Esercitazione 6

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 sul 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 ai deadlock: leggere solo da processi figlio 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
https://www.corpusdata.org/iweb_samples.asp

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, la 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 si verifichi 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 fare 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 può leggersi questo articolo che spiega come ottimizzare la velocità delle pipe Linux

https://mazzo.li/posts/fast-pipes.html

Esercizio 2

Due processi devono suddividersi del lavoro che leggono da un file condiviso con una lista di task così definita:

- un task per riga
- il task ha un nome (senza spazi) e un tempo di esecuzione

Esempio:

```
task1 1
task1 3
task2 1
task1 3
task2 2
…
```

Ogni processo, appena libero, deve prelevare il task in cima alla lista e simularne l'esecuzione chiamando la funzione run_task("task", ms), che farà una sleep per i millisecondi indicati. Per indicare che il task è stato prelevato il processo dovrà riscrivere il file contenente la lista dei task eliminando la riga con il task prelevato.

Per evitare collisioni fra i due processi e garantire un accesso esclusivo al file con i task, occorre instaurare un meccanismo di sincronizzazione tra i processi. Un approccio è usare la primitiva **flock**, che è bloccante quando non si riesce ad ottenere un accesso esclusivo ad un file. Questo approccio però ha la limitazione di poter funzionare solo per processi che risiedono sullo stesso computer e può non essere portabile su sistemi operativi differenti.
Un meccanismo più generale è usare un protocollo di sincronizzazione tramite scambio di messaggi con un processo che funge da coordinatore (in questo modo il lock è generalizzabile a processi che comunicano fra di loro in rete).
Implementare tale meccanismo di scambio messaggi mediante:

- un processo padre che fa partire i due figli che consumano i task
- prima di provare ad accedere al file, ciascun figlio manda al padre la richiesta req_lock con un messaggio tramite una pipe
- ciascun figlio rimane bloccato in lettura in attesa di ricevere il messaggio lock_granted prima di accedere al file
- finito di accedere al file ciascun figlio manda il messaggio release_lock al padre

- se un figlio dovesse chiudersi inaspettatamente mentre è in possesso del lock, il lock deve venire rilasciato

Osservazioni

Quando tra processi è possibile avere memoria condivisa (file memory mapped), ma non sono presenti primitive di sincronizzazione, per massimizzare il throughput si può implementare un forma di accesso esclusivo basato su busy waiting. Vi sono vari algoritmi in letteratura, per curiosità potete guardare l'algoritmo di Peterson: https://en.wikipedia.org/wiki/Peterson%27s_algorithm