Γιώργος Δημόπουλος Α.Μ. 2964

Άγγελος Κατσαλήρος Α.Μ. 2997

1η ΑΣΚΗΣΗ ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

Οι αλλαγές που έγινα για στον **server:**

Αρχικά στο πρόγραμμα μας προσθέσαμε τις εξής βιβλιοθήκες:

- <sys/time.h>
- <stdio.h>
- <pthread.h>
- <unistd.h>

Έπειτα βάλαμε με define τις εξής μεταβλητές για να μπορούμε να τις αλλάζουμε:

- QUEUE_SIZE
- NUM_OF_THREADS

Χρησιμοποιήσαμε 5 Κλειδαριές Αμοιβαίου Αποκλεισμού (Mutexes):

1) pthread_mutex_t for_queue_producer = PTHREAD_MUTEX_INITIALIZER, το χρησιμοποιούμε ώστε να διατηρήσουμε σε συνεπή κατάσταση την ουρά. Δηλαδή ένα νήμα παραγωγού να χρησιμοποιεί την ουρά αλλά ταυτόχρονα να μπορεί και μόνο ένα νήμα καταναλωτή να την χρησιμοποιεί.
2) pthread_mutex_t for_queue_consumer = PTHREAD_MUTEX_INITIALIZER, το χρησιμοποιούμε ώστε να διατηρήσουμε σε συνεπή κατάσταση την ουρά. Δηλαδή ένα νήμα καταναλωτή να χρησιμοποιεί την ουρά αλλά ταυτόχρονα να μπορεί και μόνο ένα νήμα παραγωγού να την χρησιμοποιεί.
3) pthread_mutex_t for_var = PTHREAD_MUTEX_INITIALIZER, ώστε όταν θα πρέπει να χρησιμοποιούμε τις κοινόχρηστες μεταβλητές να μπορεί να τις αλλάζει μόνο ένα νήμα.
4) pthread_mutex_t for_ReaderWriter = PTHREAD_MUTEX_INITIALIZER, ώστε όταν θα χρειαστεί να χρησιμοποιήσουμε τις μεταβλητές reader_count και writer_count να μπορεί να το κάνει μόνο ένα νήμα.
5) pthread_mutex_t for_put = PTHREAD_MUTEX_INITIALIZER, επειδή μπορούμε να κάνουμε ένα-ένα put, πρέπει δηλαδή τα put να εκτελούνται ακολουθιακά “κλειδώνω” με ένα ξεχωριστώ mutex την βάση μου όταν θέλει να κάνει put.

Χρησιμοποιήσαμε 4 Μεταβλητές Συνθήκης (Condition variables):

1. pthread_cond_t non_empty_queue = PTHREAD_COND_INITIALIZER, όταν είναι άδεια η ουρά να μπορεί να στέλνει σήματα στα άλλα νήματα.
2. pthread_cond_t non_full_queue = PTHREAD_COND_INITIALIZER, όταν είναι γεμάτη η ουρά να μπορεί να στέλνει σήματα στα άλλα νήματα.

3. pthread_cond_t writer = PTHREAD_COND_INITIALIZER, όταν χρησιμοποιούμε το reader_count να μπορεί να στέλνει σήματα στα άλλα νήματα.
4. pthread_cond_t reader = PTHREAD_COND_INITIALIZER, όταν χρησιμοποιούμε το writer_count να μπορεί να στέλνει σήματα στα άλλα νήματα.

Υλοποίηση ουράς:

Αρχικά βάλαμε τις μεταβλητές tail,head και count. Η ουρά θα έχει μέσα αντικείμενα τύπου Queue. Το Queue είναι ένα struct το οποίο έχει ένα int πεδίο fd για να ξέρουμε το socket και ένα πεδίο struct timeval για να μπορούμε να μετρήσουμε τους χρόνους που κάνει μια αίτηση στην ουρά. Το μέγεθος της ουράς το ορίζουμε με define μεταβλητή (QUEUE_SIZE). Η ουρά μας δέχεται 2 λειτουργείες(put,get). Για την put δημιουργούμε μια συνάρτηση put η οποία ελέγχει για αρχή με τον count αν είναι άδεια η γεμάτη και μετά βάζει το αντικείμενο μέσα στην ουρά. Επειδή η ουρά είναι κυκλική καθώς αυξάνουμε το tail παίρνουμε το module του tail με mod το μέγεθος της ουράς. Για την get ελέγχουμε με τον counter αν η ουρά είναι άδεια, αν δεν είναι παίρνουμε το αντικείμενο που θέλουμε, έπειτα αυξάνουμε το head κατά ένα και μετά όπως και στο tail επειδή η ουρά είναι κυκλική παίρνουμε το module του head με mod το μέγεθος της ουράς.

Υλοποίηση Παραγωγού:

Ο παραγωγός τρέχει μέσα στην main στην while(1). Για κάθε νέα αίτηση που δέχεται ο παραγωγός βάζει μέσα στην ουρά τον περιγραφέα αρχείου καθώς και τον χρόνο που μπήκε εκείνη την στιγμή, που αυτό μπορούμε να το πετύχουμε με την συνάρτηση **gettimeofday()**. Επειδή πρέπει να διατηρούμε σε συνεπή κατάσταση την ουρά όταν πειράζουμε την ουρά στον *παραγωγό* θα χρησιμοποιούμε τον mutex: pthread_mutex_t for_queue_producer. Έτσι μόνο ένα νήμα *παραγωγού* μπορεί να πειράξει την ουρά κάθε φορά αλλά ταυτόχρονα και μόνο ένα νήμα *καταναλωτή* μπορεί να πειράξει την ουρά. Στον παραγωγό έχουμε μια ειδική περίπτωση που θέλει να βάλει μια αίτηση στην ουρά αλλά η ουρά αυτή την στιγμή είναι γεμάτη. Για αυτό τον λόγο βάζουμε τον παραγωγό να περιμένει μέχρι να του έρθει σήμα ότι άδειασε η ουρά. Αυτό γίνεται με την εντολή **pthread_cond_wait()**, μέσω την μεταβλητής συνθήκης non_full_queue. Όταν ο παραγωγός βάλει κάποιο αντικείμενο στην ουρά στέλνει σήμα μέσω της *non_empty_queue* στους καταναλωτές για να τους “ξυπνήσει”, αν περιμένουν να βάλουν κάποιο άλλο στοιχείο στην ουρά.

Υλοποίηση Καταναλωτή:

Έχουμε ορίσει τους καταναλωτές να τρέχουν μέσα στην συνάρτηση ***consumers()**. Εμείς θέλουμε ΜΌΝΟ ένα νήμα καταναλωτή την φορά να βγάζει μια αίτηση από την ουρά για αυτό τον λόγο θα χρησιμοποιήσουμε το mutex: for queue consumer. Με αυτόν τον τρόπο κανένας άλλος καταναλωτής δεν μπορεί να πειράξει την ουρά αλλά μπορεί να την πειράξει ταυτόχρονα μόνο ένα νήμα από τον παραγωγό και έτσι πετυχαίνουμε αυτό που θέλουμε (να διατηρηθεί η ουρά σε συνεπή κατάσταση). Άμα η ουρά είναι άδεια θα δημιουργηθεί κάποιο πρόβλημα οπότε πρέπει να το λύσουμε και αυτό! Στη περίπτωση που η ουρά είναι άδεια οι καταναλωτές μέσω της συνάρτησης **pthread_cond_wait()** περιμένουν σήμα από τον παραγωγό ότι έχει βάλει ένα αίτημα στην ουρά. Όταν οι καταναλωτές παίρνουν ένα αντικείμενο από την ουρά πρέπει να στείλουν σήμα στον παραγωγό ότι άδειασε ένα στοιχείο στην ουρά. Αυτό επιτυχάνεται με την εντολή **pthread_cond_signal()** με την βοήθεια της μεταβλητής συνθήκης non full queue. Μόλις πάρει ο καταναλωτής πάρει την αίτηση από την ουρά καλούμε την **process_request()** με όρισμα το συγκεκριμένο περιγραφέα αρχείου για να εξυπηρετηθεί. Επίσης μας ζητείτε να υπολογίσουμε τους χρόνους εξυπηρέτησης,αναμονής και πόσες αιτήσεις εξυπηρετήθηκαν. Για αυτό λοιπόν την στιγμή που *βγάζουμε* μια αίτηση από την ουρά καλούμε την συνάρτηση **gettimeofday()** και αυτό τον χρόνο το αποθηκεύουμε στην anamoni. Μόλις ο καταναλωτής τελειώσει την αίτηση (αμέσως μετά την κλήση της **process_request()**) ξανακαλώ την συνάρτηση **gettimeofday()** και αποθηκεύω το χρόνο στο support. Πρέπει τώρα να ενημερώσω τις μεταβλητές completed_request, total_waiting_time, total_service_time. Αυτές οι μεταβλητές είναι global οπότε θα χρησιμοποιήσω το mutex: for var και θα λοκάρω τον κώδικα που ενημερώνω τις μεταβλητές ώστε μόνο ένα νήμα να μπορεί να τις πειράξει. Το αντικείμενο dim3 έχει την χρονική στιγμή που μπήκε στην ουρά και η anamoni έχει τον χρόνο που βγήκε από την ουρά. Προφανώς λοιπόν ο συνολικός χρόνος αναμονής στην ουρά είναι η διαφορά τους. Ο χρόνος support είχε την χρονική στιγμή που εξυπηρετήθηκε η αίτηση. Προφανώς λοιπόν ο συνολικός χρόνος εξυπηρέτησης είναι η διαφορά support-anamoni.

Αλλαγές της συνάρτησης process_request:

Τροποποιούμε αυτή την συνάρτηση ώστε να διατηρηθεί η αποθήκη κλειδιού-τιμής. Θα υλοποιήσουμε το πρόβλημα αναγνωστών-γραφέων για PUT και GET.

- ✓ **GET:** Όταν θέλουμε να κάνουμε get οι καναταλωτές πριν την εκτελέσουν θα πρέπει να δουν αν υπάρχουν γραφείς. Στην περίπτωση που δεν υπάρχουν γραφείς τότε το reader count αυξάνεται κατά 1 έτσι ώστε αν έρθει κάποιος γράφεας να μην μπορεί να κάνει κάτι και να περιμένει να τελειώσει ο αναγνώστης. Στην περίπτωση όμως που υπάρχουν γραφείς με την εντολή **pthread_cond_wait()** οι καταναλωτές περιμένουν μέχρι να τελειώσουν οι γραφείς. Τις παραπάνω ενέργειες πρέπει να τις κάνουμε lock επειδή πειράζουμε τις κοινόχρηστες μεταβλητές. Για αυτό τον λόγο χρησιμοποιούμε το mutex: for_ReaderWriter. Όταν τελειώσει και η ανάγνωση από την βάση

χρησιμοποιούμε πάλι το mutex: for_ReaderWriter, γιατί πρέπει να μειώσουμε το reader_count κατά 1 και να στείλουμε μήνυμα στους γραφείς με την συνάρτηση **pthread_cond_signal()** ότι ένας αναγνώστης τελείωσε.

- ✓ **PUT:** Όταν θέλουμε να κάνουμε put οι καταναλωτές πριν την εκτελέσουν θα πρέπει να δουν αν υπάρχουν αναγνώστες. Στην περίπτωση που δεν υπάρχουν αναγνώστες τότε το writer_count αυξάνεται κατά 1 έτσι ώστε αν έρθει κάποιος αναγνώστης να μην μπορεί να κάνει κάτι και να περιμένει να τελειώσει ο γραφέας. Στην περίπτωση όμως που υπάρχουν αναγνώστες με την εντολή **pthread_cond_wait()** οι καταναλωτές περιμένουν μέχρι να τελειώσουν οι αναγνώστες. Τις παραπάνω ενέργειες πρέπει να την κάνουμε lock επειδή πειράζουμε κονόχρηστες μεταβλητές. Για αυτό τον λόγο χρησιμοποιούμε το mutex: for_ReaderWriter. Στην put έχουμε μια ειδική περίπτωση μπορούμε να έχουμε ΜΟΝΟ ένα γραφέα την φορά. Για αυτό τον κάνουμε lock τον κώδικα:

```
if(KISSDB_put(db, request->key,request->value))
        sprintf(response_str, “PUT ERROR\n”);
else
        sprint(response_str, “PUT OK\n”);
```

με μια άλλη μεταβλητή συνθήκη: pthread_mutex_t for_put έτσι ώστε να μπορούμε να εκτελούμε τις put ακολουθιακά. Αμέσως μετά που τελειώνει και η γραφή στην βάση χρησιμοποιούμε πάλι το mutex: for_ReaderWriter, γιατί πρέπει να μειώσουμε το writer_count κατά 1 και να στείλουμε μήνυμα στους αναγνώστες με την συνάρτηση **pthread_cond_signal()** ότι ένας γραφέας τελείωσε.

Υλοποιήση σήματος cntl-z:

Για την υλοποίηση του σήματος cntl-z δημιουργήσαμε την συνάρτηση **void sima (int s)** και μέσα σε αυτήν κάναμε τα εξής:

- Ορίσαμε δύο μεταβλητές όπου αποθηκεύουμε τον μέσο όρο αναμονής και τον μέσο όρο εξυπηρέτης.
- Κλείσαμε την βάση μας.
- Τυπώσαμε τις συνολικές εξυπηρετήσεις τον μέσο όρο αναμονής στην ουρά καθώς και τον μέσο όρο εξυπηρέτησης.
- Κάνουμε detach τα νήματα μας όπως μας ζητήθηκε.

Όταν ληφθεί το σήμα cntl-z που αυτό γίνεται είτε πατώντας cntl-z είτε με την εντολή kill -20 (pid του server) θα εκτελεστούν οι παραπάνω ενέργειες.

Οι αλλαγές που έγιναν για τον **client:**

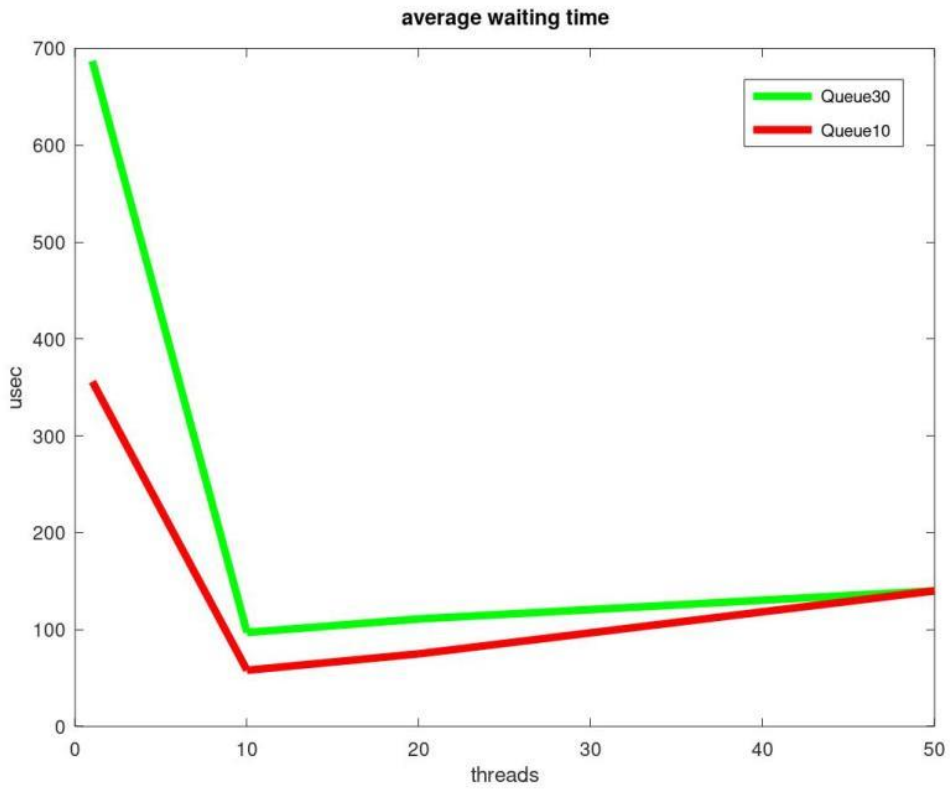
Αρχικά βάλαμε μια define μεταβλητή **#NUM_OF_THREADS** όπου μπορούμε να αλλάξουμε των αριθμών των νημάτων του client ώστε να μπορεί να κάνει πιο πολλές αιτήσεις. Όσα περισσότερα νήματα βάζουμε στον client τόσο πιο πολλές αιτήσεις κάνει. Δημιουργήσαμε δύο συναρτήσεις *client_threads_put (void *args)* και την συνάρτηση *client_threads_get (void *args)*. Σε κάθε μια συνάρτηση τρέχουμε μισά από τα συνολικά νήματα που έχουμε δημιουργήσει. Αυτό το κάνουμε ώστε να μπορούμε να έχουμε ταυτόχρονα put και get. Με αυτόν τον τρόπο ελέγχουμε τους αναγνώστες-γραφείς που έχουμε δημιουργήσει στον server.

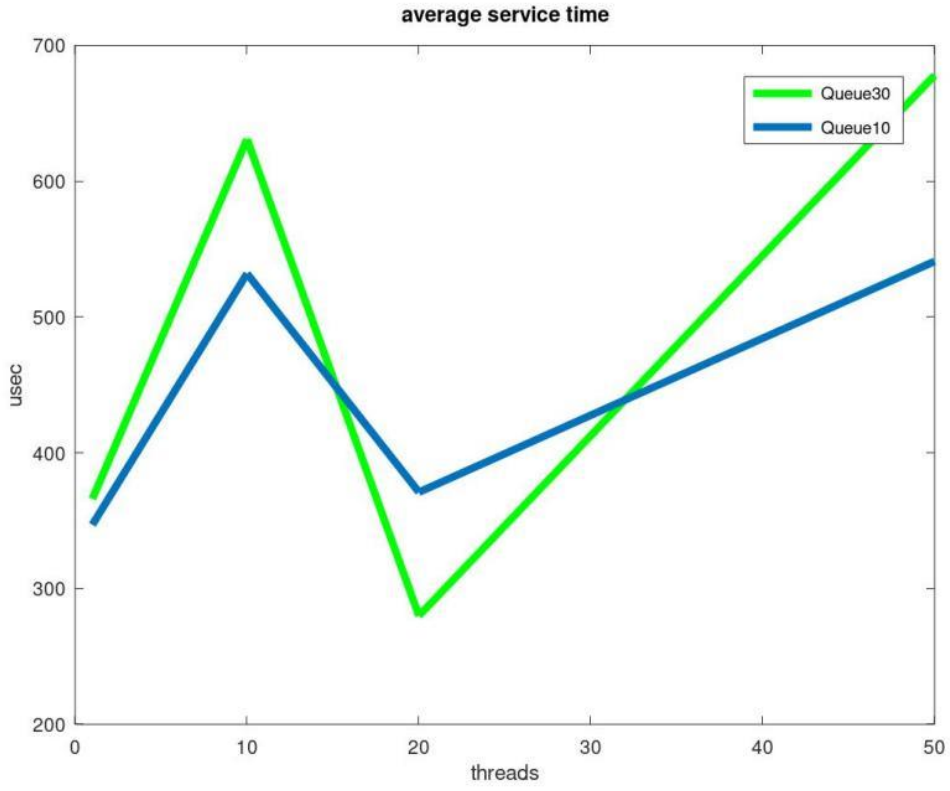
Ουσιαστικά η συνάρτηση *client_threads_put(void *args)* στέλνει αιτήσεις put στον server ενώ η συνάρτηση *client_threads_get(void *args)* στέλνει αιτήσεις get στον server. Επειδή έχουμε φτιάξει δύο συναρτήσεις στις οποίες τα μισά νήματα τρέχουν στην μία συνάρτηση και τα άλλα μισά τρέχουν στην άλλη συνάρτηση, πρέπει να περάσουμε ορισμένες μεταβλητές από την *main* σε αυτές τις συναρτήσεις μέσω των νημάτων. Αυτό το πετυχαίνουμε υλοποιώντας το ***πέρασμα πολλών παραμέτρων των νημάτων φτιάχνοντας ένα struct data.***

Νήματα client	**Νήματα server**	**Μέγεθος ουράς**	**Αριθμός αιτήσεων**	**Average waiting time(usec)**	**Average service time(usec)**
6	1	10	774	356	269
6	1	30	774	687	366
6	10	10	774	75	532
6	10	30	774	97	631
6	20	10	774	47	371
6	20	30	774	111	280
6	50	10	774	59	541
6	50	30	774	140	678
10	10	15	1290	111	336

- Παρατηρούμε ότι όσο πιο πολλά νήματα έχουμε στον client τόσο πιο πολλές αιτήσεις πηγαίνουν στον server.
- Παρατηρούμε ότι όσο πιο πολύ αυξάνεται το μέγεθος της ουράς με τα ίδια νήματα καταναλωτή, το waiting time αυξάνεται.

Παρακάτω αναπαριστούμε τα διαγράμματα όπως μας ζητήθηκε:


average waiting time
Queue30
Queue10
usec
threads
0
100
200
300
400
500
600
700
0
10
20
30
40
50


average service time
Queue30
Queue10
usec
threads
200
300
400
500
600
700
0
10
20
30
40
50