BIBLIO

Relazione del Progetto di Laboratorio 2 - AA 2022-2023

Tommaso Mangiavacchi — mat: 621396

Strutture dati

Le principali strutture dati adottate per la realizzazione del progetto sono:

- **Book_t**
- **Arg_t**
- **NodoAutore**
- **Linked List**
- **Unboundedqueue**

Book_t

Un oggetto di tipo **Book_t** contiene i campi necessari per descrivere un libro. La maggior parte dei campi sono di tipo stringa, ad eccezione di "anno", che è rappresentato da un numero intero, e "autore", che è un riferimento alla testa della lista degli autori.
La lista degli autori di un libro è implementata utilizzando il tipo NodoAutore.

```
struct
{
    char* titolo;
    char* editore;
    int anno;
    char* nota;
    char* collocazione;
    char* luogo_pubblicazione;
    char* descrizione_fisica;
    char* volume;
    char* scaffale;
    char prestito[19];
    struct z* autore;
}typedef Book_t;
```

NodoAutore

Questo tipo è utilizzato per incapsulare una stringa che rappresenta il nome di un autore e il riferimento all'elemento successivo della lista degli autori.

```
struct z{
    char* val;
    struct z* next;
};
typedef struct z NodoAutore;
```

Arg_t

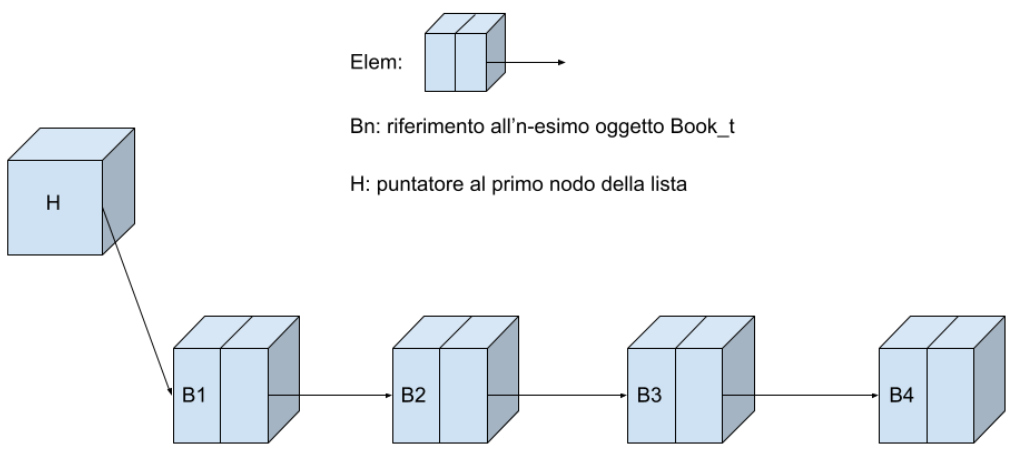
È un tipo usato per immagazzinare riferimenti a strutture dati, file e mutex.
Viene definita una sola istanza di un oggetto di tipo arg_t e il riferimento di questa istanza è passato come argomento nella creazione dei vari thread Worker.

```
typedef struct {
  Queue_t* q;
    Elem* list;
    pthread_mutex_t* mutex;
    FILE* flog;
}
arg_t;
```

Linked List

Rappresenta la principale struttura dati utilizzata dal server per immagazzinare le informazioni dei record. È realizzata mediante una struct simile a quella di NodoAutore, ma si differenzia per il tipo del valore del nodo, che in questo caso è un riferimento a un oggetto libro di tipo Book_t.

```
struct k{
    Book_t* val;
    struct k* next;
};
typedef struct k Elem;
```



UnboundedQueue

È una coda di priorità, in cui il server aggiunge riferimenti ai descrittori di file che rappresentano le socket dei client con i quali il server ha stabilito una connessione.

L'utilizzo di una coda ci permette di implementare un paradigma Produttore-Consumatore, dove il main thread(produttore) inserisce i file descriptor dei client da gestire e i thread worker (consumatori) consumano elementi dalla coda.

Le principali operazioni messe a disposizione dalla coda sono:

- **initQueue( ):** crea e restituisce il riferimento alla coda
- **push( ):** aggiunge un elemento alla coda
- **pop( ):** elimina un elemento dalla coda
- **deleteQueue( ):** dato un riferimento ad una coda, si occupa di deallocare la struttura dati

Algoritmi

Sia il client che il server hanno due operazioni in comune:

- **freeBook( ):** preso in input il riferimento ad un libro Book_t, questa funzione si occupa di deallocare la memoria utilizzata per la sua memorizzazione. In pratica, scorre ogni campo del tipo Book_t ed esegue la funzione free( ), restituendo così la memoria alla macchina.
- **bookToRecord( ):** preso in input il riferimento ad un Book_t e una stringa buffer, estrae le informazioni dal oggetto e le scrive nel buffer secondo il formato “nomecampo: valorecampo;”

Di seguito sono invece elencate le operazioni che sono state implementate dal server:

- **recordToBook( ):** Questa funzione prende in input una stringa che rappresenta un record e un riferimento ad un Book_t vuoto. Prima di tutto, la stringa viene suddivisa in token utilizzando i caratteri delimitatori. Successivamente, i dati vengono estratti dal record e vengono allocati spazi di memoria per i campi del Book_t. Infine, i dati estratti vengono copiati sui rispettivi campi del Book_t.
- **matchBook( ):** Riceve in input due riferimenti a oggetti di tipo Book_t, confronta tutti i campi non nulli dei libri eccetto "autore" e, se l'operazione **equalAuthors( )** restituisce un esito positivo, restituisce True se contengono gli stessi dati.
- **equalAuthors( ):** prese due liste di autori, restituisce True se la seconda lista contiene almeno tutti gli autori della prima
- **isPresent( ):** prende il riferimento a un libro e alla testa della lista e scorre l'intera lista alla ricerca del libro. Se lo si trova, si aggiornano i campi vuoti del libro già presente nella lista con i campi che sono invece presenti nel libro di cui si è verificata la presenza.
- **aggiornaMax( ):** prende un fd_set e un riferimento al massimo, aggiorna il valore massimo scorrendo il fd_set.
- **countAttributes( ):** preso un record, restituisce il numero di campi
- **addBibToConf( ):** Aggiunge il sockname del server al file *bib.conf*. Se è già presente da errore e termina l'esecuzione del server; altrimenti, aggiunge il sockname.
- **deleteFromConf( ):** Prima della terminazione del server, viene invocata questa funzione per cercare la riga contenente le informazioni sul sockname del server e invalidare la riga del file.
- **dumpRecord( ):** Prima di terminare il server, scorre la lista concatenata che contiene i libri raccolti dai record, chiama bookToRecord() e scrive i risultati aggiornati nel file_record.
- **Worker( ):** Il programma rimane bloccato sulla chiamata pop() in attesa che il thread principale faccia push() dei file descriptor dei client che hanno scritto qualcosa.
  Quando la pop() si sblocca, restituisce il riferimento al socket di un client e successivamente effettua un controllo del valore del file descriptor (fd).
  - se fd=-1 allora termina l’esecuzione del worker con la chiamata pthread_exit( )
  - Altrimenti, esegue 3 chiamate read() per leggere i campi del messaggio: type, length e data.
    Dopo aver letto il messaggio inviato dal client, scorre la lista concatenata del server fino a trovare i libri che hanno i valori specificati dal client. Successivamente, li converte in una stringa e invia il messaggio al client tramite la write().
    Nel caso **type='L'** fa tutto quello che è stato descritto in precedenza, ma effettua anche dei controlli sulla disponibilità del libro. Nel caso l'esito sia positivo, procede sovrascrivendo il campo *prestito* del libro contenuto nella lista e comunica tutto al client.
- **gestore()**: è il gestore dei segnali SIGINT e SIGTERM. La sua unica funzione è impostare il valore di una variabile volatile ***sigflag*** a -1.

Struttura del codice

Al di la del codice del codice di bibserver.c e bibclient.c:

- **freeBook.c**: contiene il codice della funzione freeBook( )
- **bookToRecord.c**: contiene il codice della funzione bookToRecord( )

- **unboundedqueue.c**: contiene il codice delle funzioni della coda

e freeBook.h, bookToRecord.h, unboundedqueue.h contengono le dichiarazioni delle loro rispettive funzioni.
Invece:

- **structures.h**: è un file header contenente le struct ad uso comune di tutti i file.c elencati in precedenza.
  Dentro sono presenti le definzioni di Book_t, NodoAutore, Elem, Arg_t, inoltre sono incluse le librerie comuni di tutti i file.c.
  Un esempio è **#include<string.h>** che è necessaria sia a bibclient che a bibserver.
- **util.h**: è un file di supporto che contiene macro, utilizzate da unboundedqueue.h
- **bibaccess.sh**: è lo script che fa una analisi sui file di log generati dai server
- **testclient.sh**: è uno script che lancia 40 bibclient
- Makefile: insieme di regole che semplificano compilazione, pulizia e testing

---

Esecuzione lato Server

- viene creata una sigmask e si mascherano **tutti** i segnali
- inizio della sezione critica(l'esecuzione non può interrompersi per l'arrivo di segnali)
- installa il gestore dei segnali SIGINT e SIGTERM
- controlla i parametri passati all'avvio
- esegue **addBibToConf( )** per rendersi visibile ai client
- apre il file record e scorre tutte le righe, per ogni riga viene creato un oggetto Book_t e un nodo della lista che lo incapsulerà, se il libro non è già presente nella lista allora viene inserito, l'inserimento è effettuato sempre in testa
- dopo aver estratto le informazioni dal file record, crea una socket orientata alla connessione con dominio **AF_UNIX** per accogliere i client che cercano di connettersi.
  La scelta del dominio dipende dall'ipotesi che l'esecuzione sia effettuata sulla stessa macchina, sia per il client che per il server.
- associa la socket all'indirizzo che nel nostro caso, poiché la socket appartiene al dominio AF_UNIX, sarà rappresentato dal nome della biblioteca.
- esegue la funzione **listen**(welcomeSocket, SOMAXCONN) per mettere la socket in ascolto dei client. Il limite massimo di richieste accodabili è SOMAXCONN, il cui valore dipende dalla macchina. Di default, è impostato a 128.
- inizializza la coda che servirà a immagazzinare i riferimenti alle socket dei client
- avvia un pool di thread con un numero di thread pari all'argomento "W".
  I thread non sono di tipo **DETACHED** perché ciò ci dà la garanzia che, quando il server termina la sua esecuzione, non ci sono thread dietro le quinte che non abbiano terminato di gestire i client.
- crea e inizializza due fd_set: uno rappresenta l'insieme dei file descriptor dei client di cui è stato fatto l'accept( ), mentre l'altro rappresenta i file descriptor dei client che hanno scritto qualcosa.
- fine della sezione critica
- imposta la maschera dei segnali per far passare solo i segnali **SIGINT** e **SIGTERM**
- inzio del ciclo while
- si blocca sulla chiamata **pselect( )**
- appena un cliente prova a connettersi, la **pselect( )** si sblocca e la connessione viene accettata.
- quando un client scrive per la prima volta, la pselect si sblocca e il client viene inserito nella coda e rimosso dal set di file descriptor di cui teniamo traccia
- uno dei thread che era rimasto in attesa sulla chiamata **pop( )** riceve il file descriptor del client e gestisce il messaggio del client.
- si cicla finché non arriva un segnale di tipo **SIGINT** o **SIGTERM**, che verrà gestito dal gestore
- si interrompe il ciclo while e il server inizia i preparativi per lo spegnimento inserendo dei valori "-1" nella coda.
- i thread, alla ricezione del file descriptor con valore "-1", eseguono **pthread_exit( )**.

- il main thread(server) esegue la pthread_join( ) di tutti i thread
- finito il join, il main thread esegue le istruzioni **deleteFromConf( )** per togliersi dal file *bib.conf* e **dumpRecord( )** per aggiornare il file record con i nuovi record
- chiude il file, dealloca le variabili, pulisce la memoria e termina l'esecuzione.

Esecuzione lato Client

- viene impostato il campo type ad un valore di default “Q”
- vengono letti gli argomenti passati al client e ne viene verificato il formato
- costruisce il messaggio da inviare ai server
- viene aperto il file *bib.conf*
- se non c’è nessun errore nell’apertura si scorre il file
- estrae l’indirizzo del server dalla riga corrente
- si tenta la connessione con il server
- avvenuta la connessione, si esegue la write( ) dell’intero messaggio sul file descriptor del server
- legge la risposta del server e la scrive in stdout
- chiude la socket, passa alla riga successiva del file *bib.conf* e prosegue a interrogare il prossimo server
- termina quando arriva alla fine del file di configurazione

Problematiche incontrate

I punti che hanno richiesto particolare attenzione sono stati:

- gestione delle stringhe in generale: copia , analisi, concatenazione, tokenizzazione, gestione del carattere di fine stringa
- confronto tra liste di autori
- invalidazione del record di *bib.conf* contenente l’indirizzo del server

---

Gestione della memoria

Il software Valgrind riporta l’assenza di alcuna forma di memory leak, nel client come anche nel server.
Per quanto riguarda invece l’ERROR SUMMARY, ci sono solo errori del tipo “conditional jump or move depends on uninitialised value(s)” che non ho considerato rilevanti ai fini della solidità del progetto.

```
==1828==
==1828== HEAP SUMMARY:
==1828==     in use at exit: 0 bytes in 0 blocks
==1828==   total heap usage: 760 allocs, 760 frees, 71,494 bytes allocated
==1828==
==1828== All heap blocks were freed -- no leaks are possible
==1828==
==1828== For lists of detected and suppressed errors, rerun with: -s
==1828== ERROR SUMMARY: 56 errors from 6 contexts (suppressed: 0 from 0)
> 
```

README

Per compilare il progetto:

```
> make all
```

Per l’esecuzione del test:

```
> make test
```

Per la pulizia della directory dai file eseguibili e di log:

```
> make clean
```

Per compilare ed eseguire il server:

```
> gcc bibserver.c unboundedqueue.c bookToRecord.c freeBook.c -o bibserver -lpthread
> ./bibserver name_bib file_record W
```

Per compilare ed eseguire il client:

```
> gcc bibclient.c bookToRecord.c freeBook.c -o bibclient
> ./bibclient --author="ciccio" --title="pippo"
```