A Timer implementation in a producer-consumer manner

A project for RTES

Christina Koutsou
github: @kchristin22

1 About

The purpose of this project is to implement a timer in the form of a producer-consumer type of problem. The timer for the function we're interested in executing periodically is pushed by the producers in a FIFO queue after each period. The consumers can then pop it out and execute the function. The tests focus on the accuracy of the timer's period depending on the number of timers the producer and the consumer have to handle.

2 Implementation details

Timer: The Timer object is a struct containing parameters that can store useful information, such as the total number of iterations of the timer's period, the time it enters or leaves the queue, its id, the function to execute, etc.

```
typedef struct
{
  __uint32_t Period;            // period of task execution in usec
  __uint32_t TasksToExecute;  // number of tasks to be executed
  __uint8_t StartDelay;         // start executing tasks after a StartDelay
      delay
  void *(*StartFcn)(void *);    // initiate data to be used by TimerFcn
  void *(*StopFcn)(__uint32_t id); // function to be executed after the last
      call of the TimerFcn (TasksToExecute==0)
  void *(*TimerFcn)(void *);    // function to be executed at the start of
      each period
  void *arg;
  void *(*ErrorFcn)(void *, void *); // function to be executed in case the
      queue is full
  __uint32_t id;
  struct timeval *add_queue; // pointer to save the timestamp of adding the
      object to the queue
  struct timeval *del_queue; // pointer to save the timestamp of deleting the
      object from the queue
} Timer;
```

Queue: Another important structure is the one made for the queue. For each timer, the queue contains a separate mutex for the producers. The reason behind this decision lies in the fact that the producer thread currently in execution acquires the queue's mutex and waits for its timer's period to expire to unlock the mutex. This guarantees that no other thread from the same timer attempts to insert an item into the queue prematurely. However, while this procedure is effective in case of one timer or multiple ones sharing the same period, this is not true when dealing with timers with distinct intervals. To be more specific, when the timer with a larger period acquires the mutex, the one with a shorter period cannot access the queue in time, resulting in a significant drift and, thus, introducing notable inaccuracies. In addition to implementing a unique mutex for each timer's producers, there is also a separate mutex for the consumers so they can execute in parallel. Furthermore, an extra mutex that has exclusively to do with queue operations is added to ensure that the access to the queue among all producers and consumers is atomic. To avoid passing the same argument in the queue's initialization and deletion, in order to allocate and, respectively, free the necessary space for the producer mutexes, the number of timers that access the queue is included in the queue's structure.

```c
typedef struct
{
Timer *buf;
long head, tail;
int full, empty, size, num_tasks;
pthread_mutex_t **prod_mut, *queue_mut;
pthread_cond_t *notFull, *notEmpty;
} queue;
```

Main: For the user's convenience, the ability to alter the program's characteristics, is given when the program is called. In more detail, the recognizable arguments and their associated parameter is shown in Table 2.1. The arguments are given in the form `$parameter$=$value$` and the maximum number of timers that can be processed is 3 (though this is easily altered in the c file). The arguments related to the date are relevant to the start of the timer. If any of those arguments is provided, the timer will begin at the specified date and time, and for those missing, a default value will be used instead. These values are equal to the program's start of execution, except for the seconds variable, to which a delay is added to ensure that the program will run successfully in case the seconds parameter is not specified and the arguments point to the current date. Otherwise, the timer begins after 2 seconds from the program's call.

Argument	Program Characteristic
p	No. of producer threads
q	No. of consumer threads
n	Size of queue
t	Period of a timer
y	Year to start timer execution
m	Month to start timer execution
d	Day to start timer execution
h	Hour to start timer execution
i	Min to start timer execution
s	Second to start timer execution

Table 2.1: *Argument list*

In the `main` function, the Timer objects are initialized and distributed among equal producer threads. A distribution based on the timers' period, meaning dedicating more threads as the period decreases, would not make a difference in this implementation as only one producer thread of each timer is executed at a timer's period, making the generation sequential. The consumers can pop any timer out of the queue and execute its function. After all producers join their parent thread, a flag is raised to signal that no more items will be added to the queue and the consumers waiting for the queue-not-empty signal are awaken so they can now exit themselves. The program finishes after all the threads are joined and the queue is deleted.

Producer: The timers are passed by reference in their producer threads so their information is updated for all after each period expires. We will now focus at a single timer's execution. When the first producer acquires the equivalent mutex for the first time, the start function of the timer is called which is responsible for its first execution. The start function is chosen based on the arguments given by the user. Since the function of this timer's task is already executed, it is not added in the queue. The `TasksToExecute` variable is updated and the producer holds the mutex till the period elapses. When the tasks left are less or equal to zero, the producer unlocks the mutex and exits. When each item is added by value to the queue, to avoid any race conditions, the time is noted so we can afterwards calculate its stay in the queue. By saving the time of each producer's iteration we can calculate the thread-safe time interval from its last call and fix this drift by suspending for less time, which in the next iteration would result in a negative drift. In that case, the producer sleeps for the original period. If the drift exceeds the period specified, no delay is added.

```c
long int drift = (1000000 * (start.tv_sec - previous.tv_sec) + (start.tv_usec
    - previous.tv_usec)) - T->Period; // calculate the drift of the timer due
    to the mutex locks
if (drift < 0)
{
  drift = 0; // the drift was fixed previously and the producer was called
      earlier
}
long int sleep = T->Period - drift; // fix the drift
if (sleep > 0)
  usleep(sleep); // add the delay before the next execution of the timer
else
  usleep(0);
```

Consumer: The consumer threads are not only responsible for the timer item's function execution but for the calculation of its time spent in the queue as well. Moreover, it informs of each timer's end provided that the condition `TasksToExecute==0` is verified. Since the queue is of type FIFO and the `TasksToExecute` variable is decremented atomically, it is safe to assume that no other items of the same timer will follow after this condition is met.

3 Tests

The tests with the following arguments were conducted on a Raspberry Pi Zero:

1. t=1.00
2. t=0.1
3. t=0.01
4. t=1 t=0.1 t=0.01

The statistics for both the Timers' drift and time interval between its addition to the queue and its function execution for each test are shown in the tables below.

	Drift(us)	Time interval(us)
Min	0	99
Max	5598	399
Mean	117.09	139.79
Median	113	139
Std deviation	144.18	10.23

Table 3.1: *Test 1: Statistics for timer with period 1s*

	Drift(us)	Time interval(us)
Min	0	93
Max	6199	562
Mean	118.87	136.14
Median	116	135
Std deviation	107.78	16.28

Table 3.2: *Test 2: Statistics for timer with period 0.1s*

	Drift(us)	Time interval(us)
Min	0	89
Max	12744	7274
Mean	123.07	131.57
Median	122	131
Std deviation	127.10	20.69

Table 3.3: *Test 3: Statistics for timer with period 0.01s*

	Drift(us)	Time interval(us)
Min	0	55
Max	0	437
Mean	0	126.22
Median	0	128
Std deviation	0	17.39

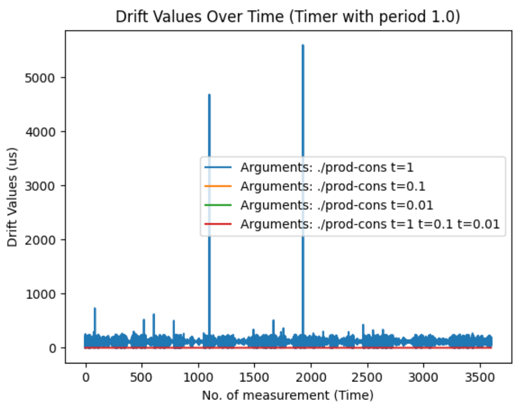
Table 3.4: *Test 4: Statistics for timer with period 1s*

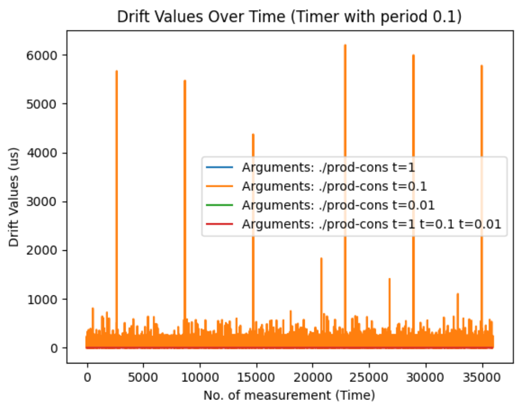
	Drift(us)	Time interval(us)
Min	0	43
Max	0	548
Mean	0	126.32
Median	0	127
Std deviation	0	19.81

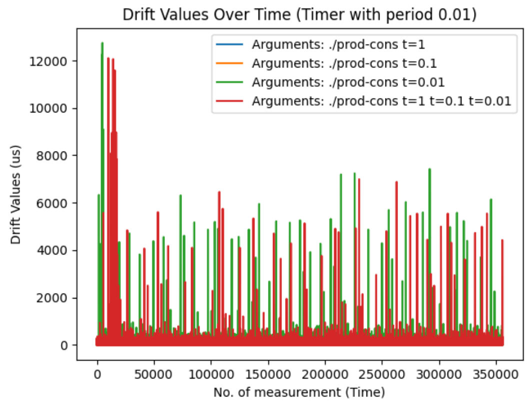
Table 3.5: *Test 4: Statistics for timer with period 0.1s*

	Drift(us)	Time interval(us)
Min	0	42
Max	13800	7266
Mean	125.80	127.99
Median	91	128
Std deviation	329.74	20.18

Table 3.6: *Test 4: Statistics for timer with period 0.01s*







From the above, it's evident that as the period shrinks, the magnitude of the drift between consecutive periods increases. This phenomenon can be attributed to the period lasting less than the OS' task scheduling time slice or finishing early in one, thus leading to longer waiting times. In addition, since the drift is the difference of the time elapsed between the last two executions and the period, subtracting a smaller period value from this time difference will result in a larger value. In case of the fourth test, we observe that only the timer with the shortest period displays a drift from its period. Due to the frequent context switch it induces and the fact that the other periods are a multiple of the smallest one, the rest of the timers are checked on more frequently and, as a result, woken up in time.

Concerning the duration of an item's stay in the queue, the consumers are notified every time an item is added and by that time they're also free to access the queue regardless of the timer's period. Hence, the mean, median and standard deviation values-which are indicative of the parameter's behavior-of all the tests do not differ greatly.

The percentages of CPU and Memory usage for each test are presented in Table 3.7. As expected, a higher timer frequency corresponds to increased resource consumption in terms of CPU time. The memory usage doesn't alter as the queue is static and we have the same amount of threads and variables.

	CPU usage (%)	Memory usage (%)
Test 1	0.3	0.3
Test 2	0.3 & 0.7 (alternating)	0.3
Test 3	3.9 & 4.2 & 4.6 (alternating)	0.3
Test 4	4.2 & 4.6 & 4.9 (alternating)	0.3

Table 3.7: *CPU and Memory usage*

In order to decrease the time interval of a producer and a consumer (the time of an item spent in the queue) to zero and make the system operating at Real-Time, we need to assess the relationship between the number of producer and consumer threads, the duration of the timer's function execution and the size of the queue. To begin with, it's a waste of resources to attribute more than one producer thread to single timer, as the generation by itself is sequential procedure in our case. Consequently, the number of producer threads should be equal to the number of timers-not the amount of periods specified.

For a certain amount of consumers, the duration of the function to be executed periodically can be derived from the following formula:

$$\min\left\{t_{\text{a producer adds an item (period included)}}\right\} \geq (\text{number of consumers} - \text{number of timers}) \cdot t_{\text{function}}$$

The aforementioned formula stems from the fact that a consumer must always be free when an item is added to the queue. The period of this event consists of the time it takes the producer to execute its commands along with the duration it sleeps (defined by the timer's period). To satisfy all the timers, we equate it to the time required for the producer of the timer with the shortest period. Moreover, we consider the worst-case scenario where consumers execute the function sequentially, not in parallel. By leaving out an amount of consumer threads equal to the number of timers, it is guaranteed that at least one consumer thread remains available for each timer during this time interval.

Alternatively, for a specific function to be executed, at least the following should be true:

$$\min\left\{t_{\text{a producer adds an item (period included)}}\right\} \geq t_{\text{function}}$$

To determine the number of consumers to be used in that case, we should consider the fact that the different timers run in parallel and may wish to add an item in the queue simultaneously or during a time interval shorter than the function's duration. Thus, it is advisable that the number of consumers should be at least equal to the amount of timers sharing the same period or having periods that are evenly divided, ensuring once again that at least one consumer is free for each item added in the queue. To minimize the potential overhead associated with an excessive number of threads, the consumer threads to be used should be as many as the lower limit dictates.

As far as the queue is considered, it should be large enough for all items to fit before being removed, in order to constrain the drift of the timer. To calculate the minimum queue size, we need to assess the worst-case scenario of having all the producers run in parallel, which are equal to the number of timers, wishing access to the queue at the same time. Continuing this thought process and assuming that at least one consumer is available for each timer when an item is added, if none of the consumers manages to lock the mutex in between, the queue's size should be equal to the size of the largest set of timers with periods that are evenly divided (e.g. such a set is the {0.01, 0.1, 1}) $(\leq N = \{\text{Number of timers}\})$. This way, we avoid the signaling overhead of having a full queue, without allocating unnecessary space. Also, it's important to note that when the queue is full and the producers waiting, the consumers are free to access the queue and reduce the time of an item passed in the queue. Thus, there's a trade-off between reducing the latter or the drift of the timer.

In reality, the timer's function will still experience some delays due to the time it takes to stash and pop an item, to signal other threads, to unlock the mutex or due to any other running process of the OS. However, when comparing the results of test 1 (t=1, p=6, q=6, n=100) and the ones where the parameters (p, q, queuesize) were adjusted to the values derived from the formulas above for the same timer, we observe a noticeable improvement in the time interval variable in the second case, while the drift has worsened.

The timer's function requires around 20us to finish. The minimum duration of a producer adding an item in the queue was assumed equal to the addition of the min value of the drift (0) and the period of the timer, satisfying the condition for the function's duration. Following the recommendations above and since there is only one timer present here, p=q=n=1.

	Drift(us)	Time interval(us)
Min	0	62
Max	7346	121
Mean	222.4	112.39
Median	106.5	114
Std deviation	938.69	7.07

Table 3.8: *Arguments: t=1 p=1 q=1 n=1, run for 60s*

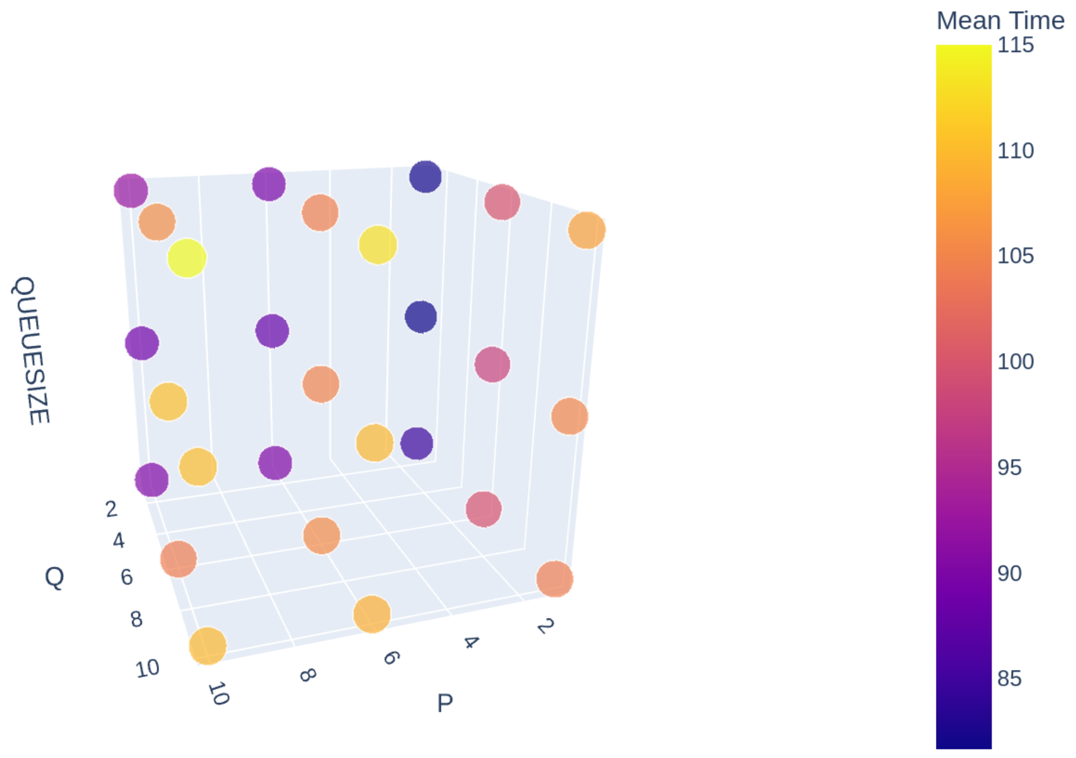


Figure 3.1: *4D plot of the mean time spent in the queue*