

## Homework-4

Satya Mehta &amp; Smitha Bhaskar

Codes executed on Raspberry PI

[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 Dr. Siewert's 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?

*3 Main Key Points in the paper "Priority Inheritance Protocols: An Approach to Real-Time Synchronization": -*

In the paper there are 3 main key points or theorems defined which solve the problem of the priority inversion and deadlock. They are defined as protocols and a schedulable feasibility test is performed which derives the necessary and sufficient condition. **Priority inversion** is the phenomenon where the higher priority job is preempted by the lower priority. In the paper the priority inversion problem has been solved using two protocols which have been further proved using feasibility tests. In priority inversion due to blocking of the higher priority job there are a lot of chances of missing deadline even when the resource utilization is too less.

1) **Priority Inheritance Protocol**: - To rectify the problem of priority inversion problem one of the approaches mentioned in the paper is Priority Inheritance protocol. There were some assumptions made before implementing the protocol. 1). Jobs do not suspend themselves. 2). Critical sections are clearly nested, and all the jobs complete the section and release the lock. 3). Suppose there are jobs J1, J2, J3 then the J1 has highest priority and J3 has lowest. 4). Highest priority job will run first, and same priority jobs will be served as FCFS. In this protocol the priorities of the job are inherited from the one which is blocked to the one which is executing. For example, if a job J blocks one or more higher priority job then it converts itself to higher priority when executing critical section. After exiting the critical section, it returns back to its original priority assessment. Priority inheritance is therefore **transitive**.

There are a couple of scenarios defined related to blocking. Direct blocking and push through blocking. Direct blocking, where a higher priority job tries to lock a locked semaphore. Push through block, where a medium priority job is blocked by the low priority job which inherits the priorities of the higher priority job.

2). **Priority Ceiling Protocol**: - The problem of deadlocks and chain blockings can be resolved by using this protocol. The basic idea of this protocol is if a job preempts all the other jobs and enters its critical section, then it must be a guarantee that the priority is inherited from the blocked job according to the priority inheritance protocol. If the condition is not satisfied, then the current job which enters into the critical section must be blocked by the system. This system is implemented by assigning a **priority semaphore** which is equal to the highest priority task that may use this semaphore. When the higher priority job is ready to run and tries to lock the semaphore S. It may be blocked if its priority is not higher than the priority ceiling of the semaphore S. The paper illustrates by giving an example which assumes two jobs with equal priority ceiling of semaphore and another job which with different priority ceiling of semaphore.

3). **Schedulability Analysis**: -

A sufficient condition is developed for the set of periodic tasks using the priority inversion protocol. The priority is assigned based on the Rate Monotonic Algorithm. Some assumptions are made. 1). Tasks are periodic. 2). Each task has deterministic execution time for both critical and noncritical sections. 3). Priorities based on the RMA. There are theorems derived using RMA and considering the time which causes the blocking of the higher priority jobs due to

priority inheritance protocol. The worst-case time is considered depending on the highest blocking time of all the tasks in the set.

The analysis states that the tasks can be scheduled by rate monotonic algorithm if below condition is satisfied.

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

The tasks if executed for  $C_i$  units of time can be delayed by  $B_i$  units of time and still meets its deadline. The  $C_i$  in the above equation can be replaced by  $C_i + B_i$  where  $B$  is worst case blocking time.

#### *Summary of Linux Torvalds as described by Jonathan Corbet and Ingo Molnar and Thomas Gleixner :*

Linus clearly states his position as not to use Priority Inheritance at all and achieve using lockless designs or carefully thought-out locking scenarios. He argues that using Priority Inheritance (PI) tends to complicate and slow down the locking code and it does not solve priority inheritance problem completely and might lead to system failure. Linus's position makes sense in not using PI in kernel space because of the heavy overhead due to multiple System calls to the kernel, which will lead to degraded performances eventually.

Ingo states that locking is inevitable and cannot be ignored. (Even in the case of Mars Pathfinder) Achieving completely lockless solutions is very hard, and the complexity of lockless algorithms often endangers the ability to do robust reviews of said code. Ingo introduces user-space PI which helps in achieving/improving determinism for user-space applications. In the best-case, it can help achieve determinism and well-bound latencies. Even in the worst-case, PI will improve the statistical distribution of locking related application delays.

#### *Our Opinion: -*

When there is blocking (slowpath), instead, the FUTEX\_LOCK\_PI is called. The requesting process is put into a special queue, and, if necessary, that process lends its priority to the process holding the contended futex. The priority inheritance is chained, so that, if the holding process is blocked on a second futex, the boosted priority will propagate to the holder of that second futex. As soon as a futex is released, any associated priority boost is removed.

As with regular futexes, the kernel only needs to know about a PI-futex while it is being contended. So, the number of futexes in the system can become quite large without serious overhead on the kernel side. Hence using Priority Inheritance futexes does not cause the same problem caused by other synchronization primitives like Mutexes which are completely implemented in kernel space, while solving the problem of unbounded priority inheritance.

#### *Unbounded priority inversion:*

Futex consists of a kernel-space wait queue which is connected to a aligned int in user-space. Therefore, a futex would not use a system call until the lock is contented. In a regular futex, a lock is acquired without any system call (completely in user space) if the resource was not locked before. If the resource is locked, then a WAIT call puts the thread to sleep and adds it in the wait queue. But in PI futexes, if the priority of the task blocking the current task is lower, than the task holding the lock is boosted to the priority of the higher priority task. This boost is transitive, i.e. if the process holding the futex is blocked on another futex, the boosted priority will be inherited by the second futex and would resume original priorities as soon as it is done executing.

Although, PI Futexes solve the problem of unbounded priority inheritance, they would still not be able to solve problems of deadlocks and chained blocking, which could be solved by Priority Ceiling Protocol which the PI Futex does not implement. But since they offer much less overhead and better performance than other kernel space synchronization primitives, they are a viable technique to solve unbounded priority inheritance problem.

2) [25 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](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).

In a multithreaded application, the application is required to be thread safe which means that all access to the shared state are controlled and guarded by using some mechanisms like mutual exclusions or semaphores. But there can be some problems in the thread safe functions which are deadlocks, race conditions, etc. Reentrancy is a term which can be defined as a function which can be entered again and again without data corruption. The scope of the variables initialized inside a reentrant function are local. Hence, a thread safe function which are reentrant guarantees expected operation without any concurrency bugs and data corruption.

*Pure functions that use only stack and have no global memory.*

In this method the functions use local variables rather than global variables. Hence, avoiding race conditions. Each function uses its own stack and hence other function running cannot interfere in the operation of this function. This ensures thread safety and reentrancy.

Implementation: - I'll create functions and call them according to my application. While calling I'll **share the data between functions by using arguments**. The advantage of using this method is that this will ensure complete safety as each function have their own local variables to operate avoiding interference to another function. Disadvantage is that we will require large amount of memory and there will be copies of the data which are passed from one function to another.

**Impact on Real-Time Systems:** - There won't be much effect in the real-time application as threads are running independently.

*Functions which use thread indexed global data.*

There are couple of implementations of the thread indexed global data. One of them is using a one index which is a global variable. The threads have access to this variable and it indexes it. Hence, other threads which are not indexed cannot use that global variable, Hence, avoiding the concurrency bug.

Another method which can be implemented by using one global data as index for a thread. A global array can be used where each index of the array is assigned to a thread. The thread can write a data into its indexed space, but it can only read other memory which corresponds to other threads. We should know in previous that how many threads are there and how many of them will use global data or share data between them. There are chances of memory leak if we assign global data and indexing without having prior knowledge of the number of threads in the application. This implementation is also known as **Thread Local Storage**. Using a global index, unique data can be provided to each thread. To access the data associated with the index the address is retrieved from the global index and stored in the local variable.

Implementation: - Considering above two methods, I'll create a global array which is assigned virtually to the threads. I'll code each thread in such a way where I'll only write to its indexed location, but I'll read other indexes in an array. Hence, the data can be shared by using a global array. Also, I'll make another global variable(flag) which is raised (according to index of the thread) when thread uses a global data and clear it when the thread has completed accessing the global data.

**Impact on real-time systems:** - There might be large effect on the real-time systems as some thread might be blocked or kept in the queue if one thread is accessing global data. This kind of application is suitable for Soft Real-Time systems. Since, there is no direct sharing of data the data corruption is very low.

*Functions which use global data but synchronize access using mutex semaphore.*

The threads in this method runs independently and the critical section is blocked using mutex. The mutex is locked using thread and it is required that the same thread unlocks it. The other thread waits on the lock to be unlocked to begin execution.

**Implementation:** - Implementing mutexes is quite simple and it guarantees safety of the critical section. Mutex are a part of POSIX standards and can be initialized using a api called as pthread\_mutex\_init(). The mutex uses a mutex variable which can be accessed by individual threads to lock and unlock using pthread\_mutex\_lock() and pthread\_mutex\_unlock(). During the access of the global variables or critical sections I'll use this api's to block other threads to use the same memory. This will ensure safety and also allow us to use the global variables efficiently avoiding data corruption.

**Impact on real-time system:** - There are many disadvantages of using mutexes. Priority inversion can cause drastic effect on real-time systems if it's not using mutex semaphores. Mutex misuse can cause deadlocks. It can also lead to high latency if a thread has locked mutex for large amount of time and other thread is waiting for the same mutex to be unlocked. In short, it is required to have a prior knowledge of using mutexes efficiently otherwise there can be many problems in real-time application.

Screenshot of the code that was executed:

```
*****Data Written*****
Data written : X-AXIS 0.203851, Y-AXIS 4.198582, Z-AXIS 4.364031
Roll 2.883304, Pitch 4.320591, Yaw 4.175524
*****Data Read*****
Data written : X-AXIS 0.203851, Y-AXIS 4.198582, Z-AXIS 4.364031
Roll 2.883304, Pitch 4.320591, Yaw 4.175524
Thread 2 timestamp : 230666973

*****Data Written*****
Data written : X-AXIS 2.555390, Y-AXIS 3.973638, Z-AXIS 4.611773
Roll 4.212016, Pitch 4.378853, Yaw 1.539620
*****Data Read*****
Data written : X-AXIS 2.555390, Y-AXIS 3.973638, Z-AXIS 4.611773
Roll 4.212016, Pitch 4.378853, Yaw 1.539620
Thread 2 timestamp : 230562091

*****Data Written*****
Data written : X-AXIS 3.266041, Y-AXIS 0.926174, Z-AXIS 2.262779
Roll 0.976237, Pitch 3.292137, Yaw 4.849173
*****Data Read*****
Data written : X-AXIS 3.266041, Y-AXIS 0.926174, Z-AXIS 2.262779
Roll 0.976237, Pitch 3.292137, Yaw 4.849173
Thread 2 timestamp : 230552012
```

3) [15 points] Download <http://mercury.pr.erau.edu/~siewerts/cec450/code/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?

*Description of deadlocks and priority inversion code:*

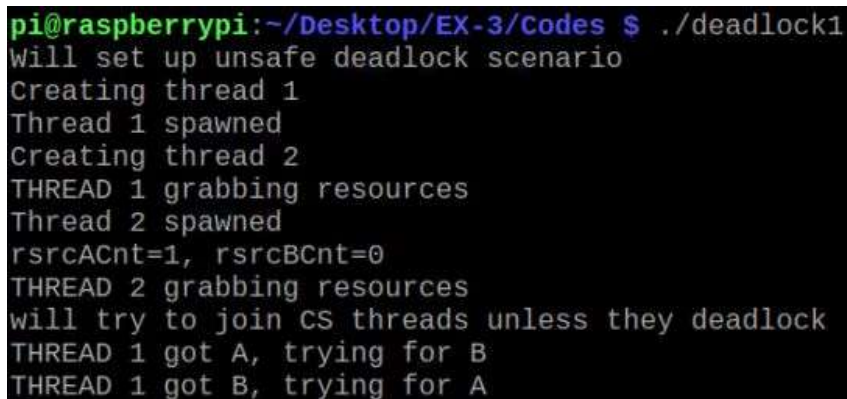
The deadlock code provided cannot make progress. To fix this problem, we modified the grabRsrcs function to use the random backoff solution. This can easily be accomplished with the pthread\_mutex\_timedlock function in the pthread library. However, there is one more issue. If these releases happen at the same time, then the deadlock will turn into what is known as livelock - The two threads will continuously acquire their first resources, recognize deadlock via timeout, and release resources. The threads will not be able to make progress in this scheme either. To solve both the problems of livelock and deadlock, we set the mutex timeout to a random value.

Priority inversion, mentioned earlier, occurs in the following situation: There are three levels of task priority - high, medium and low. The high priority and low priority threads share a resource. The medium priority task does not share the resource or mutex.

A low, medium, and high priority thread are all started, with the low and high priority threads sharing a resource. The low priority thread holds a mutex on the shared data, forcing the high priority data to block while the low priority thread holds the resource.

The pthread3ok code solves the problem of priority inversion by boosting the priority of the low priority thread to above the priority of the medium priority thread. This prevents the medium priority thread from being able to interfere with the low priority thread, thus preventing the unbounded priority inversion problem.

Screenshot of code without correcting deadlock:



```
pi@raspberrypi:~/Desktop/EX-3/Codes $ ./deadlock1
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
THREAD 2 grabbing resources
will try to join CS threads unless they deadlock
THREAD 1 got A, trying for B
THREAD 1 got B, trying for A
```



Screenshot of Code upon correcting deadlock:

```
pi@raspberrypi:~/Desktop/EX-3/Codes $ ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 1 getting B with timed lock 290383
THREAD 2 getting A with timed lock 931886
THREAD 2 could not get A, released BTHREAD 1 could not get B, released A
THREAD 1 got A, trying for B
THREAD 2 got B, trying for A
THREAD 1 getting B with timed lock 693777
THREAD 1 could not get B, released A
THREAD 1 got A, trying for B
THREAD 2 getting A with timed lock 637915
THREAD 2 could not get A, released BTHREAD 2 got B, trying for A
THREAD 1 getting B with timed lock 748702
```

Screenshot of code giving the parameter as safe

```
pi@raspberrypi:~/Desktop/EX-3/Codes $ sudo ./deadlock1 safe
Creating thread 1
Thread 1 spawned
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: 1994331248 done
Creating thread 2
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: 1994331248 done
All done
pi@raspberrypi:~/Desktop/EX-3/Codes $
```

Screenshot of code giving the parameter as race

```
pi@raspberrypi:~/Desktop/EX-3/Codes $ sudo ./deadlock race
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
Thread 2 spawned
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 getting A with timed lock 290383
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 1 got A, trying for B
THREAD 1 getting B with timed lock 931886
THREAD 1 got A and B
THREAD 2 could not get A, released BTHREAD 1 done
THREAD 2 got B, trying for A
THREAD 2 getting A with timed lock 693777
THREAD 2 got B and A
THREAD 2 done
Thread 1: 1993962608 done
Thread 2: 1985574000 done
All done
pi@raspberrypi:~/Desktop/EX-3/Codes $
```

Screenshot of executing pthread3 code

```
pi@raspberrypi:~/Desktop/EX-3/Codes $ sudo ./pthread3 5
interference time = 5 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 3
Low prio 3 thread spawned at 1552092143 sec, 759104576 nsec
Creating thread 2
Middle prio 2 thread spawned at 1552092144 sec, 759362654 nsec
Creating thread 1, CSnt=1
**** 2 idle NO SEM stopping at 1552092144 sec, 759385467 nsec
High prio 1 thread spawned at 1552092144 sec, 759434321 nsec
**** 3 idle stopping at 1552092145 sec, 759237141 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1552092147 sec, 759371115 nsec
HIGH PRIO done
START SERVICE done
All done
pi@raspberrypi:~/Desktop/EX-3/Codes $
```

### Screenshot of executing pthread3ok code

```
pi@raspberrypi:~/Desktop/EX-3/Codes $ sudo ./pthread3ok 5
interference time = 5 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 1
High prio 1 thread spawned at 1552092173 sec, 989985286 nsec
Creating thread 2
**** 1 idle stopping at 1552092173 sec, 990006276 nsec
Middle prio 2 thread spawned at 1552092173 sec, 990123723 nsec
Creating thread 3
**** 2 idle stopping at 1552092173 sec, 990162369 nsec
Low prio 3 thread spawned at 1552092173 sec, 990249296 nsec
**** 3 idle stopping at 1552092173 sec, 990284921 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
START SERVICE done
All done
```

#### *Description of RT\_PREEMPT\_PATCH:*

This patch allows you to set priorities for any tasks, including interrupts and system calls. It also removes all unbounded latencies from the Linux kernel. Therefore, we can solve the problem of unbounded priority inversion through the normal methods of priority inheritance or a priority ceiling. We can boost the priority of the low priority thread to be above that of the medium priority thread, even if the medium priority thread is caused by a system call or an interrupt.

The RT\_PREEMPT team is small, and the Linux Kernel is huge. It is likely that there are some locations where system calls can still have unbounded latencies. In rare and truly unexpected cases, unbounded priority inversion might still be possible, which at the very least could miss deadlines or turn a theoretically hard real time system into a soft real time system. For mission critical applications, where life or other important goals are of the utmost importance, it would be better to use a dedicated RTOS instead of the RT\_PREEMPT patch.

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 Altera DE1-SoC or 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?

In the POSIX message queue, we created an initialized two threads for the receiver and the sender, respectively. The receiver thread has higher priority than the sender. The message queue attributes are set to set information such as



mq\_maxmsg, mq\_msgsize, and mq\_flags. The threads are then created and spawned. Because the receiver thread has higher priority, it gets scheduled first. In this thread, the message queue is opened with mq\_open with the shared message queue at '/send\_recieve\_mq\_posix'.

In the heap implementation of the message queue, we first initialize a buffer of size 4096 bytes called imagebuff. We created an initialized two threads for the receiver and the sender, respectively. The receiver thread has higher priority than the sender. The message queue attributes are set to set information such as mq\_maxmsg, mq\_msgsize, and mq\_flags. In this implementation, the shared message queue is opened using the mq\_open function in the main function. The message queue is opened with mq\_open with the shared message queue at '/send\_recieve\_mq\_heap'.

*Similarities:*

The two-message queue applications `heap_mq.c` and `posix_mq.c` are similar in their core functionality, but differ in a few key ways. They are similar in that they both implement a messaging queue and have the same basic structure. If you look toward the bottom of either file, you find `mq_demo` and `heap_mq` for `posix_mq` and `heap_mq` respectively. These are the main functions that start the message queue demos, and they basically do the same thing – They each start sender and receiver threads, which exchange messages from sender to receiver in a queue.

*Differences:*

The main difference between the implementations is how the messages get passed under the hood. The `posix_mq` implementation uses the POSIX `mq_open` function to pass messages along a POSIX queue. The `heap_mq` file, however, accomplishes the same result by using the heap directly, passing along heap pointers to keep track of messages. The heap implementation allows us to pass larger amounts of data without actually passing the entire message, this can be done by passing the pointer to a buffer already allocated in the heap. This way is more efficient and has less overhead, as we can see we just needed to pass an 8 byte pointer to the heap through the sender to the queue and retrieved the entire message which was 4096 bytes.

### Screenshot of Heap message queue

```
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~\n
Send: message ptr 0xABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~\n successfully sent
Receive: ptr msg 0x0x76400470 received with priority = 30, length = 8, id = 999
Contents Received = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~\n
Heap space memory freed
Reading 4 bytes
```

### Screenshot of posix message queue

```
pi@raspberrypi:~/Desktop/EX-3/Codes $ sudo ./posixmq
Receiver Thread Created Successfully!
Receiver
Receiving Message from Message Queue
Sender Thread Created Successfully!
Sender
Sending Message from Message Queue
Received: msg this is a test, and only a test, in the event of a real emergency, you would be instructed ...
Received with priority = 30, length = 95
Send: Message successfully sent
pi@raspberrypi:~/Desktop/EX-3/Codes $
```

*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?*

Message queues can avoid the problem of MUTEX priority inversion. Certain RTOS's such as QNX and even VxWorks have priority inheritance enabled message queues. When a thread receives a message, they it inherits the priority of the sending thread. This message-driven priority inheritance avoids priority-inversion problems. Therefore, if a thread with medium priority tries to execute, the receiving thread will block it and continue to execute since it has higher priority. Also, if a message queue is already full and a new message is queued then, it would be blocked until the message queue has space.

The POSIX message queues also have a message priority associated with them, so messages with higher priority would get dequeued before messages with lower priority i.e. they can preempt the message with low priority. Therefore, we can say message queues solve the problem of global memory sharing.

5) [35 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 unblocks 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.

#### *WATCHDOG: -*

Watchdog timers are system timers which are used in situations of system failure. It is also known as computer operating properly timer. During normal operation the system regularly feeds the watchdog timer. In some cases, where the system is in deadlock condition or any failure it would not be able to feed the watchdog timer. Hence, the timer after some interval of time will automatically reset the system.

#### *In case of deadlock: -*

In such cases the system will be in a deadlock condition where it will require some resources to begin resumption of the correct operation. All the resources will be blocked, and system processes would not be able to feed the timer. Hence, when the watchdog timer resets, it will wait for the system to reset it and as the system would not be able to do so, it will reset the whole system. This is how the watchdog timers can be used to come out of the deadlock.

Linux provides its own set of watchdog timers. Generally, the operating systems have other ways to come out of the situation where a task get freeze. There is interface provided in the linux by the corresponding device drivers. It also has the software module named Softdog but it is not much stable. The linux watchdog daemon is configured to periodic test to look that the system is running ok. On failing some of this test, the daemon reboots the machine. The linux watchdog daemon has moderately orderly shut down process where there is sequence of shutdowns of various processes using signals. After 5 seconds there is a SIGKILL signal applied which is not ignorable and kills all the processes. After this process it saves the states of other peripherals and unmount the rootfs and resets the system using hardware timer. When the system is booted wd\_keepalive is started which takes care of the faults during the boot process. The normal watchdog is started later as it might require some of the resources which are initialized after the system is fully booted.

#### Screenshot of executing the code

```
*****Data Written*****
Data written : X-AXIS 3.186045, Y-AXIS 3.567795, Z-AXIS 1.446649
Roll 1.324767, Pitch 1.372660, Yaw 1.927086
NO new data available at 1552090834
*****Data Written*****
Data written : X-AXIS 2.420577, Y-AXIS 4.490315, Z-AXIS 2.477287
Roll 3.116964, Pitch 2.689977, Yaw 0.746833
NO new data available at 1552090844
*****Data Written*****
Data written : X-AXIS 3.695334, Y-AXIS 4.287394, Z-AXIS 3.609101
Roll 0.156047, Pitch 2.818936, Yaw 4.097259
NO new data available at 1552090854
*****Data Written*****
Data written : X-AXIS 0.984454, Y-AXIS 1.390543, Z-AXIS 4.959857
Roll 4.504945, Pitch 4.560130, Yaw 1.301195
NO new data available at 1552090864
```

#### REFERENCES:

[http://mercury.pr.erau.edu/~siewerts/cec450/documents/Papers/prio\\_inheritance\\_protocols.pdf](http://mercury.pr.erau.edu/~siewerts/cec450/documents/Papers/prio_inheritance_protocols.pdf) (For Q1)

<http://www.sat.dundee.ac.uk/psc/watchdog/watchdog-background.html> (Q5)

[https://rt.wiki.kernel.org/index.php/RT\\_PREEMPT\\_HOWTO](https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO)

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

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

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

<https://linux.die.net/man/>

<http://mercury.pr.erau.edu/~siewerts/cec450/code/>

<https://docs.microsoft.com/en-us/windows/desktop/procthread/thread-local-storage/>