# Memoria virtuale (terza parte)

G. Lettieri

4 Aprile 2017

## 1 $MMU_3$: memoria fisica in memoria virtuale

Il nostro software ha i seguenti requisiti, per quanto riguarda l'accesso alla memoria:

1. il software di livello utente deve poter accedere soltanto alle parti di memoria fisica che contengono le sue pagine virtuali.
2. il software di livello sistema (routine di page fault) deve poter accedere a *tutta* la memoria fisica; [1]

Per realizzare il requisito 1 abbiamo finora sfruttato i seguenti fatti:

- la MMU (in qualunque versione) realizza una funzione di traduzione dagli indirizzi virtuali, generati dalla CPU, agli indirizzi fisici usati per accedere alla memoria e ai dispositivi. La funzione di traduzione è codificata nella tabella di corrispondenza (che per $MMU_2$ è data dalla sequenza delle tabelle di livello 1).
- se la MMU è attiva un programma non può accedere a nessuna parte della memoria fisica che non si trovi nel codominio di questa funzione;
- la MMU è sempre attiva mentre è in esecuzione il programma utente;

Quindi è sufficiente che il codominio della funzione di traduzione contenga solo le pagine fisiche che contengono le pagine virtali del programma utente. Per come

[1] Si noti che abbiamo evidenziato "tutta" la memoria fisica nel caso della routine di page fault. Infatti, tale routine:

- deve poter accedere a $M_1$, che contiene il suo codice e i descrittori di pagina fisica;
- deve poter accedere anche a tutte le pagine fisiche di $M_2$ che contengono tabelle (deve infatti poterle consultare e modificare);
- forse in modo meno ovvio, deve poter accedere anche alle rimanenti pagine fisiche di $M_2$, quelle che o sono libere o contengono pagine virtuali del programma utente: questo perché deve trasferire le pagine virtuali tra l'area di swap e la memoria fisica e (almeno per ora) non conosciamo altro modo per farlo se non leggendo dal registro BR dell'hard disk e scrivendo in memoria, o viceversa.

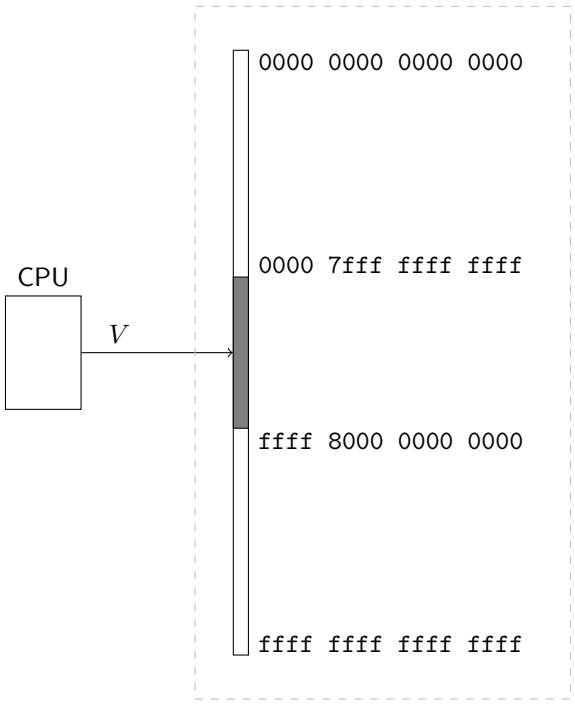


Figura 1: Macchina virtuale realizzata con Super-MMU, $MMU_1$ e $MMU_2$.

abbiamo realizzato la memoria virtuale, questa proprietà è automaticamente soddisfatta: all'inizio il codominio è vuoto e la routine di page fault aggiunge solo traduzioni relative a pagine virtuali del processo utente.

Per realizzare il requisito 2 abbiamo fatto in modo che $MMU_1$ e $MMU_2$ si disattivassero automaticamente mentre è in esecuzione la routine di page fault. Se la MMU non si disattivasse, anche la routine di page fault potrebbe accedere soltanto al codominio della funzione di traduzione che, per quanto abbiamo appena detto, contiene solo pagine virtuali del processo utente e non tutte le altre parti della memoria fisica, a cui invece la routine di page fault ha bisogno di accedere.

Questa proprietà di $MMU_1$ e $MMU_2$ è però soltanto una semplificazione. La $MMU_3$, che esaminiamo ora e che è la più vicina alla MMU che troviamo realmente nei sistemi Intel/AMD, resta invece sempre attiva, anche mentre è in esecuzione la routine di page fault. Dobbiamo quindi trovare un altro modo per permettere a tale routine l'accesso a tutta la memoria fisica, e allo stesso tempo continuare a negare questo accesso al programma utente. Poichè resta vero che, con la MMU attiva, si può accedere soltanto al codominio della funzione di traduzione, è chiaro che dobbiamo portare nel codominio anche $M_1$ e tutta $M_2$. Questo vuol dire, però, che ci devono essere degli indirizzi virtuali che permettono di raggiungere queste parti di memoria fisica, e questi indiriz-

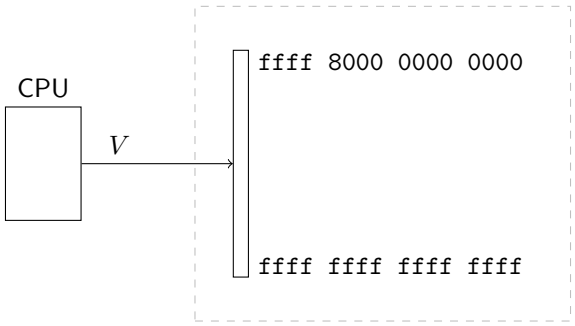


Figura 2: Macchina virtuale realizzata con $MMU_3$.

zi virtuali devono essere utilizzabili solo dalla routine di page fault e non dal programma utente. Fino a questo punto tutti i $2^{48}$ indirizzi virtuali possibili erano a completa disposizione del programma utente (Fig. 1), ma ora dobbiamo riservarne alcuni per le esigenze del sistema. Scegliamo di dividere gli indirizzi a metà fra utente e sistema: riserviamo al sistema tutti gli indirizzi che hanno 0000 nei 16 bit più significativi e lasciamo all'utente tutti gli altri. Ora la tabella di corrispondenza conterrà sia le traduzioni dell'utente, sia le traduzioni usate dal sistema. In particolare, le prime 256 entrate della tabella di livello 4 saranno per il sistema e le altre 256 per l'utente. Se non prendiamo contromisure, il programma utente potrebbe accedere alle parti di memoria fisica del sistema semplicemente generando degli indirizzi che cadono nella prima metà della memoria virtuale, violando così il requisito 1. Per evitarlo, però, è sufficiente proteggere le prime 256 entrate della tabella di livello 4 con il bit CPL. Si noti che, una volta applicata questa protezione, la macchina virtuale che l'utente vede è quella di Fig. 2, nella cui memoria manca tutta la parte superiore che c'era in Fig. 1.

Come usiamo la parte di tabella di corrispondenza riservata al sistema? La cosa più semplice è di emulare il comportamento di $MMU_1$ e $MMU_2$. Queste MMU si disattivavano quando era in esecuzione la routine di page fault. Anche se $MMU_3$ non si disattiva possiamo creare una traduzione che non abbia alcun effetto, che di fatto è equivalente a disattivarla. Creiamo quindi una traduzione "identità" che lasci inalterati gli indirizzi prodotti dal processore (li traduca in sé stessi) e non causi mai page fault (basta porre tutti i bit P ad 1). Poichè gli indirizzi fisici partono da 0 e abbiamo riservato al sistema proprio gli indirizzi virtuali che partono da 0, questo permette alla routine di page fault di continuare ad usare indirizzi fisici proprio come se la MMU fosse disattivata. È come se nella parte alta dello spazio di indirizzamento complessivo di Fig. 1 avessimo creato una finestra che permette di vedere la memoria fisica così come è.

Questa finestra deve essere creata prima di attivare la memoria virtuale. All'avvio del sistema, il processore parte con la memoria virtuale disattivata ed esegue la routine di inizializzazione. Questa può allocare e inizializzare tutte le tabelle necessarie alla definizione della finestra e poi attivare la memoria virtuale.

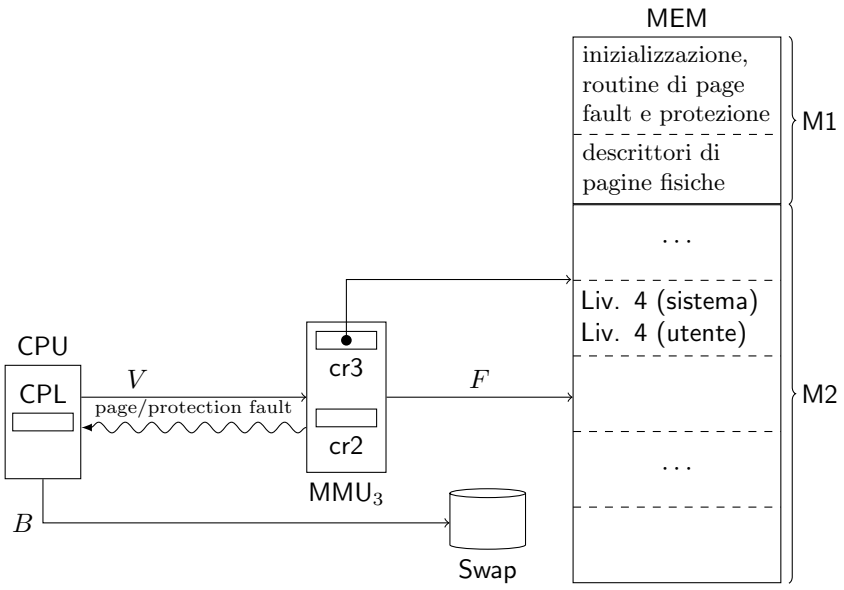


Figura 3: Implementazione della macchina virtuale di Fig. 2 con $MMU_3$.

A questo punto la routine di inizializzazione può continuare ad utilizzare indirizzi fisici come stava facendo prima dell'attivazione, e il resto dell'inizializzazione e dell'esecuzione del programma e della routine di page fault può procedere esattamente come avevamo visto nel caso di $MMU_1$ e $MMU_2$.

Per completezza mostriamo lo schema finale del sistema in Fig. 3. Rispetto al caso di $MMU_2$ manca il segnale di fine fault (inutile, in quanto $MMU_3$ non ha bisogno di sapere quando deve riattivarsi) e la tabella di liv. 4 è divisa in una parte sistema e una parte utente.

Ci restano da affrontare due problemi:

1. come facciamo con lo spazio richiesto per tutte le tabelle necessarie (Par. 2)?
2. possiamo accettare il tempo richiesto per la traduzione degli indirizzi (Par. 3)?

# 2 Pagine di grandi dimensioni

Spezzando la tabella di corrispondenza di $MMU_1$ nella tabella multilivello di $MMU_2$ abbiamo risolto il problema dello spazio richiesto per le tabelle in memoria fisica, ma la dimensione complessiva delle tabelle è tale da crearci problemi anche per lo spazio richiesto nell'area di swap. Il problema più grande è dato dalle tabelle di livello 1, che complessivamente richiedono 512 GiB di spazio. Per affrontare questo problema (sentra addentrarci nelle soluzioni che richiedono nozioni sui sistemi operativi) abbiamo sostanzialmente due possibilità.

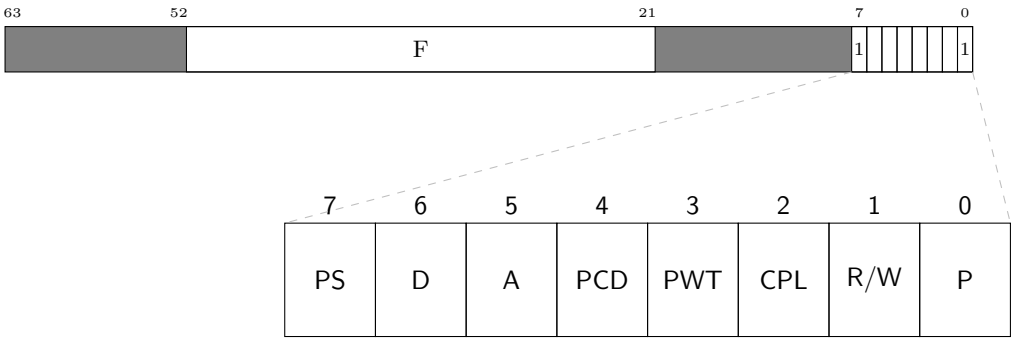


Figura 4: Descrittore di pagina virtuale da 2 MiB (tabelle di livello 2).

- Una si applica soltanto ai programmi piccoli, che però sono la maggior parte: evitiamo di allocare le tabelle e le pagine di cui il programma non ha bisogno, dal momento che non vi accederà mai. Come misura per contrastare gli errori, possiamo inserire un valore speciale nei descrittori delle tabelle e delle pagine che non abbiamo allocato in modo che, se il programma dovesse tentare di accedervi per errore, la routine di page fault possa riconoscere la situazione e terminare il programma.
- L'altra soluzione è quella di usare pagine più grandi di 4 KiB, riducendo di conseguenza il numero di tabelle richieste.

L'archietettura Intel/AMD a 64 bit ci permette di avere pagine più grandi di 4 KiB usando il bit PS nei descrittori di livello 3 e 2. Anche questo bit è scritto dalle routine di inizializzazione e page fault, e soltanto letto dalla MMU. La Fig. 4 mostra il formato del descrittore di livello 2 nel caso in cui il bit PS vale 1. Si vede che il descrittore contiene ora un campo F che va dai bit 21 a 51, invece del puntatore alla tabella di livello 1. La Fig. 5 mostra la traduzione da indirizzo virtuale a fisico eseguita in questo caso: quando la MMU arriva alla tabella di livello 2 e trova il bit PS a 1, usa come offset tutti i bit da 0 a 20 e li concatena al campo F trovato nel descrittore. In questo modo abbiamo eliminato una tabella di livello 1, risparmiando il relativo spazio. Il meccanismo è compatibile e può convivere con le pagine di 4 KiB: la MMU si comporta come sempre nei livelli 4 e 3 e scopre di dover eseguire il nuovo tipo di traduzione solo quando arriva al livello 2 e trova il bit PS pari a 1; se lo avesse trovato a 0 avrebbe proseguito fino al livello 1, come sempre. Ogni descrittore di livello 2 può avere il bit PS a 1 o a 0 indipendentemente dagli altri, quindi per lo stesso programma possiamo usare sia pagine di 2 MiB, sia pagine di 4 KiB, a seconda della convenienza.

Le pagine da 2 MiB ci permettono di risparmiare molto spazio per le tabelle, ma hanno ovviamente un costo: per caricarle dobbiamo trovare in memoria fisica uno spazio contiguo di 2 MiB invece che di soli 4 KiB, e ogni trasferimento da e verso l'area di swap richiede ora più tempo. Si noti che quest'ultima cosa

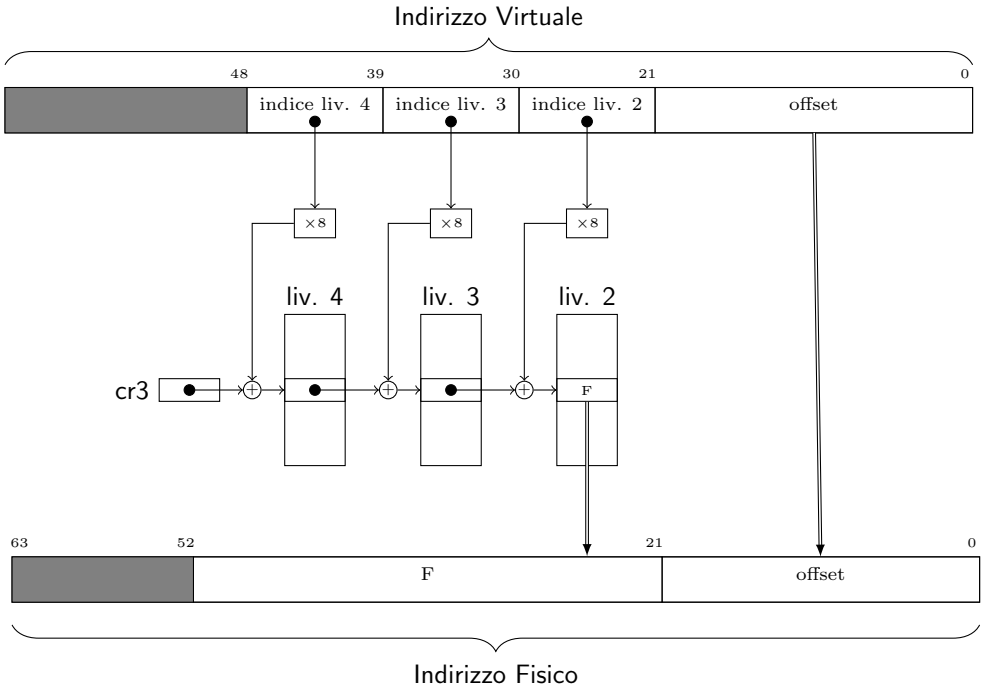


Figura 5: Traduzione da indirizzo virtuale a fisico (pagine di 2 MiB).

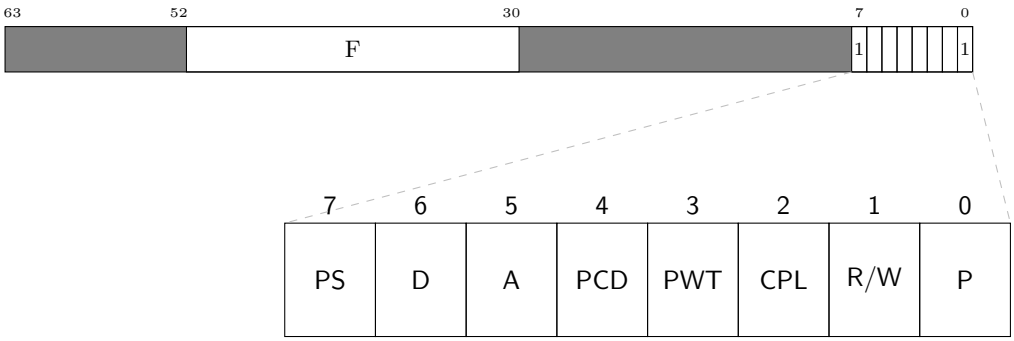


Figura 6: Descrittore di pagina virtuale da 1 GiB (tabelle di livello 3).

potrebbe non essere un problema: se il programma aveva comunque bisogno di accedere a tutto quel contenuto, caricarlo tutto in una volta può essere molto più efficiente che fare 512 trasferimenti separati.

I processori più recenti permettono di avere PS=1 anche nei descrittori di livello 3. La Fig. 6 mostra il formato del descrittore di livello 3 in questo caso, in cui si vede che il campo F sostituisce anche qui il campo che punta alla tabella di livello 2. La Fig. 7 mostra la traduzione eseguita dalla MMU in questo caso. Si vede come l'offset è ora su 30 bit, e dunque le pagine sono ora grandi 1 GiB. Questo tipo di traduzione è molto utile per creare la finestra sulla memoria fisica (si veda la sezione precedente), in quanto in quel caso tutto quello che ci interessa è soltanto la traduzione, per di più da una zona contigua della memoria virtuale a tutta la memoria fisica, e non dobbiamo caricare niente dall'area di swap.

# 3 Il TLB

Per ogni accesso in memoria la MMU deve prima consultare fino a 4 tabelle per poter eseguire la traduzione. Se consideriamo che il nostro programma deve accedere continuamente in memoria, sia per prelevare le sue stesse istruzioni, sia per prelevare o scrivere gli operandi che si trovano in memoria, ci rendiamo subito conto che l'impatto della MMU sul tempo di esecuzione di un programma può essere molto grande. È vero che subito dopo la MMU c'è la cache, e quindi possiamo sperare che molti di questi accessi non debbano realmente arrivare fino alla memoria, ma resta il fatto che, anche nel migliore dei casi, quello che prima era un unico accesso in cache si è ora trasformato in una sequenza di 5 accessi in cache.

Per affrontare questo problema si introduce una nuova cache, chiamata TLB (Translation Lookaside Buffer), che è specifica per la MMU. Lo scopo di questa cache è di ricordare le *traduzioni* utilizzate più recentemente, dove per traduzioni intendiamo, per le pagine di 4 KiB, quelle definite dai descrittori di livello 1, per le pagine di 2 MiB quelle nei descrittori di livello 2 e, infine, quelle dei

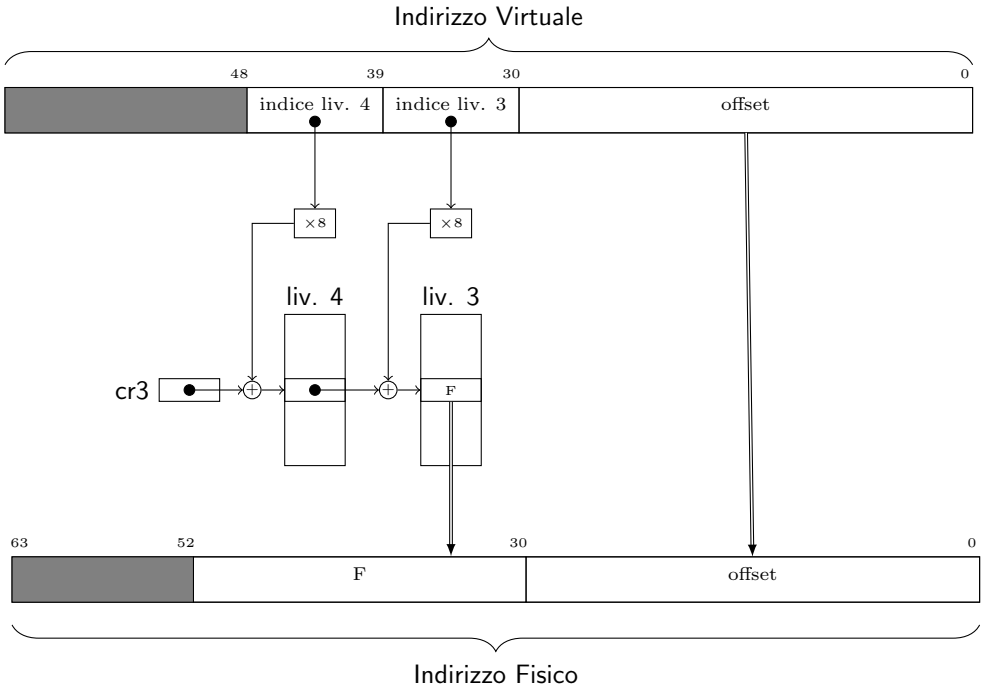


Figura 7: Traduzione da indirizzo virtuale a fisico (pagine di 1 GiB).

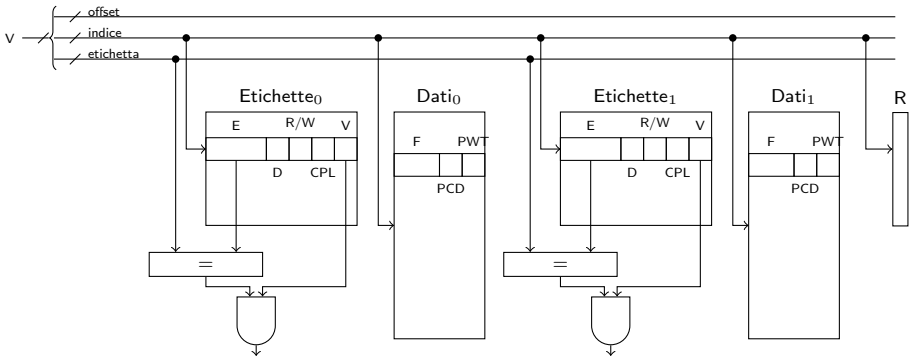


Figura 8: Un esempio di TLB a 2 vie.

descrittori di livello 3 per le pagine da 1 GiB. In ogni caso, per ognuna di queste traduzioni, non ci interessano i descrittori di livello più alto, il cui unico scopo è di permettere di raggiungere, dato l'indirizzo virtuale da tradurre, il descrittore che si trova in fondo alla catena. Una volta raggiunto questo descrittore la MMU può ricordare (nel TLB) quale sia la traduzione da fare per l'indirizzo virtuale in questione: non ha bisogno di ricordare tutto il percorso.

Limitiamoci, per semplicità, a parlare soltanto delle pagine di 4 KiB. In questo caso il TLB è una cache dei descrittori di livello 1. Quando la MMU deve tradurre un indirizzo controlla prima se il TLB contiene già il descrittore che sta cercando; altrimenti si comporta come abbiamo visto fin'ora, seguendo tutta la catena di tabelle dal livello 4 fino al livello 1; alla fine, oltre ad eseguire la traduzione e completare l'accesso in memoria per conto del processore, memorizzerà nel TLB il descrittore appena usato, possibilmente rimpiazzandone un altro. Tipicamente il TLB è una memoria associativa ad insiemi e il descrittore da rimpiazzare sarà scelto in base ad un algoritmo di pseudo-LRU. In Figura 8 è mostrato un esempio di TLB a 2 vie. I capi dati di ogni via memorizzano i campi F dei descrittori di livello 1. La parte offset dell'indirizzo V in ingresso non è utilizzata per accedere alla memoria cache, ma andrà direttamente a far parte dell'indirizzo fisico. Un TLB moderno potrebbe avere 8 vie e contenere 1024 descrittori in totale (quindi $1024/8 = 128$ indici).

Il TLB, come tutte le cache, è trasparente al software, persino al software di sistema: l'architettura non prevede istruzioni che permettano al software di esaminarne il contenuto. Le uniche istruzioni che l'architettura mette a disposizione sono quelle che permettono di *invalidarlo* in tutto o in parte. Queste operazioni si rendono necessarie quando il software modifica qualcosa nelle tabelle, rendendo dunque non più valide le informazioni che potrebbero trovarsi nel TLB. In particolare, le istruzioni disponibili per l'invalidazione sono due:

- `movq %rax, %cr3`, che già conosciamo; questa istruzione cambia potenzialmente tutta la tabella di livello 4 usata fino al momento prima, quindi tutte le informazioni contenute nel TLB sono da considerarsi non più valide

e l'intero TLB viene svuotato;

- `invlpg` *operando in memoria*: questa istruzione dice al TLB di invalidare la traduzione relativa all'indirizzo dell'operando passato come argomento;

Notiamo ora cosa comporta il fatto che, se il descrittore si trova già nel TLB, la MMU non percorre la catena delle tabelle:

1. che succede quando la routine di page fault rimpiazza una pagina vittima?

2. la MMU può usare i bit PCD e PWT che trova nel TLB, ma cosa accade ai controlli sui bit R/W e CPL, che andavano eseguiti per ogni descrittore della catena, e non solo sull'ultimo?

3. la MMU non aggiornerà i bit A nei descrittori dopo la prima volta che ha usato un descrittore, almeno fino a quando quel descrittore resta nel TLB;

4. supponiamo che un descrittore venga caricato a causa di una operazione di lettura e abbia in quel momento il bit D a zero; poi il processore esegue una operazione di scrittura all'interno della stessa pagina e la MMU trova la traduzione nel TLB. Che accade al bit D nel descrittore in memoria?

Nel caso 1 la routine di page fault deve invalidare (usando l'istruzione `invlpg`) la traduzione della pagina vittima dopo aver posto il bit P a zero nel suo descrittore di livello 1. Se non lo facesse, e il TLB avesse una copia del descrittore caricata precedentemente, la MMU continuerebbe a tradurre l'indirizzo virtuale della pagina rimossa ottenendo l'indirizzo della pagina fisica che, dopo il rimpiazzamento, contiene ora una pagina virtuale completamente diversa.

Il caso 2 è risolto direttamente dalla MMU. Consideriamo per esempio il bit R/W, che vieta le scritture in una pagina virtuale quando vale 0 in uno qualunque dei descrittori nella catena. La MMU, mentre percorre la catena, calcola l'AND di tutti i bit R/W incontrati. Alla fine salva questo AND nella sua copia del descrittore di livello 1 nel TLB, nella posizione del bit R/W. Quando in seguito dovrà controllare se una operazione di scrittura sulla pagina descritta da questo descrittore è permessa o meno, il controllo sul bit R/W contenuto nel TLB sarà equivalente al controllo di tutti i bit R/W della catena. Un meccanismo analogo funziona anche per i bit CPL.

Consideriamo il caso 3. Il problema, in questo caso, è che il mancato aggiornamento dei bit A per tutti i descrittori che si trovano nel TLB falsa le statistiche sugli accessi calcolate dalla routine di page fault. Quel che è peggio, proprio le pagine il cui descrittore si trova nel TLB, e che sono quindi probabilmente quelle usate più di recente, vedrebbero i loro contatori non aggiornati. Per rimediare a questo problema si richiede un intervento della routine di page fault: dopo aver letto e azzerato tutti i bit A, deve *invalidare* l'intero TLB. In questo modo gli accessi successivi costringeranno la MMU a percorrere la catena e rimettere a 1 i bit A delle pagine a cui il programma sta accedendo. Per invalidare tutto il TLB basta eseguire la coppia di istruzioni `movq %cr4, %rax; movq %rax, %cr3`. La seconda, infatti, invalida il TLB anche se il contenuto di `cr3` non cambia.

Consideriamo infine il caso 4. Il problema, in questo caso, è che la routine di page fault deve sapere se ci sono state scritture su una pagina perché, se la pagina è scelta come vittima, deve sapere se è necessario ricopiarla nell'area di swap. La routine di page fault ricava questa informazione dal bit D nei descrittori che si trovano in memoria (ricordiamo che non ha modo di sapere cosa ci sia dentro il TLB), e dunque il valore di tale bit deve essere corretto. Questo problema è risolto dalla MMU, che agisce nel seguente modo:

1. quando il descrittore cercato non è nel TLB si comporta come già detto (percorre tutta la catena e poi carica il descrittore trovato nel TLB); questa operazione porta anche il valore corrente del bit D nel TLB;

2. se invece il descritore cercato è nel TLB e

   (a) l'accesso è in scrittura e il bit D vale 0, si comporta come se il descrittore *non* fosse nel TLB;

   (b) negli altri casi (lettura, oppure scrittura con D=1) usa il descrittore nel TLB.

Il caso interessante è il 2.(a). Se in questo caso la MMU si limitasse ad usare l'informazione nel TLB senza andare ad aggiornare il descrittore in memoria, la routine di page fault potrebbe non sapere mai che c'è stata una scrittura su quella pagina. Invece la MMU percorre tutta la catena e aggiorna il bit D. Alla fine ricopia anche il descrittore nel TLB, e questa volta avrà il bit D=1: in questo modo (caso 2.(b)) questa operazione viene eseguita soltanto per la prima scrittura sulla pagina.