Memoria virtuale (quarta parte)

G. Lettieri

23 Maggio 2017

In questa ultima parte vediamo una implementazione completa (anche se semplificata in alcuni punti) della memoria virtuale in un sistema multiprogrammato.

1 La memoria virtuale e i processi.

Abbiamo introdotto la memoria virtuale per risolvere un particolare problema, quello di come eseguire un programma più grande della memoria realmente installata sul sistema. Nel risolvere il problema siamo arrivati ad introdurre la MMU, che è in grado di fare due cose:

1. trasformare gli indirizzi usati dal programma, traducendoli da "virtuali" a "fisici";
2. generare una eccezione di page fault se tale traduzione non può essere completata.

La gestione del page fault è quella che ci ha permesso di risolvere efficacemente il problema dei programmi troppo grandi, creando l'illusione di una memoria più capiente di quella realmente installata. Ma, in un contesto multiprogrammato, è la *trasformazione* (o *traduzione*) degli indirizzi generati dal programma che si rivela essere la funzione più utile della MMU, al punto che il page fault diventa una funzione aggiuntiva, utile ma secondaria. Grazie alla traduzione operata dalla MMU possiamo assegnare uno *spazio di indirizzamento distinto* ad ogni processo. Ogni processo avrà la sua tabella di corrispondenza, in modo che il significato di un indirizzo dipenda dal processo che lo usa. Per realizzare questa idea è sufficiente cambiare il contenuto di `CR3` ogni volta che si cambia processo (nel nostro caso, il descrittore di processo avrà un campo anche per `CR3` e la `carica_stato` caricherà anche questo). Ogni processo ha dunque la sua "visione" della memoria. In generale, questo ci permette di

1. collegare ogni programma senza curarci dell'esistenza di altri programmi: al momento di caricare il programma possiamo sempre predisporre una memoria virtuale in cui tutti gli indirizzi sono disponibili, anche se la memoria (fisica) contiene già altri programmi (eseguiti da altri processi). In

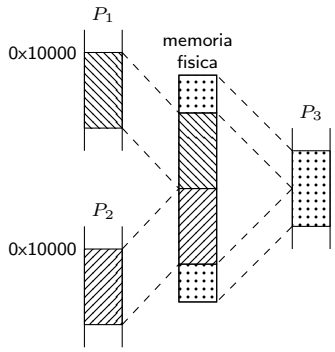

Figura 1: Memoria virtuale e processi.

Figura 1, i processi P_1 e P_2 usano entrambi gli indirizzi che partono da 0x10000. Se volessimo caricarli entrambi in memoria fisica, senza memoria virtuale, almeno uno dei due andrebbe rilocato. Con la memoria virtuale possiamo lasciarli immutati e far corrispondere i loro indirizzi a parti distinte della memoria fisica.

2. Ci permette inoltre di *isolare* i vari processi, in modo che gli errori di un programma non si propaghino in altri processi. Il processo P_1 in Figura 1 non può in alcun modo alterare la memoria del processo P_2 (e viceversa), in quanto nessuno dei suoi indirizzi raggiunge la memoria fisica utilizzata da P_2. Allo stesso tempo, il meccanismo è abbastanza flessibile da permetterci di condividere memoria tra i processi, se lo desideriamo.

3. Possiamo caricare un programma sfruttando tutto lo spazio libero in memoria, anche se questo non è contiguo (si veda il processo P_3 in Figura 1). Questo è reso possible dal fatto che ogni pagina è tradotta indipendentemente dalle altre.

Anche il meccanismo del page fault ha la sua utilità, una volta che lo abbiamo realizzato. Tramite il meccanismo del page fault le pagine dei processi vengono caricate solo quando sono riferite, eventualmente ripiazzandone altre (anche di altri processi). In questo modo otteniamo "gratis" il salvataggio e ripristino della memoria dei processi, che è l'altra parte dello stato (oltre ai registri del processore) che va salvata e ripristinata ogni volta che si cambia processo: quando saltiamo ad un nuovo processo cambiamo semplicemente `CR3` e lasciamo che sia il processo stesso a caricare le pagine che gli servono, accedendovi e causando i relativi page fault. Inoltre, non eliminiamo le pagine del processo uscente nel momento in cui cambiamo processo: lasciamo che sia il meccanismo del rimpiazzamento a decidere se e quando devono essere eliminate. In questo modo, se c'è spazio a sufficienza per tutti, ogni processo caricherà le sue pagine e queste resteranno in memoria; ogni processo le ritroverà, senza causare altri page fault, ogni volta che viene messo in esecuzione, e dunque il cambio di processo non ci costerà niente in termini di caricamenti/scaricamenti di pagine.

1.1 La memoria virtuale nel nucleo

Realizzeremo un caso ibrido, in cui i processi hanno sia zone di memoria condivise tra tutti, sia zone di memoria private per ciascuno. In compenso, tutti i processi avranno caricato nella propria memoria virtuale un unico programma: quello contenuto nel modulo utente. Ogni processo, però, partirà eseguendo una funzione del programma che può anche essere diversa per ciascuno.

La memoria virtuale di ogni processo sarà organizzata come in Figura 2. La Figura mostra la memoria virtuale di due processi, P_1 e P_2, insieme al possibile contenuto della memoria fisica. La parte a sfondo grigio di ogni memoria virtuale è accessibile solo quando il processore si trova a livello sistema. La memoria virtuale di ogni processo è suddivisa nello stesso modo: la parte che va dall'indirizzo 0 all'indirizzo (esadecimale) 0000.7fff.ffff.ffff ($2^{47}-1$) è dedicata al sistema, mentre la parte che va da ffff.8000.0000.0000 a ffff.ffff.ffff.ffff è dedicata all'utente.

Ogni parte è suddivisa in ulteriori sezioni. Sulla sinistra della memoria virtuale di P_1 abbiamo mostrato alcune costanti (definite in `sistema/sistema.cpp`) che contengono gli indirizzi di inizio e fine delle varie sezioni. Per "indirizzo di fine" intendiamo il primo indirizzo che non fa parte della sezione: gli indirizzi di una sezione vanno da quello di inizio, incluso, a quello di fine, escluso. Il nome delle costanti è composto da tre parti: 1) la stringa "`ini`" per l'indirizzo di inizio o "`fin`" per l'indirizzo di fine; 2) la stringa "`sis`" per le sezioni *sistema*, "`mio`" per la sezione *modulo I/O* e la stringa "`utn`" per le sezioni *utente*; 3) il carattere "`c`" per le sezioni *condivise* o "`p`" per quelle private. Le sezioni condivise contengono le stesse traduzioni in tutti i processi.

Le sezioni della memoria virtuale di ogni processo sono elencate di seguito. Le sezioni indicate come "residenti" devono essere presenti in memoria fisica per tutto il tempo in cui i processi le utilizzano, mentre le pagine di quelle indicate come "non residente" possono essere caricate e scaricate dinamicamente dal meccanismo di page fault.

sistema/condivisa : contiene la finestra di memoria fisica. La dimensione della sezione è decisa *a priori*, mentre la dimensione della finestra dipende dalla memoria fisica installata e può essere più piccola (come illustrato in Figura). Il resto della sezione è lasciato inutilizzato.

sistema/privata (residente): contiene la pila sistema del processo. Si noti che questa pila è utilizzata sia dalle funzioni del modulo sistema, sia da quelle del modulo I/O, in quanto questo gira a livello sistema e ogni processo ha una sola pila di livello sistema.

IO/condivisa (residente): contiene il modulo I/O, vale a dire le sezioni `.text` e `.data` estratte dal file `build/io` e caricate al loro indirizzo di collegamento.

utente/condivisa (residente): contiene il modulo utente, vale a dire le sezioni `.text` e `.data` estratte dal file `build/utente` e caricate al loro indirizzo di collegamento. Contiene inoltre lo *heap utente*, vale a dire la zona

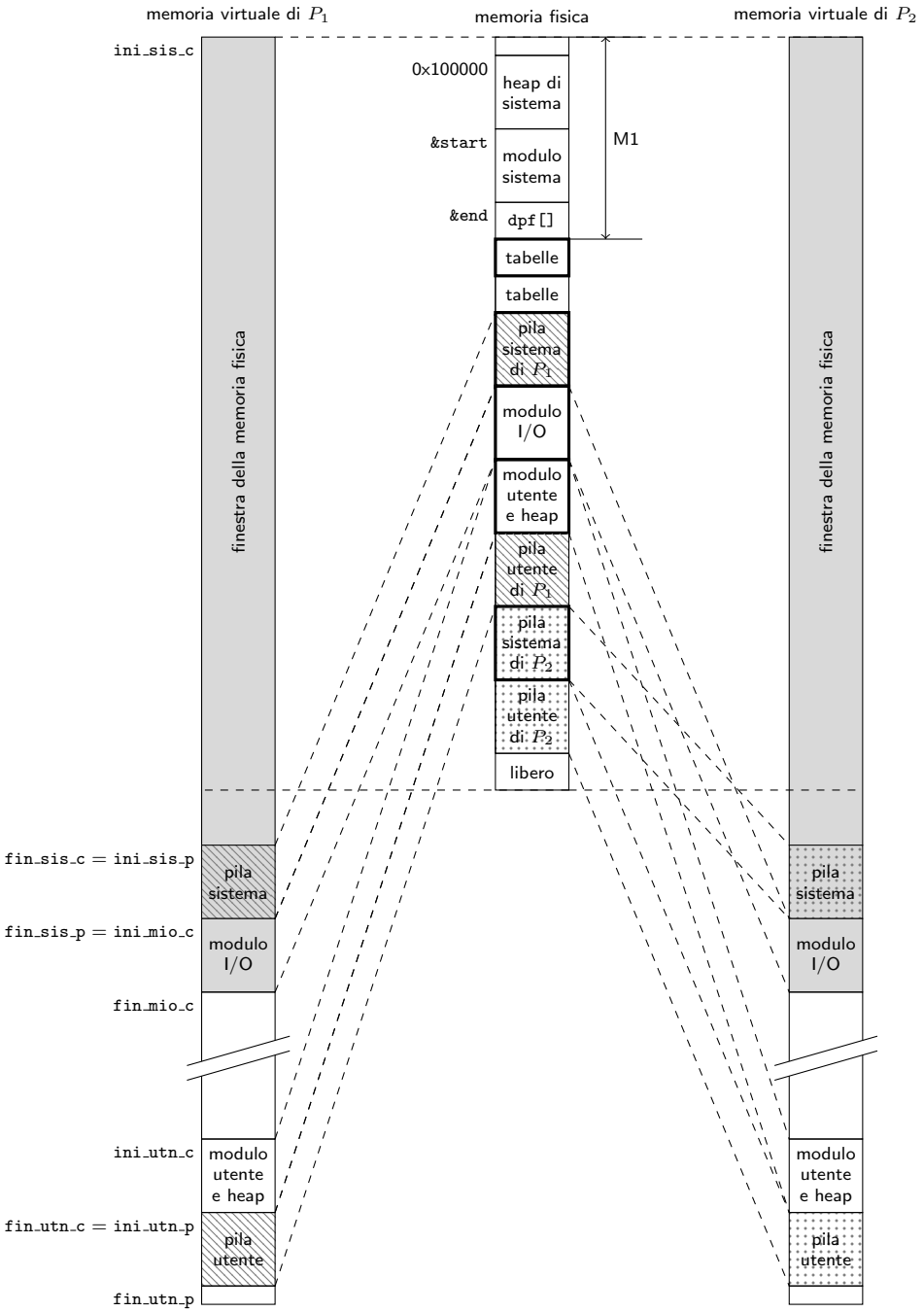

Figura 2: Esempio di memoria virtuale con due processi (figura non in scala).

di memoria su cui lavorano le funzioni `mem_alloc()` e `mem_free()` della libreria utente.

utente/privata (non residente): contiene la pila utente del processo.

Per quanto riguarda la finestra di memoria, non ha senso chiedersi se è residente o meno. Ricordiamo infatti che la finestra è solo un trucco per disabilitare di fatto la MMU mentre è in esecuzione il modulo sistema.

Si noti che la sola sezione che è normalmente residente in un sistema reale è la sistema/privata, in quanto contiene la pila sistema a cui accede il modulo sistema. Per semplicità di implementazione abbiamo deciso di rendere residenti anche tutte le sezioni condivise, ed è solo per questo che anche IO/condivisa e utente/condivisa sono residenti.

La memoria fisica è suddivisa, come al solito, in una parte M1 e una parte M2 (per motivi di spazio, solo la parte M1 è indicata esplicitamente in Figura; la parte M2 è quella rimanente). La suddivisione della parte M2 della memoria fisica in Figura 2 è puramente esemplificativa: i processi saranno in generale più di due; inoltre, se osserviamo il sistema ad un qualunque istante, potremo trovare le varie parti in un ordine qualsiasi e in pagine non contigue della memoria fisica; infine, nel caso di sezioni rimpiazzabili, alcune pagine potrebbero anche essere assenti.

La parte M1 della memoria fisica è organizzata nel modo seguente.

- Il primo MiB è riservato per ragioni storiche (parte degli indirizzi sono occupati da altre cose, come la memoria video in modalità testo, che si trova all'indirizzo 0xb8000).

- La parte che va dalla fine del primo MiB all'inizio del modulo sistema contiene lo *heap di sistema*. Si tratta di una zona di memoria in cui le primitive del modulo sistema allocano i `des_proc`, i `proc_elem`, le strutture `richiesta` (usate dalla primitiva `delay`), e in generale qualunque struttura dati che debbe essere allocata dinamicamente (tramite le funzioni interne `alloca()` e `dealloca()`).

- La parte dedicata al modulo sistema contiene più precisamente le sezioni `.text` e `.data` estratte dal file `build/sistema` e caricate al loro indirizzo di collegamento. La fine della parte precedente e l'inizio della successiva verranno ottenuti, a tempo di inizializzazione, dai due simboli `start` ed `end` definiti nel modulo sistema.

- L'ultima parte di M1 contiene l'array di descrittori di pagina fisica, `dpf`, che come sappiamo contiene una entrata per ogni pagina fisica di M2.

Le linee tratteggiate tra le memorie virtuali e la memoria fisica vogliono rappresentare la corrispondenza tra le sezioni della memoria virtuale e le parti di memoria fisica in cui sono tradotte. Le parti a sfondo bianco delle memoria fisica corrispondono a sezioni condivise tra tutti i processi. Si noti, per esempio, come le sezioni "modulo I/O" e "modulo utente e heap" corrispondano alla

stessa parte della memoria fisica sia per P_1 che per P_2. Le sezioni private, invece, usano traduzioni diverse in ciascun processo e corrispondono a parti diverse della memoria fisica: si veda per esempio la sezione "pila utente", che corrisponde a "pila utente di P_1" per il processo P_1 e "pila utente di P_2" per il processo P_2; un discorso analogo vale per "pila sistema". Si noti che alcune parti della memoria fisica sono accessibili esclusivamente tramite la finestra: in particolare, tutta la parte M1 e le pagine di M2 che contengono tabelle o sono vuote (la parte indicata con "libero" in Figura 2). Alcune parti di M2 sono acessibili sia dalla finestra, sia da altre sezioni. Ciò è dovuto semplicemente al fatto che la finestra permette di accedere a tutta la memoria fisica, indipendentemente dal contenuto, e dunque contiene traduzioni anche per tutte le pagine presenti della memoria virtuale di ogni processo. Le parti di M2 con il bordo spesso contengono entità che non possono essere rimpiazzate. Si noti come hanno il bordo spesso tutte le parti di memoria fisica che contengono sezioni che abbiamo indicato come residenti. Alcune tabelle non possono essere rimpiazzate e altre sì. In particolare, non possono essere rimpiazzate tutte le tabelle di livello 4 e tutte le tabelle che traducono gli indirizzi delle sezioni residenti.

2 Implementazione

In questa sezione esaminiamo le strutture dati e le funzioni che si trovano nel modulo sistema e implementano la memoria virtuale.

Nel nostro sistema l'area di swap occupa un intero hard disk (quello collegato al canale secondario/master). Quando lanciamo il nostro sistema sulla macchina virtuale, questo hard disk è simulato da un file sulla macchina ospite (per default, il file `swap.img` nella directory `nucleo`). In un sistema realistico è più probabile che l'area di swap sia solo una partizione di un hard disk.

L'implementazione contiene le seguenti ulteriori semplificazioni rispetto ad un sistema realistico.

- Gran parte dello stato iniziale della memoria virtuale di ogni processo viene preparato una volta per tutte con uno strumento esterno al sistema (il programma di utilità `creatimg`, che gira sul sistema Linux ospite). Ciò è possibile perché tutti i processi caricano gli stessi programmi: quelli contenuti in `build/io` e `build/utente`. Si noti comunque che, anche se questo modo di operare è lontano da ciò che accade in un sistema *general purpose* come Linux o Windows, può essere vicino a come fuzionano i sistemi specializzati.

- La routine di page fault gira con le interruzioni disabilitate ed esegue tutti i trasferimenti da e verso l'area di swap a controllo di programma (quindi attendendo attivamente che il bit DRQ del registro di stato del controllore ATA passi ad 1, indicando il completamento dell'operazione di I/O). In un sistema realistico, invece, la routine di fage fault gestirebbe l'I/O a interruzione di programma, bloccando il processo che ha causato il

```
1  struct des_pf {
2          int      livello;          // 0=pagina, -1=libera
3          bool     residente;        // pagina residente o meno
4          natl     processo;         // identificatore di processo
5          natl     contatore;        // contatore per le statistiche
6          natq     ind_massa;
7          union {
8                  addr     ind_virtuale;
9                  des_pf*  prossima_libera;
10         };
11 };
```

Figura 3: La struttura `des_pf`.

fault e schedulandone un altro prima di inizare ogni trasferimento. Questa semplificazione è di gran lunga la più importante: se non la facciamo l'implementazione della memoria virtuale diventa subito molto più complicata, in quanto a quel punto può essere generato un nuovo page fault quando ancora non abbiamo finito di gestire il precedente. Si noti che, a causa di questa semplificazione, il nostro sistema può tollerare un page fault in qualunque momento, anche nelle primitive di sistema e persino in un driver, in quanto il nostro page fault non riabilita le interruzioni e non salva lo stato. Per avvicinarci di più alla situazione reale, però, considereremo comunque un errore un page fault causato da codice del modulo sistema.

2.1 Strutture dati

Per ogni pagina fisica della parte M2 avremo un descrittore di pagina fisica del tipo `des_pf` definito in Figura 3. Oltre ai campi definiti quando abbiamo introdotto MMU_2, abbiamo aggiunto:

- `processo`: il processo a cui appartiene il contenuto della pagia fisica;
- `prossima_libera`: significativo nel caso in cui la pagina fisica descritta sia vuota; serve a creare una lista di tutte le pagine vuote.

Abbiamo bisogno del campo `processo` per poter trovare il descrittore di pagina o tabella che punta all'entità contenuta nella pagina: dato il processo possiamo accedere al suo descrittore, dove troviamo il campo `cr3` che punta alla tabella di livello 4 del processo; da questa, usando i campi `livello` e `ind_virtuale` possiamo raggiungere il descrittore che ci interessa.

La lista delle pagine libere serve ad allocare e deallocare pagine fisiche. La testa della lista è puntata dalla variabile globale `prima_libera` e all'avvio del sistema contiene tutte le pagine fisiche.

I descrittori di pagina fisica sono raccolti nell'array `dpf`, che viene allocato dinamicamente all'avvio del sistema, a partire dalla fine del modulo sistema (simbolo `end` definito in `build/sistema`; vedere anche Figura 2).

```
1  void c_routine_pf()
2  {
3          addr ind_virt = readCR2();
4
5          for (int i = 3; i >= 0; i--) {
6                  natq d = get_des(esecuzione->id, i + 1, ind_virt);
7                  bool bitP = extr_P(d);
8                  if (!bitP)
9                          swap(i, ind_virt);
10         }
11 }
```

Figura 4: La routine che risponde ai page fault (parte C++).

Le tabelle della memoria virtuale vengono manipolate come semplici array di `natq`. I singoli campi all'interno di ogni entrata vengono letti e scritti con operazioni sui bit.

2.2 Routine principali

Esaminiamo le routine principali per la gestione della memoria virtuale. Si tratta della traduzione in C++ di quanto abbiamo già detto a parole quando abbiamo studiato la versione finale della memoria virtuale.

Per avere una breve descrizione di cosa fanno le funzioni usate dalle routine princiali, si faccia riferimento all'elenco in fondo al documento.

2.2.1 Routine di page fault

La routine in Figura 4 va in esecuzione ogni volta che si verifica un page fault. Alla riga 3 legge dal registro CR2 l'indirizzo che la MMU non è riuscita a tradurre, quindi carica la pagina che lo contiene e tutte le tabelle necessarie per la traduzione (se non erano già presenti). Il caricamento di ciascuna di queste entità richiede le stesse azioni, quindi la routine non fa che eseguire un ciclo (linee 5–10) partendo dal livello 3 (le tabelle di livello 4 sono sempre presenti) fino al livello 0 (che per noi rappresenta le pagine). Per ogni livello preleva il corrispondente descrittore (linea 6), esamina il bit P (linea 8) e, se l'entità non è presente (linea 8), la carica (linea 9) utilizzando la funzione `swap()`, mostrata in Figura 5.

2.2.2 Routine di rimpiazzamento

La funzione `swap(liv, ind_virt)` in Figura 5 ha il compito di caricare l'entità (tabella o pagina) di livello `liv` relativa all'indirizzo virtuale `ind_virt`, assumendo che le entità di livello superiore siano già presenti in memoria fisica. Alla riga 5 prova ad allocare una pagina fisica vuota destinata a contenere l'entità da caricare. Si noti che le pagine fisiche vengono gestite tramite il loro descrittori. Se non vi sono pagine fisiche libere (riga 6) è necessario liberarne una scegliendo come *vittima* una delle entità che si trova in questo momento in memoria fisica

```
1  void swap(int liv, addr ind_virt)
2  {
3          // "ind_virt" e' l'indirizzo virtuale non tradotto
4          // carica una tabella delle pagine o una pagina
5          des_pf* nuovo_dpf = alloca_pagina_fisica_libera();
6          if (nuovo_dpf == 0) {
7                  des_pf* dpf_vittima =
8                          scegli_vittima(esecuzione->id, liv, ind_virt);
9                  // (* scegli_vittima potrebbe fallire
10                 if (dpf_vittima == 0) {
11                         flog(LOG_WARN, "memoria esaurita");
12                         in_pf = false;
13                         abort_p();
14                 }
15                 // *)
16                 bool occorre_salvare = scollega(dpf_vittima);
17                 if (occorre_salvare)
18                         scarica(dpf_vittima);
19                 nuovo_dpf = dpf_vittima;
20         }
21         natq des = get_des(esecuzione->id, liv + 1, ind_virt);
22         natl IM = extr_IND_MASSA(des);
23         // (* non tutto lo spazio virtuale e' disponibile
24         if (!IM) {
25                 flog(LOG_WARN,
26                    "indirizzo %p fuori dallo spazio virtuale allocato",
27                    ind_virt);
28                 rilascia_pagina_fisica(nuovo_dpf);
29                 in_pf = false;
30                 abort_p();
31         }
32         // *)
33         nuovo_dpf->livello = liv;
34         nuovo_dpf->residente = false;
35         nuovo_dpf->processo = esecuzione->id;
36         nuovo_dpf->ind_virtuale = ind_virt;
37         nuovo_dpf->ind_massa = IM;
38         nuovo_dpf->contatore  = 0;
39         carica(nuovo_dpf);
40         collega(nuovo_dpf);
41 }
```

Figura 5: La routine che carica una entità (pagina o tabella) assente.

```
1  des_pf* scegli_vittima(natl proc, int liv, addr ind_virtuale) //
2  {
3          des_pf *ppf, *dpf_vittima;
4          ppf = &dpf[0];
5          while ( ppf < &dpf[N_DPF] &&
6                  (ppf->residente || vietato(ppf, proc, liv, ind_virtuale)))
7                  ppf++;
8          if (ppf == &dpf[N_DPF]) return 0;
9          dpf_vittima = ppf;
10         stat();
11         for (ppf++; ppf < &dpf[N_DPF]; ppf++) {
12                 if (ppf->residente ||
13                     vietato(ppf, proc, liv, ind_virtuale))
14                         continue;
15                 if (ppf->contatore < dpf_vittima->contatore ||
16                     (ppf->contatore == dpf_vittima->contatore &&
17                      dpf_vittima->livello > ppf->livello))
18                         dpf_vittima = ppf;
19         }
20         return dpf_vittima;
```

Figura 6: La routine che seleziona una vittima in caso di rimpiazzamento.

(righe 7-8). Per liberare la pagina fisica occupata dalla vittima è necessario prima *scollegare* la vittima (riga 16), vale a dire porre a 0 il bit P nel descrittore di pagina o tabella la punta, ed eventualmente *scaricarla* (riga 17-18), vale a dire ricopiarla in memoria di massa, se necessario. Fatto ciò, possiamo riutilizzare la pagina fisica che la conteneva (riga 19).

Arriviamo alla riga 21 con il descrittore di una pagina fisica utilizzabile, sia se l'abbiamo trovata alla riga 5, sia se l'abbiamo ottenuta eliminando una vittima. A questo punto possiamo preoccuparci dell'entità da caricare. Alla riga 21 otteniamo il descrittore dell'entità, per poter estrarre il suo indirizzo in memoria di massa (riga 22). Quindi, alle righe 33–38 riempiamo i campi del descrittore con le informazioni relative all'entità da caricare: il suo livello (riga 33), il processo a cui appartiene (riga 35, si tratta evidentemente del processo che ha causato il page fault e che in questo momento è in esecuzione), l'indirizzo virtuale per la cui traduzione la stiamo caricando (riga 36), l'indirizzo in memoria di massa appena ottenuto (riga 37). Marchiamo anche l'entità come non residente (riga 34), dal momento che le entità residenti vengono caricate tutte all'avvio e non causano dunque page fault. Il contatore per le statistiche di utilizzo è inizialmente zero (verrà aggiornato alla prima chiamata della funzione `stat()`). Infine, possiamo caricare l'entità dalla memoria di massa nella pagina fisica (riga 39) e *collegarla* (riga 40), vale a dire fare in modo che l'entità di livello superiore punti all'entità appena caricata.

2.2.3 Routine di selezione vittima

La routine di selezione vittima di Figura 6, da chiamare solo quando tutte le pagine fisiche sono occupate, sceglie come vittima quella che ha il contatore minore. L'algoritmo è dunque una semplice ricerca lineare del minimo, con tre problemi da evitare:

```
1  void stat()
2  {
3          des_pf *ppf1, *ppf2;
4          addr ff1, ff2;
5          bool bitA;
6
7          for (natq i = 0; i < N_DPF; i++) {
8                  ppf1 = &dpf[i];
9                  if (ppf1->livello < 1)
10                         continue;
11                 ff1 = indirizzo_pf(ppf1);
12                 for (int j = 0; j < 512; j++) {
13                         natq& des = get_entry(ff1, j);
14                         if (!extr_P(des))
15                                 continue;
16                         bitA = extr_A(des);
17                         set_A(des, false);
18                         ff2 = extr_IND_FISICO(des);
19                         ppf2 = descrittore_pf(ff2);
20                         if (!ppf2 || ppf2->residente)
21                                 continue;
22                         ppf2->contatore >>= 1;
23                         if (bitA)
24                                 ppf2->contatore |= 0x80000000;
25                 }
26         }
27         invalida_TLB();
28 }
```

Figura 7: La routine che calcola le statistiche di utilizzo di pagine e tabelle.

1. non si possono scegliere le pagine marcate come residenti;
2. non si possono scegiere le pagine contenenti tabelle che si trovano nello stesso percorso di traduzione dell'entità che stiamo caricando;
3. non si possono scegliere tabelle che non siano "vuote".

Per il punto 2, la funzione riceve i parametri `proc`, `liv` e `ind_virtuale`, che identificano il percorso di traduzione a cui appartiene l'entità da caricare. La funzione `vietato(ppf, proc, liv, ind_virtuale)` restituisce `true` se la pagina fisica descritta da `ppf` contiene una tabella appartenente allo stesso percorso (per farlo controlla i bit opportuni del campo `ind_virtuale` del descrittore).

Nelle righe 4–7 la funzione cerca un primo valore da usare come minimo di partenza, alla riga 9. Alla riga 10 chiama la funzione che aggiorna tutte le statistiche, quindi scorre tutti i rimanenti descrittori di pagina fisica (righe 11–19) alla ricerca di quello con il campo contatore di valore minimo (riga 15). Come detto, deve saltare le pagine contenenti entità residenti o vietate (righe 12–14). Per evitare il problema di cui al punto 3 precedente, è sufficiente che tra due entità con contatore uguale si scelga sempre quella di livello minore (righe 16–17).

2.2.4 Routine delle statistiche

La routine di Figura 7 può essere chiamata ogni volta che si vogliono aggiornare le statistiche di utilizzo in base ai valori correnti dei bit A nei descrittori di tabelle e pagine virtuali. Poichè la funzione è piuttosto costosa (soprattutto per l'azione eseguita alla riga 27), abbiamo scelto di chiamarla solo quando si è verificato un page fault.

La funzione scorre tutti i descrittori di pagina fisica (righe 7–26) alla ricerca di quelli che contengono tabelle (righe 9–10: se il livello è minore di 1 la pagina fisica o è vuota o contiene una pagina virtuale). Una volta trovata, ne ottiene l'indirizzo fisico (riga 11) ed esamina tutti i suoi 512 descrittori (righe 12–25) alla ricerca di quelli che puntano a entità presenti (righe 13–15). Per ognuno di questi descrittori estrae il bit A (riga 16) e lo azzera (riga 17), ottiene un puntatore dal descrittore di pagina fisica che contiene l'entità puntata (righe 18–19), eventualmente saltando quelle residenti, di cui non importa aggiornare il contatore (righe 21–22). Infine, aggiorna il contatore in base al valore appena estratto del bit A (righe 23–25). Si noti che l'algoritmo usato per l'aggiornamento del contatore serve a implementare una approssimazione di LRU (Least Recently Used) per la scelta della vittima. L'idea è di usare il contatore come un registro a scorrimento, in cui ogni posizione rappresenta una chiamata della funzione `stat()`, dalla più recente nel bit più significativo alla meno recente nel bit meno significativo. Ogni bit contiene il bit A visto dalla corrispondente `stat()`. In questo modo il contatore mantiene la storia degli ultimi 32 bit A visti, dando maggior peso a quelli più recenti. La pagina che ha il contatore minimo è quella che non ha visto accessi da più tempo.

Alla riga 27, avendo azzerato tutti i bit A, dobbiamo invalidare il TLB, in modo che i successivi accessi possano riportare gli opportuni bit A ad 1.

2.3 Elenco delle funzioni di utliità

2.3.1 Pagine fisiche

`des_pf* descrittore_pf(addr indirizzo_pf)`
: dato un indirizzo fisico `indirizzo_pf` restituisce un puntatore al descrittore di pagina fisica corrispondente;

`addr indirizzo_pf(des_pf *ppf)`
: dato un puntatore ad un descrittore di pagina fisica `ppf` restituisce l'indirizzo fisico (del primo byte) della pagina fisica corrispondente;

`des_pf* alloca_pagina_fisica_libera()`
: restituisce un puntatore al descrittore di una pagina fisica libera, se ve ne sono, e zero altrimenti;

`void rilascia_pagina_fisica(des_pf *ppf)`
: rende di nuovo libera la pagina puntata da `ppf`.

`des_pf* alloca_pagina_fisica(natl proc, int liv, addr ind_virt)`
alloca una pagina fisica per potervi caricare l'entità identificata da `proc`, `liv` e `ind_virt`. Se non vi sono pagine libere può eseguire anche un rimpiazzamento. Restituisce un puntatore al descrittore di pagina fisica della pagina allocata, 0 non è stato possibile allocarla.

2.3.2 Descrittori di tabelle e pagine virtuali

`natq& get_entry(addr tab, natl index)`
restituisce un riferimento all'entrata `index` della tabella (di qualunque livello) di indirizzo `tab`;

`natq& get_des(natl proc, int liv, addr ind_virt)`
restituisce un riferimento al descrittore di livello `liv` da cui passa la traduzione dell'indirizzo `ind_virt` nello spazio di indirizzamento del processo `proc`;

`bool extr_P(natq descrittore)`
estrae il bit P da `descrittore`;

`bool extr_A(natq descrittore)`
estrae il bit A da `descrittore`;

`bool extr_D(natq descrittore)`
estrae il bit D da `descrittore`;

`addr extr_IND_FISICO(natq descrittore)`
estrae il campo indirizzo fisico da `descrittore` (ll campo è significativo se il bit P è 1);

`addr extr_IND_MASSA(natq descrittore)`
estrae il campo indirizzo di massa da `descrittore` (ll campo è significativo se il bit P è 0);

`void set_P(natq& descrittore, bool bitP)`
setta il valore del bit P in `descrittore` in base al valore di `bitP`;

`void set_A(natq& descrittore, bool bitP)`
setta il valore del bit A in `descrittore` in base al valore di `bitA`;

`void set_D(natq& descrittore, bool bitD)`
setta il valore del bit D in `descrittore` in base al valore di `bitD`;

`void set_IND_FISICO(natq& descrittore, addr ind_fisico)`
scrive `ind_fisico` nel campo indirizzo fisico di `descrittore`;

`void set_IND_MASSA(natq& descrittore, addr ind_massa)`
scrive `ind_massa` nel campo indirizzo di massa di `descrittore`;

2.3.3 Funzioni di supporto alla memoria virtuale

`void carica(des_pf *ppf)`
: carica la pagina fisica descritta da `ppf` in base alle informazioni contenute nel descrittore stesso (legge dallo swap dal blocco `ind_massa`);

`void scarica(des_pf *ppf)`
: copia il contenuto della pagina fisica descritta da `ppf` nel corrispondente blocco dello swap (scrive nel blocco `ind_massa`);

`void collega(des_pf *ppf)`
: rende presente l'entità contenuta in `ppf` inizializzando l'opportuno descrittore di tabella o pagina virtuale;

`void scollega(des_pf *ppf)`
: rende non più presente l'entità contenuta in `ppf`. Restituisce `true` e è necessario scaricare l'entità contenuta prima di sovrascriverla.