1 Lezione del 01-04-25

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

1.0.1 MMU Reale

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

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

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

1.0.2 Bootstrap della MMU

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

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

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

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

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

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

1.0.3 Translation Lookaside Buffer

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

Visto che era proprio la RAM ad essere, come avevamo detto, la parte più lenta del sistema, è necessario introdurre una cache a parte dedicata alla MMU, che viene detta **TLB**, *Translation Lookaside Buffer*. Il TLB tiene traccia delle coppie indirizzo virtuale -

indirizzo fisico più usate, limitando la necessità dei table walk alle sole istanze dove sono strettamente necessari.

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

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

figura

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

1.0.4 Pagine di grandi dimensioni

Vediamo un ultimo dettaglio sulla MMU approfondendo il discorso delle pagine di grandi dimensioni (*huge page*). Storicamente, trovata una huge page, la MMU la caricava interamente all'interno della cache TLB. Oggi, si mantengono 3 TLB separati per ogni dimensione di pagina (1 GiB, 2 MiB e 4 KiB).