

# RTES Exercise -1

## Answer -1 :-

In computer science, rate-monotonic scheduling (RMS) is a priority assignment algorithm used in real-time operating systems (RTOS) with a static-priority scheduling class. The static priorities are assigned according to the cycle duration of the job, so a shorter cycle duration results in a higher job priority.

These operating systems are generally preemptive and have deterministic guarantees with regard to response times. Rate monotonic analysis is used in conjunction with those systems to provide scheduling guarantees for a particular application.

According to rate monotonic policy, threads should have following properties

- Deterministic deadlines are exactly equal to periods
- Static priorities (the task with the highest static priority that is runnable immediately preempts all other tasks)
- Static priorities assigned according to the *rate monotonic* conventions (tasks with shorter periods/deadlines are given higher priorities)
- Context switch times and other thread operations are free and have no impact on the model

According to given question, the values are :-

**T1= 3 C1 = 1 T2 = 5 C2= 2 T3 =15C3 =3**

Where T – Period, C – Computation Time

CPU Utilisation(U) :- It is a measure on how busy the processor could be during the shortest repeating cycle.

The formula for finding U is  **$U=C/T$** .

If  **$U>1$**  = It is called **Overload**. It means some tasks will fail to meet its deadline no matter what algorithms you use.

**$U\leq 1$**  = It will depend on scheduling algorithms. If  **$U = 1$**  and the CPU is kept busy , all deadlines will be met.

For the given example,

$$U_1 = C_1/T_1 = 1/3 = 0.33$$

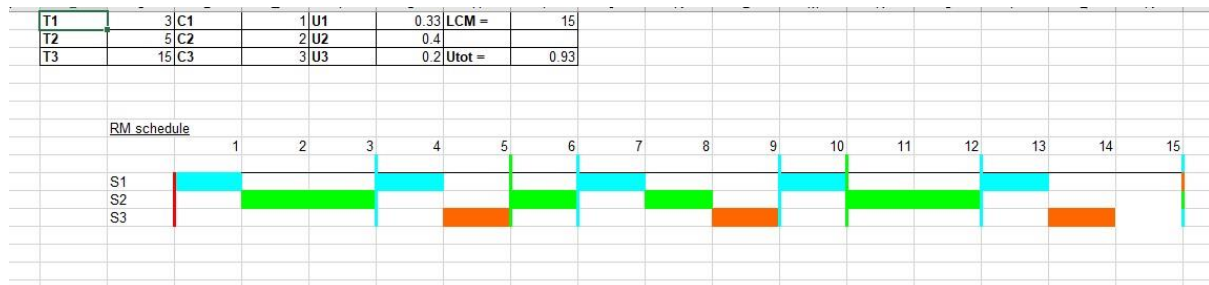
$$U_2 = C_2/T_2 = 2/5 = 0.4$$

$$U_3 = C_3/T_3 = 3/15 = 0.2$$

$$\text{Therefore, } U(\text{total}) = U_1 + U_2 + U_3$$

$$0.33 + 0.4 + 0.2 = 0.93$$

Thus, Total CPU utilisation will be **93%**.



**Figure -1 RM schedule**

### Method Description:-

The priority is defined as the , the one with the lowest period gets the highest priority . So here **S1>S2>S3** will be the order of priority.

The empty coloured boxes means CPU is not utilized.

The boxes are filled with colours according to their priority

The LCM determined in the figure, is the **LCM of 3,5,15 which is equal to 15**

Different column partitions are made according to the individual period.

### Feasibility and Safety Issues:-

For Feasibility issues, we can say the system **is feasible** because , it never misses a deadline as you can see in the graph. Thus we can say that it is mathematically repeatable as an invariant indefinitely.

But it is **not safe** since the utilisation is 93% and according to Liu Layland paper 30% should be left free for the tasks that are not real time.

**Answer 2 :-**

## **Apollo 11 lunar lander computer overload story summary**

The "Apollo Program alarms" paper indicates the details about the 1201 and 1202 program alarms that occurred during the Apollo 11 lunar landing. There were 2 people who worked on the LEM guidance computer, Peter Adler and Don Eyles at MIT Instrumentation Lab - Draper Lab. Don was responsible for LM P 60 ( Lunar Descent ) and Peter for responsible for LM P 40(Lunar Ascent )

In these computer, there were 36,864 15 bits of word called "Fixed memory" (today known as ROM) and 2048 words of Erasable memory(today known as RAM). Mostly, all of the executable code was in the Fixed memory along with constants and other similar data. Erasable memory was used for variable data, counters. With so little Erasable memory available, we were forced to use the same memory address for different purposes at different times.

These were multitasking systems. They were able to do interrupt driven and time dependent tasks. A real time operating system which was made much earlier than Bill gates developed.

e.g., turn the LM Descent Engine on at the correct time - as well as priority-ordered jobs that dealt with less time-critical things. Each scheduled job has some erasable memory to use while it was executing. This memory was used for intermediate computational results, rather data.

Each job was allocated a "core set" of 12 erasable memory locations. If a job required more temporary storage, the scheduling request asked for a VAC - vector accumulator - which had 44 erasable words. There were seven core sets and five VAC areas.

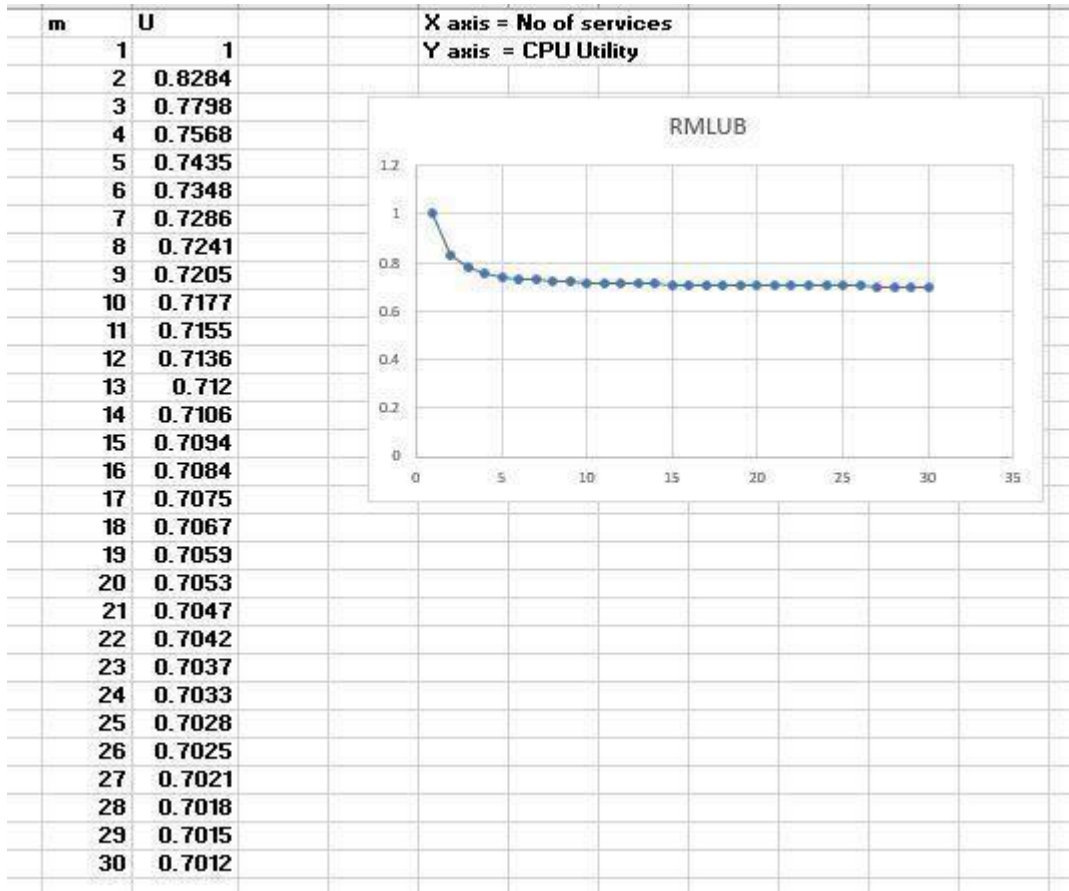
If the job to be scheduled required a VAC area, the operating system would scan the five VAC areas to find one which was available. After finding and reserving a VAC area, the core sets would be scanned to find an available core set. Scanning for a VAC area would be skipped if the scheduling request specified "NOVAC". In any case, if there were no VAC areas available, the program would branch to the Alarm/Abort routine and set Alarm 1201. Similarly, if no core sets were available, the program would branch to Alarm/Abort and set Alarm 1202.

### **Root cause Analysis and Discription:-**

The 2 alarms that went off during Apollo 11 mission were 1201 and 1202.

First of all alarm, 1202 was generated , this was because many jobs to process radar data were scheduled at the same time and the computer was not able to process everything at the same time. After that the alarm, 1201 was generated, this was because , at the time of landing the scheduling request that caused the actual overflow was one that had requested a VAC area. Now, utilisation of the processor has reached the limit and it was overflowing , thus high priority tasks were not met along with the normal tasks. This happened which made the computer to restart. But the computer which was made under supervision of Maragret Hamilton, was designed in such a manner that it was able to recognise which tasks are of higher importance , which tasks are more critical and required more attention. So when this restart happened, it opened the critical tasks which were happening before the shutdown and started working from the same place where it had stopped. Thus at the time of restart again the system did not show any specific alarms and thus it landed safely.

### Graph of RMLUB



**Figure -2 RMLUB Plot**

U = Total CPU utilisation

As we can see from the graph that as the graph increases i.e. as the value of m increases towards infinity the graph becomes more linear with the approximate value of 0.693 which is the value of natural log 2. Thus we can say that ,LUB value of RMP is 0.693. U should be 69.3% , if the task wants to achieve all deadlines.

### Key Assumptions of paper :-

- 1) The processor always executes the highest priority task.
- 2) Task priorities are assigned according to rate monotonic policy
- 3) Tasks do not synchronize with each other, meaning the request of one task does not depend of starting or completion of other task.
- 4) Each task's deadline is at the end of its period .

- 5) Tasks do not suspend themselves in the middle of computations 6) Context switches between tasks take zero time.

**Aspects of their fixed priority LUB derivation that I didn't understand:-**

- 1) Why this assumptions made? This part I did not understand

$$C_1 \leq T_2 - T_1 \lfloor T_2/T_1 \rfloor.$$

Thus, the largest possible value of  $C_2$  is

$$C_2 = T_2 - C_1 \lceil T_2/T_1 \rceil.$$

The corresponding processor utilization factor is

$$U = 1 + C_1[(1/T_1) - (1/T_2) \lceil T_2/T_1 \rceil].$$

- 2) Following the above assumption, moving down, this equation I didn't understand

$$U = 1 - (T_1/T_2)[\lceil T_2/T_1 \rceil - (T_2/T_1)][(T_2/T_1) - \lfloor T_2/T_1 \rfloor]. \quad (3)$$

For notational convenience,<sup>3</sup> let  $I = \lfloor T_2/T_1 \rfloor$  and  $f = \{T_2/T_1\}$ . Equation (3) can be written as

$$U = 1 - f(1 - f)/(I + f).$$

- 3) In the next theorem, the partial differential equation conversion I did not understand

Just as in the two-task case, the utilization bound becomes 1 if  $g_i = 0$ , for all  $i$ .

To find the least upper bound to the utilization factor, eq. (4) must be minimized over the  $g_j$ 's. This can be done by setting the first derivative of  $U$  with respect to each of the  $g_j$ 's equal to zero, and solving the resultant difference equations:

$$\partial U / \partial g_j = (g_j^2 + 2g_j - g_{j-1}) / (g_j + 1)^2 - (g_{j+1}) / (g_{j+1} + 1) = 0, \quad j = 1, 2, \dots, m-1. \quad (5)$$

The definition  $g_0 = 1$  has been adopted for convenience.

The general solution to eqs. (5) can be shown to be

$$g_j = 2^{(m-j)/m} - 1, \quad j = 0, 1, \dots, m-1. \quad (6)$$

It follows that

$$U = m(2^{1/m} - 1),$$

**Arguments for and against RM Policy :-**

The failure in the system occurred causing alarms going on , this was because of overloading of the tasks and not because of the incorrect priority handling of tasks, Now , rate monotonic policy is priority based mechanism. But it is not capable of calculating the possibility of number of events like in this case radar tasks, and then take them into consideration. These events cannot be measured on a fixed priority.

On the other hand, to overcome this problem, if all the events/operations (Prioritized tasks and Interrupt processes) have its own specific memory assigned in the RAM , then at the time of such a situation they would not overlap and thus such reboot would not have occurred. By doing this, overflow of the content can be removed.

### Answer :- 3

**Code details :-** the main objective of the code is to determine whether the clock provided by the Linux is real time accuracy or not. There is a function in the code called “delta\_t” which is used for finding the difference between 2 time intervals. Another thing in the code is the macro of RUN\_RT\_THREADS which helps to create a SCHED\_FIFO scheduling.

```
sleep_time.tv_sec=3; sleep_time.tv_nsec=0;
```

This determines that the difference between RT clock start seconds and Stop seconds should be exactly 3 sec. Anything more than that is the inaccurate value.

`clock_gettime(CLOCK_REALTIME, &rtclk_start_time);` This function is used to measure the delay between 2 intervals.

```
static struct timespec rtclk_dt = {0, 0}; static  
struct timespec rtclk_start_time = {0, 0}; static  
struct timespec rtclk_stop_time = {0, 0}; static  
struct timespec delay_error = {0, 0}; This
```

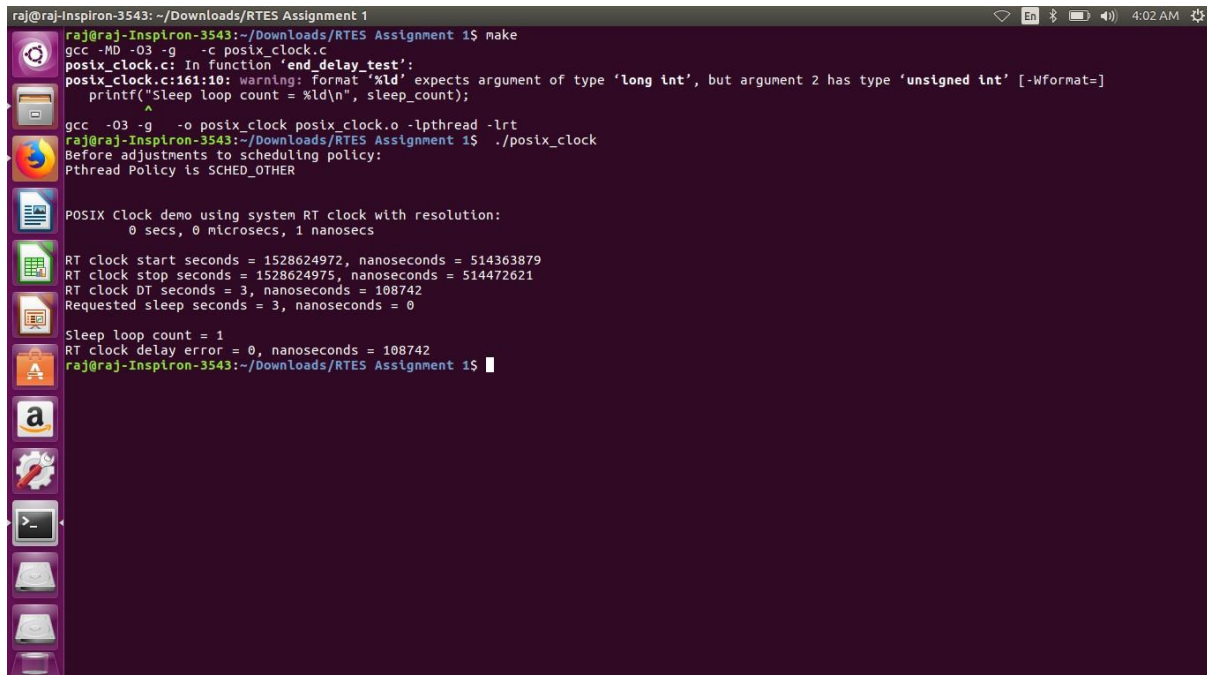
timespec is used in this function.

The interval is created with the help of nanosleep function which takes 2 para, timespec requested and timespec remaining.

There are 2 considerations for this example.

- 1) Without RT Thread

## 2) With RT Thread



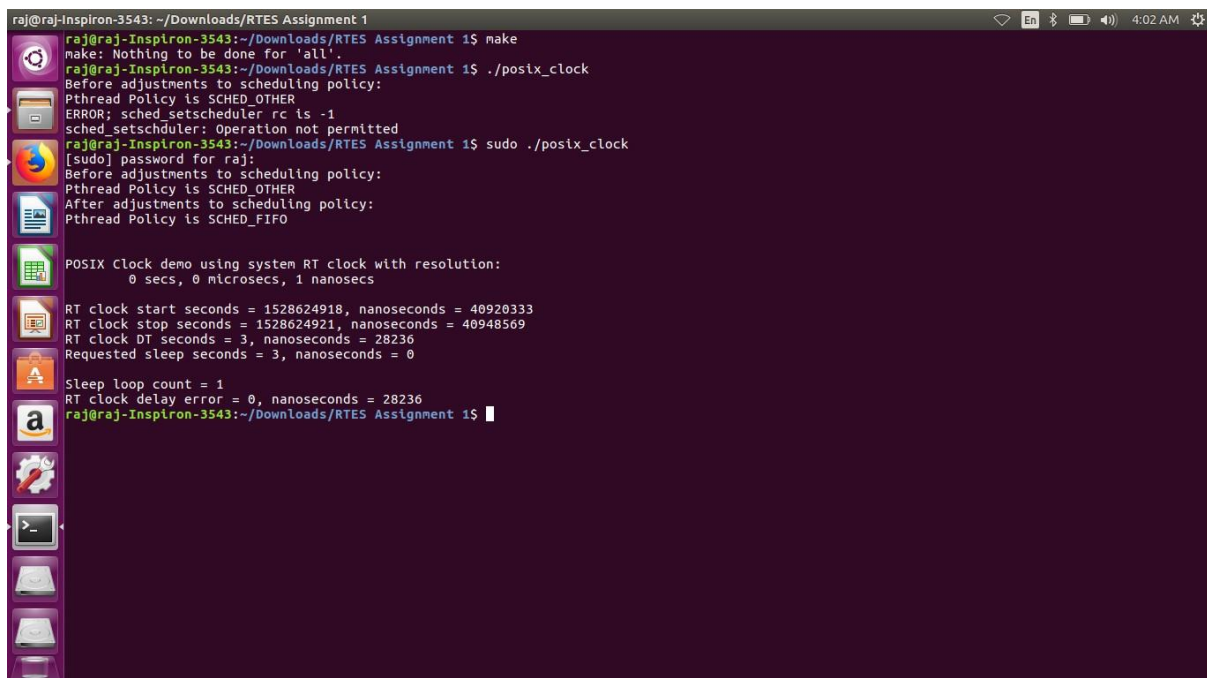
```
raj@raj-Inspiron-3543: ~/Downloads/RTES Assignment 1
raj@raj-Inspiron-3543:~/Downloads/RTES Assignment 1$ make
gcc -MD -O3 -g -c posix_clock.c
posix_clock.c: In function 'end_delay_test':
posix_clock.c:161:10: warning: format '%ld' expects argument of type 'long int', but argument 2 has type 'unsigned int' [-Wformat=]
printf("Sleep loop count = %ld\n", sleep_count);
^
gcc -O3 -g -o posix_clock posix_clock.o -lpthread -lrt
raj@raj-Inspiron-3543:~/Downloads/RTES Assignment 1$ ./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 = 1528624972, nanoseconds = 514363879
RT clock stop seconds = 1528624975, nanoseconds = 514472621
RT clock DT seconds = 3, nanoseconds = 108742
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 108742
raj@raj-Inspiron-3543:~/Downloads/RTES Assignment 1$
```

Figure -3 ( Without RT Thread )



```
raj@raj-Inspiron-3543: ~/Downloads/RTES Assignment 1
raj@raj-Inspiron-3543:~/Downloads/RTES Assignment 1$ make
make: Nothing to be done for 'all'.
raj@raj-Inspiron-3543:~/Downloads/RTES Assignment 1$ ./posix_clock
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
ERROR: sched_setscheduler rc is -1
sched_setscheduler: Operation not permitted
raj@raj-Inspiron-3543:~/Downloads/RTES Assignment 1$ sudo ./posix_clock
[sudo] password for raj:
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO

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

RT clock start seconds = 1528624918, nanoseconds = 40920333
RT clock stop seconds = 1528624921, nanoseconds = 40948569
RT clock DT seconds = 3, nanoseconds = 28236
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 28236
raj@raj-Inspiron-3543:~/Downloads/RTES Assignment 1$
```

Figure – 4 ( With RT Thread )

### Interrupt latency :-

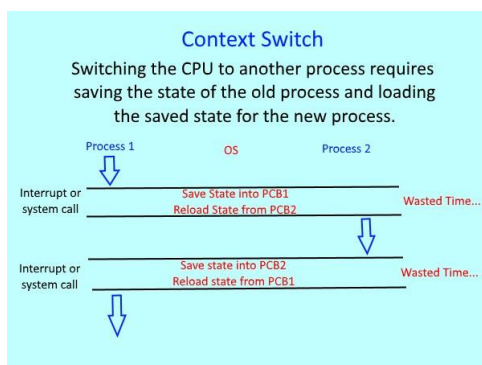
In computing, interrupt latency is the time that elapses from when an interrupt is generated to when the source of the interrupt is serviced. For many operating systems, devices are serviced as soon as the device's interrupt handler is executed.



**Low Interrupt Latency :-** It is a set of behaviours that reduce the interrupt latency for the processor, and is enabled by default. enables accesses to Normal memory, including multiword accesses and external accesses, to be abandoned part-way through execution so that the processor can react to a pending interrupt faster than would otherwise be the case. When an instruction is abandoned in this way, the processor behaves as if the instruction was not executed at all. All the real time operating systems claim regarding their low interrupt latency meaning they have faster servicing and low overhead, which is greatly useful for the **hard real time operating products like Military robot, Drone etc.**

#### **Lower Context Switch time :-**

In computing, a **context switch** is the process of storing the state of a process or of a thread, so that it can be restored and execution resumed from the same point later. Context switch overhead is less then switching is done very fast between different tasks. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking operating system.



**Figure – 5 Context Switch**

### **3) Stable timer services where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift**

When the timer interrupt occurs exactly where it is supposed to be, then it is called stable timer interrupt. When the system is running, many tasks are performed at the same time. It is very important that all the tasks which are sharing the same processor, occur at the definite time and thus timing of the tasks is very important.

**Timeout :-** An event that occurs at the end of a predetermined period of time that began at the occurrence of another specified event. They are responsible for the completion of the tasks. It makes sure that even if one task is not complete, all the others are completed.

Low Jitter is needed in the system or else timing would not be accurate. The OS performs tasks in the background keeping its own code on standby which results in jitter. It is of few milliseconds and this cannot be removed. RTOS provides jitter handling procedures to those who do not meet their deadlines.

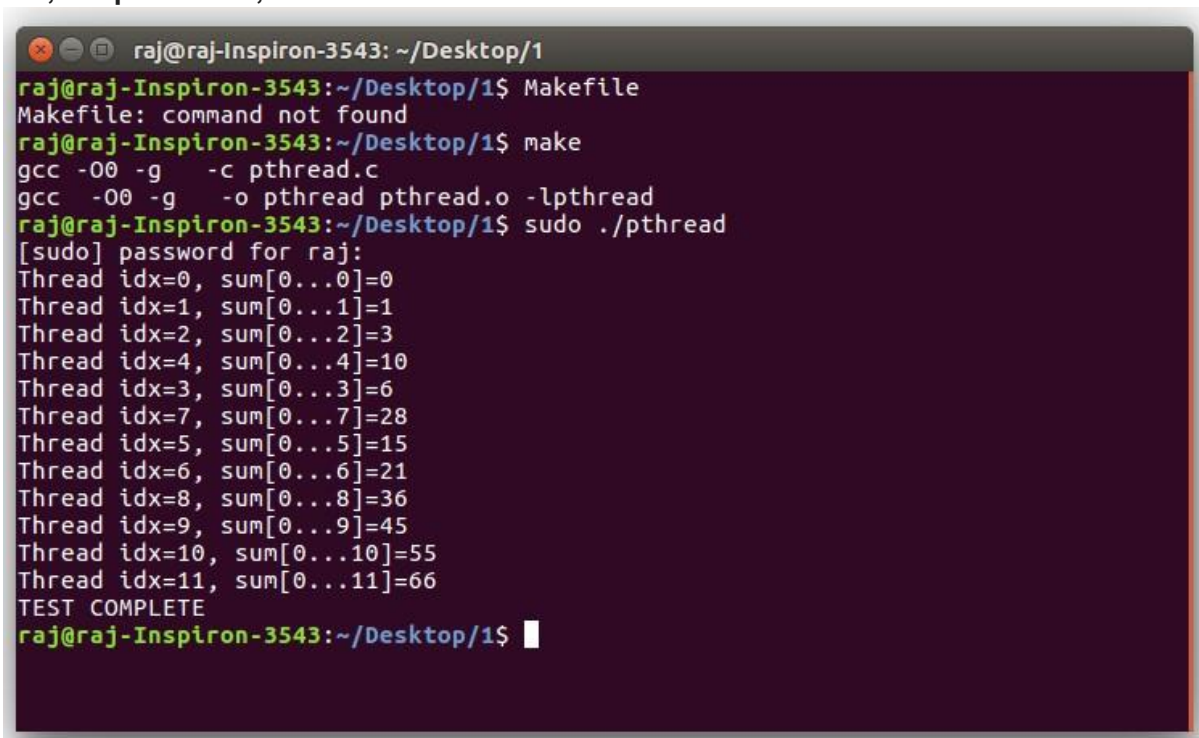
**Drift :-** It occurs when the timers move away from their desired place and disturb the precision. It is low which indicates that the deadline is very well handled in such RTOS systems.

Moreover, The RT clock provided by the function is **not accurate** enough because the requested delay is of 3 sec and the output I got is of 3 sec and 108742 nano sec. thus there is a delay of **108742 nanosec** which determines that the system is not real time.

I have taken another instance with the RT thread, where the time measurement is also not accurate because similar to the above statement, in that example also I am getting the output as 3 sec **and 28236 nano sec** which determines the system **not to be accurate**.

Answer 4 :-

For , Simple Thread,



```
raj@raj-Inspiron-3543: ~/Desktop/1
raj@raj-Inspiron-3543:~/Desktop/1$ Makefile
Makefile: command not found
raj@raj-Inspiron-3543:~/Desktop/1$ make
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
raj@raj-Inspiron-3543:~/Desktop/1$ sudo ./pthread
[sudo] password for raj:
Thread idx=0, sum[0...0]=0
Thread idx=1, sum[0...1]=1
Thread idx=2, sum[0...2]=3
Thread idx=4, sum[0...4]=10
Thread idx=3, sum[0...3]=6
Thread idx=7, sum[0...7]=28
Thread idx=5, sum[0...5]=15
Thread idx=6, sum[0...6]=21
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
raj@raj-Inspiron-3543:~/Desktop/1$
```

Figure -6 Simple Thread

In this example , pthread\_create API is used. It is used to pass certain parameters in the code like NUM\_THREADS and sum of all numbers till NUM\_THREADS are calculated.

Here, the function for count is used which counts and adds the numbers till the desired number. Initial value of sum is declared as 0 so all the summations will start from 0 to specific number As shown in the figure, for eg, sum[0...7] will mean addition of all the numbers from 0 to 7 (0+1+2+3+4+5+6+7) whose result will be 28.

Moreover, in the initial phase of the code, the value of NUM\_THREADS is declared as 12 and as a result we get total 12 results starting from 0 to 11.

This type of mechanism is used to pass function parameters to threads, so that by calling the thread we can pass some important data to the called function

For, Incdecthread,

```
raj@raj-Inspiron-3543:~/Desktop/2$ Makefile
Makefile: command not found
raj@raj-Inspiron-3543:~/Desktop/2$ make
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
raj@raj-Inspiron-3543:~/Desktop/2$ ./pthread.c
bash: ./pthread.c: Permission denied
raj@raj-Inspiron-3543:~/Desktop/2$ sudo ./pthread
[sudo] password for raj:
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
Decrement thread idx=1, gsum=-55
Decrement thread idx=1, gsum=-66
Decrement thread idx=1, gsum=-78
Decrement thread idx=1, gsum=-91
Decrement thread idx=1, gsum=-105
Decrement thread idx=1, gsum=-120
Decrement thread idx=1, gsum=-136
Decrement thread idx=1, gsum=-153
Decrement thread idx=1, gsum=-171
Decrement thread idx=1, gsum=-190
Decrement thread idx=1, gsum=-210
Decrement thread idx=1, gsum=-231
Decrement thread idx=1, gsum=-253
Decrement thread idx=1, gsum=-276
Decrement thread idx=1, gsum=-300
Decrement thread idx=1, gsum=-325
Decrement thread idx=1, gsum=-351
Decrement thread idx=1, gsum=-378
Decrement thread idx=1, gsum=-406
Decrement thread idx=1, gsum=-435
Decrement thread idx=1, gsum=-465
Decrement thread idx=1, gsum=-496
Decrement thread idx=1, gsum=-528
Decrement thread idx=1, gsum=-561
```

**Figure :7(1) Decrement thread**

```
Increment thread idx=0, gsum=-4674
Increment thread idx=0, gsum=-4620
Increment thread idx=0, gsum=-4565
Increment thread idx=0, gsum=-4509
Increment thread idx=0, gsum=-4452
Increment thread idx=0, gsum=-4394
Increment thread idx=0, gsum=-4335
Increment thread idx=0, gsum=-4275
Increment thread idx=0, gsum=-4214
Increment thread idx=0, gsum=-4152
Increment thread idx=0, gsum=-4089
Increment thread idx=0, gsum=-4025
Increment thread idx=0, gsum=-3960
Increment thread idx=0, gsum=-3894
Increment thread idx=0, gsum=-3827
Increment thread idx=0, gsum=-3759
Increment thread idx=0, gsum=-3690
Increment thread idx=0, gsum=-3620
Increment thread idx=0, gsum=-3549
Increment thread idx=0, gsum=-3477
Increment thread idx=0, gsum=-3404
Increment thread idx=0, gsum=-3330
Increment thread idx=0, gsum=-3255
Increment thread idx=0, gsum=-3179
Increment thread idx=0, gsum=-3102
Increment thread idx=0, gsum=-3024
Increment thread idx=0, gsum=-2945
Increment thread idx=0, gsum=-2865
Increment thread idx=0, gsum=-2784
Increment thread idx=0, gsum=-2702
Increment thread idx=0, gsum=-2619
Increment thread idx=0, gsum=-2535
Increment thread idx=0, gsum=-2450
Increment thread idx=0, gsum=-2364
Increment thread idx=0, gsum=-2277
Increment thread idx=0, gsum=-2189
Increment thread idx=0, gsum=-2100
Increment thread idx=0, gsum=-2010
Increment thread idx=0, gsum=-1919
Increment thread idx=0, gsum=-1827
Increment thread idx=0, gsum=-1734
Increment thread idx=0, gsum=-1640
Increment thread idx=0, gsum=-1545
```

**Figure : 7(2) Increment Thread**

There are 2 functions in this code. 1) decThread 2) incThread

They both calculate the sum of numbers till 1000 (COUNT Value) for incThread and from 1000 to 1 for decThread on every iteration. Every time it calculates the sum its value is stored in global sum which is mentioned as “gsum” in the code. In this code, we can see pthread\_create (pointer to thread descriptor), a POSIX thread with the parameters , pthread\_join (synchronisation mechanism).

Thus this code shows how threads are created , how they use pointers to functions.

For TestDigest.c ,

```

raj@raj-Inspiron-3543:~/Desktop/3$ ./pthread
Will default to 4 synthetic IO workers

***** MULTI THREAD TESTS
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
calling pthread_create
pthread create: Success
calling pthread_create
pthread create: Success
calling pthread_create
pthread create: Success
calling pthread_create
pthread create: Success

***** TOTAL PERFORMANCE SUMMARY
Thread 0 done in 0 microsecs for 100000 iterations
0.000000 MD5 digests computed per second
Thread 1 done in 0 microsecs for 100000 iterations
0.000000 MD5 digests computed per second
Thread 2 done in 0 microsecs for 100000 iterations
0.000000 MD5 digests computed per second
Thread 3 done in 0 microsecs for 100000 iterations
0.000000 MD5 digests computed per second

For 4 threads, Total rate=0.000000
raj@raj-Inspiron-3543:~/Desktop/3$ nano ./pthread.c

```

Figure :- 8(1)

```

raj@raj-Inspiron-3543:~/Desktop/3$ sudo ./testdigest
[sudo] password for raj:
Sorry, try again.
[sudo] password for raj:
Will default to 4 synthetic IO workers

SINGLE THREAD TESTS

MD5 Digest Test
Test Done in 140522 microsecs for 100000 iterations
711632.342267 MD5 digests computed per second
MD5 Digest=0x6f 0xb2 0x1d 0x9c 0x71 0xac 0x5b 0x7d 0x8a 0xaf 0x3a 0x99 0x2c 0x5a 0x75 0xb2

CRC Test
Test Done in 460084 microsecs for 100000 iterations
217351.614053 CRC32s computed per second
CRC=0xffffffff

SHA-1 Test
Test Done in 206894 microsecs for 100000 iterations
483339.294518 SHA-1 digests computed per second
SHA-1 Digest=0xb0 0xd3 0x60 0x24 0x75 0x42 0x5a 0x b 0x67 0x25 0x8c 0x62 0x54 0xf5 0x5a 0x 6 0xac 0x77 0xa8 0x4b

SHA-256 Test
Test Done in 407284 microsecs for 100000 iterations
245528.918396 SHA-256 digests computed per second
SHA-256 Digest=0x97 0x58 0x9f 0x89 0x1f 0xa6 0x92 0xbf 0xea 0x4c 0xc7 0x4c 0x 9 0x3f 0x19 0xec 0xfb 0x65 0x2a 0xe3 0xf1 0xca 0xe5 0xd9 0x49 0x1
d 0xec 0x99 0xb0 0x2b 0x7e 0xcc

***** MULTI THREAD TESTS
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
calling pthread_create
Thread 1 created
calling pthread_create
Thread 2 created
calling pthread_create
Thread 3 created
calling pthread_create
Thread 4 created
Thread 0 exit
Thread 3 exit

```

Figure:-8(2)



```

CRC=0xffffffff
SHA-1 Test
Test Done in 206894 microsecs for 100000 iterations
483339.294518 SHA-1 digests computed per second
SHA-1 Digest=0xb0 0xd3 0x60 0x24 0x75 0x42 0x5a 0x b 0x67 0x25 0x8c 0x62 0x54 0xf5 0x5a 0x 6 0xac 0x77 0xa8 0x4b

SHA-256 Test
Test Done in 407284 microsecs for 100000 iterations
245528.918396 SHA-256 digests computed per second
SHA-256 Digest=0x97 0x58 0x9f 0x89 0x1f 0xa6 0x92 0xbf 0xea 0x4c 0xc7 0x4c 0x 9 0x3f 0x19 0xec 0xfb 0x65 0x2a 0xe3 0xf1 0xca 0xe5 0xd9 0x49 0x1
d 0xec 0x99 0xb0 0x2b 0x7e 0xcc

***** MULTI THREAD TESTS
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
calling pthread_create
Thread 1 created
calling pthread_create
Thread 2 created
calling pthread_create
Thread 3 created
calling pthread_create
Thread 4 created
Thread 0 exit
Thread 3 exit
Thread 2 exit
Thread 1 exit

***** TOTAL PERFORMANCE SUMMARY
Thread 0 done in 188114 microsecs for 100000 iterations
531592.544946 MD5 digests computed per second
Thread 1 done in 190871 microsecs for 100000 iterations
523914.057138 MD5 digests computed per second
Thread 2 done in 190732 microsecs for 100000 iterations
524295.870646 MD5 digests computed per second
Thread 3 done in 185244 microsecs for 100000 iterations
539828.550452 MD5 digests computed per second

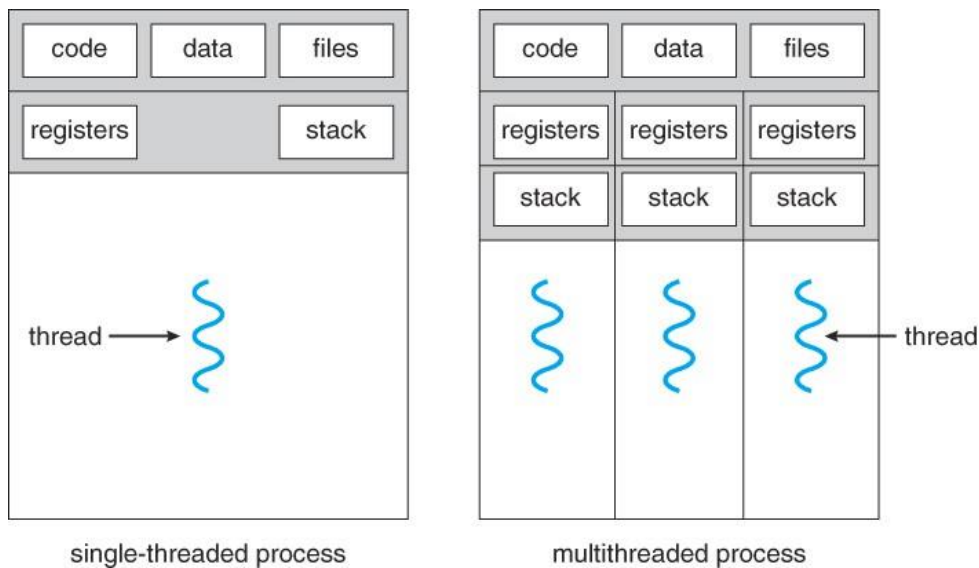
For 4 threads, Total rate=2119631.023182
raj@raj-Inspiron-3543:~/Desktop/3$

```

Figure :- 8(3)

### Description of key RTOS / Linux OS porting requirements:-

**Threading :-** A thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers. It is the smallest unit of processing that a scheduler works on. A thread of execution results from a fork of a computer program into two or more concurrently running tasks. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. Example of threads in same process is automatic spell check and automatic saving of a file while writing. Threads are basically processes that run in the same memory context. Threads may share the same data while execution.



**Figure :- 9 Simplethread vs multithread Process**

### Why it is used?

Suppose there is a process, that receiving real time inputs and corresponding to each input it has to produce a certain output. Now, if the process is not multi-threaded ie if the process does not involve multiple threads, then the whole processing in the process becomes synchronous. This means that the process takes an input processes it and produces an output.

### Problems in Threading :-

Threads share all the segments (except the stack segment) and can be pre-empted at any stage by the scheduler than any global variable or data structure that can be left in inconsistent state by pre-emption of one thread could cause severe problems when the next high priority thread executes the same function and uses the same variables or data structures.

### User thread and Kernel thread:-

A user space threads are created, controlled and destroyed using user space thread libraries. These threads are not known to kernel and hence kernel is nowhere involved in their processing. These threads follow co-operative multitasking where-in a thread releases CPU on its own wish i.e. the scheduler cannot pre-empt the thread. The advantages of user space threads is that the switching between two threads does not involve much overhead and is generally very fast while on the negative side since these threads follow co-operative multitasking so if one thread gets block the whole process gets blocked.

A kernel space thread is created, controlled and destroyed by the kernel. For every thread that exists in user space there is a corresponding kernel thread. Since these threads are managed by kernel so they follow preemptive multitasking where-in the scheduler can pre-empt a thread in execution with a higher priority thread which is ready for execution. The major advantage of kernel threads is that even if one of the thread gets blocked the whole process is not blocked as kernel threads follow preemptive scheduling while on the negative side the context switch is not very fast as compared to user space threads.

### Tasking :-

Task is a set of program instructions that are loaded in memory. Tasks are equivalent to threads in freeRTOS.

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        unsigned short usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask
                        );
```

Parameters:

- 1)**pvTaskCode** :- Pointer to the task entry function
- 2)**pcName** :-A descriptive name for the task. This is mainly used to facilitate debugging, but can also be used to obtain a task handle.
- 3)**usStackDepth**:-The number of words to allocate for use as the task's stack.
- 4)**pvParameters** :- A value that will be passed into the created task as the task's parameter.
- 5)**uxpriority** :- The priority at which the created task will execute.
- 6)**pxCreatedTask** :- Used to pass a handle to the created task out of the xTaskCreate() function. pxCreatedTask is optional and can be set to NULL.

### 2) Semaphores :-

Semaphores are a technique for coordinating or synchronizing activities in which multiple processes compete for the same operating system resources. A semaphore is a value in a designated place in operating system storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be (0 or 1) or can have additional values. The main object of semaphore is to protect a common memory.

Semaphores are commonly used for two purposes: to share a common memory space and to share access to files. Semaphores are one of the techniques for inter-process communication (IPC).

APIs for semaphore

**semget** :- Create a new semaphore

**semop** :-Acquire or release a semaphore



**semctl:-** Get info about a semaphore

**sem\_undo :-** Undo the semaphore operation if the process exits.

**sem\_close :-** close a named semaphore

**sem\_wait :-** locks the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

**sem\_sync :-** It is a OS semaphore created with initial value 0,so that initially any task which waits on it will block.

### **Synthetic workload generation:-**

In the code of Vxworks RTOS , they have used a macro called “ FIB\_TEST” in which 2 parameters are used 1)seqCnt 2)iterCnt

The synthetic workload is a set of parameterized synthetic or artificial programs which serve as the workload for a system under study. The parameterized nature of the programs allows the user to change their behavior to create different resource demands on the system. The SW is easy to use, flexible, and can be representative of a real-time workload. The SW consists of a driver and a set of synthetic tasks. The synthetic tasks are generated by a synthetic workload generator (SWG) from the user's specification written in SWSL, a synthetic workload specification language.

For the given example the value of of seqCnt is 47 and iterCnt is 710000

```
#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++;                   \
        }                            \
    }
```

**Overall Good description :-**

```
raj@raj-Inspiron-3543: ~/Downloads/RTES Assignment 1
raj@raj-Inspiron-3543:~/Downloads/RTES Assignment 1$ ./lab1
System has 4 processors configured and 4 available.
Using CPUS=1 from total available.
main_param: Operation not permitted
Pthread Policy is SCHED_OTHER
raj@raj-Inspiron-3543:~/Downloads/RTES Assignment 1$ sudo ./lab1
[sudo] password for raj:
System has 4 processors configured and 4 available.
Using CPUS=1 from total available.
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Service threads will run on 1 CPU cores
F10 runtime calibration 0.004262 msec per 100 test cycles, so 2346 required
F20 runtime calibration 0.004152 msec per 100 test cycles, so 4816 required
Start sequencer
Starting Sequencer: [S1, T1=20, C1=10], [S2, T2=50, C2=20], U=0.9, LCM=100

**** CI t=0.001012
F10 start 1 @ 0.058370 on core 3
F10 complete 1 @ 13.937995, 2346 loops
F20 start 1 @ 13.945593 on core 3
t=20.055572
```

Figure :- 10(1)

```
raj@raj-Inspiron-3543: ~/Downloads/RTES Assignment 1
Start sequencer
Starting Sequencer: [S1, T1=20, C1=10], [S2, T2=50, C2=20], U=0.9, LCM=100

**** CI t=0.001012
F10 start 1 @ 0.058370 on core 3
F10 complete 1 @ 13.937995, 2346 loops
F20 start 1 @ 13.945593 on core 3
t=20.055572
F10 start 2 @ 20.083809 on core 3
F10 complete 2 @ 26.717067, 2346 loops
F20 complete 1 @ 35.086639, 4816 loops
t=40.159157
F10 start 3 @ 40.186356 on core 3
F10 complete 3 @ 48.631781, 2346 loops
t=50.272857
F20 start 2 @ 50.305183 on core 3
t=60.301386
F10 start 4 @ 60.307994 on core 3
F10 complete 4 @ 67.279623, 2346 loops
F20 complete 2 @ 71.207135, 4816 loops
t=80.351465
F10 start 5 @ 80.373142 on core 3
F10 complete 5 @ 91.237167, 2346 loops
```

Figure:- 10 (2)

```
raj@raj-Inspiron-3543: ~/Downloads/RTES Assignment 1
F10 start 5 @ 80.373142 on core 3
F10 complete 5 @ 91.237167, 2346 loops

**** CI t=100.400464
F10 start 6 @ 100.441100 on core 3
F10 complete 6 @ 109.750002, 2346 loops
F20 start 3 @ 109.763611 on core 3
t=120.440821
F10 start 7 @ 120.447976 on core 3
F10 complete 7 @ 127.047860, 2346 loops
F20 complete 3 @ 130.258413, 4816 loops
t=140.537869
F10 start 8 @ 140.566685 on core 3
F10 complete 8 @ 149.015016, 2346 loops
t=150.654095
F20 start 4 @ 150.682479 on core 3
t=160.680076
F10 start 9 @ 160.686032 on core 3
F10 complete 9 @ 167.823052, 2346 loops
F20 complete 4 @ 171.584494, 4816 loops
t=180.729551
F10 start 10 @ 180.756123 on core 3
F10 complete 10 @ 191.992452, 2346 loops
```

Figure :- 10 (3)

```
raj@raj-Inspiron-3543: ~/Downloads/RTES Assignment 1
F10 complete 10 @ 191.992452, 2346 loops

**** CI t=200.838982
F10 start 11 @ 200.870243 on core 3
F10 complete 11 @ 208.490653, 2346 loops
F20 start 5 @ 208.501842 on core 3
t=220.869111
F10 start 12 @ 220.874953 on core 3
F10 complete 12 @ 227.906308, 2346 loops
F20 complete 5 @ 232.094828, 4816 loops
t=240.962748
F10 start 13 @ 240.990503 on core 3
t=250.992530
F10 complete 13 @ 251.796966, 2346 loops
F20 start 6 @ 251.805072 on core 3
t=261.017925
F10 start 14 @ 261.036802 on core 3
F10 complete 14 @ 267.785977, 2346 loops
F20 complete 6 @ 273.424732, 4816 loops
t=281.130392
F10 start 15 @ 281.159088 on core 3
F10 complete 15 @ 288.859291, 2346 loops

TEST COMPLETE
```

Figure :- 10 (4)

```
sem_post(&semF10); sem_post(&semF20);

case -1  usleep(20*USEC_PER_MSEC); sem_post(&semF10);
        printf("t=%lf\n", event_time=getTimeMsec() - start_time);

case -2  usleep(20*USEC_PER_MSEC); sem_post(&semF10);
        printf("t=%lf\n", event_time=getTimeMsec() - start_time);

case -3  usleep(10*USEC_PER_MSEC); sem_post(&semF20);
        printf("t=%lf\n", event_time=getTimeMsec() - start_time);

case -4  usleep(10*USEC_PER_MSEC); sem_post(&semF10);
        printf("t=%lf\n", event_time=getTimeMsec() - start_time);

case -5  usleep(20*USEC_PER_MSEC); sem_post(&semF10);
        printf("t=%lf\n", event_time=getTimeMsec() - start_time);

case -6  usleep(20*USEC_PER_MSEC);
```

#### **case -1**

Initially, sem10 and sem20 is posted. Then the cyclic scheduler sleeps for 20 ms  
After which the semaphore10 is posted. Since sem10 is at a higher value than sem20, it will pre-empt sem20.

#### **case-2**

For this case, sem10 and sem20 is posted. Then the cyclic scheduler sleeps for 20 ms  
After which the semaphore10 is posted. Since sem10 is at a higher value than sem20, it will pre-empt sem20.

#### **case -3**

For this case, sem10 and sem20 is posted. Then the cyclic scheduler sleeps for 10 ms.  
After which the semaphore20 is posted. Since sem20 is at a higher value than sem10, it will pre-empt sem10.

#### **case -4**

For this case, sem10 and sem20 is posted. Then the cyclic scheduler sleeps for 10 ms. After which the semaphore10 is posted. Since sem10 is at a higher value than sem20, it will pre-empt sem20.

#### **case - 5**

For this case, sem10 and sem20 is posted. Then the cyclic scheduler sleeps for 20 ms. After which the semaphore10 is posted. Since sem10 is at a higher value than sem20, it will pre-empt sem20.

Now the cyclic scheduler sleeps for 20 ms.

After this the whole procedure happens again.

This is the explanation of Professor Siewert's Code which I have found the best solution for the given time constraints.

References :- [https://en.wikipedia.org/wiki/Rate-monotonic\\_scheduling](https://en.wikipedia.org/wiki/Rate-monotonic_scheduling)

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0363g/BEIDDFBB.htm>

[| https://en.wikipedia.org/wiki/Interrupt\\_latency](https://en.wikipedia.org/wiki/Interrupt_latency)

[https://en.wikipedia.org/wiki/Context\\_switch](https://en.wikipedia.org/wiki/Context_switch)

<https://www.hq.nasa.gov/alsj/a11/a11.1201-pa.html>

[https://www.ece.cmu.edu/~ee349/lectures/17\\_RTOS\\_AdvancedRealtime.pdf](https://www.ece.cmu.edu/~ee349/lectures/17_RTOS_AdvancedRealtime.pdf)

[https://stackoverflow.com/questions/3042717/what-is-the-difference-between-a-threadprocess-](https://stackoverflow.com/questions/3042717/what-is-the-difference-between-a-threadprocess-task?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa)

[task?utm\\_medium=organic&utm\\_source=google\\_rich\\_qa&utm\\_campaign=google\\_rich\\_qa](https://stackoverflow.com/questions/3042717/what-is-the-difference-between-a-threadprocess-task?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa)

[https://www.thegeekstuff.com/2012/03/linux-threads-intro/?utm\\_source=tuicool](https://www.thegeekstuff.com/2012/03/linux-threads-intro/?utm_source=tuicool)

<https://www.freertos.org/a00125.html> <https://whatis.techtarget.com/definition/semaphore>

<https://flylib.com/books/en/1.410.1.110/1/>

[https://www.systutorials.com/docs/linux/man/3-sem\\_wait/](https://www.systutorials.com/docs/linux/man/3-sem_wait/)

<https://www.semanticscholar.org/paper/A-synthetic-workload-for-a-distributed-real-timeKiskis-Shin/dc586477b50226951e0d96a7da3a70c81d28a2e6>