1η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ ΛΕΙΤΟΥΡΚΙΚΑ ΣΥΣΤΗΜΑΤΑ (ΠΟΛΥΝΗΜΑΤΙΚΟΣ ΔΙΑΚΟΜΙΣΤΗΣ)

Ονοματεπώνυμα και Α.Μ ομάδας:

Ηλιάνα Βλάχου 2411
Σπύρος Τσότζολας 3099

ΣΥΝΟΨΗ

Στην παρούσα εργαστηριακή άσκηση μας ζητήθηκε να υλοποιήσουμε έναν πολυνηματικό διακομιστή αποθήκευσης ζευγών κλειδιού-τιμής, με προσέγγιση client-server. Πολλοί clients στέλνουν αιτήματα στον server, ο οποίος θα πρέπει να τους εξυπηρετεί ταυτόχρονα και να τους αποστέλλει το αποτέλεσμα του αιτήματος. Οι clients λαμβάνουν το αποτέλεσμα και το εκτυπώνουν. Επίσης κρατούνται πληροφορίες για το σύνολο των ολοκληρωμένων αιτημάτων, τον συνολικό χρόνο εξυπηρέτησης και αναμονής τους.

SERVER

- ΛΕΙΤΟΥΡΓΙΕΣ

Για την εξυπηρέτηση ταυτόχρονων αιτημάτων στον server χρειάστηκε να δημιουργήσουμε νήματα με δομή παραγωγού καταναλωτή καθώς και μια δομή ουράς η οποία κρατάει τα αιτήματα με την μορφή struct request(περιέχει τον περιγραφέα αρχείου της εισερχόμενης σύνδεσης καθώς και τον χρόνο έναρξής της).

Αρχίκα ο παραγωγός (main loop) περιμένει να του έρθουν νέες συνδέσεις/ αιτήσεις από το client. Μόλις μια αίτηση φτάσει ο παραγωγός είναι υπεύθυνος να εισάγει αυτή στην ουρά και στην συνέχεια να ενημερώσει τον καταναλωτή ότι υπάρχει αίτηση στην ουρά για εξυπηρέτηση, στέλνοντας ενα σημα. Η διαδικασία αυτή επαναλαμβάνεται μεσα σε ένα while(1) με αποτέλεσμα ίσως κάποια στιγμή να γεμίση η ουρά μας. Αν συμβεί αυτό ο παραγωγος θα πρέπει να παραμείνει αδρανές μέχρι κάποιο νήμα καταναλωτή να του στείλει σήμα ότι μια θέση άδιασε στην ουρά, οπότε μπόρει και πάλι να συνεχίσει την παραπάνω διαδικασία.

Ο καταναλωτής από την άλλη αποτελείται από ένα προκαθορισμένο αριθμό νημάτων, τα οποία είναι υπεύθυνα για την εξυπηρέτηση των αιτήσεων αναζήτησης. Συγκεκριμένα, κάθε νήμα καταναλώτη παραμένει αδρανές όσο η ουρά είναι άδεια. Μόλις λάβει το σήμα από τον παραγωγό, το οποιό σηματοδοτεί ότι μία αίτηση έχει εισαγχεί στην ούρα <<ενεργοποιείται>> πάλι και οδεύει πλεόν για την εξυπηρέτηση της αίτησης. Αρχίκα, ελέγχει αν η ουρά είναι γεμάτη. Στην περίπτωση που είναι αφαιρεί την αίτηση από την ουρά και στέλνει σήμα στον παραγωγό να τον ειδοποιήσει ότι υπάρχει πλεόν κενή θέση στην ουρά. Διαφορετικά, αφαιρεί μόνο την αίτηση από την ουρά, χωρις να στείλει σήμα. Στην

συνέχεια, ανακτά τον περιγραφεά αρχείου απο την αίτηση και εκτελεί την διαδικασία της αναζήτησης, καλώντας την process_request() με όρισμα τον περιγραφέα αρχείου που ανακτήθηκε. Στην συνάρτηση αυτή, (process_request) αρχικά γίνεται αναγνώριση του είδους της αιίτησης και στην συνέχεια, αφού εξασφαλιστούν οι περιορισμοί σχετικά με τον αριθμό αναγνωστών και γραφείς απο και προς την αποθήκη κλειδιού-τιμής την ίδια χρονική στιγμή, το νήμα εκτελεί τήν αίτηση. Τέλος, το νήμα στέλνει το αποτέλεσμα της αίτησης στο client και επιστρέφοντας από την proccess_request υπολογίζει και τυπώνει στην οθόνη το χρόνο εξυπηρέτησης και αναμονής της συγκεκριμένης αίτησης. Οι χρόνοι αυτοί προστίθενται στις κοινόχρισες μεταβηητές total_service_time και total_waiting_time αντίστοιχα, ενώ προσαυξάνεται και μια κοινόχρηστη μεταβλήτη completed_requests που περιέχει το πλήθος των αιτήσεων που έχουν εξυπηρετηθεί. Αφού ολοκλήρωσε την δουλεία του το νήμα καταναλώτη περιμένει μήπως χρειαστεί να εξυπηρετήσει στο μέλλον άλλη αίτηση.

Όταν ο χρήστης αποφασίσει να τερματίσει το πρόγραμμα στέλνει το σήμα SIGTSTP (Control+Z). Όταν συμβεί αυτό ο server εκτυπώνει στην οθόνη το πλήθος των αιτήσεων που εξυπηρετήθηκαν, το μέσο χρόνο αναμονής μιας αίτησης στην ουρά, καθώς και το μέσο χρόνο εξυπηρέτησης των αιτήσεων. Στην συνέχεια όλα τα νήματα καταστρέφονται και το πρόγραμμα τερματίζει.

- ΥΛΟΠΟΙΗΣΗ

Για την παραπάνω υλοποίηση χρειάστηκε να προσθέσουμε κάποιες δικιές μας συναρτήσεις, δομές, πίνακες, μεταβλητές, σταθερές καθώς επίσης και κάποια conditionals variables και mutex. Αναλυτικότερα προσθέσαμε τα παρακάτω, διαχωρίζοντας τα σε κατηγοριες:

Συναρτήσεις:

- **int isEmpty():** Ελέγχει αν είναι άδεια η ουρά και επιστρέφει 1, διαφορετίκα 0.
- **int isFull():** Ελέγχει αν είναι γεμάτη η ουρά και επιστρέφει 1, διαφορετίκα 0.
- **void create_consumers():** Δημιουργεί έναν προκαθορισμένο αριθμό νημάτων.
- **void *consumer(void *argv):** Είναι η συνάρτηση που εκτελούν αρχικά τα νήματα καταναλωτών κατά την δημιουργία τους. Επαναληπτικά το κάθε νήμα περιμένει αδρανές μέχρι ένα αίτημα να εισαχθεί στην ουρά και στην συνέχεια το αφαιρεί και καλεί την proccess_request για την εξυπηρέτησή του. Τέλος υπολογίζει και εκτυπώνει το χρόνο αναμονής και εξυπηρέτησης του κάθε αιτήματος .
- **void add_to_queue(int new_fd):** Προσθέτει requests στην ουρά μας.
- **request_describer remove_from_queue():** Αφαιρεί requests από την ουρά.

- **void terminate_communication():** Εκτελείται όταν ληφθεί το σήμα SIGTSTP (Control+Z) . Εκτυπώνει το πλήθος των ολοκληρωμένων αιτήσεων, καθώς και το μέσο χρόνο αναμονής και εξυπηρέτησης τους. Τέλος καλεί την exit() η οποία καταστρέφει τα νήματα και τερματίζει το πρόγραμμα.

Δομές:

- **struct request_describer:** Περιέχει το περιγραφέα αρχείου της εισερχόμενης σύνδεσης και το χρόνο έναρξης της.

Global Values:

- **int head:** Αρχικοποιείται στο 0 και αυξάνεται κάθε φορά που αφαιρείται ένα αίτημα από την ουρά.
- **int tail:** Αρχικοποιείται στο 0 και αυξάνεται με την εισαγωγή ενος αιτήματος στην ουρά.
- **int countReader:** Μετράει τους αναγνώστες μέσα στην αποθήκη κλειδιού-τιμής.
- **int countWriter:** Μετράει τους γραφείς μέσα στην απποθήκη κλειδιού-τιμής.
- **struct timeval total_waiting_time:** Μετράει το συνολικό χρόνο αναμονής των ολοκληρωμένων αιτήσεων.
- **struct timeval total_service_time:** Μετράει το συνολικό χρόνο εξυπηρέτησης των ολοκληρωμένων αιτήσεων.
- **int completed_requests:** Μετράει το πλήθος των ολοκληρωμένων αιτήσεων.

Arrays:

- **pthread_t consumers[NUM_THREADS]:** Περιέχει τα αναγνωριστίκα των pthreads.
- **request_describer my_queue[QUEUE_SIZE]:** Αποθηκεύει τα αιτήματα(ουρά).

Σταθερές:

- **NUM_THREADS:** Το πλήθος των νημάτων – καταναλωτή που δημιουργούνται.
- **QUEUE_SIZE:** Το μέγεθος της ουρας

Conditionals_variables:

- **non_empty_queue:** Χρησιμοποιείται από το παραγωγό για να στείλει σήμα στα νήματα – καταναλωτή όταν ένα αίτημα εισάγεται στην ούρα.
- **non_full_queue:** Χρησιμοποιείται από τα νήματα-καταναλωτή για να στείλουν σήμα στο παραγωγό όταν πλέον η ουρά δεν είναι γεμάτη.
- **get_function:** Χρησημοποιείται από τα νήματα που μόλις ολοκλήρωσαν μια λειτουργία get και εφοσον κανένα άλλο νήμα δεν διαβάζει στην αποθήκη(countReaders=0) στέλνουν σήμα στα νήματα που περιμένουν για γραφή στην αποθήκη, να συνεχίσουν.
- **put_function:** Χρησημοποιείται από τα νήματα που μόλις ολοκλήρωσαν μια λειτουργία put, για να στείλουν σήμα στα νήματα που περιμένουν για ανάγνωση από την αποθήκη.

mutex:

- **queue_lock:** Χρησιμοποιείται για το κλείδωμα των κρίσιμων περιοχών που αναφέρονται στην ουρά (isFull(), isEmpty(), remove_from_queue(), add_to_queue())
- **time_statistics:** Χρησιμοποιείται για το κλείδωμα των κρίσιμων περιοχών που αναφέρονται στις global values (completed_requests, total_waiting_time, total_service_time).
- **rw_mutex:** Χρησιμοποιείται για το κλείδωμα των κρίσιμων περιοχών που αναφέρονται στις global values (countReaders, countWriters). Τα κλειδώματα του συγκεκριμένου mutex γίνονται σε κατάλληλα σημεία στην process_request ωστενα εξασφαλίσουμε οτι μόνο ένας γραφέας μπόρει να τροποποιεί την αποθήκη και όχι ταυτόχρονα γραφείς και αναγνωστές. Αντιθέτως αναγνώσεις μπορούν να γίνονται ταυτόχρονα. Τέλος, έχουμε δώσει προτεραιότητα στους αναγνώστες.
- **socket_lock:** Χρησιμοποιείται για να κλειδώσουμε την περιοχή, όπου να νήματα διαβάζουν από το socket.

CLIENT

- Χρειάστηκε να τροποποιήσουμε το αρχείο client.c ώστε να στέλνει πολλαπλές αιτήσεις ταυτόχρνα. Για να το πετύχουμε αυτό δημιουργήσαμε ένα πανομιότυπο αντίγραφο της τρέχουσας διεργασίας (κλήση της συνάρτησης συστήματος fork()) κάτω απο το σημείο που δημιουργείται το socket address of server στην main. Στην συνέχεια διαχορίσαμε τον πατέρα από το παιδί. Στο παιδί δημιουργούνται ενας προκαθορισμένος αριθμός νημάτων τα οποία είναι υπεύθυνα να στείλουν αιτήσεις στον server, να λάβουν απαντήσεις από το server και να τις τυπώσουν στην οθόνη. Η ίδια ακριβώς διαδικασία γίνεται και στον πατέρα, μόνο που επιπλέον κάνει και waitpid μετά την δημιουργία νημάτων, ώστε να περιμένει την διεργασία παιδί να τερματίσει.

ΜΕΤΡΗΣΕΙΣ ΚΑΙ ΕΠΙΔΟΣΕΙΣ

- Για το πρόγραμμα τρέξαμε κάποια πειραματικά παραδειγματα:

1.

Client NUM_THREADS = 5	Server NUM_THREADS = 10	Server QUEUE_SIZE = 20
GET:		
avg_total_wait_time	0sec	53usec
avg_total_service_time	0sec	902usec
PUT:		
avg_total_wait_time	0sec	90usec
avg_total_service_time	0sec	698usec

2..

Client NUM_THREADS = 15	Server NUM_THREADS = 10	Server QUEUE_SIZE = 20
GET:		
avg_total_wait_time	0sec	53usec
avg_total_service_time	0sec	249usec
PUT:		
avg_total_wait_time	0sec	104usec
avg_total_service_time	0sec	288usec

3..

Client NUM_THREADS = 50	Server NUM_THREADS = 10	Server QUEUE_SIZE = 50

GET:

avg_total_wait_time	0sec	47usec
avg_total_service_time	0sec	147usec

PUT:

avg_total_wait_time	0sec	188usec
avg_total_service_time	0sec	462usec

4..

Client NUM_THREADS = 50	Server NUM_THREADS = 40	Server QUEUE_SIZE = 50

GET:

avg_total_wait_time	0sec	18usec
avg_total_service_time	0sec	540usec

PUT:

avg_total_wait_time	0sec	31usec
avg_total_service_time	0sec	673usec

ΠΑΡΑΤΗΡΗΣΕΙΣ:

- Με βάσει τις παραπάνω μετρήσεις, καθώς και με πολλές αλλές καταλήξαμε στα εξής:

  - Αν κρατάμε σταθερό το πλήθος των νημάτων-καταναλωτή και το μέγεθος της ουρας, καθώς αυξάνουμε το πλήθος των αιτημάτων, τότε το avg_total_service_time μειώνεται.
  - Αν μειώσουμε το πλήθος των νημάτων-καταναλωτή και το μέγεθος της ουράς, τότε το avg_total_service_time και το avg_total_wait_time αυξάνεται.
  - Αν το πλήθος νημάτων-καταναλωτή είναι ίδιο με το μέγεθος της ουράς, τότε έχουμε το μικρότερο δυνατό avg_total_wait_time.

  - Αν αυξήσουμε το μέγεθος της ουράς και κρατήσουμε τον αριθμό των καταναλωτών σταθερό ή το αντίθετο, τότε μειώνεται το avg_total_waiting_time