ECEN5623, Real-Time Embedded Systems:

Exercise #1 – Invariant LCM Schedules

Members: Nikhil Divekar, Anay Gondhalekar

Exercise #1 Requirements:

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 S_1, S_2, and S_3 with T_1=3, C_1=1, T_2=5, C_2=2, T_3=15, C_3=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?

Rate Monotonic Scheduling:

According to the Wikipedia, Rate Monotonic Scheduling (RMS) is a priority algorithm used in real-time operating system (RTOS) with 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. This operating system is preemptive and are deterministic in nature. Context switching time is considered to be negligible in this problem. Rate monotonic analysis is used in conjunction with those systems to provide scheduling guarantees for a particular application.

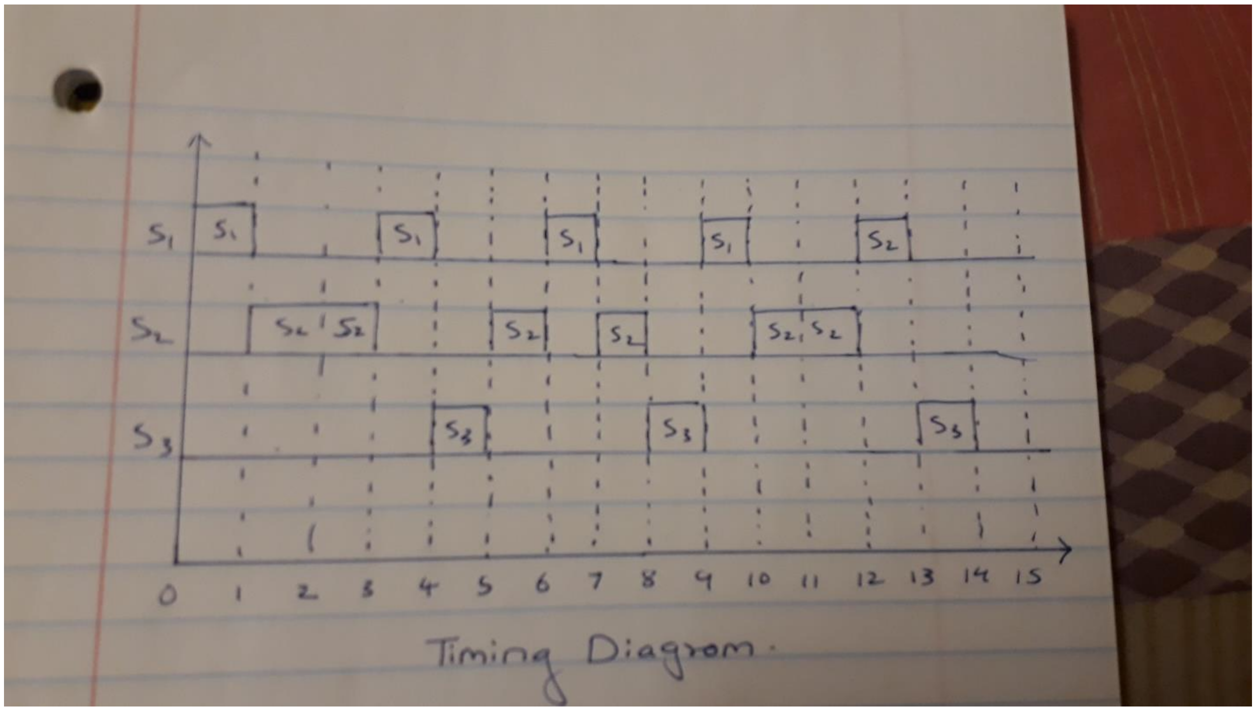
We have been given the following data:

S1: T1 = 3, C1 = 1

S2: T2 = 5, C2 = 2

S3: T3 = 15, C3 = 3

Below is the timing diagram according the RM fixed policy:



CPU Utilization:

In 1973, Liu and Layland proved that for a set of *n* periodic tasks with unique periods, a feasible schedule that will always meet deadlines exists if the CPU utilization is below a specific bound (depending on the number of tasks). The schedulability test for RMS is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

From the above equation, we get that,

CPU Utilization =

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

= C1/T1 + C2/T2 + C3/T3

= (1/3) + (2/5) + (3/15)

= (11/15) + (3/15)

= 14/15
= 93.33 %

Feasibility and Safety issues articulated:

The schedulability test for RMS for safety and feasibility is given by the following equation:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

We know that U = 93.33 %
Here n is the number of services.
Hence RHS of above equation equates to around 77 %.

When the number of services tends to infinity the above equation changes to:

$$\lim_{n\to\infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147\ldots$$

Thus, a rough estimate is that RMS can meet all of the deadlines if CPU utilization is less than 69.32%. But CPU utilization is 93.33 % which is greater.

Hence, we can conclude that the above process scheduling is feasible as shown in timing diagram since it is mathematically repeatable. However, since CPU utilization is much higher than safe limit of 69.32% hence this process is not safe. Thus, we can conclude that process is feasible but not safe.

Utility and Method description:

Method for timing diagram plot is as given below:

1. According to the question, Highest priority is assigned to the most frequently requested service and lowest priority is assigned to the least frequently requested. Hence, priority of services from highest to lowest is S1 > S2 > S3.

2. Next, find the least common multiple (LCM) of all the time periods, i.e. LCM of 3, 5 and 15 which is 15.

3. According to time period mentioned, S1 will occur once every 3 ms, S2 occurs 2 times every 5 ms and S3 will occur 3 times every 15 ms.

4. Considering above details, we draw the timing diagram using the frequency of services and their priorities.

5. From timing diagram, we can deduct CPU utilization and also Idle time. Further we can use CPU utilization to schedulability test to check feasibility and safety of the system.

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?

Apollo 11 reading and summary:

The reading describes the Apollo 11 Lunar lander overload story and as reported in RTECS event. The article starts with background of people of interest. There were 36864 15-bit word of fixed memory also called as ROM and 2048 words of RAM. Most of the executable code with constants and other data was present in fixed memory whereas erasable memory was used mostly for variables. There was constraint on erasable memory size and hence author and team were forced to use same memory address for different purposes. Thus there might be inconsistency and swap in storage of various data. Some memory locations were even used for as high as 7 purposes. But it was important that one memory location is used for single purpose at any given time.

The real-time operating system developed by team contained interrupt-driven and tim-dependent tasks. 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. So, If a job required more temporary storage, the scheduling request asked for a VAC - vector accumulator. VAC had 44 erasable words. There were seven core sets and five VAC areas.

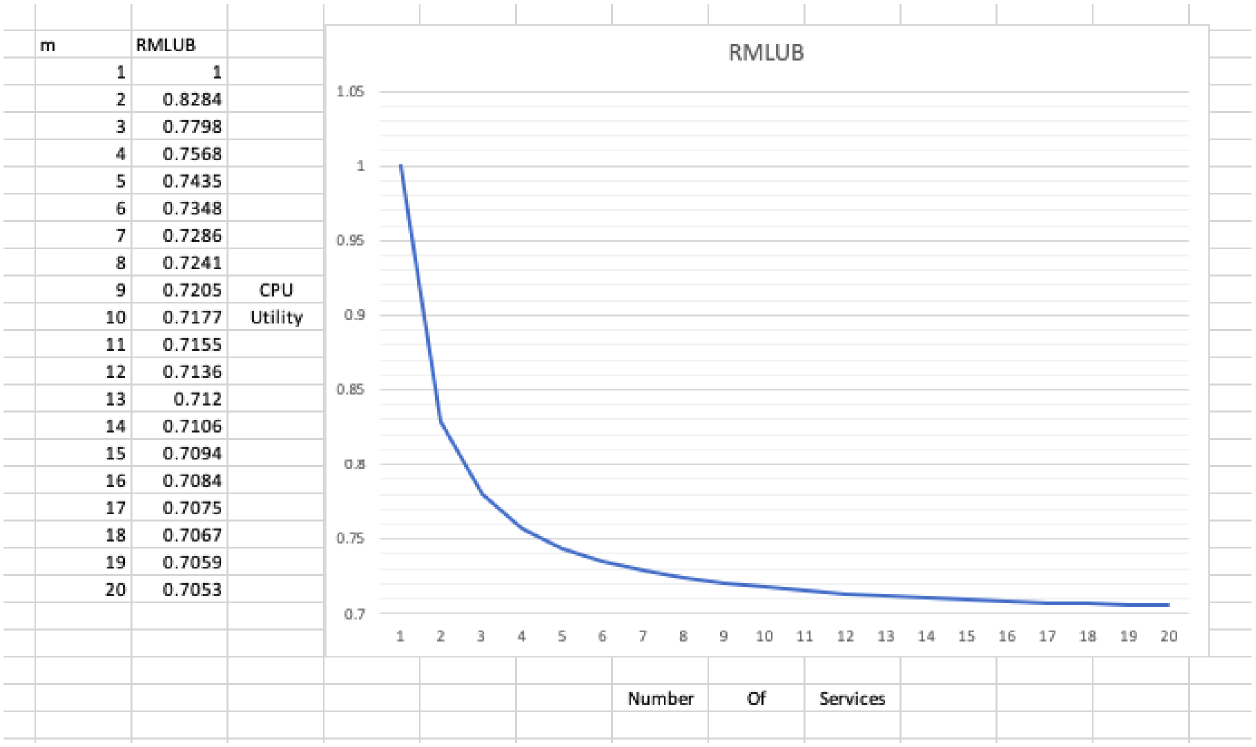
If the job to be scheduled required a VAC area, the operating system would scan the five VAC areas to find one which was available. if there were no VAC areas available, the program would branch to the Alarm/Abort routine and set Alarm 1201. Similarly, if no core sets were available, the program would branch to Alarm/Abort and set Alarm 1202.

In case of Apollo 11, there were repeated jobs to process rendezvous radar data which was not there but were scheduled because a misconfiguration of the radar switches. Thus, the core sets got filled up and a 1202 alarm was generated. The 1201 that came later in the landing was because the scheduling request that caused the actual overflow was one that had requested a VAC area. This leads to missing deadlines and hence violated rate monotonic policy. However, Apollo 11 was programmed in such a way that the computer rebooted on generation of 1201 or 1202 alarm. It restarted all the important stuff but flushed about erroneously scheduled jobs, so landing was done successfully. Rate monotonic policy states that the service with highest frequency has higher priority.

Root cause analysis and description:

The repeated jobs to process rendezvous radar data (that of course were not really there) were scheduled because a misconfiguration of the radar switches. Thus, the core sets got filled up and a 1202 alarm was generated. The 1201 that came later in the landing was because the scheduling request that caused the actual overflow was one that had requested a VAC area. Due to these generated alarms, there was unusual load on processor and thus utilization increased to a larger extent. This, in turn, led to less space available for many processes required for landing. Hence, as the jobs increased, processor asked for memory storage from VAC and eventually all core sets got filled up, ultimately leading to 1202 alarm. Later, overflow of jobs requesting VAC area lead to 1201 alarm.

RM LUB plot and description of margin:



m	RMLUB	
1	1	
2	0.8284	
3	0.7798	
4	0.7568	
5	0.7435	
6	0.7348	
7	0.7286	
8	0.7241	
9	0.7205	CPU
10	0.7177	Utility
11	0.7155	
12	0.7136	
13	0.712	
14	0.7106	
15	0.7094	
16	0.7084	
17	0.7075	
18	0.7067	
19	0.7059	
20	0.7053	

Key assumptions:

1.The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.

2. Deadlines consist of run-ability constraints only--i.e, each task must be completed before the next request for it occurs.

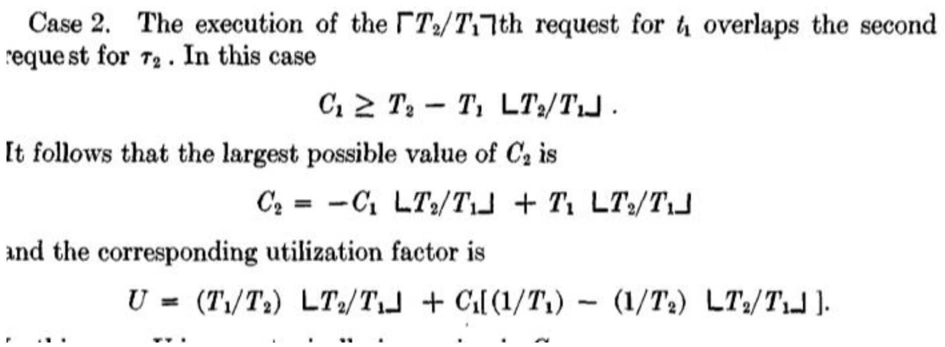
3. The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks

4. Run-time for each task is constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

5. Any nonperiodic 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.

Aspects of their derivation that I did not understand:

1. I did not understand this case:


Case 2. The execution of the $\lceil T_2/T_1 \rceil$th request for t_1 overlaps the second request for τ_2. In this case

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

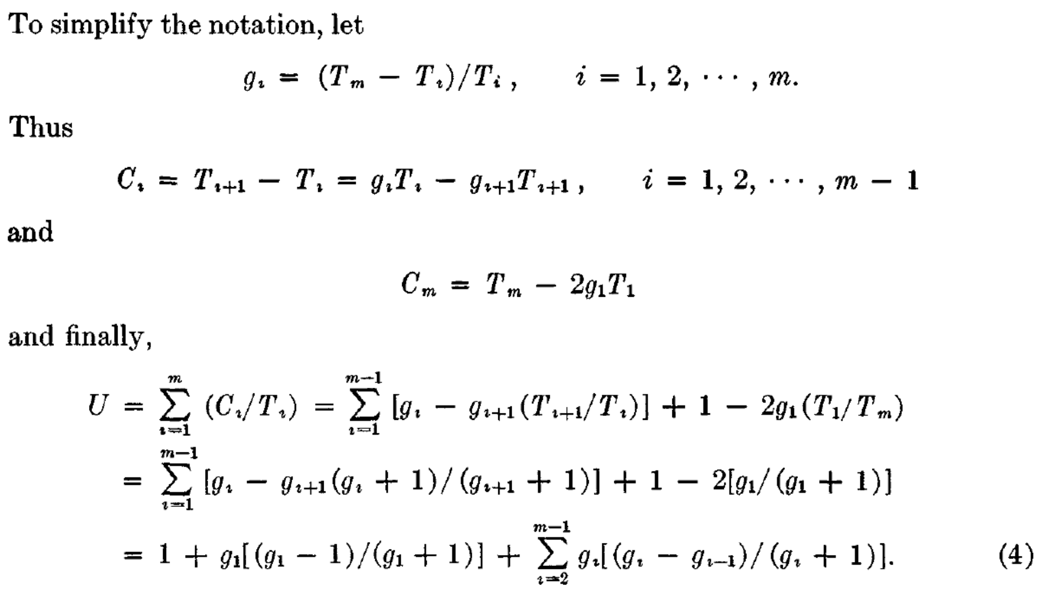
It follows that the largest possible value of C_2 is

$$C_2 = -C_1 \lfloor T_2/T_1 \rfloor + T_1 \lfloor T_2/T_1 \rfloor$$

and the corresponding utilization factor is

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

2. I did not understand simplification and final equation of Theorem 4:


To simplify the notation, let

$$g_i = (T_m - T_i)/T_i , \qquad i = 1, 2, \cdots , m.$$

Thus

$$C_i = T_{i+1} - T_i = g_i T_i - g_{i+1} T_{i+1} , \qquad i = 1, 2, \cdots , m - 1$$

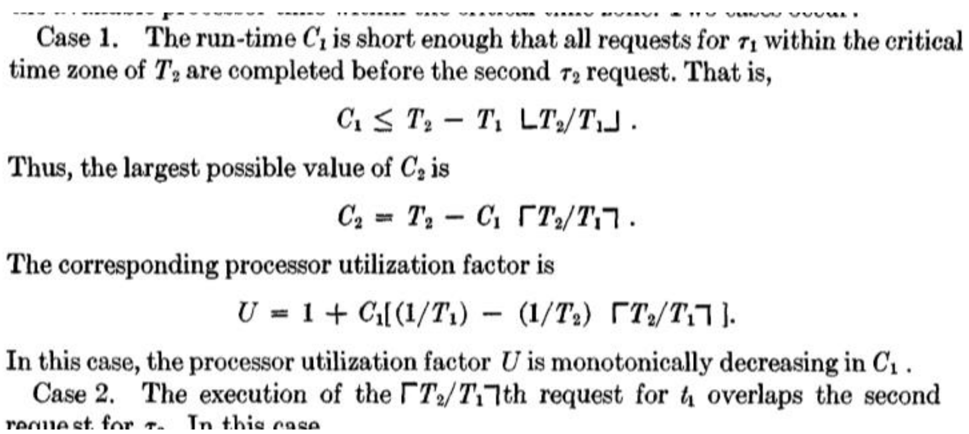
and

$$C_m = T_m - 2g_1 T_1$$

and finally,

$$\begin{aligned} U &= \sum_{i=1}^{m} (C_i/T_i) = \sum_{i=1}^{m-1} [g_i - g_{i+1}(T_{i+1}/T_i)] + 1 - 2g_1(T_1/T_m) \\ &= \sum_{i=1}^{m-1} [g_i - g_{i+1}(g_i + 1)/(g_{i+1} + 1)] + 1 - 2[g_1/(g_1 + 1)] \\ &= 1 + g_1[(g_1 - 1)/(g_1 + 1)] + \sum_{i=2}^{m-1} g_i[(g_i - g_{i-1})/(g_i + 1)]. \end{aligned} \qquad (4)$$

3. Why is this assumption made in Case 1?


Case 1. The run-time C_1 is short enough that all requests for τ_1 within the critical time zone of T_2 are completed before the second τ_2 request. That is,

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

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

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

In this case, the processor utilization factor U is monotonically decreasing in C_1.

Case 2. The execution of the $\lceil T_2/T_1 \rceil$th request for t_1 overlaps the second request for τ_2. In this case

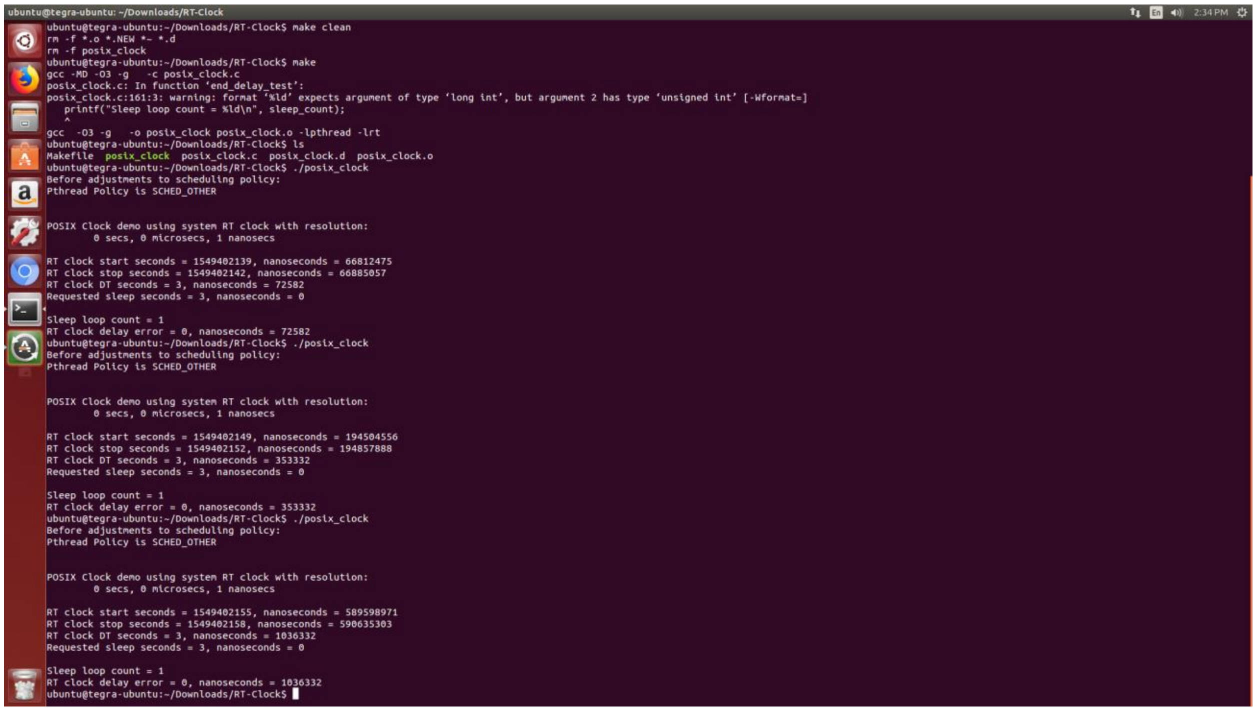
Arguments for and against RM policy and analysis prevention of Apollo 11 overload scenario:

No, I don’t think so RMA would have prevented the Apollo 11 errors. According to RMA policy, higher priority is assigned to the services happening more frequently. So with this, if there would have been frequent rendezvous data task and hence is at highest frequency. So with RMA this would have just led to overload of the core sets and would have eventually led to trigger of alarms.

RMA did not prevent the errors and potential abort since processor refreshed every time the alarm was generated, hence this led to flushing out all the unimportant stuff and hence the useful data was not lost. With RMA, all these data would have used all the core sets and would overload the processor.

3) [20 points] Download http://mercury.pr.erau.edu/~siewerts/cec450/code/RT-Clock/ 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?

Code build run and output:



```
ubuntu@tegra-ubuntu: ~/Downloads/RT-Clock                                2:34 PM
ubuntu@tegra-ubuntu:~/Downloads/RT-Clock$ make clean
rm -f *.o *.NEW *~ *.d
rm -f posix_clock
ubuntu@tegra-ubuntu:~/Downloads/RT-Clock$ make
gcc -MD -O3 -g   -c posix_clock.c
posix_clock.c: In function ‘end_delay_test’:
posix_clock.c:161:3: 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
ubuntu@tegra-ubuntu:~/Downloads/RT-Clock$ ls
Makefile  posix_clock  posix_clock.c  posix_clock.d  posix_clock.o
ubuntu@tegra-ubuntu:~/Downloads/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 = 1549402139, nanoseconds = 66812475
RT clock stop seconds = 1549402142, nanoseconds = 66885057
RT clock DT seconds = 3, nanoseconds = 72582
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 72582
ubuntu@tegra-ubuntu:~/Downloads/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 = 1549402149, nanoseconds = 194504556
RT clock stop seconds = 1549402152, nanoseconds = 194857888
RT clock DT seconds = 3, nanoseconds = 353332
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 353332
ubuntu@tegra-ubuntu:~/Downloads/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 = 1549402155, nanoseconds = 589598971
RT clock stop seconds = 1549402158, nanoseconds = 590635303
RT clock DT seconds = 3, nanoseconds = 1036332
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 1036332
ubuntu@tegra-ubuntu:~/Downloads/RT-Clock$
```

Description of code:

I ran the code on NVIDIA Jetson TK1 platform. First, print_scheduler() function is called which determines which scheduling policy is pthread affiliated with. The attributes of pthread is set and scheduling policy is set to SCHED_FIFO. Also, priority of thread is set to maximum. Thread is created using pthread_create() API and delay_test is the callback function. In this function, we measure the time taken by processor to execute a particular function by using clock_gettime(CLOCK_REALTIME, &start_time). The clock ID is set to the CLOCK_REALTIME. The given clock has the resolution of 1 nanoseconds. Further, clock is provided sleep of 3 seconds and then time is measured whether it correctly aligns with expected time of 3 seconds. We start the clock before the sleep and then stop the clock after the sleep is done. We also use delta_t function to get the time difference between the clock start and stop time. I ran the code for 3 instances and got the different output in each case which are 72582 ns, 353332 ns, 1036332 ns.

What is value of each RTOS bragging point?

1.Low Interrupt handler latency:

Interrupt latency is defined as the time difference between interrupt is generated and source interrupt is serviced. We need interrupt latency as low as possible in RTOS since we should not miss the deadline. The program must switch from the current execution to ISR as soon as possible with minimum overhead possible. If the interrupt latency is too high, that may eventually lead to deadline miss which is undesirable. Hence, Interrupt latency for RTOS should be smaller than general purpose system. CPU does some important work of storing the values on stack registers and return address which constitutes the interrupt latency.

2. Low Context switch time:

Context switch time refers to the process of storing or restoring program context and some important data as registers values, etc. Context switch happens when interrupt occurs or lower priority task is preempted by high priority task. Also, the return address is stored so that program can resume when it returns from ISR or higher priority task. RTOS has faster context switching time as compared to general-purpose Linux system.

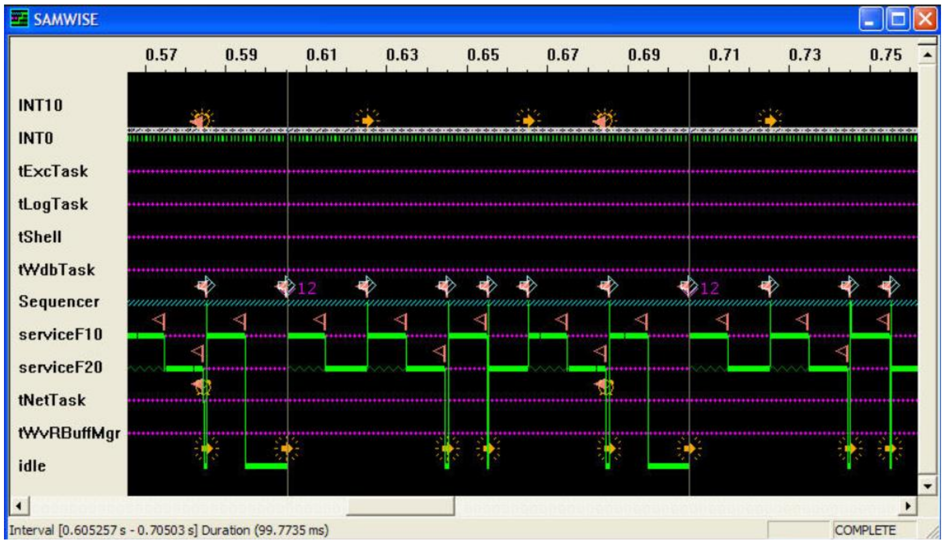
3. Stable timer services where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift:

Having stable interval timer interrupt is very important for RTOS. Stable time interrupt refers to interrupt occurs precisely even though when there is high load on CPU. Timeouts are more stable with RTOS. Since RTOS is more deterministic in nature, it has less amount of jitter and drift.

Determination and argument for or against accuracy of RT clock on system tested:

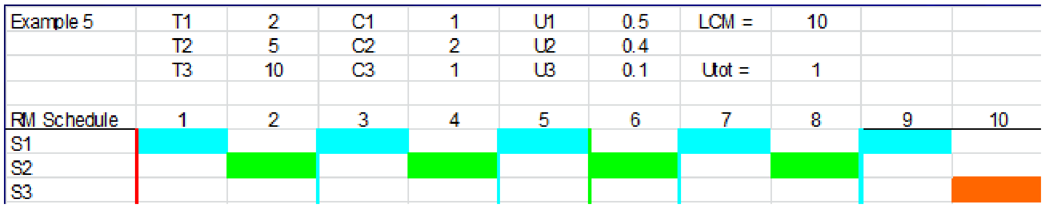
The RT clock is not very accurate. I did run the code 3 times on Jetson and got different values each time. Each time the time had some amount of deviation from expected output of 3 seconds. These deviations are majorly attributed to the context switching time, CPU Idle time and time to switch to kernel for some system calls which get invoked due to library functions. If this delay is very large than 3 seconds, then deviation would be much higher and might be a leading factor to miss time-critical deadlines. Hence, more accurate RT clock should be used with RTOS.

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:



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

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

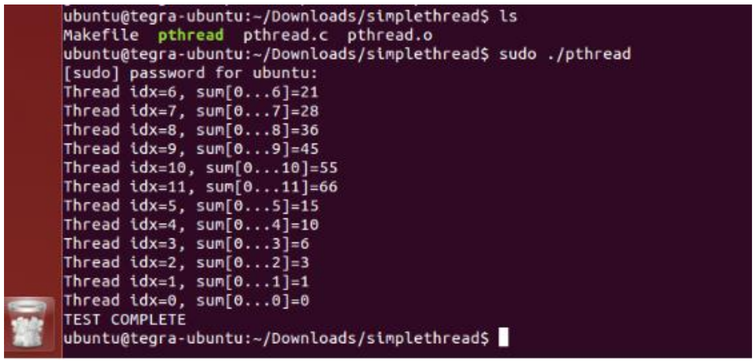


Example 5	T1	2	C1	1	U1	0.5	LCM =	10		
	T2	5	C2	2	U2	0.4				
	T3	10	C3	1	U3	0.1	Utot =	1		
RM Schedule	1	2	3	4	5	6	7	8	9	10
S1	🟦		🟦		🟦		🟦		🟦	
S2		🟩		🟩		🟩		🟩		
S3										🟧

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 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.

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 and on Canvas, but if you use it, be sure to cite it and make sure you understand it and can describe it well. If you use this resource, note how similar or dissimilar it is to the original VxWorks code and how predictable it is by comparison.

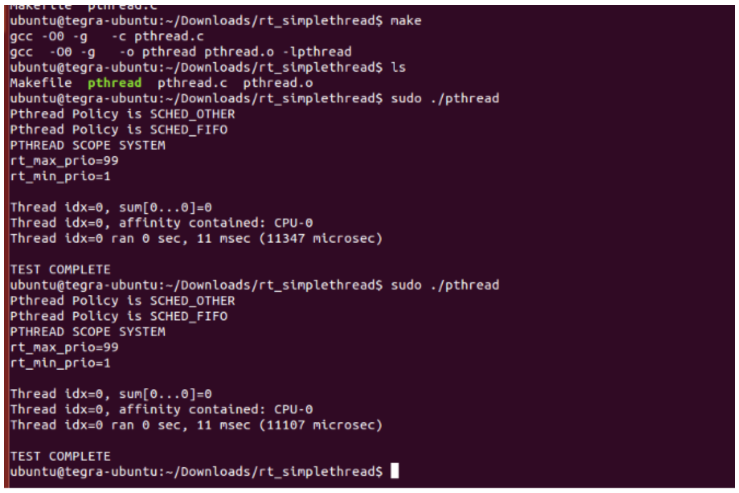
A. Simple thread:



```
ubuntu@tegra-ubuntu:~/Downloads/simplethread$ ls
Makefile  pthread  pthread.c  pthread.o
ubuntu@tegra-ubuntu:~/Downloads/simplethread$ sudo ./pthread
[sudo] password for ubuntu:
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
Thread idx=5, sum[0...5]=15
Thread idx=4, sum[0...4]=10
Thread idx=3, sum[0...3]=6
Thread idx=2, sum[0...2]=3
Thread idx=1, sum[0...1]=1
Thread idx=0, sum[0...0]=0
TEST COMPLETE
ubuntu@tegra-ubuntu:~/Downloads/simplethread$ 
```

This program creates 12 different threads and counterThread() function is called. Core affinity is not assigned

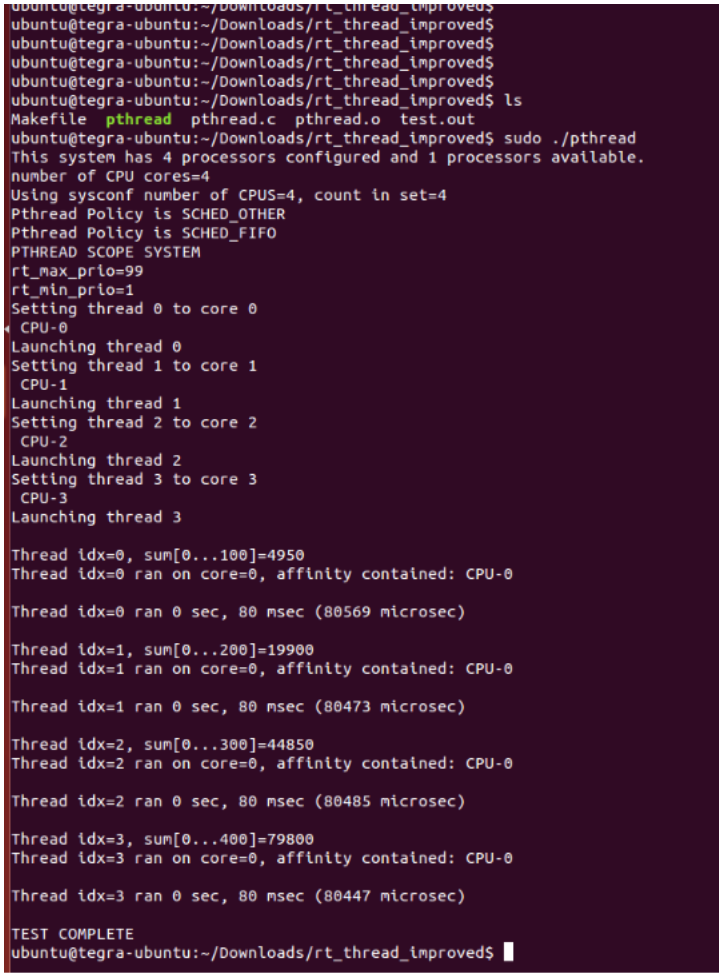
B. Rt_simple_thread:



```
ubuntu@tegra-ubuntu:~/Downloads/rt_simplethread$ make
gcc -O0 -g   -c pthread.c
gcc  -O0 -g   -o pthread pthread.o -lpthread
ubuntu@tegra-ubuntu:~/Downloads/rt_simplethread$ ls
Makefile  pthread  pthread.c  pthread.o
ubuntu@tegra-ubuntu:~/Downloads/rt_simplethread$ sudo ./pthread
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
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, 11 msec (11347 microsec)

TEST COMPLETE
ubuntu@tegra-ubuntu:~/Downloads/rt_simplethread$ sudo ./pthread
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
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, 11 msec (11107 microsec)

TEST COMPLETE
ubuntu@tegra-ubuntu:~/Downloads/rt_simplethread$ 
```

The difference between this and simple_thread is that core affinity is set here, hence all the codes are executed on single core.

C. Rt_thread_improved:



```
ubuntu@tegra-ubuntu:~/Downloads/rt_thread_improved$
ubuntu@tegra-ubuntu:~/Downloads/rt_thread_improved$
ubuntu@tegra-ubuntu:~/Downloads/rt_thread_improved$
ubuntu@tegra-ubuntu:~/Downloads/rt_thread_improved$
ubuntu@tegra-ubuntu:~/Downloads/rt_thread_improved$
ubuntu@tegra-ubuntu:~/Downloads/rt_thread_improved$ ls
Makefile  pthread  pthread.c  pthread.o  test.out
ubuntu@tegra-ubuntu:~/Downloads/rt_thread_improved$ sudo ./pthread
This system has 4 processors configured and 1 processors available.
number of CPU cores=4
Using sysconf number of CPUS=4, count in set=4
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
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=0, sum[0...100]=4950
Thread idx=0 ran on core=0, affinity contained: CPU-0

Thread idx=0 ran 0 sec, 80 msec (80569 microsec)

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

Thread idx=1 ran 0 sec, 80 msec (80473 microsec)

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

Thread idx=2 ran 0 sec, 80 msec (80485 microsec)

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

Thread idx=3 ran 0 sec, 80 msec (80447 microsec)

TEST COMPLETE
ubuntu@tegra-ubuntu:~/Downloads/rt_thread_improved$ 
```

In this case, 4 pthreads are created and each thread is et to the different CPUs. Also time stamp has been recorded for how much thread runs.

Description of Key RTOS / Linux OS porting requirements

1. Threading vs. tasking

Threading:

A thread is the smallest unit of processing that a scheduler understands. Generally in Linux it treats the process and thread as the same and doesn't understand any difference between it. But a given process can have multiple threads that can run synchronously or asynchronously. This asynchronous execution brings in the capability of each thread handling a particular work or service independently. Hence multiple threads running in a process handle their services which overall constitutes the complete capability of the process.

[Reference: https://www.thegeekstuff.com/2012/03/linux-threads-intro/]

The API for Pthread create is:

include <pthread.h>

int pthread_create(pthread_t ***thread*****, const pthread_attr_t** ***attr*****, void *(***start_routine*****) (void *), void** ***arg*****);**

Compile and link with *-pthread*.

Where the parameters are:

*thread : pointer to the pthread object

*attr : pointer to the the attributes we pass to the thread

*start_routine : pointer to the function for the thread

*arg : arguments that are passed from the calling function to the thread

[Reference: https://linux.die.net/man/3/pthread_create]

Tasking:

A real time application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context with no coincidental dependency on other tasks within the system or the RTOS scheduler itself. Only one task within the application can be executing at any point in time and the real time RTOS scheduler is responsible for deciding which task this should be. The RTOS scheduler may therefore repeatedly start and stop each task (swap each task in and out) as the application executes. As a task has no knowledge of the RTOS scheduler activity it is the responsibility of the real time RTOS scheduler to ensure that the processor context (register values, stack contents, etc) when a task is swapped in is exactly that as when the same task was swapped out. To achieve this each task is provided with its own stack. When the task is swapped out the execution context is saved to the stack of that task so it can also be exactly restored when the same task is later swapped back in.

[Reference: https://www.freertos.org/taskandcr.html]

The API for Task Create in FreeRTOS is :

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

Where the parameters are:

pvTaskCode : pointer to the task entry function

pcName : String associated with naming the task

usStackDepth : The stack that is to be assigned to a task

*pvParameters : Parameters the calling function can pass to the task

uxPriority : Priority to be assigned to the task

*pxCreatedTask : The pointer to the task handler

[Reference: https://www.freertos.org/a00125.html]

2. Semaphores,wait and sync:

Semaphore is a thread protection and safety mechanism to allow shared resources without race conditions. It basically uses a structure and queue to handle protection for a critical section of code, containing count, lock and wait queues. Semaphore can be of 2 types: Counting and Binary. Counting semaphore is where multiple threads can access a shared resource at the same time.It is generally used for read only scenarios. Binary semaphore is where only 1 thread can use the semaphore, so as the key is taken or not.

It operates as

Initialization– sem_init ( )

Increment/unlock– sem_post ( )

Decrement/lock – sem_wait ( )

Delete it– sem_destroy ( )

Semaphore is used in both task synchronization and managing shared resourses efficiently. The concept of semaphore is again same for rtos as it is for linux the only difference is the APIs.

Create -vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore)

Acquire -xSemaphoreTake(xSemaphoreHandle xSemaphore, portTickType xBlockTime)

Release - xSemaphoreGive(xSemaphoreHandle xSemaphore)

[Reference: https://exploreembedded.com/wiki/RTOS_Basics_:_Semaphore]

Synthetic Workload generation:

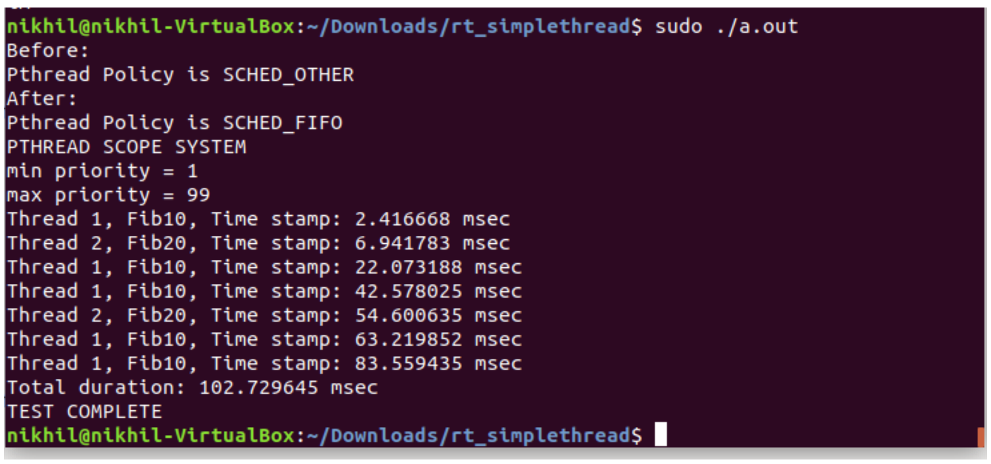
```
#define FIB_TEST(seqCnt, iterCnt)      \
  for(idx=0; idx < iterCnt; idx++)    \
  {                                   \
    fib0=0; fib1=1; jdx=1;            \
    fib = fib0 + fib1;                \
    while(jdx < seqCnt)               \
    {                                 \
      fib0 = fib1;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

This is synthetic work load generation function. It has been taken from Prof Sam Siewert's reference code.

Synthetic workload analysis and adjustment on test system:

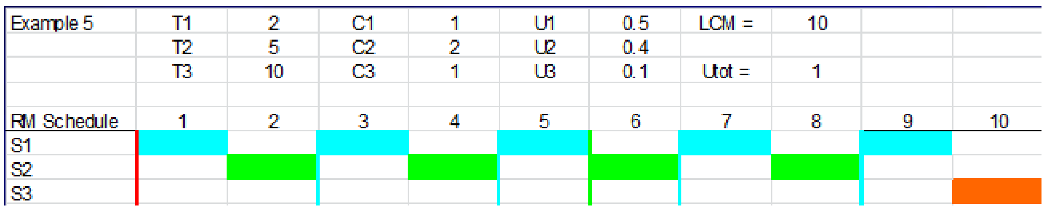
In this code, main thread spawns two threads using pthread_create() API. This code is used to find the execution time for Fibonacci series. Initially all the threads have been given certain attributes. Next scheduling policy is changed to SCHED_FIFO. Also, the main task is given the highest priority 99, thread1 has 98 while thread 2 has 97. The FIB_TEST function has been called with two parameters. First parameter is set to 47 while second parameter is changed so that I get required synthetic workload adjustment. For me the value for FIB10 is 1100000 and for FIB20 value is 2200000. In the output, thread id along with function callback and time stamp is printed. FIB10 and FIB 20 are function callbacks for 2 threads.

Below is the output of the program:



```
nikhil@nikhil-VirtualBox:~/Downloads/rt_simplethread$ sudo ./a.out
Before:
Pthread Policy is SCHED_OTHER
After:
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
min priority = 1
max priority = 99
Thread 1, Fib10, Time stamp: 2.416668 msec
Thread 2, Fib20, Time stamp: 6.941783 msec
Thread 1, Fib10, Time stamp: 22.073188 msec
Thread 1, Fib10, Time stamp: 42.578025 msec
Thread 2, Fib20, Time stamp: 54.600635 msec
Thread 1, Fib10, Time stamp: 63.219852 msec
Thread 1, Fib10, Time stamp: 83.559435 msec
Total duration: 102.729645 msec
TEST COMPLETE
nikhil@nikhil-VirtualBox:~/Downloads/rt_simplethread$
```

Overall good description of challenges and test/prototype work:



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										🟧

This timing diagram is depicted by the output with following parameters:

T1 – 20ms, T2 – 50ms, T3 – 100 ms (Total)

C1 – 10ms, C2 – 20ms

For this question, I started with rt_thread and prof sam siewert's code and made some minor changes. Main task spawns two tasks. Lets consider main task as Task 1 and other two as task 2 and 3. First both tasks 2 and 3 are allowed to run. Task 2 hits sem_wait and stops and then task 3 runs for more 10 ms. At this time main task asks task 2 to run again. Next task runs into sem wait and task 2 continues running. This process is repeating itself after some time. Major challenge in this problem was to find the right second parameter(iteration count) for FIB_TEST function. I did this by trial and error method on VM.

References:

1) http://mercury.pr.erau.edu/~siewerts/cec450/code/sequencer/lab1.c

2) http://ecee.colorado.edu/~ecen5623/ecen/labs/Linux/IS/Report.pdf

3) http://mercury.pr.erau.edu/~siewerts/cec450/documents/Papers/liu_layland.pdf

4) https://en.wikipedia.org/wiki/Rate-monotonic_scheduling

5) https://www.hq.nasa.gov/alsj/a11/a11.1201-pa.html