Semafori POSIX

I semafori POSIX sono semafori contatori che permettono di gestire la sincronizzazione dei thread POSIX. Esistono altri meccanismi che, per mancanza di tempo, menzionano solamente: I Pthread Mutex sono semafori binari mentre le Pthread Condition vengono utilizzate per 'simulare' il costrutto dei *monitor* che vedremo invece nel linguaggio Java.

I semafori POSIX si utilizzano tramite le seguenti strutture dati e funzioni:

- `sem_t sem_name`: dichiara una variabile di tipo semaforo;
- `int sem_init(sem_t *sem, int pshared, unsigned int value)` inizializza il semaforo `sem` al valore `value`. la variabile `pshared` indica se il semaforo è confiviso tra thread (uguale a 0) o processi (diverso da 0), lo useremo quindi sempre con 0.
- `int sem_wait(sem_t *sem)` esegue una P(sem);
- `int sem_post(sem_t *sem)` esegue una V(sem);
- `int sem_getvalue(sem_t *sem, int *val)` Legge il valore del semaforo e lo copia in `val`;
  ATTENZIONE: in alcune implementazioni il semaforo rosso è 0, in altre è negativo (e indica il numero di processi in attesa);
- `sem_destroy(sem_t *sem)` elimina il semaforo. Da NON usare se ci sono processi in attesa sul semaforo (comportamento non specificato).

NOTA PER GLI UTENTI MAC: OS X non supporta i semafori 'unnamed' descritti qui sopra. Per poter utilizzare i semafori POSIX si devono usare i semafori con nome (che tra l'altro sono facilmente condivisibili tra processi, un po' come le pipe con nome). Al posto della `sem_init` si deve usare la `sem_open` e al posto della `sem_destroy` si usano `sem_close` e `sem_unlink`. Ovviamente i semafori con nome si possono utilizzare anche in Linux.

ESERCIZIO 1: Riprendiamo l'ultimo esercizio della volta scorsa:

> Creare 2 thread che aggiornano ripetutamente (in un ciclo for) una variabile condivisa count per un numero elevato di volte (ad esempio 1000000). Stampare il valore finale per osservare eventuali incrementi perduti…
>
> (Seguiva una nota sulle ottimizzazioni del compilatore. Vedere le dispense della volta scorsa per maggiori dettagli)

Aggiungere un semaforo mutex per risolvere le interferenze. È sufficiente aggiungere una variabile globale `sem_t sem` e, all'inizio e alla fine del main, l'inizializzazione e la rimozione del semaforo: `sem_init(&sem,0,1)` e `sem_destroy(&sem)`. Infine, per proteggere la sezione critica, aggiungere `sem_wait(&sem)` e `sem_post(&sem)` prima e dopo la lettura/modifica di count. Notare l'esecuzione corretta al prezzo di una più bassa performance.

```
for (j=0;j<MAX;j++) {
                sem_wait(&sem);

                count++

                sem_post(&sem);
}
```

NOTA: per OS X si useranno `sem=sem_open("mymutex",O_CREAT,0700,1)` e `sem_close(sem); sem_unlink("mymutex");` facendo attenzione che in questo caso `sem` sarà un puntatore a `sem_t` (quindi anche la wait e la post andranno fatte su `sem` e non su `&sem`).

ESERCIZIO 2: implementare il Produttore-Consumatore con buffer circolare visto a lezione, utilizzando due semafori contatori più un mutex. Testarlo in presenza di più produttori e più consumatori, verificando che la presenza del mutex è fondamentale per la coerenza dei dati.

Inventarsi a tale scopo un test di qualche genere. Ad esempio si può tenere nelle celle vuote un valore speciale (-1) e testare se si sta leggendo una cella vuota o scrivendo in una piena. Ovviamente il consumatore deve scrivere nella cella il valore speciale, dopo che ha letto. Il test dovrebbe segnalare problemi di sincronizzazione nel caso non si utilizzino appropriatamente i semafori.

NOTA: l'output a video talvolta riduce le interferenze in quanto crea una alternanza molto forte nell'esecuzione dei thread. In tale caso, provare a fare la ridirezione (tramite il simbolo >) su un file.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <semaphore.h>

/*variabili globali */
int MAX=100;
int inserisci = 0;   //Variabile utile a produttore per sapere dove inserire
int preleva = 0;     //Variabile utile a consumatore per sapere dove leggere

/*Semafori*/
sem_t PIENE;
sem_t VUOTE;
sem_t MUTEX;

/* Gestisco gli errori */
die(char * s, int e) {
    printf("%s [%i]\n",s,e);
    exit(1);
}

/* Produttore */
void * codice_thread_P(void *b) {
    int * buff=(int*)b;
    int a;
    while(1) {
    a=rand() % 100;                         //produco il dato
    printf("dato prodotto: %d\n",a);        //lo stampo a video
    sem_wait(&VUOTE);                       //attendo che vi siano celle vuote
    sem_wait(&MUTEX);                       //attendo il mio turno

    if(buff[inserisci]!=-1){                //controllo se sto scrivendo su una cella vuota
        printf("cella già piena!!\n");
        die("errore in scrittura",-1);
    }
    buff[inserisci] = a;                    //scrivo nel buffer
    inserisci = (inserisci + 1) % MAX;      //incremento inserisci
    sem_post(&MUTEX);                       //cedo il turno ad un altro tharead
    sem_post(&PIENE);                       //comunico che vi è una cella da leggere

    }

    pthread_exit(NULL);                    //chiudo il thread
}



/* Consumatore*/
void *codice_thread_C(void *b) {
    int * buff=(int*)b;
 int a;
while (1) {
    sem_wait(&PIENE);                    //attendo che vi siano celle piene
    sem_wait(&MUTEX);                   //attendo il mio turno
    if(buff[preleva]==-1){              //controllo di non leggere celle vuote
        printf("cella vuota!!");
        die("cella vuota",-1);}
    a = buff[preleva];                  //leggo il dato
    buff[preleva]=-1;                   //setto la cella come vuota
    preleva = (preleva + 1) % MAX;      //incremento preleva
    sem_post(&MUTEX);                   //faccio passare al semaforo il prossimo thread
     sem_post(&VUOTE);                  //comunico che vi è una cella libera

    printf("Valore raddoppiato: %d \n" , 2*a);   //consumo il dato


}

    pthread_exit(NULL);                 //chiudo il thread
}

main() {
/*inizializzazione semafori*/
   sem_init(&PIENE, 0 , 0);
   sem_init(&VUOTE, 0, MAX);
   sem_init(&MUTEX, 0, 1);

    int i,err;

    int buff[MAX];
    for(i=0;i<MAX;i++)
```

```
    for(i=0;i<MAX;i++)
        buff[i]=-1;

    pthread_t tid[20];

    /*creo 20 thread, 10 produttori e 10 consumatori*/
    for(i=0;i<10;i++){
        if (err=pthread_create(&tid[i],NULL,codice_thread_P,buff))
            die("errore creazione produttore",err);
        }
    for(i=11;i<20;i++){
    if (err=pthread_create(&tid[i],NULL,codice_thread_C,buff))
            die("errore creazione consumatore",err);}

    /* attende i thread. */
   for (i=0;i<2;i++)
     if (err=pthread_join(tid[i], NULL))  die("errore Join",err);

    sem_destroy(&PIENE);
    sem_destroy(&VUOTE);
    sem_destroy(&MUTEX);

    printf("I thread hanno terminato l'esecuzione correttamente\n");
}
```