Sleeping Barber και Producer-Consumer problem: Μελέτη και υλοποίηση με pthreads

-

Παράλληλα και κατανεμημένα συστήματα

Οδυσσέας Νεσλεχανίδης

Επιβλέπων καθηγητής: Χρήστος Γκόγκος

May 30, 2019

Part I
Εισαγωγή

Στην παρούσα τεχνική αναφορά παρουσιάζεται η μελέτη μου πάνω στα κλασικά προβλήματα συγχρονισμού Sleeping Barber και Producer-Consumer. Θεώρησα οφέλιμο να δοκιμάσω να γράψω ένα πρόγραμμα σε γλώσσα C με pthreads με μια λύση για κάθε πρόβλημα, προκειμένου να θεμελιώσω καλύτερα την κατανόηση μου επί των θεμάτων. Στην πορεία, βρήκα μεγαλύτερο ενδιαφέρον στην προσπάθεια ταυτόχρονης λύσης και των δύο προβλημάτων σε ένα πρόγραμμα[1].

Στο παρόν, θα προσπαθήσω να κάνω μια κριτική παρουσίαση των προβλημάτων και των λύσεων που επέλεξα να υλοποιήσω γι'αυτά, αποφεύγοντας να επαναλάβω όσα παρουσιάζονται επαρκώς στα σχετικά λήμματα της wikipedia. Θα γίνει, εν τούτοις εκτενής αναφορά σε αυτά, καθώς και σε ψευδοκώδικα από άλλες ελεύθερες πηγές, πάντοτε με τη σχετική παραπομπή.

Part II
Περιγραφή της υλοποίησης

Σύγκριση και θεωρητική οριοθέτηση

Το πρόβλημα Παραγωγού-Καταναλωτή (Producer-Consumer)[2], ή αλλιώς Bounded Buffer problem, αφορά τη διαφορά στο ρυθμό εισαγωγής και αφαίρεσης δεδομένων (throughput) σε έναν περιορισμένο χώρο μνήμης, από τον παραγωγό και τον καταναλωτή αντίστοιχα. Είναι λοιπόν, στη βάση του, ένα πρόβλημα συντονισμού του έργου των δυο νημάτων, ούτως ώστε να μην υπάρχουν λάθος αιτήματα προς τη μνήμη.

Το πρόβλημα του Κοιμώμενου Κουρέα (Sleeping Barber)[3] αφορά την αποφυγή συνθηκών ανταγωνισμού (race conditions) μεταξύ του κουρέα, που αφαιρεί πελάτες από μια περιορισμένου μεγέθους αίθουσα αναμονής, και των πελατών που προσθέτουν, ας πούμε, τους εαυτούς τους, σε αυτήν. Στη βάση του αφορά, λοιπόν, την εξασφάλιση ασφάλειας νημάτωσης (thread safety)[4] σε εργασίες που απαιτούν πρόσβαση στο χώρο μνήμης που αντιπροσωπεύει την αίθουσα αναμονής. Έτσι, αποφεύγεται ο κίνδυνος υλοποιούμενες λύσεις να οδηγούν σε νεκρό κλείδωμα (deadlock).

Από τις παραπάνω παραγράφους γίνεται ξεκάθαρο ότι τα δύο προβλήματα, παρότι φαινομενικά όμοια, αφορούν διαφορετικά ζητήματα συγχρονισμού νημάτων/διεργασιών. Εν τούτοις, στις περισσότερες περιπτώσεις, τα δύο σύνολα ζητημάτων τείνουν να χρήζουν ταυτόχρονης διαχείρισης. Γι'αυτό το λόγο, επέλεξα να μελετήσω τα προβλήματα γράφοντας ένα πρόγραμμα και για τα 2.

Κουρά με ουρά

Η απλούστερη λύση για το πρόβλημα του κοιμώμενου κουρέα, όπως παρουσιάζεται στη wikipedia, μπορεί να οδηγήσει σε λιμοκτονία (starvation) ενός νήματος πελάτη. Για αυτό ευθύνεται το γεγονός ότι, ανάλογα με τις προδιαγραφές των αρχέγονων δομών συγχρονισμού (synchronization primitives) που παρέχονται με την εκάστοτε βιβλιοθήκη, μπορεί η προτεραιότητα των νημάτων να εξασφαλίζεται με διαφορετικούς τρόπους ή να αφήνεται απροσδιόριστη [5].

Αυτό με οδήγησε στη χρήση C++ προκειμένου να εκμεταλλευτώ τη δομή STL Queue. Πρόκειται για μια templated δομή, που ήλπιζα να χρησιμοποιήσω εισάγοντας σε αυτήν κλειδωμένους σημαφόρους για κάθε νεοεισερχόμενο νήμα-πελάτη. Καθώς, όμως, ο τύπος δεδομένων sem_t, που αντιπροσωπεύει το σημαφόρο της βιβλιοθήκης <semaphore.h> , δεν επιδέχεται συμβατικό χειρισμό (στα πρότυπα π.χ. του τύπου int), δεν βρήκα κάποιον τρόπο να το πετύχω. Αναγκάστηκα, τελικά, να στηριχτώ σε μια, φαινομενικά, μη αποδοτική, ad hoc υλοποίηση ουράς σημαφόρων. Η έμπνευση γι'αυτήν προήλθε από το πολύτιμο The Little Book of Semaphores [5].

```
int main(int argc, char* argv[]){
  ...
  int numChairs = atoi(argv[1]);
  waitingRoomQueue = new sem_t[numChairs+1];
  sem_init(&seatsAvailable, 0, numChairs);
  ...
  for(int i = 0; i < numChairs + 1; i++){
    sem_init(&waitingRoomQueue[i], 0, 1);
  }

}
void *customer(void *numChairs){
  int* number_of_chairs = (int*)numChairs;
  int positionInQueue{*number_of_chairs - 1};

  while(1){
    customerMessage(id, arrive_at_barbershop);
    if(sem_trywait(&seatsAvailable)==-1){
      customerMessage(id, waiting_room_full);
      }

    else{
      customerMessage(id, enter_waiting_room);
      while(positionInQueue > 0){
      sem_wait(&waitingRoomQueue[positionInQueue]);
      sem_post(&waitingRoomQueue[positionInQueue+1]);
      positionInQueue--;
```

```
            }
        }
        sem_wait(&barberChairEmpty);
        ...
        sem_post(&waitingRoomQueue[1]);
        ...
    }
}
```

Παραπάνω, η sem_trywait δοκιμάζει να μειώσει το σημαφόρο seatsAvailable. Εάν επιτύχει, που σημαίνει ότι ο σημαφόρος ήταν αρχικά μεγαλύτερος του μηδέν, επιστρέφει 0 (επιτυχία) και η εκτέλεση του προγράμματος συνεχίζει απορρίπτοντας τη συνθήκη. Σημειώστε ότι, σε αυτήν την περίπτωση, ο σημαφόρος μεταβλήθηκε κατά τον έλεγχο, παρότι ο έλεγχος απέτυχε. Αυτό είναι σκόπιμο.

Εάν η sem_trywait συναντήσει το σημαφόρο seatsAvailable σε κατάσταση που δε δύναται να μειωθεί (0), η sem_trywait επιστρέφει -1 και τερματίζει, σε αντίθεση με την sem_wait, που στην ίδια περίπτωση θα έθετε το νήμα σε αναμονή.[6] Ακολούθως, εισέρχεται στη συνθήκη, τυπώνει μήνυμα αποχώρησης του πελάτη λόγω συμφόρησης της αίθουσας αναμονής, και φτάνει στο τέλος της while για να τρέξει από την αρχή, με νέο id πελάτη (δείτε το επόμενο block κώδικα).

Το waitingRoomQueue είναι ένας πίνακας σημαφόρων μεγέθους numChairs +1 που αρχικοποιούνται με 1. Ο σημαφόρος waitingRoomQueue[numChairs] αυξάνεται σε κάθε εκτέλεση του σεναρίου else κατά 1, χωρίς ποτέ να μειωθεί. Αυτή είναι μια αδυναμία που δεν έχω σκεφτεί πώς θα επιλύσω, δε φαίνεται να δημιουργεί ωστόσο πρόβλημα βραχυπρόθεσμα.

Η τιμή του σημαφόρου waitingRoomQueue[0] δεν μεταβάλλεται σε κανένα σημείο του προγράμματος. Αφέθηκε για λόγους διαισθητικής ομαλότητας. Η αναμονή του κορυφαίου πελάτη της ουράς, στην πραγματικότητα εξαρτάται από το σημαφόρο barberChairEmpty.

Ανταγωνισμός νημάτων παραγωγής πελατών

Στη wikipedia, η λύση του χαρακτηριστικού προβλήματος του κοιμώμενου κουρέα, παρουσιάζεται σε συνθήκη ανταγωνισμού μεταξύ κουρέα και πελάτη για την πρόσβαση στο χώρο μνήμης που αντιπροσωπεύει την αίθουσα αναμονής. Στη δική μου υλοποίηση ανέβαλα αυτό το ζήτημα, αναθέτοντας στον πελάτη τα καθήκοντα προσθαφαίρεσης του εαυτού του στην ουρά αναμονής. Προχώρησα σε αυτήν τη διασκευή με την προοπτική να κάνω τη ζωή μου ευκολότερη σε μελλοντική υλοποίηση όπου θα προσθέσω πολλαπλά νήματα κουρέων.

Το ίδιο ζήτημα ανταγωνισμού για την πρόσβαση εμφανίστηκε ξανά όταν προσέθεσα πολλά νήματα-πελάτες. Η υλοποίηση της wikipedia προσομοιώνει πολλαπλούς πελάτες στηριζόμενη σε έναν ατέρμονο βρόχο του ίδιου νήματος. Θεώρησα αυτού του είδους την προσομοίωση ελάχιστα ρεαλιστική και τεχνικά απλουστευτική, και γι'αυτό επέλεξα να πολλαπλασιάσω τα νήματα-πελάτες, που ωστόσο

άφησα να λειτουργούν σε ατέρμονα βρόχο. Έτσι, η λύση της wikipedia για το συγχρονισμό κουρέα - πελατών, εμφανίζεται στη δική μου υλοποίηση μεταξύ κουρέα και "νημάτων παραγωγής πελατών". Η επιλογή αυτού του νέου ονόματος ίσως γίνει πιο κατανοητή εαν ληφθεί υπόψιν ο τρόπος που επέλεξα να χειριστώ την ονοματοδοσία κάθε πελάτη.

```
//guards critical section where customer() pulls
//a unique id from global counter new_id.
pthread_mutex_t id_lock;
int new_id{1};
...
void *customer(...){
  ...
  while(1){
    pthread_mutex_lock(&id_lock); //fetch unique id for
    id = new_id;                  //customer
    new_id++;
    pthread_mutex_unlock(&id_lock);
    ...
  }
  ...
}
```

Αξίζει να σημειωθεί ότι, στη δική μου προσέγγιση, προκύπτει ένα πρόβλημα από την ελάττωση του αριθμού ενεργών νημάτων παραγωγής πελατών, όποτε κάποιο από αυτά εισέρχεται στην αίθουσα αναμονής. Αυτή η αδυναμία γίνεται ολότελα φανερή εφόσον οριστεί μεγάλος αριθμός θέσεων στην αίθουσα αναμονής από το χρήστη (απαιτείται ως όρισμα κατά την εκτέλεση). Καθώς ο αριθμός νημάτων παραγωγής πελατών περιορίζεται στα 5, υπάρχει το ενδεχόμενο όλα τα νήματα να τεθούν σε αναμονή μέσα στην αίθουσα αναμονής. Σε μια τέτοια περίπτωση, δε θα υπάρχουν αναχωρήσεις νέων πελατών μέχρι να ελευθερωθεί το κορυφαίο νήμα από τον κουρέα, για να ξαναμπεί στο βρόχο. Δεν μπόρεσα να σκεφτώ έναν ικανοποιητικό τρόπο να αντιμετωπίσω το πρόβλημα, προς επίτευξη μεγαλύτερου ρεαλισμού, χωρίς να ξεφύγω πολύ από τις τεχνικές προδιαγραφές του προβλήματος. Το άφησα, λοιπόν, ως έχει.

```
void *customer(void *numChairs){
  ...
  while(1){
    ...
    else{
      ...
      sem_wait(&barberChairEmpty);
      pthread_mutex_lock(&room_state_change);
        customerMessage(id, check_on_barber);
```

```
            sem_post(&waitingRoomQueue[1]);
            sem_post(&seatsAvailable);
            sem_post(&barberPillow);
          pthread_mutex_unlock(&room_state_change);

          sem_wait(&seatBelt);
          customerMessage(id, leave_successfully);
          sem_post(&barberChairEmpty);
        }
    }
}
```

Παραπάνω, βλέπουμε πώς κινείται το κορυφαίο στην ουρά νήμα, μόλις ελευθερωθεί από το σημαφόρο barberChairEmpty, που θα αυξήσει ο φρεσκοκουρεμένος πελάτης πριν φτάσει στο τέλος της while και επιστρέψει για επανάληψη με νέο id (δείτε παραπάνω). Η μόνη παρέμβαση του νήματος - κουρέα είναι για την απελευθέρωση του νήματος παραγωγής πελατών μόλις αυτό φτάσει στο sem_wait(&seatBelt).

Το νήμα κουρέας

Παραπάνω, συναντάμε για πρώτη φορά το σημαφόρο barberPillow. Σε αυτό το σημαφόρο στηρίζεται ο συντονισμός του νήματος - κουρέα. Η συνάρτηση barber είναι αρκετά απλή:

```
void *barber(void *unused){
  int isAsleep{-1};
  while(1){
    sem_getvalue(&barberPillow, &isAsleep);
    if(isAsleep == 0){
      std::cout<<"The barber is sleeping.\n";
    }
  sem_wait(&barberPillow);
  std::cout <<"The barber is cutting hair.\n";
  randWait(5);
  std::cout <<"The barber has finished cutting hair.\n";
  sem_post(&seatBelt);
  randWait(1);
  }
}
```

Πριν τερματίσει, ο κουρέας απελευθερώνει το νήμα παραγωγής πελατών που έχει περάσει και κλειδώσει την κορυφή της ουράς (σημαφόρος barberChairEmpty). Το νήμα αυτό ξεκλειδώνει με τη σειρά του το σημαφόρο barberChairEmpty, πριν επιστρέψει στην αρχή του βρόχου.

Προτού ο κουρέας προλάβει να επιστρέψει στο sem_wait(&barberPillow) για να κοιμηθεί, μπορεί ένα νήμα παραγωγής πελατών που βρισκόταν σε αναμονή ξεκ-

λειδώματος του barberChairEmpty, να προλάβει να αυξήσει το σημαφόρο barberPillow. Σε αυτή την περίπτωση το νήμα κουρέας θα μειώσει απευθείας το σημαφόρο barberPillow, χωρίς να χρειαστεί να περιμένει (δηλαδή, δε θα κοιμηθεί). Η συνθήκη που βλέπουμε παραπάνω καθιστά πιθανότερο το μήνυμα ύπνου να μην εμφανίζεται όταν ένα νήμα παραγωγής πελατών έχει τρέξει το

customerMessage(id, check_on_barber);

Δεν εξασφαλίζει ωστόσο κάτι τέτοιο. Κρίνω ότι η χρήση σημαφόρου θα ήταν υπερβολή.

Τυχαιότητα και η συναρτηση randWait()

Για να πετύχω τυχαίους χρόνους κοντά σε μια ζητούμενη διάρκεια, για κάθε εργασία, (κούρεμα, προσέλευση στο κουρείο, ταχύτητα νυστάγματος), έγραψα τη συνάρτηση randWait():

```
void randWait(int approxSeconds){
std::this_thread::sleep_for(static_cast<std::chrono::milliseconds>
(approxSeconds * (rand() % 1000) ) );
}
```

Ο σπόρος τυχαιότητας (seed) για την rand() απαιτείται ως όρισμα κατά την εκτέλεση του προγράμματος. Εν τούτοις, φαντάζει άχρηστος, δεδομένου ότι ο μη ντετερμινισμός στους χρονισμούς των παράλληλων νημάτων παραγωγής πελατών είναι αρκετός, ώστε κάθε διαδοχική εκτέλεση, ακόμα και με τον ίδιο σπόρο τυχαιότητας, να διαφέρει απο την προηγούμενη.

Λοιπά ζητήματα

Για τα μηνύματα των νημάτων παραγωγής πελατών, χρειάστηκε να δημιουργήσω μια νέα συναρτηση customerMessage(int id, messageType message) (το messageType είναι enum). Αυτό επειδή κάθε χρήση του operator<< συνιστά ξεχωριστή κλήση της cout, γι'αυτό και αρκετά μηνύματα κόβονταν στην μέση λόγω cout που εκτελούνταν παράλληλα σε διαφορετικά νήματα. Η λύση[7] ήταν απλή και ανευρέθηκε με αναζήτηση στο διαδίκτυο.

Γενικά, τα απαιτούμενα ορίσματα, αυτά του αριθμού των θέσεων στην αίθουσα αναμονής και του σπόρου τυχαιότητας, δεν επαρκούν για να εκθέσουν εύκολα όλες τις διαφορετικές συνθήκες που μπορεί να επικρατούν στο μαγαζί.

Σε έναν υπολογιστή με τις προδιαγραφές του δικού μου, για παράδειγμα, δεν υπάρχει τρόπος, μέσω ορισμάτων, να αυξομειωθεί αισθητά η συχνότητα που ο κουρέας βρίσκει χρόνο να πάρει έναν υπνάκο. Κάτι τέτοιο μπορεί, ωστόσο, να επιτευχθεί εύκολα, αυξομειώνοντας το όρισμα της συνάρτησης randWait(), όταν καλείται μεταξύ της αναχώρησης ενός νέου πελάτη, και της άφιξης του στο κουρείο. Μικρομετατροπές για διευκόλυνση αφήνονται για αργότερα.

Στην παρούσα κατάσταση του κώδικά, αναμενόμενα πρέπει να είναι μικρά κυρίως στυλιστικά παραπτώματα όπως η έλλειψη συνοχής στην ονοματοδοσία των μεταβλητών (camelCase, under_score) και στην επιλογή μεταξύ mutexes και δυαδικών σημαφόρων.

Ολόκληρο το αρχείο cpp υπάρχει στη διεύθυνση:

https://github.com/thcloak/pardist/blob/master/sleepingbbq.cpp

References

[1] https://github.com/thcloak/pardist/blob/master/sleepingbbq.cpp

[2] https://en.wikipedia.org/wiki/Producer-consumer_problem

[3] https://en.wikipedia.org/wiki/Sleeping_barber_problem

[4] https://en.wikipedia.org/wiki/Thread_safety

[5] p.47-52 The Little Book of Semaphores - Green Tea Press - Allen B. Downey

[6] https://linux.die.net/man/3/sem_trywait

[7] https://stackoverflow.com/questions/15033827/multiple-threads-writing-to-stdcout-or-stdcerr