Mekonnen Mebratu Baye (Masters Student)
November 2019

Measuring Real-time Performance of Micro-controllers running POSIX based OSs To Perform Schedulability Test of Soft Real-time Applications

Introduction

The aim of this project is to measure real-time performance of micro-controllers running POSIX based nonrealtime/realtime Operating Systems by executing a multi-threaded application. The project uses execution/response time and number of deadline misses as real-time performance testing metrics. Then the test results can be used to perform Schedulability Analysis of a soft real-time application to run it on the tested platform.

The test is performed by running a multi-threaded application which simulates the solar system containing the Sun and the planets on a Linux system. The application uses preemptive FIFO threads which are implemented using the pthread library. The Allegro library is used to design the graphics of the application. This report describes both the implementation of the multi-threaded application and the timing measurements used for real-time performance testing.

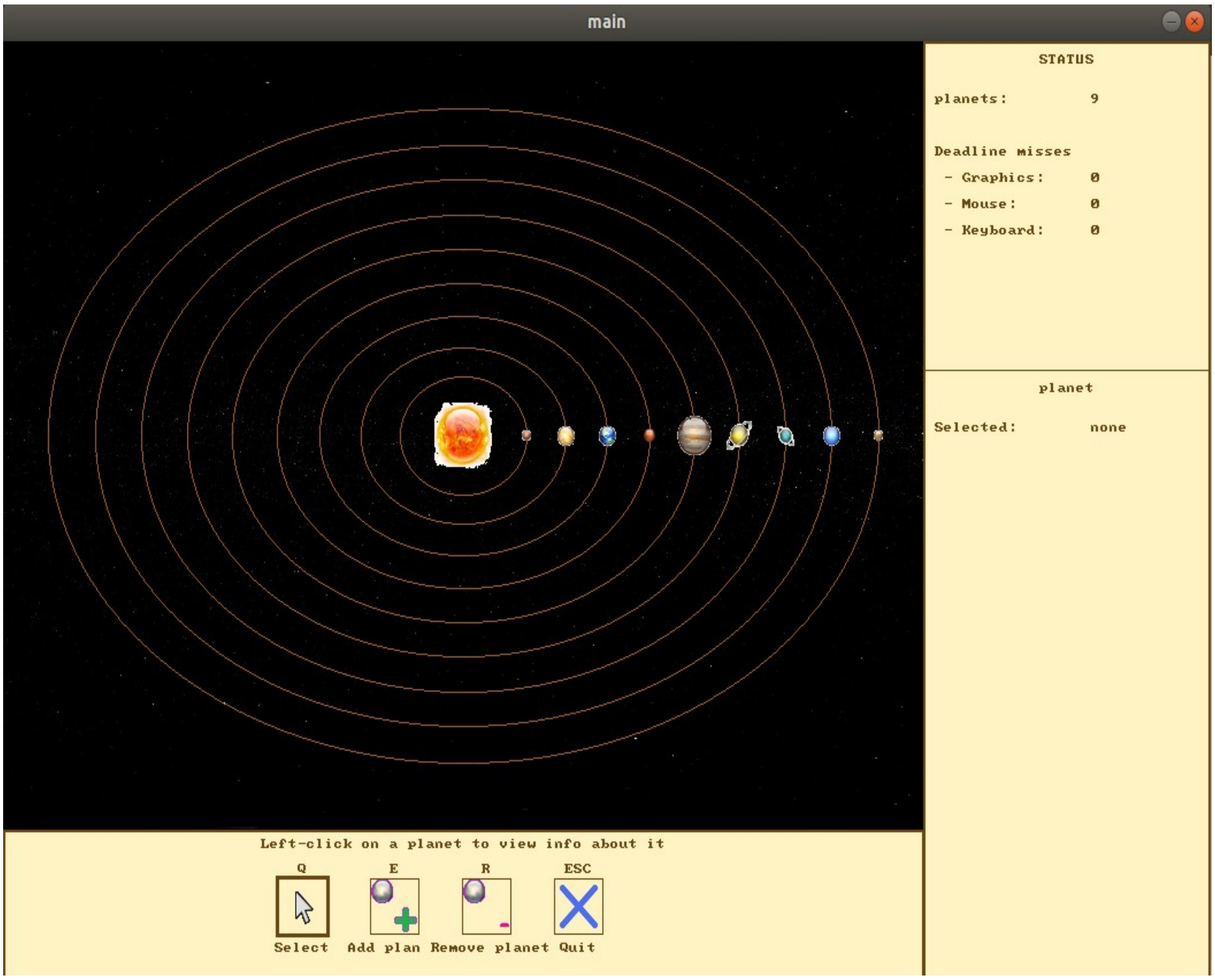
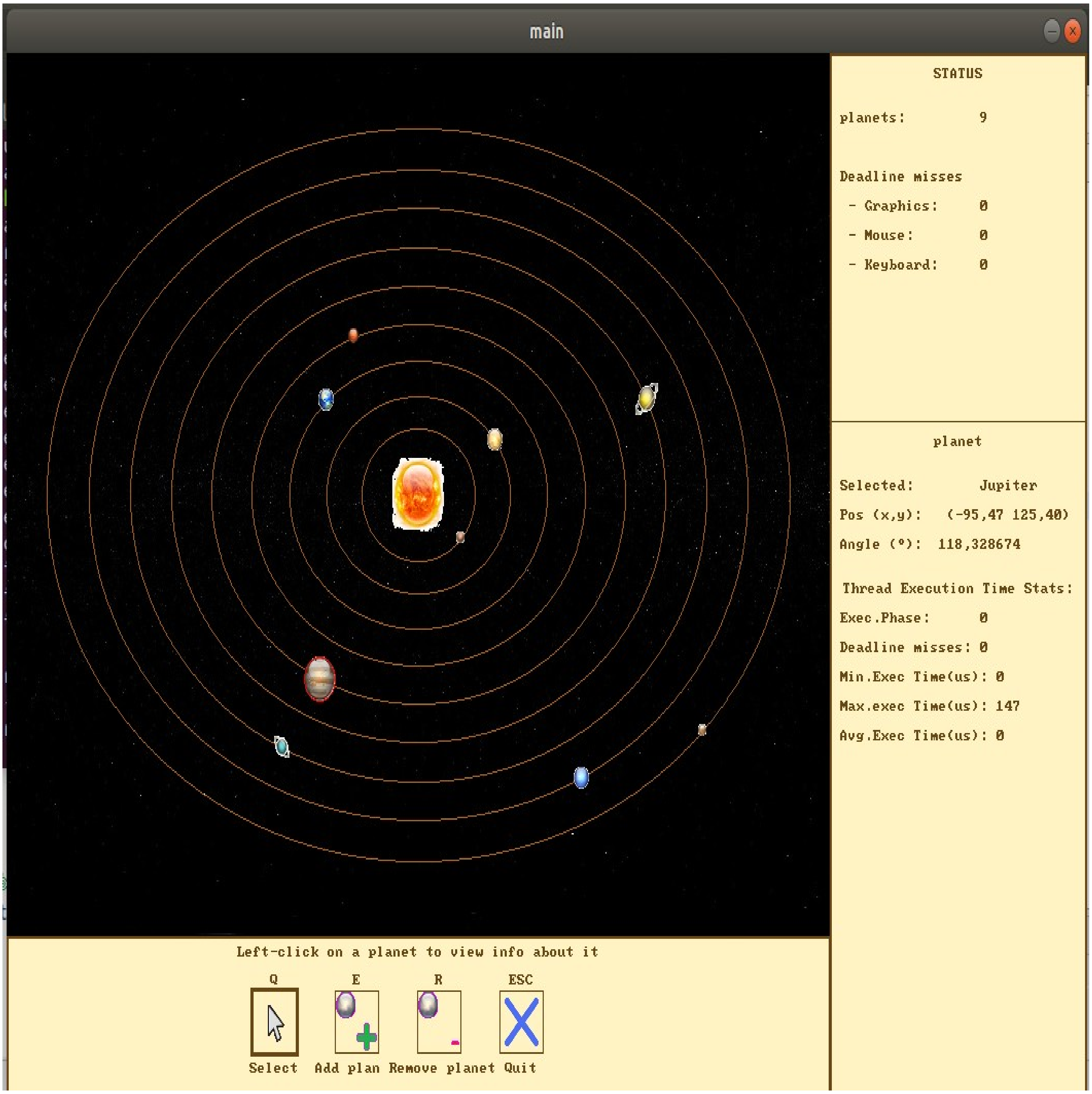


Fig 1: Planets

Part I: Implementation of The Application

The application simulates a number of planets moving around the sun following their elliptical orbiting path. All the parameters of the planet (size, speed,orbit) used to calculate the position of the planet with respect to the sun were taken from a website[1].



```
STATUS

planets:          9

Deadline misses
 - Graphics:      0
 - Mouse:         0
 - Keyboard:      0

planet

Selected:         Jupiter
Pos (x,y):    (-95,47 125,40)
Angle (°):   118,328674

Thread Execution Time Stats:
Exec.Phase:       0
Deadline misses: 0
Min.Exec Time(us): 0
Max.exec Time(us): 147
Avg.Exec Time(us): 0
```

Fig.2: Overview of the multi-threaded Application

1.Description

1.1. Solar Space

The Space is represented as a two-dimensional rectangular box. Every point in the space is identified by a unique pair of natural coordinates (x, y). Note that, the graphics library used(Allegro) supports only 2D graphics and hence the simulation is performed on a two dimensional space.

1.2. Planet

On the solar space are the sun at the center and a number of planets orbiting in an elliptical path. The number of planets on the space can be controlled by the user. Every planet evolves autonomously based on its set of parameters, it starts its movement at its own orbit at **0**° and does a forever counterclockwise orbital rotation with a certain speed.

3. Interface

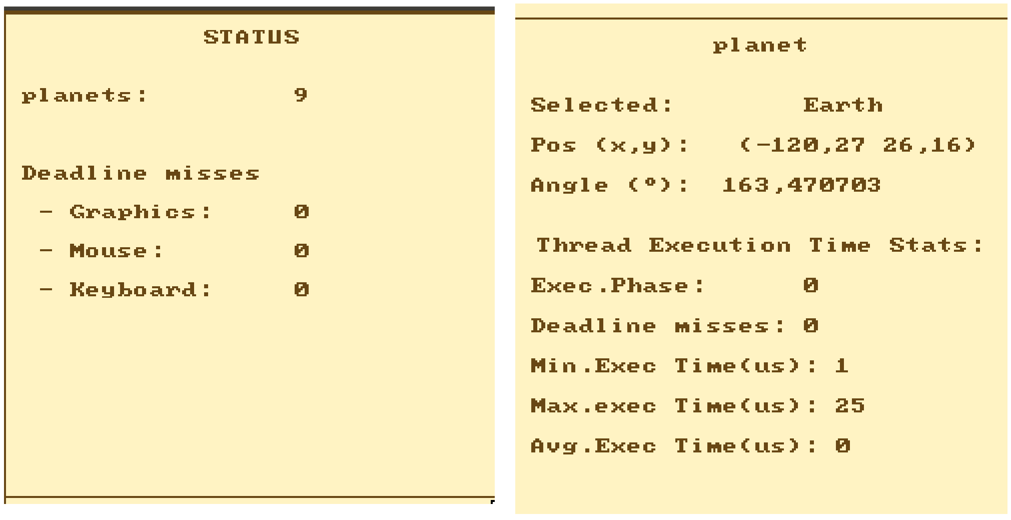
The interface is split into three areas: Solar Space, toolbar and status monitor.

3.1. Solar Space

The space features a night sky black background (Allegro bitmap), on which the Sun(Allegro Sprite) is drawn at the center and the planets (Allegro sprites) moving around on geometric ellipses.

3.2. Status monitor

The status monitor, visible on the right of the window, displays the number of planets and currently being on the space. As well, on it are the number of deadline misses of graphics, mouse and keyboard tasks.(Fig. 3). When the user click on a planet, the main information about the selected planet are plotted on an additional monitor right below: position and orientation on the space, number of deadline misses by its task so far, and its execution time statistics.



```
              STATUS

planets:              9


Deadline misses
  - Graphics:         0
  - Mouse:            0
  - Keyboard:         0
```

```
              planet

Selected:           Earth
Pos (x,y):    (-120,27 26,16)
Angle (°):   163,470703

 Thread Execution Time Stats:
Exec.Phase:         0
Deadline misses: 0
Min.Exec Time(us): 1
Max.exec Time(us): 25
Avg.Exec Time(us): 0
```

Fig. 3: Status Monitor

3.3 Toolbar

The toolbar, visible in the bottom part of the window, allows the user to switch between 4 different modes by pressing the dedicated key (Fig. 4). The button name is always shown below the corresponding icon, whereas a more detailed description about the current mode (including usage details) appears right above the toolbar. Here follows the list of all the possible modes:

- Q [Select]: left-click on a planet to display on the right panel status and statistics about the selected planet itself.
- E [Add planet]: left-click to spawn a new planet from its elliptic orbit at 0°.
- R [Remove planet]: left-click on one planet to remove it (gracefully).
- ESC [Quit]: abort the simulation and close the program.

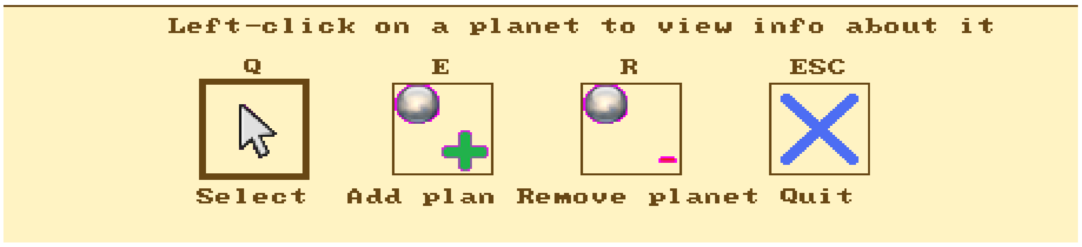


Fig. 4: Tool Bar

4. Code overview

4.1. Programming language and libraries

The application is entirely developed in C, using the libraries Allegro 4.4 [2] and pthread [3]. The project root directory has three sub-folders: ‘*conf*’ contains the configuration files, ‘*img*’ the images used by the application and ‘src’, the most important one, stores all the source files. Two additional folders, bin and build, are created at compilation time and contain object and binary files, respectively.

4.2 Data structures

The state of the entities involved in the application is represented by few data structures. Here is a list of the most important ones:

- planet (declared in planet.h) is the planet descriptor storing all the info about the current state and its constant attributes . Each planet has its own descriptor.
- planets (declared in planet.c) is a container for all the planet descriptors.
- Orbit(declared in planet.h) stores information about the dimensions of geometric elliptic orbits relative to the Sun.
- task_par (declared in rt_thread.h) is the descriptor of any real-time thread, containing information about the routine to execute, arguments and scheduling parameters.

4.3. Tasks

Multiple real-time threads run concurrently during the simulation. Three threads are allocated for refreshing the graphics, handling keyboard and mouse inputs, respectively. Each planet has its own thread, which updates its motion according to the aforementioned model.

- All graphic objects are drawn by a single graphics task
- Keyboard task allows switching between the different modes.
- Mouse task enables interaction between the user and the environment with mouse clicks.
- Planet tasks update the motion and state of the respective planet

The default configuration of scheduling parameters (in milliseconds) is shown in Table 1.

Task	Period	Deadline	Priority
Graphics	50	50	30
Keyboard	200	200	10
Mouse	50	50	20
Planets	20	20	50

Table 1: Tasks parameters

5. Simulation parameters

The user can configure and play around a bunch of parameters that affect the simulation in different ways. The following tables are meant to describe the different parameters and where they are used within the code, along with the range of admitted values and suggested ones.

5.1. Planets

Parameter	Description	Range	Recommended
MAX_PLANETS	Max. number of planets on space	N,proportional to space size	9
STEP_TIME(in seconds)	Controls single angular position step size,i.e angular position= STEP_TIME *speed	Period in seconds. STEP_TIME = TASK_PERIOD/1000	TASK_PERIOD divided by 1000
SIZE	Size of planet	Proportional to the Sun size(baseSize)	

Table 2. plant parameters

5.2. Solar Space

Parameter	Description	Range	Recommended
SPACE_WIDTH	Width of the solar space	N	640
SPACE_ HEIGHT	Height of the solar space	N	480
baseSize	Diameter of the Sun	N	50
incrX	Controls x-radius of the elliptical orbit of planets	N	60
incrY	Controls y-radius of the elliptical orbit	N	40

Table 3: Space Parameters

Part II: 6. Real-time Performance Measurements

6.1 Real Time Scheduler Support

The tasks of the application are implemented using the pthread library with the priority based FIFO scheduling policy in a Linux system, thus the application uses the Real-time (`rt_sched_class`) Linux Scheduler support [4]. Linux implements user threads in one-to-one model in which there is one kernel thread for every user thread[5].

The RT scheduler assigns a priority to every thread to schedule, and processes the threads in order of their priorities. All real-time threads whose priority is `x` are inserted into a linked list headed by `active.queue[x]`. A newly queued thread is always placed at the end of each list of a corresponding priority in the run queue. The first task on the list of the highest priority available is taken out to run. The threads with `SCHED_FIFO` policy can run until they stop or yield. There is nothing to be done every tick interrupt.

Since the platform on which the test application runs has four Intel processors(hyper-threads), it means that there are four **run queues**, one for each processor[6]. Therefore, four sets of different priority tasks can run on the system in parallel(true parallelism). Fortunately, the test application also has four sets of priority tasks, planet task set with highest priority, graphics task with second highest priority, mouse task with third highest and keyboard task is the lowest priority task. Therefore, for example, while planet task set containing nine tasks, one for each planet can be scheduled to execute on one of the processors, the three multimedia tasks; graphics, keyboard and mouse can be executed on the other three processors all in parallel.

IMPORTANT:to use the real-time scheduling policies, the user id of the process must be root. The application must be executed with root privilege.

6.2. Real-time Performance Metrics used

The project uses Execution response time and deadline miss metrics to assess real-time performance of the system(MCU + OS) running the said test application. It should be noted that these are not the only metrics used to perform real-time performance testing;for a system running a real-time OS there are other more test metrics like preemption delay, priority inversion,scheduling latency, etc. To cover all the metrics used to test real-time performance of a real-time OS is beyond the scope of this project.

6.2.1. Execution/response Time Metric

This metric can be used both to measure the response time of the application and to determine scheduling parameters of the tasks of the application that can run on the platform. Measuring execution response time can be used to check if the CPU is actually overloaded by the application tasks, or by overloading the CPU it can be used to determine the kind of real-time application the micro-controller can handle and then to set the limits of scheduling parameters of the task. Moreover if the OS is non-real-time and if deadline misses occur it can be used to check by what rate the task is preempted by higher priority kernel/hardware interrupts and set task parameters accordingly.

6.2.2. Deadline misses

Real-time processing *fails* if not completed within a specified deadline relative to an event; deadlines must always be met, regardless of system load. The project uses deadline miss metric to measure the number of deadline misses of the tasks of the application that are caused by either overloading or preemption by other application tasks or by the kernel itself.

6.3. Test Parameter Setting for the Tasks of the application

The test was performed by running the solar system application which has by default 12 periodic tasks; one for each of the nine planets,one for graphics control, one for keyboard, and one to handle application-user interaction with mouse events. Each task has its own thread,task executor, as explained in the coding section of this report. Each planet task has same scheduling parameters, i.e, same period,deadline and priority executing the same routine code(the code that update planet position) by passing a reference to the respective planet’s parameters as an initial argument. The graphics, keyboard, and mouse tasks have their own scheduling parameters(period, deadline,and priority) set based on the applications multimedia requirements.

For every task its deadline is set equal to its period. The application is not keyboard intensive thus keyboard task has longest period(for example 200 milliseconds). Mouse is used to view the current information about the planet or to remove it from the simulation space by clicking on the planet. Therefore, there can be more frequent mouse clicking events and to handle this a shorter period is required, for instance 50 ms. From physical(or Physics) point of view it may be needed to

know the position of a planet at a very small fraction of time. For this reason, planet tasks are set to have the smallest period in the application in general.
Planet tasks which update position of planets are highest priority tasks, mouse task is the next highest priority task, followed by graphics task and keyboard task which is the lowest priority task.

6.4. Time Measurement and Statistics Primer

CLOCK_MONOTONIC is used to measure the time elapsed between two events. To measure response time, the time elapsed from the start of task routine execution to its end(response), a clock_gettime(CLOCK_MONOTONIC, &t) call was made just before the start of the task specific routine execution code and the same function call was made just after the end of the execution code line of the routine. The CLOCK_MONOTONIC time is converted in to nanoseconds. The difference between the execution start time and the execution end time in nanoseconds is the **response time** of the task in nanoseconds. It was enough to report the response time in microseconds and therefore the time in nanoseconds is divided by 1000, which is the number of nanoseconds in one microsecond. **0 μs** means the response time of the task is in the order of nanoseconds.

```
          exe_start=rt_gettime();

          task→behaviour(task→data);// Execute the task routine
                                      code with task routine param.
          exe_end=rt_gettime();
          risponse_time = exe_end-exe_start;
```

The following is the definition of the user defined function **`rt_gettime()`** in which the pthread **`clock_gettime(CLOCK_MONOTONIC, &ts)`** `call is made and the time is converted into nanoseconds.`

```
nsecType rt_gettime(void)
{
        struct timespec ts;
        nsecType ns;
        int rc;

        rc = clock_gettime(CLOCK_MONOTONIC, &ts);
        if (rc != 0) {
               printf("ERROR: clock_gettime() returned %d\n", rc);
                return 0;
        }

        ts_to_nsec(&ts, &ns);//convert to nanosecond(user defined)
        return ns;
}
```

At the end, before the next activation, the task checks if the current **deadline** has been missed by calling the following 'time_cmp()' function. If the function returns 'true', the number of deadline misses is incremented.

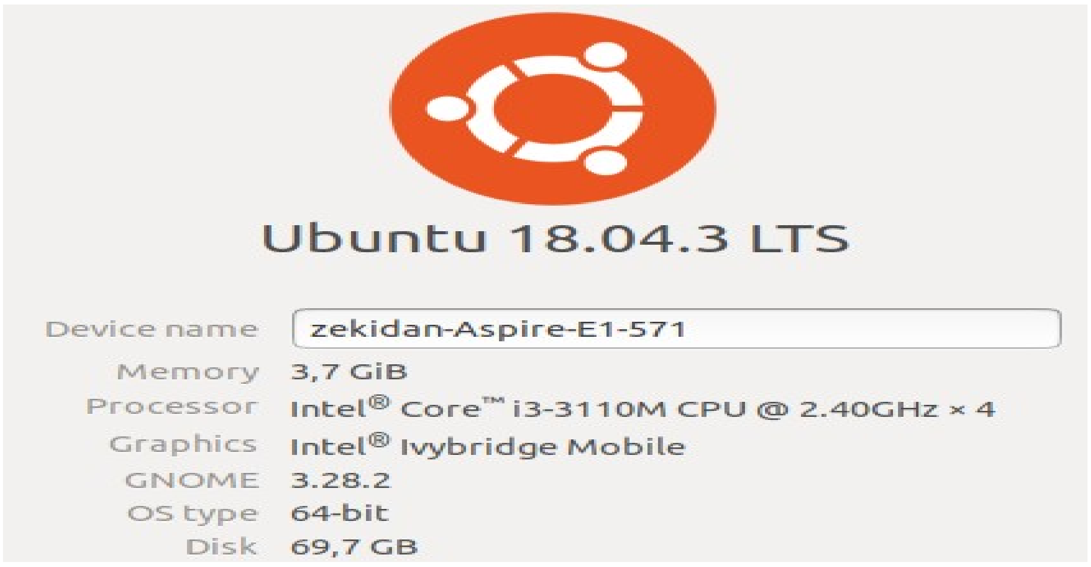
```
bool is_deadline_missed(taskType *const task) {

	struct timespec now;

	clock_gettime(CLOCK_MONOTONIC, &now);
	if (time_cmp(now, task->dl) > 0) {
		++task->dl_missed;
		return true;
	}
	return false;
}
```

After each ten-thousand iteration of the highest priority and shortest period tasks(planet tasks) the minimum, maximum, and average execution/response time statistics were reported for all the tasks. It should be noted that when the planet tasks execute 10,000 iterations, the other longer period tasks may not reach the same number of iterations. Therefore, to calculate the average execution time each task has to maintain its record of number of iterations, then the average is calculated by summing the execution time of the task at each iteration and dividing the total sum by the total number of iterations for the task.
Moreover sum of the minimums, sum of the maximums, and sum of the averages for the planet tasks which have same scheduling parameters (tasks in the same run queue)is reported. This helps to calculate and predict best case, worst case , and average response time of the planet task set respectively.
The total minimum, maximum, and average response time of all the tasks which can be used to determine the best case, worst case, and average response time of the application respectively was also reported. Calculating the total sum can help to analyse the schedulabilty of if the application was to run on a single cpu system.

6.5. Test Set Up

6.5.1 Operating System, C Compiler, and Machine used for testing:

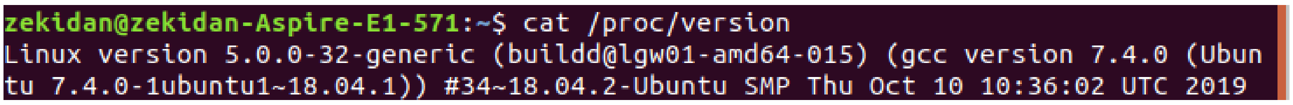
Machine: Acer Aspire (Intel® Core™ i3-3110M CPU @ 2.40GHz × 4), 4GB DDR3 RAM



Ubuntu 18.04.3 LTS

Device name	zekidan-Aspire-E1-571
Memory	3,7 GiB
Processor	Intel® Core™ i3-3110M CPU @ 2.40GHz × 4
Graphics	Intel® Ivybridge Mobile
GNOME	3.28.2
OS type	64-bit
Disk	69,7 GB

Linux version : 5.0.0-32-generic
C Compiler: gcc version 7.4.0



```
zekidan@zekidan-Aspire-E1-571:~$ cat /proc/version
Linux version 5.0.0-32-generic (buildd@lgw01-amd64-015) (gcc version 7.4.0 (Ubun
tu 7.4.0-1ubuntu1~18.04.1)) #34~18.04.2-Ubuntu SMP Thu Oct 10 10:36:02 UTC 2019
```

6.6. Experimental Results

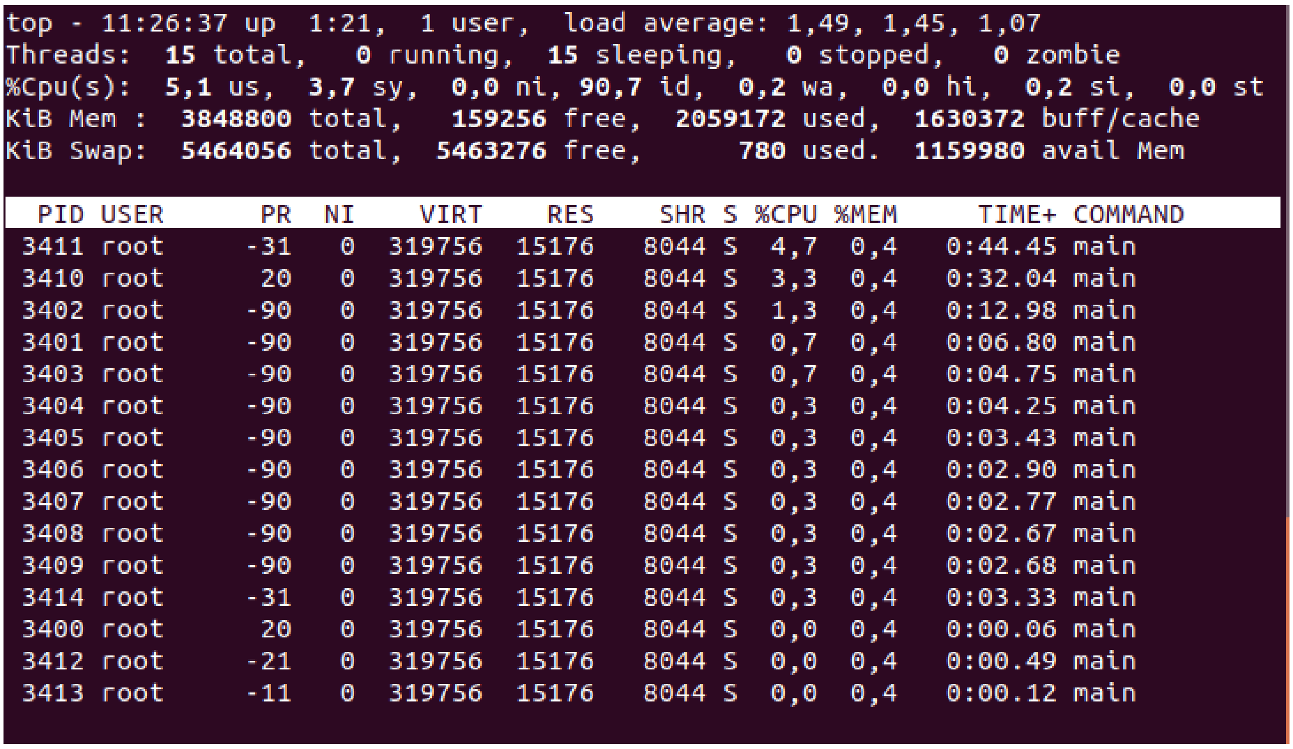
The test was run with different task parameter configurations,by changing period and priority of the tasks different results were obtained.

<u>**Test Param. 1**:</u>

Purpose:check deadline miss by planet task set at lower period(eg. Period<5)

Planet Task Set: Period=4, Priority=89, **Deadline=Period**
Graphics Task: Period=50, Priority=30, **Deadline=Period**
Mouse Task: Period=50, Priority=20, **Deadline=Period**
Keyboard Task: Period=200, Priority=10, **Deadline=Period**

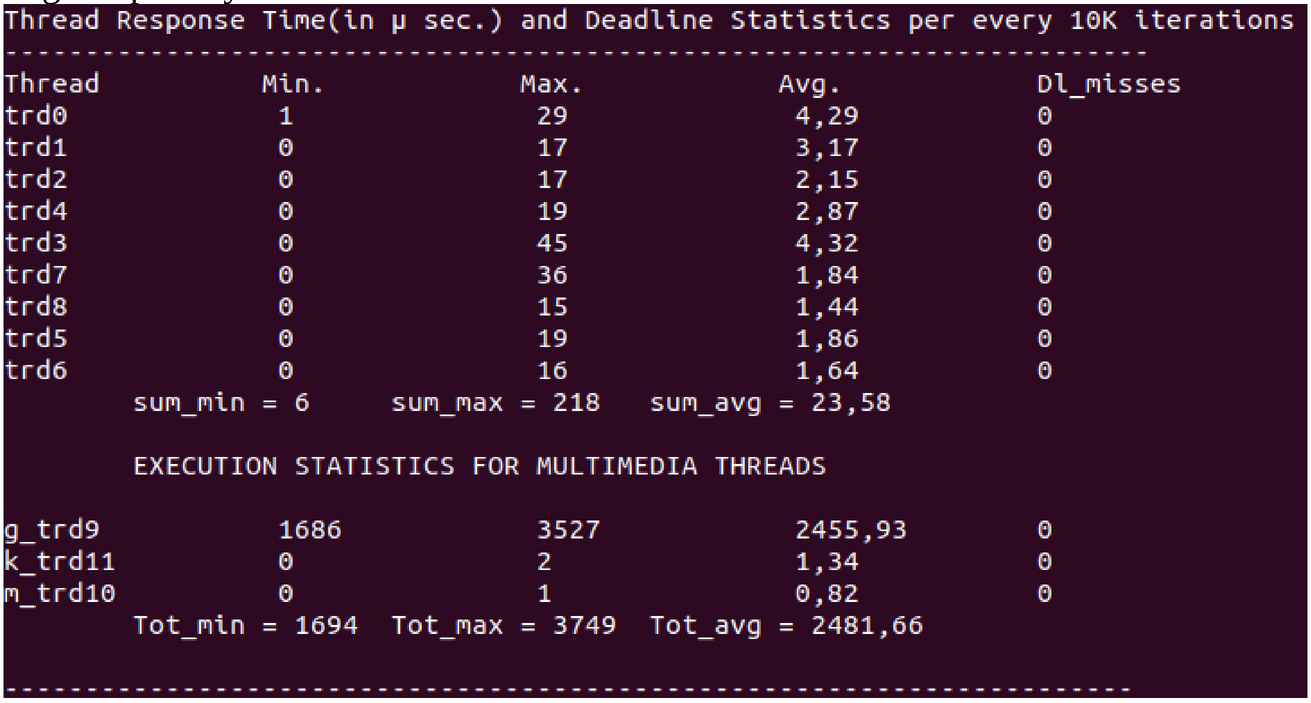
For real time tasks Linux calculates Priority value(PR) of the task using the formula:
PR = -1 - real_time_priority (real_time_priority ranges from 1 to 99).
For example, the real time priority of planet tasks is set to 89, therefor Linux calculates its PR value using the above formula and it is set to -90. Using the ‘top’ command, the following display confirms this.



```
top - 11:26:37 up  1:21,  1 user,  load average: 1,49, 1,45, 1,07
Threads:  15 total,   0 running,  15 sleeping,   0 stopped,   0 zombie
%Cpu(s):  5,1 us,  3,7 sy,  0,0 ni, 90,7 id,  0,2 wa,  0,0 hi,  0,2 si,  0,0 st
KiB Mem :  3848800 total,   159256 free,  2059172 used,  1630372 buff/cache
KiB Swap:  5464056 total,  5463276 free,      780 used.  1159980 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 3411 root     -31   0  319756  15176   8044 S  4,7  0,4   0:44.45 main
 3410 root      20   0  319756  15176   8044 S  3,3  0,4   0:32.04 main
 3402 root     -90   0  319756  15176   8044 S  1,3  0,4   0:12.98 main
 3401 root     -90   0  319756  15176   8044 S  0,7  0,4   0:06.80 main
 3403 root     -90   0  319756  15176   8044 S  0,7  0,4   0:04.75 main
 3404 root     -90   0  319756  15176   8044 S  0,3  0,4   0:04.25 main
 3405 root     -90   0  319756  15176   8044 S  0,3  0,4   0:03.43 main
 3406 root     -90   0  319756  15176   8044 S  0,3  0,4   0:02.90 main
 3407 root     -90   0  319756  15176   8044 S  0,3  0,4   0:02.77 main
 3408 root     -90   0  319756  15176   8044 S  0,3  0,4   0:02.67 main
 3409 root     -90   0  319756  15176   8044 S  0,3  0,4   0:02.68 main
 3414 root     -31   0  319756  15176   8044 S  0,3  0,4   0:03.33 main
 3400 root      20   0  319756  15176   8044 S  0,0  0,4   0:00.06 main
 3412 root     -21   0  319756  15176   8044 S  0,0  0,4   0:00.49 main
 3413 root     -11   0  319756  15176   8044 S  0,0  0,4   0:00.12 main
```

Fig5. Thread statistics of Linux RT scheduler

The following is the Statistics Report for all the tasks after 10K iterations of the highest priority task set:



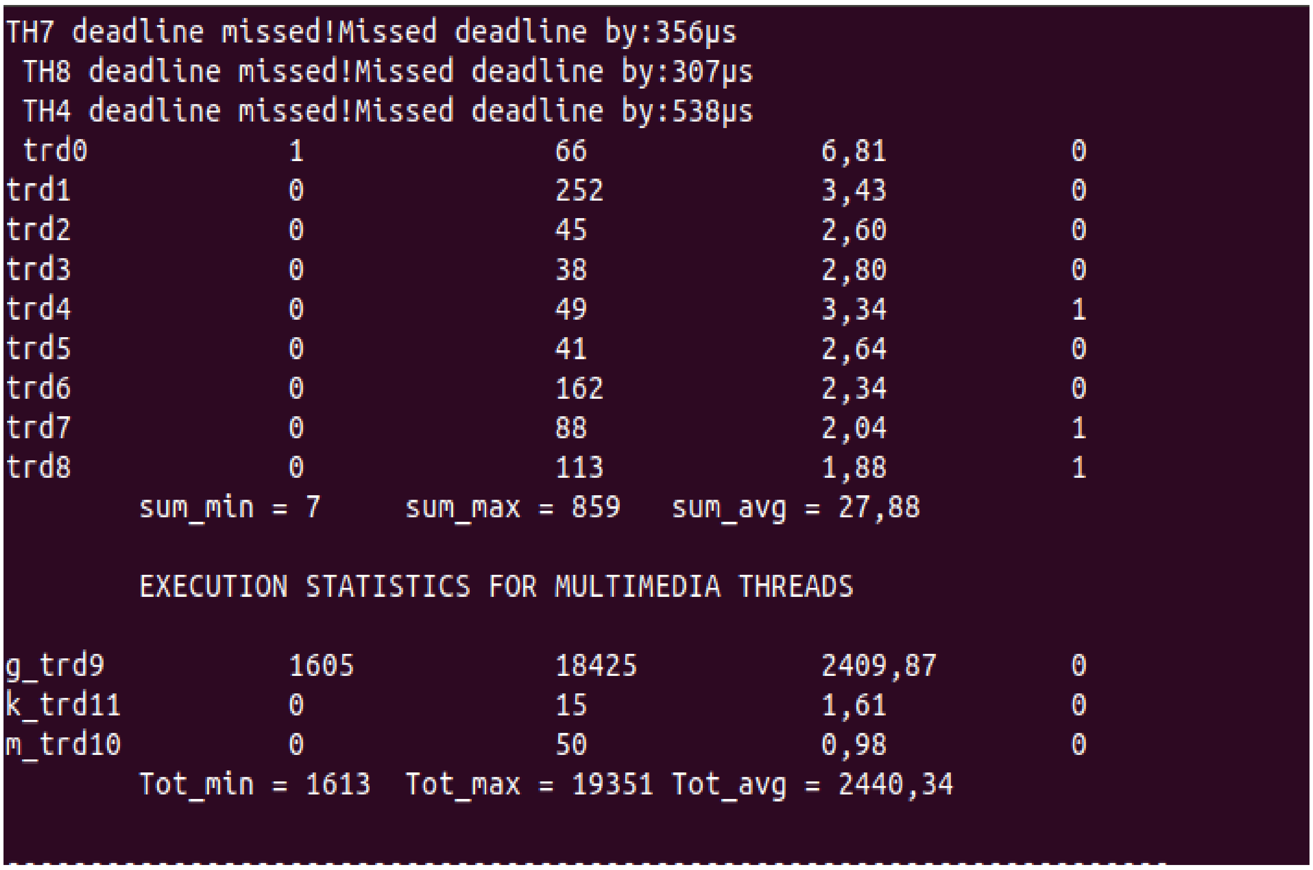
```
Thread Response Time(in μ sec.) and Deadline Statistics per every 10K iterations
----------------------------------------------------------------------
Thread          Min.            Max.            Avg.            Dl_misses
trd0             1               29              4,29            0
trd1             0               17              3,17            0
trd2             0               17              2,15            0
trd4             0               19              2,87            0
trd3             0               45              4,32            0
trd7             0               36              1,84            0
trd8             0               15              1,44            0
trd5             0               19              1,86            0
trd6             0               16              1,64            0
        sum_min = 6     sum_max = 218   sum_avg = 23,58

        EXECUTION STATISTICS FOR MULTIMEDIA THREADS

g_trd9           1686            3527            2455,93         0
k_trd11          0               2               1,34            0
m_trd10          0               1               0,82            0
        Tot_min = 1694  Tot_max = 3749  Tot_avg = 2481,66

----------------------------------------------------------------------
```

***Fig6**. Response Time and Deadline Miss Statistics*

The threads *trd0* through *trd8* are for the nine planet tasks, and *g_trd9*, *m_trd10*, and *k_tr11* are for graphics, mouse and keyboard tasks respectively.

Note: In the above table, since Min and Max response times of the tasks are reported only in microseconds, and sum_min and sum_max are calculated by summing the Min and Max values in nanoseconds respectively and finally dividing the sum by

NS_PER_US(i.e 1000), the Min/Max column sums are significantly different from actual sum_min/sum_max values.
Running the test Application alongside other application programs like Firefox, in the second execution phase(after 10K iterations) of the test app, deadline misses occur.



```
TH7 deadline missed!Missed deadline by:356µs
 TH8 deadline missed!Missed deadline by:307µs
 TH4 deadline missed!Missed deadline by:538µs
 trd0           1               66              6,81            0
trd1            0               252             3,43            0
trd2            0               45              2,60            0
trd3            0               38              2,80            0
trd4            0               49              3,34            1
trd5            0               41              2,64            0
trd6            0               162             2,34            0
trd7            0               88              2,04            1
trd8            0               113             1,88            1
        sum_min = 7     sum_max = 859   sum_avg = 27,88

        EXECUTION STATISTICS FOR MULTIMEDIA THREADS

g_trd9          1605            18425           2409,87         0
k_trd11         0               15              1,61            0
m_trd10         0               50              0,98            0
        Tot_min = 1613  Tot_max = 19351 Tot_avg = 2440,34
```

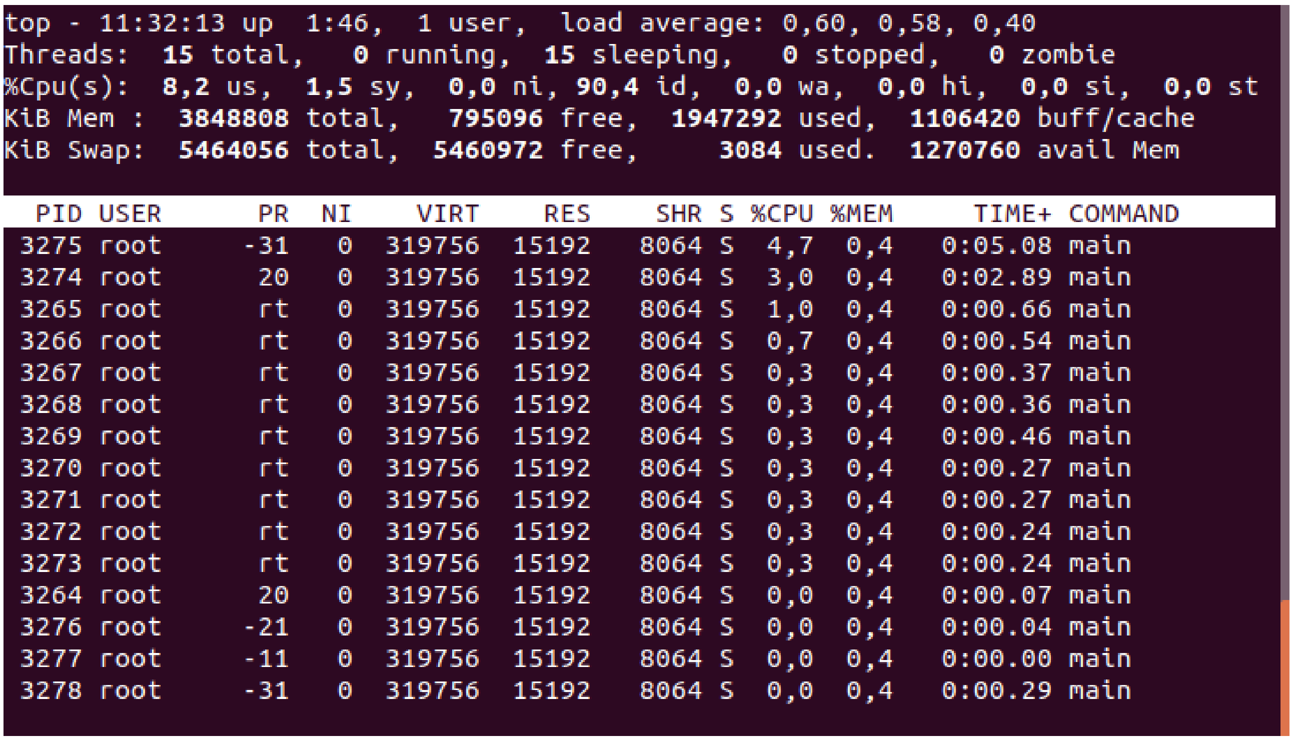
Fig7. Deadline Misses for planet tasks (period 4 ms, priority:89)

For example in the above table the first task (trd0) has two deadline misses. It has max response time *49μs*(measured just after the execution of the task specific code) and it missed the deadline by 538*μs* when it missed its first deadline. Normally the maximum response time should have been greater or equal to 538*μs*. This inconsistency happens because deadline miss is checked outside thread specific execution code and the task is preempted(interrupted) by the kernel (scheduler) after the task has executed its task specific code and before its next activation.
In normal circumstances, when there are no deadline misses, the maximum response time(the sum of max response time of each planet task) of the planet task set *as reported in Fig6* is 218*μs*, *which is by far smaller than the planet's task period, 4 milliseconds.* Therefore, *the deadline misses which occur in the second execution phase(Fig7) are not caused by overloading of the CPU where the planet task set is being executed. Since planet task set is the highest priority task in the application it must be caused by either preemption by other higher priority application or by kernel interrupt.*

To check if the test application was being preempted by the tasks of other higher priority applications in the following task parameter setting the priority of the planet task is set to **99,** which is the maximum real-time priority value in Linux.

Test Param. 2
Purpose: check if deadline misses in the 1st test were caused by preemption of planet task set by other tasks of higher priority applications. To do so, set planet task priority to 99.
Planet Task Set: Period=4, Priority=**99**, **Deadline=Period**
Graphics Task: Period=50, Priority=30, **Deadline=Period**
Mouse Task: Period=50, Priority=20, **Deadline=Period**
Keyboard Task: Period=200, Priority=10, **Deadline=Period**

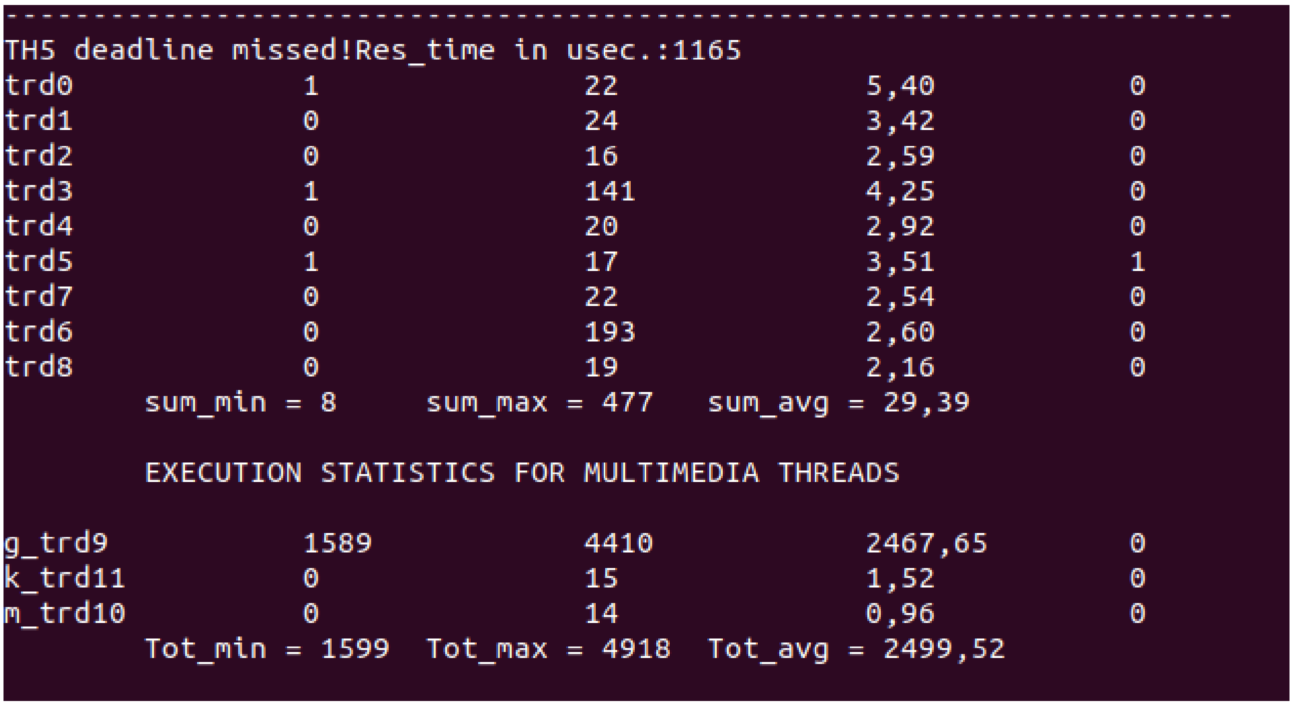
The following display shows threads of the application running in Linux with the above priority setting. The *PR* value ***rt***,which represents the value *-100* is priority of the highest priority planet tasks calculated using the formula
PR = -1 – real_time_priority, *where now* ***real_time_priority = 99.***



```
top - 11:32:13 up  1:46,  1 user,  load average: 0,60, 0,58, 0,40
Threads:  15 total,   0 running,  15 sleeping,   0 stopped,   0 zombie
%Cpu(s):  8,2 us,  1,5 sy,  0,0 ni, 90,4 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem :  3848808 total,   795096 free,  1947292 used,  1106420 buff/cache
KiB Swap:  5464056 total,  5460972 free,     3084 used.  1270760 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 3275 root     -31   0  319756  15192   8064 S  4,7  0,4   0:05.08 main
 3274 root      20   0  319756  15192   8064 S  3,0  0,4   0:02.89 main
 3265 root      rt   0  319756  15192   8064 S  1,0  0,4   0:00.66 main
 3266 root      rt   0  319756  15192   8064 S  0,7  0,4   0:00.54 main
 3267 root      rt   0  319756  15192   8064 S  0,3  0,4   0:00.37 main
 3268 root      rt   0  319756  15192   8064 S  0,3  0,4   0:00.36 main
 3269 root      rt   0  319756  15192   8064 S  0,3  0,4   0:00.46 main
 3270 root      rt   0  319756  15192   8064 S  0,3  0,4   0:00.27 main
 3271 root      rt   0  319756  15192   8064 S  0,3  0,4   0:00.27 main
 3272 root      rt   0  319756  15192   8064 S  0,3  0,4   0:00.24 main
 3273 root      rt   0  319756  15192   8064 S  0,3  0,4   0:00.24 main
 3264 root      20   0  319756  15192   8064 S  0,0  0,4   0:00.07 main
 3276 root     -21   0  319756  15192   8064 S  0,0  0,4   0:00.04 main
 3277 root     -11   0  319756  15192   8064 S  0,0  0,4   0:00.00 main
 3278 root     -31   0  319756  15192   8064 S  0,0  0,4   0:00.29 main
```

Fig8. *Linux rt threads' resources and Statistics of the test Application*

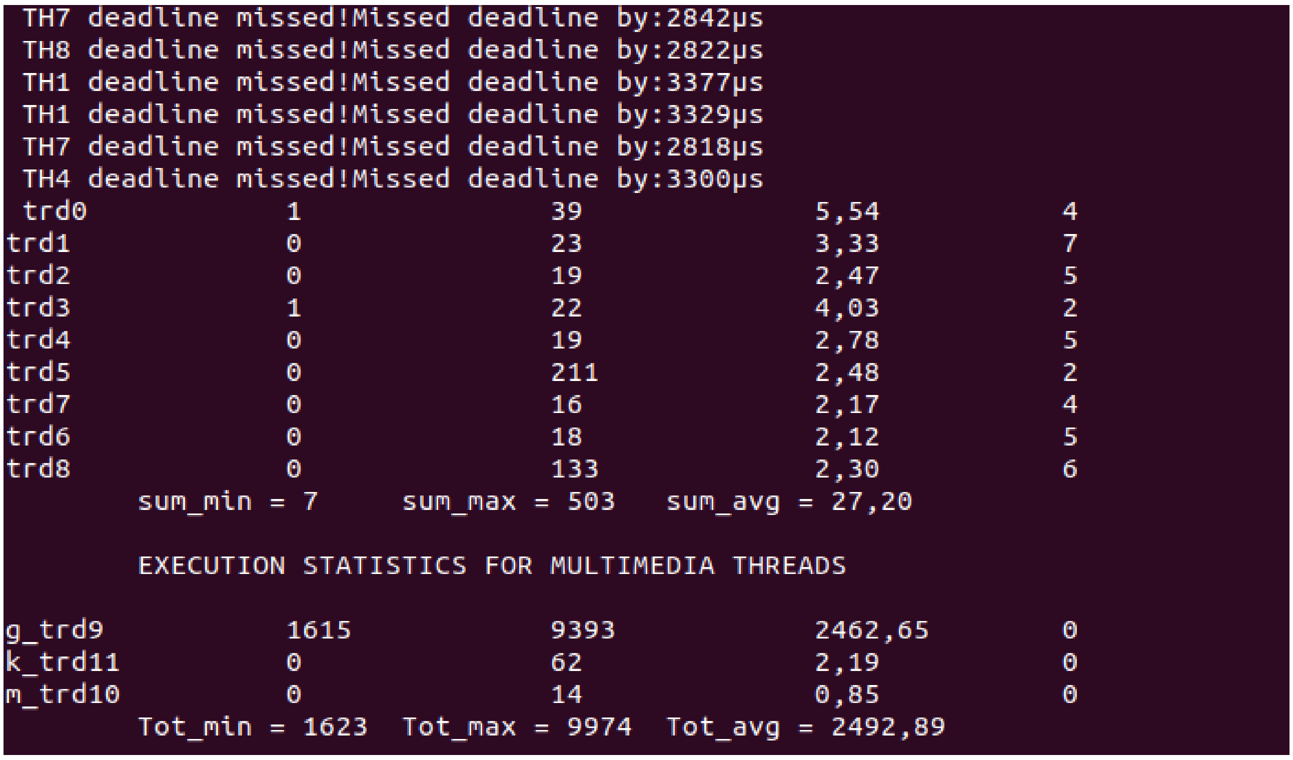
Now with the planet task priority set to the highest(99) possible rt priority, the Linux system was overloaded by running other user applications like Firefox(YouTube Live, and other Live streaming), GNU Image Editor, LibreOffice, etc. An increase in max response time in general and very rare deadline misses occur after many iterations compared to the previous settings test results.



```
----------------------------------------------------------------------
TH5 deadline missed!Res_time in usec.:1165
trd0            1               22              5,40            0
trd1            0               24              3,42            0
trd2            0               16              2,59            0
trd3            1               141             4,25            0
trd4            0               20              2,92            0
trd5            1               17              3,51            1
trd7            0               22              2,54            0
trd6            0               193             2,60            0
trd8            0               19              2,16            0
        sum_min = 8     sum_max = 477   sum_avg = 29,39

        EXECUTION STATISTICS FOR MULTIMEDIA THREADS

g_trd9          1589            4410            2467,65         0
k_trd11         0               15              1,52            0
m_trd10         0               14              0,96            0
        Tot_min = 1599  Tot_max = 4918  Tot_avg = 2499,52
```

Fig9. Deadline miss at priority 99 while other user Apps are running

However, running system utility applications like for instance ‘Wireless Settings’, resulted in immediate and many deadline misses. This behavior was repeated many times, the highest possible priority task is preempted by some utility system software.

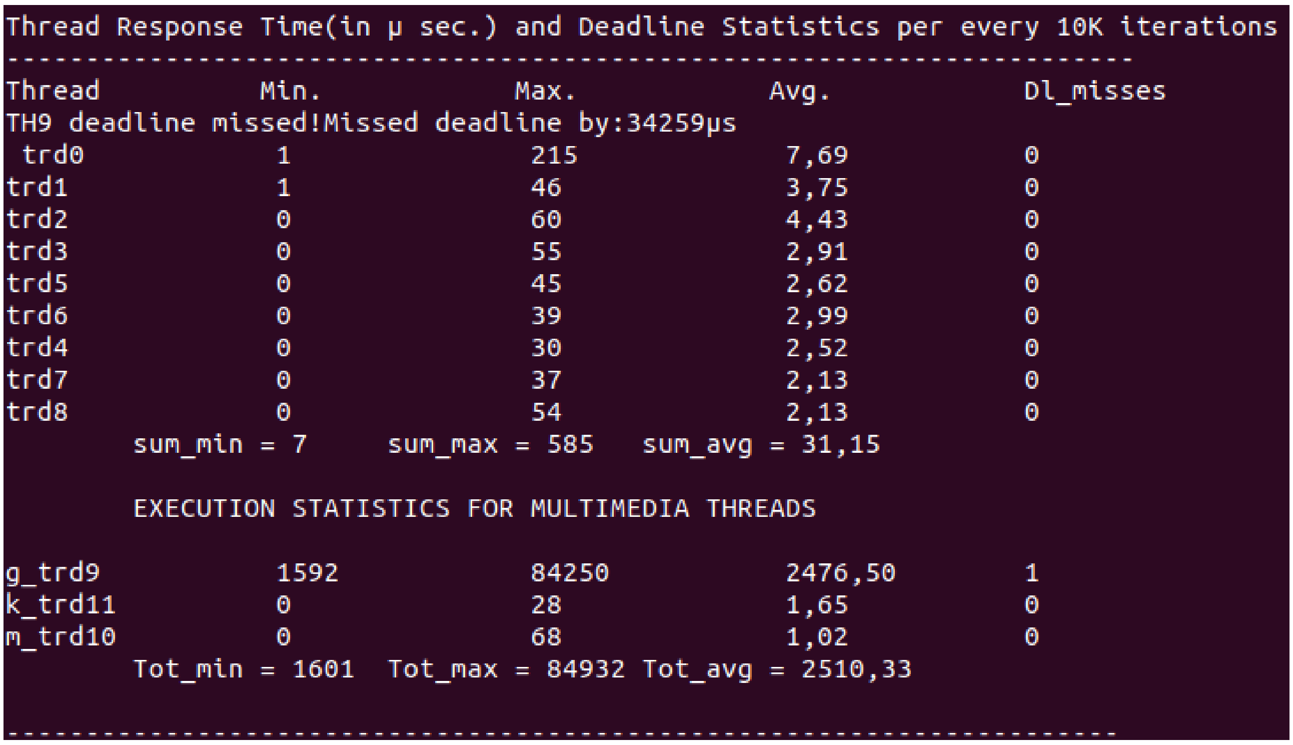


```
 TH7 deadline missed!Missed deadline by:2842μs
 TH8 deadline missed!Missed deadline by:2822μs
 TH1 deadline missed!Missed deadline by:3377μs
 TH1 deadline missed!Missed deadline by:3329μs
 TH7 deadline missed!Missed deadline by:2818μs
 TH4 deadline missed!Missed deadline by:3300μs
 trd0           1               39              5,54            4
trd1            0               23              3,33            7
trd2            0               19              2,47            5
trd3            1               22              4,03            2
trd4            0               19              2,78            5
trd5            0               211             2,48            2
trd7            0               16              2,17            4
trd6            0               18              2,12            5
trd8            0               133             2,30            6
        sum_min = 7     sum_max = 503   sum_avg = 27,20

        EXECUTION STATISTICS FOR MULTIMEDIA THREADS

g_trd9          1615            9393            2462,65         0
k_trd11         0               62              2,19            0
m_trd10         0               14              0,85            0
        Tot_min = 1623  Tot_max = 9974  Tot_avg = 2492,89
```

Fig10.Deadline miss at priority 99 while other System utility Apps are running

Test Param. 3
Purpose: Reschedule the highest priority task with longer period(20 milliseconds) and overload the system running both user and utility applications
Planet Task Set: Period=20, Priority=**99**, **Deadline=Period**
Graphics Task: Period=50, Priority=30, **Deadline=Period**
Mouse Task: Period=50, Priority=20, **Deadline=Period**
Keyboard Task: Period=200, Priority=10, **Deadline=Period**

Result: This time after many iterations the highest priority task didn't miss deadline, however, the lower priority graphics task which is the most CPU bound task missed a deadline by 34259μs as shown in the following Fig.



```
Thread Response Time(in µ sec.) and Deadline Statistics per every 10K iterations
------------------------------------------------------------------------
Thread          Min.            Max.            Avg.            Dl_misses
TH9 deadline missed!Missed deadline by:34259µs
 trd0            1               215             7,69            0
trd1             1               46              3,75            0
trd2             0               60              4,43            0
trd3             0               55              2,91            0
trd5             0               45              2,62            0
trd6             0               39              2,99            0
trd4             0               30              2,52            0
trd7             0               37              2,13            0
trd8             0               54              2,13            0
        sum_min = 7     sum_max = 585   sum_avg = 31,15

        EXECUTION STATISTICS FOR MULTIMEDIA THREADS

g_trd9           1592            84250           2476,50         1
k_trd11          0               28              1,65            0
m_trd10          0               68              1,02            0
        Tot_min = 1601  Tot_max = 84932 Tot_avg = 2510,33

-----------------------------------------------------------------------
```

Fig11. Graphics Task missed deadline

6.7. Concluding Remarks about the Test

Therefore, the conclusion is that the Linux RT scheduler may follow some strict priority based real time scheduling policy among user applications but that can be violated if utility or some other system software part of the OS is to run. Of course the Linux rt scheduler is never designed to implement strict real time requirements.

6.8. Graphical Representation of the Statistics of the Test Results

The following charts show min, max and average response times for CPU bound tasks(planet task set and graphics task), as well as the total application response time (sum:for all the tasks) . Calculating total response time of the app can help to perform schedulability analysis of the real time application if the application was to run on a single processor system.

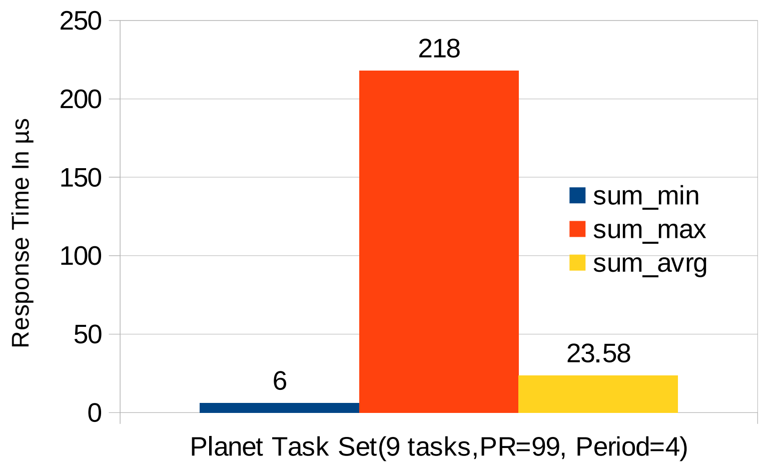


Fig11: Total min, max,and average Execution Response Time for Planet task Set (nine tasks) after 10K iterations

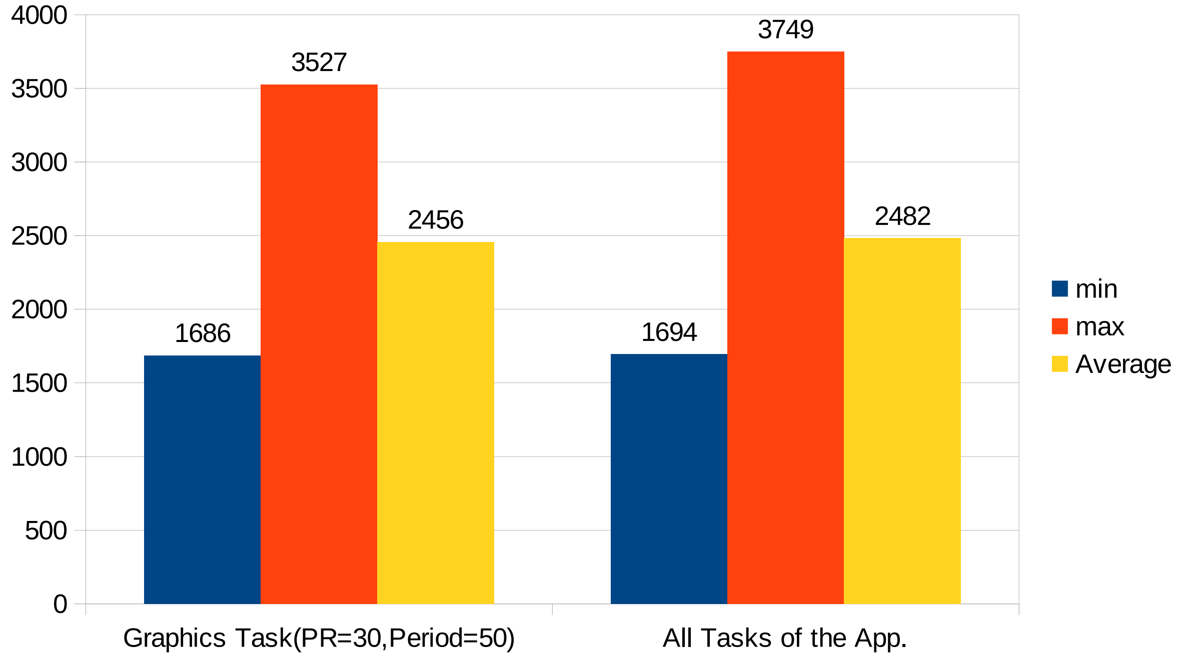


Fig11: Min, Max,and Average Execution Response Time for Graphics Task and All Tasks of the App after 10K iterations

6.9. Final Remark

By adding more performance metrics, the testing procedure and code used in this project can be used to test real-time performance of Micro-controllers with real-time operating systems.

References

(1) **Planets,** http://www.enchantedlearning.com/subjects/astronomy planets/
(2) Wikipedia,Allegro Library, https://liballeg.org/
(3) **Wikipedia,POSIX Threads,https://en.wikipedia.org/wiki/POSIX_T reads**
(4) **Linux Scheduler, https://helix979.github.io/jkoo/post/os-scheduler/**
(5) **Multi-threading Model, https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html**
(6) **Run queue, Linux run queue, https://en.wikipedia.org/wiki/Run_queue**