Università degli Studi di Pisa

Dipartimento di informatica

Progetto estivo, Laboratorio II

Studente: Leone Valerio

Matricola: 656932

Corso A

Strutturazione del codice

Il porogetto è basato su due file principali: “paroliere_srv.c” e “paroliere_cl.c”, che implementano rispettivamente il server e il client. Per migliorare la leggibilità e la manutenzione del codice ho diviso il progetto stesso in più file di header con i relativi file “.c” che insieme ai due file principali vanno a implementare il gioco del “paroliere”.

File:

- **paroliere_srv.c:** implementa il codice relativo al lato server (server multithread), con le logiche di gestione delle risposte alle richieste dei vari client mediante l’utilizzo di più thread. Ogni qualvolta un client si connette al server, questo genera un thread che si occuperà di gestire le richieste del client appena connesso. La funzioni relative ai thread che gestiscono i client sono definite all’interno di questo file
- **paroliere_cl.c:** implementa il codice relativo al lato client, con le logiche di invio di richieste ed elaborazione delle risposte ricevute dal server. Al suo interno è presente la funzione relativa al thread incaricato di ricevere il messaggio asincrono della classifica finale (approfondito in seguito)

- **strutture.h/strutture.c:** in questi file sono contenute le definizioni e le implementazioni delle strutture dati necessarie al corretto funzionamento del gioco
- **matrix.h/matrix.c:** in questi file sono contenute le definizioni e le implementazioni delle funzioni che gestiscono la generazione delle matrici e la stampa di esse
- **serializzazione_messaggi.h/serializzazione_messaggi.c:** in questi file sono contenute le definizioni e le implementazioni delle funzioni relative alla serializzazione e deserializzazione dei messaggi tra server e client
- **controllo_parole_matrix.h/controllo_parole_matrix.c:** in questi file sono contenute le definizioni e le implementazioni delle funzioni che si occupano di verificare se una parola è presente o meno nella matrice
- **format_parole.h/format_parole.c:** in questi file sono contenute le definizioni e le implementazioni delle funzioni necessarie a formattare correttamente le parole in modo tale che possano essere efficientemente usate per i vari controlli su di esse
- **macros.h:** contiene macro utili alla gestione degli errori delle syscall
- **trie.h/trie.c:** questi file contengono le definizioni e le implementazioni di strutture dati e funzioni relative all'implementazione della struttura dati ad albero del trie. Qui sono contenute anche le funzioni di inizializzazione e ricerca nel trie

Strutture dati utilizzate

Le strutture dati implementate per la gestione dei vari meccanismi di gioco sono le seguenti

- **Player:** struttura che contiene username del giocatore, relativo punteggio e un set di parole che saranno quelle già usate in una partita. Per gestire tutti i giocatori che si connettono al server ho deciso di andare a realizzare un vettore di strutture Player (condiviso tra tutti i thread), facendo in modo che ogni posizione del vettore venisse riservata ad un determinato thread, in modo da semplificare la gestione di più thread; tenendo traccia di quanti giocatori sono collegati e di conseguenza di quanti slot sono ancora liberi e in che posizione tali slot si trovino, ogni volta che un thread viene inizializzato riceverà anche l'indice della posizione del giocatore all'interno del vettore così quando il giocatore si scollega, tutte le informazioni relative ad esso vengono cancellate e quella posizione sarà pronta ad accogliere un nuovo client

```
typedef struct {
    char name[MAX_LENGTH];
    int score;
    char set[BUFFER_SIZE][MAX_LENGTH];
} Player;
```

```
errà le info dei giocatori
players = (Player*)malloc(MAX_CLIENTS * sizeof(Player));
for(int i = 0; i < MAX_CLIENTS; i++) {
    players[i].score = 0;
}
```

- **DataHandler:** struttura che contiene i parametri da passare alla funzione del thread che gestisce i client (chiamata “handler”) per la corretta gestione delle richieste/risposte. Ci sono un puntatore all’array dei giocatori, alla matrice di gioco, al file descriptor necessario alla connessione con il server, un puntatore alla radice del trie per la verifica della presenza nel dizionario di una parola, il seed per la generazione casuale delle matrici, un puntatore alla classifica generale (condivisa tra tutti i thread), l’indice della posizione occupata dal giocatore nel vettore players

```
typedef struct {
    Player* players;
    char** matrix;
    int client_fd;
    int index;
    char* csv;
    char* matrix_filename;
    int seed;
    TrieNode* root;
} DataHandler;
```

- **Message:** struttura che contiene le informazioni che defniscono I messaggi: lunghezza, tipo e contenuto

```
typedef struct {
    int length;
    char type;
    char* data;
} Message;
```

- **DataScorer:** struttura che contiene I parametri necessari alla funzione del thread scorer per generare la classifica e inviarla a tutti I client e azzerare I punteggi in vista della partita successiva

```
typedef struct {
    Player* players;
    int* client_fds;
    int* client_slots;
} DataScorer;
```

- **DataClient:** struttura che contiene I parametri per il corretto funzionamento della funzione relativa al thread che si occupa delle ricezione del messaggio della classifica finale nel client

```
typedef struct {
    int client_fd;
    Message msg;
    char* buffer;
    char* serialized_msg;
    int* msg_size;
} DataClient;
```

- **TrieNode:** implementa un nodo del trie: la variabile isEndOfWord mi dice se ho finito di inserire la parola e di poter affermare che essa appartiene all'albero e dunque al dizionario, vettore di nodi figli di dimensione dell'alfabeto per ogni lettera dell'alfabeto per implementare I rami

```c
typedef struct TrieNode {
    struct TrieNode* children[ALPHABET_SIZE];
    int isEndOfWord;
} TrieNode;
```

- **bacheca:** implementa la bacheca contenente I messaggi, realizzata come un array bidimensionale, con un mutex per gestire la corretta scrittura/lettura dei messaggi mediante le funzione read_message e add_message

```c
/*Bacheca*/
char bacheca[MAX_MSG][MAX_SIZE_MSG];
int tail = 0;
int count = 0;
pthread_mutex_t mutex_bacheca = PTHREAD_MUTEX_INITIALIZER;
```

Algoritmi e soluzioni utilizzate

Gli algoritmi utilizzati in questo progetto sono principalmente quelli per verificare la presenza di una parola all'interno della matrice (seguendo I criteri dettati dall regole del gioco) e la ricerca della parola all'interno del dizionario.

Per quanto riguarda la verifica della presenza di una determinata parola all'interno della matrice ho utilizzato tre funzioni:

is_valid_move: Questa funzione ausiliaria verifica se uno spostamento a una determinata cella nella matrice è valido. Controlla che le coordinate siano all'interno dei limiti della matrice e che la cella non sia già stata visitata

search_from: Questa funzione ricorsiva cerca la parola partendo da una data posizione nella matrice:

- Se l'indice della parola è uguale alla lunghezza della parola, significa che la parola è stata trovata e ritorna 1.
- Verifica se il movimento corrente è valido e se il carattere corrente nella matrice corrisponde al carattere corrente della parola.

- Segna la cella come visitata e prova a muoversi nelle 8 direzioni
- Se una delle direzioni restituisce 1, la parola è stata trovata.
- Ripristina lo stato "non visitato" della cella se la ricerca in quella direzione non ha avuto successo e ritorna 0.

exist: Questa funzione applica l'algoritmo di ricerca partendo da ogni cella della matrice:

- Inizializza una matrice di visitati a 0.
- Itera su ogni cella della matrice e chiama search_from per verificare se la parola può essere trovata a partire da quella cella.
- Se search_from restituisce 1 per una qualsiasi cella, la funzione ritorna 1, altrimenti ritorna 0.

Per quanto riguarda la ricerca di una parola nel trie ho implementato la funzione **search**, essa prende in input una parola e la radice del trie e scende lungo I rami per la lunghezza della parola inserita. Se si raggiunge un nodo tale che isEndOfWord è 1 abbiamo trovato la parola

Meccanismo di comunicazione con messaggi

Per implementare la comunicazione tra server e client come da specifiche (messaggi con tipo, lunghezza e contenuto) ho ritenuto opportuno creare una struttura dati che rappresentasse il messaggio con I suoi tre campi e utilizzare il meccanismo di serializzazione e deserializzazione del messaggio per utilizzare una sola write/read per inviare e ricevere I messaggi e poter riutilizzare le stesse variabili per rispondere o inviare altre richieste. Quando viene inviato un messaggio, I tre campi che lo costituiscono vengono concatenati a formare un’unica stringa che sarà appunto il contenuto della write/read e sarà lungo esattamente quanto la somma delle dimensioni dei campi che lo compongono

```
void deserialize_message(char* buffer, Message* msg) {
    memcpy(&msg -> length, buffer, sizeof(int));
    buffer += sizeof(int);
    memcpy(&msg -> type, buffer, sizeof(char));
    buffer += sizeof(char);
    memcpy(msg -> data, buffer, msg -> length);
}

void serialize_message(Message* msg, char* buffer) {
    memcpy(buffer, &msg -> length, sizeof(int));
    buffer += sizeof(int);
    memcpy(buffer, &msg -> type, sizeof(char));
    buffer += sizeof(char);
    memcpy(buffer, msg -> data, msg -> length);
}
```

Logica del server

Il server è multi-thread per garantire la corretta gestione di più client in contemporanea. All'avvio del server vengono controllati I parametri forniti da linea di comando: se il file da cui estrarre le matrici è valido verrà aperto e utilizzato dalla funzione **random_matrix_generator,** lo stesso vale per il seed e per il nome del file del dizionario, che verrà utilizzato dalla funzione **initTrie** per inizializzare l'albero. Oltre alle strutture relative alla gestione delle parole vengono inizializzate quelle per la gstione dei giocatori: viene inizializzato a 0 il vettore **client_slots** che conterrà poi 1 negli indici in cui si registreranno I client e viene inizializzato l'array condiviso tra I vari thread che contiene le informazioni dei giocatori attualmente connessi. Viene poi invocata la funzione **alarm** che serve a scandire le varie fasi di gioco, così come definito nella funzione **gestore_alarm** che riceve in input il segnale relativo all'alarm inviato ogni qualvolta avviene un cambio della fase di gioco. La matrice viene generata da **random_matrix_generator** ad ogni inizio partita. A questo punto il server è in attesa di connessioni da parte dei client, ogni volta che un client cerca di connettersi il server verifica la presenza di slot liberi e in caso positivo consente la connessione del client, andando a settare ad 1 la rispettiva posizione in **client_slots.** La corretta sincronizzazione è gestitata da vari mutex e condition variables. Per la gestione dei messaggi sincroni tra server e client ho implementato un **thread handler** che gestisce le richieste del client: riceve messaggi serializzati che verranno destrutturati per poter essere letti e processati in modo tale da determinare la corretta risposta a seconda del tipo di messaggio **MSG_TYPE** ricevuto.

Come richiesto da specifiche del progetto il messaggio della classifica finale viene inviato da un **thread scorer** che stila la classifica, genera un csv che la contiene e sveglia con una broadcast il **thread async** che si occupa di inviare a tutti I client la classifica finale, che torna in attesa sulla variabile di condizione fino al termine della partita successiva.

Ogni volta che un client si disconnette si libera in **client_slots** una posizione (quindi viene impostata a 0), e la posizione da liberare è contenuta nei dati del thread. Per permettere questo meccanismo ho dichiarato client_slots come globale.

La gestione del segnale **SIGINT** viene gestita da un apposito handler.

Logica del client

Il client consente al giocatore di partecipare effettivamente al gioco. Una volta avviato, vengono controllati I parametri passati da linea di comando e viene dunque stabilita la connessione al server. Se la connessione viene stabilità con successo verrà visualizzato un messaggio con le regole e I comandi disponibili. A questo punto l'utente potrà digitare I comandi che verranno serializzati dal client ed inviati al server. Non appena il client riceve una risposta questa viene processata dal **thread async** il cui compito è quello di processare solamente messaggi di tipo matrice e classifica

finale. La risposta, ancora serializzata, verrà quindi scomposta da questo thread che la elabora solo nel caso in cui il tipo sia **MSG_MATRICE** o **MSG_PUNTI_FINALI**. Se il messaggio di risposta non è di questi tipi allora async sveglia il thread principale (che si era messo in attesa sulla variabile di condizione) con una signal, questo porvvederà alla corretta elaborazione del messaggio. La gestione del segnale **SIGINT** viene gestita da un apposito handler.

Programmi di test e istruzioni per la compilazione

Per testare il codice ho avviato in locale, mediante l'apertura di più terminali, vari client che si connnettevano al server e che simulavano una partita. Per semplifacre il testing ho ridotto il massimo numero di client, per verificare che venissero inviati I corretti messaggi di errore e che si liberassero gli opportuni slot.

E' possibile compilare il codice sia del client che del server mediante il Makefile

- **make all:** genera tutti gli eseguibili
- **make testServer:** esegue il server senza alcun parametro opzionale
- **make testServer1:** esegue il server con i parametri –marix e –durata
- **make testServer2:** esegue il server con i parametri –matrix, --durata e –seed
- **make testServer3:** eesegue il server con i parametri –durata –seed
- **make testServer4:** esegue il server con i parametri –matrix, --durata e –diz
- **make testClient:** esegue il client

Per pulire file oggetto ed eseguibili: **make clean.**