Ενσωματωμένα συστήματα πραγματικού χρόνου

Αναφορά – Εργασία 1

- Όνομα: Στασινός Αλκιβιάδης
- ΑΕΜ: 9214
- Link: https://github.com/astasinos/University-Projects/blob/master/Real-Time%20Embedded%20Systems/First%20Project/prod-cons.c

Όπως ζητείται και στην εκφώνηση ο κώδικας ο οποίος μας δόθηκε διαμορφώθηκε κατάλληλα ώστε η ***FIFO*** ουρά να δέχεται ***workFunc structs***, στα οποία προστέθηκε και ένα *timestamp* με τη μορφή ενός ***timeval struct*** όπου καταγράφεται η χρονική στιγμή κατά την οποία το *task* προστίθεται στην ουρά μέσα στην ***queueAdd()***. Οι συναρτήσεις που υλοποιήθηκαν είναι απλές (τυπώνουν ένα μήνυμα ή το αποτέλεσμα μιας μαθηματικής πράξης). Η εκτέλεση των συναρτήσεων που περιλαμβάνονται στα ***workFunc structs*** της ουράς λαμβάνει χώρα **μετά** το **unlock** του **mutex** σε κάθε **consumer** με στόχο την παράλληλη εκτέλεση τους.

Η καλή λειτουργία του κώδικα μπορεί να διαπιστωθεί παρατηρώντας και την τιμή της μεταβλητής count, η οποία αυξάνει και προσδιορίζει το **id** του αντικειμένου που <<καταναλώθηκε>> εκείνη τη στιγμή, και θα πρέπει προφανώς στο τέλος να φτάνει την τιμή **producer_count * LOOP**. Για λόγους απλότητας η εκάστοτε τιμή της μεταβλητής αυτής χρησιμοποιείται αθροιζόμενη με το ***threaded*** (modulo 5000)και σαν όρισμα για τις συναρτήσεις των tasks. Ο λόγος που δε χρησιμοποιήθηκε απλά το ***threaded*** είναι πως με έναν producer το όρισμα θα κατέληγε συνέχεια στην τιμή **1.** (Οι producers δημιουργούνται πρώτοι).

Για τον υπολογισμό της χρονικής διαφοράς χρησιμοποιήθηκε η συνάρτηση ***gettimeofday()*** και ως χρονική στιγμή τέλους θεωρείται η στιγμή που ο consumer λαμβάνει το task ενώ αρχής η στιγμή αμέσως μετά την εκτέλεση της **queueAdd()**. Αφαιρόντας **κατάλληλα** τις δύο τιμές και αποθηκεύοντας το αποτέλεσμα έχουμε τον χρόνο που παρήλθε σε microseconds. Για τη συλλογή των στατιστικών δεδομένων ακολουθήθηκε η εξής διαδικασία:

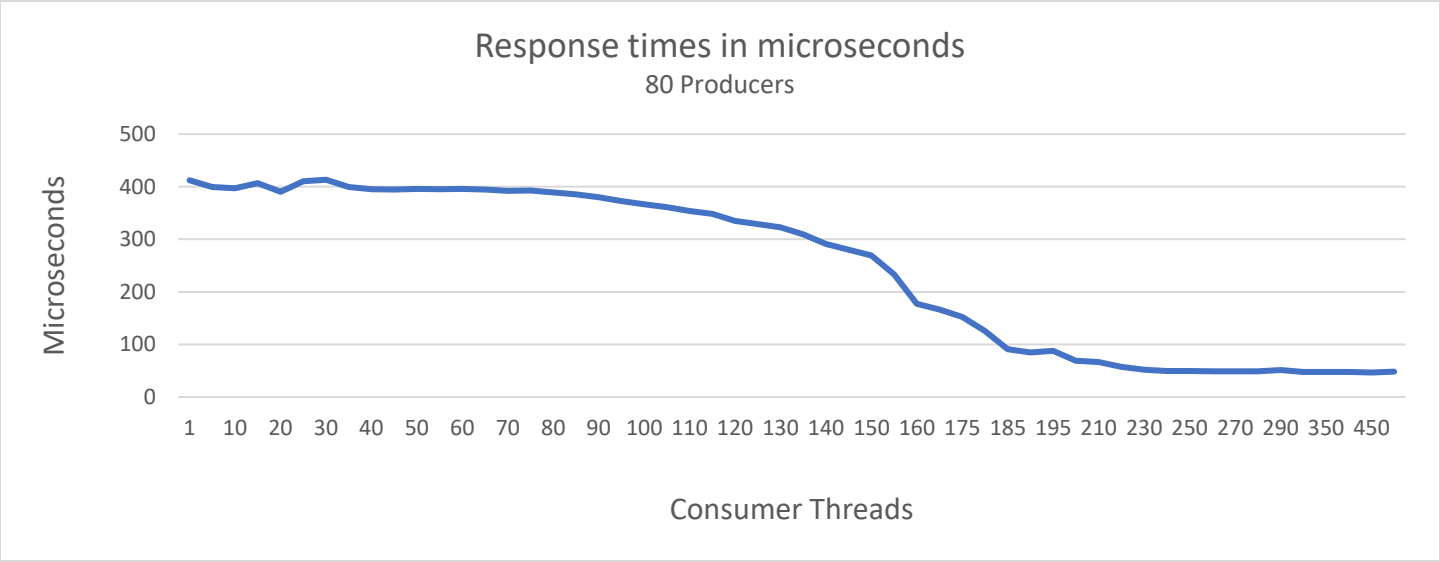
- Ο αριθμός των **producer threads** ορίστηκε αρχικά σε **80** και ύστερα σε **1.**
- Ο αριθμός **LOOP** τέθηκε **20000** με σκοπό την προσομοίωση συνθηκών υψηλού φόρτου. Με τον αριθμό αυτό διαπιστώθηκε ότι οι μετρήσεις μας συγκλίνουν ικανοποιητικά.
- Προστέθηκε προσωρινά για την απόκτηση των μετρήσεων λειτουργία καταγραφής σε αρχείο του μέσου χρόνου που υπολογίζεται σε κάθε εκτέλεση από το πρόγραμμα. (Δεν υπάρχει στην τελική έκδοση του κώδικα).
- Ύστερα προς διευκόλυνση της συλλογής των δεδομένων δημιουργήθηκε ένα **python script** το οποίο αναλαμβάνει να εκτελέσει το πρόγραμμα για αρκετούς διαφορετικούς αριθμούς **consumer threads**. Για την απόκτηση ποιοτικότερων μετρήσεων το **script** εκτελεί **δύο** φορές το πρόγραμμα για κάθε αριθμό **consumer threads**, υπολογίζει τον μέσο όρο των δύο εκτελέσεων και καταγράφει την τελική μέτρηση. Υπάρχει στο repository της εργασίας.
- Δημιουργήθηκε το διάγραμμα των χρόνων μέσω ενός **Excel workbook** που υπάρχει στο repository.

Ο αριθμός των producer και των consumer δίνεται ως όρισμα κατά την εκτέλεση, για παράδειγμα **./prod-cons 80 200** (80 producer threads και 200 consumer).

Επιπλέον σημειώνεται πως το **QUEUE SIZE** παρέμεινε **10**. Στην υλοποίηση μας η πρόσβαση στην ουρά γίνεται από ένα **κοινό mutex** και καθόλη τη διάρκεια της εκτέλεσης συμβαίνει ένας **συνεχής διαγωνισμός** ανάμεσα σε **producer** και **consumers** για το mutex αυτό και τη δυνατότητα προσθήκης ή αφαίρεσης κάποιου item από την ουρά. Όταν η ουρά γεμίζει και οι producers λαμβάνουν το σήμα **wait** τίθενται ουσιαστικά σε sleep mode και δε

συμμετέχουν στο **mutex race,** αφήνοντας εξολοκλήρου την ουρά στους διαθέσιμους **consumers.** Η υλοποίηση του μεγέθους της ουράς εξαρτάται από τις απαιτήσεις της εκάστοτε εφαρμογής. Ιδανικά θα θέλαμε ο φόρτος του συστήματος να είναι τέτοιος ώστε η ουρά να μην φτάνει **ποτέ στα όρια** της και να υπάρχει ομαλή ροή εισόδου και εξόδου των tasks, ώστε ούτε οι **producers** ούτε οι **consumers** να βρίσκονται σε αναμονή. Δηλαδή να υπάρχει ισορροπία μεταξύ του ρυθμού παραγωγής και κατανάλωσης των αντικειμένων της ουράς. Στην δική μας υλοποίηση ο ρυθμός με τον οποίο γίνεται η <<κατανάλωση>> των **tasks** είναι αρκετά μεγάλος και δε διαφέρει πολύ από τον ρυθμό με τον οποίο αυτά προστίθενται στην ουρά. Σε αυτό συνεισφέρει και η απλότητα των υλοποιημένων συναρτήσεων που δεν κρατούν τους consumers απασχολημένους για σημαντικό χρονικό διάστημα. Επομένως για τη συγκεκριμένη υλοποίηση και δεδομένων των δυνατοτήτων του μηχανήματος στο οποίο έγιναν οι μετρήσεις οι **10** θέσεις της ουράς μας είναι ικανοποιητικά αρκετές.

Οι μετρήσεις έγιναν σε σύστημα με διπύρηνο επεξεργαστή **Intel Core i7 6500U** και φαίνονται παρακάτω αρχικά για 80 consumers.



Όπως φαίνεται ο χρόνος παραλαβής ενός item από τους consumers είναι σχετικά σταθερός αρχικά, και η πτώση ξεκινά περίπου στα 85 consumer threads. Η αρχική σταθερότητα οφείλεται στο ότι ο αριθμός των consumers είναι μικρότερος από αυτόν των producers, με αποτέλεσμα να προστίθενται 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 **260** 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 **10**.