PROGETTO LABORATORIO II

LORENZO DELITALA
Matricola: 616506

Struttura del Progetto

- **bin**: Contiene gli eseguibili del progetto.
- **docs**: Contiene la documentazione del progetto.
- **config**: Contiene i file di configurazione (`bib.conf`).
- **logs**: Contiene i file di log.
- **sockets**: Contiene i file delle connessioni socket.
- **build**: Contiene file temporanei (`.o`, `.bak`, etc.), suddiviso ulteriormente in:
- **comunicazione**: File oggetto delle librerie di comunicazione.
- **my_lib**: File oggetto delle mie librerie per strutture dati (array dinamici, code, alberi binari).
- **struttura_dati**: File oggetto per l'implementazione delle strutture dati.
- **data**: Gestione dei dati, con:
- **copia originale**: Backup dei file record originali.
- **file records**: File records usati dai server.
- **include**: Tutti gli header, organizzati per categoria (comunicazione, my_lib, struttura_dati).
- **lib**: File sorgente delle librerie (.c), organizzati per categoria (comunicazione, my_lib, struttura_dati).
- **src**: Codice sorgente, inclusi script Bash utili e il codice per client e server.

Struttura Dati

La struttura dati di questo progetto è stata concepita in maniera non convenzionale, in particolare considerando il numero relativamente esiguo di libri nella biblioteca. Adottando un approccio basato sugli Alberi Binari di Ricerca (ABR), il sistema mira a garantire efficienza nella ricerca, teoricamente raggiungendo una complessità di log N, dove N rappresenta il numero totale dei libri.

Ogni libro è allocato dinamicamente in uno `struct libro` (vedere "libreria libro.h"), e il resto della struttura utilizza puntatori a questa memoria per reindirizzare le ricerche.

Il nucleo della struttura dati è un array che racchiude tutti i possibili campi. A ogni campo è associato un ABR in cui i libri sono organizzati in ordine lessicografico per valore (ogni nodo dell'albero contiene il valore e il puntatore al libro).

Viene inoltre mantenuto un array di puntatori ai libri, dove vengono salvati tutti i puntatori, utilizzato alla fine del programma per deallocare correttamente tutta

la memoria.

Per trovare libri basati su un attributo specifico, si accede all'ABR corrispondente tramite l'array dei campi e si procede alla ricerca dei libri che corrispondono al valore desiderato. Sebbene l'obiettivo sia una ricerca con complessità log N, si devono considerare alcune osservazioni: 1) La ricerca effettiva potrebbe non raggiungere log N. Se T è il numero di libri che corrispondono alla richiesta, allora il miglior risultato possibile si traduce in max{log N, T} operazioni, moltiplicato per il numero di campi specificati nella ricerca. 2) Questa struttura richiede 3,7 volte più memoria rispetto a un semplice array di stringhe che elenca i libri. 3) Il tempo necessario per costruire questa struttura dati è log N volte superiore a quello richiesto da una struttura lineare.

Importante è anche la gestione della sincronizzazione in ambienti multi-thread: invece di controllare l'accesso all'intera struttura dati, si gestisce l'accesso al singolo libro tramite la variabile `lib_inUso` dello struct libro. In questo modo, i thread possono lavorare contemporaneamente su libri diversi.

In generale, all'inizio del progetto ero entusiasta all'idea di sviluppare una struttura dati particolare ed efficiente. Tuttavia, dopo averci lavorato, ho compreso che la semplicità rappresenta la vera forza di un programmatore. Gli svantaggi di questa struttura dati potrebbero superare i vantaggi. Inoltre, ho probabilmente reso il compito di comprensione più arduo per lei, professoressa, che già deve valutare molti progetti. Per questo, mi scuso sinceramente.

Librerie

my_lib

Queste librerie sono librerie che uso abitualmente per implementare strutture dati o funzioni classiche.

- **dynamic_array.h:** libreria che implementa funzioni per gestire un array di dimensione variabile tramite uno `struct dynamic_array` definito dalla libreria.
- **binary_tree.h:** libreria che implementa un albero binario ordinato. Può essere utilizzata sia tramite la macro struttura `struct binary_tree` (utilizzando le funzioni `bt_`) che tramite i singoli nodi definiti come `struct binary_tree_node` (utilizzando le funzioni `bt_node_`)
- **static_fifo.h:** libreria che implementa una coda statica. Sfrutta la libreria dynamic_array.h per farlo.
- **thread_shared_static_fifo.h:** libreria che aggiunge alla libreria static_fifo le funzioni put e get che hanno la caratteristica di essere thread safe.
- **readers_writers2.h:** libreria che offre una implementazione inter-processo della soluzione al classico problema di sincronizzazione tra lettori

e scrittori, seconda variante (ovvero con scrittori che hanno precedenza sui lettori).

comunicazione

- **protocollo_comunicazione.h:** questo non ha associato nessun file.c perché non contiene funzioni. Semplicemente qui sono salvati i define dei messaggi tra client e server e dei path alle varie directories. In questo modo se si vuole cambiare la struttura del programma basta aggiornare questa libreria e verranno aggiornati contemporaneamente i programmi client e server. Inoltre viene qui definito lo `struct messaggio` contenente un `char type`, un `int32_t length` ed un `char* data`, secondo le specifiche richieste dal progetto per la comunicazione client server
- **bib_conf.h**: questa libreria usa la libreria `readers_writers2.h` per fornire funzioni che operino sul file bib.conf in modo inter-process safe.
- **coda_condivisa.h:** libreria che usa la libreria `thread_shared_static_fifo.h` e `protocollo_comunicazione.h` per implementare una coda composta da elementi che contengono uno `struct messaggio` ed il `client_fd` del socket del client che lo ha mandato.
- **socket_comunication.h:** per non fare confusione tra il lato server ed il lato client del protocollo di comunicazione ho preferito includerli entrambi in una libreria. Questa libreria implementa quindi le funzioni che permettono al server e al client di comunicare tramite socket.

struttura dati

- **personal_time.h:** l'obbiettivo di questa libreria è lavorare con il valore del prestito dei libri. Per fare ciò usa lo `struct tm` e fornisce funzioni che trasformano una stringa in `tm`, un `tm` in una stringa, che calcolano la differenza tra due date e che diano la data corrente.
- **libro.h:** questa libreria serve ad implementare e gestire la struttura del singolo libro. Per quanto riguarda l'accesso di lettura/scrittura di un singolo libro, usa una logica in cui si controlla anatomicamente se il libro è in uso e, se lo è, il thread corrente viene messo in attesa su una condizione e si sbloccherà solo a tempo debito.
- **arrayCampi.h:** libreria che implementa il cuore della struttura dati, ovvero la mappatura dei libri per campo e valore. Utilizza le struttura `struct campoAlbero` per associare un nome di campo a un albero binario di ricerca che organizza i valori specifici per quel campo e `struct valoreLibro` per collegare un valore di un campo a un libro specifico. Semplifica il lavoro di gestione della struttura in 3 funzioni finali, utilizzate da `struttura_dati.h`: arrCampi_aggiungiLibro(), arrCampi_generaLista(), arrCampi_free().

- **struttura_dati.h:** questa libreria sfrutta quelle precedenti per fornire al server quattro semplici funzioni per la gestione della struttura dati: una per generarla, una per cercare libri, una per aggiornare il file record ed una per deallocarla.

Problemini di allocazione e puntatori

Nel mio progetto, mi sono scontrato con un problema di puntatori molto elegante, perciò lo voglio raccontare. Inizialmente, per semplificare, avevo raggruppato tutti i libri in un unico array, con ciascun nodo informativo che puntava all'indice corrispondente dell'array. Tuttavia, durante i test, ho notato che le modifiche ai libri non persistevano dopo l'aggiornamento del file di record. Il problema era che, con l'inserimento di nuovi libri, l'array dinamico veniva riallocato in una nuova posizione di memoria, rendendo obsoleti i puntatori esistenti nei nodi informativi, che ancora indicavano la vecchia posizione.

La soluzione adottata è stata di allocare ogni libro separatamente e di utilizzare un array di puntatori ai libri anziché un array di libri stessi. Questo approccio ha impedito il problema della riallocazione, mantenendo i puntatori sempre validi, indipendentemente dall'aggiunta di nuovi libri. Tale strategia ha non solo risolto il problema specifico, ma ha anche aumentato la flessibilità e la scalabilità del sistema, facilitando la gestione dei dati e minimizzando il rischio di errori di puntatori.

README

Per compilare il programma la prima volta che si scarica il progetto, basterà semplicemente aprire la cartella contenente il progetto sul terminale e lanciare il comando make. A quel punto verranno compilati tutti i file oggetto delle librerie nella cartella build, successivamente verranno creati nella cartella bin l'eseguibie bibserver, poi bibclient ed infine verrà copiato il file bash di bibaccess.

Gli altri comandi del makefile sono: * **make clean**: pulisce la cartella di lavoro build

- **make clean_all**: ripristina il progetto allo stato originale, ovvero con i file record uguali a quelli originali, le cartelle logs e sockets vuote, il file bib.conf vuoto e la cartella bin vuota.
- **make test**: esegue il test dei programmi come richiesto. Ho lasciato l'output dei client sul terminale, come mi è sembrato di capire fosse richiesto dal progetto, anche se esce un risultato un po' confuso e l'unico vero modo per capire che è andato tutto bene è leggere il risultato dei file di log.
- **make test_valgrind**: esegue test_clean ma aggiugne valgrind per controllare che non ci siano leak di memoria

Per quanto riguarda l'evocazione dei singoli eseguibili è uguale a come stabilito dalla richiesta di progetto, anche se ovviamente andranno chiamati dalla cartella bin/. Per semplicità li riscrivo qui

./bin/bibserver nome_bib file_record W ./bin/bibclient –campo1=”valore1” ... –campoN=”valoreN” [-p] ./bibaccess –query (o –loan) file1.log file2.log ...