica: richiamo

Ogni indirizzo può essere visto diviso in due parti: la parte più significativa a che rappresenta il numero dei sottoblocchi della memoria fisica, la parte meno significativa b che rappresenta la grandezza massima di ciascun sottoblocco (2^b). La parte a possiamo vederla divisa, a sua volta in due parti: la parte più significativa

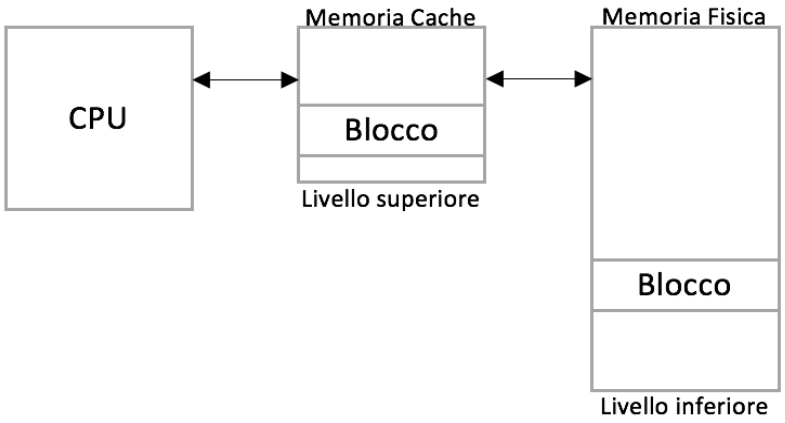


Figura 1.1: Gerarchia di memorie

E rappresenta l'etichetta (*tag*), la parte meno significativa I rappresenta l'indice. Guardando solo la parte a di un indirizzo: se è in memoria cache allora vi è anche tutto il relativo blocco (2^b locazioni).

1.3 Tripartizione di un indirizzo

L'indirizzo di una locazione di memoria generato dal processore è pensato suddiviso in tre parti:

Figura 1.2: Tripartizione di un indirizzo

N.B.: la suddivisione dell'indirizzo in figura fa riferimento al caso in cui la cache sia di 256 KB e composta da blocchi da 32 B e abbiamo:

- l'etichetta (E) è costituita dai 14 bit `A31_A18`
- l'indice (I) è costituito dai 13 bit `A17_A5`
- lo spiazzamento (S) è costituito dai 3 bit `A4_A2` per quanto riguarda la riga e dai 4 bit `/BE3_/BE0` per le locazioni all'interno della riga.

La componente I dell'indirizzo seleziona un gruppo all'interno della cache. Un gruppo è composto da:

- un bit di validità V
- un campo etichetta Ev
- un campo blocco Bv

1.4 Memoria cache a indirizzamento diretto

La componente S dell'indirizzo individua una specifica locazione all'interno del campo Bv; la componente E dell'indirizzo viene confrontata con il campo etichetta Ev del gruppo selezionato per verificarne o meno l'uguaglianza. Infine il bit di validità V e l'uscita del comparatore vengono messi in AND e il risultato dell'operazione viene interpretato come *successo* (o HIT) o come *fallimento* (o MISS).

Una memoria cache a indirizzamento diretto è così organizzata:

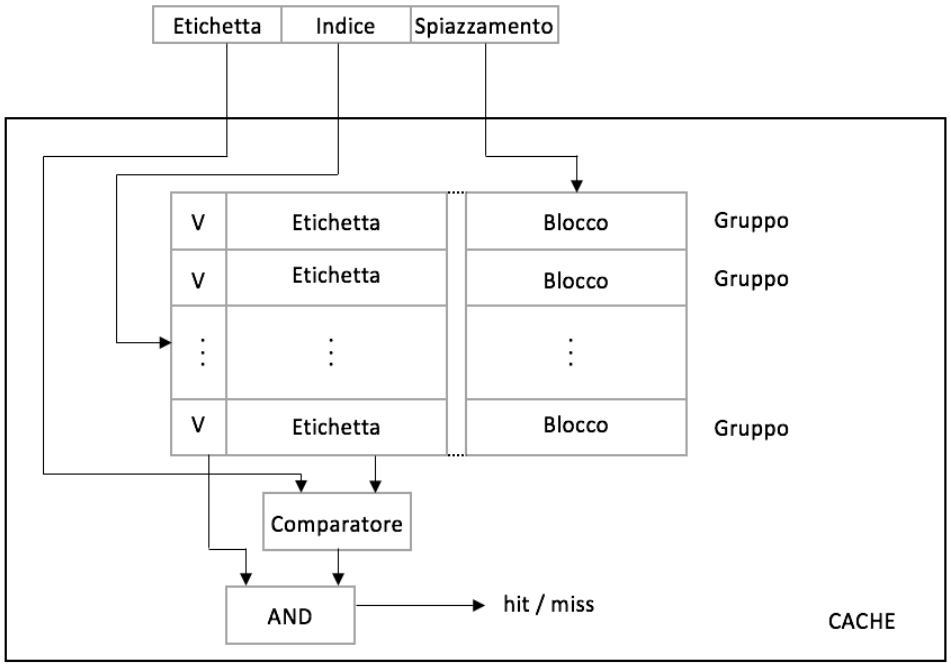


Figura 1.3: Cache a indirizzamento diretto

Dal punto di vista realizzativo, una memoria cache è costituita da due banchi: uno contenente bit di validità V ed etichette (o tags), l'altro contenente i blocchi. Questo perché possono venir effettuati accessi di tipo diverso sui due blocchi. Infatti abbiamo sempre una lettura nel primo banco e, in caso di successo, una scrittura o una lettura nel secondo.

1.4.1 Operazione di lettura

Quando il processore compie un'operazione di lettura, avviene:

1. accesso in lettura al gruppo della memoria cache individuato dalla componente I dell'indirizzo.

2. esame del bit V e confronto fra la componente E dell'indirizzo e la componente Ev del gruppo.

Queste due operazioni possono dar luogo a:

1. successo (hit):

   a. l'entità riferita viene fornita al processore.

2. fallimento (miss):

   a. accesso in scrittura nella cache nel gruppo selezionato dalla componente I dell'indirizzo con la memorizzazione di nuove quantità nel blocco del tag e trasferimento di un nuovo blocco dalla memoria fisica (rimpiazzamento del vecchio gruppo).

   b. trasferimento dell'entità informativa riferita dalla memoria fisica al processore.

1.4.2 Operazione di scrittura

Quando il processore compie un'operazione di scrittura, avviene:

1. accesso in lettura al gruppo della memora cache individuato dalla componente I dell'indirizzo.

2. esame del bit V e confronto fra la componente E dell'indirizzo e la componente Ev del gruppo.

Queste due operazioni possono dar luogo a:

1. successo (hit):

   a. l'entità informativa viene scritta sia nella cache che nella memoria fisica.

2. fallimento (miss):

   a. l'entità informativa viene memorizzata solo in memoria fisica.

   b. accesso in scrittura nella cache nel gruppo selezionato dalla componente I dell'indirizzo con la memorizzazione di nuove quantità nel blocco del tag e trasferimento del blocco che è stato modificato dalla memoria fisica (rimpiazzamento del vecchio gruppo).

Utilizzando questa regola di scrittura, detta **write through**, non traiamo nessun vantaggio dalla presenza o meno della cache. Per questo esiste un'altra regola di scrittura, detta **write back**, che sfrutta i vantaggi che la cache offre: in caso di successo l'entità informativa viene scritta solo nella memoria cache. L'entità informativa viene preliminarmente trasferita nella memoria fisica solo quando una lettura o una scrittura nella cache al gruppo selezionato provoca un fallimento e dunque necessitiamo di rimpiazzare il vecchio gruppo.

1.4.3 Esempio: memoria cache a indirizzamento diretto con 4 gruppi

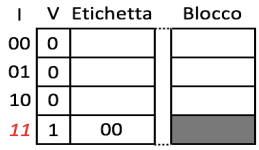


Figura 1.4: Memoria cache (4 gruppi)

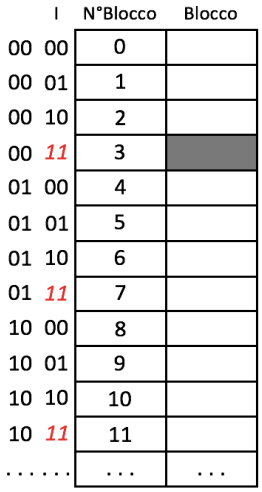


Figura 1.5: Memoria fisica a 2^4 entrate

Avendo un solo banco per i blocchi, tutti i blocchi aventi stesso indice si dice che vanno in *conflitto* sulla stessa linea di cache. Perciò se dovessimo memorizzare in cache il blocco 7 (o 11), avendo già memorizzato in cache il blocco 3, occorrerebbe sovrascrivere il blocco già presente previo salvataggio in memoria fisica dello stesso (write back). Più è piccola la memoria cache più aumentano le collisioni. Se in quest'esempio avessimo avuto 8 gruppi anziché 4, le collisioni sarebbero evidentemente diminuite.

1.5 Controllore della memoria cache

La memoria cache prevede un circuito di controllo, chiamato **controllore cache**. Questo circuito ha il compito di rispondere alle operazioni di lettura e alle operazioni di scrittura comandate dalla CPU e ad effettuare le operazioni necessarie sulla memoria cache e sulla memoria fisica. Il processore non necessita di nessuna modifica atta alla gestione di questo componente in quanto già dotato del piedino `/READY` usato dal controllore per comunicare al processore stesso che sta compiendo delle operazioni (o sulla cache o sulla memoria fisica). Il controllore cache viene montato subito a valle del processore e prima del controllore del bus. Non occorre nemmeno modificare il software poiché questo meccanismo gli è del tutto trasparente. Per l'I/O non c'è la cache poiché sarebbe totalmente inutile (si pensi alla non utilità che avrebbe "ricordarsi" di un tasto della tastiera premuto in precedenza).

Quando il controllore della cache vede che l'operazione è di I/O, la fa passare così com'è.

Il controllore cache è anche in grado di invalidare uno o più gruppi della memoria cache ponendo a 0 il corrispondente bit V (*flush*). Questa operazione viene effettuata tipicamente con comandi software. A tale scopo, il controllore possiede tre registri interni: due destinati a contenere l'indice iniziale e l'indice finale dei gruppi da invalidare e uno destinato a ricevere il comando di flush. Per fare queste operazioni il controllore viene abilitato utilizzando il piedino /S.

1.6 Memoria cache *associativa a insiemi*

Come visto, la memoria cache a indirizzamento diretto specialmente se molto piccola, ha dei limiti ed in particolare non è possibile memorizzarvi due blocchi aventi lo stesso indice I e diversa etichetta E. Per ovviare a questo, vengono usate memorie cache **associative a insiemi** (o *set-associative*). Una memoria cache associativa a insiemi con cardinalità n (n-*way*) è costituita da un numero n di parti tutte uguali, ciascuna avente la struttura di una memoria cache a indirizzamento diretto, con l'aggiunta di un'ulteriore parte relativa al meccanismo del rimpiazzamento (R).

R	V	Etichetta	Blocco	V	Etichetta	Blocco
R	V	Etichetta	Blocco	V	Etichetta	Blocco
...	...	...	...	...	...	...
R	V	Etichetta	Blocco	V	Etichetta	Blocco

Figura 1.6: Memoria cache associativa a insiemi

L'indirizzo è sempre tripartito e sia l'etichetta E che lo spiazzamento S mantengono lo stesso significato. L'indice I, in questo caso, non serve più a selezionare un gruppo ma un insieme di n gruppi. Per ogni gruppo dell'insieme di n gruppi avviene l'esame del bit V e il confronto fra l'etichetta E dell'indirizzo e l'etichetta Ev presente in ciascun gruppo dell'insieme (a tale scopo servono n comparatori). In caso di successo, lo spiazzamento S individua la locazione all'interno del blocco di quel gruppo. Se si ha fallimento per ogni gruppo dell'intero insieme, si ha fallimento globale.

Un blocco di memoria, in relazione al suo indice, può essere indifferentemente memorizzato in uno degli n gruppi dell'insieme corrispondente. Per ogni insieme è previsto un campo R (rimpiazzamento) che individua il gruppo da utilizzare per una nuova memorizzazione, rimpiazzando il gruppo preesistente. Se tutti i bit V di un insieme valgono 1, deve essere fatto un rimpiazzamento; se un bit V vale 0 può essere usato quel gruppo.

Esempio: avendo una memoria cache con cardinalità 4, quanti bit occorrono per R? Cardinalità 4$\longrightarrow$occorrono 2 bit per discriminare 4 informazioni. Bisogna però ricordarsi non solo del precedente ma anche del penultimo e del terzultimo gruppo usato.

Quindi:

per discriminare 4 $\longrightarrow$ 2 bit +
per discriminare 3 $\longrightarrow$ 2 bit +
per discriminare 2 $\longrightarrow$ 1 bit = 5 bit (minimo).

Siccome la circuiteria diventerebbe troppo complessa, diciamo allora che in questo caso occorrono 6 bit e quindi 2 bit per ogni discriminazione.

In ogni caso, oggi vengono utilizzati solo 3 bit per il campo R. Se chiamiamo questi tre bit b0, b1 e b2, abbiamo che:

- b0 ci indica se il prossimo rimpiazzamento deve avvenire tra i primi due o gli ultimi due;
- b1 se deve avvenire tra il primo e il secondo;
- b2 fra il terzo e il quarto.

Tutto ciò facendo sempre riferimento ad una memoria cache associativa a insiemi con cardinalità 4.

Questa regola di rimpiazzamento è detta **LRU** (*Least Recently Used*) e si basa sul gruppo non riferito da più tempo. Occorre però notare che basta che un programma ciclico si trovi a dover lavorare su uno spazio più grande della memoria cache che la regola di rimpiazzamento LRU diventa pessima.

Capitolo 2

Memoria Virtuale

2.1 Richiamo sul funzionamento di massima di un Hard-Disk

Occorre ricordare in breve cosa sia un Hard-Disk prima di passare a dire cosa sia la *memoria virtuale*.

Un HD è una sequenza di blocchi, ciascuno di dimensione 512 byte, ognuno dei quali identificato da un numero. Quello che possiamo chiedere all'HD è di leggere un certo blocco; l'HD lo carica sul suo buffer interno e poi, *due a due*, vengono letti i byte nel buffer. Viceversa, per scrivere un blocco, occorre scrivere i byte, *due a due* nel buffer interno dell'HD e poi l'HD stesso trasferisce il contenuto del buffer nel blocco desiderato. Queste sono ***le uniche operazioni che si possono compiere con un HD***.

2.2 Il problema della dimensione dei programmi

Il problema principale è simile a quello già visto per la memoria cache ed è sostanzialmente il fatto che vogliamo avere sempre più memoria, ma le memorie di grandi capacità o sono lente o poco pratiche da usare.

Il nostro problema è la memoria fisica che abbiamo nel sistema, cioè la RAM che abbiamo installato. Per quanto grande possa essere la RAM c'è sempre un programma che occuperebbe più memoria fisica di quella che abbiamo in realtà installato.

Esempio: programma che somma gli elementi di un array di interi.

```
1 const int N = un_numero;
2 int v[N] = {1, 5, 6, ... ,3};
3 int main(){
4         int somma = 0;
5         for(int i = 0; i<N; i++)
6                 somma+=v[i];
7         return somma;
8 }
```

In Assembly questo programma si tradurrà, prima di tutto, nel salvataggio degli elementi dell'array in memoria. Qui abbiamo già il primo problema: c'è sempre un valore di N per cui la RAM non basta a contenere tutti gli elementi dell'array. Il caso limite è dato dall'impossibilità, per questa architettura, di rappresentare più di 4 miliardi di indirizzi (2^{32}).

Supponiamo in ogni caso che 2^{32} indirizzi siano sufficienti ma non riusciamo ad avere tanta memoria a causa del costo. Un programma in cui c'è un array più grande della memoria, nelle condizioni che abbiamo visto, non lo possiamo eseguire. Questo però non vuol dire che non vi sia il modo per eseguirlo. Se i dati che non possiamo salvare nella memoria fisica, li salviamo nell'HD (che è molto più capiente) possiamo caricare in memoria, di volta in volta, una parte, sommare gli elementi di quella parte, salvare la somma, caricare un'altra parte sovrascrivendo i valori già utilizzati, sommare questi valori al risultato ottenuto in precedenza e così via.

Per esempio: sapendo quanto è grande la memoria fisica, supponiamo M, potremmo riorganizzare il programma precedentemente visto, in questo modo:

```
1 int buff[M];
2 int main(){
3         int somma = 0;
4         for(int i = 0; i<...; i++){
5                 read_hd(..., buff);
6                 for(int j = 0; j<M; j++)
7                         somma+=buff[j];
8         }
9         return somma;
10 }
```

Abbiamo comunque un array che però è solo un buffer temporaneo grande esattamente quanto la RAM; i veri dati, invece, sono sull'HD. Ciclicamente leggiamo dall'HD (`read_hd(...)`) e poi sommiamo i valori caricati.

Questo tipo di implementazione presuppone però che si sappia a priori la capacità della memoria che abbiamo montato, e se dunque, in futuro, si decidesse di espandere la RAM, occorrerebbe rivisitare anche il codice del programma in quanto il valore di M è mutato e se non lo cambiassimo, i vantaggi provenienti dall'aver aggiunto altra memoria verrebbero meno.

2.3 Utilità della memoria virtuale

Proprio per ovviare a questo problema di "non portabilità" è stato introdotto il concetto di ***memoria virtuale***.

Per contenere il programma, o parti di esso, viene anche in questo caso utilizzato l'HD; la memoria fisica la utilizzeremo come se fosse una sorta di cache dell'HD. Questo metodo vuole essere trasparente al programma cioè vogliamo prendere il programma non modificato di grandezza qualsivoglia, e poterlo eseguire sul nostro sistema indipendentemente dalla capacità della memoria fisica.

2.3.1 Meccanismo semplificato: "*super-MMU*"

Usiamo, almeno in parte, un espediente simile a quello visto per la memoria cache. Ci serviamo di un nuovo oggetto, la **MMU** (*Memory Menagement Unit*), che andiamo a montare tra la CPU e la memoria fisica, in modo tale che tutti gli accessi in memoria fatti dal processore vi passino attraverso.

Quello che il programmatore deve fare è scrivere il programma facendo finta di avere a disposizione il massimo della memoria (che è quella data dalla massima dimensione degli indirizzi, 2^{32} in questo caso) e che sia tutta per il programma che sta scrivendo. Naturalmente la memoria fisica è molto più piccola e quindi al suo interno sarà presente solo una piccola parte del programma; il resto del programma starà tutto nell'Hard-Disk.

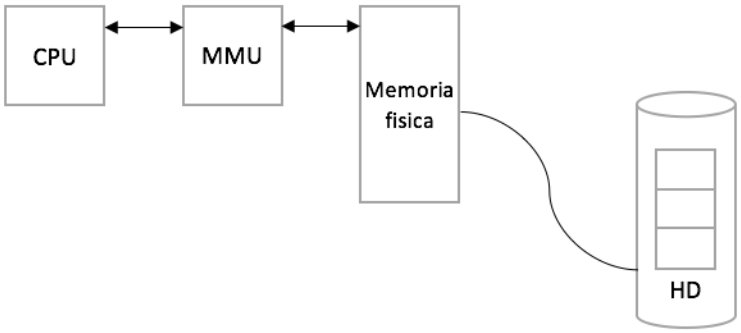


Figura 2.1: Inserzione della MMU

Adesso sorge, però, un altro problema: il processore non ha istruzioni per prelevare le informazioni dall'HD (non possiamo accedere a quello che c'è nell'HD con una mov). Ci vuole un programma che ordini all'HD di copiare il contenuto di un certo blocco nel suo buffer interno e successivamente copiare il contenuto di questo buffer da qualche parte in memoria fisica.

Qualunque cosa il processore chieda alla MMU, la MMU deve vedere se per caso si trova nella memoria fisica, altrimenti deve far partire un'operazione (una *routine*) di lettura dall'HD, caricarla da qualche parte in memoria fisica e quindi completare l'operazione che il processore aveva richiesto.

Suddividiamo la memoria virtuale in ***blocchi***. Dobbiamo poter leggere/scrivere da/su Hard-Disk un intero blocco. Chiamiamo questi blocchi ***pagine***. Anche la memoria fisica viene suddivisa in pagine. Nella nostra architettura ogni pagina è grande 4 KB (che corrisponde a 8 blocchi dell'HD, cioè 512B*8). La MMU decide di volta in volta se trasferire una pagina dall'HD alla memoria fisica o viceversa. Questo implica che gli indirizzi li stiamo nuovamente immaginando bipartiti. La parte a (20 bit più significativi), indica la pagina; la parte b (12 bit meno significativi), indica l'offset del byte nella pagina. La memoria fisica dobbiamo quindi immaginarla come il buffer nell'esempio e cioè come uno spazio che usiamo di volta in volta per quello che ci serve. Anche l'HD è organizzato in spezzoni da 8 blocchi (ciascun blocco è da 512 byte) in modo da ottenere una pagina di 4KB.

La MMU, di cui ricordiamo che ***il processore non ne è a conoscenza***, avrà bisogno di una tabella che, in prima istanza, chiamiamo **tabella di corrispondenza**. Questa tabella ha tante entrate quante sono le pagine della memoria virtuale e in ogni occorrenza della tabella deve essere specificato, oltre ad altre informazioni sulla pagina, anche se la pagina si trova in memoria fisica o nell'HD.

La MMU si serve dunque di una tabella molto grande con 1.000.000 (2^{20} pagine) di entrate. Se una pagina si trova già nella memoria fisica o meno è espresso da un bit di presenza P in ogni occorrenza della tabella. Questo bit vale 1 se la pagina virtuale cercata è già in memoria fisica, 0 se invece si trova ancora nell'HD.

Come fa la MMU a sapere quale pagina deve andare a vedere se è nella memoria fisica o nell'HD? Lo fa attraverso i 20 bit più significativi dell'indirizzo che arriva dal processore. Questi 20 bit discriminano le pagine.

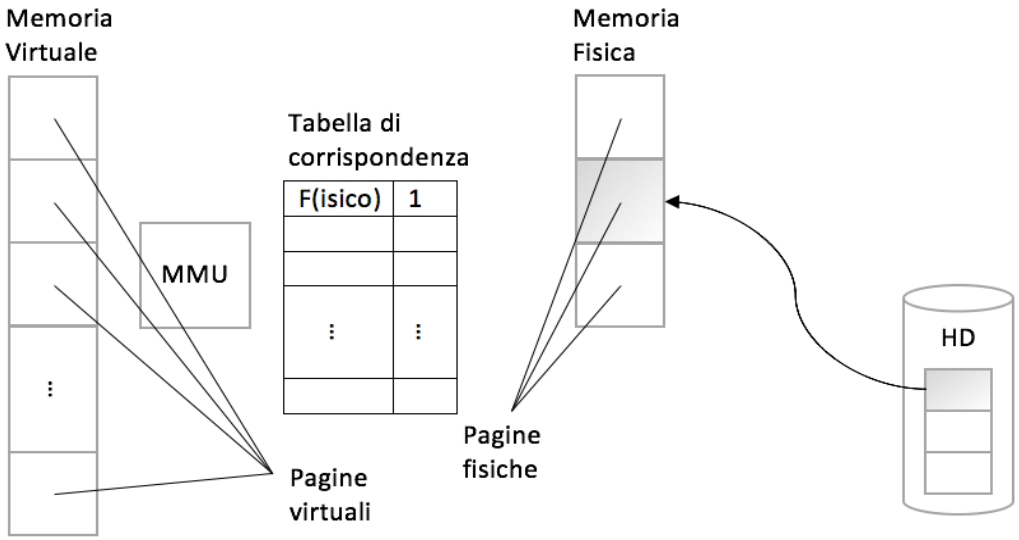


Figura 2.2: Panoramica delle strutture

2.4 Indirizzi *virtuali* e indirizzi *fisici*

Il processore emette dai suoi piedini *indirizzi virtuali*. Questi indirizzi passano attraverso la MMU che li traduce accedendo alla tabella di corrispondenza. Dopo la traduzione avremo *indirizzi fisici*. Quindi:

- tutti gli indirizzi "prima" della MMU sono *virtuali*
- tutti gli indirizzi "dopo" la MMU sono *fisici*

Un **indirizzo virtuale** riferisce una pagina virtuale. Un **indirizzo fisico** riferisce una pagina fisica (nella quale sarà stata posta precedentemente una pagina virtuale).

Una pagina fisica non è nient'altro che un "contenitore vuoto" in cui mettiamo, di volta in volta, la pagina virtuale che ci serve.

2.4.1 Traduzione degli indirizzi virtuali in indirizzi fisici

Quando un indirizzo virtuale arriva alla MMU, questa prende i primi 20 bit dell'indirizzo per accedere all'entrata della tabella corrispondente alla pagina virtuale. La traduzione "immediata" di un indirizzo virtuale in fisico, può avvenire se e solo se l'entrata della tabella relativa alla pagina virtuale cercata ha il bit P a 1; in tal caso, infatti, nella stessa entrata vi saranno anche altri 20 bit. Questi 20 bit sono i bit che identificano la posizione della pagina fisica in cui si trova la pagina virtuale che vi è stata caricata. E gli altri 12 bit? Questi rimangono invariati poiché, rispetto alla pagina stessa, l'offset dei byte rimane il medesimo. Quindi la MMU, per tradurre l'indirizzo virtuale in fisico, non deve far altro che sostituire i 20 bit più significativi dell'indirizzo virtuale arrivato dal processore con i 20 bit che trova nell'entrata della tabella. Abbiamo così ottenuto un indirizzo fisico.

Se il bit P vale 0, la pagina virtuale cercata non si trova in una pagina fisica ma si trova nella memoria di massa (HD). Al posto dei nuovi 20 bit, nell'entrata

della tabella corrispondente alla pagina virtuale cercata, troviamo invece l'indirizzo iniziale del blocco dell'Hard-Disk a partire dal quale si trova la pagina virtuale.

Quindi il processore ci fornisce un indirizzo virtuale fatto così:

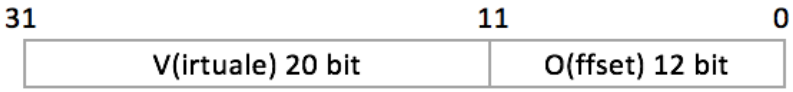


Figura 2.3: Indirizzo virtuale

Poi nella tabella di corrispondenza all'entrata di indice V(irtuale) andiamo a leggere che quella pagina virtuale è presente in RAM e si trova all'indirizzo F(isico).

Siamo quindi già in grado di poter completare l'operazione. La pagina che il processore stava cercando si troverà all'indirizzo fisico ottenuto.

Figura 2.4: Indirizzo fisico ottenuto dalla trasformazione

2.5 Esempio: simulazione del meccanismo

Supponiamo di avere la memoria fisica di 2 pagine, memoria virtuale di 4 pagine e anche l'HD di 4 pagine (32 blocchi da 512 byte).

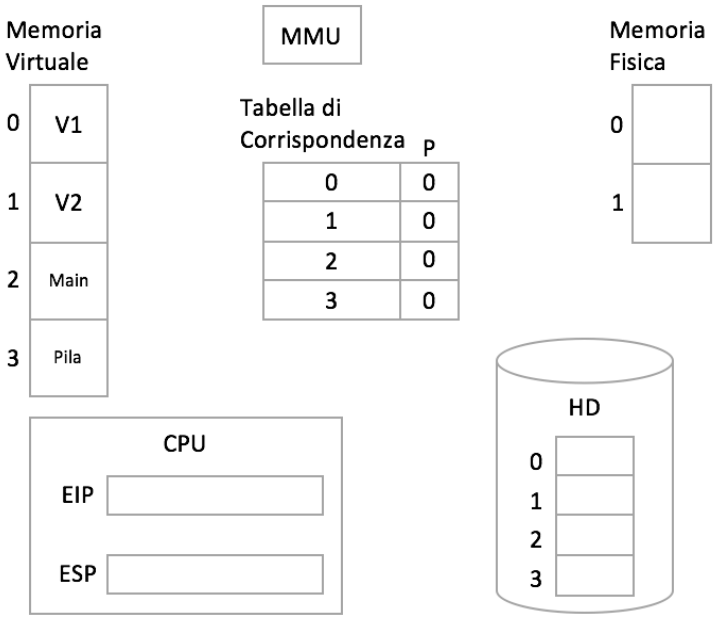


Figura 2.5: Esempio simulazione

Facciamo riferimento al programma che somma gli elementi di un array visto precedentemente. Supponiamo di aver occupato tutto. Chi prepara il programma, lo passa al sistema di caricamento in memoria virtuale e questo lo copia nell'HD.

Oltre a questo, chi carica il programma nel sistema deve anche preparare la tabella di corrispondenza. All'inizio dobbiamo mettere tutti i bit P a 0 in modo da dire che tutte le pagine virtuali non sono presenti in memoria fisica ma sono sull'Hard-Disk. Nell'altra parte della tabella dobbiamo scrivere, **nell'ordine in cui sono nella memoria virtuale**, il numero del blocco dell'HD nel quale si trovano. È importante rispettare l'ordine perché ciascuna entrata della tabella è relativa, in ordine, a ciascuna pagina della memoria virtuale (pagina 0 riga 0). Questa tabella è collegata in qualche modo alla MMU. Dopodiché il processore fa partire il programma. Scriviamo in `EIP` l'indirizzo della prima istruzione che supponiamo sia nella pagina 2 e vi scriviamo 0x2000 (2⟶pagina virtuale, 000⟶offset). Dobbiamo anche inizializzare la pila, che supponiamo essere nella pagina virtuale 3; vi scriviamo 4000 poiché la prima `push`, quella che salva `EBP`, scrive 4 byte e se partissimo dall'indirizzo 3FFF dell'ultimo byte, perderemmo quest'ultimo.

La prima cosa che il processore farà, sarà quindi un'operazione di lettura all'indirizzo 2000 (contenuto di `EIP`). Questa operazione verrà intercettata dalla MMU che dividerà in due parti l'indirizzo (2 e 000) e verifica, attraverso la tabella di corrispondenza, se la pagina 2 sia in memoria fisica o in memoria di massa. Vede che sta nel blocco 2 dell'HD (Figura 2.6, frecce 1). La MMU decide di mettere la pagina voluta in una pagina fisica libera, se ve ne sono, e ordina all'HD di copiarcela (Figura 2.6, freccia 2). Decide di trasferire il contenuto della pagina virtuale 2, nella pagina fisica 0. Aggiorna (Figura 2.6, punto 3), inoltre, i valori della tabella di corrispondenza per la pagina in oggetto. Il processore sta sempre aspettando di sapere che cosa c'è in memoria all'indirizzo 2000.

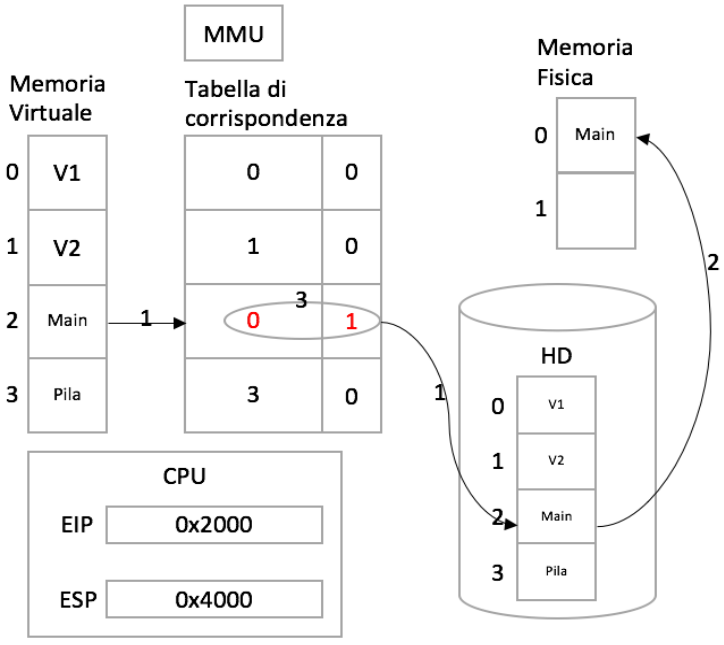


Figura 2.6: Esempio simulazione 2

Quindi, in base alle semplici regole di traduzione viste precedentemente, adesso possiamo completare l'operazione. L'indirizzo fisico a cui si trova la locazione sarà quindi 0000 (0⟶pagina fisica, 000⟶offset). A questo indirizzo fisico avremo effettivamente la `push EBP`.

Il processore esegue l'istruzione: deve sottrarre 4 a `ESP` e dunque il suo nuovo

valore sarà 3FFC. Ora dobbiamo fare un'operazione di scrittura all'indirizzo virtuale 3FFC per memorizzare il contenuto di `EBP`. Questa scrittura viene intercettata dalla MMU che deve verificare se la pagina virtuale in cui vogliamo scrivere sta già in memoria di fisica oppure occorre prelevarla dalla memoria di massa. La MMU scompone l'indirizzo virtuale (3⟶pagina virtuale, FFC⟶offset) e vede tramite confronto con la tabella di corrispondenza, che non sta in memoria fisica ma nel blocco 3 dell'HD (Figura 2.7, frecce 1). La MMU ordina all'HD di copiare la pagina virtuale in una pagina fisica vuota, se ve ne sono (Figura 2.7, freccia 2). Decide di trasferire il contenuto della pagina virtuale 3, nella pagina fisica 1. Aggiorna (Figura 2.7, punto 3) i campi dell'entrata nella tabella di corrispondenza per la pagina in oggetto.

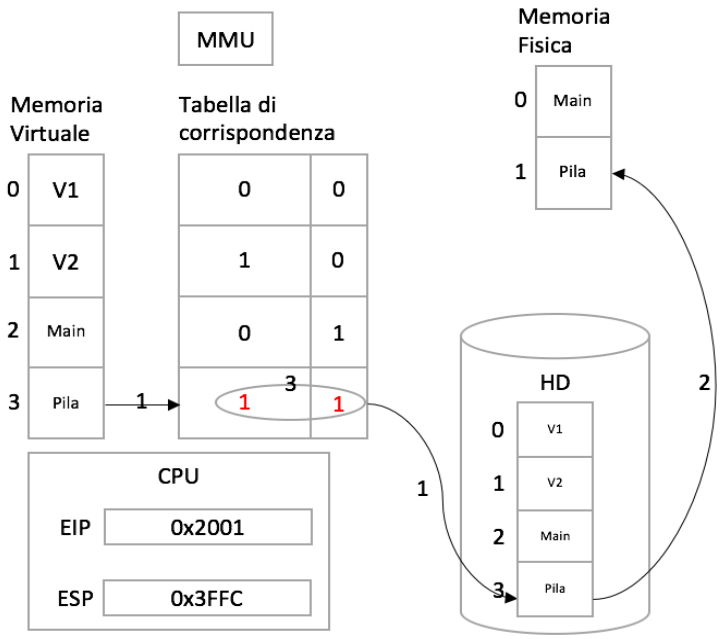


Figura 2.7: Esempio simulazione 3

Il processore sta sempre aspettando di completare l'operazione e per completarla occorre fare una scrittura all'indirizzo fisico che sarà 1FFC (1⟶pagina fisica, FFC⟶offset). Il processore deve anche incrementare `EIP` per passare alla prossima istruzione, quindi, supponendo che la precedente fosse grande 1 byte, il nuovo valore di `EIP` sarà 2001.

Successivamente il processore preleva l'istruzione all'indirizzo 2001. Questa operazione di lettura viene intercettata dalla MMU che scompone l'indirizzo virtuale in due parti (2⟶pagina virtuale, 00⟶offset). La pagina virtuale è la numero 2 che si trova già in memoria fisica, quindi non fa altro che sostituire 2 con 0, leggendo all'indirizzo 0001.

Ad un certo punto, però, questo programma avrà bisogno di accedere all'array V. Quindi il processore genererà indirizzi virtuali che puntano alle pagine virtuali 0 e 1 le quali contengono l'array V. Un'operazione di lettura a questi indirizzi virtuali sarà intercettata dalla MMU che verificherà che la pagina virtuale 0 non è in memoria fisica. Il problema è che adesso la memoria fisica è piena. Quello che dobbiamo fare è scegliere una pagina fisica nella quale mettere la pagina virtuale che ci serve adesso, dobbiamo cioè scegliere una *vittima*. Prima di eliminare il contenuto della pagina

scelta come vittima, occorre salvare il contenuto stesso nella posizione dell'HD in cui si trovava prima di essere caricata (soluzione simile a quella vista per la memoria cache); supponiamo a tal proposito di avere a disposizione un'altra struttura dati che tenga traccia della posizione originaria nell'HD. La pagina scelta come vittima è in genere quella non usata da più tempo. Supponiamo, nell'esempio che la pagina da sacrificare sia la pila (pagina fisica 1).

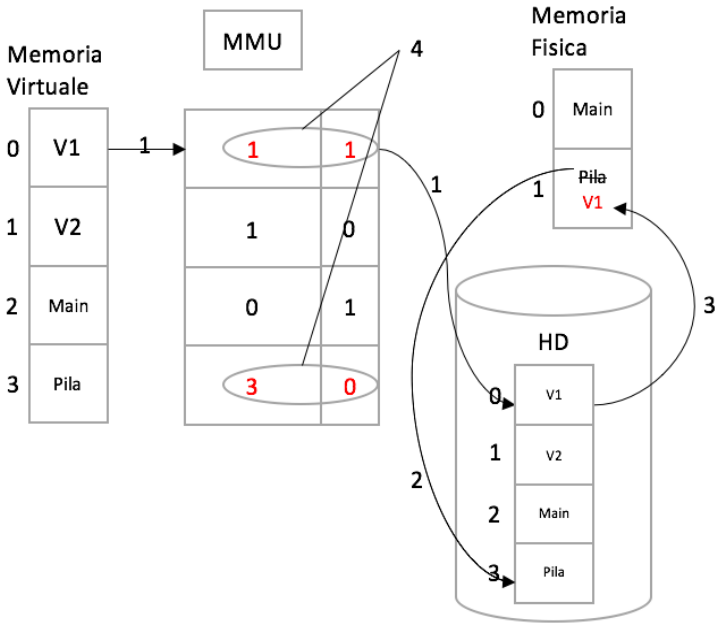


Figura 2.8: Esempio simulazione 4

Il vantaggio di tutto questo sta nel fatto che non occorre modificare il codice del programma in base alla quantità di RAM installata.

2.6 Complicazioni derivanti dall'implementazione semplificata con "*super-MMU*"

Molte complicazioni derivano dal fatto che stiamo chiedendo troppo alla MMU come oggetto hardware. Se dovessimo far fare alla MMU tutto quello che abbiamo detto, la MMU dovrebbe essere un altro computer e, essendo la tabella di corrispondenza (con 1.000.000 di entrate, ogni occorrenza delle quali di 4 byte) di 4MB, dovrebbe possedere anche una RAM di tale dimensione.

Alcune cose che abbiamo supposto faccia la MMU sono semplici (caso della pagina virtuale già in memoria), altre più complesse come quelle in cui viene ordinato all'Hard-Disk di fare delle cose: questo presuppone del software. Quindi il modo più economico di fare tutto questo sarebbe avere un altro computer. Noi, però, abbiamo già un computer e quindi, in realtà, alla MMU facciamo gestire solo il caso più semplice, cioè quello in cui la pagina virtuale cercata si trova già in una pagina fisica (bit P a 1). Se la pagina è presente in memoria fisica, sostituisce i primi 20 bit dell'indirizzo virtuale con i 20 bit che trova nella tabella di corrispondenza in modo da comporre l'indirizzo fisico. Se la pagina non è presente *lancia un'eccezione* al processore che smette di eseguire quel programma e passa ad un altro programma

che serve a recuperare la pagina virtuale assente. Inoltre la tabella di corrispondenza di 4MB si trova già nella RAM che abbiamo, poiché si comporta essa stessa come una RAM. Nella RAM si trova anche il codice del programma appena detto e anche altre strutture dati utili a dare corpo a questo meccanismo (Figura 2.9).

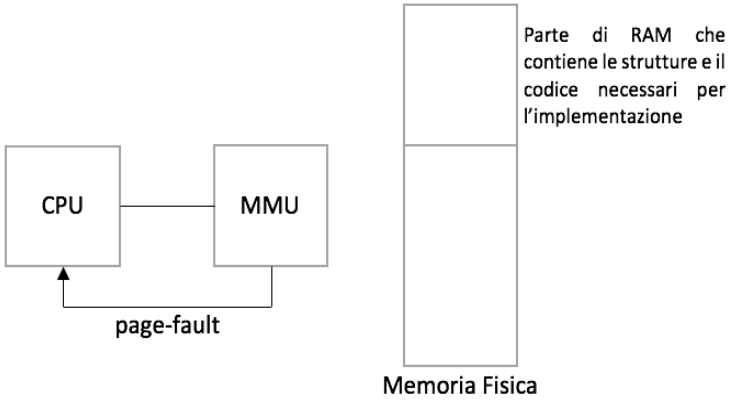


Figura 2.9: Suddivisione di massima della RAM

Quando viene lanciata un'eccezione, il processore compie un po' di azioni: salva in pila alcuni valori, in particolare l'indirizzo contenuto in `EIP`, quello ***corrente*** perché a differenza del meccanismo delle interruzioni, la *routine di page-fault* viene lanciata nel bel mezzo dell'esecuzione di un'istruzione che, una volta sistemate le cose, **deve essere rieseguita**. Quando viene lanciata la routine di page-fault, viene messa in esecuzione una routine software contenuta nella parte di RAM adibita a contenere codice e strutture per dare corpo a questo meccanismo. Questa routine ha come parametro l'indirizzo virtuale della pagina virtuale che ha causato page-fault. La MMU scrive questo indirizzo **virtuale** in un registro speciale del processore che è `CR2`; basta quindi leggere via software il contenuto di questo registro e la routine può proseguire le sue operazioni.

2.7 Riepilogo semplificazioni

Per adesso facciamo due semplificazioni:

- la tabella di corrispondenza è una ed ha 1.000.000 di entrate;
- la MMU si disattiva automaticamente quando non è in esecuzione un programma originario (utente).

2.7.1 Hardware e Software: chi comanda?

Prima di proseguire nell'abbattimento graduale delle semplificazioni fatte al meccanismo della memoria virtuale, occorre capire meglio la relazione che esiste fra l'hardware e ***i*** software.

Chi comanda è il software; l'hardware di per sé non compie nessuna azione (se il processore, ad esempio, non riceve nessuna istruzione, non esegue nessuna istruzione). Disegnare una parabola sul monitor o invertire una matrice sono operazioni che il software impartisce all'hardware di compiere.

Esistono due tipi di software ***utente*** e ***sistema***. Il software utente sono i programmi che noi scriviamo, invece, il software sistema è il sistema operativo (con tutte le sue routine). Fra i due tipi di software chi è che comanda? Comanda il software utente. *Il software di sistema è lì per servire il software utente.* Questo è vero anche se il software sistema è più potente del software utente. Concettualmente possiamo dire che il software sistema si trova allo stesso livello dell'hardware. Il software di sistema nasce dunque per sopperire al problema della realizzazione in hardware (costoso, poco flessibile, complicato) di alcune operazioni.

2.7.2 Comunicazione fra software utente e software sistema

Mentre è in esecuzione il programma utente non è in esecuzione il programma sistema. Quindi come fa il sistema a fare qualcosa per conto del programma utente come se fosse hardware? Il meccanismo delle interruzioni rappresenta soltanto una parte della soluzione al problema poiché, anche in questo caso, o è in esecuzione l'uno o è in esecuzione l'altro; il processore esegue solo un programma per volta. L'idea è quella di farsi dare una mano dall'hardware; l'hardware obbedisce agli ordini del software però possiamo fare in modo che in qualche maniera sappia che ci sono due tipi di software: obbedisce agli ordini del software attualmente in esecuzione ma deve sapere, al tempo stesso, che ci sono degli ordini superiori che arrivano da un software più potente. Il controllo dell'arrivo di ordini superiori deve essere fatto contemporaneamente all'esecuzione del software utente. Il processore quando si trova in modo utente, esegue il programma utente ma guarda anche i comandi che sono stati precedentemente lasciati dal programma sistema.

Una volta che la tabella di corrispondenza è in memoria fisica, essa rappresenta un modo con cui il software di sistema può, mentre è in esecuzione il software utente, influire indirettamente su quest'ultimo. Per esempio: una volta che il software di sistema ha scritto nella tabella di corrispondenza che una pagina virtuale è presente in memoria fisica e si trova ad un certo indirizzo, il programma utente, tornato in esecuzione, farà ignaro i suoi accessi in base a quello che il software sistema aveva precedentemente lasciato. La MMU è in un certo senso "istruita" dal software di sistema.

Un altro modo è quello di dire all'hardware che quando succedono certe cose, deve essere richiamato; passiamo cioè in modo sistema. Le eccezioni sono un esempio di questa seconda tecnica. Con le eccezioni diciamo all'hardware che quando si verifica una situazione eccezionale (per esempio page-fault), si salta al programma di sistema sempre con l'obiettivo, sia chiaro, di tornare a programma utente.

Il terzo modo è quello in cui il programma utente richiama esplicitamente il sistema tramite una *primitiva*.

Quindi il software utente è "influenzato" da alcune informazioni lasciate dal software sistema in alcune strutture dati (come la tabella di corrispondenza, la IDT e la GDT). Queste strutture dati sono lette e scritte sia dal software sistema che dall'hardware. Siccome l'hardware non può essere modificato, il formato delle strutture dati è vincolato dall'hardware stesso; il software si può solo adeguare e deve quindi sapere come sono fatte queste strutture per poter interagire correttamente con esse.

2.8 Entrate della tabella di corrispondenza

Vediamo ora come sono fatte le entrate della tabella di corrispondenza.

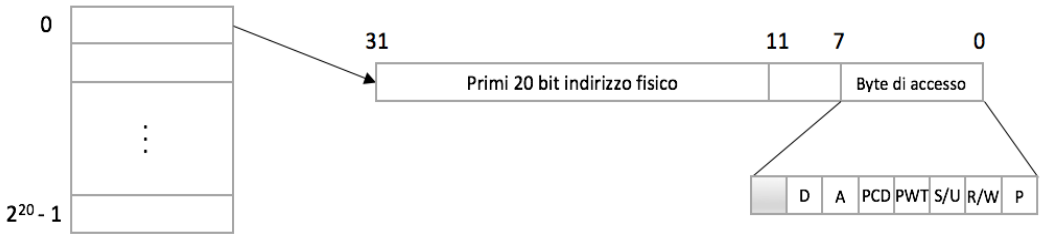


Figura 2.10: Dettaglio di un'entrata della tabella di corrispondenza

Il byte di accesso (8 bit meno significativi) contiene un po' di informazioni riguardo la pagina virtuale relativa a quell'entrata. **Tutti i bit del byte di accesso sono scritti dal software di sistema**.

Il bit P (bit di presenza) quando è settato significa che la pagina virtuale si trova in memoria fisica; se è resettato significa che la pagina virtuale si trova in memoria di massa. Se il bit P è a 1, nei 20 bit più significativi c'è il numero di pagina fisica (cioè i 20 bit che la MMU deve mettere al posto dei 20 bit virtuali generati dalla CPU). La MMU fa qualcosa solo se la pagina è presente in memoria fisica altrimenti lancia un'eccezione di page-fault delegando il problema al software di sistema. La MMU si trova in una posizione, fisicamente parlando, privilegiata poiché vede tutti gli accessi in memoria.

I bit PWT (*Page Write Through*) e PCD (*Page Cache Disable*), sono due bit tramite cui il programmatore di sistema può decidere cosa deve succedere rispetto alla cache. **La cache si trova tra la MMU e la memoria fisica**. La MMU, quando la pagina virtuale è presente in memoria fisica, se il bit PCD è a 1, per quell'accesso disabilita la cache. Se il bit PWT è a 1 viene forzato l'utilizzo del metodo write through (scrive sia in cache che in memoria fisica). Dove sta l'utilità del bit PCD? C'è un caso in cui la cache non serve: l'I/O. Anche se esiste uno spazio di ingresso/uscita, questo non basta poiché i produttori di periferiche sono molti per cui alcuni indirizzi fisici corrispondono a registri di periferiche in memoria. In genere sono indirizzi molto alti.

Il bit U/S (User/System) ci dice se quella pagina è accessibile dal programma utente oppure no.

Il bit R/W (Read/Write) dice se quella pagina può essere sia letta che scritta oppure solo letta.

I bit D (Dirty) e A (Accessed), a differenza degli altri possono essere scritti dalla MMU. Rappresentano informazioni che la MMU raccoglie e che poi servono al software di sistema che va in esecuzione quando occorre scegliere una pagina vittima nel caso in cui la memoria fisica sia piena. Queste informazioni vengono raccolte mentre è in esecuzione il programma utente. Il bit A viene messo a 1 quando viene fatta un'operazione di lettura su quella pagina. Il bit D viene messo a 1 quando la pagina viene scritta. Quest'ultima informazione, in particolare, è molto utile in quanto se la pagina virtuale x deve essere rimossa da una pagina fisica ed ha il bit D a 0, non importa che venga ricopiata in memoria di massa poiché il suo contenuto non è stato alterato.

2.9 Prime analisi sulla routine di *page-fault*

Per sapere quale pagina eliminare abbiamo pochissima informazione: un solo bit. Ma possiamo sfruttare questa cosa in modo migliore riportando a 0 tutti i bit A periodicamente. Quindi è anche preferibile scegliere come vittima la pagina virtuale che ha il bit D a 0 perché ci consente di risparmiare tempo in quanto non va ricopiata nell'HD (operazione molto costosa). Un bit del genere (D) converrebbe averlo anche nella cache.

A queste condizioni, quindi, ogni accesso in memoria si trasforma in due accessi in memoria

1. quello per consultare la tabella di corrispondenza;
2. quello per compiere veramente l'accesso.

Nella memoria fisica:

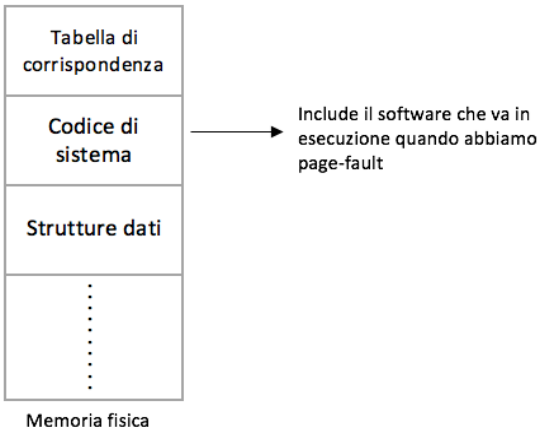


Figura 2.11: Memoria fisica: agli indirizzi più bassi troviamo software di sistema e strutture dati

La MMU scrive in `CR2`, un registro speciale del processore, l'indirizzo virtuale che non è riuscita a tradurre e il software di sistema deve risolvere la situazione. Poiché la pagina virtuale non è presente in memoria fisica, nei primi 20 bit più significativi dell'entrata corrispondente ci sarà il numero del blocco della memoria di massa dove si trova la pagina virtuale.

2.9.1 Il problema della ricollocazione della pagina vittima nella memoria di massa. Primo accenno sui *descrittori di pagina fisica*

Per sapere quale pagina eliminare, occorre *scorrere* la tabella alla ricerca dell'entrata della pagina che abbia il bit A a 0. Se ne scegliamo una che ha il bit D a 1 dobbiamo, prima di sovrascriverla, copiarla nella sua posizione originale nella memoria di massa. La posizione originale, però, non la conosciamo più dal momento che abbiamo sovrascritto quest'ultima con i primi 20 bit dell'indirizzo fisico all'interno dell'entrata. È ragionevole, perciò disporre di un'altra struttura dati per mantenere questa informazione. Conviene ricordarsi da dove preleviamo le pagine virtuali perché se non lo facessimo, nel caso fortunato, e abbastanza frequente, in cui

la pagina non viene modificata (D=0) occorrerebbe comunque ricopiarla in memoria di massa.

Ci conviene quindi avere una tabella con dei *descrittori* associati ad ogni pagina fisica. Questi descrittori ci danno informazioni sulla pagina fisica come, ad esempio, il blocco della memoria di massa dove si trovava la pagina virtuale che vi era stata posta. Possiamo fare anche delle statistiche un po' più dettagliate con il bit A, per esempio contare quante volte lo vediamo a 1. Avere un descrittore può essere utile anche per altri scopi: supponiamo che la pagina numero 1000 sia stata scelta come vittima; per toglierla occorre mettere a 0 il bit P relativo alla pagina virtuale messa in memoria fisica, c'è un metodo costosissimo che è quello di scorrere tutta la tabella di corrispondenza alla ricerca dell'entrata i cui primi 20 bit del suo contenuto siano uguali all'indirizzo della pagina in memoria fisica, oppure possiamo scrivere nel descrittore l'indirizzo virtuale che aveva quella pagina prima di essere posta in memoria fisica.

Definiamo una struttura dati interamente software, un ***array di descrittori di pagina fisica***. Ogni descrittore di pagina fisica (uno per ogni pagina) contiene un bit che indica se è libera o occupata. Se è occupata, della pagina virtuale che vi è posta vogliamo sapere:

1. contatore di statistiche;
2. indirizzo virtuale;
3. indirizzo della memoria di massa (posizione originaria della pagina virtuale).

Una volta individuata la pagina con le statistiche peggiori, bisogna liberare la pagina fisica eliminando la pagina virtuale in essa contenuta. Tramite il descrittore di pagina fisica possiamo più facilmente individuare i dati necessari per ricollocare la pagina virtuale, che in essa risiedeva, nella posizione originale all'interno della memoria di massa.

2.10 Abbattimento della prima semplificazione: dalla tabella di corrispondenza alle *tabelle delle pagine*[1]

La tabella di corrispondenza non è un'unica tabella. Essa è infatti spezzata in modo che alcune parti di essa possano non essere allocate se non utili al momento. Questo perché esiste una tabella di corrispondenza per ogni processo e considerando che i processi sono molti e che ogni tabella occupa 4MB, lo spazio occupato diventerebbe molto. Poiché un processo, delle 4GB pagine virtuali ne usa solo una parte per volta, possiamo allocare al bisogno le parti di tabella di corrispondenza che ci occorrono. A tale scopo, la tabella di corrispondenza, per ogni processo, viene suddivisa in 1024 ***tabelle delle pagine*** che complessivamente ricompongono l'intera tabella. Ogni tabella delle pagine è grande 4KB. Il fatto che la tabella delle pagine sia grande quanto una pagina non ha niente a che vedere con le pagine in sé, ma come vedremo questa cosa ci tornerà utile. L'idea è quindi quella che possiamo evitare di avere tutte le tabelle in memoria. Come possiamo poi sapere qual è la tabella delle pagine da allocare? Questa informazione può essere ottenuta aggiungendo un'altra tabella di 1024 entrate chiamata ***direttorio***.

[1] Questo paragrafo richiede qualche nozione preliminare di multiprogrammazione.

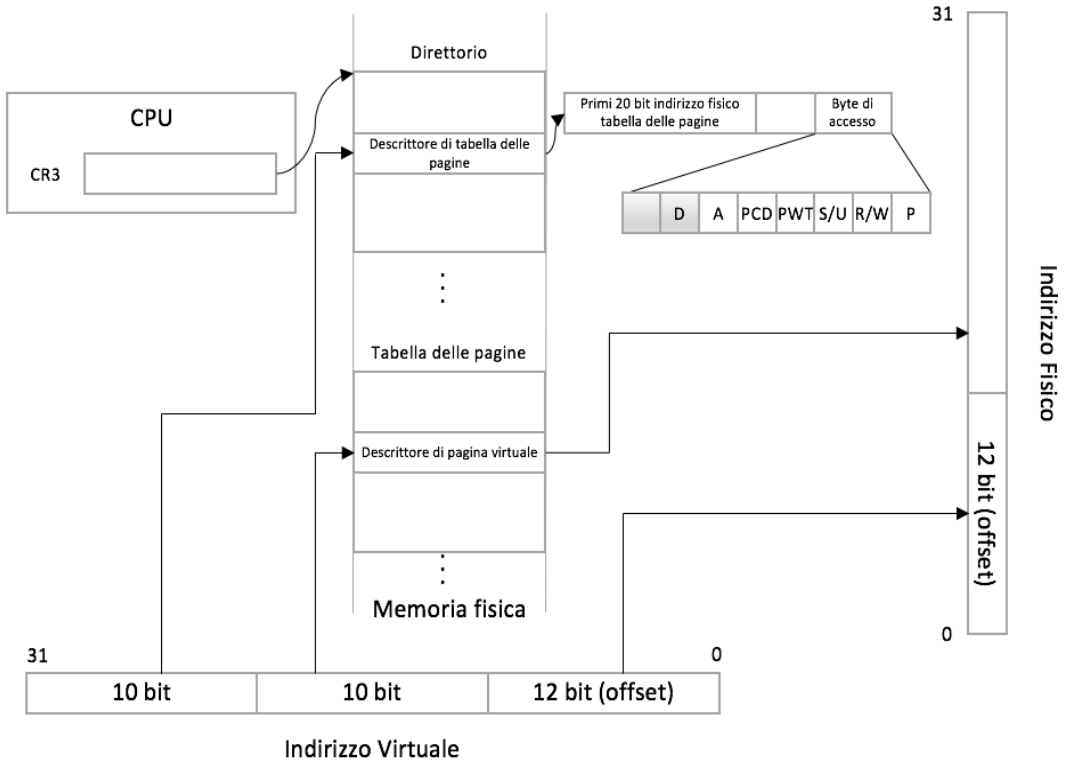


Figura 2.12: Visione d'insieme del sistema tabelle delle pagine-direttorio

Come si nota dalla figura, le entrate del direttorio sono molto simili alle entrate della tabella di corrispondenza (e quindi delle tabelle delle pagine). Da specificare, però, che il bit D nel descrittore delle tabelle delle pagine non è significativo per la tabella delle pagine a cui punta; il suo valore infatti non indica se la tabella delle pagine è stata modificata o meno. Quando la tabella delle pagine si trova in memoria fisica, essa sarà accessibile per mezzo di un descrittore di pagina virtuale presente in un'altra tabella delle pagine: il bit D di quel descrittore sarà quello che ci dice se la tabella delle pagine è stata modificata o meno.

La MMU quindi, introdotte queste nuove strutture, con l'indirizzo virtuale fa le seguenti cose:

1. con i 10 bit più significativi individua il descrittore della tabella delle pagine nel direttorio. In questa entrata scopre subito se quel pezzo di tabella è presente oppure no (se non c'è, di conseguenza non c'è nemmeno la pagina virtuale cercata). Se il pezzo di tabella è presente, trova l'indirizzo di una tabella delle pagine;

2. con i secondi 10 bit più significativi trova l'entrata della tabella delle pagine;

3. le due precedenti operazioni, insieme, compiono la stessa cosa che veniva fatta prima con l'intera tabella di corrispondenza;

4. con gli ultimi 12 bit individua l'offset del byte nella pagina fisica.

Ogni pezzo della tabella si occupa di una cosiddetta *macropagina* grande 4MB. Se manca la tabella relativa a quella macropagina, manca tutta la macropagina.

Ora abbiamo che ogni accesso in memoria si trasforma in realtà in tre accessi in memoria:

1. un accesso nel direttorio;
2. un accesso nella tabella delle pagine;
3. un accesso nella pagina fisica.

Le informazioni presenti nell'entrata del direttorio sono informazioni aggregate sulla macropagina. Per esempio, disabilitando la scrittura in una determinata entrata nel direttorio, la disabilitiamo di fatto per tutte le pagine virtuali della macropagina.

A questo punto occorre però gestire, oltre alle pagine virtuali, anche le tabelle delle pagine e il direttorio. Il fatto che queste strutture siano tutte grandi 4KB ci permette di avere semplificate le cose in quanto possediamo già l'hardware che ci serve poiché anche le pagine virtuali sono grandi 4KB. Direttorio, tabelle delle pagine e pagine virtuali sono tutte entità grandi 4KB e possono essere memorizzate indifferentemente, le une dalle altre, nelle pagine fisiche della memoria fisica.

Quindi se ogni processo aveva la sua tabella di corrispondenza, dopo quanto visto, possiamo dire che ogni processo ha il suo direttorio le entrate del quale puntano alle sue 1024 tabelle delle pagine.

2.11 Esempio: simulazione del meccanismo con le nuove aggiunte

Riprendiamo l'esempio di programma che somma gli elementi di un array

```
1 const int N = 1000000;
2 int buff[N] = {...};
3 int main(){
4         int sum = 0;
5         for(int i = 0; i<N; i++)
6                 sum += buff[i];
7         return sum;
8 }
```

Questo programma verrà compilato e consisterà di una sezione `text` e di una sezione `data`. Anche se la sezione `text` occupasse meno di una pagina, converrebbe comunque tenerla separata dalla sezione `data` poiché, a differenza della sezione `text`, le pagine relative alla sezione `data` possono essere modificate. Se perciò mettessimo insieme le due parti e mettessimo il bit R/W a 0 per la pagina in cui si trova `text`, otterremmo che i dati che sono nella stessa pagina di `text` non potranno essere modificati.

La pila parte dal fondo dello spazio di indirizzamento (cioè viene riempita a partire dagli indirizzi alti).

Prima di essere eseguito, il programma deve essere posto tutto in memoria di massa.

Prima di far partire il programma deve essere anche predisposta la tabella di corrispondenza per la MMU. Abbiamo visto che la tabella di corrispondenza è composta da un direttorio le cui entrate puntano alle 1024 tabelle delle pagine. Inoltre, grazie al direttorio, abbiamo la possibilità di specificare quali tabelle delle pagine sono già state caricate in memoria fisica e quali no. Quindi possiamo avere anche le tabelle delle pagine in memoria di massa. La sezione dati non sarà grande esattamente 4MB perché N è 1.000.000 che è diverso dal "milione informatico" 2^{20} (1.048.576).

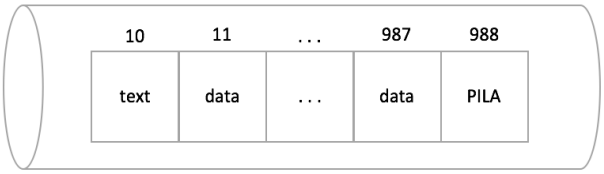


Figura 2.13: Il programma caricato nell'Hard-Disk

Le pagine dei dati sono 977 perché dobbiamo distribuire 4MB non informatici su 4KB informatici e quindi abbiamo 4.000.000 : 4096 = 977 circa.

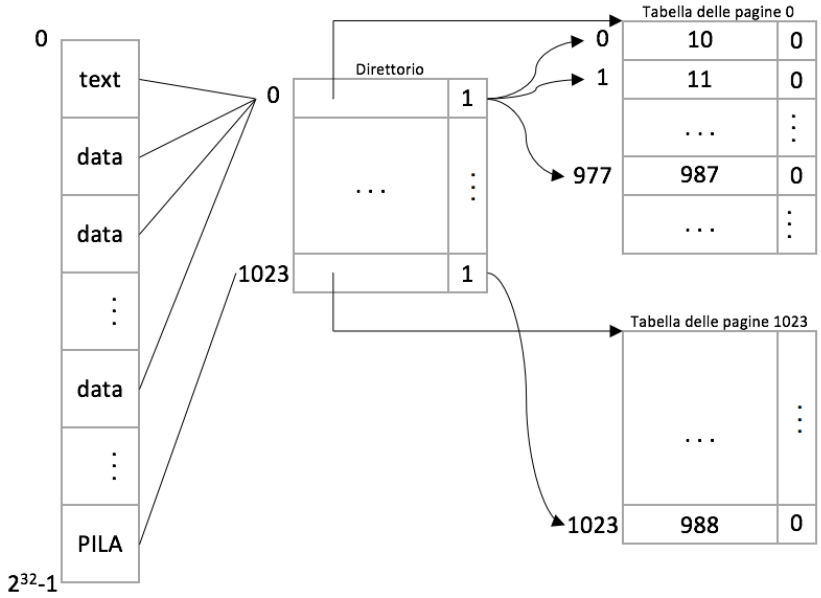


Figura 2.14: Predisposizione del sistema direttorio-tabelle delle pagine

Le prime 1024 pagine della memoria virtuale saranno gestite dalla prima occorrenza del direttorio. In queste 1024 pagine ci stanno `text` (1 pagina) e `data` (977 pagine). La pila si trova in fondo alla memoria virtuale e sarà quindi gestita dall'ultima occorrenza del direttorio. Pertanto occorrerà caricare solo due tabelle delle pagine.

La CPU quando tenterà di accedere ad uno qualunque dei byte nelle pagine, genererà un *indirizzo virtuale* che verrà intercettato dalla MMU. Di questo indirizzo prende i primi 10 bit e li usa come indice d'entrata nel direttorio. Nell'occorrenza del direttorio trova, se già presente in memoria fisica, l'indirizzo fisico della tabella delle pagine. A questo punto usa i secondi 10 bit dell'indirizzo virtuale per accedere all'entrata della tabella delle pagine relativa alla pagina virtuale cercata. Se la pagina è già presente in memoria fisica, nell'entrata della tabella delle pagine, trova i primi 20 bit del suo indirizzo fisico.

Quindi, per esempio, per individuare la pagina contenente la sezione `text`, guarderemo prima nel direttorio, se la tabella delle pagine è presente guarderemo nell'occorrenza relativa alla pagina virtuale contenente la sezione `text` e vedremo che essa è assente e si trova nel blocco 10 in memoria di massa. Lo stesso per la prima pagina relativa alla sezione *data*: vedremo che è assente e che si trova in memoria

di massa nel blocco 11.

Per tutti i primi 4MB i primi 10 bit dell'indirizzo virtuale saranno a 0 e quindi faranno sempre riferimento alla prima entrata del direttorio.

Di tutte le entrate che non abbiamo usato sia del direttorio che delle tabelle delle pagine, cosa ne facciamo? Dipende da che cosa vogliamo che accada quando il programma prova ad accedere a questi indirizzi che non puntano a nulla di significativo. Molti sistemi, per esempio, estendono la pila automaticamente in modo da non lasciare "indirizzi scoperti". Altrimenti un'altra possibilità è quella di interrompere il programma nel momento in cui questo tenta di accedere a quegli indirizzi; per farlo occorre mettere un valore speciale nelle entrate in modo che quando si verifica page-fault capisca che non c'è da caricare una pagina in quel caso.

Grazie alla presenza del direttorio, possiamo copiare anche le tabelle delle pagine in memoria di massa.

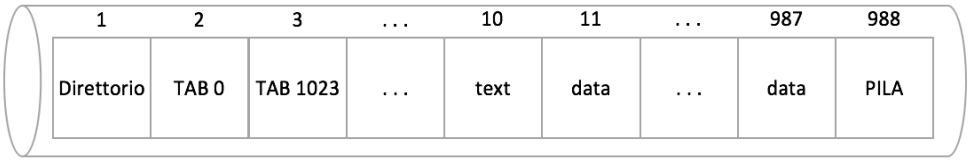


Figura 2.15: Costruzione del sistema direttorio-tabelle delle pagine

Fatto questo, sappiamo anche cosa scrivere nelle occorrenze del direttorio:

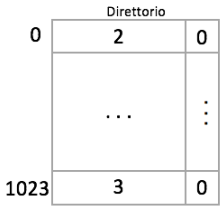


Figura 2.16: Direttorio

Poiché ogni processo ha il suo direttorio, anche il direttorio va messo in memoria di massa, supponiamo nel blocco 1 (il numero del blocco deve essere noto a priori).

Quindi, prima di far partire il programma, "qualcuno" deve preparare con tutte queste entità la memoria di massa. A questo punto si esegue il programma.

Nella parte di memoria fisica M1 (figura 2.17), non utilizzabile dal programma perché serve per gestire anche la memoria virtuale, dovremo caricare:

- il codice della routine di page fault
- l'array di descrittori di pagina fisica
- il codice di una routine di inizializzazione (parte in modalità sistema e con la paginazione disabilitata).

La routine di inizializzazione, inizializza i descrittori di pagina fisica scrivendo che tutte le pagine sono libere e poi deve caricare il direttorio che deve essere ***sempre*** in memoria fisica quando il programma è in esecuzione. Poiché il direttorio è grande 4KB può essere caricato in una pagina fisica. Poi bisogna dire alla MMU a che indirizzo si trova il direttorio; questo lo facciamo scrivendo nel registro speciale del

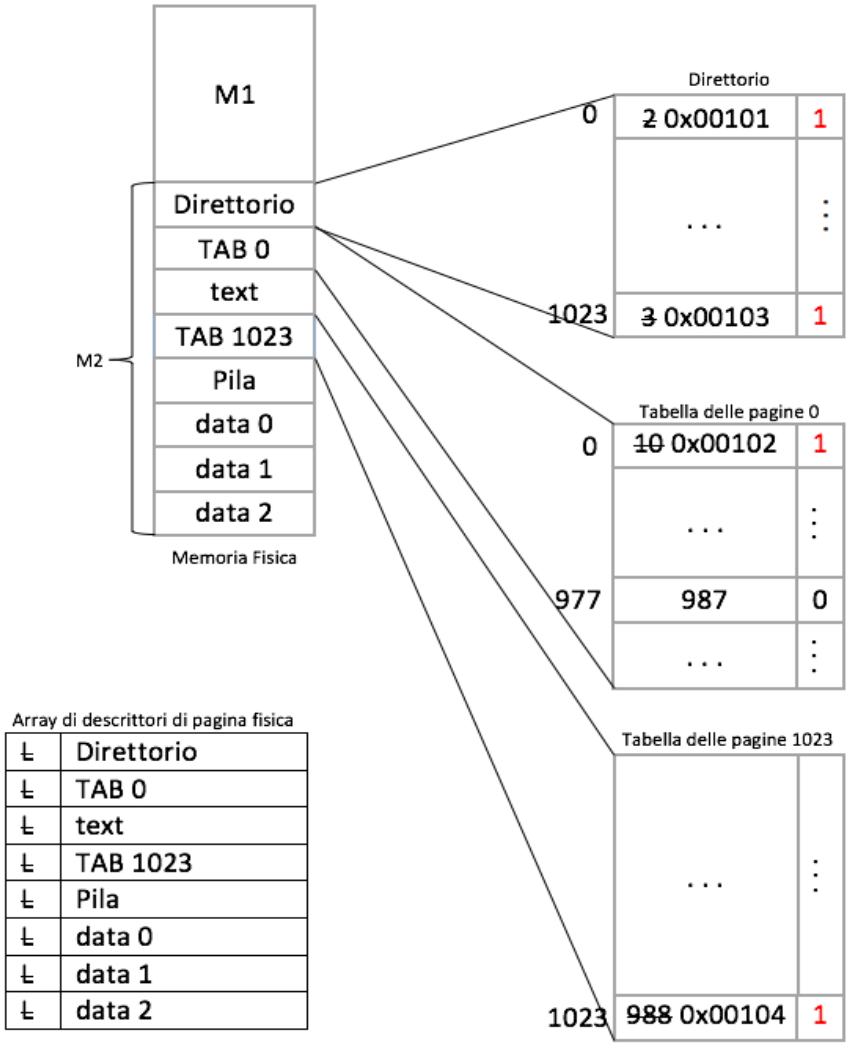


Figura 2.17: Simulazione

processore CR3: ***questo è l'unico registro del processore che contiene sempre un indirizzo fisico anche quando è attiva la paginazione***. In questo esempio supponiamo che M1 sia grande 1MB informatico, quindi scriviamo in `CR3` 2^{20} che corrisponde all'indirizzo del primo byte del direttorio. A questo punto la routine di inizializzazione può abilitare la memoria virtuale, cioè abilitare la paginazione e si fa mettendo a 1 il bit 31 del registro speciale del processore `CR0`.

Adesso la MMU comincia a lavorare soltanto quando il processore passerà in modo utente (semplificazione). Si passa in modo utente caricando in `EIP` l'indirizzo iniziale del programma che vogliamo eseguire (indirizzo virtuale).

La CPU vuole fare una lettura all'indirizzo 0; la lettura viene intercettata dalla MMU che scompone l'indirizzo virtuale; prende i primi 10 bit, accede alla prima entrata del direttorio e ci trova scritto che la corrispondente tabella delle pagine è assente e si trova in memoria di massa nel blocco 2. C'è il bit P a 0 e viene lanciata un'eccezione di page-fault. Questa eccezione, la numero[2] 14, dice al processore di interrompere il programma che è correntemente in esecuzione e di saltare alla routine di page-fault a cui passa il controllo (supponiamo che sia a livello sistema). Prima di saltare alla routine di page-fault, la MMU scriverà nel registro speciale del processore `CR2` l'indirizzo virtuale 0 (cioè l'indirizzo che non era riuscita a tradurre). La routine di page-fault deve risolvere la situazione che ha generato l'eccezione ovvero permettere alla MMU di tradurre quell'indirizzo virtuale in modo corretto.

Occorre caricare in memoria TAB 0. Nel direttorio mettiamo il bit P relativo alla tabella, a 1 e ci scriviamo anche l'indirizzo fisico della tabella che si ottiene sommando 4KB all'indirizzo iniziale del direttorio. Modifichiamo anche l'array di descrittori di pagina fisica rendendo occupata l'occorrenza corrispondente. Quindi il processore ripete l'istruzione che aveva causato page-fault e stavolta trova la tabella. Con i secondi 10 bit dell'indirizzo virtuale viene individuata l'entrata della tabella (la prima nel nostro caso) e vediamo che la pagina che stiamo cercando si trova in memoria di massa nel blocco 10; viene quindi generata un'eccezione di page-fault e si opera come nel caso precedente (in realtà la pagina viene caricata nello stesso momento in cui si risolve l'eccezione causata dall'assenza della tabella). Dopo aver caricato la pagina `text`, aggiorniamo l'occorrenza relativa alla pagina stessa mettendo il bit P a 1 e sostituendo il numero del blocco con i primi 20 bit dell'indirizzo fisico della pagina.

Arrivato in `text` leggerà la prima istruzione del programma (che sarà la consueta `pushl EBP`); questa istruzione arriva al processore che la esegue sottraendo 4 a `ESP` e scrive all'indirizzo che ottiene il contenuto di `EBP`. `ESP` andava inizializzato con 0 in modo che quando si sottrae 4 la prima volta otteniamo FFFFFFFC.

A questo punto il processore vuole fare una lettura all'indirizzo virtuale FFFFFFFC, la MMU lo intercetta e lo scompone. Con i primi 10 bit accede all'ultima entrata del direttorio in cui vede che la tabella non è presente e si trova in memoria di massa nel blocco 3. La MMU genera quindi un'eccezione di page-fault e vengono ripetute le stesse azioni di prima. Scrive in `CR2` FFFFFFFC che è l'indirizzo virtuale che non è riuscita a tradurre. Anche in questo caso la pagina cercata è assente e carica quindi anche la pagina che si trova nel blocco 988 (cioè la pila). Per queste entità caricate in memoria fisica vengono aggiornati l'array di descrittori di pagina fisica e i relativi descrittori di pagina virtuale nelle entrate della tabella delle pagine.

[2]sapere il numero dell'eccezione o dell'interruzione (hardware o software), ci serve per prelevare la prima istruzione della routine ad essa collegata all'interno della IDT.

Quando viene generata un'eccezione, tutti gli effetti dell'istruzione lasciata a metà vengono annullati (`ESP` riparte da 0, non da FFFFFFFC). L'indirizzo fisico dove verrà scritto `EBP`sarà quindi 0x00104FFC.

Ad un certo punto il programma tenterà di leggere dal buffer dovendo caricare tutte le relative pagine. Tutto prosegue in questo modo finché la memoria fisica non è piena e non bisogna fare un rimpiazzamento.

Quando occorre caricare DATA 3, la memoria fisica sarà piena e quindi andrà rimossa una pagina.

2.12 Scelta della pagina vittima: *LFU* e *LRU*

Occorre avere un criterio nel rimuovere una pagina virtuale attualmente in memoria fisica. Come già accennato, non possiamo sceglierne una a caso.

Non possiamo rimuovere il direttorio ed è anche poco conveniente rimuovere uno dei pezzi della tabella di corrispondenza. Nei descrittori di pagina fisica abbiamo un campo booleano con il quale marchiamo i contenuti che non possono essere rimossi, diciamo cioè che la pagina virtuale posta in quella pagina fisica è *residente*. Per le altre pagine non marcate occorre fare una scelta. Questa scelta viene fatta dalla routine di page-fault in base al valore di un contatore presente in ogni descrittore di pagina fisica nell'array; questo valore riflette una determinata statistica di utilizzo. Vi sono due tipi di statistiche:

1. **LFU** (*Least Frequently Used*): minor frequenza di utilizzo
2. **LRU** (*Least Recently Used*): maggior intervallo di tempo trascorso dall'ultimo utilizzo.

Queste informazioni devono essere in un qualche modo raccolte dalla MMU; quello che può fare è però solo mettere il bit A a 1 e azzerarlo periodicamente.

Al verificarsi di certi eventi va in esecuzione una routine per le statistiche che esamina il bit A di ogni descrittore di pagina virtuale, effettua la statistica e riporta il bit a 0.

Se la statistica è di tipo LFU, il contatore è semplicemente una *variabile di conteggio* che viene incrementata se il corrispondente descrittore di pagina virtuale ha il bit A ad 1.

Se la statistica è di tipo **LRU**, il contatore è un *registro a scorrimento*: la routine inserisce nella posizione più significativa il valore del bit A del corrispondente descrittore di pagina virtuale.

La pagina da rimuovere sarà quella con le statistiche peggiori. Una tabella delle pagine, inoltre, non potrà avere statistica peggiore delle pagine virtuali, allocate in memoria fisica, a cui punta.

2.13 Abbattimento della seconda semplificazione: la MMU, attivata la paginazione, è sempre attiva

Abbattiamo anche la seconda semplificazione che avevamo fatto inizialmente cioè che la MMU sia attiva solo quando il processore si trova in modo utente. La MMU è, in realtà, sempre attiva e, in particolare, lo è anche quando il processore esegue quel software con cui abbiamo implementato quelle operazioni troppo costose da realizzare in hardware. Quali problemi causa questa cosa? La routine di inizializzazione

ha un punto critico che è il momento in cui attiviamo la paginazione settando il bit 31 del registro speciale CR0 (attenzione: la paginazione sarà attiva dall'istruzione successiva[3]). Quindi se vogliamo attivare la paginazione, ad un certo punto dovremo fare qualcosa del tipo:

```
1 movl    %cr0, %eax
2 orl     $0x80000000, %eax
3 movl    %eax, %cr0 #la paginazione si attiva dopo questa istruzione.
```

Quindi la MMU di per sé non distingue se il processore si trova in modo sistema o in modo utente.

Un'altra criticità è messa in risalto nel momenti in cui ci chiediamo se, quando la MMU è attiva, sia possibile o meno accedere a tutti gli indirizzi della memoria fisica.

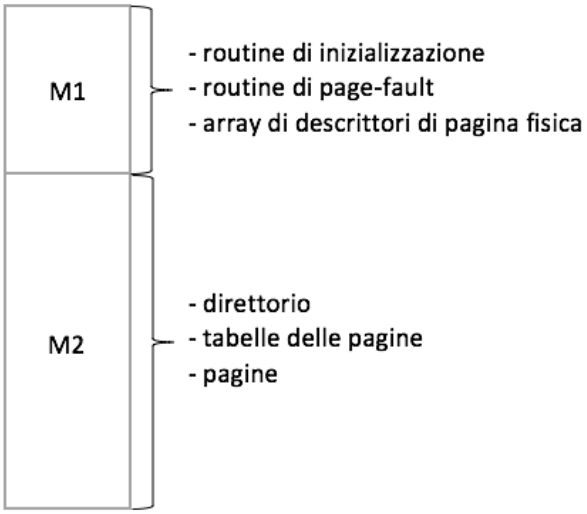


Figura 2.18: Suddivisione della Memoria Fisica

La raggiungibilità o meno di un indirizzo dipende da quali tabelle delle pagine sono allocate. Gli indirizzi fisici sono infatti *funzione di quelli virtuali*. Se, infatti, non è allocata la tabella delle pagine tramite cui si potrebbe raggiungere quello specifico indirizzo fisico, quell'indirizzo fisico non può essere in alcun modo raggiunto. Inoltre la memoria fisica accessibile è solo quella dove sono memorizzate le pagine che fanno parte del programma utente poiché le tabelle delle pagine non possono riferire la parte di memoria occupata dal software di sistema (M1). La routine di page-fault vorrebbe, però, poter accedere a tutto lo spazio di indirizzamento della memoria fisica. Quindi quello che dobbiamo fare è trovare un modo per ottenere quello che avevamo prima pur avendo la paginazione attivata, cioè indirizzo virtuale prodotto dalla CPU = indirizzo fisico reale. Questa cosa possiamo ottenerla anche se l'MMU è attiva purché programmiamo le tabelle delle pagine con le trasformazioni che vogliamo ottenere (questa operazione va fatta prima di attivare la MMU).

[3]questo vuol dire che non è il *boot-strap* ad attivare la paginazione: finché non viene settato il bit 31 del registro speciale `CR0` la paginazione non è attiva e gli indirizzi saranno solo fisici.

2.13.1 Risoluzione al problema della non-accessibilità ad alcuni indirizzi della memoria fisica: la *finestra FM*

Occorre "rubare" alla memoria virtuale uno spazio di uguali dimensioni alla capienza della memoria fisica. Per questo spazio diciamo alla MMU di tradurre gli indirizzi virtuali in se stessi. Lo spazio di memoria virtuale rimasto, verrà utilizzato con le stesse modalità di prima. Questo work-around serve esclusivamente quando il processore si trova in modo sistema; quando il processore si trova in modo utente, viene usato lo spazio virtuale rimasto a disposizione. Questo ha però un limite (ed è la ragione per cui conviene avere un processore a 64 bit) e consiste nel fatto che se abbiamo una memoria fisica molto grande, lo spazio di memoria virtuale che rimane a disposizione del programma utente si riduce notevolmente. Quindi se abbiamo un processore a 32 bit, questa soluzione continua a risultare conveniente finché la memoria fisica ha dimensioni inferiori a 1GB. Lo spazio rubato alla memoria virtuale è chiamato ***finestra FM***. Essa agisce sostanzialmente come una *finestra* attraverso la quale si vede la memoria fisica così com'è.

La finestra FM deve essere uno di quegli oggetti di cui il programma utente non sa nulla; non deve nemmeno potervi accedere in modo da evitare eventuali azioni distruttive.

Come facciamo ad impedire che, mentre è in esecuzione, il programma utente possa accedere agli indirizzi ai quali non deve accedere? Occorre prendere in considerazione il bit U/S: ve ne è uno per ogni descrittore di pagina virtuale. Se la configurazione del bit dice che la pagina è accessibile solo a livello sistema, il processore si trova a livello utente e si tenta di accedere ad una di quelle pagine, non viene fatto l'accesso e viene lanciata un'eccezione.

La finestra FM può essere ricavata in un qualunque punto della memoria virtuale. Linux, per esempio, la ricava sul fondo della memoria virtuale. È più intuitivo metterla sulla cima perché gli indirizzi virtuali risulteranno essere identici a quelli fisici ma, poiché la traduzione è stabilita da noi, altri punti di posizionamento della finestra sono del tutto equivalenti e non creano problemi.

Questa operazione **deve** essere compiuta prima di attivare la paginazione altrimenti dopo non è più possibile farlo.

2.13.2 Esempio: preparazione tabelle per la finestra FM

Parte della routine di inizializzazione:

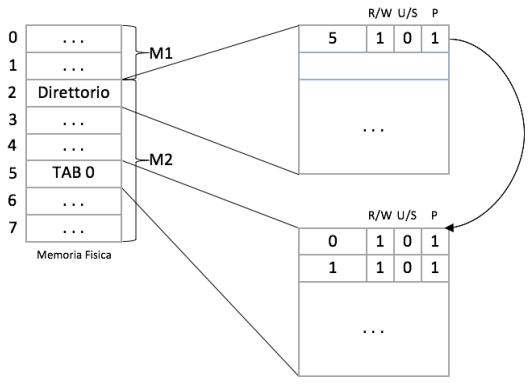


Figura 2.19: Suddivisione della Memoria Fisica

Parte di routine di page-fault:

```
1 movl    %cr3, %ebx #copia l'indirizzo fisico del direttorio
2                in %ebx
3 movl    %cr2, %ecx #copia l'indirizzo virtuale non tradotto
4                in %ecx
5 shrl    $22, %ecx #prende i 10 bit piu' significativi e li mette
6                in posizione meno significativa e vengono usati
7                come indice per accedere al direttorio
8 *movl   (%ebx,%ecx,4), %eax #accede alla tabella voluta
9 testl   $1, %eax #verifica che il bit di presenza sia a 1
10 jnz    tab_presente #se e' presente salta
11        #caricare la tabella se non presente
12
13 tab_presente:
14 movl    %cr2, %ecx #copia nuovamente l'indirizzo virtuale non tradotto
      in %ecx
15 andl    $0x003FFFFF, %ecx #si azzerano i primi 10 bit
16 shrl    $12, %ecx #secondi 10 bit piu' significativi in %ecx
17 movb    $0, %al #negli 8 bit meno significativi del descrittore si
     trova il byte di accesso
18 *movl   (%eax,%ecx,4), %eax #indirizzo fisico della pagina
19
20 *in queste istruzioni abbiamo assunto che sia presente
21 la finestra FM e che cominci a partire dall'indirizzo 0
22 della memoria virtuale.
```

2.14 Buffer dei descrittori di pagina virtuale: TLB

Dal punto di vista hardware c'è un ulteriore oggetto che si chiama **TLB** (*Translation Lookaside Buffer*) e che serve soltanto a risolvere problemi legati alle prestazioni. Abbiamo visto che, con il meccanismo della memoria virtuale, ogni accesso in memoria si traduce in almeno 3 accesi in memoria e se volessimo essere precisi sono anche più di 3 in quanto ne dobbiamo fare alcuni anche per scrivere nei vari byte di accesso al fine di modificare il valore, quando necessario dei bit A e D. Per evitare, o meglio per ridurre, la quantità di accessi, occorre inserire una cache. Questa cache è il TLB. Non va confusa con la cache della memoria fisica poiché **il TLB fa parte della MMU**. È una memoria veloce, accessibile in un solo ciclo di clock, in cui la MMU si mantiene una cache delle traduzioni fatte, cioè una cache dei descrittori di pagina virtuale che si trovano dentro le tabelle. È realizzata come una **cache associativa a insiemi**.

Il TLB è formato da 3 blocchi: R che serve per il rimpiazzamento (*pseudo*-LRU), T e D.

Vi si accede sempre attraverso l'indirizzo virtuale generato dal processore. I 20 bit più significativi vengono scomposti in due parti: i 17 più significativi costituiscono l'etichetta da confrontare con il tag presente nei 4 campi tag (T1, T2, T3, T4), i restanti 3 bit fungono da indice per selezionare la linea di cache. Nel campo tag, oltre all'etichetta vi sono anche il bit di validità V, il bit dirt D, il bit User/System U/S e il bit Read/Write R/W.

Una volta trovata la corrispondenza basta prendere i relativi 20 bit memorizzati nel relativo descrittore di pagina virtuale e sostituirli con i primi 20 bit dell'indirizzo virtuale contenuti nei descrittori in cache (D1, D2, D3, D4). Così facendo, l'MMU non ha acceduto alla memoria fisica risparmiando molti cicli.

Nei campi D oltre ai 20 bit più significativi dell'indirizzo fisico, troviamo anche i bit PCD (*Page Cache Disable*) e PWT (*Page Write Through*).

Le complicazioni si verificano nel momento in cui si considerano i bit nel byte di accesso. Per alcuni bit è semplice perché basta controllarli, come il bit U/S per esempio. Per i bit A e D la cosa non è semplice. Il bit A non è nemmeno presente nel TLB poiché rappresenterebbe un'informazione inutile in quanto se il descrittore è nel TLB, sicuramente la pagina sarà stata acceduta. Inoltre se fosse presente, quando la routine delle statistiche, dopo aver terminato il suo lavoro, resetta tutti i bit A originali, quelli presenti nel TLB rimarrebbero settati e, poiché il TLB è nascosto al software, occorrerebbe *svuotarlo* di tutto il suo contenuto. Questo in ogni caso viene fatto perché quando molti descrittori di pagina virtuale vengono coinvolti da modifiche, si preferisce svuotare tutto il TLB anziché lasciare informazioni promiscue. Per svuotare il TLB basta scrivere qualcosa in `CR3`; la MMU assumerà che sta cambiando la traduzione degli indirizzi virtuali (possiamo riscriverci anche lo stesso contenuto di prima).
Per il bit D:

- se l'accesso è in lettura è insignificante, il bit non viene toccato
- se l'accesso è in scrittura e il bit D è a 1 non viene toccato, altrimenti viene considerato fallimento anche se il descrittore è presente e quindi la MMU farà l'accesso consueto al direttorio in memoria fisica; dal direttorio accede al descrittore della pagina virtuale voluto e qui modifica il bit D mettendolo a 1 e trasferisce il descrittore modificato nella stessa posizione in cui si trovava la sua versione precedente nel TLB

Non esistono istruzioni per leggere i bit D all'interno del TLB. La routine di page-fault deve essere consapevole dell'esistenza del TLB quando azzera i bit A (`movl %cr3, %cr3`) e anche quando viene tolta una pagina dalla memoria fisica in quanto si sta rendendo non più valida un'informazione nel TLB se vi era il descrittore di quella pagina. Se non ce ne preoccupassimo si accederebbe alla nuova pagina anziché a quella a cui volevamo veramente accedere. Quindi la routine di page-fault, quando si necessita di un rimpiazzamento, deve chiedere al TLB di invalidare il descrittore della pagina che intende rimuovere qualora vi fosse memorizzata: a questo scopo il processore è dotato di un'apposita istruzione `INVLPG` (INVaLidate PaGe)[4]. Da notare che, poiché cambiando il valore di `CR3` si ha cambiamento di contesto e svuotamento di tutto il contenuto del TLB, se la routine di page-fault sceglie come pagina vittima una pagina virtuale appartenente a un processo che non è più in esecuzione, è inutile invalidare il TLB per quella pagina.

[4]questa istruzione fa parte del set di istruzioni del processore PC. L'unico operando, sorgente, deve essere di memoria; esso è l'indirizzo fisico della pagina di cui vogliamo invalidare il descrittore nel TLB. Questa istruzione è privilegiata e per poterla eseguire occorre che il processore si trovi a lavorare in modo sistema.

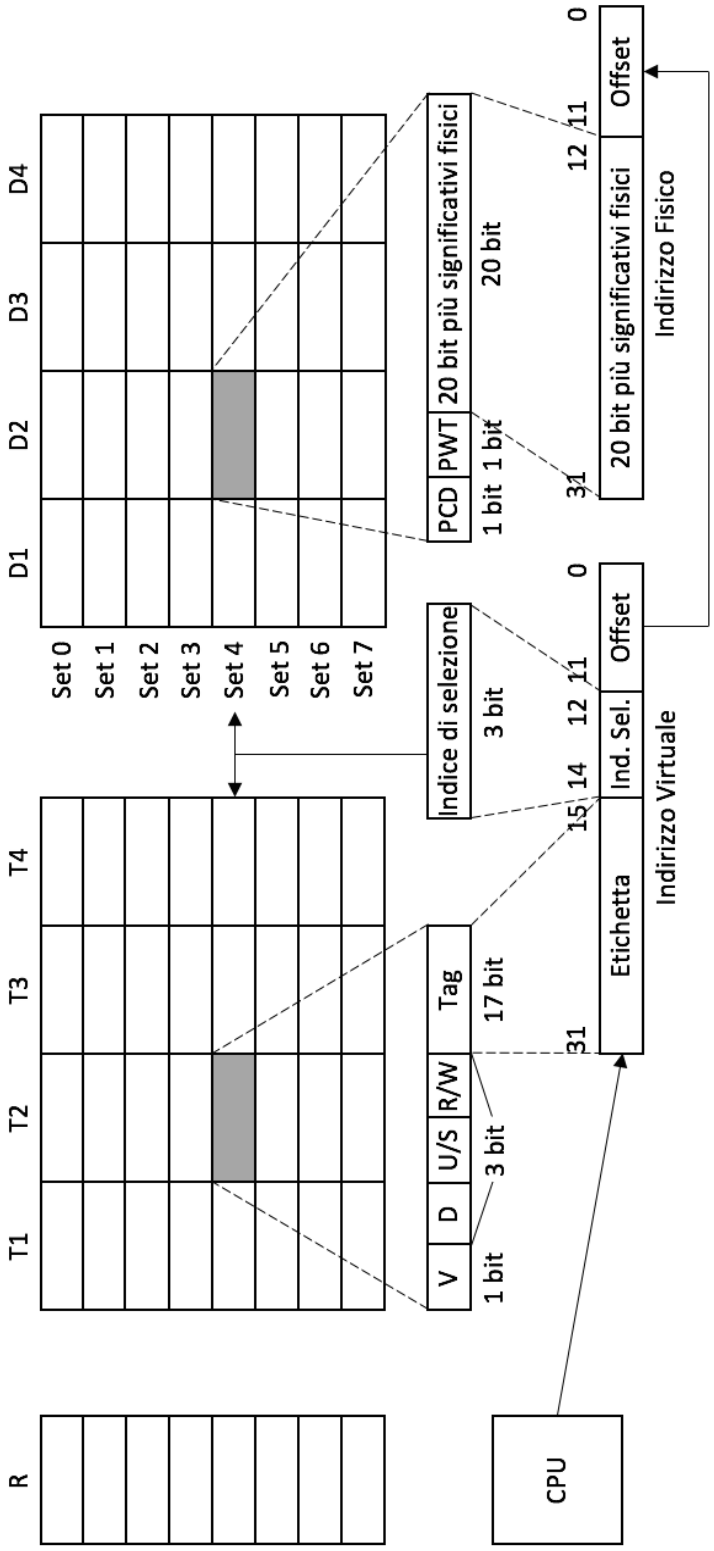


Figura 2.20: Struttura del TLB

2.14.1 Schema riassuntivo delle operazioni svolte dal meccanismo per individuare l'entità cercata dalla CPU

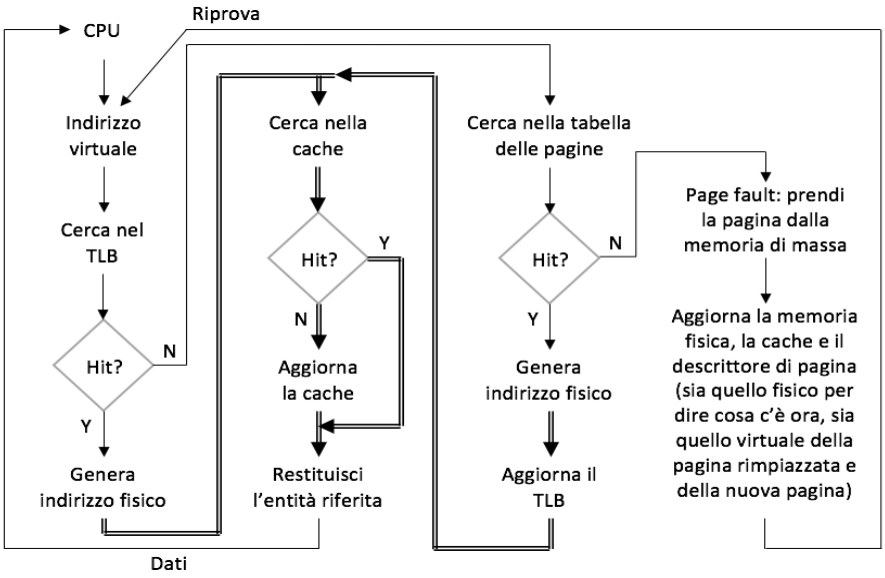


Figura 2.21: Schema delle operazioni

2.15 Gestione della memoria virtuale in un sistema multiprogrammato[5]

Rispetto a come abbiamo visto fino ad ora la memoria virtuale, il fatto che il sistema sia multiprogrammato richiede che ogni processo abbia la sua memoria virtuale. Si continua ad utilizzare la memoria virtuale, non tanto perché ci permette di avere l'impressione di lavorare su una memoria più grande di quella che abbiamo, ma perché il fatto di avere una memoria virtuale per ogni processo ci consente di tenerli separati in modo da evitare che gli errori possano propagarsi da un programma all'altro. In ogni caso occorre trovare dei compromessi a questo perché da una parte vogliamo che i processi siano isolati ma vogliamo anche che ogni tanto comunichino tra loro.

La memoria virtuale di ciascun processo viene suddivisa in parti:

- Sistema:
  - *parte_sistema_condivisa*: contiene il nucleo del sistema operativo (dati globali di livello sistema, routine di nucleo, processo dummy);
  - *parte_sistema_privata*: contiene la pila sistema del processo;
  - *parte_io_condivisa*: contiene i descrittori di I/O, le primitive di I/O e i processi esterni. Non c'è la parte privata perché la parte di I/O, trovandosi a livello sistema, ed essendo solo due i livelli di privilegio, usa la pila sistema.

[5]Questo paragrafo richiede la conoscenza dei concetti di interruzione, multiprogrammazione, processo, bus PCI.

- Utente:
  - *parte_utente_condivisa*: contiene i dati globali di livello utente e il codice dei processi utente;
  - *parte_utente_privata*: contiene la pila utente del processo.

La parte sistema, per un motivo o per un altro avrà bisogno di accedere a tutta la memoria fisica. Se deve poter accedere a tutta la memoria fisica questo vuol dire, automaticamente, che avrà accesso alla sua pila e ai sui dati (che stanno in memoria fisica). **Tutta la parte sistema non può essere letta e scritta dal processo utente**.

Se prendiamo la memoria fisica (la finestra FM), noteremo che le parti condivise corrispondono alle stesse entità mentre le parti private saranno diverse e, in particolare:

- la *parte_sistema_privata* ha un indirizzo virtuale diverso da processo a processo (una pagina occupata dalla pila sistema di un processo non può anche essere occupata dalla pila sistema di un altro processo);
- la *parte_io_condivisa* ha un indirizzo uguale per tutti i processi;
- la *parte_utente_condivisa* ha un indirizzo virtuale uguale per tutti i processi.

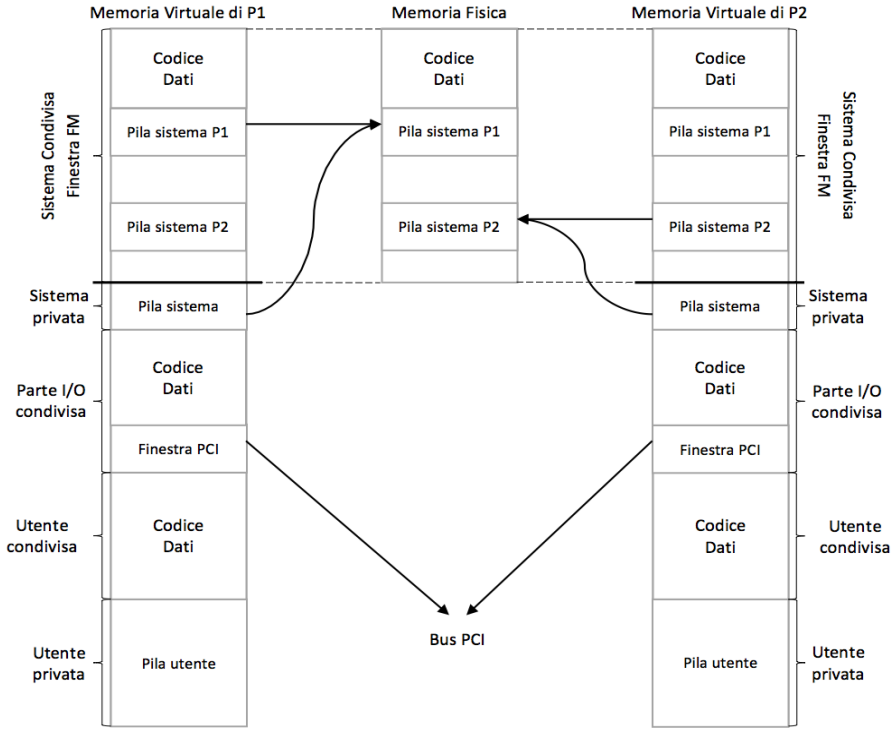


Figura 2.22: Suddivisione dello spazio virtuale e fisico

La finestra FM consente di vedere tutto ciò che è in memoria fisica.

La pila sistema varia da processo a processo: gli indirizzi virtuali che riguardano la pila sistema, nella parte di memoria virtuale riservata al processo, potranno essere gli stessi ma significheranno cose diverse da processo a processo.

Gli indirizzi virtuali della finestra PCI si trasformano in indirizzi che vanno sul bus PCI.

La parte sistema non può essere acceduta a livello utente, vi si può accedere solo in particolati situazioni causate dalle interruzioni (eccezioni, interruzioni hardware o software). Allora perché l'abbiamo messa nella memoria virtuale di ogni processo? Non avremmo potuto per questa utilizzare uno spazio di indirizzamento totalmente differente? Si potrebbe pensare all'occorrenza di accedere a questo spazio di indirizzamento cambiando il contenuto di `CR3`. Questa cosa si può fare ma ha uno svantaggio: ad ogni cambio di `CR3` viene invalidato tutto il TLB rallentando di molto il sistema. Questa cosa va evitata. In ogni caso, se lo volessimo fare, non potremmo dare all'utente tutti i 4GB perché almeno una pagina serve per memorizzarci delle entità che hanno a che fare con il meccanismo delle interruzioni. Quando viene generata un'interruzione, la CPU tramite l'indirizzo virtuale che si trova nel registro `IDTR` deve poter trovare la IDT per sapere a quale routine saltare. A questo punto la CPU deve capire se è necessario un innalzamento di livello di privilegio. Se è necessario, deve sapere dove si trova la pila di sistema e lo sa dal descrittore di processo. Dove si trova il descrittore di processo lo sa dalla GDT. Queste strutture devono stare tutte nella memoria virtuale di ciascun processo perché il microprogramma di interruzione le usa prima che si possa cambiare il valore di `CR3`.

Adottiamo, però, la prima delle soluzioni viste.

Delle entità in memoria virtuale alcune sono note fin da subito, altre si scoprono man mano quando si crea il processo.

Quali sono le cose che sono note appena il sistema parte?

- finestra FM
- finestra PCI[6]
- codice e dati utente
- codice e dati di I/O

Le entità che non sono note sono la pila sistema e la pila utente che vengono allocate quando viene creato il processo.

Cosa possiamo decidere di lasciare in memoria di massa e cosa invece occorre mettere in memoria fisica? La pila sistema di ogni processo deve essere residente perché serve al meccanismo delle interruzioni. Codice e dati del modulo I/O, codice e dati del modulo utente, poiché sono condivisi, devono essere residenti; se queste pagine fossero soggette a page-fault potrebbero crearsi copie della stessa pagina in memoria fisica venendo meno alla condivisione in quanto un processo potrebbe modificare dei dati contenuti in una pagina che dovrebbe essere condivisa mentre l'altro processo continuerebbe ad accedere ai vecchi dati e non a quelli aggiornati. Quindi ad ogni page-fault occorrerebbe aggiornare tutti i mapping e bisognerebbe fare in modo che i processi sappiano dove sono le *pagine condivise* che, in quanto tali, **devono essere puntate dagli stessi indirizzi virtuali per tutti i processi**. Lo stesso problema si verifica col rimpiazzamento: se un processo necessita di caricare in memoria fisica una pagina e ne rimpiazza un'altra di un altro processo, per far sì che non si verifichino problemi occorrerebbe effettuare, ogni volta che bisogna

[6] questa finestra serve per le periferiche PCI perché tutti i loro registri sono tipicamente mappati in memoria; se vogliamo che i programmi possano accedere a quei registri, questi dovranno essere mappati da qualche parte.

fare un rimpiazzamento, un page-fault generale in modo da rifare tutti i mapping. Quest'operazione gioca a sfavore dei programmi risultando non conveniente. ***Si effettua rimpiazzamento solo sulla pila utente***.

2.15.1 Descrittore di pagina fisica: analisi in dettaglio della struttura

Questa struttura dati ci serve per sapere, di ogni pagina fisica, tutte le informazioni di cui necessitiamo (se è occupata, da cosa è occupata, e tutte le informazioni per poter eventualmente rimuovere il contenuto attuale di quella pagina). La parte M1 è inutile immaginarla divisa in pagine perché nessuno la può toccare. La parte M2 della memoria fisica la immaginiamo invece divisa in pagine e per ognuna di esse riportiamo le seguenti informazioni:

```
1  enum tt{LIBERA, DIRETTORIO, TABELLA_CONDIVISA, TABELLA_PRIVATA,
       PAGINA_CONDIVISA, PAGINA_PRIVATA};
2
3  struct des_pf{
4          tt contenuto; //qual e' il contenuto della pagina
5          union{
6                  struct{
7                          bool residente; //pagina residente o meno
8                          natl processo; //identificatore di processo
                               non significativo per le pagine condivise
9                          natl ind_massa; //indirizzo della pagina in
                               memoria di massa
10                         addr virtuale; //indirizzo virtuale di una
                               riga della pagina (se tt = PAGINA e sono
                               significativi i primi 20 bit) oppure
                               indirizzo virtuale di una pagina (se tt =
                               TABELLA e sono significativi i primi 10
                               bit).
11                         natl contatore; //contatore per le statistiche
12                 } pt;
13                 struct{ //questo campo ci interessa solo se tt =
                       LIBERA
14                         des_pf* prossima_libera; //puntatore al
                               descrittore della prossima pagina libera
15                 } avl;
16         };
17 };
18 //i seguenti tre oggetti fanno parte della sezione .data del modulo
       sistema in M1
19
20 des_pf dpf[N_DPF]; //array di descrittori di pagina fisica
21 addr prima_pf_utile; //indirizzo fisico della prima pagina fisica di
       M2
22 des_pf* pagine_libere; //puntatore alla lista di descrittori delle
       pagine fisiche libere
```

Se `tt` vale `LIBERA`, quello che ci interessa è mantenere una lista di tutte le pagine fisiche libere. Per le pagine che sono di tutti i processi, il campo `processo` sarà inizializzato con un valore rappresentativo, per esempio con l'identificatore del processo dummy. In particolare ci interessa il campo `processo` solo per quelle pagine che appartengono al singolo processo, sono cioè private, ovvero le pile.

Quando c'è una commutazione di contesto, le pagine relative al processo uscente non vengono rimpiazzate da quelle del processo entrante; il meccanismo è sempre lo stesso: rimpiazzare solo se non c'è più spazio. Ma anche nel caso in cui non ci sia

più spazio, non è detto che le pagine del processo attualmente in esecuzione abbiano statistiche migliori di quelle ancora presenti in memoria fisica ma appartenenti al processo bloccato. Infatti è possibile che il processo attualmente in esecuzione carichi delle pagine e si concentri solo su una utilizzando così pochissimo le altre. Quindi quando viene generata un'eccezione di page-fault viene rimpiazzata la pagina con statistiche peggiori cercata fra quelle di tutti i processi e non solo fra quelle dei processi bloccati e ancora presenti in memoria.

Quando viene scelta una pagina come vittima occorre comunque sapere a quale processo appartiene per modificare nella tabella delle pagine il bit P relativo a quella pagina. Per sapere dove è la tabella delle pagine occorre sapere dove si trova il direttorio relativo al processo cui la pagina appartiene. Sappiamo dove si trova il direttorio in questo modo:
ID processo $\longrightarrow$ GDT $\longrightarrow$ descrittore di processo $\longrightarrow$ `CR3` $\longrightarrow$ indirizzo fisico del direttorio.

Il campo *addr virtuale* serve conoscerlo per rompere il mapping al momento del rimpiazzamento.

Se `tt` vale `TABELLA`, per identificare la tabella stessa, basta che *virtuale* sia uno dei 1024 indirizzi delle pagine a cui punta.

Una volta che abbiamo questa struttura dati, ne facciamo un array per dimensione uguale al numero delle pagine fisiche.

Per gestire tutte le strutture dati della paginazione occorre predisporre alcune funzioni di utilità.

Le funzioni `loadCR3(addr dir)` e `readCR3()` per caricare e leggere il registro speciale `CR3` vengono dichiarate `extern "C"` e scritte in Assembly.

2.15.2 Routine di *page-fault*

La `c_routine_pf` che va in esecuzione chiamata dalla `int_tipo_pf`, va in esecuzione ogni volta che abbiamo un page-fault. Essendo una routine di nucleo, anche questa girerà a interruzioni disabilitate; questo significa che gli accessi all'HD saranno fatti a controllo di programma.

```
1  #sistema.s
2  .TEXT
3  .GLOBAL readCR2
4  readCR2:        MOVL    %cr2, %eax
5                  RET
6
7  .EXTERN c_routine_pf
8  #routine di page-fault, coinvolto un gate di tipo Interrupt.
9  int_tipo_pf:    #salvataggio dei registri
10                 CALL    c_routine_pf
11                 #ripristino dei registri
12                 IRET
13
14 //sistema.cpp
15 extern ''C'' addr readCR2();
16
17 extern ''C'' void c_routine_pf(){
18         addr ind_virt = readCR2();
19         bool P;
20         natl dt;
21         dt = get_des(esecuzione->id, TABELLA_PRIVATA, ind_virt); //
               tabella privata perche' si puo' avere page-fault solo su
               pila utente
```

```
22          P = extr_P(dt);
23          if(!P) swap(TABELLA_PRIVATA, ind_virt);
24          swap(PAGINA_PRIVATA, ind_virt);
25 }
```

Con la funzione `readCR2()`, leggiamo qual è l'indirizzo virtuale che ha causato page-fault. Tramite la funzione `get_des()` preleviamo il descrittore della tabella. Utilizziamo la funzione di utilità `extr_P()` per estrarre il bit P relativo al descrittore passato come parametro.

La funzione `swap(...)` può fare indifferentemente lo *swap* di una tabella o di una pagina (basta passare il tipo come parametro) e nel rimpiazzare può indifferentemente rimuovere una tabella o una pagina.

```
1  enum tt{...};
2
3  void swap(tt tipo, addr ind_virt){
4          des_pf* nuovo_dpf = alloca_pagina_fisica_libera();
5          if(nuovo_dpf == 0){ //se nessuna pagina libera; operazioni di
               rimpiazzamento
6                  des_pf* dpf_vittima = scegli_vittima(tipo, ind_virt);
7                  bool occorre_salvare = scollega(dpf_vittima); //
                       restituisce un booleano che ci dice se occorre
                       salvare la pagina vittima in memoria di massa
8                  if(occorre_salvare) scarica(dpf_vittima); //se il bit
                       D e' settato, la pagina da rimuovere va
                       ricollocata in memoria di massa all'indirizzo IM
9                  nuovo_dpf = dpf_vittima;
10         }
11         natl des = get_des(esecuzione->id, tipo, ind_virt);
12         natl IM = extr_IND_MASSA(des);
13         if(!IM){ //indirizzo fuori dallo spazio virtuale allocato
14                 rilascia_pagina_fisica(nuovo_dpf);
15                 abort_p();
16         }
17         //inizializzazione nuovo_dpf
18         nuovo_dpf->contenuto = tipo;
19         nuovo_dpf->pt.residente = false;
20         nuovo_dpf->pt.processo = esecuzione->id;
21         nuovo_dpf->pt.virtuale = ind_virt;
22         nuovo_dpf->pt.ind_massa = IM;
23         nuovo_dpf->pt.contatore = 0; //giusto 0? si' perche' l'accesso
               che c'e' stato ha causato page-fault e quindi va ripetuto
24         carica(nuovo_dpf);
25         collega(nuovo_dpf);
26 }
```

Quando la routine di page-fault seleziona, per il rimpiazzamento, una tabella delle pagine vuol dire che in memoria fisica non ci sono più pagine virtuali riferite dalla tabella stessa: nei descrittori di pagina virtuale al suo interno tutti bit P saranno a 0 e compariranno esclusivamente indirizzi delle pagine in memoria di massa. Poiché le tabelle delle pagine quando non riferiscono nessuna pagina virtuale è come se fossero tornate al loro stato originale, non ha senso ricopiarle in memoria di massa.

2.16 I registri CR (Control Registers) della CPU

In questo capitolo abbiamo fatto riferimento ad alcuni *registri speciali* della CPU, i **Control Registers**, che assumono un ruolo fondamentale nel meccanismo della

memoria virtuale.
Riassumendo, abbiamo quattro registri CR:

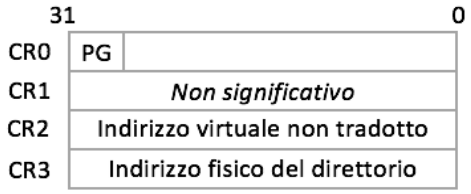


Figura 2.23: Registri speciali della CPU

- `CR0`: serve ad abilitare la paginazione, ponendo ad 1 il bit PG (PaGing, bit n.31);
- `CR1`: non ha alcuna utilità ma è mantenuto per ragioni di retrocompatibilità;
- `CR2`: serve a contenere l'indirizzo **virtuale** non tradotto quando viene generata una eccezione di page-fault;
- `CR3`: serve a contenere l'indirizzo **fisico** del direttorio. Un suo aggiornamento, oltre a cambiare lo spazio d'indirizzamento (se diverso dal valore precedente) provoca l'invalidazione del contenuto del TLB (anche se il valore è uguale al precedente). Notare che `CR3` serve alla MMU ogni volta che deve tradurre un indirizzo da virtuale a fisico.

Capitolo 3

Meccanismo delle interruzioni

3.1 Utilità del meccanismo

A cosa servono le interruzioni? Servono per evitare i cosiddetti *cicli attivi*. Durante questi cicli di attesa che si hanno per esempio quando dobbiamo aspettare che un bit abbia un certo valore oppure quando aspettiamo che una periferica sia pronta, il processore non può compiere alcuna altra operazione. L'idea è quella che mentre il processore è in attesa che la periferica sia pronta, un altro programma possa eseguire le sue operazioni. Poi, però, occorre un modo per sapere quando quello che aspettiamo è pronto.

Quindi il concetto è poter utilizzare il sistema per compiere delle operazioni mentre le periferiche stanno elaborando i dati che non hanno bisogno del processore, per poi tornare, in un qualche modo, al programma che aveva bisogno di quei dati.

Il meccanismo delle interruzioni è stato quindi introdotto per per poter utilizzare più efficacemente il processore.

N.B.: il trasferimento dei byte, per esempio dall'Hard-Disk, avviene più velocemente col metodo del ciclo attivo; infatti **le interruzioni non servono a velocizzare, servono solo a migliorare l'efficienza di utilizzo**.

Questo meccanismo, però, complica un po' le cose in quanto adesso avvengono più cose contemporaneamente e queste cose potrebbero avvenire nei momenti sbagliati.

3.2 Funzionamento del meccanismo delle interruzioni esterne (o hardware)

Se più periferiche hanno bisogno di richiedere informazioni, le loro richieste passano prima dal controllore delle interruzioni (APIC) che in qualche modo le ordina e le passa ***una alla volta*** al processore. APIC sta per *Advanced Programmable Interrupt Controller*.

Il processore esegue ciclicamente, senza mai fermarsi (a meno di apposite istruzioni) le seguenti azioni:

1. ***fetch***
2. ***decode***
3. ***execute***

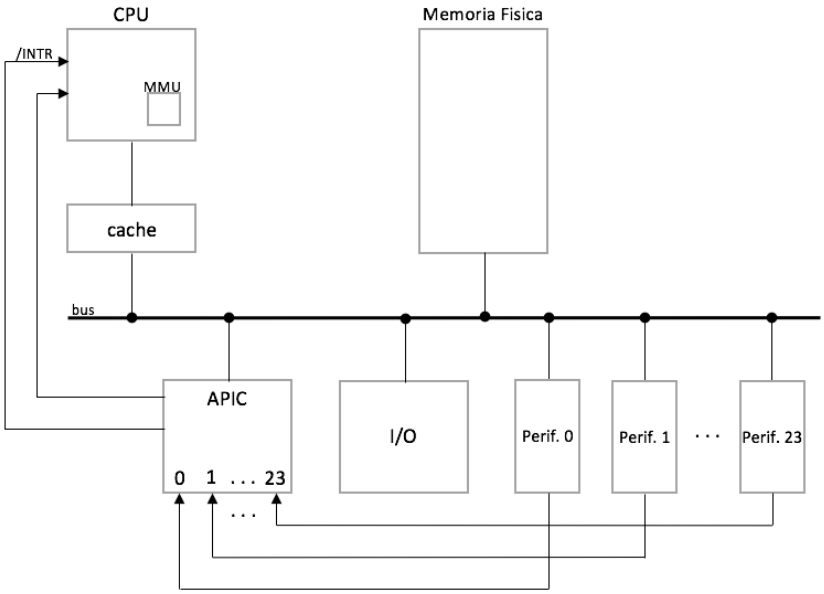


Figura 3.1: Collegamento al bus dell'APIC e delle varie periferiche

il meccanismo delle interruzioni interviene immediatamente dopo la fase *execute*. Al termine di ogni fase di esecuzione, il processore controlla se c'è una richiesta di interruzione pendente (*pending request*), cioè se lo stato logico del piedino `/intr` è uguale a 0. Se lo trova a 0, salta ad un'altra istruzione; invece di prelevare la prossima che si trova in `EIP`, ne preleva un'altra previo salvataggio del contenuto attuale di `EIP`in pila. Una volta saltato all'altra istruzione, il processore riprende ad eseguire ciclicamente le sue funzioni come se niente fosse. Per il processore, la gestione delle interruzioni finisce qui. Questo è quello che accade ad alto livello. Più in dettaglio:
il processore, nel registro dei flags `EFLAG` ha un bit IF (Interrupt Flag) che, a seconda del suo valore, dice al processore se deve stare attento al valore logico del piedino `/intr` oppure no. Se il flag è resettato, il processore non guarda se ci sono delle interruzioni; diversamente, alla fine della fase di esecuzione di ogni istruzione viene testato il piedino `/intr`.

Inoltre, automaticamente, quando il processore si accorge che c'è un'interruzione, resetta il flag IF previo salvataggio in pila dell'attuale stato del registro `EFLAG`.

Come fa a capire il processore a quale istruzione deve saltare? Più periferiche significa che bisogna sapere soddisfare richieste diverse. Per fare questo, l'APIC assegna un numero diverso ad ogni interruzione a seconda del piedino attraverso cui riceve il segnale. Questo numero viene poi comunicato al processore tramite una linea seriale o un insieme di fili dal controllore APIC. Una volta che il processore ha ricevuto il numero, consulta la tabella IDT (*Interrupt Descriptor Table*) e all'occorrenza della tabella il cui indice è dato dal quel numero, c'è scritta, oltre ad altre informazioni, l'istruzione a cui saltare. **Questo meccanismo ci permette di associare routine diverse a interruzioni diverse**.

Dal punto di vista hardware la cosa è più complicata in quanto il meccanismo, una volta aperto, va richiuso. Questo perché, se da un lato al processore non interessa nulla di questo meccanismo, perché lui esegue solo istruzioni, dall'altro il software deve sapere quando le richieste sono state soddisfatte. Inoltre tutti gli at-

tori che cooperano per il funzionamento di questo meccanismo devono essere ben sincronizzati.

Prendiamo l'esempio della tastiera in cui abbiamo un ciclo attivo nel quale attendiamo che un tasto venga premuto. Noi vogliamo che durante questo ciclo possa essere eseguita qualche altra operazione per qualche altro programma. Per fare questo occorre impedire al programma utente di "parlare" direttamente con la tastiera o con le periferiche in generale. Questo possiamo farlo dicendo che quando il programma in esecuzione è un programma utente, le istruzioni `IN` e `OUT` sono vietate. Dobbiamo però consentire al programma utente di accedere alla periferica anche se non in maniera diretta. Gli dobbiamo quindi fornire delle *primitive* tramite le quali il programma può chiedere al sistema di eseguire per lui le operazioni che gli abbiamo vietato di fare. Sia la periferica che l'APIC devono sapere se la richiesta è stata realmente soddisfatta dal processore. La tastiera, per esempio, lo sa nel momento in cui viene letto il buffer. L'APIC deve sapere che la richiesta è stata soddisfatta perché gestisce tante periferiche e tante interruzioni ma ne può soddisfare solo una per volta: quando sa che la precedente è stata soddisfatta, se ve ne sono altre pendenti, ne manda un'altra al processore. All'APIC lo comunichiamo attraverso un flag che è gestito dal processore e che si trova all'interno dell'APIC stesso. Il processore questo flag potrebbe gestirlo attraverso un'istruzione; in ogni caso, invece di aggiungere un'istruzione in più, essendo l'APIC stesso una periferica, basta che abbia un registro in cui ci si possa scrivere un qualcosa che gli indichi che l'interruzione è stata gestita. Questo registro si chiama `EOI` (*End Of Interrupt*). L'APIC è una di quelle periferiche che ha i suoi registri in memoria fisica. L'istruzione di scrittura nel registro `EOI` deve attraversare la cache così com'è e per farlo occorre settare il bit PCD (*Page Cache Disable*) del descrittore di pagina virtuale in cui si trova il registro.

3.3 Servizio effettuato dalle routine di interruzione

Le *routine di interruzione* sono fornite con il software di sistema. Vengono caricate in memoria nella fase di inizializzazione (*bootstrap*). Il bootstrap provvede a inizializzare i gate della IDT inserendo, fra le altre cose, l'indirizzo della prima istruzione della routine a cui saltare.

Le interruzioni esterne mascherabili sono generalmente prodotte dalle interfacce di ingresso/uscita. Queste interruzioni, se abilitate, arrivano direttamente ai piedini dell'APIC. Le routine di interruzione relative alle interfacce di I/O prendono il nome di ***driver***.

L'istruzione `INT` serve ad invocare routine di interruzione con un meccanismo alternativo ad una chiamata effettuata con una semplice `CALL`. La differenza risiede, prima di tutto, nel fatto che la `INT` consente un innalzamento di livello di privilegio e questo consente all'utente di richiedere al sistema di fare per lui operazioni che non può o non ha il permesso di effettuare per conto proprio. Inoltre la `INT` consente di invocare la routine attraverso il numero d'ordine e non tramite nome per cui non è necessario effettuare nessun tipo di collegamento fra queste routine e il programma chiamante. Le routine di interruzione relative a istruzioni `INT` prendono il nome di ***primitive***.

Le routine di interruzione che gestiscono le eccezioni provvedono, sostituendosi al programma in esecuzione, ad impedire che il programma stesso continui ad operare nel modo scorretto (tipico esempio è la routine di page-fault).

3.3.1 Struttura di una primitiva in Assembly e C++

In un programma principale scritto in linguaggio di alto livello come il C++, non è possibile chiamare direttamente le primitive in quanto non è disponibile un costrutto analogo all'istruzione `INT`. Inoltre le primitive devono terminare con una `IRET` e anche questa istruzione non trova analoghi costrutti in C++. Occorre dunque utilizzare il linguaggio Assembly almeno per una parte dell'implementazione. La parte elaborativa di una primitiva può essere invece facilmente scritta in C++.

Detto questo, la struttura di una primitiva è la seguente:

```
1 //main.cpp: contiene il programma principale
2 extern ''C'' void primitiva_i(/*argomenti formali*/);
3 //...
4 int main(){
5         //...
6         primitiva_i(/*argomenti attuali*/);
7         //rit...
8 }
9
10
11 #main.s: contiene il sottoprogramma di interfaccia
12 .TEXT
13
14 .GLOBAL primitiva_i
15
16 primitiva_i:
17         INT     $tipo_i
18 irit:   RET
19
20 #primitiva.s: contiene la primitiva vera e propria (a_primitiva)
21 .TEXT
22
23 .EXTERN c_primitiva_i
24
25 a_primitiva_i: #routine di tipo_i
26         CALL    c_primitiva_i
27 prit:   IRET
28
29 //primitiva.cpp: contiene la parte elaborativa della primitiva(
       c_primitiva)
30 //...
31 extern ''C'' void c_primitiva_i(/*argomenti fittizi + argomenti
       attuali*/){...}
```

In teoria una funzione, dopo aver salvato in pila `EBP` e il valore di ritorno `EIP`, inizierebbe a riferire gli argomenti attuali da `EBP+8`. Per la `c_primitiva_i`, però, non è così. Non vi siamo arrivati chiamandola direttamente nel main (semplice `CALL`) ma ci siamo arrivati attraverso una serie di chiamate a cascata che hanno messo in pila delle parole lunghe "impreviste". Quindi, per far sì che la `c_primitiva_i` operi correttamente sugli argomenti attuali passati alla `primitiva_i`, occorre scavalcare gli oggetti in più. Questo lo facciamo aggiungendo 4 argomenti fittizi da 4 byte ciascuno (`4 natl`) agli argomenti attuali sui quali poi dovrà effettivamente lavorare la funzione. La `c_primitiva_i` inizierà a riferire gli argomenti da `EBP+8+16` (Figura 3.2).

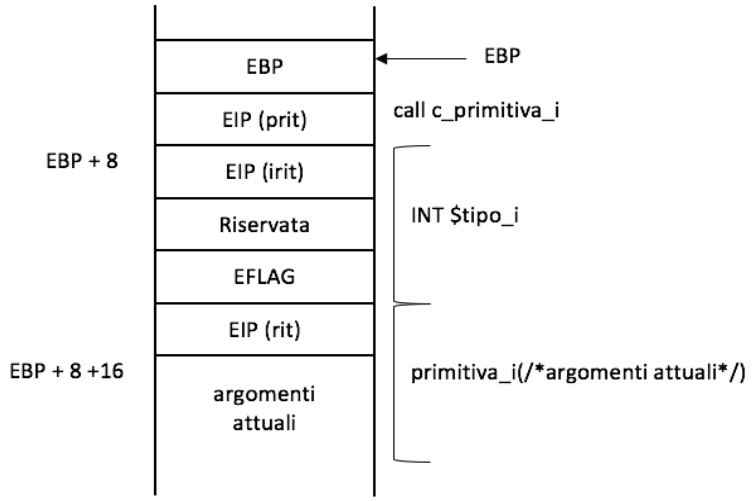


Figura 3.2: Posizione degli argomenti in pila

3.3.2 Struttura di un driver di interruzione

Un driver di interruzione è **asincrono** rispetto al programma principale. Per questa ragione non può avere dei parametri. Può avere accesso alle informazioni che gli servono per completare le operazioni richieste, tramite alcune variabili globali appositamente inizializzate dal programma principale stesso. Anche in questo caso occorre dividere l'implementazione: la parte in Assembly contenente la chiamata alla parte elaborativa in C++ `c_driver_i` e l'istruzione di ritorno al programma principale `IRET`, e l'implementazione della parte elaborativa stessa scritta in C++.

Pertanto, la struttura di un driver di interruzione è la seguente:

```
1 #driver.s
2 .TEXT
3 .EXTERN c_driver_i
4 a_driver_i:     #routine di tipo_i
5         PUSHAL  #viene coinvolto un gate di tipo interrupt
6         CALL    c_driver_i
7         MOVL    $0, 0xFEE000B0  #invio della End Of Interrupt
8         POPAL
9         IRET
10
11 //diver.cpp
12
13 extern ''C'' void c_driver_i(){...}
```

3.3.3 Gestione dell'interfaccia della tastiera 8042[1]

Bisogna sapere a priori a quale piedino dell'APIC è attaccata la periferica che ci interessa. Nella funzione `ini()`:

```
1 void ini(){
2         gate_init(241, a_leggi_linea); //associa all'entrata 241 della
              IDT la primitiva a_leggi_linea
3         gate_init(41, a_driv_tasint); //associa all'entrata 41 della
              IDT la primitiva a_driv_tasint
```

[1] questo paragrafo fa riferimenti al programma es07.1 contenuto negli esempi sull'I/O disponibili sul sito del corso al link http://calcolatori.iet.unipi.it/resources/esempiIO.tar.gz.

```
4         apic_set_VECT(1,41); //piedino 1 associato al tipo 41
5         apic_set_MIRQ(1, false); //abilitazione del piedino 1
6 }
```

La `a_leggi_linea` e la `a_driv_tasint` sono dichiarate e definite in Assembly. La `a_driv_tasint` va in esecuzione ogni volta che arriva una richiesta di interruzione dalla tastiera. Questa funzione è scritta in Assembly perché l'istruzione `IRET` non ha nessun costrutto analogo in C++ e una richiesta di interruzione va conclusa con la `IRET`. La `leggi_linea` è una di quelle funzioni che chiamiamo primitive. Se fossimo nel sistema con due livelli di privilegio, vi si potrebbe saltare direttamente dal `main()` ma dovremmo anche cambiare livello di privilegio e questa cosa non si fa con una semplice `CALL`; l'unica istruzione che ci permette di cambiare il livello di privilegio è la `INT`. La `INT` salva, oltre ad `EIP`, anche il registro dei flag. Anche `leggi_linea` è scritta in Assembly perché la `INT`, come la `IRET`, non trova costrutti analoghi in C++.

```
1 leggi_linea:
2         INT $241
3         RET
```

Il numero `$241` è quello che indica l'entrata del gate della tabella IDT in cui abbiamo predisposto la routine per l'interruzione e quando arriva una interrupt di tipo `241` saltiamo alla prima istruzione di questa routine. Tutte le funzioni "agganciate" alla IDT alla quale si arriva tramite una `INT`, alla fine devono avere una `IRET` e non una semplice `RET`.

La `c_leggi_linea` è programmata in modo che la tastiera mandi interruzioni quando sono pronti i tasti. Poiché in questa versione semplificata non abbiamo altri programmi a cui passare, utilizziamo una funzione di libreria che aspetta che un certo flag vada a 1. Questo flag verrà settato dalla routine di interruzione una volta letta la linea.

Avviando il programma vediamo il cursore lampeggiare. In questo momento il processore sta eseguendo istruzioni per controllare lo stato di un flag (`sem_wait()`); se invece avessimo avuto un altro programma da eseguire, avrebbe eseguito quello. Se viene premuto un tasto, la rappresentazione di quel tasto compare a video. La pressione di questo tasto ha causato una lunga serie di eventi:

1. pressione del tasto;
2. riconoscimento del tasto premuto da parte della tastiera;
3. invio del codice al microcontrollore che sta sulla scheda madre;
4. il microcontrollore invia un'interruzione;
5. l'interruzione arriva al controllore delle interruzioni;
6. non ci sono richieste di interruzione pendenti quindi il controllore delle interruzioni passa l'interruzione al processore;
7. il processore, che alla fine di ogni istruzione verifica se ci sono interruzioni, vede che ce n'è una e che le interruzioni sono abilitate (flag IF);
8. richiede al controllore delle interruzioni di inviargli il tipo dell'interruzione;
9. salva in pila l'attuale contenuto di `EIP`;

10. controlla la tabella IDT;

11. salta alla routine il cui indirizzo della prima istruzione si trova nel gate;

12. legge il tasto;

13. la routine fa l'echo del tasto letto, invia EOI e poi esegue una `IRET`;

14. la `IRET` ripristina il vecchio contenuto di `EIP` e il vecchio contenuto di `EFLAG`;

15. ora il processore sta di nuovo eseguendo le istruzioni per il controllo del flag (`sem_wait()`).

Tutto questo si ripete finché non viene premuto ENTER per il quale succede tutto quanto visto prima ma in più viene settato il flag il cui stato viene verificato dalla `sem_wait()`; quindi quando esegue la `IRET` per il ritorno al programma principale, `sem_wait()` vede che il flag è settato; questo provoca:

1. il termine della `sem_wait()`;

2. la stampa a video dei caratteri inseriti;

3. la terminazione del programma, che dopo aver chiamato la `pause()`, attende che venga premuto ESC.

3.3.4 Gestione dell'interfaccia di conteggio 8254[2]

Un'altra periferica che può essere usata con le interruzioni è il *timer*. È una periferica di conteggio nella quale è possibile inizializzare un opportuno registro con un certo valore, trascorso il quale, invia un segnale.

Nei PC compatibili IBM abbiamo tre periferiche di conteggio:

1. una collegata allo speaker;

2. una collegata al controllore delle interruzioni;

3. una collegata ad un altro oggetto non più utilizzato.

Il segnale che la periferica di conteggio genera alla fine, viene inviato direttamente al controllore delle interruzioni come interruzione. Quindi possiamo programmare la periferica in modo tale che invii un'interruzione dopo che è passato un certo periodo di tempo.

Anche in questo caso l'utente non ha accesso diretto alla periferica. Quindi per accedere al timer occorre l'utilizzo di una primitiva che abbia livello di privilegio superiore.

[2]questo paragrafo fa riferimenti al programma es07.2 e al programma es07.3 contenuti negli esempi sull'I/O disponibili sul sito del corso al link http://calcolatori.iet.unipi.it/resources/esempiIO.tar.gz.Tutti questi esempi sono eseguibili con emulatore QEMU i cui sorgenti si trovano http://calcolatori.iet.unipi.it/resources/qemu-ce.tar.gz

Esempio di utilizzo: es07.2

La `go_timer` è una primitiva. Si occupa di chiamare una routine che ad intervalli regolari di tempo, invia un'interruzione e stampa a video una 'D'. Anche in questo caso il programma avrà bisogno di alcune parti scritte in Assembly per poter utilizzare le istruzioni `INT` e `IRET` che non hanno costrutti analoghi in C++. Sia parte utente che parte sistema appena possono utilizzano il C++.

La funzione `ini()`, in cui vengono inizializzati i gate della IDT e l'APIC, viene chiamata dall' entry point `_start` che si trova nella libreria. Nell'entry point viene chiamata prima `ini()` e poi `main()`. In questo caso, nella funzione `ini()`, occorre richiamare un'altra funzione `ini_p_CONT0()` in cui viene inizializzato il contatore `0` dell'interfaccia di conteggio.

La primitiva

```
1 extern ''C'' void c_go_timer(natl a, natl b, natl c, natl d, natl num)
     {
2          //...
3 }
```

ha come argomenti `a`, `b`, `c`, `d` e `num`. I primi quattro sono fittizi e servono per scavalcare i parametri inseriti in pila dalla `INT` e dalla ulteriore `CALL` prima di arrivare a questa.

La periferica di conteggio, rispetto a quasi tutte le altre periferiche ha qualcosa di diverso nel modo in cui vengono gestite le interruzioni. Per la tastiera abbiamo visto che essa è di nuovo in grado di inviare un'interruzione non appena viene prelevato il codice dal buffer. Per il timer non è così. Infatti lui invierà l'interruzione ogni volta che si è esaurito quel tempo e non vi è alcuna risposta che aspetta dal software.

Il timer è una periferica per cui si possono perdere richieste di interruzione. Questo perché il controllore delle interruzioni non fa passare più di una richiesta alla volta e se vi sono interruzioni pendenti a più alta priorità e vi è già una richiesta pendente dal timer, se non si fa in tempo a smaltire tutte le richieste prima che il timer invii la prossima, quest'ultima si perde poiché il controllore è in grado di ricordarsi al più una richiesta pendente per ciascuna periferica.

Esempio di utilizzo: es07.3

Vogliamo che alla pressione di un tasto, venga emesso un suono che termina dopo una certa quantità di tempo (non al rilascio del tasto, quindi). Un modo **sbagliato** di fare questo è utilizzare un ciclo di attesa attiva (ad esempio un ciclo `for` da 0 a 1.000.000) terminato il quale disattiva la nota. Questo metodo è sbagliato perché la velocità con cui viene eseguito il `for` cambia da macchina a macchina (clock del processore, tecnologia del processore). Quindi il tempo di esecuzione di quel ciclo non sarebbe conoscibile a priori. Poi, altra motivazione, quando il compilatore C++ ottimizza, toglie i cicli `for` che non fanno nulla. Quindi è più opportuno utilizzare un'interfaccia di conteggio. È necessario dunque inizializzare due contatori: uno collegato al controllore delle interruzioni e un altro collegato allo speaker.

La funzione `delay()`, chiamata nella `c_nota(...)`, è una funzione di libreria.

Il corpo della `c_nota(...)`:

1. carica nel contatore 2 l'argomento `cc`;
2. abilita lo speaker scrivendo nel registro `SPR`;

3. richiama la primitiva `delay()` con argomento 5;

4. disabilita lo speaker scrivendo nel registro `SPR`.

Il suono verrà emesso per un tempo pari a 5*50 ms.

3.3.5 Gestione dell'interfaccia a blocchi ATA

Anche l'interfaccia a blocchi ATA può essere gestita a interruzione di programma. Per farlo, occorre abilitargli le interruzioni tramite il registro `DCR` (*Device Control Register*) nel cui bit 1 scriviamo 0 per abilitare o 1 per disabilitare le interruzioni. L'abilitazione deve essere fatta prima di specificare nel registro `CMD` il tipo dell'operazione.

Una lettura nel registro `STS` (*STatus Register*) costituisce anche una risposta ad una richiesta di interruzione.

Anche per l'HD vi è una differenza nel protocollo con cui vengono gestite le interruzioni rispetto alla tastiera. Diciamo che tastiera, Hard-Disk e timer coprono tutti i tre modi possibili di gestire le interruzioni.

La differenza per quanto riguarda l'HD sta nel fatto che può mandare interruzioni solo se precedentemente gli era stato ordinato di fare qualcosa da parte del programma. Per la tastiera è diverso perché è la persona che decide quando premere il tasto senza ricevere, ovviamente, nessun ordine da nessun programma (sorgente delle interruzioni esterna al programma).

Per scrittura e lettura compie le stesse operazioni che compiva a controllo di programma tranne che quando ha terminato, invia una richiesta di interruzione.

3.4 Controllore APIC (Advanced Programmable Interrupt Controller)

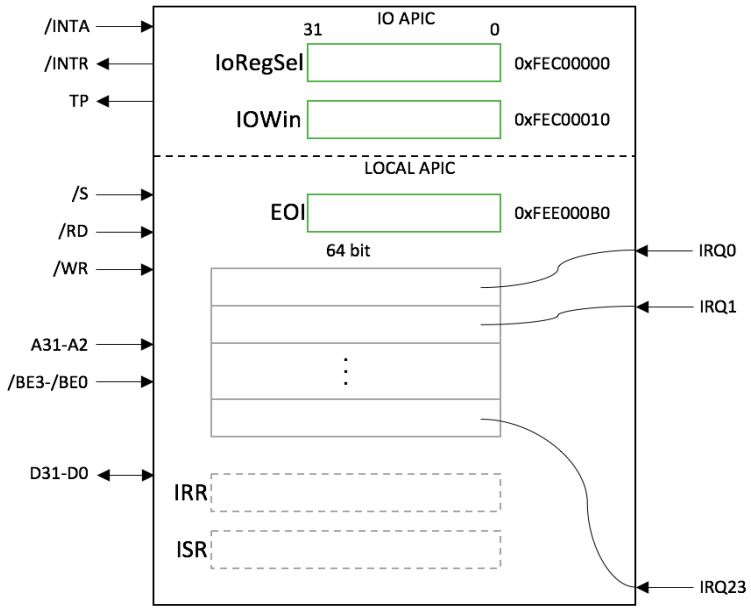


Figura 3.3: Controllore APIC e suoi piedini

L'APIC essendo esso stesso una periferica, possiede i classici piedini per il collegamento con il bus. In più ha anche i piedini per la comunicazione diretta con la CPU:

- **/INTR** segnala al processore che c'è una richiesta in attesa di essere soddisfatta;
- **/INTA** segnala al controllore APIC che la richiesta è stata accettata dal processore;
- **TP** serve per mandare serialmente al processore il tipo dell'interruzione

Ha inoltre 24 piedini da cui riceve le richieste di interruzione (`IRQ0...IRQ23`). Registri di interesse nell'APIC:

- 24 registri a 64 bit, uno per ogni piedino `IRQ`, per memorizzare alcune informazioni riguardo il segnale di interruzione che arriva su quel piedino. Una di queste informazioni è, per esempio il tipo che occupa 16 dei 64 bit. Un'altra informazione è un bit per indicare se l'interruzione proveniente da quel piedino è abilitata oppure no. Non è possibile accedere direttamente a questi registri a causa della mancanza di fili per l'indirizzo e quindi usiamo i seguenti due registri:
  - **IORegSel**: indice per l'accesso a uno dei 24 registri;
  - **IOWin**: permette di leggere/scrivere da/su uno di questi registri.
- **EOI**: scrivendo una particolare configurazione all'interno di questo registro, comunichiamo all'APIC il termine delle interruzioni da una certa periferica.

`IORegSel`, `IOWin` ed `EOI` si trovano in memoria fisica e si trovano ad indirizzi che iniziano con `0xFE`. Per scrivere all'interno di questi registri basta quindi una semplice `MOV`.

Il compito del controllore delle interruzioni consiste nel gestire il fatto che si possono avere più richieste di interruzione ma il processore può vederne solo una per volta. Questo comporta che il controllore debba in qualche modo sapere due cose:

1. di che interruzione si tratta (lo sa tramite il tipo);
2. quale di quelle pendenti deve far passare.

Per l'ultima di queste due informazioni si assume che tutte le richieste non abbiano la stessa importanza: ci sono alcune richieste che non devono aspettare troppo per essere gestite, altre che invece possono attendere. Queste richieste hanno delle *priorità*. Per esempio, le richieste che arrivano dal timer sono delicate perché si potrebbero perdere e quindi sono ad alta priorità; inoltre, dovendo inviare segnali al termine di un certo periodo di tempo, il segnale deve arrivare al termine di questo periodo e non più tardi. Altre invece sono a priorità bassa: la tastiera, per esempio, possiede un buffer interno che memorizza gli ultimi (circa 10) tasti premuti e li invia in ritardo quando è di nuovo possibile inviare richieste di interruzione; per l'Hard-Disk il peggio che può succedere è che il dato pronto "attende" nel buffer e il prossimo dato che deve essere prelevato passa sotto la testina finché il buffer non è libero per accogliere nuovi dati.

Questo controllore permette quindi di decidere quale priorità assegnare ad ogni interruzione. La priorità sta nel tipo dell'interruzione: il tipo può essere rappresentato su 8 bit (essendo i gate della IDT 256) e i suoi 4 bit più significativi rappresentano

la priorità. Quindi abbiamo 16 livelli di priorità e per ogni livello di priorità possiamo avere 16 interruzioni diverse.

Il controllore delle interruzioni si trova in parte sulla scheda logica e in parte nella CPU. In particolare i registri `IRR` (*Interrupt Request Register*) e `ISR` (*In Service Request*) si trovano nella CPU. Sono registri da 256 bit ciascuno. Ogni bit fa riferimento ad una singola entrata della tabella IDT.

In `IRR` ci stanno le pending requests indicizzate per tipo, invece che per piedino.

In `ISR` ci sono le richieste che il controllore assume essere in servizio, indicizzate per tipo invece che per piedino.

Continuamente il controllore compie queste azioni:

- monitora i piedini e quando c'è uno di questi piedini a 1, vede che tipo ha e setta il corrispondente bit in `IRR`;

- guarda se il bit settato che ha maggiore priorità in `IRR` ha anche maggiore priorità di tutti i bit in `ISR`. Se accade, invia la richiesta al processore. Una volta che il processore ha accettato la richiesta, il controllore invia il tipo e setta il corrispondente bit in `ISR`.

- se ne arriva una a priorità minore, rimane in `IRR`; se ne arriva una a maggiore priorità, invia anche quella. Questo meccanismo è detto *annidamento delle interruzioni*. Questo significa che le routine di interruzione possono essere a loro volta interrotte da richieste di interruzione a maggiore priorità.

Il controllore, in ogni caso, può arrivare a gestire due interruzioni di uno stesso tipo: una in `ISR` e una in `IRR`.

Tutto questo si assume accada se e solo se IF (*Interrupt Flag*) è settato. Ma chi si occupa di IF? Quando si esegue una `INT` vediamo, nella tabella IDT al gate corrispondente all'interruzione stessa, se deve essere resettato oppure rimanere settato. In particolare i gate Interrupt/Trap se sono Interrupt (I/T = 0) IF deve essere resettato, se sono Trap (I/T = 1) viene lasciato settato. Quindi se è la IDT che si occupa di IF, significa che chi decide se possono esserci oppure no interruzioni annidiate è il sistema (il quale si occupa di inizializzare la IDT). Possiamo farlo anche esplicitamente con le istruzioni `CLI` e `STI` che sono anch'esse privilegiate (possono essere eseguite solo in modo sistema).

Quando il software invia l'EOI, il controllore delle interruzioni azzera il bit a maggiore priorità in `ISR`. Assumiamo che il software si stia comportando bene perché assume che il software stesso abbia terminato veramente di gestire la richiesta a maggiore priorità. Quando in `IRR` ci sono richieste a maggiore priorità rispetto a quelle in servizio, incomincia a gestirle.

Abbiamo detto che ad ogni piedino è associato un registro a 64 bit. In realtà non è un unico registro ma, più correttamente, è una coppia di registri da 32 bit. Queste coppie di registri sono così organizzate:
Ogni coppia di registri contiene informazioni riguardanti l'interruzione:

- 8 bit per il tipo dell'interruzione proveniente da quel piedino;

- 1 bit P di polarità che serve a stabilire se il segnale della richiesta è attivo basso (bit a 1) oppure attivo alto (bit a 0);

- 1 bit S per stabilire la forma del segnale che costituisce la richiesta di interruzione (**livello**, bit a 1, oppure **impulso**, bit a 0);

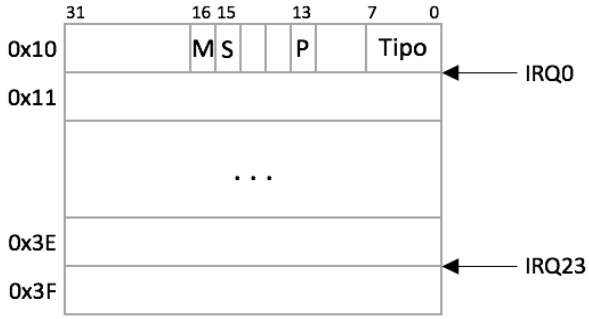


Figura 3.4: Organizzazione delle 24 coppie di registri da 32 bit

- 1 bit M di maschera che serve per mascherare, quando settato, le richieste di interruzione provenienti da quel piedino.

3.4.1 Soluzione al problema della esigua quantità di piedini dell'APIC

Poiché i piedini del controllore APIC sono solo 24 e 16 di questi sono già occupati da periferiche vecchie (come tastiera, timer, Hard-Disk...), si presenta la necessità di collegare più periferiche allo stesso piedino. Come è possibile adottare questa soluzione? Se collegassimo semplicemente più periferiche allo stesso filo, si genererebbe un cortocircuito quindi occorre dotare ogni periferica di un dispositivo che la scolleghi dal filo quando non deve inviare richieste di interruzione. Questo dispositivo si chiama ***open-collector***. Nelle periferiche, i piedini da cui viene inviato il segnale di interruzione, sono attivi bassi e si prestano ad essere collegati in OR all'ingresso di un altro circuito digitale (in questo caso l'APIC) tramite una resistenza (figura 3.5).

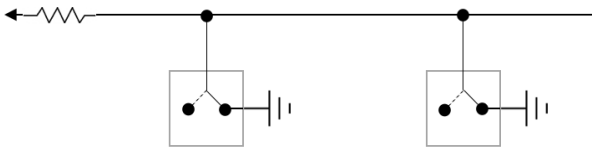


Figura 3.5: Open-Collector

Osserviamo, ora, come la forma con cui arriva il segnale della richiesta di interruzione, sia d'aiuto nella risoluzione del problema.

Il grafico in figura 3.6 mette in risalto il fatto che se abbiamo stabilito che il modo con cui arriva l'interruzione al piedino `IRQ1` è tramite un **impulso**, se ci fossero più periferiche collegate a questo piedino e se mentre una di queste viene servita, arrivasse un'altra richiesta d'interruzione da una diversa periferica (sempre collegata allo stesso piedino), l'APIC nemmeno se ne accorgerebbe perché è arrivata prima della EOI.

Il grafico in figura 3.7 mette in risalto il fatto che se abbiamo stabilito che il modo con cui arriva l'interruzione al piedino `IRQ1` è **livello**, se ci fossero più periferiche collegate a questo piedino e se mentre una di queste viene servita, arrivasse un'altra richiesta d'interruzione da una diversa periferica (sempre collegata allo stesso pie-

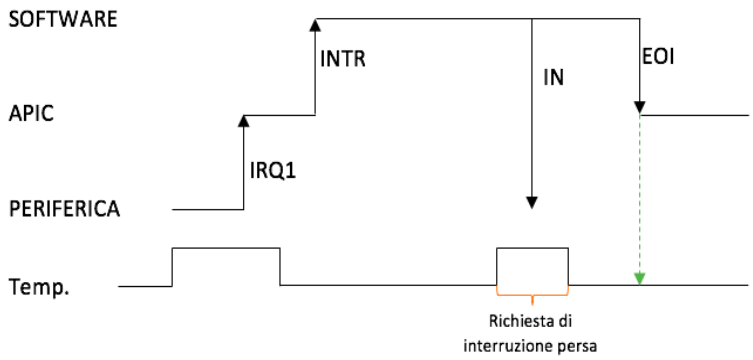


Figura 3.6: Situazione con segnale a *impulso*

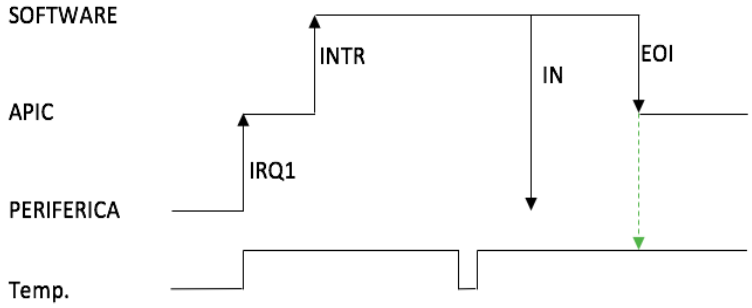


Figura 3.7: Situazione con segnale a *livello*

dino), l'APIC sa che finché il livello logico sul piedino è il livello con cui arrivano le interruzioni, deve eseguire le procedure per inviare il tipo dell'interruzione alla CPU.

Ora sorge un altro problema: come facciamo a sapere da quale delle periferiche collegate allo stesso piedino è arrivata l'interruzione? Lo possiamo sapere tramite la routine software andando, per esempio, ciclicamente a controllare i registri di stato delle periferiche collegate a quel piedino.

3.5 Tipi di gate nella IDT[3]

La tabella IDT (*Interrupt Descriptor Table*) è composta da 256 entrate (da 0 a 255). Le entrate di questa tabella prendono il nome di *gate*. Abbiamo due tipi di gate: **Interrupt/Trap** e **Task**. In realtà i gate sono tutti strutturati nello stesso modo; si differenziano nel momento in cui settiamo un bit invece che un altro oppure rendiamo significativa una parte di gate che per altri gate non lo è.

Un gate di tipo *Interrupt* si distingue da un gate di tipo *Trap* per il valore del bit IT, facente parte del campo TP che indica il tipo del gate. Se IT è resettato il gate è di tipo Interrupt se è settato è di tipo Trap.

Il gate di tipo *Trap* viene utilizzato quando si vuole che la routine venga eseguita con le interruzioni abilitate. Il gate di tipo Interrupt, invece, viene utilizzato quando si vuole che la routine venga eseguita con le interruzioni disabilitate. In entrambi i casi il campo ID non è significativo invece nel campo a 32 bit dell'indirizzo

[3]Questo paragrafo richiede la conoscenza del concetto di processo

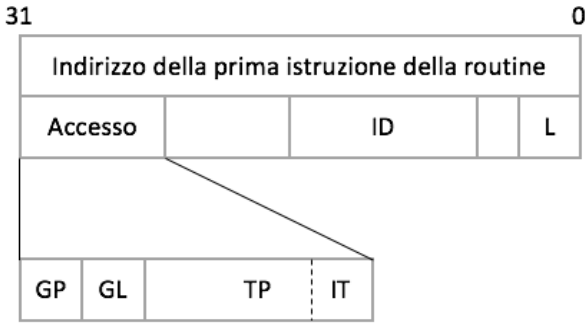


Figura 3.8: Struttura di un gate di interruzione

è contenuto l'indirizzo virtuale della prima istruzione della routine da mettere in esecuzione.

Un gate di tipo *Task*, invece, ha il campo dell'indirizzo della routine non significativo mentre il campo ID contiene l'identificatore del processo da mettere in esecuzione.

Il bit L, GP e GL hanno a che fare con il meccanismo di protezione avremo modo di approfondire la loro funzione più avanti.

3.6 Riepilogo: tipi di interruzione

Le interruzioni sono tipicamente classificate in base alla causa che le determina. Abbiamo:

- *interruzioni esterne* o *hardware*: vengono determinate da richieste che giungono al processore o tramite il piedino `/INTR` (*INTerrupt Request*), o tramite il piedino `/NMI` (*Not Maskable Interrupt*). Una richiesta che arriva tramite il piedino `/NMI` si traduce in un'effettiva interruzione (interruzione non mascherabile), mentre una richiesta che arriva tramite il piedino `/INTR` si traduce in un'effettiva interruzione solo se il flag IF (*Interrupt Flag*) è settato. Il valore del bit IF, oltre che da una interruzione che ha fatto uso di un gate di tipo Interrupt, può essere modificato via software attraverso le istruzioni `CLI` (*CLear Interrupt*) e `STI` (*SeT Interrupt*). Sono *asincrone* rispetto al programma.

- *interruzioni software*: sono prodotte dall'istruzione `INT`. Sono *sincrone* rispetto al programma.

- *single step trap*: sono prodotte al termine della fase di esecuzione di ogni istruzione solo se il bit TF (Trap Flag) è settato. Un'interruzione di questo tipo non viene prodotta dopo l'istruzione che ha provveduto a settare il bit TF. Sono *sincrone* rispetto al programma.

- *eccezioni del processore*: sono prodotte, da circuiterie di controllo interne al processore, ogni volta che si verifica una situazione anomala che impedisce il completamento dell'istruzione in corso. Sono *sincrone* rispetto al programma.

Si dicono *sincrone* rispetto al programma, le interruzioni che vengono provocate da un'istruzione sia esplicitamente che implicitamente (interruzioni software tramite

la `INT`, le single step trap dopo ogni istruzione, le eccezioni da un'istruzione che ha provocato una situazione anomala).

Si dicono *asincrone* rispetto al programma, le interruzioni che non vengono provocate da un'istruzione. Le interruzioni esterne possono arrivare in qualunque momento; fermo restando che il processore finisce di eseguire l'istruzione che sta eseguendo per poi passare alla prima istruzione del driver.

Indirizzo di ritorno

Quando arriva un'interruzione software, hardware o single step trap il processore termina l'istruzione che stava eseguendo (verrà salvato `EIP+1`).

Esistono tre tipi di eccezioni: *trap*, *fault*, *abort*. Per le prime il processore si comporta come con le normali interruzioni; per le seconde al ritorno dalla routine riesegue l'istruzione che ha causato il fault (questa possibilità comporta che gli effetti delle istruzioni sono provvisori fino al momento in cui si inizia ad eseguire la successiva), per le terze il processore provoca la terminazione forzata dell'esecuzione dell'istruzione senza che sia significativo il valore dell'indirizzo di ritorno.

Le eccezioni, inoltre, hanno il tipo obbligato perché vengono generate con quel tipo dalla CPU (ad esempio la routine di page-fault ha tipo `14`). Dal gate 20 in poi siamo liberi di riempire le entrate come preferiamo. Le eccezioni hanno il gate con livello di privilegio sistema (l'utente non deve poter fare `INT $14`).

Capitolo 4

Multiprogrammazione

4.1 Concetto di *multiprogrammazione*

Anche se un sistema di elaborazione con un solo processore può portare avanti un solo programma per volta, è possibile, ***a divisione di tempo***, portare avanti *in concorrenza* l'esecuzione di più programmi.

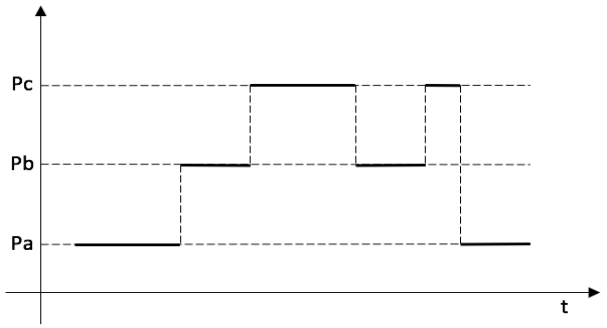


Figura 4.1: Esecuzione concorrente di più programmi Pa, Pb, Pc

Un sistema in grado di fare questo si dice **multiprogrammato**. Per rendere possibile la multiprogrammazione, un tale sistema di elaborazione deve prevedere la *virtualizzazione* del processore. Ad ogni programma corrisponde una **tabella di stato** che contiene una copia dei registri del processore. I registri all'interno di questa tabella sono detti *registri virtuali*.

Prendiamo, per esempio, i programmi P_A e P_B:

- quando il processore **reale** passa dall'esecuzione di P_A all'esecuzione di P_B, si dice che avviene una ***commutazione di contesto***;

- i valori attuali dei registri del processore reale vengono salvati nelle relative controparti presenti nella tabella di stato di P_A e nei registri del processore reale vengono caricati i valori presenti nelle relative controparti della tabella di stato di P_B.

- quando un programma va nuovamente in esecuzione, procede dal punto di arresto precedente (valore di `EIP` salvato $\longrightarrow$ istruzione successiva).

Da questo deduciamo che si hanno tanti processori virtuali quanti sono i programmi che devono essere eseguiti e, ogni processore virtuale è caratterizzato dai valori dei registri virtuali contenuti nella tabella di stato del relativo programma.

È bene notare che **un solo processore virtuale per volta può avere il possesso del processore reale**.

4.1.1 Da programma a *processo*

Si dice **processo**, o **task**, un programma in esecuzione su un processore virtuale e la sua tabella di stato prende il nome di **descrittore di processo**, o **descrittore di task**.

Si può dire anche che un processo è un'istanza di un programma, in quanto possiamo avere anche più processi istanze dello stesso programma che sono contemporaneamente in esecuzione; di volta in volta potranno essere diversi i dati su cui opera pur essendo il procedimento lo stesso. Un processo ha un proprio spazio di indirizzamento; questo spazio può coincidere o meno con lo spazio di indirizzamento del processore oppure essere un suo sottoinsieme.

In presenza di memoria virtuale i processi hanno tutti uguale spazio virtuale, differenti spazi fisici di memoria (sia fisica che di massa) e stesso spazio di I/O (non virtualizzato).

Ogni commutazione di contesto comporta una commutazione di spazio di indirizzamento fisico: per ottenere questo, basta cambiare, all'atto della commutazione, il valore di CR3 con il valore presente all'interno del descrittore di processo stesso. Il registro CR3 specifica l'indirizzo fisico del direttorio, tramite il quale è possibile accedere alle tabelle delle pagine che puntano alle pagine virtuali del processo. Questo comporta che a parità di indirizzi virtuali si potranno avere indirizzi fisici diversi da processo a processo.

I processi hanno in comune la zona di memoria dove risiede il sistema operativo e una zona di memoria per le comunicazioni fra i processi. Le pagine che contengono queste informazioni hanno per tutti i processi gli stessi indirizzi virtuali a cui corrispondono gli stessi indirizzi fisici. Perciò **il cambio di valore di `CR3` è ininfluente ai fini dell'esecuzione del sistema operativo**.

I processi, oltre che a singole pagine, possono condividere anche intere tabelle delle pagine. Nei direttori dei vari processi che condividono questa tabella comparirà lo stesso indirizzo fisico per quella tabella delle pagine.

N.B.: è teoricamente possibile che una pagina fisica condivisa corrisponda a una pagina virtuale avente indirizzo virtuale diverso da processo a processo; basta che nelle relative tabelle delle pagine compaia a indirizzi virtuali differenti, lo stesso indirizzo di memoria fisica. In ogni caso, per evitare di incorrere in problemi, ipotizziamo che pagine condivise abbiano lo stesso indirizzo virtuale corrispondente allo stesso indirizzo fisico.

4.2 Individuazione dei descrittori di processo nel processore PC

Un *descrittore di processo* è individuato da un ID di 16 bit (diverso da 0) e multiplo di 8 che costituisce anche l'identificatore del relativo processo.

Questo identificatore è utilizzato come indirizzo relativo all'interno di un'apposita tabella chiamata **GDT** (*Global Descriptor Table*) e seleziona un'entrata di 64 bit

(8 B) che contiene l'indirizzo virtuale iniziale e lo spiazzamento relativo al descrittore di processo. La base e il limite della tabella GDT sono contenuti nel registro `GDTR` (*Global Descriptor Table Register*) del processore.

I processi che la CPU è in grado di gestire sono al più 2^{13}.

Il registro `TR` (*Task Register*), presente nella CPU, è diviso in due parti:

- una parte *visibile* contenente l'ID del processo attualmente in esecuzione
- una parte *nascosta* contenente base e limite del descrittore del processo attualmente in esecuzione

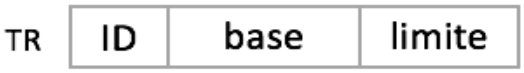


Figura 4.2: Registro TR

Alla parte nascosta possono avere accesso i microprogrammi di interruzione (hardware) o anche, ma più raramente, il microprogramma di commutazione fra task.

Quando avviene una commutazione di processo fatta dal sistema operativo, allora il sistema operativo stesso provvede a caricare in `TR`, nella sua parte visibile, l'identificatore del nuovo processo tramite l'istruzione `LDTR` (*LoaD Task Register*).

Quindi se l'interruzione è hardware, il descrittore di processo può essere individuato anche per mezzo della parte nascosta di `TR` (Figura 4.3).

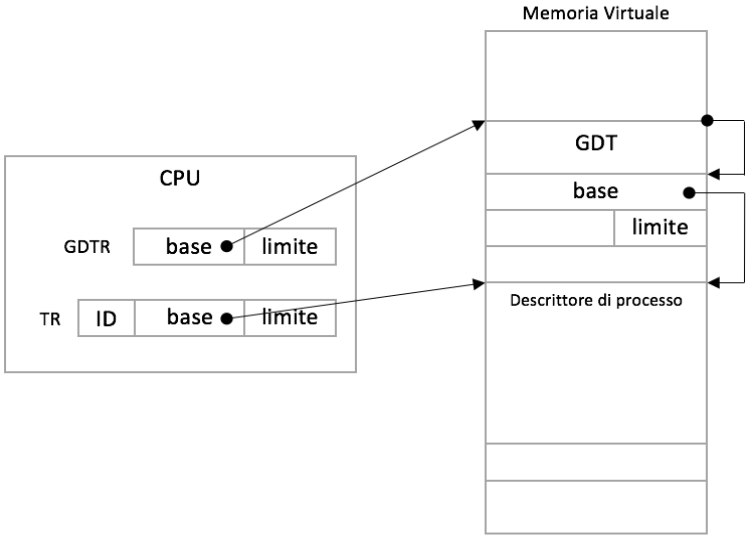


Figura 4.3: GDT, GDTR, TR e descrittore di processo

4.3 Commutazione hardware tra processi

La commutazione tra processi può avvenire per mezzo di un'interruzione hardware. Le interruzioni hardware che provocano commutazione di contesto devono **necessariamente** essere associate a gate di tipo Task. In questi gate non è significativo il campo dove nei gate Interrupt/Trap troviamo l'indirizzo della prima

istruzione della routine di interruzione, ma è significativo il campo ID che contiene l'ID del processo da mandare in esecuzione. Quando viene generata una siffatta interruzione avviene una **commutazione automatica fra processi**. Il processo attualmente in esecuzione assume il ruolo di *processo uscente* e viene sostituito dal processo da mettere in esecuzione che assume il ruolo di *processo entrante*.

Le azioni che vengono compiute dal microprogramma d'interruzione sono:

- salvataggio, in un registro d'appoggio, dell'identificatore del processo uscente contenuto nella parte visibile di `TR`;
- salvataggio dei contenuti dei registri del processore nel descrittore del processo uscente che è individuato dalla parte nascosta di `TR` (può accedervi perché questo meccanismo è hardware).
- caricamento, nella parte visibile di `TR`, dell'identificatore del processo entrante **prelevato dal gate di tipo task**; caricamento nella parte nascosta di `TR` di base e limite del descrittore del processo entrante **prelevati dalla GDT**;
- caricamento dei registri del processore con i nuovi valori prelevati dal descrittore del processo entrante, individuato da base e limite nella parte nascosta di `TR`;
- 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*) nel registro `EFLAG`;
- microsalto alla fase di chiamata.

Al momento della commutazione, il processo uscente viene "congelato" per cui, quando viene ripreso, si troverà nella stessa identica situazione in cui si trovava al momento della sua sospensione.

Quando viene incontrata l'istruzione `IRET`:

- se il bit NT è settato avviene una commutazione inversa fra i due processi;
- il processo uscente è quello attualmente attivo e il processo entrante è quello il cui identificatore è memorizzato nel campo link del descrittore di processo attualmente attivo;
- avviene la commutazione come visto in precedenza;
- il bit NT (*Nested Task*) viene resettato prima che il contenuto attuale del registro `EFLAG` del processore venga memorizzato nel descrittore del processo uscente.

Questo meccanismo consente l'*annidamento dei processi*.

4.4 Problema dei processi occupati

Ogni entrata della tabella GDT contiene un bit B (*Busy*) che viene gestito dal meccanismo automatico di commutazione fra processi con lo **scopo di impedire la ricorsione dei processi**. La motivazione sta nel fatto che un processo ha un solo

descrittore. Se la ricorsione fosse possibile si rischierebbe di spezzare un'eventuale catena di processi annidati. Un processo, per poter essere attivato dal meccanismo, deve avere il bit B a 0, deve cioè essere ***libero***; nell'atto dell'attivazione viene anche marcato come ***occupato*** mettendo il bit B a 1.

Un processo in esecuzione **rimane occupato** quando viene sospeso per effetto di una nuova interruzione.

Un processo sospeso **rimane occupato** se il suo identificatore è memorizzato nel campo link del processo in esecuzione e torna in esecuzione (rimanendo occupato) quando il processo attualmente in esecuzione esegue una istruzione `IRET`.

Un processo in esecuzione **viene marcato come libero** dall'istruzione `IRET` che viene eseguita quando termina la sua esecuzione.

Capitolo 5

Meccanismo di protezione

5.1 Utilità del meccanismo

Il ***meccanismo di protezione*** nasce come diretta conseguenza del meccanismo delle interruzioni. Un processo utente potrebbe, nel suo codice, chiedere al sistema di fare cose per lui che invece non avrebbe il permesso di fare, non tanto per il livello di privilegio inferiore, quanto invece per il fatto che proprio non dovrebbe poterle fare (potrebbe generare interruzioni software che, se venissero accettate, metterebbero in esecuzione delle routine con risultati disastrosi per il sistema o per il processo chiamante stesso). Si è elaborato allora nel tempo un sistema di protezione per il quale quando si verifica una incompatibilità di stati di privilegio, vengono lanciate eccezioni di protezione che mettono in esecuzione delle routine apposite per riparare alla chiamata non permessa. Si stabiliscono, pertanto, due ***livelli*** o ***stati*** di privilegio:

Stato o livello **sistema**	Stato o livello **utente**
- tutte le istruzioni	- istruzioni **non** privilegiate
- programmi di sistema	- programmi utente
- pila di sistema	- pila utente
- tutti i dati	- dati utente

Due punti, più degli altri, servono a distinguere i due livelli di privilegio: la possibilità di eseguire o meno certe istruzioni, la differenza di pila.

Abbiamo due tipi di istruzioni quelle ***non privilegiate*** e quelle ***privilegiate***.

Se in un programma compare un'istruzione privilegiata, il processore prima di eseguirla controlla se il livello di privilegio in cui si trova attualmente è compatibile con l'esecuzione di quella istruzione: se è così prosegue eseguendola come una normale istruzione, altrimenti lancia un'*eccezione di protezione*. Istruzioni privilegiate sono quelle di alt (`HLT`) e quelle che lavorano su registri speciali del processore come i *Control Registers*. Anche la `INVLPG` (*INVaLidate PaGe*), vista quando abbiamo parlato del TLB, è un'istruzione privilegiata. Quindi il processore può fermarsi solo se si trova a livello sistema (la `HLT` è privilegiata).

5.1.1 Diritti di accesso: regole di protezione

In ogni descrittore di pagina virtuale, come anche in ogni descrittore di tabella delle pagine, abbiamo la possibilità di specificare, attraverso il bit U/S (User/System) se quella pagina o tabella delle pagine (e di conseguenza tutte le 1024 pagine a cui essa punta), possono essere accedute solo a livello sistema oppure anche a livello utente. Un livello di protezione ulteriore per quella pagina o macropagina, è dato dal bit R/W che specifica se si ha il permesso di leggere e scrivere oppure solo di leggere.

Quindi in ogni descrittore di pagina virtuale il bit U/S specifica il livello di privilegio cui si deve trovare il processore per poter operare su quella pagina. Le istruzioni possono essere **prelevate** (*fetch*) solo se il processore ha livello di privilegio uguale al livello di privilegio di quella pagina (CPL=U/S).

Se invece dobbiamo **modificare** una pagina, occorre che il livello di privilegio sia maggiore o uguale a quello della pagina a cui dobbiamo apportare quella modifica (CPL>=U/S).

5.1.2 Pile e livelli di privilegio

Abbiamo due pile differenti per il modo sistema e per il modo utente. Questa scelta è data da due motivi:

1. una sola pila che si trova a livello sistema, proprio perché a livello sistema, non può essere acceduta da livello utente;

2. una sola pila che si trova a livello utente risulta inutilizzabile a livello sistema poiché, proprio perché a livello utente, non può essere utilizzata per contenere informazioni che riguardano lo stato sistema.

Se un processo è di livello sistema avrà un'unica pila, quella sistema. Se un processo è di livello utente avrà due pile, quella sistema e quella utente. Quindi quando si parla di pila in presenza di più di un processo occorre sempre specificare a quale livello di privilegio appartiene il processo e se è di livello utente a quale delle due pile stiamo facendo riferimento.

5.2 Cambiamento del livello di privilegio

Il livello di privilegio di un processo è memorizzato nel suo descrittore. Quando il processo va in esecuzione, il suo livello di privilegio viene trasferito nel registro relativo allo stato del processore. Si hanno **processi sistema** e **processi utente**.

Il livello di privilegio può essere modificato per mezzo del meccanismo delle interruzioni.

Un processo utente può portarsi *volontariamente* a livello sistema eseguendo un'istruzione `INT` (interruzione software) quando intende mandare in esecuzione una routine di sistema operativo che fornisca servizi che l'utente non può (o non deve) effettuare autonomamente.

Un processo utente può essere interrotto da un'interruzione hardware (esterna) che manda in esecuzione un driver che, per le azioni che compie, deve avere livello di privilegio sistema.

5.3 Protezione nel processore PC

Lo stato di privilegio in cui si trova a lavorare attualmente il processore è memorizzato un registro speciale che si chiama CPL (*Current Privilege Level*): se questo vale 0 il processore si trova in stato sistema, se invece vale 1 si trova in stato utente.

Il livello di privilegio di una pagina virtuale è dato da due bit presenti nel suo descrittore di pagina virtuale. Questi bit sono:

1. U/S (User/System): 1⟶stato sistema, 0⟶stato utente;
2. R/W (Read/Write): 0⟶sola lettura, 1⟶lettura e scrittura.

Il direttorio e le tabelle delle pagine si trovano in pagine che hanno livello di privilegio sistema. Allora come fa un processo utente ad accedere alla pagina virtuale su cui deve operare? Lo può comunque fare perché l'accesso alla pagina non lo fa direttamente, ma ci pensa la MMU che, essendo un oggetto hardware, può ugualmente avere accesso alle pagine (sempre ammesso che alla fine queste abbiano livello di privilegio utente).

Per modificare le pagine *contenenti la pila*, occorre che il livello di privilegio del processore quando si appresta ad accedere a quelle pagine sia lo stesso contenuto nel bit U/S delle pagine stesse (CPL = U/S).

Lo stato sistema è privilegiato rispetto allo stato utente anche se questa gerarchia non si rispecchia nella loro codifica all'interno del registro `CPL`.

La GDT, la IDT e l'array di descrittori di pagina fisica si trovano in pagine che hanno livello sistema.

Le istruzioni di I/O possono essere o meno privilegiate. Questo viene stabilito scrivendo un'opportuna configurazione nel campo IOPL (*IO Privilege Level*) del registro `EFLAG`. Ogni modifica del campo IOPL può essere fatta solo se il processore si trova a livello sistema.

Le istruzioni di abilitazione/disabilitazione delle richieste di interruzioni mascherabili, `CLI` (*CLear Interrupt*) e `STI` (*SeT Interrupt*), hanno lo stesso privilegio delle istruzioni di I/O e per questo si dice che sono *I/O sensitive*.

5.4 Interruzioni e protezione

Il descrittore di processo contiene, in posizione prefissata, un puntatore alla pila sistema del processo stesso. Questo campo è significativo solo se il processo è di livello utente.

Il registro `TR` (*Task Register*), nella sua parte nascosta cioè accessibile da hardware, contiene base e limite del descrittore di processo il cui ID è contenuto nella parte visibile del registro TR stesso. Se l'interruzione è software, il descrittore di processo può essere accessibile tramite l'ID del processo, contenuto nella parte visibile di TR, utilizzato come indice di accesso alla GDT (nella quale sono contenuti base e limite del descrittore di processo; la tabella GDT è raggiungibile attraverso il registro `GDTR` che contiene base e limite della GDT). In ogni caso, una volta raggiunto il descrittore, è possibile accedere al puntatore alla pila sistema che è contenuto in un campo del descrittore stesso.

Il registro `IDTR` del processore contiene base e limite della tabella IDT. Le entrate di questa tabella si chiamano gate, ognuno dei quali è fatto in questo modo:

Come già detto, i gate possono essere di due tipi: Interrupt/Trap o Task.

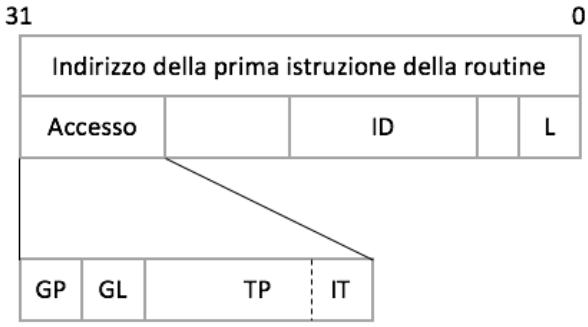


Figura 5.1: Struttura di un gate di interruzione

In entrambi i casi è significativo il bit L che rappresenta il livello di privilegio futuro. Nel byte di accesso di ciascun gate, sono presenti un bit di presenza GP e un bit di privilegio GL. Quando arriva un'interruzione viene esaminata, via hardware l'entrata della tabella IDT specificata dal tipo dell'interruzione e viene, come prima cosa, esaminato il bit GP. Se questo viene trovato a 0, viene generata un'eccezione per *tipo di interruzione non implementato*; vuol dire che quel gate non è stato inizializzato.

Successivamente viene esaminato il bit GL: se si tratta di un'interruzione software il valore di GL deve essere minore o uguale al livello di CPL altrimenti viene lanciata un'eccezione di protezione. Se invece si tratta di un'interruzione hardware o di un'eccezione, il livello di privilegio di GL può essere qualsiasi.

La altre azioni dipendono dal tipo del gate.

Analizziamo cosa avviene quando il gate è di tipo Interrupt/Trap. Le azioni che vengono svolte coinvolgono in particolare i valori del bit L e del registro del processore `CPL`. Quando incontriamo un gate di tipo Interrupt/Trap, viene esaminato il bit L e viene confrontato con il valore del registro `CPL`:

- se L < CPL viene generata un'eccezione per *cambiamento di privilegio non lecito.*
- se L = CPL
  - vengono memorizzati in pila i valori attuali di `EFLAG`, `CPL`, ed `EIP`;
  - viene caricato `CPL` con il valore di L (anche se sono uguali) e viene caricato `EIP` con l'indirizzo della prima istruzione della routine;
    - se il gate è di tipo Trap viene azzerato solo il bit TF (il bit IF ***non*** viene toccato);
    - se il gate è di tipo Interrupt viene azzerato sia il bit TF sia il bit IF.
  - viene azzerato il flag NT (*Nested Task*);
  - microsalto alle operazioni di chiamata.
- se L > CPL
  - viene salvato in un registro d'appoggio l'attuale valore di `ESP`;
  - `ESP` viene caricato con il valore del puntatore alla pila sistema prelevato dal descrittore di processo in esecuzione (acceduto attraverso base e limite del registro `TR`);

- memorizzazione nella nuova pila (che sarà di livello sistema) di una prima parola lunga non significativa e del puntatore alla pila utente (cioè il valore di `ESP` precedentemente salvato);
- memorizzazione nella nuova pila dei valori attuali dei registri `EFLAG`, `CPL`, `EIP`;
- viene caricato `CPL` con il valore di L e viene caricato `EIP` con l'indirizzo della prima istruzione della routine;
  - se il gate è di tipo Trap viene azzerato solo il bit TF (il bit IF ***non*** viene toccato);
  - se il gate è di tipo Interrupt viene azzerato sia il bit TF sia il bit IF.
- viene azzerato il flag NT (*Nested Task*);
- microsalto alle operazioni di chiamata.

Abbiamo cambiamento di privilegio solo se L > CPL e quindi, in questo caso, avremo anche un cambiamento di pila. La nuova pila avrà la seguente configurazione:

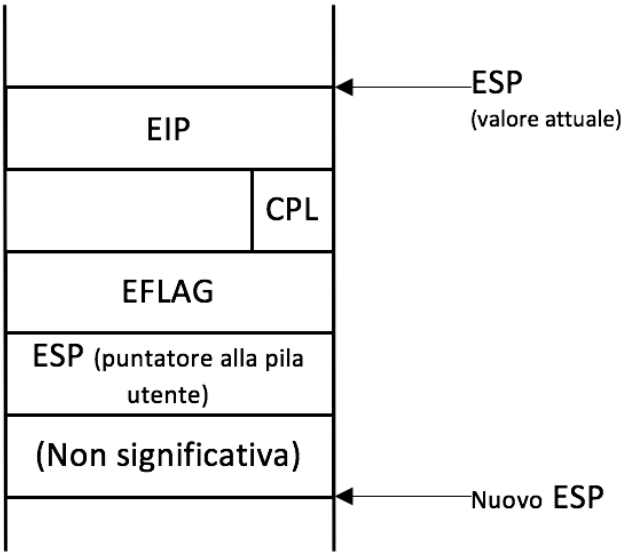


Figura 5.2: Configurazione della pila sistema

Un'interruzione hardware (esterna) o software, che comporta il trasferimento di una nuova quantità nel registro CPL, può provocare il passaggio da stato utente a stato sistema. **Non può avvenire l'inverso**. Il ritorno da stato sistema può avvenire, al termine della routine di interruzione, con l'esecuzione dell'istruzione `IRET`.

È bene specificare, dunque, che il cambiamento di livello di privilegio può avvenire solo da utente a sistema. Un processo sistema non può richiedere che venga abbassato il proprio livello di privilegio.

Un processo utente, quindi, non può darsi da solo un livello di privilegio superiore; può invocare solamente routine di sistema (primitive) attraverso il loro numero d'ordine (ossia il tipo dell'interruzione), oppure può essere interrotto da cause esterne con una conseguente modifica automatica del livello di privilegio stesso.

Le pagine in cui sono contenute le routine che vanno in esecuzione come conseguenza di un'interruzione, devono avere lo stesso livello di privilegio che ha attualmente il processore. Non sarebbe congruente con il meccanismo di protezione se il

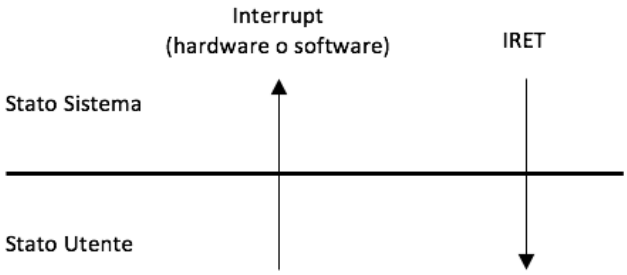


Figura 5.3: Meccanismo di interruzione e cambiamento di livello di privilegio

processore a livello sistema potesse eseguire routine di livello utente perché vorrebbe dire abbassare il livello di privilegio.

5.5 Problema del cavallo di Troia

Un processo utente può richiamare una primitiva di sistema, passandogli come parametri indirizzi di variabili (per esempio buffer di memoria). L'indirizzo trasmesso potrebbe riferirsi a una pagina virtuale con livello di privilegio sistema; il processo in stato utente non avrebbe il premesso di accedervi ma, se non si provvedesse ad evitare ciò, il processo potrebbe andare a modificare, con risultati disastrosi, importanti dati di sistema. Occorre dunque che la primitiva effettui alcune verifiche. Deve determinare:

- tramite la pila sistema, il valore di `CPL` del processo chiamante;
- tramite le tabelle delle pagine, determinare il livello di privilegio della pagina virtuale a cui si riferisce l'indirizzo trasmesso;
- in caso di non consistenza, negare l'accesso.

Una situazione tipica prevede che le pagine virtuali di livello sistema abbiano indirizzo virtuale che inizia con 1, invece le pagine virtuali di livello utente abbiano indirizzo virtuale che inizia con 0. In tal caso non si presenta la necessità di accedere alle tabelle delle pagine in quanto il livello di privilegio è già implicito nell'indirizzo trasmesso.

Capitolo 6

Nucleo Multiprogrammato

La multiprogrammazione col processore PC prevede un solo processore reale e più processori virtuali, la memoria virtuale paginata e che le pagine virtuali possano avere un livello di privilegio (o utente o sistema).

6.1 Processi nel nucleo multiprogrammato

Un processo ha un livello di privilegio proprio in base al quale distinguiamo se si tratta di un processo sistema o di un processo utente. **Solo un processo utente è in grado di portarsi volontariamente a livello sistema con una `INT` e poi tornare a livello utente con una `IRET`.**

Un processo è caratterizzato da:

- un descrittore;
- un corpo costituito da un *codice* e da un'*area dati privata*;
- un'*area dati comune*.

Ogni descrittore di processo (entità che si trova nello spazio comune a livello sistema) contiene:

- il valore iniziale del puntatore di pila per il livello sistema (significativo solo se il descrittore è di un processo utente);
- il contenuto del registro speciale `CR3`;
- il contenuto dei registri generali del processore.

Il codice di un processo, quando è costituito solo da istruzioni e costanti (non subisce modifiche), può essere condiviso da più processi che devono avere lo stesso livello di privilegio del processo che condivide il codice; in questo caso il codice deve trovarsi nello spazio comune.

L'area dati privata di ogni processo contiene le pile per i due livelli di privilegio se il processo è utente, solo la pila sistema se il processo è sistema.

L'area dati comune serve per accedere a informazioni comuni o a scambiare informazioni fra i processi.

I processi **iniziano** e **terminano**. Quando un processo inizia, vengono creati il suo descrittore e le sue pile (o la sua pila); viene creato anche il codice (se non

già esistente perché condiviso). Quando un processo termina, vengono distrutti il descrittore, le sue pile (o la sua pila) e il codice se non condiviso.

La commutazione di contesto è provocata da un'interruzione (hardware o software) che manda in esecuzione una routine con livello di privilegio sistema. Vengono memorizzati in pila i contenuti dei registri `EFLAG`, `CPL`, `EIP` che non hanno spazio nel descrittore di processo.

La commutazione di contesto avviene a livello sistema e prevede:

- memorizzazione del *processo uscente*;
- scelta del *processo entrante* (questo compito viene svolto da una routine specifica chiamata **schedulatore**);
- caricamento dello stato relativo al processo entrante.

La memorizzazione del processo uscente comporta:

- salvataggio, nel suo descrittore di processo, delle informazioni di stato fra cui il contenuto attuale dei registri della CPU;
- l'inserimento di un nuovo elemento, che specifica fra le altre cose l'identificatore del processo, in una determinata lista.

Abbiamo tre tipi di liste in cui un processo può trovarsi:

- lista dei ***processi pronti***;
- liste dei ***processi bloccati***;
- lista del ***processo in esecuzione***. Questa lista è ad elemento unico e punta all'elemento che identifica il processo attualmente in esecuzione.

Lo schedulatore opera solo sulla lista dei processi pronti avvalendosi anche di un'informazione di *precedenza*.

Le tre liste sopra elencate rappresentano anche gli stati in cui, ad ogni istante, può trovarsi un processo.

Un processo può trovarsi dunque:

- in **esecuzione** quando ha il controllo del processore;
- **pronto** quando è in attesa di essere mandato in esecuzione, tipicamente dallo schedulatore;
- **bloccato** quando è in attesa del verificarsi di una certa condizione di attivazione.

Quando un processo da bloccato diviene pronto, l'elemento che lo identifica viene rimosso dalla lista dei processi bloccati e inserito nella lista dei processi pronti.

6.2 Nucleo e interruzioni

Il **nucleo** di un sistema operativo **multiprogrammato** è costituito da un insieme di routine e di strutture dati in grado di gestire la multiprogrammazione.

Le routine e le strutture dati si trovano nello spazio comune a tutti i processi e **hanno livello di privilegio sistema**.

In un sistema personale, cioè *monoutente* e *multitasking*, i corpi dei processi sono costituiti dai vari programmi che l'utente attiva; il processo in esecuzione ha il controllo della tastiera e del video. Lo schedulatore, quando viene invocato esplicitamente dall'utente (click del mouse), seleziona il nuovo processo da mandare in esecuzione in base alle indicazioni fornite dall'utente stesso.

In un sistema *time-sharing* molti utenti condividono lo stesso elaboratore in maniera interattiva, mediante terminali o richieste di servizio. I corpi dei processi sono costituiti dai programmi dei vari utenti. Nel caso in cui più di un processo sia pronto per l'esecuzione, per mezzo di un contatore, lo schedulatore va in esecuzione a intervalli regolari e assegna l'elaboratore ad un utente per volta in modo ciclico.

In un sistema *in tempo reale* i processi devono essere mandati in esecuzione in seguito al verificarsi di certi eventi esterni, con i requisiti temporali imposti dall'applicazione. Ad ogni commutazione di contesto, lo schedulatore sceglie il processo a precedenza più alta.

Le **routine di nucleo** si dividono in tre categorie:

1. **routine di sistema** possono essere eseguite solo a livello sistema e in genere vanno in esecuzione in seguito al verificarsi di eccezioni come, per esempio, la routine di page-fault. Esistono routine di sistema che vanno in esecuzione ad intervalli regolari come la routine per le statistiche.

2. **primitive di nucleo** sono mandate in esecuzione a seguito di un'istruzione `INT`. Il microprogramma di interruzione verifica che la primitiva sia accessibile dal livello di privilegio dal quale è stata chiamata attraverso un confronto dell'attuale valore di `CPL` con il valore GL presente nel gate di interruzione. Se il valore del bit GL è minore o uguale del valore di `CPL`, allora la primitiva viene messa in esecuzione, altrimenti viene lanciata un'eccezione di protezione. Poiché le primitive possono essere chiamate sia livello utente che da livello sistema, il valore del bit GL rappresenterà logicamente il livello utente.

3. **driver** sono mandati in esecuzione da un'interruzione esterna (o hardware) e operano a livello sistema.

Alcune di queste routine possono determinare una commutazione di contesto.

Le **strutture dati** più rilevanti sono le liste dei processi. Ognuna di queste liste è costituita da elementi che contengono identificatori di processo e sono puntate da appositi puntatori contenuti in variabili di nucleo (globali). Molte routine lavorano sulle liste dei processi. Le liste dei processi sono risorse astratte e si trovano in uno stato **consistente** solo all'inizio e alla fine delle operazioni che le manipolano.

Un'interruzione esterna (hardware) può provocare l'esecuzione di una routine di nucleo che può andare a lavorare sulle liste dei processi. Poiché le liste dei processi si trovano in uno stato di inconsistenza durante le operazioni che vengono effettuate su di esse, è bene che lo svolgimento di queste non venga disturbato da nessuna interruzione; **si eseguono quindi con le interruzioni mascherabili disabilitate**. Questo viene fatto automaticamente in quanto le interruzioni che fanno partire le routine di nucleo operano su gate di tipo Interrupt i quali disabilitano le interruzioni.

6.2.1 Processi ed interruzioni

Il processo attualmente in esecuzione può produrre cambiamento di contesto:

- **volontariamente**, mediante la chiamata di una primitiva, effettuando delle richieste che non possono essere immediatamente soddisfatte; il processo viene bloccato e viene chiamato lo schedulatore.

- **involontariamente** perché si verifica un'interruzione esterna; in questo caso va in esecuzione un driver che:

  1. può effettuare le sue elaborazioni e poi terminare (il processo interrotto torna in esecuzione alla fine del driver);
  2. può forzare l'esecuzione di un altro processo;
  3. può far divenire pronto un processo bloccato;
  4. può semplicemente richiamare lo schedulatore.

Il cambiamento di contesto involontario è possibile solo se tutti i processi, quando non eseguono routine di nucleo, hanno le interruzioni mascherabili abilitate.

6.3 Realizzazione di una primitiva

Un processo può utilizzare le primitive di nucleo. Queste primitive si trovano a livello sistema. Poiché nei linguaggi ad alto livello, come il C++, non sono in genere possibili chiamate dirette alle primitive non essendo previsti costrutti per farlo, diciamo che le primitive sono *sottoprogrammi di interfaccia* scritti in Assembly (che permette di utilizzare le istruzioni `INT` e `IRET`).

Questo sottoprogramma di interfaccia esegue un'interruzione software (`INT $tipo`) provocando la messa in esecuzione della primitiva vera e propria (`a_primitiva`). L'indirizzo della prima istruzione della `a_primitiva` si trova nel gate di interruzione il cui numero è specificato nell'operando immediato dell'istruzione `INT`. La a_primitiva deve accedere al record di attivazione del chiamante per il prelievo dei parametri.

La `a_primitiva` termina con la `IRET`. Se non si sono avute commutazioni di contesto viene restituito il controllo al sottoprogramma di interfaccia il quale, a sua volta, terminando con la `RET` restituisce il controllo al programma che ha invocato la primitiva (Figura 6.1).

Avremo dunque una situazione del tipo:

```
1 //utente.cpp
2 //...
3 extern ''C'' void primitiva_i(/*parametri formali*/);
4 //...
5 void p(int h){
6         //...
7         primitiva_i(/*parametri attuali*/);
8         //ritp_i
9         //...
10 }
11
12 #utente .s
13 .TEXT
```

```
14 .GLOBAL primitiva_i      #sottoprogramma di interfaccia
15 primitiva_i:     INT     $tipo_i         #gate di tipo interrupt
16 rit_i:           RET
17
18 #sistema.s
19 .TEXT
20 a_primitiva_i:   #routine di tipo_i
21                  #...
22                  IRET
```

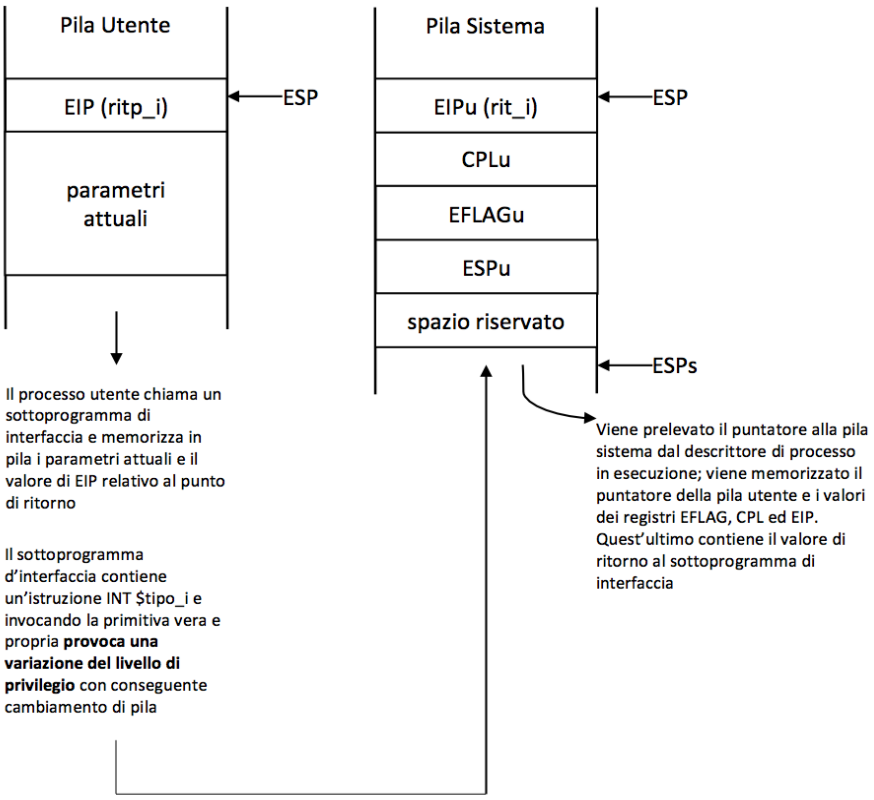


Figura 6.1: Pila utente e pila sistema dopo la chiamata a una primitiva da parte di un processo utente

Se la primitiva viene invocata da livello sistema, non abbiamo nessun cambiamento di livello di privilegio e quindi nessun cambiamento di pila (Figura 6.2).

In genere abbiamo quattro file: `utente.cpp`, `utente.s`, `sistema.s` e `sistema.cpp`. I moduli di livello utente non hanno nomi comuni con i moduli di livello sistema e non richiedono azioni di collegamento fra gli stessi. Questi moduli comunicano solo attraverso il meccanismo delle interruzioni.

6.3.1 Struttura di una `a_primitiva`

Una `a_primitiva` può provocare o meno una commutazione di contesto.
Se è possibile che la `a_primitiva` provochi una commutazione di contesto, la `a_primitiva`:

- **all'inizio** salva lo stato del processo (i valori di `EFLAG`, `CPL`, ed `EIP` che contiene il valore di ritorno alla subroutine di sistema, non fanno parte dello stato che

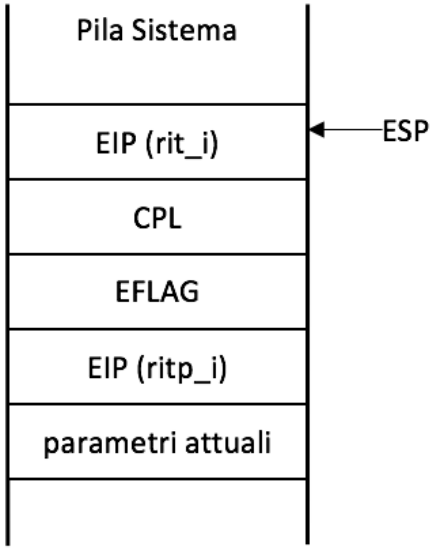


Figura 6.2: Pila sistema dopo la chiamata a una primitiva da parte di un processo sistema

viene salvato, essendo già stati memorizzati nella pila sistema dall'istruzione `INT` stessa);

- **alla fine** carica lo stato del nuovo processo (lo stato può anche coincidere con quello salvato all'inizio); dopo il caricamento del nuovo stato, l'istruzione `IRET` agisce sulla pila di sistema del nuovo processo.

Tutti i processi hanno la sommità della pila di sistema nella stessa situazione perché hanno abbandonato l'esecuzione o per effetto di un'interruzione software o per effetto di un'interruzione hardware. Quando si crea un processo occorre predisporre la pila di sistema in maniera che il suo contenuto sia compatibile con le operazioni svolte dalla `IRET`. Questo perché se la primitiva può produrre commutazione di contesto, se il processo che viene messo in esecuzione è anche un processo che non si è mai bloccato per un effetto di un'interruzione (hardware o software) o di un'eccezione, la `IRET` che agisce sulla pila del processo entrante, si ritroverebbe ad interpretare come `EFLAG`, `CPL`, `EIP` dei valori che in realtà non sono in nessun modo significativi.

L'istruzione `IRET` della `a_primitiva` produce il ritorno a un nuovo processo a partire dal suo indirizzo `rit_i` che tipicamente rappresenta l'istruzione successiva alla `INT` che ha invocato la primitiva stessa.

6.4 Realizzazione di processi

Per realizzare un processo ci serviamo di una struttura C++ che implementa un descrittore di processo. La struttura ha la seguente forma:

```
1 //sistema.cpp
2 struct des_proc{
3         natl id; //identificatore del processo
4         addr punt_nucleo; //valore iniziale del puntatore alla pila
              sistema; significativo solo se il processo e' utente
5         natl riservato;
```

```
6         addr CR3; //campo per il registro speciale CR3 che contiene l'
              indirizzo fisico del descrittore di processo
7         natl contesto[N_REG]; //campo destinato a contenere la copia
              del contenuto dei registri generali del processore. Questo
               campo ha dimensione N_REG
8         natl cpl; //livello di privilegio del processo
9 };
```

I descrittori di processo sono strutture definite nel nucleo. Un descrittore di processo è individuabile utilizzando l'ID del processo come indice d'entrata nella GDT e nella GDT nelle due parole lunghe successive troviamo la base e il limite del descrittore stesso.

Il processo è costituito da un identificatore, dal corrispondente descrittore e da un corpo (istanza di funzione).

I processi vengono gestiti attraverso liste, individuate da un puntatore e costituite da elementi di tipo `proc_elem`.

```
1 //sistema.cpp
2 struct proc_elem{
3         natl id; //identificatore del processo
4         natl precedenza; //priorita' del processo
5         proc_elem* puntatore; //puntatore che serve per mantenere una
              lista di tipo proc_elem
6 };
7
8 extern proc_elem* esecuzione; //puntatore al processo in esecuzione
9 extern proc_elem* pronti; //puntatore alla lista dei processi pronti
```

6.4.1 Attivazione di un processo

L'attivazione (creazione) di un processo utente, nel caso più semplice avviene in fase di inizializzazione (in molti sistemi è consentito che un processo padre crei dei processi figli).

Le azioni compiute sono le seguenti:

- selezione di un identificatore;
- inizializzazione del corrispondente descrittore e delle corrispondenti pile;
- allocazione in memoria dinamica di un nuovo elemento di tipo `proc_elem`;
- inserimento dell'elemento, in ordine di priorità, nella lista dei processi pronti.

Tali azioni vengono svolte dalla primitiva `activate_p(...)`:

```
1 natl activate_p(void f(int), int a, natl prio, natl liv){...}
```

Questa primitiva se termina correttamente restituisce l'ID del processo creato, altrimenti `0xFFFFFFFF`. L'argomento `prio` specifica la priorità che deve avere il processo. L'argomento `liv` specifica il livello di privilegio proprio del processo da creare.

L'argomento `f` è la funzione di cui il processo sarà istanza e l'argomento `a` sarà il parametro attuale della funzione di cui il processo sarà istanza.

Quindi per attivare un processo occorre richiamare tale primitiva con i relativi parametri. In ogni caso, la primitiva `activate_p` **consente la creazione di processi che non abbiano livello di privilegio o priorità maggiori di quelli del**

processo che richiama la primitiva stessa.

Vengono inizializzate la pila utente e la pila sistema (Figura 6.3).

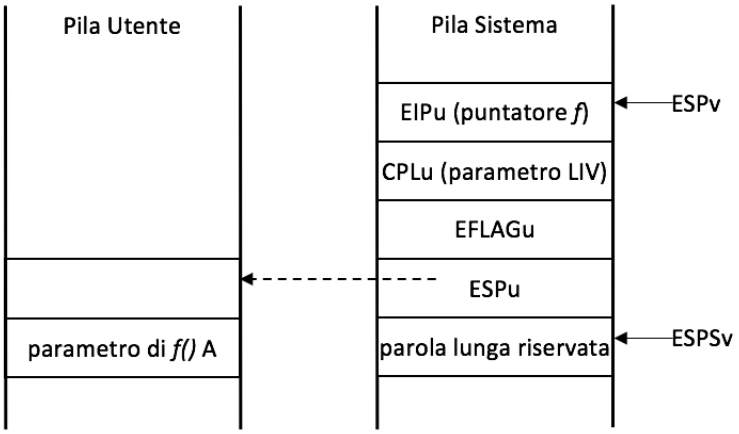


Figura 6.3: Creazione di un processo: inizializzazione delle pile

Il processo creato viene posto nella lista dei processi pronti.
Il processo da mandare in esecuzione viene scelto dallo schedulatore. Lo schedulatore non fa nient'altro che rimuovere dalla lista pronti il processo a più alta priorità (dalla testa della lista) e metterlo in esecuzione. Una volta scelto il processo da mettere in esecuzione, viene caricato il suo stato e viene eseguita l'istruzione `IRET` (che in questo caso rappresenta una specie di istruzione di chiamata).

6.4.2 Terminazione di un processo

La terminazione di un processo utente avviene per mezzo della primitiva `terminate_p()`, che non ha parametri. La `terminate_p()` richiama semplicemente lo schedulatore. **La chiamata alla primitiva `terminate_p()` è l'ultima istruzione di ogni processo**.

Nella fase di inizializzazione viene anche creato un processo `start_utente` che richiama la funzione `main()`. Con la funzione `main()`, il processo `start_utente`, attiva tutti i processi utente. Il processo `start_utente` termina, come tutti i processi, eseguendo la primitiva `terminate_p()`.

6.4.3 Area dati condivisa

Gli identificatori di processo vengono comunemente memorizzati in variabili condivise (non private dei singoli processi). Un processo può utilizzare primitive che agiscono esplicitamente su un altro processo; per esempio un processo può provocare la terminazione forzata di un altro processo.

6.5 Processo *Dummy*

Fra i processi pronti ve ne è uno con precedenza inferiore a quella di tutti i processi, il cosiddetto processo ***dummy***. È un processo di sistema che ha lo scopo di andare in esecuzione quando tutti i processi sono bloccati. Questo processo si trova a

livello sistema perché l'utente non deve alterarlo. È il primo processo ad essere creato in fase di inizializzazione prima di `start_utente` e, come tutti i processi, gira con le interruzioni abilitate. Quando tutti i processi sono bloccati, il processo dummy controlla una variabile di nucleo, "processi", che è inizializzata a 0 ed è incrementata dalle chiamate ad `activate_p()` e decrementata dalle chiamate a `terminate_p()`.

Se vi sono processi bloccati (processi>1) il processo dummy attende che uno dei processi bloccati venga risvegliato; se tutti i processi sono terminati (processi = 1, dove 1 è il processo dummy), il processo dummy stesso termina invocando una particolare primitiva `end_program()` che effettua le operazioni di chiusura.

La situazione in cui vi sono processi bloccati può essere modificata solo da un'interruzione esterna che manda in esecuzione la routine che fa passare il processo da bloccato a pronto (deve richiamare lo schedulatore).

6.6 Codice delle primitive

La chiamata a una primitiva (sottoprogramma di interfaccia) comporta l'esecuzione di una `a_primitiva` (primitiva vera e propria). Una `a_primitiva` può provocare o meno cambio di contesto: nel primo caso all'inizio occorre salvare lo stato attuale e alla fine caricare il nuovo stato.

Una primitiva può avere dei parametri (per i processi utente si trovano nella pila utente). Questi parametri, se comprendono indirizzi, devono essere sottoposti a controllo da parte della `a_primitiva` per evitare il problema del "Cavallo di Troia". Questi controlli sono in genere implementati con linguaggio Assembly in quanto coinvolgono i registri del processore. Le elaborazioni compiute da una `a_primitiva` possono essere più comodamente espresse in un linguaggio di alto livello come il C++. Chiamiamo questa parte elaborativa `c_primitiva`. Occorre rispettare gli standard di aggancio delle parti in Assembly ai sottoprogrammi scritti con un linguaggio di alto livello e, in particolare, vanno ricopiati in testa alla pila sistema i parametri che, per i processi utente, si trovano nella pila utente.

Quindi una `a_primitiva` + `c_primitiva` ha la seguente forma:

```
1  #sistema.s
2  .TEXT
3  salva_stato:    #...
4                  RET
5
6  carica_stato:   #...
7                  RET
8
9  .EXTERN c_primitiva_i
10 a_primitiva:    CALL    salva_stato
11                 #eventuale verifica e ricopiamento dei parametri
12                 CALL    c_primitiva_i
13                 #eventuale ripulitura della pila
14                 CALL    carica_stato
15                 IRET
16
17 //sistema.cpp
18 extern ''C'' void c_primitiva_i(/*parametri formali*/){
19         //...
20 }
```

Il ricopiamento dei parametri avviene dalla pila utente o dalla pila sistema in base al vecchio calore di `CPL` (memorizzato in pila sistema).

L'eventuale accesso alla pila utente da livello sistema non comporta nessuna violazione delle regole di privilegio in quanto da livello sistema la pila utente viene riferita come un insieme di pagine dati e non vengono utilizzati per accedervi né EBP né ESP. Inoltre questo è possibile poiché la pila utente va solo letta e non modificata.

La ripulitura della pila non è indispensabile poiché subito dopo avviene il caricamento di un nuovo stato (precedentemente salvato), e quindi il trasferimento in ESP di un nuovo valore.

Il sottoprogramma salva_stato memorizza il contenuto dei registri del processore nel descrittore di processo il cui identificatore è contenuto nell'elemento puntato da esecuzione (i registri EFLAG, CPL, EIP si trovano nella pila sistema e non è previsto spazio per loro nel descrittore di processo). Il salvataggio dei valori di CR3 ed ESP produce il salvataggio dell'intera pila. Il valore di ESP che deve essere salvato non corrisponde al valore attuale: esso va opportunamente incrementato poiché in testa alla pila è memorizzato l'indirizzo di ritorno dal programma salva_stato.

La c_primitiva deve lasciare nella lista ad unico elemento esecuzione, l'indirizzo del proc_elem che identifica (tramite il campo ID) il processo da mandare esecuzione.

Il sottoprogramma carica_stato, tramite il puntatore esecuzione copia in TR il valore di ID. Tramite la tabella GDT, carica nei registri reali del processore i contenuti dei corrispondenti registri virtuali prelevati dal campo contesto del descrittore appartenente al processo che deve andare (o tornare) in esecuzione. Poiché potrebbe esservi stata commutazione di contesto, il sottoprogramma carica_stato deve provvedere a trasferire dalla vecchia pila sistema alla nuova pila sistema il suo indirizzo di ritorno per poter terminare correttamente.

Una primitiva può anche restituire un valore. Il risultato, che viene prodotto dalla c_primitiva, viene lasciato in EAX. Nel caso in cui sia possibile una commutazione di contesto (la c_primitiva potrebbe invocare lo schedulatore), occorre:

- preliminarmente salvare l'identificatore del processo che ha invocato la primitiva;
- alla fine della c_primitiva, salvare EAX nel descrittore di tale processo.

Se una primitiva non effettua mai commutazione di contesto, il salvataggio e ripristino dello stato del processo chiamante, viene sostituito con salvataggio e ripristino del valore dei registri utilizzati; **se la primitiva produce un risultato non si deve salvare e ripristinare EAX**.

Per implementare una c_primitiva viene fatto uso di alcune funzioni di utilità definite nel file sistema.cpp:

- `void inserimento_lista(proc_elem*& p_lista, proc_elem* p_elem){...}`: inserisce nella lista puntata da p_lista (mantenuta in ordine decrescente di priorità), il processo identificato dall'elemento proc_elem puntato da p_elem come ultimo elemento fra quelli di uguale priorità alla sua;
- `void rimozione_lista(proc_elem*& p_lista, proc_elem* p_elem){...}`: rimuove dalla lista il primo elemento che è anche quello a più alta priorità;
- `extern "C" void inspronti(){...}`: inserisce nella lista pronti l'elemento che identifica il processo attualmente in esecuzione;

- `extern "C" void schedulatore(){...}`: estrae dalla lista dei processi pronti il primo elemento che è anche quello a più alta priorità e lo mette in `esecuzione` (`esecuzione` = `p_elem`).

6.7 Concetto di *atomicità*

In un sistema multiprogrammato alcune operazioni compiute da un processo su una risorsa, debbono avvenire in modo ***atomico*** cioè l'operazione compiuta sulla risorsa deve essere effettuata dall'inizio alla fine senza che un altro processo possa inserirsi e provocare alterazioni indesiderate all'operazione stessa.

Una **sezione critica** è costituita da un insieme di istruzioni che operano su una risorsa condivisa e che deve essere eseguita dal processo in maniera atomica.

A questo punto occorre fare una considerazione. In un sistema monoprogrammato (un processo solo), un'operazione su una risorsa viene sempre effettuata in modo atomico; pertanto la risorsa su cui va a operare si troverà sempre in uno stato **consistente**. Nel nostro caso, però, ci troviamo a dover lavorare con un sistema multiprogrammato che è in grado di gestire fino a 2^{13} processi. Occorre precisare che atomicità e non interrompibilità non sono sinonimi. Infatti un'azione non interrompibile è atomica ma non è vero l'inverso. Può infatti accadere che mentre un processo sta compiendo un'operazione su una risorsa condivisa, se le interruzioni mascherabili non sono disabilitate, può essere interrotto e il processo entrante potrebbe trovare la risorsa condivisa in uno stato **inconsistente**. Questo non deve assolutamente accadere e per evitarlo occorre che quando un processo si trova ad eseguire una sezione critica venga messo nella condizione di non essere disturbato fino alla fine della sezione. Per questo motivo all'inizio di ogni sezione critica vengono disabilitate le interruzioni mascherabili per poi essere riabilitate al termine di questa. Agendo direttamente sulle interruzioni, tutte le sezioni critiche diventeranno ***mutuamente esclusive***.

Questa tecnica è adatta quando abbiamo a che fare con sezioni critiche corte per le quali non è ragionevole studiare metodi più sofisticati e complessi. Le routine di nucleo sono sezioni critiche corte e pertanto vengono tutte eseguite con le interruzioni disabilitate.

Nei processi utente possiamo avere sezioni critiche lunghe e per non tenere disabilitate le interruzioni per troppo tempo, occorre ricorrere a tecniche più flessibili che non vadano ad agire direttamente sul meccanismo delle interruzioni.

6.8 Semafori

La ***mutua esclusione*** può essere gestita mediante un semaforo e due primitive di nucleo `sem_wait()` e `sem_signal()`. Un semaforo di mutua esclusione serve per proteggere una risorsa e possiede un contatore inizializzato a 1.

Un processo che vuole utilizzare una risorsa per compiervi un'azione, se trova che la risorsa è occupata perché un altro processo vi sta compiendo un'altra operazione, si blocca volontariamente sul semaforo della risorsa. Quando la risorsa sarà libera, e quindi in uno stato consistente, il processo bloccato diventerà pronto per l'esecuzione.

6.8.1 Realizzazione dei semafori

Il tipo *descrittore di semaforo* (`des_sem`) è una struttura con un campo `counter` per la gestione del semaforo mediante le primitive `sem_wait()` e `sem_signal()` e un campo `pointer` che punta la lista relativa ai processi che si bloccano sul semaforo stesso.

```
1 //sistema.cpp
2 struct des_sem{
3         int counter;
4         proc_elem* pointer;
5 };
```

I descrittori di semaforo sono variabili definite nel nucleo. Esse sono raggruppate in un array di descrittori di semaforo.

I semafori vengono inizializzati dal programma `main()` utilizzando la primitiva `sem_ini()`; essa ha come parametro il valore iniziale del contatore e restituisce l'indice del semaforo selezionato all'interno dell'array di descrittori di semafori oppure `0xFFFFFFFF` se non vi era nessun elemento disponibile.

La `sem_ini()` è implementata nel seguente modo:

```
1 //utente.cpp
2 extern ''C'' int sem_ini(int val);
3 natl semaforo_i;
4 int main(){
5         semaforo_i = sem_ini(VALORE);
6         //...
7         terminate_p();
8 }
9
10 #utente.s
11 .TEXT
12 .GLOBAL sem_ini
13 sem_ini:        INT     $tipo_si #gate di tipo interrupt
14                 RET
15
16 #sistema.s
17 .TEXT
18 a_sem_ini:      #routine di tipo_si
19                 #...
20                 IRET
```

Gli identificatori dei semafori vengono memorizzati in variabili condivise per poter essere utilizzabili da tutti i processi.

6.8.2 Primitive semaforiche

Primitiva `sem_wait()`

La primitiva `sem_wait(natl sem)` ha come parametro l'indice del semaforo su cui opera. Essa decrementa il contatore del semaforo e ne effettua il test: se il nuovo valore è minore di 0, il processo che l'ha invocata viene bloccato sulla coda del semaforo, viene richiamato lo schedulatore e mandato in esecuzione un nuovo processo.

Questa primitiva ha la seguente implementazione:

```
1 //utente.cpp
2 extern ''C'' void sem_wait(natl sem);
3 natl proc;
```

```
4 natl semaforo;
5 void p_code(int n){
6         //...
7         sem_wait(semaforo);
8         //...
9 }
10
11 int main(){
12         //...
13         semaforo = sem_ini(1);
14         proc = activate_p(p_code, A, PRIO, LIV);
15         //...
16 }
17
18 #utente.s
19 .TEXT
20 .GLOBAL sem_wait
21 sem_wait:       INT     $tipo_w
22                 RET
23
24 #sistema.s
25 .TEXT
26 .EXTERN c_sem_wait
27 a_sem_wait:     #routine di tipo_w
28                 CALL    salva_stato
29                 #ricopiamento del parametro sem
30                 CALL    c_sem_wait
31                 #pulizia della pila (inutile)
32                 CALL    carica_stato
33                 IRET
34
35 //sistema.cpp
36 extern ''C'' void c_sem_wait(natl sem){
37         des_sem* s;
38         s = &array_dess[sem];
39         s->counter--;
40         if(s->counter < 0){
41                 inserimento_lista(s->pointer, esecuzione);
42                 schedulatore();
43         }
44 }
```

Primitiva `sem_signal()`

La primitiva `sem_signal(natl sem)` ha come parametro l'indice del semaforo su cui opera. Essa incrementa il contatore del semaforo e ne effettua il test: se il nuovo valore è minore o uguale a 0 (vi sono cioè processi bloccati sul semaforo), quello a precedenza più alta viene tolto dalla coda del semaforo e inserito nella coda dei processi pronti, il processo in esecuzione viene anch'esso inserito in coda pronti, viene richiamato lo schedulatore che manda in esecuzione il nuovo processo.

Questa primitiva ha la seguente implementazione:

```
1 //utente.cpp
2 extern ''C'' void sem_signal(natl sem);
3 natl proc;
4 natl semaforo;
5 void p_code(int n){
6         //...
7         sem_signal(semaforo);
```

```
 8         //...
 9 }
10
11 int main(){
12         //...
13         semaforo = sem_ini(1);
14         proc = activate_p(p_code, A, PRIO, LIV);
15         //...
16 }
17
18 #utente.s
19 .TEXT
20 .GLOBAL sem_wait
21 sem_wait:        INT     $tipo_w
22                  RET
23
24 #sistema.s
25 .TEXT
26 .EXTERN c_sem_signal
27 a_sem_signal:    #routine di tipo_s
28                  CALL    salva_stato
29                  #ricopiamento del parametro sem
30                  CALL    c_sem_signal
31                  #pulizia della pila (inutile)
32                  CALL    carica_stato
33                  IRET
34
35 //sistema.cpp
36 extern ''C'' void c_sem_signal(natl sem){
37         des_sem* s;
38         s = &array_dess[sem];
39         proc_elem* lavoro;
40         s->counter++;
41         if(s->counter <= 0){
42                 rimozione_lista(s->pointer, lavoro);
43                 inserimento_lista(pronti, lavoro);
44                 inserimento_lista(pronti, esecuzione);
45                 schedulatore();
46         }
47 }
```

6.8.3 Semafori di mutua esclusione e semafori di sincronizzazione

Abbiamo due tipi di semafori che hanno due scopi e modi di utilizzo totalmente diversi fra loro.

Mutua Esclusione
inizializzato a 1

P1	P2
`sem_wait(sem)`	`sem_wait(sem)`
`<sezione critica>`	`<sezione critica>`
`sem_signal(sem)`	`sem_signal(sem)`

scopo: evitare che due processi accedano ad una stessa risorsa contemporaneamente; proteggere la risorsa.

Sincronizzazione
inizializzato a 0

P1	P2
`sem_wait(sem)`	`sem_signal(sem)`

scopo: serve a fare in modo che P1 attenda che accada qualcosa. Questo qualcosa lo farà accadere P2 e quando succede lo segnala facendo sbloccare P1. Oppure se P2 ha già eseguito la `sem_signal()`, P1 non si blocca.

6.9 *Preemption*

Si ha *preemption* (prelazione) quando un processo da bloccato diviene pronto e viene messo in lista pronti; quando il processo in esecuzione viene messo in lista pronti; quando viene chiamato lo schedulatore.

Il processo sbloccato, se ha precedenza maggiore di quello in esecuzione, va in esecuzione al posto di quello attualmente in esecuzione.

6.10 Transizioni di stato di un processo

Un processo in esecuzione può bloccarsi perché esegue una `sem_wait()`. Un processo bloccato può divenire pronto per effetto di una `sem_signal()` eseguita da un altro processo. Una transizione tra pronto ed esecuzione (e viceversa) può avvenire o quando viene invocato (direttamente o indirettamente) lo schedulatore oppure per effetto di un'interruzione esterna.

6.11 Memoria dinamica

In un sistema multiprogrammato la memoria dinamica **deve essere gestita in mutua esclusione**. Facciamo uso di due funzioni:

- `void* mem_alloc(natl dim);`
  alloca una struttura di *dim* byte e restituisce il puntatore al primo byte allocato (nel caso di allocazione di un tipo struttura, il numero dei byte allocati viene determinato applicando l'operatore *sizeof* al tipo struttura stesso).

- `void mem_free(void* pv);`
  libera la struttura indirizzata da `pv`, che deve essere stata allocata utilizzando la funzione `mem_alloc()` (e quindi ha associata l'informazione sul numero di byte da liberare).

La mutua esclusione viene garantita dall'uso di un semaforo `mem_mutex`, inizializzato a 1 dal processo main.

Le funzioni `mem_alloc()` e `mem_free()` hanno la seguente struttura:

```
1 //utente.cpp
2
3 natl mem_mutex;
4
5 void* mem_alloc(natl dim){
6         void* p;
7         sem_wait(mem_mutex);
8         //...
9         sem_signal(mem_mutex);
10        return p;
11 }
12
13 void mem_free(void* pv){
14        sem_wait(mem_mutex);
15        //...
16        sem_signal(mem_mutex);
17 }
```

Un processo per poter utilizzare la funzione `mem_alloc()` deve convertire il tipo del risultato da puntatore a void a puntatore:

```
1 //utente.cpp
2
3 void p(int i){ //codice di un processo
4         struct struttura{/*...*/};
5         struttura* ps;
6         ps = static_cast<struttura*>(mem_alloc(sizeof(struttura)));
7         mem_free(ps);
8 }
```

Il meccanismo della memoria dinamica può essere usato anche da alcune routine di nucleo per le quali è garantita la mutua esclusione in quanto girano con le interruzioni mascherabili disabilitate.

Quindi è opportuno dichiarare e definire due funzioni che abbiano come compito quello di allocare e deallocare la memoria dinamica a livello sistema:

```
1 void* alloca(natl dim){/*...*/}
2 void dealloca(void* p){/*...*/}
```

N.B.: nessuna di queste funzioni utilizza gli operatori di libreria C++ `new` e `delete` poiché il supporto a run-time del linguaggio di cui fanno uso (cioè alcune primitive di sistema operativo su cui il C++ è installato), in questa versione del nucleo non è implementato.

Quindi la memoria dinamica di un processo interessa due zone della memoria virtuale, una con livello di privilegio sistema e una con livello di privilegio utente.

6.12 Realizzazione di un timer di sistema

Per realizzare un timer di sistema occorre un'interfaccia di conteggio. Essa possiede un contatore inizializzato con il valore di CTR (registro a 16 bit). Quando il conteggio arriva a 0, l'interfaccia attiva un piedino di uscita, che viene utilizzato per generare una richiesta di interruzione, e viene nuovamente inizializzato CTR. L'interfaccia possiede anche un registro di stato `STR` che consente di leggere il valore

attuale del contatore e un registro `CVR` che serve per specificare la modalità di funzionamento dell'interfaccia. In questo caso utilizziamo la modalità *conta-tempi* che viene usata per realizzare un **timer di sistema**.

L'interfaccia di conteggio viene predisposta con una frequenza di tempo (correlata con la frequenza di pilotaggio) in modo da avere un'interruzione ogni 50ms.

6.12.1 Autosospensione dei processi

In un sistema in tempo reale, i processi possono sospendersi per un certo numero di intervalli di tempo ciascuno lungo 50ms. Viene creato, a livello sistema, un puntatore (`p_sospesi`) a una lista di attesa composta da elementi che individuano i processi che si sospendono sul timer.

Il processo che vuole sospendersi, invoca la primitiva `delay()`, specificando il numero di intervalli di tempo per cui intende rimanere bloccato. Trascorso questo tempo, il processo diviene nuovamente pronto.

A livello sistema abbiamo la seguente struttura dati:

```
1 //sistema.cpp
2 struct richiesta{
3         natl d_attesa;
4         richiesta* p_rich;
5         proc_elem* pp;
6 };
7
8 richiesta* p_sospesi; //puntatore alla lista dei processi sospesi.
```

La primitiva `delay()` crea una nuova variabile di tipo richiesta, la inizializza e la inserisce nella lista puntata da `p_sospesi`.

```
1 //utente.cpp
2 extern ''C'' void delay(natl n);
3 void procd(int i){ //codice di un processo
4         //...
5         delay(N);
6         //...
7 }
8
9
10 #utente.s
11 .TEXT
12 .GLOBAL delay
13 delay:  INT     $tipo_d #gate di tipo interrupt
14         RET
15
16 #sistema.s
17 .TEXT
18 .EXTERN c_delay
19 a_delay:        #routine di tipo_d
20                 CALL    salva_stato
21                 #ricopiamento del parametro N
22                 CALL    c_delay
23                 #ripulitura pila
24                 CALL    carica_stato
25                 IRET
26
27 //sistema.cpp
28 void inserimento_lista_attesa(richiesta* p);
29 extern ''C'' void c_delay(natl n){
30         richiesta* p;
```

```
31          p = static_cast<richiesta*>(alloca(sizeof(richiesta)));
32          p->d_attesa = n;
33          p->pp = esecuzione;
34          inserimento_lista_attesa(p);
35          schedulatore();
36  }
```

La primitiva `delay()` fa uso della funzione di utilità `inserimento_lista_attesa()` che ha come parametro il puntatore all'elemento di tipo richiesta da inserire.

La lista di attesa è organizzata in modo tale che il numero degli intervalli di attesa di un certo elemento sia dato dalla somma dei campi `d_attesa` di tutti gli elementi che lo precedono più quello dell'elemento stesso.

Questa organizzazione rende più semplice la gestione della lista da parte del driver di interruzione, che si limita a decrementare il contenuto del campo `d_attesa` del primo elemento in lista e a mettere in lista pronti quelli le cui richieste si trovano in testa alla lista e hanno il campo `d_attesa` uguale a 0.

```
1  //sistema.cpp
2
3  void inserimento_lista_attesa(richiesta* p){
4          richiesta* r = p_sospesi;
5          richiesta* precedente = 0;
6          bool ins = false;
7
8          while(r!=0 && !ins){
9                  if(p->d_attesa>r->d_attesa){
10                         p->d_attesa-=r->d_attesa;
11                         precedente = r;
12                         r = r->p_rich;
13                 }
14                 else ins = true;
15         }
16         p->p_rich = r;
17         if(precedente !=0) precedente ->p_rich = p;
18         else p_sospesi = p;
19         if(r!=0) r->d_attesa -=p->d_attesa;
20 }
```

6.12.2 Interruzione da timer

Ogni 50ms arriva un'interruzione dal timer che manda in esecuzione uno specifico driver. Questo driver:

- salva lo stato del processo che si è interrotto utilizzando il sottoprogramma `salva_stato` (il meccanismo di interruzione hardware è uguale a quello software utilizzato nelle primitive);
- pone il processo interrotto nella lista dei processi pronti utilizzando il sottoprogramma `inspronti()` (senza parametri);
- esegue le elaborazioni dovute sulla lista puntata da `p_sospesi` (decrementa il contenuto del campo `d_attesa` del primo elemento in lista e immette nella lista dei processi pronti quelli relativi alle richieste che si trovano in testa alla lista e hanno il campo `d_attesa` uguale a 0);
- richiama lo schedulatore;
- carica lo stato del nuovo processo.

Driver del timer

```
1 #sistema.s
2 .TEXT
3 .EXTERN c_driver_td
4 driver_td:      CALL    salva_stato
5                 CALL    c_driver_td
6                 CALL    inviaEOI
7                 CALL    carica_stato
8                 IRET
9
10 //sistema.cpp
11 extern ''C'' void c_driver_td(void){
12         richiesta* p;
13         if(p_sospesi != 0) p_sospesi -> d_attesa--;
14         while(p_sospesi != 0 && p_sospesi->d_attesa == 0){
15                 inserimento_lista(pronti, p_sospesi->pp);
16                 p = p_sospesi;
17                 p_sospesi = p_sospesi->p_rich;
18                 dealloca(p);
19         }
20         inspronti();
21         schedulatore();
22 }
```

Capitolo 7

Operazioni di I/O

7.1 Primitive di I/O

Nel codice dei processi non possono essere usate le routine predefinite di ingresso/uscita valide per un ambiente sequenziale perché questo tipo di routine non gestisce le operazioni in modo *atomico*. Infatti se nel mezzo di un'operazione di I/O arrivasse un'interruzione, un eventuale nuovo processo potrebbe voler effettuare la stessa operazione di I/O mentre il sistema interfaccia-dispositivi si trova in uno stato **inconsistente**.

In un sistema multiprogrammato occorre definire nuove routine di I/O.

Le operazioni di I/O possono essere di due tipi: *sincrone* o *asincrone* a seconda che il processo che le esegue attenda la loro terminazione prima di proseguire, oppure prosegua compiendo altre elaborazioni.

In un sistema multiprogrammato si utilizzano le operazioni di I/O sincrone: il processo che le esegue si blocca per divenire di nuovo pronto quando l'operazione è terminata.

Quindi le operazioni di I/O in un sistema multiprogrammato vengono realizzate con particolari primitive, dette di I/O, che non effettuano commutazione di contesto e, non lavorando sulle code dei processi, **possono essere eseguite con le interruzioni mascherabili abilitate** (a differenza delle normali routine di nucleo che invece girano con le interruzioni mascherabili disabilitate). Poiché le interruzioni sono abilitate, la mutua esclusione e la sincronizzazione sono gestite tramite l'uso di semafori. Tenere il più possibile le interruzioni abilitate costituisce un particolare vantaggio quando le operazioni di I/O avvengono a interruzione di programma.

Le primitive di I/O utilizzano le istruzioni di I/O ed hanno quindi il loro stesso privilegio.

Una primitiva di I/O (`prim_io_i`) corrisponde ad un *sottoprogramma di interfaccia* scritto in Assembly. Esso esegue un'interruzione software (`INT $io_tipo_i`) con la conseguente messa in esecuzione della primitiva vera e propria (`a_prim_io_i`). La primitiva vera e propria termina con una `IRET` restituendo il controllo al sottoprogramma di interfaccia il quale, a sua volta, restituisce il controllo al processo chiamante. Se una primitiva di I/O viene invocata da un processo utente abbiamo un innalzamento del livello di privilegio e un cambio di pila.

Le interruzioni software utilizzate per le primitive di I/O coinvolgono gate di tipo *trap* che **non** disabilitano le interruzioni mascherabili (non modificano il flag IF).

```
1 //modulo utente
2
3 //utente.cpp
4
5 extern ''C'' void prim_io_i(/*parametri formali*/);
6 //...
7 void proc_i(int h){ //codice di un processo
8         prim_io_i(/*parametri attuali*/);
9 }
10
11 #utente.s
12 .TEXT
13 .GLOBAL prim_io_i #sottoprogramma di interfaccia
14
15 prim_io_i:      INT $io_tipo_i #gate di tipo trap
16 rit_io_i:       RET
17
18 //modulo sistema
19
20 #sistema.s
21 .TEXT
22 .EXTERN c_prim_io_i
23 a_prim_io_i:    #routine INT $io_tipo_i
24                 #salvataggio dei registri
25                 #eventuale verifica e ricopiamento dei parametri
26                 CALL    c_primitiva_io_i
27                 #ripulitura della pila
28                 #ripristino dei registri
29                 IRET
30 //sistema.cpp
31 extern ''C'' void c_prim_io_i(/*parametri formali*/){
32         //...
33 }
```

Nel caso in cui la primitiva di I/O non sia void ma produca un risultato, il salvataggio e il rispristino dei registri non deve includere il registro `EAX`.

7.2 Svolgimento delle operazioni di I/O in un sistema multiprogrammato

Vediamo come si svolgono le operazioni di I/O nel nucleo. Il discorso è simile a quello che avevamo fatto in precedenza per le operazioni di I/O gestite a interruzione di programma solo che in questo caso parleremo di processi e non di programmi.

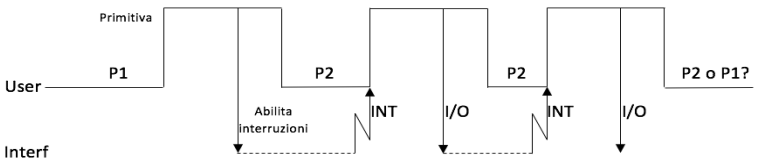


Figura 7.1: Situazione tipo

Ad un certo punto un processo richiede di fare un'operazione di I/O; per farlo dovrà invocare una primitiva. Se un processo è a livello utente, come P1 nell'esempio, non ha il diritto di accedere direttamente alle periferiche: se i registri delle periferiche si trovano nello spazio di I/O, questo processo non ha proprio il privilegio per eseguire le istruzioni `IN` e `OUT`. Possiamo impedire al processore, quando si trova in modo utente, di eseguire `IN` e `OUT` inserendo un'apposita configurazione in un campo del registro `EFLAG` che si chiama IOPL (*IO Privilege Level*): a questo punto se un processo utente tenta di eseguire la `IN` o la `OUT` viene lanciata un'eccezione di protezione. Il fatto che IOPL sia in mezzo agli altri flag del registro creerebbe problemi perché se eseguissimo la `POPF` con una configurazione che riporta IOPL alla configurazione in cui permette l'esecuzione di `IN` e `OUT` a livello utente, invalideremmo quanto detto fin ora. Per fortuna la `POPF` tiene conto di questa cosa e va a modificare solo i flag che non sono privilegiati lasciando inalterati gli altri fra cui IOPL e IF. Questo solo se siamo a livello utente, a livello sistema si può modificare tutto. Quindi se il processo utente vuole eseguire un'operazione di I/O può soltanto chiedere al sistema di farla per lui. Cioè può solo saltare alla primitiva fornita dal modulo sistema, primitiva che sarà privilegiata e potrà eseguire l'operazione richiesta. Se invece le periferiche hanno i loro registri in memoria, come facciamo ad impedire l'accesso a questi indirizzi da parte di un processo utente? Si possono o rendere non raggiungibili oppure raggiungibili ma solo a livello sistema (bit U/S del descrittore di pagina virtuale).

L'interfaccia andrà programmata per svolgere le operazioni con le interruzioni abilitate. L'interfaccia, ad un certo punto, avrà i suoi dati pronti e invierà un interrupt ogni volta che avrà un nuovo dato pronto. L'interrupt interrompe quello che c'è a livello utente e si porta a livello sistema dove va in esecuzione un *driver*. Il driver esegue l'operazione di I/O vera e propria e poi deve sapere quando l'operazione è finita. Quando l'operazione è terminata, P1 potrà tornare in esecuzione e troverà pronti i dati che aveva richiesto. Per tutto lo svolgimento dell'operazione, il processo P1 viene sospeso e nel frattempo succederà qualcos'altro: il processore non si ferma mai. Con il nucleo multiprogrammato, possiamo passare ad eseguire un altro processo P2. Ed è P2 che la periferica interrompe. Quando però l'operazione termina, come lo facciamo sapere al processo che ne aveva fatto richiesta? Usiamo un semaforo di sincronizzazione. Occorre analizzare le due tipologie di semaforo che andremo ad utilizzare: *mutua esclusione* e *sincronizzazione*. Hanno due scopi e modi di utilizzo totalmente diversi fra loro.

Mutua Esclusione

inizializzato a 1

P1	P2
`sem_wait(sem)`	`sem_wait(sem)`
`<sezione critica>`	`<sezione critica>`
`sem_signal(sem)`	`sem_signal(sem)`

scopo: evitare che due processi accedano ad una stessa risorsa contemporaneamente; proteggere la risorsa.

Sincronizzazione

inizializzato a 0

P1	P2
`sem_wait(sem)`	`sem_signal(sem)`

scopo: serve a fare in modo che P1 attenda che accada qualcosa. Questo qualcosa lo farà accadere P2 e quando succede lo segnala facendo sbloccare P1. Oppure se P2 ha già eseguito la `sem_signal()`, P1 non si blocca.

Nel caso che stiamo osservando necessitiamo di un semaforo di sincronizzazione perché P1 attende che il dato che gli serve sia pronto. P1 farà `sem_wait` sul semaforo `sync`; il driver, quando sa di aver terminato, farà `sem_signal` su quello stesso semaforo per notificare che la cosa è avvenuta; che cosa è avvenuto è codificato nel semaforo che è stato scelto; ogni semaforo ha il suo significato.

Ora abbiamo un altro problema: abbiamo due processi P1 e P2. P1, mentre P2 è in esecuzione, sta aspettando dei dati dall'interfaccia. Teniamo presente che sono due processi *slegati*. Cosa può accadere? P2, mentre P1 sta aspettando i risultati dall'interfaccia, potrebbe richiedere l'uso della stessa interfaccia. Per evitare che questa cosa accada, occorre utilizzare anche un semaforo di mutua esclusione. Finché l'interfaccia sta facendo qualcosa per conto di un processo, nessun altro la può usare. **Tutta l'operazione deve, quindi, avvenire in mutua esclusione**.

7.2.1 Operazione di lettura

Al driver occorre sapere dove si trova il buffer su cui deve lavorare e da quanti byte è composto. Queste due cose deve sceglierle l'utente. La primitiva, ad esempio per la lettura, sarà qualcosa del tipo:

```
1 extern "C" void read_n(natl interf, char* buff, natl n_byte){...}
2 //interf: indica qual e' la periferica da cui leggere
3 //buff: puntatore al buffer
4 //n_byte: quanti byte vanno letti
```

Le informazioni che si passano come parametri si trovano in pila, ma poi come le facciamo sapere al driver? Queste informazioni si trovano in **variabili globali**. Questo perché il byte che il driver legge lo deve copiare nel buffer predisposto dall'utente e quindi *il driver deve sapere dove si trova il buffer su cui operare.*

Supponendo che le periferiche siano tutte uguali, possiamo avere un array di *descrittori di operazioni I/O* indicizzato dal numero della periferica stessa. Ogni descrittore è fatto in questo modo:

```
1  struct interf_reg{
2          union{ioaddr iRBR, iTBR;} in_out;
3          union{ioaddr iCTRI, iCTRO;} ctr_io;
4  };
5
6  struct des_io{
7          natb* buff; //puntatore al buffer
8          natl n; //numero di byte coinvolti nell'operazione
9          natl mutex; //semaforo di mutua esclusione
10         natl sync; //semaforo di sincronizzazione
11         interf_reg indreg; //indirizzi dei registri della periferica
12 };
```

Vediamo più nel dettaglio come si svolge questa operazione:

```
1  //LIVELLO UTENTE
2  //utente.cpp
3
4  const natl N = ...; //numero di byte del buffer
5  natl interf = ...; //numero d'ordine dell'interfaccia
6
7  natb buff[N];
8  void corpo(){
9          //..
10         read_n(interf, buff, N);
11         //..
```

```
12 }
13
14 //utente.s
15 read_n:
16         INT $tipo_read_n //tramite la IDT, all'indice tipo_read_n,
               saltiamo alla a_read_n che si trova a livello sistema
17         RET
18
19 //LIVELLO SISTEMA
20 //sistema.s
21 a_read_n:
22         #salvataggio dei registri
23         #copia parametri //i parametri sono in pila utente ma nella
               pila sistema abbiamo il puntatore alla pila utente
24         CALL c_read_n
25         #ripulitura pila
26         #ripristino registri
27         IRET
28
29 //sistema.cpp
30 extern "C" void c_read_n(natl interf, char* buff, natl n_byte){
31         des_io* p_desi = &array_desio[interf]; //puntatore all'
               elemento dell'array corrispondente all'interfaccia che
               stiamo utilizzando
32         sem_wait(p_desi->mutex);
33         start_in(p_desi, buff, n_byte);
34         sem_wait(p_desi->sync); //dobbiamo aspettare che l'operazione
               sia conclusa. Durante l'attesa non viene rilasciata la
               mutua esclusione.
35         sem_signal(p_desi->mutex);
36 }
37 //...
38 void start_in(des_io* p_desi, natb buff[], natl n_byte){
39         p_desi->n = n_byte;
40         p_desi->buff = buff;
41         go_input(p_desi->indreg.ctr_io.iCTRI);
42 }
43 //...
```

La `start_in()` deve ricopiare `buff` ed `n_byte` nei rispettivi campi di `p_desi` (ed è da qui che leggerà il driver) e poi deve scrivere negli opportuni registri della periferica (i cui indirizzi sono nel descrittore, campo `indreg`) che la periferica è abilitata a generare interruzioni e anche tutte le altre cose che la periferica deve sapere per poter fare questa operazione.

Nelle primitive che abbiamo visto fino ad ora potevamo avere una `CALL salva_stato` e una `CALL carica_stato`. Possiamo salvare e caricare lo stato anche in `a_read_n`? No, perché ci pensano già la `sem_wait` e la `sem_signal`. Se lo facessimo anche nella `a_read_n`, sovrascriveremmo quanto salvato dalle altre. Se vogliamo usare queste primitive di I/O, dobbiamo immaginare di essere ad un livello più alto in cui non dobbiamo preoccuparci di salvare e caricare lo stato: ci appoggiamo alla `sem_wait` e alla `sem_signal` e sono loro che se ne occupano.

Le primitive viste fino ad ora devono girare con le interruzioni mascherabili disabilitate perché devono eseguire operazioni direttamente sulle liste dei processi. In questo caso, invece, non c'è bisogno di disabilitare le interruzioni poiché le liste dei processi non vengono manipolate direttamente; la `sem_wait` e la `sem_signal` ci pensano da sole a proteggersi disabilitando le interruzioni. Quindi il gate per l'interruzione `tipo_read_n` sarà di tipo *trap*; questo tipo di gate **non** disabilita le

interruzioni.

7.3 Driver

Vediamo cosa deve fare il driver ogni volta che va in esecuzione. Anche il driver avrà una parte scritta in Assembly perché ci serve la `IRET`.

```
1  #sistema.s
2  .TEXT
3  .EXTERN         c_driverin_i
4  int_tipoi_i:
5          CALL salva_stato         #coinvolto gate di tipo interrupt
6          CALL c_driverin_i
7          CALL inviaEOI
8          CALL carica_stato
9          IRET
10
11 //sistema.cpp
12 extern ''C'' void c_driverin_i(){
13         const int INTERF = ...; //numero dell'interfaccia
14         des_io* p_desi = &array_desio[INTERF]; //puntatore al
               descrittore dell'operazione di I/O con indice il numero
               dell'interfaccia
15         des_sem* sem;
16         proc_elem* work;
17         natb b;
18         p_desi->n--; //si decrementa il numero dei byte
19         if(p_desi->n == 0){ //se era l'ultimo byte
20                 halt_io(p_desi->indreg.ctr_io.iCTRI); //disabilitiamo
                       la periferica a inviare interruzioni
21                 //sem_signal(p_desi->sync) non possiamo farlo  perche
                       ', non essendoci stata commutazione di contesto,
                       sovrascriverebbe lo stato di P2
22                 sem = &array_dess[p_desi->sync];
23                 sem->counter++;
24                 if(sem->counter <= 0){
25                         rimozione_lista(sem->pointer, lavoro);
26                         inserimento_lista(pronti, lavoro);
27                         inserimento_lista(pronti, esecuzione); //
                               preemption
28                         schedulatore(); //preemption
29                 }
30         }
31         inputb(p_desi->indreg.in_out.iRBR, b);
32         *static_cast<natb*>(p_desi->buff) = b;
33         p_desi->buff = static_cast<natb*>(p_desi->buff) + 1;
34 }
```

Questo driver è il driver di *quella* periferica e deve accedere al descrittore di *quella* periferica. Sa qual è l'elemento dell'array perché il numero della periferica, le cui interruzioni lo fanno andare in esecuzione, è dichiarato costante all'interno del driver stesso.

Il driver, ogni volta che invia EOI vuol dire che ha un byte pronto; il suo scopo è prendere questo byte, copiarlo nel punto giusto del buffer che l'utente aveva predisposto e poi prepararsi per il prossimo interrupt oppure, se è l'ultimo byte, segnalare che l'operazione si è conclusa.

Il driver **non è un processo**; ruba temporaneamente le risorse al processo che viene interrotto, in questo caso P2. Il driver sta usando la pila di P2. Non possiamo

salvare lo stato del driver poiché il driver non essendo un processo, non ha un proprio descrittore di processo.

In `int_tipo_i` va soltanto salvato e caricato lo stato perché prevediamo che si possa tornare non a P2 ma a P1 nel caso in cui l'operazione effettuata dal driver sia l'ultima.

Quando viene effettuata l'ultima operazione abbiamo due processi pronti: P1 e P2. Quali dei due deve andare in esecuzione? **Quello a maggior priorità fra i due**.

Cosa dobbiamo fare quindi dopo la `halt_io()` per risvegliare il processo se la `sem_signal()` non la possiamo fare? Dobbiamo fare "a mano" quello che fa la `sem_signal()`. Occorre togliere da `p_sospesi` P1, mettendolo in coda pronti (ci va messo anche P2), e a quel punto lo schedulatore sceglie quello a maggiore priorità fra i due. Quindi "ricopiamo" quello che fa la `sem_signal()` meno il salvataggio/caricamento dello stato.

Occorre fare attenzione a non mettere uno stesso processo in più code (per esempio mettere P1 in coda pronti senza prima averlo tolto dalla coda del semaforo).

Perché occorre controllare se siamo arrivati alla fine *prima* di svolgere l'operazione? Leggere da `RBR` nella periferica ha un significato per la periferica stessa: l'interruzione è stata servita, la periferica è soddisfatta. Se è l'ultimo dato da mandare, non vogliamo altri interrupt. Se leggessimo l'ultimo dato prima di aver disabilitato le interruzioni con `halt_io()`, non appena leggiamo il dato, la periferica potrebbe inviare un'altra interruzione e accadrebbe che il byte letto venga copiato fuori dal buffer predisposto dall'utente.

Poiché il driver manipola le code dei processi ***non*** può girare con le interruzioni abilitate.

Il driver "ruba" temporaneamente le risorse al processo attualmente in esecuzione; fra queste risorse vi è anche la memoria virtuale. Le pagine che contengono il buffer predisposto dall'utente, possono essere soggette a rimpiazzamento? In generale non è possibile tollerare page-fault durante l'esecuzione del driver che, ricordiamo, deve essere eseguito in modo atomico.

7.4 Processi *esterni*

La soluzione vista per la gestione dell' I/O con i driver ha vari svantaggi derivanti tutti dal fatto che il driver non è un processo. Il driver non è un processo, non può liberamente usare primitive di nucleo come la `sem_signal` e, poiché manipola direttamente le code dei processi, deve girare con le interruzioni mascherabili disabilitate.

Tenere a lungo le interruzioni disabilitate equivale a dire che la cosa che stiamo facendo è la più importante di tutte però questo non è automaticamente vero: ci potrebbero essere interruzioni più importanti di quello che stiamo facendo. In generale, tenendo a lungo le interruzioni disabilitate aumenta il tempo di risposta alle richieste di interruzione.

Se le interruzioni fossero gestite da un processo, anziché da un driver, sarebbe meglio perché un processo ha una priorità e soprattutto ha un descrittore e quindi potrebbe essere interrotto a sua volta e fatto ripartire in qualunque momento.

L'idea è che quando arriva un'interruzione da una periferica scheduliamo un processo che dovrà gestirla.

Questi processi li distinguiamo dai normali processi utente e li chiamiamo ***processi esterni***.

I processi esterni sono dunque processi che gestiscono le interruzioni e **hanno lo stesso privilegio delle istruzioni di I/O**. Quindi per ogni sorgente di interruzione, abbiamo un processo esterno che la dovrà gestire. Il processo esterno relativo a una certa interruzione, non è di norma pronto, diverrà pronto quando arriva un'interruzione di quel tipo. Predisponiamo una tabella indicizzata dal tipo delle interruzioni e per ognuna di esse abbiamo un processo esterno che la gestisce.

Quando arriva un'interruzione, per come è fatta la gestione delle interruzioni della CPU, siamo costretti almeno per un brevissimo momento a fare qualcosa che somigli a un driver, che non sia dunque un processo e che usi temporaneamente le risorse del processo attualmente in esecuzione. Questo è però ridotto al minimo perché l'unica cosa che facciamo quando arriva l'interruzione è prendere il corrispondente processo esterno, renderlo pronto e poi terminare. Il processo esterno potremmo anche metterlo direttamente in esecuzione dal momento che **i processi esterni hanno priorità maggiore di tutti gli altri processi**.

In risposta ad un'interruzione esterna va in esecuzione una routine che chiamiamo `handler_i`. Abbiamo un handler per ogni possibile interruzione esterna.

`handler_i`:

- salva lo stato del processo interrotto (`CALL salva_stato`);
- mette in coda pronti il processo interrotto (va in testa perché se era in esecuzione vuol dire che era quello a maggior priorità);
- mette in coda esecuzione (a singolo `proc_elem`) il processo esterno i-esimo;
- carica lo stato del processo esterno (la `carica_stato` carica lo stato del processo che è in coda esecuzione in quel momento);
- `IRET`: una volta caricato lo stato del processo esterno, agisce sulla pila del processo esterno; quindi `IRET`, caricando `EIP`, salta sostanzialmente al processo esterno. Questa `IRET` agisce sulla pila sistema del processo esterno stesso.

Il codice Assembly di un `handler_i` sarà dunque il seguente:

```
1  #sistema.s
2  .DATA
3  a_p:    .SPACE  P*4 (P = numero di piedini dell'APIC)
4  .TEXT
5  .EXTERN inspronti
6  handler_i:      #vi sono P handler
7          CALL    salva_stato
8          CALL    inspronti       #inserisce in lista pronti il processo
               attualmente in esecuzione
9          MOVL    $i, %ecx
10         MOVL    a_p(,%ecx,4), %eax
11         MOVL    %eax, esecuzione        #mette in esecuzione il
               contenuto del puntatore a_p[i] associato all'handler i-
               esimo
12         CALL    carica_stato
13         IRET
```

Questa cosa che somiglia a un driver dovrà girare con le interruzioni mascherabili disabilitate. Le interruzioni vengono riabilitate con la messa in esecuzione del processo esterno.

Tutti i processi esterni hanno la stessa struttura di base:

```
1 void extern_proc(int irq_i){
2         des_io* pdes = &array_desio[irq_i];
3         for(;;){
4                 //...
5                 wfi();
6         }
7 }
```

Tutti i processi esterni li organizziamo come un ciclo infinito in fondo a ogni iterazione del quale viene invocata una primitiva, `wfi()` (*Wait For Interrupt*).

`wfi()`:

- salva lo stato del processo esterno
- invia EOI
- chiama lo schedulatore
- carica lo stato
- `IRET`

I processi esterni sono organizzati con un ciclo infinito poiché, dopo la chiamata alla `wfi()` e il conseguente salvataggio del loro stato, quando tornano in esecuzione a seguito di un'altra interruzione ripartono dall'istruzione successiva alla `wfi()` (perché `EIP` era arrivato lì) e cioè riparte il ciclo.

Per creare questi processi esterni assumiamo di avere a disposizione un'altra primitiva diversa dall'`activate_p` che chiamiamo `activate_pe` ed ha la seguente intestazione:

```
1 natl activate_pe(void f(int),int a,natl prio,natl liv,natb IRQ);
```

Tale primitiva ha gli stessi parametri dell'`activate_p` ed in più ha il parametro `IRQ` con cui diciamo al processo esterno che stiamo creando, a quale interrupt dovrà rispondere. A differenza della `activate_p`, questa primitiva inserisce il processo esterno creato in un'apposita lista (a singolo elemento) dei processi bloccati. Esiste una lista dei processi bloccati per ogni possibile interruzione che può arrivare all'APIC. Abbiamo quindi 24 liste ognuna delle quali è puntata da un elemento dell'array `a_p[P]` dove P è il numero dei piedini dell'APIC (in questo caso 24). La `activate_pe` predispone la pila sistema del processo esterno in modo tale che l'istruzione `IRET` dell'handler faccia partire il processo esterno dall'inizio del relativo codice. La pila del processo esterno viene inizializzata nuovamente alla fine di ogni iterazione dalla primitiva `wfi()`.

Il processo esterno, essendo un processo a differenza dei driver, può girare con le interruzioni mascherabili abilitate. Grazie a questa cosa possiamo gestire le *interruzioni annidiate*. Nel ciclo for il processo esterno dovrà sostanzialmente fare quello che faceva il driver.

Non c'è bisogno che `void extern_proc(int i)` faccia parte del modulo sistema poiché non deve usare simboli definiti in quest'ultimo. Invece il driver poiché gestiva direttamente le code dei processi, code che sono definite nel modulo sistema, occorreva collegarlo al modulo sistema stesso.

`void extern_proc(int i)` può invece usare direttamente le primitive come `sem_signal()`.

La gestione dell'I/O può quindi essere messa in un modulo separato; abbiamo tre moduli:

1. SISTEMA
2. I/O (vi si trova `void extern_proc(int i)`)
3. UTENTE

Questi tre moduli *non* sono collegati fra di loro. Comunicano soltanto attraverso le interruzioni.

La `handler_i` deve far parte del modulo sistema poiché usa simboli definiti in quest'ultimo.

A quale livello di privilegio facciamo girare il codice che esegue questi processi esterni? I processi esterni hanno lo stesso privilegio delle istruzioni di I/O; quindi dipende dalla configurazione del campo IOPL del registro `EFLAG`. In ogni caso noi prevediamo che le istruzioni di I/O e dunque anche i processi esterni, abbiano livello sistema.

Vediamo la `wfi()` nel dettaglio:

```
1 #io.s
2 wfi:    INT $tipo_wfi
3         RET
4
5 #sistema.s
6 a_wfi:  CALL    salva_stato
7         CALL    inviaEOI
8         CALL    schedulatore
9         CALL    carica_stato
10        IRET
```

La `a_wfi` deve stare nel modulo sistema poiché chiama `salva_stato`, schedulatore e `carica_stato` che sono simboli definiti nel modulo sistema.

Può accadere di avere processi esterni in lista pronti? Sì. Se analizziamo il caso in cui arriva un'interruzione a più alta priorità di quella che si sta servendo attualmente, accadono le seguenti cose:

- l'handler dell'interruzione appena arrivata mette in lista pronti il processo esterno attualmente in esecuzione (`inspronti`);
- schedula il processo esterno destinato a servire quella interruzione;
- una volta che il processo esterno esegue la `wfi()` viene ripreso il processo esterno precedentemente messo in lista pronti.

Abbiamo la certezza di non mettere inavvertitamente in esecuzione processi esterni quando non richiesto perché è solo l'handler che inserisce in pronti il processo attualmente in esecuzione. La `wfi()` che rappresenta il modo con cui il processo esterno manda EOI, non mette in lista pronti il processo ma si limita a salvarne lo stato qualora il byte appena trasferito non fosse stato l'ultimo. La `wfi()` chiama lo schedulatore che sovrascrive letteralmente ciò che c'è in esecuzione prelevando il primo elemento della lista pronti (che poi può essere o meno un processo esterno).

7.4.1 Operazione di lettura con processi esterni

Vediamo come diventa la primitiva per leggere dei byte con l'uso dei processi esterni anziché dei driver.

`read_n` ⟶ `a_read_n` ⟶ `c_read_n`

```
1 void extern_proc(int irq_i){ //irq_i e' il numero del piedino
2         des_io* p_desi = &array_desio[irq_i];
3         natb b;
4         for(;;){
5                 p_desi->n--;
6                 if(p_desi->n == 0){ //[1]
7                         halt_io(p_desi->indreg.ctr_io.iCTRI);
8                 }
9                 inputb(p_desi->indreg.in_out.iRBR, b); //prelievo
10                *p_desi->buff = b; //memorizzazione
11                p_desi->buff++;
12                if(p_desi->n == 0){
13                        sem_signal(p_desi->sync); //[2]
14                }
15                wfi();
16        }
17 }
18
19 //[1] La halt_io va fatta qui a differenza del driver, perche' se e' l
       'ultima operazione e non disabilitiamo subito le interruzioni, si
       potrebbe andare a scrivere fuori dal buffer.
20 //[2] La sem_signal() la facciamo qui e non subito dopo la halt_io()
       perche' altrimenti l'ultimo byte non verrebbe copiato. Non
       possiamo segnalare che l'operazione e' finita se non abbiamo
       ancora trasferito l'ultimo byte.
```

7.5 Nota su salvataggio/caricamento stato e IRET

La IRET è un'istruzione che preleva tre o cinque parole lunghe dalla cima della pila e le interpreta come EIP, CPL, EFLAG. Queste stesse informazioni, nell'ordine in cui le legge la IRET, vengono inserite in pila dalla INT, dalle eccezioni o dalle interruzioni esterne. Quindi la IRET disfarà, per esempio, quello che ha fatto una INT. Quando avviene un cambio di contesto, chi ce lo dice che in cima alla pila sistema di questo nuovo processo ci sono informazioni che la IRET è in grado di interpretare? Tutti i processi che non sono in esecuzione, sono cioè bloccati, si sono fermati per effetto di una interruzione e quindi nella loro pila sistema ci sono tutte informazioni che la IRET è in grado di interpretare correttamente. Le pile dei processi si trovano quindi tutte in uno stato *compatibile* con la IRET.

E i processi che non sono mai andati in esecuzione come hanno la pila? Anche i processi appena creati hanno la pila compatibile con la IRET perché l'`activate_p()` / `activate_pe()`, crea la pila in quello stato. Questo lo facciamo noi che scriviamo il software.

7.5.1 Creazione di un processo

Utente:

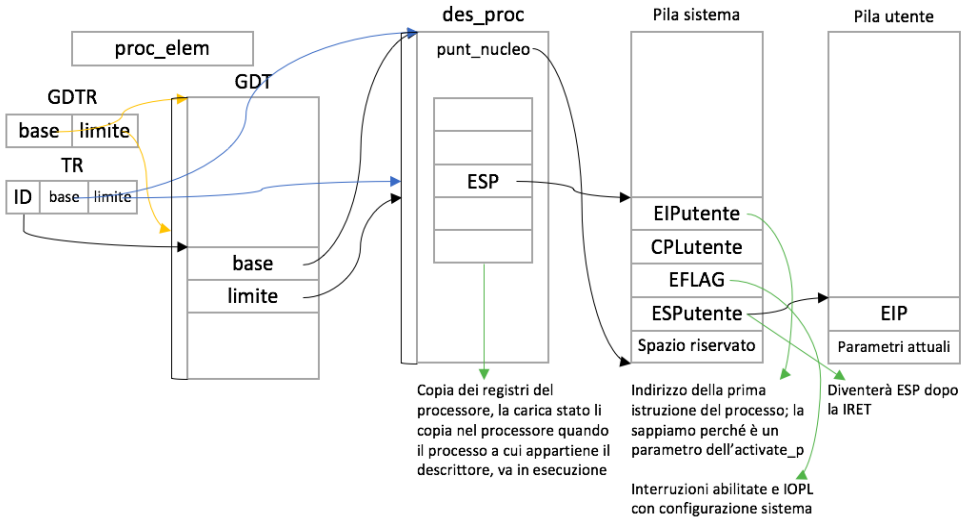


Figura 7.2: Pile e descrittore processo utente

La GDT non è infinita e se vogliamo avere un descrittore per ogni processo, avremo un numero limitato di processi (circa 8000) e come ID, per semplicità, possiamo scegliere l'indice numerico all'interno della GDT.

Se quindi vogliamo immaginare cosa succede dopo una `INT`, in pila avremo un po' di entità (5 parole lunghe perché stiamo considerando un processo utente).

Sapere dove si trova il descrittore di processo, serve al microprogramma di interruzione; ecco perché il puntatore al descrittore di processo si trova nella GDT.

Trattandosi di un processo utente, abbiamo due pile una utente e una sistema e si passa dall'una all'altra per mezzo del microprogramma di interruzione. Questo avviene quando stiamo passando a un livello di privilegio più elevato o quando stiamo tornando al livello originario. La nuova pila, quella sistema, viene trovata per mezzo di un puntatore nel descrittore di processo (`punt_nucleo`). Il microprogramma sa a quale descrittore accedere mediante il contenuto del registro `TR`.

Cosa c'è in pila sistema mentre ci troviamo a livello utente? Nulla.

Sistema:

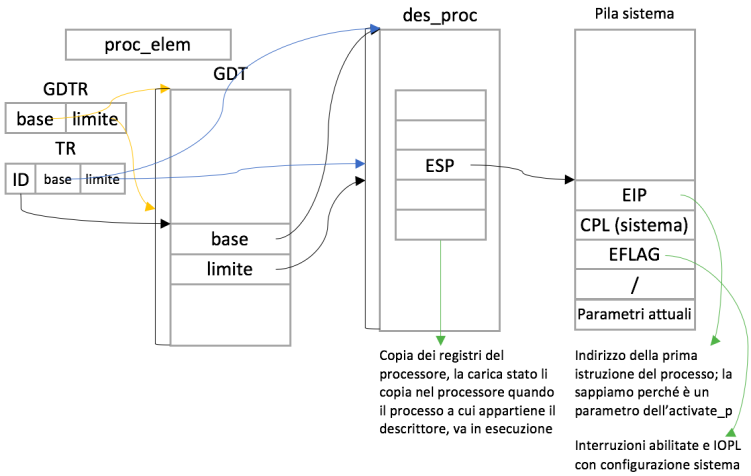


Figura 7.3: Pila e descrittore processo sistema

Poiché ci troviamo già a livello sistema, non essendo quindi necessario nessun cambiamento di pila, il campo `punt_nucleo` del descrittore di processo diventa non significativo. Inoltre, ovviamente, non esiste la pila utente per i processi che si trovano a livello sistema.

Capitolo 8

Bus PCI, Bus Mastering e DMA

8.1 Introduzione allo standard PCI

Lo standard PCI (Periferical Component Interconnect) cerca di risolvere i problemi legati al caos con cui le varie periferiche dell'architettura PC-compatibile erano state sviluppate. Che problemi ci sono nell'architettura come l'abbiamo vista fino ad ora? La periferica aveva, nello stampato, una maschera che stabiliva in quale punto dello spazio di I/O dovevano trovarsi i suoi registri. Oltre ad avere possibilità di configurazione limitata (su dove mettere i registri, al massimo 2 o 3), poteva accadere che più schede avessero i registri implementati nello stesso spazio di I/O creando conflitti e rendendo impossibile il montaggio simultaneo delle due. Inoltre, fino ad ora, il software non aveva un modo preciso di sapere se la scheda fosse presente o meno. Con lo standard PCI le schede non vengono prodotte con preimpostato l'indirizzo a cui si devono trovare i suoi registri; questa cosa è **programmabile**. Le schede sono dunque pronte ad avere i registri ad un indirizzo qualunque. E inoltre c'è un modo sicuro con cui il software può sapere se una scheda è presente o meno. Dal punto di vista architetturale, il bus PCI è un bus che non ha nulla a che vedere con il bus di cui abbiamo parlato fino ad adesso; sono due cosa distinte. Il bus PCI è anche indipendente dalla CPU ed è collegato al bus (a cui è collegata la CPU e a cui un tempo era collegata anche la memoria fisica), tramite un oggetto hardware che si chiama ***ponte host-PCI***. Vi possono essere più bus PCI collegati fra loro attraverso altri oggetti hardware chiamati *ponti PCI-PCI*.

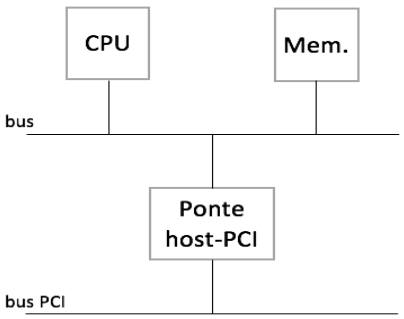


Figura 8.1: Una prima visione d'insieme

Ci possono essere ponti ulteriori verso altri tipi di bus (come USB). Per semplicità ci limiteremo solo a un bus PCI.

Tutti i bus PCI sono numerati; il primo, quello più vicino alla CPU, è il numero 0. I bus PCI possono essere al massimo 256 (8 bit per identificare ciascun bus). Su ogni bus possono essere montati fino a 32 dispositivi. Su ogni dispositivo ci possono essere fino a 8 funzioni (una scheda può fare più cose). Ogni funzione ha, in questo modo, una sorta di identificativo per il solo fatto di trovarsi in un certo punto. Quindi sapendo numero del bus, numero del dispositivo e numero della funzione, è possibile identificare la funzione stessa.

Lo standard definisce tre spazi di indirizzamento per tutto il bus: memoria, ingresso/uscita e ***configurazione***. Quest'ultimo è lo spazio di indirizzamento che permette di risolvere i problemi che abbiamo visto.

Ogni funzione, di un certo dispositivo, possiede un proprio spazio di configurazione privato costituito da 256 registri. Di questi 256 registri, lo standard definisce solo i primi, poi i restanti sono a disposizione del produttore.

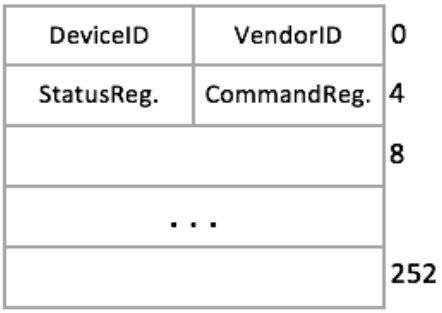


Figura 8.2: Alcuni dei registri di una funzione definiti dallo standard PCI

8.2 Operazioni sul bus PCI: *transazioni*

Le operazioni che si effettuano sul bus PCI vengono chiamate ***transazioni***. Ogni transazione ha un **iniziatore** e un **obiettivo**.

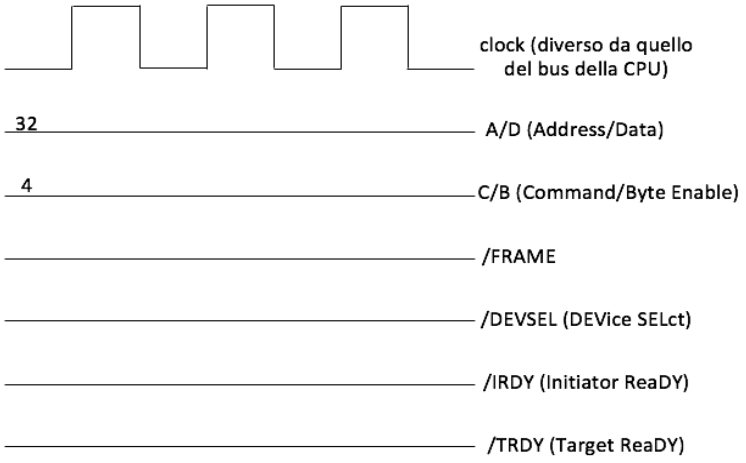


Figura 8.3: Principali linee del bus PCI

Ogni transazione è divisibile in fasi:

1. fase di ***indirizzamento***: l'iniziatore (che da inizio alla transazione) pone sul bus l'indirizzo, che può essere di memoria, di I/O o di configurazione, del dispositivo. L'iniziatore porta `/FRAME` a 0 e pilota `A/D` e `C/B`. Questa fase si conclude con l'obiettivo (che è un qualunque dispositivo) che riconoscendo l'indirizzo, porta `/DEVSEL` a 0. Se entro un certo periodo di tempo l'obiettivo non risponde si assume che l'indirizzo non sia relativo a nessun dispositivo e la transazione termina con un errore. Questo tempo si misura sul clock del bus PCI che è **diverso** da quello della CPU.

2. fase/i di ***dati***: l'iniziatore mantiene `/FRAME` a 0; utilizza BE come avviene per la memoria e sulla linea D passano i dati. Il fatto che ci siano più fasi dati indica che ogni scheda che segue lo standard PCI deve essere in grado di autoincrementare l'indirizzo finché l'iniziatore non stabilisce che l'operazione è terminata.

3. ultima fase dati: diversa dalle precedenti poiché l'iniziatore disattiva `/FRAME` portandolo a 1.

Se la fase è di scrittura, è l'iniziatore che pilota la linea dati; se è di lettura è l'obiettivo che, dalla seconda fase in poi, pilota la linea dati.

Un'altra cosa che lo standard permette è che ogni dispositivo può ricoprire il ruolo di iniziatore, il ruolo di obiettivo oppure entrambe le cose.

Nel caso più semplice il dispositivo iniziatore è il ponte host-PCI che trasforma opportunamente un ciclo di lettura o di scrittura dalla CPU.

Altri dispositivi possono decidere di essere iniziatori e fare un ciclo di scrittura o lettura per esempio nella memoria. Questo permette di trasferire dei dati in memoria quando sono pronti senza che la CPU debba intervenire (*bus mastering*); questa rappresenta la terza modalità di ingresso/uscita (controllo di programma, interruzione di programma, accesso diretto alla memoria).

Come fanno più iniziatori a condividere lo stesso bus? Non possono farlo contemporaneamente; la soluzione adottata dallo standard PCI sta in un dispositivo chiamato **arbitro** a cui gli iniziatori devono rivolgersi prima di iniziare una nuova transazione.

Tutti i dispositivi che sono iniziatori devono avere due piedini aggiuntivi `/REQ` (*REQuest*) e `/GNT` (*GraNT*) collegati ad un arbitro. C'è un arbitro per ogni bus PCI. Sul bus PCI 0 tipicamente l'arbitro è il ponte host-PCI stesso.

Quando l'iniziatore decide di iniziare una nuova transazione, mette `/REQ` a 0, l'arbitro risponderà dopo un certo tempo mettendo `/GNT` a 0. L'iniziatore aspetta che la linea `/FRAME` sia a 1 e poi può iniziare la nuova transazione.

Quando la CPU fa un'operazione in memoria, in qualche modo il ponte host-PCI può capire se questa operazione è diretta alla memoria oppure se verso un dispositivo collegato al bus PCI. Se è diretta verso uno di questi dispositivi, il ponte deve fare in modo che venga completata trasformando l'operazione in una transazione in memoria su bus PCI. Analogamente accade per le operazioni di I/O.

Cosa succede per le operazioni che coinvolgono lo spazio di configurazione? Come fa il software a generare transazioni di quel tipo? Per fare questo il ponte host-PCI ha un paio di registri quindi è esso stesso una periferica (di quelle del tipo vecchio). Questi registri si trovano nello spazio di I/O. Il software scrivendo in questi registri

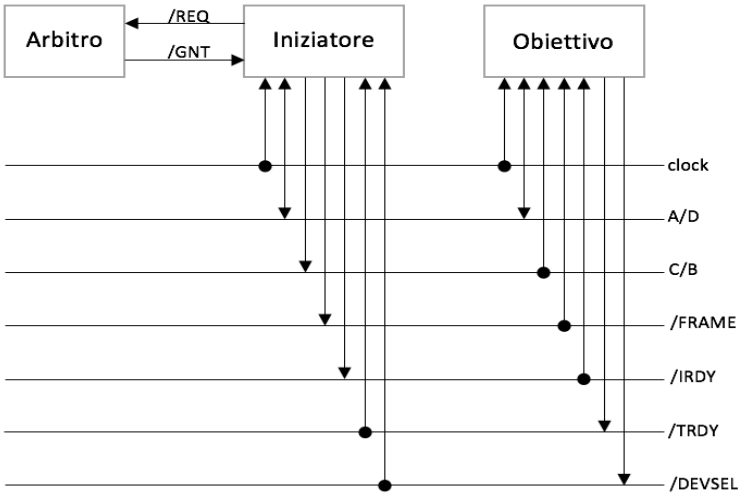


Figura 8.4: Collegamento al bus PCI di Iniziatore, Obiettivo e Arbitro

può ordinare al ponte stesso di generare transazioni di configurazione. Quindi solo un dispositivo PCI è in grado di effettuare operazioni nello spazio di configurazione; la CPU è in grado di fare soltanto operazioni in memoria e nello spazio di I/O.

8.3 Spazio di configurazione delle funzioni PCI

Ciascuna funzione di un dispositivo che segue lo standard, deve implementare, nello spazio di configurazione, dei registri definiti nello standard stesso.

Tra i registri obbligatori vi sono i seguenti:

- **vendorID**: lettura, 2 byte offset 0: id del produttore (per esempio 8086 è l'ID dell'INTEL e 04B3 è l'ID della IBM);
- **deviceID**: lettura, 2 byte, offset 2: id del dispositivo;
- **CommandRegister**: lettura/scrittura, 2 byte, offset 4;
  - il bit 0 se settato abilita la funzione a rispondere a transazioni di I/O;
  - il bit 1 se settato abilita la funzione a rispondere a transazioni di memoria;
  - il bit 2 se settato abilita la funzionalità di *bus mastering*.
- **StatusRegister**: lettura, 2 byte, offset 6: segnala la presenza o l'assenza di alcune funzionalità opzionali e informazioni relative agli errori rilevati durante le transazioni;
- **ClassCode**: lettura, 3 byte, offset 9: specifica il tipo di funzione svolta in base ad una codifica definita dallo standard PCI;
- **InterruptPin**: lettura, 1 byte, offset 57: contiene il codice di uno dei 4 piedini tramite cui la funzione può generare richieste di interruzione (0⟶nessun collegamento; 1⟶`/INTA`; 2⟶`/INTB`; 3⟶`/INTC`; 4⟶`/INTD`).

Ogni dispositivo ha al massimo 4 piedini attraverso i quali può mandare richieste di interruzione. Inoltre più dispositivi appartenenti allo stesso bus PCI, condividono le linee di interruzione.

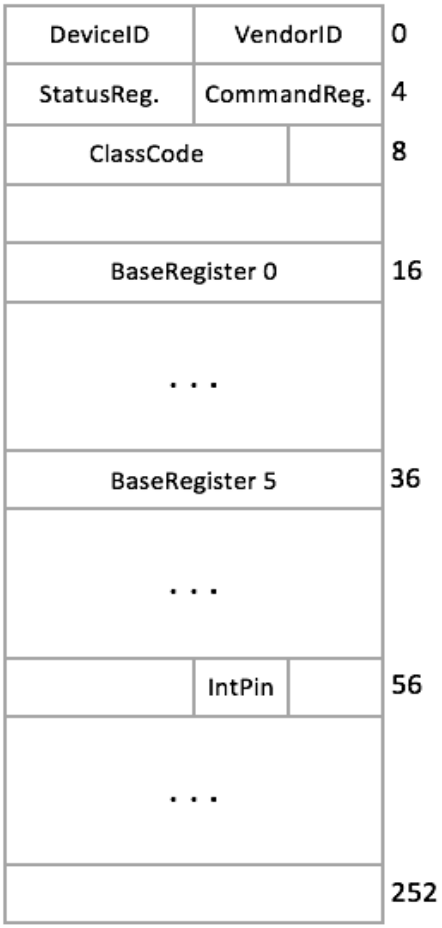


Figura 8.5: Principali registri obbligatori per ogni funzione

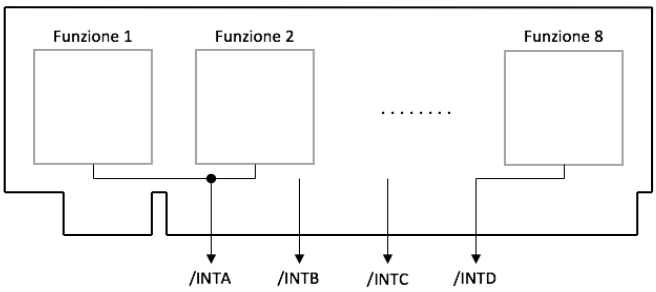


Figura 8.6: Esempio di collegamento di funzioni ad alcuni dei 4 piedini per le interruzioni

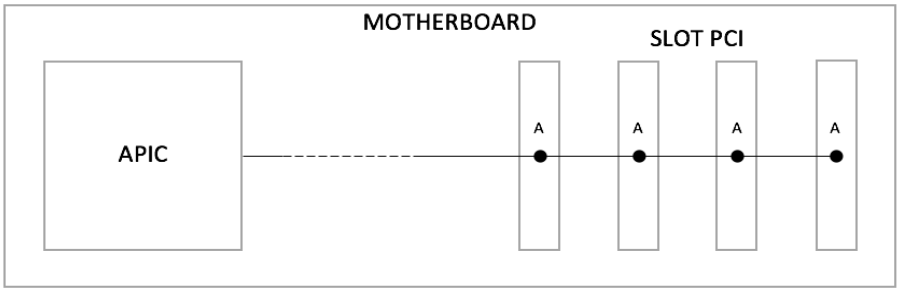


Figura 8.7: Collegamento di più piedini (dello stesso tipo) di più dispositivi all'APIC

E poiché un dispositivo possiede solo 4 piedini, anche se può implementare fino a 8 funzioni, è anche possibile che più funzioni si trovino a condividere uno stesso piedino per inviare richieste di interruzione. Tutti i piedini `/INTA`, tutti i piedini `/INTB`, tutti i piedini `/INTC` e tutti i piedini `/INTD` sono collegati a formare 4 linee che giungono all'APIC a partire dal piedino 16 (0-15 occupati da vecchi dispositivi). Il fatto di unire in uniche linee i 4 piedini di interruzione, implica che anche più funzioni di diversi dispositivi si trovano a condividere la stessa linea.

Ogni funzione può gestire blocchi di memoria o blocchi di I/O la cui dimensione è data da potenze di 2. Gli indirizzi dei blocchi sono programmabili: per fare questo ogni funzione possiede 6 **registri base** a 32 bit ciascuno. I registri si differenziano nel caso in cui riferiscano blocchi di memoria o blocchi di I/O.

Figura 8.8: Base Register

BaseRegister che punta a un blocco di memoria

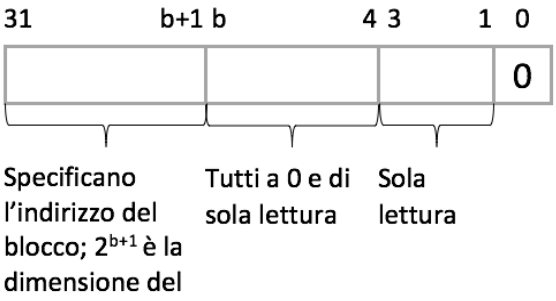


Figura 8.9: Base Register che riferisce blocchi di memoria

l'indice b può avere valori a partire da 4 quindi un blocco di memoria può essere grande minimo 16B.

BaseRegister che punta a un blocco di I/O

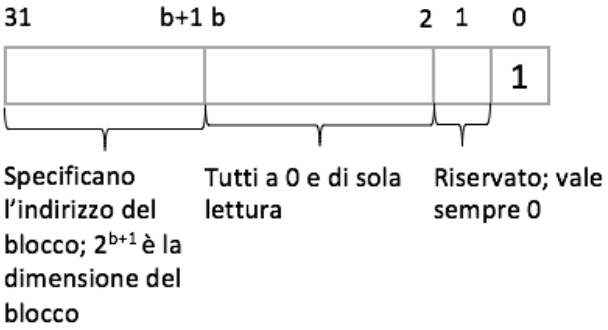


Figura 8.10: Base Register che riferisce blocchi di I/O

l'indice b può avere valori a partire da 2 quindi un blocco di I/O può essere grande al minimo 4B.

I dispositivi PCI possiedono un piedino di ingresso `IDSEL` e rispondono a transazioni di configurazione solo se `IDSEL` è attivo e `AD0` e `AD1` sono a 0. In questo caso, durante la fase di indirizzamento si legge:

1. dalle linee `AD8-AD10` il numero della funzione indirizzata;

2. dalle linee `AD0-AD7` l'offset della parola lunga indirizzata nello spazio privato di configurazione della funzione;

3. se la funzione è implementata, il dispositivo attiva `/DEVSEL`; poi si procede come in una normale transazione di lettura o scrittura.

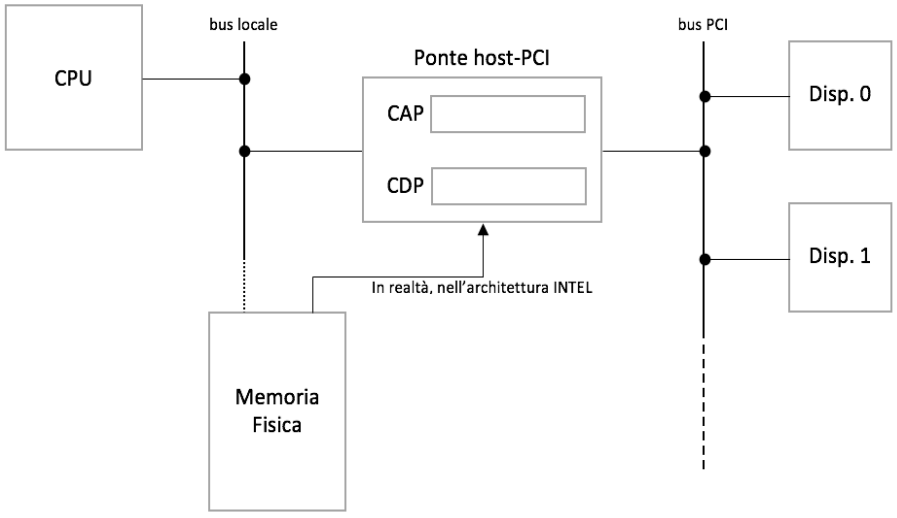


Figura 8.11: Visione d'insieme del collegamento tra bus PCI e bus locale tramite ponte host-PCI

Poiché la CPU non può generare direttamente transazioni nello spazio di configurazione (il processore infatti è in grado di produrre solo letture/scritture nello

spazio di memoria o di I/O), occorre servirci di due registri che si trovano nel ponte host-PCI. Il ponte host-PCI, essendo esso stesso una periferica collegata al bus locale, implementa parte dello spazio di I/O e quindi i suoi registri possono essere letti/scritti dalla CPU. Una lettura scrittura in questi registri provoca indirettamente transazioni nello spazio di configurazione. Questi due registri sono `CAP` (*Configuration Address Port*) e `CDP` (*Configuration Data Port*).

Il `CAP` è posizionato nello spazio di I/O all'indirizzo `0x0CF8` ed è fatto così:

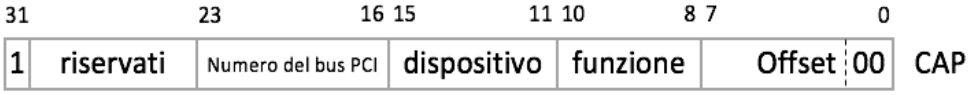


Figura 8.12: Registro CAP

Il `CDP` invece è posizionato all'indirizzo `0x0CFC`. Il processore per fare in modo che il ponte host-PCI generi delle transazioni di configurazione deve:

1. scrivere una parola lunga in `CAP`;

2. leggere/scrivere un byte, una parola, o una parola lunga in `CDP`.

La doppia parola da scrivere in `CAP` deve contenere tutte le informazioni necessarie per l'indirizzamento della parola lunga all'interno di una funzione di un dato dispositivo. Quando avviene l'operazione in `CDP`, il ponte host-PCI memorizza:

1. il tipo di operazione;

2. i valori di `BE` del bus locale, specificato dal processore, e genera transazioni di configurazione di lettura o scrittura.

Durante la fase di indirizzamento il ponte:

1. copia il bit 0-10 di `CAP` sulle linee `AD10-AD0` del bus PCI;

2. decodifica il numero del dispositivo utilizzando i bit 11-15 del `CAP` per generare il segnale `IDSEL` direttamente al dispositivo selezionato.

Nella fase successiva il ponte:

1. copia i valori di `BE` salvati sulla linea `C/BE` del bus PCI;

2. se la transazione è di scrittura copia il contenuto di `CDP` sulle linee `AD31-AD0`, altrimenti copia lo stato delle linee `AD31-AD0` in `CDP`;

3. se la transazione è di lettura e nessun dispositivo attiva `/DEVSEL` entro un tempo prestabilito (6 cicli di clock) il ponte completa ugualmente la lettura da `CDP` iniziata dalla CPU e restituisce un byte, una parola o una parola lunga composta da soli 1.

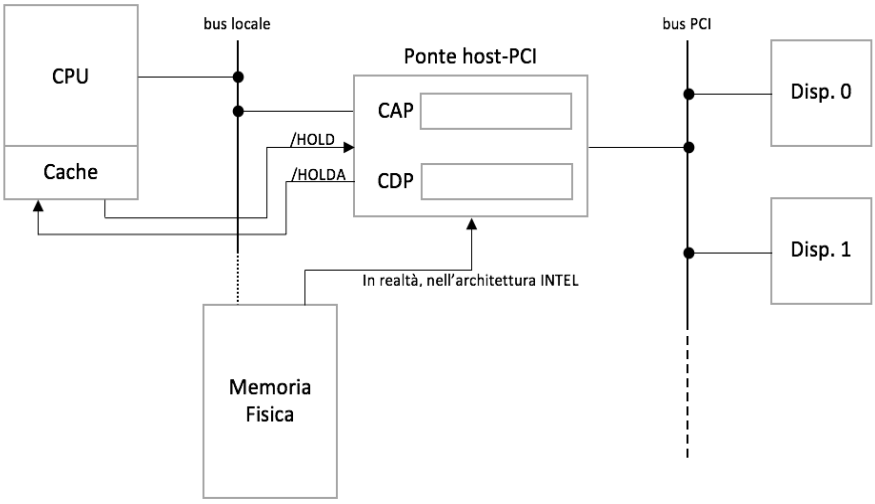


Figura 8.13: Visione d'insieme del collegamento tra bus PCI e bus locale tramite ponte host-PCI; collegamenti tra cache e ponte host-PCI

8.4 DMA tramite bus PCI: problemi e soluzioni

Abbiamo detto che sul bus PCI ogni dispositivo può anche essere iniziatore, oltre ad essere obiettivo. Questa possibilità esiste per consentire l'accesso diretto alla memoria da parte dei dispositivi PCI (DMA). Questo vuol dire che il software dice in anticipo alla scheda quanti byte deve trasferire e dove. Che vantaggio c'è? Guadagno di tempo. Infatti se si dovesse passare sempre dalla CPU si impiegherebbe più tempo rispetto a quando in realtà il dispositivo ha già i dati pronti. Occorre comunque che il software poi sappia quando il trasferimento è concluso. Il modo migliore per fare questo sono le interruzioni. Quindi l'idea è che il dispositivo possa operare per conto proprio e quando ha terminato lancia un'interruzione. Quindi il vantaggio è che disturbiamo il processore una sola volta al termine dell'operazione. Questa realizzazione comporta dei problemi non trascurabili:

1. si possono avere contemporaneamente diversi accessi alla memoria (CPU e un dispositivo), questo non deve accadere e la soluzione è che il processore si fa da parte; quando il ponte PCI vuole trasmettere fa una richiesta alla CPU e la CPU appena può gli passa il controllo mettendo tutti i suoi registri in alta impedenza, mentre il ponte toglie i suoi piedini dall'alta impedenza e pilota il bus.

2. il processore non accede direttamente alla memoria, accede alla cache. La cache accede alla memoria solo quando il processore cerca qualcosa che non è in cache (cosa che speriamo sia rara). Quindi il ponte host-PCI deve chiedere alla cache se può usare il bus locale. Il problema sta nel fatto che se i dispositivi modificano parti della memoria, la cache non ne sa nulla e il processore continua ad operare sui dati vecchi. Abbiamo anche il problema opposto: poiché la cache fa write back c'è il rischio che i dispositivi leggono in memoria vecchi dati. Questo problema le architetture INTEL lo risolvono da sole; in altre architetture invece deve essere il programmatore a gestire il problema. In che modo potrebbe occuparsene? Via software deve invalidare, tramite un'istru-

zione, la parte di cache che in memoria è interessata delle modifiche da parte dei dispositivi: il controllore della cache è esso stesso una periferica che ha un paio di registri che consentono di specificare il range di indirizzi da invalidare nella cache. Per fortuna gli INTEL ci pensano da soli sfruttando il fatto che il controllore della cache può vedere le operazioni che passano dal bus locale e invalida l'indirizzo al quale sta scrivendo un dispositivo PCI. Più complicato è il caso di lettura quando i dati più aggiornati si trovano in cache. Per ovviare a questo la memoria è in realtà collegata direttamente al ponte host-PCI e il ponte ha il tempo di sapere cosa ha fatto la cache. Potremmo inoltre sfruttare il fatto che la MMU ha un piedino per disabilitare la cache e potremmo disporre delle zone di memoria per cui la cache è disabilitata ed adibire quelle zone di memoria alle operazioni di trasferimento con i dispositivi.

Se la periferica è abilitata ad essere iniziatore, a fare cioè ***bus mastering***, occorre che abbia dei registri con gli indirizzi di memoria sui quali operare. Questi indirizzi sono fisici (non passano dalla MMU). Ora qui abbiamo un po' di problemi. Noi abbiamo il nostro programma che vuole dialogare con la periferica ed è un programma che si trova dentro lo spazio virtuale. Il programma utente non può dialogare direttamente con la periferica. Il sistema dovrà quindi scrivere nei registri della periferica l'indirizzo fisico proveniente dalla traduzione dell'indirizzo virtuale. Si può avere un problema quando abbiamo un buffer da leggere/scrivere in memoria virtuale e questo corrisponde a pagine non consecutive in memoria fisica. Infatti il dispositivo sa solo l'indirizzo di partenza e quanti byte leggere/scrivere da quell'indirizzo; non siamo in grado di gestire salti. Questo lo risolviamo facendo l'assunzione di non usare mai buffer di questo tipo; useremo solo buffer con pagine consecutive in memoria fisica. Un altro problema si ha quando una periferica sta scrivendo qualcosa in una pagina di memoria e contemporaneamente la routine di page-fault sceglie quella pagina come vittima. Questa situazione si può risolvere solo a priori dicendo nel bus mastering che quella pagina non può essere eliminata.

8.5 Bus Mastering

Abbiamo visto che il fatto che un dispositivo PCI possa fungere sia da iniziatore che da obiettivo, è sfruttabile per fare accessi diretti in memoria. Per fare questo dobbiamo dire alla periferica come operare:

1. trasferite dati da se stessa alla memoria o viceversa;
2. quanti e dove sono questi dati.

Nello spazio di configurazione della funzione vi è un registro di stato, da 2 byte, che contiene delle informazioni fra cui quella che ci dice se la funzione è in grado o meno di effettuare operazioni di bus mastering. Sempre nello spazio di configurazione, nel registro di comando, c'è un bit per abilitarla a fare bus mastering. Il modo in cui effettivamente si passano i dati dipende da periferica a periferica. Un caso è quello in cui si prevede che la periferica disponga di tre registri esplicitamente destinati al bus mastering (Figura 8.15).

Si prepara in memoria una struttura dati che è sostanzialmente un array di descrittori di buffer (Figura 8.16).

Passiamo in un *base register* il puntatore a questa struttura dati e poi la periferica lo trasferisce nel `BMPTR`. Ogni elemento dell'array avrà un puntatore al buffer, la

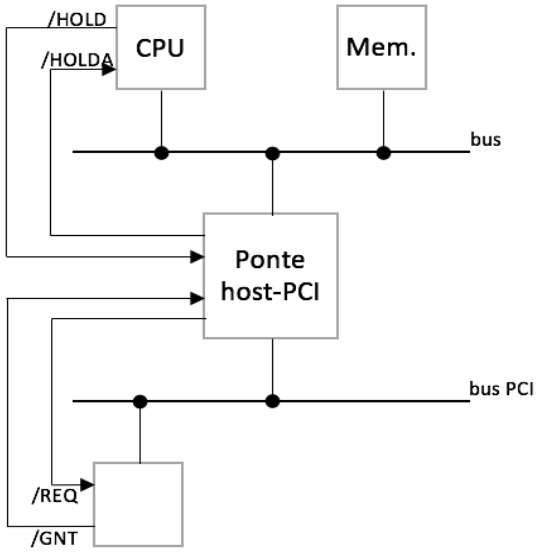


Figura 8.14: Visione d'insieme dei collegamenti fra i bus e di quelli fra ponte host-PCI, CPU e dispositivi PCI

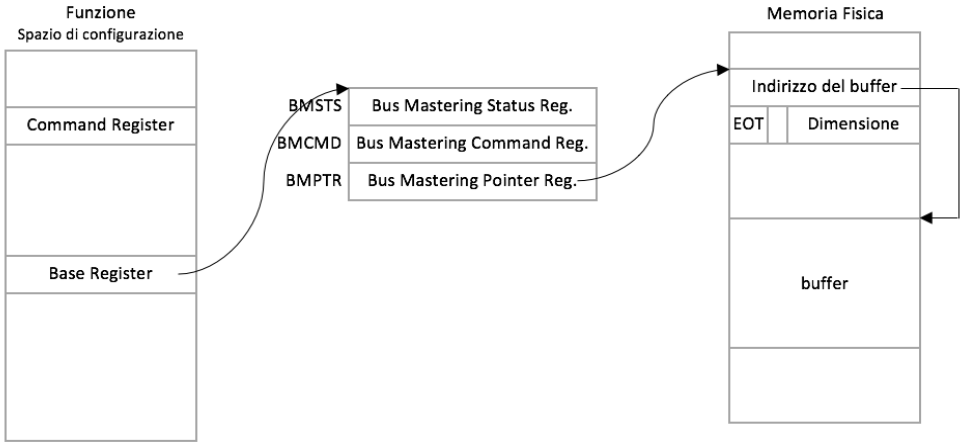


Figura 8.15: Funzione con capacità di bus mastering

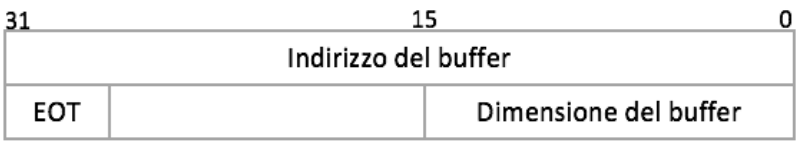


Figura 8.16: Descrittore di buffer

dimensione in byte del buffer e un bit EndOfTable (se è 0 vuol dire che l'array non è terminato, se è 1 vuol dire che quello è l'ultimo elemento dell'array). Possiamo avere più di un buffer. La periferica invierà l'interrupt per segnalare il termine della transazione; lo invierà quando ha terminato. La direzione del trasferimento sta scritta nel registro di comando quindi, per un'unica operazione, o tutti i trasferimenti sono verso la memoria o tutti i trasferimenti sono dalla memoria.

Il fatto di poter programmare la periferica in questo modo, ci permette, se vogliamo, di risolvere il problema che le pagine che compongono il buffer sono consecutive in memoria virtuale ma non in quella fisica. La periferica può "vedere" solo la memoria fisica. Dobbiamo comunque garantire che il buffer sia tutto presente e non rimpiazzabile.

Quindi a questo livello astratto, l'idea è che viene preparata la struttura dati, ne scrive l'indirizzo in `BMPTR` e nel registro di comando deve scrivere la direzione del trasferimento e il fatto che il trasferimento deve iniziare. Una volta che queste informazioni vengono scritte, la periferica prosegue in autonomia; ad un certo punto invierà un interrupt e la risposta all'interrupt sarà la lettura nel registro di stato `BMSTS` (in cui c'è scritto cosa è successo e in particolare se ci sono stati degli errori).

8.5.1 Bus mastering con l'interfaccia ATA

Dimensioniamo il problema all'Hard Disk. L'HD è un oggetto che esiste da prima dell'introduzione dello standard PCI e aveva un suo modo per fare l'accesso diretto alla memoria (DMA) che si basava sulla presenza nel sistema di un controllore DMA che oggi non è più presente.

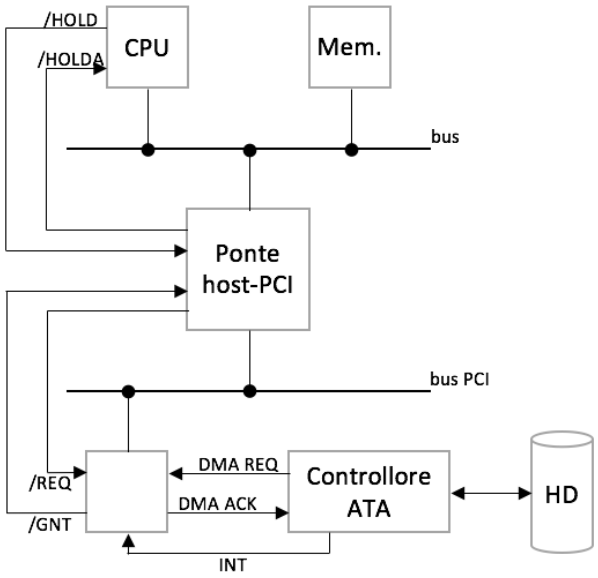


Figura 8.17: Visione d'insieme dei collegamenti fra i bus e di quelli fra ponte host-PCI, CPU, dispositivi PCI, controllore ATA e Hard-Disk

Il controllore ATA è collegato all'HD e ha due collegamenti DMAREQ (*DMA REQuest*) e DMAACK (*DMA ACKnowledge*). Questi collegamenti dovrebbero in teoria andare al controllore DMA e, quando l'HD è programmato per questo, nel momento in cui ha i dati pronti, invia una richiesta al controllore DMA tramite DMAREQ e il controllore risponde con DMAACK. Dopodiché non è l'HD a fare

direttamente il trasferimento, ma è il controllore DMA che si occupa di operare sul buffer dell'HD. Tutto questo non fa parte dello standard PCI però possiamo mettere un ponte fra il bus PCI e "qualunque" altra cosa. Uno dei chip che si occupa di fare da ponte verso altri standard, si occupa di gestire l'HD. Questo ponte è collegato all'HD tramite gli stessi piedini cui l'HD era controllato al controllore DMA. Il ponte si comporta da controllore DMA verso l'HD e come bus master verso il bus PCI. L'HD pensa di parlare con un controllore DMA. L'unico problema lo abbiamo al termine dell'operazione quando il controllore ATA mandava direttamente una richiesta di interruzione al controllore delle interruzioni per comunicare il termine delle operazioni. Il problema sta nel fatto che il controllore DMA sa di aver finito quando ha reso disponibile l'ultimo dato, ma quel dato potrebbe sempre essere nel buffer temporaneo del ponte controllore ATA-PCI e quindi non essere in realtà stato scritto in memoria. Quindi l'interruzione non può essere portata direttamente all'APIC; deve essere prima mandata al ponte controllore ATA-PCI, poi sarà questo oggetto che la invierà all'APIC quando il dato è stato realmente trasferito.

Fino ad ora abbiamo usato periferiche vecchie e abbiamo assunto di sapere dove si trovano i registri di queste periferiche; il ponte controllore ATA-PCI è una periferica PCI ed i suoi registri non sono ad un indirizzo fisso. Dobbiamo quindi sapere a che indirizzo si trovano. È il software che decide dove stanno e a questo ci pensa il BIOS quindi noi ce li abbiamo già pronti. Un'altra cosa che non sappiamo è dove si trova questa periferica, se c'è: va quindi cercata. Il ponte ATA-PCI ha come caratteristica di avere nel suo spazio di configurazione all'offset 10, due byte con contenuto 0x0101 quindi si scandiscono tutte le funzioni di tutti i dispositivi finché non si trova una funzione con quella caratteristica.

Al termine di tutta l'operazione occorre ricordare che, poiché l'HD e il ponte ATA-PCI sono due dispositivi completamente indipendenti, bisogna sia leggere nel registro di stato del controllore ATA per rispondere alla richiesta di interruzione che ha inviato al ponte ATA-PCI (anche se crede di averla mandata all'APIC), sia leggere nel registro di stato del ponte ATA-PCI per rispondere alla richiesta di interruzione che ha inviato al controllore APIC.

Capitolo 9

Architettura interna del processore

9.1 Introduzione

Vediamo una CPU un po' più moderna. La CPU vista fino ad ora abbiamo detto che fa, ciclicamente, `fetch`⟶`decode`⟶`execute` e che fra un'istruzione e l'altra controlla se ci sono richieste di interruzione. Anche se le CPU moderne, "viste da fuori", fanno la stessa cosa, dentro attuano una serie di accorgimenti per cercare di eseguire queste istruzioni il più velocemente possibile in modo tale che all'esterno l'effetto sia lo stesso ma all'interno si possa andare più velocemente. Per un lungo periodo di tempo l'aumento delle prestazioni di un processore nuovo, derivava semplicemente da un clock più veloce (praticamente raddoppiava ogni generazione). Quello che oggi i progettisti cercano di fare (anche se il clock rimane più o meno fermo) è aumentare lo spazio all'interno dello stesso chip (transistor più piccoli). Essendoci più spazio, la CPU può fare più cose e magari più cose contemporaneamente. Ultimamente quello spazio è usato per mettere all'interno dello stesso chip più processori.

9.2 *Pipeline*

Vediamo come in un unico core il processore cerchi di eseguire più istruzioni contemporaneamente (non da più flussi di esecuzione ma da uno solo). Per fare questo ci sono vari modi. Uno di questi è la ***pipeline***. Ogni istruzione deve passare da una serie di fasi. (Figura 9.1).

Prelievo istruzioni	Decodifica	Prelievo operandi	Esecuzione	Scrittura risultato

Figura 9.1: Fasi di una pipeline

Queste fasi sono eseguite da parti fisicamente diverse del processore (circuito che fa il prelievo, circuito che fa la decodifica, ALU, FPU, ecc..). Si pensi, ad esempio,

alla CEP[1] (Calcolatrice Elettronica Pisana) che ha la CPU letteralmente fatta da armadi ed ogni armadio rappresenta una diversa fase.

Se va tutto bene, quindi, nel momento in cui abbiamo prelevato un'istruzione e siamo in fase di decodifica, la parte di circuito che fa il prelievo in quel momento **non** è usata. Potremmo quindi utilizzare quella parte di circuito per prelevare la successiva istruzione (sappiamo quasi sempre l'istruzione successiva, l'unico caso in cui non lo sappiamo è quando c'è un salto).

Quindi avremo un diagramma temporale (caso ottimo) del tipo:

Figura 9.2: Diagramma temporale dell'utilizzo dei circuiti con una pipeline

Se va tutto bene questo procedimento può andare avanti a lungo.

In un processore semplice una sequenza `fetch-decode-execute`, possiamo dire che veniva fatta da un'unica rete combinatoria, in un unico seppur lungo ciclo di clock.

Se vogliamo passare a questa soluzione dobbiamo aggiungere dei registri a valle di ciascun circuito combinatorio che si occupa di ognuna delle fasi:

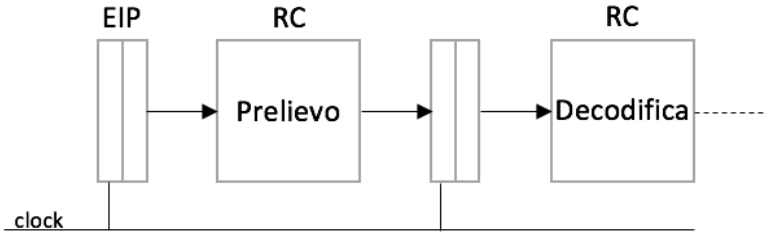


Figura 9.3: Registri posti a valle delle reti combinatorie adibite alle varie fasi

Poiché finché non arriva un segnale di sincronizzazione i registri sono insensibili alle variazioni di stato di ingresso, occorrono 5 cicli di clock per ogni istruzione. I registri sono indispensabili perché le RC cambiano "immediatamente" il valore dello stato di uscita quando c'è un nuovo stato di ingresso e per la struttura che vogliamo adottare questo rappresenterebbe un problema. Per esempio: se stessimo eseguendo la decodifica per l'istruzione i e prelevassimo l'istruzione $i + 1$, in ingresso alla RC di decodifica arriverebbe il nuovo stato dell'uscita della RC di prelievo.

Quindi ogni fase deve essere "scollegata" dalla precedente.

[1]La Calcolatrice Elettronica Pisana si trova esposta al Museo Nazionale degli Strumenti per il Calcolo di Pisa

Dal punto di vista della velocità:

- prima avevamo una macchina che in unico clock faceva "tutto" (un'istruzione, un ciclo di clock);
- ora una istruzione deve passare attraverso tutte quelle fasi; a ogni fase c'è da attraversare un registro e quindi servono 5 clock per istruzione.

La soluzione è aumentare la velocità del clock. Da cosa dipende la velocità del clock? Dal più lungo percorso tra un registro e l'altro all'interno del un circuito. E poiché nel processore la distanza fra un registro e l'altro è di almeno 5 reti combinatorie (mettiamo che il ritardo sia 5Δ), osserviamo che con questa soluzione la distanza fra un registro e l'altro diminuisce essendoci una sola RC a separali (mettiamo che il ritardo sia di 1Δ). Dal punto di vista delle singole istruzioni prima si aveva $\Delta + t_{setup1registro}$, ora $5\Delta + t_{setup5registri}$. Un po' si perde ma dal punto di vista complessivo è meglio perché abbiamo un'istruzione portata a termine ogni clock di 1Δ.

Per i processori INTEL la cosa è un po' più complicata perché non è possibile suddividere prelievo e decodifica poiché le istruzioni hanno lunghezza variabile e quindi non possiamo sapere quanto è grande l'istruzione finché non l'abbiamo letta.

Altro problema: siamo sicuri di non andare mai ad utilizzare per fasi diverse la stessa parte di circuito nella CPU?

Nel set di istruzioni dei processori INTEL purtroppo può capitare perché tutte le istruzioni possono avere gli operandi sia in memoria che nei registri e quindi se nello stesso momento la CPU deve fare un prelievo operandi di una istruzione e scrittura risultato di un'altra istruzione, potrebbe voler, per tutte le due istruzioni, accedere in memoria.

Quindi sorgono alcuni problemi.

9.3 Processori RISC

I processori RISC *(Reduced Instruction Set Computer)*, sono nati proprio con la concezione di utilizzare un set di istruzioni apposito per evitare quanto sopra descritto. Tutti i processori RISC tipicamente hanno istruzioni grandi 4 byte. All'interno di ciascuna istruzione i campi sono fissi:

Figura 9.4: Campi di un'istruzione RISC

Abbiamo anche una **netta** separazione fra le operazioni di memoria e quelle sui registri. Ci sono solo due istruzioni, `load` e `store`, per operare sulla memoria. Le istruzioni operative agiscono solo sui registri. Con queste semplificazioni si riesce a fare una pipeline.

Gli INTEL fanno questo:

via hardware le istruzioni IA32 passano da un modulo di traduzione che le trasforma in una sequenza di **istruzioni elementari** che invece sono RISC. Molte istruzioni di tipo CISC si traducono in un'unica istruzione elementare, altre no. Gli INTEL adoperano questa tecnica dal 686 (prima del Pentium).

Non possiamo sapere esattamente quali sono le IE, ma supponiamo:

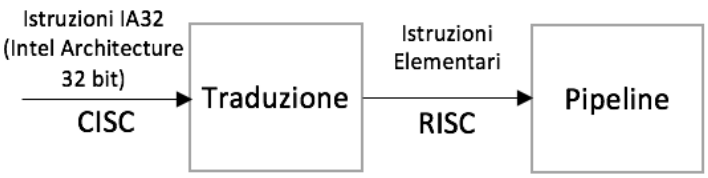


Figura 9.5: Traduzione da istruzioni CISC a istruzioni RISC

```
op dst, src1, src2 //Istruzioni Operative: tutti i registri.
    Destinatario e sorgenti sempre specificati.

op reg, offset(base) //Istruzioni di Memoria: load o store; hanno un
    unico formato.

op reg, offset //Istruzioni di Salto
```

9.4 Le e-istruzioni

Riprendiamo lo schema della pipeline:

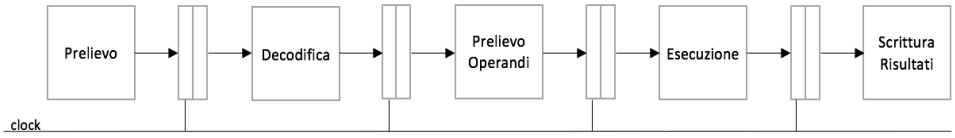


Figura 9.6: Schema pipeline completo con registi di pipeline

Ciascuno di questi circuiti si occupa di una fase diversa. Tra i vari stati inseriamo dei ***registri di pipeline*** in modo tale che ogni stato, ad ogni clock, scriva il risultato nel registro di pipeline mentre lo stato precedente sta lavorando sul risultato di prima. Per poter utilizzare questa struttura, i progettisti INTEL hanno dovuto prendere le istruzioni Assembly ereditate dalle vecchie macchine e predisporre un circuito atto a trasformarle in *e-istruzioni*. Le e-istruzioni hanno i seguenti vantaggi:

- hanno tutte la stessa lunghezza (non occorre sapere quanto sommare, sono tutte grandi 4 byte);
- il fatto che siano semplici permette di ridurre il clock al minimo. Infatti il valore minimo di clock è dato dal tempo più lungo che impiega una fase per portarsi a termine. Solo se le istruzioni sono semplici si può garantire un'uniformità fra i tempi di esecuzione delle varie fasi.

9.5 Problemi legati all'introduzione della pipeline

All'interno del flusso delle istruzioni ci sono alcune situazioni che impediscono di eseguire un'istruzione ad ogni ciclo di clock. Queste situazioni prendono il nome di *alee* e all'interno del flusso di istruzioni possono causare quelli che si chiamano ***stalli della pipeline***. Gli stalli sono cicli di clock in cui non viene incominciata una nuova istruzione e quindi perdiamo quel ciclo di clock.

Abbiamo alee di tre tipi:

- alee **strutturali**;
- alee **sui dati**;
- alee **sul controllo**.

Le **alee strutturali** dipendono dal fatto che, in teoria, per eseguire 5 istruzioni in parallelo occorre che ad ogni passo queste istruzioni facciano uso di risorse distinte del processore. Queste risorse potrebbero non essere distinte per ogni possibile combinazione di istruzioni. Basti pensare a due istruzioni i e $i+2$ di cui la prima deve scrivere il risultato e la seconda deve prelevare operandi in memoria: entrambe vorrebbero utilizzare il circuito che serve per operare sulla memoria ma ciò non è possibile, oltre al fatto che alla memoria non si può accedere contemporaneamente.

Questa situazione può essere risolta con uno stallo; fermiamo, cioè, il flusso di esecuzione e lo facciamo ripartire al clock successivo. In generale, se c'è un conflitto fra le istruzioni, possiamo sempre risolverlo mettendo la pipeline in stallo per almeno un ciclo di clock (al massimo 5 cicli di clock essendo 5 le fasi). Vogliamo però ridurre gli stalli al minimo perché ogni stallo è un clock sprecato. Assumiamo che ci sia un circuito a parte che controlla il flusso della pipeline e decide quando far entrare qualcosa di nuovo. Lo fa sapendo quale istruzione è già nella pipeline e quale sia la nuova istruzione e in base a questo decidere se lasciare in stallo la pipeline permettendo dunque ad una nuova istruzione di essere eseguita.

Le **alee sui dati** sono istruzioni che usano il risultato di un'istruzione precedente. Per esempio:

```
ADD R1, R2, R3
SUB R4, R1, R5
```

La e-istruzione `SUB` ha come operando il risultato della e-istruzione `ADD` precedente ad essa. Quando la `SUB` deve prelevare l'operando da `R1`, la `ADD` si trova sempre in fase di esecuzione. In questo caso per risolvere l'alea dovremmo aggiungere due stalli.

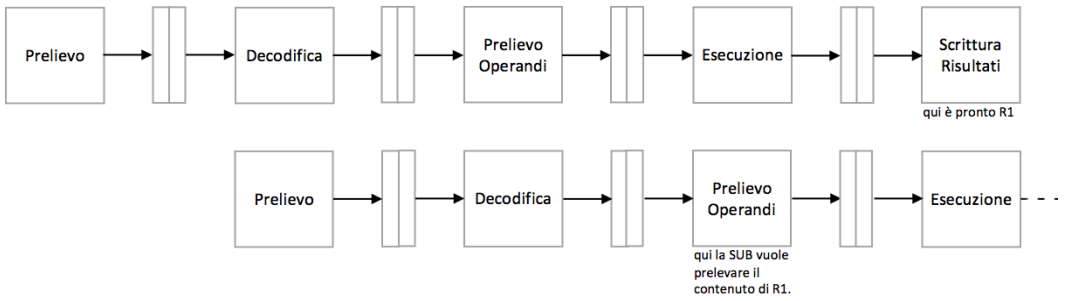


Figura 9.7: Stallo sui dati, schema relativo all'esempio

Situazioni come queste sono molto frequenti e non possiamo permetterci di perdere clock così spesso.

La soluzione è possibile ottenerla attraverso l'inserimento di un circuito di *by-pass* pilotato da quel circuito che controlla il flusso della pipeline.

È vero che l'istruzione `SUB` vuole leggere il registro `R1`, ma più che il registro le interessa il suo contenuto. Il risultato della `ADD` è pronto già subito dopo la fase di esecuzione; quindi se predisponessimo un circuito di *by-pass* che ci consentisse di ottenere il risultato appena questo è pronto, potremmo risolvere gli stalli anche

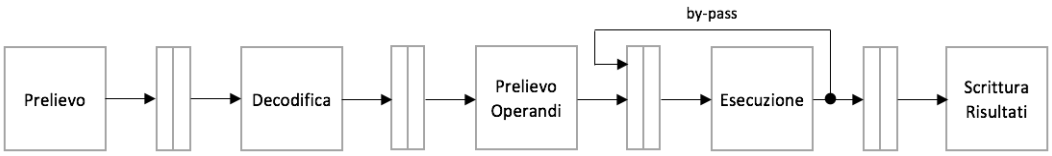


Figura 9.8: Schema pipeline con by-pass

in questo caso. Ci dovrà essere uno switch che decide quale delle due entrate del registro deve passare.

L'ultima categoria di alee sono le **alee sul controllo**. Si verificano quando ci sono istruzioni di salto, soprattutto di salto condizionato. Non sappiamo, finché non è stata seguita l'istruzione di salto, quale sia l'istruzione successiva. Quindi nel frattempo cosa inseriamo nella pipeline? La cosa più semplice sarebbe non inserire nulla e aspettare che l'istruzione abbia calcolato l'indirizzo corretto a cui saltare. Questa soluzione, però, data la frequenza con cui vengono utilizzate le istruzioni di salto, rallenterebbe troppo il flusso di esecuzione. Un'altra cosa che però possiamo fare è provare a indovinare il risultato dell'istruzione di salto. Indovinando è possibile sbagliare e magari si scopre, al termine dell'istruzione di salto, che la parte a cui siamo saltati è la parte sbagliata. Questo non rappresenta un problema perché queste istruzioni non hanno ancora fatto nulla di permanente. È sufficiente, prima che queste istruzioni abbiano scritto il loro risultato, conoscere il risultato dell'istruzione di salto ed eventualmente mettere un flag ad indicare che le istruzioni eseguite fino a quel punto non devono avere effetti. Come si fa ad "indovinare" qual è il risultato dell'istruzione di salto? Bisogna vedere cosa è successo in passato e sperare che accada più o meno la stessa cosa. Il modo più semplice per fare questa cosa è una *predizione statica*[2]: se l'offset è negativo potrebbe essere un ciclo e se è un ciclo è probabile che un po' di volte occorrerà ripetere quelle istruzioni. Se il salto è in avanti forse siamo in presenza di una condizione e quindi la predizione è molto più scarsa. Possiamo, in questo caso, predire che il salto non verrà fatto. I meccanismi di previsione che sono nel processore fanno in realtà qualcosa di più sofisticato ricordando di quella istruzione cosa sia successo in passato. Hanno una cache delle istruzioni di salto già viste e per ognuna di esse si ricordano la storia dei salti (una serie di bit).

9.6 Esecuzione fuori ordine

Una pipeline è soltanto la tecnica più elementare per cercare di velocizzare il funzionamento di un processore.

Esistono altri metodi per ottenere questo effetto. Supponiamo che ad un certo punto ci accorgiamo che una certa istruzione non può passare perché c'è un'alea strutturale e quell'istruzione ha bisogno della ALU in due stadi e non possiamo farla partire subito dopo la precedente. Invece di sprecare quel clock, possiamo vedere se possiamo far passare l'istruzione successiva. Bisogna però porre attenzione. Nel

[2]I processori che impiegano questa tecnica considerano sempre i salti verso la parte precedente del codice come "accettati" (ipotizzando che siano le istruzioni riguardanti un ciclo) e i salti in avanti sempre come "non accettati" (ipotizzando che siano uscite precoci dal ciclo o altre funzioni di programmazione particolari). Per cicli che si ripetono molte volte, questa tecnica fallisce solo alla fine del ciclo. Fonte: https://it.wikipedia.org/wiki/Predizione_delle_diramazioni

flusso di esecuzione di un programma è abbastanza frequente che istruzioni che sarebbero sequenziali, in realtà non dipendono l'una dall'altra. Esempio classico è un ciclo for per inizializzare un array di N elementi con la somma degli elementi di altri due array di N elementi. Ciascuna somma è a sé, non ha alcuna importanza l'ordine in cui vengono fatte. L'importante è che alla fine vengano fatte tutte. Quindi nel flusso delle e-istruzioni verranno fuori un migliaio di istruzioni che non importa che vengano lasciate in quell'ordine. Questa cosa si può sfruttare progettando il processore in modo che esegua le istruzioni nel primo momento possibile. Questo si può fare purché l'istruzione abbia le risorse per essere eseguita e abbia i suoi operandi. La tecnica descritta si chiama ***esecuzione fuori ordine***. Serve a risolvere gli stalli; invece di attendere, si vede se l'istruzione successiva può essere eseguita.

Possiamo eseguire una e-istruzione nel primo momento in cui ci sono risorse libere per la sua esecuzione tenendo conto delle seguenti dipendenze:

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

Una e-istruzione dipende sui **dati** se usa un risultato prodotto dalla e-istruzione precedente oppure anche *transitivamente*, cioè legge il risultato di una e-istruzione intermedia che dipende sui dati dall'istruzione ancora precedente.

La dipendenza sui nomi è un po' più particolare. Possiamo averne di due tipi:

```
//R2 sorgente e poi destinatario
ADD     R1, R2, R3
ADD     R2, R4, R5

//R1 destinatario e poi nuovamente destinatario
ADD     R1, R2, R3
ADD     R1, R4, R5
```

La dipendenza sul *controllo* si ha quando una e-istruzione può essere eseguita o meno in base al risultato della e-istruzione di salto.

Queste dipendenze sono importanti perché se una e-istruzione dipende, per un motivo qualunque, dalla e-istruzione precedente, non possiamo riordinarle a meno che non adottiamo qualche accorgimento.

Le dipendenze sui dati non possono essere risolte perché sono quello che il programma sta facendo e non possiamo modificare il significato del programma.

Le dipendenze sui nomi possono, invece, essere risolte.

9.7 Organizzazione interna del processore

Vediamo che è possibile organizzare il processore in modo da permettere l'esecuzione fuori ordine tenendo conto delle dipendenze. Possiamo avere più ALU (più ce ne sono più il processore costa).

Davanti a ciascuna ALU mettiamo una pila di e-istruzioni. Queste strutture prendono il nome di **stazioni di prenotazione**. In queste stazioni di prenotazione si vanno ad accodare le e-istruzioni in attesa che siano pronti i loro risultati.

Per ogni registro ci ricordiamo un paio di cose:

- se c'è una e-istruzione di quelle in coda che vorrebbe scriverci (W);

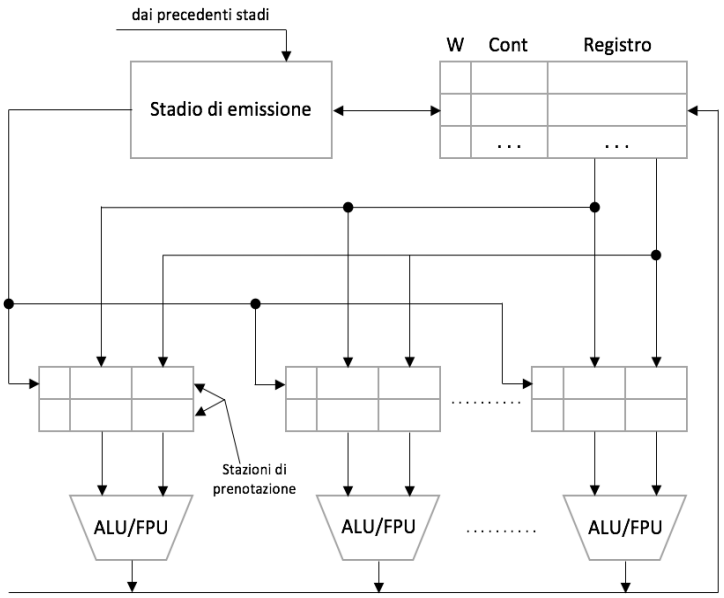


Figura 9.9: Schema con evidenziazione delle stazioni di prenotazione e della struttura dati W-Cont-Registro

- quante delle e-istruzioni che sono in cosa vorrebbero leggerci (Cont).

Lo stadio di emissione consulta la tabella ed emette le e-istruzioni, in una di quelle code dove c'è posto e dove ci sono le risorse per eseguire la e-istruzione.

I risultati delle ALU vengono portati poi nella tabella per scrivere nei registri.

Dalla struttura dati introdotta W-Cont-Registro, possiamo già capire se l'istruzione dobbiamo fermarla perché viola qualche dipendenza oppure se possiamo farla già partire.

9.7.1 Regole per emettere una e-istruzione

Per prima cosa guardiamo la destinazione e guardiamo il flag W: se W è a 1 abbiamo dipendenza sui dati perché la e-istruzione vuole leggere un risultato di una e-istruzione che deve ancora essere messa in coda; questa la emettiamo ugualmente e andrà a finire in una stazione di prenotazione, quella della ALU che la potrà eseguire, in attesa che i suoi operandi siano pronti. Quando sono pronti si prelevano e poi si va ad eseguire l'istruzione.

9.7.2 Rinomina dei registri

Come già accennato, le dipendenze sui nomi possono essere eliminate. La tecnica che utilizziamo è quella che viene chiamata **rinomina dei registri**.

Prendiamo il seguente esempio:

```
ADD     R1, R2, R3      #usiamo R2 come sorgente
ADD     R2, R4, R5      #usiamo R2 come destinatario
SUB     R6, R2, R7
```

Usiamo `R2` come sorgente nella prima `ADD` e come destinatario nella seconda `ADD`: la scelta di `R2` come destinatario non e' fondamentale infatti avremmo potuto scegliere un altro registro. Pertanto questo esempio potrebbe essere riscritto come:

```
ADD     R1, R2, R3      #usiamo R2 come sorgente
ADD     R20, R4, R5     #usiamo R20 come destinatario
SUB     R6, R20, R7
```

Apportando questa modifica non abbiamo alterato il senso del programma, ma siamo riusciti ad eliminare una dipendenza sui nomi.

Questa cosa il processore la può fare in autonomia aggiungendo un componente che fa la rinomina dei registri.

Tutto ciò si può fare "parlando" indirettamente dei registri.

Distinguiamo **registri logici** e **registri fisici**.

I registri logici sono quelli che compaiono nel flusso del programma. Quelli su cui si fanno davvero i conti sono i registri fisici. Possiamo anche avere più registri fisici che logici.

La struttura dati W-Cont-Registri, diventerà dunque una struttura del tipo W-Cont-Registri Logici-Registri Fisici.

Per ogni registro logico viene specificato mediante una struttura costituita da puntatori, quale registro fisico contiene il suo valore.

Avremo dunque una struttura del genere:

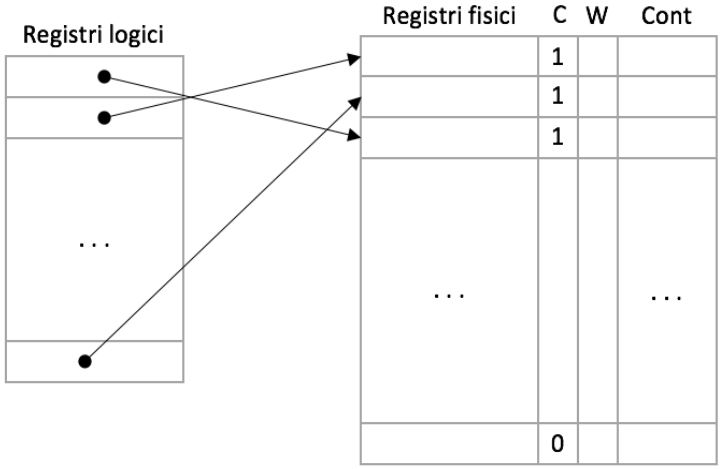


Figura 9.10: Corrispondenza registri logici-registri fisici

Quando una e-istruzione viene emessa, si utilizza la corrispondenza esistente fra registri logici sorgente `(src1, src2)` e registri fisici e viene creata una nuova corrispondenza fra il registro logico destinatario (dest) e un registro fisico libero.

I campi aggiuntivi sono associati ai registri fisici:

- W: a 1 se nel registro fisico F deve scrivere almeno una e-istruzione emessa;
- Cont: numero delle e-istruzioni emesse che deve leggere dal registro F;
- C: a 1 se vi è una corrispondenza fra un registro logico e il registro fisico F a cui il bit C appartiene.

Un registro fisico F si considera libero se:

- C = 0 $\longrightarrow$ nessuna corrispondenza di F con alcun registro logico;
- W = 0 $\longrightarrow$ nessuna e-istruzione emessa deve scrivere in F;

- Cont = 0 $\longrightarrow$ nessuna e-istruzione emessa deve leggere da F.

Emissione della e-istruzione:

- incremento di Cont dei registri fisici corrispondenti ai registri logici sorgente coinvolti;
- settaggio dei bit C e W per il registro fisico corrispondente al registro logico destinatario; dunque creiamo una nuova corrispondenza;
- resettaggio dei bit C e W per i registri fisici corrispondenti ai registri destinatari della vecchia corrispondenza.

Completamento della e-istruzione:

- decremento del campo Cont per i registri fisici corrispondenti ai registri logici sorgente;
- resettaggio di W del registro fisico corrispondente al registro logico destinatario.

9.8 Esecuzione speculativa

La tecnica della rinomina dei registri, opportunamente affiancata alla tecnica della predizione dei salti, ci permette anche di aggirare le limitazioni dovute alle dipendenze sul controllo. Possiamo eseguire le e-istruzioni dipendenti da e-istruzioni di salto non ancora risolte purché i risultati di queste vengano scritti in registri temporanei e trasferiti nei registri reali **solo** quando abbiamo la certezza che quelle e-istruzioni andavano realmente eseguite. Questa tecnica è detta ***speculazione***. Per dare corpo a questa tecnica occorre introdurre un nuovo stadio detto di *ritiro delle istruzioni*. Il completamento delle e-istruzioni provoca la scrittura dei risultati solo in registri temporanei: tali risultati diverranno effettivi solo se la e-istruzione che li ha prodotti passa lo stadio di ritiro. Lo stadio introdotto fa uso di una struttura dati chiamata ROB (*ReOrder Buffer*, buffer di riordino). Il ROB è una coda di descrittori di e-istruzioni.

Per ogni e-istruzione il ROB ne memorizza il tipo (se è operativa o di controllo), se è stata completata o è ancora in fase di esecuzione e, se la e-istruzione è di controllo, qual è l'esito previsto per il salto. Le e-istruzioni possono essere completate in un qualsiasi ordine ma sono ritirate dalla testa del ROB nel modo in cui sono immagazzinate in esso. Quando la e-istruzione in testa al ROB risulta completata, viene ritirata compiendo le seguenti azioni:

- se la e-istruzione è operativa:
  - il risultato da essa prodotto diventa effettivo e questa viene estratta dalla coda
- sa le e-istruzione è di controllo:
  - se la previsione dell'esito del salto è **sbagliata**, viene svuotato tutto il ROB con conseguente annullamento delle istruzioni eventualmente completate (scrittura nei registri fisici destinatari e modifica dei campi C, W, Cont associati ai registri fisici utilizzati);

– se la previsione dell'esito del salto è **corretta**, non viene fatta nessuna azione e questa e-istruzione viene estratta dalla testa del ROB.

Per poter rendere effettivi o annullare i risultati di una e-istruzione occorre estendere la struttura dati registri logici-fisici in modo che ogni registro logico possegga una doppia corrispondenza con registri fisici (due puntatori).

Nel ROB viene anche memorizzato temporaneamente il risultato delle e-istruzioni.

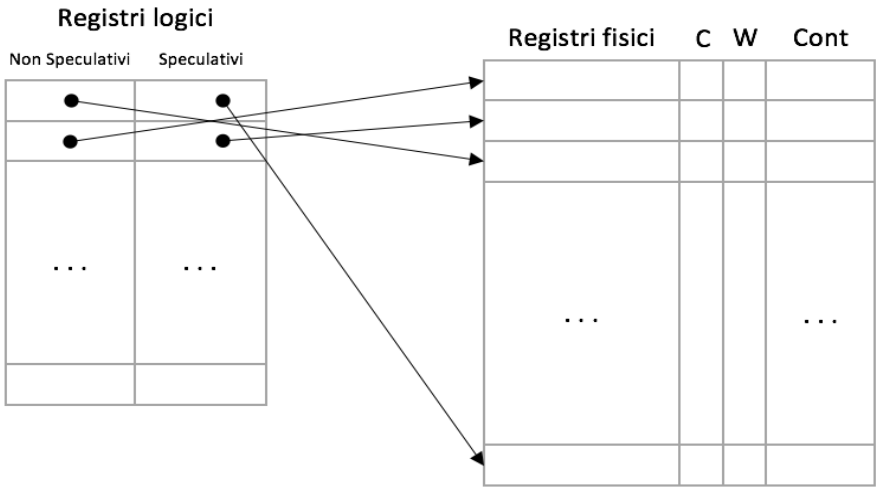


Figura 9.11: Registri logici speculativi e non speculativi e corrispondenza con registri fisici

Abbiamo quindi due tipi di corrispondenza per ogni registro logico:

- corrispondenza **non speculativa**: puntatore al registro fisico in cui è contenuto il valore dell'ultima e-istruzione ritirata dal ROB che lo aveva come destinatario;
- corrispondenza **speculativa**: puntatore al registro fisico che contiene (o conterrà) il valore corrente.

Il significato dei valori di C, W e Cont rimane invariato rispetto a quello precedentemente visto; solo alcune precisazioni:

- quando il bit W vale 1 si dice che il registro fisico è **bloccato**;
- il bit C vale 1 quando il registro fisico è selezionato da un puntatore (sia esso speculativo o non speculativo).

Il registro fisico si dice **libero** se C, W e Cont valgono 0. Un registro fisico si dice:

- **bloccato** se W = 1;
- **sbloccato** se W = 0;

Per le ragioni viste servono più registri fisici che registri logici; al massimo avremo che i registri fisici sono il doppio di quelli logici.

Quando si emette una e-istruzione, è sufficiente guardare i valori speculativi; se la e-istruzione è arrivata in cima al ROB si copia il valore speculativo in quello non speculativo. Se invece in cima al ROB arriva una e-istruzione di controllo e si scopre che per questa si era predetto il salto in modo **sbagliato**, tutti i valori speculativi sono sbagliati e possiamo rimediare alla situazione copiando gli ultimi valori che erano corretti nei valori speculativi.

9.9 Esempio sul funzionamento dell'architettura avanzata del processore

Partiamo da un frammento di programma C++

```
1 for(int i = 0; i<n; i++)
2         a[i]+=b[i];
```

trasformiamolo in Assembly:

```
 1 movl    n, %ebx
 2 movl    $0, %ebx
 3 ciclo:
 4 cmpl    %ebx, %ecx
 5 jge     fine
 6 movb    b(%ecx), %al
 7 movb    %al, a(%ecx)
 8 incl    %ecx
 9 jmp     cliclo
10 fine:
11 #...
```

Il linguaggio Assembly è quello che ci permette di avere una corrispondenza diretta con il linguaggio macchina. Abbiamo parlato di flusso delle istruzioni ma il programma scritto in Assembly non rappresenta il flusso delle istruzioni; eseguendolo, infatti, possiamo ottenere diversi flussi di esecuzione. Molte delle considerazioni che abbiamo fatto in passato non hanno senso in questo contesto: per esempio dire che l'istruzione alla riga 7 è successiva a quella alla riga 6 non ha senso perché siamo in un ciclo e se guardiamo alla singola iterazione l'affermazione è vera, ma se guardiamo tutte le iterazioni vediamo che l'istruzione alla riga 7 della prima iterazione è precedente all'istruzione alla riga 6 della seconda iterazione rendendo così l'affermazione falsa.

Quindi, supponendo n = 2, il "processo" di esecuzione sarà:

```
 1 movl    n, %ebx
 2 movl    $0, %ebx
 3 ciclo:
 4 cmpl    %ebx, %ecx
 5 jge     fine
 6 movb    b(%ecx), %al
 7 movb    %al, a(%ecx)
 8 incl    %ecx
 9 jmp     cliclo
10 cmpl    %ebx, %ecx
11 jge     fine
12 movb    b(%ecx), %al
13 movb    %al, a(%ecx)
14 incl    %ecx
15 jmp     cliclo
16 cmpl    %ebx, %ecx
```

```
17 jge      fine
18 fine:
19 #...
```

Queste sono le istruzioni che il processore effettivamente preleverà eseguendo quel programma; quindi nel flusso delle istruzioni compaiono più volte le istruzioni che si trovano all'interno del ciclo. Quindi l'ordine delle istruzioni dipende non dal programma visto prima ma dall'esecuzione del flusso delle istruzioni da parte del processore. Abbiamo inoltre visto che in realtà non è neanche questo che entra nella pipeline; queste istruzioni vengono tradotte in e-istruzioni. A questo livello abbiamo sia i registri classici architetturali (come `EBX`, `ECX`, `EDX` ecc..) sia quelli interni in più che il processore usa esclusivamente per la traduzione.

```
 1 LOAD     %ebx, n(R0)
 2 MOV      %ecx, $0
 3 SUB      R1, %ecx, %ebx
 4 JLE      fine, R1
 5 LOAD     %al, b(%ecx)
 6 LOAD     R1, a(%ecx)
 7 ADD      R1, R1, %al
 8 STORE    a(%ecx), R1
 9 ADD      %ecx, %ecx, $1
10 JMP      ciclo
11 SUB      R1, %ecx, %ebx
12 JLE      fine, R1
13 LOAD     %al, b(%ecx)
14 LOAD     R1, a(%ecx)
15 ADD      R1, R1, %al
16 STORE    a(%ecx), R1
17 ADD      %ecx, %ecx, $1
18 JMP      ciclo
19 SUB      R1, %ecx, %ebx
20 JLE      fine
```

N.B.: il processore traduce le istruzioni in e-istruzioni man mano che le preleva, ma non ha una visione completa di tutto il flusso di esecuzione.

A questo punto possiamo chiederci dove sono le dipendenze fra le istruzioni.

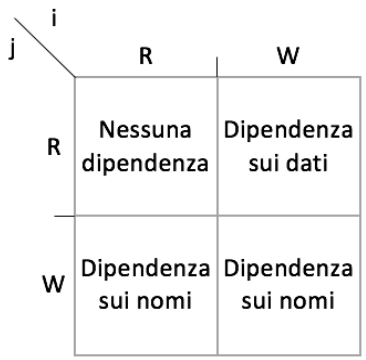


Figura 9.12: Schema delle dipendenze

dove la e-istruzione $j >$ della e-istruzione i.
Note sulle dipendenze:

- è sempre una e-istruzione successiva che dipende da una e-istruzione precedente;

- non importa che le due e-istruzioni siano adiacenti;
- una dipendenza, escludendo per il momento quelle sul controllo, esiste se e solo se due e-istruzioni fanno uso dello stesso registro.

Prese due e-istruzioni i e j con j successiva a i, la dipendenza è una cosa che impedisce di scambiare le due e-istruzioni. Nel flusso sequenziale di un processore che esegue un'istruzione per volta, la e-istruzione j verrebbe eseguita dopo la e-istruzione i. Con il processore in esame, la domanda che ci poniamo è: possiamo scambiare le due e-istruzioni senza alterare il senso del programma? Sì se fra loro non ci sono dipendenze.

Due istruzioni che leggono entrambe dallo stesso registro non si danno alcun fastidio e fra loro non esiste alcuna dipendenza.

La dipendenza sui nomi non nasce da quello che il programma deve fare poiché è conseguenza del riuso dei registri.

La memoria ha lo stesso identico problema dei registri; se vogliamo, uno stesso indirizzo di memoria è come se fosse uno stesso registro. Quindi le dipendenze fra le istruzioni si possono avere anche in memoria. Abbiamo però una complicazione in più rispetto ai registri infatti

```
LOAD    R1, a(%ecx)
#...
STORE   a(%ecx), R1
```

come facciamo a sapere l'indirizzo esatto? Non certo guardando semplicemente le e-istruzioni. Per sapere il valore dell'indirizzo occorre in parte eseguire l'istruzione.

In ogni caso, anche dal flusso delle istruzioni che viene fuori da una riga legittima di C++, si vede ad occhio che non è necessario fare tutte le cose con l'ordine prestabilito.

Il processore queste cose le fa in autonomia; in particolare potrebbe anche accorgersi che le istruzioni di diverse iterazioni dello stesso ciclo, può eseguirle anche parallelamente poiché lavorano su cose completamente diverse.

Per risolvere le dipendenze sul controllo, come già visto, il processore si serve del ROB:

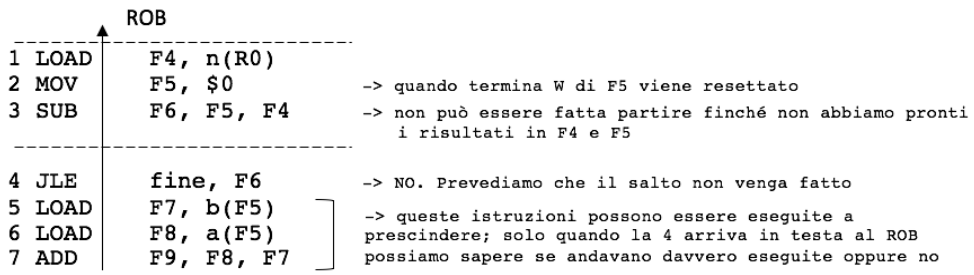


Figura 9.13: Contenuto del ROB relativo all'esempio

Se la previsione del salto era corretta, i valori non speculativi vengono aggiornati con gli attuali valori speculativi. Se la previsione del salto era sbagliata, e dunque le e-istruzioni non andavano in realtà eseguite, i valori speculativi vengono ripristinati con i corrispondenti valori non speculativi che sono gli ultimi che sicuramente andavano calcolati.

Il valore non speculativo relativo ad un registro logico, viene aggiornato quando l'istruzione esce dal ROB con il valore del registro nell'istruzione, non con il valore speculativo attuale.

La e-istruzione nella riga 3 della figura non può essere eseguita finché non ha pronti i risultati che deve utilizzare. Può, però, essere ugualmente emessa e accodata in una delle stazioni di prenotazione in attesa che i dati che le servono siano pronti. Questo ci consente di non fermarci.

I registri di destinazione li rinominiamo sempre anche quando non vi è alcun tipo di dipendenza.

Ad ogni rinomina e ad ogni istruzione aggiunta nel ROB, occorre aggiornare opportunamente la tabella Registri fisici-C-W-Cont.

Le istruzioni nel ROB possono essere eseguite in un qualunque ordine (tenendo conto delle dipendenze), una volta che hanno pronti i dati, ma devono essere ritirare dal ROB nell'ordine che rappresenta il flusso delle istruzioni voluto dal programma.

Quando un'istruzione esce dal ROB siamo sicuri che andava eseguita.

Non si smette mai di accodare le e-istruzioni nelle stazioni di prenotazione, lo facciamo solo quando non ci sono più risorse disponibili.

Vediamo adesso una struttura dati di cui si serve il processore:

Registri logici:

- **non speculativi**: ultimo valore buono calcolato;
- **speculativi**: valori correnti.

In questo caso supponiamo che le stazioni di prenotazione siano sufficientemente capienti da non causare alee strutturali.

Le uniche dipendenze che ci danno fastidio sono quelle sui dati perché quelle sui nomi le risolviamo andando a rinominare i registri di destinazione.

Approfondimento...

Come mai *gcc*, se traduciamo in Assembly, non fa le push ma usa altre istruzioni?

Prendiamo una funzione `f(a, b, c)`, noi in Assembly traduciamo:

```
1 push    c
2 push    b
3 push    a
4 call    f
5 #...
```

Così facendo abbiamo una dipendenza sui dati, le `push` non possono essere invertite perché implicitamente decrementano `ESP` e vanno fatte esattamente in questo ordine per ottenere l'ordine giusto degli argomenti attuali nel record di attivazione.

`gcc` invece fa:

```
1 sub     $12, %esp
2 movl    a, (%esp)
3 movl    b, 4(%esp)
4 movl    c, 8(%esp)
5 call    f
6 #...
```

Le `mov` hanno tutte una dipendenza con la prima istruzione, ma, fra loro, possono essere eseguite in parallelo dal processore.

Appendice A

Strutture notevoli

A.1 Gate di interruzione

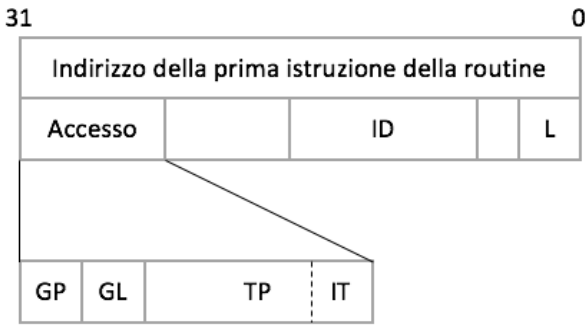


Figura A.1: Struttura di un gate di interruzione

A.2 Descrittore di pagina fisica

```
1 enum tt{LIBERA, DIRETTORIO, TABELLA_CONDIVISA, TABELLA_PRIVATA,
      PAGINA_CONDIVISA, PAGINA_PRIVATA};
2
3 struct des_pf{
4         tt contenuto; //qual e' il contenuto della pagina
5         union{
6                 struct{
7                         bool residente; //pagina residente o meno
8                         natl processo; //identificatore di processo
                              non significativo per le pagine condivise
9                         natl ind_massa; //indirizzo della pagina in
                              memoria di massa
10                        addr virtuale; //indirizzo virtuale di una
                              riga della pagina (se tt = PAGINA e sono
                              significativi i primi 20 bit) oppure
                              indirizzo virtuale di una pagina (se tt =
                              TABELLA e sono significativi i primi 10
                              bit).
11                        natl contatore; //contatore per le statistiche
12                } pt;
13                struct{ //questo campo ci interessa solo se tt =
                      LIBERA
```

```
14                          des_pf* prossima_libera; //puntatore al
                                descrittore della prossima pagina libera
15                  } avl;
16          };
17 };
18 //i seguenti tre oggetti fanno parte della sezione .data del modulo
       sistema in M1
19
20 des_pf dpf[N_DPF]; //array di descrittori di pagina fisica
21 addr prima_pf_utile; //indirizzo fisico della prima pagina fisica di
       M2
22 des_pf* pagine_libere; //puntatore alla lista di descrittori delle
       pagine fisiche libere
```

A.3 Descrittore di pagina virtuale e di tabella delle pagine

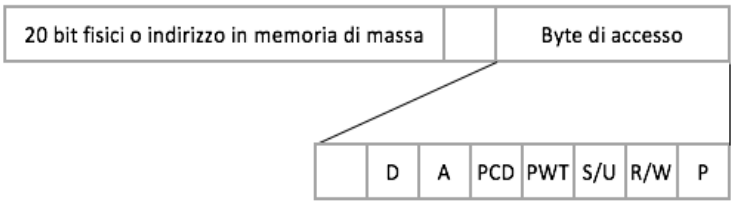


Figura A.2: Struttura di descrittore di pagina virtuale e di tabella delle pagine

A.4 Descrittore di processo

```
1 //sistema.cpp
2 struct des_proc{
3          natl id; //identificatore del processo
4          addr punt_nucleo; //valore iniziale del puntatore alla pila
               sistema; significativo solo se il processo e' utente
5          natl riservato;
6          addr CR3; //campo per il registro speciale CR3 che contiene l'
               indirizzo fisico del descrittore di processo
7          natl contesto[N_REG]; //campo destinato a contenere la copia
               del contenuto dei registri generali del processore. Questo
                campo ha dimensione N_REG
8          natl cpl; //livello di privilegio del processo
9 };
```

A.5 Struttura *proc_elem* per l'indentificazione di un processo

```
1 //sistema.cpp
2 struct proc_elem{
3          natl id; //identificatore del processo
4          natl precedenza; //priorita' del processo
5          proc_elem* puntatore; //puntatore che serve per mantenere una
               lista di tipo proc_elem
6 };
```

```
7
8 extern proc_elem* esecuzione; //puntatore al processo in esecuzione
9 extern proc_elem* pronti; //puntatore alla lista dei processi pronti
```

A.6 Descrittore di I/O

```
1 struct interf_reg{
2         union{ioaddr iRBR, iTBR;} in_out;
3         union{ioaddr iCTRI, iCTRO;} ctr_io;
4 };
5
6 struct des_io{
7         natb* buff; //puntatore al buffer
8         natl n; //numero di byte coinvolti nell'operazione
9         natl mutex; //semaforo di mutua esclusione
10        natl sync; //semaforo di sincronizzazione
11        interf_reg indreg; //indirizzi dei registri della periferica
12 };
```

A.7 Descrittore di semaforo

```
1 //sistema.cpp
2 struct des_sem{
3         int counter;
4         proc_elem* pointer;
5 };
```

A.8 Descrittore di buffer (bus mastering)

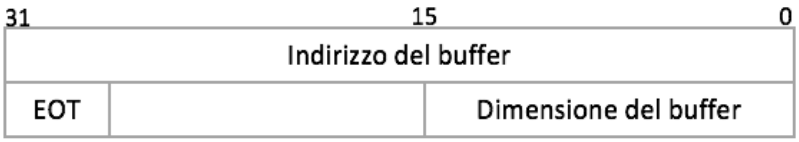


Figura A.3: Descrittore di buffer

Appendice B

Domande per argomento

Q. 1 - Che differenza c'è tra un programma e un processo?[1]

Proviamo ad illustrare la differenza con una semplice metafora: in una pizzeria, il pizzaiolo riceve una ordinazione. Il pizzaiolo è inesperto e ha bisogno di avere la ricetta della pizza davanti a sé. Stende la pasta, la condisce, la inforna, aspetta che sia cotta, la farcisce e serve la pizza. Cos'è il programma? Sono sicuro che nessuno di voi abbia dubbi: il programma è la ricetta. Cos'è il processo? Qui credo che molti risponderanno: il pizzaiolo. No. Il pizzaiolo è il processore, cioè l'entità che interpreta la ricetta e la esegue. Allora è la pizza? No. La pizza sono i dati da elaborare. Il processo è un concetto un po' più astratto. È la sequenza di stati che il sistema "pizza + pizzaiolo" attraversa, passando dalla pasta alla pizza finale, secondo le istruzioni dettate dalla ricetta, eseguite pedissequamente dal pizzaiolo inesperto. Uscendo dalla metafora, un processo è un programma in esecuzione su dei dati di ingresso. Questa esecuzione la possiamo modellare come la sequenza degli stati attraverso cui il sistema processore + memoria passa eseguendo il programma, su quei dati, dall'inizio fino alla conclusione. Notate che questa definizione si applica bene ai programmi di tipo batch, in cui gli ingressi vengono specificati tutti all'inizio, e il processo (generato dal programma in esecuzione su quei dati) prosegue indisturbato fino ad ottenere le uscite (ad esempio, pensate ad un programma per ordinare alfabeticamente un file). Una volta afferrato il concetto, però, in questo caso semplice, credo che non vi sarà difficile estenderlo ai programmi "interattivi" a cui ormai siete più abituati (praticamente tutti i programmi con interfaccia grafica, ma non solo quelli). A prima vista, il processo potrebbe sembrare molto simile al programma: la ricetta dice "stendere la pasta" e nel processo vediamo il pizzaiolo stendere la pasta; nel rigo successivo la ricetta ricetta dice "versare il condimento" e nel processo vediamo, subito dopo, il pizzaiolo versare il condimento. È molto semplice confondere i due concetti, soprattutto se il programma è molto semplice, ma si tratta di due cose completamente distinte, per i seguenti motivi:

- uno stesso programma può essere associato a più processi: tanti clienti, in genere, chiedono lo stesso tipo di pizza. In questo caso, il programma è sempre lo stesso (la ricetta per quel tipo di pizza), ma ad ogni pizza corrisponde un processo distinto, che si svolge autonomamente nel tempo.

[1] la risposta a questa domanda è tratta da una dispensa del professor Lettieri

- in generale, non è esclusivamente il programma (la ricetta) a decidere attraverso quali stati il processo dovrà passare, ma anche la richiesta del cliente (l'input): la ricetta potrebbe, infatti, prevedere delle varianti (un if), che il cliente dovrà specificare. La ricetta conterrà istruzioni per entrambe le varianti (con o senza carciofi), ma un particolare processo seguirà necessariamente *una sola* variante.

Ma c'è dell'altro, nella metafora del pizzaiolo, che ci può aiutare a capire altri punti fondamentali. Il processo, se lo guardiamo nella sua interezza, si svolge necessariamente nel tempo. Possiamo anche, però, guardarlo ad un certo istante, facendone una fotografia. La fotografia che ne facciamo deve contenere tutte le informazioni necessarie a capire come il processo si svolgerà nel seguito. Nel nostro esempio, la foto dovrà contenere:

- la pizza nel suo stato semilavorato (la memoria dati del processo);
- il punto, sulla ricetta, a cui il pizzaiolo è arrivato (il contatore di programma);
- tutte le altre informazioni che permettono al pizzaiolo di proseguire (da quel punto in poi) con la corretta esecuzione della ricetta, come, ad esempio, il tempo trascorso da quando ha infornato la pizza (i registri del processore).

Se la fotografia è fatta bene, contiene tutto il necessario per sospendere il processo e riprenderlo in un secondo tempo. Il pizzaiolo fa questa operazione continuamente, quando condisce, a turno, più pizze, o quando lascia una serie di pizze nel forno e, nel frattempo, comincia a prepararne un'altra (multiprogrammazione).

Q. 2 - In che senso la memoria fisica è mappata in memoria virtuale? Perché il bit P deve essere sempre uguale a 1?[2]

Probabilmente il modo in cui ciò avviene vi sfugge perché siete troppo legati alle astrazioni (pagine che si spostano, etc.). Volendo restare nell'astratto, potete immaginare che, all'interno della memoria virtuale, ci sia una finestra (grande quanto la memoria fisica) che permette di accedere alla memoria fisica, indipendentemente da ciò che essa (memoria fisica) contiene. Questa finestra la collochiamo a partire dall'indirizzo (virtuale) 0 in modo che, per leggere (o scrivere) alla locazione di memoria fisica x, basta leggere (o scrivere) all'indirizzo virtuale x. Se la memoria fisica è, ad esempio, di 512 MiB, questa finestra finisce all'indirizzo virtuale 2^{29} - 1. Questa finestra non viene gestita come la normale memoria virtuale: è solo un modo (normalmente riservato al codice di sistema) per accedere a tutta la memoria fisica. Possiamo pensare che la normale memoria virtuale inizi subito dopo la fine di questa finestra (quindi, nell'esempio di prima, dall'indirizzo 2^{29} fino all'indirizzo 2^{32} - 1). Come viene creata questa finestra? Qui conviene che abbandoniate le astrazioni e pensiate semplicemente a cosa sono (collettivamente) il direttorio più le tabelle delle pagine: una tabella che trasforma ogni indirizzo generato dal processore, prima che questo indirizzo arrivi alla memoria (quella fisica, l'unica che esiste davvero). Il bit P, che è associato ad ogni "pagina" di indirizzi virtuali (una pagina, in questo contesto, è solo un insieme contiguo di indirizzi virtuali) aggiunge un meccanismo

[2] la risposta a questa domanda è tratta da una dispensa del professor Lettieri

particolare alla trasformazione: se, per un certo indirizzo, P=0, la MMU non effettua la trasformazione ma, invece, solleva una eccezione. Questa trasformazione può essere usata per vari scopi, la memoria virtuale è solo uno di questi (anche se è il più significativo). Nella memoria virtuale il bit P ci serve perchè non tutti gli indirizzi sono sempre validi (appunto, sono virtuali). La nostra finestra, invece, è una applicazione particolare di questa trasformazione: in pratica, una trasformazione "identità", che lascia gli indirizzi così come erano. Per questi indirizzi il bit P non ci serve. Lo poniamo sempre a 1 perchè non vogliamo che vengano generate eccezioni quando il processore accede agli indirizzi dentro questa finestra. A questo punto, vi resta da capire che queste due trasformazioni "convivono" nella stessa tabella: nelle prime entrate ci sono le trasformazioni (identità) relative alla finestra, con bit P sempre = 1, nelle rimanenti le trasformazioni relative alla memoria virtuale, alcune con bit P=0, altre con bit P=1, a seconda di quali pagine della memoria virtuale sono "presenti in RAM". Notate che una pagina virtuale di indirizzo virtuale V, presente in RAM nella pagina di memoria fisica di indirizzo fisico F, sarà accessibile al processore da due indirizzi virtuali: V (tramite la memoria virtuale) e F (tramite la finestra).

Q. 3 Memoria cache

- A cosa serve la memoria cache?
- Come funziona la memoria cache?
- Aumento/diminuzione del campo indice e relative conseguenze. Qual è la dimensione ragionevole per un blocco?
- Cache associativa a insiemi.
- Quali problemi crea la presenza della cache nella scrittura e nella lettura in DMA?
- Come possiamo risolvere i problemi che crea la presenza contemporanea della cache e del DMA (a parte disabilitare la cache o usare il write through)? [Occorre invalidare la cache prima del trasferimento DMA, almeno la parte che contiene indirizzi interessati dal trasferimento.]
- Perché vogliamo disabilitare la cache per alcune pagine?

Q. 4 Memoria virtuale

- Concetto di virtualizzazione della memoria, memoria virtuale e suo funzionamento generico.

Indirizzi virtuali, pagine, tabelle delle pagine, direttorio

- Traduzione di un indirizzo virtuale in indirizzo fisico tramite la tabella di corrispondenza.
- Disegnare e spiegare il meccanismo di traduzione di un indirizzo virtuale in fisico.

- Quanti descrittori delle tabelle delle pagine ha il direttorio?
- Quanto è grande un descrittore?
- Quanto è grande il direttorio?
- Spiegare lo scopo del bit D nel descrittore di pagina virtuale.

Descrittore di pagina fisica e routine di page-fault

- Come è fatto un descrittore di pagina fisica?
- Quali e cosa significano i campi contenuti nel descrittore di pagina fisica?
- Cosa contiene una pagina fisica? Cosa può contenere?
- Differenze tra tabella_condivisa e tabella_privata.
- Perché si hanno i descrittori di pagina fisica anche per quelle entità che devono essere residenti?
- Esiste un descrittore di pagina fisica per ogni pagina fisica?
- Perché in M2 sono presenti anche i direttori?
- A chi serve sapere che una pagina fisica contiene il direttorio?
- A cosa serve il campo *ind_virtuale*? Quale indirizzo contiene?
- A che cosa serve il campo *ind_massa*? Cosa contiene?
- Cosa fa la routine di page-fault? Come usa il campo *ind_virtuale* del descrittore di processo?
- Come avviene il *rimpiazzamento*?
- Perché si può usare il contenuto di CR3 come se fosse un indirizzo virtuale e non fisico?
- Quali problemi si possono avere con le pagine condivise? Quali limitazioni? Come si provvede?
- Cosa succede se viene scelta come vittima una tabella delle pagine mentre una delle pagine che riferisce è sempre in memoria? Succede mai? Perché non succede mai?
- In quali casi potrebbe essere scelta come vittima una tabella delle pagine invece di una pagina se non venissero presi provvedimenti? Quali sono i provvedimenti che vengono presi?
- Se si hanno due pagine fisiche che hanno i contatori uguali e una contiene una pagina virtuale mentre un'altra una tabella delle pagine, si sceglie la tabella o la pagina? Perché?
- Con l'ausilio di quali bit presenti nel descrittore di pagina virtuale, facciamo le statistiche per il rimpiazzamento della pagina? Quali sono i due algoritmi con cui vengono effettuate queste statistiche?

TLB

- Parlare del TLB
- Perché la routine di page-fault non può accedere al TLB?
- Quando occorre rimpiazzare il contenuto di una pagina, occorre invalidare il TLB. Questa cosa ha senso anche quando la pagina virtuale che andiamo a rimuovere appartiene ad un processo che non è attualmente in esecuzione? [No, perché il TLB si è già "dimenticato" di quella traduzione poiché, essendo sicuramente avvenuto un cambio di contesto (cambio del valore di `CR3`), il TLB sarà stato svuotato di tutto il suo contenuto]

Q. 5 Meccanismo delle interruzioni, meccanismo di protezione, primitive, driver, processi esterni

- Cosa accade al processore quando riceve un'interruzione o un'eccezione?
- Quali sono le azioni compiute dal processore a seguito di una richiesta di interruzione? Specificare cosa succede per ogni tipo di interruzione.
- A che cosa serve il tipo dell'interruzione?
- Cosa è la tabella IDT, come è fatta e quali informazioni sono contenute nei gate?
- Qual è lo scopo di un gate?
- Quale informazione relativa al meccanismo di protezione è contenuto in un gate?
- Come è fatto un gate di interruzione? Quanto è grande? Cosa c'è nel byte di accesso?
- Come è possibile fare un cambio di livello di privilegio per mezzo delle interruzioni (quale campo del gate viene usato)?
- Quali azioni vengono svolte dal processore prima di eseguire la prima istruzione della routine?
- Spiegare l'istruzione INT $tipo.
- Da dove nascono le eccezioni?
- Qual è il livello di privilegio delle primitive che vengono messe in esecuzione dalle eccezioni? Perché?
- Meccanismo dell'interruzione fino alla chiamata di un handler.
- Quando viene scelto un processo da eseguire come siamo sicuri che non è un processo esterno?
- Cosa sono, dove sono e a cosa servono i parametri fittizi in una c_primitiva?
- Struttura generica di una primitiva Assembly/C++, struttura del sottoprogramma di interfaccia.

Q. 6 Multiprogrammazione e nucleo multiprogrammato

Processi, attivazione di processi, descrittori di processo, stato delle pile, commutazione di contesto, priorità

- Cosa è un processo?
- Cosa è un processore virtuale?
- Struttura `proc_elem`.
- Come si fa a creare un processo?
- Cosa fa la `activate_p` per attivare un processo?
- Qual è lo stato delle pile utente e sistema (per un processo utente) dopo l'`activate_p`?
- Inizializzazione di un descrittore di processo.
- Come è fatto un descrittore di processo (`des_proc`)?
- Come si individua un descrittore di processo?
- Cosa è la GDT?
- Le pile dei processi devono essere preparate in un modo particolare? Perché?
- Come viene messo in esecuzione un processo così creato?
- Descrizione dettagliata della `salva_stato` e della `carica_stato`.
- `carica_stato`: cosa fa, quando viene chiamata e chi va in esecuzione dopo la chiamata?
- Qual è la primissima istruzione che viene eseguita da un processo dopo la carica_stato?
- Quando un processo può finire davanti ad un altro processo in lista pronti?
- Come può avvenire una commutazione di contesto?
- Cosa succede quando avviene una commutazione di contesto?

Semafori, mutua esclusione, sicronizzazione, atomicità

- Primitive `sem_wait` e `sem_signal`.
- Concetto di atomicità.
- Che differenza c'è fra mutua esclusione e sincronizzazione?

Q. 7 Operazioni di I/O, processi esterni

- A cosa servono i descrittori di I/O (*des_io*)?
- Descrivere un driver.
- Come si attiva un processo esterno?
- Cosa è la funzione f passata come argomento?
- Per quale motivo la dimensione dell'`array_p` è quella?
- Perché il puntatore a *des_io* è inizializzato fuori dal for?
- Quanti sono i processi esterni?
- Com'è strutturato un handler?
- L'handler conosce il parametro i. Perché?
- A chi appartiene la pila utilizzata da un handler?
- Come mai i processi esterni sono ciclici?
- Com'è inizialmente la pila dei processi esterni?
- Come mai i driver **non devono** essere ciclici?
- Com'è fatto un processo esterno per lettura dati?
- Quando si intende attivare un processo esterno, quale primitiva viene chiamata? Quali parametri ha?
- Che cosa succede con la `activate_pe`?
- Cosa è e cosa fa la `wfi()`?
- Come è fatta la `wfi()`? Quale processo va in esecuzione dopo di essa quando viene lanciato per la prima volta il processo esterno e le volte successive?
- Come inizializza la pila la `wfi()`?
- Quando viene scelto un processo da eseguire, come siamo sicuri che non sia un processo esterno?
- Qual è il compito di un processo esterno?
- Un processo esterno può essere interrotto da una `sem_signal` che ne risveglia un altro?
- Quali sono i casi in cui si potrebbero generare problemi nell'interruzione di un processo esterno ma che invece sono ininfluenti?
- Che ruolo ricopre il semaforo `sync` nei processi esterni?
- Che priorità ha un processo esterno?
- Su quale pila agisce la `IRET` nell'ambito di un processo esterno?

Q. 8 Bus PCI, DMA, bus mastering

- Schema generale di PCI e DMA.
- Il bus PCI viene interessato dal trasferimento in DMA?
- Come è montata l'interfaccia ATA nello schema PCI/DMA?
- In generale, con un'interfaccia generica in bus mastering, come facciamo?
- Mentre avviene il trasferimento, il bus è quasi sempre occupato dal processore. Come fa ad avvenire il trasferimento se il bus è quasi sempre occupato? A chi viene data la priorità?

Q. 9 Architettura interna del processore

- Cosa è una pipeline?
- Quali sono i tipi di alee che si possono verificare? Quando? Come si possono risolvere?

Bibliografia

[**1.**] G. Frosini, G. Lettieri, "Architettura dei Calcolatori, Vol. II – Struttura Hardware del Processore, dei Bus, della Memoria e delle Interfacce, e gestione dell'I/O", Pisa University Press, 2013.

[**2.**] G. Frosini, G. Lettieri, "Architettura dei Calcolatori, Vol. III – Aspetti Architetturali Avanzati e Nucleo di Sistema Operativo", Pisa University Press, 2013.

[**3.**] Appunti personali delle lezioni.

Per segnalare errori, imprecisioni oppure per semplici suggerimenti, mandate una e-mail a dispensa.calcolatori.inginf@gmail.com. Grazie!