Implementazione della memoria virtuale

G. Lettieri

19 Maggio 2021

In questa ultima parte studiamo l'implementazione della paginazione nel nucleo didattico.

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. Tutti i processi utente 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 1. La Figura mostra lo spazio di indirizzamento virtuale di due processi, P_1 e P_2, insieme al possibile contenuto della memoria fisica. Le annotazioni che si trovano ai lati sinistro di P_1 valgono anche per P_2 e per qualunque altro processo, e lo stesso per le annotazioni che si trovano a destra di P_2. La memoria virtuale di ogni processo è suddivisa *a priori* nello stesso modo, come mostrato sulla destra della memoria virtuale di P_2: la parte che va dall'indirizzo `0000 0000 0000 0000` all'indirizzo `0000 7fff ffff ffff` ($2^{47} - 1$) è dedicata al sistema (bit U/S pari a 0 in tutte le traduzioni), mentre la parte che va da `ffff 8000 0000 0000` a `ffff ffff ffff ffff` è dedicata all'utente (bit U/S pari a 1 in tutte le traduzioni). Gli indirizzi virtuali della prima pagina (da 0 a `fff`) sono lasciati non mappati, per intercettare dereferenziazioni di `nullptr` indipendentemente dal livello di privilegio del processore.

Ogni parte (utente o sistema) dello spazio di indirizzamento di ogni processo è suddivisa in ulteriori sezioni. Sulla sinistra della memoria virtuale di P_1 abbiamo mostrato alcune costanti (definite in `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. Quindi, per

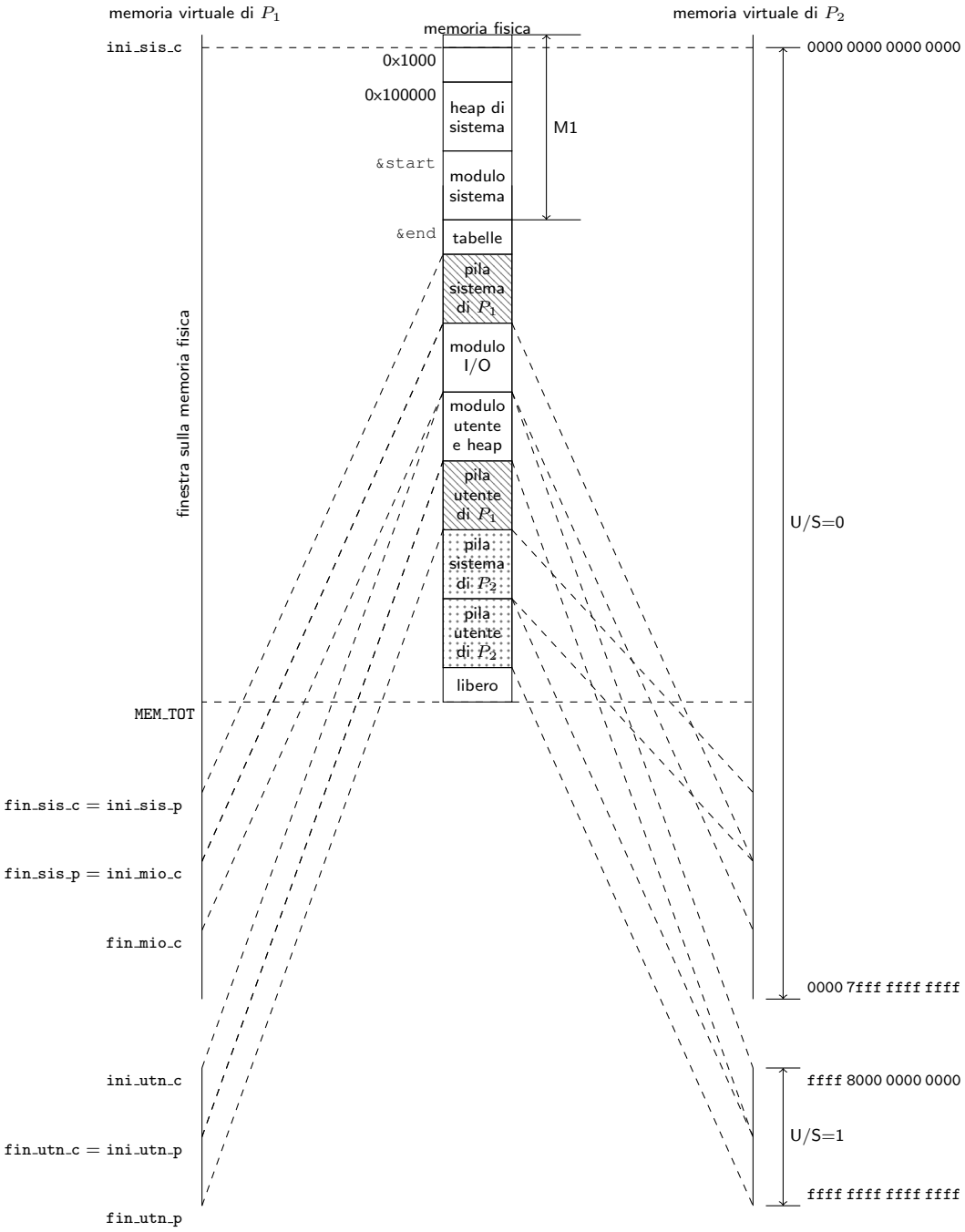

Figura 1: Esempio di memoria virtuale con due processi (figura non in scala).

esempio, `fin_sis_c` è l'indirizzo di fine della sezione sistema/condivisa. Le sezioni condivise contengono le stesse traduzioni in tutti i processi.

Le sezioni della memoria virtuale di ogni processo sono elencate di seguito.

sistema/condivisa: contiene la finestra sulla 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, la finestra arriva solo fino a `MEM_TOT`). La sezione contiene anche delle traduzioni identità (non mostrate in Figura) per quegli indirizzi che corrispondono a registri di periferiche (come per esempio l'APIC). Il resto della sezione è lasciato inutilizzato.

sistema/privata: 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: contiene il modulo I/O, vale a dire le sezioni **`.text`**, **`.data`** estratte dal file `build/io` e mappate al loro indirizzo di collegamento. Contiene inoltre lo *heap I/O*, vale a dire la zona di memoria su cui lavorano gli operatori **`new`** e **`delete`** quando usati dal modulo I/O.

utente/condivisa: contiene il modulo utente, vale a dire le sezioni **`.text`** e **`.data`** estratte dal file `build/utente` e caricate al loro indirizzo di collegamento. Analogamente al modulo I/O, contiene inoltre lo *heap utente*, che è usato dagli operatori **`new`** e **`delete`** quando sono invocati dal modulo utente.

utente/privata: contiene la pila utente del processo.

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 1 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 frame non contigui della memoria fisica.

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 `b8000`).
- 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`, le strutture `richiesta` (usate dalla primitiva `delay`), e in generale qualunque struttura dati che debbe essere allocata dinamicamente (tramite gli operatori **`new`** e **`delete`**).

- 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.

Le linee tratteggiate tra le memorie virtuali di P_1 e P_2 e la memoria fisica vogliono rappresentare la corrispondenza tra le sezioni della memoria virtuale di ciascun processo 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 utente/privata (tra gli indirizzi `ini_utn_p` e `fin_utn_p`) 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 la parte sistema/privata. Si noti che alcune parti della memoria fisica sono accessibili esclusivamente tramite la finestra: in particolare, tutta la parte M1 e i frame di M2 che o contengono tabelle o sono vuote (la parte indicata con "libero" in Figura 1). Alcune parti di M2 sono accessibili 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 tutti i frame che contengono le pagine della memoria virtuale di ogni processo.

2 Implementazione

In questa sezione esaminiamo le strutture dati e le funzioni che si trovano nel modulo sistema e sono relative all'implementazione della memoria virtuale.

La suddivisione dello spazio di indirizzamento dei processi nelle sue varie parti è stabilita da alcune costanti definite in `costanti.h` (si vedano i commenti in "suddivisione della memoria virtuale" all'interno del file). Per semplicità le varie parti occupano ciascuna un numero intero di regioni di livello massimo (quindi 512 GiB per i trie a 4 livelli), e dunque le possiamo distinguere in base alle entrate utilizzate nella tabella radice. Per esempio, la parte sistema/condivisa usa la prima entrata della radice (indice 0), mentre la parte sistema/privata usata la seconda (indice 1).

Anche le dimensioni delle varie parti *fisiche* della memoria dei processi sono decise *a priori* con delle costanti: la dimensione delle pila, sistema e utente, e la dimensione degli heap dei tre moduli (sistema, I/O e utente). Lo stesso vale per la RAM totale del sistema, che è decisa dalla costante `MEM_TOT`. Per aumentare o diminuire la RAM usata dal sistema (M1+M2) occorre modificare questa costante e ricompilare. Si noti che lo script `run` configura la macchina virtuale con la quantità di RAM stabilita da questa stessa costante.

```
1  struct des_frame {
2    union {
3      natw nvalide;
4      natl prossimo_libero;
5    };
6  };
```

Figura 2: Il descrittore di frame.

2.1 Gestione della memoria fisica

Il sistema deve sapere quali frame della memoria sono liberi o occupati, in modo da poterli allocare per le pagine e le tabelle dei processi. Dal momento che tutti i frame sono equivalenti, è sufficiente mantenere una lista dei frame liberi da cui si estrae e si inserisce sempre in testa.

In generale, il sistema ha bisogno di associare anche altre informazioni ad ogni frame. Per questo scopo il modulo sistema alloca un *descrittore di frame* per ogni frame della memoria fisica. Il descrittore ha il tipo mostrato in Figura 2 e contiene le seguenti informazioni:

- `prossimo_libero`: (significativo solo se il frame è libero) il numero del prossimo frame nella lista dei frame liberi;
- `nvalide`: (significativo solo se il frame contiene una tabella) quante entrate con P diverso da zero sono contenute nella tabella.

Si noti che, dal momento che le due informazioni non sono mai utili contemporaneamente, sono dichiarate all'interno di una **`union`**, che permette di risparmiare spazio.

I descrittori di frame sono raccolti in un array, `vdf[]`, indicizzato dal numero di frame. La funzione `init_frame()`, chiamata in fase di inizializzazione, si preoccupa di inizializzare questo array, inserendo tutti i frame di M2 nella lista dei frame liberi. Da questo punto in poi è possibile allocare e deallocare frame usando le funzioni:

- `paddr alloca_frame()`, che prova ad estrarre un frame dalla lista dei liberi e ne restituisce l'indirizzo (fisico) di base (0 se la lista è vuota);
- **`void`** `dealloca_frame(paddr p)`, che inserisce il frame di indirizzo fisico `p` nella lista dei liberi.

Le funzioni di utilià `inc_ref()`, `dec_ref()` e `get_ref()` servono a manipolare il contatore `nvalide` di una tabella di cui conosciamo l'indirizzo fisico.

2.2 Creazione delle parti condivise

Tutte le traduzioni relative alle parti condivise dei processi (sistema, I/O e utente) possono essere create una volta per tutte all'avvio del sistema. I processi

possono condividere tutto il sottoalbero che implementa queste traduzioni, a partire dal livello inferiore a quello della radice (il livello 3 nei processori con trie a 4 livelli). Per esempio, possiamo creare la traduzione della finestra sulla memoria una sola volta. Tutte le entrate di indice 0 delle tabelle radice di tutti i processi punteranno poi alla stessa tabella di livello inferiore, e dunque tutto il sottoalbero sarà condiviso tra tutti i processi. Questo ci permette di risparmiare molto spazio, e anche del tempo ogni volta che creiamo un nuovo processo.

Le traduzioni della parte sistema/condivisa sono create dal modulo sistema nella fase di inizializzazione, tramite la funzione `crea_finestra_FM()`, In particolare sono mappati in se stessi tutti gli indirizzi da `DIM_PAGINA` a `MEM_TOT`, in modo da creare la finestra sulla memoria fisica, lasciando non mappata la prima pagina in modo da poter intercettare le dereferenziazioni di `nullptr`. Le traduzioni della finestra sono implementate con pagine di livello 2 dove possibile, e sono impostate con bit R/W a 1 (scrittura permessa) e bit U/S a zero (accesso consentito solo da livello sistema). Questa traduzione permette anche di accedere alle sezioni **`.text`**, **`.data`** e **`.bss`** del modulo sistema, che sono collegate e caricate a partire dall'indirizzo `200000` (2 MiB, inizio del terzo megabyte di RAM). Si noti che la finestra comprende anche la memoria video in modalità testo, che si trova nella pagina di indirizzo base `b8000`. Nella traduzione degli indirizzi di questa pagina viene anche settato il bit PWT, in modo che le scritture raggiungano effettivamente la memoria video e non si fermino in cache per un tempo indefinito. Non è invece necessario disabilitare del tutto la cache, in quanto nel nostro caso la memoria video può essere modificata solo dal software e, dunque, possiamo sfruttare la cache per le operazioni di lettura. Sempre nella stessa funzione sono anche create le traduzioni identità che permettono di accedere all'APIC, i cui registri, come sappiamo, sono mappati nello spazio di memoria. In questo caso dobbiamo completamente disabilitare la cache, settando il bit PCD.

Le traduzioni delle parti IO/condivisa e utente/condivisa sono create nella funzione `crea_spazio_condiviso()`. Il boot loader di QEMU ha copiato i file `io` e `utente` in memoria (nel secondo megabyte di RAM). Gli indirizzi di partenza delle due copie sono scritti in una struttura dati che il boot loader passa alla routine di inizializzazione del modulo sistema[1]. La vera e propria creazione delle traduzioni per i due moduli si trova nella funzione `carica_modulo()`, invocata una volta per il modulo I/O e un'altra per il modulo utente. Questa funzione interpreta il file ELF copiato in memoria e ne consulta la tabella di programma, per sapere quali parti del file devono essere mappate nello spazio di indirizzamento. Per ciascuna di esse alloca opportunamente dei frame dalla parte M2, vi copia il corrispondente contenuto del file (o provvede ad azzerare i byte del frame, nel caso del **`.bss`**) e crea le traduzioni. Inoltre, alloca ulteriori frame destinati a contenere lo heap del modulo e li mappa in coda all'ultimo indirizzo virtuale menzionato nel file ELF (normalmente, dunque, subito dopo

[1] In realtà c'è un passaggio intermedio tramite il boot loader della `libce`, in quanto il boot loader di QEMU porta il processore soltanto nella modalità a 32 bit. Il boot loader di `libce` passa alla modalità a 64 bit e cede il controllo al modulo sistema, passandogli gli stessi parametri ricevuti dal boot loader di QEMU.

le sezioni **.data** e **.bss**). A questo punto lo spazio occupato dai due file copiati dal boot loader di QEMU può essere riutilizzato ed entra a far parte dello heap del modulo sistema, che si estende per tutto il secondo megabyte.

Tutte queste traduzioni vengono create nello stesso albero di traduzione. Una volta create, la radice dell'albero viene caricata nel registro **cr3**. Da quel punto in poi il sistema smette di usare le traduzioni che aveva preparato il boot loader e usa soltanto le proprie.

2.3 Creazione dei processi

La creazione dei processi si trova nella funzione `crea_processo()`. Oltre ad allocare e inizializzare un nuovo `des_proc` nei modi che già conosciamo, la funzione si deve preoccupare di creare tutta la memoria virtuale del processo. Per far questo alloca una nuova tabella radice e vi copia tutte le entrate relative alle parti condivise, prendendole dalla tabella radice che in questo momento è puntata da **cr3**. Nel caso dei primi processi, creati in fase di inizializzazione, questa è la tabella radice di cui abbiamo parlato nella sezione precedente. Tutti gli altri processi sono creati da altri processi, e in questo caso il registro **cr3** punta alla tabella radice del processo genitore. In ogni caso, l'effetto è che le entrate relative alle parti condivise punteranno tutte agli stessi sottoalberi, come avevamo anticipato.

La funzione `crea_processo()` deve poi creare le parti private. Queste comprendono la pila sistema (parte sistema/privata) e, per i processi di livello utente, anche la pila utente (parte utente/privata). Queste sono create allocando un numero prestabilito di frame (in base alle costanti definite in `costanti.h`) e poi mappandoli contiguamente partendo dagli indirizzi inferiori delle rispettive parti dello spazio di indirizzamento.

La funzione, ricordiamo, deve anche *inizializzare* la pila sistema, inserendovi le cinque parole quadruple che verranno poi lette dalla **iretq** la prima volta che il processo andrà in esecuzione. Qui bisogna fare particolare attenzione: la funzione non può scrivere nella pila sistema del processo appena creato usando gli indirizzi della parte sistema/privata, in quanto al momento è ancora attivo lo spazio di indirizzamento del processo *genitore* e quegli indirizzi verrebbero tradotti dalla MMU nei frame della pila sistema di questo processo, e non del processo creato. Per poter scrivere nella pila sistema del processo creato, la funzione sfrutta la finestra sulla memoria fisica, accedendo direttamente ai frame della pila tramite il loro indirizzo fisico.

2.4 Le funzioni di supporto

Il modulo sistema definisce alcune funzioni di supporto per svolgere i suoi compiti. Le illustriamo in quanto possono essere utili anche nello svolgimento degli esercizi.

Come abbiamo visto, il sistema associa ad ogni tabella un contatore delle entrate valide. Per gestire correttamente questo contatore, sono utili le seguenti funzioni:

- `paddr alloca_tab()`
  alloca un frame destinato a contenere una tabella; rispetto ad una semplice `alloca_frame()` provvede anche ad azzerare sia il frame, sia il contatore `nvalide` nel suo descrittore;

- **`void`** `rilascia_tab()`
  rilascia un frame che conteneva una tabella; rispetto ad una semplice `rilascia_frame()`, controlla anche che il contatore `nvalide` non sia diverso da zero, in modo da intercettare errori di programmazione in cui si tenta di rilasciare tabelle che ancora contengono entrate valide;

- **`void`** `set_entry(paddr tab, natl j, tab_entry se)`
  fa il contrario della `get_entry()` di `libce`, settando l'entrata `j`-esima della tabella di indirizzo fisico `tab` con il nuovo valore `se`; si preoccupa di aggiornare opportunamente il contatore `nvalide` se il valore del bit P cambia per effetto dell'assegnamento;

- **`void`** `copy_des(paddr src, paddr dst, natl i, natl n)`
  copia `n` entrate a partire dalla `i`-esima della tabella di indirizzo fisico `src` nelle corrispondenti entrate della tabelle di indirizzo fisico `dst`; si preoccupa di aggiornare opportunamente il contatore `nvalide` della tabella `dst`; questa funzione è utile, per esempio, quando si crea un nuovo processo e si devono copiare le entrate delle parti condivise dalla tabella radice del processo genitore;

- **`void`** `set_des(paddr dst, natl i, natl n, tab_entry e)`
  setta `n` entrate a partire dalla `i`-esima della tabella di indirizzo fisico `dst`, impostandole tutte uguali a `e` e aggiornando opportunamente il contatore `nvalide`; questa funzione è utile soprattutto quando `e` è zero perché si sta distruggendo un albero di traduzione (per esempio, quando un processo termina);

Le funzioni di supporto più utili sono le funzioni `map()` e `unmap()`, che il sistema usa per creare (o distruggere) le traduzioni di tutte le parti sistema, IO e utente, sia condivise che private.

La funzione `map()` riceve l'indirizzo fisico `tab` della tabella radice di un trie, gli estremi di un intervallo [**`begin`**, **`end`**) di indirizzi virtuali (allineati alla pagina) e un parametro template `getpaddr` che si deve comportare come una funzione da `vaddr` a `paddr`. La funzione `map()` creerà, nell'albero di radice `tab`, le traduzioni $v \mapsto$ **`getpaddr`**(v) per tutti gli indirizzi di pagina v nell'intervallo [**`begin`**, **`end`**). La funzione riceve anche un parametro `flags` con cui si può specificare il valore desiderato per i flag U/S, R/W, PWT e PCD. Per esempio, per creare delle traduzioni identità nell'intervallo [`1000`, `80 0000`) (esadecimale) in modo che siano accessibili in scrittura da livello sistema, si può scrivere:

```
paddr identity_map(vaddr v)
{
  return v;
```

```
}

void some_function()
{
   ...
   map(tab, 0x1000, 0x800000, BIT_RW, identity_map);
   ...
}
```

La funzione creerà il mapping

$$\texttt{1000} \mapsto \texttt{identity\_map}(\texttt{1000}) = \texttt{1000},$$

poi il mapping

$$\texttt{2000} \mapsto \texttt{identity\_map}(\texttt{2000}) = \texttt{2000}$$

e così via fino a

$$\texttt{7f 0000} \mapsto \texttt{identity\_map}(\texttt{7f 0000}) = \texttt{7f 0000}.$$

Se invece vogliamo mappare gli stessi indirizzi su dei nuovi frame di M2, basta sostituire la funzione passata come `getpaddr()`:

```
paddr my_alloc_frame(vaddr v)
{
  return alloca_frame();
}

void some_function()
{
   ...
   map(tab, 0x1000, 0x800000, BIT_RW, my_alloc_frame);
   ...
}
```

In questo caso `map()` allocherà un nuovo frame per ogni indirizzo di pagina nell'intervallo, quindi mapperà la pagina su quel frame.

Nel modulo sistema si preferisce usare espressioni *lambda* al posto dei puntatori a funzione, per comodità. Un'altra possibilità è di usare oggetti istanza di classi/strutture che ridefiniscono **`operator`**`()`. Questo è utile quando, per creare correttamente le traduzioni, non ci basta conoscere l'indirizzo virtuale e abbiamo bisogno di portarci dietro altre informazioni. Un esempio, nel modulo sistema, è dato dalla funzione `carica_modulo()`, che deve creare un mapping per ogni segmento di un file ELF. Qui vediamo un esempio più semplice: supponiamo di dover creare un mapping tra lo stesso intervallo di prima e degli indirizzi fisici arbitrari, scritti in un array `paddr a[]`. Possiamo operare così:

```
class my_addrs {
  paddr *pa;
```

```
  int i;
public:
  my_addrs(paddr *pa_): pa(pa_), i(0) {}

  paddr operator()(vaddr v) {
    return pa[i++];
  }
};

void some_function()
{
   paddr a[] = { ... };
   my_addrs m(a);

   map(tab, 0x1000, 0x800000, BIT_RW, m);
}
```

Opzionalmente, `map()` può creare le traduzioni usando pagine di livello maggiore di 1. È sufficiente passare il livello desiderato come ulteriore parametro. Per esempio, la funzione `crea_finestra_FM()` usa pagine di livello 2 e passa questo valore come ultimo argomento di `map()`.

La funzione `unmap()` esegue l'operazione inversa di `map()`: distrugge tutti le traduzioni in un dato intervallo di indirizzi virtuali. La funzione si preoccupa di deallocare (tramite `rilascia_tab()` anche tutte le tabelle che diventano vuote dopo aver eliminato le traduzioni dell'intervallo. La funzione riceve un parametro template `putpaddr` che l'utente può usare per decidere cosa fare di ogni indirizzo fisico che prima era mappato da qualche indirizzo virtuale nell'intervallo. Per esempio, per distruggere il mapping creato tramite `identity_map()` non è necessario fare niente e `putpaddr` può essere una NOP:

```
void do_nothing(paddr p, int lvl)
{
}

void some_function()
{
   ...
   unmap(tab, 0x1000, 0x800000, do_nothing);
   ...
}
```

Invece, per disfare i mapping creati tramite `my_alloc_frame()` è necessario chiamare `rilascia_frame()` su tutti i frame non più mappati:

```
void my_rel_frame(paddr p, int lvl)
{
  rilascia_frame(p);
```

```
}

void some_function()
{
   ...
   unmap(tab, 0x1000, 0x800000, my_rel_frame);
   ...
}
```

Si noti che la funzione passata a `unmap()` riceve anche un parametro **`int`** `lvl`, che è il livello della pagina che era precedentemente mappata sull'indirizzo fisico `p`, nel caso questa informazione sia necessaria per capire cosa fare di `p`.