Λειτουργικά Συστήματα

Αναφορά τρίτης άσκησης

Κωνσταντίνος Μπουραντάς (23 6145)
bourantas@ceid.upatras.gr

Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών και Πληροφορικής, Πανεπιστήμιο Πατρών

29 Νοεμβρίου 2017

3.1 multi_proc1

Στο συγκεκριμένο πρόγραμμα C καλούμαστε να συγχρονίσουμε έναν αριθμό από διεργασίες (που δίνεται από τον χρήστη) έτσι ώστε να χρησιμοποιούν κατάλληλα έναν κοινό buffer για την εκτύπωση μηνυμάτων . Χωρίς τον κατάλληλο συγχρονισμό παρατηρούμε στην αρχή ότι τα μηνύματα τυπώνονται ανακατεμένα , με αναγραμματισμούς επειδή και οι όλες οι διεργασίες έχουν πρόσβαση σε κοινή μνήμη χωρίς κάποιον έλεγχο. Για να διορθώσουμε το πρόβλημα αυτό κάνουμε χρήση των σηματοφόρων . Αρχικά δημιουργούμε δυο δομές sembuf upκαι down για να υλοποιήσουμε στην συνέχεια τους σηματοφόρους. Έπειτα με την χρήση της semget() δημιουργούμε τον σημαιοφόρο μας . Αφού κάνουμε όσα fork() χρειάζονται για την δημιουργία των ζητούμενων διεργασιών (εκτελούμε argc – 2 επαναλήψεις) , χρησιμοποιούμε τον σημαιοφόρο έτσι ώστε να κλειδώσουμε την προσπέλαση της μνήμης από άλλη διεργασία έτσι ώστε να μην έχουμε αναγραμματισμούς .

```
struct sembuf up = {0, 1, 0};
  struct sembuf down = {0, -1, 0};
  int  v1;

  int my_sem = semget(IPC_PRIVATE, 1, 0600);
  int rep = atoi(argv[1]);

  v1 = semctl(my_sem, 0, GETVAL);
```

Αυτό το καταφέρνουμε κάνοντας “down” τον σημαιοφόρο μέχρι να τελειώσει η διεργασία και έπειτα τον κάνουμε “up” για να την ξεκλειδώσουμε. Κάθε διεργασία αφού προσπελάσει την μνήμη θέτει τον σηματοφόρο “up” . Με αυτόν τον τρόπο δίνεται η πρόσβαση στην κοινή μνήμη σε μια διαδικασία την φόρα.

```
 for (i = 2 ; i < argc  ; i++) {
    v1 = semctl(my_sem, 0, GETVAL);
    int pid = fork();

    if (pid == 0) {
      v1 = semctl(my_sem, 0, GETVAL);
      //printf("child: %d\n", v1);

     for (int j = 0 ; j < rep ; j++){
            semop(my_sem, &down, 1);

        display(argv[i]);

         v1 = semctl(my_sem, 0, GETVAL);

      //printf("value sem: %d\n", v1);
      semop(my_sem, &up, 1);
     }
```

Στην συγκεκριμένη άσκηση η δυσκολία που εμφανίστηκε ήταν στο να καταφέρουμε την αποφυγή των dread locks καθώς αρχικά κάποιες από τις διεργασίες περίμεναν για πάντα.

3.2 multi_proc2

Στο παρακάτω ερώτημα η δυσκολία ήταν να συγχρονίσουμε όλες τις διεργασίες έτσι ώστε να εκτελέσουν όλες πρώτα την συνάρτηση init() και μετά να προχωρήσουν στην εκτέλεση του υπόλοιπου προγράμματος.

```
for (i = 2 ; i < argc  ; i++) {
   v1 = semctl(my_sem, 0, GETVAL);
   int pid = fork();

   if (pid == 0) {
     v1 = semctl(my_sem, 0, GETVAL);
     //printf("child: %d\n", v1);

     semop(my_sem, &down, 1);//{
     init();

     semop(my_sem,&up,1);//}

     semop(my_sem, &down, 1);

    for (int j = 0 ; j < rep ; j++){
       display(argv[i]);
    }

     v1 = semctl(my_sem, 0, GETVAL);

     //printf("value sem: %d\n", v1);
     semop(my_sem, &up, 1);
     exit(0);

   }
```

Αυτό μπορούμε να το επιτύχουμε κλειδώνοντας αρχικά την περιοχή που αφορά την κλήση της init() έτσι ώστε να αναγκάσουμε όλες τις διεργασίες να περιμένουν η μια την άλλη να τελείωσουν με την κλήση της και έπειτα να συνεχίσουν στην εκτέλεση.

3.3 multi_thread1

Στο συγκεκριμένο ερώτημα το ζητούμενο είναι το ίδιο με το πρώτο ερώτημα άλλα αυτή την φορά με την χρήση νημάτων. Αρχικά δημιουργούμε έναν πίνακα από νήματα μεγέθους ανάλογου της εισόδου του χρήστη . Έπειτα , διαμορφώνουμε κατάλληλα τις εισόδους για την συνάρτηση που θα εκτελέσει κάθε νήμα και στην συνέχεια δημιουργούμε το νήμα .

```
for (i = 0; i < num_of_threads; ++i) {

    input_args = malloc(sizeof(struct thread_args));

    (*input_args).reps = reps;
    (*input_args).string = argv[string_topass];
    string_topass++;
```

```
8
9       if (pthread_create(&threads1[i] , NULL, print_function, (void *)input_args
        )) {
10        printf("Error creating thread\n");
11        exit(1);
12      }
13    }
```

Στην main παρατηρούμε ότι κάνουμε join όλα τα νήματα καθώς είναι απαραίτητο για να έχουμε το επιθυμητό αποτέλεσμα . Σε αυτό το σημείο προέκυψε πρόβλημα με τον κώδικα καθώς αρχικά το join είχε τοποθετηθεί μέσα στον βρόγχο με συνέπεια ήταν να μην εκτυπώνονται σωστά αποτελέσματα είτε το πρόγραμμα να κολλάει . Μέσα στην συνάρτηση `print_function()` έχουμε μια επανάληψη while η οποία εκτελεί όσες φορές επιθυμεί ο χρήστης στην συνάρτηση display() . Παρατηρούμε ότι το mutex κλειδώνει μέσα στον βρόγχο και όχι απ'εξω , αυτό συμβαίνει καθώς το αντίθετο θα είχε ως αποτέλεσμα σειριακή εκτέλεση των εντολών. Πρόβλημα στον κώδικα υπήρξε ότι μερικές φόρες τα κάποιο νήμα έχανε την σειρά του και δεν τύπωνε το επιθυμητό μήνυμα.

```
1  void *print_function(void *input) {
2    int i =0;
3    struct thread_args *args = input;
4
5    while (i < args-> reps){
6      pthread_mutex_lock(&lock);
7      display(args -> string);
8      i++;
9      pthread_mutex_unlock(&lock);
10   }
11    pthread_exit(NULL);
12 }
```

3.4 multi_thread2

Στο συγκεκριμένο ερώτημα το ζητούμενο είναι το ίδιο με το δεύτερο ερώτημα άλλα αυτή την φορά με την χρήση νημάτων. Επειδή ζητούμενο είναι να συγχρονίσουμε όλα τα νήματα προκειμένου να εκτελέσουν όλα αρχικά την συνάρτηση init() και μετά να προχωρήσουν στην εκτέλεση της δισπλαψ , προχώρησα στις παρακάτω αλλαγές.

```
1  void *print_function(void *input) {
2    int i = 0;
3    struct thread_args *args = input;
4
5    pthread_mutex_lock(&lock);
6    init();
7    done++;
8    if (done == num_of_threads) {
9      predicate = 1;
10   }
11   if (predicate != 0) {
12     pthread_cond_broadcast(&cond_var);
13   }
14
```

```
15    pthread_mutex_unlock(&lock);
16
17    pthread_mutex_lock(&lock);
18
19    while (predicate == 0) {
20      pthread_cond_wait(&cond_var, &lock);
21    }
22    predicate = 1;
23    pthread_mutex_unlock(&lock);
24
25    while (i < args-> reps) {
26      pthread_mutex_lock(&lock);
27      display(args -> string);
28      i++;
29      pthread_mutex_unlock(&lock);
30
31    }
32
33    pthread_exit(NULL);
34
35  }
```

Όπως βλέπουμε θα χρησιμοποιήσουμε επιπλέον τις συναρτήσεις `pthread_cond_broadcast` και `pthread_cond_wait` . Αυτό που κάνουμε στην ουσία είναι κάθε νήμα αφού εκτελέσει την init το βάζουμε να περιμένει (με την `pthread_cond_wait`). Παράλληλα έχουμε έναν μετρητή done ο οποίος μετράει πόσα από τα νήματα εκτέλεσαν την init . Το νήμα περιμένει μέχρι η μεταβλητή predicate να γίνει ίση με 1 , το οποίο σηματοδοτεί ότι όλα τα νήματα πλέον μπορούν να πάψουν να περιμένουν και να συνεχίσουν κανονικά την εκτέλεση του προγράμματος. Ενημερώνουμε τα νήματα να σταματήσουν να περιμένουν με την χρήση της `pthread_cond_broadcast` η οποία τα ειδοποιεί όλα ταυτόχρονα.