UNIVERSITÀ DI PISA

Progetto di Laboratorio di Sistemi Operativi

A.A: 2020/2021

Pardini Luca

Corso B Matricola 583855

INDICE

INTRODUZIONE 2

SERVER 2

- **Segnali** 2
- **Struttura Dati File** 2

WORKER 3

CLIENT 3

- **Api** 3
- **Opzioni aggiuntive** 3

COMUNICAZIONE CLIENT-SERVER 4

LOG 4

MAKEFILE 4

COMANDI PER ESECUZIONE PROGRAMMA 4

CODICE DI TERZE PARTI 5

REPOSITORY GITHUB 5

INTRODUZIONE

Il progetto consiste nella realizzazione di un *server*, realizzato come un processo *multithreaded*, che avrà lo scopo di gestire le richieste di *client* su *file*.

SERVER

Esso è composto da:

- un thread main che si occuperà di inizializzare gli altri *thread* necessari, eseguire le operazioni necessarie per mettersi in ascolto di richieste di connessione da parte di client su di un socket.
- Un thread che gestirà le connessioni dei client.
- Il *thread* per la gestione dei segnali.
- Il *thread* creatore dei workers.
- I vari *thread* workers, che eseguiranno le richieste dei client, come descritto in *worker*

All' avvio verrà letto il file di configurazione contenente i vari parametri che andranno a definire tutti i valori necessari: il numero dei *thread workers*, la dimensione dello spazio di memorizzazione, il nome del *socket* file, il numero massimo di *file* memorizzabili e il nome del *file* di *log*. Successivamente sarà inizializzata la struttura dati che conterrà i *file* con cui interagirà il *server*, spiegata nella sezione dedicata, e la scrittura sul file di log. In seguito, verrà avviato un thread che avrà lo scopo di gestire i segnali (vedi paragrafo ***Segnali*** per ulteriori informazioni).
Dopo aver inizializzato anche i *thread workers*, grazie ad un *thread* che si occuperà unicamente di esso per permettere al *thread main* di occuparsi di altro, il *server* creerà il *socket* per la comunicazione con i *client* e creerà un thread (grazie alla funzione *gestioneConnessioni*) che si occuperà di gestire tutte le connessioni, mettendosi in ascolto su di un socket, e si metterà in attesa grazie ad una *thread_join* che il thread adibito alla gestione dei segnali termini. Il thread delle connessioni accetterà connessioni di nuovi client fino a che non si verificherà una delle due condizioni di arresto (l'arrivo del segnale *SIGHUP* e zero *client* connessi, oppure l'arrivo del segnale *SIGQUIT* o di *SIGINT*); fino a che questo non accade per ogni nuovo *client* che si connetterà con successo, inserirà il relativo *file descriptor* nell' apposita lista (opportunamente sincronizzata con *lock*) in modo tale che poi ogni *thread worker* possa effettuare una dequeue da essa e gestire le richieste del relativo *client*.
Una volta verificatasi una delle due condizioni di uscita, saranno eseguite tutte le apposite operazioni di chiusura del server: attesa di tutti i vari thread che stavano operando, chiusura del socket e le stampe finali delle statistiche del server.
(**N.B.:** tutto il codice per la struttura e le operazioni sulla lista è presente nel file coda.c)

Segnali

I segnali vengono gestiti da un thread che eseguirà la funzione *gestoreSegnali*, utilizzando una maschera che cattura i segnali richiesti per la terminazione del programma, comunicando il relativo indice al *thread main e* grazie ad una variabile (opportunamente sincronizzata grazie ad una lock apposita), per poter gestire in modo adeguato la terminazione.

Struttura Dati File

Tale struttura è un'*array* dinamico di *struct*, dove la dimensione sarà uguale al numero di file memorizzabili letto come parametro. La struttura utilizzata, *info_file*, contiene le informazioni necessarie al salvataggio e alla gestione di un file, ovvero: il *path*, la dimensione, il puntatore al *file* (per operazione di *fopen* e *fclose*), una stringa per contenere i byte veri e propri del file, una condition variable e un lock per ogni singolo file (per realizzare la mutua esclusione su ognuno di essi), il numero di utenti attivi su quel *file*, l'identificatore del *client* che detiene il *lock* su esso ed infine una variabile per identificare se tale *file* è aperto oppure no. Ho deciso di gestire la sincronizzazione di tutti i file tramite un'apposita lock per ognuno di essi, ma anche utilizzando un *lock* relativo a tutta la struttura dati, in modo tale che se si possa interagire con la struttura dati con maggior efficienza. Il *lock* su tutta la struttura dati si può assumere con la funzione *accediStrutturaFile()* e si può rilasciare grazie alla funzione *lasciaStrutturaFile()*.

WORKER

I *thread worker* si occuperanno di ottenere, uno alla volta, i *file descriptor* relativi ai *client* che hanno richiesto una connessione al *server*. Tale procedura avviene grazie ad una lettura da una coda di interi, la quale viene letta in mutua esclusione, con l'utilizzo di una lock dedicata, ottenuta con la funzione *accediCodaComandi()* e rilasciata grazie alla funzione *lasciaCodaComandi()*; inoltre tale lettura avviene in modo non attivo, grazie all' utilizzo della funzione *pthread_cond_wait*: essa permette di, nel caso in cui non siano presenti file descriptor all' interno della coda, non eseguire letture che risulterebbero inefficienti, fino a che non verrà svegliato.

Una volta ottenuto un *file descriptor*, vanno a leggere se sono presenti richieste di quest' ultimo e in caso positivo sarà arrivato un intero che indicherà l'operazione richiesta, mentre in caso negativo viene effettuata l'operazione di *close* su esso, perché vuol dire che la comunicazione è giunta al termine. Nel caso in cui sia arrivata un intero, il worker andrà a gestire tale richiesta, ricevendo ulteriori dati dal client e rispondendo ad esso con i dati richiesti da esso e/o una stringa che indica l'esito dell'operazione. La lettura dei dati da un file descriptor avviene grazie alla funzione *riceviDati* mentre l'invio dei dati tramite un *file descriptor* avviene tramite la funzione *inviaDati*.

I *thread workers* rimangono in esecuzione fino al raggiungimento di una delle due condizioni di terminazione del *server*:

- l'arrivo del segnale *SIGHUP* e l'assenza di *client* connessi
- l'arrivo del segnale *SIGINT* o *SIGQUIT*

CLIENT

I *client* sono dei processi separati dal server i quali vengono avviati passandogli una stringa a linea di comando, la quale conterrà tutti i vari comandi che dovrà eseguire. Per andare a leggere distintamente ogni singolo comando, e poi eseguirli, ho pensato di suddividere la stringa che viene passata come argomento in una lista di stringhe (codice presente nel file coda.c), dove ogni elemento conterrà un comando o i suoi argomenti, che poi saranno adeguatamente gestiti.

Api

Procedure utilizzate dai client per poter richiedere le operazioni al server. Qui sono contenute anche funzioni ausiliarie alle api, come *relativoToAssoluto*, ovvero una procedura che dato un file o una directory, ritorna il *path* assoluto di esso, e anche *leggiNFileDaDirectory*, la quale data una directory legge N file cercando anche nelle eventuali sotto-directory.

Opzioni aggiuntive

Oltre alle opzioni specificate nel testo, ne sono presenti anche altre disponibili al client:

- **-O**: setta il flag con il quale verrà fatta l'operazione di *openFile* a O_CREATE;
- **-L**: setta il flag con il quale verrà fatta l'operazione di *openFile* a O_LOCK;
- **-OL**: setta il flag con il quale verrà fatta l'operazione di *openFile* a CREATELOCK, ovvero viene fatta l'operazione con entrambi i flag attivi;
- **-a file1,file2[,file3]**: scrive il file1 in memoria, in seguito per ogni file passato ne recupera il contenuto dal disco e lo appende al file1. Se fosse necessario liberare memoria, i file liberati possono essere salvati lato client con l'opzione -D

COMUNICAZIONE CLIENT-SERVER

La comunicazione tra client e server avviene tramite *socket*, grazie alla funzione *inviaDati* per inviare dati e *riceviDati* per riceverli. La prima funzione, con l'ausilio della procedura *writen*, invia tramite *socket* una quantità di bytes passata come parametro. La prima funzione, con l'ausilio della procedura *readn*, legge tramite socket una quantità di bytes passata come parametro.

LOG

Per l'implementazione del log ho scelto di realizzare una funzione (*scriviSuLog*), la quale riceve come input una stringa, la quale conterrà ciò che sarà scritto sul log, e poi riceverà altri argomenti se li deve riportare nel log, altrimenti nessun' altro argomento.

All' interno di tale funzione, per implementare la scelta della ricezione di uno o più argomenti d'ingresso, ho deciso di farle utilizzare una *va_list* così che dopo un controllo sul numero degli argomenti di input, deciderà se riportare sul log solo la stringa o anche altri parametri passati.

Ogni scrittura sul file avviene in mutua esclusione, grazie ad un *lock* adibito esclusivamente alla gestione di questa operazione, il quale viene acquisito all' inizio della procedura e rilasciato alla fine.

MAKEFILE

Il file delle dipendenze di *Unix* serve ad automatizzare l'aggiornamento in modo corretto di più file con delledipendenze, dando la possibilità di mantenere il sistema in uno stato consistente e di facilitare l'utente nella creazione dei file oggetto, della libreria e del file eseguibile. Oltre a questo, permette di facilitare la pulizia dei file non più necessari dopo la terminazione del programma, come i *socket file* per esempio.

COMANDI PER ESECUZIONE PROGRAMMA

L' esecuzione dei due programmi può iniziare grazie al comando *make test1*, *make test2*, oppure *make test3* i quali si occuperanno della compilazione di tutti i file.c seguendo le dipendenze presenti al suo interno, ed eseguiranno i due processi con dei parametri d' ingresso differenti, letti dai rispettivi file.

Inserire nel terminale aperto nella directory:

bash:~$ make test1	(per avviare i test1)
bash:~$ make test2	(per avviare il test2)
bash:~$ make test3	(per avviare il test3)
bash:~$ make clean	(per ripulire la directory di lavoro dai vari file generati)

CODICE DI TERZE PARTI

Per la realizzazione del progetto è stata utilizzata la funzione *stat*, non standard *posix*, utile alla lettura di tutti i path all' interno di una directory e delle eventuali sotto-directory. Tale funzione è stata realizzata da Michael Meskes sotto licenza Free Software Foundation, License GPLv3+; per l'utilizzo di tale funzione sono necessari due include: *sys*/*stat.h* e *sys*/*types*.*h*. Riferimenti web: https://gnu.org/licenses/gpl.html e https://translationproject.org/team/.

È presente ulteriore codice non standard *posix*, presente all' interno del progetto è *realpath*, la quale, per poter essere utilizzata necessita di un include: *stdlib.h*. Tale funzione è stata realizzata da Padraig Brady, sotto licenza Free Software Foundation, License GPLv3+; per l'utilizzo di tale funzione sono necessari due include: *sys*/*stat.h* e *sys*/*types*.*h*. Riferimenti web: https://gnu.org/licenses/gpl.html e https://translationproject.org/team/. Realpath viene utilizzata nella funzione *relativoToAssoluto*, la quale restituisce il path assoluto di un file ricevuto come argomento.

Inoltre sono state utilizzate le funzioni *readn* e *writen*, utili al fine di rispettivamente leggere/scrivere dal/sul socket un numero indicato di bytes; tali funzioni sono state realizzate da W. Richard Stevens e Stephen A. Rago; anch' esse non standard *posix*.

REPOSITORY GITHUB

Tutto il codice sorgente, compresi alcuni file di test, è presente a questo link: https://github.com/Luck199/Progetto-laboratorio-SOL-20-21.git