# 2023-07-19

## Dispositivo di ingresso virtualizzato

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 semplici periferiche con un registro RBR di ingresso, tramite il quale è possibile leggere il prossimo byte disponibile. Se abilitata scrivendo 1 nel registro CTL, la perifica invia una richiesta di interruzione quando il registro RBR contiene un nuovo byte. La periferica non invia nuove richieste di interruzione fino a quando il registro RBR non viene letto.

Nel sistema è installata un'unica perififerica `ce`, ma vogliamo fare in modo che gli utenti possano usarne diverse versioni "virtuali", dette `vce`. Ciascuna periferica `vce` contiene un buffer (realizzato con un array circolare) che può contenere un certo numero di byte letti dalla periferica `ce` (al massimo `VCE_BUFSIZE`). Ad ogni istante una sola periferica `vce` è *attiva*. Le interruzioni della periferica `ce` sono sempre abilitate e il modulo I/O provvede a leggere i byte e a depositarli nel buffer della `vce` attiva (se il buffer è pieno il byte letto viene perso).

Ciascuna `vce` è identificata da un numero da 0 a `VCE_NUM` meno 1. Gli utenti possono usare la primitiva

```
void vceread_n(natl vn, char *vetti, natq quanti)
```

per leggere `quanti` byte dalla `vce` numero `vn`, ricevendoli nel vettore `vetti`. La periferica copierà i byte dal buffer interno della `vce`, sospendendosi fino a quando non sono stati ricevuti tutti. I processi possono accedere concorrentemente a `vce` diverse, ma solo uno alla volta su ciascuna `vce`. In qualunque momento, un processo può cambiare la `vce` attiva invocando la primitiva `vceswitch(vn)`.

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

```
struct vce_buf {
    /// Array circolare di byte
    char buf[VCE_BUFSIZE];
    /// indice della testa dell'array circolare
    natl head;
    /// indice della coda dell'array circolare
    natl tail;
    /// numero di byte contenuti nell'array circolare
    natl n;
    /// indice del semaforo per l'accesso in mutua esclusione al descrittore
    natl mutex;
};
```

```
bool vce_buf_write(vce_buf *vb, char c)
{
    bool r = false;
    sem_wait(vb->mutex);
    if (vb->n < VCE_BUFSIZE) {
        vb->buf[vb->tail] = c;
        vb->tail = (vb->tail + 1) % VCE_BUFSIZE;
        vb->n++;
        r = true;
    }
    sem_signal(vb->mutex);
    return r;
}

struct vce_des {
    /// buffer interno
    vce_buf buf;
    /// indice del semaforo di mutua esclusione nell'accesso alla VCE
    natl mutex;
    /// indice di un semaforo di sincronizzazione
    natl sync;
};

struct ce_des {
    /// Descrittori delle VCE di questa periferica
    vce_des vces[VCE_NUM];
    /// indice della VCE attiva
    natl active;

    ioaddr iRBR,    ///< registro di lettura
           iCTL;    ///< registro di controllo
    /// indice di un semaforo di mutua esclusione nell'accesso alla periferica
    natl mutex;
};
ce_des ce;
```

La struttura `vce_buf` descrive i buffer interni alle `vce`: `buf` è l'array circolare; i byte vanno inseriti all'indice `tail` ed estratti dall'indice `head`; il campo `n` conta il numero di byte contenuti nell'array; il campo `mutex` è l'indice di un semaforo di mutua esclusione per gli accessi ai campi del `vce_buf`.

La struttura `vce_des` descrive una periferica `vce`: `buf` è il buffer interno; `mutex` è l'indice di un semaforo di mutua esclusione per l'utilizzo del `vce` e `sync` è un semaforo che contiene un gettone per ogni byte contenuto nel buffer.

Infine, la struttura `ce_des` descrive la periferica `ce`: il campo `vces` contiene un descrittore per ogni `vce`; il campo `active` è l'indice della `vce` attiva; i campi

`iRBR` e `iCTL` contengono gli indirizzi dei rispettivi registri; il campo `mutex` è l'indice di un semaforo di mutua esclusione per l'accesso al campo `active`.

```
extern "C" void c_vceswitch(natl vn)
{
    if (vn >= VCE_NUM) {
        flog(LOG_WARN, "vce %d non valido", vn);
        abort_p();
    }

    sem_wait(ce.mutex);
    ce.active = vn;
    sem_signal(ce.mutex);
}
```

```
.global fill_io_gates
fill_io_gates:
    ...
    fill_io_gate    IO_TIPO_VS  a_vceswitch
    fill_io_gate    IO_TIPO_VR  a_vceread_n

.extern c_vceswitch
a_vceswitch:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_vceswitch
    iretq
    .cfi_endproc

.extern c_vceread_n
a_vceread_n:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_vceread_n
    iretq
    .cfi_endproc
```

Modificare i file `io.s` e `io.cpp` in modo da realizzare le parti mancanti.

```
void vce_buf_read(vce_buf *vb, char *vetti)
{
    sem_wait(vb->mutex);
    if (vb->n > 0) {
        *vetti = vb->buf[vb->head];
```

```
        vb->head = (vb->head + 1) % VCE_BUFSIZE;
        vb->n--;
    }
    sem_signal(vb->mutex);
}

extern "C" void c_vceread_n(natl vn, char *vetti, natq quanti)
{
    if (vn >= VCE_NUM) {
        flog(LOG_WARN, "vce %d non valido", vn);
        abort_p();
    }

    if (!access(vetti, quanti, true, false)) {
        flog(LOG_WARN, "parametri non validi: %p, %lu", vetti, quanti);
        abort_p();
    }

    vce_des *v = &ce.vces[vn];
    sem_wait(v->mutex);
    for (natl i = 0; i < quanti; i++) {
        sem_wait(v->sync);
        vce_buf_read(&v->buf, vetti);
        vetti++;
    }
    sem_signal(v->mutex);
}

extern "C" void estern_ce(natq)
{
    for (;;) {
        char c = inputb(ce.iRBR);
        sem_wait(ce.mutex);
        vce_des *v = &ce.vces[ce.active];
        sem_signal(ce.mutex);
        if (vce_buf_write(&v->buf, c))
            sem_signal(v->sync);
        wfi();
    }
}

bool ce_init()
{
    bool found = false;
    for (natb bus = 0, dev = 0, fun = 0;
         pci::find_dev(bus, dev, fun, 0xedce, 0x1234);
         pci::next(bus, dev, fun))
```

```
{
    if (found) {
        flog(LOG_WARN, "troppi dispositivi ce");
        break;
    }
    found = true;
    natw base = pci::read_confl(bus, dev, fun, 0x10);
    natb irq = pci::read_confb(bus, dev, fun, 0x3c);
    base &= ~0x1;
    ce.iRBR = base;
    ce.iCTL = base + 4;
    flog(LOG_INFO, "ce: %02x:%02x.%1x base=%04x IRQ=%u",
            bus, dev, fun, base, irq);

    ce.mutex = sem_ini(1);
    if (ce.mutex == 0xFFFFFFFF) {
        flog(LOG_ERR, "semafori insufficienti");
        return false;
    }
    ce.active = 0;
    for (natq i = 0; i < VCE_NUM; i++) {
        vce_des *v = &ce.vces[i];
        v->mutex = sem_ini(1);
        if (v->mutex == 0xFFFFFFFF) {
            flog(LOG_ERR, "semafori insufficienti");
            return false;
        }
        v->sync = sem_ini(0);
        if (v->sync == 0xFFFFFFFF) {
            flog(LOG_ERR, "semafori insufficienti");
            return false;
        }
        v->buf.mutex = sem_ini(1);
        if (v->buf.mutex == 0xFFFFFFFF) {
            flog(LOG_ERR, "semafori insufficienti");
            return false;
        }
        v->buf.head = v->buf.tail = v->buf.n = 0;
    }
    if (activate_pe(estern_ce, 0, MIN_EXT_PRIO + INTR_TIPO_CE, LIV_SISTEMA, irq) == 0xFF
        flog(LOG_ERR, "impossibile attivare processo esterno");
        return false;
    }
    outputb(1, ce.iCTL);
}
return true;
```

```
}
```