Γιώργος Αγγελούλης
ΑΜ 1995
Λειτουργικά Συστήματα, Εαρινό Εξάμηνο 2018
1η Εργασία

Server.c

Δομές

```
struct oura{
        int connection_fd;
        struct timeval start_time;

}
```

Κάθε ένα στοιχείο της ουράς είναι ένα τέτοιο struct. Περιέχει τον file descriptor της σύνδεσης και την χρονική στιγμή της έναρξης αυτής.

Η ουρά των αιτήσεων είναι ένας πίνακας από τέτοια struct. Για τη σωστή λειτουργία της ουράς υπάρχουν οι μεταβλητές head, tail και πλήθος που αποθηκεύουν τα αντίστοιχα πράγματα. Τα χρειαζόμαστε για την εισαγωγή και την εξαγωγή στοιχείων στην ουρά.

```
int head; // head ouras
int tail; //tail ouras
int plithos=0; //plithos stoixeiwn ouras
struct oura Q[QUEUE_SIZE]; // h oura, krataei oles tis aitiseis
```

Συναρτήσεις

Η **allagi_timwn** υπολογίζει και αποθηκεύει στις μεταβλητές total_waiting_time, total_service_time, completed_requests το συνολικό χρόνο αναμονής μιας αίτησης, το συνολικό χρόνο εξυπηρέτησης μιας αίτησης, και το πλήθος των αιτήσεων που έχουν εξυπηρετηθεί. Κάθε μία από αυτές αποτελεί κοινόχρηστη περιοχή μεταξύ των νημάτων του προγράμματος για αυτό και υπάρχει ένα mutex για την τροποποίηση αυτών.

```
void allagi_timwn(struct timeval tv1, struct timeval tv2, struct timeval tv)
{
        struct timeval difference;

         difference.tv_sec =tv1.tv_sec -tv.tv_sec ;
         difference.tv_usec =tv1.tv_usec-tv.tv_usec;

          while(difference.tv_usec<0)
          {
            difference.tv_usec+=1000000;
            difference.tv_sec -=1;
          }

    pthread_mutex_lock(&twt);
```

```
            total_waiting_time = total_waiting_time +  1000000LL*difference.tv_sec+
difference.tv_usec;
            pthread_mutex_unlock(&twt);

            pthread_mutex_lock(&tst);
            if( tv1.tv_sec == tv2.tv_sec)
                  total_service_time = total_service_time + tv2.tv_usec - tv1.tv_usec;
            else
                  total_service_time = total_service_time + tv2.tv_usec + (1000000-
tv1.tv_usec);
            pthread_mutex_unlock(&tst);

            pthread_mutex_lock(&cr);
            completed_requests++;
            pthread_mutex_unlock(&cr);
}
```

Συναρτήσεις **readerr**() και **writerr**(). Είναι οι δύο συναρτήσεις για τον έλεγχο του ταυτοχρονισμού με τη βάση την λύση συγχρονισμού αναγνωστών-γραφέων. Για αυτές τις συναρτήσεις χρησιμοποιήθηκε ο αλγόριθμος από το βιβλίο Σύγχρονα Λειτουργικά Συστήματα του Tanenbaum

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                     /* repeat forever */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc + 1;                  /* one reader more now */
          if (rc == 1) down(&db);       /* if this is the first reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          read_data_base( );            /* access the data */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc – 1;                  /* one reader fewer now */
          if (rc == 0) up(&db);         /* if this is the last reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          use_data_read( );             /* noncritical region */
     }
}
```

```
void writer(void)
{
        while (TRUE) {                  /* repeat forever */
                think_up_data( );       /* noncritical region */
                down(&db);              /* get exclusive access */
                write_data_base( );     /* update the data */
                up(&db);                /* release exclusive access */
        }
}
```

```
void readerr(Request *request,char response_str[BUF_SIZE])
{


        pthread_mutex_lock(&reader_count_mutex);
   reader_count++;
   if(reader_count==1)
        pthread_mutex_lock(&db_mutex);
   pthread_mutex_unlock(&reader_count_mutex);
   if (KISSDB_get(db, request->key, request->value))
    sprintf(response_str, "GET ERROR\n");
   else
    sprintf(response_str, "GET OK: %s\n", request->value);

        pthread_mutex_lock(&reader_count_mutex);
   reader_count--;
   if(reader_count==0)
        pthread_mutex_unlock(&db_mutex);
   pthread_mutex_unlock(&reader_count_mutex);
}

void writerr(Request *request, char response_str[BUF_SIZE])
{


        pthread_mutex_lock(&db_mutex);


   if (KISSDB_put(db, request->key, request->value))
    sprintf(response_str, "PUT ERROR\n");
   else
    sprintf(response_str, "PUT OK\n");


        pthread_mutex_unlock(&db_mutex);

}
```

Οι συναρτήσεις readerr(), writerr() καλούνται στην process_request. Η readerr() για την GET και η writerr() για την PUT.

Η συνάρτηση **sig_handler** είναι η συνάρτηση για τον χειρισμό του σήματος CTRL+Z. Όπως γράφουν και τα σχόλια, ο καταναλωτής τερματίζει όλα τα νήματα παραγωγούς, κλείνει τη βάση και υπολογίζει και τυπώνει τα στατιστικά αποτελέσματα.

```
static void sig_handler(int signo)
{


	// termatismos katanalwtwn
	join_threads();

	// kleisimo vasis
	KISSDB_close(db);

	 // Free memory.
	 if (db)
	   free(db);
	 db = NULL;

	// ypologismos kai typwma statistikwn apotelesmatwn
	ypologismos();

	exit(0);
}
```

Η συνάρτηση **join_threads** στέλνει σήμα στα νήματα παραγωγούς για να τερματίσουν και περιμένει κάθε ένα από αυτά να τερματίσει.

```
void join_threads()
{
	int i;
	for(i=0;i<THREADS;i++)
		pthread_kill(tid[i],SIGTSTP); // o paragogos stelnei sima termatismou se olous tous katanalwtes

	for(i=0;i<THREADS;i++)
		pthread_join(tid[i],NULL);
}
```

Η συνάρτηση **ypologismos**, υπολογίζει το μέσο συνολικό χρόνο αναμονής και το μέσο συνολικό χρόνο επεξεργασίας αίτησης και τυπώνει αυτά τα δύο και το συνολικό πλήθος των αιτήσεων που έχουν εξυπηρετηθεί.

```
void ypologismos()
{
	pthread_mutex_lock(&cr);
```

```
	if(completed_requests!=0)
	{
		pthread_mutex_lock(&twt);
		printf(" mesos total_waiting_time %f \n",(double)(total_waiting_time/completed_requests));
		pthread_mutex_unlock(&twt);
		pthread_mutex_lock(&tst);
		printf(" mesos total_service_time : %f \n",(double)(total_service_time/completed_requests));
		pthread_mutex_unlock(&tst);
	}

	printf(" completed_requests: %d \n",completed_requests);
	pthread_mutex_unlock(&cr);
}
```

Η συνάρτηση **enQ** εισάγει μία νέα αίτηση στο tail της ουράς. Η πρόσβαση στην ουρά είναι ελεγχόμενη, μόνο ένα νήμα επιτρέπεται να την προσπελάζει κάθε φορά. Το lock, unlock του Q_mutex που της αντιστοιχεί πραγματοποιείται στη main(). Αν το πλήθος γίνει 1, τότε αυτό σημαίνει ότι πριν ήταν 0, άρα πιθανόν να περιμένουν νήματα-καταναλωτές για αυτό και το νήμα παραγωγού που καλεί την enQ τα ειδοποιεί pthread_cond_signal(&non_empty_Queue).

```
void enQ(int new_connection)
{
	struct timeval tv;

		tail++;
		plithos++;

		gettimeofday(&tv,NULL);
		Q[tail-1].start_time=tv;
		Q[tail-1].connection_fd=new_connection;

		if(tail==QUEUE_SIZE)
		{
			tail=0;  // ksana apo tin arxi
		}

		if(plithos==1) // gia na ginei to plithos 1 , simainei oti prin htan 0
		// yparxei pithanotita na perimenoun nimata stin metavliti sinthikis
		// non_empty_Queue, kai prepei na ta ksipnisoun
		{
			pthread_cond_signal(&non_empty_Queue);
		}



	return;}
```

Η συνάρτηση **deQ** εκτελείται από τα νήματα καταναλωτές. Εξάγει μία αίτηση από το head της ουράς και την επιστρέφει. Η πρόσβαση είναι και εδώ ελεγχόμενη, ένα ένα νήμα τη φορά. Αν η ουρά ήταν γεμάτη πριν την εξαγωγή μιας αίτησης, τότε είναι πιθανόν το νήμα παραγωγός να έχει μποκάρει. Οπότε ειδοποιείται για να ξεμπλοκάρει και να συνεχίσει τη λειτουργία του , πιθανόν με την εισαγωγή νέων αιτήσεων στην ουρά.

```
struct oura * deQ()
{
	struct oura * x = NULL;

	pthread_mutex_lock(&Q_mutex);

	while(plithos == 0)
		pthread_cond_wait(&non_empty_Queue,&Q_mutex);

	head++; // h head proxoraei parakatw
	plithos--; // to plithos twn stoixeiwn tis ouras mwiwnetai
	x = &Q[head-1];
	// kikliki arithmisi, opote an h head exei ftasei sto QUEUE_SIZE
	// tin girnaw sto 0
	if(head==QUEUE_SIZE)
		head=0;

	// prin htan gemati kai tora adeiase mia thesi
	// opote mporei na mpei nea aitisi, gia auto prepei na ksipnisei to nima
	// paragwgou (an koimatai)
	if(plithos==QUEUE_SIZE-1)
		pthread_cond_signal(&non_full_Queue);

	pthread_mutex_unlock(&Q_mutex);

	return x;
}
```

Η συνάρτηση που εκτελούν τα νήματα καταναλωτές

```
void *katanalotis(void  *x)
{
	struct oura * xx;

	while(1){
		xx=deQ();
		if(xx!=NULL)
			process_request(xx->connection_fd,xx->start_time);
	}
}
```

Και η τροποποίηση που έγινε στη main

Το νήμα παραγωγός περιμένει όσο η ουρά είναι γεμάτη. Όταν κάποιο νήμα καταναλωτής εξάγει ένα στοιχείο , ειδοποιεί τον παραγωγό και αυτός συνεχίζει παρακάτω: δέχεται τη νέα αίτηση, την εισάγει στην ουρά και ξανά από την αρχή.

```
while (1) {


 // an h oura einai gemati tote to nima paragwgou prepei na perimenei
  pthread_mutex_lock(&Q_mutex);
  while(plithos==QUEUE_SIZE)
          pthread_cond_wait(&non_full_Queue, &Q_mutex);
  pthread_mutex_unlock(&Q_mutex);

   // wait for incomming connection
   if ((new_fd = accept(socket_fd, (struct sockaddr *)&client_addr, &clen)) == -1) {
    ERROR("accept()");
   }

        fprintf(stderr, "(Info) main: Got connection from '%s'\n",
inet_ntoa(client_addr.sin_addr));
        pthread_mutex_lock(&Q_mutex);
   enQ(new_fd); // eisagogi neas aitisis stin oura
        pthread_mutex_unlock(&Q_mutex);
 }
```

Client.c

Δημιουργούνται ένα σύνολο από αιτήσεις (έγινε αλλαγή από 128 σε 64 για πιο εύκολες δοκιμές). Κάθε μία από τις αιτήσεις , την αναλαμβάνει μία νέα διεργασία. Το sleep(3) μπήκε γιατί η rand() έβγαζε συνεχώς τους ίδιους τυχαίους αριθμούς, μήπως και σταματήσει να το κάνει αυτό.

```
while(--count>=0) {
    for (station = 0; station < MAX_STATION_ID; station++) {
          // gia kathe mia apo tis aitiseis tha dimiourgiso mia
          // ksexoristi diergasia pou tha tin analavei
                pid[station] = fork();
                if(pid[station] == 0 ) //child code
                {
                        memset(snd_buffer, 0, BUF_SIZE);
      if (mode == GET_MODE) {
       // Repeatedly GET.
       sprintf(snd_buffer, "GET:station.%d", station);
      } else if (mode == PUT_MODE) {
       // Repeatedly PUT.
       // create a random value.
            sleep(3);
       value = rand() % 65 + (-20);
       sprintf(snd_buffer, "PUT:station.%d:%d", station, value);
```

```
    }
    printf("Operation: %s from process %d\n", snd_buffer, getpid());
    talk(server_addr, snd_buffer);
        exit(0); // termatismos diergasias
                }
                else if (pid[station]>0) // parent code
                {
                        ;
                        //printf("Child %d dimiourgithike - aitisi aitimh gia apostolh
\n",pid[station]);
                }

    }
```

Αποτελέσματα

Για τα παρακάτω πειράματα δόθηκαν με τη σειρά οι εξής εντολές:
Σε ένα τερματικό:

- make all
- ./server >> s1.txt

Σε ένα δεύτερο τερματικό:

- ./client -a localhost -i 1 -p >> c11.txt
- ./client -a localhost -i 1 -g >> c12.txt
- ./client -a localhost -o PUT:station.40:5 >> c13.txt
- ./client -a localhost -o PUT:station.30:23 >> c13.txt
- ./client -a localhost -o GET:station.30 >> c14.txt
- ./client -a localhost -o GET:station.40 >> c14.txt

Τα αρχεία αυτά περιέχουν την έξοδο και τα στέλνουμε για την απόδειξη της ορθότητας της λειτουργίας του προγράμματος. Τα αρχεία αφορούν την $1^{η}$ περίπτωση ελέγχου για μέγεθος ουράς 10 και πλήθος νημάτων 5. Τα υπόλοιπα πειράματα πραγματοποιήθηκαν χωρίς «>> όνομα αρχείου»

Queue size	Νήματα	Μέσος χρόνος αναμονής σε mimcros	Μέσος χρόνος εξυπηρέτησης σε mimcros
10	5	39	308
20	5	254	250
10	10	40	238
20	10	580	210
10	20	229	258
20	20	10	321