```
// [...]

//////////////////////////////////////////////////////////////////////////////
//                          PAGINE FISICHE                                  //
//////////////////////////////////////////////////////////////////////////////

// avremo un descrittore di pagina fisica per ogni pagina fisica della parte
// M2.  Lo scopo del descrittore e' di contenere alcune informazioni relative
// al contenuto della pagina fisica descritta. Tali informazioni servono
// principalmente a facilitare o rendere possibile il rimpiazzamento del
// contenuto stesso.
struct des_frame
{
        int     livello;        // 0=pagina, -1=libera

// ESAME 2017-02-07

    /**
     * We change this variable from boolean to natl in order to be able to count
     * the number of times the residente() primitive is called upon a page.
     */
    natl residente;

//   ESAME 2017-02-07 )

        // identificatore del processo a cui appartiene l'entita'
        // contenuta nel frame.
        natl    processo;
        natl    contatore;      // contatore per le statistiche
        // blocco da cui l'entita' contenuta nel frame era stata caricata
        natq    ind_massa;
        // per risparmiare un po' di spazio uniamo due campi che
        // non servono mai insieme:
        // - ind_virtuale serve solo se il frame e' occupato
        // - prossimo_libero serve solo se il frame e' libero
        union {
                // indirizzo virtuale che permette di risalire al
                // descrittore che punta all'entita' contenuta nel
                // frame. Per le pagine si tratta di un qualunque
                // indirizzo virtuale interno alla pagina. Per le
                // tabelle serve un qualunque indirizzo virtuale la
                // cui traduzione passa dalla tabella.
                vaddr   ind_virtuale;
                des_frame*      prossimo_libero;
        };
};

// [...]

// EXTENSION 2017-02-07

/**
 * Page faults counter.
 */
natq pf_count = 0;

// EXTENSION 2017-02-07

void stat();

bool c_routine_pf()
{
        vaddr ind_virt = readCR2();
        natl proc = esecuzione->id;

        stat();

// EXTENSION 2017-02-07

    // increase page faults counter
```

```
    pf_count++;

// EXTENSION 2017-02-07

        for (int i = 3; i >= 0; i--) {
                tab_entry d = get_des(proc, i + 1, ind_virt);
                bool bitP = extr_P(d);
                if (!bitP) {
                        des_frame *df = swap(proc, i, ind_virt);
                        if (!df)
                                return false;
                }
        }
        return true;
}

bool vietato(des_frame* df, natl proc, int liv, vaddr ind_virt)
{
        if (df->livello > liv && df->processo == proc &&
            base(df->ind_virtuale, df->livello) == base(ind_virt, df->livello))
                return true;
        return false;
}

des_frame* scegli_vittima(natl proc, int liv, vaddr ind_virt)
{
        des_frame *df, *df_vittima;
        df = &vdf[0];
        while ( df < &vdf[N_DF] &&
                (df->residente ||
                 vietato(df, proc, liv, ind_virt)))
                df++;
        if (df == &vdf[N_DF]) return 0;
        df_vittima = df;
        for (df++; df < &vdf[N_DF]; df++) {
                if (df->residente ||
                    vietato(df, proc, liv, ind_virt))
                        continue;
                if (df->contatore < df_vittima->contatore ||
                    (df->contatore == df_vittima->contatore &&
                     df_vittima->livello > df->livello))
                        df_vittima = df;
        }
        return df_vittima;
}

void stat()
{
        des_frame *df1, *df2;
        faddr f1, f2;
        bool bitA;

        for (natq i = 0; i < N_DF; i++) {
                df1 = &vdf[i];
                if (df1->livello < 1)
                        continue;
                f1 = indirizzo_frame(df1);
                for (int j = 0; j < 512; j++) {
                        tab_entry& des = get_entry(f1, j);
                        if (!extr_P(des))
                                continue;
                        bitA = extr_A(des);
                        set_A(des, false);
                        f2 = extr_IND_FISICO(des);
                        df2 = descrittore_frame(f2);
                        if (!df2 || df2->residente)
                                continue;
                        df2->contatore >>= 1;
                        if (bitA)
                                df2->contatore |= 0x80000000;
```

```
                }
        }
        invalida_TLB();
}


// funzione di supporto per carica_tutto()
bool carica_ric(natl proc, faddr tab, int liv, vaddr ind, natl n)
{
        natq dp = dim_region(liv);

        natl i = i_tab(ind, liv + 1);
        for (natl j = i; j < i + n; j++, ind += dp) {
                tab_entry e = get_entry(tab, j);
                if (!extr_IND_MASSA(e))
                        continue;
                des_frame *df = swap(proc, liv, ind);
                if (!df) {
                        flog(LOG_ERR, "impossibile caricare pagina virtuale %p", ind);
                        return false;
                }
                df->residente = true;
                if (liv > PRELOAD_LEVEL &&
                                !carica_ric(proc, indirizzo_frame(df), liv - 1, ind, 512)
)
                        return false;
        }
        return true;
}

// carica e rende residenti tutte le pagine e tabelle allocate nello swap e
// relative alle entrate della tab4 del processo proc che vanno da i (inclusa)
// a i+n (esclusa)
bool carica_tutto(natl proc, natl i, natl n)
{
        des_proc *p = des_p(proc);

        return carica_ric(proc, p->cr3, 3, norm(i * dim_region(3)), n);
}



// super blocco (vedi e [P_SWAP] avanti)
struct superblock_t {
        char    magic[8];
        natq    bm_start;
        natq    blocks;
        natq    directory;
        void    (*user_entry)(int);
        natq    user_end;
        void    (*io_entry)(int);
        natq    io_end;
        int     checksum;
};

// descrittore di swap (vedi [P_SWAP] avanti)
struct des_swap {
        natl *free;             // bitmap dei blocchi liberi
        superblock_t sb;        // contenuto del superblocco
} swap_dev;     // c'e' un unico oggetto swap
bool swap_init();

// chiamata in fase di inizializzazione, carica in memoria fisica
// tutte le parti condivise di livello IO e utente.
bool crea_spazio_condiviso()
{

        // ( lettura del direttorio principale dallo swap
        flog(LOG_INFO, "lettura del direttorio principale...");
        addr tmp = alloca(DIM_PAGINA);
```

```
        if (tmp == 0) {
                flog(LOG_ERR, "memoria insufficiente");
                return false;
        }
        leggi_swap(tmp, swap_dev.sb.directory);
        // )

        // (  carichiamo le parti condivise nello spazio di indirizzamento del processo
        //    dummy
        faddr dummy_dir = des_p(dummy_proc)->cr3;
        copy_des((faddr)tmp, dummy_dir, I_MIO_C, N_MIO_C);
        copy_des((faddr)tmp, dummy_dir, I_UTN_C, N_UTN_C);
        dealloca(tmp);

        if (!carica_tutto(dummy_proc, I_MIO_C, N_MIO_C))
                return false;
        if (!carica_tutto(dummy_proc, I_UTN_C, N_UTN_C))
                return false;
        // )

        // ( copiamo i descrittori relativi allo spazio condiviso anche nel direttorio
        //   corrente, in modo che vengano ereditati dai processi che creeremo in seguito
        faddr my_dir = des_p(esecuzione->id)->cr3;
        copy_des(dummy_dir, my_dir, I_MIO_C, N_MIO_C);
        copy_des(dummy_dir, my_dir, I_UTN_C, N_UTN_C);
        // )

        invalida_TLB();
        return true;
}

// EXTENSION 2017-02-07

/**
 * Permanent region descriptor.
 */
struct res_des
{
    // starting virtual address
    vaddr start;

    // region size
    natq size;

    // owner process
    natl proc;
};

/**
 *
 */
res_des array_res[MAX_RES];

/**
 *
 */
natl alloca_res(vaddr start, natq size)
{
    res_des *r = 0;

    natl id = 0xffffffff;

    for (int i = 0; i < MAX_RES; i++)
    {
        r = &array_res[i];

        if (r->proc == 0)
        {
            id = i;
```

```
            break;
        }
    }

    if (r)
    {
        r->start = start;

        r->size = size;

        r->proc = esecuzione->id;
    }

    return id;
}

/**
 * Checks if the given resident id is valid: an ID is valid if it is not higher
 * than the maximum value and if it belongs to the calling process. A different
 * process can not undo resident operation performed by other processes. This is
 * because the CR3 CPU register content differs from process to process.
 */
bool res_valido(natl id)
{
    return (id < MAX_RES) && (esecuzione->id == array_res[id].proc);
}

/**
 *
 */
void rilascia_res(natl id)
{
    array_res[id].proc = 0;
}

/**
 *
 */
extern "C" natq c_countres()
{
    natq c = 0;

    for (natq i = 0; i < N_DF; i++)
    {
        des_frame* ppf = &vdf[i];

        if (ppf->livello >= 0 && ppf->residente > 0)
        {
            c++;
        }
        }

    return c | (pf_count << 32);
}

/**
 * Decreases the value of the residente field for all the tables and pages
 * of level i regarding the addressed area [base, stop).
 *
 * @param  start  starting address;
 * @param  stop   ragion size;
 * @param  i      table level.
 */
void undo_res(natq start, natq stop, int i)
{
    // retrieve calling process ID
    natl proc = esecuzione->id;

    // per capire quali tabelle/pagine di livello j dobbiamo
    // rendere non residenti calcoliamo:
```

```
    // vi: l'indirizzo virtuale della prima regione di livello i
    //     che interseca [start, stop)
    // vf: l'indirizzo virtuale della prima regione di livello i
    //     che si trova oltre vi e non interseca [start, stop)
    vaddr vi = base(start, i);
    vaddr vf = base(stop - 1, i) + dim_region(i);

    for (natq v = vi; v != vf; v += dim_region(i))
    {
        // otteniamo il descrittore che punta a questa tabella/pagina
        natq& d = get_des(proc, i + 1, v);

        // se prima era residente, deve essere presente, quindi
        // possiamo estrarre l'indirizzo fisico e ottenere da questo
        // il puntatore al descrittore di pagina fisica
        des_frame *ppf = descrittore_frame(extr_IND_FISICO(d));
        ppf->residente--;
    }
}

// EXTENSION 2017-02-07

// SOLUTION 2017-02-07

/**
 * Makes the virtual pages address by start and start+s (size) permanent.
 * It must also move the missing pages to the available frames and notify in
 * case there is not enough space for all of them.
 *
 * @param  start  virtual pages starting address;
 * @param  s      virtual pages size.
 *
 * @return  the operation ID which can be used to undo it.
 */
extern "C" void c_resident(addr start, natq s)
{
    // retrieve calling process id
    natl proc = esecuzione->id, id;

    int i;

    vaddr v;

    // retrieve start (inclusive) and end (exclusive) addresses of the virtual
    // pages to be made permanent
    vaddr a = reinterpret_cast<vaddr>(start);
    vaddr b = a + s - 1;

    // retrieve calling process descriptor
    des_proc *self = des_p(proc);

    // return value
    self->contesto[I_RAX] = 0xFFFFFFFF;

    // check if the addressed virtual pages belong to the private user memory
    // space: keep in mind that this space starts from ini_utn_p and ends at
    // fin_utn_p
    if (a < ini_utn_p || a + s < a /* overflow */ || a + s >= fin_utn_p)
    {
        // print a warning log message
        flog(LOG_WARN, "Invalid parameters: %p, %p", a, s);

        // abort calling process
        c_abort_p();

        // just return to the caller
        return;
    }

    // loop through virtual memory tables levels
```

```
    // for each level
    for (i = 3; i >= 0; i--)
    {
        // retrieve level 'i' region starting address for the virtual address
        // 'a' (a contains the virtual pages starting virtual address)
        vaddr vi = base(a, i);

        // and level 'i' region ending address for the virtual address 'b' (b
        // contains the virtual pages end virtual address): keep in mind that
        // base() will return the start address of the regione where b is
        // contained that is why we need to add the region dimension for level i
        vaddr vf = base(b, i) + dim_region(i);

        // print log message for debugging purposes
        flog(LOG_DEBUG, "liv %d: vi %p vf %p", i, vi, vf);

        // start from the region starting address, for each region
        for (v = vi; v != vf; v += dim_region(i))
        {
            // retrieve table entry of level i+1 containing v
            tab_entry& d = get_des(proc, i + 1, v);

            des_frame *ppf;

            // check the table entry P bit: it zero (the page is in the swap)
            if (!extr_P(d))
            {
                // swap in the missing page in the process addressing area
                ppf = swap(proc, i, v);

                // check if the swap-in succeeded
                if (!ppf)
                {
                    // if not, go to error
                    goto error;
                }
            }
            else
            {
                // if the entry P is set to 1: retrieve frame descriptor
                ppf = descrittore_frame(extr_IND_FISICO(d));
                        }

            // increment residente field to make the region permanent
            ppf->residente++;
        }
    }

    // create an ID for this whole operation in order to be able to undo
    // everything later using the c_nonresident()
    id = alloca_res(a, s);

    if (id == 0xffffffff)
    {
        goto error;
    }

    // return operation ID
    self->contesto[I_RAX] = id;

    return;

// in case of error all regions made permanent must be undone
error:
    for (int j = 3; j >= i + 1; j--)
    {
        undo_res(a, a + s, j);
    }

    undo_res(a, v, i);
```

```
}

// SOLUTION 2017-02-07

// EXTENSION 2017-02-07

/**
 * This method can be used to undo the operation performed by c_resident()
 * passing as ID the value returned by c_resident().
 *
 * @param  id  the id of the operation performed by c_resident() to be undone.
 */
extern "C" void c_nonresident(natl id)
{
    // permanent region descrpitor
    res_des *r;

    // check if the given resident id is valid
    if (!res_validо(id))
    {
        // print warning log message
        flog(LOG_WARN, "Invalid Resident ID: %d", id);

        // abort calling process
        c_abort_p();

        // just return to the caller
        return;
    }

    // retrieve permanent region descriptor
    r = &array_res[id];

    // retrieve region starting address
    vaddr a = r->start;

    // retrieve region size
    natq s = r->size;

    // loop through different table levels of the memory
    for (int i = 3; i >= 0; i--)
    {
        // undo each and every resident operation performed
        undo_res(a, a + s, i);
    }

    // release permanent region
    rilascia_res(id);
}

// EXTENSION 2017-02-07

// [...]
```