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