

# Real-Time Embedded Systems

## Exercise-1

Satya Mehta & Vatsal Sheth

Date:- 02/05/2019

### Q1). Rate Monotonic analysis and Timing diagrams.

Draw a timing diagram with three services S1, S2, S3 with T1=3, C1=1, T2=5, C2=2, T3=15, C3=3 where all times are in milliseconds. Label your diagram carefully and describe whether you think the schedule is feasible and safe.

#### Timing Diagram:



#### Feasibility and safe:

The given set of tasks is schedulable in the LCM of their time periods T1, T2, T3. Due to this, we see that the schedule is mathematically repeatable over any multiple of the LCM making it feasible.

According to Liu-Layland paper it is advised that there should be around 30% free of the total CPU. In the above example there is 93% CPU utilization hence the system is **not safe**.

#### Utility and Method used to generate the timing diagram:

- Service S1 comes every T1 period which is 3. The computation time C1 is 1. Hence, we allotted the S1 every T1 period.
- Service S2 comes every T2 period which is 5 and the computation time C2 is 2. After scheduling S1 we scheduled S2 at first available time slots in each of its request such that it meets the deadline.
- Service S3 comes every T3 period which is 15 and the computation time C2 is 3. S3 is also scheduled similarly to S2 by placing in the first available time slots such that it meets the deadline.

#### CPU Utilization:

The CPU utilization can be obtained from the formula:

$$U = \sum_{i=1}^m (C_i/T_i) \leq m(2^{1/m} - 1)$$

where  $i$  is the number of services. Considering this example, the CPU utilization obtained is around 93%.

## Q2) Shared CPU system overload.

### a). Summary, Root cause analysis and description of Apollo Lunar Lander Computer overload story:

An Apollo Guidance Computer was used in the command module and the lunar module for navigation assistance and to control the spacecraft. This computer was resource overloaded which nearly resulted in the aborting the mission but thanks to Margaret Hamilton, a software engineer who wrote very robust software for the guidance computer. The software was real-time operating system, interrupt driven and time dependent.

#### Memory Structure and its use in the guidance computer.

The computer had 36864 15-bit words fixed memory [ROM] and 2048 words of erasable memory [RAM]. Most of the program code, constants and similar data was stored in the fixed memory. The processes used the small erasable memory to store the data. As the new process required some memory the old data from the erasable memory was cleared and was allocated to the new process to store its data. Some memory locations were shared seven ways. Hence, it was a difficult task to ensure that the same memory was not used by more than one process at the same time.

Each process had a core set of 12 erasable memory allocation and if required any temporary storage they were allotted with VAC (Vector Accumulator). In some cases, where there are no VAC's available, Alarm 1201 are set and when there are no core sets available Alarm 1202 are set.

#### Resource Overloading.

During the landing of the Apollo, there were repeated jobs lined up to process rendezvous radar data which were scheduled because of the misconfiguration of the radar switches which caused continuous request for the core sets by the processes. This caused 1202 to generate. Also, there was one scheduled event which caused which requested VAC and due to its unavailability 1201 got generated.

#### Successful Landing.

The software designed by Margaret Hamilton was such that it gives priority to the important services. Hence, it ignores the less priority services. During the landing of the lunar module the software continuously rebooted and reinitialized some of the important tasks clearing all the processes generated by the radar switches. Since, the NASA people at the MIT Lab knew that the Alarms can be ignored, they decided not to abort the landing and go forward.

### b). How was the Rate Monotonic Policy violated?

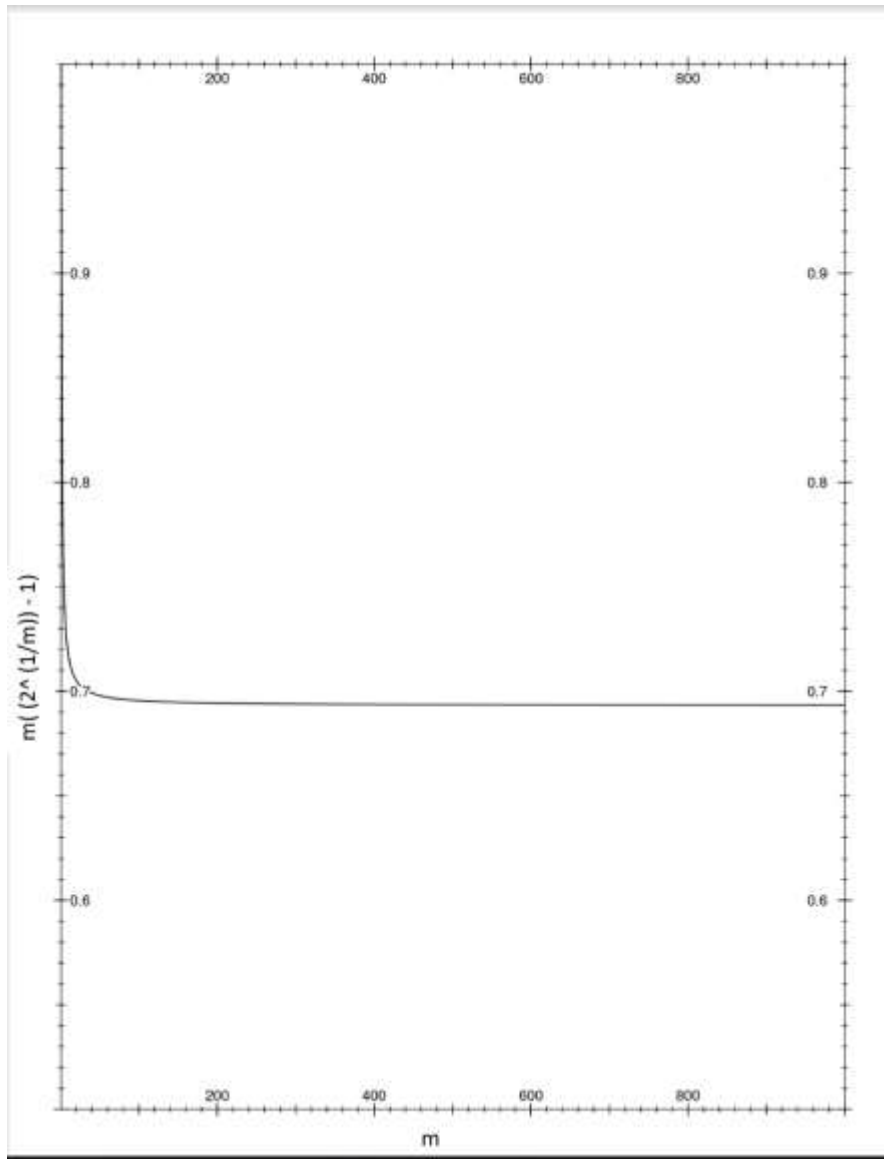
The rate monotonic policy gives highest priority to the most frequently requested tasks. In this case the processes generated by the radar switches were highly frequent and still they were ignored by the system and were given no priority. Hence, Rate monotonic policy was violated.

c) Arguments against Rate Monotonic Policy and analysis prevention of Apollo11 overload scenario.

We don't think analysis of Rate Monotonic Policy in this scenario would have prevented overload. Rate Monotonic Policy gives highest priority to the most frequent task with safety margin of around 30% as per Liu-Layland paper. Errors were coming at very high frequency which rebooted the system so in any case with this high error rate 30% safety margin would have been less.

As the Rate Monotonic Policy is the fixed priority policy so it doesn't take into account the fact that execution time of the radar computations was increasing due to misconfiguration of the radar switches and also due to preemption by error.

c). Plot for LUB as the function of number of services.



Description of Margin: - This margin gives the LUB for the intersection of case-1 and case-2 of derivation which basically means that the ratio of any two request periods of the set of task is less than 2. Given this condition the above plot show the LUB with increase in number of services from 1 to 1000.

d) Describe three key assumptions in Liu Layland paper and describe three aspects of fixed priority LUB which we don't understand.

Key Assumptions: -

- All requests for services are periodic.
- Each task must complete before next occurs.
- Runtime for each task is constant for that task.
- Higher priority task is always executed by the processor.

Three aspects of Fixed priority of LUB which we don't understand: -

- In theorem 4, we didn't understand the assumption that  $C2'' = C2 - 2\Delta$  for given  $C1 = T2 - T1 - \Delta$ .
- In theorem four, equation 4, we get the value of the utilization  $U$  in terms of  $g_i$ . To calculate the LUB we need to find minima of this function. This can be done by differentiating this equation and equating it to 0. We need to solve this differential equation to find the value of  $g$  for which the function gives minimum value. We substitute this value back to the main equation and simplify it to get the value of LUB.

$$U = m(2^{1/m} - 1)$$

We were not able to solve this maths.

- We were not able to understand the physical significance of LUB of the processor utilization.

### Q3) POSIX Real Time Clock.

a). Code Description.

The objective of the code is to measure the time difference between two points in code during execution, specifically the actual time elapsed when processor is kept on sleep for 3 seconds.

Initially we check the pthread scheduling policy and display it by using `print_scheduler()` function. The function uses `sched_getscheduler()` system call to retrieve the policy which is currently used by the system. Depending on whether 'RUN\_RT\_THREAD' is defined, the time measurement is done by creating thread using FIFO Scheduling policy or running the required delay test function directly.

With this switch set, firstly we initialized the pthread attribute named `main_sched_attr`. Then we set the inheritance of this attributes to `EXPLICIT`, this specifies that the thread scheduling attributes shall be set from the values of this attribute object. Then set the scheduling policy of this attribute to FIFO Scheduling. System call is used to set the process' scheduling policy to FIFO policy and priority to maximum. On successful completion of system call we display the current scheduler policy. Now we set the pthread's attribute to highest priority and create the pthread to run `delay_test()` function. The function

pthread\_join() is used to halt the execution of main() till the completion of the thread and then the thread is destroyed.

#### Function descriptions:

- print\_scheduler() – It finds the scheduler policy of the process from which it is called and then it is displayed.
- delta\_t(struct timespec \*stop, struct timespec \*start, struct timespec \*delta\_t) – This function calculates the time difference between start and stop.
- delay\_test(void \*threadid) – This function initially retrieves the clock resolution of CLOCK\_REALTIME which represents the time since epoch in nanoseconds and displays it. Clock time is noted at this point in a variable named as rtclk\_start\_time. Then function nanosleep() is used to send processor to sleep for 3 seconds. If the processor is interrupted and awoken from the sleep then it is again sent to sleep for remaining amount of time provided number of wake up calls are less than max sleep calls which is 3 in this case. Clock time is noted at this point in a variable named as rtclk\_stop\_time. Time difference between start and stop is measured using delta\_t function. This time difference is subtracted from the sleep time of 3 seconds to measure the delay error. Delay error is basically the overhead on the system while sleeping. If in case number of interruptions is more than the max sleep calls and the loop breaks before the sleep time then the delay error can be negative.
- end\_delay\_test(). It displays various time results and measurements.

#### Code output executed on Raspberry Pi 3b+:

##### 1). Using SCHED\_FIFO Policy: -

```
rm -f posix_clock
pi@raspberrypi:~/Desktop/Satya/RT-Clock $ make
gcc -MD -O3 -g -c posix_clock.c
gcc -O3 -g -o posix_clock posix_clock.o -lpthread -lrt
pi@raspberrypi:~/Desktop/Satya/RT-Clock $ ./posix_clock
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
ERROR: sched_setscheduler rc is -1
sched_setscheduler: operation not permitted
pi@raspberrypi:~/Desktop/Satya/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 = 1549222978, nanoseconds = 166192281
RT clock stop seconds = 1549222981, nanoseconds = 166258488
RT clock DT seconds = 3, nanoseconds = 58207
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 58207
pi@raspberrypi:~/Desktop/Satya/RT-Clock $
```

## 2) Using SCHED\_OTHER Policy: -

```
pi@raspberrypi:~ $ ls
Desktop  Downloads  Music      Public     Videos
Documents  MagPi      Pictures  Templates
pi@raspberrypi:~ $ cd Desktop/
pi@raspberrypi:~/Desktop $ cd Satya/
pi@raspberrypi:~/Desktop/Satya $ cd RT-Clock/
pi@raspberrypi:~/Desktop/Satya/RT-Clock $ make
gcc -MD -O3 -g -c posix_clock.c
gcc -O3 -g -o posix_clock posix_clock.o -lpthread -lrt
pi@raspberrypi:~/Desktop/Satya/RT-Clock $ ./posix_clock
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER

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

RT clock start seconds = 1549222465, nanoseconds = 867375267
RT clock stop seconds = 1549222468, nanoseconds = 867494109
RT clock DT seconds = 3, nanoseconds = 118842
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 118842
pi@raspberrypi:~/Desktop/Satya/RT-Clock $
```

Clock\_gettime():- We pass clock\_realtime as a parameter to this function and we get back time in nanoseconds range.

b) Why Low Interrupt Latency, Low Context switch time, timer interrupts, timeouts, Low jitter and drift are important? RTOS Bragging point.

### Low Interrupt Latency.

Interrupt latency is the time required for the processor to act upon after the interrupt has been generated. Generally, the processor takes some time to save the states of the normal operation in the registers and to make a switch to execute the ISR. In real time applications there are many prioritized interrupts required to be process. If the latency is high, then the processor will take more time to switch to ISR of the concerned interrupt. It is possible that in the same time there are other interrupts which arrive and wait to get acted upon. This will increase the number of interrupts which are waiting to be processed leading to delay which may lead to system failure in Hard Real-Time applications.

### Low Context Switch time.

A context switch time is the time required by the system to switch from one process to another. In real-time applications there are multiple processes which run simultaneously and are dependent on each other. Basically, there is some synchronization between multiple tasks in the end application. If the context switch time is high, then there will be synchronization failure between tasks resulting in delay in end application. For example, in video application if there is delay in switching between video frames and audio you may see undesirable video.

Stable timer services along with Low jitter and drift.

In some applications the processes use timer services to generate interrupts. Timeouts are used to generate interrupt or to actuate a specific task. Stable timer service allows to generate interrupts at expected time providing stable output. Jitter and drift are the factors which define shift in the timer services from the expected output. Jitter refers to variation in the functioning of the oscillator or PLL due to electrical disturbances and drift refers to the variation in the oscillator accuracy over environmental changes. It is expected that the jitter and drift should be as low as possible, so we can get accurate timing services over the period. Some vendors provide automatic calibration of the oscillator through firmware to avoid differences in timing services due to jitter and drift.

### c) Is RT\_CLOCK accurate?

RT\_CLOCK in this code uses clock id CLOCK\_REALTIME. This measures the real-time or 'wall clock' which is the time in nanoseconds since epoch. This clock can be affected by manual changes or adjustments performed by adjtime() function. So, if this time is changed than relative intervals will be unaffected but the time for absolute point is affected. Execution time of a task if measured through wall time is less then the measurement through CPU cycle, it means the task was executing concurrently in multiple cores.

RT\_CLOCK is **relatively less accurate** compare to CLOCK\_THREAD\_CPUTIME\_ID which uses CPU clock cycles to measure the time difference. As we know the processor frequency and clock cycles utilized we can get accurate time elapsed between two points.

## Q4) Code description

### a). Simple Thread:

The main objective of this code is to create 12 threads and calculate Fibonacci series on the basis of thread id. A macro named as NUM\_THREADS is assigned 12 which is used to create 12 threads. In the main function the threads are created using default attributes and giving counterThread() function as a function entry point. The created threads are then joined using pthread\_join() function.

The counterThread() function calculates Fibonacci series based on the thread id. When a particular thread runs, its id number get added into the variable sum.

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

## b). Rt\_Simplethread:

**EDIT:** Previously, while creating the threads in the following codes default attributes configurations were used ignoring the attributes configuration done before in the code. We changed the second parameter to configured attributes and executed the code.

Before, if we don't make changes and run the code as it is, the execution was depended on the SCHED\_OTHER policy. Also, all the live cores were affined to each thread rather than single core. Snippets are attached which includes output of rt\_simplethread and rt\_simplethread\_improved with and without changes in the code.

Objective of this code is to create a thread and compute Fibonacci series multiple times and measure this execution time.

Initially in main(), we clear the cpu\_set\_t object and then set the bit mask for processor number 0 which will be later used to set the CPU affinity of pthread attribute. Then, we print the current scheduler policy which is SCHED\_OTHER as default. Further going we get the process id of this main() and set it's priority to maximum and scheduling policy to SCHED\_FIFO. Then we get the scope of this pthread and print the scheduling policy and its scope which is system scope. System scope for a thread specifies that it can compete with all other threads from its calling process or other processes for resources. Info about max and min priority value is printed.

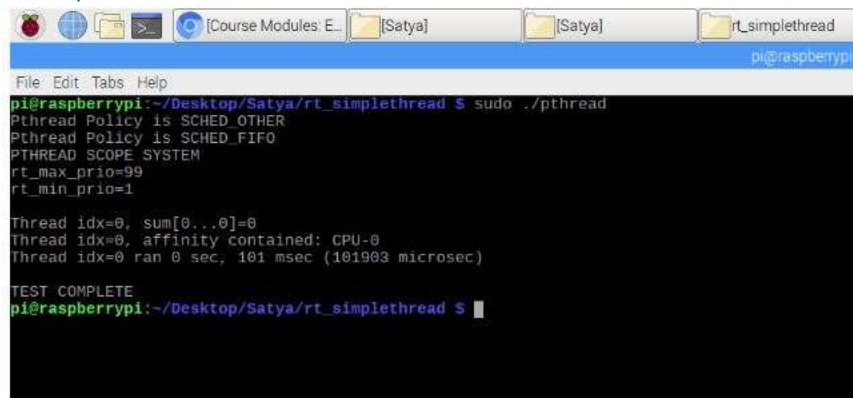
Thread is created to execute counterThread(). Before that this new thread's attributes needs to be configured. Its inheritance is set to explicit, scheduling policy is SCHED\_FIFO and CPU affinity set to processor number 0. After successful creation this thread it is joined with the main process and the test ends.

## Function Description

- Print\_scheduler function gets the current process id and prints its scheduling policy.
- delta\_t function : This function calculates the time difference between start and stop.
- **FIB\_TEST macro** calculates Fibonacci series of 47 length 1400000 times. It is used to kill CPU time.
- counterThread() function is the one which thread executes. First of all, clock time is noted in start\_time variable. It does two tasks to kill time, one is to find the sum till thread id and second is to calculate Fibonacci series. It then prints this results, CPU affinity of the thread which is executing this function and the CPU it uses. Clock time is noted in stop\_time variable. Execution time of this task is calculated as the difference of start and stop time and is printed. And then the thread is exited.



Rt\_Simplethread output: -



```
pi@raspberrypi:~/Desktop/Satya/rt_simplethread $ sudo ./pthread
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Thread idx=0, sum[0...0]=0
Thread idx=0, affinity contained: CPU-0
Thread idx=0 ran 0 sec, 101 msec (101903 microsec)
TEST COMPLETE
pi@raspberrypi:~/Desktop/Satya/rt_simplethread $
```

### c). Rt\_simple\_improved:

Objective of this code is to create 4 threads and compute Fibonacci series in each thread and measure this execution time.

Initially in main(), we get the processor cores configured on the system. Individual processor cores are used by the relative individual thread. The information of the processor usage is displayed. We clear the `cpu_set_t` object and then set the bit mask for all the available processors. This will be later used to set the CPU affinity of pthread attribute. Then, we print the current scheduler policy which is SCHED\_OTHER as default. Now, we get the process id of this main() and set its priority to maximum and scheduling policy to SCHED\_FIFO. Then we get the scope of this pthread and print the scheduling policy and it's scope which is system scope. System scope for a thread specifies that it can compete with all other threads from it's calling process or other processes for resources. Info about max and min priority value is printed.

Four threads are created to execute counterThread() function. Before that this new thread's attributes needs to be configured. All of these threads inheritance is set to explicit, scheduling policy is SCHED\_FIFO. CPU affinity of respective threads is set to processor number 0 and successively till modulo of number of available processors. For example, with four thread and two cores, affinity of thread 0 will be core 0, thread 1 will be core 1, thread 2 will be core 0 and thread 3 will be core 1. Similarly thread 0 is assigned priority 98 and next threads are assigned respective priority in decreasing order. After successful creation this thread, they are joined with the main process and the test ends.

#### Function description.

- Print\_scheduler function gets the current process id and prints its scheduling policy.
- delta\_t function: This function calculates the time difference between start and stop.
- FIB\_TEST macro calculates Fibonacci series of 47 length 10000000 times. Its use is just to kill CPU time.
- counterThread() function is the one which thread executes. First, clock time is noted in start\_time variable. It does two tasks to kill time, one is to find the sum till thread id and second is to calculate

Fibonacci series. It then prints this results, CPU affinity of the thread which is executing this function and the CPU core it uses. Clock time is noted in stop\_time variable. Execution time of this task is calculated as the difference of start and stop time and is printed. And then the thread is exited.

Rt\_simple\_improved output:

```
gcc -O0 -g -o pthread pthread.o -lpthread
pi@raspberrypi:~/Desktop/Satya/rt_thread_improved $ sudo ./pthread
This system has 4 processors configured and 4 processors available.
number of CPU cores=4
Using sysconf number of CPUS=4, count in set=4
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Setting thread 0 to core 0
CPU-0
Launching thread 0
Setting thread 1 to core 1
CPU-1
Launching thread 1
Setting thread 2 to core 2
CPU-2
Launching thread 2
Setting thread 3 to core 3
CPU-3
Launching thread 3

Thread idx=3, sum[0...400]=79800
Thread idx=3 ran on core=3, affinity contained: CPU-3
Thread idx=0, sum[0...100]=4950

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

Thread idx=0 ran 1 sec, 236 msec (236090 microsec)
CPU-1

Thread idx=1 ran 1 sec, 236 msec (236012 microsec)

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

Thread idx=2 ran 1 sec, 235 msec (235929 microsec)

Thread idx=3 ran 1 sec, 235 msec (235554 microsec)

TEST COMPLETE
pi@raspberrypi:~/Desktop/Satya/rt_thread_improved $
```

Linux replication of VxWorks for LCM Invariant schedule.

Objective of this code is to run 10ms Fibonacci and 20ms Fibonacci series every 20ms and 50ms respectively till the time external abort signal is given.

Code Flow:

->> Main Function.

1. Initialization of Semaphores using sem\_init() function for individual threads of Fib10 and Fib20.
2. Initialization of variables required during the code run.
3. Configuration of the attributes and changing the scheduling policy to SCHED\_FIFO.
4. Assigning the priorities to the threads. Fib10 -98 and Fib20- 97.
5. Set the scheduler to SCHED\_FIFO.
6. Create Pthread using pthread\_create() function choosing individual thread functions as the entry point.

7. Implementing the RM Policy on LCM invariant schedule assigning proper delay using `usleep()` and `sem_post()` based on the timing diagram implemented. This step will run continuously in a loop till abort signal is provided.
8. Execute `pthread_join()` function and destroy the threads and semaphores and print results.

->> Thread Function to calculate 10ms Fibonacci.

1. Set CPU affinity to the Core 0.
2. While loop is executed till external abort signal is given.
  - Hold the thread execution using `sem_wait()`.
  - Execute the Fibonacci for 10ms delay.
3. Exit from the function.

Thread Function to calculate 20ms Fibonacci.

4. Set CPU affinity to the Core 0.
5. While loop is executed till external abort signal is given.
  - Hold the thread execution using `sem_wait()`.
  - Execute the Fibonacci for 20ms delay.
6. Exit from the function.

#### Threading vs Tasking:

Thread is a concept for OS which uses process models. This process can either have a single thread or can have multiple threads all of which executes parallelly in the current process memory. For VxWorks there is no process model so everything runs in same memory space. In VxWorks all code except interrupt handler runs in task context. VxWorks provides the option of creating POSIX threads using the `pthread` library, but VxWorks takes this thread as task only. It implements that, by wrapping the threads as task so that it executes parallelly with main routine.

So, the main difference between the task and thread is that, threads don't have their separate memory they just run with their process stack. Whereas, the task has its own stack area. Hence, the context switching is faster in tasks.

`Pthread_create()` in linux is analogous to the `task_spawn` in VxWorks. Similarly, `sem_post()` and `sem_wait()` are analogous to `semGive()` and `semTake()`. `Sem_init()` is analogous to `SemBCreate()` in vxworks.

#### Semaphores wait and sync:

Synchronizations between two threads is achieved with the help of semaphores. After initialization of the semaphore using `sem_init` we create threads with their CPU affinity set to different cores. When the threads are created using their respective entry point we use `sem_wait()` function to hold the execution of all threads.

After creation of all the threads `Sem_post` is used to increment the semaphore of all the threads which enables the execution of the threads which were blocked in a `sem_wait()` state. Thus, all the waiting threads will start execution in sync in different cores.

### Synthetic Workload Generation:

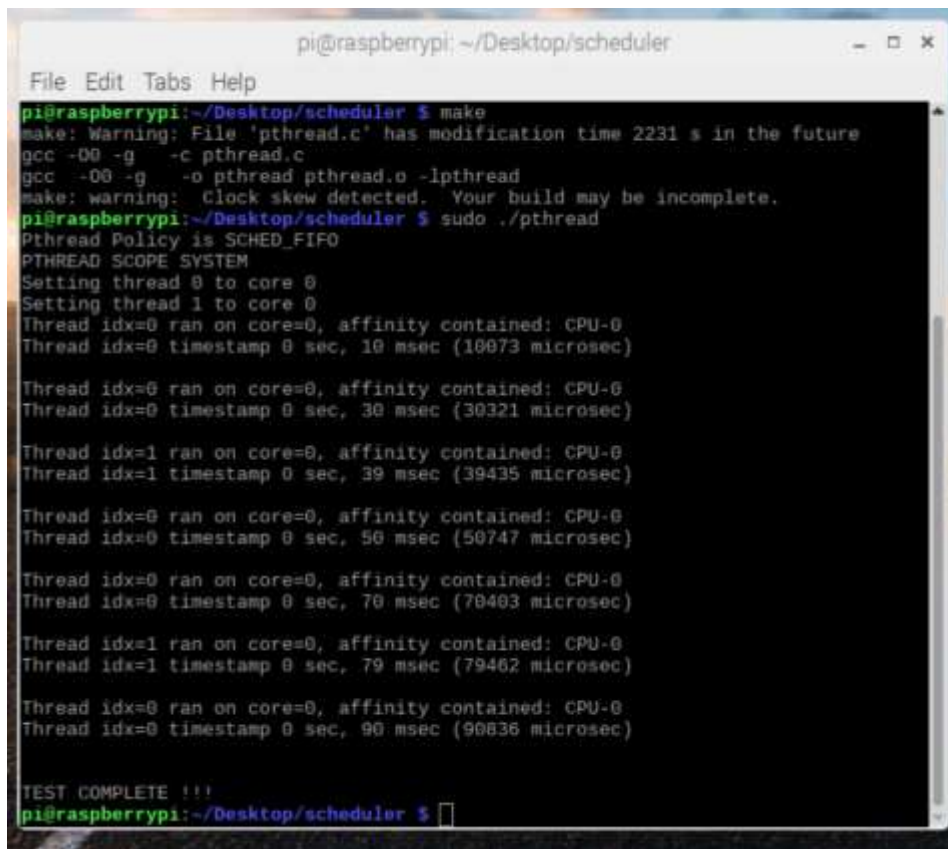
Synthetic workload generators are task used to create dummy computations to precisely kill the require amount of time. For example, we have created Fibonacci series to run for 10ms. So, this function can be used at places where we want to execute the thread for 10ms.

### Synthetic workload analysis and adjustment of test system:

Synthetic workload task includes Fibonacci computation in the code. We have two threads which computes Fibonacci for 10ms and 20ms each. Initially, our ported code was executed on the virtual machine with iterations as 10000000 and 20000000 for two threads. We calculated the time consumed for the above iterations.

Based on the time consumption for the default iteration count, we divided the default iterations with the time delay required. Accordingly, we selected the iteration count to achieve coarse adjustments. We then fine-tuned this iteration count to achieve the delay of 10ms and 20ms for respective threads.

### Code Output:



```
pi@raspberrypi: ~/Desktop/scheduler
File Edit Tabs Help
pi@raspberrypi:~/Desktop/scheduler $ make
make: Warning: File 'pthread.c' has modification time 2231 s in the future
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
make: warning: Clock skew detected. Your build may be incomplete.
pi@raspberrypi:~/Desktop/scheduler $ sudo ./pthread
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
Setting thread 0 to core 0
Setting thread 1 to core 0
Thread idx=0 ran on core=0, affinity contained: CPU-0
Thread idx=0 timestamp 0 sec, 10 msec (10073 microsec)

Thread idx=0 ran on core=0, affinity contained: CPU-0
Thread idx=0 timestamp 0 sec, 30 msec (30321 microsec)

Thread idx=1 ran on core=0, affinity contained: CPU-0
Thread idx=1 timestamp 0 sec, 39 msec (39435 microsec)

Thread idx=0 ran on core=0, affinity contained: CPU-0
Thread idx=0 timestamp 0 sec, 50 msec (50747 microsec)

Thread idx=0 ran on core=0, affinity contained: CPU-0
Thread idx=0 timestamp 0 sec, 70 msec (70403 microsec)

Thread idx=1 ran on core=0, affinity contained: CPU-0
Thread idx=1 timestamp 0 sec, 79 msec (79462 microsec)

Thread idx=0 ran on core=0, affinity contained: CPU-0
Thread idx=0 timestamp 0 sec, 90 msec (90636 microsec)

TEST COMPLETE !!!
pi@raspberrypi:~/Desktop/scheduler $
```

### Overall Description and challenges faced:

The main objective of this exercise is to learn how to schedule services theoretically and verifying it practically. We use timing diagram to implement Rate Monotonic Policy and implement it on Linux OS using Posix threads and semaphores.

We studied about Apollo11 Computer overload and related it with the rate monotonic policy. Also, we studied about the LUB as derived in the Liu-Layland paper. Further going, we studied how RT-Clock works in Linux and why it is less accurate than other sources.

Lastly, we implemented the scheduling policy in Linux using semaphores and pthreads and analysed the Rate Monotonic Policy.

### Challenges faced:

- Understanding code, working of pthreads and semaphores.
- Calculating exact iteration value for computation time of two threads which changes for different machines.

### References: -

- Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment – C.L. Liu, James W. Layland.
- Posix Thread Programming - <https://computing.llnl.gov/tutorials/pthreads/>
- RM Scheduling Feasibility Tests conducted on TI DM3730 Processor-1Ghz ARM Cortex-A8 core with Angstrom and TimeSys Linux ported on to BeagleBoard xM-  
<http://ecee.colorado.edu/~ecen5623/ecen/labs/Linux/IS/Report.pdf>