Ενσωματωμένα Συστήματα Πραγματικού Χρόνου – Απαλλακτική Εργασία 2020

Υλοποίηση Τιmer σε raspberry pi – Παράλληλα νήματα producer, consumer

---

Ραφαήλ Μπουλογεώργος 9186 – Νέο πρόγραμμα σπουδών
Εξεταστική Περίοδος Σεπτεμβρίου 2020

Σύνδεσμος Github: https://github.com/rafampou/Embedded_Real_Time_Systems

Σύντομη επεξήγηση υλοποίησης

Queue

Η υλοποίηση βασίστηκε στην πρώτη εργασία όπου είχαμε νήματα producers που τοποθετούσαν σε μια ουρά queue jobs, υπό την μορφή δεικτών σε συναρτήσεις και consumers που αναλάμβαναν να εκτελέσουν αυτές τις συναρτήσεις.

Έτσι και τώρα η υλοποίηση της ουράς παραμένει η ίδια και δέχεται ως στοιχεία struck job που περιλαμβάνει έναν pointer σε συνάρτηση και έναν Pointer για τα δεδομένα. Το μέγεθος της ουράς γίνεται define στην αρχή του queue.h ενώ μέσα στο queue.c υπάρχει και η υλοποίηση για τα νήματα consumers που θα περιγραφεί παρακάτω.

Timer

Για την υλοποίηση του timer χρησιμοποίησα ένα struct που περιέχει τα εξής

- Μεταβλητές (int) Period, TaskToExecute, StartDelay
- Δείκτες για τις συναρτήσεις StartFcn, StopFcn, TimerFcn, ErrorFcn
- Δείκτη στα δεδομένα UserData
- Μια μεταβλητή στο id του timer(για στατιστικούς λόγους)
- Ένα struct timeval startTime για την ώρα έναρξης του timer

Τα παραπάνω είναι όπως ορίστηκαν στις απαιτήσεις του μαθήματος.

Αρχικοποίηση TImer

Για να αρχικοποιήσουμε έναν Timer, χρησιμοποιούμε την TimerInit που δέχεται ως ορίσματα όλα τα παραπάνω και τα αναθέτη σε ένα αντικείμενο Timer.

Πυροδότηση TImer

Όπως μας ζητήθηκε υπάρχουν 2 συναρτήσεις που ξεκινούν τον Timer, η start και η startat με τα αντίστοιχα ορίσματα. Ως πρώτο όρισμα και οι 2 συναρτήσεις παίρνουν τον δείκτη του Timer ενώ επιστρέφουν το pthread που δημιουργείτε κατά την έναρξη.

Για την πρώτη περίπτωση η συνάρτηση start εκτελεί την StartFcn εκτυπώνει timestamp και ξεκινάει ένα thread με δείκτη στην συνάρτηση waitFunction και δεδομένα τον Timer. Επιπλέον η Startat αρχικοποιεί και την μεταβλητή StartTime όσο είναι το όρισμα αφού πρώτα τα μετατρέψει σε seconds.

Νήμα Producer

Την θέση του producer έχει η συνάρτηση waitFunction που δέχεται ως όρισμα τον Timer.

Αρχικά ελέγχει εάν γίνεται αμέσως η έναρξη του timer ή έχει προγραμματιστεί μετά από κάποια sec όπως ορίστηκαν στην startat και αφού τα μετατρέψει σε milliseconds καλεί την usleep;

Στην συνέχεια στην βασική loop που εκτελείτε όσες φορές υποδεικνύει η μεταβλητή TaskToExecute εκτελούνται τα παρακάτω.

1. Λαμβάνουμε timestamp ώστε να υπολογίσουμε το drifting και προσθέτουμε μια εικονικά καθυστέρηση της τάξεως των 10usec που αντιστοιχούν στην λήψη δεδομένων από έναν αισθητήρα
2. Δημιουργούμε ένα νέο job με δείκτη στην TimerFcn που πρέπει να εκτελεστεί και στα δεδομένα UserData.
3. Εφόσον είναι άδεια η ουρά τοποθετούμε το job και υπολογίζουμε την νέα περίοδο που πρέπει το thread να περιμένει πλην του χρόνου που μεσολάβησε για τα παραπάνω.

Βγαίνοντας από την loop εκτελούμε στην StopFcn και το thread τελειώνει την εργασία του.

Νήμα Consumer

Ο consumer εκτελείτε διαρκώς από την αρχή του προγράμματος και σβήνει όταν τελειώσουν όλα τα νήματα Producer. Περιμένει για jobs στην ουρά και εκτελεί την συνάρτηση που περιέχουν με τα αντίστοιχα δεδομένα ως ορίσματα. Αφού εκτελέσει ένα job το διαγράφει από την λίστα και περιμένει το επόμενο.

Ο ελάχιστον αριθμός των consumer για να είναι αποδοτικός ο αλγόριθμος είναι συνάρτηση του μεγέθους της ουράς αλλά κυρίως του φόρτου εργασίας της TimerFcn.

Δεδομένα – Στατιστικά – Δοκιμές

Όλες οι δοκιμές έγιναν σε raspberry pi 4 model B, 4Gb ram

Για την εκτέλεση των πειραμάτων θεωρούμε ότι η TimerFcn παράγει «τυχαίους» αριθμούς και τους αποθηκεύει σε έναν πίνακα. Ο χρόνος εκτέλεσης ανάλογα με το μέγεθος του πίνακα.
Για να βγάλουμε στατιστικά για τους χρόνους ρυθμίζουμε έτσι ώστε η CPU να δουλεύει περίπου στο 75% της συνολικής απόδοσης.

Σε αυτά τα δεδομένα δεν υπάρχουν καθυστερήσεις εξαιτίας γεμάτης ουράς ενώ έχουμε 3 thread consumer που εκτελούν την TimerFcn σε περίπου 10msec.

Θεωρούμε ότι οι χρόνοι για να εκτελέσει ο consumer ένα job είναι η καθυστέρηση στην περίοδο μεταξύ των εκτελέσεων των TimerFcn, ενώ ο χρόνος που κάνει να βάλει ο producer στην ουρά ένα job είναι το drifting time. Το δεύτερο δεν προκαλεί στην πραγματικότητα καθυστέρηση μεταξύ των περιόδων καθώς ο χρόνος που μεσολαβεί προσαρμόζεται ανάλογα στην νέα περίοδο. Έτσι η τελική καθυστέρηση είναι μόνο η οφείλεται μόνο στον εργάτης (consumer)

Παρακάτω αντιστοιχούν τα δεδομένα για τις 4 συνεδρίες.

Πείραμα 1ο : Εκτέλεση όλων των timer διάρκειας 1 ώρας

Γνωρίζουμε ότι ο φόρτος εργασία για κάθε νήμα εργάτη για την εκτέλεση της TimerFcn αντιστοιχεί σε 10msec.

Producers

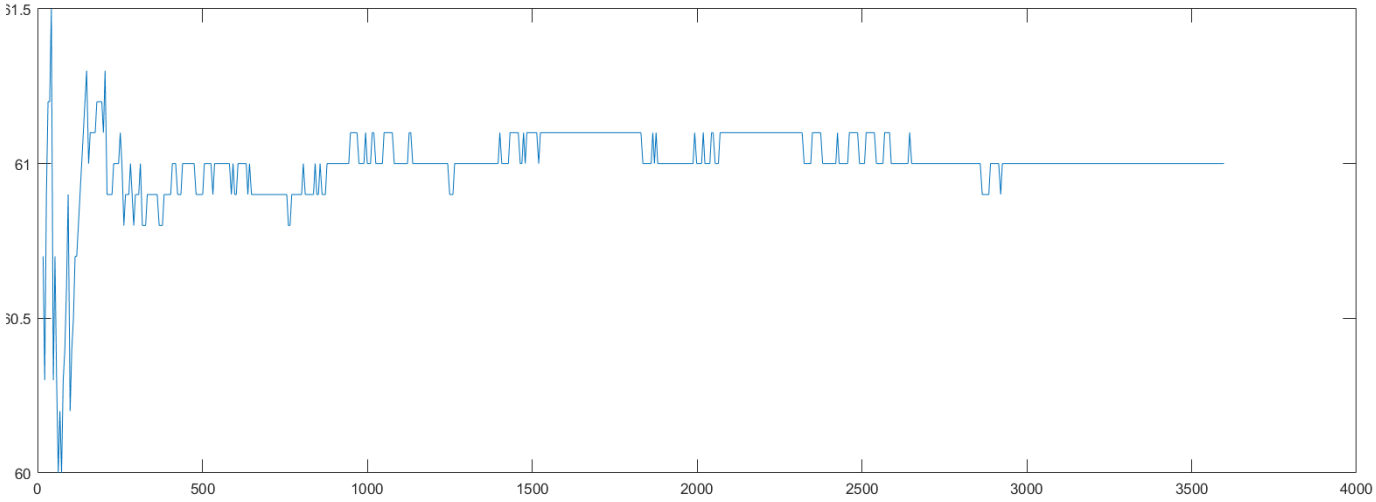
Timer 1 - 1sec drifting	usec	Timer 2 - 0.1sec drifting	usec	Timer 3 - 0.01sec drifting	usec
Ελάχιστο	43	Ελάχιστο	33	Ελάχιστο	33
Μέγιστο	3539	Μέγιστο	8579	Μέγιστο	17154
Μέση τιμή	85.383056	Μέση τιμή	80.245722	Μέση τιμή	77.870564
Τυπική απόκλιση	77.331377	Τυπική απόκλιση	114.912469	Τυπική απόκλιση	66.027198
Διασπορά	5.980142e+03	Διασπορά	1.320488e+04	Διασπορά	4.359591e+03

Consumers

Καθυστέρηση Timer 1	(usec)	Καθυστέρηση Timer 2	(usec)	Καθυστέρηση Timer 3	(usec)
Ελάχιστο	-7385	Ελάχιστο	-7391	Ελάχιστο	-9950
Μέγιστο	7860	Μέγιστο	9966	Μέγιστο	20684
Μέση τιμή	250.071964	Μέση τιμή	252.608017	Μέση τιμή	230.014689
Τυπική απόκλιση	354.472919	Τυπική απόκλιση	506.793207	Τυπική απόκλιση	302.500876
Διασπορά	1.256511e+05	Διασπορά	2.568394e+05	Διασπορά	9.150678e+04

Οι αρνητικές τιμές προκύπτουν εξαιτίας την καθυστέρησης της ουράς, καθώς για 20, 279, 545 φορές αντίστοιχα οι 3 timers έχασαν την περίοδο. Μάλιστα βλέπουμε ότι για τον Timer 3 ο χρόνος 9,9msec που καθυστέρησε είναι σχεδόν ίσος με μια περίοδο. Πράγμα που σημαίνει ότι για τις ανάγκες των RTOS είναι απαγορευτικό.

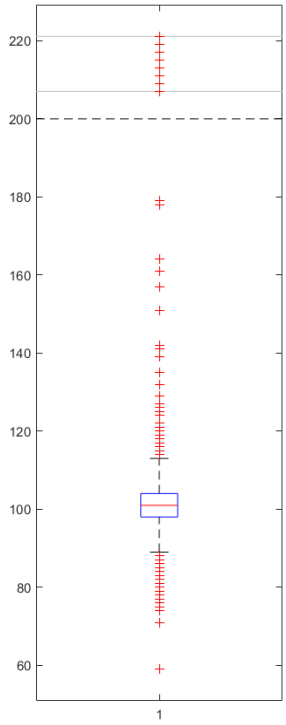
Χρήση CPU



Εικόνα 1 Χρήση της CPU % ανά second λειτουργίας της διεργασίας

Η Χρήση της CPU κυμαίνεται γύρο από το 61% και μάλιστα στην αρχή βλέπουμε την ταλάντωση που οφείλεται στην ταυτόχρονη έναρξη των timers που φθίνει εξαιτίας των διαφορετικών περιόδων.

Πείραμα 2ο : Εκτέλεση διάρκειας 1 ώρας μόνο και τον timer με περίοδο 1sec



Timer drifting – Χρόνος producer (usec):

Ελάχιστο:	59
Μέγιστο:	12638
Μέση τιμή:	119.430833
Τυπική απόκλιση:	431.068703
Διασπορά:	1.858202e+05

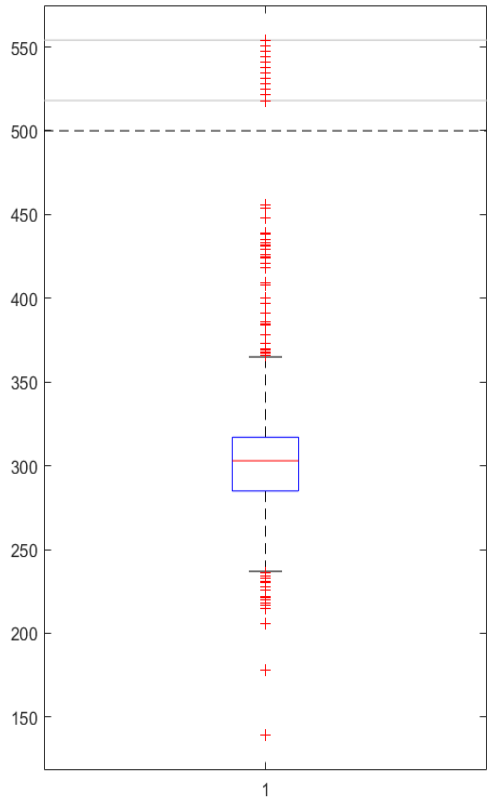
Βλέπουμε ότι ό μέγιστος χρόνος drifting είναι 12,64msec << 1sec που είναι η περίοδος. Αυτό είναι λογικό καθώς η ουρά παραμένει άδεια.

Εικόνα 2: Boxplot για τους χρόνους drifting σε usec

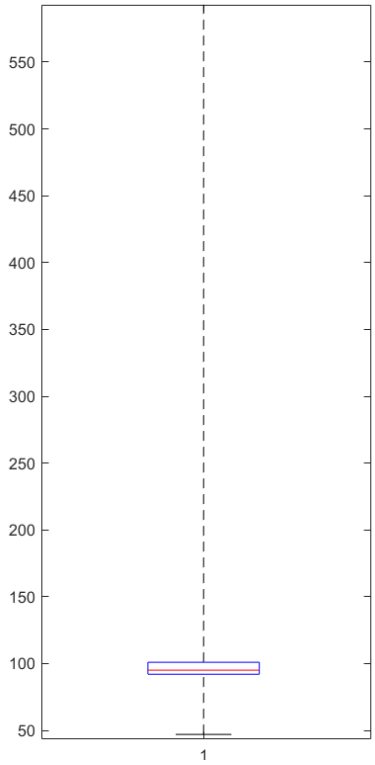
Χρόνοι consumer (usec) :

Ελάχιστο:	139
Μέγιστο:	25387
Μέση τιμή:	343.865796
Τυπική απόκλιση:	878.929071
Διασπορά:	7.725163e+05

Η χρήση της CPU στο παραπάνω πείραμα για την διεργασία μας ήταν μόλις στο 11.7% – 11.8% σε όλη την διάρκεια για αυτό και δεν την παρουσιάζω λεπτομερέστερα.



Πείραμα 3ο : Εκτέλεση διάρκειας 1 ώρας μόνο και τον timer με περίοδο 0,1sec



Timer drifting – Χρόνος producer (usec):

Ελάχιστο:	47
Μέγιστο:	11526
Μέση τιμή:	99.035167
Τυπική απόκλιση:	170.823874
Διασπορά:	2.918080e+04

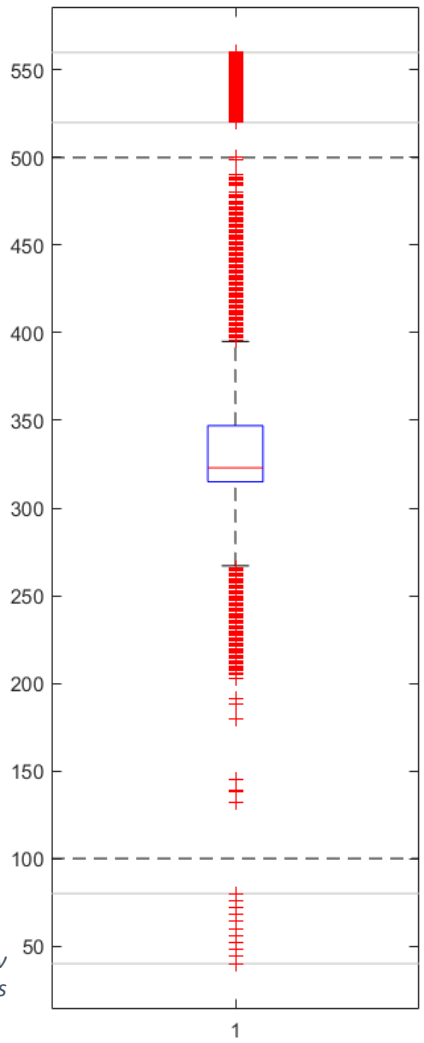
Εικόνα 3: Boxplot για τους χρόνους drifting σε usec

Χρόνοι consumer (usec) :

Ελάχιστο:	-1547
Μέγιστο:	18352
Μέση τιμή:	339.138309
Τυπική απόκλιση:	341.142418
Διασπορά:	1.163781e+05

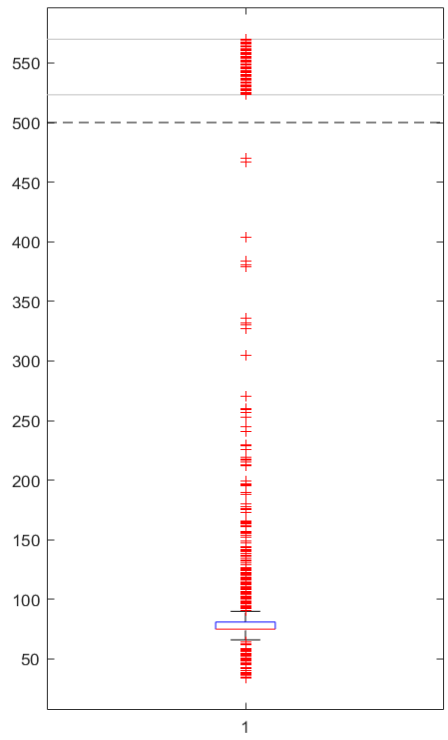
Η χρήση της CPU στο παραπάνω πείραμα για την διεργασία μας ήταν 8.6%-8.8% σε όλη την διάρκεια για αυτό και δεν την παρουσιάζω λεπτομερέστερα.

Παρότι πρόκειται για διεργασία που εκτελείται συχνότερα η χρήση της CPU σε σχέση με το πείραμα 2 είναι ελάχιστα μικρότερη καθώς ο χρόνος της TimerFcn είναι μικρότερος. Αυτό όμως δεν ισχύει στο επόμενο πείραμα που οι consumers έχουν το ίδιο φορτίο δουλειάς αλλά στο 1/10 του χρόνου.



Εικόνα 4: Boxplot με την καθυστέρηση των consumers

Πείραμα4 : Εκτέλεση διάρκειας 1 ώρας μόνο και τον timer με περίοδο 0,01sec



Εικόνα 5: Boxplot για τους χρόνους drifting σε usec

Timer drifting – Χρόνος producer (usec):

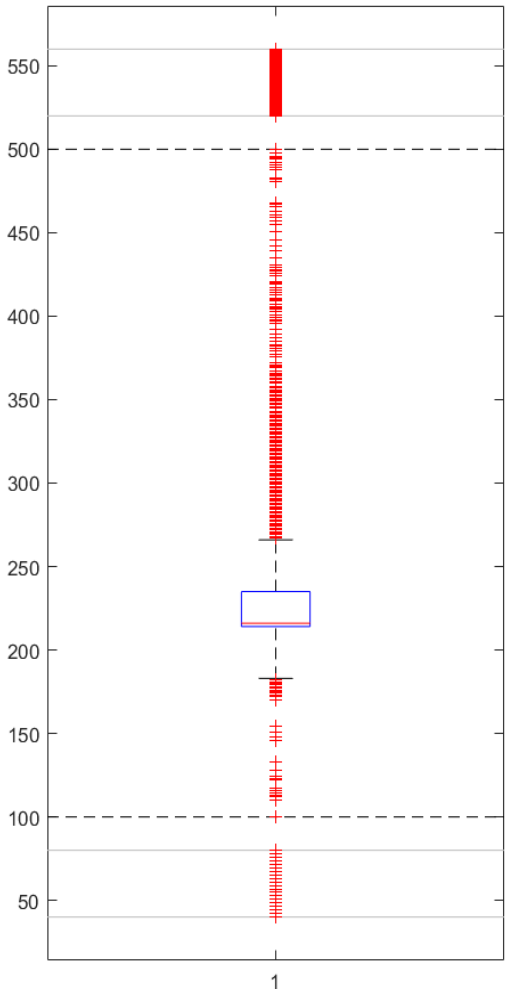
Ελάχιστο:	34
Μέγιστο:	10398
Μέση τιμή:	77.963631
Τυπική απόκλιση:	37.312442
Διασπορά:	1.392218e+03

Χρόνοι consumer (usec) :

Ελάχιστο:	-1053
Μέγιστο:	14500
Μέση τιμή:	225.285179
Τυπική απόκλιση:	100.121124
Διασπορά:	1.002424e+04

Η χρήση της CPU στο πείραμα 3 για τον Timer με περίοδο 10msec ήταν στο 45.7%-45.8% σε όλη την διάρκεια.

Η αυξημένη χρήση της CPU σε σχέση με τα προηγούμενα πειράματα είναι λογική καθώς ενώ ο χρόνος της TimerFcn παραμένει ο ίδιος ενώ η περίοδος είναι μικρότερη με αποτέλεσμα η ουρά να είναι λιγότερη ώρα άδεια.



Εικόνα 6: Boxplot με την καθυστέρηση των consumers

Βέλτιστος συνδυασμός παραμέτρων

Έχοντας βγάλει τα παραπάνω στατιστικά μπορούμε να καταλήξουμε σε ένα ασφαλές συμπέρασμα σχετικά με το μέγεθος της ουράς τον χρόνο της TimerFcn την περίοδο και τον αριθμό τον consumer.

Σε λειτουργία πραγματικού χρόνου είναι απαραίτητο η TimerFcn να τελειώσει πριν από την επόμενη περίοδο, ώστε να μην χαθεί η επόμενη εκτέλεση. Στην υλοποίησή μας συμμετέχει και η ουρά η οποία είναι αναγκαία εφόσον λειτουργούν περισσότεροι από έναν εργάτη και έχουμε πολλαπλούς Timers να προσθέτουν jobs. Για απλούστευση αρχικά θεωρούμε έναν Timer. Επομένως ανεξάρτητα από το μέγεθος της ουράς πρέπει

$$t_{TimerFcn} + T_{delays} < t_{Period}$$

Και

$$t_{drifting} < t_{Period}$$

Στο παραπάνω πρόβλημα κατά μέσω όρο για την περίπτωση $t_{Period} = 0.01sec$

$$t_{TimerFcn} = 0{,}001sec \ , T_{delays} = 0.000225\text{sec}$$

Το $t_{TimerFcn}$ παραμένει το ίδιο κατά μέσω όρο άρα έχουμε την δυνατότητα να αυξήσουμε τον χρόνο της στα 9msec.

Γενικεύοντας και για τους αλλους 2 Timer τα παραπάνω δεδομένα μπορούμε εύκολα να συμπεράνουμε ότι μια ασφαλή επιλογή παραμέτρων είναι η εξής:

Για συγκεκριμένο t_{Period} η TimerFcn πρέπει να καταναλώνει το πολύ $t_{Period} - 1msec$. Επιπλεον όσο ισχύει αυτό και το $t_{drifting} < t_{Period}$. Στην περίπτωση που ξεπεράσει τον παραπάνω χρόνο εκτέλεσης θα γεμίσει η ουρά με αποτέλεσμα να αυξηθεί το $t_{drifting}$. Στην πραγματικότητα όμως η σημαντική συνθήκη για να μην χαθεί κάποια περίοδος είναι η $t_{drifting} < t_{Period}$. Επομένως βλέπουμε ότι το όριο είναι στην πραγματικότητα λίγο μικρότερο από 2 φορές την περίοδο ώστε να έχουμε συνέπεια στην εκτέλεση των TimerFcn και την ουρά επαρκώς άδεια ώστε σε κάθε περίοδο να μπαίνει ένα job.

Μεταφράζοντας αυτά στο δικό μας πρόβλημα με 3 Timers των 1sec, 0.1sec και 0,01sec το βέλτιστο είναι να ορίσουμε μια ουρά 3+1 θέσεων (η 4^{η} είναι για την περίπτωση που έρθουν και οι 3 μαζί διεργασίες) και αντίστοιχα οι TimerFcn να εκτελούνται σε χρόνους >950msec >95msec >9msec. Εκτελώντας το παραπάνω πείραμα με την εντολή ./Timers.o 8000 500 80 βλέπουμε ότι η χρήση της CPU φτάνει στο 190% ,δεν καθηστερεί καμιά περίοδος και η ουρά δεν μένει ποτέ άδεια. Το ποσοστό αυτό 190>100% οφείλεται στο ότι το Rasberry pi 4, έχει 4 λογικούς πυρήνες και μπορεί να εκτελεί παράλληλα τα νήματα.

Οι παραπάνω δοκιμές ισχύουν μόνο για το Rasberry pi 4, model B