1η Εργαστηριακή Άσκηση Λειτουργικών Συστημάτων

Μάθημα	Λειτουργικά Συστήματα(ΜΥΥ601)
Διδάσκων	Αναστασιάδης Στέργιος
Υπεύθυνος Εργαστηρίου	Κυριαζής Ιωάννης
Βοηθοί	Καζαντζίδης Παναγιώτης, Παπακώστας Ιωάννης
Μέλη ομάδας/ΑΜ	Καζακίδης Κωνσταντίνος/4065, Καζακίδης Θεοχάρης/4679, Μητρόπουλος Γεώργιος/4733
Ακαδημαϊκό Έτος	2022-2023

Περιεχόμενα

Κεφάλαιο 1.

Εισαγωγή…… ……4

1.1 Σκοπός της αναφοράς…….....4
1.2 Περιγραφή των ζητούμενων της άσκησης……..……...……...4
1.3 Παρουσίαση του Kiwi και περιγραφή LSM tree……...……..…….5

Κεφάλαιο 2.

Υλοποίηση…….5

2.1 Προετοιμασία
2.1.1 Πρώτη εκτέλεση του κώδικα…….……...6
2.1.2 Παρουσίαση περιεχομένου σημαντικών αρχείων……...…..…..7
2.2 Υλοποίηση πολυνηματικής λειτουργίας …….…7
2.2.1 Δημιουργία νημάτων…….……..…..7
2.2.2 Υλοποίηση μικτής λειτουργίας…….……10
2.2.3 Χρήση της βάσης δεδομένων ως καθολική μεταβλητή…….……12
2.2.4 Ταυτοχρονισμός μεταξύ αναγνωστών και γραφέων…….…...……13
2.2.5 Χρονομέτρηση και στατιστικά επιδόσεων…….15

Κεφάλαιο 3.

Στατιστικά Επιδόσεων……...……17

3.1 Παρουσίαση συστήματος……..…17

3.2 Πειράματα……....17

3.2.1 read operation……...……18

3.2.2 write operation…….....……20

3.2.3 mixed operations……....……....22

Εισαγωγή

1.1 Σκοπός της αναφοράς

Η παρακάτω αναφορά έχει ως στόχο να επεξηγήσει τον κώδικα που υλοποιήσαμε στα πλαίσια της άσκησης. Συγκεκριμένα θα αναφέρουμε τόσο τις λειτουργίες που υλοποιήσαμε όσο και τον κώδικα που προσθέσαμε στα αρχεία που περιέχονται στην εικονική μηχανή. Ακόμη ,παρακάτω επισημαίνονται αναλυτικά οι αλλαγές στα αρχεία, στατιστικά και παρατηρήσεις που προέκυψαν κατά την εκτέλεση του κώδικα.

1.2 Περιγραφή των ζητούμενων της άσκησης

Αρχικά μας δίνεται σε γλώσσα C μια μηχανή αποθήκευσης και εμείς καλούμαστε να υλοποιήσουμε στις εντολές put και get ,της μηχανής αποθήκευσης ,πολυνηματική λειτουργία. Αυτό σημαίνει ότι οι εντολές put και get θα καλούνται από πολλά νήματα ταυτόχρονα, άρα να γίνονται σωστά οι ταυτόχρονες λειτουργίες και υπάρχουν στατιστικά χρόνου εκτέλεσης για κάθε μια από τις λειτουργίες που επιτελεί

1.3 Παρουσίαση του Kiwi και περιγραφή LSM tree

Το Kiwi είναι μία μηχανή αποθήκευσης κλειδιού-τιμής η οποία βασίζεται στο δέντρο log-structured merge (LSM-tree). Οι μηχανές αποθήκευσης που βασίζονται στο LSM-tree εντάσσονται στην κατηγορία των NoSQL βάσεων δεδομένων.
Η βασική ιδέα πίσω από ένα δέντρο LSM είναι να διαχωρίζει τις λειτουργίες εγγραφής και ανάγνωσης, επιτρέποντας πιο αποτελεσματική απόδοση και στις δύο περιπτώσεις. Όταν τα δεδομένα εγγράφονται στη βάση δεδομένων, αποθηκεύονται πρώτα σε ένα αρχείο καταγραφής, γεγονός που καθιστά τις εγγραφές πολύ πιο γρήγορα προσβάσιμες. Περιοδικά, τα περιεχόμενα του αρχείου καταγραφής συγχωνεύονται σε μια δομή δεδομένων στη μνήμη που ονομάζεται memtable, η οποία ταξινομείται και στη συνέχεια εγγράφεται στο δίσκο ως αμετάβλητο SSTable (Ταξινομημένος πίνακας συμβολοσειρών). Επειδή αυτά τα SSTables είναι αμετάβλητα, οι αναγνώσεις μπορούν να εκτελεστούν γρήγορα με τη διαδοχική σάρωση τους, γεγονός που αποφεύγει τις τυχαίες αναζητήσεις δίσκου και μειώνει την καθυστέρηση ανάγνωσης.
Η διαδικασία συγχώνευσης για τα SSTables είναι όπου μπαίνει το τμήμα "merge-tree" του LSM. Όταν δύο ή περισσότερα SSTables πρέπει να συγχωνευθούν, συγχωνεύονται με τρόπο που διατηρεί την ταξινομημένη σειρά τους, με αποτέλεσμα ένα νέο SSTable που περιέχει όλα τα δεδομένα από τους αρχικούς πίνακες με ταξινομημένη σειρά. Αυτή η διαδικασία μπορεί να επαναληφθεί όσες φορές χρειάζεται για να διατηρηθεί μια συμπαγής και αποτελεσματική αναπαράσταση των δεδομένων.
Εν κατακλείδι , οι μηχανές αποθήκευσης που βασίζονται σε δέντρο LSM προσφέρουν εξαιρετική απόδοση εγγραφής, καλή απόδοση ανάγνωσης και αποτελεσματική χρήση του χώρου στο δίσκο. Είναι κατάλληλα για περιπτώσεις χρήσης όπου απαιτείται υψηλή απόδοση εγγραφής, όπως σε μεγάλης κλίμακας κατανεμημένα συστήματα ή σε εφαρμογές ανάλυσης σε πραγματικό χρόνο.

Υλοποίηση

2.1 Προετοιμασία

2.1.1 Πρώτη εκτέλεση του κώδικα

Ξεκινήσαμε, κάνοντας εγκατάσταση το VMWare Workstation και φορτώνοντας την εικονική μηχανή. Έπειτα χρησιμοποιώντας τις default ρυθμίσεις δηλαδή με 1GB RAM και 1 Core από τον επεξεργαστή ακολουθήσαμε τα βήματα και μέσω της γραμμής εντολών πλοηγηθήκαμε στον κατάλογο ~/kiwi/kiwi-source καθώς έτσι θα ξεκινούσαμε να κάνουμε compile το πρόγραμμα.

Επομένως κάναμε (compile):
→ make clean ώστε να αφαιρεθούν τυχόν εκτελέσιμα αρχεία του προγράμματος και έπειτα,

→make all ώστε να δημιουργήσουμε τα εκτελέσιμα αρχεία του προγράμματος

Έπειτα μεταφερθήκαμε στον κατάλογο ~/kiwi/kiwi-source/bench και πειραματιστήκαμε τρέχοντας τις εντολές :

./kiwi-bench write 100000

και

/kiwi-bench read 100000

ώστε να δούμε τι συμβαίνει κατά την εκτέλεση του προγράμματος

2.1.2 Παρουσίαση περιεχομένου σημαντικών αρχείων

Τα σημαντικά αρχεία της άσκησης, δηλαδή τα αρχεία που χρήζουν αλλαγή είναι τα εξής:

- **bench.c** → Το αρχείο που περιέχει το κυρίως πρόγραμμα, ουσιαστικά την main
- **kiwi.c** → Είναι ουσιαστικά ένα interface που μας δίνει πρόσβαση στις λειτουργίες put και get της βάσης δεδομένων
- **db.c** → Περιέχει την υλοποίηση των βασικών λειτουργιών που επιτελεί η βάση

Εκτός από τα αρχεία που ανεφέρθηκαν παραπάνω αξίζει να αναφερθούν και τα παρακάτω καθώς υλοποιούν σημαντικές λειτουργίες της μηχανής αναζήτησης:

- **memtable.c** →Περιέχει την υλοποίηση της ταξινομημένης δομής memtable που διατηρεί τα πιο πρόσφατα δεδομένα στην μνήμη
- **skiplist.c** →Περιέχει την υλοποίηση της skiplist μια δομή δεδομένων που αποτελείται από πολλαπλές λίστες οργανωμένες σε διαφορετικά επίπεδα για να επιταχύνουν την αναζήτηση κάποιου στοιχείου
- **sst.c** →Περιέχει την υλοποίηση του Sorted String Table δηλαδή την δομή που αποθηκεύει τα δεδομένα στον δίσκο ταξινομημένα με βάση τα κλειδιά που περιέχουν συμβολοσειρές

2.2 Υλοποίηση πολυνηματικής λειτουργίας

2.2.1 Δημιουργία νημάτων

Πριν φτάσουμε στο σημείο να δημιουργήσουμε τα νήματα τα οποία εν τέλη θα εκτελέσουν παράλληλα τις λειτουργίες put και get κάναμε τις εξής μετατροπές στον κώδικα:

1. Εισαγωγή αριθμού νημάτων που θέλουμε να εκτελεστούν ως όρισμα στην main. Αρχικά ελέγχουμε αν ο χρήστης έχει δώσει τον σωστό αριθμό νημάτων (γραμμές 134-136 bench.c) και σε διαφορετική περίπτωση τερματίζουμε την λειτουργία του προγράμματος. Έπειτα εκχωρούμε την τιμή αυτή σε μια μεταβλητή **t_count** (γραμμές 143 και

188 bench.c). Τέλος ελέγχουμε αν ο χρήστης έχει δώσει αριθμό νημάτων μικρότερο του 1. Στην περίπτωση αυτή απλά βάζουμε τοναριθμό των νημάτων ίσο με ένα διότι δεν μπορεί να υπάρξει εκτέλεση με λιγότερο από ένα νήματα (γραμμές 144-147 και 189-192 bench.c).

2. Διαμοιρασμός των εργασιών μεταξύ των νημάτων. Παρατηρήσαμε ότι αν απλά κάθε νήμα εκτελούσε **count** αριθμό εργασιών τότε συνολικά θα εκτελούνταν **t_count * count** αριθμός εργασιών, πράγμα που προφανώς είναι λανθασμένο. Για να λυθεί το παραπάνω πρόβλημα διαιρούμε των αριθμό των εργασιών με τον αριθμό των νημάτων. Στην περίπτωση που η διαίρεση αυτή δεν είναι τέλεια να τελευταίο νήμα θα εκτελέσει τις επιπλέον εργασίες (γραμμές 151-152, 159-165, 196-197 και 204-211 bench.c)

```
int quotient = count / t_count;
int remainder = count % t_count;
```

```
if(i==t_count-1)
    {
        fa[i].count=quotient+remainder;
        fa[i].r = r;
    }
else{
        fa[i].count=quotient;
        fa[i].r = r;
}
if(i==t_count-1)
                {
                    fa[i].count=quotient+remainder;
                    fa[i].r = r;
                }
            else{
                    fa[i].count=quotient;
                    fa[i].r = r;
                }
```

3. Δημιουργία των συναρτήσεων που θα εκτελέσουν τα νήματα. Επειδή η συνάρτηση που δέχεται ως όρισμα η pthread_create πρέπει να είναι της μορφής **void *func(void *args)** ορίσαμε δύο καινούργιες συναρτήσεις την **void *write_op(void *arg)** και την **void *read_op(void *arg)** (γραμμές 96 και 111 bench.c) οι οποίες καλούν την _write_test και _read_test αντίστοιχα. Με τον τρόπο αυτό ικανοποιούμε τον περιορισμό αυτόν.

4. Πέρασμα των παραμέτρων των συναρτήσεων. Επειδή οι συναρτήσεις _write_test και _read_test έχουν δύο παραμέτρους δεν μπορούμε να τις περάσουμε απευθείας στην pthread_create. Έτσι ακολουθήσαμε την μεθοδολογία που παρουσιάζεται στις διαφάνειες του εργαστηρίου για το πέρασμα πολλαπλών παραμέτρων. Ορίσαμε το **struct Func_args** (γραμμές 7-10 bench.c)

```
typedef struct {
        long int count;
        int r;
} Func_args;
```

το οποίο περιέχει τα απαραίτητα ορίσματα. Έπειτα αρχικοποιούμε δυναμικά ένα πίνακα τύπου struct Func_args ο οποίος θα κρατάει τα ορίσματα για κάθε νήμα (γραμμές 149 και 194 bench.c). Μετά περνάμε σε κάθε θέση του πίνακα τα σωστά ορίσματα (γραμμές 157-167 και 202-212 bench.c). Τέλος μέσα στις συναρτήσεις write_op και read_op με την εντολή **Func_args *f = (Func_args*) arg** μπορούμε πλέον να έχουμε πρόσβαση στα ορίσματα που χρειαζόμαστε.

5. Αρχικοποίηση και εκτέλεση των νημάτων. Αρχικά αρχικοποιούμε δυναμικά ένα πίνακα τύπου pthread_t (γραμμές 148 και 193 bench.c) που θα περιέχει τα νήματα. Έπειτα με ένα for loop καλούμε την εντολή pthread_create περνώντας το σωστό αναγνωριστικό νήματος, τα σωστά

ορίσματα και την σωστή συνάρτηση κάθε φορά (γραμμές 172-175 και 217-220 bench.c).

```
for(int i=0; i<t_count; i++)
{
        pthread_create(&threads[i], NULL, write_op,
(void *) &fa[i]);
}
for(int i=0; i<t_count; i++)
{
        pthread_create(&threads[i], NULL, read_op,
(void *) &fa[i]);
}
```

6. Αποδέσμευση των νημάτων. Αφού τα νήματα ολοκληρώσουν την εργασία τους φροντίζουμε να αποδεσμεύσουμε τους πόρους που καταλαμβάνουν με κλήση της συνάρτησης pthread_join (γραμμές 177-180 και 222-225 bench.c).

```
for(int i=0; i<t_count; i++)
{
        pthread_join(threads[i], NULL);
}
```

2.2.2 Υλοποίηση μικτής λειτουργίας

Για την υλοποίηση της μικτής λειτουργίας έπρεπε να συνδυάσουμε τις λειτουργίες των νημάτων put και get με ποσοστά εργασίας του κάθε νήματος. Έτσι δημιουργήσαμε την λειτουργία mixedop δηλαδή μια μίξη των δύο βασικών λειτουργιών

1. Εισαγωγή αριθμού νημάτων που θέλουμε να εκτελεστούν ως όρισμα στην main. Αρχικά ελέγχουμε αν ο χρήστης έχει δώσει τον σωστό αριθμό νημάτων (γραμμές 232-236 bench.c) έχοντας πλέον και ένα επιπλέον όρισμα που είναι το ποσοστό read που δίνει ο χρήστης. Σε διαφορετική περίπτωση τερματίζουμε την λειτουργία του

προγράμματος. Έπειτα κάνουμε το έλεγχο για τον αριθμό νημάτων t_count (γραμμές 240-243 bench.c)για τους ίδιους λόγους που εξηγήσαμε στο 2.2.1.1

2. Ποσοστά των νημάτων. Επιλέξαμε να υλοποιήσουμε την μικτή λειτουργία αξιοποιώντας το ποσοστό των νημάτων που χρησιμοποιούνται για read. Έτσι δημιουργήσαμε την μεταβλητή readers που υπολογίζει τον αριθμό των readers νημάτων η οποία ισούται με τον αριθμό των νημάτων επι το ποσοστό που δίνει ο χρήστης(γραμμές 244-245 bench.c). Αρά π.χ. άμα οι readers κάνουν 60% της δουλείας τότε 40% θα είναι οι writers ώστε να μοιράζεται σωστά η δουλεία.

```
int reader_percentage = atoi(argv[4]);
int readers = (t_count * reader_percentage) / 100;
```

3. Διαμοιρασμός των εργασιών μεταξύ των νημάτων. Στις γραμμές (251-252 και 259-266 bench.c)μοιράζουμε την δουλεία στα νήματα όπως κάναμε και στο 2.2.1.2 για τις read και write λειτουργίες ξεχωριστά .

4. Πέρασμα των παραμέτρων των συναρτήσεων. Αντίστοιχα με τον τρόπο που παρουσιάστηκε στην ενότητα 2.2.1 ακολουθήσαμε την ίδια διαδικασία με χρήση του πίνακα τύπου struct Func_args για να περάσουμε τις σωστές παραμέτρους στις συναρτήσεις που θα καλέσουν τελικά τα νήματα (γραμμές 249 και 257-267 bench.c)

5. Αρχικοποίηση και εκτέλεση των νημάτων και αποδέσμευση. Αρχικά αρχικοποιούμε δυναμικά ένα πίνακα τύπου pthread_t (γραμμές 248 bench.c) που θα περιέχει τα νήματα. Έπειτα με ένα for loop καλούμε την εντολή pthread_create για την δημιουργία νημάτων για τους readers (γραμμές 272-275 bench.c) και ένα for loop για την δημιουργία νημάτων για τους readers (γραμμές 276-279 bench.c).Τέλος αποδεσμεύουμε τους πόρους που χρησιμοποιούσαν τα νήματα όπως κάναμε και στο 2.2.1 (γραμμές 280-283 bench.c).

```
for(int i=0; i<readers; i++)
		{
			pthread_create(&threads[i], NULL,
read_op, (void *) &fa[i]);
		}
		for(int i=readers; i<t_count; i++)
		{
			pthread_create(&threads[i], NULL,
write_op, (void *) &fa[i]);
		}
for(int i=0; i<t_count; i++)
		{
			pthread_join(threads[i], NULL);
		}
```

2.2.3 Χρήση της βάσης δεδομένων ως καθολική μεταβλητή

Κατά την ανάπτυξη της υλοποίησης φτάσαμε στο συμπέρασμα ότι όλα τα νήματα θα πρέπει να αλληλεπιδρούν με το ίδιο instance της βάσης δεδομένων. Στην αρχική υλοποίηση αυτό δεν συνέβαινε διότι η μεταβλητή που ορίζει την βάση ορίζεται μέσα στις συναρτήσεις _write_test και _read_test στο αρχείο kiwi.c. **DB* db** ως global (γραμμή 8 kiwi.c). Επίσης δημιουργήσαμε τις συναρτήσεις **void open_db()** και **void close_db()** (γραμμές 10-18 kiwi.c) οι οποίες χρησιμεύουν στο να μπορούμε να ανοίξουμε και να κλείσουμε την βάση από το κυρίως πρόγραμμα.

```
DB* db;
void open_db()
{
		db = db_open(DATAS);
}
void close_db()
{
		db_close(db);
}
```

Τέλος στην main ανοίγουμε μία φορά την βάση πριν πραγματοποιηθεί οποιαδήποτε άλλη διεργασία (γραμμή 138 bench.c) και την κλείνουμε αφού έχουν ολοκληρωθεί όλες οι άλλες διεργασίες (γραμμή 290 bench.c).

2.2.4 Ταυτοχρονισμός μεταξύ αναγνωστών και γραφέων

Το επίπεδο ταυτοχρονισμού που καταφέραμε να πετύχουμε είναι η υποστήριξη πολλών αναγνωστών να αλληλεπιδρούν με την βάση ταυτόχρονα αλλά επιτρέπουμε μόνο έναν γραφεά να αλληλεπιδρά με την βάση κάθε φορά. Για την υλοποίηση του ταυτοχρονισμού αυτού ακολουθήσαμε τα εξής βήματα:

1. Αρχικά εντός της δομής **typedef struct _db** του αρχείου db.h το οποίο αναπαριστά την οντότητα της βάσης ορίσαμε τις εξής μεταβλητές:
   - **int w_counter:** Μετρητής που κρατάει το πλήθος των ενεργών γραφέων
   - **pthread_mutex_t r_mutex:** Βοηθητική κλειδαριά που απαιτείται για την υλοποίηση της μεταβλητής συνθήκης
   - **pthread_mutex_t w_mutex:** Κλειδαριά για την κρίσιμη περιοχή της εγγραφής δεδομένων στην βάση
   - **pthread_cond_t write_con:** Μεταβλητή συνθήκης για τον συγνχρονισμό μεταξύ αναγνωστών και γραφέων

2. Επειδή αντιμετωπίσαμε errors όταν προσπαθήσαμε να αρχικοποιήσουμε τις παραπάνω μεταβλητές μεταβλητές στατικά εντός του db.h αποφασίσαμε να ακολουθήσουμε τον τρόπο της δυναμικής αρχικοποίησης. Αυτό υλοποιείται με την κλήση των κατάλληλων εντολών μέσα στην συνάρτηση **DB* db_open_ex**

```
pthread_cond_init(&self->write_con,NULL);
pthread_mutex_init(&self->r_mutex, NULL);
pthread_mutex_init(&self->w_mutex, NULL);
self->w_counter=0;
```

3. Αλλαγές στην συνάρτηση **db_add.** Η συνάρτηση αυτή υλοποιεί την εγγραφή δεδομένων στην βάση. Μέσα στην συνάρτηση ορίζουμε την τοπική μεταβλητή res (γραμμή 54 db.c) η οποία

κρατάει το αποτέλεσμα της συνάρτησης db_add για λόγους που θα εξηγήσουμε παρακάτω. Μετά κλειδώνουμε το mutex **w_mutex** έτσι ώστε να εξασφαλίζουμε ότι μόνο ένας γραφέας εισέρχεται στην κρίσιμη περιοχή. Μετά αυξάνουμε κατά ένα τον μετρητή **w_counter** διότι ενεργοποιήθηκε γραφέας και αφού ολοκληρωθεί η διεργασία της εγγραφής στην βάση μειώνουμε κατά ένα τον μετρητή **w_counter** διότι ο ενεργός γραφέας ολοκλήρωσε την δουλειά του και ξεκλειδώνουμε το mutex **w_mutex**. Τέλος αν υπάρχουν ενεργοί γραφείς και στην περίπτωση που δεν υπάρχουν ξυπνάμε τους αναγνώστες με την εντολή **pthread_cond_signal**. Η τοπική μεταβλητή res χρειάζεται διότι η κρίσιμη περιοχή περιέχει την κλήση της εντολής **memtable_add** που στον αρχικό κώδικα η τιμή της απλά γινόταν return στο τέλος της συνάρτησης κάτι που δεν θα επέτρεπε να πάρουμε την σωστή τιμή της συνάρτησης αυτής αφού θα βρισκόταν εκτός κλειδαριών.

```
int res;
pthread_mutex_lock(&self->w_mutex);
self->w_counter++;
            //κρίσιμη περιοχή
self->w_counter--;
pthread_mutex_unlock(&self->w_mutex);
if(self->w_counter==0)
{
              pthread_cond_signal(&self->write_con);
}
return res;
```

4. Αλλαγές στην συνάρτηση **db_get**. Η συνάρτηση αυτή υλοποιεί την ανάγνωση δεδομένων από την βάση. Όταν ένα νήμα εισέρθει στην συνάρτηση αυτή ελέγχει αν υπάρχουν ενεργοί γραφείς και στην περίπτωση που υπάρχουν περιμένει να ολοκληρώσουν την εργασία τους για να συνεχίσει την δική του εκτέλεση. Αυτό επιτυγχάνεται με τον παρακάτω κώδικα:

```
pthread_mutex_lock(&self->r_mutex);
while(self->w_counter!=0)
{
    pthread_cond_wait(&self->write_con, &self->r_mutex);
}
pthread_mutex_unlock(&self->r_mutex);
```

2.2.5 Χρονομέτρηση και στατιστικά επιδόσεων

Στον αρχικό κώδικα η χρονομέτρηση και η εμφάνιση των στατιστικών επιδόσεων γινόταν στο αρχείο kiwi.c εντός των συναρτήσεων _write_test και _read_test. Καταργήσαμε την υλοποίηση αυτή και κάναμε την δικιά μας. Αρχικά ορίσαμε την καθολική μεταβλητή **double total_cost** (γραμμή 5 bench.c) και την κλειδαριά **time_lock** (γραμμή 6 bench.c) την οποία και αρχικοποιούμε. Έπειτα εντός των συναρτήσεων **write_op** και **read_op** ορίζουμε τις μεταβλητές **start end** και **cost** οι οποίες αποθηκεύουν την χρονική στιγμή εκκίνησης της χρονομέτρησης, την χρονική στιγμή λήξης της χρονομέτρησης και τον συνολικό χρόνο που έτρεξε το κάθε νήμα, αντίστοιχα (γραμμές 99-100 και 114-115 bench.c). Αφού έχουμε πάρει τις τιμές των **start** και **end** και έχουμε υπολογίσει το **cost**, κλειδώνουμε την κλειδαριά **time_lock** έτσι ώστε να προσθέσουμε το τοπικό **cost** στο global **total_cost** και να εξασφαλίσουμε συγχρονισμό των νημάτων στην κρίσιμη αυτή περιοχή και κατόπιν ξεκλειδώνουμε την κλειδαριά (γραμμές 105-107 και 120-122 bench.c).

```
long long start,end;
doublecost;
start = get_ustime_sec();
        //εκτέλεση _write_test ή _read_test
end = get_ustime_sec();
cost = end - start;
pthread_mutex_lock(&time_lock);
total_cost+=cost;
pthread_mutex_unlock(&time_lock);
```

Για την εκτύπωση των στατιστικών δημιουργήσαμε την συνάρτηση print_stats
η οποία παίρνει σαν ορίσματα το αριθμό των αιτήσεων (count), τον αριθμό των νημάτων (t_count) και τον τύπο operation που την κάλεσε (type). Μέσω ενός switch κατευθυνόμαστε στην κατάλληλη περίπτωση (1→read, 2→write, 3→mixedop) και τυπώνουμε τα στατιστικά (γραμμές 39-52 bench.c)

Τέλος παρατηρήσαμε ότι αν απλά τυπώνουμε το total_cost στα στατιστικά τότε ο χρόνος εκτέλεσης που εμφανίζεται, αυξανόταν ανάλογα με τον αριθμό νημάτων. Θα εξηγήσουμε για αυτό είναι εν τέλη λάθος. Έστω εκτέλεση του προγράμματος με 4 νήματα. Αν το νήμα 1 ολοκληρώσει τις εργασίες του σε 2 sec, το νήμα 2 σε 2,1 sec, το νήμα 3 σε 1,9 sec και το νήμα 4 σε 1,8 sec τότε το total cost θα είναι 7,8 sec. Όμως αυτές οι εκτελέσεις συμβαίνουν ταυτόχρονα, άρα το total_cost θα έπρεπε να είναι περίπου ίσο με τον μεγαλύτερο χρόνο εκτέλεσης μεταξύ των νημάτων. Μετά από έρευνα που κάναμε για τον σωστό τρόπο χρονομέτρησης παράλληλων προγραμμάτων βρήκαμε ότι ένας δίκαιος τρόπος χρονομέτρησης είναι να παρουσιάζουμε ως χρόνο εκτέλεσης του προγράμματος τον μέσο όρο του total_cost (γραμμές 43,46 και 49 bench.c).

Στατιστικά Επιδόσεων

3.1 Παρουσίαση συστήματος

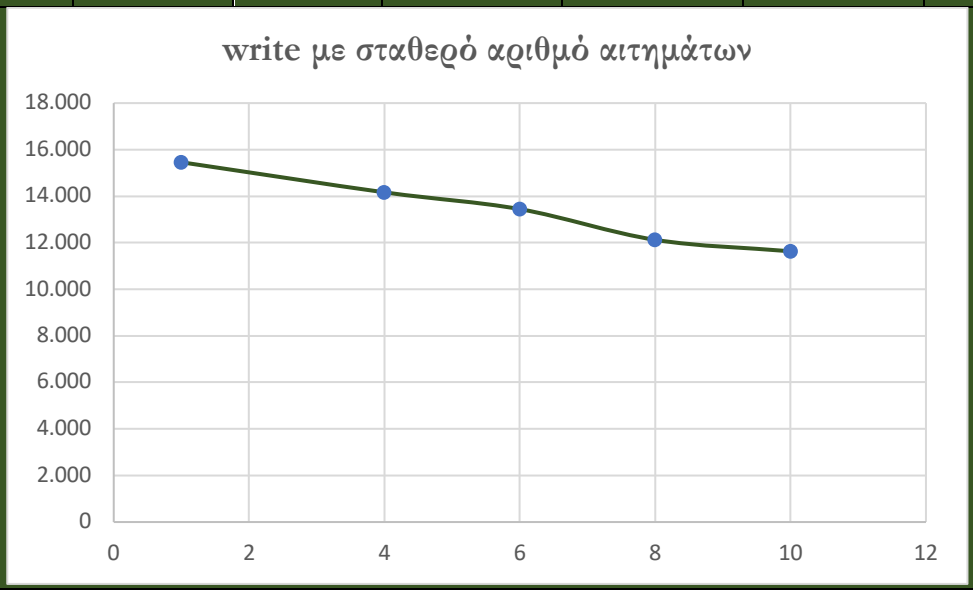
Το σύστημα στο οποίο εκτελέστηκαν τα πειράματα που θα παρουσιαστούν παρακάτω διαθέτει κεντρικό επεξεργαστή με τέσσερεις πυρήνες και 8Gb κύριας μνήμης. Το λειτουργικό σύστημά του είναι το Linux Debian 10.0 και το πρόγραμμά μας μεταφράστηκε με τον GCC 8.3.0.

3.2 Πειράματα

Τα πειράματα που εκτελέσαμε καλύπτουν τις τρεις βασικές λειτουργίες του προγράμματος μας (read operation, write operation και mixed operations). Για κάθε λειτουργία εκτελέσαμε πείραμα με σταθερό αριθμό αιτήσεων και μεταβλητό αριθμό νημάτων και πείραμα με σταθερό αριθμό νημάτων και μεταβλητό αριθμό αιτήσεων. Επιπλέον για την λειτουργία mixed operations εκτελέσαμε ακόμα ένα πείραμα με σταθερό αριθμό αιτήσεων, σταθερό αριθμό νημάτων και μεταβλητό ποσοστό. Κάθε από τις παραπάνω υποπεριπτώσεις εκτελέσθηκε τρεις φορές και υπολογίστηκε ο μέσος όρος των χρόνων των εκτελέσεων ο οποίος παρουσιάζεται γραφικά. Για την χρονομέτρηση χρησιμοποιήθηκε η συνάρτηση **get_ustime_sec()** και οι χρόνοι εκτέλεσης είναι σε δευτερόλεπτα.

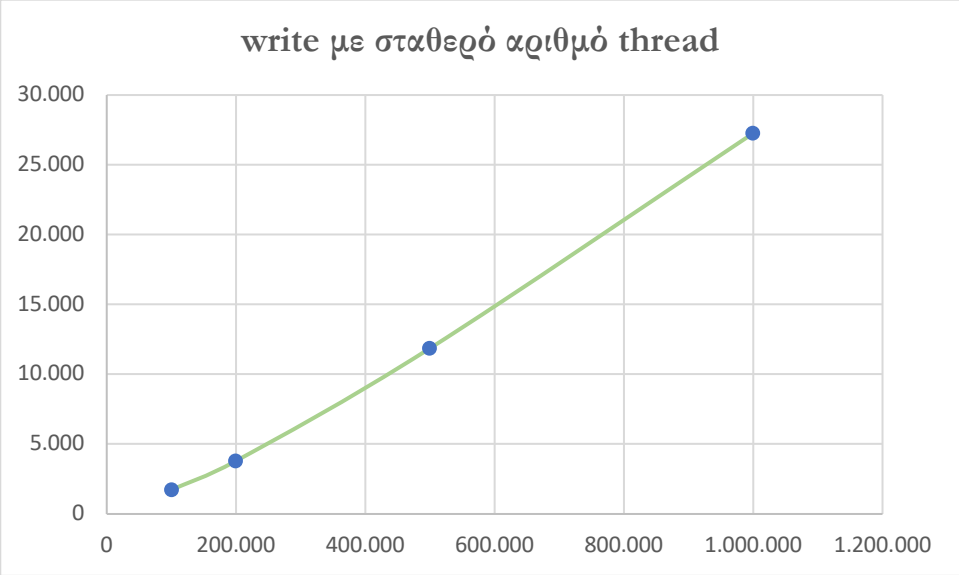
3.2.1 read operation

Operation	Αριθμός Νημάτων	Αριθμός Αιτήσεων	1η Προσπάθεια	2η Προσπάθεια	3η Προσπάθεια	Μ.Ο. Κόστους (sec)
write	1	500.000	15.500	15.250	15.625	15.458
write	4	500.000	12.500	15.250	14.750	14.166
write	6	500.000	12.833	13.667	13.833	13.444
write	8	500.000	10.375	11.125	14.875	12.125
write	10	500.000	11.400	11.200	12.300	11.633



Από τα αποτελέσματα του πειράματος αυτού συμπεραίνουμε ότι όσο αυξάνουμε τον αριθμό των νημάτων τόσο μειώνεται και ο χρόνος εκτέλεσης του προγράμματος.

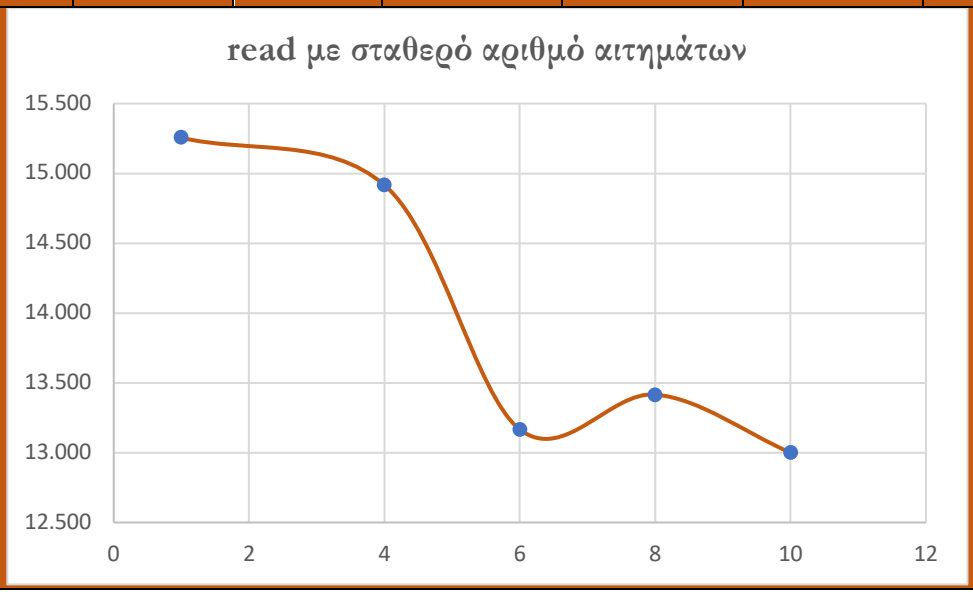
Operation	Αριθμός Νημάτων	Αριθμός Αιτήσεων	1η Προσπάθεια	2η Προσπάθεια	3η Προσπάθεια	Μ.Ο. Κόστους (sec)
write	5	100.000	2.000	1.400	1.800	1.733
write	5	200.000	4.000	3.600	3.800	3.800
write	5	500.000	11.200	11.800	12.600	11.866
write	5	1.000.000	25.800	28.800	27.200	27.266



Από τα αποτελέσματα του πειράματος αυτού συμπεραίνουμε ότι όσο αυξάνουμε τον αριθμό αιτήσεων ο χρόνος εκτέλεσης αυξάνεται σχεδόν γραμμικά. Αυτό σημαίνει ότι η υλοποίηση μας δεν έχει προσθέσει κάποια επιπλέον καθυστέρηση στο πρόγραμμα.

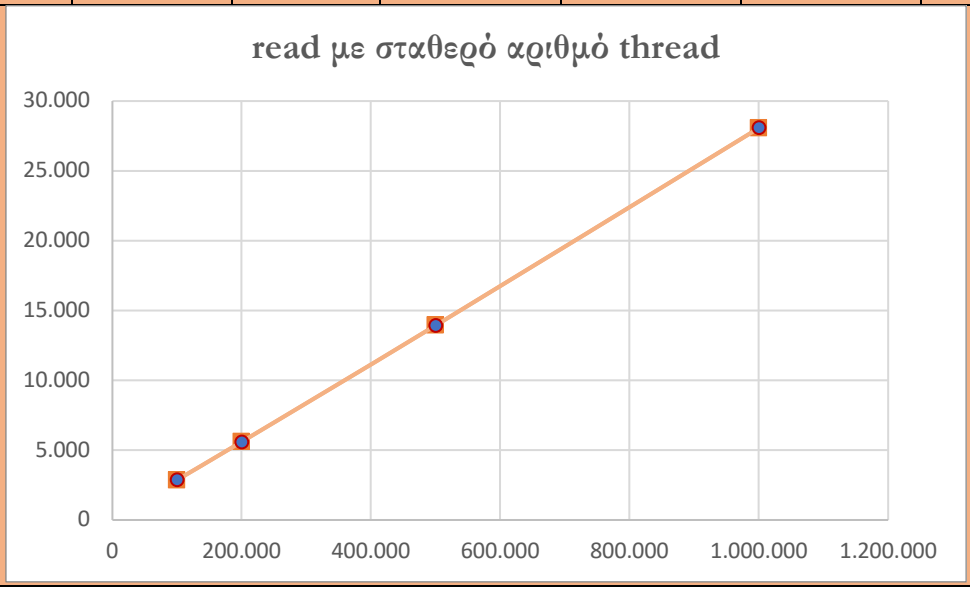
3.2.2 write operation

Operation	Αριθμός Νημάτων	Αριθμός Αιτήσεων	1η Προσπάθεια	2η Προσπάθεια	3η Προσπάθεια	Μ.Ο. Κόστους (sec)
read	1	500000	15.350	15.250	15.166	15.255
read	4	500000	14.500	15.250	15.000	14.916
read	6	500000	12.000	13.667	13.833	13.166
read	8	500000	14.000	12.500	13.750	13.416
read	10	500000	14.500	12.300	12.200	13.000



Από τα αποτελέσματα του πειράματος αυτού συμπεραίνουμε ότι όσο αυξάνουμε τον αριθμό των νημάτων τόσο μειώνεται και ο χρόνος εκτέλεσης του προγράμματος. Επίσης η μείωση του χρόνου εκτέλεσης δεν είναι τόσο μεγάλη όσο στην περίπτωση του read διότι επιτρέπουμε μόνο σε έναν γραφέα να γράφει στην βάση κάθε φορά, πράγμα που μειώνει την συνολική επίδοση του προγράμματος.

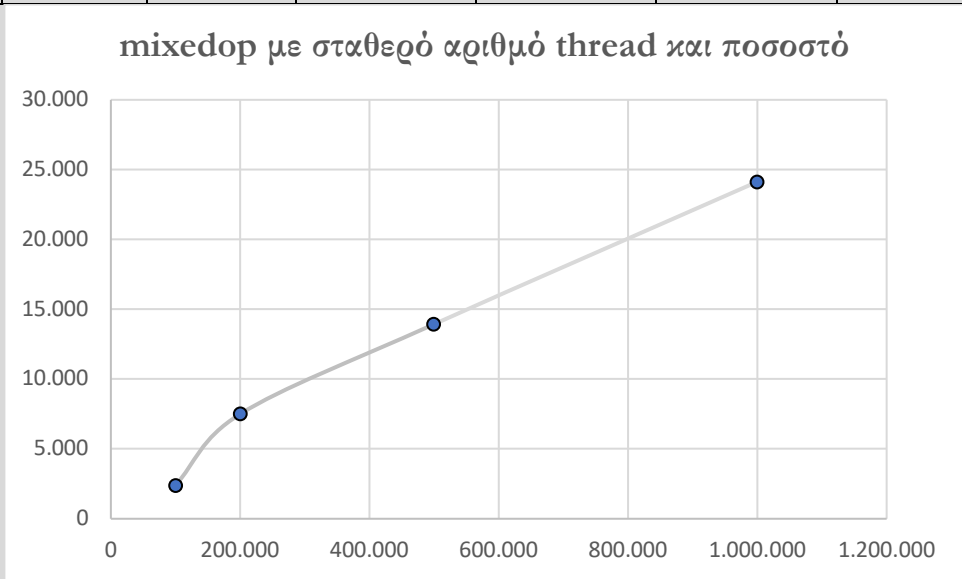
Operation	Αριθμός Νημάτων	Αριθμός Αιτήσεων	1η Προσπάθεια	2η Προσπάθεια	3η Προσπάθεια	Μ.Ο. Κόστους (sec)
read	5	100.000	3.200	2.400	3.000	2.866
read	5	200.000	6.000	5.800	5.000	5.600
read	5	500.000	12.800	15.400	13.600	13.933
read	5	1.000.000	28.000	27.600	28.600	28.066



Από τα αποτελέσματα του πειράματος αυτού συμπεραίνουμε ότι όσο αυξάνουμε τον αριθμό αιτήσεων ο χρόνος εκτέλεσης αυξάνεται σχεδόν γραμμικά, άρα δεν έχουμε προσθέσει κάποια επιπλέον καθυστέρηση στο πρόγραμμα.

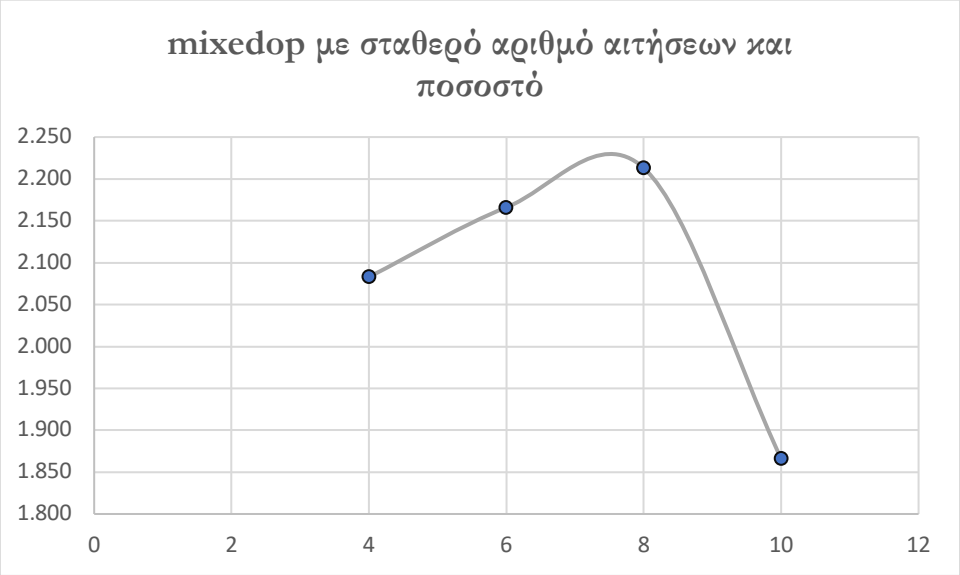
3.2.3 mixed operations

Operation	Αριθμός Νημάτων	Αριθμός Αιτήσεων	Ποσοστό	1η Προσπάθεια	2η Προσπάθεια	3η Προσπάθεια	Μ.Ο. Κόστους(sec)
mixedop	5	100.000	65	2.600	2.400	2.200	2.400
mixedop	5	200.000	65	4.800	5.800	4.400	7.500
mixedop	5	500.000	65	15.400	14.400	12.000	13.933
mixedop	5	1.000.000	65	24.200	23.600	24.600	24.133



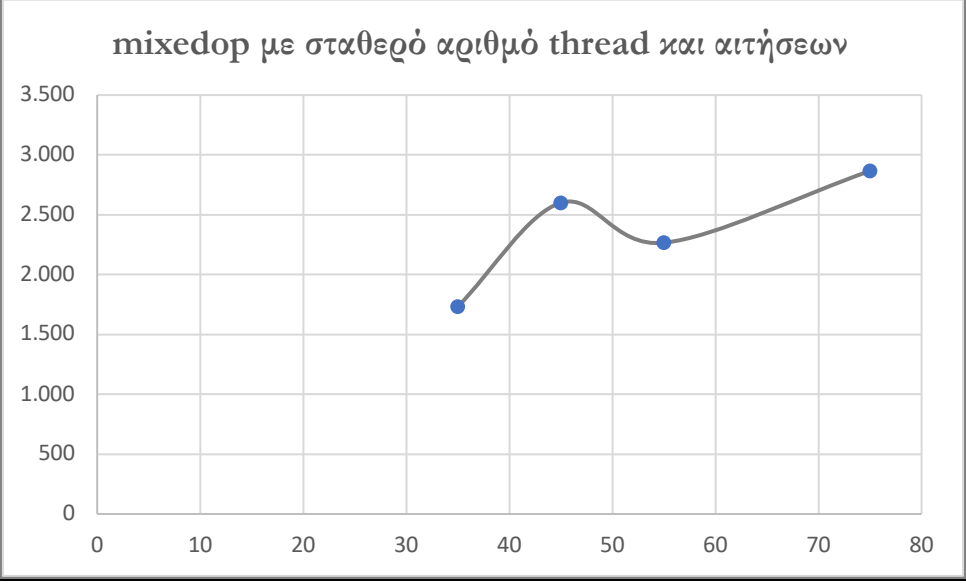
Από τα αποτελέσματα του πειράματος αυτού προκύπτει ότι ο χρόνος εκτέλεσης αυξάνεται σχεδόν γραμμικά καθώς αυξάνεται ο αριθμός των αιτήσεων. Αυτό σημαίνει ότι δεν παρουσιάζεται κάποια επιπλέον καθυστέρηση στην εκτέλεση του προγράμματος όταν αυξάνεται ο αριθμός των αιτήσεων

Operation	Αριθμός Νημάτων	Αριθμός Αιτήσεων	Ποσοστό	1η Προσπάθεια	2η Προσπάθεια	3η Προσπάθεια	Μ.Ο. Κόστους (sec)
mixedop	4	100000	65	1.750	2.500	2.000	2.083
mixedop	6	100000	65	1.500	2.500	2.500	2.166
mixedop	8	100000	65	2.125	2.250	2.265	2.213
mixedop	10	100000	65	1.600	1.600	2.400	1.866



Από τα αποτελέσματα του πειράματος αυτού συμπεραίνουμε ότι ενώ αρχικά ο χρόνος εκτέλεσης αυξάνεται με την αύξηση των νημάτων, τελικά για αριθμό νημάτων μεγαλύτερο των οκτώ μειώνεται δραστικά. Επίσης φαίνεται ο μηχανισμός συγχρονισμού μεταξύ αναγνωστών και γραφέων να επηρεάζει τον χρόνο εκτέλεσης αρνητικά.

Operation	Αριθμός Νημάτων	Αριθμός Αιτήσεων	Ποσοστό	1η Προσπάθεια	2η Προσπάθεια	3η Προσπάθεια	Μ.Ο. Κόστους (sec)
mixedop	5	100000	35	1.800	1.200	2.200	1.733
mixedop	5	100000	45	2.400	3.400	2.000	2.600
mixedop	5	100000	55	2.200	2.200	2.400	2.266
mixedop	5	100000	75	3.400	2.200	3.000	2.866



Από τα αποτελέσματα του πειράματος αυτού συμπεραίνουμε ότι όσο αυξάνεται το ποσοστό των αναγνωστών τόσο αυξάνεται και ο χρόνος εκτέλεσης. Αυτό εικάζουμε ότι συμβαίνει διότι το πλήθος των νημάτων που μένουν ανενεργά όσο οι γραφείς εκτελούν μια εργασία είναι μεγαλύτερο καθυστερώντας έτσι την εκτέλεση του προγράμματος.