Real Time Embedded Systems – ECEN 5623

Summer 2019

Professor Sam Siewert

Report By: Poorn Mehta

Platform: Jetson Nano

Homework 3 (git: https://github.com/Poorn-Mehta/RTES/tree/master/HW3)

Note: All screenshots are uploaded on GitHub as well. Please refer to that in case higher quality is required. Also, due to time restrictions, I wasn't able to highlight important parts/key points. However, all the questions are addressed in here, and if it seems that that is not the case, please let me know. Thanks!

Q1: 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?

References:

http://ecee.colorado.edu/~ecen5623/ecen/rtpapers/archive/PAPERS_READ_IN_CLASS/prio_inheritence_protocols.pdf

http://mercury.pr.erau.edu/~siewerts/cec450/documents/Papers/soc-5.pdf

https://lwn.net/Articles/178253/

https://lwn.net/Articles/177111/

https://en.wikipedia.org/wiki/Futex

https://akkadia.org/drepper/futex.pdf

Solution:

Key Points by Paper

1. **Priority Inheritance is a very useful protocol**: Since priority inversion – especially with unbounded time delay can be devastating for real time systems, a protocol which can prevent that – is certainly useful. Moreover, it includes bounded and compile time calculable blocking times – increasing its utility.

2. **Priority Inheritance doesn't prevent deadlock by design**: As explained with examples in the paper, the priority inheritance is prone to deadlocks – if the code is not developed with utmost care. The deadlocks are quite harmful, since they reduce the system availability in the best-case scenario – and makes the system completely useless in worst situation.
3. **Priority Inheritance can have bounded, but significantly large blocking time**: The paper explains the concept of chain blocking, which has the potential of causing some disruptions in a real time system. Blocking time essentially means the time during which a higher priority task is blocked by a lower priority task – and as the system is developed with more and more services, it could be quite difficult to ensure that chain blocking is not going to create issues.
4. **Priority Ceiling is a better protocol – since it prevents deadlocks and chain blocking by design**: As noted above, the priority inheritance protocol is useful, but it does suffer from a set of problems. Priority ceiling protocol, however, removes them by its inherent design. This is significant, because even if designing a system which makes use of priority ceiling could be somewhat more difficult than the one with priority inheritance – the developers don't have to worry about deadlocks, and very large blocking times due to chain blocking.
5. **Priority Ceiling is efficient, especially since a process with higher priority than a specific semaphore being used in a critical section – can preempt that process**: This feature is important and useful, as not every critical section will prevent preemption and execution of every other processes. This way, system stability can be ensured – especially with the reliable handling of critical situations.

My Stand on Priority Inversion in Linux

- The stand by Linus Torvalds – which basically states that if your project code requires using priority inheritance (in other words, as a coder you've failed to ensure that necessary conditions for priority inversion don't get satisfied), then it is already broken – definitely makes sense; however, it seriously undermines the situations where it is very difficult to ensure and prove that those necessary conditions for priority inversion are completely absent.
- Moreover, when a project is large and being developed by multitude of developers, removing all portions of the project which are 'broken' from this perspective, is largely impractical.
- Often, there are situations where preemption in critical sections makes more sense, and is quite efficient to have in the system for most cases. For the sake of priority inversion only – if that has to be avoided, it would be somewhat less than ideal scenario. For these reasons, I actually support inclusion of priority inheritance/priority ceiling protocol in the official Linux kernel.

- It could be argued that if the priority inversion conditions can't be avoided, then probably it is a good idea to just switch to a dedicated RTOS, which has such in-built features – than to include priority inheritance protocol in the Linux kernel, to save it from additional complexity and probably some bugs due to the same. While it is a valid argument, I'd say that since Linux is already adopted by millions of developers and embedded systems around the world, this seemingly extra set of efforts would be worth it.
- As Ingo Molnar strongly opines, and progressively proves – that priority inversion is almost an unavoidable situation, especially when talking about programs that run primarily run in user space. Therefore, having a priority inheritance protocol support is – as per my thinking, fairly important for Linux.
- There are some points based on facts – that bolster this position, including, inability in Linux to make user space processes uninterruptable (kernel can always interrupt any of the user space process), almost impossible to write a fairly complex application without using locks/signaling mechanisms (such as mutexes, semaphores), and that not using multiple threads with varying priorities is not an option either – you end up at a point where having priority inheritance protocol support becomes very useful.
- When the shared hardware resources are inevitable, lockless design for sure fails – and therefore, again, it can lead to a point quickly where a developer would find himself/herself in a seemingly helpless condition – without any possible way to implement priority inheritance or priority ceiling protocol.
- Having stated these points, I'd like to once again clearly state that I absolutely support the idea of integrating priority inheritance/priority ceiling protocol into the main kernel of the Linux. However, it would be obviously a good idea to extensively test it intensively, and to also explore about its possible implications on completely unrelated code – and the kernel behavior when it is being used as well. Doing this would minimize the chances of kernel corruption/exception/misbehavior.

PI-Futex

- In my opinion, the PI-Futexes do provide protection from unbounded priority inversion, using priority inheritance protocol as described in the paper. However, using it is not easy, and suffers from drawbacks such as bounded but really long blocking time, no protection against deadlocks by design (although there is a timeout feature, it just reduces the intensity of the problem a bit).
- Clearly, Futexes are tricky in nature, and using them can't really be the first choice for a developer since they can cause more harm than good. A few points highlighting the same are described below.
- The kernel handles the actual physical addresses of the futexes. This however, is not clearly observable from the Futex() system call. Consequently, if multiple processes are referring to a futex which lies in the shared memory region – then they're referring to the same object. While this certainly is useful, lack of awareness of the same can lead to some design issues.

- The normal futexes are not real time-safe – as the kernel doesn't look through the list of waiters to find the highest priority thread. Again, this is only in normal futexes – not the PI Futex. While this may seem irrelevant, it should be noted that since the default behavior doesn't support real-time applications, developers would have to be extra careful while using them.
- At most – one thread can own the futex (at any time), and if the mutex is free – then it must be followed by either one or more threads requiring lock on it, or a completely empty list of the waiters for this same futex. These requirements additionally complicate the design.
- Moving on to the operation, PI Futexes have 2 possible paths – fast and slow.
- The fast path refers to condition where PI Futex was free, and was simply locked/taken by a process. In this path, there is no need for program to go into kernel space.
- The slow path refers to condition where a process tries to lock on to the PI Futex that is already acquired by some other process. In this, kernel takes over the control for a brief amount of time, implementing a priority inversion aware rt-mutex.
- Summarizing the operation – the kernel adds each process to the waiter's queue, and the owner keeps on inheriting priorities of all these blocked processes (in case of Sched_FIFO) – to provide a protection against priority inversion.
- It is obvious though, that this method is suffering from large blocking times due to chain blocking, and also, that it is susceptible to deadlocks.
- The timeout parameter of the PI-Futex prevents the complete failure of the system against deadlocks, but having deterministic timeouts is not something easy either.
- Therefore, I'd like to say that while PI-Futexes do prevent priority inheritance, they should be avoided if possible.

Q2: 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).

References:

https://www.geeksforgeeks.org/reentrant-function/

Solution:

Method 1 – Pure Functions (No Global Memory)

Description

- These are the functions which are said to be 'thread-safe' or 'reentrant' since they do not use any global variables/memory region, and therefore are protected against data corruption due to multiple threads running concurrently.
- It should be noted that generally, 'thread-safe' and 'reentrant' are two orthogonal properties of a function. Thread-safe refers to a function which will not alter its behavior if multiple threads are using it simultaneously – either on multiple cores at same time, or on same core at different times. Reentrant refers to a function which will not have any varied behavior even if multiple calls have been made to it, before finishing off previous executions. According to the context, these definitions can be slightly different, and it should be noted that the presented ones are more of an explanation of terms in general, than exact definitions.
- The concept is quite clear, and simple, avoid using global/shared memory/data/variables at all costs.
- If this condition is met, then each thread will be forced to store all the data (within the reentrant function) in its own stack, and since no other thread will have access to it – the data will be protected, even if the execution is interrupted by some other thread using the same function.
- To further visualize it – it can be said that each of the thread has its own 'copy' of the reentrant function, and that it works as a completely separate function, which is only going to be executed by that particular thread. Interrupts leading to executions of seemingly 'different functions' have no effect then.

Example – Screenshots

```
poorn@poorn-desktop: ~/Workspace/RTES/HW3/Q2/ThreadSafe_Reentrant/Simple_Example/src
poorn@poorn-desktop:~/Workspace/RTES/HW3/Q2/ThreadSafe_Reentrant/Simple_Example/src$ ls
M1  main.c  Makefile  obj  Thread.c
poorn@poorn-desktop:~/Workspace/RTES/HW3/Q2/ThreadSafe_Reentrant/Simple_Example/src$ ./M1

 >>>Program Start<<<

Thread 1 is Printing Value 51
Thread 1 is Printing Value 52
Thread 1 is Printing Value 53
Thread 1 is Printing Value 54
Thread 1 is Printing Value 55
Thread 1 is Printing Value 56
Thread 1 is Printing Value 57
Thread 1 is Printing Value 58
Thread 1 is Printing Value 59
Thread 1 is Printing Value 60
Thread 2 is Printing Value 49
Thread 3 is Printing Value 55
Thread 3 is Printing Value 60
Thread 3 is Printing Value 65
Thread 3 is Printing Value 70
Thread 3 is Printing Value 75
Thread 3 is Printing Value 80
Thread 3 is Printing Value 85
Thread 3 is Printing Value 90
Thread 3 is Printing Value 95
Thread 3 is Printing Value 100
Thread 4 is Printing Value 45
Thread 4 is Printing Value 40
Thread 4 is Printing Value 35
Thread 2 is Printing Value 48
Thread 2 is Printing Value 47
Thread 2 is Printing Value 46
Thread 2 is Printing Value 45
Thread 2 is Printing Value 44
Thread 2 is Printing Value 43
Thread 2 is Printing Value 42
Thread 2 is Printing Value 41
Thread 2 is Printing Value 40
Thread 4 is Printing Value 30
Thread 4 is Printing Value 25
Thread 4 is Printing Value 20
Thread 4 is Printing Value 15
Thread 4 is Printing Value 10
Thread 4 is Printing Value 5
poorn@poorn-desktop:~/Workspace/RTES/HW3/Q2/ThreadSafe_Reentrant/Simple_Example/src$
```

Example - Code Explanation

- I've written a simple code to further my understanding about this method. Please find the explanation of the same below.
- The code creates 4 different threads, but uses one common function, and doesn't wait for other threads to complete execution. In other words, the function execution is not atomic, nor sequential.
- It aims to prove the orthogonal concept of reentrancy and thread safe functions.
- The common function, changes the value of a variable i – which is initialized to a specific value at the top of function.
- Moreover, this function takes an integer in argument, and changes the value of that variable i – based on the integer value passed.
- Thread 1 aims to increase value of i by 1 in each step – taking total of 10 steps before exiting.
- Other threads work similarly, with only difference being: Thread 2 decrementing the value by 1 in each step, Thread 3 incrementing the value by 5 in each step, and Thread 4 decrementing the value by 5 in each step.
- From the output it is clear that even though the threads are not synchronized, nor given separate functions in the code, they all work without any issues – since the function is fully reentrant and preemptible.

Effect on Real Time Threads/Tasks

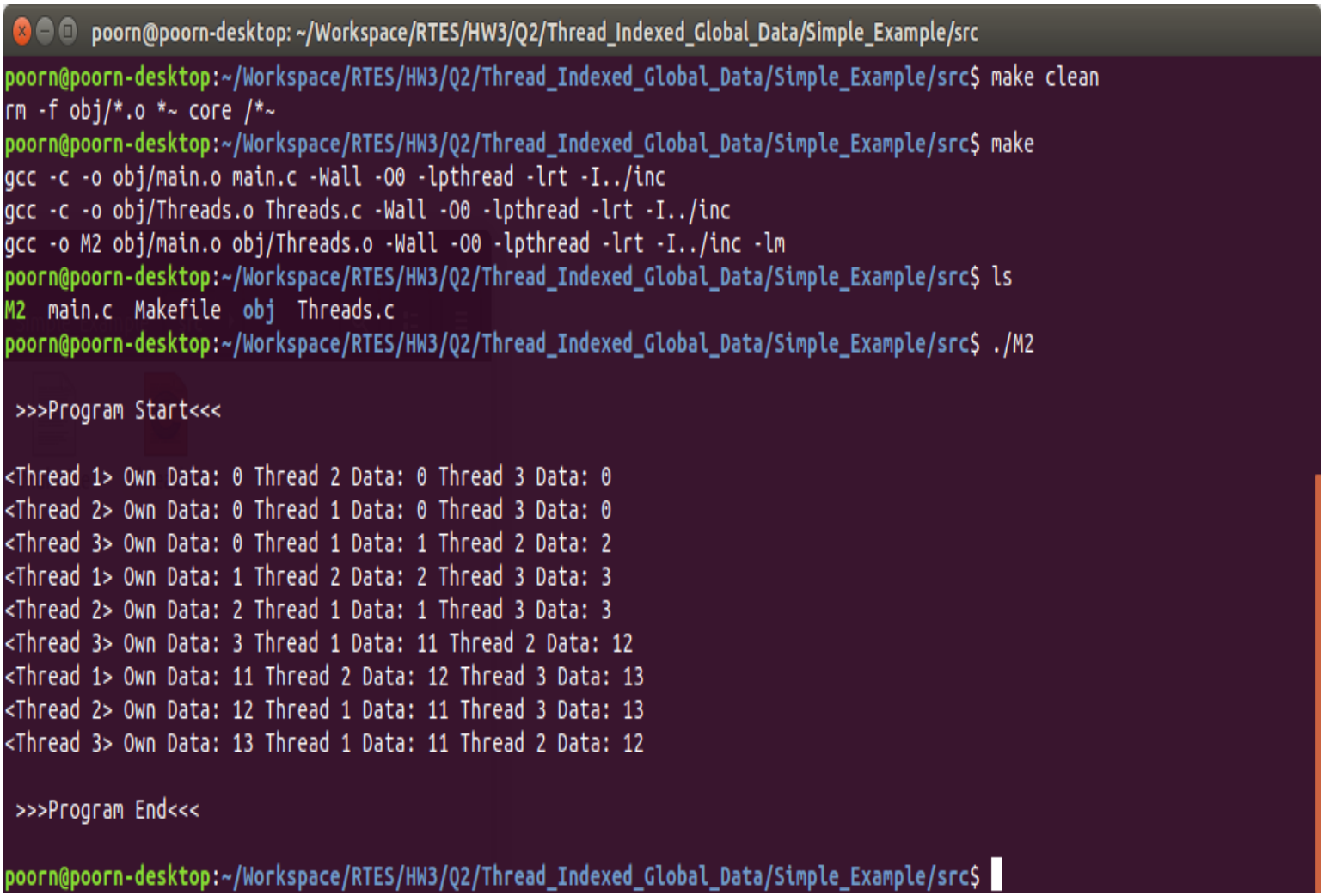
- Since real time threads have various priorities, and consequently preemption – non reentrant shared functions can lead to system instability quickly. Thus, replacing conventional functions (which utilizes global data), with reentrant ones is quite a good idea.
- Reentrant functions do not save much stack space when compared to writing individual functions, however, it does reduce the code complexity significantly. Moreover, code memory is saved as well – since retyping of same functions is not necessary.
- Priority based preemption is fully supported by this method, which can obviously be handy when designing a system that is having multiple threads with identical functionality, but real time constraints and priorities.
- Inter process communication can be done through function parameters and returns, however it is not easy to implement.
- One more thing that can be done to achieve a robust inter process communication – is to use message queues, between different functions which are completely self-contained (in the sense that they don't rely on anything that is out of scope for that particular function). This method has certain distinct advantages, clearly due to the nature and inherent architecture of message queues.

Method 2 – Thread Indexed Global Data

Description

- This method makes use of global variables – but in such a way that only one thread can have write-access to one specific global variable. Moreover, they are usually stored in contiguous memory locations – by the use of array.
- In this array, thread access their respective global variables by means of indexes. Therefore, the method is known as Thread Indexed Global Data.
- The clear advantage of this method is that of using global variables. Since they are write-protected from any other thread, there is no chance of corruption. Also, it makes data sharing between threads much easier – since there is no rule which prevents multiple readers from accessing a single global variable.
- Write and read access can be easily done through the use of pointers for each of employed threads, which makes the overall operation much more efficient.

Example – Screenshots



Example - Code Explanation

- I've written a simple code to further my understanding about this method. Please find the explanation of the same below.
- The code creates 3 different threads, uses a global array which is indexed by threads, to share data without any synchronization objects.
- It aims to prove the concept of thread indexed global data.
- There is a global array of integers, with length of 3 – each belonging to respective thread.
- These affiliations are hard coded, and can't be changed.
- To prevent one thread from writing global integers of other threads – it is handed a constant pointer to integer, which points to its own indexed element in the global array. This makes any change in the pointer value (the address of the variable) invalid. However, it can easily update it since the value it points to – isn't a constant.
- Moreover, to access other thread's global data, a point to constant integer is taken, and pointer is updated to travel through the global array.
- As it is evident from the output, even though all threads are using a common global array without synchronization, there is no data corruption.

Effect on Real Time Threads/Tasks

- If this method is used in real time systems, then in most cases it would be beneficial.
- Usually, synchronization objects (locks) are used to prevent data corruption of globally shared memory region, however they bring multitude of issues on table (as described in real time effects of using method 3). All of these can be avoided by simply using thread indexed global data.
- As it was pointed out by Steve Rizor, the single write – multi read technique to share data among threads – can result in cache line invalidations. This can harm the overall performance and reliability of the system.
- If a thread is updating its own globally indexed data, and if it is not being done atomically – then it can cause data corruption, given that preemption is enabled and a higher priority task interrupts the variable update procedure.
- There are not significant issues though, when it comes to implementation in real-time systems. Therefore, the use of such functions could actually be advantageous, if designed properly.

Method 3 – Mutexes

Description

- This is a widely used method to synchronize multiple threads, and to protect key resources against corruption – which occurs primarily due to concurrent execution.
- By use of appropriate APIs, an object is created – which can be locked only once – by a single thread.
- If any other thread attempts to lock this while it is already locked, the attempting thread is blocked till the original thread – which is holding the lock, unlocks it.
- The object can be locked and unlocked only through specific APIs, and they take care of blocking execution of a thread, as well preventing other threads from attempting to unblock a resource that is locked by some other thread.
- It is a pretty solid method – since it doesn't really let other threads interfere in so called 'critical section' – which is guarded by mutexes.
- However, it is prone to some error conditions which must be avoided by all means – such as deadlocks, priority inversion, etc.
- Developing a code based on mutexes, with fairly straightforward requirements is very robust, and easy.

Example – Screenshots (Part 1)

```
Jul  5 23:13:35 poorn-desktop HW3_Q1_M3[16883]: <301328.094us>Writer Exiting...
Jul  5 23:13:35 poorn-desktop HW3_Q1_M3[16883]: <301427.875us>Reader Exiting...
Jul  5 23:13:35 poorn-desktop HW3_Q1_M3[16883]: >>>>>>>>>> Program End <<<<<<<<<<
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: >>>>>>>>>> Program Start <<<<<<<<<<
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <490.054us>Writer Waiting to Acquire Mutex - Iteration(0)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <566.252us>Writer Started to Generate and Update Shared Structure - Iteration(0)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <582.085us>Writer Wrote Accel X: 60.501999
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <612.085us>Writer Wrote Accel Y: 62.129002
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <628.596us>Writer Wrote Accel Z: -87.023003
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <643.544us>Writer Wrote Roll: -83.516998
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <658.179us>Writer Wrote Pitch: 73.903000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <672.085us>Writer Wrote Yaw: 171.832993
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <736.252us>Writer Wrote Timestamp - Sec: 1562390060 Nano Sec: 539126364
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <749.377us>Writer Completed Writing New Data - Iteration(0)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <884.638us>Reader Started Waiting for Mutex to be Unlocked - Iteration(0)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <944.118us>Reader Acquired Mutex - Iteration(0)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <957.711us>Reader Got Accel X: 60.501999
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <970.836us>Reader Got Accel Y: 62.129002
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <983.336us>Reader Got Accel Z: -87.023003
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <995.524us>Reader Got Roll: -83.516998
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <1007.764us>Reader Got Pitch: 73.903000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <1020.107us>Reader Got Yaw: 171.832993
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <1037.139us>Reader Got Timestamp - Sec: 1562390060 Nano Sec: 539126364
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <1049.170us>Reader Completed Latest Read - Iteration(0)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <100894.711us>Writer Waiting to Acquire Mutex - Iteration(1)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <100946.688us>Writer Started to Generate and Update Shared Structure - Iteration(1)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <100962.945us>Writer Wrote Accel X: -19.340000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <100975.289us>Writer Wrote Accel Y: -39.651001
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <100987.219us>Writer Wrote Accel Z: 23.646000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <100999.297us>Writer Wrote Roll: 13.476000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101010.188us>Writer Wrote Pitch: 67.120003
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101020.336us>Writer Wrote Yaw: 122.038002
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101030.750us>Writer Wrote Timestamp - Sec: 1562390060 Nano Sec: 639420866
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101040.336us>Writer Completed Writing New Data - Iteration(1)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101228.773us>Reader Started Waiting for Mutex to be Unlocked - Iteration(1)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101255.859us>Reader Acquired Mutex - Iteration(1)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101267.844us>Reader Got Accel X: -19.340000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101280.391us>Reader Got Accel Y: -39.651001
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101292.891us>Reader Got Accel Z: 23.646000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101305.602us>Reader Got Roll: 13.476000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101317.781us>Reader Got Pitch: 67.120003
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101330.125us>Reader Got Yaw: 122.038002
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101342.781us>Reader Got Timestamp - Sec: 1562390060 Nano Sec: 639420866
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101354.352us>Reader Completed Latest Read - Iteration(1)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201179.016us>Writer Waiting to Acquire Mutex - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201221.656us>Writer Started to Generate and Update Shared Structure - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201234.828us>Writer Wrote Accel X: -2.771000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201246.656us>Writer Wrote Accel Y: 42.146000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201258.594us>Writer Wrote Accel Z: -35.662998
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201270.625us>Writer Wrote Roll: -88.917999
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201282.953us>Writer Wrote Pitch: 79.102997
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201294.688us>Writer Wrote Yaw: 349.088989
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201305.688us>Writer Wrote Timestamp - Sec: 1562390060 Nano Sec: 739695837
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201316.031us>Writer Completed Writing New Data - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201477.391us>Reader Started Waiting for Mutex to be Unlocked - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201499.625us>Reader Acquired Mutex - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201511.094us>Reader Got Accel X: -2.771000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201522.703us>Reader Got Accel Y: 42.146000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201534.938us>Reader Got Accel Z: -35.662998
```

Example – Screenshots (Part 2)

```
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101010.188us>Writer Wrote Pitch: 67.120003
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101020.336us>Writer Wrote Yaw: 122.038002
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101030.750us>Writer Wrote Timestamp - Sec: 1562390060 Nano Sec: 639420866
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101040.336us>Writer Completed Writing New Data - Iteration(1)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101228.773us>Reader Started Waiting for Mutex to be Unlocked - Iteration(1)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101255.859us>Reader Acquired Mutex - Iteration(1)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101267.844us>Reader Got Accel X: -19.340000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101280.391us>Reader Got Accel Y: -39.651001
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101292.891us>Reader Got Accel Z: 23.646000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101305.602us>Reader Got Roll: 13.476000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101317.781us>Reader Got Pitch: 67.120003
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101330.125us>Reader Got Yaw: 122.038002
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101342.781us>Reader Got Timestamp - Sec: 1562390060 Nano Sec: 639420866
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <101354.352us>Reader Completed Latest Read - Iteration(1)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201179.016us>Writer Waiting to Acquire Mutex - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201221.656us>Writer Started to Generate and Update Shared Structure - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201234.828us>Writer Wrote Accel X: -2.771000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201246.656us>Writer Wrote Accel Y: 42.146000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201258.594us>Writer Wrote Accel Z: -35.662998
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201270.625us>Writer Wrote Roll: -88.917999
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201282.953us>Writer Wrote Pitch: 79.102997
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201294.688us>Writer Wrote Yaw: 349.088989
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201305.688us>Writer Wrote Timestamp - Sec: 1562390060 Nano Sec: 739695837
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201316.031us>Writer Completed Writing New Data - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201477.391us>Reader Started Waiting for Mutex to be Unlocked - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201499.625us>Reader Acquired Mutex - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201511.094us>Reader Got Accel X: -2.771000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201522.703us>Reader Got Accel Y: 42.146000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201534.938us>Reader Got Accel Z: -35.662998
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201547.766us>Reader Got Roll: -88.917999
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201559.516us>Reader Got Pitch: 79.102997
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201571.969us>Reader Got Yaw: 349.088989
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201583.750us>Reader Got Timestamp - Sec: 1562390060 Nano Sec: 739695837
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <201601.828us>Reader Completed Latest Read - Iteration(2)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301452.781us>Writer Waiting to Acquire Mutex - Iteration(3)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301493.906us>Writer Started to Generate and Update Shared Structure - Iteration(3)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301507.438us>Writer Wrote Accel X: 5.113000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301519.125us>Writer Wrote Accel Y: -29.759001
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301531.500us>Writer Wrote Accel Z: -84.586998
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301543.188us>Writer Wrote Roll: -62.708000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301554.719us>Writer Wrote Pitch: 56.098000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301565.594us>Writer Wrote Yaw: 177.522995
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301576.219us>Writer Wrote Timestamp - Sec: 1562390060 Nano Sec: 839966380
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301586.031us>Writer Completed Writing New Data - Iteration(3)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301729.500us>Reader Started Waiting for Mutex to be Unlocked - Iteration(3)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301752.094us>Reader Acquired Mutex - Iteration(3)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301763.781us>Reader Got Accel X: 5.113000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301775.250us>Reader Got Accel Y: -29.759001
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301787.375us>Reader Got Accel Z: -84.586998
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301798.344us>Reader Got Roll: -62.708000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301809.500us>Reader Got Pitch: 56.098000
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301820.312us>Reader Got Yaw: 177.522995
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301831.656us>Reader Got Timestamp - Sec: 1562390060 Nano Sec: 839966380
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <301842.312us>Reader Completed Latest Read - Iteration(3)
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <401726.344us>Writer Exiting...
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: <402046.156us>Reader Exiting...
Jul  5 23:14:20 poorn-desktop HW3_Q1_M3[16921]: >>>>>>>>>> Program End <<<<<<<<<<
poorn@poorn-desktop:/var/log$
```

Example - Code Explanation

- The explanation for the code – which is as per the requirements of this question, is given below.
- It aims to prove the concept of thread synchronization, using mutex locks.
- A global data structure is primarily shared between two threads: reader and writer.
- The main thread initializes the structure, creates and initializes mutexes, and sets up some other things required by the program.
- The writer thread starts up, acquires mutex and resultantly protects its critical section – in which it will write new values to the shared global structure.
- After filling it up with random data, and printing the same (for verification purposes), it releases the mutex.
- The reader thread meanwhile has started, and is waiting for the mutex to go ahead with its execution.
- As soon as writer releases the mutex, reader acquires mutex and locks its own critical section, in which it simply reads the values and prints them out.
- This operation is repeated a few times – along with some delay in both threads to simulate data generation/reading rates, to show that this method works flawlessly.
- From output, it is evident that mutex locks can be very useful to ensure that the globally shared and accessed data is never going to be corrupted.

Effect on Real Time Threads/Tasks

- Since mutexes bring the concept of blocking – in which a higher priority thread can be put in blocked state by a lower priority thread, real time systems can suffer from adverse effects of mutex locks.
- Another undesirable effect is that unwise use of mutexes in quite complex real time systems, can make way for unbounded priority inversion which could lead to catastrophic failures within a very short period of time.
- Unbounded priority inversion can be prevented by use of special version of mutexes – known as PI Futexes, however it's quite complex to do so.
- One more issue is deadlock. This is also a very serious problem since all involved threads in a deadlock, can't make any progress, and that they remain in this 'stuck' state unless it is forcefully resolved by some specific methods.
- Even though this method does open doors for many added complexities in the code, and potential critical errors – is really is very useful in real time systems. Primarily due to the fact that it is widely accepted as a standard, and there are multitude of tips and tricks available out there to remove the possibility of above-mentioned problems.

Q3: 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?

References:

https://lwn.net/Articles/146861/

http://ecee.colorado.edu/~ecen5623/ecen/ex/Linux/code/example-sync/

Solution:

Deadlock Example

Code Description and Explanation of the Issue

- Since only the deadlock – and how it happens, is required by the question, I will not be describing the methods (which are already integrated in the code) to ensure that deadlock doesn't happen – in detail.
- The code creates two threads, Thread A and Thread B – and two mutexes, Resource A and Resource B. Please note that names of variables might be different from what is written in here.
- Both threads then, when in unsafe mode (which ensures that they end up in a deadlock), starts execution without having any kind of delays.
- They each acquire a mutex as soon as they're launched – Thread A acquiring and locking Resource A, while Thread B acquires and locks Resources B.
- After this, they both sleep for some time – this is done to prevent 'race' condition. It can be removed by passing specific command line argument to the program. If race condition takes place, then one process will finish before the other has chance to acquire and lock a mutex, and drive the code in deadlock.
- Since they've both slept long enough, it is ensured at the moment that both threads have locked 1 mutex.
- After this, they both try to lock other resource – Thread A trying to acquire Resource B and vice versa, and since it already acquired – they end up in a situation where they can't take any action, and keeps on waiting forever.
- The safe execution of the code ensures that one thread always finishes, before creating another thread. This results in purely sequential flow and execution of the threads.
- The root cause for deadlock is that both threads are executing simultaneously, ensuring that both of them have separate resources acquired and locked – all of which are needed by all threads, and then calling APIs which never times out.

- The resource counters are used in the code to keep track of number of locks acquired on that specific resource. For the present version of the code, both will be 0 before any thread is created, and both will be 1 before any thread comes out of their sleep (after acquiring first resource).
-

Resolved Code Description and Explanation of Random Back off

- Out of many ways to resolve this, the random back off scheme is implemented.
- In this, the idea is to make both threads give up resources – if they detect that deadlock is about to take place.
- When the code is launched with backoff command line argument, both threads, after coming out of sleep (so first resource is already acquired) – and before calling API to lock other resource, they look at the resource count (of the one that they want).
- If it is non-zero, it clearly means that calling the mutex locker API now will definitely result in deadlock. Thus, the thread gives up its resource (that it acquired and locked earlier), and sleeps for a random amount of time (which is always different than the sleeping time for other thread) before acquiring the resource it released earlier.
- Since both threads implement this – it is ensured that at least one thread will be sleeping more than the other, and since the code is written in a way that after coming out of sleep – it acquires both resources right away – deadlock can't take place.
- It is called random back off – because which thread will complete first is random, as well as the amount of time to back off is random.

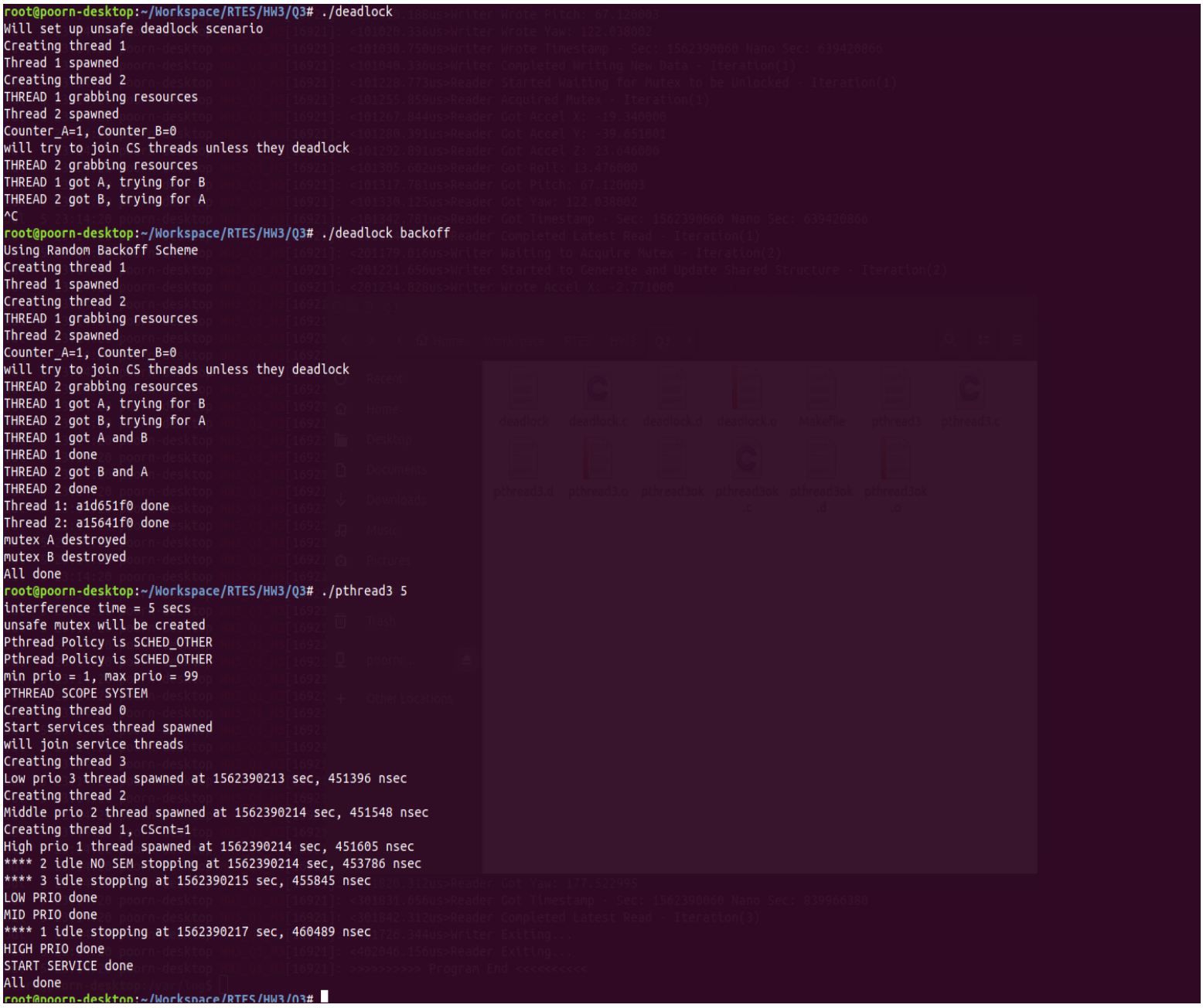
Unbounded Priority Inversion Example

Code Description and Explanation of the Issue

- Unbounded Priority Inversion is one of the most serious issues, when it comes to real time systems, with heavy use of shared resources and synchronization objects.
- The example code demos the same, by use of 3 threads – all at different priorities, and sharing a resource between lowest and highest priority threads.
- To implement this, first step is to setup Real Time environment. This is done by properly calling APIs to change scheduler, and priorities of threads.
- The High Priority and Low Priority thread works off of same reentrant function.
- This function acquires lock on a mutex, does some work, sleeps for 2 seconds, does some work again, and unlocks mutex.
- Middle priority thread only works for number of seconds that is supplied by command line argument.
- First, low priority thread is created and launched. After 1 second, middle priority thread is launched. Since scheduler is FIFO – preemption is on, and therefore, middle priority takes over CPU – before low priority thread has chance to complete processing and release its mutex.

- The higher priority thread is created and launched last, and it will be blocked since it can't acquire mutex. And since the middle priority keeps on running for the length of interference seconds – the middle priority task has introduced unbounded delay for higher priority task, even when these threads are not sharing any resource(s). Thus, it is effectively inversion of priorities.
- From screenshots it can be seen that the higher priority task is completing after middle priority task, which exemplifies the discussed problem.

Screenshots (Contains execution of all 3 discussed codes)



Unbounded Priority Inversion in Linux

- As it is evident from the example code's output, unbounded priority inversion can be quite problematic in real time systems.
- To prevent it while using locks, either priority inheritance or priority ceiling protocol has to be implemented.
- I am not sure what the term 'real fix' refers to her, however, there are methods to achieve this in Linux.
- One method is to use PI-Futexes, and the other is to use PREEMPT_RT Patch for the Linux kernel.
- Both of them implements priority inheritance protocol – which definitely solves the issue, but not without a few concerns of its own.
- PI-Futexes are described in the response to question 1, and PREEMPT_RT Patch is discussed below.

PREEMPT_RT Patch

- The PREEMPT_RT patch is introduced in Linux kernel – primarily to minimize the amount of kernel code that stays non-preemptible, and to minimize the amount of code that has to be changed to implement this – as well.
- This patch, adds many functionalities such as: Preemptible critical sections, interrupt handlers, and interrupt disable code sequences, priority inheritance for in-kernel spinlocks as well as semaphores, deferred operations, and some measures to reduce the latency.
- While all of these features are unique, and quite useful based on the context, the primary concern here is to discuss the measure that counters unbounded priority inversion.
- Since unbounded priority inversion can be quite dangerous to a real time system, there must be assurance provided in the project that prevents it from happening, if the necessary conditions are present (which makes unbounded priority inversion possible)
- There are a few ways to do that: (1) Make critical sections/semaphore locked portions of the code non-preemptible. (2) Use priority inheritance protocol. (3) Use priority ceiling protocol.
- Out of these, suppressing preemption is not always a good idea, since it can cause other issues which may result in multiple real time services missing their deadlines. Moreover, while it does make sense for kernel level spin locks to use this method, it's simply not suitable in the cases of semaphores. Blocking on a semaphore while waiting is one of the key characteristics, which also gives rise to potential priority inversion.
- Therefore, priority inheritance of priority ceiling comes in picture, to resolve the issue of unbounded priority inversion.
- The given patch makes use of priority inheritance protocol. The basic idea is that the task blocking others – temporarily (while holding the lock) inherits the priority which is highest in this group. Therefore, if there is any blocked task with higher priority than that

of the original priority of the current lock owner, to prevent inversion – the lock owner will receive a temporary boost in the priority to that of the highest priority blocked task.

- In the cases where locks are to be held for extended times, this patch introduces modified version of the same with some preemption points available throughout the execution – effectively prompting the current lock owner to drop it, if other higher priority task is seeking the same.
- Moreover, there is a peculiar issue with priority inheritance that can be seen most often in situations where a writer is transferring lock to a reader (when multiple readers are waiting). To address this, the given patch trades scalability for the solution.
- Additionally, the patch even provides the functionality by means of specific semaphores where, priority inheritance must be avoided.
- Along with all these, the generic features of priority inheritance are also included, such as: Transitivity (allows cascading priority inheritance, if multiple higher priority tasks are being blocked when a lower priority task is holding the lock), Timely Removal of Inherited Priority (the lower priority task quickly gets demoted back to its original priority), as well as Flexibility in various situations.
- It must be noted that although the PREEMPT_RT patch seems really good; it does suffer from a set of disadvantages – which are inherent to the architecture of the priority inheritance protocol itself.
- Therefore, even with the patch, the job of a developer is not made much easier. They should still be looking out to prevent deadlocks, very long (but bounded) time delays, etc.
- When it comes to RTOS vs Linux for Real Time Systems, I'd opine that this decision relies on many factors. For example, since Linux is much more widely adapted than any RTOS, it should be preferred when Hard-Real Time Services are either completely absent, or not too many compared to available resources, and other factors such as: time to market, flexibility, security concerns, robustness, etc. are carrying a significant importance. In contrast, if the most important thing is quite a few Hard-Real Time services, and the hardware is supported by a well-known RTOS, and when the other factors doesn't matter much – using RTOS makes much more sense.
- Generalizing, I'd say that for Soft Real Time Systems, Linux should be preferred, and for Hard Real Time Systems, a dedicated RTOS should be used instead.
- Based on priority inversion problem alone, it doesn't make sense at all to switch to RTOS for Soft Real Time Systems. For Hard Real Time Systems however, chain blocking can result in some critical services missing in their deadlines – and if that seems plausible by the first look of project designing, then switching to a RTOS would be better.

Q4: 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?

References:

http://ecee.colorado.edu/~ecen5623/ecen/ex/Linux/code/VxWorks-Examples/heap_mq.c

http://ecee.colorado.edu/~ecen5623/ecen/ex/Linux/code/VxWorks-Examples/posix_mq.c

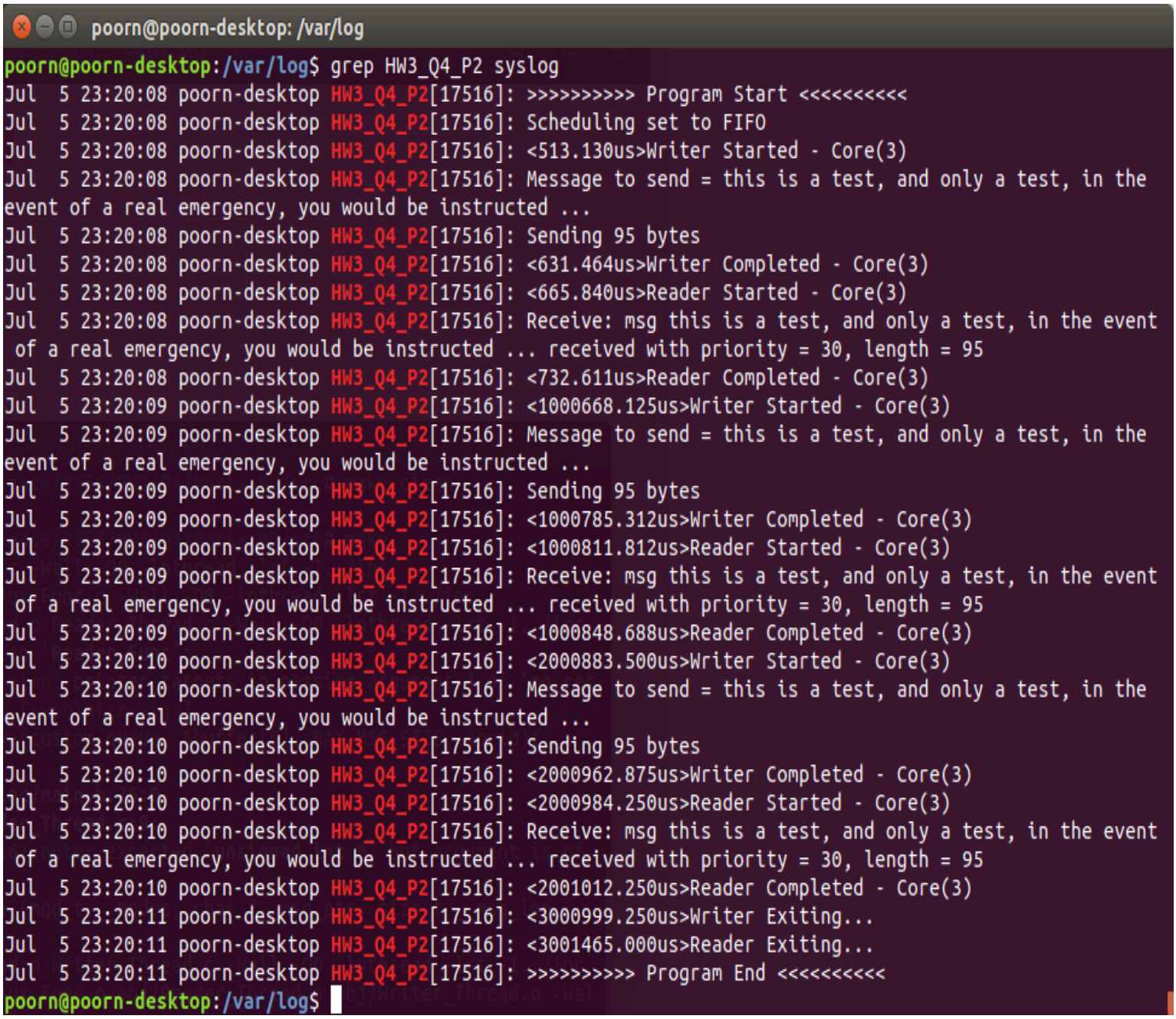
Solution:

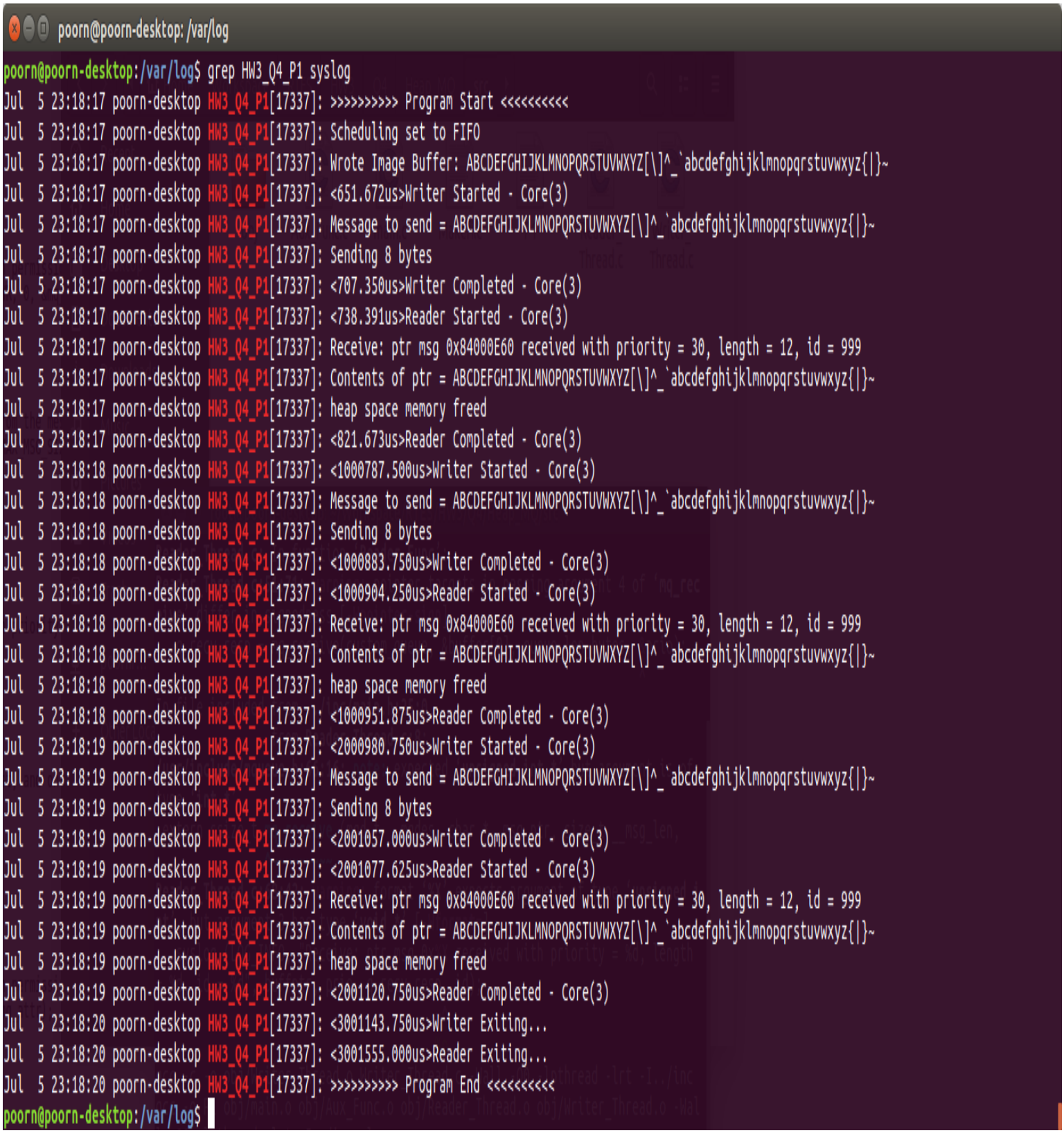
Heap MQ and Posix MQ – Code Description, Method Explanations, and Differences

- Both of these codes utilize POSIX Message Queues for Inter Process Communication, and therefore I will be explaining them first.
- POSIX Message Queues are very useful when it comes to reliable and fast IPC. There is a standard set of APIs available in Linux to implement them.
- In both codes, there are two threads – writer and reader, both of which, opens queues (however, they refer to the same object).
- Writer thread performs necessary actions, and sends the message to the queue, using mq_send() API. Reader thread on the other hand, blocks on mq_receive() waiting for a message to appear on the same queue.
- Once the message is sent/received, they both advances their executions. I have written the codes in a way that they perform this repeatedly for a few times, to show that they work without facing any issues, and finally exit.
- In the Posix MQ code, the entire message is shared directly – as it is, using the queues.
- While this is easier, it does take up quite a lot of space. Essentially, the queue length has to be more than or equal to the size of the message itself.
- It is fine for small messages, but in the case of large structures, it can end up eating up a large portion of stack, and introducing multiple issues thereafter.
- Therefore, another method of sharing messages is demoed using Heap MQ code.
- In this, instead of sending the whole message in the queue, only the pointer to a heap address is passed.
- This heap is acquired (using malloc) dynamically by one thread – which then writes desired data to it, and is freed by second thread – after it has dereferenced the pointer and used the data.

- This method is very efficient since only a few bytes have to be transferred (size of pointer to be precise).
- In Heap MQ however, one integer is passed apart from the pointer to heap, which just shows that you can pass pointer to main data region, and some other data in a single message.

Screenshots




poorn@poorn-desktop: /var/log
poorn@poorn-desktop:/var/log$ grep HW3_Q4_P1 syslog
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: >>>>>>>>>> Program Start <<<<<<<<<<
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: Scheduling set to FIFO
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: Wrote Image Buffer: ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: <651.672us>Writer Started - Core(3)
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: Sending 8 bytes
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: <707.350us>Writer Completed - Core(3)
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: <738.391us>Reader Started - Core(3)
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: Receive: ptr msg 0x84000E60 received with priority = 30, length = 12, id = 999
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: Contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: heap space memory freed
Jul 5 23:18:17 poorn-desktop HW3_Q4_P1[17337]: <821.673us>Reader Completed - Core(3)
Jul 5 23:18:18 poorn-desktop HW3_Q4_P1[17337]: <1000787.500us>Writer Started - Core(3)
Jul 5 23:18:18 poorn-desktop HW3_Q4_P1[17337]: Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Jul 5 23:18:18 poorn-desktop HW3_Q4_P1[17337]: Sending 8 bytes
Jul 5 23:18:18 poorn-desktop HW3_Q4_P1[17337]: <1000883.750us>Writer Completed - Core(3)
Jul 5 23:18:18 poorn-desktop HW3_Q4_P1[17337]: <1000904.250us>Reader Started - Core(3)
Jul 5 23:18:18 poorn-desktop HW3_Q4_P1[17337]: Receive: ptr msg 0x84000E60 received with priority = 30, length = 12, id = 999
Jul 5 23:18:18 poorn-desktop HW3_Q4_P1[17337]: Contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Jul 5 23:18:18 poorn-desktop HW3_Q4_P1[17337]: heap space memory freed
Jul 5 23:18:18 poorn-desktop HW3_Q4_P1[17337]: <1000951.875us>Reader Completed - Core(3)
Jul 5 23:18:19 poorn-desktop HW3_Q4_P1[17337]: <2000980.750us>Writer Started - Core(3)
Jul 5 23:18:19 poorn-desktop HW3_Q4_P1[17337]: Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Jul 5 23:18:19 poorn-desktop HW3_Q4_P1[17337]: Sending 8 bytes
Jul 5 23:18:19 poorn-desktop HW3_Q4_P1[17337]: <2001057.000us>Writer Completed - Core(3)
Jul 5 23:18:19 poorn-desktop HW3_Q4_P1[17337]: <2001077.625us>Reader Started - Core(3)
Jul 5 23:18:19 poorn-desktop HW3_Q4_P1[17337]: Receive: ptr msg 0x84000E60 received with priority = 30, length = 12, id = 999
Jul 5 23:18:19 poorn-desktop HW3_Q4_P1[17337]: Contents of ptr = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Jul 5 23:18:19 poorn-desktop HW3_Q4_P1[17337]: heap space memory freed
Jul 5 23:18:19 poorn-desktop HW3_Q4_P1[17337]: <2001120.750us>Reader Completed - Core(3)
Jul 5 23:18:20 poorn-desktop HW3_Q4_P1[17337]: <3001143.750us>Writer Exiting...
Jul 5 23:18:20 poorn-desktop HW3_Q4_P1[17337]: <3001555.000us>Reader Exiting...
Jul 5 23:18:20 poorn-desktop HW3_Q4_P1[17337]: >>>>>>>>>> Program End <<<<<<<<<<
poorn@poorn-desktop:/var/log$

POSIX Queues to Avoid Priority Inversion

- Since POSIX Queues are quite effective mechanism for Inter Process Communication – it can be argued that at least in some cases, mutexes can be completely replaced with the used of POSIX Queues.
- For example, if there is a set of global data – in form of shared structure, then instead of protecting reads and writes of it using mutex locks, they can be sent among the threads/processes that require access to it.
- However, it can lead to additional programming complexity – since each read and write will have to be implemented through specific API calls.
- Moreover, it has to be very clear that at what time – which process will send data to which process, and what will be the status of other process wanting to share it at the same time.
- This can be simplified by the use of multiple queues, and local copies of structures, but all of it combined – adds quite a bit of overhead.
- Especially compared to mutexes, this can rapidly complicate the overall system design, additionally so since queues bring a set of concerns of its own along with the use. For instance, length of queue (too short might result in overflow, especially for lower priority processes which are receiving many messages from higher priority processes, too high a length will eat up quite a lot of memory), length of message (having variable length of the queues is quite difficult, and thus optimizations are tough as well), responsibility to 'cleanup' (safely unlinking queues, and ensuring that unsafe older implementations are not able to distort the current execution), etc. often demand significant efforts to be taken care of.
- POSIX Queue operations are usually blocking – for example, mq_receive() blocks execution of the process until there is a message sent to it. Sure, there are timed versions of such APIs, but if not carefully designed – this could lead to large blocking times of high priority processes.
- Also, in some error conditions where sending a message to a particular queue fails, the blocking might be staying unaware of the same and – a middle priority queue might take over the CPU for execution. While this is not a typical priority inversion, it has to be taken in account since queues are more prone to errors than mutexes (it could be wrong, but I've worked with both for some time now, and this is purely based on my experience)
- It should also be noted that in some cases, it could be practically infeasible to replace all mutexes (which can cause priority inversions) with queues. This is particularly observable in nested locks.
- Having talked about all the above points, I believe that while POSIX Queues are certainly quite useful for IPC and global memory sharing, it can't always replace mutexes and prevent priority inversion scenarios. With wise integration of various queue features (message priorities, timeouts, etc.), and system development – it is possible to ensure that the threat of priority inversion is completely removed; but it surely increases coding complexity, while also reducing flexibility.

Q5: 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.

References:

http://www.sat.dundee.ac.uk/psc/watchdog/watchdog-configure.html

http://0pointer.de/blog/projects/watchdog.html

https://superuser.com/questions/689017/can-systemd-detect-and-kill-hung-processes

https://stackoverflow.com/questions/9389777/how-to-detect-and-find-out-a-program-is-in-deadlock

https://linux.die.net/man/3/pthread_mutex_timedlock

Solution:

Code Description

- This code is modified/updated version of the code used in Q2 Method 3.
- It still uses mutex locks to protect shared global data structure, however, a timeout feature is introduced.
- To make it clearly observable, the writer sleeps for random number of seconds (within a range) – while holding the mutex, when reader is using a constant number of seconds to timeout the mutex lock API call.
- If the reader fails to acquire the lock within 10 seconds, it times out & indicates the same by printing out the message saying that no new data was made available, and loops back to start waiting again.
- If the reader manages to acquire the lock before 10 seconds have passed, it indicates that new data was made available, and reads the shared global data.
- It becomes clear from this example, that timed mutex locks are quite useful, to detect anomalies – and to warn against deadlocks.

Screenshots (Part 1)

Screenshots (Part 2)

```
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13000.863ms>Writer Waiting to Acquire Mutex - Iteration(1)
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13000.865ms>Reader Acquired Mutex ***GOT NEW DATA*** - Iteration(0)
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13000.888ms>Reader Got Accel X: -59.387001
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13000.900ms>Reader Got Accel Y: 72.539001
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13000.936ms>Reader Got Accel Z: -87.621002
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13000.946ms>Reader Got Roll: 60.653000
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13000.956ms>Reader Got Pitch: -62.301998
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13000.967ms>Reader Got Yaw: 120.214996
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13000.977ms>Reader Got Timestamp - Sec: 1562390499 Nano Sec: 534628055
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13000.996ms>Reader Completed Latest Read - Iteration(1)
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13001.052ms>Writer Started to Generate and Update Shared Structure - Iteration(1)
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13001.074ms>Writer Wrote Accel X: 2.040000
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13001.086ms>Writer Wrote Accel Y: -18.528000
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13001.098ms>Writer Wrote Accel Z: 12.218000
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13001.110ms>Writer Wrote Roll: -57.868000
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13001.122ms>Writer Wrote Pitch: -86.287003
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13001.133ms>Writer Wrote Yaw: 315.687012
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13001.145ms>Writer Wrote Timestamp - Sec: 1562390512 Nano Sec: 535129012
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13001.156ms>Selected Sleep Time for Writer is: 12 Seconds
Jul  5 23:21:52 poorn-desktop HW3_Q5[17726]: <13002.126ms>Reader Started Waiting for Mutex to be Unlocked - Iteration(1)
Jul  5 23:22:02 poorn-desktop HW3_Q5[17726]: <23002.291ms>Reader Mutex Timed Out ***NO NEW DATA*** - Iteration(1)
Jul  5 23:22:02 poorn-desktop HW3_Q5[17726]: <23002.344ms>Reader Completed Latest Read - Iteration(1)
Jul  5 23:22:02 poorn-desktop HW3_Q5[17726]: <23003.426ms>Reader Started Waiting for Mutex to be Unlocked - Iteration(1)
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.307ms>Writer Completed Writing New Data - Iteration(1)
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.359ms>Writer Waiting to Acquire Mutex - Iteration(2)
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.365ms>Reader Acquired Mutex ***GOT NEW DATA*** - Iteration(1)
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.389ms>Reader Got Accel X: 2.040000
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.400ms>Reader Got Accel Y: -18.528000
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.412ms>Reader Got Accel Z: 12.218000
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.424ms>Reader Got Roll: -57.868000
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.434ms>Reader Got Pitch: -86.287003
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.443ms>Reader Got Yaw: 315.687012
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.455ms>Reader Got Timestamp - Sec: 1562390512 Nano Sec: 535129012
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.471ms>Reader Completed Latest Read - Iteration(2)
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.521ms>Writer Started to Generate and Update Shared Structure - Iteration(2)
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.543ms>Writer Wrote Accel X: -57.688999
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.555ms>Writer Wrote Accel Y: -10.735000
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.566ms>Writer Wrote Accel Z: -64.552002
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.578ms>Writer Wrote Roll: 62.681000
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.590ms>Writer Wrote Pitch: 51.634998
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.600ms>Writer Wrote Yaw: 317.304993
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.611ms>Writer Wrote Timestamp - Sec: 1562390524 Nano Sec: 535596003
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25001.623ms>Selected Sleep Time for Writer is: 9 Seconds
Jul  5 23:22:04 poorn-desktop HW3_Q5[17726]: <25002.553ms>Reader Started Waiting for Mutex to be Unlocked - Iteration(2)
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.770ms>Writer Completed Writing New Data - Iteration(2)
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.820ms>Writer Exiting...
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.828ms>Reader Acquired Mutex ***GOT NEW DATA*** - Iteration(2)
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.855ms>Reader Got Accel X: -57.688999
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.867ms>Reader Got Accel Y: -10.735000
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.879ms>Reader Got Accel Z: -64.552002
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.891ms>Reader Got Roll: 62.681000
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.902ms>Reader Got Pitch: 51.634998
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.910ms>Reader Got Yaw: 317.304993
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.922ms>Reader Got Timestamp - Sec: 1562390524 Nano Sec: 535596003
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.934ms>Reader Completed Latest Read - Iteration(3)
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: <34001.945ms>Reader Exiting...
Jul  5 23:22:13 poorn-desktop HW3_Q5[17726]: >>>>>>>>>> Program End <<<<<<<<<<
poorn@poorn-desktop:/var/log$
```

Linux Watchdog Daemon for Deadlocked Processes

- There are a few ways which can be used, in order to automatically detect deadlock in the desired process, and then take corrective action. Mostly, it is to either restart the process or to restart the entire system.
- The basic idea for deadlock detection is to somehow keep a track of the process, and have the ability to distinguish between normal blocking and deadlock.
- The implementation of this varies somewhat among various methods. It could be in form of a bash script running in the background – which keep on testing process state code, or a daemon talking to the process and then reporting to system, or a daemon checking up on the process for deadlocks while also handling watchdog.
- For the concern of this question, there are primary two methods to detect and handle software's deadlock.
- First one includes using watchdog daemons, and their repair scripts. Second one includes using systemd which fully utilizes hardware watchdog timers, and exposes a software watchdog timer interface for the above-mentioned purpose.
- Both of these methods are quite similar. They include 'heartbeat' messages which are used to 'kick' or 'reset' the watchdog timer.
- As name suggests, this conveys a simple message – that the process is alive, and running as expected.
- If a heartbeat is not received, and consequently if the watchdog timer timeouts – it could mean one of three primary things: (1) The process has been killed unexpectedly, and not in a clean way due to some exception. (2) The process is blocked unusually for some reason, and is taking longer than nominal time but is not in deadlock. (3) The process is in indefinite deadlock.
- There are ways to check for the first reason if needed, and since the solution for both (1) and (3) is largely to simply restart the process, it mostly doesn't matter to differentiate between them. Moreover, by design – no process is written in such a way to have deadlocks, and thus, the deadlock is one of the exceptions – only difference is that it didn't kill the process.
- If the watchdog timer is set up properly, and if the code is tested enough, then the probability of reason (2) can be made quite low.
- Once it has been detected that the watchdog timer has expired, the corrective action can be taken. Mostly, it is to just restart the process, probably after performing some cleanup operations.
- This can either be done through the use of repair scripts included in the watchdog daemon, or directly by systemd – depending on the kernel version and method being utilized.
- This provides quite a robust way to detect indefinite deadlocks caused by software, and to resolve them.