Esercitazione 5

7 / 9 giugno 2023

Esercizio 1

Un sistema di indicizzazione dei file di testo sul pc è composto dalle seguenti parti:

- un processo padre che cerca i file da indicizzare (es. tutti I file .txt), scandendo in profondità una directory passata come argomento della linea di comando all'avvio
- un pool di processi figli che analizza i file i cui cammini sono stati indicati sul proprio standard input, estrae le parole contenute al loro interno, costruisce una mappa con il conteggio di ciascuna parola presente e trasmette la mappa del proprio standard output

Il processo padre è responsabile di far partire i processi figli, distribuire fra di loro il lavoro e raccogliere I risultati:

- dovrà gestire il ciclo di vita dei figli e creare un canale di comunicazione bidirezionale per ciascun figlio utilizzando delle pipe
- un thread cerca i file da indicizzare e, una volta individuati, ne passa il path a un figlio attraverso la pipe
- un thread legge I risultati del conteggio dalla pipe in uscita dal medesimo figlio (attenzione al deadlock: leggere solo da processi figli che hanno del lavoro da completare)
- quando non ci sono più file da analizzare salva l'indicizzazione su un file, scrivendo per ogni parola una linea con:
    - il numero totale di occorrenze
    - i file in cui compare e le occorrenze nei file separate da spazi

Definire il protocollo di serializzazione opportuno per la comunicazione tra padre e figli.

Un esempio di corpus che si può utilizzare per i test.

Osservazioni
L'architettura basata su un pool di processi separati sembra essere un overhead inutile rispetto alla stessa basata su thread; infatti ha in più il costo di serializzazione dei messaggi per comunicare fra i processi. Tuttavia è molto usata quando I dati che il processo deve gestire sono poco affidabili e c'è la possibilità che emergano errori imprevisti. Se un thread va in panic, le sue risorse possono non essere più recuperabili, mentre se un processo muore basta ricrearne uno nuovo recuperando ogni risorsa. Un esempio sono i browser moderni: il rendering di ciascuna pagina viene eseguito da un processo dedicato.

La lettura dei dati dai figli con un unico thread, come viene richiesto, non è ottimale in quanto è possibile che, mentre un figlio ha finito il proprio lavoro, il padre sia bloccato in attesa del risultato di un altro figlio. In applicazioni reali, per evitare di far partire un thread per figlio, guardare la libreria Tokio, che implementa un runtime asincrono di Rust: attraverso le coroutine è possibile sospendere l'esecuzione di una funzione, in attesa che vi siano dati pronti da leggere su un file descriptor o un socket, senza dover far partire n thread.

In situazioni dove il throughput deve essere massimizzato, anche a costo di usare molta cpu, si può addirittura preferire qualche forma di busy waiting rispetto alla sincronizzazione. Chi fosse interessato piò leggere questo articolo che spiega come ottimizzare la velocità delle pipe Linux.