Il Paroliere online
Relazione del Progetto Finale del corso di Laboratorio II

Francesco Di Stasio
Corso di Laurea in Informatica, Università di Pisa

Anno 2023-2024

1 Struttura del Progetto

L'elaborato è suddiviso in directory relative rispettivamente a:

1. File sorgenti con estensione ".c" (directory /src)
2. File header con estensione ".h" (directory /include)
3. File relativi ai dati con estensione ".txt" (directory /data)

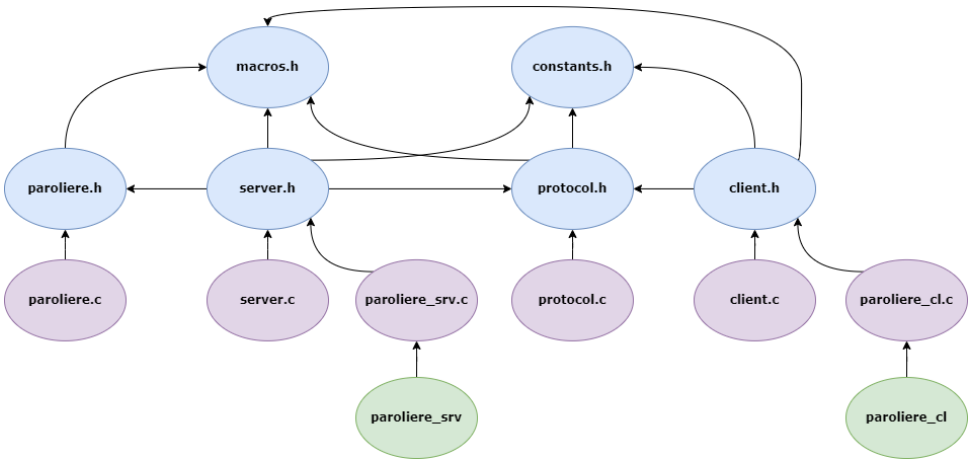
Nella cartella principale è invece riposto il Makefile.

2 Installazione

Per installare l'intero progetto sarà sufficiente eseguire il comando "make" da linea di comando (oppure "make all"). Dopodichè verranno compilati, tramite una compilazione separata, due file eseguibili: "`paroliere_srv`" e "`paroliere_cl`". Rispettivamente relativi a server e client.

3 Dipendenze dei file

Di seguito è riportata la rappresentazione di quelle che sono le dipendenze tra i vari file sorgenti e header. Per semplicità di visualizzazione dello schema sono stati omessi i file oggetto. Inoltre, per lo stesso motivo, per le varie dipendenze vale una proprietà transitiva (Esempio: "server.c" dipende da "server.h", il quale dipende da "paroliere.h". Quindi "server.c" dipende da "paroliere.h").



In questo modo, per quanto riguarda il lato server, esso è formato dai file sorgenti "`paroliere_srv.c`" (contentente la funzione "main()"), "server.c" e "paroliere.c". D'altra parte il lato client comprende "`paroliere_cl.c`" (contenente la funzione "main()") e "client.c". Il file "protocol.c" invece è condiviso sia dal server che da client in quanto definisce funzioni per il protocollo di comunicazione di essi.

I file header principali sono invece "macros.h" che contiene macro per la gestione degli errori, e "constants.h" che contiene costanti relative ai vari file sorgenti.

Inoltre i file header condivisi da più file sorgenti sono contenuti in degli `#ifndef` così da non definirli più volte durante la compilazione.

4 Makefile

Il Makefile adotta un approccio di compilazione separata. In esso vengono specificate tutte le dipendenze dei vari file sorgenti e vengono definite le seguenti regole principali:

1. all: Tramite la quale è possibile compilare tutto il progetto, andando a generare i due file eseguibili "`paroliere_srv`" e "`paroliere_cl`". Essa inoltre essendo la prima regola del makefile non necessita l'invocazione tramite il comando "make all" ma è sufficiente "make".
2. clean: Per andare a ripulire la directory da i file oggetto e i file eseguibili del progetto.
3. srv: Per andare ad eseguire il file eseguibile del server con i parametri minimi richiesti.
4. cl: Per andare ad eseguire il file eseguibile del client con i parametri minimi richiesti.

Sono state poi definite altre regole di test per verificare i vari comportamenti del server con diversi parametri da riga di comando.

5 Server

5.1 Procedimento generale

Facendo riferimento al file sorgente "`paroliere_srv.c`" esso è articolato come segue:

1. All'avvio viene definita una maschera per bloccare il segnale "SIGALRM", dato che esso non dovrà essere gestito dal processo principale.
2. Viene installato un gestore di segnale per andare a gestire i segnali "SIGINT" (generato da Ctrl-C) e "SIGUSR1" (generato da un thread arbitro).

3. Segue poi una procedura relativa alla gestione dei parametri da linea di comando. Essa comprende il controllo della correttezza e l'inizializzazione di una struttura "params" dedicata ad essi.

4. Successivamente viene creato il thread scorer. Egli è responsabile della stesura della classifica di gioco tramite una coda, condivisa tra i thread che gestiscono i client, allocata sullo heap.

5. Dopodichè viene creato il thread arbitro. Egli si occupa di inizializzare la matrice di gioco, gestire le pause tramite delle funzioni "alarm()" e di inviare a tutti i thread che gestiscono i client un segnale di tipo "SIGUSR1" (tramite la funzione "`pthread_kill()`") quando essi devono scrivere sulla coda condivisa dei punteggi.

6. A questo punto viene creato il socket e il programma inizia un loop infinito (interrotto solo dal segnale "SIGINT") dove per ogni connessione accettata istanzia un rispettivo thread. Ogni thread memorizza il suo tid in un array globale "clientHandlerThreads" e il file descriptor associato al client in un'altro array globale "clientSocketFileDescriptor". Dal momento in cui un client tenta di connettersi, si verifica se vi è un elemento libero nell'array dei tid. In tal caso si sovrascrive con il tid del nuovo thread (e con il file descriptor nell'array associato). Nel caso in cui l'array fosse pieno la connessione viene chiusa. Il numero massimo di connessioni è definito nel file "constants.h" tramite la costante "`MAX_CONNECTED_CLIENTS`" (32).

7. Dal momento in cui il processo riceve il segnale "SIGINT", l'handler di segnale si limita ad una stampa sullo standard output della disconnessione. Il loop infinito quindi termina, dato che la chiamata di sistema "accept()" è stata interrotta causando l'errore Interrupted System Call, per il quale è definito un "break" dal ciclo, e il processo, utilizzando l'array "clientSocketFileDescriptor", invia un messaggio di tipo "`MSG_FINE`" per comunicare ad ogni client che il server sta chiudendo.

5.2 Funzionamento dei Thread

Per quanto riguarda il file sorgente "server.c" esso definisce delle funzioni per gestire le operazioni descritte nella sezione precedente. In particolare definisce le funzioni per i thread che gestiscono i client, il thread scorer ed il thread arbitro.

1. I thread che gestiscono i client operano nel seguente modo:

   Inizialmente gestiscono i parametri. Essi sono rispettivamente: il file descriptor del client, il nome del file dizionario ed un puntatore alla coda condivisa.

   Successivamente entrano in un loop infinito nel quale si mettono in attesa delle richieste dei client e per ognuna inviano la risposta appropriata. Dal momento in cui una read viene interrotta dal segnale "SIGUSR1" (generato dal thread arbitro) si preoccupano di scrivere i dati sulla coda condivisa dei punteggi dei client. Per farlo ogni thread si sincronizza e confrontando il numero di dati scritti sulla coda, con il numero di client registrati (intero globale "numRegisteredClients") capiscono se sono gli ultimi a scrivere sulla coda. Solo in questo caso (per fare in modo che scorer stili la classifica con tutti i dati) inviano una "`pthread_cond_signal()`" al thread scorer in attesa sulla variabile di condizione "notFullQueue". Dopo aver fatto ciò si mettono in attesa sulla variabile di condizione "notReadyRanking" aspettando che il thread scorer abbia elaborato la classifica finale.

   Per quanto riguarda le richieste dei client, ognuna di esse viene gestita, in base al tipo, da una funzione "`handle_client_request()`" che invia al client la risposta appropriata. In particolare il server risponde a messaggi del tipo:

(a) "`MSG_REGISTRA_UTENTE`" registrando il nome proposto (se valido, non già utilizzato e non più lungo di 10 caratteri) in un array globale "clientsUsername" contenente tutti gli username e aumentando il numero di client registrati. Un nome utente è stato considerato valido solo se non contiene caratteri speciali. Questo per evitare qualsiasi tipo di errore legato all'invio di essi.

(b) "`MSG_PAROLA`" (se il gioco è in corso) controllando la validità della parola sulla matrice globale "board" e assegnando o meno i punti rispettivi. Per controllare che una parola non sia già stata proposta, ogni thread, definisce una linked list delle parole proposte, dove vengono aggiunte per ogni partita, tutte le parole valide che il client propone (al termine di ogni partita viene ripristinata).

(c) "`MSG_MATRICE`" andando a convertire la matrice di gioco in una stringa ed inviandola al client. In questo caso viene inviato anche il tempo rimanente della pausa o della partita andando a chiamare la funzione "alarm(0)", che restituisce il tempo rimanente alla prossima alarm, e andandola a richiamare subito dopo per impostarla nuovamente con il tempo restante (dato che alarm(0) annulla la alarm precedente).

(d) "`MSG_PUNTI_FINALI`" (se il gioco non e' in corso) inviando al client una stringa globale in formato CSV "ranking", inizializzata dal thread scorer, contentente la classifica dei punteggi. Questo tipo di messaggio è inviato sia su richiesta dell'utente durante la pausa, sia in modo asincrono quando la partita termina.

(e) "`MSG_FINE`" andando a ripulire i dati allocati sullo heap dal thread, chiudendo il file dizionario e andando a compiere tutte le procedure per rimuovere la sua registrazione, se è registrato. Infine il socket viene chiuso e il thread terminato.

Inoltre se il client non è registrato, il server risponde positivamente solo a messaggi di tipo "`MSG_FINE`" e "`MSG_REGISTRA_UTENTE`". Egli non risponde ad altri tipi di richieste. Infatti messaggi dell'utente non validi o il messaggio "aiuto" vengono gestiti direttamente dal client per non inviare dati superflui o errati al server.

2. Il thread arbitro è responsabile dell'inizializzazione della matrice globale "board" e della gestione delle pause.

   Egli inizialmente gestisce i parametri. Essi sono rispettivamente: il nome del file delle matrici, la durata della partita e una variabile intera che indica se generare le matrici casualmente o tramite file.

   Successivamente definisce un set di segnali dove imposta ad 1 "SIGALRM" per andarlo ad attendere in seguito con "sigwait()"

   A questo punto entra in uno dei due loop infiniti (uno rispettivo alla generazione casuale e l'altro alla generazione tramite file).

   In ogni iterazione inizializza la matrice e chiama la funzione "`timer_sheduler()`". Questa imposta due alarm: la prima relativa alla pausa e la seconda alla durata della partita. Entrambe generano quindi il segnale "SIGALRM" che viene atteso da delle "sigwait()". Successivamente tramite un flag globale "inGame" viene aggiornato lo stato di gioco (dopo il segnale di fine partita inGame sarà 0 e dopo il segnale di inizio partita sarà ad 1). L'arbitro poi, avvalendosi dell'array globale dei tid dei thread che gestiscono i client, invia ad ognuno un segnale di tipo "SIGUSR1" per comunicargli di scrivere i dati sulla coda. Inoltre prima di farlo imposta un flag globale "wroteData" ad 1, che una volta letti i dati, il thread scorer imposterà a 0, per fare in modo che i client non scrivano i dati più volte durante la pausa.

3. Il thread scorer è colui che si occupa di sincronizzarsi sulla coda condivisa dei punteggi con i thread che gestiscono i client, per stilare la classifica di gioco. Egli inizialmente gestisce il suo unico parametro, un puntatore alla coda condivisa, e poi inizia un loop infinito. Successivamente se la coda è vuota si mette in attesa sulla variabile di condizione "notFullQueue", dove verrà risvegliato dall'ultimo thread che gestisce i client che scrive i suoi dati. A questo punto il thread aggiorna il flag globale "wroteData" a 0, copia la coda in un array di tipo "player" e ordina quest'ultimo tramite la funzione "qsort()" in ordine decrescente di punteggio. Successivamente ripulisce la coda eliminando gli elementi e scrive sulla stringa globale "ranking" la classifica di gioco andandola a formattare tramite la funzione "`make_ranking()`" . Infine effettua una "`pthread_cond_broadcast()`" per risvegliare tutti i thread che stavano aspettando la classifica sulla rispettiva variabile di condizione "notReadyRanking".

5.3 Algoritmo del Paroliere

Per verificare la correttezza di una parola e per gestire la matrice di gioco, il file sorgente "paroliere.c" definisce varie funzioni. In particolare:

1. La funzione "`word_checker`", utilizzata per controllare la validità di una parola nella matrice, adotta un comportamento in stile Depth First Search. Infatti essa definisce una matrice di interi degli elementi visitati dove traccia , tramite una funzione ricorsiva "`word_rec_check()`", ogni percorso possibile partendo dalla prima corrispondenza con l'elemento della matrice e la prima lettera della parola. Nello specifico la funzione ricorsiva controlla ogni elemento adiacente e, se corrisponde alla lettera successiva della parola, effettua una chiamata ricorsiva. Quindi dal momento in cui la funzione arriva all'ultimo elemento, la parola è corretta.

   Per gestire il carattere "Qu" la matrice "board" è stata definita come matrice di stringhe di 2 elementi. Nello specifico ogni funzione che coinvolge questo carattere adotta una gestione particolare che è specifica per ogni determinata situazione.

2. Per effettuare il confronto della parola, essa viene trasformata in upper case (dato che gli elementi della matrice lo sono) tramite la funzione "`to_upper_word()`", mentre per verificare la correttezza semantica della parola nel dizionario, essa è trasformata in lower case dalla funzione "`to_lower_word()`" (dato che le parole del dizionario sono rappresentate in questo modo).

3. La funzione per la generazione random della matrice board "`init_rand_board_matrix()`" si occupa, tramite il seed passato come parametro da linea di comando oppure tramite il seed determinato da "time(NULL)", di inizializzare la matrice in modo random.

   La funzione analoga per l'inizializzazione tramite file "`init_file_board_matrix()`" legge una riga del file delle matrici alla volta ed effettua la funzione "strtok()" per gestire gli elementi di quest'ultima. Inoltre compie dei controlli di base sulla riga quali: la lunghezza minima e la trasformazione di tutti gli elementi in upper case in caso non lo fossero. Per gestire il caso in cui la riga non fosse valida oppure il file delle matrici fosse finito, al posto di restituire un errore, la funzione inizializza la matrice in modo random. In questo modo il server potrà continuare a generare matrici e smettere solo quando riceve "SIGINT".

6 Client

6.1 Procedimento Generale

Facendo riferimento al file sorgente "`paroliere_cl.c`" esso è articolato come segue:

1. Inizialmente viene installato un gestore di segnale per "SIGINT" per fare in modo di ignorarlo. Questa scelta è dettata dal fatto che i client per disconnettersi correttamente e senza alcun tipo di errore devono utilizzare il comando "fine".

2. Successivamente viene effettuato un controllo sul numero corretto di parametri da linea di comando e l'inizializzazione di essi.

3. Dopodichè viene creato il socket e il client si connette al server.

4. Vengono poi creati due thread. Il primo che invia le richieste dell'utente al server, il secondo che legge le risposte del server e le comunica all'utente.

5. Infine viene attesa la terminazione del thread lettore, dato che la terminazione di quest'ultimo implica una disconnessione dovuta alla ricezione di `"MSG_FINE"`. Questo può avvenire nel caso in cui il server venga terminato tramite "SIGINT" oppure tramite il messaggio "fine" da parte dell'utente. In quest'ultimo caso il thread scrittore invia `"MSG_FINE"` al server, il quale provvede ad effettuare correttamente le operazioni di disconnessione e ad inviare al client `"MSG_FINE"` così da permettergli di terminare.

6.2 Funzionamento dei Thread

Per quanto riguarda il file sorgente "client.c" esso definisce delle funzioni per gestire le operazioni descritte nella sezione precedente. In particolare definisce le funzioni per il thread lettore dei dati dal server e il thread scrittore dei dati verso il server.

1. Il thread scrittore dei dati letti dallo standard input e inviati verso il server, inizialmente gestisce il suo unico parametro, il descrittore del socket, e poi inizia un loop infinito. A questo punto scrive il prompt sullo standard output e si mette in attesa di richieste dell'utente. Ogni richiesta viene elaborata dalla funzione "`send_user_request()`", la quale in base al tipo di comando, interpreta la richiesta e la invia verso il server.

   Se un comando non è valido oppure è "aiuto" il thread non interpella nessun'altro ma risponde direttamente all'utente inviando la lista dei comandi disponibili. Lo stesso comportamento è adottato nella registrazione. Infatti, se l'utente prova a registrarsi tramite il comando "`registra_utente`" inserendo caratteri non validi, non verrà inviato alcun messaggio al server per evitare di generare errori. Per quanto riguarda il controllo sulla lunghezza del nome utente questo è lasciato al server.

   Il thread inoltre dopo aver gestito la richiesta effettua una "sleep(1)". Questo per sincronizzare sullo standard output la stampa del prompt dopo la risposta del server (opzione prettamente grafica che non impatta sul funzionamento).

2. Per quanto riguarda il thread lettore dei dati dal server e inviati all'utente sullo standard output, egli inizialmente gestisce il suo unico parametro, il descrittore del socket, e poi inizia un loop infinito. A questo punto si mette in attesa che il server invii una risposta relativa alla richiesta dell'utente, e in base al tipo di essa, il thread la gestisce inviando i risultati a video. Dal momento in cui riceve un messaggio di tipo `"MSG_FINE"` termina, sblocca di conseguenza la join che lo stava attendendo, e come già anticipato non essendoci una join che attende l'altro thread, il programma termina.

7 Protocollo

Client e Server comunicano con un protocollo ben definito, inviando e leggendo in ordine: lunghezza, tipo ed eventuali dati. Questo per andare ad allocare buffer di lunghezza relativa alla lunghezza comunicata e non utilizzare lunghezze fisse.

Questo approccio è gestito nel file sorgente "protocol.c", il quale utilizza due funzioni principali:

1. La funzione `read_socket()` effettua rispettivamente 3 "read()" sul descrittore di file passato come parametro e ognuna di esse opera una particolare gestione degli errori. Infatti per permettere al server di capire quando viene interrotto dal segnale "SIGUSR1", la funzione restituisce un -1 se una "read()" viene interrotta andando ad impostare errno ad ERINTR (Interrupted System Call). Oltre a ciò i dati che legge sono memorizzati in una struttura "message" contenente i campi relativi a lunghezza, tipo ed eventuali dati del messaggio.

2. La funzione `write_socket()` analogamente effettua 3 "write()" sul descrittore di file passato come parametro. Essa si avvale della funzione "`init_message()`" per inizializzare la struttura "message" da inviare.

8 Test e Gestione degli Errori

I test per determinare il funzionamento e la correttezza del progetto si sono svolti seguendo i punti principali seguenti:

1. **Correttezza dei parametri passati da riga di comando.** In questo caso, parlando del lato server, sono state testate le varie configurazioni dei parametri opzionali (disponibili come regole nel Makefile), i quali possono essere disposti in qualsiasi ordine grazie alla funzione "`init_params()`". Questa scorre l'array "argv" e in base alla parola chiave corrente, assegna al parametro opzionale (in una struttura apposita "params"), il valore che segue la parola chiave. In generale vengono poi testati, sia lato client che server, il numero minimo di parametri.

2. **Assenza di Race Conditions.** Ogni accesso a variabili condivise e variabili di condizione è infatti effettuato all'interno di un mutex.

3. **Assenza di Deadlock o Starvation.** In particolare per evitare il deadlock sono state evitate situazioni del tipo "wait while holding" e in caso di mutex annidati, l'acquisizione ed il rilascio di essi è stato sempre effettuato in ordine di acquisizione.

4. **Correttezza della gestione dei segnali e del gestore di essi.** In questo ambito il gestore di segnale si limita ad effettuare operazioni basilari e signal safe (come "write()"). Infatti la gestione principale dei segnali è effettuata al di fuori di esso tramite opportune strategie come: la funzione "sigwait()" nel thread arbitro per attendere "SIGALRM", e l'uscita dal ciclo infinito alla ricezione di "ERINTR" da parte del processo server che accetta le connessioni, per attendere "SIGINT".

5. **Operazioni permesse dall'utente.** In particolare sono state testate tutte le situazioni in cui il client potrebbe richiedere un servizio che in quel momento non gli può essere fornito. Ad esempio la registrazione, che non può avvenire se il client l'ha già effettuata, oppure il fatto di proporre parole quando il server è in pausa.

6. **Validità del file delle matrici.** Come già anticipato, il file delle matrici è stato testato per funzionare sia con matrici non valide in termini di lunghezza minima (inizializzandone una random al posto di quella non valida), sia con matrici composte da lettere minuscole (trasformandole in

maiuscole e inizializzando la matrice). Per tale scopo il file "matrix.txt" contiene vari casi gestiti grazie a queste precauzioni.

7. **Validità del file dizionario.** Analogalmente al file delle matrici, è stato definito un file dizionario `"dictionary_custom.txt"` tramite il quale effettuare test riguardanti l'utilizzo di un file dizionario personalizzato.

8. **Correttezza dell'algoritmo del paroliere.** Questo è stato testato con varie configurazioni di parole (anche contenenti "Qu") in modo tale da riconoscerle in ogni tipo di disposizione sulla matrice. E' stato anche gestito il caso in cui l'algoritmo, seguendo il percorso compiuto da una parola, si ritrovasse alla fine in un vicolo cieco con la parola errata, ma se avesse compiuto un altro tipo di percorso la parola sarebbe stata giusta. Esempio: se la parola fosse stata "pentola" e nella matrice ci fossero stati due percorsi riconducibili rispettivamente a "pentolo" e "pentola", l'algoritmo pur iniziando seguendo il primo caso, se alla fine non è valido prosegue e trova il secondo dato che man mano ha ripristinato le celle visitate.

9. **Gestione di "Qu".** Per quanto riguarda il carattere "Qu" sono stati testati e gestiti tutti gli ambiti in cui potrebbe creare problemi. Ad esempio nell'algoritmo del paroliere o nell'assegnamento dei punteggi in una parola che contiene questo carattere.

10. **Gestione delle risposte del server.** In particolare le risposte del tipo `"MSG_OK"` ed `"MSG_ERR"` spesso inviano all'utente una descrizione dettagliata del messaggio (ad esempio un errore sulla registrazione invia anche il motivo per il quale non è stato possibile effettuarla). Questo per favorire debugging e fornire più informazioni all'utente.

11. **Assenza di Memory Leaks.** Per quanto riguarda l'utilizzo dello heap, per ogni "malloc()" effettuata, dal momento in cui la risorsa non è più necessaria, questa viene liberata tramite "free()". In particolare nell'invio di messaggi tramite la funzione `"free_message()"` , definita dal protocollo (che dealloca il campo data dei messaggi), nelle linked list associate alle parole indovinate dai client oppure nella coda condivisa dei punteggi.

12. **Gestione degli errori nelle chiamate di sistema.** Tramite le macro definite in "macros.h", ogni chiamata di sistema testa il suo valore di ritorno. Oltre a queste sono definite macro per controllare i valori restituiti da "malloc()" oppure da funzioni che operano su file (come "fopen()").

13. **Correttezza della gestione delle varie connessioni.** Questo è stato testato andando a provare diversi casi in cui si raggiunge il limite massimo di connessioni disponibili, e andando a verificare che terminando altre connessioni, queste venissero gestite correttamente aggiornando sempre i rispettivi array globali "clientHandlerThreads" e "clientSocketFileDescriptor".

14. **Correttezza delle operazioni su variabili globali**. In relazione alla voce precedente, in ambito generale, è stata controllata la corretta gestione di tutte le variabili globali. Questo andando a verificare che venissero aggiornate correttamente in seguito a diverse operazioni (Esempi: array globali degli username, coda condivisa dei punteggi, matrice di gioco).

15. **Correttezza dello svolgimento delle parite**. Infine sono stati compiuti vari test andando a giocare delle partite e controllando che client e server si comportassero come previsto.