LABORATORIO DI RETI E SISTEMI DISTRIBUITI

HANDS-ON 11

Meccanismi di sincronizzazione

Antonio Mastrolembo Ventura

543644

April 11, 2025

Indice

1 Introduzione

In questo hands-on andremo a vedere l'importanza dei meccanismi di sincronizzazione nell'ambito della programmazione concorrente. Quando più thread accedono e modificano una risorsa condivisa, come una variabile globale, si possono verificare delle **race condition**. Questo accade perché le operazioni che sembrano atomiche (come l'incremento di un intero) possono essere interrotte e sovrapposte tra thread, portando a risultati errati. Per evitare questi problemi, è necessario proteggere le cosiddette **sezioni critiche**, ovvero le parti di codice in cui si accede o si modifica una risorsa condivisa, utilizzando meccanismi di sincronizzazione come mutex e semafori. In questa esercitazione andremo ad analizzare il comportamento di un programma multi-thread con e senza l'utilizzo di questi strumenti, osservando l'impatto che essi hanno sulla correttezza e sul risultato finale.

Tempo	**Thread T1**	**Thread T2**	**Valore di counter**
T0	Legge `counter` (0)		0
T1		Legge `counter` (0)	0
T2	Incrementa (0 + 1 = 1)		0
T3		Incrementa (0 + 1 = 1)	0
T4	Scrive 1 in `counter`		1
T5		Scrive 1 in `counter`	1

Table 1: Esempio di race condition tra due thread

2 Definizione del problema

Per dimostrare come l'utilizzo di questi strumenti sia fondamentale per garantire che non ci siano incongruenze o accessi contemporanei a variabili/dati condivisi realizzeremo tre programmi differenti utilizzando i **lock** ed i **semafori**. Nel primo esercizio vedremo un tipico scenario in cui non abbiamo meccanismi di protezione dalla sezione critica e dunque vedremo una vera e propria race

condition. Nel secondo esercizio effettueremo delle modifiche introducendo i lock mutex e vedremo come questi strumenti ci permettono di risolvere il problema delle race condition; infine, nell'ultimo esercizio vedremo come permettere a più thread (indicati da noi) di poter stare in sezione critica contemporaneamente.

3 Metodologia e risultati

Realizzeremo tre programmi: `NoSync.c`, `Sync.c` e `Semaphore.c` ed analizzeremo i risultati ottenuti.

3.1 Esercizio 1

In questo primo esercizio `NoSync.c` vedremo uno scenario tipico in cui non utilizzeremo dei meccanismi di sincronizzazione.

3.1.1 Variabile globale

```
1 #define NUM_THREADS 10
2 #define MAX_VALUE 10000
3
4 int i = 0;
```

Per prima cosa definiamo un numero massimo di thread `NUM_THREADS` ed un numero `MAX_VALUE` che indica quanti incrementi effettuerà ogni thread. Successivamente inizializziamo una variabile **globale** `i` mettendola fuori da ogni funzione del programma, in questo modo sarà visibile ed accessibile da ogni thread.

3.1.2 Funzione counter

```
void* counter(void *arg){
    int t = *(int*)arg;
    for (int j = 0; j < MAX_VALUE; j++){
        printf("Thread %d: i = %d\n", t, i);
        i++;
    }
    pthread_exit(NULL);
}
```

Questa è la funzione che eseguirà ogni thread, incrementando la variabile `i` per un numero di volte pari a `MAX_VALUE`.

3.1.3 Funzione principale

```
int main(){
    pthread_t threads[NUM_THREADS];

    for (int t = 0; t < NUM_THREADS; t++){
        pthread_create(&threads[t], NULL, counter, (int*)&t);
    }

    for (int t = 0; t < NUM_THREADS; t++){
        pthread_join(threads[t], NULL);
    }

    printf("Final i = %d\n", i);

    pthread_exit(NULL);
    return 0;
}
```

Abbiamo adesso il main del nostro programma che creerà 10 thread in esecuzione contemporaneamente ai quali passiamo la funzione definita al punto 3.1.2.

3.1.4 Output del programma

Ci dovremmo aspettare come output il valore `100.000`, in quanto ogni thread (`10`) incrementerà di `10.000` il valore di `i`, vediamo cosa succede.

```
Thread 0: i = 0
Thread 1: i = 0
Thread 2: i = 1
Thread 0: i = 2
Thread 3: i = 2
Thread 4: i = 3
...
Thread 9: i = 9987
Thread 6: i = 9990
Thread 1: i = 9991
Final i = 99897
```

Listing 1: Output parziale di NoSync.c

Come possiamo osservare, l'output prodotto dai thread è interlacciato e non ordinato, a causa dell'esecuzione concorrente. Inoltre, il valore finale di `i` non è pari a `100.000` come ci saremmo aspettati, ma più basso (nell'esempio `99897`). Questo è dovuto al fatto che più thread accedono e modificano la variabile condivisa `i` simultaneamente, generando delle race condition di cui abbiamo parlato prima.

3.2 Esercizio 2

Andiamo ora a modificare l'esercizio precedente introducendo uno strumento di sincronizzazione: il `mutex`. Questo ci permetterà di proteggere l'accesso alla variabile condivisa `i`, evitando che più thread possano modificarla contemporaneamente e causando quindi incongruenze.

3.2.1 Variabile mutex

```
pthread_mutex_t lock;
```

Dichiariamo la variabile globale `lock` di tipo `pthread_mutex_t` in quanto dovrà essere visibile da ogni thread.

3.2.2 Inizializzazione del mutex

```
pthread_mutex_init(&lock, NULL);
```

A questo punto inizializziamo il mutex prima di utilizzarlo passando come parametro alla funzione `pthread_mutex_init` il mutex da inizializzare ed eventuali attributi.

3.2.3 Protezione della sezione critica

```
    // Acquisisco il lock
    pthread_mutex_lock(&lock);

    for (int j = 0; j < MAX_VALUE; j++){
        printf("Thread %d: i = %d\n", t, i);
        i++;
    }

    // Rilascio il lock
    pthread_mutex_unlock(&lock);
```

Una volta individuata la sezione critica, ovvero il blocco di codice in cui avviene l'accesso e la modifica della variabile condivisa `i`, possiamo utilizzare il mutex `lock` per proteggerla: `pthread_mutex_lock` è la funzione che permette ad un thread di acquisire il lock; a questo punto, il sistema operativo verifica se il mutex è libero oppure occupato da un altro thread: se il mutex è **libero**, viene acquisito dal thread corrente che può ora eseguire il codice presente nella sezione critica, se invece il mutex è **occupato** il thread viene messo in sleep e sarà risvegliato solo quando il mutex viene rilasciato.
La funzione `pthread_mutex_unlock` si occupa dunque di rilasciare il lock acquisito dal thread e permette ad un altro thread di poter entrare in sezione critica garantendo così la **mutua esclusione**.

3.2.4 Output del programma

Con l'aggiunta del lock mutex come meccanismo di sincronizzazione ci aspettiamo che l'output sia corretto.

```
Thread 0: i = 0
Thread 0: i = 1
...
Thread 0: i = 9999
Thread 5: i = 10000
Thread 5: i = 10001
...
Thread 5: i = 19999
...
Thread 9: i = 90000
Thread 9: i = 90001
...
Thread 9: i = 99999
Final i = 100000
```

Listing 2: Output parziale di Sync.c

Come possiamo vedere il risultato finale è corretto. Non necessariamente i thread verranno eseguiti in ordine, in quanto dipende dal thread che acquisisce

il lock per primo, l'importante è che l'accesso alla variabile condivisa avvenga in modo esclusivo per ogni thread e che la mutua esclusione sia garantita, come nel nostro caso.

3.3 Esercizio 3

Per il terzo esercizio vediamo un meccanismo di sincronizzazione diverso dai lock mutex, che ci permette di gestire un numero arbitrario di thread che possono entrare in sezione critica: i **semafori**.

3.3.1 Definizione del semaforo

```
1 #include <semaphore.h>
2
3 sem_t sem;
```

Per prima cosa includiamo la libreria `semaphore.h` e definiamo una variabile di tipo `sem_t` chiamata `sem`.

3.3.2 Inizializzazione del semaforo

```
1     sem_init(&sem, 0, 2);
```

Nel main andiamo ad inizializzare il semaforo con la funzione `sem_init()` alla quale passiamo come argomenti il puntatore alla variabile semaforo che vogliamo inizializzare, un valore (0) che indica se il semaforo sarà utilizzato tra thread all'interno dello stesso processo (nel nostro caso si) ed il valore iniziale del semaforo (2), che rappresenta il numero massimo di thread che possono entrare simultaneamente nella sezione critica.

3.3.3 Gestione della sezione critica

```
    sem_wait(&sem);

    printf("Thread %d: entra nella sezione critica\n", i);
    sleep(2);
    printf("Thread %d: esce dalla sezione critica\n", i);

    sem_post(&sem);
```

All'interno della funzione eseguita da ogni thread, individuiamo la sezione critica e ne limitiamo l'accesso attraverso l'uso del semaforo. La chiamata a `sem_wait()` decrementa il valore del semaforo: se il valore è maggiore di zero, il thread può entrare nella sezione critica; se è uguale a zero, viene bloccato in attesa. In questo caso, il semaforo è inizializzato con il valore `2`, quindi può consentire l'accesso a due thread contemporaneamente. Questo tipo di semaforo non garantisce la mutua esclusione ma controlla quanti thread possono trovarsi contemporaneamente in sezione critica.
Una volta eseguite le istruzioni nella sezione critica il thread, tramite la chiamata `sem_post()`, incrementa il valore del semaforo permettendo eventualmente a un altro thread in attesa di entrare. In questo modo si garantisce che al massimo due thread siano attivi nella sezione critica nello stesso momento.

3.3.4 Deallocazione del semaforo

```
    sem_destroy(&sem);
```

Una volta che tutti i thread hanno terminato la loro esecuzione e il semaforo non è più necessario, deallochiamo le risorse ad esso associate utilizzando la funzione `sem_destroy()`. Questa chiamata prende come parametro il puntatore al semaforo e ne libera la memoria interna allocata dal sistema.

3.3.5 Output del programma

```
Thread 0: entra nella sezione critica
Thread 1: entra nella sezione critica
Thread 0: esce dalla sezione critica
Thread 2: entra nella sezione critica
Thread 1: esce dalla sezione critica
Thread 3: entra nella sezione critica
Thread 2: esce dalla sezione critica
Thread 4: entra nella sezione critica
Thread 3: esce dalla sezione critica
Thread 4: esce dalla sezione critica
```

Listing 3: Output di Semaphore.c

Come possiamo vedere dall'output del programma, due thread al massimo sono in sezione critica contemporaneamente, ed appena si "libera un posto" un nuovo thread è pronto ad entrare.

4 Conclusioni

In questa esercitazione abbiamo analizzato il comportamento di un programma multithread in tre scenari diversi, osservando l'importanza dei meccanismi di sincronizzazione.

Nel primo esercizio, abbiamo omesso qualsiasi forma di protezione sull'accesso alla variabile condivisa. Il risultato è stato una race condition evidente, con un valore finale errato della variabile a causa di scritture simultanee da parte di più thread.

Nel secondo esercizio abbiamo introdotto un mutex, garantendo così la mutua esclusione all'interno della sezione critica. In questo caso l'output è risultato corretto e coerente, a dimostrazione del fatto che l'uso dei lock permette di gestire l'accesso concorrente in modo sicuro.

Infine, nel terzo esercizio, abbiamo implementato un semaforo per consentire

l'accesso contemporaneo alla sezione critica a un numero limitato di thread (massimo due). Pur non garantendo la mutua esclusione, i semafori sono degli strumenti molto utili per consentire ad esempio l'accesso ad una data risorsa in modo limitato.