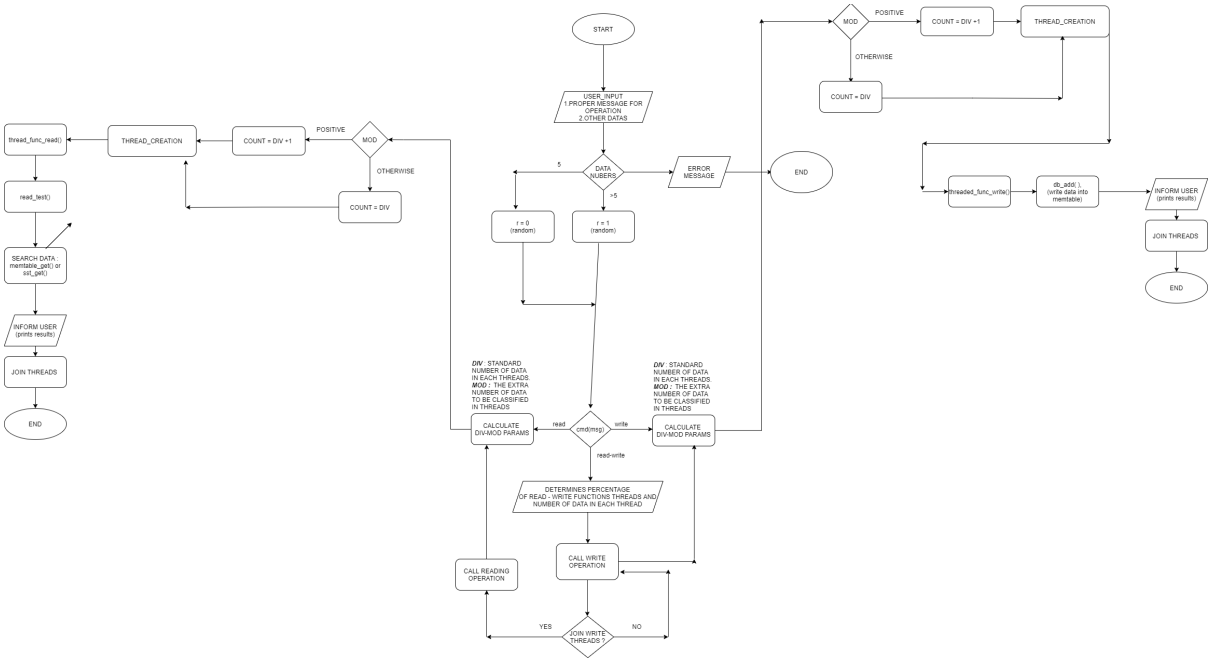
Περιεχόμενα……

1. Περίληψη Εργασίας

Σαν πρώτη εργαστηριακή άσκηση, θα σας παρουσιάσουμε τη δική μας λύση πάνω στη μηχανή αποθήκευσης δεδομένων Kiwi που μας δώσατε. Οι τροποποιήσεις που εφαρμόσαμε είχαν σκοπό μία πολυνηματική υλοποίηση των λειτουργιών put και get για ζεύγη κλειδιού-τιμής. Επιπλέον, προσθέσαμε και τη λειτουργία “readwrite” που υλοποιεί ταυτόχρονα put και get λειτουργίες. Μελετήσαμε και κατανοήσαμε τα μονοπάτια που ακολουθούν οι λειτουργίες put και get και υλοποιήσαμε τα πολλαπλά νήματα με κατάλληλη επέκταση του bench.c. Έπειτα, διαπιστώσαμε ακριβώς αυτό που αναφέρετε και εσείς στην εκφώνηση για το segmentation fault. Αυτό οφειλόταν στο γεγονός ότι κάποιο νήμα είχε πρόσβαση σε χώρο της μνήμης που δεν του αναλογούσε. Για την επίλυση αυτού του προβλήματος, χρησιμοποιήθηκαν κατάλληλα κλειδαριές σε διάφορα μέρη του κώδικα. Στη συνέχεια, το επίπεδο ταυτοχρονισμού που προσπαθήσαμε να πετύχουμε είναι αυτό του ενός γραφέα και πολλαπλών αναγνωστών. Για αυτό το λόγο έχουμε προσθέσει επιπλέον ταυτοχρονισμό σε διάφορα επίπεδα του LSM-tree. Κατόπιν, επεκτείναμε τα ορίσματα της γραμμής εντολών προσθέτοντας το συνολικό αριθμό των νημάτων και το ποσοστό από τον κάθε τύπο λειτουργίας για την περίπτωση της λειτουργίας “readwrite”. Τέλος, προσθέσαμε και το συνολικό χρόνο απόκρισης των λειτουργιών put και get.

Παρακάτω ακολουθεί ένα περιληπτικό διάγραμμα των λειτουργιών της μηχανής αποθήκευσης δεδομένων Kiwi που υλοποιήσαμε.



2. Επεξήγηση Κώδικα

Η πρώτη μας σκέψη ήταν να δημιουργήσουμε ένα struct έτσι ώστε να έχουμε άμεση πρόσβαση στα δεδομένα των νημάτων (threads). Αυτό έγινε στο αρχείο ./kiwi-source/bench/bench.c ως έχει στη σειρά 137, όπου count ο αριθμός των λειτουργιών για κάθε νήμα και r η επιλογή των κλειδιών random ή όχι:

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

Για την δημιουργία των νημάτων, αρχικά, ορίσαμε ένα επιπλέον όρισμα στη γραμμή εντολών που μας παρέχει το συνολικό αριθμό νημάτων (**NUMTHRDS = atoi(argv[3]);**) που θέλουμε η κάθε λειτουργία να χρησιμοποιήσει (read, write ή readwrite). Παράδειγμα: ./kiwi-bench write 100000 **100**. Επιπλέον, για κάθε διαφορετική λειτουργία χρησιμοποιούμε τη μεταβλητή **divv** για τον υπολογισμό του βασικού αριθμού λειτουργιών που θα εκτελέσει κάθε νήμα. Με τη χρήση της μεταβλητής **mod** υπολογίζουμε τον αριθμό των πρόσθετων λειτουργιών που υπολείπονται από μία μη ακέραια διαίρεση της divv. Αυτές οι επιπλέον λειτουργίες κατανέμονται δίκαια στα νήματα. Για παράδειγμα στην περίπτωση που μας υπολείπονται δύο λειτουργίες (mod=2) τότε επιβαρύνουμε δύο νήματα κατά μία λειτουργία, αντί να

επιβαρύνω ένα νήμα κατά δύο.

Στη συνέχεια, επιλέξαμε να ανοίξουμε και να κλείσουμε τη βάση δεδομένων μέσα στη **main** στο αρχείο **bench.c** και όχι κάθε φορά που εκτελούνται οι _write_test και _read_test έτσι ώστε τα νήματα να έχουν άμεση πρόσβαση στη βάση δεδομένων και να μην ανοιγοκλείνει τη βάση ένα νήμα κάθε φορά. Για αυτό το λόγο καλούμε μέσα στη main πριν τη δημιουργία των νημάτων τη συνάρτηση **db_control()** που υλοποιείται στο αρχείο kiwi.c. Η συνάρτηση αυτή παίρνει ως όρισμα ένα ακέραιο αριθμό, πιο συγκεκριμένα όταν δέχεται ως όρισμα το "0" ανοίγει τη βάση δεδομένων, ενώ σε οποιοδήποτε άλλο ακέραιο αριθμό την κλείνει (π.χ. σειρά 242: db_control(1);). Μετά το άνοιγμα της βάσης ακολουθεί η δημιουργία των νημάτων μέσα σε for loop με όριο τον συνολικό αριθμό νημάτων που έδωσε ο χρήστης . Στην περίπτωση που έχουμε mod θετικό ακέραιο προσθέτουμε μία επιπλέον λειτουργία στο νήμα που θα δημιουργηθεί. Έτσι περνάμε τις κατάλληλες τιμές στα structure parameters (count,r), όπου αυτά αναφέρονται στα δεδομένα των νημάτων. Έπειτα μειώνουμε κατά μία τιμή το mod (μιας και η ανάλογη επιβάρυνση έχει περάσει στα δεδομένα του νήματος) έως ότου φτάσει στην επιθυμητή τιμή που είναι το 0 (mod=0) με σκοπό να δημιουργήσουμε νήματα τα οποία δεν θα επιβαρυνθούν (δηλαδή το else). Τέλος δημιουργούμε το νήμα κάνοντας χρήση της εντολής **pthread_create** που μας δίνεται από την βιβλιοθήκη **<pthread.h>,** την οποία την κάνουμε include στην αρχή του bench.c. Συγκεκριμένα η pthread_create παίρνει 4 ορίσματα. Το πρώτο όρισμα είναι το όνομα του νήματος που δημιουργείται εκείνη την στιγμή, το οποίο έχει αρχικοποιηθεί στην αρχή της main (**pthread_t threadID[NUMTHRDS]**). Το δεύτερο όρισμα είναι η τιμή NULL, το τρίτο είναι η **threaded_func_write** για την λειτουργία put και **threaded_func_read** για την λειτουργία get αντίστοιχα και το τελευταίο όρισμα είναι το **&data_threads.** Μετά την for loop ακολουθεί ένα νέο for loop όπου εκεί εκτελείται η εντολή **pthread_join** για να μπλοκάρουμε την main έως ότου εκτελεστούν τα νήματα. Κατόπιν κλείνουμε την βάση με την χρήση της **db_control(1)** όπως αναφέραμε και πριν. Τέλος εκτυπώνουμε τα στατιστικά για την συγκεκριμένη λειτουργία καλώντας τις **write_print_stats** και **read_print_stats** για τις λειτουργίες put και get αντίστοιχα. Και για τις δύο λειτουργίες put και get χρησιμοποιήθηκε η ίδια τεχνικη και λογική.

Προηγουμένως αναφερθήκαμε σε δύο συναρτήσεις, τις threaded_func_write και threaded_func_read, που αποτελούν όρισμα για τη

δημιουργία νημάτων στην εντολή pthread_create. Αυτές αποτελούν βοηθητικές συναρτήσεις και δημιουργήθηκαν με σκοπό την κλήση των συναρτήσεων **_write_test** και **_read_test** αντίστοιχα. Πριν την κλήση τους χρησιμοποιούμε mutex (τα write_lock και read_lock στις συναρτήσεις threaded_func_write και **threaded_func_read** αντοίστοιχα) με σκοπό το κλείδωμα για την αποφυγή προσπέλασης πολλαπλών νημάτων που μεταφέρουν διαφορετικά δεδομένα. Η αρχικοποίησή τους γίνεται global στο αρχείο bench.c.

Επιπρόσθετα, για την επίτευξη ταυτοχρονισμού ενός γραφέα και πολλαπλών αναγνωστών χρησιμοποιήσαμε κλειδαριές (mutexes) γύρω από κρίσιμες περιοχές και σε άλλα σημεία του κώδικα. Συγκεκριμένα, στο αρχείο: ./kiwi-source/engine/db.c τοποθετήσαμε κλειδαριές στη συνάρτηση db_add (pthread_mutex_lock(&self->write_lock);) και db_get (pthread_mutex_lock(&self->read_lock);) τα οποία mutexes αρχικοποιήθηκαν στο αρχείο db.h. Στο αρχείο ./kiwi-source/engine/memtable.c χρησιμοποιήθηκαν κλειδαριές στις συναρτήσεις memtable_add (pthread_mutex_lock(&self->mem_lock);), memtable_get (pthread_mutex_lock(&list->skiplist_lock);) και memtable_edit (pthread_mutex_lock(&self->mem_lock);). Η αρχικοποίηση του κλειδιού mem_lock γίνεται στο αρχείο memtable.h ενώ του κλειδιού skiplist_lock στη skiplist.h. Στο αρχείο ./kiwi-source/bench/kiwi.c χρησιμοποιήθηκαν κλειδαριές, ιδιαιτέρως για την εγγραφή του κόστους (write_cost_lock και read_cost_lock) τα οποία αρχικοποιούνται global στην αρχή του kiwi.c.

Για την απεικόνιση των χρόνων των εκάστοτε λειτουργιών χρησιμοποιήσαμε βοηθητικές συναρτήσεις που υλοποιήσαμε στο αρχείο bench.c (write_print_stats, read_print_stats, readwrite_print_stats). Οι τιμές τους αποθηκεύονται σε μεταβλητές μέσα στο αρχείο kiwi.c. Συγκεκριμένα για τη λειτουργία put αποθηκεύεται η τιμή write_cost μέσα στη συνάρτηση _write_test, η τιμή read_cost μέσα στη συνάρτηση _read_test, ενώ η τιμή bfound επιστρέφει το πλήθος κλειδιών που βρέθηκαν. Οι αρχικοποιήσεις όλων τους γίνονται μέσα στο bench.h.

Για το συγχρονισμό των λειτουργιών put και get (read-write) υλοποιήσαμε μία νέα λειτουργία που ονομάσαμε “readwrite”. Για αυτήν προσθέσαμε ένα επιπλέον όρισμα στη γραμμή εντολών που αναφέρει ένα ποσοστό για να προσδιορίσουμε το μίγμα λειτουργιών put και get και το

ποσοστό από τον κάθε τύπο (π.χ ./kiwi-bench write 100000 100 **60-40**.). Πιο συγκεκριμένα, η μεταβλητή read_per παίρνει το ποσοστό που αναγράφεται αριστερά της "-" . Η μεταβλητή read_count_per υπολογίζει το ποσοστό των λειτουργιών get, ενώ η write_count_per των put. Η μεταβλητή write_NUMTHRDS υπολογίζει τον αριθμό των νημάτων που θα χρειαστούν για την εκτέλεση get. Επίσης δημιουργήθηκαν και νέες μεταβλητές div_write, div_read, mod_write και mod_read που αποσκοπούν στο ίδιο ακριβώς που αναφέραμε και πριν για τις μεταβλητές div και mod. Πάλι χρησιμοποιούμε δύο for loop ένα για τη λειτουργία put και ένα για τη λειτουργία get όπου το for της put έχει ως άνω όριο την τιμή του write_NUMTHRDS, ενώ το for της get το έχει ως κάτω όριο και ως άνω το συνολικό NUMTHRDS. Η υπόλοιπη διαδικασία που ακολουθείται μέσα σε αυτά τα for είναι ίδιας λογικής και κατασκευής που ακολουθήσαμε στις for των put και get όπως αναφέραμε και παραπάνω. Τέλος, ακολουθεί μία for loop όπου καλούμε pthread_join έτσι ώστε η main να περιμένει να εκτελεστούν όλα τα νήματα για να συνεχίσει την εκτέλεσή της. Μετά από αυτό εκτυπώνονται οι αποδόσεις των put και get μέσω της συνάρτησης readwrite_print_stats(). Αυτή παίρνει σαν ορίσματα τα write_cost, div_write, read_cost, div_read,και bfound όπως αναφέραμε.

3.Παρατηρήσεις

Παρατηρήσαμε ένα bug, που έχει να κάνει με το πέρασμα έξτρα δεδομένων που μεταφέρει κάθε νήμα. Συγκεκριμένα, αφού εκτελέσουμε την πράξη **divv = count/NUMTHRDS;** και εισερχόμαστε μέσα στο for loop στην περίπτωση όπου το mod είναι θετικό, εκτελείται η εντολή **data_threads.count = divv +1;** . Ενώ η τιμή αποθηκεύεται σωστά στο count της data_threads (το ελέγξαμε με εκτυπώσεις μέσα στο if) η τιμή αυτή δεν μεταφέρεται στη βοηθητική συνάρτηση που παίρνει σαν τρίτο όρισμα η pthread_create. Αντ' αυτού η τιμή που παίρνει είναι πάντα η τιμή του divv. Είναι δηλαδή σα να γίνεται η πράξη χωρίς την πρόσθεση της μονάδας. Προσπαθήσαμε να κάνουμε την πράξη πριν το for loop και να περάσουμε την τιμή σε μία νέα μεταβλητή και μετά να την περάσουμε στο data_threads.count αλλά με το ίδιο αποτέλεσμα. Αυτό έχει ως αποτέλεσμα σε περίπτωση μη ακέραιας διαίρεσης και ύπαρξης θετικού αριθμού mod τα νήματα δεν παίρνουν αυτές τις έξτρα λειτουργίες. Για παράδειγμα, στην περίπτωση όπου οι λειτουργίες είναι 21 και τα νήματα είναι 5

τότε 1 νήμα θα έπαιρνε 4+1 λειτουργίες και τα υπόλοιπα από 4 λειτουργίες. Όμως, εξαιτίας αυτού του bug ή λάθους θα έχουμε 5 νήματα με 4 λειτουργίες και αυτή την πρόσθετη λειτουργία δεν θα την πάρει κανένα. Τελικά καταλάβαμε ότι το λάθος ήταν η δήλωση του struct data_threads στην αρχή της main, καθώς η σωστή λύση ήταν να την δηλώνουμε κάθε φορά ξεχωριστά μέσα στη for loop. Μία φορά για την περίπτωση που το mod είναι θετικό και μία φορα για την περίπτωση που δεν είναι για κάθε λειτουργία put,get και put-get ξεχωριστά.

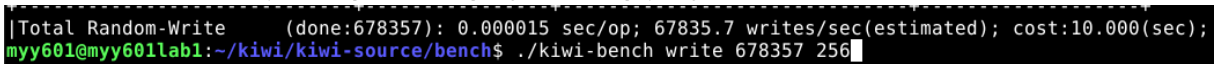
Επιπλέον παρατηρήσαμε ότι η εναλλαγή adding-searching είναι πιο ευδιάκριτη όταν θα τρέξετε την λειτουργία read-write για αρκετά μεγάλο αριθμό count (πχ. Για 500.000) αφού πρώτα κάνετε make clean,έπειτα rm -rf testdb και στη συνέχεια κάνοντας make all.

4. Αποτελέσματα αποδόσεων και εντολές

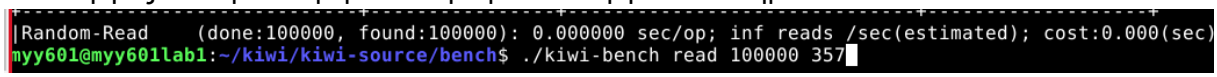
Παρακάτω παραθέτουμε τις διαφορετικές εντολές για τις διαφορετικές λειτουργίες που χρησιμοποιήσαμε για να τρέξουμε το kiwi-bench

Για παράδειγμα για 100 χιλιάδες λειτουργίες write 256 νήματα έχουμε το παρακάτω αποτέλεσμα. Δεν συμπεριλάβαμε όλα τα print των νημάτων γιατί όπως καταλαβαίνετε ήταν πάρα πολλά.

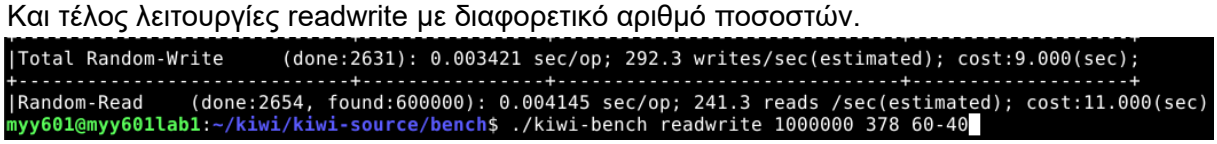
Πάλι write αλλά για περισσότερες λειτουργίες.



Λειτουργίες read με διαφορετικό αριθμο λειτουργιών και νημάτων.



Και τέλος λειτουργίες readwrite με διαφορετικό αριθμό ποσοστών.



Ευχαριστούμε πολυ!!!