

# 1 Lezione del 07-04-25

## 1.0.1 Implementazione di map() e unmap()

Vediamo brevemente come sono implementate in pratica la map() e la unmap().

- map(): si basa su una visita *anticipata* del trie:

```
1 vaddr map(paddr tab, vaddr begin, vaddr end, natl flags, T& getpaddr,
2     int ps_lvl = 1)
3 {
4     vaddr v; /* indirizzo virtuale corrente */
5     int l; /* livello (del TRIE) corrente */
6     natq dr; /* dimensione delle regioni di livello ps_lvl */
7     [...]
8     // controlli
9     [...]
10
11     // usiamo un iteratore di tabella per effettuare una visita
12     // anticipata
13     tab_iter it(tab, begin, end - begin);
14     for ( /* niente */ ; it; it.next()) {
15         tab_entry& e = it.get_e();
16         l = it.get_l();
17         v = it.get_v();
18
19         // new verra' popolato dal nuovo indirizzo fisico da collegare a
20         // una tabella o a una traduzione, indistintamente
21         paddr new_f = 0;
22
23         if (l > ps_lvl) {
24             // nodo non foglia
25
26             if (!(e & BIT_P)) {
27                 // va allocata una tabella, chiama alloca_tab()
28                 new_f = alloca_tab(); // caso 1) si crea una nuova tabella
29
30                 // controlli
31                 [...]
32
33             } else if (e & BIT_PS) {
34                 // errore: e' una huge page
35             }
36         } else {
37             // va allocata una traduzione, chiama get_paddr()
38             new_f = getpaddr(v); // caso 2) si crea una nuova traduzione
39
40             // controlli
41             [...]
42
43             // configura i flag
44             if (l > 1)
45                 e |= BIT_PS;
46
47             e |= (flags & (BIT_PWT|BIT_PCD));
48         }
49
50         if (new_f) {
51             // siamo qui per il caso 1) o per il caso 2), cioe':
52             // - caso 1) bisogna creare una tabella
```

```

51 // - caso 2) bisogna creare una traduzione
52 // in entrambi i casi si fanno le stesse operazioni
53
54 // 'e' non puntava a niente e ora deve puntare a new_f
55 set_IND_FISICO(e, new_f);
56 e |= BIT_P;
57
58 // dobbiamo incrementare il contatore delle entrate
59 // valide della tabella a cui 'e' appartiene
60 inc_ref(it.get_tab());
61 }
62
63 // configura altri flag
64 e |= (flags & (BIT_RW|BIT_US));
65 }
66 return end;

```

Vediamo quindi che la situazione rispetto all'ultima volta si complica: non abbiamo bisogno soltanto della `getpaddr()`, per l'ottenimento degli indirizzi fisici, ma anche della `alloca_tab()` per l'allocazione di tabelle del trie. Questo ha senso, in quanto la tabella si distingue dal semplice frame di collegato a una pagina, per il fatto che necessita di un contatore di entrate valide che ne faciliti la pulizia in caso di inutilizzo. Come vedremo, in ogni caso, sia la `getpaddr()` che la `alloca_tab()` vengono spesso definite, ad esempio nel nucleo, sulla base della stessa funzione helper per l'ottenimento di memoria libera (`allocafree()`);

- `unmap()`: si basa su una visita *posticipata* del trie:

```

1 void unmap(paddr tab, vaddr begin, vaddr end, T& putpaddr)
2 {
3     // usiamo un iteratore di tabella per effettuare una visita
4     // posticipata
5     tab_iter it(tab, begin, end - begin);
6     for (it.post(); it; it.next_post()) {
7         tab_entry& e = it.get_e();
8
9         // non eliminare tabelle non allocate
10        if (!(e & BIT_P))
11            continue;
12
13        paddr p = extr_IND_FISICO(e);
14        if (!it.is_leaf()) {
15            // l'entrata punta a una tabella.
16
17            // qui entra in gioco il numero di entrate valide:
18            // la get_ref() ci permette di ottenere le sottotabelle con P
19            // alto
20
21            if (!get_ref(p)) {
22                // se la tabella non contiene piu' entrate
23                // valide la deallochiamo
24                rilascia_tab(p);
25            } else {
26                // altrimenti non facciamo niente
27                // (la tabella serve per traduzioni esterne
28                // all'intervallo da eliminare)
29                continue;
30            }
31        } else {
32            // se la tabella non contiene piu' entrate
33            // valide la deallochiamo
34            rilascia_tab(p);
35        }
36    }
37 }

```

```

30 // l'entrata punta ad una pagina (di livello it.get_l())
31 vaddr v = it.get_v();
32 int l = it.get_l();
33
34 // controlli
35 [...]
36
37 putpaddr(v, p, l);
38 }
39
40 // azzeriamo l'entrata di tabella
41 e = 0;
42 // decrementiamo i riferimenti
43 dec_ref(it.get_tab());
44 }
45 }

```

Vediamo qui ancor meglio come è necessario mantenere separatamente il numero di sottotabelle occupate di una tabella, in modo da capire quando si può procedere alla tabella con `rilascia_tab()`, o quando questa mantiene ancora sottotabelle utili ad altre traduzioni. L'eliminazione delle traduzioni stesse, e quindi delle locazioni fisiche allocate, invece, viene svolta dalla `putpaddr()`.

## 1.1 Gestione della memoria fisica

Osservando come la `map()` e la `unmap()` hanno bisogno di funzioni (`alloca_tab()` e `dealloca_tab()`, nonché `getpaddr()` e `putpaddr()`, comunque queste siano implementate) che si occupano di ottenere effettivamente memoria fisica. Vediamo come queste vengono implementate.

### 1.1.1 Descrittori di frame

Ci rendiamo quindi conto di aver bisogno di una struttura dati, contenuta in memoria sistema ( $M_1$ ), che gestisce i frame di memoria nella parte alta ( $M_2$ ). Questa struttura è implementata come un'array:

```

1 // descrittore di frame
2 struct des_frame {
3     union {
4         // numero di entrate valide (se il frame contiene una tabella)
5         natw nvalide;
6         // prossimo frame libero (se il frame e' libero)
7         natl prossimo_libero;
8     };
9 };
10
11 // array dei descrittori di frame
12 des_frame vdf[N_FRAME];

```

### 1.1.2 Gestione dei frame

Definiti descrittori di frame, si potrà allocare e deallocare 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:

```

1 paddr alloca_frame() {
2     if (!num_frame_liberi) {
3         flog(LOG_ERR, "out of memory");
4         return 0;
5     }
6     natq j = primo_frame_libero;
7     primo_frame_libero = vdf[primo_frame_libero].prossimo_libero;
8     vdf[j].prossimo_libero = 0;
9     num_frame_liberi--;
10    return j * DIM_PAGINA;
11 }

```

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

```

1 void rilascia_frame(paddr f) {
2     natq j = f / DIM_PAGINA;
3     if (j < N_M1) {
4         fpanic("tentativo di rilasciare il frame %lx di M1", f);
5     }
6     // dal momento che i frame di M2 sono tutti equivalenti, e'
7     // sufficiente inserire in testa
8     vdf[j].prossimo_libero = primo_frame_libero;
9     primo_frame_libero = j;
10    num_frame_liberi++;
11 }

```

### 1.1.3 Gestione di tabelle

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

- **Allocazione:**

```

1 paddr alloca_tab() {
2     paddr f = alloca_frame();
3     if (f) {
4         memset(voidptr_cast(f), 0, DIM_PAGINA);
5         vdf[f / DIM_PAGINA].nvalide = 0;
6     }
7     return f;
8 }

```

- **Deallocazione:**

```

1 void rilascia_tab(paddr f) {
2     if (int n = get_ref(f)) {
3         fpanic("tentativo di deallocare la tabella %lx con %d entrate valide", f, n);
4     }
5     rilascia_frame(f);
6 }

```

Queste funzioni sono proprio quelle che davamo in argomento a `map()` e `unmap()` per la gestione del trie. In altre parole, stiamo gestendo l'albero di traduzione attraverso le funzioni `map()` e `unmap()`, le tabelle attraverso le `alloca_tab()` e `rilascia_tab()`, e i frame di memoria fisica attraverso le `alloca_frame()` e `rilascia_frame()`.

## 1.2 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:

```

1 // punto 1
2 paddr root_tab = alloca_tab();
3 if (!root_tab) {
4     flog(LOG_ERR, "ATTENZIONE: impossibile allocare la tabella radice");
5     return;
6 }
7
8 // punto 2
9 if (!crea_finestra_FM(root_tab, mem_tot)) {
10     flog(LOG_ERR, "ATTENZIONE: fallimento in crea_finestra_FM()");
11     return;
12 }
13
14 // punto 3
15 loadCR3(root_tab);
16
17 // punto 4
18 // (equivale a comunicare con un interfaccia)
19 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*:

```

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

```

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

```

1 // prima regione non mappata, intercetta nullptr
2 natq first_reg = dim_region(1);
3
4 // [0, DIM_PAGINA): non mappato
5 // [DIM_PAGINA, 0xa0000): memoria normale
6 if (map(root_tab, DIM_PAGINA, 0xa0000, BIT_RW, identity_map) != 0xa0000)
7     return false;
8 // [0xa0000, 0xc0000): memoria video
9 if (map(root_tab, 0xa0000, 0xc0000, BIT_RW|BIT_PWT, identity_map) != 0xc0000)
10     return false;
11 // [0xc0000, first_reg): memoria normale
12 if (map(root_tab, 0xc0000, first_reg, BIT_RW, identity_map) != first_reg)
13     return false;
14
15 // mappiamo il resto della memoria, se esiste, con PS settato
16 if (mem_tot > first_reg) {

```

```

17     if (map(root_tab, first_reg, mem_tot, BIT_RW, identity_map, 2) !=
18         mem_tot)
19         return false;
20 }
21 flog(LOG_INFO, "Creata finestra sulla memoria centrale:  [%16llx, %16llx
22     )", DIM_PAGINA, mem_tot);
23 // qui la memoria e' finita, tutto quello che segue interessa al bus PCI
24 // mappiamo tutti gli altri indirizzi, fino a 4GiB, settando sia PWT che
25 // PCD.
26 // questa zona di indirizzi e' utilizzata in particolare dall'APIC per
27 // mappare i propri registri.
28 vaddr beg_pci = allinea(mem_tot, 2*MiB),
29     end_pci = 4*GiB;
30 if (map(root_tab, beg_pci, end_pci, BIT_RW|BIT_PCD|BIT_PWT, identity_map
31     , 2) != end_pci)
32     return false;
33 flog(LOG_INFO, "Creata finestra per memory-mapped-I/O:  [%16llx, %16llx
34     )", beg_pci, end_pci);
35 return true;

```

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

```

1 # settiamo il bit 31 di CR0
2 movl %cr0, %eax
3 orl $0x80010000, %eax # paging & write-protect
4 movl %eax, %cr0
5 # da qui in poi la MMU e' attiva

```

Visto che dall'esecuzione della `MOVL` in poi il processore emetterà indirizzi che verranno tradotti 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.

### 1.3 Partizione della memoria nel nucleo

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:**

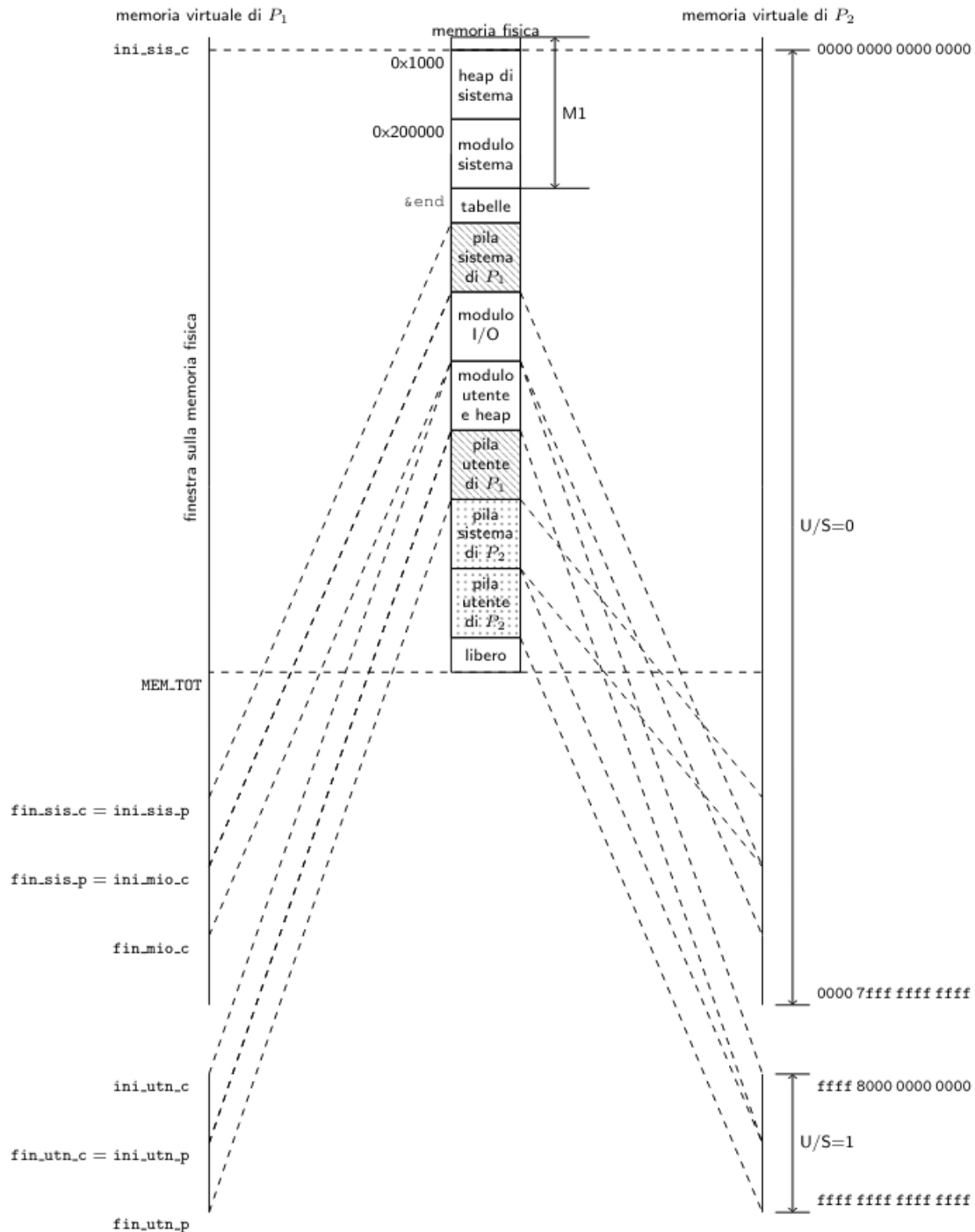
- **Memoria sistema condivisa:** qui si manterranno informazioni riguardo ai frame della memoria  $M_2$  (quella al di sopra della partizione), e alle tabelle, in un apposita struttura dati (un array). La struttura dati contiene, fra l'altro, anche il contatore delle entrate valide di ogni tabella (che 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.

Lo schema complessivo è quindi del tipo seguente:



in questo caso relativa ai trie di due processi, da cui notiamo ancora come la prima pagina non è tradotta, in quanto rappresenta l'area raggiungibile dai nullptr.

Facciamo allora alcune semplificazioni riguardo a questa struttura, in modo da avvicinarci ad un'implementazione effettiva:

- 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. Queste regioni seguono lo schema visto finora, che viene definito in codice come:

```

1 #define I_SIS_C 0 // prima entrata sistema/condivisa (inizio M1)
2 #define I_SIS_P 1 // prima entrata sistema/privata
3 #define I_MIO_C 2 // prima entrata modulo IO/condivisa
4 #define I_UTN_C 256 // prima entrata utente/condivisa (inizio M2)
5 #define I_UTN_P 384 // prima entrata utente/privata
6
7 #define N_SIS_C 1 // numero entrate sistema/condivisa
8 #define N_SIS_P 1 // numero entrate sistema/privata
9 #define N_MIO_C 1 // numero entrate modulo IO/condivisa
10 #define N_UTN_C 128 // numero entrate utente/condivisa
11 #define N_UTN_P 128 // numero entrate utente/privata

```

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 punterà 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).

### 1.3.1 Sequenza di boot

Possiamo quindi vedere nel dettaglio la sequenza di boot e avvio del kernel, anticipando alcuni dettagli sull'I/O che verranno meglio spiegati nella sezione 18. Abbiamo che il processo di boot si autodocumenta stampando messaggi sulla porta seriale (attraverso la funzione di `libce.h` chiamata `flog()`). Vediamo quindi l'intero processo dividendolo in fasi e controllando cosa viene stampato.

#### 1. Bootloader:

```

1 INF - Boot loader di Calcolatori Elettronici, v1.0
2 INF - Memoria totale: 32 MiB, heap: 636 KiB
3 INF - Argomenti: ~/CE/lib/ce/boot.bin
4 INF - Il boot loader precedente ha caricato 3 moduli:
5 INF - - mod[0]: start=10d000 end=117ba0 file=boot/0-sistema
6 INF - . seg[1]: off 0 vaddr 200000 size 6973
   memsize 6973
7 INF - . seg[2]: off 7000 vaddr 207000 size 2f24
   memsize 2f24

```



```

8 INF - . seg[3]: off      a000 vaddr          20a000 size      180
    memsize 17404
9 INF - - mod[1]: start=118000 end=11e5d8 file=boot/1-io
10 INF - . seg[1]: off      0 vaddr          10000000000 size      3ddf
    memsize 3ddf
11 INF - . seg[2]: off     4000 vaddr          10000004000 size      19ac
    memsize 19ac
12 INF - . seg[3]: off     6000 vaddr          10000006000 size      190
    memsize 200
13 INF - - mod[2]: start=11f000 end=1224b0 file=boot/2-utente
14 INF - . seg[1]: off      0 vaddr ffff800000000000 size      1a69
    memsize 1a69
15 INF - . seg[2]: off     2000 vaddr ffff8000000002000 size      974
    memsize 974
16 INF - . seg[3]: off     3000 vaddr ffff8000000003000 size       60
    memsize 1690
17 INF - Copio mod[0] agli indirizzi specificati nel file ELF:
18 INF - - copiati 6973 byte da 10d000 a 200000
19 INF - - copiati 2f24 byte da 114000 a 207000
20 INF - - copiati 180 byte da 117000 a 20a000
21 INF - - azzerati ulteriori 17284 byte
22 INF - - entry point 20569b
23 INF - Crea finestra sulla memoria centrale: [          1000,
    2000000)
24 INF - Crea finestra per memory-mapped-I/O: [          2000000,
    100000000)
25 INF - Inizializzo l'APIC
26 INF - Inizializzo video e tastiera
27 INF - Attivo la modalita' a 64 bit e cedo il controllo a mod[0]...

```

Inizia il bootloader, definito in `libce.h`. Questo non è il primo bootloader che viene lanciato sulla macchina virtuale: QEMU dispone infatti del suo bootloader, che si occupa di ottenere informazioni di base sulla memoria (che vengono visualizzate alla riga 2) e caricare dal disco le immagini dei moduli da caricare (riga 3).

Questi moduli sono, nel dettaglio:

- Il modulo sistema, contenuto in `mod[0]`;
- Il modulo di I/O, contenuto in `mod[1]`;
- Il modulo utente, contenuto in `mod[2]`.

Ogni modulo si porta dietro le sue sezioni definite nell'**ELF** (*Executable and Linkable Format*), divise in 3 *segmenti*:

- 1) Il primo segmento è di tipo **testo** e contiene il codice del modulo, quindi va caricato in sola lettura;
- 2) Il secondo segmento è di tipo **dati in sola lettura**, quindi va caricato in sola lettura;
- 3) Il terzo segmento è di tipo **dati** (variabili inizializzate, fra cui lo spazio da dedicare a tutte le varie tabelle di descrittori, da cui la grande quantità di spazio `memsize` da liberare senza scrivere), quindi va caricato in lettura/scrittura.

Il bootloader si occupa quindi di caricare solo il primo modulo, quello sistema, da 200000 in poi (righe 17-22), in quanto non sa come questo vorrà gestire I/O e utente.

Procede quindi alla creazione della finestra FM, come abbiamo visto dalla `crea_finestra_FM()` (righe 23 e 24). Infine, inizializza l'APIC alla riga 25, video e tastiera alla riga 26,

semplicemente configurando i registri giusti (ad esempio attiva le interruzioni tastiera e disattiva il PIC), attiva la paginazione e cede il controllo a `mod[0]`, cioè al modulo sistema, alla riga 27.

## 2. Kernel:

```

1 INF 0 Nucleo di Calcolatori Elettronici, v8.2
2 INF 0 Numero di frame: 546 (M1) 7646 (M2)
3 INF 0 Suddivisione della memoria virtuale:
4 INF 0 - sis/cond [          0,          8000000000)
5 INF 0 - sis/priv [      8000000000,      10000000000)
6 INF 0 - io /cond [      10000000000,      18000000000)
7 INF 0 - usr/cond [ffff800000000000, ffffc00000000000)
8 INF 0 - usr/priv [ffffc00000000000,          0)
9 INF 0 Carico il modulo I/O
10 INF 0 - segmento sistema read-only mappato a [      10000000000,
      10000004000)
11 INF 0 - segmento sistema read-only mappato a [      10000004000,
      10000006000)
12 INF 0 - segmento sistema read/write mappato a [      10000006000,
      10000007000)
13 INF 0 - heap: [      10000007000,
      10000107000)
14 INF 0 - entry point: _start [io.s:17]
15 INF 0 Carico il modulo utente
16 INF 0 - segmento utente read-only mappato a [ffff800000000000,
      ffff800000002000)
17 INF 0 - segmento utente read-only mappato a [ffff800000002000,
      ffff800000003000)
18 INF 0 - segmento utente read/write mappato a [ffff800000003000,
      ffff800000005000)
19 INF 0 - heap: [ffff800000005000,
      ffff800000105000)
20 INF 0 - entry point: _start [utente.s:15]
21 INF 0 Frame liberi: 7116 (M2)
22 INF 0 Heap del modulo sistema: aggiunto [100000, 200000)
23 INF 0 Attivo il timer (DELAY=59659)
24 INF 0 Creo il processo main I/O
25 INF 0 Attendo inizializzazione modulo I/O...

```

Il kernel prende quindi il controllo della macchina. A questo punto ha accesso alla finestra FM, quindi può controllare il numero di frame liberi e creare la partizione di memoria. L'albero sistema come definito in 17.3 (righe da 2 a 8). Notiamo come le regioni che crea coincidono con la dimensione delle entrate di livello 4 (512 GiB ciascuna). Inoltre, notiamo il salto alla prima tabella utente, da  $M_1$  a  $M_2$  (indirizzi che iniziano con  $0xffff\dots$ ).

Si inizia quindi il caricamento del modulo I/O (riga 9), mappato nella regione I/O condivisa, e del modulo utente (riga 15), mappato nella memoria utente condivisa.

Si sceglie poi l'entry point del modulo utente (riga 20), si aumenta la dimensione dell'heap sistema (inizializzato dal primo bootloader) (riga 22), si attiva il timer (riga 23) e si crea il processo I/O (riga 24), che da qui in poi prenderà il controllo.

## 3. I/O:

```

1 INF 1 Heap del modulo I/O: 100000B [0x10000007000, 0x10000107000)
2 INF 1 Inizializzo la console (kbd + vid)
3 INF 1 estern=2 entry=estern_kbd(unsigned long) [io.cpp:197](0) prio
      =1104 (tipo=50) liv=0 irq=1
4 INF 1 kbd: tastiera inizializzata
5 INF 1 vid: video inizializzato
6 INF 1 Inizializzo la gestione dell'hard disk
7 INF 1 bm: 00:01.1

```

```

8 INF 1 estern=3 entry=estern_hd(unsigned long) [io.cpp:557](0) prio
    =1120 (tipo=60) liv=0 irq=14
9 INF 1 Processo 1 terminato

```

Il modulo di I/O fa un pò di configurazione aggiuntiva: inizializza la console avviando il processo `estern_kbd` (riga 3). In seguito, deve *trovare* l'unico dispositivo finora non considerato, cioè l'hard disk (riga 6) che trova sul bus principale al dispositivo 1, funzione 1 (vedremo in seguito che questo corrisponde al bus PCI-ATA del PIIX3). Una volta trovato l'hard disk, lo attiva gestendolo col processo `estern_hd` (riga 8).

In seguito, termina e restituisce il controllo al kernel.

#### 4. Utente:

```

1 INF 0 Creo il processo main utente
2 INF 0 Cedo il controllo al processo main utente...
3 INF 4 Heap del modulo utente: 100000B [0xffff800000004690, 0
    xffff800000104690)

```

L'ultimo a prendere il controllo è l'utente, attraverso il kernel che lo attiva dall'entry point trovato prima. Da qui in poi ad eseguire è il codice che definiamo noi, interfacciandosi con le primitive di kernel e i processi di I/O.

### 1.4 Creazione di processi in memoria

Possiamo quindi vedere più nel dettaglio la creazione di processi, in particolare riguardo alla memoria e alle tabelle e traduzioni create.

#### 1.4.1 Albero di traduzione

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

```

1 des_proc* crea_processo(void f(natq), natq a, int prio, char liv) {
2     [...]
3
4     p->cr3 = alloca_tab(); // la nuova tabella di livello 4
5     if (p->cr3 == 0)
6         goto err_rel_id;
7     init_root_tab(p->cr3);
8
9     [...]
10 }

```

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

```

1 void init_root_tab(paddr dest) {
2     // cr3 del processo corrente
3     paddr pdir = esecuzione->cr3;
4
5     // copia le tabelle di livello 3
6     copy_des(pdir, dest, I_SIS_C, N_SIS_C);
7     copy_des(pdir, dest, I_MIO_C, N_MIO_C);
8     copy_des(pdir, dest, I_UTN_C, N_UTN_C);
9 }

```

Questa ha una duale, che semplicemente libera le entrate create:

```

1 void clear_root_tab(paddr dest) {
2     // eliminiamo le entrate create da init_root_tab()
3     set_des(dest, I_SIS_C, N_SIS_C, 0);
4     set_des(dest, I_MIO_C, N_MIO_C, 0);
5     set_des(dest, I_UTN_C, N_UTN_C, 0);
6 }

```

### 1.4.2 Pila

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

```

1 bool crea_pila(paddr root_tab, vaddr bottom, natq size, natl liv)
2 {
3     vaddr v = map(root_tab,
4         bottom - size,
5         bottom,
6         BIT_RW | (liv == LIV_UTENTE ? BIT_US : 0),
7         [](vaddr) { return alloca_frame(); });
8
9     // caso di errore
10    if (v != bottom) {
11        unmap(root_tab, bottom - size, v,
12            [](vaddr, paddr p, int) { rilascia_frame(p); });
13        return false;
14    }
15    return true;
16 }

```

che ottiene una pila di una dimensione prestabilita allocando i frame necessari.

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

```

1 void distruggi_pila(paddr root_tab, vaddr bottom, natq size) {
2     unmap(
3         root_tab,
4         bottom - size,
5         bottom,
6         [](vaddr, paddr p, int) { rilascia_frame(p); });
7 }

```

che si limita a liberare i frame usati.

Dal punto di vista della `crea_processo()`, quindi, vogliamo prima inizializzare la pila sistema, e poi:

- Se siamo in contesto utente:
  1. Inizializzare la pila sistema;
  2. Creare la pila utente;
  3. Inizializzare la pila utente.
- Se invece siamo in contesto sistema, ci limitiamo ad inizializzare la pila sistema.

Questo in codice si traduce come:

```

1 // creazione della pila sistema
2 static_assert(DIM_SYS_STACK > 0 && (DIM_SYS_STACK & 0xFFF) == 0);
3
4 // siamo in un altro processo, quindi dobbiamo accedere alla pila sistema
  tramite la finestra FM

```

```

5 pila_sistema = trasforma(p->cr3, fin_sis_p - 1) + 1;
6
7 // convertiamo a puntatore a natq, per accedervi piu' comodamente
8 pl = ptr_cast<natq>(pila_sistema);
9
10 if (liv == LIV_UTENTE) {
11     // processo di livello utente
12     // inizializzazione della pila sistema
13     pl[-5] = int_cast<natq>(f);           // RIP (codice utente)
14     pl[-4] = SEL_CODICE_UTENTE;          // CS (codice utente)
15     pl[-3] = BIT_IF;                     // RFLAGS
16     pl[-2] = fin_utn_p - sizeof(natq);   // RSP
17     pl[-1] = SEL_DATI_UTENTE;            // SS (pila utente)
18
19     // eseguendo una IRET da questa situazione, il processo
20     // passerà ad eseguire la prima istruzione della funzione f,
21     // usando come pila la pila utente (al suo indirizzo virtuale)
22
23     // creazione della pila utente
24     static_assert(DIM_USR_STACK > 0 && (DIM_USR_STACK & 0xFFF) == 0);
25
26     // inizialmente, il processo si trova a livello sistema, come
27     // se avesse eseguito una istruzione INT, con la pila sistema
28     // che contiene le 5 parole quaduple preparate precedentemente
29     p->contesto[I_RSP] = fin_sis_p - 5 * sizeof(natq);
30
31     p->livello = LIV_UTENTE;
32
33     // dal momento che usiamo traduzioni diverse per le parti sistema/
34     // private
35     // di tutti i processi, possiamo inizializzare p->punt_nucleo con un
36     // indirizzo (virtuale) uguale per tutti i processi
37     p->punt_nucleo = fin_sis_p;
38
39     // tutti gli altri campi valgono 0
40 } else {
41     // processo di livello sistema
42     // inizializzazione della pila sistema
43     pl[-6] = int_cast<natq>(f);           // RIP (codice sistema)
44     pl[-5] = SEL_CODICE_SISTEMA;          // CS (codice sistema)
45     pl[-4] = BIT_IF;                     // RFLAGS
46     pl[-3] = fin_sis_p - sizeof(natq);   // RSP
47     pl[-2] = 0;                          // SS
48     pl[-1] = 0;                          // ind. rit.
49
50     // inizializzazione del descrittore di processo
51     p->contesto[I_RSP] = fin_sis_p - 6 * sizeof(natq);
52
53     p->livello = LIV_SISTEMA;
54
55     // tutti gli altri campi valgono 0
56 }

```