[semafori] Imbuto 2.0

Si devono sincronizzare un certo numero di thread `pallina` in modo che entrino in un imbuto solo a gruppi di una dimensione prefissata `N` e, dopo un certo tempo, escono dall'imbuto. Solo nel momento in cui sono usciti tutti, il gruppo successivo di `N` thread `pallina` può accedere.

ATTENZIONE: In questa versione dell'esercizio non è il thread principale a sbloccare i gruppi di thread pallina, ma sono le palline stesse che si sincronizzano autonomamente.

Il thread principale si limita a inizializzare i semafori, creare i thread `pallina` attendere la loro terminazione e eliminare i semafori, secondo il seguente schema. La dimensione dei gruppi `N` viene passata a `inizializza_sem` in modo da poter essere utilizzata, successivamente, per la sincronizzazione.

```
inizializza_sem(N); // inizializza i semafori e salva N

// Crea i thread pallina e attende la loro terminazione

distruggi_sem(); // distruggi i semafori
```

Lo schema del thread "pallina" è il seguente:

```
entra_imbuto(); // attende di entrare nell'imbuto

// entra nell'imbuto e percorri tutta la strada verso il fondo

esci_imbuto(); // esce dall'imbuto
```

L'obiettivo della verifica è di implementare le 2 funzioni di sincronizzazione `entra_imbuto` e `esci_imbuto` tramite semafori (più le 2 funzioni `inizializza_sem` e `distruggi_sem` per inizializzare e eliminare i semafori) in modo da realizzare il comportamento richiesto.

IMPORTANTE: la funzione `esci_imbuto` deve necessariamente contare il numero di palline che sono uscite dall'imbuto per poi sbloccare il gruppo successivo: questa sincronizzazione non è realizzabile utilizzando solamente `wait` e `post` sui semafori. Utilizzare, a tale scopo, una **variabile globale** proteggendo opportunamente la sezione critica. Non ci sono problemi a fare delle `post` su altri semafori dentro una sezione critica perché le `post` non sono mai bloccanti.

Le funzioni da implementare sono nel file `soluzione.c` che viene incluso da `imbuto2.c` (scarica lo zip). Compilare con `gcc imbuto2.c -o imbuto2 -pthread -fno-common` (l'opzione `-fno-common` non è necessaria ma evita che ci sia conflitto tra variabili globali dichiarate nei due `.c`).

Prima di consegnare compilare con l'opzione `-DSTRESSTEST` che testa il programma con un numero elevato di thread e senza sleep (in modo da aumentare la probabilità di race condition). Per ripetere il test in automatico potete usare questo semplice script bash: `while true; do ./imbuto2; done`

Nascondi soluzione

```
/*
 * COMMENTO GENERALE:
 *
 * PREMESSA: La dimensione N del gruppo di palline viene salvata
 * durante l'invocazione di inizializza_sem. Nel seguito usiamo
 * N per indicare tale dimensione.
 *
 * Utilizziamo una variabile globale n_uscite per contare quante
 * palline sono uscite dall'imbuto in modo da poter sbloccare
 * il gruppo successivo di N palline.
 *
 * - n_uscite: quante palline sono uscite dall'imbuto. Viene
 *   impostata a 0 e incrementata ogni volta che una pallina
 *   esce. La pallina che esce per ultima sblocca successive
 *   N palline. La variabile deve essere protetta da un
 *   mutex (sezione critica)
 *
 * Utilizziamo un semaforo sem_imbuto per regolare l'accesso
 * all'imbuto e un semaforo mutex per poter gestire il conteggio
 * delle palline uscite dall'imbuto:
 *
 * - sem_imbuto: corrisponde alla risorsa imbuto, che le palline
 *   richiedono per poter accedervi. Inizialmente vale
 *   N in quanto N palline possono accedere;
 *
 * - mutex: serve a proteggere l'aggiornamento della variabile
 *   globale n_uscite che conta quante palline sono uscite
 *   dall'imbuto. Inizialmente vale 0.
 *
 * SINCRONIZZAZIONE:
 *
 * Thread pallina: per accedere all'imbuto la pallina effettua
 * una wait sul semaforo sem_imbuto (funzione entra_imbuto) che
 * ne regola, appunto, l'accesso. Quando esce dall'imbuto
 * incrementa la variabile n_uscite. Se n_uscite e' uguale a
 * N il thread effettua N post sul semaforo sem_imbuto in modo da
 * sbloccare il successivo gruppo di N palline. Il contatore
 * n_uscite viene resettato a 0 in modo da ricominciare il
 * conteggio. Tutta la parte di codice che gestisce l'uscita
 * tramite il contatore globale n_uscite viene protetta
 * da mutex, in modo da evitare interferenze (funzione esci_imbuto).
 *
 * NOTA: poiche' non vengono fatte wait dentro la sezione critica
 * non ci sono problemi di eventuali stalli.
```

```
 *
 * Riconduzione a un problema standard: la soluzione è, di
 * fatto, una variante del produttore-consumatore in cui
 * i thread pallina consumano una risorsa imbuto, prodotta
 * a "lotti" di N. L'ultimo thread pallina che esce
 * dall'imbuto produce un altro "lotto" di risorse imbuto
 * facendo N post sul semaforo sem_imbuto.
 *
 * La sincronizzazione è quindi uno-a-molti: l'ultima pallina
 * che esce sblocca N successivi thread (effettuando N post,
 * che concedono N nuove risorse).
 */

// dichiarazione semafori e variabili globali
sem_t   sem_imbuto, // regola l'accesso all'imbuto
        mutex;      // protegge l'aggiornamento di n_uscite

int n_uscite=0;  // numero di palline uscite dall'imbuto
int N=0; // memorizza la dimensione del gruppo

// inizializza i semafori
void inizializza_sem(int dim) {
    /*
     * memorizza in N la dimensione del gruppo di
     * palline che puo' entrare nell'imbuto.
     * Il semaforo sem_imbuto viene inizializzato a N
     * perche' inizialmente possono entrare N palline.
     * Il semaforo mutex viene inizializzato a 1
     */
    N=dim; // memorizza la dimensione del gruppo

    sem_init(&sem_imbuto,0,N); // N accessi all'imbuto
    sem_init(&mutex,0,1); // mutex per proteggere n_uscite
}

// distruggi i semafori
void distruggi_sem() {
    sem_destroy(&sem_imbuto); // elimina il semaforo sem_imbuto
    sem_destroy(&mutex);  // elimina il semaforo pallina
}

// attende di entrare nell'imbuto
void entra_imbuto() {
    /*
     * per accedere all'imbuto la pallina effettua una wait sul
     * semaforo sem_imbuto che ne regola, appunto, l'accesso:
     * di fatto, consuma una risorsa imbuto.
     */
    sem_wait(&sem_imbuto);
}

// esce dall'imbuto
void esci_imbuto() {
    /*
     * Tutto il codice è in sezione critica tramite mutex.
     * si incrementa n_uscite e quando raggiunge N vengono
     * effettuate N sem_post su sem_imbuto in modo da sbloccare
     * il successivo gruppo di N thread pallina. Il contatore
     * viene messo a 0 per ricominciare il conteggio.
     */
    int i;

    sem_wait(&mutex);   // entra in sezione critica
    n_uscite++;         // la pallina è uscita
    // se sono uscite N palline ne fa accedere altre N
    if (n_uscite == N) {
```

```
111             // fa accedere altre N palline
112             for (i=0;i<N;i++)
113                 sem_post(&sem_imbuto); // sblocca una pallina
114             n_uscite = 0;   // resetta il contatore
115         }
116         sem_post(&mutex); // esce dalla sezione critica
117     }
```