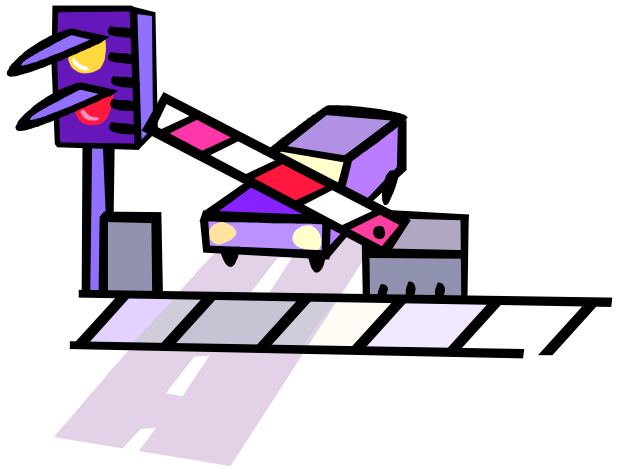
Semafori POSIX



- Sono una particolare implementazione dei semafori.
- Possono svolgere il ruolo di semafori binari o generali (n-ari) a seconda di quale valore viene loro assegnato all'atto della inizializzazione
- Simili a mutex+cond ma non perdono le signal (qui le signal si chiamano **sem_post**) perche' le memorizzano in un contatore.
- Però ciascun semaforo ha una propria mutex per garantire la mutua esclusione delle proprie variabili interne, mentre con le mutex+cond posso proteggere più cond con una stessa mutex.

Tipi di POSIX Semaphores

- Named Semaphores
  - Permettono di sincronizzare processi anche non imparentati, oltre ai thread.
  - Si inizializzano con **`sem_open`**
  - Identificati da un percorso
  - Persistenti e Visibili in tutto il sistema.

- Unnamed Semaphores 
  - Permettono di sincronizzare processi imparentati e thread
  - Condivisi tra thread o tra processi
  - Si inizializzano con **`sem_init`**

POSIX Semaphores

- I Semafori sono una variable di tipo **`sem_t`**
- Include **`<semaphore.h>`**
- **`Linkare con -lpthread`**
- Operazioni (unnamed semaphores)

```
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_destroy(sem_t *sem);
```

- **Atomiche** : **`OPERANO IN MUTUA ESCLUSIONE, in particolare nell'accesso al contatore che contengono:`**

```
int sem_wait(sem_t *sem);        /* wait   V   verhogen */
int sem_post(sem_t *sem);        /* signal P   proberen */
int sem_trywait(sem_t *sem);
```

Unnamed Semaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned value);
```

- inizializza un unnamed semaphore
- Restituisce
  - 0 in caso tutto vada bene
  - -1 in caso di fallimento, nel qual caso setta **`errno`**
- Parametri
  - **`sem`**:
    - Target semaphore
  - **`pshared`**:
    - 0: solo i threads del processo che lo crea puo' usare il semaforo
    - Non-0: altri processi possono usare il semaforo
  - **`value`**:
    - Initial value of the semaphore
- **NOTA BENE**
  - **Non e' corretto creare una copia di un semaforo, diventano due semafori diversi e non permettono piu' la sincronizzazione. Per piu' processi, creare semafori in memoria condivisa.**

Condivisione di Semafori

- Per condividere un semaforo tra piu' processi, occorre crearlo in un;area di memoria condivisa e inizializzarlo con il parametro **`pshared != 0`** `(1 va bene)`
  - Una fork() crea una copia del semaforo e non consente piu' la sincronizzazione tra processi.

- Per condividere un semaforo tra soli thread, occorre solo inizializzarlo con il parametro **`pshared = 0`**

Inizializzazione **sem_init** (puo' fallire)

```
#include <semaphore.h>
      int sem_init(sem_t *sem, int pshared,
                   unsigned int initial_value );
```

- inizializza il semaforo e assegna il valore iniziale
- in caso di insuccesso
  - **sem_init** returns -1 and sets **errno**

errno	cause
EINVAL	**Value > sem_value_max**
ENOSPC	Resources exhausted
EPERM	Insufficient privileges

pshared ==0 significa gestire solo pthread, non processi

initial_value deve essere >= 0

```
sem_t semA;

if (sem_init(&semA, 0, 1) == -1)
  perror("Failed to initialize semaphore semA");
```

Distruzione Semaforo

```
#include <semaphore.h>
        int sem_destroy(sem_t *sem);
```

- Elimina il semaforo e le risorse da questo allocate

- Restituisce
  - 0 in caso di successo
  - -1 in caso di fallimento, e setta **`errno`**
- Parametri
  - **`sem`**:
    - Target semaphore

- Nota bene
  - Puoi distruggere un semaforo solo una volta.
  - Distruggere un semaforo su cui un thread e' bloccato causa risultati impredicibili.

Sblocca Semaforo

```
#include <semaphore.h>
        int sem_post(sem_t *sem);
```

- Sblocca il semaforo, cioe'
  - Incrementa di 1 il valore del semaforo
  - Se il valore diventa > 0 qualche altro thread bloccato sulla sem_wait puo' continuare l'esecuzione

- Restituisce
  - 0 in caso di successo
  - -1 in caso di errore e setta **`errno`**
  - (unico errore possibile **`EINVAL`** se il semaforo non esiste)
- Parametri
  - **`sem`**:
    - Target semaphore
    - sem > 0: no threads were blocked on this semaphore, the semaphore value is incremented
    - sem == 0: one blocked thread will be allowed to run
- Nota
  - **`sem_post()`** e' rientrante e puo' essere invocato in un gestore di segnali dei processi

Blocca su Semaforo

```
#include <semaphore.h>
        int sem_wait(sem_t *sem);
```

- Blocca il thread sul semaforo
  - controlla il valore del semaforo
  - se il valore e' minore o uguale a zero
    - si blocca e riparte uscendo dalla wait quando diventa uguale a zero
    - poi decrementa il valore del semaforo
  - se il valore e' maggiore di zero
    - prosegue uscendo subito dalla wait
    - poi decrementa il valore del semaforo.
- Restituisce
  - 0 in caso di successo
  - -1 in caso di errore, setta **`errno`** (**`== EINTR`** se interrotto da un segnale)
- Parametri
  - **`sem`**: Target semaphore
    - sem > 1: decrementa e basta
    - sem == 1: decrementa e blocca altri thread (thread acquires lock)
    - sem <= 0: si blocca e quando viene sbloccato decrementa (thread blocks)

Controlla Condizione Semaforo

```
#include <semaphore.h>
        int sem_trywait(sem_t *sem);
```

- Valuta la condizione del semaforo.
- Non Blocca il thread
- Restituisce
  - 0 in caso di successo
  - -1 in caso di insuccesso, e setta errno
    - se **errno == EAGAIN il semaforo e' gia' bloccato**

- Parametri
  - **sem**: Target semaphore
    - sem > 1: decrementa e basta
    - sem == 1: decrementa e blocca altri thread (thread acquires lock)
    - sem <= 0: NON si blocca e non decrementa, ma restituisce -1 con errno == EAGAIN

Valore Iniziale Assegnato ai Semafori

- Il valore iniziale assegnato al semaforo coincide con il numero di risorse che, inizialmente, possono essere contemporaneamente accedute da piu' thread.

- Semafori binari

  - Valore iniziale **= 1** significa che se il primo thread chiama la sem_wait blocca tutti gli altri e prosegue. E' come se la wait fosse la lock su un mutex libero.
  - Valore iniziale **= 0** significa partire con una risorsa gia' occupata. Se il primo thread chiama la sem_wait si blocca fino a che un qualche altro thread non fa una sem_post.

- Semafori n-ari (Generali)

  - Valore iniziale = **N>0** significa che i primi N thread possono chiamare la sem_wait e proseguire senza bloccarsi. Lo N+1 esimo thread che chiama la sem_wait si blocca fino a che un thread non fa la sem_post.

Esempio Semafori Binari

```
sem_t  do_a, do_b;
```

Main

```
sem_init( &do_a, /*solo thread*/0, /*val.iniziale*/ 1 );
sem_init( &do_b, /*solo thread*/0, /*val.iniziale*/ 0 );
```

Thread **A**

```
while(1) {
    sem_wait(&do_a);
    a; /*sezione critica*/
    sem_post(&do_b);
}
```

Thread **B**

```
while(1) {
    sem_wait(&do_b);
    b; /*sezione critica*/
    sem_post(&do_a);
}
```

- L'inizializzazione di **do_a** al valore **1** consente al Thread A di passare la prima sem_wait , risvegliare il thread B con la sem_post su do_b, e poi di mettersi in attesa di essere a sua volta risvegliato dal thread B.
- L'inizializzazione di **do_b** al valore **0** blocca il Thread B nella prima sem_wait e lo fa attendere fino a che A non fa la sem_post.

Esempio Semaforo Generale

```
/*   sem_example.c   */
#define NUMADMITTED  6
#define  NUMTHREADS 8
sem_t   sem;
```

Main

```
sem_init( &sem , 0, /*val.iniziale*/ NUMADMITTED );
for ( i=0; i< NUMTHREADS; i++ )
   pthread_create( ......, ThreadFunction, ... );
```

ThreadFunction (void *arg) {

```
sem_wait(&sem);
printf("critical region"
        " a 6 posti");
sleep(1);
sem_post(&sem);
```

}

- I primi NUMADMITTED pthread superano la sem_wait decrementando fino a zero il semaforo e poi aspettano 1 secondo.
- Gli altri 2 pthread si bloccano sulla sem_wait fino a che due dei primi pthread terminano la sleep e eseguono la sem_post.
- Anche gli ultimi due pthread superano la sem_wait

Esercizio: implementare semafori con mutex e cond (1)

Esercizio: implementare i Semafori POSIX mediante Mutexes e Condition Variable

Scaricare e scompattare l'archivio contenuto in

http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/PTHREAD/POSIX_SEMAPHORES/BASE_MYSEM.tgz

Implementare le funzioni i cui prototipi sono in mySem.h

mySem_init, mySem_destroy, mySem_wait, mySem_post.

Al posto della struttura sem_t utilizzare la struttura **mySem_t** dichiarata in mySem.h

Semplificazione 1:

in caso di successo, le finzioni restituiscono 0.

in caso di errore le funzioni terminano il processo con exit(1);

Semplificazione 2:

implementare solo il caso sem_init con pshared==0 cioe' solo semafori per soli thread.

L'archivio BASE_MYSEM.tgz contiene anche un main.c in cui vengono usate le funzioni da implementare, ed un makefile per compilare e linkare tutto.

```
Implementate le vostre funzioni mySem* in un modulo  mySem.c

segue:
```

Esercizio: implementare semafori con mutex e cond (2)

Il manuale dice esattamente cosa implementare (dove dice *sem_t* sostituire con *mySem_t*):

DESCRIPTION

sem_wait() decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

sem_post() increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait(3) call will be woken up and proceed to lock the semaphore.

segue:

Qui trovate una soluzione, ma non guardatela subito.

http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/PTHREAD/POSIX_SEMAPHORES/MYSEM/mySem.c