Prova pratica di Calcolatori Elettronici

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

1 febbraio 2023

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(const 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] = 4 * 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] * 8;
                v3[i] = ar2[i];
        }
}
cl cl::elab1(const 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. Il modulo I/O del nucleo contiene le primitive `readhd_n()` e `writehd_n()` che permettono di leggere o scrivere blocchi dell'hard disk. Vogliamo velocizzare le due primitive introducendo una *buffer cache* che mantenga in memoria i blocchi letti più di recente, in modo che eventuali letture di blocchi che si trovino nella buffer cache possano essere realizzate con una semplice copia da memoria a memoria, invece che con una costosa operazione di I/O. Per quanto riguarda le scritture adottiamo una politica write-back/write-allocate. Per il rimpiazzamento adottiamo la politica LRU (Least Recently Used): se la cache è piena e deve essere letto un blocco non in cache, si rimpiazza il blocco a cui non si accede da più tempo (nota: per acesso ad un blocco si intende una qualunque `readhd_n()` o `writehd_n()` che lo ha coinvolto).

   Per realizzare la buffer cache definiamo la seguente struttura dati nel modulo I/O:

```
    struct buf_des {
        natl block;
        bool full;
        int next, prev;
        natb buf[DIM_BLOCK];
// altri campi da definire
    };
```

   La struttura rappresenta un singolo elemento della buffer cache. I campi sono significativi solo se `full` è `true`. In quel caso `buf` contiene una copia del blocco `block`. I campi `next` e `prev` servono a realizzare la coda LRU come una lista doppia (si veda più avanti). Aggiungiamo poi i seguenti campi alla struttura `des_ata` (che è il descrittore dell'hard disk):

```
    buf_des bufcache[MAX_BUF_DES];
    int lru, mru;
```

   Il campo `bufcache` è la buffer cache vera e propria; il campo `lru` è l'indice in `bufcache` del prossimo buffer da rimpiazzare (testa della coda LRU) e il campo `mru` è l'indice del buffer acceduto più di recente (ultimo elemento della coda LRU). I campi `next` e `prev` in ogni elemento di `bufcache` sono gli indici del prossimo e del precedente buffer nella coda LRU.

   Aggiungiamo infine la primitiva `void synchd()` (tipo 0x48, senza argomenti), che "sincronizza" l'hard disk con la cache, cioè aggiorna il contenuto di tutti i blocchi che ne hanno bisogno.

   Nota: in nessun caso la cache deve eseguire operazioni di lettura/scrittura dall'hard disk che non siano strettamente necessarie.

   Modificare i file `io.cpp` `io.s` in modo da realizzare il meccanismo descritto.