RTES ECEN-5623
Exercise-1 Report

-by Jithendra H S and Suhas Reddy S

Question-1: [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) 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].

b) 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).

c) What is the total CPU utilization by the three services?

Answer:

a)

RTES Question 1															
	T1	3	C1	1	U1	0.333	LCM =	15							
	T2	5	C2	2	U2	0.4									
	T3	15	C3	3	U3	0.2	Utot =	0.933							
RM Schedule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S1	1			1			1			1			1		x
S2		1	2			1		2			1	2			x
S3					1				2					3	x

Indicates Service 1 i.e. S1 Indicates Service 2 i.e. S2 Indicates Service 3 i.e. S3 Indicates Idle State

The number within the color indicates the number of milliseconds spent in the service's current cycle.

T -> Time period and Deadline

U -> Utility Factor

Description:

First, we have to find the LCM of the time periods of all the services doing so we get LCM as 15 which is the same as time period T_3. In a total of 15 msec S1 was executed 3 times, S2 was executed 5 times, S3 was executed 1 time and 1 msec was idle. S1 gets the highest priority as it has the least shortest deadline and S3 has the least priority as it has the longest priority. This follows the Optimum Rate Monotonic Policy.

Assuming that requests for all the services arrive at the same time, S1 starts and completes in 1 msec first based-on priority then S2 and S3.

b) Liu and Layand have proved that total CPU utilization should be bound by a factor to schedule tasks that never miss deadlines making it feasible. The Least Upper Bound factor is given by $m(2^{1/m} - 1)$ where m is the number of services.

Therefore, the feasibility of processor utilization is given by, $U_{tot} <= m(2^{1/m} -1)$

Now substituting the values to LUB factor we get, $3(2^{1/3} -1) = 0.7797 =$ 77.97 %

Calculating total Utility factor (U_{tot})
U = C/T ; C is the Run-Time of each service and T is the Time period/Deadline.
The subscripts 1, 2, and 3 correspond to the services S1, S2 and S3.
$U_1 = C_1/T_1 = 1/3 = 0.333$
$U_2 = C_2/T_2 = 2/5 = 0.4$
$U_3 = C_3/T_3 = 3/15 = 0.2$
$U_{tot} = U_1+U_2+U_3 = 0.933 =$ 93.3% —-->1

U_{tot} is 93.3% and is greater than 77.97%, due to this the provided services may not always meet all the deadlines. Which might be due to longer run-time of one or more tasks, context switching between tasks, or some other delays caused by the system. Though there is no need to adhere strictly to this bound when the service run-times includes all the extreme delays, it is wise to satisfy feasibility conditions when we are unsure of the run-time, various delays involved.

c) As calculated in b) total Utility factor of all the services is given by U_{tot} = 93.3%. The method used to calculate is simple. We need to sum the ratio of run-time (C) to Time period (T) of each service.

Question-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.
a) What was the root cause of the overload and why did it violate Rate Monotonic policy?
b) 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.
c) Plot this Least Upper bound as a function of the number of services.
d) Describe 3 key assumptions they make and document 3 or more aspects of their fixed priority LUB derivation that you don't understand.
e) Would RM analysis have prevented the Apollo 11 1201/1202 errors and potential mission abort? Why or why not?

Answer:
Summary
During the Apollo 11 mission, the on-board computer memory comprised seven core-sets and five Vector Accumulators (VAC). Each task necessitated a core-set, and if needed, one of the five VACs could be utilized for additional memory. Consequently, the on-board computer had the capability to concurrently execute a maximum of seven tasks.

However, the lunar module's guidance computer encountered an overload issue when it unexpectedly received an influx of data from the rendezvous radar system. Despite being unnecessary for the landing phase, the radar system began transmitting excessive and nonexistent rendezvous radar data to the computer while Neil Armstrong and Buzz Aldrin were attempting to land on the moon. This surge of data overwhelmed the computer's processing capabilities, resulting in an overflow in the core-sets that triggered the 1202 alarm and an overflow in the VAC that triggered the 1201 alarm.

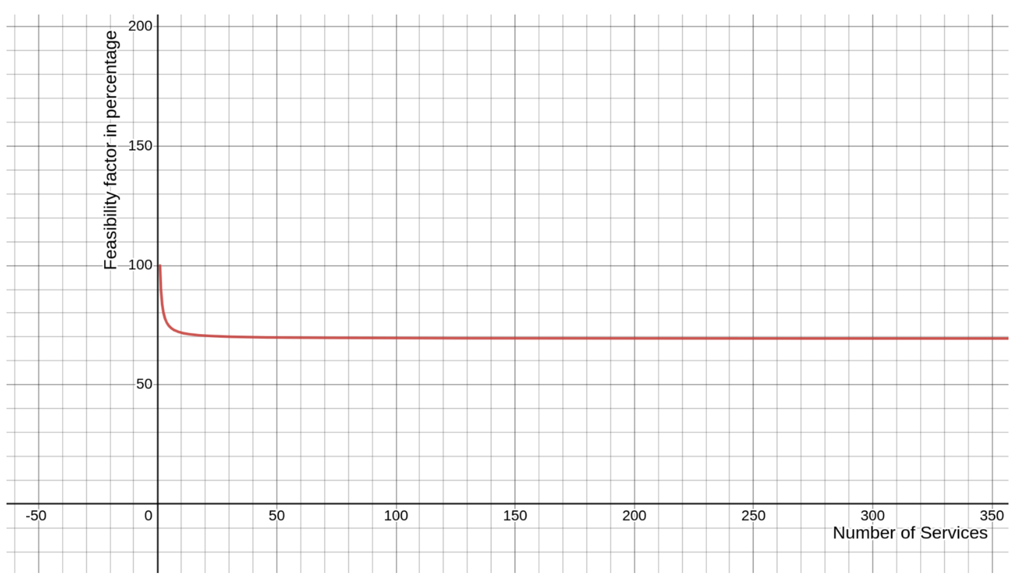
Despite these challenges, Margaret Hamilton's well-designed software and error detection mechanisms enabled the computer to recognize the problem and take recovery action. The software effectively prioritized critical landing tasks over low priority radar tasks which were non-essential ones, ensuring the successful completion of the Apollo 11 lunar landing mission.

a)The root cause of the overload was the enormous amount of erroneous data transmitted by the rendezvous radar. Although it was a low-priority task, it occupied all the memory spaces—seven core-sets and five VAC spaces—and still requested more space, ultimately triggering alarms 1202 and 1201. The issue may have arisen due to faulty hardware switches. Margaret's fault-handling code detected this and initiated a reboot, ignoring the low-priority task from the radar and scheduling only the high-priority tasks necessary for landing.

The overload violated the Rate Monotonic policy, which prioritizes tasks based on their deadlines and ensures that higher-priority tasks are completed within their specified timeframes. In this case, the unexpected influx of data from the radar system disrupted the normal processing flow of the computer, causing it to prioritize non-essential tasks over critical landing computations. This violation of the Rate Monotonic policy led to the generation of the 1201 and 1202 program alarms, indicating executive overflows and the computer's inability to complete all tasks in real-time.

c) Plot for Least Upper bound as a function of the number of services. $y = m(2^{1/m} - 1)$
y -> Feasibility factor for m -> Number of services (m >=1)

The value of feasibility factor is shown as percentage and the value plateaus at ~ 70% as it should do according to the equation.



d) Assumptions made by Liu and Layland:
A1) 'The requests for all tasks, for which hard deadlines exist, are periodic, with constant intervals between requests.' This assumption is crucial, as it considers hard-real time tasks to be periodic, implying that tasks must be requested at regular intervals, and these time periods align with the deadlines of the services. In turn, this enables the use of periodic mathematical models like Rate Monotonic analysis for task scheduling, which can be easily implemented on a system.

A3) 'The tasks are independent, implying that requests for a certain task do not depend on the initiation or completion of requests for other tasks.' However, in real-world scenarios where tasks are interdependent, it may be necessary to extend the deadline of the dependent task to a time after the task on which it relies is completed. This adjustment accommodates the inherent dependencies among tasks.

A4) 'Run-time for each task is constant for that task and does not vary with time. Run-time here refers to the time taken by a processor to execute the task without interruption.' This constant run-time assumption is fundamental to fixed-priority scheduling for periodic hard-real time tasks, as it facilitates the analysis and determination of how tasks must be scheduled. It aids in calculating the CPU utilization during task execution and contributes to effective scheduling strategies.

Explanation: These assumptions collectively provide a structured foundation for analyzing and scheduling hard-real time tasks, emphasizing periodicity, independence, and constant run-time. While acknowledging real-world complexities, these assumptions enable the application of mathematical models and scheduling strategies to meet stringent deadlines in embedded systems.

Three aspects not understood from Fixed Priority Least Upper Bound derivation given by liu and layland:

- Use of delta in theorem 4 is unclear and not explained.
- Unclear how we arrived at $C_m = T_m - 2(C_1 + C_2 \ldots\ldots + C_{m-1})$
- The proof for theorem 5 was unclear.

e) Rate Monotonic Analysis (RMA) would not have directly prevented the Apollo 11 1201/1202 errors. These errors were caused by an overflow in the Apollo Guidance Computer, resulting from unexpected input and computational load. RMA, designed for scheduling tasks in periodic systems, couldn't account for the complex and non-periodic nature of the Apollo Guidance Computer's tasks and interrupts. The errors required a combination of rigorous testing, fault tolerance, and tailored software design to address the specific challenges of the lunar landing mission. Margaret Hamilton thought about and implemented a fault-safe software that would safeguard the on-board computer and protect the mission.

<u>Question-3:</u> [20 points] Download the RT-Clock code from http://mercury.pr.erau.edu/~siewerts/cec450/code/RT-Clock/ or from Canvas (Modules ->Exercises->Exercise 1->Code) and build it on a Jetson board, Raspberry Pi, or Altera DE1- SOC board and execute the code.
a) 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).
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?

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

Answer:

a)
The provided code demonstrates the utilization of the nanosleep function and real-time clock (CLOCK_REALTIME) within a multithreaded context. Initially, the code checks the current scheduling policy of the main thread or parent process. Subsequently, it creates a new set of scheduling policy attributes such as priority, which can be applied to pthreads, by enforcing the use of "PTHREAD_EXPLICIT_SCHED". This configuration ensures that the parent process adopts a FIFO scheduling policy with the highest priority, guaranteeing immediate CPU access when ready to execute. Then, it sets these scheduling parameters to the pthread attribute object, ensuring that any pthread created using this object inherits the same scheduling parameters, thus maintaining consistent scheduling behavior across all threads within the application. A thread is then created for the "delay_test" routine, and the program waits until the thread completes its execution. During the thread's execution, the code attempts to sleep for 3 seconds using the nanosleep function while tracking real-time to evaluate the accuracy of creating delays using nanosleep, as demonstrated in the provided example code.

The clock_gettime API retrieves time information from the specified clock source, which can be monotonic or real-time. By leveraging this function, one can determine the execution time of code segments by calculating the delta between the time recorded at the start and end of a function. In this context, the code measures the delta between the expected delay and the actual delay achieved through the utilization of nanosleep.

b)
Low Interrupt handler latency:

- Interrupt handler latency refers to the time it takes for the system to respond to an external event or interrupt.
- Ensuring low interrupt handler latency is crucial for promptly handling critical events in real-time systems.
- In situations where timely responses are essential, minimizing interrupt handler latency helps meet stringent timing requirements.
- In a real-time embedded system designed for home security, swift response to events is critical. Let's say a motion sensor installed near the back door detects movement. Instantly, it sends an interrupt signal to the microcontroller unit (MCU) overseeing the security system. Without delay, the MCU's interrupt handler swiftly processes the input, analyzing the sensor data to identify the source and location of the motion. Recognizing the potential security breach, the MCU triggers an immediate response, activating security cameras, sounding alarms, and sending alerts to the homeowner's smartphone. This seamless sequence of events, orchestrated with low interrupt handler latency, ensures that the security system reacts promptly to threats, enhancing the safety and protection of the home environment.

Low Context switch time:

- Context switch time represents the duration required to save the state of one task, restore the state of another, and transfer control between them.

- Efficient context switching enables the system to allocate CPU time to different tasks while minimizing overhead.
- In real-time systems, reducing context switch time is imperative to ensure that tasks are scheduled and executed within their allotted time slices.
- In a multi-user server environment, low context switch time is paramount for maintaining responsive performance and efficient resource utilization. As the server juggles multiple client connections simultaneously, quick transitions between processing tasks are essential to minimize overhead and maximize CPU availability. With low context switch time, the server can swiftly switch between executing different tasks, ensuring that client requests are handled promptly without significant delays. This efficiency enhances the overall throughput and scalability of the server, enabling it to accommodate increasing numbers of users or requests while delivering a seamless and responsive user experience.

Stable timer services:

- Stable timer services imply that the system can maintain precise timing behavior over extended periods.
- Timer services are crucial for accurate timer interrupts, task scheduling, and managing timeouts.
- In safety-critical systems where timing accuracy is paramount, stable timer services help prevent timing anomalies that could lead to system failures or unpredictable behavior.
- In the realm of flight control systems, stable timer services are indispensable for ensuring the precise and reliable operation of autopilot functions. These timer services play a vital role in monitoring flight dynamics, adjusting control surfaces, and executing scheduled tasks according to the flight plan. They enable timely responses to dynamic flight conditions and pilot commands, facilitating smooth transitions between control modes and accurate aircraft control inputs. With stable timer services, flight control systems can integrate seamlessly with avionics systems, execute emergency procedures, and maintain safe and efficient flight operations from takeoff to landing. The reliability and accuracy of timer services are paramount for enhancing aviation safety and passenger comfort in commercial and general aviation environments.

c)

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

RT clock start seconds = 1707347999, nanoseconds = 379980600
RT clock stop seconds = 1707348002, nanoseconds = 380330800
RT clock DT seconds = 3, nanoseconds = 350200
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 350200
```

The reported delay error of 350200 nanoseconds may indicate that the actual sleep time deviated significantly from the requested sleep time. This deviation could be attributed to factors such as scheduling overhead, context switches, and system interrupts.

The irregularity observed might be caused by the reasons mentioned in https://linux.die.net/man/2/nanosleep: If the specified delay interval is not an exact multiple of the underlying clock's granularity, the interval will be rounded up to the next multiple. Furthermore, after the sleep completes, there may still be a delay before the CPU becomes free to execute the calling thread once again.

The fact that nanosleep() operates with a relative interval can be problematic if the call is repeatedly restarted after being interrupted by signals. The time between interruptions and restarts of the call may lead to drift in the time when the sleep finally completes.

Question-4: [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) Download, build and run code in the following three example programs: 1) simplethread, 2) rt_simplethread, and 3) 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].

b) 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.

c) 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: The observed timing above fits our theory for RM policy on a priority preemptive scheduling system as shown by the timing diagram below: Your description should outline how you would implement code equivalent to the VxWorks synthetic load generation and schedule emulator.

d) 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 a Virtual Machine, or on a Jetson, Altera or Raspberry Pi system (they are preferred and should result in more reliable results). Based upon POSIX Pthread code and examples of the use of interval timers and semaphores, please attempt to implement two services using POSIX threading that compute a sequence (synthetic load) and match the original VxWorks diagram with: S1=f10, C1=10 msec, T1=20 msec, D1=T1 and S2=f20, C1=20 msec, T2=50 msec, D2=T2 as is diagrammed above in the timing diagram and shown with the VxWorks trace. You may want to review example code for help (sequencer, sequencer_generic) and look at a more complete example in this contributed Linux code from one of our student assistants in Report.pdf. Recall that U=90%, and the two services f10 and f20 simply burn CPU cycles computing a sequence and run over the LCM of their periods – 100 msec. The trace above was based on this original VxWorks code and your code should match this schedule and timing as best you can.

e) Describe whether you are able to achieve predictable reliable results when you implemented the equivalent code using Linux and pthreads to replicate the LCM nvariant 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.

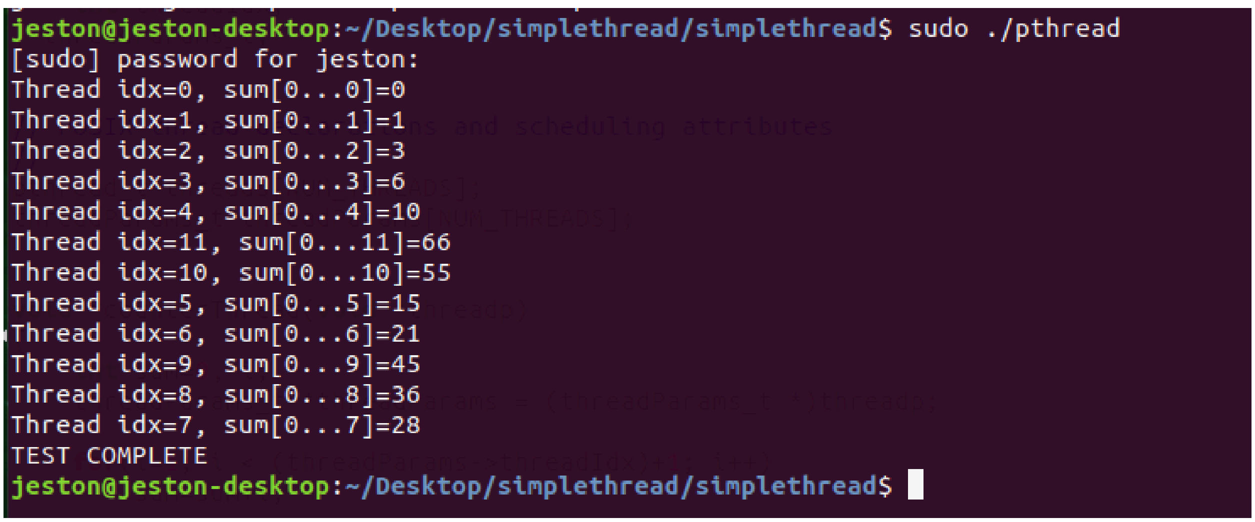
Answer:

a)

SimpleThread:

- The SimpleThread program generates 12 counter threads.
- The counterThread() function is responsible for each thread's execution.
- Within the function, it iterates from 1 to the thread index (threadParams->threadIdx+1). During the iteration, it calculates the sum of integers from 1 to the thread index and stores the result in the sum variable.
- The final result is printed. The output indicates that all 12 threads are executed, but the order is non-deterministic.
- It is noted that thread index 0 is consistently executed first, while the scheduling of the other threads occurs randomly.

Image:



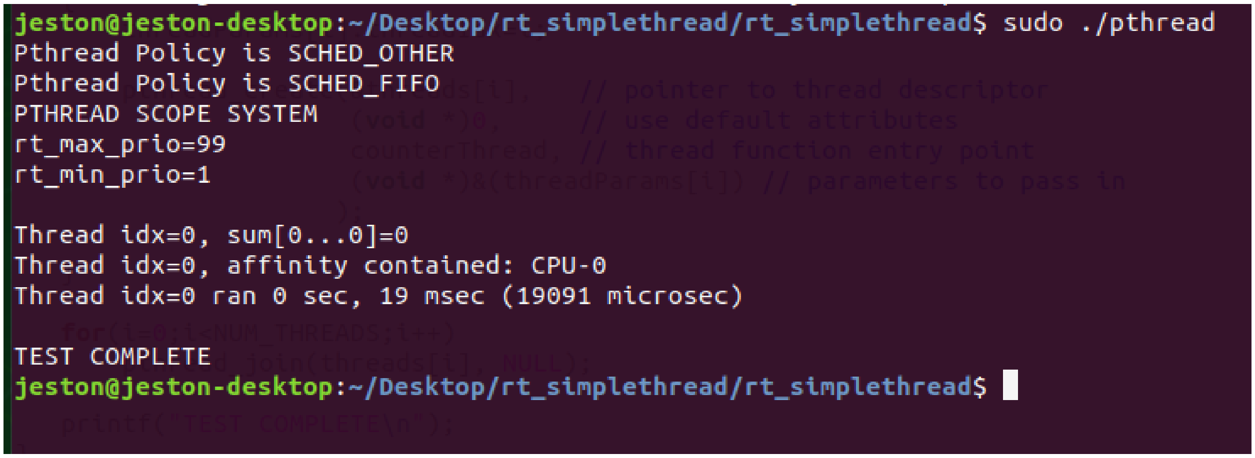
```
jeston@jeston-desktop:~/Desktop/simplethread/simplethread$ sudo ./pthread
[sudo] password for jeston:
Thread idx=0, sum[0...0]=0
Thread idx=1, sum[0...1]=1
Thread idx=2, sum[0...2]=3
Thread idx=3, sum[0...3]=6
Thread idx=4, sum[0...4]=10
Thread idx=11, sum[0...11]=66
Thread idx=10, sum[0...10]=55
Thread idx=5, sum[0...5]=15
Thread idx=6, sum[0...6]=21
Thread idx=9, sum[0...9]=45
Thread idx=8, sum[0...8]=36
Thread idx=7, sum[0...7]=28
TEST COMPLETE
jeston@jeston-desktop:~/Desktop/simplethread/simplethread$
```

RT SimpleThread:

- The Rt_simplethread program is an enhanced version of simplethread that showcases real-time multi-threading using threads.
- It incorporates various features, including a dedicated thread function for computations, Fibonacci sequence calculations, time profiling, scheduler information printing, and CPU affinity settings.
- In the original program, the thread creation line used default attributes:
- pthread_create(&threads[i], **(void *)0**, counterThread, (void *)&(threadParams[i]));
- The modification replaced the default attributes with defined attributes:
- pthread_create(&threads[i], **&rt_sched_attr[i]**, counterThread, (void *)&(threadParams[i]));
- The change involves using the thread attributes (&rt_sched_attr[i]) explicitly instead of default attributes.
- The new approach includes specifying attributes such as affinity and priority for the created thread.
- Observation: Despite no apparent difference when using a single thread with either default or defined attributes, executing multiple threads showed a significant decrease in execution time when using defined attributes.

- The main function creates a thread, each with its own attributes, affinity, and priority.
- The program serves as an illustration of real-time programming concepts in a multi-threaded context.

Image:



RT Thread Improved:

- The RT_Thread_Improved program is an improvement over rt_simple_thread.
- The thread routine is similar to rt_simple_thread but now includes printing information about the core on which the thread is running.
- The main function is built upon the rt_simple_thread main function, aiming to run 4 threads concurrently on 4 cores, with each thread assigned to its respective core.
- The thread routine remains the same as rt_simple_thread, with the addition of printing information about the core on which the thread is currently running.
- The main function is adapted from rt_simple_thread to facilitate running 4 threads on 4 cores.
- Each thread is assigned its own core to execute on.
- Initially, the pthread creation step used default attributes:
- pthread_create(&threads[i], (void *)0, counterThread, (void *)&(threadParams[i]));
- The code is modified to make use of desired attributes during pthread creation:
- pthread_create(&threads[i], &desired_attributes[i], counterThread, (void *)&(threadParams[i]));
- The system has 4 processors configured and 4 processors available.
- The program aims to optimize thread execution by assigning each thread to a specific core, leveraging desired attributes for pthread creation.

Image:

```
jeston@jeston-desktop:~/Desktop/rt_thread_improved/rt_thread_improved$ sudo ./pthread
This system has 4 processors configured and 2 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=0, affinity contained: CPU-0

Thread idx=0 ran 0 sec, 315 msec (315064 microsec)

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

Thread idx=1 ran 0 sec, 315 msec (315242 microsec)

TEST COMPLETE
jeston@jeston-desktop:~/Desktop/rt_thread_improved/rt_thread_improved$
```

b)

Function call description:

The program uses two syscalls pthread_create() and pthread_join():

pthread_create():

- Functionality: Creates a new thread and starts its execution.
- Syntax: int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
- thread: Pointer to a pthread_t structure that will store the thread ID.
- attr: Pointer to pthread_attr_t structure specifying thread attributes (or NULL for default attributes).
- start_routine: Function pointer to the function that the new thread will execute.
- arg: Pointer to the argument that will be passed to the start_routine.
- Returns 0 on success; otherwise, an error number is returned.

pthread_join():

- Functionality: Waits for a specified thread to terminate and collects its exit status.
- Syntax: int pthread_join(pthread_t thread, void **retval);
- thread: The thread ID of the thread to be waited upon.
- retval: Pointer to a location where the exit status of the joined thread will be stored.

- Returns 0 on success; otherwise, an error number is returned.

In the initial test run it was observed that during transitions from increment to decrement or vice versa the values weren't correct. And also the threads executed now deterministically.

Using semaphores to tackle race conditions and achieve determinism.

- Two semaphores, sem_inc and sem_dec, are used for controlling access to shared resources.

```
// Initialize semaphores
sem_init(&sem_inc, 0, 1); // Initialize the increment semaphore to 1
sem_init(&sem_dec, 0, 0); // Initialize the decrement semaphore to 0
```

- sem_inc is initialized with one resource (sem_init(&sem_inc, 0, 1)).
- sem_dec is initialized with zero resources (sem_init(&sem_dec, 0, 0)).
- sem_inc starts with one resource, allowing the increment thread to proceed initially.
- sem_dec starts with zero resources, ensuring the decrement thread waits until a resource is available.

```
for(i=0; i<COUNT; i++)
{
    sem_wait(&sem_inc); // Wait for the increment semaphore
    gsum = gsum + i;
    printf("Increment thread idx=%d, gsum=%d\n", threadParams->threadIdx, gsum);
    sem_post(&sem_dec); // Release the decrement semaphore
}
```

```
for(i=0; i<COUNT; i++)
{
    sem_wait(&sem_dec); // Wait for the decrement semaphore
    gsum = gsum - i;
    printf("Decrement thread idx=%d, gsum=%d\n", threadParams->threadIdx, gsum);
    sem_post(&sem_inc); // Release the increment semaphore
}
```

- When the increment thread enters its execution section, it uses sem_wait(&sem_inc) to decrement the resource count for sem_inc to 0.
- After completing its operation, it uses sem_post(&sem_dec) to increase the resource count for sem_dec, allowing the decrement thread to proceed.
- A shared global variable, gsum, is locked using a mechanism such as a mutex to avoid race conditions.
- Locking ensures that only one thread can operate on gsum at a time, preventing conflicts and data corruption.
- sem_inc: Manages the access to the increment thread, controlling when it can execute.
- sem_dec: Controls the access to the decrement thread, ensuring synchronization and preventing conflicts with the increment thread.
- The program consistently starts with the increment thread due to the initial resource configuration of the semaphores. This can be reversed by swapping the resource value of increment and decrement of the semaphores in the init step.

Output: The output demonstrates the synchronized execution of an increment thread (idx=0) and a decrement thread (idx=1) using semaphores to handle race conditions. The shared variable gsum consistently increases with each increment thread operation, and the decrement thread effectively decrements it without reaching zero. The output pattern continues until both the increment and decrement functions reach 1000 iterations.

Increment thread idx=0, gsum=0
Decrement thread idx=1, gsum=0
Increment thread idx=0, gsum=1
Decrement thread idx=1, gsum=0
Increment thread idx=0, gsum=2
Decrement thread idx=1, gsum=0
.
.
.
.
.
.
Increment thread idx=0, gsum=997
Decrement thread idx=1, gsum=0
Increment thread idx=0, gsum=998
Decrement thread idx=1, gsum=0
Increment thread idx=0, gsum=999
Decrement thread idx=1, gsum=0
TEST COMPLETE

Using SCHED_FIFO to tackle race conditions and achieve determinism.

- The main function initializes parameters and sets the scheduling policy of the main process to SCHED_FIFO.
- CPU affinity is set for the specified number of CPUs which is one in this case.
- The functions, incThread and decThread remain the threads that need to be executed deterministically without race conditions.
- The scheduling parameters of each thread, including priority and CPU affinity, are explicitly defined and set using the pthread attributes.
- Main thread is given the highest priority, followed by increment thread and decrement thread.
- Once the main program starts the increment thread is fully executed first and then the decrement starts and concludes.
- Because the threads don’t run concurrently there won’t be any race conditions while accessing the gsum variable.

Output:

Pthread Policy is SCHED_OTHER
rt_max_prio=99
rt_min_prio=1

Pthread Policy is SCHED_FIFO
Increment thread idx=0, priority = 98, gsum=0
Increment thread idx=0, priority = 98, gsum=1
Increment thread idx=0, priority = 98, gsum=3
Increment thread idx=0, priority = 98, gsum=6
Increment thread idx=0, priority = 98, gsum=10
Increment thread idx=0, priority = 98, gsum=15
.
.
Increment thread idx=0, priority = 98, gsum=497503
Increment thread idx=0, priority = 98, gsum=498501
Increment thread idx=0, priority = 98, gsum=499500
Decrement thread idx=1, priority = 97, gsum=45
Decrement thread idx=1, priority = 97, gsum=499499
Decrement thread idx=1, priority = 97, gsum=499497
Decrement thread idx=1, priority = 97, gsum=499494
.
.
.
Decrement thread idx=1, priority = 97, gsum=6972
Decrement thread idx=1, priority = 97, gsum=5979
Decrement thread idx=1, priority = 97, gsum=4985
Decrement thread idx=1, priority = 97, gsum=3990
Decrement thread idx=1, priority = 97, gsum=2994
Decrement thread idx=1, priority = 97, gsum=1997
Decrement thread idx=1, priority = 97, gsum=999
Decrement thread idx=1, priority = 97, gsum=0
TEST COMPLETE

This way determinism and race condition avoidance is achieved via two different scheduling approaches.

c)
Replicating code from a VxWorks RTOS to Linux can present several challenges:

1. Scheduling Differences:
RTOSs often provide deterministic scheduling mechanisms with fixed priorities and precise timing guarantees. Linux, on the other hand, offers various scheduling policies like SCHED_FIFO, SCHED_RR, and SCHED_OTHER, which may not provide the same level of determinism or real-time performance.

2. Thread Management:
RTOSs typically have lightweight threading models optimized for real-time applications. Porting thread management code to Linux may require adjustments due to differences in APIs, thread priorities, and synchronization primitives.

3. Timer and Clock Resolution:

RTOSs often offer high-resolution timers and clocks suitable for real-time applications. Linux may not provide the same level of precision, and developers need to carefully select and configure timer mechanisms to meet real-time requirements.

As we aim to replicate our code from VxWorks RTOS to a Linux platform using the POSIX API and semaphores while ensuring suitable time management with clock_gettime, we need to follow these steps:

1. Thread Creation and Scheduling:
   - We'll utilize POSIX threads (`pthread`) to create threads for our tasks like `fib10` and `fib20`.
   - We'll set the scheduling policy to `SCHED_FIFO` for real-time behavior.
   - Configuring thread priorities using `pthread_attr_setschedparam` will be crucial.
2. Synchronization:
   - We'll implement synchronization between threads using POSIX semaphores (`sem_init`, `sem_wait`, `sem_post`, `sem_destroy`).
   - Using semaphores, we'll control the execution flow and timing of our tasks to ensure they run according to the desired schedule.
3. Timing and Clock Resolution:
   - We plan to utilize `clock_gettime` with the `CLOCK_REALTIME` clock ID to measure time intervals accurately.
   - Calculating the time difference between events will ensure precise timing for our tasks.
   - We'll adjust timing parameters as necessary to match the desired schedule, such as the period and deadline of our tasks.
4. Resource Management:
   - We'll ensure proper resource management, including memory allocation and deallocation, to prevent resource leaks and ensure efficient operation.
   - Managing CPU affinity and core utilization will be important to optimize performance and minimize contention among threads.

By adhering to these steps and leveraging the POSIX API, semaphores, and accurate time management provided by `clock_gettime`, we believe we can effectively replicate the behavior of our code from VxWorks RTOS to a Linux platform. Thorough testing and validation will be necessary to verify that our implementation meets the desired real-time requirements and behaves as expected.

d)

Code implementation can be found in the Appendix.

e)

In our attempt to replicate the Least Common Multiple (LCM) invariant schedule using Linux and pthreads, we encountered challenges in achieving predictable and reliable results. Despite our efforts to confine threads to a single core to mitigate some irregularities, we still observed deviations in thread execution due to varying thread loads.

To address these issues, we opted to measure the time required for the execution of threads. By dynamically adjusting the number of iterations based on these measurements, we aimed to approximate the desired behavior and attain a more consistent execution pattern. This adaptive strategy allowed us to compensate for the unpredictability inherent in the Linux and pthreads environment, ultimately enhancing the reliability of our implementation.

So the results of our implementation were close to the expected timing, but they did not perfectly match the expected timings. Despite our efforts to optimize and fine-tune the implementation, variations in thread execution and system load contributed to some level of deviation from the expected timing. However, by closely monitoring the execution time of threads and adjusting the number of iterations dynamically, we were able to approximate the desired behavior and achieve results that were close to the expected timing, although not entirely perfect.

Output:

```
gcc -O3   -c pthread.c
gcc -g  -O3   -o pthread pthread.o -lpthread
TEST STARTED
Pthread Policy is SCHED_OTHER
rt_max_prio=99
rt_min_prio=1
Pthread Policy is SCHED_FIFO
-----------------------------------------------------------
               Trial 1
-----------------------------------------------------------
THREAD      PRIORITY       STATUS       TIMESTAMP
-----------------------------------------------------------
FIB_10        97          started       0 sec, 0 msec
FIB_10        97         Completed      0 sec, 9 msec
FIB_20        96          started       0 sec, 9 msec
FIB_10        97          started       0 sec, 20 msec
FIB_10        97         Completed      0 sec, 28 msec
FIB_20        96         Completed      0 sec, 28 msec
FIB_10        97          started       0 sec, 40 msec
FIB_10        97         Completed      0 sec, 55 msec
FIB_20        96          started       0 sec, 55 msec
FIB_10        97          started       0 sec, 60 msec
FIB_10        97         Completed      0 sec, 70 msec
FIB_20        96         Completed      0 sec, 74 msec
FIB_10        97          started       0 sec, 81 msec
FIB_10        97         Completed      0 sec, 89 msec
-----------------------------------------------------------
fib10 ran count 5, fib20 ran count 2
-----------------------------------------------------------
TEST COMPLETE
```

Syslog output:

```
Feb10 15:07:59DESKTOP-4OVQQO0 Fibonacci[1216]: *************schedule started*************
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 9
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 28
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_20 completed at 28
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 55
```

Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 70
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_20 completed at 74
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 89

Appendix

4b. Semaphore

```
/****************************************************************************
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable for
 * any misuse of this material.
 * ****************************************************************************/
/**
 * @file pthread.c
 * @brief Multithreaded increment and decrement operations using semaphores.
 *
 * This program demonstrates multithreaded increment and decrement operations
 * using POSIX threads and semaphores for synchronization. Two threads are created:
 * incThread and decThread, each responsible for performing either increment or
 * decrement operations on a global variable gsum.
 *
 * @author Jithendra and Suhas
 * @date February 10, 2024
 */

#define _GNU_SOURCE
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <semaphore.h>

#define COUNT  (1000)
#define NUM_THREADS (2)

typedef struct
{
   int threadIdx;
} threadParams_t;

pthread_t threads[NUM_THREADS];
threadParams_t threadParams[NUM_THREADS];

// Semaphores
sem_t sem_inc, sem_dec;
```

```
// Unsafe global
int gsum=0;

/**
 * @brief Increment thread function.
 *
 * This function performs the increment operation on the global variable gsum
 * using semaphores for synchronization.
 *
 * @param threadp Pointer to thread parameters.
 * @return None.
 */
void *incThread(void *threadp)
{
    int i;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    for(i=0; i<COUNT; i++)
    {
        sem_wait(&sem_inc); // Wait for the increment semaphore
        gsum = gsum + i;
        printf("Increment thread idx=%d, gsum=%d\n", threadParams->threadIdx, gsum);
        sem_post(&sem_dec); // Release the decrement semaphore
    }
}

/**
 * @brief Decrement thread function.
 *
 * This function performs the decrement operation on the global variable gsum
 * using semaphores for synchronization.
 *
 * @param threadp Pointer to thread parameters.
 * @return None.
 */
void *decThread(void *threadp)
{
    int i;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    for(i=0; i<COUNT; i++)
    {
        sem_wait(&sem_dec); // Wait for the decrement semaphore
        gsum = gsum - i;
        printf("Decrement thread idx=%d, gsum=%d\n", threadParams->threadIdx, gsum);
```

```
        sem_post(&sem_inc); // Release the increment semaphore
    }
}

/**
 * @brief Main function to control the execution of the program.
 *
 * This function initializes semaphores, creates threads, and waits for them to complete.
 *
 * @return 0 on successful execution.
 */
int main (int argc, char *argv[])
{
   int rc;
   int i=0;

   // Initialize semaphores
   sem_init(&sem_inc, 0, 1); // Initialize the increment semaphore to 1
   sem_init(&sem_dec, 0, 0); // Initialize the decrement semaphore to 0

   // Create increment and decrement thread
    for(i=0; i<NUM_THREADS; i++){
       threadParams[i].threadIdx = i;
       pthread_create(&threads[i], (void*)0, (i%2 ? decThread : incThread), (void *)&(threadParams[i]));
    }

   // Wait for threads to complete
   for(i=0; i<NUM_THREADS; i++)
       pthread_join(threads[i], NULL);

   // Destroy semaphores
   sem_destroy(&sem_inc);
   sem_destroy(&sem_dec);

   printf("TEST COMPLETE\n");

   return 0;
}
```

****

Output: Complete output can be found in the output.txt file present in semaphore.zip

Increment thread idx=0, gsum=0
Decrement thread idx=1, gsum=0
Increment thread idx=0, gsum=1
Decrement thread idx=1, gsum=0

```
Increment thread idx=0, gsum=2
Decrement thread idx=1, gsum=0
.
.
.
.
.
.
Increment thread idx=0, gsum=997
Decrement thread idx=1, gsum=0
Increment thread idx=0, gsum=998
Decrement thread idx=1, gsum=0
Increment thread idx=0, gsum=999
Decrement thread idx=1, gsum=0
TEST COMPLETE
********************************************************************************
```

4b. SCHED_FIFO

```
/********************************************************************************
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable for
 * any misuse of this material.
 * *****************************************************************************/
/**
 * @file pthread.c
 * @brief Multithreaded increment and decrement operations with real-time scheduling.
 *
 * This program demonstrates multithreaded increment and decrement operations
 * using POSIX threads with real-time scheduling policies. It defines two threads:
 * incThread and decThread, each responsible for performing either increment or
 * decrement operations on a global variable gsum.
 *
 * @author Jithendra and Suhas
 * @date February 10, 2024
 */

#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sched.h>
```

```
#define COUNT 1000
#define NUM_THREADS (2)
#define NUM_CPUS (1)

// Unsafe global
int gsum=0;

typedef struct {
  int threadIdx;
} threadParams_t;

pthread_t threads[2];
threadParams_t threadParams[2];
int rt_max_prio, rt_min_prio;

// Schedule param
struct sched_param main_param;

/**
 * @brief Increment thread function.
 *
 * This function performs the increment operation on the global variable gsum.
 *
 * @param threadp Pointer to thread parameters.
 * @return None.
 */
void *incThread(void *threadp) {
 int i;
 int policy = SCHED_FIFO;
 struct sched_param param;
 threadParams_t *threadParams = (threadParams_t *)threadp;

 /* scheduling parameters of target thread */
 int ret = pthread_getschedparam (pthread_self(), &policy, &param);

 for (i = 0; i < COUNT; i++) {
  gsum = gsum + i;
      printf("Increment thread idx=%d, priority = %d, gsum=%d\n", threadParams->threadIdx,
param.sched_priority,  gsum);
 }
}

/**
 * @brief Decrement thread function.
 *
```

```
 * This function performs the decrement operation on the global variable gsum.
 *
 * @param threadp Pointer to thread parameters.
 * @return None.
 */
void *decThread(void *threadp) {
  int i;
  int policy = SCHED_FIFO;
  struct sched_param param;
  threadParams_t *threadParams = (threadParams_t *)threadp;

  /* scheduling parameters of target thread */
  int ret = pthread_getschedparam (pthread_self(), &policy, &param);

  for (i = 0; i < COUNT; i++) {
    gsum = gsum - i;
        printf("Decrement thread idx=%d, priority = %d, gsum=%d\n", threadParams->threadIdx,
param.sched_priority,  gsum);
  }
}

/**
 * @brief Prints the scheduling policy of the current process.
 *
 * This function retrieves the scheduling policy of the current process
 * and prints the corresponding string representation.
 *
 * @return None.
 */
void print_scheduler(void) {
  int schedType;

  schedType = sched_getscheduler(getpid());

  switch (schedType) {
  case SCHED_FIFO:
    printf("Pthread Policy is SCHED_FIFO\n");
    break;
  case SCHED_OTHER:
    printf("Pthread Policy is SCHED_OTHER\n");
    break;
  case SCHED_RR:
    printf("Pthread Policy is SCHED_RR\n");
    break;
  default:
    printf("Pthread Policy is UNKNOWN\n");
```

```
  }
}

/**
 * @brief Main function to control the execution of the program.
 *
 * This function initializes parameters, creates threads, sets scheduling policies,
 * and waits for threads to complete.
 *
 * @return 0 on successful execution.
 */
int main(int argc, char *argv[]) {
    int rc;
    int i = 0;
    struct sched_param rt_param;
    int policy = SCHED_FIFO;
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    for(i=0; i < NUM_CPUS; i++)
       CPU_SET(i, &cpuset);
    print_scheduler();

    // Get scheduling parameters of main process
    rc = sched_getparam(getpid(), &main_param);
    rt_max_prio = sched_get_priority_max(SCHED_FIFO);
    rt_min_prio = sched_get_priority_min(SCHED_FIFO);
    printf("rt_max_prio=%d\n", rt_max_prio);
    printf("rt_min_prio=%d\n", rt_min_prio);

    // Set main process priority to maximum
    main_param.sched_priority = rt_max_prio;
    rc = sched_setscheduler(getpid(), SCHED_FIFO, &main_param);
    if (rc < 0)
      perror("main_param");

    print_scheduler();

    // Create threads
    for (i = 0; i < NUM_THREADS; i++) {
        threadParams[i].threadIdx = i;

        // Create thread with FIFO scheduling policy
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        rc=pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
        pthread_attr_setschedpolicy(&attr, policy);
```

```
        rc=pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpuset);
        rt_param.sched_priority = rt_max_prio - i -  1;
        pthread_attr_setschedparam(&attr, &rt_param);

        // Decide which thread function to execute based on thread index
        rc = pthread_create(&threads[i], &attr, (i % 2 ? decThread : incThread), (void *)&

(threadParams[i]));
        if (rc) {
           printf("ERROR; return code from pthread_create() is %d\n", rc);
           exit(-1);
        }
    }

    // Wait for threads to complete
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(threads[i], NULL);

    printf("TEST COMPLETE\n");

    return 0;
}
```

****

Output: Complete output can be found in the output.txt file present in sched_fifo.zip

```
Pthread Policy is SCHED_OTHER
rt_max_prio=99
rt_min_prio=1
Pthread Policy is SCHED_FIFO
Increment thread idx=0, priority = 98, gsum=0
Increment thread idx=0, priority = 98, gsum=1
Increment thread idx=0, priority = 98, gsum=3
Increment thread idx=0, priority = 98, gsum=6
Increment thread idx=0, priority = 98, gsum=10
Increment thread idx=0, priority = 98, gsum=15
.
.
Increment thread idx=0, priority = 98, gsum=497503
Increment thread idx=0, priority = 98, gsum=498501
Increment thread idx=0, priority = 98, gsum=499500
Decrement thread idx=1, priority = 97, gsum=45
Decrement thread idx=1, priority = 97, gsum=499499
Decrement thread idx=1, priority = 97, gsum=499497
Decrement thread idx=1, priority = 97, gsum=499494
.
```

.
.
Decrement thread idx=1, priority = 97, gsum=6972
Decrement thread idx=1, priority = 97, gsum=5979
Decrement thread idx=1, priority = 97, gsum=4985
Decrement thread idx=1, priority = 97, gsum=3990
Decrement thread idx=1, priority = 97, gsum=2994
Decrement thread idx=1, priority = 97, gsum=1997
Decrement thread idx=1, priority = 97, gsum=999
Decrement thread idx=1, priority = 97, gsum=0
TEST COMPLETE
****

4d.

```c
/******************************************************************************
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable for
 * any misuse of this material.
 * ****************************************************************************/
/**
 * @file pthreads.c
 * @brief Multithreaded Fibonacci sequence computation with real-time scheduling.
 *
 * This program demonstrates multithreaded Fibonacci sequence computation
 * using POSIX threads with real-time scheduling policies. It defines three
 * threads: sequencer, fib10, and fib20, each responsible for specific tasks
 * in the Fibonacci sequence calculation.
 *
 * @authors Jithendra and Suhas
 * @date February 10, 2024
 */
#define _GNU_SOURCE
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <sched.h>
#include <time.h>
#include <semaphore.h>
#include <unistd.h>
#include <syslog.h>

#define NUM_THREADS (3)
```

```
#define ITER (3)

#define NSEC_PER_SEC (1000000000)
#define NSEC_PER_MSEC (1000000)
#define NSEC_PER_MICROSEC (1000)
#define DELAY_TICKS (1)
#define ERROR (-1)
#define OK (0)
#define TEST_CYCLES (10000000)
#define FIB_10_CI (10)
#define FIB_20_CI (20)

typedef struct
{
    int threadIdx;
} threadParams_t;

// POSIX thread declarations and scheduling attributes
//
pthread_t threads[NUM_THREADS];
threadParams_t threadParams[NUM_THREADS];
pthread_attr_t rt_sched_attr[NUM_THREADS];
int rt_max_prio, rt_min_prio;
struct sched_param rt_param[NUM_THREADS];
struct sched_param main_param;
pthread_attr_t main_attr;
pid_t mainpid;

sem_t sem_fib10, sem_fib20;
int fib10Cnt=0, fib20Cnt=0;
struct timespec start_time = {0, 0};

unsigned int idx = 0, jdx = 1;
unsigned int seqIterations = 47;
volatile unsigned int fib = 0, fib0 = 0, fib1 = 1;
volatile int abort_test = 0;

// Calculate Fibonacci sequence and store it in fib
#define FIB_TEST(seqCnt, iterCnt)      \
   for(idx=0; idx < iterCnt; idx++)    \
   {                                   \
      fib = fib0 + fib1;               \
      while(jdx < seqCnt)              \
      {                                \
         fib0 = fib1;                  \
         fib1 = fib;                   \
```

```
      fib = fib0 + fib1;            \
      jdx++;                        \
    }                               \
  }                                 \

/**
 * @brief Calculates the time difference between two timespec structures.
 *
 * This function calculates the time difference between two timespec structures
 * representing start and stop times, and stores the result in another timespec
 * structure.
 *
 * @param stop Pointer to the timespec structure representing the stop time.
 * @param start Pointer to the timespec structure representing the start time.
 * @param delta_t Pointer to the timespec structure where the time difference will be stored.
 * @return Always returns 1 to indicate success.
 */
int delta_t(struct timespec *stop, struct timespec *start, struct timespec *delta_t)
{
  // Calculate the difference in seconds and nanoseconds between stop and start times
  int dt_sec = stop->tv_sec - start->tv_sec;
  int dt_nsec = stop->tv_nsec - start->tv_nsec;

  // Adjust the time difference if necessary to ensure it's valid
  if (dt_sec >= 0)
  {
    if (dt_nsec >= 0)
    {
      // If both seconds and nanoseconds are positive or zero, no adjustment is needed
      delta_t->tv_sec = dt_sec;
      delta_t->tv_nsec = dt_nsec;
    }
    else
    {
      // If nanoseconds are negative, adjust the time difference accordingly
      delta_t->tv_sec = dt_sec - 1;
      delta_t->tv_nsec = NSEC_PER_SEC + dt_nsec; // Add one second's worth of nanoseconds
    }
  }
  else
  {
    // If seconds are negative, adjust the time difference accordingly
    if (dt_nsec >= 0)
    {
      delta_t->tv_sec = dt_sec;
      delta_t->tv_nsec = dt_nsec;
```

```c
        }
        else
        {
            // If both seconds and nanoseconds are negative, adjust the time difference accordingly
            delta_t->tv_sec = dt_sec - 1;
            delta_t->tv_nsec = NSEC_PER_SEC + dt_nsec; // Add one second's worth of nanoseconds
        }
    }

    // Return 1 to indicate success
    return 1;
}

/**
 * @brief Thread function to calculate Fibonacci sequence for FIB_10.
 *
 * This function estimates the number of iterations required to compute
 * the Fibonacci sequence for FIB_10 and performs the computation.
 *
 * @param threadp Pointer to thread parameters.
 * @return None.
 */
void *fib10(void *threadp)
{
    // Obtain the thread ID
    pthread_t fib10_t = pthread_self();

    // Define real-time scheduling parameters
    struct sched_param rt_param;

    // Define real-time scheduling policy
    int policy = SCHED_FIFO;

    // Get the thread's scheduling parameters
    pthread_getschedparam(fib10_t , &policy, &rt_param);

    // Define finish time of the thread
    struct timespec finish_time = {0, 0};

    // Define time difference for the thread
    struct timespec thread_dt = {0, 0};

    // Define thread parameters
    threadParams_t *threadParams = (threadParams_t *)threadp;

    // Define start time for testing
```

```c
    struct timespec test_start_time = {0, 0};

    // Define end time for testing
    struct timespec test_end_time = {0, 0};

    // Define time difference for testing
    struct timespec test_delta_time = {0, 0};

    // Print estimation message
    printf("Estimating FIB_10 required iterations\n");

    // Get start time for testing
    clock_gettime(CLOCK_REALTIME, &test_start_time);

    // Test Fibonacci sequence
    FIB_TEST(seqIterations, TEST_CYCLES);

    // Get end time for testing
    clock_gettime(CLOCK_REALTIME, &test_end_time);

    // Calculate testing time difference
    delta_t(&test_end_time, &test_start_time, &test_delta_time);

    // Estimate Fibonacci 10 iterations
        int fib10_iter_count = ((float)FIB_10_CI/(test_delta_time.tv_nsec / NSEC_PER_MSEC)) * TEST_CYCLES;

    // Execute Fibonacci calculations until abort_test is set
    while(!abort_test){
        // Wait for semaphore signal
        sem_wait(&sem_fib10);

        // Break loop if abort_test is true
        if(abort_test) break;

        // Get finish time
        clock_gettime(CLOCK_REALTIME, &finish_time);

        // Calculate thread time difference
        delta_t(&finish_time, &start_time, &thread_dt);

        // Print thread start message
        printf("FIB_10 %d started %ld sec, %ld msec \n", rt_param.sched_priority, thread_dt.tv_sec, (thread_dt.tv_nsec / NSEC_PER_MSEC));

        // Compute Fibonacci sequence
```

```
            FIB_TEST(seqIterations, fib10_iter_count);

        // Increment Fibonacci 10 count
        fib10Cnt++;

        // Get finish time
        clock_gettime(CLOCK_REALTIME, &finish_time);

        // Calculate thread time difference
        delta_t(&finish_time, &start_time, &thread_dt);

        // Print thread completion message
         printf("FIB_10        %d        Completed     %ld sec, %ld msec \n", rt_param.sched_priority,
thread_dt.tv_sec, (thread_dt.tv_nsec / NSEC_PER_MSEC));

        // Log thread completion
        syslog (LOG_INFO, "Thread FIB_10 completed at %ld", (thread_dt.tv_nsec / NSEC_PER_MSEC));
    }

    // Exit thread
    pthread_exit(NULL);
}


/**
 * @brief Thread function to calculate Fibonacci sequence for FIB_20.
 *
 * This function estimates the number of iterations required to compute
 * the Fibonacci sequence for FIB_20 and performs the computation.
 *
 * @param threadp Pointer to thread parameters.
 * @return None.
 */
void *fib20(void *threadp)
{
    // Obtain the thread ID
    pthread_t fib20_t = pthread_self();

    // Define real-time scheduling parameters
    struct sched_param rt_param;

    // Define real-time scheduling policy
    int policy = SCHED_FIFO;

    // Get the thread's scheduling parameters
    pthread_getschedparam(fib20_t , &policy, &rt_param);
```

```
    // Define finish time of the thread
    struct timespec finish_time = {0, 0};

    // Define time difference for the thread
    struct timespec thread_dt = {0, 0};

    // Define thread parameters
    threadParams_t *threadParams = (threadParams_t *)threadp;

    // Define start time for testing
    struct timespec test_start_time = {0, 0};

    // Define end time for testing
    struct timespec test_end_time = {0, 0};

    // Define time difference for testing
    struct timespec test_delta_time = {0, 0};

    // Print estimation message
    printf("Estimating FIB_20 required iterations\n");

    // Get start time for testing
    clock_gettime(CLOCK_REALTIME, &test_start_time);

    // Test Fibonacci sequence
    FIB_TEST(seqIterations, TEST_CYCLES);

    // Get end time for testing
    clock_gettime(CLOCK_REALTIME, &test_end_time);

    // Calculate testing time difference
    delta_t(&test_end_time, &test_start_time, &test_delta_time);

    // Estimate Fibonacci 20 iterations
        int fib20_iter_count = ((float)FIB_20_CI/(test_delta_time.tv_nsec / NSEC_PER_MSEC)) *
TEST_CYCLES;

    // Execute Fibonacci calculations until abort_test is set
    while(!abort_test){
        // Wait for semaphore signal
        sem_wait(&sem_fib20);

        // Break loop if abort_test is true
        if(abort_test) break;
```

```
        // Get finish time
        clock_gettime(CLOCK_REALTIME, &finish_time);

        // Calculate thread time difference
        delta_t(&finish_time, &start_time, &thread_dt);

        // Print thread start message
          printf("FIB_20        %d        started      %ld sec, %ld msec \n", rt_param.sched_priority,
thread_dt.tv_sec, (thread_dt.tv_nsec / NSEC_PER_MSEC));

        // Compute Fibonacci sequence
              FIB_TEST(seqIterations, fib20_iter_count);

        // Increment Fibonacci 20 count
        fib20Cnt++;

        // Get finish time
        clock_gettime(CLOCK_REALTIME, &finish_time);

        // Calculate thread time difference
        delta_t(&finish_time, &start_time, &thread_dt);

        // Print thread completion message
         printf("FIB_20        %d        Completed     %ld sec, %ld msec \n", rt_param.sched_priority,
thread_dt.tv_sec, (thread_dt.tv_nsec / NSEC_PER_MSEC));

        // Log thread completion
        syslog (LOG_INFO, "Thread FIB_20 completed at %ld", (thread_dt.tv_nsec / NSEC_PER_MSEC));
    }

    // Exit thread
    pthread_exit(NULL);
}

/**
 * @brief Prints the scheduling policy of the current process.
 *
 * This function retrieves the scheduling policy of the current process
 * and prints the corresponding string representation.
 *
 * @return None.
 */
void print_scheduler(void) {
  int schedType;

  // Get the scheduling policy of the current process
```

```
  schedType = sched_getscheduler(getpid());

  // Switch based on the scheduling policy type
  switch (schedType) {
  case SCHED_FIFO:
  // Print message for SCHED_FIFO
    printf("Pthread Policy is SCHED_FIFO\n");
    break;
  case SCHED_OTHER:
   // Print message for SCHED_OTHER
    printf("Pthread Policy is SCHED_OTHER\n");
    break;
  case SCHED_RR:
   // Print message for SCHED_RR
    printf("Pthread Policy is SCHED_RR\n");
    break;
  default:
   // Print message for unknown policy
    printf("Pthread Policy is UNKNOWN\n");
  }
}


/**
 * @brief Sequencer function to control the execution of Fibonacci threads.
 *
 * This function coordinates the execution of Fibonacci threads, controlling
 * the timing and frequency of their execution.
 *
 * @param threadp Pointer to thread parameters (unused).
 * @return None.
 */
void *sequencer(void* threadp){
    int times = 0; // Counter for the number of iterations
    while(times < ITER){ // Iterate until the specified number of iterations is reached
        // Print header for trial
        printf("----------------------------------------------------------\n");
        printf("                    Trial %d\n", (times + 1));
        printf("----------------------------------------------------------\n");
        printf("THREAD       PRIORITY         STATUS          TIMESTAMP\n");
        printf("----------------------------------------------------------\n");

        // Get current timestamp
        clock_gettime(CLOCK_REALTIME, &start_time);

        // Log scheduling start
```

```
    syslog (LOG_INFO, "*************schedule started*************");

    // Signal Fibonacci threads to start execution
    sem_post(&sem_fib10);
    sem_post(&sem_fib20);

    // Sleep to control timing
    usleep(20000);

    // Signal Fibonacci threads to continue execution
    sem_post(&sem_fib10);

    // Sleep to control timing
    usleep(20000);

    // Signal Fibonacci threads to continue execution
    sem_post(&sem_fib10);

    // Sleep to control timing
    usleep(10000);

    // Signal Fibonacci threads to continue execution
    sem_post(&sem_fib20);

    // Sleep to control timing
    usleep(10000);

    // Signal Fibonacci threads to continue execution
    sem_post(&sem_fib10);

    // Sleep to control timing
    usleep(20000);

    // Signal Fibonacci threads to continue execution
    sem_post(&sem_fib10);

    // Sleep to control timing
    usleep(20000);

    times++; // Increment trial counter

    // Print counts for fib10 and fib20 threads
    printf("----------------------------------------------------------\n");
    printf("fib10 ran count %d, fib20 ran count %d\n", fib10Cnt, fib20Cnt);
    printf("----------------------------------------------------------\n");
}
```

```c
    // Set abort_test flag to terminate threads
    abort_test = 1;

    // Signal Fibonacci threads to continue execution
    sem_post(&sem_fib10);
    sem_post(&sem_fib20);

    // Exit sequencer thread
    pthread_exit(NULL);
}

/**
 * @brief Main function to control the execution of the program.
 *
 * This function initializes parameters, creates threads, sets scheduling policies,
 * and waits for threads to complete.
 *
 * @return 0 on successful execution.
 */
int main(){

    int rc;
    int i = 0;

    // Print start message
    printf("TEST STARTED\n");

    // Open system log
    openlog ("Fibonacci", LOG_CONS | LOG_PID | LOG_NDELAY, LOG_LOCAL1);
    syslog (LOG_INFO, "TEST_STARTED");

    // Print current scheduler type
    print_scheduler();

    // Get scheduling parameters of main process
    rc = sched_getparam(getpid(), &main_param);
    rt_max_prio = sched_get_priority_max(SCHED_FIFO);
    rt_min_prio = sched_get_priority_min(SCHED_FIFO);
    printf("rt_max_prio=%d\n", rt_max_prio);
    printf("rt_min_prio=%d\n", rt_min_prio);

    // Set main process priority to maximum
    main_param.sched_priority = rt_max_prio;
    rc = sched_setscheduler(getpid(), SCHED_FIFO, &main_param);
    if (rc < 0)
```

```
    perror("main_param");

// Print updated scheduler type
print_scheduler();

// Initialize semaphores
sem_init(&sem_fib10, 0, 0);
sem_init(&sem_fib20, 0, 0);

// Define the thread routine function pointer type
typedef void *(*Threadpointer)(void*);
Threadpointer Threads[] = {sequencer, fib10, fib20};

// Create threads for Fibonacci computations
for (i = 0; i < NUM_THREADS; i++) {
    threadParams[i].threadIdx = i;
    // Create thread with FIFO scheduling policy
    pthread_attr_t attr;
    rc = pthread_attr_init(&attr);
    rc = pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
    rc = pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
    rt_param[i].sched_priority = rt_max_prio - i - 1;
    pthread_attr_setschedparam(&attr, &rt_param[i]);

    // Decide thread function based on thread index
    rc = pthread_create(&threads[i], (void *)&attr, Threads[i], (void *)&(threadParams[i]));
    if (rc) {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
    }
}

// Wait for threads to complete
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(threads[i], NULL);

// Destroy semaphores
sem_destroy(&sem_fib10);
sem_destroy(&sem_fib20);

// Print completion message
printf("TEST COMPLETE\n");

// Close system log
closelog ();
```

```
    return 0;
}
```

****

Output: By running the bash script run_pthread.sh present the fibonacci.zip the following output can be produced.

```
gcc -O3   -c pthread.c
gcc -g  -O3   -o pthread pthread.o -lpthread
TEST STARTED
Pthread Policy is SCHED_OTHER
rt_max_prio=99
rt_min_prio=1
Pthread Policy is SCHED_FIFO
----------------------------------------------------------
                Trial 1
----------------------------------------------------------
THREAD      PRIORITY       STATUS       TIMESTAMP
----------------------------------------------------------
Estimating FIB_10 required iterations
FIB_10         97          started       0 sec, 7 msec
FIB_10         97          Completed      0 sec, 15 msec
Estimating FIB_20 required iterations
FIB_10         97          started       0 sec, 20 msec
FIB_10         97          Completed      0 sec, 28 msec
FIB_20         96          started       0 sec, 30 msec
FIB_10         97          started       0 sec, 40 msec
FIB_10         97          Completed      0 sec, 48 msec
FIB_20         96          Completed      0 sec, 48 msec
FIB_20         96          started       0 sec, 50 msec
FIB_20         96          Completed      0 sec, 59 msec
FIB_10         97          started       0 sec, 60 msec
FIB_10         97          Completed      0 sec, 68 msec
FIB_10         97          started       0 sec, 80 msec
FIB_10         97          Completed      0 sec, 90 msec
----------------------------------------------------------
fib10 ran count 5, fib20 ran count 2
----------------------------------------------------------
----------------------------------------------------------
                Trial 2
----------------------------------------------------------
THREAD      PRIORITY       STATUS       TIMESTAMP
----------------------------------------------------------
FIB_10         97          started       0 sec, 0 msec
FIB_10         97          Completed      0 sec, 9 msec
FIB_20         96          started       0 sec, 9 msec
```

FIB_10 97 started 0 sec, 20 msec
FIB_10 97 Completed 0 sec, 29 msec
FIB_20 96 Completed 0 sec, 29 msec
FIB_10 97 started 0 sec, 40 msec
FIB_10 97 Completed 0 sec, 49 msec
FIB_20 96 started 0 sec, 50 msec
FIB_20 96 Completed 0 sec, 60 msec
FIB_10 97 started 0 sec, 60 msec
FIB_10 97 Completed 0 sec, 68 msec
FIB_10 97 started 0 sec, 80 msec
FIB_10 97 Completed 0 sec, 90 msec
----
fib10 ran count 10, fib20 ran count 4

Trial 3

THREAD PRIORITY STATUS TIMESTAMP

FIB_10 97 started 0 sec, 0 msec
FIB_10 97 Completed 0 sec, 9 msec
FIB_20 96 started 0 sec, 9 msec
FIB_10 97 started 0 sec, 20 msec
FIB_10 97 Completed 0 sec, 28 msec
FIB_20 96 Completed 0 sec, 28 msec
FIB_10 97 started 0 sec, 40 msec
FIB_10 97 Completed 0 sec, 55 msec
FIB_20 96 started 0 sec, 55 msec
FIB_10 97 started 0 sec, 60 msec
FIB_10 97 Completed 0 sec, 70 msec
FIB_20 96 Completed 0 sec, 74 msec
FIB_10 97 started 0 sec, 81 msec
FIB_10 97 Completed 0 sec, 89 msec
----
fib10 ran count 15, fib20 ran count 6

TEST COMPLETE
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: TEST_STARTED
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: *************schedule started*************
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 15
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 28
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 48
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_20 completed at 48
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 68
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_20 completed at 59

Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 90
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: *************schedule started*************
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 9
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 29
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_20 completed at 29
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 49
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_20 completed at 60
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 68
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 90
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: *************schedule started*************
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 9
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 28
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_20 completed at 28
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 55
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 70
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_20 completed at 74
Feb 10 15:07:59 DESKTOP-4OVQQO0 Fibonacci[1216]: Thread FIB_10 completed at 89

References:

1. GitHub - siewertsmooc/RTES-ECEE-5623