

# ΑΝΑΦΟΡΑ ΠΑΡΑΔΟΣΗΣ ΠΡΩΤΗΣ ΕΡΓΑΣΤΗΡΙΑΚΗΣ ΑΣΚΗΣΗΣ ΣΤΑ ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΟΝΟΜΑΤΕΠΩΝΥΜΟ: Στεργίου Βασίλειος  
ΑΜ :4300

## Περιγραφή του στόχου της Εργαστηριακής Άσκησης

Στη παρούσα εργαστηριακή άσκηση, το βασικό ζητούμενο που ζητείται να υλοποιηθεί συνιστά η χρήση του πολυνηματισμού, μέσω των εργαλείων που προσφέρει η βιβλιοθήκη pthread της γλώσσας C, ώστε να επιτευχθεί παράλληλη εγγραφή και αναζήτηση δεδομένων σε μια δοθείσα βάση δεδομένων.

Οι περιορισμοί της παρούσας εργαστηριακής άσκησης είναι δύο και αφορούν τον ορθό συγχρονισμό των νημάτων:

➤ **Περιορισμός 1:** Σχετικά με τα νήματα γραφείς, **μόνο ένα νήμα γραφέας** δύναται να εισάγει κάποια εγγραφή κάθε φορά, ενώ παράλληλα **δεν επιτρέπεται** να εισάγουν εγγραφές τα υπόλοιπα νήματα γραφείς ή να γίνεται αναζήτηση από από τα νήματα αναγνώστες.

➤ **Περιορισμός 2:** Σχετικά με τα **νήματα αναγνώστες**, τα νήματα αυτά μπορούν να κάνουν **παράλληλη αναζήτηση** ενός ζεύγους κλειδιού- τιμής, υπό την προϋπόθεση ότι **δεν υπάρχει** την ίδια στιγμή κάποιο **νήμα γραφέας** που να πραγματοποιεί εισαγωγή τιμής στη βάση δεδομένων.

Παράλληλα με τα δύο βασικές απαιτήσεις της εργαστηριακής άσκησης, ζητείται και η **κατανομή των νημάτων** σε νήματα αναγνώστες και νήματα γραφείς, καθώς και ο **διαμοιρασμός των αιτήσεων** στα νήματα αυτά

Για τις ανάγκες της πολυνηματικής υλοποίησης της επεξεργασίας της βάσης δεδομένων, έχουν τροποποιηθεί κατάλληλα τα ακόλουθα αρχεία, όπου για το καθένα θα αναφερθεί αναλυτικά ο κώδικας κάθε μεταβλητής, συνάρτησης ή macro που έχει προστεθεί ή τροποποιηθεί σε καθένα από τα αρχεία αυτά:

- bench.h
- bench.c
- kiwi.c
- db.c

# APXEIO bench.h

Στο αρχείο **bench.h** έχουν προστεθεί μεταβλητές που αφορούν το διαμοιρασμό των αιτήσεων στα νήματα, καθώς και συναρτήσεις που είναι υπεύθυνες για την εκτέλεση των διαφορετικών λειτουργιών που πρέπει να υποστηρίζονται από την υλοποίηση.

Αρχικά, θα γίνει αναλυτική αναφορά στις χρησιμοποιούμενες μεταβλητές και στα macros τα οποία έχουν οριστεί στο αρχείο αυτό.

**NUMBER\_OF\_THREADS**: Το συγκεκριμένο macro δηλώνει τον αριθμό των νημάτων (**threads**) που θα υποστηρίζει η υλοποίηση.

Ο χρήστης έχει τη **δυνατότητα να μεταβάλει τη παράμετρο** αυτή και κατά συνέπεια να επηρεάζεται το πλήθος των αιτήσεων που θα πραγματοποιείται από το κάθε νήμα.

**DATAS:** Το macro αυτό αρχικά χρησιμοποιούνταν στο αρχείο `kiwi.c`, καθώς για κάθε αίτηση εγγραφής ή ανάγνωσης άνοιγε πολλαπλές φορές η βάση δεδομένων. Εφόσον, πλέον, το άνοιγμα και κλείσιμο της βάσης πραγματοποιείται στο αρχείο `bench.c`, το macro αυτό έχει προστεθεί στο αρχείο `bench.h` προκειμένου να αναγνωρίζεται από το κώδικα του αρχείου `bench.c`.

Στη **παρούσα υλοποίηση**, η βάση πρέπει να ανοίγει μόνο μία φορά από το εκάστοτε νήμα, προκειμένου να αποφευχθεί ανεπιθύμητη συμπεριφορά που μπορεί να οδηγήσει σε **Segmentation Fault**, όπως ένα νήμα να πραγματοποιήσει εγγραφή ή ανάγνωση στη βάση, ενώ ένα νήμα ενδέχεται να την έχει κλείσει νωρίτερα.

Το **άνοιγμα και κλείσιμο** της βάσης δεδομένων πραγματοποιείται στο αρχείο **bench.c**. Αναλυτικότερες λεπτομέρειες θα επισημανθούν κατά την επεξήγηση του κώδικα του αρχείου bench.c.

**write\_percentage:** Η μεταβλητή αυτή χρησιμοποιείται για τον καθορισμό του ποσοστού των νημάτων που θα αποτελούν τα **νήματα γραφείς** και, κατά συνέπεια, και των **νημάτων αναγνοστών**, καθώς και των αιτήσεων που θα αποτελούν τα writes και τα reads από το πλήθος όλων των αιτήσεων. Η τιμή της αποτελεί **input argument** που δίδεται από το χρήστη από τη γραμμή εντολών (**terminal**) και χρησιμοποιείται στο αρχείο bench.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>

// In this header file, some extra macros, variables and function prototypes are define
// as there used in bench.c, kiwi.c and db.c files

// The number of threads that the system uses
// Its use is in bench.c file to distribute the requests on the threads
// and execute the multithreaded reading-writing process
#define NUMBER_OF_THREADS (250)
#define DATAS ("testdb")

#define KSIZE (16)
#define VSIZE (1000)

#define LINE "+-----+-----+-----+-----+\n"
#define LINE1 "-----\n"

// The write percentage that the user gives as an
// input argument
// It is used in bench.c file
extern float write_percentage;

// The total number of write requests that
// the writer threads will perform
// It is used in bench.c file
extern int writes_per_thread;
extern int reads_per_thread;

// The total writing execution time
// for the writer threads to perform all
// write requests
// It is initialized in bench.c and kiwi.c files and incremented in kiwi.c file
extern double total_write_cost;

// The total reading execution time
// for the threads to perform all
// read requests
// It is used in bench.c file
// It is initialized in bench.c and kiwi.c files and incremented in kiwi.c file
extern double total_read_cost;
```

**writes\_per\_thread/reads\_per\_thread**: Οι μεταβλητές αυτές χρησιμοποιούνται για το **διαμοιρασμό** των writes και reads στα **νήματα γραφείς** και **αναγνώστες** αντίστοιχα, αφού πρώτα έχει καθοριστεί πόες από τις αιτήσεις αυτές επρόκειτο για writes και πόες από αυτές για reads. Οι τιμές τους ανατίθενται στο αρχείο **bench.c**, όπου πραγματοποιούνται όλες οι λειτουργίες που αφορούν τα νήματα και τον διαμοιρασμό των αιτήσεων.

**total\_write\_cost/total\_read\_cost**: Οι μεταβλητές αυτές χρησιμοποιούνται για τον υπολογισμό του συνολικού χρόνου που απαιτείται για τα reads και τα writes, αλλά και για τη γενίκευση των στατιστικών που υπολογίζονται στο αρχείο kiwi.c.

Οι μεταβλητές αυτές **αρχικοποιούνται** τόσο στο αρχείο **bench.c** όσο και στο αρχείο **kiwi.c**.

Στο αρχείο kiwi.c υπολογίζεται η τελική τιμή τους και στο αρχείο bench.c η τιμές αυτές χρησιμοποιούνται για να γίνουν print στο χρήστη.

Το γεγονός αυτό το επιτρέπει η **δήλωση των μεταβλητών αυτών στο αρχείο bench.h**, το οποίο γίνεται include και στα δύο αρχεία .c

**thread\_write\_func () /thread\_read\_func ()**: Οι συναρτήσεις αυτές χρησιμοποιούνται ως ορίσματα στη κλήση **pthread\_create()** του λειτουργικού συστήματος, με την οποία γίνεται δημιουργία των νημάτων. Οι συναρτήσεις αυτές εκτελούν τις συναρτήσεις **write\_test()** και **read\_test()** αντίστοιχα.

**execute\_reads\_writes() / execute\_reads\_only () / execute\_writes\_only ()**: Οι συναρτήσεις αυτές εκτελούν την πολυνηματική υλοποίηση πολλαπλών αιτήσεων reads-writes, καθώς και είτε μόνο αιτήσεων reads είτε μόνο αιτήσεων writes.

**initialise\_threads()**: Η συνάρτησης αυτή αποτελεί την αρχή της πολυνηματικής υλοποίησης. Στη συνάρτηση αυτή γίνεται ο διαμοιρασμός των αιτήσεων στα νήματα και , ανάλογα με τη τιμή της μεταβλητής write\_percentage, εκτελείται η αντίστοιχη συνάντηση για την λειτουργία που επιθυμεί να εκτελέσει ο χρήστης.

```
// The function that is used by the writer threads to perform
// the multithreaded writing process
// its argument is NULL, which is passed on pthread_create() system call
// It is used in bench.c file
void * thread_write_func(void * arg);

// The function that is used by the reader threads to perform
// the multithreaded reading process
// its argument is NULL, which is passed on pthread_create() system call
// It is used in bench.c file
void * thread_read_func(void * arg);

// The function that initiates the read-write operation
// It is used in bench.c file
int execute_reads_writes();

// The function that initiates the read operation
// It is used in bench.c file
int execute_reads_only();

// The function that initiates the write operation
// It is used in bench.c file
int execute_writes_only();

// The function that initiates the respective operation (read,write,read-write)
// depending on the mode that the user gives as an input
// its argument is the write percentage that the user gives as input
// and the mode that the user will run (read,write,read-write)
// It is used in bench.c file
int initialise_threads(int write_percentage);
```

## APXEIO bench.c

Στο αρχείο **bench.c** υλοποιούνται οι συναρτήσεις που περιγράφηκαν παραπάνω, οι οποίες έχουν ως σκοπό την **πραγματοποίηση της πολυνηματικής υλοποίησης**. Μέσω του αρχείου bench.c, ουσιαστικά, πραγματοποιούνται τα εξής:

- Καθορισμός της **λειτουργίας** που θα εκτελέσει ο χρήστης
- Καθορισμός των **νημάτων γραφένων** και των **νημάτων αναγνωστών** και ανάθεση των αιτήσεων read/write στα νήματα
- **Εκτέλεση** της πολυνηματικής λειτουργίας που επιθυμεί ο χρήστης

Στο σημείο αυτό θα γίνει αναφορά στις βιβλιοθήκες που προστέθηκαν στο αρχείο bench.c, αλλά και στις μεταβλητές που **προστέθηκαν ή τον οποίων άλλαξε η τοπικότητα τους**, για να επιτευχθεί η πολυνηματική υλοποίηση

**bench.h:** το αρχείο bench.h πλέον περιλαμβάνει όλες τις συναρτήσεις, μεταβλητές και macro που χρησιμοποιούνται στο αρχείο bench.h, τον οποίων ο ρόλος περιγράφηκε ωρίτερα.

**pthread.h:** Η βιβλιοθήκη αυτή είναι απαραίτητη ώστε να μπορούν να γίνουν κλήσεις τους λειτουργικού συστήματος, όπως η **pthread\_create()** και η **pthread\_join()**, ώστε να μπορεί να στηριχθεί η πολυνηματική υλοποίηση.

**math.h:** Για τον ακριβή υπολογισμό των αιτήσεων write και reads που θα πραγματοποιηθούν, βάσει του ποσοστού (write\_percentage) που δίνεται από τον χρήστη, έχει χρησιμοποιηθεί η συνάρτηση round() που απαιτεί την βιβλιοθήκη math.h

```
#include "../engine/db.h" / #include "../engine/variant.h" :
```

Τα αρχεία db.h, variant.h περιέχουν απαραίτητες συναρτήσεις για το **άνοιγμα και κλείσιμο** της βάσης δεδομένων, η οποία πλέον χρησιμοποιείται στο αρχείο bench.c αντί του kiwi.c, καθώς πλέον απαιτείται να ανοίγει και κλείνει **μόνο μία φορά** κατά την εκτέλεση της εφαρμογής

```
// In bench.h header file, there are some additions in terms of
// functions and variables
// Extensive details are given on the report
// Right now, there will be described the way the functions
// and variables are used

#include "bench.h"

// the pthread library is used so that multithreading can be achieved
#include <pthread.h>

// the math library is included, as the round() function is used
// in the write/read requests distribution in the threads
#include <math.h>

// In order to implement the readers - writer algorithm
// some functions from other files are required on bench.c file
// example given, the _read_test(), _read_test() functions
// and the functions that are used to open and close the DB
// (db_add() , db_close())

#include "../engine/db.h"
#include "../engine/variant.h"
```

Οι μεταβλητές **r**, **count** αρχικά ήταν **τοπικές** μεταβλητές στη συνάρτηση **main()** του αρχείου bench.c. Οι μεταβλητές αυτές, πλέον, χρησιμοποιούνται στις συναρτήσεις **thread\_read\_func()** και **thread\_write\_func()**.

Αλλάζοντας την εμβέλειά τους (**scope**) σε καθολικές (**global**), δεν είναι απαραίτητη η χρήση struct για να μεταβιβάζονται ως ορίσματα στις συναρτήσεις που καλούνται από τα νήματα. Επιπλέον, οι τιμές τους υπολογίζονται **πριν χρησιμοποιηθούν** από τα νήματα και **δεν μεταβάλλονται** με τον οποιοδήποτε τρόπο από αυτά, επομένως δεν αποτελούν κοινόχρηστους πόρους στους οποίους πρέπει να δοθεί προσοχή στις τιμές που τους ανατίθενται.

```
// At this point, count, r variables were set as globals
// Originally, they were local variables in the main() function
// Those variables are used by the thread functions so they were
// set as globals so they are recognised by the threads

long int count;
int r;

// Depending on the write percentage the user gives as input
// there is also a specific number of read/write operations
// that will be executed by the threads
```

Οι μεταβλητές που έχουν δηλωθεί στο αρχείο bench.h χρησιμοποιούνται στο αρχείο bench.c ως καθολικές (global) μεταβλητές, με σκοπό τον διαμοιρασμό των νημάτων σε νήματα γραφείς και νήματα αναγνώστες, τον καθορισμό των αιτήσεων που θα αποτελούν αιτήσεις εγγραφής (writes) και αιτήσεις ανάγνωσης (reads), αλλά και τον διαμοιρασμό των αιτήσεων αυτών στα νήματα που αντιστοιχούν.

Η χρήση των μεταβλητών αυτών επισημάνθηκε στο σημείο που αναλύθηκε ο ρόλος των μεταβλητών που έχουν δηλωθεί στο αρχείο bench.h

```
// At this point, count, r variables were set as globals
// Originally, they were local variables in the main() function
// Those variables are used by the thread functions so they were
// set as globals so they are recognised by the threads

long int count;
int r;

// Depending on the write percentage the user gives as input
// there is also a specific number of read/write operations
// that will be executed by the threads

long int write_count;
long int read_count;

// The variable that stores the write percentage the user gives as input
float write_percentage;

//The variable that stores the read percentage
int read_percentage;

// The variables that determine how many reads and writes
// the threads will execute
int writes_per_thread;
int reads_per_thread;

// The variables that store the total costs of executing the read/write operations
double total_write_cost;
double total_read_cost;

// The DB was originally used in the kiwi.c file
// It is now needed on the bench.c file for the purpose
// of Multithreading
DB* db;

// The variables that determine how many reader and writer
// threads will be used, depending on the write percentage
int reading_threads;
int writing_threads;
```

Στη παρούσα φάση της αναφοράς θα γίνει διεξοδική περιγραφή του κώδικα που προστέθηκε ή τροποποιήθηκε στο **αρχείο bench.c** για τις ανάγκες της πολυνηματικής επεξεργασίας της βάσης δεδομένων, ξεκινώντας από τη συνάρτηση **main()** και μεταβαίνοντας **σταδιακά** σε μεγαλύτερο βάθος, ακολουθώντας τη **πορεία των συναρτησιακών κλήσεων**.

Στη συνάρτηση **main()**, οι μεταβλητές **count**, **r** είχαν αρχικά τοπική εμβέλεια και ήταν όρισμα στις συναρτήσεις **\_read\_test()** και **\_write\_test()**.

Οι συναρτήσεις αυτές πλέον χρησιμοποιούνται από τα νήματα και για το λόγο αυτό έχει αλλάξει η εμβέλεια τους σε καθολική για να υπολογιστούν πρώτα οι τιμές τους στη συνάρτηση **main()** και ύστερα να χρησιμοποιηθούν από τις συναρτήσεις που καλούνται από τη συνάρτηση **initialise\_threads()**.

Τα ορίσματα που δέχεται πλέον ως **input arguments** από τη γραμμή εντολών (**terminal**) η συνάρτηση είναι είτε 4 είτε 5, καθώς πλέον ως **input argument** εισάγεται και το **write\_percentage** και ως πέμπτο όρισμα μπορεί να εισαχθεί μία τυχαία τιμή, η οποία υποδηλώνει ότι η τιμή της μεταβλητής **r** θα είναι 1. Ο έλεγχος για τον ορθό πλήθος ορισμάτων έχει αλλαχθεί ώστε να εξετάζεται αν τα ορίσματα είναι τουλάχιστον 4, αντί για 3 που ήταν αρχικά.

Για τις λειτουργίες **read**, **write** που προσφέρονται στο χρήστη και αρχικά ελέγχονταν ποια από αυτές θα εκτελεστεί μέσω της δομής **if**, έχουν συγχωνευτεί σε μία λειτουργία τη **readwrite**, όπου η **τιμή 100** της μεταβλητής **write\_percentage** υποδηλώνει ότι όλα τα requests θα είναι **write**, η **τιμή 0** ότι όλα τα requests θα είναι **read**, ενώ **κάθε ενδιάμεση τιμή** δηλώνει ότι ένα πλήθος αιτήσεων θα είναι **write** και οι υπόλοιπες **read**.

Στη δομή **if**, ανοίγει μία φορά η βάση δεδομένων μέσω της συνάρτησης **db\_add()**, καλείται η συνάντηση **initialise\_threads()** με όρισμα το **write\_percentage** και αφότου ολοκληρωθεί επιτυχώς η πολυνηματική επεξεργασία της βάσης δεδομένων, η βάση κλείνει με τη συνάρτηση **db\_close()**.

```
// In the main() function, the count,r variables were set as globals,
// as mentioned above, for the purpose of the multithreading
// Original, they had local scope in the main() function

int main(int argc,char** argv)
{
    srand(time(NULL));

    // The main() function gets 4 input arguments now instead of 3
    // so the check is also changed properly
    if (argc < 4) {
        fprintf(stderr,"Usage: db-bench <readwrite> <count> <write percentage> <r> (optional)\n");
        exit(1);
    }

    // The value of r variable is set to 1 when 5 input arguments are given
    // instead of 4 previously

    // In each case, the database has to open and close once
    // As opening and closing it in each thread can cause
    // memory management

    if (strcmp(argv[1], "readwrite") == 0) {
        r = 0;

        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();

        if (argc == 5)
            r = 1;

        db = db_open(DATAS);
        int threads = initialise_threads(atoi(argv[3]));

        if (threads > 0){
            printf("\n\nMultithreading finished successfully");
        }

        db_close(db);
        //_read_test(count, r);
    }
    else {
        fprintf(stderr,"Usage: <readwrite> <count> <write percentage> <r> (optional)\n");
        exit(1);
    }

    return 1;
}
```

Η συνάρτηση **initialise\_threads()** καθορίζει τη λειτουργία που θα εκτελεστεί, βάσει του **write\_percentage**, ο πλήθος των αιτήσεων που θα αποτελούν τα **writes** και τα **reads**, ανάλογα με την επιθυμητή λειτουργία και τον διαμοιρασμό τους στα νήματα.



Η μεταβλητή **ok\_execution** πρόκειται για τη τιμή που επιστρέφεται από τις συναρτήσεις **execute\_reads\_writes()**, **execute\_reads\_only()**, **execute\_writes\_only()** ώστε να μπορεί να κατανοήσει ο χρήστης την **επιτυχή ή όχι** εκτέλεση της πολυνηματικής επεξεργασίας της βάσης.

Οι μεταβλητές **write\_count** και **read\_count** αντιπροσωπεύουν το πλήθος των **writes** και **reads**. Αρχικά υπολογίζεται το πλήθος των αιτημάτων **write** ως ποσοστό επί του πλήθους όλων των αιτήσεων και τα **reads** αποτελούν τις εναπομείναντες αιτήσεις.

Στη συνέχεια, εάν η τιμή του **write\_percentage** είναι μεταξύ του 0 και του 100, αυτό υποδηλώνει ότι ο χρήστης επιθυμεί να εκτελέσει τη **λειτουργία read – write**.

Τα νήματα που θα εκτελέσουν τα **writes** υπολογίζονται ως ποσοστό επί του πλήθους όλων των νημάτων και νήματα που θα εκτελέσουν τα **reads** αποτελούν τα εναπομείναντα νήματα.

Οι αιτήσεις μοιράζονται στα νήματα και εκτελείται η συνάντηση **execute\_reads\_writes()**. Εφόσον η επιστρεφόμενη τιμή είναι θετική, τυπώνονται οι συνολικοί χρόνοι για την πραγματοποίηση των **writes** και των **reads**, το άθροισμα των χρόνων και τα στατιστικά για τα **writes** και τα **reads**.

```
// The initialise_threads() function is responsible for
// executing the multithreading read-only, write-only
// or simultaneous read-write operation
// under the rules of the mutual exclusion

// Its argument is the write percentage that the user give as an input
// Its return value is the return value of the called functions implemented above
// which indicates the successful execution of multithreading or not
int initialise_threads(int write_percentage){

    int ok_execution;

    // The number of write operations is directly determined
    // by the write percentage
    // the round() function is used in order to have a better
    // approximation of the reads and writes that will be executed
    write_count = round(count * write_percentage/100);

    // the read operations are the rest of the operations
    // meaning that once the write operations are calculated
    // the write operations are extracted from the total
    // number of operation to get the read operations number
    read_count = count - write_count;

    printf(" INITIALIZE %d %d",write_count,read_count);

    // If the write percenatage is an integer in between 0 and 100
    // that means the read-write operation is executed

    if (write_percentage > 0 && write_percentage < 100){

        // After the number of read and write requests is calculated
        // each thread gets an equal ammount of read and write
        // requests by deviding the read/write requests by the
        // number of threads

        writing_threads = round(NUMBER_OF_THREADS * write_percentage/100);
        reading_threads = NUMBER_OF_THREADS - writing_threads;

        reads_per_thread = read_count/reading_threads;
        writes_per_thread = write_count/writing_threads;

        ok_execution = execute_reads_writes();

        if (ok_execution > 0){

            printf("\n\ntotal_read_cost: %f\n\n",total_read_cost);
            printf("total_write_cost: %f\n\n",total_write_cost);
            printf("total cost: %f\n\n",total_write_cost + total_read_cost);
            printf("|Random-Read: %.6f sec/op; %.1f reads /sec(estimated); cost:%.3f(sec)\n",
                (double)(total_read_cost / read_count),
                (double)(read_count/ total_read_cost),
                total_read_cost);

            printf("|Random-Write: %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec);\n",
                (double)(total_write_cost/ write_count),
                (double)(write_count / total_write_cost),
                total_write_cost);
            printf("\n\n");

        }

    }

}
```

Εάν η τιμή του `write_percentage` είναι η **τιμή 100**, αυτό υποδηλώνει ότι ο χρήστης επιθυμεί να εκτελέσει τη **λειτουργία write**.

Τα νήματα που θα εκτελέσουν τα `writes` πρόκειται για το σύνολο όλων των διαθέσιμων νημάτων, καθώς δεν υπάρχουν αιτήσεις `reads` στη περίπτωση αυτή.

Οι αιτήσεις μοιράζονται στα νήματα και εκτελείται η συνάντηση **`execute_writes_only()`**. Εφόσον η επιστρεφόμενη τιμή είναι θετική, τυπώνονται οι συνολικοί χρόνοι για την πραγματοποίηση των `writes` και τα στατιστικά για τα `writes`.

Εάν η τιμή του `write_percentage` είναι η **τιμή 0**, αυτό υποδηλώνει ότι ο χρήστης επιθυμεί να εκτελέσει τη **λειτουργία read**.

Τα νήματα που θα εκτελέσουν τα `reads` πρόκειται για το σύνολο όλων των διαθέσιμων νημάτων, καθώς δεν υπάρχουν αιτήσεις `writes` στη περίπτωση αυτή.

Οι αιτήσεις μοιράζονται στα νήματα και εκτελείται η συνάντηση **`execute_reads_only()`**. Εφόσον η επιστρεφόμενη τιμή είναι θετική, τυπώνονται οι συνολικοί χρόνοι για την πραγματοποίηση των `reads` και τα στατιστικά για τα `reads`.

Τέλος, επιστρέφεται η τιμή της μεταβλητής **`ok_execution`**, η οποία με θετική τιμή δηλώνει την **επιτυχή εκτέλεση** της πολυνηματικής επεξεργασίας της βάσης.

```
// If the write percentage is equal to 100
// that means there are no read operations and the write-only
// operation is executed

else if (write_percentage == 100){
    write_count = count;
    writing_threads = NUMBER_OF_THREADS;
    writes_per_thread = write_count/writing_threads;
    ok_execution = execute_writes_only();

    if (ok_execution > 0){
        printf("\n\ntotal_write_cost: %f\n\n",total_write_cost);
        printf("|Random-Write: %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec)\n",
            (double)(total_write_cost/ write_count),
            (double)(write_count / total_write_cost),
            total_write_cost);
    }
    //printf("total_write_cost %f\n\n",total_write_cost);
}

// If the write percentage is equal to 0
// that means there are no write operations and the read-only
// operation is executed

else if (write_percentage == 0){
    read_count = count;
    reading_threads = NUMBER_OF_THREADS;
    reads_per_thread = read_count/reading_threads;
    ok_execution = execute_reads_only();

    if (ok_execution > 0){
        printf("\n\ntotal_read_cost: %f\n\n",total_read_cost);
        printf("|Random-Read: %.6f sec/op; %.1f reads /sec(estimated); cost:%.3f(sec)\n",
            (double)(total_read_cost / read_count),
            (double)(read_count/ total_read_cost),
            total_read_cost);
    }
    //printf("total_read_cost %f\n\n",total_read_cost);
}

// Returning the value of 1 indicates that the multithreading succeeded
return 1;
}
```



Η συνάρτηση **execute\_reads\_writes()**, η οποία αναφέρθηκε νωρίτερα και καλείται από τη συνάρτηση **initialise\_threads()**, εκτελεί τη διεργασία μέσω νημάτων που δημιουργούνται σε αυτή.

Η τιμή της μεταβλητής δηλώνει την επιτυχή εκτέλεση της πολυνηματικής επεξεργασίας της βάσης και επιστρέφεται στη συνάρτηση **initialise\_threads()**.

Οι μεταβλητές **writing\_threads** και **reading\_threads** έχουν λάβει τις τιμές τους στη συνάρτηση **initialise\_threads()** και επειδή είναι **global** μεταβλητές, δεν είναι απαραίτητο το πέρασμά τους ως ορίσματα της συνάρτησης.

Για τα νήματα γραφείς και τα νήματα αναγνώστες, δεσμεύεται ο **απαραίτητη μνήμη** μέσω της κλήσης **malloc()**.

Μέσω της κλήσης **pthread\_create()**, δημιουργούνται τα νήματα γραφείς που εκτελούν τη συνάρτηση **thread\_write\_func()** και τα νήματα αναγνώστες που εκτελούν τη συνάρτηση **thread\_read\_func()**.

Για καθένα από τα νήματα αυτά εξετάζεται εάν η δημιουργία τους ήταν επιτυχής μέσω της μεταβλητής **thread\_created\_successfully**, που στη τιμή 0 που επιστρέφει η κλήση **pthread\_create()** δηλώνεται επιτυχία και σε οποιαδήποτε άλλη τιμή η αποτυχία δημιουργίας του νήματος.

Στη περίπτωση αυτή, επιστρέφεται η **τιμή -1** ως ένδειξη της **αποτυχημένης δημιουργίας** του νήματος.

```
// Depending on which mode is executed (read,write,read-write)
// There are three functions to support it

// The following function performs the read-write operation
// Its return value is an int that indicates whether the
// multithreading is executed successfully or not
int execute_reads_writes(){

    // The thread_no is a variable that is used on the for loops below
    // to indicate the thread that is running
    int thread_no;

    // The thread_created_successfully is used to determine whether a
    // thread is successfully created
    int thread_created_successfully;

    // The following variables are two arrays of pthread_t type
    // They store the reader and writer threads
    // The appropriate memory is allocated using the malloc() call of the operating system
    pthread_t * writer_threads = (pthread_t *) malloc(writing_threads * sizeof(pthread_t));
    pthread_t * reader_threads = (pthread_t *) malloc(reading_threads * sizeof(pthread_t));

    // The writer and reader threads are created in the respective for loop
    // The following for loop initiates the writer threads

    for (thread_no=0;thread_no<writing_threads;thread_no++){

        thread_created_successfully = pthread_create(&writer_threads[thread_no], NULL, thread_write_func, NULL);

        // pthread_create() function returns 0 on successful creation
        // and a positive value otherwise
        // so it is checked if the value is not 0 to determine unsuccessful creation
        // and -1 is returned as an indicator of unsuccessful thread creation
        if (thread_created_successfully!=0){
            printf("HERE");
            return -1;
        }

        //usleep(2);
        printf("thread %d created",thread_no);
    }

    // The following for loop initiates the reader threads

    for (thread_no=0;thread_no< reading_threads;thread_no++){

        thread_created_successfully = pthread_create(&reader_threads[thread_no], NULL, thread_read_func, NULL);

        // pthread_create() function returns 0 on successful creation
        // and a positive value otherwise
        // so it is checked if the value is not 0 to determine unsuccessful creation
        // and -1 is returned as an indicator of unsuccessful thread creation
        if (thread_created_successfully!=0){
            return -1;
        }

        //usleep(2);
        printf("thread %d created",thread_no);
    }
}
```

Τα νήματα, είτε πρόκειται για γραφείς είτε για αναγνώστες, **πρέπει να περιμένουν** το καθένα να τελειώσει η **εκτέλεση των υπολοίπων**. Αυτό εξασφαλίζεται μέσω της κλήσης **pthread\_join()**, η οποία στη τιμή 0 δηλώνεται επιτυχία και σε οποιαδήποτε άλλη τιμή ότι προέκυψε κάποιο σφάλμα κατά την αναμονή ενός νήματος.

Στη περίπτωση αυτή, επιστρέφεται η **τιμή -1** ως ένδειξη σφάλματος **κατά την αναμονή** ενός νήματος να τελειώσει η εκτέλεση κάποιου άλλου.

Τέλος, η **μνήμη** που δεσμεύτηκε για τα νήματα γραφείς και τα νήματα αναγνώστες **ελευθερώνεται** μέσω της **κλήσης free()** και επιστρέφεται η τιμή 1 ως ένδειξη επιτυχούς εκτέλεσης του πολυνηματικής επεξεργασίας της βάσης.

```
// For each reader and writer thread, it is joined
// so that it can wait the other threads to finish,
// if it has already finished

int i;

// The thread_joined_succesfully variable indicates
// whether each thread in successfully joined
int thread_joined_succesfully;
for (i = 0; i < writing_threads ; i++)
    thread_joined_succesfully = pthread_join(writer_threads[i], NULL);

    // pthread_join() function returns 0 on successful thread joining
    // so it is checked if the value is not 0 to determine unsuccessful creation
    // and -1 is returned as an indicator of unsuccessful thread creation

    if (thread_joined_succesfully!=0){
        return -1;
    }

for (i = 0; i < reading_threads ; i++)
    thread_joined_succesfully = pthread_join(reader_threads[i], NULL);

    // pthread_join() function returns 0 on successful thread joining
    // so it is checked if the value is not 0 to determine unsuccessful creation
    // and -1 is returned as an indicator of unsuccessful thread creation

    if (thread_joined_succesfully!=0){
        return -1;
    }

// Finally, the memory allocated for the arrays
// that store the reader/writer threads is now free
// using the free() function

free(writer_threads);
free(reader_threads);

// Returning the value of 1 indicates that the multithreading succeeded
return 1;
```

Η συνάρτηση **execute\_writes\_only()**, η οποία αναφέρθηκε νωρίτερα και καλείται από τη συνάρτηση `initialise_threads()`, εκτελεί τη διεργασία μέσω νημάτων που δημιουργούνται σε αυτή.

Η τιμή της μεταβλητής δηλώνει την επιτυχή εκτέλεση της πολυνηματικής επεξεργασίας της βάσης και επιστρέφεται στη συνάρτηση `initialise_threads()`.

Οι μεταβλητή `writing_threads` έχει λάβει τη τιμή της στη συνάρτηση `initialise_threads()` και επειδή είναι global μεταβλητή, δεν είναι απαραίτητο το πέρασμα της ως ορίσμα της συνάρτησης.

Για τα νήματα γραφείς, δεσμεύεται ο απαραίτητη μνήμη μέσω της κλήσης `malloc()`.

Μέσω της κλήσης `pthread_create()`, δημιουργούνται τα νήματα γραφείς που εκτελούν τη συνάρτηση `thread_write_func()`.

Για καθένα από τα νήματα αυτά εξετάζεται εάν η δημιουργία τους ήταν επιτυχής μέσω της μεταβλητής `thread_created_successfully`, που στη τιμή 0 που επιστρέφει η κλήση `pthread_create()` δηλώνεται επιτυχία και σε οποιαδήποτε άλλη τιμή η αποτυχία δημιουργίας του νήματος.

Στη περίπτωση αυτή, επιστρέφεται η τιμή -1 ως ένδειξη της αποτυχημένης δημιουργίας του νήματος.

```
// The following function performs the write operation
// Its return value is an int that indicates whether the
// multithreading is executed successfully or not
int execute_writes_only(){

    // The thread_no is a variable that is used on the for loop below
    // to indicate the thread that is running
    int thread_no;

    // The thread_created_succefully is used to determine whether a
    // thread is successfully created
    int thread_created_succefully;

    // The following variable is an array of pthread_t type
    // It stores the writer threads
    // The appropriate memory is allocated using the malloc() call of the operating system
    pthread_t * writer_threads = (pthread_t *) malloc(writing_threads * sizeof(pthread_t));

    for (thread_no=0;thread_no<writing_threads;thread_no++){

        thread_created_succefully = pthread_create(&writer_threads[thread_no], NULL, thread_write_func, NULL);

        // pthread_create() function returns 0 on successful creation
        // and a positive value otherwise
        // so it is checked if the value is not 0 to determine unsuccessful creation
        // and -1 is returned as an indicator of unsuccessful thread creation

        if (thread_created_succefully!=0){
            return -1;
        }

        //usleep(2);
        printf("thread %d created",thread_no);
    }

    int i;

    // The thread_joined_succefully variable indicates
    // whether each thread in successfully joined
    int thread_joined_succefully;
    for (i = 0; i < writing_threads; i++)
        thread_joined_succefully = pthread_join(writer_threads[i], NULL);

    // pthread_join() function returns 0 on successful thread joining
    // so it is checked if the value is not 0 to determine unsuccessful creation
    // and -1 is returned as an indicator of unsuccessful thread creation

    if (thread_joined_succefully!=0){
        return -1;
    }

    // Finally, the memory allocated for the array
    // that store the writer threads is now free
    // using the free() function

    free(writer_threads);

    // Returning the value of 1 indicates that the multithreading succeeded
    return 1;
}
```

Τα **νήματα γραφείς** πρέπει να **περιμένουν** το καθένα να τελειώσει η **εκτέλεση των υπολοίπων**. Αυτό εξασφαλίζεται μέσω της κλήσης **pthread\_join()**, η οποία στη τιμή 0 δηλώνεται επιτυχία και σε οποιαδήποτε άλλη τιμή ότι προέκυψε κάποιο σφάλμα κατά την αναμονή ενός νήματος.

Στη περίπτωση αυτή, επιστρέφεται η **τιμή -1 ως ένδειξη σφάλματος** κατά την αναμονή ενός νήματος να τελειώσει η εκτέλεση κάποιου άλλου.

Τέλος, η **μνήμη που δεσμεύτηκε** για τα **νήματα γραφείς** ελευθερώνεται μέσω της κλήσης **free()** και επιστρέφεται η τιμή 1 ως ένδειξη επιτυχούς εκτέλεσης του πολυνηματικής επεξεργασίας της βάσης.

Η συνάρτηση **execute\_reads\_only()**, η οποία αναφέρθηκε νωρίτερα και καλείται από τη συνάρτηση **initialise\_threads()**, εκτελεί τη διεργασία μέσω νημάτων που δημιουργούνται σε αυτή.

Η τιμή της μεταβλητής δηλώνει την επιτυχή εκτέλεση της πολυνηματικής επεξεργασίας της βάσης και επιστρέφεται στη συνάρτηση **initialise\_threads()**.

Οι μεταβλητή **reading\_threads** έχει λάβει τη τιμή της στη συνάρτηση **initialise\_threads()** και επειδή είναι **global** μεταβλητή, δεν είναι απαραίτητο το πέρασμα της ως όρισμα της συνάρτησης.

Για τα νήματα αναγνώστες, δεσμεύεται ο απαραίτητη μνήμη μέσω της κλήσης **malloc()**.

Μέσω της κλήσης **pthread\_create()**, δημιουργούνται τα νήματα γραφείς που εκτελούν τη συνάρτηση **thread\_read\_func()**.

Για καθένα από τα νήματα αυτά εξετάζεται εάν η δημιουργία τους ήταν επιτυχής μέσω της μεταβλητής **thread\_created\_successfully**, που στη τιμή 0 που επιστρέφει η κλήση **pthread\_create()** δηλώνεται επιτυχία και σε οποιαδήποτε άλλη τιμή η αποτυχία δημιουργίας του νήματος.

Στη περίπτωση αυτή, επιστρέφεται η τιμή -1 ως ένδειξη της αποτυχημένης δημιουργίας του νήματος.

```

// The following function performs the read operation
// Its return value is an int that indicates whether the
// multithreading is executed successfully or not
int execute_reads_only(){
    // The thread_no is a variable that is used on the for loop below
    // to indicate the thread that is running
    int thread_no;
    // The thread_created_succesfully is used to determine whether a
    // thread is successfully create
    int thread_created_succesfully;
    // The following variable is an array of pthread_t type
    // It stores the reader threads
    // The appropriate memory is allocated using the malloc() call of the operating system
    pthread_t * reader_threads = (pthread_t *) malloc(reading_threads * sizeof(pthread_t));
    for (thread_no=0;thread_no<reading_threads;thread_no++){
        thread_created_succesfully = pthread_create(&reader_threads[thread_no], NULL, thread_read_func, NULL);
        // pthread_create() function returns 0 on successful creation
        // and a positive value otherwise
        // so it is checked if the value is not 0 to determine unsuccessful creation
        // and -1 is returned as an indicator of unsuccessful thread creation
        if (thread_created_succesfully!=0){
            return -1;
        }
        //usleep(2);
        printf("thread %d created",thread_no);
    }
    int i;
    // The thread_joined_succesfully variable indicates
    // whether each thread in successfully joined
    int thread_joined_succesfully;
    for (i = 0; i < reading_threads; i++){
        thread_joined_succesfully = pthread_join(reader_threads[i], NULL);
        // pthread_join() function returns 0 on successful thread joining
        // so it is checked if the value is not 0 to determine unsuccessful creation
        // and -1 is returned as an indicator of unsuccessful thread creation
        if (thread_joined_succesfully!=0){
            return -1;
        }
    }
    // Finally, the memory allocated for the array
    // that store the reader threads in now free
    // using the free() function
    free(reader_threads);
    // Returning the value of 1 indicates that the multithreading succeeded
    return 1;
}

```

Τα  
νήματα  
αναγν

ώστες πρέπει να περιμένουν το καθένα να τελειώσει η **εκτέλεση των υπολοίπων**. Αυτό εξασφαλίζεται μέσω της κλήσης **pthread\_join()**, η οποία στη τιμή 0 δηλώνεται επιτυχία και σε οποιαδήποτε άλλη τιμή ότι προέκυψε κάποιο σφάλμα κατά την αναμονή ενός νήματος.

Στη περίπτωση αυτή, επιστρέφεται η **τιμή -1 ως ένδειξη σφάλματος** κατά την αναμονή ενός νήματος να τελειώσει η εκτέλεση κάποιου άλλου.

Τέλος, η μνήμη που δεσμεύτηκε για τα νήματα αναγνώστες ελευθερώνεται μέσω της κλήσης **free()** και επιστρέφεται η τιμή 1 ως ένδειξη επιτυχούς εκτέλεσης του πολυνηματικής επεξεργασίας της βάσης.

```

int i;

// The thread_joined_succefully variable indicates
// whether each thread in successfully joined
int thread_joined_succefully;

for (i = 0; i < reading_threads; i++)
    thread_joined_succefully = pthread_join(reader_threads[i], NULL);

    // pthread_join() function returns 0 on successful thread joining
    // so it is checked if the value is not 0 to determine unsuccessful creation
    // and -1 is returned as an indicator of unsuccessful thread creation

    if (thread_joined_succefully!=0){
        return -1;
    }

// Finally, the memory allocated for the array
// that store the reader threads in now free
// using the free() function

free(reader_threads);

// Returning the value of 1 indicates that the multithreading succeded
return 1;
}

// The initialise_threads() function is responsible for
// executing the multithreading read-only, write-only
// or simultaneous read-write operation
// under the rules of the mutual exclusion

// Its argument is the write percentage that the user give as an input
// Its return value is the return value of the called functions implemented above
// which indicates the successful execution of multithreading or not

```

Οι συναρτήσεις που καλούνται εν τέλει για την **ανάγνωση** και **εγγραφή τιμών** είναι οι συναρτήσεις **thread\_read\_func()** και **thread\_write\_func()**, των οποίων τα **prototypes** έχουν δηλωθεί στο αρχείο **bench.h**.

Οι συναρτήσεις αυτές κάνουν κλήση των συναρτήσεων **\_write\_test()** και **\_read\_test()** αντίστοιχα για τη πραγματοποίηση της ανάγνωσης και εγγραφής τιμών στη βάση δεδομένων.

Όπως έχει επισημανθεί και νωρίτερα, οι συναρτήσεις **\_write\_test()** και **\_read\_test()** έχουν, πλέον, ένα επιπλέον όρισμα (db), καθώς η βάση δεδομένων ανοίγει και κλείνει μόνο μία φορά για τις ανάγκες της πολυνηματικής υλοποίησης και όχι πολλαπλές φορές όπως συνέβαινε αρχικά.

```

// The functions that are executed by the threads
// in order to run the _write_test(),_read_test()
// functions that are responsible for the read/write requests
void * thread_write_func(void * arg){
    _write_test(writes_per_thread,r,db);
}

void * thread_read_func(void * arg){
    _read_test(reads_per_thread,r,db);
}

```

AP  
XE  
IO

## kiwi.c

Στο αρχείο **kiwi.c** έχει προστεθεί **επιπλέον κώδικας**, προκειμένου να υπολογίζονται τα οι **συνολικοί χρόνοι** εκτέλεσης για την **εγγραφή και ανάγνωση** τιμών. Οι τιμές αυτές χρησιμοποιούνται εν τέλει από το **αρχείο bench.c** , όπως παρουσιάστηκε παραπάνω, για τη προβολή τους στο χρήστη, αφότου ολοκληρωθεί η πολυνηματική επεξεργασία της βάσης.



Στο σημείο αυτό, επισημαίνονται οι μεταβλητές που χρησιμοποιούνται στο αρχείο kiwi.c, οι οποίες έχουν δηλωθεί στο αρχείο bench.h, καθώς και οι μεταβλητές που συμβάλουν **στη προστασία των κρίσιμων περιοχών** που υπολογίζονται τα συνολικά κόστη εγγραφής και ανάγνωσης.

Οι μεταβλητές **total\_write\_cost** και **total\_read\_cost** χρησιμοποιούνται για τον υπολογισμό των συνολικών χρόνων εκτέλεσης για την εγγραφή και ανάγνωση τιμών αντίστοιχα και , μέσω της δήλωσης τους στο αρχείο bench.h, μπορούν να χρησιμοποιηθούν στο αρχείο bench.c που γίνεται εκτύπωση των τιμών τους στο χρήστη.

Οι μεταβλητές **readers\_cost\_mutex** και **writers\_cost\_mutex** χρησιμοποιούνται για την προστασία των κρίσιμων περιοχών που αφορούν τον ορθό υπολογισμό των συνολικών χρόνων εγγραφής και ανάγνωσης, προκειμένου ο χρόνος που έχει υπολογίσει ένα νήμα **να μην αλλοιωθεί** από κάποιο άλλο που **ενδέχεται να υπολογίζει** τον χρόνο αυτό την ίδια στιγμή.

```
// In the kiwi.c file, the total time it
// takes for the read/write requests to be executed
// is calculated here

// As stated in the comments of bench.h header file
// the values of total_write_cost and total_read_cost variables
// are calculated here and since they are defined in the bench.h header file
// bench.c can also use them since it includes the bench.h header file
double total_write_cost=0;
double total_read_cost=0;

// Since reading/writing is a multithreaded process
// the value of the cost it takes for a request to be completed
// is a shared resource
// Meaning the cost has to be protected as it is calculated,
// so that is not possible for another thread to alter it at the same time

// The cost of a reading request is protected by the readers_cost_mutex
pthread_mutex_t readers_cost_mutex = PTHREAD_MUTEX_INITIALIZER;

// The cost of a writing request is protected by the writers_cost_mutex
pthread_mutex_t writers_cost_mutex = PTHREAD_MUTEX_INITIALIZER;

// _write_test() function is modified to have the DB as an argument
// Originally, the DB opened and closed when the function was called
// But now opening and closing it multiple times can cause problems
// So it opens/closes on kiwi.c file and passed as an argument here
```

Στον ήδη **υπάρχοντα κώδικα** της συνάρτησης **\_write\_test()**, αφότου υπολογιστεί το κόστος που απαιτείται για τις εγγραφές που πραγματοποιούνται από ένα νήμα γραφέα, η προσαύξηση της μεταβλητής **total\_write\_cost** αποτελεί μία **κρίσιμη περιοχή** διότι υπάρχει η περίπτωση ένα νήμα να βρίσκεται στο **στάδιο υπολογισμού** της μεταβλητής **cost** ενώ άλλο να τη **προσαναζάνει**. Για να αποφευχθεί η παραπάνω περίπτωση, είναι απαραίτητος ο προσδιορισμός της κρίσιμης περιοχής και η χρήση των κλήσεων **lock()/ unlock()** για να αυξηθεί ορθά η τιμή της μεταβλητής **total\_write\_cost**.

```

end = get_uptime_sec();
cost = end - start;

// After the write cost is calculated, it is added
// on total_write_cost variable inside a critical section
// to prevent its value being modified by another thread and
// resulting in inaccurate total writing time
pthread_mutex_lock(&writers_cost_mutex);

total_write_cost += (double)(cost / count);

pthread_mutex_unlock(&writers_cost_mutex);

```

Στον ήδη υπάρχοντα κώδικα της συνάρτησης `_read_test()`, αφότου υπολογιστεί το κόστος που απαιτείται για τις αναγνώσεις που πραγματοποιούνται τα νήματα γραφείς, η προσαύξηση της μεταβλητής `total_read_cost` αποτελεί μία **κρίσιμη περιοχή** διότι υπάρχει η περίπτωση ένα νήμα να βρίσκεται στο **στάδιο υπολογισμού** της μεταβλητής `cost` ενώ άλλο να τη προσανξάνει. Για να αποφευχθεί η παραπάνω περίπτωση, είναι απαραίτητος ο προσδιορισμός της κρίσιμης περιοχής και η χρήση των κλήσεων `lock()/ unlock()` για να αυξηθεί ορθά η τιμή της μεταβλητής `total_read_cost`.

```

// After the read cost is calculated, it is added
// on total_read_cost variable inside a critical section
// to prevent its value being modified by another thread and
// resulting in inaccurate total reading time

pthread_mutex_lock(&readers_cost_mutex);

total_read_cost += (double)(cost / count);

pthread_mutex_unlock(&readers_cost_mutex);

```

## APXEIO db.h

Το αρχείο **db.h** αποτελεί το header file στο οποίο συμπεριλαμβάνονται οι μεταβλητές συνθήκης (**condition variables**) οι οποίες χρησιμοποιούνται για τη πραγματοποίηση της υλοποίησης του αμοιβαίου αποκλεισμού, καθώς επίσης και οι μεταβλητές που εξασφαλίζουν το έλεγχο εισόδου των αναγνωστών (**readers**) και των γραφέων (**writers**) στη κοινόχρηστη βάση δεδομένων.

Στο αρχείο αυτό έχει οριστεί και το macro **DEBUGGING\_PRINTS\_ENABLED**, η χρήση του οποίου δίνει τη δυνατότητα στο χρήστη να **τυπώσει τα μηνύματα εκσφαλμάτωσης** και , κατ' επέκταση, να τα τυπώσει σε ένα **αρχείο txt** αντί του **standard output** με τη χρήση του ">" στη γραμμή εντολών (**terminal**).

Στη παρούσα φάση, θα αναφερθεί ο ρόλος των μεταβλητών που έχουν δηλωθεί στο αρχείο db.h και αναλυτικότερη επεξήγηση της χρήσης του θα συμπεριληφθεί κατά την επεξήγηση της υλοποίησης του αλγορίθμου αναγνωστών – γραφέων.

**writers\_mutex**: Η μεταβλητή αυτή αποτελεί τη κοινόχρηστη κλειδαριά η οποία χρησιμοποιείται από τις κλήσεις **pthread\_cond\_wait()** και **pthread\_cond\_broadcast()**.

Η μεταβλητή αυτή χρησιμοποιείται για την διεξαγωγή της επικοινωνίας μεταξύ αναγνωστών – γραφέων, καθώς οι κλήσεις **pthread\_cond\_wait()**, **pthread\_cond\_broadcast()** , με κατάλληλη συνθήκη αναμονής στις **δομές while** όπου χρησιμοποιούνται , περικλείονται από τις κλήσεις **pthread\_mutex\_lock()** και **pthread\_mutex\_unlock()** με χρησιμοποιούμενη κλειδαριά τη μεταβλητή **writers\_mutex**.

Η κοινόχρηστη αυτή μεταβλητή εξασφαλίζει ότι οι **αναγνώστες** θα χρειαστεί να αναμείνουν έως **την έξοδο του γραφέα** και το ξεκλείδωμα της βάσης δεδομένων, καθώς θα περιμένουν μέσω της κλήσης **pthread\_cond\_wait()** που έχει ως όρισμα τη κοινόχρηστη κλειδαριά και τη μεταβλητή συνθήκης των αναγνωστών. Ο **γραφέας**, επίσης, με τη κλήση **pthread\_cond\_broadcast()** ειδοποιεί τους **αναγνώστες** που αναμένουν να ολοκληρώσει τη διαδικασία εγγραφής του και να επιτραπεί εκ νέου η πρόσβαση στη κοινόχρηστη βάση δεδομένων.

Η μεταβλητή αυτή, στη περίπτωση που η βάση είναι ξεκλειδωτή και διεκδικούν να εισέλθουν στη κοινόχρηστη βάση ταυτόχρονα, το αίτημα που θα κάνει απόπειρα εισαγωγής στη βάση, θα εισέλθει το πρώτο αίτημα και τα επόμενα θα αναμείνουν μέχρις ότου να υπάρχουν οι προϋποθέσεις εισαγωγής τους

**cond\_var\_readers/ cond\_var\_writers**: Οι μεταβλητές αυτές αποτελούν τις **μεταβλητές συνθήκης** που χρησιμοποιούνται από τους αναγνώστες και τους γραφείς αντίστοιχα, για τις ανάγκες της **επικοινωνίας** μεταξύ αναγνωστών και γραφέων, σύμφωνα με τους κανόνες επικοινωνίας που υπαγορεύει ο **αλγόριθμος αναγνωστών – γραφέων**.

Η χρήση τους εξηγείται αναλυτικότερα στη φάση επεξήγησης του αλγορίθμου αναγνωστών – γραφέων, όπου παρουσιάζεται η μεταξύ τους επικοινωνία

**write\_enabled/ read\_enabled**: Οι μεταβλητές αυτές αποτελούν τους μετρητές (**counters**) που , σε κάθε χρονική στιγμή, η τιμές τους χρησιμοποιούνται για τον **έλεγχο της εισόδου** των αναγνωστών και του εκάστοτε γραφέα, προκειμένου να μην υπάρχει παραβίαση των συνθηκών που προβλέπει ο αλγόριθμος αναγνωστών - γραφέων .

```

// the following macro allows the user to
// read the debugging messages or store them into
// a file by using the " > " to print them into a file
// instead of the standard output
// the user can enable the debugging prints in the db.c file
// by setting the macro to the value of 1 and on the value of 0
// to disable them
#define DEBUGGING_PRINTS_ENABLED 0

// In order for the readers-writers algorithm to execute
// the wait()/broadcast() system calls are required
// These calls also require to be surrounded by a mutex
// So a common mutex is used because if a writer is inside,
// the readers will wait for the writer to finish and then enter in their critical section
// That also goes on for the writer, who enters once there are no readers and no other writer
extern pthread_mutex_t writers_mutex;

// The condition variable for the readers, which is used to
// notify them when to wait and when to enter the critical section
// under the rules of the mutual exclusion
extern pthread_cond_t cond_var_readers;

// The condition variable for the writer, which is used to
// notify it when to wait and when to enter the critical section
// under the rules of the mutual exclusion
extern pthread_cond_t cond_var_writers;

// The number of readers currently inside the DB
// it is checked by the writers so they have to wait when readers
// are currently inside the DB and notify one of them to enter
// when no readers are inside the DB
extern int read_enabled;

// The number of writers currently inside the DB
// it is checked by the readers so they have to wait when a writer
// are currently inside the DB and notify them to enter
// when the writer is no longer inside the DB
// This variable has the values of 0 or 1, meaning either only one
// or no writer is inside the library

extern int write_enabled;

```

## APXEIO db.c

Ο κώδικας που έχει προστεθεί στο αρχείο db.c αποτελεί **τη βάση της υλοποίησης** του αμοιβαίου αποκλεισμού με γνώμονα τον αλγόριθμο αναγνωστών – γραφέων.

Σύμφωνα με τον αλγόριθμο αυτό, πολλαπλά νήματα αναγνωστών και γραφέων **διεκδικούν** να πραγματοποιήσουν εγγραφές τιμών στη βάση δεδομένων και αναγνώσεις τιμών με βάση ένα κλειδί από αυτή. Ωστόσο, ισχύουν **δύο βασικές αρχές** που αναφέρθηκαν στην **αρχή της αναφοράς** σχετικά με τους κανόνες που πρέπει να ακολουθούν τα νήματα γραφείς και αναγνώστες, ώστε να **μην υπάρξει κίνδυνος σφάλματος** κατά την εγγραφή ή την ανάγνωση:

➤ Τα νήματα αναγνώστες έχουν τη δυνατότητα να αναζητούν παράλληλα το καθένα τις τιμές τους από τη βάση δεδομένων, με το κλειδί που του έχει καθοριστεί, αλλά δεν επιτρέπεται την ίδια στιγμή να υπάρχει ήδη ένα νήμα γραφέας στη βάση δεδομένων.

➤ Για τα νήματα γραφείς ισχύει ο κανόνας ότι μόνο ένα νήμα γραφέας έχει τη δυνατότητα να πραγματοποιεί εγγραφές κάθε φορά και τα υπόλοιπα νήματα γραφείς και τα νήματα αναγνώστες, να αναμείνουν έως ότου να τελειώσει το νήμα γραφέας τη διαδικασία της εγγραφής.

Για την υλοποίηση του αμοιβαίου αποκλεισμού μεταξύ των συναρτήσεων **db\_add()** και **db\_get()**, οι οποίες πρόκειται για τις συναρτήσεις που πραγματοποιούν το ρόλο των γραφών και των αναγνωστών αντίστοιχα, έχουν οριστεί μία **καθολική κλειδαριά** και **δύο μεταβλητές συνθήκης** που ελέγχουν την είσοδο των γραφών και των αναγνωστών στη βάση δεδομένων, δηλαδή καθορίζουν πότε πρέπει τα νήματα γραφείς και τα νήματα αναγνώστες να περιμένουν και πότε να συνεχίσουν τη διαδικασία που τους έχει ανατεθεί.

Η κοινόχρηστη κλειδαριά (**writers\_mutex**) έχει οριστεί με σκοπό τον συγχρονισμό των κλήσεων **pthread\_cond\_wait()/pthread\_cond\_broadcast()** από τους γραφείς και τους αναγνώστες.

Κάθε φορά που **ολοκληρώνει** το αίτημα εγγραφής ένας γραφέας, ειδοποιεί τους αναγνώστες μέσω τη κοινόχρηστη κλειδαριά με τη κλήση **pthread\_cond\_broadcast()**, η οποία χρησιμοποιεί τη μεταβλητή συνθήκης των αναγνωστών (**cond\_var\_readers**).

Με τον ίδιο τρόπο ειδοποιούν οι αναγνώστες τους γραφείς μέσω της κλήσης **pthread\_cond\_broadcast()**, η οποία χρησιμοποιεί τη μεταβλητή συνθήκης των γραφών (**cond\_var\_writers**) στη περίπτωση αυτή.

Επιπρόσθετα, έχουν οριστεί οι μεταβλητές **read\_enabled** και **write\_enabled** που αξιοποιούνται ως μετρητές (**counters**) για να γνωρίζουν οι γραφείς και οι αναγνώστες τι είδους αιτήσεις υπάρχουν στη βάση δεδομένων, ώστε να **καθορίζουν ανάλογα** τη συμπεριφορά τους.

```
// In order for the readers-writers algorithm to execute
// the wait()/broadcast() system calls are required
// These calls also require to be surrounded by a mutex
// So a common mutex is used because if a writer is inside,
// the readers will wait for the writer to finish and then enter in their critical section
// That also goes on for the writer, who enters once there are no readers and no other writer
pthread_mutex_t writers_mutex = PTHREAD_MUTEX_INITIALIZER;

// The condition variable for the readers, which is used to
// notify them when to wait and when to enter the critical section
// under the rules of the mutual exclusion
pthread_cond_t cond_var_readers = PTHREAD_COND_INITIALIZER;

// The condition variable for the writer, which is used to
// notify it when to wait and when to enter the critical section
// under the rules of the mutual exclusion
pthread_cond_t cond_var_writers = PTHREAD_COND_INITIALIZER;

// The number of readers currently inside the DB
// it is checked by the writers so they have to wait when readers
// are currently inside the DB and notify one of them to enter
// when no readers are inside the DB
int read_enabled = 0;

// The number of writers currently inside the DB
// it is checked by the readers so they have to wait when a writer
// are currently inside the DB and notify them to enter
// when the writer is no longer inside the DB
// This variable has the values of 0 or 1, meaning either only one
// or no writer is inside the library
int write_enabled = 0;
```

Στη συνάρτηση **db\_add()**, η οποία πρόκειται για τη συνάρτηση του **γραφέα**, η είσοδος των γραφών εξαρτάται άμεσα από τη παρουσία των αναγνωστών στη βάση δεδομένων. Οι γραφείς **περιμένουν πρώτα**, μέσω της κλήσης συστήματος **pthread\_cond\_wait()**, μέχρις ότου να μην υπάρχει **κανένας αναγνώστης** στη βάση δεδομένων. Η κλήση **pthread\_cond\_wait()** πρέπει συνοδεύεται από τις κλήσεις **pthread\_mutex\_lock()**, **pthread\_mutex\_unlock()** οπότε έχει συμπεριληφθεί μεταξύ των κλήσεων αυτών.

Η κλήση **pthread\_cond\_wait()** έχει τη δυνατότητα να **ξεκλειδώνει** τη κοινόχρηστη κλειδαριά (**writers\_mutex**), με αποτέλεσμα να υπάρχει απόπειρα να εισέλθουν και γραφείς και αναγνώστες στη κοινόχρηστη βάση. Οι γραφείς αναμένουν τους αναγνώστες να ολοκληρώσουν τη διαδικασία ανάγνωσης όσο ικανοποιείται η συνθήκη αναμονής.

Παράλληλα, στη συνάρτηση **db\_get()** η κοινόχρηστη κλειδαριά εξακολουθεί να **παραμένει ξεκλειδωτή** και, εφόσον δεν υπάρχουν γραφείς στη βάση, εισέρχονται μαζικά για να εκτελέσουν τη διαδικασία της αναζήτησης τιμών.



Μόλις εξασφαλίσουν οι γραφείς ότι δεν υπάρχει **κανένας** αναγνώστης στη κοινόχρηστη βάση δεδομένων, μπορούν να προχωρήσουν στη διαδικασία της εγγραφής, αφότου τους **αφυπνίσουν οι αναγνώστες** μέσω της κλήσης **pthread\_cond\_broadcast()**.

Η περιοχή που είναι υπεύθυνη για την διαδικασία της εγγραφής αποτελεί τη **κρίσιμη περιοχή** των γραφείων, οπότε πρέπει να ελεγχθεί η πρόσβαση των γραφείων μέσω των κλήσεων pthread\_mutex\_lock() και pthread\_mutex\_unlock() στη κρίσιμη περιοχή.

Όταν εισέρχεται ένας γραφέας στη κρίσιμη περιοχή, η μεταβλητή **write\_enabled** αυξάνεται κατά 1 και, εφόσον **μόνο ένας γραφέας** εισέρχεται κάθε φορά, η τιμή της μεταβλητής **write\_enabled** είναι είτε 1 είτε 0.

Στην **αρχική εκδοχή** της συνάρτησης **db\_add()**, η τιμή της συνάρτησης **memtable\_add()** επιστρεφόταν απευθείας με την χρήση της δομής **return**.

Στη παρούσα υλοποίηση, μπορεί να προκληθεί πρόβλημα αν η συνάρτηση db\_add() επιστρέψει τη τιμή αυτή προτού ξεκλειδώσει η κρίσιμη περιοχή με τη κλήση pthread\_mutex\_unlock(), οπότε χρησιμοποιείται η τοπική μεταβλητή **value\_added** για το λόγο αυτό.

Μόλις **ολοκληρωθεί η εγγραφή** μίας τιμής στη βάση δεδομένων, η τιμή της μεταβλητής **write\_enabled** **μειώνεται κατά 1**, που σηματοδοτεί ότι ο **γραφέας έχει αποχωρήσει**.

Εν τέλει, ο γραφέας **ενημερώνει τους αναγνώστες** ότι βρίσκονται σε θέση να εισέλθουν στη βάση δεδομένων μέσω της κλήσης **pthread\_cond\_broadcast()** και η τιμή της μεταβλητής value\_added επιστρέφεται.

```
int db_add(DB* self, Variant* key, Variant* value)
{
    // As explained above, wait()/broadcast() system calls
    // are surrounded by lock()/unlock() system calls
    pthread_mutex_lock(&writers_mutex);

    // When a writer attempts to enter the DB, it has to first check if
    // no readers are inside the DB. If there is at least one, the writer has to wait
    while(read_enabled > 0){
        #if DEBUGGING_PRINTS_ENABLED == 1
            printf("IN WRITERS read_enabled %d write_enabled %d\n\n", read_enabled, write_enabled);
        #endif
        pthread_cond_wait(&cond_var_writers, &writers_mutex);
    }

    // Once there are no readers inside the DB, the writers are notified and can
    // now enter the DB
    pthread_mutex_unlock(&writers_mutex);

    // Since only one writer is allowed at a time, the writer has to lock again,
    // before entering the critical section
    pthread_mutex_lock(&writers_mutex);

    // The writer increments its counter by 1 and since there is only one writer at a time
    // the value of write_enabled variable can be either 0 or 1
    write_enabled++;

    #if DEBUGGING_PRINTS_ENABLED == 1
        printf(" WRITER STARTED read_enabled %d write_enabled %d\n\n", read_enabled, write_enabled);
    #endif

    // The return value of memtable_add() function is now stored in
    // the value_added variable
    // Originally, it was returned directly by the function
    // Returning it before a mutex is unlocked can cause problems
    // to the system, so it has to be returned after the mutex is unlocked
    int value_added;

    if (memtable_needs_compaction(self->memtable))
    {
        INFO("Starting compaction of the memtable after %d insertions and %d deletions",
            self->memtable->add_count, self->memtable->del_count);
        sst_merge(self->sst, self->memtable);
        memtable_reset(self->memtable);
    }

    // The return value of memtable_add() function stored in
    // the value_added variable once the writer finishes
    value_added = memtable_add(self->memtable, key, value);

    // The writer has finished so it decrease the value of
    // write_enabled variable to 0
    write_enabled--;

    #if DEBUGGING_PRINTS_ENABLED == 1
        printf(" WRITER FINISHED read_enabled %d write_enabled %d\n\n", read_enabled, write_enabled);
    #endif

    // There is no writer inside so it notifies the readers that they can now enter
    // and unlocks the mutex that goes along with the condition variable of the readers
    pthread_cond_broadcast(&cond_var_readers);
    pthread_mutex_unlock(&writers_mutex);

    // The return value of memtable_add() function can now be returned
    // since there is no locked mutex and causes no problem to the system
    return value_added;
}
```

Η συνάρτηση **db\_get()**, η οποία πρόκειται για τη συνάρτηση που **υλοποιεί τους αναγνώστες**,



**εξαρτάται άμεσα** από τη παρουσία του **γραφέα** στη βάση δεδομένων. Όπως και στη περίπτωση των γραφείων, χρησιμοποιούνται οι δομές **pthread\_mutex\_lock()** και **pthread\_mutex\_unlock()**, ώστε να περιβάλουν τη κλήση **pthread\_cond\_wait()**, με την οποία οι αναγνώστες **αναμένουν** έως ότου **να μην υπάρχει γραφέας** στη βάση.

Η κλήση **pthread\_cond\_wait()** έχει τη δυνατότητα να **ξεκλειδώνει** τη κοινόχρηστη κλειδαριά (**writers\_mutex**), με αποτέλεσμα να υπάρχει απόπειρα να εισέλθουν και γραφείς και αναγνώστες στη κοινόχρηστη βάση. Οι αναγνώστες αναμένουν τον εκάστοτε γραφέα να ολοκληρώσει τη διαδικασία εγγραφής, όσο ικανοποιείται η συνθήκη αναμονής.

Παράλληλα, στη συνάρτηση **db\_add()** η κοινόχρηστη κλειδαριά εξακολουθεί να **παραμένει ξεκλειδωτή** και, εφόσον δεν υπάρχουν αναγνώστες στη βάση, εισέρχονται μαζικά για να εκτελέσουν τη διαδικασία της εγγραφής τιμών, αλλά η εφόσον υπάρχει η κλήση **pthread\_mutex\_lock()**, εισέρχονται ένας – ένας και εκτελείται το αίτημα εγγραφής.

Μόλις ο τελευταίος γραφέας αποχωρήσει από τη βάση, οι αναγνώστες **μπορούν να εισέλθουν μαζικά** στη βάση για να προχωρήσουν στη διαδικασία της αναζήτησης τιμών.

Για τον ίδιο λόγο με τη συνάρτηση **db\_add()**, η επιστροφή τιμής με τη δομή **return** ενδέχεται να προκαλέσει πρόβλημα όταν η τιμή της συνάρτησης **memtable\_get()** ή **sst\_get()** επιστραφεί πριν τη κλήση **pthread\_mutex\_unlock()**.

Η τιμή αυτή επιστρέφεται με τη χρήση της μεταβλητής **return\_value**. Η τιμή αυτή, ωστόσο, είναι είτε η **τιμή 1** όταν ο αναγνώστης έχει εντοπίσει την αναζητούμενη τιμή στη **δομή memtable** είτε η **τιμή** της συνάρτησης **sst\_get()**, η οποία καλείται όταν ο αναγνώστης δεν εντόπισε τη τιμή στο **memtable** και χρειάζεται να αναζητήσει τη τιμή στα επίπεδα του **sst**.

Κάθε αναγνώστης, μόλις **πραγματοποιήσει το αίτημα ανάγνωσης** από τη βάση δεδομένων, ειδοποιεί τους γραφείς με τη κλήση **pthread\_cond\_broadcast()**.

Οι γραφείς ελέγχουν τη **συνθήκη εισόδου** τους προτού συνεχίσουν στη διαδικασία της εγγραφής, δηλαδή να μην υπάρχουν αναγνώστες στη βάση και σε αντίθετη περίπτωση αναμένουν έως ότου δεν υπάρχουν αναγνώστες στη βάση.

Υπάρχει η περίπτωση η βάση δεδομένων, σε μία χρονική στιγμή, να **είναι ξεκλειδωτή** και να επιχειρήσουν και οι γραφείς και οι αναγνώστες να εκτελέσουν τη διαδικασία που τους αναλογεί. Όταν συμβεί αυτό, αυτός που θα προλάβει να εισέλθει στη βάση θα ενημερώσει την αντίστοιχη μεταβλητή (**read\_enabled** για τους αναγνώστες και **write\_enabled** για τους γραφείς) και οι υπόλοιπες αιτήσεις θα ακολουθήσουν τους κανόνες του αμοιβαίου αποκλεισμού.

```

int db_get(DB* self, Variant* key, Variant* value)
{
    // As explained above, wait()/broadcast() system calls
    // are surrounded by lock()/unlock() system calls
    pthread_mutex_lock(&writers_mutex);

    // When a reader is attempting to enter the DB,
    // first, it increments the read_enabled variable
    // as there can be multiple readers inside the DB
    read_enabled++;

    // Each reader checks if a writer is inside the DB and if so,
    // they have to wait the writer to finish and then enter the DB
    while(write_enabled == 1){
        #if DEBUGGING_PRINTS_ENABLED == 1
            printf("IN READERS read_enabled %d write_enabled %d\n\n",read_enabled,write_enabled);
        #endif
        pthread_cond_wait(&cond_var_readers,&writers_mutex);
    }

    // Once the writer is no longer inside the DB, the readers may now proceed
    pthread_mutex_unlock(&writers_mutex);

    // Unlike the writer, the reader does not have to lock the reading section
    // as there can be multiple readers in the DB, as long there is no writer

    // A reader searches first in the memtable and then in the sst
    // if the value is not found in the memtable
    // A respective value is returned using the return_value variable
    // For the same reason the value_added variable is used in db_add() function
    int return_value;

    // The found_in_memtable variable is used to determine if
    // the value was found in the memtable
    // Its use is more to make the code easier to read and be understood
    int found_in_memtable;
    #if DEBUGGING_PRINTS_ENABLED == 1
        printf("READERS STARTED read_enabled %d write_enabled %d\n\n",read_enabled,write_enabled);
    #endif

    found_in_memtable = memtable_get(self->memtable->list, key, value);

    if (found_in_memtable == 1){
        return_value = 1;
    }
    else{
        return_value = sst_get(self->sst, key, value);
    }

    pthread_mutex_lock(&writers_mutex);

    // One of the readers finished and the number of readers decreases
    read_enabled--;

    #if DEBUGGING_PRINTS_ENABLED == 1
        printf("READERS FINISHED read_enabled %d write_enabled %d\n\n",read_enabled,write_enabled);
    #endif

    // Once a reader finishes searching a value, it will notify the writers
    // The writers will then check their condition, meaning if there are no readers
    // inside the DB and enter once there is no reader
    // If there is at least one reader inside, they will wait

    pthread_cond_broadcast(&cond_var_writers);
    pthread_mutex_unlock(&writers_mutex);

    // The value of return_value variable can now be returned
    // since there is no locked mutex and causes no problem to the system
    return return_value;
}

```

## Επιβεβαίωση Ορθής Λειτουργίας του Αλγορίθμου Αναγνωστών - Γραφών

Στα πλαίσια της υλοποίησης του αλγορίθμου αναγνωστών – γραφέων, μέσω του οποίου εξασφαλίζεται ο αμοιβαίος αποκλεισμός, πραγματοποιήθηκαν **πειραματικές δοκιμές**, προκειμένου να αποδειχθεί η ορθή εκτέλεση του.

Συγκεκριμένα, προστέθηκαν στο κώδικα του αρχείου db.c ορισμένα μηνύματα εκσφαλμάτωσης (**debugging prints**) για να παρατηρηθεί η συμπεριφορά των αναγνωστών και των γραφέων, όταν αυτοί εισέρχονται στη βάση.

Παράλληλα με τα debugging prints, ο editor που χρησιμοποιήθηκε (Sublime Text 3) ,για τη προσθήκη/τροποποίηση του υπάρχοντα κώδικα, προσφέρει τη δυνατότητα αναζήτησης κειμένου με χρήση κανονικών εκφράσεων (**regular expressions**)

Τα debugging prints έχουν μεταφερθεί σε ένα αρχείο results.txt με τη χρήση του ορίσματος ">" στη γραμμή εντολών ως εξής:

```
./kiwi-bench readwrite 100000 50 > results.txt
```

Η εκτέλεση της εντολής μπορεί να πραγματοποιηθεί και με **διαφορετικά ορίσματα**. Ο **χρήστης** πρέπει να προσέξει ότι η εικονική μηχανή διαθέτει **περιορισμένο αποθηκευτικό χώρο**, οπότε πρέπει να **αυξήσει** τον αποθηκευτικό χώρο της εικονικής, ώστε να μπορούν να αποθηκευτούν τα δεδομένα για μεγαλύτερο αριθμό αιτημάτων, όπως στη περίπτωση των 500000 αιτήσεων.

Με τις **default ρυθμίσεις** της εικονικής μηχανής, ο χρήστης μπορεί να μεταφέρει χωρίς κάποιο πρόβλημα τα debugging prints της παραπάνω ενδεικτικής εντολής.

Οι γραφείς εισέρχονται ένας – ένας στη βάση δεδομένων, ενώ παράλληλα δεν υπάρχουν αναγνώστες στη βάση δεδομένων, όπως παρατηρείται από την παρακάτω εικόνα.

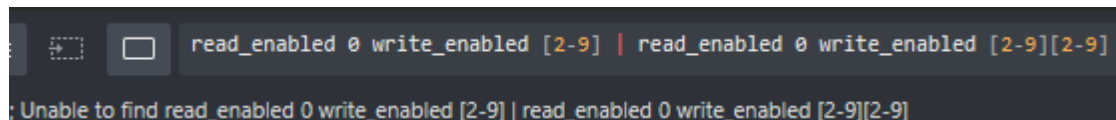
[illegible]

Η περίπτωση που πρέπει να εξεταστεί είναι να υπάρχουν περισσότεροι από ένας γραφείς, που θεωρείται μη αποδεκτή περίπτωση ορθής λειτουργίας.

Για το λόγο αυτό, η αναζήτηση του αριθμού των αναγνωστών και των γραφέων πραγματοποιείται με τη κανονική έκφραση:

```
read_enabled 0 write_enabled [2-9] | read_enabled 0 write_enabled [2-9][2-9]
```

Η κανονική αυτή έκφραση εξασφαλίζει ότι **δεν υπάρχουν περισσότεροι από ένας γραφείς** (είτε μονοψήφιος αριθμός είτε διψήφιος αριθμών γραφέων ) και ότι δεν υπάρχουν αναγνώστες την ίδια στιγμή, με το πλήθος των αναγνωστών να είναι 0.



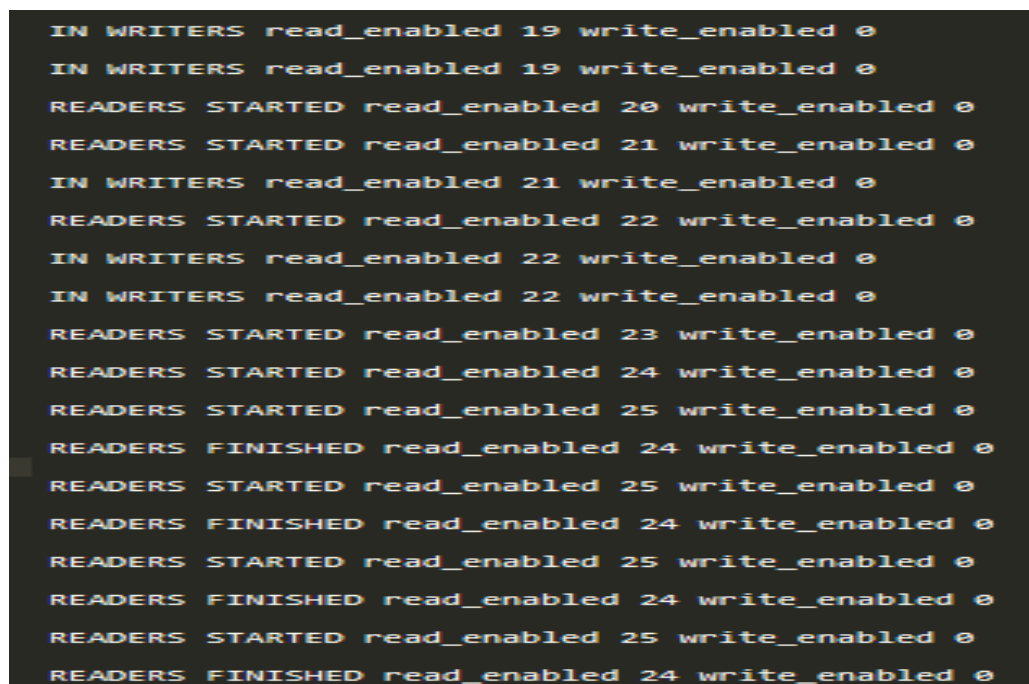
```
read_enabled 0 write_enabled [2-9] | read_enabled 0 write_enabled [2-9][2-9]
: Unable to find read_enabled 0 write_enabled [2-9] | read_enabled 0 write_enabled [2-9][2-9]
```

Οι αναγνώστες πιστοποιείται ότι εισέρχονται παράλληλα μέσω των τιμών της μεταβλητής `read_enabled`, ενώ ο αριθμός των γραφέων είναι ίσος με 0.

Στην ακόλουθη εικόνα παρουσιάζεται, παράλληλα, η απόπειρα των γραφέων να εισέλθουν στη βάση δεδομένων όσο υπάρχουν αναγνώστες στη βάση, αλλά εμποδίζονται από την κλήση `pthread_cond_wait()` στη συνάρτηση `db_add()` των γραφέων.

Μία εξίσου σημαντική περίπτωση που πρέπει να ελεγχθεί αποτελεί η αναμονή των γραφέων όσο υπάρχει τουλάχιστον ένας αναγνώστης που πραγματοποιεί τη διαδικασία αναζήτησης τιμής (`read`).

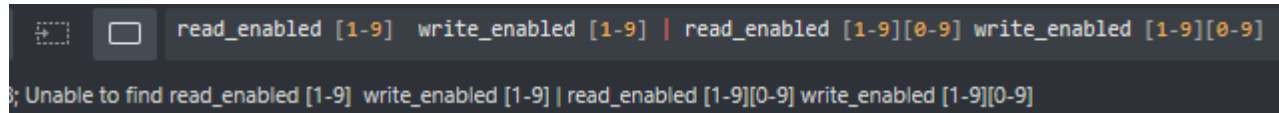
Τα παρακάτω μηνύματα εκσφαλμάτωσης πιστοποιούν επίσης ότι οι γραφείς αναμένουν τους αναγνώστες να ολοκληρώσουν τη διαδικασία της ανάγνωσης.



```
IN WRITERS read_enabled 19 write_enabled 0
IN WRITERS read_enabled 19 write_enabled 0
READERS STARTED read_enabled 20 write_enabled 0
READERS STARTED read_enabled 21 write_enabled 0
IN WRITERS read_enabled 21 write_enabled 0
READERS STARTED read_enabled 22 write_enabled 0
IN WRITERS read_enabled 22 write_enabled 0
IN WRITERS read_enabled 22 write_enabled 0
READERS STARTED read_enabled 23 write_enabled 0
READERS STARTED read_enabled 24 write_enabled 0
READERS STARTED read_enabled 25 write_enabled 0
READERS FINISHED read_enabled 24 write_enabled 0
READERS STARTED read_enabled 25 write_enabled 0
READERS FINISHED read_enabled 24 write_enabled 0
READERS STARTED read_enabled 25 write_enabled 0
READERS FINISHED read_enabled 24 write_enabled 0
READERS STARTED read_enabled 25 write_enabled 0
READERS FINISHED read_enabled 24 write_enabled 0
```

Η παρακάτω κανονική έκφραση εξασφαλίζει ότι δεν υπάρχει **μονοψήφιος ή διψήφιος αριθμός γραφών** τη στιγμή που υπάρχει κάποιος αναγνώστης την ίδια χρονική στιγμή στη βάση δεδομένων.

Ο έλεγχος αυτός μπορεί να επεκταθεί και για έλεγχο για **μεγαλύτερο αριθμό γραφών** στη βάση με **κατάλληλη επέκταση** της κανονικής έκφρασης.



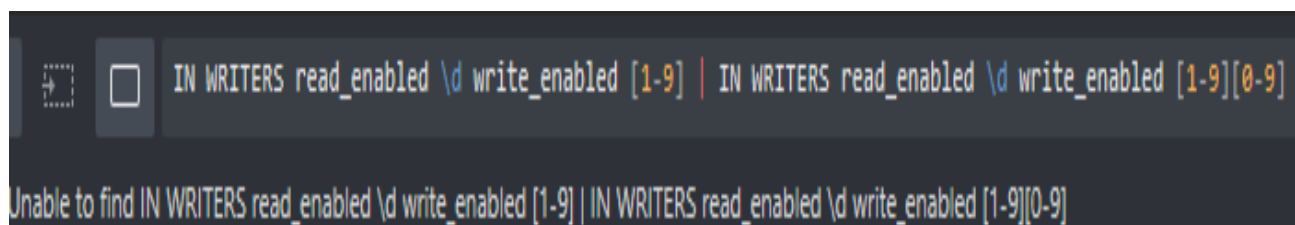
```
read_enabled [1-9] write_enabled [1-9] | read_enabled [1-9][0-9] write_enabled [1-9][0-9]
```

Unable to find read\_enabled [1-9] write\_enabled [1-9] | read\_enabled [1-9][0-9] write\_enabled [1-9][0-9]

Για να πιστοποιηθεί ότι **δεν υπάρχουν παράλληλα γραφείς** στη βάση δεδομένων κατά τη διάρκεια **αναμονής των γραφών** να ολοκληρώσουν τη λειτουργία τους οι αναγνώστες, χρησιμοποιείται η παρακάτω κανονική έκφραση:

```
IN WRITERS read_enabled \d write_enabled [1-9] | IN WRITERS read_enabled \d write_enabled [1-9][0-9]
```

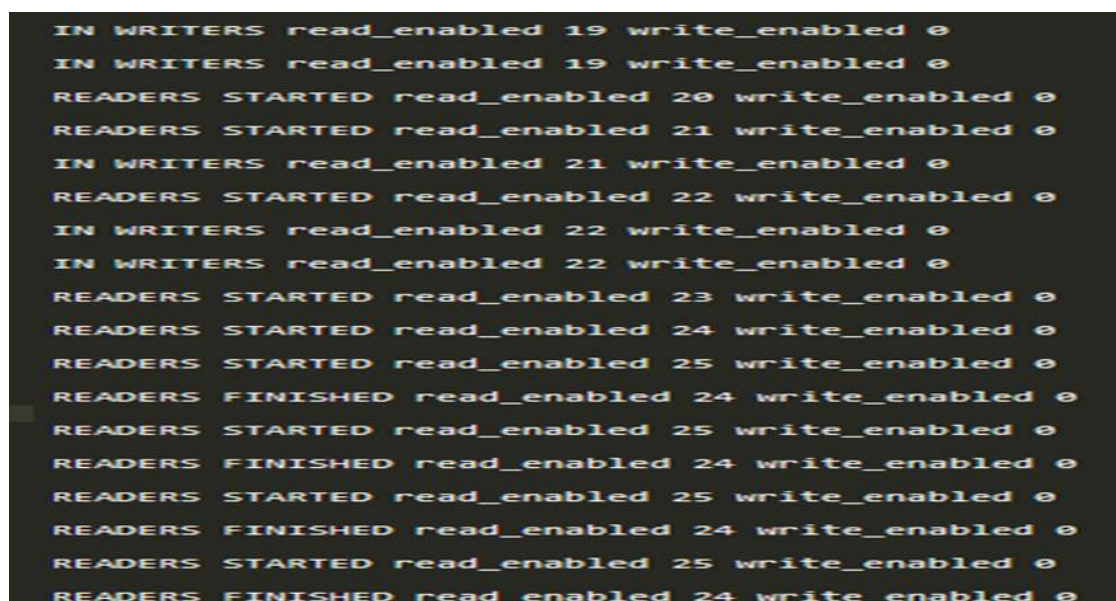
Η κανονική αυτή έκφραση πιστοποιεί ότι με **οποιοδήποτε αναγνώστες στη βάση δεδομένων**, ο αριθμός των γραφών που αναμένουν δεν είναι μεγαλύτερος του 0, είτε μονοψήφιος είτε διψήφιος.



```
IN WRITERS read_enabled \d write_enabled [1-9] | IN WRITERS read_enabled \d write_enabled [1-9][0-9]
```

Unable to find IN WRITERS read\_enabled \d write\_enabled [1-9] | IN WRITERS read\_enabled \d write\_enabled [1-9][0-9]

Η ισχύς της κανονικής έκφρασης που επισημάνθηκε αποδεικνύεται και από την εικόνα που παρατέθηκε για την απόδειξη ότι δεν υπάρχουν γραφείς στη βάση δεδομένων, κατά το διάστημα αναμονής των γραφών να ολοκληρωθούν τα αιτήματα ανάγνωσης, καθώς επίσης ότι η αναμονή των γραφών συνεπάγεται την περαιτέρω είσοδο/έξοδο των αναγνωστών στη βάση δεδομένων.



```
IN WRITERS read_enabled 19 write_enabled 0
IN WRITERS read_enabled 19 write_enabled 0
READERS STARTED read_enabled 20 write_enabled 0
READERS STARTED read_enabled 21 write_enabled 0
IN WRITERS read_enabled 21 write_enabled 0
READERS STARTED read_enabled 22 write_enabled 0
IN WRITERS read_enabled 22 write_enabled 0
IN WRITERS read_enabled 22 write_enabled 0
READERS STARTED read_enabled 23 write_enabled 0
READERS STARTED read_enabled 24 write_enabled 0
READERS STARTED read_enabled 25 write_enabled 0
READERS FINISHED read_enabled 24 write_enabled 0
READERS STARTED read_enabled 25 write_enabled 0
READERS FINISHED read_enabled 24 write_enabled 0
READERS STARTED read_enabled 25 write_enabled 0
READERS FINISHED read_enabled 24 write_enabled 0
READERS STARTED read_enabled 25 write_enabled 0
READERS FINISHED read_enabled 24 write_enabled 0
```



## Αξιολόγηση Επιδόσεων με τη Χρήση Πολυνηματισμού

Στους **παρακάτω πίνακες** παρουσιάζονται οι **χρόνοι** που απαιτούνται για τη πραγματοποίηση των λειτουργιών **read**, **write** και **read-write**, καθώς αυξάνεται το πλήθος των **αιτήσεων** και το πλήθος των νημάτων (**threads**).

Οι **χρόνοι εκτέλεσης** των λειτουργιών **επηρεάζονται** και από τον παράγοντα της **τυχαιότητας**, καθώς δεν γνωρίζουμε **εκ των προτέρων** εάν οι αιτήσεις ανάγνωσης ολοκληρωθούν ταχύτερα από τις αιτήσεις εγγραφής.

Στη **γενική περίπτωση**, οι αιτήσεις ανάγνωσης (read) **αναμένονται να ολοκληρωθούν ταχύτερα** από τις αιτήσεις εγγραφής (write), καθώς οι αιτήσεις **ανάγνωσης** εκτελούνται **παράλληλα**, ενώ οι αιτήσεις **εγγραφής** εκτελούνται **μία-μία** και η ολοκλήρωση τους αναμένεται να διαρκέσει για μεγαλύτερο χρονικό διάστημα.

Επιπλέον, οι χρόνοι εκτέλεσης των λειτουργιών read, write και read-write αναμένονται να **ακολουθούν αυξητική τάση** διότι η καθυστέρηση στην ολοκλήρωση των λειτουργιών αυτών επηρεάζεται από τους εξής παράγοντες:

- Η **χρήση νημάτων** εισάγει **καθυστερήσεις** στην εκτέλεση των λειτουργιών. Οι καθυστερήσεις αυτές δεν είναι γνωστό **το εύρος των τιμών** τους οπότε ενδέχεται στην εκτέλεση ενός πειράματος οι καθυστερήσεις αυτές να αυξομειώνονται.
- Η **διαδικασία της εγγραφής** δεδομένων στη βάση είναι γενικότερα **πιο χρονοβόρα** διαδικασία από τη διαδικασία της **αναζήτησης** τιμών, οπότε ο συνολικός χρόνος που απαιτείται για τις εγγραφές ενδέχεται να **αυξάνει το συνολικό κόστος**. Στη διαδικασία εγγραφής, ο χρόνος εκτέλεσης των εγγραφών επηρεάζεται από τη πιθανότητα να πραγματοποιηθεί σύμπτυξη (**compaction**) από το υπεύθυνο νήμα, με αποτέλεσμα η διαδικασία εγγραφής τιμών να χρειαστεί μεγαλύτερο χρονικό διάστημα για να ολοκληρωθεί.
- Κατά τη διάρκεια εκτέλεσης των αιτημάτων ανάγνωσης, ο **εκάστοτε αναγνώστης** ενδέχεται να αναζητήσει την τιμή στη **δομή sst**, σε περίπτωση που η τιμή **δεν βρέθηκε** στη δομή **memtable** και ,κατά συνέπεια, να αυξάνεται ο συνολικός χρόνος που χρειάζονται οι αναγνώστες για την εύρεση των αναζητούμενων τιμών.

Αρχικά, θα γίνει σχολιασμός των λειτουργιών **read** και **write**, καθώς **αυξάνεται** το πλήθος των **αιτημάτων** και το πλήθος των **νημάτων**.

Για τα αιτήματα **read**, παρατηρείται ότι ο **συνολικός χρόνος εκτέλεσης** τους είναι **μικρότερος** από το συνολικό χρόνο των **αιτημάτων write**, για το ίδιο πλήθος αιτημάτων.

Καθώς αυξάνεται το πλήθος των αιτημάτων και των νημάτων, είναι αναμενόμενο να **αυξάνονται** οι **συνολικοί χρόνοι εκτέλεσης** των λειτουργιών read και write, λόγω της επιρροής των παραγόντων που αναφέρθηκαν.

Οι χρόνοι εκτέλεσης των **αιτημάτων read**, παρά την αύξηση των νημάτων και του πλήθους των αιτημάτων, **παραμένουν μικρότεροι** από τους χρόνους εκτέλεσης των **αιτημάτων write**.

Η συμπεριφορά αυτή είναι επίσης **αναμενόμενη** διότι τα **αιτήματα read** πραγματοποιούνται **παράλληλα** και τα **αιτήματα write ένα-ένα**, ακολουθώντας τις αρχές του αλγορίθμου αναγνωστών γραφών.



100000			200000			500000			1000000		
THREADS	READ	WRITE	THREADS	READ	WRITE	THREADS	READ	WRITE	THREADS	READ	WRITE
20	0.004000	0.008200	20	0.008000	0.008900	20	0.005800	0.010120	20	0.005200	0.008540
50	0.025000	0.050000	50	0.025000	0.048750	50	0.035000	0.054900	50	0.030000	0.054150
100	0.100000	0.175000	100	0.132500	0.195000	100	0.127200	0.198800	100	0.120900	0.207900
150	0.225225	0.370871	150	0.335334	0.440380	150	0.256228	0.431443	150	0.270027	0.458598
180	0.324324	0.623423	180	0.324032	0.533753	180	0.373084	0.589125	180	0.388839	0.634743
200	0.400000	0.734000	200	0.585000	0.874000	200	0.480000	0.748000	200	0.480000	0.780800
250	0.625000	0.987500	250	0.623750	1.090000	250	0.747500	1.173000	250	0.743250	1.253500

Αναφορικά με τη **λειτουργία read-write**, όπως και στη περίπτωση των λειτουργιών read και write, οι **χρόνοι εκτέλεσης** ακολουθούν επίσης **ανοδική τάση**, με την αύξηση των αιτημάτων και του πλήθους των νημάτων. Στη λειτουργία read-write η αύξηση του **ποσοστού των νημάτων** που πραγματοποιούν τις **λειτουργίες write** και των αιτημάτων write επιφέρει **αύξηση** στο συνολικό χρόνο εκτέλεσης της λειτουργίας read-write.

Αυξάνοντας **σταδιακά** το ποσοστό των νημάτων και των αιτημάτων write από 10% σε 90%, παρατηρείται **σταδιακή αύξηση** στο **συνολικό κόστος εκτέλεσης** της λειτουργίας read-write. Η συμπεριφορά αυτή είναι αναμενόμενη, καθώς τα **αιτήματα write** γίνονται **σταδιακά περισσότερα** από τα αιτήματα read, που συνεπάγεται μεγαλύτερο χρόνο εκτέλεσης τους βάσει των κανόνων του αλγορίθμου αναγνωστών γραφείων.

Οι αύξηση των νημάτων και του πλήθους των αιτημάτων έχει ως άμεση συνέπεια την αύξηση το συνολικού χρόνου εκτέλεσης της λειτουργίας read-write, για τον ίδιο λόγο που περιγράφηκε και στη περίπτωση εκτέλεσης των λειτουργιών read και write.

Τέλος, εξετάζοντας τους **συνολικούς χρόνους** της λειτουργίας **read-write** καθώς αυξάνονται τα **νήματα** και το **πλήθος των αιτημάτων**, παρατηρείται ότι το άθροισμα των συνολικών χρόνων των λειτουργιών **read** και **write** είναι **μικρότερο** από το συνολικό χρόνο εκτέλεσης της λειτουργίας **read-write**, καθώς αυξάνεται το πλήθος των νημάτων, των αιτημάτων και το ποσοστό των αιτημάτων που αποτελούν τα αιτήματα write επί του πλήθους όλων των αιτημάτων.

Η εξήγηση στη παρατήρηση αυτή οφείλεται στο γεγονός ότι στη λειτουργία read-write τα **αιτήματα read** εκτελούνται **παράλληλα** από πολλαπλά νήματα και **αιτήματα write ένα – ένα**, οπότε **εξοικονομείται** ένα χρονικό διάστημα από την παράλληλη πραγματοποίηση των αιτημάτων read.

Στους πίνακες που ακολουθούν, παρατίθενται οι χρόνοι εκτέλεσης των λειτουργιών read-write, με σταδιακή αύξηση του πλήθους των αιτημάτων, του πλήθους των νημάτων, καθώς και του ποσοστού των νημάτων που θα αποτελούν τα νήματα γραφείς και των αιτημάτων εγγραφής, σε σχέση με τα νήματα αναγνώστες και τα αιτήματα ανάγνωσης.

READWRITE 10000						
THREADS	10% WRITE – 90% READ	30% WRITE – 70% READ	50% WRITE – 50% READ	70% WRITE – 30% READ	90% WRITE – 10% READ	
20	~0,00	~0,00	~0,00	~0,00	~0,00	
50	~0,00	~0,00	~0,00	~0,00	~0,00	
100	~0,00	~0,00	~0,00	~0,00	~0,00	
150	~0,00	~0,00	~0,00	~0,00	~0,00	
180	~0,00	~0,00	~0,00	~0,00	~0,00	
200	~0,00	~0,00	~0,00	~0,00	~0,00	
250	~0,00	~0,00	~0,00	~0,00	~0,00	
READWRITE 100000						
THREADS	10% WRITE – 90% READ	30% WRITE – 70% READ	50% WRITE – 50% READ	70% WRITE – 30% READ	90% WRITE – 10% READ	
20	0.004000	0.005200	0.006000	0.007600	0.008000	
50	0.031000	0.025000	0.037500	0.043500	0.044500	
100	0.100000	0.130000	0.150000	0.146000	0.180000	
150	0.249249	0.292793	0.304805	0.366366	0.426426	
180	0.353153	0.273874	0.446847	0.542342	0.518919	
200	0.378000	0.496000	0.482000	0.674000	0.748000	
250	0.545000	0.737500	0.842500	0.795000	0.890000	
READWRITE 200000						
THREADS	10% WRITE – 90% READ	30% WRITE – 70% READ	50% WRITE – 50% READ	70% WRITE – 30% READ	90% WRITE – 10% READ	
20	0.002300	0.002600	0.005000	0.004800	0.005800	
50	0.026250	0.028500	0.037500	0.047500	0.056000	
100	0.125500	0.115000	0.138000	0.155000	0.184000	
150	0.237059	0.242311	0.225806	0.305326	0.392348	
180	0.327633	0.346535	0.324032	0.459946	0.548155	
200	0.513000	0.415000	0.398000	0.583000	0.708000	
250	0.638750	0.602500	0.621250	0.723750	0.816250	
READWRITE 500000						
THREADS	10% WRITE – 90% READ	30% WRITE – 70% READ	50% WRITE – 50% READ	70% WRITE – 30% READ	90% WRITE – 10% READ	
20	0.005760	0.005520	0.005760	0.007600	0.010280	
50	0.034800	0.029600	0.038200	0.047800	0.058800	
100	0.115600	0.123000	0.132800	0.152200	0.200400	
150	0.247225	0.219622	0.257726	0.299130	0.387939	
180	0.360821	0.357940	0.370544	0.433561	0.558156	
200	0.456800	0.484800	0.494800	0.569200	0.756000	
250	0.685500	0.705000	0.678500	0.811000	0.997500	
READWRITE 1000000						
THREADS	10% WRITE – 90% READ	30% WRITE – 70% READ	50% WRITE – 50% READ	70% WRITE – 30% READ	90% WRITE – 10% READ	
20	0.005760	0.005520	0.005760	0.007600	0.010280	
50	0.034800	0.029600	0.038200	0.047800	0.058800	
100	0.115600	0.123000	0.132800	0.152200	0.200400	
150	0.247225	0.219622	0.257726	0.299130	0.387939	
180	0.360821	0.357940	0.370544	0.433561	0.558156	
200	0.456800	0.484800	0.494800	0.569200	0.756000	
250	0.685500	0.705000	0.678500	0.811000	0.997500	

## Μεταγλώττιση Παραδοτέων Αρχείων Κώδικα

Τέλος, μαζί με τους χρόνους εκτέλεσης των λειτουργιών για τις παραμέτρους που εκτελείται ο κώδικας, δηλαδή **πλήθος threads**, **πλήθος αιτημάτων** και **ποσοστό αιτημάτων εγγραφής και νημάτων γραφέν** (στη περίπτωση της λειτουργίας **read-write**), παρατίθενται screenshots με την εκτέλεση των εντολών **make clean** και **make all** στο directory **kiwi-source**, που αποδεικνύουν ότι τα αρχεία μεταγλωττίζονται επιτυχώς **χωρίς errors**.

```

myy601@myy601lab1:~/Desktop/kiwi/kiwi-source$ make clean
cd engine && make clean
make[1]: Entering directory '/home/myy601/Desktop/kiwi/kiwi-source/engine'
rm -rf *.o libindexer.a
make[1]: Leaving directory '/home/myy601/Desktop/kiwi/kiwi-source/engine'
cd bench && make clean
make[1]: Entering directory '/home/myy601/Desktop/kiwi/kiwi-source/bench'
rm -f kiwi-bench
rm -rf testdb
make[1]: Leaving directory '/home/myy601/Desktop/kiwi/kiwi-source/bench'
myy601@myy601lab1:~/Desktop/kiwi/kiwi-source$ make all
cd engine && make all
make[1]: Entering directory '/home/myy601/Desktop/kiwi/kiwi-source/engine'
CC db.o
CC memtable.o
CC indexer.o
CC sst.o
CC sst_builder.o
CC sst_loader.o
CC sst_block_builder.o
CC hash.o
CC bloom_builder.o
CC merger.o
CC compaction.o
CC skiplist.o
CC buffer.o
CC arena.o
CC utils.o
CC crc32.o
CC file.o
CC heap.o
CC vector.o
CC log.o
CC lru.o
AR libindexer.a

```

```

make[1]: Leaving directory '/home/myy601/Desktop/kiwi/kiwi-source/engine'
cd bench && make all
make[1]: Entering directory '/home/myy601/Desktop/kiwi/kiwi-source/bench'
gcc -g -gdb -Wall -Wno-implicit-function-declaration -Wno-unused-but-set-variable bench.c kiwi.c -L ../engine -lindexer -lpthread -lsnappy -o kiwi-bench
bench.c: In function 'execute_reads_writes':
bench.c:218:2: warning: this 'for' clause does not guard... [-Wmisleading-indentation]
    for (i = 0; i < writing_threads ; i++)
    ^~
bench.c:225:3: note: ...this statement, but the latter is misleadingly indented as if it were guarded by the 'for'
    if (thread_joined_successfully!=0){
    ^~
bench.c:229:2: warning: this 'for' clause does not guard... [-Wmisleading-indentation]
    for (i = 0; i < reading_threads ; i++)
    ^~
bench.c:236:3: note: ...this statement, but the latter is misleadingly indented as if it were guarded by the 'for'
    if (thread_joined_successfully!=0){
    ^~
bench.c: In function 'execute_writes_only':
bench.c:291:2: warning: this 'for' clause does not guard... [-Wmisleading-indentation]
    for (i = 0; i < writing_threads; i++)
    ^~
bench.c:298:3: note: ...this statement, but the latter is misleadingly indented as if it were guarded by the 'for'
    if (thread_joined_successfully!=0){
    ^~
bench.c: In function 'execute_reads_only':
bench.c:353:2: warning: this 'for' clause does not guard... [-Wmisleading-indentation]
    for (i = 0; i < reading_threads; i++)
    ^~
bench.c:360:3: note: ...this statement, but the latter is misleadingly indented as if it were guarded by the 'for'
    if (thread_joined_successfully!=0){
    ^~
bench.c: In function 'initialise_threads':
bench.c:399:23: warning: format '%d' expects argument of type 'int', but argument 2 has type 'long int' [-Wformat=]
    printf(" INITIALIZE %d %d",write_count,read_count);
                        ^~
bench.c:399:26: warning: format '%d' expects argument of type 'int', but argument 3 has type 'long int' [-Wformat=]
    printf(" INITIALIZE %d %d",write_count,read_count);
                        ^~

```

```

bench.c:399:26: warning: format '%d' expects argument of type 'int', but argument 3 has type 'long int' [-Wformat=]
    printf(" INITIALIZE %d %d",write_count,read_count);
                        ^~
bench.c: In function 'thread_write_func':
bench.c:141:1: warning: control reaches end of non-void function [-Wreturn-type]
}
^
bench.c: In function 'thread_read_func':
bench.c:145:1: warning: control reaches end of non-void function [-Wreturn-type]
}
^
make[1]: Leaving directory '/home/myy601/Desktop/kiwi/kiwi-source/bench'
myy601@myy601lab1:~/Desktop/kiwi/kiwi-source$

```

Συνοδευτικά με τα screenshots της επιτυχούς μεταγλώττισης του κώδικα, έχουν συμπεριληφθεί και Screenshots με την εκτέλεση του κώδικα με παραδείγματα από τις τρεις ζητούμενες λειτουργίες, βάσει της τιμής του επιπλέον ορίσματος (η **τιμή 0** για τη λειτουργία **read**, η **τιμή 100** για τη λειτουργία **write** και οποιαδήποτε **ενδιάμεση** τιμή για τη λειτουργία **read-write**).

```
./kiwi-bench readwrite 5000000 0
```

```
total_write_cost: 0.142500
|Random-Write: 0.000001 sec/op; 701754.4 writes/sec(estimated); cost:0.143(sec);
Multithreading finished successfully[7356] 01 Apr 18:39:48.121 . db.c:64 Closing database 1192
```

```
./kiwi-bench readwrite 1000000 10
```

```
total_read_cost: 0.100750
total_write_cost: 0.145750
total cost: 0.246500
|Random-Read: 0.000000 sec/op; 9925558.3 reads /sec(estimated); cost:0.101(sec)
|Random-Write: 0.000000 sec/op; 6861063.5 writes/sec(estimated); cost:0.146(sec);
```

```
log file testdb/51/24.log
```

```
./kiwi-bench readwrite 100000 100
```

```
total_write_cost: 0.342500
|Random-Write: 0.000003 sec/op; 291970.8 writes/sec(estimated); cost:0.343(sec);
Multithreading finished successfully[10134] 01 Apr 18:49:49.255 . db.c:64 Closing database 1192
```