# Prova pratica di Calcolatori Elettronici

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

24 luglio 2019

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

```
struct st {
        long vv2[4];
        char vv1[4];
};
class cl {
        st s;
public:
        cl(char v[]);
        void elab1(st& ss, int d);
        void stampa()
        {
                for (int i = 0; i < 4; i++)
                        cout << (int)s.vv1[i] << ' ';
                cout << '\t';
                for (int i = 0; i < 4; i++)
                        cout << s.vv2[i] << ' ';
                cout << endl;
                cout << endl;
        }
};
```

   Realizzare in Assembler GCC le funzioni membro seguenti.

```
cl::cl(char v[])
{
        for (int i = 0; i < 4; i++) {
                s.vv1[i] = s.vv2[i] = v[3 - i];
        }
}
void cl::elab1(st& ss, int d)
{
        for (int i = 0; i < 4; i++) {
                if (d >= ss.vv2[i])
                        s.vv1[i] += ss.vv1[i];
                s.vv2[i] = d - i;
        }
}
```

2. Vogliamo fornire ai processi la possibilità di bloccare l'esecuzione di un processo a scelta, quando questo passa da una certa istruzione. Per far questo forniamo alcune primitive. Con la primitiva

`bpattach(natl id, vaddr rip)` un processo *master* installa un breakpoint (istruzione `int3`, codice operativo `0xCC`) all'indirizzo `rip` per il processo `id`, che diventa *slave*. Da quel momento in poi il processo slave si blocca se passa da `rip`. Nel frattempo, usando la primitiva `bpwait()`, il processo master può sospendersi in attesa che lo slave passi dal breakpoint. Infine, con la primitiva `bpremove()`, il processo master rimuove il breakpoint e risveglia eventualmente lo slave, il quale prosegue la sua esecuzione come se non fosse mai stato intercettato.

Si noti che processi che non sono slave non devono essere intercettati. Inoltre, se un processo esegue `int3` senza che ciò sia stato richiesto da un master, il processo deve essere abortito.

Aggiungiamo i seguenti campi ai descrittori di processo:

```
des_proc *bp_master;
des_proc *bp_slave;
vaddr bp_addr;
natb bp_orig;
natl bp_slave_id;
struct proc_elem  *bp_waiting;
```

dove: `bp_master` punta al processo master di questo processo (nullo se il processo non ha un master); `bp_slave` punta al processo slave di questo processo (nullo se il processo non ha uno slave); `bp_addr`, `bp_orig` e `bp_slave_id` sono sinificativi solo per il processi master e contengono, rispettivamente, l'indirizzo a cui è installato breakpoint; il byte originariamente contenuto a quell'indirizzo e l'id dello slave; `bp_waiting` è una coda su cui i processi master e slave si possono bloccare: il master su quella dello slave e viceversa.

Si modifichino i file `sistema/sistema.s` e `sistema/sistema.cpp` per implementare le seguenti primitive (abortiscono il processo in caso di errore):

- `bool bpattach(natl id, vaddr rip)`: (tipo 0x59, realizzata in parte): se il processo `id` esiste e non è già uno slave o un master, installa il breakpoint all'indirizzo `rip` e restituisce `true`, altrimenti restituisce `false`; è un errore se `rip` non appartiene all'intervallo [`ini_utn_c`, `fin_utn_c`) (zona utente/condivisa) o se il processo cerca di diventare master di se stesso.
- `natl bpwait()`: (tipo 0x5a, già realizzata): attende che il processo slave passi dal breakpoint; è un errore invocare questa primitiva se il processo non è master;
- `void bpdetach()` (tipo 0x5b, da realizzare): disfa tutto ciò che ha fatto la `bpattach()` e risveglia eventualmente il processo slave; è un errore invocare questa primitiva se il processo non è master;

**Attenzione**: bisogna fare in modo che solo i processi slave vengano intercettati, ma la parte utente/condivisa è appunto condivisa tra tutti i processi.

```
/**
 * File: cc.h
 *       Contains the declaration for the data structures used in the exercise.
 *
 * Author: Rambod Rahmani <rambodrahmani@autistici.org>
 *         Created on 14/09/2019.
 */

#include <iostream>

using namespace std;

struct st
{
    long vv2[4];
    char vv1[4];
};

class cl
{
    st s;

public:
    cl(char v[]);

    void elab1(st& ss, int d);

    void stampa()
    {
        for (int i = 0; i < 4; i++)
        {
            cout << (int)s.vv1[i] << ' ';
        }
        cout << '\t';

        for (int i = 0; i < 4; i++)
        {
            cout << s.vv2[i] << ' ';
        }

        cout << endl;
        cout << endl;
    }
};
```

```
/**
 * File: es1.cpp
 *       Contains the C++ code to be translated into Assembly (es1.s file).
 *
 * Author: Rambod Rahmani <rambodrahmani@autistici.org>
 *         Created on 14/09/2019.
 */

#include "cc.h"

cl::cl(char v[])
{
    for (int i = 0; i < 4; i++)
    {
        s.vv1[i] = s.vv2[i] = v[3 - i];
    }
}

void cl::elab1(st& ss, int d)
{
    for (int i = 0; i < 4; i++)
    {
        if (d >= ss.vv2[i])
        {
            s.vv1[i] += ss.vv1[i];
        }

        s.vv2[i] = d - i;
    }
}
```

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

#-----------------------------------------------------------------------------
.TEXT
.GLOBAL _ZN2clC1EPc                                          # cl::cl(char v[])
#-----------------------------------------------------------------------------
# activation record:
# ------------------
#  i            -20
#  &v           -16
#  this         -8
#  %rbp         0
#-----------------------------------------------------------------------------
_ZN2clC1EPc:
# set stack locations labels
    .set this, -8
    .set v,    -16
    .set i,    -20

# prologue: activation frame
    pushq %rbp
    movq  %rsp, %rbp
    subq  $24, %rsp                 # reserve stack space for actual arguments

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

# for 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
    movq   v(%rbp), %rsi            # &v -> %rsi
    movslq i(%rbp), %rcx            # i => %rcx
    movq   $3, %r8                  # 3 -> %r8
    subq   %rcx, %r8                # 3 - i -> %r8
    movb   (%rsi, %r8, 1), %al      # v[3 - i] -> %al
    movsbq %al, %rax                # v[3 - i] => %rax
    movq   %rax, (%rdi, %rcx, 8)    # s.vv2[i] = v[3 - i];
    movq   %rax, 32(%rdi, %rcx, 1)  # s.vv1[i] = v[3 - i];

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

finefor:

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

#-----------------------------------------------------------------------------
.GLOBAL _ZN2cl5elab1ER2sti
#-----------------------------------------------------------------------------
# activation record:
# ------------------
#  i            -24
#  d            -20
#  &ss          -16
#  this         -8
#  %rbp         0
```

```
#------------------------------------------------------------------------------
_ZN2cl5elab1ER2sti:
# set stack locations labels
    .set this, -8
    .set ss,   -16
    .set d,    -20
    .set i,    -24

# prologue: activation record
    pushq %rbp
    movq  %rsp, %rbp
    subq  $24, %rsp                 # reserve stack space for actual arguments

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

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

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

# for loop body
    movq   this(%rbp), %rdi         # &this -> %rdi
    movq   ss(%rbp), %rsi           # &ss -> 5rsi
    movslq d(%rbp), %rdx            # d => %rdx
    movslq i(%rbp), %rcx            # i => %rcx

# if (d >= ss.vv2[i])
    movq  (%rsi, %rcx, 8), %rax     # ss.vv2[i] -> %rax
    cmpq  %rax, %rdx                # compare d and ss.vv2[i]
    jl    fineif                    # exit if (d < ss.vv2[i])
    movb  32(%rsi, %rcx, 1), %bl    # ss.vv1[i] -> %bl
    addb  %bl, 32(%rdi, %rcx, 1)    # s.vv1[i] += ss.vv1[i];

fineif:

    subq %rcx, %rdx                 # d - i -> %rdx
    movq %rdx, (%rdi, %rcx, 8)      # s.vv2[i] = d - i;

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

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 14/09/2019.
 */

#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[])
{
    st s = { 1, 2, 3, 4, 1, 2, 3, 4 };

    char v[4] = { 10, 11, 12, 13 };

    int d = 2;

    cl cc1(v);

    cc1.stampa();

    cc1.elab1(s, d);

    cc1.stampa();
}
```

```
13 12 11 10     13 12 11 10

14 14 11 10     2 1 0 -1
```

```
// [...]

// EXTENSION 2019-07-24

/**
 * Primitives Interrupt Types declaration for the User Module.
 */

/**
 * extern "C" bool bpattach(natl id, vaddr rip);
 */
#define  TIPO_BPA  0x59

/**
 * extern "C" void bpwait();
 */
#define  TIPO_BPW  0x5a

/**
 * extern "C" void bpdetach();
 */
#define  TIPO_BPD  0x5b

// EXTENSION 2019-07-24

// [...]
```

```
// [...]

// EXTENSION 2019-07-24

/**
 *
 */
typedef natq vaddr;

/**
 * Primitives declarations for the User Module: we will be providing to the
 * processes the ability to pause a certain process when it reaches a certain
 * instruction.
 */

/**
 * A master process can use to primitive to set a breakpoint (assembly
 * instruction int3, opcode 0xCC) at the given instruction address for the slave
 * process having the given id. If a process executes the int3 instruction
 * without being a slave it must be aborted.
 *
 * The specified process must not be a slave or a master process itself.
 *
 * In case of error the calling process must be aborted.
 *
 * @param  id   id of the slave process;
 * @param  rip  process instruction address: keep in mind that it must be in the
 *              interval defined by [ini_utn_c, fin_utn_c);
 *
 * @return  true if the breakpoint is successfully installed, false otherwise.
 */
extern "C" bool bpattach(natl id, vaddr rip);

/**
 * The master process can suspend its execution and wait until the breakpoint in
 * the slave process is reached. The calling process must be a master process.
 *
 * In case of error the calling process must be aborted.
 */
extern "C" void bpwait();

/**
 * Undoes the operation done by bpattach() and reschedules the slave process.
 * The calling process must be a master process.
 *
 * In case of error the calling process must be aborted.
 */
extern "C" void bpdetach();

// EXTENSION 2019-07-24

// [...]
```

```
# [...]

# EXTENSION 2019-07-24

#--------------------------------------------------------------------------------
.GLOBAL bpattach
#--------------------------------------------------------------------------------
#                Implementation for extern "C" bool bpattach(natl id, vaddr rip)
#--------------------------------------------------------------------------------
bpattach:
    .cfi_startproc
    int $TIPO_BPA
    ret
    .cfi_endproc

#--------------------------------------------------------------------------------
.GLOBAL bpwait                      # Implementation for extern "C" void bpwait()
#--------------------------------------------------------------------------------
bpwait:
    .cfi_startproc
    int $TIPO_BPW
    ret
    .cfi_endproc

#--------------------------------------------------------------------------------
.GLOBAL bpdetach                  # Implementation for extern "C" void bpdetach()
#--------------------------------------------------------------------------------
bpdetach:
    .cfi_startproc
    int $TIPO_BPD
    ret
    .cfi_endproc

# EXTENSION 2019-07-24
```

```
# [...]

# EXTENSION 2019-07-24

    #-------------------------------------------------------------------------
    # Decrease breakpoint IDT entry DPL to USER_LEVEL in order for it to be
    # called from the User Module processes.
    #-------------------------------------------------------------------------
    carica_gate 3       breakpoint  LIV_UTENTE

# EXTENSION 2019-07-24

# [...]

# EXTENSION 2019-07-24

    # initialize bpattach IDT gate subroutine
    carica_gate  TIPO_BPA  a_bpattach  LIV_UTENTE

    # initialize bpwait IDT gate subroutine
    carica_gate  TIPO_BPW  a_bpwait    LIV_UTENTE

# EXTENSION 2019-07-24

# SOLUTION 2019-07-24

    # initialize bpdetach IDT gate subroutine
    carica_gate  TIPO_BPD  a_bpdetach  LIV_UTENTE

# SOLUTION 2019-07-24

# [...]

# EXTENSION 2019-07-24

#-------------------------------------------------------------------------------
# IDT gates subroutines assembly deiniftions.
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
#-------------------------------------------------------------------------------
a_bpattach:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato                    # save current process state
    call c_bpattach                     # call C++ subroutine implementation
    call carica_stato                   # load new process state
    iretq                               # return from interrupt
    .cfi_endproc

#-------------------------------------------------------------------------------
#-------------------------------------------------------------------------------
a_bpwait:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato                    # save current process state
    call c_bpwait                       # call C++ subroutine implementation
    call carica_stato                   # load new process state
    iretq                               # return from interrupt
    .cfi_endproc

# EXTENSION 2019-07-24

# SOLUTION 2019-07-24

#-------------------------------------------------------------------------------
```

```
#-----------------------------------------------------------------------------
a_bpdetach:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato                    # save current process state
    call c_bpdetach                     # call C++ subroutine implementation
    call carica_stato                   # load new process state
    iretq                               # return from interrupt
    .cfi_endproc

# SOLUTION 2019-07-24

# [...]

#-----------------------------------------------------------------------------
# Interrupt 3 - Breakpoint exception.
# Sets the parameters for and calls the C++ exception handler. A new process is
# scheduled in the C++ implementation.
#-----------------------------------------------------------------------------
breakpoint:
        .cfi_startproc
        .cfi_def_cfa_offset 40
        .cfi_offset rip, -40
        .cfi_offset rsp, -16
        call salva_stato                # save current process state

# SOLUTION 2019-07-24

    movq $3, %rdi                   # exception type
    movq $0, %rsi                   # exception error
    movq %rsp, %rdx                 # %address contained in rsp
    call c_breakpoint_exception     # call C++ exception handler

# SOLUTION 2019-07-24

        call carica_stato               # load new process state
        iretq                           # return from interrupt.
        .cfi_endproc

# [...]
```

```
/**
 * File: system.cpp
 *       System Module C++ implementation.
 *
 * Author: Rambod Rahmani <rambodrahmani@autistici.org>
 *         Created on 30/08/2019.
 */

#include "constants.h"
#include <libqlk.h>
#include <log.h>
#include <apic.h>

////////////////////////////////////////////////////////////////////////////////
//                                  PROCESSES                                 //
////////////////////////////////////////////////////////////////////////////////

/**
 * Maximum process priority.
 */
const natl MAX_PRIORITY = 0xfffffff;

/**
 * Minimum process priority.
 */
const natl MIN_PRIORITY = 0x0000001;

/**
 * Dummy processo priority.
 */
const natl DUMMY_PRIORITY = 0x0000000;

/**
 * Number of registers of the contest array field of the des_proc struct.
 */
const int N_REG = 16;

/**
 * Memory Virtual Address.
 */
typedef natq vaddr;

/**
 * Memory Physical Address.
 */
typedef natq faddr;

/**
 *
 */
typedef natq tab_entry;

/**
 * Process Descriptor. Each process has its own process descriptor, a system
 * stack and its own memory (which contains its code, data and user stack).
 * In order to be able to switch between processes and allow for a little
 * parallel execution we will have to take a full snapshot of the system state
 * (CPU, memory, devices etc..) in order to be able to come back to the
 * execution where it has been left.
 */
struct des_proc
{
    // hardware required
    struct __attribute__ ((packed))
    {
        natl riservato1;

        /**
         * Each process has its own system stack and the way the system stack is
         * changed moving from one process to another is up to the hardware
```

```
         * interrupt mechanism. We will place this pointer to the process system
         * stack where we know the hardware will look for it. When the process
         * is started, the CPU will save in this stack the pointer to the
         * previous stack, the content of the RFLAGS register, the previous
         * privilege level, and the address of the next instruction to be
         * executed.
         * When a process is at user level its system stack is always empty. The
         * system stack will be filled when moving to the system level and
         * emptied out when returning to user level.
         */
        vaddr system_stack;

        // 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()
    };

    // custom data
    faddr cr3;

    // process context: contains a copy of the CPU registers content
    natq context[N_REG];

    natl cpl;

/**
 * New fields must be added to the process descriptor in order to be able to
 * distinguish between master and slave process.
 */

// EXTENSION 2019-07-24

    /**
     * Pointer to the master process. Might be null if no master has been
     * defined for this process.
     */
    des_proc *bp_master;

    /**
     * Pointer to the slave process. Might be null if no slave has been defined
     * for this process.
     */
    des_proc *bp_slave;

    /**
     * Breakpoint instruction address. Meaningful only for a master process.
     */
    vaddr bp_addr;

    /**
     * Original byte contained at the address where the breakpoint has been
     * placed. Meaningful only for a master process.
     */
    natb bp_orig;

    /**
     * Slave process ID. Meaningful only for a master process.
     */
    natl bp_slave_id;

    /**
     * Process queue which can be used by the master or the slave process to
     * wait.
     */
```

```
    struct proc_elem  *bp_waiting;

// EXTENSION 2019-07-24
};

// [...]

// EXTENSION 2019-07-24

/**
 * Can be used only by master processes to be palced in the corresponding slave
 * process wait queue for them to reach the breakpoint address. Theese master
 * processes must have a bp_slave process set in their descriptor.
 */
extern "C" void c_bpwait()
{
    // retrieve calling process descriptor
    des_proc *self = des_p(esecuzione->id);

    // check if the calling process is a master process
    if (!self->bp_slave)
    {
        // if so,  print a warning log message
        flog(LOG_WARN, "Only master processes can use the bpwait() primitive.");

        // abort calling process: it must be a master process to use this
        // primitive
        c_abort_p();
    }

    // check if there is a slave process waiting
    if (!self->bp_waiting)
    {
        // if not, insert the current process in the slave process waiting queue
        self->bp_slave->bp_waiting = esecuzione;

        // schedule a new process
        schedulatore();
    }
}

/**
 * This subroutine is called in case of a breakpoint exception.
 *
 * @param  tipo    exception type (3);
 * @param  errore  exception error (0);
 * @param  p_rip   address contained in %rsp.
 */
extern "C" void c_breakpoint_exception(int tipo, natq errore, vaddr *p_rip)
{
    // retrieve calling process descriptor
    des_proc *self = des_p(esecuzione->id);

    // check if the calling process is a slave process: this breakpoint was
    // added using the bpattach() primitive
    if (!self->bp_master)
    {
        // if not, just handle the exception: the gestore_eccezioni will also
        // abort the calling process as all interrupt 3 not placed using the
        // bpattach must be aborted
        gestore_eccezioni(tipo, errore, *p_rip);

        // and return to the caller
        return;
    }

    // retrieve current rsp value
    natq* rip = reinterpret_cast<natq*>(p_rip);

    // decrease it by one
```

```
    (*rip)--;

    // insert the current slave process in the master process bp queue
    self->bp_master->bp_waiting = esecuzione;

    // check if the master process is in this slave process bp queue
    if (self->bp_waiting)
    {
        // if so, move the process in the system ready processes queue
        inserimento_lista(pronti, self->bp_waiting);

        // clear waiting master processes for this slave process
        self->bp_waiting = 0;
    }

    // schedule a new process
    schedulatore();
}

// normalmente, le parti condivise della memoria virtuale (come quelle che
// contengono il codice dei processi) sono realizzate condividendo tutto a
// partire dalle tabelle di livello 3.  Questa funzione installa nell'albero di
// traduzione del processo id una copia privata del percorso di traduzione e
// della pagina che contiene l'indirizzo v, creando, copiando e modificando
// opportunamente le tabelle di livello 3, 2 e 1 e la pagina finale.
bool duplica(natl id, vaddr v)
{
        // per ogni livello a partire dal 3 andando a scendere...
        for (int i = 3; i >= 0; i--) {
                // prendiamo l'entrata della tabella di livello superiore
                // che punta all'entita' che stiamo considerando
                tab_entry& e = get_des(id, i + 1, v);

                // allochiamo un frame per la copia
                des_frame *df_dst = alloca_frame(id, i, v);
                if (!df_dst) {
                        flog(LOG_WARN, "memoria esaurita");
                        return false;
                }
                // riempiamo i campi del descrittore di frame
                // (per consistenza e debugging)
                df_dst->processo = id;
                df_dst->residente = true;
                df_dst->livello = i;
                df_dst->ind_virtuale = v;
                df_dst->ind_massa = 0;
                df_dst->contatore = 0;

                // copiamo l'entita' vecchia nel nuovo frame
                faddr src = extr_IND_FISICO(e);
                faddr dst = indirizzo_frame(df_dst);
                memcpy(reinterpret_cast<void *>(dst), reinterpret_cast<void *>(src), 4096
);
                // facciamo in modo che la tabella di livello superiore punti alla copia
                set_IND_FISICO(e, dst);
        }
        return true;
}

/**
 * Attaches a breakpoint in the processes having the given id at the instruction
 * having the given address.
 *
 * @param  id   slave destination process id;
 * @param  rip  instruction address.
 */
extern "C" void c_bpattach(natl id, vaddr rip)
{
    // retrieve calling process descriptor (the master one)
    des_proc *self = des_p(esecuzione->id);
```

```
    // retrieve slave process id
    des_proc *dest = des_p(id);

    // check if the calling process (master) and the slave process are the same
    // process
    if (dest == self)
    {
        // if so, print a warning log message
        flog(LOG_WARN, "A master process can not call the bpattach() on itself.");

        // abort calling process
        c_abort_p();

        // just return to the caller
        return;
    }

    // chec if the given instruction address belongs to the user process shared
    // memory area
    if (rip < ini_utn_c || rip >= fin_utn_c)
    {
        // if not, print a warning log message
        flog(LOG_WARN, "rip %p out of bounds [%p, %p)", rip, ini_utn_p, fin_utn_p);

        // abort calling process
        c_abort_p();

        // return to the caller
        return;
    }

    // set return value
    self->contesto[I_RAX] = false;

    // check if the slave process is valid, if it does not have a master process
    // already set nor a slave process
    if (!dest || dest->bp_master || dest->bp_slave)
    {
        // otherwise, just return to the caller
        return;
    }

    // set slave process for the calling master process
    self->bp_slave = dest;

    // no slave processes waiting in the master bp queue
    self->bp_waiting = 0;

    // set slave process id for the calling master process
    self->bp_slave_id = id;

    // set master process for the destination slave process
    dest->bp_master = self;

    // no master processes waiting in the slave bp queue
    dest->bp_waiting = 0;

    // update return value
    self->contesto[I_RAX] = true;

// SOLUTION 2019-07-24

    // create private memory area for the given instruction address
    if (!duplica(id, rip))
    {
        // in case of error, just return to the caller
        return;
    }
```

```
    // retrieve byte pointed by the given address
    natb *bytes = reinterpret_cast<natb*>(trasforma(id, rip));

    // set breakpoint address
    self->bp_addr = rip;

    // save original instruction byte
    self->bp_orig = *bytes;

    // replace addressed byte with the int3 opcode byte
    *bytes = 0xCC;

//   SOLUZIONE 2019-07-24 )
}
//   ESAME 2019-07-24 )

// ( SOLUZIONE 2019-07-24

/**
 * Undoes the operation performed by the bpattach at memory level.
 */
void deduplica(natl id, vaddr v)
{
        for (int i = 0; i <= 3; i++) {
                tab_entry e = get_des(id, i + 1, v);
                faddr dst = extr_IND_FISICO(e);
                des_frame *pf_dst = descrittore_frame(dst);
                rilascia_frame(pf_dst);
        }
        tab_entry& e_slave = get_des(id, 4, v);
        tab_entry e_master = get_des(esecuzione->id, 4, v);
        e_slave = e_master;
}

/**
 * Removes the breakpoint inserted by the master process and reschedules the
 * slave process.
 */
extern "C" void c_bpdetach()
{
    // retrieve calling process descriptor
    des_proc *self = des_p(esecuzione->id);

    // destination (slave) process descriptor
    des_proc *dest;

    // check if the calling process is a master process (actually has a slave
    // process)
    if (!self->bp_slave)
    {
        // if not, print a warning log message
        flog(LOG_WARN, "Only master processes can use the bpdetach() primitive.");

        // abort calling process
        c_abort_p();

        // just return to the caller
        return;
    }

    // set retrieved slave process as destination process
    dest = self->bp_slave;

    // undo the duplication operation performed by the bpattach()
    deduplica(self->bp_slave_id, self->bp_addr);

    // check if there is a slave process in the master process bp queue
    if (self->bp_waiting)
    {
        // insert the calling master process in the system ready processes list
```

```
        inspronti();

        // insert the slave process in the system ready processes list
        inserimento_lista(pronti, self->bp_waiting);

        // clear any waiting slave processes
        self->bp_waiting = 0;

        // schedule a new process
        schedulatore();
        }

    // no more waiting slave process for the master process
    self->bp_slave = 0;

    // no more breakpoint address for the master process
    self->bp_addr = 0;

    // no more original byte for the master process
    self->bp_orig = 0;

    // no more slave process ID for the master process
    self->bp_slave_id = 0;

    // no more waiting master process for the slave process
    dest->bp_master = 0;
}

// SOLUTION 2019-07-24
```