Λειτουργικά Συστήματα

2η Άσκηση

Τσάμπρας Κωνσταντίνος
up1083865

➔ Σκοπός της άσκησης είναι η επίλυση του παρακάτω προβλήματος ταυτοχρονισμού νημάτων με τον ελάχιστο αριθμό σημαφόρων.

```
t1: a1 = 10
    a2 = 11
    y = a1 + c1
    print(x)

t2: b1 = 20
    b2 = 21
    w = b2 + c2
    x = z - y + w

t3: c1 = 30
    c2 = 31
    z = a2 + b1
```

➔ 1η προσέγγιση: Χρήση ενός σημαφόρου για κάθε μεταβλητή που μοιράζονται πάνω από ένα νήματα (αρχείο sem7.c), απαιτούνται 7 σημαφόροι και το αποτέλεσμα είναι πάντα σωστό.
Η εξήγηση του κώδικα είναι απλή αφού απλά προστατεύουμε κάθε πράξη κάνοντας sem_wait τους σημαφόρους των μεταβλητών των οποίων οι τιμές θα χρησιμοποιηθούν στην πράξη αυτή, ενώ μετά από κάθε πράξη στέλνουμε σήμα ότι η αντίστοιχη μεταβλητή υπολογίστηκε και είναι πλέον διαθέσιμη για χρήση από τα άλλα νήματα. Πχ:

```c
void* thread1_func(void* arg) {
    a1 = 10;

    a2 = 11;
    sem_post(&sa2);

    sem_wait(&sc1);
    y = a1 + c1;
    sem_post(&sy);

    sem_wait(&sx);
    printf("x: %d\n", x);

    return NULL;
}
```

➔ 2η προσέγγιση: Εύκολα βλέπουμε ότι σε μερικές περιπτώσεις δεν χρειαζόμαστε δυο διαφορετικούς σημαφόρους, αλλά μπορούμε με δυο wait στον ίδιο σημαφόρο να εκτελέσουμε την ίδια λειτουργία χρησιμοποιώντας συνολικά 5 σημαφόρους (αρχείο sem5.c). Πχ:

```
b1 = 20;
sem_post(&sb1);
```

```
a2 = 11;
sem_post(&sa2);
```

```
sem_wait(&sb1);
sem_wait(&sa2);
z = a2 + b1;
```

Γίνεται:

```
b1 = 20;
sem_post(&sa2b1);
```

```
a2 = 11;
sem_post(&sa2b1);
```

```
sem_wait(&sa2b1);
sem_wait(&sa2b1);
z = a2 + b1;
```

Έτσι είμαστε πάλι σίγουροι ότι η εντολή ανάθεσης τιμής στην z θα εκτελεστεί μετά τα δύο waits, δηλαδή μετά τα δύο signals, δηλαδή μετά τις δύο αναθέσεις τιμών στα a2, b2, χωρίς να μας επηρεάζει η σειρά.

➔ 3η προσέγγιση: Γίνεται να μειώσουμε ακόμη περισσότερο τους σημαφόρους που χρησιμοποιούμε (χωρίς να περιορίσουμε περισσότερο από όσο είναι αναγκαίο την ροή εκτέλεσης των εντολών του προγράμματος) «επαναχρησιμοποιώντας» σημαφόρους αφού έχουν εκτελέσει την αρχική λειτουργία τους. Στο αρχείο sem4.c βλέπουμε μία υλοποίηση όπου ο σημαφόρος sc1 «επαναχρησιμοποιείται» στην θέση του σημαφόρου sx και μετονομάζεται σε sc1_x. Βέβαια πρέπει να ελέγξουμε ότι θα υπάρχει ορθή λειτουργία σε αυτήν την περίπτωση.
Στην προηγούμενη υλοποίηση η λειτουργία του sc1 είναι να εξασφαλίσει ότι η τιμή του c1 έχει οριστεί πριν την ανάθεση της y στο thread1:

```
// Thread functions
void* thread1_func(void* arg) {
    a1 = 10;

    a2 = 11;
    sem_post(&sa2b1);

    sem_wait(&sc1);
    y = a1 + c1;
    sem_post(&szy);

    sem_wait(&sx);
    printf("x: %d\n", x);

    return NULL;
}
```

Υπάρχει περίπτωση στην νέα υλοποίηση (sem4.c) να παραβιαστεί αυτό;

```
13  void* thread1_func(void* arg) {
14      a1 = 10;
15
16      a2 = 11;
17      sem_post(&sa2b1);
18
19      sem_wait(&sc1_x);// we are sure that sem_wait(&sc1_x) in this line will only be triggered by sem_post(&sc1_x) in line
20                       // 54 (as intended) because sem_post(&sc1_x) in line 45 requires szy (twice) which is posted two lines below
21      y = a1 + c1;
22      sem_post(&szy);
23
24      //sem_wait(&sx);
25      sem_wait(&sc1_x); // reusing sc1_x instead because it is not used anymore
26
27      printf("x: %d\n", x);
28
29      return NULL;
30  }
31
32  void* thread2_func(void* arg) {
33      b1 = 20;
34      sem_post(&sa2b1);
35
36      b2 = 21;
37
38      sem_wait(&sc2);
39      w = b2 + c2;
40
41      sem_wait(&szy);
42      sem_wait(&szy);
43      x = z - y + w;
44      //sem_post(&sx);
45      sem_post(&sc1_x);
46
47
48      return NULL;
49  }
50
51  void* thread3_func(void* arg) {
52      //sleep(3); //even with sleep the order is secured
53      c1 = 30;
54      sem_post(&sc1_x);
55
```

Βλέπουμε ότι για να γίνει αυτό πρέπει να σταλθεί sem_post(&sc1_x) από το thread2 πριν από το thread3. Κάτι που ξέρουμε ότι δεν θα συμβεί αφού για να φτάσει εκεί το thread2 (line 45) πρέπει να έχει οριστεί τιμή του y το οποίο γίνεται αργότερα στο thread1 (line 21).
Παράλληλα, είμαστε σίγουροι ότι ο έλεγχός (για το x) στην γραμμή 25 δεν θα παραβιαστεί αφού θα πρέπει να έχουν γίνει δύο sem_wait για το sc1_x άρα θα έχει υπολογιστεί η τιμή του x.

➔ 4η προσέγγιση: Επεκτείνοντας την παραπάνω λογική και στους σημαφόρους μπορούμε να λύσουμε το πρόβλημα με μόνο 3 σημαφόρους χωρίς να περιορίσουμε (ουσιαστικά) την ροή εκτέλεσης των εντολών (sem3.c).

Βλέπουμε ότι αν αντικαταστήσουμε τους δύο σημαφόρους (sc1, szy) με έναν κοινό (sc1_zy) και μεταφέρουμε την εκτέλεση του w = b2+c2 πιο «κάτω» έχουμε τρία διαδοχικά sem_wait στο thread2.
Έτσι είμαστε σίγουροι για την σωστή ανάθεση των μεταβλητών c2 (για το w = b2+c2) και y,z για το x=z-y+w.

Σημειώνεται ότι η «αναβολή» της ανάθεσης τιμής στο w δεν περιορίζει το πρόγραμμα αφού το w δεν χρησιμοποιείται από κάποιο άλλο νήμα και έτσι δεν υπάρχει κάποιο νήμα που περιμένει την ανάθεση αυτή για να συνεχίσει.

```c
void* thread1_func(void* arg) {
    a1 = 10;

    a2 = 11;
    sem_post(&sa2b1);

    sem_wait(&sc1_x);// we are sure that sem_wait(&s
                     // 54 (as intended) because sem_
    y = a1 + c1;
    sem_post(&sc2_zy);

    //sem_wait(&sx);
    sem_wait(&sc1_x); // reusing sc1_x instead becau

    printf("x: %d\n", x);

    return NULL;
}

void* thread2_func(void* arg) {
    b1 = 20;
    sem_post(&sa2b1);

    b2 = 21;

    sem_wait(&sc2_zy);
    sem_wait(&sc2_zy);
    sem_wait(&sc2_zy);

    w = b2 + c2;

    x = z - y + w;
    //sem_post(&sx);
    sem_post(&sc1_x);


    return NULL;
}

void* thread3_func(void* arg) {
    //sleep(3); //even with sleep the order is secur
    c1 = 30;
    sem_post(&sc1_x);

    c2 = 31;
    sem_post(&sc2_zy);

    sem_wait(&sa2b1);
    sem_wait(&sa2b1);
    z = a2 + b1;
    sem_post(&sc2_zy);
    return NULL;
```

```c
void* thread2_func(void* arg) {
    b1 = 20;
    sem_post(&sa2b1);

    b2 = 21;

    sem_wait(&sc2);
    w = b2 + c2;

    sem_wait(&szy);
    sem_wait(&szy);
    x = z - y + w;
    //sem_post(&sx);
    sem_post(&sc1_x);


    return NULL;
}
```

```c
}

void* thread2_func(void* arg) {
    b1 = 20;
    sem_post(&sa2b1);

    b2 = 21;

    sem_wait(&sc2_zy);
    sem_wait(&sc2_zy);
    sem_wait(&sc2_zy);

    w = b2 + c2;

    x = z - y + w;
    //sem_post(&sx);
    sem_post(&sc1_x);
```