Ενσωματωμένα Συστήματα Πραγματικού Χρόνου

Εργασία 1 - Αναφορά

Ονοματεπώνυμο	ΑΕΜ	E-mail
Κοσέογλου Σωκράτης	**8837**	**sokrkose@ece.auth.gr**

- **Github:** https://github.com/Sokrkose/University-Assignments/tree/main/Real-Time%20Embedded%20Systems/1%CE%B7%20%CE%95%CF%81%CE%B3%CE%B1%CF%83%CE%B9%CE%B1

Στην συγκεκριμένη εργασία ζητήθηκε να διαμορφώσουμε τον κώδικα *prod-cons.c* (τον οποίο μπορείτε να βρείτε στο παραπάνω link) έτσι ώστε να μελετήσουμε την συμπεριφορά του κατά την μεταβολή του αριθμού των threads τόσο του **producer** όσο και του **consumer**, και τέλος να βρούμε τον **βέλτιστο** αριθμό **consumer threads.** Αρχικά, διαμορφώθηκε ο κώδικας έτσι ώστε η κυκλική ουρά **FIFO** να αποθηκεύει στοιχεία τύπου μιας δομής (**workFunction**) και όχι στοιχεία τύπου **integer**. Η δομή τύπου **workFunction** έχει μια τοπική συνάρτηση η οποία καλείτε κάθε φορά που εκτελείτε κάποιο **consumer thread** και εκτυπώνει στην κονσόλα το **ID** του **producer thread** που έβαλε στην ουρά το συγκεκριμένο στοιχείο, το οποίο μόλις «καταναλώθηκε».

Για την μέτρηση της χρονικής διαφοράς μεταξύ της στιγμής που ένας **producer** τοποθέτησε ένα στοιχείο στην ουρά μέχρι την στιγμή όπου κάποιος **consumer** «κατανάλωσε» αυτό το στοιχείο έγινε με την συνάρτηση ***gettimeofday().*** Πιο συγκεκριμένα, η αρχική μέτρηση έγινε ακριβώς μετά την συνάρτηση ***queueAdd()*** και η δεύτερη μέτρηση έγινε αμέσως μετά την εκτέλεση της συνάρτησης ***queueDel().*** Στην συνέχεια, αφαιρώντας τις δύο αυτές μετρήσεις βρήκαμε τον χρόνο αναμονή, ενός συγκεκριμένου στοιχείου της ουράς. Για να συμβεί όμως αυτό έπρεπε να δημιουργηθούν τρείς διαφορετικοί πίνακες με μέγεθος ίσο με αυτό της **FIFO** ουράς, οι οποίοι πίνακες κρατούσαν τις παραπάνω μετρήσεις για κάθε στοιχείο της ουράς.

Ακόμη, ο αριθμός **LOOP** ορίστηκε **50000** έτσι ώστε να προσομοιώνει υψηλό φόρτο εργασίας και οι μετρήσεις να μπορούν να συγκλίνουν σε κάποια τιμή. Επίσης, η συνάρτηση του **consumer thread** δεν έχει συνθήκη τερματισμού καθώς είναι μια ***while(1)*** loop. Συνεπώς, χρησιμοποιήθηκαν δύο ακόμα μετρήσεις χρόνου που χρονομετρούν την διάρκεια που τρέχει ο κώδικας, έτσι ώστε όταν περνάει το χρονικό όριο που τίθενται από τον χρήστη να βγαίνει από την ***while(1)*** μέσω της ***exit()***. Για την καταγραφή του τελικού μέσου όρου των χρόνων αναμονής χρησιμοποιήθηκε η συνάρτηση ***fprintf()*** η οποία αποθήκευε την τελική τιμή **average** σε ένα **.txt** αρχείο. Τέλος, το μέγεθος της ουράς **FIFO** παρέμεινε στις **10** θέσεις. Γενικότερα, το μέγεθος της ουράς καθορίζεται από τον φόρτο εργασίας της εκάστοτε εφαρμογής. Ιδανικά, θα θέλαμε η ουρά να μην γεμίζει ποτέ, αλλά να υπάρχει ισορροπία μεταξύ **producers** και **consumers**.

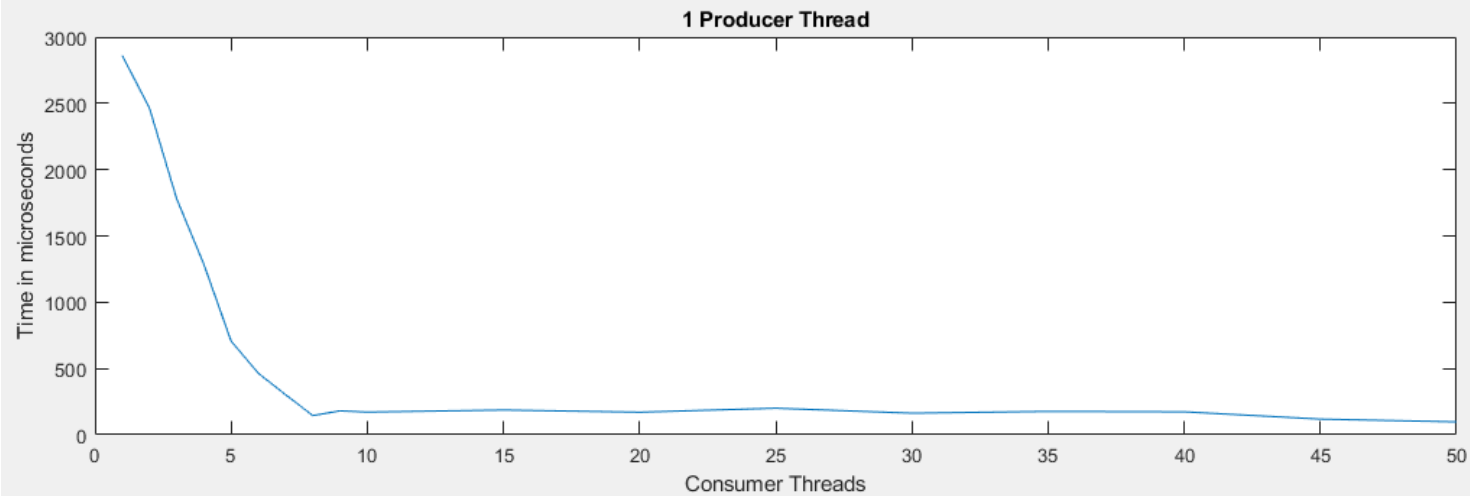
Για την συλλογή των στατιστικών δεδομένων ακολουθήθηκε η εξής διαδικασία:

- Αρχικά, ο αριθμός των **producer threads** ορίστηκε 1 και στην συνέχεια 50, αλλάζοντας όμως συνεχώς των αριθμό των consumer threads, έτσι ώστε να βρούμε τον **βέλτιστο** αριθμό.
- Ο τελικός κώδικας είναι ο ***prod-cons v5.0.c*** (μπορείτε να τον βρείτε στο παραπάνω link)και εκτελείτε με την ακολουθία των εντολών (**gcc 'prod-cons v5.0.c' -o 'prod-cons v5.0' -pthread**) και στην συνέχεια (**./'prod-cons v5.0.c' <PRODUCER THREADS NUMBER> <CONSUMER THREADS NUMBER>**). Δηλαδή εάν θέλουμε να εκτελέσουμε το πρόγραμμα με 1 producer thread και 5 consumer threads θα γράφαμε την παρακάτω εντολή (**./'prod-cons v5.0.c' 1 5**).
- Για την αυτοματοποίηση της διαδικασίας καταγραφής των στατιστικών στοιχείων δημιουργήθηκε ένα **bash script** (*miningScript.txt*) το οποίο εμπεριέχει την ακολουθία εντολών οι οποίες εκτελέστηκαν στο τερματικό.
- Ο κώδικας εκτελέστηκε **5 φορές** για κάθε συνδυασμό producers-consumers και στην συνέχεια υπολογίστηκε ο μέσος όρος από αυτές τις 5 μετρήσεις, έτσι ώστε να αποφευχθεί η εσφαλμένη τελική μέτρηση, κάτι το οποίο

μπορεί να συμβεί σε περίπτωση που το σύστημα μας είναι υπερφορτωμένο την δεδομένη στιγμή που τρέχει το πρόγραμμα.

- Στην συνέχεια, υλοποιήθηκε ένα **MATLAB script** (*scriptRTOS.m*) το οποίο παίρνει τις μετρήσεις που αποθηκεύτηκαν στο *output_file.txt* που έχει ως έξοδο το πρόγραμμα μας και κάνει **plot** τα στατιστικά δεδομένα, ώστε να μπορέσουμε να βγάλουμε τα κατάλληλα πορίσματα.

Οι παρακάτω μετρήσεις έγιναν σε επεξεργαστή **Intel Core i3-4005U @ 1.7GHz (dual core – four logical)** μέσω **Virtual Machine** και συγκεκριμένα το **VMware Workstation 15**. Λόγω τόσο του VM όσο και της χαμηλής επεξεργαστική ισχύς, οι μετρήσεις είναι σχετικά υψηλές. Παρ' όλα αυτά τόσο οι καμπύλες του χρόνου όσο και οι διακυμάνσεις των μετρήσεων είναι ικανοποιητικές, έτσι ώστε να βγάλουμε συμπεράσματα σχετικά με τον βέλτιστο αριθμό consumer threads.



Στην παραπάνω καμπύλη, όπου ο αριθμός των producer threads είναι 1, βλέπουμε ότι ο **βέλτιστος** αριθμός consumer threads είναι γύρω στα **8-10** threads. Επίσης, βλέπουμε ότι ξεκινώντας από το 1 consumer thread έως τα 10 consumer threads υπάρχει αισθητή μείωση του χρόνου αναμονής και αυτό οφείλεται στο γεγονός ότι κατά τον χρονοπρογραμματισμό υπάρχουν περισσότεροι διαθέσιμοι consumers σε σχέση με τους producers, οπότε είναι πιθανό με το που προστεθεί κάποιο στοιχείο στην ουρά να «καταναλωθεί» αρκετά σύντομα. Στην παρακάτω εικόνα, όπου ο αριθμός των producer threads είναι 1, ο **βέλτιστος** αριθμός consumer threads είναι περίπου **200-230** για τον ίδιο λόγω που επισημάνθηκε παραπάνω. Παρ' όλα αυτά, μπορούμε να παρατηρήσουμε ότι μέχρι και τα 50 consumer threads ο χρόνος αναμονής είναι σχετικά μεγάλος και σταθερός κάτι που οφείλεται στο γεγονός ότι ενώ προστίθενται στοιχεία στην ουρά, δεν υπάρχουν άμεσα διαθέσιμοι consumers ώστε να «απελευθερώσουν» τα στοιχεία της ουράς.