

ΑΝΔΡΕΑΣ ΤΡΙΑΝΤΑΦΥΛΛΟΠΟΥΛΟΣ

ΑΜ : 4504

ΜΥΡΩΝ ΚΟΥΦΟΠΟΥΛΟΣ

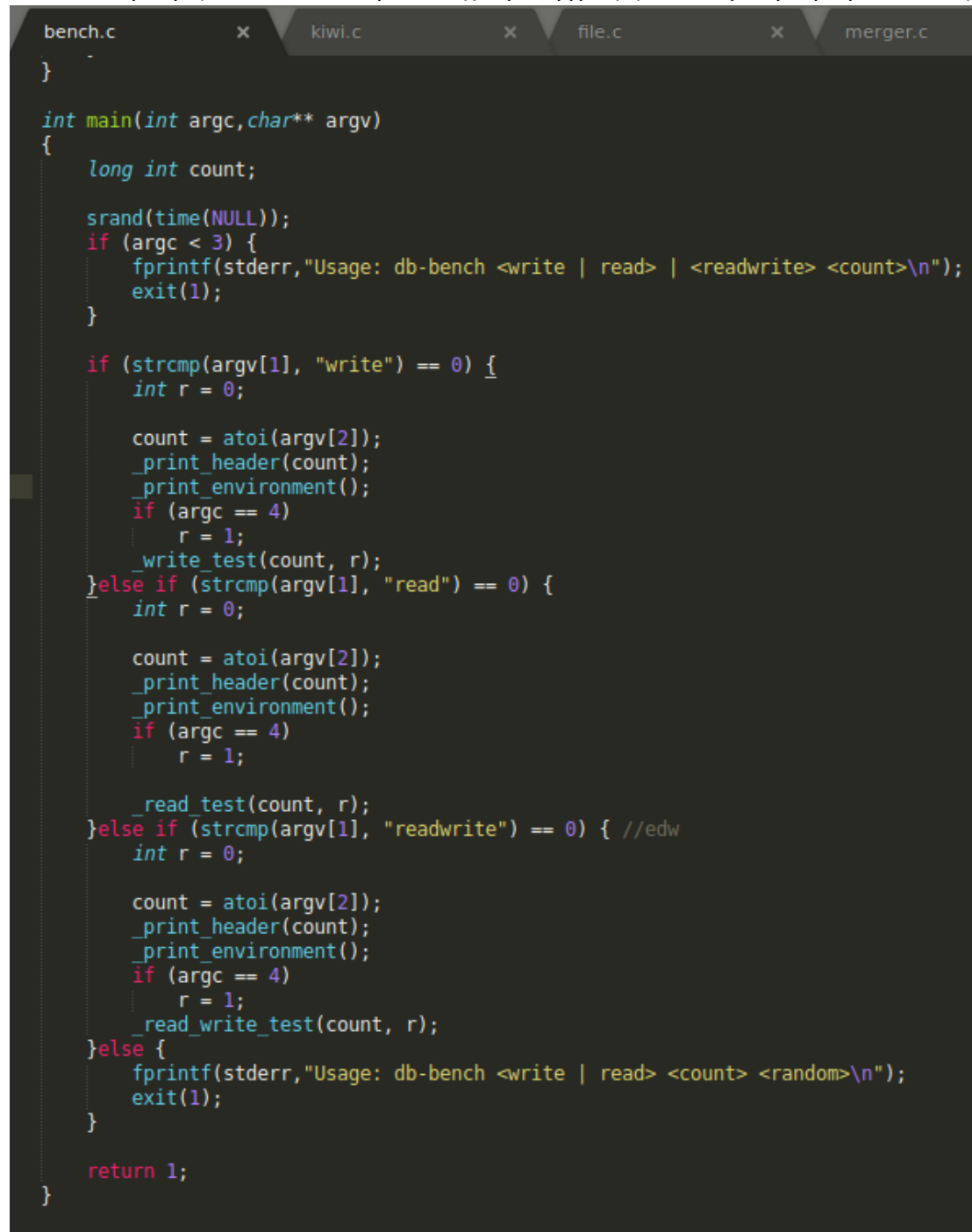
ΑΜ : 4398

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΕΡΓΑΣΙΑ 1

Ξεκινώντας την εργασία το πρώτο λάθος έγινε στην πλήρη κατανόηση των εισόδων που θα έπρεπε να έχουμε για την σωστή λειτουργία του προγράμματος. Δεν καταλάβαμε ότι ο χρήστης θα έδινε λειτουργίες και νήματα σαν δύο διαφορετικές εισόδους. Οπότε η πρώτη μας υλοποίηση ταύτιζε τα νήματα με τις λειτουργίες. Καταλάβαμε το λάθος μας μέσα από τις ερωτήσεις που ειπώθηκαν στο ecourse ! Αρχικά έγιναν αλλαγές στη main του προγράμματος, συμπληρώσαμε ένα else if το

οποίο αναγνώριζε το readwrite για να έχουμε εγγραφή και διάβασμα με μία εντολή.



```
bench.c  kiwi.c  file.c  merger.c
}

int main(int argc, char** argv)
{
    long int count;

    srand(time(NULL));
    if (argc < 3) {
        fprintf(stderr, "Usage: db-bench <write | read> | <readwrite> <count>\n");
        exit(1);
    }

    if (strcmp(argv[1], "write") == 0) {
        int r = 0;

        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();
        if (argc == 4)
            r = 1;
        _write_test(count, r);
    } else if (strcmp(argv[1], "read") == 0) {
        int r = 0;

        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();
        if (argc == 4)
            r = 1;

        _read_test(count, r);
    } else if (strcmp(argv[1], "readwrite") == 0) { //edw
        int r = 0;

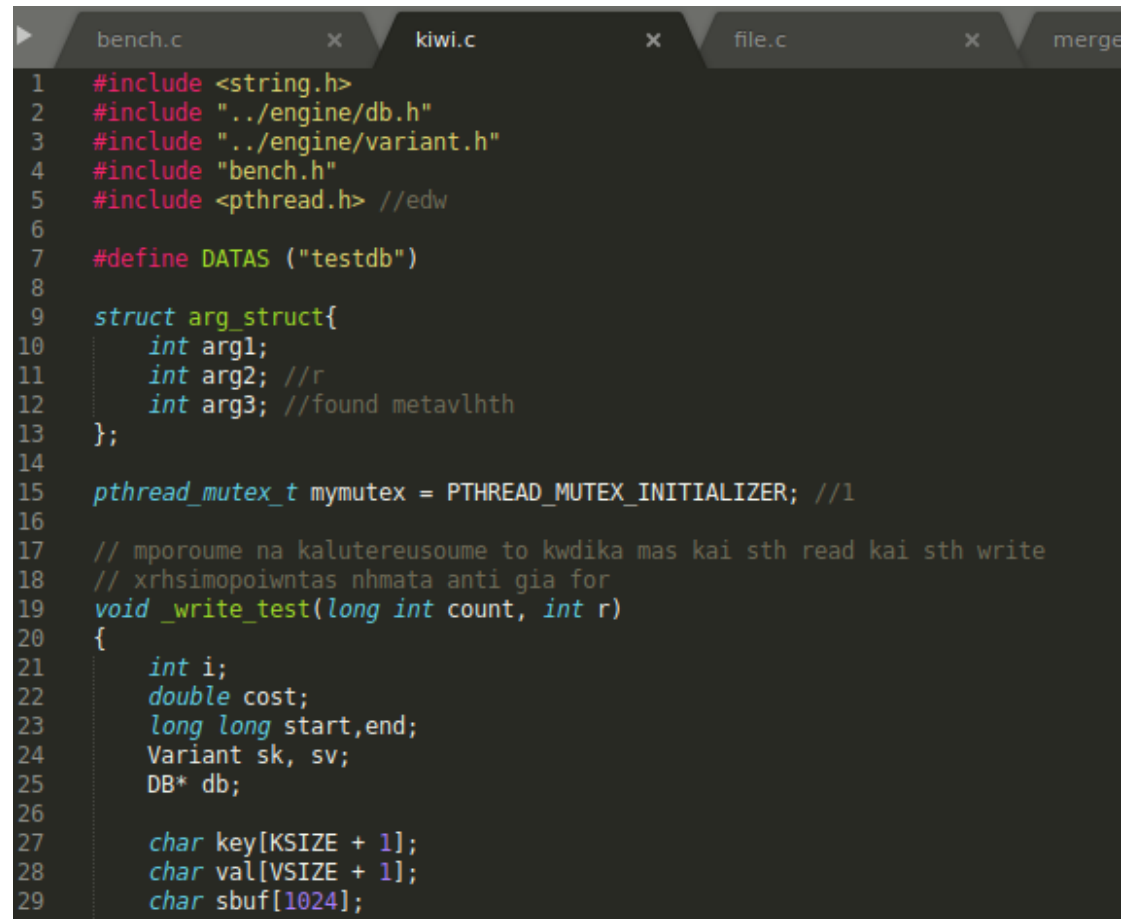
        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();
        if (argc == 4)
            r = 1;
        _read_write_test(count, r);
    } else {
        fprintf(stderr, "Usage: db-bench <write | read> <count> <random>\n");
        exit(1);
    }

    return 1;
}
```

Οι αλλαγές που ακολούθησαν έγιναν στην kiwi.c για να μπορούμε να επιτελούμε τις λειτουργίες που ζητάτε με πολυνηματισμό. Οι `_read_test` και `_write_test` που δινόντουσαν δεν αλλάχτηκαν για αυτό και δεν τις παραθέτουμε στα screenshot παρακάτω. Προστέθηκε ένα struct για την 'επικοινωνία' της `_read_write_test` με τις `_read_test1` και `_write_test1` τις οποίες δημιουργήσαμε γιατί ταυτίζοντας τις λειτουργίες με τα νήματα (το λάθος που είχαμε καταλάβει) δεν θέλαμε να έχουμε

επαναλήψεις μέσα στις συναρτήσεις μας καθώς σε κάθε νήμα αντιστοιχούσε και μία λειτουργία.

(όλες οι άλλες αλλαγές που έγιναν κρατήθηκαν στο πρόγραμμα που παραδίδεται οπότε θα αναλυθούν εκεί)



```
1 #include <string.h>
2 #include "../engine/db.h"
3 #include "../engine/variant.h"
4 #include "bench.h"
5 #include <pthread.h> //edw
6
7 #define DATAS ("testdb")
8
9 struct arg_struct{
10     int arg1;
11     int arg2; //r
12     int arg3; //found metavlhth
13 };
14
15 pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER; //1
16
17 // mporoume na kalutereusoume to kwdika mas kai sth read kai sth write
18 // xrhsimopoiwntas nhmata anti gia for
19 void _write_test(long int count, int r)
20 {
21     int i;
22     double cost;
23     long long start,end;
24     Variant sk, sv;
25     DB* db;
26
27     char key[KSIZE + 1];
28     char val[VSIZE + 1];
29     char sbuf[1024];
```

```

bench.c      x      kiwi.c      x      file.c      x
75     db_close(db);
76     //pthread_mutex_unlock(&mymutex);
77     //args->arg1 = (args->arg1) + 1;
78     return NULL;
79 }
80
81 void *_read_test1(void *arguments)    // edw gia ina mhn exoume diplh for (idio m
82 {
83
84     int ret;
85     Variant sk;
86     Variant sv;
87     DB* db;
88     char key[KSIZE + 1];
89
90     struct arg_struct *args = arguments;
91     //args->arg1 = (args->arg1) + 1;
92     pthread_mutex_lock(&mymutex);
93     db = db_open(DATAS);
94     pthread_mutex_unlock(&mymutex);
95     memset(key, 0, KSIZE + 1);
96
97     // if you want to test random write, use the following
98     if (args -> arg2)
99         _random_key(key, KSIZE);
100     else{
101         //printf("\nkey-%d\n", args -> arg1);
102         snprintf(key, KSIZE, "key-%d", args -> arg1);
103     }
104
105     fprintf(stderr, "%d searching %s\n", args -> arg1, key);
106     sk.length = KSIZE;
107     sk.mem = key;
108
109
110     pthread_mutex_lock(&mymutex);
111     ret = db_get(db, &sk, &sv);
112     args->arg1 = (args->arg1) + 1;
113     pthread_mutex_unlock(&mymutex);
114
115     if (ret) {
116         //db_free_data(sv.mem);
117         args -> arg3 = args -> arg3 + 1;
118     } else {
119         INFO("not found key#%s",
120             sk.mem);
121     }
122     if ((args -> arg1 % 10000) == 0) {
123         fprintf(stderr, "random read finished %d ops%30s\r",
124             args -> arg1,
125             "");
126         printf("\nrandom write finished %d ops\n", args -> arg1); //edw
127         fflush(stderr);
128     }
129
130     db_close(db);
131
132     return NULL;
133 }

```

```
bench.c x kiwi.c x file.c x merger.c x db.c
1 }
2
3 void *_write_test1(void *arguments) // edw gia na mhn exoume diplh for (idio me panw xwris thn for)
4 {
5     pthread_mutex_lock(&mymutex);
6     Variant sk, sv;
7     DB* db;
8
9     char key[KSIZE + 1];
10    char val[VSIZE + 1];
11    char sbuf[1024];
12
13    memset(key, 0, KSIZE + 1);
14    memset(val, 0, VSIZE + 1);
15    memset(sbuf, 0, 1024);
16
17    struct arg_struct *args = arguments;
18
19    db = db_open(DATAS);
20    //pthread_mutex_lock(&mymutex);
21    if (args -> arg2)
22        _random_key(key, KSIZE);
23    else{
24        // printf("\nkey-%d\n", args -> arg1);
25        snprintf(key, KSIZE, "key-%d", args -> arg1);
26    }
27    fprintf(stderr, "%d adding %s\n", args -> arg1, key);
28    snprintf(val, VSIZE, "val-%d", args -> arg1);
29    sk.length = KSIZE;
30    sk.mem = key;
31    sv.length = VSIZE;
32    sv.mem = val;
33    db_add(db, &sk, &sv);
34
35    if (((args -> arg1) % 10000) == 0) {
36        fprintf(stderr, "random write finished %d ops%30s\r",
37            args -> arg1,
38            "");
39        printf("\nrandom write finished %d ops\n", args -> arg1); //edw
40        fflush(stderr);
41    }
42    //pthread_mutex_lock(&mymutex); // edwwwwwww
43    args->arg1 = (args->arg1) + 1;
44    pthread_mutex_unlock(&mymutex); // edwwwwwww
45    db_close(db);
46    //pthread_mutex_unlock(&mymutex);
47    //args->arg1 = (args->arg1) + 1;
48    return NULL;
49 }
```

ΑΝΑΛΥΣΗ ΠΑΡΑΔΟΤΕΑΣ ΛΥΣΗΣ

Μετά από τα παραπάνω λάθη η σκέψη που ακολουθήθηκε είχε εξελικτική πορεία, δηλαδή υλοποιήθηκε το 1^ο βήμα της άσκησης (μια καθολική κλειδαριά στο db). Λόγω πίεσης χρόνου παραλήφθηκε το 2^ο βήμα και η υλοποίησή μας είναι το 3^ο βήμα με μερικές ελλείψεις δηλαδή δεν τρέχει πάντα χωρίς error (όμως αρκετές φορές ένα make clean και η επανάληψη της ίδιας εντολής τρέχει σωστά).

Ξεκινάμε αναλύοντας τις αλλαγές στο bench.c. Αλλαγές έγιναν μόνο στη main.

bench.c	kiwi.c
<pre> 69 } 70 71 int main(int argc, char** argv) 72 { 73 long int count; 74 int threads; 75 76 srand(time(NULL)); 77 if (argc < 4) { // na mhn dexetai ligotero apo 4 orismata 78 fprintf(stderr, "Usage: db-bench <write read> <readwrite> <count>\n"); 79 exit(1); 80 } else if (atoi(argv[2]) < 1 atoi(argv[3]) < 1) // na dexetai mono egkuro noumero sto argv[2] kai to argv[3] 81 { 82 fprintf(stderr, "Invalid number or not a number\n"); 83 exit(1); 84 } </pre>	

Αφού θέλουμε να έχουμε σαν είσοδο 4 ή παραπάνω ορίσματα, αν ο χρήστης δώσει λιγότερα γίνεται exit. Επίσης, αν ο χρήστης εισάγει μη θετικό αριθμό λειτουργιών ή νημάτων, το πρόγραμμα πάλι θα σταματήσει.

```

85     if (strcmp(argv[1], "write") == 0 || strcmp(argv[1], "read") == 0)
86     {
87         int r = 0;
88         count = atoi(argv[2]);
89         threads = atoi(argv[3]);
90         _print_header(count);
91         _print_environment();
92         if (argc == 5)
93             r = 1;
94         if (strcmp(argv[1], "write") == 0)
95         {
96             read_or_write(count, r, 1, threads); // 1 kanoume write
97         } else
98         {
99             read_or_write(count, r, 2, threads); // 2 kanoume read
100     }

```

Έχουν γίνει μικρές αλλαγές στον έλεγχο του μεμονωμένου read και write από το αρχικό. Αυτές είναι ότι καλούν την ίδια συνάρτηση (φτιαγμένη από εμάς read_or_write) με διαφορά ότι όταν κάνουμε write εισάγουμε σαν μεταβλητή των ορισμάτων το 1 και όταν κάνουμε read εισάγουμε το 2. Επίσης το πρόγραμμα πλέον δίνει τυχαία κλειδιά όταν το argc είναι ίσο με 5.

```

102 else if (strcmp(argv[1], "readwrite") == 0) // edw
103 {
104     int r = 0;
105     int per_read;
106     int per_write;
107
108     count = atoi(argv[2]); //leitourgies
109     threads = atoi(argv[3]);
110     _print_header(count);
111     _print_environment();
112     if (argc == 7 || argc == 5) // an argc = 5 tuxaia kleidia sthn test an argc = 7 tuxaia kleidia sthn percent
113     {
114         r = 1;
115     }
116     if (argc == 6 || argc == 7)
117     {
118         per_write = atoi(argv[4]);
119         per_read = atoi(argv[5]);
120         if (per_read + per_write == 100)
121             _read_write_test_percent(count, r, threads, per_write, per_read);
122         else
123         {
124             fprintf(stderr, "Invalid percentage. Doesn't add to 100%% \n");
125             exit(1);
126         }
127     }
128     else if (argc == 4 || argc == 5)
129     {
130         _read_write_test(count, r, threads);
131     }
132     else
133     {
134         fprintf(stderr, "Too many arguments given\n");
135         exit(1);
136     }
137 }

```

Εδώ γίνεται ο έλεγχος για την είσοδο readwrite όπου το πρόγραμμα εκτελεί read και write ταυτόχρονα. Οι περιπτώσεις που δεχόμαστε είναι 2.

A) 4 Είσοδοι : (π.χ.) ./kiwi-bench readwrite 1000 10

όπου 1000 ο αριθμός των λειτουργιών και 10 ο αριθμός των νημάτων.

Ο αριθμός των νημάτων σπάει στη μέση (50-50) και μοιράζονται 5 νήματα στο read και 5 στο write. Τα 5 νήματα του καθενός θα εκτελέσουν και τις 1000 λειτουργίες. Σε αυτή την περίπτωση καλείται η `_read_write_test` με τις αντίστοιχες εισόδους.

Αν δοθούν 5 είσοδοι δίνονται τυχαία κλειδιά στην δημιουργία της `_read_write_test`.

B) 6 Είσοδοι : (π.χ.) ./kiwi-bench readwrite 1000 10 60 40

όπου 1000 ο αριθμός των λειτουργιών, 10 ο αριθμός των νημάτων, 60 το ποσοστό των λειτουργιών που θα εκτελέσουν write και 40 το ποσοστό των λειτουργιών που θα εκτελέσουν read .

Ο αριθμός των νημάτων σπάει στη μέση (50-50) και μοιράζονται 5 νήματα στο read και 5 στο write. Το καθένα θα εκτελέσει το αντίστοιχο ποσοστό λειτουργιών. Σε αυτή την περίπτωση καλείται η `_read_write_test_percent` με τις αντίστοιχες εισόδους.

Ο χρήστης θα πρέπει να προσέξει τα ορίσματα του ποσοστού να αθροίζουν σε 100.

Αν δοθούν 7 είσοδοι δίνονται τυχαία κλειδιά στην δημιουργία της `_read_write_test_percent`.

Αν δοθούν πάνω από 8 ορίσματα πραγματοποιείται έξοδος με `error`.

Συνεχίζουμε με τις αλλαγές στο `kiwi.c`. (Αλλάχτηκε όλο)

```
12 struct arg_struct{
13     int arg1; // pername ta nhmata tou write pou exoume (to xrhsimopoioume gia to arg10)
14     int arg2; // r
15     int arg3; // found metavlhth
16     long int arg4; // the number of repeats for _x_test (otan exw pososto vazw to write)
17     long int arg7; // gia na kratame to threads repeat otan exoume pososto kratame to read
18     int arg8; // krataw tis allages sto pcond gia na 3erw ti paei meta
19     int arg9; // krataw to lo thread id
20     int arg10; // kratame to upoloipo twn leitourgiwn dia ta nhmata gia to write
21     int arg11; // kratame to upoloipo twn leitourgiwn dia ta nhmata gia to read(xreiazetai logo tou percent)
22     int arg12; // gia na ksekinaei to search meta to add
23     double *pinakas_xronwn; // pinakas kratanei tous xronous twn ektelesewn twn thread
24 };
```

Στην αρχή του `kiwi.c` έχουμε δημιουργήσει ένα `struct` για να περνάμε τιμές στις συναρτήσεις που χρησιμοποιούν τα νήματά μας. (`_read_test`, `_write_test`)

Ο σχολιασμός των μεταβλητών θα γίνει εντός των συναρτήσεων που τις χρησιμοποιούν και αυτών που τις αρχικοποιούν.

read or write()

```
203 void read_or_write(long int count, int r, int r_o_w, int threads)
204 {
205     double cost,time_sum,avg_time;
206     long long start,end;
207     pthread_t id[threads];
208
209     long int threads_repeat = count / threads;
210     // XRONOS GIA NHMATA
211     pid_t x = syscall(__NR_gettid);
212     start = get_ustime_sec();
213
214     struct arg_struct args;
215     args.arg1 = 0; // gia na apothikeysoyme to pid toy prwtou read h write thelei m0 mia leitourgia
216     args.arg2 = r;
217     args.arg3 = 0;
218     args.arg4 = threads_repeat;
219     args.arg7 = threads_repeat;
220     args.arg8 = 0; // arxikopoieitai sto 0 giati edw den enoxlei
221     args.arg10 = 0; //den xrhsimopoietai amesa mono gia na parei kalh timh
222     args.arg11 = 0;
223     args.arg12 = 0; // arxikopoieitai sto 1 gia na perimenoun ta read
224
225     if (count % threads != 0) // koitame an oi leitourgies diairountai akriwvs me ta nhmata
226     {
227         // auto ginetai gia na mhn xasoume kapoia leitourgia
228         args.arg10 = count % threads;
229         args.arg11 = count % threads; // einai to idio exei allaksei gia to percent
230     }
231
232     // XRONOS GIA NHMATA
233     args.pinakas_xronwn = (double*)malloc(threads * sizeof(double));
234     args.arg9 = x+1; // krataw to id ths synarthshs(+1 gia kalh afairesh)
```

Εδώ φαίνεται η συνάρτηση `read_or_write` η οποία καλείται όταν ο χρήστης θέλει να κάνει μεμονωμένα `read` ή `write`.

Γίνεται αρχικοποίηση κάποιων μεταβλητών και η πρώτη σημαντική πράξη γίνεται για να υπολογίσουμε τον αριθμό των επαναλήψεων (λειτουργιών) που θα εκτελέσει το κάθε νήμα.

Στη σειρά 211 κρατάμε το `pid` του βασικού μας προγράμματος. Δεδομένου ότι κάθε νήμα αποκτά διαφορετικό `pid` από το πρόγραμμα από το οποίο καλέστηκε, με αυτό τον τρόπο αναγνωρίζουμε κάθε νήμα και τοποθετούμε το χρόνο που χρειάζεται να εκτελεστεί, σε έναν πίνακα στο `struct`.

Όπως και το πρωτότυπο, η μεταβλητή `start` κρατάει το χρόνο εκκίνησης.

Στη συνέχεια αρχικοποιούμε το `struct` και τις μεταβλητές του.

-Στο `args.arg1` τοποθετούμε την τιμή 0, η οποία εδώ δεν θέλουμε να μας επηρεάζει καθώς χρησιμοποιείται από άλλες συναρτήσεις για να κρατάμε το πρώτο νήμα που ξεκινάει τα `read`.

-Το `args.arg2` κρατάει το `r` όπως στο πρωτότυπο.

-Το `args.arg3` κρατάει το πόσα κλειδιά έχουν βρεθεί, γι' αυτό αρχικοποιείται στο 0.

-Τα `args.arg4` και `args.arg7` κρατάνε τον αριθμό λειτουργιών (επαναλήψεων) που εκτελεί το κάθε νήμα στις `_write_test` και `_read_test` αντίστοιχα.

-Τα `args.arg8` και `args.arg12` αρχικοποιούνται στο 0 καθώς δεν θέλουμε να μας επηρεάσουν σε αυτή τη συνάρτηση δηλαδή στη `_read_test` να μην χρησιμοποιούνται τα `while`.

-Τα `args.arg10` και `args.arg11` κρατάνε τα υπόλοιπα και τα περνάνε στις `_write_test` και `_read_test` αντίστοιχα.

-Το `args.arg9` κρατάει την τιμή του `pid` της βασικής μας συνάρτησης αυξημένο κατά 1 για εύκολη τοποθέτηση των μετρήσεων στον πίνακα.

Ο έλεγχος του `if` γίνεται για να διαπιστώσουμε αν μπορούν όλες οι λειτουργίες να μοιραστούν εξ ίσου σε όλα τα νήματα. Συνεπώς αν διαιρώντας τις λειτουργίες με τα νήματα διαπιστώσουμε υπόλοιπο, θα δώσουμε τις επιπλέον λειτουργίες σε ένα νήμα (στην `read` και στην `write` αντίστοιχα).

Στη συνέχεια αρχικοποιούμε δυναμικά τον πίνακα του `struct`.

```

234     if(r_o_w == 1)    // gia na grapsei ginetai 1
235     {
236         for (int i = 0; i < threads; i++)
237         {
238             pthread_create(&id[i], NULL, _write_test, (void *) &args);
239         }
240     }
241     else if(r_o_w == 2) // gia na diavazei ginetai 2
242     {
243         for (int i = 0; i < threads; i++)
244         {
245             pthread_create(&id[i], NULL, _read_test, (void *) &args);
246         }
247     }
248     for(int i=0; i<threads; i++){
249         pthread_join(id[i], NULL);
250     }

```

Εδώ γίνεται αρχικά η δημιουργία των νημάτων μας, ανάλογα με το αν ο χρήστης ζητά read (2) ή write (1).

Στη συνέχεια, ανεξάρτητα από το αν έχει γίνει read ή write, το πρόγραμμα περιμένει να τελειώσουν όλα νήματα δημιουργήθηκαν.

```

251     end = get_ustime_sec();
252     cost = end - start;
253     // XRONOS GIA NHMATA
254     printf(LINE);
255     time_sum = 0;
256     avg_time = 0;
257     for(int i=0; i<threads; i++){
258         printf("xronos gia nhma %d = %.6f\n", i+1, *(args.pinakas_xronwn + i));
259         time_sum += *(args.pinakas_xronwn + i);
260     }
261     avg_time = time_sum / threads;
262     if(r_o_w == 1)
263     {
264         printf(LINE);
265         printf("|Random-Write (done:%ld): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec);\n",
266             count, (double)(cost / count)
267             ,(double)(count / cost)
268             ,cost);
269         printf("Random-Write average time is %.6f\n", avg_time);
270     }else if (r_o_w == 2)
271     {
272         printf(LINE);
273         printf("|Random-Read (done:%ld, found:%d): %.6f sec/op; %.1f reads /sec(estimated); cost:%.3f(sec);\n",
274             count, args.arg3,
275             (double)(cost / count),
276             (double)(count / cost),
277             cost);
278         printf("Random-Read average time is %.6f\n", avg_time);
279     }

```

Έπειτα έχουμε το end και το cost όπως και στο πρωτότυπο. Τα time_sum και avg_time χρησιμοποιούνται για τον υπολογισμό του μέσου χρόνου λειτουργίας των νημάτων της read ή της write, αθροίζοντας τις τιμές του πίνακα args.pinakas_xronwn που περιέχει τους χρόνους κάθε νήματος.

Ανάλογα αν ο χρήστης έχει ζητήσει read ή write, το πρόγραμμα τυπώνει τα αντίστοιχα αποτελέσματα.

read write test

```
284 void _read_write_test(long int count, int r, int threads_all)
285 {
286
287     double cost,time_sum, avg_time_wr, avg_time_rd;
288     long long start,end;
289     int threads;
290
291     // XRONOS GIA NHMATA
292     pid_t x = syscall(__NR_gettid);
293
294     if (threads_all == 1)
295     {
296         threads = threads_all;
297     }
298     else
299     {
300         threads = threads_all /2;
301     }
302
303     pthread_t id[threads];
304     pthread_t yo[threads];
305     long int threads_repeat = count / threads;
306
307     start = get_ustime_sec();
```

(Όσα δεν εξηγούνται και είναι ίδια σε άλλες συναρτήσεις, χρησιμοποιούνται με τον ίδιο τρόπο)

Ο τρόπος που φτιάξαμε τη συνάρτηση είναι ότι ο αριθμός όλων των νημάτων χωρίζεται στη μέση (δια 2), τα μισά νήματα εκτελούν όλες τις λειτουργίες για τα write και τα άλλα μισά όλες τις λειτουργίες για τα read. Αν έχουμε περιττό αριθμό νημάτων πχ 9 θα εκτελέσουμε 4 νήματα read και 4 write, ενώ το περισσευούμενο δεν θα εκτελεστεί. Δεδομένου ότι δίνοντας παραπάνω νήμα σε κάποια από τις λειτουργίες (read-write) θα είχαμε καλύτερο χρόνο άλλα δεν θα είχαμε έγκυρο αποτέλεσμα, αποφασίσαμε να μην δώσουμε πουθενά το έξτρα νήμα. Στην περίπτωση που ο χρήστης εισάγει 1, η συνάρτηση δίνει 1 νήμα στη read και 1 στη write.

```

309 struct arg_struct args;
310 args.arg1 = threads_all-1; // gia na apothikeysoyme to pid toy prwtoy read
311 args.arg2 = r;
312 args.arg3 = 0;
313 args.arg4 = threads_repeat; // gia na dinei to for gia thn write (to xreiazomaste gia ta pososta dn to allazw)
314 args.arg7 = threads_repeat; // gia na dinei to for gia thn read (to xreiazomaste gia ta pososta dn to allazw)
315 args.arg8 = 1; // arxikopoieitai sto 1 gia na perimenoun ta read
316 args.arg12 = 1; // arxikopoieitai sto 1 gia na perimenoun ta read
317
318 if (count % threads != 0) // koitame an oi leitourgies diairountai akriwvs me ta nhmata
319 {
320     // auto ginetai gia na mhn xasoume kapoia leitourgia
321     args.arg10 = count % threads;
322     args.arg11 = count % threads;
323 }
324 // XRONOS GIA NHMATA
325 args.pinakas_xronwn = (double*)malloc(2*threads * sizeof(double));
326 args.arg9 = x+1; // kratw to id ths synarthshs(+1 gia kalh afairesh)

```

- Στο args.arg1 δίνουμε την τιμή αυτή ώστε να αντιστοιχεί με το id του τελευταίου νήματος μιας που τα νήματα ξεκινάνε από το 0.
- Τα args.arg8 και args.arg12 θα αρχικοποιούνται στην τιμή 1 καθώς θέλουμε τα while που περιέχονται στην _read_test να περιμένουν κάποια νήματα που ο λόγος που χρησιμοποιούνται θα εξηγηθεί παρακάτω.
- Στην προκειμένη περίπτωση, λόγω του ότι χωρίζουμε τα νήματα στη μέση, αν κάποια λειτουργία χαθεί από το write θα χαθεί και από το read. Άρα περνάμε το ίδιο υπόλοιπο και στις δυο μεταβλητές.

```

329 for (int i = 0; i < threads; i++)
330 {
331     pthread_create(&id[i], NULL, _write_test, (void *) &args);
332 }
333 for (int i = 0; i < threads; i++)
334 {
335     pthread_create(&yo[i], NULL, _read_test, (void *) &args);
336 }
337
338 for (int i = 0; i < threads; i++)
339 {
340     pthread_join(id[i], NULL);
341     pthread_join(yo[i], NULL);
342 }
343
344 end = get_ustime_sec();
345 cost = end -start;
346 // XRONOS GIA NHMATA
347 printf(LINE);
348 time_sum = 0;
349 avg_time_wr = 0;
350 avg_time_rd = 0;
351 for(int i=0; i<threads; i++){
352     printf("xronos gia nhma write %d = %.6f\n", i+1, *(args.pinakas_xronwn + i));
353     time_sum += *(args.pinakas_xronwn + i);
354 }
355 avg_time_wr = time_sum / threads;
356 time_sum = 0;
357 for(int i=threads; i<threads_all; i++){
358     printf("xronos gia nhma read %d = %.6f\n", i+1, *(args.pinakas_xronwn + i));
359     time_sum += *(args.pinakas_xronwn + i);
360 }

```

Αρχικά δημιουργούμε τα νήματα του write και στη συνέχεια του read και στη συνέχεια περιμένουμε να ολοκληρωθούν όλα.

Άλλη μια διαφορά με την παραπάνω είναι ότι εδώ χρησιμοποιούμε μια ακόμα μεταβλητή για τον μέσο όρο χρόνου λειτουργίας των νημάτων (read και write αντίστοιχα).

```
361     avg_time_rd = time_sum / threads;
362     printf(LINE);
363     printf("|Random-Write  (done:%ld): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec);\n",
364           count, (double)(cost / count),
365           (double)(count / cost),
366           cost);
367     printf("|Random-Write average time is %.6f\n", avg_time_wr);
368     printf(LINE);
369     printf("|Random-Read  (done:%ld, found:%d): %.6f sec/op; %.1f reads /sec(estimated); cost:%.3f(sec)\n",
370           count, args.arg3,
371           (double)(cost / count),
372           (double)(count / cost),
373           cost);
374     printf("|Random-Read average time is %.6f\n", avg_time_rd);
375
376 }
```

Εδώ τυπώνονται τα αποτελέσματα της συνάρτησης όπως και παραπάνω.

read write test percent()

```
377 void _read_write_test_percent(long int count, int r, int threads_all, int per_write, int per_read)
378 {
379     double percent_read, percent_write, time_sum, avg_time_wr, avg_time_rd;
380     double cost;
381     long long start, end;
382     int threads, sum_for_modulo_wr, sum_for_modulo_rd;
383
384     // XRONOS GIA NHMATA
385     pid_t x = syscall(__NR_gettid);
386
387     if (threads_all == 1)
388     {
389         threads = threads_all;
390     }
391     else
392     {
393         threads = threads_all / 2;
394     }
395
396     pthread_t id[threads];
397     pthread_t yo[threads];
398
399     percent_write = per_write * 0.01;
400     percent_read = per_read * 0.01; // px 40 * 0.01 = 0,4 gia na to pollaplasiasw me to count na brw to pososto leitourgewn
401     long int threads_repeat_write = (percent_write * count) / threads; // upologizoume tis leitourgies write
402     long int threads_repeat_read = (percent_read * count) / threads; // upologizoume tis leitourgies read
403
404     start = get_ustime_sec();
```

Οι αρχικές εντολές είναι αντίστοιχες με τη συνάρτηση `_read_write_test`. Η διαφοροποίηση είναι ότι τον ακέραιο αριθμό που δίνει ο χρήστης τον πολλαπλασιάζουμε με 0,01 για να μπορέσουμε να υπολογίσουμε το ποσοστό λειτουργιών read και write.

```

406 struct arg_struct args;
407 args.arg1 = threads_all - 1;      // gia na apothikeysoyme to pid toy teleutaio read (nhmata spane sth mesh)
408 args.arg2 = r;
409 args.arg3 = 0;
410 args.arg4 = threads_repeat_write;
411 args.arg7 = threads_repeat_read;
412 args.arg8 = 1;
413
414 sum_for_modulo_wr = percent_write * count;
415 sum_for_modulo_rd = percent_read * count;
416
417 if (sum_for_modulo_wr % threads != 0) // koitame an oi leitourgies diairountai akriwvs me ta nhmata
418 {
419     // auto ginetai gia na mhn xasoume kapoia leitourgia
420     args.arg10 = sum_for_modulo_wr % threads;
421 }
422 if (sum_for_modulo_rd % threads != 0) // koitame an oi leitourgies diairountai akriwvs me ta nhmata
423 {
424     // auto ginetai gia na mhn xasoume kapoia leitourgia
425     args.arg11 = sum_for_modulo_rd % threads;
426 }
427 if(args.arg10 != 0 && args.arg11 != 0)
428 {
429     args.arg12 = 1;
430 }
431 else{
432     args.arg12 = 0;
433 }

```

Η αρχικοποίηση των struct γίνεται αντίστοιχα όπως παραπάνω με διαφορά ότι τα args.arg4 και args.arg7 παίρνουν διαφορετικό αριθμό λειτουργιών λόγω των ποσοστών στα ορίσματα. Γίνεται ένας έλεγχος ίδιος με την `_read_write_test` για να δούμε αν έχουμε υπόλοιπο, αλλά χρησιμοποιούνται διαφορετικές μεταβλητές λόγω των ποσοστών. Στη γραμμή 425 γίνεται ένας έλεγχος για το αν στις λειτουργίες της read και της write έχουμε υπόλοιπο. Εάν έχουμε και στις δύο τότε στη μεταβλητή args.arg12 περνάμε τη τιμή 1 ώστε το read να περιμένει να γίνει το write (υπάρχει while στο read). Εάν δεν ισχύουν και τα δύο ταυτόχρονα περνάμε την τιμή 0 για να 'αγνοηθεί' το while.

```

433 // XRONOS GIA NHMATA
434 args.pinakas_xronwn = (double*)malloc(2*threads * sizeof(double));
435 args.arg9 = x+1; // krataw to id ths synarthshs(+1 gia kalh afairesh)
436
437 for (int i = 0; i < threads; i++)
438 {
439     pthread_create(&id[i], NULL, _write_test, (void *) &args);
440 }
441 for (int i = 0; i < threads; i++)
442 {
443     pthread_create(&yo[i], NULL, _read_test, (void *) &args);
444 }
445 for (int i = 0; i < threads; i++)
446 {
447     pthread_join(id[i], NULL);
448     pthread_join(yo[i], NULL);
449 }
450 end = get_uptime_sec();
451 cost = end -start;

```

Εδώ φτιάχνονται ο πίνακας των χρόνων, τα νήματα και περιμένουμε να τελειώσουν, ενώ μετά σταματάει η χρονομέτρηση.

```

452 // XRONOS GIA NHMATA
453 printf(LINE);
454 time_sum = 0;
455 avg_time_wr = 0;
456 avg_time_rd = 0;
457 for(int i=0; i<threads; i++){
458     printf("xronos gia nhma write %d = %.6f\n", i+1, *(args.pinakas_xronwn + i));
459     time_sum += *(args.pinakas_xronwn + i);
460 }
461 avg_time_wr = time_sum / threads;
462 time_sum = 0;
463 for(int i=threads; i<threads_all; i++){
464     printf("xronos gia nhma read %d = %.6f\n", i+1, *(args.pinakas_xronwn + i));
465     time_sum += *(args.pinakas_xronwn + i);
466 }
467 avg_time_rd = time_sum / threads;
468 printf(LINE);
469 printf("|Random-Write (done:%.0f): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec)\n",
470     percent_write * count, (double)(cost / count),
471     (double)(count / cost),
472     cost);
473 printf("Write did %d%% of the functions \n", per_write);
474 printf("|Random-Write average time is %.6f\n", avg_time_wr);
475 printf(LINE);
476 printf("|Random-Read (done:%.0f, found:%d): %.6f sec/op; %.1f reads /sec(estimated); cost:%.3f(sec)\n",
477     percent_write * count, args.arg3, // theloume na mas tupwnei sto done posa write ekane ara na kseroume posa psaxnoume
478     (double)(cost / count),
479     (double)(count / cost),
480     cost);
481 printf("Read did %d%% of the functions \n", per_read);
482 printf("|Random-Read average time is %.6f\n", avg_time_rd);
483 }

```

Όπως και παραπάνω έτσι και εδώ υπολογίζουμε τους μέσους όρους των read και write και τυπώνουμε τα αποτελέσματα της συνάρτησης.

write test()

```
26 void *_write_test(void *arguments)
27 {
28     int i;
29     Variant sk, sv;
30     DB* db;
31
32     // XRONOS GIA NHMATA
33     double cost;
34     long long start,end;
35     start = get_ustime_sec();
36
37     char key[KSIZE + 1];
38     char val[VSIZE + 1];
39     char sbuf[1024];
40     struct arg_struct *args = (struct arg_struct *) arguments;
41     memset(key, 0, KSIZE + 1);
42     memset(val, 0, VSIZE + 1);
43     memset(sbuf, 0, 1024);
44
45     db = db_open(DATAS);
46
47     // XRONOS GIA NHMATA
48     pid_t x = syscall(__NR_gettid);
49     int y = x-(args->arg9);           //to id pou pernew katw sto nhma
```

Εδώ ξεκινάει η `_write_test` (Put). Κάνουμε αρχικοποίηση των μεταβλητών όπως στο πρωτότυπο και αποκτούμε πρόσβαση στο struct στη γραμμή 40. Στις γραμμές 48 και 49 παίρνουμε το pid του νήματος και το αφαιρούμε από το pid της γονικής συνάρτησης αυξημένο κατά 1 που είναι κρατημένο στο `args.arg9`, για να τοποθετήσουμε τους χρόνους εκτέλεσης των νημάτων στη σειρά σε ένα πίνακα μέσα στο struct. Η χρησιμότητά του φαίνεται στη γραμμή 104.


```

51  if (args->arg10 != 0 && y == 0)
52  {
53      for (i = 0; i < (args->arg4 + args->arg10); i++) { // sto for trexoume to arg4 sto write gt ekei to kratame
54          if (args->arg2) // (allaxe logo posostou)
55              _random_key(key, KSIZE);
56          else
57              snprintf(key, KSIZE, "key-%d", i);
58
59          fprintf(stderr, "%d adding %s\n", i, key);
60          snprintf(val, VSIZE, "val-%d", i);
61          sk.length = KSIZE;
62          sk.mem = key;
63          sv.length = VSIZE;
64          sv.mem = val;
65
66          db_add(db, &sk, &sv);
67
68          if ((i % 10000) == 0) {
69              fprintf(stderr, "random write finished %d ops%30s\r", i, "");
70              fflush(stderr);
71          }
72      }
73      args->arg12 = 0; // shma gia na trexei to search meta to add
74  }

```

Εδώ αρχικά ελέγχουμε αν υπάρχει υπόλοιπο στη διαίρεση των λειτουργιών με τα αντίστοιχα νήματα της write και αν υπάρχει το προσθέτει στις επαναλήψεις (args.arg4 + args.arg10). Αυτό γίνεται ώστε το υπόλοιπο να γίνει από 1 νήμα δηλαδή οι λειτουργίες που δεν μοιράστηκαν ισόποσα. Αυτές τις παραπάνω λειτουργίες θα τις εκτελέσει το 1^ο νήμα που θα μπει στη write, αυτό το εξασφαλίζουμε με τον έλεγχο του y = 0. Εσωτερικά η for εκτελείται όπως το πρωτότυπο και στη γραμμή 73 κάνουμε το args.arg12 0 για να δοθεί το σήμα στη read_test να διαβάσει και εκείνη το υπόλοιπο που έχει γραφτεί.

```

75  else{
76      for (i = 0; i < args->arg4; i++) { // sto for trexoume to arg4 sto write gt ekei
77          if (args->arg2) // to kratame (allaxe logo posostou)
78              _random_key(key, KSIZE);
79          else
80              snprintf(key, KSIZE, "key-%d", i);
81
82          fprintf(stderr, "%d adding %s\n", i, key);
83          snprintf(val, VSIZE, "val-%d", i);
84
85          sk.length = KSIZE;
86          sk.mem = key;
87          sv.length = VSIZE;
88          sv.mem = val;
89
90          db_add(db, &sk, &sv);
91
92          if ((i % 10000) == 0) {
93              fprintf(stderr, "random write finished %d ops%30s\r", i, "");
94              fflush(stderr);
95          }
96      }
97  }

```

Τα νήματα που δεν έχουν υπόλοιπο εκτελούν την else για τον αριθμό επαναλήψεων που έχει δοθεί στην args.arg4, που δόθηκε από την κάθε συνάρτηση ξεχωριστά. Η for εκτελείται με τον ίδιο τρόπο.

```

99  db_close(db);
100 args->arg8 = 0; // meta tp prwto write stelnei shma na ksekinhsoun ta read pou perimenoun
101 // XRONOS GIA NHMATA
102 end = get_ustime_sec();
103 cost = end - start;
104 *((args->pinakas_xronwn) + y) = cost;
105 return NULL;
106 }

```

Στο τέλος της write η τιμή του args.arg8 γίνεται 0 ώστε τα read που έχουν εγκλωβιστεί στη while να ξεκινήσουν να εκτελούνται. Το κάνουμε αυτό καθώς τα read χρειάζονται λιγότερο χρόνο για να εκτελεστούν οπότε χωρίς αυτή τη λειτουργία θα γινόντουσαν τα read πριν από τα write. Εμείς περιμένουμε το πρώτο write να στείλει το σήμα και όχι για κάθε write να περιμένει το read, οπότε έχουμε μια άσκοπη χρήση πόρων όμως την έχουμε περιορίσει όσο γίνεται. Αυτό σημαίνει πως το args.arg8 από την στιγμή που θα γίνει 0 μετά δεν το αλλάζουμε άλλη φορά. Στη γραμμή 104 πάμε στον πίνακα args.pinakas_xronwn και προχωράμε y θέσεις για να τοποθετήσουμε τον χρόνο εκτέλεσης του κάθε νήματος.

read test()

```

108 void *_read_test(void *arguments)
109 {
110     int i;
111     int ret;
112     Variant sk;
113     Variant sv;
114     DB* db;
115     // XRONOS GIA NHMATA
116     double cost;
117     long long start, end;
118     start = get_ustime_sec();
119
120     char key[KSIZE + 1];
121     struct arg_struct *args = (struct arg_struct *) arguments;
122
123     while(args->arg8){ // perimenei to prwto nhma tou write na teleiwsei
124     }
125
126     //XRONOS GIA NHMATA
127     pid_t x = syscall(__NR_gettid);
128     int y = x - (args->arg9); //to id pou pernew katw sto nhma
129
130     db = db_open(DATAS);

```

Στην αρχή της συνάρτησης κάνουμε τις αρχικοποιήσεις και έχουμε το while που όπως προαναφέραμε περιμένει να τελειώσει το πρώτο write ώστε να αρχίσουν να εκτελούνται όλα τα νήματα του read που έχουν εγκλωβιστεί σ' αυτό. Τα υπόλοιπα δουλεύουν όπως παραπάνω.

```

132     if (args->arg11 != 0 && y == args->arg1)
133     {
134         while(args->arg12){          // perimenei to prwto nhma tou write na teleiwsei
135         }
136         for (i = 0; i < args->arg7 + args->arg11; i++) {
137             memset(key, 0, KSIZE + 1);
138             /* if you want to test random write, use the following */
139             if (args->arg2)
140                 _random_key(key, KSIZE);
141             else
142                 snprintf(key, KSIZE, "key-%d", i);
143             fprintf(stderr, "%d searching %s\n", i, key);
144             sk.length = KSIZE;
145             sk.mem = key;
146             ret = db_get(db, &sk, &sv);
147
148             if (ret) {
149                 //db_free_data(sv.mem);
150                 args->arg3 ++;
151             } else {
152                 INFO("not found key#%s", sk.mem);
153             }
154             if ((i % 10000) == 0) {
155                 fprintf(stderr, "random read finished %d ops%30s\r", i, "");
156                 fflush(stderr);
157             }
158         }
159     }

```

Εδώ αρχικά γίνεται ο έλεγχος για το αν έχουμε υπόλοιπο στη read σε συνδυασμό με το αν βρισκόμαστε στο τελευταίο νήμα των read που έχει αποθηκευτεί στο args.arg1. Στη γραμμή 134, το while περιμένει να τελειώσει το write που έχει το υπόλοιπο, ώστε να έχουν γραφεί τα κλειδιά πριν αρχίσουμε να τα ψάχνουμε. Η μόνη αλλαγή στη for, σε σχέση με το πρωτότυπο είναι ότι όταν βρίσκουμε το κλειδί που ψάχνουμε αυξάνουμε τη μεταβλητή του struct που κρατάει το πόσα έχουμε βρεί. Στη γραμμή 136, αν η read μας έχει υπόλοιπο, το αθροίζουμε στον βασικό αριθμό επαναλήψεών μας.

```

160     else
161     {
162         for (i = 0; i < args->arg7; i++) {
163             memset(key, 0, KSIZE + 1);
164             /* if you want to test random write, use the following */
165             if (args->arg2)
166                 _random_key(key, KSIZE);
167             else
168                 snprintf(key, KSIZE, "key-%d", i);
169             fprintf(stderr, "%d searching %s\n", i, key);
170             sk.length = KSIZE;
171             sk.mem = key;
172             ret = db_get(db, &sk, &sv);
173
174             if (ret) {
175                 //db_free_data(sv.mem);
176                 args->arg3 ++;
177             } else {
178                 INFO("not found key#%s", sk.mem);
179             }
180             if ((i % 10000) == 0) {
181                 fprintf(stderr, "random read finished %d ops%30s\r", i, "");
182                 fflush(stderr);
183             }
184         }
185     }
186     db_close(db);
187     // XRONOS GIA NHMATA
188     end = get_ustime_sec();
189     cost = end - start;
190     *((args->pinakas_xronwn) + y) = cost;
191     return NULL;
192 }

```

Εδώ γίνονται αντίστοιχες λειτουργίες με την παραπάνω for στη read και ο υπόλοιπος κώδικας αντίστοιχα με παραπάνω.

Συνεχίζουμε με τα locks

```
42 DB* db_open(const char* basedir)
43 {
44
45     pthread_mutex_lock(&mymutex1);
46
47     DB *save_db_open_ex = db_open_ex(basedir, LRU_CACHE_SIZE);
48
49     //pthread_mutex_unlock(&mymutex1);
50     return save_db_open_ex;
51 }
52
53 void db_close(DB *self)
54 {
55     //pthread_mutex_lock(&mymutex1);
56     INFO("Closing database %d", self->memtable->add_count);
57
58     if (self->memtable->list->count > 0)
59     {
60         sst_merge(self->sst, self->memtable);
61         skiplist_release(self->memtable->list);
62         self->memtable->list = NULL;
63     }
64     //pthread_mutex_lock(&mymutex1);
65
66     sst_free(self->sst);
67     log_remove(self->memtable->log, self->memtable->lsn);
68     log_free(self->memtable->log);
69     memtable_free(self->memtable);
70     free(self);
71
72     pthread_mutex_unlock(&mymutex1);
73 }
```

Αυτή ήταν η υλοποίηση μας για το πρώτο βήμα με την καθολική κλειδαριά όπου κλειδώνουμε στην αρχή του db_open και ξεκλειδώνουμε στο τέλος του db_close. Όπου σιγουρεύαμε ότι οι κρίσιμες περιοχές μας δεν θα επηρεαστούν. Παρόλα αυτά εντοπίσαμε κάποια μικρά σφάλματα όταν για παράδειγμα τρέχουμε 1000 λειτουργίες και ζητάμε 500 νήματα έχουμε error για το οποίο ευθύνεται ότι στην skiplist ένα νήμα μπορεί να γράφει και το άλλο να ζητάει να διαβάσει.

Αυτή η υλοποίηση του κώδικα με το καθολικό lock δεν βρίσκεται στο παραδοτέο καθώς έχουμε προσπαθήσει να προσεγγίσουμε το 3^ο βήμα.

Στο παραδοτέο τα locks μας βρίσκονται στα:

- db.c (db_open(), db_close())
- sst.c (sst_get())
- memtable.c (memtable_add())

db.c()

```
42 DB* db_open(const char* basedir)
43 {
44
45     pthread_mutex_lock(&mymutex1);
46
47     DB *save_db_open_ex = db_open_ex(basedir, LRU_CACHE_SIZE);
48
49     pthread_mutex_unlock(&mymutex1);
50
51     return save_db_open_ex;
52 }
```

Στο db.c έχουμε υλοποιήσει μόνο locks και δεν έχει αλλαχτεί κανένα άλλο σημείο του κώδικα. Καταλάβαμε ότι το db_open είναι κρίσιμη περιοχή, κλειδώνοντας το db_open_ex παρατηρήσαμε ότι δεν είναι αποτελεσματικό διότι το return καλούσε κατευθείαν συνάρτηση όποτε δεν μπορούσαμε να την κλειδώσουμε όπως θα θέλαμε. Η λύση βρέθηκε με το να αποθηκεύσουμε την έξοδο της συνάρτησης που καλέσαμε, να ξεκλειδώσουμε και στη συνέχεια να περάσουμε το αποτέλεσμα στο return.

```
53 void db_close(DB *self)
54 {
55     pthread_mutex_lock(&mymutex1);
56     INFO("Closing database %d", self->memtable->add_count);
57
58     if (self->memtable->list->count > 0)
59     {
60         sst_merge(self->sst, self->memtable);
61         skiplist_release(self->memtable->list);
62         self->memtable->list = NULL;
63     }
64     //pthread_mutex_lock(&mymutex1);
65     sst_free(self->sst);
66     log_remove(self->memtable->log, self->memtable->lsn);
67     log_free(self->memtable->log);
68     memtable_free(self->memtable);
69     free(self);
70
71     pthread_mutex_unlock(&mymutex1);
72 }
```

Το `db_close` είναι άλλη μια κρίσιμη περιοχή που θεωρήσαμε ότι πρέπει να προστατευτεί καθώς περιέχει αναφορές σε μεταβλητές του struct `self`.

sst.c()

```
666 int sst_get(SST* self, Variant* key, Variant* value)
667 {
668     pthread_mutex_lock(&mymutex2);
669     #ifdef BACKGROUND_MERGE
670         int ret = 0;
671
672         // ...
673
674         int return_1;
675         return_1 = opt == ADD;
676         //opt == ADD;
677         pthread_mutex_unlock(&mymutex2);
678         return return_1;
679     }
680 }
681
682 #ifdef BACKGROUND_MERGE
683     pthread_mutex_unlock(&self->lock);
684 #endif
685
686 pthread_mutex_unlock(&mymutex2);
```

Αφού το `_read_test()` της `kiwi.c` χρησιμοποιεί την `db_get()` και εκείνη με τη σειρά της την `sst_get()` καταλάβαμε ότι η ευαίσθητη περιοχή βρίσκεται στην `sst_get()` για το διάβασμα των κλειδιών. Γι' αυτό το λόγο στη συνάρτηση `sst_get()` κάνουμε ένα ακόμα lock, επειδή χρησιμοποιεί ευαίσθητες περιοχές και στη γραμμή 744 της φωτογραφίας κρατάμε σε μια μεταβλητή την τιμή του αποτελέσματος `opt == ADD` για να μη φύγει εκτός του `unlock` και στη συνέχεια την επιστρέφουμε. Το `unlock` έχει υλοποιηθεί στην 747 λόγω του ελέγχου `if` για να μην επιστραφεί τιμή χωρίς να έχουμε κάνει ξεκλείδωμα, και στην 756 για το αν η ροή του προγράμματος τρέξει χωρίς να μπει στην `if`.

Memtable.c()

```
106 int memtable_add(MemTable* self, const Variant* key, const Variant* value)
107 {
108     pthread_mutex_lock(&mymutex3);
109
110     int save_memtable_edit = _memtable_edit(self, key, value, ADD);
111
112     pthread_mutex_unlock(&mymutex3);
113     return save_memtable_edit;
114 }
```

Επειδή το `_write_test()` της `kiwi.c` χρησιμοποιεί την `db_add()` και εκείνη με τη σειρά της την `memtable_add()` καταλάβαμε ότι η ευαίσθητη περιοχή βρίσκεται στην `memtable_add()` για το εγγραφή των κλειδιών. Όπως και στην `db_open()` έτσι και εδώ χρειάστηκε να κρατήσουμε το αποτέλεσμα της συνάρτησης σε μια μεταβλητή και να το επιστρέψουμε για να μπορέσουμε να κλειδώσουμε την κρίσιμη αυτή περιοχή.

Έγιναν παραπάνω προσπάθειες για lock πιο μέσα στον κώδικα, για την σωστή λειτουργία του και πριν το τελικό αποτέλεσμα που παραδόθηκε, τα έχουμε αφήσει σε σχόλια. (πχ. `Skiplist.c`, `sst_loader.c`) με `ctrl+f` και αναζήτηση της λέξης "TWRA" μπορείτε να δείτε που δοκιμάστηκαν.

Άλλη μια προσπάθεια που έγινε ήταν τα κλειδιά που εισάγουμε να είναι μοναδικά (δηλ. το κάθε νήμα να εισάγει κλειδιά 0-κ, κ-2*κ ...). Η προσπάθεια στο for ήταν σωστή απλά λόγω κάποιων error το κομμάτι αυτό αφαιρέθηκε. Τώρα το κάθε νήμα εισάγει κλειδιά από 0-κ όπου κ = λειτουργίες / νήματα.

```
160 else
161 {
162     for (i = args->arg7 * y; i < args->arg7 * (y + 1); i++) {
163         memset(key, 0, KSIZE + 1);
164         /* if you want to test random write, use the following */
165         if (args->arg2)
166             _random_key(key, KSIZE);
167         else
168             snprintf(key, KSIZE, "key-%d", i);
169     }
170 }
171
172 else{
173     for (i = args->arg4 * y; i < args->arg4 * (y + 1); i++) {
174         if (args->arg2) //to krat
175             _random_key(key, KSIZE);
176         else
177             snprintf(key, KSIZE, "key-%d", i);
178     }
179 }
```


Πως τρέχουμε το πρόγραμμα και τι αναμένεται να τυπώνει

Ξεκινάμε με τα μεμονωμένα write και read.

Απλή περίπτωση ίδιες λειτουργίες ίδια νήματα σε read write

Εντολή: ./kiwi-bench write 1000 10

Έξοδος: Γράφτηκαν 1000 κλειδιά

```
+-----+
κronos gia nhma 1 = 0.000000
κronos gia nhma 2 = 0.000000
κronos gia nhma 3 = 0.000000
κronos gia nhma 4 = 0.000000
κronos gia nhma 5 = 0.000000
κronos gia nhma 6 = 0.000000
κronos gia nhma 7 = 0.000000
κronos gia nhma 8 = 0.000000
κronos gia nhma 9 = 0.000000
κronos gia nhma 10 = 0.000000
+-----+
[Random-Write (done:1000): 0.000000 sec/op; inf writes/sec(estimated); cost:0.000(sec);
[Random-Write average time is 0.000000
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench write 1000 10
```

Εντολή: ./kiwi-bench read 1000 10

Έξοδος: Βρίσκει 1000 κλειδιά

```
+-----+
κronos gia nhma 1 = 0.000000
κronos gia nhma 2 = 0.000000
κronos gia nhma 3 = 0.000000
κronos gia nhma 4 = 0.000000
κronos gia nhma 5 = 0.000000
κronos gia nhma 6 = 0.000000
κronos gia nhma 7 = 0.000000
κronos gia nhma 8 = 0.000000
κronos gia nhma 9 = 0.000000
κronos gia nhma 10 = 0.000000
+-----+
[Random-Read (done:1000, found:1000): 0.000000 sec/op; inf reads /sec(estimated); cost:0.000(sec)
[Random-Read average time is 0.000000
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench read 1000 10
```

Λειτουργίες με περιττό αριθμό νημάτων

Εντολή: ./kiwi-bench write 1000 9

Έξοδος:

```

+-----+-----+-----+-----+-----+-----+
xronos gia nhma 1 = 0.000000
xronos gia nhma 2 = 0.000000
xronos gia nhma 3 = 0.000000
xronos gia nhma 4 = 0.000000
xronos gia nhma 5 = 0.000000
xronos gia nhma 6 = 0.000000
xronos gia nhma 7 = 0.000000
xronos gia nhma 8 = 0.000000
xronos gia nhma 9 = 0.000000
+-----+-----+-----+-----+-----+-----+
|Random-Write (done:1000): 0.000000 sec/op; inf writes/sec(estimated); cost:0.000(sec);
|Random-Write average time is 0.000000
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench write 1000 9

```

Εντολή: `./kiwi-bench write 1000 9`

Έξοδος: (όπως έχει υλοποιηθεί, κάποιες φορές βρίσκει και τα 1000 κλειδιά, κάποιες χάνει 1, δηλαδή το υπόλοιπο, δεν υπήρχε χρόνος να τελειοποιηθεί)

Αν δεν τυπωθεί το screenshot που δείχνουμε από κάτω μπορείτε να ξαναδοκιμάσετε με `make clean`.

```

xronos gia nhma 1 = 0.000000
xronos gia nhma 2 = 0.000000
xronos gia nhma 3 = 0.000000
xronos gia nhma 4 = 0.000000
xronos gia nhma 5 = 0.000000
xronos gia nhma 6 = 0.000000
xronos gia nhma 7 = 0.000000
xronos gia nhma 8 = 0.000000
xronos gia nhma 9 = 0.000000
+-----+-----+-----+-----+-----+-----+
|Random-Read (done:1000, found:1000): 0.000000 sec/op; inf reads /sec(estimated); cost:0.000(sec)
|Random-Read average time is 0.000000
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench read 1000 9

```

Εντολή: `./kiwi-bench readwrite 1000 10`

Έξοδος:

```

xronos gia nhma write 1 = 0.000000
xronos gia nhma write 2 = 0.000000
xronos gia nhma write 3 = 0.000000
xronos gia nhma write 4 = 0.000000
xronos gia nhma write 5 = 0.000000
xronos gia nhma read 6 = 0.000000
xronos gia nhma read 7 = 0.000000
xronos gia nhma read 8 = 0.000000
xronos gia nhma read 9 = 0.000000
xronos gia nhma read 10 = 0.000000
+-----+-----+-----+-----+-----+-----+
|Random-Write (done:1000): 0.000000 sec/op; inf writes/sec(estimated); cost:0.000(sec);
|Random-Write average time is 0.000000
+-----+-----+-----+-----+-----+-----+
|Random-Read (done:1000, found:1000): 0.000000 sec/op; inf reads /sec(estimated); cost:0.000(sec)
|Random-Read average time is 0.000000
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 1000 10

```

Εντολή: `./kiwi-bench readwrite 1000 9`

Έξοδος: (όπως έχει υλοποιηθεί, κάποιες φορές βρίσκει και τα 1000 κλειδιά, κάποιες χάνει 1, δηλαδή το υπόλοιπο, δεν υπήρχε χρόνος να τελειοποιηθεί)

Αν δεν τυπωθεί το screenshot που δείχνουμε από κάτω μπορείτε να ξαναδοκιμάσετε με make clean.

```
xronos gia nhma write 1 = 0.000000
xronos gia nhma write 2 = 0.000000
xronos gia nhma write 3 = 0.000000
xronos gia nhma write 4 = 0.000000
xronos gia nhma read 5 = 0.000000
xronos gia nhma read 6 = 0.000000
xronos gia nhma read 7 = 0.000000
xronos gia nhma read 8 = 0.000000
xronos gia nhma read 9 = 0.000000
+-----+
|Random-Write (done:1000): 0.000000 sec/op; inf writes/sec(estimated); cost:0.000(sec);
|Random-Write average time is 0.000000
+-----+
|Random-Read (done:1000, found:1000): 0.000000 sec/op; inf reads /sec(estimated); cost:0.000(sec)
|Random-Read average time is 0.000000
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 1000 9
```

Εντολή: ./kiwi-bench readwrite 1000 10 60 40

Έξοδος: Τα 5 νήματα εκτελούν 600 λειτουργίες write και 5 νήματα 400 λειτουργίες read

```
xronos gia nhma write 1 = 0.000000
xronos gia nhma write 2 = 0.000000
xronos gia nhma write 3 = 0.000000
xronos gia nhma write 4 = 0.000000
xronos gia nhma write 5 = 0.000000
xronos gia nhma read 6 = 0.000000
xronos gia nhma read 7 = 0.000000
xronos gia nhma read 8 = 0.000000
xronos gia nhma read 9 = 0.000000
xronos gia nhma read 10 = 0.000000
+-----+
|Random-Write (done:600): 0.000000 sec/op; inf writes/sec(estimated); cost:0.000(sec);
Write did 60% of the functions
|Random-Write average time is 0.000000
+-----+
|Random-Read (done:600, found:400): 0.000000 sec/op; inf reads /sec(estimated); cost:0.000(sec)
Read did 40% of the functions
|Random-Read average time is 0.000000
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 1000 10 60 40
```

Εντολή: ./kiwi-bench readwrite 1000 9 60 40

Έξοδος: Τα 4 νήματα εκτελούν 600 λειτουργίες write και 4 νήματα 400 λειτουργίες read (Γι' αυτό ο χρόνος του νήματος 9 είναι 0 δεν το χρησιμοποιούμε)

```
xronos gia nhma write 1 = 1.000000
xronos gia nhma write 2 = 1.000000
xronos gia nhma write 3 = 1.000000
xronos gia nhma write 4 = 1.000000
xronos gia nhma read 5 = 1.000000
xronos gia nhma read 6 = 1.000000
xronos gia nhma read 7 = 1.000000
xronos gia nhma read 8 = 1.000000
xronos gia nhma read 9 = 0.000000
+-----+
|Random-Write (done:600): 0.001000 sec/op; 1000.0 writes/sec(estimated); cost:1.000(sec);
Write did 60% of the functions
|Random-Write average time is 1.000000
+-----+
|Random-Read (done:600, found:400): 0.001000 sec/op; 1000.0 reads /sec(estimated); cost:1.000(sec)
Read did 40% of the functions
|Random-Read average time is 1.000000
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 1000 9 60 40
```

Η υλοποίηση του κώδικα με τα lock δεν είναι τελειοποιημένη, οι εντολές παραπάνω είναι απλές για την λειτουργία του κώδικα. Κάποιες φορές ο κώδικας δεν τρέχει και βγάζει σφάλματα που θα δείτε παρακάτω. Αρκετές φορές ένα make clean – make all βοηθάει στο να

τρέξουν τα πράγματα όπως θα έπρεπε. Έχουμε κατανοήσει ότι αυτό γίνεται λόγω της skiplist αλλά δεν βρέθηκε τρόπος να διορθωθεί μέσω των locks.

Μερικά σφάλματα φαίνονται παρακάτω:

```
[7011] 23 Mar 11:08:18.319 . sst_loader.c:207 Index size: 0
[7011] 23 Mar 11:08:18.319 . sst_loader.c:208 Key size: 640
[7011] 23 Mar 11:08:18.319 . sst_loader.c:209 Num blocks size: 8
[7011] 23 Mar 11:08:18.319 . sst_loader.c:210 Num entries size: 40
[7011] 23 Mar 11:08:18.319 . sst_loader.c:211 Value size: 40000
[7011] 23 Mar 11:08:18.319 . sst_loader.c:214 Filter size: 100
[7011] 23 Mar 11:08:18.319 . sst_loader.c:215 Bloom offset 2556 size: 100
[7011] 23 Mar 11:08:18.319 . sst.c:505 Deleting testdb/si/0/42.sst
[7011] 23 Mar 11:08:18.319 . sst.c:505 Deleting testdb/si/0/43.sst
[7011] 23 Mar 11:08:18.319 . sst.c:505 Deleting testdb/si/0/44.sst
[7011] 23 Mar 11:08:18.319 . sst.c:505 Deleting testdb/si/0/45.sst
[7011] 23 Mar 11:08:18.319 . sst.c:505 Deleting testdb/si/1/41.sst
[7011] 23 Mar 11:08:18.319 . sst.c:505 Deleting testdb/si/0/43.sst
[7011] 23 Mar 11:08:18.319 . sst.c:505 Deleting testdb/si/0/44.sst
[7011] 23 Mar 11:08:18.319 . sst.c:505 Deleting testdb/si/0/45.sst
[7011] 23 Mar 11:08:18.319 . sst.c:505 Deleting testdb/si/1/41.sst
[7011] 23 Mar 11:08:18.319 . file.c:66 Mapping of 2991 bytes for testdb/si/1/46.sst
Segmentation fault
myy601@myy601lab1:~/kiwi/kiwi-source/bench$
3 searching key-3
4 searching key-4
5 searching key-5
6 searching key-6
7 searching key-7
8 searching key-8
9 searching key-9
[13171] 23 Mar 11:12:42.232 . db.c:58 Closing database 0
[13171] 23 Mar 11:12:42.232 . sst.c:429 Sending termination message to the detached thread
[13171] 23 Mar 11:12:42.232 . sst.c:436 Waiting the merger thread
[13171] 23 Mar 11:12:42.232 . sst.c:180 Exiting from the merge thread as user requested
[13171] 23 Mar 11:12:42.232 . file.c:170 Truncating file testdb/si/manifest to 80 bytes
[13171] 23 Mar 11:12:42.233 . log.c:46 Removing old log file testdb/si/0.log
[13171] 23 Mar 11:12:42.233 . skiplist.c:64 SkipList refcount is at 0. Freeing up the structur
e
free(): invalid pointer
Aborted
myy601@myy601lab1:~/kiwi/kiwi-source/bench$
```

```

43 searching key-43
44 searching key-44
45 searching key-45
46 searching key-46
47 searching key-47
48 searching key-48
49 searching key-49
1 searching key-1hed 0 ops
2 searching key-2
3 searching key-3
4 searching key-4
5 searching key-5
6 searching key-6
7 searching key-7
8 searching key-8
9 searching key-9
10 searching key-10
11 searching key-11
12 searching key-12
13 searching key-13
14 searching key-14
15 searching key-15
16 searching key-16
17 searching key-17
18 searching key-18
19 searching key-19
20 searching key-20
21 searching key-21
22 searching key-22
23 searching key-23
24 searching key-24
25 searching key-25
26 searching key-26
27 searching key-27
28 searching key-28
29 searching key-29
30 searching key-30
31 searching key-31
32 searching key-32
33 searching key-33
34 searching key-34
corrupted double-linked list
Aborted
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 10000 200

```

ΤΕΛΙΚΗ ΕΞΟΔΟΣ ΕΝΤΟΛΗΣ MAKE

```

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

```