Producers Consumers Pthreads Implementation in C

Github Link: https://github.com/LambisElef/prod_cons_timer

Ανάλυση Κώδικα:

Για την καλύτερη οργάνωση του κώδικα, οι δομές και οι συναρτήσεις που αφορούσαν την ουρά και τον timer μεταφέρθηκαν σε ξεχωριστά αρχεία. Αξίζει να αναφερθεί ότι στη δομή του Timer προστέθηκαν πέρα από τις παραμέτρους period, tasksToExecute, startDelay, timerFcn() και errorFcn() και κάποιες παράμετροι που χρειάζεται ένα producer thread, όπως ένας pointer στην ουρά και κάποιοι για τα στατιστικά που θέλουμε να κρατήσουμε. Η συνάρτηση timerInit() κάνει την αρχικοποίηση του timer και πρέπει να κληθεί πριν την start(Timer *timer). Η start(Timer *timer) απλά δημιουργεί ένα producer thread.

Στη main() αρχικά ζητούνται από το χρήστη το μέγεθος της ουράς, ο χρόνος σε δευτερόλεπτα που θα τρέξουν ο(ι) timer(s) και ποιοι timers θα τρέξουν. Το πλήθος των consumer threads που θα δημιουργηθούν καθορίζεται σε ένα επαναληπτικό βρόγχο μέσα στον πηγαίο κώδικα της main(), ώστε να μπορούν να γίνουν εύκολα έλεγχοι για διάφορα πλήθη consumer threads.

Σε ότι αφορά τους χρόνους που μετρήθηκαν για μετέπειτα στατιστική ανάλυση, ο χρόνος tJobWait μετριέται από τη στιγμή που μια δουλειά μπαίνει στην ουρά μέχρι βγει, δηλαδή ο χρόνος παραμονής της ουράς στην ουρά ή ο χρόνος για να τη βγάλει ένας consumer. Ο χρόνος tJobIn αφορά το χρόνο που κάνει ένας producer να προσθέσει μια δουλειά στην ουρά. Ο χρόνος tJobDur αφορά τη διάρκεια εκτέλεσης της δουλειάς από τον consumer. Ο χρόνος tDrift αφορά τη χρονική μετατόπιση που παρατηρείται μεταξύ διαδοχικών εκτελέσεων του timer πέραν της περιόδου του, κάτι που συμβαίνει λόγω του χρόνου που παίρνει ο ίδιος ο producer για να εκτελεστεί, αλλά και λόγω λανθανουσών διακοπών που επιβάλλονται από το λειτουργικό. Επιλέχθηκε να είναι πάντα μεγαλύτερος του μηδενός και μικρότερος της περιόδου του timer που αφορά.

Ακόμη, χρησιμοποιήθηκαν δύο mutex για την καταγραφή των στατιστικών στη μνήμη χωρίς να γίνεται χρήση του mutex της ουράς, εμποδίζοντας χωρίς λόγο τη χρήση της τελευταίας. Το πρώτο mutex είναι κοινό για όλους τους producers και το δεύτερο κοινό για όλους τους consumers.

Στη συνέχεια, δημιουργούνται τα consumer threads και εκκινούνται οι timers. Η main() κοιμάται μέχρι να τελειώσουν οι timers, τερματίζει με ομαλό και ασφαλή τρόπο τους consumers οι οποίοι περιμένουν σε μια άδεια ουρά, θέτοντας το flag quit, σώζει τα στατιστικά σε αρχεία και απελευθερώνει τη μνήμη που δεσμεύτηκε για αυτό το πείραμα.

Σε ότι αφορά την producer(), αυτή ουσιαστικά υλοποιεί όλη τη λογική του timer που την ξεκίνησε, δημιουργώντας μια δουλειά ανά period χρόνο και tasksToExecute δουλειές συνολικά. Αρχικά δημιουργεί τα ορίσματα που θα δωθούν στην timerFcn() και κατόπιν ελέγχει αν η ουρά είναι γεμάτη. Στην περίπτωση που είναι, η δουλειά χάνεται και τρέχει η errorFcn() αυξάνοντας το μετρητή jobsLostCounter, ενώ αν δεν είναι, η δουλειά μπαίνει κανονικά στην ουρά και η producer() κοιμάται μέχρι την επόμενη περίοδό της.

Υπάρχουν αναλυτικά σχόλια μέσα στον κώδικα για επεξήγηση της λειτουργίας του γραμμή προς γραμμή.

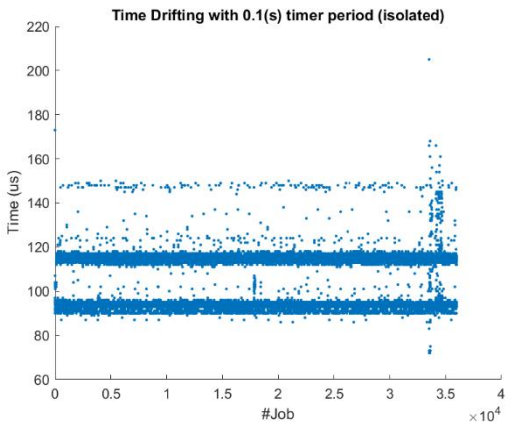
Τεχνικές πληροφορίες:

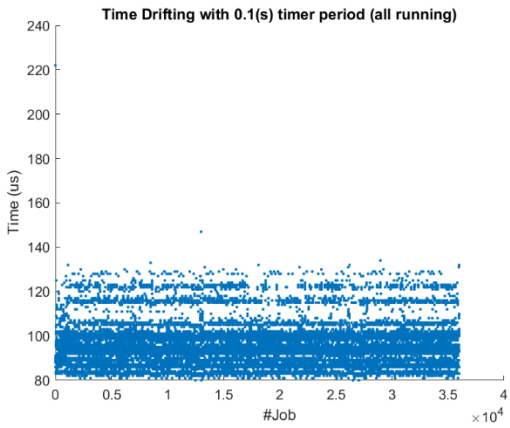
Τα πειράματα έγιναν σε Raspberry Pi 3 B στο image που μας δόθηκε. Το κάθε πείραμα διήρκησε 1 ώρα. Επιλέχθηκε μέγεθος ουράς ίσο με 3 και αριθμός consumer threads ίσος με 1. Η συχνότητα λειτουργίας του CPU ήταν σταθερή στα 600MHz και η θερμοκρασία του κάτω από 50C καθ' όλη τη διάρκεια των πειραμάτων. Ο κώδικας έγινε compile χρησιμοποιώντας τον επίσημο **cross compiler** του **gcc-10** για ARM με τα flags **-O3 -mcpu=cortex-a53 -mfpu=neon-fp-armv8**. Η χρήση CPU από κάθε πείραμα έγινε με την εντολή "**ps** -p <PID> -o %cpu" που επιστρέφει τη μέση χρήση της CPU από την εκκίνηση της διεργασίας με ακρίβεια ενός δεκαδικού ψηφίου. Περιλαμβάνει όλα τα threads που έχουν δημιουργηθεί από αυτή τη διεργασία, ακόμη και αν αυτά έχουν διαφορετικά <PID>. Για την εύρεση του <PID> και μια γενικότερη εικόνα της χρήσης των πόρων του συστήματος χρησιμοποιήθηκε το "**htop**".

Αποτελέσματα:

Χρόνος ολίσθησης (us) του timer για κάθε περίοδο					
Timer(s)	Μέσος όρος	Τυπική Απόκλιση	Διάμεσος	Ελάχιστος	Μέγιστος
1s	102.59	5.19	103	90	187
(All) 1s	90.51	3.63	90	87	280
0.1s	107.71	10.56	114	72	205
All (0.1s)	95.69	7.85	94	80	222
0.01s	93.80	6.37	92	80	171
All (0.01s)	97.27	8.54	95	77	263
*Με τη λέξη **All** στη στήλη **Timer(s)** υπονοείται ότι έτρεχαν και οι 3 timers με περιόδους **1s**, **0.1s** και **0.01s** ταυτόχρονα.					

Κοιτάζοντας τον παραπάνω πίνακα, μία απλή παρατήρηση είναι ότι ο μέσος όρος, η τυπική απόκλιση και η διάμεσος του χρόνου ολίσθησης ήταν πάντα μικρότερα όταν έτρεχαν και οι 3 timers μαζί, με εξαίρεση τον timer με περίοδο 10ms όπου συνέβη το ακριβώς αντίθετο. Από την άλλη, ο μέγιστος χρόνος ολίσθησης που παρατηρούνταν όταν έτρεχαν όλοι οι timers μαζί ήταν υψηλότερος απ' ότι όταν έτρεχαν απομονωμένοι όπως ήταν αναμενόμενο, αφού υπάρχει η περίπτωση οι timers να "σκάσουν" ταυτόχρονα και ο ένας να πρέπει να περιμένει τον άλλο για να γράψει στην ουρά. Για παράδειγμα παρακάτω βρίσκονται δύο διαγράμματα για τον χρόνο ολίσθησης του timer με περίοδο 0.1s, το ένα όταν έτρεχε απομονωμένος και το άλλο όταν έτρεχε μαζί με τους άλλους δύο. Είναι εμφανές ότι υπάρχει μια συγκέντρωση τιμών σε υψηλότερη χρονική στάθμη όταν έτρεχαν όλοι μαζί.





Ακολουθούν δύο πίνακες με τη στατιστική ανάλυση των 2 χρόνων που ζητήθηκαν στην εκφώνηση:

Χρόνος (us) που ξοδεύει η producer για να βάλει μια κλήση στην ουρά					
Timer(s)	Μέσος όρος	Τυπική Απόκλιση	Διάμεσος	Ελάχιστος	Μέγιστος
1s	4.19	1.57	4	3	79
0.1s	3.96	0.62	4	2	69
0.01s	3.72	0.48	4	3	71
All (3)	4.05	1.47	4	3	199

Χρόνος (us) που ξοδεύει η consumer για να βγάλει μια κλήση από την ουρά (ή αλλιώς χρόνος αναμονής της κλήσης στην ουρά)					
Timer(s)	Μέσος όρος	Τυπική Απόκλιση	Διάμεσος	Ελάχιστος	Μέγιστος
1s	25.50	2.56	25	24	52
0.1s	24.67	2.86	24	12	63
0.01s	23.35	1.21	23	22	64
All (3)	25.75	4.34	24	1	301

Timer(s)	**Χρήση CPU**
1s	0.0%
0.1s	0.0%
0.01s	0.9%
All (3)	1.0%

Στον διπλανό πίνακα φαίνεται η χρήση CPU ανάλογα με το πείραμα. Τα αποτελέσματα είναι τα αναμενόμενα, καθώς ένας timer με 10 φορές μικρότερη περίοδο από έναν άλλο, θα έπρεπε να χρησιμοποιεί τη CPU 10 φορές περισσότερο. Αυτό μπορεί να παρατηρηθεί αν συγκρίνουμε τη σχεδόν μηδενική χρήση του CPU από τον 0.1s timer σε σχέση με τον 0.01s timer, καθώς και το ότι η κύρια χρήση CPU όταν τρέχουν και οι τρεις timers οφείλεται στον timer με τη μικρότερη περίοδο.

Για λειτουργία πραγματικού χρόνου, το μέγεθος της ουράς εξαρτάται αρχικά από τον αριθμό των timers που τρέχουν. Με την υπόθεση ότι κάθε timer βάζει μια δουλειά στην ουρά, το μέγεθος της ουράς πρέπει να είναι τουλάχιστον ίσο με τον αριθμό των timers. Το μέγεθος της ουράς θα έπρεπε να μεγαλώσει σε περίπτωση που κάποιος timer εκτελούνταν ασύγχρονα και πρόκυπταν ριπές δουλειών που θα έπρεπε να προστεθούν στην ουρά.

Ο χρόνος εκτέλεσης της timerFcn() πρέπει να είναι σίγουρα μικρότερος της περιόδου του timer που την προσθέτει στην ουρά, ειδάλλως θα είχαμε ένα ασταθές σύστημα με διαρκώς αυξανόμενο αριθμό κλήσεων προς εκτέλεση, καθώς ο αριθμός των consumers είναι πεπερασμένος.

Όσον αφορά στον αριθμό των consumers, αυτός εξαρτάται από το τι θεωρούμε ως έγκαιρη έναρξη εκτέλεσης της δουλειάς. Για την αμεσότερη έξοδο μιας κλήσης από την ουρά, ο αριθμός των consumers πρέπει να είναι τουλάχιστον ίσος με τον αριθμό των producers. Βέβαια, ο αριθμός των consumers θα μπορούσε να είναι και μικρότερος, για παράδειγμα εάν ένας consumer προλάβαινε να εκτελέσει όλες τις δουλειές σε χρόνο μικρότερο από αυτόν της περιόδου του timer με την μικρότερη περίοδο, αρκεί να μη μας πείραζε ενδεχόμενη καθυστέρηση της εκτέλεσης κάποιων δουλειών το πολύ μέχρι την περίοδο του timer με την μικρότερη περίοδο. Συγκεκριμένα στην εργασία αυτή, η timerFcn() που επιλέχθηκε και για τους τρεις timers εκτελεί άθροισμα ημιτόνων 100-200 γωνιών από ομοιόμορφη κατανομή. Ο μέσος χρόνος εκτέλεσής της ήταν 57us και η τυπική του απόκλιση 11us. Επομένως, ακόμη και ένας consumer ήταν αρκετός για την εκτέλεση των κλήσεων που προθέτονταν και από τους τρεις timers.