Esercitazione [02]

Concorrenza

Leonardo Aniello – aniello@dis.uniroma1.it
Daniele Cono D'Elia – delia@dis.uniroma1.it
Giuseppe Laurenza – laurenza@dis.uniroma1.it
Federico Lombardi – lombardi@dis.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. 2016-2017

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: `concurrent_threads.c`

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

- Compilazione
  `gcc -o concurrent_threads concurrent_threads.c -lpthread`
- Esecuzione
  `./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

Esempio funzioni con e senza passaggio di puntatori

- Obiettivo: capire la differenza tra funzioni che prendono come argomento puntatori e variabili
- Il programma lancia due funzioni che incrementano un intero in 2 modi diversi:
  - `foo1` prende un `int*`
  - `foo2` prende un `int`

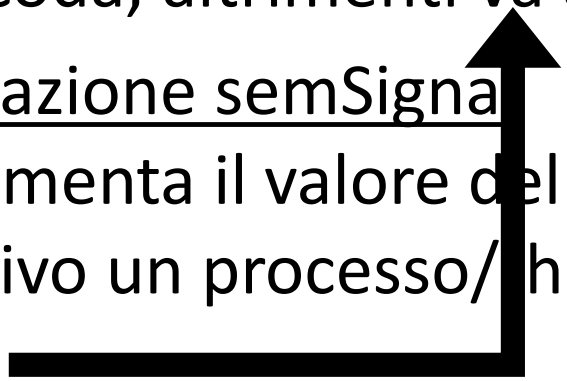
- Sorgente: `es_foo.c`
- Compilazione
  `gcc -o es_foo es_foo.c`
- Esecuzione
  `./es-foo <N>`

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

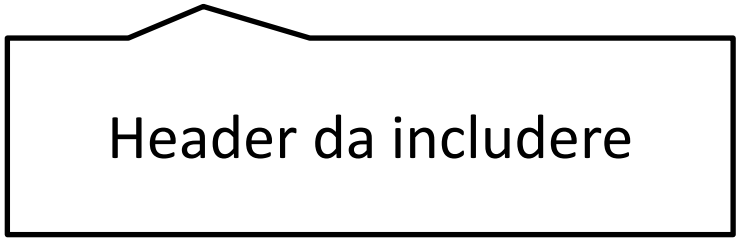
Concorrenza - Semafori

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)
```



Concorrenza in C - Semaphores

```
#include <semaphore.h>

…

sem_t sem;

…

red, value)

…

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
…
s
…
s
```

Inizializzazione del semaforo con valore `value`.
Se `pshared` vale `0`, il semaforo viene condiviso tra i thread del processo; altrimenti, il semaforo viene condiviso tra processi, 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)
…
ser
…
sem_destroy(&sem)
```

Operazione semWait 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_post(&sem)
…
ser
```

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 [2]

- 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 semWait
    - Esecuzione sezione critica
    - Rilascio lock sulla sezione critica tramite semSignal
  - Sorgente: `concurrent_threads_semaphore.c`

concurrent_threads_semaphore.c

- Compilazione
  ```
  gcc -o concurrent_threads_semaphore
  concurrent_threads_semaphore.c
  performance.c -lpthread -lrt -lm
  ```
- Viene usata la libreria `performance` per misurare il tempo di esecuzione
- Esercizio: effettuare un confronto sui tempi di esecuzione tra questa soluzione e quella senza semafori

Accesso in mutua esclusione a N risorse

- Disponibilità di un numero N di risorse, ognuna delle quali può essere usata in mutua esclusione
  - Pool di connessioni a DB
  - Pool di thread
  - etc…
- M thread in concorrenza devono accedere a queste risorse (M > N)
- Come implementarlo con i semafori?
  - Suggerimento: mentre prima solo un thread alla volta poteva accedere alla sezione critica, ora vogliamo che ciò sia possibile per N thread alla volta

Accesso in mutua esclusione a N risorse - Implementazione

- Soluzione: inizializzare il semaforo a `N` invece che a `1`
- Codice: `scheduler.c`
- Compilazione
  `gcc -o scheduler scheduler.c -lpthread`
- Come si usa
  - Lanciare `./scheduler`
  - Premendo INVIO, vengono lanciati `THREAD_BURST` thread che tentano di accedere in parallelo a `NUM_RESOURCES` risorse e le usano per processare ciascuno `NUM_TASKS` work item
  - Un work item richiede un tempo random compreso tra `0` e `MAX_SLEEP` secondi
  - Premendo CTRL+D, il programma termina
  - Osservare l'interleaving dei vari thread e il fatto che non ci sono mai nello stesso momento più di `NUM_RESOURCES` thread che hanno accesso ad una delle risorse

Accesso in mutua esclusione a N risorse - Perché funziona

- Inizializzando il semaforo a N, i primi N thread che eseguiranno la `sem_wait()` vedranno un valore non negativo dopo il proprio decremento e potranno accedere alla sezione critica
- I thread successivi effettueranno un decremento (valore del semaforo negativo) e verranno messi in attesa in coda
- Quando uno dei primi N thread esegue la `sem_post()`, il semaforo viene incrementato; siccome il valore era negativo, esso al massimo può diventare 0, e quindi uno dei thread in coda viene svegliato
  - Gli altri thread in coda rimangono lì in attesa
- Ad ogni successiva `sem_post()`, il semaforo viene incrementato
  - Finchè il semaforo non diventa positivo, vuol dire che ci sono thread in coda che verranno svegliati ad ogni `sem_post()`

Accesso in mutua esclusione a N risorse - Esercizio

- Modificare il codice per implementare la seguente semantica
  - Invece di usare la risorsa per processare ininterrottamente tutti i work item, ogni thread deve rilasciare la risorsa dopo aver completato una coppia di work item, e rimettersi quindi in coda per ottenere nuovamente l'accesso ad una risorsa
  - Una volta acquisita una risorsa, il thread deve completare la coppia successiva di work item e così via
  - Rilasciare definitivamente ogni risorsa dopo che tutti i work item sono stati processati