EXERCISE 5

BY

Amreeta Sengupta and Ayush Dhoot

04/09/2019

Assignment has been performed on Raspberry Pi.

1) [20 points] 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.

Solution:

Explanation:

The code runs a sequencer with 7 services. The sequencer runs at 30 Hz and is given the highest priority. The priority is assigned along the basis of Rate Monotonic which defines that higher priority will be given to a service with higher frequency. Accordingly, all seven services are assigned priorities. For any embedded system, there are services which are hard real-time or soft real-time. From this example, we learn how to properly schedule these services.

In the main loop, all the initialization takes place such as printing out the total configured and available CPU cores, selecting one of the cores from the available ones. Next step is to initialize the semaphores defined for each of the services and assign priority according to the Rate Monotonic. There is also a structure called 'threadParams_t' defined for storing the thread IDs and the respective periods. Then there is pthreads created for each of the seven services and one for the sequencer. All the pthreads created are joined and run to completion according to the sequencer.

The sequencer loop acts like a cyclic executive and has the highest priority defined. The sequencer loop should run at a frequency of 30 Hz, hence the sleep time is set at 33.33 msecs. Just after the nanosleep function is executed, we check for any residual time left and make sure

to increase the sequence count. Now according to the frequency of the services, we make an algorithm to post the semaphores to maintain the schedule. The semaphores get posted and then again, the sequencer goes to sleep. When the sequencer is in sleep, the semaphores which are posted runs to completion. This cycle goes on and on for the sequencer period defined (here it is 900). This is how a cyclic executive works wherein there is a loop control structure which executes multiple periodic services on a single CPU.

The services are defined with an infinite loop. In the loop there is a sem_wait() function and we keep a counter for each time the service is run. We also have time-stamps defined all over in the code to have a better understanding of the execution time.

This code can be used in Hard Real-Time systems where there are periodic tasks. The architecture used in this code was of a Cyclic Executive where there is a main loop which gives function calls to each of the services within the period of its own. We can even extend this into a Main+ISR architecture with few changes which can even handle aperiodic tasks.

The second code is same as the first one, the only difference is that the period of the sequencer is doubled to 60 Hz and the period of all the services are increased by a factor of 10.

SEQGEN:

(Code Execution with Timestamp, WCET calculation, Cheddar Analysis, CPU utilization)

```
Starting Sequencer Demo
System has 4 processors configured and 4 available.
Using CPUS=1 from total available.
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE PROCESS
rt_max_prio=99
rt_min_prio=1
Service threads will run on 1 CPU cores
pthread_create successful for service 1
pthread_create successful for service 2
Frame Sampler thread @ sec=0, msec=408
pthread_create successful for service 3
pthread_create successful for service 4
Time-stamp with Image Analysis thread @ sec=0, msec=408
Time-stamp Image Save to File thread @ sec=0, msec=408
Difference Image Proc thread @ sec=0, msec=408
pthread_create successful for service 5
pthread_create successful for service 6
pthread_create successful for service 7
Start sequencer
Send Time-stamped Image to Remote thread @ sec=0, msec=409
pthread_create successful for sequeencer service 0
Sequencer thread @ sec=0, msec=409
Processed Image Save to File thread @ sec=0, msec=409
10 sec Tick Debug thread @ sec=0, msec=409
Thread idx=1 Count = 1 ,timestamp 0 sec, 0 msec (19 microsec), ((19791 nanosec))

Thread idx=1 Count = 2 ,timestamp 0 sec, 0 msec (22 microsec), ((22084 nanosec))

Thread idx=1 Count = 3 ,timestamp 0 sec, 0 msec (41 microsec), ((41510 nanosec))

Thread idx=2 Count = 1 , timestamp 0 sec, 0 msec (40 microsec), ((40312 nanosec))

Thread idx=4 Count = 1 ,timestamp 0 sec, 0 msec (38 microsec), ((38698 nanosec))

Thread idx=6 Count = 1 ,timestamp 0 sec, 0 msec (36 microsec), ((36042 nanosec))
```

Figure 1 : Execution of seqgen.c along with timestamps

```
TEST COMPLETE
Thread idx=1 ,WCET timestamp 0 sec, 0 msec (61 microsec), ((61979 nanosec))

Thread idx=2 ,WCET timestamp 0 sec, 0 msec (57 microsec), ((57291 nanosec))

Thread idx=3 ,WCET timestamp 0 sec, 0 msec (38 microsec), ((38906 nanosec))

Thread idx=4 ,WCET timestamp 0 sec, 0 msec (43 microsec), ((43906 nanosec))

Thread idx=5 ,WCET timestamp 0 sec, 0 msec (38 microsec), ((38698 nanosec))

Thread idx=6 ,WCET timestamp 0 sec, 0 msec (41 microsec), ((41615 nanosec))

Thread idx=7 ,WCET timestamp 0 sec, 0 msec (243 microsec), ((243020 nanosec))
```

Figure 2 : WCET calculation for seqgen.c

D, T and C Table:

(Time in ms)	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
D	333.33	1000	2000	1000	2000	1000	1000
T	333.33	1000	2000	1000	2000	1000	1000
C	0.061	0.057	0.039	0.044	0.039	0.041	0.243

Table 1 : D, C and T for seqgen.c

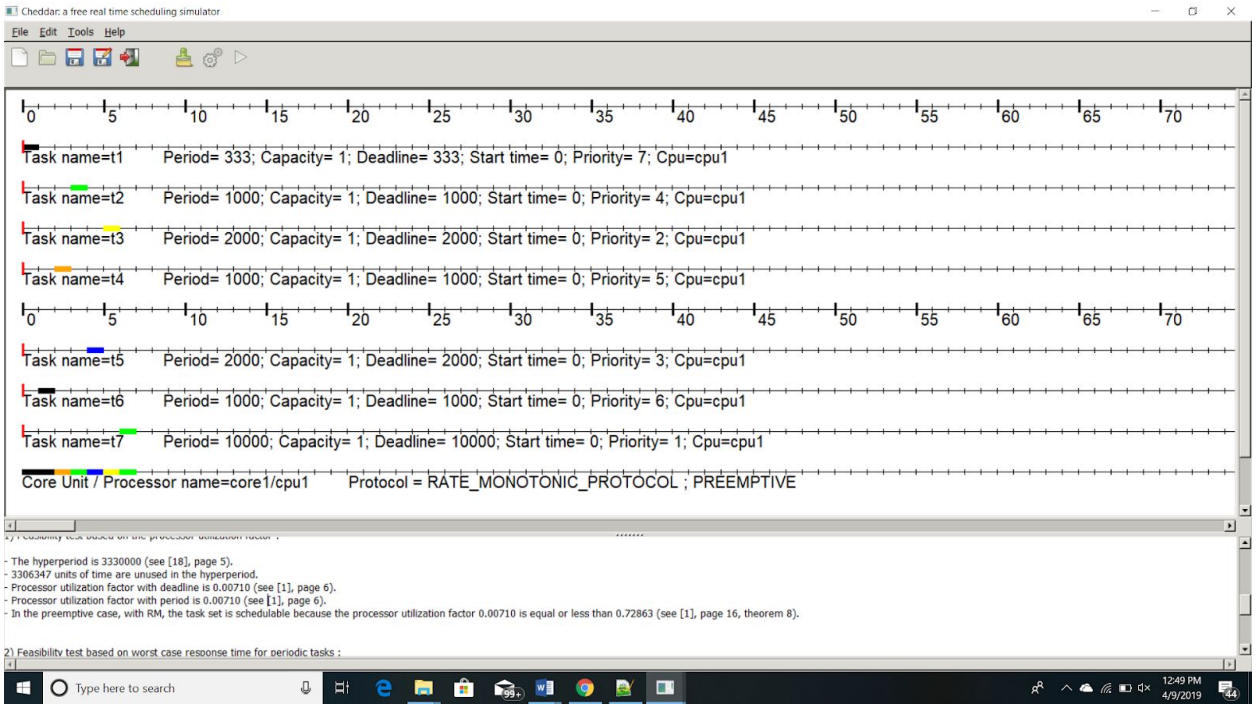


Figure 3 : Simulation in Cheddar

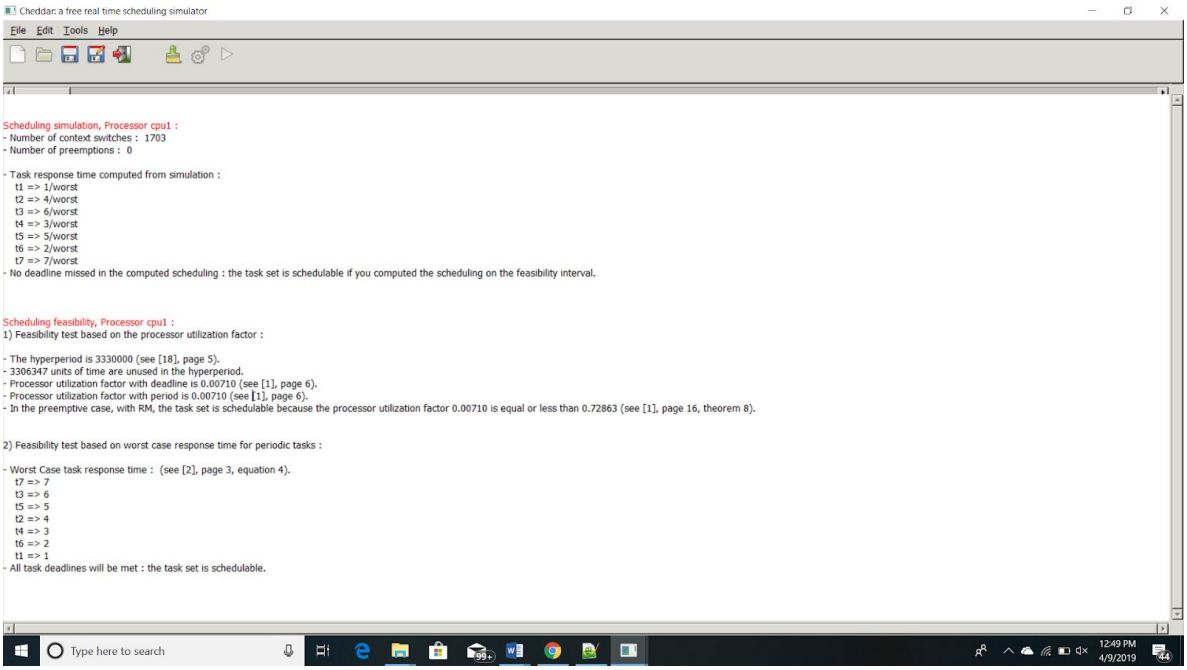


Figure 4 : Scheduling test in Cheddar

T	333.33	1000	2000	1000	2000	1000	1000		
C	0.061	0.057	0.039	0.044	0.039	0.041	0.243		
C/T	0.000183	0.000057	1.95E-05	0.000044	1.95E-05	0.000041	0.000243	CPU Utilization =	0.000607

Figure 5 : CPU Utilization for seqgen.c

SEQGEN2X:

(Code Execution with Timestamp, WCET calculation, Cheddar Analysis, CPU utilization)

```
Starting High Rate Sequencer Demo
System has 4 processors configured and 4 available.
Using CPUS=1 from total available.
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE PROCESS
rt_max_prio=99
rt_min_prio=1
Service threads will run on 1 CPU cores
pthread_create successful for service 1
pthread_create successful for service 2
pthread_create successful for service 3
Frame Sampler thread @ sec=0, msec=361
Time-stamp with Image Analysis thread @ sec=0, msec=362
pthread_create successful for service 4
Difference Image Proc thread @ sec=0, msec=362
pthread_create successful for service 5
pthread_create successful for service 6
Time-stamp Image Save to File thread @ sec=0, msec=362
Processed Image Save to File thread @ sec=0, msec=362
pthread_create successful for service 7
Start sequencer
Send Time-stamped Image to Remote thread @ sec=0, msec=362
pthread_create successful for sequeencer service 0
Second Tick Debug thread @ sec=0, msec=363
Sequencer thread @ sec=0, msec=363
Thread idx=1 Count = 1 ,timestamp 0 sec, 0 msec (47 microsec), ((47135 nanosec))

Thread idx=1 Count = 2 ,timestamp 0 sec, 0 msec (23 microsec), ((23854 nanosec))

Thread idx=1 Count = 3 ,timestamp 0 sec, 0 msec (2 microsec), ((2135 nanosec))

Thread idx=2 Count = 1 , timestamp 0 sec, 0 msec (9 microsec), ((9740 nanosec))

Thread idx=6 Count = 1 ,timestamp 0 sec, 0 msec (20 microsec), ((20885 nanosec))

Thread idx=4 Count = 1 ,timestamp 0 sec, 0 msec (18 microsec), ((18073 nanosec))

Thread idx=1 Count = 4 ,timestamp 0 sec, 0 msec (24 microsec), ((24115 nanosec))
```

Figure 6 : Execution of seqgen2x.c along with timestamps

```
TEST COMPLETE
Thread idx=1 ,WCET timestamp 0 sec, 0 msec (205 microsec), ((205520 nanosec))

Thread idx=2 ,WCET timestamp 0 sec, 0 msec (201 microsec), ((201301 nanosec))

Thread idx=3 ,WCET timestamp 0 sec, 0 msec (171 microsec), ((171562 nanosec))

Thread idx=4 ,WCET timestamp 0 sec, 0 msec (258 microsec), ((258593 nanosec))

Thread idx=5 ,WCET timestamp 0 sec, 0 msec (173 microsec), ((173437 nanosec))

Thread idx=6 ,WCET timestamp 0 sec, 0 msec (230 microsec), ((230051 nanosec))

Thread idx=7 ,WCET timestamp 0 sec, 0 msec (175 microsec), ((175937 nanosec))
```

Figure 7 : WCET calculation for seqgen2x.c

D, T and C Table:

(ms)	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
D	33.33	100	200	100	200	100	100
T	33.33	100	200	100	200	100	100
C	0.205	0.201	0.171	0.258	0.173	0.23	0.175

Table 2 : D, C and T for seqgen2x.c

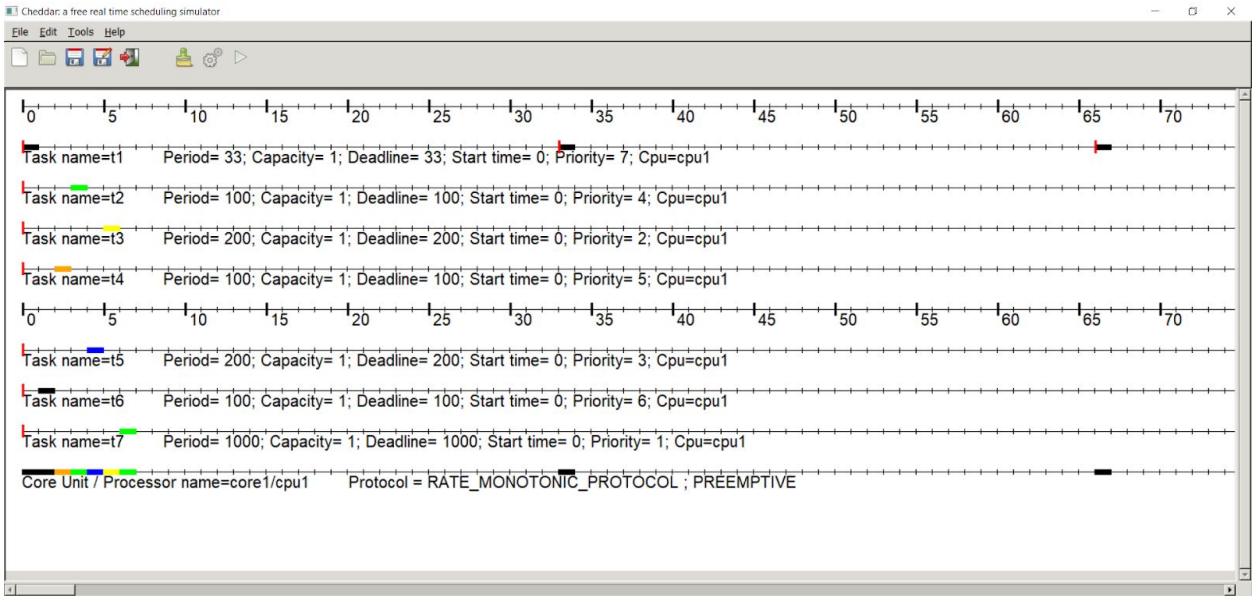


Figure 8 : Simulation in Cheddar

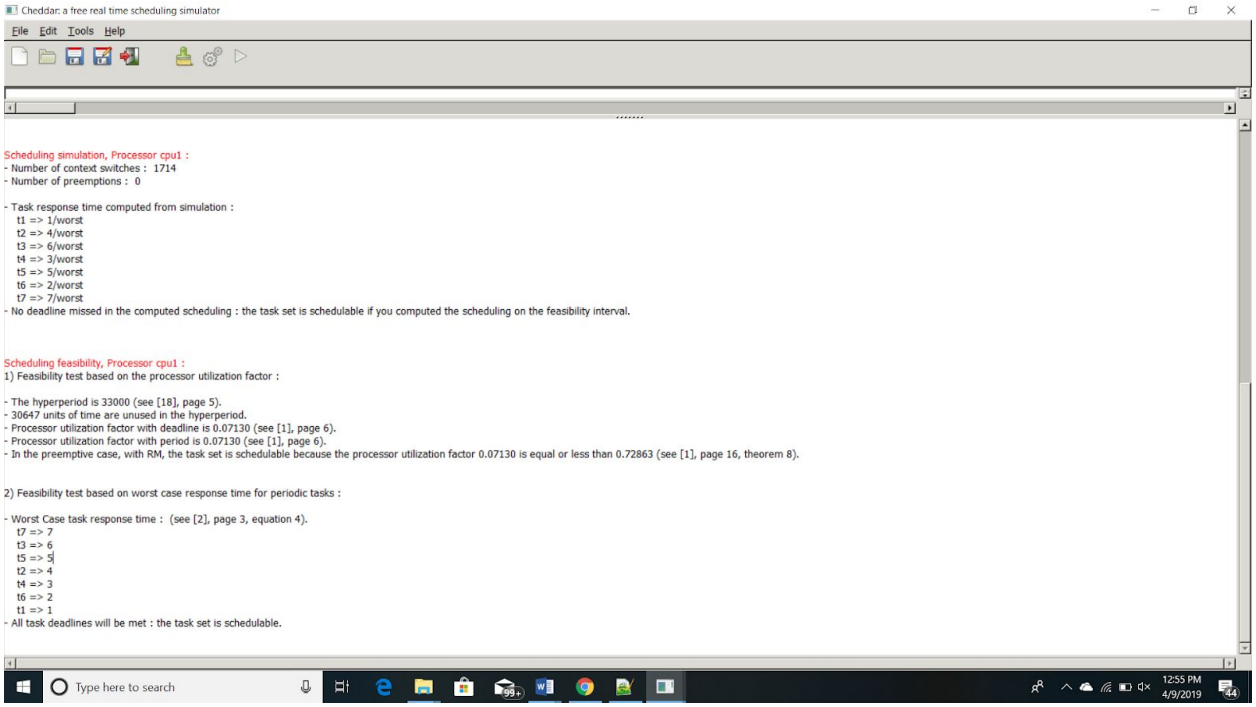


Figure 9 : Scheduling test in Cheddar

T	33.33	100	200	100	200	100	100		
C	0.205	0.201	0.171	0.258	0.173	0.23	0.175		
C/T	0.006151	0.00201	0.000855	0.00258	0.000865	0.0023	0.00175	CPU Utilization =	0.0165106

Figure 10 : CPU Utilization for seqgen2x.c

2) [40 points] 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).

Solution:

Seqgen.c

Explanation and execution of code on target

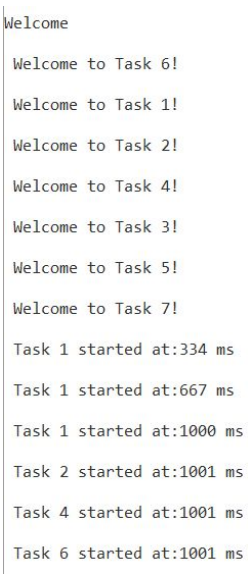


Figure 11 : Execution of seqgen.c code on FreeRTOS

In this code, seven tasks are created and these tasks are scheduled using a sequencer. A hardware timer is used and the sequencer is implemented in the ISR of this timer. The priority of each task is assigned on the basis of Rate Monotonic Policy which states that higher priority will be given to a task with higher frequency. Accordingly, all seven services are assigned priorities. Each task has a synthetic load which prints the start time of the task and the WCET of each task is printed. A variable is maintained for each task which updates according to the maximum execution time for the calculation of WCET. Semaphores are used for task synchronization. xSemaphoreGiveFromISR is used in the timer ISR for each task and xSemaphoreTake is used by each task post which the synthetic load is executed. The timer ISR is loaded such that it occurs at

the clock frequency (50000000 MHz) divided by the sequencer frequency that is required which is 30Hz in this case.The timer ISR maintains a variable called seqCnt and based on this it releases the semaphore according to respective request time for each task. When this variable count reaches 900, each task is aborted and the test completes. We use mutexes for the synthetic load so that it is not preempted by any other task which would result in the timestamp not being printed correctly as all the tasks use the same physical UART. Due to the use of these mutexes, any higher priority task is unable to preempt a lower priority task which results in priority inversion. This causes variation in the WCET of the tasks with same priority.

Timestamps and WCET estimates

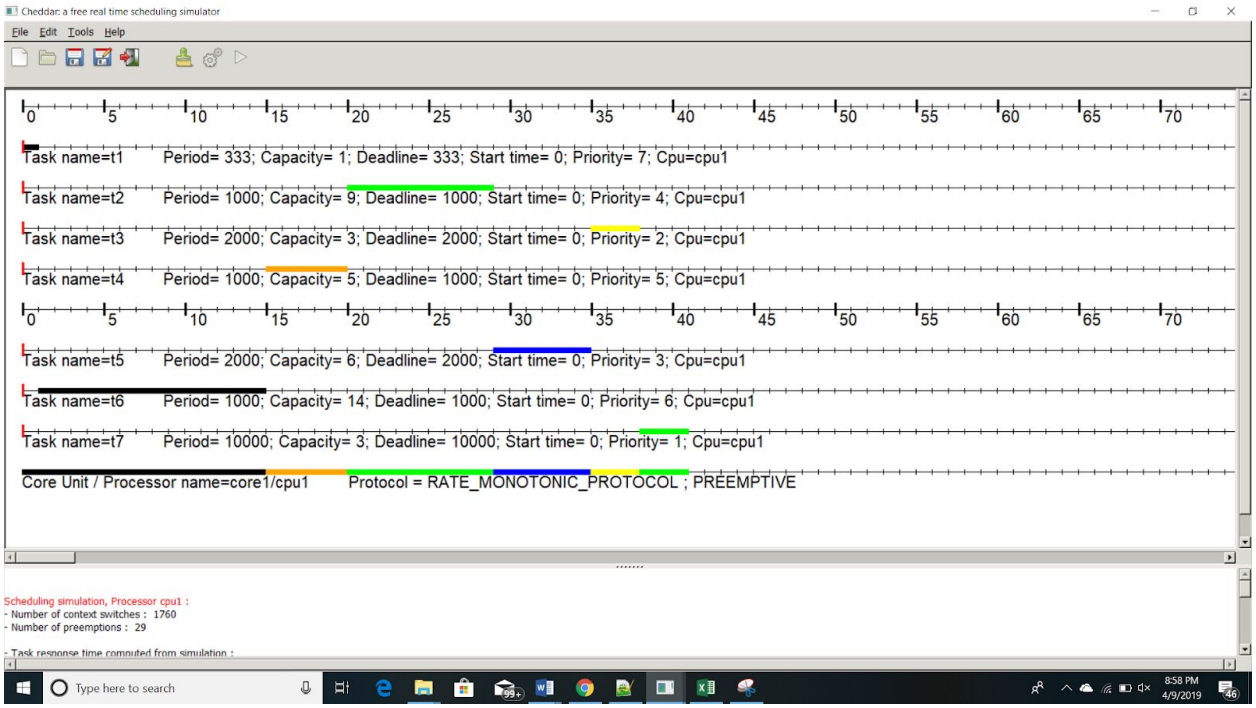
Figure 12 : Printing timestamps and WCET calculation

D, C, and T Table for all services

(ms)	S1	S2	S3	S4	S5	S6	S7
D	333.33	1000	2000	1000	2000	1000	10000
T	333.33	1000	2000	1000	2000	1000	10000
C	1	9	3	5	6	14	3

Table 3 : D, C and T for seqgen.c on FreeRTOS

Cheddar schedule and % CPU utilization



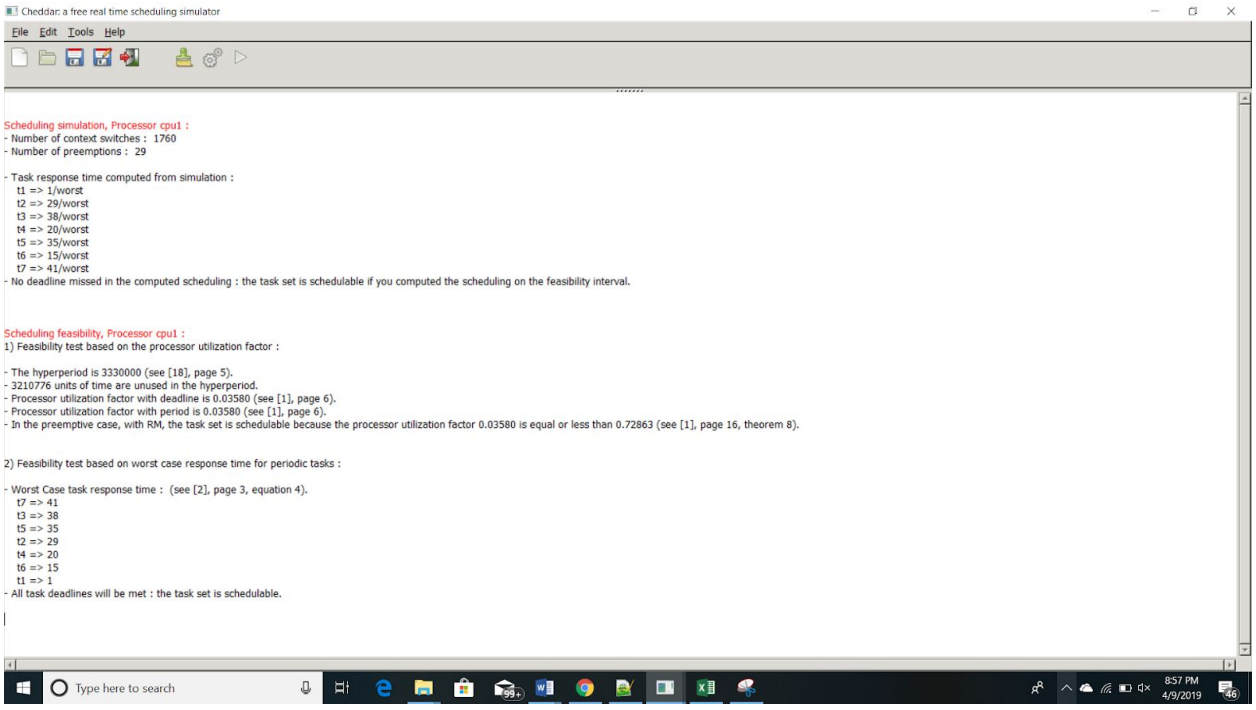


Figure 13 : Running Cheddar and calculating CPU Utilization

T	333.33	1000	2000	1000	2000	1000	10000			
C	1	9	3	5	6	14	3			
C/T	0.003	0.009	0.0015	0.005	0.003	0.014	0.0003		CPU Utilization =	0.0358

Seqgen2x

Explanation and execution of code on target

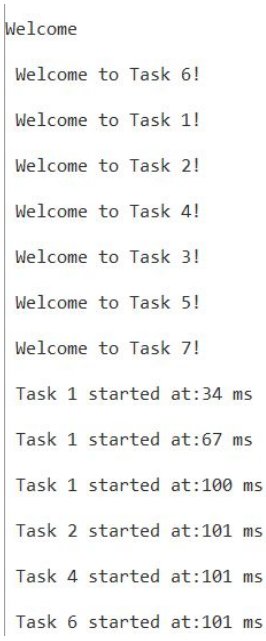


Figure 14 : Execution of seqgen2x.c code on FreeRTOS

In this code, seven tasks are created and these tasks are scheduled using a sequencer. A hardware timer is used and the sequencer is implemented in the ISR of this timer. The priority of each task is assigned on the basis of Rate Monotonic Policy which states that higher priority will be given to a task with higher frequency. Accordingly, all seven services are assigned priorities. Each task has a synthetic load which prints the start time of the task and the WCET of each task is printed. Semaphores are used for task synchronization. xSemaphoreGiveFromISR is used in the timer ISR for each task and xSemaphoreTake is used by each task post which the synthetic load is executed. The timer ISR is loaded such that it occurs at the clock frequency (50000000 MHz) divided by the sequencer frequency that is required which is 60Hz in this case.The timer ISR maintains a variable called seqCnt and based on this it releases the semaphore according to respective request time for each task. When this variable count reaches 900, each task is aborted and the test completes. We use mutexes for the synthetic load so that it is not preempted by any other task which would result in the timestamp not being printed correctly because all the tasks use the same UART. Due to the use of these mutexes, any higher priority task is unable to preempt a lower priority task which results in priority inversion. This causes variation in the WCET of the tasks with same priority.

Timestamps and WCET estimates

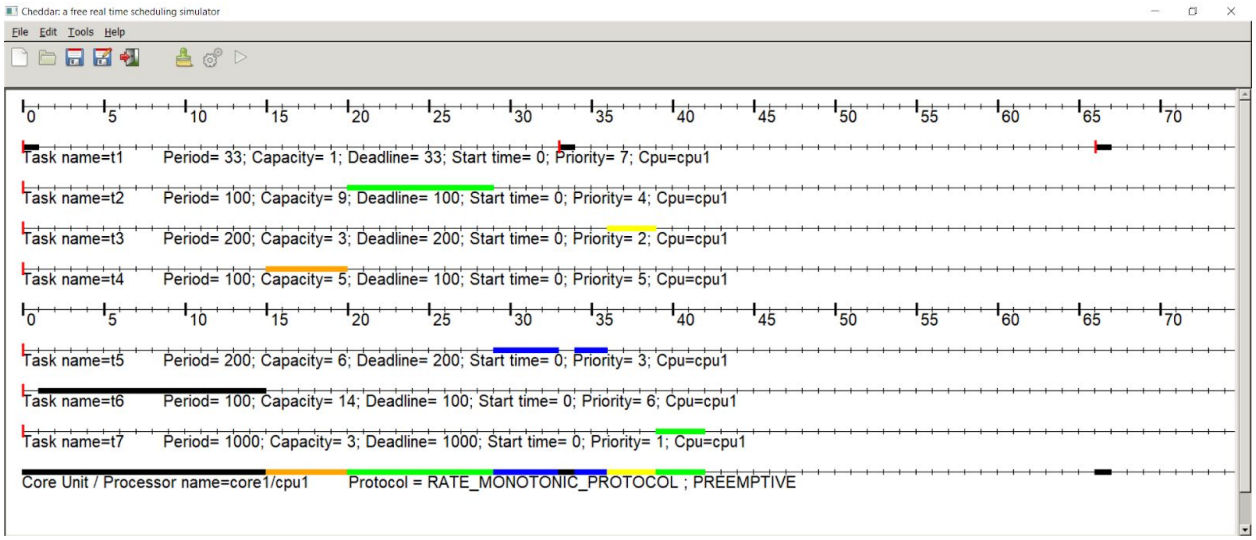
Figure 15 : Printing timestamps and WCET calculation

D, C, and T Table for all services

(ms)	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
D	33.33	100	200	100	200	100	1000
T	33.33	100	200	100	200	100	1000
C	1	9	3	5	6	14	3

Table 4 : D, C and T for seqgen2x.c on FreeRTOS

Cheddar schedule and % CPU utilization



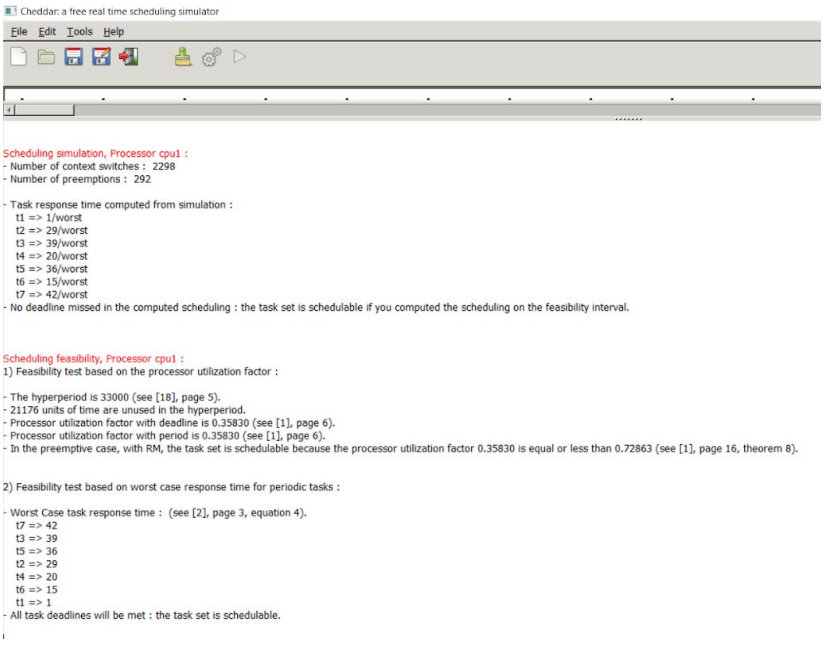


Figure 16 : Running Cheddar and calculating CPU Utilization

T	33.33	100	200	100	200	100	1000			
C	1	9	3	5	6	14	3			
C/T	0.030003	0.09	0.015	0.05	0.03	0.14	0.003		CPU Utilization =	0.358003

Comparison between Linux and FreeRTOS

(ms)	S1	S2	S3	S4	S5	S6	S7
D	333.33	1000	2000	1000	2000	1000	10000
T	333.33	1000	2000	1000	2000	1000	10000
C(Linux)	0.061	0.057	0.039	0.044	0.039	0.041	0.243
C(FreeRTOS)	1	9	3	5	6	14	3

Table 5 : Comparison of D, C and T for seqgen.c on Linux and FreeRTOS

(ms)	S1	S2	S3	S4	S5	S6	S7
D	33.33	100	200	100	200	100	1000
T	33.33	100	200	100	200	100	1000
C(Linux)	0.205	0.201	0.171	0.258	0.173	0.23	0.175
C(FreeRTOS)	1	9	3	5	6	14	3

Table 6 : Comparison of D, C and T for seqgen2x.c on Linux and FreeRTOS

From above tables, we infer that Linux takes less time to execute the codes than FreeRTOS. The main reason for this is that the time taken for execution of the synthetic load offered by the UARTPrintf() is more than the syslog() which is in range of few microseconds. In FreeRTOS, hardware timer is available which will result in accurate scheduling of tasks whereas in Linux sleep function is implemented which can vary and hence Linux is not reliable in case of hard real time systems. In Linux, SCHED_FIFO based scheduling is used which is a run to completion policy and not preemptible and hence a higher priority task will not be serviced until the low priority task finishes its execution which makes Linux unsuitable for hard real time tasks. FreeRTOS also uses run to completion policy but it allows preemption and hence it can be used in hard real time applications. Also, there is no function or method to calculate the time in nanoseconds in FreeRTOS as it gives us the accuracy in milliseconds.

3) [40 points] 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.

Solution:

Linux:

SEQGEN100X:

For the revision, there were few changes required in the code. The time for the sequencer was modified to 333333.33 ns (1/3000 Hz). Also the periods of the individual services was increased by a factor of 100. The release time according to the sequence count (SeqCnt) was reworked as shown below in Figure 19.

```
Sequencer - 3000 Hz
                    [
Service_1 - 300 Hz
                    [
Service_2 - 100 Hz
                    [
Service_3 - 50 Hz, e
                    [
Service_4 - 100 Hz,
                    [
Service_5 - 50 Hz, e
                    [
Service_6 - 100 Hz,
                    [
Service_7 - 10 Hz, e
                    [
```

Figure 17 : The updated frequencies for sequencer and the services (100x)

```
void *Sequencer(void *threadp)
{
    struct timeval current_time_val;
    struct timespec delay_time = {0,333333.33}; // delay for 0.33 msec, 3000 Hz
    struct timespec remaining_time;
```

Figure 18 : Changing the delay_time for the revised sequencer (3000 Hz)

```
// Release each service at a sub-rate of the generic sequencer rate

// Service_1 = RT_MAX-1 @ 300 Hz
if((seqCnt % 10) == 0) sem_post(&semS1);

// Service_2 = RT_MAX-2 @ 100 Hz
if((seqCnt % 30) == 0) sem_post(&semS2);

// Service_3 = RT_MAX-3 @ 50 Hz
if((seqCnt % 60) == 0) sem_post(&semS3);

// Service_4 = RT_MAX-2 @ 100 Hz
if((seqCnt % 30) == 0) sem_post(&semS4);

// Service_5 = RT_MAX-3 @ 50 Hz
if((seqCnt % 60) == 0) sem_post(&semS5);

// Service_6 = RT_MAX-2 @ 100 Hz
if((seqCnt % 30) == 0) sem_post(&semS6);

// Service_7 = RT_MIN @ 10 Hz
if((seqCnt % 300) == 0) sem_post(&semS7);
```

Figure 19 : Revised code for release of each service according to its period

```
Starting Sequencer Demo
System has 4 processors configured and 4 available.
Using CPUS=1 from total available.
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE PROCESS
rt_max_prio=99
rt_min_prio=1
Service threads will run on 1 CPU cores
pthread_create successful for service 1
pthread_create successful for service 2
Frame Sampler thread @ sec=0, msec=940
pthread_create successful for service 3
pthread_create successful for service 4
Time-stamp with Image Analysis thread @ sec=0, msec=940
Time-stamp Image Save to File thread @ sec=0, msec=940
pthread_create successful for service 5
Processed Image Save to File thread @ sec=0, msec=941
pthread_create successful for service 6
pthread_create successful for service 7
Start sequencer
Difference Image Proc thread @ sec=0, msec=940
pthread_create successful for sequeencer service 0
Sequencer thread @ sec=0, msec=942
Send Time-stamped Image to Remote thread @ sec=0, msec=941
10 sec Tick Debug thread @ sec=0, msec=941
Thread idx=1 Count = 1 ,timestamp 0 sec, 0 msec (47 microsec), ((47188 nanosec))

Thread idx=1 Count = 2 ,timestamp 0 sec, 0 msec (40 microsec), ((40573 nanosec))

Thread idx=1 Count = 3 ,timestamp 0 sec, 0 msec (42 microsec), ((42031 nanosec))

Thread idx=2 Count = 1 , timestamp 0 sec, 0 msec (39 microsec), ((39063 nanosec))

Thread idx=4 Count = 1 ,timestamp 0 sec, 0 msec (64 microsec), ((64635 nanosec))

Thread idx=6 Count = 1 ,timestamp 0 sec, 0 msec (41 microsec), ((41771 nanosec))

Thread idx=1 Count = 4 ,timestamp 0 sec, 0 msec (37 microsec), ((37657 nanosec))

Thread idx=1 Count = 5 ,timestamp 0 sec, 0 msec (56 microsec), ((56145 nanosec))

Thread idx=1 Count = 6 ,timestamp 0 sec, 0 msec (38 microsec), ((38958 nanosec))
```

Figure 20 : Execution of seqgen100x.c along with timestamps on RPi

```
TEST COMPLETE
Thread idx=1 ,WCET timestamp 0 sec, 0 msec (56 microsec), ((56145 nanosec))

Thread idx=2 ,WCET timestamp 0 sec, 0 msec (66 microsec), ((66407 nanosec))

Thread idx=3 ,WCET timestamp 0 sec, 0 msec (38 microsec), ((38385 nanosec))

Thread idx=4 ,WCET timestamp 0 sec, 0 msec (64 microsec), ((64635 nanosec))

Thread idx=5 ,WCET timestamp 0 sec, 0 msec (126 microsec), ((126979 nanosec))

Thread idx=6 ,WCET timestamp 0 sec, 0 msec (41 microsec), ((41771 nanosec))

Thread idx=7 ,WCET timestamp 0 sec, 0 msec (156 microsec), ((156771 nanosec))
```

Figure 21 : WCET calculation for seqgen100x.c

D, T and C Table:

(ms)	S1	S2	S3	S4	S5	S6	S7
D	3.333	10	20	10	20	10	100
T	3.333	10	20	10	20	10	100
C	0.056	0.066	0.038	0.064	0.126	0.041	0.156

Table 7 : D, C and T for seqgen100x.c

Figure 22 : Simulation in Cheddar

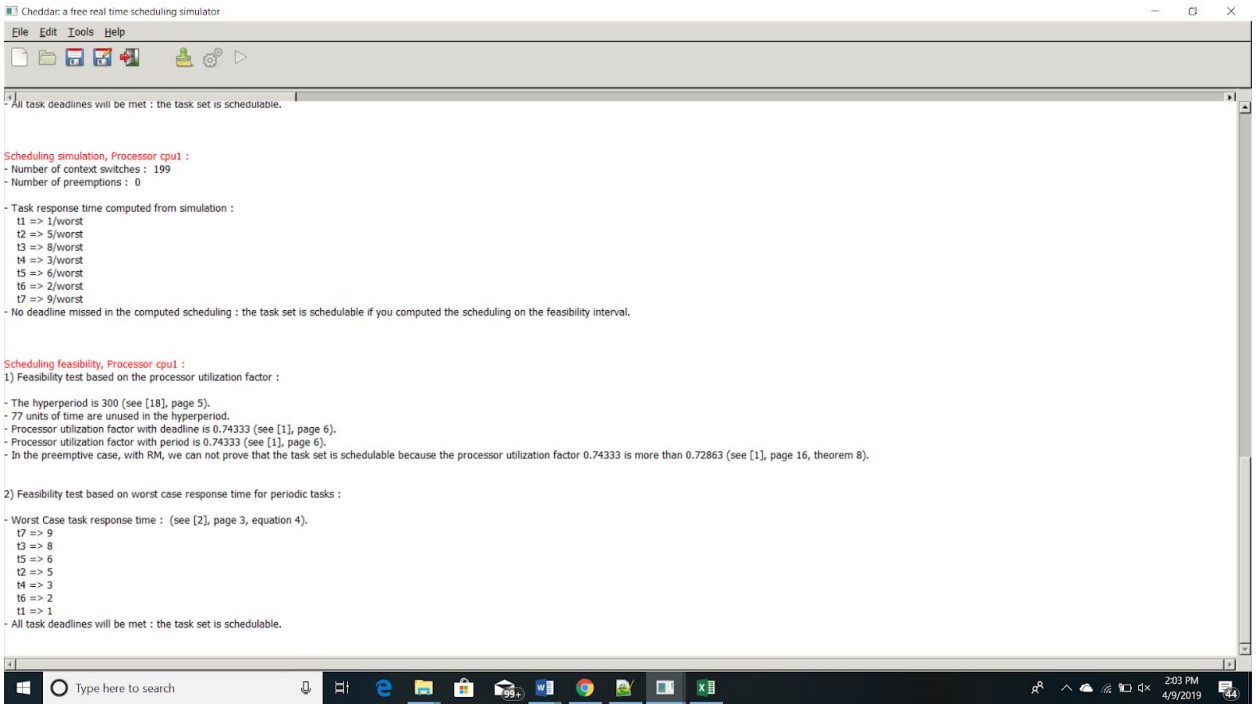


Figure 23 : Scheduling test in Cheddar

T	3.333	10	20	10	20	10	10		
C	0.056	0.066	0.038	0.064	0.126	0.041	0.156		
C/T	0.016802	0.0066	0.0019	0.0064	0.0063	0.0041	0.0156	CPU Utilization =	0.0577017

Figure 24 : CPU Utilization for seqgen100x.c

FreeRTOS:

Explanation and execution of code on target

In this code, we were expected to increase the sequencer frequency and all service frequencies by a factor of 100 i.e. 3000 Hz. We were unable to do so as with this high frequency, the semaphore give from the ISR starts to fail. The clock rate for tiva is 80MHz which is too slow for this high speed application. The synthetic load along with the use of mutexes for the printing of timestamps will take time around 8-9ms which is more than the period of the tasks and hence it is not feasible. From many trials, we found out that the code safely runs at a maximum of 100 Hz sequencer frequency. However, on running in Cheddar, the tasks were not schedulable as LUB is failing.

Timestamps and WCET estimates

```
WCET for Task 1:4 ms
Task 4 started at:9001 ms
WCET for Task 4:5 ms
Task 2 started at:9001 ms
WCET for Task 2:9 ms
Task 6 started at:9001 ms
WCET for Task 6:14 ms
Task 3 started at:9017 ms
WCET for Task 3:6 ms
Task 5 started at:9018 ms
WCET for Task 5:7 ms
Task 7 started at:9027 ms
WCET for Task 7:3 ms
```

Figure 25 : Printing timestamps and WCET calculation

D, C, and T Table for all services

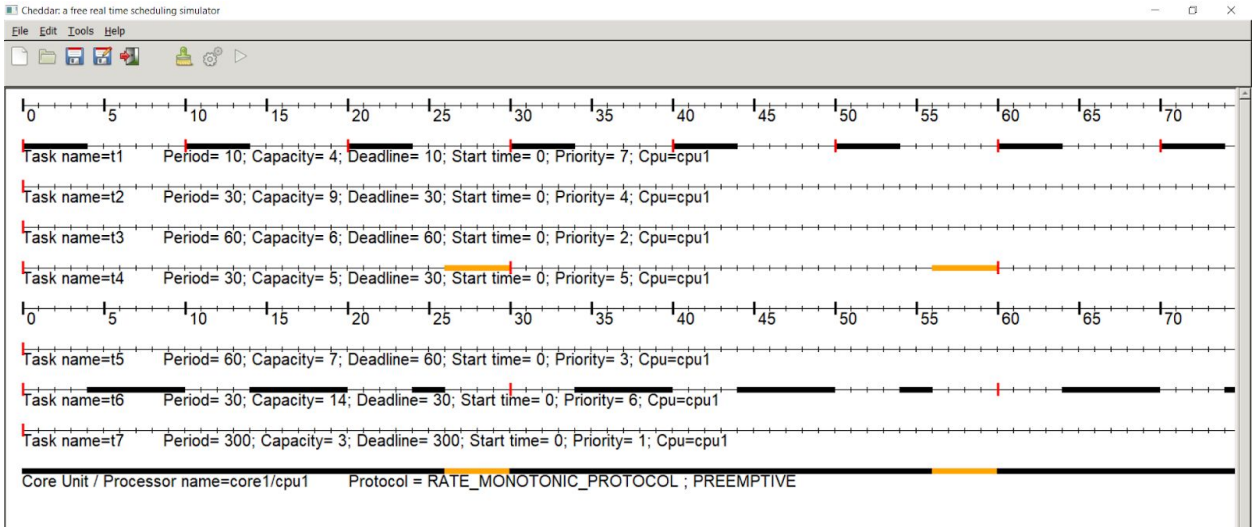
The set of services are not schedulable if we increase the periods of the sequencer and the services by a factor of 100.
Hence we ran the same code with a reduced factor. From many trials, the value for which the sequencer would work was found out to be 100Hz.

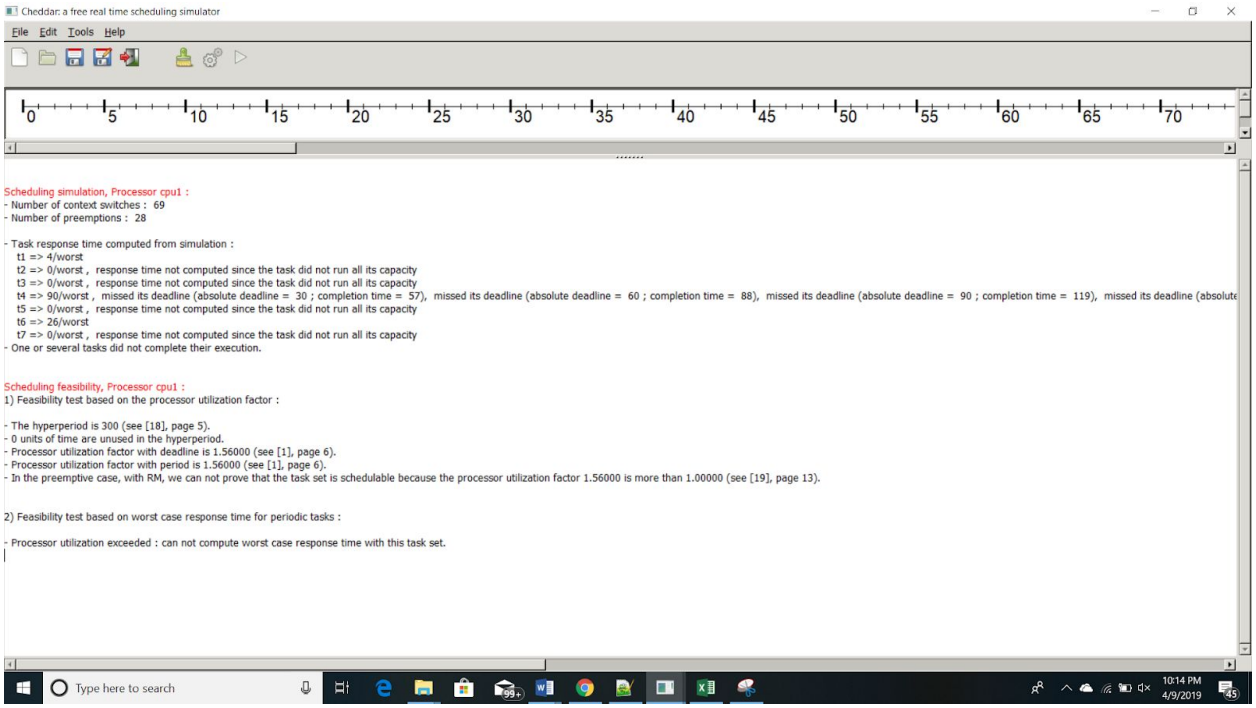
(ms)	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
D	10	30	60	30	60	30	300
T	10	30	60	30	60	30	300
C	4	9	6	5	7	14	3

Table 8 :Updated D, C and T values

Cheddar schedule and % CPU utilization

The tasks are not schedulable as the CPU utilization exceeds 1.





T	10	30	60	30	60	30	300		
C	4	9	6	5	7	14	3		
C/T	0.4	0.3	0.1	0.166667	0.116667	0.466667	0.01	CPU Utilization =	1.56

Figure 26 : Running Cheddar and calculating CPU Utilization

Comparison between Linux and FreeRTOS

- The clock rate for tiva is 80MHz which is too slow for this high speed application while Raspberry Pi has a clock rate in gigahertz range and hence we can increase the sequencer frequency and all service frequencies by a factor of 100 in Linux.
- The synthetic load along with the use of mutexes for the printing of timestamps will take time around 8-9ms which is more than the request period of the tasks and hence it is not feasible while in Linux the synthetic load which is syslog takes time in microseconds to execute which makes it feasible.
- In FreeRTOS, hardware timer is available which will result in accurate scheduling of tasks whereas in Linux sleep function is implemented which can vary and hence Linux is not reliable in case of hard real time systems.

REFERENCES

- FreeRTOS Reference manual.
- Seqgen.c and seqgen2x.c by Professor Sam Siewert.