ECEN 5623 REAL TIME EMBEDDED SYSTEMS

EXERCISE 1: INVARIENT LCM SCHEDULES

Submitted By: Suraj B Thite and Puneet Bansal **Date: 4 February 2020**

Q 1) 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). Draw a timing diagram for three services S1, S2, and S3 with T1=3, C1=1, T2=5, C2=2, T3=15, C3=3 where all times are in milliseconds. [Note that you can find examples of timing diagrams in Lecture and here and in Canvas – note that we have not yet covered dynamic priorities, just RM fixed policy described here, so ignore EDF and LLF for now]. Label your diagram carefully and describe whether you think the schedule is feasible (mathematically repeatable as an invariant indefinitely) and safe (unlikely to ever miss a deadline). What is the total CPU utilization by the three services?

Answer:
Rate Monotonic Scheduling policy is static priority scheduling algorithm which works on the principle of assignment of priorities according to the time period of its task. According to the given problem,
There are 3 tasks, S1 S2 and S3 with time period T1 = 3, T2 = 5, T3 = 15 and runtime as C1=1, C2=2 and C3=3. Thus according to Rate Monotonic Policy , the priority of tasks will be S1>S2>S3 as S1 has highest frequency of occurrence while S3 being the least.

Timing Diagram:

The least common multiple of T1, T2 and T3 is 15 .S1 gets highest priority due to its least time period. The timing diagram for each of the services is shown below.

RM Schedule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S1	🟨			🟨			🟨			🟨			🟨		
S2		🟦	🟦			🟦		🟦			🟦	🟦			
S3					🟥				🟥					🟥	

Timing Diagram

In the Diagram above, the Yellow states the CPU time slot allocated for S1 , Blue for S2 and Red for S3 respectively. The S1 has highest priority since it is the service with maximum frequency while S3 has least priority since it has minimum frequency of occurrence. Thus, the priority is **S1>S2>S3** for this situation.

The LCM of all the services is 15. Based on this LCM we can determine whether services are invariant in their period of operation. LCM (3, 5, 15) is 15.

The timing diagram has been plotted for S1, S2 and S3 services in accordance with their assigned priorities. Each service has to complete run times before their time period (T). Since all the processes don't miss their deadlines in the time period being examined, the services are feasible.

From the timing diagram, the services are mathematically repeatable as indefinitely invariant services making them feasible.

Processor Utilization Calculations:

Processor Utilization (U_i) = $\sum (C_i/T_i)$ where U is utilization, C is the execution time and T is the Time period of the services respectively.

Utilization for S1 = C1/T1 = 1/3 = 0.333 = 33.34 %

Utilization for S2 = C2/T2 = 2/5 = 0.4 = 40 %

Utilization for S3 = C3/T3 = 3/15 = 0.2 =20%

Total CPU Utilization = C1/T1 + C2/T2 C3/T3 = 1/3 + 2/5 + 3/15 = 0.93333 = **93.333%**

Feasibility and Safety Issues:

According the given scenario, the CPU is idle for 1 in every 15 time slots and gets occupied with for every 14 out of 15 processor cycles giving total CPU utilization of 93.333% and CPU Idle time of 6.6777% .

With reference to Lieu and Lay land paper, the services are considered safe if the CPU Utilization is less than or equal to the least upper bound CPU Utilization ie m((2^(1/m)-1) where m is the number of services.

Considering m = 3, the Least Upper Bound CPU Utilization comes to be = 3*(2^(1/3) -1) = 0.7797.

Thus Least Upper Bound CPU Utilization (%) = 77.97 %.

Henceforth the total CPU Utilization (%) is 93.33 % i.e. greater than Least Upper Bound CPU Utilization (%) 77.97 % making the **schedule unsafe, but still feasible.**

Q-2) [20 points] Read through the Apollo 11 Lunar lander computer overload story as reported in RTECS Notes, based on this NASA account, and the descriptions of the 1201/1202 events described by chief software engineer Margaret Hamilton as recounted by Dylan Matthews. Summarize the story. What was the root cause of the overload and why did it violate Rate Monotonic policy? Now, read Liu and Layland's paper which describes Rate Monotonic policy and the Least Upper Bound – they derive an equation which advises margin of approximately 30% of the total CPU as the number of services sharing a single CPU core increases. Plot this Least Upper bound as a function of number of services and describe 3 key assumptions they make and document 3 or more aspects of their fixed priority LUB derivation that you don't understand. Would RM analysis have prevented the Apollo 11 1201/1202 errors and potential mission abort? Why or why not?

Answer:

Reading and Summary:

The author explains about the issues faced by the team while programming the LGC which had 36,864 15-bit words. These were Read only Memory which retain their data when the power is off. Apart from Read Only Memory, It had Random Access Memory of 2048 words. The team was forced to the same memory address due to memory constraints in the system. Thus the team was compelled to use the same memory space for different purposes at different time. The access to memory location for controlling to change the data from different processes. A variable written at time t1 might be changed at time t2 while being shared between multiple processes. Exhaustive testing was carried out to ensure that memory location is used by accessed to write by only one process at a time.

The succeeding paragraph highlights the memory allocated to each service. Each job was allocated a "core set" of 12 erasable memory locations that were allocated to store temporary data. In case of more temporary storage, the scheduling request asked for a VAC (vector accumulator). There were 7 core sets and 5 VAC with each VAC containing 44 rewritable words.

The operating system scans the entire VAC and allocates one to the job requiring VAC area. Upon reservation of VAC, core sets were scanned. The scanning of VAC is skipped by operating system if a service makes a request with "NOVAC". If NO VAC AREAS were available, the 1201 alarm would be raised by the code. Also, if or no core sets were available 1202 alarm would be raised by the code.

Root cause and Analysis:

If the CPU processing radar data which was getting scheduled due to radar misconfiguration, 1201 alarm was raised. The core sets eventually got filled up leading to 1201 alarm. The scheduling request that caused the overflowing of VAC area was prime reason for 1201 alarm.

This condition was taken care by the software team by programming the software to recognize the secondary data which have been ignored but had more priority. Important tasks were given more priority. This was handled by rebooting and reinitializing the system while restarting the selected programs in their execution.

Whenever 1201 or 1202 alarm was raised, the system was made to reboot and important processes ware only restarted and did not start the erroneously scheduled radar jobs. The engineers at NASA had extensively tested the software and played a vital role in accomplishment of the mission. The Apollo 11 mission won't be able to be successful if the computer was not able to recognize the problem.

As stated by Dylan Matthews, the radar and computer –aided systems used incompatible power supplies. A point worth noting was that the radar was not playing an important role in landing and transmitted plethora of data to the computer on the basis of the electrical random noise. This overloaded the CPU with garbage data leaving no space for critical tasks. The mechanism of recognizing the faulty behavior while rebooting and restarting only important and needed processes saved the mission from failure.

The Apollo 11 mission violates rate monotonic scheduling policy by assigning lower priority to the tasks that were occurring frequently.

Arguments for and against RM policy and anaylsis prevention of Apollo 11

The RM scheduling algorithm would have not prevented the Apollo mission from abort since it would have assigned the erroneous tasks like random radar data which was occurring at highest frequency would have been given maximum priority. This would have led to filling up of core sets eventually leading to system crash. They implemented Dynamic Scheduling Policy instead of using RMS policy since the execution of most important tasks was a priority while ignoring the tasks with less priority but higher frequency.

Assumptions made in paper by Liu and Lay land:

1. Run-time for each task is constant for that task and does not vary with time.
2. Any non-periodic tasks in the system are special; they are initialization or failure-recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.
3. Deadlines consist of run-ability constraints only- i.e each task must be completed before the next request for it occurs.

Three Aspects of Fixed Priority LUB derivation, I didn't understand

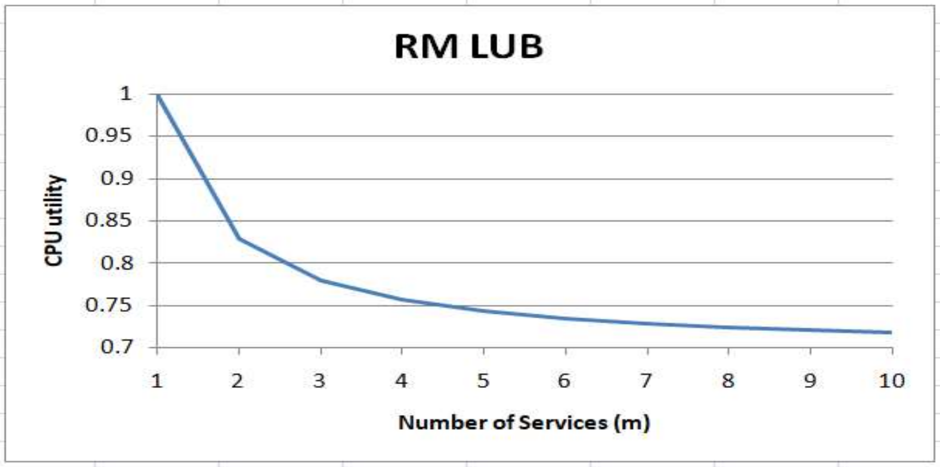
1. For calculation of Least Upper Bound CPU utilization for two tasks, they have considered two cases. The equations of C1 and C2 used are quite confounding. More details about the derivation of the equation would prove fruitful for ease of understanding of concepts.
2. The derivation of equation Cm = Tm – 2(C1 + C2 + …. + Cm-1) and its used is quite obscure and not covered anywhere in the paper.
3. A little background on series calculations and solving mathematical equations would have burgeoned better understanding and derivations of mathematical equations.

Graphical Plot and Description of Margin:

The graphical plot of No of services vs least Upper bound calculations considering equation

Upper Bound = m((2^(1/m)-1).

Services	RM LUB
1	1
2	0.828427
3	0.779763
4	0.756828
5	0.743492
6	0.734772
7	0.728627
8	0.724062
9	0.720538
10	0.717735
11	0.715452
12	0.713557
13	0.711959
14	0.710593
15	0.709412
16	0.708381
17	0.707472
18	0.706666
19	0.705946
20	0.705298
21	0.704713
22	0.704182
23	0.703698
24	0.703254
25	0.702846
26	0.702469
27	0.702121
28	0.701798
29	0.701497
30	0.701217



RM LUB is pessimistic scheduling policy which sacrifices throughput for margin.

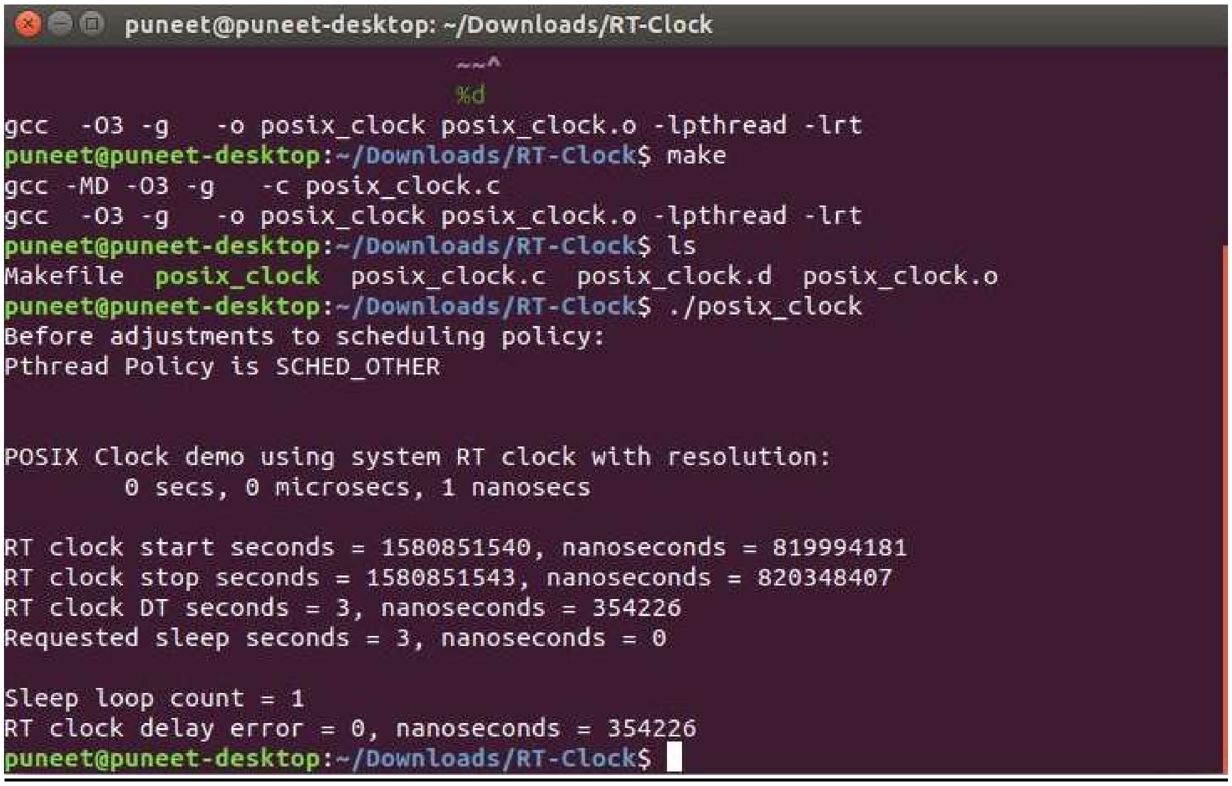
Q-3) [20 points] Download the code from http://mercury.pr.erau.edu/~siewerts/cec450/code/RT-Clock/ or from Canvas and build it on the Altera DE1-SOC, TIVA or Jetson board andexecute the code. Describe what it's doing and make sure you understand clock_gettime and how to use it to time code execution (print or log timestamps between two points in your code). 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? Do you believe the accuracy provided by the example RT-Clock code?

Answer:

Steps to Run the code:

- Download the RT-Clock zip file, extract and make the posix_clock.c to generate binary file.
- Execute the binary in Linux environment on Jetson Nano board.

Output:



```
puneet@puneet-desktop: ~/Downloads/RT-Clock
                                        ~~^
                                        %d
gcc  -O3 -g   -o posix_clock posix_clock.o -lpthread -lrt
puneet@puneet-desktop:~/Downloads/RT-Clock$ make
gcc -MD -O3 -g   -c posix_clock.c
gcc  -O3 -g   -o posix_clock posix_clock.o -lpthread -lrt
puneet@puneet-desktop:~/Downloads/RT-Clock$ ls
Makefile  posix_clock  posix_clock.c  posix_clock.d  posix_clock.o
puneet@puneet-desktop:~/Downloads/RT-Clock$ ./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 = 1580851540, nanoseconds = 819994181
RT clock stop seconds = 1580851543, nanoseconds = 820348407
RT clock DT seconds = 3, nanoseconds = 354226
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 354226
puneet@puneet-desktop:~/Downloads/RT-Clock$
```

Code Walkthrough:

- The Print_Scheduler function prints the scheduling mechanism implemented by the calling processes. The scheduling policies in a Linux System are broadly divided into Real Time and Non Real Time with SCHED_OTHER,SCHED_BATCH,SCHED_IDLE and SCHED_FIFO,SCHED_RR beneath them respectively.
- The scheduler manages the running processes and decides which process will be executed next in the kernel with each process having scheduling policy and static priorities associated with it The sched_setscheduler() function modifies those parameters while sched_getscheduler() queries the scheduling policy recognized by the calling processes pid.
- The sched_priority ranges from 1 to 99 with 99 being the maximum priority and 1 being the least. If sched_priority is not to be considered for scheduling, it must be set to 0.

- The scheduling policy used in this code is SCHED_OTHER with static_priority as 0 which is intended for processes that don't require special real-time considerations.
- DELAY_TEST(void *threadID) function calls the clock_getres() function that finds the resolution of real time clock(i.e. 1 nano second).The clock_gettime() function is called before nanosleep() is called and prior to waking up of thread. The start time and end time of the RT clock is passed to the delta function to calculate sleep time of the thread and logged into rtclk_data structure. The accuracy and precision of the clock has been improvised by passing rtclk_dt and sleep_requested structure in order to determine the drift in timing.
  The values are displayed by calling end_delay_test () function.
- The time of the clock is returned by clock_gettime () function by the means of clk_id argument passed. This function can be implemented for timing analysis for any section of the code in execution.

The steps for retrieving time taken by a particular section of code are enumerated below:

- Declare two structures rtclk_start_time, rtclk_stop_time of timespec type with values initialized to zero.
- Store the start time by calling the function clock_gettime () function and storing the return values in the defined rtclk_start_time structure.
- Store the end time by calling the function clock_gettime () function and storing the return values in the defined rtclk_stop_time structure.

RTOS vendors brag about three things:

1. Low Interrupt Handler Latency
    - Interrupt latency can be defined as time difference between interrupt request made by device and the first instruction to be executed in corresponding ISR.
    - Increasing the number and the length of the regions in which kernel disables interrupt increases the latency. Most real time operating systems disable interrupts so that the kernel may delay the handling of high priority requests which arrive in the disable window.
    - Thus the OS compromises the latency of the highest priority interrupt by disabling them in order to avoid problems incurred by handling of low priority ones.
    - RTOS that lessens the use of high latency instructions such as floating point arithmetic, string manipulators etc can minimize the interrupt latency. Also the size of interrupt service routine should be kept as small as possible just changing the process variables that drive the system.
    - All the above mentioned factors play a vital role which when implemented in RTOS lowers the Interrupt handler latency increasing the responsivity of the system. Hence RTOS is better tool for a system with Hard Real Time constraints compared to Linux based Operating System.

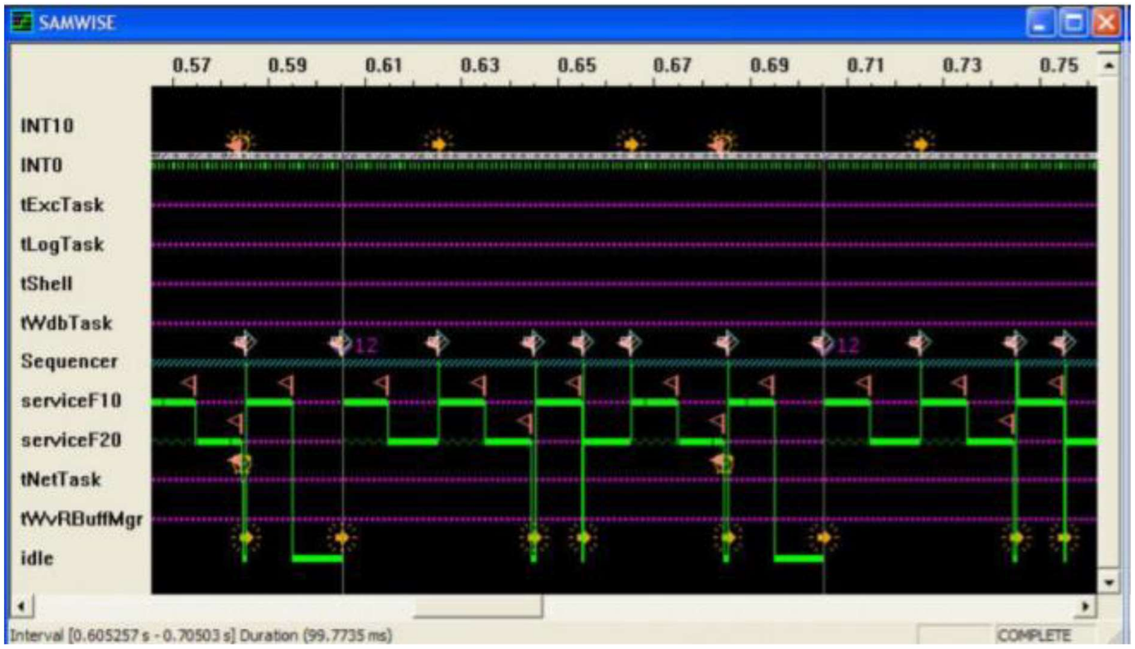
2. Context Switching Time
    - It is defined as the time taken by the kernel to transfer control of the processor from the running process to a ready process that is scheduled to run. It stores the status of the CPU registers and program counter of the running instruction and restores it after it's again scheduled for execution.
    - Thus, lower context switching time provides better system response.
    - Thus, the RTOS has less context switching time as compared to other Operating system which is the major reason for highlighting lesser context switching time by the RTOS vendors.

3. Stable timer services where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift.
    - The principle job of a timer is to provide accurate and precise timing required for operation of the processes. Hence the requirement of an accurate timer is crucial for development of real time system in order to meet the real time deadlines. The timer with minimum jitter and drift is desired for real time embedded applications. Hence a stable timer service is required to clock the services to meet hard real time constraints.
    - RTOS has a precise timer with the resolution of nano seconds which is greater than that provided by Linux kernel-based timers with a resolution of milli seconds.
    - Thus due to highly precise and accurate timing service that is provided by RTOS, interval timer interrupts, timeouts and relative time has low jitter and relative time having low jitter and drift from actual values which is often bragged by RTOS vendors.

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

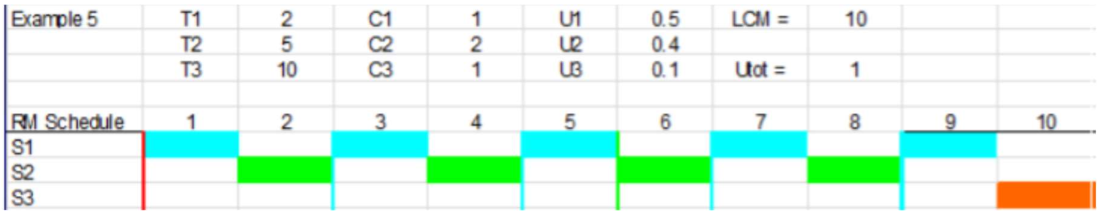
No, I don't believe the accuracy provided by the example RT-Clock code. This is because in our code, we implemented a timer for 3 seconds, however we did not achieve the exact time. The time turned out to be greater than 3 seconds. This can turn out to be dangerous in real time applications since, small offsets over time can lead to the system missing the deadlines which can result in catastrophic loss of life and property. A better approach could be to calculate the clock cycles required to execute the instructions and then calculating the time based on the clock frequency.

Q-4. This is a challenging problem that requires you to learn quite a bit about pthreads in Linux and to implement a schedule that is predictable. Download, build and run code in http://ecee.colorado.edu/~ecen5623/ecen/ex/Linux/simplethread/ and based on the example for creation of 2 threads provided by incdecthread/pthread.c, as well as testdigest.c with use of SCHED_FIFO and sem_post and sem_wait as well as reading of POSIX manual pages as needed - describe how you would attempt to implement Linux code to replicate the LCM invariant schedule implemented in the VxWorks RTOS which produces the schedule measured using event analysis shown below:



Interval [0.605257 s - 0.70503 s] Duration (99.7735 ms)

The observed timing above fits our theory for RM policy on a priority preemptive scheduling system as shown by the timing diagram below:



Example 5	T1	2	C1	1	U1	0.5	LCM =	10		
	T2	5	C2	2	U2	0.4				
	T3	10	C3	1	U3	0.1	Utot =	1		
RM Schedule	1	2	3	4	5	6	7	8	9	10
S1	🟦		🟦		🟦		🟦		🟦	
S2		🟩		🟩		🟩		🟩		
S3										🟧

You description should outline how you would implement code equivalent to the VxWorks synthetic load generation and schedule emulator. Code the Fib10 and Fib20 synthetic load generation and work to adjust iterations to see if you can at least produce a reliable 10 millisecond and 20 millisecond load on ECES Linux or a Jetson system (Jetson is preferred and should result in more reliable results). Describe whether your able to achieve predictable reliable results in terms of the C (CPU time) values alone and how you would sequence execution. Hints – You will find the LLNL (Lawrence Livermore National Labs) pages on pthreads to be quite helpful. If you really get stuck, a detailed solution and analysis can be found here, but if you use, be sure to cite and make sure you understand it and can describe well. If you use this resource, not how similar or dissimilar it is to the original VxWorks code and how predictable it is by comparison?

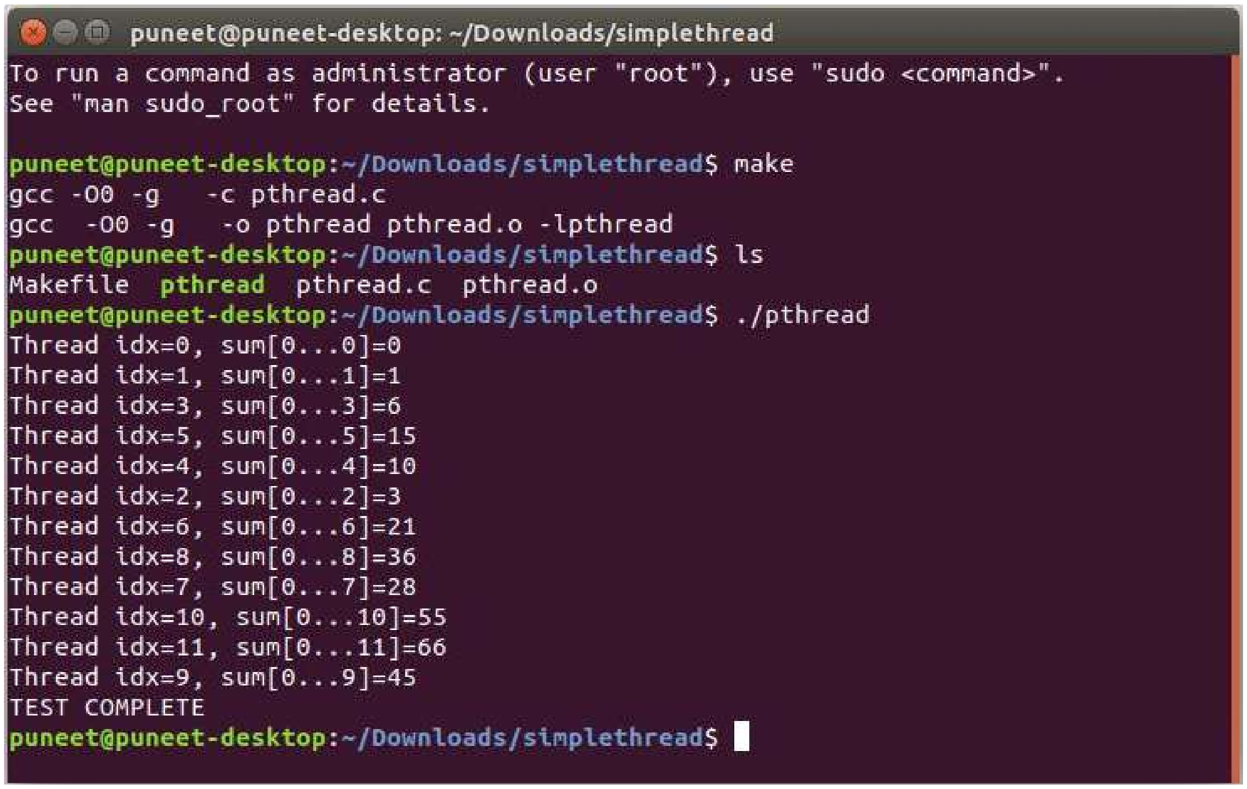
Answer:

ANALYSIS OF SIMPLETHREAD.C

Steps to run the code:

- Download the source code from the canvas link for the zipped simplethread   file containing pthread.c, make files and compile it.
- Run the binary in Jetson nano board running on ubuntu platform.

Output:



```
puneet@puneet-desktop: ~/Downloads/simplethread
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

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

Code Walk Through:

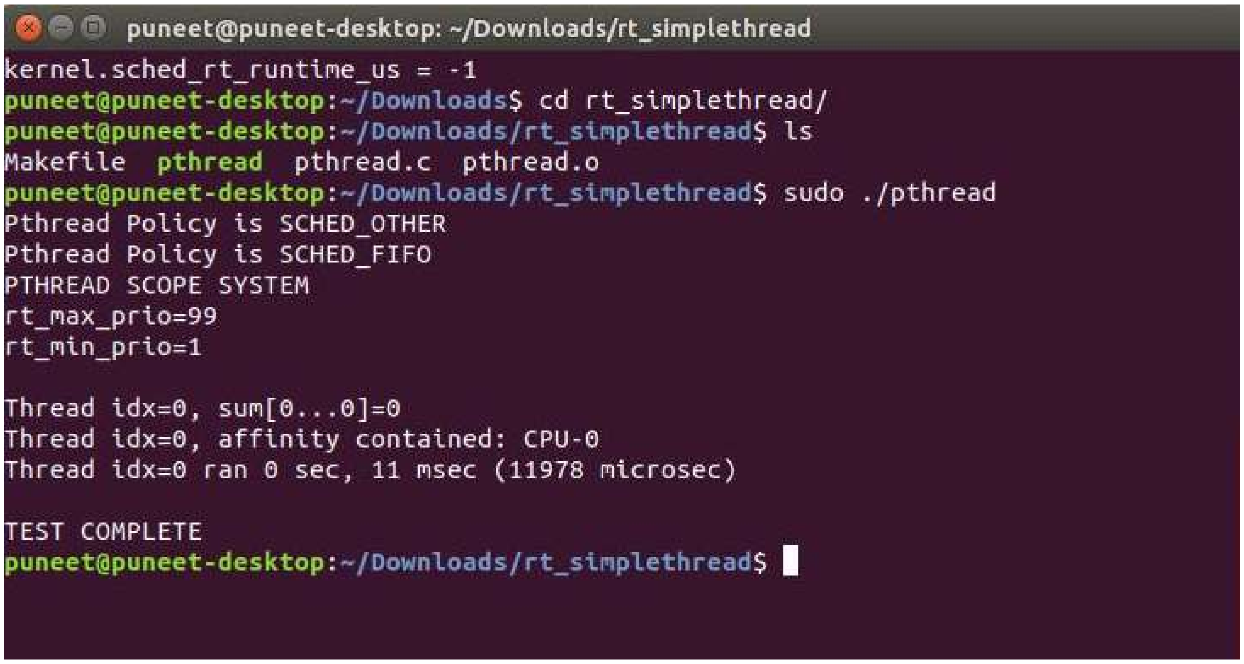
- The program intends to create threads within a process. Using the macro #define NUM_THREADS 12, we can change the number of threads that you want to create within the process.
- The structure threadParams_t is used to store the thread attribute which in this case is the user defined thread id.
- In main , we loop from 0 to 11 in order to create 12 processes using the function pthread_create(). The first parameter for the same is a pointer to the thread descriptor (in this code 'threads[i]'). The second is thread attribute which in this case is set to default attributes.The third is a routine which gets called when the thread is initialized and fourth provides parameter for the routine called above.
- All the thread calls backed by loopback function *counterThread which computes the sum from 1 to the 'Thread Index Id' + 1 of the particular thread.
- In order to restrict the execution of threads to a single core, binding it to a particular CPU core will prove fruitful.
- The main thread waits for all the child threads to exit through the function pthread_join().

ANALYSIS OF RT_SIMPLETHREAD.C:

Steps to run the code:

- Download the source code from the canvas link for the zipped rt_simplethread file containing pthread.c, make files and compile it.
- Run the binary in Jetson nano board running on ubuntu platform.

Output:



```
puneet@puneet-desktop: ~/Downloads/rt_simplethread
kernel.sched_rt_runtime_us = -1
puneet@puneet-desktop:~/Downloads$ cd rt_simplethread/
puneet@puneet-desktop:~/Downloads/rt_simplethread$ ls
Makefile  pthread  pthread.c  pthread.o
puneet@puneet-desktop:~/Downloads/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, 11 msec (11978 microsec)

TEST COMPLETE
puneet@puneet-desktop:~/Downloads/rt_simplethread$ 
```

Code Walk Through:

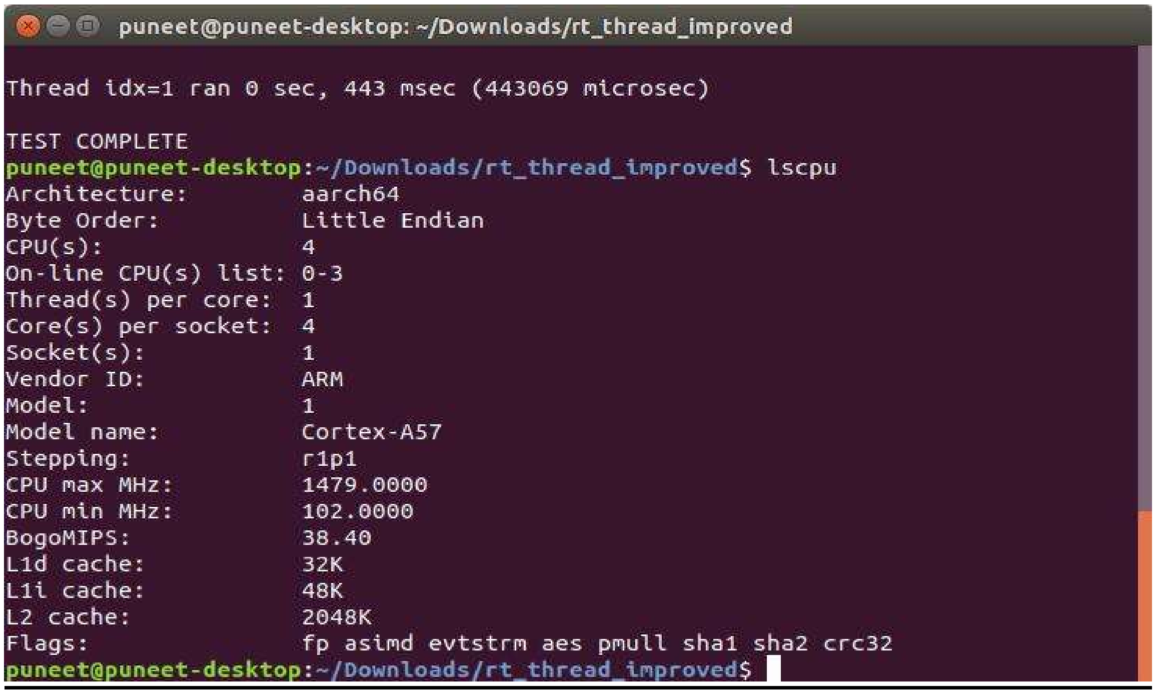
- Using the macro #define NUM_THREADS and #define NUM_CPUS the number of threads to be created and the number of CPU's to assigned respectively can be changed.
- In main, using the function CPU_ZERO(&cpuset), the cpuset variable is cleared so that it contains no CPU.
- We can add the required number of CPU's to the set using CPU_SET(). In the code, the cpuset is 0.
- The print_scheduler() prints the current scheduling policy used which in this case is SCHED_OTHER.
- Then, the main thread was provided the highest priority and the scheduling policy was modified to SCHED_FIFO using the sched_setscheduler().
- The print_scheduler now prints the updated scheduling policy i.e SCHED_FIFO.
- The scope of the attribute of main thread is checked using the pthread_attr_getscope() and printed.
- Number of threads specified by macro NUM_THREADS are created with set scheduling policy and affinity. The callback function for threads is counterThread() which computes the time required to implement Fibonacci series.
- The thread then prints the sum from 1 to the 'Thread Index Id' + 1 of the particular thread, the affinity contained and the CPU allocated to the particular thread.
- From the output it can also be observed that the thread took 11978 micro-second to execute.

ANALYSIS OF RT_THREAD_IMPROVED.C:

Steps to run the code:

- Download the source code from the canvas link for the zipped rt_thread_improved file containing pthread.c, make files and compile it.
- Run the binary in Jetson nano board running on ubuntu platform.

Output:

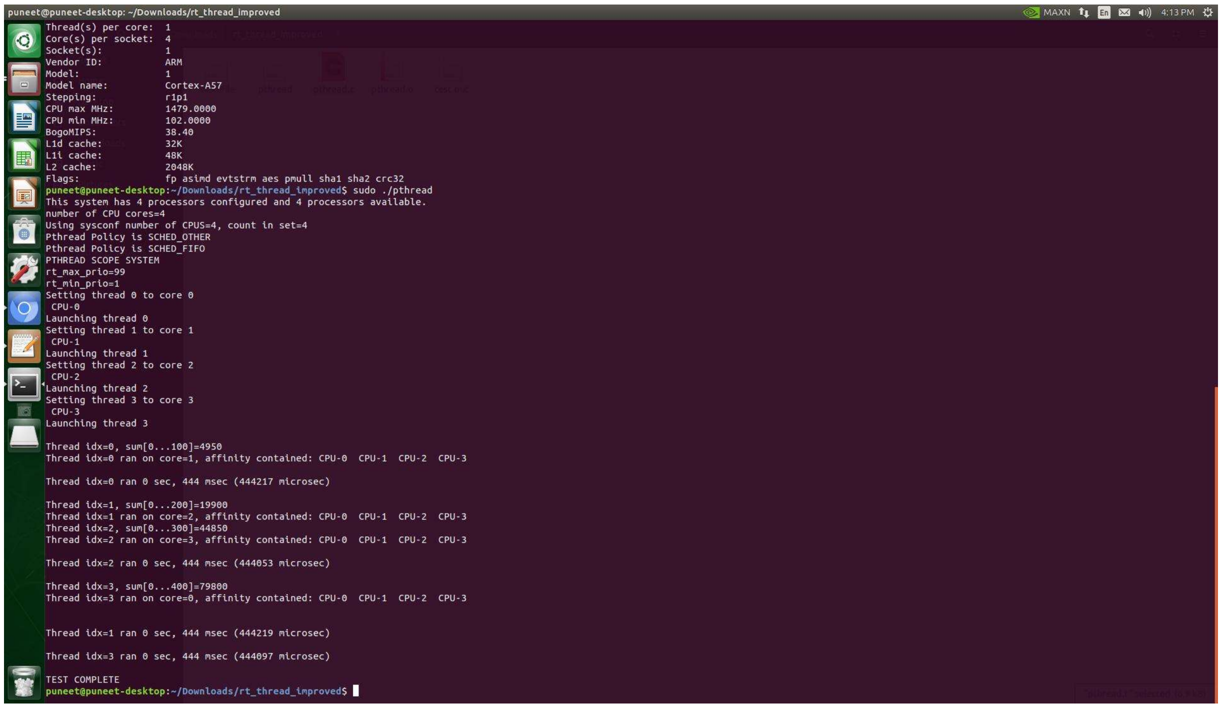


```
puneet@puneet-desktop: ~/Downloads/rt_thread_improved

Thread idx=1 ran 0 sec, 443 msec (443069 microsec)

TEST COMPLETE
puneet@puneet-desktop:~/Downloads/rt_thread_improved$ lscpu
Architecture:        aarch64
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           ARM
Model:               1
Model name:          Cortex-A57
Stepping:            r1p1
CPU max MHz:         1479.0000
CPU min MHz:         102.0000
BogoMIPS:            38.40
L1d cache:           32K
L1i cache:           48K
L2 cache:            2048K
Flags:               fp asimd evtstrm aes pmull sha1 sha2 crc32
puneet@puneet-desktop:~/Downloads/rt_thread_improved$
```

Checking the cores available



```
puneet@puneet-desktop: ~/Downloads/rt_thread_improved
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           ARM
Model:               1
Model name:          Cortex-A57
Stepping:            r1p1
CPU max MHz:         1479.0000
CPU min MHz:         102.0000
BogoMIPS:            38.40
L1d cache:           32K
L1i cache:           48K
L2 cache:            2048K
Flags:               fp asimd evtstrm aes pmull sha1 sha2 crc32
puneet@puneet-desktop:~/Downloads/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=1, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3

Thread idx=0 ran 0 sec, 444 msec (444217 microsec)

Thread idx=1, sum[0...200]=19900
Thread idx=1 ran on core=2, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3
Thread idx=2, sum[0...300]=44850
Thread idx=2 ran on core=3, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3

Thread idx=2 ran 0 sec, 444 msec (444053 microsec)

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


Thread idx=1 ran 0 sec, 444 msec (444219 microsec)

Thread idx=3 ran 0 sec, 444 msec (444097 microsec)

TEST COMPLETE
puneet@puneet-desktop:~/Downloads/rt_thread_improved$
```

Code Walk Through:

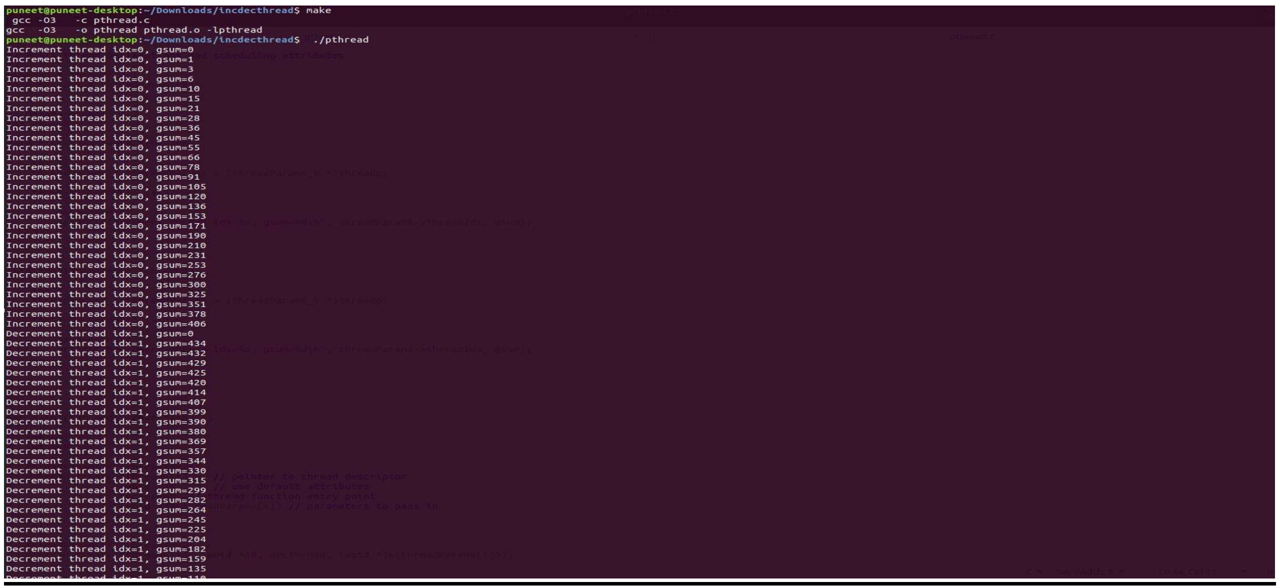
- This code implements multiple CPU core for calculation of Fibonacci series.
- The number of processors configured and the number of processors available are fetched using get_nprocs_conf() and get_nprocs() respectively.
- Using the number of processors configured the required number of CPU cores are set.
- The print_scheduler() prints the current scheduling policy used which in this case is SCHED_OTHER.
- Then, the main thread was provided the highest priority and the scheduling policy was modified to SCHED_FIFO using the sched_setscheduler().
- The print_scheduler now prints the updated scheduling policy i.e SCHED_FIFO.
- The scope of the attribute of main thread is checked using the pthread_attr_getscope() and printed.
- Each thread is assigned a CPU core , scheduling policy as FIFO and affinity.
- The counterThread () gets called for all the threads which computes the time required to implement Fibonacci series.
- Each thread also prints the sum from 1 to the ('Thread Index Id' + 1)*100 of the particular thread, the affinity contained and the CPU allocated to the particular thread.
- From the output we can observe that thread 0 was allocated core 1 and ran for a total of 444217 microsec, thread 1 was allocated core 2 and ran for a total of 444219 microsec, thread 2 was allocated core 3 and ran for a total of 444053 microsec, thread 3 was allocated core 0 and ran for a total of 444097 microsec.

ANALYSIS OF INCDECTHREAD.C:

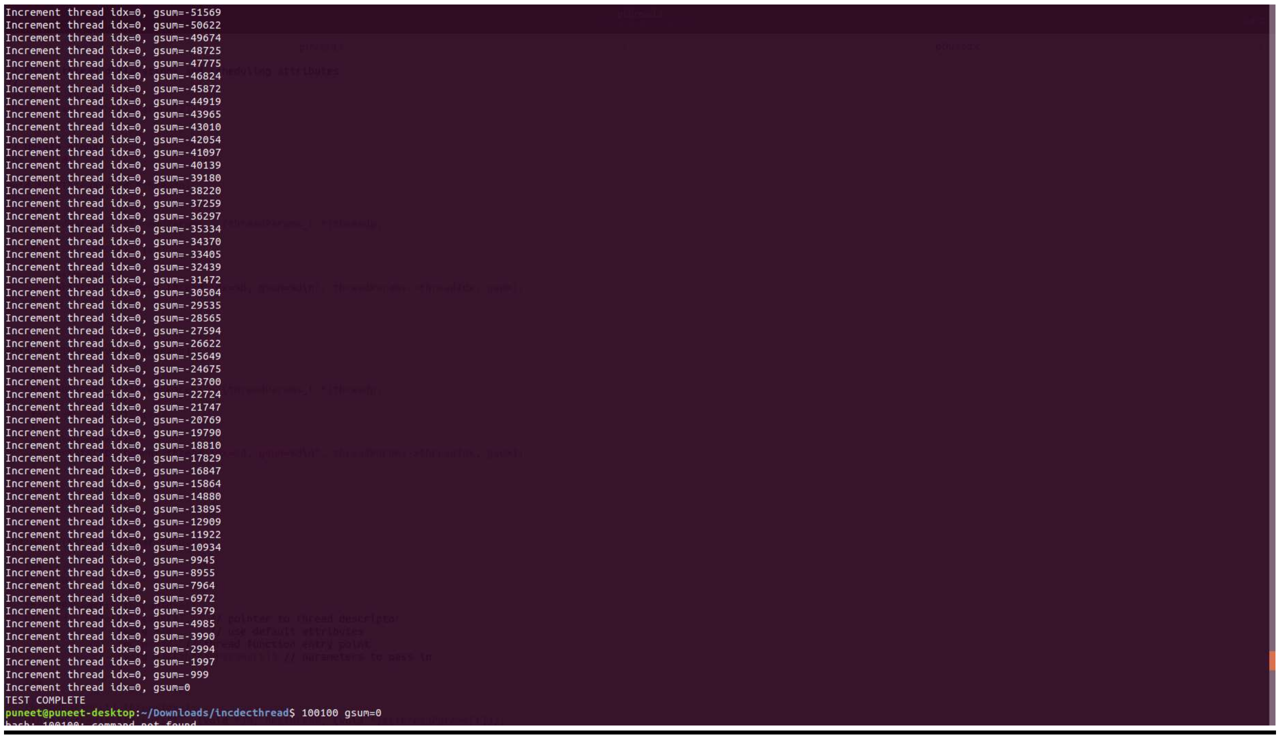
Steps to run the code:

- Download the source code from the canvas link for the zipped incdecthread file containing pthread.c, make files and compile it.
- Run the binary in Jetson nano board running on ubuntu platform.

Output:



```
puneet@puneet-desktop:~/Downloads/incdecthread$ make
 gcc -O3   -c pthread.c
gcc  -O3   -o pthread pthread.o -lpthread
puneet@puneet-desktop:~/Downloads/incdecthread$  ./pthread
Increment thread idx=0, gsum=0
Increment thread idx=0, gsum=1
Increment thread idx=0, gsum=3
Increment thread idx=0, gsum=6
Increment thread idx=0, gsum=10
Increment thread idx=0, gsum=15
Increment thread idx=0, gsum=21
Increment thread idx=0, gsum=28
Increment thread idx=0, gsum=36
Increment thread idx=0, gsum=45
Increment thread idx=0, gsum=55
Increment thread idx=0, gsum=66
Increment thread idx=0, gsum=78
Increment thread idx=0, gsum=91
Increment thread idx=0, gsum=105
Increment thread idx=0, gsum=120
Increment thread idx=0, gsum=136
Increment thread idx=0, gsum=153
Increment thread idx=0, gsum=171
Increment thread idx=0, gsum=190
Increment thread idx=0, gsum=210
Increment thread idx=0, gsum=231
Increment thread idx=0, gsum=253
Increment thread idx=0, gsum=276
Increment thread idx=0, gsum=300
Increment thread idx=0, gsum=325
Increment thread idx=0, gsum=351
Increment thread idx=0, gsum=378
Increment thread idx=0, gsum=406
Decrement thread idx=1, gsum=0
Decrement thread idx=1, gsum=434
Decrement thread idx=1, gsum=432
Decrement thread idx=1, gsum=429
Decrement thread idx=1, gsum=425
Decrement thread idx=1, gsum=420
Decrement thread idx=1, gsum=414
Decrement thread idx=1, gsum=407
Decrement thread idx=1, gsum=399
Decrement thread idx=1, gsum=390
Decrement thread idx=1, gsum=380
Decrement thread idx=1, gsum=369
Decrement thread idx=1, gsum=357
Decrement thread idx=1, gsum=344
Decrement thread idx=1, gsum=330
Decrement thread idx=1, gsum=315
Decrement thread idx=1, gsum=299
Decrement thread idx=1, gsum=282
Decrement thread idx=1, gsum=264
Decrement thread idx=1, gsum=245
Decrement thread idx=1, gsum=225
Decrement thread idx=1, gsum=204
Decrement thread idx=1, gsum=182
Decrement thread idx=1, gsum=159
Decrement thread idx=1, gsum=135
Decrement thread idx=1, gsum=110
```



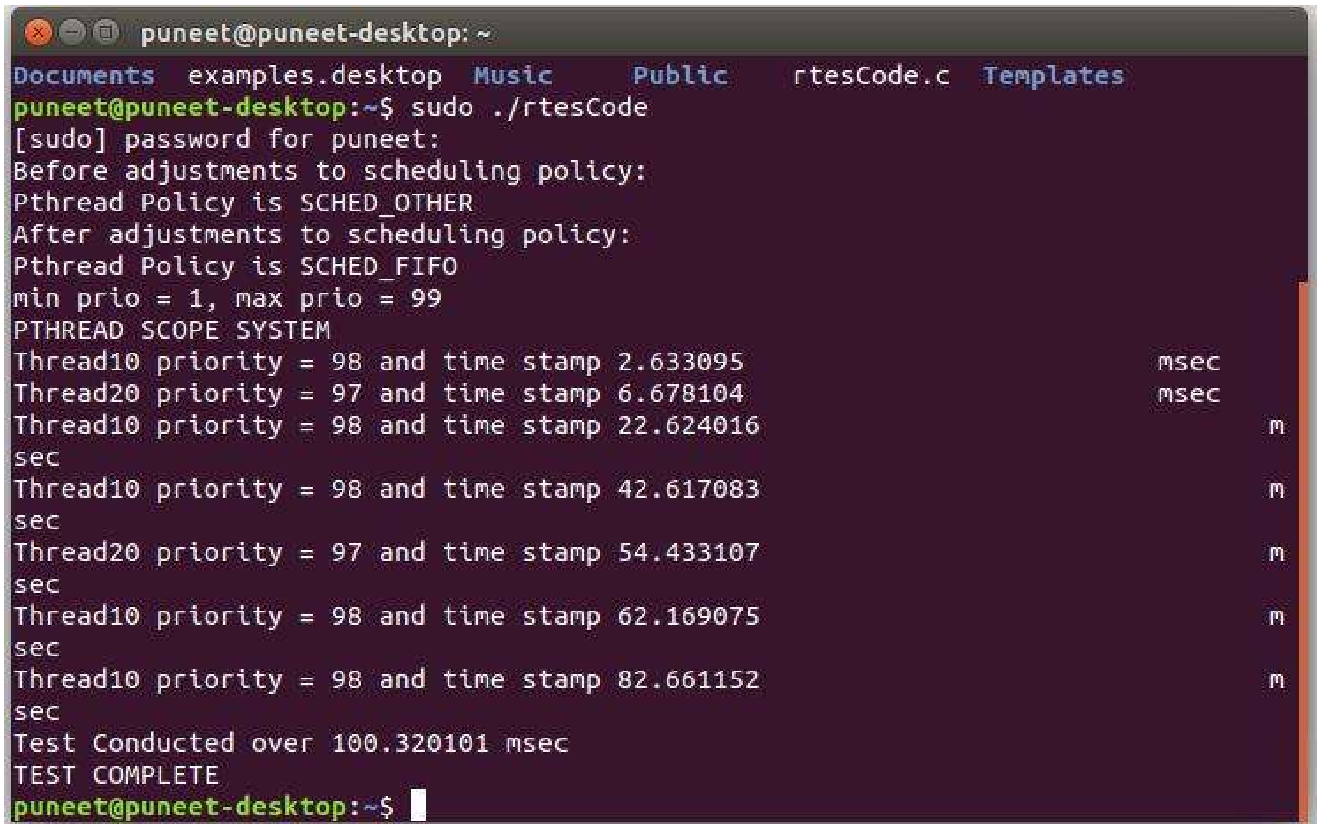
```
Increment thread idx=0, gsum=-51569
Increment thread idx=0, gsum=-50622
Increment thread idx=0, gsum=-49674
Increment thread idx=0, gsum=-48725
Increment thread idx=0, gsum=-47775
Increment thread idx=0, gsum=-46824
Increment thread idx=0, gsum=-45872
Increment thread idx=0, gsum=-44919
Increment thread idx=0, gsum=-43965
Increment thread idx=0, gsum=-43010
Increment thread idx=0, gsum=-42054
Increment thread idx=0, gsum=-41097
Increment thread idx=0, gsum=-40139
Increment thread idx=0, gsum=-39180
Increment thread idx=0, gsum=-38220
Increment thread idx=0, gsum=-37259
Increment thread idx=0, gsum=-36297
Increment thread idx=0, gsum=-35334
Increment thread idx=0, gsum=-34370
Increment thread idx=0, gsum=-33405
Increment thread idx=0, gsum=-32439
Increment thread idx=0, gsum=-31472
Increment thread idx=0, gsum=-30504
Increment thread idx=0, gsum=-29535
Increment thread idx=0, gsum=-28565
Increment thread idx=0, gsum=-27594
Increment thread idx=0, gsum=-26622
Increment thread idx=0, gsum=-25649
Increment thread idx=0, gsum=-24675
Increment thread idx=0, gsum=-23700
Increment thread idx=0, gsum=-22724
Increment thread idx=0, gsum=-21747
Increment thread idx=0, gsum=-20769
Increment thread idx=0, gsum=-19790
Increment thread idx=0, gsum=-18810
Increment thread idx=0, gsum=-17829
Increment thread idx=0, gsum=-16847
Increment thread idx=0, gsum=-15864
Increment thread idx=0, gsum=-14880
Increment thread idx=0, gsum=-13895
Increment thread idx=0, gsum=-12909
Increment thread idx=0, gsum=-11922
Increment thread idx=0, gsum=-10934
Increment thread idx=0, gsum=-9945
Increment thread idx=0, gsum=-8955
Increment thread idx=0, 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
puneet@puneet-desktop:~/Downloads/incdecthread$ 100100 gsum=0
bash: 100100: command not found
```

Code Walk Through:

- The program generates 2 threads which perform increment and decrement on a common unsafe global variable named gsum.
- pthread_create() is used to create the threads with default attributes and thread parameters.
- For one of the thread, the call-back function is incThread() and for the other it is decThread()
- The incThread() increments the value of global variable gsum (initial value 0). It prints the thread Id and the value of gsum. This loop is repeated till the time value of counter becomes equal to the COUNT specified (in this case it is 1000).
- The decThread() decrements the value of global variable gsum (initial value 0). It prints the thread Id and the value of gsum. This loop is repeated till the time value of counter becomes equal to the COUNT specified (in this case it is 1000).
- The main thread waits for the 2 threads to exit.

Implementation of lcm invariant code in linux system

The solution code has been referenced from an independent study conducted by Nisheeth Bhat titled 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 .



```
puneet@puneet-desktop: ~
Documents  examples.desktop  Music     Public    rtesCode.c  Templates
puneet@puneet-desktop:~$ sudo ./rtesCode
[sudo] password for puneet:
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
Thread10 priority = 98 and time stamp 2.633095                          msec
Thread20 priority = 97 and time stamp 6.678104                          msec
Thread10 priority = 98 and time stamp 22.624016                            m
sec
Thread10 priority = 98 and time stamp 42.617083                            m
sec
Thread20 priority = 97 and time stamp 54.433107                            m
sec
Thread10 priority = 98 and time stamp 62.169075                            m
sec
Thread10 priority = 98 and time stamp 82.661152                            m
sec
Test Conducted over 100.320101 msec
TEST COMPLETE
puneet@puneet-desktop:~$
```

The concepts used in implementation of the code in a Linux System are similar to that used in the VxWorks Real Time Operating System. The Linux system implements threads while VxWorks RTOS uses tasks concept to achieve the desired goal.

Use of threading in Linux system

Three threads Sequencer, Fib 10 and Fib 20 are created in the main process with the sequencer thread being assigned as highest priority. The priorities are assigned as Sequencer, Fib 10 and Fib 20 order in the decreasing order. Before creation, attributes and priorities of the threads are set and the remaining approach stays the same compared to VxWorks. The memory is mapped in a single space and pthread_create is used to create a new pthread.

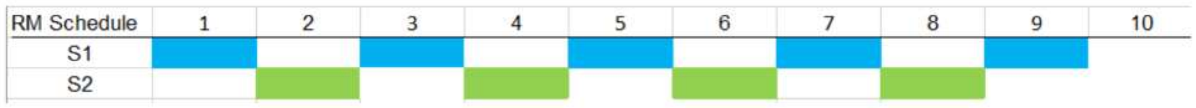
The code as referenced by Dr Sam Siewert implements thread model for code implementation similar compared to task model used in VxWorks RTOS. It initializes three threads main_param, testthread10; testthread20.The testthread10 is assigned higher priority than testthread20 since it occurs at more frequency adhering to rate monotonic scheduling policy. The scheduler is set SCHED_FIFO policy. sched_setscheduler () is used to start the scheduler for the main thread. The scheduling parameters are set using pthread_attr_setschedparam () function with calling process pid passed as argument.

The pthread_join () function waits for the thread specified by thread to terminate. The code waits for pthread sleep using usleep () function.The SEM_POST () API is used to unlock the semaphore while sem_wait ()

function is used lock the semaphore in order to provide mutual exclusion for access to the threads testthread10, testthread20.The usleep () function is used to implement suspension of execution for the desired interval in microseconds.

Thus the Linux System implements pthread model and semaphores for execution of threads to get the same results to that of VxWorks RTOS.

Overall good description of challenges and test/prototype work



RM Schedule	1	2	3	4	5	6	7	8	9	10
S1	🟦		🟦		🟦		🟦		🟦	
S2		🟩		🟩		🟩		🟩		

We have two services S1 and S2 with T1 =20 ms and T2 = 50 ms and C1=10 ms and 20 ms.

The rate monotonic scheduling policy is implemented in which the service with highest frequency have been assigned highest priority. The S1 comes in every 20 milli seconds taking 10 milli seconds for computation while S2 comes every 50 milli seconds with computational time of 20 milli seconds. After scheduling of S1, the S2 is allocated next time slot to the executed inorder to meet the deadline of 50 milli seconds. The challenges faced were:

- Implementation of semaphores and pthread required exhaustive reading of manpages.
- The iteration count was based on trial and testing approach in order to have accurate execution time of 10 ms and 20 ms.

Description of key RTOS/Linux OS porting requirements

Threading vs Tasking

In Linux, the concept of threading is used whereas in VxWorks RTOS there are Tasks. One major difference between task and thread is that tasks share the same memory over the system whereas threads have same memory within a process.

In linux, to create a thread we use the API pthread_create. The new thread starts executing by invoking the callback function with the arguments specified. Following is the API:

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);

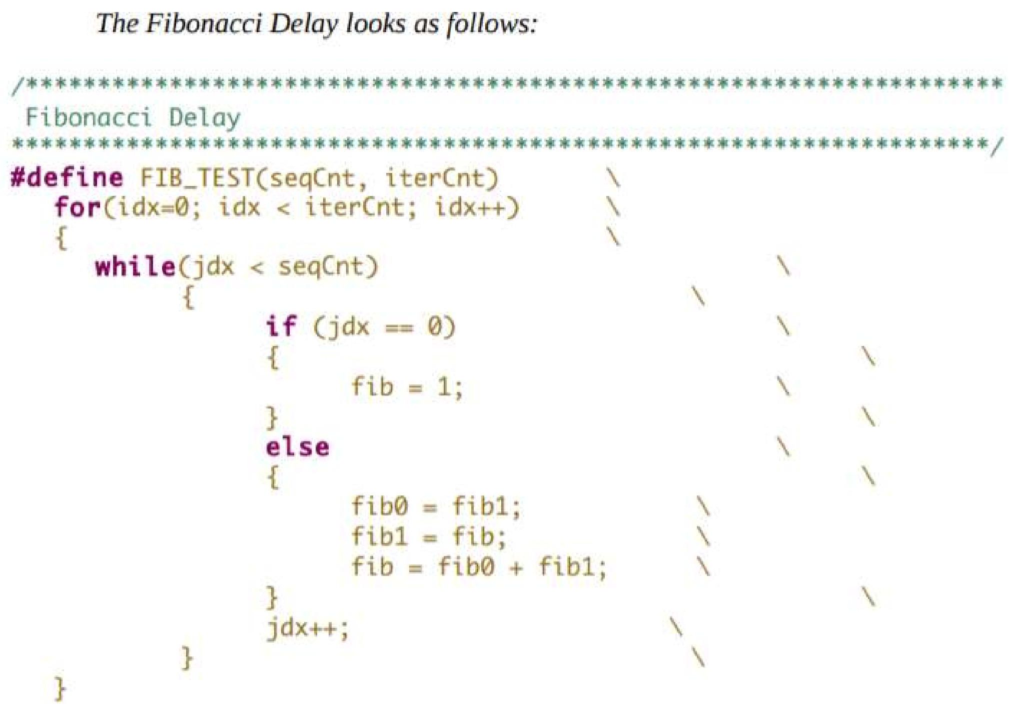
In VxWorks, new task is created using the API taskSpawn.

Semaphore wait and sync

The basic use of semaphore is to synchronize access to a common resource in a multi processing environment. In order to implement semaphore in linux, we first initialize it using sem_init(). In order to acquire the semaphore, sem_wait() is used and in order to release a semaphore sem_post() is used.In VxWorks semGive and semTake is used.

Synthetic workload Generation along with analysis and adjustment on test system

The workload can be generated using FIB_TEST MACRO on the desired core. The MACRO calculates the Fibonacci sequence based on the arguments (iteration count and sequence count) passed to the MACRO while being called upon. By varying the arguments, desired load might be generated on the core for specified interval of time consuming more number of instructional cycles for computation.



The Fibonacci Delay looks as follows:

```
/*******************************************************************
 Fibonacci Delay
*******************************************************************/
#define FIB_TEST(seqCnt, iterCnt)       \
   for(idx=0; idx < iterCnt; idx++)     \
   {                                    \
      while(jdx < seqCnt)                           \
            {                                 \
                  if (jdx == 0)                     \
                  {                                       \
                        fib = 1;                    \
                  }                                       \
                  else                              \
                  {                                       \
                        fib0 = fib1;          \
                        fib1 = fib;           \
                        fib = fib0 + fib1;    \
                  }                                       \
                  jdx++;                     \
            }                                 \
   }
```

The code (referenced from Dr. Sam Siewert) dynamically computes the time taken by the core while executing the FIB_TEST MACRO. Depending on this measured time, it takes 10 ms to compute Fib 10 and 20 ms to compute Fib20 thus occupying CPU core for the desired time period with increasing number of processor cycles.

The VxWorks code implements the same logic for duplicating the basic sequence in order to release execution time of the threads. The program is terminated upon execution of three threads in the time rang of 0-100, 100 -200 and 200-300 ms as depicted in the timing diagram.

REFERENCES:

- VxWorks code from Sam Siewert
- NASA_Apollo_11 Paper
- Context switching: https://www.sciencedirect.com/topics/engineering/context-switch
- Rate Monotonic Scheduling Concepts: https://www.embedded.com/electronics-blogs/beginner-s-corner/4023927/Introduction-to-Rate-Monotonic-Scheduling
- Geeksforgeeks for concepts on pthreads, semaphores and mutexes.