KIWI – ΠΟΛΥΝΗΜΑΤΙΚΗ ΛΕΙΤΟΥΡΓΙΑ

Δίνεται μηχανή αποθήκευσης (storage engine) υλοποιημένη στη γλώσσα C. Υλοποιήσαμε την πολυνηματική λειτουργία των εντολών put και get που παρείχε η μηχανή αποθήκευσης. Η υλοποίησή επιτρέπει πολλαπλά νήματα να καλούν τις εντολές put και get ταυτόχρονα. Η μηχανή αποθήκευσης εκτελεί τις ταυτόχρονες λειτουργίες σωστά και διατηρεί στατιστικά του χρόνου εκτέλεσης της κάθε λειτουργίας. Η υλοποίησή έγινε στην γλώσσα C και βασίζεται στην βιβλιοθήκη Pthreads του Linux.

Ο πηγαίος κώδικας που υλοποιεί την μηχανή αποθήκευσης Kiwi βασίζεται σε δέντρο log-structured merge (LSM-tree). Η προγραμματιστική διεπαφή (API) περιλαμβάνει λειτουργίες put και get για ζεύγη κλειδιού-τιμής. Η put δέχεται ως παράμετρο ζεύγος κλειδιού-τιμής που προστίθεται στην δομή. Η get δέχεται ως παράμετρο ένα κλειδί και αιτείται την αντίστοιχη τιμή εφόσον υπάρχει αποθηκευμένο στην δομή ζεύγος κλειδιού-τιμής με το συγκεκριμένο κλειδί, αλλιώς επιστρέφει σφάλμα αν το κλειδί δεν βρεθεί.

Α. ΠΟΛΥΝΗΜΑΤΙΚΗ ΥΛΟΠΟΙΗΣΗ ΣΕ KIWI

Σε 1ο στάδιο, θα εισάγουμε στον παρεχόμενο κώδικα της μηχανής,
τα απαραίτητα κομμάτια κώδικα που χρειάζονται, προκειμένου να εξασφαλίσουμε, πως:

- αφενός, το **σύστημα** θα **διαθέτει πολυνηματική λειτουργία** για τις **εντολές:**
  i. **put** (εισαγωγή)
  ii. **get** (αναζήτηση)
- αφετέρου, θα δίνει τη δυνατότητα σε **πολλαπλά νήματα** να **καλούν** αυτές τις **εντολές**.

Φυσικά, θα επιβεβαιώσουμε με κατάλληλο πειραματισμό, ότι οι αλλαγές που κάνουμε είναι επιτυχείς.

Επιπλέον, για την κάθε λειτουργία, θα διατηρούμε τα αντίστοιχα στατιστικά,
συμπεριλαμβανομένου και του χρόνου εκτέλεσης τους.

Αρχικά λοιπόν, θα κάνουμε τις κατάλληλες **τροποποιήσεις** στον **κώδικα** του **kiwi**.

Συγκεκριμένα δηλαδή, στον κώδικα υπό του **φακέλου kiwi-source**.

Οπότε, θα κάνουμε τις απαραίτητες **μετατροπές** στο αρχείο, ώστε οι **λειτουργίες**:
i. **db_add()**
ii. **db_get()**

να υποστηρίζουν τον **πολυνηματικό προγραμματισμό** που μας ζητήθηκε να υλοποιήσουμε.

Προτού περάσουμε στο γράψιμο του κώδικα, οφείλουμε να επισημάνουμε πως
αντιστοιχούμε το:

- **db_add()** ως **write operation,** δηλαδή λειτουργία **εγγραφής**
  Εισαγωγή ζεύγους κλειδιού- τιμής.
- **db_get()** ως **read operation**, δηλαδή λειτουργία **ανάγνωσης**.
  Αναζήτηση ζεύγους κλειδιού- τιμής.

ΑΡΧΕΙΑ ΓΙΑ ΤΡΟΠΟΠΟΙΗΣΗ ΤΟΥ 1^{ΟΥ} ΜΕΡΟΥΣ:

i. **module db** -> **db.h** και **db.c**

ii. **module lru** -> **lru.h** και **lru.c**

iii. **module sst** -> **sst.h** και **sst.c**

Τα αρχεία με **επέκταση:**

- **.h** είναι **αρχεία κεφαλίδας**.

  Περιέχουν τα **πρωτότυπα συναρτήσεων** και άλλες **δηλώσεις**.

  Τα αλλάζουν σε **δημόσια ορατά**, μέσω **συναρτήσεων διεπαφής**.

- **.c** είναι **αρχεία πηγαίου κώδικα**.

  Περιέχουν την **υλοποίηση** των **συναρτήσεων** των **.h** αρχείων, καθώς και **βοηθητικές**.

  Προκειμένου να έχουν **πρόσβαση** στις συναρτήσεις διεπαφής των **.h** αρχείων,
  πρέπει να περιέχουν την **εντολή #include**.

Η βασική **αλλαγή,** που πρέπει να γίνει σε όλες τις δομές που διαχειρίζονται
τις πολυνηματικές λειτουργίες, είναι η **προσθήκη** ενός **readers-writer lock.**

Ο βασικός **ρόλος** των **lock**, είναι πως επιτρέπει στο νέο αναγνώστη, να δει τι έγραψε ο προηγούμενος.

Ουσιαστικά δείχνουν ποια από τις 2 λειτουργίες έχει ενεργοποιηθεί.

Οπότε**, εισάγουμε** επανειλημμένως **lock,** για **πρόσβαση** στις **δομές**,
προκειμένου να γίνει **προσπέλαση** των **δεδομένων** της εκάστοτε δομής, για:

- **διάβασμα** ή
- **εγγραφή**.

Συγκεκριμένα, θα εισάγουμε **lock** στις **δομές**:

a. **DB**
b. **LRU**
c. **SST και SSTMetaData**

1. ΤΡΟΠΟΠΟΙΗΣΕΙΣ ΣΤΑ ΑΡΧΕΙΑ db.h και db.c

Αρχικά, περιλαμβάνουμε στην αρχή του αρχείου **db.h**, το **pthread.h**.

1η ΑΛΛΑΓΗ: Γράφουμε στο πεδίο ορισμού βιβλιοθηκών, **κώδικα εισαγωγής** εντός του αρχείο **db.h**:

```
4    #include <pthread.h>
```

Εισάγουμε στο κατάλληλο σημείο, εντός του **struct DB**, την δήλωση του **readers-writer lock** στο **db.h**.

2η ΑΛΛΑΓΗ: Γράφουμε στο **struct _db**, εντός του **db.h**, **κώδικα δήλωσης**:

pthread_**rwlock**_t **rwlock**;

```
17      pthread_rwlock_t rwlock;
```

```
12  typedef struct _db
13  {
14      char basedir[MAX_FILENAME+1];
15      SST* sst;
16      MemTable* memtable;
17      pthread_rwlock_t rwlock;
18  } DB;
```

Παρατήρηση: για την σωστή χρήση του **pthread_rwlock_t**, θέτουμε ως **standard** το **gnu99,** αντί του **std99,** καθορισμένο στο αρχείο **defs.mk**.

Το **rwlock** που μόλις ορίσαμε, το:

- **αρχικοποιούμε** εντός της **db_open()**
- **αποδεσμεύουμε** εντός της **db_close()**

3η ΑΛΛΑΓΗ: Γράφουμε στην **db_open_ex()** εντός του **db.c**, τον κώδικα **αρχικοποίησης**:

```
if (pthread_rwlock_init(&self->rwlock, NULL))
{
        PANIC("pthread_rwlock_init() failed!");
}
```

```
15      if (pthread_rwlock_init(&self->rwlock, NULL))
16      {
17          PANIC("pthread_rwlock_init() failed!");
18      }
```

4η ΑΛΛΑΓΗ: Γράφουμε στην **db_close()** εντός του **db.c**, τον κώδικα **αποδέσμευσης**:

```
if (pthread_rwlock_destroy(&self->rwlock))
{
        PANIC("pthread_rwlock_destroy() failed!");
}
```

```
70      if (pthread_rwlock_destroy(&self->rwlock))
71      {
72          PANIC("pthread_rwlock_destroy() failed!");
73      }
74      free(self);
```

Παρατήρηση: η μέθοδος **PANIC**, την οποία θα χρησιμοποιήσουμε επανειλημμένως, ενημερώνει το σύστημα, σε περίπτωση σφάλματος, κατά την εκάστοτε τρέχουσα διαδικασία.

Αφότου προσθέσαμε τον παραπάνω κώδικα, το **db_add()** έχει αναλάβει επιτυχώς ρόλο **γραφέα**.

Άρα, τώρα, θα προσαρμόσουμε την συνάρτηση **db_add()**, εντός του αρχείου **db.c**, προκειμένου, να λαμβάνουμε το **write lock** στην **αρχή**.

5η ΑΛΛΑΓΗ: Γράφουμε στην **db_open_ex()**, εντός του **db.c**, τον κώδικα **λήψης** του **write lock**:

```
if (pthread_rwlock_wrlock(&self->rwlock))
{
        PANIC("pthread_rwlock_wrlock() failed!");
}
```

```
20      if (pthread_rwlock_wrlock(&self->rwlock))
21      {
22          PANIC("pthread_rwlock_wrlock() failed!");
23      }
```

Αρχικοποιούμε με το rwlock κλειδωμένο.

Στη συνέχεια, θα **ξεκλειδώσουμε** το **lock**, που λάβαμε, σε όλα τα μονοπάτια εξόδου.

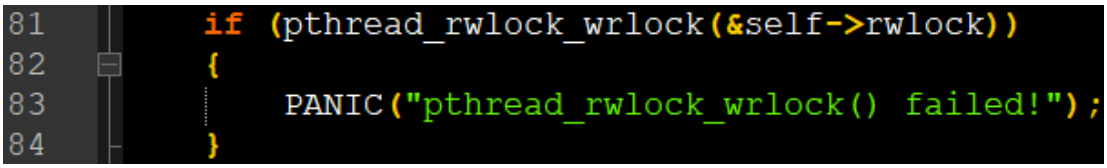
6η ΑΛΛΑΓΗ: Γράφουμε στην **db_close()** εντός του **db.c**, τον κώδικα **λήψης** του **write lock**:

```
if (pthread_rwlock_wrlock(&self->rwlock))
{
        PANIC("pthread_rwlock_wrlock() failed!");
}
```

```
48      if (pthread_rwlock_wrlock(&self->rwlock))
49      {
50          PANIC("pthread_rwlock_wrlock() failed!");
51      }
```

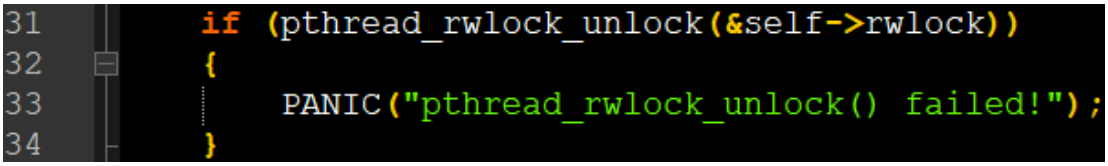
7η ΑΛΛΑΓΗ: Γράφουμε στην **db_add()** εντός του **db.c**, τον κώδικα **λήψης** του **write lock**:

```
if (pthread_rwlock_wrlock(&self->rwlock))
{
        PANIC("pthread_rwlock_wrlock() failed!");
}
```



8η ΑΛΛΑΓΗ: Γράφουμε στην **db_open_ex()** εντός του **db.c**, τον κώδικα **ξεκλειδώματος** του **lock**:

```
if (pthread_rwlock_unlock(&self->rwlock))
{
        PANIC("pthread_rwlock_unlock() failed!");
}
```



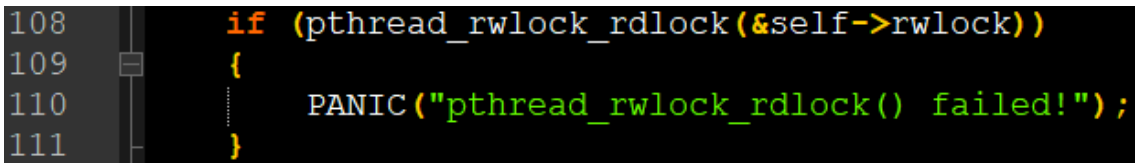
Ομοίως με το **db_add()**, και το **db_get()** έχει αναλάβει τώρα επιτυχώς ρόλο **αναγνώστη**.

Συνεπώς θα ακολουθήσουμε παρόμοια διαδικασία, εφαρμόζοντας τις απαραίτητες αλλαγές.

Άρα, τώρα, θα προσαρμόσουμε την συνάρτηση **db_get()**, εντός του αρχείου **db.c**, προκειμένου, να λαμβάνουμε το **read lock** στην **αρχή**.

9η ΑΛΛΑΓΗ: Γράφουμε στην **db_get()** εντός του **db.c**, τον κώδικα **λήψης** του **read lock**:

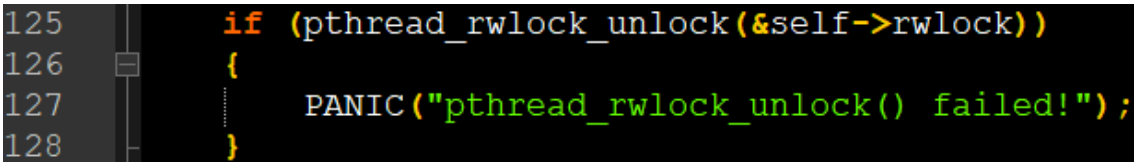
```
if (pthread_rwlock_rdlock(&self->rwlock))
{
        PANIC("pthread_rwlock_rdlock() failed!");
}
```



Στη συνέχεια, θα **ξεκλειδώσουμε** το **lock**, που λάβαμε, σε όλα τα μονοπάτια εξόδου.

10η ΑΛΛΑΓΗ: Γράφουμε στην **db_get()** εντός του **db.c**, τον κώδικα **ξεκλειδώματος** του **lock**:

```
if (pthread_rwlock_unlock(&self->rwlock))
{
        PANIC("pthread_rwlock_unlock() failed!");
}
```



2. ΤΡΟΠΟΠΟΙΗΣΕΙΣ ΣΤΑ ΑΡΧΕΙΑ sst.h και sst.c

Ενδέχεται να μην εντοπιστεί επιτυχώς το **κλειδί** στο **in-memory data structure**.

Αν συμβεί αυτό, θα χρειαστεί μέσω της **db_get()** να το **αναζητήσουμε** εντός του **SST**.

Επιπλέον, είναι απαραίτητο να εκτελέσουμε αναζήτηση και εντός του **temporary memtable**, καθώς ενδέχεται το κλειδί να μεταβιβάστηκε στο **background merger thread**, προκειμένου να το **καταγράψει** (**merge**) στον δίσκο, μαζί με την υπόλοιπη **SST δομή**.

Συνεπώς, πρέπει να εξασφαλίσουμε πως είναι συγχρονισμένοι οι **readers**, εντός των:

- **sst_get()**
- **background merger thread**

Γι' αυτό τον σκοπό, προσθέσαμε έναν **επιπλέον readers-writer lock**.

11η ΑΛΛΑΓΗ: Γράφουμε στην **struct sst** εντός του **sst.h** τον κώδικα **δήλωσης** του **επιπλέον lock**:

pthread_**rwlock**_t **rwlock**.

```
72      pthread_rwlock_t rwlock;
```

12η ΑΛΛΑΓΗ: Γράφουμε στη **sst_new()** εντός του **sst.c** τον κώδικα **αρχικοποίησης** του **επιπλέον lock**:

```
if (pthread_rwlock_init(&self->rwlock, NULL))
{
        PANIC("pthread_rwlock_init() failed!");
}
```

```
385     if (pthread_rwlock_init(&self->rwlock, NULL))
386     {
387         PANIC("pthread_rwlock_init() failed!");
388     }
```

13η ΑΛΛΑΓΗ: Γράφουμε στην **sst_free()** εντός του **sst.c** τον κώδικα **αποδέσμευσης** του **επιπλέον lock**:

```
if (pthread_rwlock_destroy(&self->rwlock))
{
        PANIC("pthread_rwlock_destroy() failed!");
}
```

```
481     if (pthread_rwlock_destroy(&self->rwlock))
482     {
483         PANIC("pthread_rwlock_destroy() failed!");
484     }
```

Αφότου προσθέσαμε τον παραπάνω κώδικα, το **sst_get()** έχει αναλάβει επιτυχώς ρόλο **αναγνώστη**.

14η ΑΛΛΑΓΗ: Γράφουμε στην **sst_get()** εντός του **sst.c** τον κώδικα **λήψης** του **read lock**:

```
if (pthread_rwlock_rdlock(&self->rwlock))
{
        PANIC("pthread_rwlock_rdlock() failed!");
}
```

```
720     if (pthread_rwlock_rdlock(&self->rwlock))
721     {
722         PANIC("pthread_rwlock_rdlock() failed!");
723     }
```

15η ΑΛΛΑΓΗ: Γράφουμε στην **sst_get()** εντός του **sst.c** τον κώδικα **ξεκλειδώματος** του **read lock**

(στα 3 μονοπάτια εξόδου):

```
if (pthread_rwlock_unlock(&self->rwlock))
{
        PANIC("pthread_rwlock_unlock() failed!");
}
```

```
737     if (pthread_rwlock_unlock(&self->rwlock))
738     {
739         PANIC("pthread_rwlock_unlock() failed!");
740     }
```

```
807         if (pthread_rwlock_unlock(&self->rwlock))
808         {
809             PANIC("pthread_rwlock_unlock() failed!");
810         }
```

```
819     if (pthread_rwlock_unlock(&self->rwlock))
820     {
821         PANIC("pthread_rwlock_unlock() failed!");
822     }
```

Θα ακολουθήσουμε παρόμοια διαδικασία με το **sst_get()**, για το **background merger thread**, εφαρμόζοντας τις απαραίτητες αλλαγές.

Η ειδοποιός διαφορά είναι ότι **background merger thread** αποτελεί **γραφέα**, το οποίο συνεπάγεται πως προκειμένου να **προσπελάσει** επιτυχώς την **immutable list**, χρειάζεται να λάβει, εκ των προτέρων, το **write lock** στο **rwlock**.

16η ΑΛΛΑΓΗ: Γράφουμε στην **merge_thread()** εντός του **sst.c** τον κώδικα **λήψης** του **write lock**:

```
if (pthread_rwlock_wrlock(&sst->rwlock))
{
        PANIC("pthread_rwlock_wrlock() failed!");
}
```

```
165     if (pthread_rwlock_wrlock(&sst->rwlock))
166     {
167         PANIC("pthread_rwlock_wrlock() failed!");
168     }
```

17η ΑΛΛΑΓΗ: Γράφουμε στην **merge_thread()** εντός του **sst.c** τον κώδικα **ξεκλειδώματος** του **write lock**:

```
if (pthread_rwlock_unlock(&sst->rwlock))
{
        PANIC("pthread_rwlock_unlock() failed!");
}
```

```
216     if (pthread_rwlock_unlock(&sst->rwlock))
217     {
218         PANIC("pthread_rwlock_unlock() failed!");
219     }
```

Η συνάρτηση **sst_merge**, είναι υπεύθυνη για τον ορισμό του **immutable list**.

Όμοια με την **background merger thread**, αποτελεί **γραφέα**.

Γεγονός που συνεπάγεται ότι λαμβάνει ομοίως το **write lock** από το **rwlock**
και έπειτα το **ξεκλειδώνει**.

18η ΑΛΛΑΓΗ: Γράφουμε στην **sst_merge()** εντός του **sst.c** τον κώδικα **λήψης** του **write lock**:

```
if (pthread_rwlock_wrlock(&self->rwlock))
{
        PANIC("pthread_rwlock_wrlock() failed!");
}
```

```
648     if (pthread_rwlock_wrlock(&self->rwlock))
649     {
650         PANIC("pthread_rwlock_rdlock() failed!");
651     }
```

19η ΑΛΛΑΓΗ: Γράφουμε στην **sst_merge()** εντός του **sst.c** τον κώδικα **ξεκλειδώματος** του **write lock**:

```
if (pthread_rwlock_unlock(&sst->rwlock))
{
        PANIC("pthread_rwlock_unlock() failed!");
}
```

```
659     if (pthread_rwlock_unlock(&self->rwlock))
660     {
661         PANIC("pthread_rwlock_unlock() failed!");
662     }
```

Τώρα επανερχόμαστε στην μέθοδο **sst_get()**, όπου εντοπίζουμε ένα πιθανό **πρόβλημα**.

Καθώς ενδέχεται εντός της **sst_get()**,
να συνυπάρχουν **παραπάνω από ένας reader**, εξαιτίας της εντολής **self->targets**,
όλοι τους τροποποιούν μια συγκεκριμένη μεταβλητή.

Προκειμένου να αποτρέψουμε αυτό το λανθασμένο ενδεχόμενο,
θα εισάγουμε εντός της μεθόδου **sst_get()** μια τοπική μεταβλητή,
που θα την ονομάσουμε **targets**.

Με αυτό τον τρόπο, θα δημιουργεί για τον **κάθε reader** μια προσωπική **μεταβλητή**.
Άρα δεν θα υπάρχει πιθανότητα να αναμειχθούν οι μεταβλητές του καθενός.

20η ΑΛΛΑΓΗ: Γράφουμε στην **sst_get()** εντός του **sst.c** τον κώδικα **δήλωσης-αρχικοποίησης** του **targets**:

Vector ***targets** = vector_new();

```
718     Vector *targets = vector_new();
```

vector_clear(**targets**);

```
746     vector_clear(targets);
```

Γράφουμε εντός του **sst.c** τον κώδικα **αποδέσμευσης** του **targets**, σε κάθε μονοπάτι τερματισμού.

21η ΑΛΛΑΓΗ: Γράφουμε στην **sst_get()** εντός του **sst.c** τον κώδικα **αποδέσμευσης** του **targets**:

vector_**free**(**targets**);

```
733     if (ret)
734     {
735         vector_free(targets);
736
737         if (pthread_rwlock_unlock(&self->rwlock))
738         {
739             PANIC("pthread_rwlock_unlock() failed!");
740         }
741         return ret;
742     }
```

```
806         vector_free(targets);
807         if (pthread_rwlock_unlock(&self->rwlock))
808         {
809             PANIC("pthread_rwlock_unlock() failed!");
810         }
811         return opt == ADD;
```

```
818     vector_free(targets);
819     if (pthread_rwlock_unlock(&self->rwlock))
820     {
821         PANIC("pthread_rwlock_unlock() failed!");
822     }
823     return 0;
```

Τέλος, εντοπίσαμε ένα ακόμη πρόβλημα εντός της **sst_get()**.

Στο τελικό **for** της μεθόδου, μετά που έχει επιλεχθεί το **SSTMetadata**, δηλαδή όταν:

SSTMetadata* **target** = (**SSTMetadata** *)vector_get(**targets**, i);

```
782         OPT opt;
783
784         SSTMetadata* target = (SSTMetadata *)vector_get(targets, i);
```

Παρατηρούμε πως απευθείας αλλάζει στο σημείο: if (--**target**->allowed_seeks <= 0)

```
791         if (--target->allowed_seeks <= 0)
792         {
793             _schedule_compaction(self);
794             target->allowed_seeks = target->filesize / 16384;
795         }
```

Οπότε ενδέχεται εντός της **sst_get()**, να συνυπάρχουν **παραπάνω από ένα νήμα αναγνώστη**,
ενώ αναμέναμε να έχουμε κάποιο μοναδικό για το **SSTMetadata**.

Προκειμένου να αποτρέψουμε αυτό το λανθασμένο ενδεχόμενο,
θα ορίσουμε **lock** εντός του **struct SSTMetadata.**

22η ΑΛΛΑΓΗ: Γράφουμε στο **SSTMetadata** εντός του **sst.h** τον κώδικα **δήλωσης** του **lock**:

pthread_mutex_t **lock**

```
37      pthread_mutex_t lock;
```

23η ΑΛΛΑΓΗ: Γράφουμε στην **sst_metadata_new()** εντός του **sst.c** τον κώδικα **αρχικοποίησης** του **lock**:

```
if (pthread_mutex_init(&self->lock, NULL))
{
        PANIC("pthread_mutex_init() failed!");
}
```

```
830     if (pthread_mutex_init(&self->lock, NULL))
831     {
832         PANIC("pthread_mutex_init() failed!");
833     }
```

24η ΑΛΛΑΓΗ: Γράφουμε στην **sst_metadata_free()** εντός του **sst.c** τον κώδικα **αποδέσμευσης** του **lock**:

```
if (pthread_mutex_destroy(&self->lock))
{
        PANIC("pthread_mutex_destroy() failed!");
}
```

```
868     if (pthread_mutex_destroy(&self->lock))
869     {
870         PANIC("pthread_mutex_destroy() failed!");
871     }
```

25η ΑΛΛΑΓΗ: Γράφουμε στην **sst_get()** εντός του **sst.c** τον κώδικα **εναλλαγής από lock σε unlock**, όποτε αλλάζει το **SSTMetadata target**:

```
if (pthread_mutex_lock(&target->lock))
{
        PANIC("pthread_mutex_lock() failed!");
}
```

```
786         if (pthread_mutex_lock(&target->lock))
787         {
788             PANIC("pthread_mutex_lock() failed!");
789         }
```

26η ΑΛΛΑΓΗ: Γράφουμε στην **sst_get()** εντός του **sst.c** τον κώδικα **εναλλαγής από lock σε unlock**, όποτε αλλάζει το **SSTMetadata target**:

```
if (pthread_mutex_unlock(&target->lock))
{
        PANIC("pthread_mutex_unlock() failed!");
}
```

```
797         if (pthread_mutex_unlock(&target->lock))
798         {
799             PANIC("pthread_mutex_unlock() failed!");
800         }
```

Στην συνάρτηση **evaluate_compaction** αλλάζουν τα:

- **self->comp_score**
- **self->comp_level**.

Άρα θα χρειαστεί πάλι να προσθέσω μια **write lock** εντός της **SST**, όπως δείξαμε προηγουμένως.

Μολαταύτα, οι συνθήκες εδώ διαφέρουν, καθώς θα λάβω το **read lock**
κατά την κλήση της **sst_get**, από την παρούσα συνάρτηση.

Όμως αυτό συνεπάγεται πως **δεν** μπορώ να το **εναλλάξω** σε **write lock**.

Συνεπώς, θα εισάγω **write lock**, εντός του struct **SST**, που θα χρησιμοποιείται μόνο από τις:

- **comp_score**
- **comp_level**

27η ΑΛΛΑΓΗ: Γράφω στο struct **SST** εντός του **sst.h** τον κώδικα **δήλωσης** του **write lock**:

```
pthread_mutex_t comp_lock;
```

```
47      pthread_mutex_t comp_lock;
```

28η ΑΛΛΑΓΗ: Γράφω στην **evaluate_compaction** εντός του **sst.c** τον κώδικα **αρχικοποίησης** του **lock**:

```
if (pthread_mutex_lock(&self->comp_lock))
{
        PANIC("pthread_mutex_lock() failed!");
}
```

```
83      if (pthread_mutex_lock(&self->comp_lock))
84      {
85          PANIC("pthread_mutex_lock() failed!");
86      }
```

29η ΑΛΛΑΓΗ: Γράφω στην **evaluate_compaction** εντός του **sst.c** τον κώδικα **κλειδώματος** του **lock**:

```
if (pthread_mutex_unlock(&self->comp_lock))
{
        PANIC("pthread_mutex_unlock() failed!");
}
```

```
109     if (pthread_mutex_unlock(&self->comp_lock))
110     {
111         PANIC("pthread_mutex_unlock() failed!");
112     }
```

3. ΤΡΟΠΟΠΟΙΗΣΕΙΣ ΣΤΑ ΑΡΧΕΙΑ lru.h και lru.c

Εντός της **lru (lru.h/lru.c)** εντοπίζουμε μια λανθασμένη διαχείριση.

Ενώ ο καθορισμένος ρόλος της **cache** είναι αποκλειστικά η **ταχύτερη ανάγνωση**,
παρατηρούμε ότι εκτελεί παραπανίσιες λειτουργίες.

Για παράδειγμα, η **lru_get() τροποποιεί** την δομή της **lru**,
ώστε να **προσθέτει** το **κλειδί** που δόθηκε μετά το κάλεσμα της.

Προκειμένου να επιλύσουμε αυτό το θέμα, προσθέσαμε μια **write lock** εντός του **lru**.

30η ΑΛΛΑΓΗ: Γράφουμε στο **struct _lru** εντός του **lru.h** τον κώδικα **δήλωσης** του **lock:**

pthread_mutex_t **lock**.

```
37      pthread_mutex_t lock;
```

31η ΑΛΛΑΓΗ: Γράφουμε στην **lru_new** εντός του **lru.c** τον κώδικα **αρχικοποίησης** του **write lock:**

```
if (pthread_mutex_init(&self->lock, NULL))
{
        PANIC("pthread_mutex_init() failed!");
}
```

```
11      if (pthread_mutex_init(&self->lock, NULL))
12      {
13          PANIC("pthread_mutex_init() failed!");
14      }
```

32η ΑΛΛΑΓΗ: Γράφουμε στην **lru_free** εντός του **lru.c** τον κώδικα **αποδέσμευσης** του **write lock:**

```
if (pthread_mutex_destroy(&self->lock))
{
        PANIC("pthread_mutex_destroy() failed!");
}
```

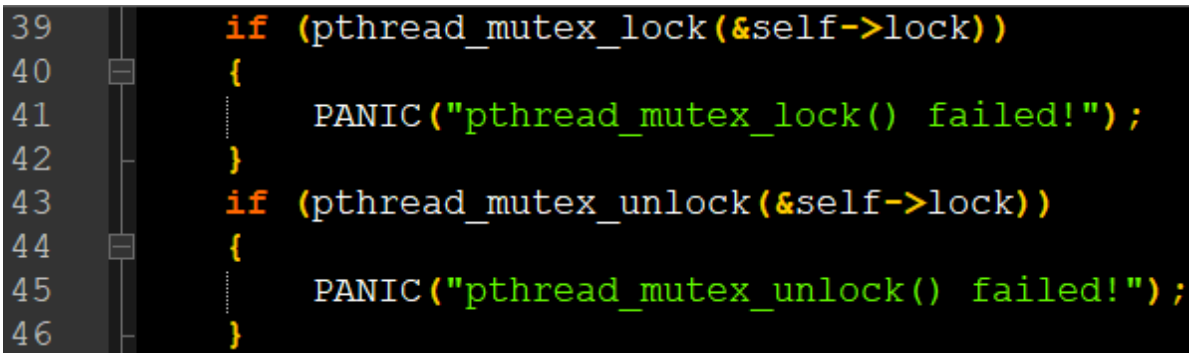
```
55      if (pthread_mutex_destroy(&self->lock))
56      {
57          PANIC("pthread_mutex_destroy() failed!");
58      }
```

Προκειμένου να ολοκληρωθεί επιτυχώς η διαδικασία της εκάστοτε **αποδέσμευσης**,
χρειάζεται να **προσδιοριστεί** αρχικά το **lock**,
ώστε να **οριστούν** οι **μεταβλητές** της **LRU**.

Η **διαδικασία λήψης** του **lock** θα γίνει μονάχα **μια φορά**,
καθώς επαρκεί για να καθοριστούν τα απαραίτητα δεδομένα.

33η ΑΛΛΑΓΗ: Γράφουμε στην **lru_free** εντός του **lru.c** κώδικα **εναλλαγής από lock σε unlock**:

```
if (pthread_mutex_lock(&self->lock))
{
        PANIC("pthread_mutex_lock() failed!");
}
if (pthread_mutex_unlock(&self->lock))
{
        PANIC("pthread_mutex_unlock() failed!");
}
```



Β. Υλοποίηση Benchmark

Σε 2ο στάδιο, τροποποιήσαμε, εντός του φακέλου **bench**, τα αρχεία **bench.c, bench.h** και **kiwi.c**.

- Πρώτα θα σχολιάσουμε τη λειτουργία του **benchmark**.
  - Αρχικά ορίζουμε τις απαραίτητες μεταβλητές:
    i. **Ορίζουμε** στο **main thread,** το **πλήθος** των **benchmark threads,** που θα προσθέσουμε.
      - Καθένα από αυτά, θα εκτελέσει μια από τις μεθόδους **add()** και **get()**.
    ii. **Ορίζουμε** το **πλήθος**, των εκτελέσιμων **operation,** του **καθενός**.
    iii. **Ορίζουμε** την **πιθανότητα,** εκτέλεσης της **add() operation**, (αντί **get()**), του **καθενός**.
      - Η πιθανότητα αυτή, συμβάλλει:
        - στην **ταυτοποίηση** του καθενός
        - στην διαχείριση ξεχωριστών **workloads**.
  - Στη συνέχεια συλλέγουμε τα δεδομένα:
    I. Το **main thread** στέλνει το σήμα εκκίνησης των **benchmark threads**.
    II. Κάθε **benchmark thread**, εκτελεί το **πλήθος** των **operation,** που έλαβε από το σήμα.
      - Καθένα τους, διατηρεί τα δεδομένα, της κάθε εκτέλεσης του.
    III. Αφότου τελειώσουν όλες οι εκτελέσεις, το **main thread καταγράφει** τα **δεδομένα**, που έλαβε από τα νήματα, και τα ενώνει.

- Τώρα θα αναλύσουμε το καθένα από τα παραπάνω στάδια.
  - Τα **benchmark threads** βρίσκονται εντός του **kiwi.c**.
  - Η **main thread** χρησιμοποιεί το **πρότυπο** της **συνάρτησης** τους, το οποίο βρίσκεται στο **bench.h**:

    void *benchmark_thread(void *thread_**parameters**)
  - Τα δεδομένα της κάθε **operation,** διατηρούνται στην **δομή, struct statistics**.
    Την ορίζουμε στο **bench.h**,
      - ώστε να είναι προσβάσιμη από τα **benchmark** και την **main**.

Η **δομή περιέχει struct statistics:**

- **counters** για κάθε **operations**
- **συνολικούς χρόνους**
- **throughput**.

- Κατά την δημιουργία των **benchmark threads**, από την **main thread,** τους εισάγουμε μια **δομή καθοδήγησης**, ως παράμετρο.

  Την ονομάζουμε **struct bench_parameters,** και την ορίζουμε εντός του **bench.h**.

34η ΑΛΛΑΓΗ: Γράφουμε εντός του **bench.c** κώδικα **δήλωσης struct bench_parameters**:

```
38  // Parameteroi toy benchmark gia thn ektelesh apo to thread
39
40  struct bench_parameters
41  {
42      unsigned int seed;              // seed gia tyxaio ariumo piuanothtvn
43      unsigned int random_key_seed;   // seed gia ta tyxaia kleidia
44      long int count;                 // plhuos ektelesimvn leityrgivn
45      double prob_write;              // piuanothta na einai leitoyrgia eggrafhs
46      struct statistics stats;        // xvros apouhkeyshs tvn statistikvn tvn thread
47      DB *db;                         // to database pros xrhsh
48
49      pthread_cond_t *cv;             // metablhth synuhkhs kai mutex gia to main thread
50      pthread_mutex_t *mutex;
51      int *start;                     // orizetai s alhuew gia ekkinhsh ths leitoyrgias
52  };
```

Παρακάτω αναλύουμε αναλυτικά τα εξής, εντός των ανάλογων κατηγοριών:

- **unsigned int seed**
- **unsigned int random_key_seed**
- **pthread_cond_t *cv;**
- **pthread_mutex_t *mutex;**
- **int *start;**

1) ΕΠΕΞΗΓΗΜΑΤΙΚΗ ΑΝΑΛΥΣΗ ΓΙΑ struct bench_parameters

Μεταβλητές:

- **long int count**
  το **πλήθος operation** που λαμβάνει το εκάστοτε **benchmark thread**,
  από την **main thread,** η οποία το λαμβάνει από την γραμμή εντολών.

35η ΑΛΛΑΓΗ: Γράφουμε στην **lru_free** εντός του **bench.c** κώδικα **δήλωσης count**:

count = atoi(argv[2]);

```
122     count = atoi(argv[2]);
```

- **double prob_write**
  η **πιθανότητα,** εκτέλεσης της **add()** (αντί της **get()**), του εκάστοτε **benchmark thread**,
  την οποία λαμβάνει η **main thread**, από την γραμμή εντολών.

36η ΑΛΛΑΓΗ: Γράφουμε στην **struct bench_parametrs** εντός του **bench.h** κώδικα **δήλωσης prob_write**:

double prob_write;

```
45      double prob_write;
```

- **DB *db**
  η **βάση** που χρειάζονται τα **benchmark threads,** για **εκτέλεση των λειτουργιών**.

37η ΑΛΛΑΓΗ: Γράφουμε στην **struct bench_parametrs** εντός του **bench.h** κώδικα **δήλωσης *db**:

DB *db;

```
47      DB *db;
```

- **struct statistics stats**
  η **δομή** διατήρησης των δεδομένων της κάθε **λειτουργίας.**

38η ΑΛΛΑΓΗ: Γράφουμε στην **struct bench_parametrs** εντός του **bench.h** κώδικα **δήλωσης stats:**

struct statistics stats;

```
46      struct statistics stats;
```

Αφότου τελειώσουν όλες οι εκτελέσεις, το **main** έχει πρόσβαση στη δομή,
ώστε να λάβει τα δεδομένα της κάθε **benchmark thread**.

Με αυτό τον τρόπο διαχείρισης των δεδομένων, αποφεύγουμε την χρήση **locks**.

1) ΕΠΕΞΗΓΗΜΑΤΙΚΗ ΑΝΑΛΥΣΗ ΓΙΑ random numbers

Θα χρειαστεί να διαχειριστούμε **τυχαίους αριθμούς** 2 φορές, ενόσω τρέχει το **benchmark**. Για:

i. τον ορισμό της **πιθανότητας** εκτέλεσης της **add()** (αντί της **get()**), του κάθε **thread**.

ii. τα **τυχαία κλειδιά** που θα χρειαστούν οι **add**() και **get**().

Η **διαχείριση** των **random numbers** στο **benchmark** επηρεάζεται από **2 θέματα**:

i. Καθώς η **rand**() **δεν** είναι **thread-safe**, θα αξιοποιήσω την **rand_r()**.

Συνεπώς θα τροποποιήσουμε την **_random_key**(),
ώστε να **λαμβάνει** την μεταβλητή **seed**,
και να της **δίνει μοναδική τιμή** σε κάθε κλήση της συνάρτησης.

39η ΑΛΛΑΓΗ: Γράφουμε στην **_random_key** εντός του **bench.c** κώδικα **δήλωσης key**:

key[i]=salt[rand_r(seed)%36];

```
18          key[i] = salt[rand_r(seed) % 36];
```

ii. Είναι απαραίτητο να είναι δυνατή η **αναπαραγωγή** του **benchmark**.

Καθώς θα χρειαστεί να **επαναλάβουμε** τον πειραματισμό με το **ίδιο workload**,

προκειμένου να προσδιορίσουμε προς τα **που τείνουν** να κατευθύνονται οι **τιμές**,

ανάλογα με τις **τιμές** που θα **εισάγουμε** στην γραμμή εντολών κατά την εκκίνηση του,

είναι απαραίτητο να **διατηρήσουμε** τη **σειρά δημιουργίας** των **τυχαίων τιμών**.

Οπότε, θα εισάγουμε στην **γραμμή εντολών** του **benchmark**,
το **αρχικό seed** που χρησιμοποιεί το **main thread**,
ώστε να υπολογίσει τα υπόλοιπα, που θα μοιράσει στα μεμονωμένα **benchmark**.

40η ΑΛΛΑΓΗ: Γράφουμε στην **_random_key** εντός του **bench.c** κώδικα **δήλωσης key**:

seed = atoi(argv[1]);

```
120         seed = atoi(argv[1]);
```

2) ΕΠΕΞΗΓΗΜΑΤΙΚΗ ΑΝΑΛΥΣΗ ΓΙΑ conditions

Προκειμένου το πείραμα να λειτουργήσει με **όλα** τα **benchmark threads** (σχεδόν) **ταυτόχρονα**, **ορίσαμε** τα **condition variables**.

Θα αναλάβουν τον ρόλο ειδοποίησης, που θα οδηγήσει σε έναρξη των **benchmark threads**.

Τα δηλώνουμε και τα αρχικοποιούμε εντός του **main thread**.

Στα **benchmark threads** τα εισάγουμε ως **pointer** (δηλαδή με **&**) **παραμέτρους**,
για να εξασφαλίσουμε πως σε περίπτωση που τροποποιηθεί η τιμή εντός μιας μεθόδου,
θα διατηρηθούν οι αλλαγές που του έγιναν, στις άλλες μεθόδους που θα την χρησιμοποιήσουν.

Στην **main thread** εντός του **bench.c** υπάρχει ο κώδικας **δήλωσης** των **condition variables**:

```
pthread_cond_t bench_start_cond;

pthread_mutex_t bench_start_mutex;

int start_cond = 0;
```

```
94      pthread_cond_t bench_start_cond;
95      pthread_mutex_t bench_start_mutex;
96      int start_cond = 0;
97      pthread_t *bench_threads;
```

Η μεταβλητή **start_cond**:

- **Αρχικοποιείται** στο **0**, ώστε
- Όταν γίνει **1**, να δοθεί η ειδοποίηση για εκκίνηση των **benchmark threads**.

Η **mutex**: Εξασφαλίζει ότι θα έχουμε αποκλειστική πρόσβαση στην **start_cond**.

Προκειμένου να αποφύγουμε τον αναγκαίο επαναλαμβανόμενο έλεγχο της,
θα ορίσουμε μια **condition variable**, η οποία θα ενημερώνει για την διαθεσιμότητα της.

41η ΑΛΛΑΓΗ: Γράφουμε στην **main thread** εντός του **bench.c** τον κώδικα **αρχικοποίησης** των **condition**:

```
if (pthread_mutex_init(&bench_start_mutex, NULL) != 0)
{
        perror("pthread_mutex_init() error");
        exit(1);
}
```

```
154     if (pthread_mutex_init(&bench_start_mutex, NULL) != 0)
155     {
156         perror("pthread_mutex_init() error");
157         exit(1);
158     }
```

42η ΑΛΛΑΓΗ: Γράφουμε στην **main thread** εντός του **bench.c** τον κώδικα **αρχικοποίησης** των **condition**:

```
if (pthread_cond_init(&bench_start_cond, NULL) != 0)
{
        perror("pthread_cond_init() error");
        exit(1);
}
```

```
160     if (pthread_cond_init(&bench_start_cond, NULL) != 0)
161     {
162         perror("pthread_cond_init() error");
163         exit(1);
164     }
```

Εφόσον προκύψει πρόβλημα, **ενημερώνουμε** τον χρήστη, και **τερματίζουμε**.

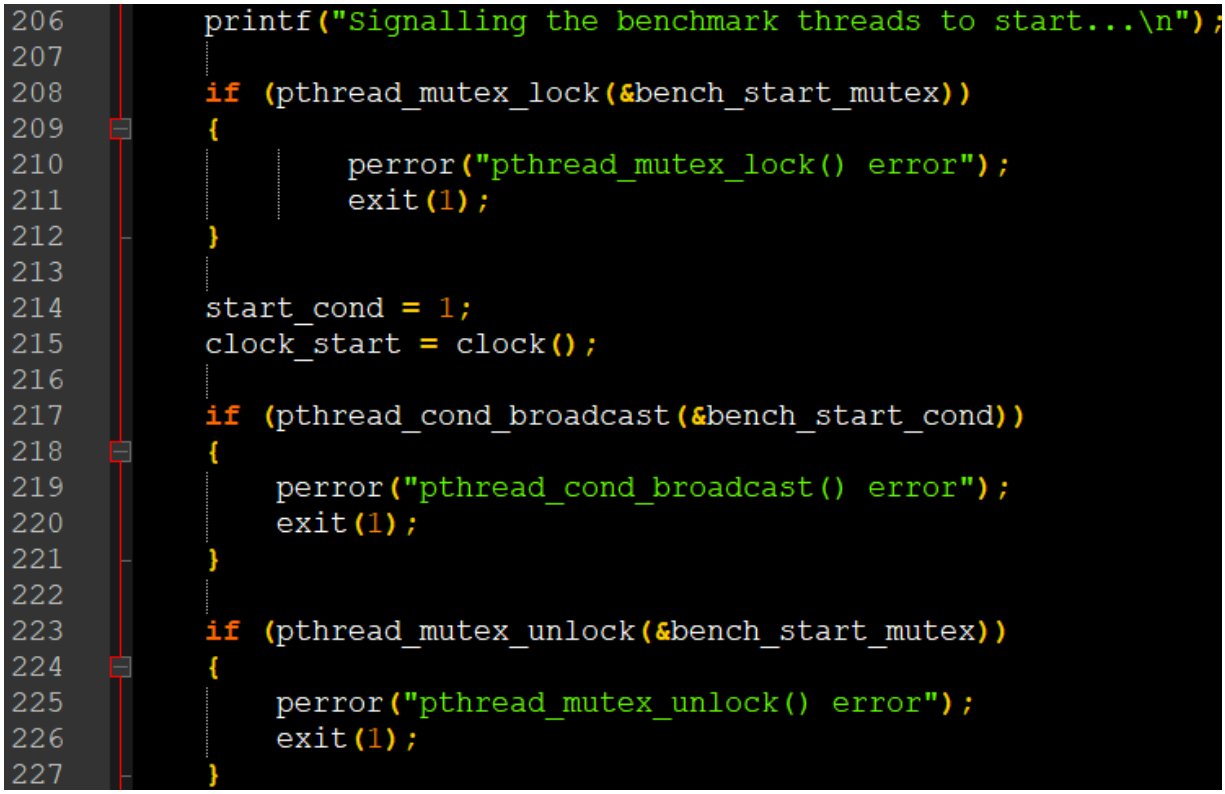
43η ΑΛΛΑΓΗ: Γράφουμε στη **main** εντός του **bench.c** τον κώδικα **ορισμού ως παραμέτρους**:

```
parameters[i].start = &start_cond;

parameters[i].cv = &bench_start_cond;

parameters[i].mutex = &bench_start_mutex;
```

```
166     printf("Initializing parameters for the benchmark threads...\n");
167
168     for (i = 0; i < threads_count; ++i)
169     {
170         parameters[i].seed = (unsigned) rand_r(&seed);
171         parameters[i].random_key_seed = (unsigned) rand_r(&seed);
172         parameters[i].count = count;
173         parameters[i].prob_write = atof(argv[4 + i]);
174         printf("\t...Benchmark thread %d has probability of writing %f\n", i, parameters[i].prob_write);
175         parameters[i].db = db;
176         parameters[i].start = &start_cond;
177         parameters[i].cv = &bench_start_cond;
178         parameters[i].mutex = &bench_start_mutex;
179     }
```

44η ΑΛΛΑΓΗ: Γράφουμε στη **main** εντός του **bench.c** κώδικα **απελευθέρωσης** και **μετάδοσης** για **ενημέρωση** του **mutex**:

```
if (pthread_mutex_lock(&bench_start_mutex))
{
        perror("pthread_mutex_lock() error");
        exit(1);
}
start_cond = 1;
if (pthread_cond_broadcast(&bench_start_cond))
{
        perror("pthread_cond_broadcast() error");
        exit(1);
}
```



```
206     printf("Signalling the benchmark threads to start...\n");
207
208     if (pthread_mutex_lock(&bench_start_mutex))
209     {
210             perror("pthread_mutex_lock() error");
211             exit(1);
212     }
213
214     start_cond = 1;
215     clock_start = clock();
216
217     if (pthread_cond_broadcast(&bench_start_cond))
218     {
219         perror("pthread_cond_broadcast() error");
220         exit(1);
221     }
222
223     if (pthread_mutex_unlock(&bench_start_mutex))
224     {
225         perror("pthread_mutex_unlock() error");
226         exit(1);
227     }
```

Μεταβάλλουμε την **start_cond** στο **1**, ώστε να **ενημερωθούν** τα **benchmark threads**.

Ενόσω η **start_cond** παραμένει **0**, το **benchmark thread** την ανανεώνει, ώσπου να γίνει 1.

Καθώς η **ανάγνωση** εκτελείται όσο το **mutex** παραμένει **κρατημένο**,
βάζουμε κάθε **benchmark thread** να **αναμένει** το **condition variable**.

Όταν **ειδοποιηθεί**, ελέγχει το **start_cond**, ώστε να διαπιστώσει αν εναλλάχτηκε σε **1**.

Εφόσον δεν υπάρχουν άλλοι παράγοντες, θα ειδοποιηθεί μονάχα 1 φορά, από το **broadcast operation**.

45η ΑΛΛΑΓΗ: Γράφουμε στη **main** εντός του **kiwi.c** τον κώδικα **ειδοποίησης** για **ενημέρωση**:

```
if (pthread_mutex_lock(params->mutex))
{
        perror("pthread_mutex_lock() error");
        exit(1);
}

while (1)
{
        if (pthread_mutex_lock(params->mutex))
        {
                perror("pthread_mutex_lock() error");
```

```
                exit(1);
        }

        if (*(params->start) == 1)
        {
                if (pthread_mutex_unlock(params->mutex))
                {
                        perror("pthread_mutex_unlock() error");
                        exit(1);
                }

                break;
        }
```

```
59      if (pthread_mutex_lock(params->mutex))
60      {
61          perror("pthread_mutex_lock() error");
62          exit(1);
63      }
64      while (1)
65      {
66          if (*(params->start) == 1)
67          {
68              if (pthread_mutex_unlock(params->mutex))
69              {
70                  perror("pthread_mutex_unlock() error");
71                  exit(1);
72              }
73
74              break;
75          }
```

```
        if (pthread_cond_wait(params->cv, params->mutex))
        {
                perror("pthread_cond_wait() error");
                exit(1);
        }
```

```
77          if (pthread_cond_wait(params->cv, params->mutex))
78          {
79              perror("pthread_cond_wait() error");
80              exit(1);
81          }
```

3) ΕΠΕΞΗΓΗΜΑΤΙΚΗ ΑΝΑΛΥΣΗ ΓΙΑ main thread

Η λειτουργία της **main thread** είναι η ακόλουθη:

i. Αναλύει τα **command line arguments** προκειμένου να εντοπίσει:

- Την **τιμή** του **seed**,
- Το **πλήθος** των **λειτουργιών** που θα **εκτελεστούν** από το κάθε **benchmark thread**,
- Το **πλήθος** των **benchmark threads**, που θα εκκινήσει,
- Την **πιθανότητα** για επιλογή του **write operation**, στο κάθε **benchmark thread**.

ii. **Ανοίγει** την **βάση**.

iii. **Ορίζει** τις **δομές** των **παραμέτρων**, οι οποίες θα περαστούν μέσω της **malloc**.

iv. **Αρχικοποιεί** τις **δομές** των **conditions**.

v. **Ορίζει** μεμονωμένα τις **παραμέτρους** του κάθε **benchmark thread**:

```
for (i = 0; i < threads_count; ++i)
{
        parameters[i].seed = (unsigned) rand_r(&seed);

        parameters[i].random_key_seed = (unsigned) rand_r(&seed);

        parameters[i].count = count;

        parameters[i].prob_write = atof(argv[4 + i]);

        parameters[i].db = db;

        parameters[i].start = &start_cond;

        parameters[i].cv = &bench_start_cond;

        parameters[i].mutex = &bench_start_mutex;
}
```

vi. **Δημιουργεί** μέσω της **malloc** και του **pthread_create** τα **threads**.

Το εκάστοτε **benchmark thread** λαμβάνει την κατάλληλη **παράμετρος parameters[i]** και όλα τους **εκτελούν** την λειτουργία του **benchmark_thread**.

vii. **Στέλνει** το **μήνυμα έναρξης** στα **benchmark threads**.

viii. **Αναμένει την ολοκλήρωση** των **benchmark threads**, μέσω της **pthread_join():**

```
// Wait for the threads to finish

for (i = 0; i < threads_count; ++i)
{
        if (pthread_join(bench_threads[i], &ret))
        {
                perror("pthread_join() error");
                exit(1);
        }
}
```

ix. **Λαμβάνει** τα **μεμονωμένα δεδομένα**, και τα ενώνει σε κοινή αποθήκευση.

x. **Εκτυπώνει** στην οθόνη ορισμένα προκαθορισμένα **στατιστικά**.

xi. Εκτελούμε τις **λειτουργίες τερματισμού**, δηλαδή:

- **κλείνει την βάση**, που άνοιξε στην αρχή,
- **αποδεσμεύει το mutex** και τα **condition variable**,
- **αποδεσμεύει** την **δεσμευμένη μνήμη**.

4) ΕΠΕΞΗΓΗΜΑΤΙΚΗ ΑΝΑΛΥΣΗ ΓΙΑ benchmark thread

Το **benchmark thread** βρίσκεται εντός του αρχείου **kiwi.c** :

void ***benchmark_thread**(void *data)

```
22    void* benchmark_thread(void *data)
```

Το **thread** μεταβάλλει την τιμή της **είσοδου,** στην δομή που την αναμένει:

struct **bench_parameters** ***params** = (struct **bench_parameters***)data;

```
24        struct bench_parameters *params = (struct bench_parameters*)data;
```

Η λειτουργία του **benchmark thread** είναι η ακόλουθη:

i. **Αρχικά θέτει** τα **δεδομένα** του στο 0.

ii. **Αναμένει** το **μήνυμα έναρξης** από το **main thread**.

iii. **Εκτελεί count operations:**

a. **Σε κάθε λειτουργία υπολογίζει** την **πιθανότητα** εκτέλεσης της **add()** (ή **get()**):

double **rndDouble** = (double) **rand_r**(&seed) / RAND_MAX;

bool **performWrite** = (**rndDouble** <= params->prob_write);

```
90            double rndDouble = (double) rand_r(&seed) / RAND_MAX;
91
92            bool performWrite = (rndDouble <= params->prob_write);
```

b. Ενεργοποιεί όμοιες λειτουργίες με τις **_write_test()** και **_read_test()**.

c. Στην **_random_key() εισάγεται** το **τοπικό seed** του εκάστοτε **benchmark thread**:

random**key**(**key**, **KSIZE**, &params->**random_key**_seed);

```
98                _random_key(key, KSIZE, &params->random_key_seed);
```

d. **ανανεώνει** κάποια **δεδομένα που αφορούν** την λειτουργία.

iv. **Ενημερώνει** κατάλληλα τα εναπομείναντα **δεδομένα.**

v. **Τερματίζει** τη διαδικασία με χρήση της **pthread_exit()**.

C. ΑΠΟΤΕΛΕΣΜΑΤΑ ΕΝΔΕΙΚΤΙΚΩΝ ΠΑΡΑΔΕΙΓΜΑΤΩΝ ΕΚΤΕΛΕΣΗΣ

Προτού ξεκινήσουμε τα πειράματα, πρέπει να κάνουμε **compile** και **build,** το αρχείο **kiwi,** χρησιμοποιώντας το **make**.

Οπότε μετακινούμαστε στον φάκελο **kiwi-source** και **εκτελώ** την εντολή **make**.

Παραθέτουμε screenshot της διαδικασίας:

Για να εκτελέσουμε τα παραδείγματα, μετακινούμαστε στον φάκελο **bench** και τρέχουμε με τη σειρά τις εντολές:

i. **make clean**

ii. **make**

iii. **./kiwi_bench [parameters]**

Εξετάσαμε τα **σενάρια**, με τις παρακάτω **συνθήκες**:

- Θέτω το **αρχικό seed** στο **15**,
- Θέτω το **πλήθος** των **operations** του **κάθε benchmark thread** στο **100**,
- Χρησιμοποιώ **2 benchmark threads**.

Στο **κάθε σενάριο** παρουσιάζουμε την **έξοδο** για **3 διαφορετικές εκτελέσεις**.

1. **RUN1 ./kiwi-bench 15 100 2 0.9 0.9**

   Και **τα 2 threads** κάνουν **περισσότερα writes**

	RUN 1.1	RUN 1.2	RUN 1.3
Total Operations			
done	200	200	200
sec/op	0.000016	0.00002	0.000031
ops/sec	126023.9	98231.83	64998.38
cost(sec)	0.001587	0.002036	0.003077
Random-Write			
done	179	179	179
sec/op	0.000003	0.000004	0.000009
writes/sec	366053.2	257554	116158.3
cost(sec)	0.000489	0.000695	0.001541
Random-Read			
done	21	21	21
sec/op	0.000001	0.000002	0.000005
reads/sec	1105263	636363.6	201923.1
cost(sec)	0.000019	0.000033	0.000104

2. **RUN2 ./kiwi-bench 15 100 2 0.1 0.1**

Και **τα 2 threads** κάνουν **περισσότερα reads**

	RUN 2.1	RUN 2.2	RUN 2.3
Total Operations			
done	200	200	200
sec/op	0.000017	0.00174	0.000013
ops/sec	114942.5	104986.9	149700.6
cost(sec)	0.00174	0.001905	0.001336
Random-Write			
done	14	14	14
sec/op	0.000005	0.000003	0.000003
writes/sec	181818.2	358974.4	388888.9
cost(sec)	0.000077	0.000039	0.000036
Random-Read			
done	186	186	186
sec/op	0.000002	0.000002	0.000001
reads/sec	603896.1	641379.3	925373.1
cost(sec)	0.000308	0.00029	0.000201

3. **RUN3 ./kiwi-bench 15 100 2 0.1 0.8**

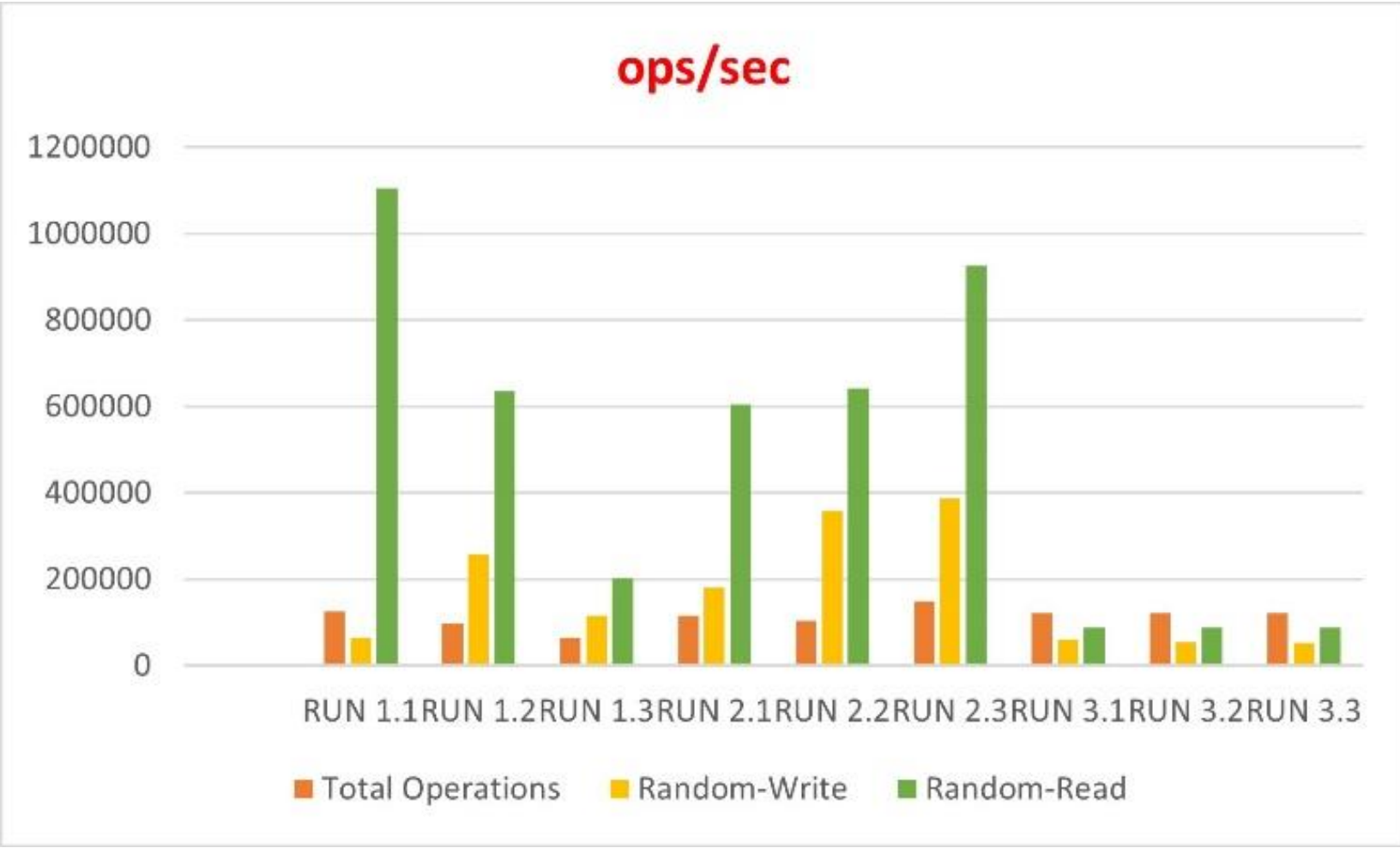
Το **1ο thread** κάνει **περισσότερα reads** και
το **2ο thread** κάνει **περισσότερα writes**

	RUN 3.1	RUN 3.2	RUN 3.3
Total Operations			
done	200	200	200
sec/op	0.000016	0.000016	0.000016
ops/sec	122774.7	121951.2	122324.2
cost(sec)	0.001629	0.00164	0.001635
Random-Write			
done	88	88	88
sec/op	0.000017	0.000018	0.000019
writes/sec	59742.02	54489.16	53789.73
cost(sec)	0.001473	0.001615	0.001636
Random-Read			
done	112	112	112
sec/op	0.000011	0.000011	0.000011
reads/sec	88959.49	88677.75	88537.55
cost(sec)	0.001259	0.001263	0.001265

ΔΙΑΓΡΑΜΜΑΤΑ

	RUN 1	RUN 2	RUN 3
Total Operations	200	200	200
Random-Write	179	14	88
Random-Read	21	186	112

	RUN 1.1	RUN 1.2	RUN 1.3	RUN 2.1	RUN 2.2	RUN 2.3	RUN 3.1	RUN 3.2	RUN 3.3
Total Operations	0.000016	0.00002	0.000031	0.000017	0.000019	0.000013	0.000016	0.000016	0.000016
Random-Write	0.000003	0.000004	0.000009	0.000005	0.000003	0.000003	0.000017	0.000018	0.000019
Random-Read	0.000001	0.000002	0.000005	0.000002	0.000002	0.000001	0.000011	0.000011	0.000011



	RUN 1.1	RUN 1.2	RUN 1.3	RUN 2.1	RUN 2.2	RUN 2.3	RUN 3.1	RUN 3.2	RUN 3.3
Total Operations	126023.9	98231.83	64998.38	114942.5	104986.9	149700.6	122774.7	121951.2	122324.2
Random-Write	64998.38	257554	116158.3	181818.2	358974.4	388888.9	59742.02	54489.16	53789.73
Random-Read	1105263	636363.6	201923.1	603896.1	641379.3	925373.1	88959.49	88677.75	88537.55

	RUN 1.1	RUN 1.2	RUN 1.3	RUN 2.1	RUN 2.2	RUN 2.3	RUN 3.1	RUN 3.2	RUN 3.3
Total Operations	0.001587	0.002036	0.003077	0.00174	0.001905	0.001336	0.001629	0.00164	0.001635
Random-Write	0.000489	0.000695	0.001541	0.000077	0.000039	0.000036	0.001473	0.001615	0.001636
Random-Read	0.000019	0.000033	0.000104	0.000308	0.00029	0.000201	0.001259	0.001263	0.001265

ΣΥΝΟΛΙΚΑ

	RUN 1.1	RUN 1.2	RUN 1.3	RUN 2.1	RUN 2.2	RUN 2.3	RUN 3.1	RUN 3.2	RUN 3.3
Total Operations									
done	200	200	200	200	200	200	200	200	200
sec/op	0.000016	0.00002	0.000031	0.000017	0.00174	0.000013	0.000016	0.000016	0.000016
ops/sec	126023.9	98231.83	64998.38	114942.5	104986.9	149700.6	122774.7	121951.2	122324.2
cost(sec)	0.001587	0.002036	0.003077	0.00174	0.001905	0.001336	0.001629	0.00164	0.001635
Random-Write									
done	179	179	179	14	14	14	88	88	88
sec/op	0.000003	0.000004	0.000009	0.000005	0.000003	0.000003	0.000017	0.000018	0.000019
writes/sec	366053.2	257554	116158.3	181818.2	358974.4	388888.9	59742.02	54489.16	53789.73
cost(sec)	0.000489	0.000695	0.001541	0.000077	0.000039	0.000036	0.001473	0.001615	0.001636
Random-Read									
done	21	21	21	186	186	186	112	112	112
sec/op	0.000001	0.000002	0.000005	0.000002	0.000002	0.000001	0.000011	0.000011	0.000011
reads/sec	1105263	636363.6	201923.1	603896.1	641379.3	925373.1	88959.49	88677.75	88537.55
cost(sec)	0.000019	0.000033	0.000104	0.000308	0.00029	0.000201	0.001259	0.001263	0.001265