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:

```
vaddr map(paddr tab, vaddr begin, vaddr end, natl flags, T& getpaddr,
     int ps_lvl = 1)
{
  vaddr v;  /* indirizzo virtuale corrente */
  int l;    /* livello (del TRIE) corrente */
  natq dr;  /* dimensione delle regioni di livello ps_lvl */
  [...]

  // controlli
  [...]

  // usiamo un iteratore di tabella per effettuare una visita
    anticipata
  tab_iter it(tab, begin, end - begin);
  for ( /* niente */ ; it; it.next()) {
    tab_entry& e = it.get_e();
    l = it.get_l();
    v = it.get_v();

    // new verra' popolato dal nuovo indirizzo fisico da collegare a
    una tabella o a una traduzione, indistintamente
    paddr new_f = 0;

    if (l > ps_lvl) {
      // nodo non foglia

      if (!(e & BIT_P)) {
        // va allocata una tabella, chiama alloca_tab()
        new_f = alloca_tab(); // caso 1) si crea una nuova tabella

        // controlli
        [...]

      } else if (e & BIT_PS) {
        // errore: e' una huge page
      }
    } else {
      // va allocata una traduzione, chiama get_paddr()
      new_f = getpaddr(v);  // caso 2) si crea una nuova traduzione

      // controlli
      [...]

      // configura i flag
      if (l > 1)
        e |= BIT_PS;

      e |= (flags & (BIT_PWT|BIT_PCD));
    }

    if (new_f) {
      // siam qui per il caso 1) o per il caso 2), cioe':
      // - caso 1) bisogna creare una tabella
```

```
      // - caso 2) bisogna creare una traduzione
      // in entrambi i casi si fanno le stesse operazioni

      // 'e' non puntava a niente e ora deve puntare a new_f
      set_IND_FISICO(e, new_f);
      e |= BIT_P;

      // dobbiamo incrementare il contatore delle entrate
      // valide della tabella a cui 'e' appartiene
      inc_ref(it.get_tab());
    }

    // configura altri flag
    e |= (flags & (BIT_RW|BIT_US));
  }
  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 facilità 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 (`allocaframe()`);

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

```
void unmap(paddr tab, vaddr begin, vaddr end, T& putpaddr)
{
  // usiamo un iteratore di tabella per effettuare una visita
    posticipata
  tab_iter it(tab, begin, end - begin);
  for (it.post(); it; it.next_post()) {
    tab_entry& e = it.get_e();

    // non eliminare tabelle non allocate
    if (!(e & BIT_P))
      continue;

    paddr p = extr_IND_FISICO(e);
    if (!it.is_leaf()) {
      // l'entrata punta a una tabella.

      // qui entra in gioco il numero di entrate valide:
      // la get_ref() ci permete di ottenere le sottotabelle con P
    alto

      if (!get_ref(p)) {
        // se la tabella non contiene piu' entrate
        // valide la deallochiamo
        rilascia_tab(p);
      } else {
        // altrimenti non facciamo niente
        // (la tabella serve per traduzioni esterne
        // all'intervallo da eliminare)
        continue;
      }
    } else {
```

```
      // l'entrata punta ad una pagina (di livello it.get_l())
      vaddr v = it.get_v();
      int l = it.get_l();

      // controlli
      [...]

      putpaddr(v, p, l);
    }

    // azzeriamo l'entrata di tabella
    e = 0;
    // decrementiamo i riferimenti
    dec_ref(it.get_tab());
  }
}
```

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:

```
// 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];
```

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:

```
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++;
}
```

1.1.3 Gestione di tabelle

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

- **Allocazione:**

```
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 sono proprio quelle che davamo in argomento a le `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:

```
// 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.

1.3 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:

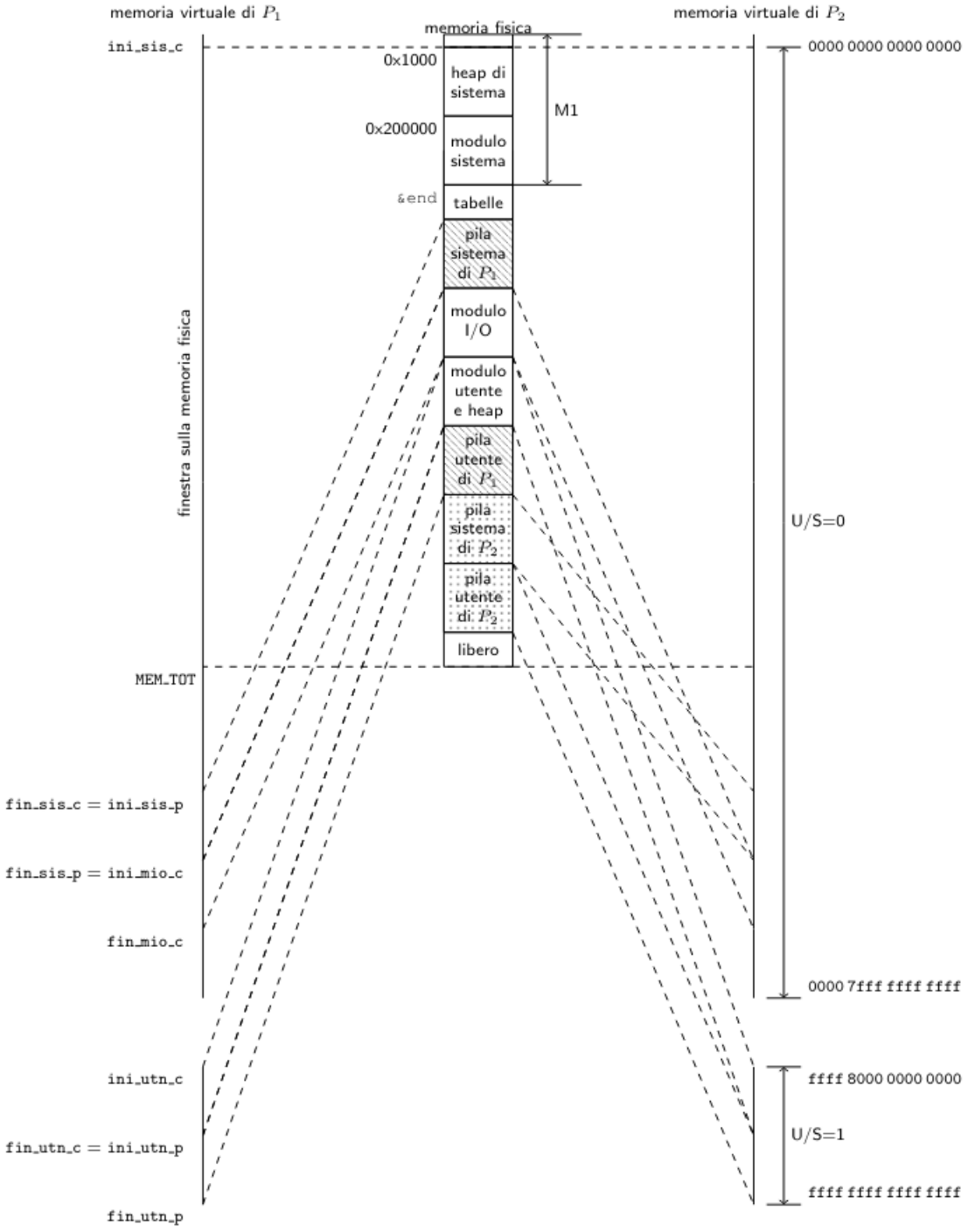
- **Memoria sistema:**
  - **Memoria sistema condivisa**: qui si manterranno informazioni riguardo ai frame della memoria M_2 (quella al di sopra dela partizione), e alle tabelle, in un apposita struttura dati (un array). La struttura dati contiene, fra l'altro, anche il contatore delle entrate valide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

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:

```
#define I_SIS_C   0   // prima entrata sistema/condivisa (inizio M1)
#define I_SIS_P   1   // prima entrata sistema/privata
#define I_MIO_C   2   // prima entrata modulo IO/condivisa
#define I_UTN_C   256 // prima entrata utente/condivisa (inizio M2)
#define I_UTN_P   384 // prima entrata utente/privata

#define N_SIS_C   1   // numero entrate sistema/condivisa
#define N_SIS_P   1   // numero entrate sistema/privata
#define N_MIO_C   1   // numero entrate modulo IO/condivisa
#define N_UTN_C   128 // numero entrate utente/convidisa
#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 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).

1.4 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:

```
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);
}
```

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

```
void clear_root_tab(paddr dest) {
  // eliminiamo le entrate create da init_root_tab()
  set_des(dest, I_SIS_C, N_SIS_C, 0);
  set_des(dest, I_MIO_C, N_MIO_C, 0);
  set_des(dest, I_UTN_C, N_UTN_C, 0);
}
```

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()`:

```
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(); });

  // caso di errore
  if (v != bottom) {
    unmap(root_tab, bottom - size, v,
      [](vaddr, paddr p, int) { rilascia_frame(p); });
    return false;
  }
  return true;
}
```

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

Questa ha ancora una 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); });
}
```

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:

```
// creazione della pila sistema
static_assert(DIM_SYS_STACK > 0 && (DIM_SYS_STACK & 0xFFF) == 0);

// siamo in un altro processo, quindi dobbiamo accedere alla pila sistema tramite la finestra FM
pila_sistema = trasforma(p->cr3, fin_sis_p - 1) + 1;

// convertiamo a puntatore a natq, per accedervi pi  comodamente
pl = ptr_cast<natq>(pila_sistema);

if (liv == LIV_UTENTE) {
  // processo di livello utente
  // inizializzazione della pila sistema
  pl[-5] = int_cast<natq>(f);          // RIP (codice utente)
  pl[-4] = SEL_CODICE_UTENTE;          // CS (codice utente)
  pl[-3] = BIT_IF;                     // RFLAGS
  pl[-2] = fin_utn_p - sizeof(natq);   // RSP
  pl[-1] = SEL_DATI_UTENTE;            // SS (pila utente)

  // eseguendo una IRET da questa situazione, il processo
  // passer  ad eseguire la prima istruzione della funzione f,
  // usando come pila la pila utente (al suo indirizzo virtuale)

  // creazione della pila utente
  static_assert(DIM_USR_STACK > 0 && (DIM_USR_STACK & 0xFFF) == 0);

  // inizialmente, il processo si trova a livello sistema, come
  // se avesse eseguito una istruzione INT, con la pila sistema
  // che contiene le 5 parole quadruple preparate precedentemente
  p->contesto[I_RSP] = fin_sis_p - 5 * sizeof(natq);

  p->livello = LIV_UTENTE;

  // dal momento che usiamo traduzioni diverse per le parti sistema/private
  // di tutti i processi, possiamo inizializzare p->punt_nucleo con un
  // indirizzo (virtuale) uguale per tutti i processi
  p->punt_nucleo = fin_sis_p;

  // tutti gli altri campi valgono 0
} else {
  // processo di livello sistema
  // inizializzazione della pila sistema
  pl[-6] = int_cast<natq>(f);          // RIP (codice sistema)
  pl[-5] = SEL_CODICE_SISTEMA;         // CS (codice sistema)
  pl[-4] = BIT_IF;                     // RFLAGS
  pl[-3] = fin_sis_p - sizeof(natq);   // RSP
  pl[-2] = 0;                          // SS
  pl[-1] = 0;                          // ind. rit.

  // inizializzazione del descrittore di processo
  p->contesto[I_RSP] = fin_sis_p - 6 * sizeof(natq);
```

```
51
52    p->livello = LIV_SISTEMA;
53
54    // tutti gli altri campi valgono 0
55  }
```