LABORATORIO DI RETI E SISTEMI DISTRIBUITI

HANDS-ON 12

Semafori

Antonio Mastrolembo Ventura

543644

April 16, 2025

Indice

1 Introduzione

Nell'hands-on precedente abbiamo visto due meccanismi di sincronizzazione: i mutex ed i semafori. In questa esercitazione andremo ad utilizzare nuovamente i semafori per risolvere diversi problemi classici di concorrenza e sincronizzazione. A differenza dei mutex, che consentono l'accesso esclusivo a una risorsa condivisa da parte di un solo thread alla volta, i semafori utilizzano un contatore interno per regolare il numero di thread che possono accedere simultaneamente a una determinata risorsa. Possono quindi essere utilizzati sia per implementare la mutua esclusione (semafori binari), sia per regolare l'accesso concorrente a risorse limitate.

2 Definizione del problema

In questa esercitazione andremo a vedere due problemi classici della programmazione concorrente, entrambi utilizzando i semafori come meccanismo di sincronizzazione.

Il primo problema è la **barriera di sincronizzazione**, in cui l'obiettivo è fare in modo che un gruppo di thread inizi l'esecuzione simultaneamente solo quando tutti sono pronti. Questo tipo di sincronizzazione è utile ad esempio in contesti in cui i thread devono attendersi prima di procedere a una fase comune di lavoro.

Il secondo problema è il classico **produttore-consumatore** con buffer limitato, in cui un thread (produttore) genera dati e li inserisce in un buffer, mentre un altro thread (consumatore) preleva tali dati per elaborarli. Un sistema del genere deve garantire che il produttore non scriva nel buffer quando è pieno e che il consumatore non legga dal buffer quando è vuoto.

3 Metodologia e risultati

Realizzeremo gli esercizi attraverso il linguaggio C, i programmi scritti saranno: `Barrier.c` e `ProdCons.c`.

3.1 Esercizio 1

In questo primo esercizio vediamo come costruire una barriera di sincronizzazione che ci permette di far partire N thread contemporaneamente.

3.1.1 Definizione del semaforo

```
sem_t barrier;
```

Per prima cosa, definiamo il semaforo che ci servirà per implementare la barriera di sincronizzazione. La variabile `barrier` è di tipo `sem_t` e rappresenta il meccanismo che verrà utilizzato da tutti i thread per attendere un segnale comune di partenza.

3.1.2 Inizializzazione del semaforo

```
    sem_init(&barrier, 0, 0);
```

Nel main andiamo ad inizializzare il semaforo con la funzione `sem_init()` impostandolo a `0`, adesso vedremo meglio perché.

3.1.3 Creazione dei thread

```
    for (int i = 0; i < NUM_THREADS; i++) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, threadFunc, &ids[i]);
    }
```

Andiamo adesso a creare i thread passando come argomento la funzione da eseguire.

3.1.4 Funzione dei thread

```
void* threadFunc(void* arg) {
    int id = *(int*)arg;

    printf("Thread %d: in attesa di partire...\n", id);

    sem_wait(&barrier);

    printf("Thread %d: partito!\n", id);
    return NULL;
}
```

Avendo impostato il semaforo a zero, tutti i thread che eseguiranno la funzione `sem_wait()` su di esso verranno bloccati in attesa. Solo quando il thread principale eseguirà un numero sufficiente di `sem_post()` (pari al numero di thread), essi potranno sbloccarsi e iniziare l'esecuzione simultaneamente.

3.1.5 Avvio simultaneo dei thread

```
    sleep(2);
    printf("MAIN: Tutti i thread possono partire!\n");

    // Sblocco tutti i thread con N sem_post()
    for (int i = 0; i < NUM_THREADS; i++) {
        sem_post(&barrier);
    }
```

Dopo una breve pausa simulata con `sleep(2)`, il thread principale esegue un numero di chiamate a `sem_post()` pari al numero totale di thread. Ogni chiamata a `sem_post()` incrementa il valore del semaforo di uno, consentendo così a un thread in attesa su `sem_wait()` di sbloccarsi e proseguire l'esecuzione. Poiché il semaforo è stato inizializzato a zero, tutti i thread erano rimasti bloccati fino a questo momento, di conseguenza tutti i thread vengono sbloccati quasi contemporaneamente e iniziano l'esecuzione in modo sincronizzato, simulando un avvio simultaneo.

3.1.6 Output del programma

L'output del programma sarà il seguente:

```
Thread 0: in attesa di partire...
Thread 1: in attesa di partire...
Thread 3: in attesa di partire...
Thread 2: in attesa di partire...
Thread 4: in attesa di partire...
MAIN: Tutti i thread possono partire!
Thread 0: partito!
Thread 1: partito!
Thread 4: partito!
Thread 2: partito!
Thread 3: partito!
```

Inizialmente tutti i thread saranno in attesa ma dopo un paio di secondi partiranno tutti contemporaneamente seguendo la logica della barriera.

3.2 Esercizio 2

In questo secondo esercizio vediamo uno dei grandi classici dei problemi di sincronizzazione e di concorrenza: il problema del produttore-consumatore con buffer limitato.

3.2.1 Buffer condiviso

```
int buffer[BUFFER_SIZE];

int in = 0;
int out = 0;
```

Per prima cosa definiamo il buffer condiviso tra i due thread (il produttore ed il consumatore) e inizializziamo due variabili di aiuto che serviranno per tenere traccia delle posizioni di lettura e scrittura all'interno del buffer. La variabile `in` rappresenta l'indice della prossima posizione libera dove il produttore inserirà un nuovo elemento, mentre `out` rappresenta la posizione da cui il consumatore preleverà il prossimo elemento da consumare.

3.2.2 Strumenti di sincronizzazione

```
sem_t empty;

sem_t full;

pthread_mutex_t lock;
```

Per gestire correttamente l'accesso al buffer e le modalità con cui sia il produttore che il consumatore accederanno, realizziamo due semafori: `empty` e `full`. Il semaforo `empty` tiene traccia del numero di posizioni vuote (dunque libere) disponibili nel buffer, mentre il semaforo `full`, tiene traccia del numero di posizioni occupate nel buffer. Infine utilizziamo un mutex chiamato per proteggere l'accesso al buffer durante le operazioni di scrittura e lettura.

3.2.3 Inizializzazione dei semafori e del mutex

```
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&lock, NULL);
```

I semafori vengono inizializzati nel seguente modo: il semaforo `empty` viene inizializzato al valore `BUFFER_SIZE` poiché inizialmente il buffer è completamente vuoto. Il semaforo `full` è inizializzato a 0 perché all'inizio non ci sono elementi da consumare.

3.2.4 Produttore

```
void* producer(void* arg) {
    for (int i = 0; i < ELEMENTS; i++) {
        int elem = i;

        sem_wait(&empty);
        pthread_mutex_lock(&lock);

        buffer[in] = elem;
        printf("Produttore: inserito %d nella posizione %d\n", elem,
    in);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&lock);
        sem_post(&full);

        sleep(1);
    }
    return NULL;
}
```

Il produttore funziona nel seguente modo: in un ciclo che si ripete per un numero di volte pari a `ELEMENTS`, genera un nuovo elemento e tenta di inserirlo nel buffer.

Prima di poter inserire un elemento, esegue `sem_wait(&empty)`, in cui attende che ci sia almeno una posizione vuota nel buffer, se si decrementa il contatore delle posizioni vuote. Una volta che c'è spazio, entra nella sezione critica acquisendo il mutex tramite `pthread_mutex_lock()` per accedere in modo esclusivo al buffer.

All'interno della sezione critica, il produttore scrive l'elemento nella posizione `in`, stampa un messaggio di conferma e aggiorna l'indice `in` facendo in modo che, una volta finito il buffer, ricominci da capo nell'inserimento.

Dopo aver terminato l'inserimento, rilascia il mutex e segnala la disponibilità di un nuovo elemento incrementando il semaforo full con `sem_post(&full)`.

3.2.5 Consumatore

```
void* consumer(void* arg) {
    for (int i = 0; i < ELEMENTS; i++) {

        sem_wait(&full);
        pthread_mutex_lock(&lock);

        int elem = buffer[out];
        printf("Consumatore: estratto %d dalla posizione %d\n", elem,
    out);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&lock);
        sem_post(&empty);

        sleep(1);
    }
    return NULL;
}
```

Il consumatore, analogamente al produttore, esegue un ciclo per un numero di iterazioni pari a `ELEMENTS`, durante le quali consuma gli elementi presenti nel

buffer.

Per prima cosa esegue `sem_wait(&full)`, che lo blocca nel caso in cui il buffer sia vuoto, ovvero non ci siano elementi da consumare. Quando full è maggiore di zero, significa che c'è almeno un elemento pronto, e quindi il thread può procedere.

Ottenuto l'accesso, il consumatore entra nella sezione critica acquisendo il mutex, legge l'elemento presente nella posizione `out` del buffer, stampa un messaggio per mostrare l'elemento consumato e la posizione, ed infine aggiorna l'indice seguendo la stessa logica già vista per il produttore.

Dopo aver terminato la lettura, rilascia il mutex e segnala che ora c'è uno spazio libero nel buffer incrementando il semaforo empty con `sem_post(&empty)`.

3.2.6 Output del programma

L'output del programma con `BUFFER_SIZE` impostato a 5 e `ELEMENTS` impostato a 6 sarà il seguente:

```
Produttore: inserito 0 nella posizione 0
Consumatore: estratto 0 dalla posizione 0
Produttore: inserito 1 nella posizione 1
Consumatore: estratto 1 dalla posizione 1
Produttore: inserito 2 nella posizione 2
Consumatore: estratto 2 dalla posizione 2
Produttore: inserito 3 nella posizione 3
Consumatore: estratto 3 dalla posizione 3
Produttore: inserito 4 nella posizione 4
Consumatore: estratto 4 dalla posizione 4
Produttore: inserito 5 nella posizione 0
Consumatore: estratto 5 dalla posizione 0
```

Avendo messo uno `sleep(1)` alla fine del produttore e del consumatore vediamo come il produttore inserisce l'elemento nel buffer e questo viene immediatamente prelevato dal consumatore. Si potrebbe giocare con i tempi dello

sleep e vedere come, con buffer pieno, non vengono aggiunti nuovi elementi finché il consumatore non ne preleva almeno uno.

4 Conclusioni

In questa esercitazione abbiamo approfondito l'utilizzo dei semafori come meccanismo di sincronizzazione in programmazione concorrente. In particolare, abbiamo visto come un semplice semaforo possa essere utilizzato per implementare una barriera di sincronizzazione, permettendo a più thread di iniziare l'esecuzione in modo simultaneo. Successivamente, abbiamo affrontato il problema del produttore-consumatore con buffer finito, mostrando come sia possibile, tramite l'uso combinato di due semafori (`empty` e `full`) e un mutex, regolare l'accesso ad un buffer condiviso garantendo una giusta alternanza tra produzione e consumo di un elemento.