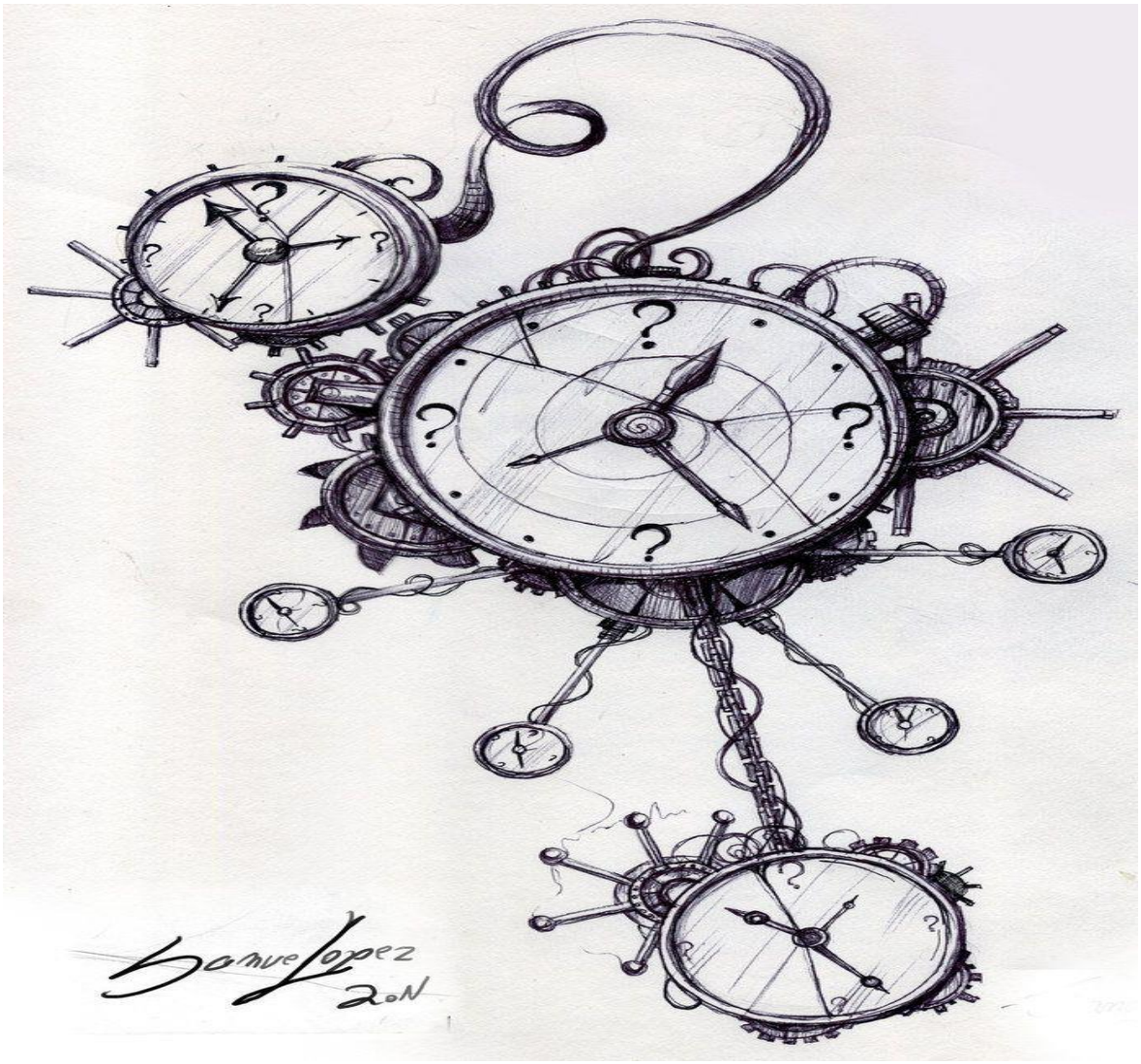
Ενσωματωμένα Συστήματα Πραγματικού Χρόνου

Αναφορά απαλλάκτικής εργασίας



Ο κώδικας υλοποιήθηκε στην c

Η πλατφόρμα που τεσταρίστηκε το raspberry pi 3

Δημήτρης Μουτζόγλου, 8319

Περιγραφή Κώδικα:

Σκοπός της εργασίας ήταν η δημιουργία timer objects τα οποία θα προκαλούν την εκτέλεση μίας συνάρτησης με δεδομένη χρονική περίοδο με τη βοήθεια νημάτων **producer-consumer** που υλοποιήσαμε στην προηγούμενη εργασία. Η απαραίτητη χρονική υστέρηση για την τήρηση της περιόδου επιτυγχάνεται με την χρήση της συνάρτησης usleep().

Περιγραφή νέων Συναρτήσεων και Δομών:

- **struct timer**: περιέχει όλες τις απαραίτητες μεταβλητές για την υλοποίηση του timer όπως η περίοδος , ο αριθμός εκτελέσεων , η αρχική καθυστέρηση , η εκτελούμενη συνάρτηση και δείκτες στην ουρά, τα δεδομένα του χρήστη.
- **timer *TimeInit()**: συνάρτηση που δημιουργήθηκε για την αυτοματοποίηση της διαδικασίας της αρχικοποίησης των timer structs.
- **void start()**: δέχεται ως όρισμα έναν δείκτη σε δομή τύπου timer και ουσιαστικά δημιουργεί ένα νήμα producer με τα χαρακτηριστικά λειτουργίας (περίοδο,εκτελούμενη συνάρτηση κτλ) που περιγράφει η δομή timer.
- **void startat()**: δέχεται ως όρισματα την ημερομηνία, χρονολογία και ώρα με ακρίβεια δευτερολέπτου και είναι υπεύθυνη για την εκκίνηση του timer σύμφωνα με τα ορίσματα της. Στην παραπάνω λειτουργία συμμετέχει η συνάρτηση difftime() που υπολογίζει τον αριθμό των δευτερολέπτων που μεσολαβούν ανάμεσα στην χρονική στιγμή που τρέχει ο κώδικας και της μελλοντικής στιγμής που θέλουμε να εκκινήσει το ενσωματωμένο.
- **void StartFcn()**: εκτελείται με το πέρας της αρχικής χρονικής υστέρησης σε κάθε νήμα producer και εκτυπώνει μήνυμα με το οποίο σηματοδοτεί την έναρξη περιοδικής λειτουργίας.
- **void StopFcn()**: Με την ολοκλήρωση του ζητούμενου αριθμού εκτελέσεων , ελέγχεται η τιμή clear1 που λειτουργεί ως σημαία και για τιμή 1 εκτυπώνεται μήνυμα αποπεράτωσης της εργασίας του producer.
- **void ErrorFcn()**: Συνάρτηση που καλείται κάθε φορά που ο producer αποτυγχάνει να βρει θέση στην ουρά για την αποθήκευση της διεργασίας του και διατηρεί τον αριθμό των αποτυχιών σε μία global μεταβλητή.

- **void dpfiler()**: Δέχεται ως όρισμα έναν δείκτη σε δομή timer και είναι υπεύθυνη για την αποθήκευση των μετρήσεων για την ολίσθηση και τον χρόνο που απαιτούσε η προσθήκη μιας εργασίας στην ουρά από τον producer.
- **void cfiler()**: Συνάρτηση που αποθηκεύει τον χρόνο παραμονής κάθε εργασίας στην ουρά μέχρι την στιγμή της εκτέλεσής της.

Συνοπτική ροή κώδικα:

Στην **main**() δημιουργούμε πίνακες δεδομένων τύπου p_thread και μεγέθους ίσου με τις constant τιμές που έχουμε ορίσει. Στην συνέχεια δημιουργούμε και αρχικοποιούμε μία δομή που υλοποιεί την ουρά όπου και αποθηκεύονται προσωρινά οι δείκτες των συναρτήσεων μέχρις εκτέλεσής τους. Ακολουθεί η δημιουργία timer δομών με την χρήση της συνάρτησης TimeInit() και η δέσμευση χώρου με την malloc() για την αποθήκευση των χρονικών διαστημάτων που απαιτεί το πλήθος των consumer εως ότου αφαιρέσουν μια εργασία από την ουρά. Σειρά έχει η δημιουργία των νημάτων consumer με pthread_create και νημάτων producer με την χρήση της start(). Με την εκτέλεση του ζητούμενου πλήθος εργασιών τα νήματα producer και consumer παύουν με την χρήση της pthread_join. Η main κλείνει με την αποθήκευση των μετρήσεων με την χρήση των dpfiler,cfiler ,την διαγραφή της ουράς και την απελευθέρωση του χώρου που καταλαμβάνουν οι δομές και ο πίνακας claps.

Το υπολειπόμμενο μέρος του προγράμματος αντιστοιχεί σε μεγάλο βαθμό με την υλοποίηση της προηγούμενης εργασίας οπότε θα γίνει ειδική μνεία στα ζητούμενα της εργασίας και τα διαφοροποιημένα κομμάτια.

Στην συνάρτηση **producer** που υλοποιεί και την λειτουργικότητα του νήματος δημιουργούμε δομές τύπου timeval για την παρακολούθηση της περιόδου με την χρήση της συνάρτησης gettimeofday() και τις μεταβλητές τύπου drift , sent που θα χρησιμοποιήσουμε για την καταγραφή των μετρήσεων. Έπειτα από την αρχική ζητούμενη χρονική υστέρηση μπαίνουμε σε μία επανάληψη που θα «τρέχει» μέχρι την τέλεση του πλήθους των ζητούμενων εργασιών. **Πριν** πάρουμε το mutex της ουράς αποθηκεύουμε στην δομή send (τύπου timeval) την χρονική στιγμή , με το πέρας του ελέγχου για πλήρη ουρά και την προσθήκη της εργασίας , ξαναμετράμε την χρονική στιγμή. Η διαφορά των δύο στιγμών αποτελεί τον χρόνο που ορίστηκε ως ο χρόνος που ξοδεύει η producer να βάλει μια εργασία στην ουρά. Στη συνέχεια θα ελευθερώσουμε το mutex και θα προχωρήσουμε στον υπολογισμό της ολίσθησης στον χρόνο. Αυτή ορίζεται ως η διαφορά της τωρινής χρονικής στιγμής (που πήραμε με gettimeofday)

από την προηγούμενη χρονική στιγμή που βρισκόμασταν σε αυτό το σημείο εκτέλεσης μείον της περίοδο που κρατάμε. Έπειτα , θα ορίσουμε το χρονικό διάστημα που το νήμα θα «κοιμηθεί» ως το προηγούμενο χρονικό διάστημα που κοιμήθηκε μείον την υπολογισθείσα ολίσθηση. Με αυτόν τον τρόπο , προσπαθούμε να πετύχουμε παράλληλα και την περιοδική προσθήκη εργασιών στην ουρά με αυξημένη ακρίβεια , όσο και την λειτουργία γύρω από τις χρονικές στιγμές που ένας ιδανικός περιοδικός timer θα επιτύγχανε.

Ως χρόνο εξυπηρέτησης από μεριάς του consumer ορίζουμε την μεταβλητή elapsed που απεικονίζει την διαφορά της χρονικής στιγμής που η queueAdd() θα προσθέσει τον pointer της εκτελούμενης διεργασίας στην ουρά από την χρονική στιγμή που η συνάρτηση θα κληθεί από την ουρά και θα εκτελεστεί από τον consumer.

Πειράματα και σχολιασμοί επί των στατιστικών αποτελεσμάτων:

Ζητούμενο της εργασίας ήταν η εκτέλεση 4 πειραμάτων , όπου το κάθε πείραμα θα διαρκούσε μία ώρα και θα υλοποιούσε έναν timer με περίοδο λειτουργίας 1s, 100ms, 10ms και το τελικό πείραμα θα υλοποιούσε και τους 4 timers ταυτόχρονα. Τα πειράματα εκτελέστηκαν με αμετάβλητο αριθμό consumers (3) προκειμένου να έχουμε «1-1» αντιστοιχία producers-consumers στην ταυτόχρονη λειτουργία και για την ευκολότερη σύγκριση των αποτελεσμάτων των μεμονωμένων timers με της ταυτόχρονης. Τα αποτελέσματα έχουν μονάδα μέτρησης τα microseconds.

Timer Period 10ms:

	Drift	Producer	Consumer
Mean	50.23	1.39	41.02
Median	14	1	32
Min	0	0	13
Max	1004	788	580
Std	75.61	2.5	20.99

Timer Period 100ms:

	Drift	Producer	Consumer
Mean	40.8	1.35	78.93
Median	22	1	68
Min	0	1	30
Max	489	206	776
Std	57.62	1.97	15.28

Timer Period 1s:

	Drift	Producer	Consumer
Mean	85.72	1.83	70.13
Median	30	1	72
Min	0	1	32
Max	602	352	1005
Std	66.76	4.53	17.68

Παρατηρούμε πως η ολίσθηση αυξάνει με την αύξηση της περιόδου και αυτό οφείλεται στην ανακρίβεια της συνάρτησης usleep() ειδικότερα για μεγάλους χρόνους αλλά από την ένδειξη της διάμεσου της ολίσθησης μπορούμε να συμπεράνουμε πως εν κατακλείδι αυτή διατηρείται σε χαμηλά επίπεδα ως προς την τηρούμενη περίοδο. Τα αποτελέσματα της producer είναι αναμενόμενα καθότι χρησιμοποιούμε buffer μεγέθους 10 θέσεων και σπαταλάται ελάχιστος χρόνος μέχρι η queueAdd() να προσθέσει εργασία στην ουρά. Τέλος , ο consumer παρουσιάζει μεγαλύτερες τιμές εξυπηρέτησης για μεγαλύτερες συχνότητες , γεγονός που ίσως οφείλεται στην αδράνεια του συστήματος για μεγαλύτερα χρονικά διαστήματα.

Ταυτόχρονη λειτουργία (10ms,100ms,1s)

	Drift	Producer	Consumer
Mean	91.52	6.74	71.56
Median	32	2	42
Min	0	0	2
Max	12427	16301	1743
Std	208	70.39	21.8

Για την ταυτόχρονη λειτουργία των timers , οι τιμές των producers και των drifts υπολογίστηκαν από κοινού για να έχουμε πληρέστερη εποπτεία. Παρατηρούμε αρχικά την αύξηση στους μέσους όρους όλων των μετρικών.Η αύξηση της ολίσθησης οφείλεται στο ότι η usleep() και η ανακρίβεια της επιδρά πλέον (360000 + 36000 + 3600) φορές με διαφορετικές μάλιστα τιμές περιόδων. Χαρακτηριστική είναι η αύξηση του χρόνου που σπαταλά ο producer έως ότου αποθηκεύσει μία εργασία στην στοίβα, φαινόμενου που πιθανώς οφείλεται στην διαδικασία racing στην οποία επιδίδονται τα νήματα για το mutex , με αποτέλεσμα αυξημένες συγκρούσεις και καθυστερήσεις. Ο consumer από την άλλη δεν σημειώνει μεγάλη μεταβολή σε σχέση με τις μεμονωμένες εκτελέσεις των timers , φαινόμενο που πιθανώς να οφείλεται ότι για σταθερό αριθμό consumer threads , έχουμε μικρή αύξηση του φόρτου σε σχέση με την λειτουργία του timer για 10ms.

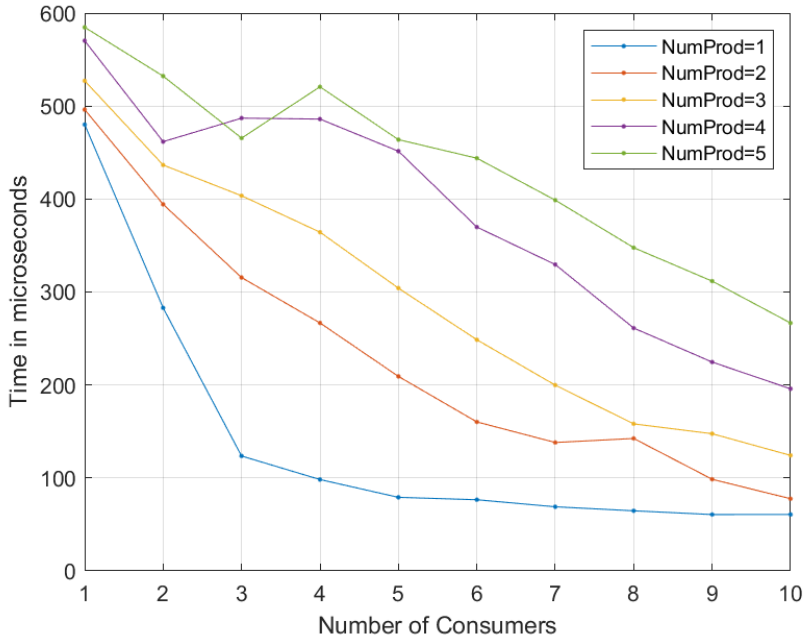
Κατανάλωση Ισχύος

Για την εύρεση της κατανάλωσης ισχύος χρησιμοποιήθηκε η εντολή time ./ .

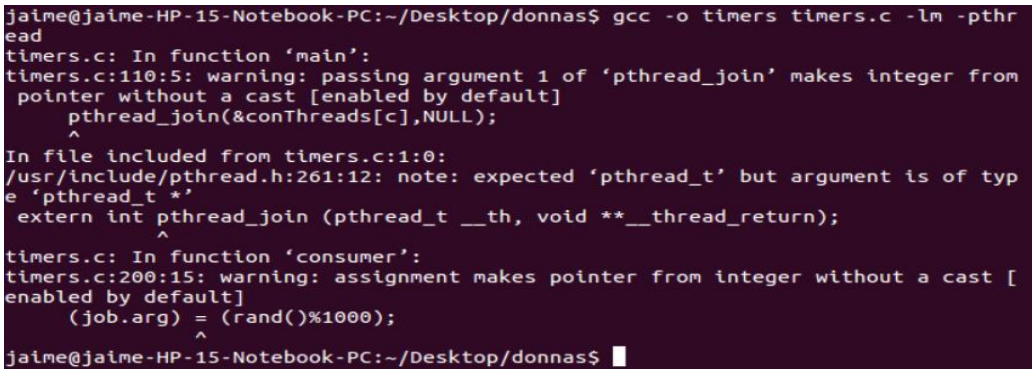
	Cpu usage
10ms	0,85%
100ms	0,086%
1s	0,001%
10ms/100ms/1s	0,883%

Από την ανάγνωση των αποτελεσμάτων συμπεραίνουμε πως όσο μικρότερη είναι η περίοδος , τόσο μεγαλύτερη είναι η κατανάλωση καθώς το σύστημα παραμένει αδρανές για μικρότερο χρονικό διάστημα ενώ και το πλήθος των εκτελούμενων συναρτήσεων είναι 10 φορές μικρότερο για περίοδο 100ms και 100 φορές για περίοδο 1s (σε σχέση με λειτουργία περιόδου 10ms). Για την ταυτόχρονη λειτουργία , έχουμε τροπον τινά το άθροισμα των καταναλώσεων των επιμέρους timers , γεγονός που είναι λογικό καθώς πρόκειται για το άθροισμα των φόρτων εργασίας τους.

Με την εφαρμογή μικρών σε διάρκεια και σύμφωνα με τα συμπεράσματα της προηγούμενης εργασίας καταλήγουμε πως για 3 νήματα producers , 3 νήματα consumers είναι το σημείο μετά το γόνατο της καμπύλης που εξυπηρετεί κατά το δυνατό αποτελεσματικότερα τον φόρτο.



Υποσημείωση: ο gc compiler εμφανίζει μερικές προειδοποιήσεις κατά το compiling. Η επίλυση τους οδηγεί σε segmentation faults , core dumped. Με την διατήρηση του κώδικα που παράγει warnings αυτός παραμένει λειτουργικός.



Πλήρης Κώδικας:

```
#include <pthread.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <math.h>

#include <sys/time.h>

#include <time.h>


#define NumPro 1

#define NumCon 3

#define QUEUESIZE 10


#define PI 3.1415926535
```

```
void *producer (void *args);
void *consumer (void *args);

void *divider(void *p);
void *siner(void *p);
void *printer(void *p);

typedef void * (*work)(void *);

typedef struct workFunction {
  void * (*work)(void *);
  void * arg;
  struct timeval tv;
}workFunction;

work functions[3] = {divider,siner,printer};

typedef struct {
  long head, tail;
  int full, empty;
  workFunction buff[QUEUESIZE];
  pthread_mutex_t *mut;
  pthread_cond_t *notFull, *notEmpty;
} queue;

queue *queueInit (void);
```

```
void queueDelete (queue *q);
void queueAdd (queue *q, struct workFunction in);
void queueDel (queue *q, struct workFunction *out);

typedef struct {
  pthread_t tid;
  int TimerPeriod;
  unsigned int TasksToExecute;
  unsigned int StartDelay;
  work TimerFcn;
  int *UserData;
  queue *thequeue;
  int *calc1,*calc2;
} timer;

timer *TimeInit (int Period, unsigned int Tasks, unsigned int Delay, void * (*work)(void
*), queue *q, pthread_t id);
void start(timer *t1);
void StartFcn (long p);
int ErrorFcn (long p);
void StopFcn (long p);
void dpfiler (timer *t);
void cfiler(unsigned int totaltasks);

void *function();
int clear1=0;
```

```
int ErrCount = 0;
double *claps;
int k=0;



int main(){

   pthread_t proThreads[NumPro];
   pthread_t conThreads[NumCon];

   queue *fifo;
   fifo = queueInit ();
   if (fifo ==  NULL) {
     fprintf (stderr, "main: Queue Init failed.\n");
     exit (1);
   }

   timer *t1,*t2,*t3;
   t1 = TimeInit(10,360000,1,divider,fifo,proThreads[0]);
   t2 = TimeInit(100,36000,1,siner,fifo,proThreads[1]);
   t3 = TimeInit(1000,3600,1,printer,fifo,proThreads[2]);
   claps = (double *) malloc(sizeof(double)*(t1->TasksToExecute+ t2->TasksToExecute+
t3->TasksToExecute));

   long c;
   for(c=0; c<NumCon;c++){
```

```
  printf("In main: creating consumer thread %ld\n",c);
  pthread_create (&conThreads[c], NULL, consumer, fifo);
 }

 start(t1);
 start(t2);
 start(t3);

 pthread_join(t1->tid,NULL);
 pthread_join(t2->tid,NULL);
 pthread_join(t3->tid,NULL);
 printf("Done joining the pro threads\n");

 dpfiler(t1); // kanonika katw apo join twn con threads
 dpfiler(t2);
 dpfiler(t3);
 cfiler(t1->TasksToExecute+ t2->TasksToExecute+ t3->TasksToExecute); //+ t2-
>TasksToExecute

 for (c=0;c<NumCon;c++){
 pthread_join(&conThreads[c],NULL);
 }
 queueDelete (fifo);

 free(t1);
 free(t2);
```

```
  free(t3);


  free(claps);
  printf("Total Number of Errors %d\n",ErrCount);
  return 0;
}

void *producer (void *q)
{

 timer *data = (timer *)q;
 queue *fifo;
 workFunction job;
 struct timeval tv,send;
 fifo = data->thequeue;
 job.work=data->TimerFcn;
 StartFcn (data->tid);
 long double sleeptime = data->TimerPeriod*1000;
 long double last=0;
 long double drift=0;
 long double sent=0;

 usleep(data->StartDelay*1000*1000);
 printf("slept my assigned delay time.\n");

 int i;
```

```
 for (i = 0; i < data->TasksToExecute; i++){
  gettimeofday(&send,NULL); // keeping tabs on time just before we try to store a function pointer
  pthread_mutex_lock (fifo->mut);
  while (fifo->full) {
    ErrorFcn(data->tid);
    pthread_cond_wait (fifo->notFull, fifo->mut);
  }

  queueAdd (fifo, job);

  gettimeofday(&tv,NULL);// keeping tabs on time right after we stored a function pointer
  sent = tv.tv_sec *1000000 + tv.tv_usec;

  pthread_mutex_unlock (fifo->mut);
  pthread_cond_signal (fifo->notEmpty);

  data->calc2[i] =  sent - (send.tv_sec*1000000 + send.tv_usec);

  if (i > 0){
    drift = (tv.tv_sec*1000000+tv.tv_usec -last) - (data->TimerPeriod*1000);
    sleeptime = sleeptime - drift;
    printf("Drift = %Lf.\n",drift);
  }
  last = sent;
```

```
    data->calc1[i] = drift;
    if (sleeptime < 0){
      sleeptime = 0; // enallaktika data_>TimerPeriod
    }

    usleep(sleeptime);

  }
    clear1=1;
    return (NULL);
}

void *consumer (void *q)
{

  queue *fifo;
  fifo = (queue *)q;
  long tid = pthread_self();
  struct timeval feed,beg;
  long elapsed;
  workFunction job;


  while(1) {
    pthread_mutex_lock (fifo->mut);
    while (fifo->empty) {
```

```
  //printf ("consumer: queue EMPTY.\n"); //testing
  pthread_cond_wait (fifo->notEmpty, fifo->mut);
}

job = fifo->buff[fifo->head];

gettimeofday(&beg, NULL);//keeping tabs on time just before execution

(job.arg) = (rand()%1000);
job.work(job.arg);

feed=job.tv;

queueDel (fifo, &job);

pthread_mutex_unlock (fifo->mut);
pthread_cond_signal (fifo->notFull);

if (clear1==1){
  StopFcn(tid); // afotou ektelesei thn teleutaia
}

elapsed = (beg.tv_sec - feed.tv_sec)*1000000 + beg.tv_usec - feed.tv_usec;
*(claps+k)=elapsed;
k++;
```

```
  }
    return (NULL);
}

queue *queueInit (void)
{
  queue *q;

  q = (queue *)malloc (sizeof (queue));
  if (q == NULL) return (NULL);

  q->empty = 1;
  q->full = 0;
  q->head = 0;
  q->tail = 0;
  q->mut = (pthread_mutex_t *) malloc (sizeof (pthread_mutex_t));
  pthread_mutex_init (q->mut, NULL);
  q->notFull = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
  pthread_cond_init (q->notFull, NULL);
  q->notEmpty = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
  pthread_cond_init (q->notEmpty, NULL);

  return (q);
}

void queueDelete (queue *q)
```

```
{
  pthread_mutex_destroy (q->mut);
  free (q->mut);
  pthread_cond_destroy (q->notFull);
  free (q->notFull);
  pthread_cond_destroy (q->notEmpty);
  free (q->notEmpty);
  free (q);
}

void queueAdd (queue *q, struct workFunction in)
{

  gettimeofday(&in.tv,NULL);//keeping tabs on time just before storing our function
pointer in fifo
  q->buff[q->tail] = in;
  q->tail++;
  if (q->tail == QUEUESIZE)
    q->tail = 0;
  if (q->tail == q->head)
    q->full = 1;
  q->empty = 0;

  return;
}
```

```
void queueDel (queue *q, struct workFunction *out)
{

  *out = q->buff[q->head];

  q->head++;
  if (q->head == QUEUESIZE)
    q->head = 0;
  if (q->head == q->tail)
    q->empty = 1;
  q->full = 0;

  return;
}

timer *TimeInit (int Period, unsigned int Tasks, unsigned int Delay, void * (*work)(void
*), queue *q, pthread_t id)
{
    timer *t;
    t = (timer *)malloc (sizeof (timer));
    if (t == NULL) return (NULL);

    t->TimerPeriod = Period;
    t->TasksToExecute = Tasks;
    t->StartDelay = Delay;
    t->TimerFcn = work;
```

```
    t->thequeue=(queue *)q;
    t->tid = id;
    int *c1 = (int *) malloc(sizeof(int)*Tasks);
    int *c2 = (int *) malloc(sizeof(int)*Tasks);
    int i = 0;
    for (i=0; i<Tasks;i++){
      c1[i]=0;
      c2[i]=0;
    }
    t->calc1 = c1;
    t->calc2 = c2;
    return (t);

}

void start (timer *t1){
  pthread_create(&(t1->tid),NULL,producer,t1);
}

void StartFcn (long p){
  printf ("producer %ld: Work,Work,Work just like Rihanna.\n",p);
}

int ErrorFcn (long p){
  printf ("producer %ld: queue FULL.\n",p);
  ErrCount++;
```

```
  return 0;
}

void StopFcn (long p){
  printf("producer %ld: My job here is done.\n",p);
  clear1=0;
}

void startat(timer *t1,int y,int m,int d,int h,int min,int sec){

  time_t futt;
  struct tm fut;
  fut.tm_year = y - 1900;
  fut.tm_mon = m - 1;
  fut.tm_mday = d;
  fut.tm_hour = h;
  fut.tm_min = min;
  fut.tm_sec = sec;
  fut.tm_isdst = -1;
  futt = mktime(&fut);

  time_t now;

  time(&now);
  printf("Difference is %.2f seconds",difftime(futt,now));
```

```
  sleep(difftime(futt,now)-120); //sleeps for the bulk of time in seconds apart from a whole minute to compensate for sleep function inaccuracy and another for our time delay
  time(&now);
  sleep(difftime(futt,now)-60); //sleeps for the rest of time in seconds except for a minute that corresponds to our delay
  pthread_create(&(t1->tid),NULL,producer,t1);

}

void *divider(void *p){
  if ((long int)p % 3 == 0){
  printf("The long integer %ld is a multiple of 3.\n",(long int)p);
  }
  else {
  printf ("The long integer %ld is not a multiple of 3.\n",(long int)p);
}

}

void *siner(void *p){
  double ret, val;
  long int x = ( long int) p;
  val = PI/180;
  ret = sin(val* (double) x);
  printf("The sine value of %ld is %lf degrees\n",x,ret);
}
```

```
void *printer(void *p){
  printf("Printer called and printer printed a random integer %ld.\n",(long int)p);
}

void dpfiler (timer *t){
  char namea[20];
  printf("Enter the name of the first file to be created:\n");
  fgets(namea,30,stdin);
  FILE *fptra = fopen(namea,"a");

  char nameb[20];
  printf("Enter the name of the second file to be created:\n");
  fgets(nameb,30,stdin);
  FILE *fptrb = fopen(nameb,"a");

  int j;
  for (j=0;j<t->TasksToExecute;j++){
    fprintf(fptra,"%d\n",t->calc1[j]);
    fprintf(fptrb,"%d\n",t->calc2[j]);

  }

  fclose(fptra);
  fclose(fptrb);
  printf("Couple of drift and producer files created\n");
```

```
  free(t->calc1);
  free(t->calc2);
}

void cfiler(unsigned int totaltasks){
  char name[20];
  printf("Enter the name of the consumer file to be created:\n");
  fgets(name,30,stdin);
  FILE *fptr = fopen(name,"a");
  int j;
  for (j=0;j<totaltasks;j++){
    fprintf(fptr,"%lf\n",claps[j]);
  }
  fclose(fptr);
  printf("Total consumer file created\n");
}
```