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

Hands-On #13

Thread POSIX in C: creazione, sincronizzazione e pattern

Last Update: 4 maggio 2026

Scenario

In questo HO useremo l'API POSIX dei thread (`pthreads`) per costruire programmi C concorrenti, partendo dalle primitive di base fino ai pattern architetturali usati in produzione. Ogni esercizio è autonomo e introduce uno o più concetti nuovi; gli esercizi sono in ordine crescente di difficoltà.

Il programma si compila sempre con il flag `-pthread`:

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

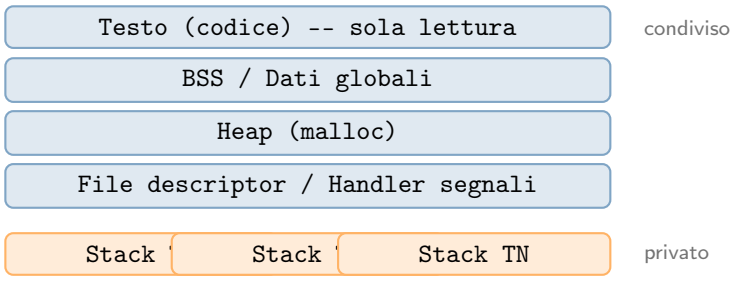
[!] Compilare senza `-pthread`

Omettere `-pthread` non produce un errore di compilazione ma genera un eseguibile silenziosamente difettoso: le funzioni `pthread_*` potrebbero risolvere versioni stub non thread-safe della libreria C. Usare *sempre* `-pthread`.

Mappa concettuale

Il modello di memoria dei thread

Tutti i thread di un processo condividono lo stesso spazio di indirizzamento: heap, variabili globali, file descriptor aperti, handler dei segnali. Ogni thread possiede invece il proprio **stack**, i propri registri CPU, la propria maschera dei segnali e il proprio `errno`.



Il ciclo di vita di un thread

Un thread viene creato con `pthread_create()`, esegue la funzione associata e termina in uno di tre modi: `return` dalla funzione thread, chiamata esplicita a `pthread_exit()`, oppure cancellazione da parte di un altro thread con `pthread_cancel()`. Il thread rimane nello stato *zombie* (le risorse non vengono liberate) finché un altro thread non ne raccoglie il valore di ritorno con `pthread_join()`, oppure finché non viene marcato come *detached* con `pthread_detach()`.

[!] Chiamare `exit()` da un thread

`exit()` termina l'intero processo, non solo il thread corrente. Per uscire dal solo thread corrente usare `pthread_exit()`.

Esercizio 1 — Somma parallela di un array

Obiettivo

Dividere un array di N interi in T segmenti uguali, calcolare la somma parziale di ciascun segmento in un thread separato, raccogliere i risultati con `pthread_join()` e verificare che la somma totale sia corretta.

Questo esercizio consolida: `pthread_create()`, `pthread_join()`, passaggio di strutture come argomento via `void *`.

Specifica

Scrivere il programma `sum.c` che:

1. Alloca un array di `N = 1 000 000` interi inizializzati con `a[i] = i % 100`.
2. Lancia `T = 4` thread; il thread k calcola la somma degli elementi nell'intervallo $[k \cdot N/T,\ (k{+}1) \cdot N/T)$.
3. Raccoglie i risultati con `pthread_join()` e stampa la somma totale.
4. Verifica il risultato calcolando la stessa somma in modo sequenziale e confrontando i due valori.

Struttura suggerita

La funzione thread riceve un puntatore a una struttura che descrive il suo segmento di lavoro e dove scrivere il risultato parziale:

Listing 1: Struttura argomento e funzione thread per sum.c

```c
typedef struct {
    const int *arr;    /* puntatore all'array condiviso (sola lettura) */
    int        from;   /* indice iniziale (incluso)                    */
    int        to;     /* indice finale   (escluso)                    */
    long long  result;/* risultato parziale: scritto dal thread        */
} WorkArg;

static void *partial_sum(void *arg) {
    WorkArg *w = (WorkArg *)arg;
    long long s = 0;
    for (int i = w->from; i < w->to; i++)
        s += w->arr[i];
    w->result = s;
    return NULL; /* join riceve NULL; il risultato e' gia' in w->result */
}
```

▷ Suggerimento

Il campo `result` viene scritto dal thread e letto dal `main` solo *dopo* `pthread_join()`. Il join garantisce che la scrittura del thread sia visibile al main: non serve nessun altro meccanismo di sincronizzazione in questo esercizio.

Output atteso

```
Somma parallela  (T=4):  49500000
Somma sequenziale:       49500000
[OK] i risultati coincidono
```

Esercizio 2 — Barriera: map-reduce parallelo

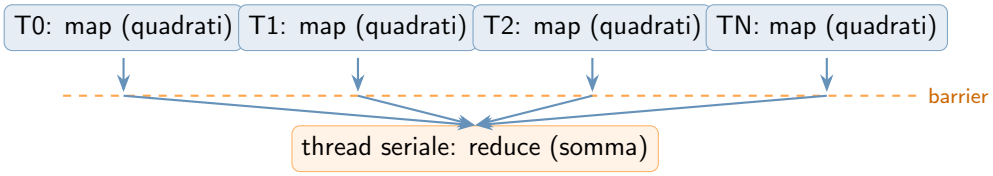
Obiettivo

Implementare uno schema map-reduce in due fasi usando `pthread_barrier_t`: N thread eseguono una fase di map in parallelo, si sincronizzano alla barriera, poi il thread "seriale" esegue il reduce. Questo schema è la base di molti framework di calcolo parallelo.

Questo esercizio consolida: `pthread_barrier_init()`, `pthread_barrier_wait()`, `PTHREAD_BARRIER_SERIAL_THREAD` e la progettazione di pipeline parallele.

Scenario

Dato un array di N interi, ogni thread calcola il quadrato degli elementi nel proprio segmento (fase map). Quando tutti hanno finito, il thread designato — il solo che riceve `PTHREAD_BARRIER_SERIAL_THREAD` — somma tutti i quadrati (fase reduce) e stampa il risultato.



Specifica

Scrivere il programma `mapreduce.c` che:

1. Alloca un array `a[N]` con `N = 800 000` interi `a[i] = i % 1000`.
2. Lancia `T = 4` thread; il thread k calcola `a[i] = a[i] * a[i]` per il proprio segmento (fase map).
3. Tutti i thread si sincronizzano su una `pthread_barrier_t` inizializzata per T thread.
4. Il thread che riceve `PTHREAD_BARRIER_SERIAL_THREAD` esegue la fase di reduce: somma tutti gli elementi e stampa il risultato. Gli altri thread terminano senza fare nulla dopo la barriera.
5. Il main verifica il risultato calcolando la stessa somma di quadrati in modo sequenziale.

Listing 2: Schema della barriera in mapreduce.c

```c
static pthread_barrier_t barrier;

static void *thread_fn(void *arg) {
    WorkArg *w = (WorkArg *)arg;

    /* --- Fase MAP: ogni thread lavora sul proprio segmento --- */
    for (int i = w->from; i < w->to; i++)
        w->arr[i] = w->arr[i] * w->arr[i];

    /* --- Barriera: aspetta che tutti i thread abbiano finito il map ---
       */
    int rc = pthread_barrier_wait(&barrier);

    /* --- Fase REDUCE: solo il thread "seriale" esegue il reduce --- */
    if (rc == PTHREAD_BARRIER_SERIAL_THREAD) {
```

```
15          long long sum = 0;
16          for (int i = 0; i < w->total; i++)
17              sum += w->arr[i];
18          printf("Somma dei quadrati (parallelo):  %lld\n", sum);
19      }
20      /* Gli altri thread hanno rc == 0 e terminano qui */
21      return NULL;
22  }
```

▷ Chi esegue PTHREAD_BARRIER_SERIAL_THREAD?

Lo standard POSIX garantisce che *esattamente uno* tra i thread che raggiungono la barriera riceva il valore `PTHREAD_BARRIER_SERIAL_THREAD`; gli altri ricevono 0. Non è garantito quale thread sia: può variare tra esecuzioni. È l'unico meccanismo di selezione "uno tra N" offerto direttamente da pthreads senza mutex.

Output atteso

```
Somma dei quadrati (parallelo):   332833500000
Somma dei quadrati (sequenziale): 332833500000
[OK] i risultati coincidono
```

Variante opzionale

Modificare il programma per usare *due* fasi di map separate da due barriere: nella prima fase calcolare i quadrati, nella seconda fase calcolare la radice intera dei quadrati (ottenendo i valori originali). Verificare che alla fine dell'array i valori siano identici a quelli iniziali.

Compilazione e test riepilogativo

```
# Compilare tutti gli esercizi
gcc -Wall -Wextra -pthread -o sum        sum.c
gcc -Wall -Wextra -pthread -O1 -o tls    tls.c
gcc -Wall -Wextra -pthread -o once       once.c
gcc -Wall -Wextra -pthread -o cancel     cancel.c
gcc -Wall -Wextra -pthread -o mapreduce  mapreduce.c

# Verificare la TLS con objdump (esercizio 2)
gcc -O1 -pthread -c tls.c -o tls.o
objdump -d tls.o | grep -A 15 "<increment_both>"

# Verificare le librerie collegate (si vede libpthread o libc)
objdump -p ./sum | grep NEEDED

# Eseguire con valgrind per rilevare memory leak o use-after-free
valgrind --leak-check=full ./cancel
```

▷ Thread sanitizer

GCC e Clang offrono `-fsanitize=thread`: compila il programma con strumentazione che rileva race condition a runtime, senza modificare il sorgente. È molto utile già da ora, prima di introdurre i mutex:

```
gcc -Wall -pthread -fsanitize=thread -o tls_san tls.c
./tls_san
```

L'esercizio 2 con `-fsanitize=thread` mostrerà un avviso di race su `shared_counter`: è esattamente ciò che ci aspettiamo, e anticipa il tema della prossima lezione.