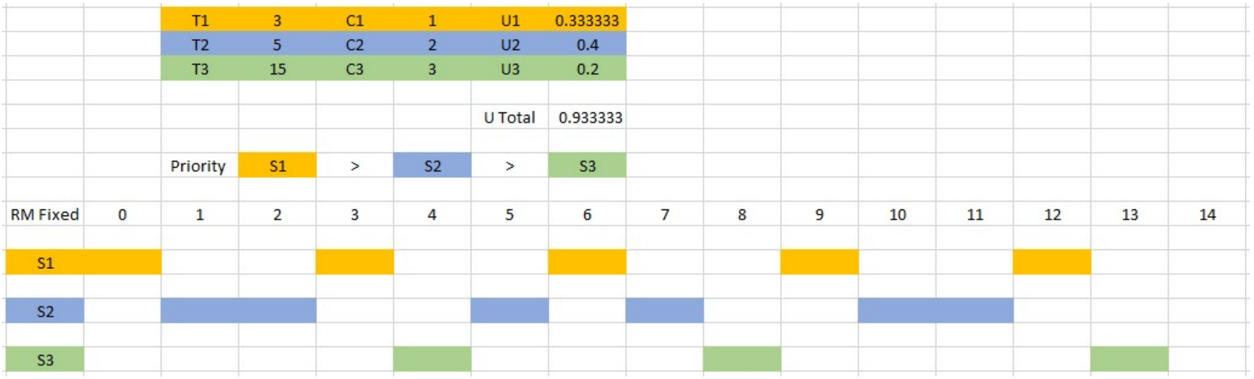
Monish Nene
Real Time Embedded Systems
Exercise 1 Report

1) [20 points] The Rate Monotonic Policy states that services which share a CPU core should multiplex it (with context switches that preempt and dispatch tasks) based on priority, where highest priority is assigned to the most frequently requested service and lowest priority is assigned to the least frequently requested AND total shared CPU core utilization must preserve some margin (not be fully utilized or overloaded). Draw a timing diagram for three services S1, S2, and S3 with T1=3, C1=1, T2=5, C2=2, T3=15, C3=3 where all times are in milliseconds. [Note that you can find examples of timing diagrams in Lecture and here – note that we have not yet covered dynamic priorities, just RM fixed policy described here, so ignore EDF and LLF for now]. 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?

**Answer:**



| | | | | | |
|---|---|---|---|---|---|
| T1 | 3 | C1 | 1 | U1 | 0.333333 |
| T2 | 5 | C2 | 2 | U2 | 0.4 |
| T3 | 15 | C3 | 3 | U3 | 0.2 |
| | | | | U Total | 0.933333 |
| Priority | S1 | > | S2 | > | S3 |

| RM Fixed | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S1 | 🟨 | | | 🟨 | | | 🟨 | | | 🟨 | | | 🟨 | | |
| S2 | | 🟦 | 🟦 | | | 🟦 | | 🟦 | | | 🟦 | 🟦 | | | |
| S3 | | | | | 🟩 | | | | 🟩 | | | | | 🟩 | |

**Method:**

- ➔ S1 occurs every 3rd clock tick and pre-empts other 2 services.
- ➔ S2 occurs when the S1 is not due and the CPU is free.
- ➔ S2 pre-empts S3 if it's interrupt occurs.
- ➔ S3 has lowest priority, so S3 occurs only when the S1 and S2 is not due and the CPU is free.

Un = Cn/Tn
U Total = Σ Un
The CPU utilization is 93.33% (U Total = 0.9333)

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

Least upper bound (for m =3) = 0.7797

U Total > Least upper bound, so we will use Lehoczky, Sha and Ding Theorem to determine if it is a safe scheduling schedule.

The LCM is 15 for T1, T2 and T3. Over a period of 15 clock cycles we can schedule the 3 services without missing any deadline.

**We can conclude that if the services are periodic,**

- ➔ The Schedule is feasible.
- ➔ The Schedule is mathematically repeatable as invariant indefinitely.
- ➔ The Schedule is safe and unlikely to miss a deadline.

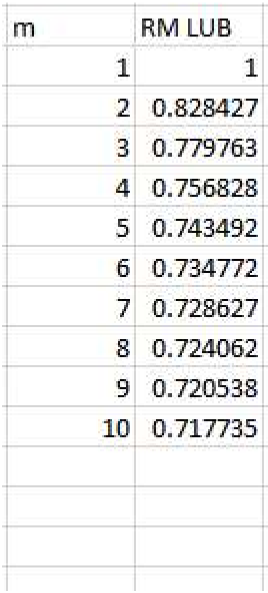
2) [20 points] Read through the Apollo 11 Lunar lander computer overload story as reported in RTECS Notes, based on this NASA account, and the descriptions of the 1201/1202 events described by chief software engineer Margaret Hamilton as recounted by Dylan Matthews. Summarize the story. What was the root cause of the overload and why did it violate Rate Monotonic policy? 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. Plot this Least Upper bound as a function of number of services and describe 3 key assumptions they make and document 3 or more aspects of their fixed priority LUB derivation that you don't understand. Would RM analysis have prevented the Apollo 11 1201/1202 errors and potential mission abort? Why or why not?

**Answer:**

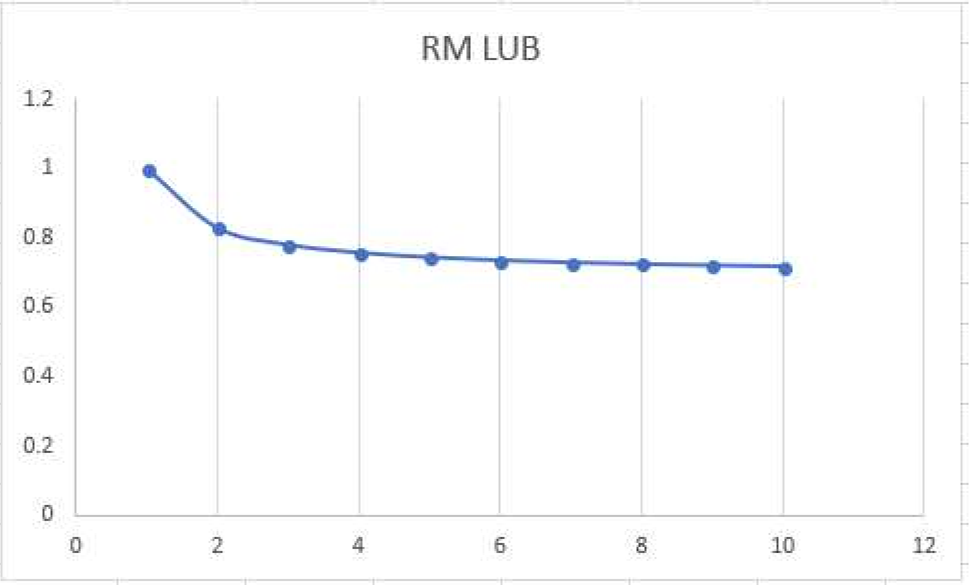
Repeated events to process rendezvous radar data were scheduled because of misconfiguration of the radar switches. These filled the core set of 12 erasable memory locations (seven core sets and five VAC areas) and error 1202 occurred and 1202 alarm turned on. The actual memory overload occurred while landing because the Vector accumulator (VAC) was out of erasable memory locations. At this time, the Alarm 1201 was activated.  The software rebooted and reinitialized the computer, and then restarted selected programs at a point in their execution flow near where they had been when the restart occurred. This solved the problem and the Apollo 11 landed on the moon safely.

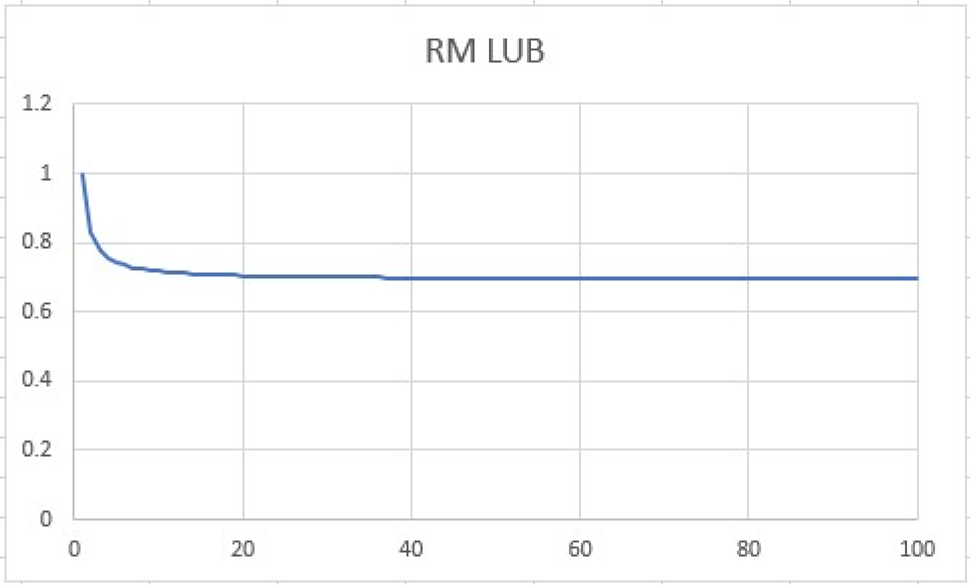
The rate monotonic policy was violated because the 'rendezvous radar data' service which was supposed to be less frequent was having a higher or equal priority than the 'engine burn for a maneuver' service which was supposed to be more frequent.

**RM LUB Plot:**



| m | RM LUB |
|---|---|
| 1 | 1 |
| 2 | 0.828427 |
| 3 | 0.779763 |
| 4 | 0.756828 |
| 5 | 0.743492 |
| 6 | 0.734772 |
| 7 | 0.728627 |
| 8 | 0.724062 |
| 9 | 0.720538 |
| 10 | 0.717735 |





**Key assumptions they make**

- All services are independent of each other.
- All services are periodic.
- The tasks are not recursive.
- Worst case periods are considered while scheduling.
- Tn > Cn

**Aspects of their fixed priority LUB that I don't understand:**

- Is there a highest frequency defined for a service which could have avoided the excessive events of 'rendezvous radar data'?
- If 2 services such that T1 = T2 then but C1 > C2 then how is the priority decided?
- If 2 services such that T1 < T2 then but C1 > C2 then how is the priority decided?

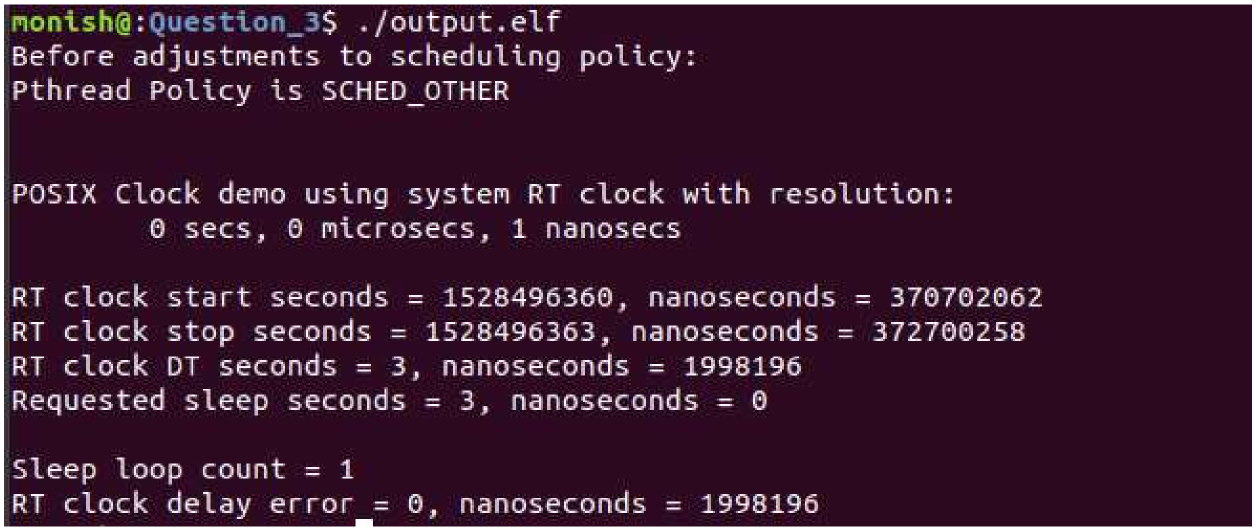
Yes, RM analysis would have prevented the Apollo 11 1201/1202 errors and potential mission abort because it provides optimum scheduling policy for all hard real-time systems based on their deadlines and each task has a chance to be performed.

If high priority was given to the 'engine burn for a maneuver' service which was supposed to be more frequent than the 'rendezvous radar data' service, then even if the misconfiguration of radar switches occurred, the 'engine burn for a maneuver' service would have pre-empted the 'rendezvous radar data' service and thus the core sets would have been available for use.

3) [20 points] Download http://mercury.pr.erau.edu/~siewerts/cec450/code/RT-Clock/ and build it on a Jetson (or ECES Linux if you have not mastered the Jetson yet) and execute the code. Describe what it's 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). 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? Do you believe the accuracy provided by the example RT-Clock code?

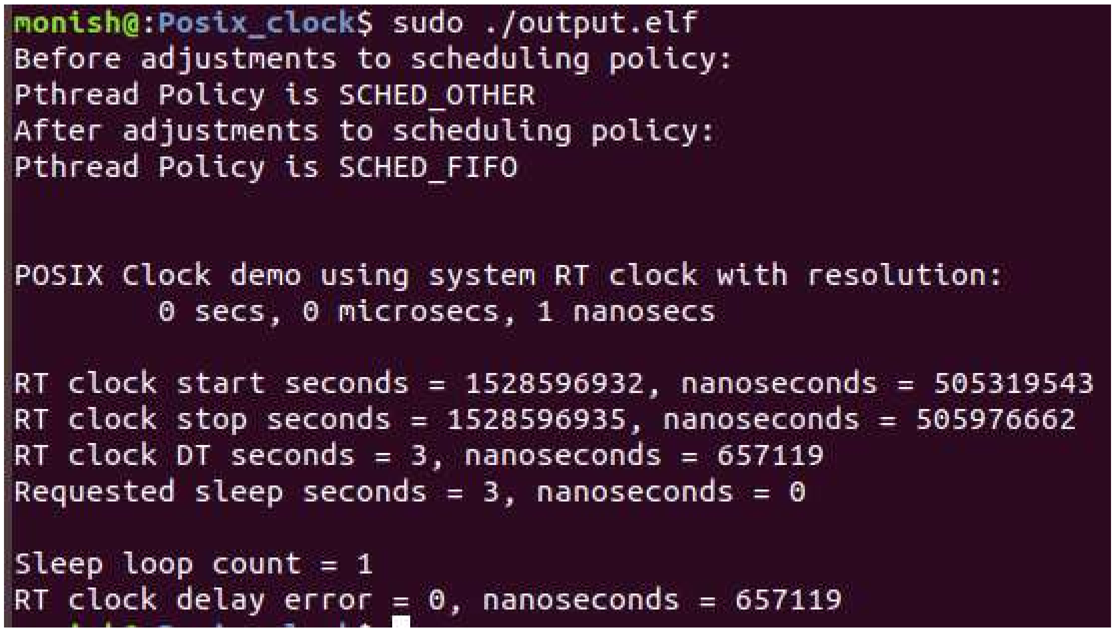
**Answer:**

**Executed posix_clock.c**



```
monish@:Question_3$ ./output.elf
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER


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

RT clock start seconds = 1528496360, nanoseconds = 370702062
RT clock stop seconds = 1528496363, nanoseconds = 372700258
RT clock DT seconds = 3, nanoseconds = 1998196
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 1998196
```

**Executed posix_clock.c after uncommenting #define RUN_RT_THREAD**



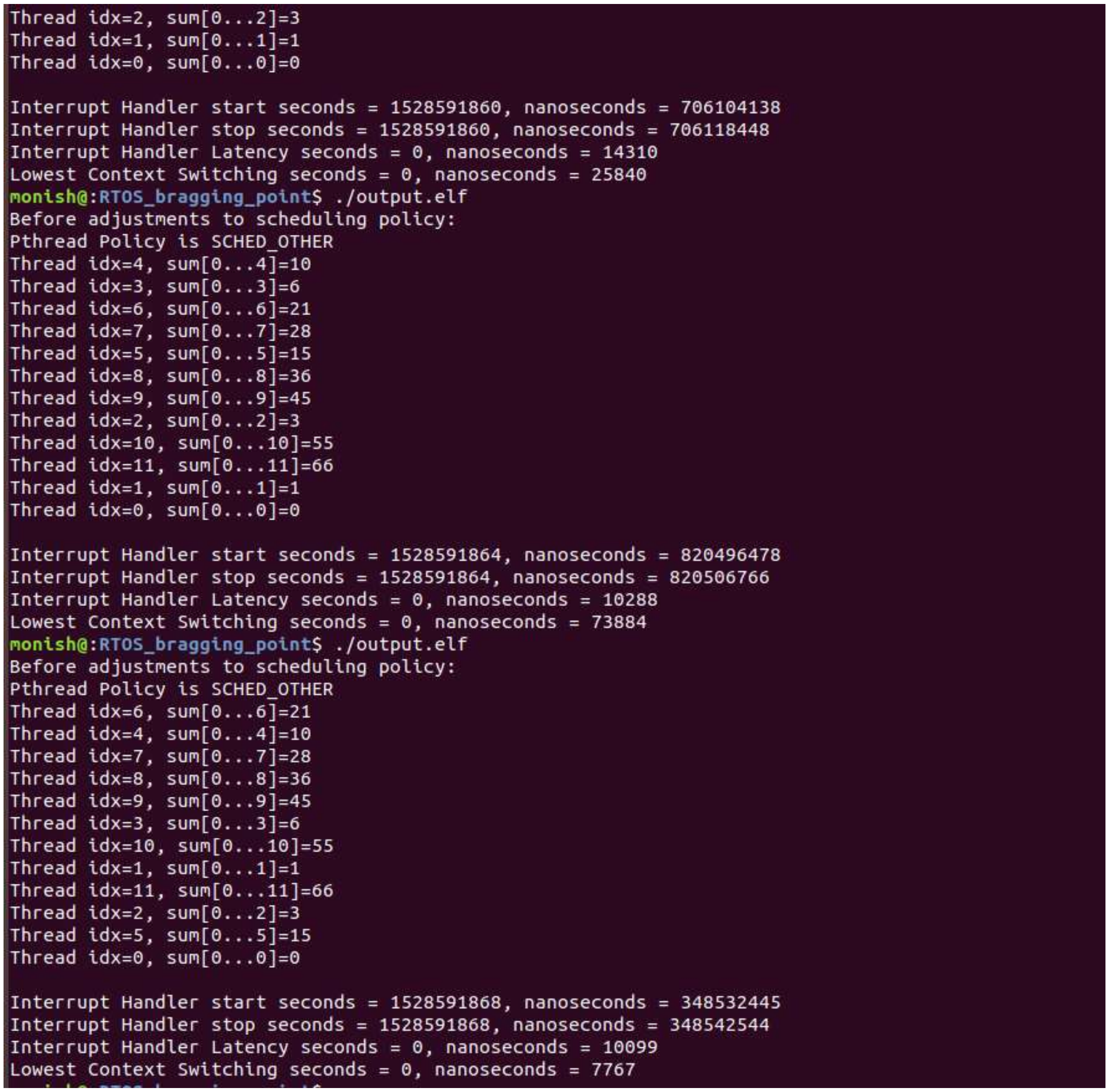
```
monish@:Posix_clock$ sudo ./output.elf
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO


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

RT clock start seconds = 1528596932, nanoseconds = 505319543
RT clock stop seconds = 1528596935, nanoseconds = 505976662
RT clock DT seconds = 3, nanoseconds = 657119
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 657119
```

**Description of code:**

- The code initially checks which Pthread scheduling policy is used and prints it.
- Then it checks the RT clock resolution and displays it.
- It uses the clock_gettime() fuction to get the Standard Real Time Clock value in seconds and nano seconds and saves it as rtclk_start_time.
- pthread SCHED_FIFO policy is selected.
- main thread is created with start routine delay_test and a NULL pointer as argument.
- The thread then calls delay_test function which uses nanosleep() function and puts the processor to sleep for 3 seconds. It is ensured that it enters sleep again if the sleep duration is less than 3 seconds using a do-while loop.
- The clock_gettime() function is used again to get Standard Real Time Clock value in seconds and nano seconds and the value is saved as rtclk_stop_time.
- The difference between the start and stop time is calculated and stored in rtclk_dt.
- The start time, stop time and the difference between them is printed.
- The time difference between rtclk_dt and sleep duration is considered as RT clock delay error and is printed on the terminal.

**Bragging Points:**

I wrote the code using posix_clock.c and pthread.c



```
Thread idx=2, sum[0...2]=3
Thread idx=1, sum[0...1]=1
Thread idx=0, sum[0...0]=0

Interrupt Handler start seconds = 1528591860, nanoseconds = 706104138
Interrupt Handler stop seconds = 1528591860, nanoseconds = 706118448
Interrupt Handler Latency seconds = 0, nanoseconds = 14310
Lowest Context Switching seconds = 0, nanoseconds = 25840
monish@:RTOS_bragging_point$ ./output.elf
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
Thread idx=4, sum[0...4]=10
Thread idx=3, sum[0...3]=6
Thread idx=6, sum[0...6]=21
Thread idx=7, sum[0...7]=28
Thread idx=5, sum[0...5]=15
Thread idx=8, sum[0...8]=36
Thread idx=9, sum[0...9]=45
Thread idx=2, sum[0...2]=3
Thread idx=10, sum[0...10]=55
Thread idx=11, sum[0...11]=66
Thread idx=1, sum[0...1]=1
Thread idx=0, sum[0...0]=0

Interrupt Handler start seconds = 1528591864, nanoseconds = 820496478
Interrupt Handler stop seconds = 1528591864, nanoseconds = 820506766
Interrupt Handler Latency seconds = 0, nanoseconds = 10288
Lowest Context Switching seconds = 0, nanoseconds = 73884
monish@:RTOS_bragging_point$ ./output.elf
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
Thread idx=6, sum[0...6]=21
Thread idx=4, sum[0...4]=10
Thread idx=7, sum[0...7]=28
Thread idx=8, sum[0...8]=36
Thread idx=9, sum[0...9]=45
Thread idx=3, sum[0...3]=6
Thread idx=10, sum[0...10]=55
Thread idx=1, sum[0...1]=1
Thread idx=11, sum[0...11]=66
Thread idx=2, sum[0...2]=3
Thread idx=5, sum[0...5]=15
Thread idx=0, sum[0...0]=0

Interrupt Handler start seconds = 1528591868, nanoseconds = 348532445
Interrupt Handler stop seconds = 1528591868, nanoseconds = 348542544
Interrupt Handler Latency seconds = 0, nanoseconds = 10099
Lowest Context Switching seconds = 0, nanoseconds = 7767
```

**Interrupt Handler Latency:**

- ➔ 7.7us (min) and 800us (max observed):
- ➔ Interrupt latency is the time required by RTOS to respond to external event or when an interrupt is generated.
- ➔ It is the delay between the interrupt call and vector location execution.
- ➔ If it is too high, services might miss deadlines.
- ➔ It must be as low as possible for a better RTOS.

**Low Context Switching time:**

- ➔ The measurement was taken when a thread was entered so the value is not accurate
- ➔ 7.7us (min) and 200us (max observed)
- ➔ It is the time required to switch between two threads.
- ➔ The lower the context switching time the better is the RTOS.

**Stable Timer Services:**

- ➔ The system clock is maintained by operating system
- ➔ Jitter is caused because interrupt have high priority than system calls.
- ➔ When interrupt occurs, system call is stalled.
- ➔ The jitter introduces inaccuracy to the system clock value that the thread gets from the kernel.
- ➔ In commercially available OS the jitter can be several milli seconds.
- ➔ The jitter must be kept minimum for having a good RTOS.

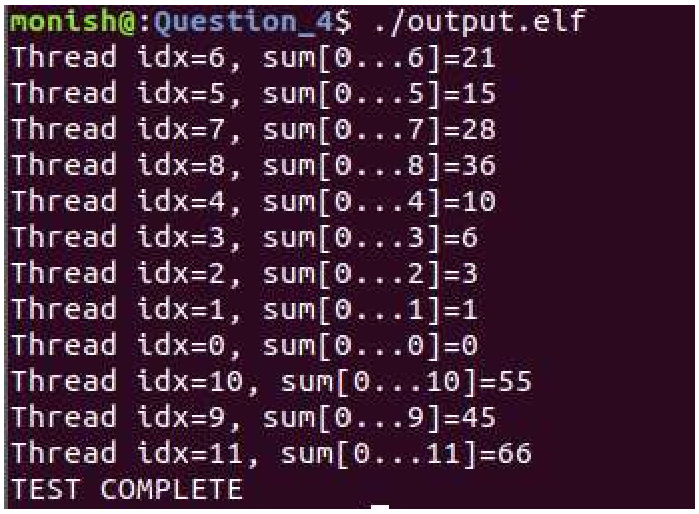
**Accuracy of the RT clock on Linux system**

- ➔ The Accuracy of the RT clock on Linux system is not good because of jitter.
- ➔ The deviation depends on the background processing of the kernel.
- ➔ The deviation is random and can’t be predicted, so it can’t be compensated in the code.
- ➔ If a jitter of few milliseconds can be tolerated, the clock can be reliably used.

4) [40 points] 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. Download, build and run code in http://mercury.pr.erau.edu/~siewerts/cec450/code/simplethread/ and based on the example for creation of 2 threads provided by incdecthread/pthread.c, as well as testdigest.c with use of SCHED_FIFO and sem_post and sem_wait as well as reading of POSIX manual pages as needed - describe how you would attempt to implement Linux code to replicate the LCM invariant schedule implemented in the VxWorks RTOS which produces the schedule measured using event analysis shown below:
The observed timing above fits our theory for RM policy on a priority preemptive scheduling system as shown by the timing diagram below:
You description should outline how you would implement code equivalent to the VxWorks synthetic load generation and schedule emulator. Code the Fib10 and Fib20 synthetic load generation and work to adjust iterations to see if you can at least produce a reliable 10 millisecond and 20 millisecond load on ECES Linux or a Jetson system (Jetson is preferred and should result in more reliable results). Describe whether your able to achieve predictable reliable results in terms of the C (CPU time) values alone and how you would sequence execution.

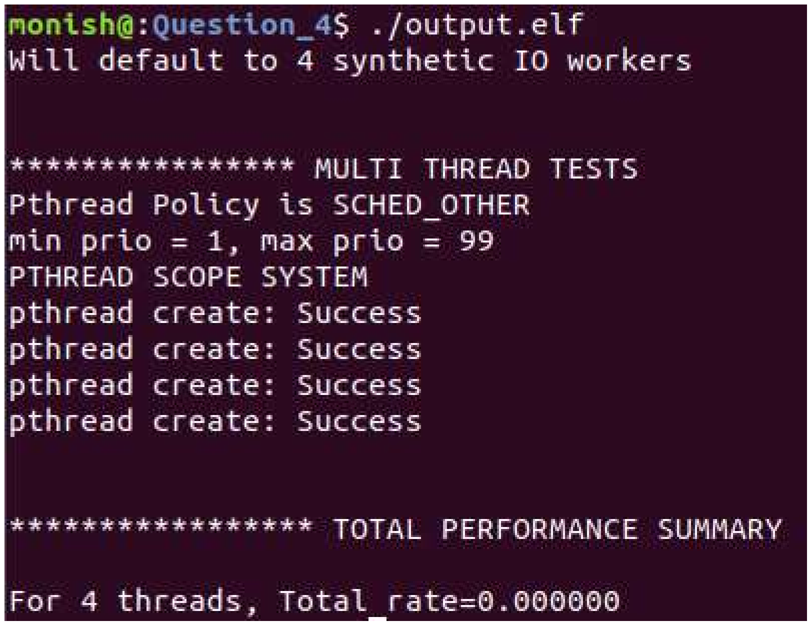
**Answer:**

**Executed pthread.c**



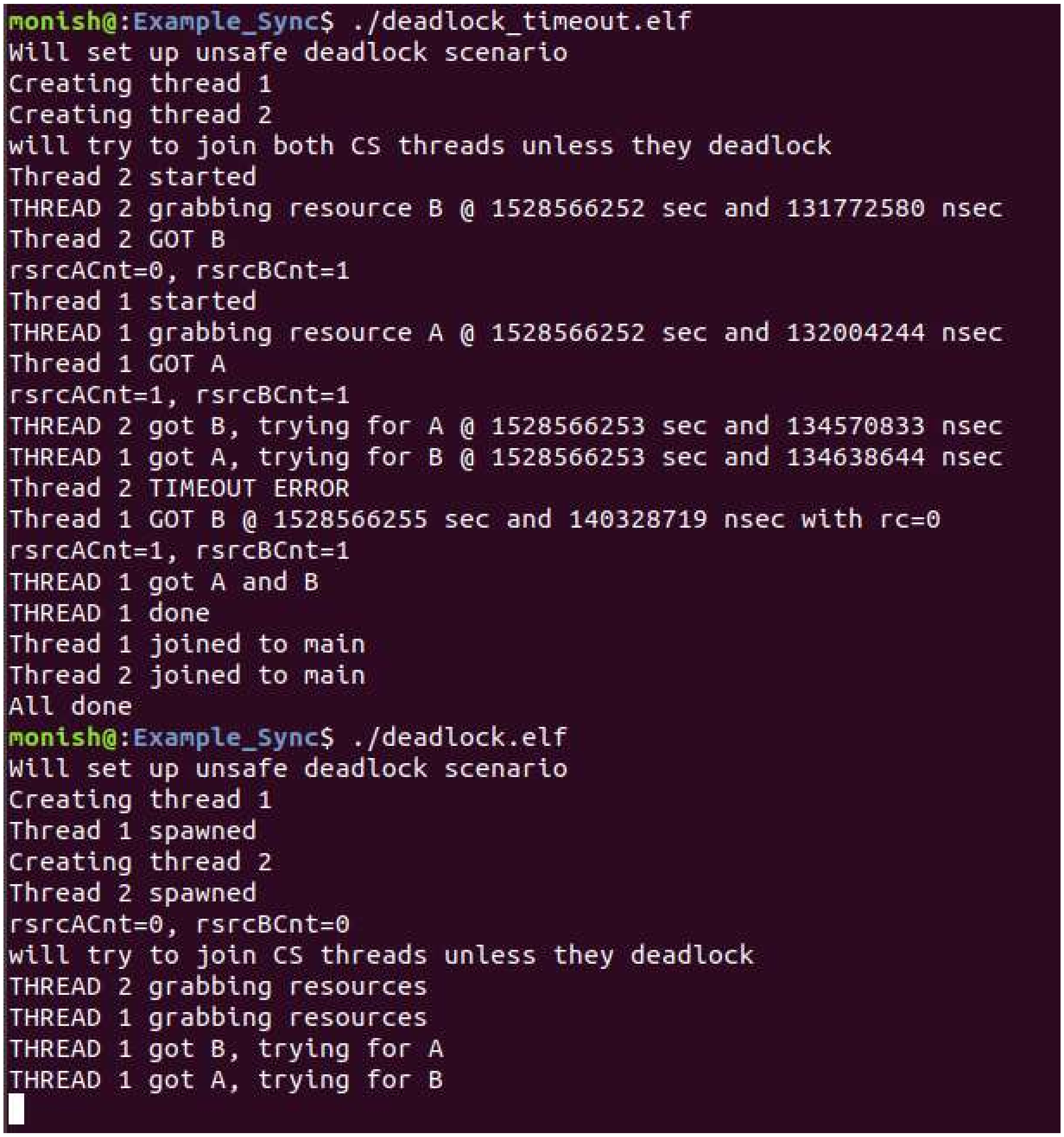
```
monish@:Question_4$ ./output.elf
Thread idx=6, sum[0...6]=21
Thread idx=5, sum[0...5]=15
Thread idx=7, sum[0...7]=28
Thread idx=8, sum[0...8]=36
Thread idx=4, sum[0...4]=10
Thread idx=3, sum[0...3]=6
Thread idx=2, sum[0...2]=3
Thread idx=1, sum[0...1]=1
Thread idx=0, sum[0...0]=0
Thread idx=10, sum[0...10]=55
Thread idx=9, sum[0...9]=45
Thread idx=11, sum[0...11]=66
TEST COMPLETE
```

**Executed testdigest.c**



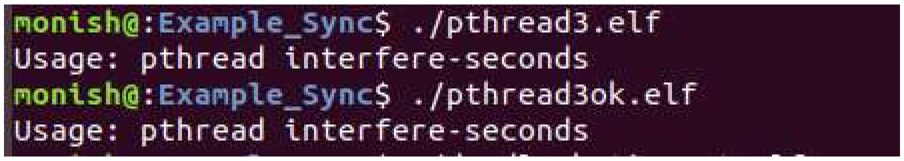
```
monish@:Question_4$ ./output.elf
Will default to 4 synthetic IO workers


**************** MULTI THREAD TESTS
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
pthread create: Success
pthread create: Success
pthread create: Success
pthread create: Success


***************** TOTAL PERFORMANCE SUMMARY

For 4 threads, Total_rate=0.000000
```

**Executed deadlock.c and deadlock_timeout.c**



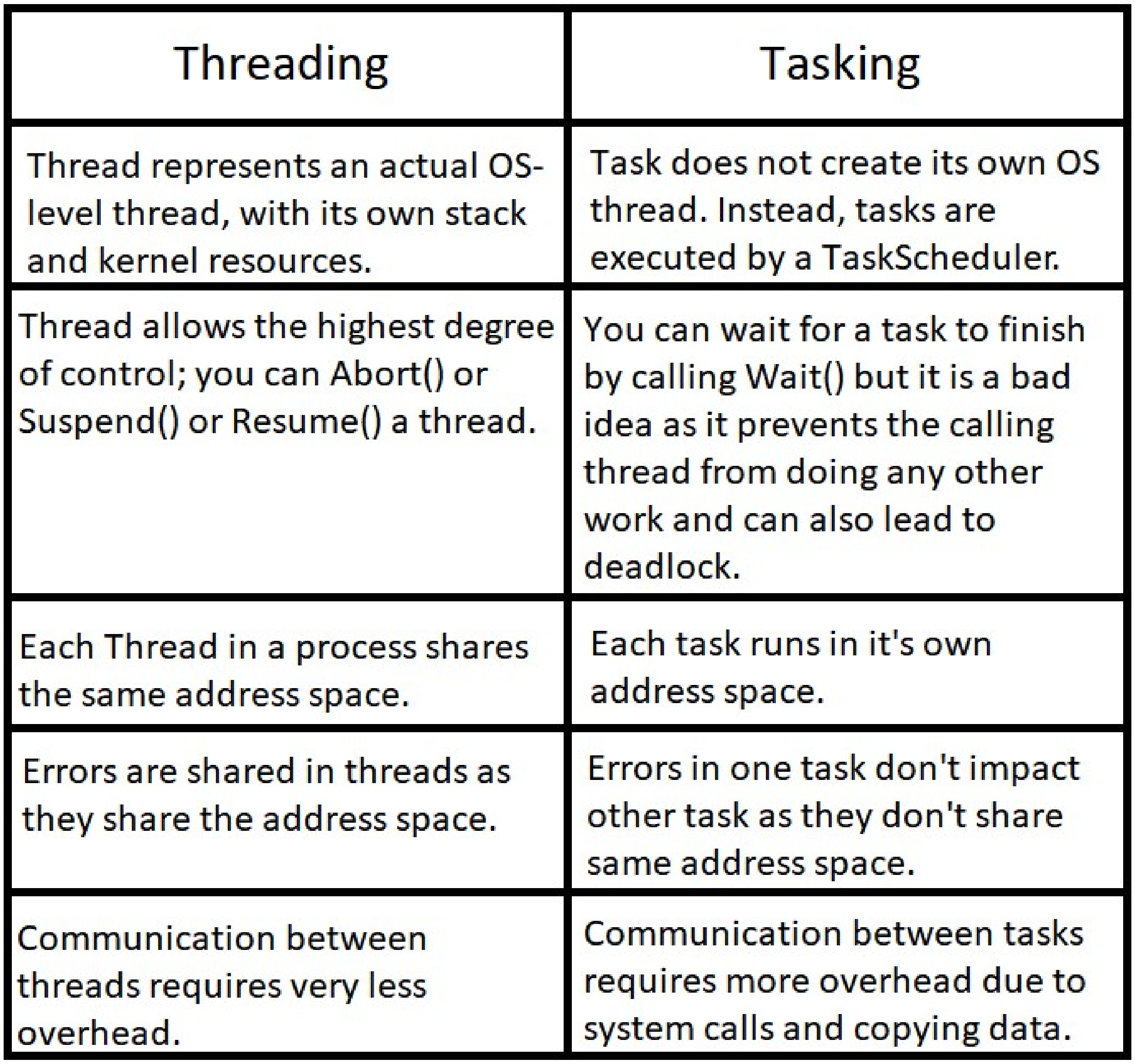
```
monish@:Example_Sync$ ./deadlock_timeout.elf
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1528566252 sec and 131772580 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
Thread 1 started
THREAD 1 grabbing resource A @ 1528566252 sec and 132004244 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1528566253 sec and 134570833 nsec
THREAD 1 got A, trying for B @ 1528566253 sec and 134638644 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1528566255 sec and 140328719 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
Thread 1 joined to main
Thread 2 joined to main
All done
monish@:Example_Sync$ ./deadlock.elf
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 grabbing resources
THREAD 1 got B, trying for A
THREAD 1 got A, trying for B
```

**Executed pthread3.c and pthread3ok.c**



```
monish@:Example_Sync$ ./pthread3.elf
Usage: pthread interfere-seconds
monish@:Example_Sync$ ./pthread3ok.elf
Usage: pthread interfere-seconds
```

**Threading vs. Tasking:**



| Threading | Tasking |
|---|---|
| Thread represents an actual OS-level thread, with its own stack and kernel resources. | Task does not create its own OS thread. Instead, tasks are executed by a TaskScheduler. |
| Thread allows the highest degree of control; you can Abort() or Suspend() or Resume() a thread. | You can wait for a task to finish by calling Wait() but it is a bad idea as it prevents the calling thread from doing any other work and can also lead to deadlock. |
| Each Thread in a process shares the same address space. | Each task runs in it's own address space. |
| Errors are shared in threads as they share the address space. | Errors in one task don't impact other task as they don't share same address space. |
| Communication between threads requires very less overhead. | Communication between tasks requires more overhead due to system calls and copying data. |

**Semaphores:**

A semaphore is a variable used to control access to a resource by multiple threads in an operating system. It is used to solve critical section problems and to achieve process synchronization. They are important tool to prevent race conditions. We have semaphore.h standard c library for semaphores.

**Sem_post:** sem_post() is a function used to unlock a semaphore. It has a pointer to a semaphore as argument. It has a pointer to a semaphore as argument. If there are multiple threads waiting for a semaphore then SCHED_FIFO (first in first out ) or SCHED_RR( round robin) scheduler policy is used to decide the priority of the threads. To unlock a semaphore, it's value is incremented by this function.

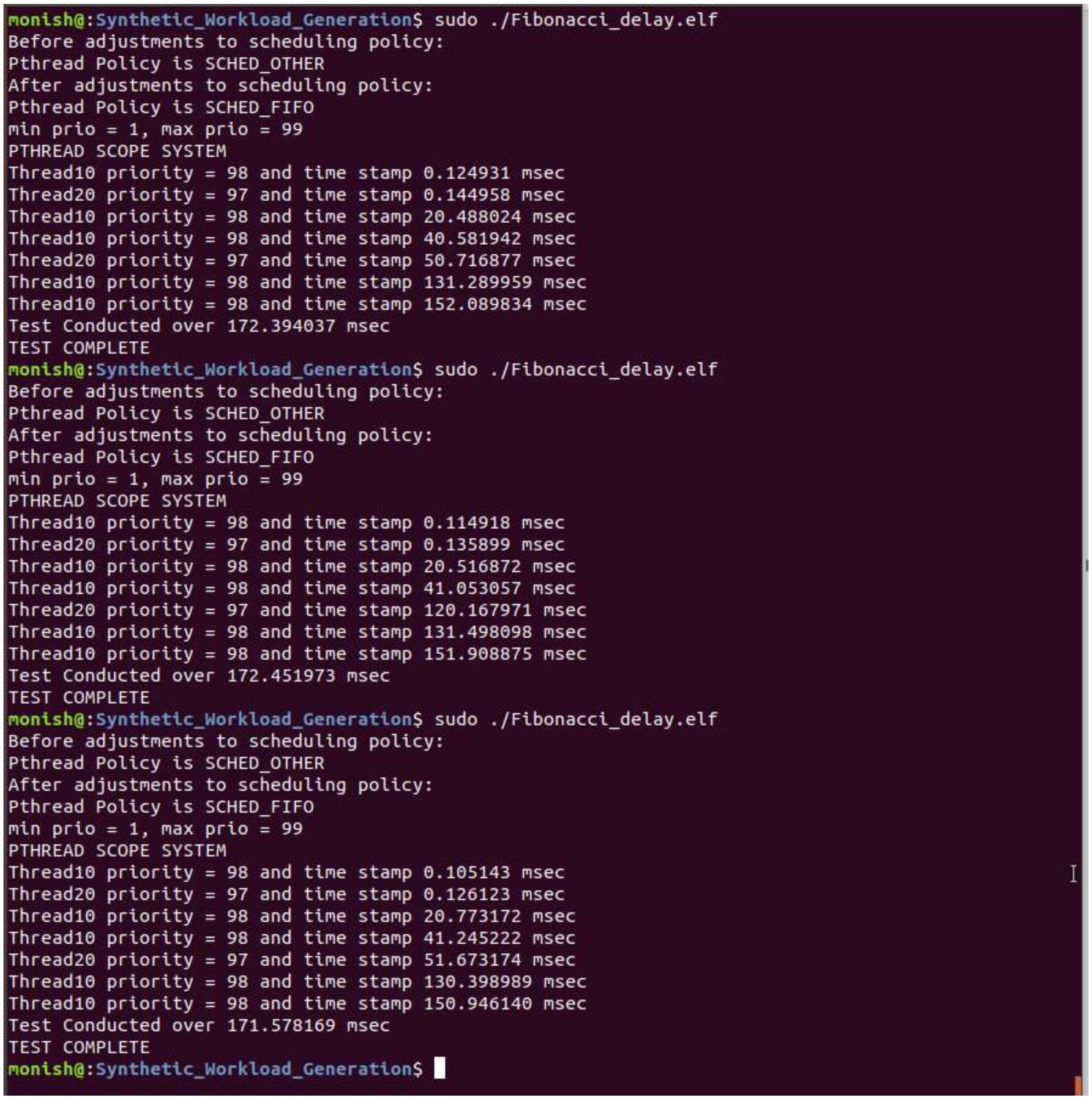
**Sem_wait:** sem_wait() is a function used to lock a semaphore. It is a function that makes a thread wait for a particular semaphore until it is unlocked. To lock a semaphore, it's value is decremented to 0 by this function.

**Synthetic workload generation:**

It is used to schedule a task in a RTOS by creating a sequence of requests based on a workload table specified by the user. A scheduler then executes the task to test the time constraints.

**Synthetic workload analysis and adjustment on test system:**

Synthetic workload analysis is done to test if a system can be repeatedly run for given execution time with stability. Fibonacci loop is fine tuned to achieve execution times of 10ms and 20ms by changing the sequence and iterations in the source code.



```
monish@:Synthetic_Workload_Generation$ sudo ./Fibonacci_delay.elf
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Thread10 priority = 98 and time stamp 0.124931 msec
Thread20 priority = 97 and time stamp 0.144958 msec
Thread10 priority = 98 and time stamp 20.488024 msec
Thread10 priority = 98 and time stamp 40.581942 msec
Thread20 priority = 97 and time stamp 50.716877 msec
Thread10 priority = 98 and time stamp 131.289959 msec
Thread10 priority = 98 and time stamp 152.089834 msec
Test Conducted over 172.394037 msec
TEST COMPLETE
monish@:Synthetic_Workload_Generation$ sudo ./Fibonacci_delay.elf
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Thread10 priority = 98 and time stamp 0.114918 msec
Thread20 priority = 97 and time stamp 0.135899 msec
Thread10 priority = 98 and time stamp 20.516872 msec
Thread10 priority = 98 and time stamp 41.053057 msec
Thread20 priority = 97 and time stamp 120.167971 msec
Thread10 priority = 98 and time stamp 131.498098 msec
Thread10 priority = 98 and time stamp 151.908875 msec
Test Conducted over 172.451973 msec
TEST COMPLETE
monish@:Synthetic_Workload_Generation$ sudo ./Fibonacci_delay.elf
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Thread10 priority = 98 and time stamp 0.105143 msec
Thread20 priority = 97 and time stamp 0.126123 msec
Thread10 priority = 98 and time stamp 20.773172 msec
Thread10 priority = 98 and time stamp 41.245222 msec
Thread20 priority = 97 and time stamp 51.673174 msec
Thread10 priority = 98 and time stamp 130.398989 msec
Thread10 priority = 98 and time stamp 150.946140 msec
Test Conducted over 171.578169 msec
TEST COMPLETE
monish@:Synthetic_Workload_Generation$
```

**Challenges Faced:**

- ➔ Understanding various function calls, Compile time switches, priority levels, inheritance concepts, etc
- ➔ The system required different duration on every execution.
- ➔ The test system works perfectly as per scheduling.

**References:**

1. Real-Time Embedded Components and Systems – Sam Siewart
2. Apollo 11 Program Alarms – Peter Adler https://www.hq.nasa.gov/alsj/a11/a11.1201-pa.html
3. Liu and Layland's paper – Scheduling Algorithms for Multiprogramming in a Hard-real-time Environment
4. LLNL (Lawrence Livermore National Labs) pages on pthreads https://computing.llnl.gov/tutorials/pthreads/
5. The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior – Jon Lehoczky, Lui Sha and Ye Ding
6. Independent Study – RM Scheduling Feasibility Tests conducted on TI DM3730 Processor - 1 GHz ARM Cortex-A8 core with Angstrom and TimeSys Linux ported on to BeagleBoard xM - Nisheeth Bhat http://ecee.colorado.edu/~ecen5623/ecen/labs/Linux/IS/Report.pdf
7. https://stackoverflow.com/questions/13429129/task-vs-thread-differences
8. https://en.wikipedia.org/wiki/Semaphore_(programming)
9. http://pubs.opengroup.org/onlinepubs/009604499/functions/sem_post.html
10. http://pubs.opengroup.org/onlinepubs/009604499/functions/sem_wait.html