Ενσωματωμένα Συστήματα Πραγματικού Χρόνου

Αποστόλης Γεροδήμος

ΑΕΜ 7862

Απαλλακτική Εργασία 2020

https://github.com/boicotaz/rtes2020

Ανάλυση Δομής Αρχείων

object_definitions.c: Περιλαμβάνει τις βιβλιοθήκες καθώς και των ορισμό όλων των δομών που χρησιμοποιήθηκαν

main.c: Μαζεύει τα αρχεία που περιέχουν την λογική του κώδικα. Αρχικοποίηση όλων των απαραίτητων οντοτήτων και εκκίνηση της βασικής διεργασίας καθώς και όλων των thread

Ανάλυση Δομών

queue:

Η ουρά αποτελείται από τον buffer που κρατάει τα **entries**

struct QueueEntry entry: περιλαμβάνει την διεύθυνση της συνάρτησης (callback) που θα εκτελεστεί, τα ορίσματα της (*struct CallbackArgs*) και τα απαραίτητα timestamps για τις μετρήσεις

Η υλοποίηση της ουράς γίνεται με λογική cyclic buffer. Μπορεί να εξυπηρετείται από παραπάνω του ενός consumer

long head δηλώνει στον consumer από που πρέπει να παίρνει entries

long tail δηλώνει στον producer που πρέπει να προσθέτει entries

int full, empty χρησιμοποιούνται από τα thread για να κρίνουν αν θα πρέπει να μπουν σε sleep state

int onlineTimers το consumer thread τρέχει μέσα σε μία while loop με έλεγχο αυτήν την μεταβλητή, με σκοπό το thread να κλείσει αυτόματα με το τέλος όλων των timer. Η λήξη ενός timer σημαίνει την λήξη του υπεύθυνου producer thread, το οποίο την μικραίνει κατά ένα.

pthread_cond_t *queueNotEmptyOrTimerFinished: Χρησιμοποιείται από τον producer για να ξυπνήσει το consumer thread σε 2 περιπτώσεις:

1) ο producer πρόσθεσε νέο entry
2) έχει χρήση στην περίπτωση που το producer thread κλείνει και το consumer thread κοιμάται περιμένοντας το condition_signal.

Αυτό το condition έχει 2 σημασίες στον κώδικα, το consumer thread οφείλει να το μεταφράσει σωστά

pthread_cond_t notFull: με το που βγάλει ένα entry από τον buffer ο consumer, στέλνει αυτό το signal για να ξυπνήσει τον producer δηλώνοντας πως υπάρχει ελεύθερος χώρος στην ουρά

void queueAdd(*queue *q, QueueEntry in****)***: χρησιμοποιείται από τον producer για να βάλει ένα νέο entry στον buffer της ουράς. Περιέχει λογική για να παραμείνει ο buffer κυκλικός. Περιέχει λογική για να αλλάξει το state της ουράς σε full

void * queuePop(*queue *q****)***:χρησιμοποιείται από τον consumer για να πάρει το νωρίτερο entry από τον buffer της ουράς. Περιέχει λογική για να παραμείνει ο buffer κυκλικός. Περιέχει λογική για να αλλάξει το state της ουράς σε empty

void queueDelete(*queue *q****)***: αποδεσμεύει την μνήμη του buffer, conditions, mutexes.

queue *queueInit(*int queueSize****)***: Αρχικοποιεί την ουρά και τις μεταβλητές της.

Timer:

Κάθε producer thread εξυπηρετεί έναν timer

struct Timer *StartFcn(…): Δημιουργία ενός Instance Timer και αρχικοποίηση των μεταβλητών του, που ορίστηκαν από τα specification της εκφώνησης.

Επιπλέον μεταβλητές είναι

timer_id βοηθητική μεταβλητή για τις μετρήσεις

*queue *q* κάθε timer κρατάει μία αναφορά στην ουρά

void SetInterval(*void(*timerFcn)(), struct Timer *t, CallbackArgs *ca* ***)***:

timerFcn: Η συνάρτηση που εκτελείται από τον consumer

*CallbackArgs *ca*: Πρόκειται για ένα struct που περιέχει τα ορίσματα της timerFcn

*struct Timer *t*: Αν και πρόκειται για συνάρτηση μέλος του struct Timer. Η C δεν μεταφέρει το context, όπως η C++ με τον pointer this

Εδώ δημιουργείται το thread του producer Με όρισμα *το struct ProducerData* που περιέχει αναφορά στον Timer και στο entry που θα εντάξει στην ουρά ο producer (check object_definitions.c)

Ανάλυση Κώδικα

Note: Για περισσότερη λεπτομέρεια δες την ανάλυση των Δομών

Ξεκινάμε με την αρχικοποίηση της ουράς και των timer

Το consumer thread ξεκινά με όρισμα την ουρά και παραμένει ανενεργό, καθώς η ουρά είναι άδεια, ως ότου το πρώτο **entry** μπει στην ουρά από το/α producer thread/s.

> **How does consumer stay inactive?** Εάν είναι άδεια η ουρά, το thread εκτελεί την *pthread_cond_wait*, που το βάζει για ύπνο μέχρι ότου λάβει το condition_signal από το/α producer thread/s

Τώρα η δουλειά γίνεται στο producer thread, ξεκινώντας με την σειρά του ελέγχει αν η ουρά είναι γεμάτη, εφόσον δεν είναι εκτελεί την queueAdd

Στην συνέχεια στέλνει το signal *queueNotEmptyOrTimerFinished* για να ξυπνήσει το consumer thread και καλεί την *usleep*. Με το πέρας του *Period* ελέγχει εάν η ουρά είναι γεμάτη, εάν **δεν** είναι συνεχίζει με την ίδια λογική.

Εάν η ουρά είναι γεμάτη τώρα με την σειρά του πέφτει για ύπνο εκτελώντας την *pthread_cond_wait* περιμένοντας το σήμα *notFull* από τον consumer.

Γυρνάμε στον consumer εφόσον η ουρά δεν είναι άδεια η λάβει το signal από τον producer βγάζει ένα entry από την ουρά και στέλνει το σήμα *notFull* για τυχόν producer που βρίσκεται σε sleep state κάνοντας χρήση του pthread_cond_signal. Αυτή η συνάρτηση στέλνει το σήμα το **πολύ** σε ένα producer για να μην έχουμε race conditions στις μεταβλητές που διαχειρίζονται το state της ουράς

Στην συνέχεια παίρνει από το entry την διεύθυνση της συνάρτησης και τα ορίσματα της και τα εκτελεί. Και συνεχίζει τον ίδιο κύκλο εκτέλεσης

Συνθήκες λήξης

consumer: ελέγχει την μεταβλητή *onlineTimers* της ουράς για να συμπεράνει αν έχουν τελειώσει όλοι οι timers

producer: Ελέγχει την μεταβλητή *TasksToExecute* του Timer για τον οποίο είναι υπεύθυνος. Σε κάθε επανάληψη της εκτέλεσης του την μειώνει κατά 1. Εφόσον δεν έχει άλλα tasks για την ουρά απελευθερώνει την μνήμη που δέσμευσε ο Timer, εκτελώντας την συνάρτηση *StopFucn.* Μειώνει κατά ένα την *OnlineTimers* της ουράς και εκτελεί την *pthread_cond_broadcast* με το σήμα *qeueNotEmptyOrTimerFinished* για να ξυπνήσει τυχόν consumers που κοιμούνται περιμένοντας ακόμα νέο entry.

Ανάλυση Αποτελεσμάτων

Από την εκφώνηση μας ζητήθηκε να τρέξουμε τα πειράματα για 3 timers με περιόδους 1s, 0.1s, 0.01s και να συγκρίνουμε την απομονωμένη συμπεριφορά τους με αυτή που τρέχουν και οι 3ς ταυτόχρονα.

Για να έχουμε πιο ελέγξιμα συμπεράσματα θα βάλουμε και μερικά ακόμα requirements.

Σε κάθε επανάληψη του πειράματος θα έχουμε μόνο 1 consumer

Ο χρόνος εκτέλεσης της *TimerFcn θα είναι γύρω στα 0.01s ή 10000us*. Αυτό γίνεται κάνοντας χρήση της usleep.

Το μήκος της ουράς θα είναι 1. Ώστε να είναι πιο έντονη η αλλαγή της συμπεριφοράς όταν εξυπηρετούμε 3 timers ταυτόχρονα.

Drift

Το **drift** υπολογίστηκε με ως εξής:

> Ο χρόνος μεταξύ 2 διαδοχικών εκτελέσεων του producer thread, χωρίς να παίρνουμε υπόψιν το χρόνο που κοιμάται το thread.
>
> Ο producer τρέχει μέσα σε μία while loop ελέγχοντας την μεταβλητή TasksToExecute, έτσι τα *timestamps* τα παίρνουμε σε κάθε αρχή της while loop

Period: 1s

	All Together	Alone
mean	12192.13 us	277.26 us
max	25485 us	7540 us
min	110 us	64 us
std	5529.395 us	157.06 us
median	14906 us	266 us

Period: 0.1s

	All Together	Alone
mean	1839.77 us	247.19 us
max	23748 us	16217 us
min	127 us	48 us
std	1205.75 us	165.32 us
median	1554 us	22 us

Period: 0.01s

	All Together	Alone
mean	1321.19 us	227.71 us
max	33659 us	37478 us
min	52 us	71 us
std	3252.42 us	498.52 us
median	153 us	184 us

Το πρώτο ενδιαφέρον είναι πως όλες οι απομονωμένες εκτελέσεις παρουσιάζουν σχεδόν ίδιες μέσες τιμές, με μικρή τυπική απόκλιση. Γεγονός που ήταν αναμενόμενο,

ο consumer χρειάζεται χρόνο της τάξης 0.01s για να εκτελέσει την timerFcn

ο πιο γρήγορος producer κάνει produce σε χρόνο της τάξης του 0.01s

Σε αυτό το πλαίσιο αν και η ουρά έχει μήκος 1 σπάνια κάποιος producer να συναντήσει την ουρά γεμάτη με αποτέλεσμα να αυξήσει το drift time, δηλ. το σύστημα είναι ευσταθές

Από την άλλη αν συγκρίνουμε τις απομονωμένες με τις ομαδικές βλέπουμε μεγάλη διαφορά στο drift time

Καθώς το μήκος της ουράς είναι 1 οι producers μπορεί να συναντήσουν την ουρά γεμάτη. Βέβαια, αυτό από μόνο του δεν δικαιολογεί αυτή την διαφορά. Καθώς το period του γρηγορότερου producer είναι της ίδιας τάξης με τον χρόνο που χρειάζεται ο consumer για να εκτελέσει την timerFcn, τότε οποιοδήποτε νέο φόρτο θα σπάσει την οριακή ευστάθεια του συστήματος.

Producer-Consumer

Μας ζητήθηκε να μελετήσουμε τον χρόνο

που θέλει ο producer για να βάλει μία κλήση στην ουρά (για συντομία **produce time)**

που θέλει ο consumer για να βγάλει μία κλήση στην ουρά (για συντομία **consume time**)

Για τις μετρήσεις

produce time = μετράει τον χρόνο από την εκκίνηση producer μέχρι να καλέσει την queueAdd

consume time = μετράει τον χρόνο από το σημείο που ο producer έβαλε το entry στην ουρά μέχρι ο consumer να καλέσει την queuePop

All together

	Produce time	Consume time
	1312.44 us	9195.14 us
max	33488 us	26174 us
min	11 us	24 us
std	3282.27 us	2973.11 us
median	22 us	10090 us

1 Second

	Produce time	Consume time
mean	41.71 us	61.71 us
max	2669 us	2841 us
min	24 us	15 us
std	58.40 us	67.65 us
median	35 us	53 us

0.1 second

	Produce time	Consume time
mean	34.21 us	51.51 us
max	15878 us	4855 us
min	16 us	21 us
std	124.99 us	77.52 us
median	29 us	50 us

0.01 second

	Produce time	Consume
mean	29.01 us	299.74 us
max	37137 us	20716 us
min	12 us	24 us
std	140.95 us	1473.80 us
median	24 us	53 us

Από τους παραπάνω πίνακες φαίνεται ξεκάθαρα ότι η από κοινού εκτέλεση παρουσιάζει τους χειρότερους χρόνους. Ενώ από τις απομονωμένες εκτελέσεις βλέπουμε πολύ θετικά αποτελέσματα

Για να εξηγήσουμε όμως γιατί συμβαίνει αυτό, θα πρέπει να μετρήσουμε

πόσο tasks μπαίνουν στην ουρά ανά μονάδα χρόνου

πόσο tasks βγαίνουν από την ουρά ανά μονάδα χρόνου

έστω ότι ανάγουμε τον χρόνο σε αυτό του γρηγορότερου producer δηλαδή 10000 us, έτσι έχουμε

Από τους producers: $1\frac{TasksIn}{Time} + 10\frac{TasksIn}{Time} + 100\frac{TasksIn}{Time} = 111\ TasksIn/Time$

Ενώ ο consumer χρειάζεται τουλάχιστον 10000 us για να βγάλει ένα task δηλ.: $1\ TaskOut/Time$

Υπό αυτό το πλαίσιο το σύστημα μας δεν μπορεί να εγγυηθεί έγκυρη εκτέλεση

CPU Usage

Για να ελέγξουμε την κατανάλωση πόρων της CPU για κάθε πείραμα κάναμε χρήση της εντολής *ps*

Mean CPU Usage
All together: 1.67%
Second: almost 0%

Demi Second: 0.15%
Centi Second: 1.52%

Η χρήση σε όλα τα πειράματα είναι αρκετά ασήμαντη, γεγονός που ήταν αναμενόμενο καθώς για να πετύχουμε τον χρόνο εκτέλεσης της TimerFcn έκανα χρήση της usleep

Για λειτουργία πραγματικού χρόνου και έγκαιρη έναρξη, πόσο μεγάλη πρέπει να είναι η ουρά, μέχρι πόσο χρόνο να καταναλώνει η TimerFcn, με ποια περίοδο, και πόσοι πρέπει να είναι οι εργάτες;

Για να έχουμε πραγματικό χρόνο, πρέπει να γνωρίζουμε τις συνθήκες υπό τις οποίες μπορούμε να ορίσουμε ένα αυστηρό χρονικό όριο που δεν πρέπει να παραβιάζεται σε όλη την διάρκεια του πειράματος

Από την άλλη για να έχουμε **και** έγκυρη εκτέλεση πρέπει αυτό το όριο να είναι αρκετά μικρό σε σχέση με την περίοδο κατά την οποία βάζουμε tasks προς εκτέλεση στην ουρά

Για να πετύχουμε αυτές τις προδιαγραφές πρέπει

$$\frac{TaskIn}{Time} \leq \frac{TasksOut}{Time}$$

Για να επαληθεύσουμε αυτήν την ανισότητα τρέξαμε διάφορα πειράματα για *TasksToExecute* 30-80

Ξεκινάμε με ορισμούς

$TimeUnit = 10000\ us$

Μήκος ουράς = L

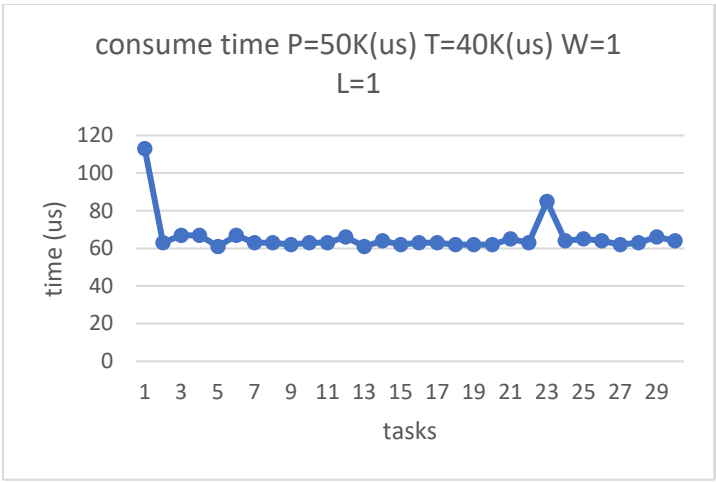
Μέγιστος Χρόνος TimerFcn = T

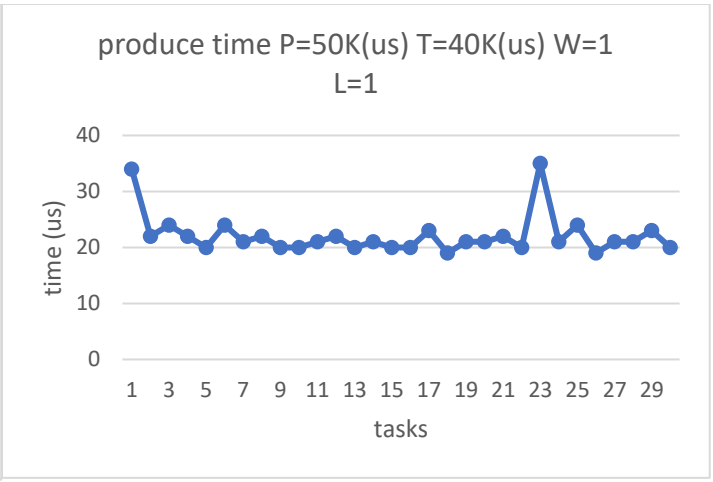
Περίοδος producer = P

Consumer Εργάτες = W

produce time = μετράει τον χρόνο από την εκκίνηση producer μέχρι να καλέσει την queueAdd

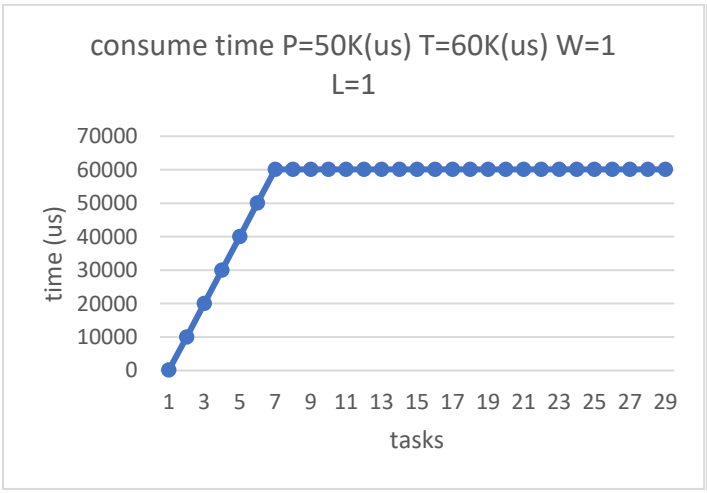
consume time = μετράει τον χρόνο από το σημείο που ο producer έβαλε το entry στην ουρά μέχρι ο consumer να καλέσει την queuePop

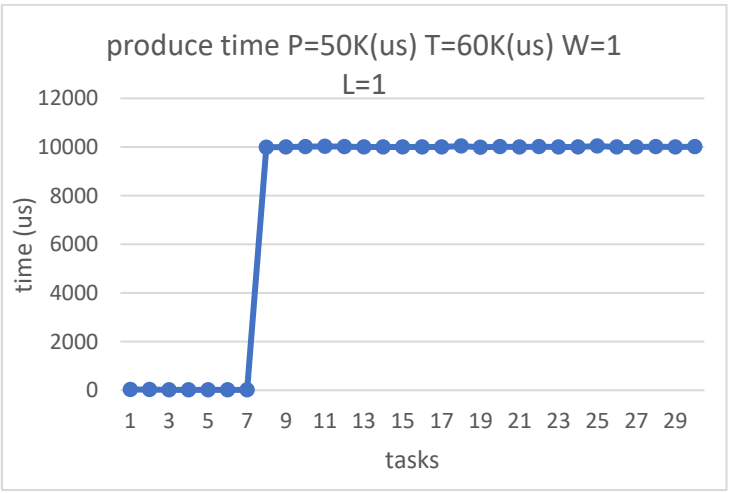




Εδώ παρατηρούμε έγκυρη και πραγματικού χρόνου εκτέλεση με

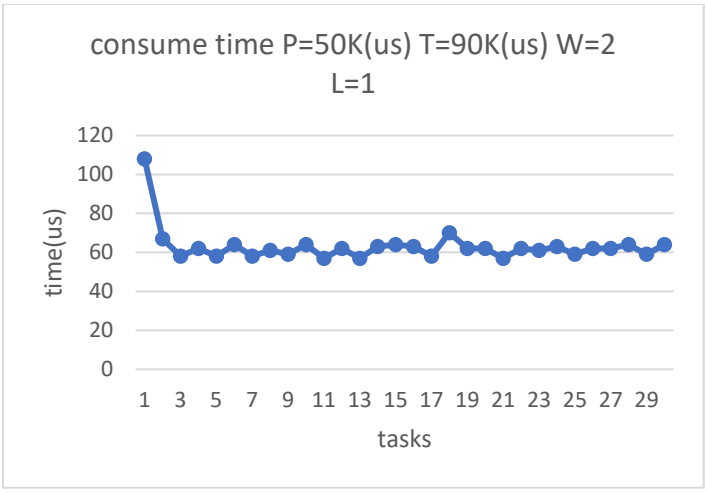
$$\frac{1}{5} * \frac{TaskIn}{Time} < \frac{1}{4} * \frac{TasksOut}{Time}$$

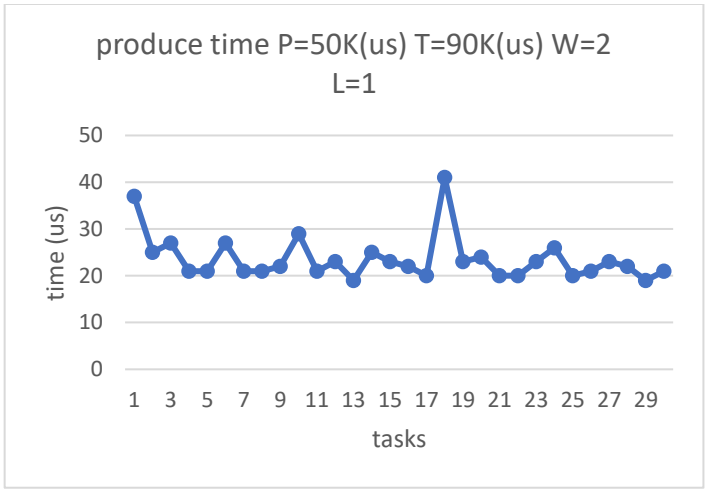




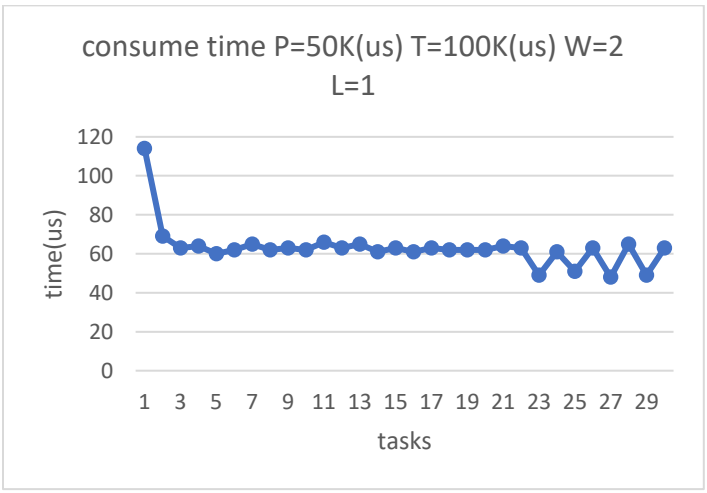
Ενώ στο δεύτερο πείραμα δεν μπορούμε να εγγυηθούμε έγγυρη εκτέλεση

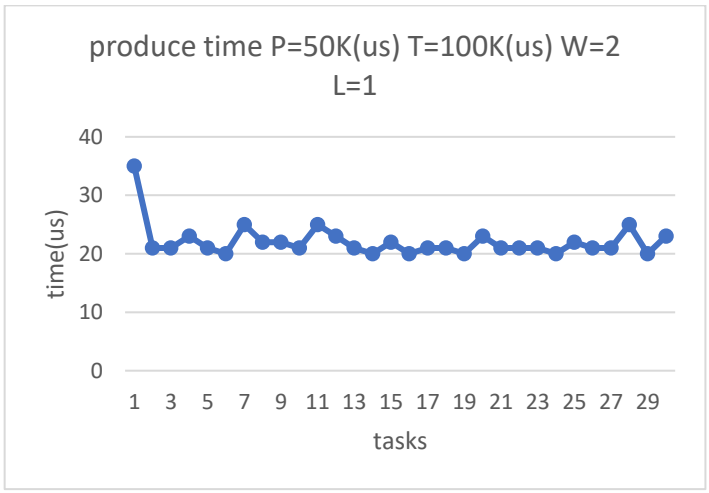
$$\frac{1}{5} * \frac{TaskIn}{Time} > \frac{1}{6} * \frac{TasksOut}{Time}$$



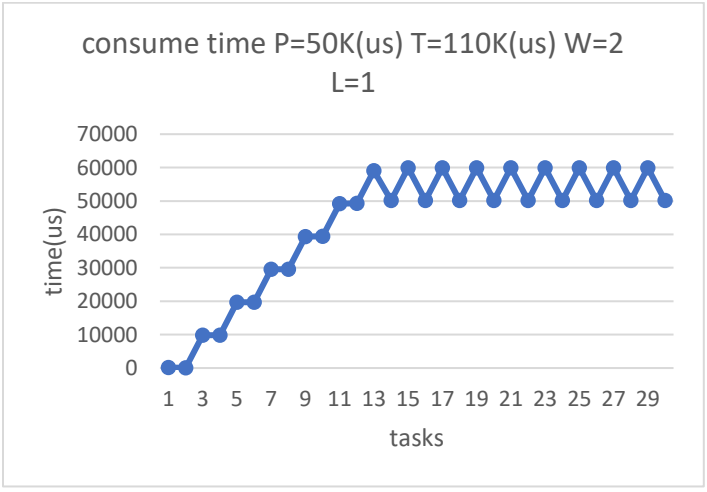


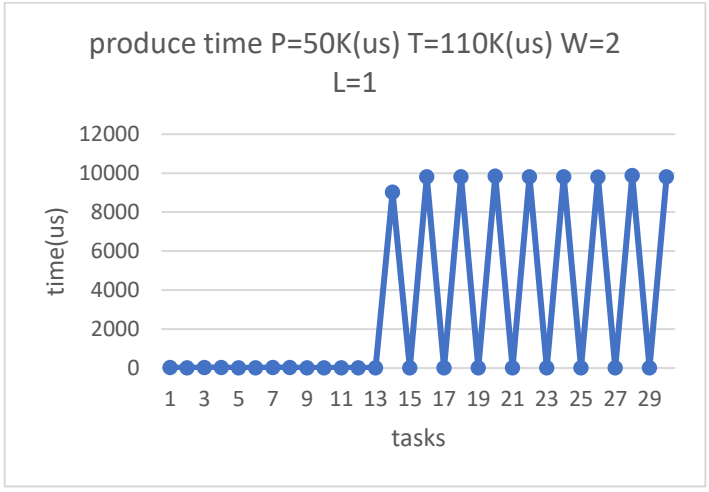
$$\frac{1}{5} * \frac{TaskIn}{Time} \leq \frac{2}{9} * \frac{TasksOut}{Time}$$





$$\frac{1}{5} * \frac{TaskIn}{Time} \leq \frac{1}{5} * \frac{TasksOut}{Time}$$



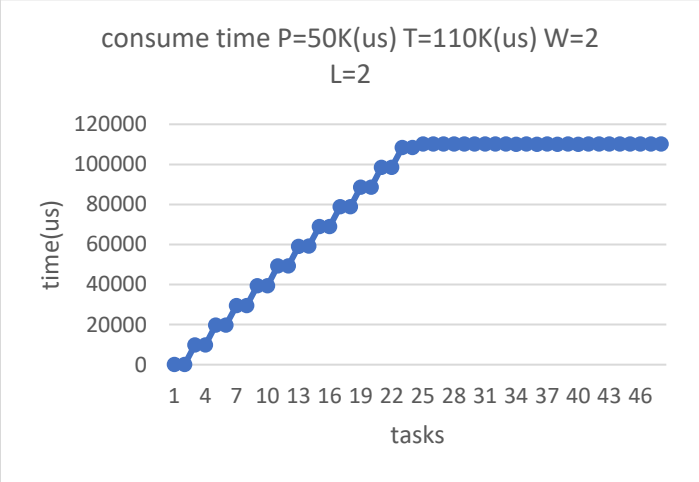


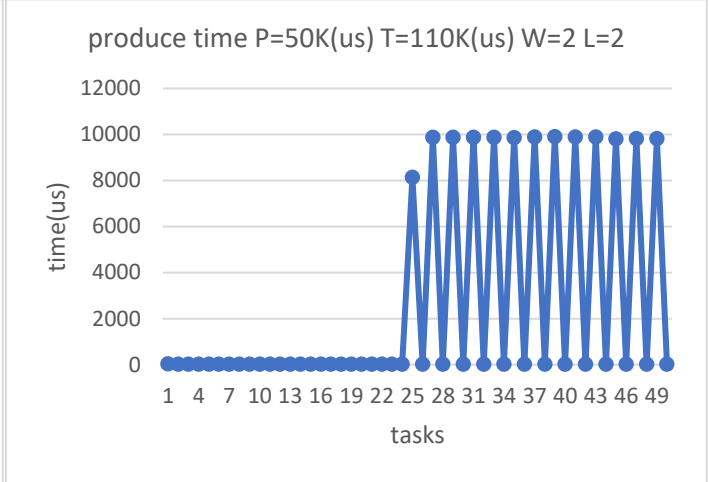
$$\frac{1}{5} * \frac{TaskIn}{Time} > \frac{2}{11} * \frac{TasksOut}{Time}$$

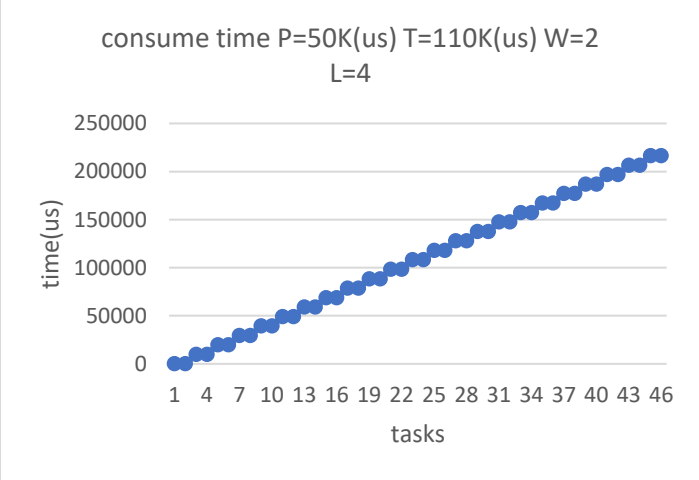
Σε αυτά τα πειράματα βλέπουμε όπως περιμέναμε ότι τα $\frac{TasksOut}{Time}$

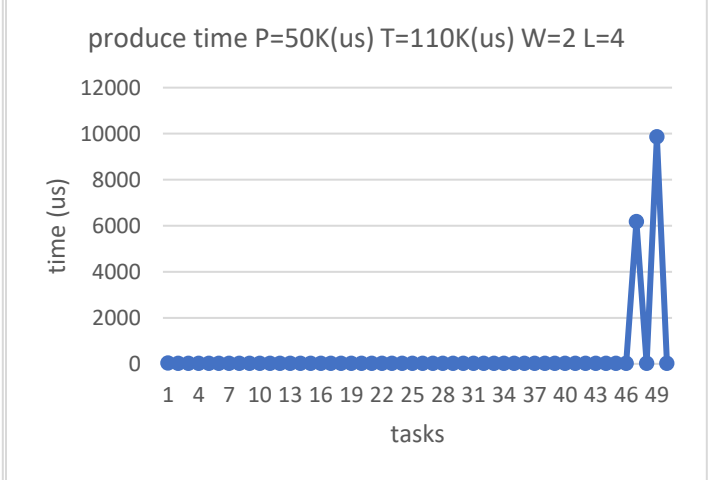
επηρεάζονται και από τον αριθμό των consumer εργατών

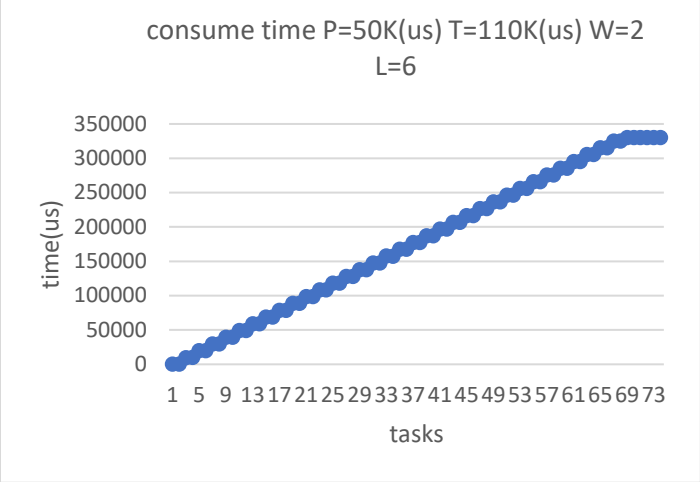
Συνεχίζοντας θα αυξήσουμε και το μέγεθος της ουράς για να εξετάσουμε την συνεισφορά του για την επίτευξης της έγκυρης εκτέλεσης

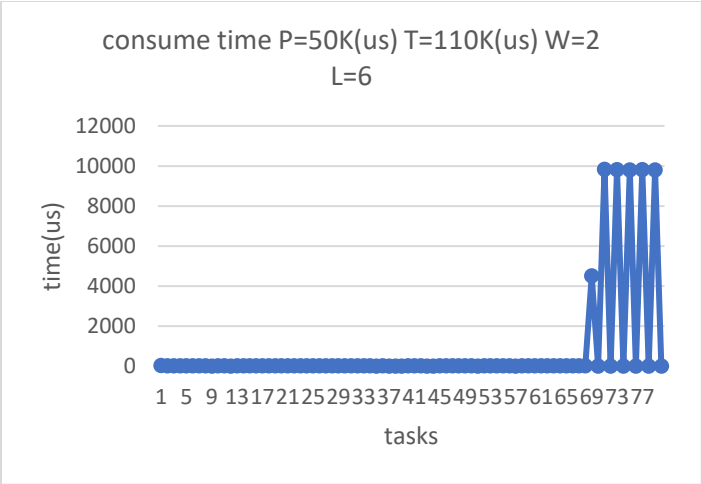












Τα παραπάνω πειράματα δείχνουν ένα σύστημα που όσο εκτελεί tasks τόσο πιο ασταθές γίνεται

γεγονός που επαληθεύει την ανισότητα για την έγκυρη εκτέλεση

υπό αυτό το πλαίσιο παρατηρούμε ότι το μήκος τη ουράς L δεν μπορεί να μας εγγυηθεί την έγκυρη εκτέλεση

Βέβαια, βλέπουμε ότι το *produce time* φτάνει πιο αργά στην αστάθεια

λόγω του μεγαλύτερου μήκους της ουράς