

**ECEN 5623, Real-Time Embedded Systems:**  
**Exercise #1- Invariant LCM Schedules**

by

Sarthak Jain  
Siddhant Jajoo

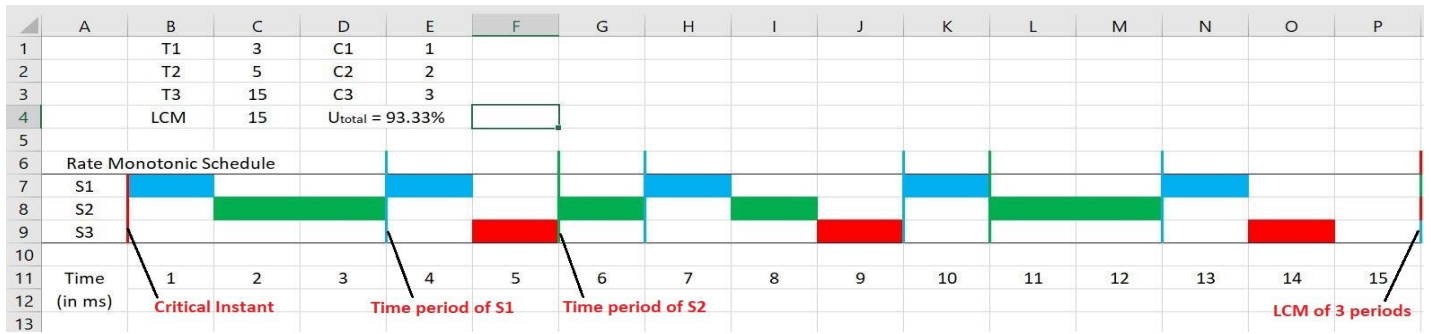


University of Colorado  
Boulder

**Under the guidance of:**  
**Prof. Timothy L. Scherr**  
**Dated: February 5, 2019**

## Question 1

### a) Timing Diagram for Services S1, S2, S3



### b) Schedule feasibility and Safety

The above system has been scheduled successfully with the Rate Monotonic policy over the LCM of the time periods of the three services as:

$T_1 = 3$ ;  $T_2 = 5$ ;  $T_3 = 15$ ;  $C_1 = 1$ ;  $C_2 = 2$ ;  $C_3 = 3$ ;  $LCM = 15$ .

Being schedulable with the RM policy, we can say the system is feasible.

However, the Liu-Layland paper advises a 30% margin for processor utilization to consider a system real-time safe. If, due to some external signal's interference, any service gets delayed, the entire system loses its schedulability. Hence, we term the system real-time unsafe.

Total CPU utilization by the three services is 93.33%.

### c) Utility and Method description

We know the utility of Rate Monotonic Scheduling policy to be that if the RM policy is a sufficient feasibility test, in the sense that if a schedule passes the Rate Monotonic test, it is definitely schedulable by any other algorithm as well. If the planned schedule doesn't pass the Rate Monotonic test, it may or may not be able to be scheduled.

Keeping the Rate Monotonic policy in mind, 'service 1' was scheduled first, since it has the highest frequency of being called every 3 seconds with a run-time of 1 second. Now 'service 2' is scheduled, and lastly, 'schedule 3'. Thus, this policy leads to a feasible schedule for the given set of services.

## Question 2

### a) Apollo 11 Lunar lander Summary:

The Apollo 11 Lunar Lander would not have landed, and Neil Armstrong would not have taken a giant leap for mankind if not for Margaret Hamilton and her beautifully designed code. The Apollo 11 had 36,864 15-bit words of fixed memory (ROM) and 2048 words erasable memory (known as RAM nowadays). With so little erasable memory, programmers had to reuse the same memory locations for different tasks and ensure that none of the memory data is used by no other processes than the one's intended to use it. The same memory location was shared in as many as seven ways. One can imagine the complexity and the various testing methods employed to successfully ensure that only an intended program uses the data at a specific memory location.

Now, each scheduled job was assigned certain erasable memory called as the core set and if the job required more storage, it would ask the Vector Accumulator (VAC) for more sets of storage. A total of 7 core sets and 5 VAC areas were available. Whenever a job was to be scheduled, the core and VAC areas used to be scanned to check if they were available. If the scheduled job specified NOVAC, then the scanning of VAC area was skipped. If there were no VAC and core areas available, alarms 1201 and 1202 were raised respectively.

During the Apollo 11 mission, the core sets and the VAC area got filled up because of the rendezvous radar which kept on sending data though it had no part in the landing portion of Apollo 11. This left no space for the necessary tasks that were required during landing. This led to the alarm 1202 followed by 1201. But thankfully, this situation was taken care of, the computer was programmed in such a way that if such a situation comes up, the computer drops the low priority tasks and run tasks that are of primary importance. The computer would reboot instantaneously and would recognize the task as secondary importance and shut them off like the radar data. Thus, executing the tasks required at that particular point of time. So, each time 1201 or 1201 alarm was sounded, the computer would reboot and restart the important tasks at that particular point of time. If the programmers had not taken care of this situation, the moon landing would never have happened.

The root cause of overload in Apollo 11 was due to the rendezvous radar sending data based on electrical noise thus filling up the core and VAC sets even though when it was not required. As per the rate monotonic policy, the more frequent tasks have higher priority and preempts the current threads but in the Apollo 11 landing the rate monotonic policy was violated. The only viable reason for violating the rate monotonic policy is that the deadline driven Scheduling policy (i.e. Dynamic) was implemented concurrently with the rate monotonic policy which is called as the mixed scheduling algorithm. As soon as the system rebooted, priority was given to the important task required and ignored all the other radar data.

b) Plot this Least Upper bound as a function of number of services.



U = Utilization Factor.

M= Number of Services.

c) Describe 3 key assumptions they make and document 3 or more aspects of their fixed priority LUB derivation that you don't understand.

The key assumptions made in the derivation are:

- Assuming the rate monotonic policy and assigning the priorities under the mentioned policy i.e. the more frequent task has higher priority. Example:  $T_1 < T_2$ , So  $T_1$  has higher priority than  $T_2$ .
- The runtime has been adjusted to fully utilize the processor time within the critical time zone.
- The Ratio between any two request periods is less than 2.

3 aspects of their fixed priority LUB which we didn't understand are:

- The Liu-Layland paper wishes to prove that  $C_1 = T_2 - T_1$ . In order to do this, they consider a step-factor,  $\Delta$ , and consider two systems. The first system considers  $C_1'$  as  $C_1 + \Delta$ , and the second system considers  $C_1''$  as  $C_1 - 2\Delta$ . The use of  $2\Delta$  is what confused us, as the use of  $C_1 - \Delta$  made more sense for the use of this derivation. It was our understanding that the case of  $C_1 + \Delta$  and  $C_1 - \Delta$  would be used to prove that the only possible value for  $\Delta$  would be 0.
- The resultant solution to the difference equation of  $\delta U / \delta g_j$  is found as:  
 $g_j = 2^{(m-j)/m} - 1$ . Liu-Layland have stated that  $U = m(2^{1/m} - 1)$  follows from the solution. We tried substituting the solution in the equation for the utilization factor, however, the term of '-m' did not follow from any calculation of ours.
- In the next theorem, Liu-Layland replace the task  $\tau_i$  by  $\tau_i'$ , such that  $T_i' = qT_i$  and  $C_i' = C_i$ . Now they increase  $C_m$  by a certain amount of time to again fully utilize the processor. They have termed this increase as being at most  $C_i(q - 1)$ . However, the relation between  $C_m$  and  $C_i$  is one we are not able to comprehend. Increasing  $C_m$  by an amount equal to the difference between  $T_m$  and the sum of run-times of all other tasks would make sense.

**d) Would RM analysis have prevented the Apollo 11 1201/1202 errors and potential mission abort? Why or why not?**

The RM analysis would have worked only if the request rates of the desired job would have been greater than the radar job or the 1202/1201 alarms, thus, preempting it and carrying out the necessary function.

If by any chance the radar or 1201/1202 alarms had a request rate more than the desired job, the scheduler would always execute the radar function in spite of the desired one. Due to this backlog of services, the scheduler would anyway be unable to handle such a large task request rate, and the system would anyway have resulted in the same conclusion as it did. So RM analysis would not have prevented the potential mission abort.

### Question 3

**a) Code Description**

The basic functionality of the code is to print the time elapsed in executing a particular thread or a process and even to find the delay error. This is achieved using the `clock_gettime` call. The `clock_gettime` has 2 arguments `clockid_t` and `timespec`. The first argument is the identifier of a particular clock on which to act. The following clocks are supported : `CLOCK_REALTIME`, `CLOCK_MONOTONIC`, `CLOCK_PROCESS_CPUTIME_ID`, `CLOCK_THREAD_CPUTIME_ID`. In this program, we use the system-wide real-time clock i.e. `CLOCK_REALTIME`. The second argument is the `timespec` structure which stores the time value retrieved by the function in seconds and nanoseconds. We use this function call between two points in the code in which we ought to get the time elapsed. Thus, subtracting the first value from the second in order to get the time

consumed by the program to execute a certain section of code. The `delta_t` (`struct timespec *stop, struct timespec *start, struct timespec *delta_t`) function defined carries out this task of subtracting and calculating the time required using the parameters. The `delay_test` (`void *threadID`) initializes and sets up the high-resolution clock and starts the time stamp. The kernel sleeps for 3 seconds after starting the time stamp using the `nanosleep` (`const struct timespec *req, struct timespec *rem`) function. The time stamp is noted once again at the last after it completes the duration of its sleep. The `end_delay_test` (`void`) function simply prints all the values obtained after the calculation.

At the start of code execution, the default scheduling policy is `SCHED_OTHER`. If `RUN_RT_THREAD` is defined, we change the scheduling policy to `SCHED_FIFO` and set the scheduling policy to the maximum priority. In addition to this, we also create a thread with `SCHED_FIFO` as scheduling policy and `main_param` as the maximum priority. Later, we call the function `pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)` to create a thread with certain modified attributes and `delay_test` (`void *threadID`) as the starting function. The function `pthread_join(pthread_t thread, void **retval)` is used to stall the program until the thread completes its execution and then the pthread is destroyed using `pthread_attr_destroy(pthread_attr_t *attr)`. Now, if `RUN_RT_THREAD` is not defined, the program simply runs the `delay_test` (`((void *)0)`) function skipping the part of creating a thread.

```

ubuntu@tegra-ubuntu:~/hw1$ make
gcc -MD -g -c posix_clock.c
posix_clock.c: In function 'end_delay_test':
posix_clock.c:161:3: warning: format '%ld' expects argument of type 'long int',
but argument 2 has type 'unsigned int' [-Wformat=]
    printf("Sleep loop count = %ld\n", sleep_count);
    ^
gcc -g -o posix_clock posix_clock.o -lpthread -lrt
ubuntu@tegra-ubuntu:~/hw1$
ubuntu@tegra-ubuntu:~/hw1$
ubuntu@tegra-ubuntu:~/hw1$
ubuntu@tegra-ubuntu:~/hw1$
ubuntu@tegra-ubuntu:~/hw1$
ubuntu@tegra-ubuntu:~/hw1$ ./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 = 1549181124, nanoseconds = 762184468
RT clock stop seconds = 1549181127, nanoseconds = 762261859
RT clock DT seconds = 3, nanoseconds = 77391
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 77391
ubuntu@tegra-ubuntu:~/hw1$

```

Screenshot of make and run of the program when RUN\_RT\_THREAD is not defined

```

ubuntu@tegra-ubuntu:~/RTES/hw1$ make
gcc -MD -O3 -g -c posix_clock.c
posix_clock.c: In function 'end_delay_test':
posix_clock.c:162:3: warning: format '%ld' expects argument of type 'long int', but argument 2 has type 'unsigned int' [-Wformat=]
    printf("Sleep loop count = %ld\n", sleep_count);
    ^
gcc -O3 -g -o posix_clock posix_clock.o -lpthread -lrt
ubuntu@tegra-ubuntu:~/RTES/hw1$ sudo ./posix_clock
[sudo] password for ubuntu:
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 = 1549218744, nanoseconds = 418391308
RT clock stop seconds = 1549218747, nanoseconds = 418444029
RT clock DT seconds = 3, nanoseconds = 52721
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 52721
ubuntu@tegra-ubuntu:~/RTES/hw1$

```

Screenshot of make and run of the program when RUN\_RT\_THREAD is defined

- b) **Most RTOS vendors brag about three things:** 1) Low Interrupt handler latency, 2) Low Context switch time and 3) Stable timer services where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift. Why are each important?

1. Low Interrupt Handler Latency:

The Low Interrupt Handler Latency is important to control machinery in real-time. Some interrupt handler routines disable the interrupts when an interrupt is triggered to protect the critical sections of code. By doing so, there is a risk of missing an interrupt if an interrupt is generated during the critical section of the code. Some interrupts move into the pending state and are executed when the code gets out of the critical section. This only increases the latency to respond to the interrupt and may affect the real time services. If such a latency results in the delay of hard real time services, the consequences can be catastrophic. For this reason, we want the Interrupt Handler Latency to be as minimum as possible. The context switching plus the time taken in the vector table should be as minimum as possible to have a low Interrupt Handler Latency.

2. Low Context Switch Time:

Context Switching is the process of switching from one task to another. In doing so it stores the stack pointer and the program counter to return to. Context switching too requires some processor cycles to execute. We want the context switch time to be as low as possible because virtually we are wasting our processor cycles doing nothing in context switching. Context switching also increases our code execution time which may affect real time services.

3. Stable Timer Services.

Time values that fall between two non-negative integer multiples of the resolution are truncated down to the smaller multiple of the resolution. So the timer values are not so accurate. These inaccurate timer values will result in inaccurate timer loops leading to significant problems in real time systems. Jitter is the variation of PLL or Oscillator due to electrical noises and drift refers to the variation in oscillator frequency due to environment. For stable timer services the jitter and drift should be as low as possible because these affect the time values and their counting which can result in inaccurate timer services.

- c) **Do you believe the accuracy provided by the example RT-Clock code?**

The accuracy depends on many factors such as how accurate the frequency oscillator is, the jitter and drift of the internal clock. However, the RT-clock cannot be termed as perfectly accurate, as even with a requested delay of 3 seconds, it gives a delay of 3 seconds and 0.257 milliseconds, which is an inaccuracy of roughly .008%.



## Question 4

### a) Code Description

#### 1. simplethread

This code attempts to simply create a specified number of threads, print the thread numbers and sum of numbers till that number, and then joins the threads to the process, thus completing it. It does so by running a loop for “NUM\_THREADS” number of times. In each iteration, the thread ID for that thread is assigned ‘i’, where ‘i’ is the loop count. pthread\_create() function is used to create a thread, entering the function ‘counterThread’ with default attributes and a parameter of thread ID being passed. The function prints the thread ID with sum of numbers till that thread ID and exits. This function is called as many times as the number of threads to be created. Finally, threads are joined, destroyed and the program runs to completion. A screenshot of the output is attached below:



```
ubuntu@tegra-ubuntu:~/RTES/Exercise1/simplethread$ make
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
ubuntu@tegra-ubuntu:~/RTES/Exercise1/simplethread$ ./pthread
Thread idx=6, sum[0...6]=21
Thread idx=7, sum[0...7]=28
Thread idx=8, sum[0...8]=36
Thread idx=5, sum[0...5]=15
Thread idx=9, sum[0...9]=45
Thread idx=10, sum[0...10]=55
Thread idx=4, sum[0...4]=10
Thread idx=11, sum[0...11]=66
Thread idx=3, sum[0...3]=6
Thread idx=2, sum[0...2]=3
Thread idx=1, sum[0...1]=1
Thread idx=0, sum[0...0]=0
TEST COMPLETE
ubuntu@tegra-ubuntu:~/RTES/Exercise1/simplethread$
```

The output print of threads is jumbled for a particular reason. As we understand, as each thread is created, it is not necessary for the system to be free. The first thread at which the processor is found to be free, is created, and its ID is printed. Following this, the pending threads are processed.

#### 2. rt\_simplethread

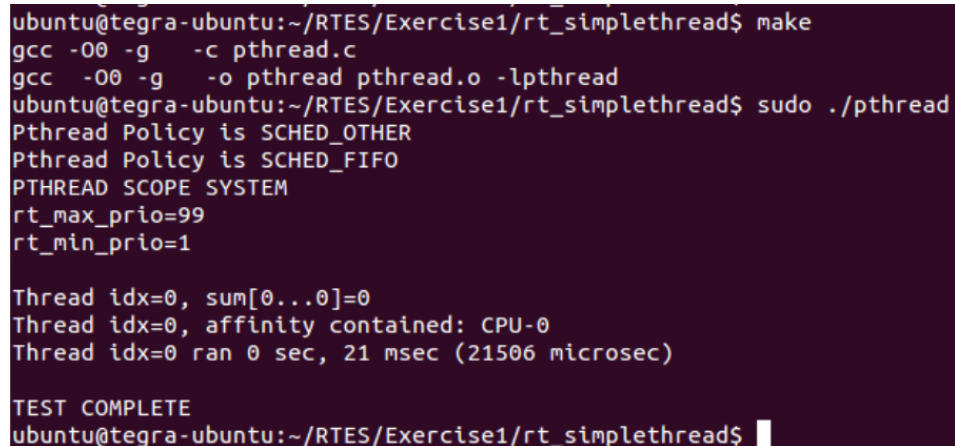
The purpose of this program is to simply keep a track of the current thread's scheduling policy, its scope, CPU affinity, time for which the thread runs, and print the same. The program makes use of some of the code described in Q.3 to display the time elapsed while executing the particular thread.

The program begins by adding CPU0 to the current set of CPUs, 'cpu\_set'. The maximum and minimum priorities attainable for the FIFO scheduling policy are obtained and

stored in variables. A function to print the current scheduling policy is called, and it is printed on terminal as "OTHER", which makes sense as no policy has been set yet. Now, maximum priority obtained above is assigned to the process, and the current process' scheduling policy is set to FIFO. Function 'print\_scheduler()' is called again, and we can see the policy change from "OTHER" to "FIFO". The thread scope is printed as a safe measure and is found to be systemic. The maximum and minimum priorities are also printed for good measure.

Now, the code for thread attributes and parameters is executed. The pthread attributes are initialized to default, it's scheduling inheritance of attributes set to the attributes object specified, it's scheduling policy set, and CPU affinity set to CPU0. Using the previously set scheduling priorities, the thread parameters' priority is set to maximum as well. Now the thread is created, using default attributes, 'counterThread' as the entry function and thread ID being passed as parameter.

The thread now begins execution, and three objects of type 'timespec' are created', to keep track of start time, finish time and difference between the two. Using the 'clock\_gettime()' function, start time is updated. The computation section begins now, where using two pre-defined integers, calculation of Fibonacci numbers is performed. Using 'pthread\_getaffinity\_np', the previously set affinity is again obtained, and printed along with the thread ID. It can be seen as CPU0, which makes sense. Finally, finish time is updated, and using the 'delta\_t' function, difference between the two is stored in 'thread\_dt' and printed. We can see the difference as 20 ms, implying the thread took 20 ms for the computation of Fibonacci series. Finally, thread is joined to the main program. A screenshot of the output is attached below:



```
ubuntu@tegra-ubuntu:~/RTES/Exercise1/rt_simplethread$ make
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
ubuntu@tegra-ubuntu:~/RTES/Exercise1/rt_simplethread$ sudo ./pthread
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1

Thread idx=0, sum[0...0]=0
Thread idx=0, affinity contained: CPU-0
Thread idx=0 ran 0 sec, 21 msec (21506 microsec)

TEST COMPLETE
ubuntu@tegra-ubuntu:~/RTES/Exercise1/rt_simplethread$
```

### 3. rt\_thread\_improved

This program is quite similar to the one above, in that it keeps a track of the thread's scheduling policy, it's scope, CPU affinity and time for which the thread runs. The only difference is that the program creates a pre-defined number of threads (in this case, 4) and attempts to run each on a different CPU, if available.

The program first prints the number of processors configured and available. Based on this number, it sets the corresponding bits in the 'allcpuset'. From here on, the program

proceeds similar to the one above till the point where thread attributes and parameters are set. Now, in the loop running for 'number of threads' times, a previously initialized bitmask, 'threadcpu' the corresponding bit for each CPU is set as the loop iterates. Eg. If number of processors = 2 and number of threads = 4, the iteration for thread 0 sets 0<sup>th</sup> bit in 'threadcpu', the iteration for thread 1 sets 1<sup>st</sup> bit in 'threadcpu', and so on.

Thread attributes and parameters are set after this step, and thread is created. The thread runs to completion, printing the thread ID, it's core, core affinity and time taken to compute the Fibonacci series as it does so. Finally, thread is joined to the main program.

One discrepancy we note is that although the scheduling policy attribute has been assigned as FIFO, the cores are not being executed in that order, and neither is thread 'i' running on CPU 'i'. This is because of a possible bug in the code, that at the time of thread creation, default attributes are used, which seems incongruous with the earlier assignment of scheduling policy and priority to 'rt\_sched\_attr[i]'. If we use 'rt\_sched\_attr[i]' as the attribute in the function of 'pthread\_create()', we see that CPU 'i' is assigned to thread 'i', and the threads are executed in FIFO order as well. Screenshot 1 shows the erroneous output, while screenshot 2 shows the synchronous output. Output screenshot is attached below:

```
ubuntu@tegra-ubuntu:~/RTES/Exercise1/rt_thread_improved$ sudo ./pthread
This system has 4 processors configured and 4 processors available.
number of CPU cores=4
Using sysconf number of CPUs=4, count in set=4
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Setting thread 0 to core 0
CPU-0
Launching thread 0
Setting thread 1 to core 1
CPU-1
Launching thread 1
Setting thread 2 to core 2
CPU-2
Launching thread 2
Setting thread 3 to core 3
CPU-3
Launching thread 3

Thread idx=0, sum[0...100]=4950
Thread idx=0 ran on core=3, affinity contained: CPU-0 CPU-1 CPU-2 CPU-3
Thread idx=2, sum[0...300]=44850
Thread idx=2 ran on core=0, affinity contained: CPU-0 CPU-1 CPU-2 CPU-3

Thread idx=2 ran 0 sec, 363 msec (363280 microsec)

Thread idx=1, sum[0...200]=19900
Thread idx=1 ran on core=1, affinity contained: CPU-0 CPU-1 CPU-2 CPU-3

Thread idx=1 ran 0 sec, 364 msec (364049 microsec)

Thread idx=3, sum[0...400]=79800
Thread idx=3 ran on core=2, affinity contained: CPU-0 CPU-1 CPU-2 CPU-3

Thread idx=3 ran 0 sec, 350 msec (350445 microsec)

Thread idx=0 ran 0 sec, 370 msec (370918 microsec)

TEST COMPLETE
ubuntu@tegra-ubuntu:~/RTES/Exercise1/rt_thread_improved$
```

Screenshot 1

```

ubuntu@tegra-ubuntu:~/RTES/Exercise1/rt_thread_improved$ sudo ./pthread
This system has 4 processors configured and 4 processors available.
number of CPU cores=4
Using sysconf number of CPUs=4, count in set=4
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Setting thread 0 to core 0
CPU-0
Launching thread 0
Setting thread 1 to core 1
CPU-1
Launching thread 1
Setting thread 2 to core 2
CPU-2
Launching thread 2
Setting thread 3 to core 3
CPU-3
Launching thread 3

Thread idx=1, sum[0...200]=19900
Thread idx=1 ran on core=1, affinity contained: CPU-1
Thread idx=0, sum[0...100]=4950
Thread idx=0 ran on core=0, affinity contained: CPU-0

Thread idx=0 ran 0 sec, 308 msec (308366 microsec)

Thread idx=1 ran 0 sec, 308 msec (308381 microsec)

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

Thread idx=2 ran 0 sec, 308 msec (308698 microsec)

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

Thread idx=3 ran 0 sec, 308 msec (308749 microsec)

TEST COMPLETE

```

Screenshot 2

#### b) Semaphores, synchronization, tasks and threads

A semaphore is simply a variable, being used here to perform synchronization between the two threads of execution. It's basically serving here as a lock to stall a particular thread's execution until we wish it to run. In this way, we have synchronized the two tasks, by unlocking the semaphore for one thread when the other has completed running. Task priority also comes into the picture and is assigned using thread attributes. Semaphore manipulation done by using the functions 'sem\_wait()' and 'sem\_post()', the description of which is given below.

Giving a small introduction about the 'sem\_wait()' and 'sem\_post()' to be used ahead, 'sem\_wait()' decrements the semaphore value pointed to by the argument of the function. If the semaphore value consequently becomes 0, the thread proceeds. However, if the value is already zero before function call, then the call blocks the thread until it is possible to decrement the semaphore. 'sem\_post()' increments the value pointed to by semaphore, making it available for other threads to lock the same.

A process is an active executing program. In Linux, a process is generally referred to as task. A thread is a single path of execution through code. Task is much more complicated than threads. A task consists of more than one thread. The process has a whole virtualize memory address to itself. The threads have a unique program counter, process stack and set of registers in the memory address space. To Linux, a thread is just a special kind of process.

Thread is a process that is associated with a process model. VxWorks tasks run in the same memory space. The code will execute based on task priority. The system resources will be given to the task which has higher priority. In conclusion, tasks in VxWorks can be somewhat thought as threads in Linux.

### c) Code Description

Our initial few lines of code would be similar to how the above codes are executed. We would create two pthread objects, 'thread\_fib10' and 'thread\_fib20'. Two control variables for each thread's execution are also created, which can be called 'ctrl1' and 'ctrl2'. Both are initially assigned a value of 1. Alongside this, we would create pthread\_attribute objects, semaphore objects and scheduling parameters. In the main function, we initialize the semaphores for both threads, to the value of 1. This is done using the function 'sem\_init()', which is analogous to the function 'semBCreate' in VxWorksRTOS. We also create two objects of type useconds\_t and use them for the nanosleep to be provided to the main execution thread. Each thread's attributes are created, their scheduling attribute's inheritors set, and their scheduling policies assigned. Priorities are assigned to each thread, with the main thread of execution being given maximum, 'thread\_fib10' being given priority of max – 1 and 'thread\_fib20' being given priority of max – 2.

Proceeding with the program, 'thread\_fib10' is created (using 'pthread\_create()', which is analogous to function 'taskSpawn' in VxWorksRTOS), and given an entry function of 'delay\_fib10'. The function 'delay\_fib10' runs a while loop, with control variable of 'ctrl1'. While 'ctrl1' equals 1, the loop keeps decrementing the semaphore for 'thread\_fib10' by using the function 'sem\_wait()'. We can call this semaphore 'sem1'. It also computes a Fibonacci series such that the run-time of a single iteration of the loop is 10 ms.

When the function 'sem\_wait()' is called above, it's semaphore value is 1 and then decremented to 0. The function proceeds onwards to compute Fibonacci series.

Secondly, 'thread\_fib20' is created, and given an entry function of 'delay\_fib20'. The function 'delay\_fib20' runs a while loop, with control variable of 'ctrl2'. While 'ctrl2' equals 1, the loop keeps decrementing the semaphore for 'thread\_fib20' by using the function 'sem\_wait()'. We can call this semaphore 'sem2'. It also computes a Fibonacci series such that the run-time of a single iteration of the loop is 20 ms.

When the function 'sem\_wait()' is called above, it's semaphore value is 1 and then decremented to 0. The function proceeds onwards to compute Fibonacci series for 20 ms.

The logic for scheduling algorithm can be given as:

```
usleep(20);           // pause main thread execution for 20 ms
                      // thread1 and thread2 are allowed to run for 10 ms each
                      // respectively
sem_post(&sem1);       // main thread wakes up, increments 'sem1'
usleep(20);           // pause main thread execution for 20 ms
                      // thread1 and thread2 are allowed to run for 10 ms each
                      // respectively
```

```

sem_post(&sem1);           // main thread wakes up, increments 'sem1'
usleep(10);                // pause main thread execution for 10 ms
                           // thread1 is allowed to run for 10 ms

ctrl2 = 0;                 // thread2 period has ended, and we want both threads to
                           // execute only // for 1 LCM of periods. Further execution of thread2
                           // after this period is // stopped

sem_post(&sem2);           // 'sem2' is incremented
usleep(10);                // pause main thread execution, allowing thread2 to run for 10 ms
sem_post(&sem1);           // 'sem1' is incremented
usleep(20);                // pause main thread execution for 20 ms
                           // Both threads are allowed to run for 10 ms each

ctrl1 = 0;                 // Further execution of thread1 is stopped
sem_post(&sem1);           // Last service of thread1 is executed
usleep(20);                // pause main thread execution for 20 ms

```

#### d) Output Description

After many iterations, we reach the optimum values for generation of synthetic load. The load was executed on the Jetson TK1 by NVIDIA. It is rather difficult to achieve predictable and reliable results on the basis of CPU time values alone, as the time values are machine specific, and depend on overhead as well. It depends on how one's code is written, as well as on the machine. Execution sequence is as outlined above.

```

ubuntu@tegra-ubuntu:~/RTES/Exercise1/new_code$ sudo ./pthread
[sudo] password for ubuntu:
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1

Thread idx=0 priority = 98 timestamp: 10 msec (10079 microsec)utilizing CPU-0
Thread idx=0 priority = 98 timestamp: 30 msec (30097 microsec)utilizing CPU-0
Thread idx=1 priority = 97 timestamp: 40 msec (40071 microsec)utilizing CPU-0
Thread idx=0 priority = 98 timestamp: 50 msec (50097 microsec)utilizing CPU-0
Thread idx=0 priority = 98 timestamp: 70 msec (70157 microsec)utilizing CPU-0
Thread idx=1 priority = 97 timestamp: 80 msec (80145 microsec)utilizing CPU-0
Thread idx=0 priority = 98 timestamp: 90 msec (90229 microsec)utilizing CPU-0

The test took 100 msec (100252 microsec)

TEST COMPLETE
ubuntu@tegra-ubuntu:~/RTES/Exercise1/new_code$ █

```

Code output

e) **Description of challenges faced**

The first challenge faced was that of comprehending the functions being executed in the file 'testdigest.c' and the code written for VxWorksRTOS. Understanding the use of semaphores was quite time-consuming. An interesting concept we noted was how a single semaphore can be used to synchronize tasks. Executing the code on a machine was another challenge. Unless a core is specifically assigned for the program, the machine selects its default number of threads. During the initial testing phase, we had neglected to set the thread affinity and discovered that each thread was running on a different core. Another challenge was that of finding the correct set of parameters that would give an approximate timing of 10 and 20 ms each.

**References-**

1. nanosleep and POSIX 1003.1b RT clock demonstration by Dr. Sam Siewert
2. Independent study on RM Scheduling feasibility tests conducted by Nisheeth Bhat, link provided here: <http://ecee.colorado.edu/~ecen5623/ecen/labs/Linux/IS/Report.pdf>
3. Linux online manual pages: <http://man7.org/linux/man-pages/index.html>