ECEN 5623: REAL-TIME EMBEDDED SYSTEMS

EXERCISE 3

Submitted By: Krishna Suhagiya and Suhas Reddy
Submission Date: 9th March 2024
Hardware: Jetson Nano

Table of Contents

Question 1

[10 points] [All papers here also on Canvas] Read Sha, Rajkumar, et al paper, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization"

a) Summarize 3 main key points the paper makes. Read Dr. Siewert's summary paper on the topic as well which might be easier to understand.

Answer: The paper primarily examines the synchronization issue in the context of priority-driven preemptive scheduling. It examines two protocols that fall under the priority inheritance protocol class: the priority ceiling protocol and the basic priority inheritance protocol. It also outlines the protocols' presumptions, issues, and benefits.

- Priority inversion problem
  - Priority inversion is a phenomenon where a higher priority job is blocked by lower priority jobs.
  - This type of scenario usually occurs when two jobs try to access common data. If the higher priority job gains access first and then the correct priority order is maintained but if the lower priority job gains access first and then the higher priority job requests access to the shared data, the higher priority job is blocked until the lower priority job completes its access to the shared data.
  - One kind of priority inversion is blocking, in which a higher priority job has to wait for a lower priority activity to be processed. Even with minimal resource usage, extended blocking can cause deadlines to be missed.
- Priority inheritance protocol
  - In the priority inheritance protocol, a task that blocks one or more higher priority jobs runs its crucial section at the highest priority level among all the jobs it blocks, disregarding its initial priority assignment. After exiting its critical section, the job returns to its original priority level.
  - The paper also explains with examples that the priority inversion protocol does not prevent deadlocks.
- Priority ceiling protocol
  - Under the priority ceiling protocol, a task that preempts another job's critical section and then executes its own critical section will always execute at a higher priority than all of the preempted critical sections' inherited priorities.
  - The priority ceiling protocol is simple to implement and prevents the deadlocks and reduces the blocking to at most one critical section.
  - The protocol can be enhanced by defining priority floor as the priority of the lowest priority job that may access it.

b) Read the historical positions of Linus Torvalds as described by Jonathan Corbet and Ingo Molnar and Thomas Gleixner on this topic as well. Take a position on this topic yourself and write at least one well supported paragraph or more to defend your position based on what we have learned in class. Does the PI-futex (Futex, Futexes are Tricky) that is described by Ingo Molnar provide safe and accurate protection from un-bounded priority inversion as described in the paper? If not, what is different about it?

Answer: The historical perspectives of Linus Torvalds, detailed by Jonathan Corbet, Ingo Molnar, and Thomas Gleixner, highlight the ongoing debate surrounding Linux's real-time functionalities and priority inversion. Torvalds has consistently emphasized the general-purpose nature of Linux. In contrast, Molnar and Gleixner have worked for enhancing Linux's real-time capabilities, supporting patches like RT_PREEMPT.

- Taking a stance on this issue, we argue that while patches such as RT_PREEMPT are beneficial for boosting Linux's real-time performance, they may not offer a complete solution to unbounded priority inversion. The PI-futex (Priority Inheritance futex) proposed by Ingo Molnar does provide some safeguard against priority inversion. However, it may not be foolproof in all complex scenarios.
- The PI-futex operates by enabling a task holding a futex lock to inherit the priority of a waiting task, thus preventing priority inversion by temporarily boosting the lower-priority task's priority. Nonetheless, the effectiveness of PI-futex can be limited in intricate situations involving multiple resources or deeply nested locks.
- In real-time systems, particularly those with stringent requirements, effectively addressing unbounded priority inversion often requires a combination of proper application design, utilization of synchronization mechanisms such as priority inheritance and priority ceiling emulation, and improvements at the kernel level. Although PI-futex represents improvement, it may not fully guard against unbounded priority inversion in all circumstances.
- To summarize, while the PI-futex mechanism proposed by Ingo Molnar provides some protection against unbounded priority inversion, it is not a flawless solution. RT_PREEMPT and similar enhancements are valuable for enhancing Linux's real-time capabilities. Nonetheless, real-time system designers should carefully assess their application needs and implement additional measures to effectively mitigate the risks of priority inversion.

c) Note that more recently Red Hat has added support for priority ceiling emulation and priority inheritance and has a great overview on use of these POSIX real-time extension features here – general real-time overview. The key systems calls are pthread_mutexattr_setprotocol, pthread_mutexattr_getprotocol as well as pthread_mutex_setpriorityceiling and pthread_mutex_getpriorityceiling. Why did some of the Linux community resist addition of priority ceiling and/or priority inheritance until more recently? (Linux has been distributed since the early 1990's and the problem and solutions were known before this).

Answer: The delay in adding priority ceiling and priority inheritance mechanisms to the Linux kernel's real-time capabilities can be mainly because of following reasons:

- The implementation of these mechanisms is complex and has potential impacts on kernel stability and performance.
- Linux was initially developed as a general-purpose OS, not real time OS. Also, there were external patch sets available, satisfying real-time needs to some extent. So, adding features such as priority ceiling and priority inheritance might introduce complexity for non-real-time users.
- Since linux is an open-source model, it requires careful consideration, discussion, and testing before major features are added.
- Maintaining compatibility with existing code and behavior was crucial, requiring thorough evaluation.

- There might have been some concerns on making the code open-source GPL license compliant.
- As industries like industrial automation and telecommunications demanded real-time capabilities, the pressure to add these features to Linux increased. Also, there was a competition from real-time OSs offering these mechanisms, pushing for their inclusion.
- We believe that balancing the demands of a larger user base with the particularities of real-time systems caused a delay in the inclusion of priority ceiling and priority inheritance in Linux's real-time capabilities. The Linux community worked to overcome obstacles and incorporate these techniques into the kernel over time as demand increased and the benefits became more apparent.

d) Given that there are two solutions to unbounded priority inversion, priority ceiling emulation and priority inheritance, based upon your reading on support for both by POSIX real-time extensions, which do you think is best to use and why?

Answer: Priority ceiling emulation and priority inheritance are mechanisms designed to address the problem of unbounded priority inversion. They can be used based on the requirements and capability of the system. Following are the major implementation pointers that can help deciding the solution for a specific system:

- Priority Ceiling Emulation:
    - The designing of priority ceiling emulation is simple based on a fixed upper bound on the priority of tasks holding a lock.
    - It prevents deadlocks and reduces blocking to at most one critical section.
    - Since its behavior is static, it is less flexible compared to priority inheritance.
- Priority Inheritance:
    - The design of priority inheritance is complex as it dynamically adjusts the priority of tasks holding a lock to that of the highest-priority task waiting for the lock.
    - It may introduce more overhead due to dynamic priority adjustments.

Question 2

[25 points] Review the terminology guide (glossary in the textbook)
a) Describe clearly what it means to write "thread safe" functions that are "re-entrant". There are generally three main ways to do this: 1) pure functions that use only stack and have no global memory, 2) functions which use thread indexed global data, and 3) functions which use shared memory global data, but synchronize access to it using a MUTEX semaphore critical section wrapper.
Answer:
Thread-Safe Functions: Thread safety implies that a function can be safely executed by multiple threads concurrently without causing unexpected behavior or data corruption. A thread-safe function ensures that shared resources are accessed in a way that avoids conflicts, often through the use of synchronization mechanisms like mutexes. The primary goal of thread safety is to prevent data races and ensure that the function behaves as expected when called concurrently by different threads. This is achieved by protecting critical sections of code using mutexes/semaphores or other synchronization mechanisms, allowing only one thread to access shared resources at a time.

Reentrant Functions: A reentrant function goes a step further; it not only allows concurrent execution by multiple threads but also allows for interruption and resumption of its execution without affecting the function's correctness. Reentrant functions typically avoid the use of global variables, and if they do use them, they ensure proper synchronization to prevent conflicts. The reentrancy of a function is crucial in scenarios where the function may be interrupted and then resumed by the same or a different thread. This interruption can occur, for example, in preemptive multitasking environments or when asynchronous events occur. Reentrant functions achieve this by either allocating unique copies of global data for each execution context (as in VxWorks task variables), providing mutex protection for global data, or avoiding the use of global data entirely by implementing the function as a pure function. Having all these attributes makes a reentrant function inherently thread-safe, ensuring correct behavior even in the presence of interruptions and concurrent executions.

b) Describe each method and how you would code it and how it would impact real-time threads/tasks.
Answer:
Methods for Writing Thread-Safe and Reentrant Functions:
Pure Functions with Stack-only Memory:
Coding Approach: In this method, functions are designed to be pure functions, meaning they rely only on local stack memory and do not use any global variables. They avoid shared state altogether.
Impact on Real-Time Threads/Tasks: This method is highly suitable for real-time systems where determinism is crucial. Since there are no shared global variables, there is no need for synchronization mechanisms like mutexes. Real-time threads can execute these functions without the risk of contention over shared data, ensuring predictable and consistent performance.

Functions with Thread-Indexed Global Data:

Coding Approach: Functions using this method allocate a unique copy of global data for each thread. Thread-specific data or thread-local storage is often utilized to achieve this.
Impact on Real-Time Threads/Tasks: This method is suitable for real-time systems where threads have their own independent data. It minimizes contention since each thread operates on its dedicated global data. However, care must be taken to manage the thread-local data appropriately, especially when threads need to communicate or share information.

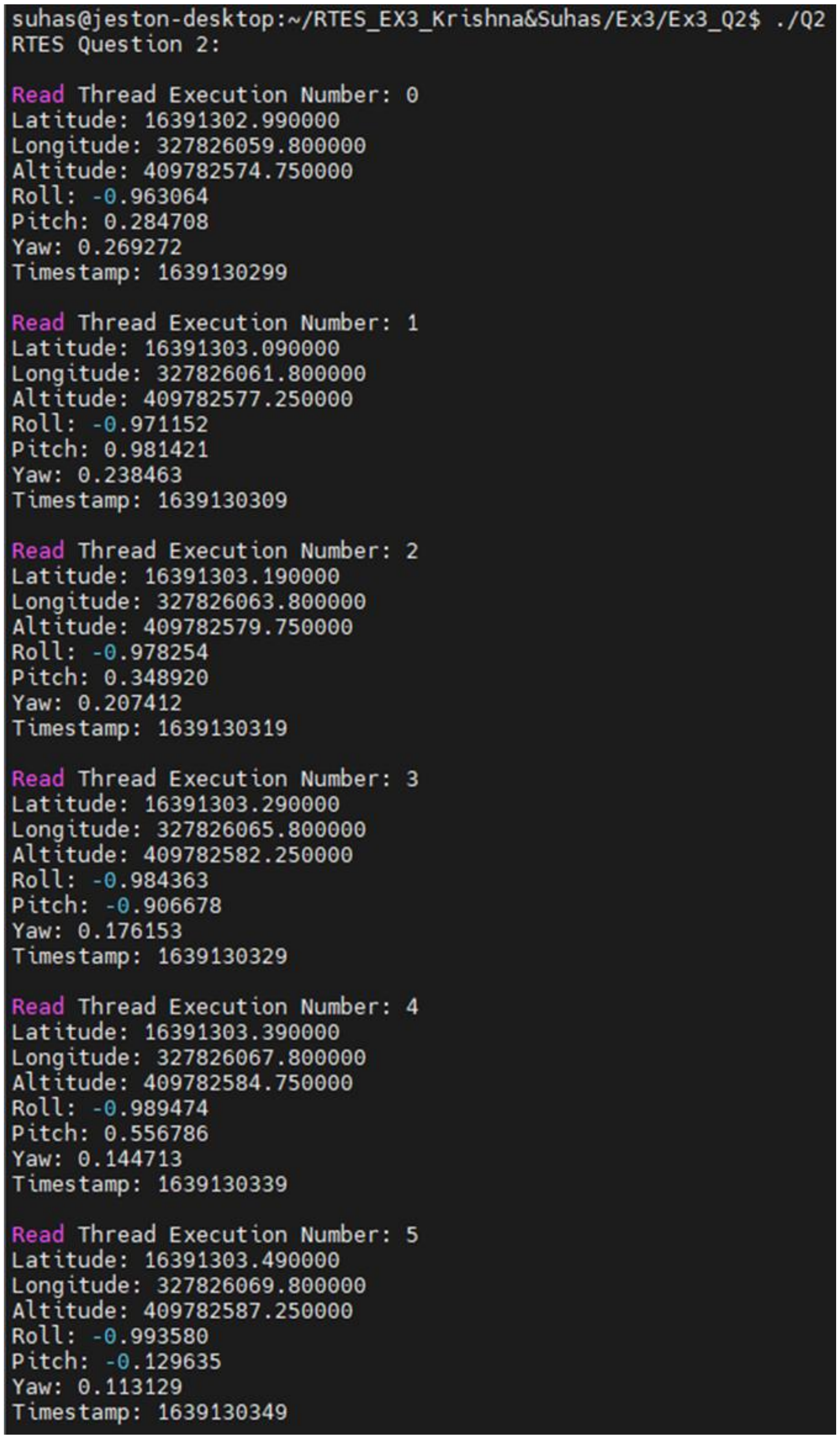
Functions with Shared Memory and MUTEX Synchronization:
Coding Approach: Functions in this category use shared global data but ensure access to this data is synchronized using mutexes or other semaphore mechanisms.
Impact on Real-Time Threads/Tasks: This method is applicable in scenarios where shared data is necessary but must be accessed safely. The use of mutexes introduces some level of contention management, which might impact real-time performance. It is crucial to minimize the time spent holding the mutex to avoid delaying other threads.

In real-time systems, the choice of method depends on the specific requirements and constraints of the application. Pure functions with stack-only memory are preferable for critical, low-latency tasks, while the other methods provide flexibility when shared state is necessary, but proper synchronization is required to maintain thread safety and reentrancy.

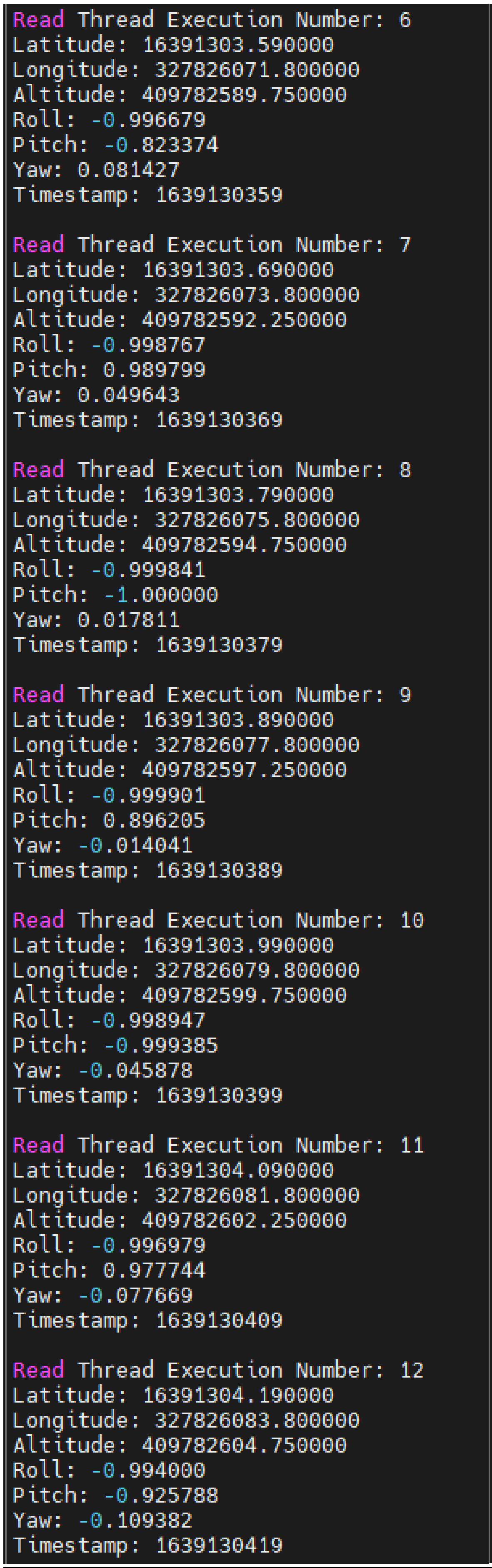
b) Now, using a MUTEX, provide an example using RT-Linux Pthreads that does a thread safe update of a complex state (3 or more numbers – e.g., Latitude, Longitude and Altitude of a location) with a timestamp (pthread_mutex_lock). Your code should include two threads and one should update a timespec structure contained in a structure that includes a double precision attitude state of {Lat, Long, Altitude and Roll, Pitch, Yaw at Sample_Time} and the other should read it and never disagree on the values as function of time. You can just make up values for the navigational state using math library function generators (e.g., use simple periodic functions for Roll, Pitch, Yaw sin(x), cos(x2), and cos(x), where x=time and linear functions for Lat, Long, Alt) and see http://linux.die.net/man/3/clock_gettime for how to get a precision timestamp). The second thread should read the time-stamped state without the possibility of data corruption (partial update of one of the 6 floating point values). There should be no disagreement between the functions and the state reader for any point in time. Run this for 180 seconds with a 1 Hz update rate and a 0.1 Hz read rate. Make sure the 18 values read are correct. E.g., Function Generators - https://www.desmos.com/calculator

Answer:
Output: The code does not utilize a global variable for the nav_state struct; instead, it employs a parameterized approach to implement thread-safe operations. Additionally, it utilizes mutex for synchronized access in case of global variables for nav_state this scenario can be seen in the Q5 solution. Additionally, it employs signal mutex to ensure deterministic behavior, always starting with the update function before the read function.

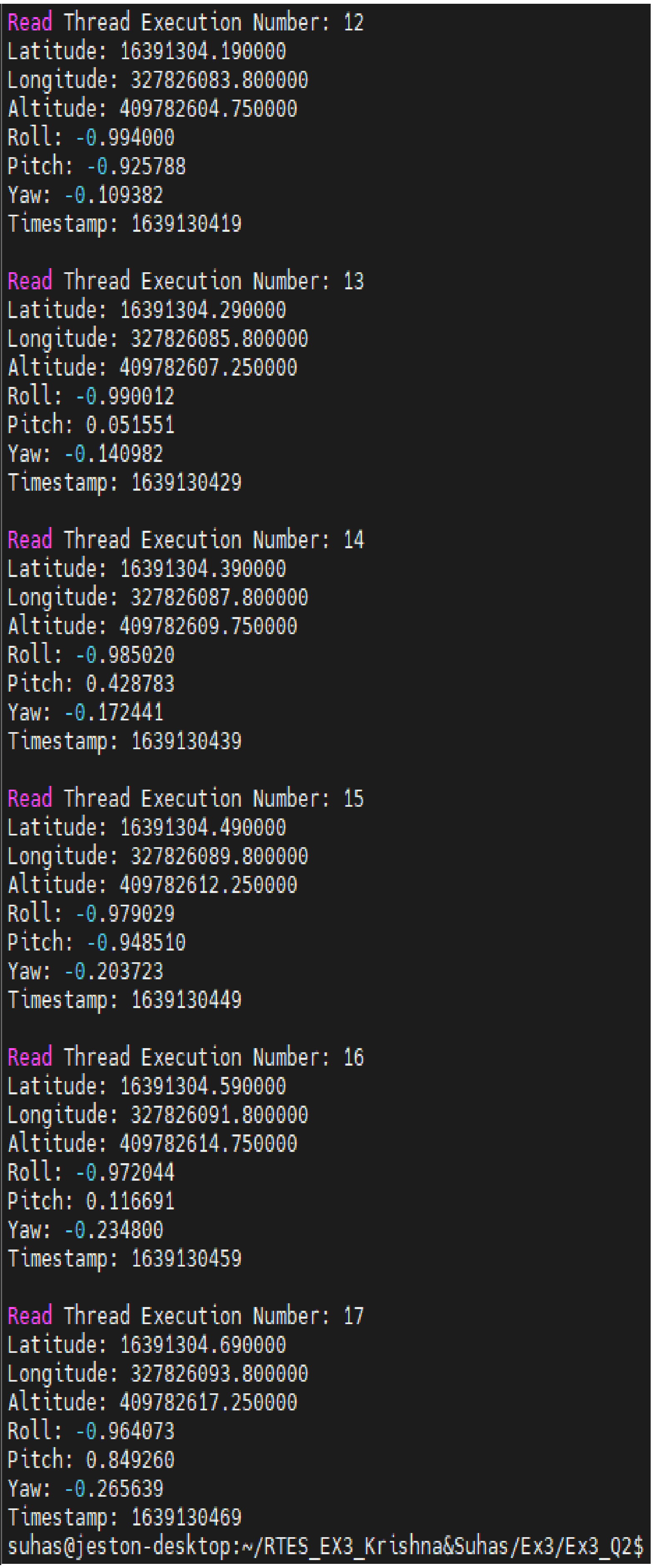
The threads run for a total of 180 seconds, where nav_state is updated every second and read and printed every 10 seconds.



```
suhas@jeston-desktop:~/RTES_EX3_Krishna&Suhas/Ex3/Ex3_Q2$ ./Q2
RTES Question 2:

Read Thread Execution Number: 0
Latitude: 16391302.990000
Longitude: 327826059.800000
Altitude: 409782574.750000
Roll: -0.963064
Pitch: 0.284708
Yaw: 0.269272
Timestamp: 1639130299

Read Thread Execution Number: 1
Latitude: 16391303.090000
Longitude: 327826061.800000
Altitude: 409782577.250000
Roll: -0.971152
Pitch: 0.981421
Yaw: 0.238463
Timestamp: 1639130309

Read Thread Execution Number: 2
Latitude: 16391303.190000
Longitude: 327826063.800000
Altitude: 409782579.750000
Roll: -0.978254
Pitch: 0.348920
Yaw: 0.207412
Timestamp: 1639130319

Read Thread Execution Number: 3
Latitude: 16391303.290000
Longitude: 327826065.800000
Altitude: 409782582.250000
Roll: -0.984363
Pitch: -0.906678
Yaw: 0.176153
Timestamp: 1639130329

Read Thread Execution Number: 4
Latitude: 16391303.390000
Longitude: 327826067.800000
Altitude: 409782584.750000
Roll: -0.989474
Pitch: 0.556786
Yaw: 0.144713
Timestamp: 1639130339

Read Thread Execution Number: 5
Latitude: 16391303.490000
Longitude: 327826069.800000
Altitude: 409782587.250000
Roll: -0.993580
Pitch: -0.129635
Yaw: 0.113129
Timestamp: 1639130349
```



```
Read Thread Execution Number: 6
Latitude: 16391303.590000
Longitude: 327826071.800000
Altitude: 409782589.750000
Roll: -0.996679
Pitch: -0.823374
Yaw: 0.081427
Timestamp: 1639130359

Read Thread Execution Number: 7
Latitude: 16391303.690000
Longitude: 327826073.800000
Altitude: 409782592.250000
Roll: -0.998767
Pitch: 0.989799
Yaw: 0.049643
Timestamp: 1639130369

Read Thread Execution Number: 8
Latitude: 16391303.790000
Longitude: 327826075.800000
Altitude: 409782594.750000
Roll: -0.999841
Pitch: -1.000000
Yaw: 0.017811
Timestamp: 1639130379

Read Thread Execution Number: 9
Latitude: 16391303.890000
Longitude: 327826077.800000
Altitude: 409782597.250000
Roll: -0.999901
Pitch: 0.896205
Yaw: -0.014041
Timestamp: 1639130389

Read Thread Execution Number: 10
Latitude: 16391303.990000
Longitude: 327826079.800000
Altitude: 409782599.750000
Roll: -0.998947
Pitch: -0.999385
Yaw: -0.045878
Timestamp: 1639130399

Read Thread Execution Number: 11
Latitude: 16391304.090000
Longitude: 327826081.800000
Altitude: 409782602.250000
Roll: -0.996979
Pitch: 0.977744
Yaw: -0.077669
Timestamp: 1639130409

Read Thread Execution Number: 12
Latitude: 16391304.190000
Longitude: 327826083.800000
Altitude: 409782604.750000
Roll: -0.994000
Pitch: -0.925788
Yaw: -0.109382
Timestamp: 1639130419
```



```
Read Thread Execution Number: 12
Latitude: 16391304.190000
Longitude: 327826083.800000
Altitude: 409782604.750000
Roll: -0.994000
Pitch: -0.925788
Yaw: -0.109382
Timestamp: 1639130419

Read Thread Execution Number: 13
Latitude: 16391304.290000
Longitude: 327826085.800000
Altitude: 409782607.250000
Roll: -0.990012
Pitch: 0.051551
Yaw: -0.140982
Timestamp: 1639130429

Read Thread Execution Number: 14
Latitude: 16391304.390000
Longitude: 327826087.800000
Altitude: 409782609.750000
Roll: -0.985020
Pitch: 0.428783
Yaw: -0.172441
Timestamp: 1639130439

Read Thread Execution Number: 15
Latitude: 16391304.490000
Longitude: 327826089.800000
Altitude: 409782612.250000
Roll: -0.979029
Pitch: -0.948510
Yaw: -0.203723
Timestamp: 1639130449

Read Thread Execution Number: 16
Latitude: 16391304.590000
Longitude: 327826091.800000
Altitude: 409782614.750000
Roll: -0.972044
Pitch: 0.116691
Yaw: -0.234800
Timestamp: 1639130459

Read Thread Execution Number: 17
Latitude: 16391304.690000
Longitude: 327826093.800000
Altitude: 409782617.250000
Roll: -0.964073
Pitch: 0.849260
Yaw: -0.265639
Timestamp: 1639130469
suhas@jeston-desktop:~/RTES_EX3_Krishna&Suhas/Ex3/Ex3_Q2$
```

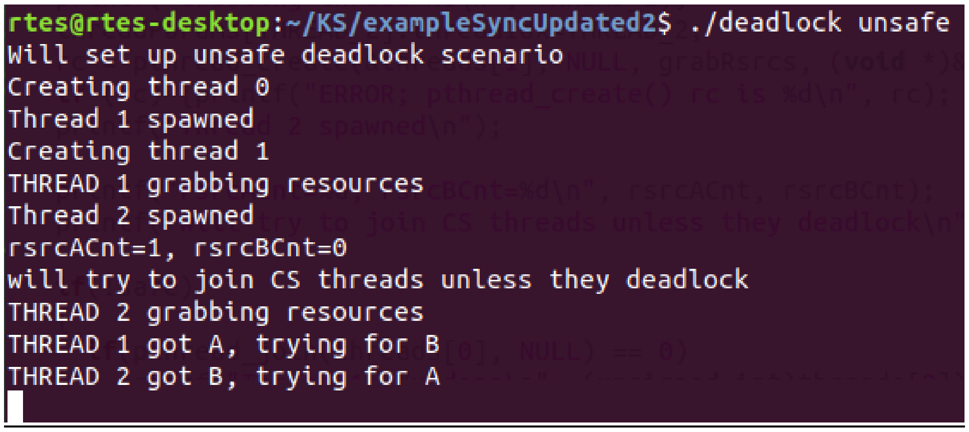
Question 3

[20 points] Download example-sync-updated-2/ and review, build, and run it.
a) Describe both the issues of deadlock and unbounded priority inversion and the root cause for both in the example code.

Answer:

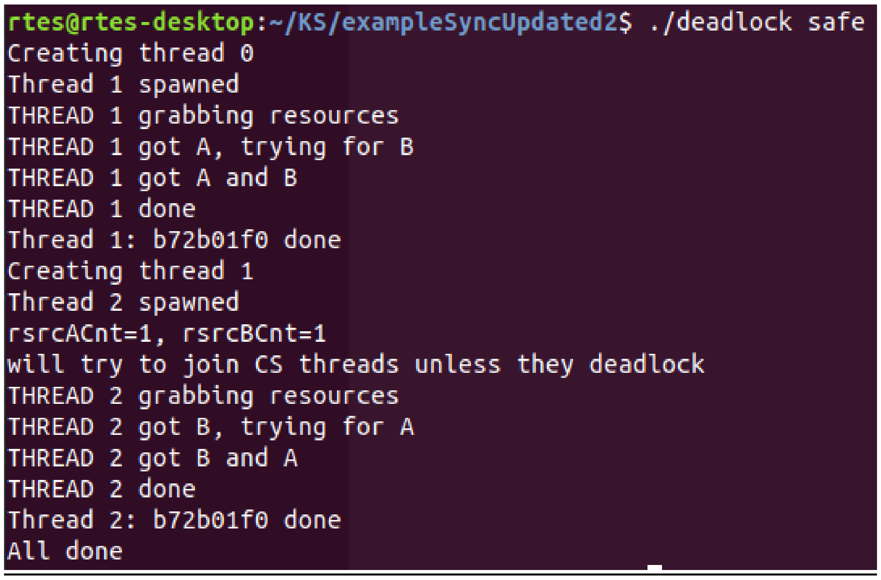
deadlock.c:

- If the 'unsafe' argument is passed as shown below, there is a chance of a deadlock occurring. This is because both threads are attempting to acquire resources in a different order. If the timing is such that THREAD_1 acquires rsrcA and THREAD_2 acquires rsrcB, but then each thread tries to acquire the resource the other thread holds (rsrcB for THREAD_1 and rsrcA for THREAD_2), they will both be blocked waiting for the other to release the resource they need and the system results into deadlock.



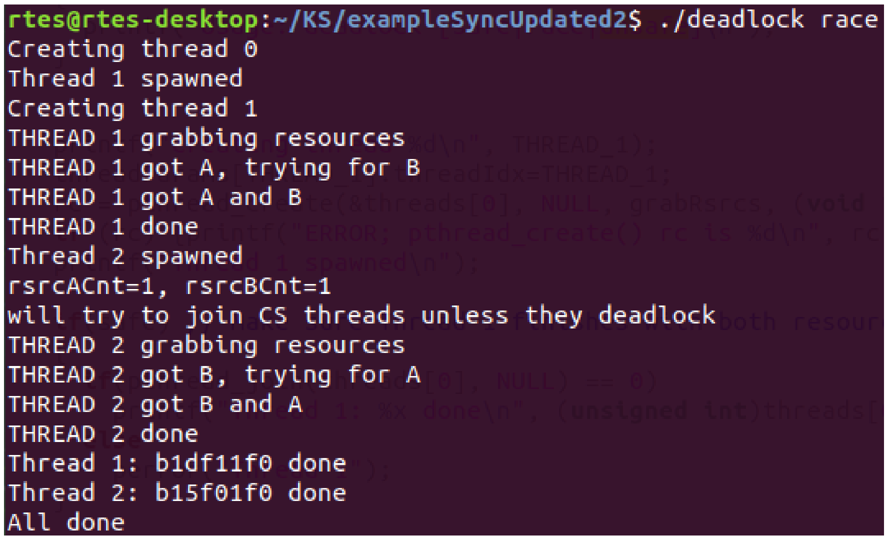
```
rtes@rtes-desktop:~/KS/exampleSyncUpdated2$ ./deadlock unsafe
Will set up unsafe deadlock scenario
Creating thread 0
Thread 1 spawned
Creating thread 1
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got A, trying for B
THREAD 2 got B, trying for A
```

- When the code is executed with the 'safe' argument, it ensures that Thread 1 finishes with both resources (rsrcA and rsrcB) before Thread 2 begins. This is achieved by using the 'safe' argument to set the 'safe' variable to 1.



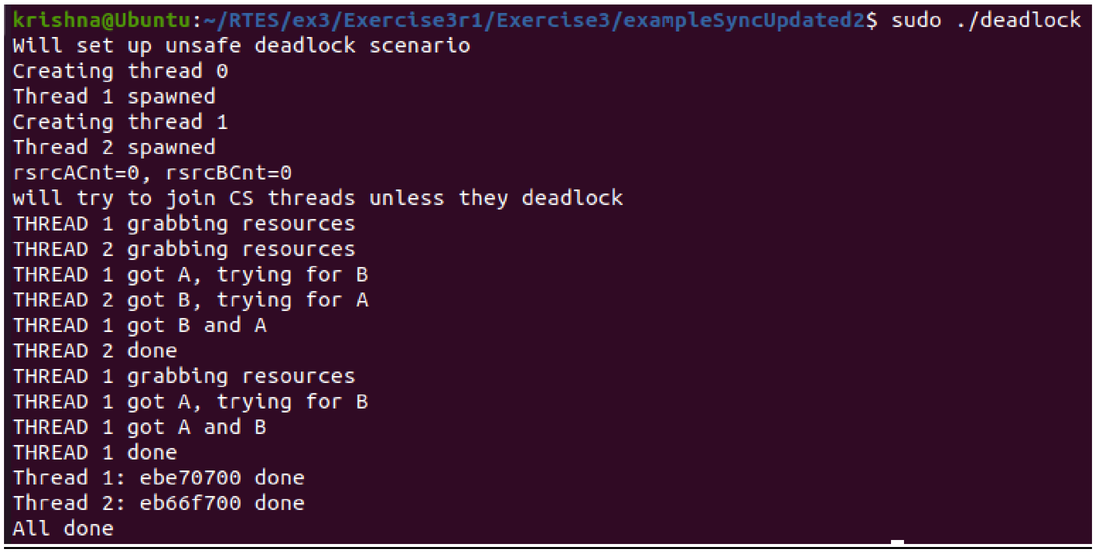
```
rtes@rtes-desktop:~/KS/exampleSyncUpdated2$ ./deadlock safe
Creating thread 0
Thread 1 spawned
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: b72b01f0 done
Creating thread 1
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: b72b01f0 done
All done
```

- Running the code with the 'race' argument allows for a race condition where the behavior of the program is unpredictable. Threads compete for resources without proper synchronization which can lead to the shared resource conflicts and varying results each time the program is executed. This scenario illustrates why proper synchronization mechanisms (like mutexes, semaphores, etc.) are crucial in multithreaded programming to avoid race conditions and ensure predictable behavior.



```
rtes@rtes-desktop:~/KS/exampleSyncUpdated2$ ./deadlock race
Creating thread 0
Thread 1 spawned
Creating thread 1
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 1: b1df11f0 done
Thread 2: b15f01f0 done
All done
```

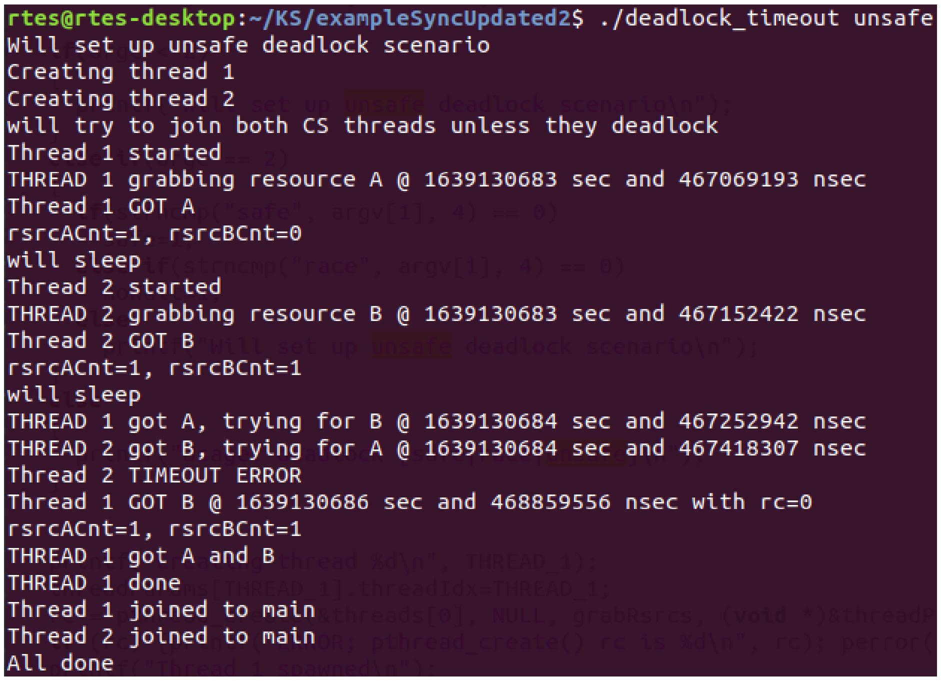
- The deadlock is removed by incorporating the random back-off scheme and the system behaves smoothly as demonstrated in the above screenshot.



```
krishna@Ubuntu:~/RTES/ex3/Exercise3r1/Exercise3/exampleSyncUpdated2$ sudo ./deadlock
Will set up unsafe deadlock scenario
Creating thread 0
Thread 1 spawned
Creating thread 1
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 1 grabbing resources
THREAD 2 grabbing resources
THREAD 1 got A, trying for B
THREAD 2 got B, trying for A
THREAD 1 got B and A
THREAD 2 done
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: ebe70700 done
Thread 2: eb66f700 done
All done
```

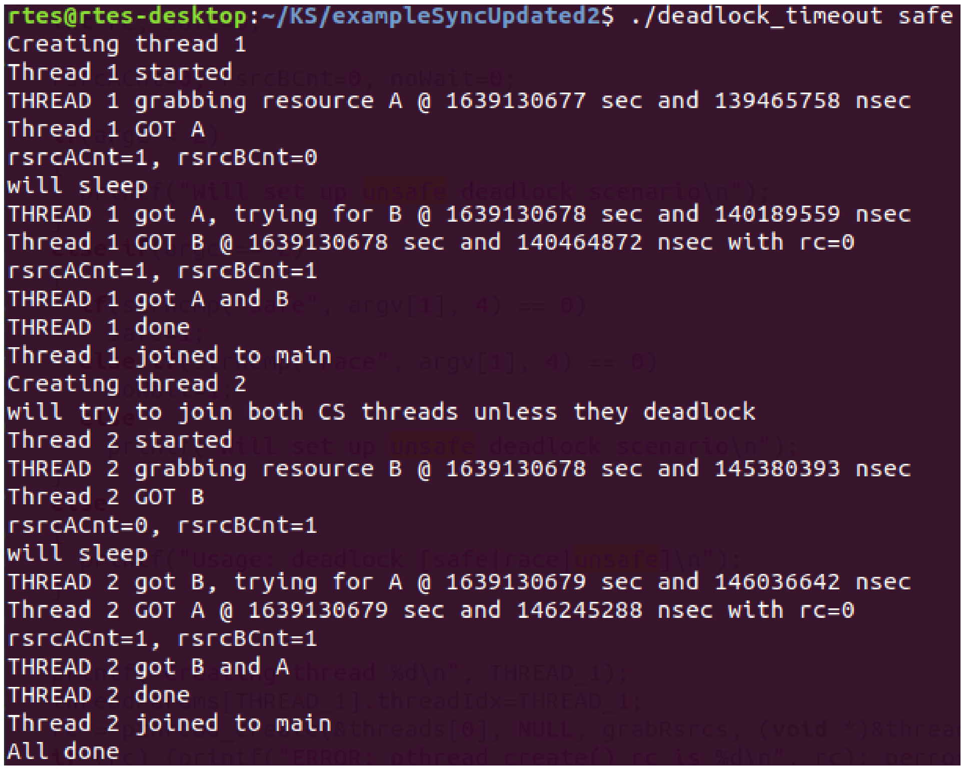
deadlock_timeout.c:

- Unlike the 'unsafe' scenario without timeouts, the code with timeouts prevents a full deadlock by allowingthreads to exit if they cannot acquire all resources within their specified timeouts.



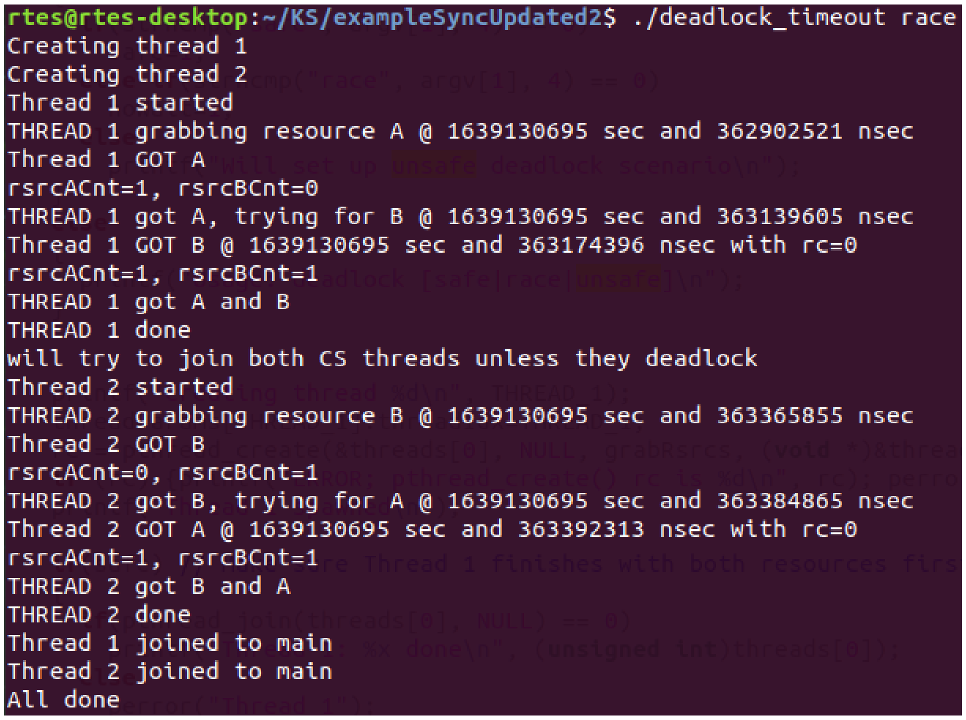
```
rtes@rtes-desktop:~/KS/exampleSyncUpdated2$ ./deadlock_timeout unsafe
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 1 started
THREAD 1 grabbing resource A @ 1639130683 sec and 467069193 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
will sleep
Thread 2 started
THREAD 2 grabbing resource B @ 1639130683 sec and 467152422 nsec
Thread 2 GOT B
rsrcACnt=1, rsrcBCnt=1
will sleep
THREAD 1 got A, trying for B @ 1639130684 sec and 467252942 nsec
THREAD 2 got B, trying for A @ 1639130684 sec and 467418307 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1639130686 sec and 468859556 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
```

- ‘safe’: THREAD_1 completes before THREAD_2 starts, ensuring both threads do not conflict.



```
rtes@rtes-desktop:~/KS/exampleSyncUpdated2$ ./deadlock_timeout safe
Creating thread 1
Thread 1 started
THREAD 1 grabbing resource A @ 1639130677 sec and 139465758 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
will sleep
THREAD 1 got A, trying for B @ 1639130678 sec and 140189559 nsec
Thread 1 GOT B @ 1639130678 sec and 140464872 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
Thread 1 joined to main
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1639130678 sec and 145380393 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
will sleep
THREAD 2 got B, trying for A @ 1639130679 sec and 146036642 nsec
Thread 2 GOT A @ 1639130679 sec and 146245288 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B and A
THREAD 2 done
Thread 2 joined to main
All done
```

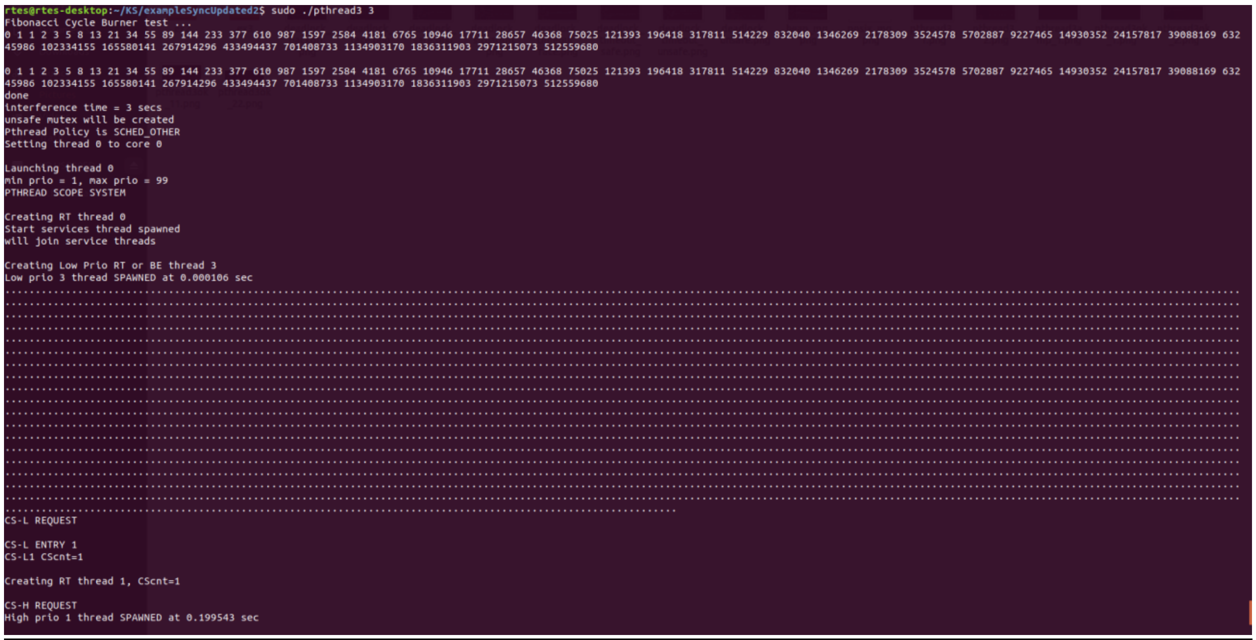
- ‘race’: The timeout mechanism helps prevent indefinite waiting, but it does not guarantee safe execution in a scenario where threads are racing to acquire resources.



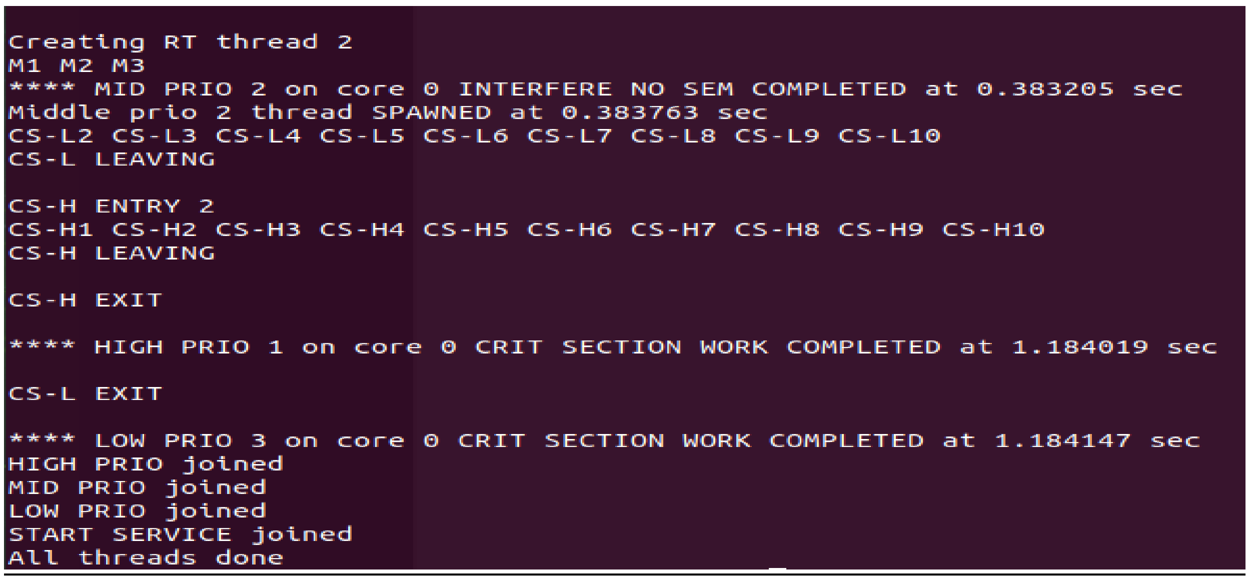
```
rtes@rtes-desktop:~/KS/exampleSyncUpdated2$ ./deadlock_timeout race
Creating thread 1
Creating thread 2
Thread 1 started
THREAD 1 grabbing resource A @ 1639130695 sec and 362902521 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
THREAD 1 got A, trying for B @ 1639130695 sec and 363139605 nsec
Thread 1 GOT B @ 1639130695 sec and 363174396 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1639130695 sec and 363365855 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1639130695 sec and 363384865 nsec
Thread 2 GOT A @ 1639130695 sec and 363392313 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B and A
THREAD 2 done
Thread 1 joined to main
Thread 2 joined to main
All done
```

pthread3.c:

- The priority inversion is explained in question 1. The following example code results in priority inversion:
    - Both LOW_PRIO_SERVICE and HIGH_PRIO_SERVICE acquire the sharedMemSem mutex before accessing the critical section. So, if HIGH_PRIO_SERVICE tries to acquire the mutex while LOW_PRIO_SERVICE holds it, HIGH_PRIO_SERVICE will be blocked. This creates a situation where a high-priority task is waiting for a low-priority task to release a mutex, resulting in priority inversion.
    - MID_PRIO_SERVICE performs unbounded computation depending upon the interference time passed in the second argument, which can delay LOW_PRIO_SERVICE from releasing the mutex. This increases the time HIGH_PRIO_SERVICE is blocked, leading to unbounded priority inversion.



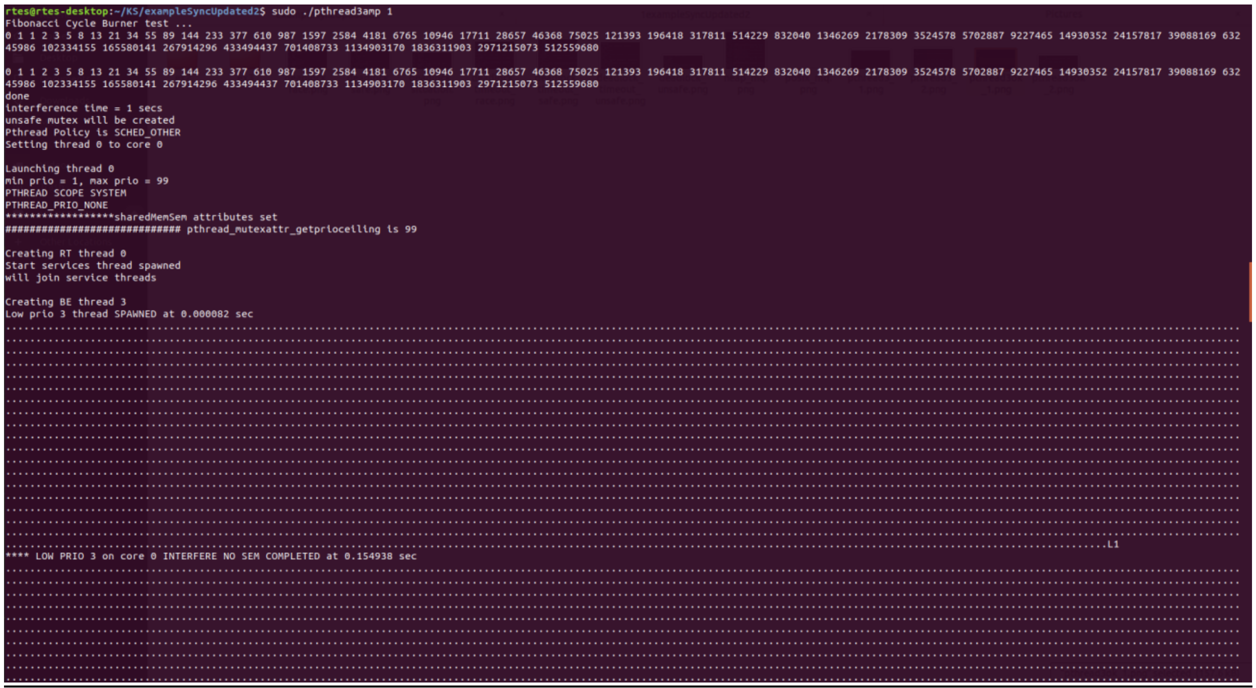
```
rtes@rtes-desktop:~/KS/exampleSyncUpdated2$ sudo ./pthread3 3
Fibonacci Cycle Burner test ...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 632
45986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 512559680

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 632
45986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 512559680
done
interference time = 3 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Setting thread 0 to core 0

Launching thread 0
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM

Creating RT thread 0
Start services thread spawned
will join service threads

Creating Low Prio RT or BE thread 3
Low prio 3 thread SPAWNED at 0.000106 sec
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
..............................................................................................................
CS-L REQUEST

CS-L ENTRY 1
CS-L1 CScnt=1

Creating RT thread 1, CScnt=1

CS-H REQUEST
High prio 1 thread SPAWNED at 0.199543 sec
```



```
Creating RT thread 2
M1 M2 M3
**** MID PRIO 2 on core 0 INTERFERE NO SEM COMPLETED at 0.383205 sec
Middle prio 2 thread SPAWNED at 0.383763 sec
CS-L2 CS-L3 CS-L4 CS-L5 CS-L6 CS-L7 CS-L8 CS-L9 CS-L10
CS-L LEAVING

CS-H ENTRY 2
CS-H1 CS-H2 CS-H3 CS-H4 CS-H5 CS-H6 CS-H7 CS-H8 CS-H9 CS-H10
CS-H LEAVING

CS-H EXIT

**** HIGH PRIO 1 on core 0 CRIT SECTION WORK COMPLETED at 1.184019 sec

CS-L EXIT

**** LOW PRIO 3 on core 0 CRIT SECTION WORK COMPLETED at 1.184147 sec
HIGH PRIO joined
MID PRIO joined
LOW PRIO joined
START SERVICE joined
All threads done
```

pthread3amp.c:

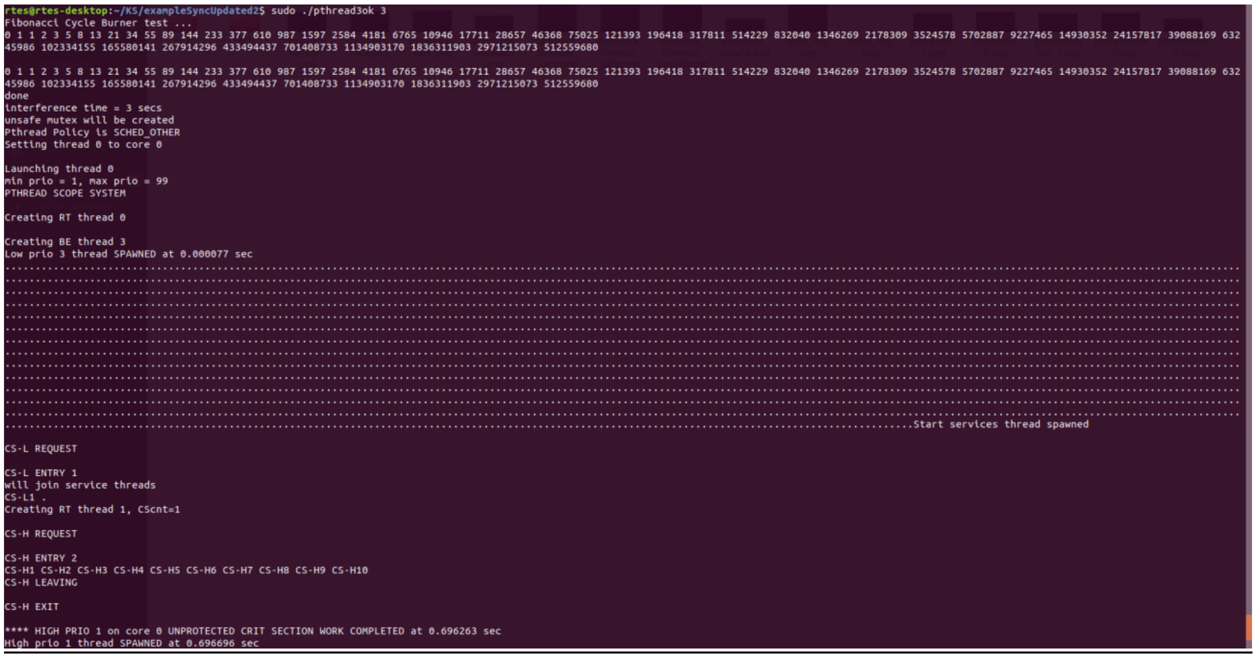
- The code does not create the HIGH_PRIO_SERVICE and MID_PRIO_SERVICE threads after the LOW_PRIO_SERVICE thread. So, LOW_PRIO_SERVICE thread is the only one running, and there is no actual priority inversion happening.



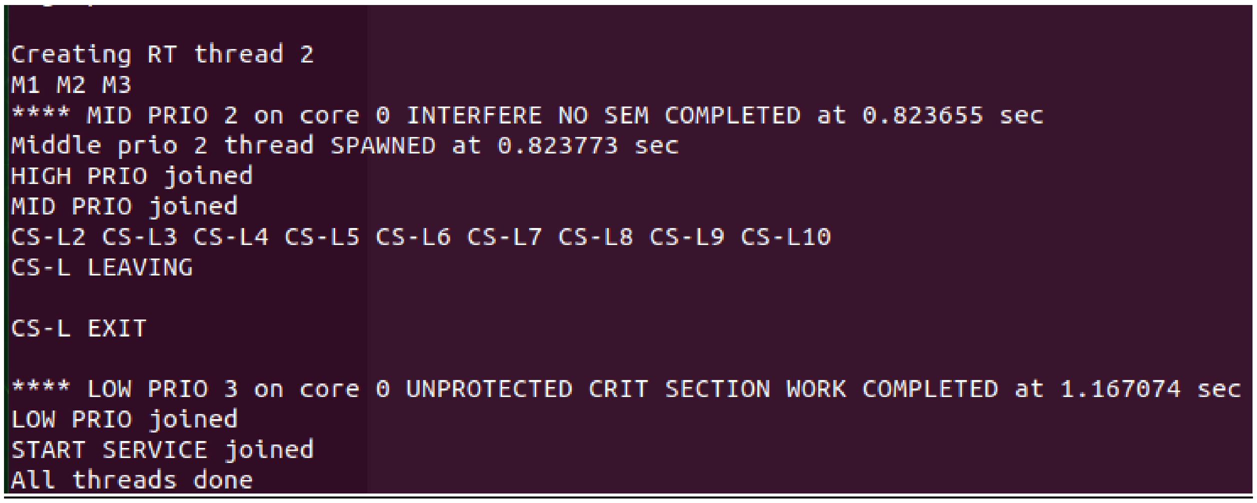
```
rtes@rtes-desktop:~/KS/exampleSyncUpdated2$ sudo ./pthread3amp 1
Fibonacci Cycle Burner test ...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 632
45986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 512559680

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 632
45986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 512559680
done
interference time = 1 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Setting thread 0 to core 0

Launching thread 0
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
PTHREAD_PRIO_NONE
*****************sharedMemSem attributes set
############################# pthread_mutexattr_getprioceiling is 99

Creating RT thread 0
Start services thread spawned
will join service threads

Creating BE thread 3
Low prio 3 thread SPAWNED at 0.000082 sec
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
..........................................................................................................................................................................................L1
**** LOW PRIO 3 on core 0 INTERFERE NO SEM COMPLETED at 0.154938 sec
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
........................................................................................................................................................................................................
```

pthread3ok.c:

- The code results in unbounded priority inversion due to the continuous interference from MID_PRIO_SERVICE, which prevents LOW_PRIO_SERVICE from completing its critical section.
- Implementing a proper synchronization mechanism or changing the thread priorities could resolve this issue.



```
rtes@rtes-desktop:~/KS/exampleSyncUpdated2$ sudo ./pthread3ok 3
Fibonacci Cycle Burner test ...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 632
45986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 512559680

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 632
45986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 512559680
done
interference time = 3 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Setting thread 0 to core 0

Launching thread 0
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM

Creating RT thread 0

Creating BE thread 3
Low prio 3 thread SPAWNED at 0.000077 sec
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
..............................................................................................................................................................................................................
.........................................................................................................................................................Start services thread spawned

CS-L REQUEST

CS-L ENTRY 1
will join service threads
CS-L1 .
Creating RT thread 1, CScnt=1

CS-H REQUEST

CS-H ENTRY 2
CS-H1 CS-H2 CS-H3 CS-H4 CS-H5 CS-H6 CS-H7 CS-H8 CS-H9 CS-H10
CS-H LEAVING

CS-H EXIT

**** HIGH PRIO 1 on core 0 UNPROTECTED CRIT SECTION WORK COMPLETED at 0.696263 sec
High prio 1 thread SPAWNED at 0.696696 sec
```



```
Creating RT thread 2
M1 M2 M3
**** MID PRIO 2 on core 0 INTERFERE NO SEM COMPLETED at 0.823655 sec
Middle prio 2 thread SPAWNED at 0.823773 sec
HIGH PRIO joined
MID PRIO joined
CS-L2 CS-L3 CS-L4 CS-L5 CS-L6 CS-L7 CS-L8 CS-L9 CS-L10
CS-L LEAVING

CS-L EXIT

**** LOW PRIO 3 on core 0 UNPROTECTED CRIT SECTION WORK COMPLETED at 1.167074 sec
LOW PRIO joined
START SERVICE joined
All threads done
```

b) Fix the deadlock so that it does not occur by using a random back-off scheme to resolve. For the unbounded inversion, is there a real fix in Linux – if not, why not?

Answer: The deadlock scenario is resolved by employing 'pthread_mutex_trylock' to check if the mutex is already in use. If it is, the current thread releases the mutex it acquired and enters a sleep state for a random duration. A random timeout value is generated using a random number generator. Meanwhile, the other thread can proceed with necessary operations. When the sleeping thread attempts to access the shared resource again, it finds it available for use.

There is RT_PREEMPT improvement patch in the linux kernel that helps with the unbounded priority inversion but the patch is not a complete solution to fix the problem.

c) What about a patch for the Linux kernel? For example, Linux Kernel.org recommends the

RT_PREEMPT Patch, also discussed by the Linux Foundation Realtime Start and this blog, but would this really help? Read about the patch and describe why think it would or would not help with unbounded priority inversion. Based on inversion, does it make sense to simply switch to an RTOS and not use Linux at all for both HRT and SRT services?

Answer: The RT_PREEMPT patch is a significant improvement for the Linux kernel, designed to convert it into a real-time operating system suitable for embedded development. The patch addresses unbounded priority inversion by enhancing the kernel's responsiveness and reducing latencies. It achieves that by preempting non-real-time tasks in favor of real-time ones and minimizing interrupt and scheduling latencies, making task execution more predictable and efficient.

- For soft real-time services, the RT_PREEMPT patch often provides adequate real-time capabilities without needing to switch to a dedicated real-time operating system. It offers a balance between real-time performance and the flexibility and features of Linux, making it a practical choice for many embedded applications. However, in cases where hard real-time requirements are particularly strict, such as in safety-critical systems, specialized RTOS solutions can be beneficial. However, RT_PREEMPT patch can offer a level of soft real-time capability suitable for various applications.
- The decision to use the RT_PREEMPT patch or opt for an RTOS depends on factors like the specific needs of the embedded system and the trade-offs between using a general-purpose operating system like Linux and a specialized RTOS. Despite its strengths, the patch may not completely eliminate unbounded priority inversion, but it significantly reduces its impact, making it a valuable tool for achieving real-time capabilities with Linux.

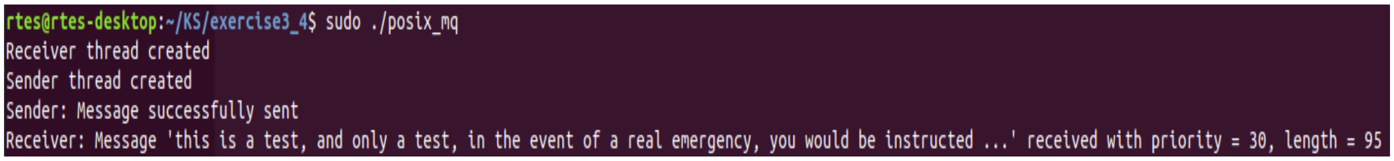
Question 4

[15 points] Review POSIX-examples and especially POSIX_MQ_LOOP and build the code related to POSIX message queues and run them to learn about basic use.
a) First, re-write the simple message queue demonstration code in heap_mq.c and posix_mq.c so that it uses RT-Linux Pthreads (FIFO) instead of SCHEDULE_OTHER, and then write a brief paragraph describing how the two message queue applications are similar and how they are different. Prove that you got the POSIX message queue features working in Linux on your target board.

Answer: The heap_mq.c and posix_mq.c are ported to the POSIX platform using the supported APIs and setting the FIFO scheduling. The priorities are reversed while porting because in VxWorks, the high priority number corresponds to the lowest priority number and vice versa.

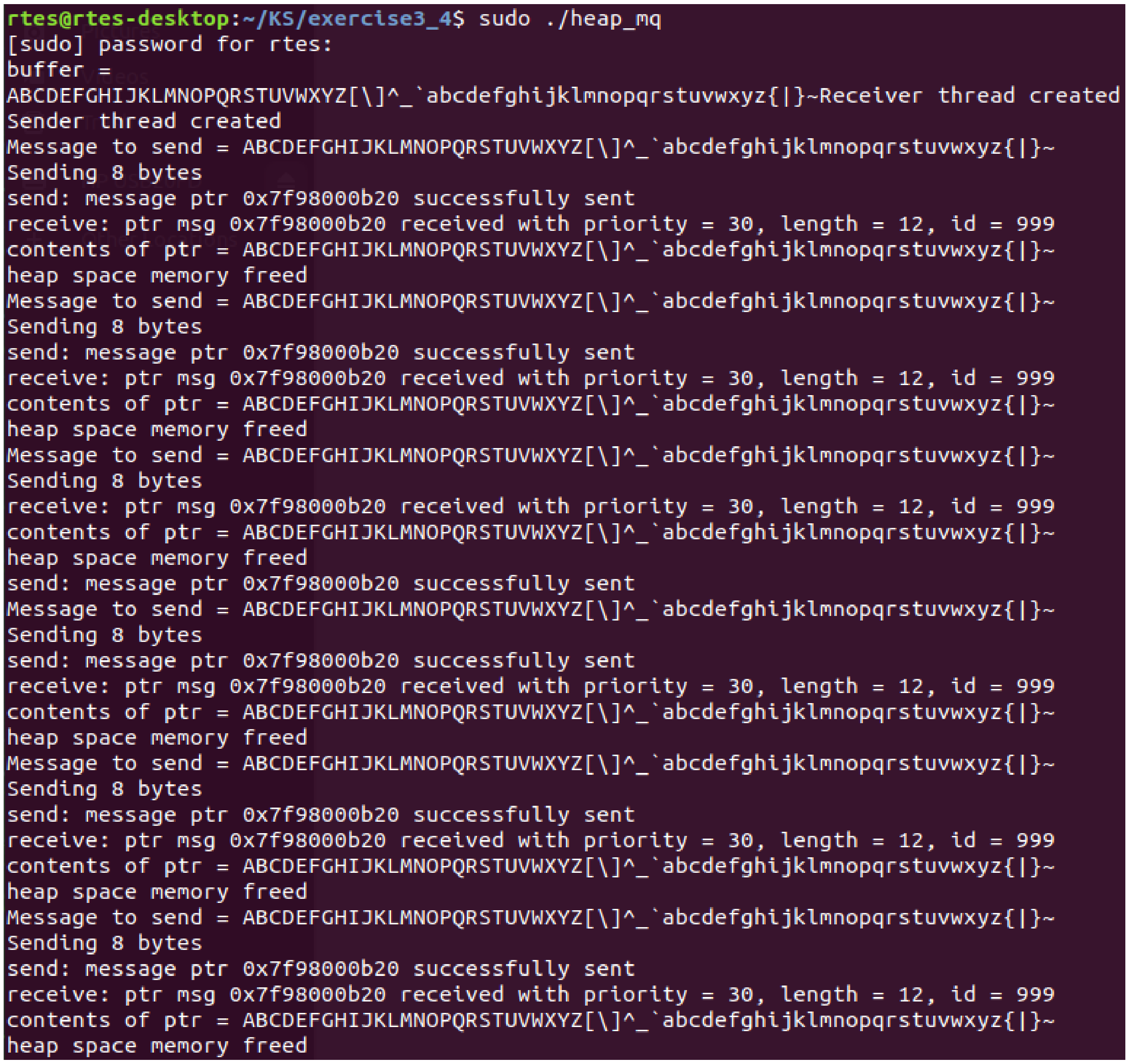
Following are the similarities and differences of the two message queue applications:

- Similarities:
  - Both codes are examples of POSIX message queue implementations.
  - They both demonstrate the message send and receive functionality via a message queue, though the data format differs.
- Differences:
  - Code 1 is a simple demonstration involving sending and receiving string messages whereas code 2 presents a more sophisticated approach.
  - In Code 2, dynamic memory allocation is utilized for creating a buffer (buffptr) to send more complex data structures like an image buffer (imagebuff). It continuously sends messages every 3 seconds, contrasting with Code 1's single message sending behavior.
  - Additionally, Code 2 demonstrates passing and receiving pointers through the message queue, extracting an ID and the buffer itself in the receiver. It also allocates the memory dynamically and ensures proper resource management by freeing allocated memory after message processing.
  - Overall, Code 2 exhibits a more advanced and structured approach to message passing, making it suitable for scenarios involving continuous data transmission and complex data structures.
- posix_mq output:



```
rtes@rtes-desktop:~/KS/exercise3_4$ sudo ./posix_mq
Receiver thread created
Sender thread created
Sender: Message successfully sent
Receiver: Message 'this is a test, and only a test, in the event of a real emergency, you would be instructed ...' received with priority = 30, length = 95
```

- heap_mq output:



```
rtes@rtes-desktop:~/KS/exercise3_4$ sudo ./heap_mq
[sudo] password for rtes:
buffer =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~Receiver thread created
Sender thread created
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0x7f98000b20 successfully sent
receive: ptr msg 0x7f98000b20 received with priority = 30, length = 12, id = 999
contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0x7f98000b20 successfully sent
receive: ptr msg 0x7f98000b20 received with priority = 30, length = 12, id = 999
contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
receive: ptr msg 0x7f98000b20 received with priority = 30, length = 12, id = 999
contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
send: message ptr 0x7f98000b20 successfully sent
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0x7f98000b20 successfully sent
receive: ptr msg 0x7f98000b20 received with priority = 30, length = 12, id = 999
contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0x7f98000b20 successfully sent
receive: ptr msg 0x7f98000b20 received with priority = 30, length = 12, id = 999
contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0x7f98000b20 successfully sent
receive: ptr msg 0x7f98000b20 received with priority = 30, length = 12, id = 999
contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
```

c) Message queues are often suggested as a way to avoid MUTEX priority inversion. Would you agree that this really circumvents the problem of unbounded inversion? If so why, if not, why not?

Answer: Message queues can help avoid mutex priority inversion in certain situations by allowing tasks to communicate without directly sharing resources.

- Instead of directly locking shared resources with a mutex, tasks can send messages to each other through a queue. This way, a high-priority task can send a message to a low-priority task to request data or actions, without needing to wait on a mutex held by the low-priority task.
- Message queues are a useful tool to help mitigate mutex priority inversion as it allows direct communication between the tasks without locking shared resources.
- Since the tasks do not share the resources, the chances of priority inversion get reduced.
- However, in case there are other external shared resources, message queues do not entirely fix the problem and implementation becomes complex.

Question 5

[20 points] Watchdog timers, timeouts and timer services – First, read this overview of the Linux Watchdog Daemon and the Linux manual page on the watchdog daemon - https://linux.die.net/man/8/watchdog . Also see the Watchdog Explained.

a) Describe how it might be used if software caused an indefinite deadlock.

Answer: The Linux Watchdog timer is employed as a safeguard mechanism against software-induced indefinite deadlocks. In a scenario where software causes an indefinite deadlock, the Watchdog timer is crucial for preventing the entire system from becoming unresponsive. Here's how the Linux WD timer might be used in such a situation.

Watchdog Initialization: The system is equipped with a hardware Watchdog timer and a corresponding software Watchdog Daemon. The Watchdog Daemon is configured to communicate with the Watchdog timer, often through a device file like /dev/watchdog.

Setting Timeout Period: The Watchdog Daemon is configured with a timeout period (watchdog-timeout), representing the duration within which it expects to receive a signal or refresh from the software.

Regular Software Refresh: The software, which might be susceptible to indefinite deadlocks, periodically sends signals or refreshes to the Watchdog Daemon. These signals or refreshes serve as indications that the software is still functioning correctly.

Monitoring Periodic Refresh: The Watchdog Daemon continually monitors the receipt of signals or refreshes within the configured timeout period. If the daemon receives the expected signals or refreshes within the defined interval, it assumes that the software is operating normally.

Detecting Deadlock: In the event of a software-induced indefinite deadlock, the software fails to send the expected signals or refreshes to the Watchdog Daemon.

Timeout Expiry: When the configured timeout period elapses without receiving the anticipated signals or refreshes, the Watchdog Daemon considers it an indication of a deadlock or software failure.

Watchdog Triggering Reset: In response to the lack of activity within the specified time frame, the Watchdog Daemon triggers the Watchdog timer. The Watchdog timer, upon expiration, initiates a hardware reset of the entire system.

System Reboot: The hardware reset caused by the Watchdog timer leads to a controlled reboot of the system. This reboot provides a fresh start, terminating the deadlocked software and allowing the system to recover.

Preventing Prolonged Unresponsiveness: By cyclically refreshing the Watchdog timer, the Watchdog Daemon ensures that the system remains responsive. The periodic hardware resets, triggered by the Watchdog timer, prevent prolonged unresponsiveness caused by software failures or deadlocks.

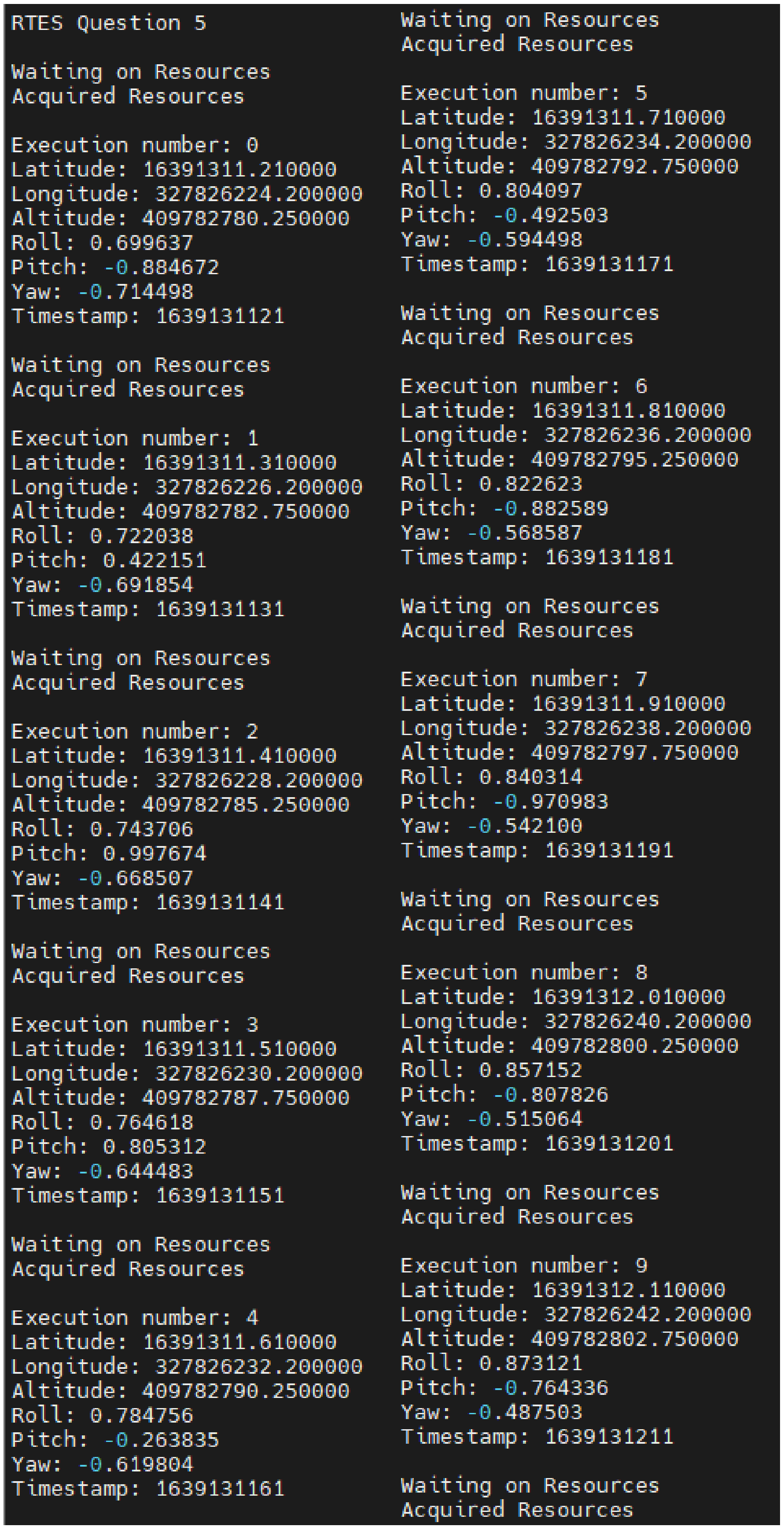
Conclusion: The Linux Watchdog timer, when integrated with the Watchdog Daemon, serves as a critical component for handling software-induced indefinite deadlocks. It acts as a fail-safe mechanism, initiating controlled reboots to prevent prolonged system unresponsiveness and facilitate system recovery.

b) Next, to explore timeouts, use your code from #2 and create a thread that waits on a MUTEX semaphore for up to 10 seconds and then un-blocks and prints out "No new data available at " and then loops back to

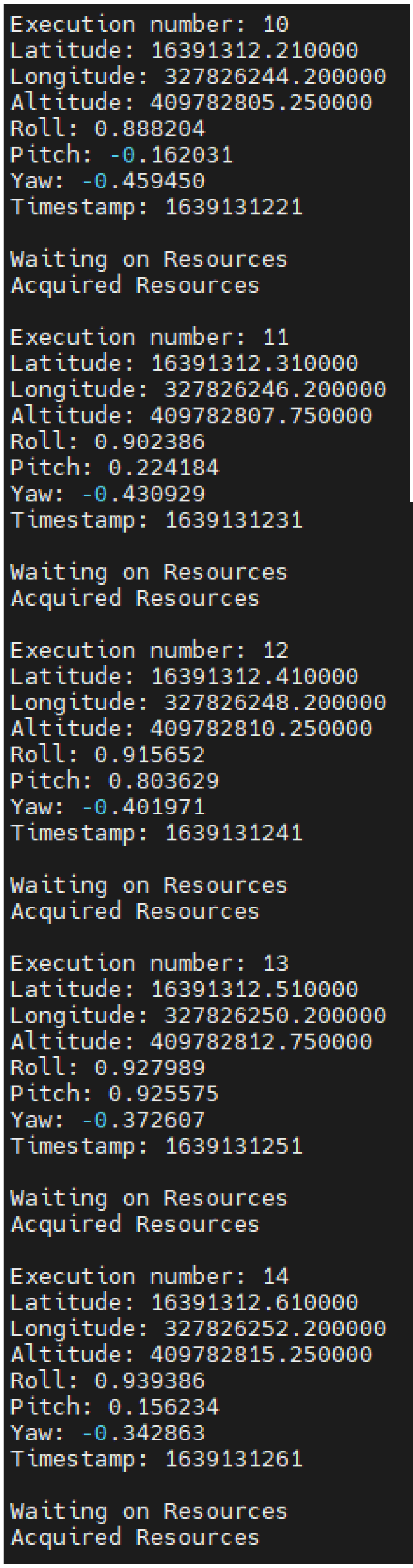
wait for a data update again. Use a variant of the pthread_mutex_lock called pthread_mutex_timedlock to solve this programming problem.

Answer:

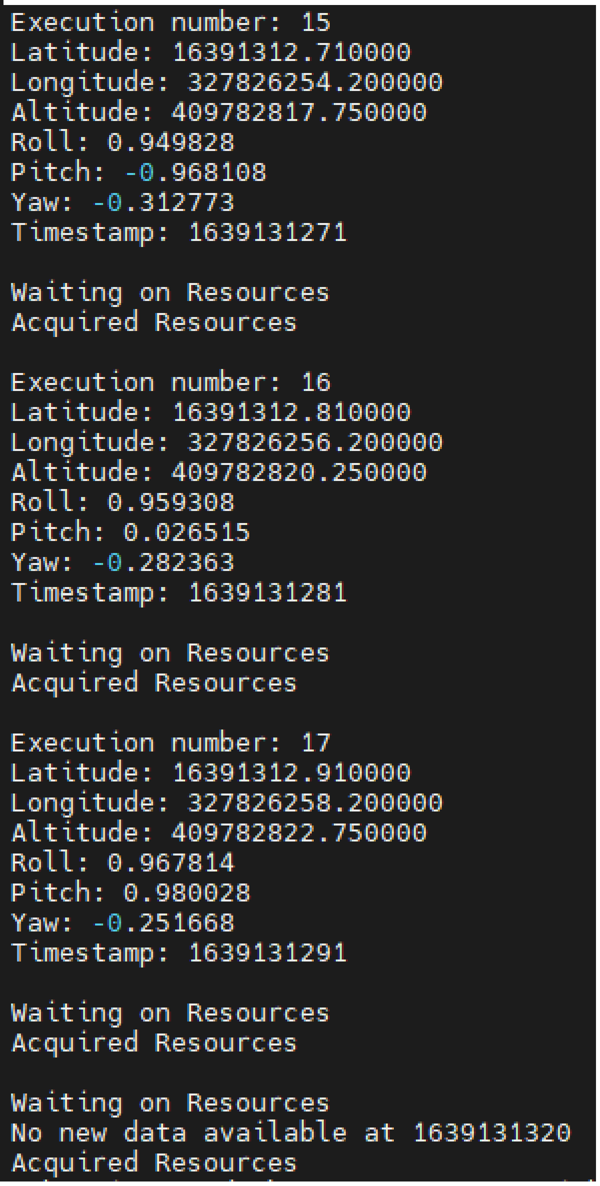
Output: The code from the Q2 solution has been modified to incorporate a global variable for nav_state and an additional thread dedicated to timeout functionality. The update_nav_state function is altered to simulate a scenario where a mutex is locked but not released, triggering a failure case in timeout_thread. The timeout_thread waits for the resource, sleeps for 10 seconds after acquiring it, and in case the timeout value is exceeded, prints a message stating 'no new data available at <timestamp>' and unlocks the mutex that was locked in the update function.



```
RTES Question 5

Waiting on Resources
Acquired Resources

Execution number: 0
Latitude: 16391311.210000
Longitude: 327826224.200000
Altitude: 409782780.250000
Roll: 0.699637
Pitch: -0.884672
Yaw: -0.714498
Timestamp: 1639131121

Waiting on Resources
Acquired Resources

Execution number: 1
Latitude: 16391311.310000
Longitude: 327826226.200000
Altitude: 409782782.750000
Roll: 0.722038
Pitch: 0.422151
Yaw: -0.691854
Timestamp: 1639131131

Waiting on Resources
Acquired Resources

Execution number: 2
Latitude: 16391311.410000
Longitude: 327826228.200000
Altitude: 409782785.250000
Roll: 0.743706
Pitch: 0.997674
Yaw: -0.668507
Timestamp: 1639131141

Waiting on Resources
Acquired Resources

Execution number: 3
Latitude: 16391311.510000
Longitude: 327826230.200000
Altitude: 409782787.750000
Roll: 0.764618
Pitch: 0.805312
Yaw: -0.644483
Timestamp: 1639131151

Waiting on Resources
Acquired Resources

Execution number: 4
Latitude: 16391311.610000
Longitude: 327826232.200000
Altitude: 409782790.250000
Roll: 0.784756
Pitch: -0.263835
Yaw: -0.619804
Timestamp: 1639131161
Waiting on Resources
Acquired Resources

Execution number: 5
Latitude: 16391311.710000
Longitude: 327826234.200000
Altitude: 409782792.750000
Roll: 0.804097
Pitch: -0.492503
Yaw: -0.594498
Timestamp: 1639131171

Waiting on Resources
Acquired Resources

Execution number: 6
Latitude: 16391311.810000
Longitude: 327826236.200000
Altitude: 409782795.250000
Roll: 0.822623
Pitch: -0.882589
Yaw: -0.568587
Timestamp: 1639131181

Waiting on Resources
Acquired Resources

Execution number: 7
Latitude: 16391311.910000
Longitude: 327826238.200000
Altitude: 409782797.750000
Roll: 0.840314
Pitch: -0.970983
Yaw: -0.542100
Timestamp: 1639131191

Waiting on Resources
Acquired Resources

Execution number: 8
Latitude: 16391312.010000
Longitude: 327826240.200000
Altitude: 409782800.250000
Roll: 0.857152
Pitch: -0.807826
Yaw: -0.515064
Timestamp: 1639131201

Waiting on Resources
Acquired Resources

Execution number: 9
Latitude: 16391312.110000
Longitude: 327826242.200000
Altitude: 409782802.750000
Roll: 0.873121
Pitch: -0.764336
Yaw: -0.487503
Timestamp: 1639131211

Waiting on Resources
Acquired Resources
```



```
Execution number: 10
Latitude: 16391312.210000
Longitude: 327826244.200000
Altitude: 409782805.250000
Roll: 0.888204
Pitch: -0.162031
Yaw: -0.459450
Timestamp: 1639131221

Waiting on Resources
Acquired Resources

Execution number: 11
Latitude: 16391312.310000
Longitude: 327826246.200000
Altitude: 409782807.750000
Roll: 0.902386
Pitch: 0.224184
Yaw: -0.430929
Timestamp: 1639131231

Waiting on Resources
Acquired Resources

Execution number: 12
Latitude: 16391312.410000
Longitude: 327826248.200000
Altitude: 409782810.250000
Roll: 0.915652
Pitch: 0.803629
Yaw: -0.401971
Timestamp: 1639131241

Waiting on Resources
Acquired Resources

Execution number: 13
Latitude: 16391312.510000
Longitude: 327826250.200000
Altitude: 409782812.750000
Roll: 0.927989
Pitch: 0.925575
Yaw: -0.372607
Timestamp: 1639131251

Waiting on Resources
Acquired Resources

Execution number: 14
Latitude: 16391312.610000
Longitude: 327826252.200000
Altitude: 409782815.250000
Roll: 0.939386
Pitch: 0.156234
Yaw: -0.342863
Timestamp: 1639131261

Waiting on Resources
Acquired Resources
```



```
Execution number: 15
Latitude: 16391312.710000
Longitude: 327826254.200000
Altitude: 409782817.750000
Roll: 0.949828
Pitch: -0.968108
Yaw: -0.312773
Timestamp: 1639131271

Waiting on Resources
Acquired Resources

Execution number: 16
Latitude: 16391312.810000
Longitude: 327826256.200000
Altitude: 409782820.250000
Roll: 0.959308
Pitch: 0.026515
Yaw: -0.282363
Timestamp: 1639131281

Waiting on Resources
Acquired Resources

Execution number: 17
Latitude: 16391312.910000
Longitude: 327826258.200000
Altitude: 409782822.750000
Roll: 0.967814
Pitch: 0.980028
Yaw: -0.251668
Timestamp: 1639131291

Waiting on Resources
Acquired Resources

Waiting on Resources
No new data available at 1639131320
Acquired Resources
```

Appendix

****

Question 2 code

```
/*
 * File: Q2.c
 * Author: Suhas Reddy and Krishna Suhagiya
 * Description: This program demonstrates a multi-threaded system for updating
 *              and reading navigation state data. It includes two threads:
 *              one for updating navigation state, and one for reading the state.
 *              The update thread updates the navigation state variables periodically,
 *              the read thread reads and prints the state.
 *              The program utilizes POSIX threads and synchronization mechanisms
 *              such as mutexes and condition variables.
 * Date: 9th March 2023
 */

#include <pthread.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <stdbool.h>

#define PI 3.14
#define NUM_THREADS 2

bool run_complete = false;
pthread_cond_t signal_read = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

typedef struct {
    double Latitude;
    double Longitude;
    double Altitude;
    double Roll;
    double Pitch;
    double Yaw;
    struct timespec timestamp;
} nav_state;

typedef struct {
    int thread_idx;
    nav_state *state;
} thread_param;

void *update_nav_state(void *threadp) {
    thread_param *tp = (thread_param *)threadp;

    while (!run_complete) {
        pthread_mutex_lock(&mutex);
        nav_state *state = tp->state;
        clock_gettime(CLOCK_REALTIME, &state->timestamp);
        state->Latitude = 0.01 * (state->timestamp.tv_sec);
        state->Longitude = 0.2 * (state->timestamp.tv_sec);
        state->Altitude = 0.25 * (state->timestamp.tv_sec);
        state->Roll = sin(2 * PI * (state->timestamp.tv_sec));
        state->Pitch = cos(2 * PI * (state->timestamp.tv_sec) * (state->timestamp.tv_sec));
        state->Yaw = cos(2 * PI * (state->timestamp.tv_sec));
        pthread_cond_signal(&signal_read);     // Signal read function when update is complete
        pthread_mutex_unlock(&mutex);
        sleep(1); // Update rate of 1 Hz
    }

    return NULL;
}

void *read_nav_state(void *threadp) {
    thread_param *tp = (thread_param *)threadp;

    for (int i = 0; i < 18; i++) {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&signal_read, &mutex);  // Wait until update is complete
        nav_state *state = tp->state;
        printf("\nRead Thread Execution Number: %d", i);
        printf("\nLatitude: %lf", state->Latitude);
        printf("\nLongitude: %lf", state->Longitude);
        printf("\nAltitude: %lf", state->Altitude);
        printf("\nRoll: %lf", state->Roll);
        printf("\nPitch: %lf", state->Pitch);
        printf("\nYaw: %lf", state->Yaw);
        printf("\nTimestamp: %lu\n", state->timestamp.tv_sec);
        pthread_mutex_unlock(&mutex);
        sleep(10); // Read rate of 0.1 Hz
    }
    run_complete = true;
    return NULL;
}

int main() {
    printf("RTES Question 2:\n");

    pthread_t threads[NUM_THREADS];
    struct timespec timestamp;
    clock_gettime(CLOCK_REALTIME, &timestamp);
    nav_state state = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, timestamp};

    thread_param thread0 = {0, &state}, thread1 = {1, &state};

    pthread_create(&threads[0], NULL, update_nav_state, (void *)&thread0);
    pthread_create(&threads[1], NULL, read_nav_state, (void *)&thread1);

    // Wait for threads to finish
    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);

    pthread_mutex_destroy(&mutex);

    return 0;
}
```

****

Question 3 code

```
/*
 * File: deadlock.c
 * Author: Krishna Suhagiya and Suhas Reddy
 * Description: This file is modified to fix the deadlock in 'unsafe' option of the 'deadlock' application.
 * Date: 9th March 2023
 */

#include <pthread.h>
#include <stdio.h>
#include <sched.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdbool.h>
#include <errno.h>

#define NUM_THREADS 2
#define THREAD_1 0
#define THREAD_2 1

typedef struct
{
    int threadIdx;
} threadParams_t;


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

struct sched_param nrt_param;

// On the Raspberry Pi, the MUTEX semaphores must be statically initialized
//
// This works on all Linux platforms, but dynamic initialization does not work
// on the R-Pi in particular as of June 2020.
//
pthread_mutex_t rsrcA = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rsrcB = PTHREAD_MUTEX_INITIALIZER;
```

```
volatile int rsrcACnt=0, rsrcBCnt=0, noWait=0;


void *grabRsrcs(void *threadp)
{
   threadParams_t *threadParams = (threadParams_t *)threadp;
   int threadIdx = threadParams->threadIdx;

    while(1){
       if(threadIdx == THREAD_1)
       {
         printf("THREAD 1 grabbing resources\n");
         pthread_mutex_lock(&rsrcA);

         rsrcACnt++;
         if(!noWait)
          sleep(1);

          printf("THREAD 1 got A, trying for B\n");
          if(pthread_mutex_trylock(&rsrcB) == EBUSY)  // Retry in while(1) loop if the mutex is busy
          {
            rsrcACnt--;
            pthread_mutex_unlock(&rsrcA);
            int backoff_time = rand() % 1000;
            usleep(backoff_time*1000);
          }
          else
          {
            rsrcBCnt++;
            printf("THREAD 1 got A and B\n");
            pthread_mutex_unlock(&rsrcB);
            pthread_mutex_unlock(&rsrcA);

            printf("THREAD 1 done\n");
            break;
          }
        }
        else
        {
```

```
        printf("THREAD 2 grabbing resources\n");
        pthread_mutex_lock(&rsrcB);
        rsrcBCnt++;
        if(!noWait)
        sleep(1);

        printf("THREAD 2 got B, trying for A\n");
        if(pthread_mutex_trylock(&rsrcA) == EBUSY)  // Retry in while(1) loop if the mutex is busy
        {
          rsrcBCnt--;
          pthread_mutex_unlock(&rsrcB);
          int backoff_time = rand() % 1000;
          usleep(backoff_time*1000);
        }
        else
        {
          rsrcACnt++;
          printf("THREAD 1 got B and A\n");
          pthread_mutex_unlock(&rsrcA);
          pthread_mutex_unlock(&rsrcB);

          printf("THREAD 2 done\n");
          break;
        }
      }
   }

  pthread_exit(NULL);
}


int main (int argc, char *argv[])
{
  int rc, safe=0;

  rsrcACnt=0, rsrcBCnt=0, noWait=0;

  if(argc < 2)
  {
    printf("Will set up unsafe deadlock scenario\n");
```

```
}
else if(argc == 2)
{
  if(strncmp("safe", argv[1], 4) == 0)
    safe=1;
  else if(strncmp("race", argv[1], 4) == 0)
    noWait=1;
  else
    printf("Will set up unsafe deadlock scenario\n");
}
else
{
  printf("Usage: deadlock [safe|race|unsafe]\n");
}

srand(time(NULL));

printf("Creating thread %d\n", THREAD_1);
threadParams[THREAD_1].threadIdx=THREAD_1;
rc = pthread_create(&threads[0], NULL, grabRsrcs, (void *)&threadParams[THREAD_1]);
if (rc) {printf("ERROR; pthread_create() rc is %d\n", rc); perror(NULL); exit(-1);}
printf("Thread 1 spawned\n");

if(safe) // Make sure Thread 1 finishes with both resources first
{
  if(pthread_join(threads[0], NULL) == 0)
    printf("Thread 1: %x done\n", (unsigned int)threads[0]);
  else
    perror("Thread 1");
}

printf("Creating thread %d\n", THREAD_2);
threadParams[THREAD_2].threadIdx=THREAD_2;
rc = pthread_create(&threads[1], NULL, grabRsrcs, (void *)&threadParams[THREAD_2]);
if (rc) {printf("ERROR; pthread_create() rc is %d\n", rc); perror(NULL); exit(-1);}
printf("Thread 2 spawned\n");

printf("rsrcACnt=%d, rsrcBCnt=%d\n", rsrcACnt, rsrcBCnt);
printf("will try to join CS threads unless they deadlock\n");
```

```
  if(!safe)
  {
    if(pthread_join(threads[0], NULL) == 0)
      printf("Thread 1: %x done\n", (unsigned int)threads[0]);
    else
      perror("Thread 1");
  }

  if(pthread_join(threads[1], NULL) == 0)
    printf("Thread 2: %x done\n", (unsigned int)threads[1]);
  else
    perror("Thread 2");

  if(pthread_mutex_destroy(&rsrcA) != 0)
    perror("mutex A destroy");

  if(pthread_mutex_destroy(&rsrcB) != 0)
    perror("mutex B destroy");

  printf("All done\n");

  exit(0);
}
```

```
***************************************************************************************
```

Question 4 code

heap_mq.c

```
/*
 * File: heap_mq.c
 * Author: Krishna Suhagiya and Suhas Reddy
 * Description: This file ports the provided VxWorks posix_mq.c implementation to POSIX with
SCHED_FIFO scheduling.
 * Date: 9th March 2023
 */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <mqueue.h>
#include <string.h>
#include <unistd.h>
#include <sched.h>

#define SNDRCV_MQ "/send_receive_mq"

struct mq_attr mq_attr;
mqd_t mymq;

void *receiver(void *arg)
{
    char buffer[sizeof(void *)+sizeof(int)];
    void *buffptr;
    int prio;
    int nbytes;
    int id;

    while (1)
    {
        /* Read oldest, highest priority msg from the message queue */
        if ((nbytes = mq_receive(mymq, buffer, (size_t)sizeof(void *)+sizeof(int), &prio)) == -1)
        {
            perror("mq_receive");
        }
        else
        {
            buffer[nbytes] = '\0';
            memcpy(&buffptr, buffer, sizeof(void *));
            memcpy((void *)&id, &(buffer[sizeof(void *)]), sizeof(int));
            printf("receive: ptr msg %p received with priority = %d, length = %d, id = %d\n", buffptr, prio, nbytes,
id);
            printf("contents of ptr = %s\n", (char *)buffptr);
            free(buffptr);
            printf("heap space memory freed\n");
        }
    }
    return NULL;
}

static char imagebuff[4096];

void *sender(void *arg)
{
    char buffer[sizeof(void *)+sizeof(int)];
    void *buffptr;
    int prio;
    int nbytes;
    int id = 999;

    while (1)
    {
        /* Send malloc'd message with priority=30 */
        buffptr = (void *)malloc(sizeof(imagebuff));
        strcpy(buffptr, imagebuff);
        printf("Message to send = %s\n", (char *)buffptr);

        printf("Sending %ld bytes\n", sizeof(buffptr));

        memcpy(buffer, &buffptr, sizeof(void *));
        memcpy(&(buffer[sizeof(void *)]), (void *)&id, sizeof(int));

        if ((nbytes = mq_send(mymq, buffer, (sizeof(void *) + sizeof(int)), 30)) == -1)
        {
            perror("mq_send");
        }
        else
        {
           printf("send: message ptr %p successfully sent\n", buffptr);
        }

        // Introduce a delay
        usleep(3000000); // 3 seconds

    }
    return NULL;
}

static int sid, rid;

void heap_mq(void)
{
    pthread_t receiver_thread, sender_thread;
    pthread_attr_t receiver_attr, sender_attr;
    struct sched_param receiver_param, sender_param;

    int i, j;
    char pixel = 'A';

    for(i=0;i<4096;i+=64) {
    pixel = 'A';
    for(j=i;j<i+64;j++) {
      imagebuff[j] = (char)pixel++;
    }
    imagebuff[j-1] = '\n';
    }
    imagebuff[4095] = '\0';
    imagebuff[63] = '\0';

    printf("buffer =\n%s", imagebuff);

    // Setup common message queue attributes
    mq_attr.mq_maxmsg = 100;
    mq_attr.mq_msgsize = sizeof(void *)+sizeof(int);
    mq_attr.mq_flags = 0;
```

```
// Initialize attributes
pthread_attr_init(&receiver_attr);
pthread_attr_init(&sender_attr);

// Set scheduling parameters
pthread_attr_setschedpolicy(&receiver_attr, SCHED_FIFO);
pthread_attr_setschedpolicy(&sender_attr, SCHED_FIFO);

// Set priority for receiver and sender threads
receiver_param.sched_priority = 100; // Higher priority for receiver
sender_param.sched_priority = 90;

pthread_attr_setschedparam(&receiver_attr, &receiver_param);
pthread_attr_setschedparam(&sender_attr, &sender_param);

// Create message queue
mymq = mq_open(SNDRCV_MQ, O_CREAT | O_RDWR, 777, &mq_attr);
if (mymq == (mqd_t)-1)
{
   perror("mq_open");
   exit(1);
}

// Create receiver and sender threads with the specified attributes
if (pthread_create(&receiver_thread, &receiver_attr, receiver, NULL) != 0)
{
   perror("pthread_create");
   exit(1);
}
else
{
   printf("Receiver thread created\n");
}

if (pthread_create(&sender_thread, &sender_attr, sender, NULL) != 0)
{
   perror("pthread_create");
   exit(1);
}
else
```

```
    {
        printf("Sender thread created\n");
    }

    // Clean up attributes
    pthread_attr_destroy(&receiver_attr);
    pthread_attr_destroy(&sender_attr);

    // Wait for threads to complete
    pthread_join(receiver_thread, NULL);
    pthread_join(sender_thread, NULL);

    // Close message queue
    mq_close(mymq);
}

void shutdown(void)
{
  mq_close(mymq);
}

int main()
{
mq_unlink(SNDRCV_MQ);  // Make sure that SNDRCV_MQ is cleanly available
    heap_mq();
    return 0;
}
```

****

posix_mq.c

```
/*
 * File: posix_mq.c
 * Author: Krishna Suhagiya and Suhas Reddy
 * Description: This file ports the provided VxWorks posix_mq.c implementation to POSIX with
SCHED_FIFO scheduling.
 * Date: 9th March 2023
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <mqueue.h>
#include <string.h>
#include <unistd.h>
#include <sched.h>

#define SNDRCV_MQ "/send_receive_mq"
#define MAX_MSG_SIZE 128

struct mq_attr mq_attr;

void *receiver(void *arg)
{
   mqd_t mymq;
   char buffer[MAX_MSG_SIZE];
   int prio;
   int nbytes;

   // Open the message queue for reading
   mymq = mq_open(SNDRCV_MQ, O_CREAT | O_RDWR, 777, &mq_attr);
   if (mymq == (mqd_t)-1)
   {
      perror("mq_open");
      exit(1);
   }

   // Read oldest, highest priority message from the message queue
   if ((nbytes = mq_receive(mymq, buffer, MAX_MSG_SIZE, &prio)) == -1)
   {
      perror("mq_receive");
   }
   else
   {
      buffer[nbytes] = '\0';
      printf("Receiver: Message '%s' received with priority = %d, length = %d\n", buffer, prio, nbytes);
   }

   // Close the message queue
```

```
    mq_close(mymq);
    return NULL;
}

static char canned_msg[] = "this is a test, and only a test, in the event of a real emergency, you would be
instructed ...";

void *sender(void *arg)
{
    mqd_t mymq;
    int prio;
    int nbytes;

    // Open the message queue for writing
    mymq = mq_open(SNDRCV_MQ, O_RDWR, 777, &mq_attr);
    if (mymq == (mqd_t)-1)
    {
        perror("mq_open");
        exit(1);
    }

    // Send message with priority=30
    if ((nbytes = mq_send(mymq, canned_msg, sizeof(canned_msg), 30)) == -1)
    {
        perror("mq_send");
    }
    else
    {
        printf("Sender: Message successfully sent\n");
    }

    // Close the message queue
    mq_close(mymq);
    return NULL;
}

void mq_demo(void)
{
    pthread_t receiver_thread, sender_thread;
    pthread_attr_t receiver_attr, sender_attr;
```

```
struct sched_param receiver_param, sender_param;

// Setup common message queue attributes
mq_attr.mq_maxmsg = 100;
mq_attr.mq_msgsize = MAX_MSG_SIZE;
mq_attr.mq_flags = 0;

// Initialize attributes
pthread_attr_init(&receiver_attr);
pthread_attr_init(&sender_attr);

// Set scheduling parameters
pthread_attr_setschedpolicy(&receiver_attr, SCHED_FIFO);
pthread_attr_setschedpolicy(&sender_attr, SCHED_FIFO);

// Set priority for receiver and sender threads
receiver_param.sched_priority = 100;
sender_param.sched_priority = 90;

pthread_attr_setschedparam(&receiver_attr, &receiver_param);
pthread_attr_setschedparam(&sender_attr, &sender_param);

// Create receiver and sender threads with the specified attributes
if (pthread_create(&receiver_thread, &receiver_attr, receiver, NULL) != 0)
{
   perror("pthread_create");
   exit(1);
}
else
{
   printf("Receiver thread created\n");
}

if (pthread_create(&sender_thread, &sender_attr, sender, NULL) != 0)
{
   perror("pthread_create");
   exit(1);
}
else
{
        printf("Sender thread created\n");
    }

    // Clean up attributes
    pthread_attr_destroy(&receiver_attr);
    pthread_attr_destroy(&sender_attr);

    // Wait for threads to complete
    pthread_join(receiver_thread, NULL);
    pthread_join(sender_thread, NULL);
}

int main()
{
    mq_unlink(SNDRCV_MQ);   // Make sure that SNDRCV_MQ is cleanly available
    mq_demo();
    return 0;
}
```

****

Question 5 code

```
/*
 * File: Q5.c
 * Author: Suhas Reddy and Krishna Suhagiya
 * Description: This program demonstrates a multi-threaded system for updating
 *              and reading navigation state data. It includes three threads:
 *              one for updating navigation state, one for reading the state,
 *              and one for timeout handling. The update thread updates the
 *              navigation state variables periodically, the read thread reads
 *              and prints the state, and the timeout thread handles resource
 *              acquisition timeouts. The program utilizes POSIX threads and
 *              synchronization mechanisms such as mutexes and condition variables.
 * Date: 9th March 2023
 */

#include <pthread.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <stdbool.h>
#include <errno.h>

#define PI 3.14
#define NUM_THREADS 3

bool run_complete = false;
pthread_cond_t signal_read = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

typedef struct {
    double Latitude;
    double Longitude;
    double Altitude;
    double Roll;
    double Pitch;
    double Yaw;
    struct timespec timestamp;
```

```
} nav_state;

typedef struct {
    int thread_idx;
    nav_state *state;
} thread_param;

static nav_state state;

void *update_nav_state(void *threadp) {
    thread_param *tp = (thread_param *)threadp;

    while (!run_complete) {
        pthread_mutex_lock(&mutex);
        nav_state *state = tp->state;
        clock_gettime(CLOCK_REALTIME, &state->timestamp);
        state->Latitude = 0.01 * (state->timestamp.tv_sec);
        state->Longitude = 0.2 * (state->timestamp.tv_sec);
        state->Altitude = 0.25 * (state->timestamp.tv_sec);
        state->Roll = sin(2 * PI * (state->timestamp.tv_sec));
        state->Pitch = cos(2 * PI * (state->timestamp.tv_sec) * (state->timestamp.tv_sec));
        state->Yaw = cos(2 * PI * (state->timestamp.tv_sec));
        pthread_cond_signal(&signal_read);   // Signal read function when update is complete
        pthread_mutex_unlock(&mutex);
        sleep(1); // Update rate of 1 Hz
    }

    pthread_mutex_lock(&mutex);
    return NULL;
}

void *read_nav_state(void *threadp) {
    thread_param *tp = (thread_param *)threadp;
    for (int i = 0; i < 18; i++) {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&signal_read, &mutex);  // Wait until update is complete
        nav_state *state = tp->state;
        printf("\nExecution number: %d", i);
        printf("\nLatitude: %lf", state->Latitude);
        printf("\nLongitude: %lf", state->Longitude);
        printf("\nAltitude: %lf", state->Altitude);
        printf("\nRoll: %lf", state->Roll);
        printf("\nPitch: %lf", state->Pitch);
        printf("\nYaw: %lf", state->Yaw);
        printf("\nTimestamp: %lu\n", state->timestamp.tv_sec);
        pthread_mutex_unlock(&mutex);
        sleep(10); // Read rate of 0.1 Hz
    }
    run_complete = true;
    return NULL;
}

void *timeout_thread(void *threadp) {
    for (int i = 0; i <= 19; i++) {
        struct timespec timeout;
        clock_gettime(CLOCK_REALTIME, &timeout);
        timeout.tv_sec += 10;
        printf("\nWaiting on Resources");
        int mutex_acquired;
        while ((mutex_acquired = pthread_mutex_timedlock(&mutex, &timeout)) != 0) {
            if (mutex_acquired == ETIMEDOUT) {
                printf("\nNo new data available at %lu", time(NULL));
                pthread_mutex_unlock(&mutex);
            }
        }

        pthread_mutex_unlock(&mutex);
        printf("\nAcquired Resources\n");

        sleep(10);  // Check for data at a rate of 0.1 Hz
    }

    return NULL;
}

int main() {
    printf("RTES Question 5\n");

    pthread_t threads[NUM_THREADS];
    struct timespec timestamp;
    clock_gettime(CLOCK_REALTIME, &timestamp);
    state.Latitude = 0.0;
    state.Longitude = 0.0;
    state.Altitude = 0.0;
    state.Roll = 0.0;
    state.Pitch = 0.0;
    state.Yaw = 0.0;
    state.timestamp = timestamp;

    thread_param thread0 = {0, &state}, thread1 = {1, &state};

    pthread_create(&threads[0], NULL, update_nav_state, (void *)&thread0);
    pthread_create(&threads[1], NULL, read_nav_state, (void *)&thread1);
    pthread_create(&threads[2], NULL, timeout_thread, NULL);

    // Wait for threads to finish
    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);
    pthread_join(threads[2], NULL);

    pthread_mutex_destroy(&mutex);

    return 0;
}
```

****

References

[1] Linux Kernel Watchdog Explained (linuxhint.com)
[2] Linux Watchdog configuring (crawford-space.co.uk)
[3] watchdog(8): software watchdog daemon - Linux man page (die.net)
[4] https://drive.google.com/file/d/1Dtw4l5H1wEhmuNmLnoKHQY0KNz-2xaKW/view
[5] https://www.linuxfoundation.org/blog/blog/intro-to-real-time-linux-for-embedded-developers
[6] https://man7.org/linux/man-pages/