

# Producer Consumer Pthreads Implementation in C with Timer

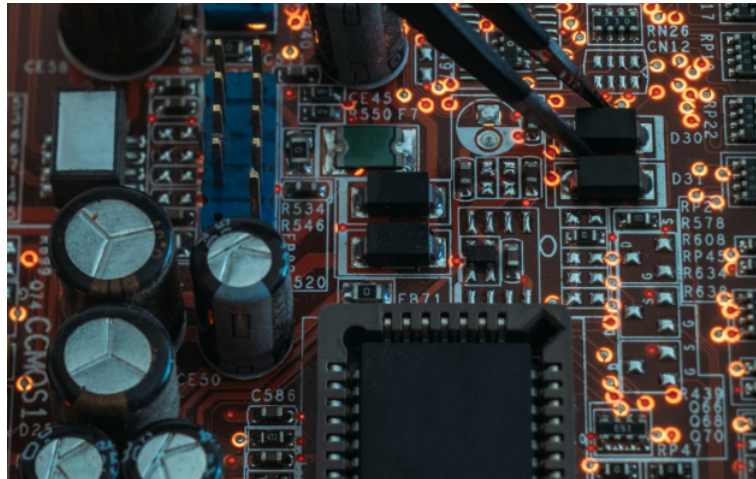
Ανδρόνικος Κώστας

Ενσωματωμένα Συστήματα Πραγματικού Χρόνου  
Assignment II

akostasp@ece.auth.gr

**9754**

[Github](#)



## Περιεχόμενα

<b>1</b>	<b>Εισαγωγή</b>	<b>4</b>
<b>2</b>	<b>Ανάλυση Κώδικα</b>	<b>5</b>
2.1	Timer Struct . . . . .	5
2.2	Timer Initialization . . . . .	6
2.3	startAt Function . . . . .	7
2.4	Producer and Consumer Logic . . . . .	8
2.5	main logic . . . . .	10
<b>3</b>	<b>Technical Information</b>	<b>11</b>
3.1	Hardware Set up . . . . .	11
3.2	Running the .c in Raspberry . . . . .	12
<b>4</b>	<b>Ανάλυση και Σχολιασμός Δεδομένων</b>	<b>13</b>
4.1	Drifting . . . . .	13
4.1.1	Plots . . . . .	13
4.1.2	Σχολιασμός Αποτελεσμάτων για το drifting . . . . .	15
4.1.3	Επίλυση του Drifting . . . . .	16
4.2	Χρόνοι εκτέλεσης της queue add και της queue del . . . . .	16
4.2.1	Σχολιασμός Πινάκων . . . . .	16
4.3	Λειτουργία Πραγματικού Χρόνου . . . . .	17

## Κατάλογος Σχημάτων

1	Timer Struct . . . . .	5
2	Initialization of the timer struct . . . . .	6
3	The logic of ending the experiment . . . . .	6
4	startAt Function . . . . .	7
5	Producer Function Part I . . . . .	8
6	Producer Function Part II . . . . .	9
7	Consumer Function . . . . .	9
8	main Function, mode = ISOLATED . . . . .	10
9	Raspberry Hardware Set Up . . . . .	11
10	Commands to build and run how the .c in raspberry . . . . .	12
11	Period drifting at 1s . . . . .	13
12	Period drifting at 0.1s . . . . .	14

13	Period drifting at 0.01s . . . . .	15
14	Drifting Solution Logic . . . . .	16

## Κατάλογος Πινάκων

1	Outliers percentages for each timer . . . . .	15
2	Statistical Timing Calculations of queueadd() . . . . .	16
3	Statistical Timing Calculations of queuedel() . . . . .	16

# 1 Εισαγωγή

Στην παρούσα αναφορά θα υλοποιηθεί ένας timer που θα τοποθετεί δείκτες συναρτήσεων ανά τακτά χρονικά διαστήματα μέσα σε μια ουρά FIFO. Οι δείκτες θα τοποθετούνται μέσα στην ουρά μέσω της συνάρτησης producer και θα εκτελούνται και θα βγαίνουν από την ουρά μέσω της συνάρτησης consumer. Οι δύο αυτές συναρτήσεις θα τρέχουν παράλληλα ως διαφορετικά νήματα μέσω της βιβλιοθήκης pthread. Σκοπός της εργασίας είναι η υλοποίηση τριών timer με περιόδους 1sec, 0.1sec 0.01sec οι οποίοι θα μπορούν να τρέχουν ξεχωριστά αλλά και παράλληλα. Όσον αφορά τις προσομοιώσεις θα παρουσιαστούν πίνακες και γραφήματα αναφορικά με το drifting, και τους χρόνους εκτέλεσης των συναρτήσεων που απαιτείται για να βάλουν ή να βγάζουν από την ουρά οι συναρτήσεις producer και consumer αντίστοιχα. Τέλος, θα παρατηρηθεί και θα σημειωθεί η χρήση της CPU σε κάθε πείραμα που θα διεξαχθεί. Ο μικροελεγκτής που χρησιμοποιήθηκε ήταν το Raspberry Pi Zero . Το visualization των δεδομένων από τα πειράματα πραγματοποιήθηκε με χρήση Matlab και τα δεδομένα εισόδου ήταν .txt αρχεία που δημιουργήθηκαν κατάλληλα στο κώδικα με χρήση timestamps.

## 2 Ανάλυση Κώδικα

### 2.1 Timer Struct

```
typedef struct
{
    int period;           // time between the calls of TimerFcn (ms)
    int tasksToExecute; // how many times the TimerFcn will be executed
    int startDelay;       // how many seconds will be passed before the first start of TimerFcn occur
    // all the four below variables are pointers that point to functions (StartFunc, StopFunc, TimerFunc, ErrorFunc)
    void (*startFunc)(void *arg); // function pointer //
    //
    void (*stopFunc)(void *arg); // function pointer //
    //
    void (*timerFunc)(void *arg); // function pointer //
    //
    void (*errorFunc)(void *arg); // function pointer //
    //
    queue *fifo;
    //
    workFunctionData timer_func;
    // necessary for the pthread_create() function //
    pthread_t thread_id;
    //
    void (*producer)(void *arg);
} Timer;
```

Figure 1: Timer Struct

Ο timer δημιουργήθηκε με την μορφή ενός [ Fig. 1 ] που περιείχε μεταβλητές όπως η περίοδος με την οποία ο producer βάζει items μέσα στην ουρά, function pointers που δείχνουν σε συναρτήσεις όπως η συνάρτηση που θα εκτελείται από τον consumer και τέλος κάποια στοιχεία αναγκαία για το νήμα του producer όπως μια μεταβλητή τύπου pthread αλλά και ένας function pointer που θα δείχνει στην συνάρτηση producer(). Ακόμη περιέχει και δύο struct, την ουρά τύπου queue και την timer\_func τύπου workFunctionData που υλοποιήθηκαν στην προηγούμενη εργασία.

## 2.2 Timer Initialization

Η αρχικοποίηση του timer struct και η σύνδεση των μεταβλητών με το υπόλοιπο πρόγραμμα πραγματοποιείται μέσω της `init()` [ Fig. 2 ] που επιστρέφει έναν δείκτη τύπου `Timer` . Μέσω της `init` υλοποιείται και ο τρόπος με τον οποίο πετυχαίνουμε συγκεκριμένη διάρκεια πειράματος μιας ώρας μέσω των μεταβλητών `TASKS_TO_EXECUTE` και `DONE_TASKS` που αποτελεί τον τρόπο τερματισμού της `while` της συνάρτησης `consumer` [ Fig. 3 ].

```
/// @brief initialization of the timer struct
/// @param fifo pointer that points to the queue struct
/// @param period time between consecutive calls of the TimerFunc
/// @param startDelay delay before the first execution of the TimerFunc
/// @return Timer
Timer *init(queue *fifo, int period, int startDelay)
{
    Timer *t = (Timer *)malloc(sizeof(Timer));
    t->period = period;
    // tasks to execute calculation based on how many seconds we want to run the timer //
    float temp = (SECONDS / (period * 0.001));
    t->tasksToExecute = (long long int)temp ;
    printf("tasksToExecute : %d \n", t->tasksToExecute);
    TASKS_TO_EXECUTE = t->tasksToExecute;
    printf("time to run : %d sec \n", SECONDS);
    t->startDelay = startDelay;
    t->startFunc = StartFunc;
    t->stopFunc = StopFunc;
    t->errorFunc = ErrorFunc;
    t->timerFunc = TimerFunc;
    t->fifo = fifo;
    t->timer_func.work = TimerFunc;
    t->timer_func.arg = NULL;
    t->producer = producer;
    return t;
}
```

Figure 2: Initialization of the timer struct

```
while (DONE_TASKS < TASKS_TO_EXECUTE)
{
    // lock the mut of the fifo to provide access to onlu one thread at a time in fifo //
    pthread_mutex_lock(fifo->mut);
    while (fifo->empty)
```

Figure 3: The logic of ending the experiment

## 2.3 startAt Function

Η συγκεκριμένη συνάρτηση [ Fig. 4 ] δημιουργεί ένα νήμα producer και άρα τρέχει την συνάρτηση producer σε μία πολύ συγκεκριμένη χρονική στιγμή με ακρίβεια δευτερολέπτων. Για παράδειγμα, η συνάρτηση αυτή είναι πολύ χρήσιμη αν θέλουμε να ξεκινήσουμε τα πειράματά μας όχι τώρα αλλά μετά από λίγο.

```
/// @brief function that calculates the delay before the first execution of TimerFcn happens given the d/m/y h:min:sec
/// @param t pointer that points in Timer struct
/// @param year year to start
/// @param month month to start (1-12)
/// @param day day to start (1-31)
/// @param hour hour to start (0-23)
/// @param min min to start (0-59)
/// @param sec sec to start (0-59)
void startAt(Timer *t, int year, int month, int day, int hour, int min, int sec)
{
    struct tm time_info;
    time_t start_time, current_time;
    time_info.tm_year = year - 1900; // years since 1900
    time_info.tm_mon = month - 1;    // Months from 0 to 11
    time_info.tm_mday = day;         // Day of the month (1-31)
    time_info.tm_hour = hour;        // Hours (0-23)
    time_info.tm_min = min;          // Minutes (0-59)
    time_info.tm_sec = sec;          // Seconds (0-59)
    // SECONDS BEFORE THE PRODUCER STARTS //
    // convert the time info structure into a time_t value
    start_time = mktime(&time_info);
    printf("start_time : %s \n", ctime(&start_time));
    time(&current_time);
    // calculate the seconds that are remaining //
    double seconds_remaning = difftime(start_time, current_time);
    // pass this value to the Timer startDelay variable
    t->startDelay = (int)seconds_remaning;
    // thread creation //
    usleep(t->startDelay);
    pthread_create(&t->thread_id, NULL, t->producer, t);
}
```

Figure 4: startAt Function

## 2.4 Producer and Consumer Logic

```
/// @brief the producer function puts function pointers to the FIFO if it is not full
/// @param q
/// @return
void *producer(void *q)
{
    // in : instance of workfunction struct //
    workfunctionData in;
    // necessary typecast that connects with the pthread_create() in which the *arg
    // It is the pointer to the argument that will be passed to the producer function
    // Also we need the *t to connect the two fifos with each other
    timer *t = (timer *)q;
    // make the two pointers to look at the same address // basically make the two fifos to be the same //
    queue *fifo;
    fifo = t->fifo;
    // here the producer starts to put tasks at the FIFO //
    for (int i = 0; i < t->tasksToExecute; i++)
    {
        // store the current time in the start_time struct //
        gettimeofday(&start_time_s, NULL);
        // timestamps for each call of the producer //
        producer_timestamps_period[i] = start_time_s.tv_sec * 1000000 + start_time_s.tv_usec;
        // lock the mutex of the fifo to provide access to only one thread at a time in fifo //
        pthread_mutex_lock(fifo->mutex);
        printf("task to execute = %d \n", i);
        while (fifo->full)
        {
            printf("producer: queue FULL.\n");
            pthread_cond_wait(fifo->notFull, fifo->mutex); // block the producer thread until the 'notFull' condition variable
                                                         // is signaled by the consumer thread to inform the producer that
                                                         // can continue put tasks in the queue. While waiting the thread
                                                         // automatically releases the mutex, allowing other threads to
                                                         // access the critical section if needed
        }
    }
}
```

Figure 5: Producer Function Part I

Η producer τοποθετεί μέσα στην ουρά δείκτες συναρτήσεων που θα κάνει dequeue και θα εκτελεί η consumer() όπως περιγράψαμε και στην πρώτη εργασία. Τοποθετεί μόνο όταν η ουρά δεν είναι γεμάτη, και όσο είναι γεμάτη το producer thread γίνεται blocked μέσω της notFull mutex μεταβλητής. Στην παρούσα υλοποίηση όμως πρέπει να ληφθεί υπόψη και ο timer που περιγράφηκε παραπάνω. Συγκεκριμένα, όπως φαίνεται και στο [ Fig. 5] οι δείκτες των δύο ουρών (της main και του timer) πρέπει να δείχνουν στην ίδια διεύθυνση. Αυτό επιτυγχάνεται μέσω typecasting του δείκτη του ορίσματος σε δείκτη τύπου Timer ώστε να αποκτηθεί πρόσβαση στη queue του Timer. Ακόμη, λαμβάνονται timestamps μέσω της συνάρτησης gettimeofday() που γεμίζουν κατάλληλους πίνακες (producer\_timestamps\_period) που θα χρησιμοποιηθούν στην main για την εξαγωγή των αρχείων .txt.

Στην συνέχεια του κώδικα που φαίνεται στο [ Fig. 6] πραγματοποιείται σύνδεση της TimerFunc που στην προκειμένη περίπτωση απλώς τυπώνει το μήνυμα " I am working now !!! " με τα δεδομένα της μεταβλητής in που είναι τύπου workFunctionData. Τέλος, η TimerFunc τοποθετείται στην ουρά μέσω της queueAdd() , υπολογίζεται ο χρόνος που χρειάζεται να μπει στην ουρά και δημιουργείται η περιοδικότητα μέσω της usleep(). Αναφορικά με την consumer [ Fig. 7] δεν υπάρχουν σημαντικές αλλαγές σε σχέση με την πρώτη εργασία. Αυτό που αξίζει να σημειωθεί είναι ότι το thread consumer τερματίζει όταν εκτελεστούν όλα τα tasks. Ακόμη έχουν προστεθεί και γραμμές κώδικα που αφορούν την εξαγωγή δεδομένων.



```

// associate the struct workFunction with the TimerFunc
in.arg = NULL;
in.work = TimerFunc;
// Timing Calculation
long int start_pro_queue_add, end_pro_queue_add;
gettimeofday(&start_pro_queue_add_s, NULL);
start_pro_queue_add = start_pro_queue_add_s.tv_sec * (int)1e6 + start_pro_queue_add_s.tv_usec;
queueAdd(t->fifo, in);
gettimeofday(&end_pro_queue_add_s, NULL);
end_pro_queue_add = end_pro_queue_add_s.tv_sec * (int)1e6 + end_pro_queue_add_s.tv_usec;
elapsed_time_queue_add[i] = end_pro_queue_add - start_pro_queue_add;
// Timing Calculation //
// release the mutex to grant the consumer thread access to the FIFO //
pthread_mutex_unlock(fifo->mut);
// signal to the consumer thread that the FIFO is not empty, so it can perform dequeue //
pthread_cond_signal(fifo->notEmpty);
// usleep to support periodic calls of the TimerFunc //
usleep(1000 * (t->period));
}
return (NULL);
}

```

Figure 6: Producer Function Part II

```

void *consumer(void *q)
{
    queue *fifo;
    fifo = (queue *)q;
    // out : instance of workfunction struct //
    workFunctionData out;
    while (DONE_TASKS < TASKS_TO_EXECUTE)
    {
        // lock the mut of the fifo to provide access to onlu one thread at a time in fifo //
        pthread_mutex_lock(fifo->mut);
        while (fifo->empty)
        {
            printf("consumer: queue EMPTY.\n");
            // block the consumer thread until 'notEmpty' condition variable is signaled by the producer thread //
            pthread_cond_wait(fifo->notEmpty, fifo->mut);
        }
        // TIMING CALCULATIONS FOR THE CONSUMER //
        long int start_con_queue_del, end_con_queue_del;
        gettimeofday(&start_con_queue_del_s, NULL);
        start_con_queue_del = start_con_queue_del_s.tv_sec * 1000000 + start_con_queue_del_s.tv_usec;
        // delete from fifo //
        queueDel(fifo, &out);
        gettimeofday(&end_con_queue_del_s, NULL);
        end_con_queue_del = end_con_queue_del_s.tv_sec * 1000000 + end_con_queue_del_s.tv_usec;
        elapsed_time_queue_del[DONE_TASKS] = end_con_queue_del - start_con_queue_del;
        // execute the function //
        out.work(out.arg);
        free(out.arg);
        // release the mutex to grant the producer thread access to the FIFO //
        pthread_mutex_unlock (fifo->mut);
        // sends the signal 'notFull', so the producer can add another task to the queue
        pthread_cond_signal (fifo->notFull);
        printf("consumer: recieved");
        printf("\n");
        DONE_TASKS++;
    }
    return (NULL);
}

```

Figure 7: Consumer Function

## 2.5 main logic

Στην main δημιουργούμε την ουρά, θέτουμε το mode που θέλουμε να τρέξουμε το οποίο μπορεί να είναι είτε ISOLATED ή COMBINED. Το πρώτο αφορά το πείραμα του για κάθε περίπτωση timer ξεχωριστά, ενώ στην άλλη περίπτωση όλες οι περιπτώσεις των timers τρέχουν ταυτόχρονα. Στην συνέχεια ξεκινάει ο timer αφού αρχικοποιηθεί με τις κατάλληλες παραμέτρους και έπειτα ξεκινάει ο consumer [ Fig. 8 ]. Αφού ολοκληρωθεί το run execution των consumer η ουρά παύει να υπάρχει και ελευθερώνεται η μνήμη που είχε δεσμεύσει. Στο τέλος της main πραγματοποιείται η εξαγωγή των δεδομενων στα κατάλληλα αρχεία.

```
int main()
{
    // Create a pointer of type queue //
    queue *fifo;
    // Init the fifo
    fifo = queueInit();
    // if the queueInit don't run properly, print an error message //
    if (fifo == NULL)
    {
        fprintf(stderr, "main: Queue Init failed.\n");
        exit(1);
    }
    int mode = ISOLATED; //
    // Declare an instance t of type Timer //
    Timer *t;
    // Init the Timer t instance //
    t = init(fifo, 1000, 2);
    printf("SPECIFICATIONS \n");
    printf("PERIOD = %d ms \n", t->period);
    // create the consumer threads after the timer creation //
    pthread_t *c;
    c = (pthread_t *)malloc(CONSUMERS * sizeof(pthread_t));
    for (int i = 0; i < CONSUMERS; i++)
        pthread_create(&c[i], NULL, consumer, fifo);

    // start at 09/19/23 at 15:56:10z
    // startAt(t, 2023, 9, 20, 0, 8, 40);

    // Start the timer and the producer thread //
    printf("START DELAY = %d sec \n", t->startDelay);
    start(t);
    // producer thread join //
    pthread_join(t->thread_id, NULL);
    // consumer thread join //
    for (int i = 0; i < CONSUMERS; i++)
        pthread_join(c[i], NULL);
    // FIFO Delete
    queueDelete(fifo);
    printf("FIFO DELETED\n");
}
```

Figure 8: main Function, mode = ISOLATED

## 3 Technical Information

### 3.1 Hardware Set up

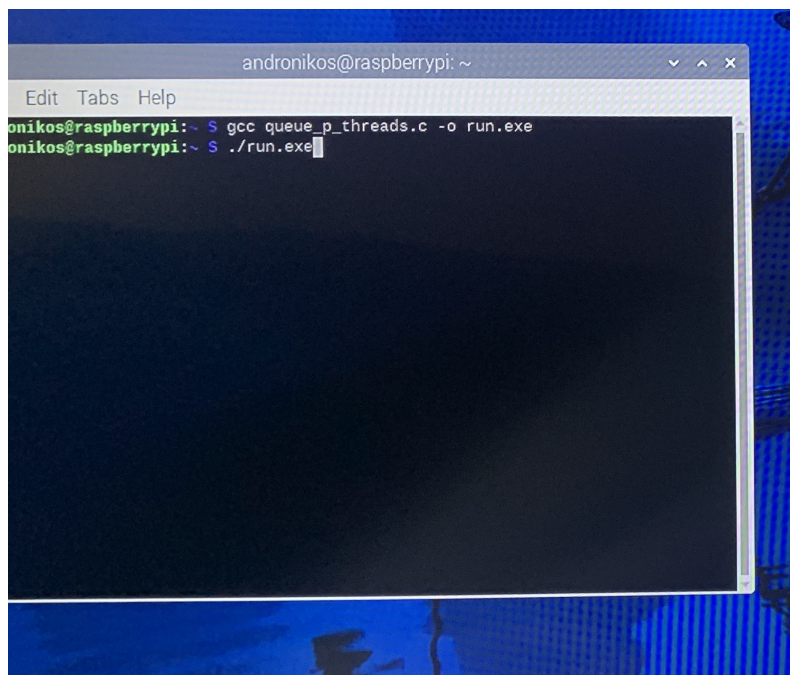
Ο μικροελεγκτής ο οποίος χρησιμοποιήθηκε για την εργασία είναι το Raspberry Pi Zero W και τα περιφερειακά που βοήθησαν στο set up ήταν ένα πληκτρολόγιο, ένα ποντίκι, ένα usb-hub και ένας usb-flash driver για τη μεταφορά δεδομένων. [Fig. 9].



Figure 9: Raspberry Hardware Set Up

## 3.2 Running the .c in Raspberry

Αρχικά πρέπει να περαστεί το αρχείο .c στο raspberry . Αυτό μπορεί να γίνει είτε με scp (secure copy protocol) που βασίζεται στο ssh protocol ή μέσω usb flash driver μό ο usb-hub . Χρησιμοποιήθηκε η δεύτερη περίπτωση καθώς το raspberry pi zero είναι αρκετά αργό με αποτέλεσμα να αδυνατεί να μεταφέρει αρχεία μέσω του διαδικτύου. Αφού υλοποιηθούν τα παραπάνω βήματα τρέχουμε τις εντολές που φαίνονται στο [ Fig. 10 ].

A screenshot of a terminal window titled 'andronikos@raspberrypi: ~'. The window has a menu bar with 'Edit', 'Tabs', and 'Help'. The terminal shows two commands being entered: 'gcc queue\_p\_threads.c -o run.exe' and './run.exe'. The background of the terminal is black with green text. The window is set against a blue background with a grid pattern.

```
andronikos@raspberrypi: ~
Edit Tabs Help
onikos@raspberrypi:~$ gcc queue_p_threads.c -o run.exe
onikos@raspberrypi:~$ ./run.exe
```

Figure 10: Commands to build and run how the .c in raspberry

## 4 Ανάλυση και Σχολιασμός Δεδομένων

### 4.1 Drifting

#### 4.1.1 Plots

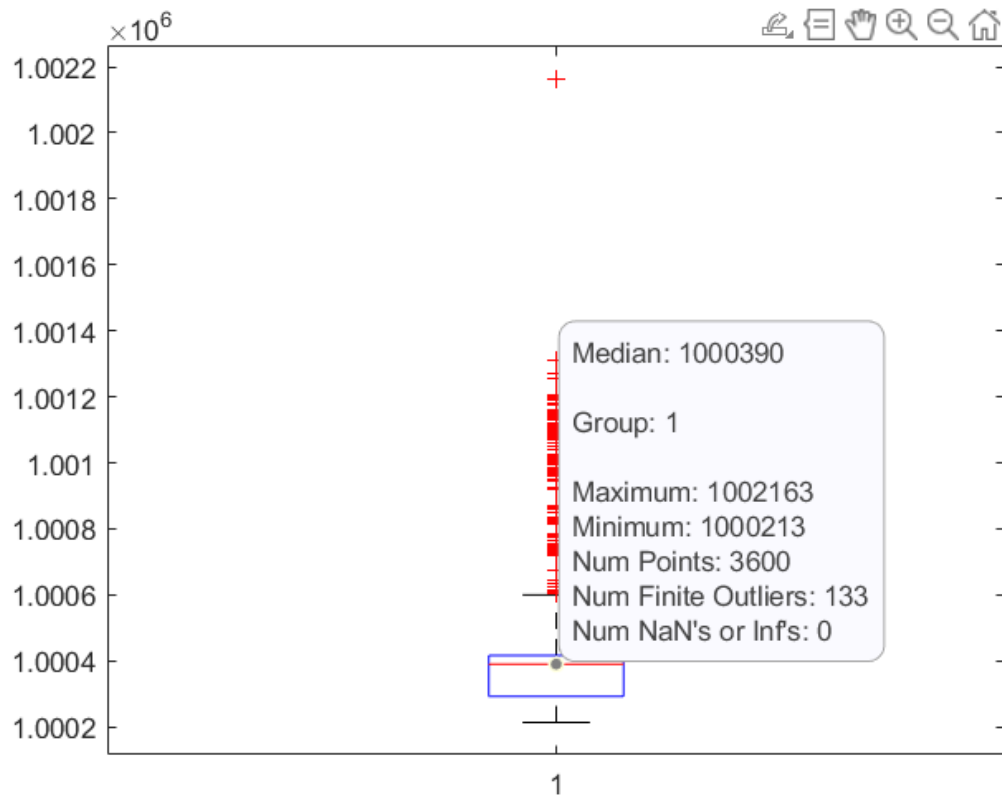


Figure 11: Period drifting at 1s

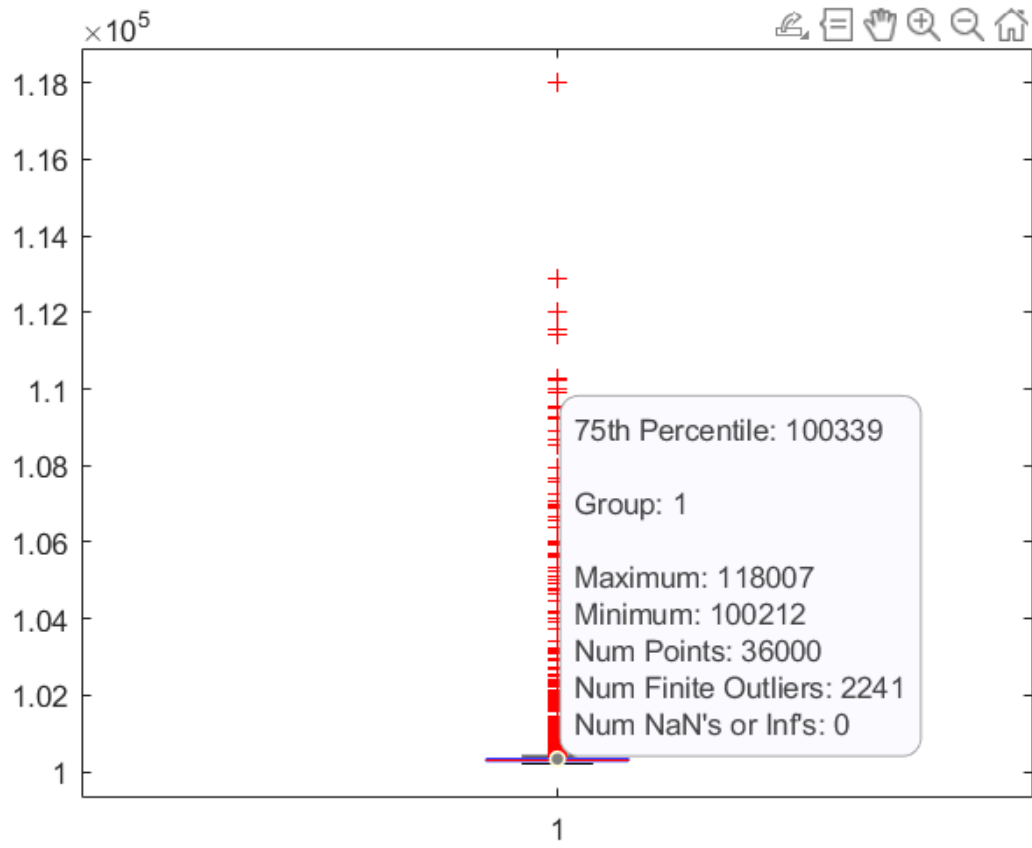


Figure 12: Period drifting at 0.1s

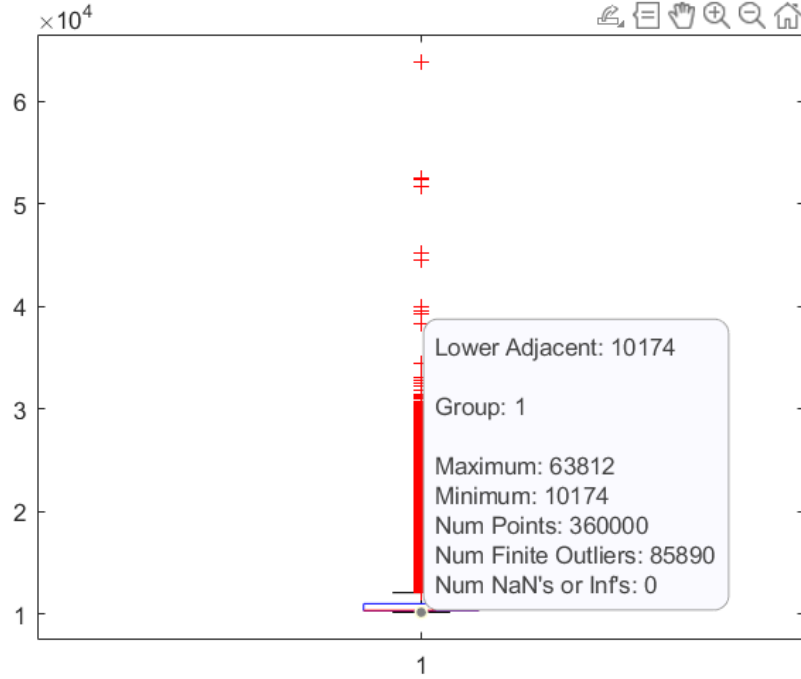


Figure 13: Period drifting at 0.01s

#### 4.1.2 Σχολιασμός Αποτελεσμάτων για το drifting

Από τα διαγράμματα [ Fig. 11 ], [ Fig. 12 ], [ Fig. 13 ] παρατηρούμε ότι όσο μειώνεται ο χρόνος περιόδου τόσο πιο έντονη είναι η εμφάνιση του drifting και αυτό μπορεί να επιβεβαιωθεί από τον αριθμό των outliers ποσοστιαία σε κάθε θηκόγραμμα, όπως φαίνεται και στον [ Table 1 ].

Timer	Outliers Percentage
1000 ms	3.7 %
100 ms	6.2 %
10 ms	23.8 %

Table 1: Outliers percentages for each timer

### 4.1.3 Επίλυση του Drifting

Η λύση που προτείνεται για μείωση του φαινομένου drifting είναι η χρονομέτρηση της συνάρτησης του producer και η αφαίρεση του συγκεκριμένου χρόνου από την αντίστοιχη περίοδο του εκάστοτε timer όπως φαίνεται και στο [ Fig. 14 ].

```
gettimeofday(&tProdExecEnd, NULL);  
// tDrift  
long int tDrift = tProdExecEnd.tv_sec * (int)1e6 + tProdExecEnd.tv_usec - (tProdExecStart.tv_sec * (int)1e6 + tProdExecStart.tv_usec);  
usleep(1000 * (t->period) - tDrift);  
  
// usleep to support periodic calls of the TimerFunc //
```

Figure 14: Drifting Solution Logic

## 4.2 Χρόνοι εκτέλεσης της queue add και της queue del

Timer	Max	Min	Mean	Median	Standard Deviation
1000 ms	165 us	3 us	4.8414 us	5 us	3.2269 us
100 ms	159 us	2 us	4.9451 us	4 us	4.4770 us
10 ms	378 us	2 us	4.2024 us	4 us	4.4895 us

Table 2: Statistical Timing Calculations of queueadd()

Timer	Max	Min	Mean	Median	Standard Deviation
1000 ms	65 us	2 us	4.5653 us	4 us	1.8762 us
100 ms	84 us	2 us	4.6098 us	4 us	3.7871 us
10 ms	5809 us	2 us	3.9913 us	3 us	17.6 us

Table 3: Statistical Timing Calculations of queuedel()

### 4.2.1 Σχολιασμός Πινάκων

Στους πίνακες [ T. 2 ], [ T. 3 ] παρατηρούμε ότι όσον αφορά την μέση τιμή, διάμεσο, το ελάχιστο και για τις δύο συναρτήσεις όπως και για κάθε timer τα δεδομένα είναι



αρκετά παρόμοια. Ακόμη σε κάθε περίπτωση, όσο μικρότερη είναι η περίοδος του timer τόσο μεγαλύτερη είναι η τυπική απόκλιση και αυτό ισχύει και για τις δύο συναρτήσεις με ακραία τιμή να παρουσιάζει ο timer περιόδου 10 ms, κάτι που το περιμέναμε. Τέλος εμφανίζει και μία αρκετά ακραία μέγιστης τιμής στα 5809 us που ίσως δεν μπορεί να ερμηνευτεί.

### 4.3 Λειτουργία Πραγματικού Χρόνου

Το μέγεθος της ουράς εξαρτάται αρχικά από τον αριθμό των timers που τρέχουν. Με την υπόθεση ότι κάθε timer βάζει μια δουλειά στην ουρά, το μέγεθος της ουράς πρέπει να είναι τουλάχιστον ίσο με τον αριθμό των timers. Το μέγεθος της ουράς θα έπρεπε να μεγαλώσει σε περίπτωση που κάποιος timer εκτελούνταν ασύγχρονα και πρόκυπταν ριπές δουλειών που θα έπρεπε να προστεθούν στην ουρά. Ο χρόνος εκτέλεσης της `timerFcn()` πρέπει να είναι σίγουρα μικρότερος της περιόδου του timer που την προσθέτει στην ουρά, ειδάλλως θα είχαμε ένα ασταθές σύστημα με διαρκώς αυξανόμενο αριθμό κλήσεων προς εκτέλεση, καθώς ο αριθμός των consumers είναι πεπερασμένος. Αναφορικά με τον αριθμό των consumers, αυτός εξαρτάται από το τι θεωρούμε ως έγκαιρη έναρξη εκτέλεσης της δουλειάς. Για την αμεσότερη έξοδο μιας κλήσης από την ουρά, ο αριθμός των consumers πρέπει να είναι τουλάχιστον ίσος με τον αριθμό των producers. Βέβαια, ο αριθμός των consumers θα μπορούσε να είναι και μικρότερος, για παράδειγμα εάν ένας consumer προλάβαινε να εκτελέσει όλες τις δουλειές σε χρόνο μικρότερο από αυτόν της περιόδου του timer με την μικρότερη περίοδο, αρκεί να μη μας πείραζε ενδεχόμενη καθυστέρηση της εκτέλεσης κάποιων δουλειών το πολύ μέχρι την περίοδο του timer με την μικρότερη περίοδο.