Relazione Progetto Preappello Laboratorio II

Federico Miraglia 620062

Struttura Generale

La struttura del progetto richiede una suddivisone in due processi:

- `MasterWorker`:
  - genera un thread `master` e n thread `worker`;
- `Collector`:
  - agisce da dispatcher aprendo un thread, chiamato `serverWorker`, per ogni richiesta di connessione che accetta.

I thread `worker`, dopo aver elaborato i dati ricevuti, mandano i risultati su una socket.

I thread `serverWorker`, dopo aver letto dalla socket, stampano il risultato in standard output.

Esecuzione

Il processo main si occupa del parsing degli argomenti per poi eseguire una `fork`.

Processo Padre

Un primo thread `master` viene creato e, utilizzando la lista dei file passata per parametro, li controlla uno a uno per poi passarli su una coda ai thread `worker` che vengono aperti subito dopo.
Il Processo Padre aspetta quindi la chiusura dei thread, libera la memoria occupata e esegue l'`unlink` dal file della socket.

Aprire un thread `master` a parte mi ha facilitato nell'implementazione permettendomi una migliore gestione dei segnali e, inoltre, ritengo che sia una suddivisione logica delle task adeguata.

Worker Thread

I thread `workers` preparano una socket su cui richiedere la connessione per inviare i risultati dei file.

Attendono che la connessione venga accettata e procedono con la valutazione del contenuto dei file in arrivo sulla coda.
Inseriscono quindi i valori da inviare in una struttura dati, li mandano sulla socket e ripetono il calcolo per il file successivo.

Quando il thread `worker` legge la stringa di terminazione `<<EOJ` arrivata dal `master`, conclude l'esecuzione riscrivendo la stessa stringa sulla coda per far si che anche gli altri thread, seguendo la stessa logica, terminino.

Processo Figlio

Il Processo Figlio prepara la connessione alla socket e, in qualità di server, si mette in attesa di richieste di connessioni da accettare.
Per ogni connessione che viene accettata, il processo apre un thread `serverWorker` che si occuperà della comunicazione col thread `worker` (client).

ServerWorker

I threads `serverWorker` leggono dalla socket tanti byte quanti ne ha la struttura dati inviata dai `worker` threads e ne stampano il contenuto.
Una volta finito di leggere il contenuto sulla socket, i `serverWorkers` liberano la memoria occupata e terminano l'esecuzione.

Gestione dei Segnali

SIGPIPE

Ho scelto di non bloccare l'esecuzione all'arrivo del segnale `SIGPIPE`.
Lo ignoro nel processo padre e lo blocco nel processo figlio.

SIGHUP, SIGINT, SIGQUIT, SIGTERM

Per rispondere a questi segnali ho creato un handler custom che semplicemente setta una variabile volatile `intGive`, utilizzata come flag, che blocca l'inserimento di nuovi nomi di file nella coda da parte del thread `master`, il quale invece aggiungerà direttamente la stringa di terminazione `<<EOJ`.

Per come ho strutturato il mio progetto, tutti i thread e i processi aperti si chiudono autonomamente una volta giunti al termine del loro lavoro.

Questa soluzione è di facile implementazione e permette uno svolgimento ordinato dei task rimanenti e della fase di chiusura.

Socket

Per la gestione di una connessione multi client ho scelto di allestire il server multi threaded, dove il processo principale si comporta da `dispatcher`.

La connessione socket viene richiesta da ogni thread `worker` nel processo padre.
Quando la connessione viene accettata, il processo figlio crea un thread che si occuperà della comunicazione uno a uno con il thread `worker` richiedente.

Trovo questo un sistema comodo per la gestione di una connessione socket multiclient e per mantenere una struttura pulita e veloce.