Prova pratica di Calcolatori Elettronici

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

15 giugno 2016

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

```
struct st1 {
        char vc[8];
};
struct st2 {
        long vd[4];
};
class cl {
        st1 s;
        int v[4];
public:
        cl(char c, st1& s2);
        void elab1(st1 s1, st2& s2);
        void stampa()
        {
                for (int i = 0; i < 8 ;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 < 8; i++) {
                s.vc[i] = c + i;
        }
        for (int i = 0; i < 4; i++) {
                v[i] = s2.vc[i] - s.vc[i];
        }
}
void cl::elab1(st1 s1, st2& s2)
{
        cl cla('f', s1);
        for (int i = 0; i < 4; i++) {
                if (s.vc[i] < s1.vc[i])
                        s.vc[i] = cla.s.vc[i];
                if (v[i] < cla.v[i])
                        v[i] = cla.v[i] + i;
```

```
        }
}
```

2. Colleghiamo al sistema delle periferiche PCI di tipo `ce`, con vendorID `0xedce` e deviceID `0x1234`. Ogni periferica `ce` usa 16 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia b.

   Le periferiche `ce` sono periferiche di ingresso in grado di operare in PCI Bus Mastering. I registri accessibili al programmatore sono i seguenti:

   1. **BMPTR** (indirizzo b, 4 byte): puntatore al buffer di destinazione;
   2. **BMLEN** (indirizzo $b + 4$, 4 byte): numero di byte da trasferire;
   3. **CMD** (indirizzo $b + 8$, 4 byte): registro di comando;
   4. **STS** (indirizzo $b + 12$, 4 byte): registro di stato.

   L'interfaccia è in grado di trasferire in memoria BMLEN byte, partendo dall'indirizzo fisico contenuto in BMPTR e proseguendo agli indirizzi fisici contigui. Per iniziare il trasferimento è necessario scrivere 1 nel registro di comando. L'interfaccia invia una richiesta di interruzione dopo aver trasferito l'ultimo byte. Le interruzioni sono sempre abilitate e la lettura del registro di stato funziona da risposta alle richieste di interruzione (l'interfaccia non invia una nuova richiesta se una richiesta precedente non ha ancora avuto risposta).

   Vogliamo fornire all'utente una primitiva

   ```
   cedmaread(natl id, char *buf, natl quanti)
   ```

   che permetta di leggere `quanti` byte dalla periferica numero `id` (tra quelle di tipo `ce`) copiandoli nel buffer `buf`. Notare che `buf` è un indirizzo virtuale e il buffer potrebbe attraversare più pagine virtuali: la primitiva dovrà funzionare in ogni caso, eventualmente eseguendo più trasferimenti.

   Per descrivere le periferiche `ce` aggiungiamo le seguenti strutture dati al modulo I/O:

   ```
   struct des_ce {
           ioaddr iBMPTR, iBMLEN, iCMD, iSTS;
           natl sync;
           natl mutex;
           char *buf;
           natl quanti;
   };
   des_ce array_ce[MAX_CE];
   natl next_ce;
   ```

   La struttura `des_ce` descrive una periferica di tipo `ce` e contiene al suo interno gli indirizzi dei registri BMPTR, BMLEN, CND e STS, l'indice di un semaforo inizializzato a zero (`sync`), l'indice di un semaforo inizializzato a 1 (`mutex`), il numero di byte che restano da trasferire (`quanti`) e l'indirizzo virtuale a cui vanno trasferiti (`buf`).

   I primi `next_ce` elementi del vettore `array_ce` contengono i descrittori, opportunamente inizializzati, delle periferiche di tipo `ce` effettivamente rilevate in fase di avvio del sistema. Ogni periferica è identificata dall'indice del suo descrittore.

   Modificare i file `io.s` e `io.cpp` in modo da realizzare la primitiva come descritto.

   Nota: il modulo sistema mette a disposizione la primitiva

   ```
   addr trasforma(addr ind_virtuale)
   ```

   che restituisce l'indirizzo fisico che corrisponde all'indirizzo virtuale passato come argomento, nello spazio di indirizzamento del processo in esecuzione.

```
/**
 * File: cc.h
 *       Contains the declaration for the data structures used in the exercise.
 *
 * Author: Rambod Rahmani <rambodrahmani@autistici.org>
 *         Created on 24/08/2019.
 */

#include <iostream>

using namespace std;

struct st1
{
    char vc[8];
};

struct st2
{
    long vd[4];
};

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

public:
    cl(char c, st1 & s2);

    void elab1(st1 s1, st2 & s2);

    void stampa()
    {
        for (int i = 0; i < 8 ;i++)
        {
            cout << s.vc[i] << ' ';
        }
        cout << endl;

        for (int 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 24/08/2019.
 */

#include "cc.h"

cl::cl(char c, st1 & s2)
{
    for (int i = 0; i < 8; i++)
    {
        s.vc[i] = c + i;
    }

    for (int i = 0; i < 4; i++)
    {
        v[i] = s2.vc[i] - s.vc[i];
    }
}

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

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

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

```
#*****************************************************************************
# File: es1.s
#       Contains the Assembly translation for es1.cpp.
#
# Author: Rambod Rahmani <rambodrahmani@autistici.org>
#         Created on 24/08/2019.
#*****************************************************************************

#-----------------------------------------------------------------------------
.TEXT
.GLOBAL _ZN2clC1EcR3st1                                 # cl::cl(char c, st1 & s2)
#-----------------------------------------------------------------------------
# activation record:
# ------------------
#  &s2          -24
#  c            -16
#  i            -12
#  &this        -8
#  %rbp         0
#-----------------------------------------------------------------------------
_ZN2clC1EcR3st1:
# set stack locations labels
    .set  this, -8
    .set  i,    -12
    .set  c,    -16
    .set  s2,   -24

# 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, c(%rbp)
    movq %rdx, s2(%rbp)

# for loop 1 initialization
    movl $0, i(%rbp)                # i = 0

for1:
    cmpl $8, i(%rbp)                # check if i < 4
    jge  finefor1                   # end for loop (i >= 8)

# for loop 1 body
    movq   this(%rbp), %rdi         # this -> %rdi
    movslq i(%rbp), %rcx
    movb   c(%rbp), %al             # c -> %al
    addb   i(%rbp), %al             # c + i -> %al
    movb   %al, (%rdi, %rcx, 1)     # s.vc[i] = c + i

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

finefor1:

# for loop 2, initialization
    movl $0, i(%rbp)                # i = 0

for2:
    cmpl $4, i(%rbp)                # check if i < 4
    jge  finefor2                   # end for loop (i >= 4)

# for loop 2, body
    movq   this(%rbp), %rdi         # this -> %rdi
    movslq i(%rbp), %rcx            # i -> %rcx
    movq   s2(%rbp), %rsi           # &s2 -> %rsi

    movb   (%rsi, %rcx, 1), %al     # s.vc[i] -> %al
    subb   (%rdi, %rcx, 1), %al     # s2.vc[i] - s.vc[i] -> %al
```

```
    movsbl %al, %eax                # %al -> %eax
    movl   %eax, 8(%rdi, %rcx, 4)   # v[i] = s2.vc[i] - s.vc[i];

    incl   i(%rbp)                  # i++
    jmp    for2                     # loop again

finefor2:
    movq   this(%rbp), %rax         # return initialized object address
    leave                           # movq %rbp, %rsp; popq %rbp
    ret

#-----------------------------------------------------------------------------
.GLOBAL _ZN2cl5elab1E3st1R3st2              # void cl::elab1(st1 s1, st2 & s2)
#-----------------------------------------------------------------------------
# activation record:
# ------------------
#  i           -52
#  cla_s       -48
#  cla_v       -40
#  &s2         -24
#  s1 LSB      -16
#  s1 MSB      -12
#  &this       -8
#  %rbp        0
#-----------------------------------------------------------------------------
_ZN2cl5elab1E3st1R3st2:
# set stack locations labels
    .set this,  -8
    .set s1,    -16
    .set s2,    -24
    .set cla_v, -40
    .set cla_s, -48
    .set i,     -52

# prologue: activation frame
    pushq %rbp
    movq  %rsp, %rbp
    subq  $56, %rsp                 # reserve stack space for actual arguments

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

# prepare actual arguments to call constructor
    leaq  -48(%rbp), %rdi           # &cla
    movb  $'f', %sil                # 'f'
    leaq  s1(%rbp), %rdx            # s1
    call  _ZN2clC1EcR3st1           # cl cla('f', s1);

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

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

# for loop, body
if1:
    movslq i(%rbp), %rcx
    movq   this(%rbp), %rsi
    leaq   s1(%rbp), %rdi

    movb   (%rdi, %rcx, 1), %bl     # s1.vc[i] -> %bl
    movb   (%rsi, %rcx, 1), %al     # s.vc[i] -> %al
    cmpb   %al, %bl                 # if (s.vc[i] > s1.vc[i])
    jl     fineif1                  # exit if

# if1 body
    leaq   cla_s(%rbp), %rdi        # &cla.s.vc[i] -> %rsi
```

```
    movb   (%rdi, %rcx, 1), %al       # cla.s.vc[i] -> %cl
    movb   %al, (%rsi, %rcx, 1)       # s.vc[i] = cla.s.vc[i];

fineif1:

#if2:
    leaq  cla_v(%rbp), %rsi
    movl  (%rsi, %rcx, 4), %eax
    movq  this(%rbp), %rdi
    movl  8(%rdi, %rcx, 4), %ebx      # this.v[i] -> %ebx
    cmpl  %ebx, %eax                  # if (v[i] > cla.v[i])
    jl    fineif2                     # exit if

# if2 body
    addl  i(%rbp), %eax               # cla.v[i] + i -> %eax
    movl  %eax, 8(%rdi, %rcx, 4)

fineif2:
    incl  i(%rbp)                     # i++
    jmp   for                         # loop again

finefor:

    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 24/08/2019.
 */

#include "cc.h"

/**
 * Developer harness test.
 *
 * @param  argc  command line arguments counter.
 * @param  argv  command line arguments.
 *
 * @return       execution exit code.
 */
int main(int argc, char * argv[])
{
    st1 s1 = { 'e', 'b', 'f', 'd', 'a', 'r', 'x', 'i' };

    st1 s3 = { 'm', 'n', 'c', 'j', 's', 'h', 'u', 't' };

    st1 sa = { 1, 20, 3, 40 };

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

    cl cla('h', sa);

    cla.stampa();

    cla.elab1(s3, sb);

    cla.stampa();
}
```

```
h i j k l m n o
-103 -85 -103 -67

f g j k l m n o
7 8 -3 4
```

```
// EXTENSION 2016-06-15

    // prendiamo una pagina si' e una no, in modo che lo spazio
    // comune non sia contiguo in memoria fisica
    vdf = reinterpret_cast<des_frame*>(vdf_start);

    frame_liberi = &vdf[0];

    natl i;

    for (i = 0; i < N_DF; i += 2)
    {
        vdf[i].livello = -1;
        vdf[i].prossimo_libero = &vdf[i + 2];
    }

    vdf[i - 2].livello = -1;

    vdf[i - 2].prossimo_libero = 0;

// EXTENSION 2016-06-15
```

```
// EXTENSION 2016-06-15

/**
 * Interrupt type for the cedmaread() primitive.
 */
#define IO_TIPO_CEREAD      0x79

/**
 * Interrupt type for the cedmawrite() primitive.
 */
#define IO_TIPO_CEWRITE     0x7a

// EXTENSION 2016-06-15
```

```
// EXTENSION 2016-06-15

/**
 * Primitive declaration for the User Module. The definition in user/user.s only
 * calls an interrupt having the type declared in kernel/include/constants.h for
 * this primitive.
 */
extern "C" void cedmaread(natl id, char *buf, natl quanti);

// EXTENSION 2016-06-15
```

```
# EXTENSION 2016-06-15

#--------------------------------------------------------------------------------
.GLOBAL cedmaread
#--------------------------------------------------------------------------------
cedmaread:
    int $IO_TIPO_CEREAD
    ret

# EXTENSION 2016-06-15
```

```
# SOLUTION 2016-06-15

        fill_io_gate    IO_TIPO_CEREAD  a_cedmaread

# SOLUTION 2016-06-15

# SOLUTION 2016-06-15

# cedmaread(natl id, char * buf, natl quanti)
# id -> %rdi
# &buf -> %rsi
# quanti -> %rdx
.EXTERN c_cedmaread
a_cedmaread:
    cavallo_di_troia %rsi           # check the buffer starting address
    movl %edx, %edx                 # zero out %rdx upper 32 bits
    cavallo_di_troia2 %rsi %rdx     # check the entire buffer
    call c_cedmaread                # call C++ primitive implementation
    iretq                           # return from interrupt

# SOLUTION 2016-06-15
```

```
// EXTENSION 2016-06-15

/**
 * Maximum number of CE devices to be loaded at boot.
 */
static const int MAX_CE = 16;

/**
 * CE device descriptor.
 */
struct des_ce
{
    // destination buffer address
    ioaddr iBMPTR;

    // bytes to be transferred
    ioaddr iBMLEN;

    // command register: write 1 to start a transfer
    ioaddr iCMD;

    // statu register: reading it will answer the interrupt request
    ioaddr iSTS;

    // synchronization semaphore initialized to 0
    natl sync;

    // mutex: at any point of time, only one thread can work with the entire
    // buffer
    natl mutex;

    // virtual address of the destination buffer
    char *buf;

    // number of bytes to be transferred
    natl quanti;
};

/**
 * Descriptors of the CE devices actually loaded at boot.
 */
des_ce array_ce[MAX_CE];

/**
 * Number of the next CE device to be loaded.
 */
natl next_ce;

// EXTENSION 2016-06-15


// SOLUTION 2016-06-15
/**
 * Reads 'quanti' bytes from CE PCI device having the specfified 'id' into
 * 'buf'. Keep in mind that CE devices will send an interrupt request at the end
 * of each transfer. We will therefore have to make the first transfer right
 * here and wait and handle the CE device interrupt request in order to finish
 * the remaining transfers.
 *
 * @param  id      CE PCI device ID;
 * @param  buf     memory buffer where to store retrieved data;
 * @param  quanti  number of bytes to retrieve from the PCI device.
 */
extern "C" void c_cedmaread(natl id, char *buf, natl quanti)
{
    // check if the given CE PCI device id is valid
    if (id >= next_ce)
    {
        // if not, print a warning log message for the user
        flog(LOG_WARN, "CE Device %d does not exist.", id);
```

```
        // abort the current process under execution
        abort_p();
    }

    // retrieve selected CE device descriptor
    des_ce *c = &array_ce[id];

    // wait for the CE device mutex
    sem_wait(c->mutex);

    // retrieve physical address from virtual address for the destination buffer
    addr f = trasforma(buf);

    // get the number of bytes available in the frame containing the buffer
    natw rem = 4096 - ((natq)f & 0xfff);

    // if there are more bytes available in the frame than the ones to be
    // transferred
    if (rem > quanti)
    {
        // set number of bytes to be transferred to the remaning bytes available
        // in the frame
        rem = quanti;
    }

    // print debugging log message with transfer infos
    flog(LOG_DEBUG, "virtual %lx physical %lx first transfer: %d byte", buf, f, rem);

    // update CE device descriptor destination buffer pointer address: set value
    // after transfer
    c->buf = buf + rem;

    // update CE device descriptor number of bytes to be transferred: set value
    // after transfer
    c->quanti = quanti - rem;

    // write destination buffer physical address
    outputl((natq)f, c->iBMPTR);

    // write number of bytes to be trasferred
    outputl(rem, c->iBMLEN);

    // write to the command register: start transfer
    outputl(1, c->iCMD);

    // wait for the the sync semaphore: set by estern_ce when all transfers have
    // been completed
    sem_wait(c->sync);

    // notify CE device mutex
    sem_signal(c->mutex);
}

/**
 * Called everytime the CE device identified by id sends an interrupt request.
 * CE Devices send an interrupt request once they are done transferring the last
 * byte after the status register was set to 1 (start transfer command).
 *
 * This method checks if there are still bytes to be transferred from the device
 * in which case it starts an infinite loop transfering chunks of data of the
 * size of a page at each transfer.
 *
 * @param  id  external CE device id. This id is always good because the extern
 *             process was initialized when the CE device was first initialized.
 */
extern "C" void estern_ce(int id)
{
    // retrieve the CE device descriptor
    des_ce *c = &array_ce[id];
```

```
    // byte buffer to retrieve the status register
    natl b;

    // this infinite for loop is needed because once the wfi() is done sending
    // the EOI to the APIC it will also schedule a new process; when a new
    // interrupt request is received from this ce device this process will wake
    // up again and start from where it was ended: without the for loop the
    // function will just end resulting in a dead lock
    for (;;)
    {
        // read CE device status register into buffer b: interrupt ak
        inputl(c->iSTS, b);

        // check if there are still bytes to be transferred from the device
        if (c->quanti > 0)
        {
            // retrieve remaining number of bytes to be transferred
            natw rem = c->quanti;

            // check if there are more bytes than 4 Kib (page size)
            if (rem > 4096)
            {
                // if so, set next bytes to be transferred to 4 Kib
                rem = 4096;
            }

            // retrieve physical address from virtual address
            addr f = trasforma(c->buf);

            // print debugging log message with transfer infos
            flog(LOG_DEBUG, "virtual %lx physical %lx trasfer: %d byte", c->buf, f, rem);

            // update CE device descriptor destination buffer address pointer:
            // set value after current transfer
            c->buf += rem;

            // update CE device descriptor number of byte to be transferred:
            // set value after current transfer
            c->quanti -= rem;

            // write destination buffer physical address
            outputl((natq)f, c->iBMPTR);

            // write number of bytes to be transferred
            outputl(rem, c->iBMLEN);

            // write to the command register: start transfer
            outputl(1, c->iCMD);
        }
        else
        {
            // all bytes transferred, notify synchronization semaphore
            sem_signal(c->sync);
        }

        // send APIC EOI and schedule a new process
        wfi();
    }
}

// SOLUTION 2016-06-15

// EXTENSION 2016-06-15

/**
 * Called at the end of the I/O subsystem initialization, it initializes
 * the CE devices descriptors array.
 */
bool ce_init()
```

```
{
    // loop through the CE devices on the PCI bus
    for (natb bus = 0, dev = 0, fun = 0;
             pci_find_dev(bus, dev, fun, 0xedce, 0x1234);
             pci_next(bus, dev, fun)
        )
    {
        // check if the maximum number of devices is not exceeded
        if (next_ce >= MAX_CE)
        {
            // print warning message
            flog(LOG_WARN, "Too many CE devices.");

            // exit loop
            break;
        }

        // retrieve pointer to the next available CE device descriptor
        des_ce *ce = &array_ce[next_ce];

        // retrieve base register content
        ioaddr base = pci_read_confl(bus, dev, fun, 0x10);

        // set bit n. 0 to 0: retrieve base address
        base &= ~0x1;

        // set device destination buffer address: base address
        ce->iBMPTR = base;

        // set device number of transfer bytes: base + 4
        ce->iBMLEN = base + 4;

        // set command register address: base + 8
        ce->iCMD = base + 8;

        // set status register address: base + 12
        ce->iSTS = base + 12;

        // initialize sync semaphore to 0
        ce->sync = sem_ini(0);

        // initialize mutex to 1
        ce->mutex = sem_ini(1);

        // retrieve external device APIC ir pin
        natb irq = pci_read_confb(bus, dev, fun, 0x3c);

        // activate external device interrupt process
        activate_pe(estern_ce, next_ce, PRIO, LIV, irq);

        // log device info
        flog(LOG_INFO, "ce%d %2x:%1x:%1x base=%4x IRQ=%d", next_ce, bus, dev, fun, base,
irq);

        // increment CE devices counter
        next_ce++;
    }

    // return initialization successful
    return true;
}

// EXTENSION 2016-06-15

////////////////////////////////////////////////////////////////////////////
//                  INIZIALIZZAZIONE DEL SOTTOSISTEMA DI I/O                  //
////////////////////////////////////////////////////////////////////////////

// inizializza i gate usati per le chiamate di IO
//
```

```
extern "C" void fill_io_gates(void);

extern "C" natl end;
// eseguita in fase di inizializzazione
//
extern "C" void cmain(int sem_io)
{

        fill_io_gates();
        mem_mutex = sem_ini(1);
        if (mem_mutex == 0xFFFFFFFF) {
                flog(LOG_ERR, "impossible creare semaforo mem_mutex");
                abort_p();
        }
        unsigned long long end_ = (unsigned long long)&end;
        end_ = (end_ + DIM_PAGINA - 1) & ~(DIM_PAGINA - 1);
        heap_init((void *)end_, DIM_IO_HEAP);
        if (!console_init())
                abort_p();
        if (!com_init())
                abort_p();
        if (!hd_init())
                abort_p();

// EXTENSION 2016-06-15

    // initialize CE devices
    if (!ce_init())
    {
        // abort current process if the initialization does not succeed
        abort_p();
    }

// EXTENSION 2016-06-15

        sem_signal(sem_io);
        terminate_p();
}
```