Laboratorio di Reti e Sistemi Distribuiti

14: Task Synchronization

Roberto Marino, PhD[1]
roberto.marino@unime.it

[1]Dipartimento di Matematica, Informatica, Fisica e Scienze della Terra
Future Computing Research Laboratory
Università di Messina

Last Update: 15th April 2025

Introduzione

Necessità di protezione

In un sistema a **memoria condivisa** bisogna sempre assicurare che le risorse siano protette da **accessi concorrenti**. Questo avviene perchè più thread di esecuzione possono accedere e manipolare i dati nello stesso istante e/o trovare i dati in uno stato **inconsistente**.

Prima dell'avvento del Simmetrical Multi Processing (SMP) questo obiettivo era di facile soluzione. Con l'avvento dell'SMP (kernel 2.0) e dello scheduler **preemptive** (kernel 2.6, diritto di prelazione, a.k.a. diritto di preferenza) le cose si sono fatte molto più complesse.

Regioni Critiche e Race Conditions

Def. Regione Critica

Sequenza di istruzioni (thread di esecuzione) che accede e manipola dati **condivisi**

Def. Atomicità

Per prevenire accessi concorrenti il programmatore deve essere certo che l'accesso alla regione critica avvenga in modo atomico, cioè come se fosse una unica, indivisibile, istruzione eseguita senza interruzioni.

Def. Race Condition

Se non si garantisce atomicità, due flussi di esecuzione possono trovarsi simultaneamente nella stessa regione critica e si trovano in una condizione di competizione chiama "race condition"

Def. Sincronizzazione

Prevenire una race condition significa "sincronizzare" le sequenze di istruzioni

Esempio

Consideriamo una semplice risorsa condivisa, una variabile intera, ed una semplice regione critica, ossia l'operazione di incremento su di essa:

```
1    int i; /* risorsa condivisa */
2    i++;   /* regione critica */
```

A livello macchina questa viene tradotta in:

1. Prendi il valore corrente di i dalla memoria e copialo in un registro (**get**)
2. Aggiungi uno al valore (**increment**)
3. Copia il nuovo valore di i in memoria (**write back**)

Esempio

Supponiamo di avere due thread che eseguono entrambi la stessa regione critica di codice e che la variabile sia inizializzata a 7.
Il risultato desiderato è il seguente:

Thread 1:
get i (7)
increment i ($7 \rightarrow 8$)
write back i (8)
–
–
–

Thread 2:
–
–
–
get i (8)
increment i ($8 \rightarrow 9$)
write back i (9)

Critical Region Interleaving

In realtà, senza alcun meccanismo di sincronizzazione, otteniamo qualcosa del genere (inconsistenza):

Thread 1:
get i (7)
increment i ($7 \rightarrow 8$)
–
write back i (8)
–
–

Thread 2:
get i (7)
–
increment i ($7 \rightarrow 8$)
–
write back i (8)

Atomic Operation

Fortunatamente tutti i processori moderni posseggono delle istruzioni di incremento atomiche cioè che permettono di leggere, incrementare e riscrivere in memoria in un'unico, indivisibile, passo di esecuzione (no interleaving).

Thread 1:
increment i ($7 \rightarrow 8$)
–

Thread 2:
–
increment i ($8 \rightarrow 9$)

oppure:

Thread 1:
–
increment i ($8 \rightarrow 9$)

Thread 2:
increment i ($7 \rightarrow 8$)
–

Locking

Supponiamo ora di avere non una variabile ma una **struttura dati condivisa**. Come fare in modo di agire atomicamente su di essa evitando stati inconsistenti della memoria? Bisogna pensare a dei **meccanismi software** che permettano l'accesso atomico (indivisibile) alle strutture dati (qualunque esse siano).

Locking

Immaginiamo una stanza dietro ad una porta come una sezione critica (ad esempio un bagno). La serratura funziona da **lock**. Due persone vogliono entrare nella stanza. La prima che arriva si chiude dietro la porta (**acquisisce il lock**) a chiave mentre la seconda aspetta. Appena finito agisce nuovamente sulla serratura, aprendola (**rilascia il lock**). Chi stava aspettando dietro la porta può ora entrare.

Lock Example

Thread 1:
T1: try to lock the queue
T2: succeeded: acquired lock
T3: access queue...
T4: unlock the queue
T5: ...
T6: ...
T7: ...

Thread 2:
T1: try to lock the queue
T2: failed: waiting...
T3: waiting...
T4: waiting...
T5: succeeded: acquired lock
T6: access queue...
T7: unlock the queue

Implementazione a basso livello

I lock e tutti i meccanismi di sincronizzazione si basano su istruzioni atomiche specifiche del processore (test-and-set: settano un intero ad uno solo se zero. Zero significa unlocked). **Se si eseguono contemporaneamente due istruzioni test-and-set, ciascuna in una unità di elaborazione diversa, queste vengono eseguite in modo sequenziale ed in ordine arbitrario**.

Cosa causa concorrenza?

- Se due thread accedono alla stessa regione critica dopo un evento di preemption, su una macchina monoprocessore siamo in presenza di **pseudo-concorrenza**.
- Se due thread accedono alla stessa regione critica su una macchina SMP, quindi potenzialmente nello stesso identico istante di tempo, siamo in presenza di **concorrenza reale**.

Cosa causa concorrenza?

1. Interrupts: un interrupt può occorrere in ogni momento.
2. Kernel Preemption: un task può essere interrotto dallo scheduler
3. SMP: due o più processori/core possono eseguire regioni critiche nello stesso identico istante

Il nostro obiettivo è realizzare codice **interrupt-safe**, **SMP-safe** e **Preemption-safe**.

Uso improprio dei lock

La programmazione concorrente è difficile!

Deadlock

Uno o più threads vogliono accedere ad uno o più risorse ma ognuna delle risorse è in stato di lock (Esempio dell'incrocio e delle quattro automobili)

Starvation

Un thread non riesce mai ad accedere alla sezione critica alla quale vuole accedere, e muore di fame! (starvation)

SpinLock vs Semafori/Mutex

A livello kernel esistono due tipologie principali di lock:

SpinLock

Il thread che resta fuori dalla sezione critica controlla in busy-loops (spins) finchè non riesce ad acquisire il lock. Velocissimo ma consuma CPU. Permette un unico accesso alla volta.

Semafori/Mutex

Il thread che resta fuori dalla sezione critica viene messo in SLEEP. Un semaforo permette N accessi simultanei. Se N=1 siamo in presenza di un mutex (mutua esclusione). Più lento ma non consuma CPU.

Semafori vs Mutex

Mutex (Semaforo binario)

Può assumere solo due stati lock/unlocked. La transizione tra stati è atomica. Permette l'accesso in sezione critica ad un solo thread per volta. L'altro viene messo in SLEEP.

Semaforo (inizializzato ad N)

Ad ogni acquisizione il valore del semaforo viene decrementato atomicamente di uno, ad ogni rilascio viene incrementatoatomicamente. Quando il semaforo arriva a zero non permette più accessi. Quindi N thread possono accedere simultaneamente alla sezione critica, **e sono in potenziale race condition**.

Use Case 1: Accessi limitati

Quando più thread o processi devono accedere a un numero limitato di risorse (come connessioni a un database o stampanti), i semafori possono limitare il numero di accessi simultanei. Ad esempio, se ci sono solo tre connessioni disponibili a un database, un semaforo può essere inizializzato a 3. Ogni volta che un thread vuole utilizzare una connessione, decrementa il semaforo; quando termina, lo incrementa, garantendo che non più di tre thread accedano contemporaneamente al database.

Use Case 2: Sincronizzazione tra produttore e consumatore

Nel problema classico del produttore-consumatore, un produttore genera dati e un consumatore li utilizza. I semafori **possono sincronizzare questi processi**, assicurando che il consumatore attenda finché non ci sono dati disponibili e che il produttore attenda se il buffer è pieno.

Quanti semafori servono?

Sincronizzazione thread in avvio (Barrier)

Quando un'applicazione deve avviare più thread ma vuole che tutti inizino l'esecuzione simultaneamente dopo una certa inizializzazione, **un semaforo può far attendere tutti i thread finché non sono pronti, rilasciandoli poi tutti insieme**.
Quanti semafori servono, ed inizializzati a che valore?

Conditions Variables

Abbiamo visto che:

- i mutex sono utili per gestire la mutua esclusione, cioè l'ingresso atomico in regione critica, ma non ci danno controllo sull'ordine di ingresso.
- i semafori permettono di contare quanti thread sono in sezione critica, senza gestire l'accesso concorrente, finchè il semaforo ha valore maggiore di zero.

Manca però un meccanismo esplicito che ci permetta di mettere in attesa un thread fino a quando non si verifica una condizione. Questo meccanismo prende il nome di **condition variables**

Inizializzazione statica mutex/conditions

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t q = PTHREAD_COND_INITIALIZER;
```

In alternativa si possono inizializzare dinamicamente con:

```
1 pthread_mutex_init(&lock, NULL);
2 pthread_cond_init(&q, NULL);
```

Waiting su condizione

```
1 int pthread_cond_wait(pthread_cond_t *q, pthread_mutex_t *mutex);
```

Sospende il thread chiamante e rilascia il `mutex`, il tutto atomicamente. Il thread verrà risvegliato solo dopo una chiamata a `pthread_cond_signal` o `pthread_cond_broadcast`, e solo dopo aver riacquisito il `mutex`.

Nota: è essenziale controllare la condizione in un ciclo `while` per evitare risvegli spurii (interrupt, risvegli di massa con condizione non vera).

Waiting in quattro passi

1. Il thread rilascia il mutex (in modo atomico) e si mette in attesa sulla condition variable.
2. Quando un altro thread chiama `pthread_cond_signal` o `pthread_cond_broadcast`, il thread in attesa viene svegliato, ma...
3. Non riparte subito! Prima deve riacquisire il mutex.
4. Solo dopo che ha riottenuto il lock, riprende l'esecuzione esattamente dopo la chiamata a `pthread_cond_wait`.

Il **while** serve per evitare spourious wake-up, il **mutex** per evitare race condition sulla condizione

Signaling

```
1 int pthread_cond_signal(pthread_cond_t *q);
```

Risveglia **uno** dei thread sospesi sulla variabile di condizione.

```
1 int pthread_cond_broadcast(pthread_cond_t *cond);
```

Risveglia **tutti** i thread sospesi su quella condition variable.

```
1 int pthread_cond_timedwait(pthread_cond_t *q,
2                            pthread_mutex_t *mutex,
3                            const struct timespec *abstime);
```

Come `pthread_cond_wait`, ma permette di impostare un timeout assoluto.

Destroy

```
1 pthread_cond_destroy(&cond);
2 pthread_mutex_destroy(&lock);
```

Queste funzioni vanno chiamate solo dopo che nessun thread sta più usando il mutex o la condition variable.

Considerazioni avanzate

- `pthread_cond_t` è una struttura dati, non una variabile binaria. Può essere immaginata come una **area di sosta** cui il thread si iscrive in attesa che si verifichi la condizione.
- I **risvegli spurii** sono previsti dallo standard POSIX
- E' una coda ma non è garantito il comportamento deterministico, nel caso di broadcast.
- La condizione reale da verificare non è "inclusa" nella **condition variable**: va mantenuta separatamente e le funzioni di libreria le agiscono "intorno"

Codice d'esempio

```
1  pthread_mutex_t lock;
2  pthread_cond_t q;
3  int condizione = 0;
4
5  void* thread_attesa(void* arg) {
6      pthread_mutex_lock(&lock);
7      while (!condizione) {
8          pthread_cond_wait(&q, &lock);  // rilascia lock, attende
       segnale, poi riacquisisce lock
9      }
10     // condizione vera: fai qualcosa
11     pthread_mutex_unlock(&lock);
12     return NULL;
13 }
```

Codice d'esempio

```
1 void* thread_segnalatore(void* arg) {
2     pthread_mutex_lock(&lock);
3     condizione = 1;
4     pthread_cond_signal(&q); // oppure pthread_cond_broadcast(&cond);
5     pthread_mutex_unlock(&lock);
6     return NULL;
7 }
```