Real Time Embedded Systems

Koro Erika 9707

January, 2024

Introduction

The objective of this assignment is to the construction of timers in a real-time system. The code is written in C and the target platform is the Raspberry Pi Zero with the image provided. The code is cross-compiled on a Linux machine with the **aarch64-linux-gnu-gcc** compiler. The code is then copied to the Raspberry Pi Zero and executed.

Implementation

Specifically, each timer object(thread-producer) will put a task in the queue every T seconds. The thread consumers are created once a timer object is initialized and get activated when a producer-thread puts a task in the queue. The consumers will execute the task and then go back to sleep. When the timer object is destroyed, the consumers will finish their execution and then terminate.

For the proper operation of the code, when all timers have finished enqueueing tasks and all the tasks have been executed,the workers must exit. To accomplish that, the last timer enqueues NULL tasks for all the workers. When a worker receives a NULL task, it exits.

Collecting Execution Times

We are interested in the producer times, which means the time it takes for a timer to enqueue a task and the consumer times, which means the time it takes for a worker to dequeue a task. In order to do that, we use the **gettimeofday()** function which returns the time since epoch in microseconds, as shown in the pseudocode below.

```c
void *producer(void *arg) {

    for (int i = 0; i < t->tasksToExecute; i++) {

        struct timeval start, end;

        // get start time since epoch
        gettimeofday(&start, NULL);

        workFunction in;

        queueAdd(fifo, in);

        // get end time since epoch
        gettimeofday(&end, NULL);

        // save the time it took to enqueue a task
        prodTimes[i] = end - start;
    }
}

void *consumer(void *arg) {

    workFunction w;

    // dequeue a task
    queueDel(a->q, &w);

    struct timeval time_deq;

    // get time in which a task was dequeued
    gettimeofday(&time_deq, NULL);

    // calculate the time that the task was in the queue
   time_in_queue = time_deq - w.timestamp;

}
```

Drifting

In order to eliminate drifting from the Implementation, the time that the task remains in the queue is calculated. Then, it is compared to the period of the timer. If the time that the task remains in the queue is less than the period of the timer, then the producer sleeps for the remaining time.

```

    // Eliminate drifting
    if(t->producerTimers[i] < t->period * 1000){
        usleep(t->period * 1000 - t->producerTimers[i]);
    }
    else{
        // dont sleep at all and increas the drift counter
        t->driftCounter++;
    }
```

Results

The results are presented in four sections, one for each experiment. In each experiment, the mean, the median, the standard deviation and min max are measured for the producer and consumer times. All the tests ran for 1 hour with queue size 50 and 4 consuming threads.

Experiment 1

Producers

In this experiment, only the timer with 10ms period is active. The next graph shows the distribution of the time it took the producer to insert a task in the queue.

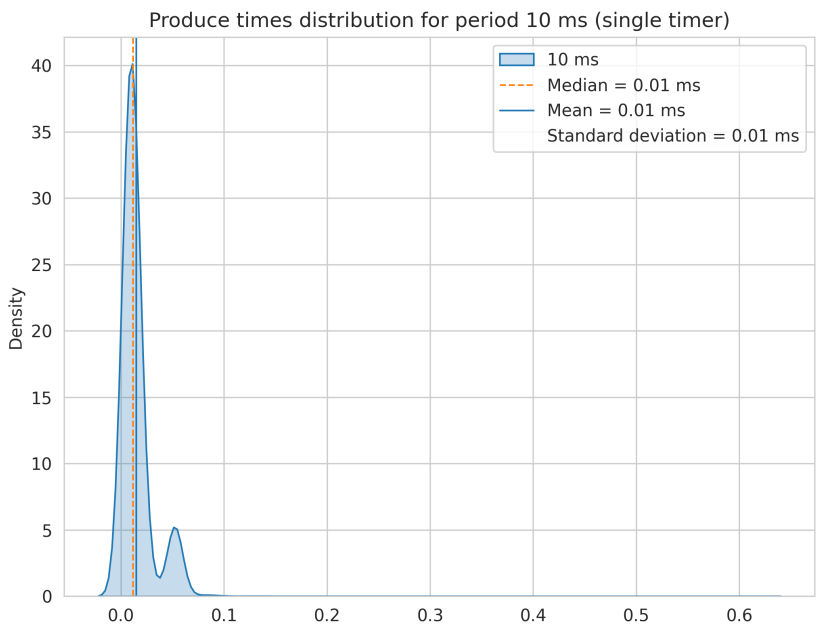

Figure 1: Distribution of execution times for 10ms timer

The mean and the median are very close to each other, which means that the distribution is symmetric. The standard deviation is very small, which means that the values are close to the mean. Overall the timer was able to keep the period of 10ms, without missing any deadlines.

Consumers

On the consumer side, the distribution of the time it took for a worker to dequeue a task is shown below for a single worker. The rest of the workers had similar results.

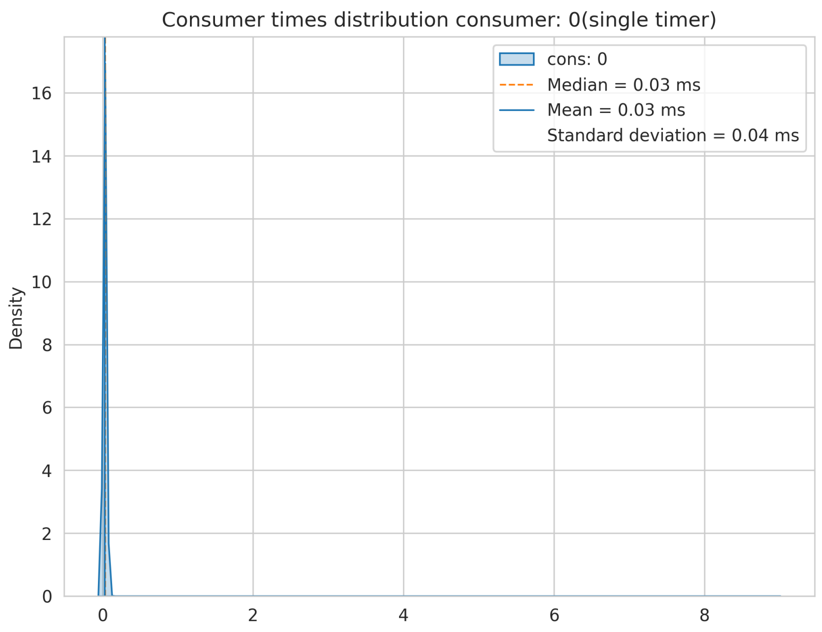

Figure 2: Distribution of execution times for 10ms timer

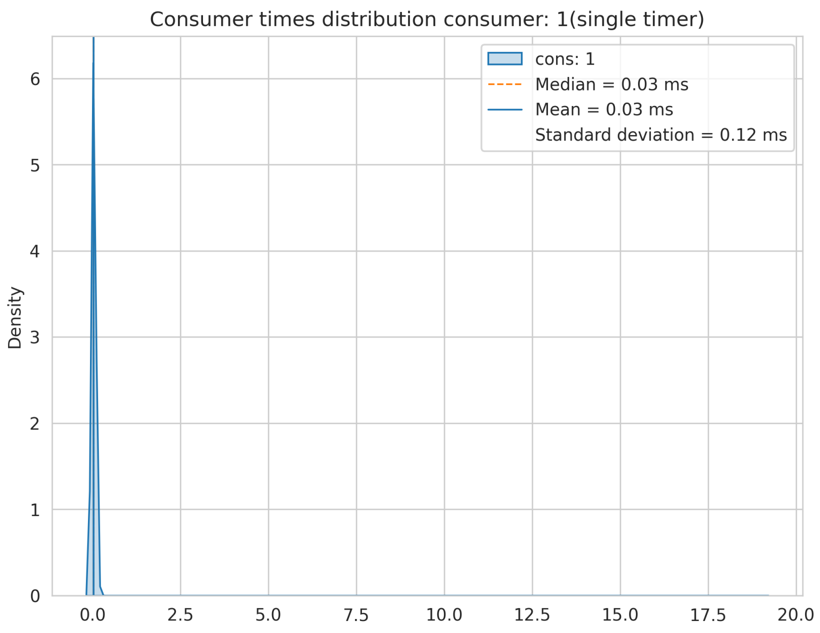

Figure 3: Distribution of execution times for 10ms timer

The consumption time is very consistant suggesting that the consumers are able to keep up with the producers without the queue getting full.

Experiment 2 and 3

In this experiment, the timers with 100ms and 1000ms period are active respectively. The results are very similar to the previous experiment. This is expected since both timers have bigger periods than the previous experiment.

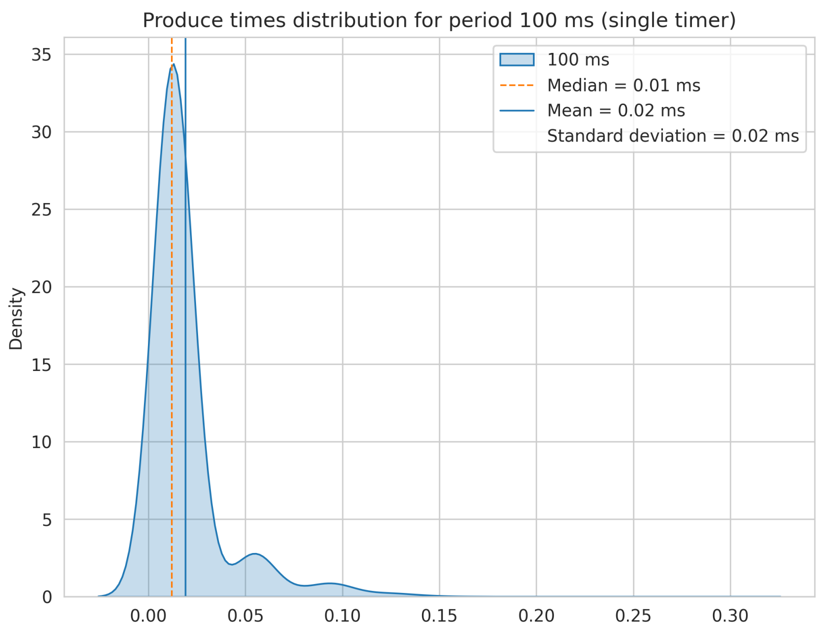

Figure 4: Distribution of execution times for 100ms timer

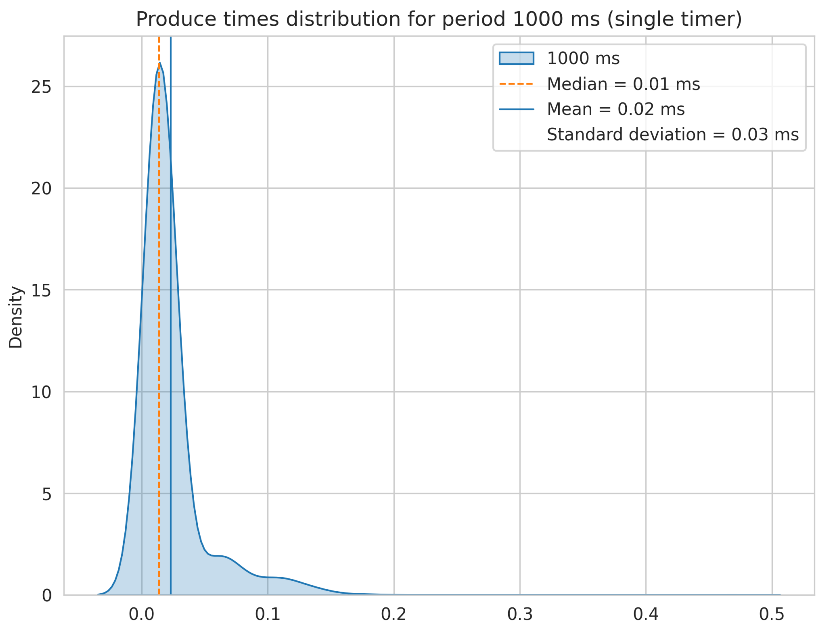

Figure 5: Distribution of execution times for 1000ms timer

Experiment 4

Producers

In this experiment, all the timers are active at the same time. The distribution of the timers is shown below.

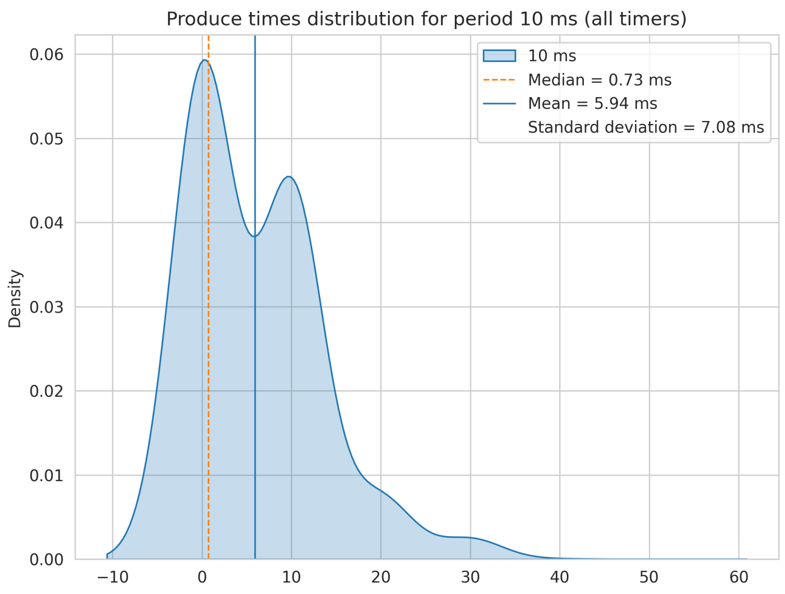

Figure 6: Distribution of execution times for 10ms timer

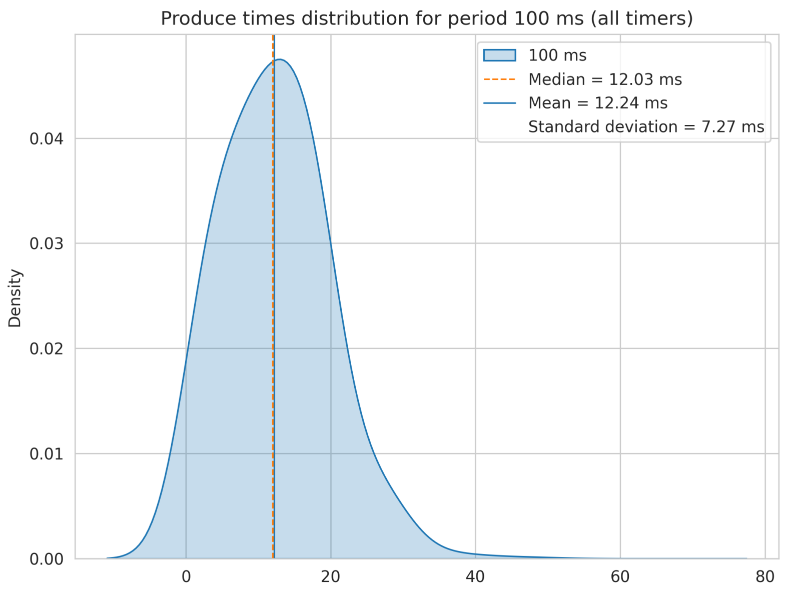

Figure 7: Distribution of execution times for 100ms timer

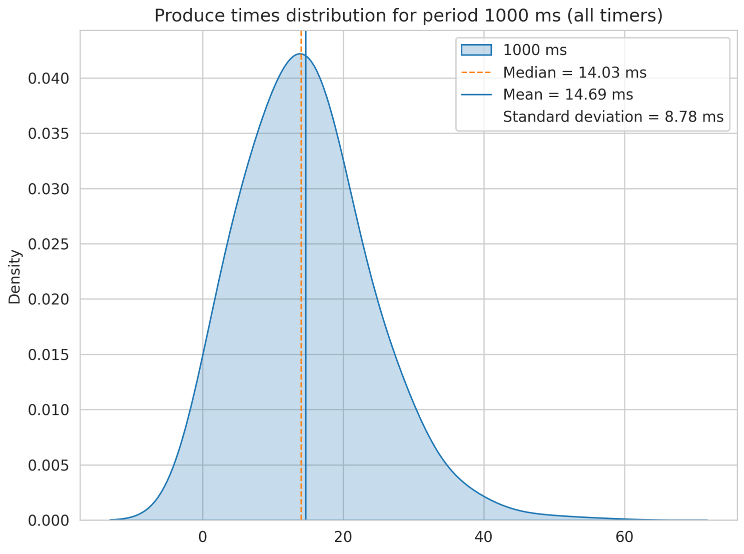

Figure 8: Distribution of execution times for 1000ms timer

The 100ms and the 1000ms period timers are able to keep their period, but the 10ms timer is not. The distribution of the 10ms timer is bimodal, with the first peak at 0.73ms and the second at 10ms. This means that the timer is able to keep its period for the most part, but sometimes it misses the deadline. The median of the timer is 0.73ms, which means that the majority of the tasks are inserted in time.

Consumers

The consumption time distribution for one of the consumers is shown below.

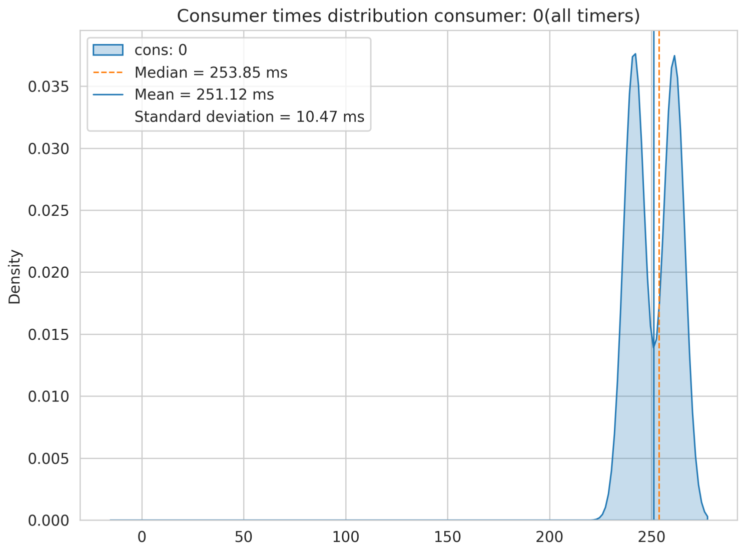

Figure 9: Distribution of execution times for 10ms timer

The distribution is also bimodal, with peaks at 246ms and 264ms. The consumption time is substantially bigger than the previous experiments, which means that the queue is getting full.

Comparison of the results

In this section the experiments 1, 2 and 3 are compared with experiment 4. The diagrams presented below are the same as the ones presented in the previous sections but they are presentred side by side for easier comparison.

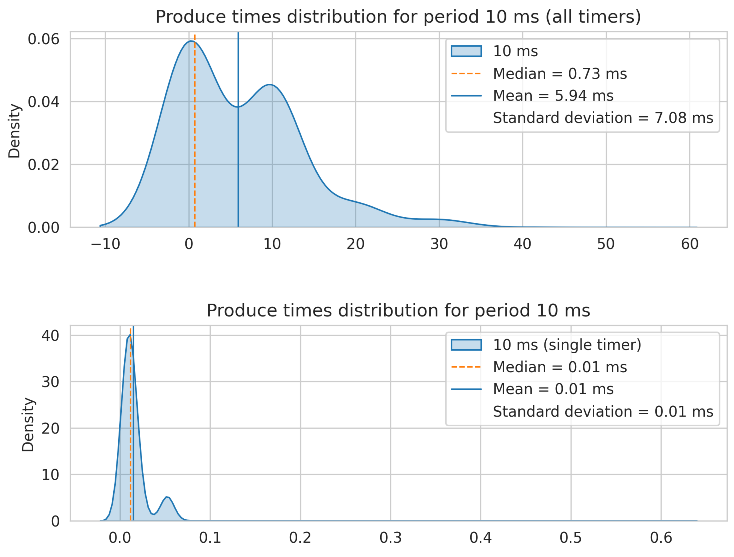

Figure 10: Comparison of 10ms timers

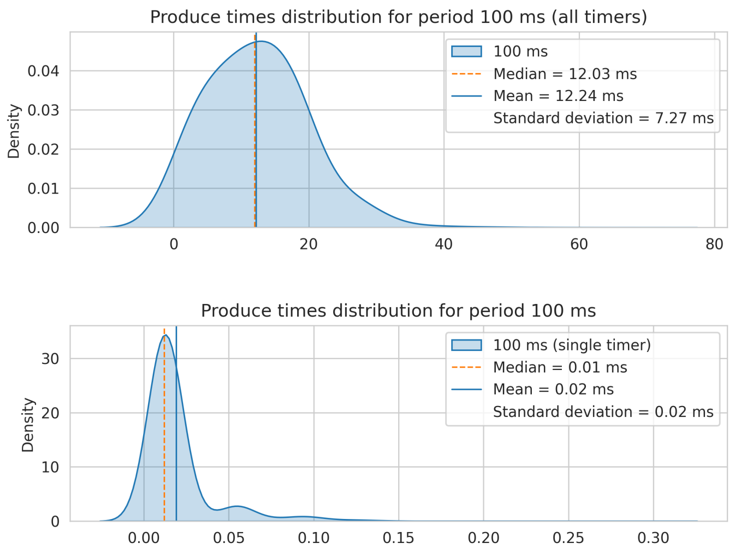

Figure 11: Comparison of 100ms timers

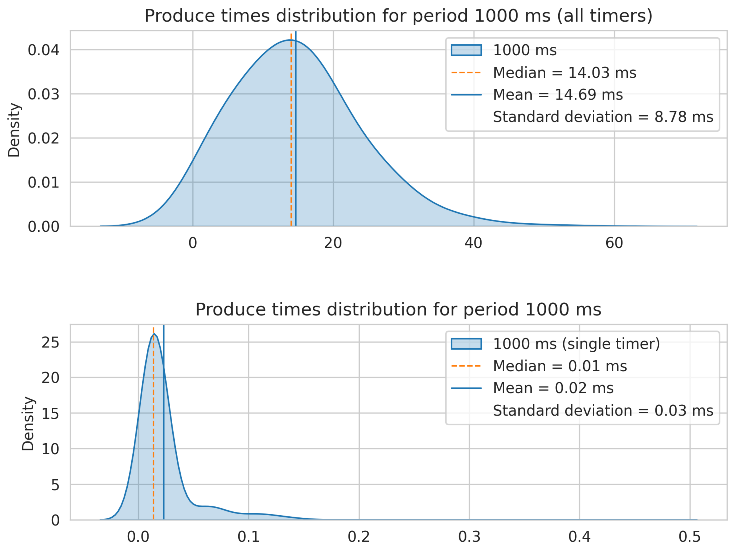

Figure 12: Comparison of 1000ms timers

The comparison of the timers when running alone and when running together with the other timers shows that the consistency of the timers is affected. In all the cases, the standard deviation is bigger when the timers are running together.

This behaviour is normal since the consumer threads are not able to keep up with the producers and the queue is getting full. This means that the producers have to wait for the queue to have space, which increases the enqueue time.

CPU Usage

The average CPU usage was measured during the execution of the experiments. The results are shown below.

Experiment	**CPU Usage**
10 ms timer	1.3%
100 ms timer	0.22%
1000 ms timer	0.03%
All timers	1.79%

Table 1: CPU Usage

Real Time Operation

All the timers are able to run in real time when running alone. When all the timers are running simontaneously, the 10ms timer is not able to keep its period. Increasing the queue size can be helpfull but the best solution is to increase the number of consumers.

Finaly, the time each task takes to execute plays a big role in the real time operation. If the time it takes for a task to execute is bigger than the period of the timer, then the size of the queue will not matter, since given enough time, the queue will get full.