1 Lezione del 04-04-25

1.1 Funzioni di supporto alla paginazione

Riprendiamo la trattazione della memoria virtuale paginata, discutendo quali funzioni il kernel studiato mette a supporto della sua operazione.

1.1.1 Funzioni sugli indirizzi

Nell'architettura x86_64 un indirizzo, virtuale o fisico, sta su 64 bit, e quindi definiamo i tipi `vaddr` e `paddr`:

```
typedef natq /* (uint64_t) */ vaddr; // indirizzo virtuale
typedef natq /*    idem    */ paddr; // indirizzo fisico
```

Dotiamoci quindi di alcune funzioni per la gestione di questi indirizzi. Di base, vorremmo un modo per verificare la normalizzazione di un indirizzo (quindi il fatto che i bit dal 48 al 63 siano uguali al bit 47, che possiamo fare con la `norm(vaddr)`):

```
if(norm(v) != v) {
  // errore: indirizzo non normalizzato
}
```

Definiamo poi due funzioni per trovare l'indirizzo della prima pagina contenuta e della prima pagina immediatamente dopo una regione di memoria $[x, y]$. Avevamo trattato questo problema nella sezione 2.1.3, e usando quanto avevamo detto possiamo definire le funzioni `base(vaddr)` e `limit(vaddr)`.

```
vaddr base(vaddr v, int liv)
{
  natq mask = dim_region(liv) - 1;
  return v & ~mask;
}

vaddr limit(vaddr v, int liv)
{
  natq dr = dim_region(liv);
  natq mask = dr - 1;
  return (v + dr - 1) & ~mask;
}
```

dove notiamo che l'indice di pagina resta comunque nei 36 (o meno nel caso di huge page) bit dedicati al numero di pagina (nel caso più semplice, azzeriamo i 12 bit di offset di pagina).

La funzione helper `dim_region(int)` calcola la dimensione di una pagina ad un certo livello di paginazione (sui i livelli $4, 3, 2, 1$, indicati come $3, 2, 1, 0$), ed è definita come:

```
natq dim_region(int liv)
{
  natq v = 1ULL << (liv * 9 + 12);
  return v;
}
```

1.1.2 Funzioni sulle tabelle

Un singolo descrittore di tabella viene rappresentato dal tipo `tab_entry`, che entra (come abbiamo visto nella sezione 14.0.3) in 64 bit, per cui possiamo dire:

```
typedef natq /* (uint64_t) */ tab_entry;
```

L'interazione vera e propria con il descrittore avverrà attraverso maschere e funzioni che applicano maschere. Ad esempio, se per modificare il bit di presenza di una pagina si può dire:

```
// e e' un riferimento ad una tabella
e |= BIT_P; // BIT_P maschera il bit di presenza
```

mentre se si vuole estrarre o modificare l'indirizzo fisico si possono sfruttare rispettivamente le funzioni `extr_IND_FISICO(tab_entry)` e `set_IND_FISICO(tab_entry)`:

```
// e e' un riferimento ad una tabella
paddr p;

// prendi l'indirizzo fisico di e
p = extr_IND_FISICO(e);

// per esempio, rimetticelo
set_IND_FISICO(e, p);
```

Esiste poi le funzione `i_tab(vaddr v, int liv)` per l'accesso alla regione da 9 bit che indirizza le tabelle di livello `liv` di un indirizzo `v`. Questa potrà essere usata con le funzioni `get_entry(paddr, int)` e `set_entry(paddr, int, tab_entry)`, che si occupano rispettivamente di ottenere un entrata di una tabella e modificarne una, sostituendo l'intero descrittore.

Ad esempio, potremo dire:

```
// questo e' un indirizzo, diciamo che la tabella tab e' di livello 2
vaddr v;

// e e' un descrittore di tabella di livello 1
tab_entry& e = get_entry(tab, i_tab(v, 2));

// creiamo un descrittore per sostituire e
tab_entry se;
// ... imposta se

// modifica la stessa entrata
set_entry(tab, i_tab(v, 2), se);
```

Copie o sovrascritture in massa si possono fare poi con le funzioni `copy_des()` e `set_des()`, che per adesso non vediamo nel dettaglio.

1.1.3 Funzioni sulla MMU

Notiamo l'esistenza della `loadCR3()` per l'attivazione di un nuovo albero di traduzione, che carica un indirizzo fisico nel registro `CR3`, e `readCR3()`, che permette successivamente di rileggerlo.

Esiste anche la `readCR2()`, che permette la lettura di `CR2`, dove si trova l'ultimo indirizzo la cui traduzione ha causato un pagefault.

Esistono poi le funzioni di interazione con il TLB: `invalida_entrata_TLB(vaddr v)` permette di invalidare l'indirizzo virtuale `v` sfruttando l'istruzione assembler `INVLPG`, mentre `invalida_TLB()` invalida l'intero albero di traduzione (cosa che ricordiamo faceva già la `loadCR3()`).

1.1.4 Iteratori di tabella

La gestione ad alto livello dei trie (*alberi di traduzione*) viene effettuata sfruttando l'iteratore `tab_iter`.

Questo serve per effettuare visite in diversi ordini (che vederemo fra poco), a livello di *pagina*: l'offset di pagina andrà comunque conservato a parte, in quanto l'iteratore ci porterà, al massimo, solo fino all'indirizzo fisico della pagina giusta. Sui `tab_iter` sono definite alcune funzioni membro: le `get_e()`, `get_tab()` e `get_l()` permettono di ottenere, rispettivamente, un riferimento all'entrata su cui si trova l'iteratore, l'indirizzo fisico della tabella che contiene questa entrata 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.

Il `tab_iter` viene inizializzato attraverso un indirizzo virtuale, o una coppia di questi in modo da esplorare un intera regione di indirizzi virtuali. Una volta definito un'oggetto `tab_iter`, si possono sfruttare le funzioni `up`, `down` e `right` per spostarsi rispettivamente nella tabella di livello superiore, inferiore, e a destra fra le tabelle di livello corrente. La funzione `done()`, che si ottiene anche dall'operatore di conversione a `bool`, restituisce falso quando la visita è terminata (siamo arrivati alla pagina o non possiamo proseguire).

Attraverso queste funzioni, e alle `next()` (che avanza l'iteratore in avanti in visita anticipata, cercando di raggiungere il prossimo indirizzo) e `next_post()` (che avanza l'iteratore in avanti in visita posticipata) si possono realizzare quindi diversi tipi di visita ad un trie, fra cui:

- **Visita anticipata di un singolo indirizzo:** si percorre l'intero percorso di traduzione di un indirizzo `v` come segue:

```
for(tab_iter it(tab, v); it; it.next()) {
  // it e' l'elemento corrente
}
```

  Una soluzione alternativa si ha sfruttando direttamente `down()`:

```
tab_iter it(tab, v);
while (it.down()) {
  // it e' l'elemento corrente
}
```

- **Visita anticipata di una regione di indirizzi:** analoga a sopra, ma si forniscono entrambi gli indirizzi al costruttore del `tab_iter`:

```
for(tab_iter it(tab, v_lo, v_hi); it; it.next()) {
  // it e' l'elemento corrente
}
```

- **Visita posticipata di una regione di indirizzi:** ancora analoga a sopra, ma si scende del tutto sfruttando la `post()`, e poi si prosegue con la `next_post()`:

```
tab_iter it(tab, v);
for (it.post(); it; it.next_post()} {
  // it e' l'elemento corrente
}
```

  Notiamo che la `post()` non implementa altro che la visita anticipata che abbiamo visto prima:

```
void tab_iter::post()
{
  // controlli di validita'
  if (done())
    return;

```

```
  while (down())
    ;
}
```

1.1.5 Trasformazione

Possiamo quindi definire la funzione `trasforma(paddr, vaddr)` che usa l'albero di traduzione puntato dall'indirizzo fisico al primo argomento per tradurre, in un altro indirizzo fisico, l'indirizzo virtuale al secondo argomento. Solitamente l'albero di traduzione che ci interesserà sara quello attualmente caricato in `readCR3()`, cioè:

```
// v e' un indirizzo virtuale
paddr p = trasforma(readCR3(), v);
```

L'implementazione della `trasforma()` si riduce effettivamente a:

1. Esegui una visita in ordine anticipato fino alla pagina corretta;
2. Verifica se la pagina è stata effettivamente ottenuta (altrimenti restituisci l'indirizzo 0);
3. Combina l'indirizzo fisico di pagina con l'offset di pagina.

cioè in codice:

```
paddr trasforma(paddr root_tab, vaddr v)
{
  // punto 1
  tab_iter it(root_tab, v);
  while (it.down())
    ;

  // punto 2
  tab_entry e = it.get_e();
  if (!(e & BIT_P))
    return 0;

  // punto 3 (con un dettaglio riguardante le huge page: si prende come
    offset la maschera ottenuta da dim_region())
  int l = it.get_l();
  natq mask = dim_region(l - 1) - 1;
  return (e & ~mask) | (v & mask);
}
```

1.1.6 map() e unmap()

Vediamo infine due funzioni che permettono di mappare e liberare regioni di memoria contigue nello spazio virtuale, ottenendo attraverso un qualche helper `alloca_tab()` nuovi frame fisici (anche non contigui) e rilasciandoli con `rilascia_tab()`. Queste sono la `map()` e la `unmap()`.

La chiamata della `map()` è:

```
vaddr map(paddr tab, vaddr begin, vaddr end, natl flags, T& getpaddr, int
    ps_lvl = 1)
```

cioè dobbiamo specificare la tabella madre, gli indirizzi virtuali di confine della regione, una double word contenente eventuali flag R/W o U/S, e una funzione per l'ottenimento sequenziale di indirizzi fisici, che viene passata come *funtore* o come *lambda*

(non si entra nei dettagli dello standard C++, va bene una funzione come una classe che ridefinisce l'operatore di chiamata).

La chiamata della `unmap()` è invece:

```
void unmap(paddr tab, vaddr begin, vaddr end, T& putpaddr)
```

dove vediamo servono principalmente gli stessi parametri, più la funzione `putpaddr()`, che può essere usata per fare pulizia degli indirizzi fisici non più usati.

Ad esempio, se questi vengono posti in memoria dinamica attraverso l'helper passato a `getpaddr` della `map()`, e si sfrutta qualche struttura dati che tiene conto di quali regioni di memoria fisica sono effettivamente in utilizzo, il parametro `putpaddr()` ci permette di definire una funzione che ripulisca i frame utilizzati una volta che questi vengono liberati, liberando le pagine di memoria corrispondente.

Un'ultima precisazione va fatta riguardo alle funzioni usate da `map()` e `unmap()`: queste saranno la `copy_des()` e la `set_des()` che avevamo visto in 16.1.2, più ottimizzate per la modifica in massa.