Prova pratica di Calcolatori Elettronici

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

27 luglio 2016

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

```
struct st {
        char vv1[4];
        long vv2[4];
};
class cl {
        char a, b;
        st s;
public:
        cl();
        cl(char v[]);
        void elab1(st& ss, int d);
        void stampa()
        {
                cout << (int)a << ' ' << (int)b << endl;
                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() { }
cl::cl(char v[])
{
        a = v[0]++;
        b = v[1];
        for (int i = 0; i < 4; i++) {
                s.vv1[i] = v[i] + a;
                s.vv2[i] = v[i] + b;
        }
}
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] = a + d;
        }
}
```

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.

   La 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 ai descrittori di trasferimento;
   2. **CMD** (indirizzo $b + 4$, 4 byte): registro di comando;
   3. **STS** (indirizzo $b + 8$, 4 byte): registro di stato.

   La periferica accumula internamente dei byte da una fonte esterna e ogni volta che si scrive il valore 1 nel registro CMD cerca di trasferirli tutti in memoria. Non è possibile sapere *a-priori* il numero di byte disponibli all'interno della periferica I byte verranno trasferiti in una sequenza di zone di memoria descritte da un vettore di *descrittori di trasferimento*, il cui indirizzo è contenuto in BMPTR. Ciascun descrittore specifica un indirizzo fisico di partenza e una dimensione. La periferica userà tutte le zone in ordine, fino al trasferimento di tutti i byte disponibili al suo interno. È possibile che le zone non siano sufficienti, nel qual caso i byte in eccesso saranno persi. In ogni caso la periferica invia uma richiesta di interruzione al completamento dell'operazione (o perché non ha più byte da trasferire, o perché sono terminate le zone).

   Le interruzioni sono sempre abilitate. La lettura del registro di stato funziona da risposta alle richieste di interruzione.

   I descrittori di trasferimento hanno la seguente forma:

   ```
   struct ce_buf_des {
     natl addr;
     natl len;
     natb eod;
     natb eot;
   };
   ```

   Prima di avviare una operazione il campo `addr` deve contenere l'indirizzo fisico di una zona di memoria e `len` la sua dimensione in byte; il campo `eod` deve valere 1 se questo è l'ultimo descrittore. Al completamento dell'operazione la periferica scrive in `len` quanti byte della zona ha utlizzato e scrive 1 in `eot` se con questa zona è riuscita a completare il trasferimento di tutti i byte interni.

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

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

   che permette di leggere al massimo `quanti` byte dalla periferica numero `id` (tra quelle di questo tipo), copiandoli nel buffer `buf`. La primitiva scrive nel parametro `quanti` il numero di byte effettivamente letti. Inoltre, la primitiva restituisce `true` se il buffer è stato sufficiente a contenere tutti i byte da trasferire, e `false` altrimenti.

   È un errore se `buf` non è allineato alla pagina e se `quanti` è zero o è più grande di 10 pagine. In caso di errore la primitiva abortisce il processo chiamante. Controllare tutti i problemi di Cavallo di Troia.

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

```
struct des_ce {
        natw iBMPTR, iCMD, iSTS;
        natl sync;
        natl mutex;
        ce_buf_des buf_des[MAX_CE_BUF_DES];
} __attribute__((aligned(128)));
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, STS e RBR, l'indice di un semaforo inizializzato a zero (`sync`), l'indice di un semaforo inizializzato a 1 (`mutex`) e un vettore di descrittori di trasferimento.

I primi `next_ce` elementi del vettore `array_ce` contengono i destrittori, opportunamente inizializzati, delle periferiche di tipo `ce` effettivamente rilevate in fase di avvio del sistema. Ogni periferica è identificata dall'indice del suo descrittore. Durante l'inizializzazione, il registro BMPTR della periferica viene fatto puntare al campo `buf_des` del suo descrittore.

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 14/09/2019.
 */

#include <iostream>

using namespace std;

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

class cl
{
    char a, b;
    st s;

public:
    cl();

    cl(char v[]);

    void elab1(st& ss, int d);

    void stampa()
    {
        cout << (int)a << ' ' << (int)b << endl;

        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()
{ }

cl::cl(char v[])
{
    a = v[0]++;

    b = v[1];

    for (int i = 0; i < 4; i++)
    {
        s.vv1[i] = v[i] + a;
        s.vv2[i] = v[i] + b;
    }
}

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] = a + d;
    }
}
```

```
#*****************************************************************************
# 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)

# a = v[0]++;
    movl   $0, i(%rbp)               # i = 0
    movslq i(%rbp), %rcx             # i -> %rcx
    movb   (%rsi, %rcx, 1), %al      # v[0] -> %al
    movb   %al, (%rdi, %rcx, 1)      # a = v[0];
    incb   (%rsi, %rcx, 1)           # v[0]++

# b = v[1];
    incq %rcx
    movb (%rsi, %rcx, 1), %al        # v[1] -> %al
    movb %al, (%rdi, %rcx, 1)        # b = v[1];

# 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 $0, %r8                     # $0 -> %r8
    movb (%rdi, %r8, 1), %al         # a -> %al

    incq %r8                         # increment %r8
    movb (%rdi, %r8, 1), %bl         # b -> %bl

    addb (%rsi, %rcx, 1), %al        # v[i] + a -> %al
    movb %al, 8(%rdi, %rcx, 1)       # s.vv1[i] = v[i] + a, [0]

    addb   (%rsi, %rcx, 1), %bl      # v[i] + b -> %bl
    movsbq %bl, %rbx                 # %bl -> %rbx
    movq   %rbx, 16(%rdi, %rcx, 8)   # s.vv2[i] = v[i] + b;
```

```
    incl i(%rbp)
    jmp  for

finefor:

    movq  this(%rbp), %rax            # return initialized object address
    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 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, 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 for loop (i >= 4)

# for looop body
    movq   this(%rbp), %rdi
    movq   ss(%rbp), %rsi
    movslq i(%rbp), %rcx

# if (d >= ss.vv2[i])
    movq 8(%rsi, %rcx, 8), %rax     # ss.vv2[i] -> %rax
    cmpl %eax, d(%rbp)              # d >= ss.vv2[i]
    jl   fineif                     # exit if (d < ss.vv2[i])
    movb (%rsi, %rcx, 1), %al       # ss.vv1[i] -> %al
    addb %al, 8(%rdi, %rcx, 1)      # s.vv1[i] += ss.vv1[i];

fineif:
    movq   $0, %r8                  # 0 -> %r8
    movb   (%rdi, %r8, 1), %al      # a -> %al
    movsbl %al, %eax                # %al => %eax
    addl   d(%rbp), %eax            # d + a -> %eax
    movslq %eax, %rax               # %eax => %rax
    movq   %rax, 16(%rdi, %rcx, 8)  # s.vv2[i] = a + d;

    incl i(%rbp)
    jmp  for1

finefor1:
```

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

#*****************************************************************************

#############################################################################
# [0]
# When it comes to structs memory alignment the following criteria is applied:
# "In general, a struct instance will have the alignment of its widest scalar
#  member. Compilers do this as the easiest way to ensure that all the members
#  are self-aligned for fast access".
# In this case the struct will have 8-byte alignment because it contains a field
# of type long.
#
# Primitive data types typical alignments:
#  - A char (one byte) will be 1-byte aligned.
#  - A short (two bytes) will be 2-byte aligned.
#  - An int (four bytes) will be 4-byte aligned.
#  - A long (eight bytes) will be 8-byte aligned.
#  - Any pointer (eight bytes) will be 8-byte aligned.
#############################################################################
```

```
/**
 * 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();
}
```

```
10 11
21 21 22 23     22 22 23 24

10 11
22 23 22 23     12 12 12 12
```

```
// EXTENSION 2016-07-27

/**
 * Interrupt type for the cedmaread primitive.
 */
#define IO_TIPO_CEDMAREAD       0x79

// EXTENSION 2016-07-27
```

```
// EXTENSION 2016-07-27

/**
 * This user primitive can be used to retrieve into the addressed buffer 'buf'
 * a maximum number of specified bytes (quanti) - as the CE device might contain
 * more available bytes which will therefore be lost - from the CE device having
 * the given id. It must check for the given buffer address to be aligned to the
 * page, for 0 <= quanti <= 10*pages. In case of error the primitive must abort
 * the calling process.
 *
 * @param  id      CE device id;
 * @param  quanti  number of bytes to be transferred;
 * @param  buf     destination buffer address;
 *
 * @return  true if the given buffer was big enough for all available bytes,
 *          false otherwise.
 */
extern "C" bool cedmaread(natl id, natl& quanti, char *buf);
extern "C" bool cedmaread2(natl id, natl* quanti, char *buf);

// EXTENSION 2016-07-27
```

```
# EXTENSION 2016-07-27

#-------------------------------------------------------------------------------
.GLOBAL cedmaread, cedmaread2
#-------------------------------------------------------------------------------
cedmaread:
cedmaread2:
        int $IO_TIPO_CEDMAREAD
        ret

# EXTENSION 2016-07-27
```

```
# SOLUTION 2016-07-27

    # set interrupt handler
    fill_io_gate        IO_TIPO_CEDMAREAD       a_cedmaread

# SOLUTION 2016-07-27

# SOLUTION 2016-07-27

#--------------------------------------------------------------------------------
.EXTERN c_cedmaread                         # C++ implementation for a_cedmaread
#--------------------------------------------------------------------------------

a_cedmaread:                                # IO_TIPO_CEDMAREAD interrupt handler
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    cavallo_di_troia %rsi             # check buffer length reference address
    cavallo_di_troia2 %rsi $4         # 'quanti' need 4 bytes
    cavallo_di_troia %rdx             # check destination buffer starting address
    movl (%rsi), %r9d                 # move destination buffer length to %r9d
    cavallo_di_troia2 %rdx %r9        # check destination buffer length
    call c_cedmaread                  # call C++ interrupt handler implementation
    iretq                             # return from interrupt
    .cfi_endproc

# SOLUTION 2016-07-27
```

```
//EXTENSION 2016-07-27

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

/**
 * The provided buffer for the cedmaread primitive must be aligned to its page,
 * the number of pages to be transferred must be greater than 0 and smaller
 * than 10 pages.
 */
static const int MAX_CE_BUF_DES = 10;

/**
 * The cedmaread primitive will transfer the available bytes to the memory
 * spaces addressed by a vector of transfer descriptors. The address of the
 * vector must be placed in the CE device BMPTR register.
 *
 * Destination buffer descriptor for CE device transfers.
 */
struct ce_buf_des
{
    // memory location physical address
    natl addr;

    // memory location length
    natw len;

    // set to 1 if this is the last descriptor
    natb eod;

    // set to 1 by the CE device if all internal bytes have been transferred
    // till this buffer descriptor
    natb eot;
};

/**
 * CE Device descriptor.
 *
 * Ce devices are capable of working in bus mastering. Each device stores a
 * certain amount if bytes and when the CMD register is set to 1 it will try
 * and move them to the memory space in Bus Mastering (DMA). It is not possible
 * to know the number of stored bytes. The bytes will be transferred to a
 * sequence of memory locations addressed by a vector of transfers descriptors
 * (ce_buf_des) addressed in BMPTR. Each descriptor must provide a starting
 * physical destination address and a length. The device will entirely use all
 * available memory locations until all its internal bytes have been
 * transferred. If the provided transfer descriptors does not provide enough
 * memory locations for all the available bytes, the remaining data will be lost
 * in the transfer. Anyway, the CE device will send an interrupt request the
 * transfer operation is complited (either because there are no more bytes to
 * be transferred or memory locations available). Interrupt requests will always
 * be enabled and reading from the status register will work as interrupt ak.
 */
struct des_ce
{
    // BMPTR register address
        natw iBMPTR;

    // command register address
    natw iCMD;

    // status register address
    natw iSTS;

    // synchronization semaphore
        natl sync;

    // mutex semaphore
```

```
        natl mutex;

    // destination buffers descriptors
        ce_buf_des buf_des[MAX_CE_BUF_DES];

} __attribute__((aligned(128)));

/**
 * Initialized CE devices decriptors.
 */
des_ce array_ce[MAX_CE];

/**
 * Number of initialized CE devices.
 */
natl next_ce;

// EXTENSION 2016-07-27

// SOLUTION 2016-07-27

/**
 * Starts the transfers from the CE device having the given ID. The transfers
 * are executed by the PCI device in Bus Mastering (DMA) using the given buffer
 * descriptor array. The CE device will send an interrupt request when the
 * transfers are done. The estern_ce method will handle such interrupt request
 * and set the synchronization semaphore.
 *
 * When the synchronization semaphore is set, al DMA transfers have been
 * completed and we can loop through available buffer descriptors to
 * count the number of bytes actually transferred and check if any of them
 * contains the eot flag.
 *
 * The user will provide a virtual address in 'buf' and a number of bytes to be
 * read. We will have to create the buffer descriptors (ce_buf_des) array
 * manually.
 */
extern "C" bool c_cedmaread(natl id, natl& quanti, char *buf)
{
    // check if the given id is valid
    if (id >= next_ce)
    {
        // print warning log message
        flog(LOG_WARN, "CE device not found: %d", id);

        //
        abort_p();
    }

    // check if the buffer is aligned to the page: to check the alignment we use
    // a bitwise AND wich will return true if at least one of the last 12 least
    // significant bits is not equal to zero in which case the given address is
    // not a multiple of 4096 (the page size)
    if ((natq)buf & 0xfff)
    {
        // print warning log message
        flog(LOG_WARN, "Address %x not aligned to the page.", buf);

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

    // check if the number of bytes to be transferred is greater than zero and
    // smaller than 10 pages
    if (quanti == 0 || quanti > MAX_CE_BUF_DES * 4096)
    {
        // if so, print warning log message
        flog(LOG_WARN, "Invalid value for transfer bytes: %d", quanti);

        // abort current process under execution
```

```
        abort_p();
    }

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

    // wait for CE device mutex semaphore
    sem_wait(ce->mutex);

    // print log message for debugging purposes
    flog(LOG_DEBUG, "virt %p len %d", buf, quanti);

    // last buffer descriptor index
    int i;

    // loop through available buffer descriptors and until the number of bytes
    // to be transferred is reached
    for (i = 0; i < MAX_CE_BUF_DES && quanti; i++)
    {
        // retrieve number of bytes to be transferred
        natw len = quanti;

        // check if len is not bigger than the page size
        if (len > 4096)
        {
            // otherwise decrease it to the page size
            len = 4096;
        }

        // set i-th buffer descriptor physical address
        ce->buf_des[i].addr = (natq)trasforma(buf);

        // set i-th buffer descriptor transfer length (bytes)
        ce->buf_des[i].len = len;

        // set i-th buffer descriptor eot and eod to 0
        ce->buf_des[i].eot = ce->buf_des[i].eod = 0;

        // decrease number of bytes to be transferred
        quanti -= len;

        // increase buffer virtual address by the bytes transferred
        buf += len;

        // print log message for debugging purposes
        flog(LOG_DEBUG, "des[%d] addr %x len %d", i, ce->buf_des[i].addr, ce->buf_des[i].
len);
    }

    // set the last buffer descriptor eod
    ce->buf_des[i - 1].eod = 1;

    // write to the command register: start transfer in BUS Mastering: DMA
    outputl(1, ce->iCMD);

    // wait for the synchronization semaphore: set by estern_ce
    sem_wait(ce->sync);

    // clear bytes to be transferred
    quanti = 0;

    //
    int j;

    // completition flag
    bool complete = false;

    // loop through CE device available buffer descriptors
    for (j = 0; j < i; j++)
    {
```

```
        // count transferred bytes for each buffer descriptor to be returned to
        // the caller
        quanti += ce->buf_des[j].len;

        // check if the eot is set (all bytes available transferred)
        if (ce->buf_des[j].eot)
        {
            // set completition flag to be returned
            complete = true;

            // exit for loop
            break;
        }
    }

    // notify mutex semaphore
    sem_signal(ce->mutex);

    // return completition flag
    return complete;
}

/**
 * Called everytime an interrupt request from the CE device having the given id
 * is accepted.
 *
 * @param  id  CE device id.
 */
extern "C" void estern_ce(int id)
{
    // retrieve CE device descriptor
    des_ce *ce = &array_ce[id];

    // input byte buffer
    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: interrupt request ak
        inputl(ce->iSTS, b);

        // notify synchronization sempahore: all transfers completed
        sem_signal(ce->sync);

        // send EOI to the APIC and schedule a new process
        wfi();
    }
}
// SOLUTION 2016-07-27

// EXTENSION 2016-07-27

/**
 * Initializes the CE device. Called at the end of the I/O module
 * initialization.
 *
 * Loops through all PCI devices available on bus 0 and looks for those having
 * vendor ID 0xedce and device ID 0x1234. A maximum of MAX_CE devices can be
 * initialized: the remaining ones will simply be ignored.
 */
bool ce_init()
{
    // loop through PCI bus device having the required vendor and device id
    for (natb bus = 0, dev = 0, fun = 0;
         pci_find_dev(bus, dev, fun, 0xedce, 0x1234);
```

```
         pci_next(bus, dev, fun)
        )
    {
        // check if more CE devices can be initialized
        if (next_ce >= MAX_CE)
        {
            // print warning lo message: maximum number of CE devices exceeded
            flog(LOG_WARN, "Too many CE devices.");

            // exit for loop
            break;
        }

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

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

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

        // set BMPTR register address: base address
        ce->iBMPTR = base;

        // set command register address: base address + 4
        ce->iCMD = base + 4;

        // set status register address: base address + 8
        ce->iSTS = base + 8;

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

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

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

        // retrieve physical address of the destination buffers descriptors
        addr iff = trasforma(&ce->buf_des[0]);

        // write destination buffers descriptor to the BMPTR register
        outputl(reinterpret_cast<natq>(iff), ce->iBMPTR);

        // activate external process for the APIC pin
        activate_pe(estern_ce, next_ce, PRIO, LIV, irq);

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

        // increase CE device counter
        next_ce++;
    }

    // return true: initialization succeeded
    return true;
}

// EXTENSION 2016-07-27

////////////////////////////////////////////////////////////////////////////
//                  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 2016-07-27

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

// EXTENSION 2016-07-27

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