Ενσωματωμένα συστήματα πραγματικού χρόνου
Παππάς Γιώργος Εφραίμ AEM: 9124

Απαλακτική εργασία 2020

1. Σύντομη περιγραφή κώδικα

Στα πλάισια της εργασίας καλούμαστε να υλοποιήσουμε έναν **Timer** σε C, που θα εκτελεί αυτόματα κώδικα ανά τακτά χρονικά διαστήματα. Πρακτικά λοιπόν ο **Timer** είναι ένα struct που περιέχει τις απαραίτητες μεταβλητές ώστε οριστούν οι ζητούμενες λειτουργίες του timer, όπως είναι ο ορισμός της περιόδου, ο ορισμός των **tasks** που θα εκτελεστούν κτλ (οι λειτουργίες του αναφέρονται στην εκφώνηση).

Ουσιαστικά, ο τρόπος που λειτουργεί το πρόγραμμα είναι ο εξής:

- Φτιάχνεται ένα **Timer** object και του δίνονται οι προδιαγραφές υπό τις οποίες θα λειτουργεί (δηλαδή περίοδος, συνάρτηση που θα πρέπει να εκτελείται μια φορά ανά την περίοδο κτλ)
- Ανοίγει 1 **producer thread** (για τον κάθε Timer object) (το κάθε timer object ανοίγει ένα producer thread) που πρακτικά βλέπει τις προδιαγραφές του Timer object στο οποίο «ανήκει»
- Στο producer thread εκτελείται η συνάρτηση producer η οποία περιλαμβάνει μια λούπα που τρέχει **TasksToExecute** φορές, με ρυθμό που του δίνει η περίοδος του αντίστοιχου Timer, βάζοντας ένα task ανά επανάληψη του loop (1 task ανα χρόνο περιόδου) στην ουρά (που περιλαμβάνει τα προς εκτέλεση tasks)
- Οι **consumers** δουλεύουν ακριβώς όπως στην πρώτη εργασία.

2. Περιγραφή αντιμετώπισης της χρονικής μετατόπισης (drifting)

Επειδή ο βασικός σκοπός του **Timer** είναι να εκτελεί κώδικα μία φορά ανά χρόνο περιόδου, σε κάθε iteration του loop του **producer**, στο τέλος του, γίνεται χρήση της **usleep(Period),** ώστε στην ιδανική περίπτωση (που δεν υπάρχουν καθυστερήσεις), να ξεκινάει το iteration, να κάνει τη δουλειά που πρέπει να κάνει (να τοποθετεί το task στην ουρά) και μετά να «κοιμάται» για χρόνο ίσο με την περίοδο, ώστε τα διαστήματα μεταξύ δύο διαδοχικών add tasks στην ουρά να απέχουν χρόνο ίσο με αυτόν της περιόδου. Ωστόσο επειδή, **πρώτον** εκτελούνται βοηθητικοί υπολογισμοί ενδιάμεσα στη συνάρτηση producer (πχ αποθήκευση της χρονικής στιγμής που μπήκε ένα task στην ουρά, και άλλοι που έχουν να κάνουν με τα στατιστικά λειτουργίας του προγράμματος), **δεύτερον** για να μπει κάτι στην ουρά ο producer πρέπει να λάβει το **mutex**, το οποίο μπορεί να είναι κατειλημμένο από άλλον producer (από άλλον timer) ή από κάποιον consumer, κάτι που προσθέτει καθυστερήσεις μη προβλέψιμες, **τρίτον** η συνάρτηση **usleep()** δεν είναι «τέλεια» και προσθέτει απο μόνη της μια κάποια καθυστέρηση, προκύπτουν χρονικές μετατοπίσεις που τελικά κάνουν τα διαστήματα μεταξύ των διαδοχικών add tasks να μην απέχουν *χρόνο = timerPeriod,* αλλά κάτι παραπάνω από αυτό.

Για τη διόρθωση αυτής της μετατόπισης λοιπόν έγιναν τα εξής:

1. Στην αρχή του εκάστοτε iteration, το producer thread προσπαθεί να πάρει το mutex, αφού το πάρει τοποθετεί το task στην ουρά, και επιστρέφει το mutex. Διατηρούμε λοιπόν σε πρώτη φάση τη χρονική στιγμή που μπήκε το task στην ουρά. (Το βασικό ενδιαφέρον λοιπόν είναι αυτή η χρονική στιγμή να απέχει από την επόμενη αντίστοιχη χρονική στιγμή χρόνο ίσο με το period του timer).
2. Υπολογίζουμε λοιπόν, αφού ξέρουμε τι ώρα μπήκε το task στην ουρά στο προηγούμενο iteration και τι ώρα μπήκε σε αυτό το iteration, το χρονικό διάστημα που πέρασε (**addToaddTime**), και βλέπουμε πόσο απέχει αυτό το διάστημα από το επιθυμητό (επιθυμητό διάστημα = timerPeriod). (Ονομάζουμε τη διαφορά τους, **offset=addToaddTime-Period**). Άρα πχ αν η περίοδος είναι 100ms, και μετρήσουμε ότι από **add** σε **add** πέρασε χρόνος =110ms, offset=10ms. Αυτός ο χρόνος περιλαμβάνει την «ατέλεια» της usleep() και τον μη προβλέψιμο χρόνο που πέρασε στο προηγούμενο iteration, καθώς ο producer προσπαθούσε να πάρει το mutex. (Να σημειωθεί εδώ ότι αυτη τη μεταβλητη addToaddTime στο εκάστοτε iteration, τη διατηρώ σε ένα αρχείο για να υπολογιστούν αργότερα τα στατιστικά του Drift)

3. Στο τέλος του εκάστοτε iteration, (μετά ακριβώς τη usleep()), κρατάμε μια μεταβλητή **sleptFor**=sleptFor-offset. Στην περίπτωση που βρισκόμαστε στο πρώτο iteration, η sleptFor έχει αρχικοποιηθεί με την τιμή sleptFor=Period.
   Άρα πχ αν Period=1000ms, **1ο iteration** έχουμε usleep(Period), sleptFor=Period.
   2ο iteration υπολογίζουμε την addToaddTime=1100, άρα offset=100, usleep(**sleptFor-offset-extraOffset**)=>usleep(1000-100-extraOffset) (το extraOffset θα εξηγηθεί αμέσως μετά), **sleptFor**=sleptFor-offset=> sleptFor=1000-100=900.
   3ο iteration, υπολογίζουμε addToaddTime=1010ms, άρα offset=10, άρα usleep(900-10-extraOffset) και sleptFor=900-10=890
   4ο iteration, υπολογιζουμε addToaddTime=999, άρα offset=-1, άρα usleep(890-(-1)-extraOffset) και sleptFor=890-(-1)=891 κτλ
4. Ακριβώς πριν εκτελεστεί η **usleep(),** διατηρούμε τη χρονική στιγμή που βρισκόμαστε, και σε μια μεταβλητή που θα την πούμε εδώ **extraOffset,** κρατάμε τον χρόνο που πέρασε απο τη στιγμή που μπήκε το task στην ουρά, μέχρι να γίνουν οι διάφοροι υπολογισμοι που αναφέρθηκαν προηγουμένως και να φτάσουμε στην usleep(). Για παράδειγμα, αν το task μπήκε στην ουρά τη χρονική στιγμή **t1=25ms** και φτάσουμε στη **usleep()** τη χρονική στιγμή **t2=60ms** (επειδή απο t1 έως t2 εκτελούνται βοηθητικοί υπολογισμοί όπως η αποθήκευση των διάφορων χρονικών στιγμών, διάφορες if, και γενικά κωδικάς που προσθέτει μη επιθυμητές καθυστερήσεις) τότε **extraOffset=60-25=35ms**.
5. Άρα καταλήγω να έχω σαν όρισμα στη usleep(), το **(sleptFor-offset-extraOffset)** που έδωσε τελικά αρκετά καλά αποτελέσματα όπως θα φανεί παρακάτω.

3. Μετρήσεις και αποτελέσματα της λειτουργίας του προγράμματος

Εκτελέστηκαν συνολικά 4 διαφορετικά πειράματα. Στο **πρώτο** πείραμα ενεργός είναι ο Timer με περίοδο **1sec**, στο **δεύτερο** μόνο αυτός με περιοδο=**0,1sec**,στο **τρίτο** αυτός με περίοδο=**0,01sec** και στο **τέταρτο** ήταν ενεργοί και οι 3 αυτοί timers. Το κάθε πείραμα είχε διάρκεια **μια ώρα.**

Σε κάθε πείραμα κρατήθηκαν τα εξής στατιστικά λειτουργίας:

Α. Το **drifting** (η χρονική ολίσθηση απο την περίοδο του timer)

Β. Ο **χρόνος** που χρειάζεται ο εκάστοτε Timer για να βάλει το κάθε task στην ουρά (ο χρόνος δηλαδή απο τη στιγμή που ξεκινάει να «προσπαθεί» να πάρει το **mutex** ο timer, μέχρι να το πάρει και να εκτελεστεί η συνάρτηση **queueAdd**) (Θα αναφερθεί ως Time to queue -> **TTQ**)

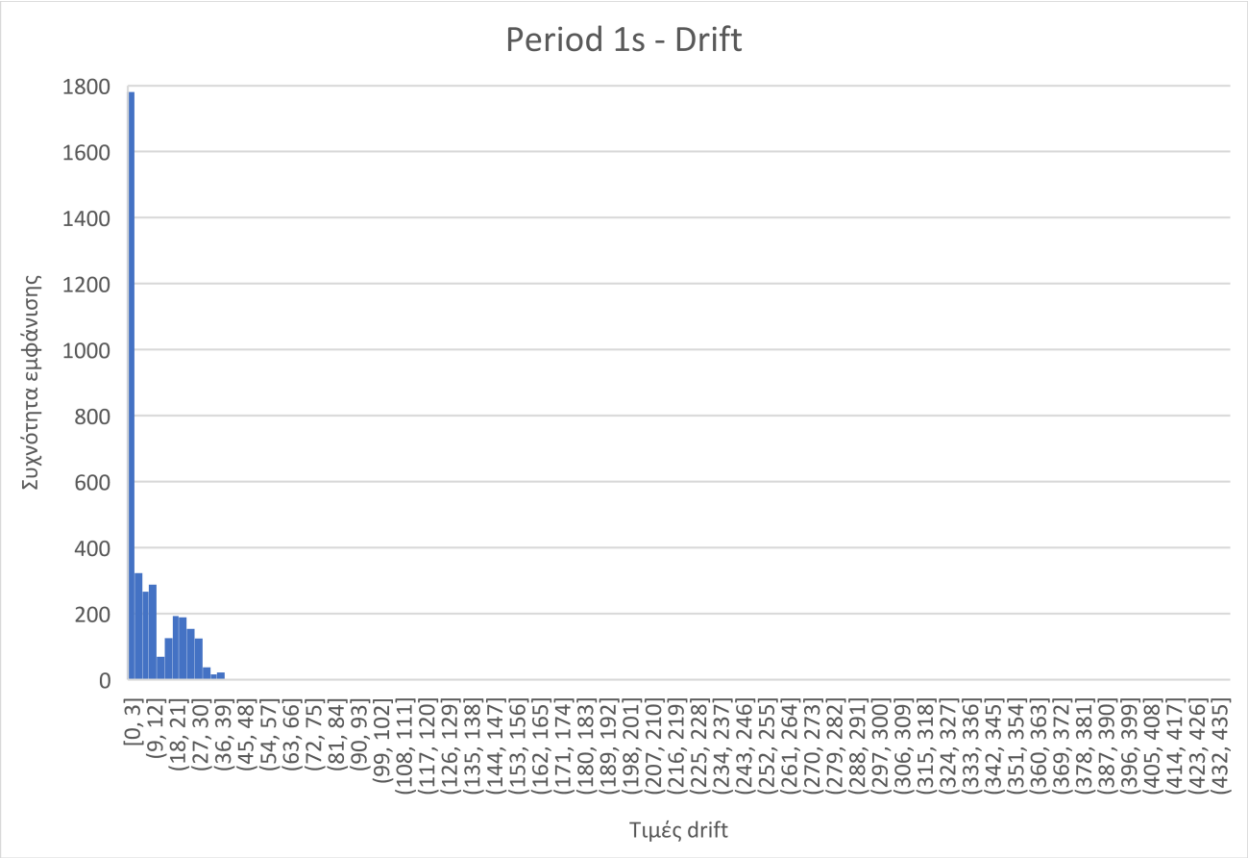
Γ. Ο χρόνος που ξοδεύεται κάθε φορά απο τον **consumer** για να βγάλει μια κλήση από την ουρά. (Ο χρόνος δηλαδή απο τη στιγμή που μπήκε το task στην ουρά μέχρι ένας consumer να το αφαιρέσει με τη χρήση της συνάρτησης **queueDel**) (Θα αναφερθεί ως **consumer time**)

Ακολουθεί παρουσίαση αυτών των αποτελεσμάτων, σύγκριση της λειτουργίας του εκάστοτε timer οταν είναι μόνος του με την περίπτωση που τρέχει ταυτόχρονα με τους άλλους δύο και ανάλυση της εκάστοτε συμπεριφοράς:

Α. Πρώτα θα παρουσιαστούν τα αποτελέσματα για τις αυτόνομες εκτελέσεις του εκάστοτε Timer

Timer με period=1s:

Ενσωματωμένα συστήματα πραγματικού χρόνου
Παππάς Γιώργος Εφραίμ ΑΕΜ: 9124



Drift

Min drift: 0us – **Max drift**: 436us – **Average drift:** 8.772992498us – **Median drift:** 4us – **Τυπική απόκλιση:** 13.13701425us

Η περίοδος ειναι 1000000us, και η μέση απόκλιση απο αυτήν είναι 8.77us άρα μπορούμε με ασφάλεια να πούμε ότι ο Timer λειτουργεί με αρκετή ακρίβεια ως προς την περίοδό του. Το max drift θα έλεγε κανείς ότι ειναι σχετικά υψηλό (σε σύγκριση με το average drift) αλλά η χαμηλή τιμή του average drift (8.77us) μας δείχνει ότι μια τέτοια «μεγάλη» απόκλιση απο την περίοδο συμβαίνει πολύ σπάνια.

Time to queue

Min TTQ (time to queue): 1us – **Max TTQ:** 4us – **Median TTQ:** 4us – **Mean TTQ:** 2.54us **Τυπική απόκλιση TTQ:** 0.5us

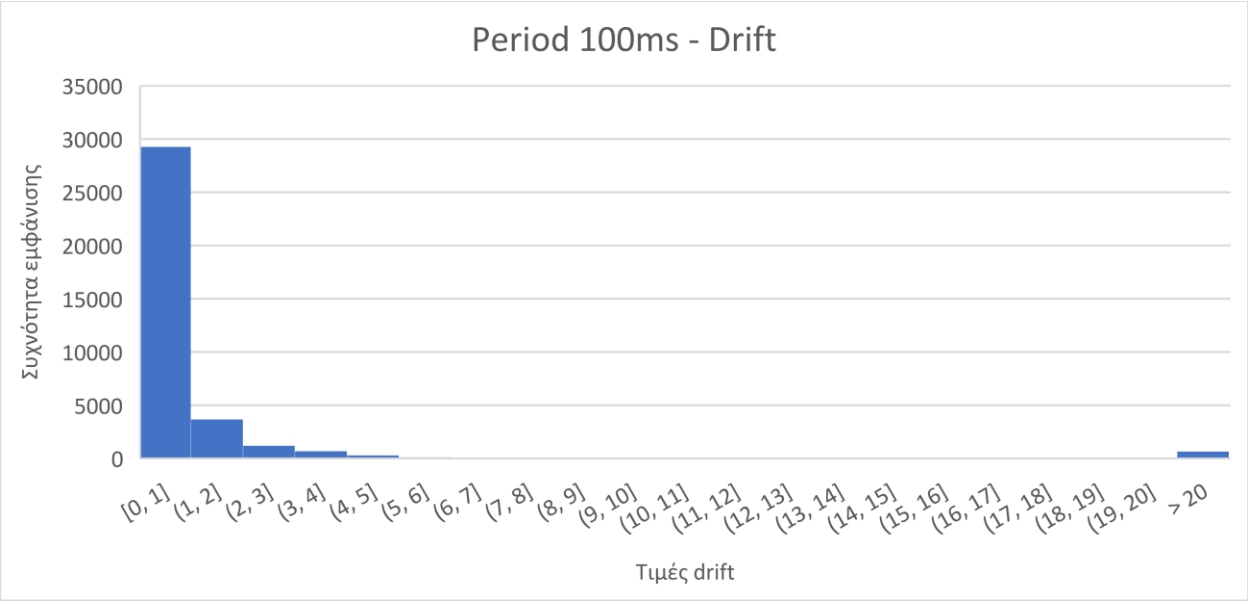
Όπως αναφέρθηκε, ο time to queue είναι ο χρόνος που «παλεύει» ο producer να πάρει το **mutex** και να προσθέσει το εκάστοτε task στην ουρά. Η περίοδος είναι αρκετά υψηλή σε σχέση με τα tasks που εκτελούνται (με τον χρόνο εκτέλεσης των tasks δηλαδή), επομένως όταν ο producer «θελήσει» να βάλει κάτι στην ουρά (το κάνει κάθε 1sec = period) η ουρά δεν περιέχει μέσα tasks, άρα δεν προσπαθεί κανένας consumer να πάρει το mutex ώστε να αφαιρέσει task από την ουρά, αφού ξεμπερδεύει με αυτό αρκετά γρήγορα και ταυτόχρονα δεν υπάρχουν άλλοι producers (timers) ενεργοί ώστε να προσπαθούν οι ίδιοι να «κλεψουν» το mutex. Γι αυτό λοιπόν ο χρόνος που κάνει ο producer να βάλει κάτι στην ουρά είναι σταθερός και μικρός. (Πρακτικά όποτε ζητήσει το mutex, αυτό είναι διαθέσιμο, και το παίρνει.)

Consumer Time

Max Consumer Time: 98us – **Min:** 21us – **Mean:** 56.97us – **Median:** 60us – **Τυπική απόκλιση:** 11,35us

Timer με period=100ms

Ενσωματωμένα συστήματα πραγματικού χρόνου
Παππάς Γιώργος Εφραίμ ΑΕΜ: 9124



Drift

Min drift: 0us – **Max drift**: 876us – **Average drift:** 1.40065us – **Median drift:** 1us – **Τυπική απόκλιση:** 7.1029us

Αντίστοιχα εδώ έχουμε επίσης σταθερά μικρό χρόνο απόκλισης απο την περίοδο. Πρακτικά επειδή ο κάθε timer σε αυτή τη φάση δουλεύει μόνος του, ο φόρτος στο σύστημα είναι χαμηλός και έτσι έχει μεγαλύτερη συνέπεια ως προς τους επιθυμητους χρόνους. (Θα δούμε αργότερα ότι όταν μπουν και οι άλλοι timers σε λειτουργία, οι διορθώσεις του drift, δεν ειναι εξ ίσου αποτελεσματικές γιατί αυτές γενικά λειτουργούν προσπαθώντας να προβλέψουμε την απόκλιση που πρόκειται να έρθει με βάση την προηγούμενη συμπεριφορα. Με περισσότερους timers σε λειτουργία, θα είναι πιο απρόβλεπτη η συμπεριφορά κάτι που θα αποτυπωθεί και ως υψηλότερος μέσος χρόνος drift.)

Time to queue

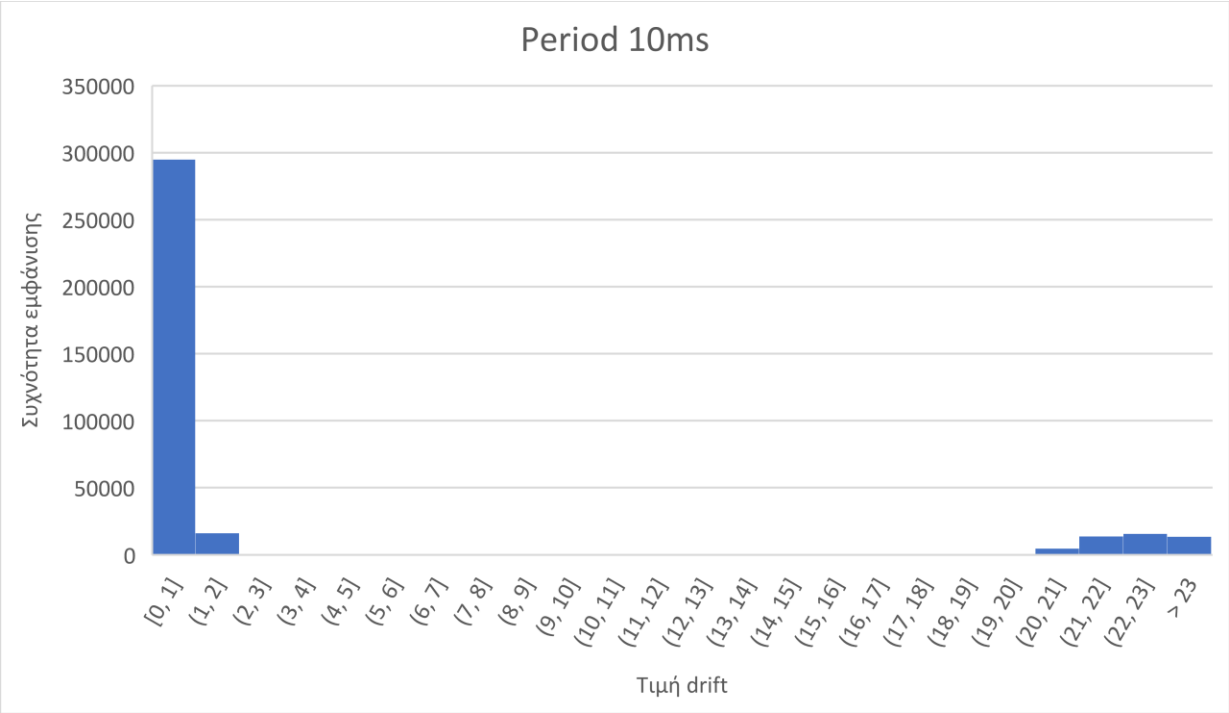
Min TTQ (time to queue): 0us – **Max TTQ:** 3us – **Median TTQ:** 2us – **Mean TTQ:** 2.39us **Τυπική απόκλιση TTQ:** 0.48us

Consumer Time

Max Consumer Time: 127us – **Min:** 16us – **Mean:** 37.39us – **Median:** 37us – **Τυπική απόκλιση:** 15,35us

Timer με period=10ms

Ενσωματωμένα συστήματα πραγματικού χρόνου
Παππάς Γιώργος Εφραίμ ΑΕΜ: 9124



Drift

Min drift: 0us – **Max drift**: 31261us – **Average drift:** 4.87us – **Median drift:** 1us – **Τυπική απόκλιση:** 116.1029us

(Εδώ βλέπουμε ένα max drift αρκετά μεγάλο κάτι που αποδίδεται κυρίως στα πάρα πολλά δείγματα που πάρθηκαν κατα τη μία ώρα λειτουργίας του γρήγορου αυτού timer. Φαίνεται εξάλλου από το average drift και απο το διάγραμμα ότι τόσο μεγάλες αποκλίσεις είναι υπερβολικά ελάχιστες. Οπότε ουσιαστικά αυτή η τεράστια καθυστέρηση είναι μεμονομένο γεγονός.)

Time to queue

Min TTQ (time to queue): 1us – **Max TTQ:** 31287us – **Median TTQ:** 2us – **Mean TTQ:** 3.05us **Τυπική απόκλιση TTQ:** 3.05us

Κρίνοντας απο το max ttq, το μεγάλο max drift που σχολιάστηκε προηγουμένως οφείλεται λογικά σε κάποια μεμονομένη καθυστέρηση ως προς τη διαθεσιμότητα του mutex.

Consumer Time

Max Consumer Time: 125us – **Min:** 17us – **Mean:** 21.39us – **Median:** 21us – **Τυπική απόκλιση:** 1.59us

Β. Ακολουθούν τα αποτελέσματα για τον κάθε Timer ενώ είναι και οι 3 ενεργοί ταυτόχρονα

Timer με period=1s:

Drift

Min drift: 0us – **Max drift**: 46702us – **Average drift:** 878.20us – **Median drift:** 62us – **Τυπική απόκλιση:** 3656.29us

Time to queue

Min TTQ (time to queue): 2us – **Max TTQ:** 46731us – **Median TTQ:** 113us – **Mean TTQ:** 510.08us **Τυπική απόκλιση TTQ:** 2624.836us

Ενσωματωμένα συστήματα πραγματικού χρόνου
Παππάς Γιώργος Εφραίμ AEM: 9124

Timer με period=0.1s:

Drift

Min drift: 0us – **Max drift**: 46336us – **Average drift:** 35.48us – **Median drift:** 1us – **Τυπική απόκλιση:** 751.21us

Time to queue

Min TTQ (time to queue): 2us – **Max TTQ:** 46343us – **Median TTQ:** 2us – **Mean TTQ:** 19.31us **Τυπική απόκλιση TTQ:** 531.49us

Timer με period=0.01s:

Drift

Min drift: 0us – **Max drift**: 37738us – **Average drift:** 6.99us – **Median drift:** 1us – **Τυπική απόκλιση:** 226.49us

Time to queue

Min TTQ (time to queue): 1us – **Max TTQ:** 37738us – **Median TTQ:** 2us – **Mean TTQ:** 3.73us **Τυπική απόκλιση TTQ:** 198.6168us

Consumer Time (Για την περίπτωση που δουλεύουν ταυτόχρονα όλοι οι timers)

Max Consumer Time: 968us – **Min:** 4us – **Mean:** 29us – **Median:** 23us – **Τυπική απόκλιση:** 14.59us

4. Σύγκριση των αποτελεσμάτων αυτών με τις αντίστοιχες τιμές των πειραμάτων που δούλευε μόνο ένας timer κάθε φορά

Παρατηρούμε αρχικά ότι τα **drifts** έχουν επηρεαστεί αρκετά τώρα που είναι ενεργοί και οι 3 timers. Βλέπουμε πως στην περίπτωση του timer με το **1sec περίοδο**, **αυξήθηκε** κατά πολύ η χρονική απόκλιση από την περίοδο. Στην περίπτωση του timer με το **0.1sec περίοδο** έχουμε μικρότερο «πρόβλημα» ενώ ο timer με το **0.01sec** περίοδο έμεινε **σχεδόν ανεπηρέαστος**.

Ο timer με περίοδο **0.01sec** ουσιαστικά βλέπει τους άλλους 2 ανενεργούς κατά βάση. Ανά 100 “ticks” του timer με **period 0.01sec** έχουμε ένα “tick” αυτού με **period=1sec** και 10 “ticks” αυτου με **period 0.1sec.** Πρακτικά ο πιο αργός timer (1sec period) έχει να ανταγωνιστεί συνεχώς τους άλλους 2 ως προς το **mutex,** ο μεσαίος (0.1sec period) «ανταγωνίζεται» τον 1sec μία φορά στα 10ticks του ενώ τον 0.01sec period συνεχώς (άρα έχει μικρότερο ανταγωνισμό σε σχέση με τον πιο αργό timer κάτι που φαίνεται και στον βαθμό που επηρεάστηκε σε σχέση με τον πιο αργό timer συγκρίνοντας με τα πειράματα που τρέχει κάθε φορά ένας timer), ενώ ο πιο γρήγορος timer (0.01sec period) έμεινε σχεδόν ανεπηρέαστος αφού ανταγωνίζεται (πρακτικα) 1 στις 10 τον μεσαίο timer και 1 στις 100 τον αργό timer.

Αντίστοιχα, ο έξτρα φόρτος στην ουρά (αφού πλέον τοποθετούνται περισσότερα tasks στην ουρά από ότι στις απομονωμένες εκτελέσεις), κρατάει περισσότερη ώρα ενεργούς τους **consumers** που έχουν πλέον να αφαιρέσουν περισσότερα tasks απο την ουρά, άρα έχουμε μεγαλύτερο ανταγωνισμό για το **mutex.**

Γενικά κατά τη διόρθωση του drift, η πιο απρόβλεπτη καθυστέρηση έχει να κάνει με το να λάβει ο producer το mutex. Στην περίπτωση που τρέχουν και οι 3 timers ταυτόχρονα έχουμε μια αρκετά απρόβλεπτη συμπεριφορά για την περίπτωση του timer με 1sec period, λιγότερο, αλλα παλι αρκετά απροβλεπτη για την περίπτωση του timer με 0.1sec, και λίγο απρόβλεπτη για την περίπτωση του πιο γρήγορου timer. Αυτό έρχεται να μας το επιβεβαιώσει και η μέτρηση **Mean Time To Queue (Mean TTQ),** που δείχνει όπως αναφέρθηκε προηγουμένως το χρόνο που χρειάζεται ο εκάστοτε producer να προσθέσει ένα task στην ουρά (κάτι που απαιτεί τη λήψη του mutex από αυτόν), που αυξήθηκε με αντίστοιχο τρόπο για τον κάθε timer (επηρεάστηκε στον ίδιο βαθμό με τα **average drifts**)

5. Μέτρηση χρήσης επεξεργαστή

Για να βρεθεί το ποσοστό χρήσης του επεξεργαστή, χρησιμοποιήθηκε στην εκτέλεση του προγράμματος η εντολή **time ./main** (main ειναι το πρόγραμμά μας). Με βάση την έξοδό της, με κατάλληλη πράξη, υπολογίστηκε ο φόρτος του επεξεργαστή δίνοντας τα παρακάτω αποτελέσματα:

Πείραμα με **Timer -> Period=1s**: **CPU** usage **0.02%**

Πείραμα με **Timer -> Period=0.1s**: **CPU** usage **0.02%**

Πείραμα με **Timer -> Period=0.01s**: **CPU** usage **1.5%**

Πείραμα με **τους τρεις timers ενεργούς**: **CPU** usage **2.7%**

Όπως αναμενόταν, ο επεξεργαστής ειδικά στις περιπτώσεις των αργών timer (1s και 0.1s period) δούλεψε πολύ κοντά στο 0%, ενώ το ποσοστό αυξήθηκε στην περίπτωση του πιο γρήγορου timer καθώς και στην περίπτωση των 3 ενεργών timer.

6. Απαιτήσεις λειτουργίας πραγματικού χρόνου (σχετικά με το μέγεθος της ουράς, το «βάρος» των tasks, το period του timer και τον αριθμό των consumers)

Λειτουργία πραγματικού χρόνου σημαίνει, όταν προκύπτει ένα task (ένα αίτημα) για το σύστημα πραγματικού χρόνου, αυτό να μπαίνει άμεσα στην ουρά και να ξεκινάει άμεσα η διεκπεραίωσή του.

Αν είχαμε μικρή **ουρά**, τα tasks ήταν χρονοβόρα και οι consumers λιγοι, αλλά προέκυπταν αιτήματα προς το σύστημα με μεγάλο ρυθμό, θα υπήρχε πρόβλημα. Άρα χρειαζόμαστε μέγεθος ουράς τέτοιο ώστε να **μη γεμίζει** πρακτικά ποτέ (ώστε να ικανοποιούμε το ρυθμό έλευσης tasks στο σύστημα) Φυσικά σε μία περίπτωση που η ουρά γεμίζει επειδή το σύστημά μας είναι αργό, δε θα βαλουμε μεγάλη ουρά, γιατί αυτό δε θα μας λύσει το πρόβλημα, απλά θα το μετατοπίσει λίγο στο χρόνο. Ιδανική λύση σε περίπτωση που γεμίζει η ουρά, είναι η αύξηση των consumers.

Ο αριθμός των **consumers** πρέπει να είναι τόσος ώστε ποτέ να μην ξεμένουμε απο consumers (κάτι που μας το καθορίζει πρώτον ο ρυθμός έλευσης tasks και δεύτερον το βάρος των tasks). Εδω βέβαια μπαίνει στο παιχνίδι και η διάρκεια εκτέλεσης της **timerFcn**, που αν είναι αρκετά μεγάλη για τις δυνατότητες του συστήματός μας, θα καταλήξουμε να έχουμε σε κάθε περίπτωση κατειλλημένους consumers συνεχώς (άρα και μεγάλη αναμονή των tasks στην ουρά). Εδώ η λύση είναι είτε να μειώσουμε το ρυθμό εισαγωγής αιτημάτων στην ουρά (μείωση περιόδου του timer) είτε με κάποιο τρόπο να μειώσουμε (με βελτιστοποιήσεις) το «βάρος» των tasks.

Η **περίοδος** του timer πρέπει να είναι τέτοια ώστε να μην γεμίζει την ουρά (αν θεωρήσουμε ότι έχουμε κάποιον συγκεκριμένο περιορισμό στην ουρά) και να είναι σίγουρα (αρκετά) μεγαλύτερη από το μέσο χρόνο εισόδου κάποιου task στην ουρά.

Προφανώς από ότι είδαμε στη δικιά μας εφαρμογή, από τις μετρήσεις που προηγήθηκαν, καλύφθηκαν όλες αυτές οι απαιτήσεις.

Γενικά λοιπόν θέλουμε να επιτύχουμε το εξής: **ρυθμός εισόδου**<=**ρυθμός εξόδου** και ουρά τέτοια ώστε να μη γεμίζει ποτέ. Το **ρυθμό εισόδου** τον καθορίζει η **περίοδος του timer** ενώ τον **ρυθμό εξόδου**, ο **αριθμός των consumers** και η **διάρκεια εκτέλεσης** της Timerfcn

Ανάλογα λοιπόν την εφαρμογή που θα έχουμε κάθε φορα, θα προσαρμόζουμε κατάλληλα την περιοδο του timer, την ουρά και τους consumers ώστε να καλύπτουμε τις απαιτήσεις που υπάρχουν.

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