Σχολή
Ηλεκτρολόγων Μηχανικών
και
Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα

3η σειρά ασκήσεων

oslabb12
Γκούμας Βασίλης — 03113031
Ζαρίφης Νικόλαος — 03112178

Άσκηση 1

Ο κώδικας της άσκησης βρίσκεται στο αρχείο simplesync.c

Στόχος της άσκησης είναι ο συγχρονισμός δύο νημάτων εκτέλεσης, ένα που αυξάνει τη τιμή μιας μοιραζόμενης μεταβλητής και ένα που την μειώνει, ώστε μετά το πέρας της εκτέλεσης των νημάτων να πάρουμε τα σωστά αποτελέσματα.
Αυτό ζητείται να υλοποιηθεί με δύο τρόπους, με atomic operations και mutex locks.

Ερωτήματα 1-2

Μελετώντας το Makefile που μας δίνεται, βλέπουμε πως απο το ίδιο αρχείο κώδικα (simplesync.c) παράγονται δύο εκτελέσιμα, τα simplesync-atomic και simplesync-mutex. Αυτό πετυχαίνεται μέσω των #define macros που υπάρχουν στον κώδικα. Συγκεκριμένα κατά το compilation, στη μία παραλλαγή του εκτελέσιμου περνάμε την παράμετρο DSYNC_MUTEX, η οποία συμπεριλαμβάνει τα κομμάτια κώδικα που βρίσκονται μέσα στα τμήματα #ifdefine SYNC_MUTEX.

Όσον αφορά το χρόνο εκτέλεσης των δύο προγραμμάτων έχουμε τα παρακάτω "benchmark":

- simplesync:
  real 0m0.044s
  user 0m0.036s
  sys 0m0.000s

- simplesync-atomic:
  real 0m0.140s
  user 0m0.132s
  sys 0m0.000s

- simplesync-mutex:
  real 0m0.767s
  user 0m0.764s
  sys 0m0.000s

Βλέπουμε λοιπόν πως ταχύτερο είναι το εκτελέσιμο χωρίς συγχρονισμό, ακολουθεί το simplesync-atomic και τέλος το simplesync-mutex. Η παραπάνω διαφορά στον χρόνο εκτέλεσης, οφείλεται στο ότι το simplesync-atomic υλοποιείται με atomic operations, οι οποίες μεταφράζονται σε **μία** (κατα προσέγγιση)εντολή assembly. Απο την άλλη, η έννοια των mutex αποτελεί μια πιο high level ιδεά και ενσωματώνει την έννοια των atomic operations για να υλοποιηθεί. Τέλος, το ασυγχρόνιστο εκτελέσιμο είναι το γρηγορότερο καθώς δε χρειάζεται να ασκηθεί κάποιος περιορισμός στην εκτέλεση των δύο threads .

Παρακάτω βλέπουμε το αποτέλεσμα της εκτέλεσης των τριών παραπάνω προγραμμάτων.

```
oslabb12@leykada:~/ex3$ ./simplenotsync
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = -4162065.
oslabb12@leykada:~/ex3$ ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
oslabb12@leykada:~/ex3$ ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Ερώτημα 3

Παρακάτω βλέπουμε τις εντολές assembly στις οποίες μεταγλωττίζεται η sync_fetch_and_sub. Για την αντίστοιχη add απλά λείπουν οι εντολές που εκτελούν την αντιστροφή στην αριθμητική συμπληρώματος ως προς δύο. Έχουμε:

```
        .loc 1 78 0
        movl    -4(%ebp), %eax
        movl    $1, %edx
        negl    %edx
        lock addl       %edx, (%eax)
        .loc 1 73 0
        addl    $1, -8(%ebp)
```

Μετά τον κώδικα διαχείρισης της στοίβας (δημιουργία του stack frame) για την κλήση της συνάρτησης, έχουμε την κύρια λειτουργία της atomic λειτουργίας, που υλοποιείται με την εντολή lock addl.

Ερώτημα 4

Ο αντίστοιχος κώδικας assembly που παράγεται για την pthread_mutex_lock() φαίνεται παρακάτω:

```
.L3:
        .loc 1 54 0
        movl    $mutex, (%esp)
        call    pthread_mutex_lock
        .loc 1 57 0
        movl    -4(%ebp), %eax
        movl    (%eax), %eax
        leal    1(%eax), %edx
        movl    -4(%ebp), %eax
        movl    %edx, (%eax)
        .loc 1 59 0
        movl    $mutex, (%esp)
        call    pthread_mutex_unlock
        .loc 1 45 0
        addl    $1, -8(%ebp)
```

Άσκηση 2

Ο κώδικας της άσκησης βρίσκεται στο αρχείο mande.c

Ερώτημα 1

Στο πρόγραμμά μας χρειάστηκαν συνολικά N σημαφόροι, ένας για κάθε thread. Συγκεκριμένα για κάθε νήμα, ο σημαφόρος του ενεργοποιείται με τη λήξη της εργασίας του $N-1$ νήματος και με την ολοκλήρωση του σηματοδοτεί την έναρξη του $N+1$ νήματος. Αναλυτικότερα το σχήμα συγχρονισμού μας λειτουργεί σαν κυκλικός δακτύλιος, με το κάθε νήμα να παίρνει την άδεια προς εκτέλεση απο το προηγούμενό του και την άδεια 0.

Ερώτημα 2

Σε διπύρηνο μηχάνημα χρονομετρήσαμε την εκτέλεση των δύο προγραμμάτων και έχουμε:

- σειριακή εκδοχή
  real 0m0.778s
  user 0m0.776s
  sys 0m0.000s

- παράλληλη εκδοχή (2 νήματα)
  real 0m0.395s
  user 0m0.772s
  sys 0m0.008s

Ερώτημα 3

Όπως βλέπουμε απο τις παραπάνω μετρήσεις το παράλληλο πρόγραμμα εμφανίζει επιτάχυνση. Αυτο είναι λογικό, καθώς η φάση υπολογισμού γίνεται παράλληλα και ο συγχρονισμός επιστρατεύεται ώστε να γίνει με τη κατάλληλη σειρά το τύπωμα. Σε αντίθετη περίπτωση δε θα είχαμε επιτάχυνση, καθώς το κάθε νήμα θα περίμενε το προηγούμενο για να υπολογίστει, με αποτέλεσμα το πρόγραμμα να εκφυλίζεται στο αντίστοιχο σειριακό, με πιθανώς χειρότερη επίδοση λόγω context switching.

Ερώτημα 4

Με το πάτημα του CTRL-C το πρόγραμμα μας τερματίζει, καθώς του στέλνται σήμα SIGINT. Ως αποτέλεσμα το χρώμα των γραμμών του τερματικού παραμένει σε αυτό που είχε επιλεχθεί για τύπωμα ακριβώς πριν τον τερματισμό του προγράμματος.
Μια λύση είναι να υλοποιήσουμε τον δικό μας signal handler που με το πάτημα του CTRL-C να επαναφέρει το default χρώμα και στη συνέχεια να τερματίζει.

Άσκηση 3

Ο κώδικας της άσκησης βρίσκεται στο αρχείο kgarten.c

Το αποτέλεσμα της εκτέλεσης για ενδεικτικές τιμές που δοκιμάζουν το σχήμα συγχρονισμού βρίσκονται στον φάκελο out3.

Στην άσκηση αυτή μας ζητείται η υλοποίηση ενός σχήματος συγχρονισμού για ένα νηπιαγωγείο. Ο χρυσός κανόνας είναι ότι αν t, c, r είναι το πλήθος των δασκάλων, των παιδιών και το ratio που δίνεται, πρέπει κάθε στιγμή μέσα στο νηπιαγωγείο να ισχύει η σχέση:

$$t \cdot r \geq c$$

Για την παραπάνω υλοποίηση μεταβάλλαμε μόνο τον κώδικα των συναρτήσεων *_enter και *_exit.

Στη συνέχεια περιγράφεται το σχήμα συγρονισμού που χρησιμοποιήθηκε. Οι κρίσιμες λειτουργίες είναι οι child_enter και η teacher_exit καθώς αυτές απαιτούν τον έλεγχο της αντίστοιχης συνθήκης πριν να είναι δυνατή η έξοδος/είσοδος.

Αντιθέτως, οι συναρτήσεις child_exit και teacher_enter δεν έχουν κάποιο περιορισμό καθώς κάποιο παιδί μπορεί να φύγει οποιαδήποτε στιγμή ή κάποιος δάσκαλος να έρθει, χωρίς να επηρεάζουν την αλήθεια της συνθήκης. Επομένως μετά την αυξομείωση των κοινών resources οι παραπάνω συναρτήσεις οφείλουν να σηματοδοτήσουν τις child_enter, teacher_exit, σε περίπτωση που ισχύει η αντίστοιχη συνθήκη.

Το παραπάνω γεγονός το αναπαριστούμε με τις conditional variables με όνομα teacher_out, child_in που σηματοδοτούνται όταν ισχύει η κατάλληλη συνθήκη. Η σηματοδότηση γίνεται απο όλες τις συναρτήσεις, ώστε να εξαλείψουμε στο έπακρο τυχόν race conditions.

Επίσης προσθέσαμε τη μεταβλητή count που συμβολίζει το πλήθος των παιδιών που μπορούν να μπουν ακόμα και απλουστεύει τον έλεγχο των συνθηκών και τέλος η πρόσβαση σε όλους τους κοινούς πόρους έγινε μέσω κλειδώματος με mutex.

Εξίσου σημαντικό είναι ότι μετά απο οποιδήποτε ενέργεια ελέγχεται ξανά η συνθήκη ώστε να σηματοδοτήσει την αντίστοιχη είσοδο/έξοδο.

Το σχήμα συγχρονισμού μας είναι ορθό και εξασφαλίζει τη πρόοδο (progress). Ορθο διότι πριν απο κάθε είσοδο παιδιού/έξοδο δασκάλου, εξασφαλίζει ότι η συνθήκη είναι είναι αληθής.Επίσης εξασφαλίζει πρόοδο γιατί κάθε ενέργεια ¨προωθεί¨ κάθε επόμενη διαθέσιμη ενέργεια, δηλαδή μετά απο κάθε έισοδο/έξοδο, ελέγχεται αν μπορεί να βγεί κάποιος δάσκαλος ή να μπεί κάποιο παιδί.

Έτσι αποτρέπονται τα **τεχνητά** race conditions, δηλαδή απο παράβλεψη του σχήματος υλοποίησης και επομένως εμφανίζονται μόνο τα φυσικά, που εξαρτώνται απο τη σειρά με την οποία θα έρθουν τα νήματα. Το σχήμα μας είναι best effort με την έννοια ότι δεν κάνει διακρίσεις στα νήματα που έρχονται και προσπαθεί να τα ικανοποιήσει όλα χωρίς να εισάγει νέα κολλήματα και είναι ευάλωτο αποκλειστικά στη σειρά και το πλήθος των νημάτων που καταφθάνουν.

Ερώτημα 1

Στο σχήμα συγχρονισμού που περιγράψαμε και υλοποιήσαμε, όταν υπάρχουν παιδιά και δάσκαλοι που περιμένουν να βγουν/μπουν, τότε εξυπηρετείται τυχαία κάποιο αίτημα, ανεξαρτήτου του χρόνου αναμονής. Για παράδειγμα ένα παιδί που μόλις έφτασε μπορεί να μπεί αμέσως, ενώ ένας δάσκαλος να περιμένει αρκετή ώρα για να βγεί. Αυτο εξαρτάται απο τη σειρά εκτέλεσης των νημάτων.

Ερώτημα 2

Το παραπάνω race condition εμφανίζεται και στον έλεγχο ορθότητας του kgarten.c . Επομένως το πρόγραμμά μας είναι έρμαιο αποκλειστικά της τυχαιότητας καθώς όπως είδαμε η συνθήκη δε παραβιάζεται ποτέ και η υλοποιήση δε μπλοκάρει καμία είσοδο/έξοδο του νηπιαγωγείου.

Προαιρετικές Άσκήσεις

Ερώτημα 1

Ένας παρακολουθητής (Monitor) που υλοποιεί τη λειτουργία σημαφόρου θα είχε δύο μεθόδους, η πρώτη είναι η wait() που παγώνει την εκτέλεση της διεργασίας, αναμένοντας την signal().Η δεύτερη είναι η signal(), που ξυπνάει κάποια διεργασία απο το σωρό νημάτων (thread pool)

Ερώτημα 2

Το δοσμένο πρόγραμμα παίρνει προαιρετικά ως παράμετρο έναν αριθμό απο τη γραμμή εντολών και τυπώνει τόσους τυχαίους αριθμούς (σε περίπτωση που δε δωθεί όρισμα τυπώνει 10). Τρέχοντας όμως το πρόγραμμα παρατηρούμε ότι δεν παίρνουμε τα αναμενόμενα αποτελέσματα, αλλα τους ίδιους αριθμούς.

Αυτό συμβαίνει, γιατί οι spawned processes που αναλαμβάνουν το τύπωμα έχουν κληρονομήσει το ίδιο seed, το οποίο χρησιμοποιεί η rand για τη παραγωγή ¨τυχαίων¨ αριθμών, και ως αποτέλεσμα όλες οι κλήσεις στην rand θα επιστρέψουν την ίδια τιμή (ακόμα αν είχαμε και παραπάνω απο μία τιμές για κάθε διεργασία).

Αυτό οφείλεται στο ότι η εντολή srand(time(NULL)) που χρησιμοποιείται ως αρχικοποίηση του seed, η τιμή του οποίου εξαρτάται απο την τρέχουσα

ώρα, καλείται μια φορά στην αρχή του προγράμματος, αντί σε κάθε διεργασία ξεχωριστά, έτσι όλες οι γεννήτριες ψευδοτυχαίων αριθμών κληρονομούν το ίδιο seed και άρα θα τυπώνουν ίδιους αριθμούς.

Ερώτημα 3

Μια υλοποίηση που θα έδινε διαφορετική πιθανότητα επιλογής σε παιδιά και δασκάλους, θα ήταν η ύπαρξη δύο ουρών, μία για τους δασκάλους και μία για τα παιδιά.

Επομένως κάθε φορά που επιλέγεται νήμα προς εκτέλεση θα επιλέγεται με μεγαλύτερη πιθανότητα η ουρά με τα νήματα παιδιών.