ECEN 5623 Exercise #3 Threading and Real-Time Synchronization

Course Name: **Real-Time Embedded Systems**
Submission Date: **July 08th 2019**
Student Name: **Rushi James Macwan**
Board Used: Raspberry Pi 3B+

Note: Correct answer is in Blue Font

Credits: *All codes utilized for completing this assignment have been reused and modified accordingly. The codes are originally authored by* ***Dr. Siewert*** *and is* ***NOT*** *my original work. All code fragments have been reused and duly credited here for its reuse. Appropriate code explanations have been made wherever required.*

---

1) [10 points] 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?

Sol.

Three priority inheritance key points made in the paper:

The points discuss synchronization problems in multi-threaded applications. It focusses on solutions for preventing unbounded priority inversion as described in the paper. Also, it focuses on types of blocking, schedulability and future work incentives as given in the paper.

1. The paper talks about the need of **synchronization in multi-threaded applications**. To maintain the fidelity of critical logical and physical resources shared by various threads of execution in a RT (real-time) environment, the use of synchronization primitives like semaphores, monitors, or Ada rendezvous is used. Often, these primitives give rise to **blocking** where a high-priority job is indirectly or directly pre-empted by a low-priority job in a RT environment. The wide effects of blocking can make the system miss critical deadlines which can be catastrophic in most hard RT-systems. To prevent such harsh failures, the paper provides solutions for a RT-system that has priority-driven pre-emptive scheduling features.

   The principle idea behind the solution is the implementation of priority inheritance protocols. This works with the basic idea that when a low-priority job (task/thread) executed by a RT-system that has priority-driven pre-emptive scheduling features, enters a critical section, it essentially blocks a high-priority job that attempts to use the same critical section. This happens because the low-priority job locks the critical resources through the use of synchronization primitives like semaphores. While the critical resources remain locked by the low-priority job, the high-priority job must wait for the resources to be available. However, often than not, medium-priority jobs may pre-empt the low-priority job while it is executing its critical section, for which the high-priority job is blocked. The medium-priority job therefore runs until its completion and since there can be more than one medium-priority jobs that are assigned a priority greater than the pre-empted low-priority job, the pre-emption for the low-priority job is indefinite. As a result of that, the high-priority job is blocked indefinitely and an arbitrary blocking takes place where a high-priority job is indirectly pre-empted by a job assigned a lower priority. This situation is called **unbounded priority inversion** (as the

blocking is indefinite and therefore unbounded priority inversion) that the below two points revolve around, as can be found in the paper.

2. To ***prevent unbounded priority inversion***, the paper introduces two priority inheritance protocols, called the *basic* ***priority inheritance protocol*** and the ***priority ceiling protocol***. To prevent unbounded priority inversion, the job that has entered a critical section should not be pre-empted. This works when the critical section is short enough. But, when it is long, chances are that a high-priority job which does not need that critical section may actually be blocked and can result into the same catastrophic results as indefinite blocking in hard RT-systems.

   The ***basic priority inheritance protocol*** is therefore designed such that when a high-priority job is blocked by a low-priority job (because the critical section required by the high-priority job is locked by the low-priority job), to prevent unbounded priority inversion, the low-priority job ignores its previous priority assignment. Instead, the low-priority job now inherits the highest priority of all the jobs that it is blocking because of its locked critical section execution. This enforces that no other medium-priority job pre-empts the low-priority job that is executing a critical section which is requested for execution by a blocked high-priority job. Thus, ***unbounded priority inversion*** is prevented.

   The basic priority inheritance protocol is ***sufficient*** to prevent priority inversion. But, it is pessimistic and it overlooks some of the other problems like ***deadlocks and chained blocking*** that can easily take place. The ***priority ceiling protocol*** on the other hand prevents both of the above issues. Plus, it ***reduces the worst case task (job) blocking time*** to at most the duration of one subcritical section. This is possible because the priority ceiling protocol allows a job to pre-empt the critical section of another job and run its own critical section while its priority is guaranteed to be higher than the ones inherited by all the pre-empted critical section jobs. This ensures that chained blocking is prevented and potential deadlocks are prevented. This is because the priority ceiling protocol allows a high-priority job to run a critical section that is available and is yet of higher priority than the highest inherited priorities of all the blocking critical sections. This is possible because each critical section semaphore is assigned the priority ceiling, meaning – the highest priority of all the jobs that can use that semaphore. Thus, data fidelity is maintained in synchronized RT-systems where serialized access to the critical resources is a must and yet sharing of the system resources is managed such that unbounded priority inversion is prevented in addition to chained blocking and deadlocks. Since the semaphores are assigned priority ceiling in the priority ceiling protocol, the blocking time is reduced to that of one subcritical section.

3. Finally, the paper talks about ***schedulability, types of blocking and some future work incentives***.

   Blocking can be of three types:
   a. ***Direct blocking*** where a high-priority job requires an inaccessible critical section locked by a low-priority job.
   b. ***Push-through blocking*** where a medium-priority job takes over the system resources from the low-priority job simply by the privilege of the assigned priorities while a high-priority job remains blocked.
   c. ***Ceiling blocking*** where although a high-priority has access to the required critical section, it cannot access it because a group of semaphores have priority ceiling which allows the low-priority job to seize control of all the semaphores (locked or unlocked) with the same priority.

   For ***schedulability***, the paper shows how periodic jobs following the priority ceiling protocol can be scheduled according to the rate-monotonic algorithm. Based on the worst case blocking time and the period of the jobs, according to the rate-monotonic algorithm it is proved in the paper how a set of jobs can meet all the deadlines and can be scheduled.
   As for the ***future work incentives***, the paper discusses how priority floor condition under the priority limit protocol eliminates ceiling blocking along with the benefits of the priority ceiling protocol. In addition to that, the job conflict protocol provides better responses for RT-systems than the priority limit and priority ceiling protocols.

Why the Linux position makes sense or not:

The **PI-FUTEX provides a good RT-system service** by eliminating the need of extensive system overload because of dynamic or run-time scheduling and the associated latencies. As both the Linux Torvalds and Ingo Molnar take their position, I stand by it that PI-FUTEX provides a reasonable solution to offload the scheduling overhead inefficiencies from RT-Linux. With the use of PI-FUTEX which is light in terms of processing requirements as it does not require kernel involvement except for the arbitration when required, the number of PI-FUTEXES used can become quite large without significant overload on the system. This counts for a more deterministic system response as opposed to the scheduling latencies leading to a failure of suppressing preemptions in scenarios where unbounded priority inversion is prevalent RT-systems. With the efficient use of PI-FUTEXES, deterministic system responses can be obtained as it allows for swift and deterministic priority inheritance protocol implementation that in the best case, provides a deterministic view of how the system will respond and the associated well-bound latencies. In the worst case, PI-FUTEXES will improve the statistical distribution of locking related application delays and therefore brings the Linux system closer to a more RT-OS. I therefore stand by the position that PI-FUTEXES and the RT_PREEMPT patch actually makes the Linux RT-OS more suitable for soft RT-services.

Reasoning on whether FUTEX fixes:

The **FUTEX fixes the problem** of unbounded priority inversion using the help of priority inheritance protocols. As the referenced source points out: the PI-FUTEX is lightweight and does not require massive kernel involvement. It does not need kernel calls but simply uses fast atomic operations in the user-space. The in-kernal PI-FUTEX implementation is simple such that only one owner can own a lock and only the owner can unlock that lock. This allows for a robust priority inheritance implementation. The user-space locks do not allow disabling interrupts or changing a task into a non-pre-emptible one. PI-FUTEXES provide good determinism against the problems of priority inversion. This is because without PI-FUTEXES, the kernel cannot guarantee any deterministic execution of a high-priority blocked thread (by a low-priority thread). Any medium-priority task can preempt the low priority thread while it holds the lock and therefore the system provides a non-deterministic response. However, with FUTEX, the fix is possible as it allows for a deterministic system response with offloading the latencies involved with kernel execution. Since, it is lightweight, uses atomic operations, it provides a quick priority inheritance scheduling response and prevents the dangers of priority inversion.

Courtesy: Reference Link: [1]

2) [15 points] 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).

Sol.

Thread safe functions that are re-entrant:

Functions that are thread-safe and are reentrant are essentially prevalent in multi-threaded systems. Such a function is designed such that multiple threads can utilize and execute that function at one or more

instances during the execution life of the system. These functions must provide correct response to each of the thread calls. Thread-safe functions that are reentrant have the following three characteristics:

1. To prevent data infidelity and corruption, they must associate unique copies of global variables for each thread call (execution).
2. They must provide mutually exclusive access to the critical sections for the multi-threaded application by providing MUTEX protection for global data.
3. They may completely avoid the use of global variables and use only stack resources for its data handling. A pure function utilizes the same principle by implementing only stack resources for its data storage and calls only other pure functions.
4. If a function is reentrant or is a pure function, it is thread-safe. Meaning, it is primarily protected from data corruptibility by enforcing unique data copies for the thread calls or by only using the mutually exclusive data access mechanism of which stack is an example.

To summarize, a thread-safe function that is reentrant can be interrupted, called by another function (or thread) at one or more instances and provides correct response to each call. They ensure this by keeping either a mutually exclusive access to the shared critical resources or by completely eliminating the use of it (i.e. using stack only).

Description and Impact of each of the methods to real-time services:

The below section explains the three ways to implement a thread-safe function that is reentrant:

1. Pure functions only using stack and no global memory:

Description:

Pure functions that only use stack are thread-safe and reentrant but with the condition that it only calls other pure functions. Such pure functions that use only stack and no global memory are free from data infidelity/corruption issues in multi-threaded environments. This is because when more than one thread is calling a pure function, the data pertaining to each thread execution is stored separately in the stack. Because stack uses the LIFO (last in first out) principle, there is a mutually exclusive data handling for pure functions that use stack memory. When more than one thread uses a pure function, it calls the pure function but with a different background memory that is stored in the stack. In simple words, the execution of different threads utilizing the same pure function is completely thread-safe and mutually exclusive to each other and therefore they provide correct response, which means pure functions are both thread-safe and reentrant.

Impact to real-time services:

RT-services can have a lot of benefit from the use of thread-safe and reentrant pure functions. As previously mentioned, there is data consistency due to the mutually exclusive nature of the thread execution. On the downside, pure functions can only call other pure functions. Therefore, RT-services cannot be always built effectively with the use of pure functions as they do not communicate or share execution resources with a function other than a pure function. Speedy, simple and highly deterministic multi-threaded soft RT applications requiring a minimal amount of memory, can rely on pure functions for reliable and error-free RT services. However, there is also a second downside that stack can only be accessed in the LIFO order and therefore in complex hard RT-system implementations, there is a chance of potential deadlock/blocking or unbounded priority inversion when a thread must wait to access the critical information stored in the stack in the LIFO order. This can lead to catastrophic results if the deadlines are missed or the service is not deterministic.

2. Functions using thread indexed global data:

Description:

Functions using thread indexed global data are the ones that create separate and unique copies of data in the global memory pertaining to each thread execution. This ensures that multi-threaded applications can access the unique and error-free global data in RT at all instances – unlike the pure functions where the stack responds in a LIFO order. Since there is a separate and unique copy of data for each separate thread execution, these functions are both reentrant and thread-safe.

Impact to real-time services:

For RT-services, functions using thread indexed global data can certainly add some redundancy to the system. This can cost in terms of memory space required, the power and capabilities required by the I/O to access multiple copies of data in RT for different threads of execution. Also, it requires some amount of higher processing abilities as one or more threads have the capability in RT to access the thread indexed global data unlike the stack that follows a LIFO order. This requires a better scheduling policy that can prevent blocking and unbounded priority inversion. However, with sufficient system resources, thread indexed global data can provide the best hard RT-service in complex multi-threaded environments where missing deadlines can be catastrophic (e.g. flight control systems).

3. **Functions using shared memory global data but enforcing a mutually exclusive access to it by the use of MUTEX wrappers:**

Description:

Functions using shared memory global data for but enforce a mutually exclusive access to the data for multi-threaded applications require the use of MUTEX to prevent data corruption and inconsistency. These functions are thread-safe and reentrant as they can be called by multiple such functions and can be interrupted and re-serviced from the point where it was dropped off in its execution cycle. With the use of MUTEX and sufficient priority inheritance mechanisms with efficient priority-preemptive scheduling can provide service to multiple threads in RT.

Impact to real-time services:

Functions that use shared memory global data, showcase a significant reduction in the data redundancy and memory requirements of the RT-service. In addition to that, it saves more processing power that is instead spent on creating separate and unique copies of the data for multi-threaded execution in RT-services. These functions are much friendlier to RT-services that are optimal and require a moderate amount of capabilities. They reduce power consumption, processing & memory needs and sometimes, I/O needs as well. On the other side, these functions will require a stronger scheduling policy to prevent system failure that can prevent unbounded priority inversion, chained blocking and deadlocks. The design of such functions are more prone to these qualities in RT-services as there is only set of copy of global data pertaining to the function for its execution across multiple threads of execution. If a high-priority thread requires an access to all or most of the data of the function while a low-priority has locked an access to some of the required data of the function, then necessary scheduling policies like the priority ceiling protocol must be implemented to prevent the previously mentioned conditions of system failure.

Correct shared state update code:

```
pi@raspberrypi:~ $ sudo nano prob2.c
pi@raspberrypi:~ $ gcc prob2.c -o exefile -lpthread
prob2.c: In function 'pthread_counter':
prob2.c:72:9: warning: implicit declaration of function 'usleep' [-Wimplicit-function-declaration]
         usleep(3000000);
         ^~~~~~
pi@raspberrypi:~ $ ./exefile
x = -150, y = 200, z = 3
Acc = 10
Roll = 50
Pitch = 0
Yaw = 215
Mon 2019-07-08 23:24:47 MDT
TEST COMPLETED SUCCESSFULLY
x = 1975514224, y = 200, z = 3
Acc = 10
Roll = 50
Pitch = 7
Yaw = 202
Mon 2019-07-08 23:24:50 MDT
TEST COMPLETED SUCCESSFULLY
x = 1967121520, y = 198, z = 9
Acc = 20
Roll = 46
Pitch = 14
Yaw = 189
Mon 2019-07-08 23:24:53 MDT
TEST COMPLETED SUCCESSFULLY
x = 1958728826, y = 197, z = 12
Acc = 25
Roll = 44
Pitch = 21
Yaw = 176
Mon 2019-07-08 23:24:56 MDT
TEST COMPLETED SUCCESSFULLY
x = 1950336122, y = 196, z = 15
Acc = 30
Roll = 42
Pitch = 28
Yaw = 163
Mon 2019-07-08 23:24:59 MDT
^C
pi@raspberrypi:~ $
```

The above code showcases how thread 1 updates the navigation data along with the current real-time and thereafter thread 2 will read and post these values. There is a **100 milli-second** gap between the updating of thread-1 values every time. Thread 2 efficiently and continuously reads thread-1 updates and efficiently writes them back. Thread-2 will put out the values based on the values that are updated and changed every 100 milli-seconds by thread-1.

3) [15 points] 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?

Sol.

Code explanation:

```
pi@raspberrypi:~ $ sudo nano prob_3.c
pi@raspberrypi:~ $ gcc prob_3.c -o exefile -lpthread
prob_3.c: In function ‘grabRsrcs’:
prob_3.c:90:8: warning: implicit declaration of function ‘usleep’ [-Wimplicit-function-declaration]
        usleep(10000);
        ^~~~~~
prob_3.c: In function ‘main’:
prob_3.c:142:9: warning: implicit declaration of function ‘strncmp’ [-Wimplicit-function-declaration]
      if(strncmp("safe", argv[1], 4) == 0)
         ^~~~~~
pi@raspberrypi:~ $ ./exefile
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
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 1 got A and B
THREAD 1 done
Thread 1: 76e2f470 done
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: 7662e470 done
mutex A destroy: Success
mutex B destroy: Success
All done
pi@raspberrypi:~ $
```

As can be seen in the code, both the threads primarily check if both the critical resources are available and are unlocked. If not, they will perform a random back-off. Due to the time different in the random back-off, one of them, thread 1 in this case will complete its execution prior to thread 2. Essentially, in the back-off, the threads will release the locks and will wait for a certain cooling period after which they will reattempt to lock the resources.

Demonstration and description of deadlock with threads:

Description:

Threads of execution can end up being in a deadlock when the required system resources are not available at a given time. This can happen because of one or more of the following detailed reasons:

a. If a thread is waiting for a system I/O response from the external environment which can be an indefinite event.
b. If a thread requires access to critical resources (in the stack memory, or global data memory that is protected for consistency by only allowing mutually exclusive access with the help of MUTEX wrapper) that are not available (and the blocking may be indefinite) while one or more of other threads depend on the thread waiting for the critical resources.
c. If a thread has a partial access to a set of critical resources while one or more of other threads requiring complete access to the critical resources have only acquired partial access. The threads in this case wait for a timeout for the resources to available to one of them based on the scheduling policy.

Demonstration:

In the code, it can be seen that the deadlock is eliminated using a random back-off scheme. This scheme essentially requires the threads/processes to release the locked critical sections while it also stops attempting to lock a critical section when there is a potential system deadlock.

In the code, there are two threads and two critical resources, A and B. For a complete execution, each thread is required to have an access and be able to lock both the A and B resources simultaneously. While the same is true, each thread only acquires one of the two critical resources and contributes to the potential deadlock. To prevent this, each thread execution primarily ensures if both of the critical resources are available for moving forward. When only one of the resources is available, the thread unlocks both the resources and backs-off for a random period of time. After the timeout, the thread reattempts to ensure if both of the resources are available. If that is true, the thread will continue its execution sequence by locking the critical resources one after another. While this is true, the code is designed such that the random back-off times will allow a grace period after a deadlock during which only one of the two threads will be able to reattempt ensuring that the resources are both available. If they are available, the thread grabs both the resources, completes its execution and unlocks the resources so that the other thread can continue to attempt the same. This way, only one of the thread will see if both the resources are available at a time and will attempt to complete its execution while the other will continue (because of the back-off scheme) to wait for a random time until all the resources are once again available.

Demonstration and description of priority inversion with threads:

Description:

When a high-priority task requires access to critical resources that are locked by a low-priority task, priority inversion takes place. The high-priority task is blocked in this case and must wait until the low-priority task has executed the locked critical section. Under this condition, one or more medium-priority tasks that do not require an access to the locked critical section may preempt the low-priority task and that can result into a priority inversion. In such circumstances, a medium-priority task indirectly blocks the high-priority task and therefore it is a case of priority inversion.

Demonstration:

Priority inversion can be eliminated through the use of proper inheritance protocols as mentioned in the paper referred for problem #1. The priority ceiling inheritance protocol will help avoid unbounded priority inversion while also reducing the blocking time, preventing deadlocks and chained blocking. This can be demonstrated by designing a scheduling policy using heuristic algorithms such that critical section semaphores acquire the highest priority of all the jobs that will attempt to lock it. That way, whenever a critical section is accessed by a low-priority task, it only preempts tasks that have lesser or equal priority than the highest-priority task that can attempt locking the critical section. This way, if a high-priority task with a priority greater than the priority ceiling of the semaphore attempts to access a separate critical section, it will be allowed to lock the low-priority critical section execution and will prevent unbounded priority inversion. Moreover, this policy will also ensure that the blocking time is reduced to just one subcritical section execution time.

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

Led by Ingo Molnar, RT_PREEMPT Patch was added to Linux as it did not have RT-capabilities. To make it a more like a RTOS, a RT_PREEMPT Patch was added to the Linux configuration. It essentially changed the system kernel permissions by taking away the spin-lock configurations that can lead to blocking and unbounded priority inversion. Instead, it implements the use of mutexes with priority inheritance scheduling and allows kernel preemption while rewriting the interrupts as individual threads with the appropriate priorities to make it more RT.
As Paul McKenney mentions in his work, the patch maximizes the amount of kernel code that is pre-emptible. According to his remarks, the patch is more of an addition of a new CPU to the system with the use of normal locking mechanisms (i.e. mutexes). Based on his underlying work, the RT_PREEMPT patch can allow Linux to behave like a very soft RTOS. The following points focus on the same:

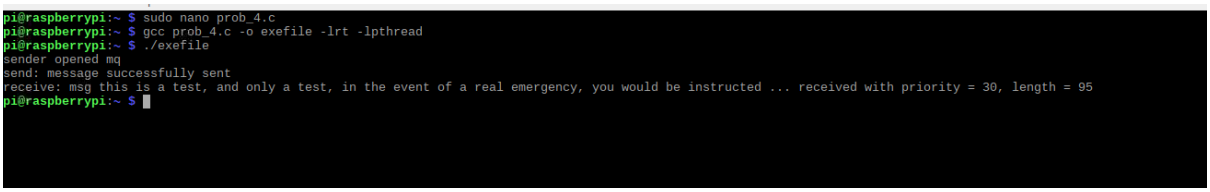
- As part of the patch, the **critical sections of the kernel are designed to be pre-emptible**. This guarantees that the system can be modified with the required priority scheduling policies to serve in hard RT-service applications where missing deadlines for high-priority tasks can often lead to catastrophic outcomes. However, the pre-emptible critical sections can actually be run on a different CPU core to balance the system load in a hard RT-environment.
- The use of **pre-emptible interrupt handlers and pre-emptible interrupt disable code sequences** allows for a much safer RT-system response. This pre-emption features protect the Linux RT-system from potential race conditions where the low-priority interrupts produce a race condition with important tasks. The pre-emptible RT-system features as available in the patch can help avoid and fix issues of blocking and/or unbounded priority inversion.
- As mentioned by Paul, his work claims that unbounded priority inversions can take place in RT-systems but the Linux Patch attempts to address this challenge with the use of two methods: **suppressing preemption** and **priority inheritance**. However, RT patch in Linux is not capable of suppressing preemption completely due to the impact of scheduling latencies, which can be observed with the Linux system where there is a minor latency which is sufficient for not letting the system suppress preemption. On the other hand, priority inheritance prevents this feature but due to its very **transitive nature** can often lead to unbounded or indefinite **chained blocking**. This is because if tasks A, B, C and D are in the ascending order of priority and if C has been blocked by A because C cannot access the critical resources that A has locked, A inherits the priority of D and continues to execute its critical section at that priority. Now, if D requires the critical section, then A will complete the critical section at D's priority and so will C after that. In the meanwhile, if E with the highest priority requires the same critical section, it will have to wait until all A, C and D have completed using the critical section although the inherited priority will be the highest. So, this adds to **chain blocking** which can take place in the RT_PREEMPT Patch for Linux. So, although due to its scheduling latencies, Linux can switch from **suppressing preemption** to **priority inheritance**, there can be a consistent amount of chained blocking **because of which the RT patch for Linux IS NOT SUITABLE for hard RT-services where the consequences of missing a deadline can be catastrophic.**

Courtesy: Reference Links: [1] [2]

4) [15 points] 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?

Sol.

Demonstration of message queues on R-Pi 3B+ adapted from examples:



In this code, a message queue is successfully opened and receives the user specified message. Once the message is successful, with a priority and length. Once the message is received, the code is terminated.

The code has been completely reused and unmodified for this problem which is owned by Dr. Siewert. The code can be found at this link:

http://ecee.colorado.edu/%7Eecen5623/ecen/ex/Linux/code/POSIX-Examples/posix_mq.c

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

Message queues can be a good solution to global memory sharing issues in RT-services. Global memory sharing for functions that are called by threads in a multi-threaded application can often lead to blocking. This is because a thread may require an access to the latest data across the shared global memory pertaining to a function or a group of functions that two or more threads may call. It can happen that one of the threads may be required to wait until other thread(s) update the shared global memory. An example of this can be the positioning and navigation system in a space craft. If the thread providing the space coordinates has only partially updated the location coordinates, and if the navigation system tries to read a combination of these coordinates containing new and stale information, the thread processing this inconsistent data can provide false responses and the system can end up in complete failure which can be catastrophic or hard RT-systems.

Message queues can prevent this sort of an execution nature by allowing the threads to share required information over the message queue channel. Instead of blocking the execution of one or more threads sharing a critical global memory space for acquiring complete access to the resource or partial that is consistent and RT, systems can use message queues. With the help of message queues, threads that are normally blocked to acquire response from other processes or threads can instead send out a request to read or a request to write as per the requirement instead of blocking the system and causing potential unbounded priority inversion. Message queues can thus increase RT-system reliability and responsiveness and can also simultaneously prevent blocking and/or unbounded priority inversion. Message queues can also contribute to an increase in the CPU utilization while reducing the wastage of CPU resources due to scheduling and/or management inefficiencies.

On the downside, message queues can have its own limited capability to enhance to a RT-system. This is because it depends on the design of the message queue regarding its response time. It is more along the lines of the stack memory usage. While a combination of message queues and shared global memory can be more ideal, message queues may not always help increase the efficiency and reliability of the RT-system while reducing the blocking. Again, this is due to the design of the message queue. If the design is simple, for moderately loaded RT-systems with optimal global memory space, the RT-system responses can be enhanced with the use of message queues as it reduces the blocking. However, with a large number of processes/threads and limited global memory space, the use of message queues does not help much with the enhancement of the RT-system responses. However, even if the message queue is full, the system can use separate buffers to notify a write/read wait state for a particular thread. Thus, message queues provide significant benefits to RT-systems but the system enhancement cannot be guaranteed as it heavily depends on the design and size of message queues, the availability of shared global data, the number of system services/threads/processes and finally, whether it is a hard/soft RT system which decides its overloading nature and consequences.

5) [30 points] 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 <time>" 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.

Sol.

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

If the system software causes an indefinite deadlock, there are multiple ways that the Linux WD timer can help with recovery from a total software loss. Essentially, it primarily depends on the underlying issues that lead to a system software deadlock. If the system software is irresponsive and is into a potential deadlock, it can be because of many reasons. The Linux WD timer can effectively address the issues arising out of software inconsistency. Whenever there is a potential system deadlock, the Linux WD timer will run tests and will often try to reboot the system if the tests detect an issue.

The Linux WD runs are not limited to but include some of these areas:

- Underlying hardware failure
- Network error
- System overload
- Data inconsistency amongst shared resources or thread executions and logical errors
- File inconsistency which is more along the lines of inconsistent data
- Failure of process monitoring
- Physical memory issues

When a system software deadlock occurs, the Linux WD timer might potentially look for problems with data inconsistency, memory issues, monitors the processes. It can even check for file errors that can potentially lead to a software deadlock. The system can also run into a deadlock with an overload of information from the system network because of malicious software or the bugs present in the software design. Software deadlocks can often be associated with system overloads due to the underlying software scheduling policies in RT-systems, as was the case with Apollo-11. The Linux WD timer can detect these issues by running one of its tests and if the tests are positive, the system can trigger its Linux WD timer which can reboot the system and reload the hardware drivers from a safe-point. A safe-point can be a point from where the execution can be error-free or the system is not overloaded while the system can guarantee that the hardware will be reset.

For an example, if there is a software deadlock due to a file system error, the Linux WD timer can perform file monitoring to check for any inconsistency with the file information like file age, activity on log files, incoming data, and etc. to see if the system functions are using the right file with the correct location and copy while it is possible that the system may have stored multiple copies with different variations. This can possibly prevent errors like out of process table space, memory errors, etc. The system can thus check the file consistency by verifying it with timestamps and the tests can potentially signal a software deadlock if the files are non-responsive. This can ultimately trigger the system to reset.

Thus, it is possible in many ways to use Linux WD timer to fix the software issues and potentially indefinite software deadlocks.

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

Please, refer to the screenshot available in the attached file.

In this code, there occurs a deadlock initially and thread 2 will wait for exactly 10 seconds. After that, it will fix the deadlock according to problem #2 code and will run the same data reading operation.