Πανεπιστήμιο Ιωαννίνων

Πολυτεχνική Σχολή

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

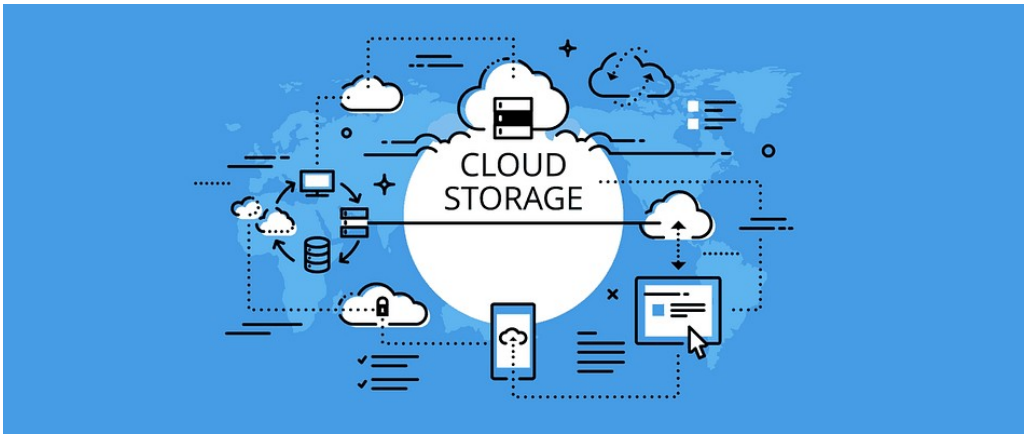
Προπτυχιακό Μάθημα: «Λειτουργικά Συστήματα»

1η Εργαστηριακή Άσκηση

Μέλη Ομάδας - Α.Μ.:

Παναγιώτης Ντουνέτας - 2781

Γεώργιος Κρομμύδας – 3260



ΙΩΑΝΝΙΝΑ, 2021

Αναφορά 1ης Εργαστηριακής Άσκησης

(Kiwi με μηχανή αποθήκευσης βασισμένη στο δέντρο LSM)

(Υλοποίηση πολυνηματικής λειτουργίας σε μηχανή αποθήκευσης δεδομένων)

Μέρος Α: Ιδέα και ζητούμενα

Η γενική ιδέα της άσκησης είναι να τροποποιηθούν αρχεία της επιλογής μας για την επίτευξη λειτουργιών, που επεκτείνουν τις δυνατότητες write και read (put και get) του Kiwi. Αυτό βέβαια θα γίνει με τη χρήση πολλαπλών νημάτων. Επιπλέον, προστίθεται η λειτουργία “readwrite” η οποία όχι μόνο κάνει αξιοποίηση των νημάτων, αλλά δέχεται και ποσοστιαίο διαμοιρασμό των λειτουργιών.

Έχουμε σαν δεδομένο ότι ο αμοιβαίος αποκλεισμός με μία καθολική κλειδαριά στο σύστημα δεν είναι η βέλτιστη λύση, καθώς επιφέρει πρόβλημα ότι μόνο ένας γραφέας ή ένας αναγνώστης λειτουργεί κάθε φορά. Το επόμενο βήμα με την επίτρεψη πολλαπλών αναγνωστών και ενός γραφέα αποτελεί σίγουρα πολύ καλύτερη λύση , αλλά και πάλι όχι βέλτιστη. Τέλος , αν θέλουμε να ικανοποιήσουμε και το 3ο βήμα υλοποίησης , σημαίνει ότι πρέπει να επέμβουμε σε ακόμα μεγαλύτερο βάθος στα αρχεία του Kiwi, πράγμα που το καθιστά ακόμα πιο καλή αλλά ταυτόχρονα δύσκολη λύση.

Ζητούνται, λοιπόν, τρόποι επέκτασης της ήδη υπάρχουσας υλοποίησης η οποία σε αρχικό στάδιο εκτελεί μόνο απλά read και write ώστε να εκτελείται πολυνηματισμός, καθώς και μετρήσεις κόστους στις λειτουργίες αυτές.

Μέρος Β: Ανάλυση ερωτημάτων – Τρόπος σκέψης – Πορεία δράσης

Ερώτημα Α:

Οι λειτουργίες put (write) και get (read) βρίσκονται στο αρχείο kiwi.c όπου αναπτύσσεται ο κώδικας τους και που ύστερα θα επέμβουμε για την επέκταση της λύσης. Οι συναρτήσεις αυτές μετά, καλούνται στη main συνάρτηση του αρχείου bench.c, που βρίσκεται στο φάκελο bench όπως και το kiwi.c, που θα γίνει η επιλογή για το ποια λειτουργία θα εκτελεστεί (και συνεπώς η προσθήκη της παραπάνω ζητούμενης συνάρτησης "readwrite"). Παρατηρούμε ότι οι αναφερόμενες συναρτήσεις db_add και db_get αρχικοποιούνται στο αρχείο db.h και υλοποιούνται στο db.c του directory "engine". Το documentation του Kiwi φάνηκε πάρα πολύ χρήσιμο ώστε να κατανοήσουμε το τρόπο συσχέτισης των συναρτήσεων μεταξύ τους , καθιστώντας πολύ πιο εύκολη τη διαδικασία εύρεσης της σωστής τακτικής, καθώς και των παραπάνω συσχετίσεων μεταξύ των αρχείων.

Το default benchmark του Kiwi ήταν αρκετά απλό ως προς τον τρόπο εκτέλεσης. Αν κάναμε write με πάρα πολλές εγγραφές, παρατηρούσαμε σε μεγαλύτερο βαθμό τον τρόπο λειτουργίας μέσα από τα μηνύματα στης οθόνης, όπως την διαδικασία ανοίγματος της βάσης, την εγγραφή στοιχείων στο αρχείο log και ο διαχωρισμός από το level 0 έως όποιο level γινόταν η εγγραφή ανάλογα με το μέγεθος των εισαγόμενων στοιχείων κ.ο.κ. Οι εκτελέσεις της read , γίνονταν κανονικά είτε είχε γίνει πριν write είτε όχι, το οποίο δεν ήταν πρόβλημα καθώς ήταν απλά για την παρατήρηση του τρόπου εκτέλεσης.

Ερώτημα Β:

Το αρχείο bench.c όπως αναφέραμε, περιέχει την main function που εκτελεί τον κώδικά μας. Η προσθήκη ορίσματος στην γραμμή εντολών για την εισαγωγή του επιθυμητού αριθμού νημάτων είναι μια απλή σκέψη. Αφήσαμε το "r" να παραμένει το τελευταίο όρισμα στη γραμμή (διορθώνοντας βέβαια τους κατάλληλους ελέγχους). Η απόδοση τιμής σε μεταβλητή γίνεται με τον παρακάτω τρόπο:

```
int numot = atoi(argv[3]);  //Number Of Threads - Syntomia "numot" pou tha dinontai san orisma.
```

όπου argv[3], το 4ο όρισμα πχ. στην εντολή ./kiwi-bench write 100 **5**.

Η μεταβλητή δηλώνεται μία φορά, πριν την επιλογή μίας εκ των 3 λειτουργιών read/write/readwrite. Επίσης, απευθείας την μετατρέπουμε από τύπο char σε integer ώστε να είναι έτοιμη για χρήση από το πρόγραμμα. Τροποποιούμε την συνθήκη που ελέγχει την λανθασμένη είσοδο τιμών:

```
if (argc < 4 && ((strcmp(argv[1], "write") == 0)|| (strcmp(argv[1], "read") == 0))){
    fprintf(stderr,"Usage: db-bench <write / read / readwrite> <count> <num_of_threads> <r> \n");
    exit(1);
}
```

(Θα εξηγηθεί και παρακάτω γιατί η συνθήκη έγινε και πιο περίπλοκη)

Ερώτημα Γ:

Το θέμα της επίτευξης ταυτοχρονισμού δεν είναι απλό ζήτημα. Χρειάστηκε αρκετή δουλειά και ψάξιμο ώστε να φτάσουμε πολύ κοντά στην ολοκλήρωση του 2ου βήματος (πολλοί αναγνώστες vs 1 γραφέας). Αρχικά, το 1ο βήμα, που αφορούσε την πιο απλοϊκή υλοποίηση, δεν ήταν παρά μόνο μια καθολική κλειδαριά. Για παράδειγμα , τοποθετήσαμε μέσα στο αρχείο db.h μια απλή μεταβλητή τύπου pthread_mutex_t, την lockDB. Άρα η pthread_mutex_t lockDB καλείται στην αρχή και το τέλος σε κάθε μία απο τις db_add και db_get αντίστοιχα σε locks και unlocks. Αυτό μαρκάρει σαν κρίσιμες περιοχές όλο το περιεχόμενο των db_add και db_get. Όμως, αυτή η ενέργεια ΔΕΝ μας επιτρέπει ταυτοχρονισμό διότι κλειδώνει σε έναν γραφέα ή έναν αναγνώστη κάθε φορά.

Όσον αφορά το 2ο βήμα, (θα εξηγηθεί πολύ πιο αναλυτικά παρακάτω) απαιτούσε αρκετές τροποποιήσεις και προσθήκες. Σύμφωνα με την κρίση μας, τοποθετήσαμε locks και unlocks εντός του αρχείου kiwi.c για μια μεταβλητή που θα μας βοηθήσει στον υπολογισμό κόστους. Επίσης, το ίδιο κάναμε για μεταβλητές τύπου pthread_mutex αλλά και pthread_cond , όπως είχαμε ενθαρρυνθεί από τον διδάσκοντα , σαν μία παραλλαγή του παραδείγματος των αναγνωστών – γραφέων με τις σημαφόρους. Αυτό το αναφέρουμε, διότι υλοποιήσαμε μια παραλλαγή με μεταβλητές συνθήκης, όπως και είχε ζητηθεί.

Όσο για το 3ο βήμα, θεωρούμε πως θα ήταν μια ακόμα επέκταση του 2ου βήματος απλά σε αρχεία που πηγαίνουν πιο βαθιά στη δομή του κώδικα. Περισσότερες κρίσιμες περιοχές σε σωστά σημεία , σημαίνει και περισσότερη λειτουργικότητα.

Ερώτημα Δ:

Εδώ τα πράγματα είναι λιγότερο σύνθετα από από το προηγούμενο ερώτημα. Όπως και στο ερώτημα Α, επεκτείνουμε τη γραμμή εντολών αποθηκεύοντας σε μια μεταβλητή το ποσοστό για κάθε λειτουργια.

Πχ. η εντολή:

./kiwi-bench readwrite 100000 20 60 40

είναι η λειτουργία readwrite, με 100.000 requests, 20 νήματα, 60% puts και 40% gets. Αυτό συμβαίνει εδω:

```
double wper = atoi(argv[4]);
wper = wper / 100;
double rper = atoi(argv[5]);
rper = rper / 100;
```

όπου τελικά τα wper, rper κρατάνε το ποσοστό και θα διαχωρίσουν τις λειτουργίες.

Επίσης, επεκτάθηκε η συνθήκη για τον εντοπισμό λάθος εισόδου:

if (argc < 6 && (strcmp(argv[1], "readwrite") == 0)){

fprintf(stderr,"Usage: db-bench <write / read / readwrite> <count> <num_of_threads> <write_percentage> <read_percentage> <r>\n");

exit(1);

}

Εδώ εξηγείται γιατί και η συνθήκη εισόδου για λιγότερα ορίσματα στο ερώτημα ήταν πιο περίπλοκη. Εδώ έχουμε 6 βασικά ορίσματα με 7ο το ‘r’.

Ερώτημα Ε:

Η διατήρηση των στατιστικών απόδοσης γίνεται ως εξής: Με δύο νέες μεταβλητές εφαρμόζουμε mutex_lock και mutex_unlock στο τέλος κάθε μίας από τις _write_test και _read_test. Οι συναρτήσεις αυτές διέθεταν έναν τρόπο υπολογισμού χρόνου με την αφαίρεση των μεταβλητών start και end. Οπότε αξιοποιούμε την μέτρηση αυτή αθροίζοντας το αποτέλεσμά της κάθε φορά σε ένα γενικό σύνολο της λειτουργίας. Το σύνολο αυτό ορίζεται σαν κρίσιμη περιοχή , έτσι ώστε να υπάρχει συγχρονισμός μεταξύ διαφορετικών νημάτων. Η εκτύπωσή τους γίνεται στην main της bench.c με τη χρήση βοηθητικής συνάρτησης, που υπολογίζει διάφορες αποδόσεις.

Μέρος Γ: Περιγραφή του παραδιδόμενου κώδικα.

Η περιγραφή του κώδικα θα γίνει ανά αρχείο για να ακολουθήσουμε μια σειρά ως προς την επεξήγηση.

Bench header file (bench.h):

Αρχικά προσθέσαμε το #include <pthread.h> . Εφόσον το bench.h αρχείο γίνεται include και στο bench.c και kiwi.c, όπου μέσα εκεί γίνονται αρχικοποιήσεις και επεξεργασία μεταβλητών που αφορά τα νήματα, δεν χρειάζεται να κληθεί ξανά ξεχωριστά.

Επίσης προσθέσαμε τις παρακάτω γραμμές:

```
struct holder
{
    int count_holder;
    int thread_holder;
    int r_holder;
};

pthread_mutex_t numwrites;
pthread_mutex_t numreads;
double writescost;
double readscost;
```

Το struct holder , είναι μία βοηθητική δομή που θα κρατάει τα βασικά στοιχεία της κάθε κλήσης από της γραμμή εντολών έτσι ώστε να είμαστε ευέλικτοι στη χρήση τους αργότερα σε συναρτήσεις και στον υπόλοιπο κώδικα. Το count_holder αντιπροσωπεύει τον αριθμό request που ζητήθηκαν, thread_holder τα νήματα και το r_holder το r της εκτέλεσης εφόσον είδαμε ότι αποτελεί όρισμα στις κλήσεις των _write_test και _read_test αντίστοιχα.

Επιπλέον έχουμε τις pthread_mutex_t numwrites και numreads. Είναι οι μεταβλητές που θα μας χρειαστούν στον υπολογισμό του κόστους. Ο τύπος τους εξαρτάται από το γεγονός ότι θα χρησιμοποιηθούν ως οι μεταβλητές για το lock και unlock ώστε να μαρκάρουμε την κρίσιμη περιοχή για την καταγραφή του χρόνου. Οι μεταβλητές writecost και readcost είναι αυτές που θα διατηρούν την τιμή του χρόνου μέσα στην κρίσιμη περιοχή.

Bench.c αρχείο:

Το αρχείο αυτό είναι ίσως το πιο τροποποιημένο από όλα. Πάνω από τη main έχουμε ορίσει μερικές βοηθητικές συναρτήσεις:

```
///////WRITERS AND READERS CALLED /////////
void *request_handler_w(void *arg) // Genikh voithitiki synarthsh gia ton xeirismo twn zhtoumenwn write h readmesa sth main.
{
    struct holder* rh = (struct holder *)arg; //Arxikopoihsh mias metavliths gia thn diaxeirhsh tou struct px. rh apo to request handler
    _write_test(rh->count_holder, rh->r_holder, rh->thread_holder); //tote kalese thn _write_test apo to kiwi.c arxeio
    return 0;
}

void *request_handler_r(void *arg){
    struct holder* rh = (struct holder *)arg;
    _read_test(rh->count_holder, rh->r_holder, rh->thread_holder); //kalese thn _read_test
    return 0;
}
//////////////////////////////////////
```

Οι request_hander_w και request_handler_r (w για writer, r για reader) είναι βοηθητικές συναρτήσεις οι οποίες καλούν αποκλειστικά η μία λειτουργία read ή write η καθεμία. (*Γιατί να μην τις αφήσουμε ως είχαν στον αρχικό κώδικα?*) Με την προσθήκη του βοηθητικού struct holder θέλουμε να γράφουμε κάθε φορά τι κρατάει το struct σε εκείνη του τη μεταβλητή. Για την κάθε λειτουργία που θα κάνουμε θα πρέπει να δημιουργούμε νήματα. Δεν ήταν καθόλου βολικό να έχουμε μία τεράστια κλήση της write ή read κάθε φορά στη δημιουργία νήματος, οπότε καλούμε απλά τη συνάρτηση να κάνει τη δουλειά.

Η υλοποίησή τους είναι αρχικά η αρχικοποίηση μιας μεταβλητής τύπου που συμφωνεί με το struct μας. Έτσι , τροφοδοτούμε κάθε φόρα την απαραίτητη συνάρτηση με τα ορίσματα που έρχονται από το struct (και κατα συνέπεια πως το struct έχει οριστεί σε κάθε λειτουργία read/write/readwrite της main). (*Γιατί όχι σε μία συνάρτηση και να διαλέγετε ανάλογα αν θα κάνετε read ή write?*) Θα μπορούσαμε να τις βάλουμε σε μία συνάρτηση και απλά να διαλέγεται να θα γίνει read ή write test κλήση που θα ελέγχεται με ένα όρισμα πχ το argv[1] της γραμμής εντολών δηλαδή ποια λειτουργία θα γίνει στο πρόγραμμα. Το θέμα ήταν πως αυτό δημιουργούσε Floating exception error.

Επιπλέον, η συνάρτηση printer():

```
//// CALCULATE AND PRINT RESULTS//////////

void *printer(char* action, int count, double all){
    if(strcmp("write",action) == 0){
        printf("WRITERS statistics: \n Number of requests: %.d \n Req. service time: %3f \n Writes pe
    }
    if(strcmp("read",action) == 0){
        printf("READERS statistics: \n Number of requests: %.d \n Req. service time: %3f \n Reads per
    }
    return 0;
}
////////////////////////////////////////
```

Η συνάρτηση αυτή έχει 3 ορίσματα:

-char *action: Όρισμα που μας επιτρέπει να δούμε αν θα κάνουμε read ή write.

-int count: Το σύνολο των νημάτων της λειτουργίας.

-int all: Το συνολικό κόστος που έχει υπολογιστεί.

Αυτό που τυπώνεται κάθε φορά, είναι μια παραλλαγή των εντολών printf() που υπήρχαν στο αρχείο kiwi.c και πλέον τις έχουμε αφήσει σε σχόλια.

Η υλοποίηση της main έχει γίνει ως εξής:

Αρχικά:

```
int main(int argc,char** argv)
{
    long int count;
    int numot = atoi(argv[3]);  //Number Of Threads - Syntomia "num

    pthread_t *genthreads, *wthreads, *rthreads;

    genthreads = (pthread_t*)malloc(numot*sizeof(pthread_t)); //ger
    wthreads = (pthread_t*)malloc(numot*sizeof(pthread_t)); //Ayth
    rthreads = (pthread_t*)malloc(numot*sizeof(pthread_t)); //Ta ka

    struct holder general,writers,readers; //Tha kratame se kathe i

    pthread_mutex_init(&numwrites, NULL); //Arxikopoihsh
    pthread_mutex_init(&numreads, NULL);
    writescost = 0;
    readscost = 0;

    srand(time(NULL));
```

Είδαμε και παραπάνω την μεταβλητή int numot ότι χρησιμοποιείται για να κρατάει το πλήθος το νημάτων από τη γραμμή εντολών.

Μετά έχουμε τις μεταβλητές τύπου pthread που θα μας χρειαστούν για την αρχικοποίηση και δημιουργία των νημάτων. ‘Εχουμε 3 τέτοιες μεταβλητές:

- η genthreads θα χρησιμοποιηθεί από την σκέτη read ή write. Μία μεταβλητή είναι αρκετή διότι κάθε φορά θα γίνεται μία από αυτές τις λειτουργίες.

-wthreads και rtrheads: (write threads / read trheads) Μεταβλητές για τα νήματα για την λειτουργία readwrite. Εφόσον θέλουμε να γίνεται διαφοροποίηση μεταξύ των ταυτόχρονων λειτουργιών read και write, χρειαζόμαστε μια μεταβλητή για την καθεμία.

Οι παραπάνω μεταβλητές αρχικοποιούνται με malloc με μέγεθος ίσο με αυτό του πλήθους των νημάτων. (Ίσως οι wthreads και rthreads να είναι αχρείαστα στο συνολικό μέγεθος εφόσον θα πραγματοποιούν ποσοστιαίες αρχικοποιήσεις)

Οι μεταβλητες τύπου struct holder: Με αυτές θα περνάμε κάθε φορά στο struct (που είδαμε στο bench.h) τις τιμές των request, νηματων και r. Έχουν ονόματα τα οποία μοιάζουν πολύ με τις παραπάνω μεταβλητές για τα νήματα επίτηδες.

Τέλος, για την παραπάνω εικόνα , οι αρχικοποιήσεις των τεσσάρων μεταβλητών του bench.h που προσθέσαμε για τον υπολογισμό του κόστους.

Ας δούμε τις βασικές λειτουργίες που υλοποιούν νήματα στη main:

1) Write:

```
if (strcmp(argv[1], "write") == 0) {
    int r = 0;
    count = atoi(argv[2]); //Orisma gia arithmo aithmatwn
    _print_header(count);
    _print_environment();

    if (argc == 5) //Kai edw ayksisi tou arithmou synthikis efoson prosthesame allo 1 orisma(nimata), dhladh 5 mazi me to r.
        r = 1;

    handleDB("open"); //Anoigei h vasi gia write.

    //apothikeyoume sto struct tis prosorines times twn count,threads,r
    general.count_holder = count;
    general.thread_holder = numot;
    general.r_holder = r;

    //Twra theloume na dhmioyrghsoume ta nimata pou tha epitelesoun thn leitoyrgia tou writing
    int i,j = 0;
    while(i<numot){
        pthread_create(&genthreads[i], NULL, request_handler_w, (void*)&general);
        i++;
    }

    while(j<numot){
        pthread_join(genthreads[j], NULL);
        j++;
    }

    handleDB("close"); // Kleisimo vasis.

    //Ektypwsh apotelesmatwn
    printer("write",general.count_holder,writescost);
```

Αποτελεί επέκταση της ήδη υπάρχουσας write. Αρχικά αυξήσαμε τον αριθμό των arguments με τα νήματα, άρα ενημερώνουμε τη συνθήκη σε if(argc == 5)

από 4 γιατί θέλουμε το r σαν 5ο όρισμα. Τώρα, κάνουμε άνοιγμα και κλείσιμο της βάσης με μια βοηθητική συνάρτηση, την handleDB() την οποία θα δούμε παρακάτω. Σημασία έχει εδώ ότι ανοίγουμε τη βάση για γράψιμο και κλείνουμε πριν γίνει το ολικό print μέσω της printer. Στο ενδιάμεσο βλέπουμε ότι αποδίδονται οι τιμές που πήραμε από τη γραμμή εντολών σε κάθε ένα γνώρισμα του struct. Έτσι, προχωράμε αρχικά στη δημιουργία των νημάτων με μια επανάληψη για το πλήθος τους, με την pthread_create με ορίσματα (πίνακα που φτιάξαμε για τα νήματα, NULL, την συναρτηση request_handler_w για να καλέσουμε την write test, το struct general με τις πληροφορίες). Ύστερα, πάλι με μία επανάληψη, κάνουμε την pthread_join, για τον τερματισμό κάθε νήματος και την έναρξη του επόμενου, με όρισμα τον πίνακα των νημάτων και NULL. Τέλος, γίνεται η εκτύπωση των αποτελεσμάτων μέσω της βοηθητικής συνάρτησης printer().

2)Read:

```
else if (strcmp(argv[1], "read") == 0) {
  int r = 0;
  count = atoi(argv[2]); //Orisma gia arithmo aithmatwn
  _print_header(count);
  _print_environment();

  if (argc == 5) //Kai edw ayksisi tou arithmou synthikis efoson prosthesame allo 1 orisma(nimata), dhladh 5 mazi me to r.
      r = 1;


  handleDB("open"); //Anoigei h vasi gia write.

  //apothikeyoume sto struct tis prosorines times twn count,threads,r
  general.count_holder = count;
  general.thread_holder = numot;
  general.r_holder = r;
    //Twra theloume na dhmioyrghsoume ta nimata pou tha epitelesoun thn leitoyrgia tou reading
    int i = 0; int j = 0;
    while(i<numot){
        pthread_create(&genthreads[i], NULL, request_handler_r, (void*)&general);
        i++;
    }

    while(j<numot){
        pthread_join(genthreads[j], NULL);
        j++;
    }

    handleDB("close"); // Kleisimo vasis.

    //Ektypwsh apotelesmatwn
    printer("read",general.count_holder,readscost);
```

Είναι η επέκταση της ήδη υπάρχουσας read.

Εδώ, η σειρά των λειτουργιών είναι ακριβώς ίδια με την τροποποίηση της λειτουργίας write(άνοιγμα και κλείσιμο βάσης, επαναλήψεις, εκτύπωση) Η μοναδική διαφορά τους είναι ότι στη επανάληψη για την δημιουργία των νημάτων αλλάζει το όρισμα σε request_handler_r που είναι η συνάρτηση υπεύθυνη για την κλήση της _read_test.

3)Readwrite:

```
else if (strcmp(argv[1], "readwrite") == 0) {
  int r = 0;
  count  = atoi(argv[2]);
  _print_header(count);
  _print_environment();
  if(argc == 7){
      r = 1;
  }

  double wper = atoi(argv[4]); //Pososto pou theloume gia PUT (write percentage)
  wper = wper / 100;
  double rper = atoi(argv[5]); //Pososto pou theloume gia GET (read percentage)
  rper = rper / 100;

  handleDB("open");

  //Arxikopoihsh ths domhs gia readers
  readers.count_holder = (long)(count*rper);
  readers.thread_holder = (int)(numot*rper);
  readers.r_holder = r;

  //Dhmiourgia nimatwn readers.
  int k = 0; int l = 0;
  while(k<numot*rper){
      pthread_create(&rthreads[k], NULL, request_handler_r, (void*)&readers);
      k++;
  }

  while(l<numot*rper){
      pthread_join(rthreads[l], NULL);
      l++;
  }
  //Arxikopoihsh ths domhs gia writers
  writers.count_holder = (long)(count*wper);
  writers.thread_holder = (int)(numot*wper);
  writers.r_holder = r;

  //Dhmiourgia nhmatwn writers.
  int i = 0; int j = 0;
  while(i<numot*wper){
      pthread_create(&wthreads[i], NULL, request_handler_w, (void*)&writers);
      i++;
  }

  while(j<numot*wper){
      pthread_join(wthreads[j], NULL);
      j++;
  }

  handleDB("close");

  //Ektypwsh apotelesmatwn.
  printer("write",writers.count_holder,writescost);
  printer("read",readers.count_holder,readscost);
```

Αποτελεί την πρόσθετη ζητούμενη λειτουργία για διαμοιρασμό των λειτουργιών read και write. Είναι ένας συνδυασμός των λειτουργιών παραπάνω με μερικές βασικές διαφορές. Η διαδικασία ως προς την διαχείριση της βάσης είναι ίδια.

Βασική διαφορά είναι οι μεταβλητές wper και rper. Αυτές περιέχουν το ποσοστό των νημάτων που θα δουλέψει το write και read. Αρχικά, τις διαβάζουμε από τη γραμμή εντολών(μετατρέποντας το char σε int για να ειναι ακέραιοι) και ύστερα κάνουμε δια 100 για να πάρουμε το δεκαδικό αριθμό για τους πολλαπλασιασμούς. Επίσης τα ορίσματα εδώ είναι 7:

πχ. ./kiwi-bench readwrite 200000 20 60 40,

όπου 200.000 requests, 20 νήματα, 60 ποσοστό writes, 40 ποσοστό reads.

Το 7ο είναι πάλι το r. Έχουμε , πάλι. Τροποποιήσει τη συνθήκη των arguments να αλλάζει το r αν είναι στην 7η θέση.

Έχουμε 2 διαφορετικές αρχικοποιήσεις του struct holder μια για writes, μια για reads.

Με την ίδια μέθοδο κάνουμε επαναλήψεις για create και join με τη διαφορά ότι η επανάληψη τρέχει για (πλήθος νημάτων) * ποσοστό. Έτσι εξασφαλίζουμε ότι θα γίνει ο κατάλληλος αριθμός επαναλήψεων (και κατά συνέπεια σωστή δημιουργία και προσπέλαση των νημάτων).

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

free(genthreads); //Apodesmeysi twn malloc
free(wthreads);
free(rthreads);
return 1;
```

Τέλος, καλούμε την βοηθητική συνάρτηση printer() με κάθε φορά όρισμα να διαλέγει τι θα εκτυπώσει (στατιστικά read ή write).

Η τελευταία συνθήκη , είναι το default else του κώδικα, με μια τροποποίηση στο τι θα τυπώσει σε περίπτωση λάθος εισόδου.

Τέλος για τη Main, κάνουμε αποδέσμευση του χώρου που πήραμε με τα malloc στην αρχικοποίηση των πινάκων για τα νήματα.

Kiwi.c αρχείο:

Στο kiwi.c γινόταν το άνοιγμα και κλείσιμο της βάσης εντός των συναρτήσεων write test και read test. Αυτό το καταργήσαμε και κάναμε τις λειτουργίες διαχείρισης της βάσης σε κάθε συνθήκη της main. Αυτό γιατί κάθε γραφέας και αναγνώστης την άνοιγε μια φορά και την έκλεινε, πράγμα το οποίο θα δημιουργούσε πρόβλημα. Εδώ όμως βρίσκεται η συνάρτηση που χρησιμοποιούσαμε (και αναφέραμε) πριν:

```
DB* db;

void handleDB(char* action){ //Synarthsh gia anoigma-kleisimo vasis. Thn kanoyme epeidh theloume na kanoume energeies aytes sto arxeio bench
    if(strcmp("open",action) == 0){
        db = db_open(DATAS);
    }
    if(strcmp("close",action) == 0){
        db_close(db);
    }
}
```

Ουσιαστικά, είναι βοηθητική συνάρτηση που ανάλογα με το όρισμα open ή close θα κάνει την ανάλογη λειτουργία. Πλέον , για κάθε λειτουργία η βάση θα ανοίγει και θα κλείνει μόνο μία φορά.

_write_test:

```
void _write_test(long int count, int r, int threads) //Prosthetoume to orisma "threads". Edw tha diamoirasoume poses leitoyrgies tha ginoun ana thread.
{
    int i;
    double cost;
    long long start,end;
    Variant sk, sv;
    long int split; //Metavliti pou tha krataei posa writes/trhead
    //DB* db;

    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);

    //db = db_open(DATAS); //Syanrthsh ths engine sto db.c-anoigma vashs
start = get_ustime_sec();
split = count/threads; // Leme posa requests na ginoun ana thread.
for (i = 0; i < split; i++) { //Allazoume to hdh yparxon "count". H eggrafh na treksei gia ta sygkekrimena requests.
    if (r)
        _random_key(key, KSIZE);
    else
        snprintf(key, KSIZE, "key-%d", i);
    fprintf(stderr, "%d adding %s\n", i, key); // Print pou ginetai me thn ektelesh tou paradeigmatos ekfwnhshs
    snprintf(val, VSIZE, "val-%d", i);

    sk.length = KSIZE;
    sk.mem = key;
    sv.length = VSIZE;
    sv.mem = val;
      db_add(db, &sk, &sv);
      if ((i % 10000) == 0) {
          fprintf(stderr,"random write finished %d ops%30s\r",
                  i,
                  "");

          fflush(stderr);
      }
}

//db_close(db); //Syanrthsh ths engine sto db.c-kleisimo vashs

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

pthread_mutex_lock(&numwrites);

writescost = writescost + cost;

pthread_mutex_unlock(&numwrites);
```

Παραπάνω φαίνεται η αλλαγμένη write test συνάρτηση.

Πρώτη και σημαντική διαφορά είναι η προσθήκη του νέου ορίσματος των νημάτων της λειτουργίας. Περάστηκε σαν όρισμα κατά την κλήση της στο σημείο της main που ζητήθηκε μαζί με τις υπόλοιπες παραμέτρους, που είναι οι τιμές κάθε στοιχείου του struct στην εκάστοτε λειτουργία.

Ορίσαμε μια νέα μεταβλητή split. Η μεταβλητή αυτή, κρατάει το πηλίκο της πράξης (count = πλήθος requests / threads = νήματα). Ουσιαστικά, είναι σαν να ορίζουμε πόσα requests να πραγματοποιηθούν ανά νήμα, εφόσον στην κλήση της πλέον η συνάρτηση γίνεται για κάθε νήμα ξεχωριστά. Οπότε , αλλάζουμε και τη συνθήκη επανάληψης της for σε:

for (i = 0; i < split; i++)

δηλαδή, στο νήμα που βρισκόμαστε, να τρέξει αυτά τα requests.

Παρατηρούμε ότι , η συνθήκη αυτή έχει στην αρχή και στο τέλος της κλήσεις συναρτήσεων υπολογισμού χρόνου. Η μεταβλητή cost (που προ-υπήρχε) είναι χρήσιμη γιατί θα αποτελέσει κάθε φορά την τιμή που θα προσθέτουμε στην writecost (που αρχικοποιήσαμε στο bench.c).

Τέλος, υπολογίζουμε μέσα στην κρίσιμη περιοχή (το writescost = writescost + cost; μεταξύ των pthread_mutex_lock(&numwrites); και pthread_mutex_unlock(&numwrites);) το συνολικό κόστος. Ο λόγος που το κάνουμε αυτό είναι διότι θέλουμε τον ταυτοχρονισμό και την αποθήκευση των μετρήσεων για όλα μας τα νήματα.

Τέλος, η γραμμή printf δεν μας είναι πλέον απαραίτητη διότι όλα θα τυπώνονται μετά το κλείσιμο της βάσης σε κάθε λειτουργία της main.

_read_test:

```
void _read_test(long int count, int r, int threads) // Antistoixa prosthetoume to "threads".
{
    int i;
    int ret;
    int found = 0;
    double cost;
    long long start,end;
    Variant sk;
    Variant sv;
    long int split; //Metavliti pou tha krataei posa reads/trhead

    //DB* db;
    char key[KSIZE + 1];

    //db = db_open(DATAS); //Syanrthsh ths engine sto db.c-anoigma vashs
start = get_ustime_sec();
split = count/threads; // Leme posa requests na ginoun ana thread.
for (i = 0; i < split; i++) {
    memset(key, 0, KSIZE + 1);

    /* if you want to test random write, use the following */
    if (r)
        _random_key(key, KSIZE);
    else
        snprintf(key, KSIZE, "key-%d", i);
    fprintf(stderr, "%d searching %s\n", i, key); // Print pou ginetai me thn ektelesh tou paradeigmatos ekfwnhshs
    sk.length = KSIZE;
    sk.mem = key;
    ret = db_get(db, &sk, &sv);
    if (ret) {
        //db_free_data(sv.mem);
        found++;
    } else {
        INFO("not found key#%s",
                sk.mem);
    }

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

        fflush(stderr);
    }
}

//db_close(db); //Syanrthsh ths engine sto db.c-kleisimo vashs

end = get_ustime_sec();
cost = end - start;
pthread_mutex_lock(&numreads);
readscost = readscost + cost;
pthread_mutex_unlock(&numreads);
```

Όπως ακριβώς και στην write test, έτσι και στην read κάνουμε ακριβώς τις ίδιες αλλαγές, όσο αφορά την προσθήκη του ορίσματος για τα νήματα, την αφαίρεση διαχείρισης της βάσης, τη νέα μεταβλητή split που κάνει ότι και στην write, καθώς και την αλλαγή στη συνθήκη for. Η μόνη διαφορά είναι ότι πλέον υπολογίζεται το κόστος των αναγνωστών με τον ίδιο τρόπο όπως και στην write_test.

Db header file(db.h):

```
typedef struct _db {
//    char basedir[MAX_FILENAME];
    char basedir[MAX_FILENAME+1];
    SST* sst;
    MemTable* memtable;

    // Metavlites gia tin diaxeirisi writers-readers gia na epiteyxthei h leioyrgeia
    // toy 2ou vhmatos poluplokothtas ylopoihshs: Polloi readers vs 1 writer se olh th domh.

    pthread_mutex_t orderhandler; // Ypeythino gia to markarisma krisimwn perioxwn analoga pou th
    pthread_cond_t checkread; //Gia ta wait/signal
    int rc; // Readers Count: gia ton elexo tou arithmou twn readers
    pthread_cond_t checkwrite; //Gia ta wait/signal
    int ifw; //If Writer: Kathorizei pote tha grapsoume.
} DB;
```

Η ύπαρξη των db_add() και db_get() μέσα στις συναρτήσεις write test και read test που είπαμε ακριβώς παραπάνω, μας ενθαρρύνει να ερευνήσουμε και τα αρχεία db. h και db.c. Για να επιτύχουμε τον πλήρη ταυτοχρονισμό πρέπει να ορίσουμε τελικά και κρίσιμες περιοχές εντός των συναρτήσεων db add και get. Για αυτό, στο db.h ορίζουμε αρχικά βοηθητικές μεταβλητές που θα χρειαστούν στον έλεγχο περιπτώσεων αλλά και για την διαχείριση μεταβλητών συνθήκης. Αρχικά έχουμε το pthread_mutex_t orderhandler; . Θα χρειαστεί για να οριοθετήσουμε τις κρίσιμες περιοχές στο κώδικά μας.

Οι μεταβλητές pthread_cond_t checkread και pthread_cond_t checkwrite θα βοηθήσουν στις μεταβλητές συνθήκης. Τέλος, οι int rc και ifw είναι για έλεγχο απλών συνθηκών όπως αν υπάρχουν ακόμα αναγνώστες ή αν ακόμα γίνεται read.

(Γιατί να δηλωθούν μέσα στο struct _db και όχι global ή μέσα στο db.c?)

Όπως θα εξηγήσουμε και παρακάτω, αυτή η δήλωση αποτελεί μία καλή σκοπιμότητα. Παρατηρούμε ότι για το άνοιγμα της βάσης κάθε φορά από την db_open_ex() δηλώνεται η μεταβλητή self τύπου DB, η οποία κάνει calloc με όρισμα το μέγεθος της βάσης. Άρα μπορούμε να αξιοποιήσουμε τη μεταβλητή για ότι εργασία θέλουμε να κάνουμε στο αρχείο. (Παρόμοια λογική – ιδέα με το struct holder που κάναμε για το αρχείο bench.c και πως χρησιμοποιήσαμε τις μεταβλητές του)

<u>db.c αρχείο:</u>

```
DB* db_open_ex(const char* basedir, uint64_t cache_size)
{
    DB* self = calloc(1, sizeof(DB));

    // Arxikopoihseis metavlhtwn. O logos pou tis arxikopoih
    // tis leitourgies ths vashs. Etsi eykola me aples metav
    //Etsi, parathroume oti h metavliti self kanei calloc me
    ////////////////////////////////////////////
    pthread_mutex_init(&self->orderhandler, NULL);
    pthread_cond_init(&self->checkread, NULL);
    self->rc = 0;
    pthread_cond_init(&self->checkwrite, NULL);
    self->ifw = 0;
    ////////////////////////////////////////////
     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;
}
```

Μέσα στη συνάρτηση db_open_ex() κάνουμε την αρχικοποίηση των μεταβλητών που θα χρειαστούμε. Παρατηρούμε ότι οι συναρτήσεις που θέλουμε να επεξεργαστούμε (db add και get) παίρνουν σαν όρισμα το DB* self. Οπότε με αυτή τη λογική, χρησιμοποιούμε τις μεταβλητές που αρχικοποιήσαμε σαν μέλη του struct _db στην open.

int db_add(DB* self, Variant* key, Variant* value):

```
int db_add(DB* self, Variant* key, Variant* value)
{
    int haswritten = 0;

    pthread_mutex_lock(&self->orderhandler);//(1) Block pou shmadeyei thn krisimi peioxi.

    while(self->rc > 0){    // Oso yparxoun akoma readers tote
        pthread_cond_wait(&self->checkread, &self->orderhandler); // Blockarei to reading. Edw t
        //To dwsmeno mutex mporei na epikoinwnhsei kai me to read kai write opote blockarontas t
    }
    self->ifw = 1; //Efoson tha ginei write, kanoyme th metavliti 1.

    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);
    }
    haswritten = memtable_add(self->memtable, key, value); // Edw ginetai to "write" sti
    self->ifw = 0; //Egine ena write, ksana sto 0 h metavliti.
    pthread_cond_signal(&self->checkwrite); // Kseblokarei to write. Tha to kanei mono an
    pthread_mutex_unlock(&self->orderhandler);//(1)

    return haswritten;
}
```

Οριοθετούμε την λειτουργία της συνάρτησης ολόκληρη σαν κρίσιμη περιοχή.

Η συνθήκη :

```
while(self->rc > 0){    // Oso yparxoun akoma readers tote
    pthread_cond_wait(&self->checkread, &self->orderhandler);
    //To dwsmeno mutex mporei na epikoinwnhsei kai me to read
}
```

Ελέγχει την ύπαρξη των readers. Επειδή όμως εμείς εδώ θέλουμε να γράψουμε, τότε στέλνει σήμα να σταματήσει η ανάγνωση (αν γίνεται εκείνη τη στιγμή).

Τότε , αμέσως μετά θέτουμε τη μεταβλητή ifw (που ελέγχει αν γράφουμε) στο 1. Γίνεται ύστερα το compaction. Αποθηκεύουμε την κλήση της συνάρτησης

memtable_add(self->memtable, key, value) από το return που ήταν αρχικά στη μεταβλητή που είχαμε δηλώσει μέσα στη συνάρτηση haswritten. Αυτό γίνεται γιατί θέλουμε όλες οι ενέργειες γίνονται στην κρίσιμη περιοχή, οπότε επιστρέφουμε εκτός της περιοχής το τελικό αποτέλεσμα της κλήσης.

Τέλος, εφόσον γράψαμε στη βάση, ξανακάνουμε τη μεταβλητή ifw = 0 και με την εντολή pthread_cond_signal(&self→checkwrite) δίνει σήμα ότι μπορεί να ξεμπλοκάρει η διαδικασία εγγραφής (αν ήταν ήδη μπλοκαρισμένη) και ακολουθεί το τελικό mutex_lock.

int db_get(DB* self, Variant* key, Variant* value):

```
int db_get(DB* self, Variant* key, Variant* value)
{
    int hasread = 0;

    pthread_mutex_lock(&self->orderhandler);// (2) Block pou shmadeyei thn krisimi peioxi.
    while(self->ifw == 1){
        pthread_cond_wait(&self->checkwrite, &self->orderhandler);// Blockarei to writing. Edw theloume na kanoum
    }
    self->rc = self->rc + 1; //Efoson theloume read, vazei allon enan reader.

    if (memtable_get(self->memtable->list, key, value) == 1){ //Edw kanoume thn prosthiki ths dikh mas metavlitis
        hasread = 1; //edw tha parameinei apla 1 kai telos.
    }else{
        hasread = sst_get(self->sst, key, value); //Edw tha epistrepsei thn arxiki timi pou ekane return o defaul
    }

    self->rc = self->rc - 1; //Egine ena read, ara meiwnw antistoixa kata 1.

    if (self->rc == 0){
        pthread_cond_signal(&self->checkread); //Teleiwsan oi anagnwstes. Kane "SIGNAL" kai ksekleidws
    }

    pthread_mutex_unlock(&self->orderhandler); //(2)

    return hasread;
}
```

Η διαδικασία για την db_get() είναι λίγο πιο ιδιαίτερη. Έχουμε πολλούς γραφείς και έναν αναγνώστη. Άρα πρέπει κάπως να ελέγχουμε τις ενέργειες μας με βάση ανάλογα με την ύπαρξη γραφέα.

Αρχικά κάνουμε mutex_lock με όρισμα το νήμα που θα κάνει handle τον παραλληλισμό. Ύστερα:

```
while(self->ifw == 1){
    pthread_cond_wait(&self->checkwrite, &self->orderhandler)
}
```

Ελέγχουμε αν κάποιος πάει να γράψει. Άρα βάζουμε σε αναμονή το writing και προχωράμε για ανάγνωση.

Μεταβάλλουμε την τρέχουσα τιμή του rc κατα +1, για να δείξουμε ότι ήθρε και άλλος αναγώστης.

```
if (memtable_get(self->memtable->list, key, value) == 1){ //Edw kanoume thn
    hasread = 1; //edw tha parameinei apla 1 kai telos.
}else{
    hasread = sst_get(self->sst, key, value); //Edw tha epistrepsei thn arx
}
```

Η παραπάνω συνθήκη if επέστρεφε απλά 1. Την τροποποιήσαμε ώστε πάλι η μεταβλητή που ορίσαμε για το αποτέλεσμα(μεταβλητή hasread στην αρχή της συνάρτησης) που θα επιστρέφει η db_get() να δίνει πάλι 1 για τη συνθήκη, αλλιώς να επιστρέφει την τιμή του return που υπήρχε στον default κώδικα στο τέλος της συνάρτησης.

Εφόσον γίνεται η ανάγνωση, μειώνουμε το πλήθος των readers. Ελέγχουμε αν υπάρχουν ακόμα readers. Αν όχι, (if (self->rc == 0)) , τότε ξεμπλοκάρει το read και προχωράει η εκτέλεση. Τέλος , γίνεται mutex_unlock και επιστρέφεται η τιμή hasread.

Όλα τα παραπάνω εξυπηρετούν το σκοπό υλοποίησης του μέχρι και του 2ου βήματος.

Μέρος Δ: Αποτελέσματα – Γενικά σχόλια

Make all τελικού παραδοτέου:

```
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$
```

Μερικές εκτελέσεις:

./kiwi-bench write 350000 20

```
WRITERS statistics:
 Number of requests: 350000
 Req. service time: 0.000163
 Writes per second: 6140
 Total Cost: 57.000 sec
```

./kiwi-bench read 350000 20

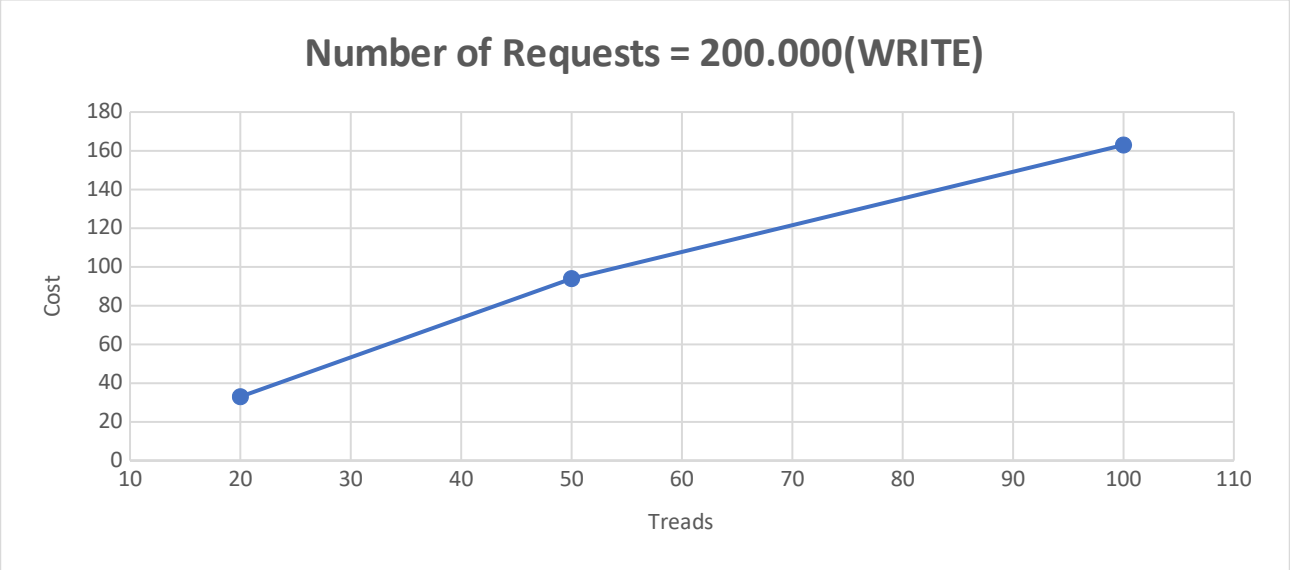
```
READERS statistics:
 Number of requests: 350000
 Req. service time: 0.000057
 Reads per second: 17500
 Total Cost: 20.000 sec
```

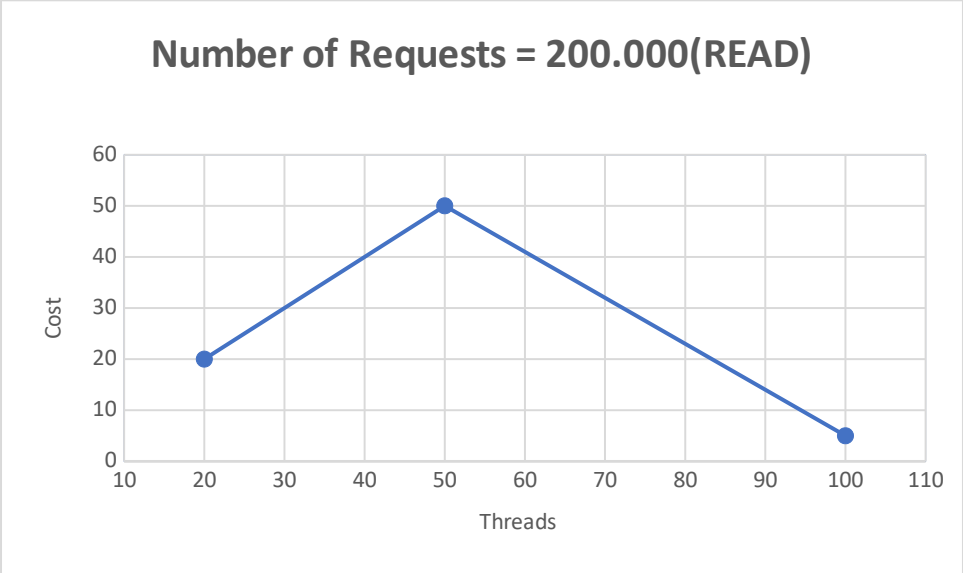
./kiwi-bench readwrite 400000 20 50 50

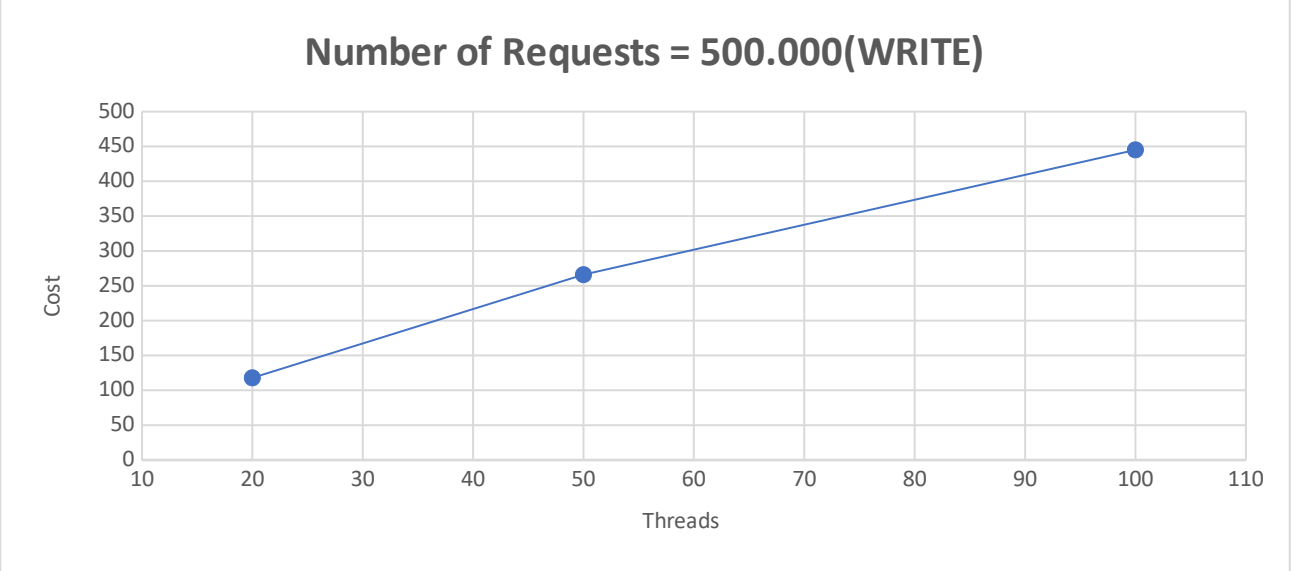
```
WRITERS statistics:
 Number of requests: 200000
 Req. service time: 0.000050
 Writes per second: 20000
 Total Cost: 10.000 sec
READERS statistics:
 Number of requests: 200000
 Req. service time: 0.000100
 Reads per second: 10000
 Total Cost: 20.000 sec
```

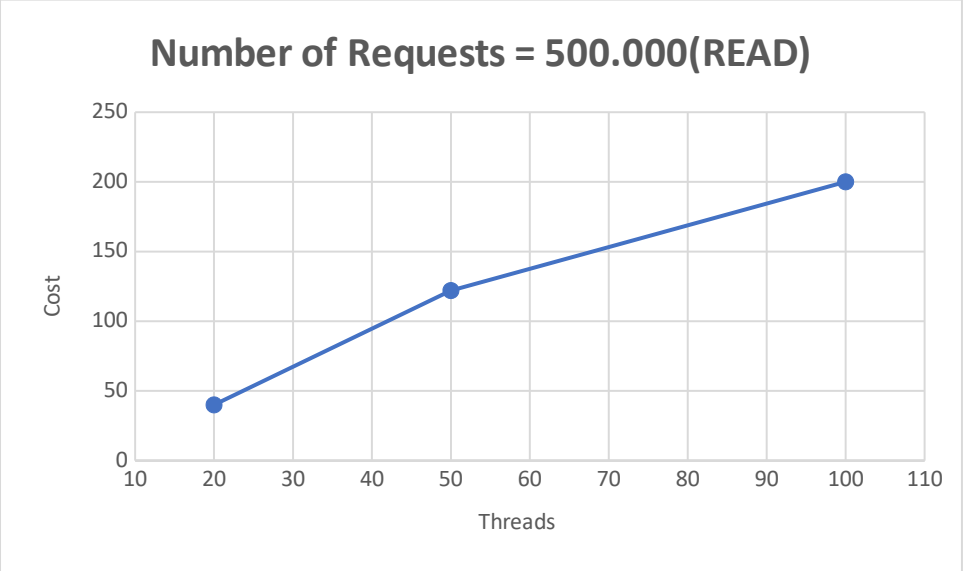
Επίσης μερικές εκτελέσεις μαζί με τα γραφήματα τους:

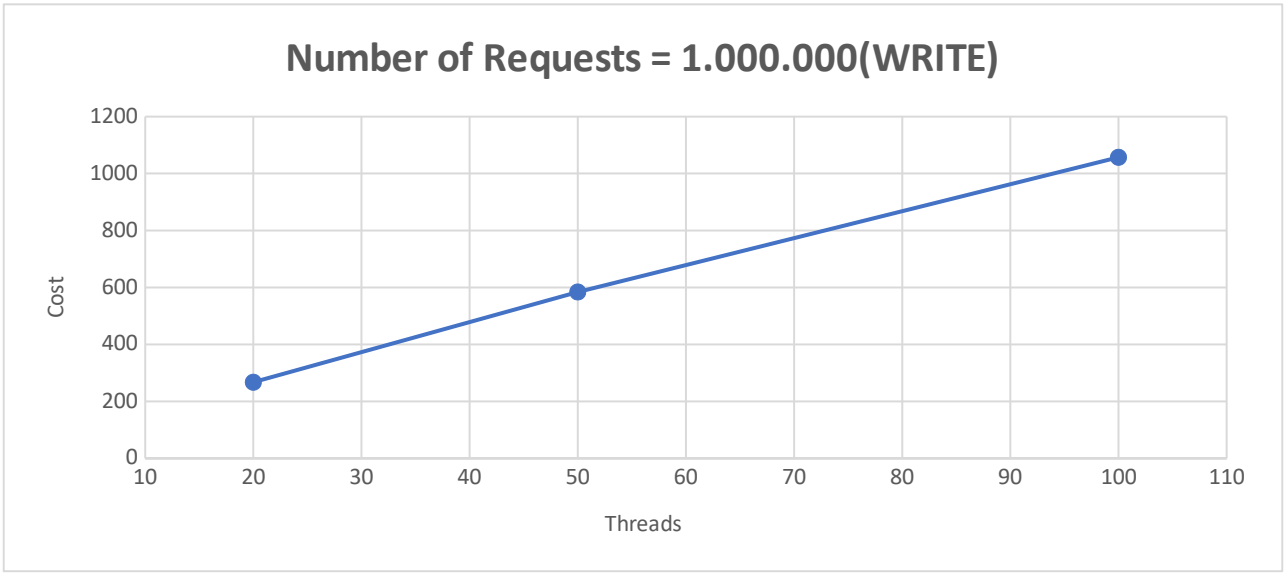
(Σημείωση: Η εκτέλεση των παρακάτω έγινε ως εξής: write X read X – make clean – make all – write Y – read Y κ.ο.κ)

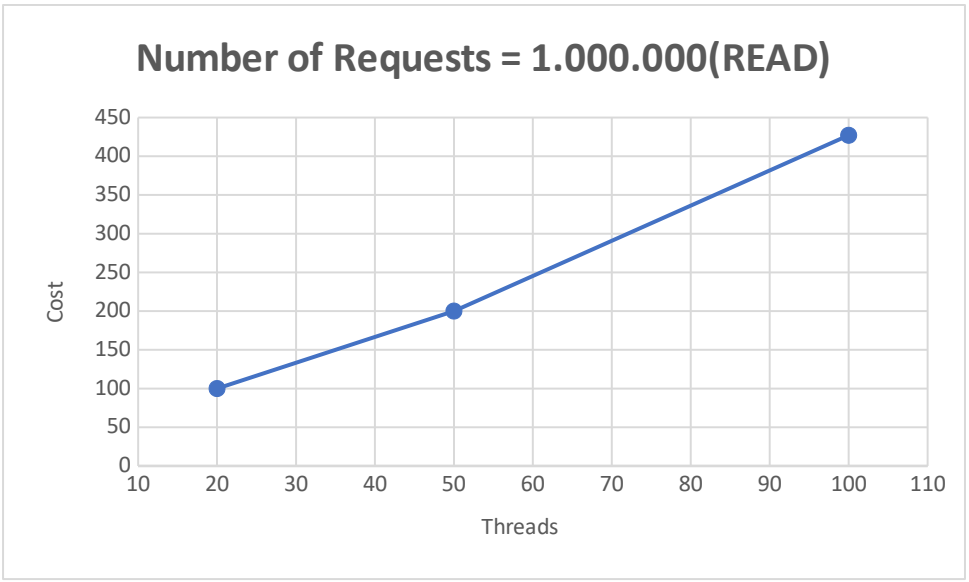



Number of Requests = 200.000(READ)
Cost
Threads
0
10
20
30
40
50
60
10
20
30
40
50
60
70
80
90
100
110


Number of Requests = 500.000(WRITE)
Cost
Threads
0
50
100
150
200
250
300
350
400
450
500
10
20
30
40
50
60
70
80
90
100
110

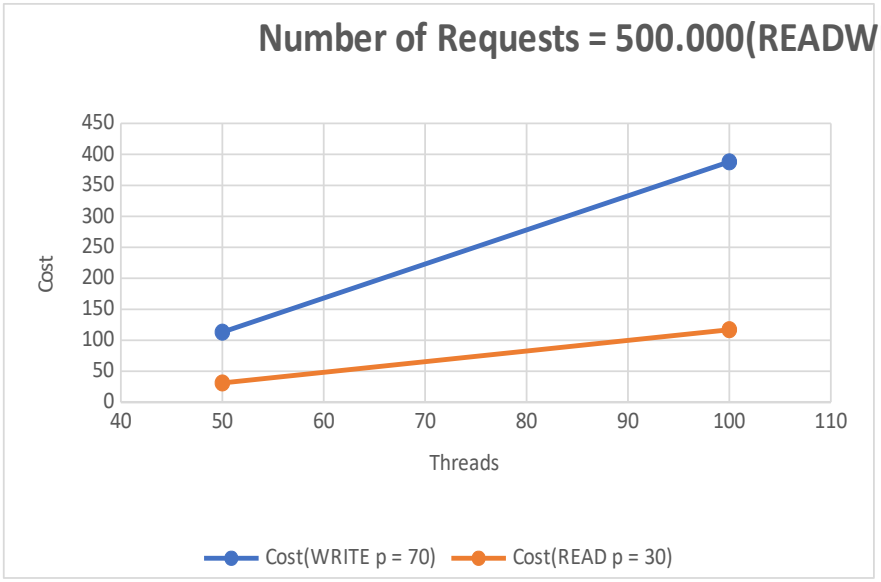

Number of Requests = 500.000(READ)
Cost
Threads
0
50
100
150
200
250
10
20
30
40
50
60
70
80
90
100
110

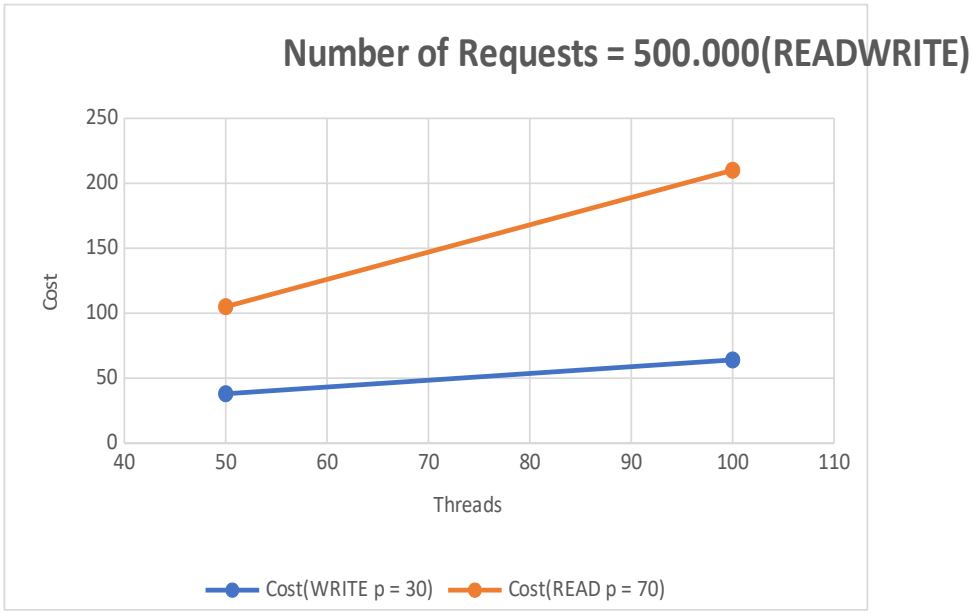

Number of Requests = 1.000.000(WRITE)
Cost
1200
1000
800
600
400
200
0
10
20
30
40
50
60
70
80
90
100
110
Threads

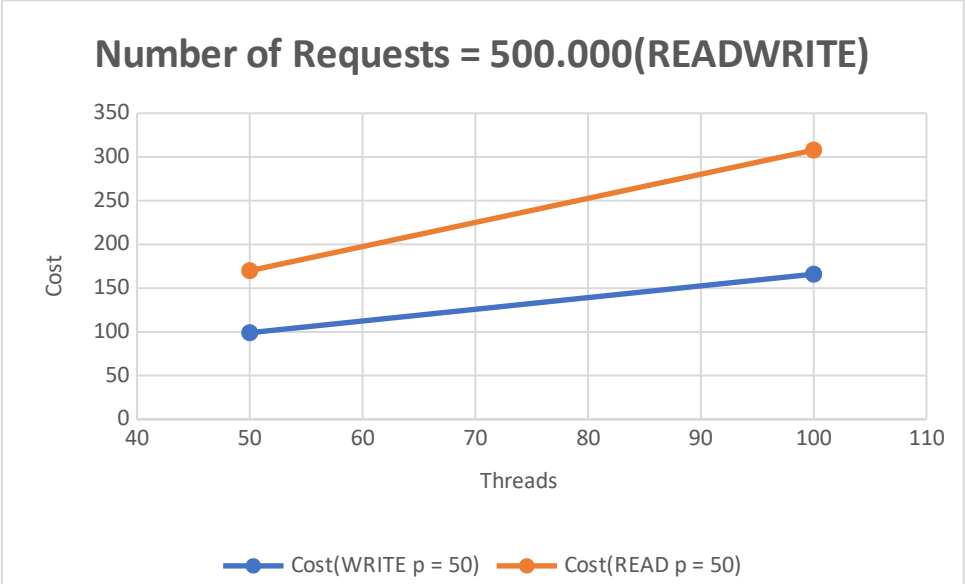

Number of Requests = 1.000.000(READ)
Cost
450
400
350
300
250
200
150
100
50
0
10
20
30
40
50
60
70
80
90
100
110
Threads

Επιπλέον μερικά παραδείγματα readwrite:

(Τα ποσοστά επιλογής κάθε εκτέλεσης φαίνονται κάτω ακριβώς απά κάθε διάγραμμα πχ. WRITE= 70 , READ = 30, δηλαδή 70% write 30% read).







Γενικά:

Η εργασία υλοποιήθηκε σε στάδια μέχρι και το 2ο βήμα, αλλά είναι πιθανόν να υπάρχουν ελλείψεις ή λογικά λάθη που μπορεί να μην δίνουν σωστό αποτέλεσμα. Προσπαθήσαμε να παραδώσουμε μια αξιόλογη προσπάθεια.

Σε σημεία που στα screenshots λείπουν κάποια σχόλια, υπάρχουν εκ νέου στον τελικό κώδικα. Επίσης υπάρχουν και όσα δεν φαίνονται καλά ή έχουν αποκοπεί.

Η άσκηση τρέχει χωρίς προβλήματα. (Segmentation faults, bus errors, floating point exceptions κτλπ). Για πολύ μικρές τιμές των read πχ λιγότερες από 150.000 δεν εκτυπώνεται χρόνος διότι εκτελείται γρήγορα οπότε η έξοδος θα βγάλει inf στο reads per second.

Ευχαριστούμε για το χρόνο σας. Καλή διόρθωση!