Λειτουργικά Συστήματα
2025

Εργαστήριο 1:Υλοποίηση πολυνηματικής λειτουργίας σε μηχανή αποθήκευσης δεδομένων

Τσουμάνη Βασιλική Ελένη Α.Μ 5168
Παγώνης Δημήτριος Α.Μ 4985
Τζαλοκώστας Δημήτριος Α.Μ 4994

ΠΕΡΙΕΧΟΜΕΝΑ

ΕΙΣΑΓΩΓΗ

Στην παρούσα άσκηση κληθήκαμε να υλοποιήσουμε την πολυνηματική λειτουργία των εντολών add και get που παρέχει η μηχανή αποθήκευσης . Ο πηγαίος κώδικας που μας δίνεται υλοποιεί την μηχανή αποθήκευσης Kiwi που βασίζεται σε δέντρο log-structured merge (LSM-tree).

Το LSM-tree είναι η δομή δεδομένων στην οποία συχνά βασίζονται οι μηχανές αποθήκευσης. Η παρεχόμενη προγραμματιστική διεπαφή (API) περιλαμβάνει λειτουργίες add και get για ζεύγη κλειδιού-τιμής.

Τα αρχεία που τροποποιήθηκαν για τη δημιουργία πολλαπλών νημάτων είναι τα **kiwi.c, bench.c, db.c, db.h, utils.c** .

SPECS ΚΑΙ VS CODE INSTALLATION

Τα Tests που πραγματοποιήσαμε, εκτελέστηκαν τα εξής specs:

Intel-i713700H με 20 CPUs, 2,4GHz, 32GB RAM με λειτουργικό σύστημα Windows 11 Pro και πιο συγκεκριμένα στο vm του debian linux δώσαμε 5.7 GB RAM, 8 processors και 10GB hard disk.

Επιπλέον, εγκαταστήσαμε την εφαρμογή Visual Studio Code στο linux, με στόχο τη πιο εύκολη και αποτελεσματική εκτέλεση της εργασίας.Αρχικά συνδεθήκαμε ως διαχειριστής στο σύστημα (root) με κωδικό myy601, ανοίξαμε ένα τερματικό και οι εντολές που εκτελέσαμε ειναι:

1. apt install sudo
2. sudo apt install wget -y
3. wget -qO- https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor | sudo tee /usr/share/keyrings/packages.microsoft.gpg > /dev/null
4. echo "deb [arch=amd64 signed-by=/usr/share/keyrings/packages.microsoft.gpg] https://packages.microsoft.com/repos/code stable main" | sudo tee /etc/apt/sources.list.d/vscode.list
5. sudo apt update
6. sudo apt install code -y

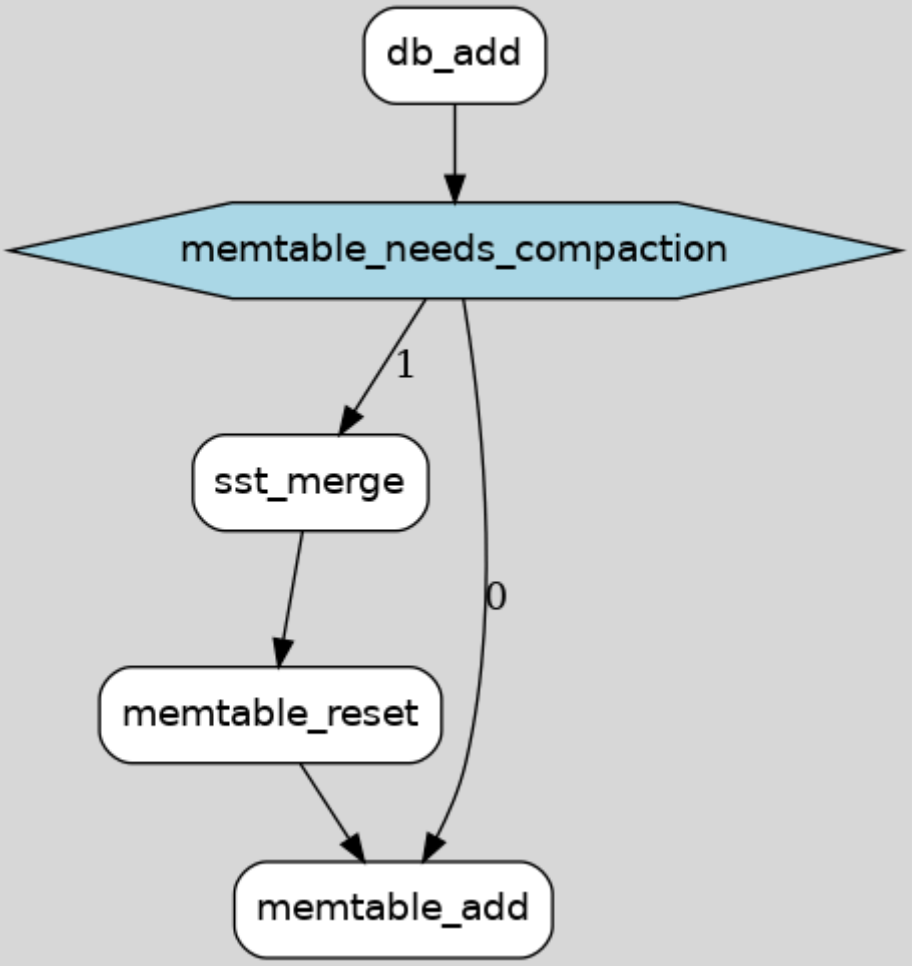
Και τέλος, για να ανοίξει η εφαρμογή, αποσυνδεθήκαμε από το root και συνδεθήκαμε πάλι ως myy601, ανόιξαμε τερματικό και εκτελέσαμε την εντολή:

1. code

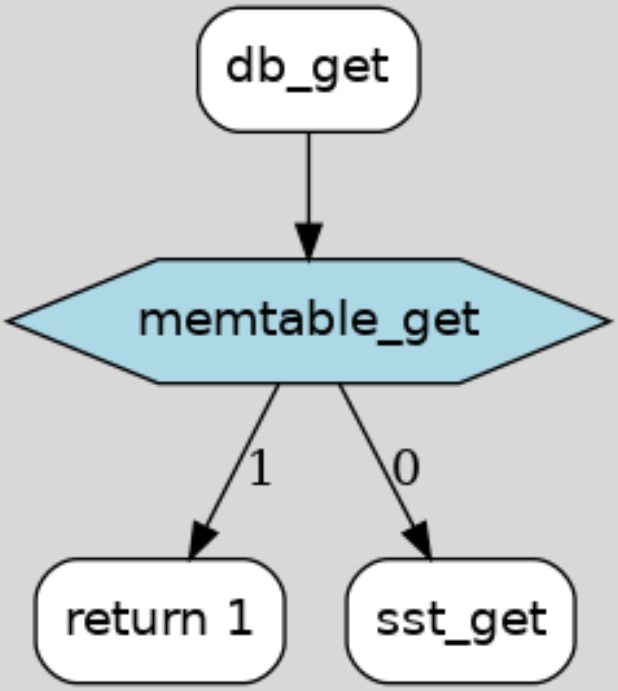
και μας άνοιξε η εφαρμογή.

Κατανόηση των Λειτουργιών add και get στη Μηχανή Αποθήκευσης Δεδομένων

Η μέθοδος db_add υλοποιείται στον κώδικα του αρχείου db.c και είναι υπεύθυνη για την προσθήκη ενός νέου στοιχείου στη δομή αποθήκευσης δεδομένων. Η μέθοδος αυτή διαχειρίζεται δυναμικά τη μεταφορά δεδομένων από τη μνήμη στον δίσκο ώστε να διατηρηθεί η απόδοση του συστήματος. Αρχικά, πραγματοποιείται έλεγχος μέσω της memtable_needs_compaction για να διαπιστωθεί εάν το memtable έχει γεμίσει και χρειάζεται μεταφορά στο δίσκο. Αν η απάντηση είναι αρνητική , τότε η συνάρτηση memtable_add προσθέτει απευθείας το νέο ζεύγος key-value στο memtable.
Αν η απάντηση είναι θετική, τότε καλείται η sst_merge, η οποία μεταφέρει τα δεδομένα του memtable στο πρώτο επίπεδο αποθήκευσης SST. Αυτή η διαδικασία πραγματοποιείται μέσω ενός νήματος συμπίεσης (compaction thread), το οποίο αναλαμβάνει να μετατρέψει τα δεδομένα της κύριας μνήμης σε μια πιο αποδοτική μορφή αποθήκευσης στον δίσκο. Μετά την ολοκλήρωση τη συγχώνευση (sst_merge), καλείται η memtable_reset, η οποία καθαρίζει το memtable για να δεχτεί νέα δεδομένα. Αφού ολοκληρωθεί το reset, η μέθοδος συνεχίζει με την εισαγωγή του νέου ζεύγους key-value στο memtable.



Η μέθοδος db_get υλοποιείται στον κώδικα του αρχείου db.c και είναι υπεύθυνη για την ανάκτηση δεδομένων από τη μηχανή αποθήκευσης. Αρχικά αναζητά πρώτα τα δεδομένα στη μνήμη (memtable) και, αν δεν υπάρχουν εκεί, αναζητά τα δεδομένα στους αποθηκευτικούς δίσκους (SST αρχεία). Ύστερα, η συνάρτηση καλεί τη memtable_get, η οποία ψάχνει αν το κλειδί (key) υπάρχει στο memtable. Αν βρεθεί, επιστρέφει την τιμή (value) και η διαδικασία ολοκληρώνεται επιτυχώς. Εάν το κλειδί δεν βρεθεί στο memtable, η μέθοδος συνεχίζει την αναζήτηση στα αποθηκευμένα αρχεία SST μέσω της sst_get. Η προσέγγιση αυτή εξασφαλίζει ότι τα δεδομένα που έχουν προσπελαστεί πρόσφατα είναι άμεσα διαθέσιμα στη μνήμη, βελτιώνοντας την ταχύτητα ανάκτησης.



8Τελικά, οι βασικές λειτουργίες put και get, που υλοποιούνται μέσω των συναρτήσεων db_add και db_get, αντίστοιχα, διαχειρίζονται την προσθήκη και ανάκτηση δεδομένων με τρόπο που εξασφαλίζει απόδοση και αξιοπιστία.

ΕΚΤΈΛΕΣΗ ΕΓΓΡΑΦΏΝ ΚΑΙ ΑΝΑΓΝΏΣΕΩΝ ΜΕΣΩ ΤΟΥ BENCHMARK

Οι εντολές που χρησιμοποιήσαμε είναι οι εξής:

- **gdb kiwi-bench**
  Εκκινεί τον GDB και φορτώνει το εκτελέσιμο αρχείο kiwi-bench για αποσφαλμάτωση.

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ gdb kiwi-bench
GNU gdb (Debian 13.1-3) 13.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kiwi-bench...
(gdb)
```

- **file kiwi-bench**
  Φορτώνει το symbol table από το εκτελέσιμο kiwi-bench. Ο πίνακας συμβόλων περιέχει πληροφορίες για συναρτήσεις, μεταβλητές και διευθύνσεις μνήμης

```
(gdb) file kiwi-bench
Load new symbol table from "kiwi-bench"? (y or n) y
Reading symbols from kiwi-bench...
(gdb)
```

- **run write 100000**
  Εκτελεί μια λειτουργία εγγραφής δεδομένων 100.000 φορές, με διάρκεια 1 sec.

```
[2243] 31 Mar 18:04:41.292 - sst.c:878 Range [key-98808, key-99999] DOES NOT overlap in level 0. Checking others
[2243] 31 Mar 18:04:41.292 - sst.c:893 Range [key-98808, key-99999] DOES  overlap in level 1. Checking others
[2243] 31 Mar 18:04:41.292 . sst.c:930 Using level 0 for memtable compaction [key-98808, key-99999]
[2243] 31 Mar 18:04:41.292 . file.c:200 Creating directory structure: testdb/si/0
[2243] 31 Mar 18:04:41.292 . sst.c:634 Compaction of 1192 [1214648 bytes allocated] elements started
[2243] 31 Mar 18:04:41.292 - sst_builder.c:167 Index block @ offset: 0x137F2 size: 5692
[2243] 31 Mar 18:04:41.292 - sst_builder.c:168 Meta block @ offset: 0x137AA size: 72
[2243] 31 Mar 18:04:41.292 - sst_builder.c:171 Bloom block @ offset: 0x12C72 size: 2872
[2243] 31 Mar 18:04:41.292 - file.c:170 Truncating file testdb/si/0/33.sst to 85622 bytes
[2243] 31 Mar 18:04:41.293 . file.c:65 Mapping of 85622 bytes for testdb/si/0/33.sst
[2243] 31 Mar 18:04:41.293 - sst_loader.c:183 Index @ offset: 79858 size: 5692
[2243] 31 Mar 18:04:41.293 - sst_loader.c:184 Meta Block @ offset: 79786 size: 72
[2243] 31 Mar 18:04:41.293 . sst_loader.c:201 Data size:         76914
[2243] 31 Mar 18:04:41.293 . sst_loader.c:203 Index size:        0
[2243] 31 Mar 18:04:41.293 . sst_loader.c:204 Key size:          19072
[2243] 31 Mar 18:04:41.293 . sst_loader.c:205 Num blocks size:  239
[2243] 31 Mar 18:04:41.293 . sst_loader.c:206 Num entries size: 1192
[2243] 31 Mar 18:04:41.293 . sst_loader.c:207 Value size:       1192000
[2243] 31 Mar 18:04:41.293 . sst_loader.c:210 Filter size:      2872
[2243] 31 Mar 18:04:41.293 . sst_loader.c:211 Bloom offset 76914 size: 2872
[2243] 31 Mar 18:04:41.293 . sst.c:636 Compaction of 1192 elements finished
[2243] 31 Mar 18:04:41.293 - file.c:170 Truncating file testdb/si/manifest to 224 bytes
[2243] 31 Mar 18:04:41.293 . sst.c:51 --- Level 0 [  4 files, 948 KiB  ]---
[2243] 31 Mar 18:04:41.293 . sst.c:55 Metadata filenum:30 smallest: key-86457 largest: key-90573
[2243] 31 Mar 18:04:41.293 . sst.c:55 Metadata filenum:31 smallest: key-90574 largest: key-94690
[2243] 31 Mar 18:04:41.293 . sst.c:55 Metadata filenum:32 smallest: key-94691 largest: key-98807
[2243] 31 Mar 18:04:41.294 . sst.c:55 Metadata filenum:33 smallest: key-98808 largest: key-99999
[2243] 31 Mar 18:04:41.294 . sst.c:51 --- Level 1 [  1 files,   5 MiB  ]---
[2243] 31 Mar 18:04:41.294 . sst.c:55 Metadata filenum:29 smallest: key-0 largest: key-9999
[2243] 31 Mar 18:04:41.294 . sst.c:51 --- Level 2 [  1 files, 286 KiB  ]---
[2243] 31 Mar 18:04:41.294 . sst.c:55 Metadata filenum:0 smallest: key-0 largest: key-999
[2243] 31 Mar 18:04:41.294 . sst.c:51 --- Level 3 [  0 files,   0 bytes]---
[2243] 31 Mar 18:04:41.294 . sst.c:51 --- Level 4 [  0 files,   0 bytes]---
[2243] 31 Mar 18:04:41.294 . sst.c:51 --- Level 5 [  0 files,   0 bytes]---
[2243] 31 Mar 18:04:41.294 . sst.c:51 --- Level 6 [  0 files,   0 bytes]---
[2243] 31 Mar 18:04:41.294 . log.c:46 Removing old log file testdb/si/23.log
[2243] 31 Mar 18:04:41.294 . sst.c:170 Merge successfully completed. Releasing the skiplist
[2243] 31 Mar 18:04:41.294 . skiplist.c:57 SkipList refcount is at 0. Freeing up the structure
[2243] 31 Mar 18:04:41.294 - sst.c:176 Exiting from the merge thread as user requested
[2243] 31 Mar 18:04:41.294 - file.c:170 Truncating file testdb/si/manifest to 224 bytes
[2243] 31 Mar 18:04:41.295 . log.c:46 Removing old log file testdb/si/24.log
+--------------------------+---------------+---------------------------+------------------+
|Random-Write   (done:100000): 0.000010 sec/op; 100000.0 writes/sec(estimated); cost:1.000(sec);
```

- **run read 100000**
  Εκτελεί ανάγνωση δεδομένων 100.000 φορές, με διάρκεια 1 sec.

```
99986 searching key-99986
99987 searching key-99987
99988 searching key-99988
99989 searching key-99989
99990 searching key-99990
99991 searching key-99991
99992 searching key-99992
99993 searching key-99993
99994 searching key-99994
99995 searching key-99995
99996 searching key-99996
99997 searching key-99997
99998 searching key-99998
99999 searching key-99999
[2247] 31 Mar 18:05:29.089 . db.c:41 Closing database 0
[2247] 31 Mar 18:05:29.089 . sst.c:415 Sending termination message to the detached thread
[2247] 31 Mar 18:05:29.089 . sst.c:422 Waiting the merger thread
[2247] 31 Mar 18:05:29.089 - sst.c:176 Exiting from the merge thread as user requested
[2247] 31 Mar 18:05:29.089 - file.c:170 Truncating file testdb/si/manifest to 81 bytes
[2247] 31 Mar 18:05:29.094 . log.c:46 Removing old log file testdb/si/0.log
[2247] 31 Mar 18:05:29.094 . skiplist.c:57 SkipList refcount is at 0. Freeing up the structure
+----------------------------+---------------+----------------------------+------------------+
|Random-Read    (done:100000, found:100000): 0.000010 sec/op; 100000.0 reads /sec(estimated); cost:1.000(sec)
```

- **lay next**
  Ανοίγει τον πηγαίο κώδικα μέσα στο περιβάλλον του GDB, ώστε να είναι ευκολότερη η παρακολούθηση της εκτέλεσης του προγράμματος.

```
   0x555555556bde <_print_environment+1274>        lea    -0x220(%rbp),%rax
   0x555555556be5 <_print_environment+1281>        mov    %rax,%rsi
   0x555555556be8 <_print_environment+1284>        lea    0xc53b(%rip),%rax
   0x555555556bef <_print_environment+1291>        mov    %rax,%rdi
   0x555555556bf2 <_print_environment+1294>        mov    $0x0,%eax
   0x555555556bf7 <_print_environment+1299>        call   0x555555556050 <pri
   0x555555556bfc <_print_environment+1304>        nop
   0x555555556bfd <_print_environment+1305>        leave
   0x555555556bfe <_print_environment+1306>        ret
   0x555555556bff <main>                           push   %rbp
   0x555555556c00 <main+1>                         mov    %rsp,%rbp
   0x555555556c03 <main+4>                         sub    $0x20,%rsp
   0x555555556c07 <main+8>                         mov    %edi,-0x14(%rbp)
exec No process In:                                                L??   PC: ??
warning: Source file is more recent than executable.
(gdb)
```

- **break main**
  Δημιουργεί ένα breakpoint στη συνάρτηση main, ώστε η εκτέλεση του προγράμματος να σταματήσει εκεί, επιτρέποντας την ανάλυση της αρχικής κατάστασης των μεταβλητών και της μνήμης.

```
(gdb) break main
Breakpoint 1 at 0x555555556c0e: file bench.c, line 75.
(gdb)
```

- **set args write 1000**
  Ορίζει τα ορίσματα που θα χρησιμοποιηθούν στην εκτέλεση του προγράμματος. Εδώ το πρόγραμμα kiwi-bench θα πραγματοποιήσει 1000 εγγραφές.

```
(gdb) set args write 10000
(gdb) run
Starting program: /home/myy601/kiwi/kiwi-source/bench/kiwi-bench write 10000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=3, argv=0x7fffffffe178) at bench.c:75
(gdb)
```

- **run**
  Ξεκινά την εκτέλεση του προγράμματος με τα ορίσματα που έχουν καθοριστεί από το set args.

- **next**
  Προχωρά στην επόμενη εντολή του προγράμματος σε επίπεδο C, εκτελώντας ολόκληρες συναρτήσεις χωρίς να μπαίνει μέσα στις κλήσεις τους.
- **nexti**
  Παρόμοια με το next, αλλά λειτουργεί σε επίπεδο assembly, επιτρέποντας βηματική εκτέλεση σε χαμηλότερο επίπεδο.
- **bt** (Backtrace)
  Εμφανίζει τη call stack, δείχνοντας τις συναρτήσεις που έχουν κληθεί μέχρι το σημείο που βρίσκεται η εκτέλεση. Αυτό βοηθάει στον εντοπισμό του πώς έφτασε το πρόγραμμα σε ένα συγκεκριμένο σημείο.

```
(gdb) bt
#0  main (argc=3, argv=0x7fffffffe178) at bench.c:75
```

- **info thread**
  Εμφανίζει πληροφορίες για τα νήματα που εκτελούνται στο πρόγραμμα.

```
(gdb) info thread
  Id   Target Id                                          Frame
* 1    Thread 0x7ffff7cc7740 (LWP 4201) "kiwi-bench" main (argc=3,
    argv=0x7fffffffe178) at bench.c:75
```

ΤΡΟΠΟΠΟΙΗΣΗ ΑΡΧΕΙΟΥ BENCH.C

Ορίσαμε αρχικά τα παρακάτω global variables, καθώς και τις μεθόδους initialize_mutexes και destroy_mutexes, οι οποίες αρχικοποιούν και διαγράφουν τις κλειδαριές αντίστοιχα και καλούνται στη main μέθοδο. Οι μεταβλητές writers και readers χρησιμοποιούνται για τη διαχείριση του συγχρονισμού μεταξύ των νημάτων. Το writers παρακολουθεί τον αριθμό των ενεργών νημάτων που εκτελούν την λειτουργία add , διασφαλίζοντας ότι μόνο ένα νήμα μπορεί να εκτελεί λειτουργίες τη φορά για να διατηρήσει τη συνέπεια των δεδομένων. Οι αναγνώστες-νήματα παρακολουθούν τον αριθμό των ενεργών νημάτων που εκτελούν την λειτουργία get , επιτρέποντας σε πολλούς αναγνώστες να έχουν πρόσβαση στη βάση δεδομένων ταυτόχρονα, εφόσον δεν είναι ενεργός κανένας συγγραφέας-νήμα.

```c
DB* db;
pthread_mutex_t readwrite_mutex;
pthread_mutex_t compaction_mutex;
pthread_cond_t readwrite_cond;
int writers;
int readers;

void initialize_mutexes() {
    pthread_mutex_init(&readwrite_mutex, NULL);
    pthread_mutex_init(&compaction_mutex, NULL);
    pthread_cond_init(&readwrite_cond, NULL);

}

void destroy_mutexes() {
    pthread_mutex_destroy(&readwrite_mutex);
    pthread_mutex_destroy(&compaction_mutex);
    pthread_cond_destroy(&readwrite_cond);
}
```

Πιο συγκεκριμένα καλείται η πρώτη μέθοδος πριν την μονονηματική ή πολυνηματική λειτουργία:

```
int main(int argc,char** argv)
{
    writers = 0;
    long int count;
    int threads;
    int write_percentage;
    initialize_mutexes();
```

Και ύστερα η δεύτερη μέθοδος καλείται αμέσως μετά το τέλος της λειτουργίας, είτε πολυνηματική είτε μονονηματική.

```
else {
    fprintf(stderr,"Usage: db-bench <write | read> <count> <random>\n");
    exit(1);
}

destroy_mutexes();

return 1;
```

ΥΛΟΠΟΙΗΣΗ ADD ΚΑΙ GET ΜΕ ΠΟΛΛΑΠΛΑ ΝΗΜΑΤΑ

Το αρχικό αρχείο που τροποποιήσαμε είναι το bench.c και συγκεκριμένα την main μέθοδο. Αρχικά προσθέσαμε ένα else if condition το οποίο ελέγχει εάν έχουμε πολυνηματισμό.

```
} else if (strcmp(argv[1], "multi_write") == 0) {

    if (argc < 4) {
        fprintf(stderr,"Usage: db-bench <multi_write | multi_read> <count> <number of threads>\n");
        exit(1);
    }

    int r = 0;



    count = atoi(argv[2]);
    threads = atoi(argv[3]);
    _print_header(count);
    _print_environment();
    if (argc == 5)
        r = 1;
    db = db_open(DATAS);

    multi_write_test(count, r, threads);


}
```

Το παραπάνω `if (strcmp(argv[1], "multi_write") == 0)" χειρίζεται λειτουργίες εγγραφής πολλαπλών νημάτων. Απαιτεί τον αριθμό των πράξεων και τον αριθμό των νημάτων ως ορίσματα από τον χρήστη. Αρχικοποιεί τη βάση δεδομένων,

ρυθμίζει το περιβάλλον και καλεί το 'multi_write_test' για να εκτελέσει ταυτόχρονες εγγραφές, το οποίο βρίσκεται στο αρχείο kiwi.c.

```
else if (strcmp(argv[1], "multi_read") == 0) {
    if (argc < 4) {
        fprintf(stderr,"Usage: db-bench <multi_write | multi_read> <count> <number of threads>\n");
        exit(1);
    }

    int r = 0;



    count = atoi(argv[2]);
    threads = atoi(argv[3]);
    _print_header(count);
    _print_environment();
    if (argc == 5)
        r = 1;
    db = db_open(DATAS);

    multi_read_test(count, r, threads);
}
```

Το παραπάνω `if (strcmp(argv[1], "multi_read") == 0)" χειρίζεται λειτουργίες ανάγνωσης πολλαπλών νημάτων. Ζητάει τον αριθμό των πράξεων και τον αριθμό των νημάτων ως ορίσματα. Αρχικοποιεί τη βάση δεδομένων, ρυθμίζει το περιβάλλον και καλεί το «multi_read_test» για να εκτελέσει ταυτόχρονες αναγνώσεις, το οποίο βρίσκεται στο kiwi.c.

Τροποποίηση αρχειου bench.h

Προσθέσαμε όλες τις απαραίτητες βιβλιοθήκες με σκοπό τα αρχεία bench.c, kiwi.c να έχουν πρόσβαση στις παρακάτω βιβλιοθήκες.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#include <pthread.h>
#include "../engine/db.h"
```

Έπειτα ορίσαμε την δομή thread_data_t η οποία έχει σχεδιαστεί για να διατηρεί τα ορίσματα που απαιτούνται για κάθε νήμα. Περιλαμβάνει έναν δείκτη στη βάση δεδομένων (db), το εύρος των λειτουργιών που έχουν εκχωρηθεί στο νήμα (αρχή και τέλος), μια σημαία (r) για να υποδεικνύεται εάν οι λειτουργίες πρέπει να χρησιμοποιούν τυχαία κλειδιά, το μοναδικό αναγνωριστικό του νήματος (thread_id) και τον συνολικό αριθμό νημάτων (num_threads) που υλοποιούνται.

Αυτή η δομή διασφαλίζει ότι κάθε νήμα έχει όλες τις απαραίτητες πληροφορίες για να εκτελεί τις εργασίες που του έχουν ανατεθεί ανεξάρτητα.

```c
typedef struct { // thread arguments
    DB* db;
    long int begin, end;
    int r;
    int thread_id;
    int num_threads;
} thread_data_t;



void multi_write_test(long int count, int r, int num_threads);
void *thread_write(void * arg);
void multi_read_test(long int count, int r, int num_threads);
void *thread_read(void * arg);
void readwrite_test (long int count, int r, int threads, int write_percentage);
```

Ορίσαμε τις παραπάνω μεθόδους έτσι ώστε να είναι ορατές στο Kiwi.c και σε όλα τα αρχεία που έχουν πρόσβαση στο bench.h.

ΤΡΟΠΟΠΟΙΗΣΗ ΑΡΧΕΊΟΥ KIWI.C

Αρχικά οι μεταβλητές «write_mutex» και «read_mutex» είναι mutexes που χρησιμοποιούνται για τον συγχρονισμό της πρόσβασης σε κοινόχρηστους πόρους κατά τη διάρκεια λειτουργιών εγγραφής και ανάγνωσης πολλαπλών νημάτων, διασφαλίζοντας την ασφάλεια των νημάτων και αποτρέποντας τις συνθήκες αγώνα. Οι μεταβλητές "added" και "found" είναι μετρητές που παρακολουθούν τον συνολικό αριθμό επιτυχών λειτουργιών εγγραφής και τον αριθμό των κλειδιών που βρέθηκαν κατά τις λειτουργίες ανάγνωσης, αντίστοιχα. Αυτές οι μεταβλητές είναι κρίσιμες για τη διατήρηση ακριβών στατιστικών στοιχείων.

```c
pthread_mutex_t write_mutex;
pthread_mutex_t read_mutex;

long int added;
long int found;
```

ΥΛΟΠΟΙΗΣΗ ADD ΛΕΙΤΟΥΡΓΊΑΣ

ΔΗΜΙΟΥΡΓΙΑ ΜΕΘΟΔΟΥ MULTI_WRITE_TEST() ΣΤΟ KIWI.C

Η μέθοδος «multi_write_test()» εκτελεί λειτουργίες εγγραφής πολλαπλών νημάτων στη βάση δεδομένων. Αρχικοποιεί μια σειρά από ορίσματα νήματος («thread_args») και αναγνωριστικά νημάτων («tid»), υπολογίζει το εύρος των λειτουργιών για κάθε

νήμα και δημιουργεί πολλαπλά νήματα εγγραφής χρησιμοποιώντας το «pthread_create». Κάθε νήμα εκτελεί τη συνάρτηση «thread_write» για να εκτελέσει τις λειτουργίες εγγραφής που του έχουν ανατεθεί. Αφού δημιουργηθούν όλα τα νήματα, η μέθοδος περιμένει να τελειώσουν χρησιμοποιώντας το «pthread_join». Υπολογίζει τον συνολικό χρόνο εκτέλεσης, εκτυπώνει στατιστικά στοιχεία απόδοσης και καθαρίζει πόρους καταστρέφοντας το mutex, ελευθερώνοντας την εκχωρημένη μνήμη και κλείνοντας τη βάση δεδομένων. Αυτή η μέθοδος επιτρέπει την αποτελεσματική συγκριτική αξιολόγηση της απόδοσης ταυτόχρονης εγγραφής.

```c
void multi_write_test(long int count,int r, int threads) {
    extern DB* db;       // DB OPENED IN MAIN
    thread_data_t* thread_args = malloc(threads * sizeof(thread_data_t));
    pthread_t* tid = malloc(threads * sizeof(pthread_t));      // array of thread ids
    long long start, end;        // start time and end time
    double cost;        // the total time of operations


    pthread_mutex_init(&write_mutex, NULL); // initialize the stats mutex for writing
    start = get_ustime_sec();        // calculate starting time for the operation
    for (int i = 0; i < threads; i++) {
        // calculate the thread arguments for each thread
        thread_args[i].db = db;
        thread_args[i].begin = i * count / threads;
        thread_args[i].end = (i + 1) * count / threads;
        thread_args[i].r = r;
        thread_args[i].thread_id = i;
        thread_args[i].num_threads = threads;
        // create writing threads
        pthread_create(&tid[i], NULL, thread_write, &thread_args[i]);

    }
    for (int i = 0; i < threads; i++) {
        pthread_join(tid[i], NULL);
    }
    db_close(db);            // close the database
    end = get_ustime_sec();            // calculate end time of the operation
    cost = end - start;            // calculate operating time

    printf(LINE);
    printf("|Random-Multi-Thread-Write (done:%ld, added:%ld): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec); Threads: %d\n",
       count, added, (double)(cost / count),
       (double)(count / cost),
       cost, threads);

    pthread_mutex_destroy(&write_mutex);

    free(thread_args);
    free(tid);
}
```

ΔΗΜΙΟΥΡΓΙΑ ΜΕΘΟΔΟΥ THREAD_WRITE()

Η συνάρτηση «thread_write()» εκτελεί λειτουργίες εγγραφής σε ένα περιβάλλον πολλαπλών νημάτων, όπου κάθε νήμα γράφει μια σειρά από ζεύγη κλειδιών-τιμών στη βάση δεδομένων. Η κρίσιμη περιοχή αυτής της συνάρτησης είναι το τμήμα όπου καλείται η συνάρτηση «db_add» και ενημερώνεται η global ακέραια μεταβλητή «added». Αυτή η περιοχή προστατεύεται από το «write_mutex» για να διασφαλιστεί η ασφάλεια του νήματος.

Συγκεκριμένα, η κρίσιμη ενότητα ξεκινά με "pthread_mutex_lock(&write_mutex)" και τελειώνει με "pthread_mutex_unlock(&write_mutex)". Μέσα σε αυτήν την ενότητα, η συνάρτηση «db_add» γράφει το δημιουργημένο ζεύγος κλειδιού-τιμής στη βάση δεδομένων και ακέραια μεταβλητή «added» αυξάνεται κατά 1 για να παρακολουθεί τον συνολικό αριθμό επιτυχημένων εγγραφών. Χωρίς αυτό το mutex, πολλά νήματα θα μπορούσαν ταυτόχρονα να έχουν πρόσβαση και να τροποποιούν κοινόχρηστους πόρους («db_add» και «added»), οδηγώντας σε race conditions , deadlocks, καταστροφή δεδομένων ή ασυνεπή αποτελέσματα. Με το κλείδωμα του mutex, μόνο ένα νήμα μπορεί να εκτελέσει αυτήν την ενότητα κάθε φορά, διασφαλίζοντας ότι η βάση δεδομένων και το added παραμένουν συνεπείς και ακριβείς.

```c
void *thread_write(void * arg) {
    thread_data_t *data = (thread_data_t *) arg;
    int i;
    double cost;
    long long start, end;
    Variant sk, sv;

    char key[KSIZE + 1];
    char val[VSIZE + 1];
    char sbuf[1024];

    memset(key, 0, KSIZE + 1);
    memset(val, 0, VSIZE + 1);
    memset(sbuf, 0, 1024);

    start = get_ustime_sec(); // start time of the operation
    for (i = data->begin; i < data->end; i++) { // loop from count_begin of thread to count end
        if (data->r)
            _random_key(key, KSIZE);
        else
            snprintf(key, KSIZE, "key-%d", i);
        fprintf(stderr, "%d adding %s\n", i, key);
        snprintf(val, VSIZE, "val-%d", i);

        sk.length = KSIZE;
        sk.mem = key;
        sv.length = VSIZE;
        sv.mem = val;
        pthread_mutex_lock(&write_mutex); // lock the statistics mutex for writing

        db_add(data -> db, &sk, &sv);
        added++;

        pthread_mutex_unlock(&write_mutex); // unlock the statistics mutex for writing
```

```c
        if ((i % 10000) == 0) {
            fprintf(stderr, "random write finished %d ops%30s\r",
                    i,
                    "");

            fflush(stderr);
        }

    }

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

    return NULL;
}
```

ΕΚΧΩΡΗΣΗ ΜΕΤΑΒΛΗΤΩΝ ΣΤΟ ΑΡΧΕΊΟ DB.C

Οι μεταβλητές compaction_mutex, readwrite_mutex και readwrite_cond χρησιμοποιούνται για τον συγχρονισμό της πρόσβασης σε κοινόχρηστους πόρους στη βάση δεδομένων κατά τη διάρκεια λειτουργιών πολλαπλών νημάτων. Το compaction_mutex διασφαλίζει ότι μόνο ένα νήμα μπορεί να εκτελεί compaction τη φορά, αποτρέποντας τις συγκρούσεις κατά τη συγχώνευση του memtable στο SST. Το readwrite_mutex και το readwrite_cond χρησιμοποιούνται για τον συντονισμό αναγνωστών και εγγραφών, διασφαλίζοντας ότι οι αναγνώστες μπορούν να έχουν πρόσβαση στη βάση δεδομένων ταυτόχρονα ενώ αποκλείουν τους writers και

αντίστροφα. Οι μεταβλητές writers και readers παρακολουθούν τον αριθμό των ενεργών νημάτων συγγραφέα και αναγνώστη, αντίστοιχα, επιτρέποντας τον σωστό συγχρονισμό add-get. Η μεταβλητή compacting υποδεικνύει εάν μια διαδικασία compaction βρίσκεται σε εξέλιξη αυτήν τη στιγμή. Μαζί, αυτές οι μεταβλητές διασφαλίζουν λειτουργίες ασφαλείς ως προς το νήμα και διατηρούν τη συνέπεια των δεδομένων σε ένα περιβάλλον πολλαπλών νημάτων. Το extern χρησιμοποιείται για τη δήλωση μεταβλητών που ορίζονται σε άλλο αρχείο, επιτρέποντάς τους να κοινοποιούνται σε πολλαπλά αρχεία, διασφαλίζοντας σωστό συγχρονισμό. Οι μεταβλητές που έχουν το extern δίπλα από τον τύπο τους, ορίζονται στο αρχείο bench.c.

```
extern pthread_mutex_t compaction_mutex;
extern pthread_mutex_t readwrite_mutex;
extern pthread_cond_t readwrite_cond;
extern int writers;
extern int readers;
int compacting;
```

ΤΡΟΠΟΠΟΊΗΣΗ ΜΕΘΟΔΟΥ DB_ADD() ΤΟΥ ΑΡΧΕΙΟΥ DB.C

Η μέθοδος «db_add» είναι υπεύθυνη για την προσθήκη ενός ζεύγους κλειδιού-τιμής στη βάση δεδομένων, διασφαλίζοντας παράλληλα την ασφάλεια του νήματος και τον σωστό συγχρονισμό. Αρχικά κλειδώνει το "compaction_mutex" για να ελέγξει εάν το memtable χρειάζεται συμπίεση. Εάν απαιτείται συμπίεση, κλειδώνει το readwrite_mutex για να διασφαλίσει ότι δεν είναι ενεργοί άλλοι εγγραφείς, εκτελεί τη συμπύκνωση συγχωνεύοντας το memtable στο SST, επαναφέρει το memtable και, στη συνέχεια, μειώνει τον μετρητή writers ενώ σηματοδοτεί άλλα νήματα χρησιμοποιώντας το pthread_cond_broadcast. Μετά τη συμπύκνωση, η μέθοδος προσθέτει το ζεύγος κλειδιού-τιμής στο memtable. Οι κρίσιμες ενότητες προστατεύονται από το readwrite_mutex για να διασφαλιστεί ότι μόνο ένας συγγραφέας μπορεί να εκτελεί λειτουργίες κάθε φορά, αποτρέποντας τις race conditions. Αυτές οι ενότητες περιλαμβάνουν την αναμονή για να τελειώσουν οι άλλοι συντάκτες, την αύξηση και τη μείωση του μετρητή "writers" και τη σηματοδότηση των νημάτων αναμονής για να προχωρήσουν. Αυτό διασφαλίζει ότι οι λειτουργίες εγγραφής και συμπίεσης εκτελούνται με ασφάλεια σε περιβάλλον πολλαπλών νημάτων.

ΚΡΊΣΙΜΕΣ ΠΕΡΙΟΧΈΣ

- Έλεγχος συμπίεσης (compaction_mutex)** Η πρώτη κρίσιμη περιοχή ξεκινά με «pthread_mutex_lock(&compaction_mutex)» και διασφαλίζει ότι μόνο ένα νήμα μπορεί να ελέγχει και να εκτελεί συμπίεση κάθε φορά. Εάν το memtable χρειάζεται συμπίεση, το νήμα προχωρά στο κλείδωμα του «readwrite_mutex» για να αποκλείσει άλλους readers και writers. Αυτή η περιοχή αποτρέπει τις race conditions κατά τη διάρκεια της διαδικασίας συμπίεσης, όπου το memtable συγχωνεύεται στο SST και επαναφέρεται. Με το κλείδωμα του

«compaction_mutex», το πρόγραμμα διασφαλίζει ότι η συμπίεση εκτελείται με ασφάλεια χωρίς παρεμβολές από άλλα νήματα.

- Συγχρονισμός συγγραφέα κατά το compaction (readwrite_mutex)** Όταν απαιτείται compaction, το νήμα κλειδώνει το readwrite_mutex και περιμένει μέχρι να μην είναι ενεργοί άλλοι εγγραφείς (writers == 0). Αυτό διασφαλίζει ότι το compaction μπορεί να πραγματοποιηθεί χωρίς παρεμβολές από ταυτόχρονες λειτουργίες εγγραφής. Αφού αυξηθεί ο μετρητής "writers" για να υποδείξει ότι ένας συγγραφέας εκτελεί compaction, το νήμα ξεκλειδώνει το mutex, εκτελεί compaction και στη συνέχεια μειώνει τον μετρητή "writers". Τέλος, σηματοδοτεί άλλα νήματα αναμονής χρησιμοποιώντας το «pthread_cond_broadcast».

- Προσθήκη ζεύγους κλειδιού-τιμής (readwrite_mutex)** Πριν προσθέσουμε ένα ζεύγος κλειδιού-τιμής στο memtable, το νήμα κλειδώνει το readwrite_mutex και περιμένει μέχρι να μην είναι ενεργοί άλλοι συντάκτες (writers == 0). Αυτό διασφαλίζει ότι μόνο ένας συγγραφέας μπορεί να τροποποιήσει το memtable κάθε φορά. Μετά την αύξηση του μετρητή «writers», το νήμα εκτελεί τη λειτουργία εγγραφής και στη συνέχεια μειώνει τον μετρητή. Σηματοδοτεί άλλα νήματα αναμονής χρησιμοποιώντας το "pthread_cond_broadcast". Αυτό το κρίσιμο τμήμα αποτρέπει race conditions κατά τις λειτουργίες εγγραφής, διασφαλίζοντας ότι το memtable παραμένει συνεπές.

- Κρίσιμη περιοχή για συγχρονισμό συγγραφέα Αυτή η Κρίσιμη περιοχή είναι υπεύθυνη για τη σηματοδότηση άλλων νημάτων ότι ένας συγγραφέας έχει ολοκληρώσει τη λειτουργία του. Ξεκινά με το κλείδωμα του readwrite_mutex για να διασφαλιστεί η ασφαλής πρόσβαση στο νήμα στον κοινόχρηστο μετρητή εγγραφών (writers). Ο μετρητής συγγραφέων μειώνεται για να υποδείξει ότι ο τρέχων συγγραφέας έχει ολοκληρώσει την εργασία του. Στη συνέχεια, το pthread_cond_broadcast καλείται να αφυπνίσει όλα τα νήματα που περιμένουν στη μεταβλητή συνθήκης readwrite_cond, επιτρέποντάς τους να προχωρήσουν εάν πληρούνται οι προϋποθέσεις (π.χ., δεν υπάρχουν ενεργοί εγγραφείς). Τέλος, το mutex ξεκλειδώνεται για να επιτραπεί σε άλλα νήματα να έχουν πρόσβαση στους κοινόχρηστους πόρους. Αυτή η ενότητα διασφαλίζει τον σωστό συγχρονισμό μεταξύ των νημάτων, αποτρέποντας τις συνθήκες αγώνα και επιτρέποντας ομαλές μεταβάσεις μεταξύ συγγραφέων και αναγνωστών.

```c
int db_add(DB* self, Variant* key, Variant* value)
{
    pthread_mutex_lock(&compaction_mutex);

    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);

        pthread_mutex_lock(&readwrite_mutex);
        while(writers == 1){
            pthread_cond_wait(&readwrite_cond,&readwrite_mutex);
        }
        writers ++;
        pthread_mutex_unlock(&readwrite_mutex);
        sst_merge(self->sst, self->memtable);

        memtable_reset(self->memtable);

        // signal any waiting writers
        pthread_mutex_lock(&readwrite_mutex);
        writers --;

        pthread_cond_broadcast(&readwrite_cond);
        pthread_mutex_unlock(&readwrite_mutex);
    }
```

```c
    int result = memtable_add(self->memtable, key, value);

    pthread_mutex_unlock(&compaction_mutex);

    //lock gia na graphei enas ka8e fora
    pthread_mutex_lock(&readwrite_mutex);
    while (writers == 1)
    {
        pthread_cond_wait(&readwrite_cond, &readwrite_mutex);
    }
    writers ++;
    pthread_mutex_unlock(&readwrite_mutex);

    //ksypna tous ypoloipous
    pthread_mutex_lock(&readwrite_mutex);
    writers --;
    pthread_cond_broadcast(&readwrite_cond);
    pthread_mutex_unlock(&readwrite_mutex);



    return result;
}
```

Υλοποιηση GET λειτουργιας

Δημιουργια μεθοδου multi_read_test() στο kiwi.c

Η μέθοδος «multi_read_test» εκτελεί λειτουργίες ανάγνωσης πολλαπλών νημάτων στη βάση δεδομένων. Αρχικοποιεί μια σειρά από ορίσματα νήματος («thread_args») και αναγνωριστικά νημάτων («tid»), υπολογίζει το εύρος των κλειδιών που θα διαβάσει κάθε νήμα και δημιουργεί πολλαπλά νήματα ανάγνωσης χρησιμοποιώντας το «pthread_create». Κάθε νήμα εκτελεί τη συνάρτηση «thread_read» για να εκτελέσει τις λειτουργίες ανάγνωσης που του έχουν ανατεθεί. Αφού δημιουργηθούν όλα τα νήματα, η μέθοδος περιμένει να τελειώσουν χρησιμοποιώντας το «pthread_join». Υπολογίζει τον συνολικό χρόνο εκτέλεσης, εκτυπώνει στατιστικά στοιχεία απόδοσης (συμπεριλαμβανομένου του αριθμού των κλειδιών που βρέθηκαν) και καθαρίζει τους πόρους καταστρέφοντας το mutex, ελευθερώνοντας την εκχωρημένη μνήμη και κλείνοντας τη βάση δεδομένων.

```c
void multi_read_test(long int count, int r, int threads) {
    extern DB* db;       // DB OPENED IN MAIN
    thread_data_t* thread_args = malloc(threads * sizeof(thread_data_t));
    pthread_t* tid = malloc(threads * sizeof(pthread_t));      // array of thread ids
    long long start, end;        // start time and end time
    double cost;        // the total time of operations


    pthread_mutex_init(&read_mutex, NULL); // initialize the stats mutex for writing
    start = get_ustime_sec();        // calculate starting time for the operation
    for (int i = 0; i < threads; i++) {
        // calculate the thread arguments for each thread
        thread_args[i].db = db;
        thread_args[i].begin = i * count / threads;
        thread_args[i].end = (i + 1) * count / threads;
        thread_args[i].r = r;
        thread_args[i].thread_id = i;
        thread_args[i].num_threads = threads;
        // create reading threads
        pthread_create(&tid[i], NULL, thread_read, &thread_args[i]);
    }
    for (int i = 0; i < threads; i++) {
        pthread_join(tid[i], NULL);
    }
    db_close(db);            // close the database
    end = get_ustime_sec();           // calculate end time of the operation
    cost = end - start;           // calculate operating time

    printf(LINE);
    printf("|Random-Multi-Thread-Read (done:%ld, found:%ld): %.6f sec/op; %.1f reads/sec(estimated); cost:%.3f(sec); Threads: %d\n",
           count,found, (double)(cost / count),
           (double)(count / cost),
           cost, threads);

    pthread_mutex_destroy(&read_mutex);

    free(thread_args);
    free(tid);
}
```

Δημιουργια μεθοδου thread_read()

Η μέθοδος «thread_read» εκτελεί λειτουργίες ανάγνωσης σε περιβάλλον πολλαπλών νημάτων, όπου κάθε νήμα διαβάζει μια σειρά κλειδιών από τη βάση δεδομένων. Δημιουργεί κλειδιά (είτε διαδοχικά είτε τυχαία, με βάση τη σημαία «r») και επιχειρεί να ανακτήσει τις αντίστοιχες τιμές τους χρησιμοποιώντας τη συνάρτηση «db_get». Η κρίσιμη περιοχή, που προστατεύεται από το «read_mutex», διασφαλίζει την ασφαλή πρόσβαση σε νήματα στην global μεταβλητή found, ο οποίος παρακολουθεί τον αριθμό των κλειδιών που ανακτήθηκαν επιτυχώς. Το mutex αποτρέπει πιθανά race conditions και deadlocks όταν πολλαπλά νήματα ενημερώνουν Την μεταβλητή ταυτόχρονα.

```c
void *thread_read(void *arg) {
    thread_data_t *data = (thread_data_t *) arg; // cast the arguments passed from void to thread_data_t
    int i;
    int ret;
    double cost;
    long long start, end;
    Variant sk, sv;

    char key[KSIZE + 1];
    char sbuf[1024];

    memset(key, 0, KSIZE + 1);
    memset(sbuf, 0, 1024);

    start = get_ustime_sec();
    for (i = data->begin; i < data->end; i++) {
        if (data->r)
            _random_key(key, KSIZE);
        else
            snprintf(key, KSIZE, "key-%d", i);
        fprintf(stderr, "%d searching %s\n", i, key);

        sk.length = KSIZE;
        sk.mem = key;

        pthread_mutex_lock(&read_mutex);



        ret = db_get(data->db, &sk, &sv);
        if (ret) {
            found++;
        }

        pthread_mutex_unlock(&read_mutex);
```

```c
261
262             if ((i % 10000) == 0) {
263                 fprintf(stderr, "random read finished %d ops%30s\r",
264                         i,
265                         "");
266
267                 fflush(stderr);
268             }
269         }
270
271         end = get_ustime_sec();
272         cost = end - start;
273
274
275
276
277
278         return NULL;
279     }
```

ΤΡΟΠΟΠΟΊΗΣΗ ΜΕΘΟΔΟΥ DB_GET() ΤΟΥ ΑΡΧΕΙΟΥ DB.C

Η μέθοδος db_get ανακτά ένα ζεύγος κλειδιού-τιμής από τη βάση δεδομένων, ενώ παράλληλα διασφαλίζει την ασφάλεια του νήματος χρησιμοποιώντας έναν μηχανισμό συγχρονισμού read-write. Αρχικά κλειδώνει το «readwrite_mutex» και αυξάνει τον μετρητή «readers» για να υποδείξει ότι ένας reader είναι ενεργός. Εάν υπάρχουν ενεργοί writers (writers >= 1), ο αναγνώστης περιμένει τη μεταβλητή συνθήκης readwrite_cond μέχρι να τελειώσουν οι συγγραφείς. Μόλις επιτραπεί, ξεκλειδώνει το mutex και επιχειρεί να ανακτήσει το κλειδί από το memtable χρησιμοποιώντας το

«memtable_get». Εάν το κλειδί δεν βρεθεί στο memtable, αναζητά το SST χρησιμοποιώντας το «sst_get». Μετά τη λειτουργία ανάγνωσης, κλειδώνει ξανά το mutex, μειώνει τον μετρητή «readers» και σηματοδοτεί τα νήματα αναμονής εάν δεν υπάρχουν αναγνώστες. Αυτή η μέθοδος διασφαλίζει ότι οι αναγνώστες μπορούν να έχουν πρόσβαση στη βάση δεδομένων ταυτόχρονα, ενώ αποκλείουν τους εγγραφείς, διατηρώντας τη συνέπεια των δεδομένων και την ασφάλεια των νημάτων.

```
int db_get(DB* self, Variant* key, Variant* value)
{
    pthread_mutex_lock(&readwrite_mutex);
    readers ++;

    while (writers >= 1) {
        pthread_cond_wait(&readwrite_cond, &readwrite_mutex);
    }
    pthread_mutex_unlock(&readwrite_mutex);

    int result = memtable_get(self->memtable->list, key, value); //kaneis read sthn mnhmh
    pthread_mutex_lock(&readwrite_mutex);
    readers --;
    if(readers == 0){
        pthread_cond_broadcast(&readwrite_cond);
    }

    pthread_mutex_unlock(&readwrite_mutex);
    if (result == 1)
        return 1;

    result = sst_get(self->sst, key, value); //an de to vrei sto memtable psaxnei sto sst
    return result;
}
```

ΚΡΊΣΙΜΕΣ ΠΕΡΙΟΧΈΣ

- Η πρώτη κρίσιμη περιοχή ξεκινά με "pthread_mutex_lock(&readwrite_mutex)" και τελειώνει με "pthread_mutex_unlock(&readwrite_mutex)". Σε αυτήν την περιοχή η global μεταβλητή readers αυξάνεται για να υποδείξει ότι ένας νέος αναγνώστης είναι ενεργός. Εάν υπάρχουν ενεργοί εγγραφείς (writers >= 1), το νήμα περιμένει στη μεταβλητή συνθήκης readwrite_cond μέχρι να τελειώσουν οι εγγραφείς. Αυτό διασφαλίζει ότι οι αναγνώστες δεν παρεμβαίνουν στις τρέχουσες λειτουργίες εγγραφής. Μόλις πληρούνται οι προϋποθέσεις, το mutex ξεκλειδώνεται, επιτρέποντας στον αναγνώστη να προχωρήσει στη λειτουργία ανάγνωσης.

- Η δεύτερη κρίσιμη περιοχή ξεκινά με "pthread_mutex_lock(&readwrite_mutex)" και τελειώνει με "pthread_mutex_unlock(&readwrite_mutex)". Μετά την ολοκλήρωση της λειτουργίας ανάγνωσης, η global μεταβλητή «readers» μειώνεται για να υποδείξει ότι ο αναγνώστης έχει τελειώσει. Εάν ο μετρητής "readers" φτάσει στο μηδέν, το νήμα καλεί το "pthread_cond_broadcast(&readwrite_cond)" για να ειδοποιήσει τους συντάκτες που αναμένουν ότι δεν είναι ενεργοί αναγνώστες, επιτρέποντάς τους να συνεχίσουν. Αυτή η περιοχή διασφαλίζει τον σωστό συγχρονισμό μεταξύ αναγνωστών και συγγραφέων, αποτρέποντας

τις race conditions και διασφαλίζοντας ότι οι συγγραφείς μπορούν να τροποποιήσουν με ασφάλεια τη βάση δεδομένων όταν δεν υπάρχουν ενεργοί αναγνώστες.

ΥΛΟΠΟΙΗΣΗ ΤΑΥΤΟΧΡΟΝΗΣ ADD ΚΑΙ GET ΛΕΙΤΟΥΡΓΊΑΣ ΑΡΧΕΙΟΥ BENCH.C

Αρχικά προσθέσαμε μια νέα if συνθήκη στην main μέθοδο του αρχείου bench.c, όπως φαίνεται στην εικόνα παρακάτω :

```
else if (strcmp(argv[1], "readwrite") == 0) {
    if (argc < 5) {
        fprintf(stderr,"Usage: db-bench <readwrite> <count> <number of threads> <write percentage>\n");
        exit(1);
    }

    int r = 0;

    count = atoi(argv[2]);
    threads = atoi(argv[3]);
    write_percentage = atoi(argv[4]);



    _print_header(count);
    _print_environment();
    if (argc == 6)
        r = 1;

    db = db_open(DATAS);

    readwrite_test(count, r, threads, write_percentage);



}
```

Χειρίζεται τη λειτουργία «readwrite» . Αναλύει τα ορίσματα της γραμμής εντολών για να ανακτήσει τον συνολικό αριθμό πράξεων («count»), τον αριθμό των νημάτων («νήματα») και το ποσοστό των νημάτων που αφιερώνονται στη γραφή («write_percentage»). Εάν παρέχεται ένα έκτο όρισμα, ενεργοποιεί την τυχαία δημιουργία κλειδιού. Η μέθοδος αρχικοποιεί τη βάση δεδομένων, εκτυπώνει πληροφορίες συστήματος και συγκριτικής αξιολόγησης και καλεί τη συνάρτηση «readwrite_test» για να εκτελέσει ταυτόχρονες λειτουργίες ανάγνωσης και εγγραφής με βάση την καθορισμένη κατανομή νημάτων.

ΤΡΟΠΟΠΟΊΗΣΗ ΤΟΥ ΑΡΧΕΙΟΥ KIWI.C

Αρχικά δημιουργήσαμε την μέθοδο write_ceil(),η οποία χρησιμοποιείται για τον υπολογισμό του ανώτατου ορίου του ποσοστού των νημάτων που διατίθενται για εγγραφή. Συγκεκριμένα, το (νήματα * write_percentage) / 100,0 υπολογίζει το κλάσμα των συνολικών νημάτων που αφιερώνονται στη γραφή και το "write_ceil" διασφαλίζει ότι οποιαδήποτε κλασματική τιμή στρογγυλοποιείται στον πλησιέστερο ακέραιο αριθμό. Το αποτέλεσμα αποθηκεύεται σε wthreads και στη συνέχεια μεταδίδεται σε έναν ακέραιο ως write_threads. Αυτό διασφαλίζει ότι ο αριθμός των νημάτων εγγραφής είναι πάντα ένας ακέραιος αριθμός, ακόμα κι αν ο υπολογισμός του ποσοστού έχει ως αποτέλεσμα μια κλασματική τιμή. Αυτή η λογική είναι κρίσιμη για τη σωστή κατανομή των νημάτων μεταξύ των πράξεων ανάγνωσης και εγγραφής στη μέθοδο «readwrite_test».

```
int write_ceil(double num) {
    int inum = (int)num;
    if (num > inum) {
        return inum + 1;
    }
    return inum;
}
```

Έπειτα υλοποιήσαμε τη μέθοδο read_write_test(), η οποία αξιολογεί την απόδοση ταυτόχρονων πράξεων ανάγνωσης και εγγραφής σε ένα περιβάλλον πολλαπλών νημάτων. Υπολογίζει τον αριθμό των νημάτων που αφιερώνονται στη γραφή και την ανάγνωση με βάση το συνολικό αριθμό νημάτων και το καθορισμένο ποσοστό εγγραφής. Αρχικοποιεί ορίσματα νήματος και δημιουργεί ξεχωριστά νήματα για ανάγνωση και γραφή χρησιμοποιώντας το «pthread_create». Η μέθοδος παρακολουθεί τους χρόνους έναρξης και λήξης και για τις δύο λειτουργίες για να υπολογίσει το ανεξάρτητο κόστος τους. Αφού όλα τα νήματα ολοκληρώσουν τις εργασίες τους, εκτυπώνει στατιστικά στοιχεία απόδοσης, συμπεριλαμβανομένων των συνολικών λειτουργιών, της απόδοσης εγγραφής και ανάγνωσης και των χρόνων εκτέλεσης. Τέλος, καθαρίζει τους πόρους καταστρέφοντας τα mutexes, ελευθερώνοντας την εκχωρημένη μνήμη και κλείνοντας τη βάση δεδομένων.

ΕΓΓΡΑΦΕΣ

/KIWI-BENCH MULTI _WRITE 500000 1

RANDOM-MULTI-THREAD-WRITE (DONE:500000): 0.000012 SEC/OP;83333.3 WRITES/SEC(ESTIMATED); COST:6.000(SEC); THREADS: 1

/KIWI-BENCH MULTI _WRITE 500000 2

RANDOM-MULTI-THREAD-WRITE (DONE:500000): 0.000024 SEC/OP;41666.7 WRITES/SEC(ESTIMATED); COST:12.000(SEC); THREADS: 2

/KIWI-BENCH MULTI _WRITE 500000 4

RANDOM-MULTI-THREAD-WRITE (DONE:500000): 0.000016 SEC/OP;62500.0 WRITES/SEC(ESTIMATED); COST:8.000(SEC); THREADS: 4

/KIWI-BENCH MULTI _WRITE 500000 16

RANDOM-MULTI-THREAD-WRITE (DONE:500000): 0.000018 SEC/OP;55555.6 WRITES/SEC(ESTIMATED); COST:9.000(SEC); THREADS: 16

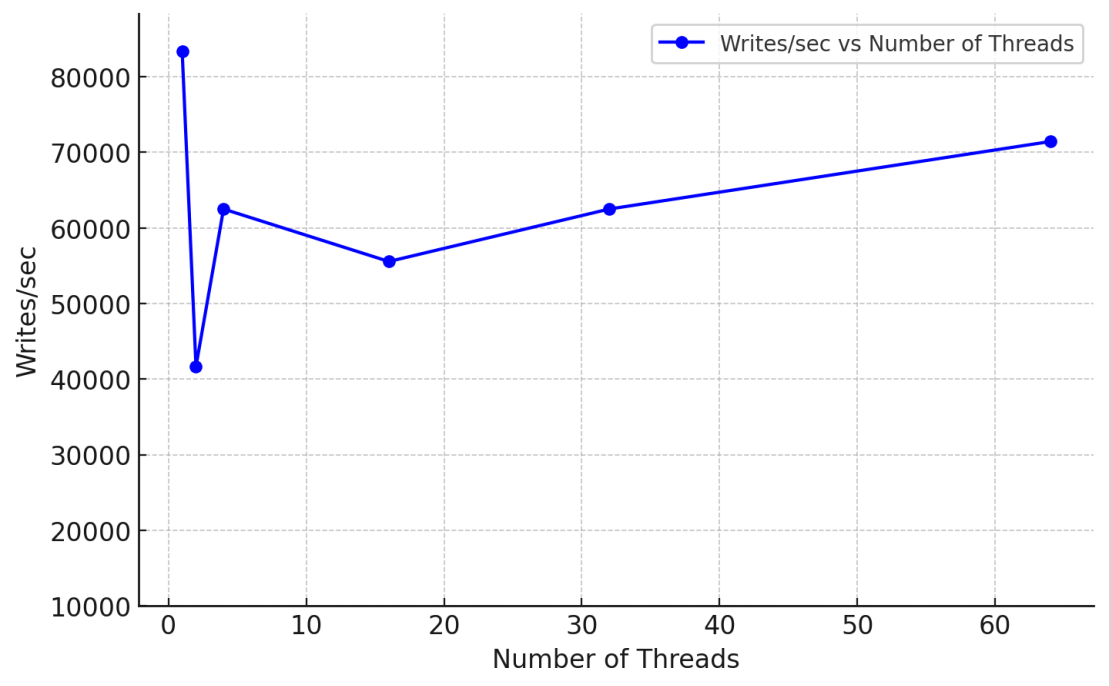
/KIWI-BENCH MULTI _WRITE 500000 32

RANDOM-MULTI-THREAD-WRITE (DONE:500000): 0.000016 SEC/OP;62500.0 WRITES/SEC(ESTIMATED); COST:8.000(SEC); THREADS: 32

/KIWI-BENCH MULTI _WRITE 500000 64

RANDOM-MULTI-THREAD-WRITE (DONE:500000): 0.000014 SEC/OP;71428.6 WRITES/SEC(ESTIMATED); COST:7.000(SEC); THREADS: 64

ΣΤΟ ΠΑΡΑΚΑΤΩ ΓΡΑΦΗΜΑ ΠΑΡΑΤΗΡΟΎΜΕ ΠΩΣ Η ΑΥΞΗΣΗ ΤΟΥ ΑΡΙΘΜΟΥ ΤΩΝ ΝΗΜΑΤΩΝ ΔΕΝ ΣΥΝΕΠΆΓΕΤΑΙ ΚΑΙ ΜΕ ΑΥΞΗΣΗ ΤΟΥ ΡΥΘΜΟΥ ΕΓΓΡΑΦΩΝ. ΑΥΤΟ ΣΥΜΒΑΙΝΕΙ ΛΟΓΩ ΤΟΥ ΟΤΙ ΟΙ ΕΓΓΡΑΦΕΣ ΓΙΝΟΝΤΑΙ ΣΕΙΡΙΑΚΑ.



ΑΝΑΓΝΩΣΕΙΣ

/KIWI-BENCH MULTI _READ 500000 1

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000012 SEC/OP;83333.3 READS/SEC(ESTIMATED); COST:6.000(SEC); THREADS: 1

/KIWI-BENCH MULTI _READ 500000 2

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000020 SEC/OP;50000.0 READS/SEC(ESTIMATED); COST:10.000(SEC); THREADS: 2

/KIWI-BENCH MULTI _READ 500000 4

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000014 SEC/OP;71428.6 READS/SEC(ESTIMATED); COST:7.000(SEC); THREADS: 4

/KIWI-BENCH MULTI _READ 500000 16

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000010 SEC/OP;100000.0 READS/SEC(ESTIMATED); COST:5.000(SEC); THREADS: 16

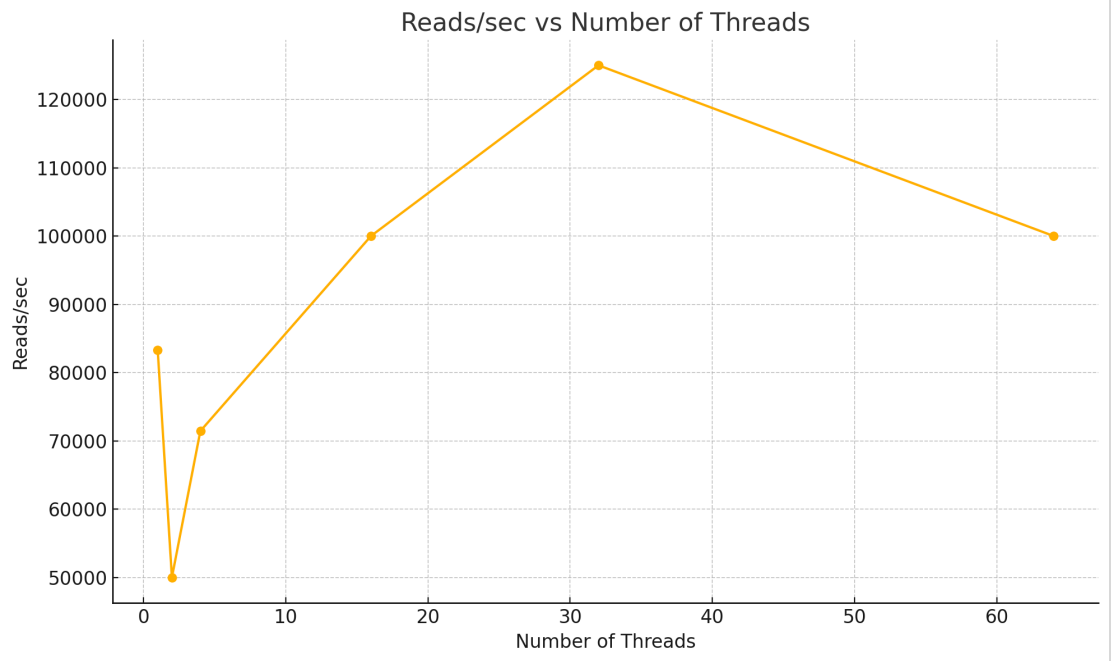
/KIWI-BENCH MULTI _READ 500000 32

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000008 SEC/OP;125000.0 READS/SEC(ESTIMATED); COST:4.000(SEC); THREADS: 32

/KIWI-BENCH MULTI _READ 500000 64

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000010 SEC/OP;100000.0 READS/SEC(ESTIMATED); COST:5.000(SEC); THREADS: 64

ΣΤΟ ΠΑΡΑΚΑΤΩ ΔΙΑΓΡΑΜΜΑ ΠΑΡΑΤΗΡΟΥΜΕ ΑΠΟ ΕΝΑ ΣΥΓΚΕΚΡΙΜΕΝΟ ΑΡΙΘΜΟ ΝΗΜΑΤΩΝ ΚΑΙ ΕΠΕΙΤΑ ΕΧΟΥΜΕ ΑΥΞΗΣΗ ΤΟΥ ΡΥΘΜΟΥ ΑΝΑΓΝΩΣΗΣ.
ΑΥΤΟ ΟΦΕΙΛΕΤΑΙ ΣΤΙΣ ΠΕΡΙΣΣΟΤΕΡΕΣ ΑΝΑΓΝΩΣΕΙΣ ΠΟΥ ΕΧΟΥΜΕ ΛΟΓΩ ΤΩΝ ΠΕΡΙΣΣΟΤΕΡΩΝ ΝΗΜΑΤΩΝ



ΤΑΥΤΟΧΡΟΝΕΣ ΕΓΓΡΑΦΕΣ-ΑΝΑΓΝΩΣΕΙΣ

50% OF THREADS WRITING

/KIWI-BENCH READWRITE 500000 2 50

RANDOM-MULTI-THREAD-READWRITE (DONE:500000): 0.000060 SEC/OP;16666.7 WRITES&READS/SEC(ESTIMATED); COST:30.000(SEC); THREADS: 2

RANDOM-MULTI-THREAD-WRITE (DONE:500000, ADDED:500000): 0.000030 SEC/OP;33333.3 WRITES/SEC(ESTIMATED); COST:15.000(SEC); THREADS: 1

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000030 SEC/OP;33333.3 WRITES/SEC(ESTIMATED); COST:15.000(SEC); THREADS: 1

70% OF THREADS WRITING

/KIWI-BENCH READWRITE 500000 4 70

RANDOM-MULTI-THREAD-READWRITE (DONE:500000): 0.000056 SEC/OP;17857.1 WRITES&READS/SEC(ESTIMATED); COST:28.000(SEC); THREADS: 4

RANDOM-MULTI-THREAD-WRITE (DONE:500000, ADDED:500000): 0.000024 SEC/OP;41666.7 WRITES/SEC(ESTIMATED); COST:12.000(SEC); THREADS: 3

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000032 SEC/OP;31250.0 WRITES/SEC(ESTIMATED); COST:16.000(SEC); THREADS: 1

/KIWI-BENCH READWRITE 500000 8 70

RANDOM-MULTI-THREAD-READWRITE (DONE:500000): 0.000054 SEC/OP;18518.5 WRITES&READS/SEC(ESTIMATED); COST:27.000(SEC); THREADS: 8

RANDOM-MULTI-THREAD-WRITE (DONE:500000, ADDED:500000): 0.000020 SEC/OP;50000.0 WRITES/SEC(ESTIMATED); COST:10.000(SEC); THREADS: 6

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000034 SEC/OP;29411.8 WRITES/SEC(ESTIMATED); COST:17.000(SEC); THREADS: 2

/KIWI-BENCH READWRITE 500000 16 70

RANDOM-MULTI-THREAD-READWRITE (DONE:500000): 0.000052 SEC/OP;19230.8 WRITES&READS/SEC(ESTIMATED); COST:26.000(SEC); THREADS: 16

RANDOM-MULTI-THREAD-WRITE (DONE:500000, ADDED:500000): 0.000018 SEC/OP;55555.6 WRITES/SEC(ESTIMATED); COST:9.000(SEC); THREADS: 12

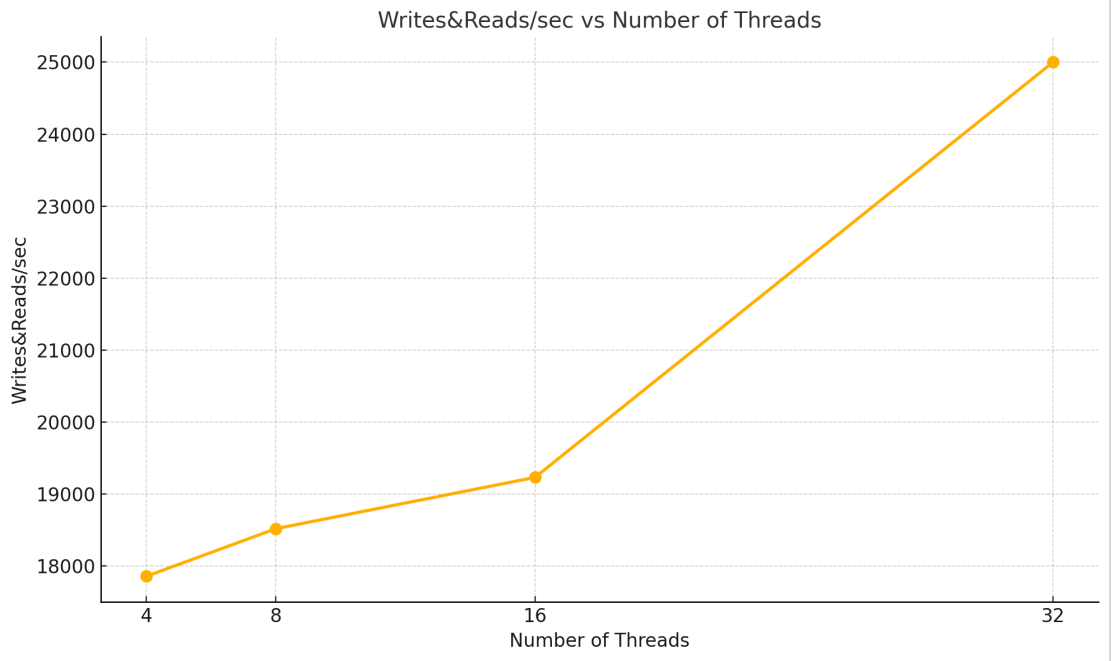
RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000034 SEC/OP;29411.8 WRITES/SEC(ESTIMATED); COST:17.000(SEC); THREADS: 4

/KIWI-BENCH READWRITE 500000 32 70

RANDOM-MULTI-THREAD-READWRITE (DONE:500000): 0.000040 SEC/OP;25000.0 WRITES&READS/SEC(ESTIMATED); COST:20.000(SEC); THREADS: 32

RANDOM-MULTI-THREAD-WRITE (DONE:500000, ADDED:500000): 0.000016 SEC/OP;62500.0 WRITES/SEC(ESTIMATED); COST:8.000(SEC); THREADS: 23

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000024 SEC/OP;41666.7 WRITES/SEC(ESTIMATED); COST:12.000(SEC); THREADS: 9



30% OF THREADS WRITING

/KIWI-BENCH READWRITE 500000 4 30

RANDOM-MULTI-THREAD-READWRITE (DONE:500000): 0.000060 SEC/OP;16666.7 WRITES&READS/SEC(ESTIMATED); COST:30.000(SEC); THREADS: 4

RANDOM-MULTI-THREAD-WRITE (DONE:500000, ADDED:500000): 0.000030 SEC/OP;33333.3 WRITES/SEC(ESTIMATED); COST:15.000(SEC); THREADS: 3

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000030 SEC/OP;33333.3 WRITES/SEC(ESTIMATED); COST:15.000(SEC); THREADS: 1

/KIWI-BENCH READWRITE 500000 8 30

RANDOM-MULTI-THREAD-READWRITE (DONE:500000): 0.000060 SEC/OP;17857.1 WRITES&READS/SEC(ESTIMATED); COST:28.000(SEC); THREADS: 8

RANDOM-MULTI-THREAD-WRITE (DONE:500000, ADDED:500000): 0.000030 SEC/OP;33333.3 WRITES/SEC(ESTIMATED); COST:15.000(SEC); THREADS: 3

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000030 SEC/OP;33333.3 WRITES/SEC(ESTIMATED); COST:15.000(SEC); THREADS: 5

/KIWI-BENCH READWRITE 500000 16 30

RANDOM-MULTI-THREAD-READWRITE (DONE:500000): 0.000052 SEC/OP;19230.8 WRITES&READS/SEC(ESTIMATED); COST:26.000(SEC); THREADS: 16

RANDOM-MULTI-THREAD-WRITE (DONE:500000, ADDED:500000): 0.000026 SEC/OP;38461.5 WRITES/SEC(ESTIMATED); COST:13.000(SEC); THREADS: 5

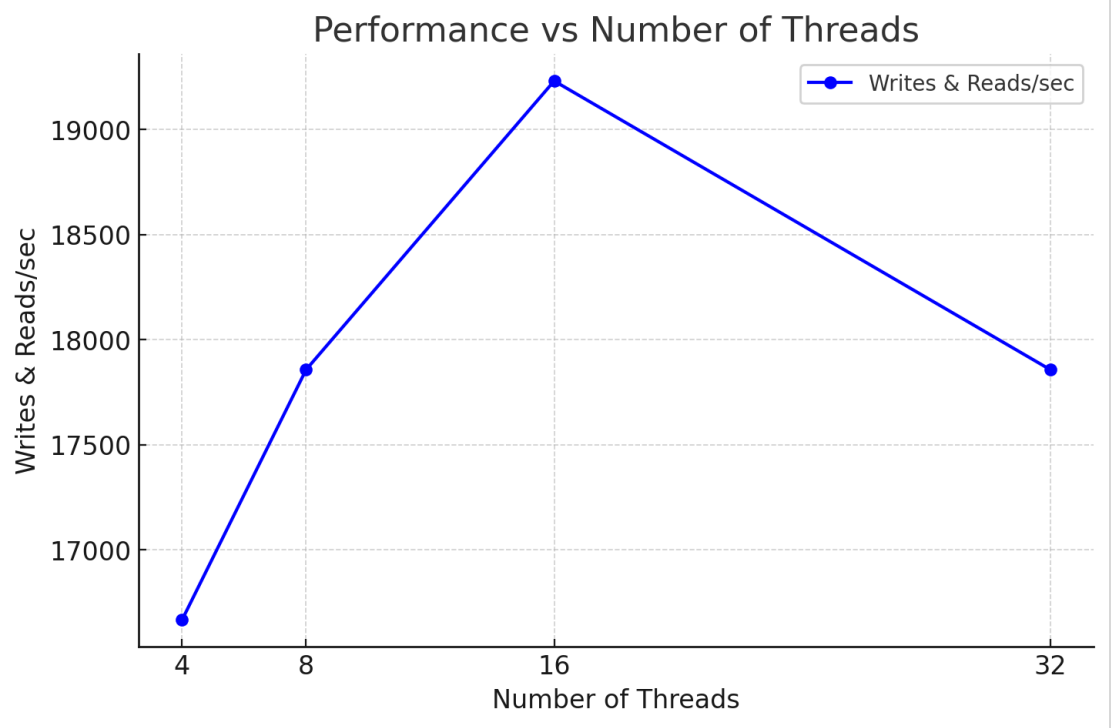
RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000026 SEC/OP;38461.5 WRITES/SEC(ESTIMATED); COST:13.000(SEC); THREADS: 11

/KIWI-BENCH READWRITE 500000 32 30

RANDOM-MULTI-THREAD-READWRITE (DONE:500000): 0.000056 SEC/OP;17857.1 WRITES&READS/SEC(ESTIMATED); COST:28.000(SEC); THREADS: 32

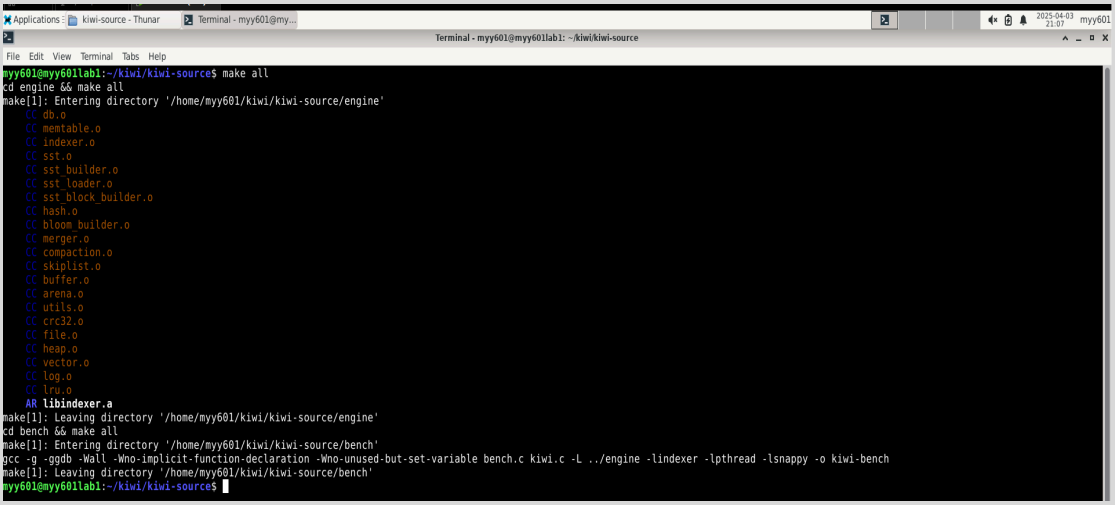
RANDOM-MULTI-THREAD-WRITE (DONE:500000, ADDED:500000): 0.000028 SEC/OP;35714.3 WRITES/SEC(ESTIMATED); COST:14.000(SEC); THREADS: 10

RANDOM-MULTI-THREAD-READ (DONE:500000, FOUND:500000): 0.000028 SEC/OP;35714.3 WRITES/SEC(ESTIMATED); COST:14.000(SEC); THREADS: 22



Φαίνεται ότι με την αύξηση του αριθμού των νημάτων, η ταχύτητα εκτέλεσης των λειτουργιών αυξάνεται γενικά. Αυτό οφείλεται στο γεγονός ότι περισσότερες διεργασίες εκτελούνται ταυτόχρονα, ενώ οι εγγραφές διαχειρίζονται με μονοπάτι. Για τη μέτρηση αυτής της επίδοσης, χρησιμοποιούμε τις λειτουργίες ανά δευτερόλεπτο, καθώς αυτές παρουσιάζουν τις πιο εμφανείς διαφορές στα γραφήματα.

ΕΞΟΔΟΣ MAKE


Applications
kiwi-source - Thunar
Terminal - myy601@my...
2025-04-03
21:07
myy601
Terminal - myy601@myy601lab1: ~/kiwi/kiwi-source
File Edit View Terminal Tabs Help
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 -lsnappy -o kiwi-bench
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/bench'
myy601@myy601lab1:~/kiwi/kiwi-source$