



start, stop είναι για να δηλώσουν σε ποιο εύρος τιμών να γίνουν τα count. Το wicththread παίρνει τιμή ανάλογα με τη σειρά δημιουργίας του σε σχέση με τα υπόλοιπα νήματα.

- 4) Ακόμη προστέθηκαν τα συνολικά κόστη για όλα τα νήματα, τα οποία ενημερώνονται από τις συναρτήσεις read\_test, write\_test και τυπώνονται στο εφόσον τελειώσουν όλα τα νήματα.
- 5) Εδώ υπάρχουν οι ορισμοί των απαραίτητων mutexes για την ενημέρωση των στατιστικών.
- 6) Τέλος μπήκε εδώ ο δείκτης για τη db και αφαιρέθηκε από τη kiwi, διότι υπάρχουν οι κλήσεις open και close στη bench.c

### Αλλαγές στο αρχείο bench.c:

- 1) Υποστήριξη μίας ακόμη λειτουργίας “readwrite” η οποία εκτελεί reads, writes με είσοδο στη γραμμή εντολών. Θα εξηγηθεί παρακάτω.
- 2) Υποστήριξη δημιουργίας νημάτων με είσοδο στη γραμμή εντολών.

Πλέον η είσοδος είναι της μορφής:

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

- 3) Να σημειωθεί ότι το άνοιγμα και κλείσιμο της βάσης, μεταφέρθηκε στη bench.c με σκοπό να μην ανοιγοκλείνει η βάση κάθε φορά και επίσης να μην ανοίγει κάθε νήμα την ίδια βάση. Ήταν έτσι στην αρχή και δημιουργούσε αρκετά προβλήματα κατά την εκτέλεση των εντολών.

- 4) Παραμετροποίηση της εντολής write:

```
if (strcmp(argv[1], "write") == 0)
{
    if (argc == 3){
        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();

        db = db_open(DATAS);

        data_t data;
        data.count = count;
        data.r = 0;
        data.start = 0;
        data.stop = count - 1;

        _write_test((void *) &data);
    }
    else if (argc == 4)
    {
        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();

        db = db_open(DATAS);

        data_t data;
        data.count = count;
        data.r = 0;
        data.start = 0;
        data.stop = count - 1;

        if(atoi(argv[3]) != 0)
            data.r = 1;
        _write_test((void *) &data);
    }
}
```

- Για είσοδο ./kiwi-bench write <count>, δηλαδή για args == 3: Γίνεται ένας αριθμός από writes με βάση το όρισμα count
- Για είσοδο ./kiwi-bench write <count> <random>, δηλαδή για args == 4: Πέρα από τη παραπάνω λειτουργία, δίνεται η επιλογή για εισαγωγή στη βάση random κλειδιών ή όχι. Η προεπιλεγμένη τιμή για random είναι το 0, δηλαδή ‘όχι random’.

- Για είσοδο ./kiwi-bench write <count> <random> <threads>, δηλαδή για args==5:

```

else if (argc == 5)
{
    int total_count = atoi(argv[2]);
    _print_header(total_count);
    _print_environment();

    db = db_open(DATAS);

    int r = 0;
    int thread = 0;

    if(atoi(argv[3]) != 0)
        r = 1;
    int threads = atoi(argv[4]);
    if(threads < 1){
        printf("This can not be applied. Threads input have to be >= 1\n");
        exit(1);
    }
    if(threads > total_count){
        printf("This can not be applied. Threads input must <= Count input\n");
        exit(1);
    }
    if(threads >= 1)
        thread = 1;

    db = db_open(DATAS);

    pthread_t id[threads];
    int i;
    int flag = 0;

    double poses_aithseis_to_ka8e_nhma = (double) total_count / threads;
    if(isInteger(poses_aithseis_to_ka8e_nhma) == 1)
        flag = 1;

    int count = (long int) floor(poses_aithseis_to_ka8e_nhma);

    first_write_start_time = LONG_MAX;
    total_write_cost = 0;
    total_write_count = 0;
    pthread_mutex_init(&m_total_write_cost, NULL);
    pthread_mutex_init(&m_total_write_count, NULL);

    data_t array_structs[threads];

    for (i = 0; i < threads; i++){
        array_structs[i].r = r;
        array_structs[i].thread = thread;
        array_structs[i].whichThread = i;
        array_structs[i].start = (int) count * i;
        array_structs[i].stop = (int) count * i + count - 1;

        if( (i == threads-1) && (flag == 1) )
            array_structs[i].stop = total_count - 1;

        pthread_create(&id[i], NULL, _write_test, (void *) &array_structs[i]);
    }
    for (i = 0; i < threads; i++)
        pthread_join(id[i], NULL);

    db_close(db);

    total_write_cost = last_write_stop_time - first_write_start_time;

    printf("|Random-Write   (done:%ld): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec);\n"
        ,total_write_count, (double)(total_write_cost / total_write_count)
        ,(double)(total_write_count / total_write_cost)
        ,total_write_cost);
}
else{

```

Πέρα από τις δύο παραπάνω λειτουργίες, εδώ υπάρχει επιπλέον η επιλογή του πλήθους των νημάτων από το όρισμα “threads”.

Επιπλέον, εφόσον ο χρήστης έχει επιλέξει να έχει threads, πρέπει να εισάγει πάνω από ένα νήμα σε αυτή την επιλογή. Ακόμη, ο αριθμός των νημάτων θα πρέπει να μη ξεπερνάει των αριθμό των counts (λειτουργιών).

Επίσης υπάρχει ένα flag, το οποίο βρίσκει εάν υπάρχει ίση κατανομή των λειτουργιών count στα νήματα και αυτό γίνεται με βάση το αποτέλεσμα της διαίρεσης των count με τα threads.

Στη συνέχεια ανατίθεται σε κάθε νήμα το ποσοστό των write που του αναλογούν.

Στη περίπτωση όπου η διαίρεση δεν είναι ακέραια, τα # νήματα -1 παίρνουν τόσα write όσα γυρίζει η πράξη  $\text{floor}((\text{double}) \text{count} / \text{threads})$  και το τελευταίο νήμα παίρνει τα υπόλοιπα.

Επειδή για  $\text{args} == 5$  υπάρχουν νήματα, τα οποία τρέχουν παράλληλα, για τον υπολογισμό των στατιστικών ( χρόνου δηλαδή ) ακολουθείται η εξής λογική:

Αρχικοποίηση μεταβλητής `first_write_start_time` στη μέγιστη τιμή που μπορεί να πάρει ένας αριθμός τύπου `long` η οποία παίρνει τη πρώτη σε σειρά χρονική στιγμή δημιουργίας από όλα τα νήματα ( αυτό γίνεται στη `kiwi.c` ).

Αρχικοποίηση μεταβλητής `total_write_stop_time` στη τελευταία σε σειρά χρονική στιγμή τερματισμού όλων των νημάτων (επίσης στη `kiwi`).

Οπότε όταν τερματίσουν όλα, υπολογίζοντας τη διαφορά τους, έχουμε το πόσο έτρεξαν σύνολο αυτά τα νήματα.

Οι μεταβλητές `total_write_cost`, `total_write_count` είναι για τις συνολικές πράξεις και χρόνο που εκτελούν όλα τα νήματα. Για τα `print` δηλαδή.

Ακόμη γίνεται αρχικοποίηση των `mutexes` που χρησιμοποιεί η `kiwi.c`

Όσον αφορά τη δημιουργία των `threads`:

Δημιουργείται ένας πίνακα θέσεων `threads`, τύπου `pthread_t`

Ακόμη επειδή χρειάζονται `struct` για τα ορίσματά τους, υπάρχει αρχικοποίηση `struct` σε ένα πίνακα τύπου `data_t`, το τύπο του `struct` δηλαδή. Αυτό χρησιμεύει ώστε κάθε νήμα να έχει το δικό του `struct`, δηλαδή τις δικές του μεταβλητές. Είναι ένας τρόπος δυναμικής εκχώρησης μνήμης για τις μεταβλητές που χρειάζεται το κάθε νήμα. Δοκιμάστηκε στην αρχή με ένα `struct` και τα νήματα μπέρδευαν τις μεταβλητές μεταξύ τους.

## 5) Παραμετροποίηση της εντολής `read`:

```
else if (strcmp(argv[1], "read") == 0)
{
    if (argc == 3)
    {
        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();

        db = db_open(DATAS);

        data_t data;
        data.count = count;
        data.r = 0;
        data.start = 0;
        data.stop = count - 1;
        pthread_mutex_init(&m_found, NULL);

        _read_test((void *) &data);
    }
    else if (argc == 4)
    {
        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();

        data_t data;
        data.count = count;
        data.r = 0;
        data.start = 0;
        data.stop = count - 1;
        pthread_mutex_init(&m_found, NULL);

        db = db_open(DATAS);

        if(atoi(argv[3]) != 0)
            data.r = 1;
        _read_test((void *) &data);
    }
    else if (argc == 5)
    {
        // ...
    }
}
```

- Για είσοδο `./kiwi-bench read <count>`, δηλαδή για `args == 3`:  
Γίνεται ένας αριθμός από reads με βάση το όρισμα `count`
- Για είσοδο `./kiwi-bench read <count> <random>`, δηλαδή για `args == 4`:  
Πέρα από τη παραπάνω λειτουργία, δίνεται η επιλογή για ανάγνωση `random` κλειδιών από τη βάση ή όχι.  
Η προεπιλεγμένη τιμή για `random` είναι το 0, δηλαδή 'όχι random'.
- Για είσοδο `./kiwi-bench read <count> <random> <threads>`, δηλαδή για `args==5`:

```

else if (argc == 5)
{
    int total_count = atoi(argv[2]);
    _print_header(total_count);
    _print_environment();

    int r = 0;
    int thread = 0;

    if(atoi(argv[3]) != 0)
        r = 1;
    int threads = atoi(argv[4]);
    if(threads < 1){
        printf("This can not be applied. Threads input must be >= 1\n");
        exit(1);
    }
    if(threads > total_count){
        printf("This can not be applied. Threads input must <= Count input\n");
        exit(1);
    }
    if(threads >= 1)
        thread = 1;

    db = db_open(DATAS);

    pthread_t id[threads];
    int i;
    int flag = 0; //an to count/threads den einai akeraios

    double poses_aithseis_to_ka8e_nhma = (double) total_count / threads;
    if(!isInteger(poses_aithseis_to_ka8e_nhma) == 1)
        flag = 1;
    int count = (long int) floor(poses_aithseis_to_ka8e_nhma);

    total_read_cost = 0;
    total_read_count = 0;
    first_read_start_time = LONG_MAX;
    pthread_mutex_init(&total_read_cost, NULL);
    pthread_mutex_init(&total_read_count, NULL);
    pthread_mutex_init(&found, NULL);
    found = 0;
    data_t array_structs[threads];

    for (i = 0; i < threads; i++){
        array_structs[i].r = r;
        array_structs[i].thread = thread;
        array_structs[i].whichThread = i;
        array_structs[i].start = (int) count * i;
        array_structs[i].stop = (int) count * i + count - 1;

        if( (i == threads-1) && (flag == 1) )
            array_structs[i].stop = total_count - 1;

        pthread_create(&id[i], NULL, _read_test, (void *) &array_structs[i]);
    }
    for (i = 0; i < threads; i++)
        pthread_join(id[i], NULL);

    db_close(db);
    total_read_cost = last_read_stop_time - first_read_start_time;

    printf("Random-Read    (done:%ld, found:%d): %.5f sec/op; %.1f reads /sec(estimated); cost: %.3f(sec)\n",
        total_read_count, found,
        (double)(total_read_cost / total_read_count),
        (double)(total_read_count / total_read_cost),
        total_read_cost);
}
else{
    fprintf(stderr, "Usage: db-bench <write | read | readwrite> <count> <random> <threads>\n");
    exit(1);
}

```

Πέρα από τις δύο παραπάνω λειτουργίες, εδώ υπάρχει επιπλέον η επιλογή του πλήθους των νημάτων από το όρισμα "threads".

Επιπλέον, εφόσον ο χρήστης έχει επιλέξει να έχει threads, πρέπει να εισάγει πάνω από ένα νήμα σε αυτή την επιλογή. Ακόμη, ο αριθμός των νημάτων θα πρέπει να μη ξεπερνάει των αριθμό των counts (λειτουργιών).

Επίσης υπάρχει ένα flag, το οποίο βρίσκει εάν υπάρχει ίση κατανομή των λειτουργιών count στα νήματα και αυτό γίνεται με βάση το αποτέλεσμα της διαίρεσης των count με τα threads.

Στη συνέχεια ανατίθεται σε κάθε νήμα το ποσοστό των write που του αναλογούν.

Στη περίπτωση όπου η διαίρεση δεν είναι ακέραια, τα # νήματα -1 παίρνουν τόσα read όσα γυρίζει η πράξη  $\text{floor}((\text{double}) \text{count} / \text{threads})$  και το τελευταίο νήμα παίρνει τα υπόλοιπα.

Επειδή για `args == 5` υπάρχουν νήματα, τα οποία τρέχουν παράλληλα, για τον υπολογισμό των στατιστικών ( χρόνου δηλαδή ) ακολουθείται η εξής λογική:

Αρχικοποίηση μεταβλητής `first_read_start_time` στη μέγιστη τιμή που μπορεί να πάρει ένας αριθμός τύπου `long` η οποία παίρνει τη πρώτη σε σειρά χρονική στιγμή δημιουργίας από όλα τα νήματα ( αυτό γίνεται στη `kiwi.c` ).

Αρχικοποίηση μεταβλητής `total_read_stop_time` στη τελευταία σε σειρά χρονική στιγμή τερματισμού όλων των νημάτων (επίσης στη `kiwi`).

Οπότε όταν τερματίσουν όλα, υπολογίζοντας τη διαφορά τους, έχουμε το πόσο έτρεξαν σύνολο αυτά τα νήματα.

Οι μεταβλητές `total_read_cost`, `total_read_count` είναι για τις συνολικές πράξεις και χρόνο που εκτελούν όλα τα νήματα. Για τα `print` δηλαδή. Ακόμη γίνεται αρχικοποίηση των `mutexes` που χρησιμοποιεί η `kiwi.c`

Όσον αφορά τη δημιουργία των threads:

Δημιουργείται ένας πίνακας θέσεων threads, τύπου `pthread_t`. Ακόμη επειδή χρειάζονται struct για τα ορίσματά τους, υπάρχει αρχικοποίηση struct σε ένα πίνακα τύπου `data_t`, το τύπο του struct δηλαδή. Αυτό χρησιμεύει ώστε κάθε νήμα να έχει το δικό του struct, δηλαδή τις δικές του μεταβλητές. Είναι ένας τρόπος δυναμικής εκχώρησης μνήμης για τις μεταβλητές που χρειάζεται το κάθε νήμα. Δοκιμάστηκε στην αρχή με ένα struct και τα νήματα μπέρδευαν τις μεταβλητές μεταξύ τους.

Το έξτρα που υπάρχει στη read είναι η αρχικοποίηση ενός mutex `found` αλλά και μία μεταβλητή `found` το οποίο είναι για το πόσα κλειδιά βρήκε όντως οι λειτουργίες της.

## 6) Δημιουργία της εντολής readwrite:

```
else if (strcmp(argv[1], "readwrite") == 0)
{
    if (argc == 3)
    {
        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();

        db = db_open(DATAS);

        data_t data;
        data.count = count;
        data.r = 0;
        data.start = 0;
        data.stop = count - 1;

        int n = rand() % 2;
        if(n){
            _read_test((void *) &data);
        }else{
            _write_test((void *) &data);
        }
    }
    else if (argc == 4)
    {
        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();

        db = db_open(DATAS);

        data_t data;
        data.count = count;
        data.r = 0;
        data.start = 0;
        data.stop = count - 1;

        if(atoi(argv[3]) != 0)
            data.r = 1;

        // It is up to system to decide an operation. Either read or write
        int n = rand() % 2;
        if(n){
            read_test((void *) &data);
        }else{
            _write_test((void *) &data);
        }
    }
    else if (argc == 5)
    {

```

- Για είσοδο ./kiwi-bench read <count>, δηλαδή για args == 3:  
Γίνεται ένας αριθμός από reads ή writes με βάση το όρισμα count. Αποφασίζεται τυχαία ποια λειτουργία θα εκτελεστεί.
- Για είσοδο ./kiwi-bench read <count> <random>, δηλαδή για args == 4:  
Πέρα από τη παραπάνω λειτουργία, δίνεται η επιλογή για ανάγνωση random κλειδιών από τη βάση ή όχι.  
Η προεπιλεγμένη τιμή για random είναι το 0, δηλαδή 'όχι random'.
- Για είσοδο ./kiwi-bench read <count> <random> <threads>, δηλαδή για args==5:

```

else if (argc == 5)
{
    int total_count = atoi(argv[2]);
    _print_header(total_count);
    _print_environment();

    int r = 0;
    int thread = 0;

    if(atoi(argv[3]) != 0)
        r = 1;

    int threads = atoi(argv[4]);
    if(threads < 1){
        printf("This can not be applied. Threads input must be >= 1\n");
        exit(1);
    }
    if(threads > total_count){
        printf("This can not be applied. Threads input must <= Count input\n");
        exit(1);
    }
    if(threads >= 1)
        thread = 1;

    db = db_open(DATAS);

    first_read_start_time = LONG_MAX;
    first_write_start_time = LONG_MAX;
    total_write_count = 0;
    total_read_count = 0;
    pthread_mutex_init(&m_total_read_count, NULL);
    pthread_mutex_init(&m_total_write_count, NULL);
    found = 0;

    total_read_cost = 0;
    total_write_cost = 0;
    pthread_mutex_init(&m_total_write_cost, NULL);
    pthread_mutex_init(&m_total_read_cost, NULL);
    pthread_mutex_init(&m_found, NULL);

    int flag = 0; //an to count/threads den einai akeraios

    double poses_aithseis_to_ka8e_nhma = (double) total_count / threads;
    if(isInteger(poses_aithseis_to_ka8e_nhma) == 1)
        flag = 1;

    int count = (long int) floor(poses_aithseis_to_ka8e_nhma);

    pthread_t id[threads];
    data_t array_structs[threads];
    int i;
    int eginan_reads = 0;
    int eginan_writes = 0;

    for (i = 0; i < threads; i++){
        //n = rand() % 2;

        if(rand() % 2){ // writes
            eginan_writes = 1;
            array_structs[i].r = r;
            array_structs[i].wichThread = i;
            array_structs[i].thread = thread;
            array_structs[i].start = (int) count * i;
            array_structs[i].stop = (int) count * i + count - 1;

            if( (i == threads-1) && (flag == 1) )
                array_structs[i].stop = total_count - 1;
        }
    }
}

```



```

426         array_structs[i].stop = total_count - 1;
427
428         pthread_create(&id[i], NULL, _write_test, (void *) &array_structs[i]);
429     }
430     else{ // reads
431         eginan_reads = 1;
432         array_structs[i].r = r;
433         array_structs[i].thread = thread;
434         array_structs[i].whichThread = i;
435         array_structs[i].start = (int) count * i;
436         array_structs[i].stop = (int) count * i + count - 1;
437
438         if( (i == threads-1) && (flag == 1) )
439             array_structs[i].stop = total_count - 1;
440
441         pthread_create(&id[i], NULL, _read_test, (void *) &array_structs[i]);
442     }
443 }
444 for (i = 0; i < threads; i++)
445     pthread_join(id[i], NULL);
446
447 db_close(db);
448 total_write_cost = last_write_stop_time - first_write_start_time;
449 total_read_cost = last_read_stop_time - first_read_start_time;
450
451 if(eginan_reads)
452     printf("Random-Read  (done:%ld, found:%d): %.6f sec/op; %.1f reads /sec(estimated); cost:%.3f(sec):\n",
453         total_read_count, found,
454         (double)(total_read_cost / total_read_count),
455         (double)(total_read_count / total_read_cost),
456         total_read_cost);
457 if(eginan_writes)
458     printf("Random-Write  (done:%ld): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec):\n",
459         total_write_count, (double)(total_write_cost / total_write_count),
460         (double)(total_write_count / total_write_cost),
461         total_write_cost);
462 }
463 }
464 else{
465     fprintf(stderr, "Usage: db-bench <write | read | readwrite> <count> <random> <threads>\n");
466     exit(1);
467 }
468 return 1;
469 }

```

Πέρα από τις δύο παραπάνω λειτουργίες, εδώ υπάρχει επιπλέον η επιλογή του πλήθους των νημάτων από το όρισμα "threads".

Επιπλέον, εφόσον ο χρήστης έχει επιλέξει να έχει threads, πρέπει να εισάγει πάνω από ένα νήμα σε αυτή την επιλογή. Ακόμη, ο αριθμός των νημάτων θα πρέπει να μη ξεπερνάει των αριθμό των counts (λειτουργιών).

Επίσης υπάρχει ένα flag, το οποίο βρίσκει εάν υπάρχει ίση κατανομή των λειτουργιών count στα νήματα και αυτό γίνεται με βάση το αποτέλεσμα της διαίρεσης των count με τα threads. Οι λειτουργίες που αναφέρθηκαν μόλις γίνονται όπως στη write και τη read αντίστοιχα.

Επιπλέον εδώ υπάρχουν:

Αρχικοποίηση μεταβλητών, mutexes για reads και writes.

Για τη δημιουργία των νημάτων, σε κάθε επανάληψη:

Αποφασίζεται τυχαία αν θα δημιουργηθεί ένα νήμα που θα εκτελεί write και αρχικοποιείται όπως και στη write και αν θα δημιουργηθεί νήμα που θα εκτελεί read, του οποίου η αρχικοποίηση των παραμέτρων είναι ίδια με τη read.

Αλλάζουν μόνο δύο μεταβλητές που ελέγχουν αν έγιναν read / write για να τυπώσουν τα κατάλληλα στατιστικά.

7) Για κάθε εντολή που υλοποιήθηκε και υπάρχουν νήματα:

- Thread == 1
- Τα στατιστικά τυπώνονται στη bench και όχι στη kiwi
- Οι χρόνοι εκτέλεσης ( για τα στατιστικά συνολικά των νημάτων ) μετριοούνται με κώδικα της kiwi και της bench. Δηλαδή:

Αρχικοποιούνται μεταβλητές στη bench σε μία μέγιστη τιμή τύπου long.

Στη kiwi, συγκρίνεται αυτό το νούμερο με τον αρχικό χρόνο δημιουργίας του κάθε νήματος. Αν αυτός είναι μικρότερος, τότε το αντικαθιστά. Έτσι καταφέρνουμε να έχουμε το πότε ξεκίνησε το 1<sup>ο</sup> νήμα.

Αρχικοποιείται μία τιμή σε 0, και στη kiwi συγκρίνεται αυτή με το πότε τελείωσε το κάθε νήμα. Αν ο χρόνος του νήματος είναι μεγαλύτερος τότε την αντικαθιστά. Έτσι έχουμε το πότε τελείωσε και το τελευταίο νήμα.  
Η αφαίρεση αυτών των δύο μεταβλητών, δίνει το συνολικό χρόνο που έτρεξαν τα νήματα.

### Αλλαγές στο αρχείο makefile:

Ακόμη χρειάστηκε να αλλάξει η μεταγλώττιση και να προστεθεί ένα flag:

```
all:
    gcc $(DEBUG) $(WARN) bench.c -lm kiwi.c -L ../engine -lindexer -lpthread -lsnappy -o kiwi-bench
clean:
```

Το -lm αμέσως μετά το bench.c γιατί αλλιώς υπήρχαν σφάλματα στη μεταγλώττιση τα οποία αφορούσαν τη μέθοδο floor στη βιβλιοθήκη math και δεν ήταν επιτυχής.

Ακόμη και στο makefile που βρίσκεται στο κατάλογο kiwi-source αλλά και στο makefile που βρίσκεται στο κατάλογο bench προστέθηκε ένα flag που βοηθά την εκτέλεση του προγράμματος μέσω του gdb:

```
4 CFLAGS += -g
5
```

### Αλλαγές στο αρχείο kiwi.c:

1) Αλλαγές στη write\_test:

```

kiwi.c - Mousepad
File Edit Search View Document Help

1 #include <string.h>
2 // #include "../engine/db.h"
3 #include "../engine/variant.h"
4 #include "bench.h"
5
6 void *_write_test(void *arg)
7 {
8     data_t *d = (data_t *) arg;
9
10    long int count = d->count;
11    int r = d->r;
12    int thread = d->thread;
13
14    pthread_mutex_lock(&m_total_write_count);
15    total_write_count += d->stop - d->start + 1;
16    pthread_mutex_unlock(&m_total_write_count);
17
18    int i;
19    double cost;
20    long long start, end;
21    Variant sk, sv;
22
23    char key[KSIZE + 1];
24    char val[VSIZE + 1];
25    char sbuf[1024];
26
27    memset(key, 0, KSIZE + 1);
28    memset(val, 0, VSIZE + 1);
29    memset(sbuf, 0, 1024);
30
31    printf("Write      start: %d,      stop: %d\n", d->start, d->stop);
32
33    start = get_uptime_sec();
34
35    pthread_mutex_lock(&m_total_write_cost);
36    if(start < first_write_start_time)
37        first_write_start_time = start;
38    pthread_mutex_unlock(&m_total_write_cost);
39
40    for (i = d->start; i <= d->stop; i++) {
41        if (r)
42            _random_key(key, KSIZE);
43        else
44            snprintf(key, KSIZE, "key-%d", i);
45
46        if(thread != 1) // no threads
47            fprintf(stderr, "%d adding %s\n", i, key);
48
49        snprintf(val, VSIZE, "val-%d", i);
50
51        sk.length = KSIZE;
52        sk.mem = key;
53        sv.length = VSIZE;
54        sv.mem = val;
55
56        db_add(db, &sk, &sv, i, key, pthread_self(), d->withThread, d->thread);
57        if ((i % 10000) == 0) {
58            fprintf(stderr, "random write finished %d ops%30s\r",
59                    i,
60                    "");
61
62            fflush(stderr);
63        }
64    }
65
66    if(thread != 1)
67        db_close(db);
68
69    end = get_uptime_sec();
70    cost = end - start;
71
72    pthread_mutex_lock(&m_total_write_cost);
73    //total_write_cost += cost;
74    if(end > last_write_stop_time)
75        last_write_stop_time = end;
76    pthread_mutex_unlock(&m_total_write_cost);
77
78    if(thread == 1){
79        printf(LINET);
80        printf("%ld\n", pthread_self());
81    }else
82        printf(LINET);
83    if(thread != 1)
84        printf("[Random-Write {done:%ld}: %.6f sec/op; %.1f writes/sec(estimated); cost: %.3f(sec);\n",
85              count, (double)(cost / count)
86              , (double){count / cost}
87              , cost);
88
89    return 0;
90 }
91

```

Μέσω του δείκτη arg δημιουργείται ένας δείκτης στο struct τύπου data\_t που ήρθε ως όρισμα και με βάση αυτόν, παίρνει η συνάρτηση τα ορίσματά της.

Με το lock, unlock του m\_total\_write\_count, ενημερώνεται η μεταβλητή για το πόσα count γίνονται συνολικά στα νήματα. Αναγκαστικά μπαίνει μεταξύ αυτών μιας και είναι global μεταβλητή για αυτά.

Έπειτα προστέθηκε η λειτουργία του να τυπώνεται η αρχή και το τέλος των count που έχουν ανατεθεί στο κάθε νήμα.

Με το lock, unlock του m\_total\_write\_cost, ενημερώνεται η μεταβλητή first\_write\_start\_time για να λάβει το χρόνο του πρώτου νήματος. Αναγκαστικά μπαίνει μεταξύ αυτών μιας και είναι global μεταβλητή για κάθε νήμα.

Ακόμη τα count πλέον γίνονται από ένα όριο ως ένα άλλο, το οποίο γίνεται μέσω των μεταβλητών start, stop.

Στη συνέχεια, αν δεν υπάρχουν νήματα ( thread != 1) τότε γίνονται εδώ τα print, αλλιώς γίνονται αλλού. Θα εξηγηθεί στη συνέχεια το γιατί.

Αν δεν υπάρχουν νήματα η βάση κλείνει σε αυτό το σημείο, στη kiwi.

Με το lock,unlock του m\_total\_write\_cost γίνεται η ενημέρωση του πότε τελείωσε το πιο αργό νήμα. Μέσα σε mutexes γιατί η last\_write\_stop\_time είναι global variable.

Ακόμη όταν τελειώνει ένα νήμα την εκτέλεσή του, υπάρχει ξεχωριστή γραμμή που έχει οριστεί στο bench.h , με το id του thread ώστε να ξεχωρίζει.

Τέλος όπως προαναφέρθηκε, αν δεν υπάρχουν νήματα τα print γίνονται εδώ.

**!!!! Σημαντικό !!!!**

Θεωρήθηκε ότι ήταν πιο σωστό τα νήματα να παίρνουν διαφορετικά νούμερα στα count που θα κάνουν. Δηλαδή: να μη γράφουν δύο νήματα από το 0 ως το 8, αλλά να γράφουν από το 0 ως το 4 και από το 4 ως το 8. Γι αυτό το λόγο υπάρχουν οι μεταβλητές start, stop.

2) Αλλαγές στη read\_test:

```

92 void *_read_test(void *arg)
93 {
94     data_t *d = (data_t *) arg;
95
96     long int count = d->count;
97     int r = d->r;
98     int thread = d->thread;
99
100     pthread_mutex_lock(&m_total_read_count);
101     total_read_count += d->stop - d->start + 1;
102     pthread_mutex_unlock(&m_total_read_count);
103
104     int i;
105     int ret;
106     double cost = 0;
107     long long start,end;
108     Variant sk;
109     Variant sv;
110     char key[KSIZE + 1];
111
112     start = get_ustime_sec();
113
114     pthread_mutex_lock(&m_total_read_cost);
115     if(start < first_read_start_time)
116         first_read_start_time = start;
117     pthread_mutex_unlock(&m_total_read_cost);
118
119     printf("read    start: %d,      stop: %d\n", d->start, d->stop);
120
121     for (i = d->start; i <= d->stop; i++) {
122         memset(key, 0, KSIZE + 1);
123
124         /* if you want to test random write, use the following */
125         if (r)
126             _random_key(key, KSIZE);
127         else
128             sprintf(key, KSIZE, "key-%d", i);
129
130         if(thread != 1) // no threads
131             fprintf(stderr, "%d searching %s\n", i, key);
132
133         sk.length = KSIZE;
134         sk.mem = key;
135
136         ret = db_get(db, &sk, &sv, i, key, pthread_self(), d->wchThread, d->thread);
137
138         if (ret) {
139             //db_free_data(sv.mem);
140             pthread_mutex_lock(&m_found);
141             found++;
142             //printf("found %d\n", found);
143             pthread_mutex_unlock(&m_found);
144
145         } else {
146             INFO("not found key%s",
147                 sk.mem);
148         }
149
150         if ((i % 10000) == 0) {
151             fprintf(stderr, "random read finished %d ops%30s\r",
152                     i,
153                     "");
154
155             fflush(stderr);
156         }
157     }
158
159     if(thread != 1)
160         db_close(db);
161
162     end = get_ustime_sec();
163     cost = end - start;
164
165     pthread_mutex_lock(&m_total_read_cost);
166     if(end > last_read_stop_time)
167         last_read_stop_time = end;
168     pthread_mutex_unlock(&m_total_read_cost);
169
170     if(thread == 1){
171         printf(LINET);
172         printf("%ld\n", pthread_self());
173     }else
174         printf(LINE);
175
176     if(thread != 1)
177         printf("[Random-Read    (done:%ld, found:%d): %.6f sec/op; %.1f reads /sec(estimated); cost:%.3f(sec)\n",
178               count, found,
179               (double){cost / count},
180               (double){count / cost},
181               cost);
182
183     return 0;
184 }

```

Μέσω του δείκτη `arg` δημιουργείται ένας δείκτης στο struct τύπου `data_t` που ήρθε ως όρισμα και με βάση αυτόν, παίρνει η συνάρτηση τα ορίσματά της.

Με το `lock`, `unlock` του `m_total_read_count`, ενημερώνεται η μεταβλητή για το πόσα `count` γίνονται συνολικά στα νήματα. Αναγκαστικά μπαίνει μεταξύ αυτών μιας και είναι `global` μεταβλητή για αυτά.

Έπειτα προστέθηκε η λειτουργία του να τυπώνεται η αρχή και το τέλος των count που έχουν ανατεθεί στο κάθε νήμα.

Με το lock, unlock του m\_total\_read\_cost, ενημερώνεται η μεταβλητή first\_read\_start\_time για να λάβει το χρόνο του πρώτου νήματος. Αναγκαστικά μπαίνει μεταξύ αυτών μιας και είναι global μεταβλητή για κάθε νήμα.

Ακόμη τα count πλέον γίνονται από ένα όριο ως ένα άλλο, το οποίο γίνεται μέσω των μεταβλητών start, stop.

Στη συνέχεια, αν δεν υπάρχουν νήματα ( thread != 1) τότε γίνονται εδώ τα print, αλλιώς γίνονται αλλού. Θα εξηγηθεί στη συνέχεια το γιατί.

Το έξτρα που έχει η read\_test είναι ότι όταν επιστρέφει αν το βρήκε, υπάρχει κώδικας σε lock,unlock λόγω της global variable m\_found η οποία αυξάνει κάθε φορά αυτό το μετρητή αν βρέθηκε το κλειδί.

Αν δεν υπάρχουν νήματα η βάση κλείνει σε αυτό το σημείο, στη kiwi.

Με το lock,unlock του m\_total\_read\_cost γίνεται η ενημέρωση του πότε τελείωσε το πιο αργό νήμα. Μέσα σε mutexes γιατί η last\_read\_stop\_time είναι global variable.

Ακόμη όταν τελειώνει ένα νήμα την εκτέλεσή του, υπάρχει ξεχωριστή γραμμή που έχει οριστεί στο bench.h , με το id του thread ώστε να ξεχωρίζει.

Τέλος όπως προαναφέρθηκε, αν δεν υπάρχουν νήματα τα print γίνονται εδώ.

## 1<sup>η</sup> υλοποίηση Αμοιβαίου Αποκλεισμού

### Αλλαγές στο αρχείο db.h:

```
19
20 void db_close(DB* self);
21 int db_add(DB* self, Variant* key, Variant* value, int i, char* writing_key, pthread_t t_id, int wichThread, int thread);
22 int db_get(DB* self, Variant* key, Variant* value, int i, char* reading_key, pthread_t t_id, int wichThread, int thread);
23 int db_remove(DB* self, Variant* key);
```

- 1) Αλλαγή της διεπαφής της kiwi με τις έξω εφαρμογές. Κρίθηκε απαραίτητο για λόγους debugging. Είναι γνωστό ότι σε περίπτωση που η kiwi χρησιμοποιηθεί σε άλλη εφαρμογή εκείνη θα πρέπει να προσαρμοστεί σε αυτά τα ορίσματα, αλλά είναι μόνο αυτά τα 5 ορίσματα σε αυτές τις δύο συναρτήσεις για λόγους debugging οι οποίοι δεν είναι τόσο εμφανείς στη 1<sup>η</sup> υλοποίηση τους αμοιβαίου αποκλεισμού. Στη συγκεκριμένη υλοποίηση σκοπό έχουν όμως να γίνουν τα print της kiwi σε αυτό το σημείο του κώδικα, δηλαδή εκεί που τρέχει μόνο ένα νήμα. Έτσι θα φαίνεται λίγο πιο καθαρά πότε τρέχει τι.

- 2) Ορισμός ενός mutex στο αρχείο db.h ώστε να το βλέπουν τα αρχεία που το χρειάζονται.

```
//-----
pthread_mutex_t amoivaios_apokleismos
//-----
```

- 3) Πριν από κάθε λειτουργία στη βάση, απαιτείται να ανοίξει η βάση. Άρα εκεί γίνεται το initialize του mutex.

### Αλλαγές στο αρχείο db.c:

```

DB* db_open(const char* basedir)
{
    //-----
    pthread_mutex_init(&amoivaios_apokleismos, NULL);
    //-----

    return db_open_ex(basedir, LRU_CACHE_SIZE);
}

```

- 4) Στις λειτουργίες της βάσης, κλειδώνουμε τις λειτουργίες `db_add`, `db_get` και τις ξεκλειδώνουμε στο τέλος ώστε να είναι ενεργό μόνο από ένα νήμα τη φορά. Ακόμα έχοντας μία και μοναδική κλειδαριά και για τις δύο εργασίες, επιτυγχάνουμε να τρέχει μόνο μία από τις δύο κάθε φορά και με ένα νήμα ενεργό.

```

62 int db_add(DB* self, Variant* key, Variant* value, int i, char* writing_key, pthread_t t_id, int wichThread, int thread)
63 {
64     pthread_mutex_lock(&amoivaios_apokleismos);
65
66     if(thread == 1)
67         fprintf(stderr, "%d adding %s, by thread: %d ----- started %dst\n", i, writing_key, t_id, wichThread);
68
69     if (memtable_needs_compaction(self->memtable))
70     {
71         INFO("Starting compaction of the memtable after %d insertions and %d deletions",
72             self->memtable->add_count, self->memtable->del_count);
73         sst_merge(self->sst, self->memtable);
74         memtable_reset(self->memtable);
75     }
76
77     int return_value = memtable_add(self->memtable, key, value);
78
79     pthread_mutex_unlock(&amoivaios_apokleismos);
80     //-----
81
82     return return_value;
83 }
84
85 int db_get(DB* self, Variant* key, Variant* value, int i, char* reading_key, pthread_t t_id, int wichThread, int thread)
86 {
87     //-----
88     pthread_mutex_lock(&amoivaios_apokleismos);
89
90     if(thread == 1)
91         fprintf(stderr, "%d searching %s, by thread: %d -----started %dst\n", i, reading_key, t_id, wichThread);
92
93     int return_value;
94
95     if (memtable_get(self->memtable->list, key, value) == 1)
96         return_value = 1;
97
98     return_value = sst_get(self->sst, key, value);
99
100    pthread_mutex_unlock(&amoivaios_apokleismos);
101    //-----
102
103    return return_value;
104 }
105

```

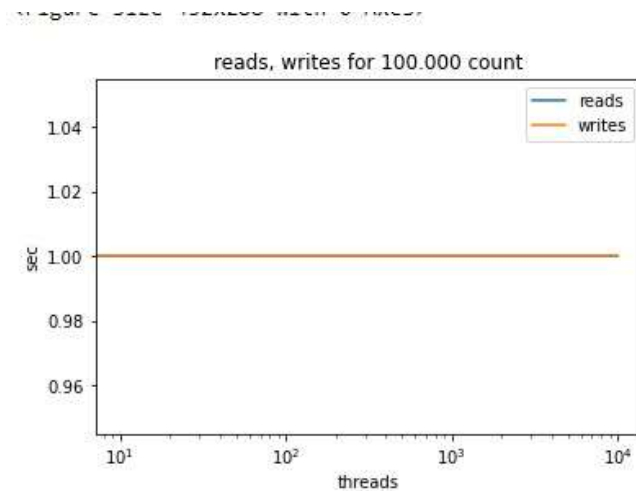
Τα print γίνονται μόνο αν έχουμε νήματα. Τις εξόδους των συναρτήσεων που καλούν αυτές οι δύο αποθηκεύονται προσωρινά σε αυτές τις μεθόδους ώστε να μην είναι αναγκαίο να πειραχτούν άλλα αρχεία, ώστε να γίνει εκεί το unlock του mutex.



### Αποτελέσματα:

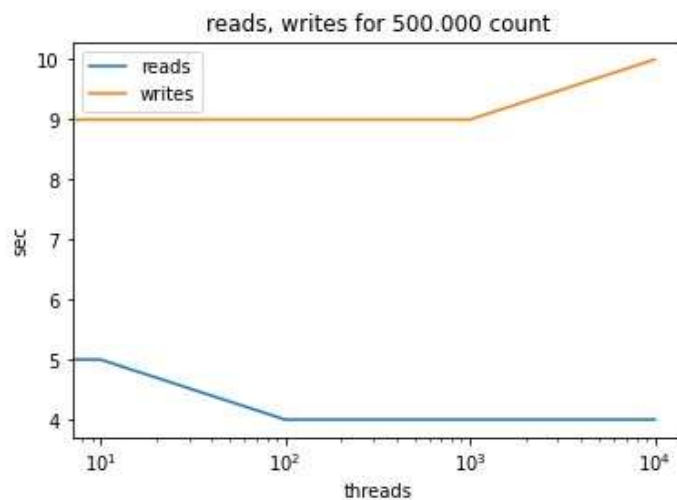
./kiwi-bench write 100000, ./kiwi-bench read 100000

σε ένα γράφημα:



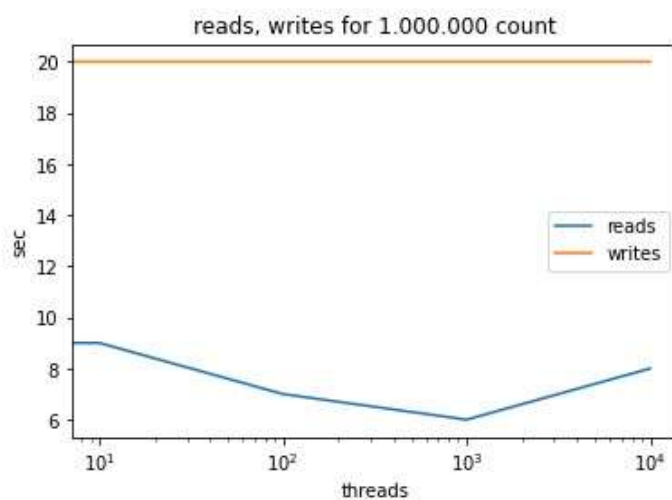
./kiwi-bench write 500000, ./kiwi-bench read 500000

σε ένα γράφημα:



./kiwi-bench write 1000000, ./kiwi-bench read 1000000

σε ένα γράφημα:



Τα παραπάνω αποτελέσματα είναι από μετρήσεις που έγιναν με τις παραπάνω εντολές ξεχωριστά.



Ο άξονας 'x' είναι σε λογαριθμική κλίμακα για καλύτερη παρουσίαση των αποτελεσμάτων και αναπαριστά το πλήθος των νημάτων, ενώ ο 'y' είναι ο συνολικός χρόνος των νημάτων για κάθε λειτουργία.

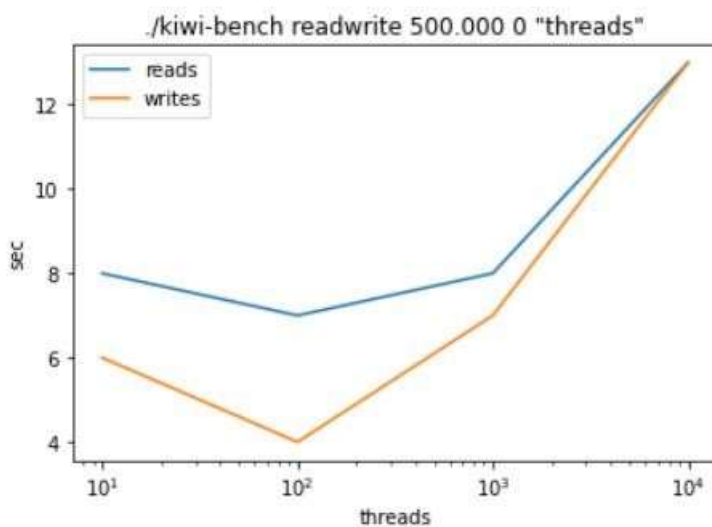
Για count = 100.000, μιας και είναι μικρό το νούμερο δε βλέπουμε διαφορά.

Για count = 500.000: Τα writes βλέπουμε ότι ακολουθούν σταθερή πορεία στο χρόνο που κάνουν, με διαφορά να γίνεται στη τελευταία εκτέλεση για  $10^4$  νήματα. Το οποίο απλά έτυχε. Στα reads επίσης ισχύει το ίδιο, καθώς οι τιμές είναι στα 4 sec.

Για count = 1.000.000: τα writes κρατάνε σταθερά το χρόνο τους. Αυτό μπορεί να έτυχε κιόλας. Αναφέρεται αυτό διότι μερικά compaction οι τιμές στις καθυστερήσεις διαφέρουν, αλλά στη προκείμενη περίπτωση σε όλα τα run ήταν άδεια η βάση. Τα read τώρα κυμαίνονται γύρω στο 7 σε όσα run κι αν κάναμε, απλά κρατήσαμε αυτές τις τιμές.

**Σημείωση:** Οι μετρήσεις αυτές είναι με το πρώτο run για τις παραπάνω εντολές. Δηλαδή δεν έτρεξαν 20 φορές οι εντολές `./kiwi-bench write 1000000`, να ληφθεί ο μέσος όρος και να μπει στη γραφική γιατί θα ήταν αρκετά χρονοβόρο.

`./kiwi-bench readwrite 500000 0 10/100/1000/10000`



Παρατηρούμε ότι για χαμηλό αριθμό νημάτων, τα νούμερα είναι σχεδόν ίδια, ενώ όσο τα αυξάνουμε, λόγω των καθυστερήσεων μεταξύ τους από τον αμοιβαίο αποκλεισμό, κάνουν περισσότερη ώρα να τερματίσουν

Επιβεβαιώθηκε ότι το πρόγραμμα δουλεύει σωστά, έπειτα από πληθώρα εκτελέσεων αλλά και είσοδος `sleep(x)` στη μία πλευρά της εκτέλεσης και `print` στην άλλη και είδαμε ότι όντως δε τύπωνε.

Παρουσίαση του τρόπου με του οποίου χωρίζονται τα counts σε threads:

./kiwi-bench write 4 0 2

```
0 adding key-0, by thread: 139640028874496
1 adding key-1, by thread: 139640028874496
+-----+
2 adding key-2, by thread: 139640037267200
3 adding key-3, by thread: 139640037267200
+-----+
[1149] 24 Mar 00:52:09.522 . db.c:35 Closing database 4
[1149] 24 Mar 00:52:09.522 . sst.c:596 IN sst merge the REFCOUNT IS at 2
[1149] 24 Mar 00:52:09.522 . sst.c:416 Sending termination message to the detached thread
[1149] 24 Mar 00:52:09.522 . sst.c:423 Waiting the merger thread
[1149] 24 Mar 00:52:09.522 . sst.c:166 The merge thread received a MERGE job
[1149] 24 Mar 00:52:09.522 . sst.c:167 Merging inside compaction thread
[1149] 24 Mar 00:52:09.522 . sst.c:609 Compacting the memtable to a SST file
[1149] 24 Mar 00:52:09.522 . sst.c:888 Range [key-0, key-3] DOES NOT overlap in level 0. Checking others
[1149] 24 Mar 00:52:09.522 . sst.c:834 Extracted range: [key-0, key-3]
[1149] 24 Mar 00:52:09.522 . sst.c:834 Extracted range: [key-0, key-3]
[1149] 24 Mar 00:52:09.522 . sst.c:940 Using level 2 for memtable compaction [key-0, key-3]
[1149] 24 Mar 00:52:09.522 . file.c:200 Creating directory structure: testdb/si/2
[1149] 24 Mar 00:52:09.522 . file.c:211 -> Creating testdb/si/2
[1149] 24 Mar 00:52:09.522 . sst.c:634 Compaction of 4 [4076 bytes allocated] elements started
[1149] 24 Mar 00:52:09.522 . sst_builder.c:167 Index block @ offset: 0x15E size: 30
[1149] 24 Mar 00:52:09.522 . sst_builder.c:168 Meta block @ offset: 0x116 size: 72
[1149] 24 Mar 00:52:09.522 . sst_builder.c:171 Bloom block @ offset: 0x106 size: 16
[1149] 24 Mar 00:52:09.522 . file.c:170 Truncating file testdb/si/2/0.sst to 452 bytes
[1149] 24 Mar 00:52:09.523 . file.c:65 Mapping of 452 bytes for testdb/si/2/0.sst
[1149] 24 Mar 00:52:09.523 . sst_loader.c:183 Index @ offset: 350 size: 30
[1149] 24 Mar 00:52:09.523 . sst_loader.c:184 Meta Block @ offset: 278 size: 72
[1149] 24 Mar 00:52:09.523 . sst_loader.c:201 Data size: 262
[1149] 24 Mar 00:52:09.523 . sst_loader.c:203 Index size: 0
[1149] 24 Mar 00:52:09.523 . sst_loader.c:204 Key size: 64
[1149] 24 Mar 00:52:09.523 . sst_loader.c:205 Num blocks size: 1
[1149] 24 Mar 00:52:09.523 . sst_loader.c:206 Num entries size: 4
[1149] 24 Mar 00:52:09.523 . sst_loader.c:207 Value size: 4000
[1149] 24 Mar 00:52:09.523 . sst_loader.c:210 Filter size: 16
[1149] 24 Mar 00:52:09.523 . sst_loader.c:211 Bloom offset 262 size: 16
[1149] 24 Mar 00:52:09.523 . sst.c:636 Compaction of 4 elements finished
[1149] 24 Mar 00:52:09.523 . file.c:170 Truncating file testdb/si/manifest to 44 bytes
[1149] 24 Mar 00:52:09.524 . sst.c:52 --- Level 0 [ 0 files, 0 bytes]---
[1149] 24 Mar 00:52:09.524 . sst.c:52 --- Level 1 [ 0 files, 0 bytes]---
[1149] 24 Mar 00:52:09.524 . sst.c:52 --- Level 2 [ 1 files, 452 bytes]---
[1149] 24 Mar 00:52:09.524 . sst.c:61 Metadata filename:0 smallest: key-0 largest: key-3
[1149] 24 Mar 00:52:09.524 . sst.c:52 --- Level 3 [ 0 files, 0 bytes]---
[1149] 24 Mar 00:52:09.524 . sst.c:52 --- Level 4 [ 0 files, 0 bytes]---
[1149] 24 Mar 00:52:09.524 . sst.c:52 --- Level 5 [ 0 files, 0 bytes]---
[1149] 24 Mar 00:52:09.524 . sst.c:52 --- Level 6 [ 0 files, 0 bytes]---
[1149] 24 Mar 00:52:09.524 . log.c:46 Removing old log file testdb/si/-1.log
[1149] 24 Mar 00:52:09.524 . sst.c:171 Merge successfully completed. Releasing the skiplist
[1149] 24 Mar 00:52:09.524 . skiplist.c:57 Skiplist refcount is at 0. Freeing up the structure
[1149] 24 Mar 00:52:09.524 . sst.c:177 Exiting from the merge thread as user requested
[1149] 24 Mar 00:52:09.524 . file.c:170 Truncating file testdb/si/manifest to 44 bytes
[1149] 24 Mar 00:52:09.525 . log.c:46 Removing old log file testdb/si/0.log
[Random-Write (done:4): 0.000000 sec/op; inf writes/sec(estimated); cost:0.000(sec);
myy60l@myy60l1lab1:~/kiwi/kiwi-source/bench$
```

./kiwi-bench read 3 0 2

```
0 searching key-0, by thread: 140097324099328
+-----+
1 searching key-1, by thread: 140097332492032
2 searching key-2, by thread: 140097332492032
+-----+
[1602] 24 Mar 01:08:56.829 . db.c:35 Closing database 0
[1602] 24 Mar 01:08:56.829 . sst.c:416 Sending termination message to the detached thread
[1602] 24 Mar 01:08:56.829 . sst.c:423 Waiting the merger thread
[1602] 24 Mar 01:08:56.829 . sst.c:177 Exiting from the merge thread as user requested
[1602] 24 Mar 01:08:56.829 . file.c:170 Truncating file testdb/si/manifest to 44 bytes
[1602] 24 Mar 01:08:56.829 . log.c:46 Removing old log file testdb/si/0.log
[1602] 24 Mar 01:08:56.829 . skiplist.c:57 Skiplist refcount is at 0. Freeing up the structure
[Random-Read (done:3, found:3): 0.000000 sec/op; inf reads /sec(estimated); cost:0.000(sec)
myy60l@myy60l1lab1:~/kiwi/kiwi-source/bench$
```

./kiwi-bench write 7 0 4





## 2<sup>η</sup> υλοποίηση Γραφέας-Αναγνώστες

### Αλλαγές στο αρχείο db.h:

```
19
20 void db_close(DB* self);
21 int db_add(DB* self, Variant* key, Variant* value, int i, char* writing_key, pthread_t t_id, int wichThread, int thread);
22 int db_get(DB* self, Variant* key, Variant* value, int i, char* reading_key, pthread_t t_id, int wichThread, int thread);
23 int db_remove(DB* self, Variant* key);
24
```

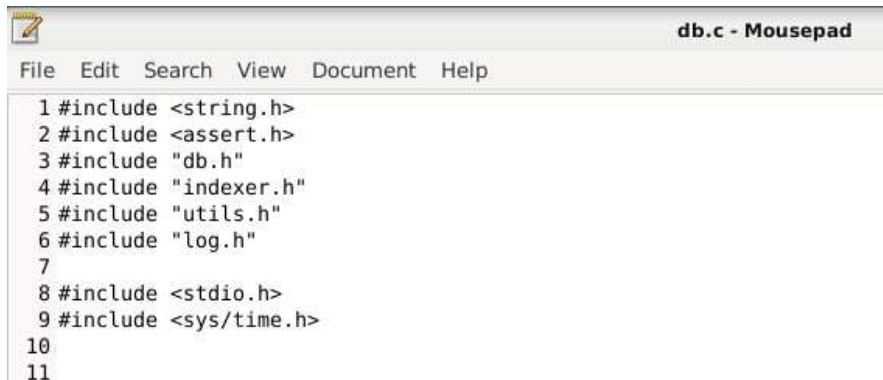
Διατηρήθηκε η αλλαγή στις διεπαφές των συναρτήσεων.

Από αυτή την υλοποίηση και μετά ήταν χρήσιμη αυτή η αλλαγή.

```
78
79 pthread_mutex_t m_writers;
80 pthread_mutex_t m_readers;
81
82 pthread_cond_t can_read;
83 pthread_cond_t can_write;
84 int readers;
85 int writers;
86 int waiting_writers;
87 //-----
```

Δήλωση mutex, condition\_variable γραφείς και αναγνώστες, καθώς και μεταβλητές που μετράνε πόσοι readers, writers είναι ενεργοί και μία μεταβλητή waiting\_writers που δηλώνει το πόσοι γραφείς περιμένουν.

### Αλλαγές στο αρχείο db.c:



```
db.c - Mousepad
File Edit Search View Document Help
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
8 #include <stdio.h>
9 #include <sys/time.h>
10
11
```

Include του stdio, sys/time. Το ένα είναι για τα print και το άλλο, για τη χρήση της συνάρτησης sleep τόσο για debug όσο και για απόδειξη ορθότητας.

```
27
28 DB* db_open(const char* basedir)
29 {
30     //-----
31     pthread_mutex_init(&m_writers, NULL);
32     pthread_mutex_init(&m_readers, NULL);
33     pthread_cond_init(&can_read, NULL);
34     pthread_cond_init(&can_write, NULL);
35     //-----
36
37     return db_open_ex(basedir, LRU_CACHE_SIZE);
38 }
39
40 void db_close(DB *self)
```

Αρχικοποίηση των mutexes και των condition variables στην open μιας και είναι η πρώτη συνάρτηση που καλείται για να ανοίξει η βάση.

```

27
60 int db_add(DB* self, Variant* key, Variant* value, int i, char* writing_key, pthread_t t_id, int wichThread, int thread)
61 {
62     //=====
63     pthread_mutex_lock(&m_writers);
64
65     //printf("WRITER readers: %d writers: %d\n", readers, writers);
66
67     //sleep(1);
68     waiting_writers += 1;
69     while( writers > 0 || readers > 0){
70         //printf("Writer thread %ld is waiting\n",pthread_self());
71         //pthread_mutex_lock(&m_readers);
72
73         pthread_cond_wait(&can_write, &m_writers);
74         //pthread_mutex_lock(&m_writers);
75         //pthread_cond_wait(&can_read, &m_readers);
76     }
77
78     if(thread == 1)
79         fprintf(stderr, "%d adding %s, by thread: %ld ----- started %dst\n", i, writing_key, t_id, wichThread);
80     writers += 1;
81
82     //pthread_cond_wait(&can_read, &m_readers);
83
84     pthread_mutex_unlock(&m_writers);
85     //=====
86
87     if (mtable_needs_compaction(self->mtable))
88     {
89         INFO("Starting compaction of the mtable after %d insertions and %d deletions",
90             self->mtable->add_count, self->mtable->del_count);
91         sst_merge(self->sst, self->mtable);
92         mtable_reset(self->mtable);
93     }
94
95     int return_value = mtable_add(self->mtable, key, value);
96
97     //=====
98     pthread_mutex_lock(&m_writers);
99     writers -= 1;
100     waiting_writers -= 1;
101     //printf("waiting_writers: %d, writer:%d\n", waiting_writers, writers);
102
103
104
105     pthread_cond_signal(&can_write);
106     //pthread_mutex_unlock(&m_writers);
107
108     pthread_cond_broadcast(&can_read);
109     pthread_mutex_unlock(&m_readers);
110
111     pthread_mutex_unlock(&m_writers);
112     //=====
113
114     return return_value;
115 }
116

```

Η ιδέα είναι ότι πρέπει να υπάρχει ένας που γράφει (db\_add), και πολλοί που διαβάζουν (db\_get) κάθε χρονική στιγμή

Για να πετύχουμε να δουλεύει ένας writer τη φορά:

Πρέπει αρχικά το πρώτο νήμα 'x' που θα τρέξει τη συνάρτηση να περάσει στο κώδικα και να κλειδώσει τη κλειδαριά ώστε να περιμένουν τα υπόλοιπα. Στη συνέχεια αυξάνει ένα μετρητή που δηλώνει το πόσοι γραφείς περιμένουν. Ασχέτως αν θα περιμένει αυτός ή όχι. Αν δε περιμένει, θα συνεχίσει και θα μειωθεί αυτή η τιμή στο κατάλληλο σημείο.

Μετά από αυτό για να συνεχίσει πρέπει να βεβαιωθεί ότι δεν υπάρχει ενεργός γραφέας ή αναγνώστης αντίστοιχα και αυτό γίνεται με τις δύο μεταβλητές reader, writers. Αν υπάρχει κάτι από αυτά, το νήμα θα καλέσει wait(&can\_write) και θα περιμένει. Ταυτόχρονα θα ξεκλειδωθεί η κλειδαριά για επόμενα νήματα.

Αν το νήμα κάνει τον έλεγχο και δε βρει αναγνώστη ή γραφέα ενεργό ή αντίστοιχα ξυπνήσει και δει ότι η συνθήκη είναι ψευδής, τότε προχωράει στο κώδικα.

Στο σημείο εκείνο υπάρχουν print που επιβεβαιώνουν ότι το νήμα αυτό, κάνει αυτή τη λειτουργία και αναφέρει επίσης και ποιο νήμα στη σειρά είναι σε σχέση με αυτά που δημιουργήθηκαν.

Στη συνέχεια δίνεται η άνεση στο σύστημα να ξεκλειδώσει τη κλειδαριά m\_writers ώστε να περάσει το επόμενο νήμα 'y' τη κλειδαριά. Δεν υπάρχει θέμα όμως καθώς θα κολλήσει στον έλεγχο και θα περιμένει να τελειώσει τουλάχιστον ο ενεργός γραφέας 'x' εφόσον ο ίδιος έχει αυξήσει τη μεταβλητή writers. Οπότε εξασφαλίζεται ότι είναι ενεργός ένας γραφέας τη φορά.

Το νήμα 'x' λοιπόν εφόσον ξεκλειδώσει το mutex m\_writers όπως αναφέρθηκε, εκτελεί το κώδικα που είναι υπεύθυνο για να βάλει το κλειδί στη βάση. Όταν τελειώσει η γραφή του

συγκεκριμένου κλειδιού, το νήμα 'χ' κλειδώνει ξανά τη κλειδαριά ώστε να αλλάξει τις global variables που αφορούν τη σωστή λειτουργία του αλγορίθμου writer-readers.

Εφόσον μειώσει το πλήθος των writers αλλά και αυτών που περιμένουν (waiting\_writers), ενημερώνει τη μεταβλητή συνθήκης ότι κάποιος μπορεί να γράψει με χρήση της signal(&can\_write). Δηλαδή θα ξυπνήσουν τα νήματα που περιμένουν, με λίγα λόγια αυτά που κάλεσαν wait(&can\_write).

Ακόμη με τη κλήση της broadcast(&can\_read) ξυπνάει όσους αναγνώστες περίμεναν από κλήση wait(&can\_read) επειδή υπήρχε ενεργός γραφέας και ξεκλειδώνει τη κλειδαριά m\_readers σε περίπτωση που δε υπάρχουν αναγνώστες που περιμένουν από τη κλήση wait(&can\_read) αλλά περιμένουν να περάσουν από τη κλειδαριά m\_readers.

Τέλος, ξεκλειδώνει επίσης τη κλειδαριά των writers m\_writers σε περίπτωση που δε περιμένουν writers από κλήση wait(&can\_write) οι οποίοι θα ξυπνούσαν κανονικά από τη signal(&can\_write) αλλά περιμένουν στη κλειδαριά των writers και περιμένουν να μπου.

```
116
117 int db_get(DB* self, Variant* key, Variant* value, int i, char* reading_key, pthread_t t_id, int wichThread, int thread)
118 {
119     //=====
120     pthread_mutex_lock(&m_readers);
121
122     printf("READER readers: %d writers: %d\n", readers, writers);
123     //sleep(4);
124
125     //if(waiting_writers > 0){
126     //pthread_cond_wait(&can_read, &m_readers);
127     while(writers > 0){ //false
128         printf("Reader thread %ld is waiting\n",pthread_self());
129         pthread_cond_wait(&can_read , &m_readers);
130     }
131     //sleep(4);
132
133     if(thread == 1)
134         fprintf(stderr, "%d searching %s, by thread: %ld -----started %dst\n", i, reading_key, t_id, wichThread);
135     readers += 1;
136
137     //pthread_mutex_unlock(&m_writers);
138
139     //pthread_cond_signal(&can_write);
140     pthread_mutex_unlock(&m_readers);
141     //=====
142
143     int return_value;
144
145     if (memtable_get(self->memtable->list, key, value) == 1)
146         return_value = 1;
147
148     return_value = sst_get(self->sst, key, value);
149
150     //=====
151     pthread_mutex_lock(&m_readers);
152     readers -= 1;
153     //printf("readers: %d\n", readers);
154
155
156     pthread_cond_broadcast(&can_read);
157     //pthread_mutex_unlock(&m_readers);
158
159     pthread_cond_signal(&can_write);
160     pthread_mutex_unlock(&m_writers);
161
162     pthread_mutex_unlock(&m_readers);
163     //=====
164
165     return return_value;
166 }
```

Για να πετύχουμε να δουλεύουν πολλοί readers τη φορά:

Πρέπει αρχικά το πρώτο νήμα 'χ' που θα τρέξει τη συνάρτηση να περάσει στο κώδικα και να κλειδώσει τη κλειδαριά ώστε να περιμένουν τα υπόλοιπα.

Μετά από αυτό για να συνεχίσει πρέπει να βεβαιωθεί ότι δεν υπάρχει ενεργός γραφέας με βάση τη μεταβλητή writers. Αν υπάρχει ενεργός γραφέας, το νήμα θα καλέσει wait(&can\_read) και θα περιμένει. Ταυτόχρονα θα ξεκλειδωθεί η κλειδαριά για επόμενα νήματα.

Αν το νήμα κάνει τον έλεγχο και δε βρει γραφέα ενεργό ή αντίστοιχα ξυπνήσει και δει ότι η συνθήκη είναι ψευδής, τότε προχωράει στο κώδικα.

Στο σημείο εκείνο υπάρχουν print που επιβεβαιώνουν ότι το νήμα αυτό, κάνει αυτή τη λειτουργία και αναφέρει επίσης και ποιο νήμα στη σειρά είναι σε σχέση με αυτά που δημιουργήθηκαν.

Στη συνέχεια δίνεται η άνοση στο σύστημα να ξεκλειδώσει τη κλειδαριά `m_readers` ώστε να περάσει το επόμενο νήμα 'γ' τη κλειδαριά. Δεν υπάρχει θέμα όμως καθώς θα κολλήσει στον έλεγχο και θα περιμένει να τελειώσει τουλάχιστον ο ενεργός γραφέας 'χ' σε περίπτωση που έχει αρχίσει κάποιος αλλιώς θα περάσει στη περιοχή που γίνεται `read` και θα υπάρχουν περισσότεροι από δύο αναγνώστες.

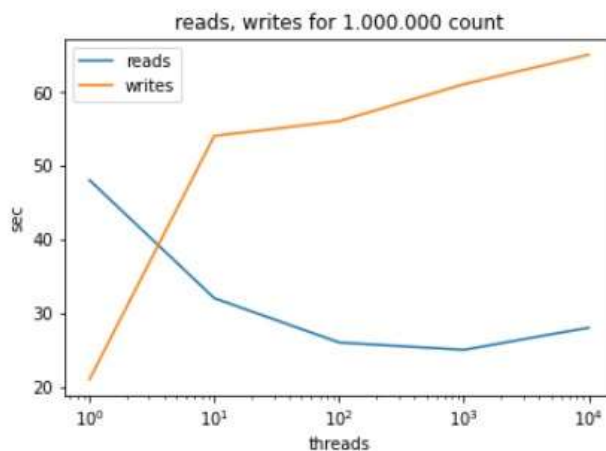
Το νήμα 'χ' λοιπόν εφόσον ξεκλειδώσει το mutex `m_readers` όπως αναφέρθηκε, εκτελεί το κώδικα που είναι υπεύθυνο για την αναζήτηση του κλειδιού στη βάση. Όταν τελειώσει το διάβασμα του συγκεκριμένου κλειδιού, το νήμα 'χ' κλειδώνει ξανά τη κλειδαριά `m_readers` ώστε να αλλάξει τις `global variables` που αφορούν τη σωστή λειτουργία του αλγορίθμου `writer-readers`.

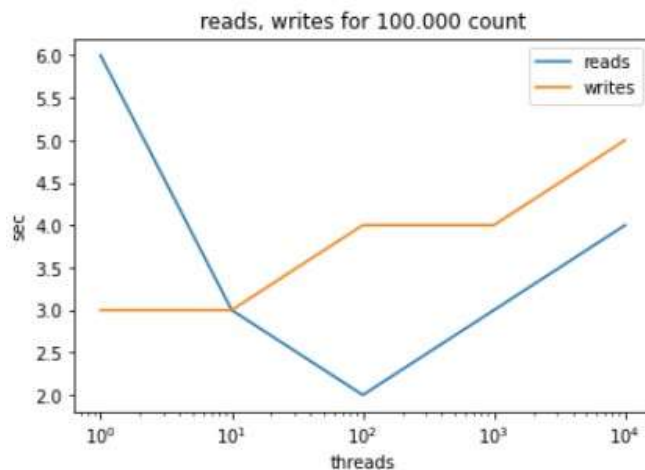
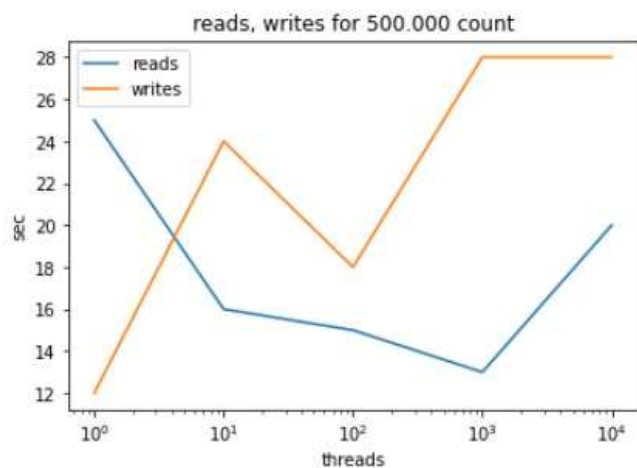
Εφόσον μειώσει το πλήθος των `readers`, ενημερώνει τη μεταβλητή συνθήκης ότι τα νήματα που περιμένουν μπορούν να διαβάσουν με χρήση της `broadcast(&can_read)`. Δηλαδή θα ξυπνήσουν τα νήματα που περιμένουν, με λίγα λόγια αυτά που κάλεσαν `wait(&can_read)` επειδή υπήρχε ενεργός γραφέας.

Ακόμη με τη κλήση της `signal(&can_write)` ξυπνάει ένα γραφέα που πιθανώς περίμενε από κλήση `wait(&can_write)` και ξεκλειδώνει τη κλειδαριά `m_writers` σε περίπτωση που δε υπάρχουν γραφέας που περιμένει από τη κλήση `wait(&can_write)` αλλά υπάρχουν γραφείς οι οποίοι περιμένουν να περάσουν από τη κλειδαριά `m_writers`.

Τέλος, ξεκλειδώνει τη κλειδαριά των `readers m_readers` σε περίπτωση που δε περιμένουν `readers` από κλήση `wait(&can_read)` οι οποίοι θα ξυπνούσαν κανονικά από τη `broadcast(&can_read)` αλλά περιμένουν στη κλειδαριά των `readers m_readers` και περιμένουν να μπουν.

## Στατιστικά:



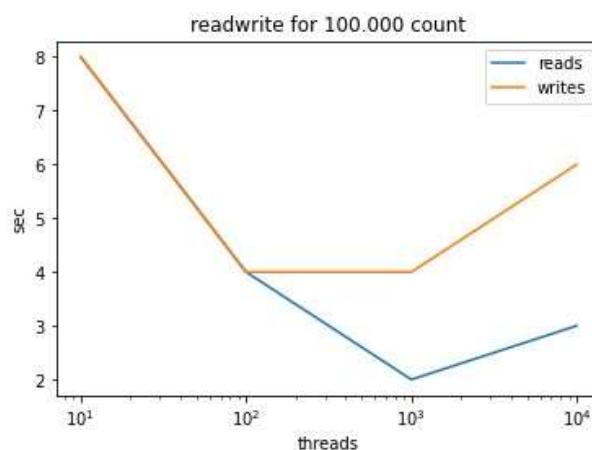
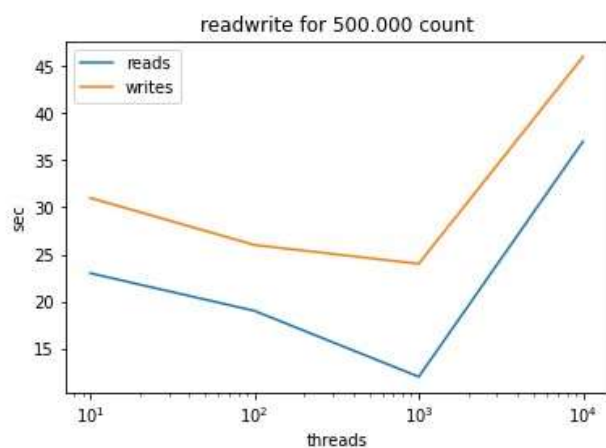
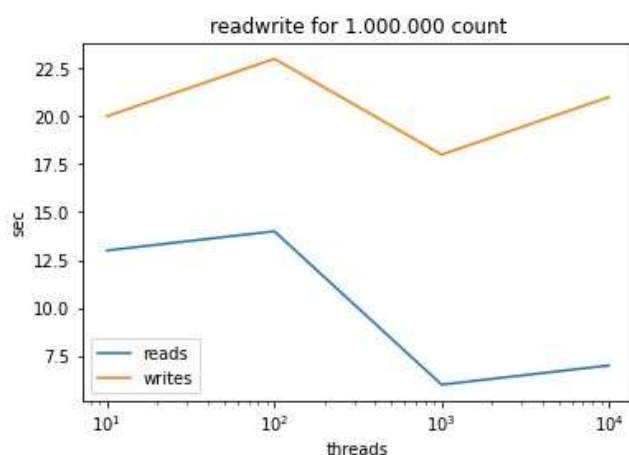


Οι παραπάνω εκτελέσεις κώδικα είναι read, write ξεχωριστά!

Βλέπουμε ότι για τα reads: ο χρόνος εκτέλεσης μειώνεται καθώς αυξάνουμε τον αριθμό των νημάτων και αυτό γίνεται γιατί τρέχουν παράλληλα. Μέχρι ένα σημείο όμως γιατί μετά μπαίνουν ορισμένες καθυστερήσεις συστήματος.

Για τα write: όπως είναι λογικό, όσο αυξάνουμε το πλήθος νημάτων τόσο αυξάνεται και η καθυστέρηση, καθώς τα ίδια δε τρέχουν παράλληλα εφόσον έχουμε ένα γραφέα τη φορά.

### Εκτελέσεις readwrite:





Να σημειωθεί ότι για εκτελέσεις readwrite έχουν συμπεριληφθεί στα γραφήματα και οι εκτελέσεις για  $10^0 = 1$  νήμα. Οπότε οι χρόνοι εκεί σαφώς αυξημένοι στην αρχή.

Για ταυτόχρονες εκτελέσεις της read, write μαζί, δηλαδή κλήση readwrite παρατηρείται το εξής:

Όλες οι τιμές έχουν ανέβει αναλογικά προς τα επάνω. Αυτό συμβαίνει προφανώς λόγω των καθυστερήσεων. Άλλο να τρέχει η read μόνη της και άλλο να τρέχει περιμένοντας τη write π.χ.

Κατά τα άλλα, τα αποτελέσματα είναι ίδια. Έχουμε πτώση τιμών για κάποια ιδανική τιμή των νημάτων και μετά άνοδο, λόγω πολλών καθυστερήσεων.

**Σημείωση:** Οι μετρήσεις αυτές είναι με το πρώτο run για τις παραπάνω εντολές. Δηλαδή δεν έτρεξαν 20 φορές οι εντολές ./kiwi-bench write 1000000 0 1000, να ληφθεί ο μέσος όρος και να μπει στη γραφική γιατί θα ήταν αρκετά χρονοβόρο. Δηλαδή στα παραπάνω μπορεί να τύχει κάποιο compaction το οποίο μπορεί να τύχει να πάρει περισσότερη ώρα. Έτσι αντί να τρέξουν 20 φορές π.χ., κάθε φορά τα αποτελέσματα αυτά ήταν με άδεια τη βάση.

### 3<sup>η</sup> υλοποίηση Αμοιβαίου Αποκλεισμού

Αλλαγές στο αρχείο db.h:

```
19
20 void db_close(DB* self);
21 int db_add(DB* self, Variant* key, Variant* value, int i, char* writing_key, pthread_t t_id, int wichThread, int thread);
22 int db_get(DB* self, Variant* key, Variant* value, int i, char* reading_key, pthread_t t_id, int wichThread, int thread);
23 int db_remove(DB* self, Variant* key, int i, char* reading_key, pthread_t t_id, int wichThread, int thread);
24
```

Ως προσθήκη εδώ είναι η αλλαγή της διεπαφής remove. Προσθήκη των ορισμάτων που έχουν και οι δύο πάνω. Αυτό γίνεται διότι η db\_remove καλεί την memtable\_remove η οποία με τη σειρά της καλεί την memtable\_edit η οποία χρειάζεται αυτά τα ορίσματα για λόγους που θα εξηγηθούν παρακάτω.

```
75
76 //-----
77 pthread_mutex_t m_writers;
78 //-----
```

Ακόμη αφαιρέθηκαν οι δηλώσεις των mutexes από τη db.h καθώς το πρόβλημα μεταφέρεται

σε εσωτερικότερες δομές. Αλλαγές στο αρχείο db.c:

```
7
8 #include <stdio.h>
9 #include <sys/time.h>
10
```

Εξακολουθούν να υπάρχουν include οι βιβλιοθήκες από το προηγούμενο ερώτημα.

```
28 DB* db_open(const char* basedir)
29 {
30     //-----
31     pthread_mutex_init(&m_writers, NULL);
32     //-----
33
34     return db_open_ex(basedir, LRU_CACHE_SIZE);
35 }
```

Ακόμα στη db\_open είναι η αρχικοποίηση του mutex m\_writers.

```

54
55 int db_add(DB* self, Variant* key, Variant* value, int i, char* writting_key, pthread_t t_id, int wichThread, int thread)
56 {
57     pthread_mutex_lock(&m_writers);
58     if (memtable_needs_compaction(self->memtable))
59     {
60         INFO("Starting compaction of the memtable after %d insertions and %d deletions",
61             self->memtable->add_count, self->memtable->del_count);
62         sst_merge(self->sst, self->memtable);
63         memtable_reset(self->memtable);
64     }
65     pthread_mutex_unlock(&m_writers);
66     int return_value = memtable_add(self->memtable, key, value, i, writting_key, t_id, wichThread, thread);
67     return return_value;
68 }

```

Στη db\_add έμεινε το lock και unlock του mutex των writers και ο λόγος είναι πως αν δεν έμπαινε αυτό εδώ, παρόλο την υλοποίηση που έχει γίνει στο memtable.c έβγαζε αρκετά συχνά segmentation. Οπότε ως είναι safe η υλοποίηση του compaction και ως αργεί λίγο παραπάνω.

Η υλοποίηση writer – readers όμως γίνεται μέσω της memtable.

```

74 int db_get(DB* self, Variant* key, Variant* value, int i, char* reading_key, pthread_t t_id, int wichThread, int thread)
75 {
76     if (memtable_get(self->memtable->list, key, value, i, reading_key, t_id, wichThread, thread) == 1)
77         return 1;
78     return sst_get(self->sst, key, value, i, reading_key, t_id, wichThread, thread);
79 }
80
81
82 int db_remove(DB* self, Variant* key, int i, char* reading_key, pthread_t t_id, int wichThread, int thread)
83 {
84     return memtable_remove(self->memtable, key, i, reading_key, t_id, wichThread, thread);
85 }
86

```

Εδώ απλά προστέθηκαν ορίσματα σε αυτές τις δύο συναρτήσεις. Memtable\_get, memtable\_remove, αλλά και στη memtable\_add και αφορούν τα print.

### !!! Σημαντικό !!!

Η αλλαγή όπως ειπώθηκε στην αρχή της άσκησης ως προς τη διεπαφή της kiwi με τις έξω εφαρμογές, έγινε με σκοπό την επιβεβαίωση από τον βαθμολογητή ότι τα σημεία λειτουργούν σωστά.

Π.χ. τα έξτρα print που υπάρχουν και σε αυτή την υλοποίηση αλλά και στη προηγούμενη στις δομές μέσα στη kiwi είναι τα print που έκανε η kiwi. Επειδή όμως, τα νήματα περιμένουν σε αυτό το σημείο και όχι στη kiwi, για οπτικούς λόγους αλλά και για επιβεβαίωση λειτουργίας ότι ένα νήμα π.χ. περιμένει όντως με τη συγκεκριμένη λειτουργία, αποφασίστηκε να μπουν εδώ.

**Αλγόριθμος writer – readers :** με ανάγνωση κατά τη γραφή, διαφέρει με τη προηγούμενη μόνο στη περίπτωση όπου υπάρχει γραφέας ο οποίος γράφει στη δομή skiplist και αναγνώστες οι οποίοι διαβάζουν από το sstable στα διάφορα επίπεδα. Όταν δεν έχουν βρει κάτι δηλαδή στο sstable.

Για να υλοποιηθεί αυτό, έπρεπε να αλλάξει η δομή memtable.

### Αλλαγές στο αρχείο memtable.c:

```

8 #include <stdio.h>
9 #include <sys/time.h>
10

```

Για το print και το debug

```

11
12 MemTable* memtable_new(Log* log)
13 {
14     MemTable* self = malloc(sizeof(MemTable));
15
16     if (!self)
17         PANIC("NULL allocation");
18
19     self->list = skiplist_new(SKIPLIST_SIZE);
20     skiplist_acquire(self->list);
21
22     self->needs_compaction = 0;
23     self->add_count = 0;
24     self->del_count = 0;
25
26     self->log = log;
27     self->lsm = 0;
28
29     log_recovery(log, self->list);
30
31     //-----
32     pthread_mutex_init(&m_readers, NULL);
33     pthread_cond_init(&can_read, NULL);
34     pthread_cond_init(&can_write, NULL);
35     //-----
36
37     return self;
38 }

```

Αρχιτοκτονία των mutex m\_readers και condition variables can\_read, can\_write.

```

62 static int memtable_edit(MemTable* self, const Variant* key, const Variant* value, OPT opt, int i, char* writing_key, pthread_t t_id, int wchThread, int thread)
63 {
64     // Here we need to insert the new node that has as a skipcode's key
65     // an encoded string that encompasses both the key and value supplied
66     // by the user.
67     //
68
69     size_t klen = variant_length(key->length); // key length
70     size_t vlen = (opt == DEL) ? 1 : variant_length(value->length + 1); // value length - 0 is reserved for tombstone
71     size_t encoded_len = klen + vlen + key->length + value->length;
72
73     if (opt == DEL)
74         assert(value->length == 0);
75
76     char *mem = malloc(encoded_len);
77     char *node_key = mem;
78
79     encode_varint32(node_key, key->length);
80     node_key += klen;
81
82     memcpy(node_key, key->mem, key->length);
83     node_key += key->length;
84
85     if (opt == DEL)
86         encode_varint32(node_key, 0);
87     else
88         encode_varint32(node_key, value->length + 1);
89
90     node_key += vlen;
91     memcpy(node_key, value->mem, value->length);
92
93     self->needs_compaction = log_append(self->log, mem, encoded_len);
94
95     //-----
96     pthread_mutex_lock(&m_writers);
97
98     //printf("WRITER readers: %d writers: %d\n", readers, writers);
99
100     //sleep(1);
101     waiting_writers += 1;

```

Μέχρι να γίνει lock(&m\_writers) ο κώδικας είναι τοπικός στο κάθε νήμα οπότε δε δημιουργείται πρόβλημα. Από το σημείο εκείνο και κάτω αρχίζει ο κώδικας για τον αλγόριθμο γραφέα αναγνώστη.

```

98 //printf("WRITER readers: %d writers: %d\n", readers, writers);
99
100 //sleep(1);
101 waiting_writers += 1;
102 while( writers > 0){
103     //printf("Writer thread %ld is waiting\n",pthread_self());
104     //pthread_mutex_lock(&m_readers);
105
106     pthread_cond_wait(&can_write, &m_writers);
107     //pthread_mutex_lock(&m_writers);
108     //pthread_cond_wait(&can_read, &m_readers);
109 }
110
111 if(thread == 1)
112     fprintf(stderr, "%d adding %s, by thread: %ld ----- started %dst\n", i, writting_key, t_id, wichThread);
113 writers += 1;
114
115 //pthread_cond_wait(&can_read, &m_readers);
116
117 pthread_mutex_unlock(&m_writers);
118 //=====
119
120 //sleep(1);
121
122
123
124 if (skiplist_insert(self->list, key->mem, key->length, opt, mem) == STATUS_OK_DEALLOC)
125     free(mem);
126
127 if (opt == ADD)
128     self->add_count++;
129 else
130     self->del_count++;
131
132 // DEBUG("memtable_edit: %.*s %.*s opt: %d", key->length, key->mem, value->length, value->mem, opt);
133
134
135 //=====
136 pthread_mutex_lock(&m_writers);
137 writers -= 1;
138 waiting_writers -= 1;
139 if(thread == 1)
140     printf("waiting_writers: %d, writer:%d\n", waiting_writers, writers);
141

```

Η ιδέα είναι ότι πρέπει να υπάρχει ένας που γράφει στη skiplist, και πολλοί αναγνώστες που διαβάζουν στα sst αρχεία κάθε χρονική στιγμή ή περιμένουν πολλοί αναγνώστες να διαβάσουν από τη mem.

Για να πετύχουμε να δουλεύει ένας writer τη φορά:

Πρέπει αρχικά το πρώτο νήμα 'x' που θα τρέξει τη συνάρτηση να περάσει στο κώδικα και να κλειδώσει τη κλειδαριά ώστε να περιμένουν τα υπόλοιπα. Στη συνέχεια αυξάνει ένα μετρητή που δηλώνει το πόσοι γραφείς περιμένουν. Ασχέτως αν θα περιμένει αυτός ή όχι. Αν δε περιμένει, θα συνεχίσει και θα μειωθεί αυτή η τιμή στο κατάλληλο σημείο.

Μετά από αυτό για να συνεχίσει πρέπει να βεβαιωθεί ότι δεν υπάρχει ενεργός γραφέας ή αναγνώστης αντίστοιχα και αυτό γίνεται με τις δύο μεταβλητές reader, writers. Αν υπάρχει κάτι από αυτά, το νήμα θα καλέσει wait(&can\_write) και θα περιμένει. Ταυτόχρονα θα ξεκλειδωθεί η κλειδαριά για επόμενα νήματα.

Αν το νήμα κάνει τον έλεγχο και δε βρει αναγνώστη ή γραφέα ενεργό ή αντίστοιχα ξυπνήσει και δει ότι η συνθήκη είναι ψευδής, τότε προχωράει στο κώδικα.

Στο σημείο εκείνο υπάρχουν print που επιβεβαιώνουν ότι το νήμα αυτό, κάνει αυτή τη λειτουργία και αναφέρει επίσης και ποιο νήμα στη σειρά είναι σε σχέση με αυτά που δημιουργήθηκαν.

Στη συνέχεια δίνεται η άνεση στο σύστημα να ξεκλειδώσει τη κλειδαριά m\_writers ώστε να περάσει το επόμενο νήμα 'y' τη κλειδαριά. Δεν υπάρχει θέμα όμως καθώς θα κολλήσει στον έλεγχο και θα περιμένει να τελειώσει τουλάχιστον ο ενεργός γραφέας 'x' εφόσον ο ίδιος έχει αυξήσει τη μεταβλητή writers. Οπότε εξασφαλίζεται ότι είναι ενεργός ένας γραφέας τη φορά.

Το νήμα 'x' λοιπόν εφόσον ξεκλειδώσει το mutex m\_writers όπως αναφέρθηκε, εκτελεί το κώδικα που είναι υπεύθυνο για να βάλει το κλειδί στο skiplist. Όταν τελειώσει η γραφή του

συγκεκριμένου κλειδιού, το νήμα 'χ' κλειδώνει ξανά τη κλειδαριά ώστε να αλλάξει τις global variables που αφορούν τη σωστή λειτουργία του αλγορίθμου writer-readers.

Εφόσον μειώσει το πλήθος των writers αλλά και αυτών που περιμένουν (waiting\_writers), ενημερώνει τη μεταβλητή συνθήκης ότι κάποιος μπορεί να γράψει με χρήση της signal(&can\_write). Δηλαδή θα ξυπνήσουν τα νήματα που περιμένουν, με λίγα λόγια αυτά που κάλεσαν wait(&can\_write).

Ακόμη με τη κλήση της broadcast(&can\_read) ξυπνάει όσους αναγνώστες περίμεναν από κλήση wait(&can\_read) επειδή υπήρχε ενεργός γραφέας και ξεκλειδώνει τη κλειδαριά m\_readers σε περίπτωση που δε υπάρχουν αναγνώστες που περιμένουν από τη κλήση wait(&can\_read) αλλά περιμένουν να περάσουν από τη κλειδαριά m\_readers.

Τέλος, ξεκλειδώνει επίσης τη κλειδαριά των writers m\_writers σε περίπτωση που δε περιμένουν writers από κλήση wait(&can\_write) οι οποίοι θα ξυπνούσαν κανονικά από τη signal(&can\_write) αλλά περιμένουν στη κλειδαριά των writers και περιμένουν να μπουν.

```
168 int mentable_get(SkipList* list, const Variant* key, Variant* value,    int i, char* reading_key, pthread_t t_id, int wichThread, int thread)
169 {
170     //=====
171     pthread_mutex_lock(&m_readers);
172
173     if(thread == 1)
174         printf("READER readers: %d  writers: %d\n", readers, writers);
175     //sleep(4);
176
177     //if(waiting_writers > 0){
178         //pthread_cond_wait(&can_read, &m_readers);
179     while(writers > 0){ //false
180         printf("Reader thread %ld is waiting\n",pthread_self());
181         pthread_cond_wait(&can_read , &m_readers);
182     }
183     //sleep(4);
184
185     if(thread == 1)
186         fprintf(stderr, "%d searching %s, by thread: %ld -----started %dst\n", i, reading_key, t_id, wichThread);
187     readers += 1;
188
189     //pthread_mutex_unlock(&m_writers);
190
191     //pthread_cond_signal(&can_write);
192     pthread_mutex_unlock(&m_readers);
193     //=====
194
195     SkipNode* node = skiplist_lookup(list, key->mem, key->length);
196
197     if (!node){
198
199         //=====
200         pthread_mutex_lock(&m_readers);
201         readers -= 1;
202         //printf("readers: %d\n", readers);
203
204
205         pthread_cond_broadcast(&can_read);
206         //pthread_mutex_unlock(&m_readers);
207
208         pthread_cond_signal(&can_write);
209         pthread_mutex_unlock(&m_writers);
210
211         pthread_mutex_unlock(&m_readers);
212         //=====
213     }
```

```

212 //=====
213
214     return 0;
215 }
216
217 const char* encoded = node->data;
218 encoded += varint_length(key->length) + key->length;
219
220 uint32_t encoded_len = 0;
221 encoded = get_varint32(encoded, encoded + 5, &encoded_len);
222
223 if (encoded_len > 1)
224     buffer_putnstr(value, encoded, encoded_len - 1);
225 else{
226
227     //=====
228     pthread_mutex_lock(&m_readers);
229     readers -= 1;
230     //printf("readers: %d\n", readers);
231
232     pthread_cond_broadcast(&can_read);
233     //pthread_mutex_unlock(&m_readers);
234
235     pthread_cond_signal(&can_write);
236     pthread_mutex_unlock(&m_writers);
237
238     pthread_mutex_unlock(&m_readers);
239     //=====
240
241     return 0;
242 }
243

```

προτιμότερο ο κώδικας για τον αποκλεισμό των αναγνωστών να μπει μία φορά στη συνάρτηση memtable\_get .

```

241
242     return 0;
243 }
244
245 //=====
246 pthread_mutex_lock(&m_readers);
247 readers -= 1;
248 //printf("readers: %d\n", readers);
249
250
251 pthread_cond_broadcast(&can_read);
252 //pthread_mutex_unlock(&m_readers);
253
254 pthread_cond_signal(&can_write);
255 pthread_mutex_unlock(&m_writers);
256
257 pthread_mutex_unlock(&m_readers);
258 //=====
259
260 return 1;
261 }
262

```

Για να πετύχουμε να δουλεύουν πολλοί readers τη φορά:

Πρέπει αρχικά το πρώτο νήμα 'x' που θα τρέξει τη συνάρτηση να περάσει στο κώδικα και να κλειδώσει τη κλειδαριά m\_readers ώστε να περιμένουν τα υπόλοιπα.

Μετά από αυτό για να συνεχίσει πρέπει να βεβαιωθεί ότι δεν υπάρχει ενεργός γραφέας με βάση τη μεταβλητή writers. Αν υπάρχει ενεργός γραφέας, το νήμα θα καλέσει wait(&can\_read, &m\_readers) και θα περιμένει. Ταυτόχρονα θα ξεκλειδωθεί η κλειδαριά m\_readers για επόμενα νήματα.

Αν το νήμα κάνει τον έλεγχο και δε βρει γραφέα ενεργό ή αντίστοιχα ξυπνήσει και δει ότι η συνθήκη είναι ψευδής, τότε προχωράει στο κώδικα.

Στο σημείο εκείνο υπάρχουν print που επιβεβαιώνουν ότι το νήμα αυτό, κάνει αυτή τη λειτουργία και αναφέρει επίσης και ποιο νήμα στη σειρά είναι σε σχέση με αυτά που δημιουργήθηκαν.

Στη συνέχεια δίνεται η άνεση στο σύστημα να ξεκλειδώσει τη κλειδαριά m\_readers ώστε να περάσει το επόμενο νήμα 'y' τη κλειδαριά. Δεν υπάρχει θέμα όμως καθώς θα κολλήσει στον έλεγχο και θα περιμένει να τελειώσει τουλάχιστον ο ενεργός γραφέας 'x' σε περίπτωση που έχει αρχίσει κάποιος αλλιώς θα περάσει στη περιοχή που γίνεται read και θα υπάρχουν περισσότεροι από δύο αναγνώστες.

Ο κώδικας για τους αναγνώστες μπήκε στο αρχείο memtable.c και στη συνάρτηση memtable\_get γιατί:

Η db\_get αρχικά αναζητεί για το κάποιο κλειδί με τη βοήθεια της συνάρτησης memtable\_get η οποία ψάχνει στη δομή skiplist αρχικά για να δει αν υπάρχει το κλειδί στη μνήμη. Αν το βρει επιστρέφει 1. Αν δε το βρει όμως καλεί την sst\_get, η οποία με τη σειρά της ψάχνει στη δομή skiplist. Αν το βρει επιστρέφει, αλλιώς αναζητά στα επίπεδα του sst.

Οπότε, επειδή η memtable\_get καλείται από δύο σημεία και κάνει lookup στο skiplist το οποίο είναι η δομή που γράφουν οι γραφείς, είναι



Το νήμα 'χ' λοιπόν εφόσον ξεκλειδώσει το mutex `m_readers` όπως αναφέρθηκε, εκτελεί το κώδικα που είναι υπεύθυνο για την αναζήτηση του κόμβου που περιέχει το κλειδί στη βάση. Όταν τελειώσει το διάβασμα του συγκεκριμένου κλειδιού, το νήμα 'χ' κλειδώνει ξανά τη κλειδαριά `m_readers` ώστε να αλλάξει τις `global variables` που αφορούν τη σωστή λειτουργία του αλγορίθμου `writer-readers`.

Εφόσον μειώσει το πλήθος των `readers`, ενημερώνει τη μεταβλητή συνθήκης ότι τα νήματα που περιμένουν μπορούν να διαβάσουν με χρήση της `broadcast(&can_read)`. Δηλαδή θα ξυπνήσουν τα νήματα που περιμένουν, με λίγα λόγια αυτά που κάλεσαν `wait(&can_read)` επειδή υπήρχε ενεργός γραφέας.

Ακόμη με τη κλήση της `signal(&can_write)` ξυπνάει ένα γραφέα που πιθανώς περίμενε από κλήση `wait(&can_write)` και ξεκλειδώνει τη κλειδαριά `m_writers` σε περίπτωση που δε υπάρχουν γραφέας που περιμένει από τη κλήση `wait(&can_write)` αλλά υπάρχουν γραφείς οι οποίοι περιμένουν να περάσουν από τη κλειδαριά `m_writers`.

Τέλος, ξεκλειδώνει τη κλειδαριά των `readers m_readers` σε περίπτωση που δε περιμένουν `readers` από κλήση `wait(&can_read)` οι οποίοι θα ξυπνούσαν κανονικά από τη `broadcast(&can_read)` αλλά περιμένουν στη κλειδαριά των `readers m_readers` και περιμένουν να μουν.

Να σημειωθεί απλά ότι υπάρχει ο ίδιος κώδικας πριν από τα 3 `return`.

Μπήκε λοιπόν ο κώδικας των αναγνωστών σε αυτό το σημείο διότι:

Αν υπάρχουν γραφείς στη δομή `skiplist`, οι αναγνώστες που βρίσκονται σε φάση αναζήτησης κλειδιού στη `skiplist` θα περιμένουν. Ενώ αν έχουν περάσει από αυτή τη φάση μπορούν πολλοί αναγνώστες να διαβάσουν από τη δομή `sst`, καθώς ο κώδικας θα γυρίσει στη συνάρτηση `sst_get` και θα συνεχίσει από εκεί που ψάχνει στα `levels` του `sst`.

Η λειτουργία αυτή επιβεβαιώνεται με τα `print` του `sst`. Δηλαδή όταν υπάρχει νήμα που επρόκειτο να διαβάσει από τη δομή των `sst files` θα τυπωθεί ένα μήνυμα ακριβώς τότε που λέει ότι γίνεται αυτή η λειτουργία. Αν πριν από αυτό το `print`, το `print` του `enabled writers` δώσει 1, σημαίνει ότι υπάρχουν αναγνώστες που διαβάζουν ενώ υπάρχει και γραφέας ενεργός.

Η λειτουργία αυτή επιβεβαιώνεται με το παρακάτω `screenshot`:

1. Φαίνεται ότι ξεκίνησε ο writer με id ...8464.
2. Κάνει 3 read στα `sst_files`
3. Ξεκινάει ένας γραφέας με id ...1168
4. Και εκείνη τη στιγμή συνεχίζει το νήμα με id ...8464, ο προηγούμενος αναγνώστης δηλαδή και διαβάζει ξανά από τα `sst_files`.
5. Ταυτόχρονα φαίνεται ότι υπάρχει ενεργός γραφέας που δεν έχει τελειώσει το `write`.

6. Οπότε πράγματι, ο αλγόριθμος δουλεύει σε αυτή τη περίπτωση και υπάρχουν αναγνώστες ότι υπάρχει ενεργός γραφέας.

```
[11552] 03 Apr 19:30:58.540 . sst.c:52 --- Level 0 [ 3 files, 9 KiB ]---
[11552] 03 Apr 19:30:58.540 . sst.c:61 Metadata filenum:57 smallest: key-0 largest: key-99
[11552] 03 Apr 19:30:58.540 . sst.c:61 Metadata filenum:58 smallest: key-10 largest: key-9
[11552] 03 Apr 19:30:58.540 . sst.c:52 --- Level 1 [ 1 files, 6 MiB ]---
[11552] 03 Apr 19:30:58.540 . sst.c:61 Metadata filenum:56 smallest: largest: key-99999
[11552] 03 Apr 19:30:58.540 . sst.c:52 --- Level 2 [ 1 files, 288 KiB ]---
[11552] 03 Apr 19:30:58.540 . sst.c:61 Metadata filenum:0 smallest: key-0 largest: key-99
[11552] 03 Apr 19:30:58.540 . sst.c:52 --- Level 3 [ 0 files, 0 bytes]---
[11552] 03 Apr 19:30:58.540 . sst.c:52 --- Level 4 [ 0 files, 0 bytes]---
[11552] 03 Apr 19:30:58.540 . sst.c:52 --- Level 5 [ 0 files, 0 bytes]---
[11552] 03 Apr 19:30:58.540 . sst.c:52 --- Level 6 [ 0 files, 0 bytes]---
read start: 30, stop: 39
READER readers: 0 writers: 0
30 searching key-30, by thread: 139896782878464 -----started 5st
Enabled writers = 0
IN SST: 30 searching key-30, by thread: 139896782878464 -----started 5st
READER readers: 0 writers: 0
31 searching key-31, by thread: 139896782878464 -----started 5st
Enabled writers = 0
IN SST: 31 searching key-31, by thread: 139896782878464 -----started 5st
READER readers: 0 writers: 0
32 searching key-32, by thread: 139896782878464 -----started 5st
Enabled writers = 0
IN SST: 32 searching key-32, by thread: 139896782878464 -----started 5st
Write start: 24, stop: 29
24 adding key-24, by thread: 139896791271168 ----- started 4st
READER readers: 0 writers: 0
33 searching key-33, by thread: 139896782878464 -----started 5st
Enabled writers = 1
IN SST: 33 searching key-33, by thread: 139896782878464 -----started 5st
READER readers: 0 writers: 1
Reader thread 139896782878464 is waiting
Write start: 18, stop: 23
read start: 12, stop: 17
READER readers: 0 writers: 1
Reader thread 139896808056576 is waiting
Write start: 6, stop: 11
Write start: 0, stop: 5
waiting_writers: 3, writer:0
25 adding key-25, by thread: 139896791271168 ----- started 4st
Reader thread 139896782878464 is waiting
Reader thread 139896808056576 is waiting
waiting_writers: 3, writer:0
26 adding key-26, by thread: 139896791271168 ----- started 4st
Reader thread 139896782878464 is waiting
Reader thread 139896808056576 is waiting
waiting_writers: 3, writer:0
27 adding key-27, by thread: 139896791271168 ----- started 4st
Reader thread 139896782878464 is waiting
Reader thread 139896808056576 is waiting
waiting_writers: 3, writer:0
```

```
156 int memtable_add(MemTable* self, const Variant* key, const Variant* value, int i, char* writting_key, pthread_t t_id, int wichThread, int thread)
157 {
158     return _memtable_edit(self, key, value, ADD, i, writting_key, t_id, wichThread, thread);
159 }
160
161 int memtable_remove(MemTable* self, const Variant* key, int i, char* writting_key, pthread_t t_id, int wichThread, int thread)
162 {
163     Variant value;
164     value.length = 0;
165     return _memtable_edit(self, key, &value, DEL, i, writting_key, t_id, wichThread, thread);
166 }
```

Αλλαγές στις διεπαφές της memtable\_add μας και χρειάζονται τα ορίσματα για την edit, εφόσον εκεί χρειάζονται τα print.

Και προαιρετικά αλλαγές στη db\_remove γιατί καλεί επίσης την edit και χρειάζονται τα ορίσματα, ασχέτως αν δε τη καλούμε εμείς στην άσκηση.



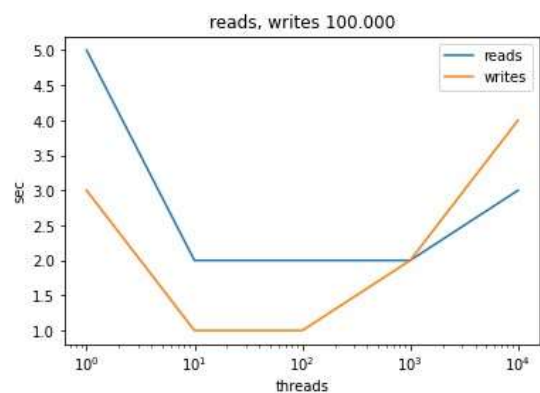
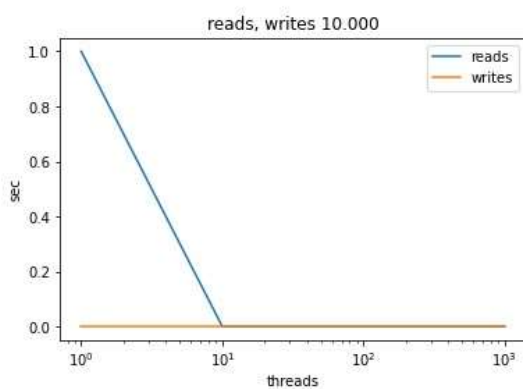
```

653 int sst_get(SST* self, Variant* key, Variant* value, int i, char* reading_key, pthread_t t_id, int wichThread, int thread)
654 {
655     #ifdef BACKGROUND_MERGE
656         int ret = 0;
657         pthread_mutex_lock(&self->cv_lock);
658         if (self->immutable)
659         {
660             DEBUG("Serving sst.get request from immutable memtable");
661             ret = memtable_get(self->immutable_list, key, value, i, reading_key, t_id, wichThread, thread);
662         }
663         pthread_mutex_unlock(&self->cv_lock);
664         if (ret)
665             return ret;
666         pthread_mutex_lock(&self->lock);
667     #endif
668     if (thread == 1) {
669         fprintf(stderr, "Enabled writers = %d\n", writers);
670         fprintf(stderr, "IN SST: %d searching %s, by thread: %d -----started %dst\n", i, reading_key, t_id, wichThread);
671     }
672     vector_clear(self->targets);
673     for (int level = 0; level < MAX_LEVELS; level++)
674     {
675         if (self->num_files[level] == 0)
676             continue;
677         if (level == 0)

```

Εισαγωγή εδώ στο κατάλληλο σημείο το print ώστε να ελέγχει το παραπάνω run (screenshot) όπου enabled writers = 1.

Στατιστικά:

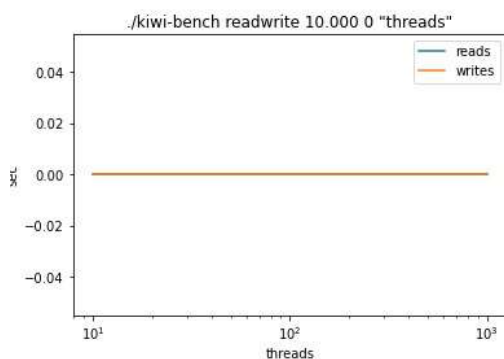


Από τις γραφικές φαίνεται ότι για 10.000 count τα νούμερα είναι όλα στο 0. Έτυχε το συγκεκριμένο run της read να βγει 1.

Για 100.000 counts:

Παρατηρείται ότι για λίγα thread (= 1) η read αργεί περισσότερο από ότι όταν τα αυξάνουμε και αυτό είναι λογικό γιατί τρέχουν παράλληλα. Αυτό σαφώς γίνεται μέχρι κάποιο όριο.

Στα write γενικά υπάρχει μία διακύμανση ίσως λόγω μερικών compaction αλλά γενικά πρέπει να είναι σε σταθερές τιμές για λίγα threads και αργότερα να αυξάνει ο χρόνος μιας και δε συγχρονίζονται.



Αντίστοιχα σε εκτελέσεις των readwrites για μικρά νούμερα δουλεύουν με αυτό το τρόπο. Για τιμές κοντά στο 500.000 count και μεγάλο πλήθος νημάτων, δε λειτουργεί σωστά, οπότε δεν υπάρχουν οι αντίστοιχες γραφικές.

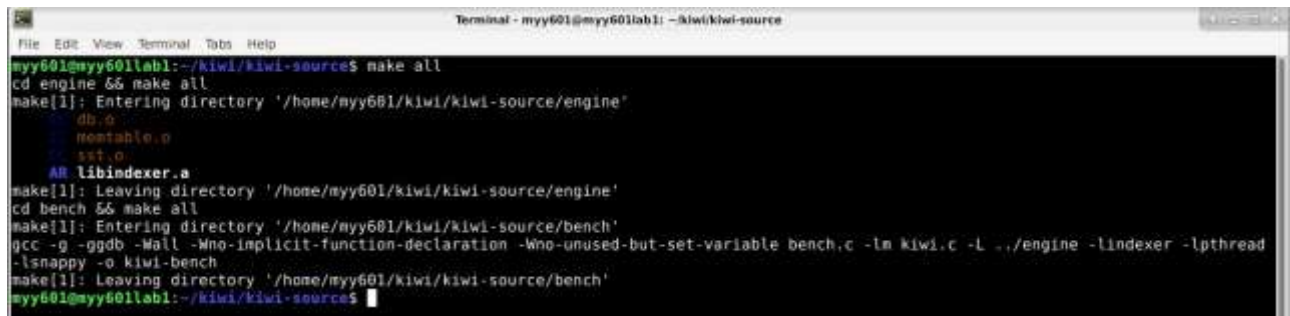
**Σημείωση:** Οι μετρήσεις αυτές είναι με το πρώτο run για τις παραπάνω εντολές. Δηλαδή δεν έτρεξαν 20 φορές οι εντολές ./kiwi-bench write 100000 0 1000, να ληφθεί ο μέσος όρος και να μπει στη γραφική γιατί θα ήταν αρκετά

χρονοβόρο. Δηλαδή στα παραπάνω μπορεί να τύχει κάποιο compaction το οποίο μπορεί να τύχει να πάρει περισσότερη

ώρα. Έτσι αντί να τρέξουν 20 φορές π.χ., κάθε φορά τα αποτελέσματα αυτά ήταν με άδεια τη βάση.

### Σημειώσεις:

- 1) Και η δεύτερη και η Τρίτη υλοποίηση αποδείχθηκαν ότι λειτουργούν με χρήση των print που άλλαξαν και μπήκαν στα κατάλληλα σημεία ώστε να φαίνεται ποιος περιμένει ποιον.  
Ακόμη δοκιμάστηκαν sleep(x) στους writers και τα αντίστοιχα print στους readers στη κρίσιμη περιοχή, ώστε να φανεί ότι όντως οι readers περιμένουν. Υπάρχουν και κατάλληλα μηνύματα στους writers όταν οι ίδιοι περιμένουν.
- 2) Όσον αφορά τη 2<sup>η</sup>, 3<sup>η</sup> υλοποίηση, υπάρχει ακριβώς μετά το lock των readers ένας έλεγχος: if (waiting\_writers > 0) => wait(&can\_read, &m\_readers). Αυτό είναι για να δώσει προτεραιότητα στους γραφείς. Ελέγχθηκε σε μερικές περιπτώσεις και λειτουργούσε αλλά δεν ελέγχθηκε σε όλες, οπότε και είναι σε σχόλια.
- 3) Η αρχικοποίηση των νημάτων γίνεται στο bench.c γιατί:  
Α. Εκεί έχουμε άμεσα τον αριθμό των νημάτων που έδωσε ο χρήστης.  
Β. Για δημιουργία νημάτων πρέπει να αλλάξει η συνάρτηση που καλούμε, οπότε είναι πιο σωστό να αλλάξουν οι ορισμοί των συναρτήσεων στο kiwi παρά να αλλάξουν οι διεπαφές των ορισμάτων στις συναρτήσεις της βάσης στο db.c
- 4) Έξοδος του make all στο κατάλογο /home/myy601/kiwi/kiwi-source



```
Terminal - myy601@myy601lab1: ~/kiwi/kiwi-source
myy601@myy601lab1:~/kiwi/kiwi-source$ make all
cd engine && make all
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/engine'
  gcc -g -ggdb -Wall -Wno-implicit-function-declaration -Wno-unused-but-set-variable bench.o -lm kiwi.o -L ../engine -lindexer -lpthread
  ar rcs 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 -lm kiwi.o -L ../engine -lindexer -lpthread
  ar rcs testdb
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/bench'
myy601@myy601lab1:~/kiwi/kiwi-source$
```

- 5) Έξοδος του make clean στο κατάλογο /home/myy601/kiwi/kiwi-source



```
myy601@myy601lab1:~/kiwi/kiwi-source$ make clean
cd engine && make clean
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/engine'
  rm -rf *.o libindexer.a
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/engine'
cd bench && make clean
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/bench'
  rm -f kiwi-bench
  rm -rf testdb
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/bench'
myy601@myy601lab1:~/kiwi/kiwi-source$
```

- 6) Σε όλες τις εκτελέσεις για τα printscreen που υπάρχουν στην αναφορά, τα writes ξεκινάνε από άδεια βάση και τα read ξεκινάνε με τη βάση να έχει μέσα τόσα κλειδιά όσα και τα read που θα υλοποιηθούν.
- 7) Στις readwrite, απλά υπήρχαν τόσα write όσα count επρόκειτο να εκτελεστούν.
- 8) Στο παραδοτέο φάκελο υπάρχουν και τα τρία engines. Ενεργό θα είναι μόνο η δεύτερη υλοποίηση με όνομα engine. Οι άλλες δύο θα είναι με όνομα engine\_1h, engine\_3h.

Από τις τρεις υλοποιήσεις, με βάση τα πειράματα τουλάχιστον που έγιναν, αλλά και την αναφορά αυτών ως γραφικές αναπαραστάσεις κρίθηκε ότι σωστά υλοποιημένες είναι η πρώτη και η δεύτερη.

Λειτουργεί και η Τρίτη με ικανοποιητικούς χρόνους, αλλά για `count >= 100000` δε λειτουργεί και περισσότερα από 10 νήματα, καθώς παρατηρούνται segmentation fault.