Exercise 1- Invariant LCM schedules

1) **The Rate Monotonic Policy states that services which share a CPU core should multiplex it (with context switches that preempt and dispatch tasks) based on priority, where highest priority is assigned to the most frequently requested service and lowest priority is assigned to the least frequently requested AND total shared CPU core utilization must preserve some margin (not be fully utilized or overloaded).**

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.**

Rate Monotonic (RM) policy is a priority assignment algorithm used in real-time systems for scheduling tasks. It is based on the principle that tasks with shorter periods have higher priorities. The RM policy follows a pre-emptive scheduling approach, where tasks with higher priorities can interrupt lower priority tasks.

The following is the analysis of priorities of the 3 tasks:

Service	Period		Freq f	**Priority**	Run-time	
S1	T1	3	0.33333	**1**	C1	1
S2	T2	5	0.2	**2**	C2	2
S3	T3	15	0.06667	**3**	C3	3

The feasibility of the services will be scheduled in the period equal to the least common multiple of T1, T2, and T3. In this case, it is 15ms. Thus, all the services will follow the rate monotonic policy. The following timing diagram is designed.

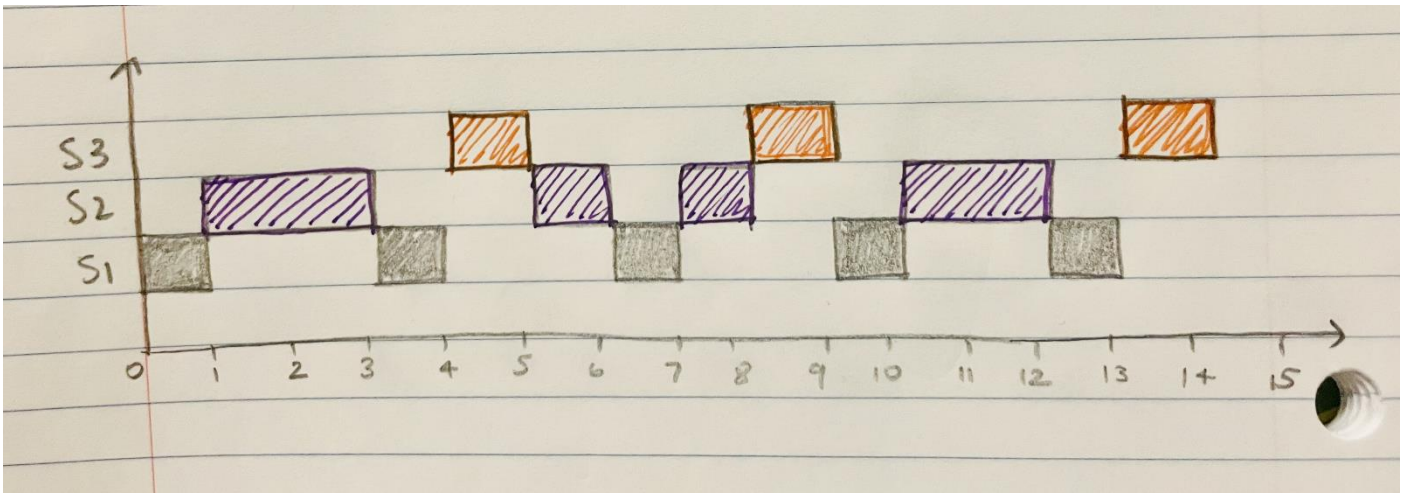


Figure 1: Timing diagram of the services-S1, S2, S3 following the rate monotonic policy

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). What is the total CPU utilization by the three services?**

Feasibility: The services are feasible as shown in the timing diagram over a time period of 15ms. Since the schedule is feasible over the LCM ie. 15ms, the tasks are feasible for all time ie. Mathematically repeatable as an invariant indefinitely.

CPU utilisation: The CPU utilisation can be calculated can be calculated with the following formula, $U = \Sigma (Ci/Ti)$. It is calculated for each task as shown in the table below:

Utility no.	CPU utilisation
U1	0.33
U2	0.40
U3	0.13

The total CPU utilisation (%) is the sum of all tasks CPU utilisation, which is equal to 93.33%.

However, the scheduling is only safe (unlikely to ever miss a deadline) if the CPU utilisation is less than or equal to the Least Upper bound utilization. The LUB utilisation can be calculated with this formula:

$$U = \sum_{i=1}^{m} (Ci / Ti) \leq m(2^{\frac{1}{m}} - 1)$$

where m= the number of tasks. With this, the LUB is calculated to be 77.97%. Since the calculated CPU utilisation (93.3333%) is greater than the LUB utilisation, the schedule is un-safe.

2) **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.**

The LEM guidance computer had significant constraints in terms of memory and processing power. It had a limited amount of fixed memory – 36864 15-bit words (ROM) and erasable memory- 2048 words (RAM). The fixed memory stored executable code and constants, while the erasable memory was used for variable data and counters. Due to the limited erasable memory available, the same memory address had to be used for different purposes at different times. This required thorough testing to ensure that memory conflicts did not occur.

The LGC supported interrupt-driven, time-dependent tasks as well as priority-ordered jobs. Each job was allocated a "core set" of 12 erasable memory locations for its execution. Additional temporary storage could be requested through vector accumulators (VAC), which had 44 erasable words. The operating system managed the allocation of resources and scheduled jobs accordingly. If there were no available VAC areas or core sets, the system would trigger an alarm (1201 or 1202) and abort the operation.

a) **What was the root cause of the overload and why did it violate Rate Monotonic policy?**

During the Apollo 11 mission, the 1201/1202 alarms were triggered due to a misconfiguration of the rendezvous radar switches. As the Lunar Module (LM) descended towards the lunar surface, the misconfigured rendezvous radar started sending extraneous data to the guidance computer causing an overload of data. Initially, the core sets got filled up causing a 1202 alarm. The 1201 alarm that came later was because the scheduling request that caused the actual overflow was one that had requested a VAC area.

When the alarms are triggered, the computer would reboot and reinitialise from the start. It was made to recognise the priority of the secondary data and after restart, it would continue the important tasks only from a point near where they were before the reboot. The restart capability had been extensively tested by MIT, and the NASA personnel in the Mission Operations Control Room (MOCR) knew that the mission could proceed despite the alarms, as critical functions would resume correctly.

This scheduling **violates** the rate monotonic policy as the more frequent task would ideally have the highest priority. In the Apollo 11 mission, the receiving of extraneous data was ignored and priority was given to more important tasks.

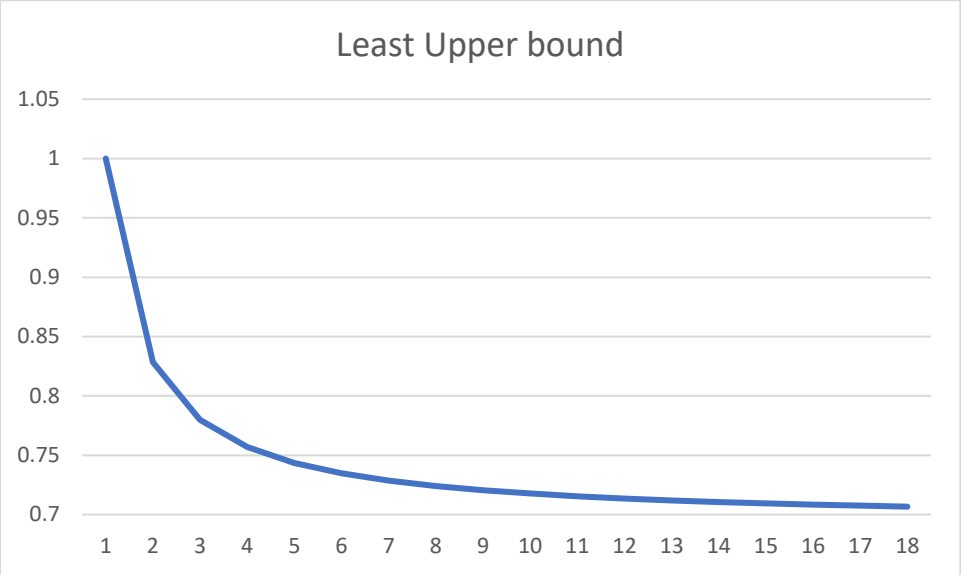
b) Now, read Liu and Layland's paper which describes Rate Monotonic policy and the Least Upper Bound – they derive an equation which advises margin of approximately 30% of the total CPU as the number of services sharing a single CPU core increases.

The paper analyses the performance of an optimum fixed priority scheduler and discover that it has a maximum processor utilization limit, which can be as low as 70 percent when dealing with large sets of tasks.

Furthermore, the paper introduces a dynamic priority assignment approach that achieves full processor utilization by assigning priorities based on the current deadlines of the tasks. This technique dynamically adjusts the priorities to optimize the processor's efficiency. Liu and Layland also discuss the potential benefits of combining both the fixed priority scheduler and the dynamic priority assignment method.

The paper concludes with the limitations of an optimum fixed priority scheduler in terms of processor utilization and proposes a dynamic priority assignment strategy as a solution to achieve full utilization. The combined approach of fixed and dynamic priority scheduling is also considered as a possible optimization technique.

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.**

Assumptions made by Liu and Layland in the paper – 'Scheduling Algorithms for multiprogramming in a hard-real-time environment':

1. The requests for all tasks for which hard deadlines exist are periodic, with a constant interval between requests. This assumption states that tasks must be periodically recurring and that they should have a fixed interval between each request.
2. Deadlines consist of run-ability constraints only--i.e., each task must be completed before the next request for it occurs. This assumption does not take into account dependencies between tasks or external dependencies and simplifies the scheduling process.
3. Runtime for each task is constant for that task and does not vary with time. Runtime refers to the time taken by a processor to execute the task without interruption. This assumption assumes that once the task is determined, the runtime for it remains the same throughout its execution. This enables deterministic behaviour as the execution time is unaffected by external factors.

Aspects of the fixed priority LUB derivation that I did not understand:

1. In theorem 3, for the derivation of the LUB for the processor utilization factor for a set of two tasks with fixed priority assignment, there are two cases defined. I was unable to comprehend the requirement for different cases. The run time of the second task (C_2) is defined differently for both the cases. In the derivation, the processor utilization factor in case 1 is defined to be greater than 1, which is another confusing aspect.

2. In theorem 4, the run time of task 2 is proven to be the difference between the two periods of the highest priority tasks for a set of tasks that fully utilise the processor. However, during the proof, C_1',.., C_m' and the set C_2'',…,C_m'' are also defined for an unstated reason. I did not understand how this formula came to be: $C_m = T_m - 2(C_1 + C_2 + \ldots + C_{m-1})$.
3. The first derivative of the processor utilisation is taken with respect to each of the g ratio equal to zero and the resulting difference equations are solved. The differential calculus part is also another aspect I did not understand.

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

The Apollo 11 1201 and1202 alarms were caused to the overload of incoming data coming from the rendezvous radar. This placed a resource constraint on the LEM guidance computer.

RM analysis alone would not have prevented the 1201/1202 errors in the Apollo 11 mission for the following reasons:

1. The policy requires the tasks to be known before implementation with fixed and known execution and request time. In the case of the Apollo 11 mission, the overload was caused due to an unexpected condition. RM policy does not account for unforeseen events and their impacts. Further, if The RM policy had been adapted, the highly frequent tasks would have received higher priority. This means that the erroneous tasks would have been given higher priority and, in that case, the more important tasks would have been ignored and this would have led to system failure.
2. Dynamic Resource Management: The errors introduced a priority inversion issue. RM analysis does not consider dynamic resource management and handle priority inversion problem caused by shared resources.

3) 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).

The code demonstrated the usage of POSIX real-time clock for accurate delay operations and checks its accuracy. This is done with the usage of POSIX defined threads-pthreads. The following functions are used in the RT-Clock code:

1. Void print_scheduler(void): The function gets the scheduling policy used by a particular process with the POSIX standard function, sched_getscheduler. The following scheduling policies are mentioned:

- SCHED_OTHER is the default scheduling policy for non-real-time processes. It uses a time-sharing algorithm and allows the system to dynamically allocate CPU time among eligible processes.
- SCHED_FIFO: First-In, First-Out scheduling policy is used for real-time processes. Processes scheduled with this will run until they yield the CPU or are preempted by a higher-priority process.
- SCHED_RR: Round-Robin scheduling policy is used for real-time processes. The processes are given a time slice (quantum) to execute before being preempted and placed at the end of the scheduling queue.

2. Double d_ftime: The function calculates the time difference between two timespec structures in double format.
3. Int delta_t: The function calculates the time difference between two timespec structures, taking into account any potential roll-over.
4. Void *delay_test(void *threadID): The delay_test function is the main thread function. It runs a 100 delay tests. The delay is performed with the function nanosleep which has nanosecond resolution. When the sleep is interrupted by a signal or another process, the function returns (-1) and the remaining time is stored in a timespec structure. Before nanosleep is called, the start time stamp is received by the function clock_gettime. For all time-related operations, there are some clock options like – CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_MONOTONIC_RAW. After the requested delay is successful, the stop time stamp is recorded again. The difference of start and stop time stamps are calculated. Further the difference between the requested sleep period and the previous difference is calculated to get the error in delay caused by the nanosleep function.

5. Void end_delay_test(void): The function prints the results of the sleep operation: the actual sleep time and delay error.
6. Main function: In the main process, the pthread attributed objects are initialised with default values. These attributes usually include the scheduling policy, priority, stack size, etc. These attributes are modified using the functions pthread_attr_setinheritsched and pthread_attr_setschedpolicy. The former one sets the thread to PTHREAD_EXPLICIT_SCHED which indicates that the thread should only use scheduling attributes set in the attribute object, rather than inheriting them from the creating thread. The scheduling policy is set to SCHED_FIFO in which the threads are executed in a strict priority order. The maximum and minimum priority of this policy is received through the functions: sched_get_priority_max and sched_get_priority_min. This range is usually 0-99 but it can vary depending on the operating systems or the underlying scheduler. The scheduling policy and priority of the current process (obtained using 'getpid()') ie. the main thread is set to SCHED_FIFO and maximum priority. The scheduling parameters stored in 'main_param' are assigned to the thread attribute object 'main_sched_attr' using the function pthread_attr_setschedparam so that a new thread created with 'main_sched_attr' inherits the scheduling parameters. A new thread is created with the function pthread_create based on the 'main_sched_attr' attributes and the entry point function is assigned as 'delay_test'. Thus, there will be two concurrent threads: main thread and newly created thread running the 'delay_test' function.

How to use it to time code execution: The code start time stamp can be obtained by the function, clock_gettime(CLOCK_MONOTONIC, &clk_start_time). This function will store the start time in the timespec structure- clk_start_time. In the main thread, a thread needs to be created for running the function for which the code execution time needs to be measured. Once the code execution is over, the stop time stamp can be obtained with the same function. The difference in these two time stamps which are stored in the timespec structures will be the code execution time. Ideally, logging this difference with syslog is recommended over printing the values on the terminal.

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.

The CLOCK_REALTIME represents the represents the system's wall-clock time and can be adjusted by system time changes due to NTP (network time protocol) synchronisation. It provides the time since the epoch (typically January 1, 1970). It is ideally used for application where time in relation to real-world time is required, such as timestamping network events or logging.

CLOCK_MONOTONIC represents monotonic time, which continuously increases at a uniform rate. It is not subject to adjustments and is not affected by system time changes. It provides a reliable measure of elapsed time between two points.

CLOCK_MONOTONIC_RAW: This clock is similar to CLOCK_MONOTONIC but provides access to a raw hardware-based clock source. It does not undergo adjustments and does not include time spent in sleep states.

For our application, to time the code execution, CLOCK_MONOTONIC_RAW is the most ideal as it is not dependent on the wall-clock time changing or is not depended on NTP synchronisation. In the figures shown below, the code is executed with the clock set as both CLOCK_MONOTONIC_RAW and CLOCK_MONOTONIC. In this case, there is no particular difference.

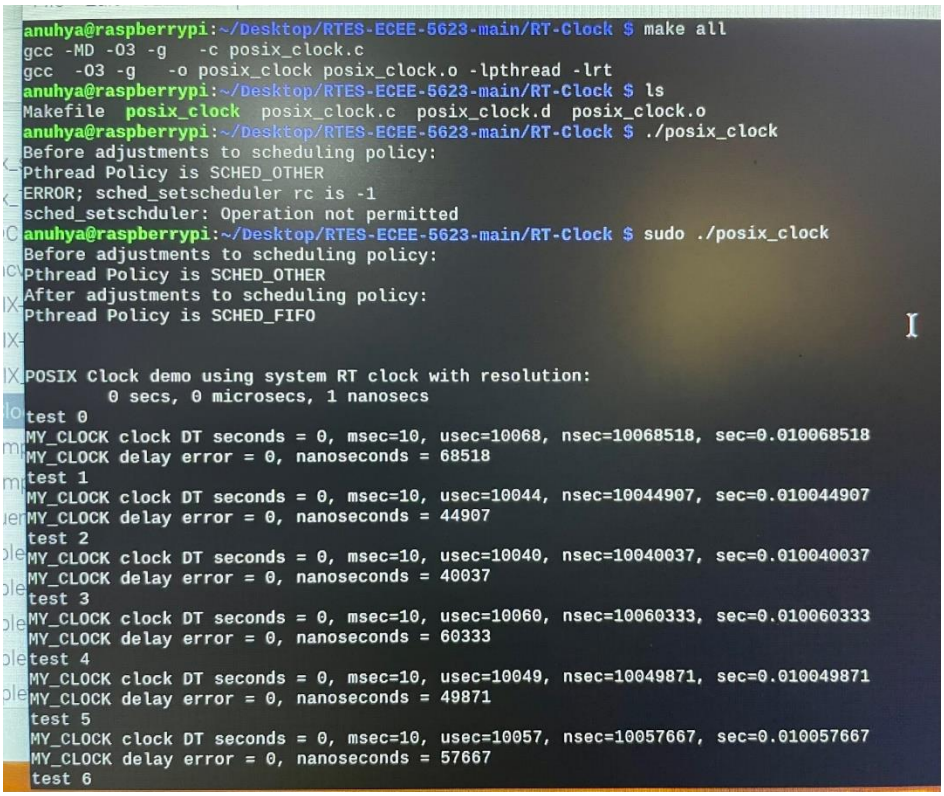


Figure 2: RT-clock code execution with the clock set as CLOCK_MONOTONIC_RAW

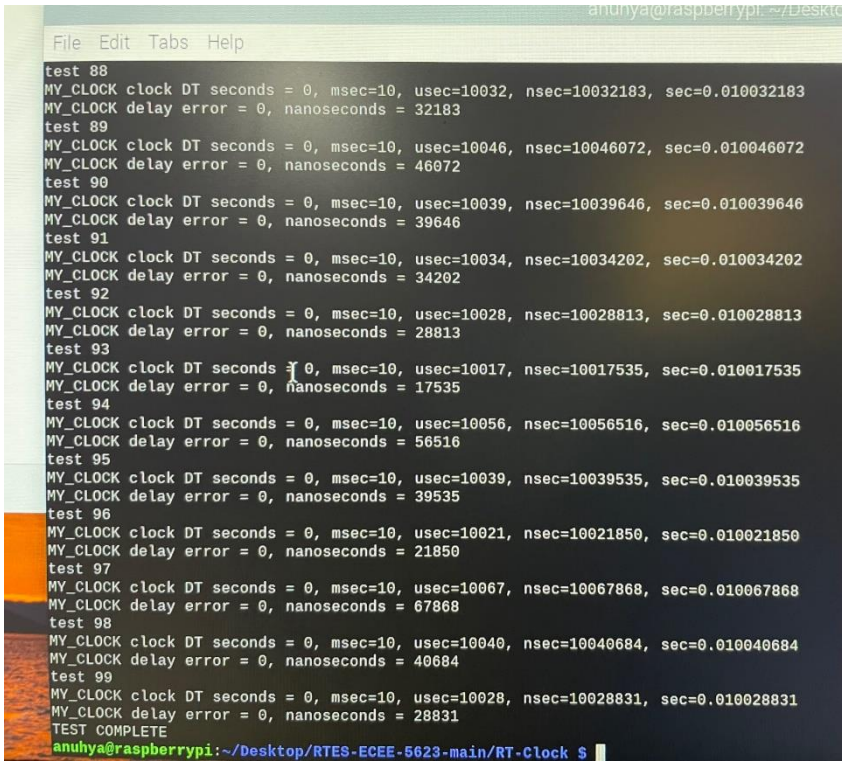


Figure 3: RT-CLOCK code executed with clock set as CLOCK_MONOTONIC

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?

Each of the aspects stated above are important for the following reasons:

Low Interrupt Handler Latency: Interrupts are crucial in real-time systems as they are used to handle time-critical events and hardware interactions. Low interrupt handler latency refers to the amount of time it takes for the system to respond to an interrupt and start executing the associated interrupt handler. Minimizing interrupt handler latency is

crucial because it ensures that time-critical tasks can be handled promptly, reducing the chances of missing important events or deadlines.

Low Context Switch Time: Context switching is the process of saving the current execution context of a task and restoring the saved context of another task. The kernel stores the status of the CPU's register and program counter during context switching. Efficient context switching is important for real-time systems as it allows tasks with strict timing requirements to be scheduled quickly and efficiently, minimizing delays and maximizing responsiveness.

Stable Timer Services with Low Jitter and Drift: Timers are most commonly used in real-time systems for purposes like scheduling tasks, measuring time intervals, and triggering events. Thus, stable timer services with provided an accurate measurement is crucial for such applications. Low jitter means that the timer events occur with minimal variation in their timing. Low drift means that the timers maintain accurate timekeeping over long periods without significant deviations. Having stable timer services with low jitter and drift is essential for real-time systems to ensure precise timing, meet deadlines, and maintain synchronization with external events.

In summary, low interrupt handler latency, low context switch time, and stable timer services with low jitter and drift are crucial in real-time systems to achieve predictable and reliable behaviour, and ensure accurate and consistent timing operations.

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

The accuracy of the RT-Clock code is not sufficient if the system requires nanoseconds precision or even for 0.001ms precision. The figures shown above indicate, there is an indeterminate delay in every test case. This value changes unpredictably and is subject to the clock type used as well. Thus, it is unreliable and cannot be used for real-time applications where high precision timing measurements are required.

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

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.

The **Linux/simplethread/** code creates the specified number of threads using the POSIX standard defined pthreads. Each of them has the same entry function- counterThread with their own thread index passed as a parameter. The function calculates the sum of numbers from 1 till the passed thread index and prints the result. The main thread will wait for all threads to finish before exiting. This is made sure with the function pthread_join. Each of the thread can be seen as an implementation of service but this code shows a non-deterministic behaviour as the real time nature has not been assigned yet. In this case, it follows SMP (Symmetric multiprocessing) scheduling where multiple threads can run concurrently on different processors. The thread will be assigned to the processor core that is the least busy. The scheduler determines which thread gets executed on which processor based on factors such as thread priorities, processor availability, and implements load balancing. The code execution is shown in Figure 4 below.

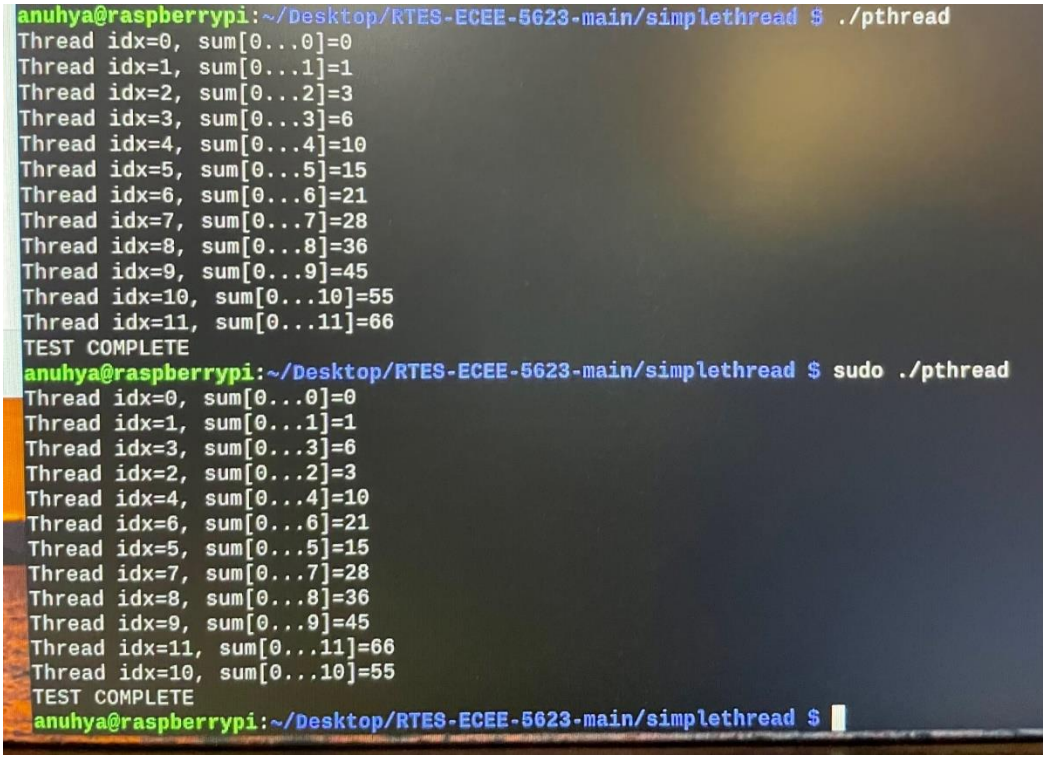


Figure 4: Code execution of simplethread.

The **Linux/simplethread-affinity** code emulates the AMP behaviour. AMP behaviour can be defined as multiple CPU cores with local memory with application specific task workloads running on multiple OS instances. By using the SCHED_FIFO, rate monotonic policy is applied and the services are assigned to run only on one specific core. The CPU affinity is demonstrated in this code. By setting the CPU affinity, the execution of threads and processes can be restricted to run on a specific CPU in a multi-core system. This is useful in situations where cache thrashing or contention between threads must be avoided and where deterministic execution behaviour is required. All the created threads are assigned the scheduling policy - SCHED_FIFO (First-In, First-Out). In this policy, the processes will run until they voluntarily yield the CPU or are preempted by a higher-priority process.

In the simplethread-affinity code the threads are all created with the maximum priority that the SCHED_FIFO policy can allow. Thus, an initially created thread cannot be pre-empted by a thread created later. This explains the deterministic output seen in Figure 5.

Setting the CPU affinity: Initially, the threads are initialised such that the thread attributes are explicitly inherited from the thread attibutes structure- fifo_sched_attr. The CPU is set to use only the 4th core using the CPU_SET macro. The function pthread_attr_setaffinity_np is called to set the threads associated with the thread attribute structure- fifo_sched_attr will be scheduled to run only on the specified CPUs in the previous instruction. The process policy is set first using the sched_setscheduler and then using the pthread_attr_setschedparam() function, the scheduling parameters (scheduling policy and thread priority) are set for the thread attributed object passed as a parameter.

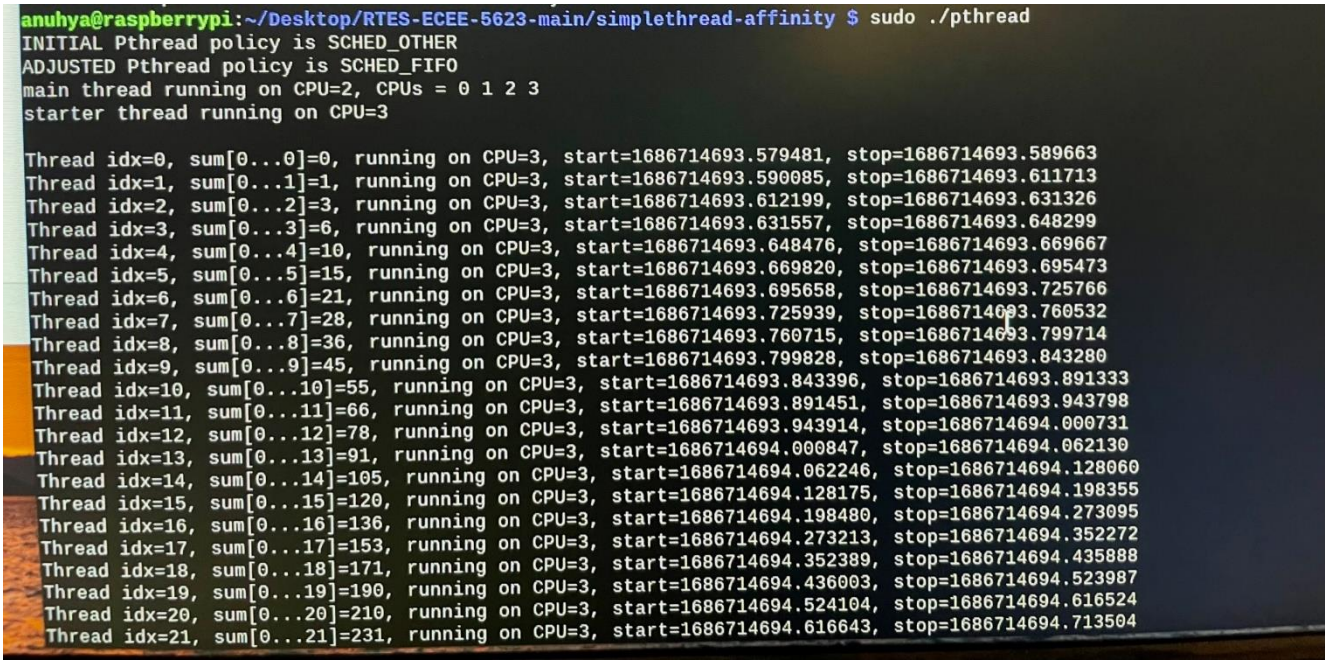


Figure 5: Code execution of simplethread-affinity.

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.

Incdecthread: The incdecthread creates two threads which run on an SMP system so the behaviour is not deterministic. The code creates two threads and performs increment and decrement operations on a global variable gsum. Both the threads are started at the same time without any fixed priority or a scheduling policy. The code demonstrates usage of a shared resource and the protocols that need to be in placed for it. Figure 6 shows the output of the code.

The commenting of the provided code is in the folder 'incdecthread'. While, a more deterministic approach is written using semaphores in the folder 'incdecthread_modified'.

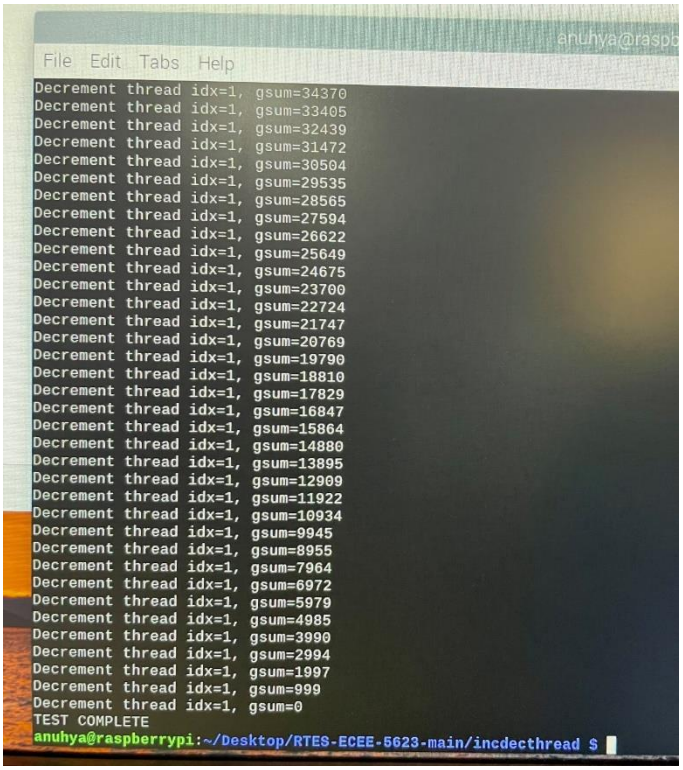


Figure 6.1: Output of incdecthread.

The output of the incdecthread_modified shows that the global variable is incremented and decremented by the respective threads in an order. This is shown in the Figure 6.2.

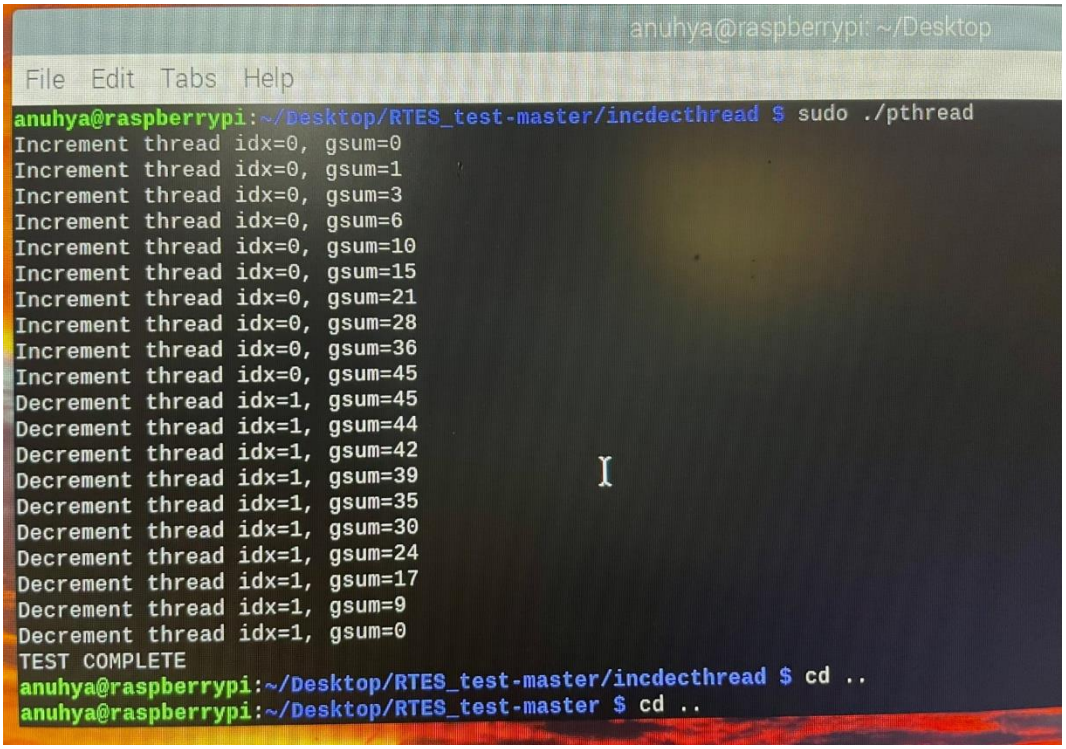


Figure 6.2: Output of incdecthread modified with the use of semaphores.

Simplethread-affinity-fifo:

Simplethread-affinity-fifo is essentially the same as simple-thread-affinity code, which is explained in question 4a).

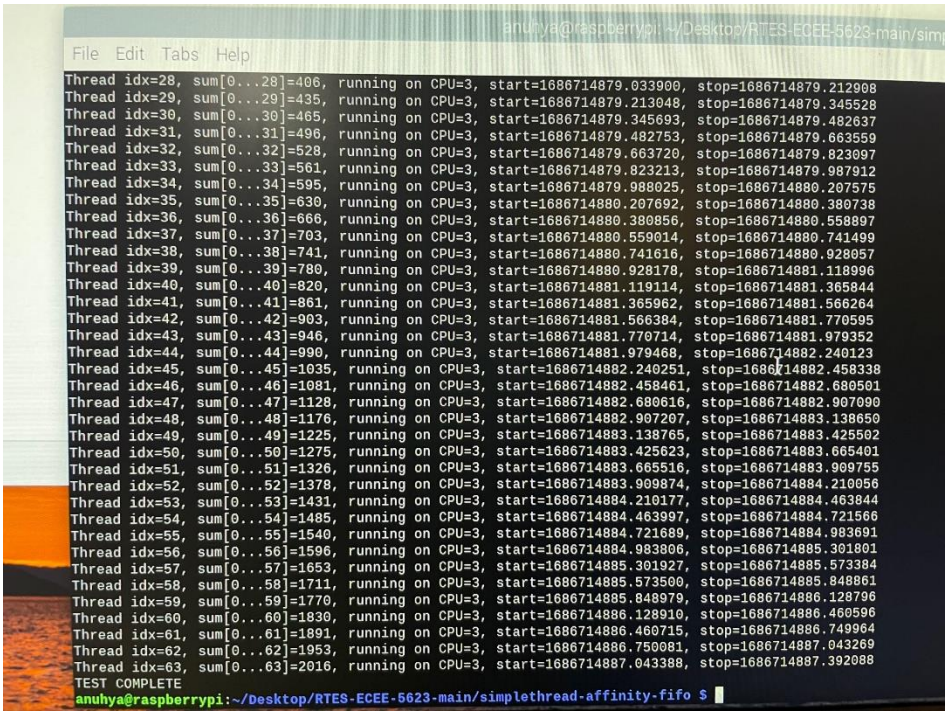


Figure 7: Output of simplethread-affinity-fifo

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.

The SCHED_DEADLINE policy is used for sporadic task model deadline scheduling. It is currently implemented using Earliest Deadline first. A sporadic task is a task that needs to activated at most once per period. Three parameters need to be set for this policy: runtime, deadline and period. The runtime is to more than the average computation time (or worst-case execution time for hard real-time tasks). The kernel requires that:

sched_runtime <= sched_deadline <= sched_period

The threads admitted to the SCHED_DEADLINE policy, they are assigned the highest priority in the system; if any such thread is runnable, it will pre-empt any thread scheduled under of the other policies. In the figure 8, the output if simple-deadline is shown. Since the period is set to 2 seconds, the thread is called every 2 seconds. When the thread calls sched_yield, it will yield the current job and wait for a new period to begin.

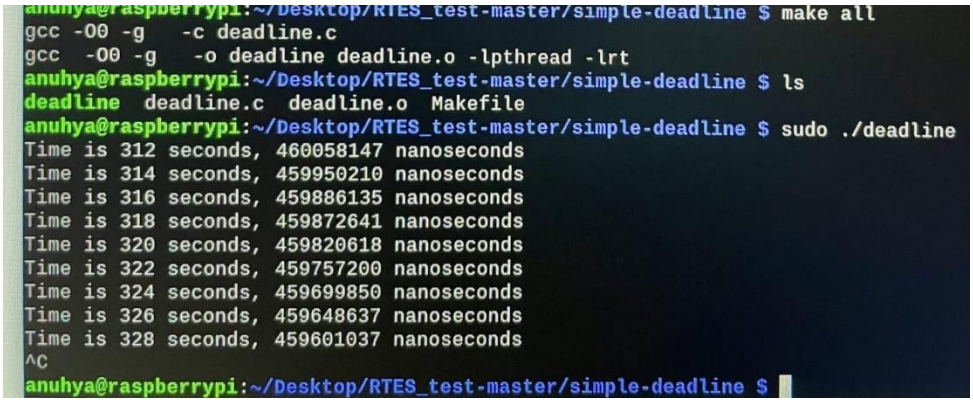


Figure 8: Output of simple-deadline with the SCHED_DEADLINE policy

In the SCHED_FIFO scheduling policy, the scheduling parameters like runtime, deadline and period are not considered. Thus, the task runs repeatedly, in this case, the time is printed out repeatedly. This is shown in the Figure 9.

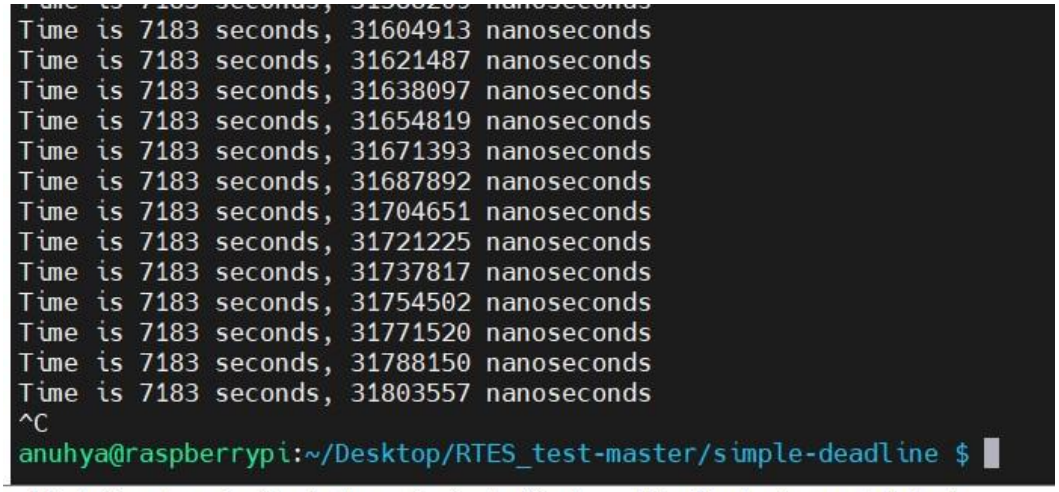


Figure 9: Output of simple-deadline with SCHED_FIFO policy.

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.

A synthetic workload of a task to generate Fibonacci sequence is set up for run times of 10s and 20s in **rmschedule.c**. The run time of the function macro is checked, and by this method the number of cycles to run for the required time period is calculated. Three threads are being run on the same CPU core to generate AMP behaviour. The scheduler thread which will set up the tasks F10, F20 is run with the highest priority for a period of 10ms. The output of the

code, rmschedule is shown in Figures 10 & 11. The code is being run of 3 times of their LCM period (100ms) ie. 300ms. The code can be found in the folder ‘rmschedule’.

```
anuhya@raspberrypi:~/Desktop/RTES_test-master/question_4_printf $ sudo ./rmschedule
System has 4 processors configured and 4 available.
Using CPUS=1 from total available.
rt_max_prio=99
rt_min_prio=1
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE PROCESS
Service threads will run on 1 CPU cores
Start sequencer
Sequencer thread @ msec=2001857.464856000
F10 runtime calibration 0.046777 msec per 500 test cycles, so 213 required
F20 runtime calibration 0.046851 msec per 500 test cycles, so 426 required
sequencer at msec=2001867.524826000
sequencer at msec=2001877.563111000
***F10 start 1 @ 20.139532 on core 3
***F10 complete 1 @ 29.659825, 213 loops
sequencer at msec=2001887.604378000
sequencer at msec=2001897.633700000
***F10 start 2 @ 40.201251 on core 3
***F10 complete 2 @ 50.178945, 213 loops
sequencer at msec=2001907.667430000
***F20 start 1 @ 50.264555 on core 3
sequencer at msec=2001917.722215000
***F10 start 3 @ 60.289970 on core 3
***F10 complete 3 @ 69.982260, 213 loops
sequencer at msec=2001927.755278000
***F20 complete 1 @ 80.017989, 426 loops
sequencer at msec=2001937.808063000
***F10 start 4 @ 80.374559 on core 3
***F10 complete 4 @ 89.897295, 213 loops
sequencer at msec=2001947.855256000
sequencer at msec=2001957.926485000
***F10 start 5 @ 100.550442 on core 3
sequencer at msec=2001968.054084000
***F10 complete 5 @ 111.785229, 213 loops
***F20 start 2 @ 111.833377 on core 3
```

Figure 10: Output of rmschedule.c (part 1)

```
***F20 start 2 @ 111.833377 on core 3
sequencer at msec=2001978.123980000
***F10 start 6 @ 120.716734 on core 3
***F10 complete 6 @ 130.242175, 213 loops
sequencer at msec=2001988.178691000
sequencer at msec=2001998.209494000
***F10 start 7 @ 140.784156 on core 3
***F10 complete 7 @ 150.326560, 213 loops
sequencer at msec=2002008.251761000
***F20 complete 2 @ 150.832127, 426 loops
***F20 start 3 @ 150.851515 on core 3
sequencer at msec=2002018.290102000
***F10 start 8 @ 160.861190 on core 3
***F10 complete 8 @ 170.379149, 213 loops
sequencer at msec=2002028.325887000
***F20 complete 3 @ 180.786818, 426 loops
sequencer at msec=2002038.357654000
***F10 start 9 @ 180.925575 on core 3
***F10 complete 9 @ 190.444423, 213 loops
sequencer at msec=2002048.393606000
sequencer at msec=2002058.423614000
***F10 start 10 @ 200.994702 on core 3
***F10 complete 10 @ 210.497254, 213 loops
***F20 start 4 @ 210.524846 on core 3
sequencer at msec=2002068.460380000
sequencer at msec=2002078.487944000
***F10 start 11 @ 221.055735 on core 3
***F10 complete 11 @ 230.578176, 213 loops
sequencer at msec=2002088.520433000
***F20 complete 4 @ 239.740048, 426 loops
sequencer at msec=2002098.550403000
***F10 start 12 @ 241.117343 on core 3
***F10 complete 12 @ 250.638450, 213 loops
```

Figure 11: Output of rmschedule.c (Part 2)

a) 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.

The table shows the scheduling of the tasks, F10 and F20, and their respective start and competition times. As analysed the from the table, the tasks follow the timing diagram shown in Figure. The code functions as expected and designed.

Task	iteration number	time stamp
F10	start 1	20.139532
F10	complete 1	29.659825
F10	start 2	40.201251
F10	complete 2	50.178945
F20	start 1	50.264555
F10	start 3	60.28997
F10	complete 3	69.98226
F20	complete 1	80.017989
F10	start 4	80.374559
F10	complete 4	89.897295
F10	start 5	100.55044
F10	complete 5	111.78522
F20	start 2	111.83337
F10	start 6	120.71673
F10	complete 6	130.24217
F10	start 7	140.78415
F10	complete 7	150.32656
F20	complete 2	150.83212

Task	iteration number	time stamp
F20	start 3	150.85151
F10	start 8	160.86119
F10	complete 8	170.37914
F20	complete 3	180.78681
F10	start 9	180.92557
F10	complete 9	190.44442
F10	start 10	200.9947
F10	complete 10	210.49725
F20	start 4	210.52484
F10	start 11	221.05573
F10	complete 11	230.57817
F20	complete 4	239.74004
F10	start 12	241.11734
F10	complete 12	250.63845
F20	start 5	251.15244
F10	start 13	261.19885
F10	complete 13	270.71478
F20	complete 5	279.91109

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)