Prova pratica di Calcolatori Elettronici

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

12 giugno 2019

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

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

   Realizzare in Assembler GCC le funzioni membro seguenti.

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

2. Vogliamo fornire ai processi la possibilità di scoprire se l'esecuzione di un altro processo passa da una certa istruzione. Per far questo forniamo una primitiva `breakpoint(vaddr rip)` che installa un breakpoint

(istruzione `int3`, codice operativo `0xCC`) all'indirizzo `rip`, quindi blocca il processo chiamante, sia P_1. Quando (e se) un altro processo P_2 arriva a quell'indirizzo, il processo P_1 deve essere risvegliato. Si noti che il processo P_2 non si blocca e deve proseguire la sua esecuzione indisturbato (salvo che potrebbe dover cedere il processore a P_1 per via della preemption).

Prevediamo le seguenti limitazioni del meccanismo:

1. per ogni chiamata di `breakpoint(rip)` viene intercettato solo il primo processo che passa da `rip`: altri processi che dovessero passarvi dopo il primo non vengono intercettati;
2. Un solo processo alla volta può chiamare `breakpoint()`; la primitiva restituisce un errore se un altro processo sta già aspettando un breakpoint.

Si noti che se un processo esegue `int3` senza che ciò sia richiesto da una primitiva `breakpoint()` attiva, il processo deve essere abortito.

Si modifichino dunque i file `sistema/sistema.s` e `sistema/sistema.cpp` per implementare la seguente primitiva:

- `natl breakpoint(vaddr rip)`: (tipo 0x59): blocca il processo chiamante in attesa che un altro processo provi ad eseguire l'istruzione all'indirizzo `rip`; restituisce l'id del processo intercettato, o 0xFFFFFFFF se un altro processo sta già aspettando un breakpoint (a qualunque indirizzo); abortisce il processo se `rip` non appartiene all'intervallo [`ini_utn_c`, `fin_utn_c`) (zona utente/condivisa).

Suggerimento: Il comando `process dump` del debugger è stato modificato in modo da mostrare il disassemblato del codice intorno al valore di `rip` salvato in pila.

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

```
#include "cc.h"
cl::cl(char c, st1 s2)
{
        for (int i = 0; i < 4; i++) {
                s.vc[i] = c;
                v[i] = s2.vc[i] - c;
        }
}
void cl::elab1(st1& s1)
{
        cl cla('x', s1);
        for (int i = 0; i < 4; i++) {
                if (s.vc[i] <= s1.vc[i]) {
                        s.vc[i] = cla.s.vc[i];
                        v[i] = cla.v[i] + i;
                }
        }
}
```

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

#-----------------------------------------------------------------------------
.TEXT
.GLOBAL _ZN2clC1Ec3st1                                  # cl::cl(char c, st1 s2)
#-----------------------------------------------------------------------------
# activation frame:
# -----------------
# i             -17
# s2            -13
# c             -9
# &this         -8
# %rbp          0
#-----------------------------------------------------------------------------
_ZN2clC1Ec3st1:
# set stack locations labels:
    .set this, -8
    .set c,    -9
    .set s2,   -13
    .set i,    -17

# 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)
    movb %sil, c(%rbp)
    movl %edx, s2(%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:
    movslq i(%rbp), %rcx            # i -> %rcx
    movq   this(%rbp), %rdi         # &this -> %rdi
    movb   c(%rbp), %al             # c -> %al
    movb   %al, 32(%rdi, %rcx, 1)   # s.vc[i] = c
    leaq   s2(%rbp), %rsi           # &s2 -> %rsi
    movsbq (%rsi, %rcx, 1), %rbx    # s2.vc[i] -> %rbx
    movsbq %al, %rax                # %al -> %rax
    subq   %rax, %rbx               # s2.vc[i] - c -> %rbx
    movq   %rbx, (%rdi, %rcx, 8)    # v[i] = s2.vc[i] - c;

    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 _ZN2cl5elab1ER3st1                              # void cl::elab1(st1& s1)
#-----------------------------------------------------------------------------
# activation frame:
# -----------------
# i             -60
```

```
# cla.v[0]      -56
# cla.v[1]      -48
# cla.v[2]      -40
# cla.v[3]      -32
# cla.s         -24
# &s1           -16
# &this         -8
# %rbp          0
#-------------------------------------------------------------------------
_ZN2cl5elab1ER3st1:
# set stack location labels:
    .set this, -8
    .set s1,   -16
    .set cla,  -56
    .set i,    -60

# prologue: activation frame
    pushq %rbp
    movq  %rsp, %rbp
    subq  $64, %rsp                 # reserve stack space for actual arguments

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

# cl cla('x', s1);
    leaq cla(%rbp), %rdi
    movb $'x', %sil
    movq s1(%rbp), %rdx
    movl (%rdx), %edx
    call _ZN2clC1Ec3st1

# 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:
    movslq i(%rbp), %rcx            # i -> %rcx
    movq   s1(%rbp), %rsi           # &s1 -> %rsi
    movq   this(%rbp), %rdi         # &this -> %rdi
    movb   32(%rdi, %rcx, 1), %bl   # s.vc[i] -> %bl
    movb   (%rsi, %rcx, 1), %al     # s1.vc[i] -> %al
    cmpb   %al, %bl                 # compare s.vc[i] and s1.vc[i]
    jg     fineif                   # exit if (s.vc[i] > s1.vc[i])
    leaq   cla(%rbp), %rsi          # &cla -> %rsi
    movq   this(%rbp), %rdi         # &this -> %rdi
    movb   32(%rsi, %rcx, 1), %al   # cla.s.vc[i] -> %al
    movb   %al, 32(%rdi, %rcx, 1)   # s.vc[i] = cla.s.vc[i]
    movq   (%rsi, %rcx, 8), %rax    # cla.v[i] -> %rax
    addq   %rcx, %rax               # cla.v[i] + i -> %rax
    movq   %rax, (%rdi, %rcx, 8)    # v[i] = cla.v[i] + i;

fineif:

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

finefor1:

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

```
// prova1.cpp
#include "cc.h"
int main()
{
        st1 s3 = { 'm', 'n', 'c', 'j' };
        st1 sa = { 1, 20, 3, 40 };
        cl cla('c', sa);
        cla.stampa();
        cla.elab1(s3);
        cla.stampa();
}
```

```
c c c c
-98 -79 -96 -59

x x x x
-11 -9 -19 -11
```

```
// [...]

// EXTENSION 2019-06-12

/**
 * Interrupt type for the breakpoint() primitive.
 */
#define  TIPO_B  0x59

// EXTENSION 2019-06-12

// [...]
```

```
// [..]

// EXTENSION 2019-06-12

/**
 * Virtual address definition for the User Module.
 */
typedef natq vaddr;

/**
 * We want to provide to the processes a mechanism to allow them to know if the
 * execution of one of the other processes reaches a certain instruction. Only
 * the first process reaching the breakpoint instruction must be intercepted.
 * All other processes should work seamlessly.
 */

/**
 * This primitive allows a user process to install a breakpoint (assembly
 * instruction int3, opcode 0xCC) at the address pointed by rip. The calling
 * process (P1) must be paused. When another process (P2) reaches the breakpoint
 * address P1 must be rescheduled. The P2 process is not paused, however it
 * might have to be rescheduled in order to guarantee processes priority.
 *
 * All other process executing the int3 instruction must be aborted.
 *
 * The rip address must belong to the process user/shared memory area. Otherwise
 * the calling process must be aborted.
 *
 * @param  rip  breakpoint instruction address.
 *
 * @return  the id of the intercepted process at the given address or 0xFFFFFFFF
 *          if another process has already requested a breakpoint (at any given
 *          address).
 */
extern "C" natl breakpoint(vaddr rip);

// EXTENSION 2019-06-12

// [...]
```

```
# [...]

# EXTENSION 2019-06-12

#--------------------------------------------------------------------------------
.GLOBAL breakpoint                      # extern "C" natl breakpoint(vaddr rip)
#--------------------------------------------------------------------------------
breakpoint:
    .cfi_startproc
    int $TIPO_B
    ret
    .cfi_endproc

# EXTENSION 2019-06-12
```

```
# [...]

# EXTENSION 2019-06-12

    #--------------------------------------------------------------------------
    # The interrupt 3 IDT gate DPL must be redifed to User Level in order for
    # the User module processes to call the int3 instruction.
    #--------------------------------------------------------------------------
    carica_gate  3  breakpoint  LIV_UTENTE

# EXTENSION 2019-06-12

# [...]

# EXTENSION 2019-06-12

    # initialize IDT gate for the TIPO_B (breakpoint() primitive) interrupt
    carica_gate  TIPO_B  a_breakpoint  LIV_UTENTE

# EXTENSION 2019-06-12

# [...]

# SOLUTION 2019-06-12

#------------------------------------------------------------------------------
# When the breakpoint() primitive is called, if everything goes well the calling
# process is placed in the system global breakpoint descriptor queue. Otherwise,
# in case of error, the calling process is aborted. In either case the state
# of the current process has to be saved and the state of a new process must be
# loaded.
#------------------------------------------------------------------------------
a_breakpoint:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato            # save current process state
    call c_breakpoint           # call C++ implementation
    call carica_stato           # load new process state
    iretq                       # return from interrupt
    .cfi_endproc

# SOLUTION 2019-06-12

# [...]

#------------------------------------------------------------------------------
# Interrupt 3 - Breakpoint Exception handler
# The breakpoint exception handler must be redefined in order for the user
# process which called the breakpoint() primitive to be rescheduled. When the
# breakpoint exception is handled, if the breakpoint was inserted by the
# breakpoint() primitive both processes must be place in the system ready
# processes queue and a new process must be scheduled. If it is a breakpoint
# exception not related to the breakpoint() primitive, the calling process
# must be aborted a new one must be scheduled. In either case the current
# process state must be saved and a new process state must be loaded.
#------------------------------------------------------------------------------
breakpoint:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato

# SOLUTION 2019-06-12

    movq $3, %rdi                   # interrupt type
    movq $0, %rsi                   # error type
    movq %rsp, %rdx                 # address contained in %rsp
```

```
    call c_breakpoint_exception     # call C++ handler

# SOLUTION 2019-06-12

    call carica_stato               # load new process state
    iretq
    .cfi_endproc

# [...]
```

```
// [...]

// SOLUTION 2019-06-12

/**
 * Sysmte global breakpoint descriptor struct.
 */
struct b_info
{
    /**
     * Wait queue for the first process which calls the breakpoint() primitive.
     * All other processes calling the breakpoint() primitive must be ignored.
     */
    struct proc_elem *waiting;

    // %rip
    natq rip;

    // original byte addressed by %rip
    natb orig;

  // system global breakpoint descriptor
} b_info;

/**
 * @param  rip  the address where the breakpoint should be placed.
 */
extern "C" void c_breakpoint(natq rip)
{
    // retrieve calling process descriptor
    struct des_proc *self = des_p(esecuzione->id);

    // check if there is already a process which called the breakpoint()
    // primitive and is waiting
    if (b_info.waiting)
    {
        // if so, set return value
        self->contesto[I_RAX] = 0xFFFFFFFF;

        // just return to the caller
        return;
        }

    // check if the given 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;
        }

    // retrieve the first byte pointed by %rsp
    natb *bytes = reinterpret_cast<natb*>(rip);

    // save %rip
    b_info.rip = rip;

    // save original byte pointed by %rip
    b_info.orig = *bytes;

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

    // queue the calling process
    b_info.waiting = esecuzione;
```

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

/**
 * Called when the breakpoint exception occurs.
 *
 * @param  tipo         interrupt type (3);
 * @param  errore       error code (0);
 * @param  p_saved_rip  content of %rip.
 */
extern "C" void c_breakpoint_exception(int tipo, natq errore, vaddr* p_saved_rip)
{
    // check if there is a process waiting in the system global breakpoint
    // descriptor wait queue
    if (!b_info.waiting || *p_saved_rip != b_info.rip + 1)
    {
        // if not, handle breakpoint exception: the calling process is aborted
        // in the gestore_eccezioni()
        gestore_eccezioni(tipo, errore, *p_saved_rip);

        // just return to the caller
        return;
    }

    // otherwise...

    // retrieve byt pointed by the value of %rip saved in the global breakpoint
    // descriptor
    natb *bytes = reinterpret_cast<natb*>(b_info.rip);

    // write the original byte back
    *bytes = b_info.orig;

    // decrease %rip for the calling process
    (*p_saved_rip)--;

    // retrieve process descriptor for the process in the wait queue of the
    // system global breakpoint descriptor
    des_proc *dest = des_p(b_info.waiting->id);

    // set return value for such process (which is the process that originally
    // called the breakpoint() primitive)
    dest->contesto[I_RAX] = esecuzione->id;

    // place the calling process in the system ready processes queue
    inspronti();

    // place the process which called the breakpoint() primitive in the system
    // ready processes queue
    inserimento_lista(pronti, b_info.waiting);

    // clear system global breakpoint descriptor wait queue (the breakpoint()
    // primitive can now be used by another process)
    b_info.waiting = 0;

    // schedule a new process
    schedulatore();
}

// SOLUTION 2019-06-12
```

```
/**
 * File: pbreak.in
 *       Extension 2019-06-12_22 test program.
 *
 * Author: Rambod Rahmani <rambodrahmani@autistici.org>
 *         Created on 22/09/2019.
 */

#include <sys.h>
#include <lib.h>

/**
 *
 */
process bad1 body bad(1), 20, LIV_UTENTE;

/**
 *
 */
process bad2 body bad(2), 8, LIV_UTENTE;

/**
 *
 */
process bad4 body badb(3), 21, LIV_UTENTE;

/**
 *
 */
process bad5 body badb(4), 22, LIV_UTENTE;

/**
 *
 */
process usr1 body usr(1), 5, LIV_UTENTE;

/**
 *
 */
process usr2 body usr(2), 15, LIV_UTENTE;

/**
 *
 */
process usr3 body usr(3), 3, LIV_UTENTE;

/**
 *
 */
process dbg1 body debugger(0), 10, LIV_UTENTE;

/**
 *
 */
process dbg2 body debugger(1), 11, LIV_UTENTE;

/**
 *
 */
process dbg3 body debugger(2), 4, LIV_UTENTE;

/**
 *
 */
process last body last_body(0), 1, LIV_UTENTE;

semaphore sync value 0;

/**
 * Bad process body 1: calls the int3 instruction without using the breakpoint()
```

```
 * primitive.
 */
process_body bad(int a)
{
    asm("int3");
    printf("processo errato %d", a);
}

/**
 *
 */
void catch_me(int a)
{
    printf("proc%d: eseguo funzione", a);
}

/**
 *
 */
vaddr bad_addr[] = { 1000, 0xffffc00000000000 };

/**
 * Bad process body 2: calls the breakpoint() primitive with the wrong address.
 * The breakpoint() primitive can be called only from addresses belonging to the
 * user process shared memory area.
 */
process_body badb(int a)
{
    breakpoint(bad_addr[a - 3]);
    printf("processo errato %d", a);
}

/**
 * User process:
 */
process_body usr(int a)
{
    if (a % 2 == 0)
    {
        sem_wait(sync);
    }

    printf("proc%d: prima della funzione", a);

    catch_me(a);

    printf("proc%d: dopo la funzione", a);
}

/**
 *
 */
process_body debugger(int a)
{
    printf("debugger %d: chiamo breakpoint", a);

    natl proc = breakpoint(reinterpret_cast<natq>(catch_me));

    if (proc == 0xFFFFFFFF)
    {
        printf("debugger %d: occupato", a);
        }
    else
    {
        sem_signal(sync);
        printf("debugger %d: breapoint intercettato, processo: %d", a, proc);
    }
}

/**
```

```
 *
 */
process_body last_body(int a)
{
    pause();
}
```