Prova pratica di Calcolatori Elettronici

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

20 settembre 2016

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 + i;
                v[i] = s2.vd[i] + s.vc[i];
        }
}
void cl::elab1(st1 s1, st2 s2) {
        cl cla('a', s2);
        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];
                }
        }
}
```

2. Introduciamo un meccanismo di *broadcast* tramite il quale un processo può inviare un messagio ad un insieme di processi. Per ricevere un broadcast i processi si devono preventivamente registrare. Un processo può inviare un brodcast tramite la primitiva `broadcast(msg)`, la quale attende anche che tutti i processi che risultano registrati ricevano il messaggio. Un processo registrato può ricevere un broadcast invocando la primitiva `listen()`, che attende che sia disponibile il prossimo messaggio.

   Per realizzare i broadcast introduciamo la seguente struttura dati:

```
struct broadcast {
    int registered;
    int nlisten;
    natl msg;
    proc_elem *listeners;
    proc_elem *broadcaster;
};
```

Dove: `registered` è il numero di processi registrati; `nlisten` conta i processi che hanno invocato `listen()` dall'ultimo completamento di una operazione di broadcast; `msg` contiene l'ultimo messaggio di broadcast; `listeners` è la coda dei processi in attesa del prossimo messaggio; `broadcaster` è la coda in cui attende il processo che vuole inviare il broadcast, in attesa che tutti i processi registrati invochino `listen()`.

Aggiungiamo inoltre le seguenti primitive (abortiscono il processo in caso di errore):

- `void reg()` (tipo 0x3a, già realizzata): registra il processo per la ricezione dei broadcast; non fa niente se il processo è già registrato;
- `natl listen()` (tipo 0x3b, da realizzare): riceve il prossimo messagio di broadcast; è un errore se il processo non è registrato;
- `void broadcast(natl msg)` (tipo 0x3c, da realizzare): invia in broadcast il messaggio `msg`; è un errore se il processo è registrato.

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

Per semplicità si assuma che, durante tutta la sua esecuzione, ogni processo provi al più una sola volta a inviare un broadcast o ad ascoltare un messaggio. Inoltre, al più un processo alla volta tenta di eseguire un broadcast.

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

```
/**
 * 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 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 14/09/2019.
 */

#include "cc.h"

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

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

    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];
        }
    }
}
```

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

#-----------------------------------------------------------------------------
.TEXT
.GLOBAL _ZN2clC1EcR3st2                                  # cl::cl(char c, st2& s2)
#-----------------------------------------------------------------------------
# activation frame:
# -----------------
#  i            -28
#  &s2          -24
#  c            -9
#  this         -8
#  %rbp         0
#-----------------------------------------------------------------------------
_ZN2clC1EcR3st2:
# set stack locations labels:
    .set this, -8
    .set c,    -9
    .set s2,   -24
    .set i,    -28

# prologue: activation frame
    pushq %rbp
    movq  %rsp, %rbp
    subq  $28, %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)

# 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   s2(%rbp), %rsi           # &s2 -> %rsi
    movslq i(%rbp), %rcx            # i => %rcx
    movb   c(%rbp), %al             # c -> %al
    addb   %cl, %al                 # c + i -> %al
    movb   %al, (%rdi, %rcx, 1)     # s.vc[i] = c + i;
    movsbl (%rdi, %rcx, 1), %ebx    # s.vc[i] -> %bl
    movl   (%rsi, %rcx, 4), %eax    # s2.vd[i] -> %eax
    addl   %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                        # loop again

finefor:

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

#-----------------------------------------------------------------------------
.GLOBAL _ZN2cl5elab1E3st13st2                  # void cl::elab1(st1 s1, st2 s2)
#-----------------------------------------------------------------------------
# activation frame:
```

```
# -----------------
#  i            -76
#  cla.s        -72
#  cla.v[0]     -64
#  cla.v[1]     -56
#  cla.v[2]     -48
#  cla.v[3]     -40
#  s2 [MSB]     -32
#  s2 [LSB]     -24
#  s1           -12
#  this         -8
#  %rbp         0
#-------------------------------------------------------------------------------
_ZN2cl5elab1E3st13st2:
# set stack locations labels
    .set this, -8
    .set s1,   -16
    .set s2,   -32
    .set cla,  -72
    .set i,    -76

# prologue: activation frame
    pushq %rbp
    movq  %rsp, %rbp
    subq  $80, %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)
    movq %rcx, -24(%rbp)

# cl cla('a', s2);
    leaq cla(%rbp), %rdi
    movb $'a', %sil
    leaq s2(%rbp), %rdx
    call _ZN2clC1EcR3st2

# for loop 1 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 1 body
# if (s.vc[i] <= s1.vc[i])
    movq   this(%rbp), %rdi         # this -> %rdi
    movslq i(%rbp), %rcx            # i => %rcx
    leaq   s1(%rbp), %rsi           # &s1 -> %rsi
    movb   (%rsi, %rcx, 1), %al     # s1.vc[i] -> %al
    movb   (%rdi, %rcx, 1), %bl     # s.vc[i] -> %bl
    cmpb   %al, %bl                 # compare s.vc[i] and s1.vc[i]
    jg     fineif                   # exit if (s.vc[i] > s1.vc[i])

    # if body #
    leaq cla(%rbp), %r8
    movb (%r8, %rcx, 1), %al        # cla.s.vc[i] -> %al
    movb %al, (%rdi, %rcx, 1)       # s.vc[i] = cla.s.vc[i];

    movq 8(%r8, %rcx, 8), %rbx      # cla.v[i] -> %rbx
    leaq 8(%rdi), %r9               # &v -> %r9
    movq %rbx, (%r9, %rcx, 8)       # v[i] = cla.v[i];


fineif:

    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[])
{
    st1 s1 = { 'e', 'a', '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 b c d
98 118 102 140

a b c d
107 118 129 104
```

```
// EXTENSION 2016-09-20

/**
 * PRIMITIVES INTERRUPT TYPES DEFINITIONS.
 */

#define TIPO_R                  0x3a    // reg()
#define TIPO_LS                 0x3b    // listen()
#define TIPO_B                  0x3c    // broadcast(natl msg)

// EXTENSION 2016-09-20
```

```
// EXTENSION 2016-09-20

/**
 * We want to add a broadcasting mechanism to the system allowing a process to
 * send a broadcast messages to all of the registered listener processes. To do
 * it we provide the reg() primitive which can be used to register a process
 * as a listener, the listen() primitive which can be used to receive the
 * broadcast message and broadcast(natl msg) which can be used by the
 * broadcaster process to send a broadcast message.
 */

/**
 * PRIMITIVES DECLARATIONS.
 */

/**
 * Registers the process as a listener of broadcast messages. No action will
 * take place if the process is already a registered listener. We will be using
 * a global broadcast descriptor and all processes will be registered as
 * listener of this global broadcast descriptor.
 */
extern "C" void reg();

/**
 * Waits for the next broadcast message. An error should be rised if the process
 * is not registered as a listener of the broadcast messages. We will be using
 * a global broadcast descriptor and this primitive can be used to wait for the
 * next message sent on the global broadcast descriptor.
 */
extern "C" natl listen();

/**
 * Broadcasts the given message to all registered processes. An error should be
 * rised if the process is not registered as a listener of the broadcast
 * messages. We will be using a global broadcast descriptor for simplicity and
 * the broadcaster register can use this primitive to send a new message on the
 * global broadcast.
 *
 * @param  msg  the message to be broadcasted.
 */
extern "C" void broadcast(natl msg);

// EXTENSION 2016-09-20
```

```
# EXTENSION 2016-09-20

##
# Definitions for the primitives declared in sys.h. They all call the
# corresponding interrupts declared in costanti.h.
##

#-----------------------------------------------------------------------------
.GLOBAL reg                                 # Primitive void reg() implementation
#-----------------------------------------------------------------------------
reg:
    int $TIPO_R
    ret

#-----------------------------------------------------------------------------
.GLOBAL listen                            # Primitive natl listen() implementation
#-----------------------------------------------------------------------------
listen:
    int $TIPO_LS
    ret

#-----------------------------------------------------------------------------
.GLOBAL broadcast            # Primitive void broadcast(natl msg) implementation
#-----------------------------------------------------------------------------
broadcast:
    int $TIPO_B
    ret

# EXTENSION 2016-09-20
```

```
# EXTENSION 2016-09-20

##
# PRIMITIVES INTERRUPTS PINS.
##

    # load void reg() primitive interrupt handler in the IDT
    carica_gate TIPO_R     a_reg            LIV_UTENTE

# EXTENSION 2016-09-20

# SOLUTION 2016-09-20

    # load natl listen() primitive interrupt handler in the IDT
    carica_gate TIPO_LS    a_listen         LIV_UTENTE

    # load void broadcast(natl msg) interrupt primitive handler in the IDT
    carica_gate TIPO_B     a_broadcast      LIV_UTENTE

# SOLUTION 2016-09-20

# EXTENSION 2016-09-20

##
# PRIMITIVES INTERRUPTS HANDLERS.
##

#-----------------------------------------------------------------------------
.GLOBAL a_reg                           # void reg() primitive interrupt handler
#-----------------------------------------------------------------------------
a_reg:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_reg                      # call C++ implementation
    iretq                           # return from interrupt
    .cfi_endproc

# EXTENSION 2016-09-20

# SOLUTION 2016-09-20

#-----------------------------------------------------------------------------
.GLOBAL a_listen                     # natl listen() primitive interrupt handler
#-----------------------------------------------------------------------------
# The listen() primitive will hang the calling process 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 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                # save current process state
    call c_listen                   # call C++ implementation
    call carica_stato               # load new process state
    iretq                           # return from interrupt
    .cfi_endproc

#-----------------------------------------------------------------------------
.GLOBAL a_broadcast       # void broadcast(natl msg) interrupt primitive handler
#-----------------------------------------------------------------------------
# The c_broadcast C++ implementation for this IDT subroutine will queue the
# broadcaster process in either the global broadcast descriptor broadcaster
# queue (there are still some listener processes which must call the listen()
# primitive to receive the broadcast message) or in the system ready process
```

```
# (all listener processes have received the broadcast message using the
# broadcast_all utility method). 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                # save current process state
    call c_broadcast                # call C++ implementation
    call carica_stato               # load new process state
    iretq                           # return from interrupt
    .cfi_endproc

# SOLUTION 2016-09-20
```

```
/**
 * Process descriptor. We must edit this struct and add the listen_reg boolean
 * field which is set to true when a process registers as a listener of the
 * global broadcast descriptor. For simplicity we will assume that for the whole
 * system execution only one process at a time will try to send one and only one
 * broadcast message and that the listen primitive() will be called by each
 * listener process one and only one time.
 */
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 2016-09-20

    // true if the process is registered to the global broadcast descriptor
        bool listen_reg;

// EXTENSION 2016-09-20
};

// EXTENSION 2016-09-20

/**
 * Global broadcast descriptor struct.
 */
struct broadcast
{
    // number of registered listener processes
    int registered;

    // number of registered processes which have already called the listen()
    // primitive
    int nlisten;

    // last broadcast message sent
    natl msg;

    // processes waiting for the next broadcast message queue: processes which
    // have called the listen() primitive
    proc_elem *listeners;

    // broadcaster wait queue: the broadcaster process waits here until all
    // registered listener processes have called the listen() primitive
    proc_elem *broadcaster;
};

/**
 * Global broadcast descriptor. For simplicity we will have one single system
 * wide broadcast descriptor which can be used to broadcast messages. Also, we
 * are assuming that for the whole system execution one and only one process
```

```
 * will broadcast one and only one message. No listener process will call the
 * listen() primitive more than once.
 */
broadcast global_broadcast =
{
    0,  // registered
    0,  // nlisten
    0,  // msg
    0,  // listeners
    0   // broadcaster
};

/**
 * Implementation for the void reg() primitive. If the processes calling this
 * method is already registered to the global broadcast no action will be
 * performed.
 */
extern "C" void c_reg()
{
    // retrieve calling process descriptor, [0]
    struct des_proc *p = des_p(esecuzione->id);

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

    // check if the process is already a registered listener
    if (p->listen_reg)
    {
        // if so, just return: nothing to do
        return;
    }

    // otherwise, register the process to the listeners
    b->registered++;

    // set the process broadcast listener flag: this will be checked when the
    // process calls the listen() primitive
    p->listen_reg = true;
}
// EXTENSION 2016-09-20

// SOLUTION 2016-09-20

/**
 * Called when all registered listeners are ready to receive the broadcast
 * message. This happens when registered == nlisten for the global broadcast.
 * Sends the current broadcast message to all waiting processes.
 */
void broadcast_all()
{
    // retrieve pointer to the global broadcast descriptor
    struct broadcast *b = &global_broadcast;

    // process descriptor
    struct proc_elem *work;

    // while there are still listener processes in the queue
    while (b->listeners)
    {
        // remove top process from listeners wait queue
        rimozione_lista(b->listeners, work);

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

        // deliver broadcast message
        w->contesto[I_RAX] = b->msg;

        // place process in the system ready processes queue
        inserimento_lista(pronti, work);
```

```
    }

    // all process have received the broadcast message
    b->nlisten = 0;
}

/**
 * All registered listeners must call this method to receive the broadcast
 * message. If the broadcaster has already sent the broadcast message this
 * function will deliver it to the calling listener and remove it from the
 * listeners processes queue and check if all registered listeners have called
 * the listen() primitive. If so it will call the broadcast_all method.
 * Otherwise it will insert the current process in the system ready processes
 * queue and wait for the next listener to call the listen() primitive.
 */
extern "C" void c_listen()
{
    // retrieve calling process descriptor, [0]
    struct des_proc *p = des_p(esecuzione->id);

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

    // check if the calling process is registered as listener
    if (!p->listen_reg)
    {
        // print warning log message
        flog(LOG_WARN, "Process not registered as broadcast listener.");

        // abort current process under execution
        c_abort_p();

        // just return to the caller
        return;
    }

    // increase number of listeners awaiting broadcast message
    b->nlisten++;

    // if there is no process in the broadcaster queue yet: broadcast(natl msg)
    // not called yet
    if (!b->broadcaster)
    {
        // insert the process in the global broadcast listeners wait queue: a
        // new process is scheduled at the end
        inserimento_lista(b->listeners, esecuzione);
        }
    else
    {
        // otherwise, deliver the message to the current listener process
        p->contesto[I_RAX] = b->msg;

        // insert current process in the system ready processes list
        inserimento_lista(pronti, esecuzione);

        // check if all listener processes have called the listen() primitive
        if (b->nlisten == b->registered)
        {
            // if so, deliver broadcast message to all listener processes
            broadcast_all();

            // after that,we need a process descriptor
            struct proc_elem *work;

            // to retrieve the broadcaster process
            rimozione_lista(b->broadcaster, work);

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

```
        // if all the listener process have not called the listen() primitive we
        // will have to wait for the next listener process calling it and check
        // again if all listener processes are ready, deliver the broadcast
        // message to all of them and remove the broadcaster process from the
        // queue in order for it to be rescheduled
    }

    // schedule a new process
    schedulatore();
}

/**
 * Called by the broadcaster process to send the given broadcast message to all
 * registered listener processes. If all registered listener processes have
 * called the listen() primitive the broadcast_all method will deliver the
 * broadcast message to all of them. Otherwise, the broadcaster process will be
 * inserted in the broadcasyer wait queue waiting for all listeners to be ready.
 *
 * @param  msg  the message to be broadcasted.
 */
extern "C" void c_broadcast(natl msg)
{
    // retrieve calling process descriptor
    struct des_proc *p = des_p(esecuzione->id);

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

    // check if the process is not registered as listener
    if (p->listen_reg)
    {
        // print warning log message
        flog(LOG_WARN, "Listener process can not send broadcast messages.");

        // abort current process under execution
        c_abort_p();

        // return to the caller
        return;
    }

    // set broadcast message
    b->msg = msg;

    // check if all listeners have invoked the listen primitive
    if (b->nlisten == b->registered)
    {
        // if so, insert the current process under execution (the broadcaster)
        // in the system ready processes queue: a new process is scheduled at
        // the end
        inserimento_lista(pronti, esecuzione);

        // send broadcast message to all listeners
        broadcast_all();
        }
    else
    {
        // otherwise, wait for all listeners to be ready
        // insert current process in the broadcaster process queue
        inserimento_lista(b->broadcaster, esecuzione);
    }

    // schedule a new process
    schedulatore();
}

// SOLUTION 2016-09-20

// [...]
```

```
/**
 * The method used to destroy processes must be edited in order to check if the
 * process being destructed is a registered listener and if so remove it from
 * the global broadcast listeners.
 */
void distruggi_processo(proc_elem* p)
{
        des_proc* pdes_proc = des_p(p->id);

// EXTENSION 2016-09-20

    // check if the process is a registered listener
    if (pdes_proc->listen_reg)
    {
        // in that case, decrease registered processes
        global_broadcast.registered--;
    }

// 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);
}
```