Esercitazione: Thread

Tutorato di Sistemi Operativi

Giugno - Luglio 2025

4.a Somma di vettori in parallelo

Obiettivo Dato $N > 1$, lunghezza dei vettori A e B, calcolare il vettore somma $C = A + B$ impiegando più thread e confrontare le prestazioni con l'implementazione sequenziale.

Specifiche di base

1. Il programma accetti da riga di comando: `N` (lunghezza) e `T` (numero di thread da creare). Se `T`=1 si ottiene automaticamente la versione sequenziale.
2. Allochi i tre vettori su *heap* e li inizializzi con valori pseudocasuali in virgola mobile (double).
3. Suddivida il range $[0, N)$ in porzioni *contigue* di ampiezza $\lfloor N/T \rfloor$ e assegni a ciascun thread una sola porzione. (gestendo eventuali elementi rimanenti dal resto della divisione)
4. Ogni thread calcoli i propri elementi di C senza usare dati globali condivisi, salvo i puntatori ai tre vettori.
5. Il *main* misuri:
   - tempo di allocazione e inizializzazione;
   - tempo puro di somma (da `pthread_create` a `pthread_join`);
   - speed-up $S = \dfrac{t_{\text{seq}}}{t_T}$.

Output minimo richiesto Una riga tipo:

```
N=100000000  T=8  sum_time=0.166 s  speedup=1.34
```

4.b Staffetta cronometrata con barriera

Obiettivo Simulare una gara di *thread–corridori* e usare due barriere per coordinare la partenza sincronizzata e l'arrivo collettivo. Il thread eletto dalla seconda barriera elabora i risultati e mostra una classifica ordinata.

Specifiche di base

1. Il programma accetti da riga di comando `N` (# di corridori) e `max_delay` (massimo tempo di "corsa" in millisecondi).
2. All'avvio il `main`:
   2.1. allochi un array di `struct` con *id* e tempi `double`,
   2.2. crei due barriere inizializzate a N,
   2.3. generi N thread, passando un puntatore alla propria entry.
3. Ogni thread *corridore* esegua:
   3.1. `pthread_barrier_wait` sulla barriera di *start*;
   3.2. salvi `t_start` con `clock_gettime(CLOCK_MONOTONIC)`;
   3.3. dorma un intervallo casuale uniforme in $[0, \texttt{max\_delay}]$;
   3.4. salvi `t_stop` e calcoli $\Delta t = t_{\text{stop}} - t_{\text{start}}$;
   3.5. registri Δt nella propria cella dell'array condiviso;
   3.6. `pthread_barrier_wait` sulla barriera di *finish*.
4. Se la seconda `pthread_barrier_wait` restituisce `PTHREAD_BARRIER_SERIAL_THREAD`, quel thread diventa "cronometrista" e deve:
   4.1. ordinare l'array per tempo crescente,
   4.2. stampare la classifica completa (posizione, id, tempo ms),
   4.3. calcolare media, minimo e massimo dei tempi.
5. Tutti i thread si riuniscono con `pthread_join`; qualsiasi memoria o barriera va distrutta prima dell'uscita.

Output minimo atteso `N=8 max_delay=4000 ms`

```
Pos Corridore  Δt(ms)
  1  5  137.4
  2  2  841.7
     ...
  8  6  3932.1
min = 137.4  max = 3932.1  mean = 2198.6
```

4.c Logger concorrente con coda circolare

Obiettivo Realizzare un piccolo sottosistema di logging basato sul pattern *Produttore/Consumatore* con buffer circolare e sincronizzazione mediante `mutex` e `condition variable` (nessun busy–wait).

Specifiche di base

1. Il programma accetti da riga di comando: `N` (numero di messaggi da produrre) e `K` (capacità del buffer).
2. *Produttore* Un singolo thread genera `N` stringhe di massimo 128 byte, numerate in ordine crescente, e le inserisce nel buffer circolare. Ogni stringa può avere il formato:

   ```
   [seq=42] Hello, world!\n
   ```

3. *Consumatori* Due thread distinti prelevano dalla stessa coda:
   - **C1** scrive i messaggi dentro `debug.log`;
   - **C2** li invia su `stdout` (facoltativamente colorando la riga con escape ANSI).
4. **Sincronizzazione** Usare un unico `pthread_mutex_t` e due `pthread_cond_t`: `not_full` (produttore in attesa) e `not_empty` (consumatori in attesa).
5. **Terminazione ordinata** Quando il produttore ha inserito l'ultimo messaggio, imposta un flag `done=1`, trasmette `broadcast` su `not_empty` e termina. I consumatori escono dal ciclo se `done==1` *e* il buffer è vuoto. Il `main` effettua i tre `pthread_join` e poi chiude il file di log.
6. **Statistiche finali** Alla fine stampare: numero di messaggi processati dai consumatori, riempimento medio del buffer (può essere stimato con un contatore aggiornato ogni operazione) e durata totale del logging (`clock_gettime`).

Estensioni facoltative

- Rendere parametrico il numero di consumatori (`M`) e misurare la *scalabilità* al crescere di `M`.
- Introdurre messaggi con *priorità* (INFO, WARN, ERR) e gestire una policy di scarto quando il buffer è pieno (es. *drop tail*).
- Implementare la versione con due `sem_t` POSIX (*counting semaphore*) invece delle condition variable.

4.d Thread-pool — Conteggio parole in parallelo

Obiettivo Implementare una versione concorrente del classico comando `wc -w` usando un pool fisso di thread che processano file diversi in parallelo.

Traccia essenziale

1. Da riga di comando: `P` (dimensione del pool). L'elenco dei percorsi dei file arriva su `stdin` – uno per riga – fino a `EOF`.
2. Il `main`:
   2.1. crea una coda protetta da `mutex + condition`;
   2.2. inserisce in coda ogni percorso letto;
   2.3. avvia P thread lavoratori;
   2.4. al termine dell'input marca la coda come "chiusa" e fa `broadcast` per svegliare eventuali worker in attesa.
3. Ogni *worker* in ciclo:
   3.1. preleva un percorso (bloccante se la coda è vuota);
   3.2. apre il file in sola lettura, conta le parole (basta scorrere con `fgetc`);
   3.3. deposita il conteggio in una struttura condivisa usando un `mutex` dedicato o un'operazione atomica (`__sync_fetch_and_add`);
   3.4. termina quando la coda risulta vuota e "chiusa".
4. Il `main` aspetta i worker con `pthread_join` e infine stampa, per ogni file, il numero di parole e la somma totale.

4.e Statistiche *live* su più liste di numeri

Il programma deve:

- creare N liste, ognuna lunga M numeri;
- avviare N "aiutanti" (uno per lista) che sorvegliano la propria lista;
- fare in modo che la parte principale del programma, ogni `T` millisecondi, cambi a caso *un solo* numero in *ogni* lista;
- quando un aiutante si accorge che la sua lista è cambiata, ricalcola subito e stampa:
  - la media dei numeri,
  - la mediana (il valore centrale dopo averli ordinati),
  - la variazione interna (quanto si discostano in media dal centro),
  - il valore minimo,
  - il valore massimo.

Come lanciare il programma

```
./stats_live  N  M  T
```

`N` quante liste / quanti aiutanti (predefinito 4)
`M` quanti numeri per lista (predefinito 1000)
`T` tempo in millisecondi fra due modifiche (predefinito 500)

Flusso di lavoro (in breve)

1. La parte principale del programma parte, prepara le liste e fa partire gli aiutanti.
2. Entra in un giro infinito: ogni `T` millisecondi sceglie un numero di elementi in maniera pseudocasuale in una o più liste e lo rimpiazza con un nuovo numero generato al momento.
3. Ogni aiutante, quando nota che la sua lista è stata toccata, effettua un ricalcolo e mostra il risultato.