2025-01-27 - Periferiche `ce` virtuali

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 di uscita con un registro TBR (Transmit Buffer Register), nel quale è possibile scrivere il prossimo byte da inviare. La perifica invia una richiesta di interruzione quando ha concluso la tramissione del contenuto di TBR; fino ad allora, il registro TBR è occupato e ulteriori scritture vengono ignorate.

Nel sistema è installata un'unica perififerica `ce`, ma vogliamo fare in modo che gli utenti possano usarne diverse versioni "virtuali", dette VCE, per non dover attendere il completamento delle trasmissioni. Ciascuna periferica VCE contiene un buffer (realizzato con un array circolare) che può contenere un certo numero di byte (al massimo `VCE_BUFSIZE`) in attesa di essere trasmessi sulla periferica reale. Dopo la conclusione di ogni trasmissione, il processo esterno associato alla periferica estrae un byte da una delle VCE attive e lo trasmette.

Le VCE sono private per processo. I processi che vogliono usare una VCE devono prima allocare la propria istanza usando la primitiva:

```
bool vcenew()
```

La primitiva restituisce `false` se non è stato possibile allocare la VCE. A quel punto il processo può usare la primitiva

```
void vcewrite(char c)
```

per inviare un byte nella VCE. La primitiva non attende la conclusione del trasferimento, ma può bloccare il processo in attesa che si liberi spazio per il byte nel buffer della VCE. La primitiva `vcenew()` abortisce il processo se questo aveva già una VCE; la primitiva `vcewrite()` lo abortisce se non ce l'aveva.

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

```
struct vce_des {
    char buf[VCE_BUFSIZE];
    natl head;
    natl tail;
    natl n;
    natl sync;
    bool waiting;
    bool terminated;
};

bool vce_des_init(vce_des *v)
{
```

```
    v->head = v->tail = v->n = 0;
    if ( (v->sync = sem_ini(0)) == 0xFFFFFFFF ) {
        flog(LOG_WARN, "vce: semafori terminati");
        return false;
    }
    v->waiting = v->terminated = false;
    return true;
}

bool vce_des_write(vce_des *v, char c)
{
    while (v->n == VCE_BUFSIZE)
        return false;
    v->buf[v->tail] = c;
    v->tail = (v->tail + 1) % VCE_BUFSIZE;
    v->n++;
    return true;
}

natl vce_des_read(vce_des *v)
{
    natl c = 0xFFFFFFFF;
    if (v->n > 0) {
        c = v->buf[v->head];
        v->head = (v->head + 1) % VCE_BUFSIZE;
        v->n--;
    }
    return c;
}

natl getpid()
{
    return getmeminfo().pid;
}

struct ce_des {
    vce_des *vces[MAX_PROC];
    bool busy;
    ioaddr iTBR;
    natl mutex;
} ce;

vce_des* vce_des_next(natl& current)
{
    for (natl i = 0; i < MAX_PROC; i++) {
        current = (current + 1) % MAX_PROC;
```

```
        vce_des *v = ce.vces[current];
        if (v && v->n > 0)
            return v;
    }
    return nullptr;
}
```

La struttura `vce_des` 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; quando un processo vuole inserire un byte, ma il buffer è pieno, pone `waiting` a `true` e si sospende sul semaforo di sincronizzazione `sync`; il campo `terminated` è true se il processo propietario della VCE è terminato.

La struttura `ce_des` descrive la periferica `ce`: l'array `vces`, indicizzato dai pid dei processi, contiene i puntatori alla `vce_des` di ogni processo (`nullptr` per i processi che non esistono o non hanno una VCE); il campo `busy` è `true` quando c'è una trasmissione in corso; il campo `iTBR` contiene l'indirizzo del registro TBR; il campo `mutex` è l'indice di un semaforo di mutua esclusione per l'accesso al `ce_des` e a tutte le VCE. Per permettere ai vari processi di usare le VCE mentre è in corso una tramissione precedente, il `mutex` deve essere occupato solo per il tempo strettamente necessario.

Le VCE dei processi devono essere allocate nello heap del modulo I/O (tramite `new`) e deallocate (tramite `delete`) quando il loro processo proprietario termina. L'allocazione avviene nella `vcenew()`, ma la responsabilità della deallocazione ricade sul processo esterno associato alla periferica `ce`: il processo esterno deve deallocare la VCE quando il processo proprietario e terminato e tutti i byte contenuti nella VCE sono stati tramessi. Per sapere quando il proprietario termina, il processo esterno sfrutta un meccanismo di notifica che è stato aggiunto al modulo sistema. Al momento dell'attivazione, il processo è stato registrato (tramite un nuovo parametro della `activate_pe()`) per la ricezione di notifiche di terminazione. Da quel momento in poi, ogni volta che un processo utente termina, il modulo sistema notificherà il processo esterno, in particolare rimettendolo in esecuzione se si era bloccato nella `wfi()`. La prima volta che va in esecuzione, e ogni volta che si sveglia dalla `wfi()`, il processo esterno deve invocare la primitiva `evget()` per sapere come mai è stato risvegliato. La primitiva restituisce 0 in caso di richiesta di interruzione (caso normale) e un valore maggiore di zero in caso di notifica di terminazione. Nel secondo caso, il valore restituito è il pid del processo terminato (se si chiama ripetutamente `evget()` senza ripassare dalla `wfi()`, la primitiva restituisce `0xFFFFFFFF`).

```
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.iTBR = base;
        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.busy = false;
        for (natl i = 0; i < MAX_PROC; i++) {
            ce.vces[i] = nullptr;
        }
        if (activate_pe(estern_ce, 0, MIN_EXT_PRIO + INTR_TIPO_CE, LIV_SISTEMA, irq, true) =
            flog(LOG_ERR, "impossibile attivare processo esterno");
            return false;
        }
    }
    return true;
}
```

Modificare il `io.cpp` in modo da realizzare le parti mancanti.

```
extern "C" bool c_vcenew()
{
    sem_wait(ce.mutex);
    natl pid = getpid();
    if (ce.vces[pid] != nullptr) {
        flog(LOG_ERR, "vcenew: il processo possiede già una VCE");
        sem_signal(ce.mutex);
        abort_p();
    }

    vce_des *v = new vce_des;
    if (v == nullptr) {
        flog(LOG_WARN, "vcenew: memoria esaurita");
        sem_signal(ce.mutex);
        return false;
```

```
    }
    if (!vce_des_init(v))  {
        delete v;
        sem_signal(ce.mutex);
        return false;
    }
    ce.vces[pid] = v;
    sem_signal(ce.mutex);
    return true;
}

extern "C" void c_vcewrite(char c)
{
    natl pid = getpid();
    sem_wait(ce.mutex);
    vce_des *v = ce.vces[pid];
    if (!v) {
        flog(LOG_WARN, "vcewrite: il processo non ha una VCE");
        sem_signal(ce.mutex);
        abort_p();
    }
    // possiamo proseguire se la periferica è libera o se c'è posto nella VCE,
    // altrimenti dobbiamo bloccare il processo e riprovare
    while (ce.busy && v->n == VCE_BUFSIZE) {
        v->waiting = true;
        // liberiamo il des_ce per permettere ad altri di usarlo
        sem_signal(ce.mutex);
        // blocchiamo il processo
        sem_wait(v->sync);
        // qualcuno ci ha svegliati: riacquisiamo la mutua esclusione
        // e vediamo se ora è possibile proseguire
        sem_wait(ce.mutex);
    }
    if (!ce.busy) {
        // se non ci sono trasferimenti in corso, inviamo il byte
        // direttamente
        outputb(c, ce.iTBR);
        ce.busy = true;
    } else {
        vce_des_write(v, c);
    }
    sem_signal(ce.mutex);
}

extern "C" void estern_ce(natq)
{
```

```
    natl current = 0;
    for (;;) {
        natl pid = evget();
        sem_wait(ce.mutex);
        if (pid) {
            vce_des *v = ce.vces[pid];
            if (v) {
                v->terminated = true;
                if (!v->n) {
                    delete v;
                    ce.vces[pid] = nullptr;
                }
            }
        } else {
            ce.busy = false;
            if (vce_des *v = vce_des_next(current)) {
                char c = vce_des_read(v);
                outputb(c, ce.iTBR);
                ce.busy = true;
                if (v->waiting) {
                    v->waiting = false;
                    sem_signal(v->sync);
                } else if (!v->n && v->terminated) {
                    delete v;
                    ce.vces[current] = nullptr;
                }
            }
        }
        sem_signal(ce.mutex);
        wfi();
    }
}
```