Università di Messina — CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

Hands-On #14

Sincronizzazione tra thread: mutex, timedlock e rwlock

Last Update: 6 maggio 2026

Scenario

In questo HO useremo le primitive POSIX di sincronizzazione tra thread per risolvere tre problemi classici di accesso concorrente a risorse condivise. Gli strumenti sono `pthread_mutex_t`, `pthread_mutex_timedlock()` e `pthread_rwlock_t` — costruiti sopra i futex del kernel Linux che abbiamo analizzato a lezione.

Tutti gli esercizi si compilano con:

```
gcc -Wall -Wextra -pthread -o nome nome.c
```

[!] Flag `-pthread` obbligatorio

Senza `-pthread` il linker non include la libreria `libpthread` e i simboli `pthread_*` non vengono risolti. L'errore tipico è `undefined reference to 'pthread_create'`.

Esercizio 1 — Conto corrente condiviso

Il problema

Una banca ha N sportelli che operano in parallelo sullo stesso conto corrente. Ogni sportello esegue in loop operazioni di deposito e prelievo: il prelievo è condizionato al saldo disponibile, che non deve mai scendere sotto zero.

Il codice di partenza è volutamente **non protetto**: il saldo può diventare negativo e il risultato cambia ad ogni esecuzione. Osserva il comportamento prima di aggiungere qualsiasi lock.

Listing 1: banca.c — codice di partenza (con race condition)

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define N_SPORTELLI  4
#define N_OPERAZIONI 100000

int saldo = 100000;  /* saldo iniziale in euro */

void *sportello(void *arg) {
    int id = *(int *)arg;
    for (int i = 0; i < N_OPERAZIONI; i++) {
        int importo = (rand() % 100) + 1;
        if (i % 2 == 0) {
            saldo += importo;              /* deposito  */
        } else {
            if (saldo >= importo)          /* prelievo  */
                saldo -= importo;
        }
```

```c
    }
    return NULL;
}

int main(void) {
    pthread_t threads[N_SPORTELLI];
    int ids[N_SPORTELLI];
    for (int i = 0; i < N_SPORTELLI; i++) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, sportello, &ids[i]);
    }
    for (int i = 0; i < N_SPORTELLI; i++)
        pthread_join(threads[i], NULL);
    printf("Saldo finale: %d euro\n", saldo);
    return 0;
}
```

Cosa fare

1. Compila ed esegui il codice più volte. Osserva che il saldo finale è diverso ad ogni esecuzione e può essere negativo. Spiega perché il controllo `if (saldo >= importo)` non è sufficiente a garantire la correttezza.
2. Dichiara un `pthread_mutex_t` globale e inizializzalo con `PTHREAD_MUTEX_INITIALIZER`.
3. Proteggi ogni accesso a `saldo` — sia lettura che scrittura — con `pthread_mutex_lock()` e `pthread_mutex_unlock()`. Attenzione: la sezione critica deve coprire sia il controllo `if` che la modifica, altrimenti la race condition persiste.
4. Aggiungi la chiamata a `pthread_mutex_destroy()` prima del `return` nel `main()`.
5. Verifica che il saldo finale sia sempre non negativo e identico tra esecuzioni diverse (a parità di seme di `rand()`).

▷ **La sezione critica giusta**

Non basta proteggere solo `saldo -= importo`. Se due thread leggono `saldo >= importo` entrambi veri nello stesso istante, entrambi eseguiranno il prelievo — e il saldo diventerà negativo. L'intera sequenza *leggi / controlla / modifica* deve essere atomica rispetto agli altri thread.

Esercizio 2 — Stampante condivisa con timeout

Il problema

Un ufficio ha una sola stampante condivisa tra N utenti. Ogni utente tenta di acquisire la stampante: se non si libera entro 3 secondi, rinuncia e logga il timeout. Una stampa richiede 2 secondi. Gli utenti arrivano uno ogni 0.2 secondi.

Listing 2: stampante.c — codice di partenza

```c
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

#define N_UTENTI 6

pthread_mutex_t stampante = PTHREAD_MUTEX_INITIALIZER;
```

```c

void *utente(void *arg) {
    int id = *(int *)arg;

    /* Costruisci il timeout: ora corrente + 3 secondi */
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += 3;

    /* TODO: tentare di acquisire la stampante con timeout      */
    /* Se acquisita: simulare una stampa (sleep 2s), poi unlock */
    /* Se timeout:   stampare un messaggio di rinuncia           */

    printf("Utente %d: ???\n", id);
    return NULL;
}

int main(void) {
    pthread_t threads[N_UTENTI];
    int ids[N_UTENTI];
    for (int i = 0; i < N_UTENTI; i++) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, utente, &ids[i]);
        usleep(200000);              /* un utente ogni 0.2 s */
    }
    for (int i = 0; i < N_UTENTI; i++)
        pthread_join(threads[i], NULL);
    pthread_mutex_destroy(&stampante);
    return 0;
}
```

Cosa fare

1. Completa la funzione `utente()` usando `pthread_mutex_timedlock(&stampante, &ts)`. Controlla il valore di ritorno: `0` significa lock acquisito, `ETIMEDOUT` significa scaduto.
2. Se il lock è acquisito, simula la stampa con `sleep(2)` e poi rilascia con `pthread_mutex_unlock()`. Stampa un messaggio all'inizio e alla fine della stampa.
3. Se scade il timeout, stampa un messaggio di rinuncia (non chiamare `unlock` in questo caso).
4. Esegui il programma e conta quanti utenti riescono a stampare e quanti rinunciano. Prima di guardare l'output, ragiona: con stampa da 2 s, timeout da 3 s, e un utente ogni 0.2 s, quale utente è il primo a rinunciare?
5. Modifica la durata della stampa a 4 secondi e osserva come cambia il comportamento. Quanti utenti rinunciano ora?

▷ `CLOCK_REALTIME` e `CLOCK_MONOTONIC`

`pthread_mutex_timedlock()` richiede un tempo assoluto basato su `CLOCK_REALTIME` (l'orologio di sistema). Non usare `CLOCK_MONOTONIC`: la funzione non lo accetta e restituirebbe `EINVAL`. Il tempo assoluto si ottiene leggendo l'ora corrente con `clock_gettime()` e sommando i secondi di attesa desiderati.

[!] Non chiamare `unlock` senza aver acquisito

Se `pthread_mutex_timedlock()` restituisce `ETIMEDOUT`, il mutex **non è stato acquisito**: chiamare `unlock()` in questo caso produce comportamento non definito. Usa

sempre una variabile per tracciare se il lock è stato ottenuto.

Esercizio 3 — Rubrica telefonica con rwlock

Il problema

Una rubrica telefonica è condivisa tra molti thread lettori — che fanno ricerche frequenti e veloci — e un singolo thread scrittore, che aggiorna i contatti raramente ma richiede accesso esclusivo. Un `pthread_mutex_t` ordinario funzionerebbe, ma serializzerebbe anche le letture tra loro: inutilmente, perché leggere in parallelo è sempre sicuro. Il `pthread_rwlock_t` risolve esattamente questo problema.

Listing 3: rubrica.c — codice di partenza

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define N_LETTORI 8

typedef struct { char nome[32]; char numero[16]; } Contatto;

Contatto rubrica[] = {
    {"Alice", "333-1111"},
    {"Bob",   "333-2222"},
    {"Carlo", "333-3333"},
};
int n_contatti = 3;

/* TODO: dichiarare pthread_rwlock_t rubrica_lock */

void *lettore(void *arg) {
    int id = *(int *)arg;
    for (int i = 0; i < 5; i++) {
        /* TODO: acquisire il lock in lettura */
        int idx = i % n_contatti;
        printf("Lettore %d: %s -> %s\n",
               id, rubrica[idx].nome, rubrica[idx].numero);
        usleep(100000);
        /* TODO: rilasciare il lock */
    }
    return NULL;
}

void *scrittore(void *arg) {
    sleep(1);  /* lascia partire i lettori */
    /* TODO: acquisire il lock in scrittura */
    printf("Scrittore: aggiorno il numero di Bob...\n");
    strcpy(rubrica[1].numero, "333-9999");
    sleep(2);  /* simula aggiornamento lento */
    printf("Scrittore: aggiornamento completato.\n");
    /* TODO: rilasciare il lock */
    return NULL;
}

int main(void) {
    /* TODO: inizializzare rubrica_lock */
    pthread_t lettori[N_LETTORI], scr;
    int ids[N_LETTORI];
    pthread_create(&scr, NULL, scrittore, NULL);
```

```
48      for (int i = 0; i < N_LETTORI; i++) {
49          ids[i] = i;
50          pthread_create(&lettori[i], NULL, lettore, &ids[i]);
51      }
52      pthread_join(scr, NULL);
53      for (int i = 0; i < N_LETTORI; i++)
54          pthread_join(lettori[i], NULL);
55      /* TODO: distruggere rubrica_lock */
56      return 0;
57  }
```

Cosa fare

1. Dichiara un `pthread_rwlock_t rubrica_lock` globale e inizializzalo nel `main()` con `pthread_rwlock_init(&rubrica_lock, NULL)`.
2. Nella funzione `lettore()`, acquisisci il lock con `pthread_rwlock_rdlock()` prima della lettura e rilascialo con `pthread_rwlock_unlock()` dopo.
3. Nella funzione `scrittore()`, acquisisci il lock con `pthread_rwlock_wrlock()` e rilascialo con `pthread_rwlock_unlock()` dopo l'aggiornamento.
4. Distruggi il lock nel `main()` con `pthread_rwlock_destroy()`.
5. Verifica nell'output che i lettori procedano in parallelo tra loro (i loro messaggi si interleaving) ma si blocchino tutti durante i 2 secondi di aggiornamento dello scrittore.
6. Verifica che nessun lettore legga il numero vecchio di Bob (`333-2222`) *dopo* che lo scrittore ha stampato "aggiornamento completato".

▷ **Lettori e scrittori non si escludono tra loro**

`pthread_rwlock_rdlock()` non blocca altri lettori: più thread possono tenere il read lock simultaneamente. Blocca solo se è in corso o in attesa un write lock. `pthread_rwlock_wrlock()` invece aspetta che *tutti* i lettori correnti abbiano rilasciato il lock prima di procedere.

▷ **Come riconoscere il parallelismo nell'output**

Nell'output corretto vedrai i messaggi dei lettori mischiati tra loro (interleaving), poi un blocco silenzioso di 2 secondi durante la scrittura, poi la ripresa dei lettori con il nuovo numero di Bob. Se i messaggi dei lettori appaiono tutti in sequenza ordinata per id, probabilmente stai usando un mutex ordinario invece di un rwlock.