Funzioni di supporto per la paginazione

G. Lettieri

11 Aprile 2024

La libreria `libce` definisce i tipi numerici `paddr vaddr`, che rappresentano rispettivamente indirizzi fisici e virtuali. I due tipi hanno solo lo scopo di ricordare al programmatore quando si suppone che un indirizzo debba essere virtuale o fisico, ma sono entrambi equivalenti ad un intero senza segno su 64 bit.

La libreria contiene anche alcune strutture dati e funzioni di uso generale, a cui si può accedere includendo il file `vm.h`. Nel seguito descriviamo il contenuto di questo file.

1 Calcoli con gli indirizzi

Queste funzioni eseguono semplici calcoli sugli indirizzi virtuali.

1.1 La funzione `norm()`

La funzione `vaddr norm(vaddr a)` serve a *normalizzare* un indirizzo virtuale, cioè a rendere i 16 bit più significativi tutti uguali al bit numero 47. Può essere anche usata per controllare se un dato indirizzo (per esempio ricevuto da una sorgente non fidata, come il livello utente) è normalizzato o meno:

```
if (norm(v) == v) {
  // v è normalizzato
} else {
  // v non è normalizzato: errore
}
```

1.2 La funzione `dim_region()`

La funzione `natq dim_region(int liv)` restituisce la dimensione in byte di una regione di livello `liv`. Si ricordi che abbiamo chiamato "regione di livello i" l'intervallo di indirizzi coperti da una singola entrata di un tabella di livello $i+1$. Quindi, per esempio, una regione di livello 0 è grande 4096 byte (pagina di livello 1), mentre una regione di livello 1 è grande 2 MiB (pagina di livello 2). La funzione può essere anche utilizzata per ottenere le maschere che permettono

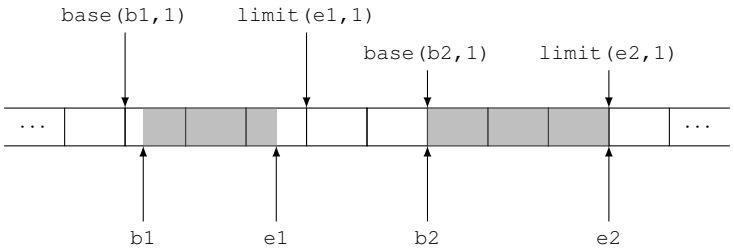

Figura 1: Esempio di calcolo di `base()` e `limit()` per due intervalli di indirizzi, [`b1`, `e1`) e [`b2`, `e2`).

di estrarre da un indirizzo virtuale il numero di pagina e l'offset. Per esempio, se `v` è un indirizzo virtuale e vogliamo sapere in che pagina di livello 2 cade, e a quale offset:

```
natq mask = dim_region(1) - 1;
natq base = v & ~mask;  // indirizzo base della pagina
natq offset = v & mask; // offset nella pagina
```

Questo perché `dim_region(liv)` ha la forma di 2^n, che in binario è un 1 seguito da n zeri, e dunque `dim_region(liv)- 1` produce una maschera di n bit ad 1.

1.3 Le funzioni `base()` e `limit()`

La base della regione di livello `liv` in cui cade un indirizzo `v` si può ottenere anche dalla funzione `vaddr base(vaddr v, int liv)`. Invece, la funzione `vaddr limit(vaddr e, int liv)` serve a calcolare la base della prima regione di livello `liv` che si trova a destra di un intervallo [`b`, `e`) senza toccarlo (si veda la Figura 1).

2 Manipolare singole entrate

Queste definizioni e funzioni aiutano a leggere/modificare singole entrate delle tabelle.

2.1 Il tipo `tab_entry`

Il tipo `tab_entry` rappresenta una entrata di una tabella, di qualunque livello. Il file contiene la definizione di un po' di costanti, una per ogni bit del byte di accesso delle entrate, che possono essere usati come maschere per estrarre, settare o resettare i vari bit. Per esempio, se `e` è un riferimento ad una entrata di una tabella, possiamo

- esaminare il bit P con "`if (e & BIT_P){ /*qualcosa */}`";

- settare il bit U/S con "`e |= BIT_US`";
- resettare il bit R/W con "`e &= ~BIT_RW`"

e così via.

2.2 Le funzioni `extr/set_IND_FISICO()`

La funzione `paddr extr_IND_FISICO(tab_entry e)` può essere usate per estrarre l'indirizzo fisico contenuto nell'entrata `e`. Tale indirizzo rappresenta l'indirizzo (fisico) della tabella di livello inferiore o, nel caso di tabelle foglia, la base del frame in cui è mappata una pagina virtuale.

La funzione `set_IND_FISICO(tab_entry& e, paddr p)` serve invece a settare il campo "numero di frame" dell'entrata `e` con il numero di frame dell'indirizzo fisico `p`, senza modificare il byte di accesso.

3 Lavorare su singole tabelle

Queste funzioni aiutano a leggere/modificare una tabella del TRIE alla volta.

3.1 Le funzioni `i_tab()` e `get/set_entry()`

La funzione **`int`** `i_tab(vaddr v, int liv)` estrae dall'indirizzo virtuale `v` l'indice che la MMU usa per consultare le tabelle di livello `liv`. Per esempio, se `liv` è 2, la funzione restituisce l'indice contenuto nei bit 29–21 di `v` (si veda avanti, sezione 5.3.1).

La funzione `tab_entry& get_entry(paddr t, int i)`, dato l'indirizzo fisico `t` di una tabella e un indice `i`, restituisce un riferimento all'entrata `i`-esima della tabella stessa (il riferimento potrà essere effettivamente usato solo se è accessibile la finestra sulla memoria fisica).

Per esempio, se vogliamo settare il bit R/W nell'entrata relativa ad un indirizzo virtuale `v` in una tabella di livello 2 di indirizzo fisico `tab`, possiamo scrivere

```
tab_entry& e = get_entry(tab, i_tab(v, 2));
e |= BIT_RW;
```

Come ulteriore esempio, supponiamo che `tab4`, `tab3`, `tab2` e `tab1` siano gli indirizzi fisici di quattro tabelle inizialmente vuote e che vogliamo costruire il percorso di traduzione che mappi l'indirizzo virtuale `v` nell'indirizzo fisico `p`, in modo che sia consentito l'accesso in scrittura ma non quello da livello utente. Possiamo scrivere:

```
// aggancio tab4->tab3
tab_entry& e4 = get_entry(tab4, i_tab(v, 4));
set_IND_FISICO(e4, tab3);
e4 |= BIT_P | BIT_RW;
```

```
// aggancio tab3->tab2
tab_entry& e3 = get_entry(tab3, i_tab(v, 3));
set_IND_FISICO(e3, tab2);
e3 |= BIT_P | BIT_RW;
// aggancio tab2->tab1
tab_entry& e2 = get_entry(tab2, i_tab(v, 2));
set_IND_FISICO(e2, tab1);
e2 |= BIT_P | BIT_RW;
// installo la traduzione v -> p
tab_entry& e1 = get_entry(tab1, i_tab(v, 1));
set_IND_FISICO(e1, p);
e1 |= BIT_P | BIT_RW;
```

Esiste anche la funzione

```
void set_entry(paddr tab, natl j, tab_entry se)
```

che può essere usata per sovrascrivere completamente l'entrata `j` della tabella `tab` con il nuovo valore `se`. Rispetto a modificare una entrata tramite il riferimento ottenuto da `get_entry(tab, j)`, ha il vantaggio di gestire automaticamente l'eventuale contatore di entrate valide della tabella `tab` (si veda più avanti, sezione 5.3.1).

3.2 Le funzioni `copy_des()` e `set_des()`

Queste sono due funzioni di utilità che permettono di copiare una o più entrate da una tabella ad un'altra (`copy_des()`) o di sovrascrivere una o più entrate di una tabella con uno stesso valore (`set_des()`). Rispetto a scrivere un semplice ciclo, hanno il vantaggio di gestire automaticamente l'eventuale contatore di entrate valide nella tabella di destinazione (si veda più avanti, sezione 5.3.1).

4 Accesso all'hardware

Queste funzioni permettono di accedere ai registri della MMU o di interagire con il TLB.

4.1 Le funzioni `readCR3()`/`loadCR3()` e `readCR2()`

Queste funzioni sono scritte in assembler e permettono di leggere i registri **`cr3`** e **`cr2`**, e scrivere nel registro **`cr3`**. In particolare, `loadCR3()` va usata per attivare un nuovo albero di traduzione, passandole l'indirizzo fisico della tabella radice. Si ricordi che ha l'effetto collaterale di invalidare tutto il TLB.

Il registro **`cr2`** contiene l'ultimo indirizzo virtuale che ha causato una eccezione di page fault e non è scrivibile da software.

4.2 Le funzioni per invalidare il TLB

La funzione `invalida_entrata_TLB(vaddr v)` serve a invalidare la traduzione associata all'indirizzo virtuale `v`, nel caso il TLB ne stesse conservando una copia. È scritta in assembler e usa l'istruzione `invlpg`. La funzione va usata ogni qual volta si cambia qualcosa nel percorso di traduzione di `v`. Non solo, dunque, se si cambia la traduzione, ma anche se si cambia qualche bit di uno qualunque dei byte di accesso che si incontrano nel percorso di traduzione di `v`, perché il TLB memorizza anche quelli, o comunque assume che siano sempre nello stato in cui li aveva visti la MMU al momento del caricamento della traduzione.

La funzione `invalida_TLB()` serve ad invalidare tutto il contenuto del TLB. È equivalente a `loadCR3(read(CR3))`. Ha senso chiamarla se sono stati fatti molti cambiamenti (per esempio, azzeramento di tutti i bit A dopo averli esaminati) e dunque diventa conveniente rispetto a chiamare tante volte `invalida_entrata_TLB()`.

5 Lavorare con interi TRIE

Le funzioni più utili messe a disposizione dalla libreria permettono di lavorare con interi TRIE, usando internamente le funzioni definite qui sopra.

5.1 L'iteratore `tab_iter`

Si tratta di un iteratore che permette di visitare tutte le entrate dell'albero di traduzione coinvolte, a tutti i livelli, nella traduzione di tutti gli indirizzi di un dato intervallo. La visita è del tipo *depth first* e può essere eseguita sia in ordine anticipato che posticipato. Tutte le entrate sono visitate una sola volta e le entrate foglia (che contengono le traduzioni) sono visitate rispettando l'ordine crescente degli indirizzi virtuali.

L'iteratore va costruito specificando l'indirizzo fisico della tabella radice dell'albero, la base dell'intervallo e la sua lunghezza (1 per default). Ad ogni istante, a meno ché la visita non sia terminata, l'iteratore si trova su una qualche entrata di una qualche tabella dell'albero. Appena costruito si troverà sull'entrata della tabella radice relativa all'indirizzo base dell'intervallo da visitare. L'iteratore può essere spostato sulla prossima entrata (secondo l'ordine depth first) con il metodo `next()`. L'operatore di conversione a **`bool`** restituisce **`false`** quando la visita è terminata. Le funzioni membro `get_e()`, `get_tab()` e `get_l()` e permettono di ottenere, rispettivamente, un riferimento all'entrata su cui si trova l'iteratore, l'indirizzo fisico della tabella che contiene questa entrate e il livello (4, 3, 2 o 1) di questa tabella. La funzione `get_v()`, invece, restituisce il più piccolo indirizzo virtuale la cui traduzione passa da questa entrata.

Consideriamo prima il caso particolare in cui l'intervallo consiste di un unico indirizzo e rifacciamoci all'esempio alla fine della Sezione 3.1. Possiamo stampare tutto il percorso di traduzione di `v` nel seguente modo:

```
for (tab_iter it(tab4, v); it; it.next()} {
  printf("tab %x, liv %d, entry %x\n",
    it.get_tab(),
    it.get_l(),
    it.get_e());
}
```

Il ciclo **for** costruisce un iteratore per l'albero di radice `tab4` e per l'intervallo di indirizzi virtuali $[\mathtt{v}, \mathtt{v}+1)$ (non avendo passato la lunghezza dell'intervallo come terzo argomento viene assunto il default di 1). Nella prima iterazione `it` si trova sull'entrata di `tab4` che abbiamo chiamato `e4` nella sezione 3.1. La `printf()` stamperà dunque l'indirizzo `tab4`, il livello 4 e il contenuto di `e4`, vale a dire l'indirizzo `tab3` e il byte di accesso. Nella seconda iterazione `it` si sposterà su `e3` e la `printf()` stamperà l'indirizzo `tab3`, il livello 3 e il contenuto di `e3`, cioè l'indirizzo `tab2` e il byte di accesso. Lo stesso accadrà per il livello 2 e il livello 1. A quel punto la visita è terminata e l'espressione "`it`" (che invoca l'operatore di conversione a **bool**) restituirà **false**, terminando il ciclo.

Supponiamo di modificare il codice qui sopra cambiando il ciclo for in

```
for (tab_iter it(tab4, v, 2*DIM_PAGINA); it; it.next()) {
```

Ora vogliamo visitare le traduzioni delle due pagine `v` e `v+DIM_PAGINA` (la pagina successiva a `v`, supponendo che `v` non sia l'ultima pagina dello spazio di indirizzamento e non sia adiacente al "buco" nello spazio). Supponendo che l'albero sia sempre quello creato in Sezione 3.1, l'iteratore seguirebbe lo stesso percorso di prima e, dopo aver visitato `e1`, si sposterebbe sull'entrata di `tab1` successiva, oppure, se `e1` fosse l'ultima entrata di `tab1` (quella di indice 511), sull'entrata di `tab2` successiva a `e2` (o ancora più su, se anche `e2` fosse l'ultima entrata di `tab2`). La `printf()` mostrerebbe la tabella e il livello su cui l'iteratore si è fermato e il contenuto dell'entrata corrente, che nel nostro caso sarebbe tutto nullo. Al prossimo passo la visita sarebbe terminata, perché la pagina `v+DIM_PAGINA` non ha una traduzione e non ci sono altre pagine nell'intervallo. Se invece anche `v+DIM_PAGINA` avesse una traduzione, l'iteratore scenderebbe lungo il suo percorso, fermandosi ad ogni livello fino al livello 1, e solo allora terminerebbe la visita.

La visita in ordine anticipato è utile quando vogliamo *costruire* l'albero di traduzione per un certo intervallo di indirizzi. Ogni volta che l'iteratore si ferma possiamo esaminare il bit P dell'entrata corrente e, se vale 0 e non siamo ancora arrivati al livello 1, allocare e agganciare una nuova tabella di livello inferiore. Per far questo sfruttiamo il fatto che `get_e()` ci restituisce un *riferimento* all'entrata su cui si trova l'iteratore, permettendoci dunque di modificarla. Al prossimo passo l'iteratore si sposterà sulle entrate rilevanti di questa nuova tabella e noi potremo continuare ad allocare ed agganciare le tabelle di livello inferiore oppure, arrivati al livello 1, installare le traduzioni. Qui possiamo usare la funzione membro `get_v()` per chiedere all'iteratore qual è l'indirizzo

virtuale il cui percorso di traduzione porta all'entrata su cui ci troviamo, in modo da poter scegliere correttamente la traduzione da associarvi. Questo è il meccanismo usato dalla funzione `map()` del modulo sistema.

L'iteratore può essere usato anche per eseguire una visita in ordine posticipato, nel seguente modo

```
tab_iter it(tab4, v);
for (it.post(); it; it.next_post()} {
  printf("tab %x, liv %d, entry %x\n",
    it.get_tab(),
    it.get_l(),
    it.get_e());
}
```

In questo caso il codice mosterà le entrate del percorso partendo dal livello 1 e salendo fino al 4. La visita in ordine posticipato è utile quando vogliamo *distruggere* un albero di traduzione, in quanto ci permette di eliminare i livelli inferiori dell'albero prima di esaminare i livelli superiori. Questo è il meccanismo usato dalla funzione `unmap()` del modulo sistema.

Quando vogliamo esaminare il percorso di traduzione di un singolo indirizzo conviene usare il seguente codice

```
tab_iter it(tab4, v);
while (it.down())
  ;
```

La funzione membro `down()` prova soltanto a scendere nell'albero, seguendo il percorso di traduzione di `v`, fermandosi alla prima foglia (sia perché ha trovato un bit P a zero, sia perché è arrivata alla traduzione). All'uscita dal **`while`** l'iteratore si trova ancora sull'entrata foglia, che possiamo così esaminare e/o modificare. Questo non è sempre vero con gli altri tipi di visita.

5.2 La funzione `trasforma()`

La funzione `paddr trasforma(paddr root, vaddr v)` permette di tradurre l'indirizzo virtuale `v` nel corrispondente indirizzo fisico in base al TRIE con radice `root`. Internamente la funzione usa un `tab_iter` per visitare il TRIE in software nello stesso modo in cui lo farebbe la MMU in hardware. Se `v` non ha traduzione, restituisce 0.

Per tradurre usando il TRIE corrente è sufficiente usare `readCR3()`:

```
paddr p = trasforma(readCR3(), v);
```

5.3 Le funzioni `map` e `unmap`

In molti casi il modo più semplice di manipolare i TRIE per creare o eliminare traduzioni è di usare queste funzioni.

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

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

poi il mapping

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

e così via fino a

$$7\texttt{f}\,\texttt{f}000 \mapsto \texttt{identity\_map}(7\texttt{f}\,\texttt{f}000) = 7\texttt{f}\,\texttt{f}000.$$

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.

Al posto dei puntatori a funzione si possono usare espressioni *lambda*, che sono spesso più comode. 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(vaddr v, 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(vaddr v, paddr p, int lvl)
{
  rilascia_frame(p);
}

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

Si noti che la funzione `putpaddr` riceve anche un parametro `vaddr v` e un parametro **`int`** `lvl`, che contengno l'indirizzo virtuale e il livello della pagina che era precedentemente mappata sull'indirizzo fisico `p`, nel caso queste informazioni siano necessarie per capire cosa fare di `p`.

5.3.1 Funzioni usate da **`map`** e **`unmap`**

Le funzioni `map` e `unmap` usano alcune funzioni per allocare e deallocare le tabelle. La `libce` fornisce una versione semplificata di queste funzioni, in cui le tabelle vengono allocate sullo heap e non vengono mai deallocate. Il modulo sistema, invece, fornisce una versione più sofisticata che mantiene per ogni tabella un contatore delle entrate valide (cioè le entrate con il bit di presenza P settato) e permette di deallocare le tabelle quando questo contatore vale zero.