

ΛΕΙΤΟΥΡΓΙΚΑ 2021

ΧΑΡΙΣΗΣ ΑΛΕΞΑΝΔΡΟΣ 3361

ΜΕΡΟΛΛΙ ΑΓΓΕΛΕ 3278

Περιεχόμενα:

• Ανάλυση Κώδικα κι αλλαγών	2
❖ Bench.h	2
❖ Kiwi.c	3
❖ Bench.c	5
▪ write	6
▪ read	8
▪ readwrite	9
❖ db.h	12
❖ db.c	12
• Αποτελέσματα (Φωτογραφίες)	15
• Προβλήματα	19

Αρχικά ας αναλύσουμε τις αλλαγές που έγιναν στον φάκελο kiwi-source\bench, και συγκεκριμένα στα αρχεία bench.c, bench.h και kiwi.c.

Πριν αναφέρουμε λεπτομερώς κάθε αλλαγή και γιατί έγινε, θα σας δώσουμε μία μικρή περίληψη της σκεπτικής από πίσω. Θέλωντας να κάνουμε πολυνηματική τη υλοποίησή μας, καταλήξαμε πως το DB πρέπει να ανοίγει μόνο μία φορά, πριν την δημιουργία των νημάτων, οπότε κάναμε τις κατάλληλες αλλαγές. Επίσης ορίσαμε στο αρχείο bench.h διάφορες "global" μεταβλητές και #includes, ώστε ό,τι είναι κοινότυπο να υιοθετείται από τα άλλα δύο αρχεία, που τώρα πια κάνουν μόνο #include "bench.h" .

Ας δούμε τις αλλαγές σε βάθος:

Bench.h:

Προσθήσαμε τα `#include "../engine/db.h"`

διότι και τα δύο αρχεία που την υιοθετούν χρησιμοποιούν το DB.

Έπειτα `pthread_mutex_t write_lock;` δύο mutexes τα οποία χρησιμοποιούνται στο kiwi.c
`pthread_mutex_t read_lock;` και αρχικοποιούνται στο bench.c

Τα υπόλοιπα είναι *globals*:

`DB * db;` Η βάση που χρησιμοποιείται και στα δύο αρχεία.

`double totalReadTime;` Μεταβλητές που βοηθάνε στην ακριβής μέτρηση του συνολικού
`double totalWriteTime;` κόστους χρόνου.

`long int writeCount;` Μεταβλητές για την εκτύπωση των σωστών μεταβλητών σε κάθε
`int totalFound;` τελικό print.
`long int readCount;`

kiwi.c:

Αφαιρέσαμε τα `#include <string.h>` και `#include "../engine/db.h"` , διότι πια υιοθετούνται από το `bench.c`.

Μεταφέραμε το `#define DATAS ("testdb")` στο `bench.c` επειδή από εκεί πια ανοίγουμε το αρχείο.

Πάμε στη μέθοδο `_write_test`:

Αφαιρέσαμε τα `double cost;` Διότι έχουμε ήδη μία μεταβλητή στο `bench.h` και τα
και τα περνάμε αμέσως εκεί.
`DB* db;` Επειδή έχουμε ήδη τη μεταβλητή στο `bench.h`.

`db = db_open(DATAS);` Μεταφέρθηκαν όπως και είπαμε σαν λειτουργίες στο
`db_close(db);` `bench.c`.

Note: Δε πειράξαμε/αλλάξαμε τίποτα ενδιάμεσα στα δύο παραπάνω.

Έπειτα

`cost = end - start;`

Το μετατρέψαμε σε

`totalWriteTime = end - start;` και περνάμε εδώ τον χρόνο πια.

Και προσθέσαμε

`writeCount += count;` Μεταβλητή που μετράει τα συνολικά `count` για το `write`.

Καθώς και

`pthread_mutex_lock(&write_lock);`

`pthread_mutex_unlock(&write_lock);`

Περικλείωντας τα δύο που αναφέραμε ακριβώς από επάνω, ώστε να μη τα πειράζουν πολλαπλά νήματα ταυτόχρονα. Το `totalWriteTime` και το `writeCount` πρέπει να αλλάζουν μόνο από ένα `write` νήμα τη κάθε στιγμή.

Ομοίως έχουμε αλλάξει **σχεδόν ακριβώς** τα ίδια και στην `_read_test`:

Αφαιρέσαμε τα `double cost;` Διότι έχουμε ήδη μία μεταβλητή στο `bench.h` και τα
και τα περνάμε αμέσως εκεί.
`DB* db;` Επειδή έχουμε ήδη τη μεταβλητή στο `bench.h`.
`db = db_open(DATAS);` Μεταφέρθηκαν όπως και είπαμε σαν λειτουργίες στο
`db_close(db);` `bench.c`.

Note: Δε πειράξαμε/αλλάξαμε τίποτα ενδιάμεσα στα δύο παραπάνω.

Έπειτα
`cost = end - start;`
Το μετατρέψαμε σε
`totalReadTime = end - start;` και περνάμε εδώ τον χρόνο πια.

Και προσθέσαμε
`readCount += count;` Μεταβλητή που μετράει τα συνολικά `count` για το `read`.

Μοναδική διαφορά με τις αλλαγές που κάναμε στη `write` (εκτός από το ό,τι χρησιμοποιούν μεταβλητές με `read` από μπροστά κι όχι με `write` (πχ `readCount` κι όχι `writeCount`) είναι το

`totalFound += found;` στο οποίο μετράμε και αναφέρουμε τα συνολικά `found` από
κάθε `read` που έχει τρέξει.

Καθώς και
`pthread_mutex_lock(&read_lock);` Περικλείωντας τα τρία που αναφέραμε ακριβώς
`pthread_mutex_unlock(&read_lock);` από επάνω, ώστε να μη τα πειράζουν πολλαπλά
νήματα ταυτόχρονα. Το `totalReadTime`, το
`readCount` και `totalFound` πρέπει να αλλάζουν
μόνο από ένα `read` νήμα τη κάθε στιγμή.

Τέλος :

Στο τέλος κάθε μεθόδου έχουμε διαγράψει τα `printf(...)`; και τα έχουμε μετακινήσει
στο `bench.c`, το οποίο και θα αναλύσουμε τώρα.

bench.c:

Οι αλλαγές αρχίζουν κάτω από το #include όπου και προσθέσαμε το

```
#define DATAS ("testdb")
```

 για τη μελλοντική χρήση στο άνοιγμα αρχείου παρακάτω.
Το πήραμε από το kiwi.c.

Υπάρχει ένα καινούριο struct

```
struct data
{
    long int count;
    int r;
};
```

Το οποίο χρησιμεύει μόνο για το πέρασμα τιμών στις παραμέτρους των read και write του kiwi.c. Αν παρατηρήσετε, το count και r μέσα στο struct αποτελούν και τις μοναδικές παραμέτρους των _read_test() και _write_test() του kiwi.c.

Θα εξηγηθεί περισσότερο αναλύοντας και το υπόλοιπο έγγραφο.

Δεν υπάρχουν αλλαγές στις ήδη υπάρχουσες μεθόδους

```
_random_key(...) , _print_header(...), _print_environment() .
```

Παρακάτω βλέπουμε την ύπαρξη δύο νέων μεθόδων, που είδαμε και στο μάθημα, passWriteParams(...), passReadParams(...)

τους οποίους θα αναλύσουμε και παρακάτω, αλλά ουσιαστικά είναι οι μέθοδοι που χρησιμοποιούμε στις pthread_create ώστε να καλέσουμε τις μεθόδους read και write από το kiwi.c και να περάσουμε τα κατάλληλα ορίσματα (που έδωσε ο χρήστης) στις παραμέτρους τους.

Μπαίνουμε στη main:

Πρώτη διαφορά είναι η προσθήκη και αρχικοποίηση όλων των μεταβλητών που είδαμε στο bench.h

```
totalFound = 0;
totalReadTime = 0;
totalWriteTime = 0;
readCount = 0;
writeCount = 0;
```

καθώς και των mutex

```
pthread_mutex_init(&read_lock, NULL);
pthread_mutex_init(&write_lock, NULL);
```

Μπαίνουμε στα if.

Μεταξύ των περιπτώσεων read και write υπάρχουν διαφορές κυρίως μόνο στις μεταβλητές που χρησιμοποιούνται (πχ readCount στο read κομμάτι και writeCount στο write). Οπότε ό,τι αλλάζει στο ένα έχει αλλάξει και στο άλλο, απλώς με τις κατάλληλες μεταβλητές.

Θα αναλυθούν και τα δύο αλλά το δεύτερο λιγότερο αναλυτικά, εφόσον αποτελεί σχεδόν αντιγραφή του πρώτου.

Μπαίνοντας στο : `if (strcmp(argv[1], "write") == 0)`

Η πρώτη αλλαγή που βλέπουμε είναι στη δημιουργία μιας μεταβλητής για τον αριθμό νημάτων threadCount, η οποία από default ορίζεται στο 1, εφόσον άμα δε δώσει ο χρήστης αριθμό νημάτων, τότε αυτό να τρέξει μόνο σε ένα.

`struct data dataWrite;` Αποτελεί το struct στο οποίο θα αποθηκεύσουμε τις τιμές που θέλουμε να δώσουμε στις κλήσεις των write από το kiwi.c.

Έπειτα

<code>if (argc >= 4)</code>	Βλέπουμε άμα ο χρήστης έχει δώσει αριθμό
<code>threadCount = atoi(argv[3]);</code>	επιθυμητών νημάτων, και τον αναθέτουμε.

<code>pthread_t threads[threadCount];</code>	Εφόσον έχουμε πάρει τον ζητούμενο αριθμό
	νημάτων από τον χρήστη (άμα έχει δωθεί)
	δημιουργούμε και τον ανάλογο πίνακα νημάτων.

<code>if (argc == 5)</code>	Εδώ, διότι αλλάξαμε τη σειρά και τον αριθμό των ορισμάτων που
<code>r = 1;</code>	δίνει ο χρήστης (προσθέσαμε το threadCount), αλλάξαμε τον
	αριθμό από 4 σε 5 για να διατηρηθεί σωστό.

Μετά από αυτό, διαγράψαμε το `_write_test(count, r);` που είχε το αρχικό αρχείο, γιατί θα την καλούμε με άλλο τρόπο παρακάτω.

```
dataWrite.count = count / threadCount;  
dataWrite.r = r;
```

Στις παραπάνω γραμμές κώδικα αναθέσαμε τις κατάλληλες τιμές στα πεδία του struct.

`r` : το έχεμου βρεί/ορίσει από πριν.

`count` : αποτελείται από τον αριθμό των στοιχείων που έχει δώσει ο χρήστης για να γράψει το πρόγραμμα, διά τον αριθμό των νημάτων που έχουν δωθεί (default 1) , ώστε να χωρίσει

τον φόρτο εργασίας κατάλληλα σε κάθε κλήση της `_write_test(..)` αργότερα.
Έστω έχουμε 10000 στοιχεία και 10 νήματα, κάθε διεργασία θα τρέξει από 1000 στοιχεία καθεμία.

`db = db_open(DATAS);` Το αντιγράψαμε από το `kiwi.c` και ανοίγει την βάση που είδαμε και στο `bench.h`.

Ύστερα

```
int i;
for (i = 0; i < threadCount; i++)
    pthread_create(&threads[i], NULL, passWriteParams, (void*)&dataWrite);
for (i = 0; i < threadCount; i++)
    pthread_join(threads[i], NULL);
```

Δημιουργούμε όσα νήματα μπορούμε, καλώντας τη μέθοδο `passWriteParams` και δίνοντάς της σαν ορίσματα τα στοιχεία `r`, `count` από το `dataWrite` struct που ορίσαμε και αναθέσαμε τιμές νωρίτερα.

Ουσιαστικά παίρνουμε τις τιμές που μας δίνει ο χρήστης, τις περνάμε στο struct και μέσω αυτού δίνουμε τιμές στις παραμέτρους του `_write_test()` του `kiwi`, το οποίο καλούμε να τρέξει το κάθε νήμα μέσω της `passWriteParams`.

Επίσης περιμένουμε να τελειώσει το κάθε νήμα τη δουλειά του.

`db_close(db);` Κλείνουμε εννοείται το `db` μας μετά. Αυτό το αντιγράψαμε από το `kiwi.c`.

Τέλος, αντιγράφουμε το `printf` από το `kiwi.c` πάλι και του περνάμε τις σωστές μεταβλητές ώστε να εκτυπώνει τα σωστά αποτελέσματα.

```
printf(LINE);
printf("|Random-Write      (done:%ld): %.6f sec/op; %.1f writes/sec(estimated);
cost:%.3f(sec);\n"
    , writeCount
    , (double)(totalWriteTime / writeCount / threadCount)
    , (double)(writeCount / totalWriteTime * threadCount)
    , totalWriteTime / threadCount);
```

Ακολουθεί το : `else if (strcmp(argv[1], "read") == 0)`

Πολύ εύκολα αναγνωρίζουμε ότι έχουμε κάνει ακριβώς τις ίδιες αλλαγές με το παραπάνω (αλλά με τις σωστές μεταβλητές φυσικά).

Διαγράψαμε το `_read_test(count, r);` στο τέλος του `if`.

Έχουμε προσθέσει στην αρχή τα

`int threadCount = 1;` για τα νήματα

καθώς και το

`struct data dataRead;` για το πέρασμα τιμών στις παραμέτρους του `_read_test` στο `kiwi.c`.

Πιο κάτω υπάρχει η συνθήκη για την ανάθεση των νημάτων, άμα έχει δώσει επιθυμητή τιμή ο χρήστης

```
if (argc >= 4)
    threadCount = atoi(argv[3]);
```

```
pthread_t threads[threadCount];
```

Εφόσον έχουμε πάρει τον ζητούμενο αριθμό νημάτων από τον χρήστη (άμα έχει δωθεί) δημιουργούμε και τον ανάλογο πίνακα νημάτων.

```
if (argc == 5)
    r = 1;
```

και όπως και πριν, αλλάξαμε το 4 σε 5, για τον ίδιο λόγο.

Αναθέτουμε τις τιμές

```
dataRead.count = count / threadCount;
dataRead.r = r;
```

και ανοίγουμε τη βάση

```
db = db_open(DATAS);
```

Δημιουργούμε τα νήματα τα οποία τρέχουν ουσιαστικά το `_read_test()` από το `kiwi.c` με τις σωστές τιμές για παραμέτρους, και περιμένουμε να τελειώσουν.

```
int i;
for (i = 0; i < threadCount; i++)
    pthread_create(&threads[i], NULL, passReadParams, (void*)&dataRead);
for (i = 0; i < threadCount; i++)
    pthread_join(threads[i], NULL);
```


Κλείνουμε το db

```
db_close(db);
```

Και τέλος, αντιγράφουμε τα printf που πήραμε από τη _read_test() της kiwi.c και περνάμε τις σωστές μεταβλητές για να εκτυπώνει σωστά αποτελέσματα.

```
printf(LINE);
printf("|Random-Read (done:%ld, found:%d): %.6f sec/op; %.1f reads /sec(estimated);
cost:%.3f(sec)\n",
    readCount, totalFound,
    (double)(totalReadTime / readCount / threadCount),
    (double)(readCount / totalReadTime * threadCount),
    totalReadTime / threadCount);
```

Τέλος, έχουμε υλοποιήσει μία καινούρια εξ' ολοκλήρου else if μαζί με τα περιεχόμενά της, για τη περίπτωση ταυτόχρονων read & write.

```
else if (strcmp(argv[1], "readwrite") == 0)
```

Αρχικά ορίζουμε και αρχικοποιούμε σχεδόν όλες τις μεταβλητές που θα μας χρειαστούν (εκτός από τους pthreads πίνακες).

```
int r = 0;
// default threads = 2, one for each method
long int totalThreads = 2;
// To calculate the threads for each method
long int readThreads;
long int writeThreads;
// To calculate the count for each method
long int localWriteCount;
long int localReadCount;
```

Το totalThreads αποτελεί τα συνολικά νήματα που έδωσε ο χρήστης, το οποίο by default είναι ορισμένο στα 2, ένα για το read κι ένα για το write.

Τα υπόλοιπα αποτελούν μεταβλητές για να διαχωρίσουμε ύστερα τα συνολικά threads & count.

Noted, πως έχουμε ορίσει όλα ως long int για ευκολία στις πράξεις.

<pre>struct data dataRead;</pre>	Ίδιος σκοπός με τα data struct των άλλων if, απλώς
<pre>struct data dataWrite;</pre>	ένα για το read & ένα για το write.

Έχουμε αντιγράψει από τα ήδη έτοιμα if τα παρακάτω:

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

```
if (argc >= 4)
    totalThreads = (long int)atoi(argv[3]);
```

των οποίων λειτουργίες γνωρίζουμε.

Έπειτα

```
//default values 50/50 percentage read-write
localReadCount = count / 2;
localWriteCount = count - localReadCount;
// default threads 50/50 as well
readThreads = totalThreads / 2;
writeThreads = totalThreads - readThreads;
```

Χωρίζουμε τα συνολικά count και threads στη μέση, μισά για το read, μισά για το write.

Πχ άμα έχουν 100 count, τα διαιρούμε δια 2, άρα localReadCount = 50 και το localWriteCount έχει τα υπόλοιπα, δηλ count – localReadCount = 100 – 50 = 50. Αναλόγως και για τα νήματα.

```
//Check to see if user has input % to divide the count & threads, ex: 10/90
//cannot be single digit on the first, if threads < 100
if (argc >= 5)
{
    int num;
    char str[2];
    strncpy(str, argv[4], 2);
    if (str[2] == '/')
    {
        strncpy(str, argv[4], 1);
        num = atoi(str);
    }
    else
        num = atoi(str);

    // divide count accordingly
    localReadCount = num * count / 100;
    localWriteCount = count - localReadCount;

    // divide threads accordingly
    readThreads = num * totalThreads / 100;
    writeThreads = totalThreads - readThreads;
}
```

Άμα έχει δώσει % read/write που επιθυμεί ο χρήστης, τότε διαβάζει τη τιμή και τα διαχωρίζει ανάλογα.

Όπως αναφέρεται και στα σχόλια, άμα τα threads είναι μικρότερα του 100 και ο χρήστης δώσει single digit καπου (πχ 5/95), τότε παρουσιάζεται παρακάτω στον κώδικα floating point exception επειδή μία μεταβλητή πχ readThreads == 0 αργότερα διαιρεί το localReadCount.

Δημιουργούμε τους ανάλογους πίνακες νημάτων

```
pthread_t read_threads[writeThreads], write_threads[readThreads];
```

```
if (argc == 6)                                και κοιτάμε για το random
    r = 1;
```

```
db = db_open(DATAS);                          και ύστερα ανοίγουμε τη βάση
```

Μετά περνάμε τις σωστές τιμές για read ή write στο ανάλογο data struct.

```
// For the read method
dataRead.count = localReadCount / readThreads;
dataRead.r = r;
// For the write method
dataWrite.count = localWriteCount / writeThreads;
dataWrite.r = r;
```

Παρατηρείστε αυτό που υπόθηκε προηγουμένως, δηλαδή πως άμα readThreads == 0, τότε θα εμφανιστεί σφάλμα τύπου floating point exception.

Δημιουργούμε τα νήματα και τα κάνουμε join αφού έχουμε ξεκινήσει και των δύο ειδών.

```
for (i = 0; i < writeThreads; i++)
    pthread_create(&write_threads[i], NULL, passWriteParams, (void*)&dataWrite);
for (i = 0; i < readThreads; i++)
    pthread_create(&read_threads[i], NULL, passReadParams, (void*)&dataRead);

for (i = 0; i < writeThreads; i++)
    pthread_join(write_threads[i], NULL);
for (i = 0; i < readThreads; i++)
    pthread_join(read_threads[i], NULL);
```

Κλείνουμε το db

```
db_close(db);
```

και εκτυπώνουμε τα κατάλληλα δεδομένα

```
printf(LINE);
printf("|Random-Read (done:%ld, found:%d): %.6f sec/op; %.1f reads /sec(estimated);
cost:%.3f(sec)\n",
    readCount, totalFound,
    (double)(totalReadTime / readCount / readThreads),
    (double)(readCount / totalReadTime * readThreads),
    totalReadTime / readThreads);
printf(LINE);
printf("|Random-Write      (done:%ld): %.6f sec/op; %.1f writes/sec(estimated);
cost:%.3f(sec);\n"
    , writeCount
    , (double)(totalWriteTime / writeCount / writeThreads)
    , (double)(writeCount / totalWriteTime * writeThreads)
    , totalWriteTime / writeThreads);
```

Τελειώσαμε με τις αλλαγές στα αρχεία bench.h, bench.c και kiwi.c
Και μεταφερόμαστε στις αλλαγές που υλοποιήσαμε στα αρχεία db.h και db.c.
Αφορούν απλές αλλαγές για την υλοποίηση συγχρονισμού μεταξύ read και write.

db.h:

Μοναδικές αλλαγές εδώ έγιναν στο `struct _db`.

Προσθέσαμε

```
pthread_mutex_t readwrite_lock;  
pthread_cond_t no_more_readers;  
int readCount;
```

Θα εξηγηθούν πως χρησιμοποιούνται παρακάτω.

db.c:

Οι μέθοδοι που μας απασχολούν μόνο είναι οι `db_add()` (write) και `db_get()` (read), διότι αυτές καλούνε οι μέθοδοι `_read_test(..)`, `_write_test(..)` από το `kiwi.c`.

Πρώτα όμως, μέσα στη μέθοδο `db_open_ex` η οποία καλείται όταν κάνουμε `open` τη βάση,

Αρχικοποιήσαμε τις μεταβλητές

```
pthread_mutex_init(&self->readwrite_lock, NULL);  
pthread_cond_init(&self->no_more_readers, NULL);
```

και θέτουμε

```
self->readCount = 0;
```

Η λογική πίσω από την υλοποίησή μας:

`readCount` αποτελεί έναν μετρητή στον οποίο προσθέτουμε +1, κάθε φορά που μία `read` ξεκινάει και -1, κάθε φορά που τελειώνει.

Άμα `readCount == 0`, δηλαδή δεν υπάρχουν άλλοι `readers` ενεργοί αυτή τη στιγμή, τότε κάνει `signal` στη `write`, η οποία περιμένει το σήμα `no_more_readers` και προχωράει με την εγγραφή στοιχείων στη βάση.

Έχουμε υλοποιήσει παραλληλισμό νήματων read, διότι δε μας πειράζει να διαβάζουν από τη βάση πολλοί ταυτόχρονα, ενώ μόνο έναν writer ανά στιγμή, κι αυτός μόνο αφότου δεν υπάρχουν άλλοι readers. Πάμε να το δούμε και στον κώδικα.

Στην επόμενη σελίδα βρίσκεται μία φωτογραφία.

Για να καταφέρουμε παραλληλισμό μεταξύ read threads, δεν έχουμε κλειδώσει το critical section κι έχουμε προσθέσει lock/unlock μόνο πριν και μετά αυτού, για να επεξεργαστούμε το readCount. Στο δεύτερο lock/unlock, αφού έχει τρέξει το critical section, τότε κοιτάμε αν υπάρχουν κι άλλοι readers και άμα όχι κάνουμε signal το no_more_readers;

Τώρα, στη write με το που τη καλούμε κάνουμε lock και ενώ ο readCount είναι μεγαλύτερος του 0, περιμένουμε. Όταν το read κάνει signal το no_more_readers, τότε θα ξανατρέξει ο κώδικας εκεί, θα περάσει το while condition και να τρέξει το critical section του write. Μετά από αυτό, θα κάνει unlock.

Μοναδικές αλλαγές πέρα από αυτές τις προσθήκες κώδικα που αναφέρθηκαν πριν και μετά τα critical sections, αποτελούν οι μεταβλητές result σε κάθε μέθοδο, στους οποίους περνάμε εκεί το αποτέλεσμα πια.

Επίσης κάνουμε ένα και μόνο return κι αυτό είναι στο τέλος της μεθόδου. Έχουμε αφαιρέσει οποιοδήποτε άλλο return, ώστε να μη κάνει return και τελειώσει σαν μέθοδος πριν τρέξει τον κώδικα για το unlock του mutex.

```

int db_add(DB* self, Variant* key, Variant* value)
{
    int result;

    pthread_mutex_lock(&self->readwrite_lock);
    while (self->readCount > 0)
        pthread_cond_wait(&self->no_more_readers, &self->readwrite_lock);

    // Critical section
    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);
    }
    result = memtable_add(self->memtable, key, value);
    // end of critical section

    pthread_mutex_unlock(&self->readwrite_lock);
    return result;
}

int db_get(DB* self, Variant* key, Variant* value)
{
    int result;

    pthread_mutex_lock(&self->readwrite_lock);

    self->readCount++;
    pthread_mutex_unlock(&self->readwrite_lock);

    // Critical section
    if (memtable_get(self->memtable->list, key, value) == 1)
        result = 1;
    else
        result = sst_get(self->sst, key, value);
    // end of critical section

    pthread_mutex_lock(&self->readwrite_lock);

    self->readCount--;
    if (self->readCount == 0)
        pthread_cond_signal(&self->no_more_readers);
    pthread_mutex_unlock(&self->readwrite_lock);
    return result;
}

```

Αποτελέσματα

Έξοδος της τελικής εντολής *make*:

```
myy601@myy601lab1:~/kiwi/kiwi-source$ make all
cd engine && make all
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/engine'
  CC db.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 -lsnappy -o kiwi-bench
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/bench'
myy601@myy601lab1:~/kiwi/kiwi-source$
```

./kiwi-bench write

./kiwi-bench write 1000000

```
+-----+-----+-----+
+-----+-----+-----+
|Random-Write  (done:1000000): 0.000022 sec/op; 45454.5 writes/
sec(estimated); cost:22.000(sec);
myy601@myy601lab1:~/kiwi/kiwi-source/bench$
```

./kiwi-bench write 1000000 100

```
+-----+-----+-----+
+-----+-----+-----+
|Random-Write  (done:1000000): 0.000010 sec/op; 96246.4 writes/
sec(estimated); cost:10.390(sec);
myy601@myy601lab1:~/kiwi/kiwi-source/bench$
```

./kiwi-bench read

./kiwi-bench read 1000000

```
-----+-----+
|Random-Read      (done:1000000, found:1000000): 0.000014 sec/op;
71428.6 reads /sec(estimated); cost:14.000(sec)
myy601@myy601lab1:~/kiwi/kiwi-source/bench$
```

./kiwi-bench read 1000000 100

```
-----+-----+
|Random-Read      (done:1000000, found:1000000): 0.000003 sec/op;
342465.8 reads /sec(estimated); cost:2.920(sec)
myy601@myy601lab1:~/kiwi/kiwi-source/bench$
```

./kiwi-bench readwrite

./kiwi-bench readwrite 1000000

```
-----+-----+
|Random-Read      (done:500000, found:500000): 0.000034 sec/op; 29
411.8 reads /sec(estimated); cost:17.000(sec)
+-----+-----+-----+-----+
|Random-Write     (done:500000): 0.000034 sec/op; 29411.8 writes/s
ec(estimated); cost:17.000(sec);
myy601@myy601lab1:~/kiwi/kiwi-source/bench$
```

./kiwi-bench readwrite 1000000 100

```
-----+-----+
|Random-Read      (done:500000, found:500000): 0.000014 sec/op; 69
444.4 reads /sec(estimated); cost:7.200(sec)
+-----+-----+-----+-----+
|Random-Write     (done:500000): 0.000017 sec/op; 57339.4 writes/s
ec(estimated); cost:8.720(sec);
myy601@myy601lab1:~/kiwi/kiwi-source/bench$
```


./kiwi-bench readwrite 1000000 10

./kiwi-bench readwrite 1000000 10 90/10

```
-----+-----+
|Random-Read      (done:900000, found:900000): 0.000007 sec/op; 15
0000.0 reads /sec(estimated); cost:6.000(sec)
+-----+-----+
-----+-----+
|Random-Write      (done:100000): 0.000060 sec/op; 16666.7 writes/s
ec(estimated); cost:6.000(sec);
mvv601@mvv601lab1:~/kiwi/kiwi-source/bench$
```

./kiwi-bench readwrite 1000000 10 40/60

```
-----+-----+
|Random-Read      (done:400000, found:400000): 0.000032 sec/op; 30
769.2 reads /sec(estimated); cost:13.000(sec)
+-----+-----+
-----+-----+
|Random-Write      (done:600000): 0.000025 sec/op; 40449.4 writes/s
ec(estimated); cost:14.833(sec);
mvv601@mvv601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite
```

./kiwi-bench readwrite 1000000 10 75/25

```
-----+-----+
|Random-Read      (done:749994, found:749994): 0.000015 sec/op; 68
181.3 reads /sec(estimated); cost:11.000(sec)
+-----+-----+
-----+-----+
|Random-Write      (done:249999): 0.000041 sec/op; 24193.5 writes/s
ec(estimated); cost:10.333(sec);
```

./kiwi-bench readwrite 1000000 100

./kiwi-bench readwrite 1000000 100 51/49

```
-----+-----+
|Random-Read    (done:510000, found:510000): 0.000019 sec/op; 52
228.9 reads /sec(estimated); cost:9.765(sec)
+-----+-----+
-----+-----+
|Random-Write    (done:490000): 0.000025 sec/op; 39360.7 writes/s
ec(estimated); cost:12.449(sec);
```

./kiwi-bench readwrite 1000000 100 48/52

```
+-----+-----+
-----+-----+
|Random-Read    (done:480000, found:480000): 0.000015 sec/op; 66
976.7 reads /sec(estimated); cost:7.167(sec)
+-----+-----+
-----+-----+
|Random-Write    (done:520000): 0.000019 sec/op; 52812.5 writes/s
ec(estimated); cost:9.846(sec);
```

./kiwi-bench readwrite 1000000 100 58/42

```
+-----+-----+
-----+-----+
|Random-Read    (done:580000, found:580000): 0.000011 sec/op; 88
759.9 reads /sec(estimated); cost:6.534(sec)
+-----+-----+
-----+-----+
|Random-Write    (done:420000): 0.000020 sec/op; 49830.5 writes/s
ec(estimated); cost:8.429(sec);
myv601@myv601lab1:~/kiwi/kiwi-source/bench$
```

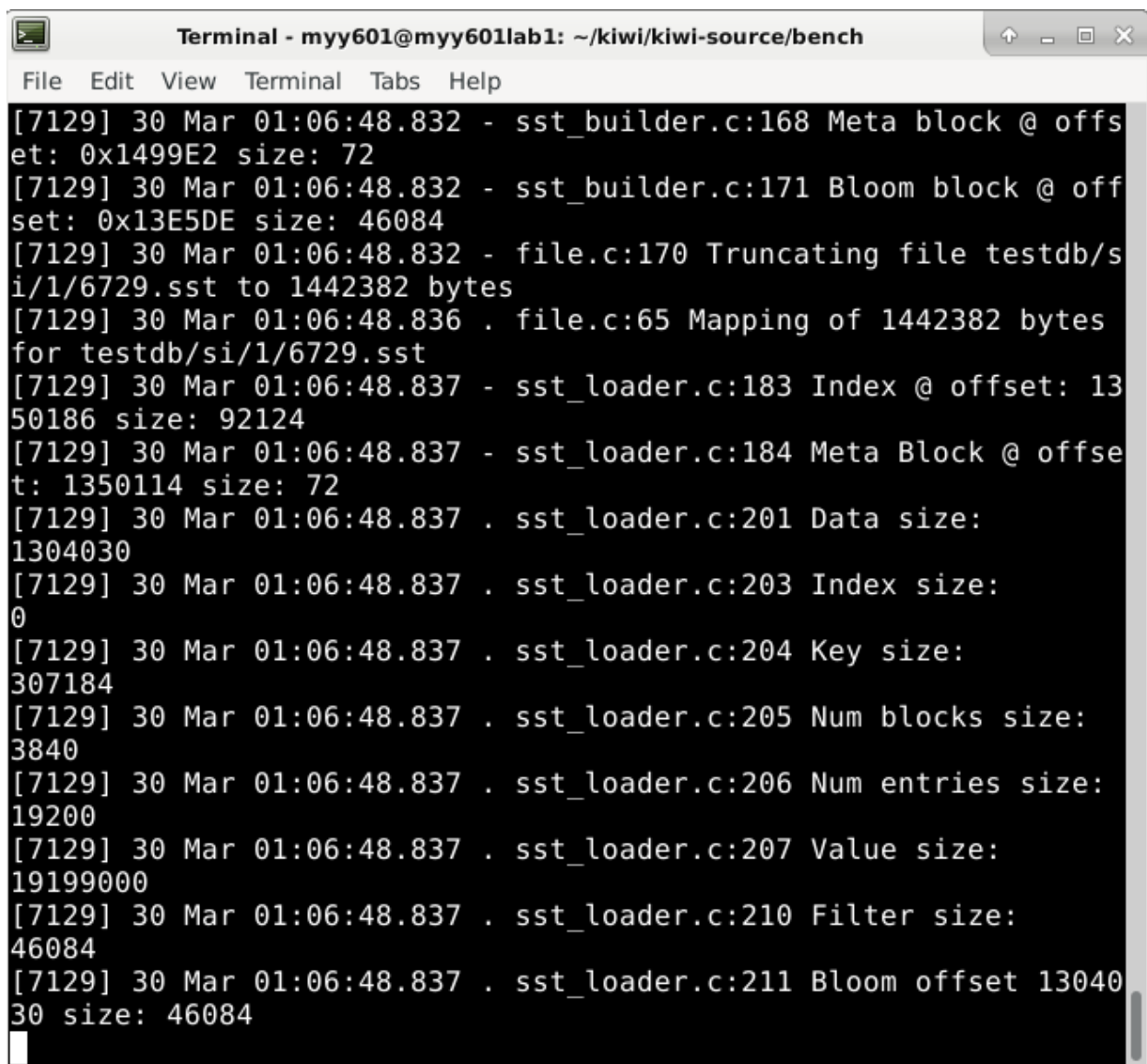
Προβλήματα

Προβλήματα παρουσιάστηκαν μόνο στο readwrite και συγκεκριμένα, όταν ο χρήστης έδινε ποσοστό %.

Για την εντολή ./kiwi-bench readwrite 1000000 100

Δηλαδή για 100 νήματα, άμα δέχοντας ποσοστά μικρότερα του 48/52 ή μεγαλύτερα του 58/42 τότε εμφανιζε είτε segm. fault, είτε σταματούσε να τρέχει / κολλούσε στη παρακάτω εικόνα

Πχ. για ./kiwi-bench readwrite 1000000 100 90/10



```
Terminal - myy601@myy601lab1: ~/kiwi/kiwi-source/bench
File Edit View Terminal Tabs Help
[7129] 30 Mar 01:06:48.832 - sst_builder.c:168 Meta block @ offset: 0x1499E2 size: 72
[7129] 30 Mar 01:06:48.832 - sst_builder.c:171 Bloom block @ offset: 0x13E5DE size: 46084
[7129] 30 Mar 01:06:48.832 - file.c:170 Truncating file testdb/si/1/6729.sst to 1442382 bytes
[7129] 30 Mar 01:06:48.836 . file.c:65 Mapping of 1442382 bytes for testdb/si/1/6729.sst
[7129] 30 Mar 01:06:48.837 - sst_loader.c:183 Index @ offset: 1350186 size: 92124
[7129] 30 Mar 01:06:48.837 - sst_loader.c:184 Meta Block @ offset: 1350114 size: 72
[7129] 30 Mar 01:06:48.837 . sst_loader.c:201 Data size: 1304030
[7129] 30 Mar 01:06:48.837 . sst_loader.c:203 Index size: 0
[7129] 30 Mar 01:06:48.837 . sst_loader.c:204 Key size: 307184
[7129] 30 Mar 01:06:48.837 . sst_loader.c:205 Num blocks size: 3840
[7129] 30 Mar 01:06:48.837 . sst_loader.c:206 Num entries size: 19200
[7129] 30 Mar 01:06:48.837 . sst_loader.c:207 Value size: 19199000
[7129] 30 Mar 01:06:48.837 . sst_loader.c:210 Filter size: 46084
[7129] 30 Mar 01:06:48.837 . sst_loader.c:211 Bloom offset 1304030 size: 46084
```

Επίσης προβλήματα εμφανιζόντουσαν και όταν δίνονταν λιγότερα νήματα (10), αλλά σε μικρότερη κλίμακα σε σχέση με τα 100 νήματα. Δούλεψε για μεγαλύτερο εύρος τιμών ποσοστού.

Για `./kiwi-bench readwrite 1000000 10 10/90`, εμφάνιζε `segm. fault`.

Γενικά, εκτός από όταν δινόταν συγκεκριμένο ποσοστό %, δεν υπήρξε κανένα πρόβλημα.