Lab Exercise 1 – Invariant LCM Schedules

Observations and Analysis

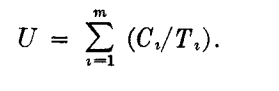
1.

Timing diagram of RM fixed policy schedule is as below :



*Fig 1.1 Timing Diagram for one LCM (15 cycles)*

The utilization for processor is defined by fraction of processor time spent in execution of task set [1] and mathematically as following:

(i)

where:

U is Utilization Factor

Ci – Worst case Execution time of a Task (run time) for Task i

C1, C2 and C3 for above Task 1 ,2 and 3

Ti – Request Time period (act as Deadline/Release time for RMS) for Task i

T1, T2 and T3 for above Task 1 ,2 and 3

Ci/Ti is fraction of processor time spent in executing Task

Therefore, calculating Ci/Ti for each task

For Task1: C1/T1 = 1/3.

For Task2: C2/T2 = 2/5.

For Task3: C3/T3 =3/15 = 1/5.

Utilization, U= 1/3+ 2/5 + 1/5 = **0.933** (as observed in above Timing diagram) --------(ii)

RM (Rate Monotonic fixed policy) states that higher priority is assigned to a Task having most frequently requested and lowest priority is assigned to task least frequently requested. The timing diagram is plotted by this definition.

Further Rate Monotonic Scheduling (RMS) Least Upper Bound(LUB) [2] is defined as follow:

(iii)

where m: is the number of services.

Using [iii] for three services U less than (2^ (1/3)-1) = .779763.

As **.933 > .7797,** indicates that it is not feasible.

But as we can see from Fig 1, Task 1 ,2 and 3 are plotted for a LCM of 15 ms and none of the services miss the deadline and is LCM invariant (does not change when harmonically repeated), it is unlikely to miss a deadline concluding to be safe using RMS.

According to [1] this Least upper bound test is a sufficient condition but **not a necessary** condition for feasibility. RMS is a pessimistic feasibility test and thus if the U is well above the required value, still it can be feasible.

**In conclusion, the above services with priority assigned according to RM are feasible and safe.**

2.

Ans

**Summary**

While descending Apollo 11 at moon CPU suffered from an overload condition, which in turn could have resulted in aborting the first landing of moon. When Apollo 11 was trying to land on lunar, an alarm 1202 started buzzing indicating that the CPU is does not have the resources to systems that are required safe landing. Thus system working unreliably in a way providing outputs delayed as CPU was over utilized.

Overloading of CPU was due to a radar system tasks erroneously requesting for CPU resources for tasks that was not required for landing. Thus a low priority task which didn’t have any purpose in landing was utilizing CPU and preventing the landing.

The best part for Apollo 11 code was smartly designed code. It was designed in asynchronous executive and whenever overload occurred it would restart/reboot the system and dropped lower priority task and allowed only higher priority task to use the CPU.

This design of software saved the mission and allowed Apollo descent on moon.

**Root Cause Analysis**

As described above 1202 alarm was buzzing continuously when Apollo 11 was landing on Lunar. During the designs of that time function call would scan VAC areas availability for executing them. The unavailability of this areas buzzed 1201 and if no core sets were available 1202.

This would be background of buzzing of alarms. Further the unavailability of VAC areas and Core sets was due to radar tasks picking up electrical noise and requesting this resources. The Radar task was a trivial task for landing, but delaying the CPU and important task were missing deadlines

**Rate Monotonic Scheduling would prevent Apollo 1201/1202 errors?**

**I**n normal higher priority task would have taken CPU cycle and executed correctly. However, in RMS higher priority is given to frequently occurring tasks.

First assumption from Apollo incident would be that radar task was frequently occurring than landing which was particular onetime event. Secondly assuming that radar was requesting CPU after fixed period.

If RMS policy was used for Apollo 11 scheduling of tasks and a radar failure event would have occurred.

1. It could have failed the mission by giving higher priority to Radar system and not sparing CPU for lower priority(RMS) task such as landing which is more important task.
2. Secondly the Assumption [A5] mentioned below of RMS is violated as landing is hard deadline non- periodic task

Thus tasks could have been not feasible due to above reasons if RMS was applied

**RMS LUB plot**

Fig2.1 is Plot of Least upper bound as function of number of services according to RMS LUB policy as mentioned in Question 1 point (iii). An Excel Sheet has been attached for reference under submission folder Q2.

*Fig 2.1 Least Upper Bound as function of number of services*

RMS policy has three key assumptions are:

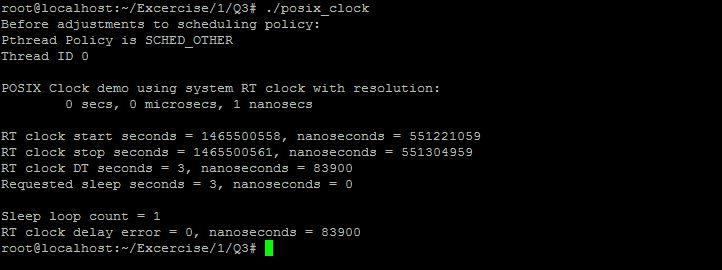
**(A1)** The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.  
**(A2)** Deadlines consist of run-ability constraints only--i.e., each task must be completed before the next request for it occurs. Such as sensors taking readings periodically.

(A5) 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

3.

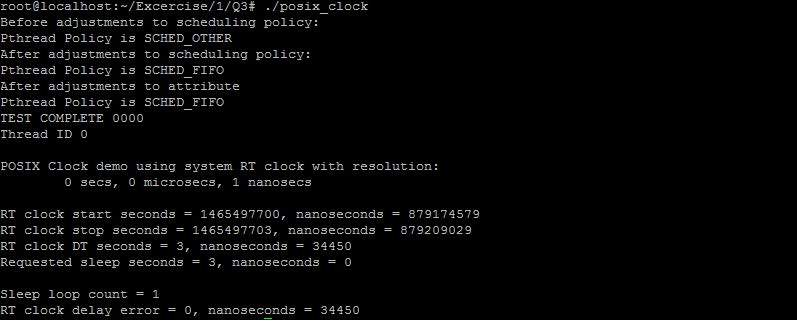
Ans

(i) Running RT\_Clock posix\_clock.c code on Ubuntu hosted on Altera DE1-SoC with scheduled FIFO\_OTHER as scheduling policy.



*Fig 3.1 Screen shot of execution of RT Clock Code using SCHED\_OTHER*

(i) Running RT\_Clock posix\_clock.c code on Ubuntu hosted on Altera DE1-SoC with scheduled FIFO as set scheduling policy



*Fig 3.2 Screen shot of execution of RT Clock Code using SCHED\_FIFO and creating a thread*

**Description of code**

The code at basic point is creating a single thread(process) and assigning scheduling policy FIFO. The main aim of the code us to check the consistency and accuracy of Real time clock of Linux.

If RUN\_RT\_THREAD is **undefined** a process/function delay\_test is called which checks without creating a thread and main is by default executed in SCHED\_OTHER policy. The Delay error of RT\_CLOCK is observed to be nearly around **83900 ns** as seen in Fig3.1**.**

If RUN\_RT\_THREAD is **defined** a main\_thread is created and scheduling parameter attribute is set to SCHED\_FIFO policy (which is FIFO scheduling policy). The delay test function call is assigned to this thread and Delay error of RT\_CLOCK is observed to be nearly around **34450 ns** as seen in Fig3.2.

**Observations**

(i)An error of multiple of ten thousand of nano seconds can be seen real time clock in Ubuntu (executed on Altera DE1 SoC) which may have **catastrophic** effects when accumulated over time for real hard time systems. As ten thousands of nanoseconds is micro seconds of deviation from actual deadline in real time which would add up to milli seconds at later point.

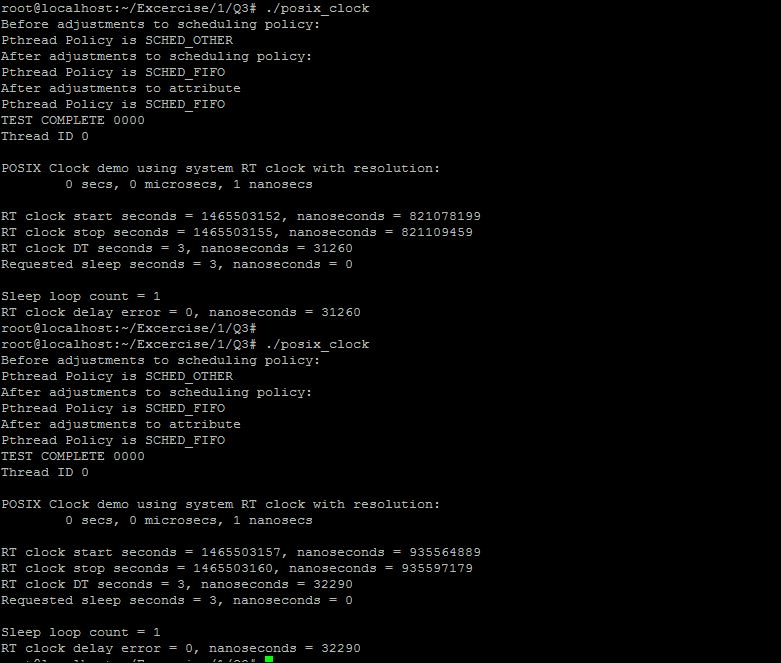
(ii)One interesting observation is error is varying by approx. 40000 ns when scheduling policy has been assigned. The RT Clock is more accurate with SCHED\_POLICY. I can’t analyze the reason for the same.

(iii)Importance of

1. Interrupt handler Latency[2] – It can be defined as the time from which an interrupt is asserted and Program counter is vectored to an interrupt handler . It is time between interrupt occurrence and execution of first ISR instruction. This latency can depend on time for saving the PC in stack, having registers (saving a context), hardware delay, other interrupt (ISR) execution etc. Lower latency leads to more deterministic results of real time hard deadlines.
2. Low Context Switch – It is time taken to a run new thread and save the context (PC and registers) of presently running thread which is preempted by new thread, so when it is dispatched again it can be start running from previous preempted point. It is significant to this machine code instructions (Context overhead) to meet deadlines accurately and deterministically. Keeping overhead value less helps to reduce drift in execution time
3. Service Timers – The accuracy of Stable timers provides perfect timings thus allowing no deviation from course of thread and right time of preemption. This further reduces drift and jitter. Further, more the timers are unstable, it would lead incorrect and miss of deadlines. Even if there are issues with timers and are deterministic (known and constant time of drift) it can be corrected while designing the system.

(iv)Accuracy of RT Clock Code –

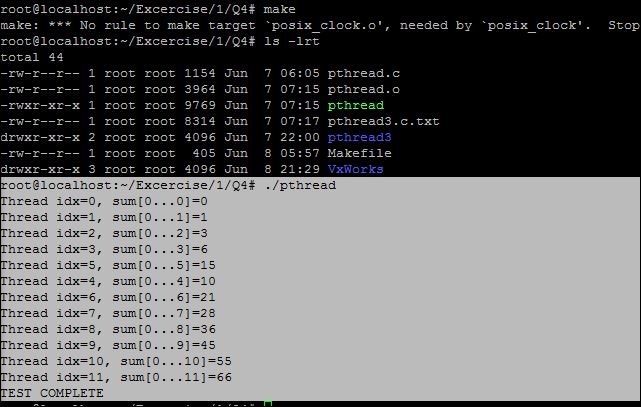
As Can be seen from Fig 3.3 and observation (ii) there is variation in execution of time thus changing values of delay in thousands of nano seconds. It uses nanosleep function call to suspend thread and clock\_gettime which may be causing variations. The inaccuracy in timer services cause jitter in timing leading to inaccuracy in RT Clock analysis code. The context switch time variation also would be leading to this issues . Therefore, the Real Time Clock tested on this system is less accurate in my belief as above considerations are not incorporated.



*Fig 3.3 Screen shot of execution of RT Clock Code using SCHED\_FIFO and variations in delay error*

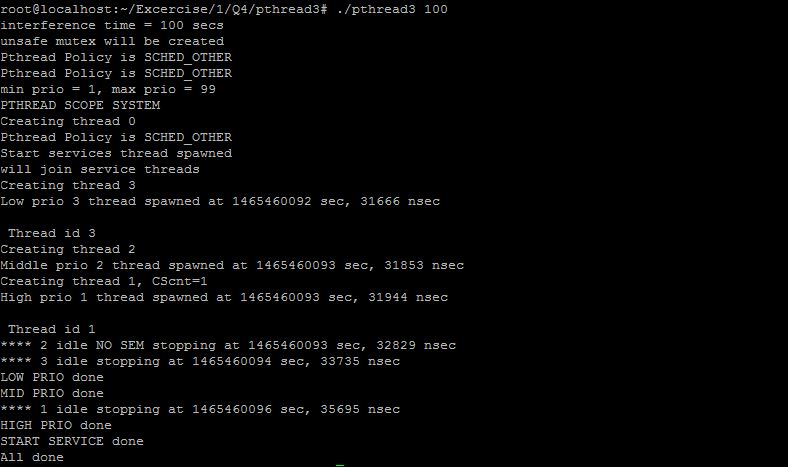
4.

(i) Running Simple Thread pthread.c code on Ubuntu hosted on Altera DE1-SoC



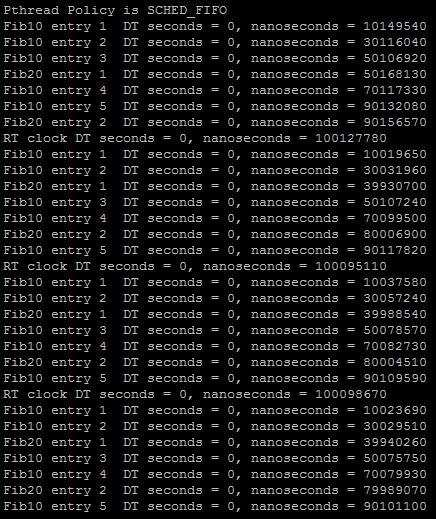
*Fig 4.1 Screen shot of execution of Simple Pthread*

(ii) Running Sync Thread (pthread3.c) code on Ubuntu hosted on Altera DE1-SoC



*Fig 4.2 Screen shot of execution of Sync Pthread*

(iii)(a) Running Binary Semaphore Linux code (binSemLinux.c) ported from VXwroks LCM variant implementation on Ubuntu hosted on Altera DE1 SoC. Please find the code and makefile attached with submission in Vxworks folder for Q4.



*****Fig 4.3 Screen shot of execution(Testing) of Scheduling LCM Invariant Fib10 and Fib20 affinity (Set one)*

*Fig 4.4 Screen shot of execution of Binary Semaphore Pthread (VxWorks) source from Question*

(b)For Testing the implementation of LCM invariant scheduling in accordance to VxWorks (as seen Fig4.4). The code calculates time from **start point** (like event 1 in Fig 4.4) and those events of completion (execution time) of Fib10 and Fib20 marked event points and flag points using clock\_gettime and delta\_t fucntions

(c) Flag points are marked for Fib10 by orange points (2,3,5,6,8)

Flag points are marked for Fib20 by blue points (4,7)

Event 1 – start of the Event

Event 9 – End of the Event in Fig 4.4 and Fig4.3

From Fig4.3 we can see the complete Variant is for approx. **100ms**

(d) By comparing these above points between Fig 4.4 and Fig 4.5, we can see that it is meeting the timings and scheduled correctly as per RMS scheduling policy

Porting from VxWorks Explanation

While porting code from Vx Works to Linux the code incorporates/changes the following:

(I)Start () in Vxworks is main () function as in Linux which creates the Sequencer thread which creates the Fib10 and Fib20 threads.

(ii)Instead of using taskSpawn to create a Task in VxWorks, Linux uses pthread\_create to create Thread and assigns priority and assign the start function to be executed. Task is similar to Thread(process) in Linux with assignment of Stack and End/Exit point for them.

(iii)To use Smeaphores in Linux system call functions Sem\_wait () comparable to SemTake and Sem\_post() is comparable to SemGIve of VXworks are available . By default, the Semaphore in Linux are counting and allow threads to increment using SemWait and decrement using SemTake.As Binary Semaphore is used for synchronization Sem\_Wait() blocks the execution of thread while Sem\_Post() releases the semaphore allowing thread to complete execution .

For scheduling for the LCM invariant higher priority was assigned to Fib10 as it was repeating period is more than Fib20 following RMS policy.

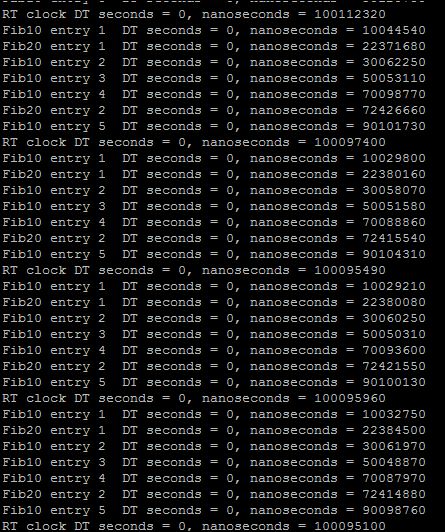
(iv)For creating a dummy execution of task a synthetic work load of using FIBTEST was created with iterations. This uses the CPU cycles and thus a software generates a synthetic load.The iterations used provided variability in load generation. The point 2) in challenges discusses value used in code for generating Fib10 for 10 ms and Fib20 20 ms .

Challenges faced and Synthetic Load Adjustment:

Major challenged faced after completing(porting) initial code of scheduler is that

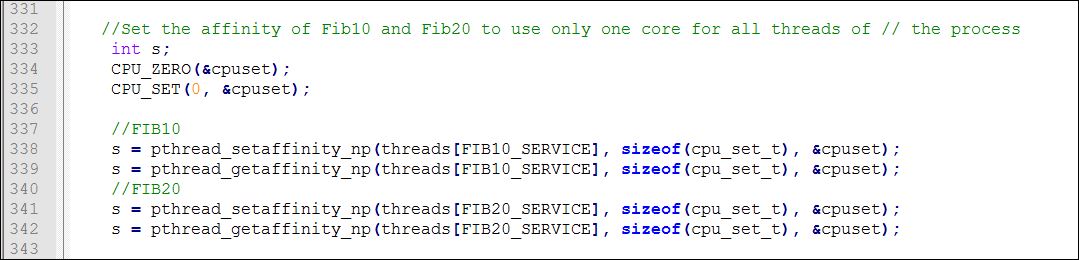
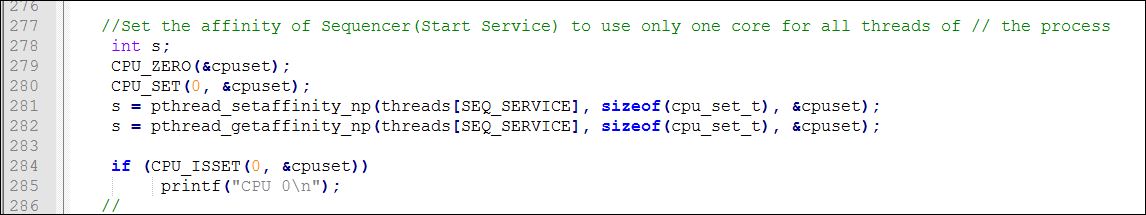
1. (i)Thread Fib10 and Fib20 were executing simultaneously even after different priority and FIFO scheduling assigned. This can been seen from Fig 4.5 where Fib20 thread is completing execution after the aprrox 20 ms from start time where it should have completed only after 30 ms.Thus Fib10 and Fib20 were executing simultaneously using multiple cores of Ubuntu .

(ii)However after incorporating use of affinity and allowing usage of only one Core for all the threads of this process as seen code snippet Fig4.6 and Fig 4.7. The execution of Fib10 and Fib20 was as seen in Fig4.3



*Fig 4.5 Screen shot of execution of Binary Semaphore Pthread (Linux) using set using multiple core*

*Fig 4.6 Snippet of code of affinity in main for Start Service (Scheduler)*



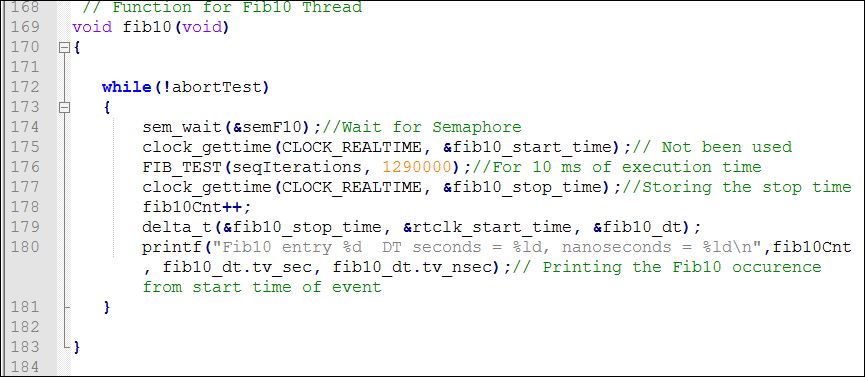
*Fig 4.7 Snippet of code of affinity in Start Service for Fib10 and Fib20*

(i)While testing after implementation of affinity usage, the Fib10 and Fib20 were still not able to schedule at the right time, As the Fib20 was been preempted as its execution time high and its deadline was being missed. Fib10 was able to preempt being a higher priority value.

(ii) TO schedule Fib20 it before Fib10 would preempt this thread, the load was being changed in Fib10 and Fib20 by changing the iteration value.After scheduling Fib10 and Fib20 once, twice and then multiple times. The value of Fib10 was adjusted first from 1700000 with multiple attempts and found **1290000** giving optimum close to 10 ms(Fig4.3 point 2 ) and not preempting Fib20 before completion of execution (Fig4.8).

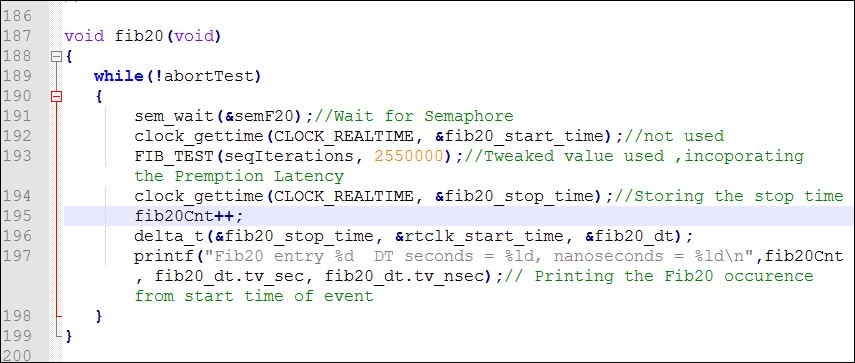
(iii)Similarly Fib20 load is also been adjusted from 3400000 to **2550000** to meet the execution time of 20 ms (Fig4.3 and point 4 completing 30 ms from start event) and complete execution(Fig4.9)

(iv)One of the reasons for the is **Context Switch** time which changes the time deviation for execution of Fib20 , as it is been preempted by Fub10



*Fig4.8 Load adjustment in Fib20 Value 1290000 for Ubuntu Altera DE1 SoC*

*Fig4.9 Load adjustment in Fib20 Value 2550000 for Ubuntu Altera DE1 SoC*



**References**

[1] Lui and Layland Paper <http://ecee.colorado.edu/~ecen5623/ecen/rtpapers/archive/PAPERS_READ_IN_CLASS/liu_layland.pdf>

[2] Real Time Embedded Components and Systems by Sam Siewert and John Pratt