Prova pratica di Calcolatori Elettronici

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

10 settembre 2025

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

```
struct st1 { char vc[4]; }; struct st2 { char vd[4]; };
class cl {
        st1 c1;
        long v[4];
        st1 c2;
public:
        cl(char c, st2& s);
        void elab1(st1 s1, st2 s2);
        void stampa()
        {
                for (int i=0; i < 4; i++) cout << v[i] << ' '; cout << "\n";
                for (int i=0; i < 4; i++) cout << c1.vc[i] << ' '; cout << "\n";
                for (int i=0; i < 4; i++) cout << c2.vc[i] << ' '; cout << "\n\n";
        }
};
```

   Realizzare in Assembler GCC le funzioni membro seguenti.

```
cl::cl(char c, st2& s2) {
        for (int i = 0; i < 4; i++) {
                c1.vc[i] = c; c2.vc[i] = c++;
                v[i] = s2.vd[i] + c2.vc[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 semplici periferiche di ingresso con un registro RBR (Receive Buffer Register), dal quale è possible leggere un byte. La periferica invia una richiesta di interruzione quando dispone di un nuovo byte; la richiesta rimane attiva fino a quando il byte non viene letto.

   Vogliamo fornire agli utenti un meccanismo di lettura *asincrono*. Gli utenti devono invocare una primitiva `ceasyncread_n(id, buf, quanti)` che ordina il trasferimento di `quanti` byte dalla periferica CE `id` al buffer `buf`. La primitiva non attende che il trasferimento sia concluso (e nemmeno che la periferica sia libera), ma si limita ad accodare la richiesta in una coda di trasferimenti della periferica e a restituire al chiamante un *identificatore di trasferimento*, `trid`. Il trasferimento verrà avviato non appena la periferica è libera. Nel frattempo, il processo utente può proseguire con altre azioni (eventualmente,

anche con altre richieste di trasferimento, anche sulla stessa periferica). In un secondo momento, un processo (eventualmente anche diverso da quello che aveva iniziato il trasferimento) potrà attendere il termine del trasferimento invocando `cewait(id, trid)`. I trasferimenti sono avviati nell'ordine in cui sono stati richiesti, ma i processi possono attendere la loro terminazione in qualunque ordine.

Per realizzare il meccanismo definiamo le seguenti strutture dati:

```
struct des_ce_tr {
    char *buf;
    natl quanti;
    natl next;
};

struct des_ce {
    ioaddr iCTL, iSTS, iRBR;
    des_ce_tr tr[MAX_CE_ASYNC];
    natl cur;
    natl last;
    natl waiting;
    natl sync;
    natl mutex;
};
```

Il campo `iCTL` contiene l'indirizzo del registro di controllo della periferica. Il bit 1 del registro abilita (se settato) o disabilita (se resettato) la periferica a inviare richieste di interruzione. Il campo `iSTS` contiene l'indirizzo del registro di stato, che possiamo ignorare. Il campo `iRBR` contiene l'indirizzo del registro di ingresso.

L'array `tr` contiene dei descrittori di trasferimento con indici da 0 a `MAX_CE_ASYNC-1`. L'indice all'interno dell'array `tr` funge da identificatore di trasferimento. Il campo `cur` contiene l'indice del trasferimento attivo, o `0xFFFFFFFF` se non vi sono trasferimenti attivi. Ogni descrittore `des_ce_tr` contiene il puntatore `buf` alla destinazione del prossimo byte da leggere, il numero `quanti` di byte ancora da leggere, e l'indice `next` del trasferimento da iniziare al termine di questo (`0xFFFFFFFF` se non ve ne sono). Il capo `last` contiene l'indice dell'ultimo trasferimento in `tr`, in ordine di richiesta. Un descrittore di trasferimento è considerato libero se `buf` è `nullptr`. Se tutti i descrittori sono liberi, `last` vale `0xFFFFFFFF`. Se tutti i descrittori sono occupati (`buf` diverso da `nullptr`) non è possibile accettare nuove richieste. Un descrittore occupato si libera solo quando `quanti` arriva a zero (tutti i byte sono stati trasferiti) e un processo invoca `cewait()` sul suo identificatore.

Il campo `mutex` è l'indice di un semaforo di mutua esclusione da usare per proteggere gli accessi ai campi `tr`, `cur`, `last` e `waiting`. Il campo `sync` contiene l'indice di un semaforo su cui si sospendono i processi che hanno invocato `cewait()` su un trasferimento non ancora concluso. Il campo `waiting` contiene il numero di processi sospesi su `sync`.

Aggiungiamo infine le seguenti primitive (abortiscono il processo in caso di errore):

- `natl ceasyncread_n(natl id, char *buf, natl quanti)`: accoda, ed eventualmente avvia, un nuovo trasferimento e ne restituisce l'identificatore (`0xFFFFFFFF` se non è possibile accodare). È un errore se la periferica non esiste, se `quanti` è zero, se ci sono problemi di Cavallo di Troia, se il buffer non si trova nella zona utente condivisa o non è scrivibile.
- `void cewait(natl id, natl trid)`: attende la terminazione del trasferimento `trid` sulla periferica `id`. È un errore se la periferica non esiste o se l'identificatore `trid` non corrisponde ad un descrittore occupato.

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