1 Lezione del 07-04-25

ha parlato dell'implementazione di map e unmap

1.0.1 Descrittori di frame e gestione

Abbiamo quindi visti come abbiamo bisogno di una struttura dati, contenuta in memoria sistema (M_1), che gestisce i frame di memoria nella parte alta (M_2). da qualche parte specifica sto discorso M_1 contro M_2 Questa struttura è implementata come un'array:

```
// descrittore di frame
struct des_frame {
  union {
    // numero di entrate valide (se il frame contiene una tabella)
    natw nvalide;
    // prossimo frame libero (se il frame e' libero)
    natl prossimo_libero;
  };
};

// array dei descrittori di frame
des_frame vdf[N_FRAME];
```

A questo punto si potra allocare e deallocare frame come segue:

- **Allocazione:** prendiamo il primo frame libero, che manteniamo in un apposita variabile (appositamente inizializzata), sostituiamo il suo puntatore a frame libero con il numero di entrate valide nullo, e prendiamo il suo puntatore a prossimo frame libero come nuovo puntatore globale, ovvero:

```
paddr alloca_frame()
{
  if (!num_frame_liberi) {
    flog(LOG_ERR, "out of memory");
    return 0;
  }
  natq j = primo_frame_libero;
  primo_frame_libero = vdf[primo_frame_libero].prossimo_libero;
  vdf[j].prossimo_libero = 0;
  num_frame_liberi--;
  return j * DIM_PAGINA;
}
```

- **Deallocazione:** prendiamo il frame come primo frame libero e impostiamo il suo puntatore a prossimo frame libero al puntatore a frame libero corrente, ovvero:

```
void rilascia_frame(paddr f)
{
  natq j = f / DIM_PAGINA;
  if (j < N_M1) {
    fpanic("tentativo di rilasciare il frame %lx di M1", f);
  }
  // dal momento che i frame di M2 sono tutti equivalenti, e'
  // sufficiente inserire in testa
  vdf[j].prossimo_libero = primo_frame_libero;
  primo_frame_libero = j;
  num_frame_liberi++;
}
```

1.0.2 Gestione di tabelle

Vorrremo usare le `alloca_frame()` e `rilascia_frame()` per allocare e deallocare intere tabelle di frame, attraverso le funzioni:

- **Allocazione:**

```
paddr alloca_tab()
{
  paddr f = alloca_frame();
  if (f) {
    memset(voidptr_cast(f), 0, DIM_PAGINA);
    vdf[f / DIM_PAGINA].nvalide = 0;
  }
  return f;
}
```

- **Deallocazione:**

```
void rilascia_tab(paddr f)
{
  if (int n = get_ref(f)) {
    fpanic("tentativo di deallocare la tabella %lx con %d entrate
    valide", f, n);
  }
  rilascia_frame(f);
}
```

Queste funzioni non sono altro che le `getpaddr` e `putpaddr` che davamo in argomento a le `map()` e `unmap()` per la gestione del trie.

In altre parole, staimo quindi gestendo l'albero di traduzione attraverso le funzioni `map()` e `unmap()`, e i frame di memoria fisica attraverso le `alloca_tab()` e `rilascia_tab()`.

1.1 Bootloader

Vediamo quindi il **bootloader**, cioè quella parte del kernel che si occupa di effettuare il *bootstrap* e portare il sistema in uno stato operativo.

Riguardo alla memoria virtuale, avremo che dovremo in sequenza:

1. Creare la radice dell'albero di traduzione;
2. Creare la finestra FM;
3. Prima di attivare la paginazione, caricare l'indirizzo fisico radice dell'albero di traduzione nel registro `CR3`;
4. Attivare la paginazione.

Cosa che facciamo come:

```
// punto 1
paddr root_tab = alloca_tab();
if (!root_tab) {
  flog(LOG_ERR, "ATTENZIONE: impossibile allocare la tabella radice");
  return;
}

// punto 2
if (!crea_finestra_FM(root_tab, mem_tot)) {
```

```
  flog(LOG_ERR, "ATTENZIONE: fallimento in crea_finestra_FM()");
  return;
}

// punto 3
loadCR3(root_tab);

// punto 4
// (equivale a comunicare con un interfaccia)
attiva_paginazione(info, info->mod[0].entry_point, MAX_LIV);
```

Potrebbe interessarci l'implementazione della `crea_finestra_FM()`. Questa parte creando una traduzione identità attraverso una *lambda*:

```
auto identity_map = [] (vaddr v) -> paddr { return v; };
```

e quindi mappando diverse regioni di memoria in base al loro scopo:

```
  // prima regione non mappata, interecetta nullptr
  natq first_reg = dim_region(1);

  // [0, DIM_PAGINA): non mappato
  // [DIM_PAGINA, 0xa0000): memoria normale
  if (map(root_tab, DIM_PAGINA, 0xa0000, BIT_RW, identity_map) != 0xa0000)
    return false;
  // [0xa0000, 0xc0000): memoria video
  if (map(root_tab, 0xa0000, 0xc0000, BIT_RW|BIT_PWT, identity_map) != 0
    xc0000)
    return false;
  // [0xc0000, first_reg): memoria normale
  if (map(root_tab, 0xc0000, first_reg, BIT_RW, identity_map) != first_reg
    )
    return false;

  // mappiamo il resto della memoria, se esiste, con PS settato
  if (mem_tot > first_reg) {
    if (map(root_tab, first_reg, mem_tot, BIT_RW, identity_map, 2) !=
    mem_tot)
      return false;
  }

  flog(LOG_INFO, "Creata finestra sulla memoria centrale:  [%16llx, %16llx
    )", DIM_PAGINA, mem_tot);

  // mappiamo tutti gli altri indirizzi, fino a 4GiB, settando sia PWT che
     PCD.
  // questa zona di indirizzi e' utilizzata in particolare dall'APIC per
    mappare i propri registri.
  vaddr beg_pci = allinea(mem_tot, 2*MiB),
    end_pci = 4*GiB;
  if (map(root_tab, beg_pci, end_pci, BIT_RW|BIT_PCD|BIT_PWT, identity_map
    , 2) != end_pci)
    return false;

  flog(LOG_INFO, "Creata finestra per memory-mapped-IO:    [%16llx, %16llx
    )", beg_pci, end_pci);
  return true;
```

Un dettaglio interessante è nella `attiva_paginazione()`. Questa è scritta in assembler come:

```
# settiamo il bit 31 di CR0
```

```
movl %cr0, %eax
orl $0x80010000, %eax # paging & write-protect
movl %eax, %cr0
# da qui in poi la MMU e' attiva
```

Visto che dall'esecuzione della `MOVL` in poi il processore emetterà indirizzi che verrano tradoti dalla MMU, sarà necessario che l'indirizzo puntato in quel momento dal RIP sia contenuto nella finestra creata prima, così che si mantenga la continuità fra le istruzioni del programma.

1.2 Partizione della memoria nel nucleo

qui ci dovrebbe essere lo schema sulla FM d prima

Abbiamo quindi visto come la memoria indirizzabile è divisa in due regioni da 2^47 bit ciascuna (cioè la divisione data dagli indirizzi a 48 bit normalizzati). Vediamo come questa memoria è divisa nel nucleo. Abbiamo che nella regione bassa allochiamo memoria sistema, come segue:

- **Memoria sistema:**
  - **Memoria sistema condivisa**: qui si manterranno informazioni riguardo ai frame della memoria M_2 (quella al di sopra dela partizione), e alle tabelle, in un apposita struttura dati (un array). La struttura dati contiene, fra l'altro, anche il contatore delle entrate valide di ogni tabella (che abbiamo visto prima viene consultato dalla `unmap()` per effettuare rimozioni di tabelle);
  - **Memoria sistema privata** (**pila sistema**);
  - **Memoria sistema del modulo I/O condivisa**.

La parte alta alloca invece memoria utente, come segue:

- **Memoria utente:**
  - **Memoria utente condivisa** (**codice** e **heap**): questa è organizzata perché qualsiasi processo in esecuzione la mappi sempre nella stessa regione;
  - **Memoria utente privata** (**pila utente**): questa è organizzata perché ogni processo mappi la *sua* pila nella stessa regione.

Facciamo alcune semplificazioni riguardo a questa struttura:

- Tutte le parti di livello più alte vengono create come multipli di 512 GiB, in modo che occupino intere entrate di livello 4;
- Le parti condivise sono *"fisse"*, riferite da tabelle di livello 3 che vengono puntate nuovamente in ogni tabella di livello 4 che creiamo come radice degli alberi di traduzione di ogni processo (e che sono le stesse dell'albero di traduzione del nucleo).

Vediamo che le tabelle di livello 3 della parte utente e sistema condivise si possono quindi creare una volta sola all'avvio del sistema (si dovrebbe ricaricare la parte codice dei processi nel caso di un sistema che carica software dal disco). Questa allocazione viene fatta usando la `map()` aiutata da `alloca_frame()`, per ottenere memoria e disporre traduzioni per regioni di memoria di indirizzi prestabiliti (che seguono lo schema definito finora).

Le uniche cose che vanno quindi create da zero ogni volta che si crea un processo sono la **pila sistema** in memoria sistema privata e la **pila utente** in memoria utente privata.

Avremo quindi che alla creazione di un nuovo processo dovremo creare una *nuova* tabella di livello 4, che puntera alle tabelle di livello 3 delle parti condivise (memoria utente e sistema condivisa), già esistenti, e che creerà nuove tabelle di livello 3, e quindi di livello 2, ecc... per le parti private (pila utente e pila sistema).

1.3 Creazione di processi

Possiamo quindi vedere più nel dettaglio la creazione di processi.

1.3.1 Albero di traduzione

Abbiamo detto avevamo bisogno di creare nuove tabelle di livello 4 per ogni processo. Facciamo questo come segue:

```
des_proc* crea_processo(void f(natq), natq a, int prio, char liv) {
  [...]

  p->cr3 = alloca_tab(); // la nuova tabella di livello 4
  if (p->cr3 == 0)
    goto err_rel_id;
  init_root_tab(p->cr3);

  [...]
}
```

dove la `init_root_tab()`, come avevamo detto, si limita a copiare le tabelle di livello 3 delle parti condivise:

```
void init_root_tab(paddr dest) {
  // cr3 del processo corrente
  paddr pdir = esecuzione->cr3;

  // copia le tabelle di livello 3
  copy_des(pdir, dest, I_SIS_C, N_SIS_C);
  copy_des(pdir, dest, I_MIO_C, N_MIO_C);
  copy_des(pdir, dest, I_UTN_C, N_UTN_C);
}
```

quali sono? scrivi anche sopra penso

ha la duale clear$_root_tab$

1.3.2 Pila

Veniamo quindi all'inizializzazione della pila. Questa si fa, sia per la pila utente che per la pila sistema, attraverso la `crea_pila()`:

```
bool crea_pila(paddr root_tab, vaddr bottom, natq size, natl liv)
{
  vaddr v = map(root_tab,
    bottom - size,
    bottom,
    BIT_RW | (liv == LIV_UTENTE ? BIT_US : 0),
    [](vaddr) { return alloca_frame(); });
  if (v != bottom) {
    unmap(root_tab, bottom - size, v,
      [](vaddr, paddr p, int) { rilascia_frame(p); });
```

```
    return false;
  }
  return true;
}
```

che fa cose

Questa ha ancora la duale, `distruggi_pila()`:

```
void distruggi_pila(paddr root_tab, vaddr bottom, natq size)
{
  unmap(
    root_tab,
    bottom - size,
    bottom,
    [](vaddr, paddr p, int) { rilascia_frame(p); });
}
```

commenta anche lei

capisci discorso pila utente / pila sistema