Prova pratica di Calcolatori Elettronici

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

7 febbraio 2017

1. Siano date le seguenti dichiarazioni, contenute nel file `cc.h`:

```
struct st1 { char vi[4]; };
struct st2 { long vd[4]; };
class cl {
        char v1[4]; int v3[4]; long v2[4];
public:
        cl(st1 ss);
        cl(st1& s1, int ar2[]);
        cl elab1(char ar1[], const st2& s2);
        void stampa() {
                for (int i = 0; i < 4; i++) cout << (int)v1[i] << ' '; cout << endl;
                for (int i = 0; i < 4; i++) cout << (int)v2[i] << ' '; cout << endl;
                for (int i = 0; i < 4; i++) cout << v3[i] << ' '; cout << endl << endl;
        }
};
```

   Realizzare in Assembler GCC le funzioni membro seguenti.

```
cl::cl(st1 ss)
{
        for (int i = 0; i < 4; i++) {
                v1[i] = ss.vi[i]; v2[i] = ss.vi[i];
                v3[i] = 4 * ss.vi[i];
        }
}
cl::cl(st1& s1, int ar2[])
{
        for (int i = 0; i < 4; i++) {
                v1[i] = s1.vi[i]; v2[i] = -s1.vi[i];
                v3[i] = ar2[i];
        }
}
cl cl::elab1(char ar1[], const st2& s2)
{
        st1 s1;
        for (int i = 0; i < 4; i++) s1.vi[i] = ar1[i] - i;
        cl cla(s1);
        for (int i = 0; i < 4; i++) cla.v3[i] = s2.vd[i];
        return cla;
}
```

2. Vogliamo estendere il nucleo in modo da permettere ai processi di rendere temporaneamente residenti alcune delle loro pagine virtuali private (nella zona utente/privata, contenente la pila di livello utente).

   A tale scopo aggiungiamo al nucleo le seguenti primitive:

   - `natl resident(addr base, natq size)`: rende residenti le pagine virtuali che contengono gli indirizzi da `base` (incluso) a `base`+`size` (escluso). Restituisce un identificatore che può poi essere passato a `nonresident()` per disfare l'operazione, o `0xffffffff` in caso di fallimento; È un errore se gli indirizzi da `base` (incluso) a `base`+`size` (escluso) non sono all'interno della zona utente/privata;
   - `void nonresident(natl id)`: disfa l'operazione di una precedente chiamata a `resident()`; è un errore se `id` non corrisponde ad una precedente operazione `resident()`.

   Se la primitiva `resident()` ha successo, non devono essere più possibili page fault nelle pagine interessate fino alla corrispondente `nonresident()`. Questo vuol dire che la primitiva deve anche caricare le necessarie pagine o tabelle assenti, e marcarle tutte come residenti in modo che non possano essere rimpiazzate. La primitiva `resident()` può fallire se non riesce a caricare una pagina o tabella (ad. es., perchè tutta la memoria è occupata da pagine residenti).

   Le stesse pagine o tabelle possono essere coinvolte in più operazioni `resident()` distinte. Per permettere ciò trasformiamo il campo `residente` dei descrittori di pagina fisica in un contatore (conta il numero di operazioni `resident()` non ancora disfatte sulla tabella o pagina corrispondente). Se la primitiva `resident()` fallisce, deve riportare i contatori `residente` al valore che avevano prima dell'inizio della primitiva.

   Le primitive abortiscono il processo chiamante in caso di errore.

   Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti.

   SUGGERIMENTI:

   - si osservi con attenzione la funzione `undo_res()`, già presente nel codice, e come è usata in `c_nonresident()`;
   - è possibile usare la funzione `des_pf* swap2(natl proc, int liv, addr ind_virt)` che carica nello spazio del processo `proc` la tabella o pagina di livello `liv` relativa all'indirizzo `ind_virt` e restituisce il puntatore al descrittore di pagina fisica della pagina in cui l'entità è stata caricata; restituisce 0 se non è stato possibile caricare la pagina (memoria piena e impossibile rimpiazzare).
   - La parte utente privata va da `ini_utn_p` (incluso) a `fin_unt_p` (escluso).

   Sistema a 32bit: usare `natl` al posto di `natq`; la funzione `swap2` usa `tt tipo` come secondo argomento; la parte utente privata va da `inizio_utente_privato` (incluso) a `fine_utente_privato` (escluso).

```
/**
 * File: cc.h
 *       Contains the declaration for the data structures used in the exercise.
 *
 * Author: Rambod Rahmani <rambodrahmani@autistici.org>
 *         Created on 17/09/2019.
 */

#include <iostream>

using namespace std;

struct st1
{
    char vi[4];
};

struct st2
{
    long vd[4];
};

class cl
{
    char v1[4];
    int v3[4];
    long v2[4];

public:
    cl(st1 ss);

    cl(st1& s1, int ar2[]);

    cl elab1(char ar1[], const st2& s2);

    void stampa()
    {
        for (int i = 0; i < 4; i++)
        {
            cout << (int)v1[i] << ' ';
        }
        cout << endl;

        for (int i = 0; i < 4; i++)
        {
            cout << (int)v2[i] << ' ';
        }
        cout << endl;

        for (int i = 0; i < 4; i++)
        {
            cout << v3[i] << ' ';
        }
        cout << endl << endl;
    }
};
```

```
/**
 * File: es1.cpp
 *       Contains the C++ code to be translated into Assembly (es1.s file).
 *
 * Author: Rambod Rahmani <rambodrahmani@autistici.org>
 *         Created on 17/09/2019.
 */

#include "cc.h"

cl::cl(st1 ss)
{
    for (int i = 0; i < 4; i++)
    {
        v1[i] = ss.vi[i];
        v2[i] = ss.vi[i];
        v3[i] = 4 * ss.vi[i];
    }
}

cl::cl(st1& s1, int ar2[])
{
    for (int i = 0; i < 4; i++)
    {
        v1[i] = s1.vi[i];
        v2[i] = -s1.vi[i];
        v3[i] = ar2[i];
    }
}

cl cl::elab1(char ar1[], const st2& s2)
{
    st1 s1;

    for (int i = 0; i < 4; i++)
    {
        s1.vi[i] = ar1[i] - i;
    }

    cl cla(s1);

    for (int i = 0; i < 4; i++)
    {
        cla.v3[i] = s2.vd[i];
    }

    return cla;
}
```

```
#*****************************************************************************
# File: es1.s
#       Contains the Assembly translation for es1.cpp.
#
# Author: Rambod Rahmani <rambodrahmani@autistici.org>
#         Created on 14/09/2019.
#*****************************************************************************

#-----------------------------------------------------------------------------
.TEXT
.GLOBAL _ZN2clC1E3st1                                          # cl::cl(st1 ss)
#-----------------------------------------------------------------------------
# activation frame:
# -----------------
#  i            -16
#  ss           -12
#  &this        -8
#  %rbp         0
#-----------------------------------------------------------------------------
_ZN2clC1E3st1:
# set stack locations labels
    .set this, -8
    .set ss,   -12
    .set i,    -16

# prologue: activation frame
    pushq %rbp
    movq  %rsp, %rbp
    subq  $16, %rsp                # reserver stack space for actual arguments

# copy actual arguments to the stack
    movq %rdi, this(%rbp)
    movl %esi, ss(%rbp)

# foor loop initialization:
    movl $0, i(%rbp)               # i = 0

for:
    cmpl $4, i(%rbp)               # check if i < 4
    jge  finefor                   # end for loop (i >= 4)

# for loop body:
    movq   this(%rbp), %rdi        # &this -> %rdi
    leaq   ss(%rbp), %rsi          # &ss -> %rsi
    movslq i(%rbp), %rcx           # i => %rcx
    movb   (%rsi, %rcx, 1), %al    # ss.vi[i] -> %al
    movb   %al, (%rdi, %rcx, 1)    # v1[i] = ss.vi[i];
    movsbq %al, %rax               # %al => %rax
    movq   %rax, 24(%rdi, %rcx, 8) # v2[i] = ss.vi[i];
    movsbl %al, %eax               # %al => %eax
    sal    $2, %eax                # 4 * ss.vi[i] -> %eax
    movl   %eax, 4(%rdi, %rcx, 4)  # v3[i] = 4 * ss.vi[i];

    incl i(%rbp)                   # i++
    jmp  for                       # loop again

finefor:

    movq  this(%rbp), %rax         # return initialized object address
    leave                          # movq %rbp, %rsp; popq %rbp
    ret

#-----------------------------------------------------------------------------
.GLOBAL _ZN2clC1ER3st1Pi                          # cl::cl(st1& s1, int ar2[])
#-----------------------------------------------------------------------------
# activation record:
# ------------------
#  i            -28
#  &ar2         -24
#  &s1          -16
```

```
#  this         -8
#  %rbp         0
#-----------------------------------------------------------------------------
_ZN2clC1ER3st1Pi:
# set stack locations labels
    .set this, -8
    .set s1,   -16
    .set ar2,  -24
    .set i,    -28

# prologue: activation frame
    pushq %rbp
    movq  %rsp, %rbp
    subq  $32, %rsp                 # reserve stack space for actual arguments

# copy actual arguments to the stack
    movq %rdi, this(%rbp)
    movq %rsi, s1(%rbp)
    movq %rdx, ar2(%rbp)

# for loop intialization
    movl $0, i(%rbp)                # i = 0

for1:
    cmpl $4, i(%rbp)                # check if i < 4
    jge  finefor1                   # end for loop (i >= 4)

# for loop body
    movq   this(%rbp), %rdi         # &this -> %rdi
    movq   s1(%rbp), %rsi           # &s1 -> %rsi
    movq   ar2(%rbp), %rdx          # &ar2 -> %rdx
    movslq i(%rbp), %rcx            # i => %rcx
    movb   (%rsi, %rcx, 1), %al     # s1.vi[i] -> %al
    movb   %al, (%rdi, %rcx, 1)     # v1[i] = s1.vi[i];
    neg    %al                      # -s1.vi[i] -> %al
    movsbq %al, %rax                # %al => %rax
    movq   %rax, 24(%rdi, %rcx, 8)  # v2[i] = -s1.vi[i];
    movl   (%rdx, %rcx, 4), %eax    # ar2[i] -> %eax
    movl   %eax, 4(%rdi, %rcx, 4)   # v3[i] = ar2[i];

    incl i(%rbp)                    # i++
    jmp  for1                       # loop again

finefor1:

    leave                           # movq %rbp, %rsp; popq %rbp
    ret

#-----------------------------------------------------------------------------
.GLOBAL _ZN2cl5elab1EPcRK3st2          # cl cl::elab1(char ar1[], const st2& s2)
#-----------------------------------------------------------------------------
# activation frame:
# -----------------
#  i            -100
#  cla.v1       -96
#  cla.v3[0-1]  -92
#  cla.v3[2-3]  -84
#  cla.v2[0]    -72
#  cla.v2[1]    -64
#  cla.v2[2]    -56
#  cla.v2[3]    -48
#  s1           -40
#  &s2          -32
#  &ar1         -24
#  this         -16
#  indo         -8
#  %rbp         0
#-----------------------------------------------------------------------------
_ZN2cl5elab1EPcRK3st2:
# set stack locations labels
```

```
    .set indo, -8
    .set this, -16
    .set ar1,  -24
    .set s2,   -32
    .set s1,   -40
    .set cla,  -96
    .set i,    -100

# prologue: activation frame
    pushq %rbp
    movq  %rsp, %rbp
    subq  $100, %rsp                # reserve stack space for actual arguments

# copy actual arguments to the stack
    movq %rdi, indo(%rbp)
    movq %rsi, this(%rbp)
    movq %rdx, ar1(%rbp)
    movq %rcx, s2(%rbp)

# for loop 1 initialization
    movl $0, i(%rbp)                # i = 0

for2:
    cmpl $4, i(%rbp)                # check if i < 4
    jge  finefor2                   # end for loop (i >= 4)

# for loop 1 body
    movq   this(%rbp), %rdi         # &this -> %rdi
    movq   ar1(%rbp), %rsi          # &ar1 -> %rsi
    movq   s2(%rbp), %rdx           # &s2 -> %rdx
    movslq i(%rbp), %rcx            # i => %rcx
    movb   (%rsi, %rcx, 1), %al     # ar1[i] -> %al
    subb   %cl, %al                 # ar1[i] - i -> %al
    leaq   s1(%rbp), %r8            # &s1 -> %r8
    movb   %al, (%r8, %rcx, 1)      # s1.vi[i] = ar1[i] - i;

    incl i(%rbp)                    # i++
    jmp  for2                       # loop again

finefor2:

# cl cla(s1);
    leaq cla(%rbp), %rdi
    movl s1(%rbp), %esi
    call _ZN2clC1E3st1

# for loop 2 initialization
    movl $0, i(%rbp)                # i = 0

for3:
    cmpl $4, i(%rbp)                # check if i < 4
    jge  finefor3                   # end for loop (i >= 4)

# for loop 2 body
    movq   this(%rbp), %rdi         # &this -> %rdi
    leaq   -92(%rbp), %rsi          # &cla.v3 -> %rsi
    movq   s2(%rbp), %rdx           # &s2 -> %rdx
    movslq i(%rbp), %rcx            # i => %rcx
    movq   (%rdx, %rcx, 8), %rax    # s2.vd[i] -> %rax
    movl   %eax, (%rsi, %rcx, 4)    # cla.v3[i] = s2.vd[i];

    incl i(%rbp)                    # i++
    jmp  for3                       # loop again

finefor3:

# copy return object from stack to the address in indo
    leaq cla(%rbp), %rsi            # rep movsq source address
    movq indo(%rbp), %rdi           # rep movsq destination address
    movabsq $7, %rcx                # rep movsq repetitions
```

```
    rep movsq                       # rep movsq, [0]
    movq indo(%rbp), %rax           # return initialized object address

    leave                           # movq %rbp, %rsp; popq %rbp;
    ret

#*****************************************************************************
```

```
/**
 * File: prova1.cpp
 *       This file contains a developer harness test for es1.s.
 *
 *       Compile with:
 *          g++ -o es1 -fno-elide-constructors es1.s prova1.cpp
 *
 *       Test your result with:
 *          ./es1 | diff - es1.out
 *
 * Author: Rambod Rahmani <rambodrahmani@autistici.org>
 *         Created on 17/09/2019.
 */

#include "cc.h"

/**
 * Developer harness test.
 *
 * @param   argc    command line arguments counter.
 * @param   argv    command line arguments.
 *
 * @return          execution exit code.
 */
int main(int argc, char * argv[])
{
    st1 s1 = { 1, 2, 3, 4 };

    st2 s2 = { 5, 6, 7, 8 };

    char a1[4] = { 11, 12, 13, 14 };

    int a2[4] = { 15, 16, 17, 18 };

    cl cla1(s1);

    cla1.stampa();

    cl cla2(s1, a2);

    cla2.stampa();

    cla1 = cla2.elab1(a1, s2);

    cla1.stampa();

    return 0;
}
```

```
1 2 3 4
1 2 3 4
4 8 12 16

1 2 3 4
-1 -2 -3 -4
15 16 17 18

11 11 11 11
11 11 11 11
5 6 7 8
```

```
// [...]

// EXTENSION 2017-02-07

/**
 *
 */
#define MAX_RES     10

// EXTENSION 2017-02-07

// [...]

// EXTENSION 2017-02-07

/**
 * User module primitives interrupt types definitions.
 */

/**
 * natl resident(addr base, natq size);
 */
#define TIPO_RES     0x59

/**
 * void nonresident(natl id);
 */
#define TIPO_NONRES  0x5a

/**
 * natq countres();
 */
#define TIPO_CRES    0x5b

// EXTENSION 2017-02-07

// [...]
```

```
// [..]

// EXTENSION 2017-02-07

/**
 * We want to provide to the kernel the ability to allow user processes to make
 * their virtual pages permanent. To this end, we will provide the resident()
 * primitive which will make the addressed virtual pages permanent until the
 * nonresident() primitive is used to undo the operation.
 */

/**
 * User Primitives declarations.
 */

/**
 *
 */
extern "C" natq countres();

/**
 * Makes permanent all virtual pages starting from address base (inclusive) till
 * address base+size (exclusive). If the primitive execution succedes not more
 * page fault can happen for the addressed virtual pages (since there are
 * permanently stored in the physical memory) until the nonresident() is called.
 * This means that the primitive will also take care of loading into main memory
 * the missing pages. This might also be the reason of a failure if there is not
 * enough memory.
 *
 * @param  base  virtual pages starting address;
 * @param  size  virtual pages length;
 *
 * @return  the id to be given to the nonresident() primitive to undo the
 *          operation or 0xffffffff in case of failure.
 */
extern "C" natl resident(addr base, natq size);

/**
 * Reverts the operation performed using the resident() primitive.
 *
 * @param  id  the id of the operation to be undone.
 */
extern "C" void nonresident(natl id);

// EXTENSION 2017-02-07

// [...]
```

```
# [..]

# EXTENSION 2017-02-07

#-------------------------------------------------------------------------------
# User Primitives definitions.
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
.GLOBAL countres                                    # extern "C" natq countres();
#-------------------------------------------------------------------------------
countres:
    int $TIPO_CRES
    ret

#-------------------------------------------------------------------------------
.GLOBAL resident               # extern "C" natl resident(addr base, natq size);
#-------------------------------------------------------------------------------
resident:
    int $TIPO_RES
    ret

#-------------------------------------------------------------------------------
.GLOBAL nonresident                        # extern "C" void nonresident(natl id);
#-------------------------------------------------------------------------------
nonresident:
    int $TIPO_NONRES
    ret

# EXTENSION 2017-02-07

# [...]
```

```
# [...]

#------------------------------------------------------------------------
# Initialize IDT gates for the user primitives. This gates must have user level
# GLs as the User module must be able to use theese interrupts from the provided
# primitives.
#------------------------------------------------------------------------

# EXTENSION 2017-02-07

    # natq countres();
    carica_gate  TIPO_CRES    a_countres     LIV_UTENTE

    # void nonresident(natl id);
    carica_gate  TIPO_NONRES  a_nonresident  LIV_UTENTE

# EXTENSION 2017-02-07 )

# SOLUTION 2017-02-07

    # natl resident(addr base, natq size);
    carica_gate TIPO_RES      a_resident     LIV_UTENTE

# SOLUTION 2017-02-07 )

# [...]

# EXTENSION 2017-02-07

#------------------------------------------------------------------------
# IDT Subroutines definitions.
#------------------------------------------------------------------------

#------------------------------------------------------------------------
.EXTERN c_countres                                          # natq countres();
#------------------------------------------------------------------------
a_countres:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_countres                 # call C++ implementation
    iretq                           # return from interrupt
    .cfi_endproc

#------------------------------------------------------------------------
.EXTERN c_nonresident                               # void nonresident(natl id);
#------------------------------------------------------------------------
a_nonresident:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato                # save current process state
    call c_nonresident              # call C++ implementation
    call carica_stato               # load new process state
    iretq                           # return from interrupt
    .cfi_endproc

# EXTENSION 2017-02-07

# SOLUTION 2017-02-07

#------------------------------------------------------------------------
.extern c_resident                      # natl resident(addr base, natq size);
#------------------------------------------------------------------------
a_resident:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
```

```
    .cfi_offset rsp, -16
    call salva_stato                # save current process state
    cavallo_di_troia %rdi           # check addressed pages base
    cavallo_di_troia2 %rdi %rsi     # check addressed pages length
    call c_resident                 # call C++ implementation
    call carica_stato               # load new process state
    iretq                           # return from interrupt
    .cfi_endproc

# SOLUTION 2017-02-07 )

# [...]
```

```
// [...]

//////////////////////////////////////////////////////////////////////////////
//                          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

// [...]
```