# Paginazione

G. Lettieri

14 Aprile 2024

Ricordiamo che lo stato di un processo contiene sia lo stato della CPU, sia lo stato della memoria (privata del processo). Per quanto visto finora, ogni volta che si esegue un cambio di processo tutta la memoria privata del processo uscente, contenuta in M2, deve essere ricopiata sull'hard disk e sostituita con la memoria privata del processo entrante, letta dall'hard disk. L'hard disk (o una sua partizione) dedicato a contenere le immagini delle memorie private dei processi è detto (dispositivo o partizione di) *swap* (scambio).

Nel caso generale in cui i processi potrebbero anche non avere parti condivise, può essere necessario trasferire da e verso lo swap l'intero contenuto di M2. Si noti che questo è vero anche per i processi che non usano tutta la memoria M2, in quanto il sistema non può sapere a quali parti della memoria i processi accedono mentre hanno il controllo della CPU.

Quello che vogliamo fare ora è di eliminare, o quantomeno ridurre, queste copie da e verso lo swap. Teniamo però presente che la soluzione che trasferisce tutta M2, per quanto costosa, ha dei vantaggi architetturali che vorremmo comunque preservare:

1. *Isolamento tra i processi.* Ciascun processo può accedere solo alla propria memoria privata (la memoria privata degli altri processi è al sicuro nell'area di swap, a cui i processi utente non hanno accesso).

2. *Semplicità nel collegamento dei programmi.* Il collegatore può sempre assumere che ogni programma abbia a disposizione l'intera memoria M2.

3. *Semplicità nel caricamento/scaricamento dei processi.* La memoria privata di ogni processo viene ricaricata esattamente nella stessa posizione ogni volta che il processo torna in esecuzione, quindi tutti gli indirizzi che il processo usa sono sempre validi.

4. *Possibilità di condivisione della memoria.* Se più processi hanno bisogno di condividere memoria tra loro, il sistema può concederlo evitando di sostituire le parti di memoria condivise ogni volta che cambia processo.

Un modo per riassumere la situazione è di dire che ogni processo "pensa" di avere una CPU e una memoria M2 tutte per sé, ma in realtà ha una CPU virtuale e una M2 virtuale. La vera CPU e la vera memoria M2 incarnano ad ogni istante la CPU virtuale e la memoria M2 virtuale del processo attualmente

La figura mostra una singola colonna verticale divisa in più zone. In alto ci sono la sezione .text, che contiene il codice eseguibile, e subito sotto la sezione .data, che contiene i dati statici. Più in basso c'è un'unica grande regione condivisa da heap e stack. L'heap parte dalla parte alta di questa regione e cresce verso il basso, mentre lo stack parte dalla parte bassa e cresce verso l'alto. Le due frecce indicano quindi che heap e stack si espandono uno verso l'altro. Il messaggio della figura è che il processo ha aree iniziali fisse per codice e dati, mentre heap e stack occupano gli estremi opposti di una stessa area dinamica e possono crescere fino a incontrarsi.

Figura 1: Schema semplificato della memoria di un processo.

in esecuzione, mentre l'ultimo stato delle CPU virtuali e delle memorie M2 virtuali degli altri processi sono al sicuro, rispettivamente, in M1 (nei campi `contesto` dei descrittori di processo) e nell'area di swap.

# 1 Ridurre i trasferimenti da/verso lo swap

Per ridurre i trasferimenti introdurremo una serie di meccanismi via via più sofisticati, in modo da capire le motivazioni dietro il meccanismo che adotteremo alla fine.

## 1.1 Registri limite inferiore e superiore

Supponiamo di sapere di quanta memoria ha bisogno ogni processo. Possiamo supporre che sia il programmatore stesso a dirlo al sistema (magari aiutato da compilatore e collegatore). Un processo avrà bisogno di un po' di memoria per contenere la sezione **`.text`**, contenente il codice del programma da eseguire, e di altra memoria per la sezione **`.data`**, contenente (in C++) le variabili globali. La dimensione di queste sezioni è nota al collegatore. Ci sono però altre due sezioni, inizialmente vuote, che si possono espandere durante l'esecuzione: la pila e lo heap (usato per la memoria dinamica). Per queste il programmatore deve stabilire un massimo. La memoria di un processo viene tipicamente organizzata come in Figura 1, con tutte le sezioni contigue e con lo heap e la pila che condividono una stessa zona di memoria. È sufficiente che il programmatore dica al sistema quanto deve essere grande la zona utilizzata dallo heap e dalla pila. Il compilatore, il collegatore, o anche il sistema stesso, possono assumere un valore di default, per non costringere il programmatore a specificare questa dimensione nei casi più comuni.

Una volta che questa informazione è nota al sistema, questo può sfruttarla per copiare da e verso lo swap solo la memoria effettivamente usata dai processi entranti e uscenti, invece che l'intera M2. Ricordiamo, però, che il sistema non può fidarsi dei processi utente, quindi non può essere sicuro che, mentre un

Ogni sottofigura rappresenta una colonna di memoria con il modulo M1 in alto e vari processi caricati sotto: $P_1$, $P_2$, $P_3$ e $P_4$. Nella sottofigura (a) i quattro processi sono caricati in memoria uno sotto l'altro. Nella sottofigura (b) il processo in esecuzione è $P_2$ e compaiono due limiti, LINF e LSUP, che delimitano la parte di memoria accessibile: in pratica il processo può accedere solo alla propria zona e non a quella degli altri. Nella sottofigura (c) i valori di LINF e LSUP vengono modificati, e di conseguenza cambia anche la regione di memoria accessibile. Il messaggio della figura è che il sistema può controllare quali parti della memoria sono accessibili a seconda del processo attivo, modificando opportunamente i limiti inferiore e superiore.

Figura 2: Tre istantanee della memoria che mostrano come cambia la zona accessibile a un processo.

processo utente ha il controllo della CPU, acceda realmente soltanto alla parte di M2 che aveva dichiarato: per eseguire questo controllo c'è bisogno dell'aiuto dell'hardware.

Per il momento abbiamo previsto nella CPU un meccanismo che protegge la memoria M1 mentre la CPU è sotto il controllo dei processi utente. Ricordiamo in cosa consiste questo meccanismo:

- abbiamo introdotto un registro, chiamiamolo `LINF`, scrivibile solo da livello sistema, che contiene un indirizzo limite inferiore;
- abbiamo modificato la CPU in modo che, quando si trova a livello utente, controlli che ogni operazione in memoria (prelievo istruzioni e lettura/-scrittura operandi) avvenga a indirizzi maggiori o ugali di quello contenuto in `LINF`, sollevando una eccezione in caso contrario;
- all'avvio del sistema, il registro `LINF` viene inizializzato con il primo indirizzo di M2.

Per garantire che ciascun processo non acceda al di fuori della memoria massima dichiarata, possiamo fare un'altra piccola modifica all'hardware della CPU, aggiungendo anche un registro `LSUP` (limite superiore), anch'esso scrivibile solo da livello di privilegio sistema. Quando la CPU lavora in modalità utente, deve controllare che ogni accesso in memoria (sia esso per prelevare dati o istruzioni) avvenga ad indirizzi *compresi* nell'intervallo [`LINF`, `LSUP`). In caso contrario la CPU deve generare una eccezione che restituisca il controllo al sistema, che potrà decidere di terminare il processo con un errore. Durante i cambi di processo, il sistema dovrà anche provvedere a modificare opportunamente il contenuto di `LSUP`, mentre `LINF` resta costante.

Con questo meccanismo manteniamo tutti i vantaggi 1–4 e possiamo ridurre la quantità di byte da trasferire da/verso lo swap ad ogni cambio di processo, ma non il numero dei trasferimenti.

## 1.2 Caricamento di più processi contemporaneamente

Una volta che abbiamo i registri `LINF` e `LSUP`, possiamo pensare di usarli in un altro modo. L'idea di partenza è che la maggior parte dei processi non avrà bisogno di tutta la memoria M2, quindi potremmo pensare di caricare più di uno stato alla volta e di tenere in memoria anche lo stato dei processi che non sono in esecuzione, in modo da averli già pronti quando verranno schedulati. Si veda l'esempio in Figura 2(a). In pratica la memoria M2 si comporterà come una cache dello swap, con gli stessi problemi da risolvere: quando tutta M2 è piena e dobbiamo mettere in esecuzione un processo il cui stato non è in M2, dobbiamo fare spazio togliendo lo stato di qualche altro processo; quale scegliamo? Non ci addentreremo però in questi argomenti, che appartengono al corso di Sistemi Operativi. Ci limitiamo invece a studiare i meccanismi che permettono di realizzare questa cache.

Ovviamente, nella situazione di Figura 2(a), dobbiamo preoccuparci di mantere l'isolamento tra i processi, che altrimenti potrebbero accedere liberamente alla memoria privata degli altri processi. Per garantire l'isolamento è sufficiente che il sistema scriva in `LINF` non il primo indirizzo di M2, ma l'indirizzo della prima locazione appartenente al processo in esecuzione, come mostrato in Figura 2(b)–(c). Entrambi i registri devono avere una posizione nel vettore `contesto` dei descrittori di processo, siano `I_LINF` e `I_LSUP`.

- Ogni volta che un processo viene caricato dallo swap (la prima volta, o dopo che era stato rimosso per fare spazio) il sistema deve inizializzare i campi `contesto[I_LINF]` e `contesto[I_LSUP]` con l'indirizzo iniziale e finale della parte di M2 occupata dal processo.

- Ogni volta che si cambia processo, il sistema aggiorna anche il contenuto di `LINF` e `LSUP` con i valori presi dal descrittore del processo entrante.

In questo modo la CPU controllerà che ciascun processo non possa accedere al di fuori della zona di memoria che gli è stata assegnata.

Questa soluzione ci permette di ridurre enormemente il numero di trasferimenti da/verso lo swap. In compenso, però, perdiamo tutti i vantaggi architetturali che avevamo, tranne il primo:

- Non è più così semplice collegare i programmi. Prima tutti i processi venivano sempre caricati a partire dallo stesso indirizzo (l'indirizzo di partenza di M2), mentre ora possono essere caricati ovunque, in base a quali altri processi si trovano già in memoria al momento del caricamento. L'indirizzo di caricamento non è dunque noto durante la compilazione e il collegamento: come scegliere gli indirizzi a cui collegare i programmi? Ci sono due modi per risolvere questo problema: fare in modo che sia il caricatore a rilocare il programma (con una tecnica del tutto simile a quella usata dal collegatore) in modo da adattarlo all'indirizzo di caricamento; oppure, compilare tutti i programmi in modo che siano indipendenti dalla posizione.

In tutte le sottofigure la memoria è rappresentata come una colonna verticale con M1 in alto e vari processi sotto. In (a) si vede lo stato iniziale: sono presenti $P_1$, $P_2$, $P_3$ e $P_4$. In (b) il processo $P_2$ viene temporaneamente rimosso per fare posto a $P_5$. In (c) i processi $P_3$ e $P_4$ vengono temporaneamente rimossi per poter ricaricare $P_2$. In (d) $P_3$ viene ricaricato e $P_5$ termina; nonostante questo, non rimane più spazio sufficiente per ricaricare anche $P_4$. Il significato della figura è che, con memoria limitata e processi di dimensioni diverse, il caricamento e lo scaricamento temporaneo dei processi può portare a frammentazione o a situazioni in cui non è più possibile reinserire tutti i processi rimossi.

Figura 3: Quattro istantanee della memoria che illustrano caricamenti, rimozioni temporanee e ricaricamenti di processi.

- C'è però un secondo svantaggio, molto più grave: che succede se un processo viene rimosso dalla memoria e poi caricato in una posizione diversa, come il processo $P_2$ in Figura 3(c)? In generale non funzionerà, in quanto il suo stato potrebbe ora contenere indirizzi che erano validi solo nella posizione originaria: si pensi agli indirizzi di ritorno salvati in pila, o ai puntatori al prossimo elemento scritti in qualche lista. Questi indirizzi andrebbero corretti in base alla nuova posizione, ma in questa architettura è praticamente impossibile trovarli: lo stato è una sequenza di byte e non c'è niente che permetta di distinguere un indirizzo da qualunque altra cosa. In pratica perdiamo il vantaggio numero 3.
- Perdiamo anche il vantaggio numero 4: una eventuale parte di memoria condivisa tra i processi esisterebbe ora in più copie e non sarebbe dunque condivisa, a meno che il sistema non la copi da memoria a memoria ogni volta che cambia processo.

## 1.3 Registri base e limite

Possiamo recuperare il vantaggio numero 2 e parte del 3 modificando il comportamento del registro `LINF`. Facciamo in modo che, ad ogni accesso in memoria eseguito da livello utente, la CPU *sommi* il contento di `LINF` all'indirizzo generato dal programma, controllando che il risultato non superi `LSUP`. Il registro `LINF` contiene dunque una "base" per tutti gli indirizzi generati dal processo. I programmi utente possono ora essere collegati a partire dall'indirizzo zero e i processi che li eseguono possono continuare ad usare gli stessi indirizzi per tutta la loro esecuzione, anche se il loro stato viene tolto dalla memoria e ricaricato in seguito ad un indirizzo diverso. Infatti, siccome `LINF` conterrà ora il nuovo indirizzo, tutti gli accessi in memoria verranno automaticamente aggiustati in modo da puntare alle locazioni corrette.

Si presti attenzione al fatto che questa correzione è operata a tempo di esecuzione dall'hardware ed è controllata dal software di sistema, l'unico che può scrivere nei registri `LINF` e `LSUP`. Gli utenti non possono vedere il meccanismo in opera, né tantomeno modificarlo. Quello che gli utenti vedono è che ora ogni processo sembra avere una memoria tutta per sé, come nel sistema iniziale. Questo perché il significato di un indirizzo dipende ora dal contesto: l'indirizzo $x$ di un processo $P_1$ non è lo stesso indirizzo $x$ di un diverso processo $P_2$, in quanto ogni volta che $P_1$ usa $x$ leggerà o scriverà una locazione di memoria diversa da quella letta o scritta da $P_2$, per via dei diversi valori di `LINF` automaticamente sommati a $x$. Possiamo dunque di nuovo dire che ogni processo ha acquisito una sua memoria virtuale, che è l'unica a cui ha accesso. Tutti gli indirizzi generati durante l'esecuzione di un processo (sia per prelevare le istruzioni che per leggere o scrivere gli operandi in memoria) sono relativi alla memoria virtuale del processo, e sono detti "indirizzi virtuali". L'azione di sommare `LINF` agli indirizzi generati dal processo corrisponde ad una *traduzione* da indirizzi virtuali a indirizzi *fisici*, che possono essere usati per accedere alla memoria del sistema, che da ora in poi chiameremo "memoria fisica". I processi utente non hanno alcun controllo sulla traduzione, e dunque nessun accesso diretto agli indirizzi fisici: questo ci garantisce che la loro esecuzione non possa dipendere dagli indirizzi fisici, dando la libertà al sistema di allocarli come meglio crede.

Questo meccanismo non ci permette di recuperare il vantaggio numero 4, ma nemmeno completamente il vantaggio numero 3 (semplicità nel caricamento e scaricamento dei processi). Che succede nella situazione di Figura 2(d) in cui dobbiamo caricare un processo, ma lo spazio in memoria è frammentato in porzioni troppo piccole? Potremmo risolvere questo problema ricompattando lo spazio, ma per farlo dobbiamo copiare la memoria dei processi da una zona all'altra, operazione molto costosa.

## 2 Paginazione

I problemi che abbiamo visto potrebbero essere risolti se potessimo "spezzare" la memoria di un processo in più porzioni e gestire ogni porzione separatamente: potremmo dire che alcune porzioni sono condivise e altre no e caricare ogni porzione in uno spazio diverso. In presenza di memoria virtuale, questo "spezzettamento" sarebbe possibile se potessimo tradurre in modi diversi parti diverse dello spazio di indirizzamento di un processo. Questo è ciò che ci permette di fare il meccanismo della *paginazione*.

L'idea è di considerare tutti i possibili indirizzi generabili da un processo e di raggrupparli in regioni naturali dette "pagine" e di applicare una traduzione di indirizzi diversa per ogni pagina. In questo modo il sistema è libero di caricare la memoria di un processo ovunque vi sia spazio libero, anche se non contiguo. Inoltre, due o più processi possono condividere parte della loro memoria: è sufficiente applicare le stesse traduzioni per le pagine che contengono la zona da condividere.

A sinistra c'è la CPU, che genera un indirizzo virtuale $v$. A destra è rappresentato lo spazio degli indirizzi come una colonna verticale. Sono marcati quattro valori significativi: `0000 0000 0000 0000` in alto; `0000 7fff ffff ffff` alla fine della parte bassa valida; `ffff 8000 0000 0000` all'inizio della parte alta valida; `ffff ffff ffff ffff` in basso. Tra la parte bassa e quella alta compare una zona grigia centrale, che rappresenta un intervallo di indirizzi non utilizzabile. Solo gli $x$ bit meno significativi di un indirizzo di 64 bit possono assumere un valore qualsiasi, dove $x$ vale 48 o 57 a seconda del modello della CPU. I $64 - x$ bit più significativi devono essere tutti uguali al bit n. $x - 1$ (contando da 0). Per fissare le idee consideriamo solo il caso $x = 48$. Il messaggio della figura è che lo spazio degli indirizzi virtuali a 64 bit non è completamente usabile: esistono due regioni canoniche valide, una bassa e una alta, separate da una grande zona vietata che genera eccezione se viene indirizzata.

Figura 4: Schema dello spazio di indirizzamento virtuale canonico in architettura Intel/AMD a 64 bit.

Nel caso dei processori Intel/AMD a 64 bit i possibili indirizzi sono mostrati in Fig. 4, dove si mostra solo il collegamento tra la CPU e la memoria tramite il bus degli indirizzi. L'insieme di questi indirizzi forma lo *spazio di indirizzamento virtuale* di un processo. Gli indirizzi che abbiamo usato fino ad ora (quelli a cui risponde la memoria centrale, la memoria video, l'APIC e eventuali altre periferiche con registri mappati in memoria) fanno invece parte dello *spazio di indirizzamento fisico*. Gli indirizzi che fanno riferimento allo spazio di indirizzamento virtuale sono detti *indirizzi virtuali*, e quelli che fanno riferiemnto allo spazio di indirizzamento fisico sono detti *indirizzi fisici*.

Suddividiamo idealmente lo spazio di indirizzamento virtuale di Figura 4 in regioni naturali[1], che chiamiamo *pagine*. Per realizzare la traduzione suddividiamo anche lo spazio di indirizzamento fisico in regioni naturali, dette *frame* (cornici), della stessa dimensione delle pagine. Assumiamo che le pagine e i frame siano grandi 4 KiB (0x1000 in esadecimale). Ogni indirizzo virtuale può essere scomposto in $(p, o)$, dove $p$ è il *numero di pagina* e $o$ l'offset all'interno della pagina. Similmente, ogni indirizzo fisico può essere scomposto in $(n, q)$, dove $n$ è un *numero di frame* e $q$ un offset all'interno del frame.

Il meccanismo della paginazione ci permette di mappare qualunque pagina su qualunque frame. Per farlo si introduce una struttura dati che, dato un numero di pagina $p$, ci permette di conoscere il numero di frame $n$ su cui $p$ è mappata. La più semplice struttura dati possibile è un array indicizzato dal numero di pagina: se chiamiamo `a` questo array, il numero di frame che

[1] Si ricordi che abbiamo chiamato "regioni naturali" intervalli di indirizzi allineati naturalmente, con una dimensione che sia una potenza di 2.

A sinistra è mostrata una tabella di corrispondenza, con gli indici delle sue entrate scritti accanto in esadecimale. Tra gli indici visibili compaiono i primi valori, poi i valori vicini a `7fffffffff`, quindi `800000000`, e infine l'ultimo valore. A destra è rappresentato lo spazio di indirizzamento virtuale come una colonna verticale, con gli indirizzi iniziali e finali delle pagine. Le linee tratteggiate collegano ciascuna entrata della tabella alla corrispondente pagina virtuale. La figura mette in evidenza anche il salto tra l'ultima pagina della parte bassa, con indirizzo iniziale `0000 7fffffffff 000`, e la prima pagina della parte alta, con indirizzo iniziale `ffff 800000000 000`. Il significato della figura è che ogni entrata della tabella corrisponde a una pagina virtuale; inoltre, a causa della forma canonica degli indirizzi a 64 bit, passando da un certo indice al successivo non si continua nella zona centrale proibita, ma si salta direttamente dalla regione bassa a quella alta dello spazio virtuale.

Figura 5: Relazione tra tabella di corrispondenza e pagine dello spazio di indirizzamento virtuale.

cerchiamo è $n = \mathtt{a}[p]$. Chiamiamo *tabella di corrispondenza* l'array `a`. In Fig. 5 si mostra come la tabella di corrispondenza sia un vettore in cui ogni elemento, in ordine, si occupa della traduzione della corrispondente pagina, in ordine, dello spazio di indirizzamento virtuale. Supponiamo ora che la CPU generi un indirizzo $v$, per esempio per accedere ad una variabile di un programma caricato in memoria centrale. La cella a cui la CPU vuole accedere non si trova in memoria all'indirizzo $v$: come nel caso dei registri `LINF` e `LSUP`, l'indirizzo $v$ deve essere *tradotto* prima di poter essere usato per accedere alla memoria. La traduzione tramite `LINF` consisteva semplicemente in una somma, mentre ora è più complessa. L'indirizzo $v$ cadrà all'interno di una certa pagina, sia la numero $p$, ad un certo offset $o$ all'interno della pagina. Se $p$ è caricata in $n = \mathtt{a}[p]$, la cella che corrisponde all'indirizzo $v$ sarà quella che si trova all'offset $o$ dentro il frame numero $n$. In altre parole, l'indirizzo $(p, o)$ viene tradotto in $(\mathtt{a}[p], o)$ (si noti che l'offset resta lo stesso).

Per eseguire questa traduzione di indirizzi introduciamo un nuovo dispositivo tra la CPU e la memoria: la Memory Management Unit (MMU). La MMU intercetta tutti gli indirizzi generati dalla CPU e li traduce in base alla tabella di corrispondenza attiva, seguendo il procedimento che abbiamo descritto sopra: sostituire il numero di pagina con il numero di frame letto dalla tabella di corrispondenza, lasciando l'offset inalterato. L'operazione di traduzione è riassunta in Figura 6.

La paginazione ci permette di realizzare la cache dello swap senza perdere nessuno dei vantaggi architetturali che avevamo all'inizio:

- *Isolamento tra i processi.* Si ottiene utilizzando una diversa tabella di

In alto compare l'indirizzo virtuale, diviso in due parti utili: il numero di pagina, nei bit da 12 a 47, e l'offset, nei bit da 0 a 11. I bit più alti dell'indirizzo virtuale sono mostrati separatamente e non partecipano direttamente al numero di pagina. Al centro c'è una tabella di corrispondenza: il numero di pagina virtuale entra nella tabella e produce in uscita un numero di frame fisico. In basso compare l'indirizzo fisico, composto dal numero di frame, nei bit da 12 a 51, e dallo stesso offset copiato senza modifiche nei bit da 0 a 11. Il messaggio della figura è che la traduzione virtuale-fisica sostituisce il numero di pagina con un numero di frame ottenuto dalla tabella, mentre l'offset interno alla pagina resta invariato.

Figura 6: Schema della traduzione di un indirizzo virtuale in indirizzo fisico tramite paginazione.

corrispondenza per ogni processo e impedendo che le tabelle di corrispondenza possano essere manipolate da livello utente. In questo modo ogni processo può accedere solo a quelle parti della memoria fisica che si trovano nel codominio della funzione di traduzione realizzata dalla MMU. Per impedire a un processo $P$ di accedere al contenuto di un qualunque frame $n$ è sufficiente che $n$ non compaia mai nella tabella di corrispondenza di $P$.

- *Semplicità nel collegamento dei programmi.* Utilizzando una diversa tabella di traduzione per ogni processo, possiamo creare per ogni processo l'illusione che tutti gli indirizzi siano disponibili, quindi il collegatore li può assegnare liberamente.

- *Semplicità nel caricamento/scaricamento dei processi.* I processi usano solo ed esclusivamente gli indirizzi virtuali, che non cambiano anche se i processi vengono rimossi dalla memoria e caricati in un altro punto. Inoltre, ogni pagina può essere caricata ovunque ci sia un frame libero e non è più necessario caricare un intero processo in una porzione contigua della memoria.

- *Possibilità di condivisione della memoria.* Se si vuole che due o più processi condividano della memoria, è sufficiente che il sistema inserisca gli stessi numeri di frame nelle entrate opportune delle varie tabelle.

Il programmatore deve comunque dire al sistema quanto spazio occupa ogni processo, con l'unica differenza che ora lo spazio si misura in pagine.

## 2.1 Funzioni aggiuntive

La presenza della MMU permette al sistema di realizzare anche altre funzioni. La MMU, infatti, si trova in un punto in cui può osservare tutti gli indirizzi

generati dal software e, per ognuno di essi, consulta la tabella di corrispondenza. La tabella può essere usata per associare ulteriori informazioni ad ogni pagina (oltre al numero di frame) che dicano alla MMU come comportarsi quando intercetta un indirizzo che cade in quella pagina. Le entrate delle tabelle che si trovano nell'architettura AMD/Intel includono i seguenti campi:

- un flag P che dice se la traduzione è valida;
- un flag R/W che dice se sono ammesse scritture nella pagina;
- un flag U/S che dice se sono ammessi accessi alla pagina da livello utente.
- due flag, PWT e PCD, per agire sulla cache[2].
- due flag, A e D, che danno indicazioni sugli accessi che la MMU ha osservato sull pagina.

Il flag P serve a marcare le pagine che il processo non usa. Si ricordi che la tabella di corrispondenza contiene una entrata per ogni possibile pagina. Il caricatore di sistema (nel nostro caso, la `activate_p()`), provvederà a porre P=0 in tutte le pagine di cui il processo non ha bisogno. Se la MMU riceve dalla CPU un indirizzo che porta ad una entrata con P=0, fa in modo che la CPU sollevi una eccezione. Il sistema, tipicamente, terminerà il processo con un errore[3]. Il bit P può essere anche utilizzato per fermare i processi quando cercano di dereferenziare un puntatore nullo: è sufficiente porre P=0 nell'entra di indice 0 (relativa alla pagina numero 0).

Il flag R/W può essere usato per proteggere dalla scrittura la sezione **`.text`**. Infatti, anche se l'architettura detta di "von Neumann" era stata introdotta proprio per permettere ai programmi di modificare il proprio stesso codice, questa pratica è da tempo fortemente limitata, in quanto crea molti più problemi di quanti ne risolva[4]. Tipicamente viene completamente vietata per i processi utente, settando opportunamente i flag R/W. La MMU fa sollevare una eccezione anche quando incontra R/W=0 nel tradurre l'indirizzo durante una operazione di scrittura.

Per capire la necessità del flag U/S, consideriamo che la MMU presente nei sistemi Intel/AMD è *sempre* attiva, anche quando il processore si trova a livello sistema. Se vogliamo che sia possibile saltare al codice di sistema quando un processo invoca una primitiva, genera una eccezione o riceve una interruzione esterna, dobbiamo fare in modo che tutta la memoria M1 si trovi nel codominio delle funzioni di traduzione di tutti i processi. Dobbiamo dunque riservare delle pagine nella memoria virtuale di ogni processo, in modo che vengano tradotte nei frame che coprono M1. Nel processore AMD64 sfruttiamo il fatto che la

[2] Questi flag sono stati introdotti nel processore 80486.

[3] In Unix il processo riceve una signal `SIGSEV`, che normalmente ne causa la terminazione con ritorno alla shell, che poi stampa "Segmentation fault".

[4] Si noti che l'articolo originale di von Neumann già limitava i modi in cui i programmi dovevano poter modificare il proprio codice, ma i primi calcolatori poi realizzati non applicavano queste limitazioni. In ogni caso, anche il meccanismo più limitato suggerito da von Neumann non è realmente necessario.

memoria virtuale è naturalmente divisa in due (Figura 4) e riserviamo la parte superiore al sistema e la parte inferiore all'utente[5]. Allo stesso tempo non vogliamo che i processi utente possano accedere alla memoria M1. Potremmo ricorrere nuovamente al registro limite, ma ora che abbiamo la MMU possiamo sfruttarla per farle fare anche questo controllo: basta porre U/S=sistema in tutte le entrate relative alla parte alta dello spazio di indirizzamento (le entrate con indici da `000000000` a `7fffffff` in Figura 5). Anche in questo caso la MMU fa generare una eccezione se rileva un accesso da livello utente ad una pagina con U/S=sistema.

I bit PWT e PCD sono un mezzo tramite il quale il sistema può indirettamente inviare ordini alla cache, sfruttando il fatto che la MMU si trova sul percorso che porta dal processore alla cache (la cache si trova tra la MMU e la memoria fisica). La MMU si preoccuperà di inoltrare l'ordine alla cache ogni volta che traduce un indirizzo.

1. Il bit PCD (Page Cache Disable) ordina alla cache di non intercettare l'operazione (sia essa di lettura o di scrittura) e di lasciarla passare inalterata sul bus, similmente a come si comporta per gli accessi nello spazio di I/O. La routine di inizializzazione porrà PCD=1 per tutte le pagine che contengono indirizzi di registri di I/O mappati in memoria, invece che locazioni di memoria. Un esempio è l'APIC, i cui indirizzi sono appunto mappati nello spazio di memoria.

2. Il bit PWT (Page Write Through) ordina alla cache di usare la politica *write-through* per questo particolare accesso (ovviamente, solo se si tratta di una scrittura). Se PCD è 1 il bit PWT non ha effetto. Se PCD è 0, porre PWT=1 può essere utile per la parte di indirizzi relativa alla memoria video: quando il programma scrive vogliamo che la scrittura arrivi nella vera memoria video e non si fermi in cache, in modo che il controllore video possa visualizzare l'informazione sul display; ma se il programma vuole leggere dalla memoria video possiamo tranquillamente farlo leggere dalla cache.

Incidentalmente, si noti che l'aver posizionato la cache dopo la MMU comporta che la cache non deve subire modifiche rispetto a quella che abbiamo già studiato: il controllore cache vede arrivare indirizzi fisici esattamente come prima (arrivano dalla MMU invece che direttamente dalla CPU, ma questo è irrilevante). Inoltre, la presenza della cache non turba il funzionamento della MMU: la presenza della cache è trasparente anche per la MMU. Questa è la soluzione adottata nei processi Intel/AMD64.

I bit A e D, infine, sono legati all'implementazione dell'area di swap, a cui accenniamo solo dal punto di vista teorico (non la realizzeremo nel sistema didattico). La MMU setta ad 1 il bit A di una entrata quando la usa, cioè durante un accesso ad un indirizzo all'interno della corrispondente pagina. Se l'accesso era in scrittura, la MMU setta anche il bit D. Consideriamo, per esempio, un

[5] Linux e FreeBSD fanno al contrario.

A sinistra c'è la CPU, che produce un indirizzo virtuale $v$. Al centro c'è una Super-MMU, rappresentata come un blocco contenente più tabelle, una per ciascun processo, ad esempio $P_1$ e $P_2$. A destra c'è la memoria fisica, etichettata MEM. L'indirizzo virtuale entra nella Super-MMU, che lo traduce in un riferimento fisico $f$ e lo inoltra alla memoria. Quando è in esecuzione il processo $P_i$, la Super-MMU usa la tabella corrispondente a quel processo. Il messaggio della figura è che la traduzione degli indirizzi non dipende solo dall'indirizzo richiesto, ma anche da quale processo è attivo: processi diversi possono tradurre lo stesso indirizzo virtuale in posizioni fisiche diverse usando tabelle diverse.

Figura 7: Schema di una MMU estesa che usa una tabella diversa per ciascun processo.

sistema che gestisca il caricamento (*swap-in*) e scaricamento (*swap-out*) dei processi da e verso un dispositivo di swap (per esempio, un hard disk). In questo caso lo stato iniziale dei processi si trova sullo swap. Ogni volta che il sistema carica le pagine di un processo dallo swap in memoria dovrebbe porre D=0 in tutte le entrate della tabella di corrispondenza. Al momento di eseguire uno swap-out del processo, il sistema può evitare di riscrivere nello swap le pagine che non sono state modificate mentre erano in RAM. Per scoprire quali siano queste pagine, può esaminare le entrate delle tabelle (di livello 1) notando in quali di esse il bit D è diventato 1: queste puntano a pagine che sono state modificate e vanno risalvate; per tutte le altre il salvataggio si può evitare, in quanto la copia già presente nello swap è ancora valida. Il bit A può essere usato per capire quali pagine/tabelle sono più usate o sono state usate più recentemente, ed è di ausilio soprattutto all'implementazione della *paginazione su domanda*, in cui non vengono caricati in memoria tutte le pagine di un processo ma solo le pagine a cui il processo effettivamente accede, sfruttando il bit P e intercettando l'eccezione di page fault.

# 3 La Super-MMU

Precedentemente, per fare in modo che ogni processo fosse isolato dagli altri, avevamo una coppia di valori `contesto[I_LINF]` e `contesto[I_LSUP]` per ogni processo, con una sola coppia attiva ad ogni istante (quella appartenente al processo in esecuzione). Analogamente, ora dovremo avere una intera tabella di corrispondenza distinta per ogni processo, con una sola tabella attiva ad ogni istante (quella appartenente al processo in esecuzione). Per introdurre i dettagli un po' alla volta, in modo da concentrarci prima sugli aspetti più importanti, introduciamo prima una Super-MMU che contiene essa stessa tutte le tabelle di corrispondenza, di tutti i processi, al proprio interno (Figura 7).

È disponibile un semplice esempio che illustra quando detto finora, facendo uso della Super-MMU. Uno degli scopi dell'esempio è di far vedere come la memoria virtuale sia completamente *trasparente* al programmatore utente.