Real Time Embedded Systems – ECEN 5623

Summer 2019

Professor Sam Siewert

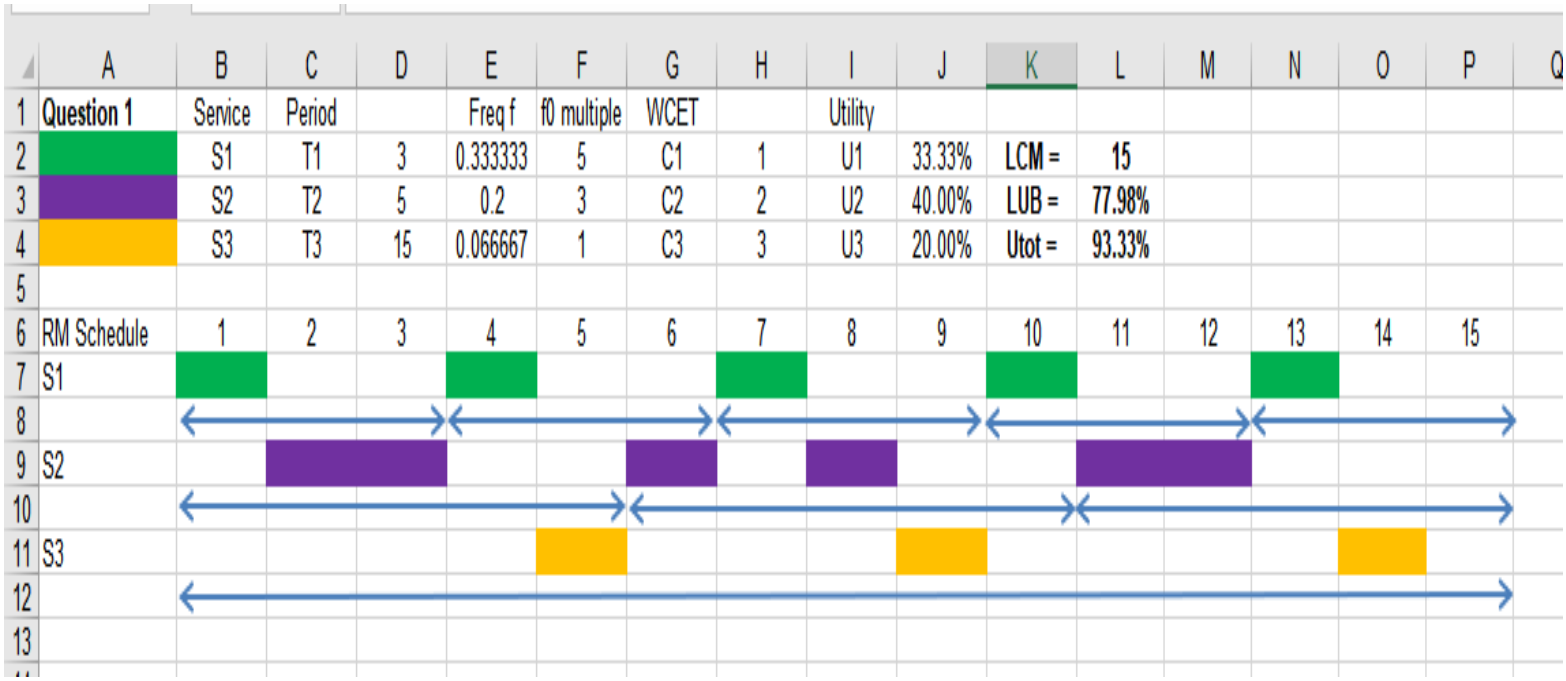
Report By: Poorn Mehta

Homework 1 (git: https://github.com/Poorn-Mehta/RTES/tree/master/HW1)

Q1: The Rate Monotonic Policy states that services which share a CPU core should multiplex it (with context switches that preempt and dispatch tasks) based on priority, where highest priority is assigned to the most frequently requested service and lowest priority is assigned to the least frequently requested AND total shared CPU core utilization must preserve some margin (not be fully utilized or overloaded). Draw a timing diagram for three services S1, S2, and S3 with T1=3, C1=1, T2=5, C2=2, T3=15, C3=3 where all times are in milliseconds. [Note that you can find examples of timing diagrams in Lecture and here – note that we have not yet covered dynamic priorities, just RM fixed policy described here, so ignore EDF and LLF for now]. Label your diagram carefully and describe whether you think the schedule is feasible (mathematically repeatable as an invariant indefinitely) and safe (unlikely to ever miss a deadline). What is the total CPU utilization by the three services?

Solution:

1. **Timing Diagram**



2. **Feasibility & Safety**
    - From the timing diagram, it is evident that the **RM schedule is feasible for the given set of services**. The diagram is drawn for LCM of all 3 services' request periods, and therefore – mathematically it can be repeated over and over again, without having any issues.
    - From timing diagram, it is clear **that the given services and schedule would be safe if capacity C is in terms of WCET** – that is, no task is exceeding their C under any kind of condition. If this condition is met – then it is highly unlikely that this schedule will ever miss a deadline, for any of the given services. Moreover, the CPU utilization is not 100% - and therefore can tolerate some minor faults over time.
    - The context switching latency – should be taken into account as well when designing a system based on this scheduler. If WCET already contains it, then there is no need for this step, however – if all CPU computation times are reflecting processing time only, then many other factors – including context switch latency, should be taken in account.
    - **Total CPU utilization is 93.33% and Least Upper Bound is 77.98%**
3. **Utility and Method**
    - The Rate Monotonic Policy is **extremely useful**, when it comes to **Hard Deadline Real Time Embedded Systems**
    - Even in cases where the tasks are of mixed nature – as in some are Hard Deadline, while other are Soft Deadline or Best Effort ones, the Rate Monotonic Policy can be easily used, along with the partial implementation of some other policies.
    - Despite being old, it is still widely used, owing to its **simplicity**, and the number of **existing tools** to make its implementation even easier, faster, and more efficient.
    - The basic method and flow go like this**: the service with shortest deadline (which is directly proportional to request frequency of that particular task), is given highest priority.**
    - This results in cascading **priorities**, which are all **fixed**.
    - **Preemption is supported**, and therefore, a task can take over the CPU – if it is having higher priority than the one currently using it.
    - The C – or capacities, are to be taken into account in such a way, that not only it covers normal execution time, but also includes variable sources of delays/jitters, and latency related to some operations. This is known as **Worst Case Execution Time**, and if it is calculated correctly – in a deterministic manner, then RM policy would never miss a single deadline (given that it's feasible over the LCM)
    - For this question, I followed the same steps as above – giving highest priority to S1, Second highest to S2, and relatively least to S3.
    - This results in peculiarities such as – S3 not being served till 5^{th} time slot, S2 being interrupt on 7^{th} slot, etc.
    - Overall, RM policy is very useful, and relatively – quite easy to implement.

Q2: Read through the Apollo 11 Lunar lander computer overload story as reported in RTECS Notes, based on this NASA account, and the descriptions of the 1201/1202 events described by chief software engineer Margaret Hamilton as recounted by Dylan Matthews. Summarize the story. What was the root cause of the overload and why did it violate Rate Monotonic policy? Now, read Liu and Layland's paper which describes Rate Monotonic policy and the Least Upper Bound – they derive an equation which advises margin of approximately 30% of the total CPU as the number of services sharing a single CPU core increase. Plot this Least Upper bound as a function of number of services and describe 3 key assumptions they make and document 3 or more aspects of their fixed priority LUB derivation that you don't understand. Would RM analysis have prevented the Apollo 11 1201/1202 errors and potential mission abort? Why or why not?

Solution:

1. **Summary:** Apollo 11 Lunar lander computer faced a severe issue while landing on the Mars. This problem seemed critical enough at a point – to consider the cancellation of the mission itself. However, it was avoided through good restart routine of the processor, which used dynamic priority assignment – and countered the cause of error. An electrical design flaw led to the mistimed startup of rendezvous radar, which filled up all the available RAM and threatening the mission.

2. **Root Cause:** In my opinion, there were **two portions of the root cause** – (1) The hardware design flaw, which included **incompatible power supplies for the rendezvous radar and the computer-aided guidance system** [1] , and (2) The software design flaw (RM policy), gave **highest priority (by nature) to the glitchy service** [2] . The first part, paired with misconfiguration of radar switches, lead to mistimed startup of the rendezvous radar – which in turn filled up all the available memory in the system, exploiting the design flaw in second part. This led to generation of alarm 1202 (all 7 core sets consisting 12 memory locations each, has been filled). Later while landing on Moon, alarm 1201 (all 5 vector accumulator areas consisting 44 memory locations each, has been filled) was also triggered since the scheduling request that overflowed the memory locations had requested a VAC area.

3. **RM Policy Violation:** The software was fortunately designed with anticipation of such problem, and implemented a solid way to break the 'chain' occurring due to the nature of Rate Monotonic Policy. **When the overflow occurred, the system restarted and then – instead of assigning fixed priorities at the startup, based only on the request rate – it actually used dynamic priority assignment. This directly resulted in faulty radar service not being started again.** It did happen a couple of times, but the system restarted almost instantaneously – correcting the original cause of error from the software side.

4. **Least Upper Bound Key Assumptions:**
    1. The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests – This assumption potentially discards all the situations wherein, a task with hard deadline can exist – without being periodic in nature. This will affect the system which can have a critical error responding behavior.
    2. The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks – This assumption prevents the use of this measure in systems wherein, multiple periodic tasks with hard deadlines, could actually be dependent on some other task.
    3. Run-time for each task is constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption – This assumption could be difficult to achieve in some systems, given that many processes have multiple paths to take while running, each path having significantly different length than the other. Therefore, ensuring that task execution time is always constant – is not easy.

5. **LUB Derivation:**

I didn't understand many points in the derivation of LUB, which are listed below.

1. Theorem 4 states a condition '… and the restriction that the ratio between any two request periods is less than two …' – in this, 'any' is the keyword – which implies that I cannot have request periods such as 1,2,3,4 – since pairs 1 & 3 as well as 1 & 4 violates the given restriction. Now, I do not understand that how this condition – transformed into showing that $C_{m-1} = T_m - T_{m-1}$.
2. How is $C_{m-1} = T_m - T_{m-1}$ translated into $C_m = T_m - 2*(C_1 + C_2 + \ldots + C_{m-1})$
3. How step 2 is derived from step 1 in following screen capture

$$= \sum_{i=1}^{m-1} [g_i - g_{i+1}(g_i + 1)/(g_{i+1} + 1)] + 1 - 2[g_1/(g_1 + 1)] \quad \ldots 1$$

$$= 1 + g_1[(g_1 - 1)/(g_1 + 1)] + \sum_{i=2}^{m-1} g_i[(g_i - g_{i-1})/(g_i + 1)]. \quad \ldots 2$$

4. The derivation of step 2 in above picture
5. How that derivation results in $U = m*(2^{1/m} - 1)$

6. This entire portion

Let

$$\begin{aligned} C_1' &= T_2 - T_1 \\ C_2' &= C_2 + \Delta \\ C_3' &= C_3 \\ &\vdots \\ C_{m-1}' &= C_{m-1} \\ C_m' &= C_m . \end{aligned}$$

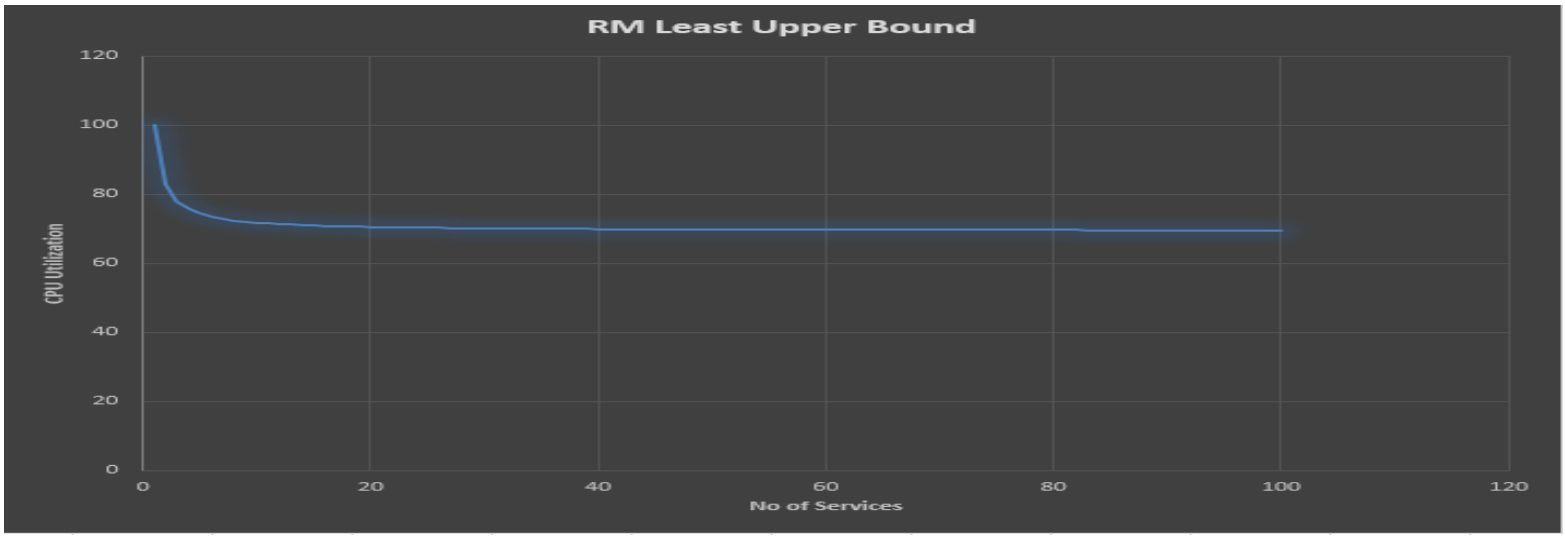
Clearly, $C_1', C_2', \cdots, C_{m-1}', C_m'$ also fully utilize the processor. Let U' denote the corresponding utilization factor. We have

$$U - U' = (\Delta/T_1) - (\Delta/T_2) > 0.$$

6. RM Policy Analysis and Effect on the Error Condition:

I do not think that RM Policy Analysis would have prevented the error 1201/1202 generation for reasons stated below:

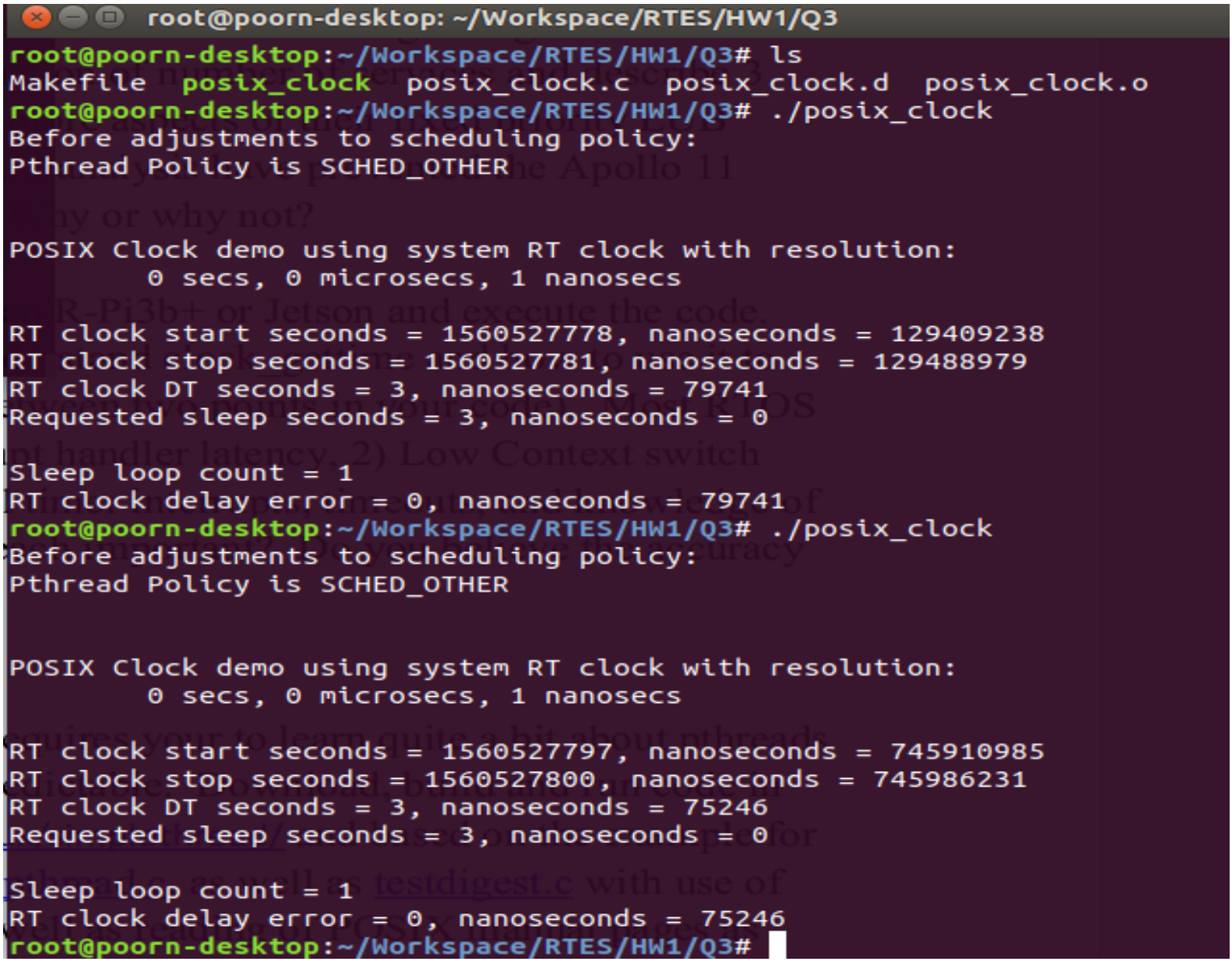
1. **The fixed priorities** – solely based on the rate of requests, was being used at the run time, which is what proposed by RM Policy, also which – in turn led to the error 1201/1202. Again, this happened because RM Policy assumes that all requests being put forth by – are genuine and requires that in order for the system to behave correctly. Since this certainly wasn't the case, RM Policy was/would be causing damage.
2. **The error condition was also somewhat disconnected from the software** – in the sense that the root cause of failure was a faulty hardware, and not software bug/glitch itself. And therefore, in a way, RM Policy implementation/analysis wouldn't have a significant effect.
3. **The margin aspect of the RM Policy – couldn't have prevented overflow** since the faulty service was consuming endless memory. Limiting resources to 70% - wouldn't really make a difference.



Q3: Download RT-Clock and build it on an R-Pi3b+ or Jetson and execute the code. Describe what it's doing and make sure you understand clock_gettime and how to use it to time code execution (print or log timestamps between two points in your code). Most RTOS vendors brag about three things: 1) Low Interrupt handler latency, 2) Low Context switch time and 3) Stable timer services where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift. Why are each important? Do you believe the accuracy provided by the example RT-Clock code?

Solution:

1. **Code ran as-is**



2. Code description

- `struct timespec` – used to store time stamps in form of two different long variables, one containing second and another containing nano-second
- `void print_scheduler (void)` – this function uses API called `sched_getscheduler` to check the currently selected scheduler by the Linux for the given process
- `int delta_t (struct timespec *stop, struct timespec *start, struct timespec *delta_t)` – this function is basically responsible for generating difference between two given timespec structures, and store into target timespec structure. It carefully considers all cases where the second has changed, but 1 full second hasn't elapsed yet.
- `void *delay_test (void *threadID)` – as name suggests, this function tests the system's delay functionality with regarding to the real-time clock provided. It starts by getting the resolution of the real-time clock by using API `clock_getres`. After this, a while loop is created, which takes into account the interruptions in `nanosleep` function call. Then, it calculates the difference between start and stop time – which is then compared to requested delay value.
- `void end_delay_test (void)` – this function simply displays the result of the delay_test and calculated error.
- `int main (void)` – this is the main function of the program. In the present configuration, it only prints the scheduling policy, and performs the delay test.

3. RTOS Bragging Points

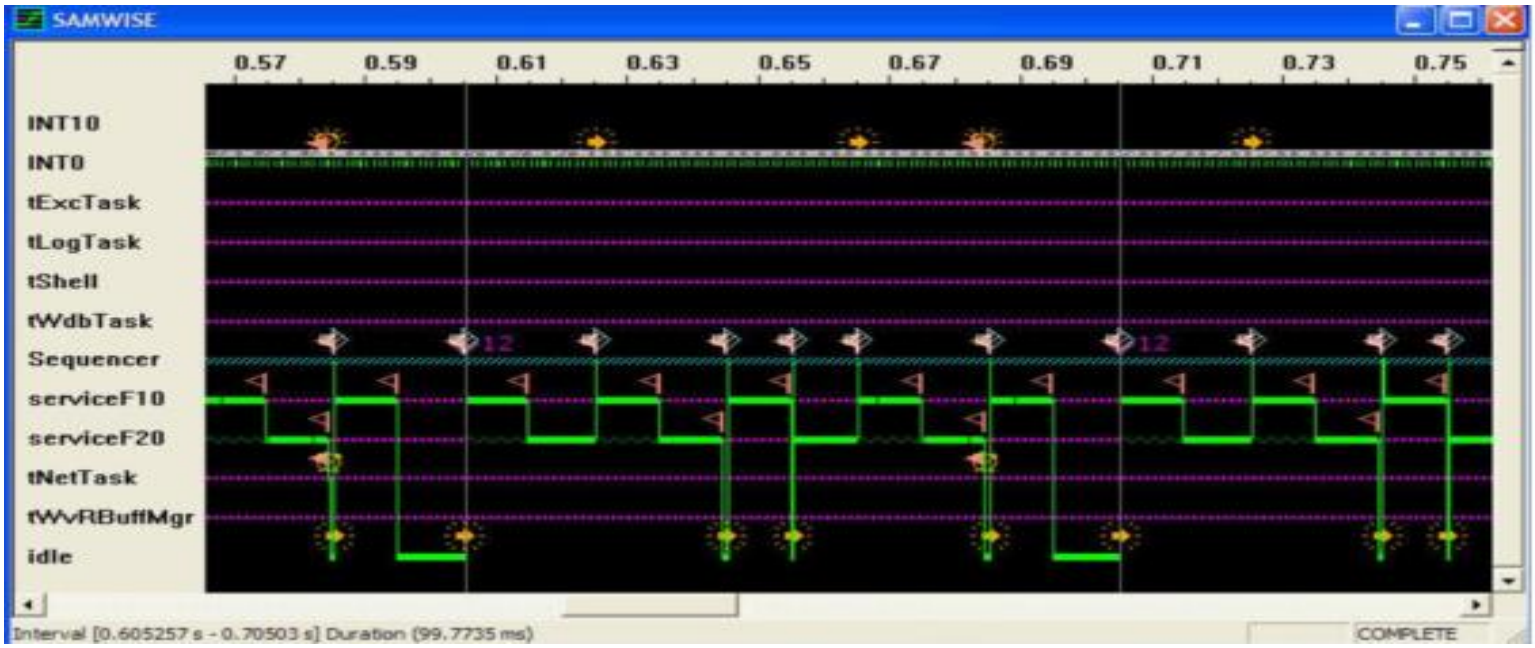
- Low Interrupt Handler Latency – This is important because **if interrupt handler latency is high, then** it is possible that **some very critical events could be delayed** (even result in priority inversion for some time), **and this may prove fatal to the system functionality**. Besides threatening system's stability, it can also lead to undefined and dangerous behavior, which should be avoided by all means. This can happen is the higher latency results in chain-reaction kind of effect by delaying other ongoing services, which can result in these services potentially missing out on their hard-deadlines.
- Low Context Switch Time – Context switch is very crucial since almost all real time systems use multiple tasks on a single processor core. Dependent on the configuration of the system, **there could be thousands of context switches occurring every second**. Now it can be easily visualized that **even a fraction of added latency in context switch time – could accumulate quickly, creating a massive backlog and throwing tasks out of sync.**
- Stable timer services where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift – This is one of the factors which is often overlooked/neglected, and for good reasons probably. Taking this into calculation can result in significant delays in the development of the product, which is not always the best option. However, in many real-time systems, determinism and reliability of the system is of the highest importance. In such cases, there is a high reliance on timer services, which can prove to be harmful if factors such as timer

behavior on internal timer interrupt generation, timeout, jitter, drift, etc. is not deterministic. In many systems, if a timer is running and any kind of interrupt occurs, the timer is stopped immediately. This can cause a lot of instability, if the code is not well written. A real-time focused timer however, can nullify this by implementing various techniques. Similarly, if the relative time measurement is having a high jitter and/or drift, the system progresses towards instability, as the time from system startup moves forward. This could lead to some very dangerous situations, if the system is critical part of a larger system, and/or is not designed to fail safely. Since the initial testing wouldn't reveal about this bug, it certainly is a great thing to have vendors focus on this while developing the RTOS.

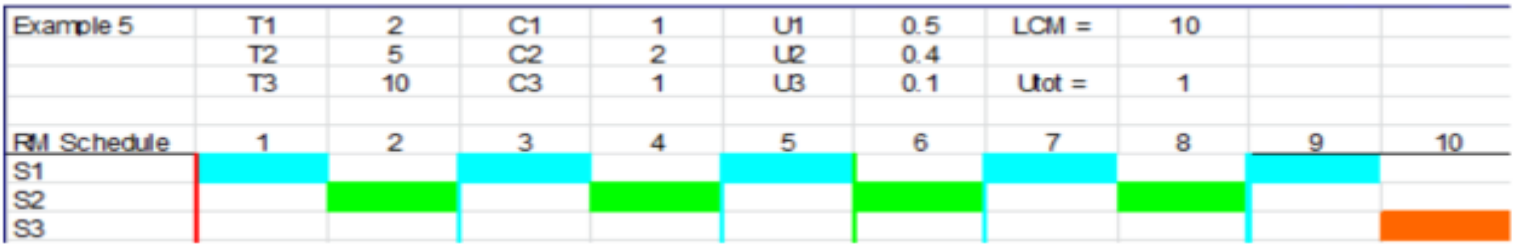
4. Accuracy of RT clock

- After executing the given code for multiple times, it was observed that the **real-time clock of the system is somewhat deterministic and accurate, but definitely not down to the scale of a nano-second**.
- Every run of the program retained variable error. The absolute error was in a few tens of microseconds, and the variation between the subsequent error readings was a few hundred nano-second.
- I would think that the real-time clock is accurate inherently, but various **factors** could be **contributing to the perceived inaccuracy**. These include but are not limited to – jitter due to **varying temperature of the timer oscillator, variable CPU speed** – and resultant firing rate of system's internal interrupts, which can potentially throw off the ongoing timer.
- Regardless of these issues however, it must be noted that the real-time clock is **fairly accurate** – especially since the system tested on is most likely to be Linux – which is designed as a best effort system using completely fair scheduler.

Q4: This is a challenging problem that requires you to learn quite a bit about pthreads in Linux and to implement a schedule that is predictable. Download, build and run code in http://ecee.colorado.edu/~ecen5623/ecen/ex/Linux/simplethread/ and based on the example for creation of 2 threads provided by incdecthread/pthread.c, as well as testdigest.c with use of SCHED_FIFO and sem_post and sem_wait as well as reading of POSIX manual pages as needed - describe how you would attempt to implement Linux code to replicate the LCM invariant schedule implemented in the VxWorks RTOS which produces the schedule measured using event analysis shown below:



The observed timing above fits our theory for RM policy on a priority preemptive scheduling system as shown by the timing diagram below:

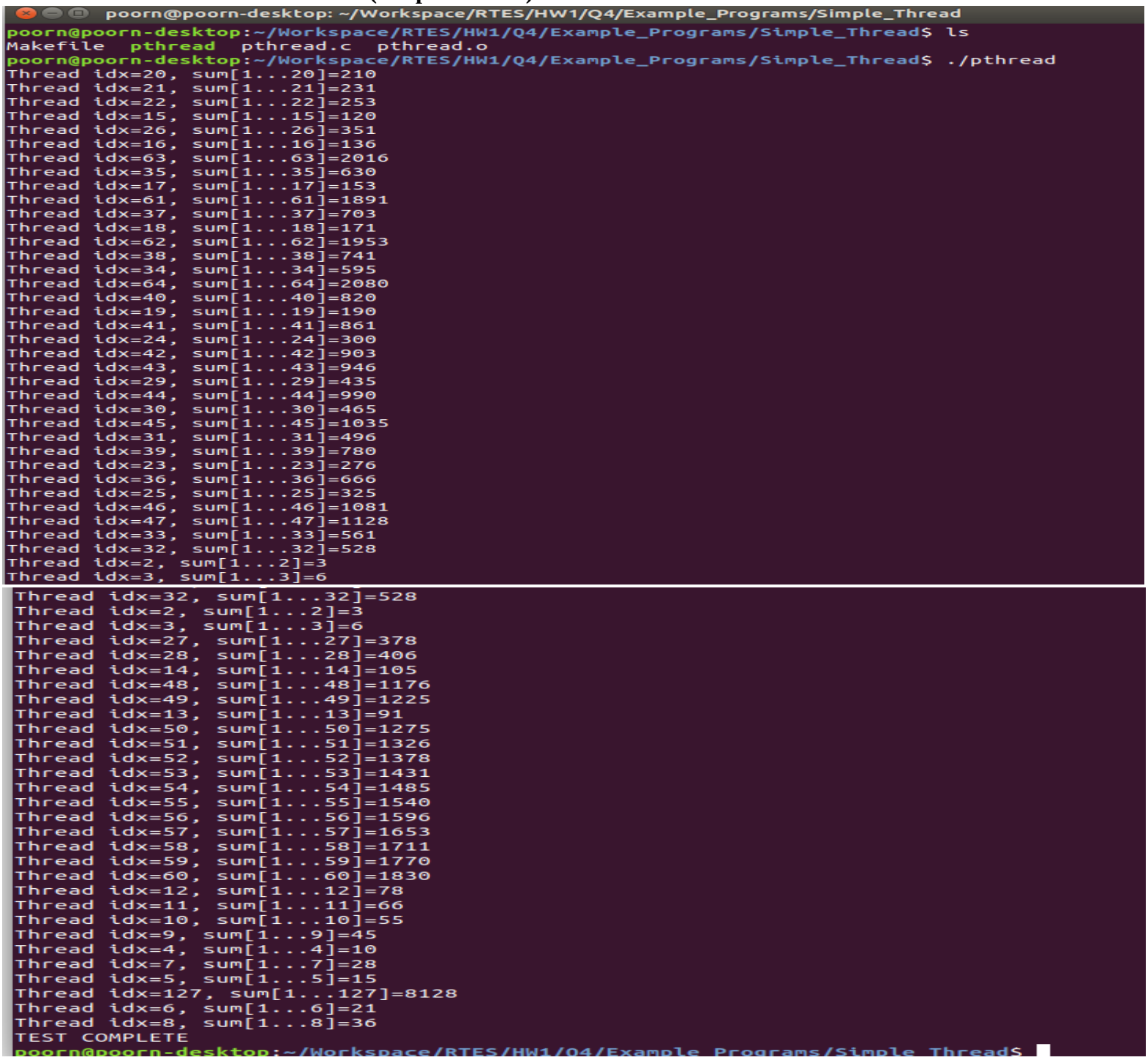


Your description should outline how you would implement code equivalent to the VxWorks synthetic load generation and schedule emulator. Code the Fib10 and Fib20 synthetic load generation and work to adjust iterations to see if you can at least produce a reliable 10 millisecond and 20 millisecond load on ECES Linux or a Jetson system (Jetson is preferred and should result in more reliable results). Describe whether your able to achieve predictable reliable results in terms of the C (CPU time) values alone and how you would sequence execution. Hints – You will find the LLNL (Lawrence Livermore National Labs) pages on pthreads to be quite helpful.

If you really get stuck, a detailed solution and analysis can be found here, but if you use, be sure to cite and make sure you understand it and can describe well. If you use this resource, not how similar or dissimilar it is to the original VxWorks code and how predictable it is by comparison.

Solution:

1. **Code ran as-is (Simple Thread)**



2. **Code ran as-is (Sync Examples)**

```
root@poorn-desktop:~/Workspace/RTES/HW1/Q4/Example_Programs/Sync_Example# ./pthread3
Usage: pthread interfere-seconds
root@poorn-desktop:~/Workspace/RTES/HW1/Q4/Example_Programs/Sync_Example# ./pthread3 2
interference time = 2 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 3
Low prio 3 thread spawned at 1560530455 sec, 633664 nsec
Creating thread 2
Middle prio 2 thread spawned at 1560530456 sec, 633821 nsec
Creating thread 1, CScnt=1
Segmentation fault (core dumped)
root@poorn-desktop:~/Workspace/RTES/HW1/Q4/Example_Programs/Sync_Example#
```

```
root@poorn-desktop:~/Workspace/RTES/HW1/Q4/Example_Programs/Sync_Example# ./pthread3ok 3
interference time = 3 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 1
High prio 1 thread spawned at 1560530574 sec, 732306 nsec
Creating thread 2
Middle prio 2 thread spawned at 1560530574 sec, 732367 nsec
Segmentation fault (core dumped)
root@poorn-desktop:~/Workspace/RTES/HW1/Q4/Example_Programs/Sync_Example#
```

```
poorn@poorn-desktop: ~/Workspace/RTES/HW1/Q4/Example_Programs/Sync_Example
poorn@poorn-desktop:~/Workspace/RTES/HW1/Q4/Example_Programs/Sync_Example$ ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
THREAD 2 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
THREAD 1 got B, trying for A
THREAD 1 got A, trying for B
^C
poorn@poorn-desktop:~/Workspace/RTES/HW1/Q4/Example_Programs/Sync_Example$ ./deadlock race
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 1: -1604718096 done
Thread 2: -1613110800 done
All done
poorn@poorn-desktop:~/Workspace/RTES/HW1/Q4/Example_Programs/Sync_Example$ ./deadlock safe
Creating thread 1
Thread 1 spawned
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: -1606811152 done
Creating thread 2
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: -1606811152 done
All done
poorn@poorn-desktop:~/Workspace/RTES/HW1/Q4/Example_Programs/Sync_Example$
```

3. **Key RTOS / Linux OS porting requirements**
    - Threading vs. Tasking
        - **Tasking is much easier** when it boils down to implementation in real time system.
        - This is obvious since – tasking is created with the goal of achieving just that.
        - For example, while using **pthreads in Linux** – to configure it as close to the task, there are **multiple different APIs** which have to be called in **very specific order**, and with **highly constrained parameters** – in order to work.
        - However, in task creation – **all of the most important factors are handled directly in a single API call**.
        - This **increases system efficiency, and reliability** as well by reducing complexity by a great extent. This covers basic parameters such as stack size to be allocated, priority, etc.
        - On the other hand, there are some areas in which threads perform better than a task. For example, a task creation in VxWorks requires many parameters to be passed inside the API call – many of them being arguments which are to be passed to the task. This is highly inefficient since in most of the cases, all of these would have to be manually set to zero. Moreover, it puts a hard limit on number of arguments as well.
        - To be able to port between systems, there has to be a clear understanding of both – with respect to the operating system, and the default scheduler as well. While most of the internal functionalities can be ported without much issues, overall functionality can't be guaranteed to stay the same – unless careful measurements have been taken.
    - Semaphores, Wait and Sync
        - Real-time operating systems such as **VxWorks** – distinctly provides ways to create a specified semaphore. In other words, it **differentiates among counting semaphores, binary semaphores – etc.**
        - This is in **contrast to Linux** – where there **is no provision to specify this to the compiler.**
        - Due to this, if there is a minor flaw in the semaphore handling logic – Linux based system can easily fail, since it does not know about the intention of the semaphore created, and being used.
        - However, a real-time operating system would actively prevent programmer from making such mistakes, since it can be told explicitly the type of semaphore being used.
        - For example, if a semaphore is supposed to be working as a binary lock (similar to mutex), and if programmer by chance misses a probability of sem_post() being called twice in Linux – it could easily create a massive problem within the system. However, VxWorks just wouldn't let this

happen at any point of time – by raising warnings, or either just keeping the created binary semaphore at 1.

- For **waiting system** however**, Linux seems to be having an advantage** over the VxWorks – since the latter uses number of 'ticks' to calculate the delay/sleep time required, which can be much lesser accurate than the **real-time clock** of the Linux – **working on scale of a nano-second**.
- Moreover, in the waiting system – **Linux** provides functionalities to protect against interrupts (nanosleep() with remaining time). This clearly results in **increased flexibility** – which is not being offered by VxWorks.
- For porting, sufficient comments and code explanations should be provided – especially when the original code is written in the Linux. So, if a choice has to be made between counting and binary semaphore while porting this code to the VxWorks – it shouldn't take much time by just taking a look at notes/comments, or the implementation of the semaphore in question.

- Synthetic workload generation
  - As long as my observation goes, there is not much difference between the way synthetic workload is generated in Linux, vs VxWorks.
  - There could be a few subtleties at work though. For example, **system interrupts of highest priority could affect the same load generating function – differently on different operating systems.**
  - One other factor to be noted here is that the **compiler optimization** can affect the load generation to a very significant scale. Same levels of optimization could mean very different things across the various operating systems.
  - The accuracy of the internal clock (if used in synthetic load generation), speed of the processor**, assembly level code of the same high-level function**, are also some factors that can be significant.
  - While porting, the compiler optimizations are one of the most crucial things that have to be taken care of, and paid high attention towards. Optimizations can result in very erratic behavior on a single system even, and moving to whole other platform without considering optimization could easily result in a catastrophic failure.

4. **Description of my approach**
    - I first went through the **VxWorks example and understood** how it was achieving the scheduling based on RM Policy.
    - Afterwards, I **started looking for APIs in Linux to make the pthread in that – equivalent to a VxWorks task**. It was made much easier by the examples provided by professor.
    - The next obvious step was to **understand the APIs in and out, while also trying them out one-by-one** on my development system. There were some issues with this, as described in section 6 of this solution.
    - Once I got these to work – namely: priorities, FIFO scheduler, and CPU core selection (binding all threads to the same CPU core) – I started **working on generating synthetic load**. An optimization flag in the makefile made this task harder at first.
    - There were some other issues with synthetic load generation as well, again – described in detail in section 6. However, after overcoming them, I **began to write my scheduling thread**, along with the **implementation of semaphores**.
    - After a few iterations, I was able to get my program and threads to behave – as expected. I have tried to run the same code many times, and it has **proven to be deterministic**.

5. **Synthetic workload analysis and adjustment on test system**
    - I had a very difficult time getting the workload function to have deterministic output. It was later found out that the reason wasn't my code, but it was the **Linux's power optimization** policy. Regardless, while trying to make it better, I ended up with a good implementation.
    - To have a very accurate load (in terms of time spent by CPU for calculations), I perform 1000 iterations of Fibonacci calculation series – up to first 45 terms, every time the thread (which is going to need this synthetic load) is called.
    - I **chose 45 since it is the highest round figure** – for the number which can be contained within the limit of uint32_t variable.
    - I also measured time right before and after this **1000 iterations** (effectively, 45000 Fibonacci calculations) – and **calculated average time** taken per iteration.
    - Then, I simply divided 9ms and 18ms respectively for 10ms and 20ms of WCET load, with this per iteration time – number.
    - This gives me highly accurate synthetic workload – and therefore execution time of the thread itself.
    - The reason behind the choice of 9ms and 18ms instead of full 10ms and 20ms – is that at times, due to **context switch latency, and some other factors** – this synthetic workload might be interrupted – or be finished after a not so small delay. This was observed directly over many trials.

6. **Challenges faced**
    - **Optimization flag in Makefile** was making it impossible to create a synthetic load in milliseconds
    - **Operation Not Permitted for sched_setscheduler()**
        - This problem took me a couple of hours to fix. What finally worked is posted here: https://devtalk.nvidia.com/default/topic/1047007/jetson-tx2/operation-not-permitted-by-using-sched_setscheduler-/
    - **Dynamic Scaling Frequency by Linux**
        - This was one of the most daunting problems I've encountered while working on this problem.
        - Regardless of how accurate I try to be with my synthetic load generator, it would always keep on changing drastically, on every single program run.
        - It wasn't solved until Steve Rizor figured it out – he told me to check the scaling frequency of the CPU core I was using – multiple times. And since it was varying, he showed me how to fix that.
        - Commands to check:

          cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq

          cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_cur_freq

        - Commands to fix:

          echo "userspace" > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor

          cat /sys/devices/system/cpu/cpu3/cpufreq/scaling_max_freq > /sys/devices/system/cpu/cpu3/cpufreq/scaling_min_freq

          cat /sys/devices/system/cpu/cpu3/cpufreq/scaling_max_freq > /sys/devices/system/cpu/cpu3/cpufreq/scaling_setspeed