LABORATORIO DI RETI E SISTEMI DISTRIBUITI

HANDS-ON 13

Condition Variables

Antonio Mastrolembo Ventura

543644

April 17, 2025

Indice

1 Introduzione

Negli hands-on precedenti abbiamo affrontato due strumenti fondamentali per la sincronizzazione tra thread: i `mutex` e i `semafori`. Tuttavia, sia i mutex che i semafori non offrono un meccanismo diretto per far sì che un thread possa mettersi esplicitamente in attesa del verificarsi di una determinata condizione. Le **condition variables** (variabili di condizione) permettono a un thread di sospendersi volontariamente finché una certa condizione non è vera, e di essere successivamente risvegliato da un altro thread che modifica lo stato condiviso. Insieme ai mutex, le condition variables ci danno un potente meccanismo per gestire la sincronizzazione basata su eventi e condizioni.

2 Definizione del problema

In questa esercitazione affronteremo due problemi legati alla sincronizzazione tra thread, utilizzando le **condition variables** come nuovo strumento di coordinamento.

Il primo esercizio prevede la simulazione del comportamento della funzione `pthread_join()`, ipotizzando che non sia disponibile. L'obiettivo è far sì che il thread padre attenda correttamente la terminazione del thread figlio utilizzando una variabile di condizione, senza però preoccuparsi del valore di ritorno del thread figlio. In questo modo possiamo realizzare una forma di attesa controllata senza fare affidamento esclusivo sulle primitive della libreria POSIX.

Nel secondo esercizio dovremo implementare un meccanismo di **barriera di sincronizzazione** utilizzando le condition variables. L'obiettivo è fare in modo che quattro thread, una volta avviati, si blocchino finché tutti non sono pronti, e solo in quel momento inizino l'esecuzione contemporaneamente. Abbiamo già visto questo problema nell'hands-on precedente, stavolta lo affronteremo in modo differente, senza l'uso dei semafori.

3 Metodologia e risultati

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

3.1 Esercizio 1

Per il primo esercizio andremo a simulare il comportamento della funzione `pthread_join()`, facendo in modo che il thread padre (il main) attenda il completamento del thread figlio. Vediamo come implementarlo.

3.1.1 Mutex e Condition

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t queue = PTHREAD_COND_INITIALIZER;

int cond = 0;
```

Per prima cosa inizializziamo le strutture necessarie per la corretta implementazione del programma: abbiamo un mutex già visto più volte negli hands-on precedenti ed una variabile `queue` di tipo `pthread_cond_t` che, a differenza di come si possa pensare, non è la condizione vera e propria ma una struttura dati che rappresenta la "coda" su cui i thread in attesa dormiranno finché la condizione non sarà soddisfatta. La condizione logica vera e propria è nella variabile condivisa `cond`.

3.1.2 Waiting su condizione

```
int main(){
    pthread_t thread;

    pthread_mutex_lock(&lock);
    printf("Creazione del thread figlio...\n");
    pthread_create(&thread, NULL, thread_func, NULL);
```

```
7
8       while(!cond){
9           printf("Thread padre in attesa...\n");
10          pthread_cond_wait(&queue, &lock);
11      }
12
13      printf("Thread padre svegliato!\n");
14      pthread_mutex_unlock(&lock);
15
16      return 0;
17  }
```

Il padre acquisisce il lock prima della creazione del figlio, successivamente viene creato il thread figlio (che resterà in attesa finché il padre non rilascia il lock, vedremo a breve). A questo punto il padre entra in un ciclo di attesa per evitare il cosiddetto *spurious wake-up*, ovvero il fatto che un thread possa svegliarsi comunque dalla sua attesa (indipendentemente dalla condizione) a causa di eventi che il programma non controlla direttamente ma che fanno parte del comportamento ammesso dallo standard POSIX. In questo ciclo abbiamo la funzione atomica `pthread_cond_wait()` che sospende il thread chiamante, lo mette in attesa nella coda e rilascia il mutex fin quando non verrà risvegliato con una chiamata `pthread_cond_signal()` o `pthread_cond_broadcast`. Al ritorno acquisirà il mutex automaticamente e riprenderà l'esecuzione partendo dall'istruzione successiva.

3.1.3 Risveglio

```
1   void* thread_func(void* arg){
2       pthread_mutex_lock(&lock);
3       printf("Thread figlio in esecuzione...\n");
4
5       sleep(2);
6
7       printf("Thread figlio terminato!\n");
8       cond = 1;
```

```
9       pthread_cond_signal(&queue);
10      pthread_mutex_unlock(&lock);
11      return NULL;
12  }
```

Il thread figlio eseguirà la seguente funzione. Acquisce il lock rilasciato dal padre con la wait, esegue le proprie operazioni e, una volta terminate, aggiorna la condizione, risveglia **uno** dei thread presenti nella coda (in questo caso solo il padre) con la funzione `pthread_cond_signal()` e rilascia il lock. Adesso il padre, appena risvegliato, ripartirà verificando la condizione del ciclo while (avremo cond = 1) che sarà appunto soddisfatta e dunque potrà uscire dal while terminando la propria esecuzione.

3.1.4 Output del programma

```
1  Creazione del thread figlio...
2  Thread padre in attesa...
3  Thread figlio in esecuzione...
4
5  "sleep 2 sec..."
6
7  Thread figlio terminato!
8  Thread padre svegliato!
```

Come possiamo vedere dall'output, il padre attende correttamente l'esecuzione del thread figlio che, una volta terminato, rilascia il controllo al padre.

3.2 Esercizio 2

In questo secondo esercizio andremo a simulare il comportamento di una barriera di sincronizzazione utilizzando le condition variables. L'obiettivo è quello di far partire simultaneamente quattro thread, che resteranno in attesa finché una determinata condizione non verrà soddisfatta dal thread principale.

3.2.1 Mutex e Condition

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t queue = PTHREAD_COND_INITIALIZER;

int cond = 0;
```

Come per l'esercizio precedente, inizializziamo un mutex e una condition variable `queue`, che funge da struttura dati su cui i thread si metteranno in attesa. La condizione logica vera e propria è rappresentata dalla variabile `cond`, inizializzata a 0.

3.2.2 Funzione dei thread

```
void* thread_func(void* arg){
    int id = *(int*)arg;
    pthread_mutex_lock(&lock);
    printf("Thread %d in attesa...\n", id);
    while(!cond){
        pthread_cond_wait(&queue, &lock);
    }
    printf("Thread %d svegliato!\n", id);
    pthread_mutex_unlock(&lock);
    printf("Thread %d terminato!\n", id);
    return NULL;
}
```

Ogni thread, una volta avviato, acquisisce il lock sul mutex e stampa un messaggio indicando che è in attesa. Successivamente entra in un ciclo while in cui invoca la funzione `pthread_cond_wait()`. Come già visto nell'esercizio precedente, questa chiamata ha l'effetto di sospendere il thread, rilasciare il mutex e inserirlo nella coda di attesa della condition variable. Solo quando la condizione `cond` verrà soddisfatta e i thread saranno risvegliati, potranno riprendere l'esecuzione.

3.2.3 Segnalazione e sblocco dei thread

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

    pthread_mutex_lock(&lock);
    cond = 1;
    printf("Condizione soddisfatta, sveglio i thread...\n");
    pthread_cond_broadcast(&queue);
    pthread_mutex_unlock(&lock);
```

Dopo aver avviato tutti i thread, il main thread acquisisce il lock rilasciato dalla wait, aggiorna la condizione logica portandola a 1, e successivamente chiama la funzione `pthread_cond_broadcast()`. A differenza della funzione `pthread_cond_signal()`, che sveglierebbe un solo thread in attesa, `pthread_cond_broadcast()` risveglia **tutti** i thread presenti nella coda associata alla condition variable. In questo modo, tutti i quattro thread si sbloccano contemporaneamente e proseguono la loro esecuzione.

3.2.4 Attesa della terminazione dei thread

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

Dopo aver inviato il segnale di risveglio, il thread principale attende che tutti i thread creati completino la loro esecuzione.

3.2.5 Output del programma

```
Thread 0 in attesa...
Thread 1 in attesa...
Thread 2 in attesa...
Thread 3 in attesa...
Condizione soddisfatta, sveglio i thread...
Thread 0 svegliato!
Thread 0 terminato!
Thread 2 svegliato!
Thread 2 terminato!
Thread 3 svegliato!
Thread 3 terminato!
Thread 1 svegliato!
Thread 1 terminato!
```

Come possiamo osservare dall'output, tutti i thread vengono inizialmente messi in attesa nella coda. Quando il main thread soddisfa la condizione e invia il broadcast, i thread si risvegliano quasi simultaneamente, stampano il messaggio di ripartenza e terminano correttamente. Un'altra osservazione importante da fare è che i thread non vengono risvegliati in ordine come un coda vera e propria, questo succede perché dalla struttura dati `pthread_cond_t` vengono estratti casualmente.

4 Conclusioni

In questa esercitazione abbiamo introdotto e utilizzato le **condition variables** come meccanismo di sincronizzazione avanzato tra thread. A differenza di mutex e semafori, le condition variables permettono ai thread di sospendere volontariamente l'esecuzione in attesa che si verifichi una determinata condizione.

Nel primo esercizio abbiamo simulato il comportamento della funzione ormai ricorrente `pthread_join()`, mostrando come un thread padre possa attendere la terminazione del thread figlio senza ricorrere direttamente alle primitive offerte dalla libreria POSIX, ma gestendo manualmente l'attesa tramite condi-

tion variable e mutex.

Nel secondo esercizio abbiamo implementato una **barriera di sincronizzazione**, utilizzando `pthread_cond_broadcast()` per risvegliare simultaneamente tutti i thread che erano in attesa di una condizione comune.

Attraverso questi esempi abbiamo visto la potenza e l'utilità delle condition variables per risolvere problemi di sincronizzazione più complessi e per gestire comunicazioni più sofisticate tra thread concorrenti. Naturalmente è possibile gestire problemi più sofisticati attraverso l'utilizzo di condizione più complesse.