Prova pratica di Calcolatori Elettronici

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

16 luglio 2025

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

```
struct st1 {
        char vc[3];
};
class cl {
        st1 s;
        long v[3];
public:
        cl(char c, st1& s2);
        void elab1(st1& s1);
        void stampa()
        {
                for (int i = 0; i < 3 ;i++) cout << s.vc[i] << ' '; cout << endl;
                for (int i = 0; i < 3; i++) cout << v[i] << ' '; cout << endl << endl;
        }
};
```

   Realizzare in Assembler GCC le funzioni membro seguenti.

```
void cl::elab1(st1& s1)
{
        cl cla('q', s1);
        for (int i = 0; i < 3; i++) {
                s.vc[i] = cla.s.vc[i];
                v[i] = cla.v[i] - i;
        }
}
```

2. Usiamo un nucleo modificato per supportare lo swap-in e swap-out dei processi utente. Un singolo processo utente può registrarsi come "swapper" e ottenere l'accesso alle primitive `swap_out()` e `swap_in()`. Lo swapper può, in qualunque momento, chiedere lo swap-out un altro processo invocando la primitiva `swap_out()`, a cui passa l'id del processo vittima P e un buffer in cui la primitiva copia il contenuto attuale della parte utente/privata di P. Il processo P diventa "rimosso". In qualunque momento, lo swapper può invocare la primitiva `swap_in()` che provvede a completare o ricaricare un processo. Nel caso di ricarica, lo swapper deve passare alla `swap_in()` un buffer con il contenuto da ripristinare.

   Se un processo sta eseguendo una operazione di I/O in DMA da o verso un buffer nella sua memoria privata, non può essere rimosso fino a quando l'operazione non è stata completata. In questo caso diciamo che il processo è *residente* per tutta la durata dell'operazione. Le primitive di I/O devono

comunicare al modulo sistema quando un processo diventa residente, o smette di esserlo, usando la nuova primitiva `resident()` (accessibile solo dal modulo I/O). Se lo swapper invoca `swap_out()` su un processo residente, lo swapper si sospende e la rimozione viene rimandata a quando il processo non sarà più residente.

Per realizzare il meccanismo aggiungiamo i seguenti campi al descrittore di processo

```
bool resident;
char* swap_out_buf;
```

Il campo `resitent` è true se e solo se il processo è residente. Il campo `swap_out_buf` è diverso da `nullptr` se lo swapper aveva tentato una `swap_out()` su questo processo mentre era residente; in questo caso, contiene il puntatore `buf` che lo swapper aveva passato a `swap_out()`.

Il pid dello swapper si trova nel campo `swapper_info.id`.

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

- `swapper()` (già realizzata): registra il processo invocante come swapper. È un errore se lo swapper esiste già.
- `bool swap_out(natl pid, char *buf)` (realizzata in parte): esegue lo swap-out del processo `pid`. Se il processo `pid` è residente, le operazioni della primitiva vengono rimandate a quando `pid` tornerà non-residente; nel frattempo, il processo swapper resta bloccato.
- `bool swap_in(natl pid, const char *buf)` (già realizzata) esegue lo swap-in o il completamento del processo `pid`.
- `resident(bool res)` (da realizzare): se `res` è `true`, rende il processo invocante residente; se `res` è `false` rende il processo invocante non più residente. Se il processo è già nello stato richiesto, non fa niente. Si preoccupa di completare una eventuale `swap_out()` rimasta in sospeso.

Modificare i file `sistema.cpp` `sistema.s` in modo da realizzare le parti mancanti. **Attenzione**: i file contengono altre strutture dati, funzioni e modifiche che sono utilizzate dallo swapper, ma possono essere ignorate (sono contrassegnate da parentesi quadre invece che tonde nei tag ESAME).