ECEN5623, Real-Time Systems:
Exercise #5 – Cyclic Executives in FreeRTOS and Linux

Submission By- Sean Duffy, Sarthak Jain, Smitha Bhaskar

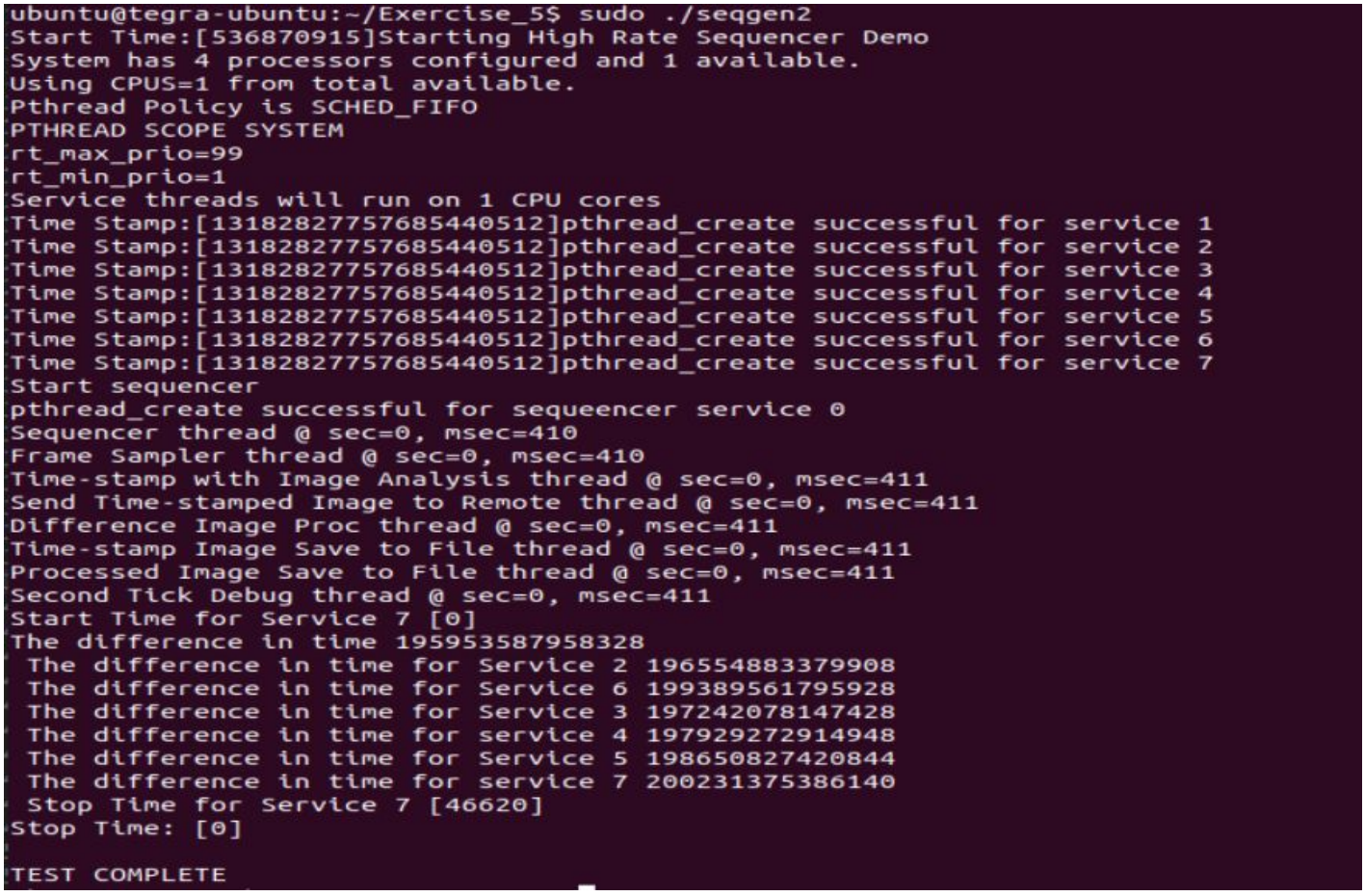
Boards Used: Jetson(Linux), Tiva(FreeRTOS)

1. Download seqgen.c and seqgen2x.c and build them in Linux on the Altera DE1-SOC, Raspberry Pi or Jetson board and execute the code. Describe what it is doing and make sure you understand how to use it to structure an embedded system. Determine the worst case execution time (WCET) for each service by printing or logging timestamps between two points in your code or by use of a profiling tool. Determine D, T, and C for each service and create an RM schedule in Cheddar using your WCET estimates. Calculate the % CPU utilization for this system.

   Sequence Generic is a program which defines 7 different set of services which run at different rates. The rate monotonic policy used for scheduling the services is based on their respective frequency of execution. Threads are defined for each service which are differentiated based on their "pthread_attr_setschedparam()" function. This function defines the rate-monotonic scheduling parameter for each service. Upon creation of threads, to create a sequence of calling the different service tasks, semaphores are defined for each service. These semaphores are used for synchronization. As the release of semaphores define the execution of the services, a sequencer has been implemented. In the sequencer, the sequencer count variable(seqCnt) is periodically checked in a modulo operation to post the semaphore which goes on to execute the respective service.
   Service 1 runs at 3 Hz and buffers 3 images per second. Service 2 runs at 1 Hz and creates a time-stamp middle sample image with cvPutText. Service 3 runs at 0.5 Hz and measures the difference between current and previous time stamped images. Service 4 runs at 1 Hz and is used to save time stamped image with cvSaveImage or write() functions. Service 5 runs at 0.5 Hz, is used to save the difference in image with cvSaveImage or write() functions. Service 6 runs at 1 Hz and writes the current time-stamped image to TCP socket server. Service 7 runs at 0.1 Hz and creates a syslog the time for debugging. Sequencer runs at the rate of 30 Hz and releases the semaphores.

   Timestamps have been added to indicate the start time and stop time of each service. The difference gives the deadline and time period information of each service . By determining the deadline (D), time-period(T), Capacity(C), we can estimate the WCET for each service. The CPU utilization is the summation of the ratio of Capacity and time-period for each service.

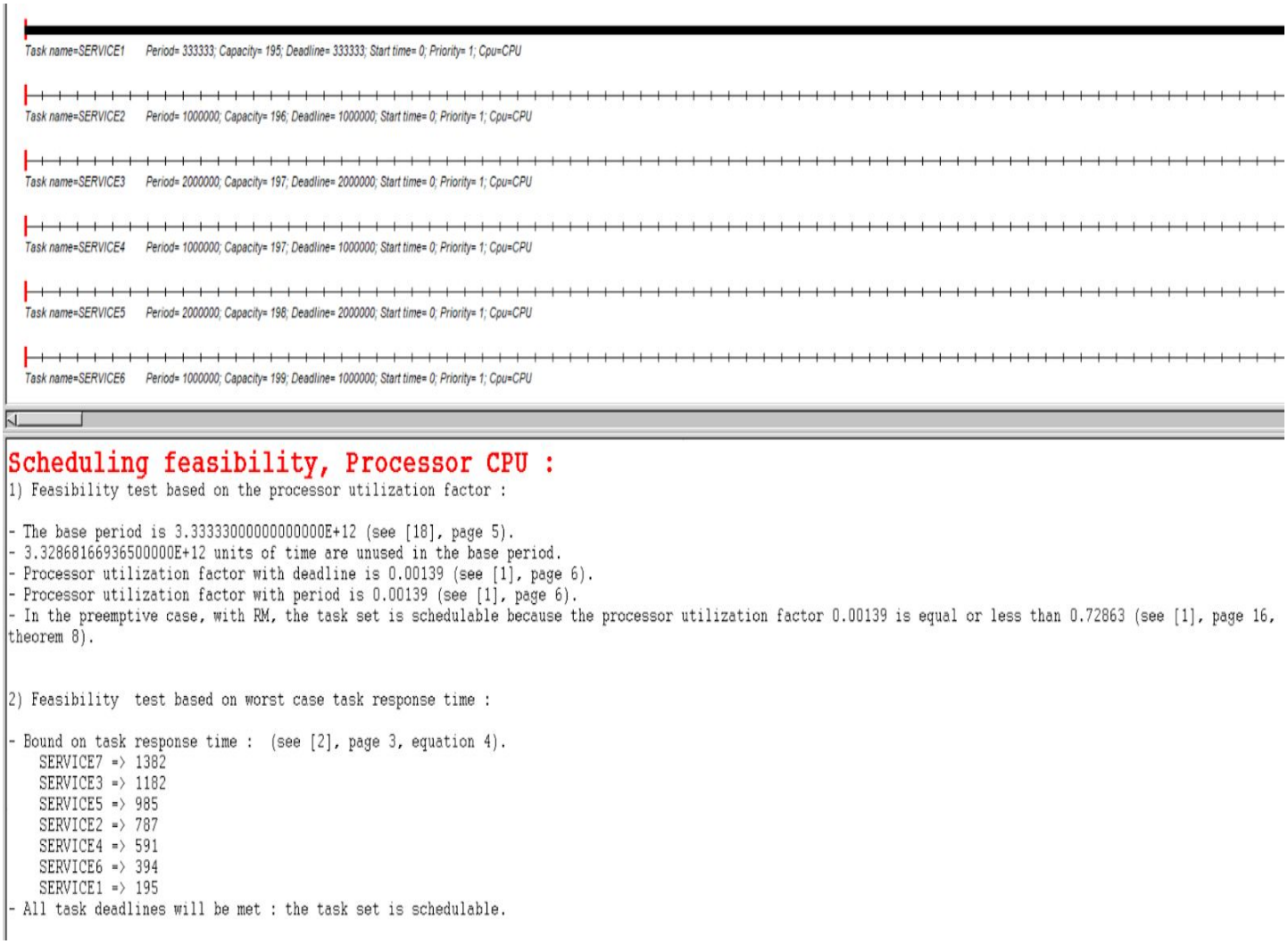
Screenshot of executing seqgen on Jetson



Tabular listing of Di,Ti,Ci for seqgen

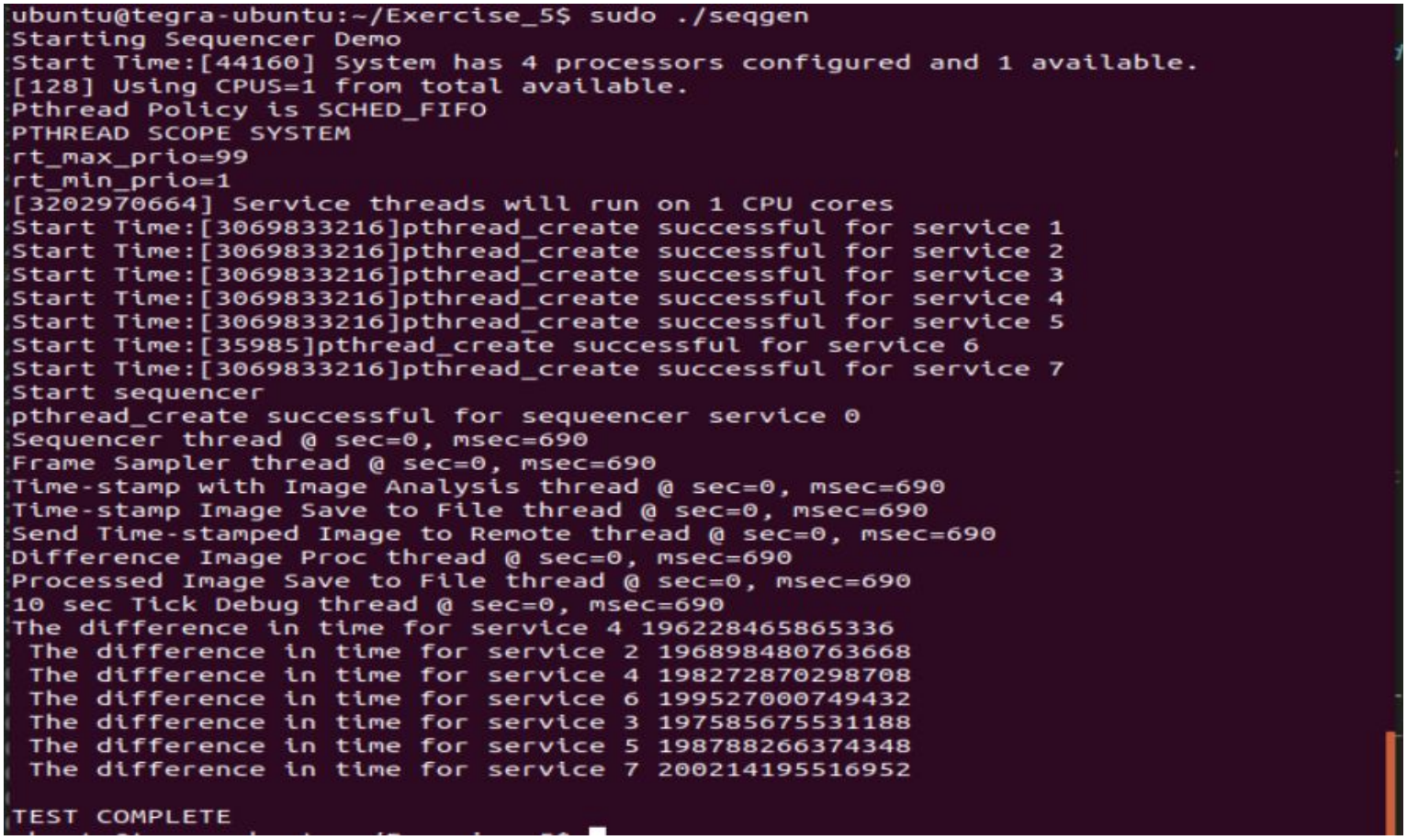
Task	Deadline(D_i)	Time-Period(T_i)	Capacity(C_i)
Service 1	333.33 ms	333.33 ms	.195us
Service 2	1000 ms	1000 ms	.196us
Service 3	2000 ms	2000 ms	.197us
Service 4	1000 ms	1000 ms	.197us
Service 5	2000 ms	2000 ms	.198us
Service 6	1000 ms	1000 ms	.199us
Service 7	10000 ms	10000 ms	.200us

The cheddar software was used for simulations.



Cheddar simulation for seqgen

Screenshot of executing seqgen2 on Jetson



Tabular listing of Di,Ti,Ci for seqgen

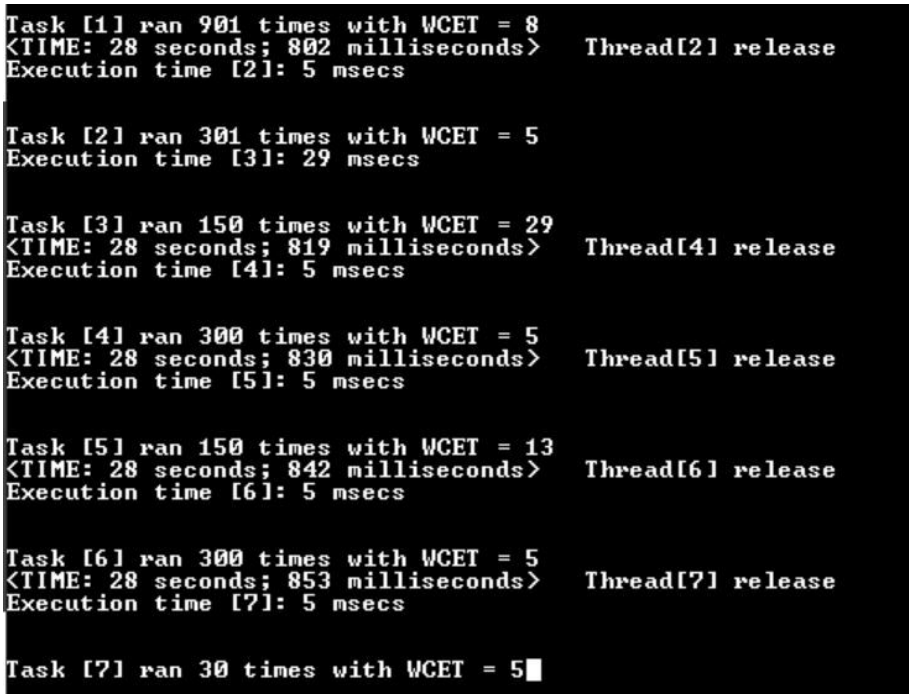
TASK	Deadline(D_i)	Time-Period(T_i)	Capacity(C_i)
Service 1	333.33 ms	333.33 ms	.196s
Service 2	1000 ms	1000 ms	.196us
Service 3	2000 ms	2000 ms	.197us
Service 4	1000 ms	1000 ms	.198us
Service 5	2000 ms	2000 ms	.198us
Service 6	1000 ms	1000 ms	.199us
Service 7	10000 ms	10000 ms	.200us

2) Revise seqgen.c and seqgen2x.c to run under FreeRTOS on the DE1-SoC or TIVA board, by making each service a FreeRTOS task. Use the associated startup file in place of the existing startup file in FreeRTOS. Use an ISR driven by the PIT hardware timer to release each task at the given rate (you could even put the sequencer in the ISR). Build and execute the new code. Determine the worst case execution time (WCET) for each service by printing or logging timestamps between two points in your code or by use of a profiling tool. Determine D, T, and C for each service and create an RM schedule in Cheddar using your WCET estimates. Calculate the % CPU utilization for this system. Compare this with the results you achieved under Linux in (1).

A logic similar to the one used by Prof. Siewert to write the sequence generator in C was used to replicate the sequence generator in FreeRTOS. 7 tasks are created, and instead of creating a separate task for the sequencer, the sequence is performed in the timer handler itself. In case of 'seqgen.c', the timer handler is called at a frequency of 33.33 ms, and respective frequencies for the 7 tasks are as used in Prof. Siewert's code itself. A semaphore is used for the synchronization of each task, and as the request time for a task approached, the timer handler releases the semaphore for that task. The task begins executing, and waits in a while on the semaphore to be released by the timer handler. Once semaphore is released, the time is recorded in a 'TickType_t' variable, displayed, execution time is recorded, and task loops back to wait on the semaphore again. This same process is repeated in each task.
A screenshot of tasks executing at their respective frequencies along with a timestamp is attached below:

```
<TIME: 27 seconds; 59 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 27 seconds; 389 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 27 seconds; 719 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 27 seconds; 720 milliseconds>   Thread[3] release
Execution time [3]: 11 msecs

<TIME: 27 seconds; 720 milliseconds>   Thread[5] release
Execution time [5]: 19 msecs

<TIME: 27 seconds; 720 milliseconds>   Thread[2] release
Execution time [2]: 27 msecs

<TIME: 27 seconds; 720 milliseconds>   Thread[6] release
Execution time [6]: 34 msecs

<TIME: 27 seconds; 720 milliseconds>   Thread[4] release
Execution time [4]: 42 msecs

<TIME: 28 seconds; 49 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 28 seconds; 379 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 28 seconds; 709 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 28 seconds; 710 milliseconds>   Thread[4] release
Execution time [4]: 11 msecs

<TIME: 28 seconds; 710 milliseconds>   Thread[2] release
Execution time [2]: 19 msecs

<TIME: 28 seconds; 710 milliseconds>   Thread[6] release
Execution time [6]: 26 msecs

<TIME: 29 seconds; 39 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 29 seconds; 369 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs
```

Execution of code sequence generator with timestamps

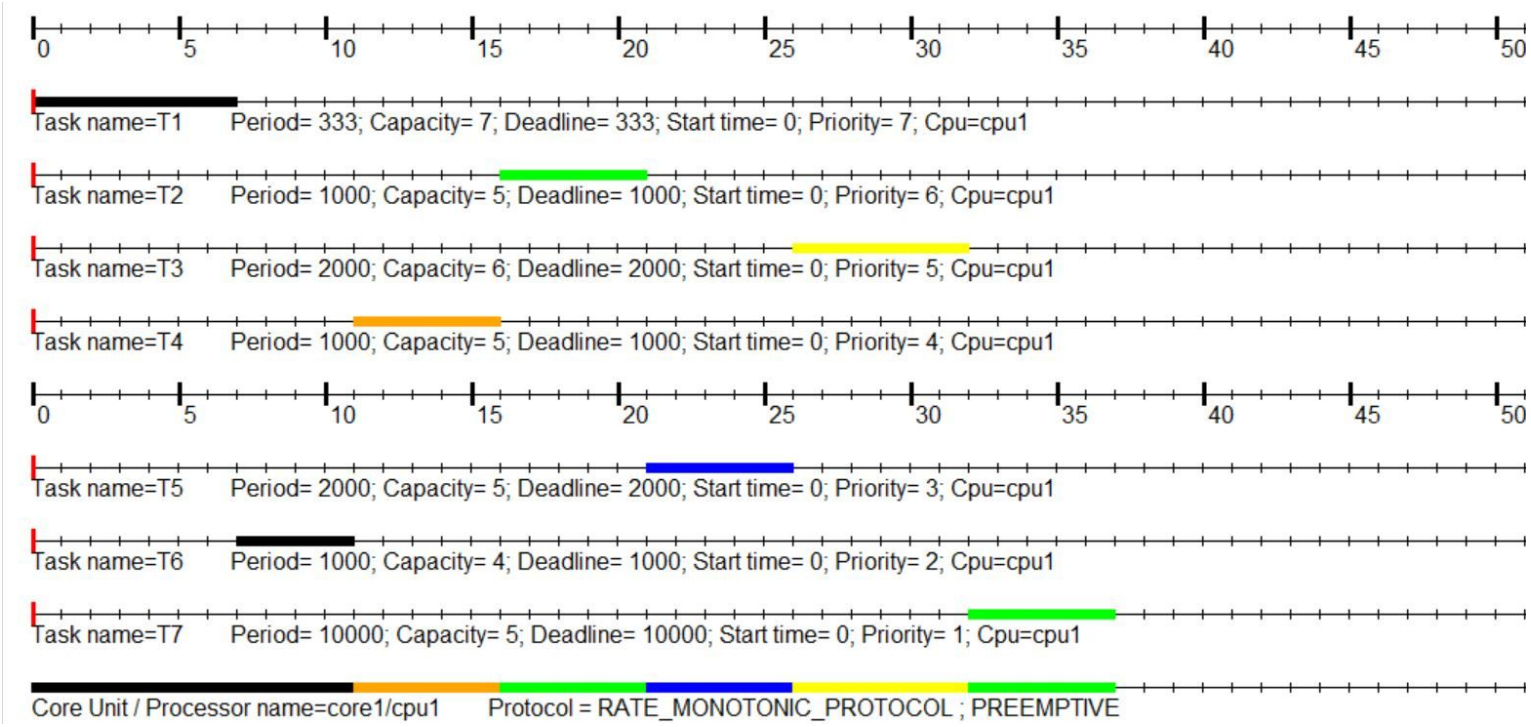


Execution of code sequence generator with worst case execution times (in ms)

Tabular listing of D, T and C for seqgen on FreeRTOS

Task	Deadline(D_i)	Time-Period(T_i)	Capacity(C_i)
Service 1	333.33 ms	333.33 ms	7 ms
Service 2	1000 ms	1000 ms	5 ms
Service 3	2000 ms	2000 ms	6 ms
Service 4	1000 ms	1000 ms	5 ms
Service 5	2000 ms	2000 ms	5 ms
Service 6	1000 ms	1000 ms	4 ms
Service 7	10000 ms	10000 ms	5 ms

Cheddar schedule of seqgen



Feasibility of seqgen

Scheduling feasibility, Processor cpu1 :
1) Feasibility test based on the processor utilization factor :

- The hyperperiod is 3330000 (see [18], page 5).
- 3193400 units of time are unused in the hyperperiod.
- Processor utilization factor with deadline is 0.04102 (see [1], page 6).
- Processor utilization factor with period is 0.04102 (see [1], page 6).
- In the preemptive case, with RM, the task set is schedulable because the processor utilization factor 0.04102 is equal or less than 0.72863 (see [1], page 16, theorem 8).

2) Feasibility test based on worst case response time for periodic tasks :

- Worst Case task response time : (see [2], page 3, equation 4).
  T7 => 37
  T3 => 32
  T5 => 26
  T2 => 21
  T4 => 16
  T6 => 11
  T1 => 7
- All task deadlines will be met : the task set is schedulable.

As we can see from the screenshots above, Cheddar proclaims sequence generator on FreeRTOS feasible as well, with a total CPU utilization of 4.1%.

```
<TIME: 28 seconds; 513 milliseconds>   Thread[1] release
Execution time [1]: 27 msecs

<TIME: 28 seconds; 543 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 28 seconds; 575 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 28 seconds; 608 milliseconds>   Thread[2] release
Execution time [2]: 3 msecs

<TIME: 28 seconds; 609 milliseconds>   Thread[4] release
Execution time [4]: 10 msecs

<TIME: 28 seconds; 609 milliseconds>   Thread[6] release
Execution time [6]: 18 msecs

<TIME: 28 seconds; 609 milliseconds>   Thread[3] release
Execution time [3]: 25 msecs

<TIME: 28 seconds; 609 milliseconds>   Thread[1] release
Execution time [1]: 33 msecs

<TIME: 28 seconds; 645 milliseconds>   Thread[1] release
Execution time [1]: 5 msecs

<TIME: 28 seconds; 609 milliseconds>   Thread[5] release
Execution time [5]: 49 msecs

<TIME: 28 seconds; 671 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 28 seconds; 704 milliseconds>   Thread[2] release
Execution time [2]: 3 msecs

<TIME: 28 seconds; 705 milliseconds>   Thread[6] release
Execution time [6]: 10 msecs

<TIME: 28 seconds; 705 milliseconds>   Thread[4] release
Execution time [4]: 18 msecs

<TIME: 28 seconds; 705 milliseconds>   Thread[1] release
Execution time [1]: 25 msecs

<TIME: 28 seconds; 735 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs

<TIME: 28 seconds; 767 milliseconds>   Thread[1] release
Execution time [1]: 4 msecs
```

Execution of code sequence generator 2 with timestamps

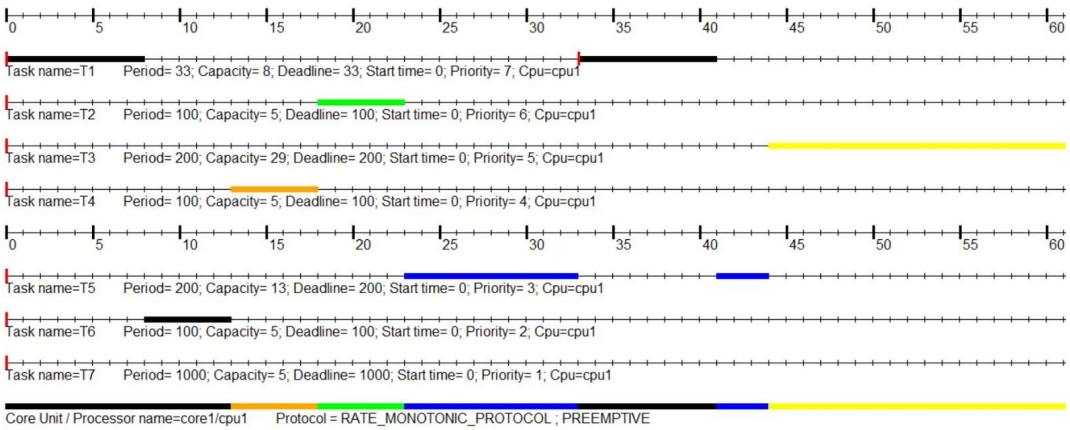
```
Task [1] ran 91 times with WCET = 7
<TIME: 29 seconds; 746 milliseconds>   Thread[2] release
Execution time [2]: 5 msecs


Task [2] ran 31 times with WCET = 5
<TIME: 29 seconds; 757 milliseconds>   Thread[3] release
Execution time [3]: 5 msecs


Task [3] ran 16 times with WCET = 5
<TIME: 29 seconds; 768 milliseconds>   Thread[4] release
Execution time [4]: 5 msecs


Task [4] ran 31 times with WCET = 5
Execution time [5]: 50 msecs


Task [5] ran 15 times with WCET = 50
<TIME: 29 seconds; 786 milliseconds>   Thread[6] release
Execution time [6]: 5 msecs


Task [6] ran 30 times with WCET = 5
<TIME: 29 seconds; 797 milliseconds>   Thread[7] release
Execution time [7]: 5 msecs


Task [7] ran 3 times with WCET = 5
```

Execution of code sequence generator with worst case execution times (in ms)

Tabular listing of D, T and C for seqgen 2 on FreeRTOS

Task	Deadline(D_i)	Time-Period(T_i)	Capacity(C_i)
Service 1	33.33 ms	33.33 ms	8 ms
Service 2	100 ms	100 ms	5 ms
Service 3	200 ms	200 ms	29 ms
Service 4	100 ms	100 ms	5 ms
Service 5	200 ms	200 ms	13 ms
Service 6	100 ms	100 ms	5 ms
Service 7	1000 ms	1000 ms	5 ms

Cheddar schedule of seqgen 2



Feasibility of seqgen 2

```
Scheduling feasibility, Processor cpu1 :
1) Feasibility test based on the processor utilization factor :

- The hyperperiod is 33000 (see [18], page 5).
- 12955 units of time are unused in the hyperperiod.
- Processor utilization factor with deadline is 0.60742 (see [1], page 6).
- Processor utilization factor with period is 0.60742 (see [1], page 6).
- In the preemptive case, with RM, the task set is schedulable because the processor utilization factor 0.60742 is equal or less than 0.72863 (see [1], page 16, theorem 8).


2) Feasibility test based on worst case response time for periodic tasks :

- Worst Case task response time :  (see [2], page 3, equation 4).
   T7 => 86
   T3 => 81
   T5 => 44
   T2 => 23
   T4 => 18
   T6 => 13
   T1 => 8
- All task deadlines will be met : the task set is schedulable.
```

As we can see from the screenshots above, Cheddar proclaims sequence generator 2 on FreeRTOS feasible as well, with a total CPU utilization of 60.7%.
There are quite a few differences between executing the code on two different OSs. FIrstly, Linux OS treats these threads quite differently to how RTOS does. Linux, unless specified to do so, uses Completely Fair Scheduler to schedule the threads. In the case of our code, we have specified the scheduler to be FIFO. In this case, Linux allows the threads to run to completion, while also placing tasks in the order of FIFO on the ready queue. RTOS, on the other hand, is preemptible, runs threads to completion, and allows a higher priority thread to preempt a lower priority thread, even in the case of FIFO. In this way, FreeRTOS is a better OS to use for the purpose of real-time computing.

Another difference is that of the use of a hardware timer in FreeRTOS to schedule the tasks, whereas in Linux, we use a different thread to schedule the other threads.
Finally, it is a little more complex to detect the passage of time in FreeRTOS, as the only means of doing so is by the use of ticks, which have a resolution of 1 ms. It is difficult to detect the passage of nanoseconds, or even microseconds in FreeRTOS. This subtlety does not make much of a difference, as each task anyway has a runtime of more than a millisecond. On the other hand, Linux on the Jetson is executing a lot faster, with a runtime of roughly a few hundred nanoseconds.

3. Revise seqgen.c from both previous systems to increase the sequencer frequency and all service frequencies by a factor of 100 (3000 Hz). Build and execute the code under Linux and FreeRTOS on your target boards as before. For both operating systems determine the worst case execution time (WCET) for each service by printing or logging timestamps between two points in your code or by use of a profiling tool. Determine D, T, and C for each service and create an RM schedule in Cheddar using your WCET estimates. Calculate the % CPU utilization for these systems. Compare results between Linux and FreeRTOS in this higher-speed case.

Execution on the Jetson was far less complex than on the TIVA board, it having a higher clock rate. The sequencer frequency was simply increased by 100 times, and code was executed. Although execution time was almost similar to that of sequencer being run at a frequency of 30 Hz, the schedule is still proclaimed infeasible by the RM protocol when run on Cheddar.

Screenshot upon executing seqgen.c code at 3000Hz. This was done by modifying the timespec delay_time parameter.

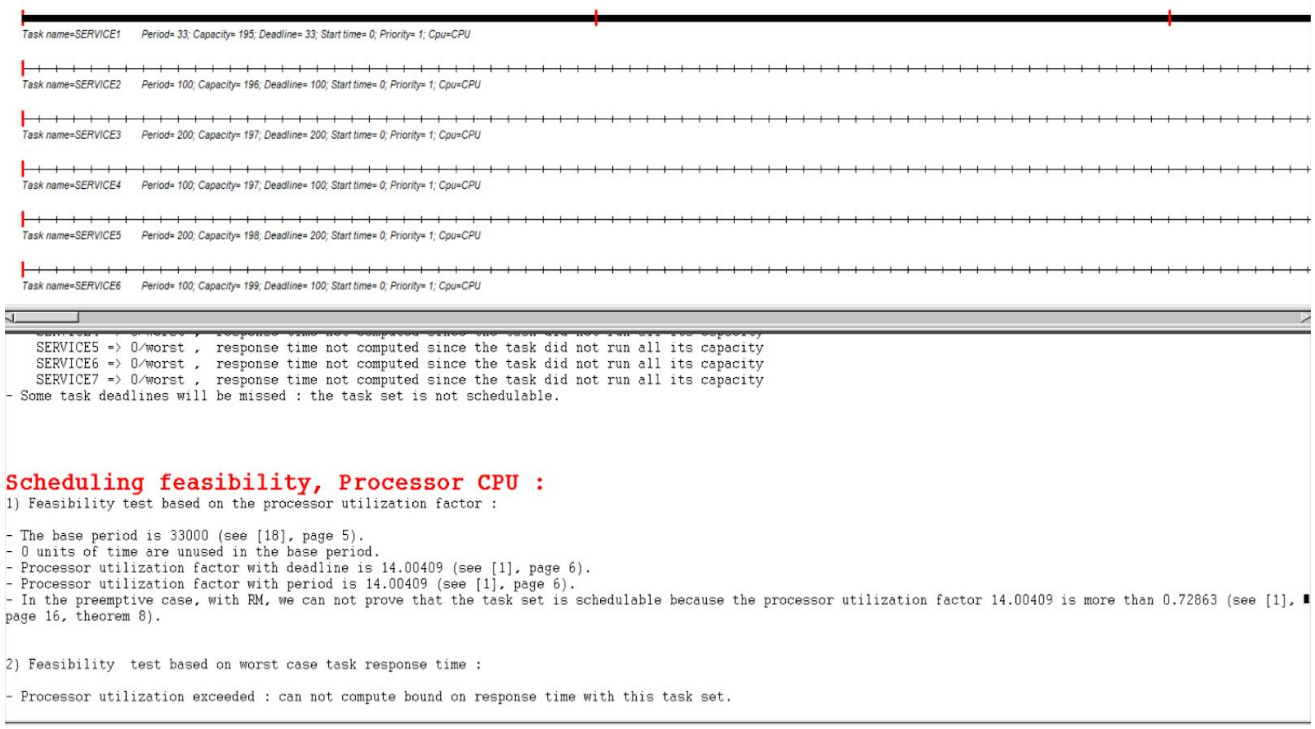
Execution of seqgen code at 3000Hz on Jetson

```
ubuntu@tegra-ubuntu:~/Exercise_5$ sudo ./seqgen
[sudo] password for ubuntu:
Starting Sequencer Demo
Start Time:[44160] System has 4 processors configured and 1 available.
[128] Using CPUS=1 from total available.
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
[3202503720] Service threads will run on 1 CPU cores
Start Time:[3069734912]pthread_create successful for service 1
Start Time:[3069734912]pthread_create successful for service 2
Start Time:[3069734912]pthread_create successful for service 3
Start Time:[3069734912]pthread_create successful for service 4
Start Time:[3069734912]pthread_create successful for service 5
Start Time:[35985]pthread_create successful for service 6
Start Time:[3069734912]pthread_create successful for service 7
Start sequencer
pthread_create successful for sequeencer service 0
Sequencer thread @ sec=0, msec=543
Frame Sampler thread @ sec=0, msec=544
Time-stamp with Image Analysis thread @ sec=0, msec=544
Time-stamp Image Save to File thread @ sec=0, msec=544
Send Time-stamped Image to Remote thread @ sec=0, msec=544
Difference Image Proc thread @ sec=0, msec=544
Processed Image Save to File thread @ sec=0, msec=544
10 sec Tick Debug thread @ sec=0, msec=544
The difference in time for service 4 196228465865336
 The difference in time for service 2 196898480763668
 The difference in time for service 4 198272870298708
 The difference in time for service 6 199527000749432
 The difference in time for service 3 197585675531188
 The difference in time for service 5 198788266374348
 The difference in time for service 7 200214195516952

TEST COMPLETE
```

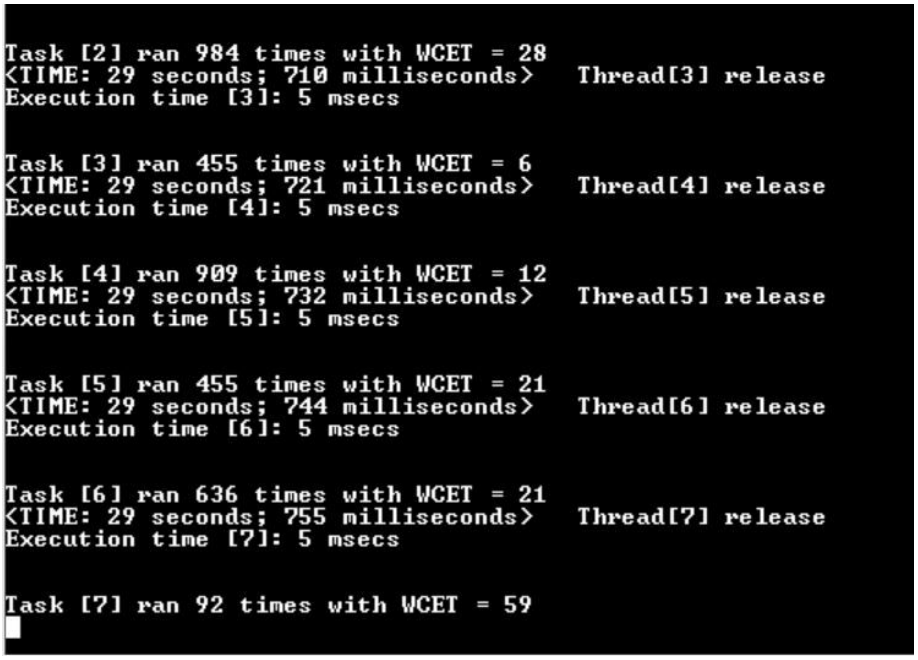
Tabular listing of Di,Ti,Ci for seqgen

TASK	Deadline(D_i)	Time-Period(T_i)	Capacity(C_i)
Service 1	0.33 ms	0.33 ms	.196us
Service 2	1 ms	1 ms	.196us
Service 3	2 ms	2 ms	.197us
Service 4	1 ms	1 ms	.198us
Service 5	2 ms	2 ms	.198us
Service 6	1 ms	1 ms	.199us
Service 7	10 ms	10 ms	.200us

Cheddar Simulation:



At this high frequency of 3 KHz, the schedule is bound to fail on the TIVA board, seeing as the request rate for task 1 occurs every 0.33 ms, and the board itself requires more time to execute the task (considering the UARTprintf statements) than its request period.
Keeping this in mind, I attempted to increase the scheduler frequency in order to find the point at which it fails. I found the frequency as 1KHz. An output screenshot showing the WCET with scheduler frequency 1KHz is attached below:



Due to task preemption, the WCETs rise quite a bit, since a higher priority task may preempt another in between, and increase its WCET.

TASK	Deadline(D_i)	Time-Period(T_i)	Capacity(C_i)
Service 1	10 ms	10 ms	1 ms
Service 2	30 ms	30 ms	28 ms
Service 3	60 ms	60 ms	6 ms
Service 4	30 ms	30 ms	12 ms
Service 5	60 ms	60 ms	21 ms
Service 6	30 ms	30 ms	21 ms
Service 7	300 ms	300 ms	59 ms

A comparison between Linux and FreeRTOS here is that the code is unable to execute at 3KHz on FreeRTOS, but can do so on Linux. This is due to the large amount of time taken by ‘UARTprintf’ to execute, whereas the faster alternative in Linux, ‘syslog’ or even ‘printf’ takes a few milliseconds.Due to this, I have checked the maximum frequency at which the TIVA can execute the given codes, which comes to 1KHz.