Corso «Sistemi Operativi» AA 2020-2021

Marco Tessarotto

Patterns di sincronizzazione, pthreads

indice

- pthread mutex statico (fatto)
- pthread mutex allocate dinamicamente: da fare
- condition variables statiche: da fare
- condition variables allocate dinamicamente: da fare (tbc)

- basic synchronization patterns:
- signaling: fatto
- rendezvous: fatto
- barrier: da fare
- reusable barrier (tbc): da fare
- queue (tbc): da fare
- producer/consumer (tbc): da fare
- readers-writers (tbc): da fare

semaforo

- È un numero intero che viene mantenuto sempre >= 0
- incremento del semaforo: post(), signal() [sinonimi]
- decremento del semaforo: wait()
- post e wait sono operazioni «atomiche» (indivisibili)
- L'operazione post non blocca mai il thread chiamante
- operazione wait: se semaforo > 0, decrementa il semaforo e ritorna; altrimenti blocca il thread chiamante (in attesa che semaforo sia incrementato da qualche altro thread)
- ci possono essere N (>= 0) threads in attesa ad un semaforo

mutex

- Un mutex impedisce a più thread di accedere contemporaneamente a una risorsa condivisa (es. una variabile globale)
- Stati di un mutex: «locked», «unlocked» (chiuso, aperto)
- Operazioni su mutex: «lock», «unlock» (chiusura, apertura)
- Appena creato, il mutex è in stato «unlocked» (aperto)

- Quando un thread «chiude» (lock) un mutex, ne diventa «proprietario», fino a quando lo «apre» (unlock)

mutex

- Un singolo thread non può chiudere/bloccare (lock) lo stesso mutex due volte
- Un thread non può aprire/sbloccare (unlock) un mutex che di cui non è proprietario (cioè di cui non ha acquisito il «lock» in precedenza).
- Un thread non può sbloccare un mutex che non è attualmente bloccato (non può invocare «unlock» su un mutex che è in stato «unlocked»)

- Una variabile di condizione consente a un thread di informare gli altri thread sui cambiamenti nello stato di una variabile condivisa (o di un'altra risorsa condivisa) e consente agli altri thread di attendere (bloccarsi) tale notifica.

signaling

Thread A

```
a1  statement a1
a2  sem.signal()
```

Thread B

```
b1  sem.wait()
b2  statement b2
```

Il semaforo garantisce che A1 sia completato prima dell'inizio di B2

A1 < B2

Tratto da 'Little book of semaphores', pag. 11

rendezvous

Thread A

```
a1 statement a1
a2 statement a2
```

Thread B

```
b1 statement b1
b2 statement b2
```

Obiettivo: (A1 < B2) AND (B1 < A2)

(A1 sia completato prima dell'inizio di B2) AND (B1 sia completato prima dell'inizio di A2)

Thread A

```
a1 statement a1
a2 aArrived.signal()
a3 bArrived.wait()
a4 statement a2
```

Thread B

```
b1 statement b1
b2 bArrived.signal()
b3 aArrived.wait()
b4 statement b2
```

aArrived e bArrived sono semafori entrambi inizializzati a 0

Tratto da 'Little book of semaphores', pag. 12

barrier

- Barrier è un «rendezvous» generalizzato (per più di due threads)
- Il requisito di sincronizzazione è che **nessun thread esegua «critical point» fino a quando tutti i thread non hanno eseguito «rendezvous»**.
- ipotesi: ci sono n thread e questo valore è memorizzato in una variabile, n, accessibile da tutti i thread.
- Quando arrivano i primi n - 1 thread, dovrebbero eseguire «rendezvous» e poi bloccarsi fino all'arrivo dell' n-mo thread; a quel punto **tutti** gli n thread possono procedere ed eseguire «critical point»

Barrier code

```
1 rendezvous
2 critical point
```

Tratto da 'Little book of semaphores', pag. 21

barrier

Barrier hint

```
1 n = the number of threads
2 count = 0
3 mutex = Semaphore(1)
4 barrier = Semaphore(0)
```

count tiene traccia di quanti thread sono arrivati.

mutex fornisce l'accesso esclusivo a count (in modo che i thread possano incrementarlo in modo sicuro)

barrier è bloccato (= 0) fino all'arrivo di tutti i thread; quindi deve essere sbloccato (valore >= 1).

Tratto da ‘Little book of semaphores’, pag. 23

barrier

Barrier solution

```
1  rendezvous
2
3  mutex.wait()
4      count = count + 1
5  mutex.signal()
6
7  if count == n: barrier.signal()
8
9  barrier.wait()
10 barrier.signal()
11
12 critical point
```

«tornello»

Tratto da ‘Little book of semaphores’, pag. 29

Turnstile (tornello)

barrier.wait()

barrier.signal()

- questo schema, wait e signal in rapida successione, si chiama tornello, perché consente il passaggio di un thread alla volta e può essere bloccato per bloccare tutti i thread.
- nel caso di «barrier», il tornello è bloccato (valore iniziale zero) ed i primi n-1 thread si bloccano lì
- quando arriva il thread n-mo, sblocca il tornello e tutti gli n thread passano attraverso.

Reusable barrier

- Spesso un insieme di thread cooperanti eseguirà una serie di passaggi in un ciclo e si sincronizzerà su una barriera dopo ogni passaggio.
- Per questa applicazione abbiamo bisogno di una barriera riutilizzabile che si blocchi dopo che tutti i thread sono passati.

Reusable barrier hint

```
1 turnstile = Semaphore(0)
2 turnstile2 = Semaphore(1)
3 mutex = Semaphore(1)
```

Tratto da ‘Little book of semaphores’, pag. 31

Reusable barrier

Reusable barrier solution

```
# rendezvous

mutex.wait()
    count += 1
    if count == n:
        turnstile2.wait()       # lock the second
        turnstile.signal()      # unlock the first
mutex.signal()

turnstile.wait()                # first turnstile
turnstile.signal()

# critical point

mutex.wait()
    count -= 1
    if count == 0:
        turnstile.wait()         # lock the first
        turnstile2.signal()      # unlock the second
mutex.signal()

turnstile2.wait()                # second turnstile
turnstile2.signal()
```

Questa soluzione viene talvolta chiamata barriera a due fasi perché forza tutti i thread ad attendere due volte: una volta per l'arrivo di tutti i thread (primo tornello) e di nuovo affinché tutti i thread eseguano la sezione critica (secondo tornello)

Tratto da ‘Little book of semaphores’, pag. 41