Ενσωματωμένα Συστήματα Πραγματικού Χρόνου
Εργασία 1η

Όνοματεπώνυμο: Αμοιρίδης Βασίλειος
ΑΕΜ: 8772

1) Προσθήκες στο υπάρχον πρόγραμμα

1.1) struct workFuncArgs;

Από τη στιγμή που ήταν απαραίτητο να προσμετράται ο χρόνος παραμονής των στοιχείων στη λίστα δημιουργήθηκε η δομή που φαίνεται παρακάτω. Ο `void *args;` της `struct workFunction` δείχνει σε στοιχεία της συγκεκριμένης δομής. Η δομή περιέχει 2 ορίσματα, το `void *func_args;`, το οποίο είναι στην ουσία το όρισμα που "περνάμε" στη συνάρτηση που εκτελούμε, και το `struct timeval tv;` το οποίο περιέχει χρονικά δεδομένα για τη συνάρτηση που εκτελούμε.

```
typedef struct workFunArgs {
      void *func_args;
      struct timeval tv;
} workFunArgs;
```

1.2) struct workFunction *funcArray;

Για τις συναρτήσεις που θα εκτελούνται από τους consumers δημιουργήθηκε ένας global pointer που δείχνει σε έναν πίνακα με τις προς εκτέλεση συναρτήσεις. Δημιουργήθηκαν 4 συνολικά συναρτήσεις οι οποίες αυτό που κάνουν είναι ότι οι 3 πρώτες υπολογίζουν τα `sin`(), `cos`(), `tan`() των γωνιών που τους δίνονται ενώ η τελευταία εκτυπώνει ένα μήνυμα στην οθόνη μέσω της `printf`(). Το αποτέλσμα των 3 πρώτων συναρτήσεων **επιστρέφει** στον consumer σε περίπτωση που χρειαζόταν η τιμή του.

2) Δοκιμές και αποτελέσματα

Το σύστημα που χρησιμοποιήθηκε για την εργασία είναι λάπτοπ Lenovo Y50-70 4K UHD με επεξεργαστή i7-4720HQ @ 2.6 GHz, **4C/8T**, 6MB Cache και 47W TDP. Έγιναν δοκιμές με διαφορετικούς συνδυασμούς και συναρτήσεις `work`() για να μελετηθεί η συμπεριφορά του συστήματος. Οι παρατηρήσεις που οδήγησαν στις διαφορετικές δοκιμές με `work`() είναι οι εξής:

- Η χρήση του πυρήνα από τον producer είναι **πολύ μικρότερη** σε σχέση με αυτή του consumer διότι ο producer απλά τοποθετεί μια συνάρητηση στην ουρά ενώ ο consumer πρέπει να τη «σβήσει» από την ουρά και να την εκτελέσει.
- Αν το πρόγραμμα έτρεχε με συναρτήσεις οι οποίες ήταν **όλες** *CPU-bound*, τότε **θεωρητικά δε θα υπήρχε βελτίωση** για μεγαλύτερο αριθμό thread από αυτόν που μπορεί να υποστηρίξει ο υπολογιστής μου. Αυτό γιατί ήδη θα χρησιμοποιούσα το 100% των πόρων του συστήματος και το ΛΣ δε θα είχε λόγο να σταματήσει τη λειτουργία ενός thread με σκοπό να ξεκινήσει άλλο.
- Αν το πρόγραμμα έτρεχε με συναρτήσεις εκ των οποίων κάποιες έπρεπε να περιμένουν ανταπόκριση από κάποια I/O συσκευή, τότε **θεωρητικά θα μπορούσε να υπάρχει βελτίωση** και για μεγαλύτερο αριθμό threads από αυτό που υποστηρίζει το σύστημα μας, διότι κατά τη διάρκεια αναμονής του I/O, το ΛΣ θα μπορούσε να «δώσει» τον πυρήνα σε ένα άλλο thread.

Με βάση τους παραπάνω ισχυρισμούς έγιναν 2 «πειράματα» πριν γίνει η μελέτη του πραγματικού συστήματος. Στο 1ο πείραμα και οι 4 συναρτήσεις προς εκτέλεση είναι *CPU-bound* με ένα «ψεύτικο» φορτίο μιας άδειας `for`() με πολλές επαναλήψεις. Στο 2ο πείραμα η 1 εκ των 4 συναρτήσεων αντί να είναι *CPU-bound,* την κάνουμε να είναι *I/O-bound,* δηλαδή να «περιμένει» κάποια I/O συσκευή. Αυτό το πετυχαίνουμε με τη χρήση της συνάρτησης `usleep`(), η οποία στην ουσία «κοιμίζει» το thread μέχρι το πέρας του χρονικού διαστήματος. Τα αποτελέσματα για διαφορετικούς αριθμούς consumer φαίνονται στον παρακάτω πίνακα:

CPU-bound	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**10**	**12**	**16**
Total time (us)	1853	932	627	477	425	395	354	345	347	344	345
Mean time (us)	1.865	0.941	0.632	0.485	0.443	0.402	0.377	0.355	0.355	0.355	0.355
I/O-bound	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**10**	**12**	**16**
Total time (us)	7832	3941	1838	1234	883	704	563	411	292	285	277
Mean time (us)	7.866	3.946	1.859	1.254	0.903	0.726	0.595	0.459	0.385	0.352	0.327

Τα παραπάνω αποτελέσματα επιβεβαιώνουν τις αρχικές υποθέσεις μας. Αρχικά όταν έχουμε κυρίως *CPU-bound* συναρτήσεις όπου όλες χρειάζονται τους πόρους του επεξεργαστή, η αύξηση του αριθμού των consumer πάνω από το μέγιστο δυνατό της CPU, δεν επιφέρει αύξηση στην απόδοση. Για τις *I/O-bound* συναρτήσεις, πάλι οι ισχυρισμοί μας αποδείχθηκαν σωστοί διότι είναι φανερή η αύξηση της απόδοσης ακόμα και για αριθμό μεγαλύτερο από αυτόν των 8 threads, διότι όσο ένα thread είναι σε sleep mode παραχωρεί τον επεξεργαστή σε κάποιο άλλο μέχρι να «ξυπνήσει». Όσο μεγαλύτερη είναι η καθυστέρηση, τόσο περισσότερο βλέπουμε βελτίωση για πολλά περισσότερα threads.

Με βάση λοιπόν τα παραπάνω, το σύστημα μας το οποίο αποτελείται από 4 συναρτήσεις που εκτελούν τις εντολές `sin`(), `cos`(), `tan`() (και επιστρέφουν στον consumer το αποτέλεσμα) και `printf`() έχουν τις εξής στατιστικές.

Consumers	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**10**
Mean	17.83	9.83	7.93	2.93	1.97	1.55	**1.47**	1.49	1.48
Std	13.44	9.22	8.96	6.65	4.9	2.6	**2.61**	6.8	3.55
Median	13	7	5	2	1	1	**1**	1	1
Max	169	88	106	106	130	153	**96**	97	109
Min	4	1	1	1	1	1	**1**	1	1

Κοιτάζοντας τα αποτελέσματα του προγράμματος μας παρατηρούμε κάποια αρκετά ενδιαφέροντα χαρακτηριστικά. Είναι προφανές ότι όσο αυξάνεται ο αριθμός των consumers η απόδοση του προγράμματος βελτιώνεται μέχρι ένα σημείο. Παρατηρείται ότι μετά τους 7 consumers (δοκιμάστηκε για 8,10,12,16,24 και 32) υπάρχει μια **αύξηση στην τυπική απόκλιση** ενώ ταυτόχρονα παρατηρείται και μια **μικρή αύξηση του συνολικού χρόνου εκτέλεσης** (δεν φαίνεται στον πίνακα) και της ίδιας της **μέσης τιμής**. Κοιτάζοντας λοιπόν και τα υπόλοιπα δεδομένα παρατηρήθηκε ότι από τους 7 consumers και μετά, η **ουρά αδειάζει** πάρα πολύ συχνά με αποτέλεσμα οι consumers να σταματάνε πολλές φορές τη λειτουργία τους μέχρι να προστεθεί κάποιο κομμάτι. Για να εκκινηθούν ξανά πρέπει να περιμένουν, μέχρι να λάβουν το αντίστοιχο σήμα από τον producer.
Ταυτόχρονα από τη στιγμή που ξεπερνάμε τα 7 συνολικά threads, δημιουργείται το εξής πρόβλημα:
Το ΛΣ πρέπει και αυτό να εκτελέσει τις δικιές του διεργασίες στον υπολογιστή, οπότε χρειάζεται να σταμάταει κάποιο από τα δικά μας thread όσο δουλεύει αυτό. Έτσι λοιπόν επειδή πλέον οι συναρτήσεις μας εκτελούνται πολύ πιο γρήγορα από τις προηγούμενες φορές που βάλαμε τεχνητό φορτίο και τεχνητή καθυστέρηση, η ταχύτητα εκτέλεσης των διεργασιών του ΛΣ είναι **συγκρίσιμη** με την εκτέλεση των συναρτήσεων μας πράγμα που επηρεάζει πολύ περισσότερο τα δεδομένα σε σχέση με πριν.

Οι παραπάνω στατιστικές έχει νόημα να παρατηρηθούν ξεχωριστά από τη στιγμή που ξεκινάει το πρόγραμμα μέχρι τη **στιγμή της σταθεροποίησης**. Κάθε φορά που ξεκινάει η εκτέλεση του προγράμματος, χρειάζεται να περάσει μερικός χρόνος μέχρι οι τιμές του χρόνου αναμονής στην ουρά να σταθεροποιηθούν. Από τη στιγμή που σταθεροποιούνται οι τιμές και μετά δεν έχει τόσο νόημα η διάμεσος, η μέγιστη και η ελάχιστη τιμή, παρά μόνο εαν η τυπική απόκλιση είναι τόσο μεγάλη που εν τέλει δεν έχω πραγματική σταθεροποίηση των τιμών. Ας δούμε λοιπόν αυτές τις στατιστικές από την εκκίνηση μέχρι και τη στιγμή (αριθμός δείγματος από τις 10000) που φτάνουμε περίπου στο 5% της μέγιστης τιμής του χρόνου αναμονής:

Consumers	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**10**
Mean	36.73	19.34	14.17	8.95	4.45	3.87	3.085	3.94	5.68
Std	14.54	15.36	13.41	25.55	11.20	8.68	6.78	9.32	10.46
Median	29	16	11	3	2	5	2	2	4
Stabilize	2425	1575	2080	410	1250	360	200	850	350

Οι τιμές είναι εντελώς διαφορετικές σε σχέση με όλο το εύρος των μετρήσεων. Αυτό που αξίζει να παρατηρηθεί είναι ότι στους 7 consumers έχουμε τη γρηγορότερη σταθεροποίηση (μόλις 200 `work`() από την εκκίνηση) και ταυτόχρονα την χαμηλότερη μέση τιμή εκκίνησης και ταυτόχρονα πολύ χαμηλό μέγιστο. Για το συγκεκριμένο σύστημα με τις συγκεκριμένες `work`(), ο συνδυασμός **(p,q)=(1,7)** είναι ο βέλτιστος. LINK