Esercitazione 3

10/12 maggio 2023

Esercizio 1

Un noto rompicapo prevede di trovare la sequenza di operazioni elementari (somma, sottrazione, moltiplicazione e divisione) necessarie per ottenere 10, partendo da 5 numeri da 0 a 9 casuali. I vincoli sono:

- le cinque cifre sono comprese tra 0 e 9 e possono ripetersi
- le cifre devono essere utilizzate tutte, in qualsiasi ordine
- non ci sono limiti sulle operazioni (es. vanno bene anche quattro somme)
- non si considera la precedenza degli operatori, le operazioni vanno applicate da sinistra a destra secondo il loro ordine

Esempio: dato 2 7 2 2 1 una soluzione può essere 7 - 2 - 1 x 2 + 2 = 10

Scrivere un programma che, letta la sequenza di cifre da command line come argomento, trovi tutte le possibili soluzioni, se ve ne sono, le salvi in una collezione di stringhe (es: “7 - 2 - 1 x 2 + 2”) e la stampi. Nella collezione le stringhe devono essere uniche.
Per risolverlo utilizzare un approccio *brute force*, ovvero elencare in un vettore tutte le possibili permutazioni delle cinque cifre e, assieme ogni permutazione di cifre, tutte le possibili permutazioni di quattro operazioni elementari, provando a calcolare il risultato per ciascuna di esse. Se il risultato è 10 la permutazione viene salvata, altrimenti viene scartata. Essendo cifre e simboli delle operazioni di tipo differente utilizzare un vettore di tuple: il primo elemento sono le 5 cifre permutate, il secondo le operazioni.

Una volta verificato il corretto funzionamento sfruttiamo i thread per velocizzare la ricerca delle soluzioni in parallelo. Si divide il vettore di tutte le possibili permutazioni in n blocchi uguali e per ciascuna si lancia un thread. Provare con n=2,3,4… ecc, misurare i tempi e trovare il numero di thread oltre il quale non vi sono vantaggi.
Cambia qualcosa se la divisione del lavoro fra thread anziché essere a blocchi è *interleaved*? Vale a dire con tre thread il primo prova le permutazioni con indice 0,3,6,... il secondo 1,4,7,... e il terzo 2,5,8,...

Esercizio 2

Adattare l’esercizio sul buffer circolare dell’esercitazione precedente facendo sì che:

- vi sia un solo processo
- producer e consumer vengono eseguiti in due thread differenti
- il buffer non sia più su file, ma una struttura in memoria condivisa tra i due thread

Le altre logiche di funzionamento rimangono inalterate, quindi il producer scrive un valore al secondo, il consumer legge i valori ogni 10.

Attenzione che anche in questo caso l'accesso al buffer e agli indici di scrittura e lettura va sincronizzato, garantendo la mutua esclusione fra i due thread.
Il buffer quindi dovrà essere una struct RingBuf con due metodi, read() -> Option<T> e write(val) -> Result<(),()> che siano *thread safe*, quindi invocabili all'interno di thread senza che il chiamante debba preoccuparsi delle operazioni di sincronizzazione.

read restituisce None quando il buffer è vuoto. mentre la write restituisce Err(()) quando il buffer è pieno e Ok(()) quando la scrittura è avvenuta con successo.

Cercare inoltre di implementare il buffer in modo generico, senza che sia specifico per SensorData.

Bonus: modificare producer e consumer in modo che scrivano e leggano sul buffer senza pause (se non quando è pieno o vuoto) e misurare il throughput del buffer con dimensioni differenti (10, 1000, 10000 valori). Misurare anche il throught aumentando il numero di producer e consumer, come varia?

Esercizio 3

Nell'esercizio sul file system dell'esercitazione precedente non si poteva restituire oggetti MatchResult contenenti directory, in quanto la soluzione avrebbe richiesto di ottenere più di un riferimento mutabile alla stessa directory e, quindi, ci eravamo limitati a cercare i file.
Per aggiungere una directory nel risultato infatti si sarebbe dovuto salvare un suo riferimento mutabile, ma, per poter proseguire la ricerca nei suoi figli, occorreva mantenere sempre un riferimento mutabile alla directory stessa, cosa che il compilatore di Rust impedisce.

Per gestire situazioni analoghe Rust mette a disposizione gli smart pointer Cell e RefCell che implementano il pattern di interior mutability, vale a dire che Cell<T> non è mutabile, ma è possibile provare ad ottenere un riferimento mutabile a T a runtime. Solo nel momento in cui viene chiesto il riferimento viene fatto un check e la richiesta fallisce se non è possibile.

Modificare l'esercizio inserendo i nodi (file e directory) in Cell o RefCell e modificare la ricerca permettendo di restituire anche directory nei risultati.