Real-Time Systems (CESE4025) - Assignment 3

Jiacong Li (5656354, jiacongli@tudelft.nl) Tongyun Yang (5651794, tongyunyang@tudelft.nl)

1 Introduction

This is the third assignment of course CESE4025 Real-Time Systems. In this assignment, three scheduling policies used in Linux kernels are worked with. It is aimed to gain a deep understanding of how these scheduling policies work in operating systems. **In the manual, it mentioned submitting a report of three to four pages. However, that might not be enough for us to include all our figures and the explanation of our observations and understanding. Lots of knowledge and experience in programming Linux kernels' scheduling has been gained, hence we would like to deliver it thoroughly.**

2 Main Questions

2.1 Question 1

Question: Explain in detail how SCHED_FIFO, SCHED_RR and SCHED_DEADLINE[1] work in Linux.

Answer: FIFO is the abbreviation of "First in-first out scheduling". This a very simple method without time slicing. When a FIFO thread is runnable, it will always immediately preempt any currently running threads using normal scheduling policies, e.g. SCHED_OTHER, SCHED_IDLE and SCHED_BATCH. When the following four events happen, the position of FIFO threads in the waiting queue can be moved:

i. A running thread is preempted by another thread of higher priority. In this situation, it stays at the head of the queue for its priority and will resume when threads of higher priority are blocked.

ii. A blocked[2] thread becomes runnable. In this situation, it will be placed at the end of the queue for its priority.

iii. A thread's priority is changed by calling any function[3] that changes the scheduling policy and priority of a thread. In this situation, the thread's position in the queue depends on the direction of the change to threads priority:

   A) If the thread's priority is raised, it is put at the end of the queue for its new priority.

   B) If the thread's priority remains unchanged, its position in the queue is unchanged as well.

   C) If the thread's priority is lowered, it is put at the front of the queue for its new priority.

iv. A thread calls function *sched_yield()*. In this situation, it is will be put at the end of the queue for its static priority.

RR is the abbreviation of "Round-robin scheduling". It is an enhanced version of FIFO with mainly the same scheduling rules, but has a time limit for every currently running thread. Meaning if a RR thread runs longer or equal to the time limit, it will be put at the end of the queue for its priority. The time limit is called the maximum time quantum. In the case of an incomplete thread being preempted, it will complete its unexpired portion of the time quantum. The time interval could be changed with function sched_setparam() or pthread_setschedparam(), and the length of the time quantum can be retrieved using sched_rr_get_interval().

DL is the abbreviation of "Sporadic task model deadline scheduling". This policy is currently implemented using GEDF[4] with CBS[5]. A sporadic task is one that has a sequence of jobs, where each job (or thread in our case) is activated at most once per period. For a DL thread, it has the following parameters:

- Arrival time (or Release time): the time when a task wakes up.
- Start time: the time when a task starts its execution.
- Computation time: the time necessary for executing the task.
- Run time: the time that a task should be executed (this is always greater or equal to computation time).
- Relative deadline: the time period that the task ends its execution.
- Absolute deadline: the time obtained by adding the relative deadline to the arrival time.
- Period: the duration of a task.

When setting DL policy for a thread, users need to specify three parameters (every parameter should be at least 1024): run time (sched_runtime), relative deadline (sched_deadline) and period (sched_period). In addition, the kernel requires that *sched_runtime* ≤

[1]The three scheduling policies will be referred to as FIFO, RR and DL in the report.

[2]Could be blocked by an I/O request, could be preempted by a higher priority thread, or could call function sched_yield().

[3]sched_setscheduler(), sched_setparam(), sched_setattr(), pthread_setschedparam(), and pthread_setschedprio()

[4]Global Earliest Deadline First

[5]Constant Bandwidth Server

sched_deadline $\leq$ *sched_period*. Moreover, the kernel performs an admittance test to ensure deadline scheduling guarantees. A situation where the set of DL threads is not schedulable should be prevented. E.g. the total utilization should be less than or equal to the total number of CPUs available. Hence, DL threads have the highest priority, meaning it will preempt any other scheduling types of threads. Last but not least, to prevent a DL thread from over-running its run time, CBS is used in DL policy.

2.2 Question 2

Question: In order to understand the differences between the FIFO, RR and OTHER scheduling policies, build a task set (by playing with NUM THREADS and periods), schedule the task set with each of these policies, report your observations, compare the resulting schedules with each other, and explain the differences.

Answer: The answer to this question consists of several different subsections. First, the designed tested task set is introduced in detail. Second, additional written code as well as its functions are explained. This includes the technique to release threads periodically and the method to analyze the execution of each thread. Then, the analysis of the task set under three different scheduling policies is given. Comparisons between the actual schedule in Kernel Shark and the hand-drawn version are provided in this subsection. And also, in order to prove some assumptions and rules mentioned in section 2.1, all the threads will not be dropped even if the deadlines are missed.

2.2.1 Task Set Design First, via observing the given code template, it is found that the computation time of whatever created task is static. Meaning that the computation time could not be adjusted in order to differ among different tasks of different priorities[6]. Moreover, in [1] it mentioned several rules of the adjustments for priority when a thread is either preempted or blocked. Hence a naive idea of designing the task set has come up - **the task set should have more than one task per priority, all tasks should be released at the same time and not have the same period, the relative deadline of the shortest task should not be long enough that enables all tasks to complete, the utilization should be less than one to ensure feasibility.** In this case, six tasks of three different priorities have been designed, each thread is killed once it is done releasing the same task five times. Assume the computation time of all tasks is 1 unit of time, then the default parameters of each task are presented as follows:

1. Task0: Priority: 3, Period: 4.5 units of time.
2. Task1: Priority: 3, Period: 5.5 units of time.
3. Task2: Priority: 2, Period: 6.5 units of time.
4. Task3: Priority: 2, Period: 7.5 units of time.
5. Task4: Priority: 1, Period: 8.5 units of time.
6. Task5: Priority: 1, Period: 9.5 units of time.

[6] The higher the value is, the higher the priority represents.

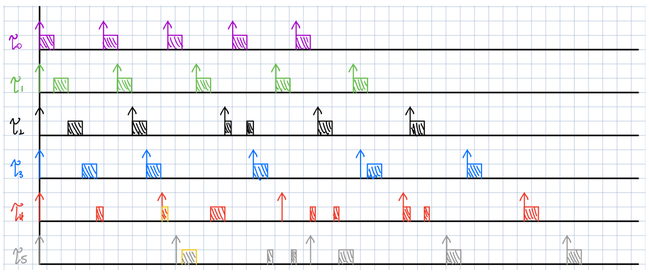


Figure 1: Ideal FIFO scheduling graph (default task set)

The utilization is $\frac{1}{4.5}+\frac{1}{5.5}+\frac{1}{6.5}+\frac{1}{7.5}+\frac{1}{8.5}+\frac{1}{9.5}=0.9141301079$. An ideal hand-drawn FIFO scheduling graph is presented in figure 1. Each color represents tasks executed in each thread concurrently, tasks with a yellow outline indicate the ones that missed their deadline. Note that in order to test different scheduling rules and policies, the task set may be slightly adjusted. However, they would be mentioned when necessary in the following subsections. E.g. the computation time, as well as the period, is adjusted during testing RR. Observations in detail are delivered in the corresponding subsection of each scheduling policy.

2.2.2 Technique & Method Explanation The first issue that requires to be solved is to create threads for the designed tasks mentioned above and release them together at time 0. Assigning values to the corresponding positions in the given arrays named priorities and periods does the job of creating the thread, the remaining process of creating the threads is handled by the provided code. However, the issue that all threads need to be released at time 0 together remains unsolved. Function *clock_nanosleep()* is recommended to be used in the tutorial provided at [2]. With the help of [2], it has been decided to use the combination of CLOCK_REALTIME and the absolute flag indicating that the thread will sleep (or will be released again) until an absolute time in CLOCK_REALTIME has passed. In structure thread_args, a new parameter of type 'timespec' is added called wake_time, it is used to calculate every release time of each thread. In order to minimize the influence of the initialization of the program on the release of threads, all threads are released 10 milliseconds after the program starts. This is realized by getting the time of CLOCK_REALTIME when the program starts. Then add 10000 microseconds to the time achieved using the provided function *timespec_add_us()*. Finally assigning the same value to the wake_time of each created thread. As for releasing threads periodically, it is realized by adding a while-loop and a condition of tasks_count[7] < PERIOD[8]. Function *clock_nanosleep()* is put at the very beginning of the while-loop, due to the fact that the next release time is incremented each iteration. Hence, the threads

[7] The variable which keeps track of the number of tasks executed.

[8] The global variable which indicates the number of tasks ready to be executed

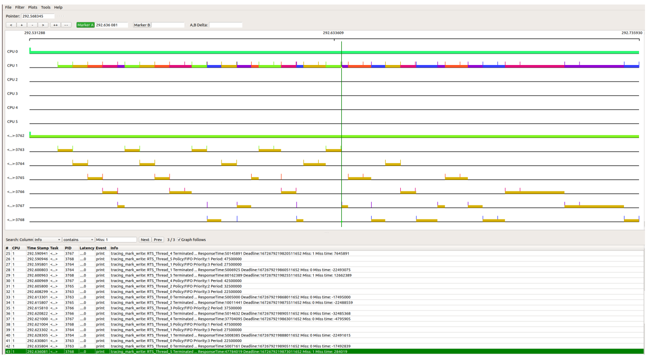
could only be released when CLOCK_REALTIME is equal to each corresponding release time.

After executing the task of each thread, the deadline and response time are calculated. The deadline is calculated by adding each release time to the thread's corresponding period (calculated in the unit of nanoseconds). The response time is calculated using the provided function *clock_diff()* between the current release time and the time the task finishes execution. It also displays the result in the unit of nanoseconds. Last but not least, the detection for deadline misses is implemented with the help of [2]. A variable miss_flag is added, and its values are assigned to 1 if the current deadline is missed and 0 if not. The deadline detection is to compare the difference between the end time of the current task and the next release time. If the current task ends before releasing the next, then it meets its deadline and vice versa. Note the two *trace_write()* functions are placed right before and after the workload. In this case, they could help determine the overhead. In the manual, it is recommended to dump all measured data into a file for analysis. Though it is preferable to analyze each task and thread in Kernel Shark with the help of miss_flag, calculated response time and deadline, this is realized and used when necessary. The function is realized with two global arrays which kept note of every calculated response time and deadline, and then dump all the data into a txt file before the main function ends. Note that both global arrays are also freed in the end.

2.2.3 Discussion on FIFO After successfully building the code and generating the trace file with the nop flag [9]. The file is opened with Kernel Shark, and the traces are displayed in figure 2. Despite the mistaken traces caused by Kernel Shark itself, figure 1 is exactly the same as 2. However, it is spotted that there are only two tasks that missed their deadline in the earlier figure, while there are three in the later one. After investigation, it is found that it is caused by the error of execution time of function *workload()*, and there is no solution to it, since it is a provided function. **Similar situations would also occur in other scheduling policies**.

Moreover, some interesting observations are reported below, they confirm some guesses while manually drawing figure 1. First, in the second period of thread 4 where the first task misses its deadline. Task 4 did not continue its execution after missing the deadline, and task 5 was executed instead. **This confirms the guess of "A missed-deadline task blocks its corresponding newly launched task and the position in the waiting queue for the priority is moved"**. Next, the sequence creating the tasks for the same priority is exchanged. Figure 3 shows that the scheduling policy is different from EDF[10] or RM[11]. The rest follows the description written in [1].

[9]The nop flag does not affect which events are traced, it simply indicates that a particular trace event should be treated as a no-op when it is recorded in the trace file.
[10]Earliest deadline first scheduling
[11]Rate-monotonic scheduling



```
Search: Column Info  contains  Miss: 1  Next  Prev  3 / 3  ✓Graph follows
#  CPU  Time Stamp Task  PID   Latency Event  Info
25 1    292.590941 <...> 3767  ....0   print  tracing_mark_write: RTS_Thread_4 Terminated ... ResponseTime:50145891 Deadline:1672679219820511652 Miss: 1 Miss time: 7645891
26 1    292.590946 <...> 3768  ....0   print  tracing_mark_write: RTS_Thread_5 Policy:FIFO Priority:1 Period: 47500000
27 1    292.595801 <...> 3764  ....0   print  tracing_mark_write: RTS_Thread_1 Policy:FIFO Priority:3 Period: 27500000
28 1    292.600803 <...> 3764  ....0   print  tracing_mark_write: RTS_Thread_1 Terminated ... ResponseTime:5006925 Deadline:1672679219860511652 Miss: 0 Miss time: -22493075
29 1    292.600963 <...> 3768  ....0   print  tracing_mark_write: RTS_Thread_5 Terminated ... ResponseTime:60162389 Deadline:1672679219825511652 Miss: 1 Miss time: 12662389
30 1    292.600969 <...> 3767  ....0   print  tracing_mark_write: RTS_Thread_4 Policy:FIFO Priority:1 Period: 42500000
31 1    292.605800 <...> 3765  ....0   print  tracing_mark_write: RTS_Thread_2 Policy:FIFO Priority:2 Period: 32500000
32 1    292.608299 <...> 3763  ....0   print  tracing_mark_write: RTS_Thread_0 Policy:FIFO Priority:3 Period: 22500000
33 1    292.613301 <...> 3763  ....0   print  tracing_mark_write: RTS_Thread_0 Terminated ... ResponseTime:5005000 Deadline:1672679219868011652 Miss: 0 Miss time: -17495000
34 1    292.615807 <...> 3765  ....0   print  tracing_mark_write: RTS_Thread_2 Terminated ... ResponseTime:10011441 Deadline:1672679219875511652 Miss: 0 Miss time: -22488559
35 1    292.615810 <...> 3766  ....0   print  tracing_mark_write: RTS_Thread_3 Policy:FIFO Priority:2 Period: 37500000
36 1    292.620822 <...> 3766  ....0   print  tracing_mark_write: RTS_Thread_3 Terminated ... ResponseTime:5014632 Deadline:1672679219890511652 Miss: 0 Miss time: -32485368
37 1    292.621000 <...> 3767  ....0   print  tracing_mark_write: RTS_Thread_4 Terminated ... ResponseTime:37704095 Deadline:1672679219863011652 Miss: 0 Miss time: -4795905
38 1    292.621004 <...> 3768  ....0   print  tracing_mark_write: RTS_Thread_5 Policy:FIFO Priority:1 Period: 47500000
39 1    292.623302 <...> 3764  ....0   print  tracing_mark_write: RTS_Thread_1 Policy:FIFO Priority:3 Period: 27500000
40 1    292.628305 <...> 3764  ....0   print  tracing_mark_write: RTS_Thread_1 Terminated ... ResponseTime:5008385 Deadline:1672679219888011652 Miss: 0 Miss time: -22491615
41 1    292.630801 <...> 3763  ....0   print  tracing_mark_write: RTS_Thread_0 Policy:FIFO Priority:3 Period: 22500000
42 1    292.635804 <...> 3763  ....0   print  tracing_mark_write: RTS_Thread_0 Terminated ... ResponseTime:5007161 Deadline:1672679219890511652 Miss: 0 Miss time: -17492839
43 1    292.636081 <...> 3768  ....0   print  tracing_mark_write: RTS_Thread_5 Terminated ... ResponseTime:47784019 Deadline:1672679219873011652 Miss: 1 Miss time: 284019
```

Figure 2: FIFO scheduling graph in Kernel Shark (default task set)

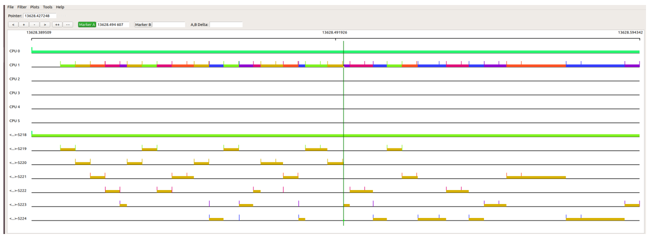


Figure 3: FIFO scheduling graph in Kernel Shark (default task set priority exchanged)

2.2.4 Discussion on RR In order to test with RR, the execution time should at least be greater than the time quantum. It could be achieved via function *sched_rr_get_interval()*. In our case, it is found that the time quantum is 100 milliseconds by default. Hence, it is reduced to 4 milliseconds with the line of code provided in the assignment manual. Figure 4 is the graph of traces in Kernel Shark.

It is observed that the behavior of RR is different from FIFO, but also obeys similar rules at the same time. These interesting observations exactly match the description in section 2.1. The graph of traces is shown in figure 2. However, there are also behaviors out of expectation. Task-switch sometimes occurs in a time shorter than 4 milliseconds. The reason behind this remains unknown, however, an immature guess to the cause of this issue is that there are more factors in the Linux scheduler that have influenced the scheduling of tasks. It is also observed that huge overhead is caused due to regular task switches resulting in longer response time. This is found by calculating the average response time of all tasks. A larger response time would have a potential risk of causing tasks that

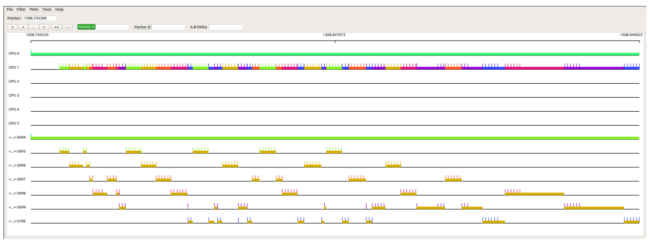


Figure 4: RR scheduling graph in Kernel Shark (default task set)

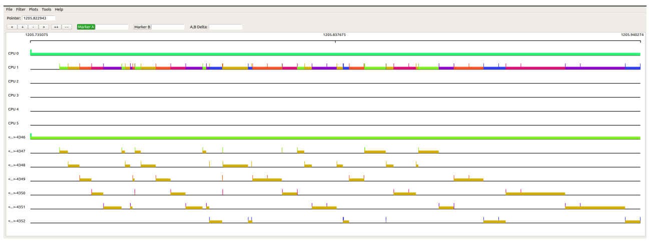


Figure 5: OTHER scheduling graph in Kernel Shark (default task set with static priority 0)

are supposed to meet their deadline to miss them in a longer period of time. On the other hand, it is true that RR scheduling allows the scheduler to switch between tasks more frequently, which can improve system responsiveness and fairness.

2.2.5 Discussion on OTHER Due to the fact that only static priority 0 can be used for the OTHER policy, the priority of each task in the default task set is adjusted to 0. Figure 5 is the graph of traces in Kernel Shark. It is observed that the behavior of OTHER is similar to RR, but different at the same time. It is observed that a kind of "time quantum" exists in OTHER as well, the scheduler would select another available task to preempt the currently running one after a certain amount of time. Unlike RR which follows a fixed priority, OTHER dynamic changes the "priority" of tasks, this can be found at the very first period in figure 5. However, though all tasks have a static priority of 0 assigned at the very beginning, meaning all tasks are equal. It is found that tasks with the earliest deadline seem to always have higher priorities. It is concerned that while the scheduler dynamically adjusts the priorities of tasks, it takes into account the deadline. Moreover, the average response time is even larger than RR. It confirms the assumption made in the observation of RR. The greater response time in OTHER leads to more deadline misses on average.

2.2.6 Discussion on Comparison In this section, a comparison of three scheduling policies on response time and deadline misses is delivered. Figure 6 displays the graph. It is shown clearly in the graph that the response time increases as the scheduling policy changes from FIFO, RR to OTHER. Meanwhile, the number of deadline misses increases as well. The differences among the scheduling graphs and the reasons behind them are discussed in the subsections above, they are not repeated again. The result shown in figure 6 confirms the assumption made during the observation. The larger the response time is, the higher the risk is for more tasks to miss their deadline.

2.3 Question 3

Question: Assume that our goal is to build periodic tasks. In that case, compare these three approaches by building a task set (and scheduling it with rate-monotonic priorities using FIFO). On that task set, choose a task and then measure the difference between the expected arrival times of the task and the actual release times. Show that, unlike the first two methods, the last one results in periodic activations. Justify your answer by adding the traces of KernelShark and visualizing the arrival times of the task.

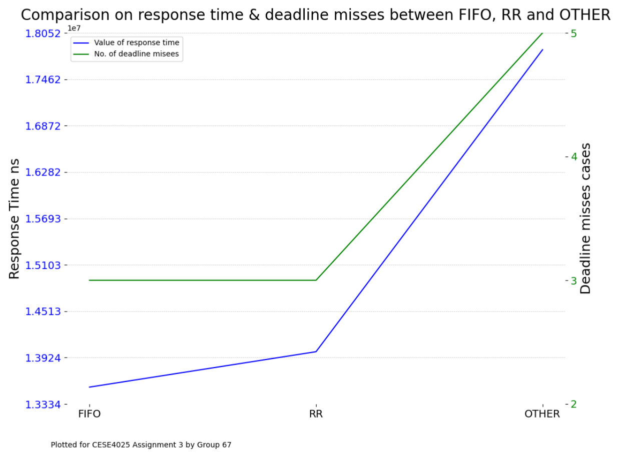


Figure 6: Comparison among policies on Response Time & Deadline Misses

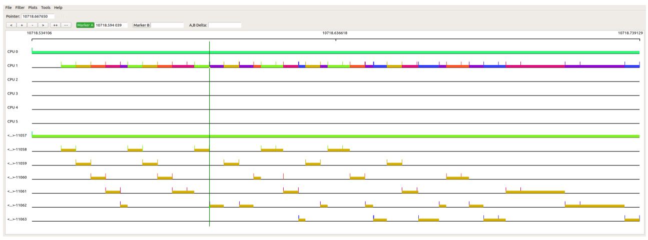
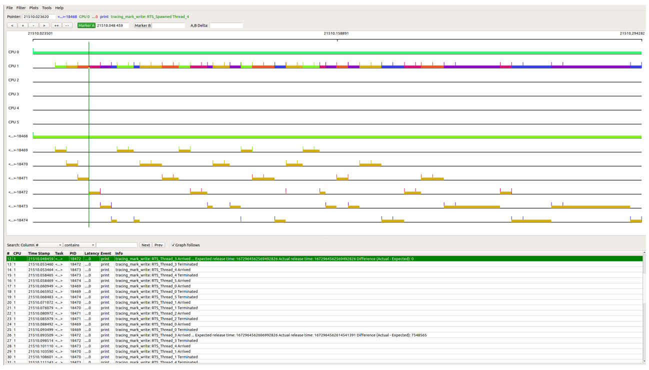


Figure 7: FIFO scheduling graph in Kernel Shark (default task set with RM priorities and correct implementation)

Answer: The answer to this question consists of three parts. The first part is a very concise introduction to what method that has been used in the section 2.2. It is because the "correct implementation" was already used to create periodically released threads. The other two parts are the result achieved using the other two alternatives. Moreover, only the priorities of the designed task set are changed, the rest parameters remain unchanged. Not to mention that FIFO policy is the only scheduling policy used.

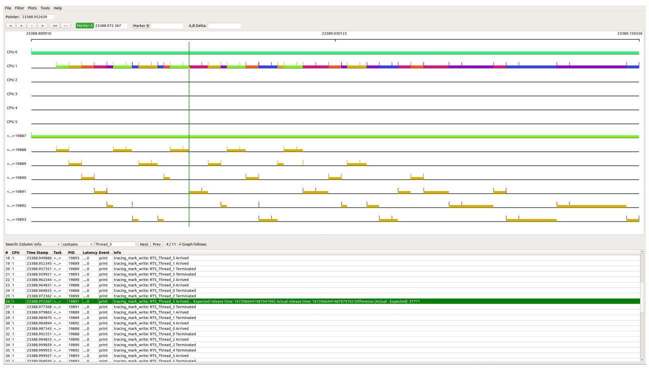
2.3.1 Correct Implementation The correct implementation is introduced in detail in subsection 2.2.2. Function *clock_nanosleep()* together with CLOCK_REALTIME and the absolute flag ensures the functionality of suspending threads. The correct implementation functions work well due to the fact that it is related to a clock which always increments despite which thread is currently executing. Hence, with an absolute time given, *clock_nanosleep()* ensures that the thread is not released until the specific absolute time has passed. Note that a thread being released does not necessarily indicates that it is being executed. The start of the execution time depends on various factors. Figure 7 shows the scheduling graph.

2.3.2 Alternative 1 Alternative 1 requires the program to suspend each thread for a certain amount of time before execution using the function *nanosleep()*.



#	CPU	Time Stamp	Task	PID	Latency	Event	Info
12	1	21510.048459	<...>	18472	0	print	tracing_mark_write: RTS_Thread_3 Arrived ... Expected release time: 1672964562569492826 Actual release time: 1672964562569492826 Difference (Actual - Expected): 0
13	1	21510.053460	<...>	18472	0	print	tracing_mark_write: RTS_Thread_3 Terminated
14	1	21510.053464	<...>	18473	0	print	tracing_mark_write: RTS_Thread_4 Arrived
15	1	21510.058465	<...>	18473	0	print	tracing_mark_write: RTS_Thread_4 Terminated
16	1	21510.058469	<...>	18474	0	print	tracing_mark_write: RTS_Thread_5 Arrived
17	1	21510.060949	<...>	18469	0	print	tracing_mark_write: RTS_Thread_0 Arrived
18	1	21510.065952	<...>	18469	0	print	tracing_mark_write: RTS_Thread_0 Terminated
19	1	21510.068483	<...>	18474	0	print	tracing_mark_write: RTS_Thread_5 Terminated
20	1	21510.071072	<...>	18470	0	print	tracing_mark_write: RTS_Thread_1 Arrived
21	1	21510.076079	<...>	18470	0	print	tracing_mark_write: RTS_Thread_1 Terminated
22	1	21510.080972	<...>	18471	0	print	tracing_mark_write: RTS_Thread_2 Arrived
23	1	21510.085979	<...>	18471	0	print	tracing_mark_write: RTS_Thread_2 Terminated
24	1	21510.088492	<...>	18469	0	print	tracing_mark_write: RTS_Thread_0 Arrived
25	1	21510.093499	<...>	18469	0	print	tracing_mark_write: RTS_Thread_0 Terminated
26	1	21510.093509	<...>	18472	0	print	tracing_mark_write: RTS_Thread_3 Arrived ... Expected release time: 1672964562606992826 Actual release time: 1672964562614541391 Difference (Actual - Expected): 7548565
27	1	21510.098514	<...>	18472	0	print	tracing_mark_write: RTS_Thread_3 Terminated
28	1	21510.101110	<...>	18473	0	print	tracing_mark_write: RTS_Thread_4 Arrived
29	1	21510.103590	<...>	18470	0	print	tracing_mark_write: RTS_Thread_1 Arrived
30	1	21510.108601	<...>	18470	0	print	tracing_mark_write: RTS_Thread_1 Terminated
31	1	21510.111143	<...>	18473	0	print	tracing_mark_write: RTS_Thread_4 Terminated

Figure 8: FIFO scheduling graph in Kernel Shark (default task set with RM priorities and alternative 1)



#	CPU	Time Stamp	Task	PID	Latency	Event	Info
18	1	23388.949886	<...>	19893	0	print	tracing_mark_write: RTS_Thread_5 Arrived
19	1	23388.952343	<...>	19889	0	print	tracing_mark_write: RTS_Thread_1 Arrived
20	1	23388.957351	<...>	19889	0	print	tracing_mark_write: RTS_Thread_1 Terminated
21	1	23388.959921	<...>	19893	0	print	tracing_mark_write: RTS_Thread_5 Terminated
22	1	23388.962344	<...>	19890	0	print	tracing_mark_write: RTS_Thread_2 Arrived
23	1	23388.964831	<...>	19888	0	print	tracing_mark_write: RTS_Thread_0 Arrived
24	1	23388.969835	<...>	19888	0	print	tracing_mark_write: RTS_Thread_0 Terminated
25	1	23388.972362	<...>	19890	0	print	tracing_mark_write: RTS_Thread_2 Terminated
26	1	23388.972367	<...>	19891	0	print	tracing_mark_write: RTS_Thread_3 Arrived ... Expected release time: 1672966441487041992 Actual release time: 1672966441487079763 Difference (Actual - Expected): 37771
27	1	23388.977368	<...>	19891	0	print	tracing_mark_write: RTS_Thread_3 Terminated
28	1	23388.979863	<...>	19889	0	print	tracing_mark_write: RTS_Thread_1 Arrived
29	1	23388.984870	<...>	19889	0	print	tracing_mark_write: RTS_Thread_1 Terminated
30	1	23388.984894	<...>	19892	0	print	tracing_mark_write: RTS_Thread_4 Arrived
31	1	23388.987343	<...>	19888	0	print	tracing_mark_write: RTS_Thread_0 Arrived
32	1	23388.992351	<...>	19888	0	print	tracing_mark_write: RTS_Thread_0 Terminated
33	1	23388.994853	<...>	19890	0	print	tracing_mark_write: RTS_Thread_2 Arrived
34	1	23388.999859	<...>	19890	0	print	tracing_mark_write: RTS_Thread_2 Terminated
35	1	23388.999923	<...>	19892	0	print	tracing_mark_write: RTS_Thread_4 Terminated
36	1	23388.999927	<...>	19893	0	print	tracing_mark_write: RTS_Thread_5 Arrived
37	1	23389.004930	<...>	19893	0	print	tracing_mark_write: RTS_Thread_5 Terminated

Figure 9: FIFO scheduling graph in Kernel Shark (default task set with RM priorities and alternative 2)

In this case, the sleep time used is set to the period of each case. Figure 8 shows the scheduling graph in Kernel Shark with added traces. It makes sense to suspend a thread for the time which equals to its corresponding period until its next release. But when a low-priority thread is preempted by higher ones and resumes afterward, the difference between the current time and the expected next release time might be much shorter than a period or even equal to zero. In this case, if the low-priority thread suspends itself for the time of a period, the deadline might be missed. Hence, the suspension of different threads needs to be flexible and can change based on the current time after the resumption from preempting or execution. With this premise, the second alternative is applied.

2.3.3 Alternative 2 Similar to Alternative 1, *nanosleep()* function is used. The difference from the last technique is that instead of suspending each thread for a fixed amount of time, it calculates the remaining time after execution. Figure 9 shows the scheduling graph in Kernel Shark with added traces. The flexible suspension is more reasonable than Alternative 1, but it is also followed by a potential problem. Threads with low priority can still be preempted at an uncertain time. If the preemption happens during the computation of the amount of time for suspension, the interval is not properly calculated.

2.3.4 Comparison Although Alternative 2 computes the time for suspension more flexibly than Alternative 1, it is still using the relative time period and can not get rid of the influence from being preempted. That is why the usage of *clock_nanosleep()* realizes the real periodic activation. The absolute time points replace the relative time periods which means "sleep until that time" rather than "sleep for a certain time". Even if a thread has been preempted for a long time, as long as it finds that the absolute sleep time point has been exceeded after the resumption, it will give up suspending.

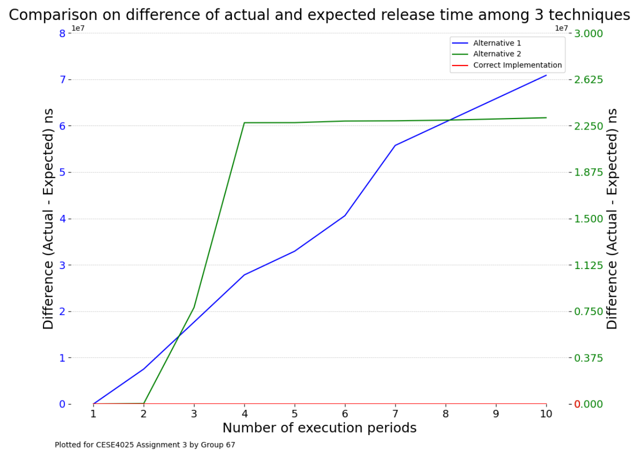


Figure 10: Comparison on difference of actual and expected release time (default task set with RM priorities, each task released 10 times)

In order to compare the difference among these three techniques, a plot of the average difference between the expected arrival time and the actual release time of task 3 is delivered. Note that instead of executing each task 5 times, they are executed 10 times respectively. Figure 10 shows the comparison of the difference between actual and expected release time among the three techniques mentioned in [2]. It is shown that the difference decreases significantly as the technology changes from alternative 1 to alternative 2 and the correct implementation.

3 Discussion & Conclusion

Due of the limited amount of time and the error caused by the inaccurate timing of function *workload()*, the answer towards a lot of questions arose due to curiosity is not yet answered. These questions remain unanswered in this report, however, the effort will be spent on them after this assignment. Answers to all questions of assignment 3 have been delivered in detail.

References

[1] "sched(7) — linux manual page," jambit GmbH. [Online]. Available: https://man7.org/linux/man-pages/man7/sched.7.html

[2] "Programming rt systems with pthreads," Giuseppe Lipari, Scuola Superiore Sant'Anna – Pisa. [Online]. Available: http://retis.sssup.it/~lipari/courses/str09/10.rtprogramming-handout.pdf