Real Time Embedded Systems - Final Assignment - ECE AUTh 8th Semester

Kitsios Konstantinos 9182

July 25, 2020

1 Introduction

In this final assignment of the course we are experimenting with Timer objects in an attempt to design a system that can execute given tasks at specific future times periodically. We would like our system to be as accurate as possible leading to minimum delay in the scheduled tasks. To achieve this, we should take into consideration different concepts from multithreaded programming.

More specifically, the Timer will be implemented using the **producer - consumer** template from the first assignment: For each Timer object there exists a producer thread that inserts functions to a **shared queue**. The insertion takes place at the specific moment the function is meant to be executed and this process is repeated for a predefined number of interations.Then, a number of consumer threads are responsible for extracting the functions from the queue and executing them. Note here that CPU occupation between producers and consumers is handled automatically by the OS, just like the first assignent.

Although this approach is elegant, there is a delay in the execution of the functions: since they are inserted in the queue at the time they are supposed to be executed, there is no guarantee that the queue will be empty or that an active consumer thread will extract them instantaneously. So we are expecting a Queue Lag delay. Furthermore, we observe yet another lag in the time a consumer spends to insert an item in the queue. This can be seen from the fact that after a producer thread wakes up, it must wait exactly one period before inserting into the queue again. However, due to the randomized CPU occupation from other threads, the producer will not instantly get back to the point where it gets to sleep again. We call the time between waking up and going back to sleep *drift time*, and unlike *queue lag* it can be efficiently minimized.

2 Implementation

Our main data type is a custom structure called *Timer*, defined as in Figure 1. Our *main()* function is very similar to the producer-consumer example except that it instantiates the *Timer* objects along with the producer and consumer

```
typedef struct{
    int Period;
    int TasksToExecute;
    int StartDelay;
    void * (*StartFnc) (void *);
    void * (*StopFnc)  (void *);
    void * (*TimerFnc) (void *);
    void * (*ErrorFnc) (void *);
    void *UserData;
} Timer;
```

Figure 1: Timer struct definition

threads. The majority of modifications take place inside the *pro()* function, the one executed by producer threads. There, we utilize the *usleep()* function to force the producer thread to sleep for the specified period. However, as we mentioned above, this approach will cause a drifting phenomenon which is our responsibility to minimize.

For this purpose, we want to measure 2 timestamps: The moment when the producer thread wakes up t_1 and the moment when the producer thread actually inserts the item in the queue t_2. Then the drift time is equal to $t_2 - t_1$. Note that for our architecture to be totally precise, these two must be equal. Of course we cannot guarantee the equality, but we can get them as close as possible with the following technique: Just before putting the thread to sleep, we calculate $drift = t_2 - t_1$ and substract the $drift$ from the actual period of the *Timer*. This way, the thread will sleep not for T seconds, but for $T - drift$ seconds where the drift represents the time delay of the previous execution. Under the assumption that the drifting time is constant(or has low standard deviation) this modification will drastically improve timing accuracy.

3 Statistical Analysis

In this section we present statistics to drive some conclusions about the stability and the real time characteristics of our system and defend our assumption that the drifting time can be seen as constant. We are particularly interested in 2 quantities as mentioned above: the *drift time* and the *queue lag*. Cancelling out the effect of these 2 leads to a stable and reliable real time system. For the measurements below we experimented with 2 *timerFnc* implementations, the first one calculating 10 *sin()* values depending on active *tid* and the second calculating 1000 such values. Note that the execution time for the 2 functions differs a lot in order for our analysis to be idepentant of *timerFnc()*. However, the results are identical if we slightly increase the number of consumers and the queue size for the second function hence they will not be presented separately.

3.1 Drift Time

Measurements indicate very small drifting time after our drift fixing adjustment. The results for the 4 requested experiments are shown in Figure 2, where the y-axis represents the drifting time and x-axis is just an increasing index(tick labels were removed because the times series have different lengths for different periods). The red horizontal line is the mean value while the two yellow horizontal lines indicate the positions $mean - std$ and $mean + std$. For each

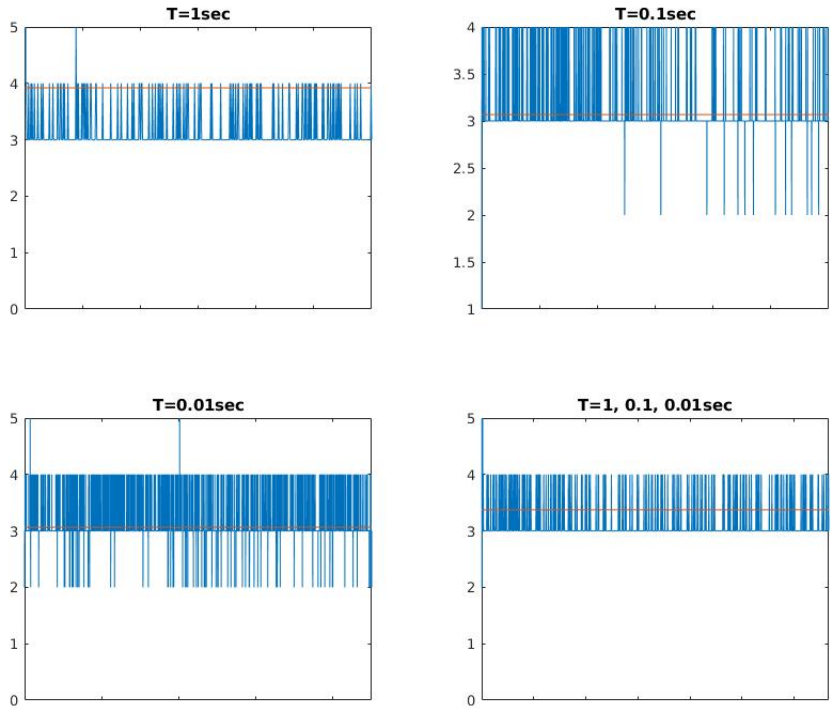

Figure 2: Drifting time measurements(time in us)

case, a drift time of about $3 - 4usec$ is observed which is acceptable for a real time system while the standard deviation is also small which supports our arguement for constant drifting time. We observe spikes at random times(height of the spikes is not seen in Figure 2 since the *ylim* was adjusted to capture small details) that could be the result of the scheduler assigning the CPU at random to different threads, statistically leading to some late assignments after a large number of repetitions. However, the spikes near zero are of particular interest. Since they are observed at each of the 4 cases(look closely near x=0) they

cannot be random. Indeed, they are the result of not applying the adjustment of substracting the drift time, since in the very first measurements drift time cannot be specified. Hence, the spikes near zero can be seen as the drifting time if we had not applied drift fixing.

3.2 Queue Lag

The following results come from measurements with $Q = 2$ consumers and $QUEUESIZE = 2$ in the first 3 plots and $Q = 4$ consumers and $QUEUESIZE = 4$ in the last one. In general, higher delay is observed due to queue lag compared to drifting as can be seen in Figure 3. However, this can be significantly

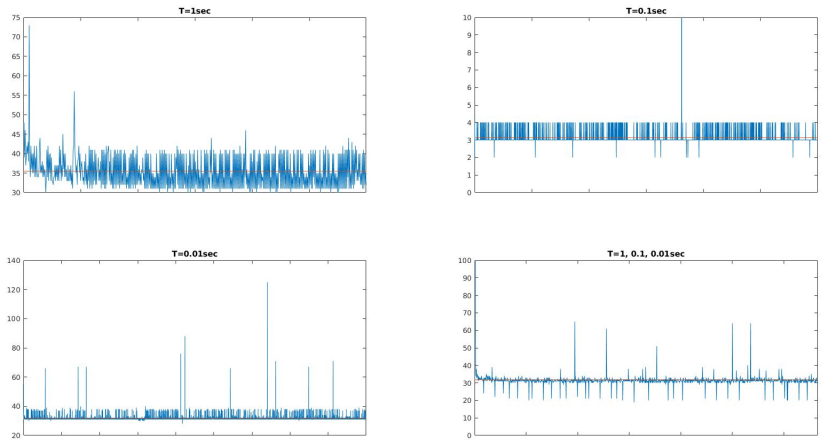

Figure 3: Queue Lag measurements(time in usec)

minimized by increasing the number of consumers and the $QUEUESIZE$. Experiments show that increasing Q and $QUEUESIZE$ to 10 , which is a high threshold, yields $\sim 50\%$ decrease in queue lag. We further observe that in the case of 3 timers running simultaneously we have more frequent spikes in both directions(high and low). This is totally expected since we employ more threads and the CPU utilization is harder to be equally divided.

4 Discussion

In the precceeded analysis we implemented an architecture for robust Timer functionality. Our main concern is that production systems based on this implementation can be guaranteed invisible delays, especially for high-importance tasks. Towards this end, real time system constraints were considered during implementation accompanied with statistical analysis to validate our claims.

Indeed, drifting time is brought down to a minimum of $3-4usec$ which is a safe threshold even for time-critical tasks. If possible, the spikes near zero could be cancelled out if the mean value of the drifting time could be calculated, maybe from previous runs.

Delay from queue lag is a bit higher$(30-40usec)$ but depending on the hardware constraints, it can be easily reduced if we employ more consumer threads and a larger queue. A more active apporach of minimizing queue lag would be to use a *moving average* filter with fixed window length and substract its value before going to sleep. Finally, emphasis was given to abstraction of the Timer from user interface by asking only for an implementation of the *timerFnc()*. The rest of the mechanism is totally invisible to the user such as producer and consumer threads. It is a plug-and-play architecture were user inputs a function to execute, that could be of arbitrary execution time as seen in **3**, and the desired period and our interface takes care of the rest.