Ενσωματωμένα Συστήματα Πραγματικού Χρόνου

Εργασία 1

Όνομα: Κίμων Συλαίος
ΑΕΜ: 9395
GitHub repo: https://github.com/AnaxKGS/ESPX

Όπως ζητείται και στην εκφώνηση ο κώδικας ο οποίος μας δόθηκε διαμορφώθηκε κατάλληλα ώστε η **FIFO** ουρά να δέχεται **workFunc structs**. Οι συναρτήσεις που υλοποιήθηκαν είναι απλές (εκτελούν τυχαία τον υπολογισμό ενός ημιτόνου ή συνημιτόνου). Η εκτέλεση των συναρτήσεων που περιλαμβάνονται στα **workFunc structs** της ουράς λαμβάνει χώρα **μετά** το **unlcok** του **mutex** σε κάθε **consumer** με στόχο την παράλληλη εκτέλεση τους.

Έχει κρατηθεί σταθερό το QUEUESIZE και η επιλογή των LOOP, producers και consumers αφήνεται στον χρήστη κατά το run-time, ώστε να διαξαχθούν και πιο έυκολα τα πειράματα που ζητούνται για την σταθεροποίηση των χρόνων αναμονής.

Στους **producers**, χρησιμοποιείται μόνο μία επαναληπτική διαδικασία και έχουν αφαιρεθεί οι καθυστερήσεις usleep(). Επίση, γίνεται χρήση της srand() με βάση το pthread id, ώστε να παράγονται κάθε φορά διαφορετικοί αριθμοί για τα ημίτονα και τα συνημίτονα. Ακόμη, τα tasks και τα arguments ειναι πίνακες μεγέθους LOOP. Αυτό γίνεται ώστε οι νέες τιμές να μην διαγράφουν τις προηγούμενες. Τέλος, όσον αφορά το argument κάθε task, είναι σε ακτίνια και συγκεκριμένα από 0 έως 2π.

Στους **consumers**, οι επαναληπτικές διαδικασίες for αντικαθίστανται από έναν ατέρμων βρόγχο. Ο εκάστοτε consumer εκτελεί κάθε φορά τα tasks που θα βρει. Επίσης, λόγω του ατέρμονα βρόγχου οι consumers δε θα ολοκληρώνουν ποτέ, καθώς θα περιμένουν ένα notEmpty signal. Το σήμα αυτό έρχεται καθέ φορά από τους producers, αλλά δε θα έρθει ξανά αφότου οι producers σταματήσουν να προσθέτουν tasks στην ουρά. Για να λυθεί το πρόβλημα αυτό χρησιμοποιούμε μία σημαία flag η οποία θα γίνεται 1 μετά το join των producers. Τότε δίνεται και το σήμα notEmpty για την ουρά. Έτσι οι consumers που περιμένουν, θα λάβουν το σήμα και θα κλείσουν.

Ώσον αφορά τα **queueAdd, queueDel** και τον **Χρονο Αναμονής**, προσθέτουμε στην struct της queue τον πίνακα stime, ώστε να αποθηκεύεται εκεί η χρονική στιγμή που

μπαίνει ένα task στην ουρά. Έπειτα στην συνάρτηση queueAdd παίρνουμε τη στιγμή αυτή με την χρήση της gettimeofday(). Αντίστοιχα στην συνάρτηση queueDel κρατάμε τον τρέχον χρόνο και αφαιρούμε από αυτόν το αντίστοιχο element του stime. Έτσι προκύπτει ο χρόνος αναμονής.

Πείραμα

Οι μετρήσεις έγιναν σε ένα VM που χρησιμοποιούσε 4 πυρήνες από έναν **AMD Ryzen 1600.**

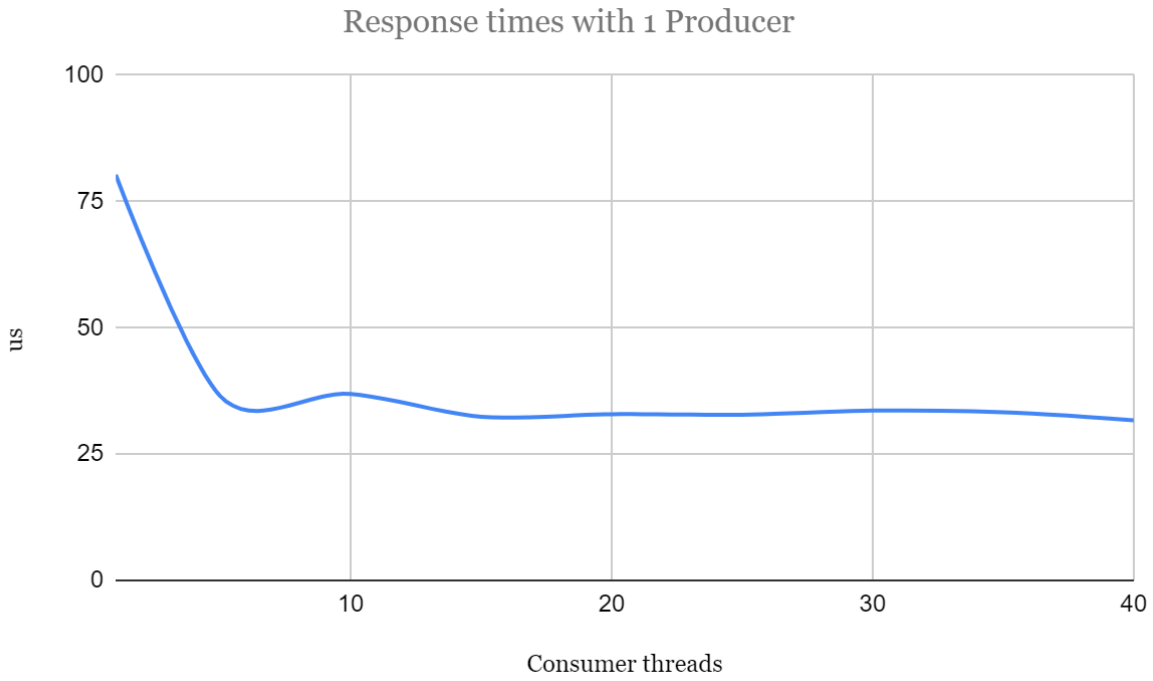
Για το πείραμα χρησιμοποιήσαμε QUEUESIZE = 10, LOOP = 2000, producers = 1 και μετά producers = 80. Κάναμε πειράματα για διάφορους αριθμούς από consumers και σταματάμε να πειραματιζόμαστε όταν βλέπουμε ότι οι αριθμοί σταθεροποιούνται.

Όπως φαίνεται και στο .xlsx αρχείο που βρίσκεται στο GitHub repository, κάθε πείραμα έγινε 10 φορές με την βοήθεια ενός bash script για ευκολία.

Όπως φαίνεται ο χρόνος παραλαβής ενός item από τους consumers είναι σχετικά σταθερός αρχικά, και η πτώση ξεκινά περίπου στα 50 consumers threads. Η αρχική σταθερότητα οφείλεται στο ότι ο αριθμός των consumers είναι μικρότερος από αυτόν των producers, με αποτέλεσμα να προστίθετνται items στην ουρά χωρίς να υπάρχει κάποιος διαθέσιμος καταναλωτής να τα παραλάβει. Από τις μετρήσεις προκύπτει πως ο βέλτιστος αριθμός από consumers στην περίπτωση μας είναι 140 κάτι που

αντικατοπτρίζεται και στο παραπάνω διάγραμμα. Ο αριθμός αυτός επιλέχθηκε επειδή ελαχιστοποιεί το χρόνο παραλαβής και η περαιτέρω αύξηση των threads δε δείχνει σημαντική βελτίωση. Σημειώνεται πως σε μια εφαρμογή με υπολογιστικά ακριβότερες συναρτήσεις, ο βέλτιστος αριθμός από consumer threads ίσως να ήταν μεγαλύτερος καθώς κάθε thread θα απασχολούταν για μεγαλύτερο χρόνο για την διεκπεραίωση του task και έτσι θα αργούσε να «ελευθερωθεί» .

Όμοια συμπεράσματα εξάγονται και στη περίπτωση του ενός producer με βάση τις ακόλουθες μετρήσεις. Ο βέλτιστος αριθμός consumers εδώ φαίνεται να είναι **15**.



Ο κώδικας βρίσκεται στο GitHub repository.