Prova pratica di Calcolatori Elettronici

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

7 febbraio 2018

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

```
struct st1 { int vi[4]; };
struct st2 { char vd[4]; };
class cl {
        long v2[4]; char v1[4]; char v3[4];
public:
        cl(st1 ss);
        cl(st1& s1, int ar2[]);
        cl elab1(char ar1[], st2 s2);
        void stampa() {
                for (int i = 0; i < 4; i++) cout << (int)v1[i] << ' '; cout << endl;
                for (int i = 0; i < 4; i++) cout << (int)v2[i] << ' '; cout << endl;
                for (int i = 0; i < 4; i++) cout << (int)v3[i] << ' '; cout << endl << endl;
        }
};
```

Realizzare in Assembler GCC le funzioni membro seguenti.

```
cl::cl(st1 ss)
{
        for (int i = 0; i < 4; i++) {
                v1[i] = ss.vi[i]; v2[i] = ss.vi[i] * 2;
                v3[i] = 2 * ss.vi[i];
        }
}
cl::cl(st1& s1, int ar2[])
{
        for (int i = 0; i < 4; i++) {
                v1[i] = s1.vi[i]; v2[i] = s1.vi[i] * 4;
                v3[i] = ar2[i];
        }
}
cl cl::elab1(char ar1[], st2 s2)
{
        st1 s1;
        for (int i = 0; i < 4; i++) s1.vi[i] = ar1[i] + i;
        cl cla(s1);
        for (int i = 0; i < 4; i++) cla.v3[i] = s2.vd[i];
        return cla;
}
```

2. Colleghiamo al sistema delle periferiche PCI di tipo `ce`, con vendorID `0xedce` e deviceID `0x1234`. Ogni periferica `ce` usa 32 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia b.

   La periferiche `ce` contengono uno o più contatori indipendenti (fino ad un massimo di 4). Le periferiche decrementano i contatori ogni secondo e inviano una richiesta di interruzione quando uno qualunque di essi raggiunge lo zero. In qualunque momento è possibile scrivere in un contatore, facendo così ripartire il conteggio dal nuovo valore e rimandando la richiesta di interruzione. Scrivendo zero il contatore si ferma, senza inviare interruzioni.

   I registri accessibili al programmatore, tutti di 4 byte, sono i seguenti:

   1. **CFG** (indirizzo b): (sola lettura) specifica il numero di contatori contenuti nel dispositivo; in ogni periferica i contatori sono identificati dai numeri da 0 fino a CFG $-$ 1;
   2. **SWD** (indirizzo $b + 4$): (scrivibile) scrivendo i in questo registro si fa ripartire il contatore i dal valore contenuto nel registro CNT;
   3. **CNT** (indirizzo $b + 8$): (scrivibile) valore iniziale per il contatore selezionato da SWD;
   4. **EWD** (indirizzo $b + 12$): (sola lettura) contiene l'identificatore di un contatore che è arrivato a zero.

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

   Vogliamo usare queste periferiche per controllare che i processi non impieghino troppo tempo a svolgere le proprie operazioni, per esempio perché entrano in un ciclo infinito a causa di un bug. Prima di iniziare un'operazione critica, un processo avvia un contatore con un certo valore e, alla fine, lo ferma. Se l'interruzione arriva prima che il processo riesca a fermare il contatore, il processo viene terminato forzatamente. Ogni contatore può essere utilizzato da un solo processo alla volta. Un processo occupa un contatore da quando lo avvia a quando il contatore arriva a zero (o da solo, o perché il processo lo ferma). È un errore cercare di avviare o fermare un contatore che non esiste, oppure cercare di fermare un contatore che il processo non aveva precedentemente avviato.

   Per realizzare questo meccanismo introduciamo le seguenti primitive di livello I/O (abortiscono il processo in caso di errore):

   - `natl startwatchdog(natl p, natl secs)`: (da realizzare) inizializza uno dei contatori liberi della periferica `p` con il valore `secs` e ne restituisce l'identificatore; restituisce `0xffffffff` se tutti i contatori della periferifa sono occupati.
   - `void stopwatchdog(natl p, natl wd)`: (da realizzare) ferma il contatore `wd` della periferica `p`.

   Modificare i file `io.s` e `io.cpp` in modo da realizzare le due primitive.

   Nota: il modulo sistema mette a disposizione la primitiva `natl getpid()` per ottenere l'identificatore del processo attualmente in esecuzione; la primitiva `void kill(natl id)` può essere usata per terminare forzatamente il processo di identificatore `id`.