1 Lezione del 31-03-25

Continuiamo il discorso sulla paginazione.

1.0.1 Funzionamento della MMU

Avevamo definito una MMU che definiva tabelle di corrispodenza fra pagine e frame, una per ogni processo in esecuzione. Avevamo quindi detto che il processo (diciamo P_1) in esecuzione ha più sezioni di dati, cui potremmo assegnare ad esempio un frame ciascuna:

Sezione	**Frame**
Text (Code) P_1	2
Data P_1	3
...	//
Stack P_1	4

I *frame* di memoria scelti e sono in posizioni arbitrarie, l'unica cosa importante è che la MMU li possa rintracciare attraverso le sue tabelle di corrispondenza.

Potremo quindi assumere che la pagina 0 sia riservata, la pagina 1 riservata al sistema, e vedere che una tabella di corrispondenza per P_1 potrebbe essere la seguente:

Pagina	**Sezione**	**Frame**
0	Null	//
1	Sistema	1
2	Text P_1	2
3	Data P_1	3
...	...	//
7	Stack P_1	4

dove si è scelto di disporre lo stack in fondo allo spazio di memoria.

Nel momento in cui un altro processo (diciamo P_2) entra in esecuzione, potremmo assegnargli le seguenti pagine:

Sezione	**Frame**
Text P_2	5
Data P_2	6
...	//
Stack P_2	7

e disporre una tabella di corrispondenza:

Pagina	**Sezione**	**Frame**
0	Null	0
1	Sistema	1
2	Text P_2	5
3	Data P_2	6
...	...	//
7	Stack P_2	7

Vediamo che la pagina sistema resta nella tabella, ergo quella pagina è **condivisa** fra più processi. Cambiare contesto significherà quindi, oltre che caricare i registri, passare da una tabella di corrispondenza di processo all'altra. Il fatto che la pagina sistema sia sempre la 1 ci assicura che i suoi indirizzi siano per il programmatore sempre gli stessi.

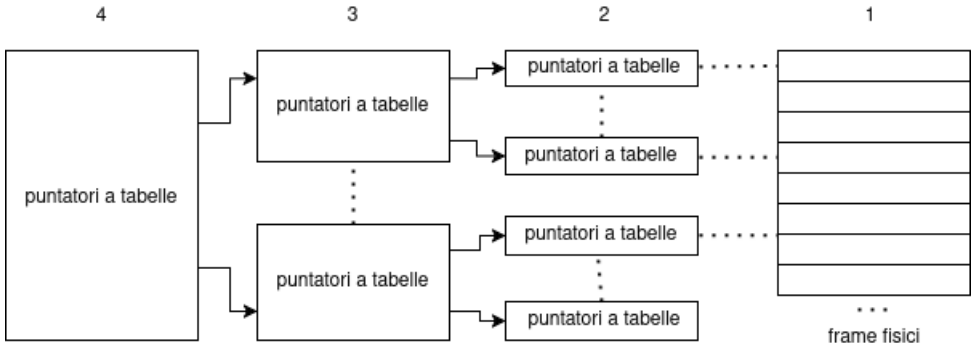
1.0.2 Verso la MMU reale

Vediamo che la MMU come l'abbiamo definita adesso è effettivamente di impossibile realizzazione. Vediamo infatti le dimensione di queste tabelle: se si prendono 12 bit di offset, per pagine da 4 KiB, si ha che nei 48 bit indirizzabili dall'architettura x86 (senza estendere a 57) lasciano 36 bit, e quindi 2^{36}, cioè 64 miliardi (64 Gi, *Gibi*) circa di pagine. Se vogliamo dedicare 8 byte ad ogni entrata di una tabella di corrispondenza, quindi, abbiamo bisogno di 512 GiB di spazio, che non è chiaramente fattibile (considerando poi che vogliamo una tabella per processo, quindi moltiplicando questo valore per un n arbitrariamente grande).

Chiamiamo quindi il modello fittizio visto finora **S-MMU** (da *Super MMU*) e ne introduciamo una versione più vicina alla realtà, che adotta una struttura dati diversa: la **T-MMU** (da *Trie MMU*).

La **trie** è una struttura dati iche nasce per effettuare ricerche chiave-valore. Sono simili agli alberi binari, con la differenza che la chiave non è memorizzata nei nodi, ma nella posizione stessa dei nodi all'interno dell'albero.

Utilizziamo le trie per realizzare una struttura dati detta **bitwise tree**, o *albero bitwise*. L'idea è quella di dividere i 36 bit di pagina in 4 porzioni da 9 bit ciascuna, sulle quali costruire delle trie. La radice della struttura che costruiamo sarà quindi una tabella di $2^9 = 512$ entrate, corrispondente alle 512 possibili configurazioni dei primi 9 dei 36 bit di pagina. Ognuna di queste entrate punterà ad un altra tabella di 512 entrate, relative ai 9 bit successivi. Si hanno quindi 4 livelli di accesso, ordinati dal 4 all'1, che bisogna attraversare per arrivare fino al frame corrispondente alla pagina che ci interessa:



Questa struttura sta in memoria, e processore è dotato di un registro **CR3** che mantiene la posizione della prima tabella (la 4). Il procedimento che ci porta dai bit di pagina all'indirizzo del frame si chiama **table walk**, o *cammino della tabella*. Ogni entrata delle tabelle di trie sarà grande 8 byte (almeno 7 bit per i flag, più ~ 48 bit di indirizzo, ricordando che lo spazio indirizzabile nell'x86_64 non corrisponde al massimo di 64 bit), per cui $2^9 \cdot 2^3 = 2^{12} = 4$ KiB di memoria ciascuna. La memoria massima e il numero di entrate di ogni livello sono quindi:

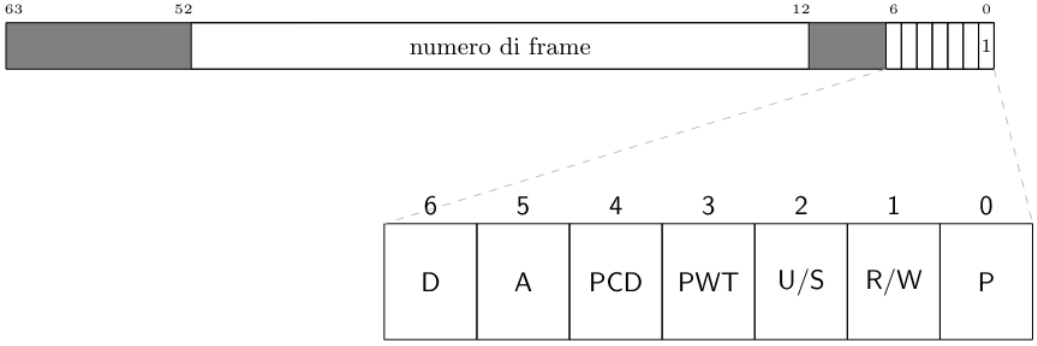
Livello	Memoria massima usata	Numero di entrate
4	$2^9 \cdot 2^3 = 2^{12} = 4$ KiB	$2^9 = 512$
3	$2^9 \cdot 2^9 \cdot 2^3 = 2^{21} = 2$ MiB	$2^9 \cdot 2^9 = 2^{18} = 256$ Ki
2	$2^9 \cdot 2^9 \cdot 2^9 \cdot 2^3 = 2^{30} = 1$ GiB	$2^9 \cdot 2^9 \cdot 2^9 = 2^{27} = 128$ Mi
1	$2^9 \cdot 2^9 \cdot 2^9 \cdot 2^9 \cdot 2^3 = 2^{39} = 512$ GiB	$2^9 \cdot 2^9 \cdot 2^9 \cdot 2^9 = 2^{36} = 64$ Gi

Potremmo chiederci dov'è il guadagno, in quanto a memoria l'ultimo livello di trie necessiterà degli stessi 512 GiB, più lo spazio necessario ai livelli precedenti. Il vantaggio sarà però quello di poter tagliare arbitrariamente rami dall'albero che abbiamo formato, cioè non tenere conto di pagine di cui non abbiamo attualmente bisogno.

1.0.3 Descrittori nella T-MMU

Vediamo come si evolvono i descrittori che avevamo messa corredo delle tabelle di corrispondenza, nella T-MMU. Avremo che ci dovrà essere una distinzione fra i descrittori di primo e di secondo, terzo e quarto livello:

- **Descrittori di primo livello:** qui vogliamo usare l'intero insieme di descrittori, che riportiamo:

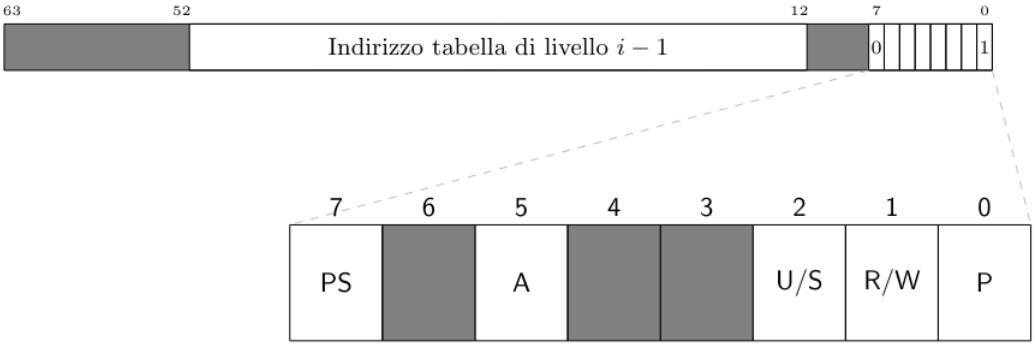


  - **P:** un bit di presenza, che definisce l'esistenza o meno di una traduzione per quell'indirizzo: nel caso di accesso a pagine non traducibili si genera un eccezione, detta **page fault**, che comporta il caricamento della pagina richiesta o la terminazione forzata del programma per **segmentation fault**.

    Ad esempio, se scegliamo 0 come la codifica del null pointer, vogliamo che la prima pagina (o le prime pagine, se vogliamo essere più larghi con accessi a strutture puntate da null pointer, che potrebbero avere offset negli struct anche consideravoli) sia non presente, e quindi si traduca in eccezione prima di effettuare accessi chiaramente erronei;
  - **S/U:** *Sistema/Utente*, indica se una pagina è accessibile o meno ad un processo utente;
  - **R/W:** *Read/Write*, indica se una pagina è accessibile solo in scrittura o solo in lettura per un certo processo. Questa può essere utile ad esempio per la sezione *text* del programma, che ricordiamo contiene il codice e non vogliamo venga modificata;
  - **PCD** e **PWT**: indicano se ignorare completamente la cache (PCD) o se adottare una politica di scrittura *write-through* (PWT). Questo può essere utile nel caso di dispositivi mappati in memoria (come l'APIC o l'adattatore video);

  - **A** e **D**: due flag che danno indicazioni agli accessi che la MMU ha individuato sulla pagina.
    Vediamo solo adesso la loro utilità: la MMU setta questi bit per dare informazioni al kernel su cosa è successo alle pagine fino all'ultimo accesso. Il bit **A**, quindi, indica che una certa pagina è stata usata (*attraversata*), mentre il bit **D** (*Dirty*) indica che si è scritto su una certa pagina. Abbiamo quindi una situazione dove è l'*hardware* ad informare il *software* del suo funzionamento, e non viceversa (come eravamo abituati). L'informazione può quindi essere usata per gestire meglio il caricamento su e da memoria delle pagine, soprattutto in sistemi che supportano la *memoria di swap*, cioè una certa porzione di memoria sul disco rigido che viene impiegata nella memorizzazione delle immagini dei processi in esecuzione (che è come, in origine, avevamo ipotizzato funzionasse il meccanismo della multiprogrammazione). In questo caso conoscere il flag D può evitare una scrittura su disco quando una pagina non è stata modificata, mentre conoscere il flag A può dare un euristica su quali pagine conviene spostare nello swap e quali mantenere nel caso di spostamento di pagine da e su disco. Per la precisione, in sistemi di questo tipo i pagefault sono normali, e vengono sfruttati per realizzare la *paginazione su domanda*: può essere che la pagina richiesta da un processo non esista, quindi comporti un'eccezione, che viene gestita caricando la pagina corrispondente (e quindi verificando i flag A se altre pagine vanno rimosse per fare spazio).

- **Descrittori di secondo, terzo e quarto livello:** il descrittore ha questo aspetto:



  in questo caso non abbiamo bisogno di **PWT**, **PCT** e **D**, mentre introduciamo un nuovo bit, **PS**, *Page Size*, che distingue due situazioni: se PS è basso, si procede come si è detto finora, altrimenti, quella entrata punta ad un unica pagina contigua di entrate (e non al livello successivo della trie). Il numero di entrate delle pagine contigue, dette **huge page**, cambia in base al livello:

Livello	Memoria indirizzata dalla huge page
4	//
3	1 GiB
2	2 MiB
1	4 Kib (default)

  Come si vede dalla tabella, il flag PS è ignorato al livello 4 e al livello 1.

1.0.4 T-MMU e memoria condivisa

La struttura ad albero delle trie ci permette, ad esempio, di far puntare un entrata di un sottoalbero della trie associata ad un processo, ad un sottoalbero di una trie di un altro processo. Questo ci permette effettivamente di realizzare pagine, o tabelle di pagine, condivise fra processi. Potremo liberamente assegnare la stessa pagina in posizioni diverse dello spazio di memoria di ogni processo, in quanto l'unica cosa importante è il *percorso* che ci porta alla tabella condivisa, che può variare di processo in processo (o meglio di trie di processo in trie di processo).