ECEN 5623 – Real Time Embedded Systems

Summer 2017

EXERCISE 1

INVARIANT LCM SCHEDULES

Jeet Baru

Email: jeet.baru@colorado.edu

ID Number: 107189037

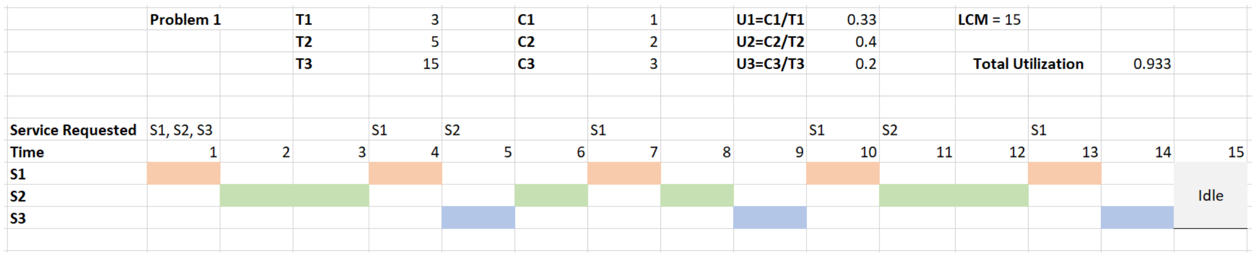
Problem 1

The purpose of a real-time scheduling algorithm is to ensure that critical timing constraints, such as deadlines and response time, of tasks are met. When necessary, decisions are made that favor the most critical timing constraints sometimes even at the cost of non-critical constraints. Tasks with critical time constraints are said to be having hard deadlines and others have soft deadlines. Not meeting the hard deadline shall lead to a catastrophe and hence the need of scheduling algorithm arises that allocates, in an embedded system, say, resources like CPU in a way such that all hard deadlines are met. One such algorithm is the Rate Monotonic Policy.

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).

Timing Diagram:

Now we are given 3 services S_1, S_2 and S_3. Each service is requested at a fixed rate, every T_1 = 3ms, T_2 = 5ms and T_3 = 15ms. Also the completion time for each service is C_1 = 1ms, C_2 = 2ms and C_3 = 3ms. We are required to draw a timing diagram of the schedule of the 3 services using rate monotonic policy and check for its feasibility.



	Problem 1		**T1**	3		**C1**	1		**U1=C1/T1**	0.33		LCM = 15			
			T2	5		**C2**	2		**U2=C2/T2**	0.4					
			T3	15		**C3**	3		**U3=C3/T3**	0.2		**Total Utilization**		0.933	
Service Requested	S1, S2, S3			S1	S2		S1			S1	S2		S1		
Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S1	🟧			🟧			🟧			🟧			🟧		
S2		🟩	🟩			🟩		🟩			🟩	🟩			Idle
S3					🟦				🟦					🟦	

Figure 1: Timing Diagram

From the timing diagram drawn above we can confirm that the schedule is feasible since it shall be mathematically repeatable as an invariant indefinitely. This is inferred by drawing the schedule for a time period equal to the LCM of T_1, T_2 and T_3 which is 15 and hence for any time beyond this point the schedule is repeatable.

Method:

We assume the worst case scenario of all three tasks being requested at once at time 0. Task 1 has highest priority, followed by Task 2 and Task 3 based on their periods of requests. So because of its highest priority we begin execution with Task 1. At time 3 we have another request of Task 1. Although Task 3 is yet to be processed we perform Task 1 due to its higher priority. At time = 5 we have completed execution of task 1 and 2 and partial (1/3) execution of task 3. So when a higher priority task is requested the current task is preempted and we go till all pending tasks are processed. At time = 14 we have completed all pending tasks and have the last time window as idle until this repeats for the next period equal to the LCM of periods of all tasks.

Safety and Feasibility:

To check for safety of our schedule we visit C. L. Liu and James W. Layland's paper on 'Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.' In this paper the authors have derived a least upper bound to processor utilization for fixed priority systems to be equal to:

$U = m(2^{1/m} – 1)$, where m is number of services scheduled

Thus LUB for our example is 0.78.

Total CPU utilization by scheduling using the rate monotonic policy is $U_1 + U_2 + U_3 = 0.933$

Although the CPU utilization is greater than the least upper bound for 3 services we shall conclude from the timing diagram that our schedule is feasible and safe. But there is a serious doubt on its safety here we have not taken any interrupt or context switch latency into account. Also jitters and drifts in frequencies might increase the total CPU utilization. Because we only have 10 percent of the CPU free and hence very little margin available I shall stay on the fence about its safety specially in hard real time applications.

Problem 2

In this problem we do a case study of 'Apollo 11 Lunar Lander computer overload.' We refer to the NASA account as well as description of the events by chief software engineer Margaret Hamilton recounted by Dylan Mathews.

Apollo 11 Case Study:

Part of NASA's Apollo program, Apollo 11 created history by becoming the first successful attempt to take man to moon. Apollo Guidance Computer or Lunar Guidance Computer was used in both command module and lunar module to provide navigational assistance. This computer was hugely constrained by the amount of available memory. The available fixed memory was used to store the executable and other fixed data and erasable memory was used to store variable data. Erasable memory was so little that same memory was sometimes shared by 7 different tasks. Hamilton's code was a real time multitasking operating system that controlled interrupt driven and time dependent tasks by prioritizing them and accordingly assigning memory resources.

While assigning each task, the scheduler assigned erasable memory called "core set" and if more storage was required, the task was allotted a VAC (vector accumulator) area. There were 5 VAC areas and 7 core sets available to be used by the program. Tasks had a NOVAC flag to signal if they required a VAC area. The scheduler first checked the flag and looked for an available VAC area to be reserved, non-availability of which resulted to set Alarm 1201 and the program to branch to Alarm/Abort routine. If VAC was wasn't required, its scanning was skipped. Similarly, the core sets were then scanned and non-availability resulted to set Alarm 1202 and the program to branch to Alarm/Abort routine.

Cause of fault:

Major issue during the Apollo 11 mission was due to misconfiguration of radar switches which led to multiple requests to process rendezvous radar data which was not available. Due to this the scheduler acknowledged these request and the limit available erasable memory got filled up first the core sets.

Due to unavailability of core sets, alarm 1202 was generated. During landing alarm 1201 was generated as the scheduling request that caused the overflow had requested the VAC area. A catastrophe was avoided as the software recognized the request of data to be of secondary importance and hence rebooted to normalcy.

The rate monotonic policy requires the hard deadline tasks to be periodic. In this case due to misconfiguration of radar switches (hardware fault) there multiple aperiodic requests. Thus violating the Rate monotonic policy. Also Rate monotonic policy states that the task with the least period should have the highest priority. In this case the radar data was not considered of primary importance and thus it violates the rate monotonic policy.

Rate Monotonic Least Upperbound of CPU Utilization:

Now we read Liu and Layland's paper which describes Rate Monotonic policy and the Least Upper Bound of CPU utilization for safe operation. In the paper the authors have derived an upper bound for processor utilization depending on the number of services. The expression of the LUB is:

$U = m(2^{1/m} – 1)$, where m is number of services scheduled

We now plot the above expression with respect to m. We observe that for 1 service CPU utilization can be 100%. But as number of services increase the maximum utilization reduces and steadies at about 70%

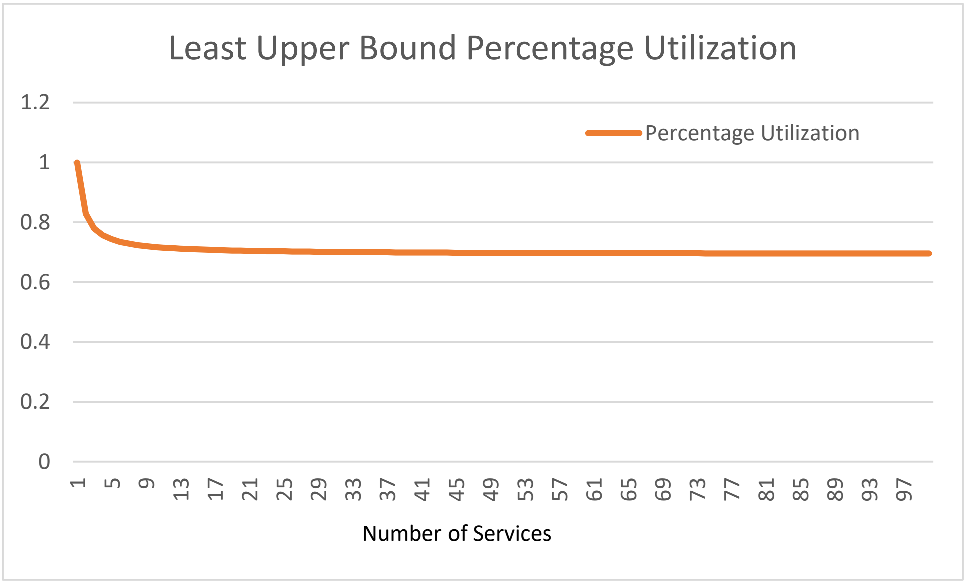


Figure 2: Plot of Maximum Utilization Vs Number of Services

On reading the research paper a couple of points that I did not understand fully were:

1. In the derivation for LUB on CPU utilization how we assumed the C2' = C2 + Δ but took C3' = C¬3
2. Also the condition where 2 tasks having same period and completion time is not considered. We could have randomly assigned higher for one of the task and then analyze timing. Although this situation is rare, but analysis for it would have been helpful.
3. Also the concept of buffer task in order to relax the least upper bound.

Key assumptions made in the paper about hard real time environment that are critical for the authors' analysis are:

1. Tasks having hard real time deadlines have periodic requests.
2. Run time for each task is constant and does not vary with time.
3. The tasks are independent and the requests for a certain task do not depend on the initiation or the completion of requests for other tasks
4. Each task must be completed before the next request for it occurs.

RM Policy for Apollo 11 Errors

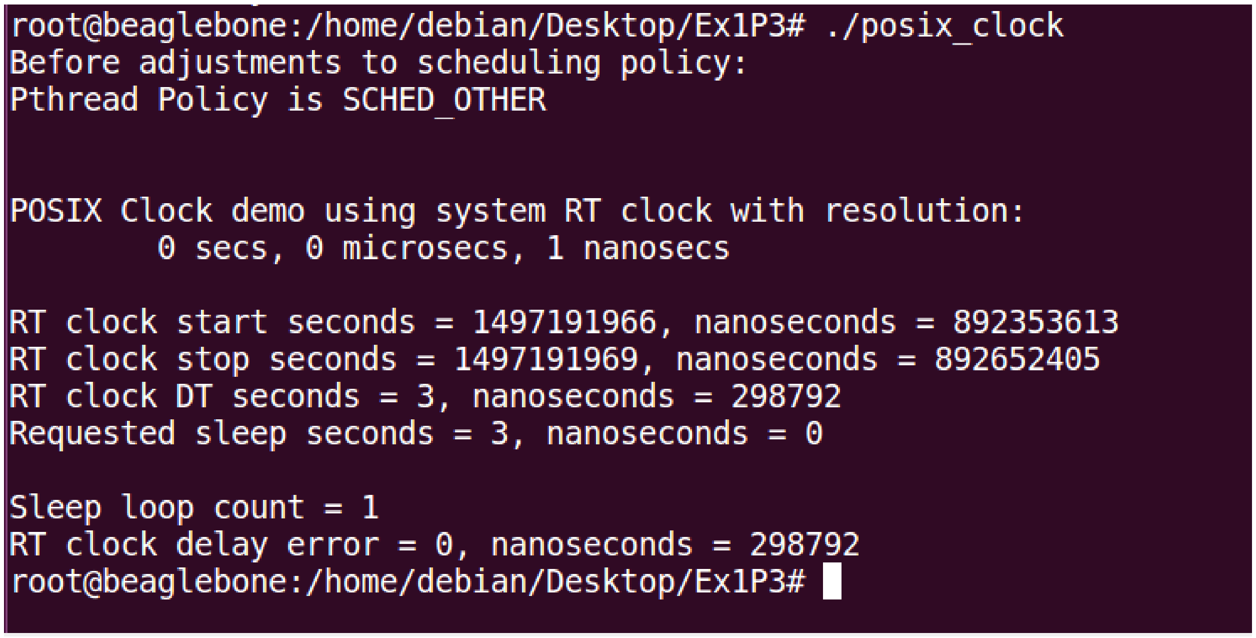
There were 2 errors generated on the Apollo 11 mission. Alarm 1202 occurred due to malfunctioning radar, which generated requests for processing data that wasn't ready. Even if Rate Monotonic Policy was implemented this error could not have been prevented because it was a hardware fault. Due to this the requests were now generated at a faster rate thus changing its periodicity and hence violating a key assumption of the policy.

Alarm 1201 was generated at the time of landing and there is no clear information what triggered the lack of VAC areas and hence the alarm. So if there was an error in scheduling a task during landing the error could have been prevented by using rate monotonic policy. But if the cause of repeated requests was a mechanical/hardware fault then due to violations of assumptions even implementing the rate monotonic policy could not have prevented the fault.

Problem 3

The code for RT clock was downloaded built and executed for BeagleBone Black that had Embedded Linux (Debian) on it.

The screenshot of the output is as shown below:



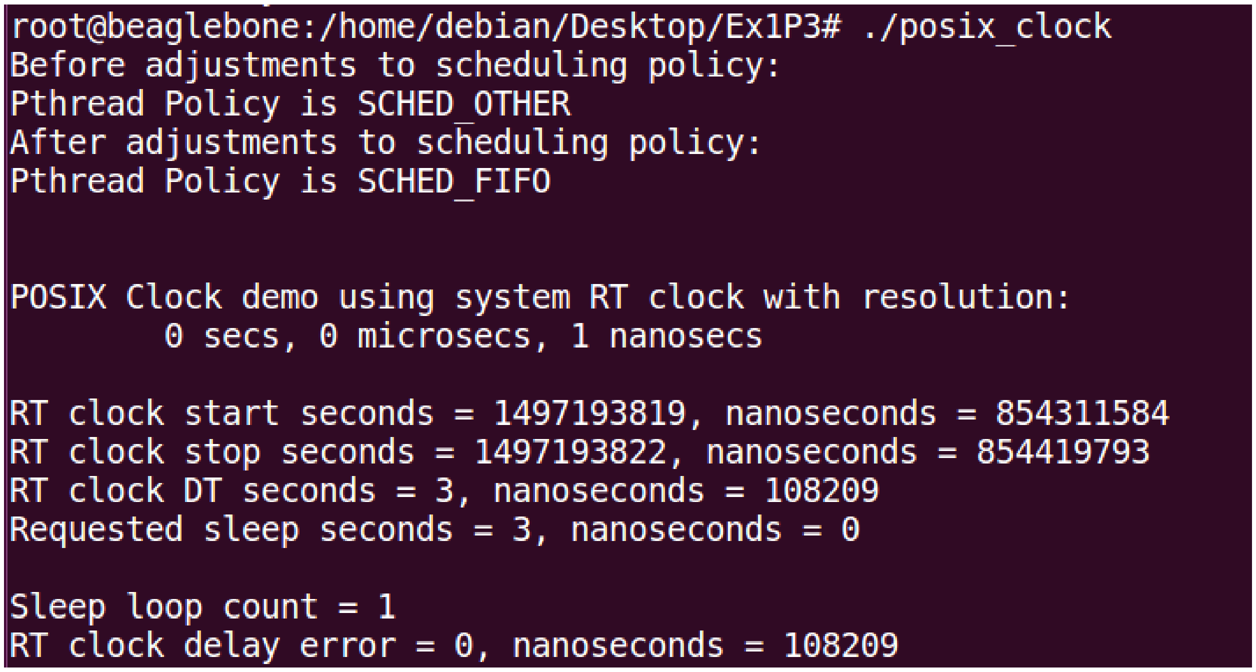
```
root@beaglebone:/home/debian/Desktop/Ex1P3# ./posix_clock
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER


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

RT clock start seconds = 1497191966, nanoseconds = 892353613
RT clock stop seconds = 1497191969, nanoseconds = 892652405
RT clock DT seconds = 3, nanoseconds = 298792
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 298792
root@beaglebone:/home/debian/Desktop/Ex1P3#
```

Figure 3: Execution with policy SCHED_OTHER



```
root@beaglebone:/home/debian/Desktop/Ex1P3# ./posix_clock
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 = 1497193819, nanoseconds = 854311584
RT clock stop seconds = 1497193822, nanoseconds = 854419793
RT clock DT seconds = 3, nanoseconds = 108209
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 108209
```

Figure 4: Execution with policy SCHED_FIFO

As seen above I have executed the code both with and without the compile time switch and hence for scheduling policy SCHED_OTHER and SCHED_FIFO.

The code makes use of the real time clock and generates a precise delay. The function print_scheduler() prints the scheduler in use for the code. The function delta_t() is used to calculate the difference between two time stamps. ‘timespec’ structures are passed to it as parameters and it returns the difference between the times. Function end_delay_test() is called after the delay in order to print the start timestamp, stop timestamp and the difference between them. It also prints the expected delay and the delay obtained and giving an idea of accuracy. Timestamps are recorded using the function clock_gettime(). This function stores the current time in a structure whose member variable store values of seconds and nanoseconds since epoch time. The important implementation and execution of delay happens in the delay_test() function. The function firstly prints the resolution of the clock using the time.h library. Then it executes a loop for the delay. The delay is generated using the nanosleep() function, that delays execution until the time specified in its parameter (struct timespec) has elapsed. The clock used in the function is CLOCK_REALTIME system clock. If this do-while loop is interrupted before the 3s delay is complete, it runs again for the remaining time. This way the program is executed without using threads. If we are to use threads we first set thread attributes, change its scheduling policy to SCHED_FIFO and set its priorities. Then the thread is created and joined to the main thread using functions from the pthread.h library. I made changes to the makefile to make it run for both with and without threading.

From the screenshots shown above we see that the code running with the FIFO policy was faster than the one using the default scheduling policy. The execution using threads was observed to be faster.

Essential Features of RTOS:

Three essential features that make a good RTOS are:

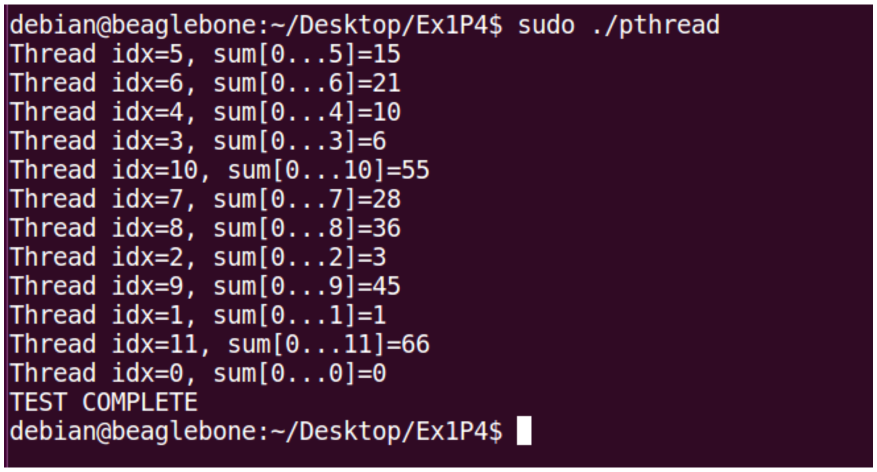
1. Low interrupt handler latency: The delay between assertion of an interrupt signal by a device and the time at which the PC is vectored to an interrupt handler is known as the interrupt latency. In most embedded systems with an RTOS, the RTOS guarantees that the interrupt latency will never cross a certain maximum value. In hard real time applications it is essential that the service meets its deadline in order to prevent any catastrophe. A low interrupt handler latency ensures less time wasted between interrupt generation and its processing and hence ensuring the critical sections meet deadline. Hence low interrupt handler latency is essential.
2. Low context switch time: A context switch is the process of storing and restoring the state of a process or thread so that execution can be resumed from the same point at a later time. Using the context switch, a large number of processes can make use of the single CPU. Context switching is a very time consuming process and a complex procedure for the RTOS to implement. It is an overhead in midst of the other essential functions that the application needs to perform. Hence low context switch time ensures faster and optimized real time operating system.
3. Stable timer services with low jitter and drift: When latency and/or timing of an operation or process changes with each iteration, this is jitter—that is, when latency/timing is not constant. If there is low jitter it means the timing estimations made can be more or less deterministic. Deterministic knowledge of time for interrupts, timeouts and relative time shall help in making robust fixed priority hard real time systems. Drift in clock is a phenomenon where frequency is not same as that of the reference clock. In case of a clock drift all calculations made shall change as the clock frequency changes. Hence it is essential to have low clock drift and jitter.

From the screenshots above we observe that the real time clock on BeagleBone Black gives an error 100 microseconds at least for 3 seconds. This includes the error due to context switching and recording timestamps. The clock used is CLOCK_REALTIME. At first this performance seems fairly good, but on further analysis we see that this error could multiply to an error of 3 seconds each day and 20 mins for the whole year. This kind of a large error shall not be acceptable for critical systems. For soft real time applications this accuracy shall be acceptable.

Problem 4

The pthread.c program simply creates 12 threads using the pthreads API. A structure is created and unique instances of this structure containing a unique thread ID is used for each of the threads. And then these unique instances are passed as arguments to the counterThread method so that there is no data corruption due to shared memory access between the threads. Each of the threads prints out its unique ID and a sum and the sequence in which these threads are released are solely at the discretion of the scheduler and OS. After all of the threads have been executed, they are then joined to the main thread.

As expected, we get a sequence of thread ID's and sums in the output on running the compiled code on BeagleBone Black:

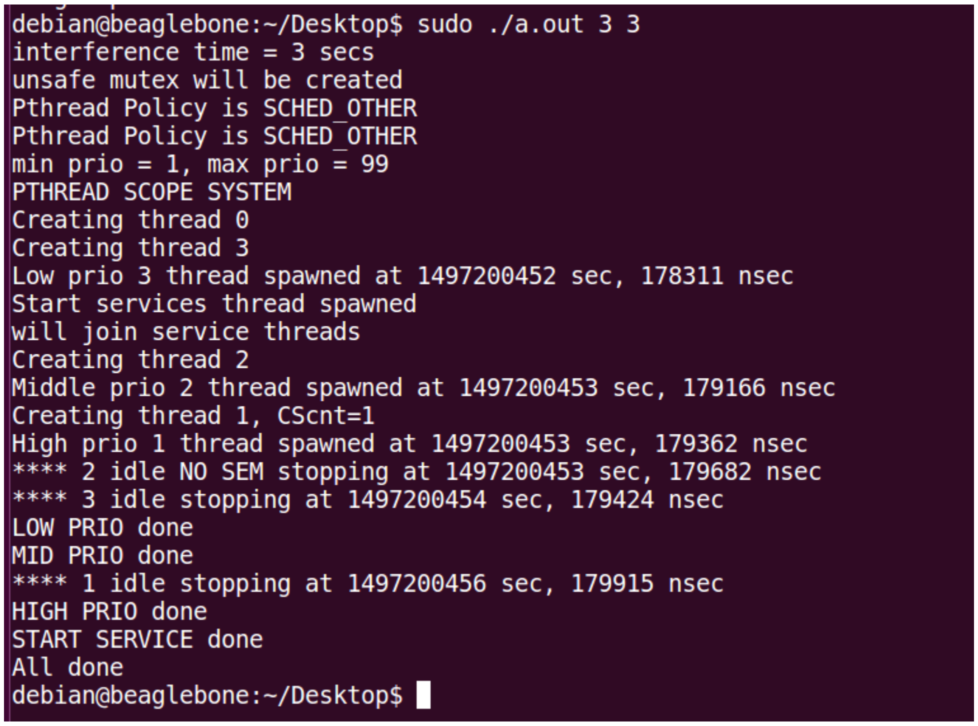


```
debian@beaglebone:~/Desktop/Ex1P4$ sudo ./pthread
Thread idx=5, sum[0...5]=15
Thread idx=6, sum[0...6]=21
Thread idx=4, sum[0...4]=10
Thread idx=3, sum[0...3]=6
Thread idx=10, sum[0...10]=55
Thread idx=7, sum[0...7]=28
Thread idx=8, sum[0...8]=36
Thread idx=2, sum[0...2]=3
Thread idx=9, sum[0...9]=45
Thread idx=1, sum[0...1]=1
Thread idx=11, sum[0...11]=66
Thread idx=0, sum[0...0]=0
TEST COMPLETE
debian@beaglebone:~/Desktop/Ex1P4$ █
```

Figure 5: Execution of pthread.c on BeagleBone Black

The pthread3.c code also uses pthread API to create 3 threads. This code helps in understanding the operation of mutex locks. In this program we have two idle functions, one with no mutex lock and other with mutex lock. Thread 2 runs the idle function without the mutex lock and threads 1 and 2 run functions with mutexes. From the output we see that. First when thread 2 is inside the idle function it is preempted by Thread 2. Due to the mutexes, the execution waits until thread 3 has completed and stopped and only then will thread 2 complete execution and finally thread 1 executes.

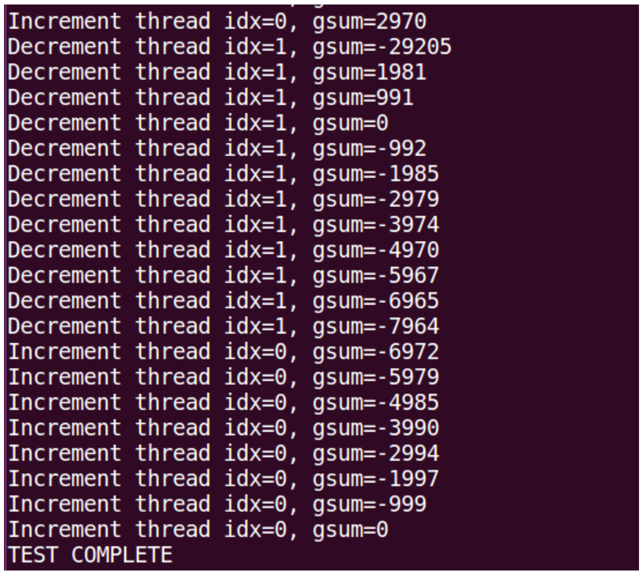
We now execute pthread3.c on BeagleBone Black. And obtain the output as shown below



```
debian@beaglebone:~/Desktop$ sudo ./a.out 3 3
interference time = 3 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Creating thread 3
Low prio 3 thread spawned at 1497200452 sec, 178311 nsec
Start services thread spawned
will join service threads
Creating thread 2
Middle prio 2 thread spawned at 1497200453 sec, 179166 nsec
Creating thread 1, CScnt=1
High prio 1 thread spawned at 1497200453 sec, 179362 nsec
**** 2 idle NO SEM stopping at 1497200453 sec, 179682 nsec
**** 3 idle stopping at 1497200454 sec, 179424 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1497200456 sec, 179915 nsec
HIGH PRIO done
START SERVICE done
All done
debian@beaglebone:~/Desktop$ █
```

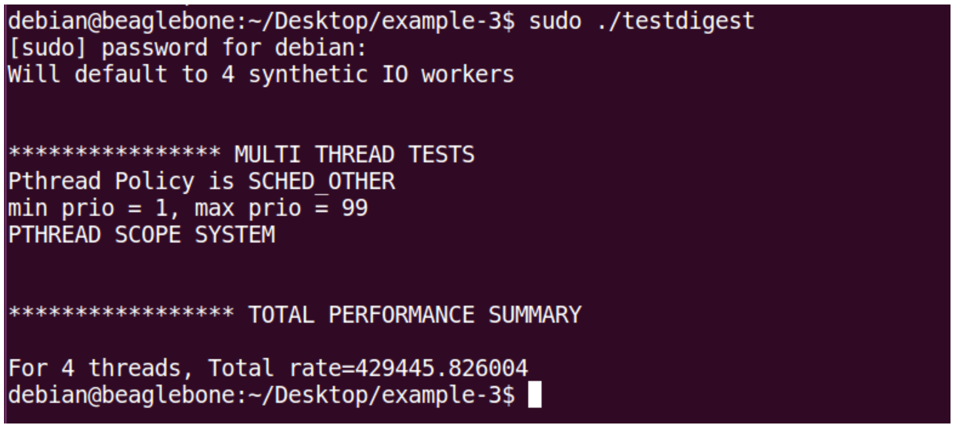
Figure 6: Execution of pthread3.c on BeagleBone Black

The incdecthread.c program creates two threads to act on a single global variable. One of them increments the variable and the other one decrements the value. The problem that this code faces is that any of the two threads can access the variable anytime without having exclusive access to it.



```
Increment thread idx=0, gsum=2970
Decrement thread idx=1, gsum=-29205
Decrement thread idx=1, gsum=1981
Decrement thread idx=1, gsum=991
Decrement thread idx=1, gsum=0
Decrement thread idx=1, gsum=-992
Decrement thread idx=1, gsum=-1985
Decrement thread idx=1, gsum=-2979
Decrement thread idx=1, gsum=-3974
Decrement thread idx=1, gsum=-4970
Decrement thread idx=1, gsum=-5967
Decrement thread idx=1, gsum=-6965
Decrement thread idx=1, gsum=-7964
Increment thread idx=0, gsum=-6972
Increment thread idx=0, gsum=-5979
Increment thread idx=0, gsum=-4985
Increment thread idx=0, gsum=-3990
Increment thread idx=0, gsum=-2994
Increment thread idx=0, gsum=-1997
Increment thread idx=0, gsum=-999
Increment thread idx=0, gsum=0
TEST COMPLETE
```

Figure 7: Execution of incdecthread



```
debian@beaglebone:~/Desktop/example-3$ sudo ./testdigest
[sudo] password for debian:
Will default to 4 synthetic IO workers


**************** MULTI THREAD TESTS
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM


***************** TOTAL PERFORMANCE SUMMARY

For 4 threads, Total rate=429445.826004
debian@beaglebone:~/Desktop/example-3$
```

Figure 8: Execution od TestDigest

Threading vs Tasking:

- Tasks are simple concept of instructions that are loaded into the memory ready for execution whereas Threads are a can split themselves into two or more running tasks.
- A Task is high level concept while a thread is low level (OS level) concept.
- Multiple threads may or may not split the work across multiple processors while multiple threads are usually has parallel implementation and gets distributed across many processors
- Threads usually split up the work into chunks such that each thread can work on one part of it. A task is something you want done and represents an asynchronous operation.

Semaphores, wait and sync:

Essentially, a semaphore is a variable that is used to control access to a common global resource by multiple processes in a concurrent system. Let's for example consider a program that has a single global variable and there are two threads wanting to access it. In the absence of a semaphore, both of these threads can act on the resource simultaneously and alter its value and therby corrupting it. To prevent this situation, we can have a semaphore to lock the resource and then release it later. Semaphores help us maintain mutual exclusion and synchronization between various tasks. To enable us to do that are two different methods called sem_wait() and sem_post(). Sem_post() essentially increments the value of the semaphore and a task is eligible to lock the semaphore if its value is greater than 0. Sem_wait() decrements the semaphore value and the semaphore is currently being held by a task. Using these techniques, we can ensure mutual exclusion and synchronization in our processes.

Synthetic workload generation:

Synthetic workload implies generating a fixed and predictable load on the CPU in order to keep the CPU at chosen utilization levels and to stress test the system to check whether it works correctly under the given scheduling with the given services. In this case, I attempted to replicate the graph below to ensure a CPU utilization of 90% such that the two threads created to run the Fibonacci sequence execute before their respective deadlines for their fixed times.

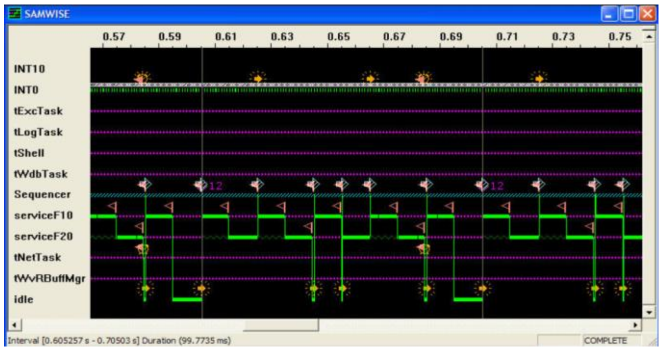


Figure 9: Required Timing Diagram

As seen from the above graph, we schedule service 1 to run for 10ms atleast once in 20 ms, and the service 2 to run for 20ms once within a 50ms deadline to achieve a CPU utilization of 90%.

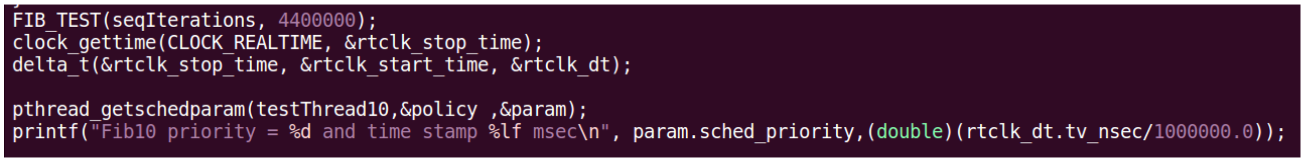
Porting VxWorks Code to Linux and Synthetic workload generation:

The VxWorks implementation is different in the sense that it doesn't have a main method. Execution begins from the start method and a task named Sequencer is spawned using a taskSpawn method. It is given a high priority (of 20) and a certain stack size. We then create two semaphores with FIFO priority and spawn two different tasks namely Fib10 and Fib20 having lower priority than itself (so that the Sequencer task itself is not preempted). In the Fib10 thread, we hold the semaphore and run the Fibonacci test for a certain number of iterations corresponding to 10msec and likewise in the Fib20 thread. While the task has not been aborted yet, we release the two semaphores. Giving a task delay of 20 ms initially to suspend the main thread for a period of 20 ms, we release the two semaphores and task 1 runs first because of its higher priority. A sequence of releases is then designed to ensure that the deadlines are met.

Citation: *The code has been adapted from Nisheeth Bhat's independent study of RM Scheduling Feasibility Tests conducted on TI DM3730 Processor - 1 GHz ARM Cortex-A8 core with Angstrom and TimeSys Linux ported on to BeagleBoard xM. Most of the code has been adapted from his study, however we have made a lot of changes regarding the timing and using the time.h library functions to port it to our hardware. (Since his implementation was for the Beaglebone)*

Porting the problem to Embedded Linux – POSIX environment, we first proceed to create two semaphores and initialize their values to one. After setting some attributes, scheduling max and min priorities and scheduling policy, we assign higher priority to the main loop relative to rt10, which has higher priority than rt20. We assign a sched_fifo sequencing for our processes and proceed to create the two threads which will be executing according to a sequence. For each of the threads, we run the Fibo method for a fixed amount of time – which is determined by trial-and-error since it varies from machine-to-machine. Trying it on my Beaglebone Black and profiling the Fibo method once and then calculating, I determined that 10ms would correspond to 4400000 iterations of the Fibo method.

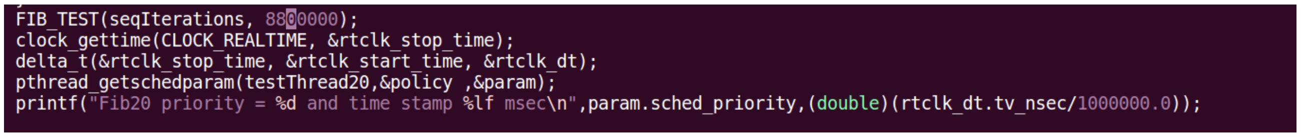
From the code snippet below, we notice that using the clock_gettime routine and CLOCK_REALTIME to time the process, we can determine the difference between start and stop to find the execution time. This helps us to verify whether the timing diagram has been implemented robustly such that deadlines are always met: -



```
FIB_TEST(seqIterations, 4400000);
clock_gettime(CLOCK_REALTIME, &rtclk_stop_time);
delta_t(&rtclk_stop_time, &rtclk_start_time, &rtclk_dt);

pthread_getschedparam(testThread10,&policy ,&param);
printf("Fib10 priority = %d and time stamp %lf msec\n", param.sched_priority,(double)(rtclk_dt.tv_nsec/1000000.0));
```

Figure 10: Fib10 parameter values

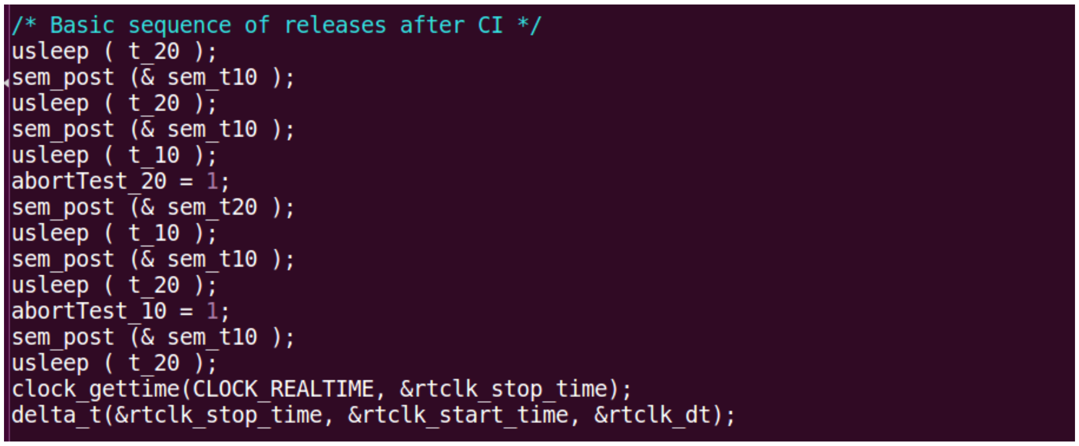


```
FIB_TEST(seqIterations, 8800000);
clock_gettime(CLOCK_REALTIME, &rtclk_stop_time);
delta_t(&rtclk_stop_time, &rtclk_start_time, &rtclk_dt);
pthread_getschedparam(testThread20,&policy ,&param);
printf("Fib20 priority = %d and time stamp %lf msec\n",param.sched_priority,(double)(rtclk_dt.tv_nsec/1000000.0));
```

Figure 11: Fib20 parameter values

The use of semaphores is very important in our implementation, since it is through the incrementing/decrementing, locking/unlocking actions of the semaphore that we are able to perform the service scheduling and synchronization to perfection. After having initialized the two semaphores to value 1 using the sem_init() function, we use the sem_post() function to unlock the semaphore. Using this function we increment the semaphore value and it is open to being locked by a waiting thread. Since the value is greater than 1, the sem_t10 semaphore is locked by the Fib_10 thread, which had been waiting for the semaphore using sem_wait(). The while loop in Fib10 thread will only run once – for 10 msec and after that the semaphore value becomes 0. So this thread will be idle for another 10msec using the usleep(20) method. Meanwhile, Fib20 thread, which had a lower priority got a chance to run for 10 msec. At the end of the 20msec period, the semaphore value is increased again using sem_wait() and the process continues.

The sequence of required load coded is shown below: -

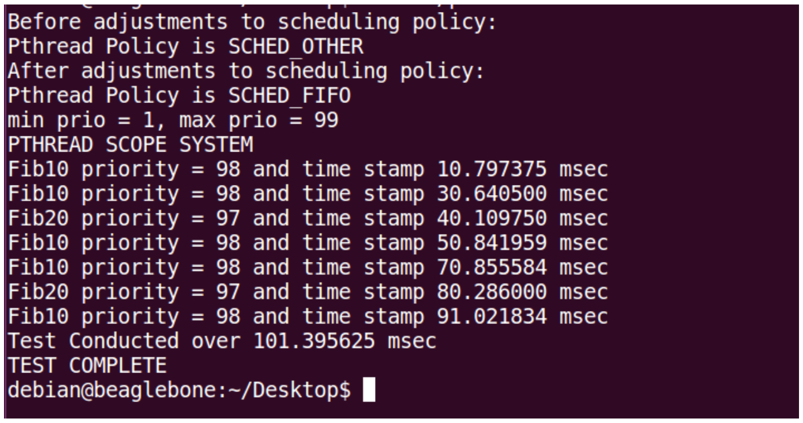


```
/* Basic sequence of releases after CI */
usleep ( t_20 );
sem_post (& sem_t10 );
usleep ( t_20 );
sem_post (& sem_t10 );
usleep ( t_10 );
abortTest_20 = 1;
sem_post (& sem_t20 );
usleep ( t_10 );
sem_post (& sem_t10 );
usleep ( t_20 );
abortTest_10 = 1;
sem_post (& sem_t10 );
usleep ( t_20 );
clock_gettime(CLOCK_REALTIME, &rtclk_stop_time);
delta_t(&rtclk_stop_time, &rtclk_start_time, &rtclk_dt);
```

Figure 12: Sequence of Sem_post()

In the snippet above, we notice that after 50msec, we abort Fib20 thread since it had a deadline of 50msec. After that, its semaphore value is increased again and it is free to run – after the higher priority Fib10 thread. We run this entire sequence for 100msec – this represents the LCM of the two deadline periods. If the code runs properly over this LCM, it is guaranteed to meet its deadlines subsequently too.

The code output observed is as below: -



```
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Fib10 priority = 98 and time stamp 10.797375 msec
Fib10 priority = 98 and time stamp 30.640500 msec
Fib20 priority = 97 and time stamp 40.109750 msec
Fib10 priority = 98 and time stamp 50.841959 msec
Fib10 priority = 98 and time stamp 70.855584 msec
Fib20 priority = 97 and time stamp 80.286000 msec
Fib10 priority = 98 and time stamp 91.021834 msec
Test Conducted over 101.395625 msec
TEST COMPLETE
debian@beaglebone:~/Desktop$ 
```

Figure 13: Output obtained

From the code execution, we observe that thread1 completes execution at roughly 10msec (before deadline of 20msec), 30msec (before deadline of 40msec), 50msec (before deadline of 60msec), 7msec (before deadline of 80msec) and 91msec (before deadline of 100msec). Also, thread2 completes execution at 40msec (before deadline of 50msec) and 80msec (before deadline of 100msec). There is an idle period of 9 msec from timestamp 91 to 100. Hence, all services are processed before the deadline over the LCM and also the CPU utilization is 90%.

Description of challenges and test/prototype work:

- Directly translating code from VxWorks to the Linux environment was a bit challenging – mainly due to the difference in the philosophies and style of coding.
- Understanding the exact sequence in which semaphores needed to be released and locked was essential. Using debugging methods such as printf(), I kept on playing with the sequence of sleep, semaphore release and lock till I got it correct.
- Even having used a major part of the code from the report, getting the 10msec and 20msec time periods correct was challenging, mainly due to the differences in hardware used. I ran the code code once to obtain an initial time for given parameter of number of iterations. I used this information to calculate number of iterations required to generate a delay of 10ms and 20ms
- Through this exercise we knowledge of threads and their functioning and how they work simultaneously in tandem. We were able to replicate the LCM invariant schedule implemented in VxWorks, in Linux. And obtained results similar to the analysis.