# Paginazione

G. Lettieri

3 Maggio 2019

All'avvio del nostro sistema lo stato iniziale dei processi si trova memorizzato in dispositivo di swap (un hard disk). Questo è stato inizializzato usando un sistema di sviluppo separato, in esecuzione su un'altra macchina. Il dispositivo di swap viene prima collegato al sistema di sviluppo e, quando lo stato iniziale dei processi è stato creato, staccato e collegato alla macchina su cui viene poi avviato il nostro sistema. La primitiva `activate_p()` copia lo stato del processo da attivare dal dispositivo di swap alla memoria (nella parte M2).

Per quanto visto finora, ogni volta che si esegue un cambio di processo l'intero contenuto di M2, che contiene lo stato attuale del processo uscente, deve essere ricopiato nello swap e sostituito con lo stato del processo entrante, letto dallo swap. Incidentalmente, si noti che tutte queste operazioni sono svolte dal codice di sistema: il codice di livello utente non ha alcun accesso al dispositivo di swap.

Quello che vogliamo fare ora è di eliminare, o quantomeno ridurre, le copie da e verso lo swap ogni volta che si cambia processo. 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. 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 il meccanismo che permette di mantenere in memoria lo stato di più di un processo.

## 1 Problemi

Affronteremo i problemi gradualmente, in modo da capire le motivazioni dietro il meccanismo che adotteremo alla fine.

### 1.1 Dimensione massima di ogni processo

Per prima cosa dobbiamo assumere di sapere di quanta memoria ha bisogno ogni processo. Supporremo che sia il programmatore stesso a dirlo al sistema (magari

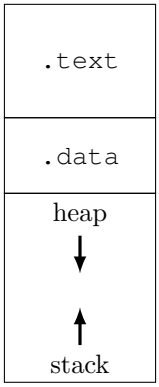


Figura 1: Organizzazione della memoria di un processo. Lo heap e lo stack si trovano ai capi opposti di un'unica regione di memoria, con lo heap che si espande verso il basso e lo stack verso l'altro.

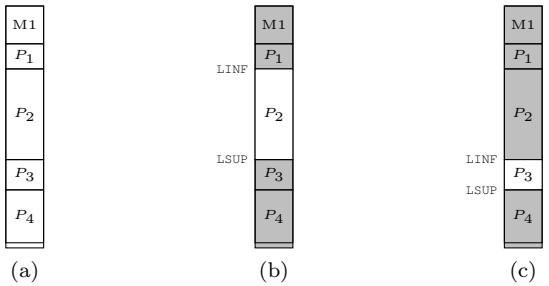


Figura 2: Un esempio di evoluzione del sistema. (a) vengono caricati $P_1$, $P_2$, $P_3$ e $P_4$; (b) quando è in esecuzione $P_2$ non si può accedere alla memoria degli altri processi; (c) cambiando il contenuto di `LINF` e `LSUP` cambia la zona di memoria accessibile.

aiutato dal compilatore). 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 compilatore. 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 può assumere un valore di default, per non costringere il programmatore a specificare questa dimensione nei casi più comuni.

Sapendo quanto è grande ogni processo, il sistema può caricarne in memoria

più di uno, come in Figura 2(a).

## 1.2 Isolamento tra i processi

Il secondo problema che dobbiamo risolvere è: come impedire che lo stato dei processi non in esecuzione venga letto o modificato dal processo in esecuzione? Per il momento abbiamo previsto nella CPU un meccanismo che protegge la memoria M1, ma non la memoria M2. Ricordiamo in cosa consiste questo meccanismo:

- abbiamo introdotto un registro, scrivibile solo da livello sistema, che contiene un indirizzo *limite*;
- 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 del limite, sollevando una eccezione in caso contrario.
- il registro limite viene inizializzato con l'ultimo indirizzo di M1 all'avvio del sistema.

Si vede che questo meccanismo è efficace nel proteggere M1, ma non impedisce in alcun modo al processo in esecuzione di accedere ovunque voglia nella memoria M2, dunque anche nelle parti che contengono lo stato degli altri processi.

Possiamo pensare di aggiungere altri due registri alla CPU, chiamiamoli `LINF` (limite inferiore) e `LSUP` (limite superiore). Nel descrittore di ogni processo prevediamo spazio anche per questi due nuovi registri, `contesto[I_LINF]` e `contesto[I_LSUP]`.

- I due nuovi registri sono scrivibili solo da livello sistema.
- Quando si trova a livello utente, la CPU controlla che ogni accesso in memoria abbia un indirizzo compreso tra `LINF` e `LSUP` (e solleva una eccezione in caso contrario).
- 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, si aggiorna anche il contenuto di `LINF` e `LSUP` con i valori presi dal descrittore del processo entrante.

Si vede come questi due registri risolvono il problema: ogni processo non può accedere al di fuori della propria zona di memoria (Figura 2(b) e (c)). Questi due registri rendono anche inutile il registro limite che avevamo introdotto prima: a maggior ragione, i processi utente non possono accedere alla memoria M1.

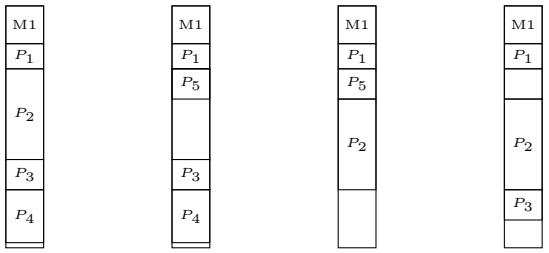


Figura 3: (a) ultimo stato dalla Figura 2; (b) $P_2$ viene temporaneamente rimosso per far posto a $P_5$; (c) $P_3$ e $P_4$ vengono temporaneamente rimossi per far posto a $P_2$; (d) viene ricaricato $P_3$ e $P_5$ termina: ora non c'è più posto per caricare $P_4$.

## 1.3 Caricamento a indirizzi variabili

Il sistema così realizzato presenta però degli svantaggi. Prima tutti i processi venivano sempre caricati a partire dallo stesso indirizzo (l'inizio 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.

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.

Per rimediare a questo problema modifichiamo 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. 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 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.

### 1.4 Condivisione e frammentazione

Il meccanismo a cui siamo arrivati ha ancora due svantaggi.

- Come facciamo se vogliamo che due o più processi condividano della memoria (che è proprio ciò che vogliamo nel nostro sistema)?
- 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?

Il secondo problema si potrebbe risolvere ricompattando lo spazio, ma per farlo dobbiamo copiare la memoria dei processi da una zona all'altra, operazione molto costosa. Entrambi i problemi, invece, 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.

## 2 Paginazione

Un meccanismo che ci permette di risolvere tutti i problemi che abbiamo illustrato è quello della *paginazione*. L'idea è di considerare tutti i possibili indirizzi generabili da un processo e di raggrupparli in regioni dette "pagine" e di applicare una traduzione di indirizzi diversa per ogni pagina. In questo modo il

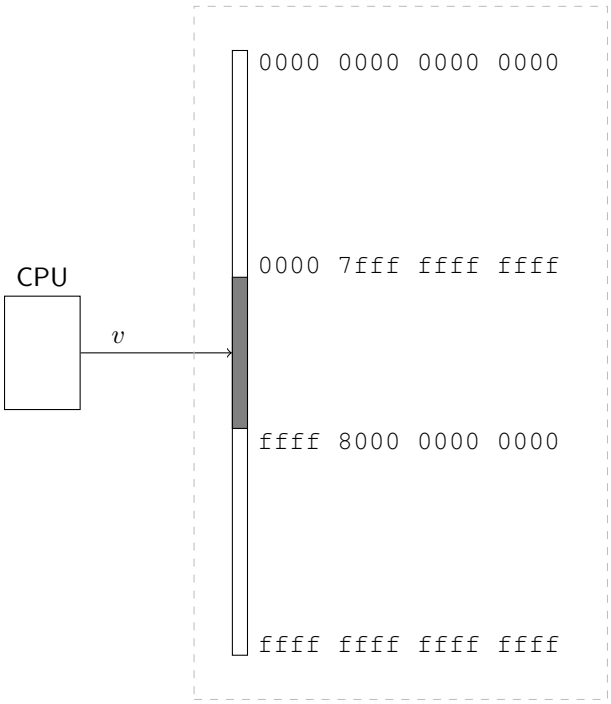


Figura 4: Spazio di indirizzamento nei processi Intel/AMD a 64 bit. Si ricordi che, per queste macchine, solo i 48 bit meno significativi di un indirizzo di 64 bit possono assumere un valore qualsiasi, mentre i 16 bit più significativi devono essere tutti uguali al bit n. 47 (contando da 0). La parte in grigio non è dunque utilizzata. Il processore genera una eccezione se si tenta di usare un indirizzo che rientra in questa parte.

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.

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.

Suddividiamo idealmente lo spazio di indirizzamento di Figura 4 in regioni[1], che chiamiamo *pagine*. Per realizzare la traduzione suddividiamo anche la memoria del sistema in regioni, 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

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

fisico può essere scomposto in $(n, o')$, dove $n$ è un *numero di frame* e $o'$ un offset all'interno del frame.

Il meccanismo della paginazione ci permette di caricare qualunque pagina in qualunque frame. Per farlo si introduce una struttura dati che, dato un numero di pagina $p$, ci permette di conoscere il numero di frame $n$ in cui $p$ è caricata. La più semplice struttura dati possibile è un array indicizzato dal numero di pagina: se chiamiamo `a` questo array, il numero di frame che cerchiamo è $n = \texttt{a}[p]$. 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, della memoria virtuale. Supponiamo ora che la CPU generi un indirizzo $v$. 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 $n = \texttt{a}[p]$, 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 $(\texttt{a}[p], o)$.

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 risolvere quasi tutti i problemi da cui eravamo partiti.

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

- *Caricamento a indirizzi variabili.* 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, visto che ogni processo ha la sua memoria virtuale indipendente, gli indirizzi di collegamento possono essere scelti liberamente senza problemi di collisione.

- *Condivisione.* 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.

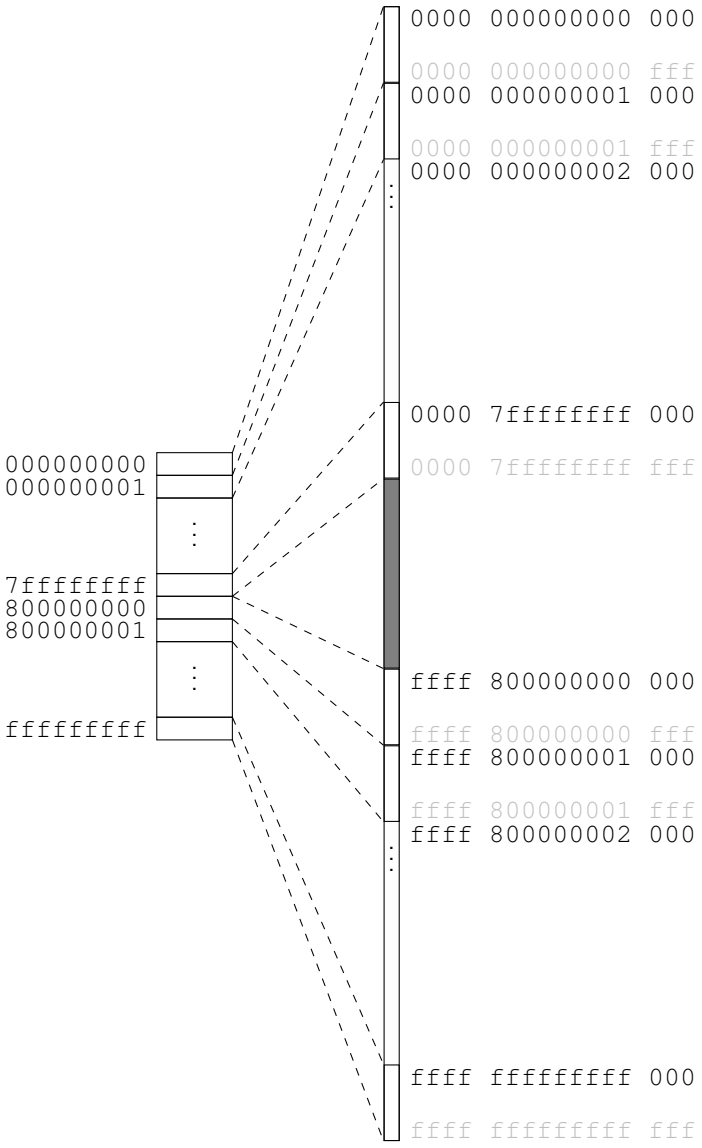


Figura 5: Tabella di corrispondenza e memoria virtuale. La tabella di corrispondenza è sulla sinistra, con a fianco gli indici (in esadecimale) delle sue entrate. Sulla destra è rappresentata la memoria virtuale, con gli indirizzi iniziali e finali delle pagine. Tutti gli indirizzi che cadono dentro una pagina sono tradotti usando l'entrata collegata tramite le linee tratteggiate. Si noti che, per il modo in cui sono definiti gli indirizzi possibili nell'architettura Intel/AMD a 64 bit, quando si passa dall'elemento di indice (in base 16) `7fffffff` a quello di indice `800000000` si salta dalla pagina virtuale di indirizzo `0000 7fffffff 000` a quella di indirizzo `ffff 800000000 000`.

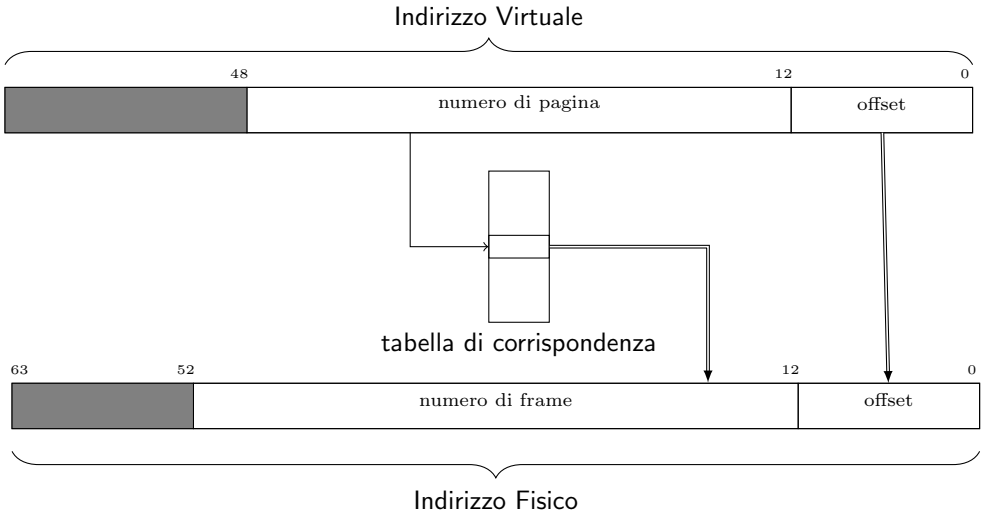


Figura 6: Traduzione da indirizzo virtuale a fisico. Si noti come in questa architettura il numero di pagina è dato solo dai bit 12–47 dell'indirizzo virtuale, mentre il numero di frame è più grande e occupa i bit 12–51 dell'indirizzo fisico.

- *Frammentazione.* Ogni pagina può essere caricata ovunque ci sia un frame libero, non è più necessario caricare un intero processo in una porzione contigua della memoria.

Per il momento, però, 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. Per il momento introduciamo 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.

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[2]. 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[3]. 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 memoria virtuale è naturalmente divisa in due (Figura 4) e riserviamo la parte superiore al sistema e la parte inferiore all'utente[4]. 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, infine, 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.

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

[3] 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.

[4] Linux e FreeBSD fanno al contrario.

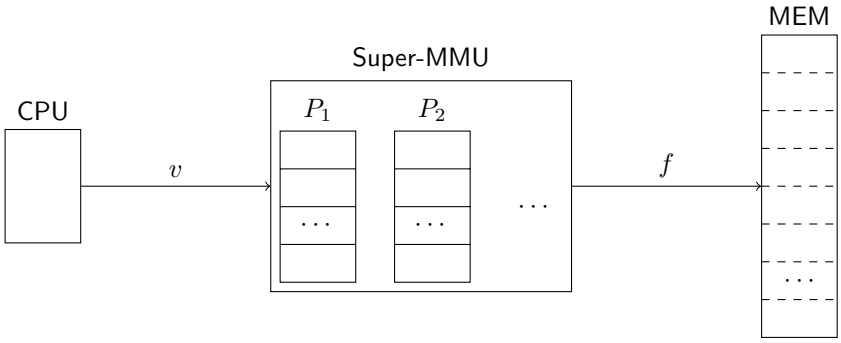


Figura 7: Traduzione degli indirizzi con una Super-MMU. Quando è in esecuzione il processo $P_i$, la Super-MMU traduce gli indirizzi usando la corrispondente tabella.

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.

# 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 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.