Χήτα Δανάη 4838

Εργαστήριο 1:Υλοποίηση πολυνηματικής λειτουργίας σε μηχανή αποθήκευσης δεδομένων

Περιεχόμενα:

- 1ο βήμα: Ταυτοχρονισμός με μία καθολική κλειδαριά
- 2ο βήμα: Ταυτοχρονισμός με αλγόριθμο αναγνωστών γραφέων, πολλαπλοί αναγνώστες και ένας γραφέας σε ολόκληρη τη βάση
- 3ο βήμα: Ταυτοχρονισμός με πολλαπλούς αναγνώστες και ένα γραφέα σε διαφορετικές δομές της βάσης(sst και memtable).

Χήτα Δανάη 4838

Πρώτο βήμα

Στο πρώτο βήμα μας ζητείται η υλοποίηση μίας καθολικής κλειδαριάς προκειμένου να επιτευχθεί ένα απλό επίπεδο πολυνηματισμού στη βάση. Αρχικά, τα πρώτα αρχεία που επεξεργαστήκαμε για την υλοποίηση αυτού του βήματος είναι το db.c από το φάκελο engine. Δημιουργήσαμε μία καθολική κλειδαριά glb για να κάνουμε lock και unlock στις μεθόδους db_add και db_get. Έτσι, επιτυγχάνεται ο αμοιβαίος αποκλεισμός μεταξύ των μεθόδων, δηλαδή εκτελείται κάθε φορά μόνο ένα add ή ένα get. Αυτή η υλοποίηση δεν αποδίδει το μέγιστο του ταυτοχρονισμού που επιθυμούμε, αφού δεν επιτρέπει πολλούς αναγνώστες ταυτόχρονα.

Παρακάτω φαίνεται ο επεξεργασμένος κώδικας db.c

```c
pthread_mutex_t glb = PTHREAD_MUTEX_INITIALIZER;
int db_add(DB* self, Variant* key, Variant* value)
{
    pthread_mutex_lock(&glb); //lock global variable glb
    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);
    }
    int val;
    val = memtable_add(self->memtable, key, value);
    pthread_mutex_unlock(&glb); //unlock global variable glb
    return val;   //return the result
}

int db_get(DB* self, Variant* key, Variant* value)
{
    int val;
    pthread_mutex_lock(&glb);  //lock global variable glb

    if (memtable_get(self->memtable->list, key, value) == 1)
        val = 1;
    else
    {
        val = sst_get(self->sst, key, value);
    }

    pthread_mutex_unlock(&glb); //unlock global variable glb
    return val;   //return the result
}
```

Παρατηρούμε ότι στη πρώτη γραμμή του κώδικα αρχικοποιούμε ένα mutex (glb), το οποίο το χρησιμοποιούμε για να κάνουμε lock στην αρχή (γραμμές 4,21) και unlock στο τέλος (γραμμές 14,30) της κάθε μεθόδου. Επίσης, ορίσαμε βοηθητική μεταβλητή (val) και στις δύο μεθόδους που αποθηκεύεται η τιμή των memtable_add, memtable_get και sst_get, έτσι ώστε πρώτα να γίνει το unlock και μετα η επιστροφή της τιμής, για να μην φτάσουμε σε αδιεξοδο.

Στη συνέχεια, θα δούμε την υλοποίηση μας στα αρχεία ***bench.h,kiwi.c*** *και* ***bench.c*** *που βρίσκονται μέσα στο φάκελο bench. Στην ουσία, δημιουργούμε τα νήματα και υπολογίζουμε το κόστος των νημάτων.*

bench.h

Στο συγκεκριμένο αρχείο ορίζουμε όλες τις καθολικές μεταβλητές που χρησιμοποιούμε στα αρχεία bench.c και kiwi.c

Παρακάτω φαίνεται ο επεξεργασμένος κώδικας bench.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <pthread.h> //library for threads

#define KSIZE (16)
#define VSIZE (1000)

#define LINE "+-----------------------------+---------------+-----------------------------+------------------+\n"
#define LINE1 "---------------------------------------------------------------------------------------------------\n"

long long get_ustime_sec(void);
void _random_key(char *key,int length);

pthread_mutex_t read_c;
pthread_mutex_t write_c;
double timeOfread;
double timeOfwrite;

struct figures
{
    long int c_count;
    int r_r;
    int t_thrds;
};
```

Ουσιαστικά σε αυτό το κώδικα προσθέσαμε την βιβλιοθήκη pthread.h για τα νήματα (γραμμή 8).

Στις γραμμές 19-22 ορίσαμε 2 mutex(read_c και write_c) 2 double μεταβλητές που θα μας χρησιμεύσουν για το αρχείο kiwi.c

Στις γραμμές 24-29 φτιάξαμε ένα struct με ονομασία figures το οποίο περιλαμβάνει 3 ορίσματα (c_count,r_r,t_thrds) που θα μας βοηθήσει για την υλοποίηση του bench.c.

kiwi.c

Στο συγκεκριμένο αρχείο υλοποιούνται οι συναρτήσεις write_test και read_test. Εμείς προσθέσαμε 3 βοηθητικές συναρτήσεις(open_close, time_cost_readers, time_cost_writer) και έναν δείκτη db.

Παρακάτω φαίνεται ο επεξεργασμένος κώδικας kiwi.c περιλαμβάνοντας εξήγηση για κάθε κομμάτι.

- **Pointer db and helpful function open_close.**

```
8   DB* db; //global db
9
10  //function which open and closes db according with flag
11  void open_close(int flag)
12  {
13      if(flag==1)
14      {
15          db = db_open(DATAS);
16      }
17      else if(flag==2)
18      {
19          db_close(db);
20      }
21  }
```

Χήτα Δανάη 4838

Παρατηρούμε ότι έχουμε ορίσει έναν σφαιρικό δείκτη db και την συνάρτηση open_close όπου ανάλογα με την τιμή του flag εξυπηρετεί το άνοιγμα ή το κλείσιμο της βάσης.

- **2 helpful functions (time_cost_readers and time_cost_writer).**

```
22  //Time setting in case of read
23  void time_cost_readers(double cost)
24  {
25      pthread_mutex_lock(&read_c);
26      timeOfread+= cost;  //timeOfread = timeOfread + cost
27      pthread_mutex_unlock(&read_c);
28  }
29
30  //Time setting in case of write
31  void time_cost_writer(double cost)
32  {
33      pthread_mutex_lock(&write_c);
34      timeOfwrite+= cost; //timeOfwrite = timeOfwrite + cost
35      pthread_mutex_unlock(&write_c);
36  }
```

Βλέπουμε ότι οι δύο αυτές συναρτήσεις λαμβάνουν ένα όρισμα cost για να επιτευχθεί ο αμοιβαίος αποκλεισμός(lock και unlock),δηλαδή θα βεβαιωθεί η ορθή προσπέλαση της μεταβλητής cost μέσα στις βασικές μας συναρτήσεις write_test και read_test. Επίσης, οι δύο αυτές βοηθητικές συναρτήσεις καλούνται στο τέλος των βασικών μας συναρτήσεων _write_test και _read_test.

Παρακάτω βλέπουμε τους επεξεργασμένους κώδικες τις _write_test και _read_test.

Χήτα Δανάη 4838

- **_write_test**

```
38  void _write_test(long int count, int r,int thrds) // We put an extra definition for threads
39  {
40      int i;
41      double cost;
42      int secondcount; // We use this variable for sharing threads the functions.
43      long long start,end;
44      Variant sk, sv;
45
46      char key[KSIZE + 1];
47      char val[VSIZE + 1];
48      char sbuf[1024];
49
50      memset(key, 0, KSIZE + 1);
51      memset(val, 0, VSIZE + 1);
52      memset(sbuf, 0, 1024);
53
54
55      start = get_ustime_sec();
56      secondcount=count/thrds; // sharing of threads the PUT(db_add)
57      for (i = 0; i < secondcount; i++) { //here we change count and it replaced by secondcount
58          if (r)
59              _random_key(key, KSIZE);
60          else
61              snprintf(key, KSIZE, "key-%d", i);
62          fprintf(stderr, "%d adding %s\n", i, key);
63          snprintf(val, VSIZE, "val-%d", i);
64
65          sk.length = KSIZE;
66          sk.mem = key;
67          sv.length = VSIZE;
68          sv.mem = val;
69
70          db_add(db, &sk, &sv);
71          if ((i % 10000) == 0) {
72              fprintf(stderr,"random write finished %d ops%30s\r",
73                      i,
74                      "");
75
76              fflush(stderr);
77          }
78      }
79
80      end = get_ustime_sec();
81      cost = end -start;
82      time_cost_writer(cost); //call of function
83  }
```

Χήτα Δανάη 4838

- **_read_test**

```
86  void _read_test(long int count, int r,int thrds)
87  {
88      int i;
89      int ret;
90      int found = 0;
91      double cost;
92      int secondcount; // We use this variable for sharing threads the functions.
93      long long start,end;
94      Variant sk;
95      Variant sv;
96      char key[KSIZE + 1];
97      start = get_ustime_sec();
98      secondcount=count/thrds; // sharing of threads the PUT(db_get)
99      for (i = 0; i < secondcount; i++) { //here we change count and it replaced by secondcount
100         memset(key, 0, KSIZE + 1);
101
102         /* if you want to test random write, use the following */
103         if (r)
104             _random_key(key, KSIZE);
105         else
106             snprintf(key, KSIZE, "key-%d", i);
107         fprintf(stderr, "%d searching %s\n", i, key);
108         sk.length = KSIZE;
109         sk.mem = key;
110         ret = db_get(db, &sk, &sv);
111         if (ret) {
112             //db_free_data(sv.mem);
113             found++;
114         } else {
115             INFO("not found key#%s",
116                     sk.mem);
117         }
118
119         if ((i % 10000) == 0) {
120             fprintf(stderr,"random read finished %d ops%30s\r",
121                     i,
122                     "");
123
124             fflush(stderr);
125         }
126     }
127     end = get_ustime_sec();
128     cost = end - start;
129     time_cost_readers(cost); //call of function
130 }
```

Ό,τι αλλαγές έχουμε κάνει και στις δύο συναρτήσεις _write_test και _read_test έχουν σκοπό την αύξηση της ακρίβειας των μετρήσεων κόστους. Αξίζει να σημειωθεί ότι η _write_test καλεί την add και η _read_test την get οι οποίες είναι από το αρχείο db.c. Και στις δύο συναρτήσεις χρησιμοποιούμε την μεταβλητή secondcount για να μπορέσουμε να μοιράσουμε τις λειτουργίες στα νήματα.

Από το αρχείο μας είχε δοθεί έχουμε αφαιρέσει όλα τα db_open και db_close και τα (printf(LINE)…) διότι γίνονται στο bench.c

Διότι θέλουμε να μας εκτυπώνεται μία φορά στο τέλος και όχι να μου το εκτυπώνει το κάθε νήμα.

Χήτα Δανάη 4838

bench.c

Αρχικά, στο αρχείο αυτό κάνουμε <include bench.h> γιατί όπως αναφέρθηκε και παραπάνω περιλαμβάνει global μεταβλητές που χρειάζονται στην υλοποίηση μας. Στη συνέχεια, δεν έχει γίνει καμία αλλαγή στις συναρτήσεις(_random_key , _print_header , _print_environment) και τις χρησιμοποιούμε στη main αυτούσιες όπως μας δόθηκαν.

Παρακάτω φαίνεται ο επεξεργασμένος κώδικας bench.c περιλαμβάνοντας εξήγηση για κάθε κομμάτι.

- **2 helpful functions (*call_readers and *call_writer).**

```
71  //Here we call the writer
72  void *call_writer(void *arg)
73  {
74      struct figures *f = (struct figures *)arg;  //here we make a struct
75      _write_test(f->c_count, f->r_r , f->t_thrds); //call the method _write_test from the file kiwi.c
76      return 0;
77  }
78
79  //Here we call the readers
80  void *call_readers(void *arg)
81  {
82      struct figures *f = (struct figures *)arg; //here we make a struct
83      _read_test(f->c_count, f->r_r , f->t_thrds);  //call the method _read_test from the file kiwi.c
84      return 0;
85  }
```

Χρησιμοποιούμε αυτές τις βοηθητικές συναρτήσεις πρώτα από όλα για να κάνουμε το πέρασμα των παραμέτρων και να αποκτήσουμε πρόσβαση στις μεθόδους _write_test και _read_test του kiwi.c και αφετέρου για να είναι πιο ευανάγνωστος ο κώδικας της main.

Χήτα Δανάη 4838

- **Code of main.**

```
88  int main(int argc,char** argv)
89  {
90      int mythreads;
91      long int count;
92      int i;
93      int per,per2; //percentages
94
95      //Create threads
96      pthread_t *number_code;
97      pthread_t *number_code1;
98      pthread_t *number_code2;
99
100     //Memory Commitment dynamically
101     number_code = (pthread_t*)malloc(1000000);
102     number_code1= (pthread_t*)malloc(1000000);
103     number_code2= (pthread_t*)malloc(1000000);
104     struct figures fig1,fig2,fig3;
105
106     //initialization of variables and threads
107     timeOfread = 0;
108     timeOfwrite = 0;
109     pthread_mutex_init(&read_c,NULL);
110     pthread_mutex_init(&write_c,NULL);
111
112     srand(time(NULL));
113     if (argc < 4) //here we change argc from 3 to 4 because we accept 4 arguments from the input
114     {
115         fprintf(stderr,"Usage: db-bench <write | read | readwrite> <count> <mythreads> <r> \n"); //Format which accept the program
116         exit(1);
117     }
```

Τα πρώτα βήματα μας στη main ήταν ο ορισμός των βοηθητικών μεταβλητών mythreads ,per,per2 και των number_codes όπου μας βοηθούν στη δυναμική δέσμευση της μνήμης. Επίσης αρχικοποιούμε τις δύο μεταβλητές timeOfread και timeOfwrite στο 0 και τα δύο pthreads που έχουμε ορίσει στο αρχείο bench.h. Μπορούμε να επισημάνουμε ότι το απαραίτητο πλήθος ορισμάτων από 3 άλλαξε σε 4 αφού στο κώδικα που μας είχε δοθεί δεν υπήρχε η επιλογή του πλήθους νημάτων. Τέλος, σε περίπτωση που το πλήθος των ορισμάτων δεν είναι σωστό το πρόγραμμα θα τερματίσει εκτυπώνοντας το μήνυμα στη γραμμή 115 που φαίνεται το αποδεκτό φορμάτ του εισόδου.

Χήτα Δανάη 4838

Περίπτωση Write

```
119     if (strcmp(argv[1], "write") == 0) {
120         int r = 0;
121 
122         count = atoi(argv[2]);
123         _print_header(count);
124         _print_environment();
125         if (argc == 5) //we change argc from 4 to 5
126             r = 1;
127 
128         mythreads = atoi(argv[3]); // Take the number of threads from the input
129 
130         open_close(1); //Open  base
131 
132         fig1.c_count = count;
133         fig1.r_r = r;
134         fig1.t_thrds = mythreads;
135 
136         //Creation of threads
137         for(i=0; i<mythreads; i++)
138         {
139             pthread_create(&number_code[i],NULL,call_writer,(void *)&fig1);
140         }
141         for(i=0; i<mythreads; i++)
142         {
143             pthread_join(number_code[i],NULL);
144         }
145 
146         open_close(2); //Close base
147 
148         printf(LINE);
149         printf("|Random-Write (done:%ld): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec);\n"
150         ,fig1.c_count, (double)(timeOfwrite / fig1.c_count)
151         ,(double)(fig1.c_count / timeOfwrite)
152         ,timeOfwrite);
153     }
```

Χήτα Δανάη 4838

Περίπτωση read

```
154     else if (strcmp(argv[1], "read") == 0) {
155         int r = 0;
156 
157         count = atoi(argv[2]);
158         _print_header(count);
159         _print_environment();
160         if (argc == 5) //we change argc from 4 to 5
161             r = 1;
162         mythreads = atoi(argv[3]); //Take the number of threads from the input
163 
164         open_close(1); //Open base
165 
166         fig1.c_count = count;
167         fig1.r_r = r;
168         fig1.t_thrds = mythreads;
169 
170         //Creation of threads
171         for(i=0; i<mythreads; i++)
172         {
173             pthread_create(&number_code[i],NULL,call_readers,(void *)&fig1);
174         }
175         for(i=0; i<mythreads; i++)
176         {
177             pthread_join(number_code[i],NULL);
178         }
179         open_close(2); //Close base
180 
181         printf(LINE);
182         printf("|Random-Read (done:%ld): %.6f sec/op; %.1f reads/sec(estimated); cost:%.3f(sec);\n"
183         ,fig1.c_count, (double)(timeOfread / fig1.c_count)
184         ,(double)(fig1.c_count / timeOfread)
185         ,timeOfread);
186     }
```

Χήτα Δανάη 4838

Περίπτωση readwrite

```
187     else if (strcmp(argv[1], "readwrite") == 0)
188     {
189         int r = 0;
190 
191         count = atoi(argv[2]);
192         _print_header(count);
193         _print_environment();
194         if (argc == 5) //we change argc from 4 to 5
195             r = 1;
196         mythreads = atoi(argv[3]); //Take the number of threads from the input
197 
198         open_close(1); //Open base
199         fig2.r_r=r;
200         printf("Enter a percentage for write:"); //Ask from the user to enter a percentage for  write
201         scanf("%d", &per);
202         fig2.c_count=(long) ((count*per)/100); //calculating total amount of write operations
203         fig2.t_thrds=(int) ((mythreads*per)/100);
204         fig3.r_r=r;
205         per2=100-per; //calculate percentage of read
206         fig3.c_count=(long) ((count*per2)/100); //calculating total amount of read operations
207         fig3.t_thrds=(int) ((mythreads*per2)/100);
208 
209         //Creation of threads
210         for(i=0;i<(mythreads*per/100);i++)
211         {
212             pthread_create(&number_code1[i],NULL,call_writer,(void *)&fig2);
213         }
214         for(i=0;i<(mythreads*per2/100);i++)
215         {
216             pthread_create(&number_code2[i],NULL,call_readers,(void *)&fig3);
217         }
```

```
218 
219         for(i=0;i<(mythreads*per/100);i++)
220         {
221             pthread_join(number_code1[i],NULL);
222         }
223 
224         for(i=0;i<(mythreads*per2/100);i++)
225         {
226             pthread_join(number_code2[i],NULL);
227         }
228         open_close(2); //Close base
229 
230         printf(LINE);
231         printf("|Random-Write (done:%ld): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec);\n"
232         ,fig2.c_count, (double)(timeOfwrite / fig2.c_count)
233         ,(double)(fig2.c_count / timeOfwrite)
234             ,timeOfwrite);
235 
236         printf(LINE);
237         printf("|Random-Read (done:%ld): %.6f sec/op; %.1f reads /sec(estimated); cost:%.3f(sec)\n",
238         fig3.c_count, (double)(timeOfread / fig3.c_count),
239         (double)(fig3.c_count / timeOfread),
240         timeOfread);
241 
242     }
243     else
244     {
245         fprintf(stderr,"Usage: db-bench <write | read | readwrite> <count> <mythreads> <r> \n"); //Format which accept the program
246         exit(1);
247     }
248     //free memory
249     free(number_code);
250     free(number_code1);
251     free(number_code2);
252     return 1;
253 }
```

Στα παραπάνω screenshot βλέπουμε τις περιπτώσεις write, read, readwrite ανάλογα με το δεύτερο όρισμα της εισόδου μας, δηλαδή argv[1].

Στις περιπτώσεις write και read:

Μπορεί να παρατηρήσει κάποιος ότι τα βήματα υλοποίησης και στις δύο περιπτώσεις είναι παρόμοια.

Το πλήθος των λειτουργιών είναι το τρίτο μας όρισμα δηλαδή argv[2] και κάνοντας κλήση της atoi μετατρέπει το string σε ακέραιο. Από το τέταρτο όρισμα της εισόδου μας δηλαδή το argv[3] παίρνουμε τον **αριθμό των νημάτων.**

Παρατηρούμε ότι ανοίγουμε τη βάση και στη συνέχεια αρχικοποιούμε τα δεδομένα του struct με τις μεταβλητές που έχουμε ορίσει.

Έπειτα κάνουμε μέσα στη for κάνουμε τη δημιουργία του thread καλώντας και την βοηθητικές συναρτήσεις call_writer ή call_readers αντίστοιχα.

Μετά μέσα στην επόμενη for κάνουμε pthread_join έτσι ώστε να περιμένουμε τα νήματα να τερματιστούν.

Τέλος, κλείνουμε τη βάση και εκτυπώνουμε τα αποτελέσματα.

Στη περίπτωση readwrite:

Εκτελείται η ίδια διαδικασία όπως και στη περίπτωση της write και read, αποτελώντας μία ανάμειξη αυτών των δύο.

Ουσιαστικά, ανοίγουμε και κλείνουμε την βάση μία φορά αλλά όπως είδαμε πιο πάνω, έχουμε δημιουργήσει δύο τοπικές μεταβλητές(fig2,fig3) struct για τον διαμερισμό των λειτουργιών write και read, ο οποίος εξαρτάται από το

ποσοστό per που ζητείται από τον χρήστη. Το ποσοστό per αναφέρεται στις λειτουργίες του write και το per2 στις εναπομείναντες για το read. Αξίζει να αναφερθεί ότι ανάλογα με το ποσοστό διαχωρίζεται και ο αριθμός των νημάτων. Τέλος, εκτυπώνονται τα αποτελέσματα

Στο τέλος του κώδικα της main παρατηρούμε ότι απελευθερώνουμε την μνήμη που δεσμεύσαμε.

Μέσα από τον κώδικα του βήματος 1,έχουμε δημιουργήσει με επιτυχία πολλαπλά νήματα με τη χρήση και της επιπλέον παραμέτρου readwrite. Στα επόμενα βήματα θα επεξεργαστούμε κατάλληλα την db για να πετύχουμε λύσεις με καλύτερο ταυτοχρονισμό.

Δεύτερο βήμα

Αφού στο πρώτο βήμα το επίπεδο του ταυτοχρονισμού δεν είναι ικανοποιητικό θα υλοποιήσουμε τον αλγόριθμο γραφέων αναγνωστών διατηρώντας **ίδια** τα αρχεία **bench.c, bench.h** και **kiwi.c** και αλλάζοντας μόνο τα αρχεία db.c και db.h

```c
#ifndef __DB_H__
#define __DB_H__

#include "indexer.h"
#include "sst.h"
#include "variant.h"
#include "memtable.h"
#include "merger.h"

typedef struct _db {
//     char basedir[MAX_FILENAME];
    char basedir[MAX_FILENAME+1];
    SST* sst;
    MemTable* memtable;
    int readcount;                    //
    pthread_mutex_t readcount_mutex; //
    pthread_mutex_t readwrite_mutex; //
} DB;
```

Χήτα Δανάη 4838

Στο κώδικα db.h δηλώνουμε global μεταβλητές που μπορούν να χρησιμοποιηθούν μόνο από το αρχείο db.c.

Η μόνη αλλαγή που πραγματοποιήσαμε είναι η προσθήκη της global μεταβλητής readcount που θα μας χρησιμεύσει στη μέθοδο db_get και των 2 mutex που τα χρησιμοποιούμε και στις δύο μεθόδους db_add και db_get, έτσι ώστε να μην έχουμε παράλληλα στο νήμα αναγνώστη και γραφέα.

```
1   #include <string.h>
2   #include <assert.h>
3   #include "db.h"
4   #include "indexer.h"
5   #include "utils.h"
6   #include "log.h"
7   #include <pthread.h>//
8
9
10  DB* db_open_ex(const char* basedir, uint64_t cache_size)
11  {
12      DB* self = calloc(1, sizeof(DB));
13
14      if (!self)
15          PANIC("NULL allocation");
16
17      strncpy(self->basedir, basedir, MAX_FILENAME);
18      self->sst = sst_new(basedir, cache_size);
19
20      Log* log = log_new(self->sst->basedir);
21      self->memtable = memtable_new(log);
22
23      //initialization
24      self->readcount=0;
25      pthread_mutex_init(&(self->readcount_mutex),NULL);
26      pthread_mutex_init(&(self->readwrite_mutex),NULL);
27
28      return self;
29  }
```

Παραπάνω φαίνονται οι αλλαγές που έχουμε κάνει στις πρώτες γραμμές του κώδικα db.c που περιλαμβάνουν τη προσθήκη της βιβλιοθήκης pthread.h και τις αρχικοποιήσεις που κάναμε στις γραμμές 24-26.

```
54  int db_add(DB* self, Variant* key, Variant* value)
55  {
56      pthread_mutex_lock(&self->readwrite_mutex); // lock the readwrite_mutex
57      if (memtable_needs_compaction(self->memtable))
58      {
59          INFO("Starting compaction of the memtable after %d insertions and %d deletions",
60              self->memtable->add_count, self->memtable->del_count);
61          sst_merge(self->sst, self->memtable);
62          memtable_reset(self->memtable);
63      }
64      int val; //a topic variable for committal of memtable_add
65      val = memtable_add(self->memtable, key, value);
66      pthread_mutex_unlock(&self->readwrite_mutex);  //unlock the readwrite_mutex
67      return val; //return the result
68  }
69
70  int db_get(DB* self, Variant* key, Variant* value)
71  {
72      int flag; //topic variable for the committal of sst_get
73      pthread_mutex_lock(&self->readcount_mutex); // // lock the readcount_mutex
74      self->readcount = self->readcount+1; //increase of readcount
75      if(self->readcount==1)
76      {
77          pthread_mutex_lock(&self->readwrite_mutex); //lock the readwrite_mutex
78      }
79      pthread_mutex_unlock(&self->readcount_mutex); //unlock the readcount_mutex
80
81      if (memtable_get(self->memtable->list, key, value) == 1)
82          flag = 1;
83      else
84      {
85          flag = sst_get(self->sst, key, value);
86      }
87      pthread_mutex_lock(&self->readcount_mutex); //lock the readcount_mutex
88      self->readcount = self->readcount-1; //decrease of readcount
89      if(self->readcount==0)
90      {
91         pthread_mutex_unlock(&self->readwrite_mutex); //unlock the readwrite_mutex
92      }
93      pthread_mutex_unlock(&self->readcount_mutex);//unlock the readcount_mutex
94      return flag;//return the result
95  }
```

Επιπλέον, έγινε επεξεργασία στις δύο μεθόδους db_add και db_get(όπως φαίνεται παραπάνω),έτσι ώστε να δίνουμε προτεραιότητα στους γραφείς. Δηλαδή, αν υπάρχει έστω και ένας γραφέας, οι αναγνώστες περιμένουν τον γραφέα και εάν

οι αναγνώστες ξεκινήσουν την ανάγνωση και υπάρχει γραφέας περιμένει να ολοκληρωθεί η ανάγνωση. Επίσης ,αξίζει να τονιστεί ότι μπορώ να έχω πολλούς αναγνώστες αλλά μόνο έναν γραφέα.

Η συνάρτηση db_add εξυπηρετεί την λειτουργία write.

Με το που εισέλθει ένας γραφέας κάνουμε lock στο readwrite_mutex και αφού ολοκληρωθεί η διαδικασία κάνουμε unlock στο readwrite_mutex . Έπειτα επιστρέφουμε την τιμή που είχαμε αναθέσει στη μεταβλητή val. Αυτή την ανάθεση την είχαμε κάνει για να μπορέσουμε να κάνουμε την επιστροφή της τιμής μετά το unlock αλλιώς θα βρισκόμασταν σε αδιέξοδο, δηλαδή θα μπλόκαραν τα νήματα μεταξύ τους.

Η συνάρτηση db_get εξυπηρετεί την λειτουργία read.

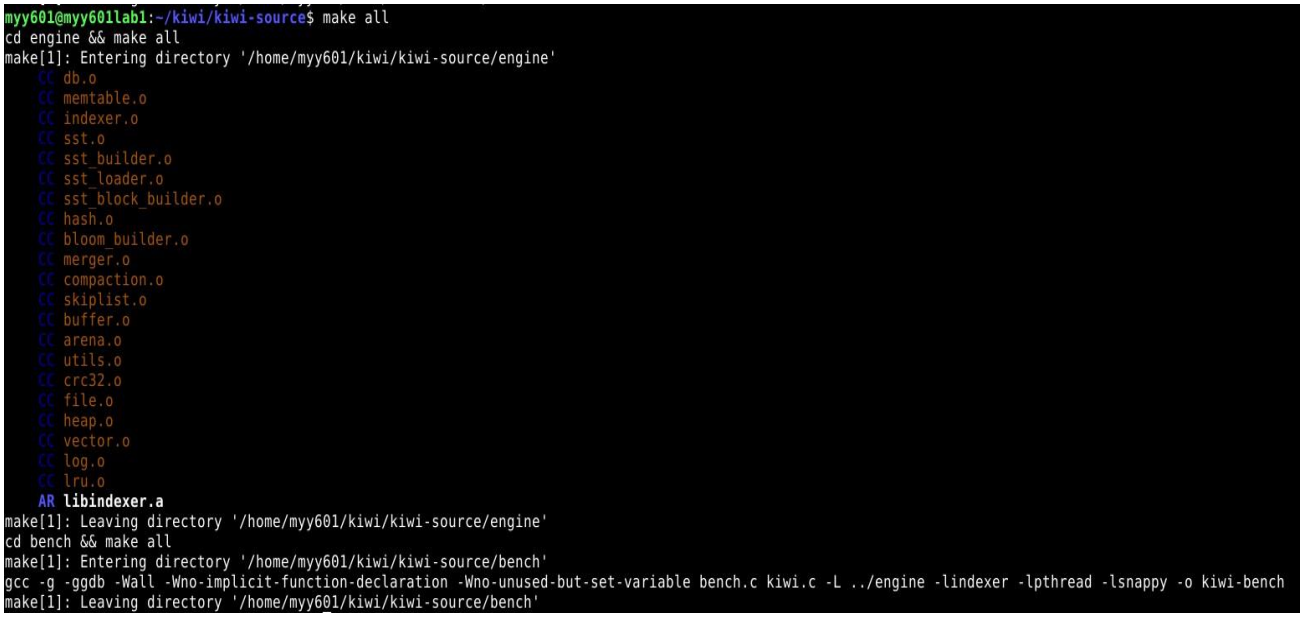
Εφόσον ένας νέος αναγνώστης εισέλθει στη συνάρτηση κάνουμε lock στο readcount_mutex και αυξάνουμε τον αριθμό των ενεργών αναγνωστών. Από τη στιγμή που υπάρχει έστω και ένας αναγνώστης κάνω lock στο readwrite_mutex για να εμποδίσω το γραφέα. Επιπλέον, όπως και στη συνάρτηση db_add έχουμε ορίσει μια τοπική μεταβλητή val για να μπορέσω στο τέλος να επιστρέψω την τιμή του αποτελέσματος. Κάθε φορά που ένας αναγνώστης τελειώνει την ανάγνωση μειώνουμε τον αριθμό των αναγνωστών κατά 1(readcount--) και εφόσον καταλήξουμε σε readcount=0 (δηλαδή τελειώσουν όλοι οι αναγνώστες) ξεμπλοκάρουμε το γραφέα.

Χήτα Δανάη 4838

Σύμφωνα με τη παραπάνω υλοποίηση του βήματος 2, όπου έχουμε καλύτερο ταυτοχρονισμό θα βγάλουμε τα συμπεράσματα μας μέσα από τα παρακάτω αποτελέσματα

Αρχικά, ανοίγουμε το τερματικό και πληκτρολογώντας την εντολή cd kiwi/kiwi-source εισερχόμαστε στο φάκελο του πηγαίου κώδικα μας. Στη συνέχεια, είναι απαραίτητο να εκτελέσουμε την εντολή make clean όπως φαίνεται στο παρακάτω στιγμιότυπο

Αμέσως μετά εκτελούμε την εντολή make all όπου εμφανίζονται τα παρακάτω



Και παρατηρούμε το επιτυχές compile όλων των αρχείων μας.

Χήτα Δανάη 4838

Για να τρέξουμε το πρόγραμμα μας πρέπει να κάνουμε cd bench και ανάλογα με το τι λειτουργία θέλουμε να τρέξουμε πληκτρολογούμε την ακόλουθη εντολή π.χ. ./kiwi-bench write 100000 5.

Όπου 100000->**Αριθμός λειτουργιών**

5-> **Αριθμός των νημάτων**

<u>Παραδείγματα της λειτουργίας write</u>

|Random-Write (done:100000): 0.000050 sec/op; 20000.0 writes/sec(estimated); cost:5.000(sec);

|Random-Write (done:300000): 0.000700 sec/op; 1428.6 writes/sec(estimated); cost:210.000(sec);

|Random-Write (done:750000): 0.000352 sec/op; 2840.9 writes/sec(estimated); cost:264.000(sec);

|Random-Write (done:1000000): 0.001702 sec/op; 587.5 writes/sec(estimated); cost:1702.000(sec);

Δοκιμάσαμε τη λειτουργία Write σε διάφορες τιμές λειτουργιών και νημάτων και προέκυψε ο παρακάτω πίνακας αποτελεσμάτων:

		Λειτουργίες				
		100000	**300000**	**500000**	**750000**	**1000000**
	5	0.000050	0.000083	0.000094	0.000099	0.000107
	15	0.000150	0.000270	0.000302	0.000352	0.000241
Νήματα	**45**	0.000440	0.000700	0.000796	0.000976	0.000749
	70	0.000630	0.000927	0.001300	0.001459	0.001156
	100	0.001090	0.001477	0.001656	0.002019	0.001702

Χήτα Δανάη 4838

Παραδείγματα της λειτουργίας read

|Random-Read (done:100000): 0.000050 sec/op; 20000.0 reads/sec(estimated); cost:5.000(sec);

|Random-Read (done:100000): 0.000360 sec/op; 2777.8 reads/sec(estimated); cost:36.000(sec);

|Random-Read (done:300000): 0.000067 sec/op; 15000.0 reads/sec(estimated); cost:20.000(sec);

|Random-Read (done:300000): 0.000100 sec/op; 10000.0 reads/sec(estimated); cost:30.000(sec);

|Random-Read (done:500000): 0.000090 sec/op; 11111.1 reads/sec(estimated); cost:45.000(sec);

Δοκιμάσαμε τη λειτουργία Read σε διάφορες τιμές λειτουργιών και νημάτων και προέκυψε ο παρακάτω πίνακας αποτελεσμάτων:

		Λειτουργίες				
		100000	**300000**	**500000**	**750000**	**1000000**
Νήματα	**5**	0.00005	0.000067	0.000078	0.000089	0.0001
	15	0.000058	0.0001	0.00009	0.00012	0.00019
	45	0.000081	0.0003	0.000268	0.000283	0.000405
	70	0.00011	0.000407	0.000414	0.000573	0.0009
	100	0.00036	0.000602	0.000659	0.000708	0.001124

Παραδείγματα της λειτουργίας readwrite

Για την εντολή readwrite ζητάμε από το χρήστη να μας δώσει το ποσοστό για το write οπότε το υπόλοιπο ποσοστό γίνεται

αυτόματα read. Στα παραδείγματα μας μεταβάλλουμε το ποσοστό όπως και τον αριθμό λειτουργιών και νημάτων.

Στις παρακάτω περιπτώσεις παραδειγμάτων εκτελούμε την εντολή ./kiwi-bench readwrite 1500000 10.

1η περίπτωση

Percentage(write) > Percentage(read) (90%-10%)

|Random-Write (done:1350000): 0.000204 sec/op; 4909.1 writes/sec(estimated); cost:275.000(sec);

|Random-Read (done:150000): 0.000193 sec/op; 5172.4 reads /sec(estimated); cost:29.000(sec)

Συνολικός χρόνος 1 = 0.000204 + 0.000193 = **0.000397**

2η περίπτωση

Percentage(write) > Percentage(read) (80%-20%)

|Random-Write (done:1200000): 0.000204 sec/op; 4898.0 writes/sec(estimated); cost:245.000(sec);

|Random-Read (done:300000): 0.000183 sec/op; 5454.5 reads /sec(estimated); cost:55.000(sec)

Συνολικός χρόνος 2 = 0.000204 + 0.000183 = **0,000387**

3η περίπτωση

Percentage(write) = Percentage(read) (50%-50%)

|Random-Write (done:750000): 0.000160 sec/op; 6250.0 writes/sec(estimated); cost:120.000(sec);

|Random-Read (done:750000): 0.000147 sec/op; 6818.2 reads /sec(estimated); cost:110.000(sec)

Συνολικός χρόνος 3 = 0.000160 + 0.000147 = **0,000307**

Χήτα Δανάη 4838

4η περίπτωση

Percentage(write) < Percentage(read) (10%-90%)

|Random-Write (done:150000): 0.000107 sec/op; 9375.0 writes/sec(estimated); cost:16.000(sec);

|Random-Read (done:1350000): 0.000100 sec/op; 10000.0 reads /sec(estimated); cost:135.000(sec)

Συνολικός χρόνος 4 = 0.000107 + 0.000100 = **0,000207**

Παρατηρήσεις για όλα τα πειράματα

- Όταν εκτελώ write και read με τις ίδιες τιμές λειτουργιών και νημάτων, ο χρόνος απόκρισης που απαιτείται για το read είναι μικρότερος του write, αφού όπως έχουμε πει υπάρχει μόνο ένας γραφέας που μπορεί να κάνει τη λειτουργία write, ενώ πολλοί αναγνώστες παράλληλα που μπορούν να κάνουν τη λειτουργία read.
- Όσο μεγαλύτερος είναι το πλήθος των λειτουργιών και των νημάτων τόσο μεγαλύτερος είναι ο χρόνος απόκρισης σε όποια λειτουργία και εάν εκτελέσουμε(write ή read).
- Μελετώντας το συνολικό χρόνο των δύο πρώτων περιπτώσεων της λειτουργίας readwrite, μπορούμε να αντιληφθούμε ότι αυξάνεται όσο μεγαλώνει το ποσοστό του write. Δηλαδή

  Συνολικός χρόνος 1 > Συνολικός χρόνος 2

- Στη 3η περίπτωση του readwrite που μοιράζονται οι λειτουργίες ισάξια, φαίνεται πάλι ότι το read εκτελείται σε λιγότερο χρόνο.
- Προφανέστατα, όσο μειώνεται το ποσοστό του write και αυξάνεται το ποσοστό του read **μειώνεται** ο συνολικός

χρόνος απόκρισης, αφού οι αναγνώστες εκτελούνται παράλληλα. Δηλαδή

Συνολικός χρόνος 1 > Συνολικός χρόνος 2 > Συνολικός χρόνος 3 > Συνολικός χρόνος 4