```
// sistema.cpp
//
#include <costanti.h>
#include <tipo.h>
#include <libce.h>
#include <sys.h>
#include <sysio.h>

//////////////////////////////////////////////////////////////////////////
//                      PROCESSI                                        //
//////////////////////////////////////////////////////////////////////////
const natl DUMMY_PRIORITY = 0;
const int N_REG = 16;   // numero di registri nel campo contesto

// descrittore di processo
struct des_proc {
        natw id;
        natw livello;
        natl precedenza;
        vaddr punt_nucleo;
        natq contesto[N_REG];
        paddr cr3;

        des_proc *puntatore;
};

des_proc *proc_table[MAX_PROC];

// numero di processi utente attivi
volatile natl processi;
// distrugge il processo puntato da esecuzione.
// Se dump == true invia sul log lo stato al momento della chiamata.
extern "C" void c_abort_p(bool selfdump = true);
// ferma tutto il sistema (in caso di bug nel sistema stesso)
extern "C" void panic(const char *msg);

//indici nell'array contesto
enum { I_RAX, I_RCX, I_RDX, I_RBX,
        I_RSP, I_RBP, I_RSI, I_RDI, I_R8, I_R9, I_R10,
        I_R11, I_R12, I_R13, I_R14, I_R15 };

des_proc *esecuzione;
des_proc *pronti;

void inserimento_lista(des_proc *&p_lista, des_proc *p_elem)
{
// inserimento in una lista semplice ordinata
//   (tecnica dei due puntatori)
        des_proc *pp, *prevp;

        pp = p_lista;
        prevp = nullptr;
        while (pp && pp->precedenza >= p_elem->precedenza) {
                prevp = pp;
                pp = pp->puntatore;
        }

        p_elem->puntatore = pp;

        if (prevp)
                prevp->puntatore = p_elem;
        else
                p_lista = p_elem;

}

// rimuove da p_lista il processo a piÃ¹ altra prioritÃ  e
// lo restituisce
des_proc *rimozione_lista(des_proc *&p_lista)
{
// estrazione dalla testa
        des_proc *p_elem = p_lista;     // nullptr se la lista Ã¨ vuota

        if (p_lista)
                p_lista = p_lista->puntatore;

        if (p_elem)
                p_elem->puntatore = nullptr;

        return p_elem;
}

// inserisce esecuzione in testa alla lista pronti
extern "C" void inspronti()
{
        esecuzione->puntatore = pronti;
        pronti = esecuzione;
}

// sceglie il prossimo processo da mettere in esecuzione
extern "C" void schedulatore(void)
{
```

```
// poichÃ© la lista Ã¨ giÃ  ordinata in base alla prioritÃ ,
// Ã¨ sufficiente estrarre l'elemento in testa
        esecuzione = rimozione_lista(pronti);
}

// dato un id, restiusce il puntatore al corrispondente des_proc
extern "C" des_proc *des_p(natl id)
{
        if (id > MAX_PROC_ID)
                panic("id non valido");

        return proc_table[id];
}

// alloca un id non utilizzato.
natl alloca_proc_id(des_proc *p)
{
        static natl next = 0;

        // La funzione inizia la ricerca partendo  dall'id successivo
        // all'ultimo restituito (salvato nella variable statica 'next'),
        // saltando quelli che risultano in uso.
        natl scan = next, found = 0xFFFFFFFF;
        do {
                if (proc_table[scan] == nullptr) {
                        found = scan;
                        proc_table[found] = p;
                }
                scan = (scan + 1) % MAX_PROC;
        } while (found == 0xFFFFFFFF && scan != next);
        next = scan;
        return found;
}

void rilascia_proc_id(natw id)
{
        if (id > MAX_PROC_ID)
                panic("id non valido");

        proc_table[id] = nullptr;
}

// si veda anche CREAZIONE E DISTRUZIONE DEI PROCESSI, piÃ¹ avanti

////////////////////////////////////////////////////////////////////////////
//                      SEMAFORI                                          //
////////////////////////////////////////////////////////////////////////////

// descrittore di semaforo
struct des_sem {
        int counter;
        des_proc *pointer;
};

// Usiamo due insiemi separati di semafori, uno per il livello utente e un
// altro per il livello sistema. I primi MAX_SEM semafori di array_dess sono
// per il livello utente, gli altri MAX_SEM sono il livello sistema.
des_sem array_dess[MAX_SEM * 2];

// restituisce il livello a cui si trovava il processore al momento
// in cui Ã¨ stata invocata la primitiva. Attenzione: funziona solo
// se Ã¨ stata chiamata salva_stato.
int liv_chiamante()
{
        // salva_stato ha salvato il puntatore alla pila sistema
        // subito dopo l'invocazione della INT
        natq *pila = reinterpret_cast<natq*>(esecuzione->contesto[I_RSP]);
        // la seconda parola dalla cima della pila contiene il livello
        // di privilegio che aveva il processore prima della INT
        return pila[1] == SEL_CODICE_SISTEMA ? LIV_SISTEMA : LIV_UTENTE;
}

// I semafori non vengono mai deallocati, quindi Ã¨ possibile allocarli
// sequenzialmente. Per far questo, Ã¨ sufficiente ricordare quanti ne
// abbiamo allocati
natl sem_allocati_utente = 0;
natl sem_allocati_sistema = 0;
natl alloca_sem()
{
        int liv = liv_chiamante();
        natl i;
        if (liv == LIV_UTENTE) {
                if (sem_allocati_utente >= MAX_SEM)
                        return 0xFFFFFFFF;
                i = sem_allocati_utente;
                sem_allocati_utente++;
        } else {
                if (sem_allocati_sistema >= MAX_SEM)
                        return 0xFFFFFFFF;
                i = sem_allocati_sistema + MAX_SEM;
                sem_allocati_sistema++;
        }
```

```
        return i;
}

// dal momento che i semafori non vengono mai deallocati,
// un semaforo Ã¨ valido se e solo se il suo indice Ã¨ inferiore
// al numero dei semafori allocati
bool sem_valido(natl sem)
{
        int liv = liv_chiamante();
        return sem < sem_allocati_utente ||
                (liv == LIV_SISTEMA && sem - MAX_SEM < sem_allocati_sistema);
}

// parte "C++" della primitiva sem_ini
extern "C" void c_sem_ini(int val)
{
        natl i = alloca_sem();

        if (i != 0xFFFFFFFF)
                array_dess[i].counter = val;

        esecuzione->contesto[I_RAX] = i;
}


extern "C" void c_sem_wait(natl sem)
{
        // una primitiva non deve mai fidarsi dei parametri
        if (!sem_valido(sem)) {
                flog(LOG_WARN, "semaforo errato: %d", sem);
                c_abort_p();
                return;
        }

        des_sem *s = &array_dess[sem];
        s->counter--;

        if (s->counter < 0) {
                inserimento_lista(s->pointer, esecuzione);
                schedulatore();
        }
}

extern "C" void c_sem_signal(natl sem)
{
        // una primitiva non deve mai fidarsi dei parametri
        if (!sem_valido(sem)) {
                flog(LOG_WARN, "semaforo errato: %d", sem);
                c_abort_p();
                return;
        }

        des_sem *s = &array_dess[sem];
        s->counter++;

        if (s->counter <= 0) {
                des_proc* lavoro = rimozione_lista(s->pointer);
                inspronti();    // preemption
                inserimento_lista(pronti, lavoro);
                schedulatore(); // preemption
        }
}

////////////////////////////////////////////////////////////////////////////////
//                          MEMORIA DINAMICA                                  //
////////////////////////////////////////////////////////////////////////////////

void* operator new(size_t size)
{
        return alloca(size);
}

void* operator new(size_t size, align_val_t align)
{
        return alloc_aligned(size, align);
}

void operator delete(void *p)
{
        dealloca(p);
}

////////////////////////////////////////////////////////////////////////////////
//                             TIMER                                          //
////////////////////////////////////////////////////////////////////////////////

// richiesta al timer
struct richiesta {
        natl d_attesa;
        richiesta *p_rich;
        des_proc *pp;
};
```

```
richiesta *p_sospesi;

void inserimento_lista_attesa(richiesta *p);
// parte "C++" della primitiva delay
extern "C" void c_delay(natl n)
{
        richiesta *p;

        p = new richiesta;
        p->d_attesa = n;
        p->pp = esecuzione;

        inserimento_lista_attesa(p);
        schedulatore();
}

// inserisce P nella coda delle richieste al timer
void inserimento_lista_attesa(richiesta *p)
{
        richiesta *r, *precedente;
        bool ins;

        r = p_sospesi;
        precedente = nullptr;
        ins = false;

        while (r != nullptr && !ins)
                if (p->d_attesa > r->d_attesa) {
                        p->d_attesa -= r->d_attesa;
                        precedente = r;
                        r = r->p_rich;
                } else {
                        ins = true;
                }

        p->p_rich = r;
        if (precedente != nullptr)
                precedente->p_rich = p;
        else
                p_sospesi = p;

        if (r != nullptr)
                r->d_attesa -= p->d_attesa;
}

// driver del timer
extern "C" void c_driver_td(void)
{
        richiesta *p;

        inspronti();

        if (p_sospesi != nullptr) {
                p_sospesi->d_attesa--;
        }

        while (p_sospesi != nullptr && p_sospesi->d_attesa == 0) {
                inserimento_lista(pronti, p_sospesi->pp);
                p = p_sospesi;
                p_sospesi = p_sospesi->p_rich;
                delete p;
        }

        schedulatore();
}


////////////////////////////////////////////////////////////////////////////
//                          ECCEZIONI                                     //
////////////////////////////////////////////////////////////////////////////

static const char *eccezioni[] = {
        "errore di divisione",          // 0
        "debug",                        // 1
        "interrupt non mascherabile",   // 2
        "breakpoint",                   // 3
        "overflow",                     // 4
        "bound check",                  // 5
        "codice operativo non valido",  // 6
        "dispositivo non disponibile",  // 7
        "doppio fault",                 // 8
        "coprocessor segment overrun",  // 9
        "TSS non valido",               // 10
        "segmento non presente",        // 11
        "errore sul segmento stack",    // 12
        "errore di protezione",         // 13
        "page fault",                   // 14
        "riservato",                    // 15
        "errore su virgola mobile",     // 16
        "errore di allineamento",       // 17
        "errore interno",               // 18
```

```
        "errore SIMD",                  // 19
        "errore di virtualizzazione",   // 20
        "riservato",                    // 21
        "riservato",                    // 22
        "riservato",                    // 23
        "riservato",                    // 24
        "riservato",                    // 25
        "riservato",                    // 26
        "riservato",                    // 27
        "riservato",                    // 28
        "riservato",                    // 29
        "eccezione di sicurezza",       // 30
        "riservato"                     // 31
};


//  il microprogramma di gestione delle eccezioni di page fault lascia in cima
//  alla pila (oltre ai valori consueti) una parola quadrupla i cui 4 bit meno
//  significativi specificano piÃ¹ precisamente il motivo per cui si Ã¨
//  verificato un page fault. Il significato dei bit Ã¨ il seguente:
//  - prot: se questo bit vale 1, il page fault si Ã¨ verificato per un errore
//  di protezione: il processore si trovava a livello utente e la pagina
//  era di livello sistema (bit US = 0 in una qualunque delle tabelle
//  dell'albero che porta al descrittore della pagina). Se prot = 0, la pagina
//  o una delle tabelle erano assenti (bit P = 0)
//  - write: l'accesso che ha causato il page fault era in scrittura (non
//  implica che la pagina non fosse scrivibile)
//  - user: l'accesso Ã¨ avvenuto mentre il processore si trovava a livello
//  utente (non implica che la pagina fosse invece di livello sistema)
//  - res: uno dei bit riservati nel descrittore di pagina o di tabella non
//  avevano il valore richiesto (il bit D deve essere 0 per i descrittori di
//  tabella, e il bit pgsz deve essere 0 per i descrittori di pagina)
static const natq PF_PROT  = 1U << 0;
static const natq PF_WRITE = 1U << 1;
static const natq PF_USER  = 1U << 2;
static const natq PF_RES   = 1U << 3;

extern "C" vaddr readCR2();
extern "C" natq end;    // ultimo indirizzo del codice sistema (fornito dal collegatore)

// gestore generico di eccezioni (chiamata da tutti i gestori di eccezioni in
// sistema.s, tranne il gestore di page fault e di non-maskable-interrupt)
void process_dump(des_proc*, log_sev sev);
extern "C" void gestore_eccezioni(int tipo, natq errore, vaddr rip)
{
        flog(LOG_WARN, "Eccezione %d (%s), errore %x, rip %lx", tipo, eccezioni[tipo], errore, rip);
        if (tipo == 14) {
                // page fault: diamo piÃ¹ informazioni
                vaddr v = readCR2();
                flog(LOG_WARN, "  indirizzo virtuale: %p %s", v,
                        (v < DIM_PAGINA) ? "(probabile puntatore NULL)" : "");
                flog(LOG_WARN, "  dettagli: %s, %s, %s, %s",
                        (errore & PF_PROT)  ? "protezione"      : "pag o tab assente",
                        (errore & PF_WRITE) ? "scrittura"       : "lettura",
                        (errore & PF_USER)  ? "da utente"       : "da sistema",
                        (errore & PF_RES)   ? "bit riservato"   : "");
                // il sistema non Ã¨ progettato per gestire page fault causati
                // dalle primitie di nucleo. Se ciÃ² si Ã¨ verificato,
                // si tratta di un bug
                if ( (!(errore & PF_USER) && rip < (vaddr)&end) || (errore & PF_RES) ) {
                        panic("ERRORE DI SISTEMA");
                }
        }
        // inviamo sul log lo stato al momento del sollevamento dell'eccezione
        // (come in questo momento si trova salvato in pila sistema e
        // descrittore di processo)
        process_dump(esecuzione, LOG_WARN);
        // abortiamo il processo (senza mostrare nuovamente il dump)
        c_abort_p(false);
}
////////////////////////////////////////////////////////////////////////////////
//                          FRAME                                             //
////////////////////////////////////////////////////////////////////////////////

struct des_frame {
        union {
                // numero di entrate valide (se il frame contiene una tabella)
                natw nvalide;
                // lista di frame liberi (se il frame Ã¨ libero)
                natl prossimo_libero;
        };
};

// numero totale di frame (M1 + M2)
natq const N_FRAME = MEM_TOT / DIM_PAGINA;
// numero totale di frame in M1 e in M2
natq N_M1;
natq N_M2;

// array dei descrittori di frame
des_frame vdf[N_FRAME];
// testa della lista dei frame liberi
```

```
natq primo_frame_libero;
// numero di frame nella lista dei frame liberi
natq num_frame_liberi;

// init_des_frame viene chiamata in fase di inizializzazione.  Tutta la memoria
// non ancora occupata viene usata per i frame.  La funzione si preoccupa anche
// di allocare lo spazio per i descrittori di frame e di inizializzarli in modo
// che tutti frame risultino liberi &end Ã¨ l'indirizzo del primo byte non
// occupato dal modulo sistema (Ã¨ calcolato dal collegatore).
void init_frame()
{
        // primo frame di M2
        paddr fine_M1 = allinea(reinterpret_cast<paddr>(&end), DIM_PAGINA);
        // numero di frame in M1 e indice di f in vdf
        N_M1 = fine_M1 / DIM_PAGINA;
        // numero di frame in M2
        N_M2 = N_FRAME - N_M1;

        if (!N_M2)
                return;

        // creiamo la lista dei frame liberi, che inizialmente contiene tutti i
        // frame di M2
        primo_frame_libero = N_M1;
#ifndef N_STEP
        // alcuni esercizi definiscono N_STEP == 2 per creare mapping non
        // contigui in memoria virtale e controllare meglio alcuni possibili
        // bug
#define N_STEP 1
#endif
        for (natq i = N_M1; i < N_FRAME - N_STEP; i++) {
                vdf[i].prossimo_libero = i + N_STEP;
                num_frame_liberi++;
        }
        vdf[N_FRAME - N_STEP].prossimo_libero = 0;
}

// estrea un frame libero dalla lista, se non vuota
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;
}

// rende di nuovo libera il frame descritto da df
void rilascia_frame(paddr f)
{
        natq j = f / DIM_PAGINA;
        if (j < N_M1) {
                panic("tentativo di rilasciare un frame di M1");
        }
        vdf[j].prossimo_libero = primo_frame_libero;
        primo_frame_libero = j;
        num_frame_liberi++;
}

// gestione del contatore di entrate valide (per i frame che contengono
// tabelle)
void inc_ref(paddr f)
{
        vdf[f / DIM_PAGINA].nvalide++;
}

void dec_ref(paddr f)
{
        vdf[f / DIM_PAGINA].nvalide--;
}

natl get_ref(paddr f)
{
        return vdf[f / DIM_PAGINA].nvalide;
}

////////////////////////////////////////////////////////////////////////////
//                          PAGINAZIONE                                   //
////////////////////////////////////////////////////////////////////////////
#include <vm.h>

// indirizzo virtuale di partenza delle varie zone della memoria
// virtuale dei proceii

const vaddr ini_sis_c = norm(I_SIS_C * dim_region(MAX_LIV - 1)); // sistema condivisa
const vaddr ini_sis_p = norm(I_SIS_P * dim_region(MAX_LIV - 1)); // sistema privata
const vaddr ini_mio_c = norm(I_MIO_C * dim_region(MAX_LIV - 1)); // modulo IO
const vaddr ini_utn_c = norm(I_UTN_C * dim_region(MAX_LIV - 1)); // utente condivisa
```

```
const vaddr ini_utn_p = norm(I_UTN_P * dim_region(MAX_LIV - 1)); // utente privata

// indirizzo del primo byte che non appartiene alla zona specificata
const vaddr fin_sis_c = ini_sis_c + dim_region(MAX_LIV - 1) * N_SIS_C;
const vaddr fin_sis_p = ini_sis_p + dim_region(MAX_LIV - 1) * N_SIS_P;
const vaddr fin_mio_c = ini_mio_c + dim_region(MAX_LIV - 1) * N_MIO_C;
const vaddr fin_utn_c = ini_utn_c + dim_region(MAX_LIV - 1) * N_UTN_C;
const vaddr fin_utn_p = ini_utn_p + dim_region(MAX_LIV - 1) * N_UTN_P;

// alloca un frame libero destinato a contenere una tabella
paddr alloca_tab()
{
        paddr f = alloca_frame();
        if (f) {
                memset(reinterpret_cast<void*>(f), 0, DIM_PAGINA);
        }
        vdf[f / DIM_PAGINA].nvalide = 0;
        return f;
}

// dealloca un frame che contiene una tabella, controllando che non contenga
// entrate valide
void rilascia_tab(paddr f)
{
        if (int n = get_ref(f)) {
                flog(LOG_ERR, "tentativo di deallocare la tabella %x con %d entrate valide", f, n);
                panic("errore interno");
        }
        rilascia_frame(f);
}

// setta l'entrata j-esima della tabella 'tab' con il valore 'se'.
// Si preoccupa di aggiustare opporutinamente il contatore delle
// entrate valide.
void set_entry(paddr tab, natl j, tab_entry se)
{
        tab_entry& de = get_entry(tab, j);
        if ((se & BIT_P) && !(de & BIT_P)) {
                inc_ref(tab);
        } else if (!(se & BIT_P) && (de & BIT_P)) {
                dec_ref(tab);
        }
        de = se;
}

// copia 'n' descrittori a partire da quello di indice 'i' dalla
// tabella di indirizzo 'src' in quella di indirizzo 'dst'
void copy_des(paddr src, paddr dst, natl i, natl n)
{
        for (natl j = i; j < i + n && j < 512; j++) {
                tab_entry se = get_entry(src, j);
                set_entry(dst, j, se);
        }
}

// setta 'n' descrittori a partire da quello di indice 'i' nella
// tabella di indirizzo 'dst' con valore 'e'
void set_des(paddr dst, natl i, natl n, tab_entry e)
{
        for (natl j = i; j < i + n && j < 512; j++) {
                set_entry(dst, j, e);
        }
}

// crea tutto il sottoalbero, con radice tab, necessario a tradurre tutti gli
// indirizzi dell'intervallo [begin, end). L'intero intervallo non deve
// contenere traduzioni pre-esistenti.
//
// I bit RW e US che sono a 1 nel parametro flags saranno settati anche in
// tutti i descrittori del sottoalbero. Se flags contiene i bit PWT e/o PWT,
// questi saranno settati solo sui descrittori foglia.
//
// Il tipo getpaddr deve poter essere invocato come 'getpaddr(v)', dove 'v' Ã¨
// un indirizzo virtuale. L'invocazione deve restituire l'indirizzo fisico che
// si vuole far corrispondere a 'v'.
//
// La funzione, per default, crea traduzioni con pagine di livello 1. Se si
// vogliono usare pagine di livello superiore (da 2 a MAX_PS_LVL) occorre
// passare anche il parametro ps_lvl.
//
// La funzione restituisce il primo indirizzo non mappato, che in caso di
// sucesso Ã¨ end. Un valore diverso da end segnala che si Ã¨ verificato
// un problema durante l'operazione (per esempio: memoria esaurita, indirizzo
// giÃ  mappato).
template<typename T>
vaddr map(paddr tab, vaddr begin, vaddr end, natl flags, T getpaddr, int ps_lvl = 1)
{
        // il mapping coinvolge sempre almeno pagine, quindi consideriamo come
        // erronei dei parametri beg, end che non sono allineati alle pagine.
        natq dr = dim_region(ps_lvl - 1);
        if (begin & (dr - 1)) {
                flog(LOG_ERR, "begin=%p non allineato alle pagine di livello %d",
```

```
                        begin, ps_lvl);
                panic("chiamata di map() non valida");
        }
        if (end & (dr - 1)) {
                flog(LOG_ERR, "end=%p non allineato alle pagine di livello %d",
                        end, ps_lvl);
                panic("chiamata di map() non valida");
        }
        if (flags & ~(BIT_RW|BIT_US|BIT_PWT|BIT_PCD)) {
                panic("flags contiene bit non validi (ammessi RW, US, PWT e PCD)");
        }
        if (ps_lvl < 1 || ps_lvl > MAX_PS_LVL) {
                flog(LOG_ERR, "ps_lvl %d non ammesso (deve essere compreso tra 2 e %d)",
                        ps_lvl, MAX_PS_LVL);
                panic("chiamata di map() non valida");
        }

        // usiamo un tab_iter per percorrere tutto il sottoalbero.  Si noti che
        // il sottoalbero verrÃ  costruito man mano che lo visitiamo.
        //
        // Si noti che tab_iter fa ulteriori controlli sulla validitÃ
        // dell'intervallo (si veda tab_iter::valid_interval in libce)
        for (tab_iter it(tab, begin, end - begin); it; it.next()) {
                tab_entry& e = it.get_e();
                int l = it.get_l();
                // new_f diventerÃ  diverso da 0 se dobbiamo settare a 1 il bit
                // P di 'e'
                paddr new_f = 0;

                if (l > ps_lvl) {
                        // per tutti i livelli non "foglia" allochiamo la
                        // tabella di livello inferiore e facciamo in modo che
                        // il descrittore corrente la punti (Si noti che la
                        // tabella potrebbe esistere giÃ , e in quel caso non
                        // facciamo niente)
                        if (!(e & BIT_P)) {
                                new_f = alloca_tab();
                                if (!new_f)
                                        return it.get_v();
                        } else if (e & BIT_PS) {
                                vaddr v = it.get_v();
                                flog(LOG_WARN, "indirizzo %p, livello %d, gia' mappato", v, l);
                                return v;
                        }
                } else {
                        // arrivati ai descrittori di livello ps_lvl creiamo la
                        // traduzione vera e propria.

                        // otteniamo l'indirizzo 'v' la cui traduzione passa
                        // dal descrittore corrente
                        vaddr v = it.get_v();
                        if (e & BIT_P) {
                                flog(LOG_WARN, "indirizzo %p, livello %d,  gia' mappato", v, l);
                                return v;
                        }
                        // otteniamo il corrispondente indirizzo fisico
                        // chiedendolo all'oggetto getpaddr
                        new_f = getpaddr(v);
                        if (!new_f)
                                return v;
                }
                if (new_f) {
                        // 'e' non puntava a niente e ora deve puntare a new_f
                        set_IND_FISICO(e, new_f);
                        e |= BIT_P;
                        // se stiamo creando una traduzione per una pagina di livello
                        // maggiore di 1 dobbiamo settare il bit PS
                        if (l > 1 && l == ps_lvl)
                                e |= BIT_PS;
                        // ricordiamoci di incrementare il contatore delle entrate
                        // valide della tabella a cui 'e' appartiene
                        inc_ref(it.get_tab());
                }

                // se sono stati richiesti i bit RW e/o US, questi vanno
                // settati su tutti i livelli, altrimenti non hanno effetto.
                e |= (flags & (BIT_RW|BIT_US));

                // i flag PWT e PCD vanno settati solo sui descrittori foglia
                if (it.is_leaf()) {
                        e |= (flags & (BIT_PWT|BIT_PCD));
                }
        }
        return end;
}

// elimina tutte le traduzioni nell'intervallo [begin, end).  Rilascia
// automaticamente tutte le sottotabelle che diventano vuote dopo aver
// eliminato le traduzioni. Per liberare le pagine vere e proprie, invece,
// chiama la funzione putpaddr() passandole l'indirizzo fisico e il livello
// della pagina da elinimare.
template<typename T>
```

```
void unmap(paddr tab, vaddr begin, vaddr end, T putpaddr)
{
        tab_iter it(tab, begin, end - begin);
        // eseguiamo una visita in ordine posticipato
        for (it.post(); it; it.next_post()) {
                tab_entry& e = it.get_e();

                if (!(e & BIT_P))
                        continue;

                paddr p = extr_IND_FISICO(e);
                if (!it.is_leaf()) {
                        // l'entrata punta a una tabella.
                        if (!get_ref(p)) {
                                // Se la tabella non contiene piÃ¹ 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())
                        // lasciamo al chiamante decidere cosa fare
                        // con l'indirizzo fisico puntato da 'e'
                        putpaddr(p, it.get_l());
                }

                // per tutte le pagine, e per le tabelle che abbiamo
                // deallocato, azzeriamo l'entrata 'e' e decrementiamo il
                // contatore delle entrate valide nella tabella che la contiene
                e = 0;
                dec_ref(it.get_tab());
        }
}

// primitiva utilizzata dal modulo I/O per controllare che i buffer passati dal
// livello utente siano accessibili dal livello utente (problema del Cavallo di
// Troia) e non possano causare page fault nel modulo I/O (bit P tutti a 1 e
// scrittura permessa quando necessario)
extern "C" bool c_access(vaddr begin, natq dim, bool writeable)
{
        if (!tab_iter::valid_interval(begin, dim))
                return false;

        // usiamo un tab_iter per percorrere tutto il sottoalbero relativo
        // alla traduzione degli indirizzi nell'intervallo [begin, begin+dim).
        for (tab_iter it(esecuzione->cr3, begin, dim); it; it.next()) {
                tab_entry e = it.get_e();

                // interrompiamo il ciclo non appena troviamo qualcosa che non va
                if (!(e & BIT_P) || !(e & BIT_US) || (writeable && !(e & BIT_RW)))
                        return false;
        }
        return true;
}

// mappa la memoria fisica in memoria virtuale, inclusa l'area PCI
bool crea_finestra_FM(paddr root_tab)
{
        auto identity_map = [] (vaddr v) -> paddr { return v; };
        // mappiamo tutta la memoria fisica:
        // - a livello sistema (bit U/S non settato)
        // - scrivibile (bit R/W settato)
        // - con pagine di grandi dimensioni (bit PS)
        //   (usiamo pagine di livello 2 che sono sicuramente disponibili)

        // vogliamo saltare la prima pagina per intercettare *NULL, e inoltre
        // vogliamo settare per il bit PWT per la pagina che contiene la memoria
        // video.  Per farlo dobbiamo rinunciare a settare PS per la prima regione
        natq first_reg = dim_region(1);
        if (map(root_tab, DIM_PAGINA, first_reg, BIT_RW, identity_map) != first_reg)
                return false;

        // settiamo il bit PWT per la pagina che contiene la memoria video.
        // Usiamo un tab_iter su una sola pagina, fermandoci sul descrittore
        // "foglia" che contiene la traduzione.
        tab_iter it(root_tab, 0xb8000, DIM_PAGINA);
        while (it.down())
                ;
        tab_entry& e = it.get_e();
        e |= BIT_PWT;

        // mappiamo il resto della memoria 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:  [%p, %p)", DIM_PAGINA, MEM_TOT);
```

```
        // Mappiamo gli ultimi 20MiB prima di 4GiB settando sia PWT che PCD.
        // Questa zona di indirizzi Ã¨ utilizzata dall'APIC per mappare i propri registri.
        vaddr start_pci = 4*GiB - 20*MiB,
              end_pci = 4*GiB;
        if (map(root_tab, start_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:    [%p, %p)", start_pci, end_pci);
        return true;
}

// restituisce l'indirizzo fisico che corrisponde a ind_virt nell'albero
// di traduzione con radice root_tab.
paddr trasforma(paddr root_tab, vaddr v)
{
        // usiamo un tab_iter su una sola pagina fermandoci sul
        // descrittore foglia lungo il percorso di traduzione di 'v'
        tab_iter it(root_tab, v);
        while (it.down())
                ;

        // si noti che il percorso potrebbe essere incompleto.
        // Ce ne accorgiamo perchÃ© il descrittore foglia ha il bit P a
        // zero. In quel caso restituiamo 0, che per noi non Ã¨ un
        // indirizzo fisico valido.
        tab_entry e = it.get_e();
        if (!(e & BIT_P))
                return 0;

        // se il percorso Ã¨ completo calcoliamo la traduzione corrispondente.
        // Si noti che non siamo necessariamente arrivati al livello 1, se
        // c'era un bit PS settato lungo il percorso.
        int l = it.get_l();
        natq mask = dim_region(l - 1) - 1;
        return (e & ~mask) | (v & mask);
}

// restituisce l'indirizzo fisico che corrisponde a ind_virt nello
// spazio di indirizzamento del processo corrente, o zero
// se la traduzione non esiste.
extern "C" paddr c_trasforma(vaddr ind_virt)
{
        return trasforma(esecuzione->cr3, ind_virt);
}

// )

////////////////////////////////////////////////////////////////////////////
//                     CREAZIONE E DISTRUZIONE DEI PROCESSI                //
////////////////////////////////////////////////////////////////////////////
const natl BIT_IF = 1L << 9;

// inizializza la tabella radice di un nuovo processo
void init_root_tab(paddr dest)
{
        paddr pdir = readCR3();

        // ci limitiamo a copiare dalla tabella radice corrente i puntatori
        // alle tabelle di livello inferiore per tutte le parti condivise
        // (sistema, utente e I/O). Quindi tutti i sottoalberi di traduzione
        // delle parti condivise saranno anch'essi condivisi. Questo, oltre a
        // semplificare l'inizializzazione di un processo, ci permette di
        // risparmiare un po' di memoria.
        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);
}

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);
}

// crea una pila processo (utente o sistema, in base a 'liv').  Creiamo una
// traduzione dagli indirizzi riservati alla pila verso frame allocati sul
// momento.
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 p, int) { rilascia_frame(p); });
                return false;
        }
        return true;
```

```
}

// distrugge una pila (non ha importanza se utente o sistema)
void distruggi_pila(paddr root_tab, vaddr bottom, natq size)
{
        unmap(root_tab, bottom - size, bottom, [](vaddr p, int) { rilascia_frame(p); });
}

// alloca un id per il processo e crea e inizializza il des_proc, la
// pila sistema e, per i processi di livello utente, la pila utente. Crea
// l'albero di traduzione completo per la memoria virtuale del processo.
des_proc* crea_processo(void f(natq), natq a, int prio, char liv, bool IF)
{
        des_proc*       p;                      // des_proc per il nuovo processo
        paddr           pila_sistema;           // pila_sistema del processo
        natq*           pl;                     // pila sistema come array di natq

        // allocazione (e azzeramento preventivo) di un des_proc
        p = new des_proc;
        if (!p)
                goto errore1;
        memset(p, 0, sizeof(des_proc));

        // rimpiamo i campi di cui conosciamo giÃ  i valori
        p->precedenza = prio;
        p->puntatore = nullptr;
        // il registro RDI deve contenere il parametro da passare alla
        // funzione f
        p->contesto[I_RDI] = a;

        // selezione di un identificatore
        p->id = alloca_proc_id(p);
        if (p->id == 0xFFFFFFFF)
                goto errore2;

        // creazione della tabella radice del processo
        p->cr3 = alloca_tab();
        if (p->cr3 == 0)
                goto errore3;
        init_root_tab(p->cr3);

        // creazione della pila sistema
        if (!crea_pila(p->cr3, fin_sis_p, DIM_SYS_STACK, LIV_SISTEMA))
                goto errore4;
        // otteniamo un puntatore al fondo della pila appena creata.  Si noti
        // che non possiamo accedervi tramite l'indirizzo virtuale 'fin_sis_p',
        // che verrebbe tradotto seguendo l'albero del processo corrente, e non
        // di quello che stiamo creando.  Per questo motivo usiamo trasforma()
        // per ottenere il corrispondente indirizzo fisico. In questo modo
        // accediamo alla nuova pila tramite la finestra FM.
        pila_sistema = trasforma(p->cr3, fin_sis_p - DIM_PAGINA) + DIM_PAGINA;

        // convertiamo a puntatore a natq, per accedervi piÃ¹ comodamente
        pl = reinterpret_cast<natq*>(pila_sistema);

        if (liv == LIV_UTENTE) {
                // inizializziamo la pila sistema.
                pl[-5] = reinterpret_cast<natq>(f); // RIP (codice utente)
                pl[-4] = SEL_CODICE_UTENTE;         // CS (codice utente)
                pl[-3] = IF ? BIT_IF : 0;           // 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
                // passera' ad eseguire la prima istruzione della funzione f,
                // usando come pila la pila utente (al suo indirizzo virtuale)

                // creazione della pila utente
                if (!crea_pila(p->cr3, fin_utn_p, DIM_USR_STACK, LIV_UTENTE)) {
                        flog(LOG_WARN, "creazione pila utente fallita");
                        goto errore5;
                }

                // inizialmente, il processo si trova a livello sistema, come
                // se avesse eseguito una istruzione INT, con la pila sistema
                // che contiene le 5 parole lunghe 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] = reinterpret_cast<natq>(f);     // RIP (codice sistema)
                pl[-5] = SEL_CODICE_SISTEMA;            // CS (codice sistema)
                pl[-4] = IF ? BIT_IF : 0;               // RFLAGS
                pl[-3] = fin_sis_p - sizeof(natq);      // RSP
```

```
                pl[-2] = 0;                             // SS
                pl[-1] = 0;                             // ind. rit.
                                                        //(non significativo)
                // i processi esterni lavorano esclusivamente a livello
                // sistema. Per questo motivo, prepariamo una sola pila (la
                // pila sistema)

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

                p->livello = LIV_SISTEMA;

                // tutti gli altri campi valgono 0
        }

        return p;

errore5:        distruggi_pila(p->cr3, fin_sis_p, DIM_SYS_STACK);
errore4:        clear_root_tab(p->cr3);
                rilascia_tab(p->cr3);
errore3:        rilascia_proc_id(p->id);
errore2:        delete p;
errore1:        return 0;
}

// distruggi_processo() dealloca tutte le strutture dati allocate da
// crea_processo(), ma bisogna stare attenti a non eliminare la pila sistema,
// se Ã¨ proprio quella che stiamo usando in questo momento.  Questo succede
// durante una terminate_() o abort_p(), quando si tenta di distrugguere
// proprio il processo che ha invocato la primitiva. Per questo motivo, se il
// processo da distruggere Ã¨ proprio quello in esecuzione, eliminamo la sua
// pila sistema solo nella carica_stato, dopo essere passati alla pila
// sistema di un altro processo.
//
// ultimo_terminato: se diverso da zero, contiene l'indirizzo fisico della
// root_tab dell'ultimo processo terminato/abortito. La carica_stato legge
// questo indirizzo per sapere se deve distruggere la pila del processo
// uscente, dopo aver effettuato il passaggio alla pila del processo entrante.
paddr ultimo_terminato;
// chiamata da carica_stato se ultimo_terminato != 0
extern "C" void distruggi_pila_precedente() {
        distruggi_pila(ultimo_terminato, fin_sis_p, DIM_SYS_STACK);
        clear_root_tab(ultimo_terminato);
        rilascia_tab(ultimo_terminato);
        ultimo_terminato = 0;
}
void distruggi_processo(des_proc* p)
{
        paddr root_tab = p->cr3;
        if (p->livello == LIV_UTENTE)
                distruggi_pila(root_tab, fin_utn_p, DIM_USR_STACK);
        ultimo_terminato = root_tab;
        if (p != esecuzione) {
                distruggi_pila_precedente();
        }
        rilascia_proc_id(p->id);
        delete p;
}

// parte "C++" della activate_p
extern "C" void
c_activate_p(void f(natq), natq a, natl prio, natl liv)
{
        des_proc *p;                    // des_proc per il nuovo processo
        natl id = 0xFFFFFFFF;           // id da restituire in caso di fallimento

        // non possiamo accettare una prioritÃ  minore di quella di dummy
        // o maggiore di quella del processo chiamante
        if (prio < MIN_PRIORITY || prio > esecuzione->precedenza) {
                flog(LOG_WARN, "priorita' non valida: %d", prio);
                c_abort_p();
                return;
        }

        // controlliamo che 'liv' contenga un valore ammesso
        // [segnalazione di E. D'Urso]
        if (liv != LIV_UTENTE && liv != LIV_SISTEMA) {
                flog(LOG_WARN, "livello non valido: %d", liv);
                c_abort_p();
                return;
        }

        // non possiamo creare un processo di livello sistema mentre
        // siamo a livello utente
        if (liv == LIV_SISTEMA && liv_chiamante() == LIV_UTENTE) {
                flog(LOG_WARN, "errore di protezione");
                c_abort_p();
                return;
        }

        // accorpiamo le parti comuni tra c_activate_p e c_activate_pe
        // nella funzione ausiliare crea_processo
```

```
        p = crea_processo(f, a, prio, liv, (liv == LIV_UTENTE));

        if (p != nullptr) {
                inserimento_lista(pronti, p);
                processi++;
                id = p->id;                     // id del processo creato
                                                // (allocato da crea_processo)
                flog(LOG_INFO, "proc=%d entry=%p(%d) prio=%d liv=%d", id, f, a, prio, liv);
        }

        esecuzione->contesto[I_RAX] = id;
}


void term_cur_proc(log_sev sev, const char *mode)
{
        des_proc *p = esecuzione;
        flog(sev, "Processo %d %s", p->id, mode);
        distruggi_processo(p);
        processi--;
        schedulatore();
}

// parte "C++" della terminate_p
extern "C" void c_terminate_p()
{
        term_cur_proc(LOG_INFO, "terminato");
}

// come la terminate_p, ma invia anche un warning al log (da invocare quando si
// vuole terminare un processo segnalando che c'Ã¨ stato un errore)
extern "C" void setup_self_dump();
extern "C" void cleanup_self_dump();
extern "C" void c_abort_p(bool selfdump)
{
        if (selfdump) {
                setup_self_dump();
                process_dump(esecuzione, LOG_WARN);
                cleanup_self_dump();
        }
        term_cur_proc(LOG_WARN, "abortito");
}

// Registrazione processi esterni
const natl MAX_IRQ  = 24;
des_proc *a_p[MAX_IRQ];

des_proc* const ESTERN_BUSY = (des_proc*)1;
extern "C" bool load_handler(natq tipo, natq irq);
// associa il processo esterno puntato da "p" all'interrupt "irq".
// Fallisce se un processo esterno era giÃ  stato associato a
// quello stesso interrupt
bool aggiungi_pe(des_proc *p, natw tipo, natb irq)
{
        if (irq >= MAX_IRQ) {
                flog(LOG_WARN, "irq %d non valido (max %d)", irq, MAX_IRQ);
                return false;
        }
        if (a_p[irq]) {
                flog(LOG_WARN, "irq %d occupato", irq);
                return false;
        }
        if (!load_handler(tipo, irq)) {
                flog(LOG_WARN, "tipo %x occupato", tipo);
                return false;
        }

        a_p[irq] = p;
        apic_set_VECT(irq, tipo);
        apic_set_MIRQ(irq, false);
        apic_set_TRGM(irq, false);
        return true;
}

extern "C" void c_activate_pe(void f(natq), natq a, natl prio, natl liv, natb irq)
{
        des_proc        *p;                     // des_proc per il nuovo processo
        natw            tipo;

        if (prio < MIN_EXT_PRIO || prio > MAX_EXT_PRIO) {
                flog(LOG_WARN, "priorita' non valida: %d", prio);
                c_abort_p();
                return;
        }

        p = crea_processo(f, a, prio, liv, true);
        if (p == 0)
                goto error1;

        tipo = prio - MIN_EXT_PRIO;
        if (!aggiungi_pe(p, tipo, irq))
                goto error2;
```

```
        flog(LOG_INFO, "estern=%d entry=%p(%d) prio=%d (tipo=%2x) liv=%d irq=%d",
                        p->id, f, a, prio, tipo, liv, irq);

        esecuzione->contesto[I_RAX] = p->id;
        return;

error2: distruggi_processo(p);
error1: esecuzione->contesto[I_RAX] = 0xFFFFFFFF;
        return;
}

void backtrace(des_proc *p, log_sev sev, const char* msg = "");
void process_dump(des_proc *p, log_sev sev)
{
        natq *pila = reinterpret_cast<natq*>(trasforma(p->cr3, p->contesto[I_RSP]));

        flog(sev, "proc %d, livello %s, precedenza %d", p->id, p->livello == LIV_UTENTE ? "UTENTE" : "SISTEMA", p->precedenza);
        if (pila) {
                flog(sev, "  RIP=%lx CPL=%s", pila[0], pila[1] == SEL_CODICE_UTENTE ? "LIV_UTENTE" : "LIV_SISTEMA");
                natq rflags = pila[2];
                flog(sev, "  RFLAGS=%lx [%s %s %s %s %s %s %s %s %s, IOPL=%s]",
                        rflags,
                        (rflags & 1U << 14) ? "NT" : "--",
                        (rflags & 1U << 11) ? "OF" : "--",
                        (rflags & 1U << 10) ? "DF" : "--",
                        (rflags & 1U << 9)  ? "IF" : "--",
                        (rflags & 1U << 8)  ? "TF" : "--",
                        (rflags & 1U << 7)  ? "SF" : "--",
                        (rflags & 1U << 6)  ? "ZF" : "--",
                        (rflags & 1U << 4)  ? "AF" : "--",
                        (rflags & 1U << 2)  ? "PF" : "--",
                        (rflags & 1U << 0)  ? "CF" : "--",
                        (rflags & 0x3000) == 0x3000 ? "UTENTE" : "SISTEMA");
        } else {
                flog(sev, "  impossibile leggere la pila del processo");
        }
        flog(sev, "  RAX=%lx RBX=%lx RCX=%lx RDX=%lx",
                        p->contesto[I_RAX],
                        p->contesto[I_RBX],
                        p->contesto[I_RCX],
                        p->contesto[I_RDX]);
        flog(sev, "  RDI=%lx RSI=%lx RBP=%lx RSP=%lx",
                        p->contesto[I_RDI],
                        p->contesto[I_RSI],
                        p->contesto[I_RBP],
                        pila ? pila[3] : 0);
        flog(sev, "  R8 =%lx R9 =%lx R10=%lx R11=%lx",
                        p->contesto[I_R8],
                        p->contesto[I_R9],
                        p->contesto[I_R10],
                        p->contesto[I_R11]);
        flog(sev, "  R12=%lx R13=%lx R14=%lx R15=%lx",
                        p->contesto[I_R12],
                        p->contesto[I_R13],
                        p->contesto[I_R14],
                        p->contesto[I_R15]);
        if (pila) {
                flog(sev, "  backtrace:");
                backtrace(p, sev, "  > ");
        }
}

////////////////////////////////////////////////////////////////////////////
//                  INIZIALIZZAZIONE                                      //
////////////////////////////////////////////////////////////////////////////

// riserviamo HEAP_SIZE byte per lo heap di sistema, a partire da HEAP_START
const natq HEAP_START = 1*MiB;
const natq HEAP_SIZE = 1*MiB;

// un primo des_proc, allocato staticamente, da usare durante l'inizializzazione
des_proc init;

// corpo del processo dummy
extern "C" void end_program();
void dummy(natq i)
{
        while (processi)
                ;
        end_program();
}

natl crea_dummy()
{
        des_proc* di = crea_processo(dummy, 0, DUMMY_PRIORITY, LIV_SISTEMA, true);
        if (di == 0) {
                flog(LOG_ERR, "Impossibile creare il processo dummy");
                return 0xFFFFFFFF;
        }
        inserimento_lista(pronti, di);
        return di->id;
```

```
}

void main_sistema(natq n);
natl crea_main_sistema(natq mbi)
{
        des_proc* m = crea_processo(main_sistema, mbi, MAX_PRIORITY, LIV_SISTEMA, false);
        if (m == 0) {
                flog(LOG_ERR, "Impossibile creare il processo main_sistema");
                return 0xFFFFFFFF;
        }
        inserimento_lista(pronti, m);
        processi++;
        return m->id;
}

// ferma il sistema e stampa lo stato di tutti i processi
extern "C" void self_dump();
extern "C" void panic(const char *msg)
{
        static int in_panic = 0;

        if (in_panic) {
                flog(LOG_ERR, "panic ricorsivo. STOP");
                end_program();
        }
        in_panic = 1;

        flog(LOG_ERR, "PANIC: %s", msg);
        flog(LOG_ERR, "  processi: %d", processi);
        flog(LOG_ERR, "------------------------------ PROCESSO IN ESECUZIONE ------------------------------");
        setup_self_dump();
        process_dump(esecuzione, LOG_ERR);
        cleanup_self_dump();
        flog(LOG_ERR, "---------------------------------- ALTRI PROCESSI ----------------------------------");
        for (natl id = 0; id < MAX_PROC; id ++) {
                if (proc_table[id] && proc_table[id] != esecuzione)
                        process_dump(proc_table[id], LOG_ERR);
        }
        end_program();
}

extern "C" void c_io_panic()
{
        panic("errore fatale nel modulo I/O");
}

// se riceviamo un non-maskerable-interrupt, fermiamo il sistema
extern "C" void c_nmi()
{
        panic("INTERRUZIONE FORZATA");
}

// periodo del timer di sistema
const natl DELAY = 59659;

extern "C" void init_gdt();

bool crea_spazio_condiviso(paddr root_tab, paddr mbi);
extern "C" void salta_a_main();
void gdb_breakpoint() { asm volatile ("" :::"memory"); }
extern "C" void main(paddr mbi)
{
        natl mid, dummy_id;

        // anche se il primo processo non Ã¨ completamente inizializzato,
        // gli diamo un identificatore, in modo che compaia nei log
        init.id = 0xFFFF;
        init.precedenza = MAX_PRIORITY;
        init.cr3 = readCR3();
        esecuzione = &init;

        flog(LOG_INFO, "Nucleo di Calcolatori Elettronici, v6.5");
        init_gdt();
        flog(LOG_INFO, "GDT inizializzata");

        apic_init(); // in libce
        apic_reset(); // in libce
        apic_set_VECT(2, TIPO_TIMER);
        flog(LOG_INFO, "APIC inizializzato");

        // iizializziamo la parte M2
        init_frame();
        flog(LOG_INFO, "Numero di frame: %d (M1) %d (M2)", N_M1, N_M2);

        flog(LOG_INFO, "sis/cond [%p, %p)", ini_sis_c, fin_sis_c);
        flog(LOG_INFO, "sis/priv [%p, %p)", ini_sis_p, fin_sis_p);
        flog(LOG_INFO, "io /cond [%p, %p)", ini_mio_c, fin_mio_c);
        flog(LOG_INFO, "usr/cond [%p, %p)", ini_utn_c, fin_utn_c);
        flog(LOG_INFO, "usr/priv [%p, %p)", ini_utn_p, fin_utn_p);

        // creiamo le parti condivise della memoria virtuale di tutti i processi
        // le parti sis/priv e usr/priv verranno create da crea_processo()
```

```
        // ogni volta che si attiva un nuovo processo
        paddr root_tab = alloca_tab();
        if (!root_tab)
                goto error;
        // finestra di memoria, che corrisponde alla parte sis/cond
        if(!crea_finestra_FM(root_tab))
                        goto error;

        gdb_breakpoint();
        // parti io/cond e usr/cond, che contengono i segmenti ELF dei
        // moduli I/O e utente caricati dal boot loader
        if (!crea_spazio_condiviso(root_tab, mbi))
                goto error;
        flog(LOG_INFO, "Create le traduzioni per le parti condivise");
        flog(LOG_INFO, "Frame liberi: %d (M2)", num_frame_liberi);

        loadCR3(root_tab);
        flog(LOG_INFO, "CR3 caricato");


        // Assegna allo heap di sistema HEAP_SIZE byte nel secondo MiB
        heap_init((void*)HEAP_START, HEAP_SIZE);
        flog(LOG_INFO, "Heap di sistema: %x B @%x", HEAP_SIZE, HEAP_START);

        // creazione del processo main_sistema
        mid = crea_main_sistema(mbi);
        if (mid == 0xFFFFFFFF)
                goto error;
        flog(LOG_INFO, "Creato il processo main_sistema (id = %d)", mid);

        // creazione del processo dummy
        dummy_id = crea_dummy();
        if (dummy_id == 0xFFFFFFFF)
                goto error;
        flog(LOG_INFO, "Creato il processo dummy (id = %d)", dummy_id);

        // selezioniamo main_sistema
        schedulatore();

        // esegue CALL carica_stato; IRETQ (vedi "sistema.s"). Il resto
        // dell'inizializzazione prosegue piÃ¹ comodamente nel processo
        // main_sistema(), che puÃ² essere interrotto e puÃ² sospendersi.
        salta_a_main();

error:
        panic("Errore di inizializzazione");
}


void (*io_entry)(natq);
void (*user_entry)(natq);

void main_sistema(natq mbi)
{
        natl sync_io;
        natl id;

        // occupiamo a_p[2] (in modo che non possa essere sovrascritta
        // per errore tramite activate_pe()) e smascheriamo il piedino
        // 2 dell'APIC
        a_p[2] = ESTERN_BUSY;
        apic_set_MIRQ(2, false);
        // attiviamo il timer, in modo che i processi di inizializzazione
        // possano usare anche delay(), se ne hanno bisogno.
        attiva_timer(DELAY);
        flog(LOG_INFO, "Timer attivato (DELAY=%d)", DELAY);

        // inizializzazione del modulo di io
        // Creiamo un processo che esegua la procedura cmain del modulo I/O.
        // Usiamo un semaforo di sincronizzazione per sapere quando
        // l'inizializzazione Ã¨ terminata.
        sync_io = sem_ini(0);
        if (sync_io == 0xFFFFFFFF) {
                flog(LOG_ERR, "Impossibile allocare il semaforo di sincr per l'IO");
                goto error;
        }
        id = activate_p(io_entry, sync_io, MAX_PRIORITY, LIV_SISTEMA);
        if (id == 0xFFFFFFFF) {
                flog(LOG_ERR, "impossibile creare il processo main I/O");
                goto error;
        }
        flog(LOG_INFO, "Creato il processo main I/O (id = %d)", id);
        flog(LOG_INFO, "attendo inizializzazione modulo I/O...");
        sem_wait(sync_io);
        flog(LOG_INFO, "... inizializzazione modulo I/O terminata");

        // creazione del processo start_utente
        id = activate_p(user_entry, 0, MAX_PRIORITY, LIV_UTENTE);
        if (id == 0xFFFFFFFF) {
                flog(LOG_ERR, "impossibile creare il processo main utente");
                goto error;
        }
```

```
        flog(LOG_INFO, "Creato il processo start_utente (id = %d)", id);


        // terminazione
        flog(LOG_INFO, "passo il controllo al processo utente...");
        terminate_p();

error:
        panic("Errore di inizializzazione");
}


// funzioni di supporto per il debugging/testing degli esercizi

#ifdef AUTOCORR
int MAX_LOG = 4;
#else
int MAX_LOG = 5;
#endif

extern "C" void c_log(log_sev sev, const char* buf, natl quanti)
{
        do_log(sev, buf, quanti);
}

extern "C" meminfo c_getmeminfo()
{
        meminfo m;

        // byte liberi nello heap di sistema
        m.heap_libero = disponibile();
        // numero di frame nella lista dei frame liberi
        m.num_frame_liberi = num_frame_liberi;
        // id del processo in esecuzione
        m.pid = esecuzione->id;

        return m;
}

#include <mboot.h>
#include <elf64.h>

// oggetto da usare con 'map' per il caricamento in memoria virtuale dei
// segmenti ELF dei moduli utente e I/O
struct copy_segment {
        // Il segmento si trova in memoria agli indirizzi (fisici) [beg, end)
        // e deve essere visibile in memoria virtuale a partire dall'indirizzo
        // virt_beg. Il segmento verrÃ  copiato (una pagina alla volta) in
        // frame liberi di M2. La memoria precedentemente occupata dal modulo
        // sarÃ  riutilizzata per lo heap di sistema.
        paddr mod_beg;
        paddr mod_end;
        vaddr virt_beg;

        // funzione chiamata da map. Deve restituire l'indirizzo fisico
        // da far corrispondere all'indirizzo virtuale 'v'.
        paddr operator()(vaddr);
};

paddr copy_segment::operator()(vaddr v)
{
        // allochiamo un frame libero in cui copiare la pagina
        paddr dst = alloca_frame();
        if (dst == 0)
                return 0;

        // offset della pagina all'interno del segmento
        natq offset = v - virt_beg;
        // indirizzo della pagina all'interno del modulo
        paddr src = mod_beg + offset;

        // il segmento in memoria puÃ² essere piÃ¹ grande di quello nel modulo.
        // La parte eccedente deve essere azzerata.
        natq tocopy = DIM_PAGINA;
        if (src > mod_end)
                tocopy = 0;
        else if (mod_end - src < DIM_PAGINA)
                tocopy =  mod_end - src;
        if (tocopy > 0)
                memcpy(reinterpret_cast<void*>(dst), reinterpret_cast<void*>(src), tocopy);
        if (tocopy < DIM_PAGINA)
                memset(reinterpret_cast<void*>(dst + tocopy), 0, DIM_PAGINA - tocopy);
        return dst;
}

// carica un modulo in M2 e lo mappa al suo indirizzo virtuale, aggiungendo
// heap_size byte di heap dopo l'ultimo indirizzo virtuale usato.
//
// 'flags' dovrebbe essere BIT_US oppure zero.
vaddr carica_modulo(multiboot_module_t* mod, paddr root_tab, natq flags, natq heap_size)
{
        // puntatore all'intestazione ELF
```

```
        Elf64_Ehdr* elf_h  = reinterpret_cast<Elf64_Ehdr*>(mod->mod_start);
        // indirizzo fisico della tabella dei segmenti
        paddr ph_addr = mod->mod_start + elf_h->e_phoff;
        // ultimo indirizzo virtuale usato
        vaddr last_vaddr = 0;

        // esaminiamo tutta la tabella dei segmenti
        for (int i = 0; i < elf_h->e_phnum; i++) {
                Elf64_Phdr* elf_ph = reinterpret_cast<Elf64_Phdr*>(ph_addr);

                // ci interessano solo i segmenti di tipo PT_LOAD
                if (elf_ph->p_type != PT_LOAD)
                        continue;

                // i byte che si trovano ora in memoria agli indirizzi (fisici)
                // [mod_beg, mod_end) devono diventare visibili nell'intervallo
                // di indirizzi virtuali [virt_beg, virt_end).
                vaddr virt_beg = elf_ph->p_vaddr,
                      virt_end = virt_beg + elf_ph->p_memsz;
                paddr mod_beg  = mod->mod_start + elf_ph->p_offset,
                      mod_end  = mod_beg + elf_ph->p_filesz;

                // se necessario, allineiamo alla pagina gli indirizzi di
                // partenza e di fine
                natq page_offset = virt_beg & (DIM_PAGINA - 1);
                virt_beg -= page_offset;
                mod_beg  -= page_offset;
                virt_end = allinea(virt_end, DIM_PAGINA);

                // aggiorniamo l'ultimo indirizzo virtuale usato
                if (virt_end > last_vaddr)
                        last_vaddr = virt_end;

                // settiamo BIT_RW nella traduzione solo se il segmento Ã¨
                // scrivibile
                if (elf_ph->p_flags & PF_W)
                        flags |= BIT_RW;

                // mappiamo il segmento
                if (map(root_tab,
                         virt_beg,
                         virt_end,
                         flags,
                         copy_segment{mod_beg, mod_end, virt_beg}) != virt_end)
                        return 0;

                flog(LOG_INFO, " - segmento %s %s mappato a [%p, %p)",
                                (flags & BIT_US) ? "utente " : "sistema",
                                (flags & BIT_RW) ? "read/write" : "read-only ",
                                virt_beg, virt_end);

                // passiamo alla prossima entrata della tabella dei segmenti
                ph_addr += elf_h->e_phentsize;
        }
        // dopo aver mappato tutti i segmenti, mappiamo lo spazio destinato
        // allo heap del modulo. I frame corrispondenti verranno allocati da
        // alloca_frame()
        if (map(root_tab,
                 last_vaddr,
                 last_vaddr + heap_size,
                 flags | BIT_RW,
                 [](vaddr) { return alloca_frame(); }) != last_vaddr + heap_size)
                return 0;
        flog(LOG_INFO, " - heap:                                  [%p, %p)",
                                last_vaddr, last_vaddr + heap_size);
        flog(LOG_INFO, " - entry point: %p", elf_h->e_entry);
        return elf_h->e_entry;
}

vaddr carica_IO(multiboot_module_t* mod, paddr root_tab)
{
        flog(LOG_INFO, "mappo il modulo I/O:");
        return carica_modulo(mod, root_tab, 0, DIM_IO_HEAP);
}

vaddr carica_utente(multiboot_module_t* mod, paddr root_tab)
{
        flog(LOG_INFO, "mappo il modulo utente:");
        return carica_modulo(mod, root_tab, BIT_US, DIM_USR_HEAP);
}

// sezioni exception-handler dei moduli (utilizzate dalle funzioni di
// stack-unwinding in libce, per implementare il backtrace)
vaddr sis_eh_frame;
natq  sis_eh_frame_len;
vaddr mio_eh_frame;
natq  mio_eh_frame_len;
vaddr utn_eh_frame;
natq  utn_eh_frame_len;

bool crea_spazio_condiviso(paddr root_tab, paddr mbi_)
{
```

```
        multiboot_info_t* mbi = reinterpret_cast<multiboot_info_t*>(mbi_);
        multiboot_module_t *mod = reinterpret_cast<multiboot_module_t*>(mbi->mods_addr);

        io_entry = reinterpret_cast<void(*)(natq)>(carica_IO(&mod[1], root_tab));
        user_entry = reinterpret_cast<void(*)(natq)>(carica_utente(&mod[2], root_tab));

        // per il supporto al backtrace
        find_eh_frame(mod[0].mod_start, sis_eh_frame, sis_eh_frame_len);
        find_eh_frame(mod[1].mod_start, mio_eh_frame, mio_eh_frame_len);
        find_eh_frame(mod[2].mod_start, utn_eh_frame, utn_eh_frame_len);

        return io_entry && user_entry;
}

// backtrace
#include <cfi.h>

// callback invocata dalla funzione cfi_backstep() per leggere
// dalla pila di un qualunque processo
natq read_mem(void *token, vaddr v)
{
        des_proc *p = static_cast<des_proc*>(token);
        paddr pa = trasforma(p->cr3, v);
        natq rv = 0;
        if (pa) {
                rv = *reinterpret_cast<natq*>(pa);
        }
        return rv;
}

// invia sul log il backtrace (stack delle chiamate) del processo p
void backtrace(des_proc *p, log_sev sev, const char* msg)
{
        cfi_d cfi;

        cfi.regs[CFI::RAX] = p->contesto[I_RAX];
        cfi.regs[CFI::RCX] = p->contesto[I_RCX];
        cfi.regs[CFI::RDX] = p->contesto[I_RDX];
        cfi.regs[CFI::RBX] = p->contesto[I_RBX];
        cfi.regs[CFI::RSP] = read_mem(p, p->contesto[I_RSP] + 24);
        cfi.regs[CFI::RBP] = p->contesto[I_RBP];
        cfi.regs[CFI::RSI] = p->contesto[I_RSI];
        cfi.regs[CFI::RDI] = p->contesto[I_RDI];
        cfi.regs[CFI::R8]  = p->contesto[I_R8];
        cfi.regs[CFI::R9]  = p->contesto[I_R9];
        cfi.regs[CFI::R10] = p->contesto[I_R10];
        cfi.regs[CFI::R11] = p->contesto[I_R11];
        cfi.regs[CFI::R12] = p->contesto[I_R12];
        cfi.regs[CFI::R13] = p->contesto[I_R13];
        cfi.regs[CFI::R14] = p->contesto[I_R14];
        cfi.regs[CFI::R15] = p->contesto[I_R15];

        cfi.token = p;
        cfi.read_stack = read_mem;

        vaddr rip = read_mem(p, p->contesto[I_RSP]);
        do {
                if (rip >= ini_sis_c && rip < fin_sis_c) {
                        cfi.eh_frame = sis_eh_frame;
                        cfi.eh_frame_len = sis_eh_frame_len;
                } else if (rip >= ini_mio_c && rip < fin_mio_c) {
                        cfi.eh_frame = mio_eh_frame;
                        cfi.eh_frame_len = mio_eh_frame_len;
                } else if (rip >= ini_utn_c && rip < fin_utn_c) {
                        cfi.eh_frame = utn_eh_frame;
                        cfi.eh_frame_len = utn_eh_frame_len;
                } else {
                        cfi.eh_frame = 0;
                        cfi.eh_frame_len = 0;
                }

                if (!cfi.eh_frame)
                        break;

                if (!cfi_backstep(cfi, rip))
                        break;

                rip = cfi.regs[CFI::RA];

                if (!rip)
                        break;

                flog(sev, "%s%p", msg, rip - 1);
        } while (true);
}
```