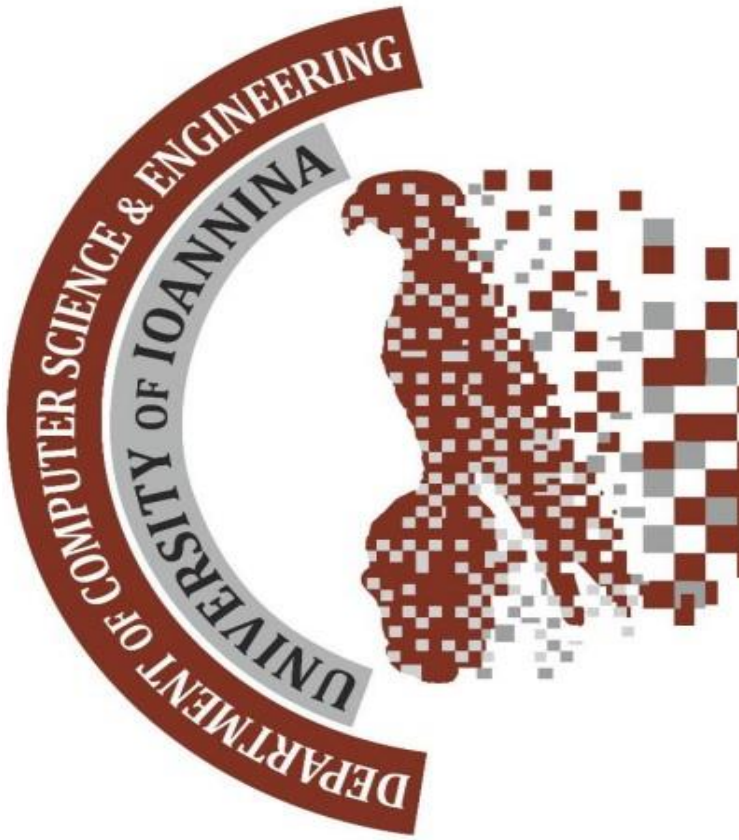
Αναφορά 1ης Εργαστηριακής Άσκησης Λειτουργικά Συστήματα



Εαρινό εξάμηνο 2021-2022

ΤΗΛΕΜΑΧΟΣ-ΜΑΡΚΟΣ ΜΠΑΖΑΚΑΣ : 3281
ΒΑΣΙΛΕΙΟΣ ΝΤΟΝΤΗΣ : 3300
ΖΗΣΗΣ-ΠΡΟΚΟΠΙΟΣ ΤΑΛΑΜΑΓΚΑΣ : 3340

ΠΕΡΙΕΧΟΜΕΝΑ ΑΝΑΦΟΡΑΣ

ΠΕΡΙΓΡΑΦΗ ΕΡΓΑΣΤΗΡΙΑΚΗΣ ΑΣΚΗΣΗΣ

Η βασική ιδέα της πρώτης εργαστηριακής άσκησης είναι η χρήση πολυνηματικής λειτουργίας σε μια μηχανή αποθήκευσης δεδομένων βασισμένη σε δέντρα LSM. Κύρια λειτουργία της μηχανής είναι η εισαγωγή ζευγαριών κλειδιών-τιμών (key-values) σε μια βάση δεδομένων και μετέπειτα η αναζήτηση ενός ή πολλαπλών τιμών δεδομένου κάποιου κλειδιού(ταυτόχρονα). Στόχος μας είναι να μπορούμε να εισάγουμε και να αναζητούμε μέσα στην βάση με την χρήση πολλαπλών νημάτων, έτσι ώστε να υπάρχει <<παραλληλία>> και μείωση του χρόνου εκτέλεσης.

ΔΙΑΔΡΟΜΗ ΕΚΤΕΛΕΣΗΣ ΠΡΟΓΡΑΜΜΑΤΟΣ

Περιληπτικά η διαδικασία που ακολουθείτε είναι:

• Για την εισαγωγή ενός ζευγαριού key-value στη μηχανή(βάση δεδομένων), αρχικά καλείτε η _write_test() από το αρχείο bench.c που και αυτή υλοποιείται στο αρχείο kiwi.c. Αυτή με την σειρά της θα καλέσει το db_add() που υλοποιείται μέσα το αρχείο db.c και εκεί καλείται αντίστοιχα το memtable_add() που προσθέτει αυτό το ζευγάρι στο memtable(πίνακας στη μνήμη RAM). Σε περίπτωση που το memtable γεμίσει δημιουργείται ένα αρχείο sst(πίνακας στο σκληρό δίσκο) και σε αυτό προστίθεται το περιεχόμενο του memtable.

• Για την ανάγνωση/εύρεση κάποιας τιμής δεδομένου ενός κλειδιού . Αρχικά καλείτε η _read_test() από την bench.c η οποία υλοποιείται μέσα στο αρχείο kiwi.c όπου και καλείτε η db_get(). Στο αρχείο db.c αφού έχουμε καλέσει την db_get() θα κληθεί πρώτα η memtable_get() και θα ψάξουμε μέσα στο memtable για να βρούμε το κλειδί που θέλουμε . Αν βρεθεί τότε επιστρέφεται το value που του αντιστοιχεί αλλιώς καλείτε η sst_get() και ψάχνουμε μέσα στα αρχεία sst για το κλειδί αυτό. Σε περίπτωση που δεν βρεθεί, επιστρέφεται το αντίστοιχο μήνυμα.

Για την εισαγωγή πολλαπλών νημάτων στο πρόγραμμά μας τροποποιήθηκαν τα αρχεία: bench.c , kiwi.c, bench.h, db.c και db.h. Παρακάτω υποδεικνύουμε αυτές τις αλλαγές και τον σκοπό τους.

BENCH

Παρακάτω θα αναλύσουμε τις αλλαγές που κάναμε σε διάφορα αρχεία του directory: bench

3.1.1.bench.h

Αρχικά ξεκινάμε με τις προσθήκες στο αρχείο bench.h , το header αρχείο της bench.

```
struct args {
    long int count;
    int r;
    int threadcount;
    DB* database;
};

long int foundall;
long int random_write_newcount;
double random_write_cost;
long int random_read_newcount;
int random_read_found;
double random_read_cost;
int helpingvar;
int helpingvar1;
pthread_mutex_t foundall_cal;
pthread_mutex_t write_cal;
```

Ορίζουμε ένα struct με όνομα args, το οποίο το χρειαζόμαστε για να προσθέσουμε ορίσματα στην κλήση των συναρτήσεων read_test και write_test, που βρίσκονται στο αρχείο kiwi.c, μέσω νημάτων. Στην συνέχεια, ορίζουμε κάποιες βοηθητικές μεταβλητές που χρειαζόμαστε στο αρχείο bench.c και στο kiwi.c. Δημιουργούμε την foundall, την οποία αρχικοποιούμε στην bench.c ως 0 και έπειτα της αλλάζει τιμές η kiwi.c, κρατάμε εκεί τον αριθμό όλων των επιτυχημένων διαβασμάτων(reads) για να το χρησιμοποιήσουμε ως στατιστικό. Οι επόμενες μεταβλητές που ακολουθούνε

(random_write_newcount, random_write_cost, random_read_newcount, random_read_found, random_read_cost, helpingvar και helpingvar1) τις δημιουργούμε για να κρατήσουμε στατιστικά για ένα τυχαίο read και ένα τυχαίο write.

3.1.2.bench.c

Εν συνεχεία, συνεχίζουμε στην main συνάρτηση εντός της bench.c. Αρχικά ορίζουμε και αρχικοποιούμε όλες τις μεταβλητές που θα μας χρειαστούνε.

```
pthread_mutex_init(&foundall_cal, NULL);
pthread_mutex_init(&write_cal, NULL);
int i;
long int count;
long long start,end,timespend;
long int threadcount;
int rate ;
double newrate;
long int writecount;
long int readcount;
long int writethreads;
long int readthreads;
random_write_newcount = 0;
random_write_cost = 0;
random_read_newcount = 0;
random_read_found = 0;
random_read_cost = 0;
helpingvar = 0;
helpingvar1 = 0;
foundall = 0;
DB* db;
```

Αρχικοποίηση όλων των βοηθητικών μεταβλητών που θα χρειαστούμε για τα στατιστικά ως 0, τα οποία μεταβάλλουμε εντός της kiwi.c. Ακολουθεί εξήγηση ύπαρξης των άλλων μεταβλητών.

- foundall_cal : mutex για την ασφαλή αλλαγή τιμής της foundall μεταβλητής.
- write_cal : mutex για την ασφαλή αλλαγή τιμής της helpingvar μεταβλητής.
- count : αριθμός συνολικών διαβασμάτων-εγγραφών, το οποίο παίρνουμε από τον χρήστη.
- start, end, timespend : μεταβλητές για την γνώση χρόνου εκτέλεσης μιας λειτουργίας.
- threadcount : συνολικός αριθμός νημάτων, το οποίο παίρνουμε από τον χρήστη.
- rate : ακέραιος αριθμός που προσδιορίζει το μίγμα λειτουργιών put και get.
- newrate : δεκαδικός αριθμός που κρατάει το rate.
- writecount : αριθμός εγγραφών
- readcount : αριθμός διαβασμάτων
- writethreads : αριθμός νημάτων που θα κάνουν εγγραφή
- readthreads : αριθμός νημάτων που θα διαβάζουν
- db : δήλωση της βάσης.

Δηλώνουμε την βάση εντός της main στο αρχείο bench.c, καθώς αλλάξαμε την υλοποίηση της kiwi.c.

```
if (argc < 4 && ((strcmp(argv[1], "write") == 0) || (strcmp(argv[1], "read") == 0))) {
    fprintf(stderr,"Usage: db-bench <write | read> <count> <count of threads> \n");
    exit(1);
}
if (argc < 5 && (strcmp(argv[1], "readwrite") == 0)) {
    fprintf(stderr,"Usage: db-bench <readwrite> <count> <count of threads> <rate of write [e.g 50 -for 50 percentage]> \n");
    exit(1);
}
```

Το παραπάνω μέρος του κώδικα υλοποιήθηκε ώστε σε περίπτωση εσφαλμένης εισαγωγής παραμέτρων στην γραμμή εντολών κατά την εκτέλεση του προγράμματος να τυπώνονται τα μηνύματα που

εξηγούν αναλυτικά στον χρήστη τι παραμέτρους περιμένει το πρόγραμμα για να τρέξει.

3.1.3.WRITE

Στην bench.c για να φτάσουμε στην σωστή κλήση της συνάρτησης (_write_test) που είναι υπεύθυνη για την εγγραφή, εκτελούμε τον παρακάτω κώδικα.

```
if (strcmp(argv[1], "write") == 0) {
    int r = 0;
    struct args a;                                          //declaring the struct we have on t
    count = atoi(argv[2]);
    threadcount = atoi(argv[3]);                            // take the number of threads
    pthread_t id[threadcount];                              // create the id for every thread
    _print_header(count);
    _print_environment();
    if (argc == 5)
        r = 1;
    db = db_open(DATAS);                                        // anoigw thn database prin kse
    a.count = count;
    a.r = r;
    a.threadcount = threadcount;
    a.database = db;                                            //passing values to our struct
    start = get_ustime_sec();
    for(i = 0; i<threadcount; i++){
        pthread_create(&id[i], NULL, _write_test, (void *) &a);
    }
    for(i = 0; i<threadcount; i++){
        pthread_join(id[i], NULL);
    }
    end = get_ustime_sec();
    timespend = end - start;                                    //time spent for writing.
    db_close(db);
    print_stats(0 , timespend, threadcount , 0, count ,0, 1);
    printf("|Random-Write thread(done:%ld): %.6f sec/op; %.1f writes/sec(estimated); thread cost:%.3f(sec);\n"
        ,random_write_newcount, (double)(random_write_cost / random_write_newcount)
        ,(double)(random_write_newcount / random_write_cost)
        ,random_write_cost);
    printf(LINE);
```

Δημιουργούμε ένα struct στο οποίο περνάμε τις τιμές που θέλουμε να δώσουμε στα ορίσματα της _write_test, τα οποία παίρνουμε από τον χρήστη. Αποθηκεύουμε στη τιμή count το 3ο όρισμα που μας δίνει ο χρήστης ως τον αριθμό των εγγραφών που θέλουμε να γίνουν και στη τιμή threadcount το 4ο όρισμα που μας δίνει ο χρήστης ως των αριθμό των νημάτων στα οποία θέλουμε να μοιράσουμε την εκτέλεση. Το pthread_t id[threadcount], ορίζει αναγνωριστικά για τα νήματα που θα δημιουργήσουμε. Το

άνοιγμα και το κλείσιμο της βάσης γινόταν εντός της kiwi.c, κάτι το οποίο αλλάξαμε και το κάνουμε στην bench.c. Ανοίγουμε την βάση πριν κάνουμε κάποια λειτουργία και την κλείνουμε όταν τελειώσουν όλες οι λειτουργίες μας. Αυτό το κάναμε γιατί θέλαμε να εκτελέσουμε τις συναρτήσεις διαβάσματος και εγγραφής παράλληλα, κάτι το οποίο μας απέτρεπε όταν κάθε συνάρτηση ανοιγόκλεινε την βάση σε κάθε κλήση της. Αυτό θα οδηγούσε σε σφάλμα επειδή μια συνάρτηση μπορεί να έκλεινε την βάση, ενώ μια άλλη βρισκόταν στο σημείο του γραψίματος και έβρισκε τη βάση κλειστή και ας έχει εκτελέσει το άνοιγμα της πιο πριν. Έτσι ανοίγουμε την βάση πριν την εκτέλεση των νημάτων εγγραφής, δίνουμε τιμές στα ορίσματα που θα δοθούν για την κλήση της συνάρτησης εγγραφής. Καλούμε την get_ustime_sec() πριν και μετά την εκτέλεση των νημάτων για να ξέρουμε πόσος χρόνος χρειάστηκε για την λειτουργία, τον οποίο αποθηκεύουμε στην μεταβλητή timespend. Με την εντολή pthread_create δημιουργούμε τα νήματα που εκτελούνε την _write_test και παίρνουνε ορίσματα τις τιμές του struct a. Έπειτα περιμένουμε να τελειώσουνε όλα τα νήματα με την εντολή pthread_join και κλείνουμε την βάση. Τέλος, καλούμε την συνάρτηση print_stats η οποία είναι υπεύθυνη για εκτύπωση των στατιστικών και θα την αναλύσουμε παρακάτω. Επίσης εκτυπώνουμε τα στατιστικά ενός random νήματος. Συγκεκριμένα, το πόσα γραψίματα έγιναν, πόσα στοιχεία γράφτηκαν, ο χρόνος ανά γράψιμο, γραψίματα ανά τον χρόνο και το κόστος του νήματος

3.1.4.READ

Ομοίως συνεχιζουμε και για την εκτέλεση των διαβασμάτων. Στην bench.c για να φτάσουμε στην σωστή κλήση της συνάρτησης (_read_test) που είναι υπεύθυνη για το διάβασμα, εκτελούμε τον παρακάτω κώδικα.

```
else if (strcmp(argv[1], "read") == 0) {
  int r = 0;
  struct args a;                  //declaring the struct we have on the header file here.
  count = atoi(argv[2]);
  threadcount = atoi(argv[3]);                // take the number of threads
  pthread_t id[threadcount];                  // create the id for every thread
  _print_header(count);
  _print_environment();
  if (argc == 5)
      r = 1;
  db = db_open(DATAS);
  a.count = count;
  a.r = r;
  a.threadcount = threadcount;
  a.database = db;                //passing values to our struct which are the arguments of our threads.
  start = get_ustime_sec();
  for(i = 0; i<threadcount; i++){
      pthread_create(&id[i], NULL, _read_test, (void *) &a);
  }
  for(i = 0; i<threadcount; i++){
      pthread_join(id[i], NULL);
  }
  end = get_ustime_sec();
  timespend = end - start;                //time spent for reading.
  db_close(db);
  print_stats(foundall , timespend, 0 , threadcount, 0 , count, 2);
  printf("|Random-Read thread(done:%ld, found:%d): %.6f sec/op; %.1f reads /sec(estimated); thread cost:%.3f(sec)\n",
      random_read_newcount, random_read_found,
      (double)(random_read_cost / random_read_newcount),
      (double)(random_read_newcount / random_read_cost),
      random_read_cost);
  printf(LINE);
```

Κάνουμε τα ίδια βήματα με την write, το μόνο που αλλάζει είναι η εκτέλεση της _read_test από τα νήματα αντί της _write_test, καθώς θέλουμε διάβασμα και όχι εγγραφή και η εκτύπωση της τιμής foundall για να εμφανιστεί ο αριθμός επιτυχημένων διαβασμάτων.

3.1.5.READWRITE

Στην bench.c για να φτάσουμε στην σωστή κλήση των συναρτήσεων εγγραφής και διαβάσματος, εκτελούμε τον παρακάτω κώδικα.

```
} else if (strcmp(argv[1], "readwrite") == 0) {
    int r = 0;
    struct args a;                  //declaring the struct we have on the header file here.
    struct args b;                  //declaring the struct we have on the header file here.
    count = atoi(argv[2]);
    threadcount = atoi(argv[3]);                // take the number of threads and make it integer
    rate = atoi(argv[4]);                       // taking the rate from the terminal on an integer
    newrate = ((double)rate/(double)100);           // make the integer rate (e.g 0,5 for 50)
    writecount = count*newrate;         // multiplying our count with the ratio
    readcount = count - writecount;
    writethreads = threadcount * newrate;       // number of threads for writing
    readthreads = threadcount - writethreads;   // number of threads for reading
    pthread_t id[threadcount];                  // create the id for every thread
    _print_header(count);
    _print_environment();
    if (argc == 6)
        r = 1;
    db = db_open(DATAS);                    // open the database before the writing starts
    a.count = writecount;               // passing values to the structs for both write and read test arguments needed f
    a.r = r;
    a.threadcount = writethreads;
    a.database = db;
    b.count = readcount;
    b.r = r;
    b.threadcount = readthreads;
    b.database = db;
    start = get_ustime_sec();
    for(i = 0; i<writethreads; i++){
        pthread_create(&id[i], NULL, _write_test, (void *) &a);
    }
    for(i = writethreads; i<threadcount; i++){
        pthread_create(&id[i], NULL, _read_test, (void *) &b);
    }
    for(i = 0; i<writethreads; i++){
        pthread_join(id[i], NULL);
    }
    for(i = writethreads; i<threadcount; i++){
        pthread_join(id[i], NULL);
    }
    end = get_ustime_sec();
    timespend = end - start;                                    //time spent for readwriting.

    db_close(db);
    print_stats(foundall , timespend, writethreads , readthreads, writecount , readcount, 3);       //created a function
    printf("Printing a Random-Write-Thread's stats:\n");
    printf("|Random-Write thread(done:%ld): %.6f sec/op; %.1f writes/sec(estimated); thread cost:%.3f(sec);\n"
        ,random_write_newcount, (double)(random_write_cost / random_write_newcount)
        ,(double)(random_write_newcount / random_write_cost)
        ,random_write_cost);
    printf(LINE);
    printf("Printing a Random-Read-Thread's stats:\n");
    printf("|Random-Read thread(done:%ld, found:%d): %.6f sec/op; %.1f reads /sec(estimated); thread cost:%.3f(sec)\n",
        random_read_newcount, random_read_found,
        (double)(random_read_cost / random_read_newcount),
        (double)(random_read_newcount / random_read_cost),
        random_read_cost);
    printf(LINE);
} else {
    fprintf(stderr,"Usage: db-bench <write | read> <count> <random>\n or Usage: db-bench <readwrite> <count> <count of t
    exit(1);
}

return 1;
```

Ο σκοπός αυτής της περίπτωσης είναι να μπορεί ο χρήστης να δώσει το ποσοστό των νημάτων που θα εκτελέσουν τη συνάρτηση _write_test και _read_test. Η διαφορά με τις δύο προηγούμενες κλήσεις είναι ότι κρατάμε στην μεταβλητή rate το 5^{ο} όρισμα που θα δώσει ο χρήστης η οποία κρατάει το ποσοστό. Η newrate μετατρέπει το ποσοστό σε δεκαδικό. Οι writecount και readcount κρατάνε τον αριθμό των εγγραφών και διαβασμάτων που θα γίνουν ανάλογα το ποσοστό που έχει δώσει ο χρήστης. Αντίστοιχα, οι writethreads και readthreads κρατάνε τον αριθμό των νημάτων για κάθε λειτουργία ανάλογα το ποσοστό που δίνει ο χρήστης. Έπειτα, περνάμε τις μεταβλητές στα structs για να τα περάσουμε ως παραμέτρους στις εκτελέσεις των συναρτήσεων. Στη συνέχεια, αντίστοιχα με τις προηγούμενες περιπτώσεις. Με την εντολή pthread_create δημιουργούμε τα νήματα που εκτελούνε την _write_test και παίρνουνε ορίσματα τις τιμές του struct a. Επίσης, με την εντολή pthread_create δημιουργούμε τα νήματα που εκτελούνε την _read_test και παίρνουνε ορίσματα τις τιμές του struct b. Έτσι επιτυγχάνουμε την παράλληλη εκτέλεση των νημάτων. Περιμένουμε να τελειώσουνε όλα τα νήματα με τις εντολές pthread_join. Τέλος, καλούμε την συνάρτηση print_stats η οποία είναι υπεύθυνη για εκτύπωση των στατιστικών και εκτυπώνουμε τα στατιστικά από ένα random-read νήμα και ένα random-write νήμα πριν κλείσουμε τη βάση.

3.2.1. KIWI.C – write_test

Αρχικά, τροποποιήσαμε τη συνάρτηση *_write_test όπως βλέπουμε παρακάτω:

```
void _write_test(long int count, int r)
```
→
```
void *_write_test(void *arg)
```

Για να μπορέσουμε να την καλέσουμε μέσω νημάτων.

Δημιουργούμε ένα καινούργιο struct για να πάρουμε τις τιμές που βρίσκονται στο struct που έχει δοθεί ως παράμετρος στη συνάρτηση *_write_test για την εκτέλεσή της.

```
struct args *values = (struct args *) arg;
long int count = values->count;
int r = values->r;
int threadcount = values->threadcount;
db = values->database;
```

Παρακάτω δημιουργούμε το newcount επειδή, θέλουμε να μοιράσουμε τον αριθμό των write, στη βάση, στα νήματά μας, για αυτό και το έχουμε ως άνω όριο στην ‘for’ μας. Δηλαδή, η for θα εκτελείται από κάθε νήμα newcount φορες.

```
long int newcount = count/threadcount;


start = get_ustime_sec();
for (i = 0; i < newcount; i++) {
```

Τέλος, επειδή τα νήματα μας τρέχουν ‘’παράλληλα’’ κλειδώνουμε αυτό το κομμάτι κώδικα ώστε να το χρησιμοποιεί μόνο ένα νήμα τη φορά, γιατί αυξάνουμε τη τιμή του helpingvar και αν δεν το κλειδώσουμε μπορούν διάφορα νήματα ταυτόχρονα να αλλάξουν τη μεταβλητή και στο τέλος να μην μπούμε ΠΟΤΕ στην if συνθήκη.

Η χρήση αυτής της if συνθήκης γίνεται, για να κρατήσουμε τις global μεταβλητές newcount και cost ενός random write νήματος, τα οποία θα τυπώσουμε στη main συνάρτηση.

```
pthread_mutex_lock(&write_cal);
helpingvar++;
if(helpingvar==1){
    random_write_newcount = newcount;
    random_write_cost = cost;
}
pthread_mutex_unlock(&write_cal);
return 0;
```

Αυτές τις μεταβλητές τις έχουμε δηλώσει στο αρχείο bench.h και για να τις διαχειριστούμε προσθέσαμε το παρακάτω include στο kiwi.c.

```
#include "bench.h"
```

Κλείνοντας, κάνουμε return 0 για την αντιμετώπιση του παρακάτω warning:

```
control reaches end of non-void function [-Wreturn-type]
```

3.2.2. KIWI.C – read_test

Αρχικά, τροποποιήσαμε τη συνάρτηση *_read_test όπως βλέπουμε παρακάτω:

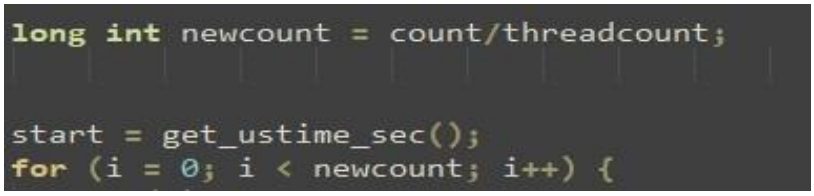
```
void *_read_test(void *arg)
```
→
```
void _read_test(long int count, int r)
```

Για να μπορέσουμε να την καλέσουμε μέσω νημάτων.

Δημιουργούμε ένα καινούργιο struct για να πάρουμε τις τιμές που βρίσκονται στο struct που έχει δοθεί ως παράμετρος στη συνάρτηση *_read_test για την εκτέλεσή της.

```
struct args *values = (struct args *) arg;
long int count = values->count;
int r = values->r;
int threadcount = values->threadcount;
db = values->database;
```

Παρακάτω δημιουργούμε το newcount επειδή, θέλουμε να μοιράσουμε τον αριθμό των read, στη βάση, στα νήματά μας, για αυτό και το έχουμε ως άνω όριο στην ‘for’ μας. Δηλαδή, η for θα εκτελείται από κάθε νήμα newcount φορες.



Τέλος, επειδή τα νήματα μας τρέχουν ‘’παράλληλα’’ κλειδώνουμε αυτό το κομμάτι κώδικα ώστε να το χρησιμοποιεί μόνο ένα νήμα τη φορά, γιατί αυξάνουμε τη τιμή του helpingvar1 και του foundall και επειδή, η διαδικασία πρόσθεσης(foundall = foundall + found) στη γλώσσα C είναι πχ:
temp0 = foundall;
temp0 = temp0 + found;
foundall = temp0;
εφόσον τα νήματα τρέχουν παράλληλα θέλουμε προστατευμένη αυτήν την περιοχή γιατί, αν δεν τη κλειδώσουμε μπορούν διάφορα νήματα ταυτόχρονα να αλλάξουν τις μεταβλητές οπότε στο τέλος να έχουμε λάθος αποτελέσματα και να μην μπούμε ΠΟΤΕ στην if συνθήκη.
Κάθε νήμα, μόλις τελειώσουν τα διαβάσματα, προσθέτει τα επιτυχημένα στο foundall.
Η χρήση αυτής της if συνθήκης γίνεται, για να κρατήσουμε τις global μεταβλητές newcount, cost και found ενός random read νήματος, τα οποία θα τυπώσουμε στη main συνάρτηση.

```
pthread_mutex_lock(&foundall_cal);

foundall = foundall + found;
helpingvar1++;
if(helpingvar1 == 1){
    random_read_newcount = newcount;
    random_read_found = found;
    random_read_cost = cost;
}
pthread_mutex_unlock(&foundall_cal);
return 0;
```

Αυτές τις μεταβλητές τις έχουμε δηλώσει στο αρχείο bench.h και για να τις διαχειριστούμε προσθέσαμε το παρακάτω include στο kiwi.c.

```
#include "bench.h"
```

Κλείνοντας, κάνουμε return 0 για την αντιμετώπιση του παρακάτω warning:

control reaches end of non-void function [-Wreturn-type]

ENGINE

Παρακάτω θα αναλύσουμε τις αλλαγές που κάναμε σε διάφορα αρχεία του directory: engine

Ο στόχος μας σε αυτό το κομμάτι είναι να υλοποιήσουμε το λεγόμενο: πολλοί αναγνώστες-ένας γραφέας. Δηλαδή να μπορεί να γίνεται ένα γράψιμο(write), μία χρονική στιγμή, από οποιοδήποτε νήμα αλλά, πολλαπλά νήματα να μπορούν να διαβάσουν(read) από τη βάση την ίδια χρονική στιγμή. Βέβαια, σε περίπτωση που γίνεται ένα γράψιμο(write) στη βάση, δεν μπορεί ταυτόχρονα να γίνεται ένα ή περισσότερα διάβασματα και αντίστοιχα, όταν γίνονται ένα ή περισσότερα διαβάσματα(reads) να μην μπορεί να γίνει ένα γράψιμο.

4.1.db.h

```
pthread_cond_t readerexists;
pthread_mutex_t operation;
pthread_cond_t writerexists;
bool flag;
int readcount;
```

Ξεκινάμε με τις δηλώσεις των παραπάνω μεταβλητών στο αρχείο db.h , το header αρχείο του db.c.

- pthread_cond_t readerexists : condition το οποίο χρησιμοποιούμε για να δώσουμε σήμα στον writer ότι, εκείνη τη χρονική στιγμή, δεν υπάρχει κάποιος reader που διαβάζει από τη βάση.
- pthread_cond_t writerrexists : condition το οποίο χρησιμοποιούμε για να δώσουμε σήμα στον reader ότι, εκείνη τη χρονική στιγμή, δεν υπάρχει κάποιος writer που γράφει στη βάση.

- pthread_mutex_t operation : mutex για να γίνεται αμοιβαίος αποκλεισμός σε ορισμένα κομμάτια των συναρτήσεων db_add και db_get.
- bool flag : δήλωση global boolean μεταβλητής, με include <stdbool.h>, η οποία μας υποδεικνύει εάν βρισκόμαστε εν ώρα γραφείς.
- int readcount: μετρητής αναγνωστών(readers).

4.2. db.c – DB* db_open_ex

Στην παρακάτω συνάρτηση, του αρχείου db.c, αρχικοποιούμε τις μεταβλητές που δηλώσαμε στο header αρχείο(db.h). Συγκεκριμένα:

- pthread_cond_init(&readerexists, NULL);
- pthread_cond_init(&writerexists, NULL);
- pthread_mutex_init(&operation, NULL);
- flag = false;
- readcount = 0;

```
DB* db_open_ex(const char* basedir, uint64_t cache_size)
{
    pthread_cond_init(&readerexists, NULL);
    pthread_cond_init(&writerexists, NULL);
    pthread_mutex_init(&operation, NULL);
    flag = false;
    readcount = 0;
    DB* self = calloc(1, sizeof(DB));

    if (!self)
        PANIC("NULL allocation");

    strncpy(self->basedir, basedir, MAX_FILENAME);
    self->sst = sst_new(basedir, cache_size);

    Log* log = log_new(self->sst->basedir);
    self->memtable = memtable_new(log);

    return self;
}
```

4.3.db.c – db_add

Σε αυτό το κομμάτι θα αναλύσουμε το μέρος του Αλγορίθμου που αφορά τους Γραφείς (Writers). Η παρακάτω συνάρτηση καλείται στο αρχείο kiwi.c(bench) και βρίσκεται στο αρχείο db.c(engine). Ξεκινάμε κλειδώνοντας το mutex με κλειδαριά &operation, αυτό το κάνουμε ώστε να έχουμε μόνο ΕΝΑ νήμα, εκείνη τη χρονική στιγμή το οποίο θα κάνει write. Με την ίδια κλειδαριά, κλειδώνουμε και στην αρχή της db_get συνάρτησης, η οποία είναι υπεύθυνη για το διάβασμα(read) από την βάση γιατί όταν ένα νήμα γράφει στη βάση, δεν θέλουμε ένα άλλο να διαβάσει ταυτόχρονα. Συνεχίζουμε προσθέτοντας ένα while condition το οποίο ελέγχει εάν υπάρχουν ένας η περισσότεροι αναγνώστες(readers) εκείνη τη στιγμή. Αν υπάρχουν, περιμένουμε(cond_wait), να δοθεί σήμα(cond_signal) από τους αναγνώστες(readers-συνάρτηση db_get) για να συνεχίσει. Αλλάζουμε τη τιμή του flag σε true για να δείξουμε ότι ένα νήμα βρίσκεται σε διαδικασία γραψίματος(write). Ύστερα έχουμε την αντιμετώπιση του merge. Η διαδικασία κατά την οποίο γεμίζει το memtable με ζευγάρια από κλειδιά-τιμές και συγχωνεύεται σε κάποιο από τα αρχεία sst είναι μια εγγραφή, δηλαδή εδώ έχουμε ένα γραφέα. Έπειτα, κρατάμε σε μία μεταβλητή την επιστροφή της memtable_add(), γιατί θέλουμε να κρατήσουμε τη λειτουργία της στη κρίσιμη περιοχή και την επιστρέφουμε στο τέλος της db_add. Τέλος, ενημερώνουμε(flag = false) τους αναγνώστες(readers) ότι δεν υπάρχει γραφέας αυτή τη στιγμή και στη συνέχεια, δίνουμε σήμα(cond_signal) σε τυχόν αναγνώστες που βρίσκονται σε αναμονή, ότι τελείωσε ο τρέχον γραφέας. Κλείνοντας, ξεκλειδώνουμε τη κλειδαριά(&operation) ώστε να κλειδώσει η επόμενη λειτουργία που βρίσκεται σε αναμονή και επιστρέφουμε

το rtrnvalue εκτός της κρίσιμης περιοχής επειδή, πλέον δε θα μπορεί να το επηρεάσει ένα άλλο νήμα που να λειτουργεί παράλληλα.

```
int db_add(DB* self, Variant* key, Variant* value)
{
    pthread_mutex_lock(&operation);
    while(readcount >=1){
        pthread_cond_wait(&readerexists,&operation);
    }
    flag = true;
    if (memtable_needs_compaction(self->memtable))
    {
        INFO("Starting compaction of the memtable after %d insertions
             self->memtable->add_count, self->memtable->del_count);
        sst_merge(self->sst, self->memtable);
        memtable_reset(self->memtable);
    }
    int rtrnvalue = memtable_add(self->memtable, key, value);
    flag = false;
    pthread_cond_signal(&writerexists);
    pthread_mutex_unlock(&operation);
    return rtrnvalue;
}
```

4.4.db.c – db_get

Σε αυτό το κομμάτι θα αναλύσουμε το μέρος του Αλγορίθμου που αφορά τους Αναγνώστες (Readers). Η παρακάτω συνάρτηση καλείται στο αρχείο kiwi.c(bench) και βρίσκεται στο αρχείο db.c(engine). Ξεκινάμε κλειδώνοντας το mutex με κλειδαριά &operation, αυτό το κάνουμε ώστε να έχουμε μόνο ένα νήμα που θα μπαίνει να εξετάζει αν γίνεται εκείνη την χρονική στιγμή εγγραφή. Αν γίνεται εγγραφή, το ελέγχουμε με την εντολή της while(flag==true), καθώς το flag όταν είναι true μας λέει ότι γίνεται εγγραφή στην βάση, οπότε αν είναι true περιμένουμε την εντολή pthread_cond_wait με παραμέτρους το writerexists που δίνει σήμα, στο τέλος της db_add, ότι έχει τελειώσει ένας γραφέας το γράψιμο.

Στην συνέχεια, αφού έχει τελειώσει η εγγραφή ή δεν υπάρχει σε περίπτωση που δεν μπούμε στο while loop, ανεβάζουμε την μεταβλητή readcount κατά ένα, η οποία προσδιορίζει το πόσοι αναγνώστες διαβάζουν από την βάση την συγκεκριμένη χρονική στιγμή. Ξεκλειδώνουμε την κλειδαριά και κάνουμε την ανάγνωση, η οποία πάει στην βάση δεδομένων, ψάχνει το κλειδί που θέλει για να βρει την τιμή. Χρησιμοποιούμε την μεταβλητή rtrnvalue για να κρατήσουμε το αποτέλεσμα της ανάγνωσης και να το επιστρέψουμε στο τέλος της συνάρτησης, γιατί δεν θέλουμε να τελειώσει εδώ αφού πρέπει ακόμα να μειώσουμε τον counter(readcount) των αναγνωστών. Στη συνέχεια, σε αυτό το σημείο επιτυγχάνουμε το ζητούμενο πολλοί αναγνώστες. βρισκόμαστε εκτός κρίσιμης περιοχής χωρίς κλειδαριές, γιατί θέλουμε τα διαβάσματα(reads) να γίνονται παράλληλα ,να έχουμε δηλαδή την δυνατότητα να υπάρχουν πολλοί αναγνώστες την ίδια χρονική στιγμή. Έπειτα, κλειδώνουμε πάλι την κλειδαριά &operation, καθώς θα μεταβάλλουμε ξανά την τιμή του readcount, αυτήν την φορά θα αφαιρέσουμε ένα από το readcount για να δείξουμε ότι το συγκεκριμένο νήμα τελείωσε την ανάγνωση από την βάση. Ακόμα, εάν δεν έχουμε κάποιον αναγνώστη εκείνη την χρονική στιγμή, ενημερώνουμε μέσω της pthread_cond_signal(&readerexists) τους γραφείς που περιμένουν, εντός της while στην db_add, να συνεχίσουνε την εκτέλεση(«ξυπνάμε» τους γραφείς όταν δεν έχουμε ενεργό αναγνώστη, καθώς όταν γίνεται εγγραφή στην βάση μας δεν πρέπει να γίνεται και διάβασμα ταυτόχρονα). Τέλος, ξεκλειδώνουμε την κλειδαριά και επιστρέφουμε το rtrnvalue.

```
int db_get(DB* self, Variant* key, Variant* value)
{
    int rtrnvalue;
    pthread_mutex_lock(&operation);
    while(flag==true){
        pthread_cond_wait(&writerexists,&operation);
    }
    readcount++;
    pthread_mutex_unlock(&operation);
    if (memtable_get(self->memtable->list, key, value) == 1){
        rtrnvalue = 1;
    }else{


        rtrnvalue = sst_get(self->sst, key, value);
    }
    pthread_mutex_lock(&operation);
    readcount--;
    if(readcount == 0){
        pthread_cond_signal(&readerexists);
    }
    pthread_mutex_unlock(&operation);
    return rtrnvalue;
}
```

5.ΕΚΤΕΛΕΣΗ

Σε αυτό το κομμάτι θα αναλύσουμε τα βήματα που πρέπει να ακολουθήσει ο χρήστης για την εκτέλεση το προγράμματος, ανάλογα με τον τρόπο που θέλει να επεξεργαστεί τη βάση δεδομένων.

Αρχικά, πρέπει να μεταφερθούμε στο directory όπου δημιουργούμε τα εκτελέσιμα του προγράμματος μας, οπότε ακολουθούμε τα παρακάτω βήματα στο terminal:

1->
```
myy601@myy601lab1:~$ cd kiwi
```

2->
```
myy601@myy601lab1:~/kiwi$ cd kiwi-source
```

3->

```
myy601@myy601lab1:~/kiwi/kiwi-source$ make clean
cd engine && make clean
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/engine'
rm -rf *.o libindexer.a
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/engine'
cd bench && make clean
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/bench'
rm -f kiwi-bench
rm -rf testdb
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/bench'
```

Σε αυτό το σημείο με την εντολή ‘make clean’ διαγράφουμε οποιαδήποτε τυχόν υπάρχοντα εκτελέσιμα αρχεία και τον κατάλογο testdb όπου βρίσκονται τυχόν αποθηκευμένα δεδομένα από προηγούμενη εκτέλεση. Αυτή τη διαδικασία πρέπει να επιστρέφουμε στο συγκεκριμένο directory και πρέπει να επαναλαμβάνεται κάθε φορά που θέλουμε να τρέξουμε το πρόγραμμα από την αρχή.

Στη συνέχεια κάνουμε ‘make all’ για να δημιουργηθούν τα εκτελέσιμα.

4->

```
myy601@myy601lab1:~/kiwi/kiwi-source$ make all
cd engine && make all
make[1]: Entering directory '/home/myy601/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/kiwi/kiwi-source/engine'
cd bench && make all
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/bench'
gcc -g -ggdb -Wall -Wno-implicit-function-declaration -Wno-unused-but-set-variable bench.c kiwi.c -L ../engine -lindexer -lpthread -lsn
appy -o kiwi-bench
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/bench'
```

5->`myy601@myy601lab1:~/kiwi/kiwi-source$ cd bench`

Με το που δημιουργηθούν τα εκτελέσιμα μεταφερόμαστε στο directory bench που περιέχει το benchmark.

Τέλος, τρέχουμε το πρόγραμμά μας ανάλογα με τη λειτουργία που επιθυμούμε.

6.1->`myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench write 100000 10`

Με το 2ο όρισμα ‘write’ υποδεικνύουμε ότι θέλουμε να γράψουμε στη βάση, με το 3ο όρισμα καθορίζουμε τον αριθμό των στοιχείων που θα εγγραφούν στη βάση και με το 4ο όρισμα καθορίζουμε τον αριθμό των νημάτων που θα μοιραστούν τον αριθμό των εγγραφών.

6.2->`myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench read 100000 10`

Ομοίως με παραπάνω, με το 2ο όρισμα ‘read’ υποδεικνύουμε ότι θέλουμε να διαβάσουμε από τη βάση, με το 3ο όρισμα καθορίζουμε τον αριθμό των στοιχείων που θα διαβαστούν από τη βάση και με το 4ο όρισμα καθορίζουμε τον αριθμό των νημάτων που θα μοιραστούν τον αριθμό των διαβασμάτων.

7->`myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 100000 10 50`

Στη περίπτωση που θέλουμε να γίνονται ποσοστιαία γραψίματα και διαβάσματα στη βάση, παράλληλα, με το 2ο όρισμα ‘readwrite’ υποδεικνύουμε ότι θέλουμε να διαβάσουμε και να γράψουμε στη βάση, με το 3ο όρισμα καθορίζουμε τον αριθμό των στοιχείων που θα διαβαστούν και εγγραφούν στη βάση, με το 4ο όρισμα καθορίζουμε τον αριθμό των νημάτων που θα μοιραστούν τον αριθμό των λειτουργιών και το 5ο όρισμα καθορίζουμε το ποσοστό το νημάτων που δηλώσαμε στο προηγούμενο όρισμα τα οποία θα κάνουν λειτουργία γραψίματος(πχ 50 = 50%). Αντίστοιχα, το υπόλοιπο ποσοστό θα κάνει λειτουργία διαβάσματος.

6.ΣΤΑΤΙΣΤΙΚΑ

Στο κομμάτι των στατιστικών θα αναλύσουμε τρεις περιπτώσεις εκτέλεσης και θα συγκρίνουμε τα αποτελέσματα εκτελέσεων με διαφορετικές μεταβλητές για να αποδείξουμε ότι με τη χρήση νημάτων αλλάζει ο χρόνος εκτέλεσης. Επίσης, θα αποδείξουμε ότι με την υλοποίηση μας, οι λειτουργίες γίνονται παράλληλα.

write

Ξεκινάμε με την πρώτη περίπτωση όπου θα κάνουμε μόνο γράψιμο στη βάση:
Για την εκτέλεση 1.000.000 γραψιμάτων με 1 νήμα,

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench write 1000000 1
```

Έχουμε τα παρακάτω στατιστικά:

```
All threads are finished after: 15.000000 seconds
|Writes (done:1000000): 0.000015 sec/op; 66666.7 writes/sec(estimated);cost:15.000(sec);
|Random-Write thread(done:1000000): 0.000015 sec/op; 66666.7 writes/sec(estimated); thread cost:15.000(sec);
```

Παρατηρούμε ότι όλη η εκτέλεση πήρε 15sec, έγιναν 1.000.000 writes, κάθε write έκανε 0,000015sec να ολοκληρωθεί, έγιναν 66666,7 writes ανά δευτερόλεπτο και ένα random νήμα(μοναδικό σε αυτή τη περίπτωση) έκανε τους ίδιους χρόνους με παραπάνω εφόσον ήταν το μόνο που δημιουργήθηκε.

Θα συγκρίνουμε τώρα τους χρόνους της ίδιας εκτέλεσης αλλά με 10 νήματα και με 50 νήματα παρακάτω:
10 νήματα->

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench write 1000000 10
All threads are finished after: 9.000000 seconds
|Writes (done:1000000): 0.000009 sec/op; 111111.1 writes/sec(estimated);cost:9.000(sec);
|Random-Write thread(done:100000): 0.000070 sec/op; 14285.7 writes/sec(estimated); thread cost:7.000(sec);
```

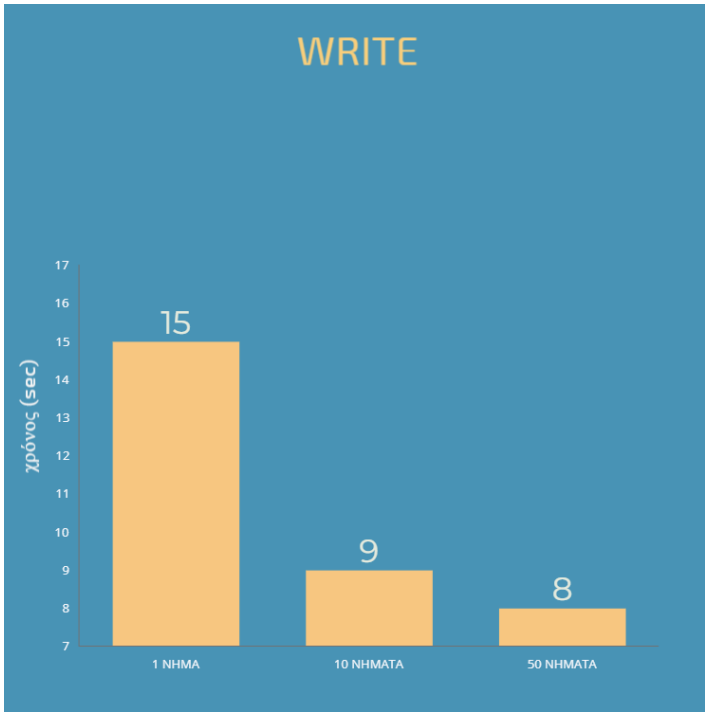
50 νήματα->

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench write 1000000 50
All threads are finished after: 8.000000 seconds
|Writes (done:1000000): 0.000008 sec/op; 125000.0 writes/sec(estimated);cost:8.000(sec);
|Random-Write thread(done:20000): 0.000300 sec/op; 3333.3 writes/sec(estimated); thread cost:6.000(sec);
```

Παρατηρούμε ότι όσο αυξάνεται ο αριθμός των νημάτων μειώνεται ο χρόνος εκτέλεσης του προγράμματος. Συγκεκριμένα:

- 1 νήμα -> 15sec, 0.000015 sec/op, 66666.7 writes/sec
- 10 νήματα -> 9sec, 0.000009 sec/op, 111111.1 writes/sec
- 50 νήματα -> 8sec, 0.000008 sec/op, 125000 writes/sec

Στην περίπτωση όμως, του γραψίματος, οι λειτουργίες write στη βάση, δε μπορούν να γίνονται ταυτόχρονα για αυτό το λόγο βρίσκονται και σε locked περιοχή. Οπότε ο χρόνος που μειώνεται οφείλεται στο ότι τα πολλά νήματα είναι έτοιμα να κάνουν τη λειτουργία write και απλά περιμένουν έξω από τη κρίσιμη περιοχή μέχρι να τους δοθεί σήμα από την προηγούμενη λειτουργία. Έτσι, δε χάνεται χρόνος για να ξανά ξεκινήσει όλη η διαδικασία γραψίματος από την αρχή. Παρακάτω βλέπουμε και ένα διάγραμμα που μας δείχνει τις διαφορές:



read

Συνεχίζουμε με την δεύτερη περίπτωση όπου θα κάνουμε μόνο διάβασμα από τη βάση:

Για την εκτέλεση 1.000.000 γραψιμάτων με 1 νήμα:

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench read 1000000 1
```

Έχουμε τα παρακάτω στατιστικά:

```
All threads are finished after: 12.000000 seconds
|Reads (done:1000000, found:1000000): 0.000012 sec/op; 83333.3 reads /sec(estimated); cost:12.000(sec)
|Random-Read thread(done:1000000, found:1000000): 0.000012 sec/op; 83333.3 reads /sec(estimated); thread cost:12.000(sec)
```

Παρατηρούμε ότι όλη η εκτέλεση πήρε 12sec, έγιναν 1.000.000 reads, κάθε read έκανε 0,000012sec να ολοκληρωθεί, έγιναν 83333,3 reads ανά δευτερόλεπτο και ένα random νήμα(μοναδικό σε αυτή τη περίπτωση) έκανε τους ίδιους χρόνους με παραπάνω εφόσον ήταν το μόνο που δημιουργήθηκε.

Θα συγκρίνουμε τώρα τους χρόνους της ίδιας εκτέλεσης αλλά με 10 νήματα και με 50 νήματα παρακάτω:

10 νήματα->

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench read 1000000 10
All threads are finished after: 4.000000 seconds
|Reads (done:1000000, found:1000000): 0.000004 sec/op; 250000.0 reads /sec(estimated); cost:4.000(sec)
|Random-Read thread(done:100000, found:100000): 0.000040 sec/op; 25000.0 reads /sec(estimated); thread cost:4.000(sec)
```

50 νήματα->

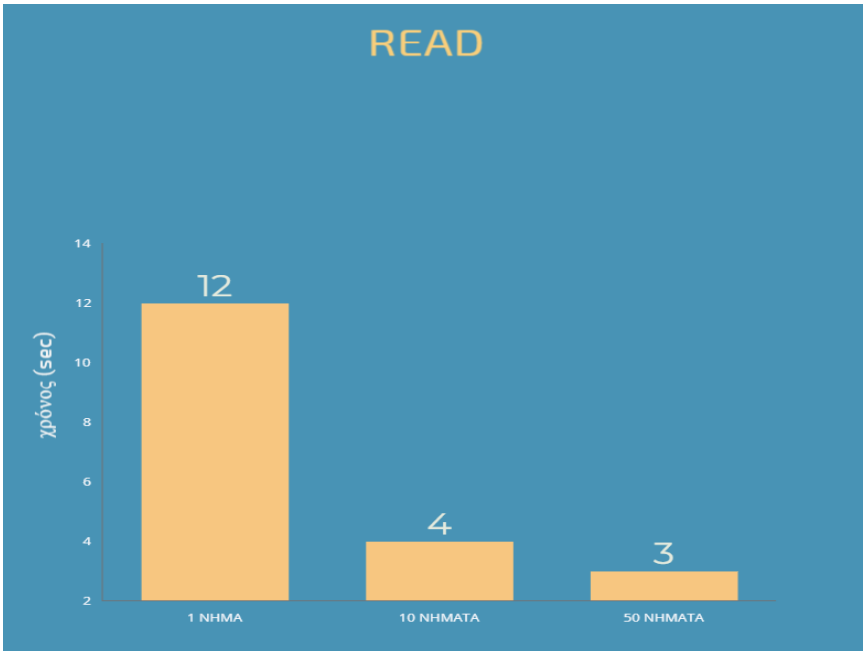
```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench read 1000000 50
All threads are finished after: 3.000000 seconds
|Reads (done:1000000, found:1000000): 0.000003 sec/op; 333333.3 reads /sec(estimated); cost:3.000(sec)
|Random-Read thread(done:20000, found:20000): 0.000150 sec/op; 6666.7 reads /sec(estimated); thread cost:3.000(sec)
```

Παρατηρούμε ότι όσο αυξάνεται ο αριθμός των νημάτων μειώνεται πολύ ο χρόνος εκτέλεσης του προγράμματος. Συγκεκριμένα:

- 1 νήμα -> 12sec, 0.000012 sec/op, 83333.3 reads/sec

- 10 νήματα -> 4sec, 0.000004 sec/op, 250.000 reads/sec
- 50 νήματα -> 3sec, 0.000003 sec/op, 333.333.3 reads/sec

Σε αυτή την περίπτωση, του διαβάσματος, οι λειτουργίες read από τη βάση, μπορούν να γίνονται ταυτόχρονα για αυτό το λόγο δε βρίσκονται και σε locked περιοχή. Οπότε, ο χρόνος μειώνεται δραστικά με τη χρήση νηματικής υλοποίησης.
Παρακάτω βλέπουμε και ένα διάγραμμα που μας δείχνει τις διαφορές:



Readwrite

Τρίτη και τελευταία περίπτωση όπου μπορούν να γίνονται παράλληλα γραψίματα και διαβάσματα με ποσοστό, από τη βάση. Να αναφέρουμε εδώ, πως όταν γίνεται διάβασμα από τη βάση υπάρχουν περιπτώσεις όπου δε θα βρεθεί κάτι να διαβαστεί. Αυτό οφείλεται στο ότι οι εγγραφές και τα διαβάσματα στη βάση γίνονται παράλληλα οπότε, τη στιγμή που πάει να γίνει διάβασμα μπορεί να μην υπάρχει εγγραφή με αυτό το κλειδί με αποτέλεσμα

να μην επιστραφεί τιμή(value) αλλά, να εμφανιστεί μήνυμα ‘not found’. Αυτό δεν είναι πρόβλημα, ίσα ίσα, αυτός είναι άλλος ένας παράγοντας που δείχνει τη παραλληλία(ένας γραφέας-πολλοί αναγνώστες) στην πολυνηματική υλοποίηση μας.

Για την εκτέλεση 1.000.000 γραψιμάτων και διαβασμάτων με 30 νήματα και ποσοστό 10% για write νήματα(90% για read νήματα) δηλαδή, 3 νήματα για write και 27 για read:
10%(write) & 90%(read) (3 write νήματα & 27 read νήματα)->

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 1000000 30 10
```

Έχουμε τα παρακάτω στατιστικά:

```
All threads are finished after: 4.000000 seconds
|Writes (done:100000): 0.000040 sec/op; 25000.0 writes/sec(estimated);cost:4.000(sec);
|Reads (done:900000, found:807465): 0.000004 sec/op; 225000.0 reads /sec(estimated); cost:4.000(sec)
Printing a Random-Write-Thread's stats:
|Random-Write thread(done:33333): 0.000120 sec/op; 8333.2 writes/sec(estimated); thread cost:4.000(sec);
Printing a Random-Read-Thread's stats:
|Random-Read thread(done:33333, found:30711): 0.000120 sec/op; 8333.2 reads /sec(estimated); thread cost:4.000(sec)
```

Παρατηρούμε ότι όλη η εκτέλεση πήρε 4sec, έγιναν 100.000 writes και 900.000 reads, κάθε write έκανε 0,000040sec και κάθε read 0,000004sec να ολοκληρωθεί, έγιναν 25000 writes ανά δευτερόλεπτο και 225.000 reads. Ένα random write νήμα έκανε 33333 εγγραφές, 0,000120 ανά εγγραφή, 8333.2 εγγραφές ανά δευτερόλεπτο. Ένα random read νήμα έκανε 33333 διαβάσματα με 30711 από αυτά να βρήκαν κάτι στη βάση, 0,000120 ανά εγγραφή, 8333.2 εγγραφές ανά δευτερόλεπτο.

Θα συγκρίνουμε τώρα τους χρόνους της ίδιας εκτέλεσης αλλά με 50%(write) νήματα και με 50%(read) νήματα και
90%(write) νήματα και με 10%(read) νήματα παρακάτω:

50%(write) & 50%(write) (15 write νήματα & 15 read νήματα)->

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 1000000 30 50
+-----------------------------+---------------+-----------------------------+------------------+
All threads are finished after: 7.000000 seconds
+-----------------------------+---------------+-----------------------------+------------------+
+-----------------------------+---------------+-----------------------------+------------------+
|Writes (done:500000): 0.000014 sec/op; 71428.6 writes/sec(estimated);cost:7.000(sec);
+-----------------------------+---------------+-----------------------------+------------------+
|Reads (done:500000, found:485707): 0.000014 sec/op; 71428.6 reads /sec(estimated); cost:7.000(sec)
+-----------------------------+---------------+-----------------------------+------------------+
Printing a Random-Write-Thread's stats:
|Random-Write thread(done:33333): 0.000180 sec/op; 5555.5 writes/sec(estimated); thread cost:6.000(sec);
+-----------------------------+---------------+-----------------------------+------------------+
Printing a Random-Read-Thread's stats:
|Random-Read thread(done:33333, found:31624): 0.000180 sec/op; 5555.5 reads /sec(estimated); thread cost:6.000(sec)
+-----------------------------+---------------+-----------------------------+------------------+
```

90%(write) & 10%(write) (27 write νήματα & 3 read νήματα)->

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 1000000 30 90
+-----------------------------+---------------+-----------------------------+------------------+
All threads are finished after: 9.000000 seconds
+-----------------------------+---------------+-----------------------------+------------------+
+-----------------------------+---------------+-----------------------------+------------------+
|Writes (done:900000): 0.000010 sec/op; 100000.0 writes/sec(estimated);cost:9.000(sec);
+-----------------------------+---------------+-----------------------------+------------------+
|Reads (done:100000, found:79421): 0.000090 sec/op; 11111.1 reads /sec(estimated); cost:9.000(sec)
+-----------------------------+---------------+-----------------------------+------------------+
Printing a Random-Write-Thread's stats:
|Random-Write thread(done:33333): 0.000180 sec/op; 5555.5 writes/sec(estimated); thread cost:6.000(sec);
+-----------------------------+---------------+-----------------------------+------------------+
Printing a Random-Read-Thread's stats:
|Random-Read thread(done:33333, found:27048): 0.000180 sec/op; 5555.5 reads /sec(estimated); thread cost:6.000(sec)
+-----------------------------+---------------+-----------------------------+------------------+
```

Παρατηρούμε ότι όσο αυξάνεται το ποσοστό των νημάτων που κάνουν τη λειτουργία γραψίματος(write), αυξάνεται ο χρόνος εκτέλεσης του προγράμματος. Συγκεκριμένα:

- 10%(write) -> 4sec
  -writes-> 0.000040 sec/op, 25.000 writes/sec
  -reads-> found 807.465, 0.000004 sec/op 225.000 reads/sec
- 50%(write) -> 7sec
  -writes-> 0.000014 sec/op, 71.428.6 writes/sec
  -reads-> found 485.707, 0.000014 sec/op, 71.428.6 writes/sec
- 90%(write) -> 9sec
  -writes-> 0.000010 sec/op, 100.000 writes/sec
  -reads-> found 79.421, 0.000090 sec/op, 11111.1 writes/sec

Αυτό, όπως εξηγήσαμε και στην ανάλυση του κώδικα παραπάνω, οφείλεται στο ότι οι λειτουργίες γραψίματος παίρνουν

περισσότερο χρόνο εφόσον πρέπει να γίνονται μία μία οπότε, όσο αναθέτουμε περισσότερα νήματα για να κάνουν γράψιμο τόσο λιγότερα νήματα αναθέτουμε για να κάνουν διάβασμα τα οποία μπορούν να γίνονται παράλληλα και να μην περιμένουν να τελειώσει το προηγούμενο.
Παρακάτω βλέπουμε και ένα διάγραμμα που μας δείχνει τις διαφορές: