Prova pratica di Calcolatori Elettronici

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

3 luglio 2019

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

```
struct st {
        char vv1[4];
        long vv2[4];
};
class cl {
        st s;
public:
        cl(char v[]);
        void elab1(int d, st& ss);
        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[i];
        }
}
void cl::elab1(int d, st& ss)
{
        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 tutti i processi che passano da una certa istruzione. Per far questo forniamo alcune primitive. Con la primitiva `bpadd(vaddr rip)` si

installa un breakpoint (istruzione `int3`, codice operativo `0xCC`) all'indirizzo `rip`. Da quel momento in poi, tutti i processi che passano da `rip` si bloccano e vengono accodati opportunamente. Nel frattempo, usando la primitiva `bpwait()`, un processo può sospendersi in attesa che un qualche altro processo passi dal breakpoint. La primitiva può essere invocata più volte, per attendere tutti i processi che si suppone debbano passare dal breakpoint. Infine, con la primitiva `bpremove()`, si rimuove il breakpoint e si risvegliano tutti i processi che vi si erano bloccati. I processi così risvegliati devono proseguire la loro esecuzione come se non fossero mai stati intercettati.

Prevediamo la seguente limitazione: ad ogni istante, nel sistema ci può essere al massimo un breakpoint installato tramite da `bpadd()`.

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

Aggiungiamo al nucleo la seguente struttura dati:

```
struct b_info {
    proc_elem *waiting;
    proc_elem *intercepted;
    proc_elem *to_wakeup;
    vaddr rip;
    natb orig;
    bool busy;
} b_info;
```

dove: `waiting` è una coda di processi che hanno invocato `bpwait()` e sono in attesa che qualche processo passi dal breakpoint; `intercepted` è una coda di processi che sono bloccati sul breakpoint e il cui identificatore non è stato ancora restituisto da una `bpwait()`; `to_wakeup` è una coda di processi bloccati sul breakpoint e i cui indentificatori sono stati già restituiti tramite `bpwait()`; `rip` è l'indirizzo a cui è installato il breakpoint; `orig` è il byte originariamente contenuto all'indirizzo `rip`; `busy` vale `true` se c'è un breakpoint installato.

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

- `bool bpadd(vaddr rip)`: (tipo 0x59, già realizzata): se non c'è un altro breakpoint già installato, 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).
- `natl bpwait()`: (tipo 0x5a, già realizzata): attende che un qualche processo passi dal breakpoint e ne restituisce l'identificatore; può essere invocata più volte per ottenere gli identificatori di tutti i processi intercettati; è un errore invocare questa primitiva se non ci sono breakpoint installati;
- `void bpremove()` (tipo 0x5b, da realizzare): rimuove il breakpoint e risveglia tutti i processi che erano stati intercettati; è un errore invocare questa primitiva se non ci sono breakpoint installati.

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 st {
        char vv1[4];
        long vv2[4];
};
class cl {
        st s;
public:
        cl(char v[]);
        void elab1(int d, st& ss);
        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;
        }
};
```

```
#include "cc.h"
cl::cl(char v[])
{
        for (int i = 0; i < 4; i++) {
                s.vv1[i] = s.vv2[i] = v[i];
        }
}
void cl::elab1(int d, st& ss)
{
        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 frame:
# -----------------
# i             -20
# &v            -16
# &this         -8
# %rbp          0
#-----------------------------------------------------------------------------
_ZN2clC1EPc:
# set stack location labels:
    .set this, -8
    .set v,    -16
    .set i,    -20

# prologue: activation frame
    pushq %rbp
    movq  %rsp, %rbp
    subq  $24, %rsp                 # reserver 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:
    movslq i(%rbp), %rcx            # i -> %rcx
    movq   this(%rbp), %rdi         # &this -> %rdi
    movq   v(%rbp), %rsi            # &v -> %rsi
    movsbq (%rsi, %rcx, 1), %rax    # v[i] -> %rax
    movb   %al, (%rdi, %rcx, 1)     # s.vv1[i] = v[i];
    movq   %rax, 8(%rdi, %rcx, 8)   # s.vv2[i] = v[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 _ZN2cl5elab1EiR2st                      # void cl:: elab1(int d, st& ss)
#-----------------------------------------------------------------------------
# activation frame:
# -----------------
# i             -28
# &ss           -24
# d             -12
# &this         -8
# %rbp          0
#-----------------------------------------------------------------------------
_ZN2cl5elab1EiR2st:
```

```
# set stack locations labels:
    .set this, -8
    .set d,    -12
    .set ss,   -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)
    movl %esi, d(%rbp)
    movq %rdx, ss(%rbp)

# 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
    movslq d(%rbp), %rdx            # d -> %rdx
    movq   ss(%rbp), %rsi           # &ss -> %rsi

# if (d >= ss.vv2[i])
    movq   8(%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   (%rsi, %rcx, 1), %bl     # ss.vv1[i] -> %bl
    addb   %bl, (%rdi, %rcx, 1)     # s.vv1[i] += ss.vv1[i]

fineif:

    addq   %rcx, %rdx               # d + i -> %rdx
    movq   %rdx, 8(%rdi, %rcx, 8)   # s.vv2 = d + i;

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

finefor1:

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

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

```
// prova1.cpp
#include "cc.h"
int main()
{   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(d, s); cc1.stampa();
}
```

```
10 11 12 13     10 11 12 13

11 13 12 13     2 3 4 5
```

```
// [...]

// EXTENSION 2019-07-03

/**
 * User Primitives interrupt types declarations.
 */

/**
 * extern "C" bool bpadd(vaddr rip);
 */
#define  TIPO_BPA  0x59

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

/**
 * extern "C" void bpremove();
 */
#define  TIPO_BPR  0x5b

// EXTENSION 2019-07-03

// [...]
```

```
// [...]

// EXTENSION 2019-07-03

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

/**
 * Primitives declarations for the User Module. We want to provide to the User
 * processes the ability to puase the execution of all processes which execute
 * a certain instruction.
 */

/**
 * Allows to install a breakpoint at the instruction having the provided virtual
 * address (keep in mind that the breakpoint exception has assembly instruction
 * int3 and opcode 0xCC). Once this method is called on a given address, all
 * processes execution will be paused and queued.
 *
 * For simplicity, at any given time, there is one and only one breakpoint
 * installed. If a process spontaneously calls the int3 instruction without
 * being added using this primitive, the calling process must be aborted.
 *
 * @param  rip  the virtual address of the instruction to be replaced with the
 *              breakpoit;
 */
extern "C" bool bpadd(vaddr rip);

/**
 * Can be used by a process to wait until a process reaches the breakpoint
 * instruction address. This primitive waits for one process to reach the
 * breakpoint instruction address and returns its id; it can be called multiple
 * times in order to retrieve the id of all the processes with the breakpoint
 * installed; it is an error if there not breakpoints installed in the system
 * global breakpoint descriptor.
 */
extern "C" natl bpwait();

/**
 * Removes the breakpoint and reschedules all paused process. The processes
 * rescheduled must execute without side effects.
 */
extern "C" void bpremove();

// EXTENSION 2019-07-03

// [...]
```

```
# [...]

# EXTENSION 2019-07-03

#--------------------------------------------------------------------------------
.GLOBAL bpadd             # Implementation for extern "C" bool bpadd(vaddr rip);
#--------------------------------------------------------------------------------
bpadd:
    .cfi_startproc
    int $TIPO_BPA
    ret
    .cfi_endproc

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

#--------------------------------------------------------------------------------
.GLOBAL bpremove                # Implementation for extern "C" void bpremove();
#--------------------------------------------------------------------------------
bpremove:
    .cfi_startproc
    int $TIPO_BPR
    ret
    .cfi_endproc

# EXTENSION 2019-07-03
```

```
# [...]

# EXTENSION 2019-07-03

    # redefine interrupt 3 dpl level to user level in order for the User Module
    # processes to be able to use the int3 instruction
    carica_gate  3  breakpoint  LIV_UTENTE

# EXTENSION 2019-07-03

# [...]

# EXTENSION 2019-07-03

    # init IDT gate subroutine for the bpadd() primitive
    carica_gate  TIPO_BPA  a_bpadd   LIV_UTENTE

    # init IDT gate subroutine for the bpwait() primitive
    carica_gate  TIPO_BPW  a_bpwait  LIV_UTENTE

# EXTENSION 2019-07-03

# SOLUTION 2019-07-03

    # init IDT gate subroutine for the bpremove() primitive
    carica_gate  TIPO_BPR  a_bpremove  LIV_UTENTE

# SOLUTION 2019-07-03

# [...]

# EXTENSION 2019-07-03

#--------------------------------------------------------------------------------
#--------------------------------------------------------------------------------
a_bpadd:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_bpadd
    call carica_stato
    iretq
    .cfi_endproc

#--------------------------------------------------------------------------------
#--------------------------------------------------------------------------------
a_bpwait:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_bpwait
    call carica_stato
    iretq
    .cfi_endproc

# EXTENSION 2019-07-03

# SOLUTION 2019-07-03

#--------------------------------------------------------------------------------
#--------------------------------------------------------------------------------
a_bpremove:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
```

```
    call salva_stato
    call c_bpremove
    call carica_stato
    iretq
    .cfi_endproc

# SOLUTION 2019-07-03

# [...]

#--------------------------------------------------------------------------------
# Interrupt 3 - Breakpoint exception.
# We must redefine the subroutine handling the breakpoint exception in order to
# call a custom C++ implementation which will reschedule waiting processes and
# udpate the system global breakpoint descriptor status. This assembly routine
# loads the arguments for and calls the C++ handler.
#--------------------------------------------------------------------------------
breakpoint:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato

# SOLUTION 2019-07-03

    movq $3, %rdi                   # exception type
    movq $0, %rsi                   # exception error
    movq (%rsp), %rdx               # current value addressed by %rsp
    call c_breakpoint_exception

# SOLUTION 2019-07-03

    call carica_stato
    iretq
    .cfi_endproc

# [...]
```

```
// [...]

// EXTENSION 2019-07-03

// traduce l'indirizzo virtuale ind_virt nel corrispondente
// indirizzo fisico nello spazio virtuale del processo di
// identificatore id (il processo deve esistere)
extern "C" faddr trasforma(natl id, vaddr ind_virt)
{
    natq d;
    for (int liv = 4; liv > 0; liv--)
    {
        d = get_des(id, liv, ind_virt);

        if (!extr_P(d))
        {
            flog(LOG_WARN, "impossibile trasformare %lx: non presente a livello %d", ind_
virt, liv);
            return 0;
            }

        if (extr_PS(d))
        {
            // pagina di grandi dimensioni
            natq mask = (1UL << ((liv - 1) * 9 + 12)) - 1;
            return norm((d & ~mask) | (ind_virt & mask));
        }
    }

    return extr_IND_FISICO(d) | (ind_virt & 0xfff);
}

/**
 * System global breakpoint descriptor struct.
 */
struct b_info
{
    /**
     * Wait queue of the processes which called the bpwait() primitive and are
     * waiting for a process to reach the breakpoint address.
     */
    proc_elem *waiting;

    /**
     * Wait queue for all the process which have reached the breakpoint and
     * which IDs have not been yet retrieved using the bpwait() primitive.
     */
    proc_elem *intercepted;

    /**
     * Wait queue for all the processes which have reached the brakpoint and
     * which IDs have already been retrieved using the bpwait() and need to
     * be rescheduled.
     */
    proc_elem *to_wakeup;

    /**
     * Breakpoint virtual address.
     */
    vaddr rip;

    /**
     * Original byte in the replaced instruction.
     */
    natb orig;

    /**
     * True when there is a breakpoint installed.
     */
    bool busy;
```

```
  // system global breakpoint descriptor
} b_info;

/**
 * Adds a breakpoint at the given virtual address.
 *
 * @param  rip  the address where to add the breakpoint.
 */
extern "C" void c_bpadd(vaddr rip)
{
    // retrieve calling process descriptor
    des_proc *self = des_p(esecuzione->id);

    // check if there is a global system breakpoint installed
    if (b_info.busy)
    {
        // if so, return false: breakpoint already present
        self->contesto[I_RAX] = false;

        // 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)
    {
        // print a warning log message
        flog(LOG_WARN, "rip %p out of bounds [%p, %p)", rip, ini_utn_p, fin_utn_p);

        // abort the calling process
        c_abort_p();

        // just return to the caller
        return;
        }

    // retrieve byte address by the given virtual address
    natb *bytes = reinterpret_cast<natb*>(rip);

    // save the given virtual address for later use
    b_info.rip = rip;

    // save the original byte being replace for later use
    b_info.orig = *bytes;

    // replace the retrieved byte with the opcode of the int3 instruction
    *bytes = 0xCC;

    // set system global breakpoint descriptor busy flag to true: one breakpoint
    // is already present and no more will be accepted
    b_info.busy = true;

    // return true: breakpoint successfully placed
    self->contesto[I_RAX] = true;
}

/**
 *
 */
extern "C" void c_bpwait()
{
    // retrieve calling process descriptor
    des_proc *self = des_p(esecuzione->id);

    // check if there is a breakpoint already placed
    if (!b_info.busy)
    {
        // if not, return no breakpoints present
        self->contesto[I_RAX] = 0xFFFFFFFF;
```

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

    // check if there is any process which have rached the breakpoint address
    if (b_info.intercepted)
    {
        // if so, we need a process descriptor
        proc_elem *work;

        // remove one of such processes from the queue
        rimozione_lista(b_info.intercepted, work);

        // return the process id to the caller
        self->contesto[I_RAX] = work->id;

        // place the process in the wakup list
        inserimento_lista(b_info.to_wakeup, work);
    }
    else
    {
        // otherwise, place the calling process in the waiting queue
        inserimento_lista(b_info.waiting, esecuzione);

        // schedule a new process
        schedulatore();
    }
}
// EXTENSION 2019-07-03 )

// SOLUTION 2019-07-03

/**
 * Removes the breakpoint and reschedules all the process which had reached the
 * breakpoint address and were placed in the intercepted wait queue.
 */
extern "C" void c_bpremove()
{
    // check if there is any process which have called the bpwait()
    // or the busy flag is set
    if (b_info.waiting || !b_info.busy)
    {
        // if not, no breakpoint can be removed either because there is none or
        // because there are process waiting
        flog(LOG_WARN, "Unable to perform bpremove().");

        // abort calling process
        c_abort_p();

        // just return to the caller
        return;
    }

    // retrieve address to the replaced byte
    natb *bytes = reinterpret_cast<natb*>(b_info.rip);

    // replace the byte with the original value
    *bytes = b_info.orig;

    // process descriptor
    proc_elem *work;

    // while there are processes which have reached the breakpoint address
    while (b_info.intercepted)
    {
        // remove them from the intercepted queue
        rimozione_lista(b_info.intercepted, work);

        // place them in the wakeup queue
```

```
        inserimento_lista(b_info.to_wakeup, work);
        }

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

    // while there are processes which have reached the breakpoint address and
    // need to be rescheduled
    while (b_info.to_wakeup)
    {
        // retrieve next process to wake up
        rimozione_lista(b_info.to_wakeup, work);

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

        // retrieve process virtual %rsp value
        natq rsp_v = dp->contesto[I_RSP];

        // retrieve physical address of %rsp
        natq *rsp = reinterpret_cast<natq*>(trasforma(work->id, rsp_v));

        // decrease it of one
        (*rsp)--;

        // place the process in the system ready processes list
        inserimento_lista(pronti, work);
    }

    //
    b_info.busy = false;

    // schedule a new process
    schedulatore();
}

/**
 * Called when a breakpoint exception occurs.
 *
 * @param  tipo    interrupt type (3);
 * @param  errore  error type (0);
 * @param  rip     current value address by %rsp.
 */
extern "C" void c_breakpoint_exception(int tipo, natq errore, vaddr rip)
{
    // check if there is any breakpoint in the system global breakpoint
    // descriptor
    if (!b_info.busy || rip != b_info.rip + 1)
    {
        // if not, the bpadd() primitive was not used: handle the exception and
        // abort the calling process
        gestore_eccezioni(tipo, errore, rip);

        // just return to the caller
        return;
    }

    // check if there is any process waiting for a breakpoint
    if (b_info.waiting)
    {
        // if so, we need to notify such processes that an external process has
        // reached the breakpoint
        proc_elem *work;

        // retrieve such process proc_elem
        rimozione_lista(b_info.waiting, work);

        // retrieve process descriptor
        des_proc *dp = des_p(work->id);
```

```
        // notify the waiting process that the current process in execution has
        // reached the breakpoint address
        dp->contesto[I_RAX] = esecuzione->id;

        // place the current process in the breakpoint descriptor to_wakeup
        // queue
        inserimento_lista(b_info.to_wakeup, esecuzione);

        // insert the waiting process in the system ready processes queue
        inserimento_lista(pronti, work);
        }
    else
    {
        // otherwise, just place the current process under execution in the
        // intercepted processes queue to wait for a process to call the
        // bpwait() primitive
        inserimento_lista(b_info.intercepted, esecuzione);
    }

    // schedule a new process
    schedulatore();
}

// SOLUTION 2019-07-03
```