

Esercitazione [02]

Concorrenza

Daniele Cono D'Elia – delia@diag.uniroma1.it

Riccardo Lazzeretti – lazzeretti@diag.uniroma1.it

Luca Massarelli – massarelli@diag.uniroma1.it

Sistemi di Calcolo - Secondo modulo (SC2)

Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2017-2018

Sommario

- Recap esercizio su accesso concorrente a variabili condivise (senza semafori)
 - Problematiche sul passaggio dei parametri
 - Esempi passaggio parametri a funzioni
- Concorrenza: breve riepilogo e semafori in C
- Accesso sezione critica in mutua esclusione
 - Misurazione overhead semafori
- Accesso in mutua esclusione a N risorse

Recap: soluzione esercizio accesso concorrente a variabili condivise (1/3)

- Esercizio: implementare una soluzione al seguente problema senza meccanismi di sincronizzazione:
 - N thread effettuano in parallelo M incrementi di valore V
 - Al termine, il main thread verifica che tali incrementi equivalgano complessivamente a $N * M * V$
 - Suggerimento: lavorare sulle strutture dati per evitare accessi concorrenti in scrittura
- Soluzione
 - ogni thread incrementa una locazione di memoria diversa
 - alla fine il main thread somma tutti i valori
 - sorgente: `sol_concurrent_threads.c`

Recap: soluzione esercizio accesso concorrente a variabili condivise (2/3)

- Compilazione

```
gcc -o sol_concurrent_threads  
sol_concurrent_threads.c -lpthread
```

- Esecuzione

```
./sol_concurrent_threads <N> <M> <V>
```

- Non dovrebbero risultare add perse

- Cosa succede se invece di usare `&thread_ids[i]` usiamo `&i` ?

Recap: soluzione esercizio accesso concorrente a variabili condivise (3/3)

- Non si può avere alcuna garanzia riguardo a quando verrà eseguita l'istruzione

```
int thread_idx = *((int*)arg);
```

- Nel mentre, può succedere che il valore nella locazione di memoria puntata da `arg` venga cambiato
 - È il valore del contatore `i`
 - Più thread con la stessa «identità» (`thread_idx`)
 - Si ripropone il problema dell'accesso concorrente

Soluzione alternativa

- Ogni thread lavora su variabili locali e restituisce il valore tramite `pthread_exit`
- Il main raccoglie i valori tramite `pthread_join` e li somma
- **Compilazione**
`gcc -o sol2_concurrent_threads sol2_concurrent_threads.c -lpthread`
- **Esecuzione**
`./sol2_concurrent_threads <N> <M> <V>`

Concorrenza - Semafori

1. Inizializzazione

Assegna un valore iniziale non negativo al semaforo

2. Operazione semWait

Decrementa il valore del semaforo, se il valore è negativo il processo/thread viene messo in attesa in una coda, altrimenti va avanti

3. Operazione semSignal

Incrementa il valore del semaforo, se il valore non è positivo un processo/thread viene risvegliato dalla coda

Concorrenza in C - Semaphores

```
#include <semaphore.h>
```

```
...
```

```
sem_t sem;
```

```
...
```

```
sem_init(&sem, pshared, value)
```

```
...
```

```
sem_wait(&sem)
```

```
...
```

```
sem_post(&sem)
```

```
...
```

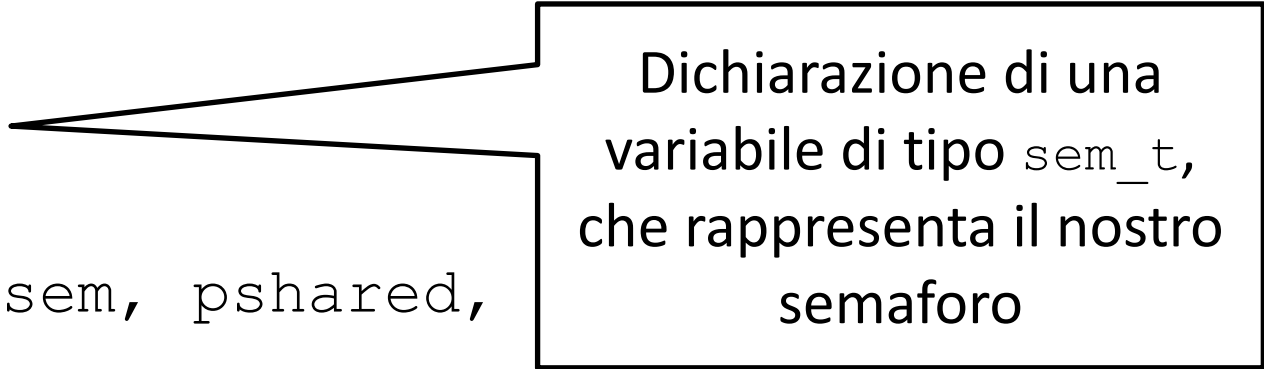
```
sem_destroy(&sem)
```



Header da includere

Concorrenza in C - Semaphores

```
#include <semaphore.h>
...
sem_t sem;
...
sem_init(&sem, pshared,
...
sem_wait(&sem)
...
sem_post(&sem)
...
sem_destroy(&sem)
```



Dichiarazione di una
variabile di tipo `sem_t`,
che rappresenta il nostro
semaforo

Concorrenza in C - Semaphores

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

...

S

Inizializzazione del semaforo con valore `value`.

...

S

Se `pshared` vale 0, il semaforo viene condiviso tra i thread del processo; altrimenti, il semaforo viene condiviso tra processi,

...

S

a patto che sia in una porzione di memoria condivisa

(quest'ultimo caso non verrà esaminato nel corso).

In caso di successo, viene ritornato 0; in caso di errore, -1.

Concorrenza in C - Semaphores

```
#include <semaphore.h>
...
sem_t sem;
...
sem_init(&sem, pshared, value)
...
sem_wait(&sem)
...
sem_post(&sem)
```

Operazione semWait sul semaforo sem.
In caso di successo, viene ritornato 0; in
caso di errore, -1.

```
sem_post(&sem)
```

Concorrenza in C - Semaphores

```
#include <semaphore.h>
...
sem_t sem;
...
sem_init(&sem, pshared, value)
...
sem_wait(&sem)
...
sem_post(&sem)
...
sem_
```

Operazione semSignal sul semaforo sem.
In caso di successo, viene ritornato 0; in
caso di errore, -1.

Concorrenza in C - Semaphores

```
#include <semaphore.h>
```

```
...
```

```
sem_t sem;
```

```
...
```

```
sem_init(&sem, pshared, value)
```

```
...
```

```
sem_wait(&sem)
```

```
...
```

```
sem
```

```
...
```

```
sem_destroy(&sem)
```

Distrugge il semaforo sem.

In caso di successo, viene ritornato 0; in caso di errore, -1.

Obiettivi Esercitazione

- Imparare ad usare i semafori in C
 - a. Come si implementa la mutua esclusione per l'accesso ad una sezione critica?
 - b. Quanto vale l'overhead dei semafori?
 - c. Come si implementa l'accesso in mutua esclusione a N risorse distinte?

Accesso sezione critica in mutua esclusione

- Riprendiamo `concurrent_threads` e risolviamo il problema delle race condition
 - Sezione critica: `shared_variable += v;`
 - Va protetta con un semaforo
 - Acquisizione lock sulla sezione critica tramite `sem_wait`
 - Esecuzione sezione critica
 - Rilascio lock sulla sezione critica tramite `sem_post`
 - Sorgente: `concurrent_threads.c`

concurrent_threads_semaphore.c

- Garantire mutua esclusione utilizzando i semafori
 - Creare una copia del file e chiamarla `concurrent_threads_semaphore.c`
 - Introdurre opportunamente i semafori
 - Compilazione:

```
gcc -o concurrent_threads_semaphore  
concurrent_threads_semaphore.c performance.c  
-lpthread
```


concurrent_threads_semaphore.c

- Opzionale: misurazione delle prestazioni
 - Viene usata la libreria `performance` per misurare il tempo di esecuzione
 - effettuare un confronto sui tempi di esecuzione tra questa soluzione e quelle senza semafori
- Compilazione:

```
gcc -o concurrent_threads_semaphore  
concurrent_threads_semaphore.c  
performance.c -lpthread -lrt -lm
```

Produttore Consumatore

- Prendiamo il codice `producer_consumer.c` in cui non viene garantita la mutua esclusione
- Ruolo delle variabili globali:
 - R: numero di risorse (lunghezza dell'array)
 - N: numero di produttori
 - M: numero di consumatori
 - O: numero di operazioni per produttori e consumatori
- Compilazione:
`gcc -o producer_consumer
producer_consumer.c -lpthread`

Produttore Consumatore

- Risolvere il problema nei seguenti quattro scenari, come illustrato nella lezione di lunedì 19 marzo
 - $1 : 1$ (1 produttore, 1 consumatore)
 - $1 : M$ (1 produttore, M consumatori)
 - $N : 1$ (N produttori, 1 consumatore)
 - $N : M$ (N produttori, M consumatori)
- Si consiglia di creare 4 soluzioni distinte