ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΔΑΣΚΩΝ
ΣΤΕΡΓΙΟΣ ΑΝΑΣΤΑΣΙΑΔΗΣ

1Η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

Υλοποίηση απλού πολυνηματικού διακομιστή αποθήκευσης ζευγών κλειδιού-τιμής

ΠΑΝΑΓΙΩΤΙΔΗΣ ΔΙΚΤΑΜΠΑΝΗΣ ΑΡΙΣΤΕΙΔΗΣ
2323

Πανεπιστήμιο Ιωαννίνων
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ | ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ 2019 - 2020

Πίνακας περιεχομένων

Ξεκινώντας 2
Ζητούμενα 2
Μοντέλα 2
Παραγωγός-Καταναλωτής 2
Αναγνώστες-Γραφείς 3
Για την ουρά 3
Υλοποίηση 4
Διακομιστής (server.c) 4
Σταθερές (#define) 4
Σφαιρικές Μεταβλητές (global variables) 4
Συναρτήσεις 5
Συνθήκες Αμοιβαίου Αποκλεισμού και Μεταβλητές Συνθήκης 6
Για το Σήμα και τη Χρονομέτρηση 9
Πελάτης (client.c) 9
Σταθερές (#define) 9
Συναρτήσεις 9
Μετρήσεις 10

Ξεκινώντας

Η υλοποίησή είχε σα βάση ένα απλό μοντέλο πελάτη-διακομιστή αποθήκευσης ζευγών κλειδιού-τιμής. Ο πελάτης προσδιορίζει το κλειδί για ανάκτηση ή το ζεύγος κλειδιού-τιμής για αποθήκευση. Στη συνέχεια επικοινωνεί με τον διακομιστή μέσω network socket στέλνοντάς του την αίτηση προς εξυπηρέτηση.
Με τη σειρά του ο διακομιστής ανακτά την αίτηση, προσπελάζει τη βάση δεδομένων για ανάκτηση ή αποθήκευση και επιστρέφει το αποτέλεσμα στον πελάτη. Τέλος ο πελάτης τυπώνει το αποτέλεσμα στην οθόνη.
Η δημιουργία της βάσης δεδομένων καθώς και οι λειτουργίες επί αυτής υλοποιούνται με χρήση της βιβλιοθήκης ανοιχτού κώδικα KISSDB και του API του.
https://github.com/adamierymenko/kissdb

Ζητούμενα

Οι βασικές απαιτήσεις της άσκησής μας αφορούν

1. την πολυνηματική υλοποίηση του διακομιστή με χρήση διαφορετικών μοντέλων
2. την πολυνηματική υλοποίηση του πελάτη για ταυτόχρονη αποστολή αιτήσεων και κατά συνέπεια έλεγχο της σωστής λειτουργίας του διακομιστή
3. τη συγκέντρωση στατιστικών στοιχείων για το χρόνο αναμονής και εξυπηρέτησης των αιτήσεων

Η πολυνηματική υλοποίηση βασίζεται στη βιβλιοθήκη POSIX threads του Linux.

Μοντέλα

Στην πολυνηματική υλοποίηση του διακομιστή θα κάνουμε χρήση δύο διαφορετικών μοντέλων.

1. **Παραγωγός-Καταναλωτής**
   Διατηρούμε ένα νήμα παραγωγού το οποίο είναι υπεύθυνο για τη λήψη διαφορετικών αιτήσεων απ' τον πελάτη. Συγκεκριμένα, περιμένει εισερχόμενες αιτήσεις σύνδεσης και με την άφιξη της εκάστοτε αίτησης συντάσσει μια περιγραφή αυτής. Η περιγραφή περιλαμβάνει τον περιγραφέα αρχείου της εισερχόμενης σύνδεσης καθώς και το χρόνο άφιξής της. Στη συνέχεια το νήμα παραγωγού προσθέτει σε μια κοινόχρηστη δομή (ουρά τύπου FIFO) την περιγραφή της σύνδεσης.
   Έχοντας εκ των προτέρων δημιουργήσει έναν αριθμό από νήματα καταναλωτές τα ειδοποιούμε ότι μια νέα αίτηση έχει μπει στην ουρά (μέχρι τότε αυτά περιμένουν αδρανή) και ότι είναι έτοιμη για εξυπηρέτηση. Το νήμα που θα κάνει την εξυπηρέτηση αφαιρεί την περιγραφή απ' την ουρά, ανακτά το κλειδί (στην περίπτωση της ανάκτησης) ή το ζεύγος κλειδιού-τιμής (στην περίπτωση της αποθήκευσης) και εκτελεί την αίτηση. Μετά την ολοκλήρωσή της, υπολογίζει το χρόνο αναμονής της αίτησης στην ουρά, το χρόνο που χρειάστηκε η εκάστοτε

λειτουργία PUT ή GET και ενημερώνει τις αντίστοιχες κοινόχρηστες μεταβλητές οι οποίες αθροίζουν αυτούς τους χρόνους για κάθε εξυπηρέτηση. Παράλληλα ενημερώνει άλλη μια κοινόχρηστη μεταβλητή σκοπός της οποίας είναι η αρίθμηση των διεκπεραιωμένων αιτήσεων. Τέλος, το νήμα καταναλωτή στέλνει το string με το αποτέλεσμα της αίτησης στον πελάτη.

2. **Αναγνώστες-Γραφείς**
   Τα νήματα καταναλωτή κατηγοριοποιούνται σε αναγνώστες ή γραφείς ανάλογα με τη λειτουργία που επιτελούν επί της βάσης. Ένα νήμα είναι αναγνώστης αν κάνει GET και γραφέας αν κάνει PUT.
   Βασική προϋπόθεση της άσκησης είναι να μπορούν να εκτελούνται ταυτόχρονα όσοι αναγνώστες υπάρχουν σε αντίθεση με τους γραφείς που μπορεί να εκτελείται μόνο ένας τη φορά. Επιπλέον δεν μπορούν να δρουν ταυτόχρονα επί της βάσης αναγνώστες και γραφείς.
   Δεν ορίζουμε συγκεκριμένη προτεραιότητα επί των αναγνωστών/γραφέων πράγμα που σημαίνει ότι στην περίπτωση που φτάσει πρώτα αίτηση GET θα επιτραπούν να τρέξουν ταυτόχρονα μόνο αιτήσεις GET. Κάθε αίτηση PUT θα περιμένει μέχρι το πλήθος των αναγνωστών να μηδενίσει.
   Στην περίπτωση που φτάσει πρώτα αίτημα PUT τότε όλα τα υπόλοιπα νήματα περιμένουν μέχρι αυτό να ολοκληρώσει τη δουλεία του.

Για την ουρά

Η ουρά αποτελεί μια κοινόχρηστη δομή μέσω της οποίας επικοινωνούν τα νήματα παραγωγού και καταναλωτή. Ένας παραγωγός τη φορά εισάγει αιτήσεις ενώ ένας καταναλωτής τη φορά εξάγει αιτήσεις. Ένας παραγωγός όμως μπορεί να τροποποιεί την ουρά ταυτόχρονα με ένα καταναλωτή.
Υπάρχει η περίπτωση η ουρά να γεμίσει. Τότε το νήμα παραγωγού περιμένει μέχρι να το ειδοποιήσει κάποιο νήμα καταναλωτή ότι έχει αδειάσει κάποια θέση και άρα να προσθέσει μια καινούρια αίτηση.
Ταυτόχρονα υπάρχει και η περίπτωση η ουρά να αδειάσει. Τότε τα νήματα καταναλωτή περιμένουν μέχρι να τα ειδοποιήσει κάποιο νήμα παραγωγού ότι κάποιο στοιχείο έχει προστεθεί στην ουρά και άρα μπορούν να το εξάγουν για να το εξυπηρετήσουν.
Σημειώνουμε ότι το μέγεθος της ουράς είναι προκαθορισμένο και σταθερό. Διαφορετικά μεγέθη θα χρησιμοποιηθούν για τον έλεγχο την επίδρασης τους στο χρόνο αναμονής και εξυπηρέτησης των αιτήσεων.

Υλοποίηση

Στη συνέχεια θα περιγράψουμε με λεπτομέρεια τις διαφορετικές πτυχές της υλοποίησης του πολυνηματικού πελάτη & διακομιστή, τις δομές και τις συναρτήσεις που χρησιμοποιήθηκαν, τις συνθήκες αμοιβαίου αποκλεισμού και τη θέση τους στον κώδικα.

Διακομιστής (server.c)

Σταθερές (#define)

1. **NUMBER_OF_CONSUMER_THREADS** που καθορίζει το πλήθος των νημάτων καταναλωτών.
2. **QUEUE_SIZE** που καθορίζει το μέγεθος της ουράς.
3. **BILLION** που ορίστηκε ως 1000000000 για ευκολία στη μετατροπή των χρόνων στις μετρήσεις.

Σφαιρικές Μεταβλητές (global variables)

1. **pthread_t thread_id[NUMBER_OF_CONSUMER_THREADS]**
   πίνακας με το thread_id που επιστρέφει η pthread_create για κάθε νήμα.
   Χρειάστηκε να είναι global για τη χρήση του στην pthread_join στον signal handler
2. **double total_waiting_time = 0**
   Ο συνολικός χρόνος αναμονής των αιτήσεων στην ουρά.
3. **double total_service_time = 0**
   Ο συνολικός χρόνος εξυπηρέτησης των αιτήσεων. Πόσο χρόνο χρειάστηκε δηλαδή η πράξη επί της βάσης δεδομένων.
4. **int completed_requests = 0**
   Το πλήθος των αιτήσεων που εξυπηρετήθηκαν.
5. **int reader_count = 0**
   Το πλήθος των ενεργών νημάτων που εκτελούν λειτουργία ανάγνωσης επί της βάσης.
6. **int writer_count = 0**
   Το πλήθος των ενεργών νημάτων που εκτελούν λειτουργία εγγραφής επί της βάσης.
7. **int stop = 0**
   Μεταβλητή που θα χρησιμοποιηθεί από τον signal handler για να σηματοδοτήσει το τέλος εκτέλεσης των νημάτων καταναλωτών.
8. **connection_info queue[QUEUE_SIZE]**
   Η δομή ουράς που θα χρησιμοποιήσουμε για επικοινωνία των νημάτων παραγωγού με τα νήματα καταναλωτές.
   Σημείωση: ο τύπος connection_info είναι δομή τύπου struct που ορίζει αυτό που παραπάνω αναφέρω ως «περιγραφή της σύνδεσης». Περιέχει δύο μεταβλητές:
   int fd ο file descriptor που επιστρέφει η accept() στη main().
   struct timespec connection_start ο χρόνος άφιξης της εκάστοτε σύνδεσης.
9. **int queue_is_empty = 1**
   Flag μεταβλητή για τον έλεγχο άδειας ουράς
10. **int queue_is_full = 0**
    Flag μεταβλητή για τον έλεγχο γεμάτης ουράς.
11. **int head = 0**
    Αυξάνεται για κάθε στοιχείο που εξάγεται απ’ την ουρά. Με κάθε κλήση dequeue(),

head++. Σημειώνω ότι το head αυξάνεται κυκλικά δηλαδή κάθε φορά που φτάνει το QUEUE_SIZE μηδενίζει. Πρακτικά η μεταβλητή head δείχνει ποιο στοιχείο της ουράς θα εξάγω.

12. **int tail = 0**
    Αυξάνεται για κάθε στοιχείο που εισάγω στην ουρά. Όμοια με το head, αυξάνεται κυκλικά και υποδηλώνει σε ποια θέση της ουράς θα εισάγω το επόμενο στοιχείο.
13. **int items_in_queue = 0**
    Μετρητής των στοιχείων που περιλαμβάνει κάθε στιγμή η ουρά. Χρειάζεται για τον έλεγχο γεμάτης και άδειας ουράς.

Συναρτήσεις

1. **void print_globals(char *called_by)**
   Αποκλειστική χρήση για αποσφαλμάτωση του κώδικα της ουράς.
   Τυπώνει όλες τις global μεταβλητές που αφορούν την ουρά (μεταβλητές 9 με 13 της προηγούμενης ενότητας).
2. **void enqueue(connection_info new_connection_info)**
   Ελέγχει αν το tail % QUEUE_SIZE είναι μηδέν. Αυτό θα συμβεί στην περίπτωση που το tail == 0 ή tail == QUEUE_SIZE. Σε αυτές τις περιπτώσεις το tail γίνεται (ή παραμένει) μηδέν (και ξαναρχίζει τον κύκλο). Στη συνέχεια προσθέτει μια νέα περιγραφή σύνδεσης στην ουρά και αυξάνει το tail και το items_in_queue κατά 1.
3. **connection_info dequeue()**
   Ελέγχει αν το head % QUEUE_SIZE είναι μηδέν. Αυτό θα συμβεί στην περίπτωση που το head == 0 ή head == QUEUE_SIZE. Σε αυτές τις περιπτώσεις το head γίνεται (ή παραμένει) μηδέν (και ξαναρχίζει τον κύκλο). Στη συνέχεια αυξάνει το head και μειώνει το items_in_queue κατά 1. Τέλος επιστρέφει το στοιχείο της ουράς που έδειχνε το head όταν κλήθηκε η dequeue().
4. **int check_if_queue_is_empty()**
   Επιστρέφει 1 για άδεια ουρά, αλλιώς μηδέν.
5. **int check_if_queue_is_full()**
   Επιστρέφει 1 για γεμάτη ουρά, αλλιώς μηδέν.
6. **void signal_handler(int sigid)**
   Χειρίζεται στο σήμα SIGTSTP (CTRL + Z). Το σήμα «πιάνεται» από το νήμα παραγωγό και καλεί τη signal_handler για το χειρισμό του. Με την κλίση της κάνει τη global μεταβλητή stop = 1 τερματίζοντας το βρόγχο των νημάτων καταναλωτή. Τυπώνει τα στατιστικά που συγκεντρώθηκαν κατά την εκτέλεση του διακομιστή, κλείνει τη βάση (KISSDB_close(db) [κλήση API της βάσης]) και τερματίζει το πρόγραμμα.
7. **Request *parse_request(char *buffer)**
   Συνάρτηση που παρέχεται στη βασική υλοποίηση του μοντέλου πελάτη-διακομιστή. Στην ουσία «σπάει» το αίτημα που φτάνει απ' τον πελάτη σε κομμάτια (tokens) και αρχικοποιεί μια μεταβλητή τύπου Request την οποία και επιστρέφει. Η μεταβλητή Request (υλοποιημένη σα struct) είναι επίσης τμήμα της βασικής υλοποίησης και περιέχει τα τμήματα της αίτησης. Το είδος της (PUT ή GET), το κλειδί και την τιμή.

8. **void process_request(void *arg)**
   Η συνάρτηση που εκτελούν τα νήματα καταναλωτή. Χονδρικά, τα νήματα περιμένουν έως ότου η ουρά πάψει να είναι άδεια. Στη συνέχεια ένα ένα δρουν επί της βάσης εξάγοντας μια περιγραφή κάποιας σύνδεσης. Ενημερώνουν το νήμα καταναλωτή ότι η ουρά έπαψε να είναι γεμάτη (και άρα να συνεχίσει με την προσθήκη νέων περιγραφών αιτήσεων) και εξυπηρετούν την αίτηση που έλαβαν όπως περιγράφηκε στην ενότητα Αναγνώστες-Γραφείς. Τέλος ενημερώνει τις μεταβλητές χρονομέτρησης και κλείνει το file descriptor της αίτησης που εξυπηρέτησε. Περισσότερα για τις συνθήκες αμοιβαίου αποκλεισμού στη συνέχεια.
9. **int main()**
   Η συνάρτηση του νήματος παραγωγού. Αρχικά «στήνει» το network socket κάνοντας τις κατάλληλες κλήσεις σε socket(), bind() και listen(). Στη συνέχεια δημιουργεί τη βάση δεσμεύοντας χώρο γι' αυτήν και κάνοντας την κατάλληλη κλήση στο API της [KISSDB_open()]. Τέλος μπαίνει σε ένα loop (while(1)) στο οποίο δέχεται τις αιτήσεις από τον client μέσω της accept(). Σημειώνουμε ότι σ' αυτό το σημείο δημιουργείται η περιγραφή της σύνδεσης φτιάχνοντας ένα νέο στιγμιότυπο της μεταβλητής connection_info (struct με το fd της accept() και το χρόνο άφιξης του αιτήματος). Αυτή η μεταβλητή εισάγεται στην ουρά και έπειτα από έλεγχο για συνθήκη γεμάτης ουράς, ενημερώνει τα νήματα καταναλωτή να πιάσουν δουλειά. Όπως και για την process_request() θα περιγράψω ακριβώς τις συνθήκες αμοιβαίου αποκλεισμού στην επόμενη παράγραφο.

Συνθήκες Αμοιβαίου Αποκλεισμού και Μεταβλητές Συνθήκης

Νήμα Παραγωγού

Έχουμε να διασφαλίσουμε τρεις λειτουργίες. Η πρώτη αφορά την αναμονή του νήματος στην περίπτωση γεμάτης ουράς. Κάνοντας χρήση της μεταβλητής συνθήκης pthread_cond_t **full_queue_cond_var**, με την είσοδό του στην while(1) το νήμα ελέγχει για γεμάτη ουρά μέσω της συνάρτησης check_if_queue_is_full(). Όσο η συνάρτηση αυτή επιστρέφει 1 το νήμα παραγωγού περιμένει εκτελώντας ***pthread_cond_wait(&full_queue_cond_var, &full_queue_cond_mutex)***. Τη σειριακή είσοδο στον έλεγχο αυτό διασφαλίζουμε με το mutex pthread_mutex_t **full_queue_cond_mutex**.
Έπειτα συμβαίνει η accept() και η πρώτη χρονομέτρηση (η ώρα δηλαδή που το αίτημα έφτασε στον παραγωγό) και σχηματίζεται η περιγραφή της αίτησης.
Η δεύτερη λειτουργία αφορά τη σωστή τοποθέτηση της περιγραφής αυτής στην κοινόχρηστη ουρά. Παρόλο που το νήμα παραγωγού είναι μοναδικό η κλήση της enqueue() επεξεργάζεται global μεταβλητές άρα θεώρησα φρόνιμο να την προφυλάξω με τη χρήση ενός Mutex, το **enqueue_mutex**. Κλειδώνει πριν την κλήση της enqueue και ξεκλειδώνει αμέσως μετά.
Τρίτη και τελευταία λειτουργία είναι η ενημέρωση των νημάτων καταναλωτών ότι η ουρά δεν είναι άδεια και άρα παίρνουν τη σκυτάλη για την εξαγωγή και εξυπηρέτηση των αιτήσεων. Αυτή η ενημέρωση θα γίνει μέσω της μεταβλητής συνθήκης **empty_queue_cond_var**.
Το πλήθος των νημάτων καταναλωτών είναι > 1 οπότε και επιλέγω τη χρήση της *pthread_cond_broadcast(&empty_queue_cond_var)* ώστε να ενημερωθούν ταυτόχρονα και να ανταγωνιστούν μεταξύ τους για το ποιο θα «προλάβει» να εξάγει την αίτηση.

Φυσικά ο έλεγχος αν η ουρά δεν είναι άδεια (και κατά συνέπεια η ενημέρωση των καταναλωτών) περιβάλλεται από το κατάλληλο lock και unlock του mutex **empty_queue_cond_mutex.**

Νήματα Καταναλωτών

Αρχικά, να σημειώσουμε ότι για την επαναχρησιμοποίηση του κάθε νήματος καταναλωτή μετά το πέρας της λειτουργίας του, τοποθετούμε όλη τη λειτουργικότητα της συνάρτησης process_request σε ένα βρόγχο while(!stop). Τα νήματα καταναλωτές συνεχίζουν να διεκδικούν την εξαγωγή αιτήσεων απ' την ουρά μέχρι το νήμα παραγωγός να τα ειδοποιήσει μέσω της κοινόχρηστης μεταβλητής stop ότι το αίτημα που εξυπηρετούν είναι το τελευταίο. Θυμίζω, η μεταβλητή stop γίνεται 1 μετά το σήμα CTRL + Z (βλέπε *Συναρτήσεις, νο6*).

Στη συνέχεια έχουμε να διασφαλίσουμε τη σωστή προσπέλαση των στοιχείων της ουράς. Με την είσοδό τους στο βρόγχο τα νήματα ελέγχουν τη συνθήκη άδειας ουράς. Όσο αυτή η συνθήκη είναι αληθής, περιμένουν. Όταν ο παραγωγός εισάγει κάποια αίτηση στην ουρά τα νήματα καταναλωτές ειδοποιούνται μέσω της μεταβλητής συνθήκης **empty_queue_cond_var**. Στη συνέχεια τα νήματα (όλα πλέων ξύπνια) «εφορμούν» να εξάγουν κάποιο αίτημα απ' την ουρά μέσω της dequeue(). Θέλουμε να διασφαλίσουμε ότι τα νήματα καταναλωτές τροποποιούν την ουρά ανά ένα. Χρησιμοποιούμε το Mutex **dequeue_mutex** για να διασφαλίσουμε αυτή τη λειτουργία.

Εδώ εντοπίζουμε μια ακραία περίπτωση (edge case). Έστω ότι υπάρχει ένα στοιχείο στην ουρά. Δεν είναι άδεια και κατά συνέπεια τα νήματα ξυπνούν και ανταγωνίζονται για τη dequeue(). Το νήμα που μπαίνει πρώτο κλειδώνει το **dequeue_mutex,** κάνει σωστό dequeue() και εκτελεί τη λειτουργία του με το πέρας της οποίας θα ξεκλειδώσει το **dequeue_mutex.** Η ουρά τώρα είναι άδεια αλλά το επόμενο νήμα καταναλωτής θα κλειδώσει το dequeue_mutex και θα πάει να κάνει dequeue() σε μια άδεια ουρά.
Το πρόβλημα αυτό λύνεται με έναν extra έλεγχο μετά το κλείδωμα του dequeue_mutex για άδεια ουρά. Αν είναι άδεια ξεκλειδώνει το dequeue_mutex (για αποφυγή αδιεξόδου) και κάνει continue επιστρέφοντας στην κορυφή του while όπου ο έλεγχος για άδεια ουρά αδρανοποιεί το εκάστοτε νήμα.

Σημειώνω επίσης ότι επιτρέπονται τροποποιήσεις επί της ουράς από ένα νήμα παραγωγό και ένα νήμα καταναλωτή ταυτόχρονα. Για να το διασφαλίσουμε αυτό κάνω χρήση διαφορετικού mutex για την εισαγωγή στοιχείου και διαφορετικό για την εξαγωγή. Έτσι υπάρχει αμοιβαίος αποκλεισμός μεταξύ των νημάτων παραγωγού (ας είναι ένα στη δική μας περίπτωση) και αμοιβαίος αποκλεισμός μεταξύ των νημάτων καταναλωτή αλλά όχι μεταξύ παραγωγού και καταναλωτή.

Μετά την ολοκλήρωση της εξαγωγής ενός νέου στοιχείου απ' την ουρά κάνουμε νέα χρονομέτρηση για τον υπολογισμό του χρόνου που το στοιχείο αυτό έμεινε στην ουρά (*χρονική στιγμή που βγήκε απ' την ουρά – χρονική στιγμή που μπήκε*) και ενημερώνουμε την καθολική μεταβλητή **total_waiting_time**. Τέλος ελέγχουμε για γεμάτη ουρά. Αν μετά την εξαγωγή κάποιας αίτησης η ουρά πάψει να είναι γεμάτη, ειδοποιούμε το νήμα παραγωγό να ξυπνήσει και να προσθέσει μια καινούρια. Αυτό επιτυγχάνεται μέσω της μεταβλητής συνθήκης **full_queue_cond_var**.

Για το συγχρονισμό αναγνωστών γραφέων θέλουμε να πετύχουμε τη λειτουργικότητα που αναφέρεται στην ενότητα Αναγνώστες-Γραφείς.

Αρχικά ένα νήμα καταναλωτής επικοινωνεί με τον πελάτη διαβάζοντας απ' το socket την αίτηση που έχει στείλει (μέσω του file descriptor που επέστρεψε η accept() και αποθηκεύτηκε στην ουρά απ' τον παραγωγό καλώντας τη συνάρτηση read_str_from_socket() με τα κατάλληλα ορίσματα). Ελέγχουμε την εγκυρότητα του μηνύματος και αμέσως ελέγχουμε το πλήθος των γραφέων. Αν έχει φτάσει ήδη κάποια αίτηση PUT τότε το writer_count θα έχει γίνει 1 και κατά συνέπεια όλα τα νήματα καταναλωτές, ανεξαρτήτως του αιτήματος που πάνε να διεκπεραιώσουν, περιμένουν. Μόνο όταν το writer_count γίνει μηδέν θα ειδοποιηθούν απ' τη μεταβλητή συνθήκης **writer_cond_var** για να συνεχίσουν.

Μπαίνουμε λοιπόν στη switch η οποία και αντιστοιχίζει το κάθε αίτημα.

Για GET (αναγνώστης), αρχικά αυξάνω το πλήθος των αναγνωστών (reader_count++). Πρόκειται για κοινόχρηστη μεταβλητή οπότε και περιβάλλεται από τη μεταβλητή αμοιβαίου αποκλεισμού **reader_mutex.**
Το unlock του reader_mutex συμβαίνει αμέσως μετά την προσαύξηση του reader_count καθώς επιτρέπονται πάνω από ένας αναγνώστες τη φορά. Η πράξη επί της βάσης λοιπόν δεν προστατεύεται και κατά συνέπεια μπορεί να εκτελεστεί παράλληλα απ' τα νήματα καταναλωτές. Αμέσως μετά την πράξη αυτή κλειδώνουμε το mutex reader_cond_mutex της μεταβλητής συνθήκης (**reader_cond_var**) και μειώνουμε το reader_count κατά 1. Σ' αυτή τη φάση ελέγχω αν το reader_count έχει γίνει μηδέν οπότε και ειδοποιώ τα νήματα καταναλωτές που περιμένουν σ' αυτή τη συνθήκη να συνεχίσουν (θα είναι αποκλειστικά γραφείς).

Για PUT (γραφέας), ελέγχω αν υπάρχουν ενεργοί αναγνώστες οπότε και περιμένω. Όταν το read_count γίνει μηδέν (ειδοποιείται απ' τη μεταβλητή συνθήκης **reader_cond_var**) ο γραφέας μπορεί να συνεχίσει. Τότε κλειδώνει το mutex **writer_mutex** και αυξάνει το **writer_count** (όλα τα νήματα που θα έρθουν από 'δω και πέρα θα περιμένουν). Με το **writer_mutex** κλειδωμένο, ο γραφέας εκτελεί την πράξη επί της βάσης, κάνει το **writer_count = 0** και ειδοποιεί όλα τα νήματα που περιμένουν στη μεταβλητή συνθήκης **writer_cond_mutex** να συνεχίσουν. Τότε και μόνο ξεκλειδώνει το **writer_mutex** γιατί όπως είπαμε μόνο ένας γραφέας τη φορά μπορεί να δρα στη βάση.

Τέλος το νήμα καταναλωτής, θριαμβευτής, παίρνει το χρόνο μετά το πέρας της λειτουργίας του, υπολογίζει πόσο χρόνο έκανε να εξυπηρετήσει την αίτηση
(*χρόνος μετά την πράξη στη βάση – χρόνος που το αίτημα βγήκε απ' την ουρά*) και γράφει το αποτέλεσμα στο socket. Τέλος ενημερώνει τις κοινόχρηστες μεταβλητές **total_service_time** και **completed_requests**. Είπαμε κοινόχρηστες άρα περικλείονται από το mutex **update_stats_mutex**.
Λίγο πριν κλείσει το loop της while και το νήμα καταναλωτής επιστρέψει στην αρχή για να επαναχρησιμοποιηθεί, κλείνει το file descriptor της αίτησης που εξυπηρέτησε. Σημειώνουμε ότι η ενέργεια αυτή μέχρι πρότινος υλοποιούταν στη main.

Για το Σήμα και τη Χρονομέτρηση

Για το χειρισμό του σήματος SIGTSTP (CTRL + Z) έκανα χρήση της συνάρτησης ***sigaction().***
Η κλήση της γίνεται στη συνάρτηση που θέλουμε να ανακατευθύνει το χειρισμό του σήματος σε μια άλλη (δικιά μας) συνάρτηση. Τα ορίσματα που παίρνει είναι:

1. Το σήμα που θέλουμε να αγνοήσει/ανακατευθύνει
2. Ένα struct τύπου sigaction του οποίου τα πεδία αρχικοποιώ με τη συνάρτηση που θα χειριστεί το σήμα, ένα σετ με τα λοιπά σήματα που θέλω να αγνοήσω (σ' αυτή την περίπτωση δεν είναι κάποιο άλλο άρα sigemptyset(&sact.sa_mask)) και θέτω τα flags = 0
3. Ως τρίτο όρισμα δίνω NULL. Αν δεν ήταν NULL, κατά την επιστροφή θα περιείχε την περιγραφή της παλιάς ενέργειας.

Στη συνάρτηση των νημάτων καταναλωτή κάνω χρήση της ***pthread_sigmask()***.
Ελαφρά διαφορετικός χειρισμός απ' τη sigaction() (αρκεί ένα sigset_t set που περιέχει το/τα σήματα που τα νήματα θα αγνοήσουν και πρώτο όρισμα αντί για handler παίρνει τη μορφή της ενέργειας που θα εκτελέσουν τα νήματα).

Για τη χρονομέτρηση έκανα χρήση της συνάρτησης ***clock_gettime()*** με είδος ρολογιού το CLOCK_REALTIME. Όλες οι μετρήσεις έχουν μετατραπεί σε nanosecond.

Πελάτης (client.c)

Σταθερές (#define)

1. **NUMBER_OF_THREADS**
   Το πλήθος των νημάτων του πελάτη.
2. **REQUESTS_PER_THREAD**
   Το πλήθος των αιτημάτων που θα στείλει κάθε νήμα (για έλεγχο ακραίων περιπτώσεων ταυτοχρονισμού)
3. **MODE**
   Διατήρησα την αρχική λειτουργία του πελάτη (με ένα νήμα). Για MODE = 0 ο πελάτης τρέχει τον αρχικό κώδικα (ορίσματα στη main και ένα thread). Για MODE = 1 ο πελάτης είναι πολυνηματικός και εκτελεί καινούριο κώδικα (τρέχει με ./client).

Συναρτήσεις

1. **void print_usage()**
   Τυπώνει πληροφορίες χρήσης για την αρχική έκδοση του κώδικα.
2. **void talk(const struct sockaddr_in server_addr, char *buffer)**
   Η συνάρτηση υπεύθυνη επικοινωνίας με τον Διακομιστή. Δημιουργεί το socket και συνδέεται με τον υποδοχέα δικτύου.
   Στέλνει το αίτημα στο Διακομιστή μέσω της συνάρτησης write_str_to_socket() (το όρισμα buffer της talk) και περιμένει την απάντηση την οποία δέχεται μέσω της read_str_from_socket(). Τέλος κλείνει τον περιγραφέα αρχείου που ανταποκρίνεται στην παραπάνω σύνδεση.
3. **void *func(void *arg)**
   Η συνάρτηση που εκτελούν τα νήματα στην περίπτωση της πολυνηματικής υλοποίησης. Σαν όρισμα παίρνει τη διεύθυνση του server όπως αρχικοποιήθηκε στη main() και τελικά περνάει ως παράμετρο στην talk().

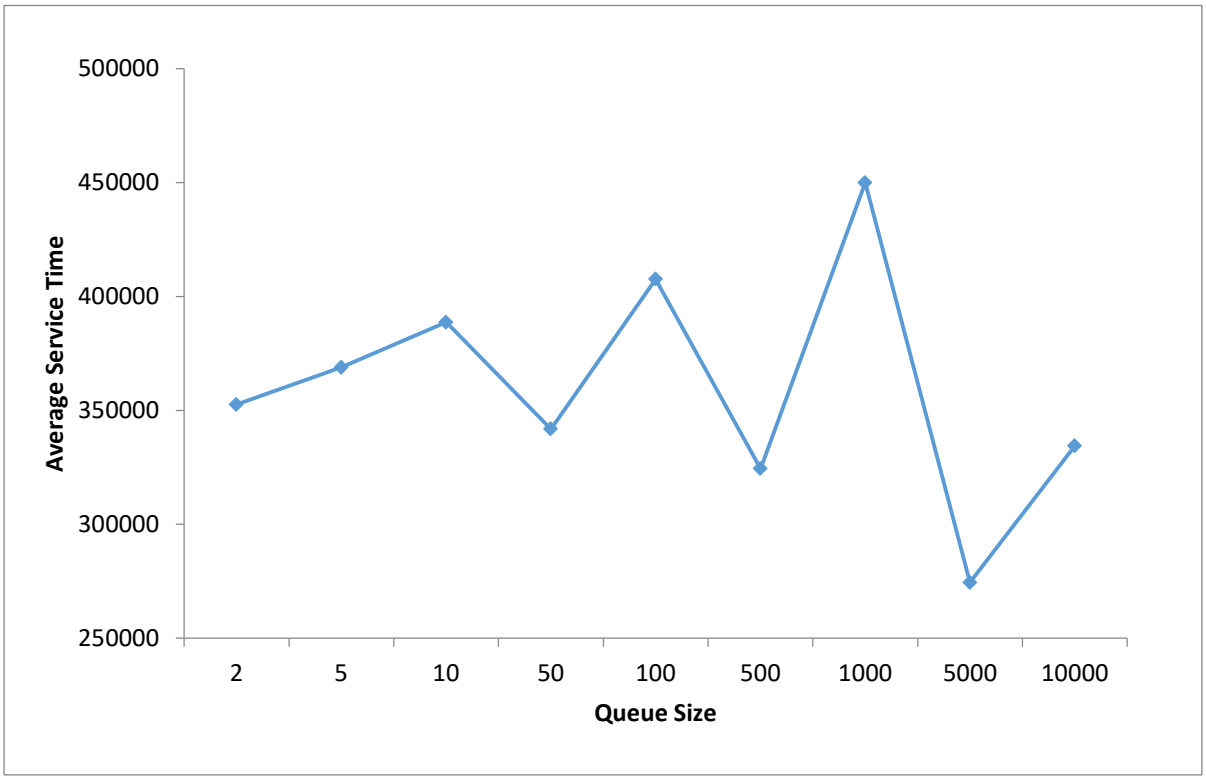
Ανάλογα με το πλήθος των αιτημάτων ανά νήμα (REQUESTS_PER_THREAD) εκτελεί τόσες επαναλήψεις. Σε κάθε επανάληψη, το νήμα, δημιουργεί ένα τυχαίο αίτημα (GET με τυχαίους αριθμούς κλειδιών, PUT με τυχαίους αριθμούς κλειδιών και τιμών). Για να προσομοιώσω την τυχαιότητα του αιτήματος σε κάθε επανάληψη, παράγω έναν τυχαίο αριθμό που μπορεί να είναι 0 ή 1. Αν είναι 0 το αίτημα είναι GET, αν είναι 0 το αίτημα είναι PUT. Αυτή είναι η τελική έκδοση του κώδικα, κατά την υλοποίηση του server εξετάστηκαν τα σενάρια με πολλαπλές GET μόνο ή πολλαπλές PUT μόνο.

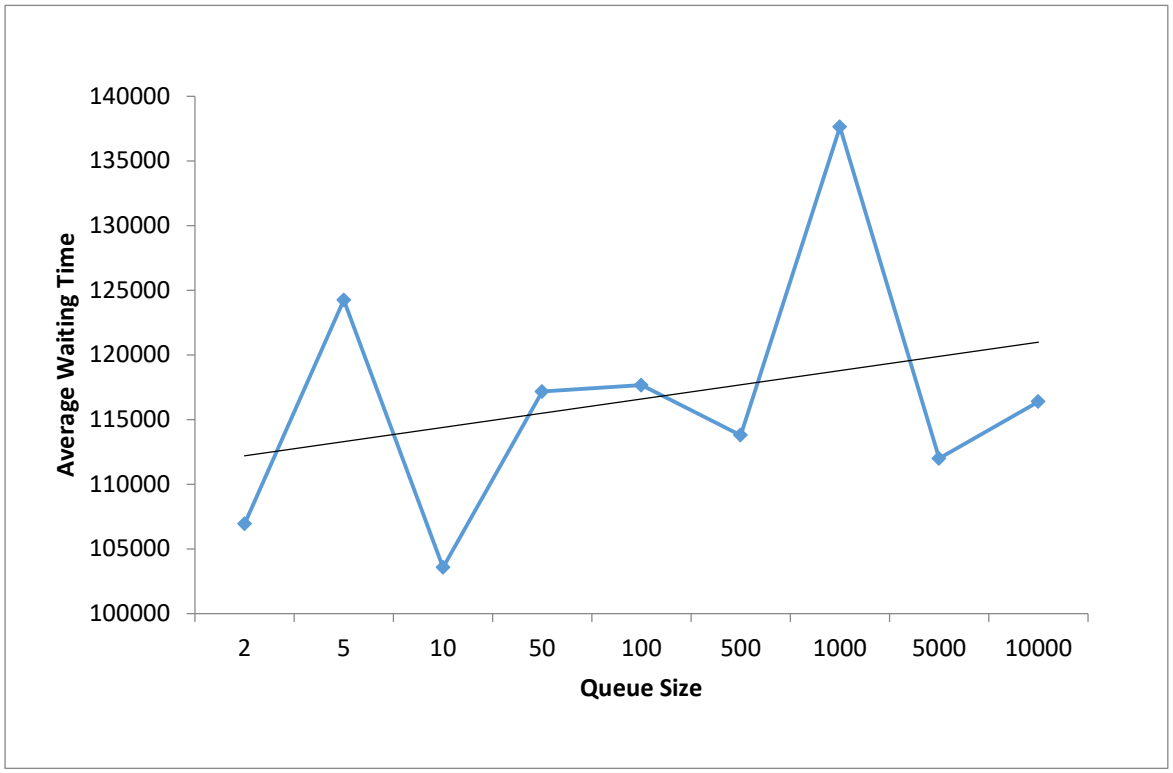
4. **int main(int argc, char **argv)**
   Στην περίπτωση που το MODE είναι 1 και άρα έχουμε την πολυνηματική υλοποίηση του πελάτη, ο host ορίζεται χειροκίνητα ως «localhost». Έπειτα δημιουργείται ένα πλήθος από νήματα που εκτελούν τη συνάρτηση func. Πριν το τέλος γίνεται join ώστε να περιμένουμε τον τερματισμό των νημάτων.
   Κάνω επίσης και μια μέτρηση του χρόνου απ' τη δημιουργία των νημάτων (πριν την pthread_create()) μέχρι και τον τερματισμό τους (μετά το τέλος των pthread_join()) για να έχω μια εικόνα των επιδόσεων του πελάτη.
   Για τη μέτρηση και εδώ χρησιμοποιώ την clock_gettime() με ρολόι το CLOCK_REALTIME. Οι μετρήσεις είναι επίσης σε nanosecond ενώ συμπεριλαμβάνεται κώδικας για μέτρηση σε second με χρήση της clock().

Μετρήσεις

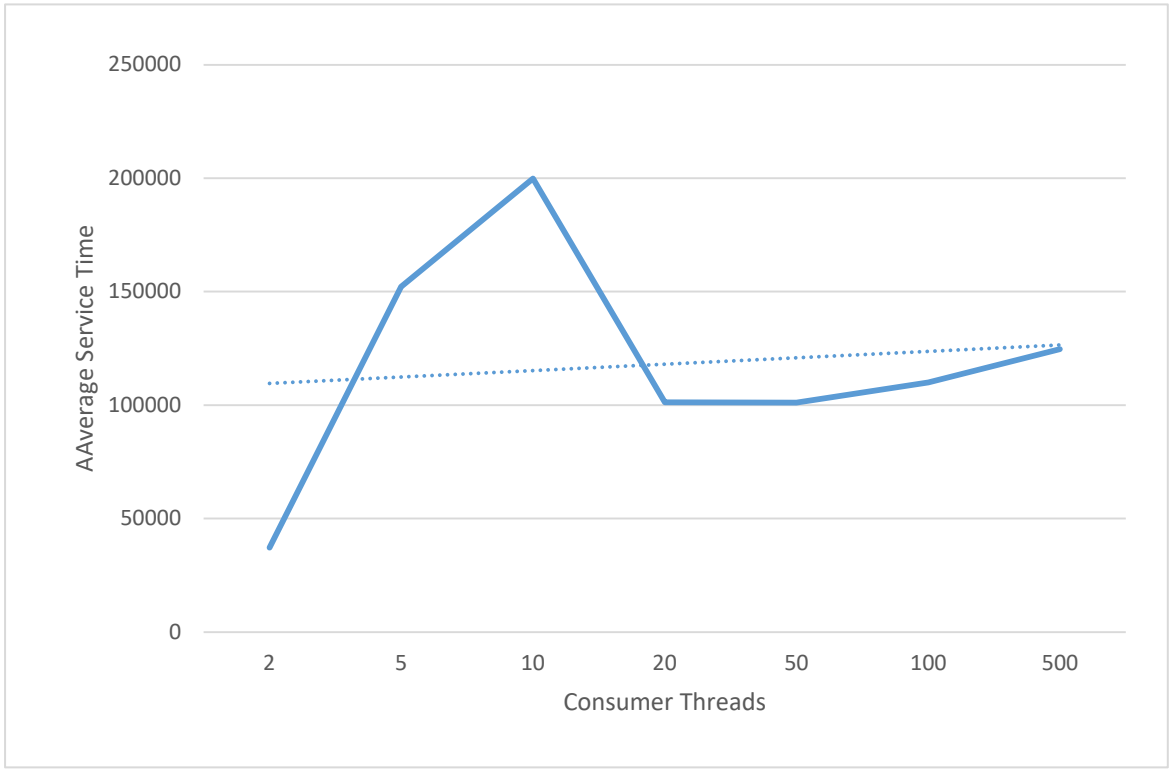
Όλες οι μετρήσεις χρόνου είναι σε nanosecond.

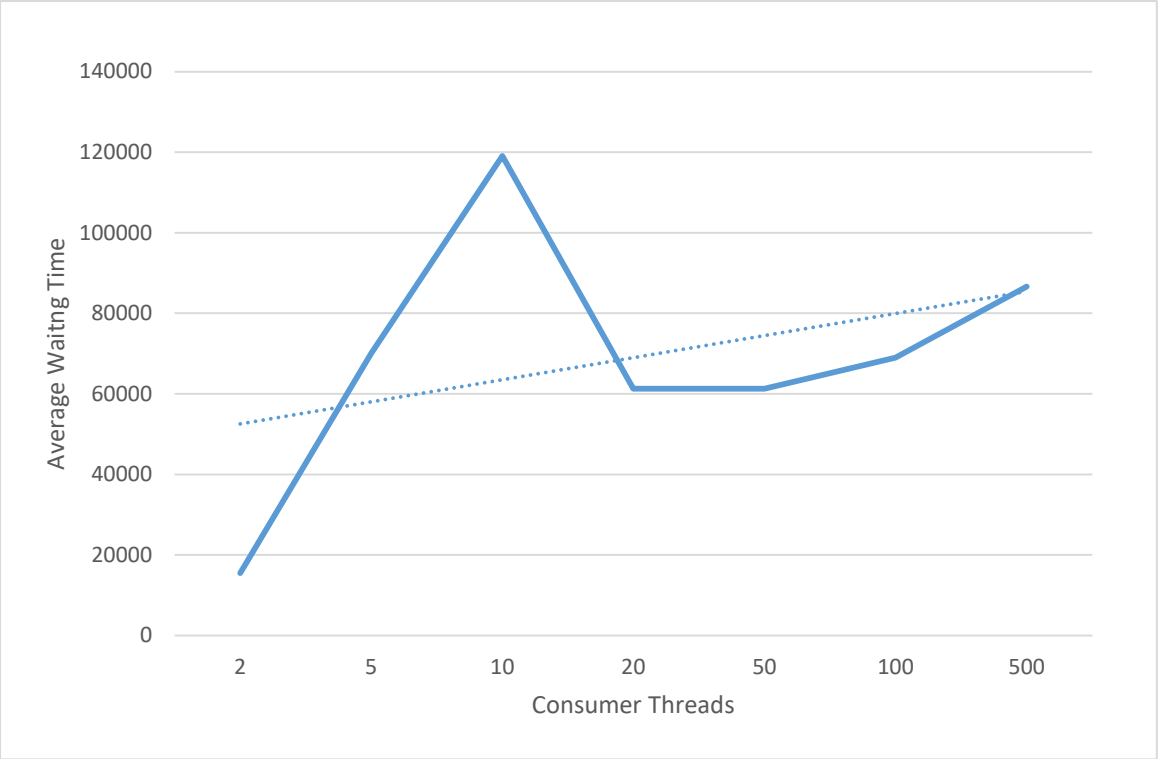
Σταθερό πλήθος νημάτων σε παραγωγό (10) και καταναλωτή (10), κυμαινόμενο μέγεθος ουράς (ο πελάτης έστελνε 70 αιτήσεις σε κάθε περίπτωση).



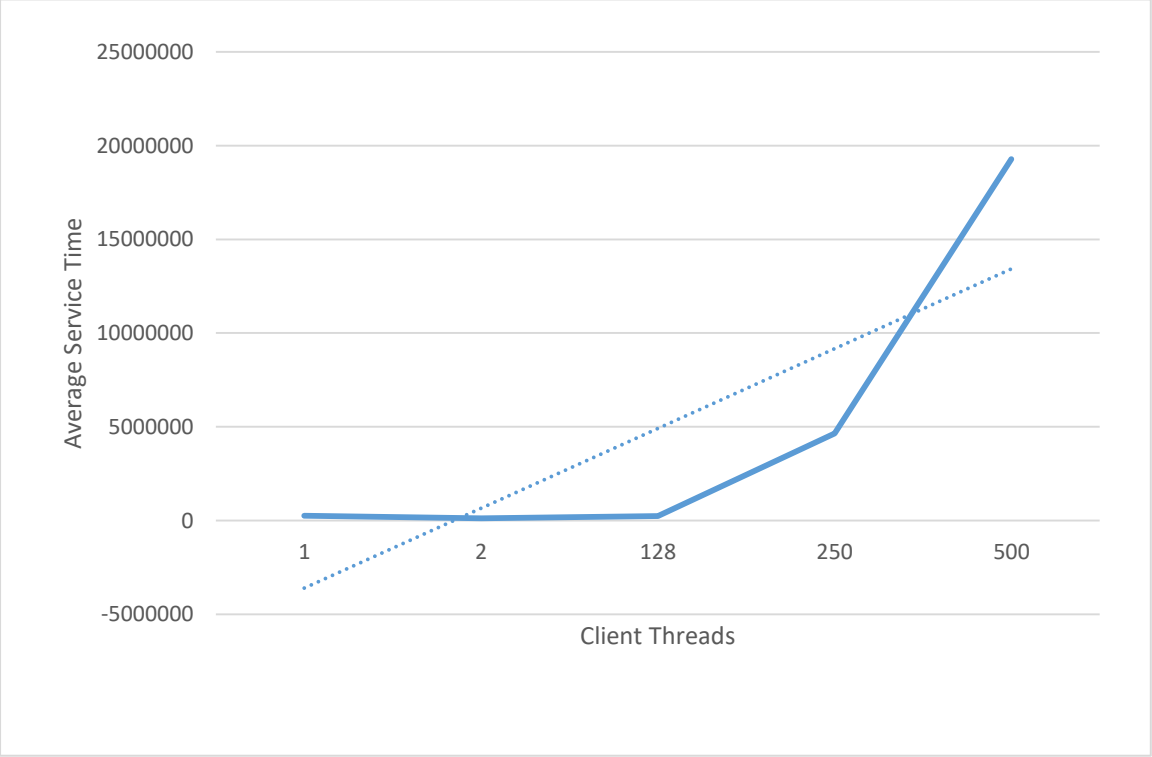


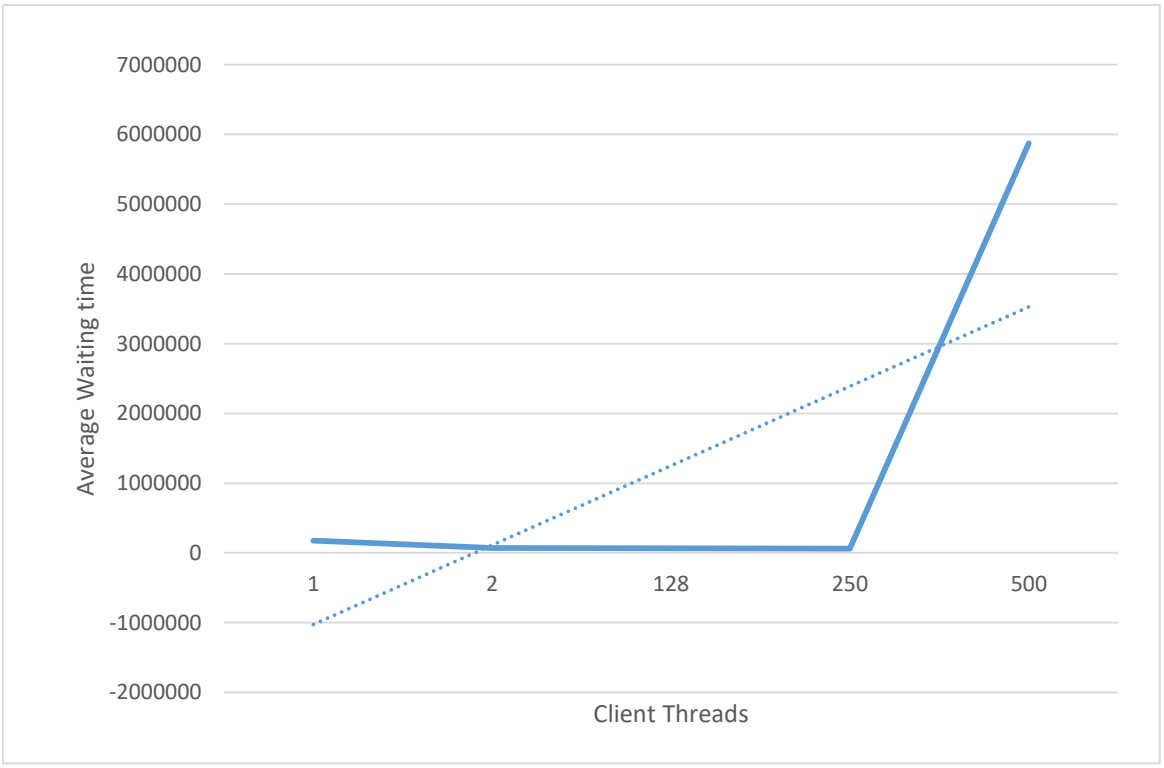
Σταθερό μέγεθος ουράς (500), σταθερό πλήθος νημάτων πελάτη (2) και σταθερό πλήθος αιτήσεων ανά νήμα πελάτη (50) (ο πελάτης έστελνε 100 αιτήσεις τη φορά.





Σταθερό μέγεθος ουράς (500), σταθερό πλήθος νημάτων καταναλωτών (50) κυμαινόμενο πλήθος νημάτων πελάτη, σταθερό πλήθος αιτήσεων ανά νήμα.




7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
-1000000
-2000000
Average Waiting time
1
2
128
250
500
Client Threads