Prova pratica di Calcolatori Elettronici

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

29 gennaio 2024

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

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

   Realizzare in Assembler GCC le funzioni membro seguenti.

```
void cl::elab1(st1 s1, st2 s2)
{
        cl cla(s1.vc, s2);
        for (int i = 0; i < 4; i++) {
                if (s.vc[i] < s1.vc[i])
                        s.vc[i] = cla.s.vc[i];
                if (v[i] <= cla.v[i])
                        v[i] -= cla.v[i];
        }
}
```

2. Aggiungiamo al nucleo il meccanismo delle *barriere con timeout*.

   Una barriera *senza* timeout serve a sincronizzare un certo numero di processi e funziona nel modo seguente: la barriera è normalmente chiusa; un processo che arriva alla barriera si blocca; la barriera si apre solo quando sono arrivati tutti i processi attesi, che a quel punto si sbloccano; una volta aperta e sbloccati tutti i processi, la barriera si richiude e il meccanismo si ripete.

   Il timeout cambia le cose nel seguente modo:

   - il primo processo che arriva alla barriera dopo una chiusura fa partire il timeout;
   - se tutti gli altri processi attesi arrivano prima dello scatto del timeout, la barriera si comporta normalmente (si apre, tutti i processi si sbloccano e poi la barriera si richiude);

- altrimenti la barriera entra in uno stato "erroneo": si apre (quindi, processi già arrivati si risvegliano) e resta aperta fino a quando non sono arrivati tutti i processi attesi, ma tutti i processi la attraversano ricevendo un errore; quando arriva l'ultimo processo, la barriera esce dallo stato erroneo e si richiude.

Nota: sopra e nel seguito, dove diciamo "dall ultima chiusura", intendiamo anche l'istante in cui la barriera è stata creata.

Per rappresentare una barriera introduciamo la seguente struttura dati:

```
struct barrier_t {
    natl nproc;
    natl narrived;
    natl timeout;
    bool bad;
    des_proc *waiting;
    des_proc *first;
};
```

Dove: `nproc` è il numero di processi che devono sincronizzarsi sulla barriera; `narrived` conta i processi arrivati alla barriera dall'ultima chiusura; `timeout` è il timeout che regola l'entrata nello stato erroneo; `bad` è true se e solo se la barriera si trova nello stato erroneo; `waiting` è la coda dei processi che attendono l'apertura della barriera; `first` punta al descrittore del primo processo arrivato dall'ultima chiusura (quello che ha avviato il timeout corrente).

Aggiungiamo anche il seguente campo ai descrittori di processo:

```
natl barrier_id;
```

Se questo campo è diverso da 0xFFFFFFFF, vuol dire che questo processo è bloccato sulla barriera con identificatore `barrier_id`, ed è il primo processo ad essere arrivato su quella barriera dall'ultima chiusura.

Aggiungiamo inoltre le seguenti primitive:

- `natl barrier_create(natl nproc, natl timeout)` (già realizzata): crea una nuova barriera che sincronizza `nproc` processi con timeout `timeout` e ne restituisce l'identificatore (`0xFFFFFFFF` se non è stato possibile completare l'operazione).
- `bool barrier(natl id)` (da realizzare): fa giungere il processo corrente alla barriera di identificatore `id`. È un errore se tale barriera non esiste. Restituisce `true` quando termina normalmente, e `false` quando termina perchè la barriera è stata attraversata nello stato erroneo.

Le primitive abortiscono il processo chiamante in caso di errore e tengono conto della priorità tra i processi.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti.