Prova pratica di Calcolatori Elettronici

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

18 gennaio 2017

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

```
struct st1 { char vc[4]; }; struct st2 { int vd[4]; };
class cl
{       st1 s; long v[4];
public:
        cl(char c, st2 s2);
        void elab1(st1 s1, st2& s2);
        void stampa()
        {       int i;
                for (i=0;i<4;i++) cout << s.vc[i] << ' '; cout << endl;
                for (i=0;i<4;i++) cout << v[i] << ' '; cout << endl << endl;
        }
};
```

   Realizzare in Assembler GCC le funzioni membro seguenti.

```
cl::cl(char c, st2 s2)
{
        for (int i = 0; i < 4; i++) {
                s.vc[i] = c;
                v[i] = s2.vd[i] - s.vc[i];
        }
}
void cl::elab1(st1 s1, st2& s2)
{
        cl cla('f', s2);
        for (int i = 0; i < 4; i++) {
                if (s.vc[i] < s1.vc[i])
                        s.vc[i] = cla.s.vc[i];
                if (v[i] <= cla.v[i])
                        v[i] += cla.v[i];
        }
}
```

2. Introduciamo un meccanismo di *broadcast* tramite il quale un processo può inviare un messagio ad un insieme di processi. Per ricevere o inviare un broadcast i processi si devono preventivamente registrare come *listener* o *broadcaster*, rispettivamente. Un solo processo alla volta può essere registrato come broadcaster (un nuovo processo può diventare broadcaster solo quando il precedente termina).

Il sistema ricorda tutti i messaggi di broadcast inviati (fino ad un massimo dato dalla costante `MAX_BROADCAST`) e ciascun processo listener li riceve tutti, in ordine. I messaggi sono di tipo `natl`.

Per realizzare il sistema aggiungiamo il seguente tipo enumerato:

```
enum broadcast_role { B_NONE, B_BROADCASTER, B_LISTENER };
```

e il seguente campo ai descrittori di processo:

```
broadcast_role b_reg;
```

Il campo è posto a `B_NONE` alla creazione del processo.

Aggiungiamo infine le seguenti primitive:

- `void reg(broadcast_role role)` (tipo 0x3a, da realizzare): registra il processo per il ruolo dato da `role`; è un errore se role non specifica né un broadcaster, né un listener, se il processo era già registrato (per lo stesso o un altro ruolo), o se cè già un broadcaster e si tenta di registrarne un altro;
- `natl listen()` (tipo 0x3b, da realizzare): restituisce il prossimo messagio di broadcast non ancora letto dal processo; se il processo li ha già letti tutti, si blocca in attesa del prossimo; è un errore se il processo non è registrato come listener;
- `void broadcast(natl msg)` (tipo 0x3c, da realizzare): invia in broadcast il messaggio `msg`; è un errore se il processo non è registrato come broadcaster o se il limite di messaggi di broadcast è stato superato.

Le primitive abortiscono il processo chiamante in caso di errore e tengono conto della priorità tra i processi.

Modificare i file `sistema.cpp` e `sistema.S` in modo da realizzare le primitive mancanti. Attenzione: il candidato deve definire anche le necessarie strutture dati.

```
/**
 * 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 vc[4];
};

struct st2
{
    int vd[4];
};

class cl
{
    st1 s;
    long v[4];

public:
    cl(char c, st2 s2);

    void elab1(st1 s1, st2& s2);

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

        for (i = 0; i < 4; i++)
        {
            cout << v[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(char c, st2 s2)
{
    for (int i = 0; i < 4; i++)
    {
        s.vc[i] = c;
        v[i] = s2.vd[i] - s.vc[i];
    }
}

void cl::elab1(st1 s1, st2& s2)
{
    cl cla('f', s2);

    for (int i = 0; i < 4; i++)
    {
        if (s.vc[i] < s1.vc[i])
        {
            s.vc[i] = cla.s.vc[i];
        }

        if (v[i] <= cla.v[i])
        {
            v[i] += cla.v[i];
        }
    }
}
```

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

#-----------------------------------------------------------------------------
.TEXT
.GLOBAL _ZN2clC1Ec3st2                                  # cl::cl(char c, st2 s2)
#-----------------------------------------------------------------------------
# activation record:
# ------------------
#  i            -40
#  s2 [MSB]     -32
#  s2 [LSB]     -24
#  c            -9
#  &this        -8
#  %rbp         0
#-----------------------------------------------------------------------------
_ZN2clC1Ec3st2:
# set stack locations labels
    .set this, -8
    .set c,    -9
    .set s2,   -32
    .set i,    -40

# prologue: activation record
    pushq %rbp
    movq  %rsp, %rbp
    subq  $40, %rsp                 # reserve stack space for actual arguments

# copy actual arguments to the stack
    movq %rdi, this(%rbp)
    movb %sil, c(%rbp)
    movq %rdx, s2(%rbp)
    movq %rcx, s2+8(%rbp)

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

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

# for loop body
    movq   this(%rbp), %rdi         # &this -> %rdi
    movslq i(%rbp), %rcx            # i => %rcx
    movb   c(%rbp), %al             # c -> %al
    movb   %al, (%rdi, %rcx, 1)     # s.vc[i] = c;
    movb   (%rdi, %rcx, 1), %bl     # s.vc[i] -> %bl
    movsbl %bl, %ebx                # %bl => %ebx
    leaq   s2(%rbp), %rsi           # &s2 -> %rsi
    movl   (%rsi, %rcx, 4), %eax    # s2.vd[i] -> %eax
    subl   %ebx, %eax               # s2.vd[i] - s.vc[i] -> %eax
    movslq %eax, %rax               # %eax => %rax
    movq   %rax, 8(%rdi, %rcx, 8)   # v[i] = s2.vd[i] - s.vc[i];

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

finefor:

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

#-----------------------------------------------------------------------------
.GLOBAL _ZN2cl5elab1E3st1R3st2                # void cl::elab1(st1 s1, st2& s2)
#-----------------------------------------------------------------------------
```

```
# activation record:
# -----------------
#  i            -68
#  cla.s        -64
#  cla.v[0]     -56
#  cla.v[1]     -48
#  cla.v[2]     -40
#  cla.v[3]     -32
#  &s2          -24
#  s1           -16
#  &this        -8
#  %rbp         0
#-------------------------------------------------------------------------------
_ZN2cl5elab1E3st1R3st2:
# set stack locations labels
    .set this, -8
    .set s1,   -16
    .set s2,   -24
    .set cla,  -64
    .set i,    -68

# prologue: activation record
    pushq %rbp
    movq  %rsp, %rbp
    subq  $72, %rsp                 # reserve stack space for actual arguments

# copy actual arguments to the stack
    movq %rdi, this(%rbp)
    movl %esi, s1(%rbp)
    movq %rdx, s2(%rbp)

# cl cla('f', s2);
    leaq cla(%rbp), %rdi
    movb $'f', %sil
    movq s2(%rbp), %r8
    movq (%r8), %rdx
    movq 8(%r8), %rcx
    call _ZN2clC1Ec3st2

# for loop initialization
    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
    movslq i(%rbp), %rcx            # i => %rcx
    leaq   s1(%rbp), %rsi           # &s1 -> %rsi

# if (s.vc[i] < s1.vc[i])
    movb (%rsi, %rcx, 1), %al       # s1.vc[i] -> %al
    movb (%rdi, %rcx, 1), %bl       # s.vc[i] -> %bl
    cmpb %bl, %al                   # compare s.vc[i] and s1.vc[i]
    jle  fineif1                    # exit if (s1.vc[i] <= s.vc[i])
    movb cla(%rbp, %rcx, 1), %al    # cla.s.vc[i]; -> %al
    movb %al, (%rdi, %rcx, 1)       # s.vc[i] = cla.s.vc[i];

fineif1:

# if (v[i] <= cla.v[i])
    leaq cla(%rbp), %rsi            # &cla -> %rsi
    movq 8(%rsi, %rcx, 8), %rax     # cla.v[i] -> %rax
    movq 8(%rdi, %rcx, 8), %rbx     # v[i] -> %rbx
    cmpq %rax, %rbx                 # compare v[i] and cla.v[i]
    jg   fineif2                    # exit if (v[i] > cla.v[i])
    addq %rax, 8(%rdi, %rcx, 8)     # v[i] += cla.v[i];

fineif2:
```

```
    incl i(%rbp)
    jmp  for1

finefor1:

    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 = { 'e', 'b', 'f', 'd' };

    st2 sa = { 1, 20, 3, 40 };

    st2 sb = { 10, 2, 30, 4 };

    cl cla('a', sa);

    cla.stampa();

    cla.elab1(s1, sb);

    cla.stampa();
}
```

```
a a a a
-96 -77 -94 -57

f f f f
-188 -77 -166 -57
```

```
// EXTENSION 2017-01-18

/**
 * Maximum number of broadcast messages. Once this value is reached any attempt
 * of sending a broadcast message will result in the calling process being
 * aborted.
 */
#define MAX_BROADCAST           50

// EXTENSION 2017-01-18

// EXTENSION 2016-09-20

/**
 * Interrupts types definitions for user primitives for the broadcast mechanism.
 * In this implementation we will be taking one step further the one provided in
 * 2016-09-20_22.
 */
#define TIPO_R                  0x3a    // void reg(broadcast_role role)
#define TIPO_LS                 0x3b    // natl listen()
#define TIPO_B                  0x3c    // void broadcast(natl msg)

// EXTENSION 2016-09-20
```

```
// [...]

// EXTENSION 2016-09-20

/**
 * In this new implementation we will be taking the one provided in
 * 2016-09-20_22 one step further by introducing broadcasting roles
 * and allowing for multiple broadcast messages: the system will store
 * all broadcasting messages (which can be sent only by proceseses
 * registered to the global broadcast as broadcasters) until a maximum
 * value defined in costanti.h.
 */

/**
 * Broadcasting role: each process can register to the global system broadcast
 * either as a broadcaster or a listener. When a process is created in the
 * system module its role is set to B_NONE.
 */
extern "C" enum broadcast_role
{
    B_BROADCASTER = 1,
    B_LISTENER
};

/**
 * Registers the current process as a listener of the global broadcast with the
 * given role. The calling process must be aborted if the specified role is not
 * one between broadcaster or listener as well as if the process is already
 * registered to the global broadcast or there is already a broadcaster process
 * registered.
 *
 * @param  role  the broadcast role to be used for the process registrations.
 */
extern "C" void reg(enum broadcast_role role);

/**
 * Returns to the calling process the next broadcast message. If the process has
 * already retrieved all available broadcast messages it will be placed in the
 * listeners wait queue. All processes in this queue will be rescheduled when a
 * new broadcast message is sent.
 */
extern "C" natl listen();

/**
 * Sends the given broadcast message using the system global broadcast
 * descriptor. The calling process must be aborted if it is not the currentl
 * registered broadcaster.
 *
 * @param  msg  the broadcast message to be sent.
 */
extern "C" void broadcast(natl msg);

// EXTENSION 2016-09-20
```

```
# [...]

# EXTENSION 2016-09-20

##
# PRIMITIVES DEFINITIONS. Each primitive will only call the corresponding
# interrupt type and return. The interrupt will be handled in the system
# module by the subroutine loaded in the IDT. This calling mechanism allows for
# the user module to be able to interact with the system module (privileges
# escalation) while maintaining isolation.
##

#-------------------------------------------------------------------------------
.GLOBAL reg                                 # Primitive void reg() implementation
#-------------------------------------------------------------------------------
# Registers the calling process to the global system broadcast.
#-------------------------------------------------------------------------------
reg:
    int $TIPO_R
    ret

#-------------------------------------------------------------------------------
.GLOBAL listen                            # Primitive natl listen() implementation
#-------------------------------------------------------------------------------
# Retrieves the next broadcast message if there is any. If not the calling
# process will be placed in the global system broadcast wait queue.
#-------------------------------------------------------------------------------
listen:
    int $TIPO_LS
    ret

#-------------------------------------------------------------------------------
.GLOBAL broadcast            # Primitive void broadcast(natl msg) implementation
#-------------------------------------------------------------------------------
# Broadcasts the given message (type natl) using the system global broadcast.
#-------------------------------------------------------------------------------
broadcast:
    int $TIPO_B
    ret

# EXTENSION 2016-09-20
```

```
# [...]

# SOLUTION 2017-01-18

#-----------------------------------------------------------------------------
# Load IDT entries.
#-----------------------------------------------------------------------------
    carica_gate TIPO_R          a_reg            LIV_UTENTE
        carica_gate     TIPO_LS         a_listen         LIV_UTENTE
        carica_gate     TIPO_B          a_broadcast      LIV_UTENTE

# SOLUTION 2017-01-18

# [...]

# SOLUTION 2017-01-18

#-----------------------------------------------------------------------------
# IDT entries subroutines definitions.
#-----------------------------------------------------------------------------
# Registers the calling process as either a listener or a broadcaster. One of
# these roles must be specified and one and only one process can be registered
# as broadcaster.
#-----------------------------------------------------------------------------
.GLOBAL a_reg
#-----------------------------------------------------------------------------
a_reg:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_reg
    iretq
    .cfi_endproc

#-----------------------------------------------------------------------------
.GLOBAL a_listen
#-----------------------------------------------------------------------------
# The listen() primitive will hang the calling process if all messages have
# already been delivered until the next broadcast message is sent. At the of the
# C++ implementation c_listen() the calling process is placed in the global
# broadcast descriptor listeners queue if its b_id (last retrieved broadcast
# message id) is equal to the system broadcast last_id and the scheduler
# is called. This is why we have to save the current process state (salva_stato)
# and load a new process (carica_stato).
#-----------------------------------------------------------------------------
a_listen:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_listen
    call carica_stato
    iretq
    .cfi_endproc

#-----------------------------------------------------------------------------
.GLOBAL a_broadcast
#-----------------------------------------------------------------------------
# The broadcast() primitive will move the calling process to the system ready
# processes queue after delivering the broadcast message to the available
# listeners. At the end of the C++ implementation a new process is scheduled.
# That’s why we need to save the current process (broadcaster) process and load
# a new process state (the scheduler is called at the end of the C++
# implementation).
#-----------------------------------------------------------------------------
a_broadcast:
    .cfi_startproc
    .cfi_def_cfa_offset 40
```

```
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_broadcast
    call carica_stato
    iretq
    .cfi_endproc

# SOLUTION 2016-09-20
```

```
// sistema.cpp
//
#include "costanti.h"
#include "libce.h"

//////////////////////////////////////////////////////////////////////////////
//                      PROCESSI                                            //
//////////////////////////////////////////////////////////////////////////////
const natl MAX_PRIORITY = 0xfffffff;
const natl MIN_PRIORITY = 0x0000001;
const natl DUMMY_PRIORITY = 0x0000000;
const int N_REG = 16;   // numero di registri nel campo contesto


// EXTENSION 2017-01-18

/**
 * Available broadcaste roles.
 */
enum broadcast_role
{
    B_NONE,         // no role is assigned when the process is created
    B_BROADCASTER,  // broadcaster (can use the broadcast() primitive)
    B_LISTENER      // listener (can use the listen() primitive)
};

// EXTENSION 2017-01-18

// si veda in PAGINAZIONE per il significato di questi typedef
typedef natq vaddr;
typedef natq faddr;
typedef natq tab_entry;

// descrittore di processo
struct des_proc {
        // parte richiesta dall'hardware
        struct __attribute__ ((packed)) {
                natl riservato1;
                vaddr punt_nucleo;
                // due quad  a disposizione (puntatori alle pile ring 1 e 2)
                natq disp1[2];
                natq riservato2;
                //entry della IST, non usata
                natq disp2[7];
                natq riservato3;
                natw riservato4;
                natw iomap_base; // si veda crea_processo()
        };
        //finiti i campi obbligatori
        faddr cr3;
        natq contesto[N_REG];
        natl cpl;

// EXTENSION 2017-01-18

    // process broadcast role
        broadcast_role b_reg;

// EXTENSION 2017-01-18

// SOLUTION 2017-01-18

    // process last retrieved broadcast message id
    natl b_id;

// SOLUTION 2017-01-18
};

// [...]
```

```
/**
 * Broadcast descriptor struct.
 */
struct broadcast
{
    // true if the broadcaster is registered
    bool broadcaster_registered;

// ( SOLUZIONE 2017-01-18

    // last broadcast message id
    natl last_id;

    // sent broadcast messages array
    natl msg[MAX_BROADCAST];

    // registered listeners array
    proc_elem *listeners;

//   SOLUZIONE 2017-01-18 )
};

/**
 * System global broadcast descriptor.
 */
broadcast global_broadcast;

/**
 * Initializes the global broadcast descriptor.
 */
void broadcast_init()
{
    // no initial broadcaster registered
    global_broadcast.broadcaster_registered = false;
// ( SOLUZIONE 2017-01-18

    // no broadcast messages registered at initialization
    global_broadcast.last_id = 0;

    // no initial listeners registered
    global_broadcast.listeners = 0;

//   SOLUZIONE 2017-01-18 )
}

// ( SOLUZIONE 2016-09-20

/**
 * Registers a process to the global broadcast with the specified broadcast
 * role. If the given role is not valid (!B_BROADCASTER and !B_LISTENER) or
 * if the given process is already registered, or if there is already a
 * broadcaster registered the current process must be aborted.
 *
 * @param  role  the of the process being registered (either B_BROADCASTER or
 *               B_LISTENER).
 */
extern "C" void c_reg(enum broadcast_role role)
{
    // retrieve calling process descriptor
    struct des_proc *p = des_p(esecuzione->id);

    // retrieve global broadcast descriptor
    struct broadcast *b = &global_broadcast;

    // check if the given broadcast role is valid: B_NONE is invalid
    if (role != B_BROADCASTER && role != B_LISTENER)
    {
        // print warning log message
        flog(LOG_WARN, "Invalid broadcast role: %d", role);
```

```
        // abort calling process
        c_abort_p();

        // just return
        return;
        }

    // check if the process is already registered to the global broadcast
    if (p->b_reg != B_NONE)
    {
        // print warning log message
        flog(LOG_WARN, "Process already registered as %s",
             (p->b_reg == B_BROADCASTER ? "broadcaster." : "listener."));

        // abort current process under execution
        c_abort_p();

        // just return
        return;
        }

    // check if the given role is broadcaster
    if (role == B_BROADCASTER)
    {
        // check if there is already a registered broadcaster
        if (b->broadcaster_registered)
        {
            // if so, print a warning log message
            flog(LOG_WARN, "Broadcaster already registered.");

            // abort current process under execution
            c_abort_p();

            // just return
            return;
        }

        // set broadcaster registered to true
        b->broadcaster_registered = true;
    }

    // update process broadcast role
    p->b_reg = role;
}

/**
 * Called by listener processes to retrieve the next broadcast message. If the
 * process has already received all the broadcast messages it must be placed in
 * the wait queue for the next broadcast message. If the process is not a
 * registered listener it must be aborted.
 */
extern "C" void c_listen()
{
    // retrieve calling process descriptor
    struct des_proc *p = des_p(esecuzione->id);

    // retrieve global broadcast descriptor
    struct broadcast *b = &global_broadcast;

    // check if the current process is not a registered listener
    if (p->b_reg != B_LISTENER)
    {
        // if so, print a warning log message
        flog(LOG_WARN, "Process not registered as listener.");

        // abort calling process
        c_abort_p();

        // just return to the caller
        return;
```

```
    }

    // check if there are broadcast messages to be retrieved
    if (p->b_id < b->last_id)
    {
        // if so, retrieve they nex broadcast message
        p->contesto[I_RAX] = b->msg[p->b_id];

        // increase last retrieved broadcast message id
        p->b_id++;

        // just return to the caller
        return;
    }

    // otherwise, insert the current process in the listeners processes queue:
    // it will have to wait until another broadcast message is sent by the
    // broadcaster process in which case it will receive the broadcast message
    // and be placed in the system ready processes queue and eventually
    // rescheduled
    inserimento_lista(b->listeners, esecuzione);

    // schedule a new process
    schedulatore();
}

/**
 * Sends the given broadcast message. It must check if the calling process is
 * registered as broadcaster and if the maximum number of broadcast messages is
 * not exceeded. If both conditions are not met the calling processes is
 * aborted.
 *
 * @param  msg  the broadcast message to be sent.
 */
extern "C" void c_broadcast(natl msg)
{
    // retrieve current process descriptor
    struct des_proc *p = des_p(esecuzione->id);

    // retrieve global broadcast descriptor
    struct broadcast *b = &global_broadcast;

    // check if the current process is registered as broadcaster
    if (p->b_reg != B_BROADCASTER)
    {
        // if not, print a warning log message
        flog(LOG_WARN, "Broadcast message from invalid process.");

        // abort current process under execution
        c_abort_p();

        // just return
        return;
    }

    // check if the number of maximum broadcast messages has been reached
    if (b->last_id >= MAX_BROADCAST)
    {
        // if so, print a warning log message
        flog(LOG_WARN, "Too many broadcast messages.");

        // abort the current process under execution
        c_abort_p();

        // just retun
        return;
    }

    // set broadcast message
    b->msg[b->last_id] = msg;
```

```
    // increase last broadcast message id
    b->last_id++;

    // insert the current process at the top of the ready processes queue
    inspronti();

    // deliver the new broadcast message to all listeners in the wait queue:
    // theese processes have already retrieved all previous broadcast messages
    // and called the listen() primitive one more time which resulted for them
    // beign placed in the global broadcaster descriptor listeners wait queue
    while (b->listeners)
    {
        // process descriptor
        struct proc_elem *work;

        // extract top indexed listener process
        rimozione_lista(b->listeners, work);

        // retrieve process descriptor
        struct des_proc *w = des_p(work->id);

        // deliver broadcast message to the listener process
        w->contesto[I_RAX] = msg;

        // increase listener process broadcast messages last id
        w->b_id++;

        // insert the listener process in the system ready processes queue
        inserimento_lista(pronti, work);
    }

    // schedule a new process
    schedulatore();
}

// SOLUTION 2016-09-20

// [...]

/**
 * In this new implementation of the broadcast system there can be multiple
 * broadcaster processes. However, only one process can be active with the role
 * of broadcaster. When each process is destroyed we have to check if it is the
 * broadcaster process and in that case remove the broadcaster from the global
 * broadcast descriptor.
 */
void distruggi_processo(proc_elem* p)
{
        des_proc* pdes_proc = des_p(p->id);

// EXTENSION 2016-09-20

    // check if the process is a broadcaster
    if (pdes_proc->b_reg == B_BROADCASTER)
    {
        // if so, remove the global broadcast broadcaster
        global_broadcast.broadcaster_registered = false;
    }

// EXTENSION 2016-09-20

        faddr tab4 = pdes_proc->cr3;
        riassegna_tutto(p->id, tab4, I_MIO_C, N_MIO_C);
        riassegna_tutto(p->id, tab4, I_UTN_C, N_UTN_C);
        rilascia_tutto(tab4, I_UTN_P, N_UTN_P);
        ultimo_terminato = tab4;
        if (p != esecuzione) {
                distruggi_pila_precedente();
        }
```

```
        rilascia_tss(id_to_tss(p->id));
        dealloca(pdes_proc);
}

// [...]

void main_sistema(int n)
{
        natl sync_io;

        // ( caricamento delle tabelle e pagine residenti degli spazi condivisi ()
        flog(LOG_INFO, "creazione o lettura delle tabelle e pagine residenti condivise...
");
        if (!crea_spazio_condiviso())
                goto error;
        // )

        gdb_breakpoint();

        // ( inizializzazione del modulo di io
        flog(LOG_INFO, "creazione del processo main I/O...");
        sync_io = sem_ini(0);
        if (sync_io == 0xFFFFFFFF) {
                flog(LOG_ERR, "Impossibile allocare il semaforo di sincr per l'IO");
                goto error;
        }
        // occupiamo l'entrata del timer
        aggiungi_pe(ESTERN_BUSY, 2);
        if (activate_p(swap_dev.sb.io_entry, sync_io, MAX_PRIORITY, LIV_SISTEMA) == 0xFFF
FFFFF) {
                flog(LOG_ERR, "impossibile creare il processo main I/O");
                goto error;
        }
        flog(LOG_INFO, "attendo inizializzazione modulo I/O...");
        sem_wait(sync_io);
        // )

        // ( creazione del processo start_utente
        flog(LOG_INFO, "creazione del processo start_utente...");
        if (activate_p(swap_dev.sb.user_entry, 0, MAX_PRIORITY, LIV_UTENTE) == 0xFFFFFFFF
) {
                flog(LOG_ERR, "impossibile creare il processo main utente");
                goto error;
        }
        // )
        // (* attiviamo il timer
        attiva_timer(DELAY);
        flog(LOG_INFO, "attivato timer (DELAY=%d)", DELAY);
        // *)

// ( ESAME 2017-01-18

    // initialize global broadcast descriptor
        broadcast_init();

//   ESAME 2017-01-18 )

        // ( terminazione
        flog(LOG_INFO, "passo il controllo al processo utente...");
        terminate_p();
        // )
error:
        panic("Errore di inizializzazione");
}
```