**ECEN 5623 – REAL TIME EMBEDDED SYSTEMS – SRICHARAN KIDAMBI**

**Reason for Late submission: Unable to obtain hardware for installation of RPI. (The device I possessed doesn’t have a HDMI port and had to get all the necessary equipment like keyboard, monitor and mouse etc.)**

**In addition to that I had some technical Glitches which I had to wait a long time to solve. Initially RPI board got hung up multiple times.**

1. [15 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 here – 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?

**Solution:**

1. Rate Monotonic theory: - **TIMING DIAGRAM**

**Basic Definition:** Rate monotonic Scheduling is a fixed priority scheduling algorithm that states that “The task out of a given task set multiplexed in a single CPU which requests more frequently will be given the higher priority” in this type of scheduling.

* 1. Timing Diagram for the Request task set with Rate Monotonic Scheduling.

S1 requests for 1 unit of work to be done every 3 milliseconds

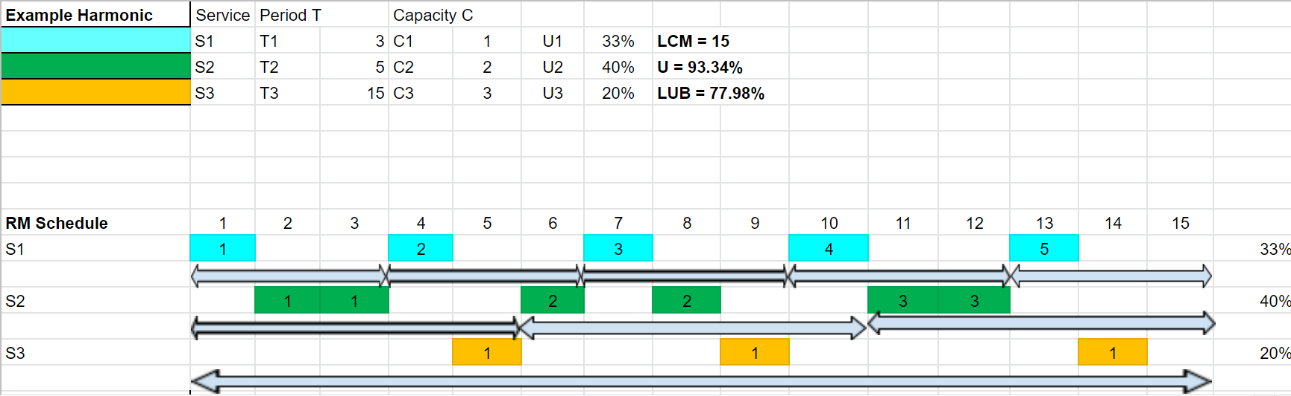
S2 requests for 2 units of work to be done every 5 milliseconds

S3 requests for 3 units of work to be done every 15 milliseconds

Now, let us consider the Deadline of all the 3 tasks to be the LCM of the periods of all the 3 tasks.

The system is said to be feasible if there is no overflow of any task at the deadline.

Please have a look at the timing diagram for the 3 services below executed by rate monotonic scheduling



**All timing units are in Milli Seconds.**

* 1. **Schedule feasibility and Safety:**
     1. Liu and Layland’s Paper on Fixed Priority Scheduling states that a task is said to feasible if the tasks are scheduled so that no overflow occurs.

The above timing diagram having a CPU utilization is 93.34% this is a clear declaration that the CPU is not overloaded and is a feasible process with rate monotonic scheduling. It is free for 1 millisecond out of every 15 milliseconds.

* + 1. All tasks have a single fixed WCET, which can be trusted as safe are tightly upper bound

A CPU scheduled with fixed priority scheduling is considered safe if its percentage utilization is less than the least upper bound curve of that set of services.

**Calculation of Total CPU Utilization:** U = ∑ (Ci/Ti), where i ranges from 0 to n

n = number of services requested by the CPU

U = 1/3 + 2/5 + 3/15 = 0.334 + 0.4 + 0.2 = 0.934

Therefore, the CPU is 93.34 % utilized.

**Calculation of LUB:** According to Liu and Layland’s paper on Scheduling Mechanisms.

**LUB** of any fixed priority scheduling of n tasks = n (21/n – 1)

**Therefore, LUB = 3 (21/3 – 1) => 0.7798**

CPU’s least upper bound is 77.98 %.

Hence, we can conclude that, the system is **not safe enough** (we can’t confirm the safety of this system), but it is feasible to design such a system using rate monotonic scheduling.

* 1. **Total CPU utilization** – Sum of ratio of Capacity and Period of all the tasks in the task set of the CPU. i.e., ∑ (Ci/Ti), where i ranges from 0 to n and where n is the number of services requested by the processor.

Hence U = (1/3) + (2/5) + (3/15) => 0.334 + 0.4 + 0.2 => 0.9334

**93.34 % CPU utilized by the above services.**

**References:**

* [Rate-Monotonic-Theory-Liu-and-Layland.pdf - Google Drive](https://drive.google.com/file/d/1aPUwPSREH8PUa2oDnaUVP4-OqAX9RQys/view) – Liu and LayLand’s Paper
* <https://www.youtube.com/watch?v=TDR-rgWopgM> – A view on SCHED\_DEADLINE
* <https://docs.google.com/spreadsheets/d/1NkpFfMW5_Vws27cS3Uo910rnMPL2oxBJ/edit#gid=1967144253>
* <https://www.coursera.org/lecture/real-time-embedded-theory-analysis/rm-lub-derivation-introduction-fOM43>

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 increase.

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

d) Describe three key assumptions Liu and Layland make and document three or more aspects of their fixed priority LUB derivation that you do not understand.

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

**SOLUTION:**

Apollo 11 LGC (Lunar Guidance Computer) Alarm Hazard

**Technical Details of Flight Hardware and Software**

Before getting into the technical details, lets discuss what happened to the Lunar Guidance Computer of Apollo 11

The apollo 11 system buffered CPU resource overload that threatened to cause descent guidance services to miss deadlines and almost resulted in aborting the first landing to the moon. The alarm 1202 triggered because of more computing than could be handled by the required deadlines when alarm processing was added to the normal workload. The alarm which was added was alarm 1201.

Now we will examine why alarm 1201 triggered

The computer consisted of 36,864 15-bit words of ROM memory, 2048 words of RAM memory.

Most of the executable code present in the ROM, along with constants and similar data.

RAM size was small and was used to store variables. Since RAM was size to allocate space for each process, same memory address was stored for different processes for different times.

The LM descent engine ran some priority ordered jobs with less time critical things. The RAM was used to compute intermediate results and was shared between the programs.

Each job was allocated a “core set” consisting of 12 RAM memory locations.

A job requiring more temporary storage can request for a VAC (vector accumulator) – which had 44 words. If the current job that’s been scheduled requires a VAC area, then the operating system would scan all the 5 VAC areas to find which one is available, it would skip this process if the scheduling request specified “NOVAC”. If there is no VAC available upon request, that trigger’s alarm 1201, if no core sets were available, that would trigger alarm 1202. So, if a process requested a VAC area and the OS couldn’t find one, ALARM 1201 would be triggered. This alarm is a process and needs to have executable memory to get executed. This might have filled the core sets up and triggered alarm 1202.

1. **Root Cause of the Overload:** Erasable Memory locations were multiplexed for 2 jobs, such that no 2 jobs have the same memory locations at the same time. The designers managed to provide 7 core-sets (with 12 erasable memory locations) for each job and were able to provide a VAC unit in case of additional temporary storage was required. Each VAC having 44 erasable words. But upon landing repeated jobs to process radar data were scheduled due to misconfiguration of radar switches. This caused its core to fill up and cause an overflow causing ALARM 1202 to be triggered. This was followed by ALARM 1201 since the job that caused overflow requested the VAC area.

**Violation of Rate Monotonic theory:** As stated in the Apollo 11 Lunar Surface Journal, the radar switches were misconfigured. Hence some radar data that was not that much required for the landing of Apollo 11 was running repeatedly. This executable data picked up the RAM space (which was already short for such a mission). Hence causing the core sets to fill up. So, during landing, for processes that was related to landing couldn’t get executed because of low RAM space. While landing the requests that is related to safe landing would be requested multiple times and given priority. But cannot be executed because couldn’t accommodate RAM. Which is clear violation of rate monotonic theory because low priority tasks were given executing priority. This was thankfully resolved because on rebooting the system the main tasks were again given priority because the random radar tasks do not run-on reboot. The system was designed like that.

1. **Reading Liu and Layland’s Paper:**
2. **Least upper Bound as a function of Services**

Theorem 5 in layland’s paper mentions that m tasks with fixed priority order, the least upper bound to processor utilization is U = m(21/m – 1)

In fixed priority scheduling mechanisms such as rate monotonic scheduling the maximum possible utilization is **lim(n->infinity) = m(21/m – 1) = ln(2) => 0.6931 which is almost 70%.**

Hence Least upper bound curve for U = m(21/m – 1)

Chart

Description automatically generated

1. **Three Key assumptions Liu and Layland**
2. Hard real time tasks have periodic request. – For pure process control.
3. Each task must be completed before the next request for it occurs.
   1. For this to be true a small buffer should exist for each peripheral function.
4. Run time is constant. (Time taken by processor to execute the task without interruption).
   1. Run time can be interpreted as maximum processing time for a task. This is ok because large main memories exist and overlapping of transfers between main and auxiliary storage and program execution in modern computer systems.

Fixed Priority Derivation

1. For a set of m tasks with fixed priority assignment, the least upper bound to the processor utilization factor is U = 2(21/m-1)

Areas I did not understand are:

1. How did the idea to assume that request ratio of m tasks to be less than 2 and start derivation and how did it end up proving good for all different request ratios?
2. From where did equation relating Capacity and period arrive?

i.e., where did they derive C1 <= T2 – T1(Lower integer[T2/T1])

and C2 <= T1 – C2(higher integer[T2/T1])

how does that make U = 1 + C1[(1/T1) – (1/T2) (higher integer[T2/T1])

1. How in a fixed priority task C1 = T2 – T1. i.e., How capacity of a task is dependent on the period of some other task or the next immediate service request?

Some assumption really feels like out of the blue I couldn’t follow the steps to make sure I’m following the derivations correctly.

1. **How Rate monotonic theory would have solved this issue?**

Using RMS in Apollo 11, the tasks required for landing can be given a better priority compared to other unwanted tasks during landing since landing requests will occur more frequently. Making it possible to occupy the erasable memory during its execution. What happened might be a deadline-based scheduling which did not focus on the current event making the response time slow for high priority events. As a result Alarm 1201 would not have triggered which in turn would have prevented the 1202 Alarm from triggering as well.

3) [20points] Download RT-Clock and build it on an R-Pi3b+ or newer 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) Which clock is best to use? CLOCK\_REALTIME, CLOCK\_MONOTONIC or CLOCK\_MONOTONIC\_RAW? Please choose one and update code and improve the commenting.

c) 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?

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

**SOLUTION**

**Given Details:**

Details that provided so far are,

1. A C program with Real-Time Clock
2. A make file that can create an object file and can be accessed from a linux terminal.

**Execution Procedure:**

Once the Raspbian OS is installed and ready for use:

1. Open the terminal and navigate to the location of the file.
2. Type “make”, to create object file.
3. Type in sudo ./<filename> and see the program run.
4. **Summary of the program**

Print\_scheduler function finds the type of scheduler

1. Initially we determine the initially implemented policy and found to be SCHED\_OTHER,

**SCHED\_OTHER** is a non-real time policy that fairly slices the CPU time based on the “nice” policy. This policy states that the less CPU you require, the nicer you are, and hence you get a higher priority.

**SCHED\_FIFO** this is a static priority real-time scheduling that states that, a running thread that has been preempted by another thread of higher priority will stay at the head of the list for its priority and will resume execution as soon as all threads of higher priority are blocked again.

**SCHED\_RR** this type of scheduling is a round robin scheduling policy is an enhancement of SCHED\_FIFO except that each thread is allowed to run only for a maximum time quantum.

**Clock\_gettime ():**

This is a posix linux function to get the current time of a clock.

Syntax of the clock\_gettime () – int clock\_gettime (clockid\_t clock\_id, struct timespec \*tp).

The return type of this function is an integer, returns 0 if successfully obtained the time.

Returns -1 in case if an error is occurred due to various might be due to a fault occurred trying to access the buffers provided, or an invalid clock ID, or the process doesn’t exist.

The above function is used in the program to print the time of the entire execution.

How to calculate the time taken for the response:

First declare 2 data structures rtclk\_start\_time and rtclk\_stop\_time of timespec type and set it to {0,0}

Call the clock\_gettime (CLOCK\_ID, rtclk\_start\_time) to the point where you want start calculating the time.

Call the clock\_gettime (CLOCK\_ID, rtclk\_stop\_time) to the point where you want to stop calculation the time.

Print the time.

Pthread\_attr\_init() – Used to initialize the attributes so that you can use those attributes to create threads.

Pthread\_create() – is used to create threads.

Pthread\_join() – You can join one or more threads with this function together.

Pthread\_destory() – destroys the thread once the process is completed.

**DELAY\_TEST** This function calls the clock\_getres function – This finds the resolution of real-time clock.

In the system, the start and stop time is measure by RT clock. The delta\_t() function to calculate the time elapsed .

1. There are 3 clocks used in this program.

Total time taken to execute the entire code is ran in the raspberry PI terminal and figured out that CLOCK\_MONOTONIC clock provides the least of the time duration.

Total time executed by these three clocks in ascending order.

1. Time taken by CLOCK\_MONOTONIC clock
2. Time taken by CLOCK\_REALTIME clock
3. Time taken by CLOCK\_MONOTONIC\_RAW clock.

**CLOCK\_MONOTONIC CLOCK**

**Graphical user interface, text, application

Description automatically generated**

**CLOCK\_REALTIME clock**

**Text

Description automatically generated**

**CLOCK MONOTONIC RAW**

**Graphical user interface, text, application

Description automatically generated**

1. **i. Low Interrupt Handler Latency:**

In total there are 4 basic latencies that aggregate to the system response time.

1. IO latency
2. Context Switch Latency
3. Execution Time
4. Potential Interference from other higher priority tasks

Interrupt latency is the time taken for the execution to handle the interrupt. i.e., it is the time taken from triggering an interrupt till the interrupt is serviced. A low interrupt handler latency indicates a lower response time for a particular process. If interrupts are disabled, that would stop the low priority events from pre-empting the higher priority events making the entire system a real-time system. A RTOS must ensure that a task should be completed within a stipulated time to meet real-time deadlines, an Interrupt latency is a key criterion to determine whether the system is feasible and safe on a real-time perspective.

**ii. Low Context Switch Time:**

Consider a system which is designed to manage 2 tasks concurrently. When you try to schedule the 2 processes. Obviously 1 task will have a higher priority over the other. Which takes a higher priority depends on the type of scheduling mechanism you plan to implement. Consider that 1 process is 50% done and second process having a higher priority is pre-empting the existing process. Then the system saves the existing state of the current process before moving to the new process. This is **Context Switching Time** During scheduling multiple processes in a CPU wherein all tasks must be real time. The tasks are multiplexed in the CPU and context switching time must be very less for the system to not miss deadlines.

Generally, a typical system will aim at maximum CPU utilization. Consider the below sample figure for example.

Calendar

Description automatically generated

Here for 3 tasks with such scheduling capabilities occupies utilization of 100%. The context switching could be negligible enough so that the CPU doesn’t get overloaded. High context switching time removes the above system from meeting deadlines.

**iii. Stable timer’s services where interval timer interrupts timeouts, and knowledge of relative time has low jitter and drift.**

For continuous media applications like live media streaming. Frame jitters are potential cause to real time threats. In case a decoded frame is not received within its real time frame. Then that is a clear case of a soft-real time deadline miss. For these kinds of applications, despite having IO latencies and CPU execution speed on par. It is highly important to have **jitter time** less enough to be a meaningful real-time system.

**Drift Time:** This is another potential real time parameter for continuous media applications. Both ends of streaming connection should stay in sync with the master clock. In case of asynchronous communications, both ends might drift apart in time causing technical glitches to occur. Hence for a device having low drift time, the clock rates of sender and receiver are almost equal. E.g., audio streaming.

**d. RT\_CLOCK accuracy:** The below is the screenshot of the outputs of RT\_CLOCK. The timing accuracy of this clock is less than that of a CLOCK\_MONOTIC accuracy. The execution time of this clock is

10604us. Whereas for a CLOCK\_MONOTONIC clock, uses rate monotonic scheduling of multi-processes completes the same activity in 10448us. RT clock model is the default scheduling algorithm of the CPU and default scheduling mechanisms implemented. Since there is a minor delay with the RT\_CLOCK compared to the CLOCK\_MONOTONIC clock. There is exists a possibility of scheduling the device with that clock being unsafe and likely to not meet deadlines some point down the line. Hence, I believe that scheduling with RT\_CLOCK is **inaccurate.**

Text

Description automatically generated

**REFERENCES:**

1. <https://www.qnx.com/developers/docs/7.1/index.html#com.qnx.doc.neutrino.lib_ref/topic/c/clock_gettime.html>
2. Sam Siewart and John Pratt’s book on Real Time Embedded Components and Systems with Linux. – Chapter 1 and 2.
3. <https://man7.org/linux/man-pages/man2/sched_setscheduler.2.html>
4. <https://man7.org/linux/man-pages/man3/pthread_attr_init.3.html>

4)[45points] This is a challenging problem that requires you to learn quite a bit about Pthreads in Linux and to implement a schedule that is predictable.

a) Download, build and run code in Linux/simplethread/and describe how it works and what it does and compare it to Linux/simplethread-affinity.

b) Download and run the examples for creation of 2 threads provided by incdecthread, as well as Linux/simplethread-affinity-fifo. Describe POSIX API functions used by reading of 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)Download this SCHED\_DEADLINE code and modify the code so that it can use SCHED\_DEADLINE or SCHED\_FIFO and add a POSIX clock time print out. Run the code with both policies and note any differences you see.

d)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 mesec, T2=50 msec, D2=T2 as is diagrammed below in Figure 2 and shown with the Figure 1 VxWorks trace. You may want to review example code for help (

sequencer, sequencer\_generic) and look at a more complete example using an interval timer including this contributed Linux code from one of our student assistants (Example Analysis and Code for different polices – SCHED\_RR, SCHED\_OTHER, SCHED\_FIFO). 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 below was based on this original VxWorks code and your code should match this schedule and timing as best you can. e) 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.

Figure 1: VxWorks Trace of 2 Services f10 and f20 using 90% of CPU Utility

The observed timing above fits our theory for RM policy on a priority preemptive scheduling system as shown by the timing diagram below (1 period is 10 msec).

Figure 2: Ideal RM Trace of 2 Services S1=f10 and S2=f20 using 90% of CPU Utility and S3=slack stealer using last 10% of CPU available (ignore slack stealer for exercise)

**Solution:**

**Please find the program on Q-4 Part1**

**4a Code Explanation:**

This system is a best effort system rather than a real-time system.

The Linux/SimpleThread program produces functionally correct results. i.e., the summation is correct for all the thread values given in the program. But the order is different than what is expected. So, for whatever threads, the order and sequencing might be different, but the values are right and not a single piece of line is missing. So far that is right. But what we should additionally take care of is the timing correctness and make it deterministic. There is nothing wrong with this code from a functional perspective. It is a fair but has unsynchronized timing requirements.

SCHED\_OTHER is the scheduling in the question\_4ai.

“**SCHED\_OTHER is a non-real time POLICY that GIVES FAIR SHARE TO THE CPU DUE TO IT WORKING WITH NICE PRIORITY”**

Real time systems always require predictable response, this can be achieved by either parallel processing or concurrency.

The 1st Code Linux/SimpleThread code is functionally correct code but not correctness in terms of timing correctness.

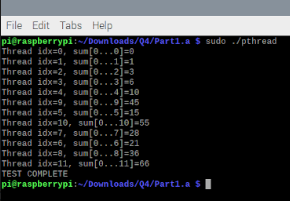
“**SCHED\_FIFO POLICY is a real-time scheduling policy that GIVES THE FIRST THREAD TO EXECUTE AND GIVEN A HIGHER PRIORITY HENCE THE SYSTEM WILL EXECUTE IN THE ORDER REQUIRED. FOR THIS SET OF PROBLEM SCHED\_FIFO IS THE BEST IF TIMING CORRECTNESS IS ALSO AS IMPORTANT AS FUNCTIONAL CORRECTNESS”**

The 2nd Code Linux/SimpleThread-affinity code involves timing correctness including functional correctness.

Threading is done carefully and SCHED\_FIFO scheduling is implemented.

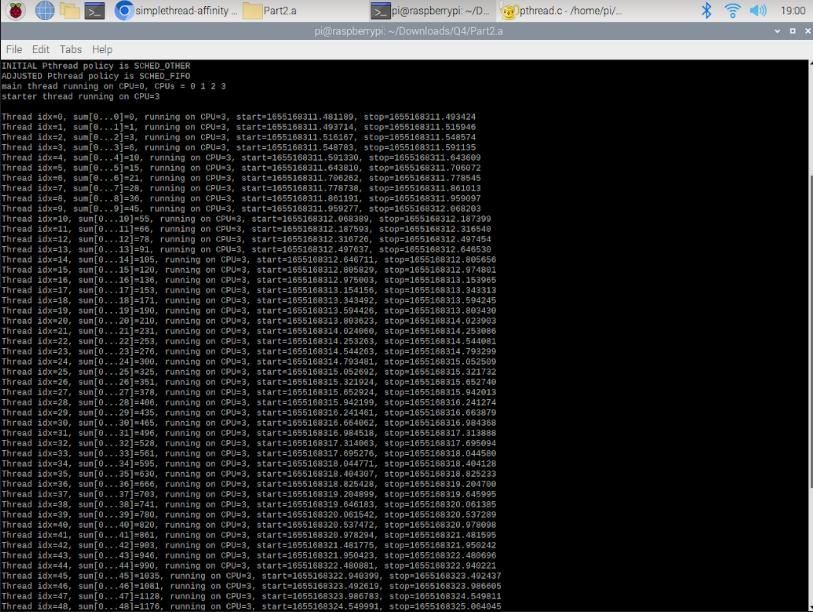
The below is the screenshot of output Linux/SimpleThread code in Raspberry PI 4 OS.

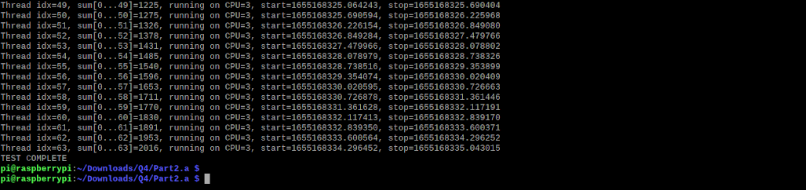
Clearly the number sequencing is not in order. (Algorithmic correctness observed).



The below is the screenshot of output Linux/SimpleThreadAffinity code in Raspberry PI 4 OS

Now, we can see that this code can both algorithmic and timing correctness due to a result of scheduling algorithms properly implemented.





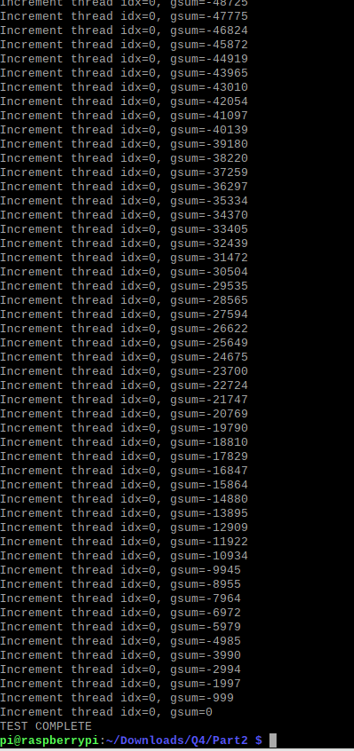
**4b Pthread\_Create:** Pointer to thread descriptor.

This is a POSIX function used to create the thread

**Pthread Join:** Join all the possible threads so that all the processes runs in only one seamless process.

**Pthread\_attr\_init** Initialize the posiX attitude.

**Semaphore:** When more than one threads get executed. It is important to synchronize all threads so that the CPU can handle the process seamlessly. At this point it is more important to us to understand



The default code was SCHED\_OTHER scheduling policy which declares CPU times based on “nice” priority. That is not a real time scheduling policy. Hence you might find the priorities assigned in improper order. Because that doesn’t have any priority as such.

To make this code functionally correct as well. We can do 2 things

1. You can try to schedule the program with a real-time scheduling policy for e.g., SCHED\_FIFO, this scheduling is based on a priority-based scheduling. Once priorities are assigned the first assigned priority will be executed first and the issue might be resolved.
2. There is some other concept called the **semaphore** which is used to sync 2 or more threads so that processing happens synchronously. It is an unsigned int which changes the integer value to atomic.
   1. If one thread process increments the integer and other thread process decrements the integer those operations cannot decrement each other.
   2. We can only interact with semaphore using 2 operations
   3. Wait () and post()
   4. Post is called signal sometimes.

Wait waits until the attribute is a positive value. If it is negative it doesn’t process until it becomes positive making it a little deterministic

Post just increments the ID.

**4c SCHED\_DEADLINE** code prints whatever inside printf statement for every 2 seconds on an infinite loop. i.e., every iteration will take 2 seconds.

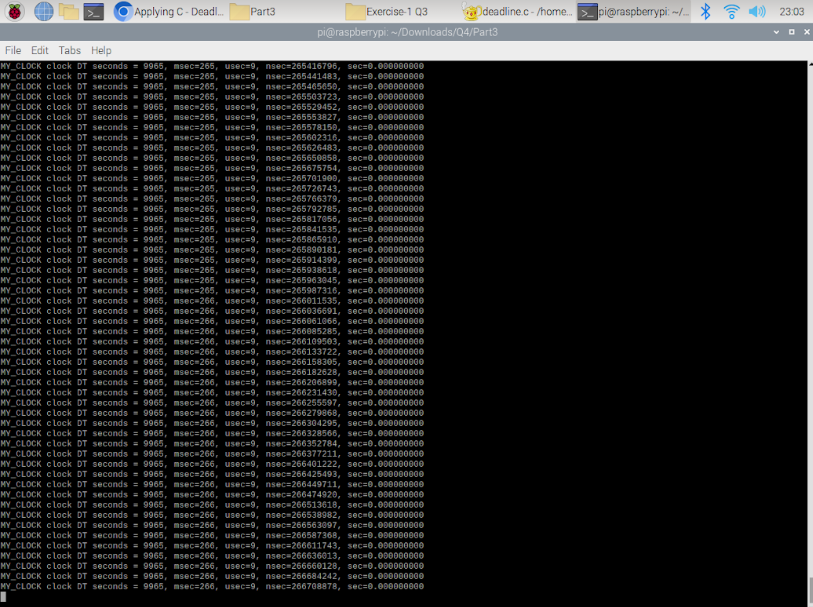
In a SCHED\_FIFO – A higher priority task will always pre-empt any currently running task so, A part of the process would be completed in the first iteration and comes back the second time and continue executing another time. So, the iterations occur at a faster rate. Hence you can see the for loop coming back with multiple iterations per second.

In a SCHED\_DEADLINE – The control doesn’t move to another task until the first one is complete. Hence considering for loop as a process. The CPU comes back once all the process is done which takes 2 seconds.

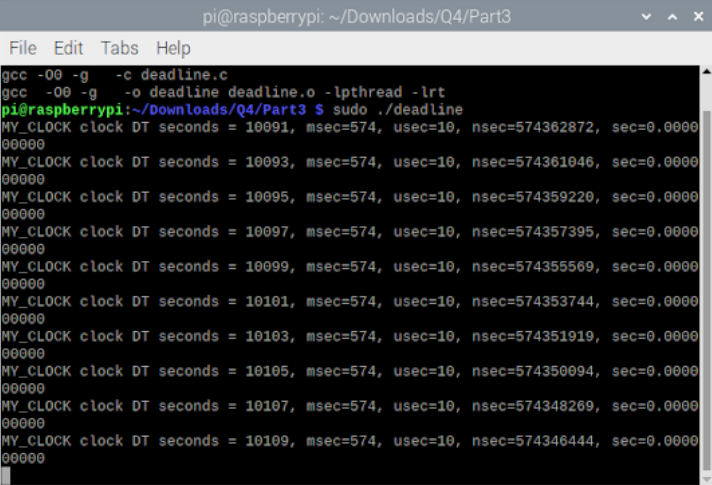
Fixed priority scheduling will aim towards pre-emption of a higher priority task that is fixed initially itself. Whereas in Dynamic scheduling, the scheduler will give priority to task completion rather pre-emption in most cases.

Consider (for (infinite loop) a process. This process being lower priority can pre-empt in a fixed priority scheduling but not so much in dynamic based scheduling in case the functions inside the CPU body has a deadline earlier than this.

Below are the sample screenshots of both SCHED\_FIFO and SCHED\_DEADLINE time stamps.

****

**Output of the timestamp files using SCHED\_FIFO, iterations happen multiple times a second.**

****

**Output of the timestamp files using SCHED\_DEADLINE, iterations happen for every 2 seconds.**

**REFERENCES:**

1. Posix.pdf in canvas - https://www.sciencedirect.com/topics/computer-science/rate-monotonic-analysis#:~:text=The%20critical%20instant%20in%20rate%2Dmonotonic%20scheduling.&text=As%20EQ.,and%20process%202%20is%20feasible.
2. Lecture-1 Video RTES ECN 5623.
3. https://www.sciencedirect.com/topics/computer-science/rate-monotonic-analysis#:~:text=The%20critical%20instant%20in%20rate%2Dmonotonic%20scheduling.&text=As%20EQ.,and%20process%202%20is%20feasible.
4. <https://linux.die.net/man/3/pthread_attr_destroy>
5. <https://man7.org/linux/man-pages/man3/pthread_join.3.html>
6. <https://en.wikipedia.org/wiki/Interrupt_latency#:~:text=Many%20computer%20systems%20require%20low,pass%20between%20executions%20of%20subroutines>.
7. <https://www.ibm.com/docs/en/zos/2.2.0?topic=functions-pthread-create-create-thread>
8. <https://man7.org/linux/man-pages/man3/pthread_attr_setschedparam.3.html>
9. <https://man7.org/linux/man-pages/man3/pthread_attr_setinheritsched.3.html>
10. <https://www.hq.nasa.gov/alsj/a11/a11.1201-pa.html>