Department of Electrical and Computer Engineering

University of Colorado at Boulder

ECEN5623 - Real Time Embedded Systems

Exercise 1

Submitted by

Parth Thakkar — Tirth Patel

Submitted on February 10, 2024

Contents

List of Figures

List of Tables

PDF is clickable

1 Question 1

Q: [20 points] 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).

(a) **Q: 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].**

Answer: Rate monotonic scheduling is the hard real time scheduling of choice. This is due to the fact that its a static scheduling policy which assigns higher priority for more frequent tasks in a system ,so that overall utilization of the system nears 100 percent. Also RM policy shows more deterministic behavior during all overload situations .Moreover RM, shows O(1) time complexity while adding new tasks in the list .

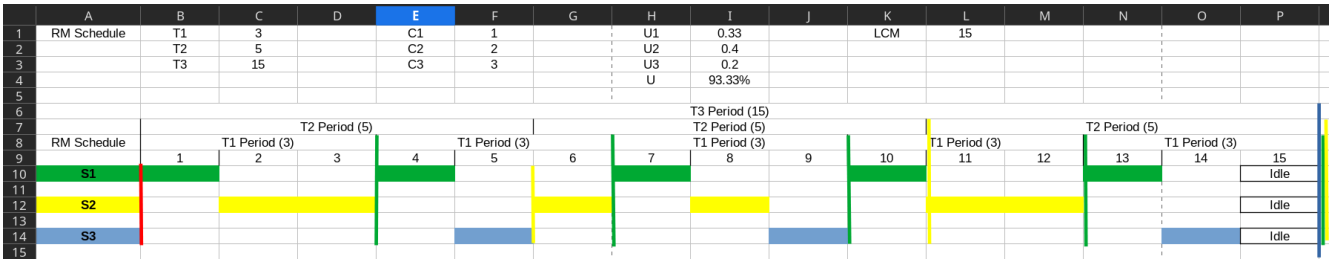

Figure 1: Rate monotonic Schedule

(b) **Q: 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).**

Answer:

S1, being the highest priority, runs first at the beginning of every 3 ms interval.

S2 runs after S1, fitting its execution around the 5 ms mark, respecting its priority over S3 but after S1.

S3, with the lowest priority, fits in where S1 and S2 allow, based on its longer period and lower frequency requirement.

The schedule is feasible because the total execution time of each service is less than the total period assigned to each service. Secondly, there has been no overlapping in execution times. Thus, this schedule is safe with no missing deadlines and each service completes its execution before its next period begins.

(c) **Q: What is the total CPU utilization by the three services?**

Answer:

CPU Utilization by 3 services:

$$U = \left(\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3}\right)$$
$$U = \left(\frac{1}{3} + \frac{2}{5} + \frac{3}{15}\right)$$
$$U = 0.33 + 0.4 + 0.2$$
$$U = 93.33$$
$$\boxed{U = 93.33\%}$$

- The CPU Utilization upon calculation is 93.33%.

2 Question 2

Q: [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.

(a) Q: Summary of the Apollo 11 Lunar Story:

Answer: The Pathfinder's applications were scheduled by the VxWorks RTOS. Since VxWorks provides pre-emptive priority scheduling of threads, tasks were executed as threads with priorities determined by their relative urgency.

Apollo Lunar Lander is a special type of computer, Apollo Guidance Computer. It is a digital computer that provides computational and electronic interfaces for guidance, navigation, and control. The first computer was based on the Silicon IC. Installed on the board of the Apollo Command Module and the Lunar Module. This computer utilizes the AGC, a 16-bit word length, and uses special ROM for core rope memory. Astronauts communicated with this AGC using the numeric display and keyboard called DSKY. During the Apollo Mission, AGC terminated the "Error 1201/1202", which indicated the computer overload. Multiple errors were caused by the Lunar Module's rendezvous radar left "ON" during the landing. The consequence of leaving the radar "ON" led to the comparatively receiving more data than the computer could handle. The software on AGC was able to prioritize the most critical tasks and ignore lesser important ones, which led to proceeding for a safe landing. This is a testament to the robustness and reliability of the AGC Software. Thanks to the team led by Chief Software Engineer, Margaret Hamilton. The 1201/1202 errors were ultimately caused by the human error. The astronauts forgot to turn the rendezvous radar "OFF".

The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus synchronized with mutual exclusion locks (mutexes). Other higher priority threads took precedence when necessary, including a very high priority bus management task, which also accessed the bus with mutexes. Unfortunately in this case, a long-running communications task, having higher priority than the meteorological task, but lower than the bus management task, prevented it from running.

Soon, a watchdog timer noticed that the bus management task had not been executed for some time, concluded that something had gone wrong, and ordered a total system reset. (Engineers later confessed that system resets had occurred during pre-flight tests. They put these down to

a hardware glitch and returned to focusing on the mission-critical landing software.)

(b) **Q: What was the root cause of the overload and why did it violate Rate Monotonic policy?**

Answer: • Priority inversion is the main root cause of the overload. A low-priority task was given the priority over a high-priority task. This caused a delay in the high-priority task's execution time. Thus, a violation of the Rate Monotonic Policy was caused as the execution time of high-priority tasks was exceeded than anticipated. Thus, caused an overloading situation.

(c) **Q: Now, read Liu and Layland's paper which describes Rate Monotonic policy and the Least Upper Bound – they derive an equation that advises a margin of approximately 30% of the total CPU as the number of services sharing a single CPU core increases.**

Answer:

(d) **Q: Plot this Least Upper bound as a function of several services.**

Answer: Least Upper Bound defines a safety margin that ensures that the system can handle the worst-case scenario of task execution times or WCET.

Number of Services	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Least Upper Bound (in %)	30	45	57	66	73	79	83	87	90	93

Table 1: Least Upper Bound

As the LUB % increases, the number of services sharing a single-CPU core also increases and so does the load with each increment of the service. However, the system's safety margin reduces as the latter parameters increases.
30% margin suggested by Liu-Layland ensures that the system can in fact handle the worst-case scenarios of task execution times. Thus, preventing the overload situation and ensuring real-time guarantees.

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

Answer: • The THREE key assumptions they make, are:

i. A. There are no precedence constraints between the tasks. Each task can execute independently of each other.
   B. Tasks have Worst-case Execution Time (WCET).
   C. Tasks are periodic and have fixed intervals between their times of beginning.

ii. The THREE aspects of Fixed Priority Least Upper Bound that were not quite understandable, are:
   A. How do they handle tasks with deadlines that are different from their periods? What is the key analysis behind this? They assume that the deadline is the same as the period, but this may not be true for every other case.
   B. How do they account for the context-switching overhead in their LUB equation? The context-switching overhead is not explicitly considered in the equation.
   C. How do they handle the task jitter-variation in the start time of execution of the task? Jitter can affect the response time of a task and may not be accounted for.

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

Answer: • The RM analysis may or may not have prevented the Apollo 11 1201/1202 errors. To support this, we present two arguments:

i. RM analysis may not have prevented the errors and mission abort because of the root cause – a Software Overload bug due to the priority inversion problem. Low-priority task was given higher priority instead of the high-priority tasks. The execution time of high-priority was extended more than anticipated and thus, a delay was caused. Priority Inversion was the problem that needed to be handled preemptively, and the RM analysis may not have been sufficient to prevent the 1201/1202 errors, and the potential mission abort.

ii. RM analysis can determine whether the system is schedulable or not by analyzing the priorities and the periods. It could have identified the error of overload and thrown out a warning about the possible consequence. By reinforcing RM analysis, it could provide a warning to the scheduler to take corrective measures before the system becomes overloaded. Apollo 1201/1202 errors happened due to an overloaded system, RM analysis could have warned the control team before the overload situation ever took place. However, unfortunately, the priority inversion is not accounted for in the RM Analysis.

3 Question 3

Q: [20 points] Download the RT-Clock code from Canvas and build it on a Jetson board, Raspberry Pi, or Altera DE1- SOC board and execute the code.

(a) **Q: Describe what the code is 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.**

Answer: The RT-clock code is a simple program demonstrating the usage of POSIX real-time clock for measuring the time of execution of delay loop. Firstly, the program sets up the sleep time for a total of 3 secs and then enters a loop where it requests the specified time to sleep using the function – 'nanosleep()'. The remaining sleep time is checked, if there is still time remaining, the process goes back to sleep. This particular process is repeated until the requested (expected) sleep time has been successfully met or until the maximum number of sleep calls has been reached. Now, the program calculates the difference between the requested and the actual sleep time and prints the results, at the end.

clock_gettime() function is used for acquiring the current time in nanoseconds from the POSIX real-time clock. It takes two arguments: clock ID and a pointer to the timespec structure. The clock ID specifies the clock to use and the pointer to the timespec structure is used for storing the current time.

(b) **Q: 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 is each important?**

Answer:

i. **Low Interrupt handler Latency:**
This is a crucial real-time systems parameter as it allows the system to respond quickly to any external event. A lower interrupt handler latency means the system can begin processing an interrupt as soon as it is received. This is often critical in time-bound systems.

ii. **Low Context Switch Time:**
This is important in real-time systems because it allows the system to switch between tasks quickly. A lower context switch time renders to be fruitful for the system as it can switch between the tasks without incurring a large overhead, this helps to ensure that tasks meet their deadlines.

iii. **Stable Timer Services:**
These are crucial for real-time systems as they provide a predictable and consistent way to measure time. Low jitter and drift ensure that the timer services are working accurately and that they can produce reliable consistent time measurements.

(c) **Q: Do you believe the accuracy provided by the example RT-Clock code? Why or why not?**

Answer:

i. The output of code RT-Clock on my system (processor AMD Ryzen 5, with debian base) is,

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

RT clock start seconds = 1707627090, nanoseconds = 724248556
RT clock stop seconds = 1707627093, nanoseconds = 724282599
RT clock DT seconds = 3, nanoseconds = 34043
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 34043
parth@rog:~/Work/All_Data/university/RTES_ECEN5623/Assignments/Exercise 1/RT-Clock$ sudo ./posix_clock
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO


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

RT clock start seconds = 1707627099, nanoseconds = 643996070
RT clock stop seconds = 1707627102, nanoseconds = 644031072
RT clock DT seconds = 3, nanoseconds = 35002
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 35002
parth@rog:~/Work/All_Data/university/RTES_ECEN5623/Assignments/Exercise 1/RT-Clock$
```

Figure 2: Rate monotonic OutPut On my linux

error was around 34000 - 36000 which is still feasible for some tasks

ii. The output of the RT_clock code on Jatson nano is around 202446ns which is much larger than my system, which would not be feasible for some system

```
parthishere@nano:~/Downloads/Exercise 1/RT-Clock$ sudo ./posix_clock
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO


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

RT clock start seconds = 1639128049, nanoseconds = 83158323
RT clock stop seconds = 1639128052, nanoseconds = 83615769
RT clock DT seconds = 3, nanoseconds = 457446
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 457446
```

Figure 3: Rate monotonic OutPut On Jatson Nano

The precision of the 'nanosleep()' function and the POSIX real-time clock will determine how accurate the RT-Clock code is. POSIX real-time clock is based on the hardware timer and the accuracy is possible on the hardware timer's precision. The accuracy of the scheduler of the Operating system and the underlying hardware are more factoring parameters that affect the nanosleep() function. This results in the RT-clock code's accuracy to differ based on the machine

it is compiled and run on. Thus, this code sufficiently explains the usage of the necessary functions but should not be deployed in a production setting without testing.

Further, the code measures the execution time of the loop using a delay loop and nanosleep() function. There are better and more precise ways to measure time as this may not be the most precise way to do so.

Thus, the RT-Clock's code is not accurate enough for some real-time applications. On top of that, it is necessary to examine and consider the needs of the application and conduct robust testing to ensure the accuracy of the required level is achieved.

4 Question 4

Q: [40 points] This is a challenging problem that requires you to learn a bit about Pthreads in Linux and to implement a schedule that is predictable.

(a) Q: Download, build and run code in the following three example programs: simplethread, rt_simplethread, and rt_thread_improved and briefly describe each and output produced. [These example programs can also be found on Canvas] [Note that for real- time scheduling, you must run any SCHED_FIFO policy threaded application with "sudo" – do man sudo if you don't know what this is]..

Answer:

i. **rt_simple_thread**

A. Sets up a thread with specific attributes to run with this real-time policy(SCHED_FIFO). The thread's priority is set just below the maximum to ensure it's treated as a high-priority task.

B. The thread does some work: first, it calculates a simple sum for each thread IDs, and then it calculates Fibonacci numbers to simulate a more intensive task.

C. Prints out information about what the thread did, including how long it took (around 7 milliseconds in this case) and which CPU it ran on (CPU-0, because the program specifies to run on one CPU).

D. Here are the screenshots of code running with 1 thread and setting threads to 12

```
make: Nothing to be done for 'all'.
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, 6 msec (6203 microsec)

TEST COMPLETE
parth@rog:~/Work/All_Data/university/RTES_ECEN5623/Assignments/Exercise 1/rt_simplethread$
```

Figure 4: Simple RT thread code with one thread

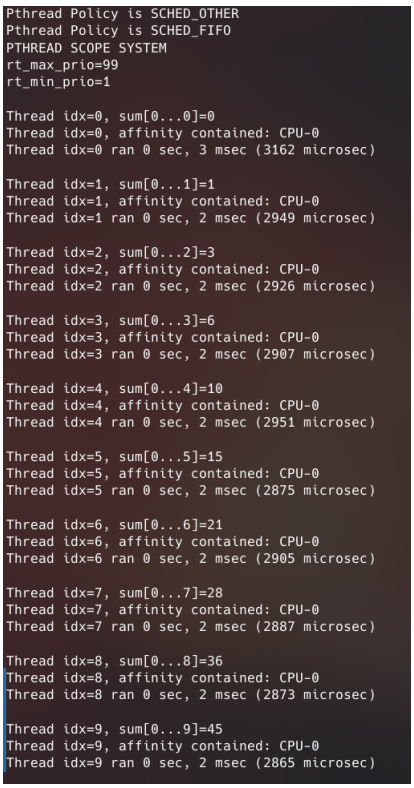

Figure 5: Simple RT thread code with 12 threads

E. **The output of this code is in the same folder named as rt_simplethread.txt and rt_simple12thread.txt (text file)**

ii. **simplethread**

A. Sets up a thread without any attributes in the main code

B. The output shows the result of each thread's calculation, that each thread successfully calculates the sum of numbers up to its index. The order of the output might vary between program runs due to the scheduler's decisions on thread execution order. (we don't have delay in our thread task or our thread task is not that intensive thats why we are getting same order everytime)

C. Here are the screenshots of output of the code,

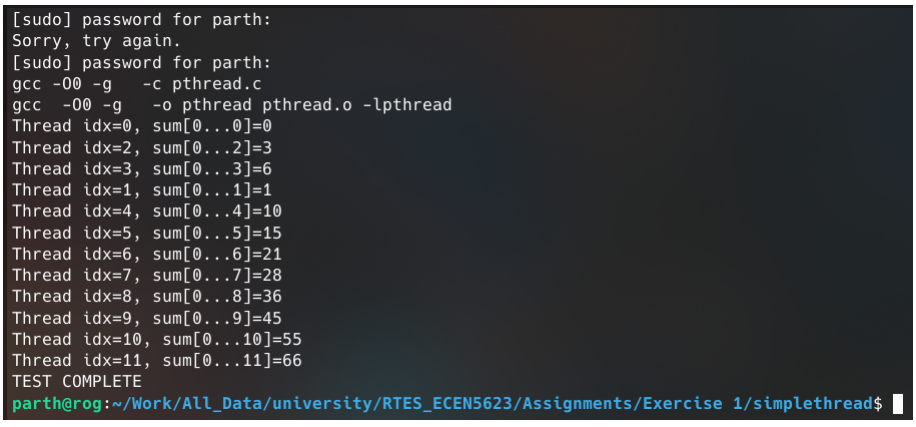

Figure 6: Simplethread code with 12 threads

D. **The output of this code is in the same folder named as simplethread.txt (text file)**

iii. **rt_thread_improved**

A. It begins by determining the number of processors available and configured on the system, aiming to utilize this information for setting CPU affinity for threads.

B. CPU Affinity: This code explicitly sets CPU affinity for each thread, ensuring that threads are bound to specific processors. This is a step further in optimizing for real-time performance by reducing the likelihood of CPU cache misses and context switches. Number of Processors Utilized: It dynamically checks and utilizes the number of processors available on the system, making it adaptable to run on different hardware configurations.

C. Prints out information about what the thread did, including how long it took (around 3 milliseconds in this case) and which CPU it ran on

D. Here are the screenshots of output of the code

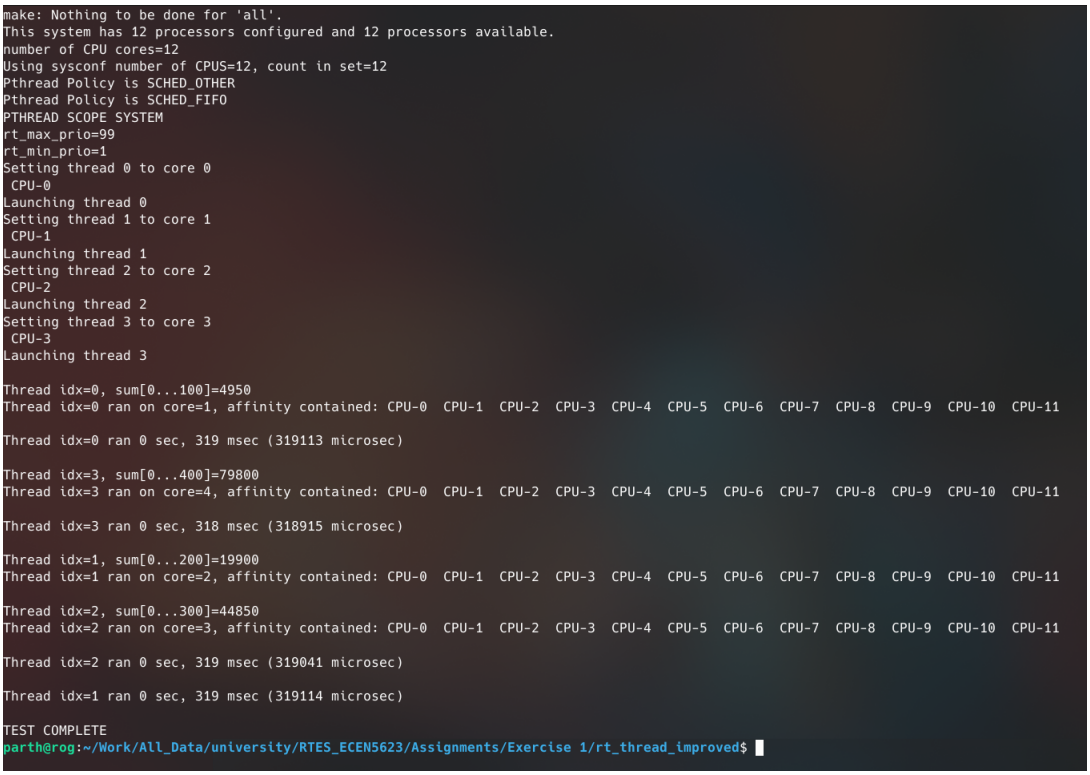

Figure 7: RT thread improved with threads running on all cores

E. **The output of this code is in the same folder named as rt_thread_improved.txt (text format)**

(b) **Q: Based on the examples for creation of 2 threads provided by incdecthread/pthread.c. Describe the POSIX API functions used by reading POSIX manual pages as needed and commenting your version of this code. Note that this starter example code - testdigest.c is an example that makes use of and sem_post and sem_wait and you can use semaphores to synchronize the increment/decrement and other concurrent threading code. Try to make the increment/decrement deterministic (always in the same order). You can make thread execution deterministic two ways – by using SCHED_FIFO priorities or by using semaphores. Try both and compare methods to make the order deterministic and compare your results.**

Answer: Summery of the code for semaphore :

i. gsum: A global variable that both threads will modify.

ii. semaphore: A semaphore used to control access to gsum, ensuring that one thread completes its operation before the other begins.

iii. By using a semaphore with an initial value of 1, the program ensures that only one thread can modify gsum at a time, making the operation order deterministic: first incrementing, then decrementing.

iv. This guarantees that gsum will first be incremented by the sum of numbers from 0 to COUNT-1, and then decremented by the same amount, ending with gsum being 0.

v. The output of this code is in the same folder named as output_sem.txt (text format)

Summery of the code for making code deterministic by using SCHED_FIFO :

i. gsum: A global variable that both threads will modify.

ii. Scheduling and CPU Affinity:

The program attempts to set a real-time scheduling policy (SCHED_FIFO) for the main process and applies specific priorities to each thread. This policy is a FIFO scheduling algorithm for real-time tasks, where a higher-priority task preempts lower-priority tasks and runs to completion unless it yields, is blocked, or a higher-priority task becomes runnable.
CPU affinity is set for each thread to run on a specific CPU core, aiming to reduce context switching and potentially increase performance predictability.

iii. Execution Flow in main:

Initializes thread attributes, sets the real-time scheduling policy (SCHED_FIFO), priorities, and CPU affinity for each thread.
Creates two threads, one for incrementing and the other for decrementing gsum. Waits for both threads to complete their execution.

iv. This guarantees that gsum will first be incremented by the sum of numbers from 0 to COUNT-1, and then decremented by the same amount

v. The output of this code is in the same folder named as output_sched_fifo.txt (text format)

(c) **Q: Describe how you would attempt to implement Linux code to replicate the LCM invariant schedule implemented in the VxWorks RTOS (sequencers/lab1.c) which produces the schedule measured using event analysis shown below:**

Answer: Here's a structured approach to achieve the schedule:

i. First, identify the periodic tasks to be scheduled, including their execution periods and computational requirements. For example, if in VxWorks you have tasks running at 10ms and 20ms intervals, you'll replicate this setup.

ii. Determine the LCM of all task periods to establish the length of the scheduling cycle. This cycle will repeat indefinitely, and tasks will be scheduled within this framework.

iii. For each task, create a POSIX thread (pthread_t). Threads provide the execution context for each task.

iv. use a real-time scheduling policy (SCHED_FIFO for priority and first in first out approach) using pthread_attr_setschedpolicy() to ensure tasks run according to their priority, which is determined by their periods (Rate Monotonic Scheduling).

v. Assign priorities based on the Rate Monotonic principle—shorter periods get higher priorities. Use pthread_attr_setschedparam() to set these priorities within the thread attributes.

vi. to improve predictability, bind each thread to a specific CPU core using pthread_attr_setaffinity_np().

vii. Semaphore Synchronization:
Initialize semaphores to control the execution start of each task within its period. Semaphores ensure that tasks do not overrun their allocated time and that tasks with longer periods do not start before tasks with shorter periods have completed their execution cycle.

viii. Timing and Execution Loop: Each thread should have an execution loop that:

A. Waits on its semaphore at the start of each cycle.

B. Performs its computational work.

C. Sleeps or waits for the next cycle based on its period.

ix. Task Synchronization for LCM Cycle: Create a master control thread or use the main function to synchronize task execution according to the LCM cycle. This involves:

A. Posting to task semaphores at the beginning of each task's period within the LCM cycle.

B. Monitoring overall cycle time to ensure adherence to the LCM invariant schedule.

(d) **Q: Describe whether you are able to achieve predictable reliable results when you implemented the equivalent code using Linux and pthreads to replicate the LCM invariant schedule. Provide a trace using syslog events and timestamps (Example syslog) and capture your trace to see if it matches VxWorks and the ideal expectation. Explain whether it does and any difference you can note.**

Answer:

i. Task Execution and Priority:
The tasks are assigned priorities inversely proportional to their execution period, adhering to the Rate Monotonic Scheduling (RMS) principle. This is evident from the priorities shown (98 for the 10ms task and 97 for the 20ms task) and their execution order.
The 10ms task (Thread10) runs more frequently than the 20ms task (Thread20), consistent with its shorter period and higher priority.

ii. Timestamps and Execution Periods:
The timestamps indicate when each task begins execution relative to the start of the test. These timestamps closely align with the expected periodic execution based on each task's period (every 10ms for Thread10 and every 20ms for Thread20), though there's a slight deviation in the arrival times due to the overhead and the precision of timing mechanisms in a general-purpose operating system like Linux.

iii. CPU Burst Time:
The burst times recorded are very close to the specified burst times for each task (10ms and 20ms). This suggests that the computational load for each task is being accurately simulated according to the task specifications.

iv. Here are screenshot for the code output

```
gcc -Werror -Wall -g  -o program lcm_invariant_schedule.o
This system has 12 processors configured and 12 processors available.
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO
Thread10 | priority = 98 | time stamp(arrival) 0.140869 msec | CPU burst time : 10.004150
Thread20 | priority = 97 | time stamp(arrival) 20.401855 msec | CPU burst time : 20.003906
Thread10 | priority = 98 | time stamp(arrival) 20.241943 msec | CPU burst time : 10.002930
Thread10 | priority = 98 | time stamp(arrival) 40.364990 msec | CPU burst time : 10.003906
Thread20 | priority = 97 | time stamp(arrival) 70.370850 msec | CPU burst time : 20.003906
Thread10 | priority = 98 | time stamp(arrival) 60.375000 msec | CPU burst time : 10.003906
Thread10 | priority = 98 | time stamp(arrival) 80.417969 msec | CPU burst time : 10.004883
Test Conducted over 100.422852 msec
```

Figure 8: RT thread improved with threads running on all cores

Which shows that the system is feasible and Thread with $C_1 = 10ms$ is getting higher priority than thread with $C_2 = 20ms$, that is why thread running at 20ms is preempting thread running at 50ms