Operating Systems Report

MYY-601

Αναφορά στην 1η Προγραμματιστική Άσκηση του μαθήματος "Λειτουργικά Συστήματα" του καθηγητή Στέργιου Αναστασιάδη.

Η εργασία υλοποιήθηκε σε *Virtual Machine* με λειτουργικό σύστημα το *Ubuntu-Linux* Χρησιμοποιήσαμε κατά τις υποδείξεις του καθηγητη το GDB debugger και τις σχετικές βιβλιοθήκες που μας δόθηκαν από την εκφώνηση του μαθήματος.

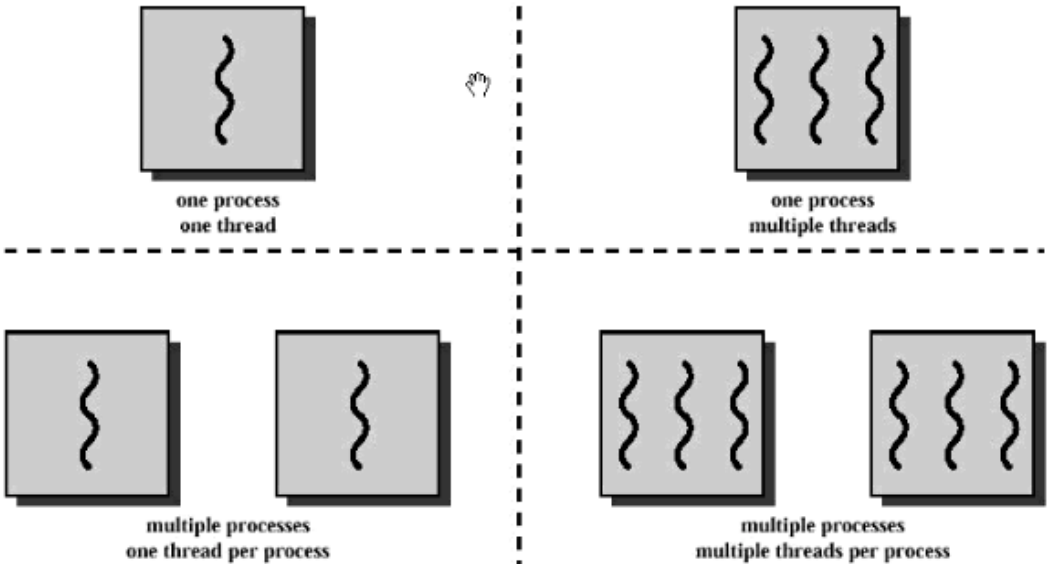
Για οποιαδήποτε περεταίρω διευκρύνιση σας παρακαλώ επικοινωνείστε μαζί μας στο mail *t.dimosiaris@gmail.com*

Θανάσης Δημοσιάρης, 3216
Σεραφείμ Αντωνίου, 2640

Εισαγωγή

Ένα νήμα εκτέλεσης (thread), είναι η μικρότερη ακολουθία προγραμματισμένων εντολών που μπορεί να υποστεί διαχείριση ανεξάρτητα από το λειτουργικό σύστημα. Μπορούν να υπάρχουν πολλαπλά νήματα μέσα στην ίδια διεργασία, τα οποία μπορούν να μοιράζονται πόρους από το σύστημα, όπως τη μνήμη. Σε έναν απλό επεξεργαστή η διαδικασία της πολυνημάτωσης (multithreading) πραγματοποιείται με τον εξής τρόπο: ο επεξεργαστής μεταπηδά μεταξύ νημάτων, αυτό γίνεται ταχύτατα ανά τακτά χρονικά διαστήματα δίνοντας έτσι την ψευδαίσθηση στον χρήστη πως οι διεργασίες εκτελούνται ταυτόχρονα. Σήμερα, σε πιο σύγχρονα και πολύπλοκα υπολογιστικά συστήματα οι επεξεργαστές (multithreaded, hyperthreaded) διαθέτουν πολλούς επεξεργαστικούς πυρήνες, στην περίπτωση αυτή τα νήματα όντως εκτελούνται ταυτόχρονα και κάθε πυρήνας εκτελεί ένα νήμα.

Εικόνες νημάτων:



*Οι λέξεις με διαφορετικό χρώμα από αυτό του κειμένου, είναι hyperlinks προς το αντίστοιχο αντικείμενο.

Στην εργαστηριακή άσκηση στα πλαίσια του μαθήματος των Λειτουργικών Συστημάτων μας ζητήθηκε να υλοποιήσουμε έναν πολυνηματικό διακομιστή αποθήκευσης ζευγών κλειδιού-τιμής με την χρήση της βάσης δεδομένων **KISSDB** (Keep It Simple Stupid Database). Η χρήση της βάσης αυτής, αποσκοπεί στην αποθήκευση των ζευγών κλειδιού-τιμής.

Βιβλιοθήκη Pthreads

Προκειμένου να έχει μία σχετική συνέπεια η κοινόχρηστη **FIFO (First-In-First-Out)** ουρά που υλοποιήσαμε, δώσαμε την δυνατότητα σε έναν παραγωγό και έναν καταναλωτή να πραγματοποιούν τροποποιήσεις στην ουρά **ταυτόχρονα**, αποκλείοντας όμως την ταυτόχρονη τροποποίηση της ουράς από περισσότερους καταναλωτές. Ο αμοιβαίος αυτός αποκλεισμός επιτεύχθηκε με την βοήθεια των έτοιμων συναρτήσεων **pthread_mutex_lock()** και **pthread_mutex_unlock()**, **pthread_cond_wait()** και **pthread_cond_signal()** οι οποίες παίρνουν την κατάλληλη μεταβλητή αμοιβαίου αποκλεισμού.

Το **A.P.I.** των **pthreads** περιέχει κάποιες υπο-ρουτίνες (subroutines) οι οποίες χωρίζονται στις εξής κατηγορίες:

- **Νηματική Διαχείρηση (Thread handling):** Ενέργειες οι οποίες σχετίζονται με διάφορες λειτουργίες των νημάτων όπως η σύνδεση / αποσύνδεσή τους, η δημιουργία τους κλπ.

- **Αμοιβαίος Αποκλεισμός (Mutual Exclusion aka Mutex):** Οι ενέργειες αυτές σχετίζονται άμεσα με το θέμα της διαχείρισης του συγχρονισμού. Διαχειρίζονται την δημιουργία / καταστροφή, καθώς και το κλείδωμα / ξεκλείδωμα των mutexes.

- **Μεταβλητές Κατάστασης (Condition Variables):** Μεταβλητές οι οποίες αφορούν τις πολυπληθείς επικοινωνίες μεταξύ των νημάτων που μοιράζονται τα διάφορα mutexes. Επιπρόσθετα, διαχειρίζονται την δημιουργία / καταστροφή ή την αποστολή / αναμονή διαφόρων σημάτων.

- **Ταυτοχρονισμός:** Τα νήματα μπορούν να εκτελούνται παράλληλα/ταυτόχρονα σε διαφορετικούς επεξεργαστές, επιτυγχάνοντας κατά αυτόν τον τρόπο την σωστή διαμοίραση των πόρων.

Αρχικοποίηση των mutexes και των pthread_cond:

```
// INITIALIZATION OF THE PTHREAD CONDITIONS
pthread_cond_t non_full_queue = PTHREAD_COND_INITIALIZER;    // SIGNAL FOR IF THE QUEUE IS NOT FULL
pthread_cond_t non_empty_queue = PTHREAD_COND_INITIALIZER;  // SIGNAL FOR IF THERE IS SOMETHING IN THE QUEUE

// INITIALIZATION OF THE MUTEXES
pthread_mutex_t queue_producer_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t queue_consumer_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t queue_size_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t stats_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t db_put_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t get_counter_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Παραδείγματα συναρτήσεων pthread_mutex_unlock() & pthread_mutex_lock():

```
98   void pushed_element(){
99     pthread_mutex_lock(&queue_size_mutex);
100    QUEUE_TAIL++;
101    QUEUE_TAIL = (QUEUE_TAIL % (int)MAX_QUEUE_SIZE);
102    QUEUE_SIZE++;
103    pthread_mutex_unlock(&queue_size_mutex);
```

```
294  // GET THE STATISTICS
295  pthread_mutex_lock(&stats_mutex);
296  gettimeofday(&service_time_temp, NULL);
297  completed_requests++;          // INCREASE THE COMPLETED REQUESTS
298  total_service_time += (service_time_temp.tv_usec - entry_time.tv_usec) +
299    (service_time_temp.tv_sec - entry_time.tv_sec) * 1000000;
300  total_waiting_time += (got_it_from_queue_time_temp.tv_usec - entry_time.tv_usec) +
301    (got_it_from_queue_time_temp.tv_sec - entry_time.tv_sec) * 1000000;
302  pthread_mutex_unlock(&stats_mutex);
```

Υλοποίηση / Implementation

Για την υλοποίηση της εργαστηριακής μας άσκησης ακολουθήσαμε τα εξής βήματα:

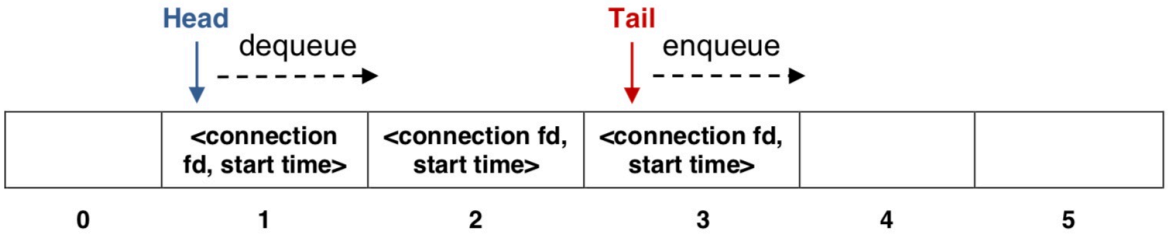
- **Αρχικοποίηση μεταβλητών:** Ορισμός default μεγέθους ουράς (MAX_QUEUE_SIZE = 10), αριθμού νημάτων, καθώς επίσης και διάφορων σταθερών για τα threads.

Αρχικοποίηση global μεταβλητών και παραμέτρων:

```
32  #define MY_PORT                 6767
33  #define BUF_SIZE                1160
34  #define KEY_SIZE                 128
35  #define HASH_SIZE               1024
36  #define VALUE_SIZE              1024
37  #define MAX_PENDING_CONNECTIONS   10
38  #define MAX_QUEUE_SIZE            10
39  #define CONSUMER_THREADS          10
40
41
42  // INITIALIZATION OF THE STATISTICS VARIABLES
43  long total_waiting_time = 0;
44  long total_service_time = 0;
45  int completed_requests = 0;
46
47  // THE COUNTER AND THE TERMINATION FLAG
48  int terminate = 0;
49  int get_counter = 0;
```

- **Κυκλική ουρά FIFO:** Κατασκευή και υλοποίηση κυκλικής ουράς **FIFO (First-In-First-Out)** όπως και διάφορων μεταβλητών και συναρτήσεων για την διαχείριση της, η **tale** της οποίας βρίσκεται πάντα δεξιότερα (όπως στην εικόνα της εκφώνησης) από το **head**. Πρόκειται στην ουσία για έναν κυκλικό πίνακα και κάθε φορά που φτάνουμε στο τέλος του κάνουμε mod για να επιστρέψουμε πάλι στην αρχή. Αποθηκεύουμε το μέγεθος του πίνακα κάθε χρονική στιγμή σε μία μεταβλητή QUEUE_SIZE. Ο πίνακας αποτελείται από ένα *head* από το οποίο ”βγάζουν” οι καταναλωτές και ένα *tail* μέσα στο οποίο προσθέτει συνεχώς ο παραγωγός. Στην δεδομένη περίπτωση ο παραγωγός και ο καταναλωτής δεν είναι άλλος παρά ο ίδιος ο server ο οποίος βάζει ένα αντικείμενο struct (παράγει) που το ονομάζουμε *queue_element* που έχουμε ορίσει εμείς και ο καταναλωτής που βγάζει ένα

τέτοιο αντικείμενο απο την ουρά (καταναλώνει). Έχουμε *έναν* παραγωγό που είναι το βασικό νήμα στη main λειτουργία του προγράμματος. Κατά την αλληλεπίδραση με το head της ουράς, *ένας* καταναλωτής τη φορά αφαιρεί, ενώ παράλληλα έχουμε τις ειδικές συναρτήσεις **pthread_condition_wait()**, **pthread_condition_signal()** καθώς και τα lock-unlock των mutexes για να εξασφαλίσουμε την προστασία τους. Συνεπώς, κάθε φορά ένας "βγάζει"-ένας "βάζει", αλλά γίνεται να βάλουν και να βγάλουν ταυτόχρονα. Για την καλύτερη οργάνωση του κώδικα μας, δημιουργήσαμε δύο συναρτήσεις την **popped_element()** και **pushed_element()** στις οποίες ενσωματώσαμε όλο το logic που χρειάζεται για να πειράξουμε την ουρά. Πιο συγκεκριμένα, χρησιμοποιεί mutexes για να μεταβάλει την κοινόχρηστη μεταβλητή **QUEUE_SIZE** η οποία αυξάνεται όταν προσθέτουμε κάποιο element και μειώνεται όταν αφαιρούμε ένα. Κάθε φορά που προσθέτουμε ένα αντικείμενο στην ουρά μας (**push**), βάζουμε το αντικείμενο στην **tail** και την αυξάνουμε για να είναι έτοιμη να δεχθεί το επόμενο αντικείμενο. Αντίστοιχα όταν βγάζουμε ένα στοιχείο από την ουρά το **QUEUE_SIZE** μειώνεται και η αφαίρεση από την ουρά (**pop)** γίνεται από το **head** που μόλις κάνουμε pop αυξάνεται και αυτό για να μπορεί να μας δώσει το επόμενο στοιχείο της ουράς. Με αυτόν τον τρόπο επειδή οι pointers για το πού κάνω push και pop είναι διαφορετικοί και με την βοήθεια της κοινόχρηστης μεταβλητής QUEUE_SIZE για τους ελέγχους της ουρας(αν δεν είναι άδεια ή δεν είναι γεμάτη), μπορούμε να βάζουμε και να βγάζουμε από την ουρά ταυτόχρονα τα στοιχεία του struct μας. Τα στοιχεία αυτά μεταφέρουν το socketfd που κάναμε accept() ώστε να ξέρει ο καταναλωτής σε ποιο socket θα αναφερθεί κάθε φορά και την στιγμή που το request μπήκε στην ουρά μας.



Δημιουργία της κυκλικής FIFO ουράς:

```
struct queue_element
{
  int socketfd;
  struct timeval timev;
};

int QUEUE_HEAD = 0; // WITH WHICH YOU EXTRACT ELEMENTS FROM THE QUEUE
int QUEUE_TAIL = 0; // WITH WHICH YOU ADD ELEMENTS FROM THE QUEUE
int QUEUE_SIZE = 0;

// THE QUEUE:
struct queue_element queue[MAX_QUEUE_SIZE];

// THE PROCESS THAT IS CALLED WHEN AN ELEMENT IS ADDED TO THE QUEUE

void pushed_element(){
  pthread_mutex_lock(&queue_size_mutex);
  QUEUE_TAIL++;
  QUEUE_TAIL = (QUEUE_TAIL % (int)MAX_QUEUE_SIZE);
  QUEUE_SIZE++;
  pthread_mutex_unlock(&queue_size_mutex);
}

// THE PROCESS THAT IS CALLED WHEN AN ELEMENT IS EXTRACTED FROM THE QUEUE

void popped_element(){
  pthread_mutex_lock(&queue_size_mutex);
  QUEUE_HEAD++;
  QUEUE_HEAD = (QUEUE_HEAD % (int)MAX_QUEUE_SIZE);
  QUEUE_SIZE--;
  pthread_mutex_unlock(&queue_size_mutex);
}
```

- Αλλεπάλληλη δημιουργία νημάτων, όπου κάθε νήμα είναι ένας καταναλωτής (consumer) για την εξυπηρέτηση των αιτήσεων απο τον client.

```
319
320   // NEED TO CREATE THREADS FOR THE CONSUMER
321    int i;
322    for (i = 0; i < CONSUMER_THREADS; i++)
323    {
324      if(pthread_create(consumer_threads + i, NULL, (void *) &process_request, NULL) != 0)
325        printf("We have a problem jack\n");                                    // INPUT
326    }
327
```

- **Σκελετός του καταναλωτή:**
  - Ο **καταναλωτής** βρίσκεται σε κατάσταση αναμονής και περιμένει ένα signal προκειμένου να “ακούσει” ότι η ουρά **δεν** είναι άδεια.
  - Με την μέθοδο **process_request()** επεξεργάζεται ένα αίτημα του πελάτη η οποία είναι σε έναν βρόχο while(1) περιμένοντας με την pthread_cond_wait(). Αρχικά παίρνει το request από την ουρά, δημιουργεί μια σύνδεση με το socket του client και ακούει με την **read_str_from_socket()**, παίρνει PUT/GET το κάνει parse, ενημερώνει την ουρά οτι βγάλαμε κάτι απο αυτήν με την popped_element(), το εξυπηρετεί και δίνει απαντηση μέσω του socket με την χρήση της συνάρτησης **write_str_to_socket().**

```
void process_request() {

  char response_str[BUF_SIZE], request_str[BUF_SIZE];
  int numbytes = 0;
  Request *request = NULL;
  int new_fd;

  // NEED VARIABLES TO STORE THE IN QUEUE TIME AND THE SERVICE TIME OF EACH REQUEST WHICH I
  struct timeval got_it_from_queue_time_temp, service_time_temp, entry_time;
  sigset_t set;
  sigemptyset(&set);

  sigaddset(&set, SIGSTOP);
// Block signal SIGUSR1 in this thread
  pthread_sigmask(SIG_BLOCK, &set, NULL);

  while(1){
    // GET THE REQUEST FROM THE QUEUE

    pthread_mutex_lock(&queue_consumer_mutex);

    while(QUEUE_SIZE == 0){                  // THE QUEUE IS EMPTY
      pthread_cond_wait(&non_empty_queue, &queue_consumer_mutex);
      if(terminate){
        return;
      }
    }

    new_fd = queue[QUEUE_HEAD].socketfd;  // THE SOCKETFD WE'RE GOING TO USE
    entry_time = queue[QUEUE_HEAD].timev;     // THE TIME THE ELEMENT ENTERED THE QUEUE

    popped_element();
    gettimeofday(&got_it_from_queue_time_temp, NULL);

    pthread_cond_signal(&non_full_queue);
    pthread_mutex_unlock(&queue_consumer_mutex);
```

Παράδειγμα μεθόδου read_str_from_socket():

```
236     // receive message.
237     numbytes = read_str_from_socket(new_fd, request_str, BUF_SIZE);
238     printf("%s\n", request_str);
```

Παράδειγμα μεθόδου write_str_to_socket():

```
291   write_str_to_socket(new_fd, response_str, strlen(response_str));
292   printf("\n\n");
```

- Καταγράφει τους αρχικούς χρόνους του συστήματος με την μέθοδο **gettimeofday()** μόλις το request μπει στην ουρά, μόλις βγει και μόλις εξυπηρετηθεί πλήρως, έτσι υπολογίζει τα στατιστικά στοιχεία αφού εξυπηρετηθούν τα αιτήματα στο τέλος της μέθοδου **process_request()** και κάνει το κατάλληλο update στις κοινόχρηστες μεταβλητές για τα στατιστικά αυτά, αφού πρώτα κλέισει με mutexes το κρίσιμο αυτό σημείο.

Παράδειγμα χρήσης της συνάρτησης gettimeofday():

```
226     popped_element();
227     gettimeofday(&got_it_from_queue_time_temp, NULL);
```

Updating the statistics variables:

```
294   // GET THE STATISTICS
295   pthread_mutex_lock(&stats_mutex);
296   gettimeofday(&service_time_temp, NULL);
297   completed_requests++;           // INCREASE THE COMPLETED REQUESTS
298   total_service_time += (service_time_temp.tv_usec - entry_time.tv_usec) +
299     (service_time_temp.tv_sec - entry_time.tv_sec) * 1000000;
300   total_waiting_time += (got_it_from_queue_time_temp.tv_usec - entry_time.tv_usec) +
301     (got_it_from_queue_time_temp.tv_sec - entry_time.tv_sec) * 1000000;
302   pthread_mutex_unlock(&stats_mutex);
```

- Η συνάρτηση **process_request()** είναι υπεύθυνη για την εξυπηρετηση αιτημάτων, ελέγχει και διαχωρίζει την διαδικασία ανάλογα με την φύση του εκάστοτε αιτήματος, αν είναι δηλαδή αίτημα τύπου **PUT** ή **GET**. Στη συνέχεια, **κλειδώνοντας** και **ξεκλειδώνοντας** τις global μεταλητές με την βοήθεια των **pthreads** και πιο συγεκριμένα με αυτή των συναρτήσεων **phtread_mutex_lock** & **pthread_mutex_unlock()**, καταφέρνει έτσι να διαχειρίζεται επιτυχώς τους αναγνώστες και τους γραφείς. Έχουμε έναν μετρητή που μας λέει αν υπάρχουν **GET** κλήσεις, και όσο αυτές είναι >0 τα νήματα με **PUT** περιμένουν. Μόλις τα **GET** γίνουν 0, τότε βγαίνουμε από τον βρόχο που έχουμε και κλειδώνοντας την κρίσιμη αυτή περιοχή με mutexes, ώστε μόνο ένας γραφέας να μπορεί να γράφει στην βάση κάθε φορά, κάνουμε update την βάση μας με **PUT**. Με αυτόν τον τρόπο διασφαλίζουμε οτι μπορούμε να έχουμε παραπάνω από έναν αναγνώστη κάθε φορά (μιας και δεν τους περιορίζουμε με κάποιον τρόπο) και ακριβώς έναν γραφέα (**PUT**) που να γράφει στην βάση δεδομένων. Αυτό είναι σημαντικό γιατί αν είχαμε παραπάνω από έναν γραφέα την φορά στην βάση θα μπορούσε να μεταβάλει και να πειράξει έμμεσα το σωστό αποτέλεσμα της πληροφορίας που θα μας έδινε η βάση μας. Σε όλη την *process_request()* αναφερόμαστε σε ένα καινούργιο socket, αυτό που έκανε **accept()** ο server και μέσω του struct πέρασε στους καταναλωτές και ονομάζεται **new_fd**.

Έλεγχος είδους αιτήματος με στην process_request():

```
// parse the request.
  if (numbytes) {
    request = parse_request(request_str);
    if (request) {
      switch (request->operation) {
        case GET:
          pthread_mutex_lock(&get_counter_mutex);
          get_counter++;
          pthread_mutex_unlock(&get_counter_mutex);
          // Read the given key from the database.
          if (KISSDB_get(db, request->key, request->value))
            sprintf(response_str, "GET ERROR\n");
          else{
            sprintf(response_str, "GET OK: %s\n", request->value);
            printf("+++GET OK: %s\n", request->value);
          }
          pthread_mutex_lock(&get_counter_mutex);
          get_counter--;
          pthread_mutex_unlock(&get_counter_mutex);
          break;
        case PUT:
          pthread_mutex_lock(&db_put_mutex);
          while(get_counter>0)
            // DO NOTHING, WAIT FOR THE GETTERS TO TERMINATE
          pthread_mutex_lock(&get_counter_mutex);
          // Write the given key/value pair to the database.
          if (KISSDB_put(db, request->key, request->value)){
            sprintf(response_str, "PUT ERROR\n");
          }else{
            sprintf(response_str, "PUT OK\n");
            printf("+++PUT OK\n");
          }
          pthread_mutex_unlock(&get_counter_mutex);
          pthread_mutex_unlock(&db_put_mutex);
          break;
        default:
          // Unsupported operation.
          sprintf(response_str, "UNKNOWN OPERATION\n");     // SAME ERROR WITH
      }
    }else{
      sprintf(response_str, "FORMAT ERROR\n");
    }
  }else{
    // Send an Error reply to the client.
    sprintf(response_str, "FORMAT ERROR\n");
    printf("ERROR IN readfromsocket()\n");
  }
```

- Στην περίπτωση χειροκίνητης διακοπής (**manual override**), κατά την οποία ο χρήστης δώσει από το πληκτρολόγιο την εντολή **CTRL+Z**, το νήμα του παραγωγού καλεί μία συνάρτηση που την ονομάζουμε **termination()** (έχουμε κρατήσει το όνομα από την περσινή μας εργασία). Η συνάρτηση *termination()* έχει την εξής λειτουργία: Ενημερώνει μία global variable/flag που την ονομάζουμε **terminate** και αφορά τα νήματα καταναλωτές και τους

```c
if (signal(SIGTSTP, termination) == SIG_ERR)
 {
   exit(1); // 1 because there was a problem with the signal handling
 }
```

υποδεικνύει οτι είναι ώρα να τερματίσουν. Ξεκλειδώνει τα σήματα καταναλωτές από την *pthread_cond_wait()* που περιμένουν και αφού έχει αλλάξει η μεταβλητή *terminate* μπαινουν μέσα στο *if* όπου και τερματίζουν ομαλά. Οι καταναλωτές αγνοούν το σήμα **SIGSTOP** μέσω της συνάρτησης που μας δόθηκε, **pthread_sigmask()**, πιο αναλυτικά κάνει **SIG_BLOCK** σε ένα set σημάτων που εμείς έχουμε ορίσει μέσω της **sigaddset()** και αποτελείται σε αυτήν την περίπτωση μόνο από το σήμα SIGSTOP. Τέλος, επιστρέφοντας στην συνάρτηση terminate(), μόλις ενημερωθούν τα νήματα καταναλωτή τα οποία είναι αποθηκευμένα σε έναν πίνακα **consumer_threads**, κάνει *join*. Η συνάρτηση αυτή εμφανίζει τα στατιστικά που έχουμε συλλέξει στην συνάρτηση process_request() και υπολογίζει τα *averages*.

Υπολογισμός μέσων όρων στατιστικών χρήσης και προβολή στην οθόνη:

```c
void termination(){
  terminate = 1;
  int i;
  for(i = 0; i < CONSUMER_THREADS; i++){
    pthread_cond_signal(&non_empty_queue);
  }

  // PRINT THE STATS
  printf("\n---------STATS----------\n\n");
  printf("Completed Requests: %d \n", completed_requests);
  printf("Total waiting time: %ld\n", total_waiting_time/(float)completed_requests);
  printf("Total service time: %ld\n---------------------------------\n", total_service_time/(float)completed_requests);


  for(i = 0; i < CONSUMER_THREADS; i++){
    pthread_join(consumer_threads[i],NULL);
  }
  KISSDB_close(db);
  exit(1);
  return;
}
```

Μπλοκάρισμα SIGSTOP από νήματα καταναλωτή

```c
sigset_t set;
sigemptyset(&set);

sigaddset(&set, SIGSTOP);
// Block signal SIGUSR1 in this thread
pthread_sigmask(SIG_BLOCK, &set, NULL);
```

Multithreaded Client

Καταφέραμε να υλοποιήσουμε ένα *πολυνηματικό client*. Πιο συγκεκριμένα, φτιάξαμε ένα νέο όρισμα στη main function, την μεταβλητή "**-t**", βασισμένοι στο ήδη υπάρχον UI που υπήρχε στον κώδικα όταν μας δόθηκε ο κορμός του. Με αυτήν την επιπλέον επιλογή κάνουμε τον **client multithreaded**. Μόλις παρουμε τα ορίσματα και φτάσουμε τέλος στην περίπτωση που επιλέξαμε, το βασικό νήμα του client, η main δημιουργεί με την συνάρτηση **pthread_create()** νήματα, με βάση σταθερές που έχουμε δώσει με *#define* και τα αποθηκεύει σε έναν πίνακα όπως κάναμε και στον πολυνηματισμό του *server*. Τα νήματα αυτά καλούν μια συνάρτηση που ονομάζεται **create_requests()** η οποία δημιουργεί αλλεπάληλα requests για κάθε ένα station με την ίδια λογική που προϋπήρχε τον κορμό του κώδικα που μας δόθηκε. Τα requests που θα δημιοργήσει είναι είτε **PUT** είτε **GET** με βάση μία συνάρτηση **rand()** που ανάγει το αποτέλεσμα της είτε σε 0 είτε σε 1 και αντίστοιχα επιλέγει αν θα κάνει PUT ή GET τυχαία. Επίσης να σημειωθεί πως οι τιμές που παίρνουν τα values στις PUT είναι πάλι από random συνάρτηση. Μόλις ολοκληρωθούν τα PUT και τα GET γίνεται **pthread_join()** και ο client τερματίζει. Ο client για να συνομιλήσει με τον server χρησιμοποιεί *sockets* και την συνάρτηση *talk()* η οποία περιέχει τις προαναφερθέντες συναρτήσεις *read_str_from_socket()* και *write_str_to_socket().*

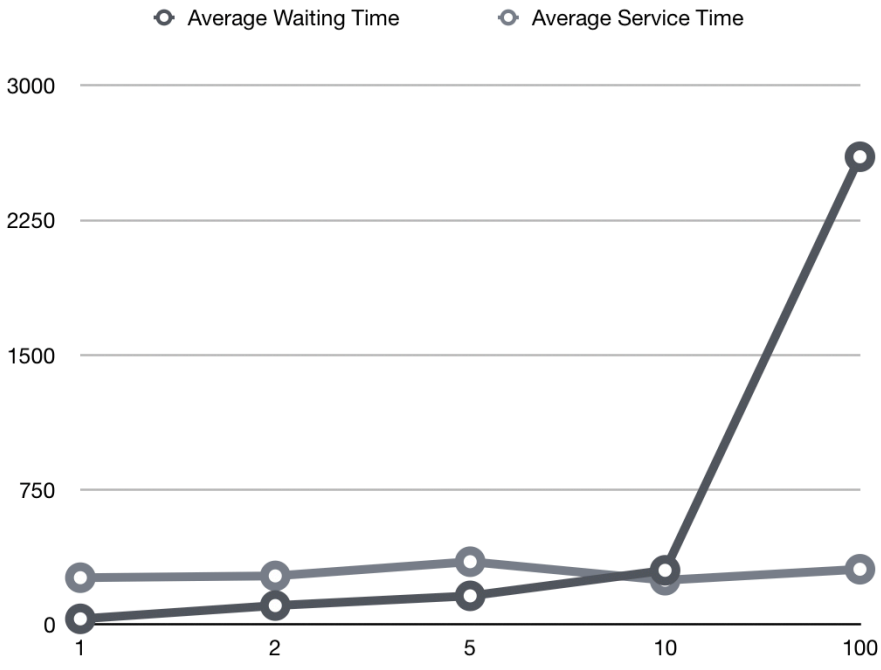
```c
void create_requests(struct sockaddr_in * server_addr){
  char snd_buffer[BUF_SIZE];
  int station, value, mode;

  for (station = 0; station <= MAX_STATION_ID; station++) {
    mode = rand()%2;
    memset(snd_buffer, 0, BUF_SIZE);
    if (mode == 0) {
      // Repeatedly GET.
      sprintf(snd_buffer, "GET:station.%d", station);
    } else if (mode == 1) {
      // Repeatedly PUT.
      // create a random value.
      value = rand() % 65 + (-20);
      sprintf(snd_buffer, "PUT:station.%d:%d", station, value);
    }
    printf("Operation: %s\n", snd_buffer);
    talk(*server_addr, snd_buffer);
  }
}
```

```
else if(mode == DIMOS_MODE){
  int i;
  for(i = 0; i < THREADS_NUM; i++){
    if(pthread_create( requests_threads+ i, NULL, (void *) &create_requests, (void *)&server_addr) != 0)
      printf("We have a problem jack\n");
  }
  for(i = 0; i < THREADS_NUM; i++){
    pthread_join(requests_threads[i],NULL);
  }
}
```

Γραφικές Παραστάσεις

Consumer Threads = 2

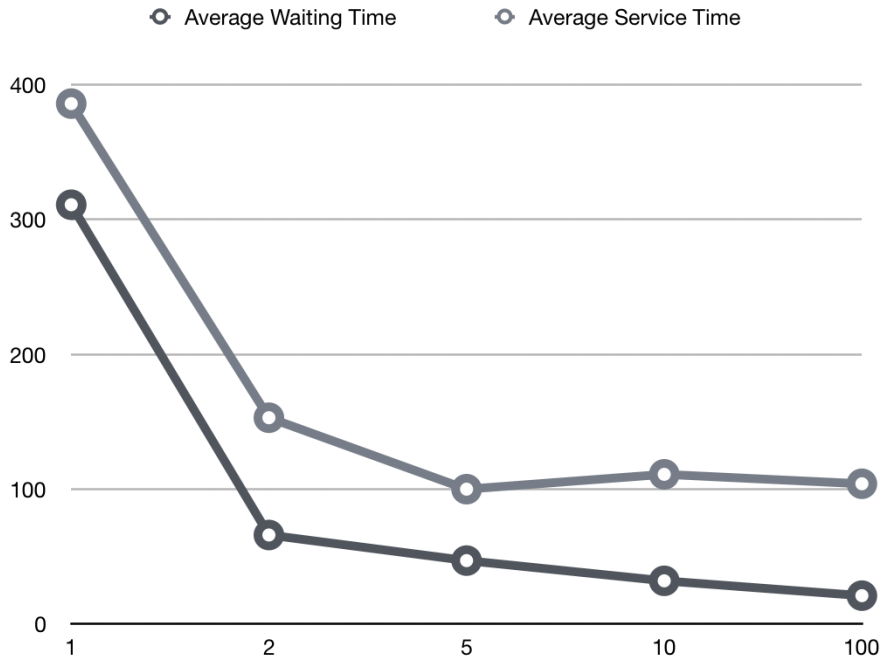
MAX QUEUE SIZE	Average Waiting Time	Average Service Time
1	32	261
2	106	271
5	160	350
10	302	247
100	2603	308



Σταθερά νήματα Καταναλωτή

MAX QUEUE SIZE = 10

Consumer Threads	Average Waiting Time	Average Service Time
1	311	386
2	66	153
5	47	100
10	32	111
100	21	104



Σταθερό μέγεθος ουράς

Ανάλυση Γραφικών Παραστάσεων

Οι παραπάνω γραφικές παραστάσεις έγιναν βασισμένες σε τιμές που πήραμε από το πρόγραμμα μας. Θεωρήσαμε πως για να έχουμε την κατάλληλη ανάλυση του προγράμματος μας, καθώς ο καθηγητής μας προέτρεψε να καθορίσουμε εμείς τον τρόπο που θα πάρουμε τις μετρήσεις μας, να κρατήσουμε θέσουμε στην μία σταθερά μας μία *στατική τιμή* και στην άλλη να μεταβάλουμε τις τιμές της για να δούμε πώς αυτή επηρεάζει την απόδοση του προγράμματος μας.

Στην πρώτη γραφική μας παράσταση θα δείτε πως έχουμε *σταθερό τον αριθμό των νημάτων και μεταβάλλουμε το μέγεθος της ουράς*. *Ο χρόνος εξυπηρέτησης* (δηλαδή αφού βγεί απο την ουρά και μέχρι να εξυπηρετηθεί η αίτηση) παραμένει σταθερός όπως και θα έπρεπε καθώς δεν επηρεάζεται με κάποιον τρόπο απο το μέγεθος της ουράς, ενώ *ο χρόνος αναμονής αυξάνεται* αφού το μέγεθος της ουράς ξεπεράσει τον αριθμό των νημάτων που ξανά είναι απολύτως λογικό αφού όλο και περισσότερες αιτήσεις θα περιμένουν μέσα στην ουρά για τον ίδιο σταθερό αριθμό καταναλωτών.

Στην δεύτερη μας γραφική παράσταση έχουμε *σταθερό το μέγεθος της ουράς και αυξάνουμε τον αριθμό των νημάτων* (θα θέλαμε να σημειώσουμε σε αυτό το σημείο πως είμαστε επιφυλακτικοί για το πώς το VM και ο αριθμός των πυρήνων του υπολογιστή μας επηρεάζουν την συγκεκριμένη γραφική παράσταση). *Ο χρόνος εξυπηρέτησης* για άλλη μία φορά παραμένει πολύ σωστά σχετικά σταθερός όπως και θα έπρεπε μιας και το μέγεθος της ουράς δεν επηρεάζει τον τρόπο με τον οποίο τα νήματα θα εξυπηρετηθούν. Τέλος, *ο χρόνος αναμονής στην ουρά όλο και μειώνεται* μέχρι που παραμένει σχετικά σταθερός όταν φτάσει στο μέγεθος της ουράς.

Σας ευχαριστούμε πολύ για τον χρόνο σας,

Θανάσης Δημοσιάρης
Σεραφείμ Αντωνίου