University of Colorado Boulder

EXERCISE #1 - INVARIANT LCM SCHEDULES

REAL TIME EMBEDDED SYSTEMS (ECEN 5623)

SUMMER 2019

REPORT BY:

OM RAHEJA

PROFESSOR:

DR. SAM SIEWERT

1. 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?

Ans: <u>Rate Monotonic Policy:</u>

Rate Monotonic Algorithm is a static priority scheduling algorithm. The algorithm states that "Priority of each task is assigned according to its period. Shorter the period or higher the frequency, higher the priority of that task."

There are three tasks: S1, S2 and S3.

Time period(T) and run-time(C) of each task/service:

S1:

T1=3, C1=1

S2:

T2=5, C2=2

S3:

T3=15, C3=3

Therefore, by observing the time period of the given services and by the Rate Monotonic Policy, we can conclude that the order of priority will be, **S1 > S2 > S3** as S1 has the least time period and the highest frequency of occurrence, while S3 has the maximum time period and the least frequency of occurrence.

<u>Timing Diagram:</u>

As the Least common multiple of T1, T2 and T3 is **15**, we will see the feasibility of the services being scheduled over the course of 15 milliseconds. We see that the time period of service S1 is the least and hence gets the highest priority. Similarly, service S3 has the highest time period, making it the least priority service. Refer to Figure 1 for the timing diagram of the three services.

<u>Utility and Method Description:</u>

- Determine the priority of the services based on the given time period. Service with least time period gets the highest priority and the service with the maximum time period gets the least priority. By keeping this in mind, we determine that the order of priority for our exercise problem would be **S1 > S2 > S3.**

- After the priorities have been decided for all the services, we calculate the LCM of the time periods of all the services. Based on the LCM, we find whether the services are invariant in that time period. In our example problem, T1 = 3, T2= 5, T3=15. Thus, the LCM of these time periods is **15.**
- Now, we plot the timing diagram for all the services along with their assigned priorities. Each service has to complete their run-times (Ci) before their time period in order to be successfully schedulable.
- In our exercise problem, all the services obey the Rate monotonic policy. Thus, we can conclude that the services are feasible as they don't miss their deadlines in the time period being examined.

<u>Feasibility:</u>

From figure 1, it can be seen that the services are mathematically repeatable as an invariant indefinitely. Thus, the services are FEASIBLE.

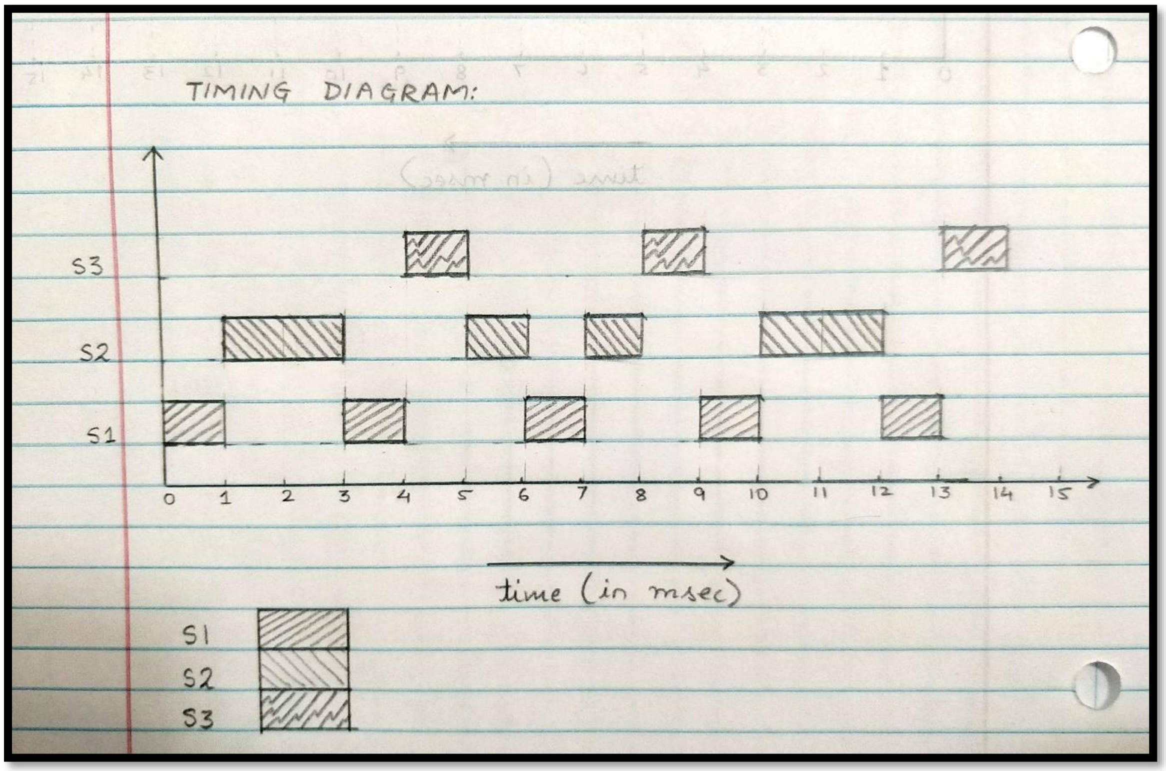


Figure 1. Rate monotonic Timing diagram for Services S1, S2 and S3

<u>Calculations of Processor Utilization:</u>

Processor Utilization (Ui) = ∑ (Ci/Ti), where " i " ranges from 1 to m.

Ui = CPU Utilization of the ith service.

Ci = Execution time or Run time of ith service.

Ti = Time period of the ith service.

M = number of tasks/services.

Using the above equation, we can calculate the CPU utilization of each task and hence the total CPU utilization.

<u>For Service S1:</u>

Time period (T1) = 3

Run time (C1) = 1

CPU Utilization for Service (S1) = C1/T1

= 1/3

= **0.33333333**

Percentage Utilization = **33.33 %** ……..(1)

<u>For Service S2:</u>

Time period (T2) = 5

Run time (C2) = 2

CPU Utilization for Service (S2) = C2/T2

= 2/5

= **0.4**

Percentage Utilization = **40 %** ……..(2)

<u>For Service S3:</u>

Time period (T3) = 15

Run time (C3) = 3

CPU Utilization for Service (S3) = C3/T3

= 3/15

= **0.2**

Percentage Utilization = **20 %** ……..(3)

From (1), (2) and (3),

Total CPU Utilization = **0.93333333333**

Total CPU Utilization (%) = **93.333333333** % ……..(4)

According to Liu Layland's paper, Processor Utilization factor is defined as the fraction of the processor time spent in the execution of the task set. Based on the previous statement, we can observe that 14 out of 15 time slots are being utilized. Meaning, if we evaluate the CPU utilization, we get that the CPU is executing the task sets every 14 out of the 15 time slots, which comes down to 93.333333333 % of CPU utilization. This way, the CPU is idle for every 1/15th of the time slots, that is 6.6666667 % of the total CPU utilization. This can be clearly seen in figure 1 that the CPU is idle for the last millisecond (14th to 15th millisecond).

According to Liu Layland's paper, Services are considered to be safe if the CPU utilization is less than or equal to the Least Upper Bound CPU utilization.

Mathematically,

U = ∑ (Ci/Ti) , i ranging from 1 to m should be less than or equal to **U= m((2^(1/m) - 1)**, that is the least upper bound CPU utilization.

M = Number of tasks

Thus, to evaluate if the schedule is safe (unlikely to ever miss a deadline), we need to calculate the Least Upper Bound CPU Utilization.

LUB CPU Utilization = m (2^(1/m) - 1)

As M = 3 in our case, we substitute the value of m to calculate the least upper bound CPU utilization.

= 3 (2^(1/3) – 1)

= 0.7797

LUB CPU Utilization (%) = 77.97% …….(5)

From (4) and (5) , we can conclude that the CPU Utilization is 93.33333%. This is greater than the least upper bound CPU utilization. Therefore, the schedule is Un-safe for the present scenario. **However, even if the schedule is un-safe, it is still Feasible.**

2. 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 increases. 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?

Ans: <u>Summary of the Apollo 11 Lunar Lander Computer Overload Story:</u>

The author, Peter Adler gives a brief description about their work at the Labs. He them talks about the constraints they had to deal with while they were programming the LGC. He mentions that the system they used had 36,864 15-bit words. These were referred as "Fixed" memory (now called as ROM). Apart from the fixed memory, the system had 2048 words of "Erasable" memory or RAM. As they had a limited amount of memory available, they were forced to use the same memory address for different purposes at different times. This was a problem because a variable stored at time 't1' might be replaced of overwritten at time 't2' (assuming t2 > t1). Moreover, some of the memory locations were such that they were being shared between 7 processes. The team had to do an extensive testing to ensure that no same memory location was being used by different programs at any point in time.

The next paragraph mainly focuses on the memory locations allocated to each job. Each job was allocated a "core set" of 12 erasable memory locations. Mostly, these locations were allocated to store temporary data. If at all the jobs required more temporary storage, the scheduling request asked for a VAC (vector accumulator). The VAC had 44 erasable words. There were 7 core sets and 5 VAC areas.

If the job to be scheduled required a VAC area, the OS scanned the entire VAC area and allocated the one which was available. Once, the VAC area had been reserved, the core sets were scanned to find one which was available. If the scheduling request cam with a "NOVAC", then scanning of the VAC area was skipped by the OS.

If NO VAC AREAs were available, the code was programmed such that it would raise a **1201 Alarm.** Similarly, if no core sets were available, the program would raise a **1202 Alarm.**

<u>REASONS FOR 1201/1202 ALARM:</u>

The reason for the 1202 alarm was that the CPU was processing radar data which was getting scheduled because of a radar switch misconfiguration. Due to this, the core sets got filled up eventually leading to 1202 alarm. The scheduling request that caused the actual overflow of the VAC area was the reason for the 12021 alarm.

Fortunately, this condition was taken care by the software team. They had programmed the software to be such that the computer could recognize that secondary data to which more priority was been given has to be ignored. Priority had to be given to the task which was of primary importance. The handled this by rebooting and reinitializing the system and restart selected programs at a point in their execution.

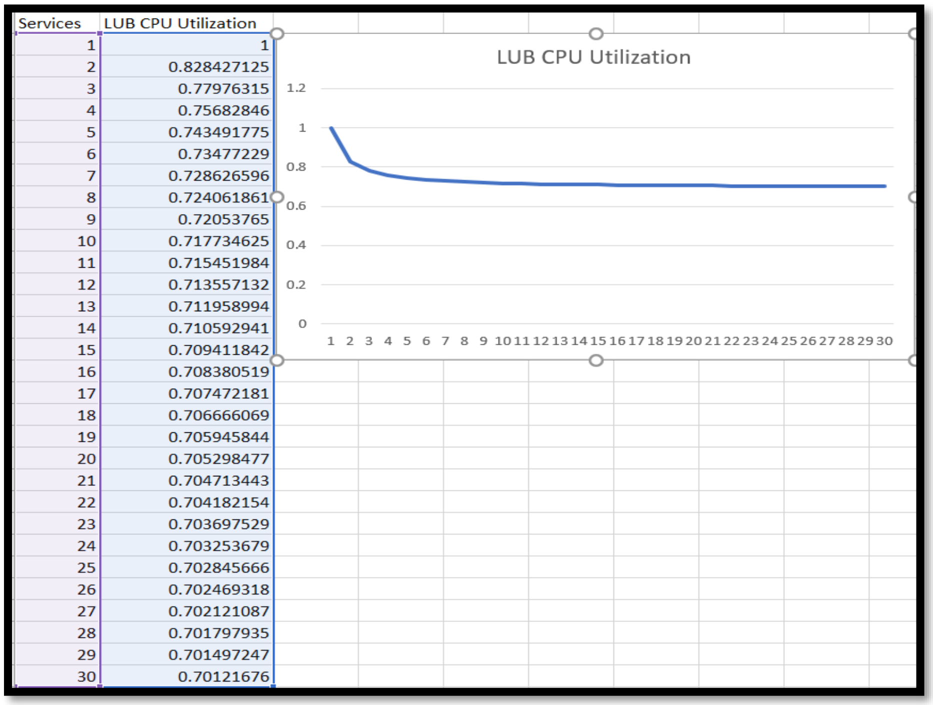
Thus, each time a 1201 or a 1202 alarm appeared, the system rebooted, restarted only the important tasks and did not start the erroneously scheduled radar jobs. The guys at NASA had extensively tested the software and it is because of this reason that the mission could go forward. If the computer would not have been able to recognize the problem, then the Apollo 11 mission wouldn't have been successful.

As recounted by Dylan Matthews, the rendezvous radar and the computer-aided guidance system used incompatible power supplies. The radar didn't have much purpose during landing; still, it continued to send lots of data to the computer based on random electrical noise. This overloaded the CPU and left no space for tasks that actually needed to be run at that point of time. Thanks to the well-designed system in which the computer could recognize the inappropriate behavior and reboot, restarting only the important and needed tasks and ignoring the rest.

The rate monotonic policy says that the highest priority is assigned to the task which occurs most frequently. Since in the Apollo 11 mission, the frequent tasks were given a lower priority, it violated the Rate monotonic policy.

Would RM analysis have prevented the Apollo 11 1201/1202 errors and potential mission abort? Why or why not?

Ans: I believe that the Rate Monotonic analysis would not have prevented the Apollo 11 mission abort. This is mainly due to the fact that if rate monotonic policy would have been adapted, then the erroneous tasks like the random radar data which was occurring at a very high frequency would have been given a higher preference. If that would have been the case, then the VAC areas along with the core sets would have got filled up and caused the system to crash. But, instead of using the RM policy, they had used the dynamic priority scheduling because of which only the most important tasks were processed whereas tasks which were of no importance yet higher frequency were ignored.



Services	LUB CPU Utilization
1	1
2	0.828427125
3	0.77976315
4	0.75682846
5	0.743491775
6	0.73477229
7	0.728626596
8	0.724061861
9	0.72053765
10	0.717734625
11	0.715451984
12	0.713557132
13	0.711958994
14	0.710592941
15	0.709411842
16	0.708380519
17	0.707472181
18	0.706666069
19	0.705945844
20	0.705298477
21	0.704713443
22	0.704182154
23	0.703697529
24	0.703253679
25	0.702845666
26	0.702469318
27	0.702121087
28	0.701797935
29	0.701497247
30	0.70121676

Figure 2. Graphical plot of Number of Services vs LUB CPU Utilization

According to paper by Liu Layland and the graph above, it is observed that as the number of tasks increase, the CPU utilization approximates to 70%, leaving a 30% margin according to the rate monotonic analysis.

<u>Assumptions in the Paper by *Liu and Layland:*</u>

- Any non-periodic tasks in the system are special; they are initialization or failure-recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.
- Deadlines consist of run-ability constraints only- i.e each task must be completed before the next request for it occurs.
- Run-time for each task is constant for that task and does not vary with time.

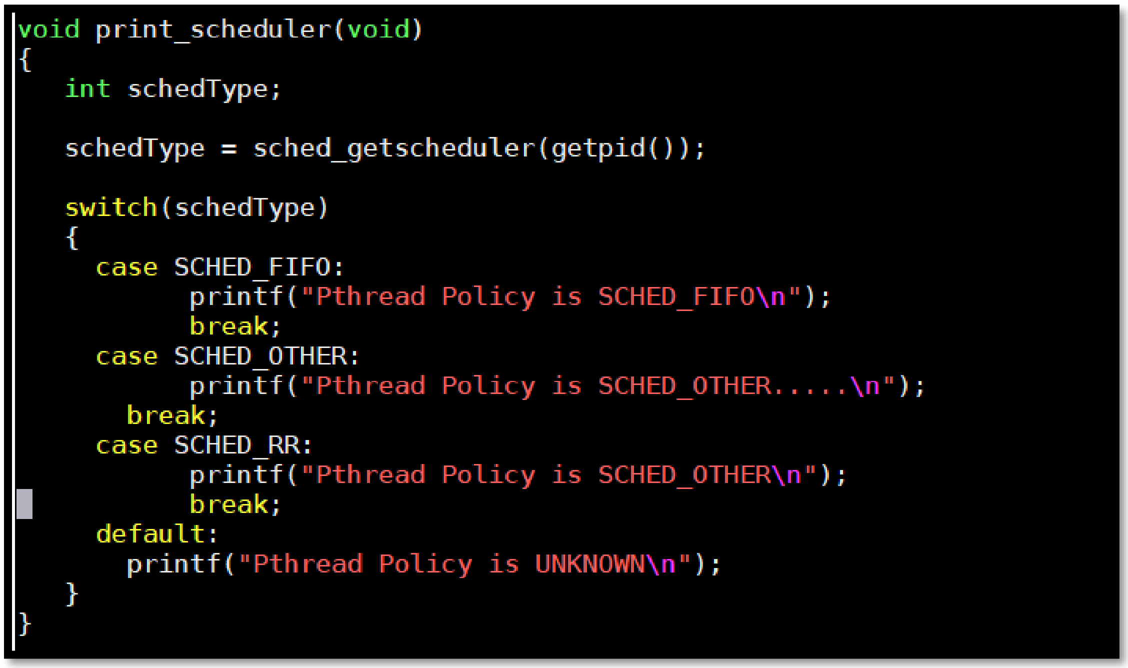
<u>Three aspects of the Fixed Priority LUB derivation, I didn't understand:</u>

- In theorem 4, where they are trying to prove that the least upper bound CPU utilization factor is $U = m((2)^{1/m)} -1)$, they write an equation,
  $C_m = T_m – 2(C_1 + C_2 + ... + C_{m-1})$. I wasn't able to contemplate the use of this equation.
- When the authors are trying to calculate the least upper bound CPU utilization for two tasks, they have considered two cases. The equations of C_1 and C_2 used in both the cases is a bit confusing and obscure. A more detailed explanation as to how and from where those equations arrived would have made it a little easy to understand the derivation.
- Some background on the general mathematical equations that have been used could have helped analyze the theorems more clearly.

3. 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?

Ans: Explanation of the Code:

PRINT_SCHEDULER(void) FUNCTION:



```
void print_scheduler(void)
{
   int schedType;

   schedType = sched_getscheduler(getpid());

   switch(schedType)
   {
     case SCHED_FIFO:
           printf("Pthread Policy is SCHED_FIFO\n");
           break;
     case SCHED_OTHER:
           printf("Pthread Policy is SCHED_OTHER.....\n");
       break;
     case SCHED_RR:
           printf("Pthread Policy is SCHED_OTHER\n");
           break;
     default:
       printf("Pthread Policy is UNKNOWN\n");
   }
}
```

Figure 3. Print_Scheduler function

The "Print_Scheduler" function prints the scheduling policy used by that process. Linux supports a range of scheduling policies. There are two sets of scheduling policies sub divided into a number of other scheduling policies.

SCHEDULING POLICIES	
NORMAL (NON-REAL TIME)	REAL TIME
1. **SCHED_OTHER** This is a standard round-robin time-shring policy.	1. **SCHED_FIFO** This is a FIFO (First-In-First_Out) scheduling policy.
2. **SCHED_BATCH** This scheduling policy is for "batch" style execution of processes.	2. **SCHED_RR** This is a Round Robin scheduling policy.
3. **SCHED_IDLE** For running very low priority background jobs.	

The sched_getscheduler() function queries the scheduling policy applied to the process identified by pid. If the pid equals zero, the policy of the calling process will be retrieved.

In the kernel, there is a scheduler that decides which runnable process will be executed next by the CPU. Each process has a scheduling policy and a static scheduling priority associated with it. These parameters can be modified by the sched_setscheduler() function.

For processes that are scheduled under one of the normal scheduling policies, sched_priority is not used in scheduling decisions, that is, it must be specified as 0. On the other hand, if processes that are scheduled under one of the real-time scheduling policies, sched_priority ranges between 1 to 99 (1 being the least priority and 99 being the highest priority).

In this program, the scheduling policy is set as SCHED_OTHER. This scheduling policy can be used as static priority 0. SCHED_OTHER is a standard Linux time-sharing scheduler that is intended for all processes that do not require the special real-time mechanisms.

DELAY_TEST(void *threadID) FUNCTION:

This function calls the clock_getres() function. This function finds the resolution (precision) of the specified clock clk_id. The resolution of the RT clock is **1 nanosecond**. Once the resolution of the RT clock is found, the duration for which the execution of the calling thread is to be paused is set. The clock_gettime() function is called just before nanosleep() is called and just after the thread wakes up from sleep. This way we get the start time and the stop time of measured by the RT clock. The start time and the stop time measured is then passed to the delta_t() function to calculate the time elapsed by the thread in sleep. Once we get the difference between the start and the stop time, these values are populated in a structure rtclk_dt. In order to calculate the accuracy of the RT clock, we pass the rtclk_dt structure along with the sleep_requested structure to determine the error that occurs in calculating the accuracy of the RT clock. Once all these values are obtained, we display the values making a function call to the end_delay_test() function.

The clock_gettime() function is used to retrieve the time of the specified clock clk_id. The clk_id argument is the identifier of the particular clock on which to act. The clock_gettime function can be used to determine the time taken by any section in the code. General steps that can be followed to obtain the time a specific code section takes are as follows:

- Declare and Define two structures of type timespec. For example rtclk_start_time and rtclk_stop_time. Initialize the members of both the structures to 0.
  Static struct timespec rtclk_start_time = {0,0};
  Static struct timespec rtclk_stop_time = {0,0};
- Call the clock_gettime(CLOCK_REALTIME, &rtclk_start_time) function to store the time at which you start measuring the time. This will store the start time in the members of the structure “rtclock_start_time”.
- Similarly, call the clock_gettime(CLOCK_REALTIME, &rtclk_stop_time) function to store the time at which you stop measuring the time. This will store the stop time in the members of the structure “rtclock_stop_time”.

- Once these values are determined, these structures can be passed to a function to determine the difference between the start and stop time. This will give us the time elapsed in the execution of a specific section of code.

RTOS vendors brag about three things:

1) Low Interrupt handler latency:

- The amount of time that elapses between a device interrupt request and the first instruction of the corresponding ISR is known as Interrupt Latency.
- Not only that, the ISRs should be kept as small and simple as possible.
- A major contributor to increased interrupt latency is the number and length of regions in which the kernel disables interrupts. By disabling interrupts, the kernel may delay the handling of high priority interrupt requests that arrive in those windows in which interrupts are disabled. Most RTOSes employ this simple architecture since it is easy to implement and the commonly used and understood mechanism.
- By disabling the interrupts, the RTOS is in effect sacrificing the latency of the highest priority interrupt to avoid problems caused by handling of lower priority interrupts.
- An RTOS that avoids high latency instructions like an integer divide and string manipulations achieves a better interrupt latency than an RTOS that ignores this restriction.
- All these factors when incorporated in an RTOS lowers the interrupt handler latency when compared to other Operating Systems like Linux. Moreover, this helps reduce the overall system response time, thus making RTOS an important tool to adapt when developers are concerned with a system involving Hard Real Time Deadlines.

2) Low Context Switch Time:

- Context switch occurs when the kernel transfers control of the CPU from an executing process to another that is ready to run. The kernel saves the context of the process.
- When the CPU context switches between multiple processes or tasks, it stores the status of the CPU's register and program counter. A register is a small amount of very fast memory inside a CPU that is used to speed the execution of computer programs by providing quick access to commonly used values.
- The time taken by the CPU to switch between multiple processes is crucial for a real time embedded system. Lower the time to context switch between processes, better is the response.
- As real time embedded systems and real time operating systems demand instant response to any event, the RTOS is designed such that the time taken to switch from one process to another is very less when compared to other Operating systems.
- This is the main reason why RTOS vendors highlight this feature.

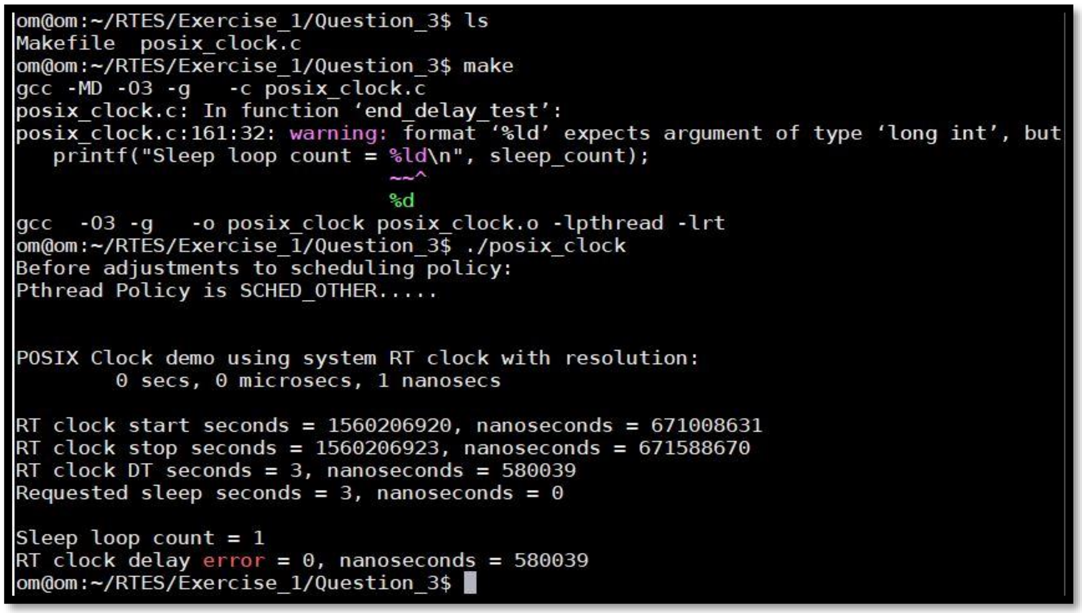
3) **<u>Stable Timer Services where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift:</u>**
   - <u>Stable Timer Services:</u>
     Timers are an important tool when it comes to Real Time Embedded Systems as they require response at precise time instances. So, having a timer which provides an accurate time measurement to meet the necessary deadline is mandatory. Although, achieving a 100% accuracy is not possible, but a timer with least amount of jitter and drift is desirable for real-time applications. Thus, a stable timer service is important such that hard real time deadlines do not get missed because of an inaccurate timer service.
   - RTOS has a very precise timer service with a resolution in the range of nanoseconds. This is much greater than the timer service of embedded linux which has an accuracy in the range of milliseconds.
   - Because of a high precision timer service provided in the RTOS interval timer interrupts, timeouts and relative time has a low jitter and drift from the actual values.

Do you believe the accuracy provided by the example RT-Clock code?

Below is the screenshot of the output generated by the execution of posix_clock.c code. We can see that the RT clock resolution is 1 nanosecond. When difference is taken of the stop and start time, we get the value of 3 seconds (which in our case is the value we are looking for). Apart from this, we also observe that the difference is not exactly 3 seconds, but a value slightly greater than 3 seconds. This additional value was observed to vary each time the code was executed. We see that there is an additional delay of 580039, 568029 nanoseconds and this value is observed to change each time the code is executed.

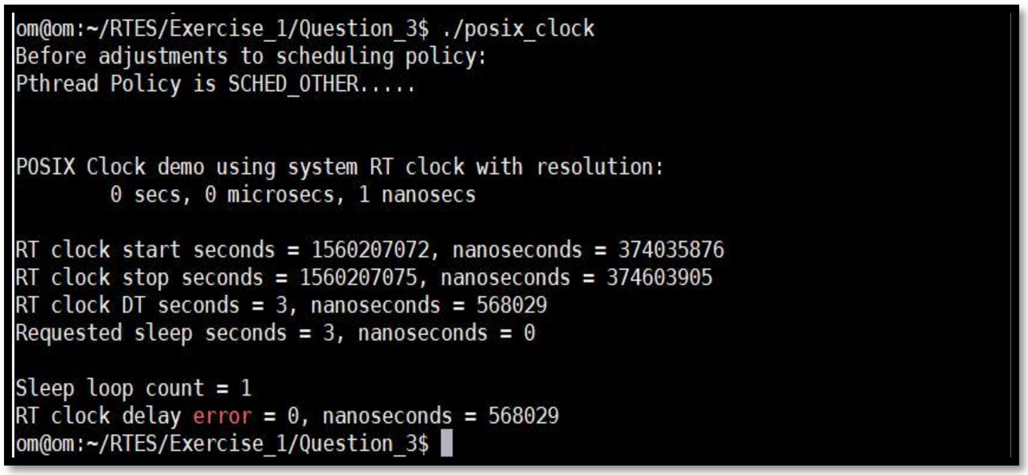
This delay can increase if the time to be measured by the RT clock increases. Hence, it can be unreliable and can be a potential threat if used in a Real time critical application which demands actions to be taken in real time without any delay.

Thus, the RT clock used in Not Completely accurate.



```
om@om:~/RTES/Exercise_1/Question_3$ ls
Makefile  posix_clock.c
om@om:~/RTES/Exercise_1/Question_3$ make
gcc -MD -O3 -g   -c posix_clock.c
posix_clock.c: In function ‘end_delay_test’:
posix_clock.c:161:32: warning: format ‘%ld’ expects argument of type ‘long int’, but
   printf("Sleep loop count = %ld\n", sleep_count);
                              ~~^
                              %d
gcc  -O3 -g   -o posix_clock posix_clock.o -lpthread -lrt
om@om:~/RTES/Exercise_1/Question_3$ ./posix_clock
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER.....


POSIX Clock demo using system RT clock with resolution:
        0 secs, 0 microsecs, 1 nanosecs

RT clock start seconds = 1560206920, nanoseconds = 671008631
RT clock stop seconds = 1560206923, nanoseconds = 671588670
RT clock DT seconds = 3, nanoseconds = 580039
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 580039
om@om:~/RTES/Exercise_1/Question_3$
```

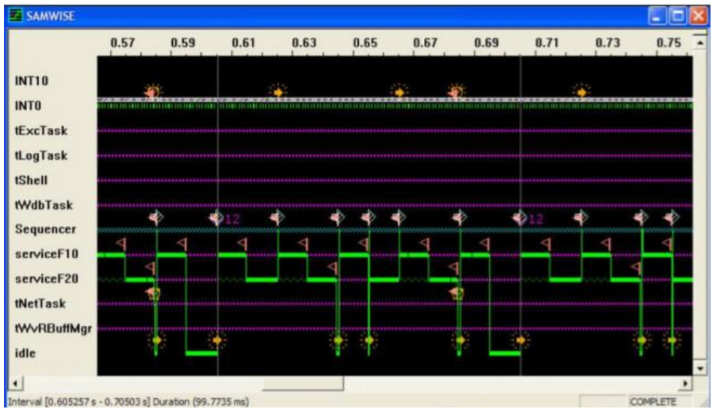
Figure 4. posix_clock.c Output



```
om@om:~/RTES/Exercise_1/Question_3$ ./posix_clock
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER.....


POSIX Clock demo using system RT clock with resolution:
        0 secs, 0 microsecs, 1 nanosecs

RT clock start seconds = 1560207072, nanoseconds = 374035876
RT clock stop seconds = 1560207075, nanoseconds = 374603905
RT clock DT seconds = 3, nanoseconds = 568029
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 568029
om@om:~/RTES/Exercise_1/Question_3$
```

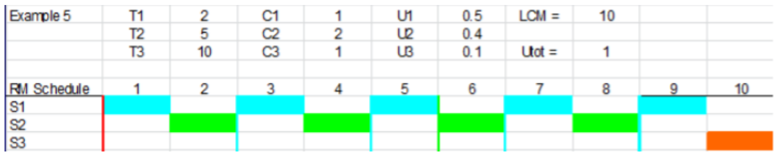
Figure 5. posix_clock.c Output

4. 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:



Interval [0.605257 s - 0.70503 s] Duration (99.7735 ms)

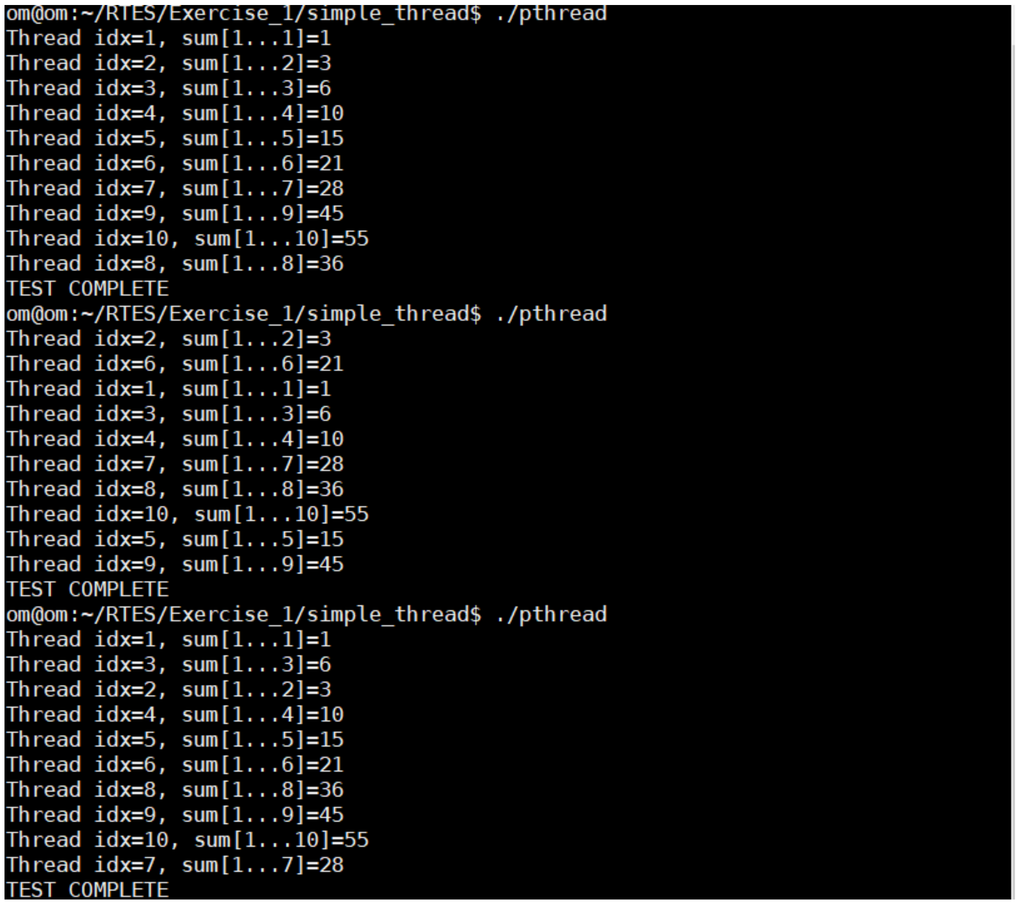
The observed timing above fits our theory for RM policy on a priority preemptive scheduling system as shown by the timing diagram below:



Example 5	T1	2	C1	1	U1	0.5	LCM =	10		
	T2	5	C2	2	U2	0.4				
	T3	10	C3	1	U3	0.1	Utot =	1		
RM Schedule	1	2	3	4	5	6	7	8	9	10
S1	🟦		🟦		🟦		🟦		🟦	
S2		🟩		🟩		🟩		🟩		
S3										🟧

You 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.

Ans: <u>Output and Explanation of pthread.c</u>



```
om@om:~/RTES/Exercise_1/simple_thread$ ./pthread
Thread idx=1, sum[1...1]=1
Thread idx=2, sum[1...2]=3
Thread idx=3, sum[1...3]=6
Thread idx=4, sum[1...4]=10
Thread idx=5, sum[1...5]=15
Thread idx=6, sum[1...6]=21
Thread idx=7, sum[1...7]=28
Thread idx=9, sum[1...9]=45
Thread idx=10, sum[1...10]=55
Thread idx=8, sum[1...8]=36
TEST COMPLETE
om@om:~/RTES/Exercise_1/simple_thread$ ./pthread
Thread idx=2, sum[1...2]=3
Thread idx=6, sum[1...6]=21
Thread idx=1, sum[1...1]=1
Thread idx=3, sum[1...3]=6
Thread idx=4, sum[1...4]=10
Thread idx=7, sum[1...7]=28
Thread idx=8, sum[1...8]=36
Thread idx=10, sum[1...10]=55
Thread idx=5, sum[1...5]=15
Thread idx=9, sum[1...9]=45
TEST COMPLETE
om@om:~/RTES/Exercise_1/simple_thread$ ./pthread
Thread idx=1, sum[1...1]=1
Thread idx=3, sum[1...3]=6
Thread idx=2, sum[1...2]=3
Thread idx=4, sum[1...4]=10
Thread idx=5, sum[1...5]=15
Thread idx=6, sum[1...6]=21
Thread idx=8, sum[1...8]=36
Thread idx=9, sum[1...9]=45
Thread idx=10, sum[1...10]=55
Thread idx=7, sum[1...7]=28
TEST COMPLETE
```

Figure 6. Output of pthread example code

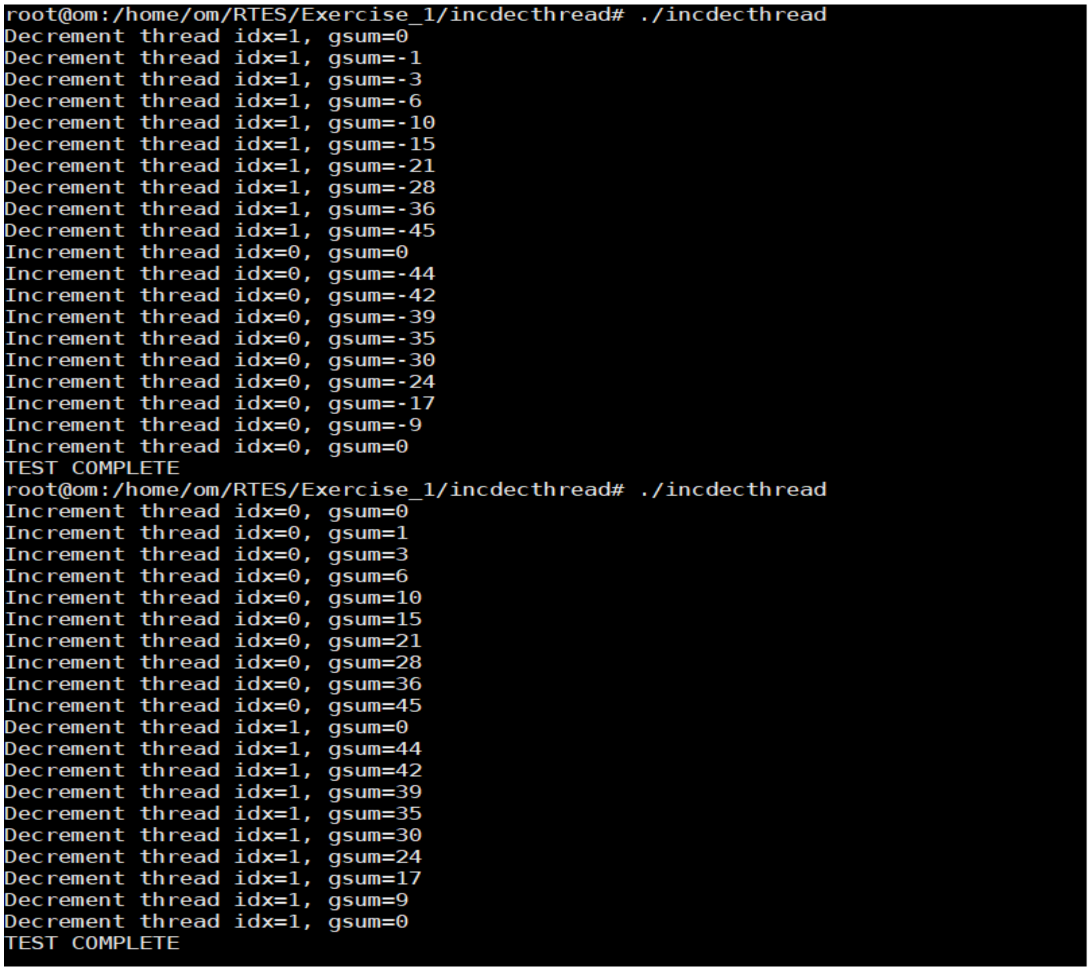
<u>Steps to run the code:</u>

- Board used: NVIDIA's Jetson Nano development kit
- Download the code from the website http://ecee.colorado.edu/~ecen5623/ecen/ex/Linux/simplethread/. Make sure to download the Makefile as well.
- Compile the code and run it.

<u>Explanation:</u>

- In the main function, 'n' number of threads are created, where n is specified by a MACRO "NUM_THREADS".
- All the threads created enter a function *counterThread, which basically calculates the sum from 1 to the 'Thread Index Id' of that particular thread.
- The main thread waits for all the created threads to run to completion and then exits.
- By default, all the threads created are spawned on different cores. In order to restrict it to a single core, we need to bind that thread to one particular CPU core.

Output and Explanation of incdecthread.c



```
root@om:/home/om/RTES/Exercise_1/incdecthread# ./incdecthread
Decrement thread idx=1, gsum=0
Decrement thread idx=1, gsum=-1
Decrement thread idx=1, gsum=-3
Decrement thread idx=1, gsum=-6
Decrement thread idx=1, gsum=-10
Decrement thread idx=1, gsum=-15
Decrement thread idx=1, gsum=-21
Decrement thread idx=1, gsum=-28
Decrement thread idx=1, gsum=-36
Decrement thread idx=1, gsum=-45
Increment thread idx=0, gsum=0
Increment thread idx=0, gsum=-44
Increment thread idx=0, gsum=-42
Increment thread idx=0, gsum=-39
Increment thread idx=0, gsum=-35
Increment thread idx=0, gsum=-30
Increment thread idx=0, gsum=-24
Increment thread idx=0, gsum=-17
Increment thread idx=0, gsum=-9
Increment thread idx=0, gsum=0
TEST COMPLETE
root@om:/home/om/RTES/Exercise_1/incdecthread# ./incdecthread
Increment thread idx=0, gsum=0
Increment thread idx=0, gsum=1
Increment thread idx=0, gsum=3
Increment thread idx=0, gsum=6
Increment thread idx=0, gsum=10
Increment thread idx=0, gsum=15
Increment thread idx=0, gsum=21
Increment thread idx=0, gsum=28
Increment thread idx=0, gsum=36
Increment thread idx=0, gsum=45
Decrement thread idx=1, gsum=0
Decrement thread idx=1, gsum=44
Decrement thread idx=1, gsum=42
Decrement thread idx=1, gsum=39
Decrement thread idx=1, gsum=35
Decrement thread idx=1, gsum=30
Decrement thread idx=1, gsum=24
Decrement thread idx=1, gsum=17
Decrement thread idx=1, gsum=9
Decrement thread idx=1, gsum=0
TEST COMPLETE
```

Figure 7. Output of incdecThread example code

Steps to run the code:

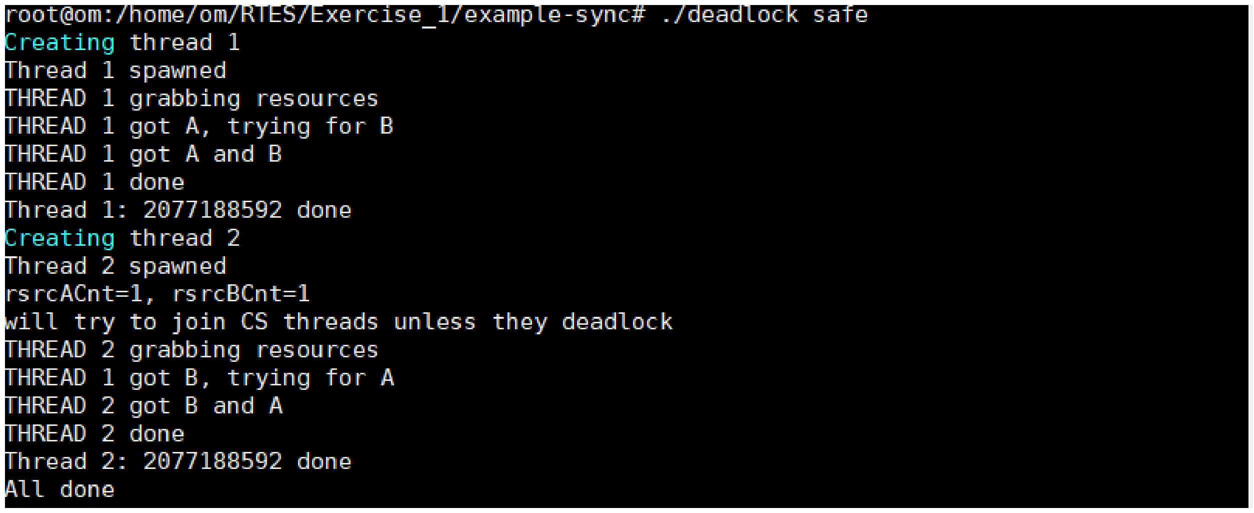
- Board used: NVIDIA's Jetson Nano development kit
- Download the code from the website http://ecee.colorado.edu/~ecen5623/ecen/ex/Linux/incdecthread/pthread.c. Make sure to download the Makefile as well.
- Compile the code and run it.

Explanation:

- The main process creates two threads, one which has an entry point at incThread and the other has an entry point at decThread.
- Both the threads are manipulating the same global variable 'gsum'. The incThread is responsible for incrementing the value of gsum variable by the value of the ith iteration. On the other hand, decThread is responsible for decrementing the value of gsum variable by the value of the ith iteration.

Output and Explanation of example-sync code:

- **DEADLOCK SAFE**

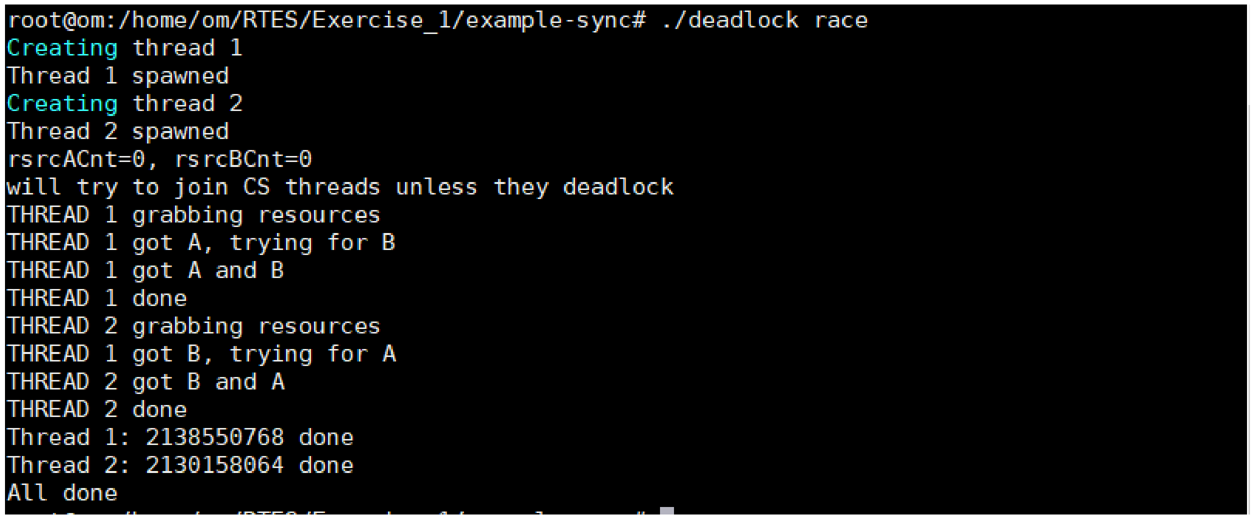


```
root@om:/home/om/RTES/Exercise_1/example-sync# ./deadlock safe
Creating thread 1
Thread 1 spawned
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: 2077188592 done
Creating thread 2
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads 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: 2077188592 done
All done
```

Figure 8. Output of deadlock example code [Safe]

When the deadlock code is executed using ‘Safe’ as the second argument, the code is programmed such that Thread 1 finishes with both the resources. The second thread is spawned only after thread 1 finishes its execution. The main process waits for thread 2 to finish its execution. Once thread 2 finishes its execution, the entire program is terminated.

- **DEADLOCK RACE**

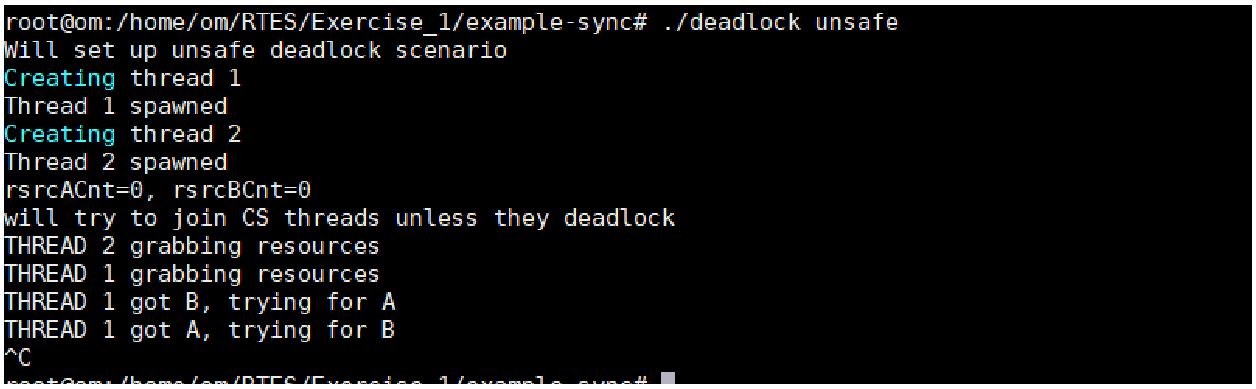


```
root@om:/home/om/RTES/Exercise_1/example-sync# ./deadlock race
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 1: 2138550768 done
Thread 2: 2130158064 done
All done
```

Figure 9. Output of deadlock example code [Race]

When the deadlock code is executed using ‘Race’ as the second argument, the code is programmed such that which ever thread gets to the resource first, will utilize it and release the resource. In this case, thread 1 is spawned first. So, thread 1 takes the resource first, uses it and then releases after which thread 2 grabs and uses the resource.

- **DEADLOCK UNSAFE:**

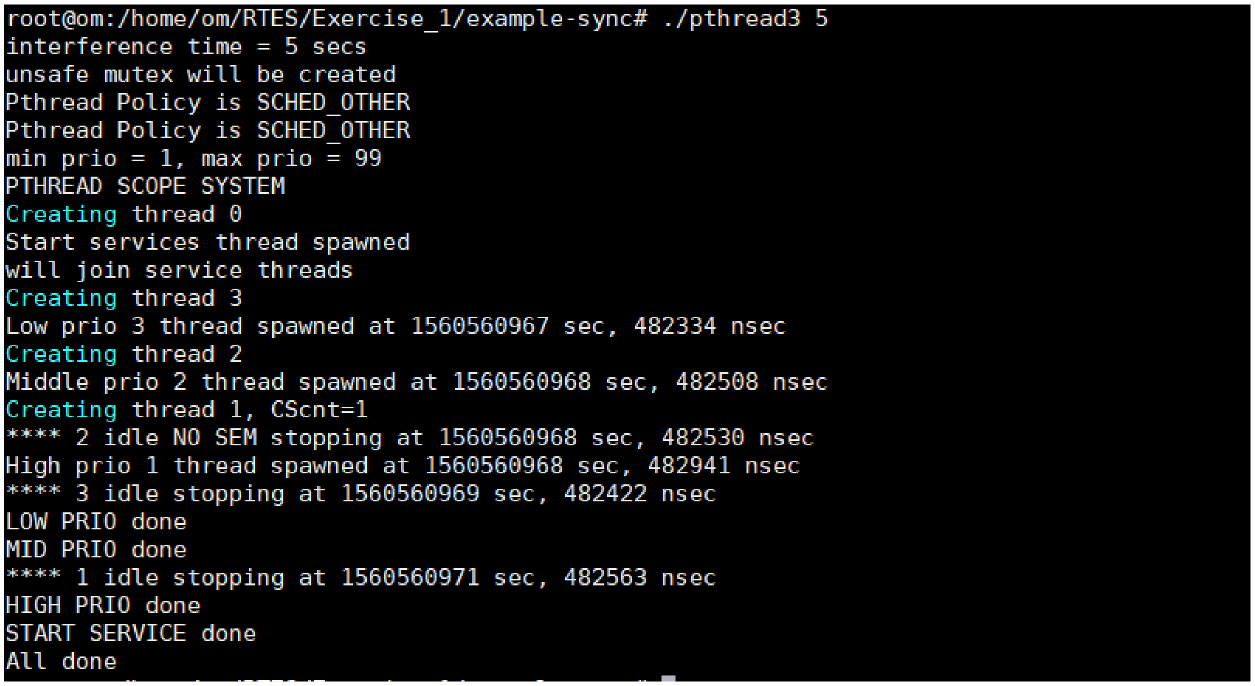


```
root@om:/home/om/RTES/Exercise_1/example-sync# ./deadlock unsafe
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 grabbing resources
THREAD 1 got B, trying for A
THREAD 1 got A, trying for B
^C
```

Figure 10. Output of deadlock example code [UnSafe]

When the deadlock code is executed using ‘Unsafe’ as the second argument, the code is programmed such that a situation similar to deadlock is being emulated. What happens here is that for thread 1 to complete its execution, it needs to make use of both, Resource A and Resource B. Similarly, thread 2 also requires Resource A and Resource B for its execution. In this situation (deadlock), Thread 1 grabs Resource A and waits for Resource B. On the other hand, the thread 2 grabs resource B and waits for resource A. For thread 1 to complete its execution, it needs resource B or else it will never release resource A. Similar is the case with thread 2. This situation leads to a deadlock where each thread waits infinitely for the other thread to release their resource.

- **PTHREAD3 5:**



```
root@om:/home/om/RTES/Exercise_1/example-sync# ./pthread3 5
interference time = 5 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 1560560967 sec, 482334 nsec
Creating thread 2
Middle prio 2 thread spawned at 1560560968 sec, 482508 nsec
Creating thread 1, CScnt=1
**** 2 idle NO SEM stopping at 1560560968 sec, 482530 nsec
High prio 1 thread spawned at 1560560968 sec, 482941 nsec
**** 3 idle stopping at 1560560969 sec, 482422 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1560560971 sec, 482563 nsec
HIGH PRIO done
START SERVICE done
All done
```

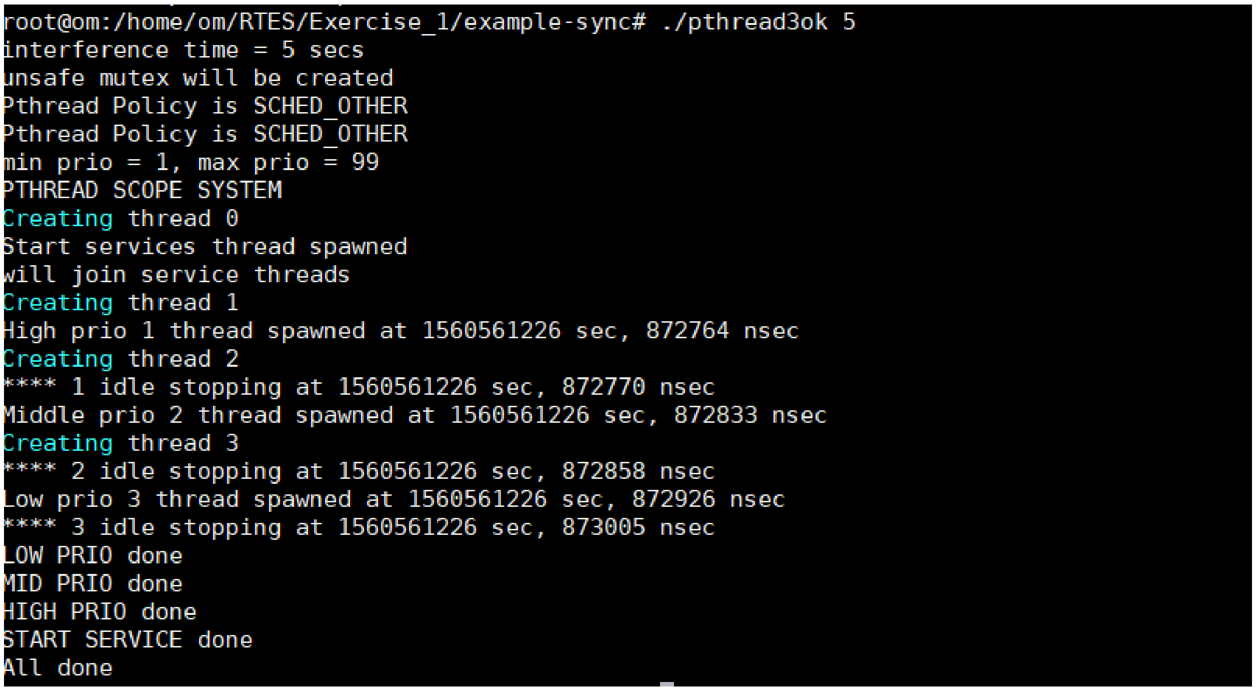
Figure 11. Output of pthread3 example code

Steps to Run Code:

- Compile the code and type the following command to run the program:
  ./pthread3 5

In this code, the lowest priority thread and the highest priority thread have the same point of entry in the thread 'idle'. What happens in this code is that the lowest priority thread enters the idle function, uses mutex to avoid any race conditions with other threads. It sleeps for 2 seconds and then releases the mutex lock. During this time, the highest priority thread is created. As these both have the same entry point, the highest priority thread tends to enter that function but cannot as its been locked by the thread having lowest priority. This leads to an inversion in priority. Such cases should be testes an avoided in a real world application. The highest priority should not be held down against a lower priority task.

- **PTHREAD3OK 5:**



```
root@om:/home/om/RTES/Exercise_1/example-sync# ./pthread3ok 5
interference time = 5 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 1560561226 sec, 872764 nsec
Creating thread 2
**** 1 idle stopping at 1560561226 sec, 872770 nsec
Middle prio 2 thread spawned at 1560561226 sec, 872833 nsec
Creating thread 3
**** 2 idle stopping at 1560561226 sec, 872858 nsec
Low prio 3 thread spawned at 1560561226 sec, 872926 nsec
**** 3 idle stopping at 1560561226 sec, 873005 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
START SERVICE done
All done
```

Figure 12. Output of pthread3ok example code

Steps to Run Code:

- Compile the code and type the following command to run the program:
  ./pthread3ok 5

In this code the above condition has been taken care of. The lowest priority thread gets preempted by the highest priority thread. That way, the higher priority thread would not have to wait for the lower priority thread to complete its execution.

# CODE DESCRIPTION:

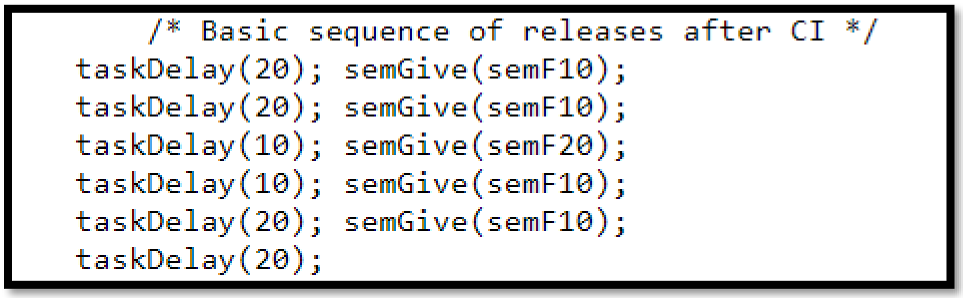
Note : The code has been written by Dr.Sam Siewert. I have used this code (unchanged) as a reference for this assignment. Here is a detailed explanation of the code. All copyrights reserved to the owner of the code.

VxWorks RTOS code Explanation:

This code demonstrates an implementation of a scheduler for two tasks which can be predicted. It replicates the LCM invariant schedule.

The main function, denoted as start() in Vxworks spawns a Sequencer Task. The sequencer task is responsible for replicating the timing diagram by designing a precise and predictable scheduler. Binary semaphores are used to emulate the basic sequence of releases after run-times of each task. The sequencer task spawns two tasks “ServiceFib10” and “ServiceFib20”. After creating the two tasks, a wvEvent API is called in order to log the user event. This is done to log the F10 and F20 events. Once the start of the LCM phasing sequence is logged, a loop for sequencing the LCM phase of Service S1 and Service S2 is run.

As we can see in the below figure, the sequencer first gives the semaphore to F10 and F20 which wait for the semaphore after their task creation. Once both F10 and F20 get the semaphore they are likely to start their execution. As F10 has the highest prioirty, it starts its execution. In the mean time, the sequencer task is asleep for 20ms. Once F10 executes for 10ms, F20 being the next in queue, startes its execution for 10ms. Now, the sequencer wakes up and gives the semaphore to F10. F10 again runs for 10ms. F20 which is yet to run for 10 more milliseconds, runs in this span. Similarly, the entire sequence of release of run times for F10 and F20 are emulated. By using semaphores, only tasks with the semaphore and with the highest priority run first, while the other task waits for their time share. This way even synchronization has been maintained between multiple tasks.



```
        /* Basic sequence of releases after CI */
   taskDelay(20); semGive(semF10);
   taskDelay(20); semGive(semF10);
   taskDelay(10); semGive(semF20);
   taskDelay(10); semGive(semF10);
   taskDelay(20); semGive(semF10);
   taskDelay(20);
```

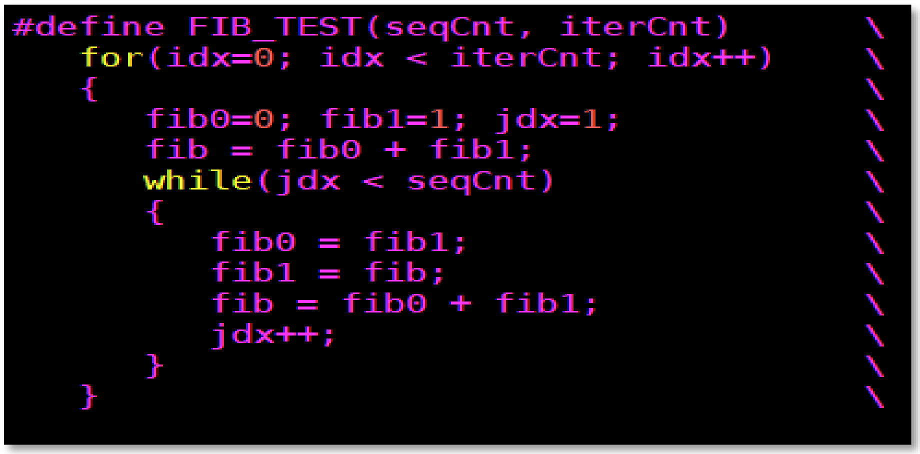
Figure 12. Code snippet from VxWorks example

Linux Code Implementation:

The implementation of this code in Linux is very similar to that in the VxWorks RTOS. The only difference is that the VxWorks makes use of tasks while Linux makes use of threads to achieve the same result.

The main process is responsible for creating three threads, Fib10, Fib20 and the sequencer thread. The sequencer thread has the highest priority and the Fib10 has the next highest priority. The Fib20 thread has the least priority amongst the other threads. The approach remains the same as that in VxWorks. Attributes and priorities are set before creation of each thread. Once the threads are created, the Fib10 and Fib20 threads wait for the sequencer thread to post the semaphore. Once the Fib10 and Fib20 threads get the semaphore, they achieve the workload desired.

Synthetic Workload Analysis:



```
#define FIB_TEST(seqCnt, iterCnt)      \
   for(idx=0; idx < iterCnt; idx++)    \
   {                                   \
      fib0=0; fib1=1; jdx=1;           \
      fib = fib0 + fib1;               \
      while(jdx < seqCnt)              \
      {                                \
         fib0 = fib1;                  \
         fib1 = fib;                   \
         fib = fib0 + fib1;            \
         jdx++;                        \
      }                                \
   }                                   \
```

Figure 12. code snippet of FIB_TEST macro

The FIB_TEST Macro is used as a way to generate load on the CPU core. This macro calculates the fibonacci series based on the sequence count and the iteration counts specified. By adjusting the sequence count and the iteration count, load can be generated on the CPU for a desired amount of time period.

The code used in this assignment (Code referenced from Dr.Sam Siewert), dynamically calculates the time taken by the CPU core being used to calculate a fibonacci series. Based on this measured time, the number of cycles for which the fibonacci series needs to be calculated to achieve a load generation of 10ms for Fib10 and 20ms for Fib20.

The logic for replicating the basic sequence to release the runtimes of threads is same as that implemented in the VxWorks code. So, the timing diagram has been generated a total of 3 times in the time range of 0-100ms, 100-200ms and 200-300ms. Once these threads finish their tasks, they exit, which results in the termination of the program.

REFERENCES:

[1] Liu and Layland paper
http://mercury.pr.erau.edu/~siewerts/cec450/documents/Papers/liu_layland.pdf

[2] Linux Man Pages

[3] Context Switch https://www.sciencedirect.com/topics/engineering/context-switch

[4] Apollo 11 https://www.vox.com/2015/5/30/8689481/margaret-hamilton-apollo-software

[5] Rate Monotonic Policy Concept: https://www.embedded.com/electronics-blogs/beginner-s-corner/4023927/Introduction-to-Rate-Monotonic-Scheduling

[6] Geeksforgeeks for concepts on pthreads, semaphores and mutexes.

[7] Code reference: Dr.Sam Siewert's course webpage
http://ecee.colorado.edu/~ecen5623/index_summer.html

[8] Github Link for my codes: https://github.com/omraheja/Real-Time-Embedded-Systems-ECEN-5623