WATOR: un simulatore distribuito di un modello biologico

Relazione del Progetto del modulo di laboratorio SOL 2014/2015

Studente: Carlo Alessi
Matricola: 526092

Indice

1) Introduzione

1.1) Strutturazione del codice

Il progetto è suddiviso in più file e fa uso di un'unica libreria chiamata libWator.a. L'idea della suddivisione è quella di far corrispondere a ciascun modulo una funzionalità. I file coinvolti sono:

- wator.h
  Contiene i tipi e i prototipi delle funzioni che verranno usate per la simulazione.
- wator.c
  Contiene le definizioni delle funzioni dichiarate in wator.h.
- help.h
  Questo file offre funzionalità a tutti gli altri moduli. In particolare definisce delle macro utilizzate per il controllo degli errori, contiene i prototipi di funzioni che verranno invocate come sub-routine, definisce nuovi tipi di dato.
- help.c
  Contiene le definizioni delle funzioni dichiarate in help.h.
- myqueue.h
  Questo file ha lo scopo di offrire tutte le funzionalità per gestire una coda che implementa una politica FIFO. A tale scopo, oltre a definire dei nuovi tipi di dato, definisce dei prototipi di funzioni per la creazione e la cancellazione di una coda, push e pop di elementi e test di coda vuota.
- myqueue.c
  Contiene le definizioni delle funzioni dichiarate in myqueue.h.
- mutex.h
  In questo file vengono dichiarate tutte le funzioni usate per accedere in mutua esclusione alle risorse del sistema.
- mutex.c
  Contiene le definizioni delle funzioni dichiarate in mutex.h. Inoltre vengono definite delle mutex usate per l'accesso in mutua esclusione delle risorse del sistema (coda, matrice del pianeta, contatori, flag, …) e delle variabili di condizione per gestire la sincronizzazione tra i thread.
- myconn.h
  In questo file vengono definiti i prototipi delle funzioni usate per la connessione tra il processo wator e il processo visualizer. Inoltre viene definito un nuovo tipo per indicare la struttura che avrà il messaggio.

- myconn.c
  Contiene le definizioni delle funzioni dichiarate in myconn.h.
- main.c
  Questo file è l'unico, oltre al file visualizer.c, ad avere la funzione principale main(). Identifica il processo wator, il client, che si occupa di gestire la simulazione.
- visualizer.c
  Identifica il processo visualizer, il server, che si occupa della visualizzazione.

1.2) Idee guida

Nella realizzazione del progetto è stato cercato di limitare il più possibile lo scope delle funzioni e delle variabili.
Certe volte è stata preferita la semplicità del codice rispetto ad un aumento delle performance.

2) Strutture Dati

2.1) La coda dei task

I task che devono svolgere i thread worker sono sotto-matrici, dei rettangoli. Ogni rettangolo è identificato da due punti. A tale scopo è stato dichiarato un nuovo tipo Point_t che ha come campi l'ascissa e l'ordinata del punto. La coda è una struttura di tipo Queue_t* , implementata mediante una linked list i cui elementi sono di tipo Rettangolo_t.
Nella struttura che rappresenta la coda è presente un puntatore head per estrarre facilmente i task e un puntatore tail per l'inserimento in coda.
Le strutture Point_t, Rettangolo_t, Queue_t sono dichiarate nell' header myqueue.h.

2.2) Il messaggio

Il messaggio inviato dal client al server è rappresentato da una struttura chiamata msg_t. Come campi ha il numero di righe e colonne della matrice e un array che contiene la codifica dei caratteri. Per rappresentare 3 caratteri {W, F, S} servono $\log(n° caratteri)$ = 2 bit, dove per log si intende il logaritmo in base 2.

Data la codifica dei caratteri, è stato utilizzato un array di int8_t con l'idea di inserire in ogni cella dell' array 4 codifiche di caratteri. Sarà compito del server effettuare la decodifica.
La lunghezza dell'array è stabilita dalla macro BUFSIZE, ciò rende statico il messaggio. Una alternativa valida sarebbe stata utilizzare un puntatore invece che un array fixed-size e allocarlo dinamicamente in base alla grandezza della matrice.
La struttura msg_t è dichiarata nel' header myconn.h.

2.2.1) Codifica e decodifica del messaggio

con un esempio illustro come avviene la codifica e la decodifica del messaggio. Supponiamo di dover riempire una cella dell'array con i caratteri, nell'ordine {W, S, F, W}. Tramite la funzione encode( ) otteniamo:

carattere	**codifica**
encode('S')	00000000
encode('F')	00000001
encode('W')	00000010

Inizialmente ogni cella dell'array è inizializzata a 0. Per inserire i caratteri in un unico intero da 8 bit si devono fare delle bitwise operation. I bit di ciascuna cella vengono riempiti da destra verso sinistra in questo modo:

stato della cella: 00000000 |
encode('W') 00000010

stato della cella: 00000010 |
encode('S') << 2 00000000

stato della cella: 00000010 |
encode('F') << 4 00010000

stato della cella: 00010010 |
encode('W') << 6 10000000

si ottiene così lo stato finale della cella: 10010010.
La decodifica avviene secondo lo stesso principio, decodificando i bit da destra verso sinistra.

2.3) Altre strutture utilizzate

Sono state utilizzate altre strutture per rappresentare i parametri con cui venivano lanciati i thread worker, dispatcher e collector chiamate rispettivamente thWorkerArgs, thDispatcherArgs e thCollectorArgs. Questa scelta implementativa è stata fatta per facilitare la lettura del codice, vedendo la definizione di queste strutture è possibile capire su quali dati andrà ad operare ogni thread.
Inoltre la struttura coordinate_t è stata utilizzata per memorizzare i punti adiacenti a ciascuna cella allo scopo di far muovere nella giusta posizione i pesci e gli squali. É stato tenuto presente che il pianeta è una sfera.
Queste ultime strutture sono dichiarate nel' header help.h.

3) Struttura del server

Il server viene chiamato con un parametro, un dumpfile oppure NULL. Dopo aver creato la socket si mette in attesa di connessioni da parte del client. Il server è gestito con un unico thread.

L'accettazione della connessione, la ricezione del messaggio e la stampa vengono gestite in un unico loop all'interno del main() ed invocando funzioni ad hoc.
Le funzioni utilizzate sono:
1) receiveMatrixAndDump( ), riceve la matrice. Invoca al suo interno la funzione dump.
2) dump( ), decodifica il messaggio e stampa.
3) decode( ), decodifica i caratteri.
Se la connessione con il client è avvenuta con successo, il server attende la lettura di un flag t.
Se t = 1, il server capisce che è iniziata la fase di terminazione, quindi chiude la connessione ed esce dal loop. Altrimenti, si mette in attesa del messaggio vero e proprio, quindi effettua il dump della matrice decodificando opportunamente ogni cella dell'array contenuto nel messaggio.

4) Struttura del client

Il client appena attivato effettua il controllo degli argomenti che gli sono stati passati da linea di comando. Successivamente setta la maschera per bloccare i segnali SIGUSR1, SIGINT e SIGTERM. Lancia i thread worker, dispatcher e collector. attiva il processo visualizer ed entra in un loop in cui si mette in attesa della ricezione di un segnale.

4.1) Sincronizzazione tra i thread

4.1.1) Accesso alla coda

Per l'accesso in mutua esclusione alla coda dei task viene utilizzata una mutex qlock e una variabile di condizione qcond. La gestione della mutua esclusione e della sincronizzazione è mascherata all'interno delle funzioni pop( ) e push( ).

Per estrarre un task viene acquisito il lock, finché la coda è vuota il thread worker si mette in attesa sulla variabile di condizione e rilascia atomicamente il lock. Una volta riattivato e acquisito il lock, se è stato inserito un elemento dal thread dispatcher, lo estrae e rilascia il lock, altrimenti, si rimette in attesa.

4.1.2) Gestione fine chronon → nuovo chronon

Per la terminazione del chronon corrente e l'attesa dell'inizio del chronon successivo sono state usate due mutex countlock e chlock rispettivamente per accedere in mutua esclusione al contatore dei rettangoli aggiornati e alla variabile 'chronon' della struttura di simulazione.
Inoltre sono state utilizzate due variabili di condizione workers_done e new_chronon rispettivamente per mettersi in attesa del completamento di tutti i task relativi al chronon corrente e per aspettare che il chronon venga aggiornato dal thread collector.

- Il thread dispatcher, una volta inseriti tutti i task nella coda, si mette in attesa dell'inizio del nuovo chronon.
- Ogni thread worker, dopo aver finito di eseguire un task, incrementa il contatore dei rettangoli aggiornati e lo segnala al thread collector.
- Il thread collector, attende che tutti i task relativi al chronon corrente siano stati completati,

azzera il contatore dei rettangoli aggiornati e segnala l'inizio di un nuovo chronon.

4.1.3) Accesso alla matrice, aggiornamento singolo per chronon

Oltre alla matrice del pianeta è stata utilizzata una matrice di flag, della stessa dimensione del pianeta.

Per l'accesso alle matrici sono state usate due mutex: pwlock e flock rispettivamente per accedere in mutua esclusione alla matrice del pianeta e alla matrice di flag.
La matrice dei flag è inizializzata a 0.
Ogni thread worker scorre la sotto-matrice con due for annidati e controlla se la cella (i, j) corrente è sul bordo di un rettangolo oppure no. Se la cella (i, j) è già stata aggiornata, si passa all'iterazione successiva.

Il lock delle matrici viene effettuato solo se la cella (i, j) a cui vogliamo accedere si trova sul bordo di una sotto-matrice e, nel caso del pianeta, la cella deve contenere un carattere diverso da WATER.

La matrice dei flag è utilizzata in questo modo:

- se flag[i][j] = 1 → la cella è già stata aggiornata, passa all'iterazione successiva.
- se un pesce/squalo si sposta dalla posizione (i,j) alla posizione (k,l) → flag[k][l] = 1. (solo se la cella (k, l) è sul bordo viene acquisito il lock)
- se un nuovo pesce/squalo nasce nella posizione (k, l) → flag[k][l] = 1.

4.2) connessione al server

La connessione al server avviene tramite la socket invocando la funzione connectToVisualizer( ). Questa funzione, che prende come secondo parametro un flag, gioca due ruoli fondamentali:

- se il flag = 1 → viene inviato un intero con valore uguale a quello del flag e si inizia la fase di terminazione.
- Se il flag = 0 → viene invocata la funzione sendMatrix( ) che si occupa di inviare un intero con valore uguale a quello del flag e successivamente invia il messaggio vero e proprio.

4.3) fase di terminazione gentile

Per gestire la fase di terminazione è stato utilizzato un flag terminazione e una mutex lockterm per accedervi in mutua esclusione.

La fase di terminazione inizia quando il processo wator riceve un segnale SIGINT o SIGTERM.

- Il flag di terminazione viene settato a 1 → il processo wator e il thread collector escono dal loop, mentre il thread dispatcher e i thread worker rimangono in loop.
- Il wator attende la terminazione del thread collector.
- Il wator manda una richiesta di cancellazione al thread dispatcher (pthread_cond_wait è un punto di cancellazione) e attende che termini.

- Il wator manda un messaggio di End Of Stream ad ogni thread worker e attende la loro terminazione.
- Il wator si connette con il visualizer tramite la socket, invia un messaggio di End Of Stream e attende la sua terminazione. Il visualizer alla ricezione del messaggio, chiude la connessione ed esce dal loop.

5) README

Il programma viene eseguito correttamente su Ubuntu (versione 14.04) .

5.1) compilazione

per compilare il codice eseguire i seguenti comandi:

- make
  crea la cartella WATOR/lib solo se non è già presente.
- make lib
  crea la libreria libWator.a.
- make wator
- make visualizer

5.2) Vincoli

per il corretto funzionamento del programma devono essere rispettati i seguenti vincoli:

- non possono essere gestite matrici il cui numero di righe o di colonne non sono multipli di K e N. La suddivisione in rettangoli della matrice è stata pensata tenendo presente di questo attributo comune tra le matrici fornite nei test.
  Trasgredire questo vincolo porterà il programma ad accedere aree di memoria non di competenza con conseguente segmentation fault.
- L'array che corrisponde al messaggio ha dimensione fissata dalla macro BUFSIZE. Se scegliessimo una costante troppo grande invieremo byte inutilmente, viceversa una costante troppo piccola farebbe fallire le assert() nel file conn.c alle righe 97 e 137.
- per fare passare i test22 e test23 sono state commentate, nel file help.c, le righe 413 e 439 – 446. Se si tolgono i commenti è possibile visualizzare l'applicazione delle regole della simulazione