ECEN5623, Real-Time Embedded Systems

Exercise #1- Invariant LCM Schedules

RTES Exercise 1 `

Question 1 : [20 points] 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 and in Canvas – 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:

What's Rate Monotonic Algorithm?

i) The Rate Monotonic Algorithm is a scheduling algorithm which works on the principle of preemption i.e a higher priority task blocks a lower priority task.
ii) The priorities of the task are decided by the time period required by the task to execute. The task with shortest time period gets the highest priority while the one with longest time period gets the least priority.

In the given question, we have three services to be executed named as S1, S2 and S3

The **Time Periods (T)** of the three tasks are : T1 = 3, T2 = 5 and T3 = 15

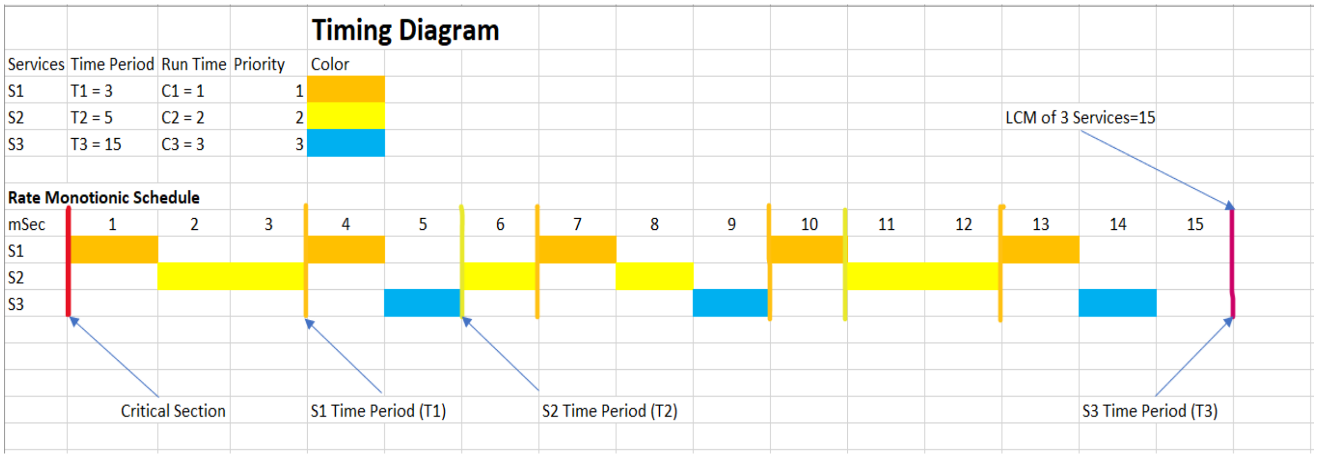
According to Rate Monotonic Algorithm Policy:

The T1 task has the shortest time period S1 and so, it's the highest priority service.

The T3 task has the longest time period S3 and therefore is the lowest priority service.

The **Service Run Time (C)** of the three tasks are : C1 =1, C2 =2 and C3 =3

Timing Diagram for three services S1, S2 and S3



Services	Time Period	Run Time	Priority	Color
S1	T1 = 3	C1 = 1	1	🟧
S2	T2 = 5	C2 = 2	2	🟨
S3	T3 = 15	C3 = 3	3	🟦

Rate Monotionic Schedule

mSec	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S1	🟧			🟧			🟧			🟧			🟧		
S2		🟨	🟨			🟨		🟨			🟨	🟨			
S3					🟦				🟦					🟦	

Figure 1: Rate Monotonic Schedule Timing Diagram

Utility & Method Description

i) According to the RM Schedule policy, all the services should have the Run times less than the time period for the services to be schedulable. This implies that the services should be executed with their run times before the next arrival of the service task place which happens when the time period has elapsed.

ii) In the RM Schedule for the 3 tasks S1, S2 and S3, S1 has the highest priority followed by S2 and S3 respectively.

iii) In the interval of 15 mSec, Task S1 gets executed 5 times, Task S2 gets executed 3 times while S3 task having maximum time period gets executed 1 time.

iv) The feasibility of the RM Schedule for these three services can be found out by comparing the CPU Utilization with the Least Upper Bound CPU Utilization.

Feasibility & Safety Issues

i) From the RM Schedule, we can deduce that all the 3 tasks get executed atleast once before the LCM of their schedules i.e 15. Therefore, the RM schedule for these 3 services seems **feasible.**

ii) To find if this RM Schedule policy for the 3 services is safe or not, we can use the mathematical function for Least Upper Bound CPU Utilization as discussed in Liu Layland's paper.

iii) The mathematical function is described below.

$$U = \sum (Ci/Ti) <= m (2^(1/m) - 1)$$

i.e The summation of the CPU utilization of all the services should be less than or equal to the Least Upper Bound CPU Utilization.

The Notations in the equation are described below:

U = CPU Utilization summation
Ci = Run time of ith service
Ti = Time period of ith service
m = Number of services

Therefore,

U = (C1 / T1) + (C2 / T2) + (C3 / T3)
   = (1 /3) + (2 / 5) + (3 / 15)
   = 0.9333

Therefore , Total CPU Utilization will be 0.9333 or 93.33 %

LUB = m * (2^(1/m) - 1)
      = 3 * (2^(1/3) – 1)
      = 0.7798

Therefore, the least upper bound CPU Utilization is 77.98 % or 0.7798

Therefore, the CPU Utilization is less than the Least Upper Bound and so the Liu Layand’s equation fails.

As a result, this proves that the RM Schedule for these three services is feasible but it isn’t safe due to failure of Liu Layand equation.

Question 2 : [20 points] 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?

Solution:

i) Account of Apollo 11 mission's success

a) Flight Routine Task Allocation: As in the Apollo Lunar Surface Journal, two young experts Margaret Hamilton and Don Eyles who worked at the MIT Instrumentation Lab - Draper Lab designed an excellent code which kept the Apollo 11 Lunar Mission from failing. They were given the responsibility of programming LM powered flight routine.

b) Memory Structure: The 36,864 15-bit words called the "Fixed" memory, now known as the ROM, and 2048 words of "Erasable" memory, now RAM were the memory constraints the two scientists had to work with. Most of the executable code was written into the fixed memory, including some constants and other similar data whereas the erasable memory was written with variable data, counters, etc. Since there was very little erasable memory available, the expert scientists were coerced to use the same memory location for different tasks.

c) Process involved in design of the algorithm: The process of code design involved carrying out various tests in order to ensure that different memory was used by each program while running simultaneously. The ingenious scientists developed a real-time multi-tasking operating system way before Bill Gates did. The operating system designed by them was interrupt-driven, had time-dependent tasks and consisted of priority ordering of tasks. Each task was given a "core set" of 12 erasable memory locations. In case more temporary storage was required by a job, scheduling request was generated for a VAC - vector accumulator - which consisted of 44 erasable words. It had 5 VAC areas and seven core sets. In case the task needed extra VAC area, the operating system created by the two scientists scanned the five VAC areas in order to find a free area. Likewise, after a free VAC area was found, the core sets were also scanned in order to search for a free set. VAC area scanning was omitted if the scheduled request specified "NOVAC" and in case of non-availability of VAC areas, the program branched into the Alarm/Abort routine and Alarm 1201 was set. Whereas if core sets were not available, the program branched into Alarm/Abort and Alarm 1202 was set.

d) How the algorithm averted the mission failure? During Apollo 11, recursive jobs to process radar data were scheduled due to a misconfiguration of radar switches. This led to filling up of the core sets which in turn set the 1202 alarm. Later during the landing of Apollo 11, the 1201 alarm also blew which was due to the scheduling request causing actual overflow was one that had requested the VAC area. Even though such error occurred it was checked by the flawless programming which distinguished data as primary and secondary. The secondary data was ignored while important computations having high priority were in progress. Before releasing the software, it had undergone rigorous testing. It was programmed to reboot and reinitialize the computer. Selected programs were then restarted at a point in their execution when it was restarted. Hence on Apollo 11, every time a 1201 or 1202 alarm blew, the computer underwent reboot and restarted the important activity such as steering the descent engine or running the DSKY tell the crew about the happenings. Hence the program made sure that all the erroneous radar jobs did not restart.

ii) Root cause of overload on Apollo 11

a) The root cause of overload on Apollo 11 was due to misconfiguration of radar switches which in turn led to the scheduling of recursive radar data processing.

b) The consequence of the error in radar switches led to filling up of the core sets and VAC areas and hence generated the 1202 alarm and 1201 respectively.

c) Hence due to the erroneous radar processing during landing there was an overflow of core sets and VAC areas.

iii) How Apollo 11 mission defied the Rate Monotonic Policy?

a) Apollo 11 lunar surface landing mission certainly breached the Rate Monotonic Policy. In Rate Monotonic Policy, each task is assigned fixed priorities to maximize the “schedulability”.

b) The policy was violated since a low priority task replaced a frequently occurring high priority task in real time in order to avoid the erroneous radar data collection during landing.

iv) Plotting of Least Upper bound as a function of number of services

No. of 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

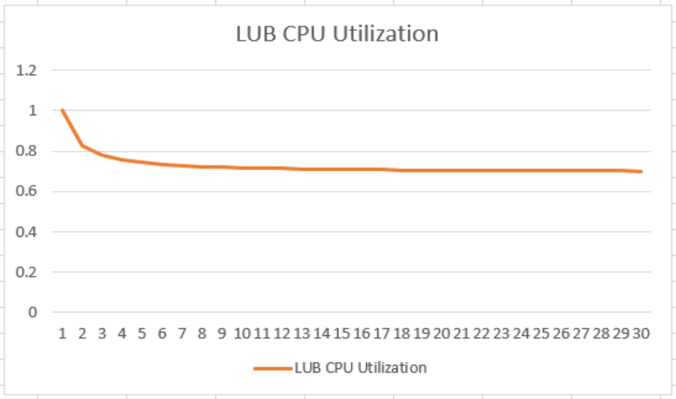


Figure 2: LUB CPU Utilization vs Services

According to the Liu and Layland's paper, the ratio of least upper bound to processor utilization factor for this algorithm is around 70% for the large task sets.

v) Assumptions

The assumptions are as follows-

a) Every task must finish before the request for next task is generated i.e. the deadlines consist of run-ability constraints.

b) Initiation and completion of requests of tasks do not affect the other tasks i.e. they are not dependant.

c) Task execution time is constant for a particular task and hence it does not change with time.

vi) Three aspects of LUB derivation that we didn't understand

a) In Theorem 4, C1 = T2 – T1 signifies the difference between periods of two tasks. In the next step C1 = T2 – T1- Δ, we did not understand the significance of the Δ; as to why they added that entity.
b) gj = 2^ (m-j)/m – 1 where j=0, 1, …..,m-1 is the general solution and it has been stated in the paper that U = m(2^(1/m)- 1) follows the above equation. We do not understand how exactly and for which value of j does our processor utilization equation follows the general solution.
c) In Theorem 5, τi task is substituted by τi' such that Ti' = qTi and Ci' = Ci. Also Cm is increased by a value which if used could utilize the processor to it's optimum. This increase in value is Ci (q - 1). We did not understand from where exactly did they derive this increase.

vii) <u>RM analysis of the Apollo 11 1201/1202 errors and potential mission abort</u>

a) Rate Monotonic Policy asserts that the tasks which have a higher request rate or which occur at a high frequency are given higher priority than any other task.
b) If Rate Monotonic Policy would have been used, the task of collecting erroneous radar data would be given higher priority, hence the mission would have been definitely aborted as the VAC and core sets would have gotten filled completely.
c) Thus, compromising the mission. Instead, the mission failure was successfully averted as a different scheduling policy was used in which importance was given to highly urgent events or tasks, thus giving least importance to tasks having high request rates.

Question 3: [20 points] Download the code from http://mercury.pr.erau.edu/~siewerts /cec450/ code/ RTClock/ or from Canvas and build it on the Altera DE1-SOC, TIVA or Jetson board 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 :

i) Program Intent: The given program aims to demonstrate the POSIX real time clock which is able to schedule system events within strict time constraints. In this program code, we are basically calculating the timing difference between the start and the end of pthread execution. The functional blocks of this code are explained in subsequent points.

ii) Pthread and its library functions in code: Pthreads are the tasks used to implement multithreaded applications. Following are some of the pthread library functions with their description used in this code.

   a) pthread_attr_init : This function uses default values to set the attributes of the object the attr is pointing to.
   b) pthread_attr_destroy : This function is used to destroy the thread attributes object which have served the purpose and won't be useful anymore. The other threads created by that object remains unaffected.
   c) pthread_attr_setinheritsched : This schedular attribute decides whether the thread created using thread attributes object attr will inherit the scheduling attributes from it or whether the calling thread will provides those attributes.
   d) pthread_attr_setschedpolicy : It provides the value specified in the policy to the scheduling policy attribute of the threads attribute object referred by attr
   e) pthread_create : Allows the calling process to begin a new thread by invoking start_routine.
   f) pthread_join : It will terminate the calling thread execution until the target thread ends. It returns 0 if there's no error or returns error number.

iii) Clock_gettime() : Specifies the current time of the clock determined by the clock_id. The second argument is the timespec struct which is a structure for storing time in seconds and milliseconds. Linux kernel supports clocks such as CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID etc. The code uses CLOCK_REALTIME and some privileges are required to set this clock. Discontinuous jumps in system time causes the alterations in this clock. We used this over

CLOCK_MONOTONIC because of better precision. The rtclk_stop_time and rtclk_start_time can be used to provide task execution time.

iv) Delay_test() : In this function, we use clock_getres for getting high resolution clock. Also, we start the timestamp by calling the clock_gettime function and then we put the kernel to sleep using nanosleep function. As soon as the remaining time of the sleep is over, again the timestamp is obtained when the kernel comes out of sleep i.e after 3 seconds. Finally, the end_delay_test function prints the start, stop, delta and the sleep time.

v) Delta_t() : As the mathematical meaning of the word, this function calculates the difference between the start and stop time and this time values are provided to the timespec structure in this code.

vi) RUN_RT_THREAD : This preprocessor directive is used along with #ifdef structure to find out the scheduling policy implemented in the code. There are different scheduling policies such as SCHED_IDLE, SCHED_BATCH, SCHED_OTHER etc. On using RUN_RT_THREAD, the scheduling policy used is SCHED_FIFO. If we use, SCHED_FIFO, it will always preempt SCHED_IDLE, SCHED_BATCH and SCHED_OTHER. If it is not used , then the scheduling policy implemented in the code is SCHED_OTHER. SCHED_OTHER is the most default time-sharing schedular. It follows round robin policy and not any real-time mechanism as such.

<u>Low Interrupt Handler Latency</u>:

i) Interrupt handler is executed to service the interrupts and plays a crucial role in real time embedded system efficiency.

ii) Interrupt latency is used to calculate the time elapsed between the arrival of interrupt and the time when PC vectored to the execution of the first instruction in the interrupt handler.

iii) Interrupt handler latency has a direct relation with the pace at which task is executed and whether the deadline is met or not.

iv) Low interrupt handler latency is preferred because having a high interrupt latency can have several consequences which are explained below in subsequent points.

v) The kernel executes the important and sensitive critical sections and disables the interrupts so that the interrupts don't disturb their execution.

vi) As a result, if the interrupt handler has high latency due to unoptimized and long interrupt handler then there is a high probability that the kernel will disable that interrupt and halt its handler execution more number of times compared to interrupt handler with low latency. Furthermore, if that high latency interrupt has high priority then chances are that the high priority task may miss its deadline due to being disabled a multiple number of times.

vii) Also, the interrupt handler can access different sections of memory like the cache, internal RAM and external memory for its execution> This can also add up to latency. A good practice is to avoid much of memory access and use flags in interrupt handler.

viii) Additionally, complex tasks in interrupt handler can be replaced with simple and processing time-friendly functions. For example, arithmetic multiplication inside interrupt handler can be replaced by bit manipulation functions such as bit rotations to execute the same task but with reduced latency.

Low Context Switch Time:

i) In real time embedded systems, context switching is considered as an important indicator for the CPU proficiency in the real time task execution. It is defined as the concept of storing the state of the currently executing process or thread before the CPU switch to the execution of some other task.

ii) Whenever the CPU switches the execution from one task to other, the CPU needs to store its CPU register values and the stack pointer value and the state of the machine for that process i.e the context of the process and this data is restored when the process is resumed again, this context is also restored with the process.

iii) The time taken for this process as well as to acknowledge the interrupt indicating the availability of this data can be referred as context switch latency.

iv) Context switch time is lowest if it takes place between two threads of same process and will be comparatively more between two different processes.

v) Low context switch time indicates that less CPU processing cycles are involved to store the context and to retrieve that data.

vi) Thus, low context switch time results in better responsiveness and performance of the real time system which might be essential in meeting the hard real time deadlines.

Stable timer services where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift. Why are each important?

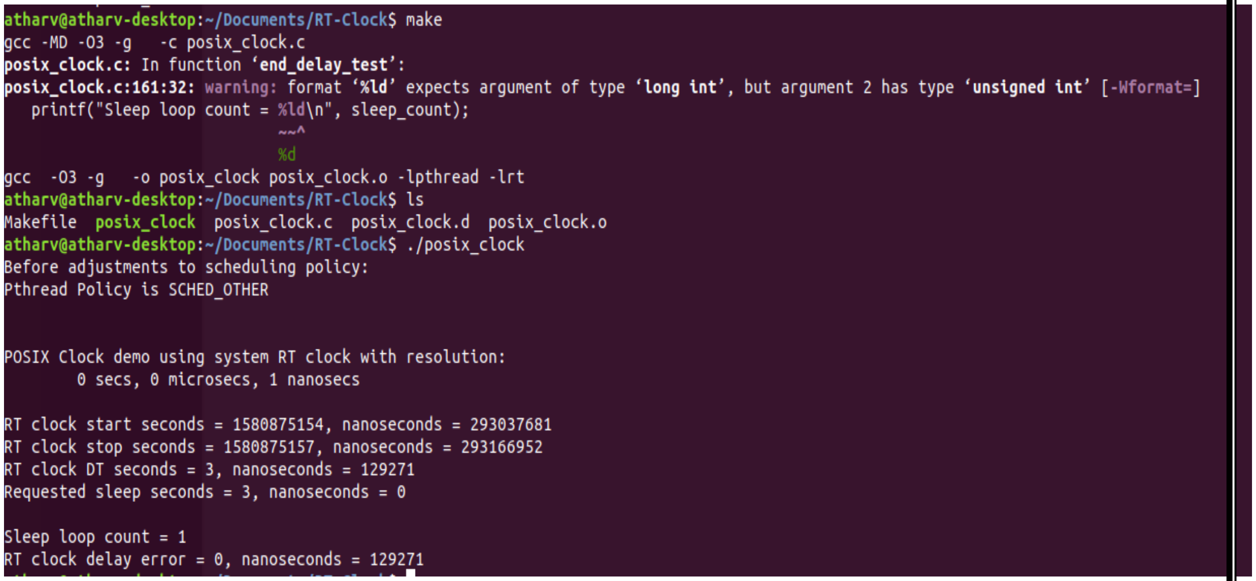
i) For embedded systems working in real time environment, the timers need to be highly accurate and stable since delay caused in hard real time systems can be even catastrophic.

ii) The timer services can be affected due to variety of external factors such as jitter and drift.

iii) Jitter is basically the inconsistency in the latency or the timing of the process execution. With each iteration, if there are fluctuations in the latency of the process operation, then it is called as jitter. This can be caused due to faults in the oscillator frequency associated with the system. While the system drift indicates that how much will the real time clock drift over time.

iv) Jitters do occur specially when the timeouts and timer intervals takes place and these conditions, if occurring frequently can cause significant delays in the systems which will degrade the system performance.

v) Thus, stable timer services are inversely proportional to jitter and drifts and these two parameters should be reduced as much as possible to improvise the overall system efficiency.



```
atharv@atharv-desktop:~/Documents/RT-Clock$ 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 argument 2 has type ‘unsigned int’ [-Wformat=]
   printf("Sleep loop count = %ld\n", sleep_count);
                              ~~^
                              %d
gcc  -O3 -g   -o posix_clock posix_clock.o -lpthread -lrt
atharv@atharv-desktop:~/Documents/RT-Clock$ ls
Makefile  posix_clock  posix_clock.c  posix_clock.d  posix_clock.o
atharv@atharv-desktop:~/Documents/RT-Clock$ ./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 = 1580875154, nanoseconds = 293037681
RT clock stop seconds = 1580875157, nanoseconds = 293166952
RT clock DT seconds = 3, nanoseconds = 129271
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 129271
```

Figure 3: Posix_clock Code Execution

<u>Do you believe the accuracy provided by the example RT-Clock code?</u>

i) The posix_clock.c code was executed on the Jetson Nano and therefore has used the timer services of this NVIDIA system. However, as discussed in the above question, the timer services can be affected by the slight errors in clock oscillator frequency or jitters too.

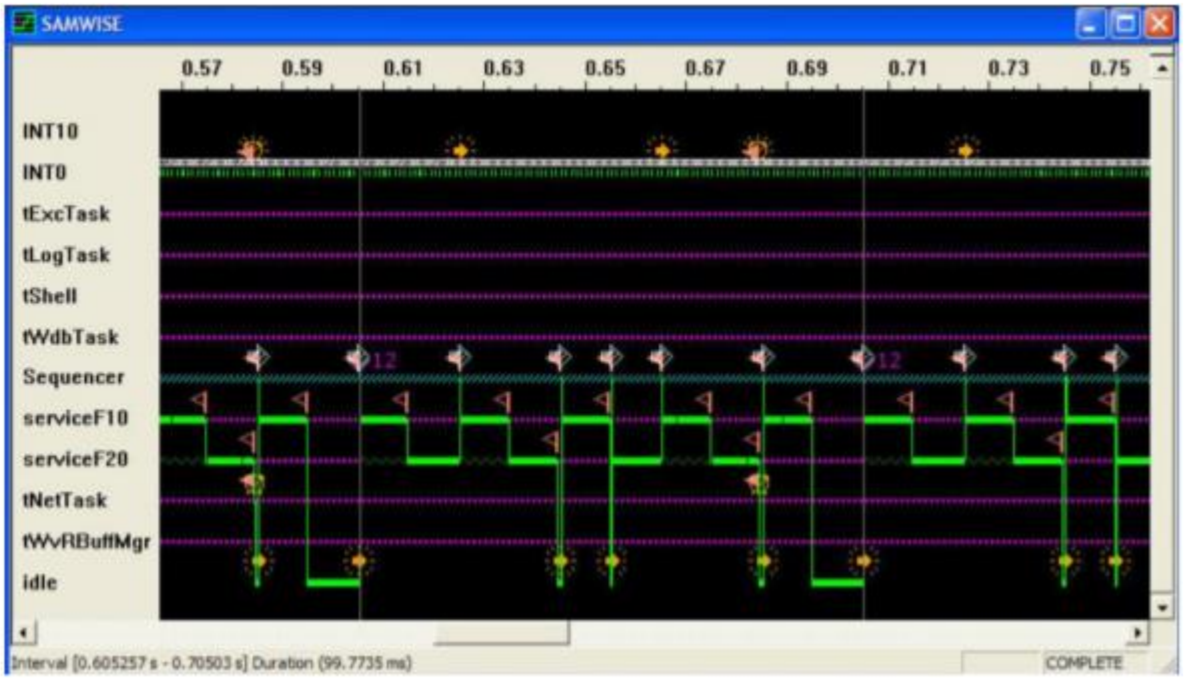
ii) As observed in the attached screenshot below of the output for RT-Clock Code, we expected a delay of 3 seconds and 0 nanoseconds.

iii) However, the observed delay was of 3 seconds and additional 129271 Nanoseconds.

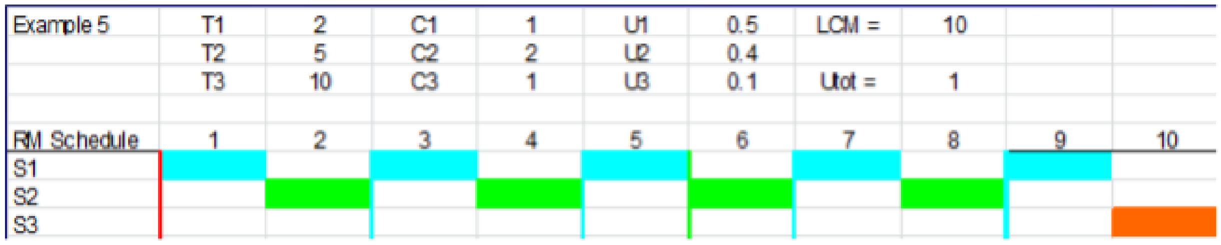
iv) For now, it seems that this error can be ignored but in future, due to multiple iterations and processes execution with time, this error can be significant and can affect the performance and results for hard real time systems.

v) Therefore, considering the substantial increase in the percentage of error with time, the RT-clock code doesn’t seems to be accurate.

Question 4 : [40 points] This is a challenging problem that requires you to learn a bit about pthreads in Linux and to implement a schedule that is predictable. Download, build and run code in the following three example programs: 1) simplethread, 2) rt_simplethread, and 3) rt_thread_improved and briefly describe each and output produced. [Note that for real-time scheduling, you must run any SCHED_FIFO policy threaded application with "sudo" – do man sudo if you don't know what this is]. Based on the examples 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:



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										🟧

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 a Virtual Machine, or on the Jetson, Altera or TIVA system (they are 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. Implement the equivalent code using Linux and pthreads to replicate the LCM invariant schedule

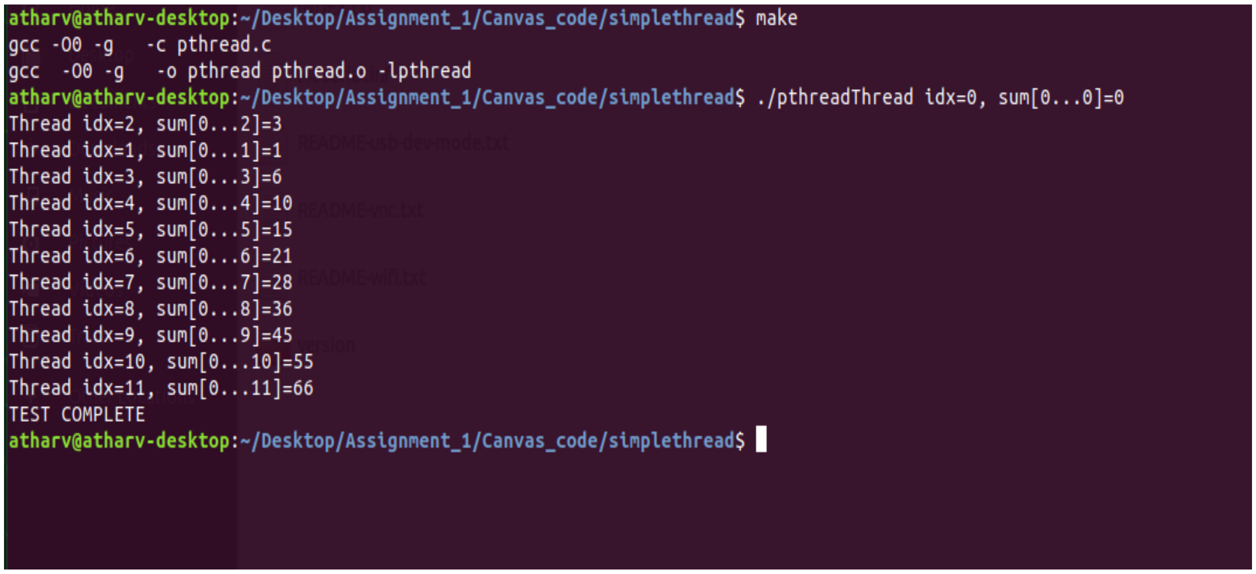
Solution:

i) PROGRAM NAME: simplethread

a) Program Intent:- This program creates simple threads and calculates the sum of all the threads one after the other till the thread ID specified.

b) Function int main (int argc, char *argv[]) : The main function consists of a FOR loop. It runs from int i to the NUM_THREADS macro value specified. This line of the code "threadParams[i].threadIdx=i" updates the member of the structure which is in turn the Thread ID. It also contains code for creating a thread which consists of parameters like pointer to a thread descriptor, attributes, thread function entry point and parameters to pass. The pthread_join() attaches the threads together to one CPU core and waits for termination of the thread.

c) Function void *counterThread(void *threadp) : This function contains a FOR loop to calculate the and print the Thread ID as well as the sum of the threads until the specified thread ID. The FOR loop runs till the Thread ID passed.



```
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code/simplethread$ make
gcc -O0 -g   -c pthread.c
gcc  -O0 -g   -o pthread pthread.o -lpthread
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code/simplethread$ ./pthreadThread idx=0, sum[0...0]=0
Thread idx=2, sum[0...2]=3
Thread idx=1, sum[0...1]=1
Thread idx=3, sum[0...3]=6
Thread idx=4, sum[0...4]=10
Thread idx=5, sum[0...5]=15
Thread idx=6, sum[0...6]=21
Thread idx=7, sum[0...7]=28
Thread idx=8, sum[0...8]=36
Thread idx=9, sum[0...9]=45
Thread idx=10, sum[0...10]=55
Thread idx=11, sum[0...11]=66
TEST COMPLETE
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code/simplethread$
```

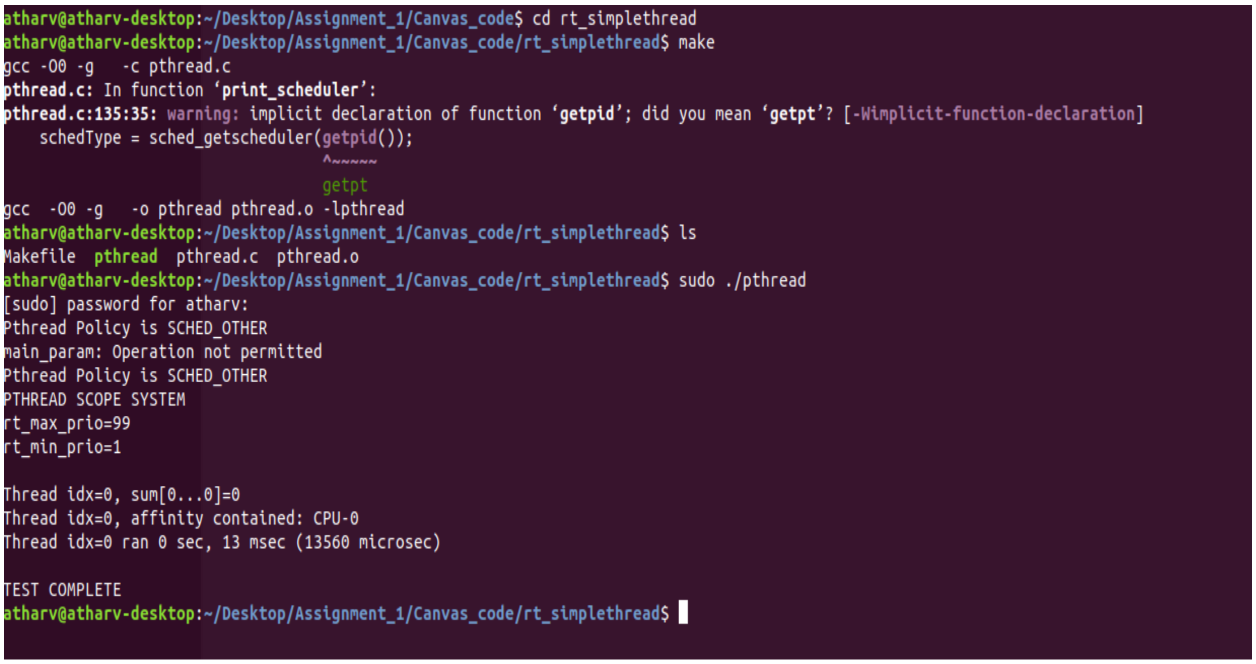
Figure 4: Simplethread Code execution

ii) PROGRAM NAME: rt_simplethread

a) Program Intent: This program is written to keep a track of the scheduling used by the thread, start and end of the pthread execution time and also prints it on the terminal on execution.

b) Function int main (int argc, char *argv[]) :This function consists of the code for setting the CPU to CPU zero. The FOR loop runs one time since the macro NUM_CPUS is defined with value one. The maximum and minimum priority set by the schedular for the FIFO schedular is set and saved. "print_schedular()" function is called and the current scheduling policy is printed. The maximum priority is then set and the scheduling policy is also set to SCHED_FIFO and "print_schedular()" function is called again. The scope is determined that where it is a PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS or if the pthread scope is unknown. Also just to keep a record of the maximum and minimum priorities, they are printed using the variables used for storing them. Next in the main function, the attributes are set. The attributes are first initialized, inherit schedular is set, scheduling policy and the affinity is set. The thread parameter's priority is also set using the maximum priority. The thread is then created using "pthread_create" in which the entry point is the "counterThread" and the parameters passed are the "threadParams[i]". The threads are bound to a CPU and it waits till termination. On execution "Test complete" message is echoed.

c) Function void *counterThread(void *threadp) : In this function, three objects of structure timespec are created and the members of the structure are initialized to zero. The three objects are as follows- start_time, finish_time and thread_dt which is the difference between two threads. The clock_gettime function is called inorder to record the start time of the execution of the thread. Sum of threads until the specified thread IDs and Fibonacci numbers are computed. "pthread_getaffinity_np(thread, sizeof(cpu_set_t), &cpuset)" is then called and the parameters are set. The thread ID and the affinity contained are printed together. The clock_gettime function is again called to record the end time of the thread. The difference between execution of two threads is also recorded and everything is printed together.



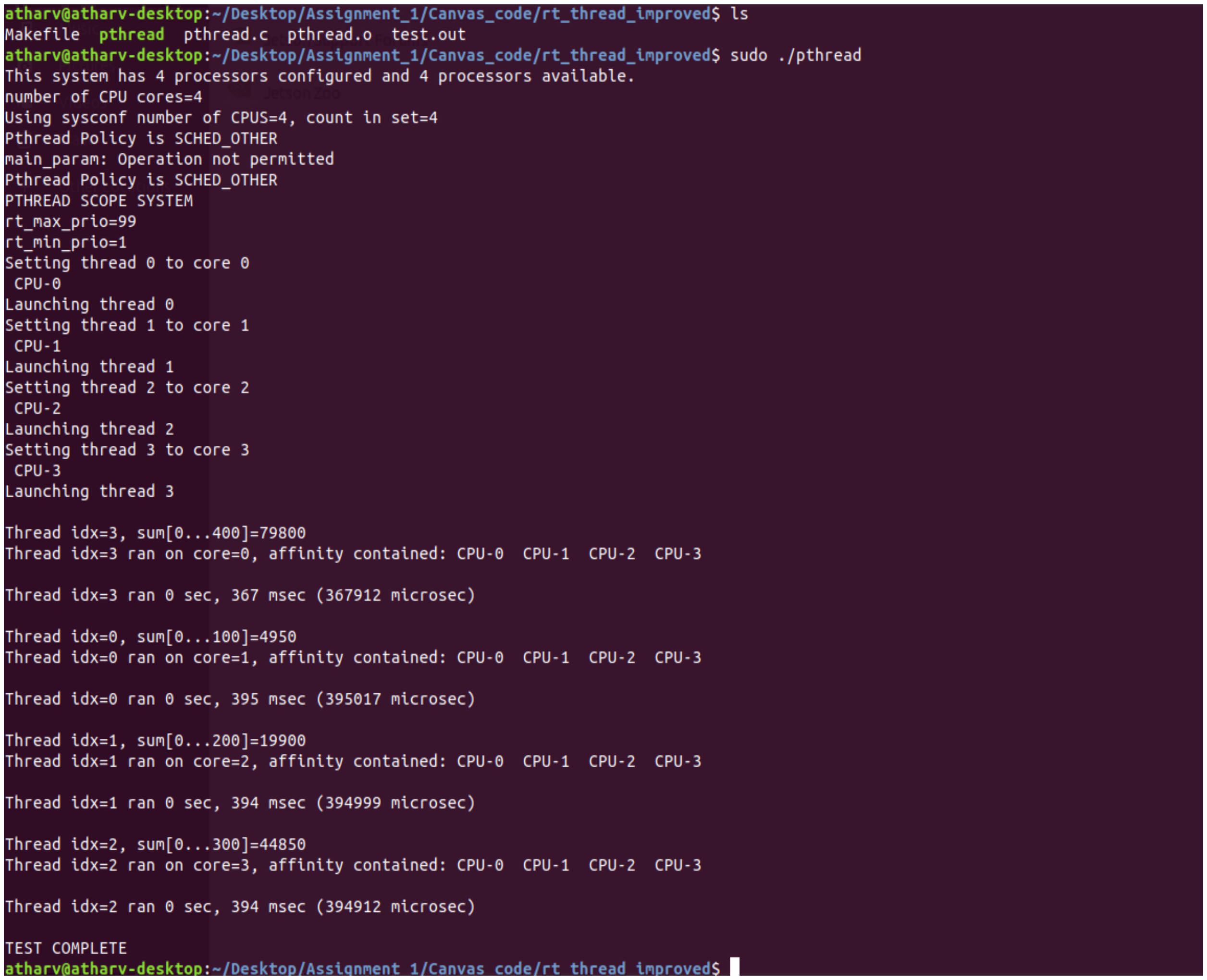
```
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code$ cd rt_simplethread
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code/rt_simplethread$ make
gcc -O0 -g   -c pthread.c
pthread.c: In function ‘print_scheduler’:
pthread.c:135:35: warning: implicit declaration of function ‘getpid’; did you mean ‘getpt’? [-Wimplicit-function-declaration]
    schedType = sched_getscheduler(getpid());
                                   ^~~~~~
                                   getpt
gcc  -O0 -g   -o pthread pthread.o -lpthread
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code/rt_simplethread$ ls
Makefile  pthread  pthread.c  pthread.o
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code/rt_simplethread$ sudo ./pthread
[sudo] password for atharv:
Pthread Policy is SCHED_OTHER
main_param: Operation not permitted
Pthread Policy is SCHED_OTHER
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1

Thread idx=0, sum[0...0]=0
Thread idx=0, affinity contained: CPU-0
Thread idx=0 ran 0 sec, 13 msec (13560 microsec)

TEST COMPLETE
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code/rt_simplethread$ 
```

Figure 5 : rt_simplethread Code Execution

iii) PROGRAM NAME: rt_thread_improved

a) Program Intent : This program is quite similar to the above program in that it keeps a track of the scheduling used by the thread, start and end of the pthread execution time and also prints it on the terminal on execution. The only difference between this program and the above one is that this one creates a pre-defined number of threads; macro defined is "NUM_THREADS 4" and it also uses different processors if possible.

b) Function int main (int argc, char *argv[]) : This function in the start of the code, prints the number of configured processors and the number of processors available using get_nprocs_conf() and get_nprocs() functions. The value of number of configured processors is saved in a variable for use at a later time. The CPU is then set. There on, the maximum and minimum priority set. The parameters and the scope are decided and the thread max and min priority is printed out. In the FOR loop, the attributes and thread parameters are initialized. The scheduling policy is set to SCHED_FIFO. The thread parameters ID is set at every iteration of the FOR loop. "pthread_create()" creates threads with the specified parameters. "pthread_join()" attaches all the threads to one core.



```
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code/rt_thread_improved$ ls
Makefile  pthread  pthread.c  pthread.o  test.out
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code/rt_thread_improved$ sudo ./pthread
This system has 4 processors configured and 4 processors available.
number of CPU cores=4
Using sysconf number of CPUS=4, count in set=4
Pthread Policy is SCHED_OTHER
main_param: Operation not permitted
Pthread Policy is SCHED_OTHER
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Setting thread 0 to core 0
 CPU-0
Launching thread 0
Setting thread 1 to core 1
 CPU-1
Launching thread 1
Setting thread 2 to core 2
 CPU-2
Launching thread 2
Setting thread 3 to core 3
 CPU-3
Launching thread 3

Thread idx=3, sum[0...400]=79800
Thread idx=3 ran on core=0, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3

Thread idx=3 ran 0 sec, 367 msec (367912 microsec)

Thread idx=0, sum[0...100]=4950
Thread idx=0 ran on core=1, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3

Thread idx=0 ran 0 sec, 395 msec (395017 microsec)

Thread idx=1, sum[0...200]=19900
Thread idx=1 ran on core=2, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3

Thread idx=1 ran 0 sec, 394 msec (394999 microsec)

Thread idx=2, sum[0...300]=44850
Thread idx=2 ran on core=3, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3

Thread idx=2 ran 0 sec, 394 msec (394912 microsec)

TEST COMPLETE
atharv@atharv-desktop:~/Desktop/Assignment_1/Canvas_code/rt_thread_improved$
```

Figure 6 : rt_thread improved Code Execution

iv) <u>Threading vs Tasking</u>

a) In threading, instructions of a program are managed by the schedular. Thread is a part of the process. A process can be made up of multiple threads. The execution of the threads is concurrent. Resources such as memory is shared by multiple threads.

b) Context switch between threads occurs at a faster rate than a process. Multithreading enables multiple threads to coexist in a single process.

c) Thread consists of its own stack and kernel resources such as registers. Therefore, threads adds up the CPU overhead due to its memory occupancy in stack and context switching. The overhead of several thread creation can be avoided by Threadpool.

d) Advantages of multithreading are as follows:- Responsiveness, faster execution, lower resource consumption, better system utilization, simplified sharing and communication, parallelization. Scheduling of threads is either pre-emptive or co-operative.

e) Linux follows threading mechanism where the process can be divided into multiple threads.
f) Therefore, these multiple threads are executed together by implementing context switching between the threads.
g) A task can be considered as a broader concept and consists of multiple threads. It is basically an asynchronous operation and also multiple tasks can run at the same time in parallel unlike multithreading. VxWorks uses the tasking concept whereas Linux uses the threading concept.
h) Linux OS uses pthread_create to create the threads. Similarly, VxWorks uses thread spawn which performs the similar function.
i) In Vxworks, multiple tasks can run at the same time because the memory is divided in blocks which can be used in parallel.

v) <u>Semaphores, Wait and sync</u>

a) In Real Time Embedded System, more than one threads are often needed to be executed concurrently so that they can access the shared resources in a mutually exclusive manner.
b) A semaphore is a kernel tool used to execute the critical sections and achieve synchronization between the processes in multithreading environment.
c) It consists of two types and follows lock or signaling mechanism.
d) Following are some of the semaphore functions used in this code to established synchronization between the threads and follow locking mechanism.
    1) Sem_init() – This function takes three arguments namely sem pointer, pshared and value. The semaphore object is initialized using sem pointer. Pshared flag decides whether the semaphore would be shared with orked processes. Value is the initial value which the semaphore holds.
    2) Sem_post() – performs the semaphore unlock operation on that semaphore and its positive value indicates that whenthe semaphore was being unlocked, none of the other threads were blocked for it. This function returns 0 without error and -1 if error occurs.
    3) Sem_destroy() – It is used in this code to delete or destroy the semaphore pointed by sem. Generally, the semaphore created by sem_init functions are destroyed using sem_destroy.
    4) Sem_wait() – Performs lock operation on the semaphore pointed by the sem. Here, the non-zero semaphore value is decremented which specifies that the locking.

e) Sem_wait () : This function returns zero on completion. The semaphore might be still busy and may need further blocking. In such cases, sem_wait() is used. The general syntax of a semaphore uses sem_wait() and sem_sync() for synchronization.

vi) Synthetic Workload Generation :

a) As the name suggests, synthetic workload generation is basically the adding up of computational overhead on the CPU through execution of code.
b) In real time embedded system, this technique can be used to calculate the throughput of the test system by executing the code for a definite amount of time.
c) In the assignment, we implemented the Fibonacci series code with multiple iterations to get a particular amount of time.
d) The code (from Sam Siewert Vx-works Sequencers) with following snippet is used to run on test system to generate synthetic workload.

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

vii) Synthetic workload analysis and adjustment on test system :

a) The code which we are running on test system with two threads which are Fibonacci for 10 mSec or 20 mSec which indicates that their computation times are 10 mSec and 20 mSec respectively.
b) In the code, we defined the macro FIB_LIMIT_FOR_32_BIT as 47. This implies that the Fibonacci loop will be iterated for 47 times.
c) The second argument for iterations can be used to adjust and generate a particular amount of time. For example, the Pentium processors can use 170000 iterations to generate a time of 10 mSec.
d) Therefore, varying the loop iteration values will generate appropriate amount of synthetic workload.

viii) Overall good description of challenges and test/prototype work

a) When we started, we had very less knowledge about pthreads.
b) We referred posix manual pages and reviewed the sample codes provided in the assignment.
c) It proved to be of great help in understanding how to create pthreads, how to set pthread attributes, scheduling policies to use etc.
d) It was a great challenge to understand how the rt_simplethread program kept a track of the start time, end time and time difference of the running tasks.
e) Writing a program to replicate the original VxWorks Code was a difficult task in itself.
f) Understanding how semaphores work and how to schedule two tasks using it a demanding task.
g) Most of our time was spent in loading the SD card image and validating it using Balena Etcher.

ix) VxWorks Code Explanation

a) The code designed by Professor. Sam Siewart, performs the task of scheduling two tasks to create a Fibonacci Series.
b) It is basically the LCM invariant schedule that we studied in this exercise. The VxWorks does not have a main function instead it has a start function which spawns tasks which is analogous to task creation.
c) The task that is spawned is the "Schedular" task. This is in turn the actual schedular which synchronizes the two tasks fib10 and fib20. The schedular code consists of semaphore which ensures the signaling and blocking of the two tasks.
d) This schedular creates two tasks serviceF10 and serviceF20 and prints out messages such as spawning successful or fail.
e) This function also calls an API function wvEvent for logging purpose.
f) It also consists of a sequencing loop for LCM phasing of F1 and F2 services which will ensure the flow of scheduling events with a task delay task called in between.
g) The schedular then starts scheduling the task using the "semGive" command.
h) First the semaphore is given to the serviceF10 and the second task waits till then.
i) Once both the tasks get the semaphore, the task with higher priority get the chance to get executed.
j) ServiceF10 get the semaphore and it starts executing till then there is a delay, later once the semaphore becomes available to the serviceF20, even that task starts executing while serviceF10 waits for the semaphore. While this juggling between the tasks takes place, the schedular is halted.
k) Hence semaphores ensure that the two tasks are scheduled only when the tasks get the semaphore.

x) <u>Linux Code Explanation</u> :

a) As discussed in the above section, this section consists of the interpretation of test code in Linux which is pretty analogous to the VxWorks Code.

b) Since Linux code deals with threads instead of tasks as in VxWorks, the code consists of three threads namely the sequencer thread and the Fibonacci threads for 10 mSec and 20 mSec respectively.

c) The start routine in VxWorks is replaced by main. The main can also function as the sequencer thread which handles the creation of the other threads such as Fibonacci 10 and 20 mSec threads.

d) Therefore, the sequencer thread has been allotted the highest priority followed by the 10 mSec Fibonacci thread and the 20 mSec thread gets the least priority. These priorities and attributes of the threads are set when they are created.

e) The sequencer manages the semaphore for the other two threads which work according to that semaphore status.

f) Finally, the code is terminated after the thread execution has been performed.

References:-

1)http://mercury.pr.erau.edu/~siewerts/cec450/code/VxWorks-sequencers/lab1.c
2)http://ecee.colorado.edu/~ecen5623/ecen/labs/Linux/IS/Report.pdf
3) https://www.hq.nasa.gov/alsj/a11/a11.1201-pa.html
4)http://mercury.pr.erau.edu/~siewerts/cec450/documents/Timing_Diagrams/
5) http://mercury.pr.erau.edu/~siewerts/cec450/documents/Papers/liu_layland.pdf
6)http://mercury.pr.erau.edu/~siewerts/cec450/code/RT-Clock/
7)https://microcontrollerslab.com/rate-monotonic-scheduling-algorithm/
8)https://www.linkedin.com/pulse/real-time-operating-systems-rtos-semaphores-part-1-ahmed/
9)https://www.geeksforgeeks.org/semaphores-in-process-synchronization/
10)http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html
11)https://pubs.opengroup.org/onlinepubs/009695399/functions/sem_post.html
12)https://blog.slaks.net/2013-10-11/threads-vs-tasks/
13)https://www.c-sharpcorner.com/article/task-and-thread-in-c-sharp/
14)https://www.embedded.com/introduction-to-rate-monotonic-scheduling/
15)https://en.wikipedia.org/wiki/Thread_(computing)