**Λειτουργικά Συστήματα**

**2η Φροντηστηριακή Άσκηση**

**Καλλιοντζής Θεόδωρος 1083848**

Στην δεύτερη φροντηστηριακή άσκηση, δημιουργώ τρία νήματα αυτά της εκφώνησης, τα οποία για να συγχρονίσουν την εκτέλεσή τους χρησιμοποιούν σημαφόρους.

**Θεωρτητικό Υπόβαθρο:**

Η βασική αρχή στηρίζεται στη χρήση σημάτων (signals) και ο συγχρονισμός της εκτέλεσης γίνεται μέσω της αποστολής και αναμονής λήψεώς τους.

Σε γλώσσα C η σημαφόροι:

* Περιλαμβάνονται στην βιβλιοθήκη <semaphore.h>, για Linux λειτουργικό σύστημα.
* Αρχικοποιούνται με την sem\_init(\_, \_, \_) περισσότερα παρακάτω
* Για την αποστολή και καταμέτρηση των σημάτων συνδυάζονται με τις θεμελιώδεις λειτουργίες semSignal/ sem\_post και την semWait/ sem\_post

Όσο αναφορά την int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

Το πρώτο όρισμα είναι ένας δείκτης σε ένα αντικείμενο τύπου sem\_t, το οποίο αντιπροσοπέυει το semaphore που πρόκειται να αριχικοποιθεί

Το δεύτερο όρισμα εναλλάσσεται μεταξύ των τιμών 0 και 1(ή μη μηδενική τιμή) και δηλώνει αν ο semaphore είναι κοινόχρηστος μεταξύ νημάτων της ίδιας διεργασίας ή μεταξύ πολλαπλών διεργασιών αντίστοιχα. Στην συγκεκριμένη άσκηση υλοποιούμε 3 threads μέσα στην ίδια διεργασία άρα θα έχει την τιμή μηδέν

Και το τρίτο όρισμα είναι το value. Δηλώνει τον αριθμό νημάτων ή διεργασιών που μπορούν να αποκτήσουν ταυτόχρονη πρόσβαση στον κοινόχρηστο χώρο.

Οι συναρτήσεις sem\_wait και sem\_post/semSignal θα έλεγε κανείς ότι έχουν αντίθετη λειτουργία.

Η sem\_wait ελέγχει την τιμή του semaphore αν είναι μεγαλύτερη του μηδενός, την μειώνει κατά 1 και το νήμα συνεχίζεται κανονικά.

Αν η τιμή είναι μηδενική, το νήμα μπαίνει σε αναμονή.

Απ’την άλλη η sem\_post αυξάνει την τιμή του semaphore κατά 1. Αν κάποιο νήμα/διεργασία είναι σε κατάσταση αναμονής επειδή η τιμή του semaphore ήταν 0 , το παλαιότερο νήμα αφυπνίζεται και προχωρά στο critical path.

Με βάση το παραπάνω θεωρητικό υπόβαθρο και το πώς υλοποιούμε τα threads στην C προχωράω στην επίλυση της άσκησης.

Τ3:

c1=30

c2=31

z=a2+b1

Τ2:

b1=20

b2=21

w=b2+c2

x=z-y+w

Τ1:

a1=10

a2=11

y=a1+c1

print(X)

Παρατηρούμε εύκολα οτι το επιθυμητό αποτέλεσμα για το xείναι:

Y=a1+c1=40

W=b2+c2=21+31=52 X = 43.

Z=a2+b1=11+20=31

X=z-y+w=31-40+52=43

Προφανώς, όπως και στην προηγούμενη άσκηση οι βιβλιοθήκες που χρησιμοποιούμε (POSSIX) δεν είναι συμβατές με windows λογισμικό, οπότε μια γρήγορη λύση που βρήκα είναι να γράφω τον κώδικα σε vscode (σε windows) και να τον στέλνω στο VM που χρησιμοποιούμε το οποίο έχει Debian Linux (μέσω mail).

Κάνοντας σενάρια για το ποία μπορεί να είναι η ακολουθία με την οποία θα «τρέξει» τα threads ο scheduler, έφτασα στο συμπέρασμα ότι είναι αναγκαία η χρήση 3 σημαφόρων με μία από αυτήν να επαναχρησιμοποιείται δηλαδή να εφαρμόζεται στον συγχρονισμό διαφορετικών πόρων

Πριν δούμε όμως το πώς υλοποιούνται οι sem\_post και sem\_wait, ας δούμε πρώτα πως αρχικοποιούνται οι σημαφόροι και τα threads.

A screen shot of a computer program

Description automatically generated

Η sem\_init υλοποιέιται έτσι για τους λόγους που ανέφερα στην 1η σελίδα, η υλοποίηση της pthread\_create έχει αναλυθεί στην προηγούμενη άσκηση.

Παρατήρησα όμως το εξής: ΑΝ τοποθετηθούν σε σωστά σημεία οι sem\_post/sem\_wait, τα threads συγχρονίζονται σωστά ανεξαρτήτως της σειράς με την οποία γίνονται create. Δηλαδή οπώς θα φανεί παρακάτω θα τρέξω τον κώδικα σε σειρά δημιουργίας των threads 1 🡪 2 🡪 3

3🡪2🡪1, 2🡪3🡪1 και 2🡪1🡪3 .

A screenshot of a computer program

Description automatically generatedΑκολουθεί η λογική υλοποίησης των threads.

A screenshot of a computer

Description automatically generatedΗ λογική είναι ότι η πρώτη σημαφόρος sem1 θα χρησιμοποιεί την sem\_wait(&sem1) στο thread 1, αντίστοιχα η σημαφόρος sem2 στο thread 2 και η σημαφόρος sem3 στο thread 3.

A black screen with green text

Description automatically generated

A screen shot of a computer

Description automatically generated

Tώρα οι σημαφόροι sem1, sem2 και sem3 θα καλούν τις αντίστοιχες sem\_post τους στο thread απ’το οποίο θέλουν να πάρουν το «σήμα»

**Κατανοώντας το αυτό καλύτερα με 2 παραδείγματα:**

Η sem\_post(&sem2) στέλνει σήμα απ’το thread 1 απ’το οποίο καλείται στο thread 2 που γίνεται ο ελέγχος

Η sem\_post(&sem3) στέλνει σήμα απ΄το thread 1 απ΄το οποίο καλείται στο thread 3 που γίνεται ο έλεγχος.

Τώρα πιο συγκεκριμένα για τους υπολογισμούς των y,z,w και x που χρειάζεται συγχρονισμός, ο υπολογισμός του Z και του X είναι πιο σύνθετος γιατί απαιτούν να έχουν δηλωθεί δεδομένα από άλλα threads. Επομένως πριν τον υπολογισμό τους χρησιμοποιώ 2 sem\_wait εντολές.

A screen shot of a computer

Description automatically generated

Thread 3

A screen shot of a computer

Description automatically generated

Thread 2

Το ενδιαφέρον είναι ότι η σημαφόρος 1 επαναχρησιμοποιείται, δηλαδή καλείται 2 φορές οι sem\_wait(&sem2) στο ίδιο thread. Εξηγώντας το αυτό καλύτερα, όταν το thread 1 περάσει την πρώτη sem\_wait(&sem1) η τιμή της σημαφόρου είναι μηδέν, επομένως για να μην χρησιμοποιήσουμε παραπάνω σημαφόρους μπορούμε να την ξαναχρησιμοποιήσουμε.

A black screen with white text

Description automatically generated

Thread 1

Τρέχω τον κώδικα στο virtual machine σε linux. Αλλάζω την σειρά με την οποία δημιουργούνται τα threads για να καταλάβω αν ο κώδικας είναι συγχρονισμένος σωστά για όλες τις περιπτώσεις ανεξαρτήτως το πώς θα τρέξει ο scheduler τα threads.

A screen shot of a computer screen

Description automatically generated

2 -> 3 -> 1 Sequence

A screen shot of a computer screen

Description automatically generated

3-> 2 -> 1 Sequence

A screen shot of a computer screen

Description automatically generated

1->2->3 Sequence

A screen shot of a computer

Description automatically generated

1-> 3 -> 2 Sequence

Και οι άλλοι 2 συνδυασμοί ομοίως.