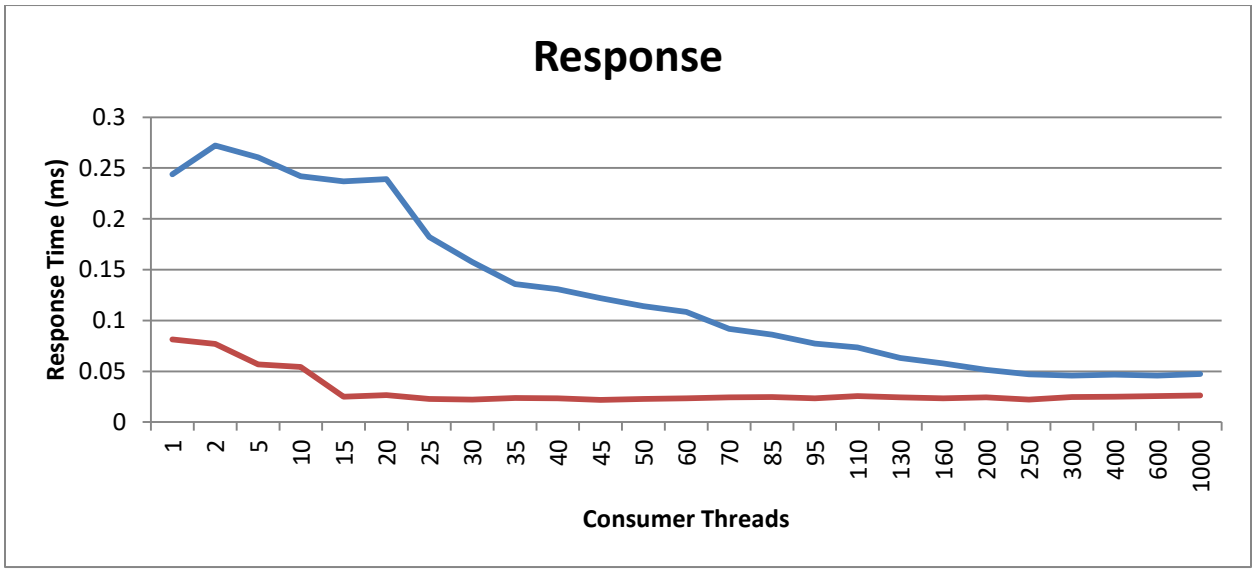
Αναφορά

Ο κώδικας εκτελέστηκε σε υπολογιστή με 4 φυσικούς πυρήνες. Το *configuration* που χρησιμοποιήθηκε ήταν το εξής:

1. Δημιουργήθηκαν 60 *threads* από *producers* (Βάζοντας λιγότερους ή έναν *producer* – κόκκινη καμπύλη στο διάγραμμα- περιμένουμε καλύτερα *response times* ανά τιμή των *consumer threads* μιας και οι *consumers* θα είχαν περισσότερες πιθανότητες να κερδίσουνε το *race* για το *mutex* καθε φορά που θα υπήρχε αντικείμενο στην ουρά)

2. *LOOP* = 7000

και έγιναν δοκιμές με διάφορες τιμές στα *threads* των *consumers* (που θα φανούν στο παρακάτω διάγραμμα) μετρώντας τον μεσο χρόνο αναμονής της καθε "δουλειάς" από τη στιγμή που ο *producer* τη βάζει στην ουρά, μέχρι να την παραλάβει ένας *consumer* (μέσα στην κρίσιμη περιοχή της συνάρτησης του *consumer* καλείται η *queueDel*() όπου και θεωρώ οτι έλαβε τη δουλειά ο consumer και αμέσως μετά κρατάω σε μια τοπική μεταβλητη την χρονική στιγμή και υπολογίζω τον νέο μέσο χρόνο αναμονής στην ουρα. Η εκτέλεση της δουλειάς ξεκινάει αμέσως όταν βγούμε απο την κρίσιμη περιοχή). Προέκυψαν λοιπόν τα εξής αποτελέσματα:



Από το διάγραμμα φαίνεται ότι την καλύτερη απόδοση (στην περίπτωση των 60 *producers*)την παίρνουμε κοντά στα 250 *threads consumers* και μετά όσο αυξάνονται τα αντίστοιχα *threads* δεν έχουμε κάποια βελτίωση.

Στον κώδικα που φαίνεται στο *link* στο τέλος της αναφοράς, έχει αλλάξει η δομή του ***struct*** *workFunc* που δινόταν απο την εκφώνηση με την προσθήκη μιας μεταβλητής ***struct*** *timeval tv,* στην οποία αποθηκεύεται το αποτέλεσμα της ***gettimeofday*()** κατα την είσοδο της δουλειάς στην ουρά απο τον *producer* (ο *producer* καλεί την *queueAdd()* και μέσα στην *queueAdd()* πρόσθεσα την εντολή *gettimeofday(&in.tv,NULL)*). Αποθηκεύεται δηλαδή η στιγμή εισόδου του εκάστοτε *task* στην ουρά. Η αλλαγή αυτή θα μπορούσε να αποφευχθεί, αξιοποιώντας με κατάλληλο τρόπο την υπάρχουσα μεταβλητή ***void** * arg,* όμως με την υλοποίηση που χρησιμοποίησα θεωρώ προέκυψε πιο κατανοητός

και πιο εύκολα διαχειρίσιμος κώδικας, ενώ ταυτόχρονα δεν επηρέασε αρνητικά το υπόλοιπο πρόγραμμα.

Οι συναρτήσεις (*workFunctions*) που υλοποιήθηκαν κάνουν όλες από μία πολύ μικρή δουλειά όπως πχ να εμφανίζουν τη ρίζα του ορίσματος με το οποίο καλούνται (το οποίο όρισμα δίνεται βάσει του *threadID* άρα θεωρείται λιγο πολύ τυχαίο και μπορεί να λάβει διάφορες τιμές), οπότε ο εκάστοτε *consumer* απασχολείται για πολύ μικρό χρόνο καθώς ξεμπερδεύει με τη δουλειά του σε σύντομο χρονικό διάστημα. Μεγαλύτερες συναρτήσεις (πιο *time consuming* δουλειές) θα οδηγούσαν στην ανάγκη για ύπαρξη περισσότερων *consumer threads* για την επίτευξη ελάχιστου *response time* και θα βλέπαμε ότι το παραπάνω διάγραμμα θα λάμβανε την ελάχιστη τιμή του πιο δεξιά. Το ότι στο υπάρχον διάγραμμα βλέπουμε πως μετά απο κάποια τιμή των *consumer threads* η απόκριση παραμένει σταθερή μας λέει πως μετά από ένα σημείο έχουμε πολλούς consumers που «κοιμούνται» και δεν προσφέρουν κάτι στην απόδοση.

Με την υπάρχουσα δομή του κώδικα βλέπουμε πως κάθε φορά μόνο ***ένα thread*** από όλα (*producers* + *consumers*) μπορεί να έχει πρόσβαση στην ουρά. Επίσης, όταν κάποιος *producer* δει πως η ουρά είναι γεμάτη, «κοιμάται» *(pthread_cond_wait())* και περιμένει μέχρι να του πει κάποιος *consumer* ότι η ουρά δεν ειναι γεμάτη ώστε να ενεργοποιηθεί πάλι. Άρα πρακτικά αν προκύψει αύξηση του ρυθμού εισόδου στην ουρά, οι *producers* που δέχονται το μήνυμα "*queue full*" σταματάνε να προσπαθούν να πάρουν το (μοναδικό) *mutex* με αποτέλεσμα να αυξάνονται οι πιθανότητες να το πάρουν οι *consumers* και να αρχίσει να αυξάνει ο ρυθμός εξαγωγής *tasks* απο την ουρά. Σε πραγματικές συνθήκες, στιγμιαία, κάποιες φορές, μπορεί ο ρυθμός εισόδου να είναι μεγαλύτερος από τον ρυθμό εξόδου (και κάποιες φορές αντίστροφα). Γι αυτό θέλουμε μια τιμή *queuesize* τέτοια ώστε στις περιπτώσεις αυξημένου ρυθμού εισόδου, η ουρά να μην προλάβει να γεμίσει, μέχρι να φτάσουμε στην περίπτωση της ισορροπίας ή στην περίπτωση γρηγορότερης εξόδου από την ουρά. Αν το σύστημά μας αδυνατεί να διαχειριστεί επαρκώς τόσα *consumer threads* ώστε να αποφύγει τον αυξημένο ρυθμό εισόδου, τότε δεν υπάρχει νόημα μεγαλύτερης ουράς γιατί είναι δεδομένο ότι όσο μεγάλη και να είναι, θα καταλήξει να γεμίσει. Με τις υπάρχουσες συναρτήσεις και την επεξεργαστική ισχύ του συστήματος που έτρεξε το πρόγραμμα, το *queuesize=10* έκανε τη δουλειά όπως έπρεπε.

Όσον αφορά στην τιμή του *LOOP*, επιλέχθηκε επαρκώς μεγάλη ώστε να ληφθούν επαρκή στατιστικά στοιχεία για να προκύψουν ασφαλή συμπεράσματα ως προς τη μέση τιμή. Κατά την εκκίνηση του προγράμματος επειδή ανοίγουν ένα ένα τα διάφορα *threads* (δημιουργούνται πρώτοι οι *producers* και ξεκινάνε να βάζουν πράγματα στην ουρά χωρίς ακόμα να έχουν δημιουργηθεί *consumers*) έχουμε αυξημένους χρόνους αναμονής στην ουρά, αλλά μετά από λίγο, κατά την κανονική ροή του προγράμματος, η μέση τιμή συγκλίνει σε μία τελική τιμή. Η τιμή 7000 που δώθηκε θα μπορούσε να είναι μικρότερη και να πάρουμε πάλι αντιπροσωπευτική μέτρηση περί της μέσης τιμής αλλά για ακριβέστερη σύγκλιση στην τελική τιμή, επιλέχθηκε έτσι.

Για να επιβεβαιώσω τη σωστή λειτουργία του προγράμματός μου, πρόσθεσα μια μεταβλητή *global int countWorks* η οποία αυξάνει μέσα στην κρίσιμη περιοχή του *consumer* και εκτυπώνεται τη στιγμή που εκτυπώνεται και η ενημερωμένη μέση τιμή χρόνου αναμονής και αναμένω να μου δείξει στο τέλος του προγράμματος τιμή ίση με *LOOP*producers* διότι, όπως ζητήθηκε, η συνάρτηση *producer* έχει μια for loop με τη βοήθεια της οποίας το κάθε *producer thread* , θα εισάγει στην ουρά *LOOP* αντικείμενα *workFunction.*

Link κώδικα: https://github.com/gpappasv/RTES/blob/master/main.c