Timer in C

Real Time Embedded Systems

Konstantinidis Dimitrios

AEM : 10106

Email: dakonstan@ece.auth.gr

1. Introduction

A timer consists of a function that executes at specific time intervals. In this assignment a timer was created, based on a software approach, using Pthreads. A previous assignment for a multithreaded consumer-producer problem with the use of a FIFO queue and Pthreads was modified so the producer can sleep for a specific time period, achieving a periodic execution.

The suspension of a producer thread was achieved using usleep. The problem with this function arises when there are more threads than processor cores or when the program is used in conjunction with a non deterministic computer system, where the Operating System might interrupt the execution of the program in order to continue executing another process. This results in an additional time where a producer thread is suspended and not executing so the total delay of this thread is greater than that of usleep. As the execution moves on these extra delays add up and cause the timer to drift from its ideal execution time. In order to ensure a stable time drift that does not accumulate as the execution of the timer progresses the time that a producer sleeps must be calculated dynamically.

The consumers remain idle up to the point that an item is placed in the queue and it is no longer empty. They must be able to execute the task of removing an item from the queue and run the work function as soon as possible so the function is run periodically and its execution time matches that of the producer. Once the required number of executions have completed the timer must be destroyed.

2. Changes compared to the provided prod-cons.c file

Besides the changes to the producers so they can “sleep”, the consumers were modified and instead of a for loop, a while loop that always runs, provided that the producers continue executing, was used. This change regarding the termination of the execution of the consumers was needed, because there might be more than one consumer threads and the number that each thread executes is non deterministic. If a for loop was used, the consumer threads will finish executing if the sum of the executed for loop iterations in all consumers equals the total number of desired executions. This means that there should be a shared variable where the number of executions is stored, creating a possible race condition and therefore a mutex is needed at each iteration of the loop. Using the variable termination, a producer increments this zero initialized variable once it has finished executing and if its value is equal to the number of producers and the FIFO queue is empty then the consumers terminate. In this case variable termination is also shared among all the producers but must be updated at max 3 times significantly less than a variable that would store the number of executions of the consumers.

Also in both the consumer and producers extra mutexes were used regarding the printing to common files so they can be performed atomically.

The queue has also changed in order to store a workFunction struct instead of an integer number, while extra mutexes were added for the printing to common files. These extra mutexes are not necessary and the standard mutex of the queue could have been used but that would mean that a consumer or producer would remain blocked waiting for another one to finish printing data instead of executing other independent instructions. By using the extra mutexes for example one consumer can remove and execute a workFunction from the Queue while another one is printing to a file.

3. Implementation

The timer is implemented in file timer.c and the helper.c has helper functions related to user inputs.

For the definition of the timer a C struct was used where the desired functions of the assignment were implemented. In order to create, initialize and delete a timer struct the functions timerCreate and timerDelete were defined. For starting a timer there are 2 options, one using the function start to start the timer immediately and another one using startat to start executing at a specified time and date (d/m/y h:min:sec).

The producers of the timer save a workFunction to the queue, that has the following characteristics:

```c
typedef struct {

 void * (*work)(void *);

 void * arg;

 struct timeval timeIn;

} workFunction;
```

Variable work and arg hold the function and argument of the function that must be executed while struct timeIn is used to store the time before a workFunction is inserted into the FIFO queue. This timeval struct is not needed by the timer in order to execute but is helpful in determining the queue wait and insert times.

4. Fixing the time-drifting

In order to fix the time drifting, the ideal time that a producer should execute is calculated. This time equals the sum of the previous ideal time, previous sleep time and the previous total drift time. By subtracting this number from the difference between the current and first execution time the new total drift for the current execution is calculated. Then the sleep time equals the difference between the timer period and the declination, if it is positive else it is zero. This implementation can fix the unpredictable delays of each execution but cannot fix constant delays that contribute to the non zero total drift but since these are small and this mechanism prevents them from accumulating the total drift remains small and acceptable for a real time system. Another approach, that aimed to reduce the constant delays, where from the total sleepTime an extra time equal to the average total drift of the previous 30 executions was subtracted, had the problem of instability with the total declination even taking negative values and therefore it was not implemented in the final version of the timer.

5. Verification of real time execution of the timer

The correct execution of the program was verified, using the total number of Executions, the Total Drift time and the JobsWaitTime. The number of executions within a specified time must equal the desired number of executions and the total drift must always be less than that of the period. This ensures that not only the program executed the correct number of times, but also that the producer placed an item in the queue within a correct time window. If the total drift exceeded at some point the period this means that an execution time window was missed and the timer skipped the execution and thus finished later than expected. Also for the correct operation of the timer the time between two consecutive work functions executions must equal the period, so the time that a work function waits in the queue is also important, with a zero time indicating that a function executed the same time that it was placed in the queue, which is the desirable behavior.

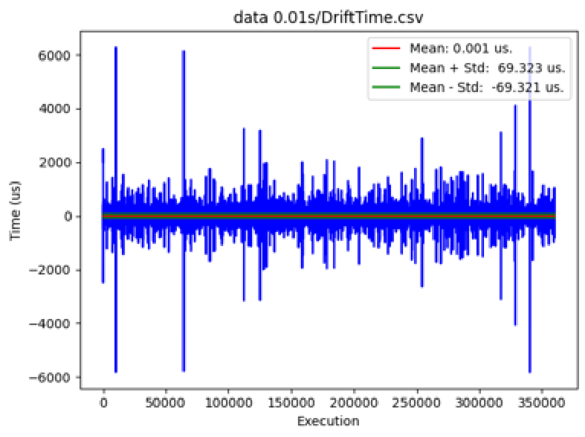
6. Running on Raspberry

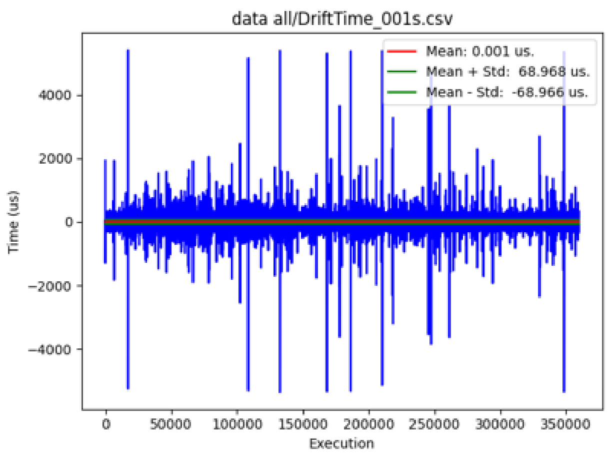
The following experiments run on a Raspberry Pi 0 W using the image provided as well as the most recent Raspberry Pi OS version. The communication with the developer's computer was achieved using ssh. In order to compile for an Arm Architecture a cross compiler was used and the compilation was performed using the command "arm-linux-gnueabihf-gcc". An alternative is to install a compiler directly on the Raspberry or to use the installed compiler, if available, as is the case with the recent Raspberry Pi OS versions. Then the usual gcc command can be used, but the compilation will be slower due to the less powerful processor. In order to transfer files between the Raspberry Pi and the developer's computer scp was used.

7. Statistics

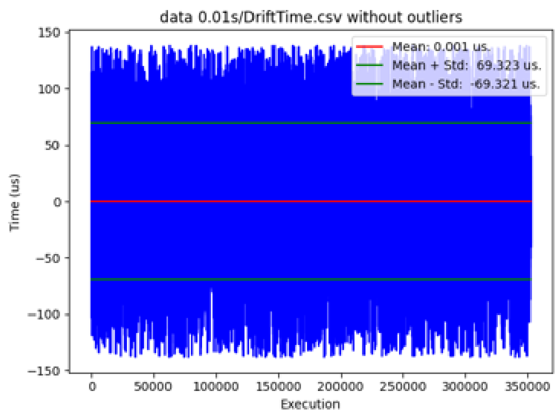
The charts with title "without outliers" don't include extreme outlier values (`df['data'][z] < mean + 2*std and df['data'][z] > mean - 2*std`). The experiments were for 1 consumer in the case of a single timer and 4 when all the 3 timers run together. The length of the Queue was 4 and no job was lost in the 4 experiments.

Drift Time refers to the difference of the actual time between two executions of a producer and the desired period. A positive drift means that the time between two executions of a producer is greater than the period desired. In an ideal timer the drift would be zero and a close to zero value would be acceptable. In our case, in all the experiments drift time has a mean value of less than 1 us with a small standard deviation (<100 us) meeting this requirement.





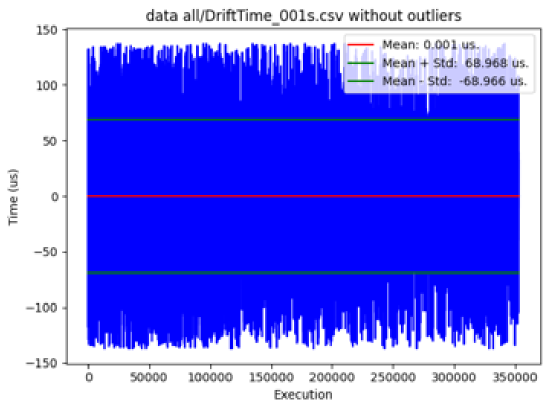
¶



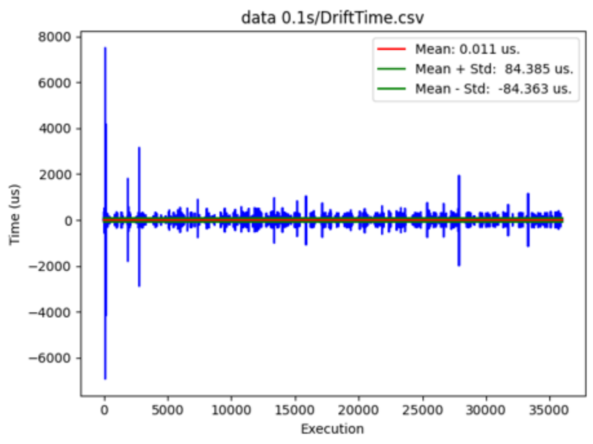
¶

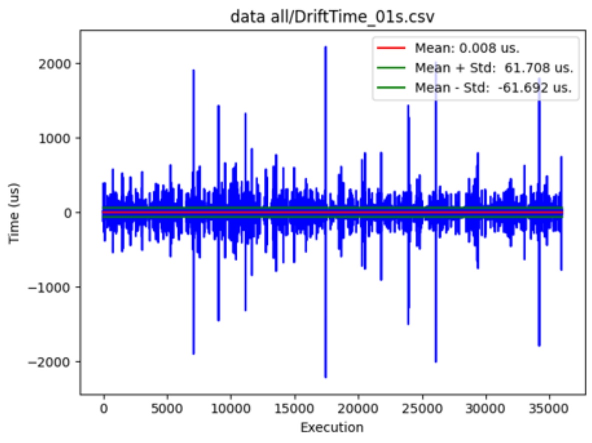
¶

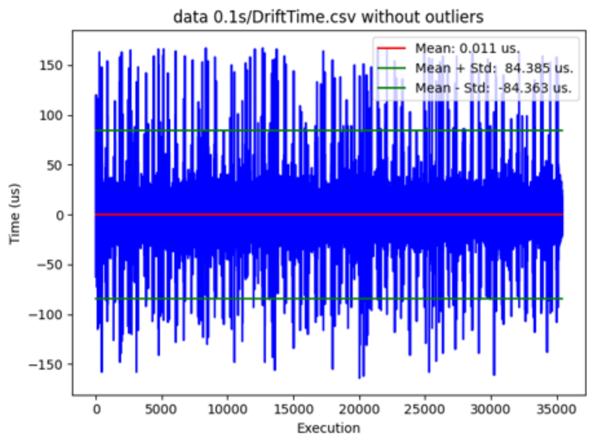
¶

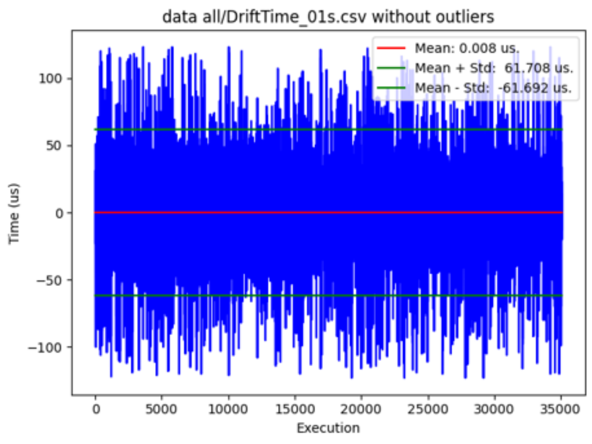


“data all/DriftTime_001s.csv” refers to the drift time of the timer with a period of 0.01s, when it runs with all the other timers.

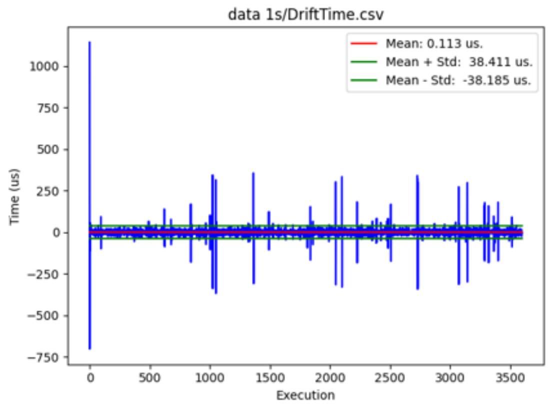


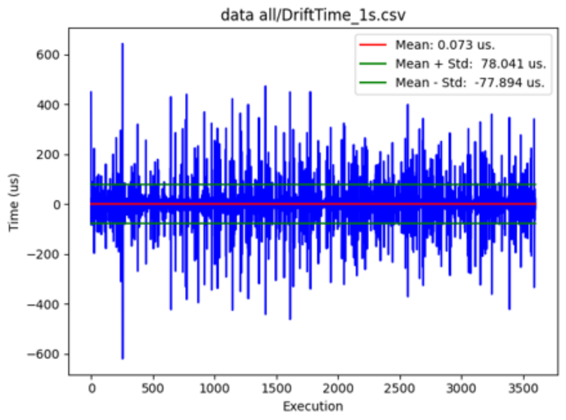


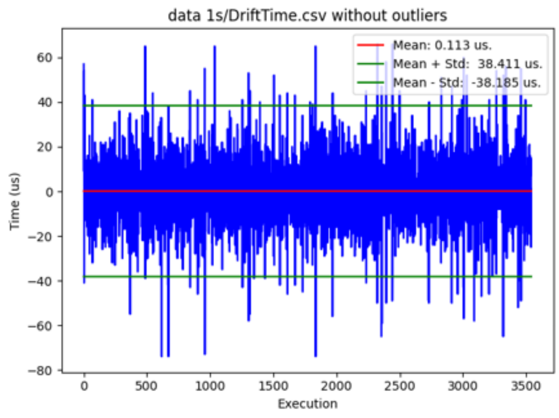


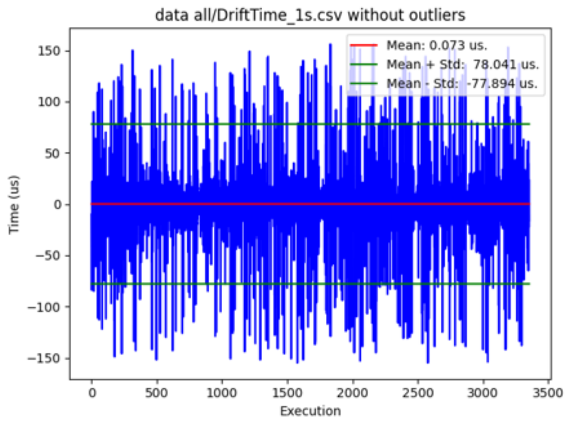


"data all/DriftTime_01s.csv" refers to the drift time of the timer with a period of 0.1s, when it runs with all the other timers.



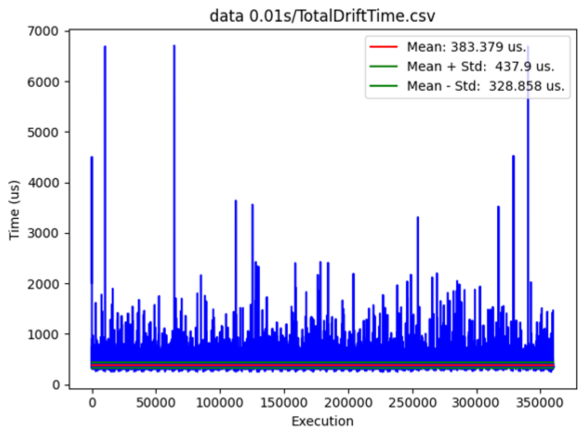


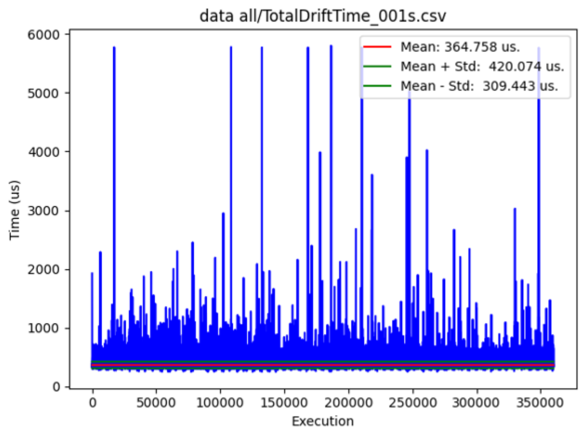


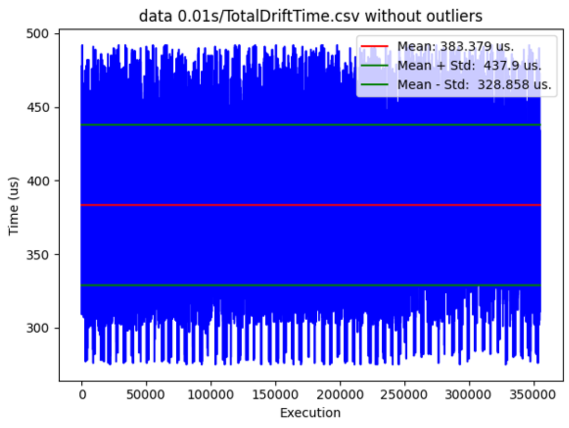


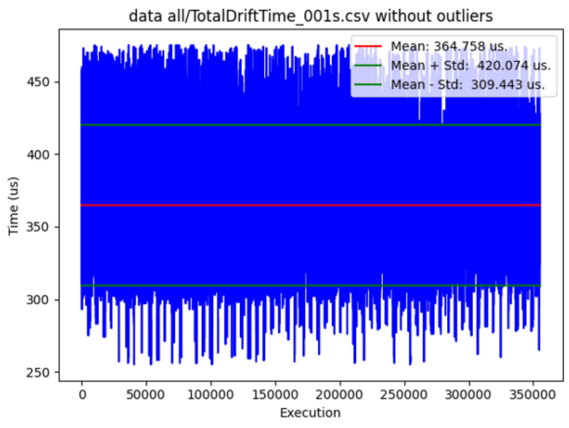
“data all/DriftTime_1s.csv” refers to the drift time of the timer with a period of 1s, when it runs with all the other timers.

Total Drift Time is the difference between time1 and time2. Time1 is the difference between the time of the current execution of a producer and its first execution and time2 equals the time of the ideal execution time of a producer minus the first time of its execution. The Total Drift Time in any of the preceding experiments is always smaller than the period of each timer and is relatively stable during the 1 hour run time. There are however some peaks, especially in the execution of a 0.01s timer that could potentially cause problems, but their number is small and the drift is corrected immediately in the following producer execution. This means that the small average total drift, compared to the timer period, combined with the small standard deviation can confirm the real time of the timer for the majority of the time.

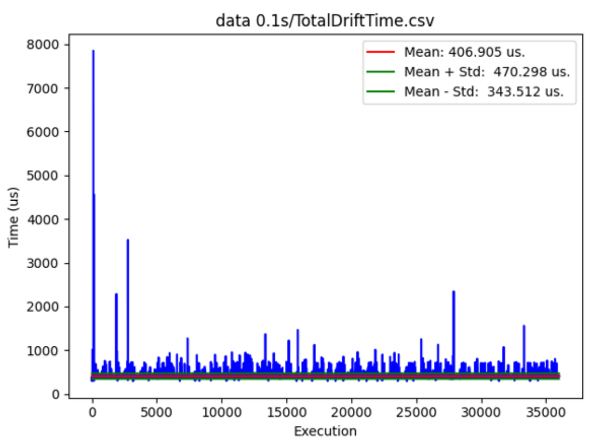


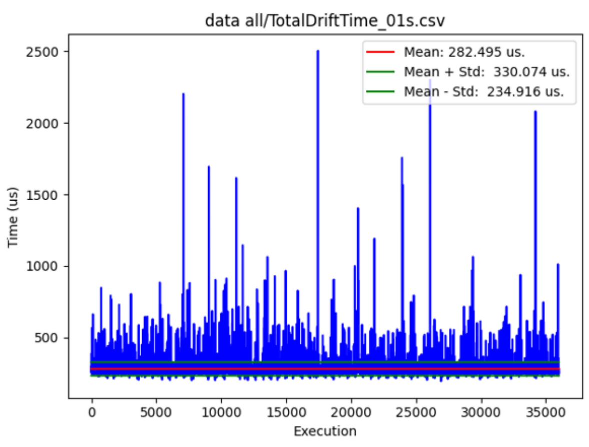


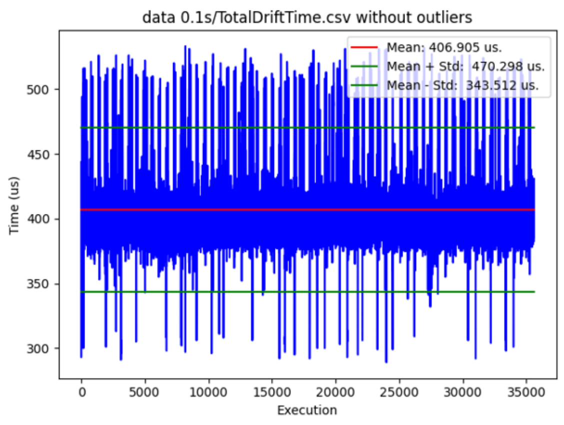


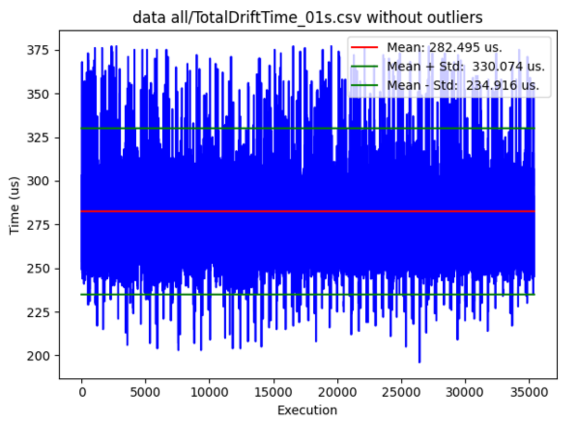


"data all/TotalDriftTime_001s.csv" refers to the drift time of the timer with a period of 0.01s, when it runs with all the other timers.

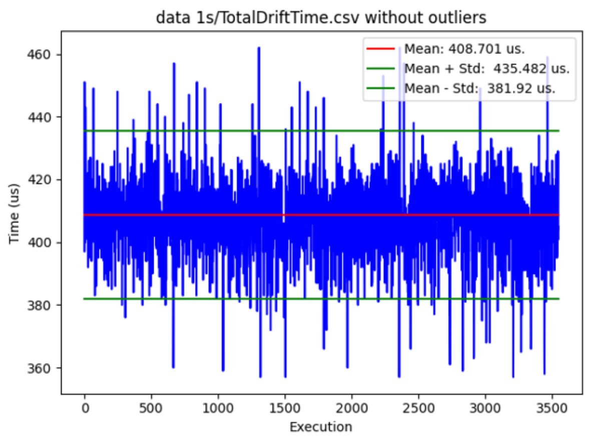


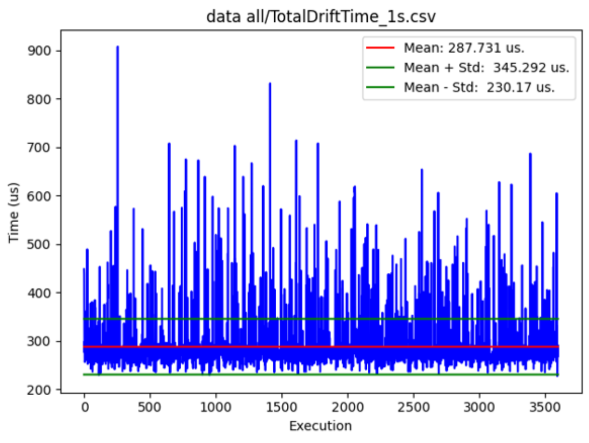


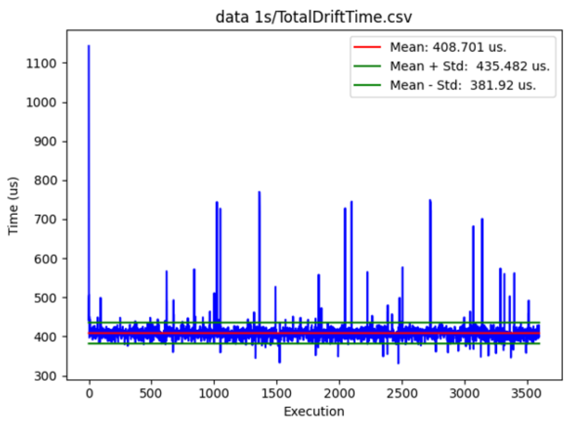


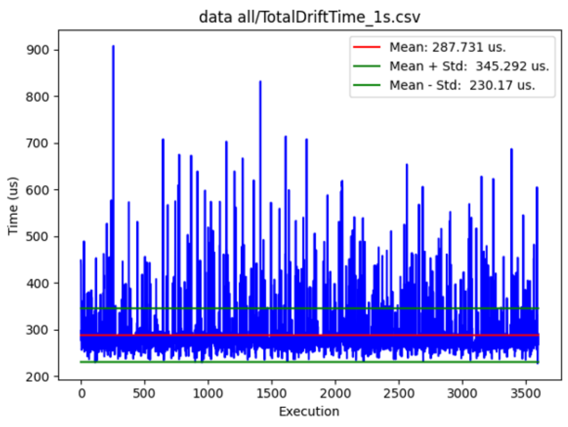


“data all/DriftTime_01s.csv” refers to the drift time of the timer with a period of 0.1s, when it runs with all the other timers.



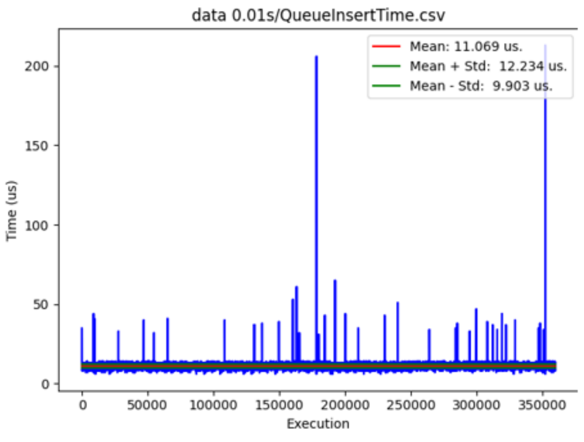
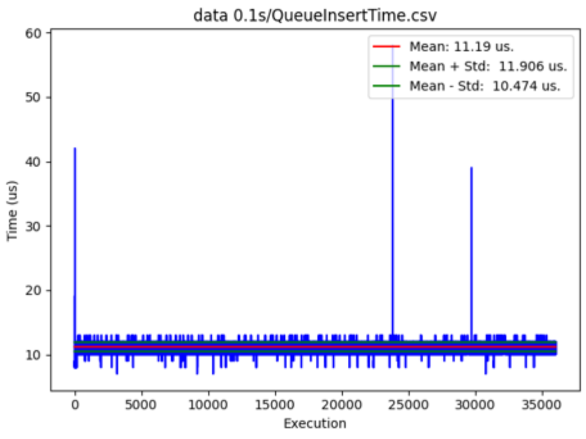
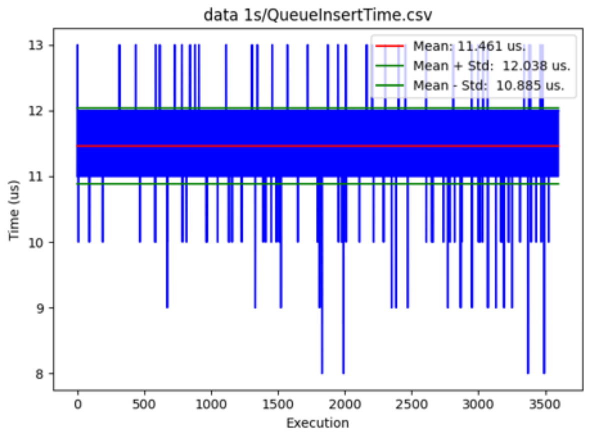
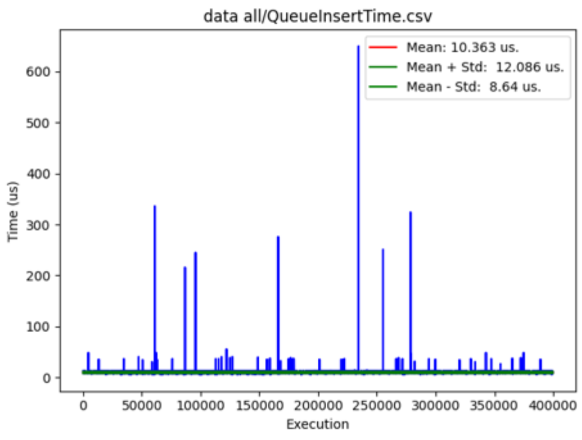


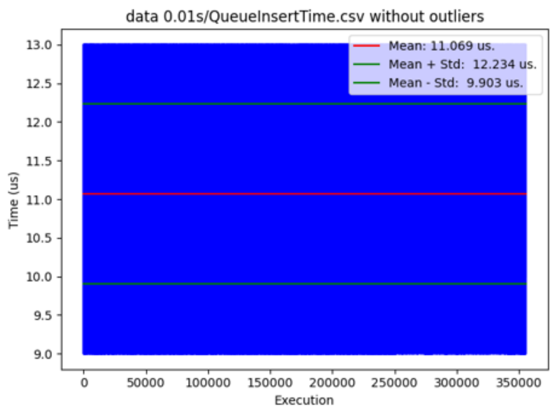


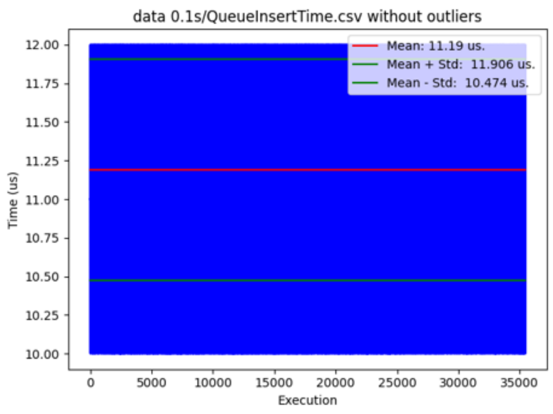


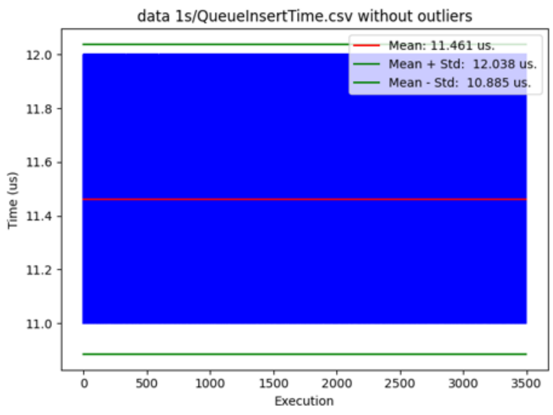
"data all/DriftTime_1s.csv" refers to the drift time of the timer with a period of 1s, when it runs with all the other timers.

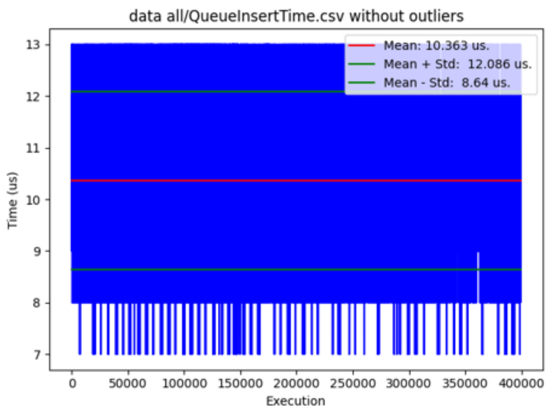
QueueInsertTime is the time that it takes to place a workFunction in the queue. From the experiments it is clear that the time that an element takes to be inserted to the queue is almost zero compared to the desired period. In any case this time is not a problem to the correct execution of the program provided that it is small enough to allow the execution of the rest of the producer's code within the timer's period.


data 0.01s/QueueInsertTime.csv
Mean: 11.069 us.
Mean + Std: 12.234 us.
Mean - Std: 9.903 us.
Time (us)
Execution

data 0.1s/QueueInsertTime.csv
Mean: 11.19 us.
Mean + Std: 11.906 us.
Mean - Std: 10.474 us.
Time (us)
Execution

data 1s/QueueInsertTime.csv
Mean: 11.461 us.
Mean + Std: 12.038 us.
Mean - Std: 10.885 us.
Time (us)
Execution

data all/QueueInsertTime.csv
Mean: 10.363 us.
Mean + Std: 12.086 us.
Mean - Std: 8.64 us.
Time (us)
Execution



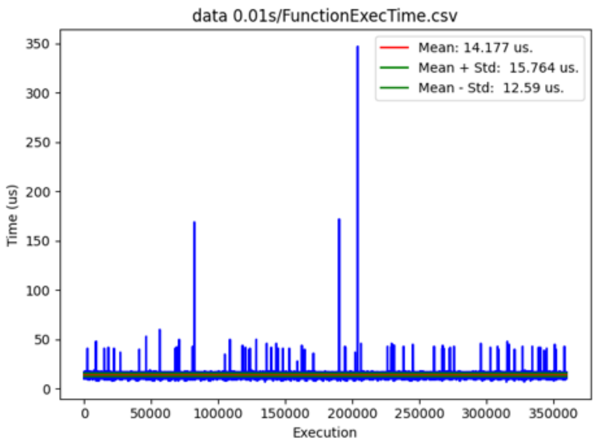


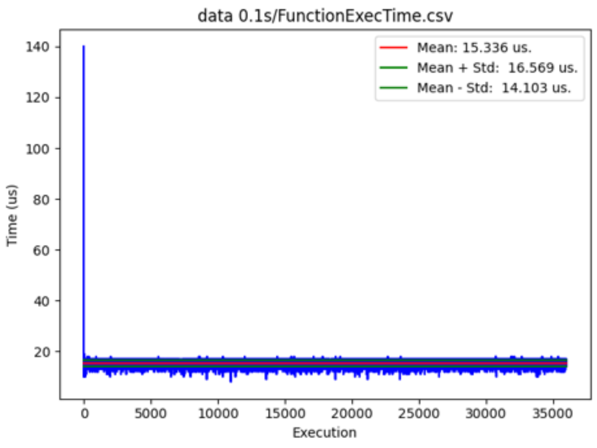


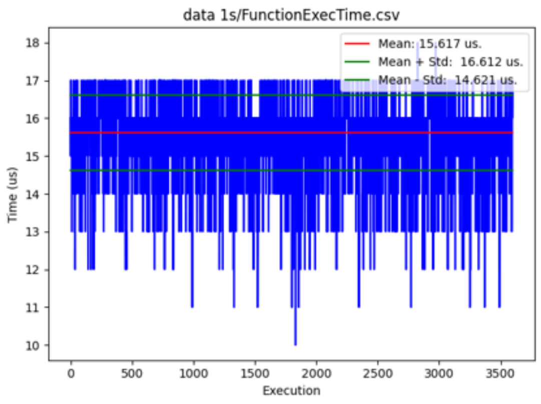


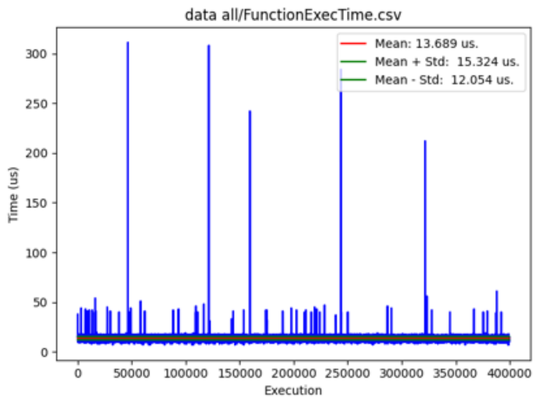
“data all/QueueInsertTime.csv” refers to the queue insertion time of a work function when all 3 timers are running.

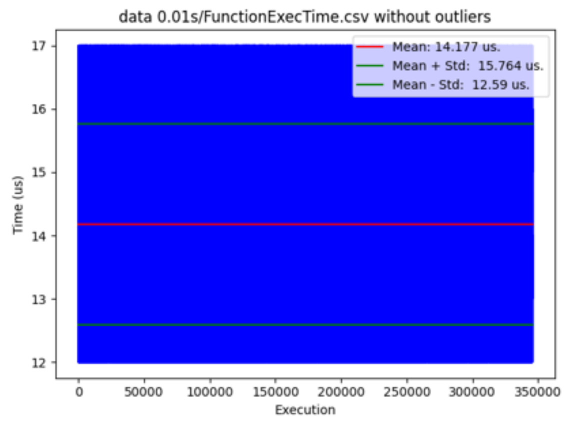
FunctionExecTime is the time that it takes for a function of the queue to execute. The function execution time is irrelevant to the real time execution, since the consumer threads can execute their respective work functions in parallel. However it is a good indicator of the number of threads that must be created, so there is always one idle consumer thread ready to remove an item from the queue immediately after it is added.

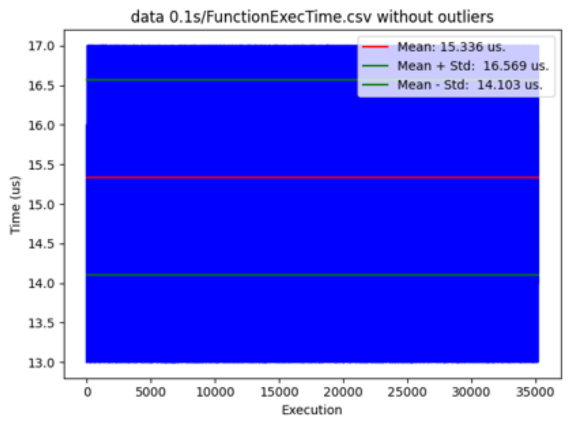

data 0.01s/FunctionExecTime.csv
Mean: 14.177 us.
Mean + Std: 15.764 us.
Mean - Std: 12.59 us.
Time (us)
Execution

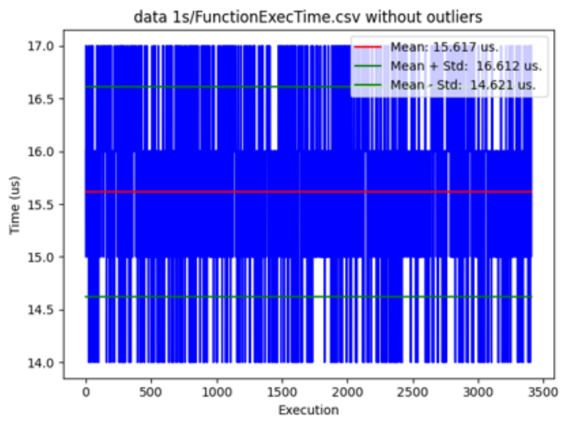

data 0.1s/FunctionExecTime.csv
Mean: 15.336 us.
Mean + Std: 16.569 us.
Mean - Std: 14.103 us.
Time (us)
Execution

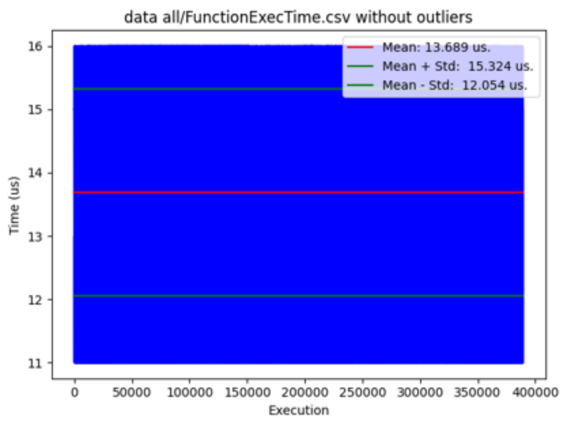

data 1s/FunctionExecTime.csv
Mean: 15.617 us.
Mean + Std: 16.612 us.
Mean - Std: 14.621 us.
Time (us)
Execution


data all/FunctionExecTime.csv
Mean: 13.689 us.
Mean + Std: 15.324 us.
Mean - Std: 12.054 us.
Time (us)
Execution



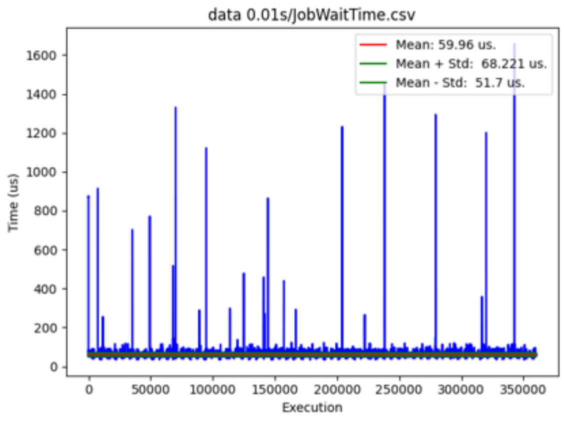


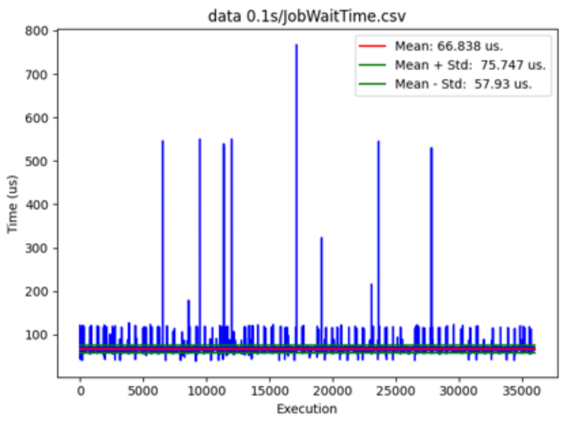


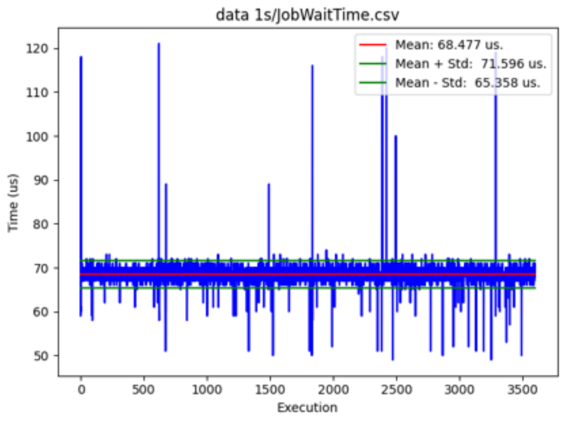


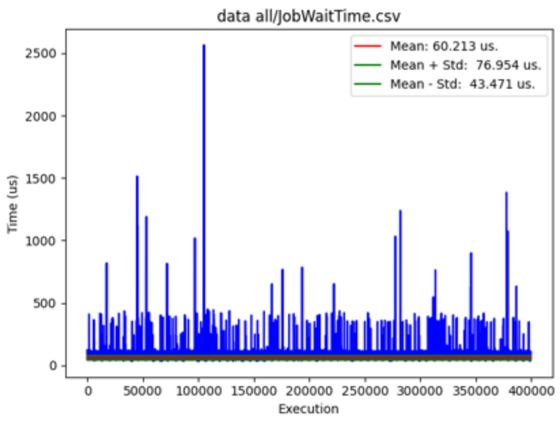
"data all/FunctionExecTime.csv" refers to the execution time of a function when all 3 timers are running.

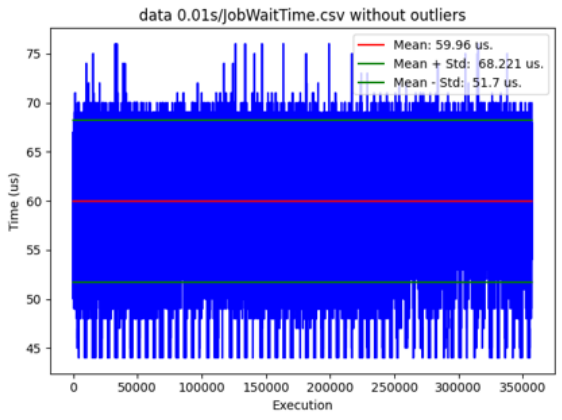
jobWaitTime is the time difference between the start of the insertion process of a workFunction into the queue and the time that it is removed from it. Since the requirement of the timer is to execute a function at constant time intervals, the work functions must be removed almost immediately from the queue after they have been inserted (in the producer consumer implementation only the insertion of function into the queue is periodic not their execution). The charts below show that in all of the experiments the average jobWaitTime is close to zero and therefore the program satisfies this condition, with the exception of some isolated cases.

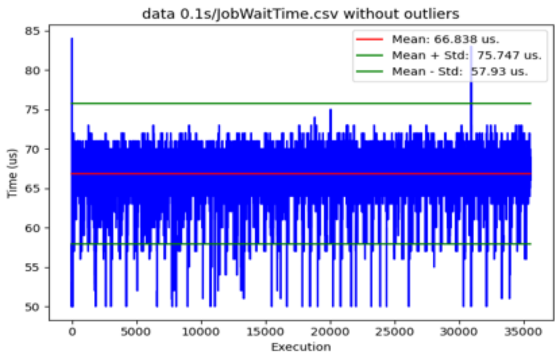

data 0.01s/JobWaitTime.csv
Mean: 59.96 us.
Mean + Std: 68.221 us.
Mean - Std: 51.7 us.
Time (us)
Execution

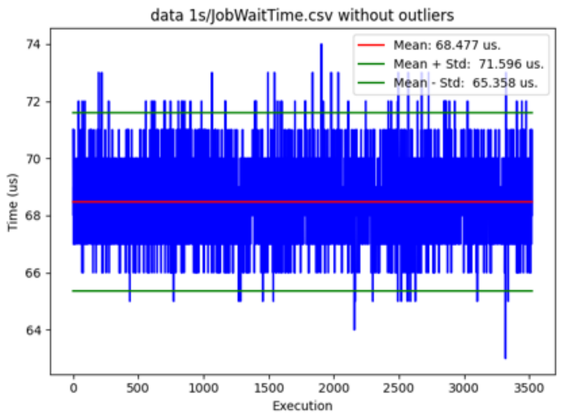

data 0.1s/JobWaitTime.csv
Mean: 66.838 us.
Mean + Std: 75.747 us.
Mean - Std: 57.93 us.
Time (us)
Execution

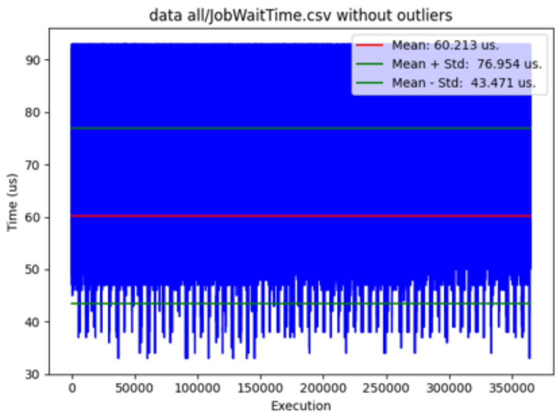

data 1s/JobWaitTime.csv
Mean: 68.477 us.
Mean + Std: 71.596 us.
Mean - Std: 65.358 us.
Time (us)
Execution


data all/JobWaitTime.csv
Mean: 60.213 us.
Mean + Std: 76.954 us.
Mean - Std: 43.471 us.
Time (us)
Execution









"data all/jobWaitTime.csv" refers to the time that a job waits in the queue when all 3 timers are running.

8. Average CPU Usage

Using the command htop/top the process id corresponding to our program with its corresponding CPU usage were found. In the most recent Raspberry Pi OS version pidstat was installed and it has the option of setting the time interval for calculating the average CPU usage. For the measurements a time interval of 6 minutes was used (pidstat 360 -p pid).

Timers	CPU Usage %
1s	0.03
0.1s	0.33
0.01s	3.05
All timers	3.32

As it was expected in all executions the CPU usage is close to zero, since our program is not computationally intensive.

9. Comments

As stated earlier the total drift was never zero but it is small enough compared to the period. This small drift is caused by some operating system procedures regarding thread and process switching and it cannot be eliminated. If the timing constraints are even stricter, then a real time operating system must be chosen or a timer hardware peripheral must be used. The use of a hardware timer with an interrupt support mechanism, set with the highest priority and a preemptive scheduler would ensure a near zero total drift, with a tiny delay, caused by the interrupt handler mechanism (saving context, stack etc) and the scheduler.

Is it also remarkable that the total drift when running on the Raspberry Pi was less than when running on a MacBook Pro, a fact that highlights the complication of using multiple processes and complex operating systems in a computer that is meant for real time operations. In the case of the computer there were more unpredictable delays with greater deviations and thus the drift was not eliminated to the same degree.