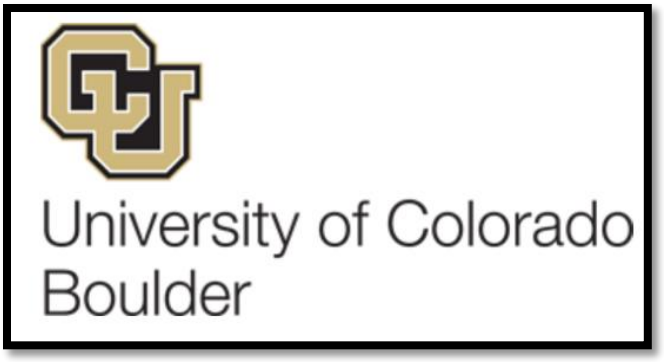

EXERCISE #3 – THREADING/TASKING AND REAL-TIME SYNCHRONIZATION

REAL TIME EMBEDDED SYSTEMS (ECEN 5623)

SUMMER 2019

BOARD USED:

NVIDIA's JETSON NANO DEVELOPMENT BOARD

REPORT BY:

OM RAHEJA

PROFESSOR:

DR. SAM SIEWERT

1) Read Sha, Rajkumar, et al paper, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization" and summarize 3 main key points the paper makes. Read my summary paper on the topic as well. Finally, read the positions of Linux 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?

Ans:

Priority inheritance paper review and unbounded priority inversion issues:

The paper on "Priority Inheritance Protocols: An approach to real-time synchronization" written by Lui Sha, Ragunathan Rajkumar and John Lehoczky primarily describes the problems related to priority inversion[*] and focuses on the two priority inheritance protocols, namely, basic priority inheritance protocol and priority ceiling protocol.

Problems that arise due to priority inversion:

The paper mentions the problems that arise as a result of priority inversion. When two jobs try to access a data that is shared with other jobs, the access to the shared data must be synchronized. By synchronized, it means at any given point in time, only one of the many jobs will have access to the shared data. Now, all jobs run at a different priority level. Thus, in a normal scenario, the highest priority job should get priority to access the shared data over the lower priority jobs. But this is not always the case. Let's examine this situation by considering two cases:

CASE 1: When a higher priority job gains access first, then a proper order is maintained.

CASE 2: When a lower priority job gains access first and then the higher priority job requests access to the shared data. In this case, the higher priority job has to wait until the lower priority job releases the resource.

In Case 1, the access to the shared data is serialized, whereas, in case 2, the higher priority job is blocked by the lower priority job for the time until which the lower priority job runs to completion. Thus, Blocking is a form of priority inversion.

Effects of blocking:

- System missing its deadlines due to long durations of blocking.
- Effects the schedulability of the system.

To address these issues, it led to the development of two protocols that would minimize the amount of blocking. The two priority inheritance protocols are as follows:

1) Basic priority inheritance protocol
2) Priority ceiling protocol

Basic priority inheritance protocol:

The basic priority inheritance protocol states that "if a job 'J' blocks one or more higher priority tasks, then, it ignores its original priority and executes its critical section at the highest priority level of all the jobs it blocks. Once, job J exits its critical section, it reverts back to its original priority that was assigned to it when it was about to enter the critical section". This way, a lower priority job can block a higher priority job until it finishes its execution of the critical section. The blocking of the higher priority job can take place in two ways:

a) Direct blocking
b) Push-through blocking

Direct Blocking: We know that semaphores, monitors, locks etc., can be used to maintain consistency of the shared data. But, we also know the potential threat they can possess for a system to be able to meet its timing requirements. Direct blocking occurs when a higher priority job tries to lock an already locked semaphore. This semaphore might be blocked by a lower priority job which is executing in its critical section. According to the basic priority inheritance protocol, the lower priority job gets assigned the priority equal to that of the highest priority job being blocked by it.

Push-through blocking: Suppose there are three jobs J_0, J_1, J_2. Let J_0 be of the highest priority and J_2 be the one with least priority. Push-through blocking occurs when a medium priority job (J_1) gets blocked by a lower priority job (J_2) which in turn inherits the priority of J_0. By doing this, we prevent the highest priority job being blocked by the medium priority job indirectly.

Problems in Basic priority inheritance protocol:

i. The basic priority inheritance protocol does not prevent deadlocks.
ii. The duration of blocking a job is bounded, but still in some cases it can be significant enough to cause a chain of blocking[*].

Priority Ceiling Protocol:

The priority ceiling protocol can solve both the issues of the basic priority inheritance protocol. The priority ceiling protocol aims at resolving the issue of deadlocks and chained blocking that persists in the basic priority inheritance protocol.

The paper perfectly describes the protocol. It states that "when a job J preempts the critical section of another job and executes its own critical section, the priority at which this new critical section will execute is guaranteed to be higher than the inherited priorities of all the preempted critical sections. If this condition cannot be satisfied, job J is denied entry into the critical section and suspended, and the job that blocks J inherits J's priority."

There are two assumptions that are made while describing the properties of the priority ceiling protocol:

Firstly, it is assumed that "job[*] is a sequence of instructions that run to completion considering that it is the only job on the processor". Secondly, "a job will release all its locks, if it holds any, before or at the end of its execution".

Schedulability Analysis:

According to the rate-monotonic algorithm, highest priority is assigned to the job which occurs at a higher frequency (or the one which has shorter time periods).

Assumptions made for Schedulability Analysis:

i. All the tasks are considered periodic, i.e., they occur after a certain regular interval.
ii. It is assumed that each job has deterministic execution times for both its critical and non-critical sections and that it does not synchronize with external events.
iii. The periodic tasks are assigned priorities according to the rate monotonic algorithm.

If we consider *n* periodic tasks that use the priority ceiling protocol, then, according to the rate monotonic policy, these tasks can be scheduled only if it satisfies the following condition:

$$\forall i,\ 1 \le i \le n, \qquad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1).$$

Where,
B_i represents the worst-case blocking duration. T_1 , T_2 represents time periods for Task1 and Task2. C_1 , C_2 represents run-times for Task1 and Task2.

Three priority inheritance key points made in paper:

i. The paper analyzes the duration for which a job J can be blocked by *n* lower priority jobs. After detailed analysis, it was concluded that the job J can be blocked for a maximum duration of one critical section in each of the $\beta_{0,i}^{*}$, $1 \le i \le n$.
ii. The paper also discusses the use of synchronization mechanisms in maintaining the consistency of the shared data. They also describe the potential damage they can cause to the system ability to meet its timing requirements. Later, they go on to state the two priority inheritance protocols which when used, avoids the abovementioned issues.
iii. According to the scheduling analysis, "A set of *n* periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm if the following conditions are satisfied":

$$\forall i,\ 1 \le i \le n, \qquad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1).$$

Where B_i represents the worst-case blocking duration.

Why the Linux position makes sense or not:

The article describes the view point of Linus Torvalds and Ingo Molnar about including the PI futex patch and merging it with the mainline Linux kernel. It is quite evident that Linus does not support the inclusion of the patch which solves the priority inversion problem. He states that if a system requires a mechanism of priority inheritance to avoid priority inversion, then the system is already broken. Linus states that there is a mechanism to avoid priority inversion by not allowing the kernel code to preempt by holding a spinlock. Linus felt that majority of the priority inheritance schemes are complicated and slow down the locking code. This may result in poor application design. Ingo, on the other hand had developed a functionality which operated in the user space and did not involve any preemption of the kernel tasks. His approach also avoided the fallbacks other implementations had. He had implemented PI-Futex in a manner which reduced the overhead on the kernel side. PI-Futex can be taken without involving the kernel. I feel that the PI-Futex patch could prove to be a good inclusion in the mainline Linux kernel. It may happen that a situation arises leading to priority inversion. Having a support to avoid priority inversion is beneficial instead of having no protection for such cases.

Reasoning on whether FUTEX fixes unbounded priority inversion:

Futex is a fast user-space locking mechanism whose implementation is similar to mutex. A futex grabs the lock without the involvement of a kernel system call which is not the case with mutexes and semaphores. Because of this reason, futex has less overhead when compared to mutexes and semaphores. The PI-Futex do solve the problem of the unbounded priority inversion. If a higher priority task is being blocked by a lower priority job, then the lower priority task is assigned the priority of the highest priority task it is blocking. This prevents the higher priority tasks to wait for an indefinite time for a lower priority task to run to completion.

2) Review the terminology and 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. Describe each method and how you would code it and how it would impact real-time threads/tasks. Now, using a MUTEX, provide an example using RT-Linux Pthreads that does a thread safe update of a complex state 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 {X,Y,Z acceleration and Roll, Pitch, Yaw rates at Sample_Time} (just make up values for the navigational state and see http://linux.die.net/man/3/clock_gettime for how to get a precision timestamp). The second thread should read the times-stamped state without the possibility of data corruption (partial update).

Ans:

THREAD SAFETY WITH GLOBAL DATA AND ISSUES FOR REAL-TIME SERVICES:

Definition of a thread-safe function:

The concept of thread safe functions comes into picture when we talk about a multi-threaded program. If multiple threads can call a function that operates on a shared data simultaneously, then the function is said to be thread safe if the access to the shared data is serialized, meaning the shared data can be accessed by only one thread at a given time.

Definition of a reentrant function:

A reentrant function is one which can be interrupted during its execution, service the function that interrupted it and then resume the same function from the point where it was interrupted. Multiple threads can call a reentrant function simultaneously if each of the calling function uses its own set of data.

A thread-safe function is always reentrant but the vice versa is not always true.

In order to write a thread safe reentrant functions, the following set of rules must be kept in mind:

i. Global or static variables must not be used.
ii. The reentrant function should not modify its own code.
iii. It should not call any other non-entrant function.

DESCRIPTION OF METHODS:

Pure functions that use only stack and have no global memory:

A function can be made thread-safe and reentrant by making use of a local variables instead of using global variables. The reason why using global variables can be a problem is shown below with the help of a sample C code:

```
/* Global variable */
int temp;


void swap_numbers(int *num1,int *num2)
{
        temp = *num1;
        *num1 = *num2;
        *num2 = temp;
}



void fun()
{
        int num1 = 5;
        int num2 = 10;
        swap(&num1,&num2);
}
```

Here, we can see that temp is a global variable used to store the value of number 1 temporarily. Assume that the function 'fun' has called swap_numbers once. Now, the swap_numbers function is currently performing the swapping of the two numbers. It may happen that while this swapping is going on, the function 'fun' will be called by some other thread. This can lead to unpredictable values of the global variable temp. Thus, making use of global data can be thread-unsafe.

A thread safe function is meant to protect the shared data from multiple threads if they try to access it concurrently. Any function that uses local variables or does not used the keyword 'static' before declaring any of the variables, then the function is called as a thread-safe function. Here's an example of a thread safe function.

```
/* Thread Safe function */
int difference(int num1,int num2)
{
        int diff;

        diff = num1 - num2;

        if(diff < 0)
        {
                diff = -diff;
        }

        return diff;
}
```

These are called a pure function because the return value of the function will remain same for a set of same input. Its return values are not affected by manipulation of any global data.

Functions which use thread indexed global data:

A common virtual address space is shared by all the threads in a process. The global variables are shared by all the threads in a process. On the other hand, the local variables are unique to each thread. By using a Thread Local Storage (TLS), unique data can be provided to each thread. This data can be accessed by the process by using a global index. One of the threads allocates the

index. This index can then be used by all the other threads to retrieve the data unique to that thread.
Suppose a process has two threads, Thread 1 and Thread 2. The process allocates two indexes for use with the thread local storage. The data can be stored in the memory blocks allocated by each thread. It stores the pointers to these memory blocks in the corresponding TLS slots. In order to access the data associated with a particular index, the threads retrieve the pointer to the memory location from the thread local storage slot and stores the local variable in it.
This way, a variable is made global but local to that thread, thus, minimizing the risk of corruption of the variable.

Functions which use shared memory global data but synchronize access to it using a MUTEX semaphore critical section wrapper:

A function that contains data shared among different threads has to be protected against any potential data corruption. This data corruption could be a result of two or more threads accessing and manipulating the shared data simultaneously. In order to avoid the above scenario, the threads need to be synchronized with one another so that the shared data can be accessed serially, one thread at a time. Various methods can be used to achieve this. Mutexes, semaphores and locks are a good way to achieve synchronization between multiple threads. Mutex lock and mutex unlock APIs can be used or semaphore wait, and semaphore post APIs can be used to achieve synchronization between threads.

Impact of each on real-time services:

The first approach which uses local variables instead of global memory, would not have any impact on the real-time services. Even if threads interrupted each other, there would not be any data corruption as the variable that the thread will manipulate will eventually be in local scope.

The second approach of using thread indexed global data can be adopted provided the functions are reentrant. Data corruption will not take place as each thread will use their own data set. When it comes to real-time services, the constraint that each thread should run to completion before it is called the next time, should stand true. Therefore, if at all a situation arises that a function calls itself before its completion, then the tasks would not be feasible.

The third approach of using mutexes, semaphores and locks to synchronize multiple threads is often used in real time embedded systems. This work well provided we have a well structured and well-designed scheduling algorithm. By this I mean that thought has been given to situation which can tackle problems like priority inversion, deadlocks and livelocks. It may happen that using mutexes and semaphores can result in deadlock between multiple threads. Thus, with proper consideration of such issues, a real time system can be designed using critical section locking mechanisms.

Correct shared state update code:

```
root@om:/home/om/RTES/Exercise_3/question2# clear
root@om:/home/om/RTES/Exercise_3/question2# sudo ./thread
************************************************** 0 ITERATION **************************************************
Updating values.........
Updating TimeStamp = 1561967025.476144
Updating X     = 797644273.000000       Updating Y     = 982082457.000000       Updating Z      = 437382747.000000
Updating Yaw   = 10572164.000000        Updating Roll  = 316170005.000000       Updating Pitch  = 420039169.000000
Reading values.........
Timestamp:[1561967025.476144 secs]
X-axis = [797644273.000000]     Y-axis = [982082457.000000]     Z-axis = [437382747.000000]
Yaw    = [10572164.000000]      Roll   = [316170005.000000]     Pitch  = [420039169.000000]

********************************************************************************************************************
************************************************** 1 ITERATION **************************************************
Updating values.........
Updating TimeStamp = 1561967025.477618
Updating X     = 1245391955.000000      Updating Y     = 589069846.000000       Updating Z      = 2032080202.000000
Updating Yaw   = 527998791.000000       Updating Roll  = 1726518718.000000      Updating Pitch  = 1995117625.000000
Reading values.........
Timestamp:[1561967025.477618 secs]
X-axis = [1245391955.000000]    Y-axis = [589069846.000000]     Z-axis = [2032080202.000000]
Yaw    = [527998791.000000]     Roll   = [1726518718.000000]    Pitch  = [1995117625.000000]

********************************************************************************************************************
************************************************** 2 ITERATION **************************************************
Updating values.........
Updating TimeStamp = 1561967025.478874
Updating X     = 838875076.000000       Updating Y     = 1716159105.000000      Updating Z      = 1459468704.000000
Updating Yaw   = 1328734275.000000      Updating Roll  = 1426780449.000000      Updating Pitch  = 1752901616.000000
Reading values.........
Timestamp:[1561967025.478874 secs]
X-axis = [838875076.000000]     Y-axis = [1716159105.000000]    Z-axis = [1459468704.000000]
Yaw    = [1328734275.000000]    Roll   = [1426780449.000000]    Pitch  = [1752901616.000000]

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

```
************************************************** 3 ITERATION **************************************************
Updating values.........
Updating TimeStamp = 1561967025.480067
Updating X     = 260247823.000000       Updating Y     = 1881928752.000000      Updating Z      = 12665130.000000
Updating Yaw   = 1295798738.000000      Updating Roll  = 1940976883.000000      Updating Pitch  = 1394620215.000000
Reading values.........
Timestamp:[1561967025.480067 secs]
X-axis = [260247823.000000]     Y-axis = [1881928752.000000]    Z-axis = [12665130.000000]
Yaw    = [1295798738.000000]    Roll   = [1940976883.000000]    Pitch  = [1394620215.000000]

********************************************************************************************************************
************************************************** 4 ITERATION **************************************************
Updating values.........
Updating TimeStamp = 1561967025.481439
Updating X     = 2035638004.000000      Updating Y     = 1249494914.000000      Updating Z      = 1345986321.000000
Updating Yaw   = 279525443.000000       Updating Roll  = 172521802.000000       Updating Pitch  = 321025445.000000
Reading values.........
Timestamp:[1561967025.481439 secs]
X-axis = [2035638004.000000]    Y-axis = [1249494914.000000]    Z-axis = [1345986321.000000]
Yaw    = [279525443.000000]     Roll   = [172521802.000000]     Pitch  = [321025445.000000]

********************************************************************************************************************
************************************************** 5 ITERATION **************************************************
Updating values.........
Updating TimeStamp = 1561967025.482574
Updating X     = 1892831926.000000      Updating Y     = 1118669719.000000      Updating Z      = 1261607900.000000
Updating Yaw   = 1681647069.000000      Updating Roll  = 182731025.000000       Updating Pitch  = 1434839724.000000
Reading values.........
Timestamp:[1561967025.482574 secs]
X-axis = [1892831926.000000]    Y-axis = [1118669719.000000]    Z-axis = [1261607900.000000]
Yaw    = [1681647069.000000]    Roll   = [182731025.000000]     Pitch  = [1434839724.000000]

********************************************************************************************************************
************************************************** 6 ITERATION **************************************************
Updating values.........
Updating TimeStamp = 1561967025.483798
Updating X     = 193303190.000000       Updating Y     = 532748032.000000       Updating Z      = 123233268.000000
Updating Yaw   = 2118350893.000000      Updating Roll  = 77899744.000000        Updating Pitch  = 111783102.000000
Reading values.........
Timestamp:[1561967025.483798 secs]
X-axis = [193303190.000000]     Y-axis = [532748032.000000]     Z-axis = [123233268.000000]
Yaw    = [2118350893.000000]    Roll   = [77899744.000000]      Pitch  = [111783102.000000]

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

```
************************************************* 7 ITERATION *************************************************
Updating values.........
Updating TimeStamp = 1561967025.485032
Updating X     = 605898535.000000       Updating Y     = 950658179.000000       Updating Z      = 1687026350.000000
Updating Yaw   = 1292444318.000000      Updating Roll  = 2065367240.000000      Updating Pitch  = 229954980.000000
Reading values..........
Timestamp:[1561967025.485032 secs]
X-axis = [605898535.000000]     Y-axis = [950658179.000000]     Z-axis = [1687026350.000000]
Yaw    = [1292444318.000000]    Roll   = [2065367240.000000]    Pitch  = [229954980.000000]

*****************************************************************************************************************
************************************************* 8 ITERATION *************************************************
Updating values.........
Updating TimeStamp = 1561967025.486238
Updating X     = 1246617867.000000      Updating Y     = 490202803.000000       Updating Z      = 1026889423.000000
Updating Yaw   = 274025990.000000       Updating Roll  = 1259282997.000000      Updating Pitch  = 283696039.000000
Reading values..........
Timestamp:[1561967025.486238 secs]
X-axis = [1246617867.000000]    Y-axis = [490202803.000000]     Z-axis = [1026889423.000000]
Yaw    = [274025990.000000]     Roll   = [1259282997.000000]    Pitch  = [283696039.000000]

*****************************************************************************************************************
************************************************* 9 ITERATION *************************************************
Updating values.........
Updating TimeStamp = 1561967025.487472
Updating X     = 407598087.000000       Updating Y     = 171850395.000000       Updating Z      = 1523520904.000000
Updating Yaw   = 1844546349.000000      Updating Roll  = 1753584408.000000      Updating Pitch  = 344372197.000000
Reading values..........
Timestamp:[1561967025.487472 secs]
X-axis = [407598087.000000]     Y-axis = [171850395.000000]     Z-axis = [1523520904.000000]
Yaw    = [1844546349.000000]    Roll   = [1753584408.000000]    Pitch  = [344372197.000000]

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

3) Download example-sync/ and describe both the issues of deadlock and unbounded priority inversion and the root cause for both in the example code. 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? What about a patch for the Linux kernel? For example, Linux Kernel.org recommends the RT_PREEMPT Patch, 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?

Ans:

Demonstration and description of deadlock with threads:

Deadlock is a situation where a set of threads/tasks/processes wait for resources held by the other. For example, in this case, we have two set of threads that need resource A and resource B in order to run to completion. Thread 1 is created first, which then moves onto acquiring resource A. Meanwhile, thread 2 is spawned which goes onto acquiring resource B. The first thread needs resource B as well. Similarly, the second thread needs resource A. Both of these threads wait for the other one to release the resource so that they can grab the resource and run to completion. Such a situation in which a set of threads wait for one another to release a resource is called as a deadlock. Deadlocks can be prevented by careful synchronization and serialization of data access between multiple threads.

Root cause: Thread 1 has resource A and awaits resource B. Thread 2 has resource B and awaits resource A. Both the threads wait for the other thread to release a resource. But that does not happen as the other thread is also waiting for the first thread to release resource A. This situation is referred as deadlock.

```
root@om:/home/om/RTES/Exercise_3/question3# ./deadlock
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 2 got B, trying for A
THREAD 1 got A, trying for B
^C
root@om:/home/om/RTES/Exercise_3/question3#
```

Deadlock Solved:

```
root@om:/home/om/RTES/Exercise_3/question3# ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
THREAD 2 grabbing resources
will try to join CS threads unless they deadlock
THREAD 1 grabbing resources
THREAD 2 got B, trying for A
THREAD 1 got A, trying for B
THREAD 2 got B and A
THREAD 2 done
Thread 1: 548045758960 done
Thread 2: 548037366256 done
mutex A destroy: Success
mutex B destroy: Success
All done
root@om:/home/om/RTES/Exercise_3/question3#
```

Demonstration and description of priority inversion with threads:

In the pthread3 example, there are three threads running at different priority levels (Low, Medium, High). The lowest priority thread and the highest priority thread share a common resource. The lowest priority thread grabs the resource first and locks it. Meanwhile, the highest priority thread waits for the lowest priority thread to release the resource. It may also happen that the medium priority thread (which does not share common resources with highest and lowest priority thread) may preempt or interrupt the lowest priority thread. If this happens, then the highest priority thread has to wait for an unpredictable amount of time until it gets the resource. This is called as priority inversion.

A more formal definition for priority inversion is when a thread with higher priority has to wait for a thread with lower priority, then it results in priority inversion.

Priority inversion can be solved by using priority inheritance protocols like the basic priority inheritance protocol and the priority ceiling protocol.

Below is a screenshot of the pthred3.c code with an interference time of 5 seconds.

```
root@om:/home/om/RTES/Exercise_3/question3# ./pthread3 5
interference time = 5 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 3
Low prio 3 thread spawned at 1562010174 sec, 739569 nsec
Creating thread 2
Middle prio 2 thread spawned at 1562010175 sec, 739745 nsec
Creating thread 1, CScnt=1
**** 2 idle NO SEM stopping at 1562010175 sec, 739759 nsec
High prio 1 thread spawned at 1562010175 sec, 739805 nsec
**** 3 idle stopping at 1562010176 sec, 739712 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1562010178 sec, 739858 nsec
HIGH PRIO done
START SERVICE done
All done
root@om:/home/om/RTES/Exercise_3/question3#
```

The pthread3ok.c solves the issue of the priority inversion. The lowest priority job blocks the highest priority job until it releases the resource. According to priority inheritance protocol, the lowest priority job will inherit the highest priority until it exits the critical section. Once the job exits its critical section, it resumes the priority it had when it entered the critical section.

```
root@om:/home/om/RTES/Exercise_3/question3# ./pthread3ok 5
interference time = 5 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 1
High prio 1 thread spawned at 1562011761 sec, 518295 nsec
Creating thread 2
**** 1 idle stopping at 1562011761 sec, 518351 nsec
Middle prio 2 thread spawned at 1562011761 sec, 518353 nsec
Creating thread 3
**** 2 idle stopping at 1562011761 sec, 518444 nsec
Low prio 3 thread spawned at 1562011761 sec, 518464 nsec
**** 3 idle stopping at 1562011761 sec, 518539 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
START SERVICE done
All done
root@om:/home/om/RTES/Exercise_3/question3#
```

Description of RT_PREEMPT_PATCH and assessment of whether Linux can be made real-time safe:

The Linux kernel by itself is not real time capable. But, with the addition of the RT_PREEMPT_PATCH it gains real-time capabilities. The main aim of introducing this patch is to make the existing Linux kernel preemptible. The RT_PREEMPT_PATCH provides features like preemptible critical sections, interrupt handlers, priority inherited in-kernel spinlocks and semaphores to make the kernel preemptible and provide a much better response time, thus making Linux + RT_PREEMPT_PATCH closer to resemble the RTOS.

With this patch, we can assign priorities even to kernel tasks such that user space threads can preempt them. These features could help solve the unbounded priority inversion issue. However, due to a small amount of latency in the Linux kernel the unbounded priority inversion cannot be fully solved. Locking mechanisms like spinlocks are implemented as mutexes and semaphores which help to preserve the critical sections against data corruption.

I think that the decision to switch from a Linux OS to an RTOS depends upon a variety of factors for an application. It depends if the system requirement is strictly hard real-time or if the system can afford to miss a few deadlines and still work safely. Predictability of the system is often decided by the kind of scheduling algorithm the system follows. Safety is one more factor which is crucial in deciding the requirements of the system. In hard real-time systems, safety is often the most important factor to consider. On the other hand, safety can be overlooked to some extent in a soft real time system. To sum up, I'd say if an application demands features which resemble to a hard-real time system then switching to an RTOS makes more sense. However, if the requirements of the system to be designed is not too strict, then a RT Linux can be a good platform to build that application.

4) Review heap_mq.c and posix_mq.c. First, re-write the VxWorks code so that it uses RT-Linux Pthreads (FIFO) instead of VxWorks tasks, and then write a brief paragraph describing how these two message queue applications are similar and how they are different. You may find the following Linux POSIX demo code useful, but make sure you not only read and port the code, but that you build it, load it, and execute it to make sure you understand how both applications work and prove that you got the POSIX message queue features working in Linux on your Jetson. 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?

Ans:

Demonstration of message queues on Jetson Nano adapted from examples:

Description of posix_mq.c code:

Flow of code:

a) Main:

- Message queue attributes (mq_maxmsg, msgsize, flags) were set.
- Assign higher priority (99) to receiver thread and lower priority (98) to sender thread.
- Initialize thread attributes.
- Set the scope of the threads to PTHREAD_SCOPE_SYSTEM so that the threads compete for resources with other threads on the system.
- Set the scheduling policy of the threads to SCHED_FIFO.
- Set the scheduling parameters.
- Create the two threads.
- Wait for the threads to join.

b) Receiver thread:

- Creates and open a message queue with read only attribute.
- Wait on mq_receive to receive a message from the sender.
- Display the message after reading it from the message queue.

c) Sender thread:

- Opens the message queue with read-write attributes.
- Sends a message using mq_send API.

```
root@om:/home/om/RTES/Exercise_3/question4/posix_mq# ./posix_msgq
Receiver thread created
Receiver thread
Receiver Thread Priority: 99
Sender thread created
Receiver opened message queue!
Sender thread
Sender Thread Priority: 98
Sender opened message queue!
Sender:Message sent successfully
Receiver:[Message:This is a test, and only a test, in the event of real emergency, you will be instructed....]
root@om:/home/om/RTES/Exercise_3/question4/posix_mq#
```

<u>Description of heap_mq.c code:</u>

Flow of code:

a) Main:
- Assign higher priority to receiver thread and lower priority to sender thread.
- Initialize thread attributes.
- Set the scope of the threads as PTHREAD_SCOPE_SYSTEM.
- Set the scheduling policy as SCHED_FIFO.
- Set the scheduling parameters,
- Set message queue attributes (mq_maxmsg, mq_msgsize, mq_flags).
- Create and open the queue with read-write attributes.
- Create the two threads.
- Wait for them to join and then terminate the program.
- Close the message queue.

b) Receiver thread:
- Wait on mq_receive to receive the message from the sender.
- Print the message received.
- Free the heap space memory.

c) Sender thread:
- Allocate memory of the size of the image.
- Send the message on the message queue.
- Sleep for 3 seconds before sending the data once again.

```
root@om:/home/om/RTES/Exercise_3/question4/heap_mq# ./heap_mq
buffer =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
Receiver thread created
Reading 8 bytes
Sender thread created
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
Sending 8 bytes
send: message ptr 0x0x7f9c000b20 successfully sent
receive: ptr msg 0x0x7f9c000b20 received with priority = 30, length = 12, id = 999
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
heap space memory freed
Reading 8 bytes
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
Sending 8 bytes
send: message ptr 0x0x7f9c000b20 successfully sent
receive: ptr msg 0x0x7f9c000b20 received with priority = 30, length = 12, id = 999
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
heap space memory freed
Reading 8 bytes
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
Sending 8 bytes
send: message ptr 0x0x7f9c000b20 successfully sent
receive: ptr msg 0x0x7f9c000b20 received with priority = 30, length = 12, id = 999
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
heap space memory freed
Reading 8 bytes
^C
root@om:/home/om/RTES/Exercise_3/question4/heap_mq#
```

In both the codes, i.e. the posix_mq.c and the heap_mq.c, message queue is used to send and receive data. The way they send the message is different in both the cases.
In posix_mq.c, the entire message is sent via the message queue. On the other hand, in heap_mq.c, the starting address of the message is copied in a pointer and the pointer is sent over the message queue. The receiver on the other hand, receives the start address of the message in the heap. It copies the entire message in a buffer and displays the message.
The heap_mq.c demonstrates a much efficient way of sending a message over a message queue. Instead of sending a message of large size, we can send just the address at which the message is located. This way, it will have less overhead when compared to sending the entire data at once.

Description of how message queues would or would not solve issues associated with global memory sharing:

Although priority inversion can never be fully removed, its effect can still be minimized by determining the source which is causing a priority inversion. Message queues can cause un-bounded priority inversion. This can happen as higher priority task can be queued behind a lower priority task in the message queue. This results in priority inversion. In order to solve this issue, we can prioritize the message queues. Thus, when a new message enters the queue with high priority, the sending thread must inherit the highest priority amongst all the messages in the queue. By adopting this approach, message queues can be used to solve the unbounded priority inversion problem. This feature is however, not available in Linux.

5) Watchdog timers, timeouts and timer services – First, read this overview of the Linux Watchdog Daemon and describe how it might be used if software caused an indefinite deadlock. 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.

Ans:

Description of how the Linux WD timer can help with recovery from a total loss of software sanity (E.g. system deadlock):

Watchdog timers (WDT):

- A hardware-based timer meant to ensure normal working of a system.
- If the system fails due to some reason (e.g. hardware fault, error in the program), the WDT resets the entire system in order to ensure that upon reboot, the system works normally. To prevent the WDT from resetting our system, we need to periodically reset it. The is commonly known as to kick the watchdog timer or to pet the watchdog timer.
- The Linux OS consists of two parts of the Watchdog:
    a) Watchdog module: Kernel driver module that can force a hard reset.
    b) Watchdog daemon: Refreshes the timer.

Watchdog Module:

- WDT can be a very important element of the system. In order to ensure that your system behaves correctly, WDT can be a good tool to ensure that.
- The Linux OS has the WDT provided as a kernel device driver.
- Linux also provides a less effective software watchdog timer called the 'softdog' timer. The reason it is less effective is due to the fact that if at all the kernel fails due to some reason, it will result in the failure of the softdog timer as well.
- The hardware WDT is helpful in providing a basic protection mechanism. If the system is successful in writing to /dev/watchdog periodically then the WDT is reset. If the system is unable to perform this operation on a periodic basis, the WDT will detect that the system is not performing normally and will eventually reset the entire system.

Watchdog Daemon:

- In Linux, a watchdog daemon can be configured in such a way that it would run a few basic tests in order to ensure that the system functions normally.
- If the test fails, the daemon can reset the system, eventually keeping a log of the reason for a reset so that the same problem can be prevented in the future.
- If the tests fail, the daemon can reboot the machine in a moderately orderly manner in order to keep a log of why it happened. This could be a good way to identify the cause of the failure and avoid those problems in the future.
- In some cases, the system stops all the processes by signaling everything with a SIGTERM and then with a non-ignorable SIGKILL.
- Tasks carried out after signaling all the processes:
    a) Update wtmp to record shutdown.

b) Update random seed to preserve entropy.
c) Sync the CMOS clock to system time. This is done so that the system time is reasonable on reboot.
d) Sync and unmount the file system before resetting the system by hardware reset.

- As the kernel stops the watchdog hardware on normal shut-down or reboot, the hardware reset approach is preferred over the kernel's reboot API. This is because the system could hang just after that point without any automatic recovery.
- Two daemons used for watchdog hardware support:
    a) wd_keepalive: provides hardware drive open/refresh/close actions.
    b) Watchdog: provides the actions same as wd-keepalive along with other system checks.

Use of Linux Watchdog Timer:

Watchdog timers can be very important to avoid software insanity issues like deadlocks. Let's consider we have two processes that pet the watchdog timer periodically, indicating correct functionality of the system. Let's assume that both processes use resources that the other process also requires in order to run to completion. Suppose a situation arises where both the processes are waiting for the other process to release a resource. In such a case, none of the two process will be able to reset the watchdog timer as both of them are in a situation where they wait for the other to release a resource. The watchdog timer eventually times-out and since it wasn't kicked by either of the two processes, it reboots the entire system in order to escape the deadlock. This way, the Watchdog timer can be useful if at all the software caused an indefinite deadlock.

Adaptation of code from #2 MUTEX sharing to handle timeouts for shared state:

```
root@om:/home/om/Real-Time-Embedded-Systems-ECEN-5623/Exercise_3/Question_5# ./watchdog
*****************************************************************************************************************************
Updating values.........

No new data available at 1562283573.496771
No new data available at 1562283583.496991
Updating TimeStamp = 1562283584.494542
Updating X      = 1429915303.000000      Updating Y      = 258310591.000000       Updating Z      = 584530566.000000
Updating Yaw    = 1933261342.000000      Updating Roll   = 1149244721.000000      Updating Pitch  = 671841706.000000

Reading values..........
Timestamp:[1562283584.494672 secs]
X-axis = [1429915303.000000]    Y-axis = [258310591.000000]     Z-axis = [584530566.000000]
Yaw    = [1933261342.000000]    Roll   = [1149244721.000000]    Pitch  = [671841706.000000]

*****************************************************************************************************************************
*****************************************************************************************************************************
Updating values.........

No new data available at 1562283593.497101

No new data available at 1562283603.497276
Updating TimeStamp = 1562283605.494883
Updating X      = 881955107.000000       Updating Y      = 1523727183.000000      Updating Z      = 313911816.000000
Updating Yaw    = 896026668.000000       Updating Roll   = 2072283213.000000      Updating Pitch  = 869519393.000000

Reading values..........
Timestamp:[1562283605.494997 secs]
X-axis = [881955107.000000]     Y-axis = [1523727183.000000]    Z-axis = [313911816.000000]
Yaw    = [896026668.000000]     Roll   = [2072283213.000000]    Pitch  = [869519393.000000]

*****************************************************************************************************************************
*****************************************************************************************************************************
Updating values.........

No new data available at 1562283613.497438

No new data available at 1562283623.497606
Updating TimeStamp = 1562283626.495219
Updating X      = 2098921365.000000      Updating Y      = 219705347.000000       Updating Z      = 225758242.000000
Updating Yaw    = 60834970.000000        Updating Roll   = 1383901625.000000      Updating Pitch  = 2011603819.000000

Reading values..........
Timestamp:[1562283626.495288 secs]
X-axis = [2098921365.000000]    Y-axis = [219705347.000000]     Z-axis = [225758242.000000]
Yaw    = [60834970.000000]      Roll   = [1383901625.000000]    Pitch  = [2011603819.000000]

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

*** Key Terminologies from Priority Inheritance protocols paper:**

[NOTE: These definitions are stated as it is in the "Priority Inheritance Protocols: An Approach to Real-Time Synchronization" **paper for immediate reference for the report].**

1) **Job:** A sequence of instructions that will continuously use the processor until its completion if it is executing alone on the processor.
2) **Periodic Task:** A sequence of the same type of job occurring at regular intervals.
3) **Aperiodic Task:** A sequence of the same type of job occurring at irregular intervals.
4) **Priority Inversion:** A situation in which a higher priority job is blocked by a lower priority job for an indefinite period of time is called as priority inversion.
5) **Priority Inheritance:** When a job blocks one or more high-priority jobs, it ignores its original priority assignment and executes its critical section at an elevated priority level. After executing its critical section and releasing its locks, the process returns to its original priority level.
6) **Priority Ceiling:** Priority ceiling of a semaphore as the priority of the highest priority job that may lock this semaphore.
7) **Deadlock:** In concurrent programming, deadlock occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process.
8) **Chained Blocking:** Chained blocking refers to a concept where in one job waits for another job for a resource which in turn waits for another job for a resource and so on. This way, the highest priority job has to wait for 'n' more jobs for its completion. This situation is coined as chained blocking.
9) **Transitive Blocking:** Let's consider three jobs, J_1, J_2 and J_3. Let J_1 be of the highest priority and J_3 of the least. Consider that J_3 blocks J_2 which in turn blocks J_1. In such a case, J_3 will inherit the priority of J_1 via J_2. This is called as transitive blocking.

References:

1) Priority Inheritance definition: https://en.wikipedia.org/wiki/Priority_inheritance
2) Deadlock definition: https://en.wikipedia.org/wiki/Deadlock
3) Watchdog timer [1]: http://www.sat.dundee.ac.uk/psc/watchdog/watchdog-background.html
4) Watchdog timer [2]: https://en.wikipedia.org/wiki/Watchdog_timer
5) Priority Inheritance Protocols: An Approach to Real-Time Synchronization: http://ecee.colorado.edu/~ecen5623/ecen/rtpapers/archive/PAPERS_READ_IN_CLASS/prio_inheritence_protocols.pdf
6) Thread safe and reentrant functions [1]: https://doc.qt.io/archives/qt-4.8/threads-reentrancy.html
7) Thread safe and reentrant functions [2]: https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.genprogc/writing_reentrant_thread_safe_code.html
8) Thread safe and reentrant functions [3]: https://deadbeef.me/2017/09/reentrant-threadsafe
9) Thread local storage: https://docs.microsoft.com/en-us/windows/desktop/procthread/thread-local-storage
10) RT PREEMPT PATCH: https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/preemptrt_setup
11) Message queues: http://www.cs.albany.edu/~sdc/CSI500/Downloads/p110-davari.pdf