Real Time Embedded Systems Producer-Consumer Problem with Timer - 2nd Assignement

- Papadakis Charalampos AEM:9128 e-mail:papadakic@ece.auth.gr

- Source code can be found in the cloud: https://www.dropbox.com/s/1odwkl4ypcq0y1a/prod-cons-timer.c?dl=0

Problem Description

In the second and final assignment we are asked to modify the producer-consumer problem that we implemented in the previous assignment , and use Timer objects so that we can execute tasks periodically. On this occasion, we have a **public** queue in which every producer thread adds a function that needs to be executed at that time. This operation happens periodically. The consumers,now, are responsible to “taking” tasks off of the queue and executing them.

Time Time-Lag Problems

Of course, it is obvious that there will be some delay between the time the producer adds the function to the queue and the time that the function is executed. Actually, we are not even sure that the queue will be empty when the producer will attempt the insertion of the function. This is the first problem we need to take in notice in our implementation and measurements .

The second problem that occurs is that the producer must wait for a whole period to be completed, even though the function would ,likely, be inserted in the queue immediately after the previous insertion. The time difference between the moment that the previous insertion was executed and the moment that the producer would add a function again is called **drift time** . As it will be analyzed below, this problem can be solved by forcing the producer to sleep for a specific space of time.

My Implementation

The main difference of this assignment from the previous one is that we define the **Timer** structure, which is defined as it was asked to. In this implementation, we use the Timer to assign the work its thread will have to execute. So, we initialize the

Timer and after that we activate the producer threads , and then the consumer threads will be responsible to execute the functions that were inserted to the queue , as I described before. There is an alternative way to activate the producer threads and is the **startAt()** function, to which we can pass the specific moment we want the thread to be activated.

Also, we should mention that in the producer() function we try to correct the *Time Drift problem.* Specifically, we subtract the drift time calculated every time from the defined *Period*. Then, we make the producer sleep for the **(Period-Drift)** time space using the function **usleep().** This “trick” will minimize the time drift problem and will , finally , make our program to run for the asked duration (1 hour). Otherwise, the program would finish much earlier and jobs would be lost in the operations.

In the previous implementation we had the producers to chose and put in the queue a random function with a random argument. However, I considered that is not so important in this case , so I just used a single function that I called **uselessFunc()** which we calculate a subtract of squares , depending on the REPS defined in the start of the program.

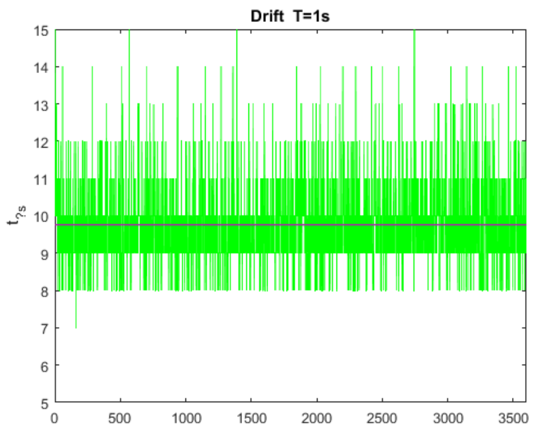
In the start of the program, I define the *OPTION* that I will execute. Option 0 means that the program will run with the defined period and Option 1 means that the program will run for the combined version (Period 1s , 0.1s , 0.01s together).

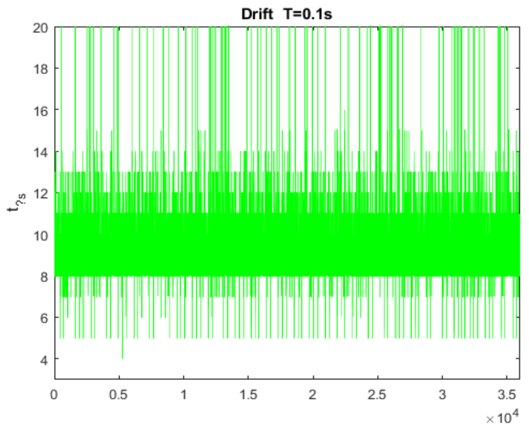
Time Results

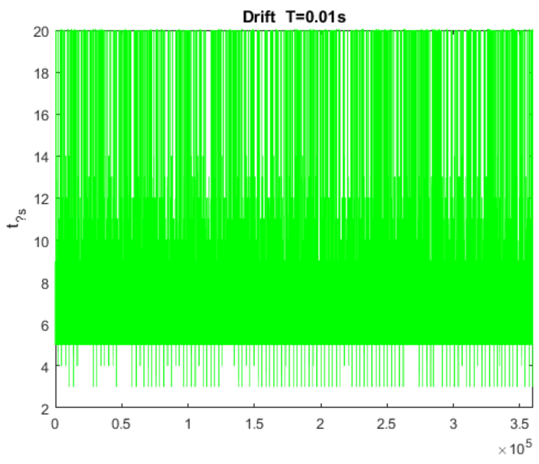
After experimenting with the QUEUESIZE and the Producers and Consumers, I ended up using Queuesize=5, Producers=1 , Consumers=2. The execution of the program was done in my Raspberry Pi Zero , and as I was asked I ran the program 4 times for one hour (total of four hours). The three first executions had period 1sec , 0.1 sec , 0.01 sec respectively, and the final execution had the problem run for the three periods combined.

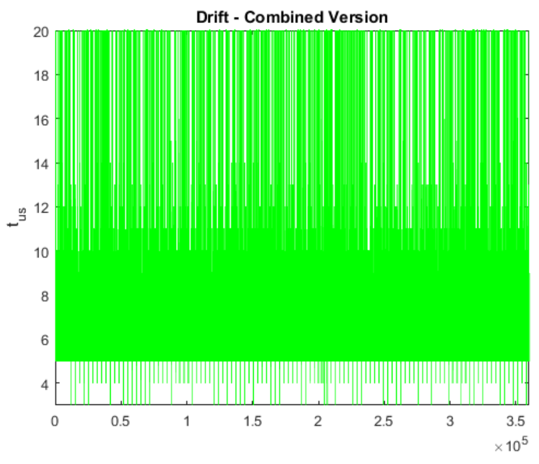
In these results and graphs , I will present the *drift time, the time that is taken for the producer to add a function to the queue, and the queue lag (time that an object waits in the queue before its taken off).* Finally, we will compute the CPU usage percentage. The results were taken by running the programs using the command **time ./exe .**

1) Drift Time:





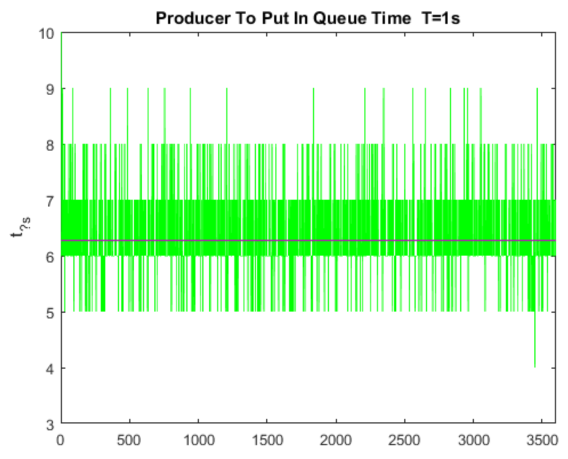


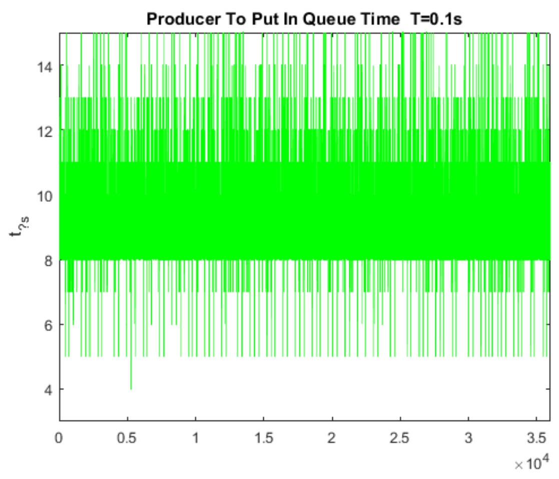


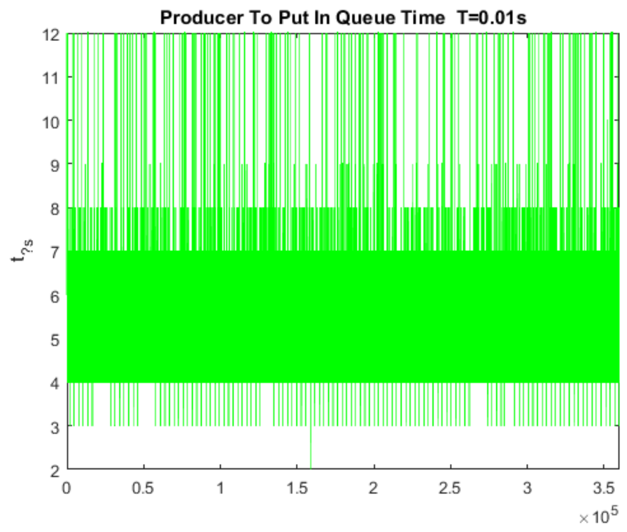
Period(s)	Max	Min	Mean	Median	Std
1s	80	7	9.76	9	2.02
0.1s	217	4	9.24	9	3.53
0.01s	616	3	6.54	6	3.25
Combined	650	3	6.92	7	3.72

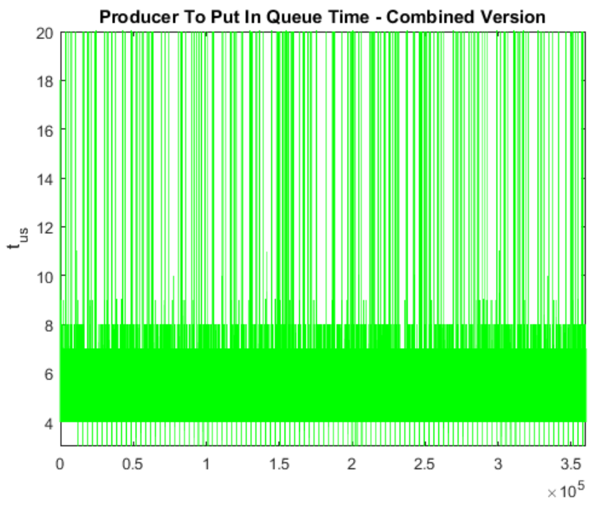
We observe that the drift time is kind of stable. Of course there are some bigger values than expected and this is because of the difficulty of the CPU to handle all these operations.

2)Time that takes to the producer to add to the queue:





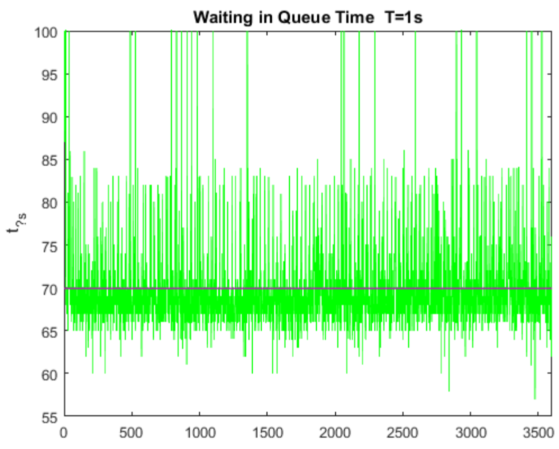


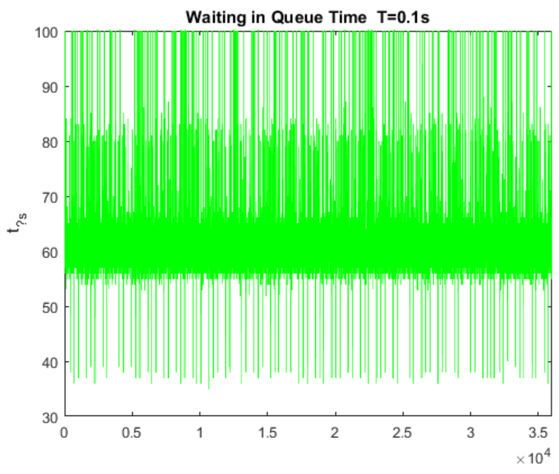


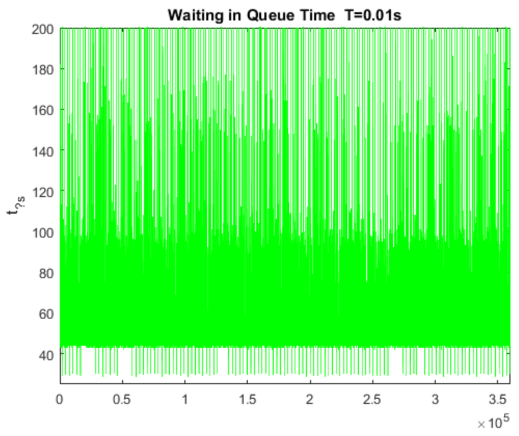
I observe that the producer most of the times puts the function in the queue really fast, however there are some high values ,maybe because of the size of the queue.

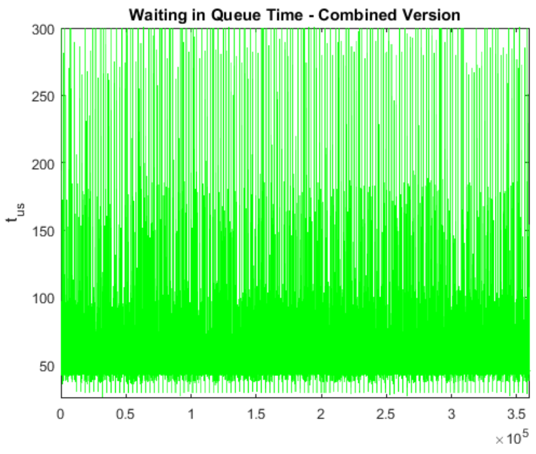
Period(s)	Max	Min	Mean	Median	Std
1s	15	4	6.27	6	0.65
0.1s	217	4	9.24	9	3.54
0.01s	219	2	4.97	5	1.53
Combined	502	2	4.95	5	2.13

3)Time that a function waits inside the queue:









Period	Max	Min	Mean	Median	Std
1s	372	57	69.93	69	7.75
0.1s	414	35	61.02	60	10.94
0.01s	800	29	48.55	48	12.63
Combined	950	26	49.72	47	14.79

Comparing the results with the period, we can observe that the “jobs” are executed pretty fast, so we can consider our system reliable.

4)CPU Usage:

As I mentioned before, after running the program with the command **time ./exe** we can calculate the CPU usage percentage by **(user time + system time) / real time.** The results that I came up with were the ones we expected. Specifically, in every version of the program the CPU Usage % was really close to zero. Also , the Real Time result showed us that the program ran for exactly an hour, indeed.