Memoria virtuale (prima parte)

G. Lettieri

24 e 28 Marzo 2017

La *memoria virtuale* è una tecnica tramite la quale si simula il funzionamento di un sistema di elaborazione in cui tutta (o quasi) la memoria principale indirizzabile dal processore è disponibile al programmatore, indipendentemente dalla dimensione della memoria principale realmente installata. In questo modo il programmatore può scrivere i suoi programmi senza preoccuparsi del fatto che la memoria a disposizione potrebbe non contenerli: tramite la memoria virtuale un programma troppo grande può girare lo stesso, anche se ad una velocità ridotta. Il meccanismo è inoltre fatto in modo tale che i programmi possano automaticamente beneficiare della disponibilità di nuova memoria (per esempio, installata successivamente) senza dover essere modificati.

Nel caso dei processori Intel/AMD a 64 bit si vuole simulare il funzionamento di una macchina come quella di Fig. 1, dove si mostra solo il collegamento tra la CPU e la memoria tramite il bus degli indirizzi. 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). Anche con questa limitazione gli indirizzi possibili sono $2^{48} = 256\,\text{TiB}$, ben al di là della capacità della memoria RAM comunemente disponibile su un singolo sistema.

L'idea è di simulare la macchina di Fig. 1 sostituendo la parte dentro le linee tratteggiate con un sistema che sia economicamente e tecnologicamente realizzabile ma che, visto dall'esterno, sia funzionalmente equivalente alla memoria di Fig. 1. Il sistema è molto complicato e lo studieremo per passi, introducendo delle semplificazioni che poi elimineremo via via. Questo ci permetterà di focalizzarci sugli aspetti più importanti all'inizio ed esaminare i dettagli solo in seguito.

1 Super-MMU

In Fig. 2 introduciamo la prima versione del sistema. La parte tratteggiata di Fig. 1 è stata sostituita con un sistema composto da una normale memoria principale (MEM), una memoria secondaria (denominata "area di swap", o semplicemente Swap) e un nuovo componente che chiamiamo Super-MMU (Super Memory Management Unit). L'idea principale è di utilizzare lo Swap per

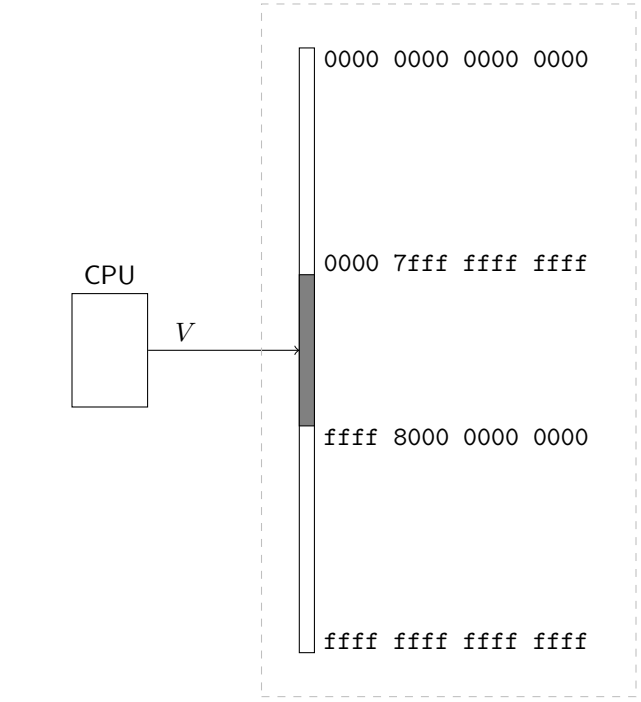

Figura 1: Macchina virtuale.

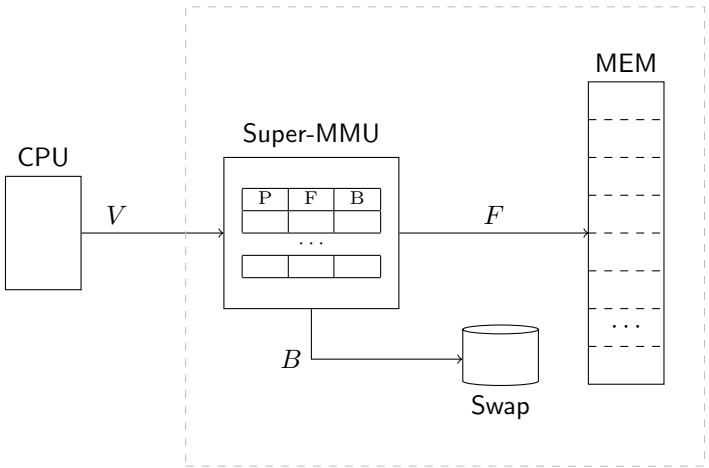

Figura 2: Implementazione della macchina di Fig. 1 con una Super-MMU.

memorizzare tutto il contenuto della memoria della macchina di Fig. 1 e di copiare le informazioni dentro MEM quando sono richieste. La necessità di avere anche MEM deriva dal fatto che la memoria secondaria non è indirizzabile con la granularità richiesta dalla CPU: la CPU può accedere a linee di 8 byte e loro sottoinsiemi (contigui), mentre la memoria secondaria può trasferire soltanto blocchi, tipicamente di 512 byte ciascuno. Si rende dunque necessario trasferire i blocchi in MEM per poi poter accedere all'informazione richiesta dalla CPU. Per ridurre l'overhead, conviene comunque eseguire questi trasferimenti in unità anche più grandi dei blocchi: chiameremo queste unità *pagine*. Per il momento assumeremo che ogni pagina abbia una dimensione di 4 KiB (pari a 8 blocchi da 512 byte).

Lo scopo della Super-MMU è di fare in modo che la CPU non possa distinguere tra la situazione in Fig. 2 e quella in Fig. 1. Questo è reso possibile (entro certi limiti), dalle seguenti considerazioni:

1. la CPU può accedere alla memoria soltanto tramite operazioni di lettura e scrittura;

2. la corretta esecuzione di molti programmi non dipende dal tempo che passa tra una operazione elementare e la successiva.

L'idea è dunque di intercettare tutte le operazioni di lettura e scrittura eseguite dalla CPU, trasferire dallo Swap in MEM la pagina che contiene l'informazione richiesta, e infine completare l'operazione. Nel caso di una operazione di scrittura sarà anche necessario riscrivere la pagina sullo Swap al termine dell'operazione. La Super-MMU fa in modo che tutto ciò appaia alla CPU come una normale operazione sulla memoria. Per una operazione di lettura, per esempio, la CPU produce l'indirizzo e si vede recapitare il contenuto della locazione indirizzata. L'unica differenza è che l'operazione avrà richiesto un tempo molto più lungo del normale, ma questo non è importante per la corretta esecuzione di molti programmi.

Non siamo obbligati ad eseguire trasferimenti dallo Swap per ogni operazione di lettura e scrittura della CPU. Conviene invece utilizzare MEM come una cache per lo Swap, ovvero: mantenere una copia delle pagine caricate, nel caso venissero richieste di nuovo, e ricopiarle nello Swap solo quando devono essere rimpiazzate per far posto a nuove pagine richieste dalla CPU (politica *write-back*). Per ogni possibile indirizzo che la CPU può emettere la Super-MMU deve essere in grado di sapere se la pagina che lo contiene è già caricata in MEM, e dove, oppure se si trova nello Swap, e dove. Se la pagina si trova nello Swap, la Super-MMU deve prima caricarla in MEM, eventualmente al posto di qualche altra pagina (che dovrà allora essere prima riscritta in Swap). Quando la pagina si trova in MEM (perché vi era già o perché vi è stata appena caricata), la Super-MMU dovrà accedere all'interno della pagina per completare l'operazione richiesta dalla CPU. Per eseguire l'accesso dovrà effettuare una operazione di lettura o scrittura in MEM, del tutto analoga all'operazione richiesta dalla CPU, ma ad un indirizzo diverso dipendente dall'indirizzo a cui la pagina è stata caricata. Possiamo dunque interpretare questa parte delle operazioni della

Super-MMU come una *traduzione* di indirizzo da V (l'indirizzo generato dalla CPU) a un nuovo indirizzo F, che è quello usato per accedere alla vera memoria. L'indirizzo V è detto *virtuale* mentre l'indirizzo F è detto *fisico*:

- gli **indirizzi virtuali** fanno riferimento alla memoria della macchina virtuale di Fig. 1; sono gli indirizzi utilizzati dal programmatore (e dagli strumenti che usa per produrre il programma, in particolare il collegatore);
- gli **indirizzi fisici** sono quelli utilizzati (in particolare dalla Super-MMU) per accedere a MEM, la memoria fisicamente installata sul sistema; non sono direttamente accessibili al programmatore.

Se la dimensione delle pagine è una potenza di due (come stiamo assumendo) e abbiamo cura di caricare le pagine solo ad indirizzi multipli della loro dimensione (allineamento naturale), alcune delle operazioni della Super-MMU si semplificano. Grazie al fatto che la pagina ha una dimensione che è una potenza di 2, gli indirizzi che arrivano dalla CPU possono essere scomposti in due parti: una parte più significativa che identifica la pagina richiesta (*numero di pagina virtuale*) e una parte meno significativa che identifica l'offset, all'interno della pagina, della locazione richiesta. L'offset avrà un numero di bit pari al logaritmo in base 2 della dimensione della pagina: 12 bit, nel nostro caso. La Super-MMU può avere una struttura dati (*tabella di corrispondenza*) indicizzata dal solo numero di pagina virtuale. In Fig. 2 tale struttura dati è rappresentata da una tabella interna alla Super-MMU che associa ad ogni possibile numero di pagina le seguenti informazioni:

- Un bit **P** (Presente) che vale 1 se la pagina è già caricata in MEM, 0 altrimenti;
- Un campo **F** (indirizzo Fisico) che contiene l'indirizzo a cui la pagina è caricata (se P vale 1, altrimenti il contenuto del campo non è significativo);
- Un campo **B** (numero di Blocco) che contiene il numero del primo blocco dello Swap che contiene la pagina (sufficiente a individuarla, assumendo che la pagina sia memorizzata in blocchi consecutivi).

Se una pagina è caricata all'indirizzo F, l'informazione richiesta dal processore si troverà all'indirizzo $F + \mathit{offset}$. Se carichiamo sempre le pagine ad indirizzi multipli della loro dimensione, però, i 12 bit meno significativi di F saranno sempre 0 e non sarà necessario eseguire una vera somma: basterà concatenare i bit più significativi di F (dal 12 in poi) con quelli dell'offset. In altre, parole, anche l'indirizzo fisico risulterà scomposto in due parti: un *numero di pagina fisica* e un offset. Il campo F della tabella di corrispondenza può contenere soltanto il numero di pagina fisica e la traduzione eseguita dalla MMU consisterà nel sostituire il numero di pagina virtuale con il numero di pagina fisica letto dalla tabella di corrispondenza, lasciando l'offset inalterato.

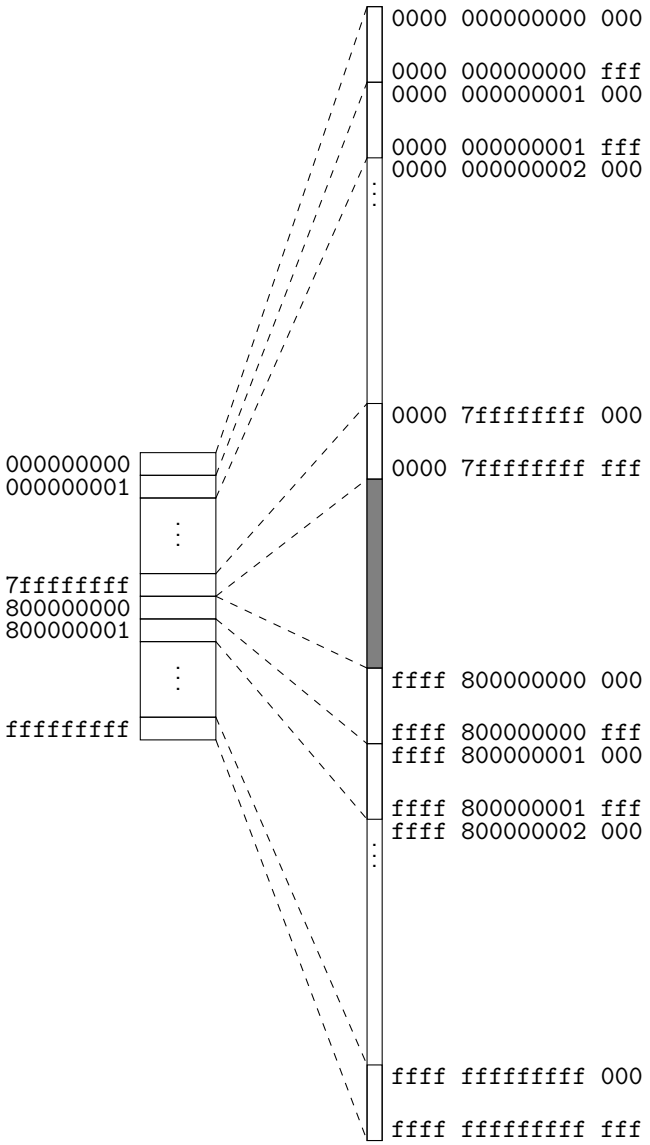

Figura 3: Tabella di corrispondenza e memoria virtuale.

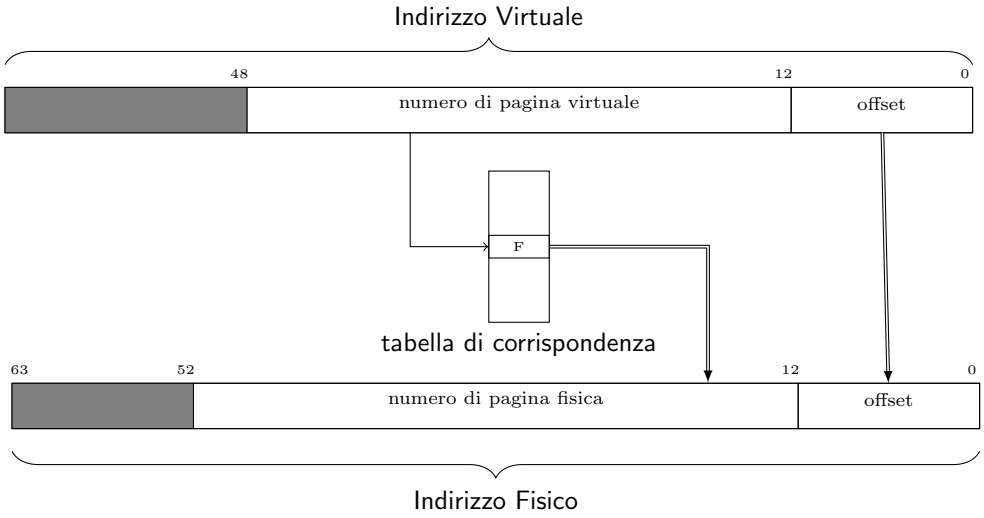

Figura 4: Traduzione da indirizzo virtuale a fisico.

1.1 Scomposizione degli indirizzi virtuali e fisici

In Fig. 3 si mostra come la tabella di corrispondenza della Super-MMU sia un vettore in cui ogni elemento, in ordine, si occupa della traduzione della corrispondente pagina, in ordine, della memoria virtuale. 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 che inizia all'indirizzo `0000 7fffffff 000` a quella che inizia a `ffff 800000000 000`. Il numero di pagina virtuale è dato soltanto dai bit che vanno da 12 a 47, mentre i 16 bit più significativi sono ignorati. Il numero di pagina virtuale, per la Super-MMU, è direttamente l'indice di un elemento della tabella di corrispondenza. I rimanenti 12 bit meno significativi compongono l'offset all'interno della pagina. Si ricordi, inoltre, che anche gli indirizzi fisici dell'architettura Intel/AMD a 64 bit non utilizzano tutti i bit a disposizione, ma soltanto 52. Il campo F, dunque, deve fornire soltanto i bit da 12 a 51 dell'indirizzo fisico. La traduzione da indirizzi virtuale a fisico avverrà dunque come illustrato in Fig. 4.

È disponibile un semplice esempio che illustra quanto detto finora. Uno degli scopi dell'esempio è di far vedere come la memoria virtuale sia completamente *trasparente* al programmatore, che si limita a scrivere un programma per una macchina virtuale simile a quella di Fig. 1 in cui non compare alcuna MMU (Super o meno), nessuna area di Swap, nessuna suddivisione della memoria in pagine, nessuna traduzione di indirizzi. Tutte queste cose servono all'implementazione della memoria virtuale e non sono accessibili al normale programmatore.

1.2 Strutture dati aggiuntive

Nell'esempio si vede che la Super-MMU ha bisogno di almeno una struttura dati (chiamata Free nell'esempio) per sapere quali pagine fisiche sono libere o occupate. Altre strutture dati non sono necessarie, ma sono utili per migliorare le prestazioni:

- un campo **D** (Dirty), di 1 bit, da aggiungere ai campi P, F e B della tabella di corrispondenza;
- un contatore delle statistiche di accesso alle pagine, che può essere un ulteriore campo della tabella.

La Super-MMU può usare il bit D per ricordare se vi sono mai state scritture sulla pagina corrispondente (basta resettarlo quando la pagina è caricata e settarlo alla prima scrittura). In questo modo può evitare di riscrivere nello Swap le pagine che non sono state modificate mentre erano caricate in MEM, in quanto la copia che si trova nello Swap va ancora bene.

Il contatore delle statistiche, invece, dovrebbe essere usato per fare scelte intelligenti al momento in cui MEM è piena e la Super-MMU deve selezionare una pagina (detta pagina *vittima*) da sovrascrivere con quella che deve essere invece caricata. L'idea è che la Super-MMU dovrebbe accumulare delle statistiche ad ogni accesso alla pagina. Quali statistiche raccogliere dipende da quale strategia la Super-MMU deve adottare per selezionare la vittima. Supponiamo che voglia selezionare la pagina che ha ricevuto meno accessi tra tutte quelle presenti (strategia LFU, per Least Frequently Used): sarà allora sufficiente contare gli accessi (se il contatore arriva al massimo si smette di sommare). Al momento della selezione della vittima, la Super-MMU sceglierà la pagina con il valore minimo del contatore.

2 MMU_1: software di sistema

I principali difetti della Super-MMU sono di essere troppo costosa, inflessibile, e di sprecare molte risorse. Possiamo però distinguere le operazioni svolte dalla Super-MMU in due:

1. la traduzione da indirizzo virtuale a fisico, operata quando la pagina virtuale acceduta dalla CPU è presente;
2. tutte le operazioni svolte quando la pagina è invece assente (caricamento della pagina virtuale, eventualmente al posto di un'altra che deve invece essere ricopiata nello Swap, e aggiornamento della tabella di corrispondenza per riflettere la nuova situazione).

In Figura 5 vediamo un esempio di svolgimento nel tempo delle varie operazioni, sia per il caso 1 che per il caso 2. Partendo dall'alto, vediamo la CPU che tenta di eseguire una operazione di lettura all'indirizzo virtuale con numero di pagina V_1 e offset o_1. La pagina è presente (caso 1) e la Super-MMU trasforma

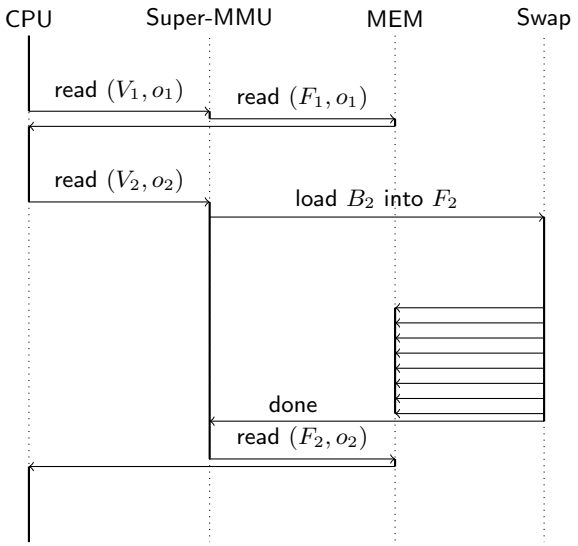

Figura 5: Attività dei vari componenti nel tempo (Super-MMU).

l'operazione in una lettura allo stesso offset, ma al numero di pagina fisica F_1, che poi la memoria completa. Dopo un po' di tempo la CPU tenta una operazione di lettura all'indirizzo (V_2, o_2), ma questa volta la pagina è assente (caso 2) e la Super-MMU deve prima ordinarne il caricamento dallo Swap per poi trasformare l'accesso come nel caso 1. Nell'esempio si assume che F_2 fosse già libera, altrimenti la Super-MMU avrebbe dovuto anche ordinare il salvataggio del precedente contenuto di F_2 nello Swap.

Le operazioni svolte nel caso 1 (pagina virtuale già presente) sono ragionevolmente semplici, mentre quelle del caso 2 sono più complesse: si deve programmare l'hard disk per eseguire il trasferimento e, nel caso in cui la memoria fisica sia piena, scegliere quale pagina vittima rimpiazzare con quella richiesta dalla CPU. La scelta della vittima, in particolare, è una operazione empirica per cui non esistono soluzioni che siano ottime in tutti i casi e al tempo stesso siano realizzabili praticamente: fissare un particolare algoritmo di scelta nell'hardware della Super-MMU renderebbe il sistema molto poco flessibile. Le operazioni del caso 2 sono le tipiche operazioni che si implementano meglio in software. Notiamo che mentre la Super-MMU sta svolgendo le operazioni del caso 2, la CPU è ferma ad aspettare il completamento dell'ultima operazione di accesso alla memoria ed è dunque inutilizzata: potremmo allora utilizzarla per farle eseguire un software che implementa le operazioni del caso 2, semplificando in questo modo la Super-MMU. (La CPU è ferma anche mentre aspetta il completamento delle operazioni nel caso 1, ma in questo caso il tempo di attesa è di poco superiore a quello di una comune operazione in memoria).

L'idea è dunque di avere una MMU, che chiameremo MMU_1, che svolge in hardware soltanto le seguenti operazioni:

- ogni volta che la CPU inizia un nuovo accesso alla memoria, all'indirizzo V, la MMU_1 lo intercetta, lo scompone in numero di pagina virtuale e offset, usa il numero di pagina virtuale per accedere alla tabella di corrispondenza;
  - se trova P=1 traduce l'indirizzo in fisico e completa l'accesso per conto della CPU;
  - se trova P=0 solleva una *eccezione di page fault* verso la CPU e smette di tradurre gli indirizzi (lasciandoli passare inalterati) fino alla ricezione di un segnale di *fine fault* (ricevuto il quale riprende il funzionamento normale).

Dal canto suo, la CPU reagisce all'eccezione salvando in pila l'indirizzo dell'istruzione che stava eseguendo (che è quella che si trova all'indirizzo V o ha tentato di accedervi) e saltando all'inizio di una particolare routine, detta *routine di page fault*. Questa routine svolge tutte le operazioni del caso 2 della Super-MMU, più precisamente:

1. individua la posizione della pagina V nello Swap (supponiamo legga il valore B dal campo B);
2. sceglie una pagina fisica libera, sia F. Se non ve ne sono:
   (a) seleziona una pagina virtuale vittima, sia V';
   (b) ricopia V' nell'area di swap;
   (c) aggiorna la tabella di corrispondenza in modo che l'entrata relativa a V' contenga P=0;
   (d) usa come F la pagina fisica che conteneva V';
3. carica V da B in F;
4. aggiorna la tabella di corrispondenza in modo che l'entrata relativa a V contenga P=1 e il valore F nel campo F;
5. emette il segnale di fine fault e termina ritornando all'indirizzo salvato in pila alla ricezione dell'eccezione.

A questo punto la CPU riesegue l'istruzione che aveva causato il fault, riemettendo dunque l'indirizzo V, che sarà intercettato nuovamente dalla MMU_1. Questa seconda volta, però, la MMU_1 troverà P=1 nella tabella di corrispondenza e potrà completare l'accesso e far proseguire l'esecuzione del programma. In Figura 6 possiamo vedere cosa avviene ora nel tempo per lo stesso esempio considerato in Figura 5. La lettura di (V_1, o_1) si svolge come prima, mentre la lettura di (V_2, o_2) causa un page fault: la CPU passa ad eseguire la routine di page fault (in rosso) e la MMU_1 si disattiva completamente (spazio senza puntini). La routine di page fault svolge le stesse azioni che in Figura 5 erano

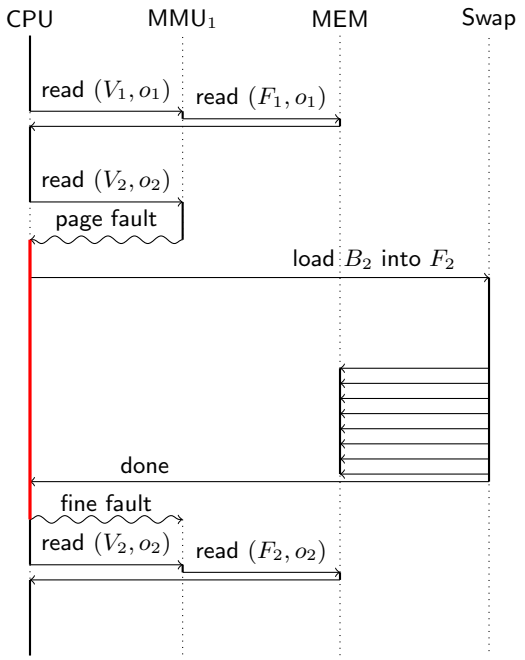

Figura 6: Attività dei vari componenti nel tempo (MMU$_1$).

svolte dalla Super-MMU. Dopo l'invio del segnale di fine fault la MMU$_1$ si riattiva e la CPU riesegue l'operazione di lettura a (V_2, o_2), che questa volta viene completata.

Per realizzare praticamente questa idea dobbiamo modificare ulteriormente la CPU:

- Al passo 1 la routine di page fault deve conoscere V, l'indirizzo che ha causato il fault. Introduciamo un registro speciale del processore, `cr2`, e modifichiamo la CPU in modo che, su ricezione dell'eccezione di page fault, salvi in `cr2` l'indirizzo V a cui stava tentando di accedere, prima di saltare alla routine di page fault. Aggiungiamo una nuova istruzione, `movq %cr2, %rax`, che la routine di page fault può eseguire per copiare l'indirizzo V in `rax` in modo che poi lo possa elaborare liberamente con le istruzioni già esistenti.

- Ai passi 2.(c) e 4 la routine di page fault deve poter modificare la tabella di corrispondenza della MMU$_1$. Invece di aggiungere nuove istruzioni specializzate, e anche in considerazione del fatto che la tabella di corrispondenza è essenzialmente un array, facciamo in modo che la tabella si trovi in memoria fisica. In questo modo la routine di page fault vi può accedere come ad una normale struttura dati. La MMU$_1$, invece, la deve raggiungere tramite accessi alla memoria fisica, e per far questo deve conoscere l'in-

dirizzo di partenza della tabella. A questo scopo prevediamo un nuovo registro, `cr3`, che contiene l'indirizzo della tabella in memoria fisica. La MMU_1 consulta questo registro per conoscere l'indirizzo della tabella, e il software può inizializzarlo tramite la nuova istruzione `movq %rax, %cr3`.

Poiché gli unici accessi all'area di swap sono via software (passi 2.(b) e 3), non c'è bisogno di collegare lo Swap direttamente alla MMU_1 e si può lasciarlo collegato al normale bus della CPU.

2.1 Descrittori di pagina virtuale e fisica

La tabella di corrispondenza è una struttura dati condivisa tra il software (le routine di inizializzazione e di page fault) e l'hardware (la MMU_1). In questi casi il formato della struttura dati è normalmente dettato dall'hardware, visto che il software si può più facilmente adattare a qualunque formato.

Nel caso di MMU_1, per avvicinarci alla MMU definitiva, prevediamo i seguenti campi per ogni entrata della tabella di corrispondenza:

- I campi **P** e **F** che già conosciamo;
- Il campo **D** introdotto in Sez. 1.2;
- Un campo **A**, di un 1 bit, che la MMU_1 pone a 1 se vi è stato un accesso (in lettura o scrittura) alla pagina corrispondente.

Si noti che manca il campo B, in quanto la MMU_1 non lo usa mai (i trasferimenti da e verso lo Swap sono operati esclusivamente dalla routine di page fault). Il campo A è nuovo, ma possiamo considerarlo una versione estremamente ridotta (un solo bit) del campo contatore introdotto in Sez. 1.2. I campi D ed A sono solo inizializzati dalla routine di page fault e poi scritti dalla MMU_1, e sono dunque informazioni che la MMU_1 raccoglie in modo che la routine di page fault possa utilizzarle. In particolare, al passo 2.(b), la routine di page fault evita di ricopiare V' nello Swap se vede che D vale 0. Gli altri campi sono solo letti dalla MMU_1: sono informazioni preparate dalla routine di page fault e usate dalla MMU_1 (e dalla routine stessa).

Il resto della funzionalità della Super-MMU deve essere riprodotto in software dalla routine di page fault. In particolare la routine dovrà predisporre delle strutture dati per:

1. sapere dove le pagine si trovano nell'area di swap;
2. sapere quali pagine fisiche sono libere o occupate;
3. mantenere delle statistiche più utili rispetto ai singoli bit A (come i contatori di Sez. 1.2).

Una struttura dati che permette di risolvere tutti questi problemi è un array di *descrittori di pagina fisica*, con un descrittore per ognuna delle pagine fisiche disponibili. Si noti che questa struttura è utilizzata esclusivamente dalla routine

di page fault, e quindi per essa non valgono le considerazioni sul formato imposto dall'hardware. Chi scrive la routine di page fault è libero di definirla come vuole, o anche di non definirla affatto e risolvere i problemi di cui sopra in qualche altro modo.

Per la routine di page fault associata a MMU_1 prevediamo un descrittore di pagina fisica con i seguenti campi:

- un flag **occupata**, che vale **true** se la pagina fisica contiene al momento una pagina virtuale; se il campo vale **false** la pagina fisica è libera e i campi successivi non sono significativi, altrimenti tutti i campi successivi si riferiscono alla pagina virtuale che sta occupando la pagina fisica;
- un campo **V** che contiene il suo numero di pagina virtuale;
- un campo **B** che contiene il numero del primo blocco della sua copia nello Swap;
- un campo **contatore** che contiene le statistiche dei suoi accessi.

Si noti che i descrittori di pagina fisica possono solo darci informazioni sulle pagine virtuali presenti (quelle, appunto, che in ogni momento si trovano caricate in qualche pagina fisica). È necessario però conoscere anche il valore di B per le pagine assenti, altrimenti non sarebbe possibile caricarle (passi 1 e 3 della routine di fault). Per queste si può utilmente sfruttare il fatto che la MMU_1 non usa il resto del descrittore di pagina virtuale quando trova P=0, quindi il campo B delle pagine assenti può essere memorizzato in quello spazio. Al passo 1, dunque, la routine di page fault accede all'entrata della tabella di corrispondenza relativa all'indirizzo virtuale V, vi trova ovviamente P=0 (altrimenti non ci sarebbe stato un page fault) e può leggere il valore B. Quando la pagina viene poi caricata e resa presente (passo 4), la routine di page fault copierà il valore B nel campo B del descrittore di F prima di scrivere nel campo F. Quando rende una pagina assente (passo 2.(c)) deve anche ricopiare il valore del campo B, preso dal descrittore di F, nel descrittore di pagina virtuale della vittima, in modo che la vittima possa essere ricaricata se il programma la richiede in seguito. Ricordare quale era il blocco da cui era stata caricata ogni pagina virtuale è utile nel caso in cui, quando la pagina virtuale viene scelta come vittima, si scopre che non è stata modificata (D=0 nel descrittore di pagina virtuale): allora è possibile riutilizzare la pagina che si trova già nello Swap, ma ovviamente bisogna sapere dov'è.

Il campo contatore viene aggiornato dalla routine di page fault in base ad un *campionamento* dei bit A delle pagine virtuali presenti. La routine scorre tutta la tabella di corrispondenza esaminando tutti i bit A che trova ad 1, aggiorna quindi i relativi contatori (sono i contatori contenuti nel descrittore della pagina fisica in cui la pagina virtuale è contenuta, facilmente individuabile dal campo F del descrittore di pagina virtuale), e poi azzera i bit A. Azzerare i bit A è fondamentale, altrimenti non si potrebbe mai vedere se vi sono stati nuovi accessi dall'ultima scansione. Questa scansione può essere effettuata in

vari momenti, per esempio periodicamente. Un modo semplice è di effettuarla subito prima di scegliere una vittima (dunque al passo 2.(a)).

Il campo V, infine, serve alla routine di page fault per ritrovare il descrittore di pagina virtuale di V' al passo 2.(c). La selezione della vittima, infatti, passa da una scansione dei contatori, che sono però associati alle pagine fisiche. Una volta trovato il contatore minimo si è individuata la pagina *fisica* F che si vuole utilizzare, ma per rimuovere la pagina virtuale in essa contenuta è necessario sapere il suo numero di pagina virtuale (V'), a meno di non voler scorrere tutta la tabella di corrispondenza alla ricerca del descrittore di pagina virtuale che contiene il valore F nel campo F.

2.2 Livelli di protezione

Chiediamoci ora che relazione c'è tra la routine di page fault e il programma originario. Si tratta in entrambi i casi di software eseguito dalla CPU, ma le due situazioni sono molto diverse. Dobbiamo infatti considerare la routine di page fault come una estensione dell'hardware: si tratta di una implementazione in software di una parte della Super-MMU che sarebbe troppo costoso realizzare in hardware. Come la Super-MMU era invisibile e inaccessibile al programma originario, così vogliamo che sia anche la coppia MMU_1/routine di page fault. Il programma originario non deve poter modificare il codice della routine di page fault, o accedere alla tabella di corrispondenza, così come non poteva modificare le operazioni della Super-MMU. Il motivo per cui vogliamo tutto ciò può non essere chiaro ora, ma lo diventerà quando scopriremo che la memoria virtuale serve anche ad isolare i programmi tra loro. Per il momento diamo per scontato che vogliamo avere una stretta equivalenza tra la Super-MMU e la coppia MMU_1/routine di page fault, se non altro per continuare a garantire la trasparenza del meccanismo, in modo che il programmatore originario non si debba preoccupare dell'esistenza della memoria virtuale e non possa involontariamente causarne un malfunzionamento.

Abbiamo dunque ora due tipi di software in esecuzione sul nostro sistema: il software "normale", che chiameremo *software utente*, e un software che estende l'hardware, che chiameremo *software di sistema*. La CPU deve sapere ad ogni istante quale tipo di software sta eseguendo, in quando alcune operazioni devono essere vietate al software utente (per esempio, l'istruzione `movq %rax, %cr3`). Il metodo adottato prevede che la CPU si possa trovare in uno tra due stati: *stato utente* e *stato sistema*, come specificato da un suo registro interno CPL (Current Privilege Level, livello di privilegio corrente). All'accensione la CPU si trova in stato sistema, in modo che una routine di sistema possa inizializzare opportunamente le strutture necessarie alla memoria virtuale (in particolare inizializzare la tabella di corrispondenza e scriverne l'indirizzo in `cr3`), quindi eseguire una particolare istruzione (vedremo che si tratta dell'istruzione `iretq`) che porti il processore a livello utente e salti alla prima istruzione del programma utente. A quel punto il processore obbedirà alle istruzioni del programma utente, rifiutandosi però di eseguire istruzioni privilegiate come `movq %rax, %cr3`, accessi diretti all'area di Swap, o istruzioni che accedono ad CPL. Se il programma

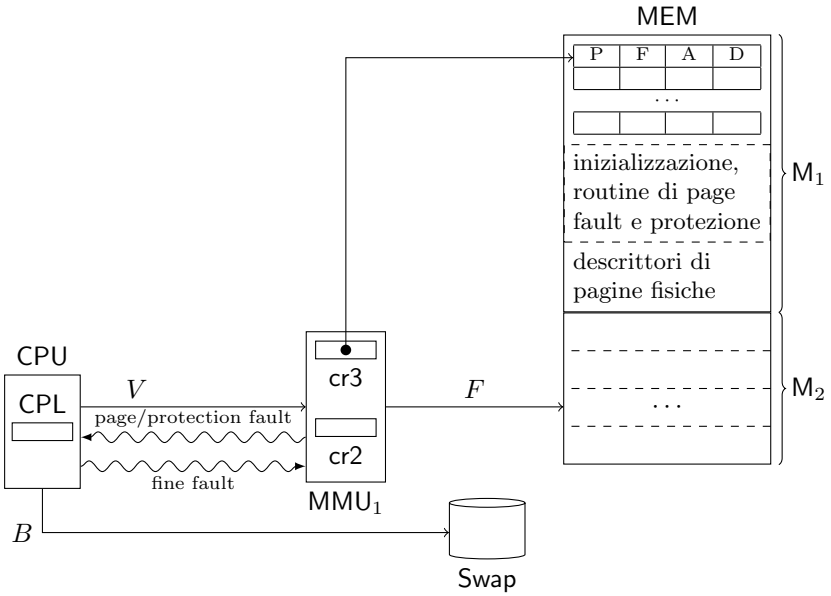

Figura 7: Implementazione della macchina di Fig. 1 con MMU_1.

utente dovesse tentare di fare queste cose, il processore genererà una *eccezione di protezione*. La ricezione di una eccezione (che nel nostro caso può, a questo punto, essere o di page fault o di protezione) riporterà il processore a livello sistema, saltando alla routine di page fault che conosciamo o ad una routine di protezione, a seconda del tipo di eccezione. La routine di protezione si limiterà a interrompere il programma stampando un messaggio di errore. La routine di page fault, invece, eseguirà tutte le operazioni che sappiamo. Essendo ora il processore a livello sistema, la routine di page fault avrà libero accesso a tutte le strutture della memoria virtuale. Al termine della routine di page fault si tornerà al programma utente, riportando al tempo stesso il processore al livello utente. Come per il caso analogo usando la Super-MMU, dal punto di vista del programma niente di tutto ciò è mai successo (se si ignora il passaggio del tempo): l'accesso in memoria si conclude come se la locazione acceduta fosse sempre stata lì.

Si noti che, per nascondere la parte software della memoria virtuale, non è sufficiente impedire al programma utente di eseguire alcune specifiche istruzioni. Dal momento che la routine di page fault e le sue strutture dati (tra cui la tabella di corrispondenza condivisa con la MMU_1) sono in memoria fisica, si deve anche impedire che il programma utente possa accedere alle locazioni che le contengono, indipendentemente dall'istruzione usata per farlo. Nel nostro caso, però, l'accesso a tutta questa parte di memoria è automaticamente impossibile: tutti gli indirizzi generati dal programma utente passano dalla MMU_1 che li traduce in indirizzi fisici dove si trovano pagine caricate dall'area di Swap (dove

si trovano solo le pagine virtuali del programma utente stesso) e mai ad indirizzi fisici dove si trovano le strutture della memoria virtuale. La situazione è quella illustrata in Fig. 7. Dividiamo la memoria fisica in una parte M_1, destinata a contenere tutto il codice e le strutture dati necessarie al meccanismo della memoria virtuale, e una parte M_2, che gioca ora il ruolo giocato da MEM in Fig. 2. La MMU_1 non traduce mai un indirizzo V in un indirizzo fisico che rientri in M_1, ma solo in M_2.

Allo stesso tempo, però, ci serve che si possa accedere alla parte M_1 quando il processore si trova a livello sistema, altrimenti non sarebbe neanche possibile prelevare le istruzioni della routine di page fault. Per questo motivo MMU_1 si disattiva quando si passa a livello sistema. Il ritorno a livello utente funge invece da segnale di fine fault, che riattiva la MMU_1. Questa disattivazione e riattivazione automatica della traduzione è una delle due semplificazioni adottate in MMU_1 rispetto alla MMU finale.

L'altra semplificazione di MMU_1 riguarda il fatto che la tabella di traduzione è un semplice array con una entrata per ogni possibile pagina virtuale, e per ora abbiamo evitato di chiederci quanto questo array dovrebbe essere grande.